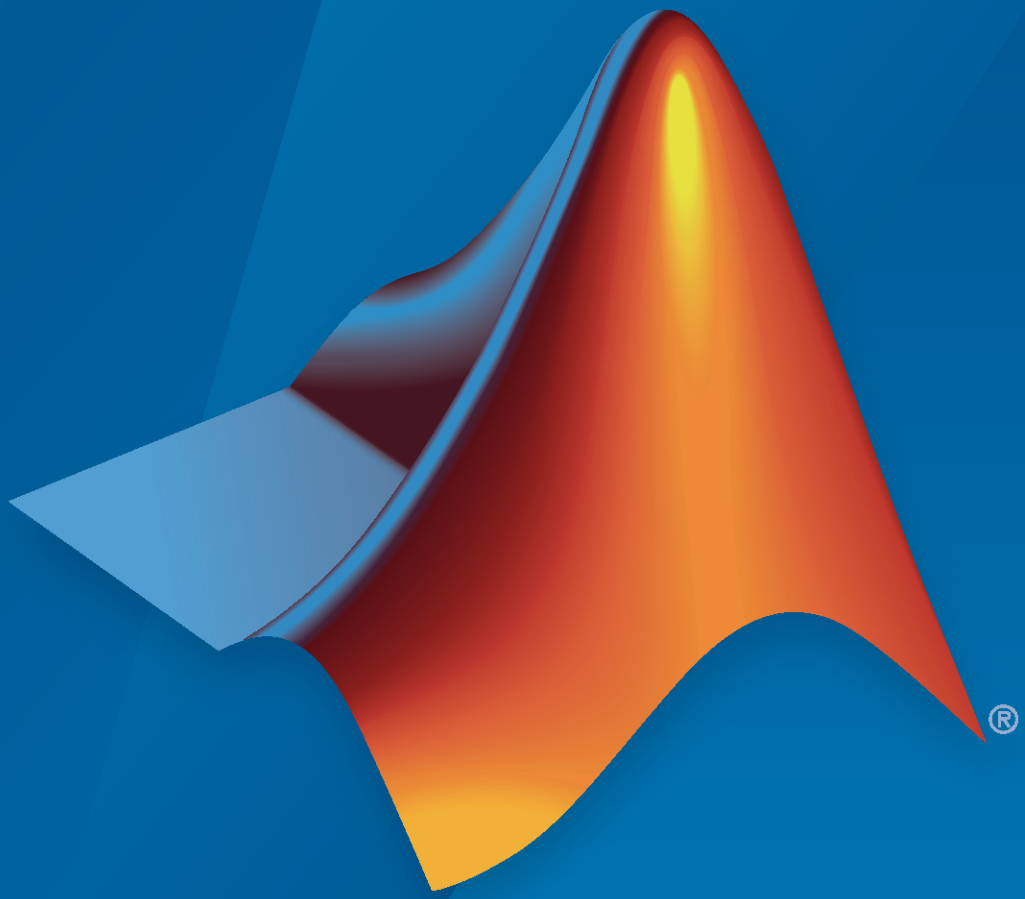
Navigation Toolbox™

Reference



MATLAB® & SIMULINK®

R2021b

MathWorks®

How to Contact MathWorks

Latest news: `www.mathworks.com`

Sales and services: `www.mathworks.com/sales_and_services`

User community: `www.mathworks.com/matlabcentral`

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

Phone: 508-647-7000

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

Navigation Toolbox™ Reference

Revision History

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

Contents

Functions

allanvar

Allan variance

Syntax

```
[avar,tau] = allanvar(Omega)
[avar,tau] = allanvar(Omega,m)
[avar,tau] = allanvar(Omega,ptStr)
[avar,tau] = allanvar(___,fs)
```

Description

Allan variance is used to measure the frequency stability of oscillation for a sequence of data in the time domain. It can also be used to determine the intrinsic noise in a system as a function of the averaging time. The averaging time series τ can be specified as $\tau = m/fs$. Here fs is the sampling frequency of data, and m is a list of ascending averaging factors (such as 1, 2, 4, 8, ...).

`[avar,tau] = allanvar(Omega)` returns the Allan variance `avar` as a function of averaging time `tau`. The default averaging time `tau` is an octave sequence given as $(1, 2, ..., 2^{\text{floor}\{\log_2[(N-1)/2]\}})$, where N is the number of samples in `Omega`. If `Omega` is specified as a matrix, `allanvar` operates over the columns of `omega`.

`[avar,tau] = allanvar(Omega,m)` returns the Allan variance `avar` for specific values of `tau` defined by `m`. Since the default frequency `fs` is assumed to be 1, the output `tau` is exactly same with `m`.

`[avar,tau] = allanvar(Omega,ptStr)` sets averaging factor `m` to the specified point specification, `ptStr`. Since the default frequency `fs` is 1, the output `tau` is exactly equal to the specified `m`. `ptStr` can be specified as `'octave'` or `'decade'`.

`[avar,tau] = allanvar(___,fs)` also allows you to provide the sampling frequency `fs` of the input data `omega` in Hz. This input parameter can be used with any of the previous syntaxes.

Examples

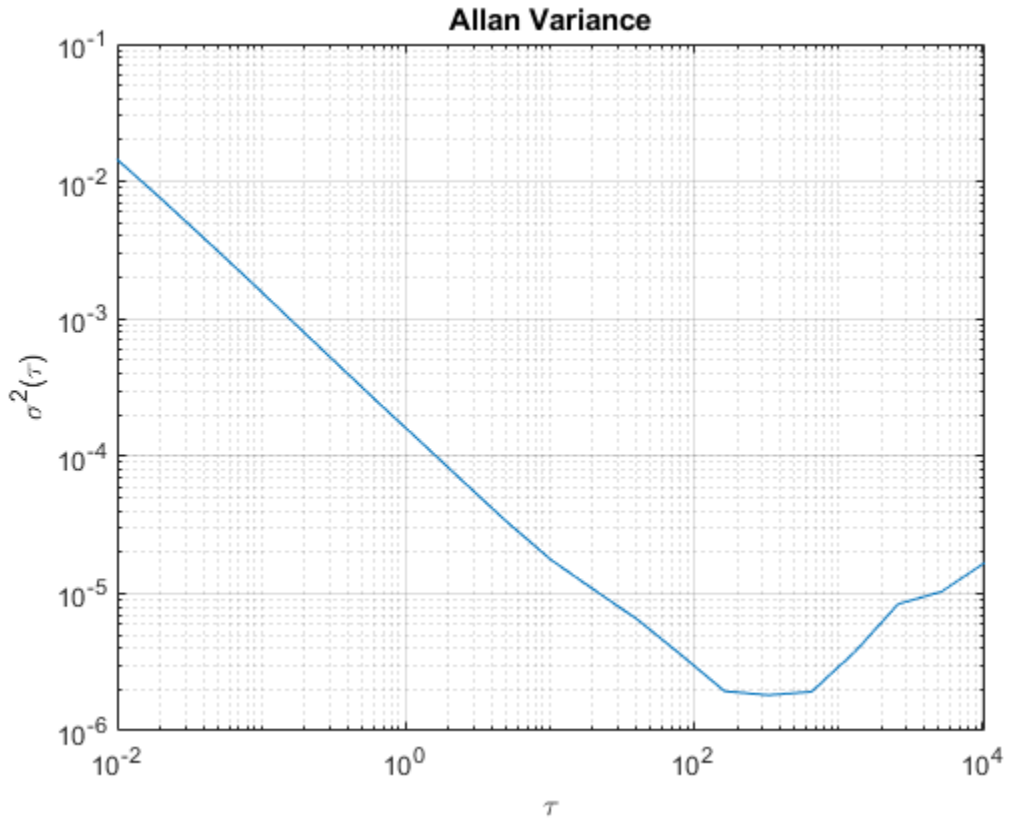
Determine Allan Variance of Single Axis Gyroscope

Load gyroscope data from a MAT file, including the sample rate of the data in Hz. Calculate the Allan variance.

```
load('LoggedSingleAxisGyroscope','omega','Fs')
[avar,tau] = allanvar(omega,'octave',Fs);
```

Plot the Allan variance on a `loglog` plot.

```
loglog(tau,avar)
xlabel('\tau')
ylabel('\sigma^2(\tau)')
title('Allan Variance')
grid on
```



Determine Allan Deviation at Specific Values of τ

Generate sample gyroscope noise, including angle random walk and rate random walk.

```
numSamples = 1e6;
Fs = 100;
nStd = 1e-3;
kStd = 1e-7;
nNoise = nStd.*randn(numSamples,1);
kNoise = kStd.*cumsum(randn(numSamples,1));
omega = nNoise+kNoise;
```

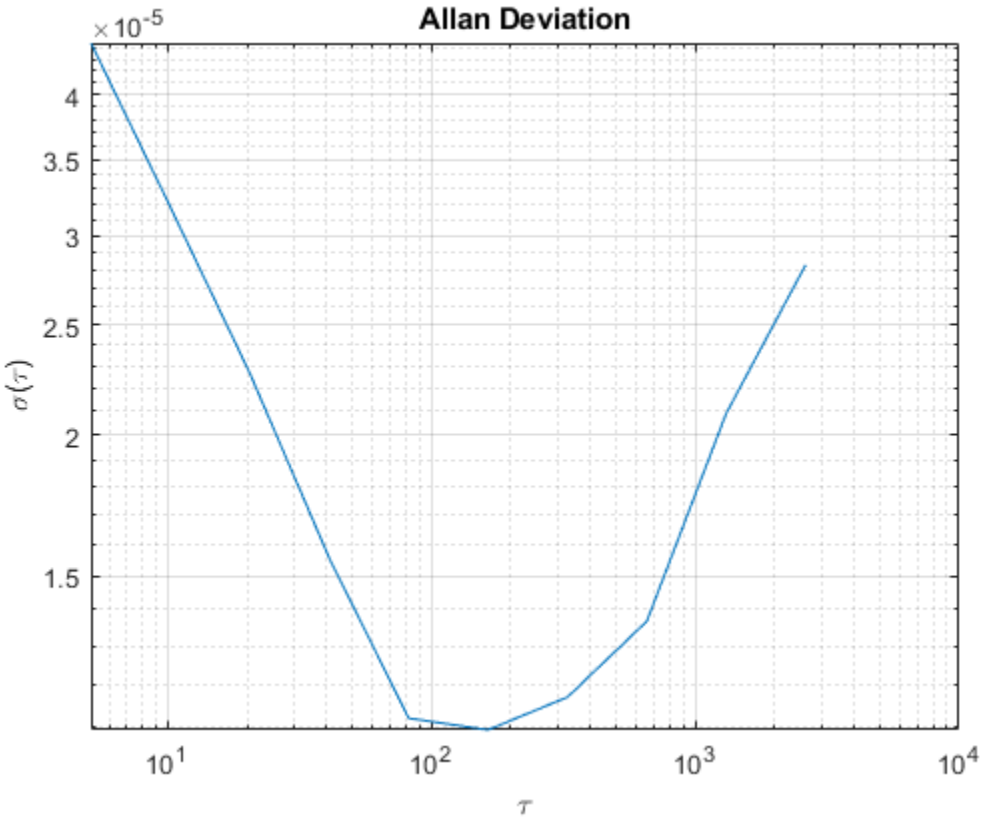
Calculate the Allan deviation at specific values of $m = \tau$. The Allan deviation is the square root of the Allan variance.

```
m = 2.^(9:18);
[avar,tau] = allanvar(omega,m,Fs);
adev = sqrt(avar);
```

Plot the Allan deviation on a `loglog` plot.

```
loglog(tau,adev)
xlabel('\tau')
ylabel('\sigma(\tau)')
```

```
title('Allan Deviation')
grid on
```



Input Arguments

`Omega` — Input data

N-by-1 vector | *N*-by-*M* matrix

Input data specified as an *N*-by-1 vector or an *N*-by-*M* matrix. *N* is the number of samples, and *M* is the number of sample sets. If specified as a matrix, `allanvar` operates over the columns of `Omega`.

Data Types: `single` | `double`

`m` — Averaging factor

scalar | vector

Averaging factor, specified as a scalar or vector with ascending integer values less than (*N*-1)/2, where *N* is the number of samples in `Omega`.

Data Types: `single` | `double`

`ptStr` — Point specification of `m`

`'octave'` (default) | `'decade'`

Point specification of `m`, specified as `'octave'` or `'decade'`. Based on the value of `ptStr`, `m` is specified as following:

- If `ptStr` is specified as `'octave'`, `m` is:

$$\left[2^0, 2^1 \ldots 2^{\left\lfloor \log_2\left(\frac{N-1}{2}\right)\right\rfloor}\right]$$

- If `ptStr` is specified as `'decade'`, `m` is:

$$\left[10^0, 10^1 \ldots 10^{\left\lfloor \log_{10}\left(\frac{N-1}{2}\right)\right\rfloor}\right]$$

N is the number of samples in `Omega`.

`fs` — Basic frequency of input data in Hz
scalar

Basic frequency of the input data, `Omega`, in Hz, specified as a positive scalar.

Data Types: `single` | `double`

Output Arguments

`avar` — Allan variance of input data
vector | matrix

Allan variance of input data at `tau`, returned as a vector or matrix.

`tau` — Averaging time of Allan variance
vector | matrix

Averaging time of Allan variance, returned as a vector, or a matrix.

See Also

`gyroparams` | `imuSensor`

Introduced in R2019a

angdiff

Difference between two angles

Syntax

```
delta = angdiff(alpha,beta)
```

```
delta = angdiff(alpha)
```

Description

`delta = angdiff(alpha,beta)` calculates the difference between the angles `alpha` and `beta`. This function subtracts `alpha` from `beta` with the result wrapped on the interval `[-pi,pi]`. You can specify the input angles as single values or as arrays of angles that have the same number of values.

`delta = angdiff(alpha)` returns the angular difference between adjacent elements of `alpha` along the first dimension whose size does not equal 1. If `alpha` is a vector of length *n*, the first entry is subtracted from the second, the second from the third, etc. The output, `delta`, is a vector of length *n-1*. If `alpha` is an *m*-by-*n* matrix with *m* greater than 1, the output, `delta`, will be a matrix of size *m-1*-by-*n*. If `alpha` is a scalar, `delta` returns as an empty vector.

Examples

Calculate Difference Between Two Angles

```
d = angdiff(pi,2*pi)
```

```
d = 3.1416
```

Calculate Difference Between Two Angle Arrays

```
d = angdiff([pi/2 3*pi/4 0],[pi pi/2 -pi])
```

```
d = 1×3

    1.5708   -0.7854   -3.1416
```

Calculate Angle Differences of Adjacent Elements

```
angles = [pi pi/2 pi/4 pi/2];
d = angdiff(angles)
```

```
d = 1×3
```

```
   -1.5708   -0.7854    0.7854
```

Input Arguments

`alpha` — Angle in radians

scalar | vector | matrix | multidimensional array

Angle in radians, specified as a scalar, vector, matrix, or multidimensional array. This is the angle that is subtracted from `beta` when specified. If `alpha` is a scalar, `delta` returns as an empty vector.

Example: `pi/2`

`beta` — Angle in radians

scalar | vector | matrix | multidimensional array

Angle in radians, specified as a scalar, vector, matrix, or multidimensional array of the same size as `alpha`. This is the angle that `alpha` is subtracted from when specified.

Example: `pi/2`

Output Arguments

`delta` — Difference between two angles

scalar | vector | matrix | multidimensional array

Angular difference between two angles, returned as a scalar, vector, or array. `delta` is wrapped to the interval `[-pi,pi]`. If `alpha` is a scalar, `delta` returns as an empty vector.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Introduced in R2015a

axang2quat

Convert axis-angle rotation to quaternion

Syntax

```
quat = axang2quat(axang)
```

Description

`quat = axang2quat(axang)` converts a rotation given in axis-angle form, `axang`, to quaternion, `quat`.

Examples

Convert Axis-Angle Rotation to Quaternion

```
axang = [1 0 0 pi/2];
quat = axang2quat(axang)
```

```
quat = 1×4

    0.7071    0.7071         0         0
```

Input Arguments

axang — Rotation given in axis-angle form

n-by-4 matrix

Rotation given in axis-angle form, specified as an *n*-by-4 matrix of *n* axis-angle rotations. The first three elements of every row specify the rotation axis, and the last element defines the rotation angle (in radians).

Example: `[1 0 0 pi/2]`

Output Arguments

quat — Unit quaternion

n-by-4 matrix

Unit quaternion, returned as an *n*-by-4 matrix containing *n* quaternions. Each quaternion, one per row, is of the form $q = [w\ x\ y\ z]$, with w as the scalar number.

Example: `[0.7071 0.7071 0 0]`

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`quat2axang`

Introduced in R2015a

axang2rotm

Convert axis-angle rotation to rotation matrix

Syntax

```
rotm = axang2rotm(axang)
```

Description

`rotm = axang2rotm(axang)` converts a rotation given in axis-angle form, `axang`, to an orthonormal rotation matrix, `rotm`. When using the rotation matrix, premultiply it with the coordinates to be rotated (as opposed to postmultiplying).

Examples

Convert Axis-Angle Rotation to Rotation Matrix

```
axang = [0 1 0 pi/2];
rotm = axang2rotm(axang)
```

```
rotm = 3×3

    0.0000         0    1.0000
         0    1.0000         0
   -1.0000         0    0.0000
```

Input Arguments

axang — Rotation given in axis-angle form

n-by-4 matrix

Rotation given in axis-angle form, specified as an *n*-by-4 matrix of *n* axis-angle rotations. The first three elements of every row specify the rotation axis, and the last element defines the rotation angle (in radians).

Example: `[1 0 0 pi/2]`

Output Arguments

rotm — Rotation matrix

3-by-3-by-*n* matrix

Rotation matrix, returned as a 3-by-3-by-*n* matrix containing *n* rotation matrices. Each rotation matrix has a size of 3-by-3 and is orthonormal. When using the rotation matrix, premultiply it with the coordinates to be rotated (as opposed to postmultiplying).

Example: `[0 0 1; 0 1 0; -1 0 0]`

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`rotm2axang`

Introduced in R2015a

axang2tform

Convert axis-angle rotation to homogeneous transformation

Syntax

```
tform = axang2tform(axang)
```

Description

`tform = axang2tform(axang)` converts a rotation given in axis-angle form, `axang`, to a homogeneous transformation matrix, `tform`. When using the transformation matrix, premultiply it with the coordinates to be transformed (as opposed to postmultiplying).

Examples

Convert Axis-Angle Rotation to Homogeneous Transformation

```
axang = [1 0 0 pi/2];
tform = axang2tform(axang)
```

```
tform = 4×4

    1.0000         0         0         0
         0    0.0000   -1.0000         0
         0    1.0000    0.0000         0
         0         0         0    1.0000
```

Input Arguments

axang — Rotation given in axis-angle form

n-by-4 matrix

Rotation given in axis-angle form, specified as an *n*-by-4 matrix of *n* axis-angle rotations. The first three elements of every row specify the rotation axis, and the last element defines the rotation angle (in radians).

Example: `[1 0 0 pi/2]`

Output Arguments

tform — Homogeneous transformation

4-by-4-by-*n* matrix

Homogeneous transformation matrix, specified by a 4-by-4-by-*n* matrix of *n* homogeneous transformations. When using the transformation matrix, premultiply it with the coordinates to be formed (as opposed to postmultiplying).

Example: `[0 0 1 0; 0 1 0 0; -1 0 0 0; 0 0 0 1]`

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`tform2axang`

Introduced in R2015a

buildMap

Build occupancy map from lidar scans

Syntax

```
map = buildMap(scans,poses,mapResolution,maxRange)
```

Description

`map = buildMap(scans,poses,mapResolution,maxRange)` creates a `occupancyMap` map by inserting lidar `scans` at the given `poses`. Specify the resolution of the resulting map, `mapResolution`, and the maximum range of the lidar sensor, `maxRange`.

Examples

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`.

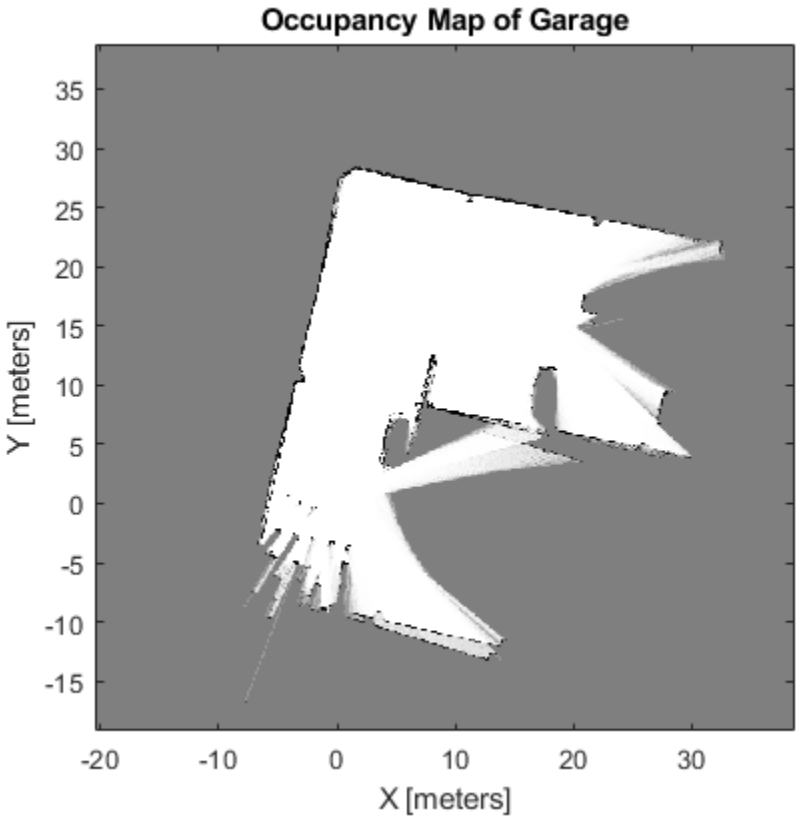
Load scan and pose estimates collected from sensors on a robot in a parking garage. The data collected is correlated using a `lidarSLAM` algorithm, which performs scan matching to associate scans and adjust poses over the full robot trajectory. Check to make sure scans and poses are the same length.

```
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')
```



Perform SLAM Using Lidar Scans

Use a `lidarSLAM` object to iteratively add and compare lidar scans and build an optimized pose graph of the robot trajectory. To get an occupancy map from the associated poses and scans, use the `buildMap` function.

Load Data and Set Up SLAM Algorithm

Load a cell array of `lidarScan` objects. The lidar scans were collected in a parking garage on a Husky® robot from ClearPath Robotics®. Typically, lidar scans are taken at a high frequency and each scan is not needed for SLAM. Therefore, down sample the scans by selecting only every 40th scan.

```
load garage_fl1_southend.mat scans
scans = scans(1:40:end);
```

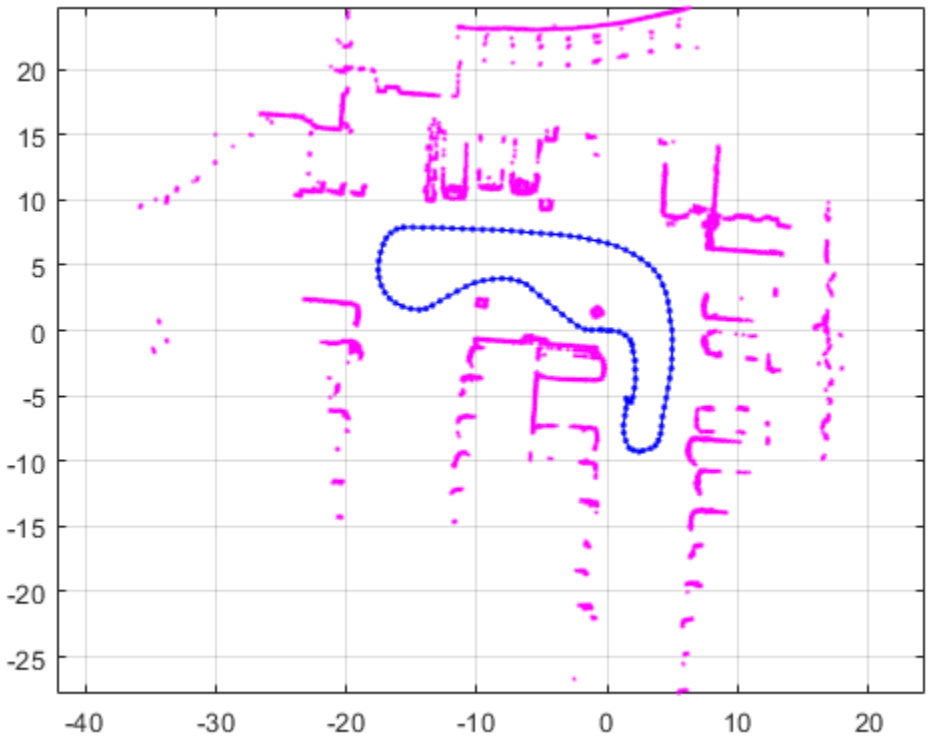
To set up the SLAM algorithm, specify the lidar range, map resolution, loop closure threshold, and search radius. Tune these parameters for your specific robot and environment. Create the `lidarSLAM` object with these parameters.

```
maxRange = 19.2; % meters
resolution = 10; % cells per meter

slamObj = lidarSLAM(resolution,maxRange);
slamObj.LoopClosureThreshold = 360;
slamObj.LoopClosureSearchRadius = 8;
```

Add Scans Iteratively

Using a `for` loop, add scans to the SLAM object. The object uses scan matching to compare each added scan to previously added ones. To improve the map, the object optimizes the pose graph whenever it detects a loop closure. Every 10 scans, display the stored poses and scans.

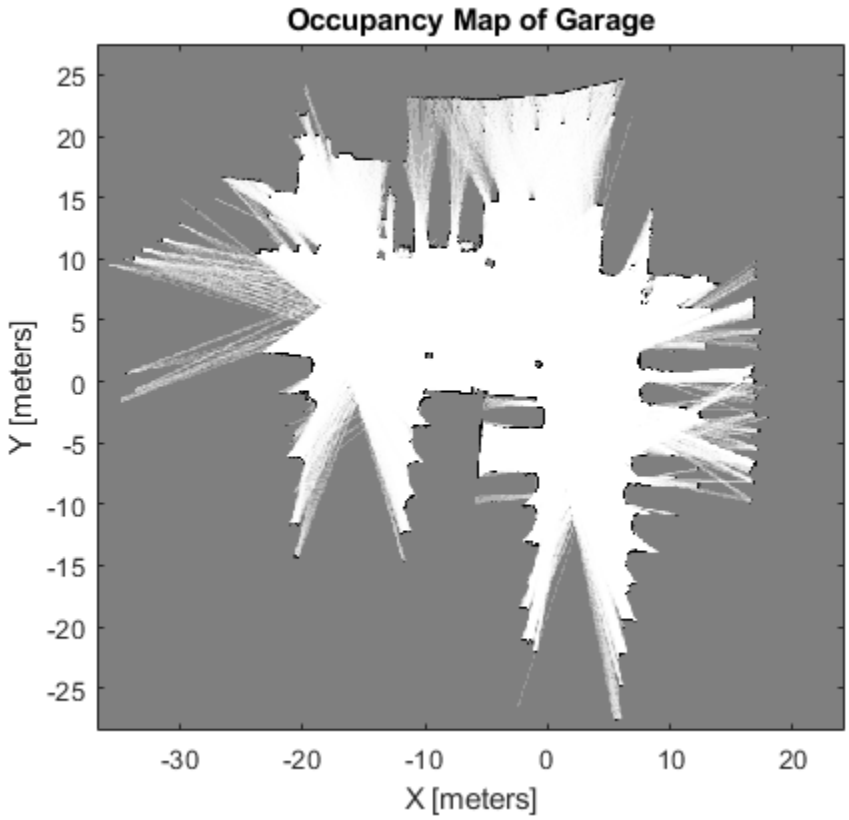
```
for i = 1:numel(scans)

    addScan(slamObj,scans{i});

    if rem(i,10) == 0
        show(slamObj);
    end
end
```



View Occupancy Map

After adding all the scans to the SLAM object, build an `occupancyMap` map by calling `buildMap` with the scans and poses. Use the same map resolution and max range you used with the SLAM object.

```
[scansSLAM,poses] = scansAndPoses(slamObj);
occMap = buildMap(scansSLAM,poses,resolution,maxRange);
figure
show(occMap)
title('Occupancy Map of Garage')
```



Input Arguments

`scans` — Lidar scans
cell array of `lidarScan` objects

Lidar scans used to build the map, specified as a cell array of `lidarScan` objects.

`poses` — Poses of lidar scans
n-by-3 matrix

Poses of lidar scans, specified as an *n*-by-3 matrix. Each row is an `[x y theta]` vector representing the *xy*-position and orientation angle of a scan.

`mapResolution` — Resolution of occupancy grid
positive integer

Resolution of the output `occupancyMap` map, specified as a positive integer in cells per meter.

`maxRange` — Maximum range of lidar sensor
positive scalar

Maximum range of lidar sensor, specified as a positive scalar in meters. Points in the `scans` outside this range are ignored.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `['MapWidth',10]`

`MapWidth` — Width of occupancy grid
positive scalar

Width of the occupancy grid, specified as the comma-separated pair consisting of `'MapWidth'` and a positive scalar. If this value is not specified, the map is automatically scaled to fit all laser scans.

`MapHeight` — Height of occupancy grid
positive scalar

Height of occupancy grid, specified as the comma-separated pair consisting of `'MapHeight'` and a positive scalar. If this value is not specified, the map is automatically scaled to fit all laser scans.

Output Arguments

`map` — Occupancy Map
`occupancyMap` object

Occupancy map, returned as a `occupancyMap` object.

See Also

Functions
`matchScans` | `matchScansGrid` | `lidarScan` | `transformScan`

Classes
`lidarSLAM` | `occupancyMap`

Topics
“Implement Simultaneous Localization And Mapping (SLAM) with Lidar Scans”
“Implement Online Simultaneous Localization And Mapping (SLAM) with Lidar Scans”

Introduced in R2019b

cart2hom

Convert Cartesian coordinates to homogeneous coordinates

Syntax

```
hom = cart2hom(cart)
```

Description

`hom = cart2hom(cart)` converts a set of points in Cartesian coordinates to homogeneous coordinates.

Examples

Convert 3-D Cartesian Points to Homogeneous Coordinates

```
c = [0.8147 0.1270 0.6324; 0.9058 0.9134 0.0975];
h = cart2hom(c)
```

```
h = 2×4

    0.8147    0.1270    0.6324    1.0000
    0.9058    0.9134    0.0975    1.0000
```

Input Arguments

`cart` — Cartesian coordinates

n-by-(*k*–1) matrix

Cartesian coordinates, specified as an *n*-by-(*k*–1) matrix, containing *n* points. Each row of `cart` represents a point in (*k*–1)-dimensional space. *k* must be greater than or equal to 2.

Example: `[0.8147 0.1270 0.6324; 0.9058 0.9134 0.0975]`

Output Arguments

`hom` — Homogeneous points

n-by-*k* matrix

Homogeneous points, returned as an *n*-by-*k* matrix, containing *n* points. *k* must be greater than or equal to 2.

Example: `[0.2785 0.9575 0.1576 0.5; 0.5469 0.9649 0.9706 0.5]`

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`hom2cart`

Introduced in R2015a

connect

Connect poses for given connection type

Syntax

```
[pathSegments,pathCosts] = connect(connectionObj,start,goal)
[pathSegments,pathCosts] = connect(connectionObj,start,
goal,'PathSegments','all')
```

Description

`[pathSegments,pathCosts] = connect(connectionObj,start,goal)` connects the start and goal poses using the specified `dubinsConnection` object. The path segment object with the lowest cost is returned.

`[pathSegments,pathCosts] = connect(connectionObj,start,
goal,'PathSegments','all')` returns all possible path segments as a cell array with their associated costs.

Examples

Connect Poses Using Dubins Connection Path

Create a `dubinsConnection` object.

```
dubConnObj = dubinsConnection;
```

Define start and goal poses as `[x y theta]` vectors.

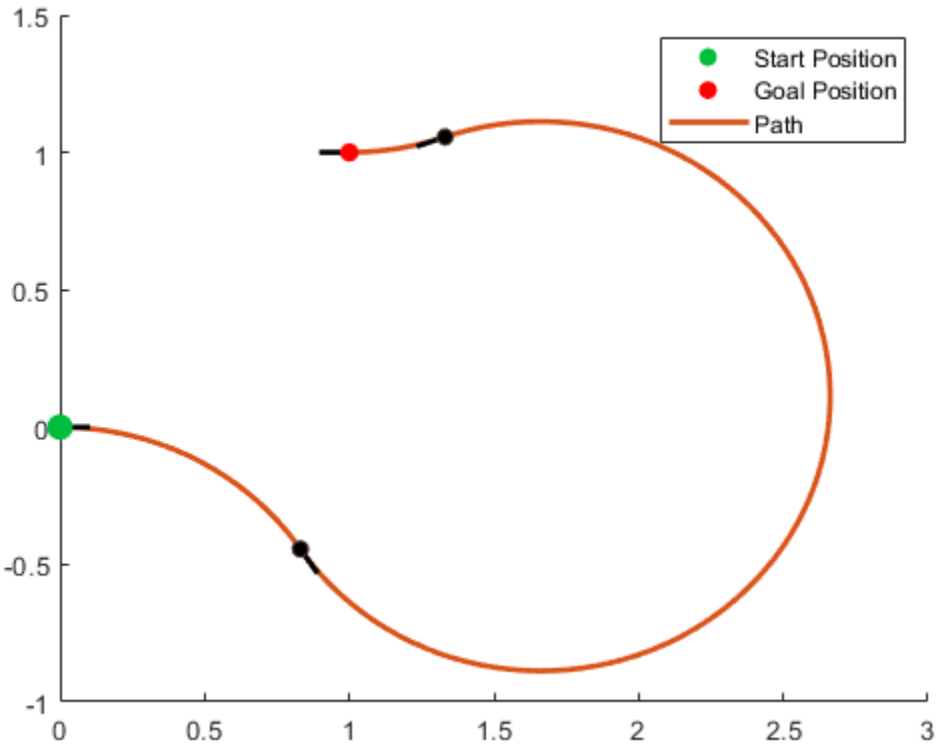
```
startPose = [0 0 0];
goalPose = [1 1 pi];
```

Calculate a valid path segment to connect the poses.

```
[pathSegObj, pathCosts] = connect(dubConnObj,startPose,goalPose);
```

Show the generated path.

```
show(pathSegObj{1})
```



Connect Poses Using ReedsShepp Connection Path

Create a `reedsSheppConnection` object.

```
reedsConnObj = reedsSheppConnection;
```

Define start and goal poses as `[x y theta]` vectors.

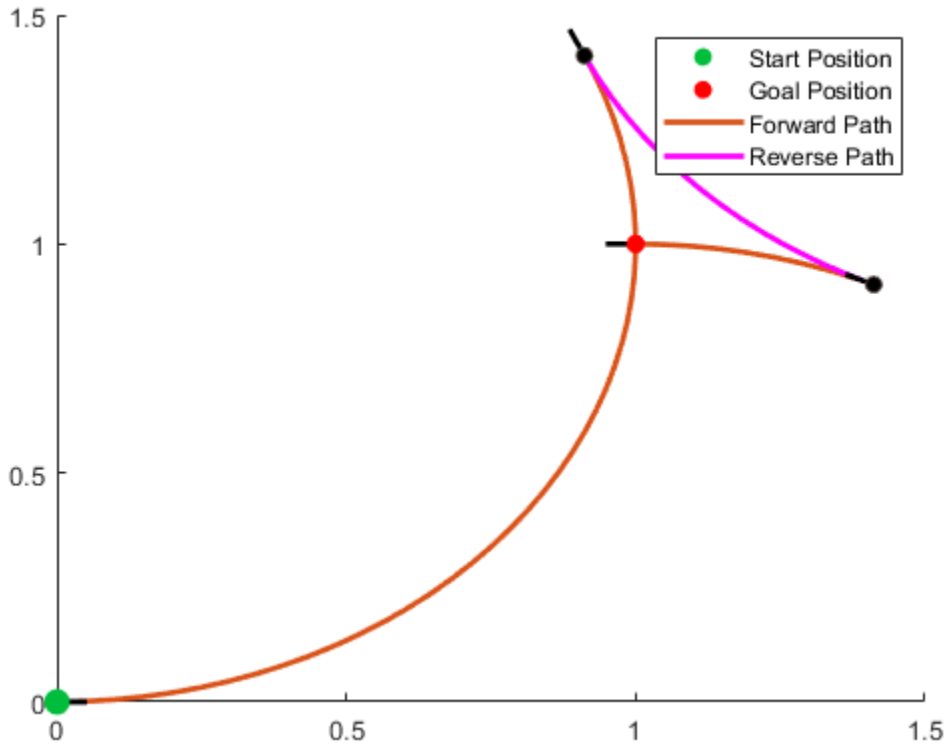
```
startPose = [0 0 0];
goalPose = [1 1 pi];
```

Calculate a valid path segment to connect the poses.

```
[pathSegObj,pathCosts] = connect(reedsConnObj,startPose,goalPose);
```

Show the generated path.

```
show(pathSegObj{1})
```



Input Arguments

`connectionObj` — Path connection type

`dubinsPathSegment` object | `reedsSheppPathSegment` object

Path connection type, specified as a `dubinsConnection` or `reedsSheppConnection` object. This object defines the parameters of the connection, including the minimum turning radius of the robot and the valid motion types.

`start` — Initial pose of robot

[x, y, Θ] vector or matrix

This property is read-only.

Initial pose of the robot at the start of the path segment, specified as an [x, y, Θ] vector or matrix. Each row of the matrix corresponds to a different start pose.

x and y are in meters. Θ is in radians.

The `connect` function supports:

- Singular start pose with singular goal pose.
- Multiple start pose with singular goal pose.

- Singular start pose with multiple goal pose.
- Multiple start pose with multiple goal pose.

The output `pathSegments` cell array size reflects the singular or multiple pose options.

`goal` — Goal pose of robot
[x, y, Θ] vector or matrix

This property is read-only.

Goal pose of the robot at the end of the path segment, specified as an [x, y, Θ] vector or matrix. Each row of the matrix corresponds to a different goal pose.

x and y are in meters. Θ is in radians.

The `connect` function supports:

- Singular start pose with singular goal pose.
- Multiple start pose with singular goal pose.
- Singular start pose with multiple goal pose.
- Multiple start pose with multiple goal pose.

The output `pathSegments` cell array size reflects the singular or multiple pose options.

Output Arguments

`pathSegments` — Path segments
cell array of objects

Path segments, specified as a cell array of objects. The type of object depends on the input `connectionObj`. The size of the cell array depends on whether you use singular or multiple `start` and `goal` poses. By default, the function returns the path with the lowest cost for each `start` and `goal` pose. When call `connect` using the `'PathSegments','all'` name-value pair, the cell array contains all valid path segments between the specified `start` and `goal` poses.

`pathCosts` — Cost of path segment
positive numeric scalar | positive numeric vector | positive numeric matrix

Cost of path segments, specified as a positive numeric scalar, vector, or matrix. Each element of the cost vector or matrix corresponds to a path segment in `pathSegment`. By default, the function returns the path with the lowest cost for each start and goal pose.

Example: `[7.6484,7.5122]`

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

Functions
`interpolate` | `show`

Objects
`dubinsConnection` | `dubinsPathSegment` | `reedsSheppConnection` | `reedsSheppPathSegment`

Introduced in R2019b

createPlanningTemplate

Create sample implementation for path planning interface

Syntax

```
createPlanningTemplate
createPlanningTemplate("StateValidator")
```

Description

`createPlanningTemplate` creates a planning template for a subclass of the `nav.StateSpace` class. The function opens a file in the MATLAB® Editor. Save your custom implementation and ensure the file is available on the MATLAB path. Alternative syntax: `createPlanningTemplate("StateSpace")`

`createPlanningTemplate("StateValidator")` creates a template for a subclass of the `nav.StateValidator` class.

Examples

Create Custom State Space for Path Planning

This example shows how to use the `createPlanningTemplate` function to generate a template for customizing your own state space definition and sampler to use with path planning algorithms. A simple implementation is provided with the template.

Call the create template function. This function generates a class definition file for you to modify for your own implementation.

```
createPlanningTemplate
```

Class and Property Definition

The first part of the template specifies the class definition and any properties for the class. Derive from the `nav.StateSpace` class. For this example, create a property for the uniform and normal distributions. You can specify any additional user-defined properties here.

```
classdef MyCustomStateSpace < nav.StateSpace & ...
        matlabshared.planning.internal.EnforceScalarHandle
    properties
       UniformDistribution
       NormalDistribution
       % Specify additional properties here
end
```

Save your custom state space class and ensure your file name matches the class name.

Class Constructor

Use the constructor to set the name of the state space, the number of state variables, and define its boundaries. Alternatively, you can add input arguments to the function and pass the variables in when you create an object.

- For each state variable, define the `[min max]` values for the state bounds.
- Call the constructor of the base class.
- For this example, you specify the normal and uniform distribution property values using predefined `NormalDistribution` and `UniformDistribution` classes.
- Specify any other user-defined property values here.

```
methods
    function obj = MyCustomStateSpace
        spaceName = "MyCustomStateSpace";
        numStateVariables = 3;
        stateBounds = [-100 100;  % [min max]
                       -100 100;
                       -100 100];

        obj@nav.StateSpace(spaceName, numStateVariables, stateBounds);

        obj.NormalDistribution = matlabshared.tracking.internal.NormalDistribution(numStateVariab
        obj.UniformDistribution = matlabshared.tracking.internal.UniformDistribution(numStateVari
        % User-defined property values here
    end
```

Copy Semantics

Specify the `copy` method definition. Copy all the values of your user-defined variables into a new object, so `copyObj` is a deep copy. The default behavior given in this example creates a new copy of the object with the same name, state bounds, and distributions.

```
function copyObj = copy(obj)
    copyObj = feval(class(obj));
    copyObj.StateBounds = obj.StateBounds;
    copyObj.UniformDistribution = obj.UniformDistribution.copy;
    copyObj.NormalDistribution = obj.NormalDistribution.copy;
end
```

Enforce State Bounds

Specify how to ensure states are always within the state bounds. For this example, the state values get saturated at the minimum or maximum values for the state bounds.

```
function boundedState = enforceStateBounds(obj, state)
    nav.internal.validation.validateStateMatrix(state, nan, obj.NumStateVariables, "enforceStateB
    boundedState = state;
    boundedState = min(max(boundedState, obj.StateBounds(:,1)'), ...
        obj.StateBounds(:,2)');

end
```

Sample Uniformly

Specify the behavior for sampling across a uniform distribution. support multiple syntaxes to constrain the uniform distribution to a nearby state within a certain distance and sample multiple states.

```
STATE = sampleUniform(OBJ)
STATE = sampleUniform(OBJ,NUMSAMPLES)
STATE = sampleUniform(OBJ,NEARSTATE,DIST)
STATE = sampleUniform(OBJ,NEARSTATE,DIST,NUMSAMPLES)
```

For this example, use a validation function to process a `varargin` input that handles the varying input arguments.

```
 function state = sampleUniform(obj, varargin)
    narginchk(1,4);
    [numSamples, stateBounds] = obj.validateSampleUniformInput(varargin{:});

    obj.UniformDistribution.RandomVariableLimits = stateBounds;
    state = obj.UniformDistribution.sample(numSamples);
 end
```

Sample from Gaussian Distribution

Specify the behavior for sampling across a Gaussian distribution. Support multiple syntaxes for sampling a single state or multiple states.

```
STATE = sampleGaussian(OBJ, MEANSTATE, STDDEV)
STATE = sampleGaussian(OBJ, MEANSTATE, STDDEV, NUMSAMPLES)


function state = sampleGaussian(obj, meanState, stdDev, varargin)
    narginchk(3,4);

    [meanState, stdDev, numSamples] = obj.validateSampleGaussianInput(meanState, stdDev, varargi

    obj.NormalDistribution.Mean = meanState;
    obj.NormalDistribution.Covariance = diag(stdDev.^2);

    state = obj.NormalDistribution.sample(numSamples);
    state = obj.enforceStateBounds(state);

end
```

Interpolate Between States

Define how to interpolate between two states in your state space. Use an input, `fraction`, to determine how to sample along the path between two states. For this example, define a basic linear interpolation method using the difference between states.

```
function interpState = interpolate(obj, state1, state2, fraction)
    narginchk(4,4);
    [state1, state2, fraction] = obj.validateInterpolateInput(state1, state2, fraction);

    stateDiff = state2 - state1;
    interpState = state1 + fraction' * stateDiff;
end
```

Calculate Distance Between States

Specify how to calculate the distance between two states in your state space. Use the `state1` and `state2` inputs to define the start and end positions. Both inputs can be a single state (row vector) or multiple states (matrix of row vectors). For this example, calculate the distance based on the Euclidean distance between each pair of state positions.

```
function dist = distance(obj, state1, state2)

    narginchk(3,3);

    nav.internal.validation.validateStateMatrix(state1, nan, obj.NumStateVariables, "distance", '
    nav.internal.validation.validateStateMatrix(state2, size(state1,1), obj.NumStateVariables, "d

    stateDiff = bsxfun(@minus, state2, state1);
    dist = sqrt( sum( stateDiff.^2, 2 ) );
end
```

Terminate the methods and class sections.

```
    end
end
```

Save your state space class definition. You can now use the class constructor to create an object for your state space.

Create Custom State Space Validator for Path Planning

This example shows how to use the `createPlanningTemplate` function to generate a template for customizing your own state validation class. State validation is used with path planning algorithms to ensure valid paths. The template function provides a basic implementation for example purposes.

Call the create template function. This function generates a class definition file for you to modify for your own implementation. Save this file.

```
createPlanningTemplate("StateValidator")
```

Class and Property Definition

The first part of the template specifies the class definition and any properties for the class. Derive from the `nav.StateValidator` class. You can specify any additional user-defined properties here.

```
classdef MyCustomStateValidator < nav.StateValidator & ...
        matlabshared.planning.internal.EnforceScalarHandle
    properties
       % User-defined properties
    end
```

Save your custom state validator class and ensure your file name matches the class name.

Class Constructor

Use the constructor to set the name of the state space validator and specify the state space object. Set a default value for the state space if one is not provided. Call the constructor of the base class. Initialize any other user-defined properties. This example uses a default of `MyCustomStateSpace`, which was illustrated in the previous example.

```
methods
        function obj = MyCustomStateValidator(space)
            narginchk(0,1)

            if nargin == 0
                space = MyCustomStateSpace;
            end

            obj@nav.StateValidator(space);

           % Initialize user-defined properties
        end
```

Copy Semantics

Specify the `copy` method definition. Copy all the values of your user-defined variables into a new object, so `copyObj` is a deep copy. The default behavior given in this example creates a new copy of the object with the same type.

```
        function copyObj = copy(obj)
            copyObj = feval(class(obj), obj.StateSpace);
        end
```

Check State Validity

Define how a given state is validated. The `state` input can either be a single row vector, or a matrix of row vectors for multiple states. Customize this function for any special validation behavior for your state space like collision checking against obstacles.

```
        function isValid = isStateValid(obj, state)
            narginchk(2,2);
            nav.internal.validation.validateStateMatrix(state, nan, obj.StateSpace.NumStateVariab
                "isStateValid", "state");

            bounds = obj.StateSpace.StateBounds';
            inBounds = state >= bounds(1,:) & state <= bounds(2,:);
            isValid = all(inBounds, 2);

        end
```

Check Motion Validity

Define how to generate the motion between states and determine if it is valid. For this example, use `linspace` to evenly interpolate between states and check if these states are valid using `isStateValid`. Customize this function to sample between states or consider other analytical methods for determining if a vehicle can move between given states.

```
        function [isValid, lastValid] = isMotionValid(obj, state1, state2)
            narginchk(3,3);
            state1 = nav.internal.validation.validateStateVector(state1, ...
                obj.StateSpace.NumStateVariables, "isMotionValid", "state1");
            state2 = nav.internal.validation.validateStateVector(state2, ...
                obj.StateSpace.NumStateVariables, "isMotionValid", "state2");

            if (~obj.isStateValid(state1))
                error("statevalidator:StartStateInvalid", "The start state of the motion is inva
            end
```

```
            % Interpolate at a fixed interval between states and check state validity
            numInterpPoints = 100;
            interpStates = obj.StateSpace.interpolate(state1, state2, linspace(0,1,numInterpPoint
            interpValid = obj.isStateValid(interpStates);

            % Look for invalid states. Set lastValid state to index-1.
            firstInvalidIdx = find(~interpValid, 1);
            if isempty(firstInvalidIdx)
                isValid = true;
                lastValid = state2;
            else
                isValid = false;
                lastValid = interpStates(firstInvalidIdx-1,:);
            end

        end
```

Terminate the methods and class sections.

```
    end
end
```

Save your state space validator class definition. You can now use the class constructor to create an object for validation of states for a given state space.

See Also

`nav.StateSpace` | `nav.StateValidator` | `stateSpaceSE2` | `validatorOccupancyMap`

Introduced in R2019b

ecompass

Orientation from magnetometer and accelerometer readings

Syntax

```
orientation = ecompass(accelerometerReading,magnetometerReading)
orientation = ecompass(accelerometerReading,magnetometerReading,
orientationFormat)
orientation = ecompass(accelerometerReading,magnetometerReading,
orientationFormat,'ReferenceFrame',RF)
```

Description

`orientation = ecompass(accelerometerReading,magnetometerReading)` returns a `quaternion` that can rotate quantities from a parent (NED) frame to a child (sensor) frame.

`orientation = ecompass(accelerometerReading,magnetometerReading, orientationFormat)` specifies the orientation format as `quaternion` or rotation matrix.

`orientation = ecompass(accelerometerReading,magnetometerReading, orientationFormat,'ReferenceFrame',RF)` also allows you to specify the reference frame `RF` of the `orientation` output. Specify `RF` as `'NED'` (North-East-Down) or `'ENU'` (East-North-Up). The default value is `'NED'`.

Examples

Determine Declination of Boston

Use the known magnetic field strength and proper acceleration of a device pointed true north in Boston to determine the magnetic declination of Boston.

Define the known acceleration and magnetic field strength in Boston.

```
magneticFieldStrength = [19.535 -5.109 47.930];
properAcceleration = [0 0 9.8];
```

Pass the magnetic field strength and acceleration to the `ecompass` function. The `ecompass` function returns a quaternion rotation operator. Convert the quaternion to Euler angles in degrees.

```
q = ecompass(properAcceleration,magneticFieldStrength);
e = eulerd(q,'ZYX','frame');
```

The angle, `e`, represents the angle between true north and magnetic north in Boston. By convention, magnetic declination is negative when magnetic north is west of true north. Negate the angle to determine the magnetic declination.

```
magneticDeclinationOfBoston = -e(1)
```

```
magneticDeclinationOfBoston = -14.6563
```

Return Rotation Matrix

The `ecompass` function fuses magnetometer and accelerometer data to return a quaternion that, when used within a quaternion rotation operator, can rotate quantities from a parent (NED) frame to a child frame. The `ecompass` function can also return rotation matrices that perform equivalent rotations as the quaternion operator.

Define a rotation that can take a parent frame pointing to magnetic north to a child frame pointing to geographic north. Define the rotation as both a quaternion and a rotation matrix. Then, convert the quaternion and rotation matrix to Euler angles in degrees for comparison.

Define the magnetic field strength in microteslas in Boston, MA, when pointed true north.

```
m = [19.535 -5.109 47.930];
a = [0 0 9.8];
```

Determine the quaternion and rotation matrix that is capable of rotating a frame from magnetic north to true north. Display the results for comparison.

```
q = ecompass(a,m);
quaterionEulerAngles = eulerd(q,'ZYX','frame')
```

```
quaterionEulerAngles = 1×3

   14.6563         0         0
```

```
r = ecompass(a,m,'rotmat');
theta = -asin(r(1,3));
psi = atan2(r(2,3)/cos(theta),r(3,3)/cos(theta));
rho = atan2(r(1,2)/cos(theta),r(1,1)/cos(theta));
rotmatEulerAngles = rad2deg([rho,theta,psi])
```

```
rotmatEulerAngles = 1×3

   14.6563         0         0
```

Determine Gravity Vector

Use `ecompass` to determine the gravity vector based on data from a rotating IMU.

Load the inertial measurement unit (IMU) data.

```
load 'rpy_9axis.mat' sensorData Fs
```

Determine the orientation of the sensor body relative to the local NED frame over time.

```
orientation = ecompass(sensorData.Acceleration,sensorData.MagneticField);
```

To estimate the gravity vector, first rotate the accelerometer readings from the sensor body frame to the NED frame using the `orientation` quaternion vector.

```
gravityVectors = rotatepoint(orientation,sensorData.Acceleration);
```

Determine the gravity vector as an average of the recovered gravity vectors over time.

```
gravityVectorEstimate = mean(gravityVectors,1)
```

```
gravityVectorEstimate = 1×3

    0.0000   -0.0000   10.2102
```

Track Spinning Platform

Fuse modeled accelerometer and gyroscope data to track a spinning platform using both idealized and realistic data.

Generate Ground-Truth Trajectory

Describe the ground-truth orientation of the platform over time. Use the `kinematicTrajectory` on page 2-462 System object™ to create a trajectory for a platform that has no translation and spins about its *z*-axis.

```
duration = 12;
fs = 100;
numSamples = fs*duration;

accelerationBody = zeros(numSamples,3);

angularVelocityBody = zeros(numSamples,3);
zAxisAngularVelocity = [linspace(0,4*pi,4*fs),4*pi*ones(1,4*fs),linspace(4*pi,0,4*fs)]';
angularVelocityBody(:,3) = zAxisAngularVelocity;

trajectory = kinematicTrajectory('SampleRate',fs);

[~,orientationNED,~,accelerationNED,angularVelocityNED] = trajectory(accelerationBody,angularVelo
```

Model Receiving IMU Data

Use an `imuSensor` on page 2-420 System object to mimic data received from an IMU that contains an ideal magnetometer and an ideal accelerometer.

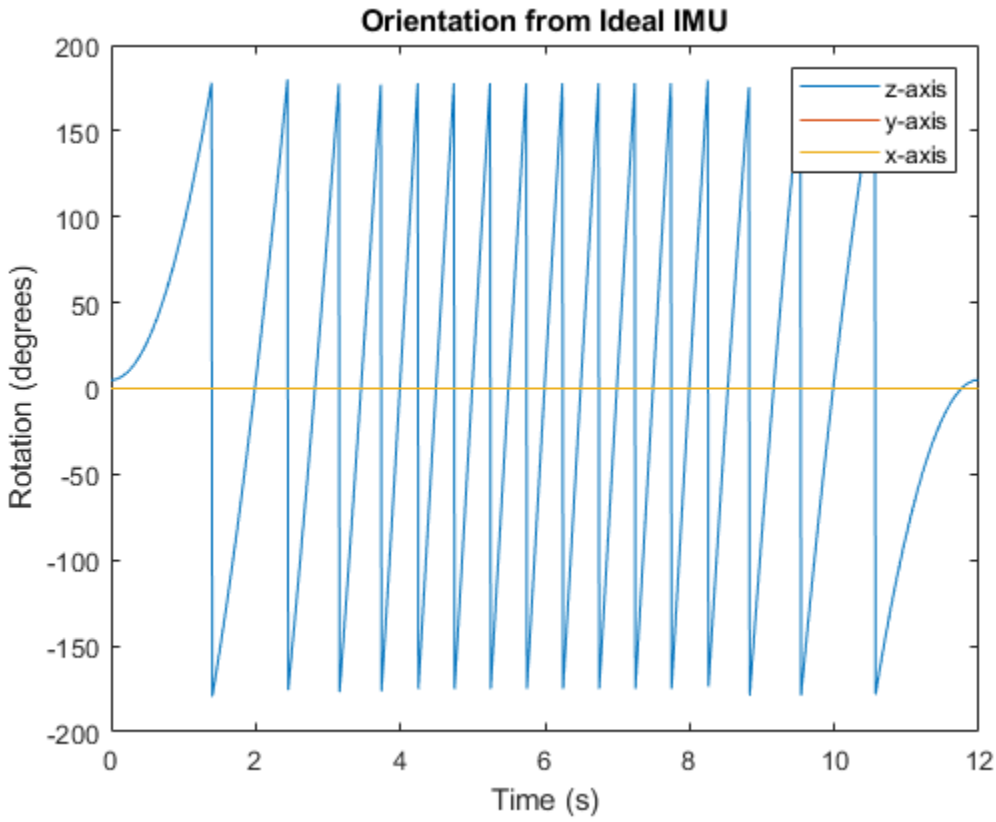
```
IMU = imuSensor('accel-mag','SampleRate',fs);
[accelerometerData,magnetometerData] = IMU(accelerationNED, ...
                                           angularVelocityNED, ...
                                           orientationNED);
```

Fuse IMU Data to Estimate Orientation

Pass the accelerometer data and magnetometer data to the `ecompass` function to estimate orientation over time. Convert the orientation to Euler angles in degrees and plot the result.

```
orientation = ecompass(accelerometerData,magnetometerData);
orientationEuler = eulerd(orientation,'ZYX','frame');

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

```
figure(1)
plot(timeVector,orientationEuler)
legend('z-axis','y-axis','x-axis')
xlabel('Time (s)')
ylabel('Rotation (degrees)')
title('Orientation from Ideal IMU')
```

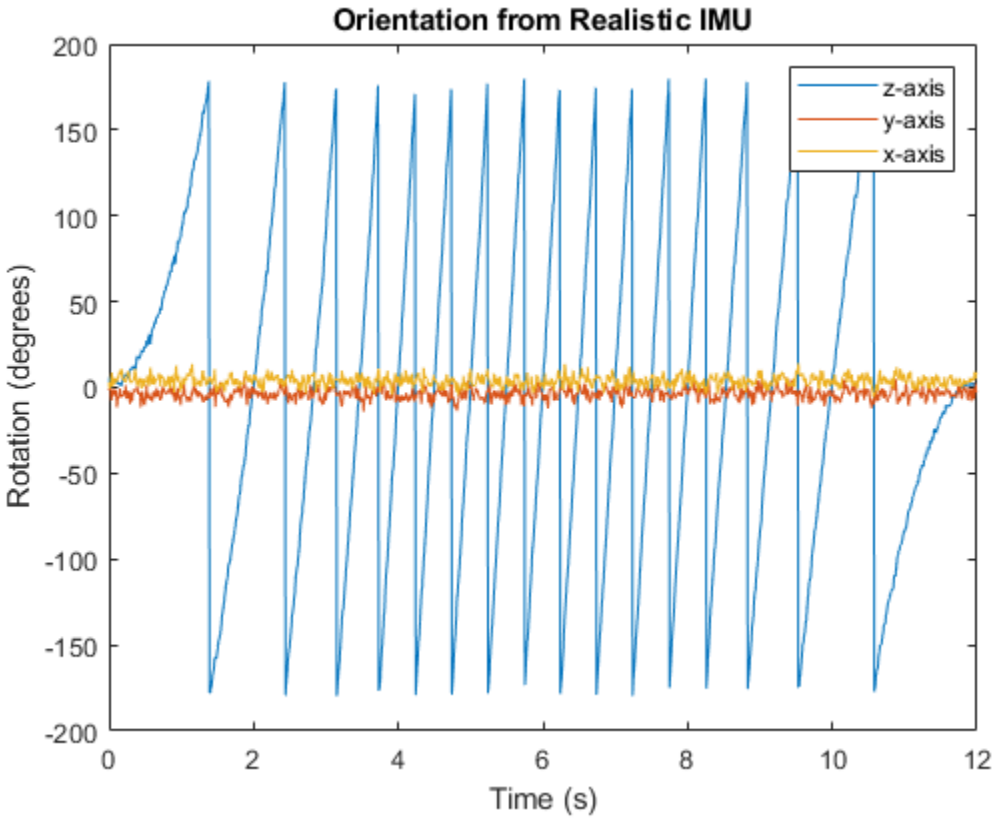


Repeat Experiment with Realistic IMU Sensor Model

Modify parameters of the `IMU` System object to approximate realistic IMU sensor data. Reset the `IMU` and then call it with the same ground-truth acceleration, angular velocity, and orientation. Use `ecompass` to fuse the IMU data and plot the results.

```
IMU.Accelerometer = accelparams( ...
    'MeasurementRange',20, ...
    'Resolution',0.0006, ...
    'ConstantBias',0.5, ...
    'AxesMisalignment',2, ...
    'NoiseDensity',0.004, ...
    'BiasInstability',0.5);
IMU.Magnetometer = magparams( ...
    'MeasurementRange',200, ...
    'Resolution',0.01);
reset(IMU)

[accelerometerData,magnetometerData] = IMU(accelerationNED,angularVelocityNED,orientationNED);
```

```
orientation = ecompass(accelerometerData,magnetometerData);
orientationEuler = eulerd(orientation,'ZYX','frame');

figure(2)
plot(timeVector,orientationEuler)
legend('z-axis','y-axis','x-axis')
xlabel('Time (s)')
ylabel('Rotation (degrees)')
title('Orientation from Realistic IMU')
```



Input Arguments

`accelerometerReading` — Accelerometer readings in sensor body coordinate system (m/s²)

N-by-3 matrix

Accelerometer readings in sensor body coordinate system in m/s², specified as an *N*-by-3 matrix. The columns of the matrix correspond to the *x*-, *y*-, and *z*-axes of the sensor body. The rows in the matrix, *N*, correspond to individual samples. The accelerometer readings are normalized before use in the function.

Data Types: `single` | `double`

`magnetometerReading` — Magnetometer readings in sensor body coordinate system (µT)

N-by-3 matrix

Magnetometer readings in sensor body coordinate system in µT, specified as an *N*-by-3 matrix. The columns of the matrix correspond to the *x*-, *y*-, and *z*-axes of the sensor body. The rows in the matrix,

N, correspond to individual samples. The magnetometer readings are normalized before use in the function.

Data Types: `single` | `double`

`orientationFormat` — Format used to describe orientation
`'quaternion'` (default) | `'rotmat'`

Format used to describe orientation, specified as `'quaternion'` or `'rotmat'`.

Data Types: `char` | `string`

Output Arguments

`orientation` — Orientation that rotates quantities from global coordinate system to sensor body coordinate system
N-by-1 vector of quaternions (default) | 3-by-3-by-*N* array

Orientation that can rotate quantities from a global coordinate system to a body coordinate system, returned as a vector of quaternions or an array. The size and type of the `orientation` depends on the format used to describe orientation:

- `'quaternion'` –– *N*-by-1 vector of quaternions with the same underlying data type as the input
- `'rotmat'` –– 3-by-3-by-*N* array the same data type as the input

Data Types: `quaternion` | `single` | `double`

Algorithms

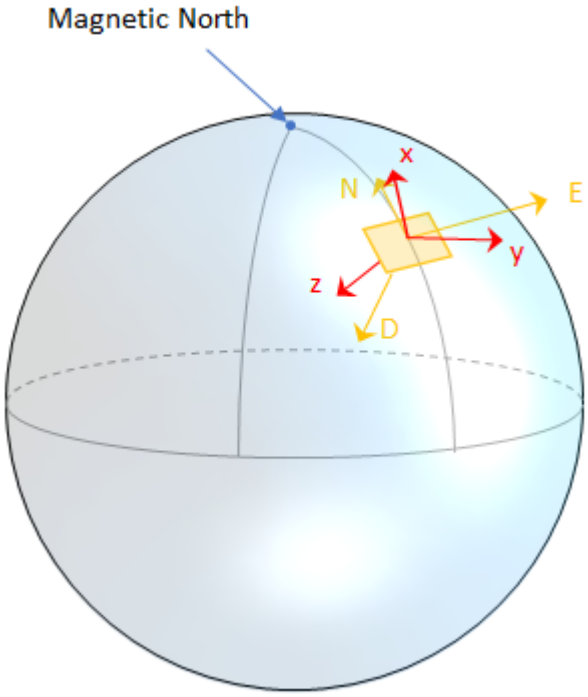
The `ecompass` function returns a quaternion or rotation matrix that can rotate quantities from a parent (NED for example) frame to a child (sensor) frame. For both output orientation formats, the rotation operator is determined by computing the rotation matrix.

The rotation matrix is first calculated with an intermediary:

$$R = \left[(a \times m) \times a \;\; a \times m \;\; a\right]$$

and then normalized column-wise. a and m are the `accelerometerReading` input and the `magnetometerReading` input, respectively.

To understand the rotation matrix calculation, consider an arbitrary point on the Earth and its corresponding local NED frame. Assume a sensor body frame, [*x*,*y*,*z*], with the same origin.



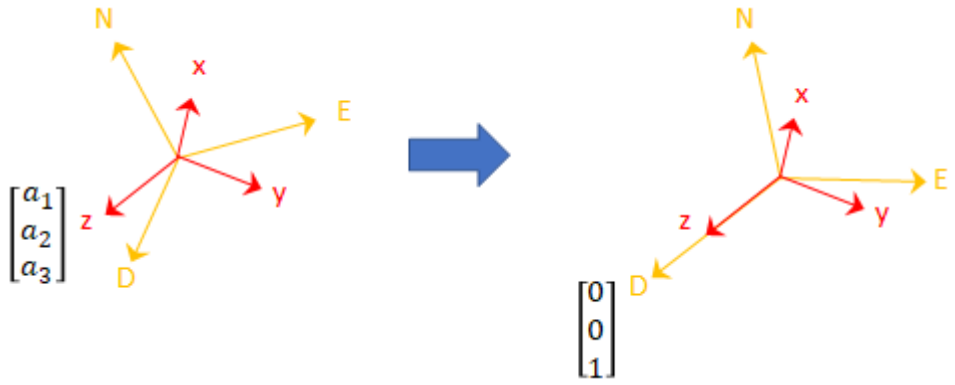
Recall that orientation of a sensor body is defined as the rotation operator (rotation matrix or quaternion) required to rotate a quantity from a parent (NED) frame to a child (sensor body) frame:

$$\left[R \right]\left[p_{\text{parent}}\right] = \left[p_{\text{child}}\right]$$

where

- R is a 3-by-3 rotation matrix, which can be interpreted as the orientation of the child frame.
- p_{parent} is a 3-by-1 vector in the parent frame.
- p_{child} is a 3-by-1 vector in the child frame.

For a stable sensor body, an accelerometer returns the acceleration due to gravity. If the sensor body is perfectly aligned with the NED coordinate system, all acceleration due to gravity is along the *z*-axis, and the accelerometer reads [0 0 1]. Consider the rotation matrix required to rotate a quantity from the NED coordinate system to a quantity indicated by the accelerometer.

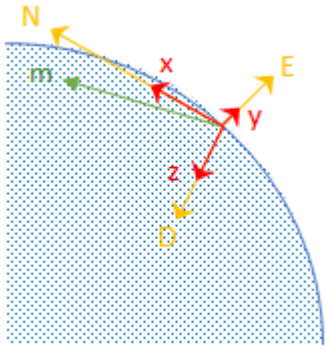


$$\begin{bmatrix} r_{11} & r_{21} & r_{31} \\ r_{12} & r_{22} & r_{32} \\ r_{13} & r_{23} & r_{33} \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix}$$

The third column of the rotation matrix corresponds to the accelerometer reading:

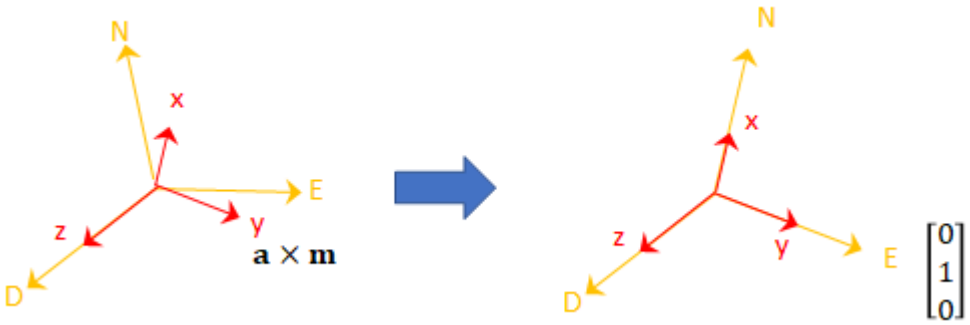
$$\begin{bmatrix} r_{31} \\ r_{32} \\ r_{33} \end{bmatrix} = \begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix}$$

A magnetometer reading points toward magnetic north and is in the *N*-*D* plane. Again, consider a sensor body frame aligned with the NED coordinate system.



By definition, the *E*-axis is perpendicular to the *N*-*D* plane, therefore $N \times D = E$, within some amplitude scaling. If the sensor body frame is aligned with NED, both the acceleration vector from the accelerometer and the magnetic field vector from the magnetometer lie in the *N*-*D* plane. Therefore $m \times a = y$, again with some amplitude scaling.

Consider the rotation matrix required to rotate NED to the child frame, [*x y z*].



$$\begin{bmatrix} r_{11} & r_{21} & r_{31} \\ r_{12} & r_{22} & r_{32} \\ r_{13} & r_{23} & r_{33} \end{bmatrix} \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} = \begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix} \times \begin{bmatrix} m_1 \\ m_2 \\ m_3 \end{bmatrix}$$

The second column of the rotation matrix corresponds to the cross product of the accelerometer reading and the magnetometer reading:

$$\begin{bmatrix} r_{21} \\ r_{22} \\ r_{23} \end{bmatrix} = \begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix} \times \begin{bmatrix} m_1 \\ m_2 \\ m_3 \end{bmatrix}$$

By definition of a rotation matrix, column 1 is the cross product of columns 2 and 3:

$$\begin{bmatrix} r_{11} \\ r_{12} \\ r_{13} \end{bmatrix} = \begin{bmatrix} r_{21} \\ r_{22} \\ r_{23} \end{bmatrix} \times \begin{bmatrix} r_{31} \\ r_{32} \\ r_{33} \end{bmatrix} = (a \times m) \times a$$

Finally, the rotation matrix is normalized column-wise:

$$R_{ij} = \frac{R_{ij}}{\sqrt{\sum_{i=1}^{3} R_{ij}^2}} \quad , \forall j$$

Note The `ecompass` algorithm uses magnetic north, not true north, for the NED coordinate system.

References

[1] Open Source Sensor Fusion. https://github.com/memsindustrygroup/Open-Source-Sensor-Fusion/tree/master/docs

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`ahrsfilter` | `imufilter`

Introduced in R2018b

enu2lla

Transform local east-north-up coordinates to geodetic coordinates

Syntax

```
lla = enu2lla(xyzENU,lla0,method)
```

Description

`lla = enu2lla(xyzENU,lla0,method)` transforms the local east-north-up (ENU) Cartesian coordinates `xyzENU` to geodetic coordinates `lla`. Specify the origin of the local ENU system as the geodetic coordinates `lla0`.

Note

- The latitude and longitude values in the geodetic coordinate system use the World Geodetic System of 1984 (WGS84) standard.
- Specify altitude as height in meters above the WGS84 reference ellipsoid.

Examples

Transform ENU Coordinates to Geodetic Coordinates

Specify the geodetic coordinates of the local origin in Zermatt, Switzerland.

```
lla0 = [46.017 7.750 1673]; % [lat0 lon0 alt0]
```

Specify the ENU coordinates of a point of interest, in meters. In this case, the point of interest is the Matterhorn.

```
xyzENU = [-7134.8 -4556.3 2852.4]; % [xEast yNorth zUp]
```

Transform the local ENU coordinates to geodetic coordinates using flat earth approximation.

```
lla = enu2lla(xyzENU,lla0,'flat')
```

```
lla = 1×3
10^3 ×

    0.0460    0.0077    4.5254
```

Input Arguments

xyzENU — Local ENU Cartesian coordinates

three-element row vector | *n*-by-3 matrix

Local ENU Cartesian coordinates, specified as a three-element row vector or an *n*-by-3 matrix. *n* is the number of points to transform. Specify each point in the form [*xEast yNorth zUp*]. *xEast*, *yNorth*, and *zUp* are the respective *x*-, *y*-, and *z*-coordinates, in meters, of the point in the local ENU system.

Data Types: `double`

`lla0` — Origin of local ENU system in geodetic coordinates
three-element row vector | *n*-by-3 matrix

Origin of the local ENU system in the geodetic coordinates, specified as a three-element row vector or an *n*-by-3 matrix. *n* is the number of origin points. Specify each point in the form `[lat0 lon0 alt0]`. *lat0* and *lon0* specify the latitude and longitude of the origin, respectively, in degrees. *alt0* specifies the altitude of the origin in meters.

Data Types: `double`

`method` — Transformation method
`'flat'` | `'ellipsoid'`

Transformation method, specified as `'flat'` or `'ellipsoid'`. This argument specifies whether the function assumes the planet is flat or ellipsoidal.

The flat Earth transformation method has these limitations:

- Assumes that the flight path and bank angle are zero.
- Assumes that the flat Earth *z*-axis is normal to the Earth at only the initial geodetic latitude and longitude. This method has higher accuracy over small distances from the initial geodetic latitude and longitude, and closer to the equator. The method calculates a longitude with higher accuracy when the variation in latitude is smaller.
- Latitude values of +90 and -90 may return unexpected values because of singularity at the poles.

Data Types: `char` | `string`

Output Arguments

`lla` — Geodetic coordinates
three-element row vector | *n*-by-3 matrix

Geodetic coordinates, returned as a three-element row vector or an *n*-by-3 matrix. *n* is the number of transformed points. Each point is in the form `[lat lon alt]`. *lat* and *lon* specify the latitude and longitude, respectively, in degrees. *alt* specifies the altitude in meters.

Data Types: `double`

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

Functions
`lla2enu` | `lla2ned` | `ned2lla`

Introduced in R2021a

eul2quat

Convert Euler angles to quaternion

Syntax

```
quat = eul2quat(eul)
quat = eul2quat(eul,sequence)
```

Description

`quat = eul2quat(eul)` converts a given set of Euler angles, `eul`, to the corresponding quaternion, `quat`. The default order for Euler angle rotations is `"ZYX"`.

`quat = eul2quat(eul,sequence)` converts a set of Euler angles into a quaternion. The Euler angles are specified in the axis rotation sequence, `sequence`. The default order for Euler angle rotations is `"ZYX"`.

Examples

Convert Euler Angles to Quaternion

```
eul = [0 pi/2 0];
qZYX = eul2quat(eul)
```

```
qZYX = 1×4

    0.7071         0    0.7071         0
```

Convert Euler Angles to Quaternion Using Default ZYZ Axis Order

```
eul = [pi/2 0 0];
qZYZ = eul2quat(eul,'ZYZ')
```

```
qZYZ = 1×4

    0.7071         0         0    0.7071
```

Input Arguments

`eul` — Euler rotation angles

n-by-3 matrix

Euler rotation angles in radians, specified as an *n*-by-3 array of Euler rotation angles. Each row represents one Euler angle set.

Example: `[0 0 1.5708]`

sequence — Axis rotation sequence
`"ZYX"` (default) | `"ZYZ"` | `"XYZ"`

Axis rotation sequence for the Euler angles, specified as one of these string scalars:

- `"ZYX"` (default) – The order of rotation angles is *z*-axis, *y*-axis, *x*-axis.
- `"ZYZ"` – The order of rotation angles is *z*-axis, *y*-axis, *z*-axis.
- `"XYZ"` – The order of rotation angles is *x*-axis, *y*-axis, *z*-axis.

Data Types: `string` | `char`

Output Arguments

quat — Unit quaternion
n-by-4 matrix

Unit quaternion, returned as an *n*-by-4 matrix containing *n* quaternions. Each quaternion, one per row, is of the form $q = [w\ x\ y\ z]$, with *w* as the scalar number.

Example: `[0.7071 0.7071 0 0]`

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also
`quat2eul` | `quaternion`

Introduced in R2015a

eul2rotm

Convert Euler angles to rotation matrix

Syntax

```
rotm = eul2rotm(eul)
rotm = eul2rotm(eul,sequence)
```

Description

`rotm = eul2rotm(eul)` converts a set of Euler angles, `eul`, to the corresponding rotation matrix, `rotm`. When using the rotation matrix, premultiply it with the coordinates to be rotated (as opposed to postmultiplying). The default order for Euler angle rotations is `"ZYX"`.

`rotm = eul2rotm(eul,sequence)` converts Euler angles to a rotation matrix, `rotm`. The Euler angles are specified in the axis rotation sequence, `sequence`. The default order for Euler angle rotations is `"ZYX"`.

Examples

Convert Euler Angles to Rotation Matrix

```
eul = [0 pi/2 0];
rotmZYX = eul2rotm(eul)
```

```
rotmZYX = 3×3

    0.0000         0    1.0000
         0    1.0000         0
   -1.0000         0    0.0000
```

Convert Euler Angles to Rotation Matrix Using ZYZ Axis Order

```
eul = [0 pi/2 pi/2];
rotmZYZ = eul2rotm(eul,'ZYZ')
```

```
rotmZYZ = 3×3

    0.0000   -0.0000    1.0000
    1.0000    0.0000         0
   -0.0000    1.0000    0.0000
```

Input Arguments

`eul` — Euler rotation angles

n-by-3 matrix

Euler rotation angles in radians, specified as an *n*-by-3 array of Euler rotation angles. Each row represents one Euler angle set.

Example: `[0 0 1.5708]`

`sequence` — Axis rotation sequence

`"ZYX"` (default) | `"ZYZ"` | `"XYZ"`

Axis rotation sequence for the Euler angles, specified as one of these string scalars:

- `"ZYX"` (default) – The order of rotation angles is *z*-axis, *y*-axis, *x*-axis.
- `"ZYZ"` – The order of rotation angles is *z*-axis, *y*-axis, *z*-axis.
- `"XYZ"` – The order of rotation angles is *x*-axis, *y*-axis, *z*-axis.

Data Types: `string` | `char`

Output Arguments

`rotm` — Rotation matrix

3-by-3-by-*n* matrix

Rotation matrix, returned as a 3-by-3-by-*n* matrix containing *n* rotation matrices. Each rotation matrix has a size of 3-by-3 and is orthonormal. When using the rotation matrix, premultiply it with the coordinates to be rotated (as opposed to postmultiplying).

Example: `[0 0 1; 0 1 0; -1 0 0]`

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`rotm2eul`

Introduced in R2015a

eul2tform

Convert Euler angles to homogeneous transformation

Syntax

```
eul = eul2tform(eul)
tform = eul2tform(eul,sequence)
```

Description

`eul = eul2tform(eul)` converts a set of Euler angles, `eul`, into a homogeneous transformation matrix, `tform`. When using the transformation matrix, premultiply it with the coordinates to be transformed (as opposed to postmultiplying). The default order for Euler angle rotations is `"ZYX"`.

`tform = eul2tform(eul,sequence)` converts Euler angles to a homogeneous transformation. The Euler angles are specified in the axis rotation sequence, `sequence`. The default order for Euler angle rotations is `"ZYX"`.

Examples

Convert Euler Angles to Homogeneous Transformation Matrix

```
eul = [0 pi/2 0];
tformZYX = eul2tform(eul)
```

```
tformZYX = 4×4

    0.0000         0    1.0000         0
         0    1.0000         0         0
   -1.0000         0    0.0000         0
         0         0         0    1.0000
```

Convert Euler Angles to Homogeneous Transformation Matrix Using ZYZ Axis Order

```
eul = [0 pi/2 pi/2];
tformZYZ = eul2tform(eul,'ZYZ')
```

```
tformZYZ = 4×4

    0.0000   -0.0000    1.0000         0
    1.0000    0.0000         0         0
   -0.0000    1.0000    0.0000         0
         0         0         0    1.0000
```

Input Arguments

`eul` — Euler rotation angles

n-by-3 matrix

Euler rotation angles in radians, specified as an *n*-by-3 array of Euler rotation angles. Each row represents one Euler angle set.

Example: `[0 0 1.5708]`

`sequence` — Axis rotation sequence

`"ZYX"` (default) | `"ZYZ"` | `"XYZ"`

Axis rotation sequence for the Euler angles, specified as one of these string scalars:

- `"ZYX"` (default) – The order of rotation angles is *z*-axis, *y*-axis, *x*-axis.
- `"ZYZ"` – The order of rotation angles is *z*-axis, *y*-axis, *z*-axis.
- `"XYZ"` – The order of rotation angles is *x*-axis, *y*-axis, *z*-axis.

Data Types: `string` | `char`

Output Arguments

`tform` — Homogeneous transformation

4-by-4-by-*n* matrix

Homogeneous transformation matrix, specified by a 4-by-4-by-*n* matrix of *n* homogeneous transformations. When using the rotation matrix, premultiply it with the coordinates to be rotated (as opposed to postmultiplying).

Example: `[0 0 1 0; 0 1 0 0; -1 0 0 0; 0 0 0 1]`

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`tform2eul`

Introduced in R2015a

exportOccupancyMap3D

Import an octree file as 3D occupancy map

Syntax

`exportOccupancyMap3D(map3D,filename)`

Description

`exportOccupancyMap3D(map3D,filename)` serializes the 3D occupancy map, `map3D`, into either an octree or binary tree file (`.ot/bt`) specified at the file location and name, `filename`

Examples

Create and Export 3-D Occupancy Map

Create an `occupancyMap3D` object.

```
map3D = occupancyMap3D;
```

Create a ground plane and set occupancy values to 0.

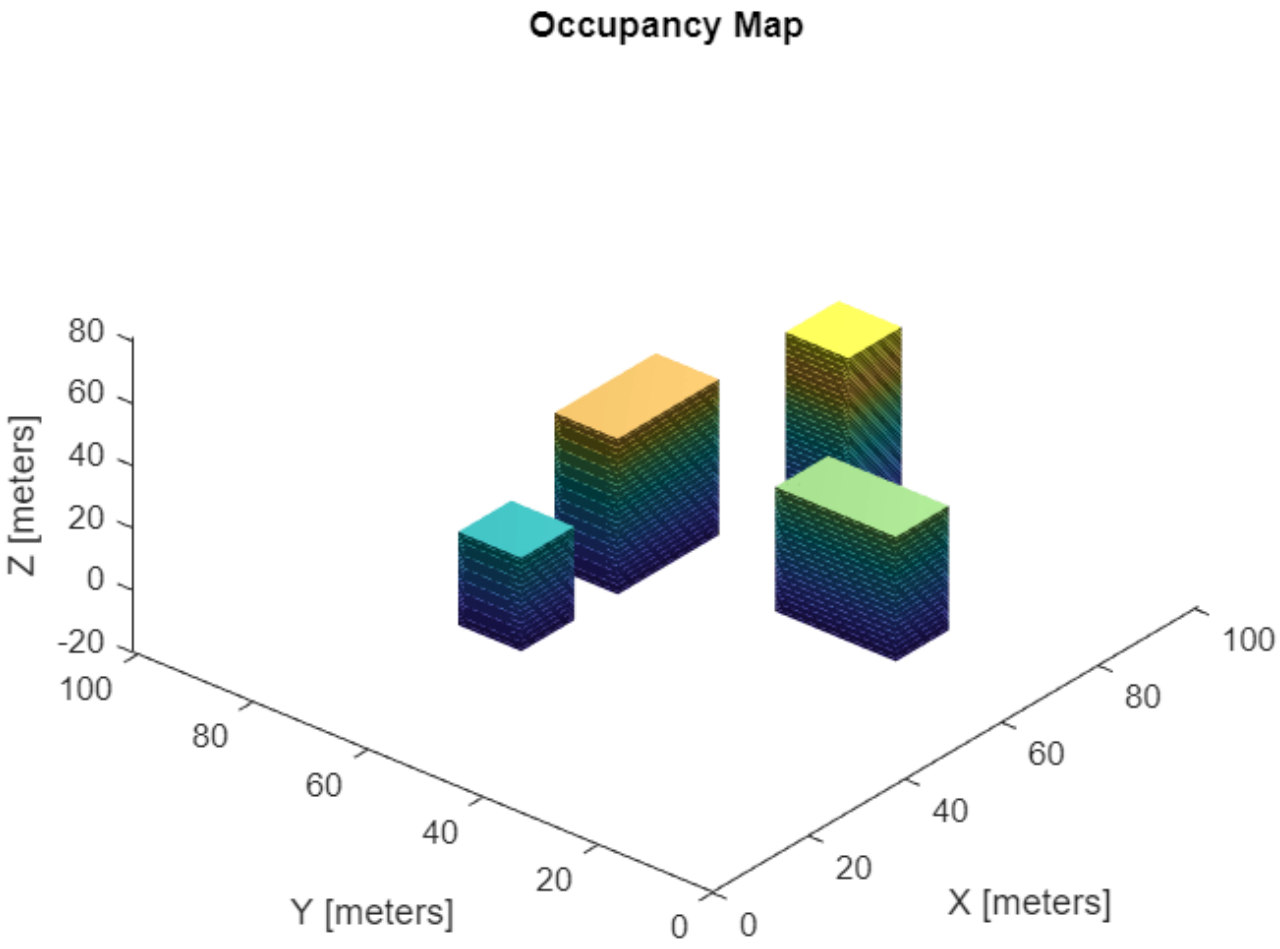
```
[xGround,yGround,zGround] = meshgrid(0:100,0:100,0);
xyzGround = [xGround(:) yGround(:) zGround(:)];
occval = 0;
setOccupancy(map3D,xyzGround,occval)
```

Create obstacles in specific world locations of the map.

```
[xBuilding1,yBuilding1,zBuilding1] = meshgrid(20:30,50:60,0:30);
[xBuilding2,yBuilding2,zBuilding2] = meshgrid(50:60,10:30,0:40);
[xBuilding3,yBuilding3,zBuilding3] = meshgrid(40:60,50:60,0:50);
[xBuilding4,yBuilding4,zBuilding4] = meshgrid(70:80,35:45,0:60);

xyzBuildings = [xBuilding1(:) yBuilding1(:) zBuilding1(:);...
                xBuilding2(:) yBuilding2(:) zBuilding2(:);...
                xBuilding3(:) yBuilding3(:) zBuilding3(:);...
                xBuilding4(:) yBuilding4(:) zBuilding4(:)];
```

Update the obstacles with new probability values and display the map.

```
obs = 0.65;
updateOccupancy(map3D,xyzBuildings,obs)
show(map3D)
```



Check if the map file named `citymap.ot` already exist in the current directory and delete it before creating the map file.

```
if exist("citymap.ot",'file')
    delete("citymap.ot")
end
```

Export the map as an octree file.

```
filePath = fullfile(pwd,"citymap.ot");
exportOccupancyMap3D(map3D,filePath)
```

Input Arguments

`map3D` — 3-D occupancy map
`occupancyMap3D` object

3-D occupancy map, specified as a `occupancyMap3D` object.

`filename` — Absolute or relative path to octree file
string scalar | character vector

Absolute or relative path to octree file (`.ot/bt`), specified as a string scalar or character vector.

Example: `"path/to/file/map.ot"`

Data Types: `char` | `string`

See Also

Classes
`occupancyMap3D` | `lidarSLAM` | `occupancyMap`

Functions
`insertPointCloud` | `inflate` | `setOccupancy` | `show`

Introduced in R2020a

extractNMEASentence

Verify and extract NMEA sentence data into string array

Syntax

```
[isValid,splitString] = extractNMEASentence(unparsedData,'MessageID')
```

Description

`[isValid,splitString] = extractNMEASentence(unparsedData,'MessageID')` verifies the checksum of an unparsed NMEA sentence, identified using its Message ID, and extracts the NMEA fields from NMEA sentence data into a string array, `splitString`.

Examples

Use extractNMEASentence to Obtain NMEA Fields of NMEA Sentence

Suppose that the Message ID of the NMEA sentence from which you need to extract data as split strings is `GLL`. You provide an unparsed GLL sentence as input, and specify the Message ID to extract the NMEA sentence into string array.

```
unparsedData = ['$GPGLL,1300.26049,N,07733.81639,E,074549.00,A,A*6C'];
[isValid, splitString] = extractNMEASentence(unparsedData,'GLL')
```

```
isValid =

  logical

   1


splitString =

  1×9 string array

    "GP"    "GLL"    "1300.26049"    "N"    "07733.81639"    "E"    "074549.00"    "A"    "A"
```

Input Arguments

`unparsedData` — Unparsed NMEA data from the device

string | character array

The unparsed NMEA data as obtained from the device.

`MessageID` — Message ID of the unparsed NMEA sentence

string | character array

The Message ID to identify the unparsed NMEA sentence.

Output Arguments

`isValid` — Validity of unparsed NMEA sentence based on checksum

`0` | `1`

Determine the validity of unparsed NMEA sentence based on checksum. A value of `1` indicates that the checksum is valid. A value of `0` indicates that the checksum is invalid; however, the fields of NMEA sentence appear in the `splitString` output if the specified MessageID is matching.

Data Types: `logical`

`splitString` — Output data as split strings

string array

Output data as split strings based on the structure that you defined. If the specified MessageID is not found in the NMEA sentence, the function returns an empty `splitString`.

Data Types: `string`

See Also

Objects

`nmeaParser`

Introduced in R2021b

flush

Flush all GPS data accumulated in the buffers and reset properties

Syntax

```
flush(gps)
```

Description

`flush(gps)` clears the buffers and resets the values of `SamplesRead` and `SamplesAvailable` properties.

Examples

Read Data from GPS Receiver as Matrix

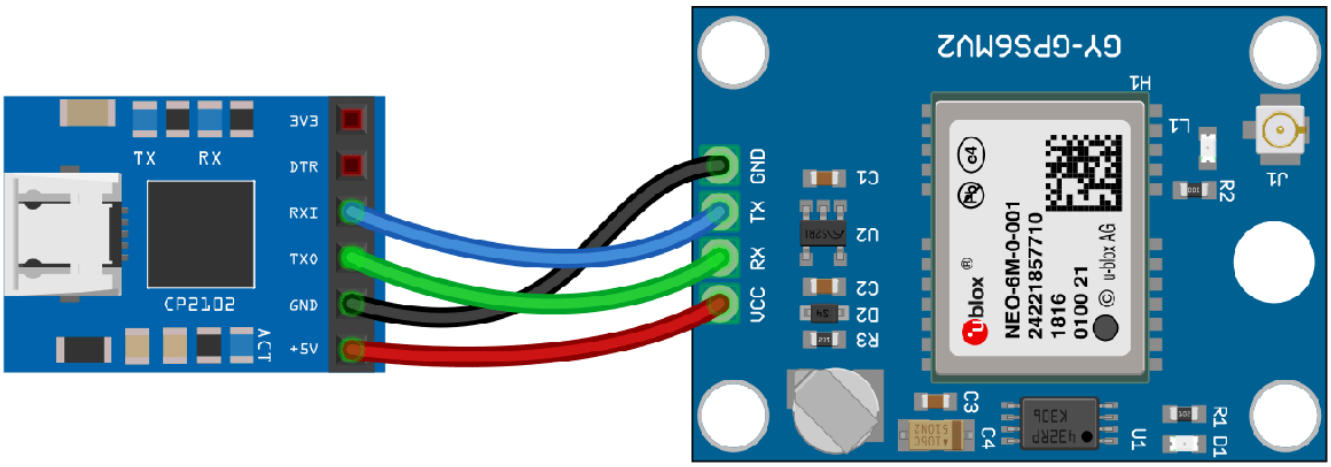
Read data from the GPS receiver connected to the host computer using `serialport` object.

Required Hardware

To run this example, you need:

- UBlox Neo-6M GPS module
- GPS antenna
- USB to UART module
- USB cable
- Connecting wires

Hardware Connection



Connect the pins on the UBlox Neo-6M GPS module to the pins on your USB to UART module. The connections are:

- VCC - +5V
- RX - TXO
- TX - RXI
- GND - GND

Connect the GPS antenna to the GPS module. Connect the USB to UART module to the host computer with a USB cable. GPS Fix can be easily acquired in locations that have a clear view of the sky. Wait for the GPS module to acquire satelite signals (Fix).This can be verified by checking the Fix LED (D1) of your GPS module.

Create GPS Object

Connect to the GPS receiver using `serialport` object. Specify the port name and the baud rate. Specify the output format of the data as matrix.

```
s = serialport('COM4',9600);
gps = gpsdev(s,'OutputFormat',"matrix")
```

```
gps =
  gpsdev with properties:

                        SerialPort: COM4
                          BaudRate: 9600 (bits/s)

                    SamplesPerRead: 1
                          ReadMode: "latest"
                       SamplesRead: 0
Show all properties all functions
```

Read the GPS data

Read the GPS data and return them as matrices.

```
[lla,speed,course,dops,gpsReceiverTime,timestamp,overruns] = read(gps)
```

```
lla = 1×3

   NaN   NaN   NaN


speed = NaN

course = NaN

dops = 1×3

   NaN   NaN   NaN


gpsReceiverTime = datetime
   NaT
```

```
timestamp = datetime
   22-Mar-2021 03:41:00.274


overruns = 1
```

Display number of samples read and the samples available in the host buffer.

```
gps.SamplesRead
```

```
ans = 1
```

```
gps.SamplesAvailable
```

```
ans = 0
```

Flush all GPS data accumulated in the buffers and reset the `SamplesRead` and `SamplesAvailable` properties.

```
flush(gps)
```

Display number of samples read and the samples available in the host buffer.

```
gps.SamplesRead
```

```
ans = 0
```

```
gps.SamplesAvailable
```

```
ans = 0
```

Clean Up

When the connection is no longer needed, clear the associated object.

```
delete(gps);
clear gps;
clear s;
```

Input Arguments

gps — GPS sensor

`gpsdev` object

The GPS sensor, specified as a `gpsdev` object.

See Also

Objects

`gpsdev`

Functions

`release` | `writeBytes` | `read` | `info`

Introduced in R2020b

gnssconstellation

Satellite locations at specified time

Syntax

```
[satPos,satVel] = gnssconstellation(t)
```

Description

`[satPos,satVel] = gnssconstellation(t)` returns the satellite positions and velocities at the `datetime` `t`. The function returns positions and velocities in the Earth-centered Earth-fixed (ECEF) coordinate system in meters and meters per second, respectively.

Examples

Get Current Satellite Positions and Velocities

Get the current satellite positions and velocities from the GNSS satellites. Access the orbital parameters from IS-GPS-200K Interface Specification and calculate the position and velocities in ECEF coordinates for the given time. Display the satellite positions.

```
t = datetime('now','TimeZone','Local');
[satPos,satVel] = gnssconstellation(t);
disp(satPos)
```

```
   1.0e+07 *

   -1.6906   -1.7108   -1.1267
   -1.2607    1.0209    2.1030
    2.1551   -0.1458   -1.5454
    2.4761    0.7749   -0.5679
   -0.7964    2.1512    1.3388
   -0.0748    2.6309    0.3560
   -1.4399   -1.0877    1.9488
    1.4025   -0.8698   -2.0810
   -0.9667   -2.1791    1.1709
   -1.9616   -0.1872    1.7808
   -0.9558    1.9396   -1.5422
    0.3405    1.5603   -2.1222
    2.2680   -1.3710   -0.1752
    0.1747   -1.5765    2.1304
    1.1818    1.7972   -1.5581
    1.9830    1.5849   -0.7811
   -2.4626   -0.5814   -0.8073
    1.4616   -0.8721    2.0389
    1.1102    1.9685    1.3953
   -1.3122    1.1845   -1.9822
    0.0670   -2.6326    0.3452
   -1.5302   -0.2110   -2.1606
   -2.2344    1.4339   -0.0740
    0.5120   -1.8714   -1.8138
```

```
   -0.5692   -1.4383   -2.1591
    2.1015   -0.0390    1.6236
    1.2738    0.9431    2.1313
```

Get Satellite Look Angles for Receiver Position

Use the `lookangles` function to get the azimuth and elevation angles of satellites for given satellite and receiver positions. Specify a mask angle of 5 degrees. Get the satellite positions using the `gnssconstellation` function.

Specify a receiver position in geodetic coordinates (latitude, longitude, altitude).

```
recPos = [42 -71 50];
```

Get the satellite positions for the current time.

```
t = datetime('now');
gpsSatPos = gnssconstellation(t);
```

Specify a mask angle of 5 degrees.

```
maskAngle = 5;
```

Get the azimuth and elevation look angles for the satellite positions. The `vis` output indicates which satellites are visible. Get the total using `nnz`.

```
[az,el,vis] = lookangles(recPos,gpsSatPos,maskAngle);
fprintf('%d satellites visible at %s.\n',nnz(vis),t);
```

```
8 satellites visible at 01-Sep-2021 18:18:49.
```

Input Arguments

`t` — Current time for satellite simulation

scalar `datetime` array

Current time for the satellite simulation, specified as a scalar `datetime` array.

The default time zone for a `datetime` array is UTC. For information on specifying a different time zone, see `datetime`.

GPS start time is Jan 6, 1980 midnight, UTC. Specifying any `datetime` prior to this time will use the GPS start time.

Example: `datetime('now','TimeZone','Local');`

Data Types: `datetime`

Output Arguments

`satPos` — Satellite positions

N-by-3 matrix of scalars

Satellite positions in the Earth-centered Earth-fixed (ECEF) coordinate system in meters, returned as an *N*-by-3 matrix of scalars. *N* is the number of satellites in the constellation.

Data Types: `single` | `double`

`satVel` — Satellite velocities
N-by-3 matrix of scalar

Satellite velocities in the Earth-centered Earth-fixed (ECEF) coordinate system in meters per second, returned as an *N*-by-3 matrix of scalars. *N* is the number of satellites in the constellation.

Data Types: `single` | `double`

More About

Orbital Parameters

The initial satellite positions and velocities are defined by orbital parameters in Table A.2-2 in GPS SPS Performance Standard, and are given in Earth-centered Earth-fixed (ECEF) coordinates.

Position calculations use equations from Table 30-II in the same IS-GPS-200K Interface Specification.

Velocity calculations use equations 8.21–8.27 in [1].

References

[1] Groves, Paul D. *Principles of GNSS, Inertial, and Multisensor Integrated Navigation Systems*. Boston: Artech House, 2013.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

Only MEX functions are supported for code generation.

See Also

Objects
`gnssSensor` | `gpsSensor` | `imuSensor`

Functions
`skyplot` | `lookangles` | `pseudoranges` | `receiverposition`

Topics
“Estimate GNSS Receiver Position with Simulated Satellite Constellations”

Introduced in R2021a

hom2cart

Convert homogeneous coordinates to Cartesian coordinates

Syntax

```
cart = hom2cart(hom)
```

Description

`cart = hom2cart(hom)` converts a set of homogeneous points to Cartesian coordinates.

Examples

Convert Homogeneous Points to 3-D Cartesian Points

```
h = [0.2785 0.9575 0.1576 0.5; 0.5469 0.9649 0.9706 0.5];
c = hom2cart(h)
```

```
c = 2×3

    0.5570    1.9150    0.3152
    1.0938    1.9298    1.9412
```

Input Arguments

`hom` — Homogeneous points

n-by-*k* matrix

Homogeneous points, specified as an *n*-by-*k* matrix, containing *n* points. *k* must be greater than or equal to 2.

Example: `[0.2785 0.9575 0.1576 0.5; 0.5469 0.9649 0.9706 0.5]`

Output Arguments

`cart` — Cartesian coordinates

n-by-(*k*–1) matrix

Cartesian coordinates, returned as an *n*-by-(*k*–1) matrix, containing *n* points. Each row of `cart` represents a point in (*k*–1)-dimensional space. *k* must be greater than or equal to 2.

Example: `[0.8147 0.1270 0.6324; 0.9058 0.9134 0.0975]`

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`cart2hom`

Introduced in R2015a

importOccupancyMap3D

Import an octree file as 3D occupancy map

Syntax

```
map3D = importOccupancyMap3D(mapPath)
```

Description

`map3D = importOccupancyMap3D(mapPath)` imports the octree file (`.ot/bt`) specified at the relative or absolute file path, `mapPath`

Examples

Check Occupancy Status and Get Occupancy Values in 3-D Occupancy Map

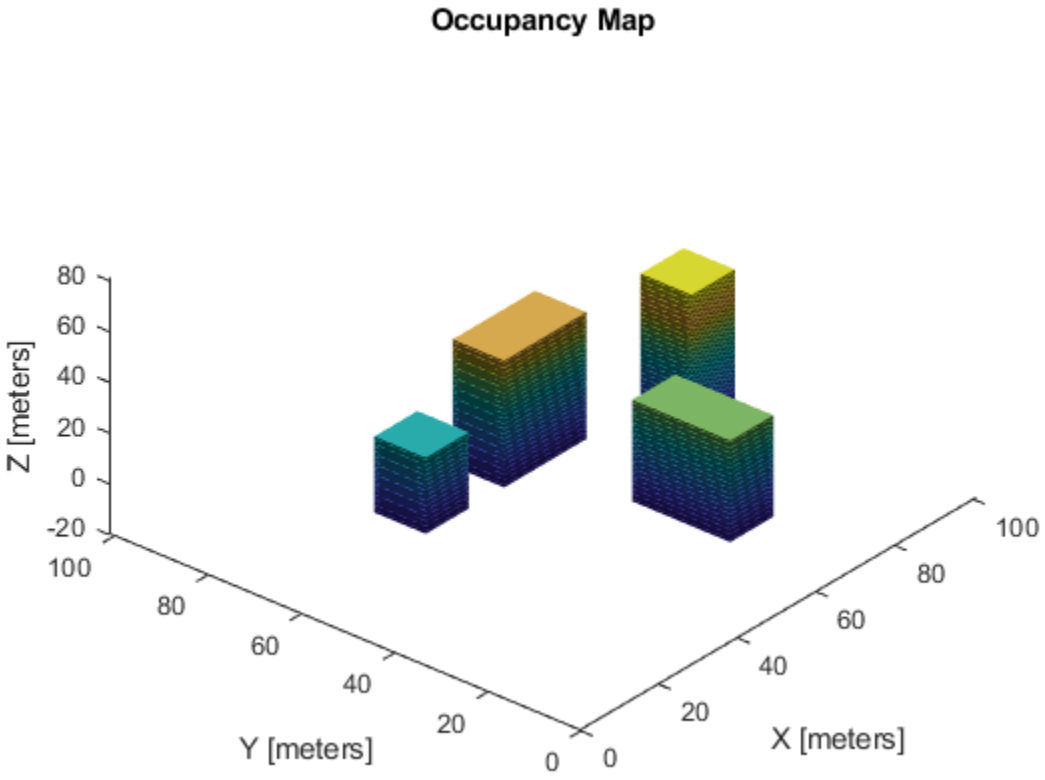
Import a 3-D occupancy map.

```
map3D = importOccupancyMap3D("citymap.ot")
```

```
map3D =
  occupancyMap3D with properties:

    ProbabilitySaturation: [1.0000e-03 0.9990]
               Resolution: 1
        OccupiedThreshold: 0.6500
            FreeThreshold: 0.2000
```

Display the map.

```
show(map3D)
```



Check the occupancy statuses of different locations and get their occupancy values.

```
iOccVal1 = checkOccupancy(map3D,[50 15 0])
```

```
iOccVal1 = 0
```

```
OccVal1 = getOccupancy(map3D,[50 15 0])
```

```
OccVal1 = 0.0019
```

```
iOccVal2 = checkOccupancy(map3D,[50 15 15])
```

```
iOccVal2 = 1
```

```
OccVal2 = getOccupancy(map3D,[50 15 15])
```

```
OccVal2 = 0.6500
```

```
iOccVal3 = checkOccupancy(map3D,[50 15 45])
```

```
iOccVal3 = -1
```

```
OccVal3 = getOccupancy(map3D,[50 15 45])
```

```
OccVal3 = 0.5000
```

Input Arguments

`mapPath` — Absolute or relative path to octree file

string scalar | character vector

Absolute or relative path to octree file (`.ot/bt`), specified as a string scalar or character vector.

Example: `"path/to/file/map.ot"`

Data Types: `char` | `string`

Output Arguments

`map3D` — 3-D occupancy map

`occupancyMap3D` object

3-D occupancy map, specified as a `occupancyMap3D` object.

See Also

Classes

`occupancyMap3D` | `lidarSLAM` | `occupancyMap`

Functions

`insertPointCloud` | `inflate` | `setOccupancy` | `show`

Introduced in R2020a

info

Read update rate, GPS lock information and number of satellites in view for the GPS receiver

Syntax

```
gpsInfo = info(gps)
```

Description

`gpsInfo = info(gps)` returns the update rate of the GPS receiver, GPS lock information and number of satellites from which the GPS can read signals. `info` gets updated after every execution of `read` command.

Examples

Read Information from GPS Receiver

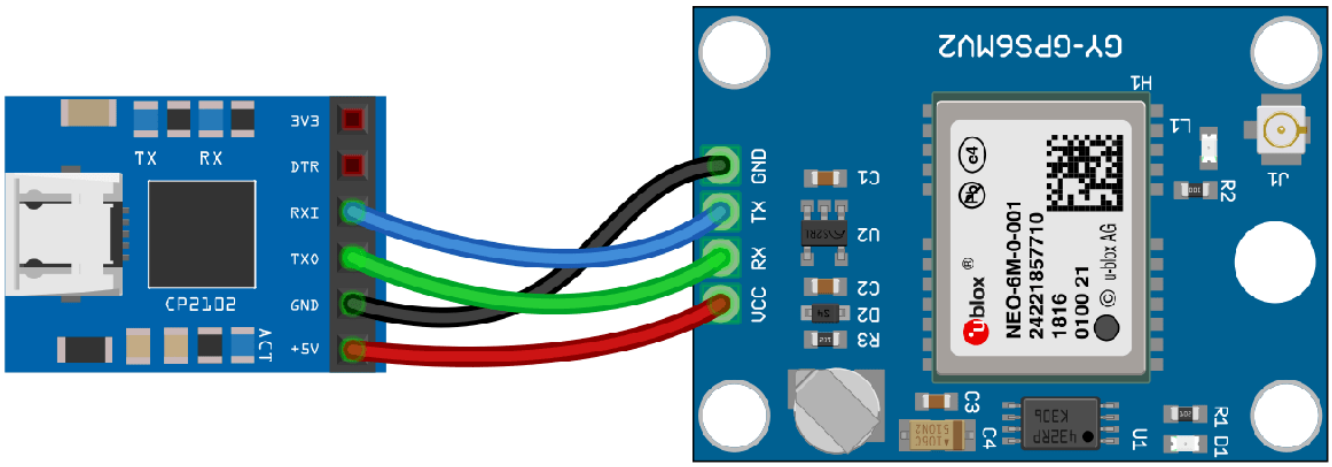
Read information from the GPS receiver connected to the host computer on a specific serial port.

Required Hardware

To run this example, you need:

- UBlox Neo-6M GPS module
- GPS antenna
- USB to UART module
- USB cable
- Connecting wires

Hardware Connection



Connect the pins on the UBlox Neo-6M GPS module to the pins on your USB to UART module. The connections are:

- VCC - +5V
- RX - TXO
- TX - RXI
- GND - GND

Connect the GPS antenna to the GPS module. Connect the USB to UART module to the host computer with a USB cable. GPS Fix can be easily acquired in locations that have a clear view of the sky. Wait for the GPS module to acquire satelite signals (Fix).This can be verified by checking the Fix LED (D1) of your GPS module.

Create GPS Object

Create a `gpsdev` object for the GPS receiver connected to a specific port.

```
gps = gpsdev('COM4')
```

```
gps =
  gpsdev with properties:

                        SerialPort: COM4
                          BaudRate: 9600 (bits/s)

                    SamplesPerRead: 1
                          ReadMode: "latest"
                       SamplesRead: 0
Show all properties all functions
```

Read the GPS Module Information

Read the GPS module information and return them as a structure.

```
gpsInfo = info(gps)
```

```
gpsInfo = struct with fields:
          UpdateRate: []
           GPSLocked: 0
    SatellitesInView: 0
```

Clean Up

When the connection is no longer needed, clear the associated object.

```
delete(gps);
clear gps;
```

Input Arguments

gps — GPS sensor

`gpsdev` object

The GPS sensor, specified as a `gpsdev` object.

Output Arguments

`gpsInfo` — GPS module information

structure

GPS module information such as update rate, and number of satellites. The output has three fields:

- `UpdateRate` — Update Rate of the GPS Module in Hz. Update Rate of GPS receiver is estimated from the difference in time at which two RMC sentences are obtained. This value might be slightly varying from actual Update Rate of the module.
- `GPSLocked` — This property specifies if GPS has enough information to get valid data. GPS signals are acquired easily in locations that have a clear view of the sky. It can be either a `0` or `1` (logical). If `GPSLocked` is `0`, the GPS does not have the lock to compute location or time information. If `GPSLocked` is `1`, GPS module has enough data to compute location or time information.
- `NumberOfSatellitesInView` — Number of satellites from which the GPS module can read the signals.

Data Types: `struct`

See Also

Objects

`gpsdev`

Functions

`flush` | `release` | `read` | `writeBytes`

Introduced in R2020b

insfilter

Create inertial navigation filter

Syntax

```
filter = insfilter
filter = insfilter('ReferenceFrame',RF)
```

Description

`filter = insfilter` returns an `insfilterMARG` inertial navigation filter object that estimates pose based on accelerometer, gyroscope, GPS, and magnetometer measurements. See `insfilterMARG` for more details.

`filter = insfilter('ReferenceFrame',RF)` returns an `insfilterMARG` inertial navigation filter object that estimates pose relative to a reference frame specified by `RF`. Specify `RF` as `'NED'` (North-East-Down) or `'ENU'` (East-North-Up). The default value is `'NED'`. See `insfilterMARG` for more details.

Examples

Create Default INS Filter

The default INS filter is the `insfilterMARG` object. Call `insfilter` with no input arguments to create the default INS filter.

```
filter = insfilter
```

```
filter =
  insfilterMARG with properties:

        IMUSampleRate: 100              Hz
    ReferenceLocation: [0 0 0]          [deg deg m]
                State: [22x1 double]
      StateCovariance: [22x22 double]

   Multiplicative Process Noise Variances
            GyroscopeNoise: [1e-09 1e-09 1e-09]       (rad/s)²
        AccelerometerNoise: [0.0001 0.0001 0.0001]    (m/s²)²
        GyroscopeBiasNoise: [1e-10 1e-10 1e-10]       (rad/s)²
    AccelerometerBiasNoise: [0.0001 0.0001 0.0001]    (m/s²)²

   Additive Process Noise Variances
    GeomagneticVectorNoise: [1e-06 1e-06 1e-06]    uT²
     MagnetometerBiasNoise: [0.1 0.1 0.1]          uT²
```

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`imufilter` | `ahrsfilter` | `insfilterErrorState` | `insfilterAsync` | `insfilterNonholonomic` | `insfilterMARG`

Introduced in R2018b

interpolate

Interpolate poses along path segment

Syntax

```
poses = interpolate(pathSeg)
poses = interpolate(pathSeg,lengths)
[poses,directions] = interpolate( ___ )
```

Description

`poses = interpolate(pathSeg)` interpolates along the path segment at the transitions between motion types.

`poses = interpolate(pathSeg,lengths)` interpolates along the path segment at the specified lengths along the path. Transitions between motion types are always included.

`[poses,directions] = interpolate( ___ )` also returns the direction of motion along the path for each section as a vector of 1s (forward) and –1s (reverse) using the previous inputs.

Examples

Interpolate Poses For Dubins Path

Create a `dubinsConnection` object.

```
dubConnObj = dubinsConnection;
```

Define start and goal poses as `[x y theta]` vectors.

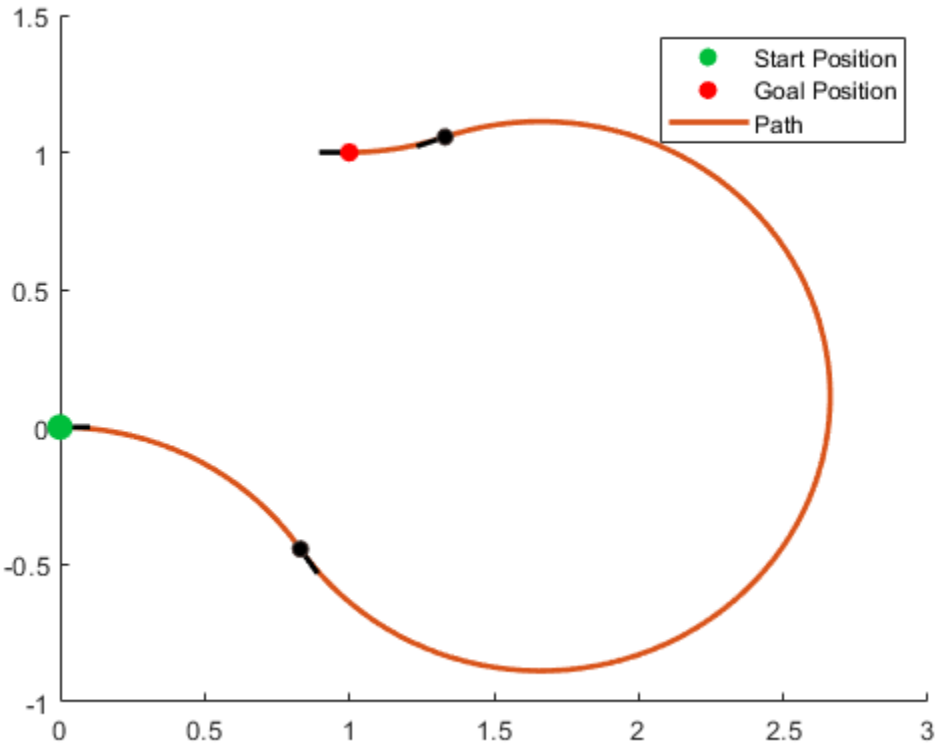
```
startPose = [0 0 0];
goalPose = [1 1 pi];
```

Calculate a valid path segment to connect the poses.

```
[pathSegObj,pathCosts] = connect(dubConnObj,startPose,goalPose);
```

Show the generated path.

```
show(pathSegObj{1})
```



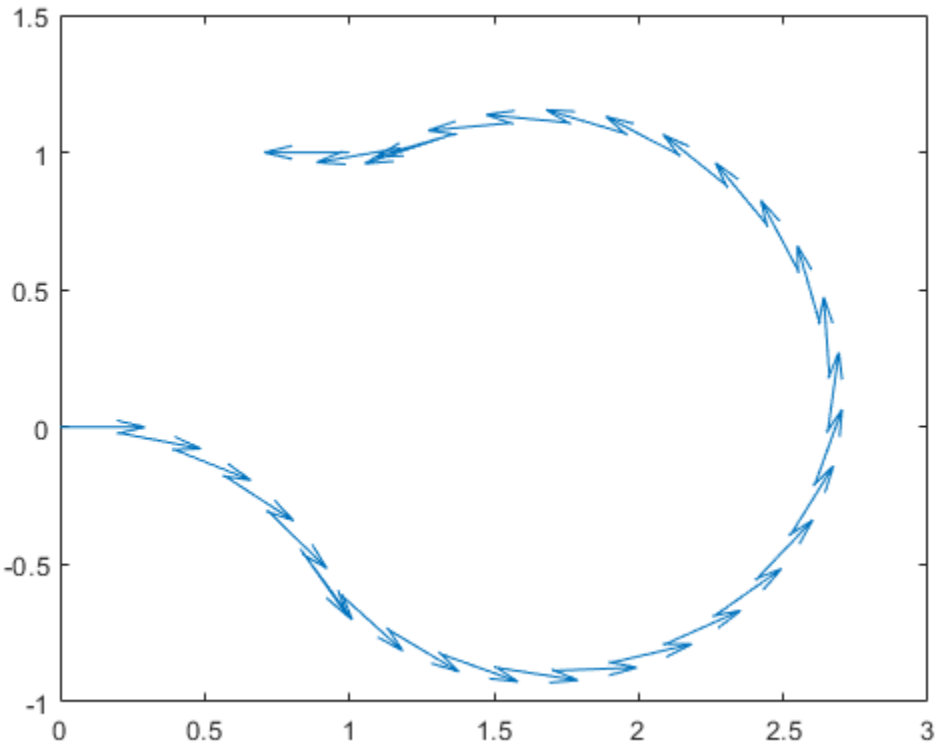
Interpolate poses along the path. Get a pose every 0.2 meters, including the transitions between turns.

```
length = pathSegObj{1}.Length;
poses = interpolate(pathSegObj{1},0:0.2:length)
```

```
poses = 32×3

         0         0         0
    0.1987   -0.0199    6.0832
    0.3894   -0.0789    5.8832
    0.5646   -0.1747    5.6832
    0.7174   -0.3033    5.4832
    0.8309   -0.4436    5.3024
    0.8418   -0.4595    5.3216
    0.9718   -0.6110    5.5216
    1.1293   -0.7337    5.7216
    1.3081   -0.8226    5.9216
       ⋮
```

Use the `quiver` function to plot these poses.

```
quiver(poses(:,1),poses(:,2),cos(poses(:,3)),sin(poses(:,3)),0.5)
```



Input Arguments

pathSeg — Path segment

`dubinsPathSegment` object | `reedsSheppPathSegment` object

Path segment, specified as a `dubinsPathSegment` or `reedsSheppPathSegment` object.

lengths — Lengths along path to interpolate at

positive numeric vector

Lengths along path to interpolate at, specified as a positive numeric vector. For example, specify `[0:0.1:pathSegObj{1}.Length]` to interpolate at every 0.1 meters along the path. Transitions between motion types are always included.

Output Arguments

poses — Interpolated poses

$[x, y, \Theta]$ matrix

This property is read-only.

Interpolated poses along the path segment, specified as an $[x, y, \Theta]$ matrix. Each row of the matrix corresponds to a different interpolated pose along the path.

x and *y* are in meters. Θ is in radians.

`directions` — Directions of motion
vector of 1s (forward) and –1s (reverse)

Directions of motion for each segment of the interpolated path, specified as a vector of 1s (forward) and –1s (reverse).

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

Functions
`connect` | `show`

Objects
`dubinsConnection` | `dubinsPathSegment` | `reedsSheppConnection` | `reedsSheppPathSegment`

Introduced in R2019b

lla2enu

Transform geodetic coordinates to local east-north-up coordinates

Syntax

```
xyzENU = lla2enu(lla,lla0,method)
```

Description

`xyzENU = lla2enu(lla,lla0,method)` transforms the geodetic coordinates `lla` to local east-north-up (ENU) Cartesian coordinates `xyzENU`. Specify the origin of the local ENU system as the geodetic coordinates `lla0`.

Note

- The latitude and longitude values in the geodetic coordinate system use the World Geodetic System of 1984 (WGS84) standard.
- Specify altitude as height in meters above the WGS84 reference ellipsoid.

Examples

Transform Geodetic Coordinates to ENU Coordinates

Specify the geodetic coordinates of the local origin in Zermatt, Switzerland.

```
lla0 = [46.017 7.750 1673]; % [lat0 lon0 alt0]
```

Specify the geodetic coordinates of a point of interest. In this case, the point of interest is the Matterhorn.

```
lla = [45.976 7.658 4531]; % [lat lon alt]
```

Transform the geodetic coordinates to local ENU coordinates using flat earth approximation.

```
xyzENU = lla2enu(lla,lla0,'flat')
```

```
xyzENU = 1×3
10^3 ×

   -7.1244   -4.5572    2.8580
```

Input Arguments

`lla` — Geodetic coordinates

three-element row vector | *n*-by-3 matrix

Geodetic coordinates, specified as a three-element row vector or an *n*-by-3 matrix. *n* is the number of points to transform. Specify each point in the form `[lat lon alt]`. *lat* and *lon* specify the latitude and longitude respectively in degrees. *alt* specifies the altitude in meters.

Data Types: `double`

`lla0` — Origin of local ENU system in geodetic coordinates
three-element row vector | *n*-by-3 matrix

Origin of the local ENU system in the geodetic coordinates, specified as a three-element row vector or an *n*-by-3 matrix. *n* is the number of origin points. Specify each point in the form `[lat0 lon0 alt0]`. *lat0* and *lon0* specify the latitude and longitude of the origin, respectively, in degrees. *alt0* specifies the altitude of the origin in meters.

Data Types: `double`

`method` — Transformation method
`'flat'` | `'ellipsoid'`

Transformation method, specified as `'flat'` or `'ellipsoid'`. This argument specifies whether the function assumes the planet is flat or ellipsoidal.

The flat Earth transformation method has these limitations:

- Assumes that the flight path and bank angle are zero.
- Assumes that the flat Earth *z*-axis is normal to the Earth at only the initial geodetic latitude and longitude. This method has higher accuracy over small distances from the initial geodetic latitude and longitude, and closer to the equator. The method calculates a longitude with higher accuracy when the variation in latitude is smaller.
- Latitude values of +90 and -90 may return unexpected values because of singularity at the poles.

Data Types: `char` | `string`

Output Arguments

`xyzENU` — Local ENU Cartesian coordinates
three-element row vector | *n*-by-3 matrix

Local ENU Cartesian coordinates, returned as a three-element row vector or an *n*-by-3 matrix. *n* is the number of transformed points. Each point is in the form `[xEast yNorth zUp]`. *xEast*, *yNorth*, and *zUp* are the respective *x*-, *y*-, and *z*-coordinates, in meters, of the point in the local ENU system.

Data Types: `double`

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

Functions
`enu2lla` | `lla2ned` | `ned2lla`

Introduced in R2021a

lla2ned

Transform geodetic coordinates to local north-east-down coordinates

Syntax

```
xyzNED = lla2ned(lla,lla0,method)
```

Description

`xyzNED = lla2ned(lla,lla0,method)` transforms the geodetic coordinates `lla` to local north-east-down (NED) Cartesian coordinates `xyzNED`. Specify the origin of the local NED system as the geodetic coordinates `lla0`.

Note

- The latitude and longitude values in the geodetic coordinate system use the World Geodetic System of 1984 (WGS84) standard.
- Specify altitude as height in meters above the WGS84 reference ellipsoid.

Examples

Transform Geodetic Coordinates to NED Coordinates

Specify the geodetic coordinates of the local origin in Zermatt, Switzerland.

```
lla0 = [46.017 7.750 1673]; % [lat0 lon0 alt0]
```

Specify the geodetic coordinates of a point of interest. In this case, the point of interest is the Matterhorn.

```
lla = [45.976 7.658 4531]; % [lat lon alt]
```

Transform the geodetic coordinates to local NED coordinates using flat earth approximation.

```
xyzNED = lla2ned(lla,lla0,'flat')
```

```
xyzNED = 1×3
10^3 ×

   -4.5572   -7.1244   -2.8580
```

Input Arguments

`lla` — Geodetic coordinates

three-element row vector | *n*-by-3 matrix

Geodetic coordinates, specified as a three-element row vector or an *n*-by-3 matrix. *n* is the number of points to transform. Specify each point in the form `[lat lon alt]`. *lat* and *lon* specify the latitude and longitude respectively in degrees. *alt* specifies the altitude in meters.

Data Types: `double`

`lla0` — Origin of local NED system in geodetic coordinates

three-element row vector | *n*-by-3 matrix

Origin of the local NED system with the geodetic coordinates, specified as a three-element row vector or an *n*-by-3 matrix. *n* is the number of origin points. Specify each point in the form `[lat0 lon0 alt0]`. *lat0* and *lon0* specify the latitude and longitude respectively in degrees. *alt0* specifies the altitude in meters.

Data Types: `double`

`method` — Transformation method

`'flat'` | `'ellipsoid'`

Transformation method, specified as `'flat'` or `'ellipsoid'`. This argument specifies whether the function assumes the planet is flat or ellipsoidal.

The flat Earth transformation method has these limitations:

- Assumes that the flight path and bank angle are zero.
- Assumes that the flat Earth *z*-axis is normal to the Earth at only the initial geodetic latitude and longitude. This method has higher accuracy over small distances from the initial geodetic latitude and longitude, and closer to the equator. The method calculates a longitude with higher accuracy when the variation in latitude is smaller.
- Latitude values of +90 and -90 may return unexpected values because of singularity at the poles.

Data Types: `char` | `string`

Output Arguments

`xyzNED` — Local NED Cartesian coordinates

three-element row vector | *n*-by-3 matrix

Local NED Cartesian coordinates, returned as a three-element row vector or an *n*-by-3 matrix. *n* is the number of transformed points. Each point is in the form `[xNorth yEast zDown]`. *xNorth*, *yEast*, and *zDown* are the respective *x*-, *y*-, and *z*-coordinates, in meters, of the point in the local NED system.

Data Types: `double`

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

Functions

`enu2lla` | `lla2enu` | `ned2lla`

Introduced in R2021a

lookangles

Satellite look angles from receiver and satellite positions

Syntax

```
[az,el,vis] = lookangles(recPos,satPos)
[az,el,vis] = lookangles(recPos,satPos,maskAngle)
```

Description

`[az,el,vis] = lookangles(recPos,satPos)` returns the look angles and visibilities of satellite positions for a given receiver position. The azimuth `az` and elevation `el` are the look angles in degrees in the Earth-centered Earth-fixed (ECEF) coordinate system. The visibility of the satellites `vis` is a logical array that the function calculates using the default receiver mask angle of 10 degrees.

`[az,el,vis] = lookangles(recPos,satPos,maskAngle)` returns the look angles and visibilities of satellites with a specified mask angle `maskAngle` in degrees.

Examples

Get Satellite Look Angles for Receiver Position

Use the `lookangles` function to get the azimuth and elevation angles of satellites for given satellite and receiver positions. Specify a mask angle of 5 degrees. Get the satellite positions using the `gnssconstellation` function.

Specify a receiver position in geodetic coordinates (latitude, longitude, altitude).

```
recPos = [42 -71 50];
```

Get the satellite positions for the current time.

```
t = datetime('now');
gpsSatPos = gnssconstellation(t);
```

Specify a mask angle of 5 degrees.

```
maskAngle = 5;
```

Get the azimuth and elevation look angles for the satellite positions. The `vis` output indicates which satellites are visible. Get the total using `nnz`.

```
[az,el,vis] = lookangles(recPos,gpsSatPos,maskAngle);
fprintf('%d satellites visible at %s.\n',nnz(vis),t);
```

```
8 satellites visible at 01-Sep-2021 18:18:49.
```

Input Arguments

`recPos` — Receiver position

three-element vector of the form `[lat lon alt]`

Receiver position in geodetic coordinates, specified as a three-element vector of the form *[latitude longitude altitude]*

Data Types: `single` | `double`

`satPos` — Satellite positions

N-by-3 matrix of scalars

Satellite positions in the Earth-centered Earth-fixed (ECEF) coordinate system in meters, specified as an *N*-by-3 matrix of scalars. *N* is the number of satellites in the constellation.

Data Types: `single` | `double`

`maskAngle` — Elevation mask angle

positive scalar

Elevation mask angle of the receiver, specified as a positive scalar in degrees.

Data Types: `single` | `double`

Output Arguments

`az` — Azimuth angles for visible satellite positions

n-element vector of angles

Azimuth angles for visible satellite positions, returned as an *n*-element vector of angles. *n* is the number of visible satellite positions in the plot. Azimuth angles are measured in degrees, clockwise-positive from the north direction looking down.

Example: `[25 45 182 356]`

Data Types: `double`

`el` — Elevation angles for visible satellite positions

n-element vector of angles

Elevation angles for visible satellite positions, returned as an *n*-element vector of angles. *n* is the number of visible satellite positions in the plot. Elevation angles are measured from the horizon line with 90 degrees being directly up.

Example: `[45 90 27 74]`

Data Types: `double`

`vis` — Satellite visibility

n-element `logical` array

Satellite visibility, returned as an *n*-element `logical` array. Each element indicates whether the satellite position given by `az` and `el` is visible.

Data Types: `logical`

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

Objects

`gnssSensor` | `gpsSensor` | `imuSensor`

Functions

`skyplot` | `gnssconstellation` | `pseudoranges` | `receiverposition`

Introduced in R2021a

ned2lla

Transform local north-east-down coordinates to geodetic coordinates

Syntax

```
lla = ned2lla(xyzNED,lla0,method)
```

Description

`lla = ned2lla(xyzNED,lla0,method)` transforms the local north-east-down (NED) Cartesian coordinates `xyzNED` to geodetic coordinates `lla`. Specify the origin of the local NED system as the geodetic coordinates `lla0`.

Note

- The latitude and longitude values in the geodetic coordinate system use the World Geodetic System of 1984 (WGS84) standard.
- Specify altitude as height in meters above the WGS84 reference ellipsoid.

Examples

Transform NED Coordinates to Geodetic Coordinates

Specify the geodetic coordinates of the local origin in Zermatt, Switzerland.

```
lla0 = [46.017 7.750 1673]; % [lat0 lon0 alt0]
```

Specify the NED coordinates of a point of interest, in meters. In this case, the point of interest is the Matterhorn.

```
xyzNED = [-4556.3 -7134.8 -2852.4]; % [xNorth yEast zDown]
```

Transform the local NED coordinates to geodetic coordinates using flat earth approximation.

```
lla = ned2lla(xyzNED,lla0,'flat')
```

```
lla = 1×3
10^3 ×

    0.0460    0.0077    4.5254
```

Input Arguments

xyzNED — Local NED Cartesian coordinates

three-element row vector | *n*-by-3 matrix

Local NED Cartesian coordinates, specified as a three-element row vector or an *n*-by-3 matrix. *n* is the number of points to transform. Specify each point in the form `[xNorth yEast zDown]`. *xNorth*, *yEast*, and *zDown* are the respective *x*-, *y*-, and *z*-coordinates, in meters, of the point in the local NED system.

Data Types: `double`

`lla0` — Origin of local NED system in geodetic coordinates
three-element row vector | *n*-by-3 matrix

Origin of the local NED system with the geodetic coordinates, specified as a three-element row vector or an *n*-by-3 matrix. *n* is the number of origin points. Specify each point in the form `[lat0 lon0 alt0]`. *lat0* and *lon0* specify the latitude and longitude respectively in degrees. *alt0* specifies the altitude in meters.

Data Types: `double`

`method` — Transformation method
`'flat'` | `'ellipsoid'`

Transformation method, specified as `'flat'` or `'ellipsoid'`. This argument specifies whether the function assumes the planet is flat or ellipsoidal.

The flat Earth transformation method has these limitations:

- Assumes that the flight path and bank angle are zero.
- Assumes that the flat Earth *z*-axis is normal to the Earth at only the initial geodetic latitude and longitude. This method has higher accuracy over small distances from the initial geodetic latitude and longitude, and closer to the equator. The method calculates a longitude with higher accuracy when the variation in latitude is smaller.
- Latitude values of +90 and -90 may return unexpected values because of singularity at the poles.

Data Types: `char` | `string`

Output Arguments

`lla` — Geodetic coordinates
three-element row vector | *n*-by-3 matrix

Geodetic coordinates, returned as a three-element row vector or an *n*-by-3 matrix. *n* is the number of transformed points. Each point is in the form `[lat lon alt]`. *lat* and *lon* specify the latitude and longitude, respectively, in degrees. *alt* specifies the altitude in meters.

Data Types: `double`

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

Functions
`enu2lla` | `lla2enu` | `lla2ned`

Introduced in R2021a

copy

Create copy of path object

Syntax

```
path2 = copy(path1)
```

Description

`path2 = copy(path1)` creates a copy of the path object, `path2`, from the path object, `path1`.

Input Arguments

path1 — path object

`navPath` object

Path object, specified as a `navPath` object.

Data Types: `object`

Output Arguments

path2 — path object

`navPath` object

Path object, returned as a `navPath` object.

Data Types: `object`

See Also

`navPath`

Introduced in R2019b

magcal

Magnetometer calibration coefficients

Syntax

```
[A,b,expmfs] = magcal(D)
[A,b,expmfs] = magcal(D,fitkind)
```

Description

`[A,b,expmfs] = magcal(D)` returns the coefficients needed to correct uncalibrated magnetometer data `D`.

To produce the calibrated magnetometer data `C`, use equation `C = (D-b)*A`. The calibrated data `C` lies on a sphere of radius `expmfs`.

`[A,b,expmfs] = magcal(D,fitkind)` constrains the matrix `A` to be the type specified by `fitkind`. Use this syntax when only the soft- or hard-iron effect needs to be corrected.

Examples

Correct Data Lying on Ellipsoid

Generate uncalibrated magnetometer data lying on an ellipsoid.

```
c = [-50; 20; 100]; % ellipsoid center
r = [30; 20; 50]; % semiaxis radii

[x,y,z] = ellipsoid(c(1),c(2),c(3),r(1),r(2),r(3),20);
D = [x(:),y(:),z(:)];
```

Correct the magnetometer data so that it lies on a sphere. The option for the calibration is set by default to 'auto'.

```
[A,b,expmfs] = magcal(D); % calibration coefficients
expmfs % Dipaly expected  magnetic field strength in uT
```

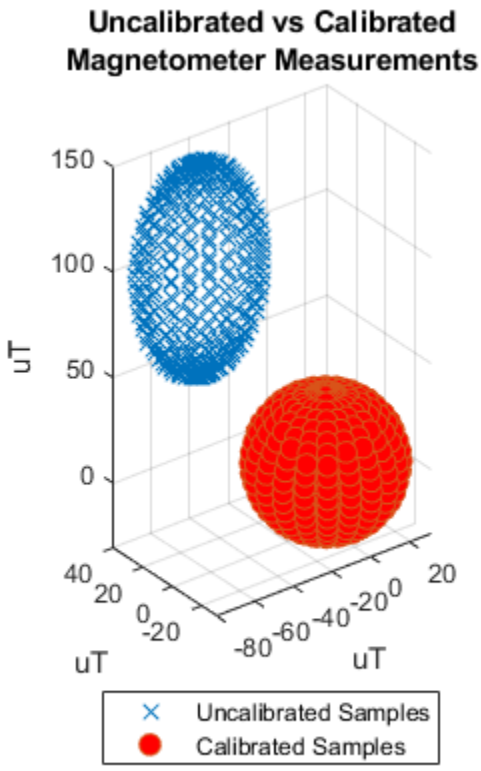
```
expmfs = 31.0723
```

```
C = (D-b)*A; % calibrated data
```

Visualize the uncalibrated and calibrated magnetometer data.

```
figure(1)
plot3(x(:),y(:),z(:),'LineStyle','none','Marker','X','MarkerSize',8)
hold on
grid(gca,'on')
plot3(C(:,1),C(:,2),C(:,3),'LineStyle','none','Marker', ...
            'o','MarkerSize',8,'MarkerFaceColor','r')
axis equal
xlabel('uT')
```

```
ylabel('uT')
zlabel('uT')
legend('Uncalibrated Samples', 'Calibrated Samples','Location', 'southoutside')
title("Uncalibrated vs Calibrated" + newline + "Magnetometer Measurements")
hold off
```



Input Arguments

`D` — Raw magnetometer data

N-by-3 matrix (default)

Input matrix of raw magnetometer data, specified as a *N*-by-3 matrix. Each column of the matrix corresponds to the magnetometer measurements in the first, second and third axes, respectively. Each row of the matrix corresponds to a single three-axis measurement.

Data Types: `single` | `double`

`fitkind` — Matrix output type

'auto' (default) | 'eye' | 'diag' | 'sym'

Matrix type for output `A`. The matrix type of `A` can be constrained to:

- `'eye'` – identity matrix
- `'diag'` – diagonal

- `'sym'` – symmetric
- `'auto'` – whichever of the previous options gives the best fit

Output Arguments

`A` — Correction matrix for soft-iron effect

3-by-3 matrix

Correction matrix for the soft-iron effect, returned as a 3-by-3 matrix.

`b` — Correction vector for hard-iron effect

3-by-1 vector

Correction vector for the hard-iron effect, returned as a 3-by-1 array.

`expmfs` — Expected magnetic field strength

scalar

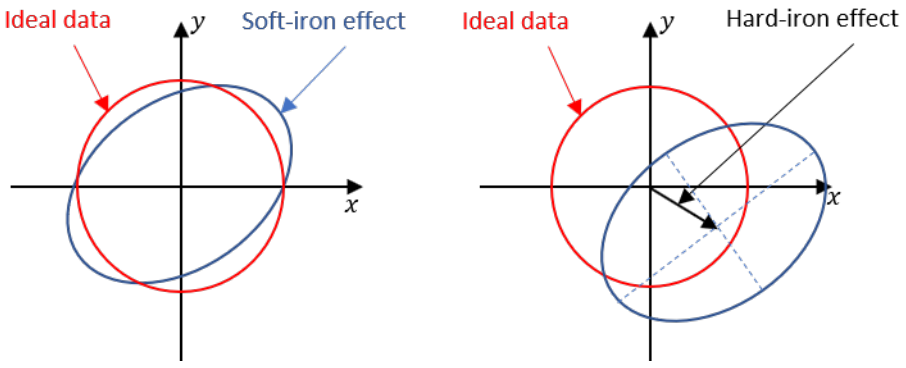
Expected magnetic filed strength, returned as a scalar.

More About

Soft- and Hard-Iron Effects

Because a magnetometer usually rotates through a full range of 3-D rotation, the ideal measurements from a magnetometer should form a perfect sphere centered at the origin if the magnetic field is unperturbed. However, due to distorting magnetic fields from the sensor circuit board and the surrounding environment, the spherical magnetic measurements can be perturbed. In general, two effects exist.

1. The soft-iron effect is described as the distortion of the ellipsoid from a sphere and the tilt of the ellipsoid, as shown in the left figure. This effect is caused by disturbances that influence the magnetic field but may not generate their own magnetic field. For example, metals such as nickel and iron can cause this kind of distortion.
2. The hard-iron effect is described as the offset of the ellipsoid center from the origin. This effect is produced by materials that exhibit a constant, additive field to the earth's magnetic field. This constant additive offset is in addition to the soft-iron effect as shown in the figure on the right.



The underlying algorithm in `magcal` determines the best-fit ellipsoid to the raw sensor readings and attempts to "invert" the ellipsoid to produce a sphere. The goal is to generate a correction matrix `A` to

account for the soft-iron effect and a vector `b` to account for the hard-iron effect. The three output options, 'eye', 'diag' and 'sym' correspond to three parameter-solving algorithms, and the 'auto' option chooses among these three options to give the best fit.

References

[1] Ozyagcilar, T. "Calibrating an eCompass in the Presence of Hard and Soft-iron Interference." *Freescale Semiconductor Ltd*. 1992, pp. 1-17.

See Also

Classes
`magparams`

Objects
`imuSensor`

Introduced in R2019a

mapClutter

Generate map with randomly scattered obstacles

Syntax

```
map = mapClutter
map = mapClutter(numObst)
map = mapClutter(numObst,shapes)
map = mapClutter( ___ ,Name,Value)
```

Description

`map = mapClutter` generates a 2-D occupancy map as a `binaryOccupancyMap` object `map`, with a width and height of 50 meters and a resolution of 5 cells per meter. The map contains 20 randomly distributed obstacles of types `Box` and `Circle`. Generated obstacles have random sizes.

`map = mapClutter(numObst)` generates a 2-D occupancy map, of the default size and resolution, with a specified number of randomly distributed obstacles, `numObst`, of default shapes.

`map = mapClutter(numObst,shapes)` generates a 2-D occupancy map, of the default size and resolution, with a specified number of obstacles, `numObst`, of specified shapes, `shapes`.

`map = mapClutter( ___ ,Name,Value)` specifies options using one or more name-value pair arguments in addition to any combination of input arguments from previous syntaxes. For example, `'MapSize',[50 30]` generates a randomly distributed obstacle map with a width of 50 meters and height of 30 meters.

Examples

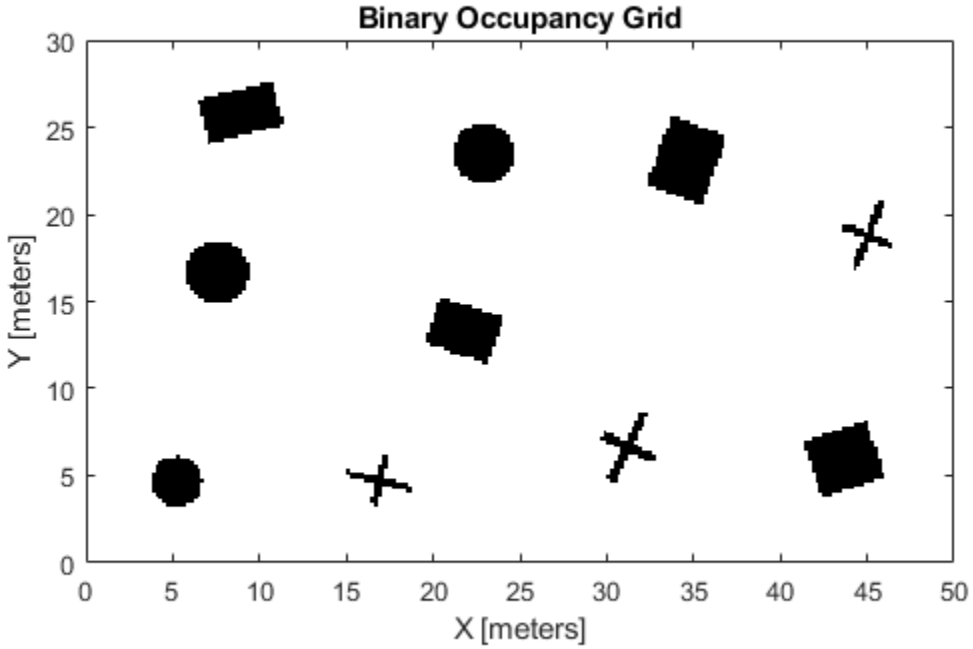
Generate Randomly Distributed Obstacle Map

Generate a 2-D occupancy map with ten randomly scattered obstacles of types `Box`, `Circle`, and `Plus`. Specify the size of the map as 50 meters wide and 30 meters high with a resolution of 5 cells per meters.

```
map = mapClutter(10,{'Box','Plus','Circle'},'MapSize',[50 30],'MapResolution',5);
```

Visualize the generated obstacle map.

```
show(map)
```



Input Arguments

`numObst` — Number of obstacles

`20` (default) | positive integer

Number of obstacles, specified as a positive integer.

Data Types: `single` | `double`

`shapes` — Obstacle shapes

`{'Box','Circle'}` (default) | string scalar | vector of strings | cell array of character vectors

Obstacle shapes, specified as a string scalar, vector of strings, or cell array of character vectors. The only valid shapes are `Box`, `Circle`, and `Plus`.

When you specify a string scalar, the function generates a map with obstacles of only the specified shape.

Example: `"Box"`

When you specify a vector of strings or a cell array of character vectors, the function generates a map with obstacles of each specified shape.

Example: `["Box","Plus"]`

Example: `{'Box','Plus','Circle'}`

Data Types: `cell` | `string`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'MapSize',[50 40]` generates a randomly distributed obstacle map with a width of 50 meters and height of 40 meters.

`MapSize` — Width and height of generated map

`[50 50]` (default) | two-element vector of positive real finite numbers

Width and height of the generated map, specified as the comma-separated pair consisting of `'MapSize'` and a two-element vector of positive real finite numbers of the form [*Width*,*Height*]. Specify both values in meters.

Example: `'MapSize',[50 30]`

Data Types: `single` | `double`

`MapResolution` — Resolution of generated map

`5` (default) | positive real scalar

Resolution of the generated map, specified as the comma-separated pair consisting of `'MapResolution'` and a positive real scalar in cells per meter.

Example: `'MapResolution',10`

Data Types: `single` | `double`

Output Arguments

`map` — Map with randomly scattered obstacles

`binaryOccupancyMap` object

A map with randomly scattered obstacles, returned as a `binaryOccupancyMap` object.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`mapMaze` | `binaryOccupancyMap` | `validatorOccupancyMap`

Introduced in R2020b

mapMaze

Generate random 2-D maze map

Syntax

```
map = mapMaze
map = mapMaze(passageWidth)
map = mapMaze(passageWidth,wallThickness)
map = mapMaze(___,Name,Value)
```

Description

`map = mapMaze` generates a random 2-D maze map, `map`, as a `binaryOccupancyMap` object with a width and height of 50 meters and a resolution of 5 cells per meter. The maze map contains straight passages, turns, and T-junctions with a passage width of 4 grid cells and wall thickness of 1 grid cell.

`map = mapMaze(passageWidth)` generates a `binaryOccupancyMap` of a maze of the default size and resolution with a specified passage width, `passageWidth`, in number of grid cells.

`map = mapMaze(passageWidth,wallThickness)` specifies a wall thickness, `wallThickness`, in number of grid cells.

`map = mapMaze(___,Name,Value)` specifies options using one or more name-value pair arguments in addition to any combination of input arguments from previous syntaxes. For example, `'MapSize',[50 30]` generates a random maze map with a width of 50 meters and height of 30 meters.

Examples

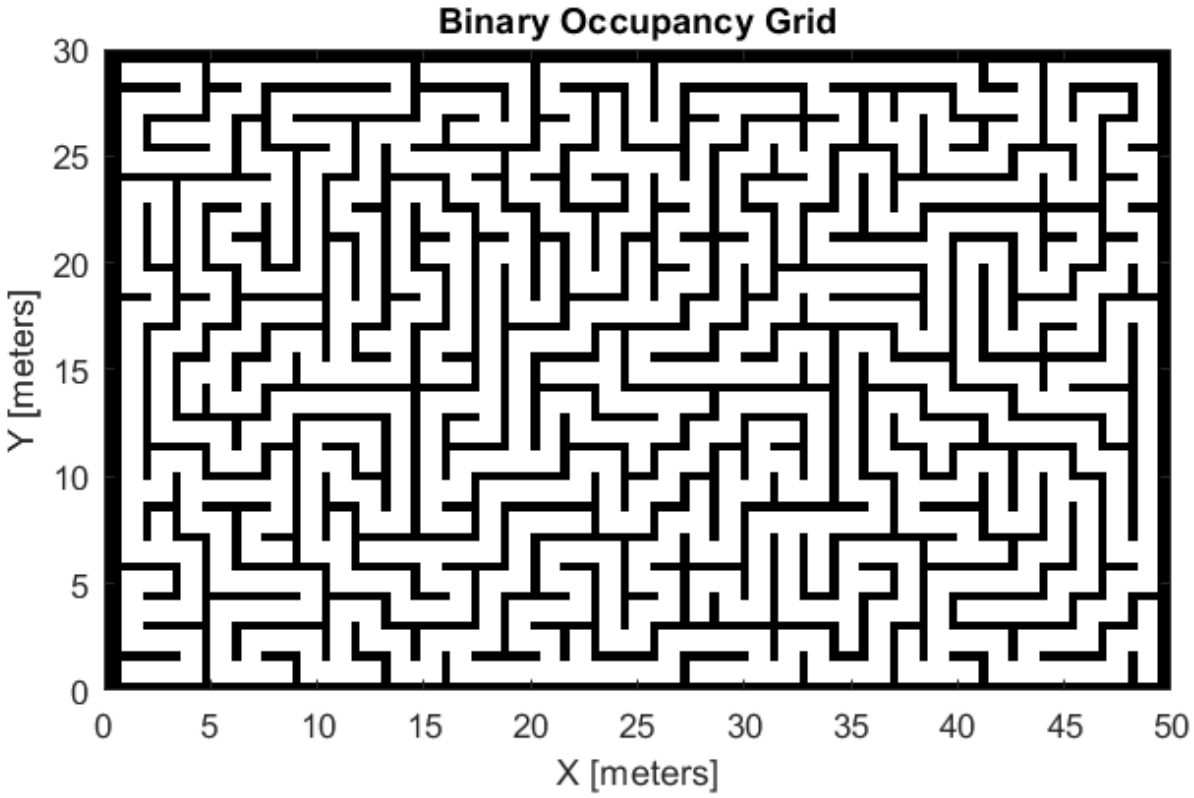
Generate Random Maze Map

Generate a random 2-D maze map with a passage width of 5 grid cells and wall thickness of 2 grid cells. Specify the size of the map as 50 meters wide and 30 meters high with a resolution of 5 cells per meter.

```
map = mapMaze(5,2,'MapSize',[50 30],'MapResolution',5);
```

Visualize the generated obstacle map.

```
show(map)
```



Input Arguments

`passageWidth` — Width of maze passage
4 (default) | positive integer

Width of maze passage, specified as a positive integer in number of grid cells.

Data Types: `single` | `double`

`wallThickness` — Thickness of maze wall
1 (default) | positive integer

Thickness of maze wall, specified as a positive integer in number of grid cells.

Data Types: `single` | `double`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'MapSize',[50 40]` generates a randomly distributed obstacle map with a width of 50 meters and height of 40 meters.

`MapSize` — Width and height of generated map
`[50 50]` (default) | two-element vector of positive real finite numbers

Width and height of the generated map, specified as the comma-separated pair consisting of `'MapSize'` and a two-element vector of positive real finite numbers of the form [*Width Height*]. Specify both values in meters.

Example: `'MapSize',[50 30]`

Data Types: `single` | `double`

`MapResolution` — Resolution of generated map
`5` (default) | positive real scalar

Resolution of the generated map, specified as the comma-separated pair consisting of `'MapResolution'` and a positive real scalar in cells per meter.

Example: `'MapResolution',10`

Data Types: `single` | `double`

Output Arguments

`map` — Random maze map
`binaryOccupancyMap` object

Random maze map, returned as a `binaryOccupancyMap` object.

Tips

- when the number of grid cells along map width could not accommodate given maze parameters, `ceil(MapWidth*MapResolution)` must be greater than or equal to `(passageWidth +2*wallThickness)`.
- when the number of grid cells along map height could not accommodate given maze parameters, `ceil(MapHeight*MapResolution)` must be greater than or equal to `(passageWidth +2*wallThickness)`.

See Also
`mapClutter` | `binaryOccupancyMap` | `validatorOccupancyMap`

Introduced in R2021a

matchScans

Estimate pose between two laser scans

Syntax

```
pose = matchScans(currScan,refScan)
pose = matchScans(currRanges,currAngles,refRanges,refAngles)
[pose,stats] = matchScans( ___ )
[ ___ ] = matchScans( ___ ,Name,Value)
```

Description

`pose = matchScans(currScan,refScan)` finds the relative pose between a reference `lidarScan` and a current `lidarScan` object using the normal distributions transform (NDT).

`pose = matchScans(currRanges,currAngles,refRanges,refAngles)` finds the relative pose between two laser scans specified as ranges and angles.

`[pose,stats] = matchScans( ___ )` returns additional statistics about the scan match result using the previous input arguments.

`[ ___ ] = matchScans( ___ ,Name,Value)` specifies additional options specified by one or more `Name,Value` pair arguments.

Examples

Match Lidar Scans

Create a reference lidar scan using `lidarScan`. Specify ranges and angles as vectors.

```
refRanges = 5*ones(1,300);
refAngles = linspace(-pi/2,pi/2,300);
refScan = lidarScan(refRanges,refAngles);
```

Using the `transformScan` (Robotics System Toolbox) function, generate a second lidar scan at an `x,y` offset of `(0.5,0.2)`.

```
currScan = transformScan(refScan,[0.5 0.2 0]);
```

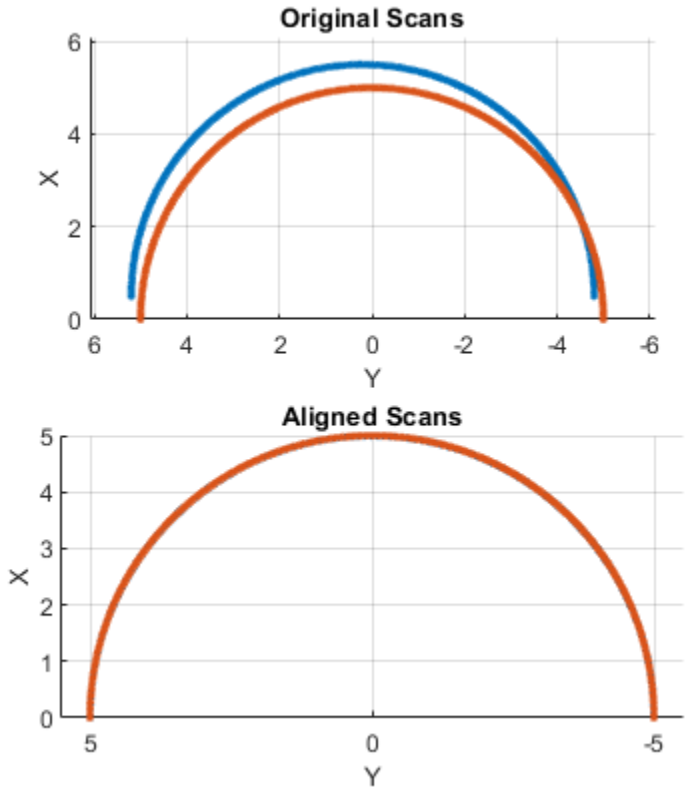
Match the reference scan and the second scan to estimate the pose difference between them.

```
pose = matchScans(currScan,refScan);
```

Use the `transformScan` function to align the scans by transforming the second scan into the frame of the first scan using the relative pose difference. Plot both the original scans and the aligned scans.

```
currScan2 = transformScan(currScan,pose);

subplot(2,1,1);
hold on
```

```
plot(currScan)
plot(refScan)
title('Original Scans')
hold off

subplot(2,1,2);
hold on
plot(currScan2)
plot(refScan)
title('Aligned Scans')
xlim([0 5])
hold off
```



Match Laser Scans

This example uses the `'fminunc'` solver algorithm to perform scan matching. This solver algorithm requires an Optimization Toolbox™ license.

Specify a reference laser scan as ranges and angles.

```
refRanges = 5*ones(1,300);
refAngles = linspace(-pi/2,pi/2,300);
```

Using the `transformScan` (Robotics System Toolbox) function, generate a second laser scan at an `x,y` offset of `(0.5,0.2)`.

```
[currRanges,currAngles] = transformScan(refRanges,refAngles,[0.5 0.2 0]);
```

Match the reference scan and the second scan to estimate the pose difference between them.

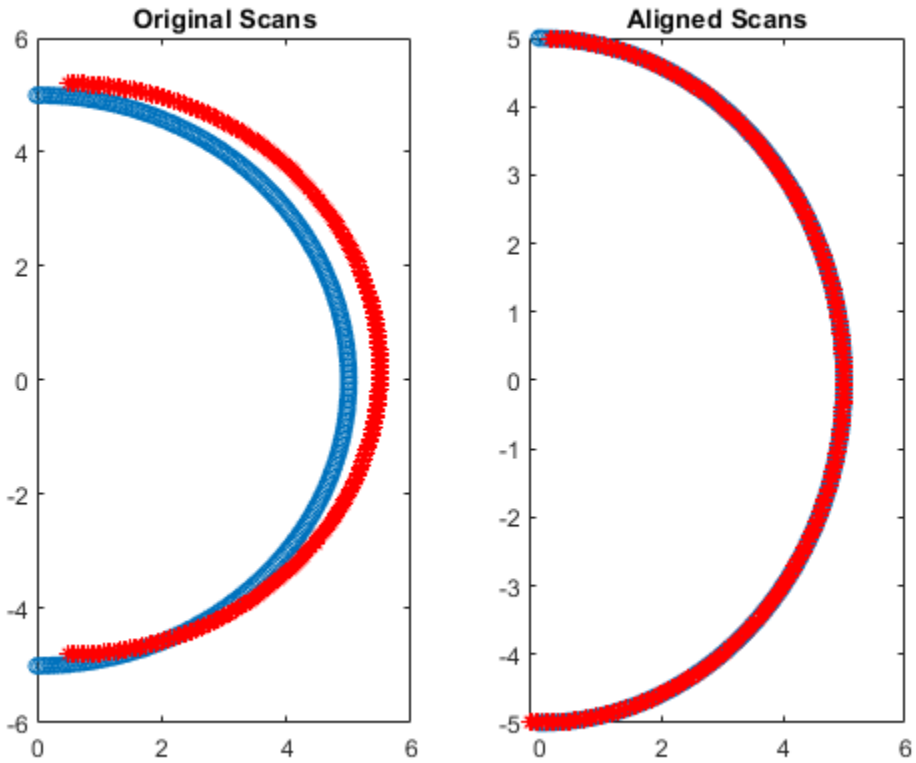
```
pose = matchScans(currRanges,currAngles,refRanges,refAngles,'SolverAlgorithm','fminunc');
```

Improve the estimate by giving an initial pose estimate.

```
pose = matchScans(currRanges,currAngles,refRanges,refAngles,...
                  'SolverAlgorithm','fminunc','InitialPose',[-0.4 -0.1 0]);
```

Use the `transformScan` function to align the scans by transforming the second scan into the frame of the first scan using the relative pose difference. Plot both the original scans and the aligned scans.

```
[currRanges2,currAngles2] = transformScan(currRanges,currAngles,pose);

[x1, y1] = pol2cart(refAngles,refRanges);
[x2, y2] = pol2cart(currAngles,currRanges);
[x3, y3] = pol2cart(currAngles2,currRanges2);

subplot(1,2,1)
plot(x1,y1,'o',x2,y2,'*r')
title('Original Scans')
subplot(1,2,2)
plot(x1,y1,'o',x3,y3,'*r')
title('Aligned Scans')
```



Input Arguments

`currScan` — Current lidar scan readings

`lidarScan` object

Current lidar scan readings, specified as a `lidarScan` object.

Your lidar scan can contain `Inf` and `NaN` values, but the algorithm ignores them.

`refScan` — Reference lidar scan readings

`lidarScan` object

Reference lidar scan readings, specified as a `lidarScan` object.

Your lidar scan can contain `Inf` and `NaN` values, but the algorithm ignores them.

`currRanges` — Current laser scan ranges

vector in meters

Current laser scan ranges, specified as a vector. Ranges are given as distances to objects measured from the laser sensor.

Your laser scan ranges can contain `Inf` and `NaN` values, but the algorithm ignores them.

`currAngles` — Current laser scan angles

vector in radians

Current laser scan angles, specified as a vector in radians. Angles are given as the orientations of the corresponding range measurements.

`refRanges` — Reference laser scan ranges

vector in meters

Reference laser scan ranges, specified as a vector in meters. Ranges are given as distances to objects measured from the laser sensor.

Your laser scan ranges can contain `Inf` and `NaN` values, but the algorithm ignores them.

`refAngles` — Reference laser scan angles

vector in radians

Reference laser scan angles, specified as a vector in radians. Angles are given as the orientations of the corresponding range measurements.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `"InitialPose",[1 1 pi/2]`

`SolverAlgorithm` — Optimization algorithm

`"trust-region"` (default) | `"fminunc"`

Optimization algorithm, specified as either `"trust-region"` or `"fminunc"`. Using `"fminunc"` requires an Optimization Toolbox™ license.

`InitialPose` — Initial guess of current pose
`[0 0 0]` (default) | `[x y theta]`

Initial guess of the current pose relative to the reference laser scan, specified as the comma-separated pair consisting of `"InitialPose"` and an `[x y theta]` vector. `[x y]` is the translation in meters and `theta` is the rotation in radians.

`CellSize` — Length of cell side
1 (default) | numeric scalar

Length of a cell side in meters, specified as the comma-separated pair consisting of `"CellSize"` and a numeric scalar. `matchScans` uses the cell size to discretize the space for the NDT algorithm.

Tuning the cell size is important for proper use of the NDT algorithm. The optimal cell size depends on the input scans and the environment of your robot. Larger cell sizes can lead to less accurate matching with poorly sampled areas. Smaller cell sizes require more memory and less variation between subsequent scans. Sensor noise influences the algorithm with smaller cell sizes as well. Choosing a proper cell size depends on the scale of your environment and the input data.

`MaxIterations` — Maximum number of iterations
400 (default) | scalar integer

Maximum number of iterations, specified as the comma-separated pair consisting of `"MaxIterations"` and a scalar integer. A larger number of iterations results in more accurate pose estimates, but at the expense of longer execution time.

`ScoreTolerance` — Lower bounds on the change in NDT score
`1e-6` (default) | numeric scalar

Lower bound on the change in NDT score, specified as the comma-separated pair consisting of `"ScoreTolerance"` and a numeric scalar. The NDT score is stored in the `Score` field of the output `stats` structure. Between iterations, if the score changes by less than this tolerance, the algorithm converges to a solution. A smaller tolerance results in more accurate pose estimates, but requires a longer execution time.

Output Arguments

`pose` — Pose of current scan
`[x y theta]`

Pose of current scan relative to the reference scan, returned as `[x y theta]`, where `[x y]` is the translation in meters and `theta` is the rotation in radians.

`stats` — Scan matching statistics
structure

Scan matching statistics, returned as a structure with the following fields:

- `Score` — Numeric scalar representing the NDT score while performing scan matching. This score is an estimate of the likelihood that the transformed current scan matches the reference scan. `Score` is always nonnegative. Larger scores indicate a better match.

- `Hessian` — 3-by-3 matrix representing the Hessian of the NDT cost function at the given `pose` solution. The Hessian is used as an indicator of the uncertainty associated with the pose estimate.

References

[1] Biber, P., and W. Strasser. "The Normal Distributions Transform: A New Approach to Laser Scan Matching." *Intelligent Robots and Systems Proceedings*. 2003.

[2] Magnusson, Martin. "The Three-Dimensional Normal-Distributions Transform -- an Efficient Representation for Registration, Surface Analysis, and Loop Detection." PhD Dissertation. Örebro University, School of Science and Technology, 2009.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Code generation is supported for the default `SolverAlgorithm`, `"trust-region"`. You cannot use the `"fminunc"` algorithm in code generation.

See Also

Functions

`matchScansGrid` | `matchScansLine` | `transformScan` | `lidarScan`

Classes

`occupancyMap` | `monteCarloLocalization`

Topics

“Estimate Robot Pose with Scan Matching”

Introduced in R2019b

matchScansGrid

Estimate pose between two lidar scans using grid-based search

Syntax

```
pose = matchScansGrid(currScan,refScan)
[pose,stats] = matchScansGrid( ___ )
[ ___ ] = matchScansGrid( ___ ,Name,Value)
```

Description

`pose = matchScansGrid(currScan,refScan)` finds the relative pose between a reference `lidarScan` and a current `lidarScan` object using a grid-based search. `matchScansGrid` converts lidar scan pairs into probabilistic grids and finds the pose between the two scans by correlating their grids. The function uses a branch-and-bound strategy to speed up computation over large discretized search windows.

`[pose,stats] = matchScansGrid( ___ )` returns additional statistics about the scan match result using the previous input arguments.

`[ ___ ] = matchScansGrid( ___ ,Name,Value)` specifies options using one or more `Name,Value` pair arguments. For example, `'InitialPose',[1 1 pi/2]` specifies an initial pose estimate for scan matching.

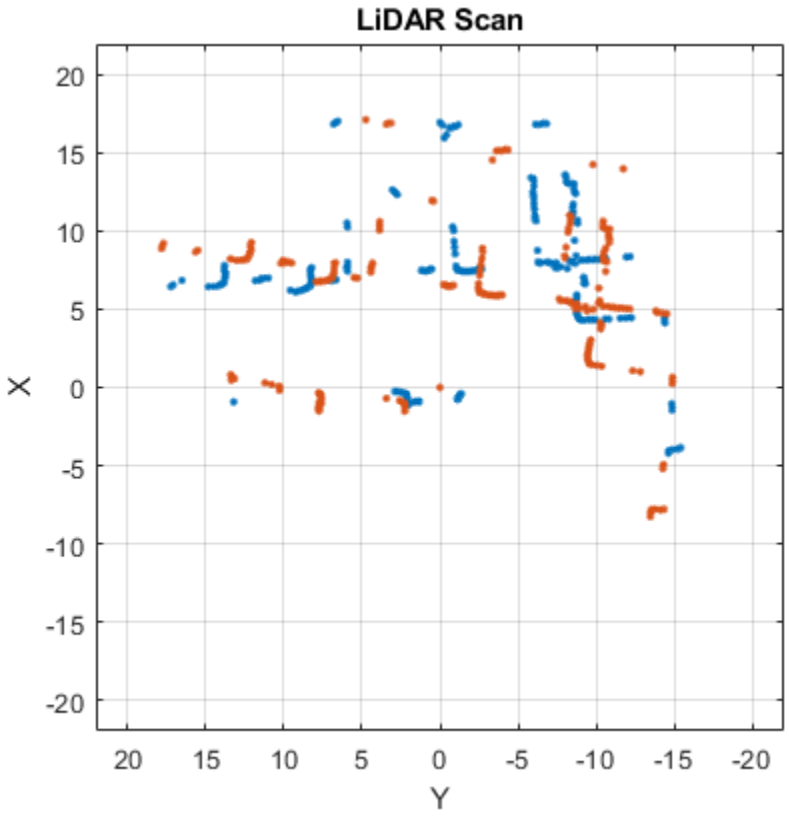
Examples

Match Scans Using Grid-Based Search

Perform scan matching using a grid-based search to estimate the pose between two laser scans. Generate a probabilistic grid from the scans and estimate the pose difference from those grids.

Load the laser scan data. These two scans are from an actual lidar sensor with changes in the robot pose and are stored as `lidarScan` objects.

```
load laserScans.mat scan scan2
plot(scan)
hold on
plot(scan2)
hold off
```

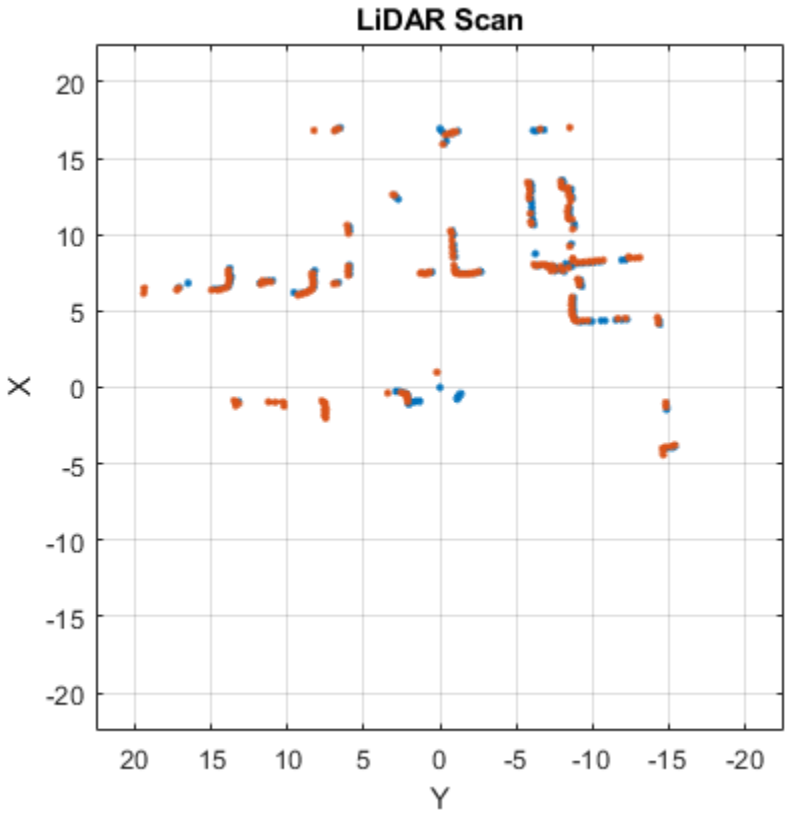


Use `matchScansGrid` to estimate the pose between the two scans.

```
relPose = matchScansGrid(scan2,scan);
```

Using the estimated pose, transform the current scan back to the reference scan. The scans overlap closely when you plot them together.

```
scan2Tformed = transformScan(scan2,relPose);

plot(scan)
hold on
plot(scan2Tformed)
hold off
```



Input Arguments

`currScan` — Current lidar scan readings

`lidarScan` object

Current lidar scan readings, specified as a `lidarScan` object.

Your lidar scan can contain `Inf` and `NaN` values, but the algorithm ignores them.

`refScan` — Reference lidar scan readings

`lidarScan` object

Reference lidar scan readings, specified as a `lidarScan` object.

Your lidar scan can contain `Inf` and `NaN` values, but the algorithm ignores them.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'InitialPose',[1 1 pi/2]`

`InitialPose` — Initial guess of current pose

`[0 0 0]` (default) | `[x y theta]`

Initial guess of the current pose relative to the reference laser scan, specified as the comma-separated pair consisting of `'InitialPose'` and an `[x y theta]` vector. `[x y]` is the translation in meters and `theta` is the rotation in radians.

`Resolution` — Grid cells per meter
`20` (default) | positive integer

Grid cells per meter, specified as the comma-separated pair consisting of `'Resolution'` and a positive integer. The accuracy of the scan matching result is accurate up to the grid cell size.

`MaxRange` — Maximum range of lidar sensor
`8` (default) | positive scalar

Maximum range of lidar sensor, specified as the comma-separated pair consisting of `'MaxRange'` and a positive scalar.

`TranslationSearchRange` — Search range for translation
`[4 4]` (default) | `[x y]` vector

Search range for translation, specified as the comma-separated pair consisting of `'TranslationSearchRange'` and an `[x y]` vector. These values define the search window in meters around the initial translation estimate given in `InitialPose`. If the `InitialPose` is given as `[x0 y0]`, then the search window coordinates are `[x0-x x0+x]` and `[y0-y y0+y]`. This parameter is used only when `InitialPose` is specified.

`RotationSearchRange` — Search range for rotation
`pi/4` (default) | positive scalar

Search range for rotation, specified as the comma-separated pair consisting of `'RotationSearchRange'` and a positive scalar. This value defines the search window in radians around the initial rotation estimate given in `InitialPose`. If the `InitialPose` rotation is given as `th0`, then the search window is `[th0-a th0+a]`, where `a` is the rotation search range. This parameter is used only when `InitialPose` is specified.

Output Arguments

`pose` — Pose of current scan
`[x y theta]` vector

Pose of current scan relative to the reference scan, returned as an `[x y theta]` vector, where `[x y]` is the translation in meters and `theta` is the rotation in radians.

`stats` — Scan matching statistics
structure

Scan matching statistics, returned as a structure with the following field:

- `Score` — Numeric scalar representing the score while performing scan matching. This score is an estimate of the likelihood that the transformed current scan matches the reference scan. `Score` is always nonnegative. Larger scores indicate a better match, but values vary depending on the lidar data used.
- `Covariance` — Estimated covariance representing the confidence of the computed relative pose, returned as a 3-by-3 matrix.

References

[1] Hess, Wolfgang, Damon Kohler, Holger Rapp, and Daniel Andor. "Real-Time Loop Closure in 2D LIDAR SLAM." *2016 IEEE International Conference on Robotics and Automation (ICRA)*. 2016.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

Functions

`matchScans` | `matchScansLine` | `lidarScan` | `transformScan`

Classes

`lidarSLAM`

Topics

“Implement Simultaneous Localization And Mapping (SLAM) with Lidar Scans”
“Implement Online Simultaneous Localization And Mapping (SLAM) with Lidar Scans”

Introduced in R2019b

matchScansLine

Estimate pose between two laser scans using line features

Syntax

```
relpose = matchScansLine(currScan,refScan,initialRelPose)
[relpose,stats] = matchScansLine( ___ )
[relpose,stats,debugInfo] = matchScansLine( ___ )
[ ___ ] = matchScansLine( ___ ,Name,Value)
```

Description

`relpose = matchScansLine(currScan,refScan,initialRelPose)` estimates the relative pose between two scans based on matched line features identified in each scan. Specify an initial guess on the relative pose, `initialRelPose`.

`[relpose,stats] = matchScansLine( ___ )` returns additional information about the covariance and exit condition in `stats` as a structure using the previous inputs.

`[relpose,stats,debugInfo] = matchScansLine( ___ )` returns additional debugging info, `debugInfo`, from the line-based scan matching result.

`[ ___ ] = matchScansLine( ___ ,Name,Value)` specifies options using one or more `Name,Value` pair arguments.

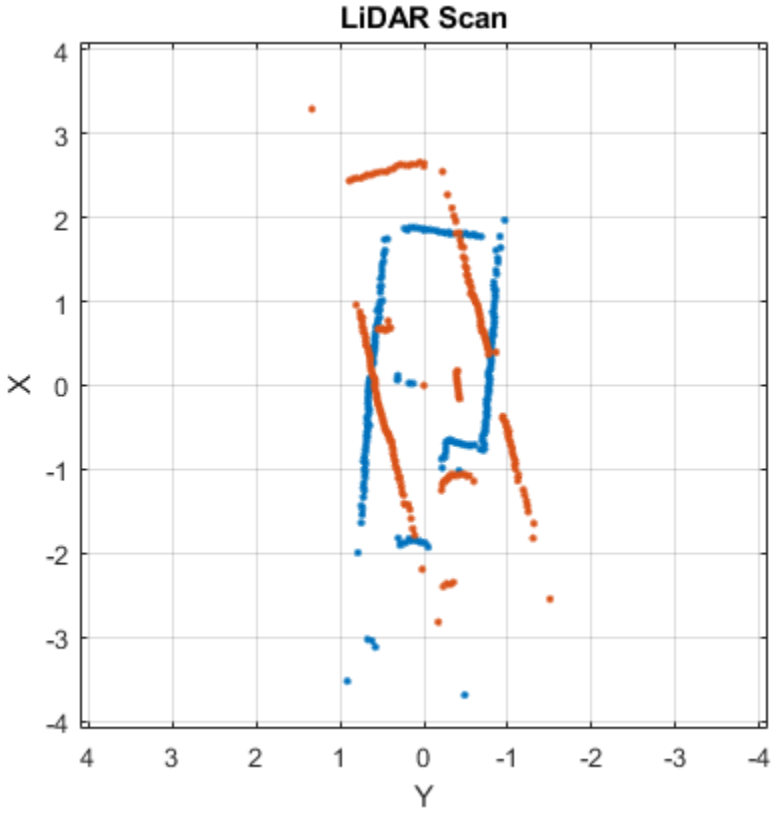
Examples

Estimate Pose of Scans with Line Features

This example shows how to use the `matchScansLine` function to estimate the relative pose between lidar scans given an initial estimate. The identified line features are visualized to show how the scan-matching algorithm associates features between scans.

Load a pair of lidar scans. The `.mat` file also contains an initial guess of the relative pose difference, `initGuess`, which could be based on odometry or other sensor data.

```
load tb3_scanPair.mat
plot(s1)
hold on
plot(s2)
hold off
```



Set parameters for line feature extraction and association. The noise of the lidar data determines the smoothness threshold, which defines when a line break occurs for a specific line feature. Increase this value for more noisy lidar data. The compatibility scale determines when features are considered matches. Increase this value for looser restrictions on line feature parameters.

```
smoothnessThresh = 0.2;
compatibilityScale = 0.002;
```

Call `matchScansLine` with the given initial guess and other parameters specified as name-value pairs. The function calculates line features for each scan, attempts to match them, and uses an overall estimate to get the difference in pose.

```
[relPose, stats, debugInfo] = matchScansLine(s2, s1, initGuess, ...
                                             'SmoothnessThreshold', smoothnessThresh, ...
                                             'CompatibilityScale', compatibilityScale);
```

After matching the scans, the `debugInfo` output gives you information about the detected line feature parameters, `[rho alpha]`, and the hypothesis of which features match between scans.

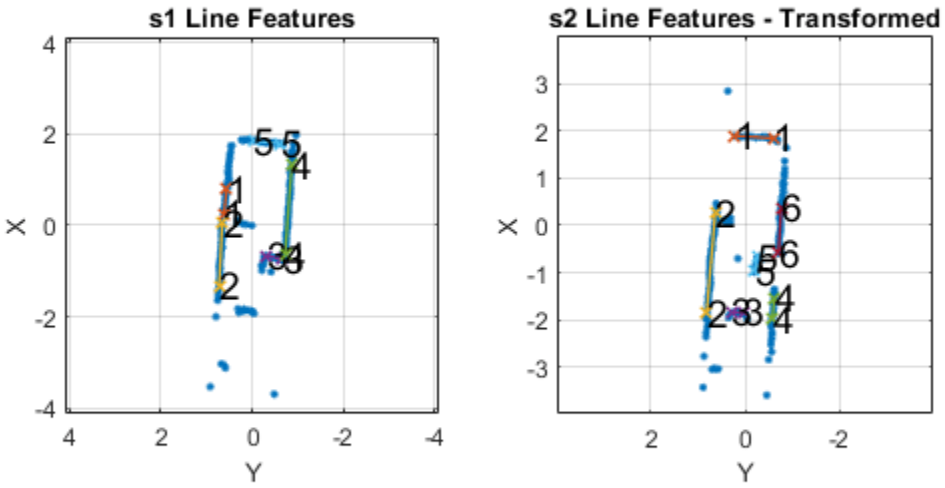
`debugInfo.MatchHypothesis` states that the first, second, and sixth line feature in `s1` match the fifth, second, and fourth features in `s2`.

```
debugInfo.MatchHypothesis
```

```
ans = 1×6

     5     2     0     0     0     4
```

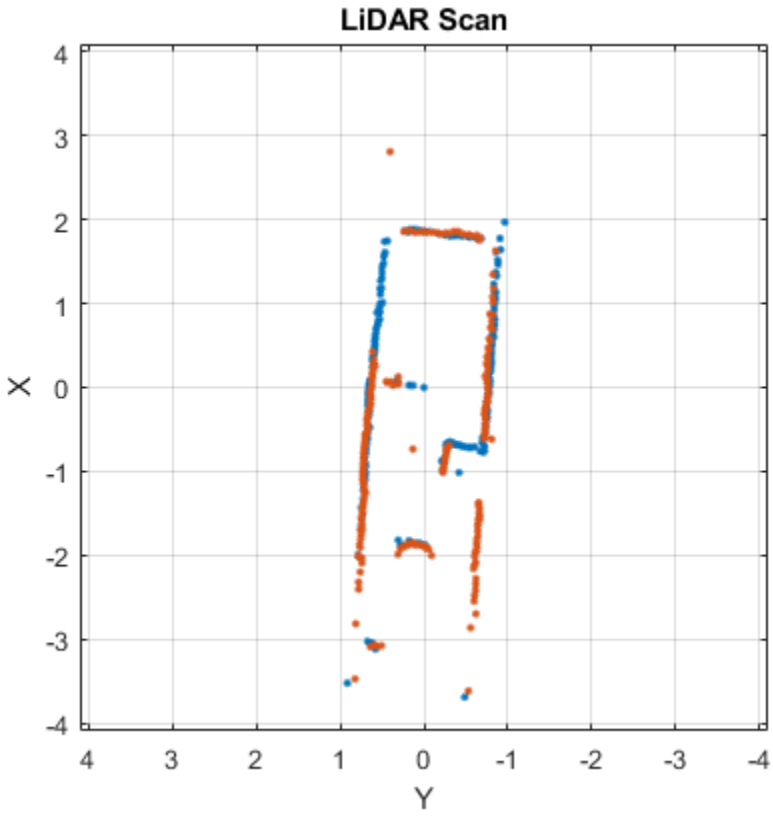
The provided helper function plots these two scans and the features extracted with labels. `s2` is transformed to be in the same frame based on the initial guess for relative pose.

```
exampleHelperShowLineFeaturesInScan(s1, s2, debugInfo, initGuess);
```



Use the estimated relative pose from `matchScansLine` to transform `s2`. Then, plot both scans to show that the relative pose difference is accurate and the scans overlay to show the same environment.

```
s2t = transformScan(s2,relPose);
clf
plot(s1)
hold on
plot(s2t)
hold off
```



Input Arguments

`currScan` — Current lidar scan readings

`lidarScan` object

Current lidar scan readings, specified as a `lidarScan`object.

Your lidar scan can contain `Inf` and `NaN` values, but the algorithm ignores them.

`refScan` — Reference lidar scan readings

`lidarScan` object

Reference lidar scan readings, specified as a `lidarScan` object.

Your lidar scan can contain `Inf` and `NaN` values, but the algorithm ignores them.

`initialRelPose` — Initial guess of relative pose

`[x y theta]`

Initial guess of the current pose relative to the reference laser scan frame, specified an `[x y theta]` vector. `[x y]` is the translation in meters and `theta` is the rotation in radians.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `"LineMergeThreshold",[0.10 0.2]`

`SmoothnessThreshold` — Threshold to detect line break points in scan
`0.1` (default) | scalar

Threshold to detect line break points in scan, specified as a scalar. Smoothness is defined by calling `diff(diff(scanData))` and assumes equally spaced scan angles. Scan points corresponding to smoothness values higher than this threshold are considered break points. For lidar scan data with a higher noise level, increase this threshold.

`MinPointsPerLine` — Minimum number of scan points in each line feature
`10` (default) | positive integer greater than 3

Minimum number of scan points in each line feature, specified as a positive integer greater than 3.

A line feature cannot be identified from a set of scan points if the number of points in that set is below this threshold. When the lidar scan data is noisy, setting this property too small may result in low-quality line features being identified and skew the matching result. On the other hand, some key line features may be missed if this number is set too large.

`LineMergeThreshold` — Threshold on line parameters to merge line features
`[0.05 0.1]` (default) | two-element vector `[rho alpha]`

Threshold on line parameters to merge line features, specified as a two-element vector `[rho alpha]`. A line is defined by two parameters:

- `rho` –– Distance from the origin to the line along a vector perpendicular to the line, specified in meters.
- `alpha` –– Angle between the `x`-axis and the `rho` vector, specified in radians.

If the difference between these parameters for two line features is below the given threshold, the line features are merged.

`MinCornerPromenance` — Lower bound on prominence value to detect a corner
`0.05` (default) | positive scalar

Lower bound on prominence value to detect a corner, specified as a positive scalar.

Prominence measures how much a local extrema stands out in the lidar data. Only values higher than this lower bound are considered a corner. Corners help identify line features, but are not part of the feature itself. For noisy lidar scan data, increase this lower bound.

`CompatibilityScale` — Scale used to adjust the compatibility thresholds for feature association
`0.0005` (default) | positive scalar

Scale used to adjust the compatibility thresholds for feature association, specified as a positive scalar. A lower scale means tighter compatibility threshold for associating features. If no features are found

in lidar data with obvious line features, increase this value. For invalid feature matches, reduce this value.

Output Arguments

`relpose` — Pose of current scan

`[x y theta]`

Pose of current scan relative to the reference scan, returned as `[x y theta]`, where `[x y]` is the translation in meters and `theta` is the rotation in radians.

`stats` — Scan matching information

structure

Scan matching information, returned as a structure with the following fields:

- `Covariance` –– 3-by-3 matrix representing the covariance of the relative pose estimation. The `matScansLine` function does not provide covariance between the `(x,y)` and the `theta` components of the relative pose. Therefore, the matrix follows the pattern: `[Cxx, Cxy 0; Cyx Cyy 0; 0 0 Ctheta]`.
- `ExitFlag` –– Scalar value indicating the exit condition of the solver:
  - `0` –– No error.
  - `1` –– Insufficient number of line features (< 2) are found in one or both of the scans. Consider using different scans with more line features.
  - `2` –– Insufficient number of line feature matches are identified. This may indicate the `initialRelPose` is invalid or scans are too far apart.

`debugInfo` — Debugging information for line-based scan matching result

structure

Debugging information for line-based scan matching result, returned as a structure with the following fields:

- `ReferenceFeatures` –– Line features extracted from the reference scan as an *n*-by-2 matrix. Each line feature is represented as `[rho alpha]` for the parametric equation, *rho* = *x*•cos(*alpha*) + *y*•sin(*alpha*).
- `ReferenceScanMask` –– Mask indicating which points in the reference scan are used for each line feature as an *n*-by-*p* matrix. Each row corresponds to a row in `ReferenceFeatures` and contains zeros and ones for each point in `refScan`.
- `CurrentFeatures` –– Line features extracted from the current scan as an *n*-by-2 matrix. Each line feature is represented as `[rho alpha]` for the parametric equation, *rho* = *x*•cos(*alpha*) + *y*•sin(*alpha*).
- `CurrentScanMask` –– Mask indicating which points in the current scan are used for each line feature as an *n*-by-*p* matrix. Each row corresponds to a row in `ReferenceFeatures` and contains zeros and ones for each point in `refScan`.
- `MatchHypothesis` –– Best line feature matching hypothesis as an *n* element vector, where *n* is the number of line features in `CurrentFeatures`. Each element represents the corresponding feature in `ReferenceFeatures`and gives the index of the matched feature in `ReferenceFeatures` is an index match the

- `MatchValue` –– Scalar value indicating a score for each `MatchHypothesis`. A lower value is considered a better match. If two elements of `MatchHypothesis` have the same index, the feature with a lower score is used.

References

[1] Neira, J., and J.d. Tardos. “Data Association in Stochastic Mapping Using the Joint Compatibility Test.” *IEEE Transactions on Robotics and Automation* 17, no. 6 (2001): 890–97. https://doi.org/10.1109/70.976019.

[2] Shen, Xiaotong, Emilio Frazzoli, Daniela Rus, and Marcelo H. Ang. “Fast Joint Compatibility Branch and Bound for Feature Cloud Matching.” *2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2016. https://doi.org/10.1109/iros.2016.7759281.

See Also

`matchScans` | `matchScansGrid` | `lidarSLAM`

Topics

“Estimate Robot Pose with Scan Matching”
“Implement Simultaneous Localization And Mapping (SLAM) with Lidar Scans”

Introduced in R2020a

optimizePoseGraph

Optimize nodes in pose graph

Syntax

```
updatedGraph = optimizePoseGraph(poseGraph)
updatedGraph = optimizePoseGraph(poseGraph,solver)
[updatedGraph,solutionInfo] = optimizePoseGraph( ___ )
[ ___ ] = optimizePoseGraph( ___ ,Name,Value)
```

Description

`updatedGraph = optimizePoseGraph(poseGraph)` adjusts the poses based on their edge constraints defined in the specified graph to improve the overall graph. You optimize either a 2-D or 3-D pose graph. The returned pose graph has the same topology with updated nodes.

This pose graph optimization assumes all edge constraints and loop closures are valid. To consider trimming edges based on bad loop closures, see the `trimLoopClosures` function.

`updatedGraph = optimizePoseGraph(poseGraph,solver)` specifies the solver type for optimizing the pose graph.

`[updatedGraph,solutionInfo] = optimizePoseGraph( ___ )` returns additional statistics about the optimization process in `solutionInfo` using any of the previous syntaxes.

`[ ___ ] = optimizePoseGraph( ___ ,Name,Value)` specifies additional options using one or more `Name,Value` pairs. For example, `'MaxIterations',1000` increases the maximum number of iterations to 1000.

Examples

Optimize a 3-D Pose Graph

Optimize a pose graph based on the nodes and edge constraints. The pose graph used in this example is taken from the MIT Dataset and was generated using information extracted from a parking garage.

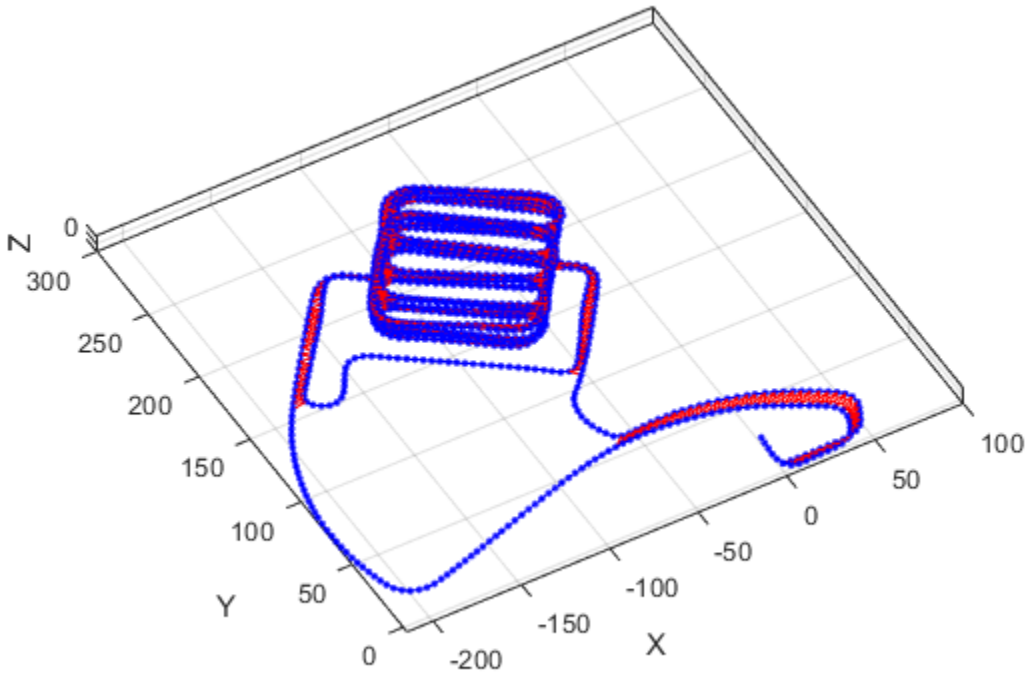
Load the pose graph from the MIT dataset. Inspect the `poseGraph3D` object to view the number of nodes and loop closures.

```
load parking-garage-posegraph.mat pg
disp(pg);
```

```
  poseGraph3D with properties:

              NumNodes: 1661
              NumEdges: 6275
    NumLoopClosureEdges: 4615
     LoopClosureEdgeIDs: [128 129 130 132 133 134 135 137 138 139 140 ... ]
        LandmarkNodeIDs: [1x0 double]
```

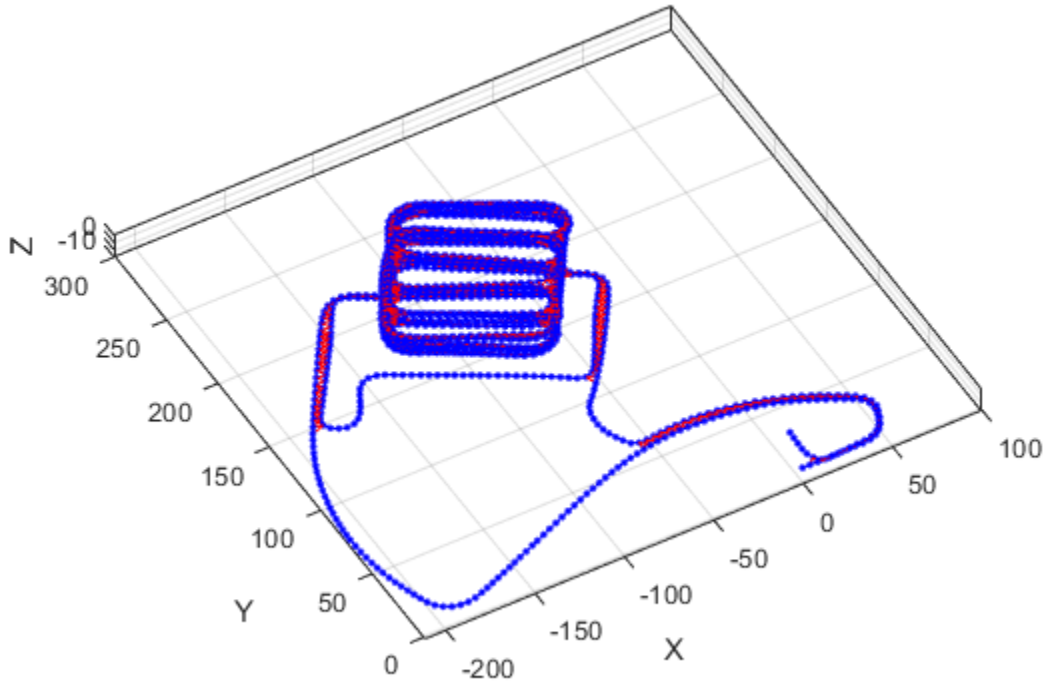
Plot the pose graph with IDs off. Red lines indicate loop closures identified in the dataset.

```
title('Original Pose Graph')
show(pg,'IDs','off');
view(-30,45)
```



Optimize the pose graph. Nodes are adjusted based on the edge constraints and loop closures. Plot the optimized pose graph to see the adjustment of the nodes with loop closures.

```
updatedPG = optimizePoseGraph(pg);
figure
title('Updated Pose Graph')
show(updatedPG,'IDs','off');
view(-30,45)
```



Input Arguments

`poseGraph` — 2-D or 3-D pose graph

`poseGraph` object | `poseGraph3D` object | `digraph` object

2-D or 3-D pose graph, specified as a `poseGraph`, `poseGraph3D`, `digraph` object.

To use the `digraph` object, generate the pose graph using `createPoseGraph` from an `imageviewset` or `pcviewset` object. You must have Computer Vision Toolbox™ and the `solver` must be set to `"builtin-trust-region"`. The `'LoopClosuresToIgnore'` and `'FirstNodePose'` name-value pairs are ignored if specified.

The edges of `digraph` object are described by `affine3d` or `rigid3d` objects.

`solver` — Pose graph solver

`"builtin-trust-region"` (default) | `"g2o-levenberg-marquardt"`

Pose graph solver, specified as either `"builtin-trust-region"` or `"g2o-levenberg-marquardt"`. To tune either solver, use the name-value pair arguments for that solver.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'MaxTime', 300`

Note Depending on the `solver` input, the function supports different name-value pairs.

If the solver input is set to "builtin-trust-region":

`MaxTime` — Maximum time allowed
`500` (default) | positive numeric scalar

Maximum time allowed, specified as the comma-separated pair consisting of `'MaxTime'` and a positive numeric scalar in seconds. The optimizer exits after it exceeds this time.

`GradientTolerance` — Lower bound on norm of gradient
`0.5e-8` (default) | scalar

Lower bound on the norm of the gradient, specified as the comma-separated pair consisting of `'GradientTolerance'` and a scalar. The norm of the gradient is calculated based on the cost function of the optimization. If the norm falls below this value, the optimizer exits.

`FunctionTolerance` — Lower bound on change in cost function
`1e-8` (default) | scalar

Lower bound on the change in the cost function, specified as the comma-separated pair consisting of `'FunctionTolerance'` and a scalar. If the cost function change falls below this value between optimization steps, the optimizer exits.

`StepTolerance` — Lower bound on step size
`1e-12` (default) | scalar

Lower bound on the step size, specified as the comma-separated pair consisting of `'StepTolerance'` and a scalar. If the norm of the optimization step falls below this value, the optimizer exits.

`InitialTrustRegionRadius` — Initial trust region radius
`100` (default) | scalar

Initial trust region radius, specified as a scalar.

`VerboseOutput` — Display intermediate iteration information
`'off'` (default) | `'on'`

Display intermediate iteration information on the MATLAB command line, specified as the comma-separated pair consisting of `'VerboseOutput'` and either `'off'` or `'on'`.

`LoopClosuresToIgnore` — IDs of loop closure edges in pose graph
vector

IDs of loop closure edges in `poseGraph`, specified as the comma-separated pair consisting of `'LoopClosuresToIgnore'` and a vector. To get edge IDs from the pose graph, use `findEdgeID`.

`FirstNodePose` — Pose of first node
`[0 0 0]` or `[0 0 0 1 0 0 0]` (default) | `[x y theta]` | `[x y z qw qx qy qz]`

Pose of the first node in `poseGraph`, specified as the comma-separated pair consisting of `'FirstNodePose'` and a pose vector.

For `poseGraph` (2-D), the pose is an `[x y theta]` vector, which defines the relative *xy*-position and orientation angle, `theta`.

For `poseGraph3D`, the pose is an `[x y z qw qx qy qz]` vector, which defines the relative *xyz*-position and quaternion orientation, `[qw qx qy qz]`.

Note Many other sources for 3-D pose graphs, including `.g2o` formats, specify the quaternion orientation in a different order, for example, `[qx qy qz qw]`. Check the source of your pose graph data before adding nodes to your `poseGraph3D` object.

If the solver input is set to "g2o-levenberg-marquardt":

`MaxIterations` — Maximum number of iterations
`300` (default) | positive integer

Maximum number of iterations, specified as the comma-separated pair consisting of `'MaxIterations'` and a positive integer. The optimizer exits after it exceeds this number of iterations.

`MaxTime` — Maximum time allowed
`500` (default) | positive numeric scalar

Maximum time allowed, specified as the comma-separated pair consisting of `'MaxTime'` and a positive numeric scalar in seconds. The optimizer exits after it exceeds this time.

`FunctionTolerance` — Lower bound on change in cost function
`1e-8` (default) | scalar

Lower bound on the change in the cost function, specified as the comma-separated pair consisting of `'FunctionTolerance'` and a scalar. If the cost function change falls below this value between optimization steps, the optimizer exits.

`VerboseOutput` — Display intermediate iteration information
`'off'` (default) | `'on'`

Display intermediate iteration information on the MATLAB command line, specified as the comma-separated pair consisting of `'VerboseOutput'` and either `'off'` or `'on'`.

`LoopClosuresToIgnore` — IDs of loop closure edges in pose graph
vector

IDs of loop closure edges in `poseGraph`, specified as the comma-separated pair consisting of `'LoopClosuresToIgnore'` and a vector. To get edge IDs from the pose graph, use `findEdgeID`.

`FirstNodePose` — Pose of first node
`[0 0 0]` or `[0 0 0 1 0 0 0]` (default) | `[x y theta]` | `[x y z qw qx qy qz]`

Pose of the first node in `poseGraph`, specified as the comma-separated pair consisting of `'FirstNodePose'` and a pose vector.

For `poseGraph` (2-D), the pose is an `[x y theta]` vector, which defines the relative *xy*-position and orientation angle, `theta`.

For `poseGraph3D`, the pose is an `[x y z qw qx qy qz]` vector, which defines the relative *xyz*-position and quaternion orientation, `[qw qx qy qz]`.

Note Many other sources for 3-D pose graphs, including `.g2o` formats, specify the quaternion orientation in a different order, for example, `[qx qy qz qw]`. Check the source of your pose graph data before adding nodes to your `poseGraph3D` object.

Output Arguments

`updatedGraph` — Optimized 2-D or 3-D pose graph

`poseGraph` object | `poseGraph3D` object

Optimized 2-D or 3-D pose graph, returned as a `poseGraph` or `poseGraph3D` object.

`solutionInfo` — Statistics of optimization process

structure

Statistics of optimization process, returned as a structure with these fields:

- `Iterations` — Number of iterations used in optimization.
- `ResidualError` — Value of cost function when optimizer exits.
- `Exit Flag` — Exit condition for optimizer:
  - `1` — Local minimum found.
  - `2` — Maximum number of iterations reached. See `MaxIterations` name-value pair argument.
  - `3` — Algorithm timed out during operation.
  - `4` — Minimum step size. The step size is below the `StepTolerance` name-value pair argument.
  - `5` — The change in error is below the minimum.
  - `8` — Trust region radius is below the minimum set in `InitialTrustRegionRadius`.

References

[1] Grisetti, G., R. Kummerle, C. Stachniss, and W. Burgard. "A Tutorial on Graph-Based SLAM." *IEEE Intelligent Transportation Systems Magazine*. Vol. 2, No. 4, 2010, pp. 31–43. doi:10.1109/mits.2010.939925.

[2] Carlone, Luca, Roberto Tron, Kostas Daniilidis, and Frank Dellaert. "Initialization Techniques for 3D SLAM: a Survey on Rotation Estimation and its Use in Pose Graph Optimization." *2015 IEEE International Conference on Robotics and Automation (ICRA)*. 2015, pp. 4597–4604.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

- Use this syntax when constructing `poseGraph` or `poseGraph3D` objects for code generation:

`poseGraph = poseGraph('MaxNumEdges',maxEdges,'MaxNumNodes',maxNodes)` specifies an upper bound on the number of edges and nodes allowed in the pose graph when generating code. This limit is only required when generating code.

- The `"g2o-levenberg-marquardt"` `solver` input argument is not supported for code generation.

See Also

Functions
`trimLoopClosures` | `addRelativePose` | `removeEdges` | `edgeNodePairs` | `edgeConstraints` | `findEdgeID` | `nodeEstimates`

Objects
`poseGraph` | `poseGraph3D` | `lidarSLAM`

Topics
“Implement Simultaneous Localization And Mapping (SLAM) with Lidar Scans”
“Implement Online Simultaneous Localization And Mapping (SLAM) with Lidar Scans”

Introduced in R2019b

plotTransforms

Plot 3-D transforms from translations and rotations

Syntax

```
ax = plotTransforms(translations,rotations)
ax = plotTransforms(translations,rotations,Name,Value)
```

Description

`ax = plotTransforms(translations,rotations)` draws transform frames in a 3-D figure window using the specified translations and rotations. The *z*-axis always points upward.

`ax = plotTransforms(translations,rotations,Name,Value)` specifies additional options using name-value pair arguments. Specify multiple name-value pairs to set multiple options.

Input Arguments

`translations` — *xyz*-positions

`[x y z]` vector | matrix of `[x y z]` vectors

xyz-positions specified as a vector or matrix of `[x y z]` vectors. Each row represents a new frame to plot with a corresponding orientation in `rotations`.

Example: `[1 1 1; 2 2 2]`

`rotations` — Rotations of *xyz*-positions

`quaternion` array | matrix of `[w x y z]` quaternion vectors

Rotations of *xyz*-positions specified as a array or *n*-by-4 matrix of `[w x y z]` quaternion vectors. Each element of the array or each row of the matrix represents the rotation of the *xyz*-positions specified in `translations`.

Example: `[1 1 1 0; 1 3 5 0]`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'FrameSize',5`

`FrameSize` — Size of frames and attached meshes

positive numeric scalar

Size of frame and attached meshes, specified as positive numeric scalar.

`InertialZDirection` — Direction of positive *z*-axis of inertial frame

`"up"` (default) | `"down"`

Direction of the positive *z*-axis of inertial frame, specified as either `"up"` or `"down"`. In the plot, the positive *z*-axis always points up.

`MeshFilePath` — File path of mesh file attached to frames
character vector | string scalar

File path of mesh file attached to frames, specified as either a character vector or string scalar. The mesh is attached to each plotted frame at the specified position and orientation. Provided `.stl` are

- `"fixedwing.stl"`
- `"multirotor.stl"`
- `"groundvehicle.stl"`

Example: `'fixedwing.stl'`

`MeshColor` — Color of attached mesh
`"red"` (default) | RGB triplet | string scalar

Color of attached mesh, specified as an RGB triple or string scalar.

Example: `[0 0 1]` or `"green"`

`Parent` — Axes used to plot transforms
`Axes` object | `UIAxes` object

Axes used to plot the pose graph, specified as the comma-separated pair consisting of `'Parent'` and either an `Axes` or `UIAxes` object. See `axes` or `uiaxes`.

Output Arguments

`ax` — Axes used to plot transforms
`Axes` object | `UIAxes` object

Axes used to plot the pose graph, specified as the comma-separated pair consisting of `'Parent'` and either an `Axes` or `UIAxes`object. See `axes` or `uiaxes`.

See Also
`quaternion` | `hom2cart` | `eul2quat` | `tform2quat` | `rotm2quat`

Introduced in R2018b

poseGraphSolverOptions

Solver options for pose graph optimization

Syntax

```
solverOptions = poseGraphSolverOptions(solverType)
```

Description

`solverOptions = poseGraphSolverOptions(solverType)` returns the set of solver options with default values for the specified pose graph solver type.

Examples

Optimize and Trim Loop Closures For 2-D Pose Graphs

Optimize a pose graph based on the nodes and edge constraints. Trim loop closed based on their edge residual errors.

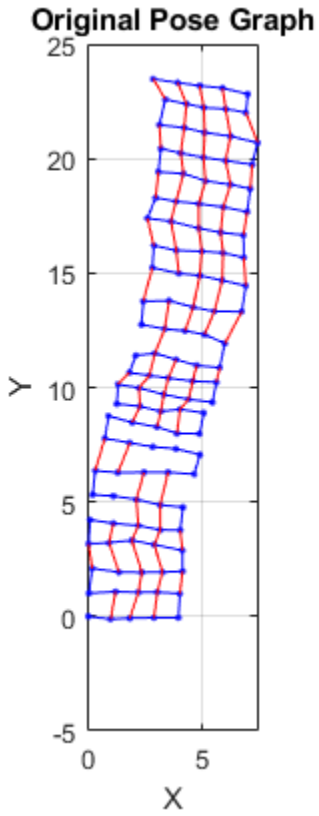
Load the data set that contains a 2-D pose graph. Inspect the `poseGraph` object to view the number of nodes and loop closures.

```
load grid-2d-posegraph.mat pg
disp(pg)
```

```
  poseGraph with properties:

              NumNodes: 120
              NumEdges: 193
    NumLoopClosureEdges: 74
     LoopClosureEdgeIDs: [120 121 122 123 124 125 126 127 128 129 130 ... ]
        LandmarkNodeIDs: [1x0 double]
```

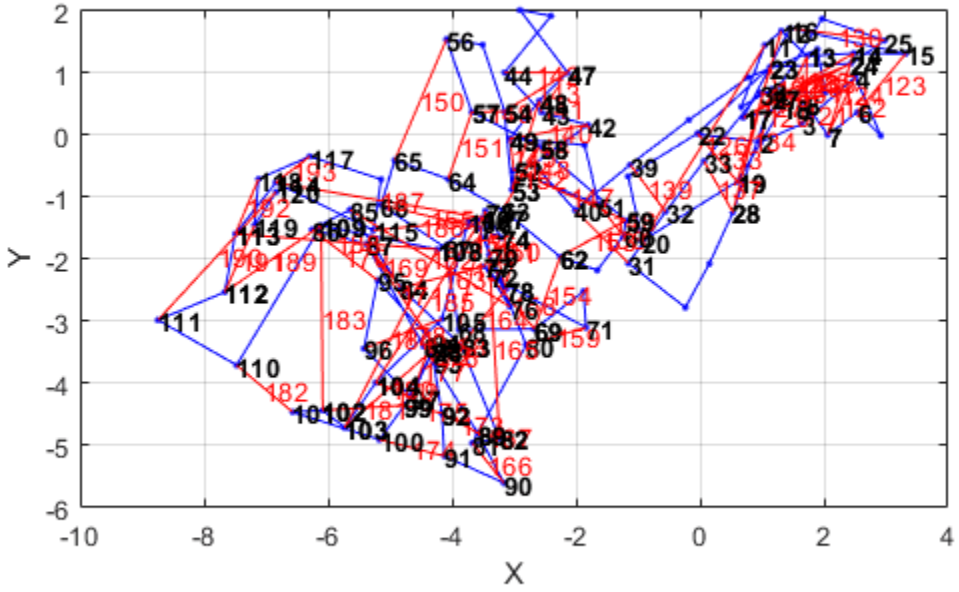
Plot the pose graph with IDs off. Red lines indicate loop closures identified in the dataset. The poses in the graph should follow a grid pattern, but show evidence of drift over time.

```
show(pg,'IDs','off');
title('Original Pose Graph')
```



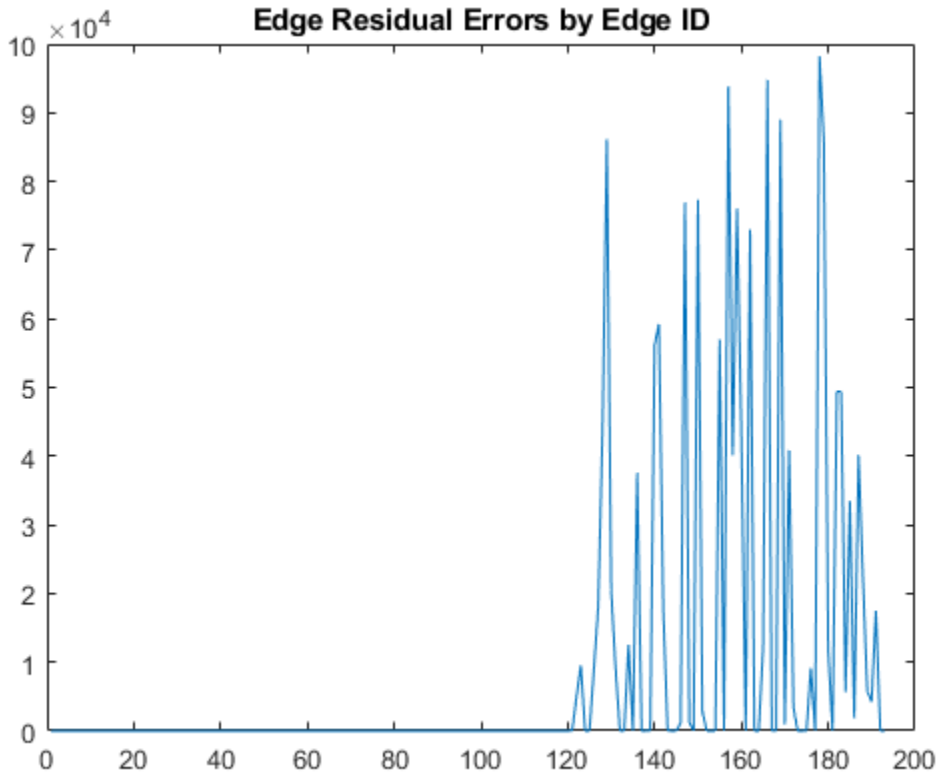
Optimize the pose grap using the `optimizePoseGraph` function. By default, this function uses the `"builtin-trust-region"` solver. Because the pose graph contains some bad loop closures, the resulting pose graph is actual not desirable.

```
pgOptim = optimizePoseGraph(pg);
figure;
show(pgOptim);
```



Look at the edge residual errors for the original pose graph. Large outlier error values at the end indicate bad loop closures.

```
resErrorVec = edgeResidualErrors(pg);
plot(resErrorVec);
title('Edge Residual Errors by Edge ID')
```



Certain loop closures should be trimmed from the pose graph based on their residual error. Use the `trimLoopClosures` function to trim therse bad loop closures. Set the maximum and truncation threshold for the trimmer parameters. This threshold is set based on the measurement accuracy and should be tuned for your system.

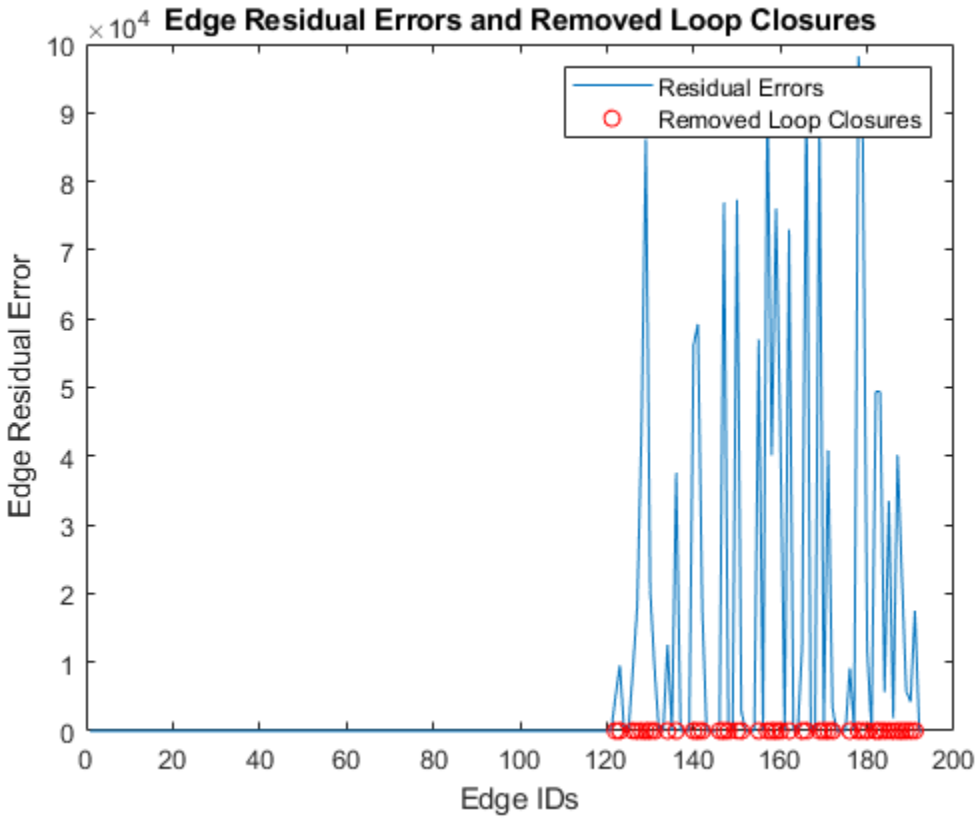
```
trimParams.MaxIterations = 100;
trimParams.TruncationThreshold = 25;

solverOptions = poseGraphSolverOptions;
```

Use the `trimLoopClosures` function with the trimmer parameters and solver options.

```
[pgNew, trimInfo, debugInfo] = trimLoopClosures(pg,trimParams,solverOptions);
```

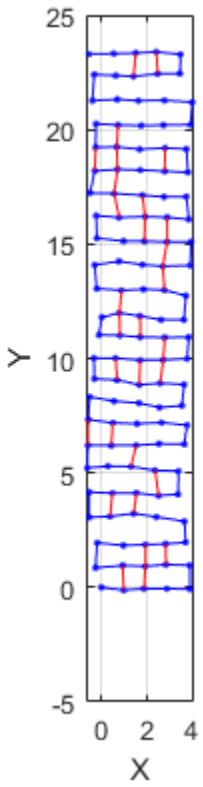
From the `trimInfo` output, plot the loop closures removed from the optimized pose graph. By plotting with the residual errors plot before, you can see the large error loop closures were removed.

```
removedLCs = trimInfo.LoopClosuresToRemove;

hold on
plot(removedLCs,zeros(length(removedLCs)),'or')
title('Edge Residual Errors and Removed Loop Closures')
legend('Residual Errors', 'Removed Loop Closures')
xlabel('Edge IDs')
ylabel('Edge Residual Error')
hold off
```



Show the new pose graph with the bad loop closures trimmed.

```
show(pgNew,"IDs","off");
```



Input Arguments

`solverType` — Pose graph solver type

`'builtin-trust-region'` (default) | `'g2o-levenberg-marquardt'`

Pose graph sovler type, specified as `'builtin-trust-region'` or `'g2o-levenberg-marquardt'`.

The function generates a set of solver options with default values for the specified pose graph solver type:

```
pgSolverTrustRegion = poseGraphSolverOptions('builtin-trust-region')

pgSolverTrustRegion =

TrustRegion (builtin-trust-region-dogleg) options:

               MaxIterations: 300
                     MaxTime: 10
           FunctionTolerance: 1.0000e-08
           GradientTolerance: 5.0000e-09
               StepTolerance: 1.0000e-12
    InitialTrustRegionRadius: 100
               VerboseOutput: 'off'

pgSolverG2o = poseGraphSolverOptions('g2o-levenberg-marquardt')
```

```
pgSolverG2o =

G2oLevenbergMarquardt (g2o-levenberg-marquardt) options:

        MaxIterations: 300
              MaxTime: 10
    FunctionTolerance: 1.0000e-09
        VerboseOutput: 'off'
```

Data Types: `char` | `string`

Output Arguments

`solverOptions` — Pose graph solver options

`poseGraphSolverOptions` parameters

Pose graph solver options, specified as a set of parameters generated by calling the `poseGraphSolverOptions` function. The function generates a set of solver options with default values for the specified pose graph solver type.

If the `solverType` input is set to `"builtin-trust-region"`:

	Default	Description
`MaxIterations`	`300`	Maximum number of iterations, specified as a positive integer. The optimizer exits after it exceeds this number of iterations.
`MaxTime`	`500`	Maximum time allowed, specified as a positive numeric scalar in seconds. The optimizer exits after it exceeds this time.
`FunctionTolerance`	`1e-8`	Lower bound on the change in the cost function, specified as a scalar. If the cost function change falls below this value between optimization steps, the optimizer exits.
`GradientTolerance`	`0.5e-8`	Lower bound on the norm of the gradient, specified as a scalar. The norm of the gradient is calculated based on the cost function of the optimization. If the norm falls below this value, the optimizer exits.
`StepTolerance`	`1e-12`	Lower bound on the step size, specified as a scalar. If the norm of the optimization step falls below this value, the optimizer exits.
`InitialTrustRegionRadius`	`100`	Initial trust region radius, specified as a scalar.
`VerboseOutput`	`'off'` or `'on'`	Display intermediate iteration information on the MATLAB command line.

If the `solver` input is set to `"g2o-levenberg-marquardt"`:

	Default	Description
`MaxIterations`	`300`	Maximum number of iterations, specified as a positive integer. The optimizer exits after it exceeds this number of iterations.
`MaxTime`	`500`	Maximum time allowed, specified as a positive numeric scalar in seconds. The optimizer exits after it exceeds this time.
`FunctionTolerance`	`1e-8`	Lower bound on the change in the cost function, specified as a scalar. If the cost function change falls below this value between optimization steps, the optimizer exits.
`VerboseOutput`	`'off'` or `'on'`	Display intermediate iteration information on the MATLAB command line.

See Also

Functions
`trimLoopClosures` | `edgeResidualErrors` | `edgeResidualErrors` | `removeEdges` | `edgeNodePairs` | `edgeConstraints`

Objects
`poseGraph` | `poseGraph3D` | `lidarSLAM`

Introduced in R2020b

poseplot

3-D pose plot

Syntax

```
poseplot
poseplot(quat)
poseplot(R)
poseplot( ___ ,position)
poseplot( ___ ,frame)
poseplot( ___ ,Name=Value)
poseplot(ax, ___ )
p = poseplot( ___ )
```

Description

`poseplot` plots the pose (position and orientation) at the coordinate origin position with zero rotation. The default navigation frame is the north-east-down (NED) frame.

`poseplot(quat)` plots the pose with orientation specified by a quaternion `quat`. The position by default is `[0 0 0]`.

`poseplot(R)` plots the pose with orientation specified by a rotation matrix `R`. The position by default is `[0 0 0]`.

`poseplot( ___ ,position)` specifies the position of the pose plot.

`poseplot( ___ ,frame)` specifies the navigation frame of the pose plot.

`poseplot( ___ ,Name=Value)` specifies pose patch properties using one or more name-value arguments. For example, `poseplot(PatchFaceColor="r")` plots the pose with red face color. For a list of properties, see PosePatch Properties.

`poseplot(ax, ___ )` specifies the parent axes of the pose plot.

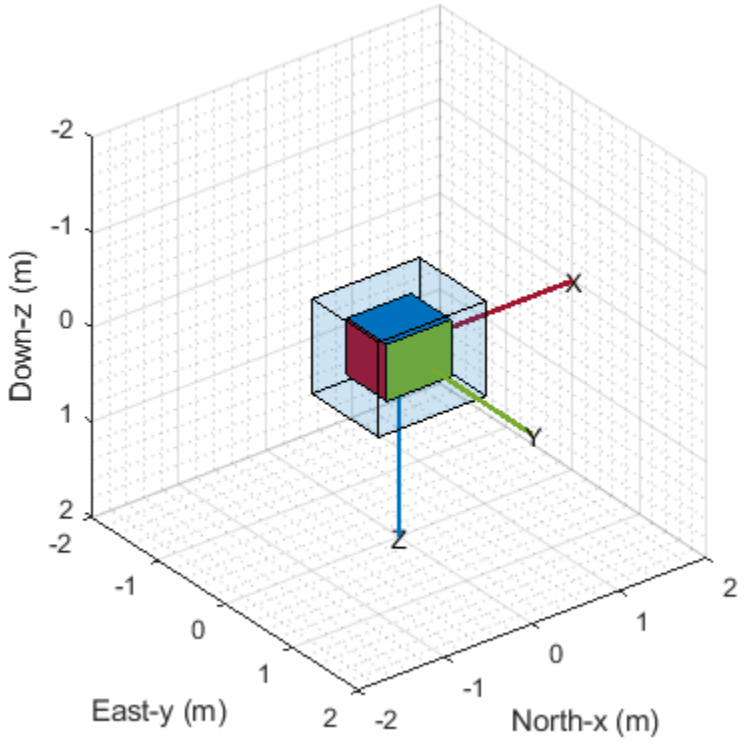
`p = poseplot( ___ )` returns the `PosePatch` object. Use `p` to modify properties of the pose patch after creation. For a list of properties, see PosePatch Properties.

Examples

Visualize Pose Using poseplot

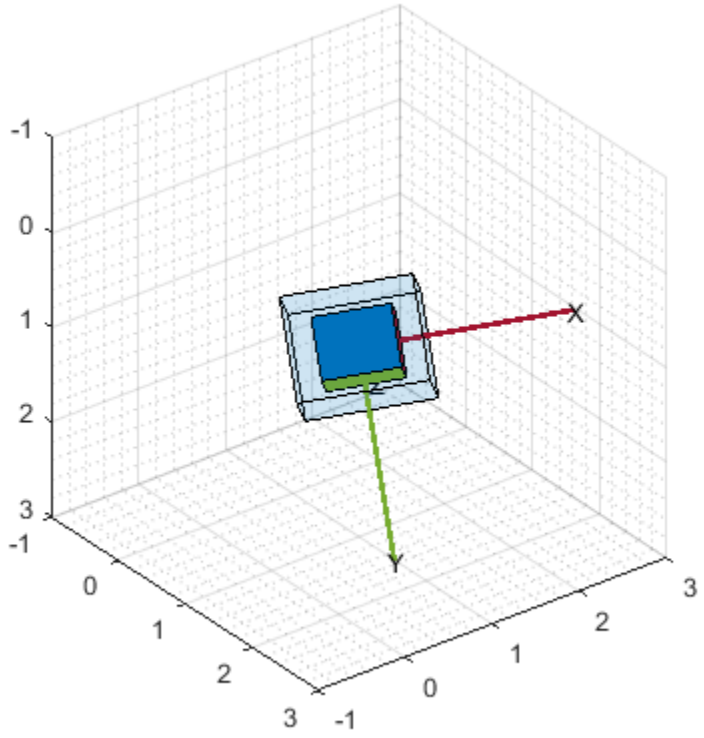
Plot the default pose using the `poseplot` function with default settings.

```
poseplot
xlabel("North-x (m)")
ylabel("East-y (m)")
zlabel("Down-z (m)");
```



Next, plot a pose with specified orientation and position.

```
q = quaternion([35 10 50],"eulerd","ZYX","frame");
position = [1 1 1];
poseplot(q,position)
```

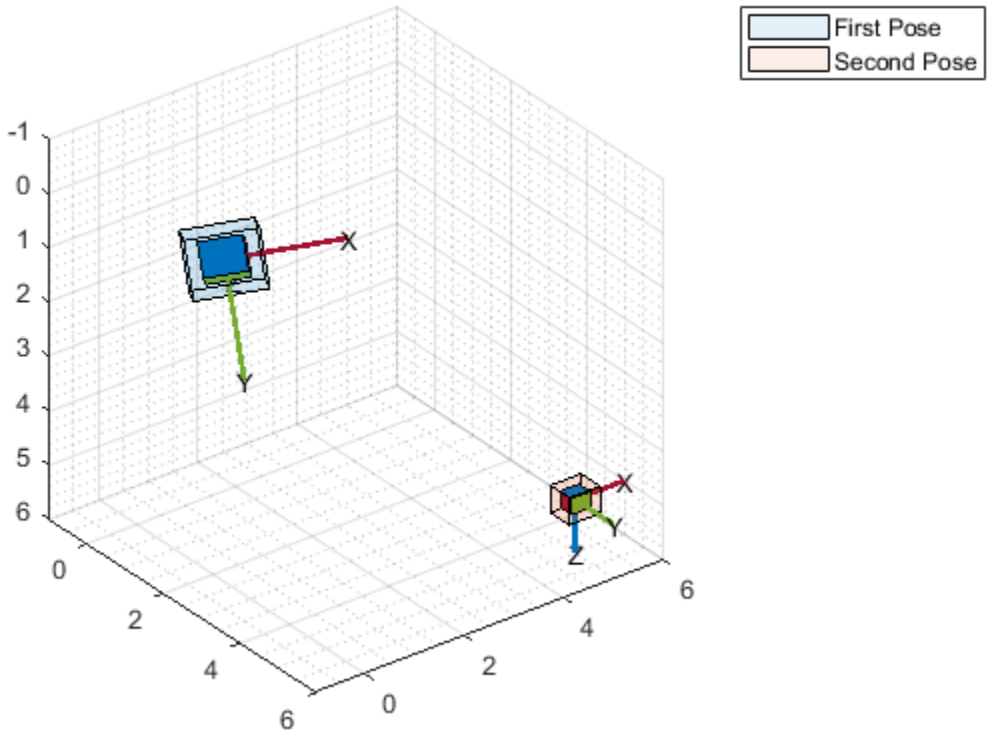


Then, plot a second pose on the figure and return the `PosePatch` object. Plot the second pose with a smaller size by using the `ScaleFactor` name-value argument.

```
hold on
p = poseplot(eye(3),[5 5 5],ScaleFactor=0.5)
```

```
p =
  PosePatch with properties:

    Orientation: [3x3 double]
       Position: [5 5 5]

  Show all properties
```

```
legend("First Pose","Second Pose")
hold off
```



Animate Pose Using `poseplot`

Animate a series of poses using the `poseplot` function. First, define the initial and final positions.

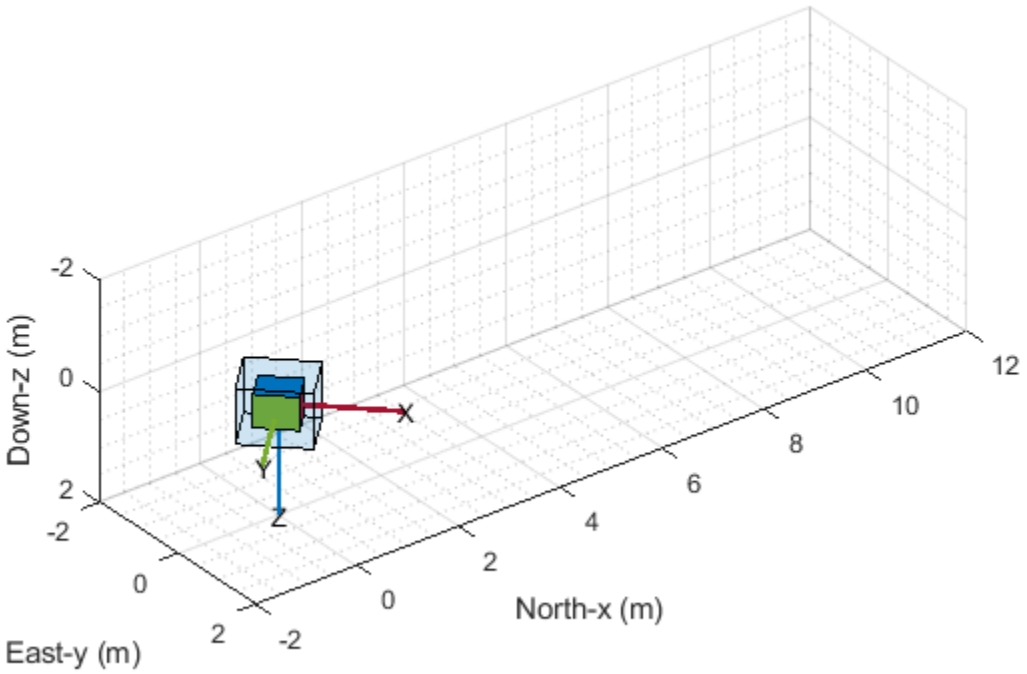
```
ps = [0 0 0];
pf = [10 0 0];
```

Then, define the initial and final orientations using the `quaternion` object.

```
qs = quaternion([45,0,0],'eulerd','ZYX','frame');
qf = quaternion([-45,0,0],'eulerd','ZYX','frame');
```

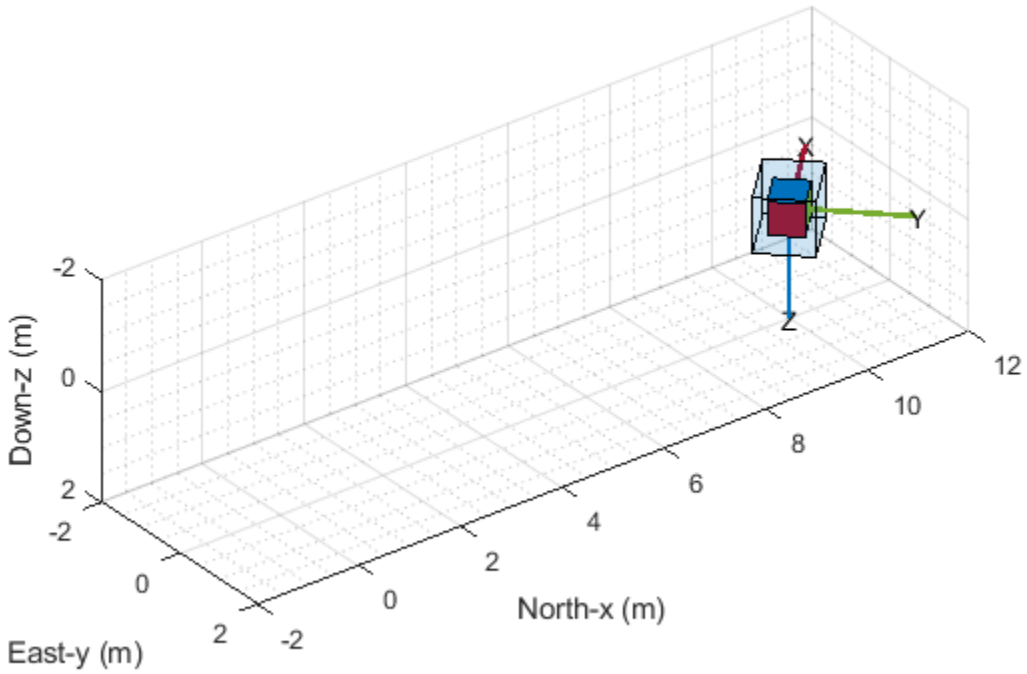
Show the starting pose.

```
patch = poseplot(qs,ps);
ylim([-2 2])
xlim([-2 12])
xlabel("North-x (m)")
ylabel("East-y (m)")
zlabel("Down-z (m)");
```



Change the position and orientation continuously using coefficients, and update the pose using the `set` object function.

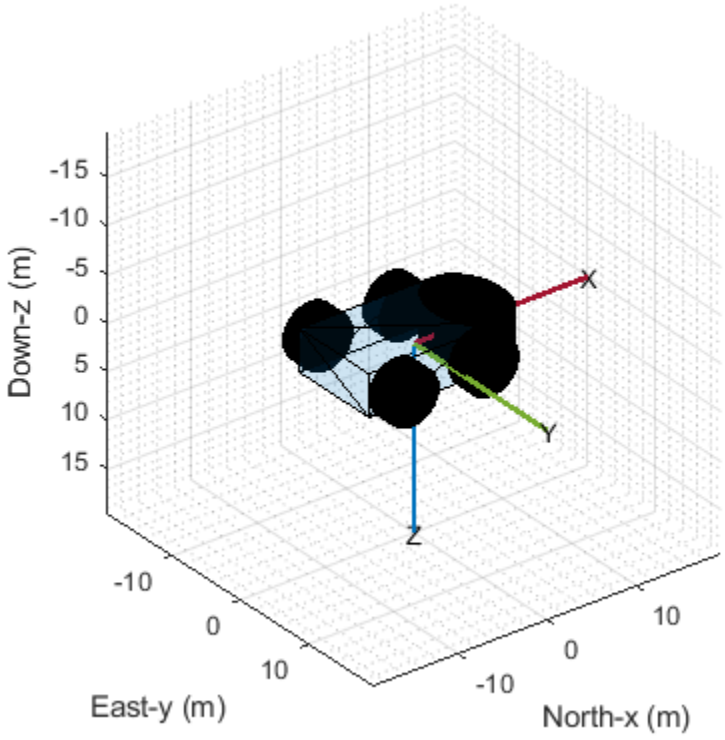
```
for coeff = 0:0.01:1
    q = slerp(qs,qf,coeff);
    position = ps + (pf - ps)*coeff;
    set(patch,Orientation=q,Position=position);
    drawnow
end
```



Show Poses with Meshes

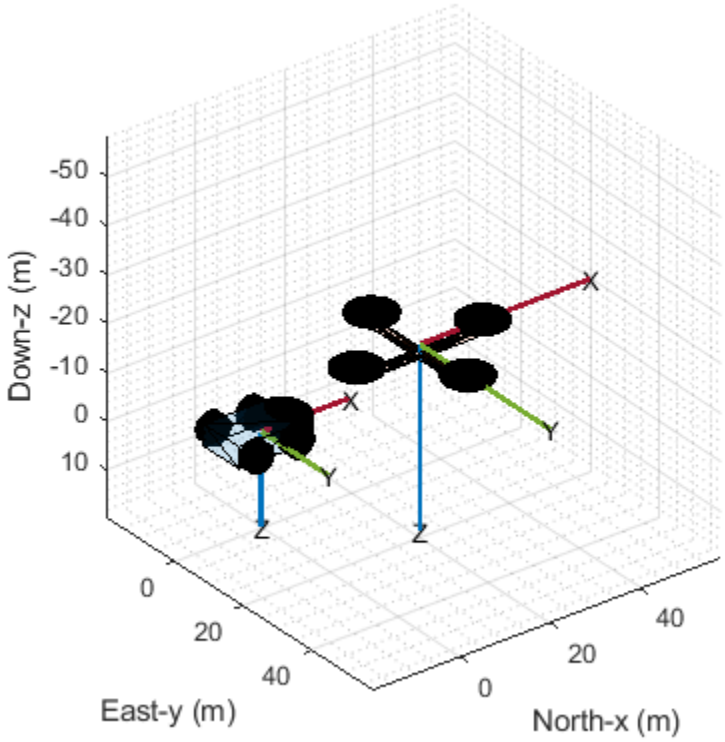
Plot orientations and positions in meshes using the `poseplot` function. First, plot a ground vehicle at the origin with zero rotation.

```
poseplot(ones("quaternion"),[0 0 0],MeshFileName="groundvehicle.stl",ScaleFactor=0.3);
xlabel("North-x (m)")
ylabel("East-y (m)")
zlabel("Down-z (m)")
```



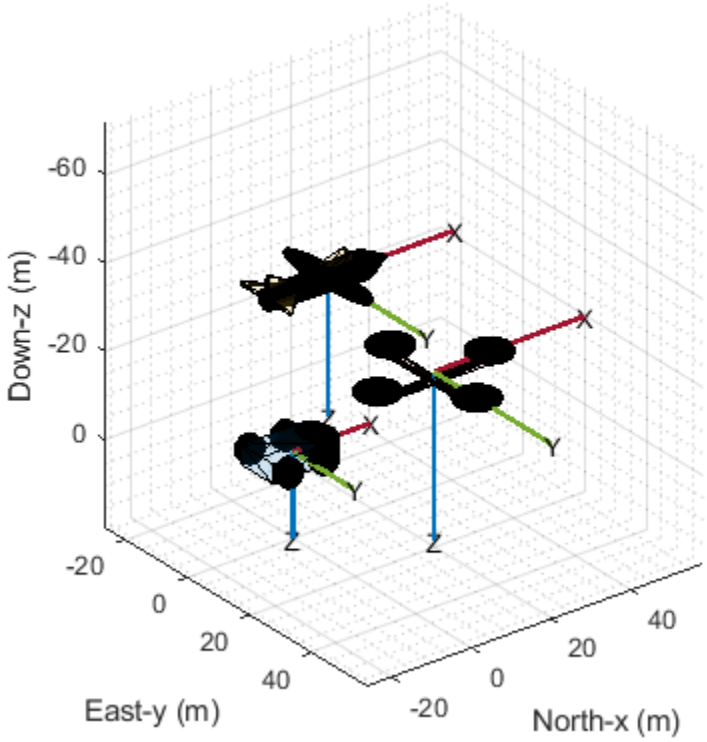
Second, plot a rotor at the position `[20 20 -20]` with zero rotation.

```
hold on
poseplot(ones("quaternion"),[20 20 -20],MeshFileName="multirotor.stl",ScaleFactor=0.2);
```



Lastly, plot a fixed-wing aircraft at the position `[5 5 -40]` with zero rotation.

```
poseplot(ones("quaternion"),[5 5 -40],MeshFileName="fixedwing.stl",ScaleFactor=0.4);
view([-37.8 28.4])
hold off
```



Input Arguments

`quat` — Quaternion

`quaternion` object

Quaternion, specified as a `quaternion` object.

`R` — Rotation matrix

3-by-3 orthonormal matrix

Rotation matrix, specified as a 3-by-3 orthonormal matrix.

Example: `eye(3)`

`position` — Position of pose plot

three-element real-valued vector

Position of the pose plot, specified as a three-element real-valued vector.

Example: `[1 3 4]`

`frame` — Navigation frame of pose plot

`"NED"` (default) | `"ENU"`

Navigation frame of the pose plot, specified as `"NED"` for the north-east-down frame or `"ENU"` for the east-north-up frame.

When the parent axes status is `hold off`, specifying the NED navigation frame reverses the y- and z-axes in the figure by setting the `YDir` and `ZDir` properties of the parent axes.

ax — Parent axes of pose plot
`Axes` object

Parent axes of the pose plot, specified as an `Axes` object. If you do not specify the axes, the `poseplot` function uses the current axes.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

The `PosePatch` properties listed here are only a subset. For a complete list, see PosePatch Properties.

Example: `poseplot(PatchFaceAlpha=0.1)`

`Orientation` — Orientation of pose plot
`quaternion` object (default) | rotation matrix

Orientation of the pose plot, specified as a `quaternion` object or a rotation matrix.

`Position` — Position of pose plot
`[0 0 0]` (default) | three-element real-valued vector

Position of the pose plot, specified as a three-element real-valued vector.

`MeshFileName` — Name of STL mesh file
string scalar | character vector

Name of Standard Triangle Language (STL) mesh file, specified as a string scalar or a character vector containing the name of the mesh file. When you specify this argument, the `poseplot` function plots the mesh instead of the orientation box.

`ScaleFactor` — Scale factor of pose plot
`1` (default) | nonnegative scalar

Scale factor of the pose plot, specified as a nonnegative scalar. The scale factor controls the size of the orientation box. When you specify the `MeshFileName` argument, the scale factor also changes the scale of the mesh.

`PatchFaceColor` — Patch face color
`[0 0 0]` (default) | RGB triplet | hexadecimal color code | `"r"` | `"g"` | `"b"` | ...

Patch face color, specified as an RGB triplet, a hexadecimal color code, a color name, or a short name.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0, 1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (`#`) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `"#FF8800"`, `"#ff8800"`, `"#F80"`, and `"#f80"` are equivalent.

Here is a list of commonly used colors and their corresponding values.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`"red"`	`"r"`	`[1 0 0]`	`"#FF0000"`	
`"green"`	`"g"`	`[0 1 0]`	`"#00FF00"`	
`"blue"`	`"b"`	`[0 0 1]`	`"#0000FF"`	
`"cyan"`	`"c"`	`[0 1 1]`	`"#00FFFF"`	
`"magenta"`	`"m"`	`[1 0 1]`	`"#FF00FF"`	
`"yellow"`	`"y"`	`[1 1 0]`	`"#FFFF00"`	
`"black"`	`"k"`	`[0 0 0]`	`"#000000"`	
`"white"`	`"w"`	`[1 1 1]`	`"#FFFFFF"`	

`PatchFaceAlpha` — Patch face transparency
`0.1` (default) | scalar in range `[0, 1]`

Patch face transparency, specified as a scalar in range `[0, 1]`. A value of `1` is fully opaque and `0` is completely transparent.

Output Arguments

`p` — Pose patch object
`PosePatch` object

Pose patch object, returned as a `PosePatch` object. You can use the returned object to query and modify properties of the plotted pose. For a list of properties, see PosePatch Properties.

See Also

PosePatch Properties

Introduced in R2021b

PosePatch Properties

Pose plot appearance and behavior

Description

`PosePatch` properties control the appearance and behavior of a `PosePatch` object. By changing property values, you can modify certain aspects of the pose plot. Use dot notation to query and set properties. To create a `PosePatch` object, use the `poseplot` function.

```
p = poseplot;
c = p.PatchFaceColor;
p.PatchFaceColor = "red";
```

Properties

Position and Orientation

`Orientation` — Orientation of pose plot

`quaternion` object (default) | rotation matrix

Orientation of the pose plot, specified as a `quaternion` object or a rotation matrix.

`Position` — Position of pose plot

`[0 0 0]` (default) | three-element real-valued vector

Position of the pose plot, specified as a three-element real-valued vector.

Color and Styling

`ScaleFactor` — Scale factor of pose plot

`1` (default) | nonnegative scalar

Scale factor of the pose plot, specified as a nonnegative scalar. The scale factor controls the size of the orientation box. When you specify the `MeshFileName` argument, the scale factor also changes the scale of the mesh.

`PatchFaceColor` — Patch face color

`[0 0 0]` (default) | RGB triplet | hexadecimal color code | `"r"` | `"g"` | `"b"` | ...

Patch face color, specified as an RGB triplet, a hexadecimal color code, a color name, or a short name.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0, 1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (`#`) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `"#FF8800"`, `"#ff8800"`, `"#F80"`, and `"#f80"` are equivalent.

Here is a list of commonly used colors and their corresponding values.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
"red"	"r"	[1 0 0]	"#FF0000"	
"green"	"g"	[0 1 0]	"#00FF00"	
"blue"	"b"	[0 0 1]	"#0000FF"	
"cyan"	"c"	[0 1 1]	"#00FFFF"	
"magenta"	"m"	[1 0 1]	"#FF00FF"	
"yellow"	"y"	[1 1 0]	"#FFFF00"	
"black"	"k"	[0 0 0]	"#000000"	
"white"	"w"	[1 1 1]	"#FFFFFF"	

`MeshFileName` — Name of STL mesh file

string scalar | character vector

Name of Standard Triangle Language (STL) mesh file, specified as a string scalar or a character vector containing the name of the mesh file. When you specify this argument, the `poseplot` function plots the mesh instead of the orientation box.

`PatchFaceAlpha` — Patch face transparency

`0.1` (default) | scalar in range `[0, 1]`

Patch face transparency, specified as a scalar in range `[0, 1]`. A value of `1` is fully opaque and `0` is completely transparent.

Parent/Children

`Parent` — Parent axes

`Axes` object

Parent axes, specified as an `Axes` object.

`Children` — Children

empty `GraphicsPlaceholder` array | `DataTip` object array

Children, returned as an empty `GraphicsPlaceholder` array or a `DataTip` object array. Currently, this property is not used and is reserved for future use.

Interactivity

`Visible` — Pose plot visibility

`"on"` (default) | `"off"` | on/off logical value

Pose plot visibility, specified as `"on"` or `"off"`, or as numeric or logical `1` (`true`) or `0` (`false`). A value of `"on"` is equivalent to `true`, and `"off"` is equivalent to `false`. Thus, you can use the value of this property as a logical value. The value is stored as an on/off logical value of type `matlab.lang.OnOffSwitchState`.

- `"on"` — Display the object.
- `"off"` — Hide the object without deleting it. You still can access the properties of an invisible object.

`HandleVisibility` — Visibility of pose patch object handle

`"on"` (default) | `"off"` | `"callback"`

Visibility of the pose patch object handle in the `Children` property of the parent, specified as one of these values:

- `"on"` — Object handle is always visible.
- `"off"` — Object handle is invisible at all times. This option is useful for preventing unintended changes by another function. Set `HandleVisibility` to `"off"` to temporarily hide the handle during the execution of that function. Hidden object handles are still valid.
- `"callback"` — Object handle is visible from within callbacks or functions invoked by callbacks, but not from within functions invoked from the command line. This option blocks access to the object at the command line, but permits callback functions to access it.

Standard Chart Properties

`DisplayName` — Pose plot name to display in legend

string scalar | character vector

Pose plot name to display in the legend, specified as a string scalar or character vector. The legend does not display until you call the `legend` command. If you do not specify the display name, then `legend` sets the label using the format `"dataN"`, where `N` is the order of pose plots shown in the axes. You can also directly specify the legend. For example: `legend("Pose1","Pose2")`.

`Type` — Type of pose plot object

`'PosePatch'` (default)

This property is read-only.

Type of pose plot object, returned as `'PosePatch'`. Use this property to find all objects of a given type within a plotting hierarchy, for example, searching for the type using the `findobj` function.

`Annotation` — Control for including or excluding object from legend

`Annotation` object

This property is read-only.

Control for including or excluding the object from a legend, returned as an `Annotation` object. Set the underlying `IconDisplayStyle` property to one of these values:

- `"on"` — Include the object in the legend (default).
- `"off"` — Do not include the object in the legend.

For example, to exclude a graphics object, `go`, from the legend, set the `IconDisplayStyle` property to `"off"`.

```
go.Annotation.LegendInformation.IconDisplayStyle = "off";
```

Alternatively, you can control the items in a legend using the `legend` function.

`SeriesIndex` — Pose plot series index

1 (default) | nonnegative integer

Pose plot series index, specified as a nonnegative integer. Use this property to reassign the marker colors of several `PosePatch` objects so that they match each other. By default, the `SeriesIndex`

property of a `PosePatch` object is a number that corresponds to the order of creation of the object, starting at `0`.

MATLAB uses the number to calculate indices for assigning colors when you call plotting functions if you do not specify the color directly. The indices refer to the rows of the arrays stored in the `ColorOrder` property of the axes.

See Also

`poseplot`

Introduced in R2021b

pseudoranges

Pseudoranges between GNSS receiver and satellites

Syntax

```
p = pseudoranges(recPos,satPos)
[p, pdot] = pseudoranges( ___ ,recVel, satVel)
[p, pdot] = pseudoranges( ___ Name,Value)
```

Description

`p = pseudoranges(recPos,satPos)` returns the pseudoranges between the receiver at position `recPos` and the satellites at positions `satPos`.

`[p, pdot] = pseudoranges( ___ ,recVel, satVel)` returns the pseudorange rates `pdot` between the receiver and satellites. Use this syntax with the input arguments in the previous syntax.

`[p, pdot] = pseudoranges( ___ Name,Value)` specifies the measurement noise for the ranges and range rates using name-value arguments. For example, `[p pdot] = pseudoranges(__,'RangeAccuracy',2)` sets the measurement noise in pseudoranges, specified as a scalar standard deviation in meters.

Examples

Get Satellite Pseudoranges for Receiver Position and Velocity

Use the `pseudoranges` function to get the pseudorange and pseudorange rate for given satellite and receiver positions and velocities. Get the satellte positions and velcoties using the `gnssconstellation` function.

Specify a receiver position in geodetic coordinates (latitude, longitude, altitude) and receiver velocity in the local navigation frame.

```
recPos = [42 -71 50];
recVel = [1 2 3];
```

Get the satellite positions for the current time.

```
t = datetime('now');
[gpsSatPos,gpsSatVel] = gnssconstellation(t);
```

Get the the pseudoranges and pseudorange rates between the receiver and satellites.

```
[p,pdot] = pseudoranges(recPos,gpsSatPos,recVel,gpsSatVel);
```

Input Arguments

recPos — Receiver position

three-element vector of the form `[lat lon alt]`

Receiver position in geodetic coordinates, specified as a three-element vector of the form *[latitude longitude altitude]*

Data Types: `single` | `double`

`satPos` — Satellite positions
N-by-3 matrix of scalars

Satellite positions in the Earth-centered Earth-fixed (ECEF) coordinate system in meters, specified as an *N*-by-3 matrix of scalars. *N* is the number of satellites in the constellation.

Data Types: `single` | `double`

`recVel` — Receiver velocity
three-element vector of the form `[vx vy vz]`

Receiver velocity in the local navigation frame using north-east-down (NED) coordinates, specified as a three-element vector of the form *[vx vy vz]*.

Data Types: `single` | `double`

`satVel` — Satellite velocities
N-by-3 matrix of scalars

Satellite velocities in the Earth-centered Earth-fixed (ECEF) coordinate system in meters per second, specified as an *N*-by-3 matrix of scalars. *N* is the number of satellites in the constellation.

Data Types: `single` | `double`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'RangeAccuracy','0.5'` sets the measurement noise of the pseudoranges to 0.5 meters.

`RangeAccuracy` — Measurement noise in pseudoranges
`1` (default) | scalar

Measurement noise in pseudoranges, specified as a scalar standard deviation in meters.

Data Types: `single` | `double`

`RangeRateAccuracy` — Measurement noise in pseudorange rates
`0.02` (default) | scalar

Measurement noise in pseudorange rates, specified as a scalar standard deviation in meters per second.

Data Types: `single` | `double`

Output Arguments

`p` — Pseudoranges between satellites and receiver
n-element vector

Pseudoranges between the satellites and receiver, returned as an *n*-element vector in meters.

Data Types: `single` | `double`

`pdot` — Pseudorange rates between satellites and receiver
`zeros(n,1)` (default) | *n*-element vector

Pseudorange rates between the satellites and receiver, returned as an *n*-element vector in meters per second. If you do not provide velocity inputs, this output is zero.

Data Types: `single` | `double`

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

Objects
`gnssSensor` | `gpsSensor` | `imuSensor`

Functions
`skyplot` | `gnssconstellation` | `lookangles` | `receiverposition`

Introduced in R2021a

quat2axang

Convert quaternion to axis-angle rotation

Syntax

```
axang = quat2axang(quat)
```

Description

`axang = quat2axang(quat)` converts a quaternion, `quat`, to the equivalent axis-angle rotation, `axang`.

Examples

Convert Quaternion to Axis-Angle Rotation

```
quat = [0.7071 0.7071 0 0];
axang = quat2axang(quat)
```

```
axang = 1×4

    1.0000         0         0    1.5708
```

Input Arguments

quat — Unit quaternion

n-by-4 matrix | n-element vector of `quaternion` objects

Unit quaternion, specified as an *n*-by-4 matrix or n-element vector of `quaternion` objects containing *n* quaternions. If the input is a matrix, each row is a quaternion vector of the form $q = [w\ x\ y\ z]$, with *w* as the scalar number.

Example: `[0.7071 0.7071 0 0]`

Output Arguments

axang — Rotation given in axis-angle form

n-by-4 matrix

Rotation given in axis-angle form, returned as an *n*-by-4 matrix of *n* axis-angle rotations. The first three elements of every row specify the rotation axis, and the last element defines the rotation angle (in radians).

Example: `[1 0 0 pi/2]`

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`axang2quat` | `quaternion`

Introduced in R2015a

quat2eul

Convert quaternion to Euler angles

Syntax

```
eul = quat2eul(quat)
eul = quat2eul(quat,sequence)
```

Description

`eul = quat2eul(quat)` converts a quaternion rotation, `quat`, to the corresponding Euler angles, `eul`. The default order for Euler angle rotations is `"ZYX"`.

`eul = quat2eul(quat,sequence)` converts a quaternion into Euler angles. The Euler angles are specified in the axis rotation sequence, `sequence`. The default order for Euler angle rotations is `"ZYX"`.

Examples

Convert Quaternion to Euler Angles

```
quat = [0.7071 0.7071 0 0];
eulZYX = quat2eul(quat)
```

```
eulZYX = 1×3

         0         0    1.5708
```

Convert Quaternion to Euler Angles Using ZYZ Axis Order

```
quat = [0.7071 0.7071 0 0];
eulZYZ = quat2eul(quat,'ZYZ')
```

```
eulZYZ = 1×3

    1.5708   -1.5708   -1.5708
```

Input Arguments

quat — Unit quaternion

n-by-4 matrix | n-element vector of `quaternion` objects

Unit quaternion, specified as an *n*-by-4 matrix or n-element vector of `quaternion` objects containing *n* quaternions. If the input is a matrix, each row is a quaternion vector of the form $q = [w\ x\ y\ z]$, with *w* as the scalar number.

Example: `[0.7071 0.7071 0 0]`

sequence — Axis rotation sequence
`"ZYX"` (default) | `"ZYZ"` | `"XYZ"`

Axis rotation sequence for the Euler angles, specified as one of these string scalars:

- `"ZYX"` (default) – The order of rotation angles is *z*-axis, *y*-axis, *x*-axis.
- `"ZYZ"` – The order of rotation angles is *z*-axis, *y*-axis, *z*-axis.
- `"XYZ"` – The order of rotation angles is *x*-axis, *y*-axis, *z*-axis.

Data Types: `string` | `char`

Output Arguments

`eul` — Euler rotation angles
n-by-3 matrix

Euler rotation angles in radians, returned as an *n*-by-3 array of Euler rotation angles. Each row represents one Euler angle set.

Example: `[0 0 1.5708]`

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also
`eul2quat` | `quaternion`

Introduced in R2015a

quat2rotm

Convert quaternion to rotation matrix

Syntax

```
rotm = quat2rotm(quat)
```

Description

`rotm = quat2rotm(quat)` converts a quaternion `quat` to an orthonormal rotation matrix, `rotm`. When using the rotation matrix, premultiply it with the coordinates to be rotated (as opposed to postmultiplying).

Examples

Convert Quaternion to Rotation Matrix

```
quat = [0.7071 0.7071 0 0];
rotm = quat2rotm(quat)
```

```
rotm = 3×3

    1.0000         0         0
         0   -0.0000   -1.0000
         0    1.0000   -0.0000
```

Input Arguments

`quat` — Unit quaternion

n-by-4 matrix | n-element vector of `quaternion` objects

Unit quaternion, specified as an *n*-by-4 matrix or n-element vector of objects containing *n* quaternions. If the input is a matrix, each row is a quaternion vector of the form $q = [w\ x\ y\ z]$, with w as the scalar number.

Example: `[0.7071 0.7071 0 0]`

Output Arguments

`rotm` — Rotation matrix

3-by-3-by-*n* matrix

Rotation matrix, returned as a 3-by-3-by-*n* matrix containing *n* rotation matrices. Each rotation matrix has a size of 3-by-3 and is orthonormal. When using the rotation matrix, premultiply it with the coordinates to be rotated (as opposed to postmultiplying).

Example: `[0 0 1; 0 1 0; -1 0 0]`

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`rotm2quat` | `quaternion`

Introduced in R2015a

quat2tform

Convert quaternion to homogeneous transformation

Syntax

```
tform = quat2tform(quat)
```

Description

`tform = quat2tform(quat)` converts a quaternion, `quat`, to a homogeneous transformation matrix, `tform`. When using the transformation matrix, premultiply it with the coordinates to be transformed (as opposed to postmultiplying).

Examples

Convert Quaternion to Homogeneous Transformation

```
quat = [0.7071 0.7071 0 0];
tform = quat2tform(quat)
```

```
tform = 4×4

    1.0000         0         0         0
         0   -0.0000   -1.0000         0
         0    1.0000   -0.0000         0
         0         0         0    1.0000
```

Input Arguments

quat — Unit quaternion

n-by-4 matrix | n-element vector of `quaternion` objects

Unit quaternion, specified as an *n*-by-4 matrix or n-element vector of `quaternion` objects containing *n* quaternions. If the input is a matrix, each row is a quaternion vector of the form $q = [w\ x\ y\ z]$, with *w* as the scalar number.

Example: `[0.7071 0.7071 0 0]`

Output Arguments

tform — Homogeneous transformation

4-by-4-by-*n* matrix

Homogeneous transformation matrix, returned as a 4-by-4-by-*n* matrix of *n* homogeneous transformations. When using the rotation matrix, premultiply it with the coordinates to be rotated (as opposed to postmultiplying).

Example: `[0 0 1 0; 0 1 0 0; -1 0 0 0; 0 0 0 1]`

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`tform2quat` | `quaternion`

Introduced in R2015a

read

Read data from GPS receiver

Syntax

```
[tt,overrun] = read(gps)
[lla,groundSpeed,course,dops,gpsReceiverTime,timestamp,overrun] = read(gps)
```

Description

`[tt,overrun] = read(gps)` returns the GPS readings in `timetable` format. This is a non blocking read which returns *N* data points in timetable format, where *N* is specified by `SamplesPerRead` property and `timetable` is specified using `OutputFormat` property of `gpsdev` object.

`[lla,groundSpeed,course,dops,gpsReceiverTime,timestamp,overrun] = read(gps)` returns matrices of measurements from the GPS. This is a non blocking read which returns *N* data points in matrix format, where *N* is specified by `SamplesPerRead` property and `matrix` is specified using `OutputFormat` property of the `gpsdev` object.

Examples

Read Data from GPS Receiver as Timetable

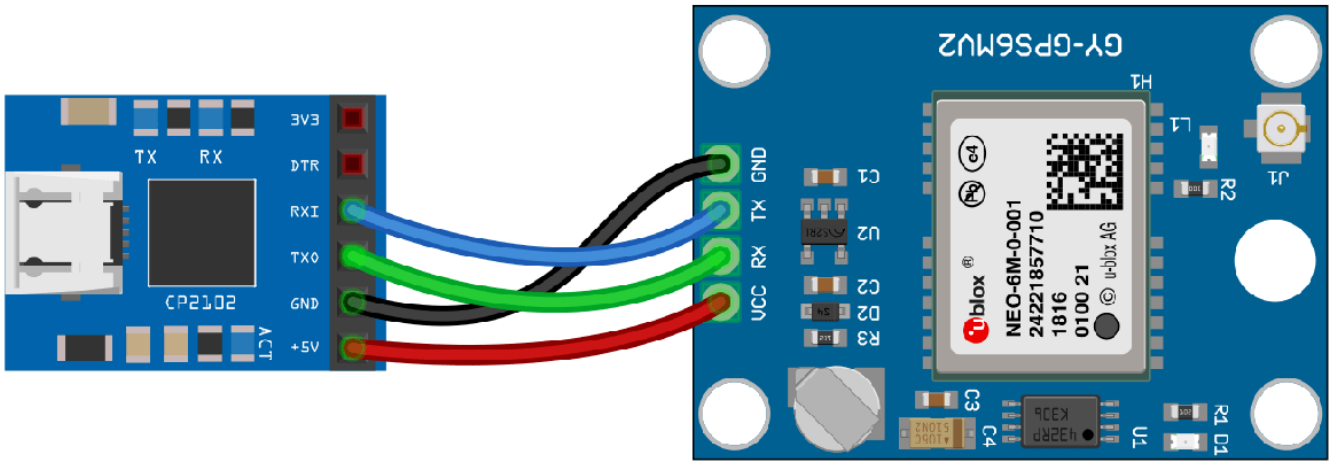
Read data from the GPS receiver connected to the host computer on a specific serial port.

Required Hardware

To run this example, you need:

- UBlox Neo-6M GPS module
- GPS antenna
- USB to UART module
- USB cable
- Connecting wires

Hardware Connection



Connect the pins on the UBlox Neo-6M GPS module to the pins on your USB to UART module. The connections are:

- VCC - +5V
- RX - TXO
- TX - RXI
- GND - GND

Connect the GPS antenna to the GPS module. Connect the USB to UART module to the host computer with a USB cable. GPS Fix can be easily acquired in locations that have a clear view of the sky. Wait for the GPS module to acquire satelite signals (Fix).This can be verified by checking the Fix LED (D1) of your GPS module.

Create GPS Object

Create a `gpsdev` object for the GPS receiver connected to a specific port. Specify the output format of the data as a timetable.

```
gps = gpsdev('COM4','OutputFormat',"timetable")
```

```
gps =
  gpsdev with properties:

                      SerialPort: COM4
                        BaudRate: 9600 (bits/s)

                 SamplesPerRead: 1
                       ReadMode: "latest"
                    SamplesRead: 0
Show all properties all functions
```

Read the GPS data

Read the GPS data and return them as a timetable.

```
[tt,overruns] = read(gps)
```

```
tt=1×5 timetable
             Time                            LLA                GroundSpeed    Course          DOP
    ________________________    _________________________    ___________    ______    ____________

    22-Mar-2021 15:31:15.190    17.47    78.343     449.6      0.25619        NaN      9.31    1.4

overruns = 0
```

Display number of samples read and the samples available in the host buffer.

```
gps.SamplesRead
```

```
ans = 1
```

```
gps.SamplesAvailable
```

```
ans = 0
```

Release the GPS object to configure the non tunable properties. The release function also clears the buffer and resets the `SamplesRead` and `SamplesAvailable` properties.

```
release(gps)
```

Specify the number of samples per read to 2. Read the GPS data.

```
gps.SamplesPerRead = 2;
read(gps)
```

```
ans=2×5 timetable
             Time                            LLA                GroundSpeed    Course          DOP
    ________________________    _________________________    ___________    ______    ____________

    22-Mar-2021 15:31:17.178    17.47    78.343       450     0.063791        NaN      9.32    1.4
    22-Mar-2021 15:31:17.178    17.47    78.343       450     0.063791        NaN      9.32    1.4
```

Display number of samples read and the samples available in the host buffer.

```
gps.SamplesRead
```

```
ans = 1
```

```
gps.SamplesAvailable
```

```
ans = 0
```

Clean Up

When the connection is no longer needed, clear the associated object.

```
delete(gps);
clear gps;
```

Read Data from GPS Receiver as Matrix

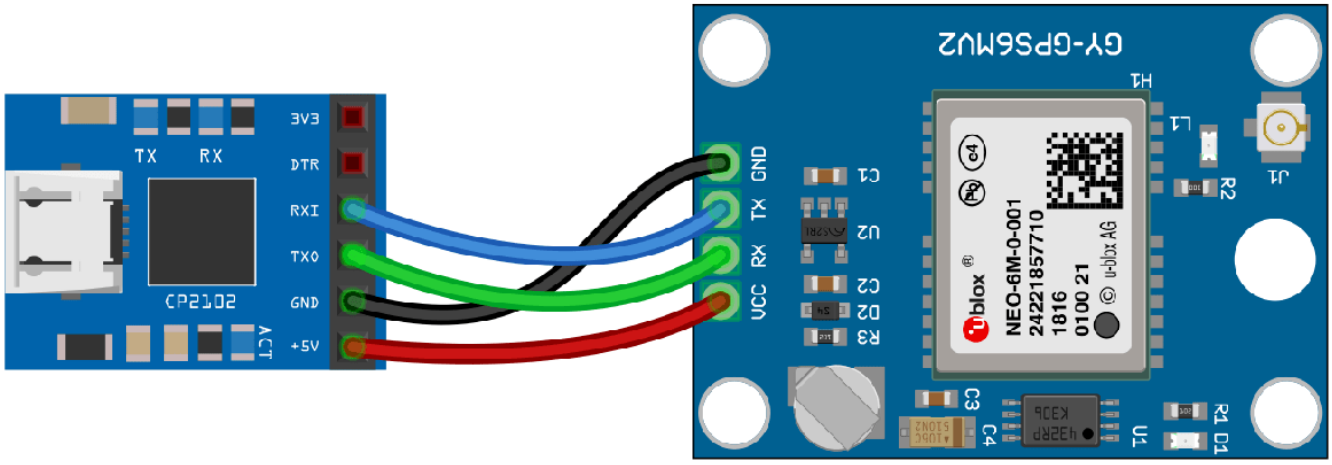
Read data from the GPS receiver connected to the host computer using `serialport` object.

Required Hardware

To run this example, you need:

- UBlox Neo-6M GPS module
- GPS antenna
- USB to UART module
- USB cable
- Connecting wires

Hardware Connection



Connect the pins on the UBlox Neo-6M GPS module to the pins on your USB to UART module. The connections are:

- VCC - +5V
- RX - TXO
- TX - RXI
- GND - GND

Connect the GPS antenna to the GPS module. Connect the USB to UART module to the host computer with a USB cable. GPS Fix can be easily acquired in locations that have a clear view of the sky. Wait for the GPS module to acquire satelite signals (Fix).This can be verified by checking the Fix LED (D1) of your GPS module.

Create GPS Object

Connect to the GPS receiver using `serialport` object. Specify the port name and the baud rate. Specify the output format of the data as matrix.

```
s = serialport('COM4',9600);
gps = gpsdev(s,'OutputFormat',"matrix")
```

```
gps =
  gpsdev with properties:
```

```
                          SerialPort: COM4
                            BaudRate: 9600 (bits/s)

                     SamplesPerRead: 1
                           ReadMode: "latest"
                        SamplesRead: 0
Show all properties all functions
```

Read the GPS data

Read the GPS data and return them as matrices.

```
[lla,speed,course,dops,gpsReceiverTime,timestamp,overruns] = read(gps)
```

```
lla = 1×3

   NaN   NaN   NaN


speed = NaN

course = NaN

dops = 1×3

   NaN   NaN   NaN


gpsReceiverTime = datetime
   NaT


timestamp = datetime
   22-Mar-2021 03:41:00.274


overruns = 1
```

Display number of samples read and the samples available in the host buffer.

```
gps.SamplesRead
```

```
ans = 1
```

```
gps.SamplesAvailable
```

```
ans = 0
```

Flush all GPS data accumulated in the buffers and reset the `SamplesRead` and `SamplesAvailable` properties.

```
flush(gps)
```

Display number of samples read and the samples available in the host buffer.

```
gps.SamplesRead
```

```
ans = 0
```

```
gps.SamplesAvailable
```

```
ans = 0
```

Clean Up

When the connection is no longer needed, clear the associated object.

```
delete(gps);
clear gps;
clear s;
```

Input Arguments

`gps` — GPS sensor

`gpsdev` object

The GPS sensor, specified as a `gpsdev` object.

Output Arguments

`tt` — GPS data

`timetable`

GPS data, returned as a `timetable`. The `timetable` returned has the following fields:

- LLA (Latitude, Longitude, Altitude)
- Ground Speed
- Course over ground
- Dilution of Precisions(DOPs), VDOP,HDOP,PDOP
- GPS Receiver Time
- Time — System time when the data is read, in `datetime` or `duration` format

Data Types: `timetable`

`lla` — Position in LLA coordinate system

N-by-3 matrix

Position of the GPS receiver in the geodetic latitude, longitude, and altitude (LLA), returned as a real finite *N*-by-3 array. Latitude and longitude are in degrees with North and East being positive. Altitude is in meters.

Data Types: `double`

`groundSpeed` — Speed in m/s

N-by-1 vector

Speed over ground, returned as a real finite *N*-by-1 vector.

Data Types: `double`

`course` — Course over ground

N-by-1 vector

Course over ground relative to true north, returned as a real finite *N*-by-1 vector of values between 0 and 2pi radians.

Data Types: `double`

`dops` — Dilution of precisions
N-by-3 matrix

Dilution of precisions, returned as a real finite *N*-by-3 matrix of the form `[PDOP,HDOP,VDOP]`.

Data Types: `double`

`gpsReceiverTime` — UTC time
N-by-1 vector

UTC time, returned as a *N*-by-1 vector.

Data Types: `datetime`

`timestamp` — Time at which GPS data is read
N-by-1 vector

Time at which GPS data is read, returned as a real finite *N*-by-1 vector. This is the system time. If the `TimeFormat` is `datetime`, the timestamp will be `datetime`. If the `TimeFormat` is a `duration`, the timestamp will be `duration`

- `datetime` — Displays the date and time at which the data is read.
- `duration` — Displays the time elapsed in seconds after the first call of the `read` function or the last execution of the `release` function.

Note If the `SamplesPerRead` is greater than 1, an extrapolation is done on the time value. Hence it might not be precise.

Data Types: `datetime` | `duration`

`overrun` — Overrun
scalar

The number of samples lost between consecutive calls to `read`. The overrun is zero when `ReadMode` is set to `oldest`.

Data Types: `double`

More About

`read` Output

The `gpsdev` object expects GPRMC, GPGGA, and GPGSA sentences as outputs from the GPS receiver to get the required values. The `read` function errors out if these sentences are not available.

The `read` function outputs `NaN` and `NaT` in the following situations:

- If the GPS module does not receive valid data because there is no satellite lock or when GPS does not give a particular value.
- If there is a checksum failure, corresponding data points will be `NaN` for numeric outputs (`lla`, `speed`, `course`, `dops`) and `NaT` for `gpsRecieverTime`. `lla` is taken from GPGGA sentence,

`speed`,`course`, and `gpsRecieverTime` is taken GPRMC sentence and `dops` are taken from GPGSA sentence.

Because `read` function is non blocking, the following is expected:

- If no new data is available, the output of `read` is the previous data. For example, if the delay between subsequent reads is less than the `UpdateRate` of the GPS receiver.

Because GPS data is validated in the first `read` operation, it might take more time compared to the subsequent `read` operations.

See Also

Objects
`gpsdev`

Functions
`flush` | `release` | `info` | `writeBytes`

Introduced in R2020b

readBinaryOccupancyGrid

Read binary occupancy grid

Syntax

```
map = readBinaryOccupancyGrid(msg)
map = readBinaryOccupancyGrid(msg,thresh)
map = readBinaryOccupancyGrid(msg,thresh,val)
```

Description

`map = readBinaryOccupancyGrid(msg)` returns a `binaryOccupancyMap` object by reading the data inside a ROS message, `msg`, which must be a `'nav_msgs/OccupancyGrid'` message. All message data values greater than or equal to the occupancy threshold are set to occupied, `1`, in the map. All other values, including unknown values (`-1`) are set to unoccupied, `0`, in the map.

Note The `msg` input is an `'nav_msgs/OccupancyGrid'` ROS message. For more info, see `OccupancyGrid`.

`map = readBinaryOccupancyGrid(msg,thresh)` specifies a threshold, `thresh`, for occupied values. All values greater than or equal to the threshold are set to occupied, `1`. All other values are set to unoccupied, `0`.

`map = readBinaryOccupancyGrid(msg,thresh,val)` specifies a value to set for unknown values (`-1`). By default, all unknown values are set to unoccupied, `0`.

Input Arguments

`msg` — `'nav_msgs/OccupancyGrid'` ROS message

`OccupancyGrid` object handle

`'nav_msgs/OccupancyGrid'` ROS message, specified as a `OccupancyGrid` object handle.

`thresh` — Threshold for occupied values

50 (default) | scalar

Threshold for occupied values, specified as a scalar. Any value greater than or equal to the threshold is set to occupied, `1`. All other values are set to unoccupied, `0`.

Data Types: `double`

`val` — Value to replace unknown values

0 (default) | 1

Value to replace unknown values, specified as either `0` or `1`. Unknown message values (`-1`) are set to the given value.

Data Types: `double` | `logical`

Output Arguments

`map` — Binary occupancy grid
`binaryOccupancyMap` object handle

Binary occupancy grid, returned as a `binaryOccupancyMap` object handle. `map` is converted from a `'nav_msgs/OccupancyGrid'` message on the ROS network. The object is a grid of binary values, where `1` indicates an occupied location and `0` indications an unoccupied location.

See Also

Objects
`OccupancyGrid` | `occupancyMap` | `binaryOccupancyMap`

Functions
`rosReadOccupancyGrid` | `rosWriteBinaryOccupancyGrid` | `rosWriteOccupancyGrid`

Introduced in R2015a

readOccupancyGrid

Read occupancy grid message

Syntax

```
map = readOccupancyGrid(msg)
```

Description

`map = readOccupancyGrid(msg)` returns an `occupancyMap` object by reading the data inside a ROS message, `msg`, which must be a `'nav_msgs/OccupancyGrid'` message. All message data values are converted to probabilities from 0 to 1. The unknown values (-1) in the message are set as 0.5 in the map.

Note The `msg` input is an `'nav_msgs/OccupancyGrid'` ROS message. For more info, see `OccupancyGrid`.

Input Arguments

`msg` — `'nav_msgs/OccupancyGrid'` ROS message

`OccupancyGrid` object handle

`'nav_msgs/OccupancyGrid'` ROS message, specified as an `OccupancyGrid` ROS message object handle.

Output Arguments

`map` — Occupancy map

`occupancyMap` object handle

Occupancy map, returned as an `occupancyMap` object handle.

See Also

Functions

`rosReadBinaryOccupancyGrid` | `rosReadOccupancyMap3D` | `rosWriteBinaryOccupancyGrid` | `rosWriteOccupancyGrid`

Introduced in R2016b

readOccupancyMap3D

Read 3-D map from Octomap ROS message

Syntax

```
map = readOccupancyMap3D(msg)
```

Description

`map = readOccupancyMap3D(msg)` reads the data inside a ROS `'octomap_msgs/Octomap'` message to return an `occupancyMap3D` object. All message data values are converted to probabilities from 0 to 1.

Input Arguments

`msg` — `Octomap` ROS message

structure

`Octomap` ROS message, specified as a structure of message type `'octomap_msgs/Octomap'`. Get this message by subscribing to an `'octomap_msgs/Octomap'` topic using `rossubscriber` on a live ROS network or by creating your own message using `rosmessage`.

Output Arguments

`map` — 3-D occupancy map

`occupancyMap3D` object handle

3-D occupancy map, returned as an `occupancyMap3D` object handle.

See Also

`occupancyMap3D` | `rosmessage` | `rossubscriber`

Introduced in R2021a

receiverposition

Estimate GNSS receiver position and velocity

Syntax

```
recPos = receiverposition(p,satPos)
[recPos,recVel] = receiverposition( ___ ,pdot,satVel)
[recPos,recVel,hdop,vdop] = receiverposition( ___ )
```

Description

`recPos = receiverposition(p,satPos)` returns the receiver position estimated from the pseudoranges and satellite positions.

`[recPos,recVel] = receiverposition( ___ ,pdot,satVel)` also returns the receiver velocity estimated from the pseudorange rates `pdot` and satellite velocities `satVel`.

`[recPos,recVel,hdop,vdop] = receiverposition( ___ )` also returns the horizontal dilation of precision `hdop` and vertical dilation of precision `vdop` associated with the position estimate.

Examples

Estimate Receiver Position from Pseudoranges and Satellite Positions

Use the `receiverposition` function to estimate a GNSS receiver position. Get the satellte positiions and velocities using the `gnssconstellation` function. Generate pseudoranges from these positions using the `pseudoranges` function.

Specify a receiver position in geodetic coordinates (latitude, longitude, altitude) and a receiver velocity in the local navigation frame.

```
recPos = [42 -71 50];
recVel = [1 2 3];
```

Get the satellite positions for the current time.

```
t = datetime('now');
[gpsSatPos,gpsSatVel] = gnssconstellation(t);
```

Get the the pseudoranges and pseudorange rates between the GNSS receiver and the satellites.

```
[p,pdot] = pseudoranges(recPos,gpsSatPos,recVel,gpsSatVel);
```

Use the pseudoranges to estimate the receiver position and velocity. The values close to your original receiver position and velocity used to generate the satellite position and pseudoranges.

```
[lla,gnssVel] = receiverposition(p,gpsSatPos,pdot,gpsSatVel)
```

```
lla = 1×3
```

```
   42.0000  -71.0000   50.0645

gnssVel = 1×3

    0.9893    1.9996    2.9882
```

Input Arguments

p — Pseudoranges between satellites and receiver

n-element vector

Pseudoranges between the satellites and receiver, specified as an *n*-element vector in meters.

Data Types: `single` | `double`

satPos — Satellite positions

N-by-3 matrix of scalars

Satellite positions in the Earth-centered Earth-fixed (ECEF) coordinate system in meters, specified as an *N*-by-3 matrix of scalars. *N* is the number of satellites in the constellation.

Data Types: `single` | `double`

pdot — Pseudorange rates between satellites and receiver

n-element vector

Pseudorange rates between the satellites and receiver, specified as an *n*-element vector in meters per second.

Data Types: `single` | `double`

satVel — Velocity readings in local navigation coordinate system (m/s)

N-by-3 matrix of scalar

Velocity readings of the GNSS receiver in the local navigation coordinate system in meters per second, specified as an *N*-by-3 matrix of scalars. *N* is the number of satellites in the constellation.

Data Types: `single` | `double`

Output Arguments

recPos — Receiver position

three-element vector of the form `[lat lon alt]`

Receiver position in geodetic coordinates, returned as a three-element vector of the form *[latitude longitude altitude]*

Data Types: `single` | `double`

recVel — Receiver velocity

three-element vector of the form `[vx vy vz]`

Receiver velocity in the local navigation frame using north-east-down (NED) coordinates, returned as a three-element vector of the form *[vx vy vz]*.

Data Types: `single` | `double`

hdop — Horizontal dilation of precision
scalar

Horizontal dilation of precision, returned as a scalar.

Data Types: `double`

vdop — Vertical dilation of precision
scalar

Vertical dilation of precision, returned as a scalar.

Data Types: `double`

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

Objects
`gnssSensor` | `gpsSensor` | `imuSensor`

Functions
`skyplot` | `gnssconstellation` | `lookangles` | `pseudoranges`

Introduced in R2021a

release

Release the GPS object

Syntax

```
release(gps)
```

Description

`release(gps)` release the system objects, allows configuration of non tunable properties, clear the buffers, and resets the values of `SamplesRead` and `SamplesAvailable` properties.

Examples

Read Data from GPS Receiver as Timetable

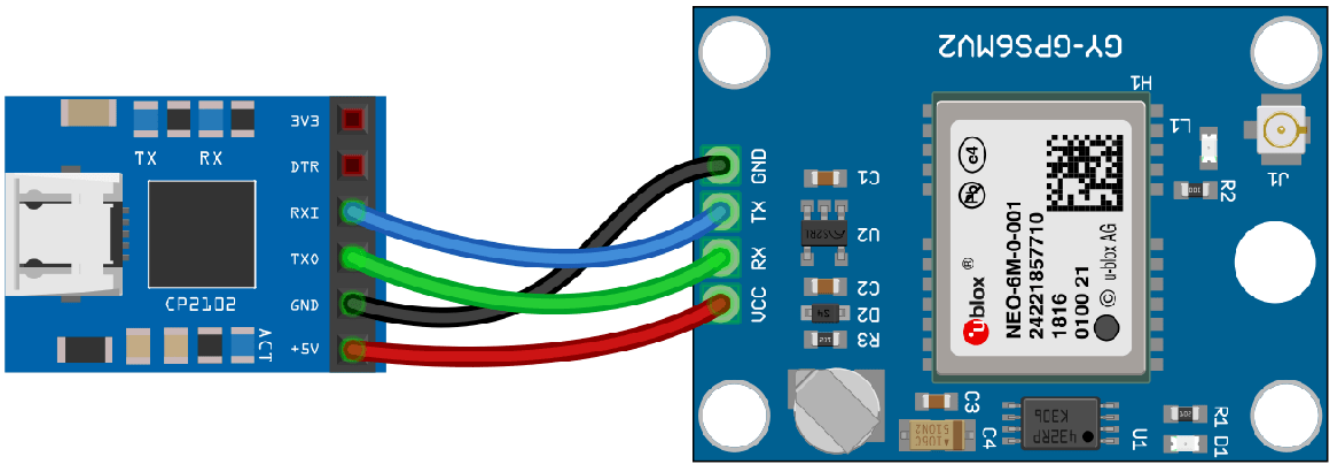
Read data from the GPS receiver connected to the host computer on a specific serial port.

Required Hardware

To run this example, you need:

- UBlox Neo-6M GPS module
- GPS antenna
- USB to UART module
- USB cable
- Connecting wires

Hardware Connection



Connect the pins on the UBlox Neo-6M GPS module to the pins on your USB to UART module. The connections are:

- VCC - +5V
- RX - TXO
- TX - RXI
- GND - GND

Connect the GPS antenna to the GPS module. Connect the USB to UART module to the host computer with a USB cable. GPS Fix can be easily acquired in locations that have a clear view of the sky. Wait for the GPS module to acquire satelite signals (Fix).This can be verified by checking the Fix LED (D1) of your GPS module.

Create GPS Object

Create a `gpsdev` object for the GPS receiver connected to a specific port. Specify the output format of the data as a timetable.

```
gps = gpsdev('COM4','OutputFormat',"timetable")
```

```
gps = 
  gpsdev with properties:

                         SerialPort: COM4
                           BaudRate: 9600 (bits/s)

                     SamplesPerRead: 1
                           ReadMode: "latest"
                        SamplesRead: 0
Show all properties all functions
```

Read the GPS data

Read the GPS data and return them as a timetable.

```
[tt,overruns] = read(gps)
```

```
tt=1×5 timetable
              Time                           LLA                 GroundSpeed    Course          DOI
    _______________________    ________________________    ___________    ______    __________

    22-Mar-2021 15:31:15.190    17.47    78.343    449.6      0.25619       NaN      9.31    1.4

overruns = 0
```

Display number of samples read and the samples available in the host buffer.

```
gps.SamplesRead
```

```
ans = 1
```

```
gps.SamplesAvailable
```

```
ans = 0
```

Release the GPS object to configure the non tunable properties. The release function also clears the buffer and resets the `SamplesRead` and `SamplesAvailable` properties.

```
release(gps)
```

Specify the number of samples per read to 2. Read the GPS data.

```
gps.SamplesPerRead = 2;
read(gps)
```

```
ans=2×5 timetable
            Time                           LLA                 GroundSpeed    Course          DOP
    ________________________    _________________________    ___________    ______    __________

    22-Mar-2021 15:31:17.178    17.47     78.343        450      0.063791        NaN     9.32    1.
    22-Mar-2021 15:31:17.178    17.47     78.343        450      0.063791        NaN     9.32    1.
```

Display number of samples read and the samples available in the host buffer.

```
gps.SamplesRead
```

```
ans = 1
```

```
gps.SamplesAvailable
```

```
ans = 0
```

Clean Up

When the connection is no longer needed, clear the associated object.

```
delete(gps);
clear gps;
```

Input Arguments

gps — GPS sensor
`gpsdev` object

The GPS sensor, specified as a `gpsdev` object.

See Also

Objects
`gpsdev`

Functions
`flush` | `writeBytes` | `read` | `info`

Introduced in R2020b

removeInvalidData

Remove invalid range and angle data

Syntax

```
validScan = removeInvalidData(scan)
validScan = removeInvalidData(scan,Name,Value)
```

Description

`validScan = removeInvalidData(scan)` returns a new `lidarScan` object with all `Inf` and `NaN` values from the input `scan` removed. The corresponding angle readings are also removed.

`validScan = removeInvalidData(scan,Name,Value)` provides additional options specified by one or more `Name,Value` pairs.

Examples

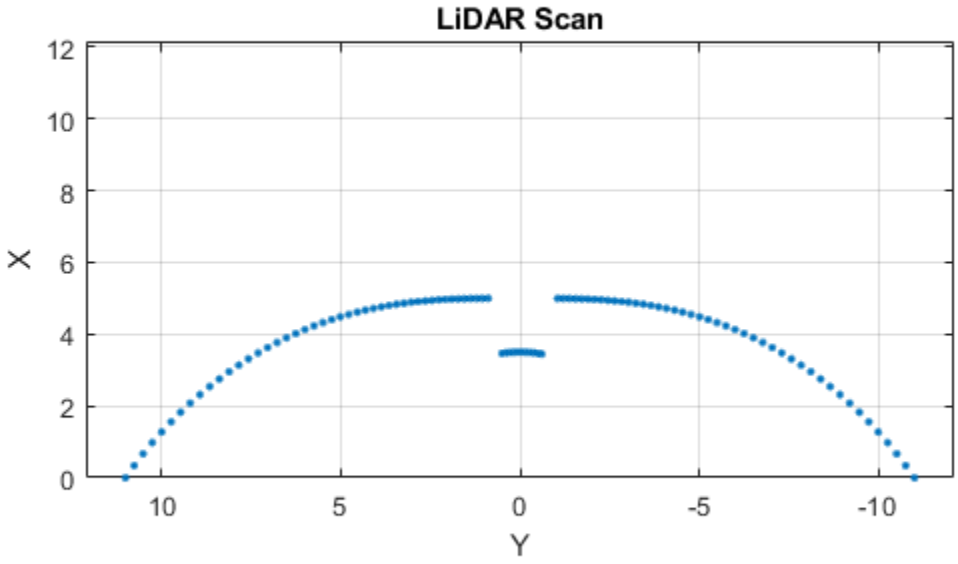
Plot Lidar Scan and Remove Invalid Points

Specify lidar data as vectors of ranges and angles. These values include readings outside of the sensors range.

```
x = linspace(-2,2);
ranges = abs((1.5).*x.^2 + 5);
ranges(45:55) = 3.5;
angles = linspace(-pi/2,pi/2,numel(ranges));
```

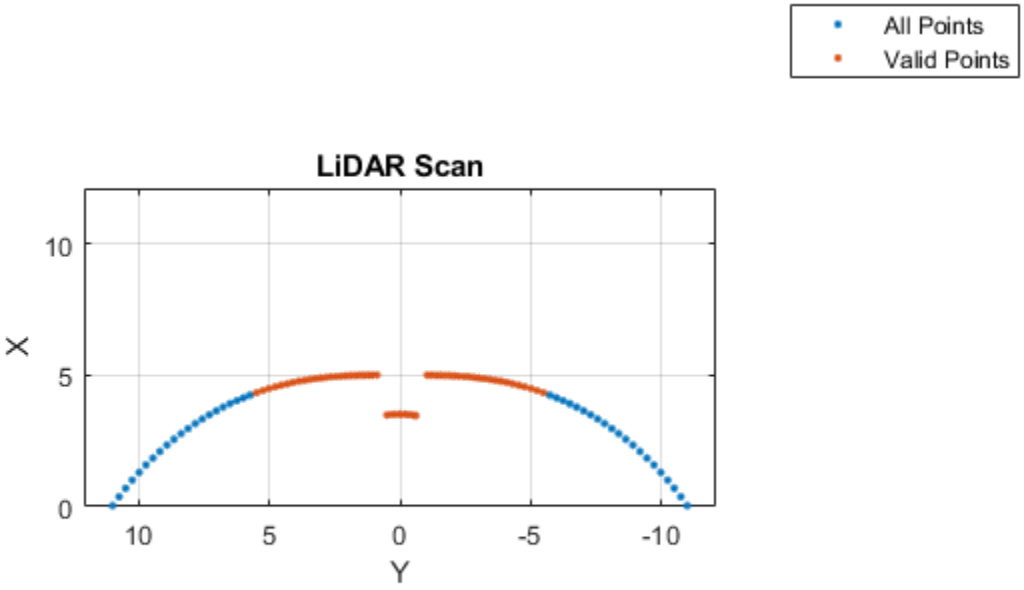
Create a lidar scan by specifying the ranges and angles. Plot all points of the lidar scan.

```
scan = lidarScan(ranges,angles);
plot(scan)
```



Remove invalid points based on a specified minimum and maximum range.

```
minRange = 0.1;
maxRange = 7;
scan2 = removeInvalidData(scan,'RangeLimits',[minRange maxRange]);
hold on
plot(scan2)
legend('All Points','Valid Points')
```



Input Arguments

`scan` — Lidar scan readings

`lidarScan` object

Lidar scan readings, specified as a `lidarScan` object.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `["RangeLimits",[0.05 2]`

`RangeLimits` — Range reading limits

two-element vector

Range reading limits, specified as a two-element vector, `[minRange maxRange]`, in meters. All range readings and corresponding angles outside these range limits are removed

Data Types: `single` | `double`

`AngleLimits` — Angle limits

two-element vector

Angle limits, specified as a two-element vector, `[minAngle maxAngle]` in radians. All angles and corresponding range readings outside these angle limits are removed.

Angles are measured counter-clockwise around the positive *z*-axis.

Data Types: `single` | `double`

Output Arguments

`validScan` — Lidar scan readings

`lidarScan` object

Lidar scan readings, specified as a `lidarScan` object. All invalid lidar scan readings are removed.

See Also

`lidarScan` | `transformScan` | `matchScans`

Introduced in R2019b

rotm2axang

Convert rotation matrix to axis-angle rotation

Syntax

```
axang = rotm2axang(rotm)
```

Description

`axang = rotm2axang(rotm)` converts a rotation given as an orthonormal rotation matrix, `rotm`, to the corresponding axis-angle representation, `axang`. The input rotation matrix must be in the premultiply form for rotations.

Examples

Convert Rotation Matrix to Axis-Angle Rotation

```
rotm = [1 0 0 ; 0 -1 0; 0 0 -1];
axang = rotm2axang(rotm)
```

```
axang = 1×4

    1.0000         0         0    3.1416
```

Input Arguments

`rotm` — Rotation matrix

3-by-3-by-*n* matrix

Rotation matrix, specified as a 3-by-3-by-*n* matrix containing *n* rotation matrices. Each rotation matrix has a size of 3-by-3 and must be orthonormal. The input rotation matrix must be in the premultiply form for rotations.

Note Rotation matrices that are slightly non-orthonormal can give complex outputs. Consider validating your matrix before inputting to the function.

Example: `[0 0 1; 0 1 0; -1 0 0]`

Output Arguments

axang — Rotation given in axis-angle form

n-by-4 matrix

Rotation given in axis-angle form, returned as an *n*-by-4 matrix of *n* axis-angle rotations. The first three elements of every row specify the rotation axis, and the last element defines the rotation angle (in radians).

Example: `[1 0 0 pi/2]`

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`axang2rotm`

Introduced in R2015a

rotm2eul

Convert rotation matrix to Euler angles

Syntax

```
eul = rotm2eul(rotm)
eul = rotm2eul(rotm,sequence)
```

Description

`eul = rotm2eul(rotm)` converts a rotation matrix, `rotm`, to the corresponding Euler angles, `eul`. The input rotation matrix must be in the premultiply form for rotations. The default order for Euler angle rotations is `"ZYX"`.

`eul = rotm2eul(rotm,sequence)` converts a rotation matrix to Euler angles. The Euler angles are specified in the axis rotation sequence, `sequence`. The default order for Euler angle rotations is `"ZYX"`.

Examples

Convert Rotation Matrix to Euler Angles

```
rotm = [0 0 1; 0 1 0; -1 0 0];
eulZYX = rotm2eul(rotm)
```

```
eulZYX = 1×3

         0    1.5708         0
```

Convert Rotation Matrix to Euler Angles Using ZYZ Axis Order

```
rotm = [0 0 1; 0 -1 0; -1 0 0];
eulZYZ = rotm2eul(rotm,'ZYZ')
```

```
eulZYZ = 1×3

   -3.1416   -1.5708   -3.1416
```

Input Arguments

`rotm` — Rotation matrix

3-by-3-by-*n* matrix

Rotation matrix, specified as a 3-by-3-by-*n* matrix containing *n* rotation matrices. Each rotation matrix has a size of 3-by-3 and is orthonormal. The input rotation matrix must be in the premultiply form for rotations.

Note Rotation matrices that are slightly non-orthonormal can give complex outputs. Consider validating your matrix before inputting to the function.

Example: `[0 0 1; 0 1 0; -1 0 0]`

sequence — Axis rotation sequence

`"ZYX"` (default) | `"ZYZ"` | `"XYZ"`

Axis rotation sequence for the Euler angles, specified as one of these string scalars:

- `"ZYX"` (default) – The order of rotation angles is *z*-axis, *y*-axis, *x*-axis.
- `"ZYZ"` – The order of rotation angles is *z*-axis, *y*-axis, *z*-axis.
- `"XYZ"` – The order of rotation angles is *x*-axis, *y*-axis, *z*-axis.

Data Types: `string` | `char`

Output Arguments

eul — Euler rotation angles

n-by-3 matrix

Euler rotation angles in radians, returned as an *n*-by-3 array of Euler rotation angles. Each row represents one Euler angle set.

Example: `[0 0 1.5708]`

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`eul2rotm`

Introduced in R2015a

rotm2quat

Convert rotation matrix to quaternion

Syntax

```
quat = rotm2quat(rotm)
```

Description

`quat = rotm2quat(rotm)` converts a rotation matrix, `rotm`, to the corresponding unit quaternion representation, `quat`. The input rotation matrix must be in the premultiply form for rotations.

Examples

Convert Rotation Matrix to Quaternion

```
rotm = [0 0 1; 0 1 0; -1 0 0];
quat = rotm2quat(rotm)
```

```
quat = 1×4

    0.7071         0    0.7071         0
```

Input Arguments

`rotm` — Rotation matrix

3-by-3-by-*n* matrix

Rotation matrix, specified as a 3-by-3-by-*n* matrix containing *n* rotation matrices. Each rotation matrix has a size of 3-by-3 and is orthonormal. The input rotation matrix must be in the premultiply form for rotations.

Note Rotation matrices that are slightly non-orthonormal can give complex outputs. Consider validating your matrix before inputting to the function.

Example: `[0 0 1; 0 1 0; -1 0 0]`

Output Arguments

`quat` — Unit quaternion

n-by-4 matrix

Unit quaternion, returned as an *n*-by-4 matrix containing *n* quaternions. Each quaternion, one per row, is of the form $q = [w\ x\ y\ z]$, with w as the scalar number.

Example: `[0.7071 0.7071 0 0]`

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`quat2rotm`

Introduced in R2015a

rotm2tform

Convert rotation matrix to homogeneous transformation

Syntax

```
tform = rotm2tform(rotm)
```

Description

`tform = rotm2tform(rotm)` converts the rotation matrix, `rotm`, into a homogeneous transformation matrix, `tform`. The input rotation matrix must be in the premultiply form for rotations. When using the transformation matrix, premultiply it with the coordinates to be transformed (as opposed to postmultiplying).

Examples

Convert Rotation Matrix to Homogeneous Transformation

```
rotm = [1 0 0 ; 0 -1 0; 0 0 -1];
tform = rotm2tform(rotm)
```

```
tform = 4×4

     1     0     0     0
     0    -1     0     0
     0     0    -1     0
     0     0     0     1
```

Input Arguments

`rotm` — Rotation matrix

3-by-3-by-*n* matrix

Rotation matrix, specified as a 3-by-3-by-*n* matrix containing *n* rotation matrices. Each rotation matrix has a size of 3-by-3 and is orthonormal. The input rotation matrix must be in the premultiply form for rotations.

Note Rotation matrices that are slightly non-orthonormal can give complex outputs. Consider validating your matrix before inputting to the function.

Example: `[0 0 1; 0 1 0; -1 0 0]`

Output Arguments

`tform` — Homogeneous transformation

4-by-4-by-*n* matrix

Homogeneous transformation matrix, specified by a 4-by-4-by-*n* matrix of *n* homogeneous transformations. When using the rotation matrix, premultiply it with the coordinates to be rotated (as opposed to postmultiplying).

Example: `[0 0 1 0; 0 1 0 0; -1 0 0 0; 0 0 0 1]`

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`tform2rotm`

Introduced in R2015a

show

Visualize path segment

Syntax

```
show(pathSeg)
show(pathSeg,Name,Value)
```

Description

`show(pathSeg)` plots the path segment with start and goal positions and their headings.

`show(pathSeg,Name,Value)` also specifies `Name,Value` pairs to control display settings.

Examples

Connect Poses Using Dubins Connection Path

Create a `dubinsConnection` object.

```
dubConnObj = dubinsConnection;
```

Define start and goal poses as `[x y theta]` vectors.

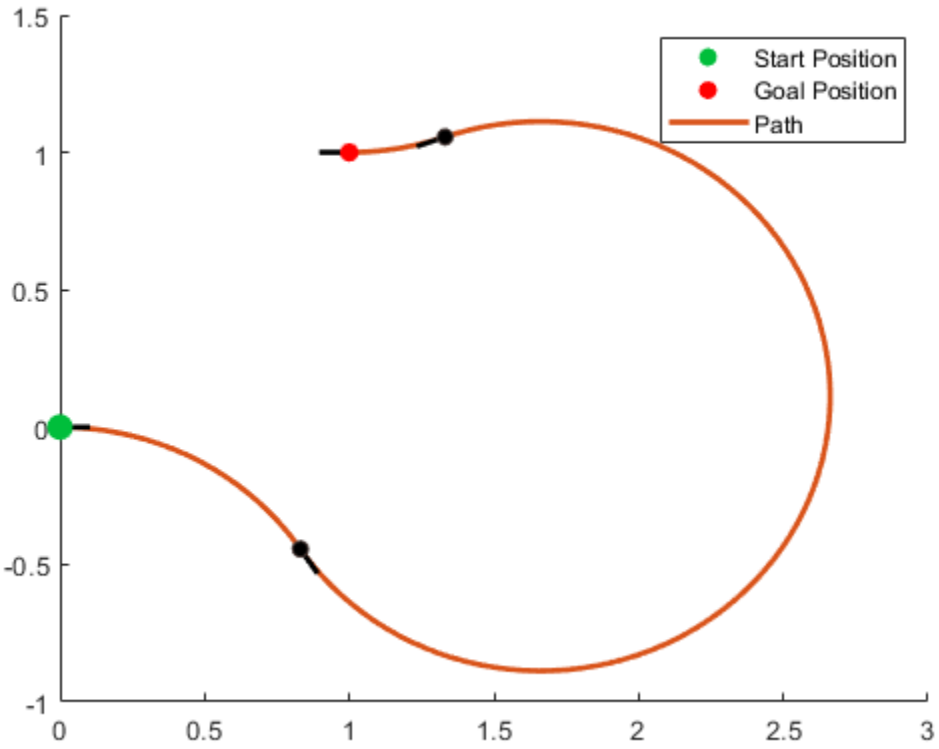
```
startPose = [0 0 0];
goalPose = [1 1 pi];
```

Calculate a valid path segment to connect the poses.

```
[pathSegObj, pathCosts] = connect(dubConnObj,startPose,goalPose);
```

Show the generated path.

```
show(pathSegObj{1})
```



Modify Connection Types for Reeds-Shepp Path

Create a `reedsSheppConnection` object.

```
reedsConnObj = reedsSheppConnection;
```

Define start and goal poses as `[x y theta]` vectors.

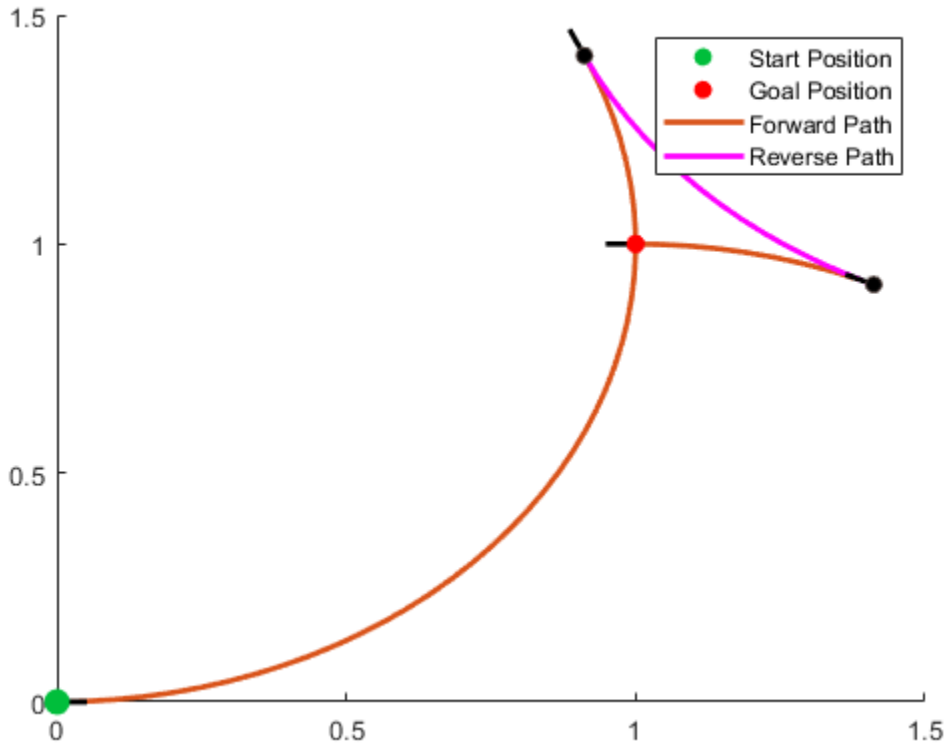
```
startPose = [0 0 0];
goalPose = [1 1 pi];
```

Calculate a valid path segment to connect the poses.

```
[pathSegObj,pathCosts] = connect(reedsConnObj,startPose,goalPose);
```

Show the generated path. Notice the direction of the turns.

```
show(pathSegObj{1})
```




```
pathSegObj{1}.MotionTypes
```

```
ans = 1x5 cell
    {'L'}    {'R'}    {'L'}    {'N'}    {'N'}
```

```
pathSegObj{1}.MotionDirections
```

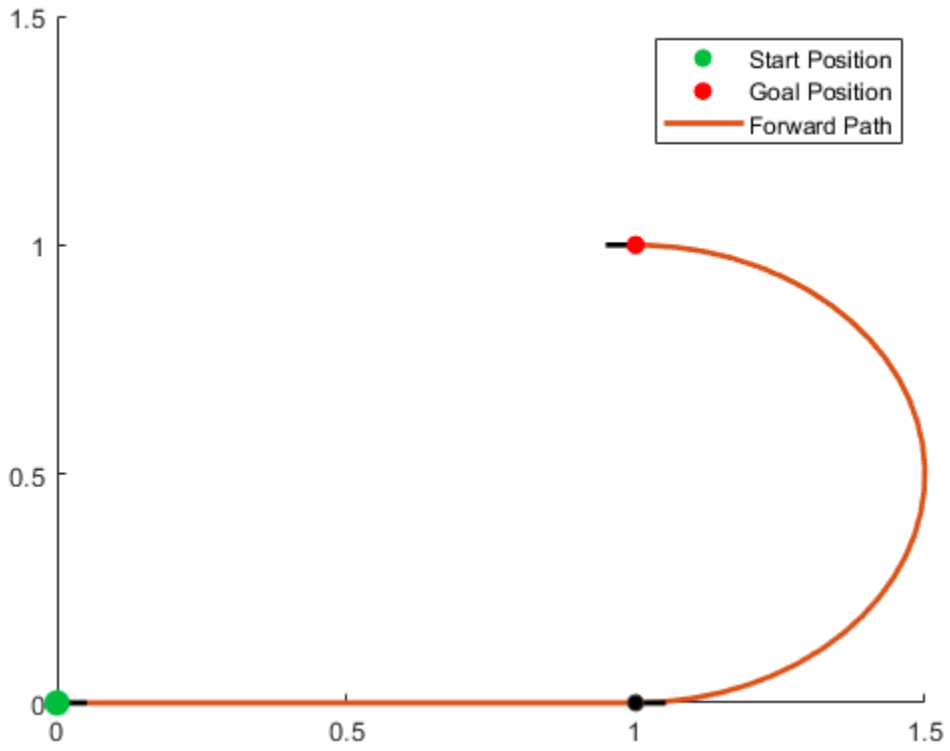
```
ans = 1×5

     1    -1     1     1     1
```

Disable this specific motion sequence in a new connection object. Reduce the `MinTurningRadius` if the robot is more maneuverable. Increase the reverse cost to reduce the likelihood of reverse directions being used. Connect the poses again to get a different path.

```
reedsConnObj = reedsSheppConnection('DisabledPathTypes',{'LpRnLp'});
reedsConnObj.MinTurningRadius = 0.5;
reedsConnObj.ReverseCost = 5;

[pathSegObj,pathCosts] = connect(reedsConnObj,startPose,goalPose);
pathSegObj{1}.MotionTypes
```

```
ans = 1x5 cell
    {'L'}    {'S'}    {'L'}    {'N'}    {'N'}
```

```
show(pathSegObj{1})
xlim([0 1.5])
ylim([0 1.5])
```



Interpolate Poses For Dubins Path

Create a `dubinsConnection` object.

```
dubConnObj = dubinsConnection;
```

Define start and goal poses as `[x y theta]` vectors.

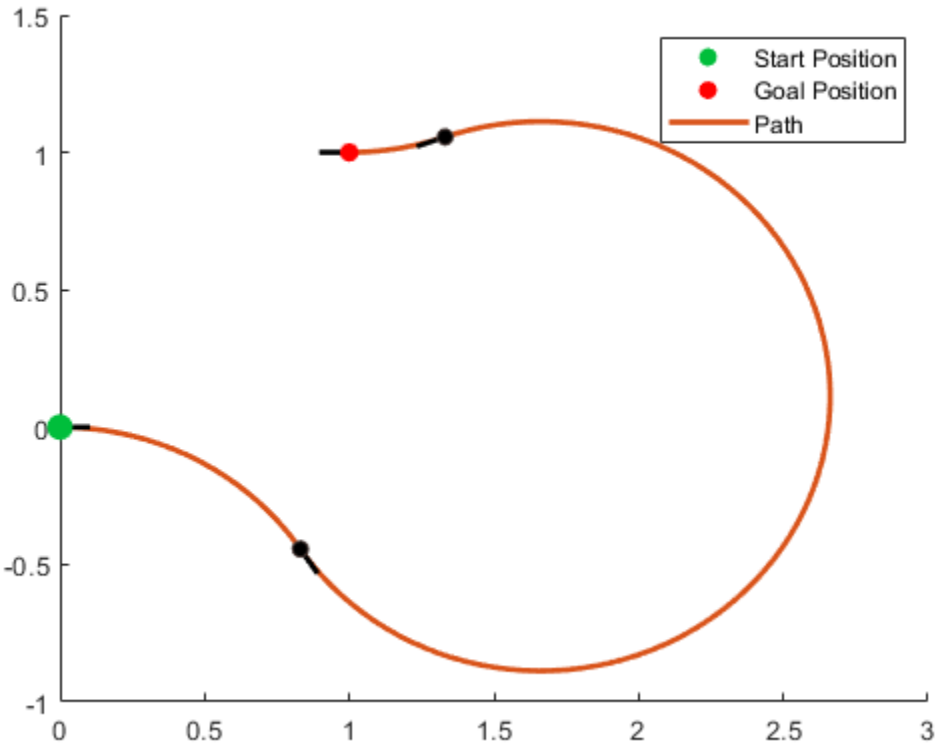
```
startPose = [0 0 0];
goalPose = [1 1 pi];
```

Calculate a valid path segment to connect the poses.

```
[pathSegObj,pathCosts] = connect(dubConnObj,startPose,goalPose);
```

Show the generated path.

```
show(pathSegObj{1})
```



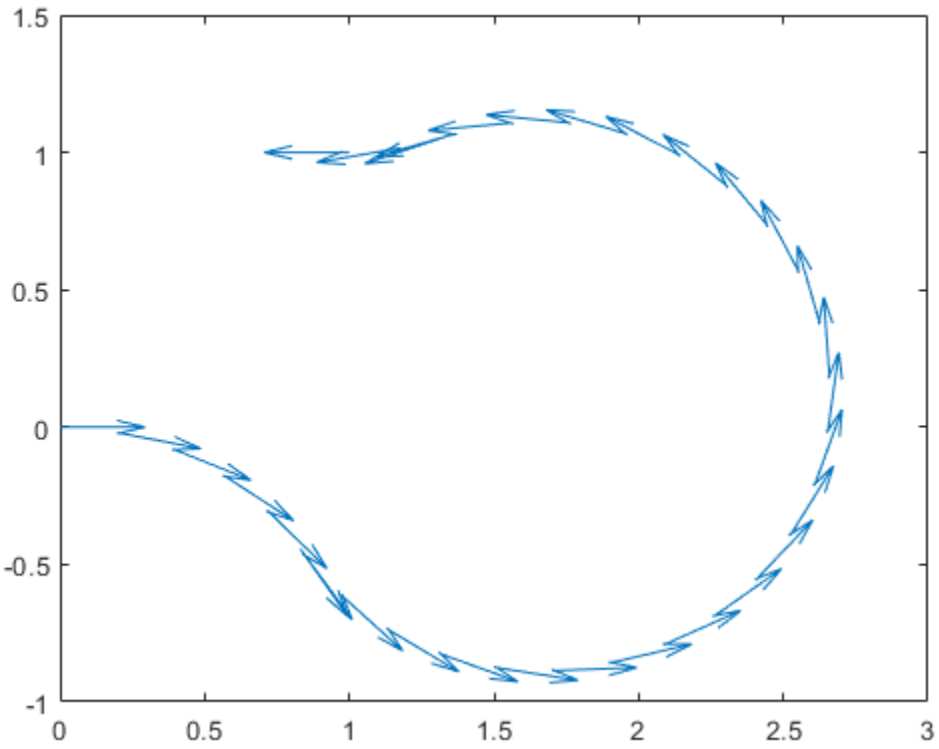
Interpolate poses along the path. Get a pose every 0.2 meters, including the transitions between turns.

```
length = pathSegObj{1}.Length;
poses = interpolate(pathSegObj{1},0:0.2:length)
```

```
poses = 32×3

         0         0         0
    0.1987   -0.0199    6.0832
    0.3894   -0.0789    5.8832
    0.5646   -0.1747    5.6832
    0.7174   -0.3033    5.4832
    0.8309   -0.4436    5.3024
    0.8418   -0.4595    5.3216
    0.9718   -0.6110    5.5216
    1.1293   -0.7337    5.7216
    1.3081   -0.8226    5.9216
       ⋮
```

Use the `quiver` function to plot these poses.

```
quiver(poses(:,1),poses(:,2),cos(poses(:,3)),sin(poses(:,3)),0.5)
```



Input Arguments

pathSeg — Path segment

`dubinsPathSegment` object | `reedsSheppPathSegment` object

Path segment, specified as a `dubinsPathSegment` or `reedsSheppPathSegment` object.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'Headings',{'transitions'}`

Parent — Axes to plot path onto

`Axes` handle

Axes to plot path onto, specified as an `Axes` handle.

Headings — Heading angles to display

cell array of character vector or string scalars

Heading angles to display, specified as a cell array of character vector or string scalars. Options are any combination of `'start'`,`'goal'`, and `'transitions'`. To disable all heading displays, specify `{''}`.

`Positions` — Positions to display
`'both'` (default) | `'start'` | `'goal'` | `'none'`

Positions to display, specified as `'both'`, `'start'`, `'goal'`, or `'none'`. The start position is marked with green, and the goal position is marked with red.

`HeadingLength` — Length of heading
positive numeric scalar

Length of heading, specified as positive numeric scalar. By default the value is calculated according to the `x`- and `y`-axis limits of the plot.

Data Types: `double`

See Also

Functions
`interpolate` | `connect`

Objects
`dubinsConnection` | `dubinsPathSegment` | `reedsSheppConnection` | `reedsSheppPathSegment`

Introduced in R2019b

skyplot

Plot satellite azimuth and elevation data

Syntax

```
skyplot(azdata,eldata)
skyplot(azdata,eldata,labeldata)
skyplot(status)
skyplot(___,Name,Value)

skyplot(parent,___)
h = skyplot(___)
```

Description

`skyplot(azdata,eldata)` creates a sky plot using the azimuth and elevation data specified as vectors in degrees. Azimuth angles are measured in degrees, clockwise-positive from the North direction. Elevation angles are measured from the horizon line with 90 degrees being directly up. For details about the sky plot figure elements, see “Main Sky Plot Elements” on page 1-203.

`skyplot(azdata,eldata,labeldata)` specifies data labels as a string array with elements corresponding to each data point in the `azdata` and `eldata` inputs.

`skyplot(status)` specifies the azimuth and elevation data in a structure with fields `SatelliteAzimuth` and `SatelliteElevation`.

`skyplot(___,Name,Value)` specifies options using one or more name-value arguments in addition to the input arguments in previous syntaxes. The name-value arguments are properties of the `SkyPlotChart` object. For a list of properties, see SkyPlotChart Properties.

`skyplot(parent,___)` creates the sky plot in the figure, panel, or tab specified by `parent`.

`h = skyplot(___)` returns the sky plot as a `SkyPlotChart` object, `h`. Use `h` to modify the properties of the chart after creating it. For a list of properties, see SkyPlotChart Properties.

Examples

View Satellite Positions from GNSS Sensor

Create a GNSS sensor model as a `gnssSensor` System Object™.

```
gnss = gnssSensor;
```

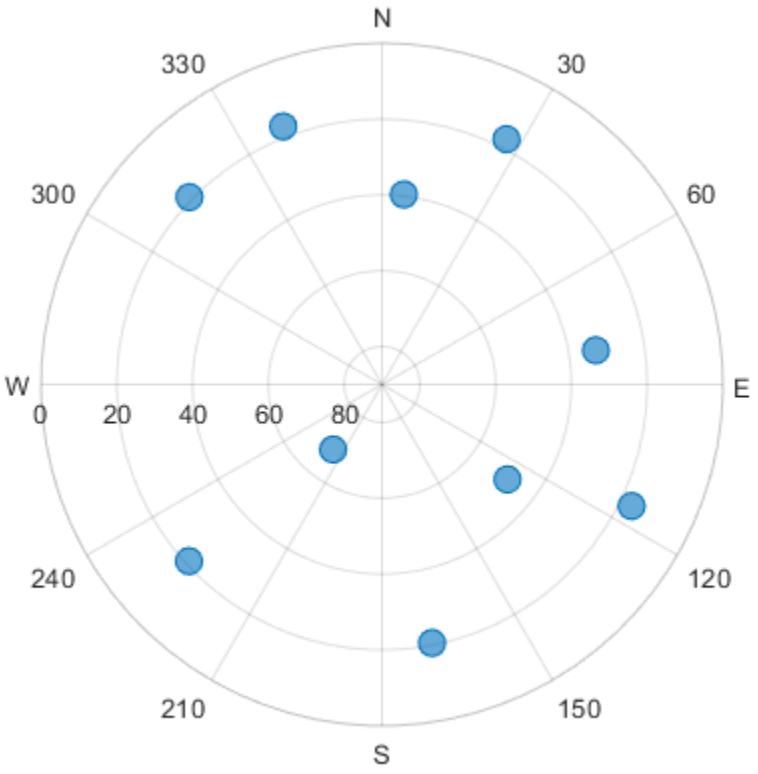
Specify the position and velocity of the sensor. Simulate the sensor readings and get status from visible satellites. Store the azimuth and elevation angles as vectors.

```
pos = [0 0 0];
vel = [0 0 0];
[~, ~, status] = gnss(pos, vel);
```

```
satAz = status.SatelliteAzimuth;
satEl = status.SatelliteElevation;
```

Plot the satellite postions.

```
skyplot(satAz,satEl)
```

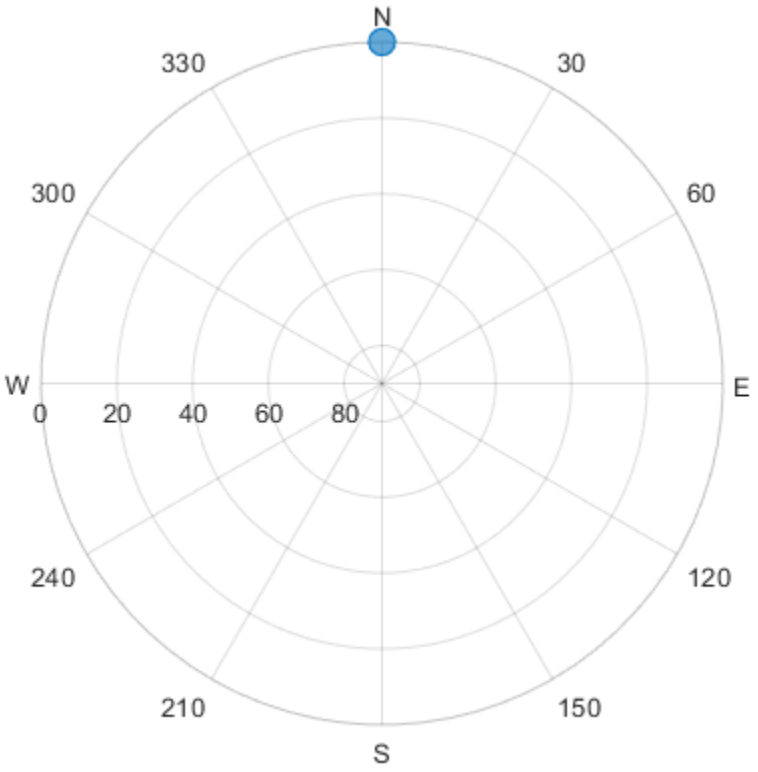


Plot Series of Satellite Positions Over Time

Animate the trajectory of satellite positions over time from a GNSS sensor.

Initialize the sky plot figure. Specify the relevant time-stepping information.

```
skyplotHandle = skyplot(0,0);
```




```
numHours = 12;
dt = 100;
numSeconds = numHours * 60 * 60;
numSimSteps = numSeconds/dt;
```

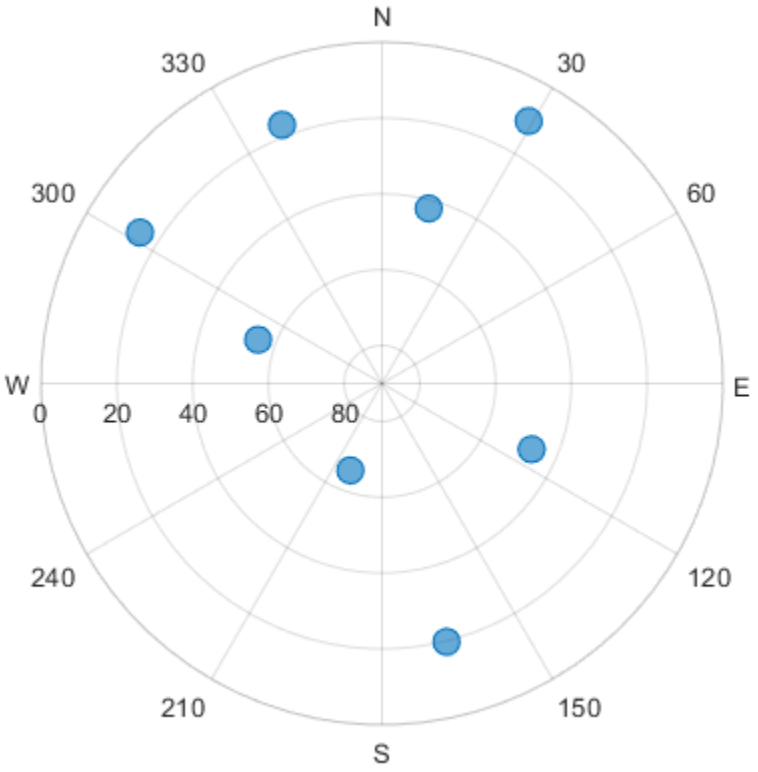
Create a GNSS sensor model as a `gnssSensor` System Object™.

```
gnss = gnssSensor('SampleRate', 1/dt);
```

Iterate through the time steps and do the following:

- Simulate the sensor readings. Specify the zero postion and velocity for the stationary sensor.
- Store the azimuth and elevation angles as vectors.
- Set the `AzimuthData` and `ElevationData` properties of the `SkyPlotChart` handle directly.

```
for i = 1:numSimSteps

    [~, ~, status] = gnss([0 0 0],[0 0 0]);

    satAz = status.SatelliteAzimuth;
    satEl = status.SatelliteElevation;

    set(skyplotHandle,'AzimuthData',satAz,'ElevationData',satEl);

    drawnow
end
```



View Satellite Positions For Different Groups

Load the azimuth and elevation data from a logfile generated by an Adafruit® GPS satellite sensor. The data provided in this example contains the azimuth and elevation of each satellite and the pseudorandom noise (PRN) codes. Store these values as vectors.

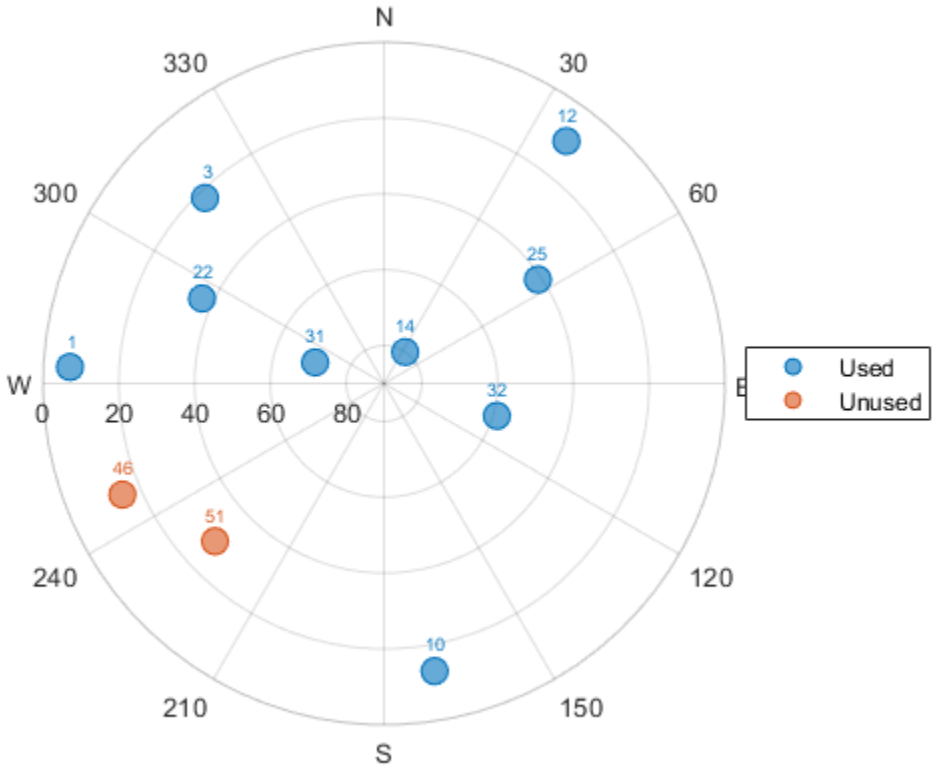
```
load('gpsHWInfo','hwInfo')
satAz = hwInfo.SatelliteAzimuths;
satEl = hwInfo.SatelliteElevations;
prn = hwInfo.SatellitePRNs;
```

Separate the satellites based on the PRN codes. To correlate each position with a group, create a `categorical` array. For this set of satellites, only the ones with PRNs less than 32 are used in the positioning solution.

```
isUnused = (prn > 32);
group = categorical(isUnused,[false true],["Used in Positioning Solution" "Unused"]);
```

Visualize the satellites and specify the categorical groups in the `GroupData` name-value argument. Specify the PRN as the label for each point. Show the legend.

```
skyplot(satAz,satEl,prn,GroupData=group)
legend('Used','Unused')
```



Input Arguments

`azdata` — Azimuth angles for visible satellite positions

n-element vector of angles

Azimuth angles for visible satellite positions, specified as an *n*-element vector of angles. *n* is the number of visible satellite positions in the plot. Azimuth angles are measured in degrees, clockwise-positive from the North direction.

Example: `[25 45 182 356]`

Data Types: `double`

`eldata` — Elevation angles for visible satellite positions

n-element vector of angles

Elevation angles for visible satellite positions, specified as an *n*-element vector of angles. *n* is the number of visible satellite positions in the plot. Elevation angles are measured from the horizon line with 90 degrees being directly up.

Example: `[45 90 27 74]`

Data Types: `double`

`labeldata` — Labels for visible satellite positions

n-element string array

Labels for visible satellite positions, specified as an *n*-element string array. *n* is the number of visible satellite positions in the plot.

Example: `["G1" "G11" "G7" "G3"]`

Data Types: `string`

`status` — Satellite status
structure array

Satellite status, specified as a structure array with fields `SatelliteAzimuth` and `SatelliteElevation`. Typically, this status structure comes from a `gnssSensor` object, which simulates satellite positions and velocities.

Example: `gnss = gnssSensor; [~,~,status] = gnss(position,velocity)`

Data Types: `struct`

`parent` — Parent container
`Figure` object | `Panel` object | `Tab` object | `TiledChartLayout` object | `GridLayout` object

Parent container, specified as a `Figure`, `Panel`, `Tab`, `TiledChartLayout`, or `GridLayout` object.

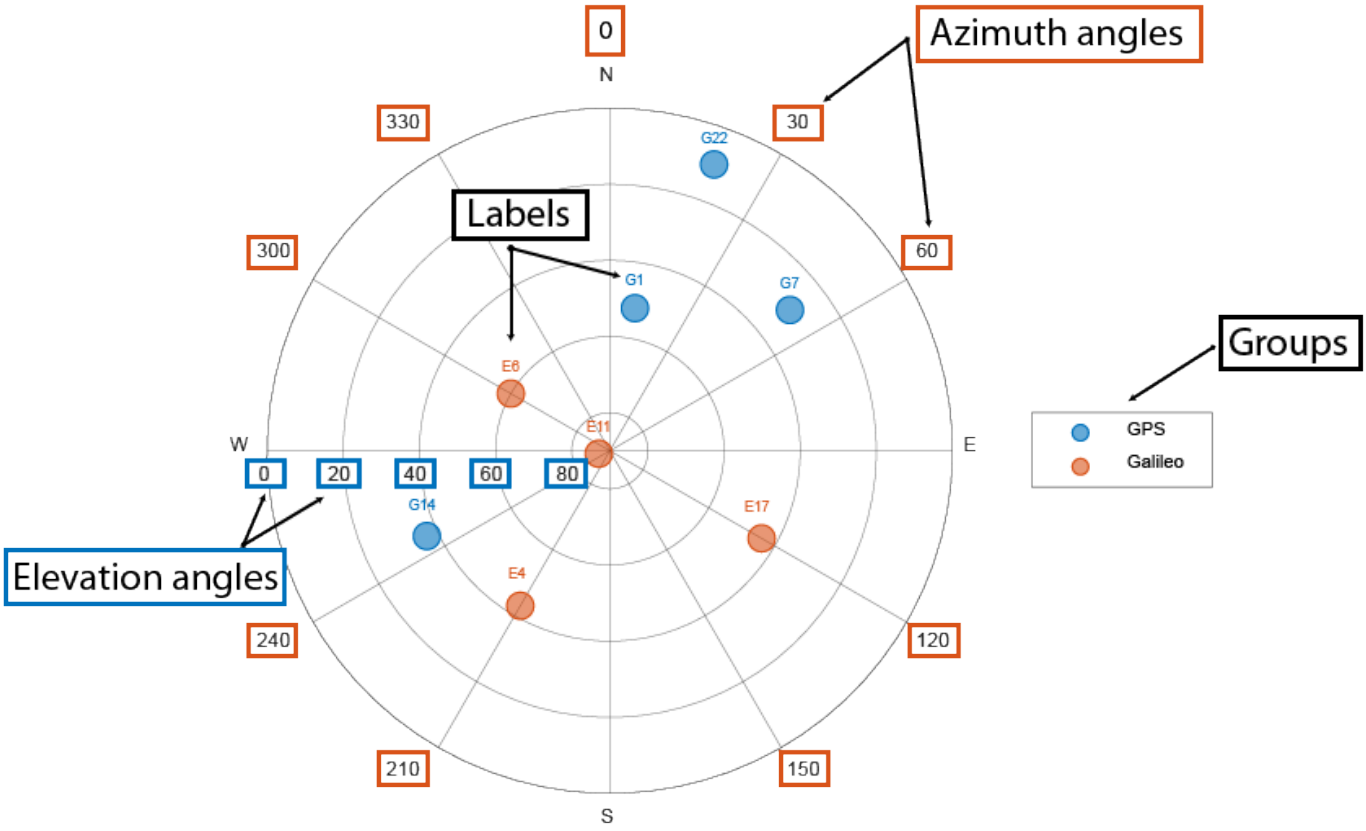
Output Arguments

`h` — Sky plot chart
`SkyplotChart` object

Sky plot chart, returned as a `SkyplotChart` object, which is a standalone visualization on page 1-203. Use `h` to set properties on the sky plot chart. For more information, see SkyPlotChart Properties.

More About

Main Sky Plot Elements



The main elements of the figure are:

- Azimuth axes — Specified by the `azdata` input argument, azimuth angle positions are measured clockwise-positive from the North direction.
- Elevation axes —Specified by the `eldata` input argument, elevation angle positions are measured from the horizon line with 90 degrees being directly up.
- Labels — Specified by the `labeldata` input argument as a string array with an element for each point in the `azdata` and `eldata` vectors.
- Groups — Specified by the GroupData property, a `categorical` array defines the group for each satellite position.

Standalone Visualization

A standalone visualization is a chart designed for a special purpose that works independently from other charts. Unlike other charts such as `plot` and `surf`, a standalone visualization has a preconfigured axes object built into it, and some customizations are not available. A standalone visualization also has these characteristics:

- It cannot be combined with other graphics elements, such as lines, patches, or surfaces. Thus, the `hold` command is not supported.

- The `gca` function can return the chart object as the current axes.
- You can pass the chart object to many MATLAB functions that accept an axes object as an input argument. For example, you can pass the chart object to the `title` function.

See Also

Functions
`polarscatter`

Properties
SkyPlotChart Properties

Objects
`gnssSensor` | `nmeaParser`

Introduced in R2021a

tform2axang

Convert homogeneous transformation to axis-angle rotation

Syntax

```
axang = tform2axang(tform)
```

Description

`axang = tform2axang(tform)` converts the rotational component of a homogeneous transformation, `tform`, to an axis-angle rotation, `axang`. The translational components of `tform` are ignored. The input homogeneous transformation must be in the premultiply form for transformations.

Examples

Convert Homogeneous Transformation to Axis-Angle Rotation

```
tform = [1 0 0 0; 0 0 -1 0; 0 1 0 0; 0 0 0 1];
axang = tform2axang(tform)
```

```
axang = 1×4

    1.0000         0         0    1.5708
```

Input Arguments

`tform` — Homogeneous transformation

4-by-4-by-*n* matrix

Homogeneous transformation, specified by a 4-by-4-by-*n* matrix of *n* homogeneous transformations. The input homogeneous transformation must be in the premultiply form for transformations.

Example: `[0 0 1 0; 0 1 0 0; -1 0 0 0; 0 0 0 1]`

Output Arguments

axang — Rotation given in axis-angle form

n-by-4 matrix

Rotation given in axis-angle form, specified as an *n*-by-4 matrix of *n* axis-angle rotations. The first three elements of every row specify the rotation axes, and the last element defines the rotation angle (in radians).

Example: `[1 0 0 pi/2]`

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`axang2tform`

Introduced in R2015a

tform2eul

Extract Euler angles from homogeneous transformation

Syntax

```
eul = tform2eul(tform)
eul = tform2eul(tform, sequence)
```

Description

`eul = tform2eul(tform)` extracts the rotational component from a homogeneous transformation, `tform`, and returns it as Euler angles, `eul`. The translational components of `tform` are ignored. The input homogeneous transformation must be in the premultiply form for transformations. The default order for Euler angle rotations is `"ZYX"`.

`eul = tform2eul(tform, sequence)` extracts the Euler angles, `eul`, from a homogeneous transformation, `tform`, using the specified rotation sequence, `sequence`. The default order for Euler angle rotations is `"ZYX"`.

Examples

Extract Euler Angles from Homogeneous Transformation Matrix

```
tform = [1 0 0 0.5; 0 -1 0 5; 0 0 -1 -1.2; 0 0 0 1];
eulZYX = tform2eul(tform)
```

```
eulZYX = 1×3

         0         0    3.1416
```

Extract Euler Angles from Homogeneous Transformation Matrix Using ZYZ Rotation

```
tform = [1 0 0 0.5; 0 -1 0 5; 0 0 -1 -1.2; 0 0 0 1];
eulZYZ = tform2eul(tform,'ZYZ')
```

```
eulZYZ = 1×3

         0   -3.1416    3.1416
```

Input Arguments

`tform` — Homogeneous transformation

4-by-4-by-*n* matrix

Homogeneous transformation, specified by a 4-by-4-by-*n* matrix of *n* homogeneous transformations. The input homogeneous transformation must be in the premultiply form for transformations.

Example: `[0 0 1 0; 0 1 0 0; -1 0 0 0; 0 0 0 1]`

`sequence` — Axis rotation sequence
`"ZYX"` (default) | `"ZYZ"` | `"XYZ"`

Axis rotation sequence for the Euler angles, specified as one of these string scalars:

- `"ZYX"` (default) – The order of rotation angles is *z*-axis, *y*-axis, *x*-axis.
- `"ZYZ"` – The order of rotation angles is *z*-axis, *y*-axis, *z*-axis.
- `"XYZ"` – The order of rotation angles is *x*-axis, *y*-axis, *z*-axis.

Data Types: `string` | `char`

Output Arguments

`eul` — Euler rotation angles
n-by-3 matrix

Euler rotation angles in radians, returned as an *n*-by-3 array of Euler rotation angles. Each row represents one Euler angle set.

Example: `[0 0 1.5708]`

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also
`eul2tform`

Introduced in R2015a

tform2quat

Extract quaternion from homogeneous transformation

Syntax

```
quat = tform2quat(tform)
```

Description

`quat = tform2quat(tform)` extracts the rotational component from a homogeneous transformation, `tform`, and returns it as a quaternion, `quat`. The translational components of `tform` are ignored. The input homogeneous transformation must be in the premultiply form for transformations.

Examples

Extract Quaternion from Homogeneous Transformation

```
tform = [1 0 0 0; 0 -1 0 0; 0 0 -1 0; 0 0 0 1];
quat = tform2quat(tform)
```

```
quat = 1×4

     0     1     0     0
```

Input Arguments

`tform` — Homogeneous transformation

4-by-4-by-*n* matrix

Homogeneous transformation, specified by a 4-by-4-by-*n* matrix of *n* homogeneous transformations. The input homogeneous transformation must be in the premultiply form for transformations.

Example: `[0 0 1 0; 0 1 0 0; -1 0 0 0; 0 0 0 1]`

Output Arguments

`quat` — Unit quaternion

n-by-4 matrix

Unit quaternion, returned as an *n*-by-4 matrix containing *n* quaternions. Each quaternion, one per row, is of the form $q = [w\ x\ y\ z]$, with w as the scalar number.

Example: `[0.7071 0.7071 0 0]`

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`quat2tform`

Introduced in R2015a

tform2rotm

Extract rotation matrix from homogeneous transformation

Syntax

```
rotm = tform2rotm(tform)
```

Description

`rotm = tform2rotm(tform)` extracts the rotational component from a homogeneous transformation, `tform`, and returns it as an orthonormal rotation matrix, `rotm`. The translational components of `tform` are ignored. The input homogeneous transformation must be in the pre-multiply form for transformations. When using the rotation matrix, premultiply it with the coordinates to be rotated (as opposed to postmultiplying).

Examples

Convert Homogeneous Transformation to Rotation Matrix

```
tform = [1 0 0 0; 0 -1 0 0; 0 0 -1 0; 0 0 0 1];
rotm = tform2rotm(tform)
```

```
rotm = 3×3

     1     0     0
     0    -1     0
     0     0    -1
```

Input Arguments

`tform` — Homogeneous transformation

4-by-4-by-*n* matrix

Homogeneous transformation matrix, specified by a 4-by-4-by-*n* matrix of *n* homogeneous transformations. The input homogeneous transformation must be in the pre-multiply form for transformations.

Example: `[0 0 1 0; 0 1 0 0; -1 0 0 0; 0 0 0 1]`

Output Arguments

`rotm` — Rotation matrix

3-by-3-by-*n* matrix

Rotation matrix, returned as a 3-by-3-by-*n* matrix containing *n* rotation matrices. Each rotation matrix has a size of 3-by-3 and is orthonormal. When using the rotation matrix, premultiply it with the coordinates to be rotated (as opposed to postmultiplying).

Example: `[0 0 1; 0 1 0; -1 0 0]`

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`rotm2tform`

Introduced in R2015a

tform2trvec

Extract translation vector from homogeneous transformation

Syntax

```
trvec = tform2trvec(tform)
```

Description

`trvec = tform2trvec(tform)` extracts the Cartesian representation of translation vector, `trvec` , from a homogeneous transformation, `tform`. The rotational components of `tform` are ignored. The input homogeneous transformation must be in the premultiply form for transformations.

Examples

Extract Translation Vector from Homogeneous Transformation

```
tform = [1 0 0 0.5; 0 -1 0 5; 0 0 -1 -1.2; 0 0 0 1];
trvec = tform2trvec(tform)
```

```
trvec = 1×3

    0.5000    5.0000   -1.2000
```

Input Arguments

`tform` — Homogeneous transformation

4-by-4-by-*n* matrix

Homogeneous transformation, specified by a 4-by-4-by-*n* matrix of *n* homogeneous transformations. The input homogeneous transformation must be in the premultiply form for transformations.

Example: `[0 0 1 0; 0 1 0 0; -1 0 0 0; 0 0 0 1]`

Output Arguments

`trvec` — Cartesian representation of a translation vector

n-by-3 matrix

Cartesian representation of a translation vector, returned as an *n*-by-3 matrix containing *n* translation vectors. Each vector is of the form $t = [x\ y\ z]$.

Example: `[0.5 6 100]`

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`trvec2tform`

Introduced in R2015a

transformMotion

Compute motion quantities between two relatively fixed frames

Syntax

```
[posS,orientS,velS,accS,angvelS] = transformMotion(posSFromP,orientSFromP,
posP)
[ ___ ] = transformMotion(posSFromP,orientSFromP,posP,orientP)
[ ___ ] = transformMotion(posSFromP,orientSFromP,posP,orientP,velP)
[ ___ ] = transformMotion(posSFromP,orientSFromP,posP,orientP,velP,accP)
[ ___ ] = transformMotion(posSFromP,orientSFromP,posP,orientP,velP,accP,
angvelP)
```

Description

`[posS,orientS,velS,accS,angvelS] = transformMotion(posSFromP,orientSFromP, posP)` computes motion quantities of the sensor frame relative to the navigation frame (`posS`, `orientS`, `velS`, `accS`, and `angvelS`) using the position of sensor frame relative to the platform frame, `posSFromP`, the orientation of the sensor frame relative to the platform frame, `orientSFromP`, and the position of the platform frame relative to the navigation frame, `posP`. Note that the position and orientation between the sensor frame and the platform frame are assumed to be fixed. Also, the unspecified quantities between the navigation frame and the platform frame (such as orientation, velocity, and acceleration) are assumed to be zero.

`[ ___ ] = transformMotion(posSFromP,orientSFromP,posP,orientP)` additionally specifies the orientation of the platform frame relative to the navigation frame, `orientP`. The output arguments are the same as those of the previous syntax.

`[ ___ ] = transformMotion(posSFromP,orientSFromP,posP,orientP,velP)` additionally specifies the velocity of the platform frame relative to the navigation frame, `velP`. The output arguments are the same as those of the previous syntax.

`[ ___ ] = transformMotion(posSFromP,orientSFromP,posP,orientP,velP,accP)` additionally specifies the acceleration of the platform frame relative to the navigation frame, `accP`. The output arguments are the same as those of the previous syntax.

`[ ___ ] = transformMotion(posSFromP,orientSFromP,posP,orientP,velP,accP, angvelP)` additionally specifies the angular velocity of the platform frame relative to the navigation frame, `angvelP`. The output arguments are the same as those of the previous syntax.

Examples

Transform State to Sensor Frame

Define the pose, velocity, and acceleration of the platform frame relative to the navigation frame.

```
posPlat = [20 -1 0];
orientPlat = quaternion(1, 0, 0, 0);
velPlat = [0 0 0];
```

```
accPlat = [0 0 0];
angvelPlat = [0 0 1];
```

Define the position and orientation offset of IMU sensor frame relative to the platform frame.

```
posPlat2IMU = [1 2 3];
orientPlat2IMU = quaternion([45 0 0], 'eulerd', 'ZYX', 'frame');
```

Calculate the motion quantities of the sensor frame relative to the navigation frame and print the results.

```
[posIMU, orientIMU, velIMU, accIMU, angvelIMU] ...
    = transformMotion(posPlat2IMU, orientPlat2IMU, ...
    posPlat, orientPlat, velPlat, accPlat, angvelPlat);

fprintf('IMU position is:\n');
```

```
IMU position is:
```

```
fprintf('%.2f %.2f %.2f\n', posIMU);
```

```
21.00 1.00 3.00
```

```
orientIMU
```

```
orientIMU = quaternion
     0.92388 +       0i +       0j + 0.38268k
```

```
velIMU
```

```
velIMU = 1×3

    -2     1     0
```

```
accPlat
```

```
accPlat = 1×3

     0     0     0
```

Input Arguments

posSFromP — Position of sensor frame relative to platform frame

1-by-3 vector of real scalars

Position of the sensor frame relative to the platform frame, specified as a 1-by-3 vector of real scalars.

Example: `[1 2 3]`

orientSFromP — Orientation of sensor frame relative to platform frame

`quaternion` | 3-by-3 rotation matrix

Orientation of the sensor frame relative to the platform frame, specified as a `quaternion` or a 3-by-3 rotation matrix.

Example: `quaternion(1,0,0,0)`

posP — Position of platform frame relative to navigation frame
N-by-3 matrix of real scalars

Position of platform frame relative to navigation frame, specified as an *N*-by-3 matrix of real scalars. *N* is the number of position quantities.

Example: `[1 2 3]`

orientP — Orientation of platform frame relative to navigation frame
N-by-1 array of `quaternion` | 3-by-3-by-*N* array of scalars

Orientation of platform frame relative to navigation frame, specified as an *N*-by-1 array of quaternions, or a 3-by-3-by-*N* array of scalars. Each 3-by-3 matrix must be a rotation matrix. *N* is the number of orientation quantities.

Example: `quaternion(1,0,0,0)`

velP — Velocity of platform frame relative to navigation frame
N-by-3 matrix of real scalars

Velocity of platform frame relative to navigation frame, specified as an *N*-by-3 matrix of real scalars. *N* is the number of velocity quantities.

Example: `[ 4 8 6]`

accP — Acceleration of platform frame relative to navigation frame
N-by-3 matrix of real scalars

Acceleration of platform frame relative to navigation frame, specified as an *N*-by-3 matrix of real scalars. *N* is the number of acceleration quantities.

Example: `[4 8 6]`

angvelP — Angular velocity of platform frame relative to navigation frame
N-by-3 matrix of real scalars

Angular velocity of platform frame relative to navigation frame, specified as an *N*-by-3 matrix of real scalars. *N* is the number of angular velocity quantities.

Example: `[4 2 3]`

Output Arguments

posS — Position of sensor frame relative to navigation frame
N-by-3 matrix of real scalars

Position of sensor frame relative to navigation frame, returned as an *N*-by-3 matrix of real scalars. *N* is the number of position quantities specified by the `posP` input.

orientS — Orientation of sensor frame relative to navigation frame
N-by-1 array of `quaternion` | 3-by-3-by-*N* array of scalars

Orientation of sensor frame relative to navigation frame, returned as an *N*-by-1 array of quaternions, or a 3-by-3-by-*N* array of scalars. *N* is the number of orientation quantities specified by the `orientP` input. The returned orientation quantity type is same with the `orientP` input.

`velS` — Velocity of sensor frame relative to navigation frame
N-by-3 matrix of real scalars

Velocity of sensor frame relative to navigation frame, returned as an *N*-by-3 matrix of real scalars. *N* is the number of position quantities specified by the `velP` input.

`accS` — Acceleration of sensor frame relative to navigation frame
N-by-3 matrix of real scalars

Acceleration of sensor frame relative to navigation frame, returned as an *N*-by-3 matrix of real scalars. *N* is the number of position quantities specified by the `accP` input.

`angvelS` — Angular velocity of sensor frame relative to navigation frame
N-by-3 matrix of real scalars

Angular velocity of sensor frame relative to navigation frame, returned as an *N*-by-3 matrix of real scalars. *N* is the number of position quantities specified by the `angvelP` input.

More About

Motion Quantities Used in `transformMotion`

The `transformMotion` function calculates the motion quantities of the sensor frame (*S*), which is fixed on a rigid platform, relative to the navigation frame (*N*) using the mounting information of the sensor on the platform and the motion information of the platform frame (*P*).

As shown in the figure, the position and orientation of the platform frame and the sensor frame are fixed on the platform. The position of the sensor frame relative to the platform frame is p_{SP}, and the orientation of the sensor frame relative to the platform frame is r_{SP}. Since the two frames are both fixed, p_{SP} and r_{SP} are constant.

To compute the motion quantities of the sensor frame relative to the navigation frame, the quantities describing the motion of the platform frame relative to the navigation frame are required. These quantities include: the platform position (p_{PN}), orientation (r_{PN}), velocity, acceleration, angular velocity, and angular acceleration relative to the navigation frame. You can specify these quantities through the function input arguments except the angular acceleration, which is always assumed to be zero in the function. The unspecified quantities are also assumed to be zero.

See Also

`quaternion` | `rotvec`

Introduced in R2020a

transformScan

Transform laser scan based on relative pose

Syntax

```
transScan = transformScan(scan,relPose)

[transRanges,transAngles] = transformScan(ranges,angles,relPose)
```

Description

`transScan = transformScan(scan,relPose)` transforms the laser scan specified in `scan` by using the specified relative pose, `relPose`.

`[transRanges,transAngles] = transformScan(ranges,angles,relPose)` transforms the laser scan specified in `ranges` and `angles` by using the specified relative pose, `relPose`.

Examples

Transform Laser Scans

Create a `lidarScan` object. Specify the ranges and angles as vectors.

```
refRanges = 5*ones(1,300);
refAngles = linspace(-pi/2,pi/2,300);
refScan = lidarScan(refRanges,refAngles);
```

Translate the laser scan by an `[x y]` offset of `(0.5,0.2)`.

```
transformedScan = transformScan(refScan,[0.5 0.2 0]);
```

Rotate the laser scan by 20 degrees.

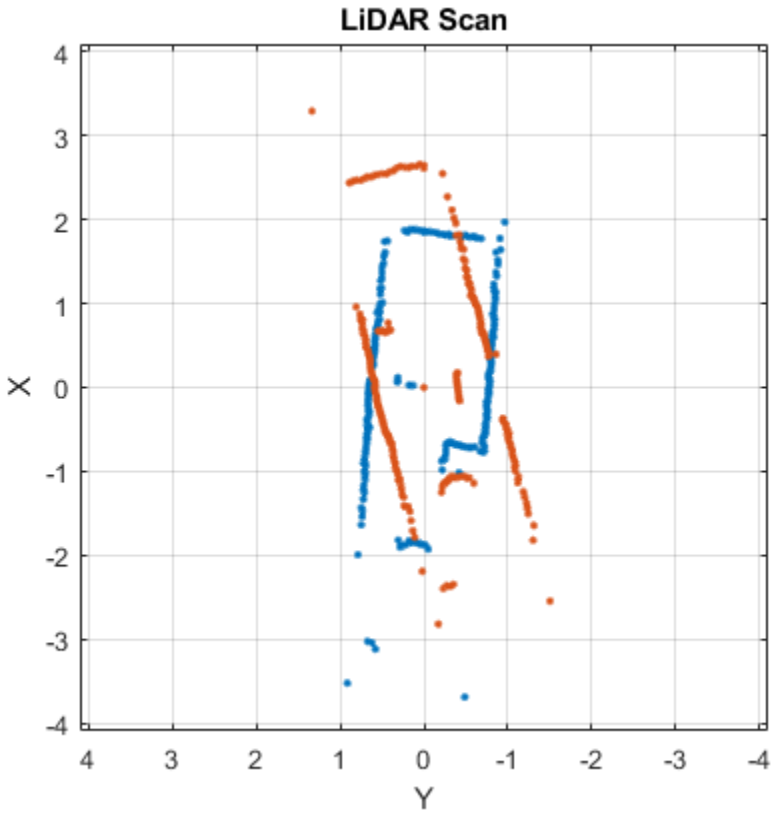
```
rotateScan = transformScan(refScan,[0,0,deg2rad(20)]);
```

Use Scan Matching to Transform Scans

Use the `matchScans` function to find the relative transformation between two lidar scans. Then, transform the second scan into the coordinate frame of the first scan.

Load a pair of lidar scans as a pair of `lidarScan` objects. They are two scans of the same scene with a change in relative pose.

```
load tb3_scanPair.mat
plot(s1)
hold on
plot(s2)
hold off
```



The relative pose is estimated from an odometry sensor and provided as a variable, `initGuess`, as `[x y theta]`.

```
disp(initGuess)
```

```
   -0.7000    0.1500   -0.3254
```

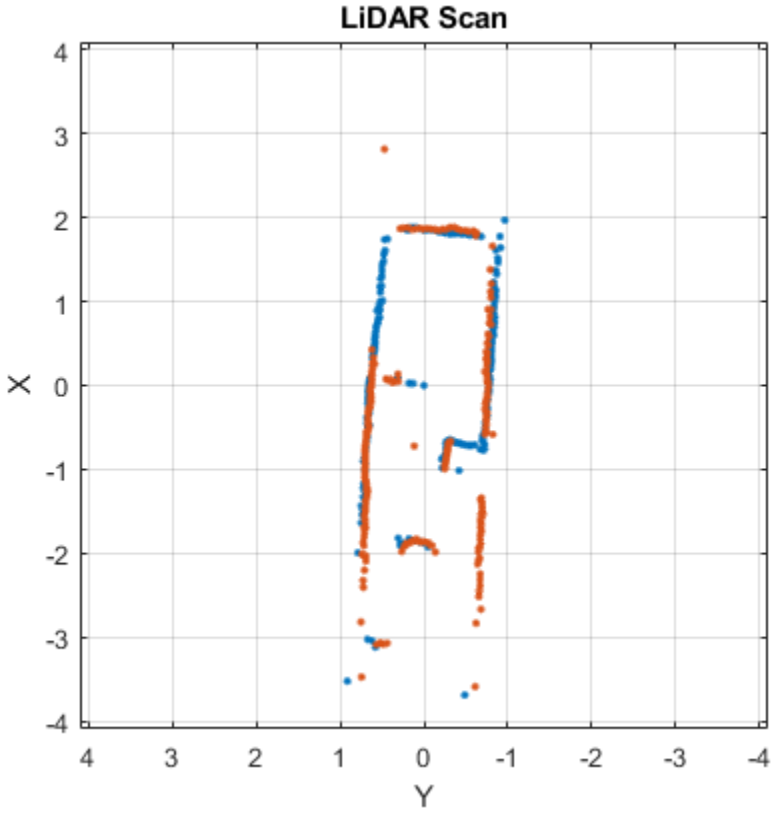
Use scan matching to find the relative pose between the two laser scans and specify the initial guess for the pose.

```
pose = matchScans(s2,s1,"InitialPose",initGuess);
disp(pose)
```

```
   -0.7213    0.1160   -0.2854
```

Transform the second scan to the coordinate frame of the first scan. Plot the two scans to see that they now overlap.

```
s2Transformed = transformScan(s2,pose);
plot(s1)
hold on
plot(s2Transformed)
hold off
```



Input Arguments

`scan` — Lidar scan readings
`lidarScan` object

Lidar scan readings, specified as a `lidarScan` object.

`ranges` — Range values from scan data
vector

Range values from scan data, specified as a vector in meters. These range values are distances from a sensor at specified `angles`. The vector must be the same length as the corresponding `angles` vector.

`angles` — Angle values from scan data
vector

Angle values from scan data, specified as a vector in radians. These angle values are the specific angles of the specified `ranges`. The vector must be the same length as the corresponding `ranges` vector.

`relPose` — Relative pose of current scan
`[x y theta]`

Relative pose of current scan, specified as `[x y theta]`, where `[x y]` is the translation in meters and `theta` is the rotation in radians.

Output Arguments

`transScan` — Transformed lidar scan readings
`lidarScan` object

Transformed lidar scan readings, specified as a `lidarScan` object.

`transRanges` — Range values of transformed scan
vector

Range values of transformed scan, returned as a vector in meters. These range values are distances from a sensor at specified `transAngles`. The vector is the same length as the corresponding `transAngles` vector.

`transAngles` — Angle values from scan data
vector

Angle values of transformed scan, returned as a vector in radians. These angle values are the specific angles of the specified `transRanges`. The vector is the same length as the corresponding `ranges` vector.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also
`matchScans` | `transformScan` | `controllerVFH` | `monteCarloLocalization`

Topics
“Estimate Robot Pose with Scan Matching”

Introduced in R2017a

trimLoopClosures

Optimize pose graph and remove bad loop closures

Syntax

```
poseGraphUpdated = trimLoopClosures(poseGraphObj,trimParams,solverOptions)
[poseGraphUpdated,trimInfo] = trimLoopClosures(poseGraphObj,trimParams,
solverOptions)
```

Description

`poseGraphUpdated = trimLoopClosures(poseGraphObj,trimParams,solverOptions)` optimizes the pose graph to best satisfy the edge constrains and removes any bad loop closure edges based on the residual error parameters specified in `trimParams`. Create the `solverOptions` input using the `poseGraphSolverOptions` function.

The function implements the graduated non-convexity (GNC) method with truncated least squares (TLS) robust cost in combination with the non-minimal pose graph solver [1] on page 1-229.

`[poseGraphUpdated,trimInfo] = trimLoopClosures(poseGraphObj,trimParams,solverOptions)` returns additional information related to the trimming process.

Examples

Optimize and Trim Loop Closures For 2-D Pose Graphs

Optimize a pose graph based on the nodes and edge constraints. Trim loop closed based on their edge residual errors.

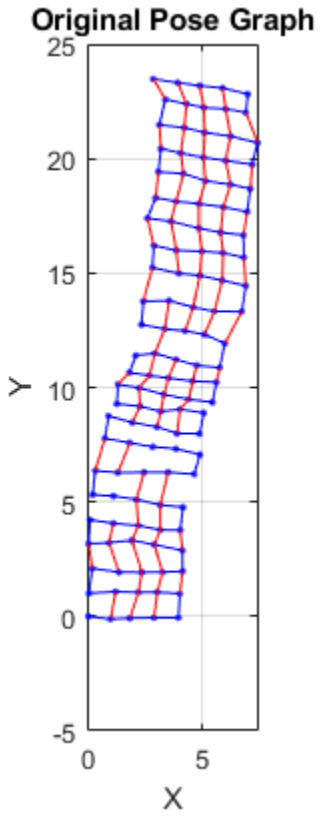
Load the data set that contains a 2-D pose graph. Inspect the `poseGraph` object to view the number of nodes and loop closures.

```
load grid-2d-posegraph.mat pg
disp(pg)
```

```
  poseGraph with properties:

              NumNodes: 120
              NumEdges: 193
    NumLoopClosureEdges: 74
     LoopClosureEdgeIDs: [120 121 122 123 124 125 126 127 128 129 130 ... ]
        LandmarkNodeIDs: [1x0 double]
```

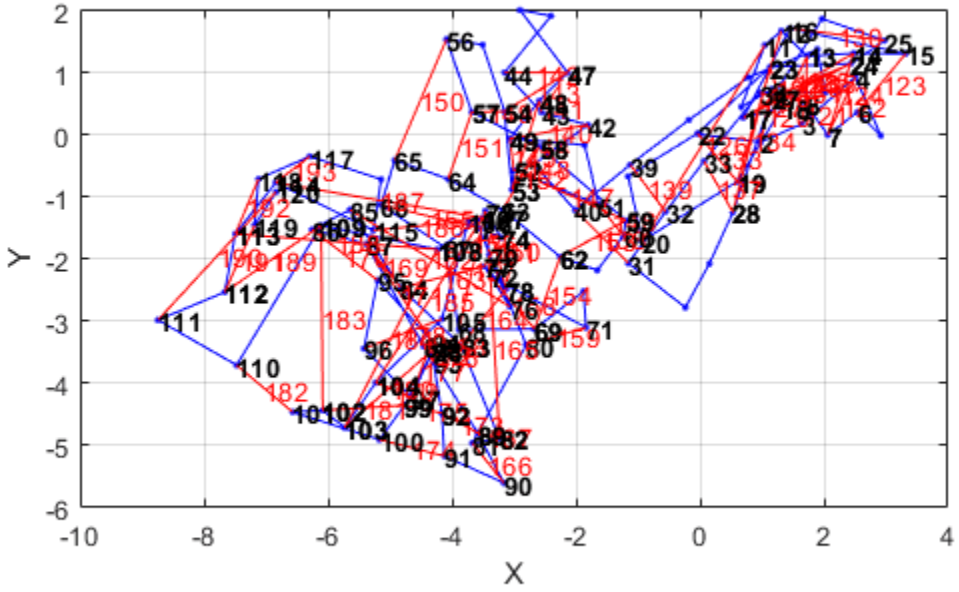
Plot the pose graph with IDs off. Red lines indicate loop closures identified in the dataset. The poses in the graph should follow a grid pattern, but show evidence of drift over time.

```
show(pg,'IDs','off');
title('Original Pose Graph')
```



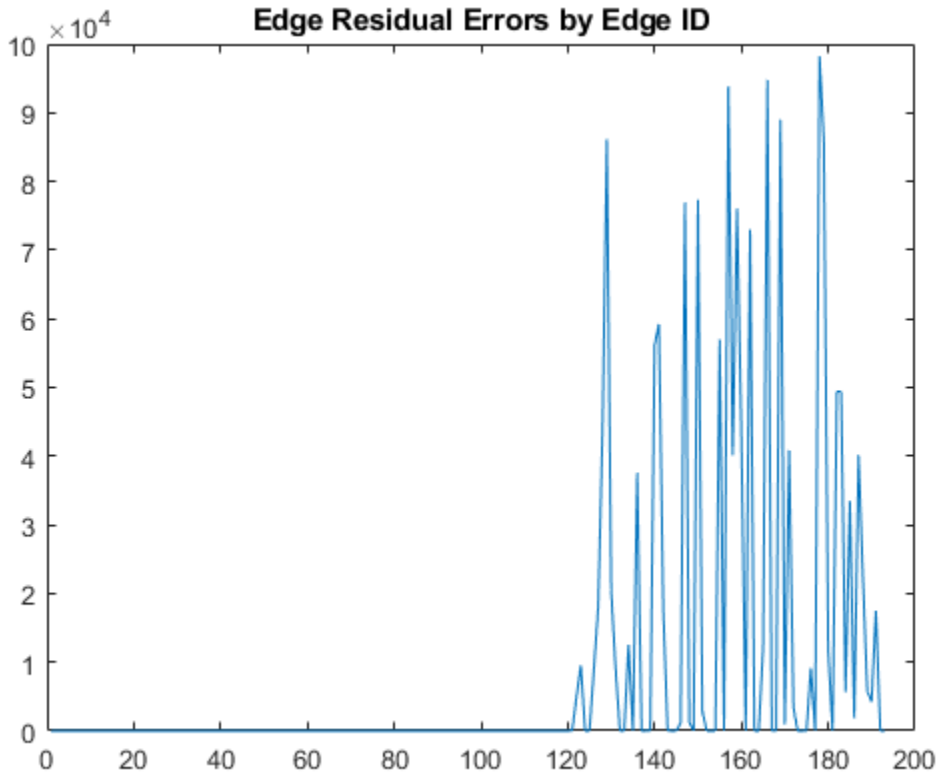
Optimize the pose grap using the `optimizePoseGraph` function. By default, this function uses the `"builtin-trust-region"` solver. Because the pose graph contains some bad loop closures, the resulting pose graph is actual not desirable.

```
pgOptim = optimizePoseGraph(pg);
figure;
show(pgOptim);
```



Look at the edge residual errors for the original pose graph. Large outlier error values at the end indicate bad loop closures.

```
resErrorVec = edgeResidualErrors(pg);
plot(resErrorVec);
title('Edge Residual Errors by Edge ID')
```



Certain loop closures should be trimmed from the pose graph based on their residual error. Use the `trimLoopClosures` function to trim therse bad loop closures. Set the maximum and truncation threshold for the trimmer parameters. This threshold is set based on the measurement accuracy and should be tuned for your system.

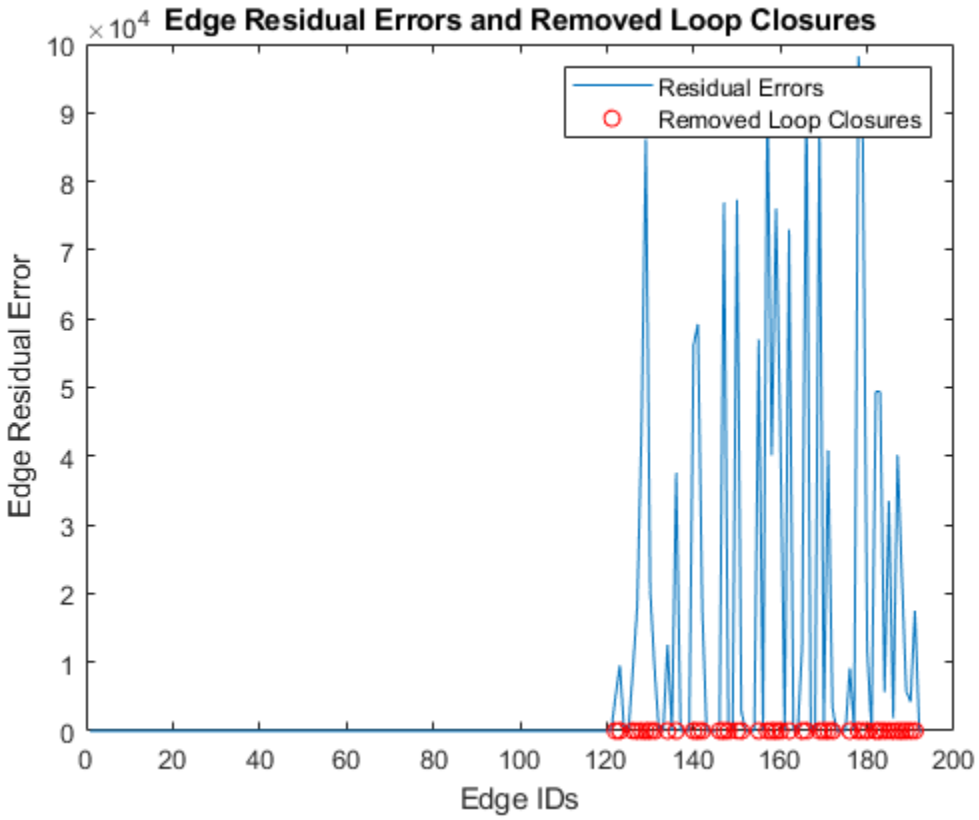
```
trimParams.MaxIterations = 100;
trimParams.TruncationThreshold = 25;

solverOptions = poseGraphSolverOptions;
```

Use the `trimLoopClosures` function with the trimmer parameters and solver options.

```
[pgNew, trimInfo, debugInfo] = trimLoopClosures(pg,trimParams,solverOptions);
```

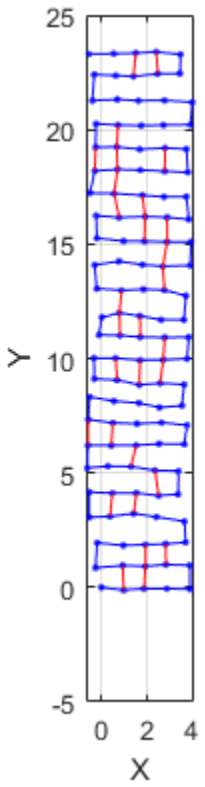
From the `trimInfo` output, plot the loop closures removed from the optimized pose graph. By plotting with the residual errors plot before, you can see the large error loop closures were removed.

```
removedLCs = trimInfo.LoopClosuresToRemove;

hold on
plot(removedLCs,zeros(length(removedLCs)),'or')
title('Edge Residual Errors and Removed Loop Closures')
legend('Residual Errors', 'Removed Loop Closures')
xlabel('Edge IDs')
ylabel('Edge Residual Error')
hold off
```



Show the new pose graph with the bad loop closures trimmed.

```
show(pgNew,"IDs","off");
```



Input Arguments

`poseGraphObj` — Pose graph
`poseGraph` object | `poseGraph3D` object

Pose graph, specified as a `poseGraph` or `poseGraph3D` object.

`trimParams` — Residual error parameters for trimming
structure

Residual error parameters for trimming loop closures, specified as a structure with fields:

- `MaxIterations` — Maximum number of iterations allowed for loop closure trimming, specified as a positive integer. In one trimming iteration, the pose graph is optimized based on the solver options and any edges outside the `TruncationThreshold` are trimmed.
- `TruncationThreshold` — Maximum allowed residual error for an edge. This value depends heavily on the pose graph you specify in `poseGraphObj`. To find a proper threshold based on all the errors, use the `edgeResidualErrors` function for the pose graph.

Example: `struct('MaxIterations',10,'TruncationThreshold',20)`

Data Types: `struct`

`solverOptions` — Pose graph solver options
`poseGraphSolverOptions` parameters

Pose graph solver options, specified as a set of parameters generated by calling the `poseGraphSolverOptions` function. The function generates a set of solver options with default values for the specified pose graph solver type:

```
pgSolverTrustRegion = poseGraphSolverOptions('builtin-trust-region')

pgSolverTrustRegion =

TrustRegion (builtin-trust-region-dogleg) options:

               MaxIterations: 300
                     MaxTime: 10
           FunctionTolerance: 1.0000e-08
           GradientTolerance: 5.0000e-09
               StepTolerance: 1.0000e-12
    InitialTrustRegionRadius: 100
               VerboseOutput: 'off'

pgSolverG2o = poseGraphSolverOptions('g2o-levenberg-marquardt')

pgSolverG2o =

G2oLevenbergMarquardt (g2o-levenberg-marquardt) options:

        MaxIterations: 300
              MaxTime: 10
    FunctionTolerance: 1.0000e-09
        VerboseOutput: 'off'
```

Modify the options to tune the solver parameters using dot notation.

```
pgSolverG2o.MaxIterations = 200;
```

Output Arguments

`poseGraphUpdated` — Pose graph with trimmed looped closures

`poseGraph` object | `poseGraph3D` object

Pose graph with trimmed looped closures, specified as a `poseGraph` or `poseGraph3D` object.

`trimInfo` — Information from trimming process

structure

Information from trimming process, returned as a structure with fields:

- `LoopClosuresToRemove` — Loop closure edge IDs to remove from the input `poseGraphObj`. These loop closures are removed in the output `poseGraphUpdated`.
- `Iterations` — Number of trimming iterations performed.

References

[1] Yang, Heng, et al. “Graduated Non-Convexity for Robust Spatial Perception: From Non-Minimal Solvers to Global Outlier Rejection.” *IEEE Robotics and Automation Letters*, vol. 5, no. 2, Apr. 2020, pp. 1127–34. *DOI.org (Crossref)*, doi:10.1109/LRA.2020.2965893.

See Also

Functions
poseGraphSolverOptions | edgeResidualErrors | removeEdges | edgeNodePairs | edgeConstraints

Objects
poseGraph | poseGraph3D | lidarSLAM

Introduced in R2020b

trvec2tform

Convert translation vector to homogeneous transformation

Syntax

```
tform = trvec2tform(trvec)
```

Description

`tform = trvec2tform(trvec)` converts the Cartesian representation of a translation vector, `trvec`, to the corresponding homogeneous transformation, `tform`. When using the transformation matrix, premultiply it with the coordinates to be transformed (as opposed to postmultiplying).

Examples

Convert Translation Vector to Homogeneous Transformation

```
trvec = [0.5 6 100];
tform = trvec2tform(trvec)
```

```
tform = 4×4

    1.0000         0         0    0.5000
         0    1.0000         0    6.0000
         0         0    1.0000  100.0000
         0         0         0    1.0000
```

Input Arguments

`trvec` — Cartesian representation of a translation vector

n-by-3 matrix

Cartesian representation of a translation vector, specified as an *n*-by-3 matrix containing *n* translation vectors. Each vector is of the form $t = [x\ y\ z]$.

Example: `[0.5 6 100]`

Output Arguments

`tform` — Homogeneous transformation

4-by-4-by-*n* matrix

Homogeneous transformation matrix, returned as a 4-by-4-by-*n* matrix of *n* homogeneous transformations. When using the rotation matrix, premultiply it with the coordinates to be rotated (as opposed to postmultiplying).

Example: `[0 0 1 0; 0 1 0 0; -1 0 0 0; 0 0 0 1]`

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`tform2trvec`

Introduced in R2015a

tunernoise

Noise structure of fusion filter

Syntax

```
noiseStruct = tunernoise(filterName)
```

Description

`noiseStruct = tunernoise(filterName)` returns the measurement noise structure for the filter with name specified by the `filterName` input.

Examples

Obtain Measurement Noise Structure of `insfilterAsync`

Obtain the measurement noise structure of the `insfilterAsync` object.

```
noiseStruct = tunernoise('insfilterAsync')
```

```
noiseStruct = struct with fields:
    AccelerometerNoise: 1
        GyroscopeNoise: 1
     MagnetometerNoise: 1
      GPSPositionNoise: 1
      GPSVelocityNoise: 1
```

Tune `insfilterAsync` to Optimize Pose Estimate

Load the recorded sensor data and ground truth data.

```
load('insfilterAsyncTuneData.mat');
```

Create timetables for the sensor data and the truth data.

```
sensorData = timetable(Accelerometer, Gyroscope, ...
    Magnetometer, GPSPosition, GPSVelocity, 'SampleRate', 100);
groundTruth = timetable(Orientation, Position, ...
    'SampleRate', 100);
```

Create an `insfilterAsync` filter object that has a few noise properties.

```
filter = insfilterAsync('State', initialState, ...
    'StateCovariance', initialStateCovariance, ...
    'AccelerometerBiasNoise', 1e-7, ...
    'GyroscopeBiasNoise', 1e-7, ...
    'MagnetometerBiasNoise', 1e-7, ...
    'GeomagneticVectorNoise', 1e-7);
```

Create a tuner configuration object for the filter. Set the maximum iterations to two. Also, set the tunable parameters as the unspecified properties.

```
config = tunerconfig('insfilterAsync','MaxIterations',8);
config.TunableParameters = setdiff(config.TunableParameters, ...
    {'GeomagneticVectorNoise', 'AccelerometerBiasNoise', ...
    'GyroscopeBiasNoise', 'MagnetometerBiasNoise'});
config.TunableParameters
```

```
ans = 1×10 string
    "AccelerationNoise"    "AccelerometerNoise"    "AngularVelocityNoise"    "GPSPositionNoise"
```

Use the tuner noise function to obtain a set of initial sensor noises used in the filter.

```
measNoise = tunernoise('insfilterAsync')
```

```
measNoise = struct with fields:
    AccelerometerNoise: 1
        GyroscopeNoise: 1
     MagnetometerNoise: 1
      GPSPositionNoise: 1
      GPSVelocityNoise: 1
```

Tune the filter and obtain the tuned parameters.

```
tunedParams = tune(filter,measNoise,sensorData,groundTruth,config);
```

```
    Iteration    Parameter               Metric
    _________    _________               ______
    1            AccelerationNoise       2.1345
    1            AccelerometerNoise      2.1264
    1            AngularVelocityNoise    1.9659
    1            GPSPositionNoise        1.9341
    1            GPSVelocityNoise        1.8420
    1            GyroscopeNoise          1.7589
    1            MagnetometerNoise       1.7362
    1            PositionNoise           1.7362
    1            QuaternionNoise         1.7218
    1            VelocityNoise           1.7218
    2            AccelerationNoise       1.7190
    2            AccelerometerNoise      1.7170
    2            AngularVelocityNoise    1.6045
    2            GPSPositionNoise        1.5948
    2            GPSVelocityNoise        1.5323
    2            GyroscopeNoise          1.4803
    2            MagnetometerNoise       1.4703
    2            PositionNoise           1.4703
    2            QuaternionNoise         1.4632
    2            VelocityNoise           1.4632
    3            AccelerationNoise       1.4596
    3            AccelerometerNoise      1.4548
    3            AngularVelocityNoise    1.3923
    3            GPSPositionNoise        1.3810
    3            GPSVelocityNoise        1.3322
    3            GyroscopeNoise          1.2998
    3            MagnetometerNoise       1.2976
    3            PositionNoise           1.2976
```

```
     3           QuaternionNoise         1.2943
     3           VelocityNoise           1.2943
     4           AccelerationNoise       1.2906
     4           AccelerometerNoise      1.2836
     4           AngularVelocityNoise    1.2491
     4           GPSPositionNoise        1.2258
     4           GPSVelocityNoise        1.1880
     4           GyroscopeNoise          1.1701
     4           MagnetometerNoise       1.1698
     4           PositionNoise           1.1698
     4           QuaternionNoise         1.1688
     4           VelocityNoise           1.1688
     5           AccelerationNoise       1.1650
     5           AccelerometerNoise      1.1569
     5           AngularVelocityNoise    1.1454
     5           GPSPositionNoise        1.1100
     5           GPSVelocityNoise        1.0778
     5           GyroscopeNoise          1.0709
     5           MagnetometerNoise       1.0675
     5           PositionNoise           1.0675
     5           QuaternionNoise         1.0669
     5           VelocityNoise           1.0669
     6           AccelerationNoise       1.0634
     6           AccelerometerNoise      1.0549
     6           AngularVelocityNoise    1.0549
     6           GPSPositionNoise        1.0180
     6           GPSVelocityNoise        0.9866
     6           GyroscopeNoise          0.9810
     6           MagnetometerNoise       0.9775
     6           PositionNoise           0.9775
     6           QuaternionNoise         0.9768
     6           VelocityNoise           0.9768
     7           AccelerationNoise       0.9735
     7           AccelerometerNoise      0.9652
     7           AngularVelocityNoise    0.9652
     7           GPSPositionNoise        0.9283
     7           GPSVelocityNoise        0.8997
     7           GyroscopeNoise          0.8947
     7           MagnetometerNoise       0.8920
     7           PositionNoise           0.8920
     7           QuaternionNoise         0.8912
     7           VelocityNoise           0.8912
     8           AccelerationNoise       0.8885
     8           AccelerometerNoise      0.8811
     8           AngularVelocityNoise    0.8807
     8           GPSPositionNoise        0.8479
     8           GPSVelocityNoise        0.8238
     8           GyroscopeNoise          0.8165
     8           MagnetometerNoise       0.8165
     8           PositionNoise           0.8165
     8           QuaternionNoise         0.8159
     8           VelocityNoise           0.8159
```

Fuse the sensor data using the tuned filter.

```
dt = seconds(diff(groundTruth.Time));
N = size(sensorData,1);
qEst = quaternion.zeros(N,1);
```

```
posEst = zeros(N,3);
% Iterate the filter for prediction and correction using sensor data.
for ii=1:N
    if ii ~= 1
        predict(filter, dt(ii-1));
    end
    if all(~isnan(Accelerometer(ii,:)))
        fuseaccel(filter,Accelerometer(ii,:), ...
            tunedParams.AccelerometerNoise);
    end
    if all(~isnan(Gyroscope(ii,:)))
        fusegyro(filter, Gyroscope(ii,:), ...
            tunedParams.GyroscopeNoise);
    end
    if all(~isnan(Magnetometer(ii,1)))
        fusemag(filter, Magnetometer(ii,:), ...
            tunedParams.MagnetometerNoise);
    end
    if all(~isnan(GPSPosition(ii,1)))
        fusegps(filter, GPSPosition(ii,:), ...
            tunedParams.GPSPositionNoise, GPSVelocity(ii,:), ...
            tunedParams.GPSVelocityNoise);
    end
    [posEst(ii,:), qEst(ii,:)] = pose(filter);
end
```

Compute the RMS errors.

```
orientationError = rad2deg(dist(qEst, Orientation));
rmsorientationError = sqrt(mean(orientationError.^2))
```

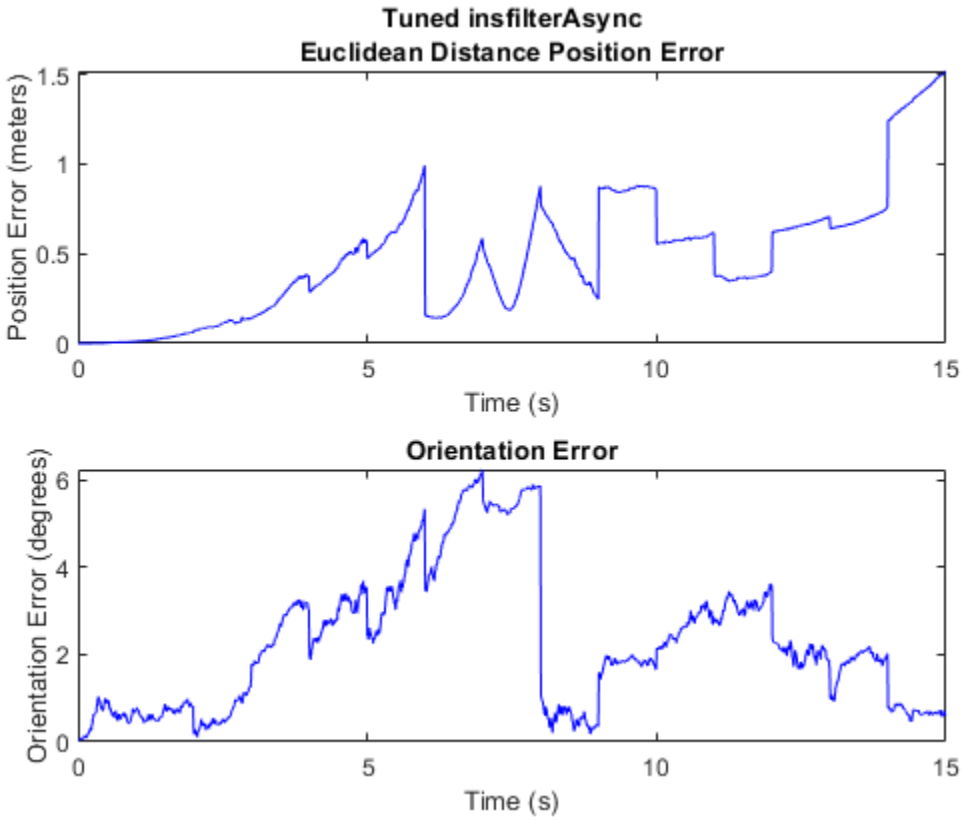
```
rmsorientationError = 2.7801
```

```
positionError = sqrt(sum((posEst - Position).^2, 2));
rmspositionError = sqrt(mean( positionError.^2))
```

```
rmspositionError = 0.5966
```

Visualize the results.

```
figure();
t = (0:N-1)./ groundTruth.Properties.SampleRate;
subplot(2,1,1)
plot(t, positionError, 'b');
title("Tuned insfilterAsync" + newline + "Euclidean Distance Position Error")
xlabel('Time (s)');
ylabel('Position Error (meters)')
subplot(2,1,2)
plot(t, orientationError, 'b');
title("Orientation Error")
xlabel('Time (s)');
ylabel('Orientation Error (degrees)');
```



Input Arguments

`filterName` — Name of fusion filter

`'insfilterAsync'` | `'ahrs10filter'` | `'insfilterMARG'` | `'insfilterNonholonomic'` | `'insfitlerErrorState'`

Name of fusion filter, specified as specified as one of these:

- `'ahrs10filter'`
- `'insfilterAsync'`
- `'insfilterMARG'`
- `'insfitlerErrorState'`
- `'insfilterNonholonomic'`

.

Output Arguments

`noiseStruct` — Structure of measurement noise

structure

Structure of measurement noise, returned as a structure. For the `insfilterAsync` object, the structure contains these fields.

Field	Description	Default
`AccelerometerNoise`	Variance of accelerometer noise, specified as a scalar in $(m^2/s)^2$	1
`GyroscopeNoise`	Variance of gyroscope noise, specified as a scalar in $(rad/s)^2$	1
`MagnetometerNoise`	Variance of magnetometer noise, specified as a scalar in $(\mu T)^2$.	1
`GPSPositionNoise`	Variance of GPS position noise, specified as a scalar in m^2	1
`GPSVelocityNoise`	Standard deviation of GPS velocity noise, specified as a scalar in $(m/s)^2$	1

To use this structure with the `tune` function, change the values of the noise to proper values as initial guesses for tuning the noise.

See Also

Introduced in R2020b

tunerPlotPose

Plot filter pose estimates during tuning

Syntax

```
stopTuning = tunerPlotPose(params,tunerValues)
```

Description

`stopTuning = tunerPlotPose(params,tunerValues)` plots the current pose estimate, consisting of orientation (and possibly position, depending on the filter), and the ground truth values. `params` contains the best estimates of the filter parameters during the current tuning iteration. `tunerValues` contains information on the tuner configuration, sensor data, and ground truth data. Use this function as the value for the `OutputFcn` property of the `tunerconfig` object to plot the tuning results during iterations.

Examples

Visualize Tuning Results Using `tunerPlotPose`

Create a `tunerconfiguration` object. Set the `tunerPlotPose` function as the output function of the object.

```
tc = tunerconfig('imufilter','OutputFcn',@tunerPlotPose)
```

```
tc =
  tunerconfig with properties:

               Filter: "imufilter"
    TunableParameters: ["AccelerometerNoise"    "GyroscopeNoise"    ...    ]
          StepForward: 1.1000
         StepBackward: 0.5000
        MaxIterations: 20
       ObjectiveLimit: 0.1000
    FunctionTolerance: 0
              Display: iter
                 Cost: RMS
            OutputFcn: @tunerPlotPose
```

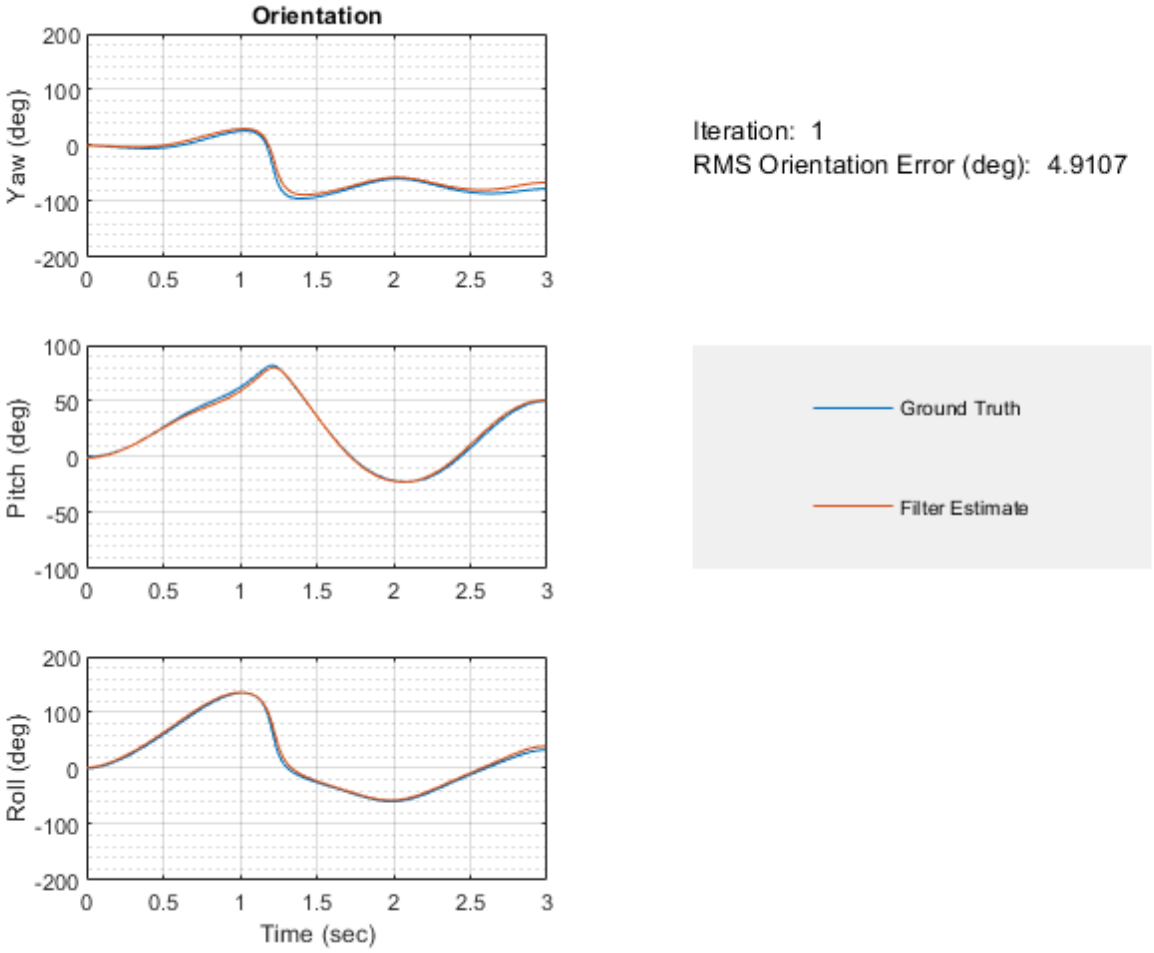
Load prerecorded sensor data.

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

Tune an `imufilter` object using the sensor data. The truth data and the estimates are shown in a figure.

```
tune(imufilter,ld.sensorData,ld.groundTruth,tc)
```

```
Iteration    Parameter                    Metric
_________    _________                    ______
1            AccelerometerNoise           0.0857
```



Input Arguments

`params` — Estimates of filter parameters

structure

Estimates of filter parameters during the current iteration of the tuning process, specified as a structure. The structure contains one field for every public property of the filter and additional fields for any required measurement noise. The exact field names vary depending on the filter being tuned.

`tunerValues` — Tuner values

structure

Tuner values, specified as a structure. The structure has these fields:

Field Name	Description
`Iteration`	Iteration count of the tuner, specified as a positive integer
`SensorData`	Sensor data input to the `tune` function
`GroundTruth`	Ground truth input to the `tune` function
`Configuration`	`tunerconfig` object used for tuning
`Cost`	Tuning cost at the end of the current iteration

Output Arguments

`stopTuning` — Stop tuning process

`false`

Stop the tuning process, returned as `false`. As a result, using the `tunerPlotPose` function as the output function of a `tunerconfig` object never terminates the tuning process of a fusion filter.

See Also

`tunerconfig` | `tunernoise` | `imufilter` | `ahrsfilter` | `ahrs10filter` | `insfilterMARG` | `insfilterAsync` | `insfilterErrorState` | `insfilterNonholonomic`

Introduced in R2021a

writeBinaryOccupancyGrid

Write values from grid to ROS message

Syntax

```
writeBinaryOccupancyGrid(msg,map)
```

Description

`writeBinaryOccupancyGrid(msg,map)` writes occupancy values and other information to the ROS message, `msg`, from the binary occupancy grid, `map`.

Input Arguments

`map` — Binary occupancy grid
`binaryOccupancyMap` object handle

Binary occupancy grid, specified as a `binaryOccupancyMap` object handle. `map` is converted to a `'nav_msgs/OccupancyGrid'` message on the ROS network. `map` is an object with a grid of binary values, where `1` indicates an occupied location and `0` indications an unoccupied location.

`msg` — `'nav_msgs/OccupancyGrid'` ROS message
`OccupancyGrid` object handle

`'nav_msgs/OccupancyGrid'` ROS message, specified as a `OccupancyGrid` object handle.

See Also

Functions
`rosReadBinaryOccupancyGrid` | `rosReadOccupancyMap3D` | `rosReadOccupancyGrid` | `rosWriteOccupancyGrid`

Introduced in R2015a

writeBytes

Write raw commands to the GPS receiver

Syntax

```
writeBytes(gps,cmdArray)
```

Description

`writeBytes(gps,cmdArray)` writes raw commands specified by `cmdArray` to configure the GPS module.

Examples

Write Configuration Commands to GPS Receiver

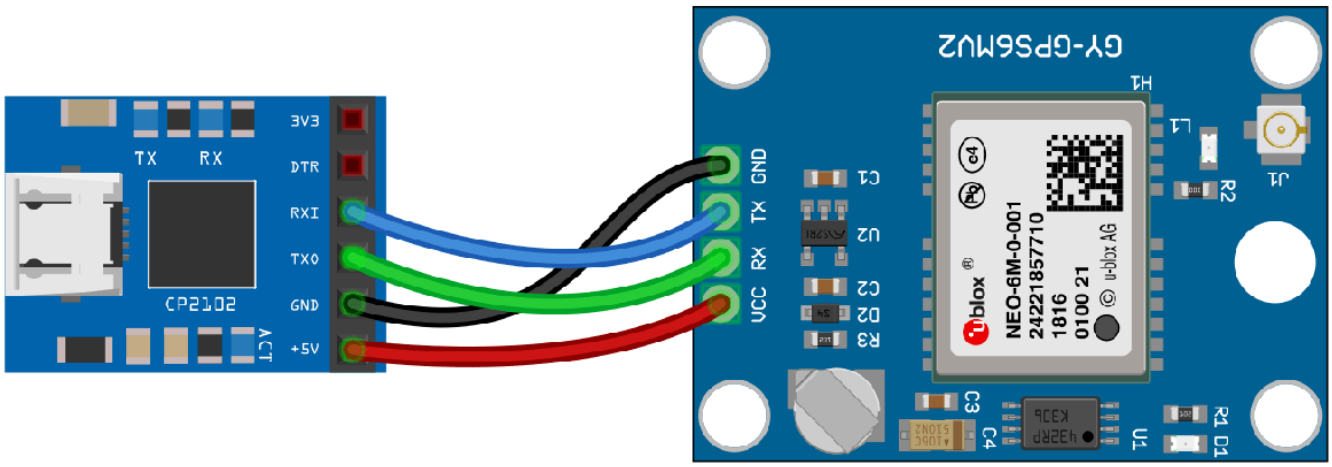
Write configuration commands to the GPS receiver connected to the host computer using `serialport` object.

Required Hardware

To run this example, you need:

- UBlox Neo-6M GPS module
- GPS antenna
- USB to UART module
- USB cable
- Connecting wires

Hardware Connection



Connect the pins on the UBlox Neo-6M GPS module to the pins on your USB to UART module. The connections are:

- VCC - +5V
- RX - TXO
- TX - RXI
- GND - GND

Connect the GPS antenna to the GPS module. Connect the USB to UART module to the host computer with a USB cable. GPS Fix can be easily acquired in locations that have a clear view of the sky. Wait for the GPS module to acquire satelite signals (Fix).This can be verified by checking the Fix LED (D1) of your GPS module.

Create GPS Object

Connect to the GPS receiver using `serialport` object. Specify the port name and the baud rate.

```
s = serialport('COM4',9600)
```

```
s =
  Serialport with properties:

                Port: "COM4"
            BaudRate: 9600
    NumBytesAvailable: 0

  Show all properties, functions
```

```
gps = gpsdev(s)
```

```
gps =
  gpsdev with properties:

                         SerialPort: COM4
                           BaudRate: 9600 (bits/s)

                    SamplesPerRead: 1
                          ReadMode: "latest"
                       SamplesRead: 0
Show all properties all functions
```

Write Configuration Commands

In the default configuration the GPS receiver returns the following NMEA messages: GPRMC, GPVTG, GPGGA, GPGSA, GPGSV, and GPGLL. The receiver can be configured to have a user defined set of output messages.

Read few lines of default messages from the serial port the GPS receiver is connected.

```
for i = 1:10
data = readline(s);
disp(data);
end
```

```
$GPRMC,,V,,,,,,,,,,N*53
$GPVTG,,,,,,,,,N*30
```

```
$GPGGA,,,,,,0,00,99.99,,,,,,*48
$GPGSA,A,1,,,,,,,,,,,,,99.99,99.99,99.99*30
$GPGSV,2,1,08,01,,,18,08,,,12,09,,,12,15,,,19*77
$GPGSV,2,2,08,23,,,13,24,,,09,25,,,10,27,,,25*79
$GPGLL,,,,,,V,N*64
$GPRMC,,V,,,,,,,,,,N*53
$GPVTG,,,,,,,,,N*30
$GPGGA,,,,,,0,00,99.99,,,,,,*48
```

Write the version monitor command to the GPS receiver to return the software and hardware version of the GPS receiver.

```
configCMD = [0xB5 0x62 0x0A 0x04 0x00 0x00 0x0E 0x34];
% writeBytes(gps,cfg)
write(s,configCMD,'uint8')
```

Read few lines of messages again to verify the version message.

```
for i = 1:10
data = readline(s);
disp(data);
end
```

```
$GPGSA,A,1,,,,,,,,,,,,,99.99,99.99,99.99*30
$GPGSV,2,1,05,01,,,13,09,,,11,15,,,16,23,,,12*74
$GPGSV,2,2,05,25,,,10*7A
$GPGLL,,,,,,V,N*64
µb
( 7.03 (45969)                  00040007  °$GPRMC,,V,,,,,,,,,N*53
$GPVTG,,,,,,,,,N*30
$GPGGA,,,,,,0,00,99.99,,,,,,*48
$GPGSA,A,1,,,,,,,,,,,,,99.99,99.99,99.99*30
$GPGSV,2,1,06,01,,,11,09,,,11,23,,,14,24,,,21*75
```

It can be observed from the output, `7.03 (45969)` is the software version and `00040007` is the hardware version.

Clean Up

When the connection is no longer needed, clear the associated object.

```
delete(gps);
clear gps;
clear s;
```

Input Arguments

gps — GPS sensor

gpsdev object

The GPS sensor, specified as a `gpsdev` object.

cmdArray — Raw command to configure GPS module

hexadecimal array

Raw command to configure the GPS module, specified as an hexadecimal array.

Example: `[0xB5 0x62 0x06 0x01 0x08 0x00 0xF0 0x08 0x00 0x01 0x00 0x00 0x00 0x00 0x08 0x60]`

Data Types: `uint8`

See Also

Objects
`gpsdev`

Functions
`flush` | `release` | `read` | `info`

Introduced in R2020b

writeOccupancyGrid

Write values from grid to ROS message

Syntax

```
writeOccupancyGrid(msg,map)
```

Description

`writeOccupancyGrid(msg,map)` writes occupancy values and other information to the ROS message, `msg`, from the occupancy grid, `map`.

Input Arguments

`msg` — `'nav_msgs/OccupancyGrid'` ROS message
`OccupancyGrid` object handle

`'nav_msgs/OccupancyGrid'` ROS message, specified as an `OccupancyGrid` ROS message object handle.

`map` — Occupancy map
`occupancyMap` object handle

Occupancy map, specified as an `occupancyMap` object handle.

See Also

Functions
`rosReadBinaryOccupancyGrid` | `rosReadOccupancyMap3D` | `rosReadOccupancyGrid` | `rosWriteOccupancyGrid`

Introduced in R2016b

Classes

accelparams

Accelerometer sensor parameters

Description

The `accelparams` class creates an accelerometer sensor parameters object. You can use this object to model an accelerometer when simulating an IMU with `imuSensor`. See the “Algorithms” on page 2-438 section of `imuSensor` for details of `accelparams` modeling.

Creation

Syntax

```
params = accelparams
params = accelparams(Name,Value)
```

Description

`params = accelparams` returns an ideal accelerometer sensor parameters object with default values.

`params = accelparams(Name,Value)` configures an accelerometer sensor parameters object properties using one or more `Name-Value` pair arguments. `Name` is a property name and `Value` is the corresponding value. `Name` must appear inside single quotes (`''`). You can specify several name-value pair arguments in any order as (`Name1,Value1,...,NameN,ValueN`). Any unspecified properties take default values.

Properties

`MeasurementRange` — Maximum sensor reading (m/s^2)
`inf` (default) | real positive scalar

Maximum sensor reading in m/s^2, specified as a real positive scalar.

Data Types: `single` | `double`

`Resolution` — Resolution of sensor measurements ($(m/s^2)/LSB$)
`0` (default) | real nonnegative scalar

Resolution of sensor measurements in $(m/s^2)/LSB$, specified as a real nonnegative scalar. Here, LSB is the acronym for least significant bit. Resolution is often referred as Scale Factor for accelerometer.

Data Types: `single` | `double`

`ConstantBias` — Constant sensor offset bias (m/s^2)
`[0 0 0]` (default) | real scalar | real 3-element row vector

Constant sensor offset bias in m/s^2, specified as a real scalar or 3-element row vector. Any scalar input is converted into a real 3-element row vector where each element has the input scalar value.

Data Types: `single` | `double`

`AxesMisalignment` — Sensor axes skew (%)
`diag([100 100 100])` (default) | scalar in the range [0,100] | 3-element row vector in the range [0,100] | 3-by-3 matrix in the range [0,100]

Sensor axes skew in percentage, specified as a scalar, a 3-element row vector, or a 3-by-3 matrix with values ranging from `0` to `100`. The diagonal elements of the matrix account for the misalignment effects for each axes. The off-diagonal elements account for the cross-axes misalignment effects. The measured state $v_{measure}$ is obtained from the true state $v_{measure}$ via the misalignment matrix as:

$$v_{measure} = \frac{1}{100} M v_{true} = \frac{1}{100} \begin{bmatrix} m_{11} & m_{12} & m_{13} \\ m_{21} & m_{22} & m_{23} \\ m_{31} & m_{32} & m_{33} \end{bmatrix} v_{true}$$

- If you specify the property as a scalar, then all the off-diagonal elements of the matrix take the value of the specified scalar and all the diagonal elements are 100.
- If you specify the property as a vector [*a b c*], then $m_{21} = m_{31} = a$, $m_{12} = m_{32} = b$, and $m_{13} = m_{23} = c$. All the diagonal elements are 100.

Data Types: `single` | `double`

`NoiseDensity` — Power spectral density of sensor noise ($m/s^2/\sqrt{Hz}$)
`[0 0 0]` (default) | real scalar | real 3-element row vector

Power spectral density of sensor noise in ($m/s^2/\sqrt{Hz}$), specified as a real scalar or 3-element row vector. This property corresponds to the velocity random walk (VRW). Any scalar input is converted into a real 3-element row vector where each element has the input scalar value.

Data Types: `single` | `double`

`BiasInstability` — Instability of the bias offset (m/s^2)
`[0 0 0]` (default) | real scalar | real 3-element row vector

Instability of the bias offset in m/s^2, specified as a real scalar or 3-element row vector. Any scalar input is converted into a real 3-element row vector where each element has the input scalar value.

Data Types: `single` | `double`

`RandomWalk` — Integrated white noise of sensor ($(m/s^2)(\sqrt{Hz})$)
`[0 0 0]` (default) | real scalar | real 3-element row vector

Integrated white noise of sensor in $(m/s^2)(\sqrt{Hz})$, specified as a real scalar or 3-element row vector. Any scalar input is converted into a real 3-element row vector where each element has the input scalar value.

Data Types: `single` | `double`

`TemperatureBias` — Sensor bias from temperature ($(m/s^2)/°C$)
`[0 0 0]` (default) | real scalar | real 3-element row vector

Sensor bias from temperature in $(m/s^2)/°C$, specified as a real scalar or 3-element row vector. Any scalar input is converted into a real 3-element row vector where each element has the input scalar value.

Data Types: `single` | `double`

`TemperatureScaleFactor` — Scale factor error from temperature (%/°C)
`[0 0 0]` (default) | real scalar in the range [0,100] | real 3-element row vector in the range [0,100]

Scale factor error from temperature in %/°C, specified as a real scalar or real 3-element row vector with values ranging from 0 to 100. Any scalar input is converted into a real 3-element row vector where each element has the input scalar value.

Data Types: `single` | `double`

Examples

Generate Accelerometer Data from Stationary Inputs

Generate accelerometer data for an imuSensor object from stationary inputs.

Generate an accelerometer parameter object with a maximum sensor reading of 19.6 m/s^2 and a resolution of 0.598 $\left(mm/s^2\right)$/LSB. The constant offset bias is 0.49 m/s^2. The sensor has a power spectral density of 3920 $\left(\mu m/s^2\right)/\sqrt{Hz}$. The bias from temperature is 0.294 $\left(m/s^2\right)/^0C$. The scale factor error from temperature is 0.02%/^{0}C. The sensor axes are skewed by 2%.

```
params = accelparams('MeasurementRange',19.6,'Resolution',0.598e-3,'ConstantBias',0.49,'NoiseDens
```

Use a sample rate of 100 Hz spaced out over 1000 samples. Create the imuSensor object using the accelerometer parameter object.

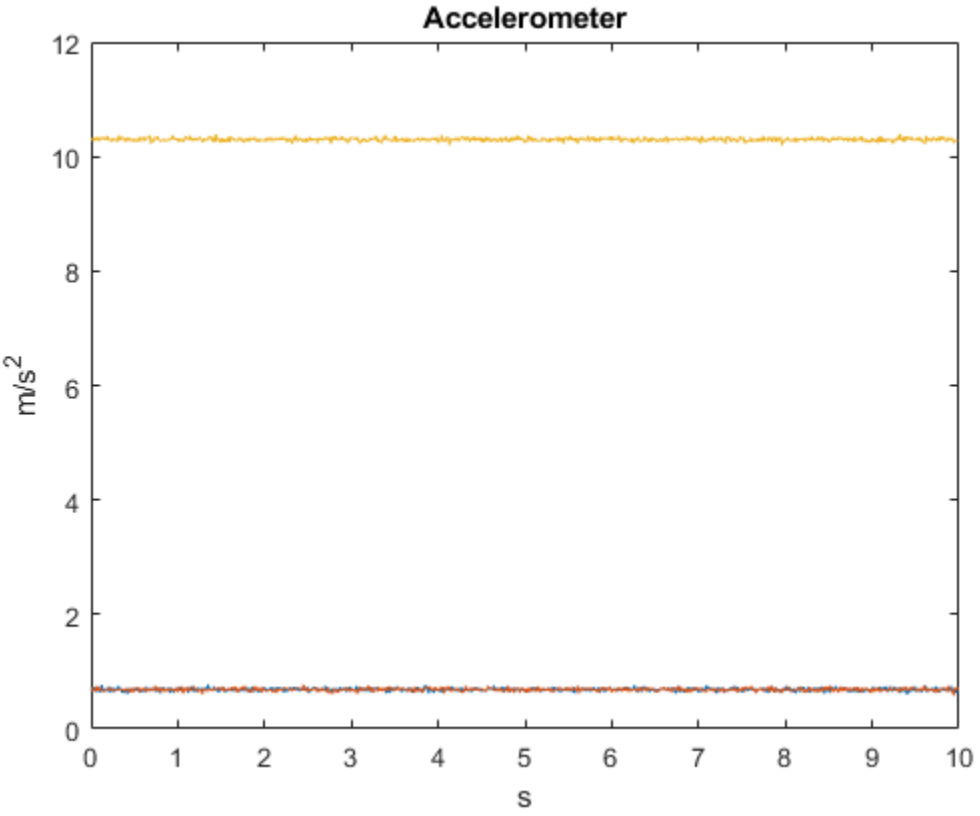
```
Fs = 100;
numSamples = 1000;
t = 0:1/Fs:(numSamples-1)/Fs;

imu = imuSensor('SampleRate', Fs, 'Accelerometer', params);
```

Generate accelerometer data from the imuSensor object.

```
orient = quaternion.ones(numSamples, 1);
acc = zeros(numSamples, 3);
angvel = zeros(numSamples, 3);

accelData = imu(acc, angvel, orient);
```

Plot the resultant accelerometer data.

```
plot(t, accelData)
title('Accelerometer')
xlabel('s')
ylabel('m/s^2')
```



Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`imuSensor` | `gyroparams` | `magparams`

Introduced in R2018b

ahrs10filter

Height and orientation from MARG and altimeter readings

Description

The `ahrs10filter` object fuses MARG and altimeter sensor data to estimate device height and orientation. MARG (magnetic, angular rate, gravity) data is typically derived from magnetometer, gyroscope, and accelerometer sensors. The filter uses an 18-element state vector to track the orientation quaternion, vertical velocity, vertical position, MARG sensor biases, and geomagnetic vector. The `ahrs10filter` object uses an extended Kalman filter to estimate these quantities.

Creation

Syntax

```
FUSE = ahrs10filter
FUSE = ahrs10filter('ReferenceFrame',RF)
FUSE = ahrs10filter(___,Name,Value)
```

Description

`FUSE = ahrs10filter` returns an extended Kalman filter object, `FUSE`, for sensor fusion of MARG and altimeter readings to estimate device height and orientation.

`FUSE = ahrs10filter('ReferenceFrame',RF)` returns an extended Kalman filter object that estimates device height and orientation relative to the reference frame `RF`. Specify RF as `'NED'` (North-East-Down) or `'ENU'` (East-North-Up). The default value is `'NED'`.

`FUSE = ahrs10filter(___,Name,Value)` sets each property `Name` to the specified `Value`. Unspecified properties have default values.

Properties

`IMUSampleRate` — Sample rate of the IMU (Hz)
`100` (default) | positive scalar

Sample rate of the IMU in Hz, specified as a positive scalar.

Data Types: `single` | `double`

`GyroscopeNoise` — Multiplicative process noise variance from gyroscope ((rad/s)²)
`[1e-9,1e-9,1e-9]` (default) | scalar | three-element row vector

Multiplicative process noise variance from the gyroscope in $(rad/s)^2$, specified as positive real finite numbers.

Data Types: `single` | `double`

`AccelerometerNoise` — Multiplicative process noise variance from accelerometer ($(m/s^2)^2$)
`[1e-4,1e-4,1e-4]` (default) | scalar | three-element row vector

Multiplicative process noise variance from the accelerometer in $(m/s^2)^2$, specified as positive real finite numbers.

Data Types: `single` | `double`

`GyroscopeBiasNoise` — Multiplicative process noise variance from gyroscope bias ($(rad/s^2)^2$)
`[1e-10,1e-10,1e-10]` (default) | scalar | three-element row vector

Multiplicative process noise variance from the gyroscope bias in $(rad/s^2)^2$, specified as positive real finite numbers.

Data Types: `single` | `double`

`AccelerometerBiasNoise` — Multiplicative process noise variance from accelerometer bias ($(m/s^2)^2$)
`[1e-4,1e-4,1e-4]` (default) | scalar | three-element row vector

Multiplicative process noise variance from the accelerometer bias in $(m/s^2)^2$, specified as positive real finite numbers.

Data Types: `single` | `double`

`GeomagneticVectorNoise` — Additive process noise for geomagnetic vector (μT^2)
`[1e-6,1e-6,1e-6]` (default) | scalar | three-element row vector

Additive process noise for geomagnetic vector in μT^2, specified as positive real finite numbers.

Data Types: `single` | `double`

`MagnetometerBiasNoise` — Additive process noise for magnetometer bias (μT^2)
`[0.1,0.1,0.1]` (default) | scalar | three-element row vector

Additive process noise for magnetometer bias in μT^2, specified as positive real finite numbers.

Data Types: `single` | `double`

`State` — State vector of extended Kalman filter
18-element column vector

State vector of the extended Kalman filter. The state values represent:

State	Units	Index
Orientation (quaternion parts)	N/A	1:4
Altitude (NED or ENU)	m	5
Vertical Velocity (NED or ENU)	m/s	6
Delta Angle Bias (XYZ)	rad/s	7:9
Delta Velocity Bias (XYZ)	m/s	10:12
Geomagnetic Field Vector (NED or ENU)	µT	13:15
Magnetometer Bias (XYZ)	µT	16:18

The default initial state corresponds to an object at rest located at `[0 0 0]` in geodetic LLA coordinates.

Data Types: `single` | `double`

`StateCovariance` — State error covariance for extended Kalman filter
`eye(18)*1e-6` (default) | 18-by-18 matrix

State error covariance for the Kalman filter, specified as an 18-by-18-element matrix of real numbers.

Data Types: `single` | `double`

Object Functions

predict	Update states using accelerometer and gyroscope data for ahrs10filter
fusemag	Correct states using magnetometer data for ahrs10filter
fusealtimeter	Correct states using altimeter data for ahrs10filter
correct	Correct states using direct state measurements for ahrs10filter
residual	Residuals and residual covariances from direct state measurements for ahrs10filter
residualmag	Residuals and residual covariance from magnetometer measurements for ahrs10filter
residualaltimeter	Residuals and residual covariance from altimeter measurements for ahrs10filter
pose	Current orientation and position estimate for ahrs10filter
reset	Reset internal states for ahrs10filter
stateinfo	Display state vector information for ahrs10filter
tune	Tune ahrs10filter parameters to reduce estimation error
copy	Create copy of ahrs10filter

Examples

Estimate Pose of UAV

Load logged sensor data, ground truth pose, and initial state and initial state covariance. Calculate the number of IMU samples per altimeter sample and the number of IMU samples per magnetometer sample.

```
load('fuse10exampledata.mat', ...
     'imuFs','accelData','gyroData', ...
     'magnetometerFs','magData', ...
     'altimeterFs','altData', ...
     'expectedHeight','expectedOrient', ...
     'initstate','initcov');

imuSamplesPerAlt = fix(imuFs/altimeterFs);
imuSamplesPerMag = fix(imuFs/magnetometerFs);
```

Create an AHRS filter that fuses MARG and altimeter readings to estimate height and orientation. Set the sampling rate and measurement noises of the sensors. The values were determined from datasheets and experimentation.

```
filt = ahrs10filter('IMUSampleRate',imuFs, ...
                    'AccelerometerNoise',0.1, ...
                    'State',initstate, ...
                    'StateCovariance',initcov);
```

```
Ralt = 0.24;
Rmag = 0.9;
```

Preallocate variables to log height and orientation.

```
numIMUSamples = size(accelData,1);
estHeight = zeros(numIMUSamples,1);
estOrient = zeros(numIMUSamples,1,'quaternion');
```

Fuse accelerometer, gyroscope, magnetometer and altimeter data. The outer loop predicts the filter forward at the fastest sample rate (the IMU sample rate).

```
for ii = 1:numIMUSamples

    % Use predict to estimate the filter state based on the accelometer and
    % gyroscope data.
    predict(filt,accelData(ii,:),gyroData(ii,:));

    % Magnetometer data is collected at a lower rate than IMU data. Fuse
    % magnetometer data at the lower rate.
    if ~mod(ii,imuSamplesPerMag)
        fusemag(filt,magData(ii,:),Rmag);
    end

    % Altimeter data is collected at a lower rate than IMU data. Fuse
    % altimeter data at the lower rate.
    if ~mod(ii, imuSamplesPerAlt)
        fusealtimeter(filt,altData(ii),Ralt);
    end

    % Log the current height and orientation estimate.
    [estHeight(ii),estOrient(ii)] = pose(filt);
end
```

Calculate the RMS errors between the known true height and orientation and the output from the AHRS filter.

```
pErr = expectedHeight - estHeight;
qErr = rad2deg(dist(expectedOrient,estOrient));

pRMS = sqrt(mean(pErr.^2));
qRMS = sqrt(mean(qErr.^2));

fprintf('Altitude RMS Error\n');
```

```
Altitude RMS Error
```

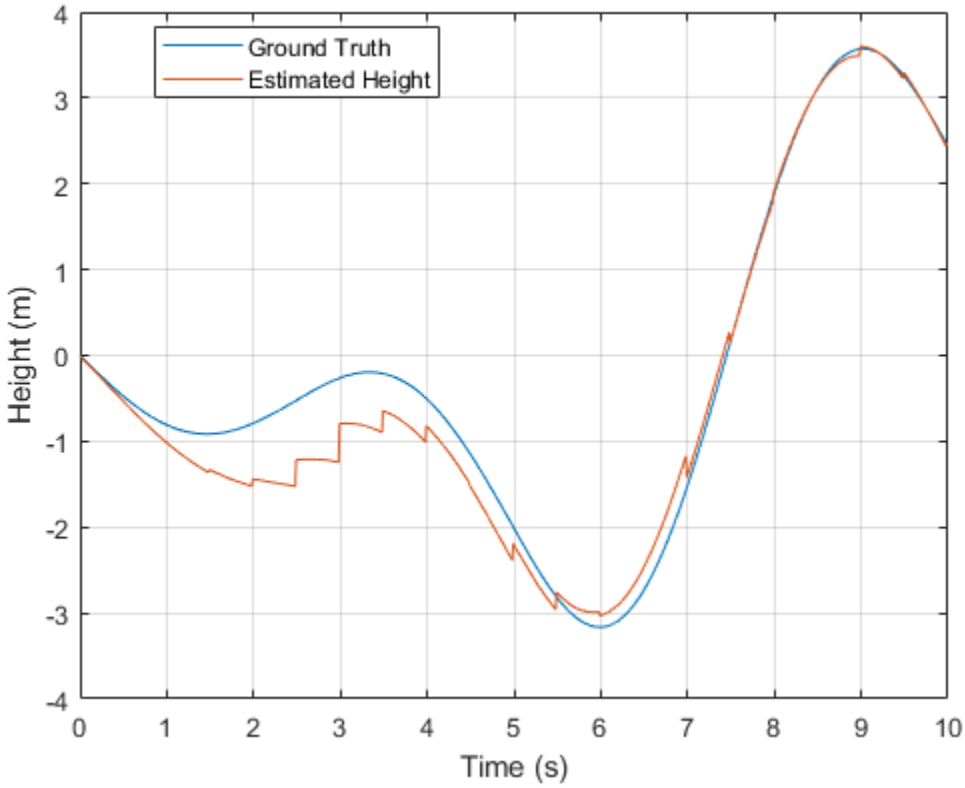
```
fprintf('\t%.2f (meters)\n\n',pRMS);
```

```
    0.38 (meters)
```

Visualize the true and estimated height over time.

```
t = (0:(numIMUSamples-1))/imuFs;
plot(t,expectedHeight);hold on
plot(t,estHeight);hold off
legend('Ground Truth','Estimated Height','location','best')
ylabel('Height (m)')
```

```
xlabel('Time (s)')
grid on
```




```
fprintf('Quaternion Distance RMS Error\n');
```

```
Quaternion Distance RMS Error
```

```
fprintf('\t%.2f (degrees)\n\n',qRMS);
```

```
    2.93 (degrees)
```

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`ahrsfilter` | `insfilter`

Introduced in R2019a

correct

Correct states using direct state measurements for `ahrs10filter`

Syntax

```
correct(FUSE,idx,measurement,measurementCovariance)
```

Description

`correct(FUSE,idx,measurement,measurementCovariance)` corrects the state and state estimation error covariance based on the measurement and measurement covariance. The measurement maps directly to the state specified by the indices `idx`.

Input Arguments

`FUSE` — `ahrs10filter` object

object

Object of `ahrs10filter`.

`idx` — State vector index of measurement to correct

N-element vector of increasing integers in the range [1,18]

State vector index of measurement to correct, specified as an *N*-element vector of increasing integers in the range [1,18].

The state values represent:

State	Units	Index
Orientation (quaternion parts)	N/A	1:4
Altitude (NED)	m	5
Vertical Velocity (NED)	m/s	6
Delta Angle Bias (XYZ)	rad/s	7:9
Delta Velocity Bias (XYZ)	m/s	10:12
Geomagnetic Field Vector (NED)	µT	13:15
Magnetometer Bias (XYZ)	µT	16:18

Data Types: `single` | `double`

`measurement` — Direct measurement of state

N-element vector

Direct measurement of state, specified as a *N*-element vector. *N* is the number of elements of the index argument, `idx`.

Data Types: `single` | `double`

`measurementCovariance` — Covariance of measurement
scalar | *N*-element vector | *N*-by-*N* matrix

Covariance of measurement, specified as a scalar, *N*-element vector, or *N*-by-*N* matrix. *N* is the number of elements of the index argument, `idx`.

Data Types: `single` | `double`

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also
`ahrs10filter` | `insfilter`

Introduced in R2019a

copy

Create copy of `ahrs10filter`

Syntax

```
newFilter = copy(filter)
```

Description

`newFilter = copy(filter)` returns a copy of the `ahrs10filter`, `filter`, with the exactly same property values.

Input Arguments

`filter` — Filter to be copied

`ahrs10filter`

Filter to be copied, specified as an `ahrs10filter` object.

Output Arguments

`newFilter` — New copied filter

`ahrs10filter`

New copied filter, returned as an `ahrs10filter` object.

See Also

`ahrs10filter`

Introduced in R2020b

fusealtimeter

Correct states using altimeter data for `ahrs10filter`

Syntax

```
[res,resCov] = fusealtimeter(FUSE,altimeterReadings,
altimeterReadingsCovariance)
```

Description

`[res,resCov] = fusealtimeter(FUSE,altimeterReadings, altimeterReadingsCovariance)` fuses altimeter data to correct the state estimate.

Input Arguments

`FUSE` — `ahrs10filter` object
object

Object of `ahrs10filter`.

`altimeterReadings` — Altimeter readings (m)
real scalar

Altimeter readings in meters, specified as a real scalar.

Data Types: `single` | `double`

`altimeterReadingsCovariance` — Altimeter readings error covariance (m^2)
real scalar

Altimeter readings error covariance in m^2, specified as a real scalar.

Data Types: `single` | `double`

Output Arguments

`res` — Measurement residual
scalar

Measurement residual, returned as a scalar in meters.

`resCov` — Residual covariance
nonnegative scalar

Residual covariance, returned as a nonnegative scalar in m^2.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

`ahrs10filter` | `insfilter`

Introduced in R2019a

fusemag

Correct states using magnetometer data for `ahrs10filter`

Syntax

```
[res,resCov] = fusemag(FUSE,magReadings,magReadingsCovariance)
```

Description

`[res,resCov] = fusemag(FUSE,magReadings,magReadingsCovariance)` fuses magnetometer data to correct the state estimate.

Input Arguments

`FUSE` — `ahrs10filter` object
object

Object of `ahrs10filter`.

`magReadings` — Magnetometer readings (µT)
3-element row vector

Magnetometer readings in µT, specified as a 3-element row vector of finite real numbers.

Data Types: `single` | `double`

`magReadingsCovariance` — Magnetometer readings error covariance (μT^2)
scalar | 3-element row vector | 3-by-3 matrix

Magnetometer readings error covariance in μT^2, specified as a scalar, 3-element row vector, or 3-by-3 matrix.

Data Types: `single` | `double`

`res` — Residual
1-by-3 vector of real values

Residual, returned as a 1-by-3 vector of real values in µT.

Data Types: `single` | `double`

`resCov` — Residual covariance
3-by-3 matrix of real values

Residual covariance, returned a 3-by-3 matrix of real values in $(\mu T)^2$.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

`ahrs10filter` | `insfilter`

Introduced in R2019a

pose

Current orientation and position estimate for `ahrs10filter`

Syntax

```
[position, orientation, velocity] = pose(FUSE)
[position, orientation, velocity] = pose(FUSE,format)
```

Description

`[position, orientation, velocity] = pose(FUSE)` returns the current estimate of the pose.

`[position, orientation, velocity] = pose(FUSE,format)` returns the current estimate of the pose with orientation in the specified orientation format.

Input Arguments

`FUSE` — `ahrs10filter` object
object

Object of `ahrs10filter`.

`format` — Output orientation format
`'quaternion'` (default) | `'rotmat'`

Output orientation format, specified as either `'quaternion'` for a `quaternion` or `'rotmat'` for a rotation matrix.

Data Types: `char` | `string`

Output Arguments

`position` — Position estimate expressed in the local coordinate system (m)
3-element row vector

Position estimate expressed in the local coordinate system of the filter in meters, returned as a 3-element row vector.

Data Types: `single` | `double`

`orientation` — Orientation estimate expressed in the local coordinate system
quaternion (default) | 3-by-3 rotation matrix

Orientation estimate expressed in the local coordinate system of the filter, returned as a scalar quaternion or 3-by-3 rotation matrix. The quaternion or rotation matrix represents a frame rotation from the local reference frame of the filter to the body reference frame.

Data Types: `single` | `double` | `quaternion`

`velocity` — Velocity estimate expressed in local coordinate system (m/s)
3-element row vector

Velocity estimate expressed in the local coordinate system of the filter in m/s, returned as a 3-element row vector.

Data Types: `single` | `double`

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`ahrs10filter` | `insfilter`

Introduced in R2019a

predict

Update states using accelerometer and gyroscope data for `ahrs10filter`

Syntax

```
predict(FUSE,accelReadings,gyroReadings)
```

Description

`predict(FUSE,accelReadings,gyroReadings)` fuses accelerometer and gyroscope data to update the state estimate.

Input Arguments

`FUSE` — `ahrs10Filter` object
object

Object of `ahrs10filter`.

`accelReadings` — Accelerometer readings in the sensor body coordinate system (m/s^2)
N-by-3 matrix

Accelerometer readings in local sensor body coordinate system in m/s^2, specified as an *N*-by-3 matrix. *N* is the number of samples, and the three columns of `accelReadings` represent the [*x y z*] measurements. Accelerometer readings are assumed to correspond to the sample rate specified by the `IMUSampleRate` property.

Data Types: `single` | `double`

`gyroReadings` — Gyroscope readings in the sensor body coordinate system (rad/s)
N-by-3 matrix

Gyroscope readings in the sensor body coordinate system in rad/s, specified as an *N*-by-3 matrix. *N* is the number of samples, and the three columns of `gyroReadings` represent the [*x y z*] measurements. Gyroscope readings are assumed to correspond to the sample rate specified by the `IMUSampleRate` property.

Data Types: `single` | `double`

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

`ahrs10filter` | `insfilter`

Introduced in R2019a

reset

Reset internal states for `ahrs10filter`

Syntax

```
reset(FUSE)
```

Description

`reset(FUSE)` resets the `State`, `StateCovariance`, and internal integrators to their default values.

Input Arguments

`FUSE` — `ahrs10filter` object

object

Object of `ahrs10filter`.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`ahrs10filter` | `insfilter`

Introduced in R2019a

residual

Residuals and residual covariances from direct state measurements for `ahrs10filter`

Syntax

```
[res, resCov]= residual(FUSE,idx,measurement,measurementCovariance)
```

Description

`[res, resCov]= residual(FUSE,idx,measurement,measurementCovariance)` computes the residual, `res`, and the residual covariance, `resCov`, based on the direct state measurement and measurement covariance. The `measurement` maps directly to the states specified by indices, `idx`.

Input Arguments

`FUSE` — `ahrs10filter`

`ahrs10filter` | object

`ahrs10filter`, specified as an object.

`idx` — State vector index of measurement to correct

N-element vector of increasing integers in the range [1,18]

State vector index of measurement to correct, specified as an *N*-element vector of increasing integers in the range [1,18].

The state values represent:

State	Units	Index
Orientation (quaternion parts)	N/A	1:4
Altitude (NED)	m	5
Vertical Velocity (NED)	m/s	6
Delta Angle Bias (XYZ)	rad/s	7:9
Delta Velocity Bias (XYZ)	m/s	10:12
Geomagnetic Field Vector (NED)	µT	13:15
Magnetometer Bias (XYZ)	µT	16:18

`measurement` — Direct measurement of state

N-element vector

Direct measurement of state, specified as a *N*-element vector. *N* is the number of elements of the index argument, `idx`.

`measurementCovariance` — Covariance of measurement

N-by-*N* matrix

Covariance of measurement, specified as an *N*-by-*N* matrix. *N* is the number of elements of the index argument, `idx`.

Output Arguments

`res` — Measurement residual
1-by-*N* vector of real values

Measurement residual, returned as a 1-by-*N* vector of real values.

`resCov` — Residual covariance
N-by-*N* matrix of real values

Residual covariance, returned as a *N*-by-*N* matrix of real values.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also
`ahrs10filter`

Introduced in R2020a

residualaltimeter

Residuals and residual covariance from altimeter measurements for `ahrs10filter`

Syntax

```
[res,resCov] = residualaltimeter(FUSE,altimeterReadings,
altimeterReadingsCovariance)
```

Description

`[res,resCov] = residualaltimeter(FUSE,altimeterReadings, altimeterReadingsCovariance)` computes the residual, `res`, and the innovation covariance, `resCov`, based on the magnetometer readings and the corresponding covariance.

Input Arguments

`FUSE` — `ahrs10filter`
`ahrs10filter` | object

`ahrs10filter`, specified as an object.

`altimeterReadings` — Altimeter readings (m)
real scalar

Altimeter readings in meters, specified as a real scalar.

Data Types: `single` | `double`

`altimeterReadingsCovariance` — Altimeter readings error covariance (m^2)
real scalar

Altimeter readings error covariance in m^2, specified as a real scalar.

Data Types: `single` | `double`

Output Arguments

`res` — Measurement residual
scalar

Measurement residual, returned as a scalar in meters.

`resCov` — Residual covariance
nonnegative scalar

Residual covariance, returned as a nonnegative scalar in m^2.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`ahrs10filter` | `insfilter`

Introduced in R2020a

residualmag

Residuals and residual covariance from magnetometer measurements for `ahrs10filter`

Syntax

```
[res,resCov] = residualmag(FUSE,magReadings,magReadingsCovariance)
```

Description

`[res,resCov] = residualmag(FUSE,magReadings,magReadingsCovariance)` computes the residual, `residual`, and the residual covariance, `resCov`, based on the magnetometer readings and the corresponding covariance.

Input Arguments

`FUSE` — `ahrs10filter`

`ahrs10filter` | object

`ahrs10filter`, specified as an object.

`magReadings` — Magnetometer readings (µT)

3-element row vector

Magnetometer readings in µT, specified as a 3-element row vector of finite real numbers.

Data Types: `single` | `double`

`magReadingsCovariance` — Magnetometer readings error covariance (μT^2)

scalar | 3-element row vector | 3-by-3 matrix

Magnetometer readings error covariance in μT^2, specified as a scalar, 3-element row vector, or 3-by-3 matrix.

Data Types: `single` | `double`

Output Arguments

`res` — Residual

1-by-3 vector of real values

Residual, returned as a 1-by-3 vector of real values in µT.

Data Types: `single` | `double`

`resCov` — Residual covariance

3-by-3 matrix of real values

Residual covariance, returned a 3-by-3 matrix of real values in $(\mu T)^2$.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`ahrs10filter`

Introduced in R2020a

stateinfo

Display state vector information for `ahrs10filter`

Syntax

```
stateinfo(FUSE)
info = stateinfo(FUSE)
```

Description

`stateinfo(FUSE)` displays the meaning of each index of the `State` property and the associated units.

`info = stateinfo(FUSE)` returns a structure with fields containing descriptions of the elements of the state vector of the filter, `FUSE`.

Examples

State information of ahrs10filter

Create an `ahrs10filter` object.

```
filter = ahrs10filter;
```

Display the state information of the created filter.

```
stateinfo(filter)
```

```
States                           Units    Index
Orientation (quaternion parts)            1:4
Altitude (NAV)                   m        5
Vertical Velocity (NAV)          m/s      6
Delta Angle Bias (XYZ)           rad      7:9
Delta Velocity Bias (XYZ)        m/s      10:12
Geomagnetic Field Vector (NAV)   µT       13:15
Magnetometer Bias (XYZ)          µT       16:18
```

Output the state information of the filter as a structure.

```
info = stateinfo(filter)
```

```
info = struct with fields:
               Orientation: [1 2 3 4]
                  Altitude: 5
          VerticalVelocity: 6
            DeltaAngleBias: [7 8 9]
         DeltaVelocityBias: [10 11 12]
    GeomagneticFieldVector: [13 14 15]
          MagnetometerBias: [16 17 18]
```

Input Arguments

`FUSE` — `ahrs10filter` object

object

Object of `ahrs10filter`.

Output Arguments

`info` — State information

structure

State information, returned as a structure with fields containing descriptions of the elements of the state vector of the filter. The values of each field are the corresponding indices of the state vector.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`ahrs10filter` | `insfilter`

Introduced in R2019a

tune

Tune `ahrs10filter` parameters to reduce estimation error

Syntax

```
tunedMeasureNoise = tune(filter,measureNoise,sensorData,groundTruth)
tunedMeasureNoise = tune( ___ ,config)
```

Description

`tunedMeasureNoise = tune(filter,measureNoise,sensorData,groundTruth)` adjusts the properties of the `ahrs10filter` filter object, `filter`, and measurement noises to reduce the root-mean-squared (RMS) state estimation error between the fused sensor data and the ground truth. The function also returns the tuned measurement noise, `tunedMeasureNoise`. The function uses the property values in the filter and the measurement noise provided in the `measureNoise` structure as the initial estimate for the optimization algorithm.

`tunedMeasureNoise = tune( ___ ,config)` specifies the tuning configuration based on a `tunerconfig` object, `config`.

Examples

Tune ahrs10filter to Optimize Pose Estimate

Load the recorded sensor data and ground truth data.

```
load('ahrs10filterTuneData.mat');
```

Create tables for the sensor data and the truth data.

```
sensorData = table(Accelerometer,Gyroscope,...
    Magnetometer,Altimeter);
groundTruth = table(Orientation, Altitude);
```

Create an `ahrs10filter` filter object.

```
filter = ahrs10filter('State', initialState, ...
    'StateCovariance', initialStateCovariance);
```

Create a tuner configuration object for the filter. Set the maximum iterations to ten and set the objective limit to 0.001.

```
cfg = tunerconfig('ahrs10filter','MaxIterations',10,...
          'ObjectiveLimit',1e-3);
```

Use the tuner noise function to obtain a set of initial sensor noises used in the filter.

```
measNoise = tunernoise('ahrs10filter')
```

```
measNoise = struct with fields:
    MagnetometerNoise: 1
```

```
    AltimeterNoise: 1
```

Tune the filter and obtain the tuned parameters.

```
tunedNoise = tune(filter, measNoise, sensorData, ...
    groundTruth, cfg);
```

```
    Iteration    Parameter                Metric
    _________    _________                ______
    1            AccelerometerNoise       0.0526
    1            GyroscopeNoise           0.0526
    1            MagnetometerNoise        0.0523
    1            AltimeterNoise           0.0515
    1            AccelerometerBiasNoise   0.0510
    1            GyroscopeBiasNoise       0.0510
    1            GeomagneticVectorNoise   0.0510
    1            MagnetometerBiasNoise    0.0508
    2            AccelerometerNoise       0.0508
    2            GyroscopeNoise           0.0508
    2            MagnetometerNoise        0.0504
    2            AltimeterNoise           0.0494
    2            AccelerometerBiasNoise   0.0490
    2            GyroscopeBiasNoise       0.0490
    2            GeomagneticVectorNoise   0.0490
    2            MagnetometerBiasNoise    0.0487
    3            AccelerometerNoise       0.0487
    3            GyroscopeNoise           0.0487
    3            MagnetometerNoise        0.0482
    3            AltimeterNoise           0.0472
    3            AccelerometerBiasNoise   0.0467
    3            GyroscopeBiasNoise       0.0467
    3            GeomagneticVectorNoise   0.0467
    3            MagnetometerBiasNoise    0.0463
    4            AccelerometerNoise       0.0463
    4            GyroscopeNoise           0.0463
    4            MagnetometerNoise        0.0456
    4            AltimeterNoise           0.0446
    4            AccelerometerBiasNoise   0.0442
    4            GyroscopeBiasNoise       0.0442
    4            GeomagneticVectorNoise   0.0442
    4            MagnetometerBiasNoise    0.0437
    5            AccelerometerNoise       0.0437
    5            GyroscopeNoise           0.0437
    5            MagnetometerNoise        0.0428
    5            AltimeterNoise           0.0417
    5            AccelerometerBiasNoise   0.0413
    5            GyroscopeBiasNoise       0.0413
    5            GeomagneticVectorNoise   0.0413
    5            MagnetometerBiasNoise    0.0408
    6            AccelerometerNoise       0.0408
    6            GyroscopeNoise           0.0408
    6            MagnetometerNoise        0.0397
    6            AltimeterNoise           0.0385
    6            AccelerometerBiasNoise   0.0381
    6            GyroscopeBiasNoise       0.0381
    6            GeomagneticVectorNoise   0.0381
    6            MagnetometerBiasNoise    0.0375
```

```
     7            AccelerometerNoise        0.0375
     7            GyroscopeNoise            0.0375
     7            MagnetometerNoise         0.0363
     7            AltimeterNoise            0.0351
     7            AccelerometerBiasNoise    0.0347
     7            GyroscopeBiasNoise        0.0347
     7            GeomagneticVectorNoise    0.0347
     7            MagnetometerBiasNoise     0.0342
     8            AccelerometerNoise        0.0342
     8            GyroscopeNoise            0.0342
     8            MagnetometerNoise         0.0331
     8            AltimeterNoise            0.0319
     8            AccelerometerBiasNoise    0.0316
     8            GyroscopeBiasNoise        0.0316
     8            GeomagneticVectorNoise    0.0316
     8            MagnetometerBiasNoise     0.0313
     9            AccelerometerNoise        0.0313
     9            GyroscopeNoise            0.0313
     9            MagnetometerNoise         0.0313
     9            AltimeterNoise            0.0301
     9            AccelerometerBiasNoise    0.0298
     9            GyroscopeBiasNoise        0.0298
     9            GeomagneticVectorNoise    0.0298
     9            MagnetometerBiasNoise     0.0296
    10            AccelerometerNoise        0.0296
    10            GyroscopeNoise            0.0296
    10            MagnetometerNoise         0.0296
    10            AltimeterNoise            0.0285
    10            AccelerometerBiasNoise    0.0283
    10            GyroscopeBiasNoise        0.0283
    10            GeomagneticVectorNoise    0.0283
    10            MagnetometerBiasNoise     0.0282
```

Fuse the sensor data using the tuned filter.

```
N = size(sensorData,1);
qEstTuned = quaternion.zeros(N,1);
altEstTuned = zeros(N,1);
for ii=1:N
    predict(filter,Accelerometer(ii,:),Gyroscope(ii,:));
    if all(~isnan(Magnetometer(ii,1)))
        fusemag(filter, Magnetometer(ii,:),tunedNoise.MagnetometerNoise);
    end
    if ~isnan(Altimeter(ii))
        fusealtimeter(filter, Altimeter(ii),tunedNoise.AltimeterNoise);
    end
    [altEstTuned(ii), qEstTuned(ii)] = pose(filter);
end
```

Compute the RMS errors.

```
orientationErrorTuned = rad2deg(dist(qEstTuned, Orientation));
rmsOrientationErrorTuned = sqrt(mean(orientationErrorTuned.^2))
```

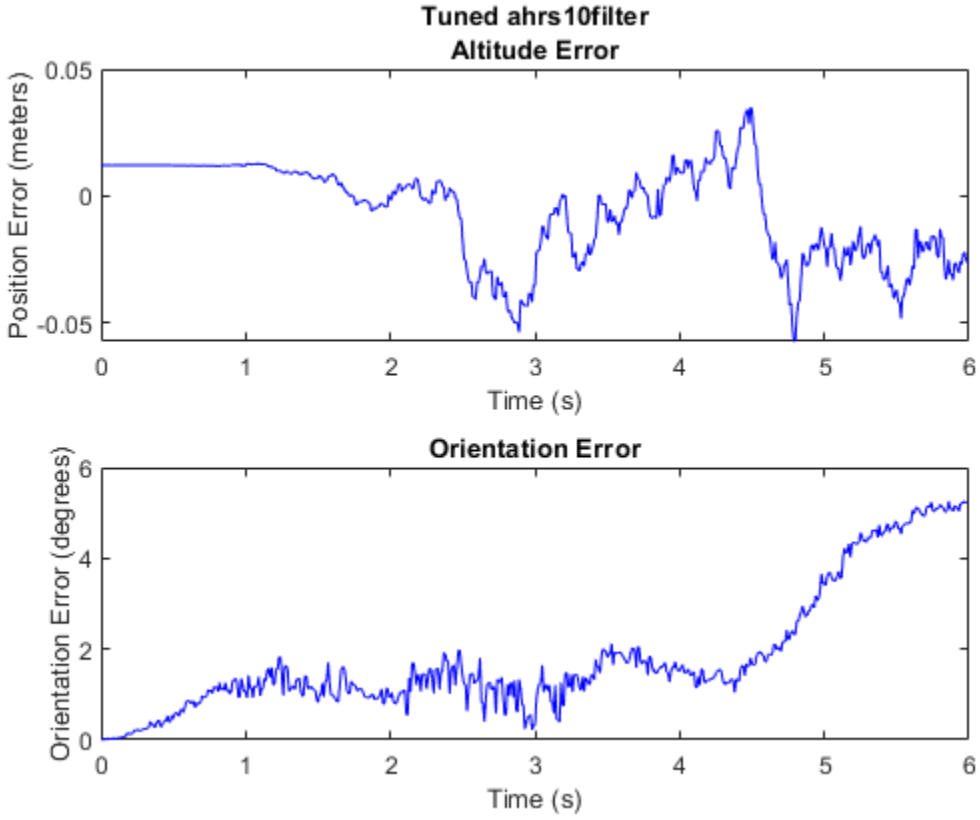
```
rmsOrientationErrorTuned = 2.2899
```

```
positionErrorTuned = altEstTuned - Altitude;
rmsPositionErrorTuned = sqrt(mean( positionErrorTuned.^2))
```

```
rmsPositionErrorTuned = 0.0199
```

Visualize the results.

```
figure;
t = (0:N-1)./ filter.IMUSampleRate;
subplot(2,1,1)
plot(t, positionErrorTuned, 'b');
title("Tuned ahrs10filter" + newline + ...
    "Altitude Error")
xlabel('Time (s)');
ylabel('Position Error (meters)')
subplot(2,1,2)
plot(t, orientationErrorTuned, 'b');
title("Orientation Error")
xlabel('Time (s)');
ylabel('Orientation Error (degrees)');
```



Input Arguments

`filter` — Filter object

`ahrs10filter` object

Filter object, specified as an `ahrs10filter` object.

`measureNoise` — Measurement noise
structure

Measurement noise, specified as a structure. The function uses the measurement noise input as the initial guess for tuning the measurement noise. The structure must contain these fields:

Field name	Description
`MagnetometerNoise`	Variance of magnetometer noise, specified as a scalar in $(\mu T)^2$
`AltimeterNoise`	Variance of altimeter noise, specified as a scalar in m^2

`sensorData` — Sensor data
`table`

Sensor data, specified as a `table`. In each row, the sensor data is specified as:

- `Accelerometer` — Accelerometer data, specified as a 1-by-3 vector of scalars in m^2/s.
- `Gyroscope`— Gyroscope data, specified as a 1-by-3 vector of scalars in rad/s.
- `Magnetometer` — Magnetometer data, specified as a 1-by-3 vector of scalars in µT.
- `Altimeter` — Altimeter data, specified as a scalar in meters.

If the magnetometer does not produce measurements, specify the corresponding entry as `NaN`. If you set the `Cost` property of the tuner configuration input, `config`, to `Custom`, then you can use other data types for the `sensorData` input based on your choice.

`groundTruth` — Ground truth data
`table`

Ground truth data, specified as a `table`. In each row, the table can optionally contain any of these variables:

- `Orientation` — Orientation from the navigation frame to the body frame, specified as a `quaternion` or a 3-by-3 rotation matrix.
- `Altitude` — Altitude, specified as a scalar in meters.
- `VertialVelocity` — Velocity in the vertical direction, specified as a scalar in m/s.
- `DeltaAngleBias` — Delta angle bias, specified as a 1-by-3 vector of scalars in radians.
- `DeltaVelocityBias` — Delta velocity bias, specified as a 1-by-3 vector of scalars in m/s.
- `GeomagneticFieldVector` — Geomagnetic field vector in navigation frame, specified as a 1-by-3 vector of scalars.
- `MagnetometerBias` — Magnetometer bias in body frame, specified as a 1-by-3 vector of scalars in µT.

The function processes each row of the `sensorData` and `groundTruth` tables sequentially to calculate the state estimate and RMS error from the ground truth. State variables not present in `groundTruth` input are ignored for the comparison. The `sensorData` and the `groundTruth` tables must have the same number of rows.

If you set the `Cost` property of the tuner configuration input, `config`, to `Custom`, then you can use other data types for the `groundTruth` input based on your choice.

`config` — Tuner configuration
`tunerconfig` object

Tuner configuration, specified as a `tunerconfig` object.

Output Arguments

`tunedMeasureNoise` — Tuned measurement noise
structure

Tuned measurement noise, returned as a structure. The structure contains these fields.

Field name	Description
`MagnetometerNoise`	Variance of magnetometer noise, specified as a scalar in $(\mu T)^2$
`AltimeterNoise`	Variance of altimeter noise, specified as a scalar in m^2

References

[1] Abbeel, P., Coates, A., Montemerlo, M., Ng, A.Y. and Thrun, S. Discriminative Training of Kalman Filters. In *Robotics: Science and systems*, Vol. 2, pp. 1, 2005.

See Also
`tunerconfig` | `tunernoise`

Introduced in R2021a

ahrsfilter

Orientation from accelerometer, gyroscope, and magnetometer readings

Description

The `ahrsfilter` System object™ fuses accelerometer, magnetometer, and gyroscope sensor data to estimate device orientation.

To estimate device orientation:

1. Create the `ahrsfilter` object and set its properties.
2. Call the object with arguments, as if it were a function.

To learn more about how System objects work, see What Are System Objects?

Creation

Syntax

```
FUSE = ahrsfilter
FUSE = ahrsfilter('ReferenceFrame',RF)
FUSE = ahrsfilter( ___ ,Name,Value)
```

Description

`FUSE = ahrsfilter` returns an indirect Kalman filter System object, `FUSE`, for sensor fusion of accelerometer, gyroscope, and magnetometer data to estimate device orientation and angular velocity. The filter uses a 12-element state vector to track the estimation error for the orientation, the gyroscope bias, the linear acceleration, and the magnetic disturbance.

`FUSE = ahrsfilter('ReferenceFrame',RF)` returns an ahrsfilter System object that fuses accelerometer, gyroscope, and magnetometer data to estimate device orientation relative to the reference frame `RF`. Specify `RF` as `'NED'` (North-East-Down) or `'ENU'` (East-North-Up). The default value is 'NED'.

`FUSE = ahrsfilter( ___ ,Name,Value)` sets each property `Name` to the specified `Value`. Unspecified properties have default values.

Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see System Design in MATLAB Using System Objects.

`SampleRate` — Input sample rate of sensor data (Hz)
`100` (default) | positive scalar

Input sample rate of the sensor data in Hz, specified as a positive scalar.

Tunable: No

Data Types: `single` | `double`

`DecimationFactor` — Decimation factor
`1` (default) | positive integer

Decimation factor by which to reduce the input sensor data rate as part of the fusion algorithm, specified as a positive integer.

The number of rows of the inputs –– `accelReadings`, `gyroReadings`, and `magReadings` –– must be a multiple of the decimation factor.

Data Types: `single` | `double`

`AccelerometerNoise` — Variance of accelerometer signal noise ((m/s²)²)
`0.00019247` (default) | positive real scalar

Variance of accelerometer signal noise in $(m/s^2)^2$, specified as a positive real scalar.

Tunable: Yes

Data Types: `single` | `double`

`MagnetometerNoise` — Variance of magnetometer signal noise (μT²)
`0.1` (default) | positive real scalar

Variance of magnetometer signal noise in μT^2, specified as a positive real scalar.

Tunable: Yes

Data Types: `single` | `double`

`GyroscopeNoise` — Variance of gyroscope signal noise ((rad/s)²)
`9.1385e-5` (default) | positive real scalar

Variance of gyroscope signal noise in $(rad/s)^2$, specified as a positive real scalar.

Tunable: Yes

Data Types: `single` | `double`

`GyroscopeDriftNoise` — Variance of gyroscope offset drift ((rad/s)²)
`3.0462e-13` (default) | positive real scalar

Variance of gyroscope offset drift in $(rad/s)^2$, specified as a positive real scalar.

Tunable: Yes

Data Types: `single` | `double`

`LinearAccelerationNoise` — Variance of linear acceleration noise (m/s²)²
`0.0096236` (default) | positive real scalar

Variance of linear acceleration noise in $(m/s^2)^2$, specified as a positive real scalar. Linear acceleration is modeled as a lowpass-filtered white noise process.

Tunable: Yes

Data Types: `single` | `double`

`LinearAccelerationDecayFactor` — Decay factor for linear acceleration drift

`0.5` (default) | scalar in the range [0,1)

Decay factor for linear acceleration drift, specified as a scalar in the range [0,1). If linear acceleration is changing quickly, set `LinearAcclerationDecayFactor` to a lower value. If linear acceleration changes slowly, set `LinearAcclerationDecayFactor` to a higher value. Linear acceleration drift is modeled as a lowpass-filtered white noise process.

Tunable: Yes

Data Types: `single` | `double`

`MagneticDisturbanceNoise` — Variance of magnetic disturbance noise (μT^2)

`0.5` (default) | real finite positive scalar

Variance of magnetic disturbance noise in μT^2, specified as a real finite positive scalar.

Tunable: Yes

Data Types: `single` | `double`

`MagneticDisturbanceDecayFactor` — Decay factor for magnetic disturbance

`0.5` (default) | positive scalar in the range [0,1]

Decay factor for magnetic disturbance, specified as a positive scalar in the range [0,1]. Magnetic disturbance is modeled as a first order Markov process.

Tunable: Yes

Data Types: `single` | `double`

`InitialProcessNoise` — Covariance matrix for process noise

12-by-12 matrix

Covariance matrix for process noise, specified as a 12-by-12 matrix. The default is:

```
  Columns 1 through 6

   0.000006092348396                   0                   0                   0
                   0   0.000006092348396                   0                   0
                   0                   0   0.000006092348396                   0
                   0                   0                   0   0.000076154354947
                   0                   0                   0                   0   0.00007615435
                   0                   0                   0                   0
                   0                   0                   0                   0
                   0                   0                   0                   0
                   0                   0                   0                   0
                   0                   0                   0                   0
                   0                   0                   0                   0
                   0                   0                   0                   0
```

```
  Columns 7 through 12

                  0                   0                   0                   0
                  0                   0                   0                   0
                  0                   0                   0                   0
                  0                   0                   0                   0
                  0                   0                   0                   0
                  0                   0                   0                   0
   0.009623610000000                  0                   0                   0
                  0   0.009623610000000                   0                   0
                  0                   0   0.009623610000000                   0
                  0                   0                   0   0.600000000000000
                  0                   0                   0                   0   0.600000000000000
                  0                   0                   0                   0
```

The initial process covariance matrix accounts for the error in the process model.

Data Types: `single` | `double`

`ExpectedMagneticFieldStrength` — Expected estimate of magnetic field strength (µT)
`50` (default) | real positive scalar

Expected estimate of magnetic field strength in µT, specified as a real positive scalar. The expected magnetic field strength is an estimate of the magnetic field strength of the Earth at the current location.

Tunable: Yes

Data Types: `single` | `double`

`OrientationFormat` — Output orientation format
`'quaternion'` (default) | `'Rotation matrix'`

Output orientation format, specified as `'quaternion'` or `'Rotation matrix'`. The size of the output depends on the input size, *N*, and the output orientation format:

- `'quaternion'` –– Output is an *N*-by-1 `quaternion`.
- `'Rotation matrix'` –– Output is a 3-by-3-by-*N* rotation matrix.

Data Types: `char` | `string`

Usage

Syntax

```
[orientation,angularVelocity] = FUSE(accelReadings,gyroReadings,magReadings)
```

Description

`[orientation,angularVelocity] = FUSE(accelReadings,gyroReadings,magReadings)` fuses accelerometer, gyroscope, and magnetometer data to compute orientation and angular velocity measurements. The algorithm assumes that the device is stationary before the first call.

Input Arguments

`accelReadings` — Accelerometer readings in sensor body coordinate system (m/s²)

N-by-3 matrix

Accelerometer readings in the sensor body coordinate system in m/s^2, specified as an *N*-by-3 matrix. *N* is the number of samples, and the three columns of `accelReadings` represent the [*x y z*] measurements. Accelerometer readings are assumed to correspond to the sample rate specified by the SampleRate property.

Data Types: `single` | `double`

`gyroReadings` — Gyroscope readings in sensor body coordinate system (rad/s)

N-by-3 matrix

Gyroscope readings in the sensor body coordinate system in rad/s, specified as an *N*-by-3 matrix. *N* is the number of samples, and the three columns of `gyroReadings` represent the [*x y z*] measurements. Gyroscope readings are assumed to correspond to the sample rate specified by the SampleRate property.

Data Types: `single` | `double`

`magReadings` — Magnetometer readings in sensor body coordinate system (µT)

N-by-3 matrix

Magnetometer readings in the sensor body coordinate system in µT, specified as an *N*-by-3 matrix. *N* is the number of samples, and the three columns of `magReadings` represent the [*x y z*] measurements. Magnetometer readings are assumed to correspond to the sample rate specified by the SampleRate property.

Data Types: `single` | `double`

Output Arguments

`orientation` — Orientation that rotates quantities from local navigation coordinate system to sensor body coordinate system

M-by-1 array of quaternions (default) | 3-by-3-by-*M* array

Orientation that can rotate quantities from the local navigation coordinate system to a body coordinate system, returned as quaternions or an array. The size and type of `orientation` depends on whether the OrienationFormat property is set to `'quaternion'` or `'Rotation matrix'`:

- `'quaternion'` –– the output is an *M*-by-1 vector of quaternions, with the same underlying data type as the inputs
- `'Rotation matrix'` –– the output is a 3-by-3-by-*M* array of rotation matrices the same data type as the inputs

The number of input samples, *N*, and the DecimationFactor property determine *M*.

You can use `orientation` in a `rotateframe` function to rotate quantities from a local navigation system to a sensor body coordinate system.

Data Types: `quaternion` | `single` | `double`

`angularVelocity` — Angular velocity in sensor body coordinate system (rad/s)

M-by-3 array (default)

Angular velocity with gyroscope bias removed in the sensor body coordinate system in rad/s, returned as an *M*-by-3 array. The number of input samples, *N*, and the `DecimationFactor` property determine *M*.

Data Types: `single` | `double`

Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

Specific to `ahrsfilter`

tune Tune ahrsfilter parameters to reduce estimation error

Common to All System Objects

step Run System object algorithm
release Release resources and allow changes to System object property values and input characteristics
reset Reset internal states of System object

Examples

Estimate Orientation Using `ahrsfilter`

Load the `rpy_9axis` file, which contains recorded accelerometer, gyroscope, and magnetometer sensor data from a device oscillating in pitch (around *y*-axis), then yaw (around *z*-axis), and then roll (around *x*-axis). The file also contains the sample rate of the recording.

```
load 'rpy_9axis' sensorData Fs
accelerometerReadings = sensorData.Acceleration;
gyroscopeReadings = sensorData.AngularVelocity;
magnetometerReadings = sensorData.MagneticField;
```

Create an `ahrsfilter` System object™ with `SampleRate` set to the sample rate of the sensor data. Specify a decimation factor of two to reduce the computational cost of the algorithm.

```
decim = 2;
fuse = ahrsfilter('SampleRate',Fs,'DecimationFactor',decim);
```

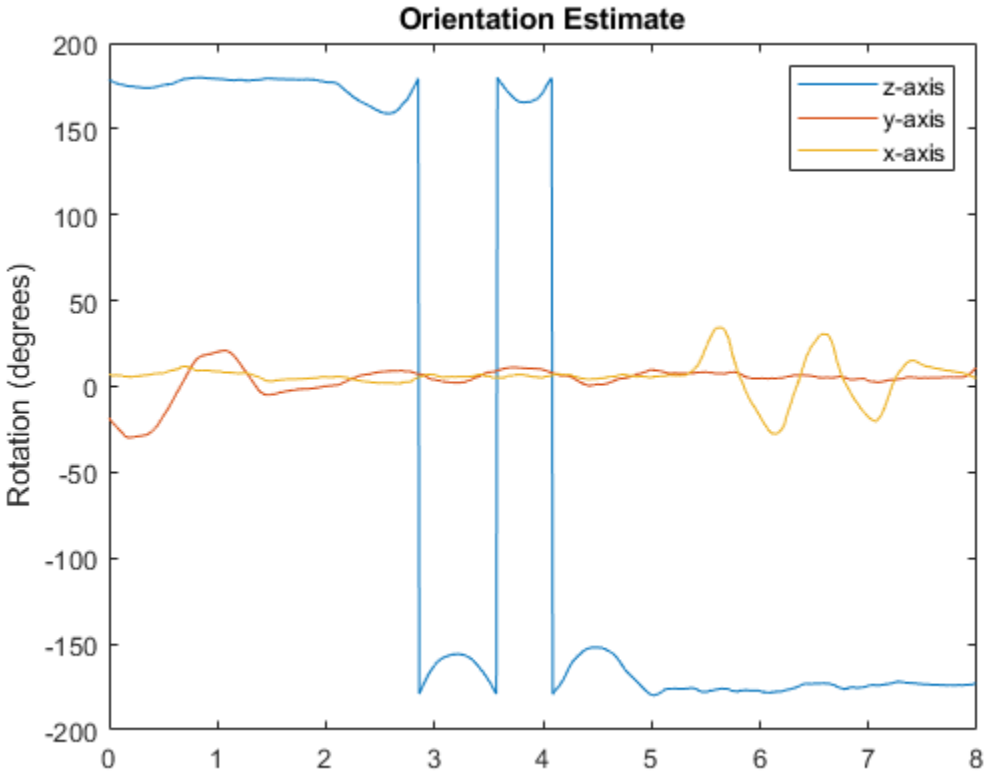
Pass the accelerometer readings, gyroscope readings, and magnetometer readings to the `ahrsfilter` object, `fuse`, to output an estimate of the sensor body orientation over time. By default, the orientation is output as a vector of quaternions.

```
q = fuse(accelerometerReadings,gyroscopeReadings,magnetometerReadings);
```

Orientation is defined by angular displacement required to rotate a parent coordinate system to a child coordinate system. Plot the orientation in Euler angles in degrees over time.

`ahrsfilter` correctly estimates the change in orientation over time, including the south-facing initial orientation.

```
time = (0:decim:size(accelerometerReadings,1)-1)/Fs;

plot(time,eulerd(q,'ZYX','frame'))
title('Orientation Estimate')
legend('z-axis', 'y-axis', 'x-axis')
ylabel('Rotation (degrees)')
```



Simulate Magnetic Jamming on `ahrsFilter`

This example shows how performance of the `ahrsfilter` System object™ is affected by magnetic jamming.

Load `StationaryIMUReadings`, which contains accelerometer, magnetometer, and gyroscope readings from a stationary IMU.

```
load 'StationaryIMUReadings.mat' accelReadings magReadings gyroReadings SampleRate

numSamples = size(accelReadings,1);
```

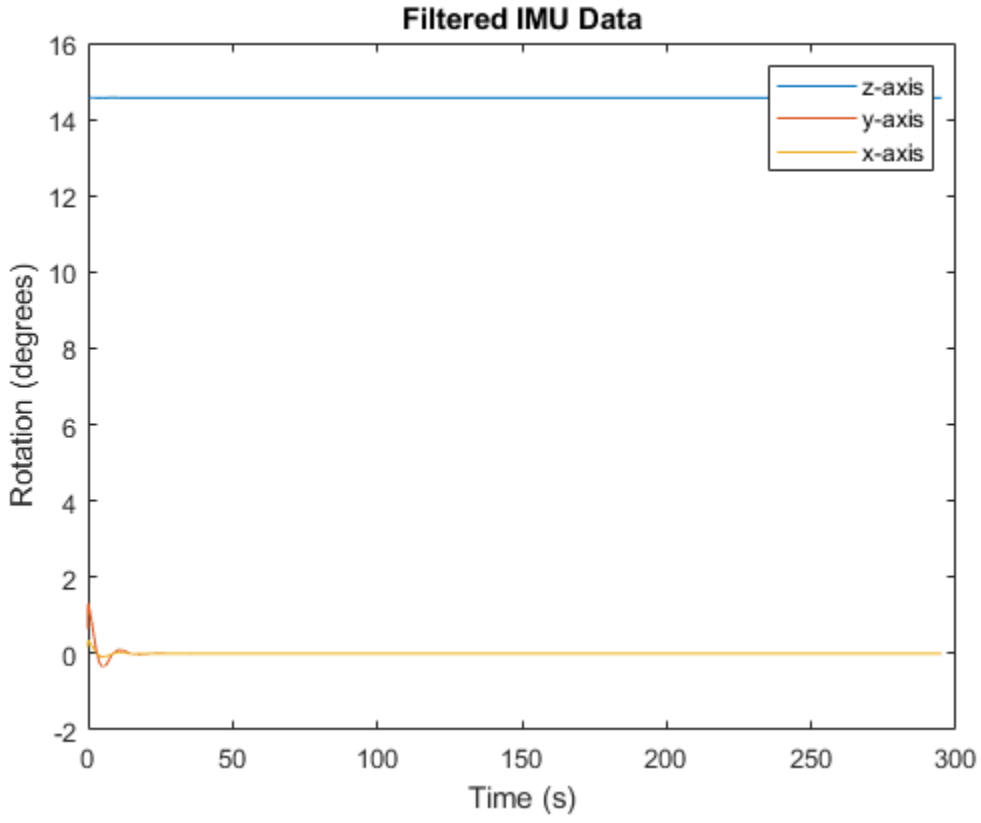
The `ahrsfilter` uses magnetic field strength to stabilize its orientation against the assumed constant magnetic field of the Earth. However, there are many natural and man-made objects which output magnetic fields and can confuse the algorithm. To account for the presence of transient magnetic fields, you can set the `MagneticDisturbanceNoise` property on the `ahrsfilter` object.

Create an `ahrsfilter` object with the decimation factor set to 2 and note the default expected magnetic field strength.

```
decim = 2;
FUSE = ahrsfilter('SampleRate',SampleRate,'DecimationFactor',decim);
```

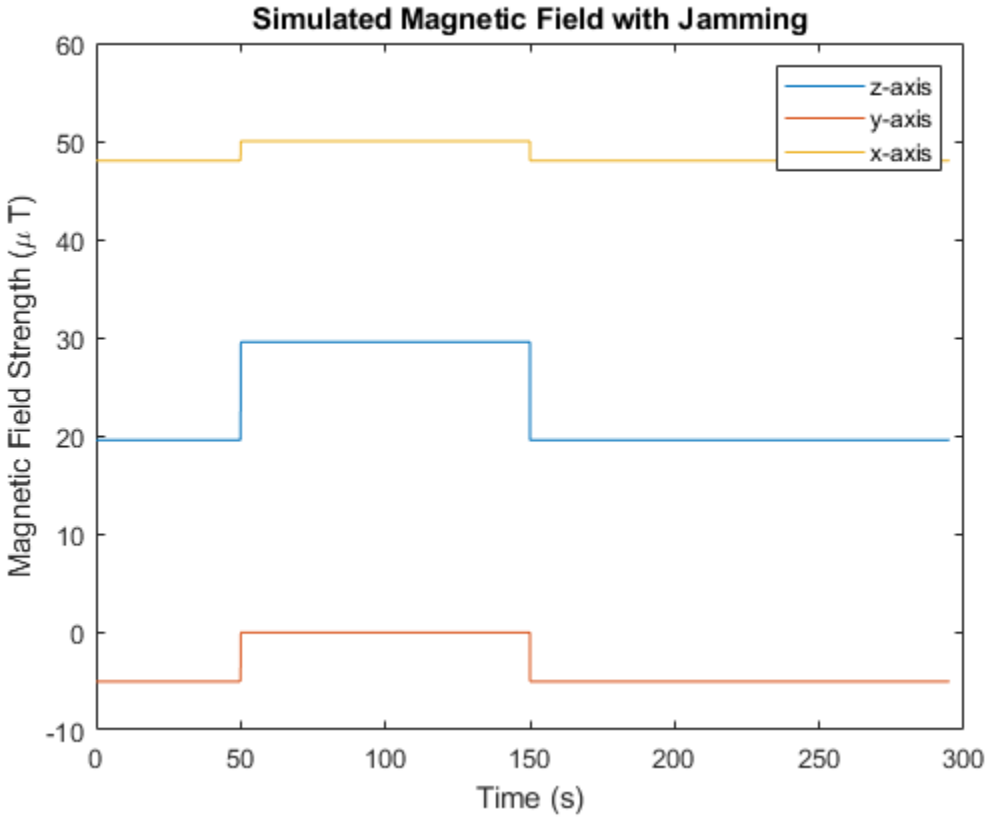
Fuse the IMU readings using the attitude and heading reference system (AHRS) filter, and then visualize the orientation of the sensor body over time. The orientation fluctuates at the beginning and stabilizes after approximately 60 seconds.

```
orientation = FUSE(accelReadings,gyroReadings,magReadings);

orientationEulerAngles = eulerd(orientation,'ZYX','frame');
time = (0:decim:(numSamples-1))'/SampleRate;

figure(1)
plot(time,orientationEulerAngles(:,1), ...
     time,orientationEulerAngles(:,2), ...
     time,orientationEulerAngles(:,3))
xlabel('Time (s)')
ylabel('Rotation (degrees)')
legend('z-axis','y-axis','x-axis')
title('Filtered IMU Data')
```



Mimic magnetic jamming by adding a transient, strong magnetic field to the magnetic field recorded in the `magReadings`. Visualize the magnetic field jamming.

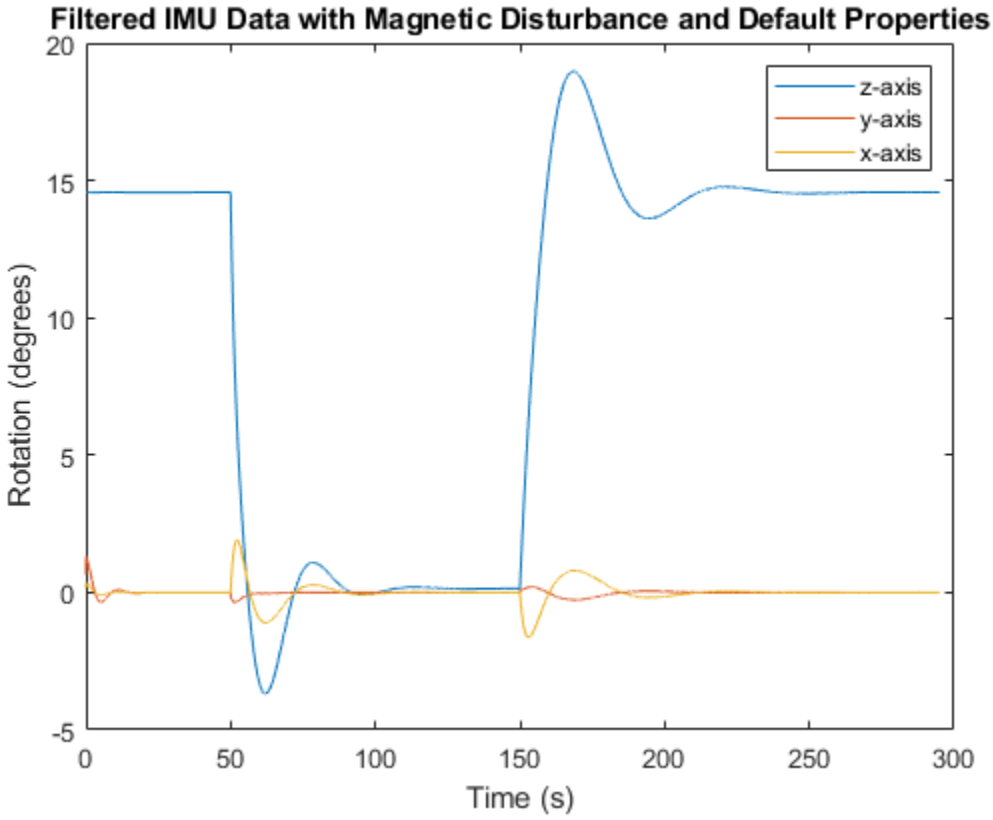
```
jamStrength = [10,5,2];
startStop = (50*SampleRate):(150*SampleRate);
jam = zeros(size(magReadings));
jam(startStop,:) = jamStrength.*ones(numel(startStop),3);

magReadings = magReadings + jam;

figure(2)
plot(time,magReadings(1:decim:end,:))
xlabel('Time (s)')
ylabel('Magnetic Field Strength (\mu T)')
title('Simulated Magnetic Field with Jamming')
legend('z-axis','y-axis','x-axis')
```



Run the simulation again using the `magReadings` with magnetic jamming. Plot the results and note the decreased performance in orientation estimation.

```
reset(FUSE)
orientation = FUSE(accelReadings,gyroReadings,magReadings);

orientationEulerAngles = eulerd(orientation,'ZYX','frame');

figure(3)
plot(time,orientationEulerAngles(:,1), ...
     time,orientationEulerAngles(:,2), ...
     time,orientationEulerAngles(:,3))
xlabel('Time (s)')
ylabel('Rotation (degrees)')
```

```
legend('z-axis','y-axis','x-axis')
title('Filtered IMU Data with Magnetic Disturbance and Default Properties')
```



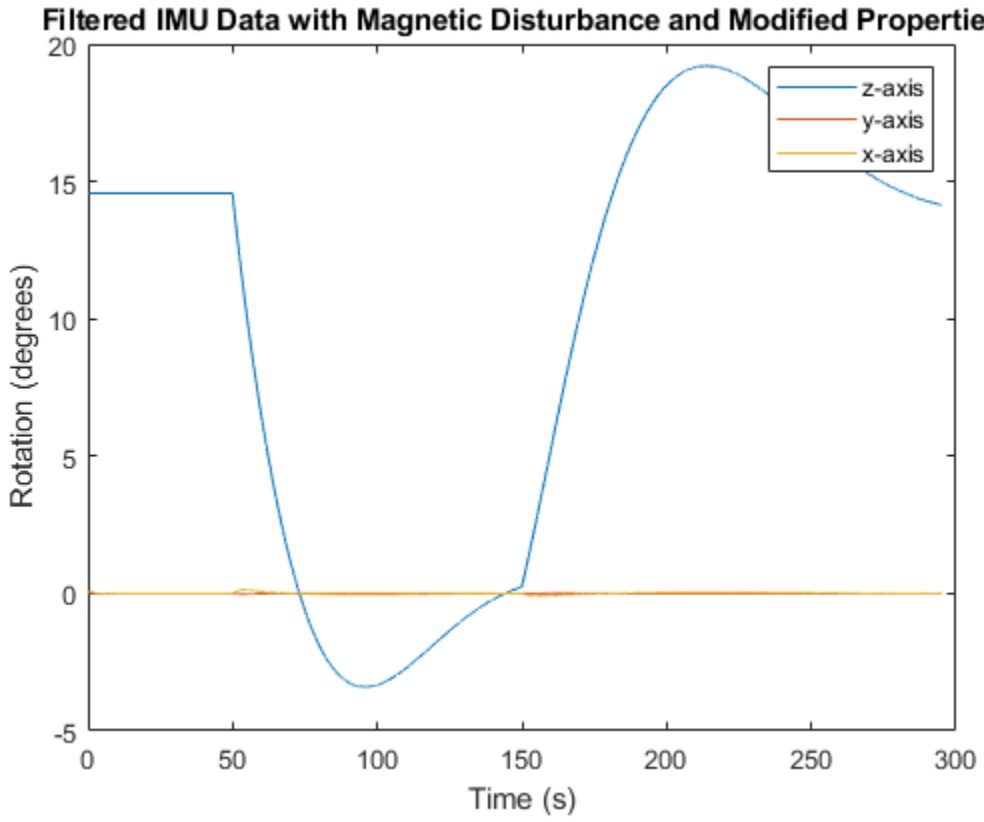
The magnetic jamming was misinterpreted by the AHRS filter, and the sensor body orientation was incorrectly estimated. You can compensate for jamming by increasing the `MagneticDisturbanceNoise` property. Increasing the `MagneticDisturbanceNoise` property increases the assumed noise range for magnetic disturbance, and the entire magnetometer signal is weighted less in the underlying fusion algorithm of `ahrsfilter`.

Set the `MagneticDisturbanceNoise` to `200` and run the simulation again.

The orientation estimation output from `ahrsfilter` is more accurate and less affected by the magnetic transient. However, because the magnetometer signal is weighted less in the underlying fusion algorithm, the algorithm may take more time to restabilize.

```
reset(FUSE)
FUSE.MagneticDisturbanceNoise = 20;

orientation = FUSE(accelReadings,gyroReadings,magReadings);

orientationEulerAngles = eulerd(orientation,'ZYX','frame');

figure(4)
plot(time,orientationEulerAngles(:,1), ...
     time,orientationEulerAngles(:,2), ...
     time,orientationEulerAngles(:,3))
xlabel('Time (s)')
```

```
ylabel('Rotation (degrees)')
legend('z-axis','y-axis','x-axis')
title('Filtered IMU Data with Magnetic Disturbance and Modified Properties')
```



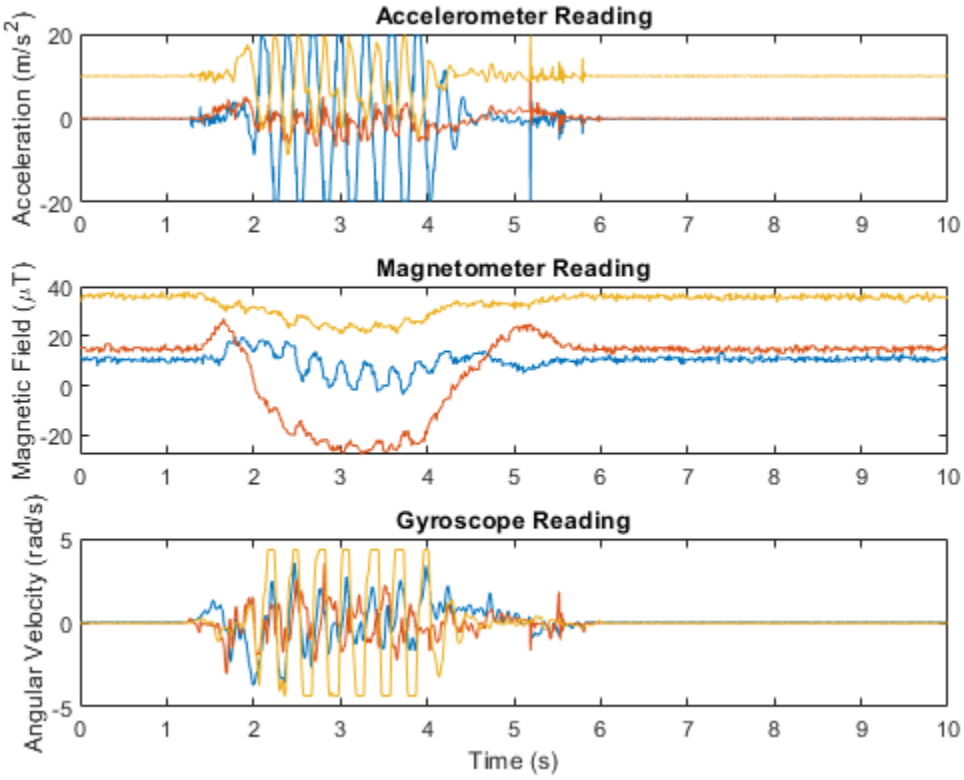
Track Shaking 9-Axis IMU

This example uses the `ahrsfilter` System object™ to fuse 9-axis IMU data from a sensor body that is shaken. Plot the quaternion distance between the object and its final resting position to visualize performance and how quickly the filter converges to the correct resting position. Then tune parameters of the `ahrsfilter` so that the filter converges more quickly to the ground-truth resting position.

Load `IMUReadingsShaken` into your current workspace. This data was recorded from an IMU that was shaken then laid in a resting position. Visualize the acceleration, magnetic field, and angular velocity as recorded by the sensors.

```
load 'IMUReadingsShaken' accelReadings gyroReadings magReadings SampleRate
numSamples = size(accelReadings,1);
time = (0:(numSamples-1))'/SampleRate;

figure(1)
subplot(3,1,1)
plot(time,accelReadings)
title('Accelerometer Reading')
ylabel('Acceleration (m/s^2)')
```

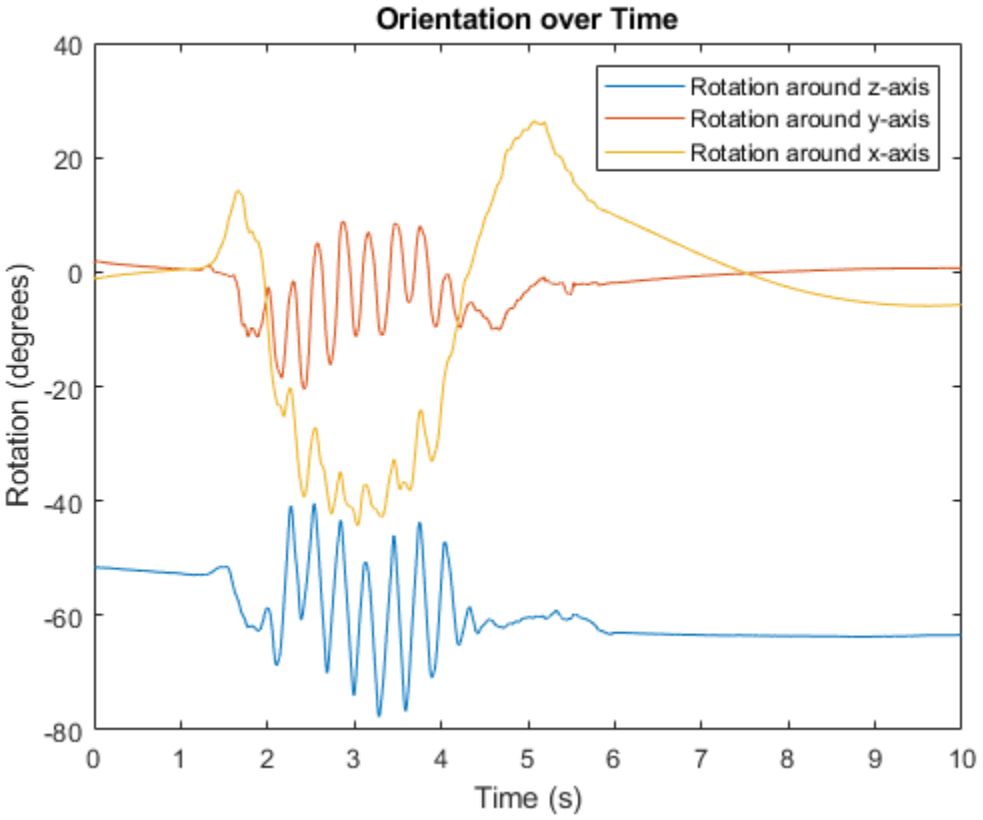
```
subplot(3,1,2)
plot(time,magReadings)
title('Magnetometer Reading')
ylabel('Magnetic Field (\muT)')

subplot(3,1,3)
plot(time,gyroReadings)
title('Gyroscope Reading')
ylabel('Angular Velocity (rad/s)')
xlabel('Time (s)')
```



Create an `ahrsfilter` and then fuse the IMU data to determine orientation. The orientation is returned as a vector of quaternions; convert the quaternions to Euler angles in degrees. Visualize the orientation of the sensor body over time by plotting the Euler angles required, at each time step, to rotate the global coordinate system to the sensor body coordinate system.

```
fuse = ahrsfilter('SampleRate',SampleRate);
orientation = fuse(accelReadings,gyroReadings,magReadings);

orientationEulerAngles = eulerd(orientation,'ZYX','frame');

figure(2)
plot(time,orientationEulerAngles(:,1), ...
     time,orientationEulerAngles(:,2), ...
     time,orientationEulerAngles(:,3))
xlabel('Time (s)')
```

```
ylabel('Rotation (degrees)')
title('Orientation over Time')
legend('Rotation around z-axis', ...
       'Rotation around y-axis', ...
       'Rotation around x-axis')
```

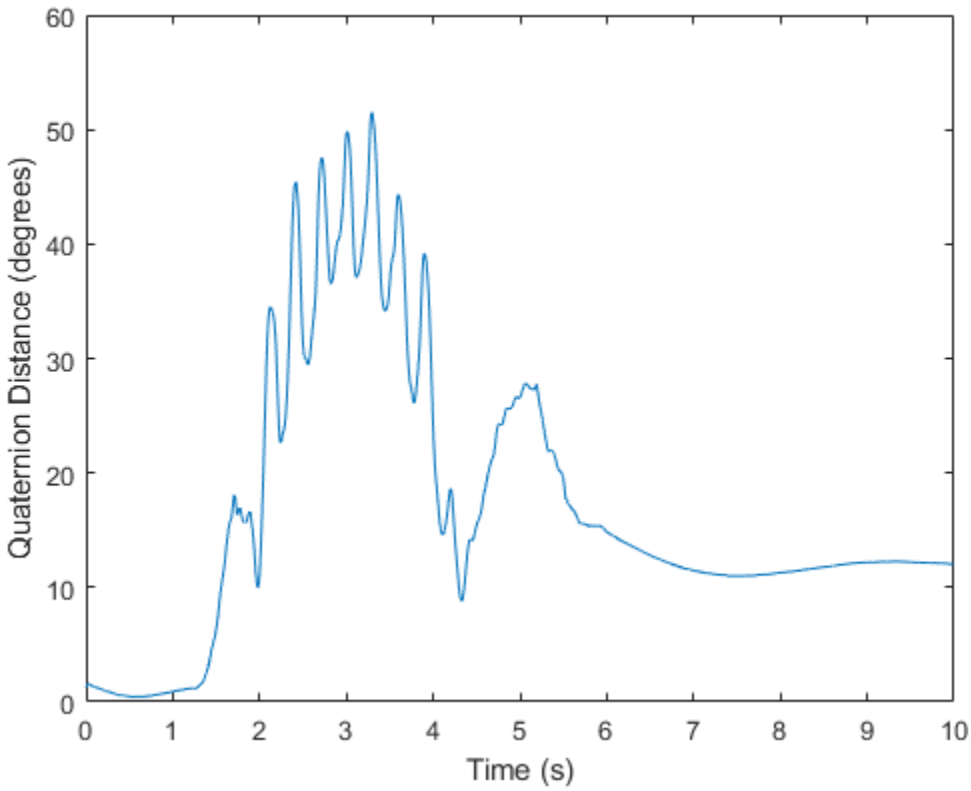


In the IMU recording, the shaking stops after approximately six seconds. Determine the resting orientation so that you can characterize how fast the `ahrsfilter` converges.

To determine the resting orientation, calculate the averages of the magnetic field and acceleration for the final four seconds and then use the `ecompass` function to fuse the data.

Visualize the quaternion distance from the resting position over time.

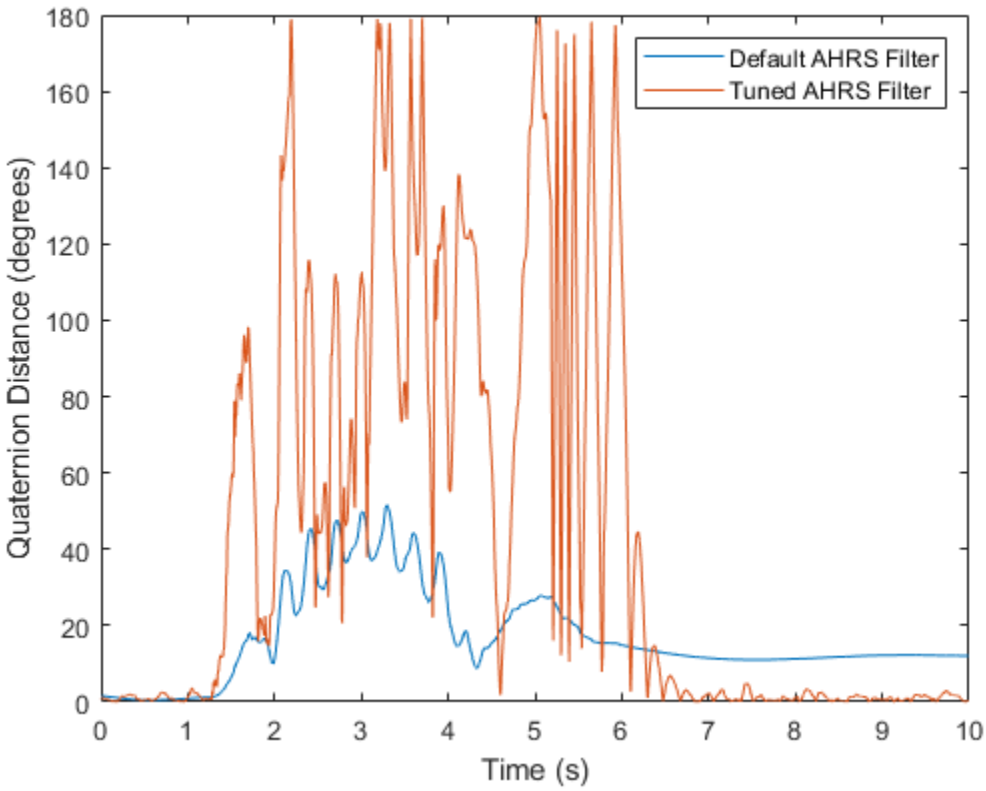
```
restingOrientation = ecompass(mean(accelReadings(6*SampleRate:end,:)), ...
                              mean(magReadings(6*SampleRate:end,:)));

figure(3)
plot(time,rad2deg(dist(restingOrientation,orientation)))
hold on
xlabel('Time (s)')
ylabel('Quaternion Distance (degrees)')
```



Modify the default `ahrsfilter` properties so that the filter converges to gravity more quickly. Increase the `GyroscopeDriftNoise` to `1e-2` and decrease the `LinearAccelerationNoise` to `1e-4`. This instructs the `ahrsfilter` algorithm to weigh gyroscope data less and accelerometer data more. Because the `accelerometer` data provides the stabilizing and consistent gravity vector, the resulting orientation converges more quickly.

Reset the filter, fuse the data, and plot the results.

```
fuse.LinearAccelerationNoise = 1e-4;
fuse.GyroscopeDriftNoise     = 1e-2;
reset(fuse)

orientation = fuse(accelReadings,gyroReadings,magReadings);

figure(3)
plot(time,rad2deg(dist(restingOrientation,orientation)))
legend('Default AHRS Filter','Tuned AHRS Filter')
```



Algorithms

Note: The following algorithm only applies to an NED reference frame.

The `ahrsfilter` uses the nine-axis Kalman filter structure described in [1]. The algorithm attempts to track the errors in orientation, gyroscope offset, linear acceleration, and magnetic disturbance to output the final orientation and angular velocity. Instead of tracking the orientation directly, the indirect Kalman filter models the error process, x, with a recursive update:

$$x_k = \begin{bmatrix} \theta_k \\ b_k \\ a_k \\ d_k \end{bmatrix} = F_k \begin{bmatrix} \theta_{k-1} \\ b_{k-1} \\ a_{k-1} \\ d_{k-1} \end{bmatrix} + w_k$$

where x_k is a 12-by-1 vector consisting of:

- θ_k –- 3-by-1 orientation error vector, in degrees, at time k
- b_k –- 3-by-1 gyroscope zero angular rate bias vector, in deg/s, at time k
- a_k –- 3-by-1 acceleration error vector measured in the sensor frame, in g, at time k
- d_k –- 3-by-1 magnetic disturbance error vector measured in the sensor frame, in µT, at time k

and where w_k is a 12-by-1 additive noise vector, and F_k is the state transition model.

Because x_k is defined as the error process, the *a priori* estimate is always zero, and therefore the state transition model, F_k, is zero. This insight results in the following reduction of the standard Kalman equations:

Standard Kalman equations:

$$x_k^- = F_k x_{k-1}^+$$
$$P_k^- = F_k P_{k-1}^+ F_k^T + Q_k$$

$$y_k = z_k - H_k x_k^-$$
$$S_k = R_k + H_k P_k^- H_k T$$
$$K_k = P_k^- H_k^T (S_k)^{-1}$$
$$x_k^+ = x_k^- + K_k y_k$$
$$P_k^+ = P_k - K_k H_k P_k^-$$

Kalman equations used in this algorithm:

$$x_k^- = 0$$
$$P_k^- = Q_k$$

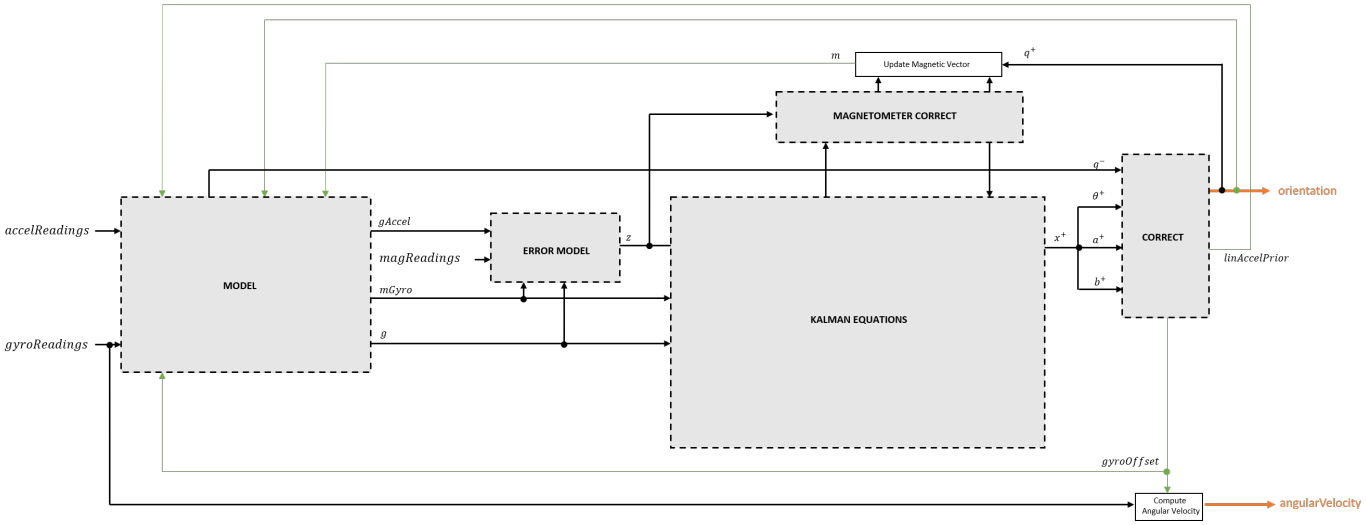
$$y_k = z_k$$
$$S_k = R_k + H_k P_k^- H_k T$$
$$K_k = P_k^- H_k^T (S_k)^{-1}$$
$$x_k^+ = K_k y_k$$
$$P_k^+ = P_k - K_k H_k P_k^-$$

where:

- x_k^- –– predicted (*a priori*) state estimate; the error process
- P_k^- –– predicted (*a priori*) estimate covariance
- y_k –– innovation
- S_k –– innovation covariance
- K_k –– Kalman gain
- x_k^+ –– updated (*a posteriori*) state estimate
- P_k^+ –– updated (*a posteriori*) estimate covariance

k represents the iteration, the superscript $^+$ represents an *a posteriori* estimate, and the superscript $^-$ represents an *a priori* estimate.

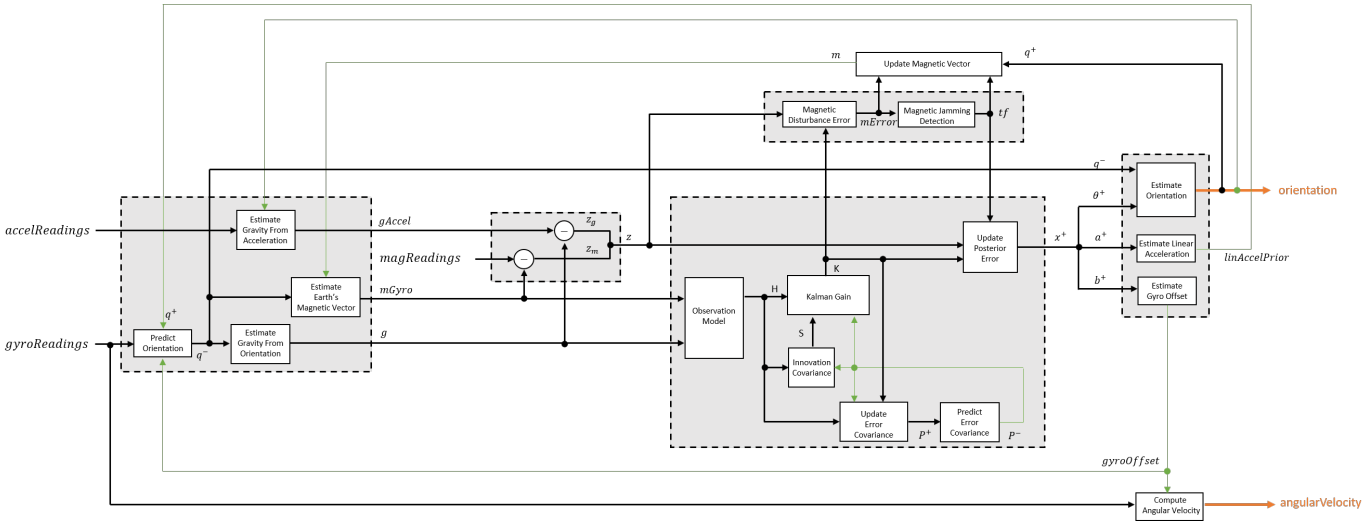
The graphic and following steps describe a single frame-based iteration through the algorithm.



Before the first iteration, the `accelReadings`, `gyroReadings`, and `magReadings` inputs are chunked into `DecimationFactor`-by-3 frames. For each chunk, the algorithm uses the most current accelerometer and magnetometer readings corresponding to the chunk of gyroscope readings.

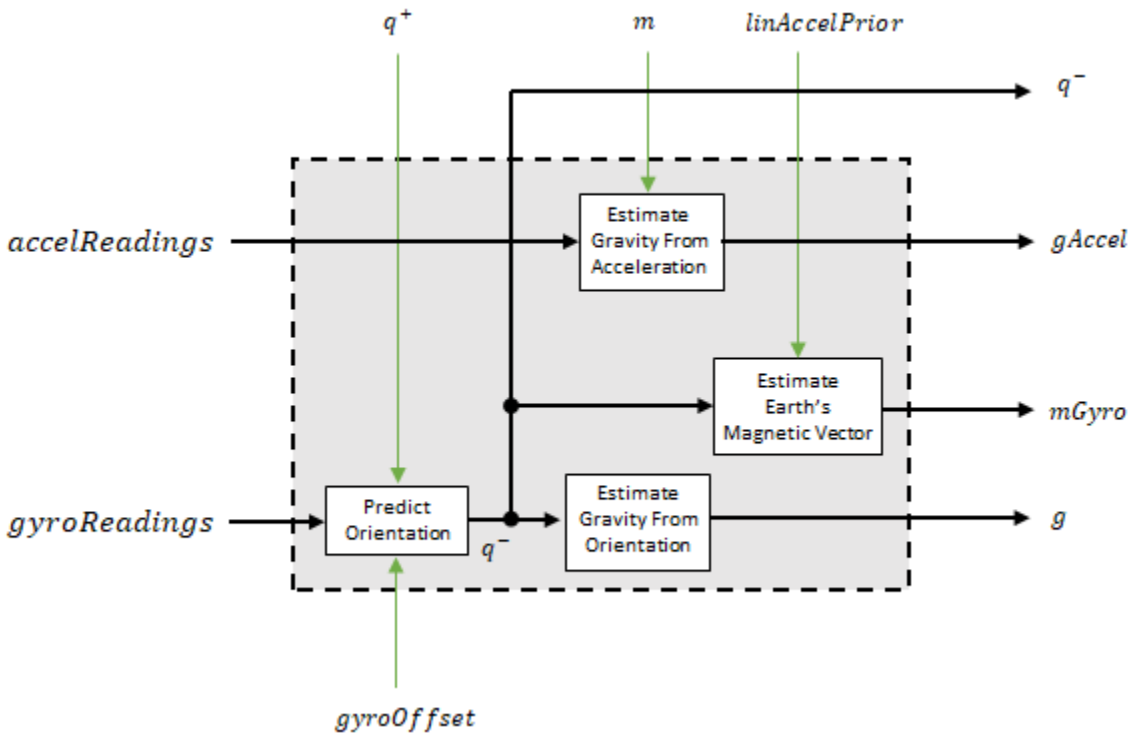
Detailed Overview

Walk through the algorithm for an explanation of each stage of the detailed overview.



Model

The algorithm models acceleration and angular change as linear processes.



Predict Orientation

The orientation for the current frame is predicted by first estimating the angular change from the previous frame:

$$\Delta\varphi_{N\times 3} = \frac{(gyroReadings_{N\times 3} - gyroOffset_{1\times 3})}{fs}$$

where N is the decimation factor specified by the DecimationFactor property and fs is the sample rate specified by the SampleRate property.

The angular change is converted into quaternions using the `rotvec` `quaternion` construction syntax:

$$\Delta Q_{N\times 1} = \text{quaternion}(\Delta\varphi_{N\times 3}, 'rotvec')$$

The previous orientation estimate is updated by rotating it by ΔQ:

$$q^-_{1\times 1} = \left(q^+_{1\times 1}\right)\left(\prod_{n=1}^{N}\Delta Q_n\right)$$

During the first iteration, the orientation estimate, q^-, is initialized by `ecompass`.

Estimate Gravity from Orientation

The gravity vector is interpreted as the third column of the quaternion, q^-, in rotation matrix form:

$$g_{1\times 3} = (rPrior(:,3))^T$$

See [1] for an explanation of why the third column of *rPrior* can be interpreted as the gravity vector.

Estimate Gravity from Acceleration

A second gravity vector estimation is made by subtracting the decayed linear acceleration estimate of the previous iteration from the accelerometer readings:

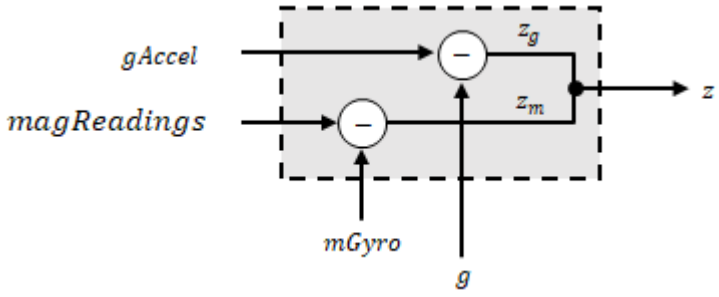
$$gAccel_{1\times 3} = accelReadings_{1\times 3} - linAccelprior_{1\times 3}$$

Estimate Earth's Magnetic Vector

Earth's magnetic vector is estimated by rotating the magnetic vector estimate from the previous iteration by the *a priori* orientation estimate, in rotation matrix form:

$$mGyro_{1\times 3} = \left((rPrior)\left(m^T\right)\right)^T$$

Error Model

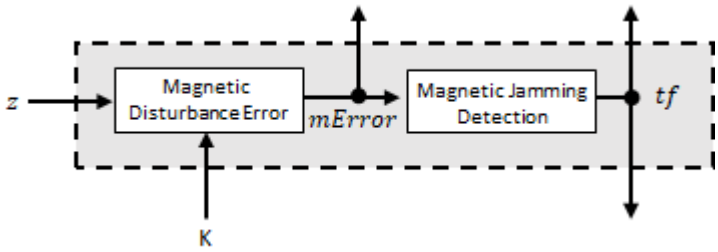


The error model combines two differences:

- The difference between the gravity estimate from the accelerometer readings and the gravity estimate from the gyroscope readings: $z_g = g - gAccel$
- The difference between the magnetic vector estimate from the gyroscope readings and the magnetic vector estimate from the magnetometer:$z_m = mGyro - magReadings$

Magnetometer Correct

The magnetometer correct estimates the error in the magnetic vector estimate and detects magnetic jamming.



Magnetometer Disturbance Error

The magnetic disturbance error is calculated by matrix multiplication of the Kalman gain associated with the magnetic vector with the error signal:

$$mError_{3\times 1} = \left((K(10{:}12,:)_{3\times 6})(z_{1\times 6})^T\right)^T$$

The Kalman gain, K, is the Kalman gain calculated in the current iteration.

Magnetic Jamming Detection

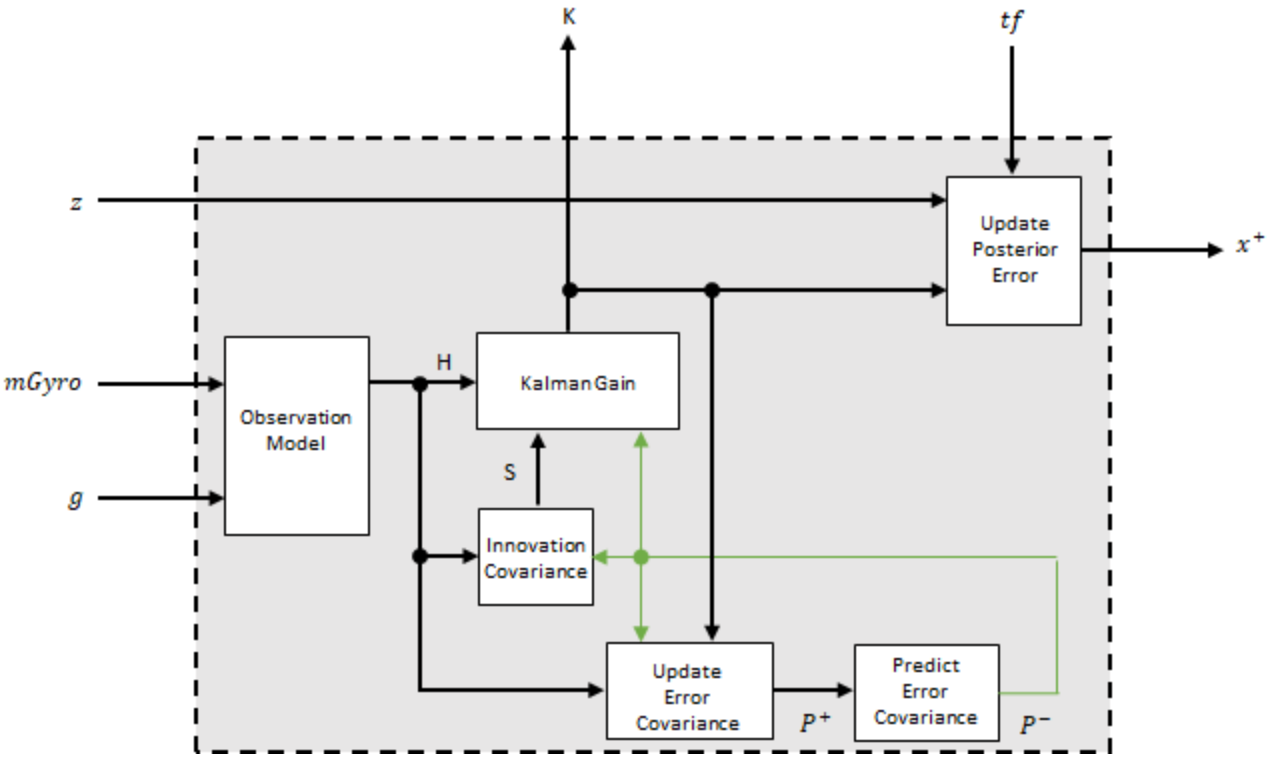
Magnetic jamming is determined by verifying that the power of the detected magnetic disturbance is less than or equal to four times the power of the expected magnetic field strength:

$$tf = \begin{cases} \text{true} & \text{if } \sum |mError|^2 > (4)(\text{ExpectedMagneticFieldStrength})^2 \\ \text{false} & \text{else} \end{cases}$$

ExpectedMagneticFieldStrength is a property of `ahrsfilter`.

Kalman Equations

The Kalman equations use the gravity estimate derived from the gyroscope readings, g, the magnetic vector estimate derived from the gyroscope readings, *mGyro*, and the observation of the error process, z, to update the Kalman gain and intermediary covariance matrices. The Kalman gain is applied to the error signal, z, to output an *a posteriori* error estimate, x^+.



Observation Model

The observation model maps the 1-by-3 observed states, g and *mGyro*, into the 6-by-12 true state, H.

The observation model is constructed as:

$$H_{3\times 9} = \begin{bmatrix} 0 & g_z & -g_y & 0 & -\kappa g_z & \kappa g_y & 1 & 0 & 0 & 0 & 0 & 0 \\ -g_z & 0 & g_x & \kappa g_z & 0 & -\kappa g_x & 0 & 1 & 0 & 0 & 0 & 0 \\ g_y & -g_x & 0 & -\kappa g_y & \kappa g_x & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & m_z & -m_y & 0 & -\kappa m_z & -\kappa m_y & 0 & 0 & 0 & -1 & 0 & 0 \\ -m_z & 0 & m_x & \kappa m_z & 0 & -\kappa m_x & 0 & 0 & 0 & 0 & -1 & 0 \\ m_y & -m_x & 0 & -\kappa m_y & \kappa m_x & 0 & 0 & 0 & 0 & 0 & 0 & -1 \end{bmatrix}$$

where g_x, g_y, and g_z are the *x*-, *y*-, and *z*-elements of the gravity vector estimated from the *a priori* orientation, respectively. m_x, m_y, and m_z are the *x*-, *y*-, and *z*-elements of the magnetic vector estimated from the *a priori* orientation, respectively. κ is a constant determined by the SampleRate and DecimationFactor properties: κ = `DecimationFactor/SampleRate`.

See sections 7.3 and 7.4 of [1] for a derivation of the observation model.

Innovation Covariance

The innovation covariance is a 6-by-6 matrix used to track the variability in the measurements. The innovation covariance matrix is calculated as:

$$S_{6x6} = R_{6x6} + (H_{6x12})\left(P^{-}_{12x12}\right)(H_{6x12})^{T}$$

where

- H is the observation model matrix
- P^{-} is the predicted (*a priori*) estimate of the covariance of the observation model calculated in the previous iteration
- R is the covariance of the observation model noise, calculated as:

$$R_{6\times 6} = \begin{bmatrix} accel_{\text{noise}} & 0 & 0 & 0 & 0 & 0 \\ 0 & accel_{\text{noise}} & 0 & 0 & 0 & 0 \\ 0 & 0 & accel_{\text{noise}} & 0 & 0 & 0 \\ 0 & 0 & 0 & mag_{\text{noise}} & 0 & 0 \\ 0 & 0 & 0 & 0 & mag_{\text{noise}} & 0 \\ 0 & 0 & 0 & 0 & 0 & mag_{\text{noise}} \end{bmatrix}$$

where

$accel_{\text{noise}}$ = AccelerometerNoise + LinearAccelerationNoise + κ^2 (GyroscopeDriftNoise + GyroscopeNoise)

and

mag_{noise} = MagnetometerNoise + MagneticDisturbanceNoise + κ^2 (GyroscopeDriftNoise + GyroscopeNoise)

The following properties define the observation model noise variance:

- κ –- DecimationFactor/SampleRate

- AccelerometerNoise
- LinearAccelerationNoise
- GyroscopeDriftNoise
- GyroscopeNoise
- MagneticDisturbanceNoise
- MagnetometerNoise

Update Error Estimate Covariance

The error estimate covariance is a 12-by-12 matrix used to track the variability in the state.

The error estimate covariance matrix is updated as:

$$P^{+}_{12\times 12} = P^{-}_{12\times 12} - (K_{12\times 6})(H_{6\times 12})\left(P^{-}_{12\times 12}\right)$$

where K is the Kalman gain, H is the measurement matrix, and P^{-} is the error estimate covariance calculated during the previous iteration.

Predict Error Estimate Covariance

The error estimate covariance is a 12-by-12 matrix used to track the variability in the state. The *a priori* error estimate covariance, P^{-}, is set to the process noise covariance, Q, determined during the previous iteration. Q is calculated as a function of the *a posteriori* error estimate covariance, P^{+}. When calculating Q, it is assumed that the cross-correlation terms are negligible compared to the autocorrelation terms, and are set to zero:

$$Q =$$

$$\begin{bmatrix} P^{+}(1)+\kappa^{2}P^{+}(40)+\beta+\eta & 0 & 0 & -\kappa\left(P^{+}(40)+\beta\right) & 0 \\ 0 & P^{+}(14)+\kappa^{2}P^{+}(53)+\beta+\eta & 0 & 0 & -\kappa\left(P^{+}(53)+ \\ 0 & 0 & P^{+}(27)+\kappa^{2}P^{+}(66)+\beta+\eta & 0 & 0 \\ -\kappa\left(P^{+}(40)+\beta\right) & 0 & 0 & P^{+}(40)+\beta & 0 \\ 0 & -\kappa\left(P^{+}(53)+\beta\right) & 0 & 0 & P^{+}(53)+\beta \\ 0 & 0 & -\kappa\left(P^{+}(66)+\beta\right) & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

where

- P^+ –– is the updated (*a posteriori*) error estimate covariance
- κ –– DecimationFactor/SampleRate
- β –– GyroscopeDriftNoise
- η –– GyroscopeNoise
- ν –– LinearAcclerationDecayFactor
- ξ –– LinearAccelerationNoise
- σ –– MagneticDisturbanceDecayFactor
- γ –– MagneticDisturbanceNoise

See section 10.1 of [1] for a derivation of the terms of the process error matrix.

Kalman Gain

The Kalman gain matrix is a 12-by-6 matrix used to weight the innovation. In this algorithm, the innovation is interpreted as the error process, z.

The Kalman gain matrix is constructed as:

$$K_{12\times 6} = \left(P^-_{12\times 12}\right)\left(H_{6\times 12}\right)^T\left(\left(S_{6\times 6}\right)^T\right)^{-1}$$

where

- P^- –– predicted error covariance
- H –– observation model
- S –– innovation covariance

Update a Posteriori Error

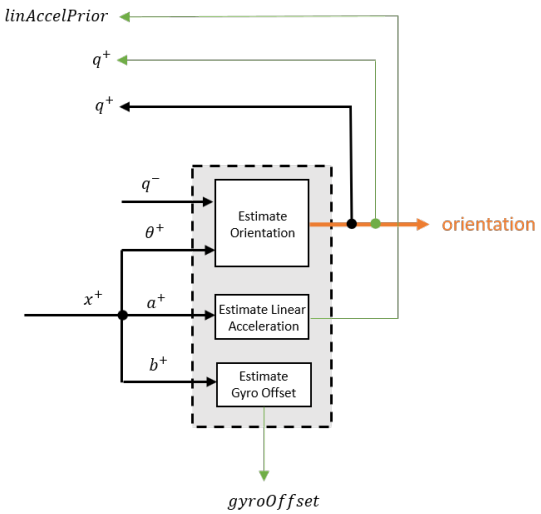
The *a posterior* error estimate is determined by combining the Kalman gain matrix with the error in the gravity vector and magnetic vector estimations:

$$x_{12\times 1} = \left(K_{12\times 6}\right)\left(z_{1\times 6}\right)^T$$

If magnetic jamming is detected in the current iteration, the magnetic vector error signal is ignored, and the *a posterior* error estimate is calculated as:

$$x_{9\times 1} = \left(K(1{:}9, 1{:}3)\right)\left(z_g\right)^T$$

Correct



Estimate Orientation

The orientation estimate is updated by multiplying the previous estimation by the error:

$$q^{+} = (q^{-})\left(\theta^{+}\right)$$

Estimate Linear Acceleration

The linear acceleration estimation is updated by decaying the linear acceleration estimation from the previous iteration and subtracting the error:

$$linAccelPrior = (linAccelPrior_{k-1})\nu - b^{+}$$

where

- ν –– LinearAcclerationDecayFactor

Estimate Gyroscope Offset

The gyroscope offset estimation is updated by subtracting the gyroscope offset error from the gyroscope offset from the previous iteration:

$$gyroOffset = gyroOffset_{k-1} - a^{+}$$

Compute Angular Velocity

To estimate angular velocity, the frame of `gyroReadings` are averaged and the gyroscope offset computed in the previous iteration is subtracted:

$$angularVelocity_{1\times 3} = \frac{\sum gyroReadings_{N\times 3}}{N} - gyroOffset_{1\times 3}$$

where N is the decimation factor specified by the `DecimationFactor` property.

The gyroscope offset estimation is initialized to zeros for the first iteration.

Update Magnetic Vector

If magnetic jamming was not detected in the current iteration, the magnetic vector estimate, m, is updated using the *a posteriori* magnetic disturbance error and the *a posteriori* orientation.

The magnetic disturbance error is converted to the navigation frame:

$$mErrorNED_{1\times 3} = \left((rPost_{3\times 3})^T (mError_{1\times 3})^T\right)^T$$

The magnetic disturbance error in the navigation frame is subtracted from the previous magnetic vector estimate and then interpreted as inclination:

$$M = m - mErrorNED$$

$$inclination = \text{atan2}(M(3), M(1))$$

The inclination is converted to a constrained magnetic vector estimate for the next iteration:

$m(1)$ = (ExpectedMagneticFieldStrength)(cos(*inclination*))
$m(2) = 0$
$m(3)$ = (ExpectedMagneticFieldStrength)(sin(*inclination*))

ExpectedMagneticFieldStrength is a property of `ahrsfilter`.

References

[1] Open Source Sensor Fusion. https://github.com/memsindustrygroup/Open-Source-Sensor-Fusion/tree/master/docs

[2] Roetenberg, D., H.J. Luinge, C.T.M. Baten, and P.H. Veltink. "Compensation of Magnetic Disturbances Improves Inertial and Magnetic Sensing of Human Body Segment Orientation." *IEEE Transactions on Neural Systems and Rehabilitation Engineering*. Vol. 13. Issue 3, 2005, pp. 395-405.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

`ecompass` | `imufilter` | `imuSensor` | `gpsSensor`

Introduced in R2018b

tune

Tune `ahrsfilter` parameters to reduce estimation error

Syntax

```
tune(filter,sensorData,groundTruth)
tune( ___ ,config)
```

Description

`tune(filter,sensorData,groundTruth)` adjusts the properties of the `ahrsfilter` filter object, `filter`, to reduce the root-mean-squared (RMS) quaternion distance error between the fused sensor data and the ground truth. The function uses the property values in the filter as the initial estimate for the optimization algorithm.

`tune( ___ ,config)` specifies the tuning configuration based on a `tunerconfig` object, `config`.

Examples

Tune `ahrsfilter` to Improve Orientation Estimate

Load recorded sensor data and ground truth data.

```
ld = load('ahrsfilterTuneData.mat');
qTrue = ld.groundTruth.Orientation; % true orientation
```

Create an `arhsfitler` object.

```
fuse = ahrsfilter;
```

Fuse the sensor data using the default, untuned filter.

```
qEstUntuned = fuse(ld.sensorData.Accelerometer, ...
    ld.sensorData.Gyroscope, ld.sensorData.Magnetometer);
```

Create a `tunerconfig` object. Tune the `ahrsfilter` object to improve the orientation estimation based on the configuration.

```
config = tunerconfig('ahrsfilter');
tune(fuse,ld.sensorData,ld.groundTruth,config);
```

```
    Iteration    Parameter                       Metric
    _________    _________                       ______
    1            AccelerometerNoise              0.1345
    1            GyroscopeNoise                  0.1342
    1            MagnetometerNoise               0.1341
    1            GyroscopeDriftNoise             0.1341
    1            LinearAccelerationNoise         0.1332
    1            MagneticDisturbanceNoise        0.1324
    1            LinearAccelerationDecayFactor   0.1317
    1            MagneticDisturbanceDecayFactor  0.1316
```

```
2           AccelerometerNoise               0.1316
2           GyroscopeNoise                   0.1312
2           MagnetometerNoise                0.1311
2           GyroscopeDriftNoise              0.1311
2           LinearAccelerationNoise          0.1300
2           MagneticDisturbanceNoise         0.1292
2           LinearAccelerationDecayFactor    0.1285
2           MagneticDisturbanceDecayFactor   0.1285
3           AccelerometerNoise               0.1285
3           GyroscopeNoise                   0.1280
3           MagnetometerNoise                0.1279
3           GyroscopeDriftNoise              0.1279
3           LinearAccelerationNoise          0.1267
3           MagneticDisturbanceNoise         0.1258
3           LinearAccelerationDecayFactor    0.1253
3           MagneticDisturbanceDecayFactor   0.1253
4           AccelerometerNoise               0.1252
4           GyroscopeNoise                   0.1247
4           MagnetometerNoise                0.1246
4           GyroscopeDriftNoise              0.1246
4           LinearAccelerationNoise          0.1233
4           MagneticDisturbanceNoise         0.1224
4           LinearAccelerationDecayFactor    0.1220
4           MagneticDisturbanceDecayFactor   0.1220
5           AccelerometerNoise               0.1220
5           GyroscopeNoise                   0.1213
5           MagnetometerNoise                0.1212
5           GyroscopeDriftNoise              0.1212
5           LinearAccelerationNoise          0.1200
5           MagneticDisturbanceNoise         0.1190
5           LinearAccelerationDecayFactor    0.1187
5           MagneticDisturbanceDecayFactor   0.1187
6           AccelerometerNoise               0.1187
6           GyroscopeNoise                   0.1180
6           MagnetometerNoise                0.1178
6           GyroscopeDriftNoise              0.1178
6           LinearAccelerationNoise          0.1167
6           MagneticDisturbanceNoise         0.1156
6           LinearAccelerationDecayFactor    0.1155
6           MagneticDisturbanceDecayFactor   0.1155
7           AccelerometerNoise               0.1155
7           GyroscopeNoise                   0.1147
7           MagnetometerNoise                0.1145
7           GyroscopeDriftNoise              0.1145
7           LinearAccelerationNoise          0.1137
7           MagneticDisturbanceNoise         0.1126
7           LinearAccelerationDecayFactor    0.1125
7           MagneticDisturbanceDecayFactor   0.1125
8           AccelerometerNoise               0.1125
8           GyroscopeNoise                   0.1117
8           MagnetometerNoise                0.1116
8           GyroscopeDriftNoise              0.1116
8           LinearAccelerationNoise          0.1112
8           MagneticDisturbanceNoise         0.1100
8           LinearAccelerationDecayFactor    0.1099
8           MagneticDisturbanceDecayFactor   0.1099
9           AccelerometerNoise               0.1099
9           GyroscopeNoise                   0.1091
```

```
    9           MagnetometerNoise                0.1090
    9           GyroscopeDriftNoise              0.1090
    9           LinearAccelerationNoise          0.1090
    9           MagneticDisturbanceNoise         0.1076
    9           LinearAccelerationDecayFactor    0.1075
    9           MagneticDisturbanceDecayFactor   0.1075
    10          AccelerometerNoise               0.1075
    10          GyroscopeNoise                   0.1066
    10          MagnetometerNoise                0.1064
    10          GyroscopeDriftNoise              0.1064
    10          LinearAccelerationNoise          0.1064
    10          MagneticDisturbanceNoise         0.1049
    10          LinearAccelerationDecayFactor    0.1047
    10          MagneticDisturbanceDecayFactor   0.1047
    11          AccelerometerNoise               0.1047
    11          GyroscopeNoise                   0.1038
    11          MagnetometerNoise                0.1036
    11          GyroscopeDriftNoise              0.1036
    11          LinearAccelerationNoise          0.1036
    11          MagneticDisturbanceNoise         0.1016
    11          LinearAccelerationDecayFactor    0.1014
    11          MagneticDisturbanceDecayFactor   0.1014
    12          AccelerometerNoise               0.1014
    12          GyroscopeNoise                   0.1005
    12          MagnetometerNoise                0.1002
    12          GyroscopeDriftNoise              0.1002
    12          LinearAccelerationNoise          0.1002
    12          MagneticDisturbanceNoise         0.0978
```

Fuse the sensor data using the tuned filter.

```
qEstTuned = fuse(ld.sensorData.Accelerometer, ...
    ld.sensorData.Gyroscope, ld.sensorData.Magnetometer);
```

Compare the tuned and untuned RMS error performances.

```
dUntuned = rad2deg(dist(qEstUntuned, qTrue));
dTuned = rad2deg(dist(qEstTuned, qTrue));
rmsUntuned = sqrt(mean(dUntuned.^2))
```

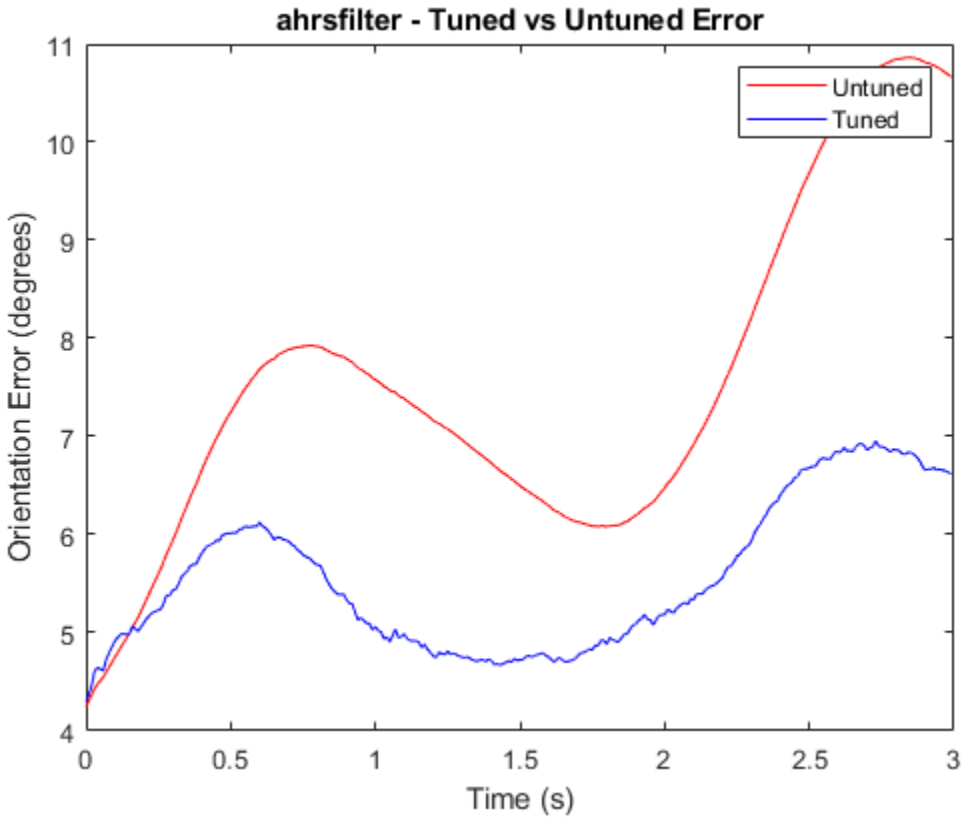
```
rmsUntuned = 7.7088
```

```
rmsTuned = sqrt(mean(dTuned.^2))
```

```
rmsTuned = 5.6033
```

Visualize the errors with respect to time.

```
N = numel(dUntuned);
t = (0:N-1)./ fuse.SampleRate;
plot(t, dUntuned, 'r', t, dTuned, 'b');
legend('Untuned', 'Tuned');
title('ahrsfilter - Tuned vs Untuned Error')
xlabel('Time (s)');
ylabel('Orientation Error (degrees)');
```



Input Arguments

`filter` — Filter object

`ahrsfilter` object

Filter object, specified as an `ahrsfilter` object.

`sensorData` — Sensor data

`table`

Sensor data, specified as a `table`. In each row, the sensor data is specified as:

- `Accelerometer` — Accelerometer data, specified as a 1-by-3 vector of scalars in m^2/s.
- `Gyroscope` — Gyroscope data, specified as a 1-by-3 vector of scalars in rad/s.
- `Magnetometer` — Magnetometer data, specified as a 1-by-3 vector of scalars in µT.

If you set the `Cost` property of the tuner configuration input, `config`, to `Custom`, then you can use other data types for the `sensorData` input based on your choice.

`groundTruth` — Ground truth data

`timetable`

Ground truth data, specified as a `table`. The table has only one column of `Orientation` data. In each row, the orientation is specified as a `quaternion` object or a 3-by-3 rotation matrix.

The function processes each row of the `sensorData` and `groundTruth` tables sequentially to calculate the state estimate and RMS error from the ground truth. Each row of the `sensorData` and the `groundTruth` tables must correspond to each other.

If you set the `Cost` property of the tuner configuration input, `config`, to `Custom`, then you can use other data types for the `groundTruth` input based on your choice.

`config` — Tuner configuration
`tunerconfig` object

Tuner configuration, specified as a `tunerconfig` object.

References

[1] Abbeel, P., Coates, A., Montemerlo, M., Ng, A.Y. and Thrun, S. Discriminative Training of Kalman Filters. In *Robotics: Science and systems*, Vol. 2, pp. 1, 2005.

See Also

Introduced in R2020b

altimeterSensor

Altimeter simulation model

Description

The `altimeterSensor` System object models receiving data from an altimeter sensor.

To model an altimeter:

1. Create the `altimeterSensor` object and set its properties.
2. Call the object with arguments, as if it were a function.

To learn more about how System objects work, see What Are System Objects?

Creation

Syntax

```
altimeter = altimeterSensor
altimeter = altimeterSensor('ReferenceFrame',RF)
altimeter = altimeterSensor(___,Name,Value)
```

Description

`altimeter = altimeterSensor` returns an `altimeterSensor` System object that simulates altimeter readings.

`altimeter = altimeterSensor('ReferenceFrame',RF)` returns an `altimeterSensor` System object that simulates altimeter readings relative to the reference frame `RF`. Specify `RF` as `'NED'` (North-East-Down) or `'ENU'` (East-North-Up). The default value is `'NED'`.

`altimeter = altimeterSensor(___,Name,Value)` sets each property `Name` to the specified `Value`. Unspecified properties have default values.

Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see System Design in MATLAB Using System Objects.

`SampleRate` — Update rate of sensor (Hz)

1 (default) | positive scalar

Update rate of sensor in Hz, specified as a positive scalar.

Data Types: `single` | `double`

`ConstantBias` — Constant offset bias (m)
`0` (default) | scalar

Constant offset bias in meters, specified as a scalar.

Tunable: Yes

Data Types: `single` | `double`

`NoiseDensity` — Power spectral density of sensor noise (m/√Hz)
`0` (default) | nonnegative scalar

Power spectral density of sensor noise in m/√Hz, specified as a nonnegative scalar.

Tunable: Yes

Data Types: `single` | `double`

`BiasInstability` — Instability of bias offset (m)
`0` (default) | nonnegative scalar

Instability of the bias offset in meters, specified as a nonnegative scalar.

Tunable: Yes

Data Types: `single` | `double`

`DecayFactor` — Bias instability noise decay factor
`0` (default) | scalar in the range [0,1]

Bias instability noise decay factor, specified as a scalar in the range [0,1]. A decay factor of `0` models the bias instability noise as a white noise process. A decay factor of `1` models the bias instability noise as a random walk process.

Tunable: Yes

Data Types: `single` | `double`

`RandomStream` — Random number source
`'Global stream'` (default) | `'mt19937ar with seed'`

Random number source, specified as a character vector or string:

- `'Global stream'` –– Random numbers are generated using the current global random number stream.
- `'mt19937ar with seed'` –– Random numbers are generated using the mt19937ar algorithm with the seed specified by the `Seed` property.

Data Types: `char` | `string`

`Seed` — Initial seed
`67` (default) | nonnegative integer scalar

Initial seed of an mt19937ar random number generator algorithm, specified as a nonnegative integer scalar.

Dependencies

To enable this property, set `RandomStream` to `'mt19937ar with seed'`.

Data Types: `single` | `double`

Usage

Syntax

```
altimeterReadings = altimeter(position)
```

Description

`altimeterReadings = altimeter(position)` generates an altimeter sensor altitude reading from the `position` input.

Input Arguments

`position` — Position of sensor in local navigation coordinate system (m)
N-by-3 matrix

Position of sensor in the local navigation coordinate system, specified as an *N*-by-3 matrix with elements measured in meters. *N* is the number of samples in the current frame.

Data Types: `single` | `double`

Output Arguments

`altimeterReadings` — Altitude of sensor relative to local navigation coordinate system (m)
N-element column vector

Altitude of sensor relative to the local navigation coordinate system in meters, returned as an *N*-element column vector. *N* is the number of samples in the current frame.

Data Types: `single` | `double`

Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

Common to All System Objects

step	Run System object algorithm
release	Release resources and allow changes to System object property values and input characteristics
reset	Reset internal states of System object

Examples

Generate Noisy Altimeter Readings from Stationary Input

Create an `altimeterSensor` System object™ to model receiving altimeter sensor data. Assume a typical one Hz sample rate and a 10 minute simulation time. Set `ConstantBias` to `0.01`, `NoiseDensity` to `0.05`, `BiasInstability` to `0.05`, and `DecayFactor` to `0.5`.

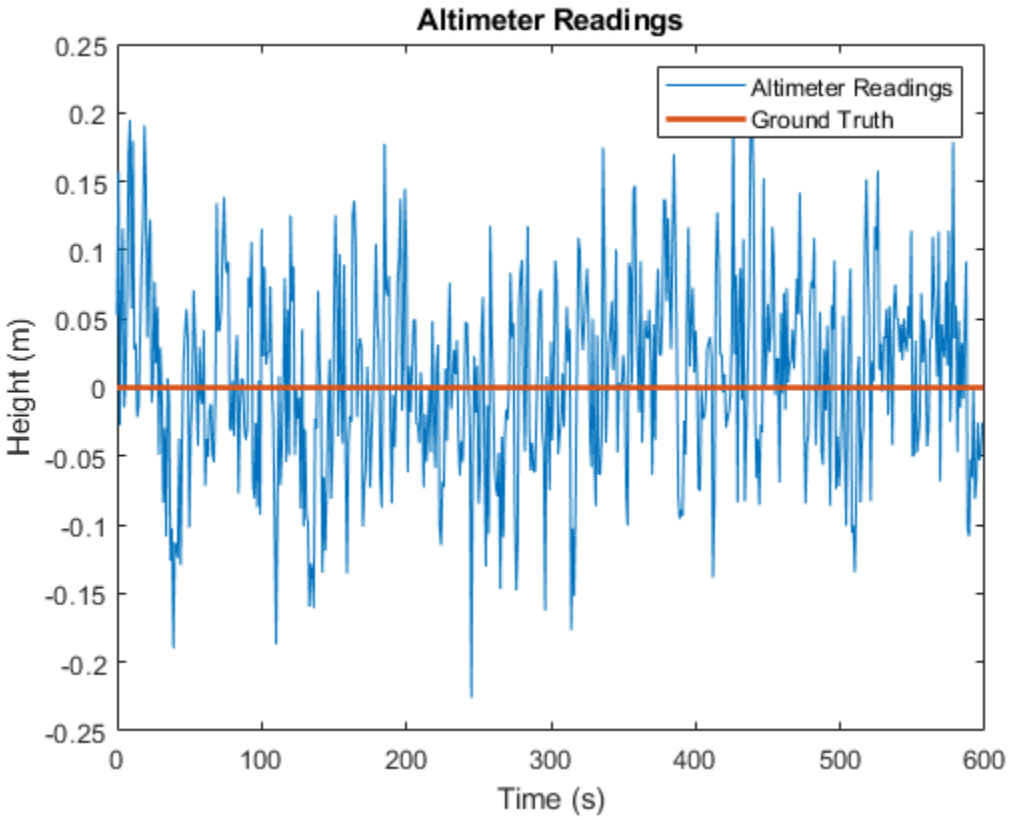
```
Fs = 1;
duration = 60*10;
numSamples = duration*Fs;


altimeter = altimeterSensor('SampleRate',Fs, ...
                            'ConstantBias',0.01, ...
                            'NoiseDensity',0.05, ...
                            'BiasInstability',0.05, ...
                            'DecayFactor',0.5);

truePosition = zeros(numSamples,3);
```

Call `altimeter` with the specified `truePosition` to model noisy altimeter readings from a stationary platform.

```
altimeterReadings = altimeter(truePosition);
```

Plot the true position and the altimeter sensor readings for height.

```
t = (0:(numSamples-1))/Fs;

plot(t,altimeterReadings)
hold on
plot(t,truePosition(:,3),'LineWidth',2)
hold off
title('Altimeter Readings')
xlabel('Time (s)')
ylabel('Height (m)')
legend('Altimeter Readings','Ground Truth')
```



Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

`gpsSensor` | `imuSensor`

Introduced in R2019a

complementaryFilter

Orientation estimation from a complementary filter

Description

The `complementaryFilter` System object fuses accelerometer, gyroscope, and magnetometer sensor data to estimate device orientation and angular velocity.

To estimate orientation using this object:

1. Create the `complementaryFilter` object and set its properties.
2. Call the object with arguments, as if it were a function.

To learn more about how System objects work, see What Are System Objects?

Creation

Syntax

```
FUSE = complementaryFilter
FUSE = complementaryFilter('ReferenceFrame',RF)
FUSE = complementaryFilter( ___ ,Name,Value)
```

Description

`FUSE = complementaryFilter` returns a `complementaryFilter` System object, `FUSE`, for sensor fusion of accelerometer, gyroscope, and magnetometer data to estimate device orientation and angular velocity.

`FUSE = complementaryFilter('ReferenceFrame',RF)` returns a `complementaryFilter` System object that fuses accelerometer, gyroscope, and magnetometer data to estimate device orientation relative to the reference frame `RF`. Specify `RF` as `'NED'` (North-East-Down) or `'ENU'` (East-North-Up). The default value is `'NED'`.

`FUSE = complementaryFilter( ___ ,Name,Value)` sets each property `Name` to the specified `Value`. Unspecified properties have default values.

Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see System Design in MATLAB Using System Objects.

`SampleRate` — Input sample rate of sensor data (Hz)
`100` (default) | positive scalar

Input sample rate of the sensor data in Hz, specified as a positive scalar.

Tunable: No

Data Types: `single` | `double`

`AccelerometerGain` — Accelerometer gain
`0.01` (default) | real scar in [0, 1]

Accelerometer gain, specified as a real scalar in the range of [0, 1]. The gain determines how much the accelerometer measurement is trusted over the gyroscope measurement for orientation estimation. This property is tunable.

Data Types: `single` | `double`

`MagnetometerGain` — Magnetometer gain
`0.01` (default) | real scar in [0, 1]

Magnetometer gain, specified as a real scalar in the range of [0, 1]. The gain determines how much the magnetometer measurement is trusted over the gyroscope measurement for orientation estimation. This property is tunable.

Data Types: `single` | `double`

`HasMagnetometer` — Enable magnetometer input
`true` (default) | `false`

Enable magnetometer input, specified as `true` or `false`.

Data Types: `logical`

`OrientationFormat` — Output orientation format
`'quaternion'` (default) | `'Rotation matrix'`

Output orientation format, specified as `'quaternion'` or `'Rotation matrix'`. The size of the output depends on the output orientation format:

- `'quaternion'` –– Output is an *N*-by-1 `quaternion`.
- `'Rotation matrix'` –– Output is a 3-by-3-by-*N* rotation matrix.

N is the number of samples.

Data Types: `char` | `string`

Usage

Syntax

```
[orientation,angularVelocity] = FUSE(accelReadings,gyroReadings,magReadings)
[orientation,angularVelocity] = FUSE(accelReadings,gyroReadings)
```

Description

`[orientation,angularVelocity] = FUSE(accelReadings,gyroReadings,magReadings)` fuses accelerometer, gyroscope, and magnetometer data to compute orientation and angular velocity. To use this syntax, set the `HasMagnetometer` property as `true`.

`[orientation,angularVelocity] = FUSE(accelReadings,gyroReadings)` fuses accelerometer and gyroscope data to compute orientation and angular velocity. To use this syntax, set the `HasMagnetometer` property as `false`.

Input Arguments

`accelReadings` — Accelerometer readings in sensor body coordinate system (m/s^2)

N-by-3 matrix

Accelerometer readings in the sensor body coordinate system in m/s^2, specified as an *N*-by-3 matrix. *N* is the number of samples, and the three columns of `accelReadings` represent the [*x y z*] measurements. Accelerometer readings are assumed to correspond to the sample rate specified by the `SampleRate` property. In the filter, the gravity constant *g* is assumed to be `9.81` m/s^2.

Data Types: `single` | `double`

`gyroReadings` — Gyroscope readings in sensor body coordinate system (rad/s)

N-by-3 matrix

Gyroscope readings in the sensor body coordinate system in rad/s, specified as an *N*-by-3 matrix. *N* is the number of samples, and the three columns of `gyroReadings` represent the [*x y z*] measurements. Gyroscope readings are assumed to correspond to the sample rate specified by the `SampleRate` property.

Data Types: `single` | `double`

`magReadings` — Magnetometer readings in sensor body coordinate system (µT)

N-by-3 matrix

Magnetometer readings in the sensor body coordinate system in µT, specified as an *N*-by-3 matrix. *N* is the number of samples, and the three columns of `magReadings` represent the [*x y z*] measurements. Magnetometer readings are assumed to correspond to the sample rate specified by the `SampleRate` property.

Data Types: `single` | `double`

Output Arguments

`orientation` — Orientation that rotates quantities from local navigation coordinate system to sensor body coordinate system

N-by-1 array of quaternions (default) | 3-by-3-by-*N* array

Orientation that rotates quantities from the local navigation coordinate system to the body coordinate system, returned as quaternions or an array. The size and type of `orientation` depends on whether the `OrienationFormat` property is set to `'quaternion'` or `'Rotation matrix'`:

- `'quaternion'` –– the output is an *N*-by-1 vector of quaternions, where *N* is the number of samples.
- `'Rotation matrix'` –– the output is a 3-by-3-by-*N* array of rotation matrices, where *N* is the number of samples.

Data Types: `quaternion` | `single` | `double`

`angularVelocity` — Angular velocity in sensor body coordinate system (rad/s)
N-by-3 array (default)

Angular velocity expressed in the sensor body coordinate system in rad/s, returned as an *N*-by-3 array, where *N* is the number of samples.

Data Types: `single` | `double`

Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

Common to All System Objects

step	Run System object algorithm
release	Release resources and allow changes to System object property values and input characteristics
reset	Reset internal states of System object
clone	Create duplicate System object
isLocked	Determine if System object is in use

Examples

Estimate Orientation from Recorded IMU Data

Load the `rpy_9axis` file, which contains recorded accelerometer, gyroscope, and magnetometer sensor data from a device oscillating in pitch (around *y*-axis), then yaw (around *z*-axis), and then roll (around *x*-axis). The file also contains the sample rate of the recording.

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

Create a complementary filter object with sample rate equal to the frequency of the data.

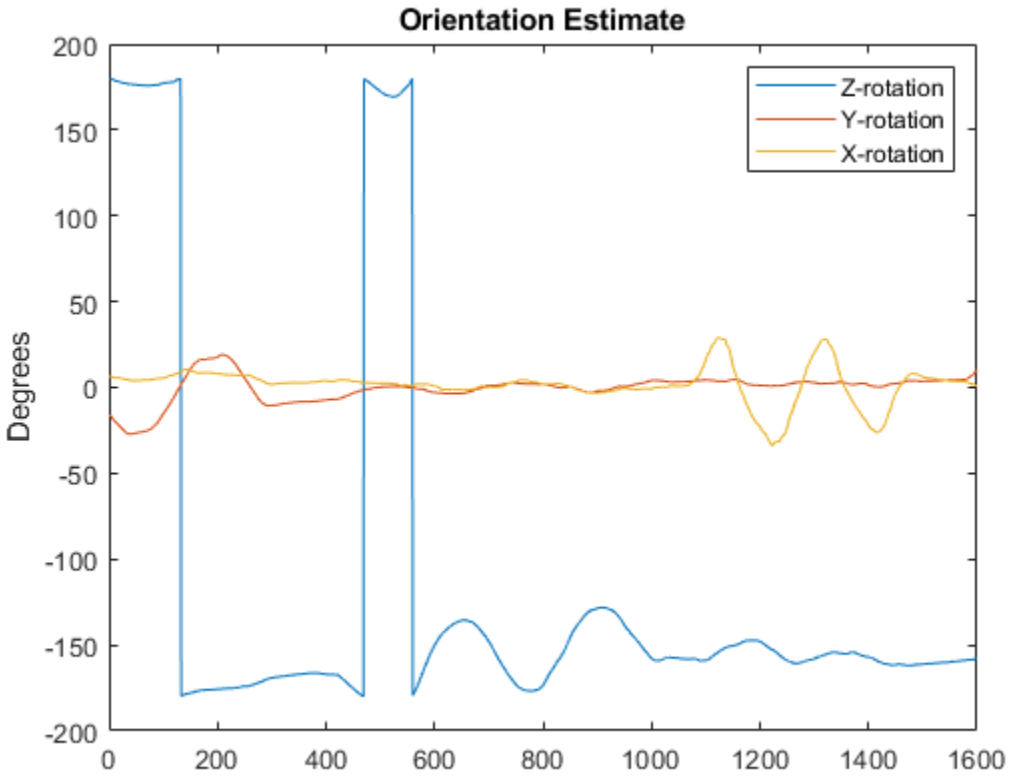
```
Fs  = ld.Fs;  % Hz
fuse = complementaryFilter('SampleRate', Fs);
```

Fuse accelerometer, gyroscope, and magnetometer data using the filter.

```
q = fuse(accel, gyro, mag);
```

Visualize the results.

```
plot(eulerd( q, 'ZYX', 'frame'));
title('Orientation Estimate');
legend('Z-rotation', 'Y-rotation', 'X-rotation');
ylabel('Degrees');
```



References

[1] Valenti, R., I. Dryanovski, and J. Xiao. "Keeping a good attitude: A quaternion-based orientation filter for IMUs and MARGs." *Sensors.* Vol. 15, Number 8, 2015, pp. 19302-19330.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`ahrsfilter` | `imufilter`

Introduced in R2019b

insfilterAsync

Estimate pose from asynchronous MARG and GPS data

Description

The `insfilterAsync` object implements sensor fusion of MARG and GPS data to estimate pose in the NED (or ENU) reference frame. MARG (magnetic, angular rate, gravity) data is typically derived from magnetometer, gyroscope, and accelerometer data, respectively. The filter uses a 28-element state vector to track the orientation `quaternion`, velocity, position, MARG sensor biases, and geomagnetic vector. The `insfilterAsync` object uses a continuous-discrete extended Kalman filter to estimate these quantities.

Creation

Syntax

```
filter = insfilterAsync
filter = insfilterAsync('ReferenceFrame',RF)
filter = insfilterAsync(___,Name,Value)
```

Description

`filter = insfilterAsync` creates an `insfilterAsync` object to fuse asynchronous MARG and GPS data with default property values.

`filter = insfilterAsync('ReferenceFrame',RF)` allows you to specify the reference frame, `RF`, of the `filter`. Specify `RF` as `'NED'` (North-East-Down) or `'ENU'` (East-North-Up). The default value is `'NED'`.

`filter = insfilterAsync(___,Name,Value)` also allows you set properties of the created `filter` using one or more name-value pairs. Enclose each property name in single quotes.

Properties

`ReferenceLocation` — Reference location (deg, deg, meters)
`[0 0 0]` (default) | three-element positive row vector

Reference location, specified as a three-element row vector in geodetic coordinates (latitude, longitude, and altitude). Altitude is the height above the reference ellipsoid model, WGS84. The reference location units are [degrees degrees meters].

Data Types: `single` | `double`

`QuaternionNoise` — Additive quaternion process noise variance
`[1e-6 1e-6 1e-6 1e-6]` (default) | scalar | four-element row vector

Additive quaternion process noise variance, specified as a scalar or four-element vector of quaternion parts.

Data Types: `single` | `double`

`AngularVelocityNoise` — Additive angular velocity process noise in local navigation coordinate system ((rad/s)^2)
`[0.005 0.005 0.005]` (default) | scalar | three-element row vector

Additive angular velocity process noise in the local navigation coordinate system in (rad/s)^2, specified as a scalar or three-element row vector of positive real finite numbers.

- If `AngularVelocityNoise` is a row vector, the elements correspond to the noise in the *x*, *y*, and *z* axes of the local navigation coordinate system, respectively.
- If `AngularVelocityNoise` is a scalar, the single element is applied to each axis.

Data Types: `single` | `double`

`PositionNoise` — Additive position process noise variance in local navigation coordinate system (m^2)
`[1e-6 1e-6 1e-6]` (default) | scalar | three-element row vector

Additive position process noise in the local navigation coordinate system in m^2, specified as a scalar or three-element row vector of positive real finite numbers.

- If `PositionNoise` is a row vector, the elements correspond to the noise in the *x*, *y*, and *z* axes of the local navigation coordinate system, respectively.
- If `PositionNoise` is a scalar, the single element is applied to each axis.

Data Types: `single` | `double`

`VelocityNoise` — Additive velocity process noise variance in local navigation coordinate system ((m/s)^2)
`[1e-6 1e-6 1e-6]` (default) | scalar | three-element row vector

Additive velocity process noise in the local navigation coordinate system in (m/s)^2, specified as a scalar or three-element row vector of positive real finite numbers.

- If `VelocityNoise` is a row vector, the elements correspond to the noise in the *x*, *y*, and *z* axes of the local navigation coordinate system, respectively.
- If `VelocityNoise` is a scalar, the single element is applied to each axis.

Data Types: `single` | `double`

`AccelerationNoise` — Additive acceleration process noise variance in local navigation coordinate system ((m/s^2)^2)
`[50 50 50]` (default) | scalar | three-element row vector

Additive acceleration process noise in (m/s^2)^2, specified as a scalar or three-element row vector of positive real finite numbers.

- If `AccelerationNoise` is a row vector, the elements correspond to the noise in the *x*, *y*, and *z* axes of the local navigation coordinate system, respectively.
- If `AccelerationNoise` is a scalar, the single element is applied to each axis.

Data Types: `single` | `double`

`GyroscopeBiasNoise` — Additive process noise variance from gyroscope bias ((rad/s)²)
`[1e-10 1e-10 1e-10]` (default) | scalar | three-element row vector

Additive process noise variance from the gyroscope bias in $(rad/s)^2$, specified as a scalar or three-element row vector of positive real finite numbers.

- If `GyroscopeBiasNoise` is a row vector, the elements correspond to the noise in the *x*, *y*, and *z* axes of the gyroscope, respectively.
- If `GyroscopeBiasNoise` is a scalar, the single element is applied to each axis.

Data Types: `single` | `double`

`AccelerometerBiasNoise` — Additive process noise variance from accelerometer bias ((m/s²)²)
`[1e-4 1e-4 1e-4]` (default) | positive scalar | three-element row vector

Additive process noise variance from accelerometer bias in $(m/s^2)^2$, specified as a scalar or three-element row vector of positive real numbers.

- If `AccelerometerBiasNoise` is a row vector, the elements correspond to the noise in the *x*, *y*, and *z* axes of the accelerometer, respectively.
- If `AccelerometerBiasNoise` is a scalar, the single element is applied to each axis.

`GeomagneticVectorNoise` — Additive process noise variance of geomagnetic vector in local navigation coordinate system (μT²)
`[1e-6 1e-6 1e-6]` (default) | positive scalar | three-element row vector

Additive process noise variance of geomagnetic vector in μT^2, specified as a scalar or three-element row vector of positive real numbers.

- If `GeomagneticVectorNoise` is a row vector, the elements correspond to the noise in the *x*, *y*, and *z* axes of the local navigation coordinate system, respectively.
- If `GeomagneticVectorNoise` is a scalar, the single element is applied to each axis.

`MagnetometerBiasNoise` — Additive process noise variance from magnetometer bias (μT²)
`[0.1 0.1 0.1]` (default) | positive scalar | three-element row vector

Additive process noise variance from magnetometer bias in μT^2, specified as a scalar or three-element row vector of positive real numbers.

- If `MagnetometerBiasNoise` is a row vector, the elements correspond to the noise in the *x*, *y*, and *z* axes of the magnetometer, respectively.
- If `MagnetometerBiasNoise` is a scalar, the single element is applied to each axis.

`State` — State vector of extended Kalman filter
28-element column vector

State vector of the extended Kalman filter. The state values represent:

State	Units	Index
Orientation (quaternion parts)	N/A	1:4
Angular Velocity (XYZ)	rad/s	5:7

State	Units	Index
Position (NED or ENU)	m	8:10
Velocity (NED or ENU)	m/s	11:13
Acceleration (NED or ENU)	m/s^2	14:16
Accelerometer Bias (XYZ)	m/s^2	17:19
Gyroscope Bias (XYZ)	rad/s	20:22
Geomagnetic Field Vector (NED or ENU)	µT	23:25
Magnetometer Bias (XYZ)	µT	26:28

The default initial state corresponds to an object at rest located at `[0 0 0]` in geodetic LLA coordinates.

Data Types: `single` | `double`

`StateCovariance` — State error covariance for extended Kalman filter
`eye(28)` (default) | 28-by-28 matrix

State error covariance for the extended Kalman filter, specified as a 28-by-28-element matrix of real numbers.

Data Types: `single` | `double`

Object Functions

predict	Update states based on motion model for insfilterAsync
fuseaccel	Correct states using accelerometer data for insfilterAsync
fusegyro	Correct states using gyroscope data for insfilterAsync
fusemag	Correct states using magnetometer data for insfilterAsync
fusegps	Correct states using GPS data for insfilterAsync
correct	Correct states using direct state measurements for insfilterAsync
residual	Residuals and residual covariances from direct state measurements for insfilterAsync
residualaccel	Residuals and residual covariance from accelerometer measurements for insfilterAsync
residualgps	Residuals and residual covariance from GPS measurements for insfilterAsync
residualmag	Residuals and residual covariance from magnetometer measurements for insfilterAsync
residualgyro	Residuals and residual covariance from gyroscope measurements for insfilterAsync
pose	Current position, orientation, and velocity estimate for insfilterAsync
reset	Reset internal states for insfilterAsync
stateinfo	Display state vector information for insfilterAsync
copy	Create copy of insfilterAsync
tune	Tune insfilterAsync parameters to reduce estimation error
tunernoise	Noise structure of fusion filter

Examples

Estimate Pose of UAV

Load logged sensor data and ground truth pose.

```
load('uavshort.mat','refloc','initstate','imuFs', ...
    'accel','gyro','mag','lla','gpsvel', ...
    'trueOrient','truePos')
```

Create an INS filter to fuse asynchronous MARG and GPS data to estimate pose.

```
filt = insfilterAsync;
filt.ReferenceLocation = refloc;
filt.State = [initstate(1:4);0;0;0;initstate(5:10);0;0;0;initstate(11:end)];
```

Define sensor measurement noises. The noises were determined from datasheets and experimentation.

```
Rmag  = 80;
Rvel  = 0.0464;
Racc  = 800;
Rgyro = 1e-4;
Rpos  = 34;
```

Preallocate variables for position and orientation. Allocate a variable for indexing into the GPS data.

```
N = size(accel,1);
p = zeros(N,3);
q = zeros(N,1,'quaternion');

gpsIdx = 1;
```

Fuse accelerometer, gyroscope, magnetometer, and GPS data. The outer loop predicts the filter forward one time step and fuses accelerometer and gyroscope data at the IMU sample rate.

```
for ii = 1:N

    % Predict the filter forward one time step
    predict(filt,1./imuFs);

    % Fuse accelerometer and gyroscope readings
    fuseaccel(filt,accel(ii,:),Racc);
    fusegyro(filt,gyro(ii,:),Rgyro);

    % Fuse magnetometer at 1/2 the IMU rate
    if ~mod(ii, fix(imuFs/2))
        fusemag(filt,mag(ii,:),Rmag);
    end

    % Fuse GPS once per second
    if ~mod(ii,imuFs)
        fusegps(filt,lla(gpsIdx,:),Rpos,gpsvel(gpsIdx,:),Rvel);
        gpsIdx = gpsIdx + 1;
    end

    % Log the current pose estimate
    [p(ii,:),q(ii)] = pose(filt);

end
```

Calculate the RMS errors between the known true position and orientation and the output from the asynchronous IMU filter.

```
posErr = truePos - p;
qErr = rad2deg(dist(trueOrient,q));

pRMS = sqrt(mean(posErr.^2));
qRMS = sqrt(mean(qErr.^2));

fprintf('Position RMS Error\n');
```

```
Position RMS Error
```

```
fprintf('\tX: %.2f, Y: %.2f, Z: %.2f (meters)\n\n',pRMS(1),pRMS(2),pRMS(3));
```

```
    X: 0.55, Y: 0.71, Z: 0.74 (meters)
```

```
fprintf('Quaternion Distance RMS Error\n');
```

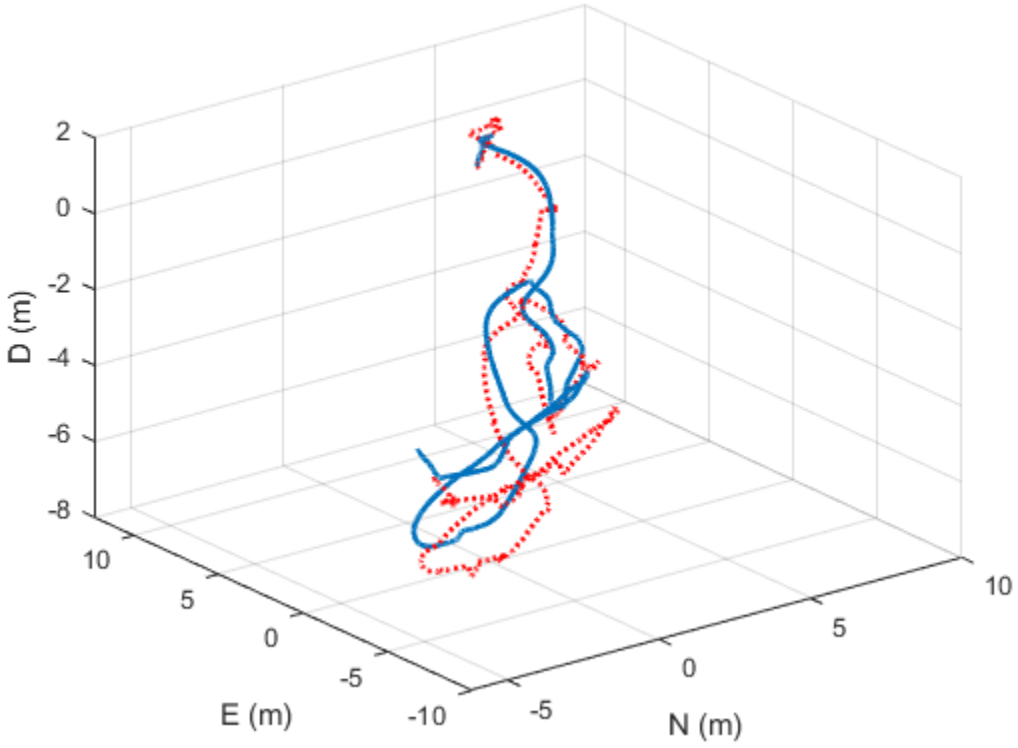
```
Quaternion Distance RMS Error
```

```
fprintf('\t%.2f (degrees)\n\n', qRMS);
```

```
    4.72 (degrees)
```

Visualize the true position and the estimated position.

```
plot3(truePos(:,1),truePos(:,2),truePos(:,3),'LineWidth',2)
hold on
plot3(p(:,1),p(:,2),p(:,3),'r:','LineWidth',2)
grid on
xlabel('N (m)')
ylabel('E (m)')
zlabel('D (m)')
```



Algorithms

Dynamic Model Used in `insfilterAsync`

Note: The following algorithm only applies to an NED reference frame.

`insfilterAsync` implements a 28-axis continuous-discrete extended Kalman filter using sequential fusion. The filter relies on the assumption that individual sensor measurements are uncorrelated. The filter uses an omnidirectional motion model and assumes constant angular velocity and constant acceleration. The state is defined as:

$$x = \begin{bmatrix} q_0 \\ q_1 \\ q_2 \\ q_3 \\ angVel_X \\ angVel_Y \\ angVel_Z \\ position_N \\ position_E \\ position_D \\ \nu_N \\ \nu_E \\ \nu_D \\ accel_N \\ accel_E \\ accel_D \\ accelbias_X \\ accelbias_Y \\ accelbias_Z \\ gyrobias_X \\ gyrobias_Y \\ gyrobias_Z \\ geomagneticFieldVector_N \\ geomagneticFieldVector_E \\ geomagneticFieldVector_D \\ magbias_X \\ magbias_Y \\ magbias_Z \end{bmatrix}$$

where

- q_0, q_1, q_2, q_3 –– Parts of orientation quaternion. The orientation quaternion represents a frame rotation from the platform's current orientation to the local NED coordinate system.
- $angVel_X$, $angVel_Y$, $angVel_Z$ –– Angular velocity relative to the platform's body frame.
- $position_N$, $position_E$, $position_D$ –– Position of the platform in the local NED coordinate system.
- ν_N, ν_E, ν_D –– Velocity of the platform in the local NED coordinate system.
- $accel_N$, $accel_E$, $accel_D$ –– Acceleration of the platform in the local NED coordinate system.
- $accelbias_X$, $accelbias_Y$, $accelbias_Z$ –– Bias in the accelerometer reading.
- $gyrobias_X$, $gyrobias_Y$, $gyrobias_Z$ –– Bias in the gyroscope reading.

- $geomagneticFieldVector_N$, $geomagneticFieldVector_E$, $geomagneticFieldVector_D$ –– Estimate of the geomagnetic field vector at the reference location.
- $magbias_X$, $magbias_Y$, $magbias_Z$ –– Bias in the magnetometer readings.

Given the conventional formation of the process equation, $\dot{x} = f(x) + w$, w is the process noise, $\dot{x}$ is the derivative of x, and:

$$f(x) = \begin{bmatrix} \frac{-(q_1)(angVel_X) - (q_2)(angVel_Y) - (q_3)(angVel_Z)}{2} \\ \frac{(q_0)(angVel_X) - (q_3)(angVel_Y) + (q_1)(angVel_Z)}{2} \\ \frac{(q_3)(angVel_X) + (q_0)(angVel_Y) - (q_1)(angVel_Z)}{2} \\ \frac{(q_1)(angVel_X) - (q_2)(angVel_Y) + (q_0)(angVel_Z)}{2} \\ 0 \\ 0 \\ 0 \\ \nu_N \\ \nu_E \\ \nu_D \\ accel_N \\ accel_E \\ accel_D \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`insfilterErrorState` | `insfilterNonholonomic` | `insfilterMARG`

Introduced in R2019a

correct

Correct states using direct state measurements for `insfilterAsync`

Syntax

```
correct(FUSE,idx,measurement,measurementCovariance)
```

Description

`correct(FUSE,idx,measurement,measurementCovariance)` corrects the state and state estimation error covariance based on the measurement and measurement covariance. The measurement maps directly to the state specified by the indices `idx`.

Input Arguments

`FUSE` — `insfilterAsync` object

object

`insfilterAsync`, specified as an object.

`idx` — State vector index of measurement to correct

N-element vector of increasing integers in the range [1, 28]

State vector index of measurement to correct, specified as an *N*-element vector of increasing integers in the range [1, 28].

The state values represent:

State	Units	Index
Orientation (quaternion parts)	N/A	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)	µT	23:25
Magnetometer Bias (XYZ)	µT	26:28

Data Types: `single` | `double`

`measurement` — Direct measurement of state

N-element vector

Direct measurement of state, specified as an *N*-element vector. *N* is the number of elements of the index argument, `idx`.

Data Types: `single` | `double`

`measurementCovariance` — Covariance of measurement
scalar | *N*-element vector | *N*-by-*N* matrix

Covariance of measurement, specified as a scalar, *N*-element vector, or *N*-by-*N* matrix. *N* is the number of elements of the index argument, `idx`.

Data Types: `single` | `double`

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also
`insfilterAsync` | `insfilter`

Introduced in R2019a

copy

Create copy of `insfilterAsync`

Syntax

```
newFilter = copy(filter)
```

Description

`newFilter = copy(filter)` returns a copy of the `insfilterAsync`, `filter`, that has exactly the same property values.

Input Arguments

`filter` — Filter to be copied

`insfilterAsync`

Filter to be copied, specified as an `insfilterAsync` object.

Output Arguments

`newFilter` — New copied filter

`insfilterAsync`

New copied filter, returned as an `insfilterAsync` object.

See Also

`insfilterAsync`

Introduced in R2020b

fuseaccel

Correct states using accelerometer data for `insfilterAsync`

Syntax

```
[res,resCov] = fuseaccel(FUSE,acceleration,accelerationCovariance)
```

Description

`[res,resCov] = fuseaccel(FUSE,acceleration,accelerationCovariance)` fuses accelerometer data to correct the state estimate.

Input Arguments

`FUSE` — `insfilterAsync` object

object

`insfilterAsync`, specified as an object.

`acceleration` — Accelerometer readings in local sensor body coordinate system (m/s^2)

3-element row vector

Accelerometer readings in local sensor body coordinate system in m/s^2, specified as a 3-element row vector

Data Types: `single` | `double`

`accelerationCovariance` — Acceleration error covariance of accelerometer measurement ($(m/s^2)^2$)

scalar | 3-element row vector | 3-by-3 matrix

Acceleration error covariance of the accelerometer measurement in $(m/s^2)^2$, specified as a scalar, 3-element row vector, or 3-by-3 matrix.

Data Types: `single` | `double`

Output Arguments

`res` — Residual

1-by-3 vector of real values

Residual, returned a 1-by-3 vector of real values in m/s^2.

`resCov` — Residual covariance

3-by-3 matrix of real values

Residual covariance, returned as a 3-by-3 matrix of real values in $(m/s^2)^2$.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`insfilterAsync` | `insfilter`

Introduced in R2019a

fusegps

Correct states using GPS data for `insfilterAsync`

Syntax

```
[res,resCov] = fusegps(FUSE,position,positionCovariance)
[res,resCov] = fusegps(FUSE,position,positionCovariance,velocity,
velocityCovariance)
```

Description

`[res,resCov] = fusegps(FUSE,position,positionCovariance)` fuses GPS position data to correct the state estimate.

`[res,resCov] = fusegps(FUSE,position,positionCovariance,velocity, velocityCovariance)` fuses GPS position and velocity data to correct the state estimate.

Input Arguments

`FUSE` — `insfilterAsync` object
object

`insfilterAsync`, specified as an object.

`position` — Position of GPS receiver (LLA)
3-element row vector

Position of GPS receiver in geodetic latitude, longitude, and altitude (LLA) specified as a real finite 3-element row vector. Latitude and longitude are in degrees with north and east being positive. Altitude is in meters.

Data Types: `single` | `double`

`positionCovariance` — Position measurement covariance of GPS receiver (m^2)
3-by-3 matrix

Position measurement covariance of GPS receiver in m^2, specified as a 3-by-3 matrix.

Data Types: `single` | `double`

`velocity` — Velocity of GPS receiver in local NED coordinate system (m/s)
3-element row vector

Velocity of the GPS receiver in the local NED coordinate system in m/s, specified as a 3-element row vector.

Data Types: `single` | `double`

`velocityCovariance` — Velocity measurement covariance of GPS receiver (m/s^2)
3-by-3 matrix

Velocity measurement covariance of the GPS receiver in the local NED coordinate system in m/s^2, specified as a 3-by-3 matrix.

Data Types: `single` | `double`

Output Arguments

`res` — Position and velocity residual

1-by-6 vector of real values

Position and velocity residual, returned as a 1-by-6 vector of real values in m and m/s, respectively.

`resCov` — Residual covariance

6-by-6 matrix of real values

Residual covariance, returned as a 6-by-6 matrix of real values.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`insfilterAsync` | `insfilter` | `insfilterMARG`

Introduced in R2019a

fusegyro

Correct states using gyroscope data for `insfilterAsync`

Syntax

```
[res,resCov] = fusegyro(FUSE,gyroReadings,gyroCovariance)
```

Description

`[res,resCov] = fusegyro(FUSE,gyroReadings,gyroCovariance)` fuses gyroscope data to correct the state estimate.

Input Arguments

`FUSE` — `insfilterAsync` object
object

`insfilterAsync`, specified as an object.

`gyroReadings` — Gyroscope readings in local sensor body coordinate system (rad/s)
3-element row vector

Gyroscope readings in local sensor body coordinate system in rad/s, specified as a 3-element row vector.

Data Types: `single` | `double`

`gyroCovariance` — Covariance of gyroscope measurement error ($(rad/s)^2$)
scalar | 3-element row vector | 3-by-3 matrix

Covariance of gyroscope measurement error in $(rad/s)^2$, specified as a scalar, 3-element row vector, or 3-by-3 matrix.

Data Types: `single` | `double`

Output Arguments

`res` — Residual
1-by-3 vector of real values

Residual, returned as a 1-by-3 vector of real values in rad/s.

`resCov` — Residual covariance
3-by-3 matrix of real values

Residual covariance, returned as a 3-by-3 matrix of real values in $(rad/s)^2$.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`insfilterAsync` | `insfilter`

Introduced in R2019a

fusemag

Correct states using magnetometer data for `insfilterAsync`

Syntax

```
[res,resCov] = fusemag(FUSE,magReadings,magReadingsCovariance)
```

Description

`[res,resCov] = fusemag(FUSE,magReadings,magReadingsCovariance)` fuses magnetometer data to correct the state estimate.

Input Arguments

`FUSE` — `insfilterAsync` object

object

`insfilterAsync`, specified as an object.

`magReadings` — Magnetometer readings (µT)

3-element row vector

Magnetometer readings in µT, specified as a 3-element row vector of finite real numbers.

Data Types: `single` | `double`

`magReadingsCovariance` — Magnetometer readings error covariance (μT^2)

scalar | 3-element row vector | 3-by-3 matrix

Magnetometer readings error covariance in μT^2, specified as a scalar, 3-element row vector, or 3-by-3 matrix.

Data Types: `single` | `double`

Output Arguments

`res` — Residual

1-by-3 vector of real values

Residual, returned a 1-by-3 vector of real values in µT.

Data Types: `single` | `double`

`resCov` — Residual covariance

3-by-3 matrix of real values

Residual covariance, returned a 3-by-3 matrix of real values in $(\mu T)^2$.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`insfilterAsync` | `insfilter`

Introduced in R2019a

pose

Current position, orientation, and velocity estimate for `insfilterAsync`

Syntax

```
[position,orientation,velocity] = pose(FUSE)
[position,orientation,velocity] = pose(FUSE,format)
```

Description

`[position,orientation,velocity] = pose(FUSE)` returns the current estimate of the pose.

`[position,orientation,velocity] = pose(FUSE,format)` returns the current estimate of the pose with orientation in the specified orientation format.

Input Arguments

`FUSE` — `insfilterAsync` object
object

`insfilterAsync`, specified as an object.

`format` — Output orientation format
`'quaternion'` (default) | `'rotmat'`

Output orientation format, specified as either `'quaternion'` for a `quaternion` or `'rotmat'` for a rotation matrix.

Data Types: `char` | `string`

Output Arguments

`position` — Position estimate expressed in the local coordinate system (m)
3-element row vector

Position estimate expressed in the local coordinate system of the filter in meters, returned as a 3-element row vector.

Data Types: `single` | `double`

`orientation` — Orientation estimate expressed in the local coordinate system
quaternion (default) | 3-by-3 rotation matrix

Orientation estimate expressed in the local coordinate system of the filter, returned as a scalar quaternion or 3-by-3 rotation matrix. The quaternion or rotation matrix represents a frame rotation from the local reference frame of the filter to the body reference frame.

Data Types: `single` | `double` | `quaternion`

`velocity` — Velocity estimate expressed in local coordinate system (m/s)
3-element row vector

Velocity estimate expressed in the local coordinate system of the filter in m/s, returned as a 3-element row vector.

Data Types: `single` | `double`

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`insfilterAsync` | `insfilter`

Introduced in R2019a

predict

Update states based on motion model for `insfilterAsync`

Syntax

```
predict(FUSE,dt)
```

Description

`predict(FUSE,dt)` updates states based on the motion model.

Input Arguments

`FUSE` — `insfilterAsync` object

object

`insfilterAsync`, specified as an object.

`dt` — Delta time to propagate forward (s)

scalar

Delta time to propagate forward in seconds, specified as a positive scalar.

Data Types: `single` | `double`

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`insfilterAsync` | `insfilter`

Introduced in R2019a

reset

Reset internal states for `insfilterAsync`

Syntax

```
reset(FUSE)
```

Description

`reset(FUSE)` resets the `State` and `StateCovariance` properties of the `insfilterAsync` object to their default values.

Input Arguments

`FUSE` — `insfilterAsync` object

object

`insfilterAsync`, specified as an object.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`insfilterAsync` | `insfilter`

Introduced in R2019a

residual

Residuals and residual covariances from direct state measurements for `insfilterAsync`

Syntax

```
[res, resCov]= residual(FUSE,idx,measurement,measurementCovariance)
```

Description

`[res, resCov]= residual(FUSE,idx,measurement,measurementCovariance)` computes the residual, `res`, and the residual covariance, `resCov`, based on the direct state measurement and measurement covariance. The `measurement` maps directly to the states specified by indices, `idx`.

Input Arguments

`FUSE` — `insfilterAsync`

`ahrs10filter` | object

`insfilterAsync`, specified as an object.

`idx` — State vector index of measurement to correct

N-element vector of increasing integers in the range [1, 28]

State vector index of measurement to correct, specified as an *N*-element vector of increasing integers in the range [1, 28].

The state values represent:

State	Units	Index
Orientation (quaternion parts)	N/A	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)	µT	23:25
Magnetometer Bias (XYZ)	µT	26:28

Data Types: `single` | `double`

`measurement` — Direct measurement of state

N-element vector

Direct measurement of state, specified as a *N*-element vector. *N* is the number of elements of the index argument, `idx`.

`measurementCovariance` — Covariance of measurement
N-by-*N* matrix

Covariance of measurement, specified as an *N*-by-*N* matrix. *N* is the number of elements of the index argument, `idx`.

Output Arguments

`res` — Measurement residual
1-by-*N* vector of real values

Measurement residual, returned as a 1-by-*N* vector of real values.

`resCov` — Residual covariance
N-by-*N* matrix of real values

Residual covariance, returned as a *N*-by-*N* matrix of real values.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also
`insfilterAsync`

Introduced in R2020a

residualaccel

Residuals and residual covariance from accelerometer measurements for `insfilterAsync`

Syntax

```
[res,resCov] = residualaccel(FUSE,acceleration,accelerationCovariance)
```

Description

`[res,resCov] = residualaccel(FUSE,acceleration,accelerationCovariance)` computes the residual, `res`, and the residual covariance, `resCov`, based on the acceleration readings and the corresponding covariance.

Input Arguments

`FUSE` — `insfilterAsync`
`ahrs10filter` | object

`insfilterAsync`, specified as an object.

`acceleration` — Accelerometer readings in local sensor body coordinate system (m/s^2)
3-element row vector

Accelerometer readings in local sensor body coordinate system in m/s^2, specified as a 3-element row vector

Data Types: `single` | `double`

`accelerationCovariance` — Acceleration error covariance of accelerometer measurement ($(m/s^2)^2$)
scalar | 3-element row vector | 3-by-3 matrix

Acceleration error covariance of the accelerometer measurement in $(m/s^2)^2$, specified as a scalar, 3-element row vector, or 3-by-3 matrix.

Data Types: `single` | `double`

Output Arguments

`res` — Residual
1-by-3 vector of real values

Residual, returned a 1-by-3 vector of real values in m/s^2.

`resCov` — Residual covariance
3-by-3 matrix of real values

Residual covariance, returned a 3-by-3 matrix of real values in $(m/s^2)^2$.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`insfilterAsync` | `insfilter`

Introduced in R2020a

residualgps

Residuals and residual covariance from GPS measurements for `insfilterAsync`

Syntax

```
[res,resCov] = residualgps(FUSE,position,positionCovariance)
[res,resCov] = residualgps(FUSE,position,positionCovariance,velocity,
velocityCovariance)
```

Description

`[res,resCov] = residualgps(FUSE,position,positionCovariance)` computes the residual, `res`, and the residual covariance, `resCov`, based on the GPS position measurement and covariance.

`[res,resCov] = residualgps(FUSE,position,positionCovariance,velocity,velocityCovariance)` computes the residual, `res`, and the residual covariance, `resCov`, based on the GPS position measurement and covariance.

Input Arguments

`FUSE` — `insfilterAsync`
`ahrs10filter` | object

`insfilterAsync`, specified as an object.

`position` — Position of GPS receiver (LLA)
3-element row vector

Position of GPS receiver in geodetic latitude, longitude, and altitude (LLA) specified as a real finite 3-element row vector. Latitude and longitude are in degrees with north and east being positive. Altitude is in meters.

Data Types: `single` | `double`

`positionCovariance` — Position measurement covariance of GPS receiver (m^2)
3-by-3 matrix

Position measurement covariance of GPS receiver in m^2, specified as a 3-by-3 matrix.

Data Types: `single` | `double`

`velocity` — Velocity of GPS receiver in local NED coordinate system (m/s)
3-element row vector

Velocity of the GPS receiver in the local NED coordinate system in m/s, specified as a 3-element row vector.

Data Types: `single` | `double`

`velocityCovariance` — Velocity measurement covariance of GPS receiver (m/s^2)
3-by-3 matrix

Velocity measurement covariance of the GPS receiver in the local NED coordinate system in m/s^2, specified as a 3-by-3 matrix.

Data Types: `single` | `double`

Output Arguments

`res` — Position and velocity residual

1-by-3 vector of real values | 1-by-6 vector of real values

Position and velocity residual, returned as a 1-by-3 vector of real values if the inputs only contain position information, and returned as 1-by-6 vector of real values if the inputs also contain velocity information.

`resCov` — Residual covariance

3-by-3 matrix of real values | 6-by-6 matrix of real values

Residual covariance, returned as a 3-by-3 matrix of real values if the inputs only contain position information, and a 6-by-6 matrix of real values if the inputs also contain velocity information.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`insfilterAsync`

Introduced in R2020a

residualgyro

Residuals and residual covariance from gyroscope measurements for `insfilterAsync`

Syntax

```
[res,resCov] = residualgyro(FUSE,gyroReadings,gyroCovariance)
```

Description

`[res,resCov] = residualgyro(FUSE,gyroReadings,gyroCovariance)` computes the residual, `res`, and the innovation covariance, `resCov`, based on the gyroscope readings and the corresponding covariance.

Input Arguments

`FUSE` — `insfilterAsync`
`ahrs10filter` | object

`insfilterAsync`, specified as an object.

`gyroReadings` — Gyroscope readings in local sensor body coordinate system (rad/s)
3-element row vector

Gyroscope readings in local sensor body coordinate system in rad/s, specified as a 3-element row vector.

Data Types: `single` | `double`

`gyroCovariance` — Covariance of gyroscope measurement error ($(rad/s)^2$)
scalar | 3-element row vector | 3-by-3 matrix

Covariance of gyroscope measurement error in $(rad/s)^2$, specified as a scalar, 3-element row vector, or 3-by-3 matrix.

Data Types: `single` | `double`

Output Arguments

`res` — Residual
1-by-3 vector of real values

Residual, returned as a 1-by-3 vector of real values in rad/s.

`resCov` — Residual covariance
3-by-3 matrix of real values

Residual covariance, returned as a 3-by-3 matrix of real values in $(rad/s)^2$.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`insfilterAsync` | `insfilter`

Introduced in R2020a

residualmag

Residuals and residual covariance from magnetometer measurements for `insfilterAsync`

Syntax

```
[res,resCov] = residualmag(FUSE,magReadings,magReadingsCovariance)
```

Description

`[res,resCov] = residualmag(FUSE,magReadings,magReadingsCovariance)` computes the residual, `residual`, and the residual covariance, `resCov`, based on the magnetometer readings and the corresponding covariance.

Input Arguments

`FUSE` — `insfilterAsync`

`ahrs10filter` | object

`insfilterAsync`, specified as an object.

`magReadings` — Magnetometer readings (µT)

3-element row vector

Magnetometer readings in µT, specified as a 3-element row vector of finite real numbers.

Data Types: `single` | `double`

`magReadingsCovariance` — Magnetometer readings error covariance (μT^2)

scalar | 3-element row vector | 3-by-3 matrix

Magnetometer readings error covariance in μT^2, specified as a scalar, 3-element row vector, or 3-by-3 matrix.

Data Types: `single` | `double`

Output Arguments

`res` — Residual

1-by-3 vector of real values

Residual, returned as a 1-by-3 vector of real values in µT.

Data Types: `single` | `double`

`resCov` — Residual covariance

3-by-3 matrix of real values

Residual covariance, returned a 3-by-3 matrix of real values in $(\mu T)^2$.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`insfilterAsync`

Introduced in R2020a

stateinfo

Display state vector information for `insfilterAsync`

Syntax

```
stateinfo(FUSE)
info = stateinfo(FUSE)
```

Description

`stateinfo(FUSE)` displays the description of each index of the `State` property of the `insfilterAsync` object and the associated units.

`info = stateinfo(FUSE)` returns a structure with fields containing descriptions of the elements of the state vector of the filter, `FUSE`.

Examples

State Information of `insfilterAsync`

Create an `insfilterAsync` object.

```
filter = insfilterAsync;
```

Display the state information of the created filter.

```
stateinfo(filter)
```

```
States                           Units    Index
Orientation (quaternion parts)            1:4
Angular Velocity (XYZ)           rad/s    5:7
Position (NAV)                   m        8:10
Velocity (NAV)                   m/s      11:13
Acceleration (NAV)               m/s^2    14:16
Accelerometer Bias (XYZ)         m/s^2    17:19
Gyroscope Bias (XYZ)             rad/s    20:22
Geomagnetic Field Vector (NAV)   µT       23:25
Magnetometer Bias (XYZ)          µT       26:28
```

Output the state information of the filter as a structure.

```
info = stateinfo(filter)
```

```
info = struct with fields:
               Orientation: [1 2 3 4]
           AngularVelocity: [5 6 7]
                  Position: [8 9 10]
                  Velocity: [11 12 13]
              Acceleration: [14 15 16]
         AccelerometerBias: [17 18 19]
             GyroscopeBias: [20 21 22]
```

```
    GeomagneticFieldVector: [23 24 25]
          MagnetometerBias: [26 27 28]
```

Input Arguments

`FUSE` — `insfilterAsync` object

object

`insfilterAsync`, specified as an object.

Output Arguments

`info` — State information

structure

State information, returned as a structure with fields containing descriptions of the elements of the state vector of the filter. The values of each field are the corresponding indices of the state vector.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`insfilterAsync` | `insfilter`

Introduced in R2019a

tune

Tune `insfilterAsync` parameters to reduce estimation error

Syntax

```
tunedMeasureNoise = tune(filter,measureNoise,sensorData,groundTruth)
tunedMeasureNoise = tune( ___ ,config)
```

Description

`tunedMeasureNoise = tune(filter,measureNoise,sensorData,groundTruth)` adjusts the properties of the `insfilterAsync` filter object, `filter`, and measurement noises to reduce the root-mean-squared (RMS) state estimation error between the fused sensor data and the ground truth. The function also returns the tuned measurement noise, `tunedMeasureNoise`. The function uses the property values in the filter and the measurement noise provided in the `measureNoise` structure as the initial estimate for the optimization algorithm.

`tunedMeasureNoise = tune( ___ ,config)` specifies the tuning configuration based on a `tunerconfig` object, `config`.

Examples

Tune `insfilterAsync` to Optimize Pose Estimate

Load the recorded sensor data and ground truth data.

```
load('insfilterAsyncTuneData.mat');
```

Create timetables for the sensor data and the truth data.

```
sensorData = timetable(Accelerometer, Gyroscope, ...
    Magnetometer, GPSPosition, GPSVelocity, 'SampleRate', 100);
groundTruth = timetable(Orientation, Position, ...
    'SampleRate', 100);
```

Create an `insfilterAsync` filter object that has a few noise properties.

```
filter = insfilterAsync('State', initialState, ...
    'StateCovariance', initialStateCovariance, ...
    'AccelerometerBiasNoise', 1e-7, ...
    'GyroscopeBiasNoise', 1e-7, ...
    'MagnetometerBiasNoise', 1e-7, ...
    'GeomagneticVectorNoise', 1e-7);
```

Create a tuner configuration object for the filter. Set the maximum iterations to two. Also, set the tunable parameters as the unspecified properties.

```
config = tunerconfig('insfilterAsync','MaxIterations',8);
config.TunableParameters = setdiff(config.TunableParameters, ...
    {'GeomagneticVectorNoise', 'AccelerometerBiasNoise', ...
```

```
    'GyroscopeBiasNoise', 'MagnetometerBiasNoise'});
config.TunableParameters
```

```
ans = 1×10 string
    "AccelerationNoise"    "AccelerometerNoise"    "AngularVelocityNoise"    "GPSPositionNoise"
```

Use the tuner noise function to obtain a set of initial sensor noises used in the filter.

```
measNoise = tunernoise('insfilterAsync')
```

```
measNoise = struct with fields:
    AccelerometerNoise: 1
        GyroscopeNoise: 1
     MagnetometerNoise: 1
      GPSPositionNoise: 1
      GPSVelocityNoise: 1
```

Tune the filter and obtain the tuned parameters.

```
tunedParams = tune(filter,measNoise,sensorData,groundTruth,config);
```

```
    Iteration    Parameter              Metric
    _________    _________              ______
    1            AccelerationNoise      2.1345
    1            AccelerometerNoise     2.1264
    1            AngularVelocityNoise   1.9659
    1            GPSPositionNoise       1.9341
    1            GPSVelocityNoise       1.8420
    1            GyroscopeNoise         1.7589
    1            MagnetometerNoise      1.7362
    1            PositionNoise          1.7362
    1            QuaternionNoise        1.7218
    1            VelocityNoise          1.7218
    2            AccelerationNoise      1.7190
    2            AccelerometerNoise     1.7170
    2            AngularVelocityNoise   1.6045
    2            GPSPositionNoise       1.5948
    2            GPSVelocityNoise       1.5323
    2            GyroscopeNoise         1.4803
    2            MagnetometerNoise      1.4703
    2            PositionNoise          1.4703
    2            QuaternionNoise        1.4632
    2            VelocityNoise          1.4632
    3            AccelerationNoise      1.4596
    3            AccelerometerNoise     1.4548
    3            AngularVelocityNoise   1.3923
    3            GPSPositionNoise       1.3810
    3            GPSVelocityNoise       1.3322
    3            GyroscopeNoise         1.2998
    3            MagnetometerNoise      1.2976
    3            PositionNoise          1.2976
    3            QuaternionNoise        1.2943
    3            VelocityNoise          1.2943
    4            AccelerationNoise      1.2906
    4            AccelerometerNoise     1.2836
    4            AngularVelocityNoise   1.2491
    4            GPSPositionNoise       1.2258
```

```
    4            GPSVelocityNoise        1.1880
    4            GyroscopeNoise          1.1701
    4            MagnetometerNoise       1.1698
    4            PositionNoise           1.1698
    4            QuaternionNoise         1.1688
    4            VelocityNoise           1.1688
    5            AccelerationNoise       1.1650
    5            AccelerometerNoise      1.1569
    5            AngularVelocityNoise    1.1454
    5            GPSPositionNoise        1.1100
    5            GPSVelocityNoise        1.0778
    5            GyroscopeNoise          1.0709
    5            MagnetometerNoise       1.0675
    5            PositionNoise           1.0675
    5            QuaternionNoise         1.0669
    5            VelocityNoise           1.0669
    6            AccelerationNoise       1.0634
    6            AccelerometerNoise      1.0549
    6            AngularVelocityNoise    1.0549
    6            GPSPositionNoise        1.0180
    6            GPSVelocityNoise        0.9866
    6            GyroscopeNoise          0.9810
    6            MagnetometerNoise       0.9775
    6            PositionNoise           0.9775
    6            QuaternionNoise         0.9768
    6            VelocityNoise           0.9768
    7            AccelerationNoise       0.9735
    7            AccelerometerNoise      0.9652
    7            AngularVelocityNoise    0.9652
    7            GPSPositionNoise        0.9283
    7            GPSVelocityNoise        0.8997
    7            GyroscopeNoise          0.8947
    7            MagnetometerNoise       0.8920
    7            PositionNoise           0.8920
    7            QuaternionNoise         0.8912
    7            VelocityNoise           0.8912
    8            AccelerationNoise       0.8885
    8            AccelerometerNoise      0.8811
    8            AngularVelocityNoise    0.8807
    8            GPSPositionNoise        0.8479
    8            GPSVelocityNoise        0.8238
    8            GyroscopeNoise          0.8165
    8            MagnetometerNoise       0.8165
    8            PositionNoise           0.8165
    8            QuaternionNoise         0.8159
    8            VelocityNoise           0.8159
```

Fuse the sensor data using the tuned filter.

```
dt = seconds(diff(groundTruth.Time));
N = size(sensorData,1);
qEst = quaternion.zeros(N,1);
posEst = zeros(N,3);
% Iterate the filter for prediction and correction using sensor data.
for ii=1:N
    if ii ~= 1
        predict(filter, dt(ii-1));
    end
```

```
    if all(~isnan(Accelerometer(ii,:)))
        fuseaccel(filter,Accelerometer(ii,:), ...
            tunedParams.AccelerometerNoise);
    end
    if all(~isnan(Gyroscope(ii,:)))
        fusegyro(filter, Gyroscope(ii,:), ...
            tunedParams.GyroscopeNoise);
    end
    if all(~isnan(Magnetometer(ii,1)))
        fusemag(filter, Magnetometer(ii,:), ...
            tunedParams.MagnetometerNoise);
    end
    if all(~isnan(GPSPosition(ii,1)))
        fusegps(filter, GPSPosition(ii,:), ...
            tunedParams.GPSPositionNoise, GPSVelocity(ii,:), ...
            tunedParams.GPSVelocityNoise);
    end
    [posEst(ii,:), qEst(ii,:)] = pose(filter);
end
```

Compute the RMS errors.

```
orientationError = rad2deg(dist(qEst, Orientation));
rmsorientationError = sqrt(mean(orientationError.^2))
```

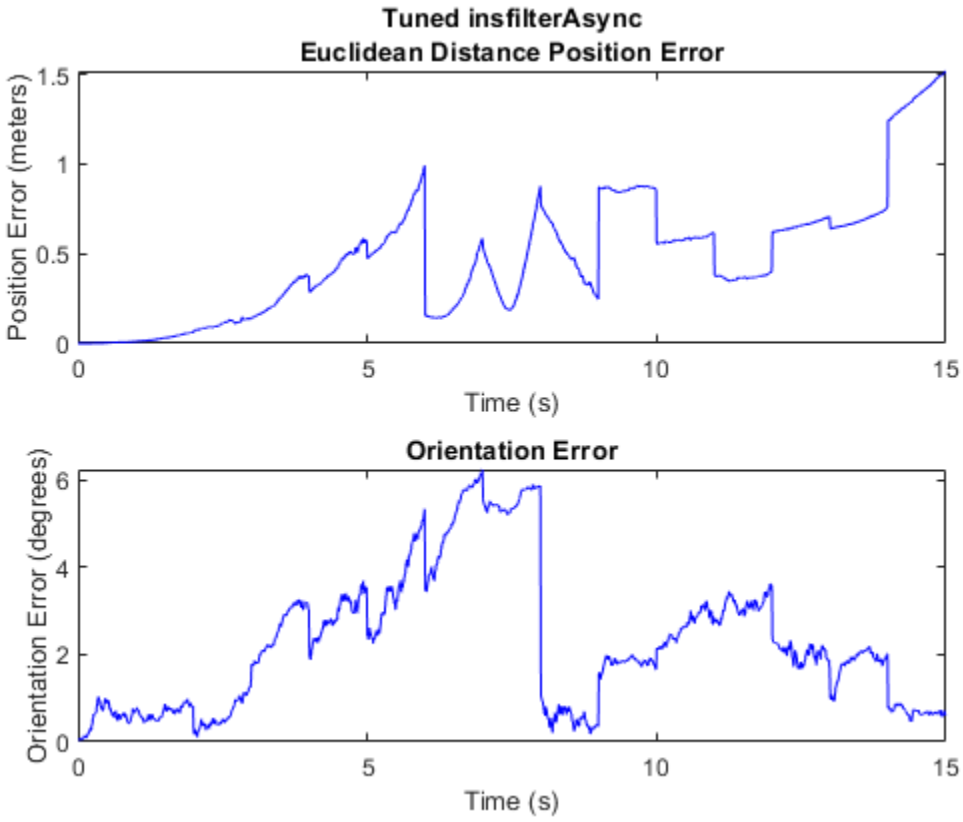
```
rmsorientationError = 2.7801
```

```
positionError = sqrt(sum((posEst - Position).^2, 2));
rmspositionError = sqrt(mean( positionError.^2))
```

```
rmspositionError = 0.5966
```

Visualize the results.

```
figure();
t = (0:N-1)./ groundTruth.Properties.SampleRate;
subplot(2,1,1)
plot(t, positionError, 'b');
title("Tuned insfilterAsync" + newline + "Euclidean Distance Position Error")
xlabel('Time (s)');
ylabel('Position Error (meters)')
subplot(2,1,2)
plot(t, orientationError, 'b');
title("Orientation Error")
xlabel('Time (s)');
ylabel('Orientation Error (degrees)');
```



Input Arguments

`filter` — Filter object

`infilterAsync` object

Filter object, specified as an `insfilterAsync` object.

`measureNoise` — Measurement noise

structure

Measurement noise, specified as a structure. The function uses the measurement noise input as the initial guess for tuning the measurement noise. The structure must contain these fields:

Field name	Description
`AccelerometerNoise`	Variance of accelerometer noise, specified as a scalar in (m^2/s)
`GyroscopeNoise`	Variance of gyroscope noise, specified as a scalar in $(rad/s)^2$
`MagnetometerNoise`	Variance of magnetometer noise, specified as a scalar in $(\mu T)^2$
`GPSPositionNoise`	Variance of GPS position noise, specified as a scalar in m^2

Field name	Description
`GPSVelocityNoise`	Variance of GPS velocity noise, specified as a scalar in $(m/s)^2$

`sensorData` — Sensor data
`duration`

Sensor data, specified as a `timetable`. In each row, the time and sensor data is specified as:

- `Time` — Time at which the data is obtained, specified as a scalar in seconds.
- `Accelerometer` — Accelerometer data, specified as a 1-by-3 vector of scalars in m^2/s.
- `Gyroscope` — Gyroscope data, specified as a 1-by-3 vector of scalars in rad/s.
- `Magnetometer` — Magnetometer data, specified as a 1-by-3 vector of scalars in µT.
- `GPSPosition` — GPS position data, specified as a 1-by-3 vector of scalars in meters.
- `GPSVelocity` — GPS velocity data, specified as a 1-by-3 vector of scalars in m/s.

If a sensor does not produce measurements, specify the corresponding entry as `NaN`. If you set the `Cost` property of the tuner configuration input, `config`, to `Custom`, then you can use other data types for the `sensorData` input based on your choice.

`groundTruth` — Ground truth data
`duration`

Ground truth data, specified as a `timetable`. In each row, the table can optionally contain any of these variables:

- `Orientation` — Orientation from the navigation frame to the body frame, specified as a `quaternion` or a 3-by-3 rotation matrix.
- `AngularVelocity` — Angular velocity in body frame, specified as a 1-by-3 vector of scalars in rad/s.
- `Position` — Position in navigation frame, specified as a 1-by-3 vector of scalars in meters.
- `Velocity` — Velocity in navigation frame, specified as a 1-by-3 vector of scalars in m/s.
- `Acceleration` — Acceleration in navigation frame, specified as a 1-by-3 vector of scalars in m^2/s.
- `AccelerometerBias` — Accelerometer delta angle bias in body frame, specified as a 1-by-3 vector of scalars in m^2/s.
- `GyroscopeBias` — Gyroscope delta angle bias in body frame, specified as a 1-by-3 vector of scalars in rad/s.
- `GeomagneticFieldVector` — Geomagnetic field vector in navigation frame, specified as a 1-by-3 vector of scalars.
- `MagnetometerBias` — Magnetometer bias in body frame, specified as a 1-by-3 vector of scalars in µT.

The function processes each row of the `sensorData` and `groundTruth` tables sequentially to calculate the state estimate and RMS error from the ground truth. State variables not present in `groundTruth` input are ignored for the comparison. The `sensorData` and the `groundTruth` tables must have the same time steps.

If you set the `Cost` property of the tuner configuration input, `config`, to `Custom`, then you can use other data types for the `groundTruth` input based on your choice.

`config` — Tuner configuration
`tunerconfig` object

Tuner configuration, specified as a `tunerconfig` object.

Output Arguments

`tunedMeasureNoise` — Tuned measurement noise
structure

Tuned measurement noise, returned as a structure. The structure contains these fields.

Field name	Description
`AccelerometerNoise`	Variance of accelerometer noise, specified as a scalar in $(m^2/s)^2$
`GyroscopeNoise`	Variance of gyroscope noise, specified as a scalar in $(rad/s)^2$
`MagnetometerNoise`	Variance of magnetometer noise, specified as a scalar in $(\mu T)^2$
`GPSPositionNoise`	Variance of GPS position noise, specified as a scalar in m^2
`GPSVelocityNoise`	Variance of GPS velocity noise, specified as a scalar in $(m/s)^2$

References

[1] Abbeel, P., Coates, A., Montemerlo, M., Ng, A.Y. and Thrun, S. Discriminative Training of Kalman Filters. In *Robotics: Science and systems*, Vol. 2, pp. 1, 2005.

See Also
`tunerconfig` | `tunernoise`

Introduced in R2020b

binaryOccupancyMap

Create occupancy grid with binary values

Description

The `binaryOccupancyMap` creates a 2-D occupancy map object, which you can use to represent and visualize a robot workspace, including obstacles. The integration of sensor data and position estimates create a spatial representation of the approximate locations of the obstacles.

Occupancy grids are used in robotics algorithms such as path planning. They are also used in mapping applications, such as for finding collision-free paths, performing collision avoidance, and calculating localization. You can modify your occupancy grid to fit your specific application.

Each cell in the occupancy grid has a value representing the occupancy status of that cell. An occupied location is represented as `true (1)` and a free location is represented as `false (0)`.

The object keeps track of three reference frames: world, local, and, grid. The world frame origin is defined by `GridLocationInWorld`, which defines the bottom-left corner of the map relative to the world frame. The `LocalOriginInWorld` property specifies the location of the origin of the local frame relative to the world frame. The first grid location with index `(1,1)` begins in the top-left corner of the grid.

Note This object was previously named `robotics.BinaryOccupancyGrid`.

Creation

Syntax

```
map = binaryOccupancyMap
map = binaryOccupancyMap(width,height)
map = binaryOccupancyMap(width,height,resolution)

map = binaryOccupancyMap(rows,cols,resolution,"grid")

map = binaryOccupancyMap(p)
map = binaryOccupancyMap(p,resolution)
map = binaryOccupancyMap(p,resolution)

map = binaryOccupancyMap(sourcemap)
map = binaryOccupancyMap(sourcemap,resolution)
```

Description

`map = binaryOccupancyMap` creates a 2-D binary occupancy grid with a width and height of 10m. The default grid resolution is one cell per meter.

`map = binaryOccupancyMap(width,height)` creates a 2-D binary occupancy grid representing a work space of `width` and `height` in meters. The default grid resolution is one cell per meter.

`map = binaryOccupancyMap(width,height,resolution)` creates a grid with the `Resolution` property specified in cells per meter. The map is in world coordinates by default.

`map = binaryOccupancyMap(rows,cols,resolution,"grid")` creates a 2-D binary occupancy grid of size (`rows`,`cols`).

`map = binaryOccupancyMap(p)` creates a grid from the values in matrix `p`. The size of the grid matches the size of the matrix, with each cell value interpreted from its location in the matrix. `p` contains any numeric or logical type with zeros (0) and ones (1).

`map = binaryOccupancyMap(p,resolution)` creates a map from a matrix with the `Resolution` property specified in cells per meter.

`map = binaryOccupancyMap(p,resolution)` creates an object with the `Resolution` property specified in cells per meter.

`map = binaryOccupancyMap(sourcemap)` creates an object using values from another `binaryOccupancyMap` object.

`map = binaryOccupancyMap(sourcemap,resolution)` creates an object using values from another `binaryOccupancyMap` object, but resamples the matrix to have the specified resolution.

Input Arguments

`width` — Map width
positive scalar

Map width, specified as a positive scalar in meters.

`height` — Map height
positive scalar

Map height, specified as a positive scalar in meters.

`p` — Map grid values
matrix

Map grid values, specified as a matrix.

`sourcemap` — Occupancy map object
`binaryOccupancyMap` object

Occupancy map object, specified as a `binaryOccupancyMap` object.

Properties

`GridSize` — Number of rows and columns in grid
two-element horizontal vector

This property is read-only.

Number of rows and columns in grid, stored as a two-element horizontal vector of the form `[rows cols]`.

`Resolution` — Grid resolution
1 (default) | scalar in cells per meter

This property is read-only.

Grid resolution, stored as a scalar in cells per meter.

`XLocalLimits` — Minimum and maximum values of *x*-coordinates in local frame
two-element vector

This property is read-only.

Minimum and maximum values of *x*-coordinates in local frame, stored as a two-element horizontal vector of the form `[min max]`. Local frame is defined by `LocalOriginInWorld` property.

`YLocalLimits` — Minimum and maximum values of *y*-coordinates in local frame
two-element vector

This property is read-only.

Minimum and maximum values of *y*-coordinates in local frame, stored as a two-element horizontal vector of the form `[min max]`. Local frame is defined by `LocalOriginInWorld` property.

`XWorldLimits` — Minimum and maximum values of *x*-coordinates in world frame
two-element vector

This property is read-only.

Minimum and maximum values of *x*-coordinates in world frame, stored as a two-element horizontal vector of the form `[min max]`. These values indicate the world range of the *x*-coordinates in the grid.

`YWorldLimits` — Minimum and maximum values of *y*-coordinates
two-element vector

This property is read-only.

Minimum and maximum values of *y*-coordinates, stored as a two-element vector of the form `[min max]`. These values indicate the world range of the *y*-coordinates in the grid.

`GridLocationInWorld` — Location of the grid in world coordinates
[0 0] (default) | two-element vector | `[xGrid yGrid]`

Location of the bottom-left corner of the grid in world coordinates, specified as a two-element vector, `[xGrid yGrid]`.

`LocalOriginInWorld` — Location of the local frame in world coordinates
[0 0] (default) | two-element vector | `[xWorld yWorld]`

Location of the origin of the local frame in world coordinates, specified as a two-element vector, `[xLocal yLocal]`. Use the `move` function to shift the local frame as your vehicle moves.

`GridOriginInLocal` — Location of the grid in local coordinates
[0 0] (default) | two-element vector | `[xLocal yLocal]`

Location of the bottom-left corner of the grid in local coordinates, specified as a two-element vector, `[xLocal yLocal]`.

`DefaultValue` — Default value for unspecified map locations
`0` (default) | `1`

Default value for unspecified map locations including areas outside the map, specified as `0` or `1`.

Object Functions

checkOccupancy	Check occupancy values for locations
getOccupancy	Get occupancy value of locations
grid2local	Convert grid indices to local coordinates
grid2world	Convert grid indices to world coordinates
inflate	Inflate each occupied grid location
insertRay	Insert ray from laser scan observation
local2grid	Convert local coordinates to grid indices
local2world	Convert local coordinates to world coordinates
move	Move map in world frame
occupancyMatrix	Convert occupancy grid to matrix
raycast	Compute cell indices along a ray
rayIntersection	Find intersection points of rays and occupied map cells
setOccupancy	Set occupancy value of locations
show	Show occupancy grid values
syncWith	Sync map with overlapping map
world2grid	Convert world coordinates to grid indices
world2local	Convert world coordinates to local coordinates

Examples

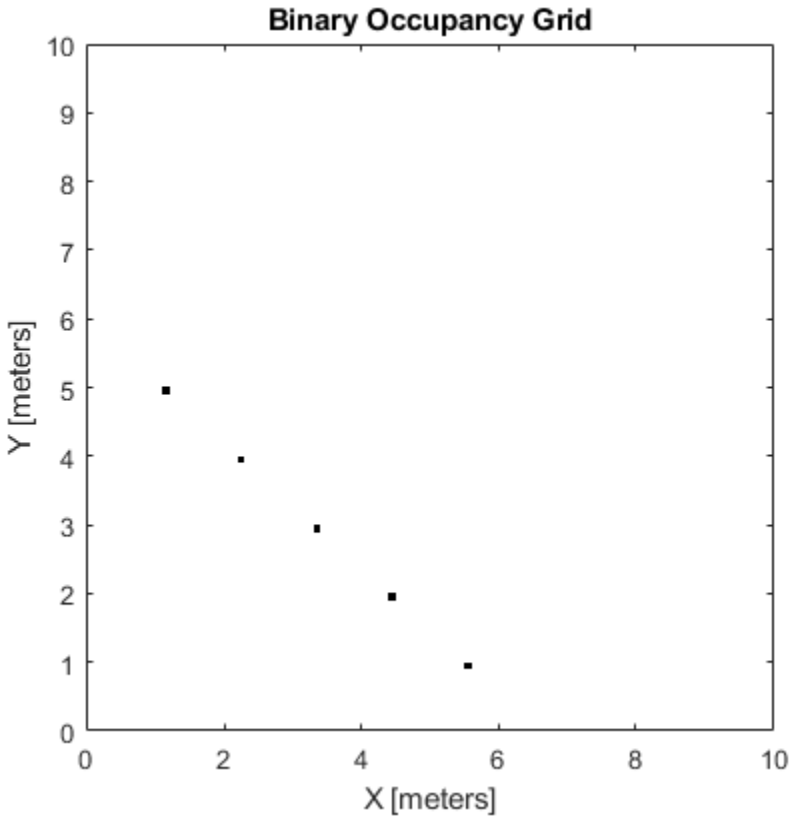
Create and Modify Binary Occupancy Grid

Create a 10m x 10m empty map.

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

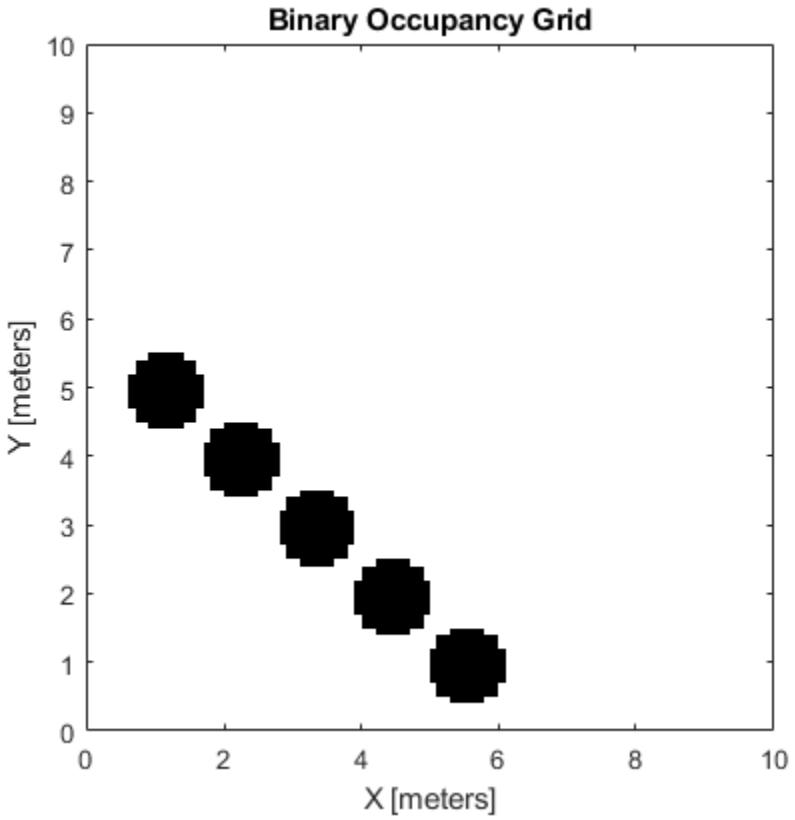
Set occupancy of world locations and show map.

```
x = [1.2; 2.3; 3.4; 4.5; 5.6];
y = [5.0; 4.0; 3.0; 2.0; 1.0];

setOccupancy(map, [x y], ones(5,1))
figure
show(map)
```



Inflate occupied locations by a given radius.

```
inflate(map, 0.5)
figure
show(map)
```



Get grid locations from world locations.

```
ij = world2grid(map, [x y]);
```

Set grid locations to free locations.

```
setOccupancy(map, ij, zeros(5,1), 'grid')
figure
show(map)
```

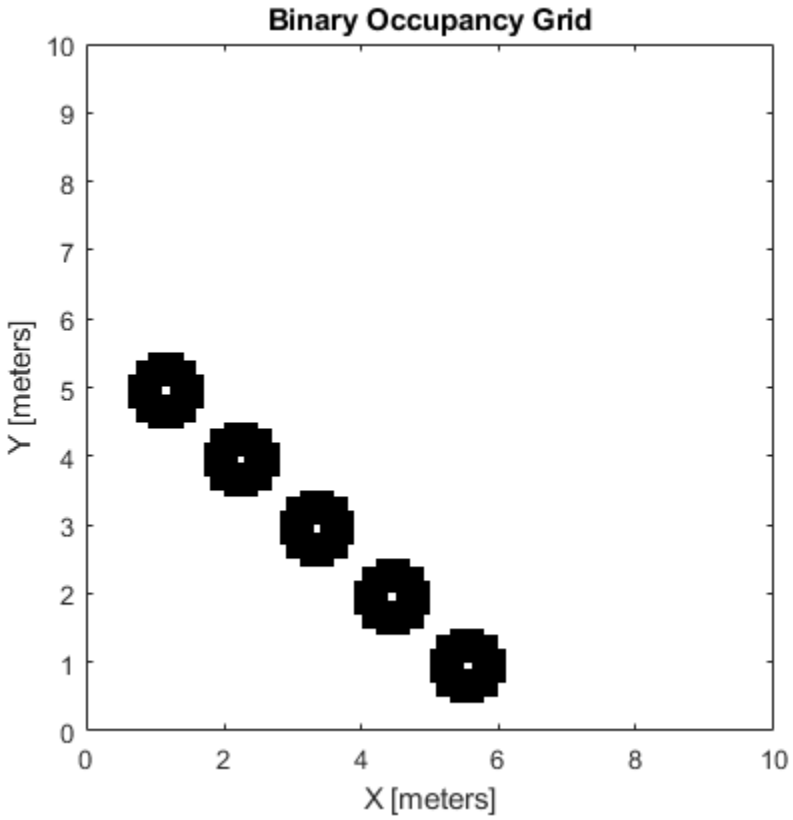


Image to Binary Occupancy Grid Example

This example shows how to convert an image to a binary occupancy grid for using with mapping and path planning.

Import image.

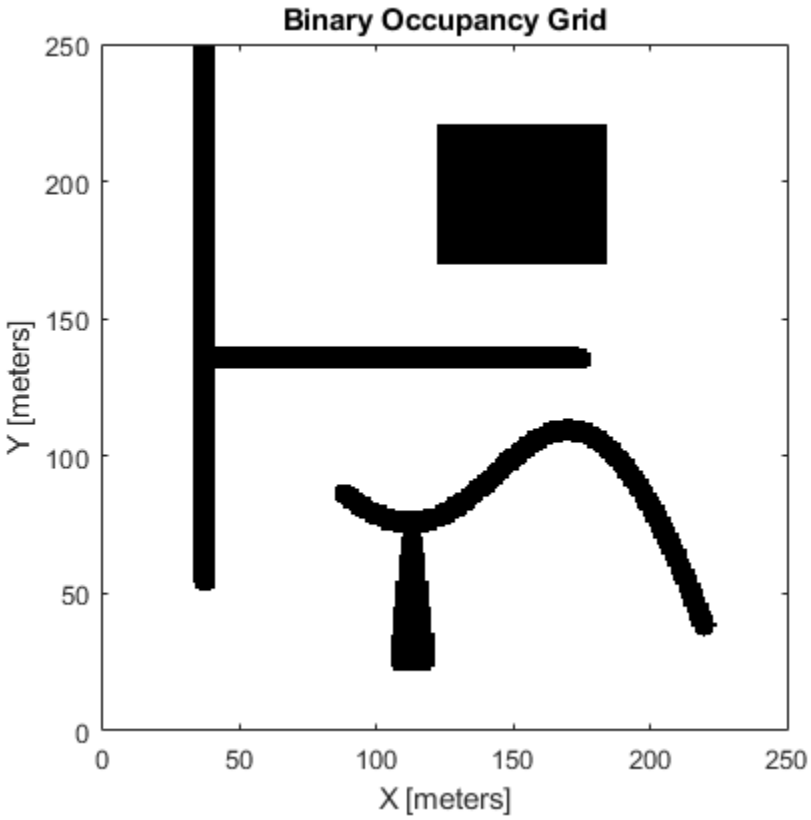
```
image = imread('imageMap.png');
```

Convert to grayscale and then black and white image based on given threshold value.

```
grayimage = rgb2gray(image);
bwimage = grayimage < 0.5;
```

Use black and white image as matrix input for binary occupancy grid.

```
grid = binaryOccupancyMap(bwimage);

show(grid)
```

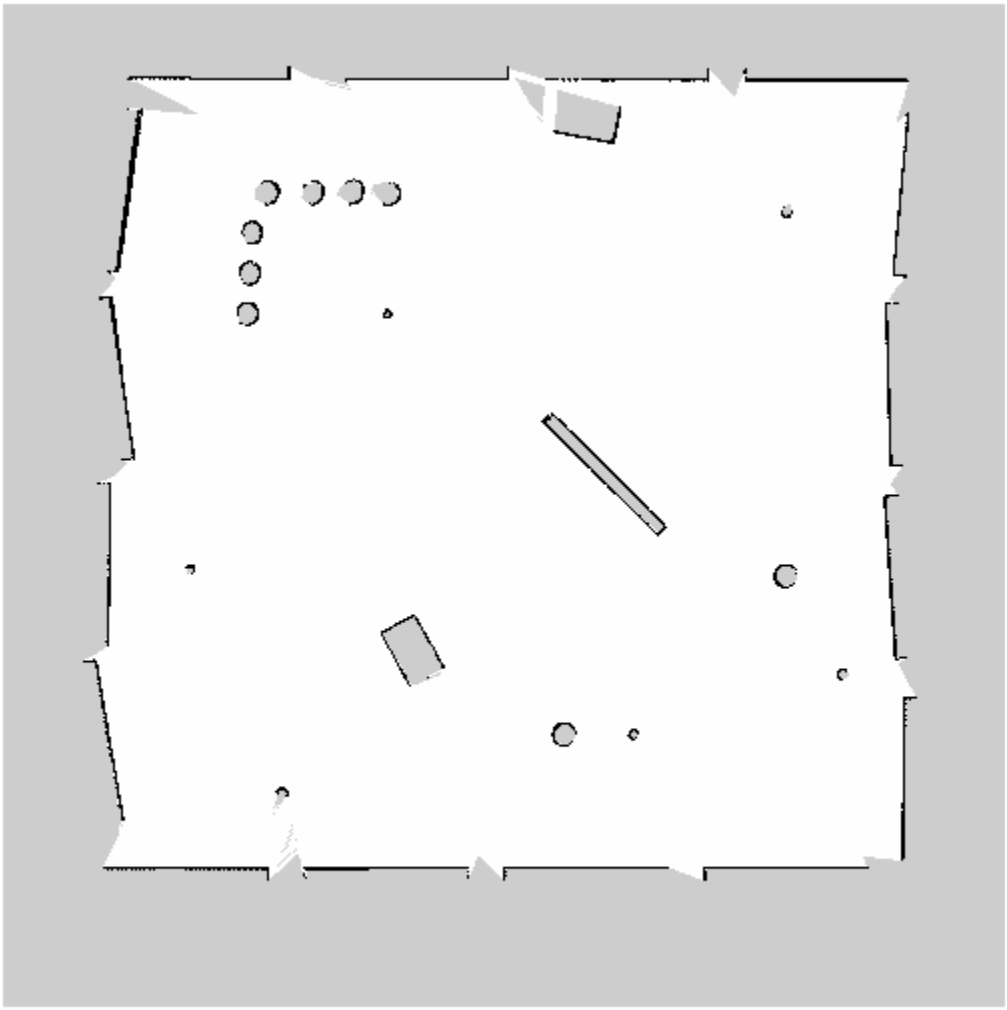


Convert PGM Image to Map

This example shows how to convert a `.pgm` file into a `binaryOccupancyMap` object for use in MATLAB.

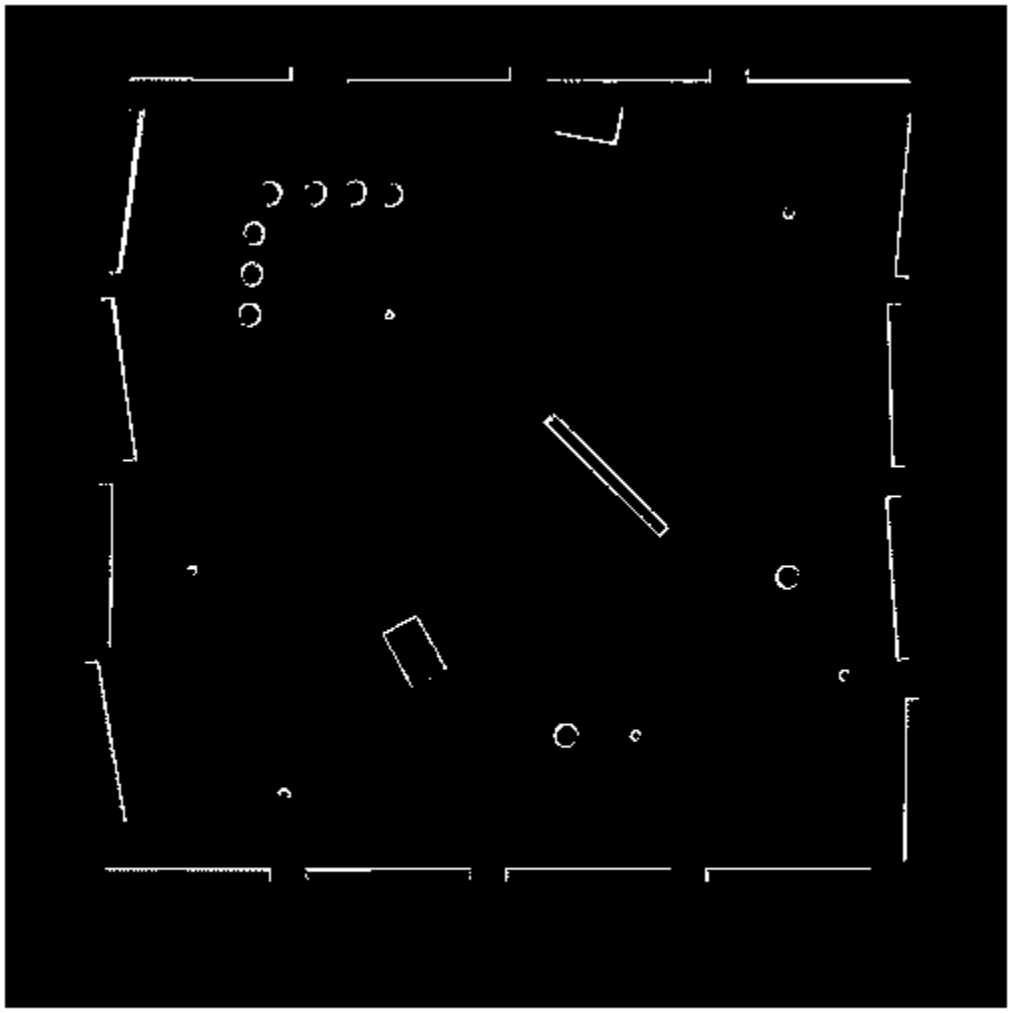
Import image using `imread`. The image is quite large and should be cropped to the relevant area.

```
image = imread('playpen_map.pgm');
imageCropped = image(750:1250,750:1250);
imshow(imageCropped)
```



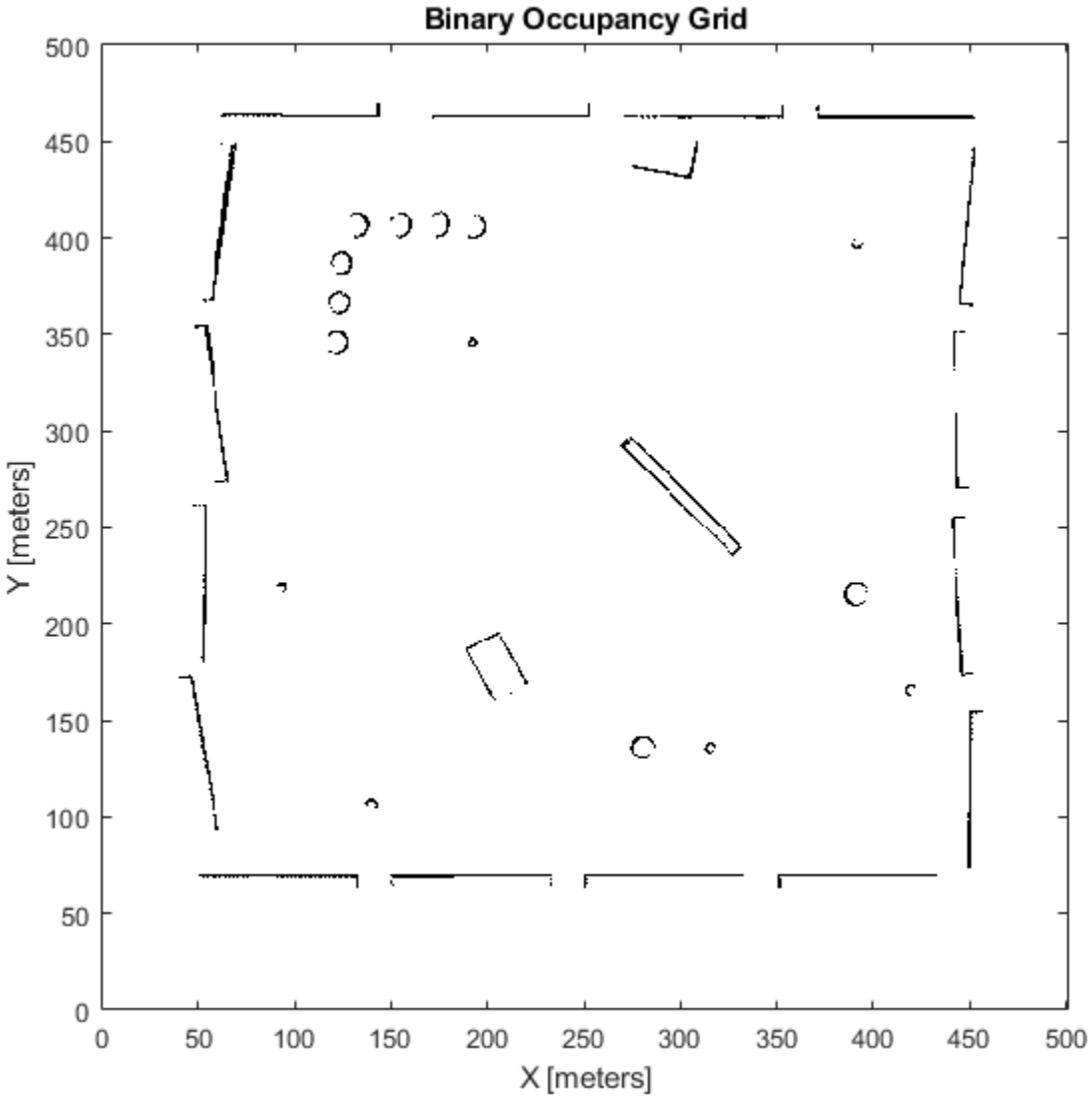
Unknown areas (gray) should be removed and treated as free space. Create a logical matrix based on a threshold. Depending on your image, this value could be different. Occupied space should be set as 1 (white in image).

```
imageBW = imageCropped < 100;
imshow(imageBW)
```



Create `binaryOccupancyMap` object using adjusted map image.

```
map = binaryOccupancyMap(imageBW);
show(map)
```



Compatibility Considerations

`binaryOccupancyMap` was renamed
Behavior change in future release

The `binaryOccupancyMap` object was renamed from `robotics.BinaryOccupancyGrid`. Use `binaryOccupancyMap` for all object creation.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

`controllerPurePursuit`

Topics

“Occupancy Grids”

Introduced in R2015a

checkOccupancy

Check occupancy values for locations

Syntax

```
occVal = checkOccupancy(map,xy)
occVal = checkOccupancy(map,xy,"local")
occVal = checkOccupancy(map,ij,"grid")
[occVal,validPts] = checkOccupancy( ___ )

occMatrix = checkOccupancy(map)
occMatrix = checkOccupancy(map,bottomLeft,matSize)
occMatrix = checkOccupancy(map,bottomLeft,matSize,"local")
occMatrix = checkOccupancy(map,topLeft,matSize,"grid")
```

Description

`occVal = checkOccupancy(map,xy)` returns an array of occupancy values at the `xy` locations in the world frame. Obstacle-free cells return 0, occupied cells return 1. Unknown locations, including outside the map, return `-1`.

`occVal = checkOccupancy(map,xy,"local")` returns an array of occupancy values at the `xy` locations in the local frame. The local frame is based on the `LocalOriginInWorld` property of the `map`.

`occVal = checkOccupancy(map,ij,"grid")` specifies `ij` grid cell indices instead of `xy` locations. Grid indices start at (1,1) from the top left corner.

`[occVal,validPts] = checkOccupancy( ___ )` also outputs an `n`-element vector of logical values indicating whether input coordinates are within the map limits.

`occMatrix = checkOccupancy(map)` returns a matrix that contains the occupancy status of each location. Obstacle-free cells return 0, occupied cells return 1. Unknown locations, including outside the map, return `-1`.

`occMatrix = checkOccupancy(map,bottomLeft,matSize)` returns a matrix of occupancy values by specifying the bottom-left corner location in world coordinates and the matrix size in meters.

`occMatrix = checkOccupancy(map,bottomLeft,matSize,"local")` returns a matrix of occupancy values by specifying the bottom-left corner location in local coordinates and the matrix size in meters.

`occMatrix = checkOccupancy(map,topLeft,matSize,"grid")` returns a matrix of occupancy values by specifying the top-left cell index in grid coordinates and the matrix size.

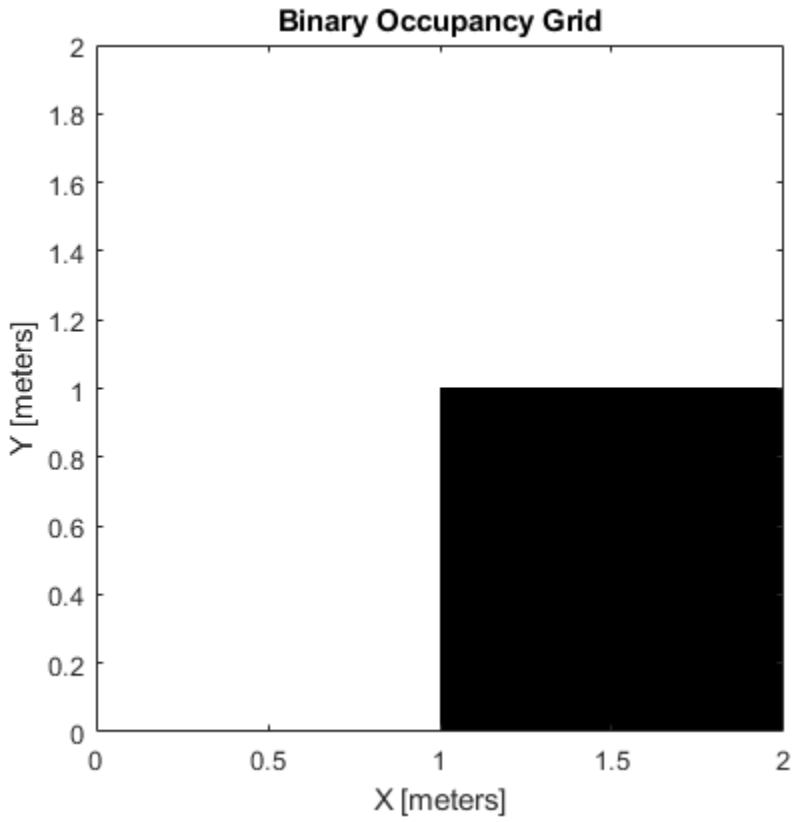
Examples

Get Occupancy Values and Check Occupancy Status

Access occupancy values and check their occupancy status based on the occupied and free thresholds of the `occupancyMap` object.

Create a matrix and populate it with values. Use this matrix to create an occupancy map.

```
p = zeros(20,20);
p(11:20,11:20) = ones(10,10);
map = binaryOccupancyMap(p,10);
show(map)
```



Get the occupancy of different locations and check their occupancy statuses. The occupancy status returns 0 for free space and 1 for occupied space. Unknown values return –1.

```
pocc = getOccupancy(map,[1.5 1]);
occupied = checkOccupancy(map,[1.5 1]);
pocc2 = getOccupancy(map,[5 5],'grid');
```

Input Arguments

`map` — Map representation
`binaryOccupancyMap` object

Map representation, specified as a `binaryOccupancyMap` object.

`xy` — Coordinates in the map
n-by-2 matrix

Coordinates in the map, specified as an *n*-by-2 matrix of `[x y]` pairs, where *n* is the number of coordinates. Coordinates can be world or local coordinates depending on the syntax.

Data Types: `double`

`ij` — Grid locations in the map
n-by-2 matrix

Grid locations in the map, specified as an *n*-by-2 matrix of `[i j]` pairs, where *n* is the number of locations. Grid locations are given as `[row col]`.

Data Types: `double`

`bottomLeft` — Location of output matrix in world or local
two-element vector | `[xCoord yCoord]`

Location of bottom left corner of output matrix in world or local coordinates, specified as a two-element vector, `[xCoord yCoord]`. Location is in world or local coordinates based on syntax.

Data Types: `double`

`matSize` — Output matrix size
two-element vector | `[xLength yLength]` | `[gridRow gridCol]`

Output matrix size, specified as a two-element vector, `[xLength yLength]`, or `[gridRow gridCol]`. Size is in world, local, or grid coordinates based on syntax.

Data Types: `double`

`topLeft` — Location of grid
two-element vector | `[iCoord jCoord]`

Location of top left corner of grid, specified as a two-element vector, `[iCoord jCoord]`.

Data Types: `double`

Output Arguments

`occVal` — Occupancy values
n-by-1 column vector

Occupancy values, returned as an *n*-by-1 column vector equal in length to `xy` or `ij` input. Occupancy values can be obstacle free (0), occupied (1), or unknown (-1).

`validPts` — Valid map locations
n-by-1 column vector

Valid map locations, returned as an *n*-by-1 column vector equal in length to `xy` or `ij`. Locations inside the map return a value of 1. Locations outside the map limits return a value of 0.

`occMatrix` — Matrix of occupancy values
matrix

Matrix of occupancy values, returned as matrix with size equal to `matSize` or the size of your `map`. Occupancy values can be obstacle free (0), occupied (1), or unknown (-1).

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`binaryOccupancyMap` | `getOccupancy` | `occupancyMap`

Introduced in R2019b

getOccupancy

Get occupancy value of locations

Syntax

```
occVal = getOccupancy(map,xy)
occVal = getOccupancy(map,xy,"local")
occVal = getOccupancy(map,ij,"grid")
[occVal,validPts] = getOccupancy( ___ )

occMatrix = getOccupancy(map)
occMatrix = getOccupancy(map,bottomLeft,matSize)
occMatrix = getOccupancy(map,bottomLeft,matSize,"local")
occMatrix = getOccupancy(map,topLeft,matSize,"grid")
```

Description

`occVal = getOccupancy(map,xy)` returns an array of occupancy values at the `xy` locations in the world frame. Unknown locations, including outside the map, return `map.DefaultValue`.

`occVal = getOccupancy(map,xy,"local")` returns an array of occupancy values at the `xy` locations in the local frame.

`occVal = getOccupancy(map,ij,"grid")` specifies `ij` grid cell indices instead of `xy` locations.

`[occVal,validPts] = getOccupancy( ___ )` additionally outputs an `n`-element vector of logical values indicating whether input coordinates are within the map limits.

`occMatrix = getOccupancy(map)` returns all occupancy values in the map as a matrix.

`occMatrix = getOccupancy(map,bottomLeft,matSize)` returns a matrix of occupancy values by specifying the bottom-left corner location in world coordinates and the matrix size in meters.

`occMatrix = getOccupancy(map,bottomLeft,matSize,"local")` returns a matrix of occupancy values by specifying the bottom-left corner location in local coordinates and the matrix size in meters.

`occMatrix = getOccupancy(map,topLeft,matSize,"grid")` returns a matrix of occupancy values by specifying the top-left cell index in grid indices and the matrix size.

Examples

Insert Laser Scans into Binary Occupancy Map

Create an empty binary occupancy grid map.

```
map = binaryOccupancyMap(10,10,20);
```

Input pose of the vehicle, ranges, angles, and the maximum range of the laser scan.

```
pose = [5,5,0];
ranges = 3*ones(100,1);
angles = linspace(-pi/2,pi/2,100);
maxrange = 20;
```

Create a `lidarScan` object with the specified ranges and angles.

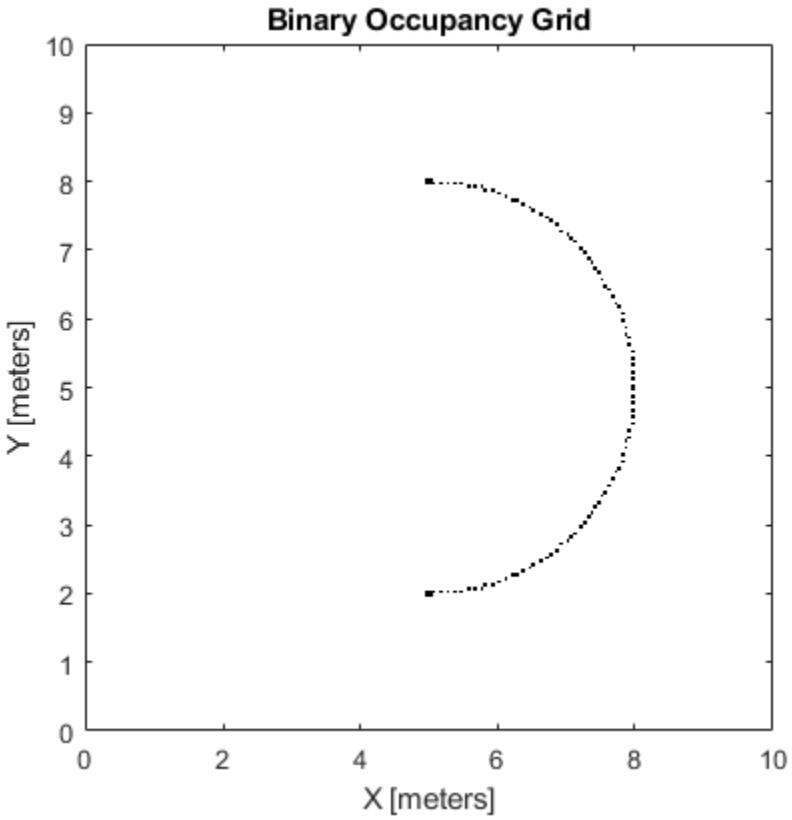
```
scan = lidarScan(ranges,angles);
```

Insert the laser scan data into the occupancy map.

```
insertRay(map,pose,scan,maxrange);
```

Show the map to see the results of inserting the laser scan.

```
show(map)
```



Check the occupancy of the spot directly in front of the vehicle.

```
getOccupancy(map,[8 5])
```

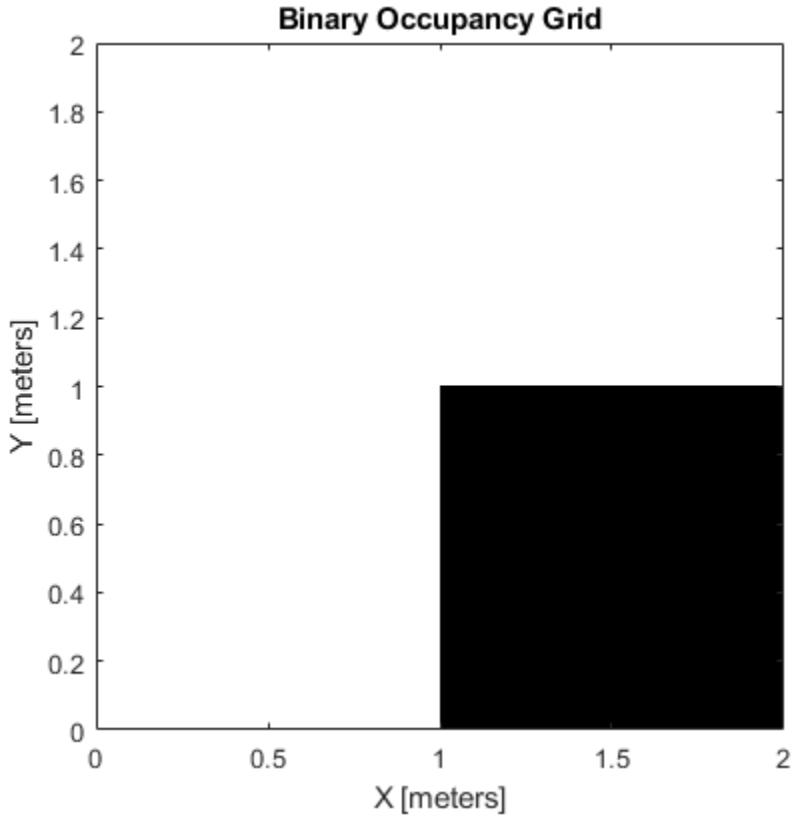
```
ans = logical
   1
```

Get Occupancy Values and Check Occupancy Status

Access occupancy values and check their occupancy status based on the occupied and free thresholds of the `occupancyMap` object.

Create a matrix and populate it with values. Use this matrix to create an occupancy map.

```
p = zeros(20,20);
p(11:20,11:20) = ones(10,10);
map = binaryOccupancyMap(p,10);
show(map)
```



Get the occupancy of different locations and check their occupancy statuses. The occupancy status returns 0 for free space and 1 for occupied space. Unknown values return –1.

```
pocc = getOccupancy(map,[1.5 1]);
occupied = checkOccupancy(map,[1.5 1]);
pocc2 = getOccupancy(map,[5 5],'grid');
```

Input Arguments

`map` — Map representation

`binaryOccupancyMap` object

Map representation, specified as a `binaryOccupancyMap` object. This object represents the environment of the vehicle.

`xy` — Coordinates in the map
n-by-2 matrix

Coordinates in the map, specified as an *n*-by-2 matrix of `[x y]` pairs, where *n* is the number of coordinates. Coordinates can be world or local coordinates depending on the syntax.

Data Types: `double`

`ij` — Grid locations in the map
n-by-2 matrix

Grid locations in the map, specified as an *n*-by-2 matrix of `[i j]` pairs, where *n* is the number of locations. Grid locations are given as `[row col]`.

Data Types: `double`

`bottomLeft` — Location of output matrix in world or local
two-element vector | `[xCoord yCoord]`

Location of bottom left corner of output matrix in world or local coordinates, specified as a two-element vector, `[xCoord yCoord]`. Location is in world or local coordinates based on syntax.

Data Types: `double`

`matSize` — Output matrix size
two-element vector | `[xLength yLength]` | `[gridRow gridCol]`

Output matrix size, specified as a two-element vector, `[xLength yLength]` or `[gridRow gridCol]`. The size is in world coordinates, local coordinates, or grid indices based on syntax.

Data Types: `double`

`topLeft` — Location of grid
two-element vector | `[iCoord jCoord]`

Location of top left corner of grid, specified as a two-element vector, `[iCoord jCoord]`.

Data Types: `double`

Output Arguments

`occVal` — Occupancy values
n-by-1 column vector

Occupancy values, returned as an *n*-by-1 column vector equal in length to `xy` or `ij`. Occupancy values can be obstacle free (0) or occupied (1).

`validPts` — Valid map locations
n-by-1 column vector

Valid map locations, returned as an *n*-by-1 column vector equal in length to `xy` or `ij`. Locations inside the map return a value of 1. Locations outside the map limits return a value of 0.

`occMatrix` — Matrix of occupancy values
matrix

Matrix of occupancy values, returned as matrix with size equal to `matSize` or the size of `map`.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`binaryOccupancyMap` | `setOccupancy`

Topics

“Occupancy Grids”

Introduced in R2015a

grid2local

Convert grid indices to local coordinates

Syntax

```
xy = grid2local(map,ij)
```

Description

`xy = grid2local(map,ij)` converts a `[row col]` array of grid indices, `ij`, to an array of local coordinates, `xy`.

Input Arguments

`map` — Map representation

`binaryOccupancyMap` object

Map representation, specified as a `binaryOccupancyMap` object. This object represents the environment of the robot. The object contains a matrix grid with binary values indicating obstacles as `true` (`1`) and free locations as `false` (`0`).

`ij` — Grid positions

n-by-2 vertical array

Grid positions, specified as an *n*-by-2 vertical array of `[i j]` pairs in `[rows cols]` format, where *n* is the number of grid positions.

Output Arguments

`xy` — Local coordinates

n-by-2 vertical array

Local coordinates, specified as an *n*-by-2 vertical array of `[x y]` pairs, where *n* is the number of local coordinates.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`binaryOccupancyMap` | `world2grid`

Introduced in R2019b

grid2world

Convert grid indices to world coordinates

Syntax

```
xy = grid2world(map,ij)
```

Description

`xy = grid2world(map,ij)` converts a `[row col]` array of grid indices, `ij`, to an array of world coordinates, `xy`.

Input Arguments

`map` — Map representation

`binaryOccupancyMap` object

Map representation, specified as a `binaryOccupancyMap` object. This object represents the environment of the robot. The object contains a matrix grid with binary values indicating obstacles as `true` (`1`) and free locations as `false` (`0`).

`ij` — Grid positions

n-by-2 vertical array

Grid positions, specified as an *n*-by-2 vertical array of `[i j]` pairs in `[rows cols]` format, where *n* is the number of grid positions.

Output Arguments

`xy` — World coordinates

n-by-2 vertical array

World coordinates, specified as an *n*-by-2 vertical array of `[x y]` pairs, where *n* is the number of world coordinates.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`binaryOccupancyMap` | `world2grid` | `grid2local`

Introduced in R2015a

inflate

Inflate each occupied grid location

Syntax

```
inflate(map,radius)
inflate(map,gridradius,'grid')
```

Description

`inflate(map,radius)` inflates each occupied position of the `map` by the radius given in meters. `radius` is rounded up to the nearest cell equivalent based on the resolution of the map. Every cell within the radius is set to `true` (`1`).

`inflate(map,gridradius,'grid')` inflates each occupied position by the radius given in number of cells.

Examples

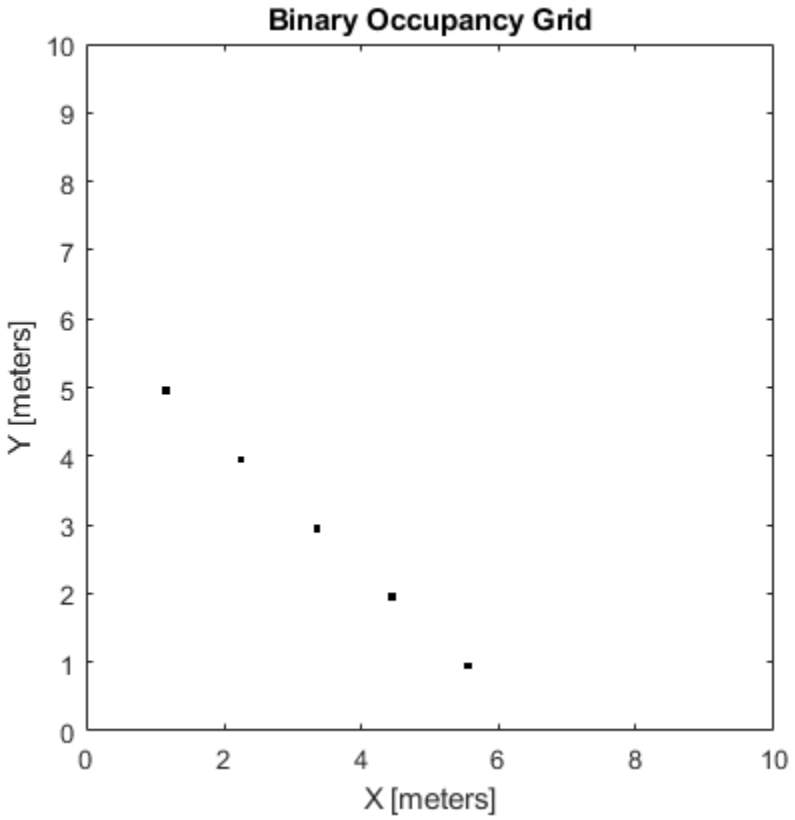
Create and Modify Binary Occupancy Grid

Create a 10m x 10m empty map.

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

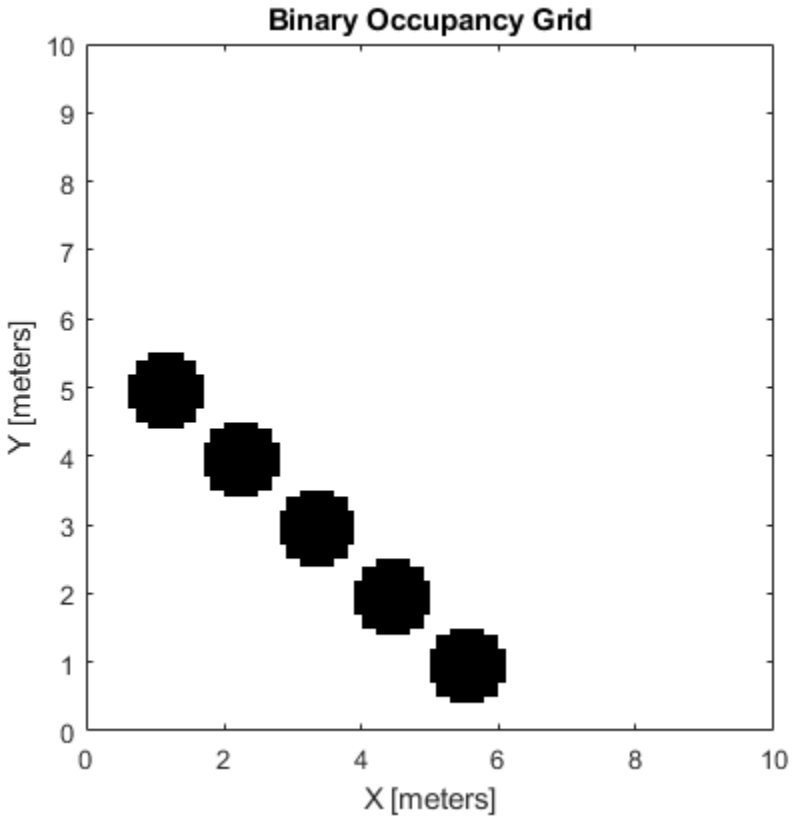
Set occupancy of world locations and show map.

```
x = [1.2; 2.3; 3.4; 4.5; 5.6];
y = [5.0; 4.0; 3.0; 2.0; 1.0];

setOccupancy(map, [x y], ones(5,1))
figure
show(map)
```



Inflate occupied locations by a given radius.

```
inflate(map, 0.5)
figure
show(map)
```

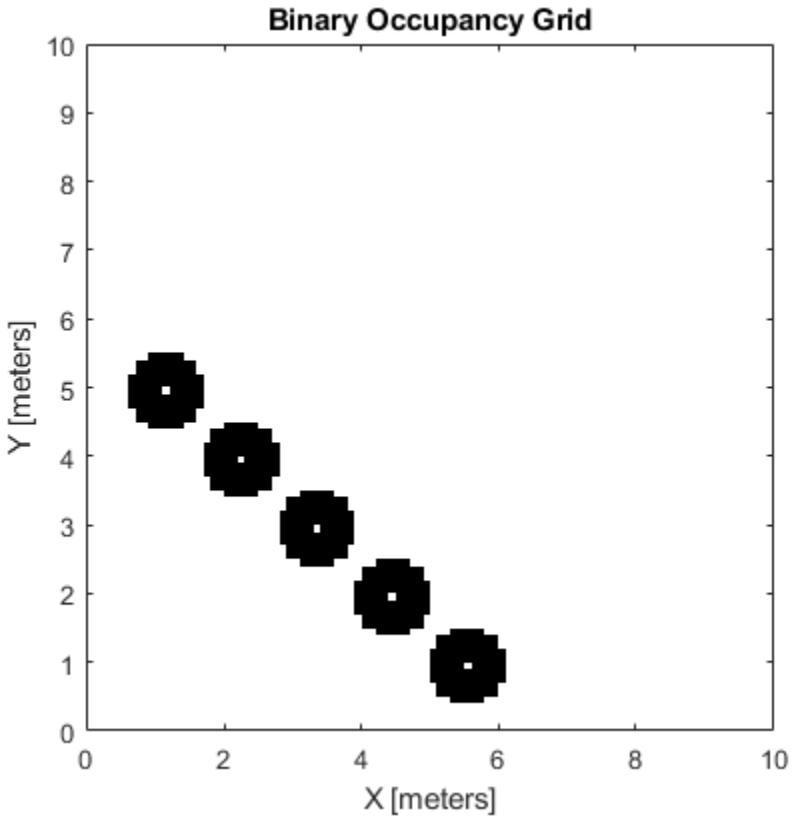


Get grid locations from world locations.

```
ij = world2grid(map, [x y]);
```

Set grid locations to free locations.

```
setOccupancy(map, ij, zeros(5,1), 'grid')
figure
show(map)
```



Input Arguments

`map` — Map representation

`binaryOccupancyMap` object

Map representation, specified as a `binaryOccupancyMap` object. This object represents the environment of the robot. The object contains a matrix grid with binary values indicating obstacles as `true` (`1`) and free locations as `false` (`0`).

`radius` — Dimension the defines how much to inflate occupied locations

scalar

Dimension that defines how much to inflate occupied locations, specified as a scalar. `radius` is rounded up to the nearest cell value.

Data Types: `double`

`gridradius` — Dimension the defines how much to inflate occupied locations

positive scalar

Dimension that defines how much to inflate occupied locations, specified as a positive scalar. `gridradius` is the number of cells to inflate the occupied locations.

Data Types: `double`

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`binaryOccupancyMap` | `setOccupancy`

Topics

“Occupancy Grids”

Introduced in R2015a

insertRay

Insert ray from laser scan observation

Syntax

```
insertRay(map,pose,scan,maxrange)
insertRay(map,pose,ranges,angles,maxrange)
insertRay(map,startpt,endpoints)
```

Description

`insertRay(map,pose,scan,maxrange)` inserts one or more lidar scan sensor observations in the occupancy grid, `map`, using the input `lidarScan` object, `scan`, to get ray endpoints. End point locations are updated with an occupied value. If the ranges are above `maxrange`, the ray endpoints are considered free space. All other points along the ray are treated as obstacle-free.

`insertRay(map,pose,ranges,angles,maxrange)` specifies the range readings as vectors defined by the input `ranges` and `angles`.

`insertRay(map,startpt,endpoints)` inserts observations between the line segments from the start point to the end points. The endpoints are updated are occupied space and other points along the line segments are updated as free space.

Examples

Insert Laser Scans into Binary Occupancy Map

Create an empty binary occupancy grid map.

```
map = binaryOccupancyMap(10,10,20);
```

Input pose of the vehicle, ranges, angles, and the maximum range of the laser scan.

```
pose = [5,5,0];
ranges = 3*ones(100,1);
angles = linspace(-pi/2,pi/2,100);
maxrange = 20;
```

Create a `lidarScan` object with the specified ranges and angles.

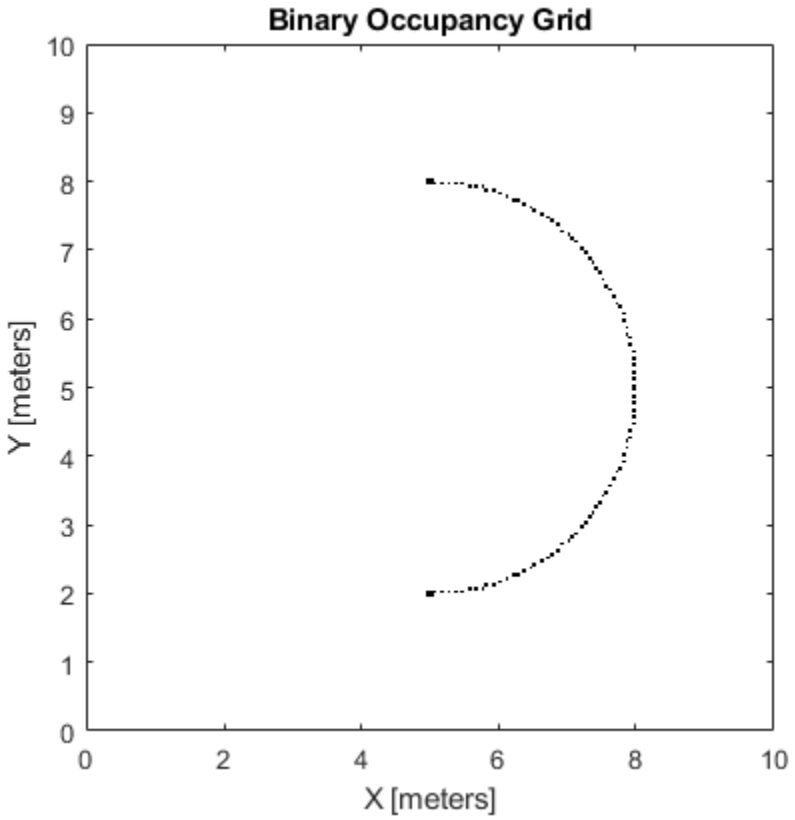
```
scan = lidarScan(ranges,angles);
```

Insert the laser scan data into the occupancy map.

```
insertRay(map,pose,scan,maxrange);
```

Show the map to see the results of inserting the laser scan.

```
show(map)
```



Check the occupancy of the spot directly in front of the vehicle.

```
getOccupancy(map,[8 5])
```

```
ans = logical
   1
```

Input Arguments

`map` — Map representation

`binaryOccupancyMap` object

Map representation, specified as a `binaryOccupancyMap` object. This object represents the environment of the robot. The object contains a matrix grid with binary values indicating obstacles as `true` (`1`) and free locations as `false` (`0`).

`pose` — Position and orientation of vehicle

three-element vector

Position and orientation of vehicle, specified as an `[x y theta]` vector. The vehicle pose is an *x* and *y* position with angular orientation *theta* (in radians) measured from the *x*-axis.

`scan` — Lidar scan readings

`lidarScan` object

Lidar scan readings, specified as a `lidarScan` object.

`ranges` — Range values from scan data
vector

Range values from scan data, specified as a vector of elements measured in meters. These range values are distances from a sensor at given `angles`. The vector must be the same length as the corresponding `angles` vector.

`angles` — Angle values from scan data
vector

Angle values from scan data, specified as a vector of elements measured in radians. These angle values correspond to the given `ranges`. The vector must be the same length as the corresponding `ranges` vector.

`maxrange` — Maximum range of sensor
scalar

Maximum range of laser range sensor, specified as a scalar in meters. Range values greater than or equal to `maxrange` are considered free along the whole length of the ray, up to `maxrange`.

`startpt` — Start point for rays
two-element vector

Start point for rays, specified as a two-element vector, `[x y]`, in the world coordinate frame. All rays are line segments that originate at this point.

`endpoints` — Endpoints for rays
n-by-2 matrix

Endpoints for rays, specified as an *n*-by-2 matrix of `[x y]` pairs in the world coordinate frame, where *n* is the length of `ranges` or `angles`. All rays are line segments that originate at `startpt`.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

`occupancyMap` | `binaryOccupancyMap` | `lidarScan` | `lidarScan`

Topics
“Occupancy Grids” (Robotics System Toolbox)

Introduced in R2019b

local2grid

Convert local coordinates to grid indices

Syntax

```
ij = local2grid(map,xy)
```

Description

`ij = local2grid(map,xy)` converts an array of local coordinates, `xy`, to an array of grid indices, `ij` in `[row col]` format.

Input Arguments

`map` — Map representation

`binaryOccupancyMap` object

Map representation, specified as a `binaryOccupancyMap` object. This object represents the environment of the vehicle.

`xy` — Local coordinates

n-by-2 matrix

Local coordinates, specified as an *n*-by-2 matrix of `[x y]` pairs, where *n* is the number of local coordinates.

Data Types: `double`

Output Arguments

`ij` — Grid positions

n-by-2 matrix

Grid positions, returned as an *n*-by-2 matrix of `[i j]` pairs in `[row col]` format, where *n* is the number of grid positions. The grid cell locations are counted from the top left corner of the grid.

Data Types: `double`

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`binaryOccupancyMap` | `occupancyMap` | `grid2world` | `grid2world`

Topics

“Occupancy Grids”

Introduced in R2019b

local2world

Convert local coordinates to world coordinates

Syntax

```
xyWorld = local2world(map,xy)
```

Description

`xyWorld = local2world(map,xy)` converts an array of local coordinates to world coordinates.

Input Arguments

`map` — Map representation

`binaryOccupancyMap` object

Map representation, specified as a `binaryOccupancyMap` object. This object represents the environment of the vehicle.

`xy` — Local coordinates

n-by-2 matrix

Local coordinates, specified as an *n*-by-2 matrix of `[x y]` pairs, where *n* is the number of local coordinates.

Data Types: `double`

Output Arguments

`xyWorld` — World coordinates

n-by-2 matrix

World coordinates, specified as an *n*-by-2 matrix of `[x y]` pairs, where *n* is the number of world coordinates.

Data Types: `double`

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

Topics

“Occupancy Grids”

Introduced in R2019b

move

Move map in world frame

Syntax

```
move(map,moveValue)
move(map,moveValue,Name,Value)
```

Description

`move(map,moveValue)` moves the local origin of the map to an absolute location, `moveValue`, in the world frame, and updates the map limits. Move values are truncated based on the resolution of the map. By default, newly revealed regions are set to `map.DefaultValue`.

`move(map,moveValue,Name,Value)` specifies additional options specified by one or more name-value pair arguments.

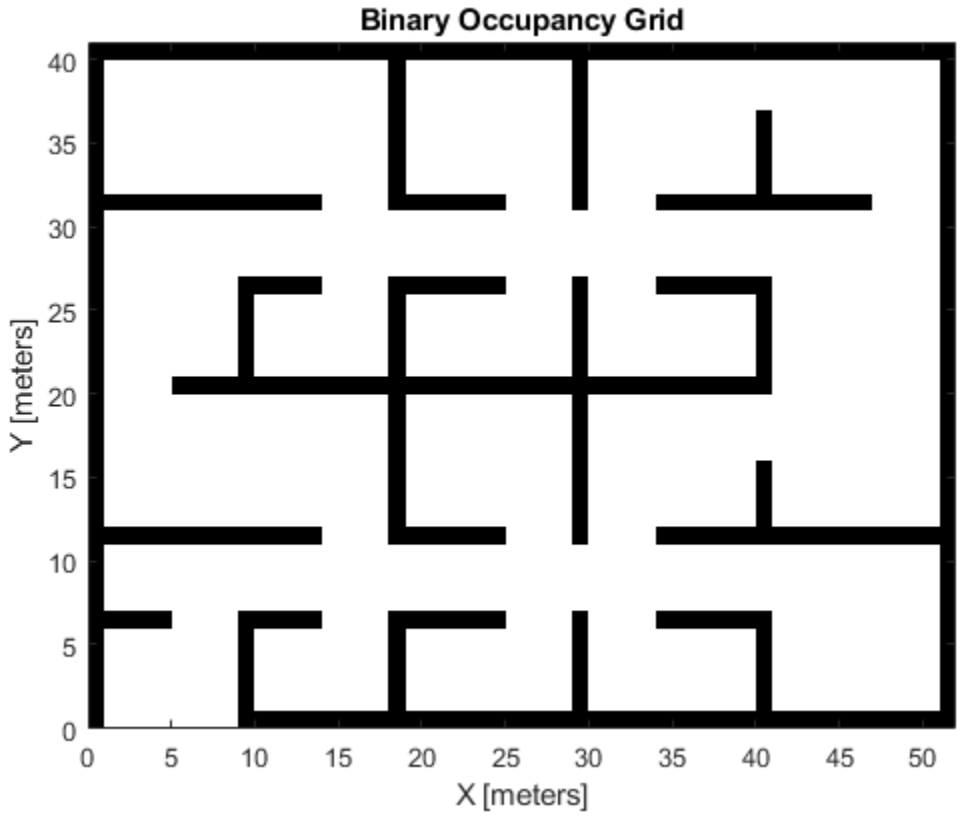
Examples

Move Local Map and Sync with World Map

This example shows how to move a local egocentric map and sync it with a larger world map. This process emulates a vehicle driving in an environment and getting updates on obstacles in the new areas.

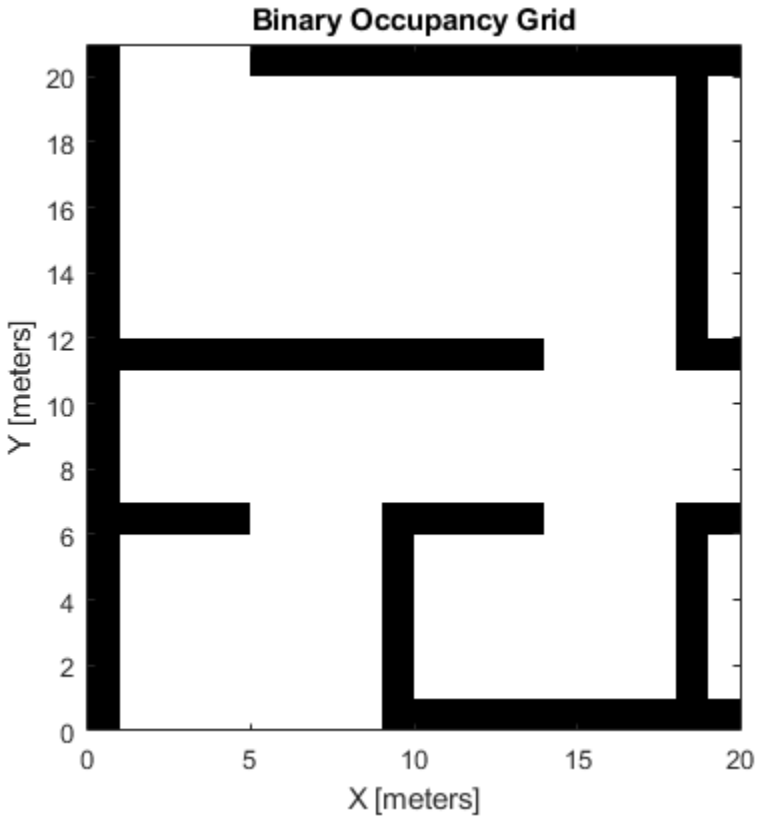
Load example maps. Create a binary occupancy map from the `complexMap`.

```
load exampleMaps.mat
map = binaryOccupancyMap(complexMap);
show(map)
```



Create a smaller local map.

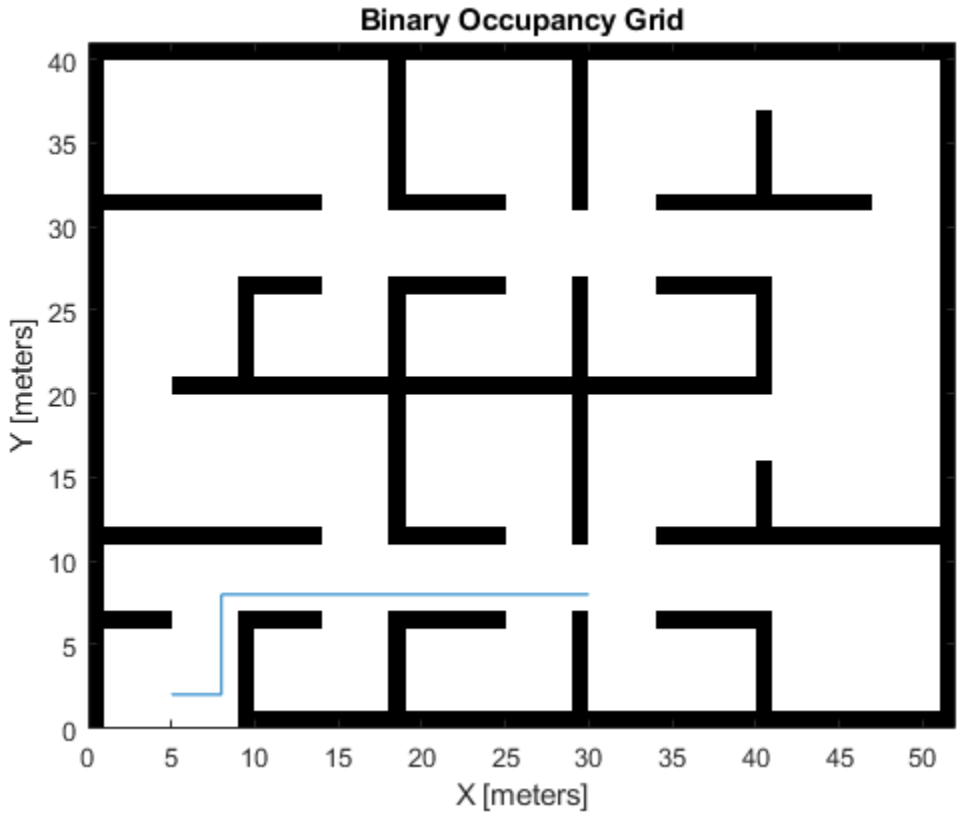
```
mapLocal = binaryOccupancyMap(complexMap(end-20:end,1:20));
show(mapLocal)
```



Follow a path planned in the world map and update the local map as you move your local frame.

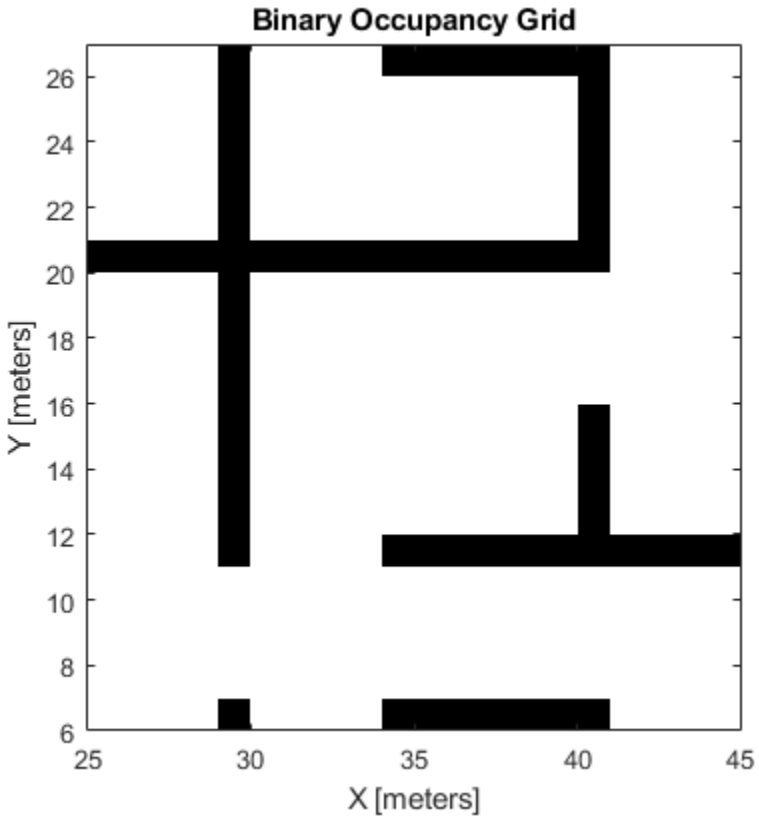
Specify path locations and plot on the map.

```
path = [5 2
        8 2
        8 8
        30 8];
show(map)
hold on
plot(path(:,1),path(:,2))
hold off
```



Create a loop for moving between points by the map resolution. Divide the difference between points by the map resolution to see how many incremental moves you can make.

```
for i = 1:length(path)-1
    moveAmount = (path(i+1,:)-path(i,:))/map.Resolution;
    for j = 1:abs(moveAmount(1)+moveAmount(2))
        moveValue = sign(moveAmount).*map.Resolution;
        move(mapLocal,moveValue, ...
            "MoveType","relative","SyncWith",map)

        show(mapLocal)
        drawnow limitrate
        pause(0.2)
    end
end
```



Input Arguments

`map` — Map representation

`binaryOccupancyMap` object

Map representation, specified as a `binaryOccupancyMap` object. This object represents the environment of the vehicle.

`moveValue` — Local map origin move value

`[x y]` vector

Local map origin move value, specified as an `[x y]` vector. By default, the value is an absolute location to move the local origin to in the world frame. Use the `MoveType` name-value pair to specify a relative move.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'MoveType','relative'`

`MoveType` — Type of move

`'absolute'` (default) | `'relative'`

Type of move, specified as `'absolute'` or `'relative'`. For relative moves, specify a relative `[x y]` vector for `moveValue` based on your current local frame.

`FillValue` — Fill value for revealed locations
`0` (default) | `1`

Fill value for revealed locations because of the shifted map limits, specified as `0` or `1`.

`SyncWith` — Secondary map to sync with
`binaryOccupancyMap` object

Secondary map to sync with, specified as a `binaryOccupancyMap` object. Any revealed locations based on the move are updated with values in this map using the world coordinates.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also
`binaryOccupancyMap` | `occupancyMap` | `occupancyMatrix`

Introduced in R2019b

occupancyMatrix

Convert occupancy grid to matrix

Syntax

```
mat = occupancyMatrix(map)
```

Description

`mat = occupancyMatrix(map)` returns occupancy values stored in the occupancy grid object as a matrix.

Input Arguments

`map` — Map representation

`binaryOccupancyMap` object

Map representation, specified as a `binaryOccupancyMap` object. This object represents the environment of the robot. The object contains a matrix grid with binary values indicating obstacles as `true` (`1`) and free locations as `false` (`0`).

Output Arguments

`mat` — Occupancy values

matrix

Occupancy values, returned as an *h*-by-*w* matrix, where *h* and *w* are defined by the two elements of the `GridSize` property of the occupancy grid object.

Data Types: `double`

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`binaryOccupancyMap` | `occupancyMap`

Topics

“Occupancy Grids”

Introduced in R2016b

raycast

Compute cell indices along a ray

Syntax

```
[endpoints,midpoints] = raycast(map,pose,range,angle)
[endpoints,midpoints] = raycast(map,p1,p2)
```

Description

`[endpoints,midpoints] = raycast(map,pose,range,angle)` returns cell indices of the specified `map` for all cells traversed by a ray originating from the specified `pose` at the specified `angle` and `range` values. `endpoints` contains all indices touched by the end of the ray, with all other points included in `midpoints`.

`[endpoints,midpoints] = raycast(map,p1,p2)` returns the cell indices of the line segment between the two specified points.

Input Arguments

`map` — Map representation

`binaryOccupancyMap` object

Map representation, specified as a `binaryOccupancyMap` object. This object represents the environment of the robot. The object contains a matrix grid with binary values indicating obstacles as `true` (`1`) and free locations as `false` (`0`).

`pose` — Position and orientation of sensor

three-element vector

Position and orientation of sensor, specified as an `[x y theta]` vector. The sensor pose is an *x* and *y* position with angular orientation *theta* (in radians) measured from the *x*-axis.

`range` — Range of ray

scalar

Range of ray, specified as a scalar in meters.

`angle` — Angle of ray

scalar

Angle of ray, specified as a scalar in radians. The angle value is for the corresponding `range`.

`p1` — Starting point of ray

two-element vector

Starting point of ray, specified as an `[x y]` two-element vector. Points are defined with respect to the world-frame.

`p2` — Endpoint of ray

two-element vector

Endpoint of ray, specified as an `[x y]` two-element vector. Points are defined with respect to the world-frame.

Output Arguments

`endpoints` — Endpoint grid indices

n-by-2 matrix

Endpoint indices, returned as an *n*-by-2 matrix of `[i j]` pairs, where *n* is the number of grid indices. The endpoints are where the `range` value hits at the specified `angle`. Multiple indices are returned when the endpoint lies on the boundary of multiple cells.

`midpoints` — Midpoint grid indices

n-by-2 matrix

Midpoint indices, returned as an *n*-by-2 matrix of `[i j]` pairs, where *n* is the number of grid indices. This argument includes all grid indices the ray intersects, excluding the endpoint.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`binaryOccupancyMap` | `insertRay` | `occupancyMap`

Topics

“Occupancy Grids” (Robotics System Toolbox)
“Occupancy Grids”

Introduced in R2019b

rayIntersection

Find intersection points of rays and occupied map cells

Syntax

```
intersectionPts = rayIntersection(map,pose,angles,maxrange)
```

Description

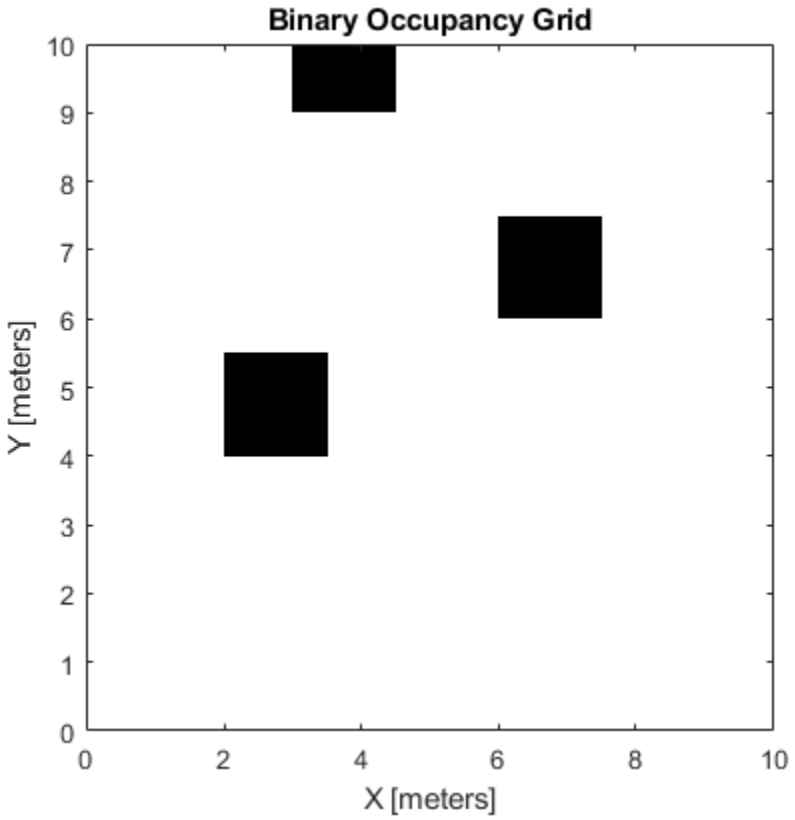
`intersectionPts = rayIntersection(map,pose,angles,maxrange)` returns intersection points of rays and occupied cells in the specified `map`. Rays emanate from the specified `pose` and `angles`. Intersection points are returned in the world coordinate frame. If there is no intersection up to the specified `maxrange`, `[NaN NaN]` is returned.

Examples

Get Ray Intersection Points on Occupancy Map

Create a binary occupancy grid map. Add obstacles and inflate them. A lower resolution map is used to illustrate the importance of the size of your grid cells. Show the map.

```
map = binaryOccupancyMap(10,10,2);
obstacles = [4 10; 3 5; 7 7];
setOccupancy(map,obstacles,ones(length(obstacles),1))
inflate(map,0.25)
show(map)
```



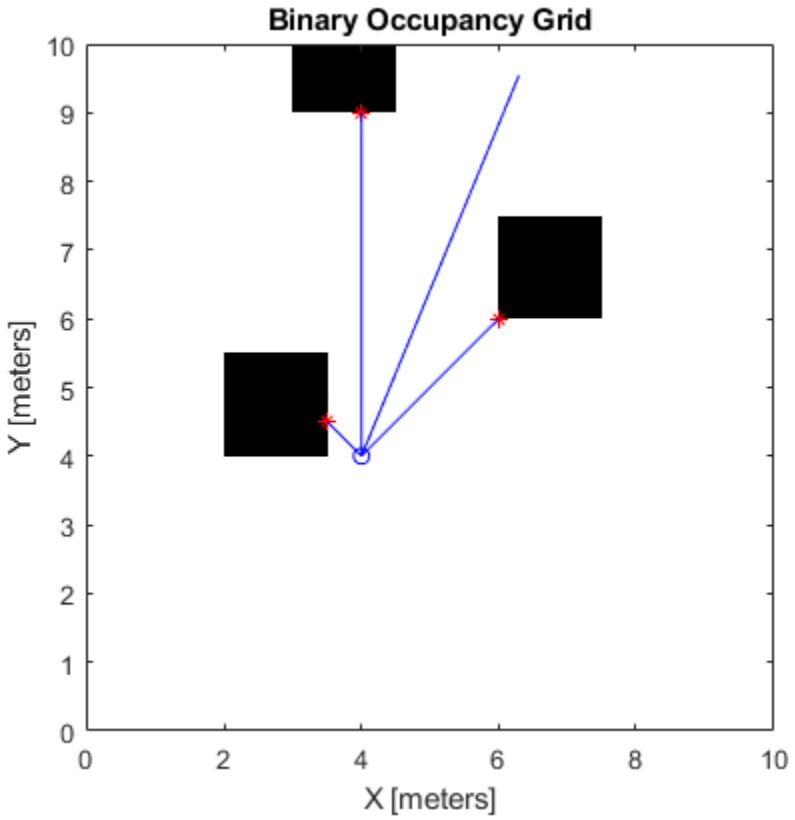
Find the intersection points of occupied cells and rays that emit from the given vehicle pose. Specify the max range and angles for these rays. The last ray does not intersect with an obstacle within the max range, so it has no collision point.

```
maxrange = 6;
angles = [pi/4,-pi/4,0,-pi/8];
vehiclePose = [4,4,pi/2];
intsectionPts = rayIntersection(map,vehiclePose,angles,maxrange)
```

```
intsectionPts = 4×2

    3.5000    4.5000
    6.0000    6.0000
    4.0000    9.0000
       NaN       NaN
```

Plot the intersection points and rays from the pose.

```
hold on
plot(intsectionPts(:,1),intsectionPts(:,2),'*r') % Intersection points
plot(vehiclePose(1),vehiclePose(2),'ob') % Vehicle pose
for i = 1:3
    plot([vehiclePose(1),intsectionPts(i,1)],...
        [vehiclePose(2),intsectionPts(i,2)],'-b') % Plot intersecting rays
end
plot([vehiclePose(1),vehiclePose(1)-6*sin(angles(4))],...
    [vehiclePose(2),vehiclePose(2)+6*cos(angles(4))],'-b') % No intersection ray
```



Input Arguments

`map` — Map representation
`binaryOccupancyMap` object

Map representation, specified as a `binaryOccupancyMap` object. This object represents the environment of the robot. The object contains a matrix grid with binary values indicating obstacles as `true` (`1`) and free locations as `false` (`0`).

`pose` — Position and orientation of sensor
three-element vector

Position and orientation of the sensor, specified as an `[x y theta]` vector. The sensor pose is an *x* and *y* position with angular orientation *theta* (in radians) measured from the *x*-axis.

`angles` — Ray angles emanating from sensor
vector

Ray angles emanating from the sensor, specified as a vector with elements in radians. These angles are relative to the specified sensor `pose`.

`maxrange` — Maximum range of sensor
scalar

Maximum range of laser range sensor, specified as a scalar in meters. Range values greater than or equal to `maxrange` are considered free along the whole length of the ray, up to `maxrange`.

Output Arguments

`intersectionPts` — Intersection points

n-by-2 matrix

Intersection points, returned as *n*-by-2 matrix of `[x y]` pairs in the world coordinate frame, where *n* is the length of `angles`.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`binaryOccupancyMap` | `occupancyMap`

Topics

“Occupancy Grids” (Robotics System Toolbox)
“Occupancy Grids” (Robotics System Toolbox)

Introduced in R2019b

setOccupancy

Set occupancy value of locations

Syntax

```
setOccupancy(map,xy,occval)
setOccupancy(map,xy,occval,"local")
setOccupancy(map,ij,occval,"grid")
validPts = setOccupancy( ___ )

setOccupancy(map,bottomLeft,inputMatrix)
setOccupancy(map,bottomLeft,inputMatrix,"local")
setOccupancy(map,topLeft,inputMatrix,"grid")
```

Description

`setOccupancy(map,xy,occval)` assigns occupancy values, `occval`, to the input array of world coordinates, `xy` in the occupancy grid, `map`. Each row of the array, `xy`, is a point in the world and is represented as an `[x y]` coordinate pair. `occval` is either a scalar or a single column array of the same length as `xy` . An occupied location is represented as `true` (`1`), and a free location is represented as `false` (`0`).

`setOccupancy(map,xy,occval,"local")` assigns occupancy values, `occval`, to the input array of local coordinates, `xy`, as local coordinates.

`setOccupancy(map,ij,occval,"grid")` assigns occupancy values, `occval`, to the input array of grid indices, `ij`, as `[rows cols]`.

`validPts = setOccupancy( ___ )` outputs an `n`-element vector of logical values indicating whether input coordinates are within the map limits.

`setOccupancy(map,bottomLeft,inputMatrix)` assigns a matrix of occupancy values by specifying the bottom-left corner location in world coordinates.

`setOccupancy(map,bottomLeft,inputMatrix,"local")` assigns a matrix of occupancy values by specifying the bottom-left corner location in local coordinates.

`setOccupancy(map,topLeft,inputMatrix,"grid")` assigns a matrix of occupancy values by specifying the top-left cell index in grid indices and the matrix size.

Examples

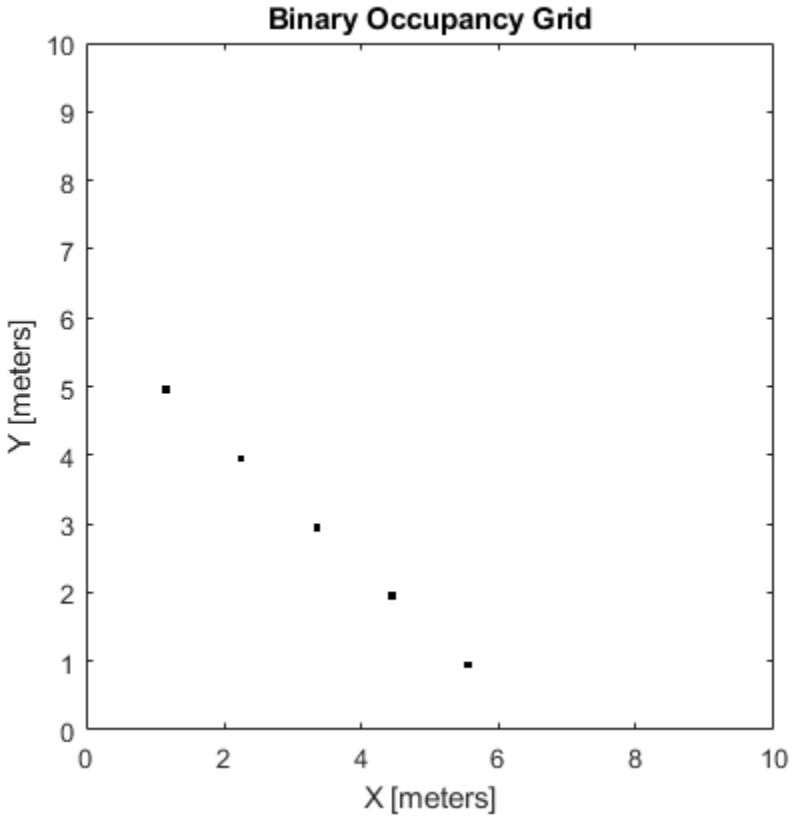
Create and Modify Binary Occupancy Grid

Create a 10m x 10m empty map.

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

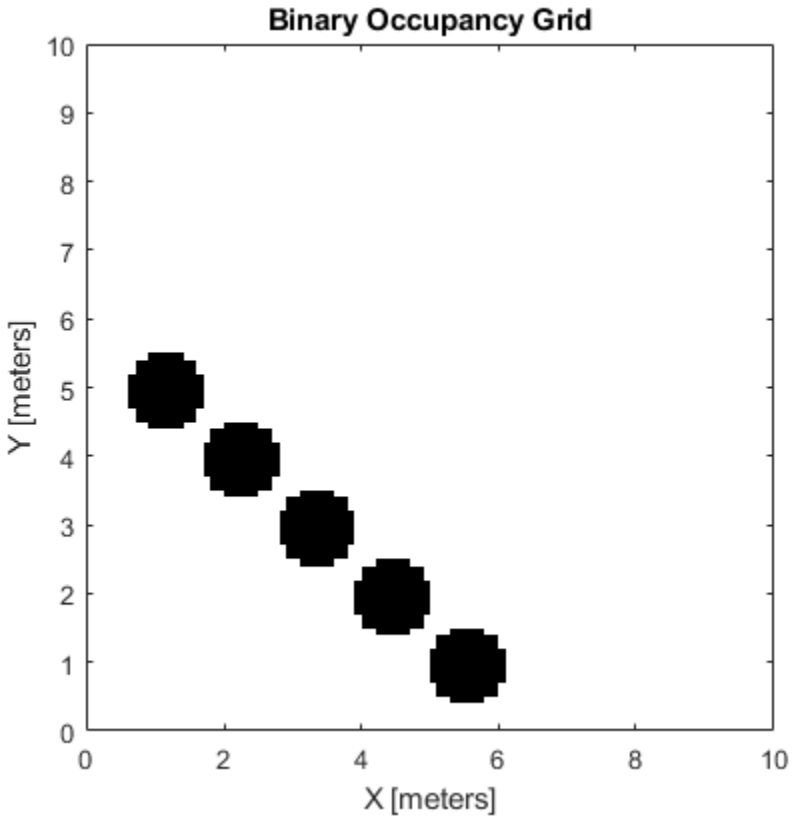
Set occupancy of world locations and show map.

```
x = [1.2; 2.3; 3.4; 4.5; 5.6];
y = [5.0; 4.0; 3.0; 2.0; 1.0];

setOccupancy(map, [x y], ones(5,1))
figure
show(map)
```



Inflate occupied locations by a given radius.

```
inflate(map, 0.5)
figure
show(map)
```

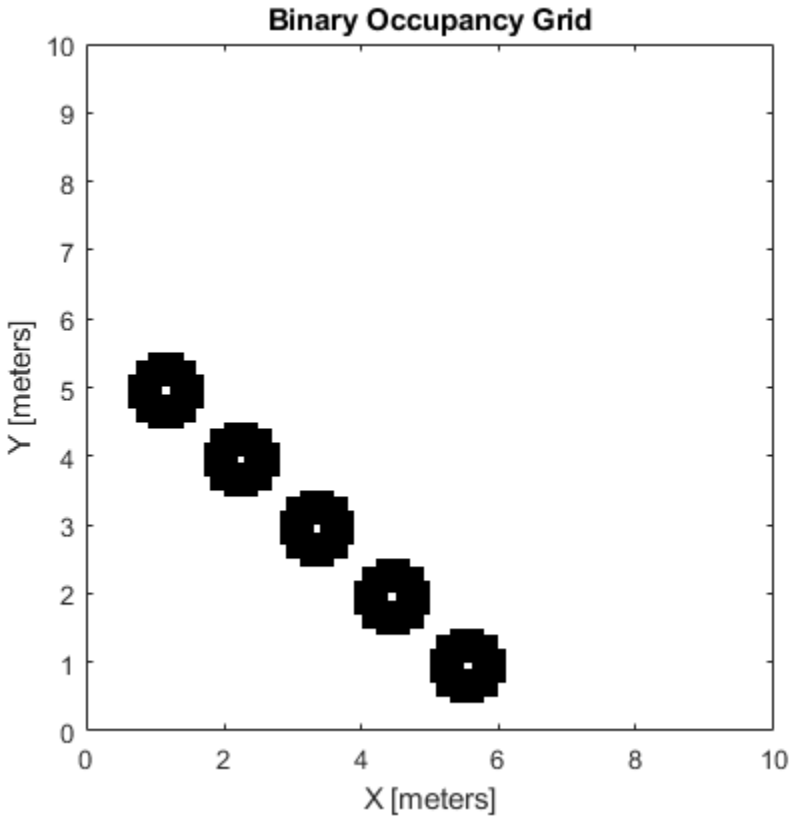


Get grid locations from world locations.

```
ij = world2grid(map, [x y]);
```

Set grid locations to free locations.

```
setOccupancy(map, ij, zeros(5,1), 'grid')
figure
show(map)
```



Input Arguments

`map` — Map representation

`binaryOccupancyMap` object

Map representation, specified as a `binaryOccupancyMap` object. This object represents the environment of the robot. The object contains a matrix grid with binary values indicating obstacles as `true` (`1`) and free locations as `false` (`0`).

`xy` — World coordinates

n-by-2 vertical array

World coordinates, specified as an *n*-by-2 vertical array of `[x y]` pairs, where *n* is the number of world coordinates.

Data Types: `double`

`ij` — Grid positions

n-by-2 vertical array

Grid positions, specified as an *n*-by-2 vertical array of `[i j]` pairs in `[rows cols]` format, where *n* is the number of grid positions.

Data Types: `double`

`occval` — Occupancy values
n-by-1 vertical array

Occupancy values of the same length as either `xy` or `ij`, returned as an *n*-by-1 vertical array, where *n* is the same *n* in either `xy` or `ij`. Values are given between 0 and 1 inclusively.

`inputMatrix` — Occupancy values
matrix

Occupancy values, specified as a matrix. Values are given between 0 and 1 inclusively.

`bottomLeft` — Location of output matrix in world or local
two-element vector | `[xCoord yCoord]`

Location of bottom left corner of output matrix in world or local coordinates, specified as a two-element vector, `[xCoord yCoord]`. Location is in world or local coordinates based on syntax.

Data Types: `double`

`topLeft` — Location of grid
two-element vector | `[iCoord jCoord]`

Location of top left corner of grid, specified as a two-element vector, `[iCoord jCoord]`.

Data Types: `double`

Output Arguments

`validPts` — Valid map locations
n-by-1 column vector

Valid map locations, returned as an *n*-by-1 column vector equal in length to `xy` or `ij`. Locations inside the map return a value of 1. Locations outside the map limits return a value of 0.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also
`binaryOccupancyMap` | `getOccupancy` | `occupancyMap`

Introduced in R2015a

show

Show occupancy grid values

Syntax

```
show(map)
show(map, "local")
show(map, "grid")
show( ___ ,Name,Value)
mapImage = show( ___ )
```

Description

`show(map)` displays the binary occupancy grid `map` in the current axes, with the axes labels representing the world coordinates.

`show(map, "local")` displays the binary occupancy grid `map` in the current axes, with the axes labels representing the local coordinates instead of world coordinates.

`show(map, "grid")` displays the binary occupancy grid `map` in the current axes, with the axes labels representing the grid coordinates.

`show( ___ ,Name,Value)` specifies additional options specified by one or more name-value pair arguments.

`mapImage = show( ___ )` returns the handle to the image object created by `show`.

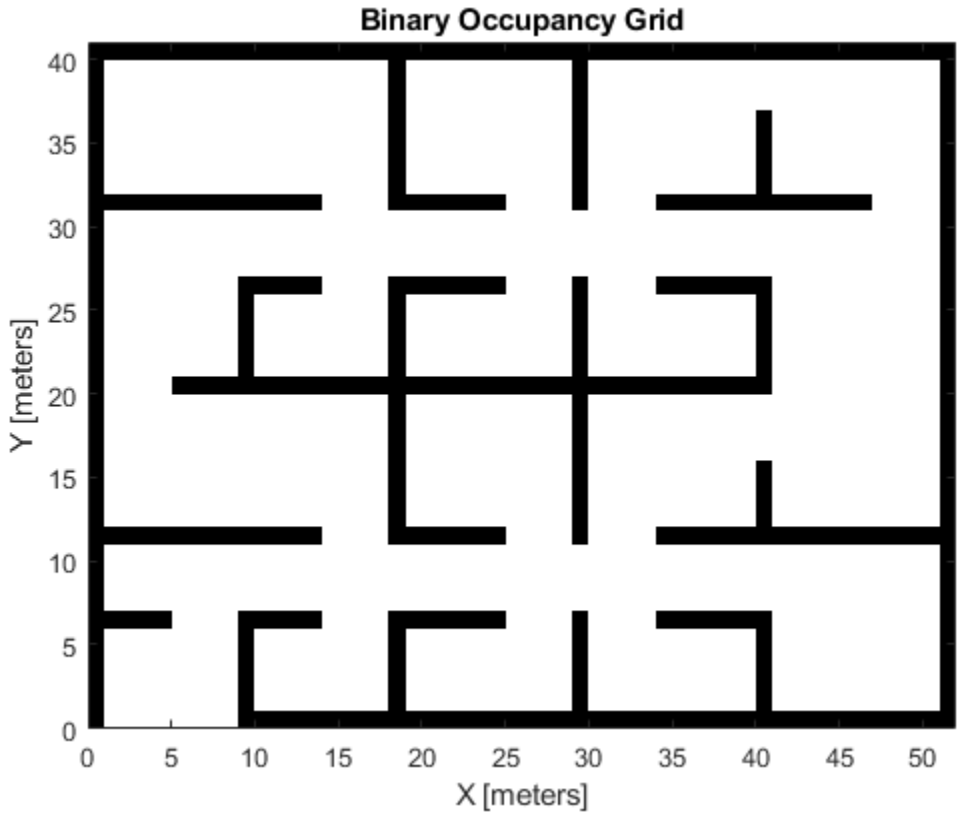
Examples

Move Local Map and Sync with World Map

This example shows how to move a local egocentric map and sync it with a larger world map. This process emulates a vehicle driving in an environment and getting updates on obstacles in the new areas.

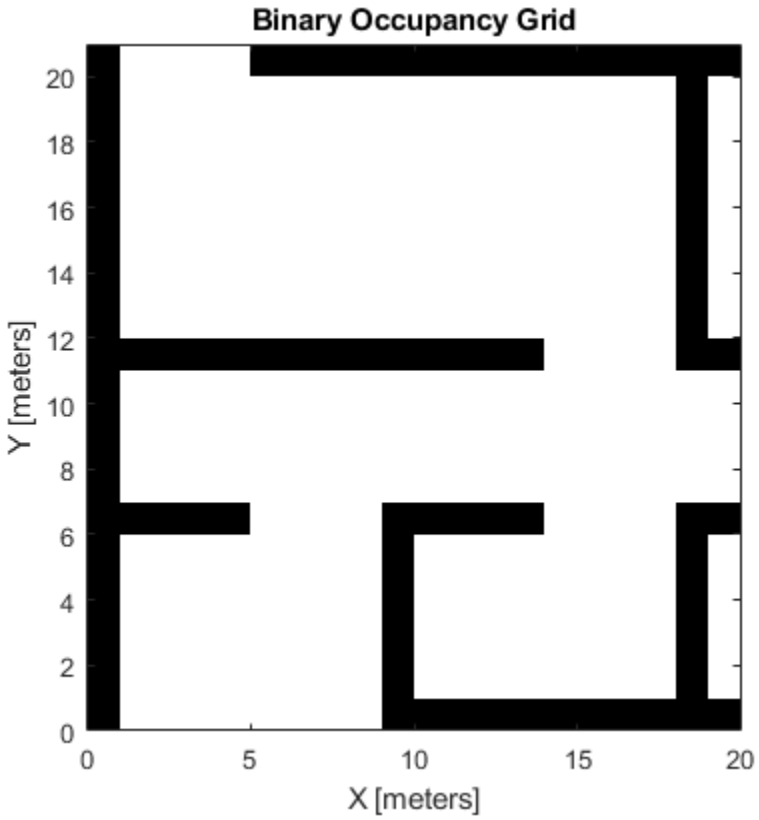
Load example maps. Create a binary occupancy map from the `complexMap`.

```
load exampleMaps.mat
map = binaryOccupancyMap(complexMap);
show(map)
```



Create a smaller local map.

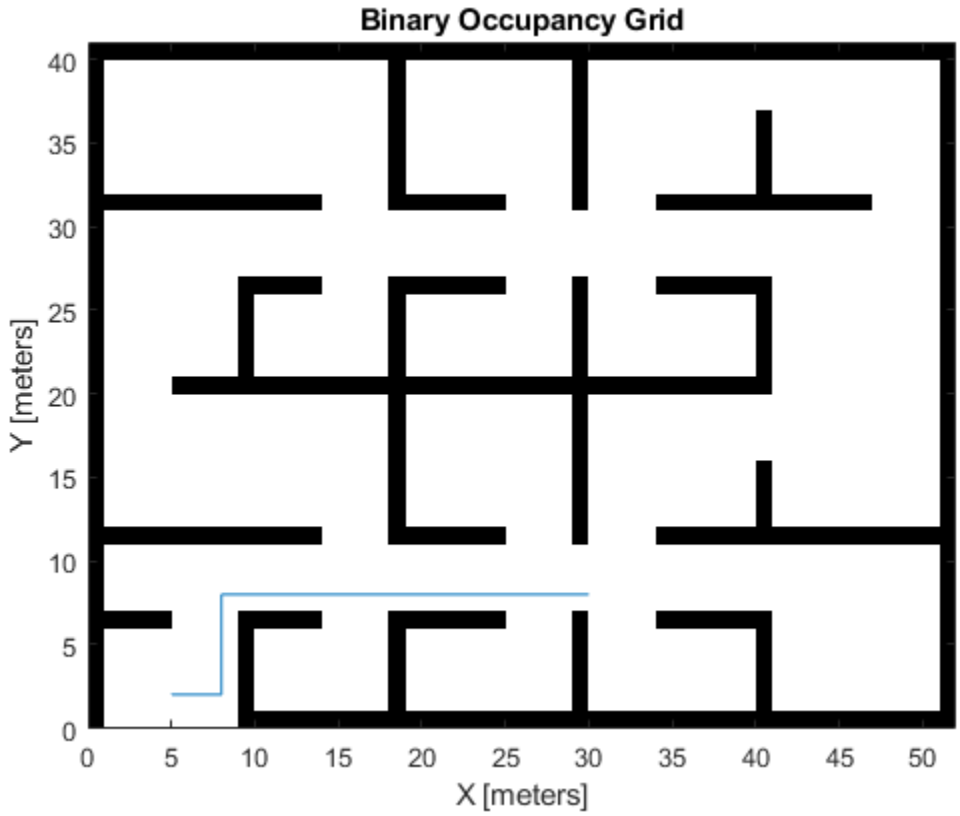
```
mapLocal = binaryOccupancyMap(complexMap(end-20:end,1:20));
show(mapLocal)
```



Follow a path planned in the world map and update the local map as you move your local frame.

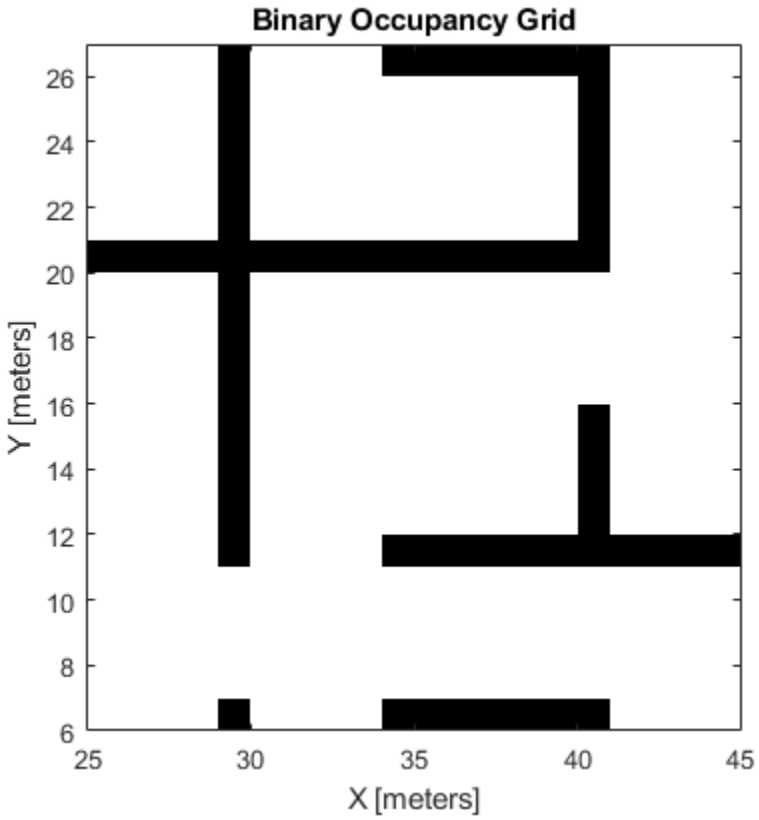
Specify path locations and plot on the map.

```
path = [5 2
        8 2
        8 8
        30 8];
show(map)
hold on
plot(path(:,1),path(:,2))
hold off
```



Create a loop for moving between points by the map resolution. Divide the difference between points by the map resolution to see how many incremental moves you can make.

```
for i = 1:length(path)-1
    moveAmount = (path(i+1,:)-path(i,:))/map.Resolution;
    for j = 1:abs(moveAmount(1)+moveAmount(2))
        moveValue = sign(moveAmount).*map.Resolution;
        move(mapLocal,moveValue, ...
            "MoveType","relative","SyncWith",map)

        show(mapLocal)
        drawnow limitrate
        pause(0.2)
    end
end
```



Input Arguments

`map` — Map representation

`binaryOccupancyMap` object

Map representation, specified as a `binaryOccupancyMap` object. This object represents the environment of the vehicle.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'Parent',axHandle`

`Parent` — Axes to plot the map

`Axes` object | `UIAxes` object

Axes to plot the map specified as either an `Axes` or `UIAxes`object. See `axes` or `uiaxes`.

`FastUpdate` — Update existing map plot

`0` (default) | `1`

Update existing map plot, specified as `0` or `1`. If you previously plotted your map on your figure, set to `1` for a faster update to the figure. This is useful for updating the figure in a loop for fast animations.

See Also

`binaryOccupancyMap` | `occupancyMap`

Introduced in R2015a

syncWith

Sync map with overlapping map

Syntax

```
mat = syncWith(map,sourcemap)
```

Description

`mat = syncWith(map,sourcemap)` updates `map` with data from another `binaryOccupancyMap` object, `sourcemap`. Locations in `map` that are also found in `sourcemap` are updated. All other cells in `map` are set to `map.DefaultValue`.

Examples

Sync Map With an Overlapping Map

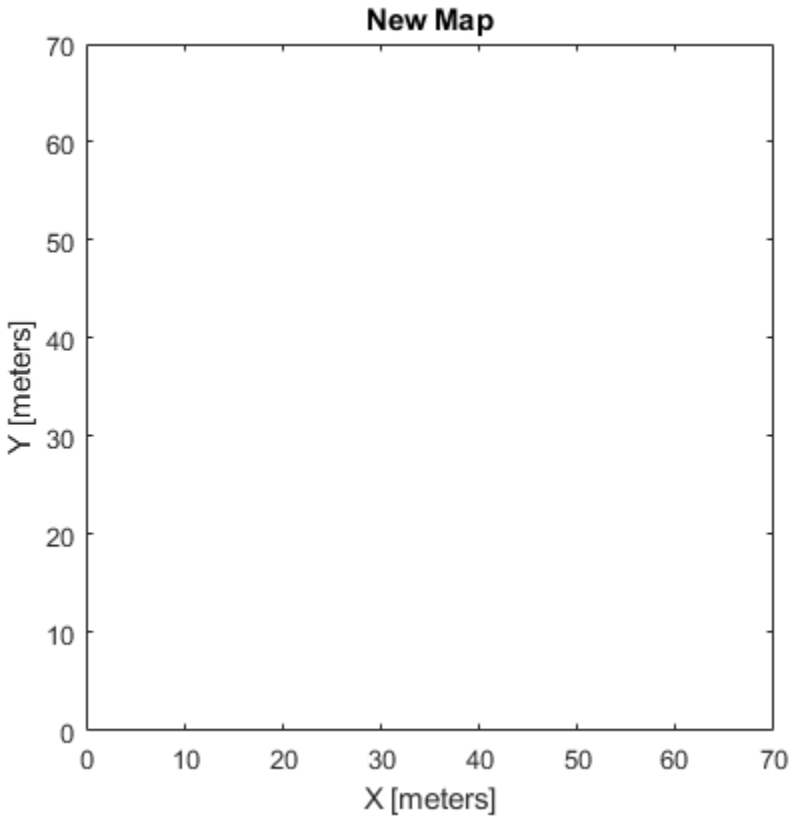
This example shows how to sync two overlapping maps using the `syncWith` function.

2-D occupancy maps are used to represent and visualize robot workspaces. In this example 2-D occupancy maps are created using existing map grid values stored inside `exampleMaps.mat`.

```
load('exampleMaps.mat');
```

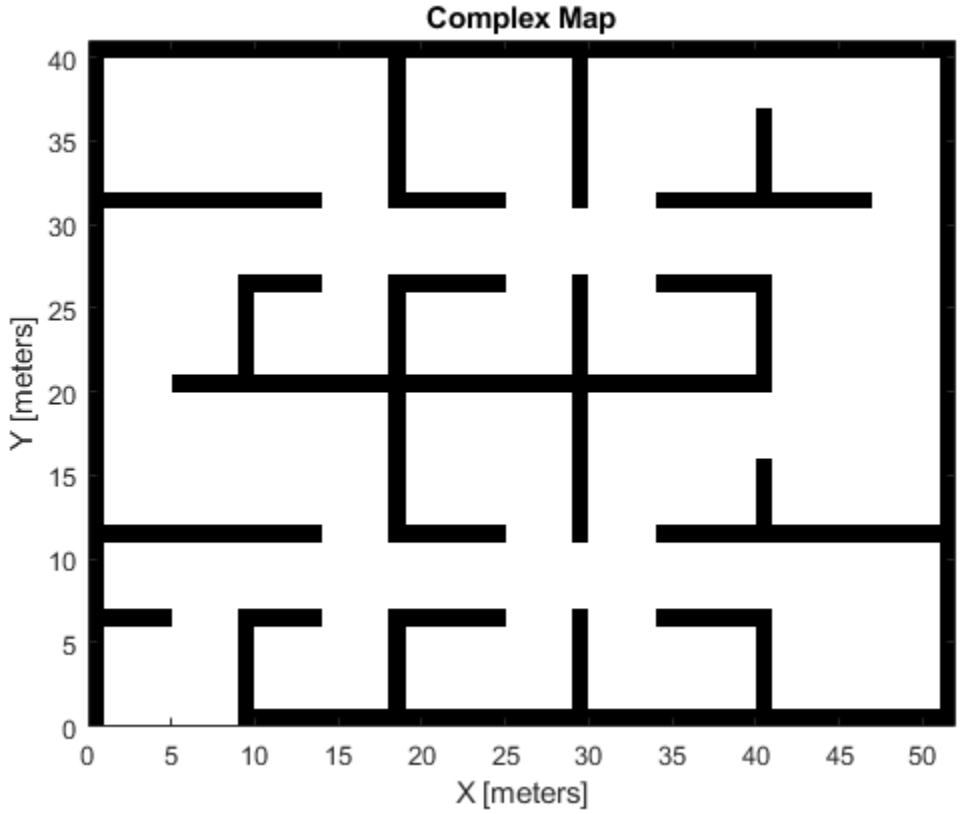
Create and display a new empty 2-D occupancy map object using `binaryOccupancyMap` function.

```
map1 = binaryOccupancyMap(70,70);
show(map1)
title('New Map')
```



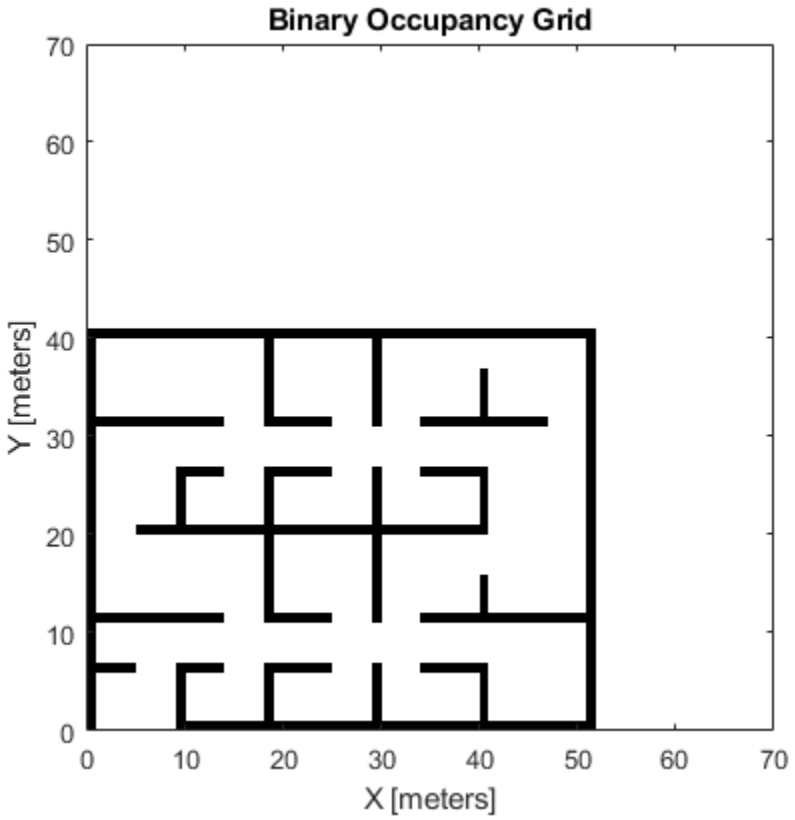
Create and display 2-D occupancy map using the map grid values stored in `complexMap`.

```
map2 = binaryOccupancyMap(complexMap);
show(map2)
title('Complex Map')
```



Now update `map1` with `map2` using the `syncWith` function.

```
syncWith(map1,map2);
show(map1)
```



Input Arguments

`map` — Map representation
`binaryOccupancyMap` object

Map representation, specified as a `binaryOccupancyMap` object.

`sourcemap` — Map representation
`binaryOccupancyMap` object

Map representation, specified as a `binaryOccupancyMap` object.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also
`binaryOccupancyMap` | `occupancyMap`

Topics
“Occupancy Grids”

Introduced in R2019b

world2grid

Convert world coordinates to grid indices

Syntax

```
ij = world2grid(map,xy)
```

Description

`ij = world2grid(map,xy)` converts an array of world coordinates, `xy`, to a `[rows cols]` array of grid indices, `ij`.

Input Arguments

`map` — Map representation

`binaryOccupancyMap` object

Map representation, specified as a `binaryOccupancyMap` object.

`xy` — World coordinates

n-by-2 vertical array

World coordinates, specified as an *n*-by-2 vertical array of `[x y]` pairs, where *n* is the number of world coordinates.

Output Arguments

`ij` — Grid indices

n-by-2 vertical array

Grid indices, specified as an *n*-by-2 vertical array of `[i j]` pairs in `[rows cols]` format, where *n* is the number of grid positions.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`binaryOccupancyMap` | `grid2world`

Introduced in R2015a

world2local

Convert world coordinates to local coordinates

Syntax

```
xyLocal = world2local(map,xy)
```

Description

`xyLocal = world2local(map,xy)` converts an array of world coordinates to local coordinates.

Input Arguments

`map` — Map representation

`binaryOccupancyMap` object

Map representation, specified as a `binaryOccupancyMap` object.

`xy` — World coordinates

n-by-2 vertical array

World coordinates, specified as an *n*-by-2 vertical array of `[x y]` pairs, where *n* is the number of world coordinates.

Output Arguments

`xyLocal` — Local coordinates

n-by-2 vertical array

Local coordinates, specified as an *n*-by-2 vertical array of `[x y]` pairs, where *n* is the number of local coordinates.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`binaryOccupancyMap` | `grid2world` | `local2world`

Introduced in R2019b

controllerVFH

Avoid obstacles using vector field histogram

Description

The `controllerVFH` System object enables your vehicle to avoid obstacles based on range sensor data using vector field histograms (VFH) . Given laser scan readings and a target direction to drive toward, the object computes an obstacle-free steering direction.

`controllerVFH` specifically uses the VFH+ algorithm to compute an obstacle-free direction. First, the algorithm takes the ranges and angles from laser scan data and builds a polar histogram for obstacle locations. Then, the input histogram thresholds are used to calculate a binary histogram that indicates occupied and free directions. Finally, the algorithm computes a masked histogram, which is computed from the binary histogram based on the minimum turning radius of the vehicle.

The algorithm selects multiple steering directions based on the open space and possible driving directions. A cost function, with weights corresponding to the previous, current, and target directions, calculates the cost of different possible directions. The object then returns an obstacle-free direction with minimal cost. Using the obstacle-free direction, you can input commands to move your vehicle in that direction.

To use this object for your own application and environment, you must tune the properties of the algorithm. Property values depend on the type of vehicle, the range sensor, and the hardware you use.

To find an obstacle-free steering direction:

1. Create the `controllerVFH` object and set its properties.
2. Call the object with arguments, as if it were a function.

To learn more about how System objects work, see What Are System Objects?

Creation

Syntax

```
VFH = controllerVFH
VFH = controllerVFH(Name,Value)
```

Description

`VFH = controllerVFH` returns a vector field histogram object that computes the obstacle-free steering direction using the `VFH+` algorithm.

`VFH = controllerVFH(Name,Value)` returns a vector field histogram object with additional options specified by one or more `Name,Value` pairs. Name is the property name and Value is the corresponding value. Name must appear inside single quotes (`' '`). You can specify several name-

value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`. Properties not specified retain their default values.

Properties

`NumAngularSectors` — Number of angular sectors in histogram

`180` (default) | positive integer

Number of angular sectors in the vector field histogram, specified as a scalar. This property defines the number of bins used to create the histograms. This property is non-tunable. You can only set this when the object is initialized.

`DistanceLimits` — Limits for range readings

`[0.05 2]` (default) | 2-element vector

Limits for range readings, specified as a 2-element vector with elements measured in meters. The range readings specified when calling the object are considered only if they fall within the distance limits. Use the lower distance limit to ignore false positives from poor sensor performance at lower ranges. Use the upper limit to ignore obstacles that are too far from the vehicle.

`RobotRadius` — Radius of vehicle

`0.1` (default) | scalar

Radius of the vehicle in meters, specified as a scalar. This dimension defines the smallest circle that can circumscribe your vehicle. The vehicle radius is used to account for vehicle size when computing the obstacle-free direction.

`SafetyDistance` — Safety distance around vehicle

`0.1` (default) | scalar

Safety distance around the vehicle, specified as a scalar in meters. This is a safety distance to leave around the vehicle position in addition to the value of the `RobotRadius` parameter. The sum of the vehicle radius and the safety distance is used to compute the obstacle-free direction.

`MinTurningRadius` — Minimum turning radius at current speed

`0.1` (default) | scalar

Minimum turning radius in meters for the vehicle moving at its current speed, specified as a scalar.

`TargetDirectionWeight` — Cost function weight for target direction

`5` (default) | scalar

Cost function weight for moving toward the target direction, specified as a scalar. To follow a target direction, set this weight to be higher than the sum of the `CurrentDirectionWeight` and `PreviousDirectionWeight` properties. To ignore the target direction cost, set this weight to zero.

`CurrentDirectionWeight` — Cost function weight for current direction

`2` (default) | scalar

Cost function weight for moving the robot in the current heading direction, specified as a scalar. Higher values of this weight produce efficient paths. To ignore the current direction cost, set this weight to zero.

`PreviousDirectionWeight` — Cost function weight for previous direction

`2` (default) | scalar

Cost function weight for moving in the previously selected steering direction, specified as a scalar. Higher values of this weight produces smoother paths. To ignore the previous direction cost, set this weight to zero.

`HistogramThresholds` — Thresholds for binary histogram computation
`[3 10]` (default) | 2-element vector

Thresholds for binary histogram computation, specified as a 2-element vector. The algorithm uses these thresholds to compute the binary histogram from the polar obstacle density. Polar obstacle density values higher than the upper threshold are represented as occupied space (1) in the binary 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).

`UseLidarScan` — Use `lidarScan` object as scan input
`false` (default) | `true`

Use `lidarScan` object as scan input, specified as either `true` or `false`.

Usage

Syntax

```
steeringDir = vfh(scan,targetDir)
steeringDir = vfh(ranges,angles,targetDir)
```

Description

`steeringDir = vfh(scan,targetDir)` finds an obstacle-free steering direction using the VFH+ algorithm for the input `lidarScan` object, `scan`. A target direction is given based on the target location.

To enable this syntax, you must set the `UseLidarScan` property to `true`. For example:

```
mcl = monteCarloLocalization('UseLidarScan',true);
...
[isUpdated,pose,covariance] = mcl(odomPose,scan);
```

`steeringDir = vfh(ranges,angles,targetDir)` defines the lidar scan with two vectors: `ranges` and `angles`.

Input Arguments

`scan` — Lidar scan readings
`lidarScan` object

Lidar scan readings, specified as a `lidarScan` object.

Dependencies

To use this argument, you must set the `UseLidarScan` property to `true`.

```
mcl.UseLidarScan = true;
```

`ranges` — Range values from scan data

vector

Range values from scan data, specified as a vector in meters. These range values are distances from a sensor at given `angles`. The vector must be the same length as the corresponding `angles` vector.

`angles` — Angle values from scan data

vector

Angle values from scan data, specified as a vector in radians. These angle values are the specific angles of the given `ranges`. The vector must be the same length as the corresponding `ranges` vector.

`targetDir` — Target direction for vehicle

scalar

Target direction for the vehicle, specified as a scalar in radians. The forward direction of the vehicle is considered zero radians, with positive angles measured counterclockwise.

Output Arguments

`steeringDir` — Steering direction for vehicle

scalar

Steering direction for the vehicle, specified as a scalar in radians. This obstacle-free direction is calculated based on the VFH+ algorithm. The forward direction of the vehicle is considered zero radians, with positive angles measured counterclockwise.

Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

Specific to controllerVFH

show Display VectorFieldHistogram information in figure window

Common to All System Objects

step Run System object algorithm
release Release resources and allow changes to System object property values and input characteristics
reset Reset internal states of System object

Examples

Create a Vector Field Histogram Object and Visualize Data

This example shows how to calculate a steering direction based on input laser scan data.

Create a `controllerVFH` object. Set the `UseLidarScan` property to `true`.

```
vfh = controllerVFH;
vfh.UseLidarScan = true;
```

Input laser scan data and target direction.

```
ranges = 10*ones(1,500);
ranges(1,225:275) = 1.0;
angles = linspace(-pi,pi,500);
targetDir = 0;
```

Create a `lidarScan` object by specifying the ranges and angles.

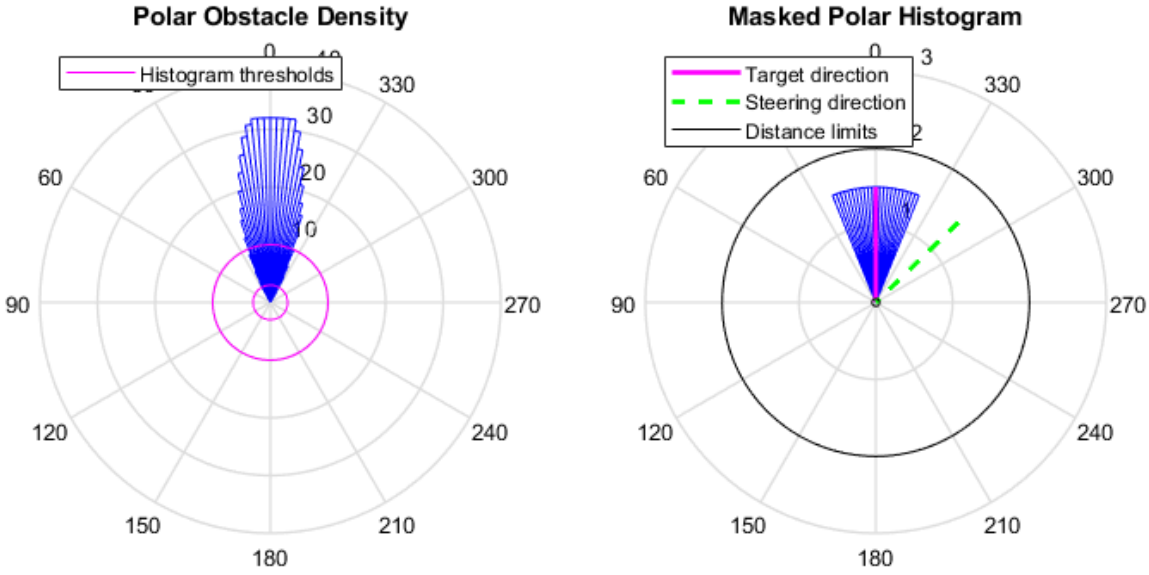
```
scan = lidarScan(ranges,angles);
```

Compute an obstacle-free steering direction.

```
steeringDir = vfh(scan,targetDir);
```

Visualize the `VectorFieldHistogram` computation.

```
h = figure;
set(h,'Position',[50 50 800 400])
show(vfh)
```



References

[1] Borenstein, J., and Y. Koren. "The Vector Field Histogram - Fast Obstacle Avoidance for Mobile Robots." IEEE Journal of Robotics and Automation. Vol. 7, Number 3, 1991, pp.278–88.

[2] Ulrich, I., and J. Borenstein. "VFH : Reliable Obstacle Avoidance for Fast Mobile Robots." *Proceedings. 1998 IEEE International Conference on Robotics and Automation*. (1998): 1572–1577.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

Use in a MATLAB Function block is not supported.

Lidar scans require a limited size in code generation. The lidar scans, `scan`, are limited to 4000 points (range and angles) as a maximum.

For additional information about code generation for System objects, see “System Objects in MATLAB Code Generation” (MATLAB Coder)

See Also

`lidarScan` | `show`

Topics

“Obstacle Avoidance with TurtleBot and VFH”
“Vector Field Histogram”

Introduced in R2019b

show

Display `VectorFieldHistogram` information in figure window

Syntax

```
show(vfh)

show(vfh,'Parent',parent)

h = show( ___ )
```

Description

`show(vfh)` shows histograms calculated by the VFH+ algorithm in a figure window. The figure also includes the parameters of the `controllerVFH` object and range values from the last object call.

`show(vfh,'Parent',parent)` sets the specified axes handle, `parent`, to the axes.

`h = show( ___ )` returns the figure object handle created by `show` using any of the arguments from the previous syntaxes.

Examples

Create a Vector Field Histogram Object and Visualize Data

This example shows how to calculate a steering direction based on input laser scan data.

Create a `controllerVFH` object. Set the `UseLidarScan` property to `true`.

```
vfh = controllerVFH;
vfh.UseLidarScan = true;
```

Input laser scan data and target direction.

```
ranges = 10*ones(1,500);
ranges(1,225:275) = 1.0;
angles = linspace(-pi,pi,500);
targetDir = 0;
```

Create a `lidarScan` object by specifying the ranges and angles.

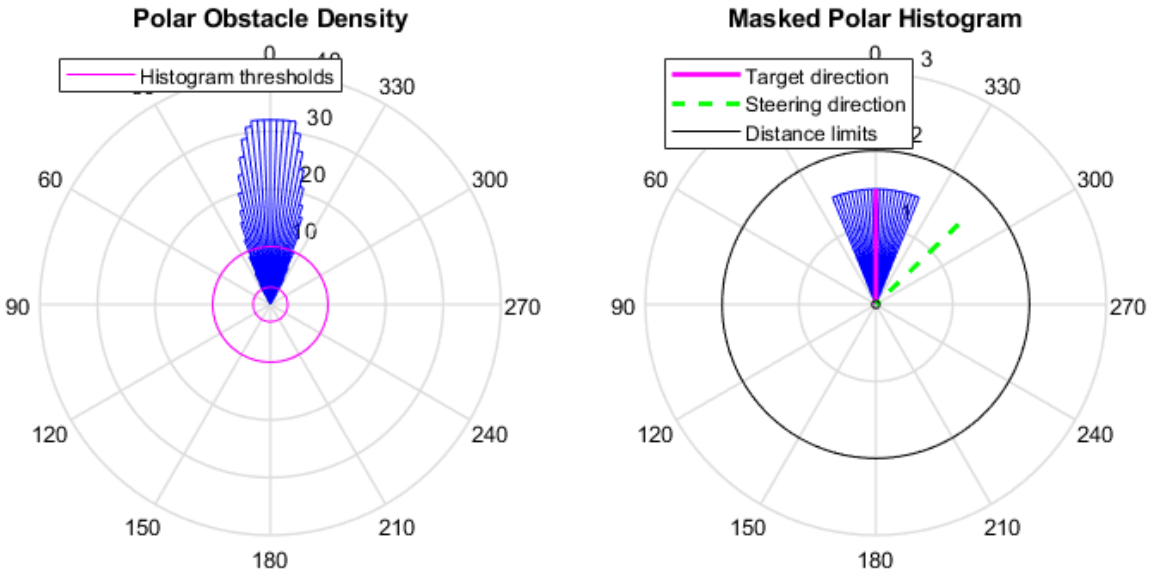
```
scan = lidarScan(ranges,angles);
```

Compute an obstacle-free steering direction.

```
steeringDir = vfh(scan,targetDir);
```

Visualize the `VectorFieldHistogram` computation.

```
h = figure;
set(h,'Position',[50 50 800 400])
show(vfh)
```



Input Arguments

`vfh` — Vector field histogram algorithm

`controllerVFH` object

Vector field histogram algorithm, specified as a `controllerVFH` object. This object contains all the parameters for tuning the VFH+ algorithm.

`parent` — Axes properties

handle

Axes properties, specified as a handle.

Output Arguments

`h` — Axes handles for VFH algorithm display

`Axes` array

Axes handles for VFH algorithm display, specified as an `Axes` array. The VFH histogram and `HistogramThresholds` are shown in the first axes. The binary histogram, range sensor readings, target direction, and steering directions are shown in the second axes.

See Also

`controllerVFH`

Introduced in R2019b

controllerPurePursuit

Create controller to follow set of waypoints

Description

The `controllerPurePursuit` System object creates a controller object used to make a differential-drive vehicle follow a set of waypoints. The object computes the linear and angular velocities for the vehicle given the current pose. Successive calls to the object with updated poses provide updated velocity commands for the vehicle. Use the `MaxAngularVelocity` and `DesiredLinearVelocity` properties to update the velocities based on the vehicle's performance.

The `LookaheadDistance` property computes a look-ahead point on the path, which is a local goal for the vehicle. The angular velocity command is computed based on this point. Changing `LookaheadDistance` has a significant impact on the performance of the algorithm. A higher look-ahead distance results in a smoother trajectory for the vehicle, but can cause the vehicle to cut corners along the path. A low look-ahead distance can result in oscillations in tracking the path, causing unstable behavior. For more information on the pure pursuit algorithm, see “Pure Pursuit Controller”.

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.

To compute linear and angular velocity control commands:

1. Create the `controllerPurePursuit` object and set its properties.
2. Call the object with arguments, as if it were a function.

To learn more about how System objects work, see What Are System Objects?

Creation

Syntax

```
controller = controllerPurePursuit

controller = controllerPurePursuit(Name,Value)
```

Description

`controller = controllerPurePursuit` creates a pure pursuit object that uses the pure pursuit algorithm to compute the linear and angular velocity inputs for a differential drive vehicle.

`controller = controllerPurePursuit(Name,Value)` creates a pure pursuit object with additional options specified by one or more `Name,Value` pairs. Name is the property name and Value is the corresponding value. Name must appear inside single quotes (`' '`). You can specify several

name-value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`. Properties not specified retain their default values.

Example: `controller = controllerPurePursuit('DesiredLinearVelocity', 0.5)`

Properties

`DesiredLinearVelocity` — Desired constant linear velocity

`0.1` (default) | scalar in meters per second

Desired constant linear velocity, specified as a scalar in meters per second. The controller assumes that the vehicle drives at a constant linear velocity and that the computed angular velocity is independent of the linear velocity.

Data Types: `double`

`LookaheadDistance` — Look-ahead distance

`1.0` (default) | scalar in meters

Look-ahead distance, specified as a scalar in meters. The look-ahead distance changes the response of the controller. A vehicle with a higher look-ahead distance produces smooth paths but takes larger turns at corners. A vehicle with a smaller look-ahead distance follows the path closely and takes sharp turns, but potentially creating oscillations in the path.

Data Types: `double`

`MaxAngularVelocity` — Maximum angular velocity

`1.0` (default) | scalar in radians per second

Maximum angular velocity, specified a scalar in radians per second. The controller saturates the absolute angular velocity output at the given value.

Data Types: `double`

`Waypoints` — Waypoints

[ ] (default) | *n*-by-2 array

Waypoints, specified as an *n*-by-2 array of `[x y]` pairs, where *n* is the number of waypoints. You can generate the waypoints from the `mobileRobotPRM` class or from another source.

Data Types: `double`

Usage

Syntax

```
[vel,angvel] = controller(pose)
[vel,angvel,lookaheadpoint] = controller(pose)
```

Description

`[vel,angvel] = controller(pose)` processes the vehicle's position and orientation, `pose`, and outputs the linear velocity, `vel`, and angular velocity, `angvel`.

`[vel,angvel,lookaheadpoint] = controller(pose)` returns the look-ahead point, which is a location on the path used to compute the velocity commands. This location on the path is computed using the `LookaheadDistance` property on the `controller` object.

Input Arguments

`pose` — Position and orientation of vehicle
3-by-1 vector in the form `[x y theta]`

Position and orientation of vehicle, specified as a 3-by-1 vector in the form `[x y theta]`. The vehicle pose is an *x* and *y* position with angular orientation θ (in radians) measured from the *x*-axis.

Output Arguments

`vel` — Linear velocity
scalar in meters per second

Linear velocity, specified as a scalar in meters per second.

Data Types: `double`

`angvel` — Angular velocity
scalar in radians per second

Angular velocity, specified as a scalar in radians per second.

Data Types: `double`

`lookaheadpoint` — Look-ahead point on path
`[x y]` vector

Look-ahead point on the path, returned as an `[x y]` vector. This value is calculated based on the `LookaheadDistance` property.

Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

Specific to controllerPurePursuit

info Characteristic information about controllerPurePursuit object

Common to All System Objects

step Run System object algorithm
release Release resources and allow changes to System object property values and input characteristics
reset Reset internal states of System object

Examples

Get Additional Pure Pursuit Object Information

Use the `info` method to get more information about a `controllerPurePursuit` object. The `info` function returns two fields, `RobotPose` and `LookaheadPoint`, which correspond to the current position and orientation of the robot and the point on the path used to compute outputs from the last call of the object.

Create a `controllerPurePursuit` object.

```
pp = controllerPurePursuit;
```

Assign waypoints.

```
pp.Waypoints = [0 0;1 1];
```

Compute control commands using the `pp` object with the initial pose `[x y theta]` given as the input.

```
[v,w] = pp([0 0 0]);
```

Get additional information.

```
s = info(pp)
```

```
s = struct with fields:
          RobotPose: [0 0 0]
    LookaheadPoint: [0.7071 0.7071]
```

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

Use in a MATLAB Function block is not supported.

For additional information about code generation for System objects, see “System Objects in MATLAB Code Generation” (MATLAB Coder)

See Also

`binaryOccupancyMap` | `occupancyMap` | `binaryOccupancyMap` | `occupancyMap` | `controllerVFH`

Topics

“Pure Pursuit Controller”

Introduced in R2019b

info

Characteristic information about `controllerPurePursuit` object

Syntax

```
controllerInfo = info(controller)
```

Description

`controllerInfo = info(controller)` returns a structure, `controllerInfo`, with additional information about the status of the `controllerPurePursuit` object, `controller`. The structure contains the fields, `RobotPose` and `LookaheadPoint`.

Examples

Get Additional Pure Pursuit Object Information

Use the `info` method to get more information about a `controllerPurePursuit` object. The `info` function returns two fields, `RobotPose` and `LookaheadPoint`, which correspond to the current position and orientation of the robot and the point on the path used to compute outputs from the last call of the object.

Create a `controllerPurePursuit` object.

```
pp = controllerPurePursuit;
```

Assign waypoints.

```
pp.Waypoints = [0 0;1 1];
```

Compute control commands using the `pp` object with the initial pose `[x y theta]` given as the input.

```
[v,w] = pp([0 0 0]);
```

Get additional information.

```
s = info(pp)
```

```
s = struct with fields:
          RobotPose: [0 0 0]
    LookaheadPoint: [0.7071 0.7071]
```

Input Arguments

`controller` — Pure pursuit controller

`controllerPurePursuit` object

Pure pursuit controller, specified as a `controllerPurePursuit` object.

Output Arguments

`controllerInfo` — Information on the `controllerPurePursuit` object
structure

Information on the `controllerPurePursuit` object, returned as a structure. The structure contains two fields:

- `RobotPose` – A three-element vector in the form `[x y theta]` that corresponds to the *x*-*y* position and orientation of the vehicle. The angle, `theta`, is measured in radians with positive angles measured counterclockwise from the *x*-axis.
- `LookaheadPoint`– A two-element vector in the form `[x y]`. The location is a point on the path that was used to compute outputs of the last call to the object.

See Also

`controllerPurePursuit`

Topics
“Pure Pursuit Controller”

Introduced in R2019b

dubinsConnection

Dubins path connection type

Description

The `dubinsConnection` object holds information for computing a `dubinsPathSegment` path segment to connect between poses. A Dubins path segment connects two poses as a sequence of three motions. The motion options are:

- Straight
- Left turn at maximum steer
- Right turn at maximum steer

A Dubins path segment only allows motion in the forward direction.

Use this connection object to define parameters for a robot motion model, including the minimum turning radius and options for path types. To generate a path segment between poses using this connection type, call the `connect` function.

Creation

Syntax

```
dubConnObj = dubinsConnection
dubConnObj = dubinsConnection(Name,Value)
```

Description

`dubConnObj = dubinsConnection` creates an object using default property values.

`dubConnObj = dubinsConnection(Name,Value)` specifies property values using name-value pairs. To set multiple properties, specify multiple name-value pairs.

Properties

`MinTurningRadius` — Minimum turning radius of the vehicle
`1` (default) | positive scalar in meters

Minimum turning radius of the vehicle, specified as a positive scalar in meters. The minimum turning radius is for the smallest circle the vehicle can make with maximum steer in a single direction.

Data Types: `double`

`DisabledPathTypes` — Path types to disable
`{}` (default) | cell array of three-element character vectors | vector of three-element string scalars

Dubins path types to disable, specified as a cell array of three-element character vectors or vector of string scalars. The cell array defines three sequences of motions that are prohibited by the vehicle motion model.

Motion Type	Description
`"S"`	Straight
`"L"`	Left turn at the maximum steering angle of the vehicle
`"R"`	Right turn at the maximum steering angle of the vehicle

To see all available path types, see the `AllPathTypes` property.

For Dubins connections, the available path types are: `{"LSL"} {"LSR"} {"RSL"} {"RSR"} {"RLR"} {"LRL"}`.

Example: `["LSL","LSR"]`

Data Types: `string` | `cell`

`AllPathTypes` — All possible path types
cell array of character vectors

This property is read-only.

All possible path types, returned as a cell array of character vectors. This property lists all types. To disable certain types, specify types from this list in `DisabledPathTypes`.

For Dubins connections, the available path types are: `{'LSL'} {'LSR'} {'RSL'} {'RSR'} {'RLR'} {'LRL'}`.

Data Types: `cell`

Object Functions

connect Connect poses for given connection type

Examples

Connect Poses Using Dubins Connection Path

Create a `dubinsConnection` object.

```
dubConnObj = dubinsConnection;
```

Define start and goal poses as `[x y theta]` vectors.

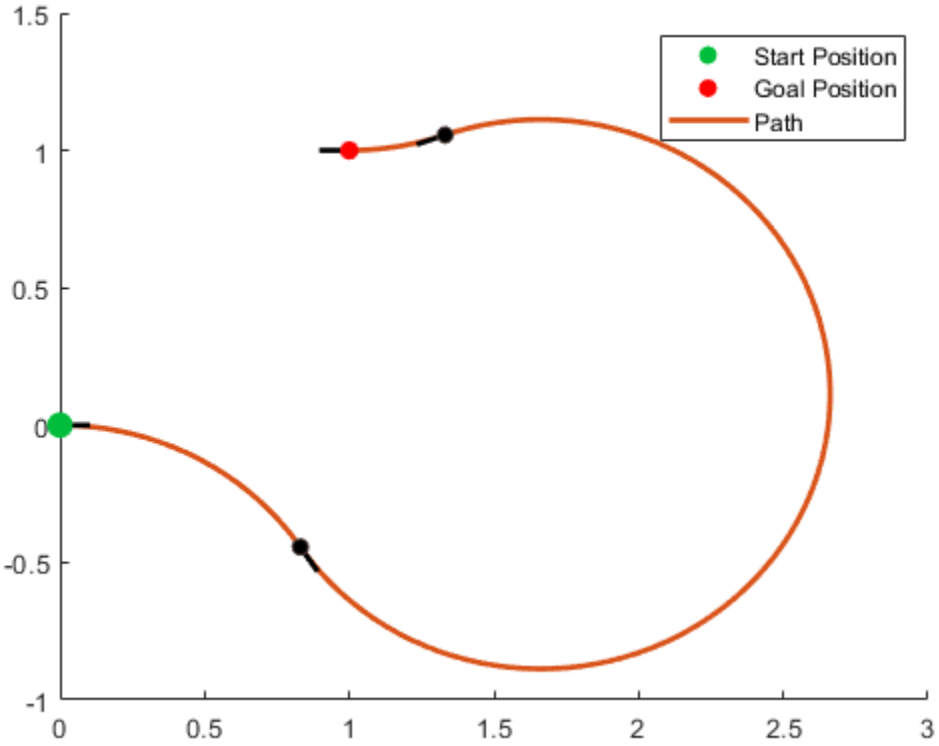
```
startPose = [0 0 0];
goalPose = [1 1 pi];
```

Calculate a valid path segment to connect the poses.

```
[pathSegObj, pathCosts] = connect(dubConnObj,startPose,goalPose);
```

Show the generated path.

```
show(pathSegObj{1})
```



Modify Connection Types for Dubins Path

Create a `dubinsConnection` object.

```
dubConnObj = dubinsConnection;
```

Define start and goal poses as `[x y theta]` vectors.

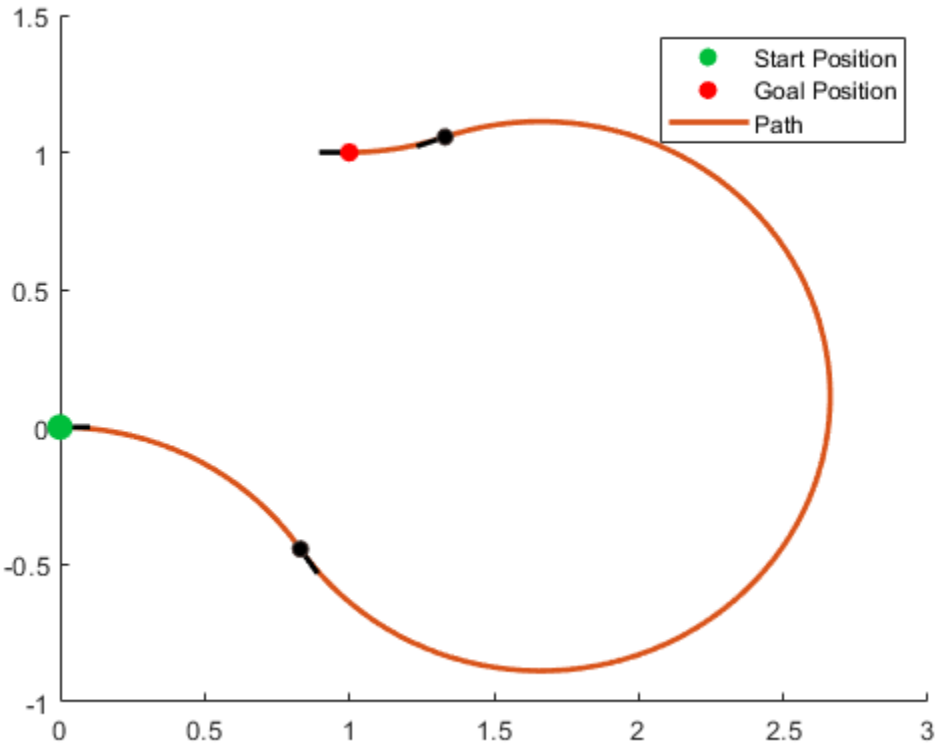
```
startPose = [0 0 0];
goalPose = [1 1 pi];
```

Calculate a valid path segment to connect the poses.

```
pathSegObj = connect(dubConnObj,startPose,goalPose);
```

Show the generated path. Notice the direction of the turns.

```
show(pathSegObj{1})
```




```
pathSegObj{1}.MotionTypes
```

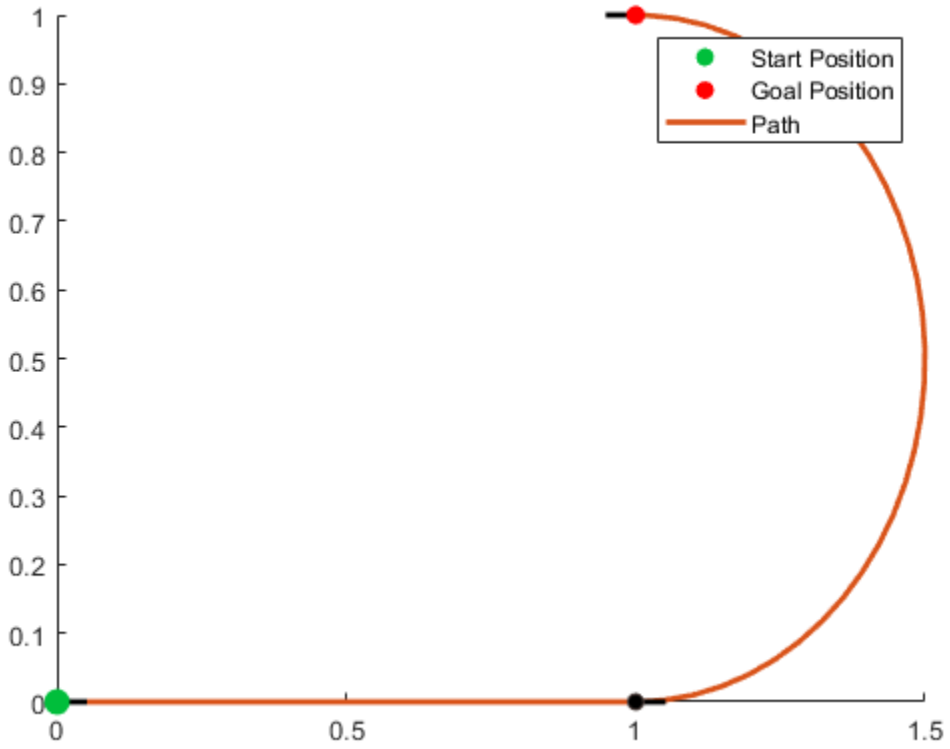
```
ans = 1x3 cell
    {'R'}    {'L'}    {'R'}
```

Disable this specific motion sequence in a new connection object. Reduce the `MinTurningRadius` if the robot is more maneuverable. Connect the poses again to get a different path.

```
dubConnObj = dubinsConnection('DisabledPathTypes',{'RLR'});
dubConnObj.MinTurningRadius = 0.5;

[pathSegObj, pathCosts] = connect(dubConnObj,startPose,goalPose);
pathSegObj{1}.MotionTypes
```

```
ans = 1x3 cell
    {'L'}    {'S'}    {'L'}
```

```
show(pathSegObj{1})
```



References

[1] Shkel, Andrei M., and Vladimir Lumelsky. "Classification of the Dubins set." *Robotics and Autonomous Systems*. Vol. 34, No. 4, 2001, pp. 179–202.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

Objects

`dubinsPathSegment` | `reedsSheppConnection` | `reedsSheppPathSegment`

Functions

`connect` | `interpolate` | `show`

Introduced in R2019b

dubinsPathSegment

Dubins path segment connecting two poses

Description

The `dubinsPathSegment` object holds information for a Dubins path segment that connects two poses as a sequence of three motions. The motion options are:

- Straight
- Left turn at maximum steer
- Right turn at maximum steer

Creation

To generate a `dubinsPathSegment` object, use the `connect` function with a `dubinsConnection` object:

`dubPathSeg = connect(connectionObj,start,goal)` connects the start and goal pose using the specified connection type object.

To specifically define a path segment:

`dubPathSeg = dubinsPathSegment(connectionObj,start,goal,motionLengths, motionTypes)` specifies the Dubins connection type, the start and goal poses, and the corresponding motion lengths and types. These values are set to the corresponding properties in the object.

Properties

`MinTurningRadius` — Minimum turning radius of vehicle
positive scalar

This property is read-only.

Minimum turning radius of the vehicle, specified as a positive scalar in meters. This value corresponds to the radius of the turning circle at the maximum steering angle of the vehicle.

Data Types: `double`

`StartPose` — Initial pose of the vehicle
$[x, y, \Theta]$ vector

This property is read-only.

Initial pose of the vehicle at the start of the path segment, specified as an $[x, y, \Theta]$ vector. x and y are in meters. Θ is in radians.

Data Types: `double`

`GoalPose` — Goal pose of the vehicle

[x, y, Θ] vector

This property is read-only.

Goal pose of the vehicle at the end of the path segment, specified as an [x, y, Θ] vector. x and y are in meters. Θ is in radians.

Data Types: `double`

`MotionLengths` — Length of each motion

three-element numeric vector

This property is read-only.

Length of each motion in the path segment, in world units, specified as a three-element numeric vector. Each motion length corresponds to a motion type specified in `MotionTypes`.

Data Types: `double`

`MotionTypes` — Type of each motion

three-element string cell array

This property is read-only.

Type of each motion in the path segment, specified as a three-element string cell array.

Motion Type	Description
`"S"`	Straight
`"L"`	Left turn at the maximum steering angle of the vehicle
`"R"`	Right turn at the maximum steering angle of the vehicle

Each motion type corresponds to a motion length specified in `MotionLengths`.

For Dubins connections, the available path types are: `{"LSL"} {"LSR"} {"RSL"} {"RSR"} {"RLR"} {"LRL"}`.

Example: `{"R" "S" "R"}`

Data Types: `cell`

`Length` — Length of path segment

positive scalar

This property is read-only.

Length of the path segment, specified as a positive scalar in meters. This length is just a sum of the elements in `MotionLengths`.

Data Types: `double`

Object Functions

interpolate Interpolate poses along path segment

show Visualize path segment

Examples

Connect Poses Using Dubins Connection Path

Create a `dubinsConnection` object.

```
dubConnObj = dubinsConnection;
```

Define start and goal poses as `[x y theta]` vectors.

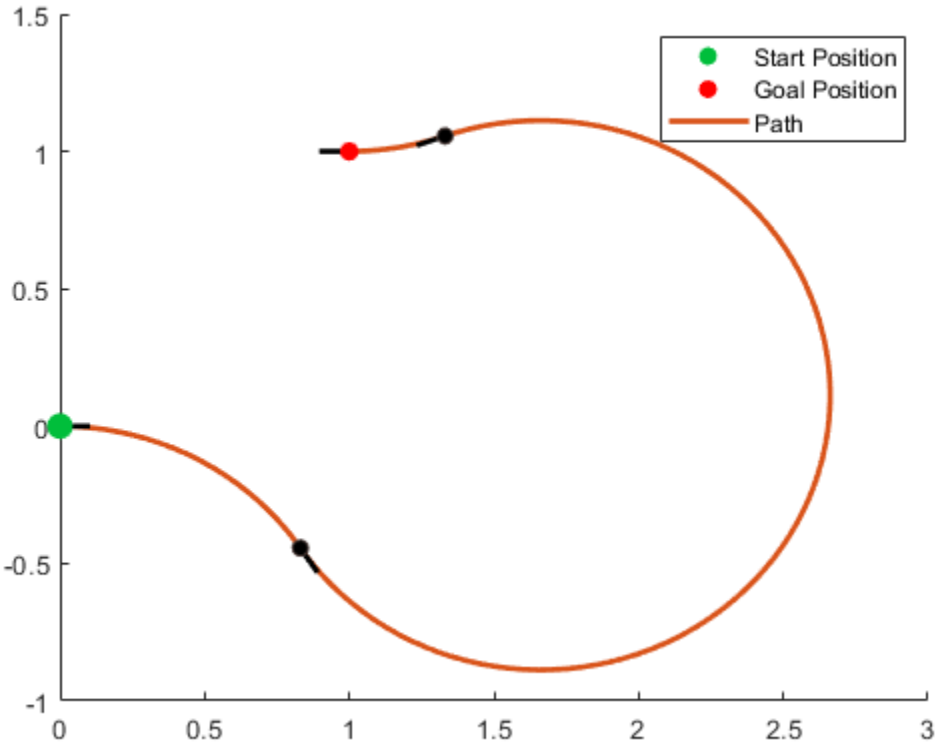
```
startPose = [0 0 0];
goalPose = [1 1 pi];
```

Calculate a valid path segment to connect the poses.

```
[pathSegObj, pathCosts] = connect(dubConnObj,startPose,goalPose);
```

Show the generated path.

```
show(pathSegObj{1})
```



Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

Objects

`dubinsConnection` | `reedsSheppConnection` | `reedsSheppPathSegment`

Functions

`connect` | `interpolate` | `show`

Introduced in R2019b

dynamicCapsuleList

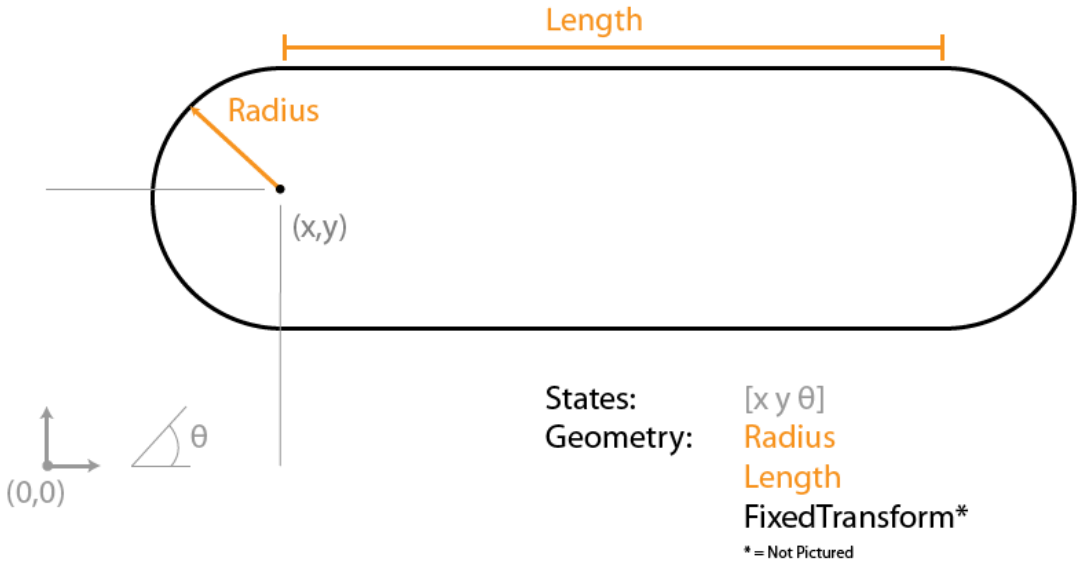
Dynamic capsule-based obstacle list

Description

The `dynamicCapsuleList` object manages two lists of capsule-based collision objects in 2-D space. Collision objects are separated into two lists, ego bodies and obstacles. For ego bodies and obstacles in 3-D, see the `dynamicCapsuleList3D` object.

Each collision object in the two lists has three key elements:

- **ID** –– Integer that identifies each object, stored in the `EgoIDs` property for ego bodies and the `ObstacleIDs` property for obstacles.
- **States** –– Location and orientation of the object as an *M*-by-3 matrix, where each row is of form `[x y theta]` and *M* is the number of states along the path of the object in the world frame. The list of states assumes each state is separated by a fixed time interval. *xy*-positions are in meters, and `theta` is in radians. The default local origin is located at the center of the left semicircle of the capsule.
- **Geometry** –– Size of the capsule-based object based on a specified length and radius. The radius applies to the semicircle end caps, and the length applies to the central rectangle length. To shift the capsule geometry and local origin relative to the default origin point, specify a fixed transform relative to the local frame of the capsule.



Use the object functions to dynamically add, remove, and update the geometries and states of the various objects in your environment. To add an ego body, see the `addEgo` object function. To add an obstacle, see the `addObstacle` object function.

After specifying all of the object states, validate the ego-body paths and check for collisions with obstacles at every step using the `checkCollision` object function. The function only checks if an ego body collides with an obstacle, ignoring collisions between only obstacles or only ego bodies.

Creation

Syntax

```
obstacleList = dynamicCapsuleList
```

Description

`obstacleList = dynamicCapsuleList` creates a dynamic capsule-based obstacle list with no ego bodies or obstacles. To begin building an obstacle list, use the `addEgo` or `addObstacle` object functions.

Properties

`MaxNumSteps` — Maximum number of time steps in obstacle list
31 (default) | positive integer

Maximum number of time steps in the obstacle list, specified as a positive integer. The number of steps determines to the maximum length of the `States` field for a specific ego body or obstacle.

Data Types: `double`

`EgoIDs` — List of IDs for ego bodies
vector of positive integers

This property is read-only.

List of identifiers for ego bodies, returned as a vector of positive integers.

Data Types: `double`

`ObstacleIDs` — IDs for obstacles
vector of positive integers

This property is read-only.

List of identifiers for obstacles, returned as a vector of positive integers.

Data Types: `double`

`NumObstacles` — Number of obstacles in list
integer

This property is read-only.

Number of obstacles in list, returned as an integer.

Data Types: `double`

`NumEgos` — Number of ego bodies in list
integer

This property is read-only.

Number of ego bodies in list, returned as an integer.

Data Types: `double`

Object Functions

addEgo	Add ego bodies to capsule list
addObstacle	Add obstacles to 2-D capsule list
checkCollision	Check for collisions between ego bodies and obstacles
egoGeometry	Geometric properties of ego bodies
egoPose	Poses of ego bodies
obstacleGeometry	Geometric properties of obstacles
obstaclePose	Poses of obstacles
removeEgo	Remove ego bodies from capsule list
removeObstacle	Remove obstacles from capsule list
show	Display ego bodies and obstacles in environment
updateEgoGeometry	Update geometric properties of ego bodies
updateEgoPose	Update states of ego bodies
updateObstacleGeometry	Update geometric properties of obstacles
updateObstaclePose	Update states of obstacles

Examples

Build Ego Body Paths and Check for Collisions with Obstacles

Build an ego body path and maintain obstacle states using the `dynamicCapsuleList` object. Visualize the states of all objects in the environment at different timestamps. Validate the path of the ego body by checking for collisions with obstacles in the environment.

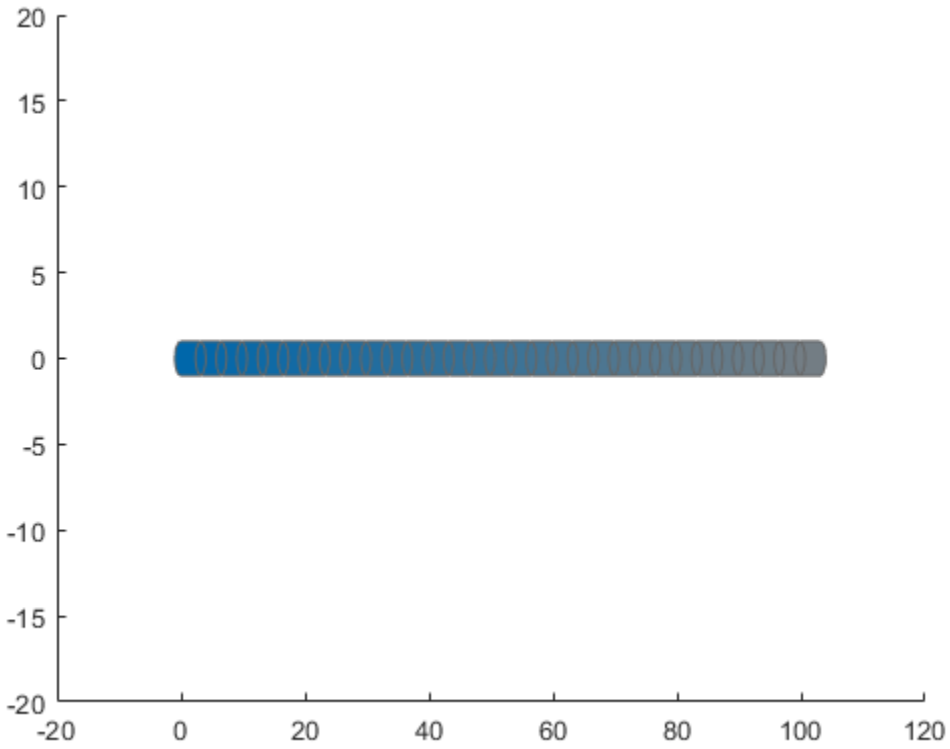
Create the `dynamicCapsuleList` object. Extract the maximum number of steps to use as the number of time stamps for your object paths.

```
obsList = dynamicCapsuleList;
numSteps = obsList.MaxNumSteps;
```

Add Ego Body

Define an ego body by specifying the ID, geometry, and state together in a structure. The capsule geometry has a length of 3 m and radius of 1 m. Specify the state as a linear path from x = 0m to x = 100m.

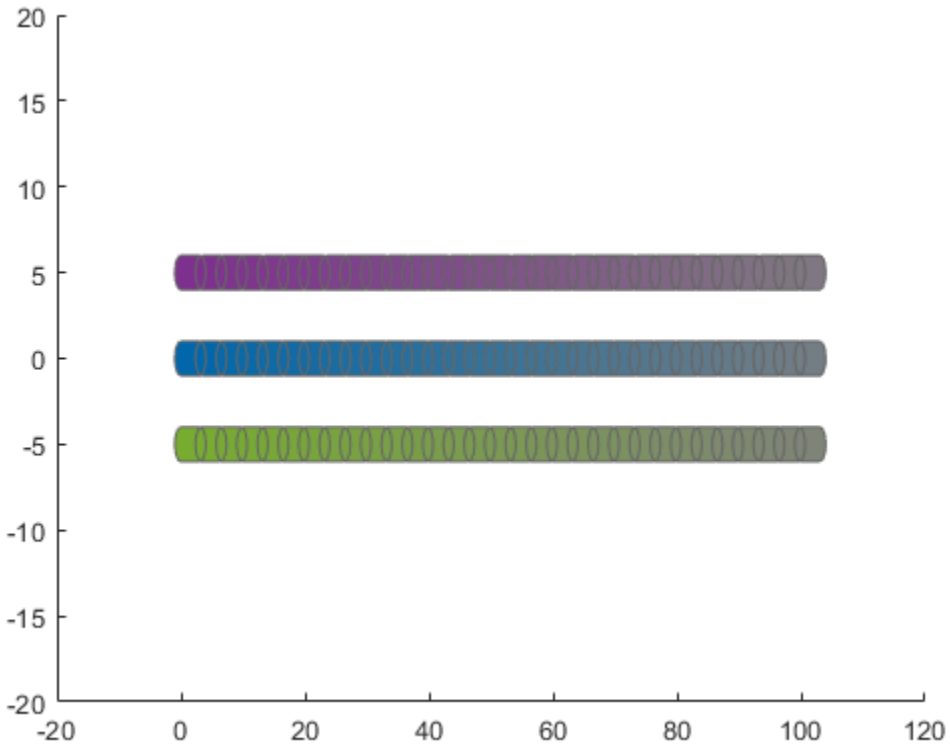
```
egoID1 = 1;
geom = struct("Length",3,"Radius",1,"FixedTransform",eye(3));
states = linspace(0,1,obsList.MaxNumSteps)'.*[100 0 0];

egoCapsule1 = struct('ID',egoID1,'States',states,'Geometry',geom);
addEgo(obsList,egoCapsule1);

show(obsList,"TimeStep",[1:numSteps]);
ylim([-20 20])
```



Add Obstacles

Specify states for two obstacles that are separated from the ego body by 5 m in opposite directions on the *y*-axis.. Assume the obstacles have the same geometry `geom` as the ego body.

```
obsState1 = states + [0 5 0];
obsState2 = states + [0 -5 0];

obsCapsule1 = struct('ID',1,'States',obsState1,'Geometry',geom);
obsCapsule2 = struct('ID',2,'States',obsState2,'Geometry',geom);

addObstacle(obsList,obsCapsule1);
addObstacle(obsList,obsCapsule2);

show(obsList,"TimeStep",[1:numSteps]);
ylim([-20 20])
```



Update Obstacles

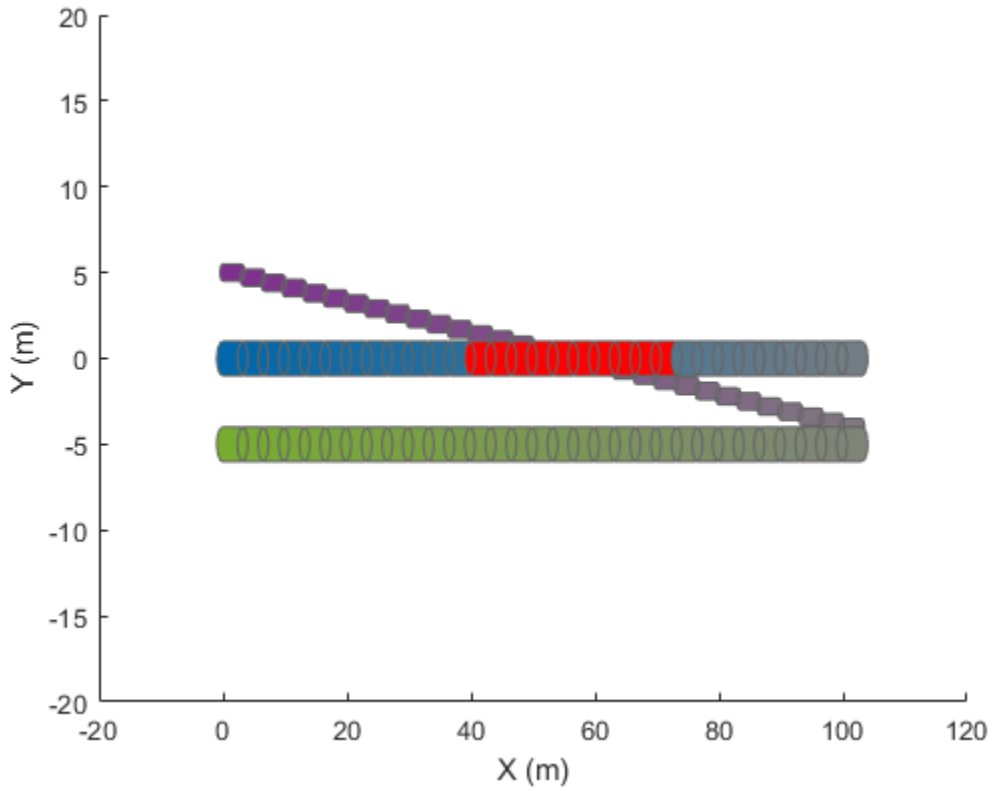
Alter your obstacle locations and geometry dimensions over time. Use the previously generated structure, modify the fields, and update the obstacles using the `updateObstacleGeometry` and `updateObstaclePose` object functions. Reduces the radius of the first obstacle to 0.5 m, and change the path to move it towards the ego body.

```
obsCapsule1.Geometry.Radius = 0.5;

obsCapsule1.States = ...
    [linspace(0,100,numSteps)' ... % x
     linspace(5,-4,numSteps)' ... % y
     zeros(numSteps,1)]; % theta

updateObstacleGeometry(obsList,1,obsCapsule1);
updateObstaclePose(obsList,1,obsCapsule1);
```

Check for Collisions

Visualize the new paths. Show where collisions between the ego body and an obstacle, which the display highlights in red. Notice that collisions between the obstacles are not checked.

```
show(obsList,"TimeStep",[1:numSteps],"ShowCollisions",1);
ylim([-20 20])
xlabel("X (m)")
ylabel("Y (m)")
```



Programmatically check for collisions by using the `checkCollision` object function. The function returns a vector of logical values that indicates the status of each time step. The vector is transposed for display purposes.

```
collisions = checkCollision(obsList)'
```

```
collisions = 1x31 logical array

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

To validate paths with a large number of steps, use the `any` function on the vector of collision values.

```
if any(collisions)
    disp("Collision detected.")
end
```

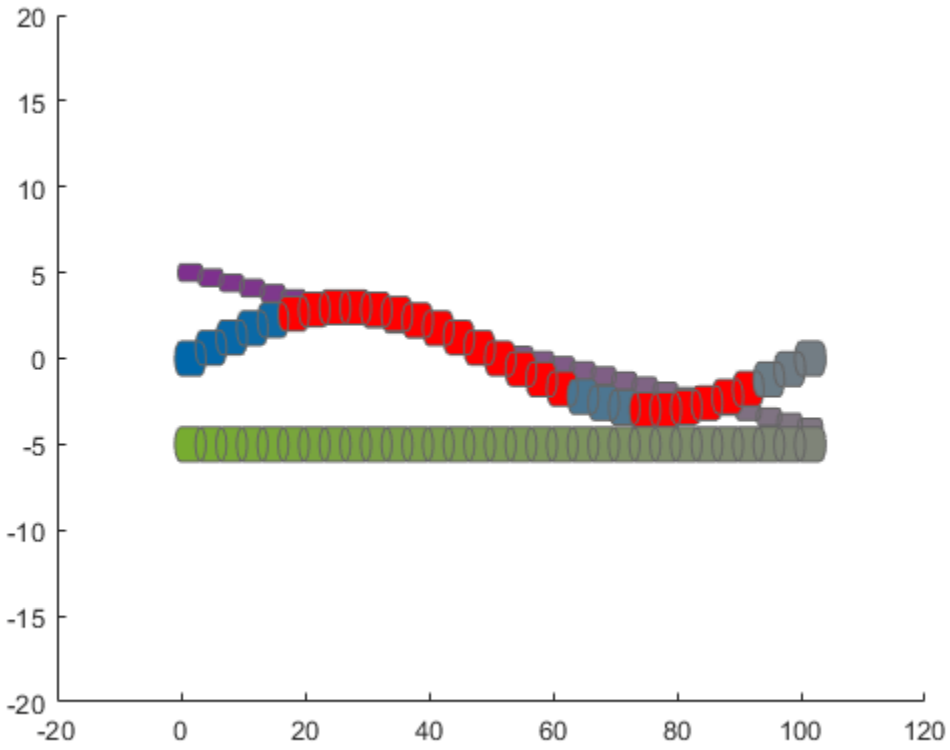
```
Collision detected.
```

Update Ego Path

Specify a new path for the ego body. Visualize the paths again, displaying collisions.

```
egoCapsule1.States = ...
    [linspace(0,100,numSteps)' ... % x
    3*sin(linspace(0,2*pi,numSteps))' ... % y
    zeros(numSteps,1)]; % theta
```

```
updateEgoPose(obsList,1,egoCapsule1);

show(obsList,"TimeStep",[1:numSteps],"ShowCollisions",1);
ylim([-20 20])
```



Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

Objects

`dynamicCapsuleList3D`

Functions

`addEgo` | `addObstacle` | `checkCollision` | `egoGeometry` | `egoPose` | `obstacleGeometry` | `obstaclePose` | `removeEgo` | `removeObstacle` | `show` | `updateEgoGeometry` | `updateEgoPose` | `updateObstacleGeometry` | `updateObstaclePose`

Topics

“Highway Trajectory Planning Using Frenet Reference Path”

Introduced in R2020b

addEgo

Add ego bodies to capsule list

Syntax

```
addEgo(capsuleListObj,egoStruct)
status = addEgo(capsuleListObj,egoStruct)
```

Description

`addEgo(capsuleListObj,egoStruct)` adds one or more ego bodies to the 2-D dynamic capsule list with the specified ID, state, and geometry values given in `egoStruct`.

`status = addEgo(capsuleListObj,egoStruct)` additionally returns an indicator of whether each specified ego body was added, updated, or a duplicate.

Examples

Build Ego Body Paths and Check for Collisions with Obstacles

Build an ego body path and maintain obstacle states using the `dynamicCapsuleList` object. Visualize the states of all objects in the environment at different timestamps. Validate the path of the ego body by checking for collisions with obstacles in the environment.

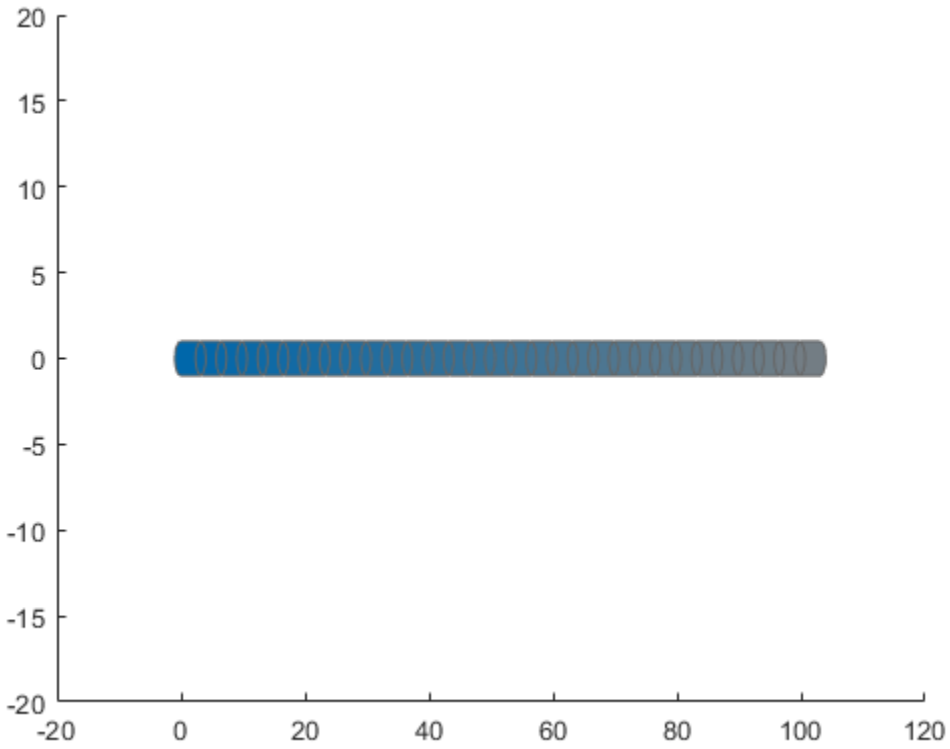
Create the `dynamicCapsuleList` object. Extract the maximum number of steps to use as the number of time stamps for your object paths.

```
obsList = dynamicCapsuleList;
numSteps = obsList.MaxNumSteps;
```

Add Ego Body

Define an ego body by specifying the ID, geometry, and state together in a structure. The capsule geometry has a length of 3 m and radius of 1 m. Specify the state as a linear path from $x = 0$m to $x = 100$m.

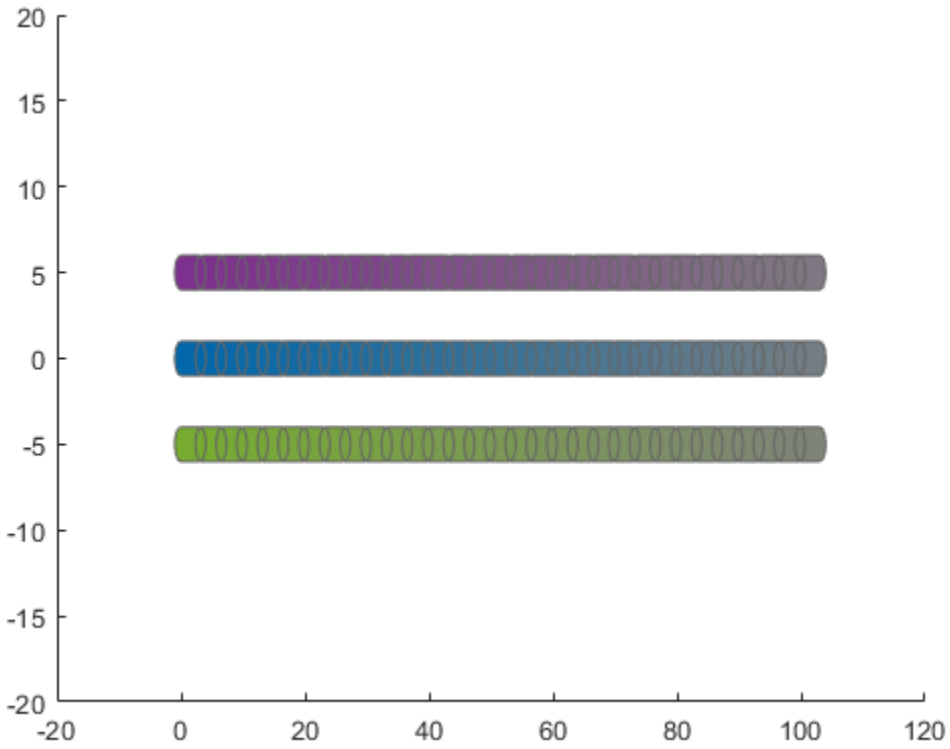
```
egoID1 = 1;
geom = struct("Length",3,"Radius",1,"FixedTransform",eye(3));
states = linspace(0,1,obsList.MaxNumSteps)'.*[100 0 0];

egoCapsule1 = struct('ID',egoID1,'States',states,'Geometry',geom);
addEgo(obsList,egoCapsule1);

show(obsList,"TimeStep",[1:numSteps]);
ylim([-20 20])
```



Add Obstacles

Specify states for two obstacles that are separated from the ego body by 5 m in opposite directions on the *y*-axis.. Assume the obstacles have the same geometry `geom` as the ego body.

```
obsState1 = states + [0 5 0];
obsState2 = states + [0 -5 0];

obsCapsule1 = struct('ID',1,'States',obsState1,'Geometry',geom);
obsCapsule2 = struct('ID',2,'States',obsState2,'Geometry',geom);

addObstacle(obsList,obsCapsule1);
addObstacle(obsList,obsCapsule2);

show(obsList,"TimeStep",[1:numSteps]);
ylim([-20 20])
```



Update Obstacles

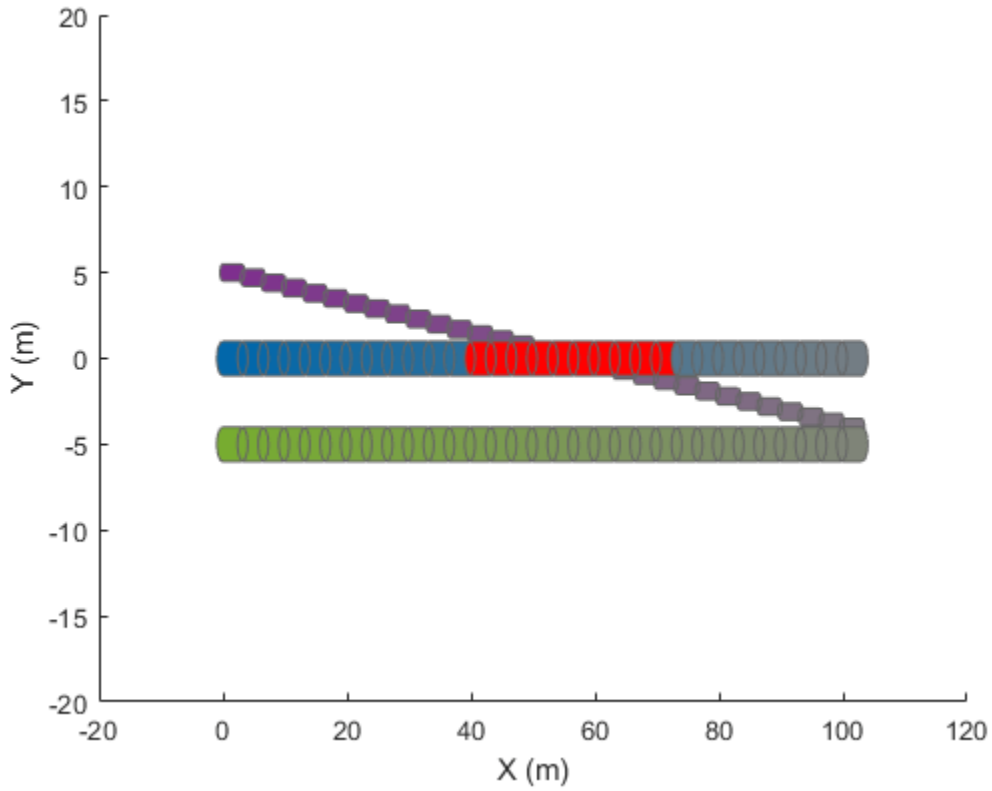
Alter your obstacle locations and geometry dimensions over time. Use the previously generated structure, modify the fields, and update the obstacles using the `updateObstacleGeometry` and `updateObstaclePose` object functions. Reduces the radius of the first obstacle to 0.5 m, and change the path to move it towards the ego body.

```
obsCapsule1.Geometry.Radius = 0.5;

obsCapsule1.States = ...
    [linspace(0,100,numSteps)' ... % x
     linspace(5,-4,numSteps)' ... % y
     zeros(numSteps,1)]; % theta

updateObstacleGeometry(obsList,1,obsCapsule1);
updateObstaclePose(obsList,1,obsCapsule1);
```

Check for Collisions

Visualize the new paths. Show where collisions between the ego body and an obstacle, which the display highlights in red. Notice that collisions between the obstacles are not checked.

```
show(obsList,"TimeStep",[1:numSteps],"ShowCollisions",1);
ylim([-20 20])
xlabel("X (m)")
ylabel("Y (m)")
```



Programmatically check for collisions by using the `checkCollision` object function. The function returns a vector of logical values that indicates the status of each time step. The vector is transposed for display purposes.

```
collisions = checkCollision(obsList)'
```

```
collisions = 1x31 logical array

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

To validate paths with a large number of steps, use the `any` function on the vector of collision values.

```
if any(collisions)
    disp("Collision detected.")
end
```

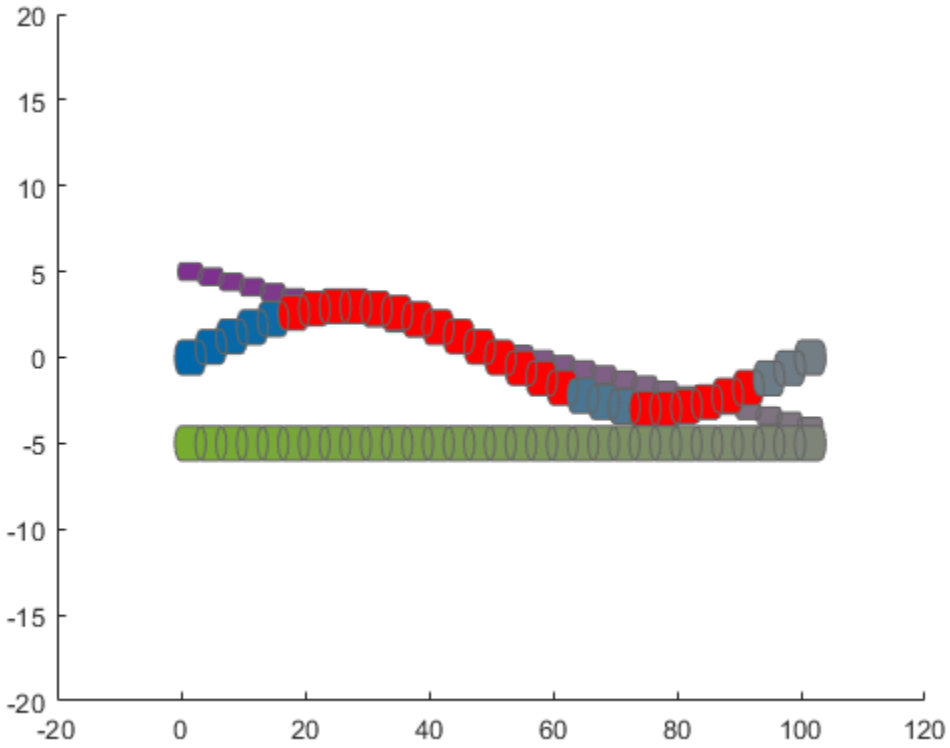
```
Collision detected.
```

Update Ego Path

Specify a new path for the ego body. Visualize the paths again, displaying collisions.

```
egoCapsule1.States = ...
    [linspace(0,100,numSteps)' ... % x
    3*sin(linspace(0,2*pi,numSteps))' ... % y
    zeros(numSteps,1)]; % theta
```

```
updateEgoPose(obsList,1,egoCapsule1);

show(obsList,"TimeStep",[1:numSteps],"ShowCollisions",1);
ylim([-20 20])
```



Input Arguments

`capsuleListObj` — Dynamic capsule list

`dynamicCapsuleList` object

Dynamic capsule list, specified as a `dynamicCapsuleList` object.

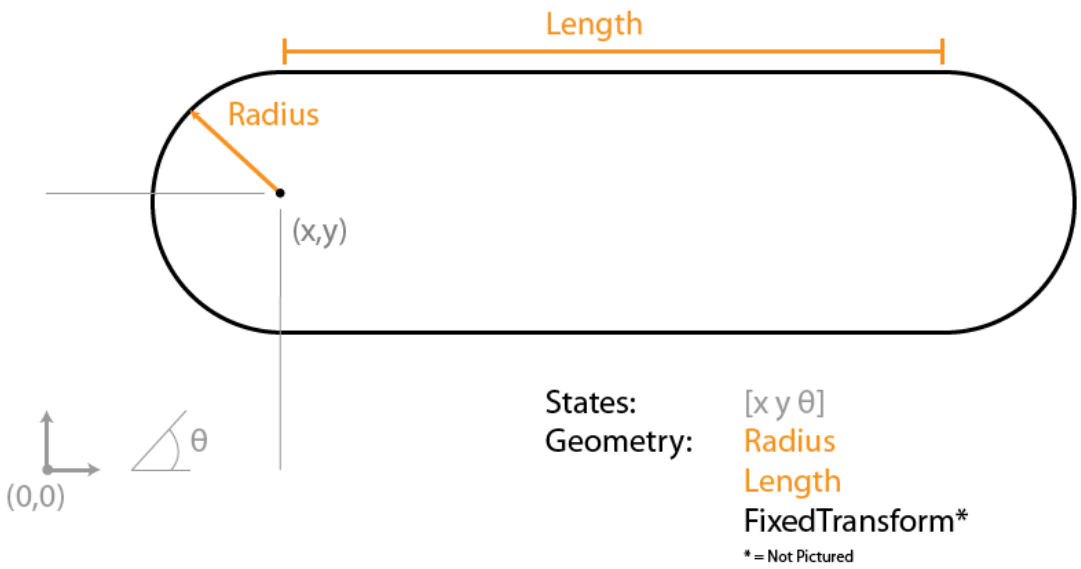
`egoStruct` — Ego body parameters

structure | *N*-element structure array

Ego body parameters, specified as an *N*-element structure or a structure array, where *N* is the number of added ego bodies. The fields of each structure define the ID, geometry, and states of an ego body:

- `ID` -– Integer that identifies each object. Stored in the `EgoIDs` property of the `dynamicCapsuleList` object specified by the `capsuleListObj` argument.
- `States` -– Location and orientation of the object as an *M*-by-3 matrix, where each row is of form `[x y theta]`, and *M* is the number of states for the specified ego body in the world frame. The list of states assumes each state is separated by a fixed time interval. *xy*-positions are in meters and `theta` is in radians.

- `Geometry` -- Structure with fields `Length`, `Radius`, and `FixedTransform`. These fields define the size of the capsule-based object using the specified length for the cylinder and semicircle radius for the end caps. To shift the capsule geometry from the default origin, specify the `FixedTransform` field as a fixed transform relative to the local frame of the capsule. To keep the default capsule origin, specify the transform as `eye(3)`.



Output Arguments

`status` — Result of adding ego bodies

N-element column vector

Result of adding ego bodies, returned as a *N*-element column vector of ones, zeros, and negative ones. *N* is the number of ego bodies specified in the `egoStruct` argument. Each value indicates whether the associated body is added (`1`), updated (`0`), or a duplicate (`-1`). While adding ego bodies, if multiple structures with the same body ID are found in the structure array `egoStruct`, then the function marks the previous entry as duplicate and ignores it.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

Objects

`dynamicCapsuleList` | `dynamicCapsuleList3D`

Functions
`addObstacle` | `checkCollision` | `egoGeometry` | `egoPose` | `obstacleGeometry` | `obstaclePose` | `removeEgo` | `removeObstacle` | `show` | `updateEgoGeometry` | `updateEgoPose` | `updateObstacleGeometry` | `updateObstaclePose`

Topics
“Highway Trajectory Planning Using Frenet Reference Path”

Introduced in R2020b

addObstacle

Add obstacles to 2-D capsule list

Syntax

```
addObstacle(capsuleListObj,obstacleStruct)
status = addObstacle(capsuleListObj,obstacleStruct)
```

Description

`addObstacle(capsuleListObj,obstacleStruct)` adds one or more obstacles to the 2-D dynamic capsule list with the specified ID, state, and geometry values given in `obstacleStruct`.

`status = addObstacle(capsuleListObj,obstacleStruct)` additionally returns an indicator of whether each specified obstacle was added, updated, or a duplicate.

Examples

Build Ego Body Paths and Check for Collisions with Obstacles

Build an ego body path and maintain obstacle states using the `dynamicCapsuleList` object. Visualize the states of all objects in the environment at different timestamps. Validate the path of the ego body by checking for collisions with obstacles in the environment.

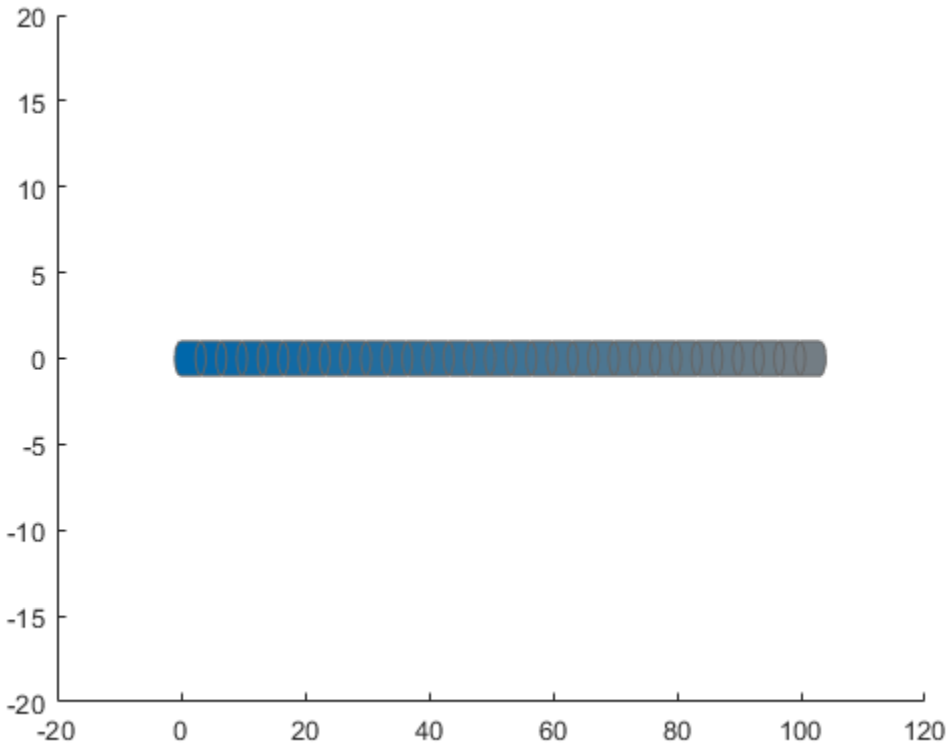
Create the `dynamicCapsuleList` object. Extract the maximum number of steps to use as the number of time stamps for your object paths.

```
obsList = dynamicCapsuleList;
numSteps = obsList.MaxNumSteps;
```

Add Ego Body

Define an ego body by specifying the ID, geometry, and state together in a structure. The capsule geometry has a length of 3 m and radius of 1 m. Specify the state as a linear path from x = 0m to x = 100m.

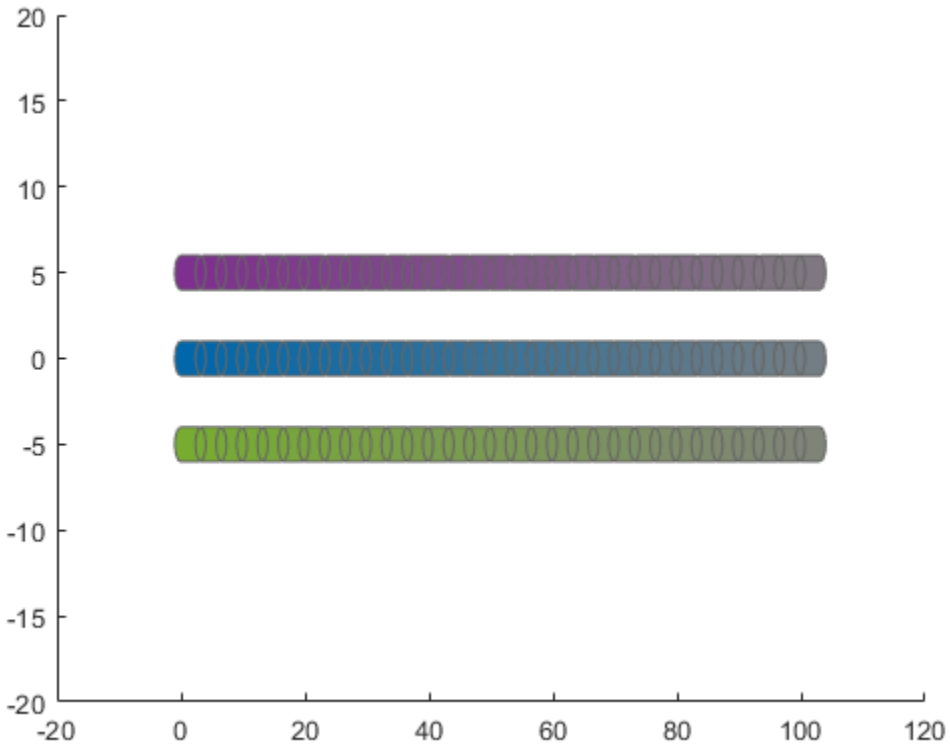
```
egoID1 = 1;
geom = struct("Length",3,"Radius",1,"FixedTransform",eye(3));
states = linspace(0,1,obsList.MaxNumSteps)'.*[100 0 0];

egoCapsule1 = struct('ID',egoID1,'States',states,'Geometry',geom);
addEgo(obsList,egoCapsule1);

show(obsList,"TimeStep",[1:numSteps]);
ylim([-20 20])
```



Add Obstacles

Specify states for two obstacles that are separated from the ego body by 5 m in opposite directions on the *y*-axis.. Assume the obstacles have the same geometry `geom` as the ego body.

```
obsState1 = states + [0 5 0];
obsState2 = states + [0 -5 0];

obsCapsule1 = struct('ID',1,'States',obsState1,'Geometry',geom);
obsCapsule2 = struct('ID',2,'States',obsState2,'Geometry',geom);

addObstacle(obsList,obsCapsule1);
addObstacle(obsList,obsCapsule2);

show(obsList,"TimeStep",[1:numSteps]);
ylim([-20 20])
```



Update Obstacles

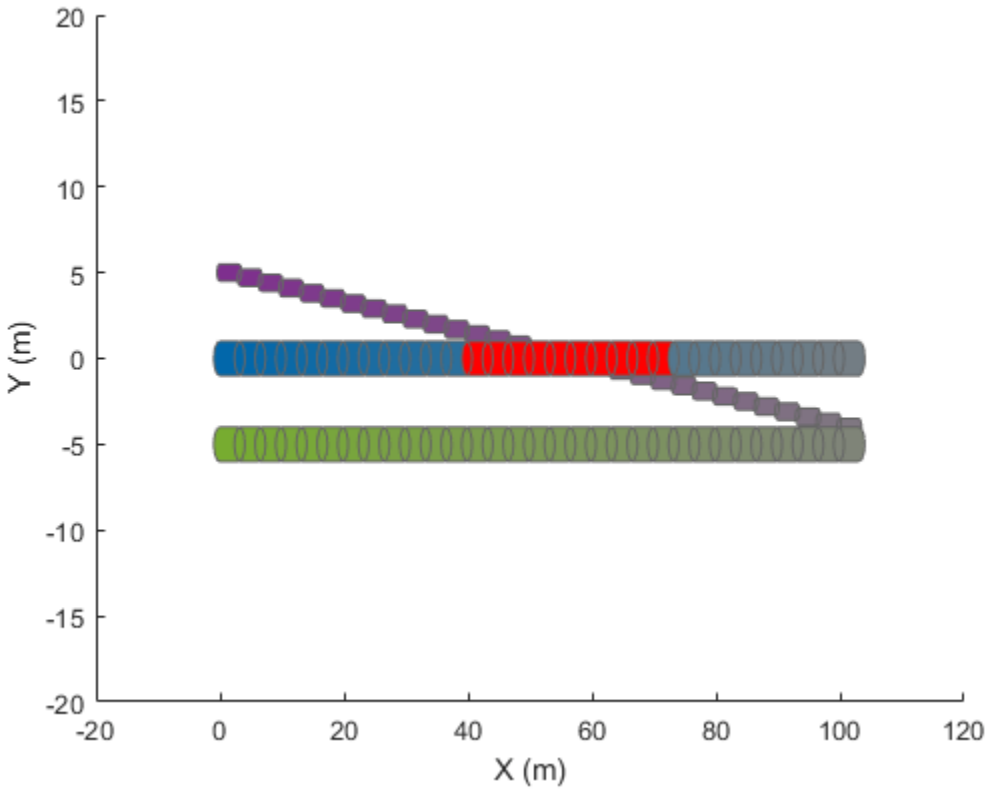
Alter your obstacle locations and geometry dimensions over time. Use the previously generated structure, modify the fields, and update the obstacles using the `updateObstacleGeometry` and `updateObstaclePose` object functions. Reduces the radius of the first obstacle to 0.5 m, and change the path to move it towards the ego body.

```
obsCapsule1.Geometry.Radius = 0.5;

obsCapsule1.States = ...
    [linspace(0,100,numSteps)' ... % x
     linspace(5,-4,numSteps)' ... % y
     zeros(numSteps,1)]; % theta

updateObstacleGeometry(obsList,1,obsCapsule1);
updateObstaclePose(obsList,1,obsCapsule1);
```

Check for Collisions

Visualize the new paths. Show where collisions between the ego body and an obstacle, which the display highlights in red. Notice that collisions between the obstacles are not checked.

```
show(obsList,"TimeStep",[1:numSteps],"ShowCollisions",1);
ylim([-20 20])
xlabel("X (m)")
ylabel("Y (m)")
```



Programmatically check for collisions by using the `checkCollision` object function. The function returns a vector of logical values that indicates the status of each time step. The vector is transposed for display purposes.

```
collisions = checkCollision(obsList)'
```

```
collisions = 1x31 logical array

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

To validate paths with a large number of steps, use the `any` function on the vector of collision values.

```
if any(collisions)
    disp("Collision detected.")
end
```

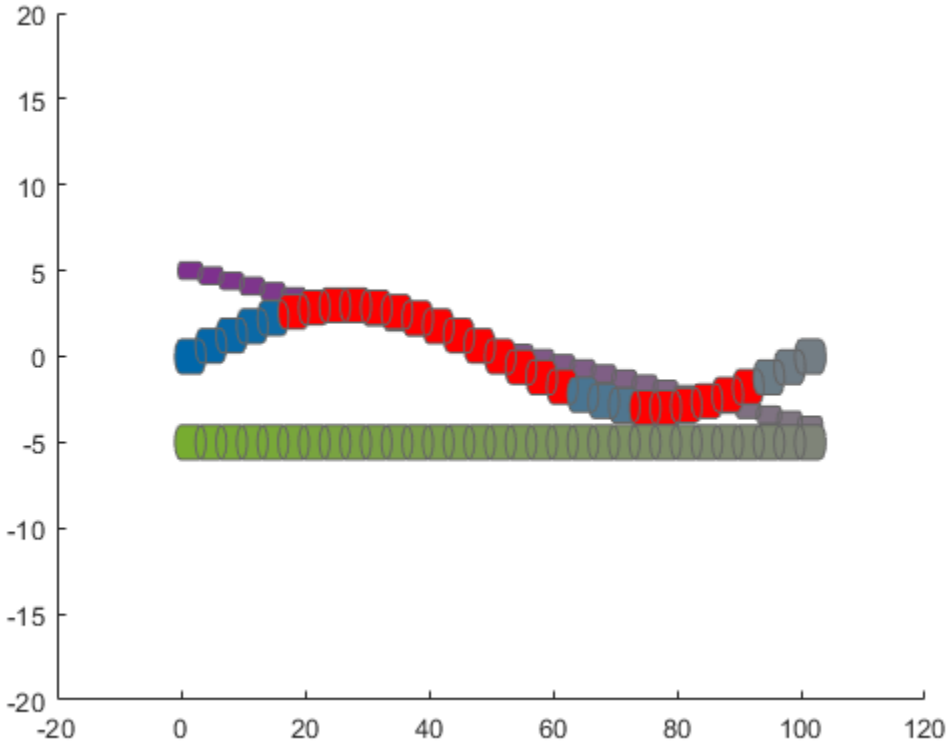
```
Collision detected.
```

Update Ego Path

Specify a new path for the ego body. Visualize the paths again, displaying collisions.

```
egoCapsule1.States = ...
    [linspace(0,100,numSteps)' ... % x
    3*sin(linspace(0,2*pi,numSteps))' ... % y
    zeros(numSteps,1)]; % theta
```

```
updateEgoPose(obsList,1,egoCapsule1);

show(obsList,"TimeStep",[1:numSteps],"ShowCollisions",1);
ylim([-20 20])
```



Input Arguments

`capsuleListObj` — Dynamic capsule list

`dynamicCapsuleList` object

Dynamic capsule list, specified as a `dynamicCapsuleList` object.

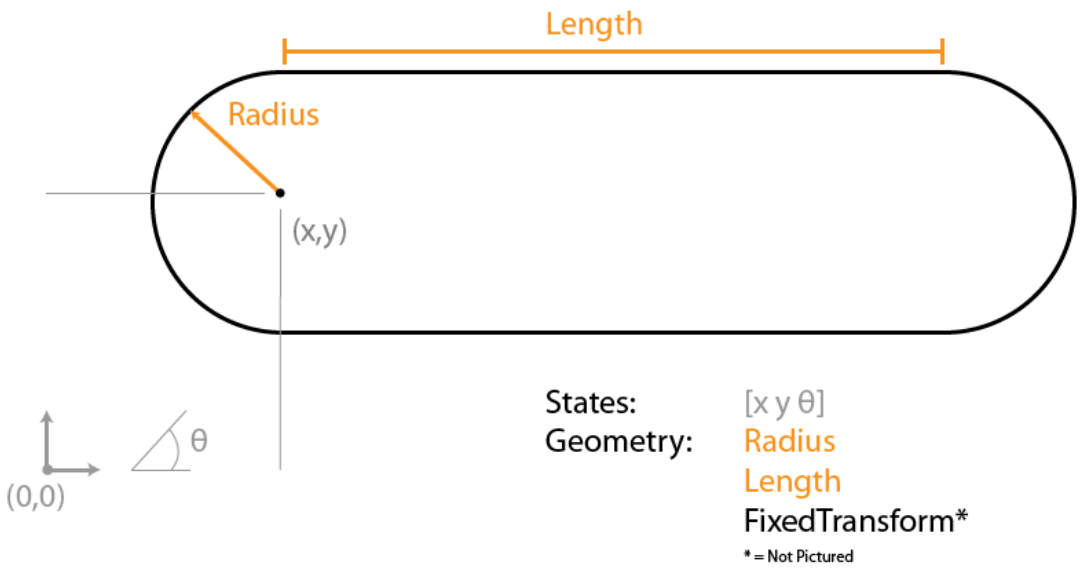
`obstacleStruct` — Obstacle parameters

structure | *N*-element structure array

Obstacle parameters, specified as an *N*-element structure or a structure array, where *N* is the number of added obstacles. The fields of each structure define the ID, geometry, and states of an obstacle:

- `ID` -– Integer that identifies each object. Stored in the `ObstacleIDs` property of the `dynamicCapsuleList` object specified by the `capsuleListObj` argument.
- `States` -– Location and orientation of the object as an *M*-by-3 matrix, where each row is of form `[x y theta]`, and *M* is the number of states for the specified obstacle in the world frame. The list of states assumes each state is separated by a fixed time interval. *xy*-positions are in meters and `theta` is in radians.

- `Geometry` -- Structure with fields `Length`, `Radius`, and `FixedTransform`. These fields define the size of the capsule-based object using the specified length for the cylinder and semicircle radius for the end caps. To shift the capsule geometry from the default origin, specify the `FixedTransform` field as a fixed transform relative to the local frame of the capsule. To keep the default capsule origin, specify the transform as `eye(3)`.



Output Arguments

`status` — Result of adding obstacles

N-element column vector

Result of adding obstacles, returned as a *N*-element column vector of ones, zeros, and negative ones. *N* is the number of obstacles specified in the `obstacleStruct` argument. Each value indicates whether the associated body is added (`1`), updated (`0`), or a duplicate (`-1`). While adding obstacles, if multiple structures with the same body ID are found in the structure array `obstaclesStruct`, then the function marks the previous entry as duplicate and ignores it.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

Objects

`dynamicCapsuleList` | `dynamicCapsuleList3D`

Functions
addEgo | checkCollision | egoGeometry | egoPose | obstacleGeometry | obstaclePose | removeEgo | removeObstacle | show | updateEgoGeometry | updateEgoPose | updateObstacleGeometry | updateObstaclePose

Topics
“Highway Trajectory Planning Using Frenet Reference Path”

Introduced in R2020b

checkCollision

Check for collisions between ego bodies and obstacles

Syntax

```
collisionFound = checkCollision(capsuleListObj)
[fullResults,distance] = checkCollision(capsuleListObj,options)
```

Description

`collisionFound = checkCollision(capsuleListObj)` checks each ego body for collisions with obstacles in the environment. The function indicates whether each ego body is in collision at each time step..

`[fullResults,distance] = checkCollision(capsuleListObj,options)` checks each ego body for collisions with obstacles in the environment, and returns the results using additional specified collision detection options `options`.

Examples

Build Ego Body Paths and Check for Collisions with Obstacles

Build an ego body path and maintain obstacle states using the `dynamicCapsuleList` object. Visualize the states of all objects in the environment at different timestamps. Validate the path of the ego body by checking for collisions with obstacles in the environment.

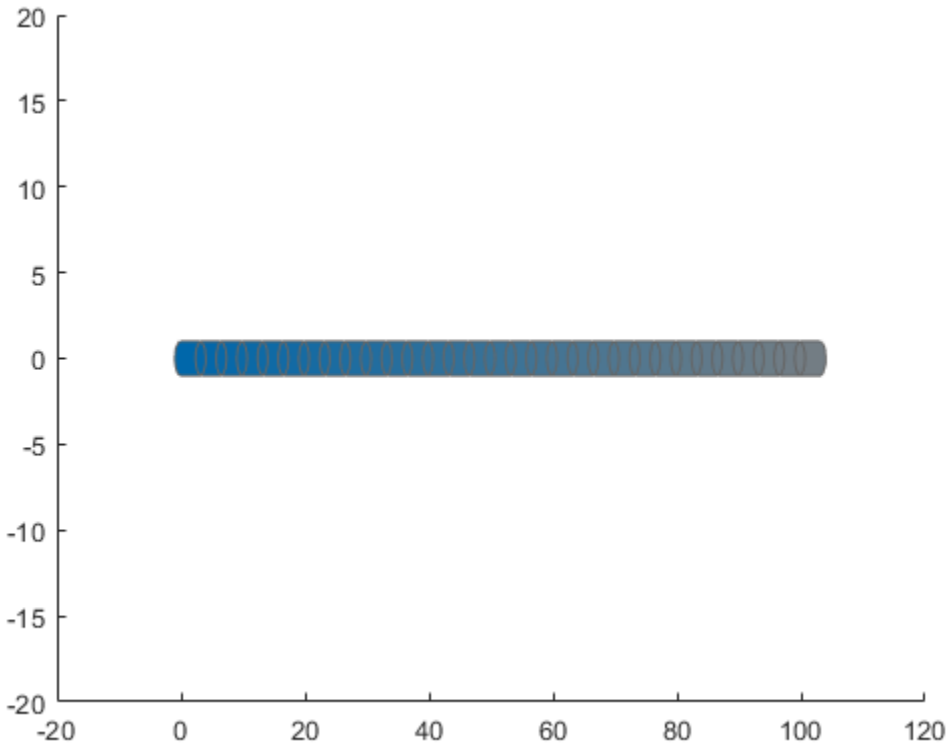
Create the `dynamicCapsuleList` object. Extract the maximum number of steps to use as the number of time stamps for your object paths.

```
obsList = dynamicCapsuleList;
numSteps = obsList.MaxNumSteps;
```

Add Ego Body

Define an ego body by specifying the ID, geometry, and state together in a structure. The capsule geometry has a length of 3 m and radius of 1 m. Specify the state as a linear path from x = 0m to x = 100m.

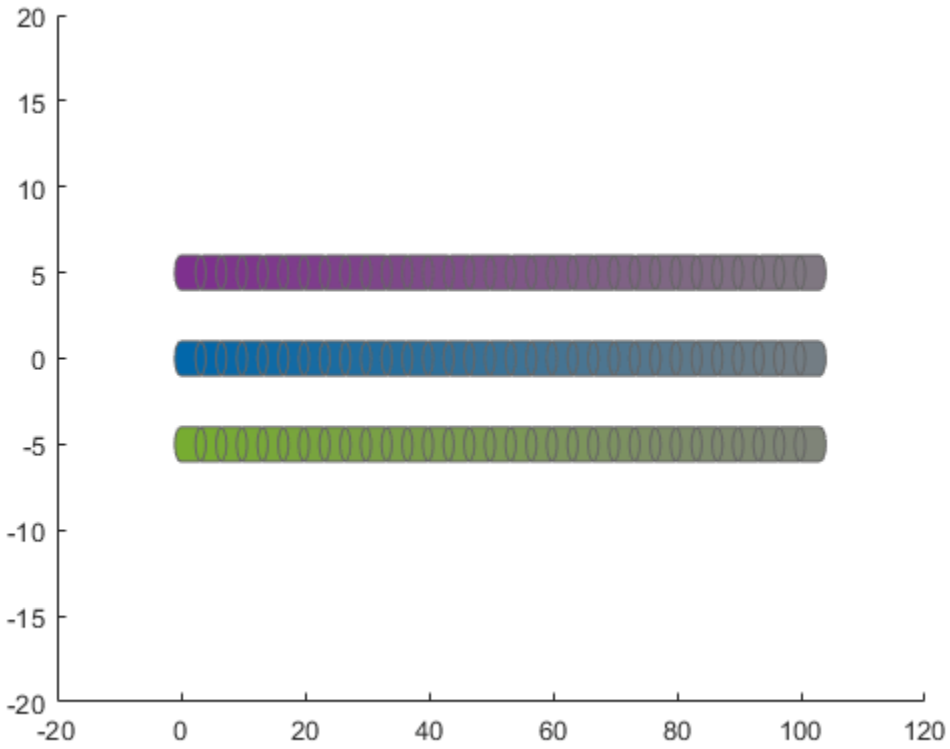
```
egoID1 = 1;
geom = struct("Length",3,"Radius",1,"FixedTransform",eye(3));
states = linspace(0,1,obsList.MaxNumSteps)'.*[100 0 0];

egoCapsule1 = struct('ID',egoID1,'States',states,'Geometry',geom);
addEgo(obsList,egoCapsule1);

show(obsList,"TimeStep",[1:numSteps]);
ylim([-20 20])
```



Add Obstacles

Specify states for two obstacles that are separated from the ego body by 5 m in opposite directions on the *y*-axis.. Assume the obstacles have the same geometry `geom` as the ego body.

```
obsState1 = states + [0 5 0];
obsState2 = states + [0 -5 0];

obsCapsule1 = struct('ID',1,'States',obsState1,'Geometry',geom);
obsCapsule2 = struct('ID',2,'States',obsState2,'Geometry',geom);

addObstacle(obsList,obsCapsule1);
addObstacle(obsList,obsCapsule2);

show(obsList,"TimeStep",[1:numSteps]);
ylim([-20 20])
```



Update Obstacles

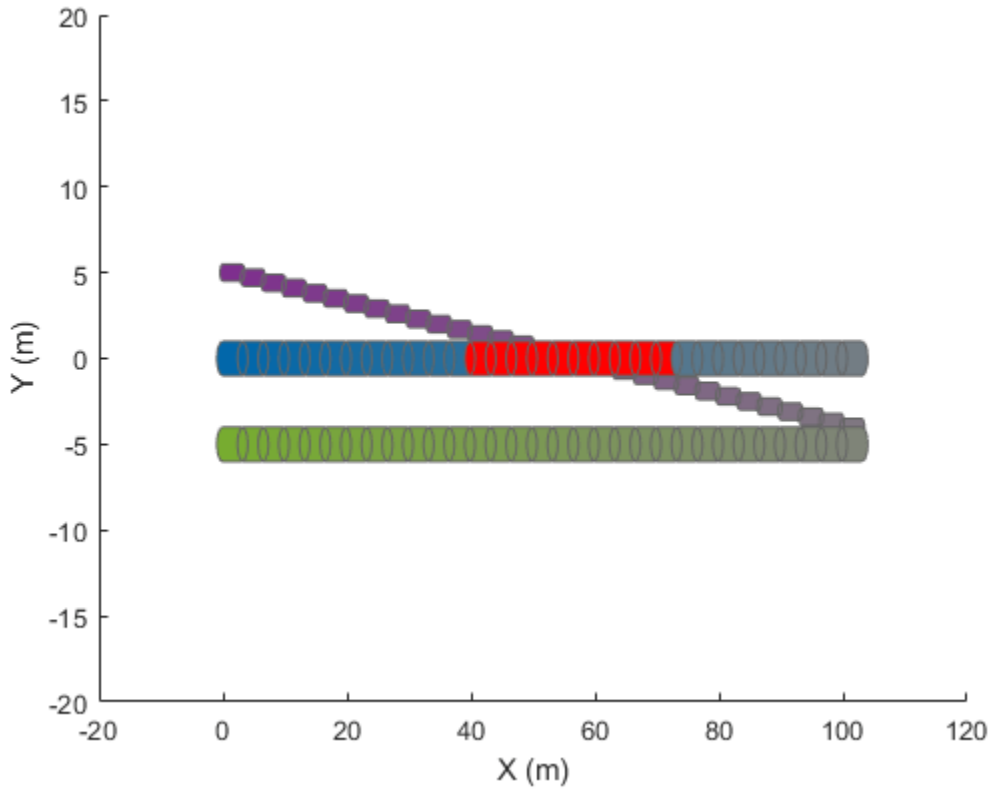
Alter your obstacle locations and geometry dimensions over time. Use the previously generated structure, modify the fields, and update the obstacles using the `updateObstacleGeometry` and `updateObstaclePose` object functions. Reduces the radius of the first obstacle to 0.5 m, and change the path to move it towards the ego body.

```
obsCapsule1.Geometry.Radius = 0.5;

obsCapsule1.States = ...
    [linspace(0,100,numSteps)' ... % x
     linspace(5,-4,numSteps)' ... % y
     zeros(numSteps,1)]; % theta

updateObstacleGeometry(obsList,1,obsCapsule1);
updateObstaclePose(obsList,1,obsCapsule1);
```

Check for Collisions

Visualize the new paths. Show where collisions between the ego body and an obstacle, which the display highlights in red. Notice that collisions between the obstacles are not checked.

```
show(obsList,"TimeStep",[1:numSteps],"ShowCollisions",1);
ylim([-20 20])
xlabel("X (m)")
ylabel("Y (m)")
```



Programmatically check for collisions by using the `checkCollision` object function. The function returns a vector of logical values that indicates the status of each time step. The vector is transposed for display purposes.

```
collisions = checkCollision(obsList)'
```

```
collisions = 1x31 logical array

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

To validate paths with a large number of steps, use the `any` function on the vector of collision values.

```
if any(collisions)
    disp("Collision detected.")
end
```

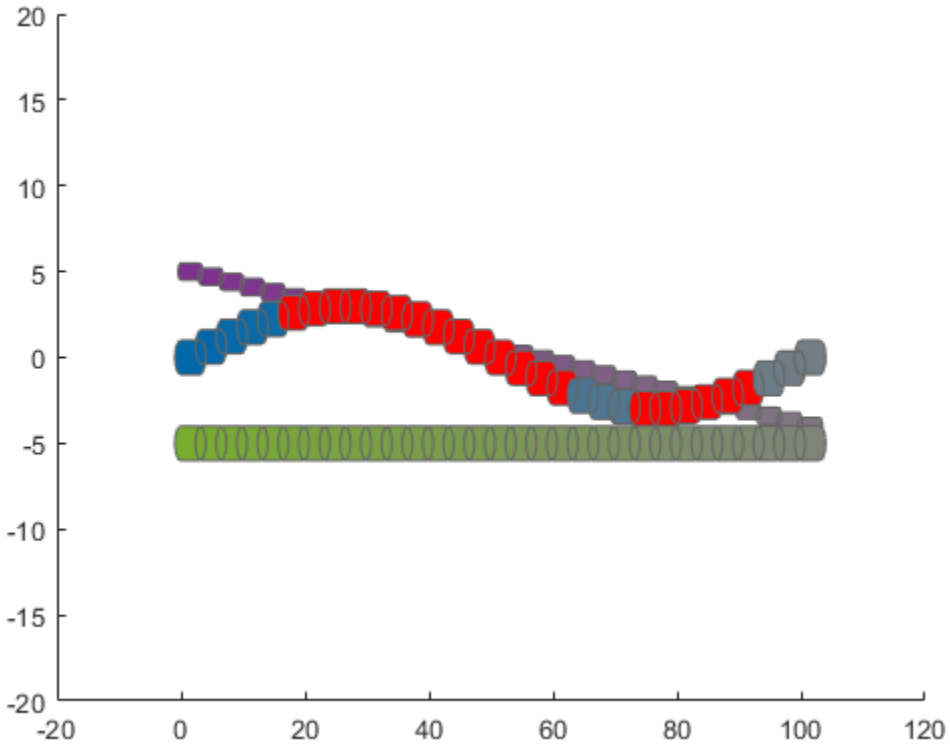
```
Collision detected.
```

Update Ego Path

Specify a new path for the ego body. Visualize the paths again, displaying collisions.

```
egoCapsule1.States = ...
    [linspace(0,100,numSteps)' ... % x
    3*sin(linspace(0,2*pi,numSteps))' ... % y
    zeros(numSteps,1)]; % theta
```

```
updateEgoPose(obsList,1,egoCapsule1);

show(obsList,"TimeStep",[1:numSteps],"ShowCollisions",1);
ylim([-20 20])
```



Input Arguments

`capsuleListObj` — Dynamic capsule list

`dynamicCapsuleList` object | `dynamicCapsuleList3D` object

Dynamic capsule list, specified as a `dynamicCapsuleList` or `dynamicCapsuleList3D` object.

`options` — Collision detection options

structure

Collision detection options, specified as a structure with these fields:

- `FullResults` -- Return the collision results for each obstacle separately , specified as a logical `0` (`false`) or `1` (`true`). See the `fullResults` output argument.
- `ReturnDistance` -- Return the distance calculation from collision checking, specified as a logical `0` (`false`) or `1` (`true`). See the `distance` output argument.

Data Types: `struct`

Output Arguments

`collisionFound` — Collision checking results

n-by-*e* matrix of logical values

Collision checking results, returned as an *n*-by-*e* matrix of logical values. By default, the function checks for any collision between any object, which returns an *n*-by-*e* matrix, where *n* is the maximum number of states for ego bodies in the specified `capsuleListobj` object, and *e* is the number of ego bodies.

Data Types: `logical`

`fullResults` — Full collision checking results for each obstacle

n-by-*o*-by-*e* array of logical values

Full collision checking results for each obstacle, returned as an *n*-by-*o*-by-*e* array of logical values. *n* is the maximum number of states for ego bodies in the specified `capsuleListobj` argument, *o* is the number of obstacles, and *e* is the number of ego bodies.

Dependencies

To return the `fullResults` output argument, specify the `options` input argument with the `FullResults` field set to `true`.

Data Types: `logical`

`distance` — Distance from obstacles

n-by-*e* numeric matrix | *n*-by-*o*by-*e* numeric array

Distance from obstacles, returned as an *n*-by-*e* numeric matrix or *n*-by-*o*by-*e* numeric array. The dimensions and behavior of the distance argument depend on the value of the `FullResults` field of the options argument

`distance` Dimensions	`FullResults` Value	Behavior
n-by-*e* numeric matrix	`false`	Returns the distance between each ego body and the closest obstacle at each time step. *n* is the maximum number of states for ego bodies specified in the `capsuleListObj` argument, and *e* is the number of ego bodies.
n-by-*o*by-*e* numeric array	`true`	Returns the distance between each ego body and each obstacle at each time step. *o* is the number of obstacles.

Dependencies

To return the `distance` output argument, specify the `options` input argument with the `ReturnDistance` field set to `true`.

Data Types: `single` | `double`

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

Objects
`dynamicCapsuleList` | `dynamicCapsuleList3D`

Functions
`addEgo` | `addObstacle` | `egoGeometry` | `egoPose` | `obstacleGeometry` | `obstaclePose` | `removeEgo` | `removeObstacle` | `show` | `updateEgoGeometry` | `updateEgoPose` | `updateObstacleGeometry` | `updateObstaclePose`

Topics
“Highway Trajectory Planning Using Frenet Reference Path”

Introduced in R2020b

egoGeometry

Geometric properties of ego bodies

Syntax

```
[egoIDs,geomStruct] = egoGeometry(capsuleListObj)
[egoIDs,geomStruct] = egoGeometry(capsuleListObj,selectEgoIDs)
[egoIDs,geomStruct,status] = egoGeometry(capsuleListObj,selectEgoIDs)
```

Description

`[egoIDs,geomStruct] = egoGeometry(capsuleListObj)` returns the ego ID and the geometry parameters for each ego body in the capsule list.

`[egoIDs,geomStruct] = egoGeometry(capsuleListObj,selectEgoIDs)` specifies which ego bodies to return the ID and geometry parameters for.

`[egoIDs,geomStruct,status] = egoGeometry(capsuleListObj,selectEgoIDs)` returns an indicator of whether each ID in `selectEgoIDs` exists.

Examples

Create and Modify Capsule-Based Ego Bodies

Add ego bodies to an environment using the `dynamicCapsuleList` object. Modify the properties of the ego bodies. Remove an ego body from the environment. Visualize the states of all objects in the environment at different timestamps.

Create the `dynamicCapsuleList` object. Extract the maximum number of steps to use as the number of time stamps for the object paths.

```
capsuleList = dynamicCapsuleList;
numSteps = capsuleList.MaxNumSteps;
```

Add Ego Bodies

Specify the states for the two ego bodies as a linear path from x = 0 m to x = 100 m. The two ego bodies are separated by 5 m in opposite directions on the y-axis.

```
egoState = linspace(0,1,numSteps)'.*[100 0 0];
egoState1 = egoState+[0 5 0];
egoState2 = egoState+[0 -5 0];
```

Generate default poses and geometric structures for the two ego bodies using ego IDs.

```
[egoIDs,egoPoseStruct] = egoPose(capsuleList,[1 2]);
[egoIDs,egoGeomStruct] = egoGeometry(capsuleList,egoIDs);
```

Update Ego Bodies

Assign the states to the ego bodies.

```
egoPoseStruct(1).States = egoState1;
egoPoseStruct(2).States = egoState2;
```

Increase the radius of the first ego body to 2 m.

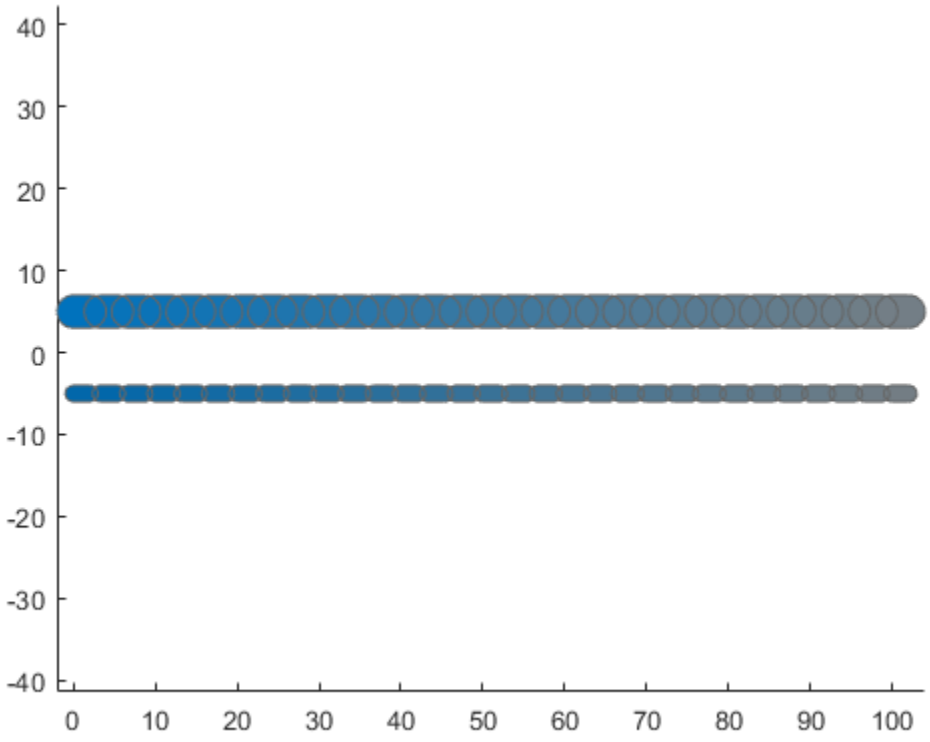
```
egoGeomStruct(1).Geometry.Radius = 2;
```

update the ego bodies using the `updateEgoPose` and `updateEgoGeometry` object functions.

```
updateEgoPose(capsuleList,egoIDs,egoPoseStruct);
updateEgoGeometry(capsuleList,egoIDs,egoGeomStruct);
```

Visualize the ego bodies.

```
show(capsuleList,'TimeStep',1:numSteps);
axis equal
```



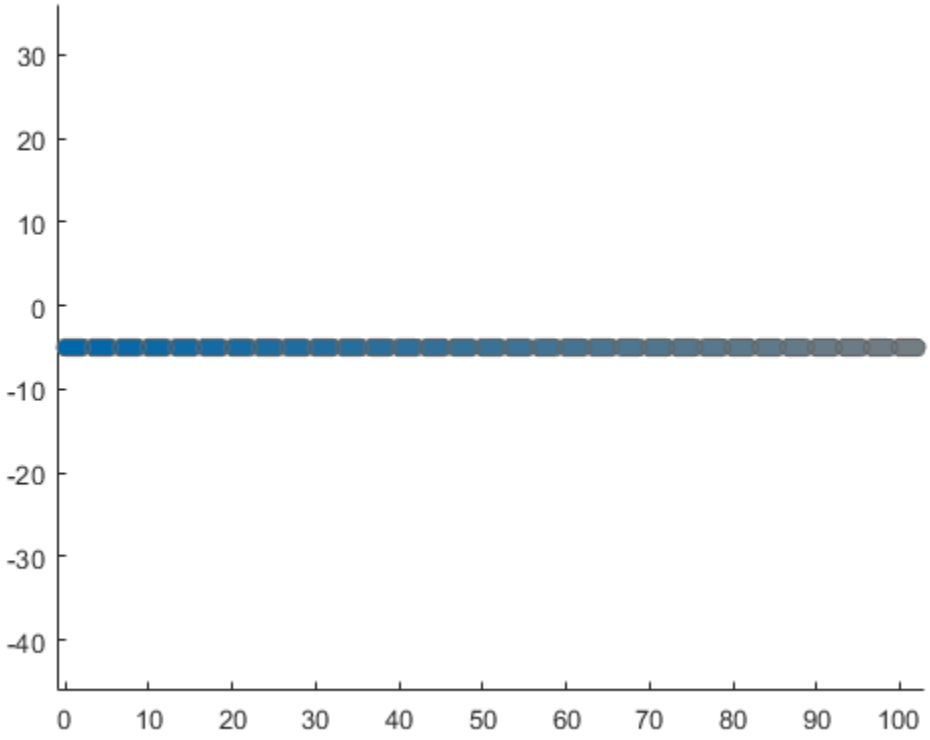
Remove Ego Body

Remove the first ego body from the capsule list by specifying its ID.

```
removeEgo(capsuleList,1);
```

Visualize the ego bodies again.

```
show(capsuleList,'TimeStep',1:numSteps);
axis equal
```



Input Arguments

`capsuleListObj` — Dynamic capsule list
`dynamicCapsuleList` object | `dynamicCapsuleList3D` object

Dynamic capsule list, specified as a `dynamicCapsuleList` or `dynamicCapsuleList3D` object.

`selectEgoIDs` — Ego body IDs
vector of positive integers

Ego body IDs, specified as a vector of positive integers. The function returns the ego IDs and geometry parameters for only the ego bodies specified in this vector.

Output Arguments

`egoIDs` — IDs of ego bodies
vector of positive integers

IDs of ego bodies, returned as a vector of positive integers.

`geomStruct` — Geometry parameters for ego bodies
structure | structure array

Geometry parameters for ego bodies, returned as a structure or structure array where each structure contains the fields from the structure in the `Geometry` field of the associated ego body. The fields of

this structure depend on whether you are using a `dynamicCapsuleList` or `dynamicCapsuleList3D` object.

`status` — Indication of ego body existence
N-element column vector

Indication of ego body existence, returned as a *N*-element column vector of ones, zeros, and negative ones. Each value indicates whether the associated body exists (`1`), updated (`0`), or a duplicate (`-1`). If you specify the same ego body ID more than once in the `selectEgoIDs` argument, then the function marks all instances of that ID after the first as duplicates and ignores them.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

Objects
`dynamicCapsuleList` | `dynamicCapsuleList3D`

Functions
`addEgo` | `addObstacle` | `checkCollision` | `egoPose` | `obstacleGeometry` | `obstaclePose` | `removeEgo` | `removeObstacle` | `show` | `updateEgoGeometry` | `updateEgoPose` | `updateObstacleGeometry` | `updateObstaclePose`

Topics
“Highway Trajectory Planning Using Frenet Reference Path”

Introduced in R2020b

egoPose

Poses of ego bodies

Syntax

```
[egoIDs,poseStruct] = egoPose(capsuleListObj)
[egoIDs,poseStruct] = egoPose(capsuleListObj,selectEgoIDs)
[egoIDs,poseStruct,status] = egoPose(capsuleListObj,selectEgoIDs)
```

Description

`[egoIDs,poseStruct] = egoPose(capsuleListObj)` returns the ego ID and the states for each ego body in the specified capsule list.

`[egoIDs,poseStruct] = egoPose(capsuleListObj,selectEgoIDs)` specifies which ego bodies to return the ID and states for.

`[egoIDs,poseStruct,status] = egoPose(capsuleListObj,selectEgoIDs)` returns an indicator of whether each ID in `selectEgoIDs` exists.

Examples

Create and Modify Capsule-Based Ego Bodies

Add ego bodies to an environment using the `dynamicCapsuleList` object. Modify the properties of the ego bodies. Remove an ego body from the environment. Visualize the states of all objects in the environment at different timestamps.

Create the `dynamicCapsuleList` object. Extract the maximum number of steps to use as the number of time stamps for the object paths.

```
capsuleList = dynamicCapsuleList;
numSteps = capsuleList.MaxNumSteps;
```

Add Ego Bodies

Specify the states for the two ego bodies as a linear path from x = 0 m to x = 100 m. The two ego bodies are separated by 5 m in opposite directions on the y-axis.

```
egoState = linspace(0,1,numSteps)'.*[100 0 0];
egoState1 = egoState+[0 5 0];
egoState2 = egoState+[0 -5 0];
```

Generate default poses and geometric structures for the two ego bodies using ego IDs.

```
[egoIDs,egoPoseStruct] = egoPose(capsuleList,[1 2]);
[egoIDs,egoGeomStruct] = egoGeometry(capsuleList,egoIDs);
```

Update Ego Bodies

Assign the states to the ego bodies.

```
egoPoseStruct(1).States = egoState1;
egoPoseStruct(2).States = egoState2;
```

Increase the radius of the first ego body to 2 m.

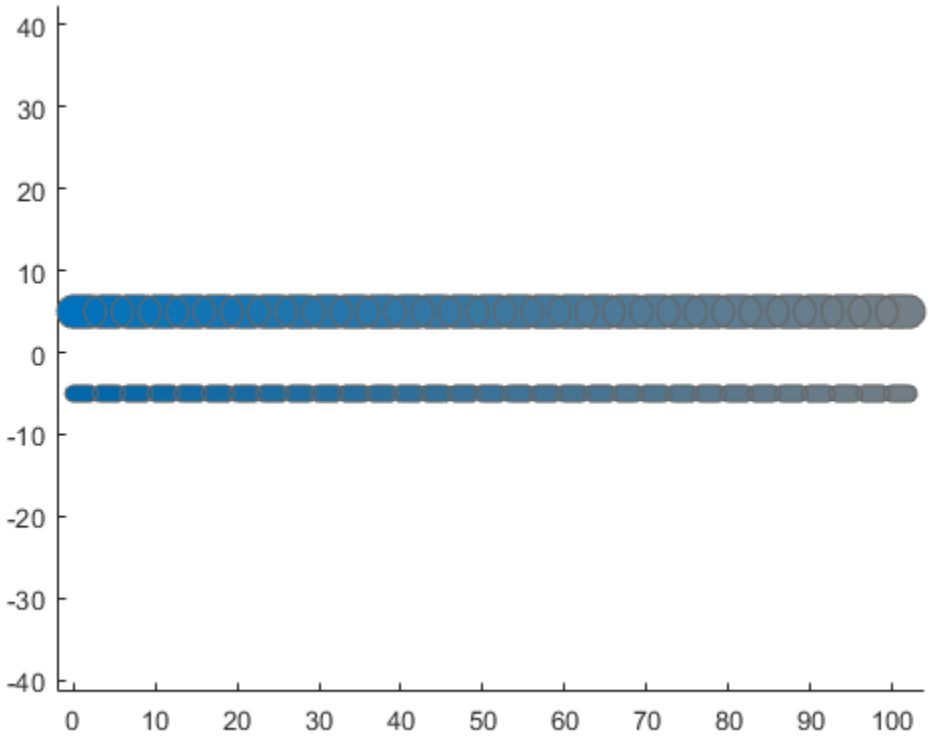
```
egoGeomStruct(1).Geometry.Radius = 2;
```

update the ego bodies using the `updateEgoPose` and `updateEgoGeometry` object functions.

```
updateEgoPose(capsuleList,egoIDs,egoPoseStruct);
updateEgoGeometry(capsuleList,egoIDs,egoGeomStruct);
```

Visualize the ego bodies.

```
show(capsuleList,'TimeStep',1:numSteps);
axis equal
```



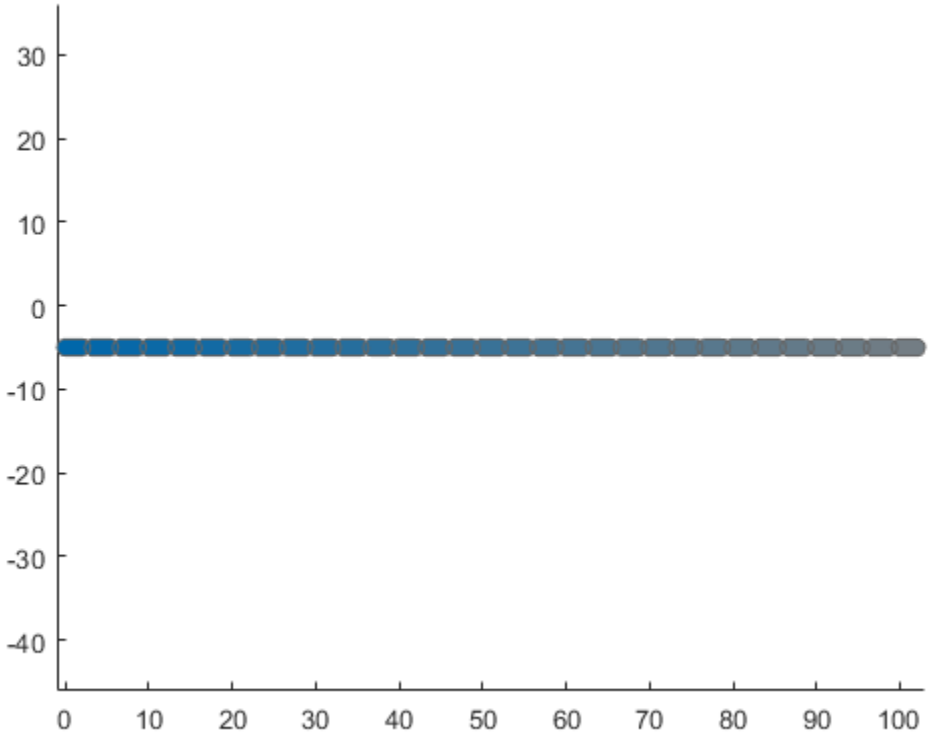
Remove Ego Body

Remove the first ego body from the capsule list by specifying its ID.

```
removeEgo(capsuleList,1);
```

Visualize the ego bodies again.

```
show(capsuleList,'TimeStep',1:numSteps);
axis equal
```



Input Arguments

`capsuleListObj` — Dynamic capsule list
`dynamicCapsuleList` object | `dynamicCapsuleList3D` object

Dynamic capsule list, specified as a `dynamicCapsuleList` or `dynamicCapsuleList3D` object.

`selectEgoIDs` — Ego body IDs
vector of positive integers

Ego body IDs, specified as a vector of positive integers. The function returns the ego IDs and states for only the ego bodies specified in this vector.

Output Arguments

`egoIDs` — IDs of ego bodies
vector of positive integers

IDs of ego bodies, returned as a vector of positive integers.

`poseStruct` — States for ego bodies
structure | structure array

States for ego bodies, returned as a structure or structure array. Each structure contains a matrix of states for each ego body. The state matrix size depends on whether you are using a `dynamicCapsuleList` or `dynamicCapsuleList3D` object.

Data Types: `struct`

`status` — Indication of ego body existence
N-element column vector

Indication of ego body existence, returned as a *N*-element column vector of ones, zeros, and negative ones. Each value indicates whether the associated body exists (`1`), updated (`0`), or a duplicate (`-1`). If you specify the same ego body ID more than once in the `selectEgoIDs` argument, then the function marks all instances of that ID after the first as duplicates and ignores them.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

Objects
`dynamicCapsuleList` | `dynamicCapsuleList3D`

Functions
`addEgo` | `addObstacle` | `checkCollision` | `egoGeometry` | `obstaclePose` | `removeEgo` | `removeObstacle` | `show` | `updateEgoGeometry` | `updateEgoPose` | `updateObstacleGeometry` | `updateObstaclePose`

Topics
“Highway Trajectory Planning Using Frenet Reference Path”

Introduced in R2020b

obstacleGeometry

Geometric properties of obstacles

Syntax

```
[obstacleIDs,geomStruct] = obstacleGeometry(capsuleListObj)
[obstacleIDs,geomStruct] = obstacleGeometry(capsuleListObj,selectObstacleIDs)
[obstacleIDs,geomStruct,status] = obstacleGeometry(capsuleListObj,
selectObstacleIDs)
```

Description

`[obstacleIDs,geomStruct] = obstacleGeometry(capsuleListObj)` returns the obstacle ID and the geometry parameters for each obstacle in the capsule list.

`[obstacleIDs,geomStruct] = obstacleGeometry(capsuleListObj,selectObstacleIDs)` specifies which obstacle to return the ID and geometry parameters for.

`[obstacleIDs,geomStruct,status] = obstacleGeometry(capsuleListObj, selectObstacleIDs)` returns an indicator of whether each ID in `selectobstacleIDs` exists.

Examples

Create and Modify Capsule-Based Obstacles

Add obstacles to an environment using the `dynamicCapsuleList` object. Modify the properties of the obstacles. Remove an obstacle from the environment. Visualize the states of all objects in the environment at different timestamps.

Create the `dynamicCapsuleList` object. Extract the maximum number of steps to use as the number of time stamps for the object paths.

```
capsuleList = dynamicCapsuleList;
numSteps = capsuleList.MaxNumSteps;
```

Add Obstacles

Specify the states for the two obstacles as a linear path from x = 0 m to x = 100 m. The two obstacles are separated by 10 m in opposite directions on the y-axis.

```
obsState = linspace(0,1,numSteps)'.*[100 0 0];
obsState1 = obsState+[0 10 0];
obsState2 = obsState+[0 -10 0];
```

Generate default poses and geometric structures for the two obstacles using obstacle IDs.

```
[obsIDs,obsPoseStruct] = obstaclePose(capsuleList,[1 2]);
[obsIDs,obsGeomStruct] = obstacleGeometry(capsuleList,obsIDs);
```

Update Obstacles

Assign the states to the obstacles.

```
obsPoseStruct(1).States = obsState1;
obsPoseStruct(2).States = obsState2;
```

Increase the radius of the first obstacle to 2 m.

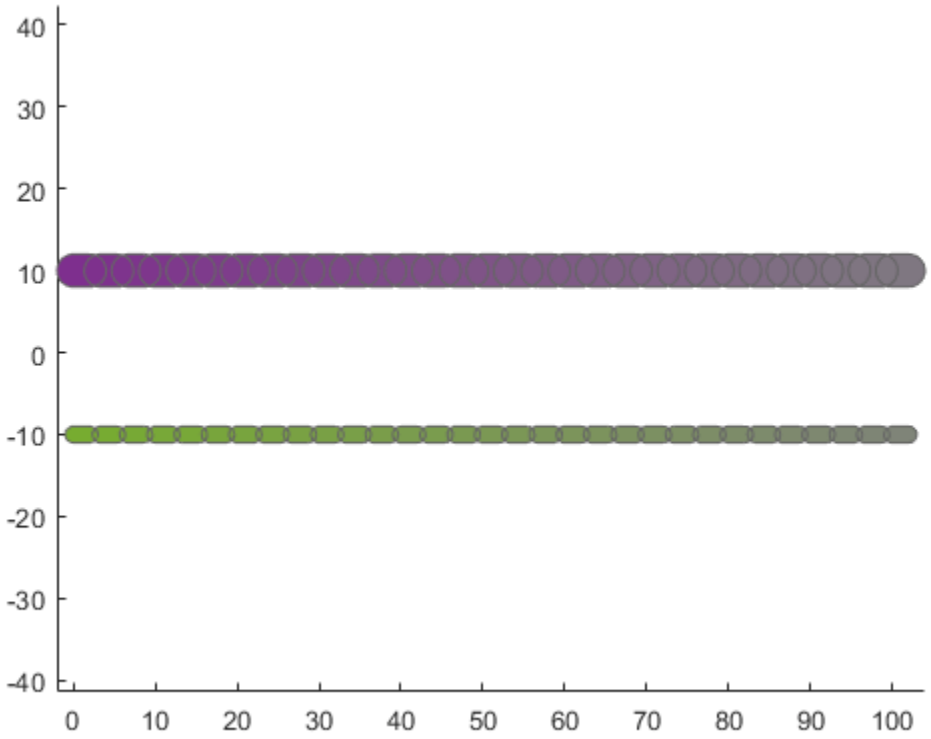
```
obsGeomStruct(1).Geometry.Radius = 2;
```

update the obstacles using the `updateObstaclePose` and `updateObstacleGeometry` object functions.

```
updateObstaclePose(capsuleList,obsIDs,obsPoseStruct);
updateObstacleGeometry(capsuleList,obsIDs,obsGeomStruct);
```

Visualize the obstacles.

```
show(capsuleList,'TimeStep',1:numSteps);
axis equal
```



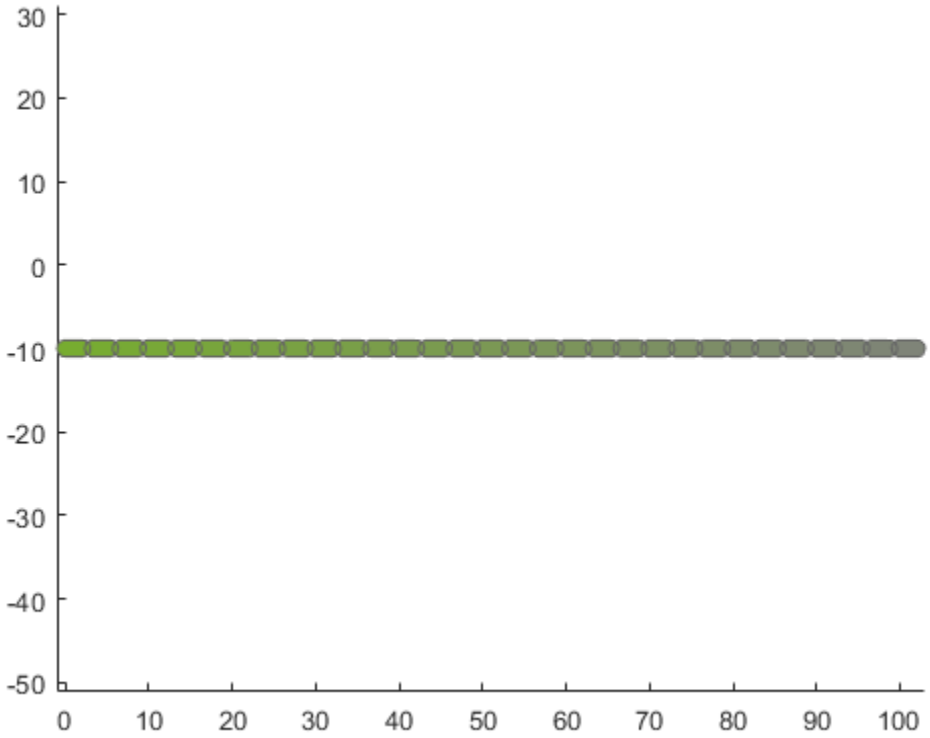
Remove Obstacles

Remove the first obstacle from the capsule list by specifying its ID.

```
removeObstacle(capsuleList,1);
```

Visualize the obstacles again.

```
show(capsuleList,'TimeStep',1:numSteps);
axis equal
```



Input Arguments

`capsuleListObj` — Dynamic capsule list
`dynamicCapsuleList` object | `dynamicCapsuleList3D` object

Dynamic capsule list, specified as a `dynamicCapsuleList` or `dynamicCapsuleList3D` object.

`selectObstacleIDs` — Obstacle IDs
vector of positive integers

Obstacle IDs, specified as a vector of positive integers. The function returns the obstacle IDs and geometry parameters for only the obstacles specified in this vector.

Output Arguments

`obstacleIDs` — IDs of obstacles
vector of positive integers

IDs of obstacles, returned as a vector of positive integers.

`geomStruct` — Geometry parameters for obstacles
structure | structure array

Geometry parameters for obstacles, returned as a structure or structure array where each structure contains the fields from the structure in the `Geometry` field of the associated obstacle. The fields of this structure depend on whether you are using a `dynamicCapsuleList` or `dynamicCapsuleList3D` object.

`status` — Indication of obstacle existence
vector of ones, zeros, and negative ones

Indication of obstacle existence, returned as a *N*-element column vector of ones, zeros, and negative ones. Each value indicates whether the associated obstacle exists (`1`), updated (`0`), or a duplicate (`-1`). If you specify the same ego body ID more than once in the `selectObstacleIDs` argument, then the function marks all instances of that ID after the first as duplicates and ignores them.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

Objects
`dynamicCapsuleList` | `dynamicCapsuleList3D`

Functions
`addEgo` | `addObstacle` | `checkCollision` | `egoGeometry` | `egoPose` | `obstaclePose` | `removeEgo` | `removeObstacle` | `show` | `updateEgoGeometry` | `updateEgoPose` | `updateObstacleGeometry` | `updateObstaclePose`

Topics
“Highway Trajectory Planning Using Frenet Reference Path”

Introduced in R2020b

obstaclePose

Poses of obstacles

Syntax

```
[obstacleIDs,poseStruct] = obstaclePose(capsuleListObj)
[obstacleIDs,poseStruct] = obstaclePose(capsuleListObj,selectObstacleIDs)
[obstacleIDs,poseStruct,status] = obstaclePose(capsuleListObj,
selectObstacleIDs)
```

Description

`[obstacleIDs,poseStruct] = obstaclePose(capsuleListObj)` returns the obstacle ID and states for each obstacle in the specified capsule list.

`[obstacleIDs,poseStruct] = obstaclePose(capsuleListObj,selectObstacleIDs)` specifies which obstacles to return the ID and states for.

`[obstacleIDs,poseStruct,status] = obstaclePose(capsuleListObj, selectObstacleIDs)` returns an indicator of whether each ID in `selectObstacleIDs` exists.

Examples

Create and Modify Capsule-Based Obstacles

Add obstacles to an environment using the `dynamicCapsuleList` object. Modify the properties of the obstacles. Remove an obstacle from the environment. Visualize the states of all objects in the environment at different timestamps.

Create the `dynamicCapsuleList` object. Extract the maximum number of steps to use as the number of time stamps for the object paths.

```
capsuleList = dynamicCapsuleList;
numSteps = capsuleList.MaxNumSteps;
```

Add Obstacles

Specify the states for the two obstacles as a linear path from x = 0 m to x = 100 m. The two obstacles are separated by 10 m in opposite directions on the y-axis.

```
obsState = linspace(0,1,numSteps)'.*[100 0 0];
obsState1 = obsState+[0 10 0];
obsState2 = obsState+[0 -10 0];
```

Generate default poses and geometric structures for the two obstacles using obstacle IDs.

```
[obsIDs,obsPoseStruct] = obstaclePose(capsuleList,[1 2]);
[obsIDs,obsGeomStruct] = obstacleGeometry(capsuleList,obsIDs);
```

Update Obstacles

Assign the states to the obstacles.

```
obsPoseStruct(1).States = obsState1;
obsPoseStruct(2).States = obsState2;
```

Increase the radius of the first obstacle to 2 m.

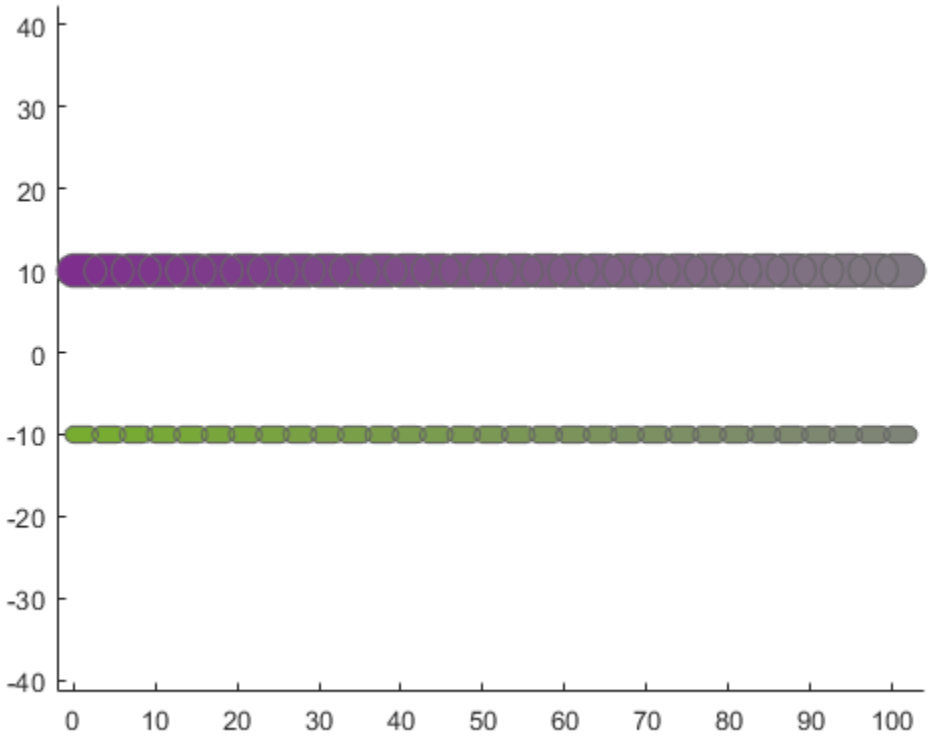
```
obsGeomStruct(1).Geometry.Radius = 2;
```

update the obstacles using the `updateObstaclePose` and `updateObstacleGeometry` object functions.

```
updateObstaclePose(capsuleList,obsIDs,obsPoseStruct);
updateObstacleGeometry(capsuleList,obsIDs,obsGeomStruct);
```

Visualize the obstacles.

```
show(capsuleList,'TimeStep',1:numSteps);
axis equal
```



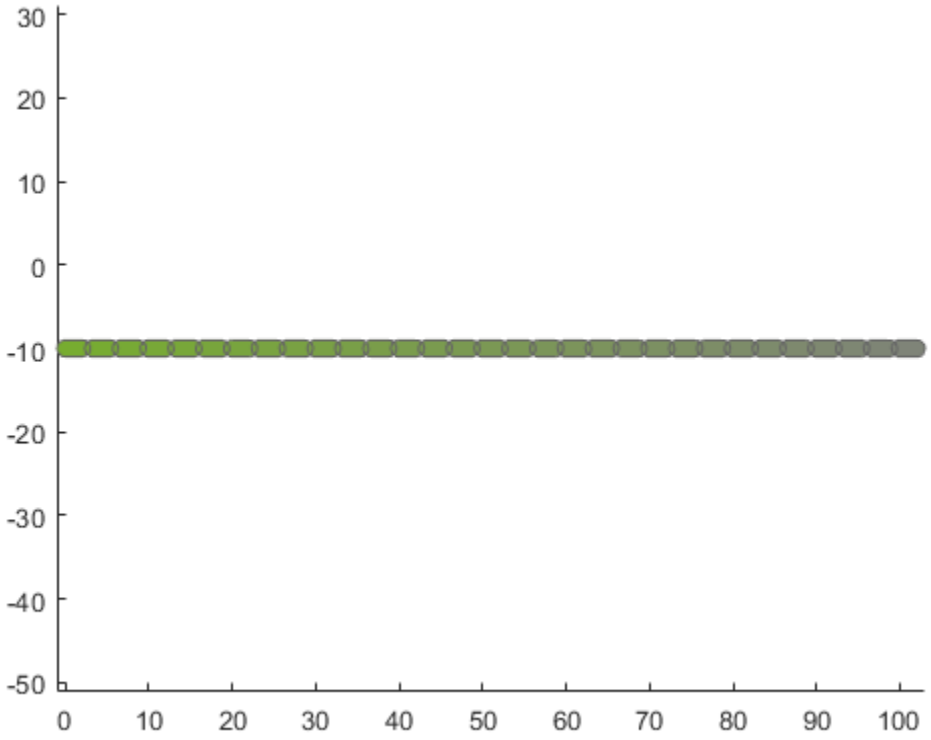
Remove Obstacles

Remove the first obstacle from the capsule list by specifying its ID.

```
removeObstacle(capsuleList,1);
```

Visualize the obstacles again.

```
show(capsuleList,'TimeStep',1:numSteps);
axis equal
```



Input Arguments

`capsuleListObj` — Dynamic capsule list

`dynamicCapsuleList` object | `dynamicCapsuleList3D` object

Dynamic capsule list, specified as a `dynamicCapsuleList` or `dynamicCapsuleList3D` object.

`selectObstacleIDs` — Obstacle IDs

vector of positive integers

Obstacle IDs, specified as a vector of positive integers. The function returns the obstacle IDs and states for only the obstacles specified in this vector.

Output Arguments

`obstacleIDs` — IDs of obstacles

vector of positive integers

IDs of obstacles, specified as a vector of positive integers.

`poseStruct` — States for ego bodies

structure | structure array

States for obstacles, returned as a structure or structure array. Each structure contains a matrix of states for each obstacle. The state matrix size depends on whether you are using a `dynamicCapsuleList` or `dynamicCapsuleList3D` object.

Data Types: `struct`

`status` — Indication of obstacle existence
N-element column vector

Indication of obstacle existence, returned as a *N*-element column vector of ones, zeros, and negative ones. Each value indicates whether the associated obstacle exists (`1`), updated (`0`), or a duplicate (`-1`). If you specify the same obstacle ID more than once in the `selectObstacleIDs` argument, then the function marks all instances of that ID after the first as duplicates and ignores them.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

Objects
`dynamicCapsuleList` | `dynamicCapsuleList3D`

Functions
`addEgo` | `addObstacle` | `checkCollision` | `egoGeometry` | `egoPose` | `obstacleGeometry` | `removeEgo` | `removeObstacle` | `show` | `updateEgoGeometry` | `updateEgoPose` | `updateObstacleGeometry` | `updateObstaclePose`

Topics
“Highway Trajectory Planning Using Frenet Reference Path”

Introduced in R2020b

removeEgo

Remove ego bodies from capsule list

Syntax

```
removeEgo(capsuleListObj,egoIDs)
status = removeEgo(capsuleListObj,egoIDs)
```

Description

`removeEgo(capsuleListObj,egoIDs)` removes ego bodies with the specified IDs from the dynamic capsule list.

`status = removeEgo(capsuleListObj,egoIDs)` additionally returns an indicator of whether an ego body is removed, not found, or a duplicate.

Examples

Create and Modify Capsule-Based Ego Bodies

Add ego bodies to an environment using the `dynamicCapsuleList` object. Modify the properties of the ego bodies. Remove an ego body from the environment. Visualize the states of all objects in the environment at different timestamps.

Create the `dynamicCapsuleList` object. Extract the maximum number of steps to use as the number of time stamps for the object paths.

```
capsuleList = dynamicCapsuleList;
numSteps = capsuleList.MaxNumSteps;
```

Add Ego Bodies

Specify the states for the two ego bodies as a linear path from x = 0 m to x = 100 m. The two ego bodies are separated by 5 m in opposite directions on the y-axis.

```
egoState = linspace(0,1,numSteps)'.*[100 0 0];
egoState1 = egoState+[0 5 0];
egoState2 = egoState+[0 -5 0];
```

Generate default poses and geometric structures for the two ego bodies using ego IDs.

```
[egoIDs,egoPoseStruct] = egoPose(capsuleList,[1 2]);
[egoIDs,egoGeomStruct] = egoGeometry(capsuleList,egoIDs);
```

Update Ego Bodies

Assign the states to the ego bodies.

```
egoPoseStruct(1).States = egoState1;
egoPoseStruct(2).States = egoState2;
```

Increase the radius of the first ego body to 2 m.

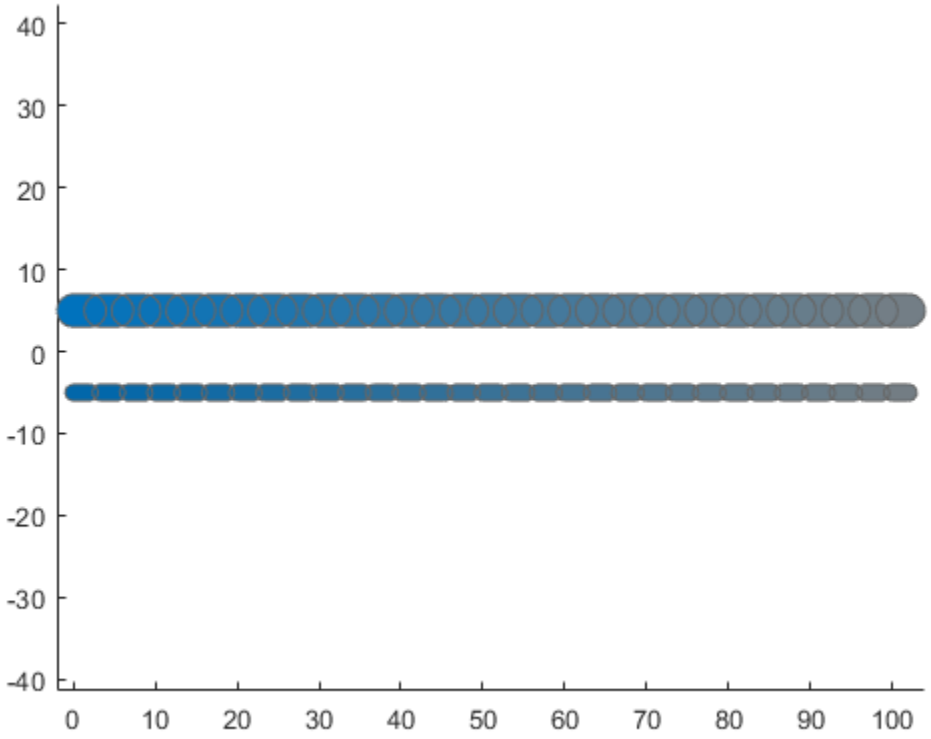
```
egoGeomStruct(1).Geometry.Radius = 2;
```

update the ego bodies using the `updateEgoPose` and `updateEgoGeometry` object functions.

```
updateEgoPose(capsuleList,egoIDs,egoPoseStruct);
updateEgoGeometry(capsuleList,egoIDs,egoGeomStruct);
```

Visualize the ego bodies.

```
show(capsuleList,'TimeStep',1:numSteps);
axis equal
```



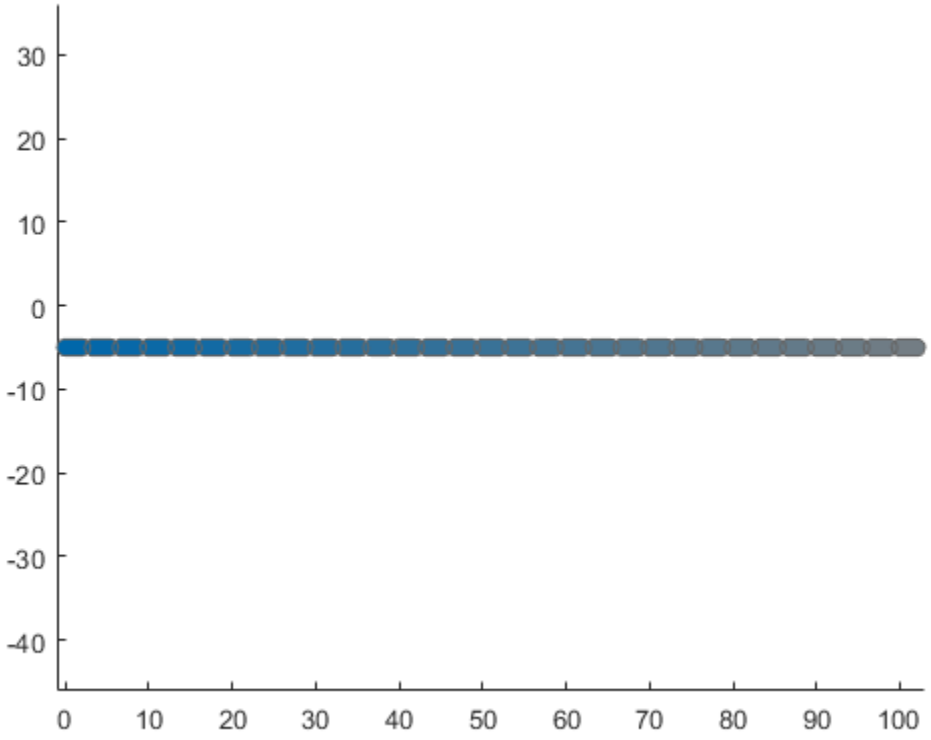
Remove Ego Body

Remove the first ego body from the capsule list by specifying its ID.

```
removeEgo(capsuleList,1);
```

Visualize the ego bodies again.

```
show(capsuleList,'TimeStep',1:numSteps);
axis equal
```



Input Arguments

`capsuleListObj` — Dynamic capsule list

`dynamicCapsuleList` object | `dynamicCapsuleList3D` object

Dynamic capsule list, specified as a `dynamicCapsuleList` or `dynamicCapsuleList3D` object.

`egoIDs` — IDs of ego bodies

vector of positive integers

IDs of ego bodies to remove, specified as a vector of positive integers.

Output Arguments

`status` — Result of removing ego bodies

N-element column vector

Result of removing ego bodies, specified as *N*-element column vector of ones, zeros, and negative ones. *N* is the number of ego bodies specified in the `egoIDs` argument. Each value indicates whether the body is removed (`1`), not found (`0`), or a duplicate (`-1`). If you specify the same ego ID multiple times in the `egoIDs` input argument, then all entries besides the last are marked as a duplicate.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

Objects

`dynamicCapsuleList` | `dynamicCapsuleList3D`

Functions

`addEgo` | `addObstacle` | `checkCollision` | `egoGeometry` | `egoPose` | `obstacleGeometry` | `obstaclePose` | `removeObstacle` | `show` | `updateEgoGeometry` | `updateEgoPose` | `updateObstacleGeometry` | `updateObstaclePose`

Topics

“Highway Trajectory Planning Using Frenet Reference Path”

Introduced in R2020b

removeObstacle

Remove obstacles from capsule list

Syntax

```
removeObstacle(capsuleListObj,obstacleIDs)
status = removeObstacle(capsuleListObj,obstacleIDs)
```

Description

`removeObstacle(capsuleListObj,obstacleIDs)` removes obstacles with the specified IDs from the dynamic capsule list.

`status = removeObstacle(capsuleListObj,obstacleIDs)` additionally returns an indicator of whether an obstacles is removed, not found, or a duplicate.

Examples

Create and Modify Capsule-Based Obstacles

Add obstacles to an environment using the `dynamicCapsuleList` object. Modify the properties of the obstacles. Remove an obstacle from the environment. Visualize the states of all objects in the environment at different timestamps.

Create the `dynamicCapsuleList` object. Extract the maximum number of steps to use as the number of time stamps for the object paths.

```
capsuleList = dynamicCapsuleList;
numSteps = capsuleList.MaxNumSteps;
```

Add Obstacles

Specify the states for the two obstacles as a linear path from x = 0 m to x = 100 m. The two obstacles are separated by 10 m in opposite directions on the y-axis.

```
obsState = linspace(0,1,numSteps)'.*[100 0 0];
obsState1 = obsState+[0 10 0];
obsState2 = obsState+[0 -10 0];
```

Generate default poses and geometric structures for the two obstacles using obstacle IDs.

```
[obsIDs,obsPoseStruct] = obstaclePose(capsuleList,[1 2]);
[obsIDs,obsGeomStruct] = obstacleGeometry(capsuleList,obsIDs);
```

Update Obstacles

Assign the states to the obstacles.

```
obsPoseStruct(1).States = obsState1;
obsPoseStruct(2).States = obsState2;
```

Increase the radius of the first obstacle to 2 m.

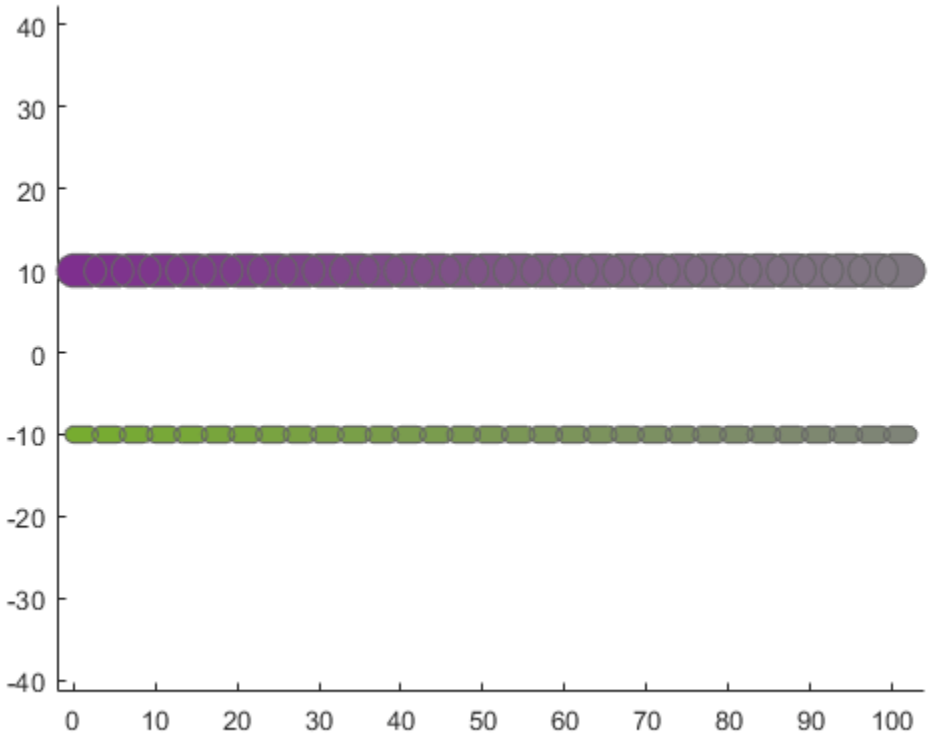
```
obsGeomStruct(1).Geometry.Radius = 2;
```

update the obstacles using the `updateObstaclePose` and `updateObstacleGeometry` object functions.

```
updateObstaclePose(capsuleList,obsIDs,obsPoseStruct);
updateObstacleGeometry(capsuleList,obsIDs,obsGeomStruct);
```

Visualize the obstacles.

```
show(capsuleList,'TimeStep',1:numSteps);
axis equal
```



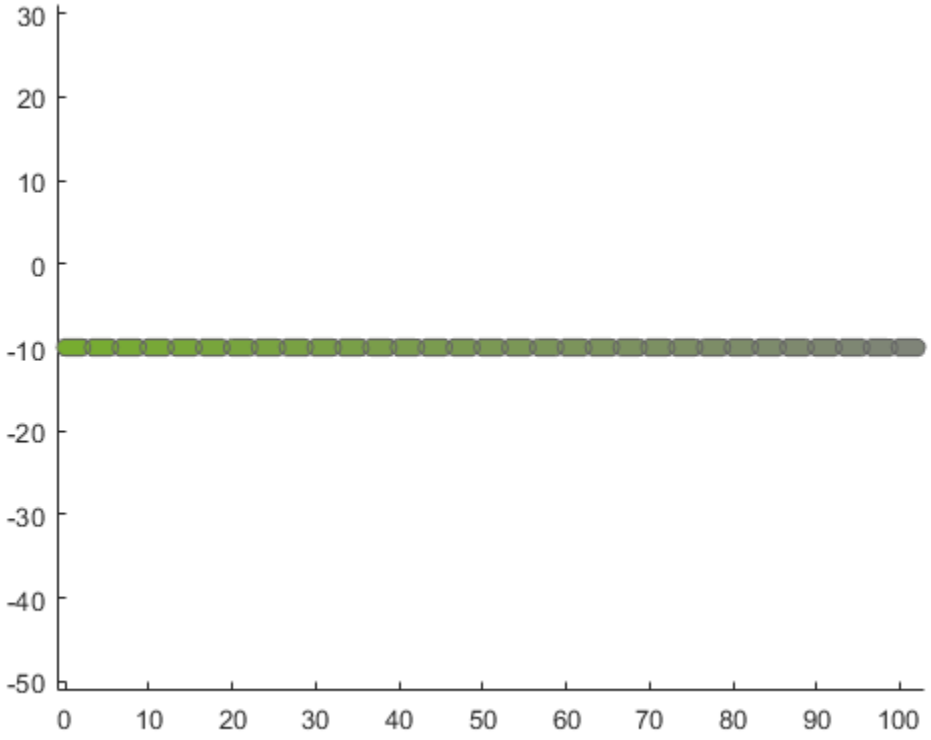
Remove Obstacles

Remove the first obstacle from the capsule list by specifying its ID.

```
removeObstacle(capsuleList,1);
```

Visualize the obstacles again.

```
show(capsuleList,'TimeStep',1:numSteps);
axis equal
```



Input Arguments

`capsuleListObj` — Dynamic capsule list
`dynamicCapsuleList` object | `dynamicCapsuleList3D` object

Dynamic capsule list, specified as a `dynamicCapsuleList` or `dynamicCapsuleList3D` object.

`obstacleIDs` — IDs of obstacles
vector of positive integers

IDs of obstacles to remove, specified as a vector of positive integers.

Output Arguments

`status` — Result of removing obstacles
N-element column vector

Result of removing obstacles, specified as *N*-element column vector of ones, zeros, and negative ones. *N* is the number of obstacles specified in the `obstacleIDs` argument. Each value indicates whether the obstacle is removed (`1`), not found (`0`), or a duplicate (`-1`). If you specify the same obstacle ID multiple times in the `obstacleIDs` input argument, then all entries besides the last are marked as a duplicate.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

Objects

`dynamicCapsuleList` | `dynamicCapsuleList3D`

Functions

`addEgo` | `addObstacle` | `checkCollision` | `egoGeometry` | `egoPose` | `obstacleGeometry` | `obstaclePose` | `removeEgo` | `show` | `updateEgoGeometry` | `updateEgoPose` | `updateObstacleGeometry` | `updateObstaclePose`

Topics

“Highway Trajectory Planning Using Frenet Reference Path”

Introduced in R2020b

show

Display ego bodies and obstacles in environment

Syntax

```
ax = show(capsuleListObj)
ax = show(capsuleListObj,Name,Value)
```

Description

`ax = show(capsuleListObj)` displays the initial state of all ego bodies and obstacles in the specified capsule list, and returns the axes handle of the plot.

`ax = show(capsuleListObj,Name,Value)` specifies options using name-value pair arguments on page 2-265. For example, `'FastUpdate',true` enables fast updates to an existing plot.

Examples

Build Ego Body Paths and Check for Collisions with Obstacles

Build an ego body path and maintain obstacle states using the `dynamicCapsuleList` object. Visualize the states of all objects in the environment at different timestamps. Validate the path of the ego body by checking for collisions with obstacles in the environment.

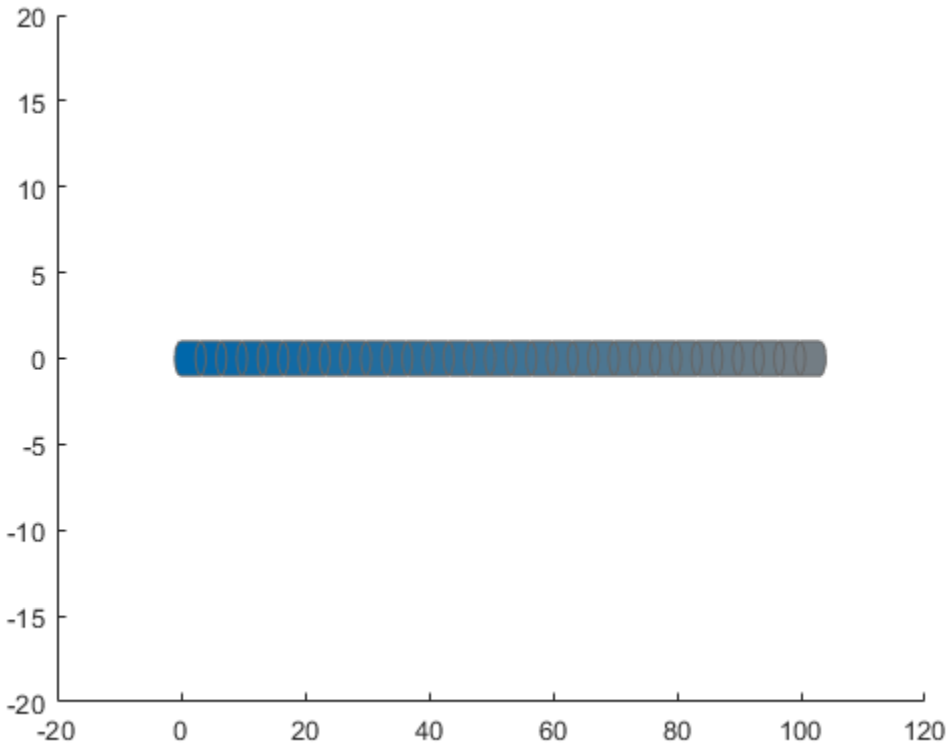
Create the `dynamicCapsuleList` object. Extract the maximum number of steps to use as the number of time stamps for your object paths.

```
obsList = dynamicCapsuleList;
numSteps = obsList.MaxNumSteps;
```

Add Ego Body

Define an ego body by specifying the ID, geometry, and state together in a structure. The capsule geometry has a length of 3 m and radius of 1 m. Specify the state as a linear path from x = 0m to x = 100m.

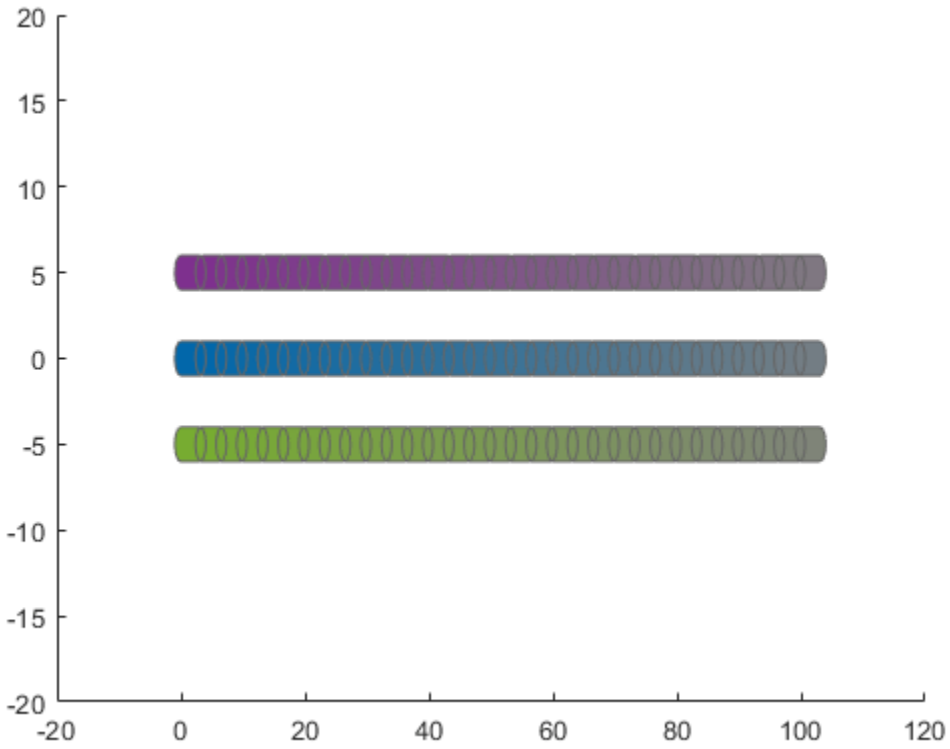
```
egoID1 = 1;
geom = struct("Length",3,"Radius",1,"FixedTransform",eye(3));
states = linspace(0,1,obsList.MaxNumSteps)'.*[100 0 0];

egoCapsule1 = struct('ID',egoID1,'States',states,'Geometry',geom);
addEgo(obsList,egoCapsule1);

show(obsList,"TimeStep",[1:numSteps]);
ylim([-20 20])
```



Add Obstacles

Specify states for two obstacles that are separated from the ego body by 5 m in opposite directions on the *y*-axis.. Assume the obstacles have the same geometry `geom` as the ego body.

```
obsState1 = states + [0 5 0];
obsState2 = states + [0 -5 0];

obsCapsule1 = struct('ID',1,'States',obsState1,'Geometry',geom);
obsCapsule2 = struct('ID',2,'States',obsState2,'Geometry',geom);

addObstacle(obsList,obsCapsule1);
addObstacle(obsList,obsCapsule2);

show(obsList,"TimeStep",[1:numSteps]);
ylim([-20 20])
```



Update Obstacles

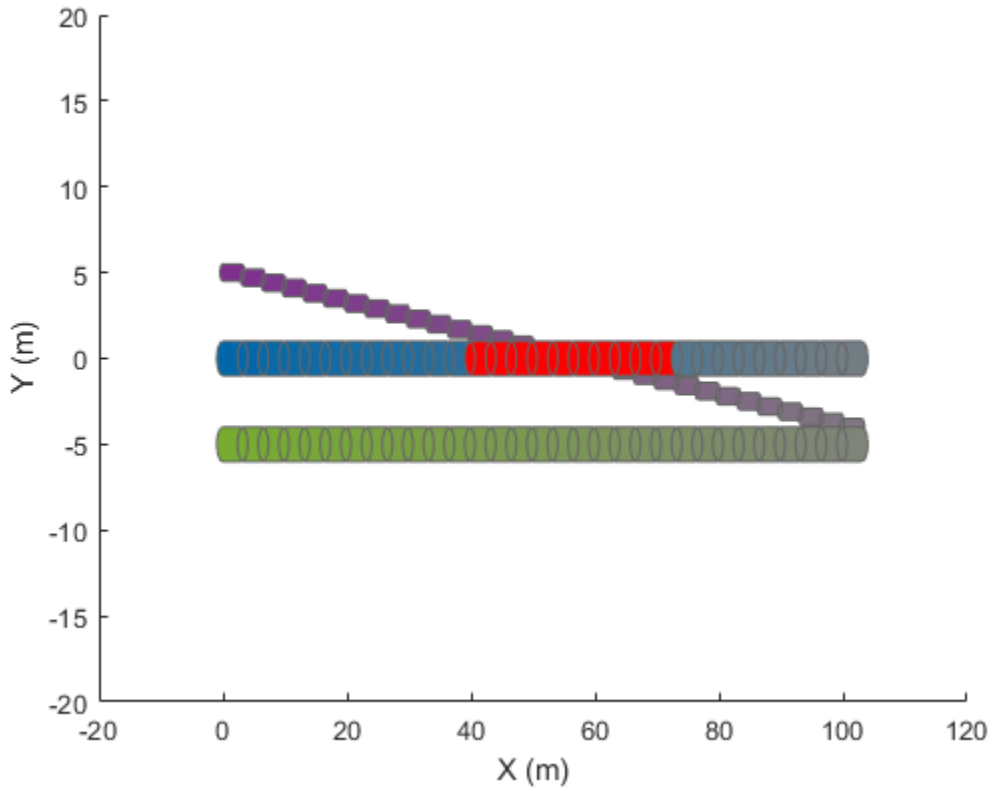
Alter your obstacle locations and geometry dimensions over time. Use the previously generated structure, modify the fields, and update the obstacles using the `updateObstacleGeometry` and `updateObstaclePose` object functions. Reduces the radius of the first obstacle to 0.5 m, and change the path to move it towards the ego body.

```
obsCapsule1.Geometry.Radius = 0.5;

obsCapsule1.States = ...
    [linspace(0,100,numSteps)' ... % x
     linspace(5,-4,numSteps)' ... % y
     zeros(numSteps,1)]; % theta

updateObstacleGeometry(obsList,1,obsCapsule1);
updateObstaclePose(obsList,1,obsCapsule1);
```

Check for Collisions

Visualize the new paths. Show where collisions between the ego body and an obstacle, which the display highlights in red. Notice that collisions between the obstacles are not checked.

```
show(obsList,"TimeStep",[1:numSteps],"ShowCollisions",1);
ylim([-20 20])
xlabel("X (m)")
ylabel("Y (m)")
```



Programmatically check for collisions by using the `checkCollision` object function. The function returns a vector of logical values that indicates the status of each time step. The vector is transposed for display purposes.

```
collisions = checkCollision(obsList)'
```

```
collisions = 1x31 logical array

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

To validate paths with a large number of steps, use the `any` function on the vector of collision values.

```
if any(collisions)
    disp("Collision detected.")
end
```

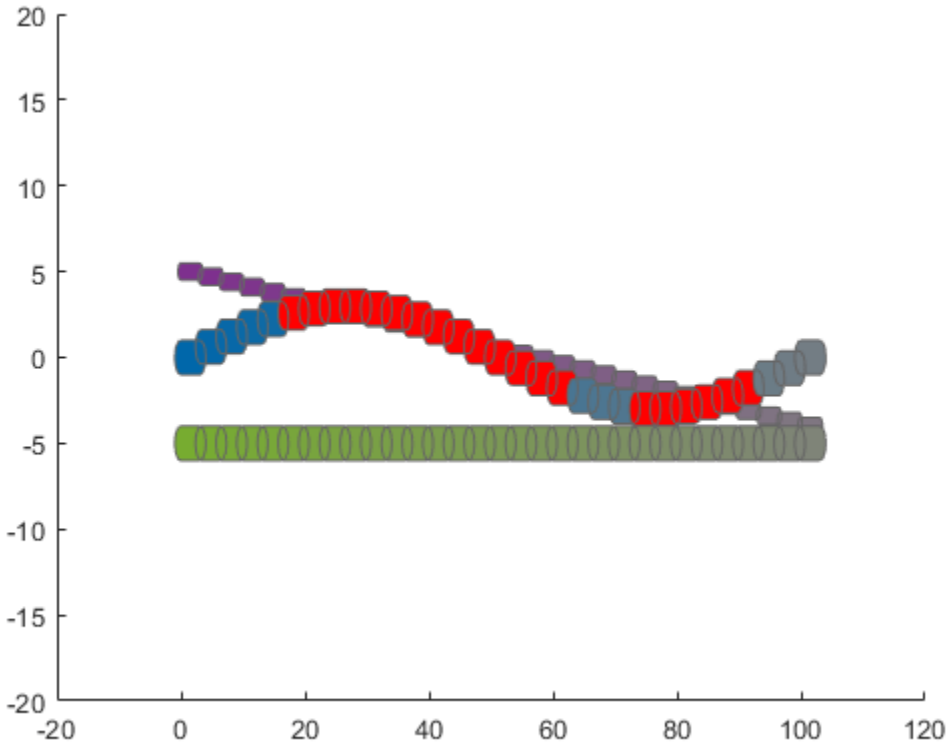
```
Collision detected.
```

Update Ego Path

Specify a new path for the ego body. Visualize the paths again, displaying collisions.

```
egoCapsule1.States = ...
    [linspace(0,100,numSteps)' ... % x
    3*sin(linspace(0,2*pi,numSteps))' ... % y
    zeros(numSteps,1)]; % theta
```

```
updateEgoPose(obsList,1,egoCapsule1);

show(obsList,"TimeStep",[1:numSteps],"ShowCollisions",1);
ylim([-20 20])
```



Input Arguments

`capsuleListObj` — Dynamic capsule list

`dynamicCapsuleList` object | `dynamicCapsuleList3D` object

Dynamic capsule list, specified as a `dynamicCapsuleList` or `dynamicCapsuleList3D` object.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'FastUpdate',true` enables the option for fast updates in an existing plot.

`Parent` — Parent axes to plot on

`gca` (default) | `Axes` handle

Parent axes to plot on, specified as the comma-separated pair consisting of `'Parent'` and an Axes handle.

`FastUpdate` — Perform fast update to existing plot
`false` or `0` (default) | `true` or `1`

Perform a fast update to an existing plot, specified as the comma-separated pair consisting of `'FastUpdate'` and a logical `0` (`false`) or `1` (`true`). You must use the `show` object function to initially display your capsule list before you can specify it with this argument.

Data Types: `logical`

`TimeStep` — Time steps to display
`1` (default) | numeric vector

Time steps to display, specified as the comma-separated pair consisting of `'TimeStep'` and numeric vector of values in the range [1, *N*], where *N* is the value of the `MaxNumSteps` property of the object specified in the `capsuleListObj` argument. Each time step corresponds to a row of the state matrix for each ego body and obstacle.

`ShowCollisions` — Check for and highlight collisions in display
`false` or `0` (default) | `true` or `1`

Check for and highlight collisions in the display, specified as the comma-separated pair consisting of `'ShowCollisions'` and a logical `0` (`false`) or `1` (`true`).

Data Types: `logical`

`EgoIDs` — Ego IDs to display
vector of positive integers

Ego IDs to display, specified as the comma-separated pair consisting of `'EgoIDs'` and a vector of positive integers. By default, the object function displays all ego bodies.

`ObstacleIDs` — Obstacle IDs to display
vector of positive integers

Obstacle IDs to display, specified as the comma-separated pair consisting of `'ObstacleIDs'` and a vector of positive integers. By default, the function displays all obstacles.

Output Arguments

ax — Parent axes of dynamic capsule list plot
`Axes` handle

Parent axes of the dynamic capsule list plot, returned as anAxes handle.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

Objects
`dynamicCapsuleList` | `dynamicCapsuleList3D`

Functions
addEgo | addObstacle | checkCollision | egoGeometry | egoPose | obstacleGeometry | obstaclePose | removeEgo | removeObstacle | updateEgoGeometry | updateEgoPose | updateObstacleGeometry | updateObstaclePose

Topics
“Highway Trajectory Planning Using Frenet Reference Path”

Introduced in R2020b

updateEgoGeometry

Update geometric properties of ego bodies

Syntax

```
updateEgoGeometry(capsuleListObj,egoIDs,geomStruct)
status = updateEgoGeometry(capsuleListObj,egoIDs,geomStruct)
```

Description

`updateEgoGeometry(capsuleListObj,egoIDs,geomStruct)` updates geometry parameters for the specified ego bodies in the capsule list. If a specified ego ID does not already exist, the function adds a new ego body with that ID to the list.

`status = updateEgoGeometry(capsuleListObj,egoIDs,geomStruct)` additionally returns an indicator of whether an ego body is added, updated, or a duplicate.

Examples

Create and Modify Capsule-Based Ego Bodies

Add ego bodies to an environment using the `dynamicCapsuleList` object. Modify the properties of the ego bodies. Remove an ego body from the environment. Visualize the states of all objects in the environment at different timestamps.

Create the `dynamicCapsuleList` object. Extract the maximum number of steps to use as the number of time stamps for the object paths.

```
capsuleList = dynamicCapsuleList;
numSteps = capsuleList.MaxNumSteps;
```

Add Ego Bodies

Specify the states for the two ego bodies as a linear path from x = 0 m to x = 100 m. The two ego bodies are separated by 5 m in opposite directions on the y-axis.

```
egoState = linspace(0,1,numSteps)'.*[100 0 0];
egoState1 = egoState+[0 5 0];
egoState2 = egoState+[0 -5 0];
```

Generate default poses and geometric structures for the two ego bodies using ego IDs.

```
[egoIDs,egoPoseStruct] = egoPose(capsuleList,[1 2]);
[egoIDs,egoGeomStruct] = egoGeometry(capsuleList,egoIDs);
```

Update Ego Bodies

Assign the states to the ego bodies.

```
egoPoseStruct(1).States = egoState1;
egoPoseStruct(2).States = egoState2;
```

Increase the radius of the first ego body to 2 m.

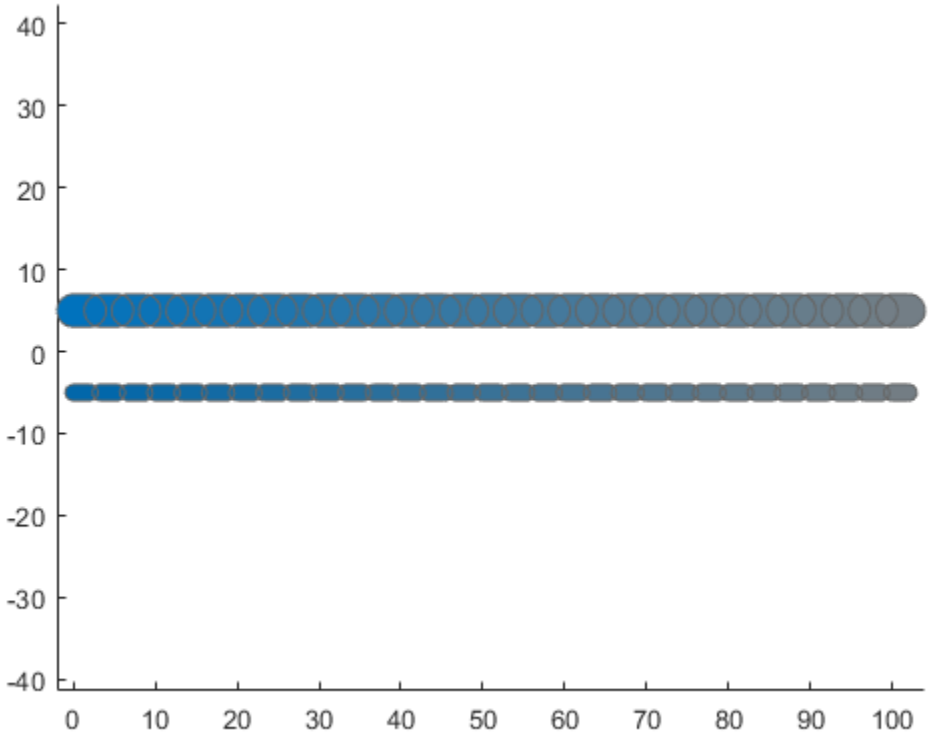
```
egoGeomStruct(1).Geometry.Radius = 2;
```

update the ego bodies using the `updateEgoPose` and `updateEgoGeometry` object functions.

```
updateEgoPose(capsuleList,egoIDs,egoPoseStruct);
updateEgoGeometry(capsuleList,egoIDs,egoGeomStruct);
```

Visualize the ego bodies.

```
show(capsuleList,'TimeStep',1:numSteps);
axis equal
```



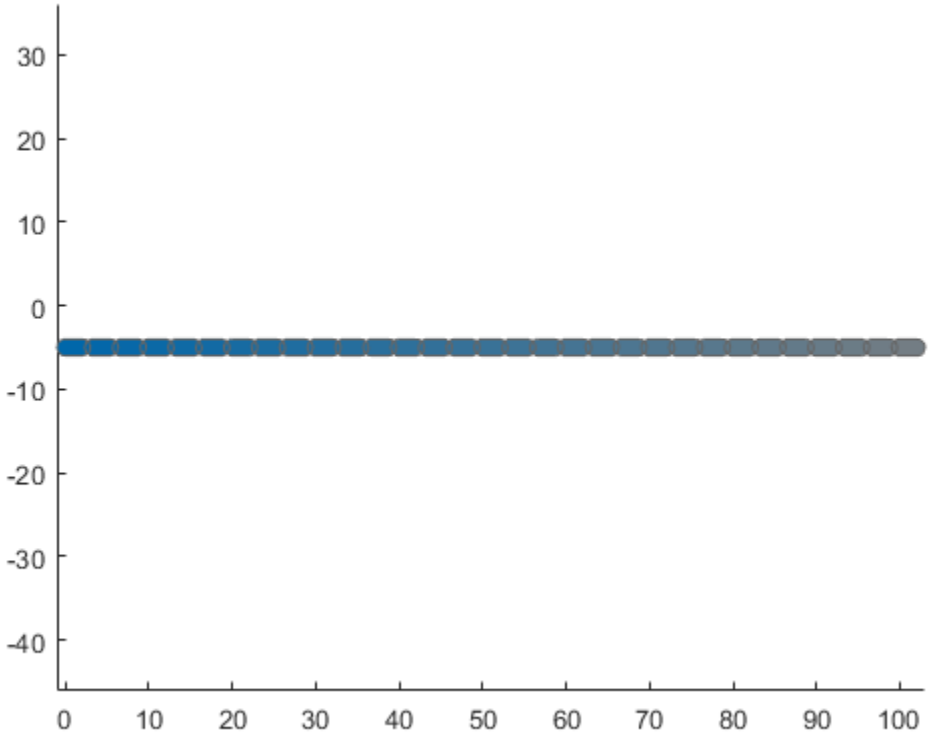
Remove Ego Body

Remove the first ego body from the capsule list by specifying its ID.

```
removeEgo(capsuleList,1);
```

Visualize the ego bodies again.

```
show(capsuleList,'TimeStep',1:numSteps);
axis equal
```



Input Arguments

`capsuleListObj` — Dynamic capsule list
`dynamicCapsuleList` object | `dynamicCapsuleList3D` object

Dynamic capsule list, specified as a `dynamicCapsuleList` or `dynamicCapsuleList3D` object.

`egoIDs` — IDs of ego bodies to update
vector of positive integers

IDs of ego bodies to update, specified as a vector of positive integers.

`geomStruct` — Geometry parameters for ego bodies
structure | structure array

Geometry parameters for ego bodies, specified as a structure or structure array, where each structure contains the fields of the structure in the `Geometry` field of the ego body to be updated. The fields of this structure depend on whether you are using a `dynamicCapsuleList` or `dynamicCapsuleList3D` object.

Data Types: `struct`

Output Arguments

`status` — Result of updating ego bodies
N-element column vector

Result of updating ego bodies, specified as *N*-element column vector of ones, zeros, and negative ones. *N* is the number of ego bodies specified in the `egoIDs` argument. Each value indicates whether the body is removed (`1`), not found (`0`), or a duplicate (`-1`). If you specify the same ego ID multiple times in the `egoIDs` input argument, then all entries besides the last are marked as a duplicate.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

Objects

`dynamicCapsuleList` | `dynamicCapsuleList3D`

Functions

`addEgo` | `addObstacle` | `checkCollision` | `egoGeometry` | `egoPose` | `obstacleGeometry` | `obstaclePose` | `removeEgo` | `removeObstacle` | `show` | `updateEgoPose` | `updateObstacleGeometry` | `updateObstaclePose`

Topics

“Highway Trajectory Planning Using Frenet Reference Path”

Introduced in R2020b

updateEgoPose

Update states of ego bodies

Syntax

```
updateEgoPose(capsuleListObj,egoIDs,poseStruct)
status = updateEgoPose(capsuleListObj,egoIDs,poseStruct)
```

Description

`updateEgoPose(capsuleListObj,egoIDs,poseStruct)` updates the states of the specified ego bodies in the capsule list. If a specified ego ID does not already exist, the function adds a new ego body with that ID to the list.

`status = updateEgoPose(capsuleListObj,egoIDs,poseStruct)` returns an indicator of whether an ego body is added, updated, or a duplicate.

Examples

Build Ego Body Paths and Check for Collisions with Obstacles

Build an ego body path and maintain obstacle states using the `dynamicCapsuleList` object. Visualize the states of all objects in the environment at different timestamps. Validate the path of the ego body by checking for collisions with obstacles in the environment.

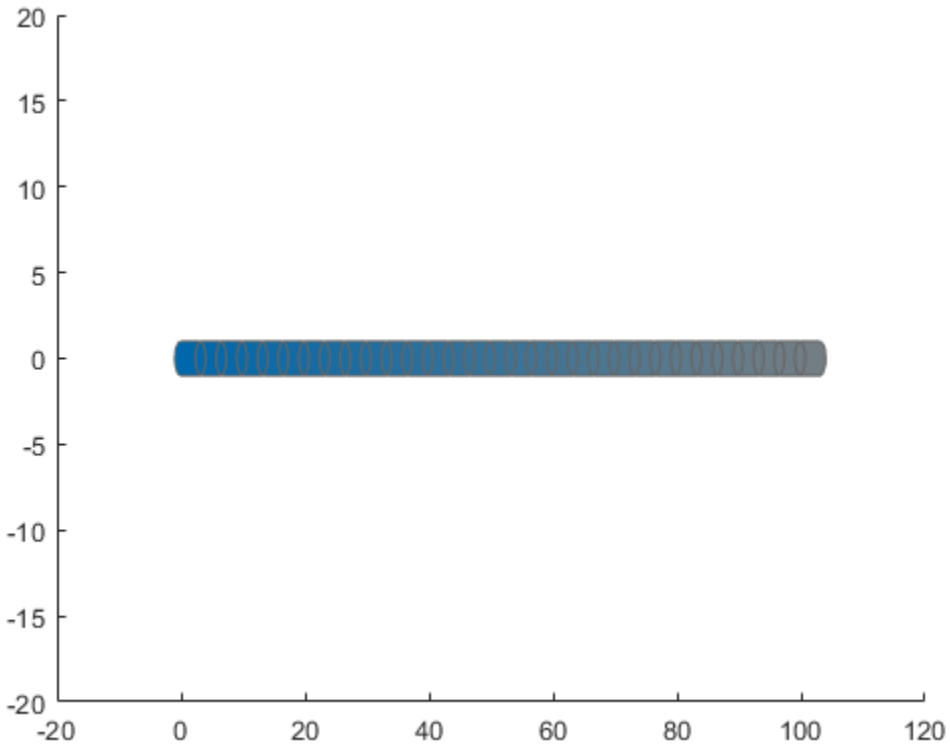
Create the `dynamicCapsuleList` object. Extract the maximum number of steps to use as the number of time stamps for your object paths.

```
obsList = dynamicCapsuleList;
numSteps = obsList.MaxNumSteps;
```

Add Ego Body

Define an ego body by specifying the ID, geometry, and state together in a structure. The capsule geometry has a length of 3 m and radius of 1 m. Specify the state as a linear path from x = 0m to x = 100m.

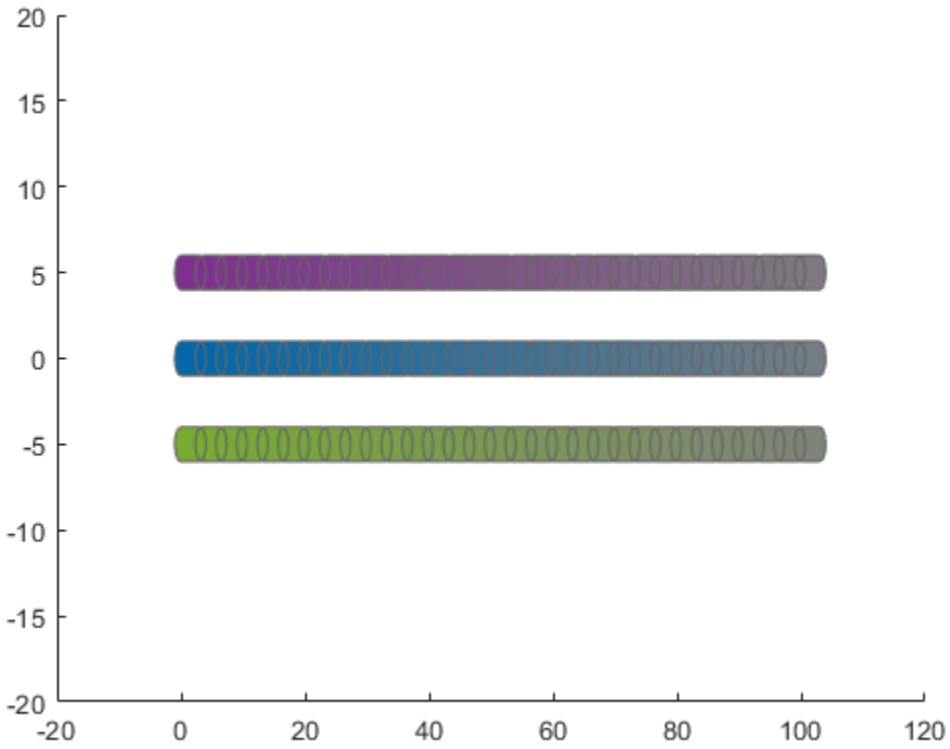
```
egoID1 = 1;
geom = struct("Length",3,"Radius",1,"FixedTransform",eye(3));
states = linspace(0,1,obsList.MaxNumSteps)'.*[100 0 0];

egoCapsule1 = struct('ID',egoID1,'States',states,'Geometry',geom);
addEgo(obsList,egoCapsule1);

show(obsList,"TimeStep",[1:numSteps]);
ylim([-20 20])
```



Add Obstacles

Specify states for two obstacles that are separated from the ego body by 5 m in opposite directions on the *y*-axis.. Assume the obstacles have the same geometry `geom` as the ego body.

```
obsState1 = states + [0 5 0];
obsState2 = states + [0 -5 0];

obsCapsule1 = struct('ID',1,'States',obsState1,'Geometry',geom);
obsCapsule2 = struct('ID',2,'States',obsState2,'Geometry',geom);

addObstacle(obsList,obsCapsule1);
addObstacle(obsList,obsCapsule2);

show(obsList,"TimeStep",[1:numSteps]);
ylim([-20 20])
```



Update Obstacles

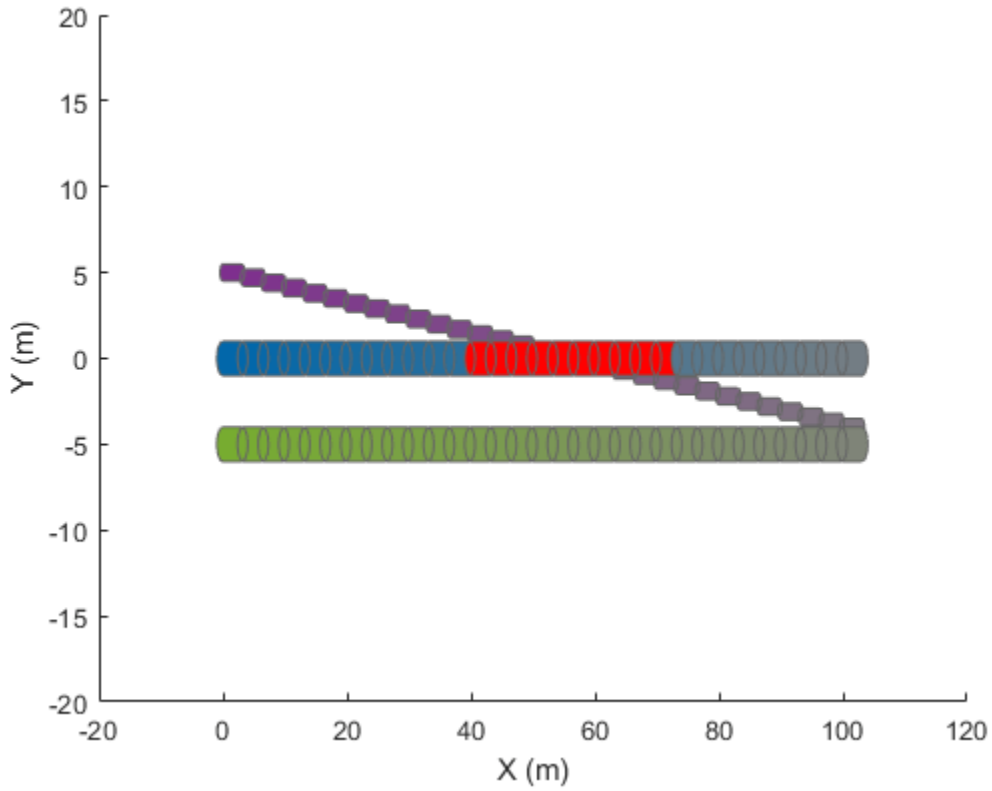
Alter your obstacle locations and geometry dimensions over time. Use the previously generated structure, modify the fields, and update the obstacles using the `updateObstacleGeometry` and `updateObstaclePose` object functions. Reduces the radius of the first obstacle to 0.5 m, and change the path to move it towards the ego body.

```
obsCapsule1.Geometry.Radius = 0.5;

obsCapsule1.States = ...
    [linspace(0,100,numSteps)' ... % x
     linspace(5,-4,numSteps)' ... % y
     zeros(numSteps,1)]; % theta

updateObstacleGeometry(obsList,1,obsCapsule1);
updateObstaclePose(obsList,1,obsCapsule1);
```

Check for Collisions

Visualize the new paths. Show where collisions between the ego body and an obstacle, which the display highlights in red. Notice that collisions between the obstacles are not checked.

```
show(obsList,"TimeStep",[1:numSteps],"ShowCollisions",1);
ylim([-20 20])
xlabel("X (m)")
ylabel("Y (m)")
```



Programmatically check for collisions by using the `checkCollision` object function. The function returns a vector of logical values that indicates the status of each time step. The vector is transposed for display purposes.

```
collisions = checkCollision(obsList)'
```

```
collisions = 1x31 logical array

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

To validate paths with a large number of steps, use the `any` function on the vector of collision values.

```
if any(collisions)
    disp("Collision detected.")
end
```

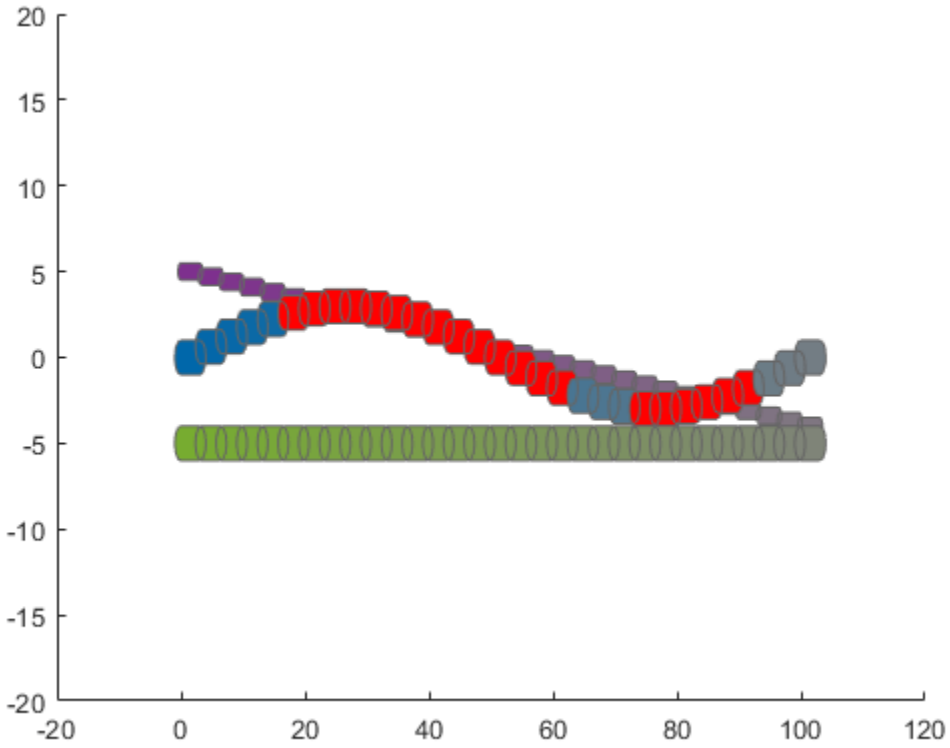
```
Collision detected.
```

Update Ego Path

Specify a new path for the ego body. Visualize the paths again, displaying collisions.

```
egoCapsule1.States = ...
    [linspace(0,100,numSteps)' ... % x
    3*sin(linspace(0,2*pi,numSteps))' ... % y
    zeros(numSteps,1)]; % theta
```

```
updateEgoPose(obsList,1,egoCapsule1);

show(obsList,"TimeStep",[1:numSteps],"ShowCollisions",1);
ylim([-20 20])
```



Input Arguments

`capsuleListObj` — Dynamic capsule list

`dynamicCapsuleList` object | `dynamicCapsuleList3D` object

Dynamic capsule list, specified as a `dynamicCapsuleList` or `dynamicCapsuleList3D` object.

`egoIDs` — IDs of ego bodies to update

vector of positive integers

IDs of ego bodies to update, specified as a vector of positive integers.

`poseStruct` — States for ego bodies

structure | structure array

States for ego bodies, specified as a structure array or structure array, where each structure contains the fields of the structure in the `Geometry` field of the ego body to be updated. Each element of the structure array contains a matrix of states for each ego body. The state matrix size depends on whether you are using a `dynamicCapsuleList` or `dynamicCapsuleList3D` object.

Data Types: `struct`

Output Arguments

`status` — Result of updating ego bodies

N-element column vector

Result of updating ego bodies, specified as *N*-element column vector of ones, zeros, and negative ones. *N* is the number of ego bodies specified in the `egoIDs` argument. Each value indicates whether the body is updated (`1`), not found (`0`), or a duplicate (`-1`). If you specify the same ego ID multiple times in the `egoIDs` input argument, then all entries besides the last are marked as a duplicate.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

Objects

`dynamicCapsuleList` | `dynamicCapsuleList3D`

Functions

`addEgo` | `addObstacle` | `checkCollision` | `egoGeometry` | `egoPose` | `obstacleGeometry` | `obstaclePose` | `removeEgo` | `removeObstacle` | `show` | `updateEgoGeometry` | `updateObstacleGeometry` | `updateObstaclePose`

Topics

“Highway Trajectory Planning Using Frenet Reference Path”

Introduced in R2020b

updateObstacleGeometry

Update geometric properties of obstacles

Syntax

```
updateObstacleGeometry(capsuleListObj,obstacleIDs,geomStruct)
status = updateObstacleGeometry(capsuleListObj,obstacleIDs,geomStruct)
```

Description

`updateObstacleGeometry(capsuleListObj,obstacleIDs,geomStruct)` updates geometry parameters for the specified obstacles in the capsule list. If a specified obstacle ID does not already exist, the function adds a new obstacle with that ID to the list.

`status = updateObstacleGeometry(capsuleListObj,obstacleIDs,geomStruct)` returns an indicator of whether an obstacle is added, updated, or a duplicate.

Examples

Build Ego Body Paths and Check for Collisions with Obstacles

Build an ego body path and maintain obstacle states using the `dynamicCapsuleList` object. Visualize the states of all objects in the environment at different timestamps. Validate the path of the ego body by checking for collisions with obstacles in the environment.

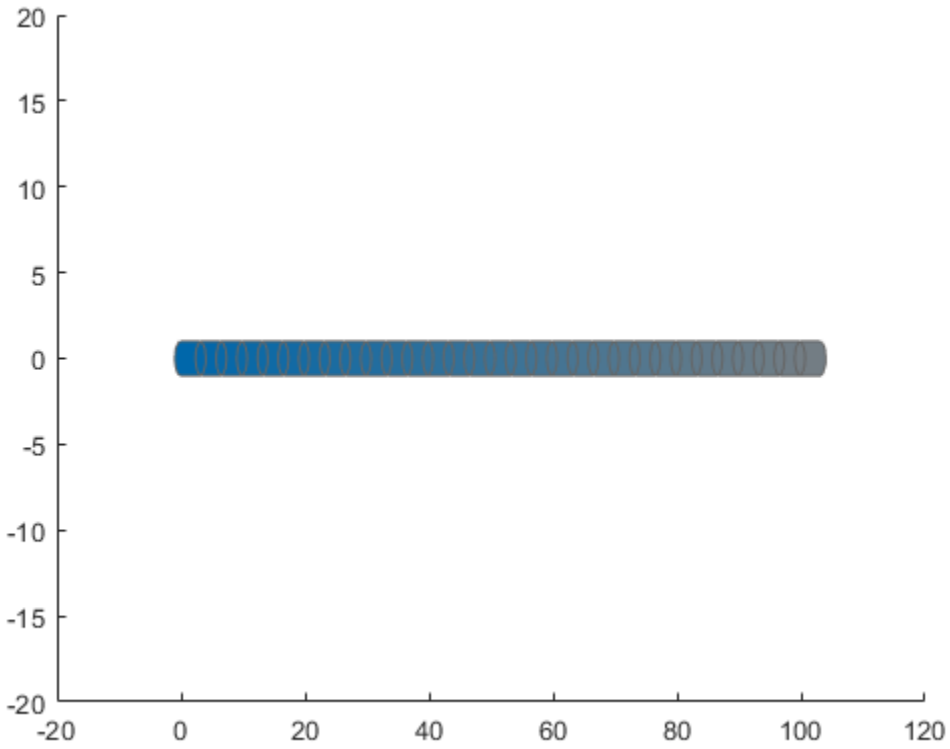
Create the `dynamicCapsuleList` object. Extract the maximum number of steps to use as the number of time stamps for your object paths.

```
obsList = dynamicCapsuleList;
numSteps = obsList.MaxNumSteps;
```

Add Ego Body

Define an ego body by specifying the ID, geometry, and state together in a structure. The capsule geometry has a length of 3 m and radius of 1 m. Specify the state as a linear path from x = 0m to x = 100m.

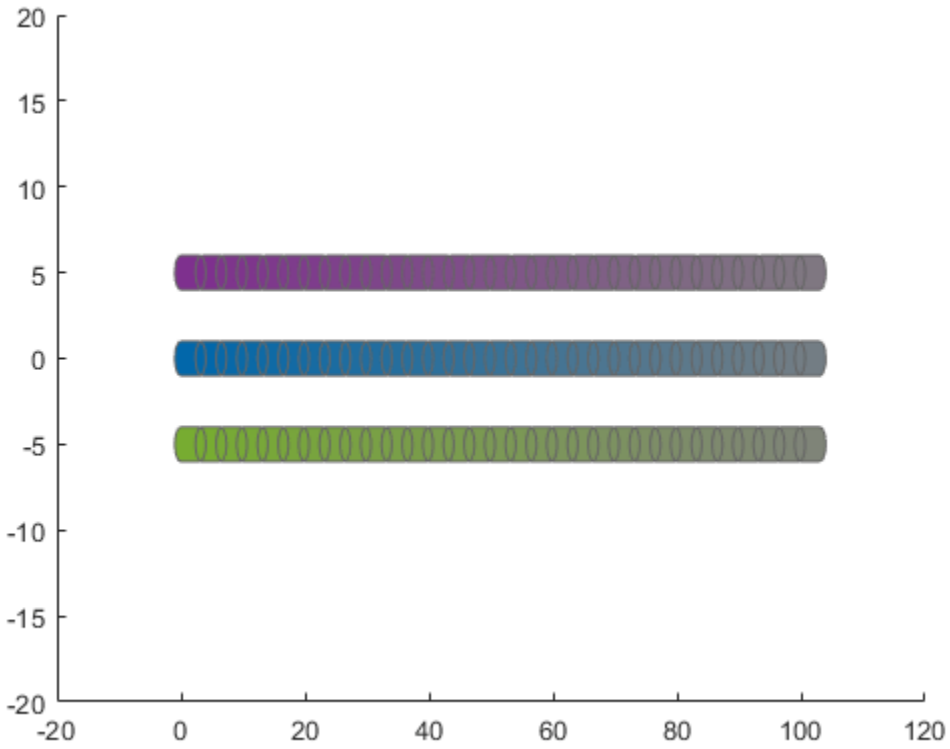
```
egoID1 = 1;
geom = struct("Length",3,"Radius",1,"FixedTransform",eye(3));
states = linspace(0,1,obsList.MaxNumSteps)'.*[100 0 0];

egoCapsule1 = struct('ID',egoID1,'States',states,'Geometry',geom);
addEgo(obsList,egoCapsule1);

show(obsList,"TimeStep",[1:numSteps]);
ylim([-20 20])
```



Add Obstacles

Specify states for two obstacles that are separated from the ego body by 5 m in opposite directions on the *y*-axis.. Assume the obstacles have the same geometry `geom` as the ego body.

```
obsState1 = states + [0 5 0];
obsState2 = states + [0 -5 0];

obsCapsule1 = struct('ID',1,'States',obsState1,'Geometry',geom);
obsCapsule2 = struct('ID',2,'States',obsState2,'Geometry',geom);

addObstacle(obsList,obsCapsule1);
addObstacle(obsList,obsCapsule2);

show(obsList,"TimeStep",[1:numSteps]);
ylim([-20 20])
```



Update Obstacles

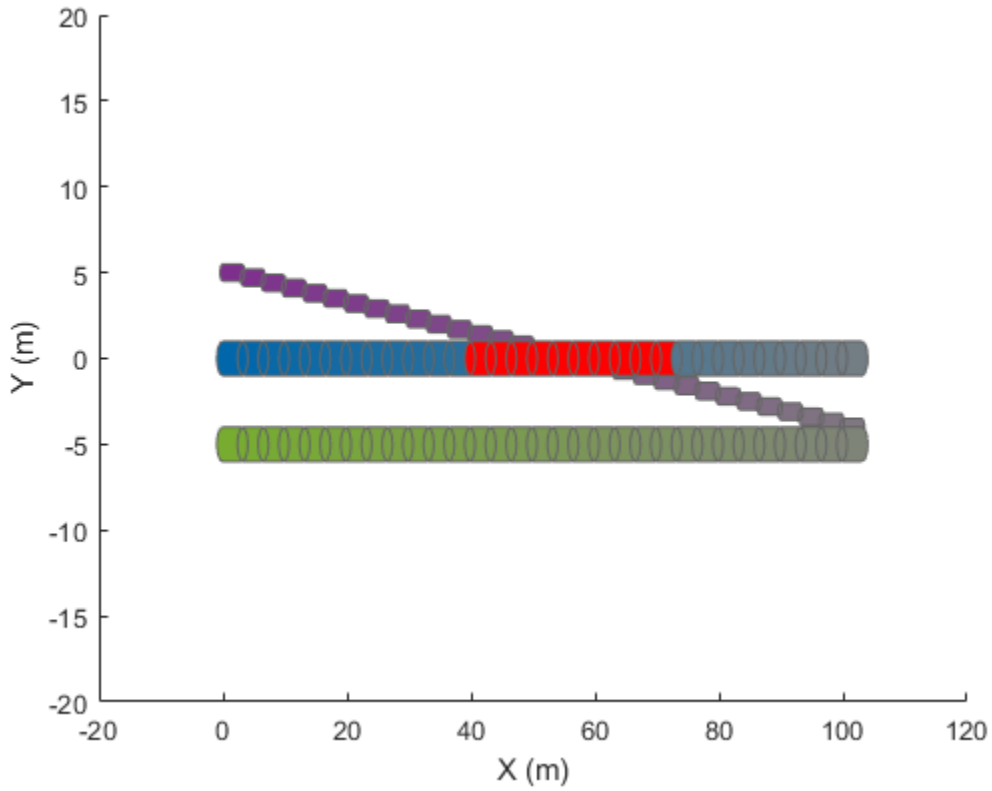
Alter your obstacle locations and geometry dimensions over time. Use the previously generated structure, modify the fields, and update the obstacles using the `updateObstacleGeometry` and `updateObstaclePose` object functions. Reduces the radius of the first obstacle to 0.5 m, and change the path to move it towards the ego body.

```
obsCapsule1.Geometry.Radius = 0.5;

obsCapsule1.States = ...
    [linspace(0,100,numSteps)' ... % x
     linspace(5,-4,numSteps)' ... % y
     zeros(numSteps,1)]; % theta

updateObstacleGeometry(obsList,1,obsCapsule1);
updateObstaclePose(obsList,1,obsCapsule1);
```

Check for Collisions

Visualize the new paths. Show where collisions between the ego body and an obstacle, which the display highlights in red. Notice that collisions between the obstacles are not checked.

```
show(obsList,"TimeStep",[1:numSteps],"ShowCollisions",1);
ylim([-20 20])
xlabel("X (m)")
ylabel("Y (m)")
```



Programmatically check for collisions by using the `checkCollision` object function. The function returns a vector of logical values that indicates the status of each time step. The vector is transposed for display purposes.

```
collisions = checkCollision(obsList)'
```

```
collisions = 1x31 logical array

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

To validate paths with a large number of steps, use the `any` function on the vector of collision values.

```
if any(collisions)
    disp("Collision detected.")
end
```

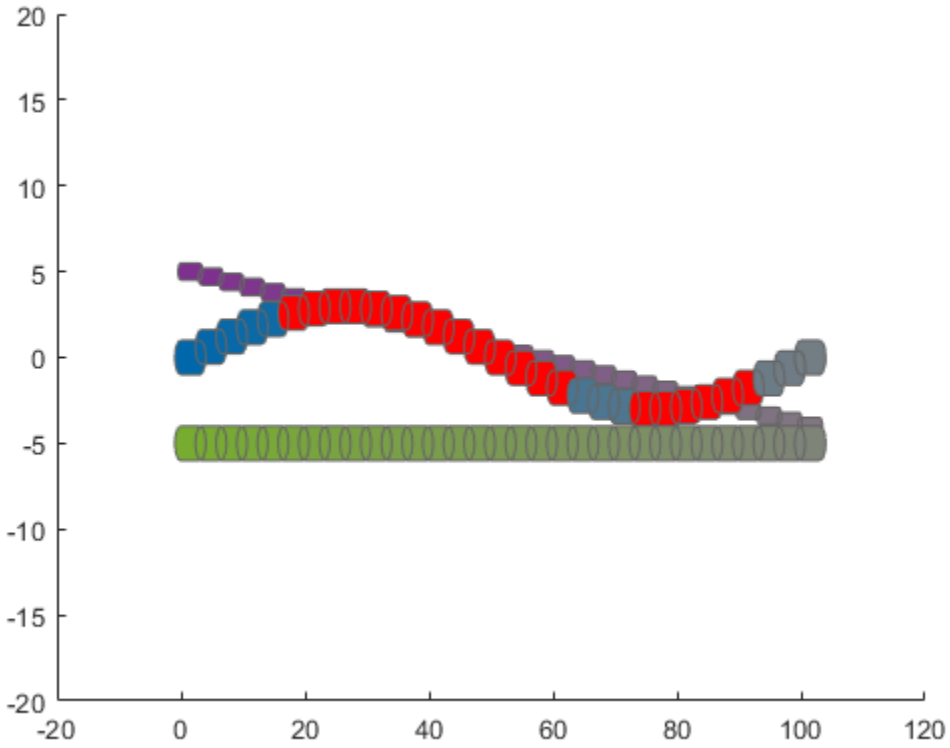
```
Collision detected.
```

Update Ego Path

Specify a new path for the ego body. Visualize the paths again, displaying collisions.

```
egoCapsule1.States = ...
    [linspace(0,100,numSteps)' ... % x
    3*sin(linspace(0,2*pi,numSteps))' ... % y
    zeros(numSteps,1)]; % theta
```

```
updateEgoPose(obsList,1,egoCapsule1);

show(obsList,"TimeStep",[1:numSteps],"ShowCollisions",1);
ylim([-20 20])
```



Input Arguments

`capsuleListObj` — Dynamic capsule list
`dynamicCapsuleList` object | `dynamicCapsuleList3D` object

Dynamic capsule list, specified as a `dynamicCapsuleList` or `dynamicCapsuleList3D` object.

`obstacleIDs` — IDs of obstacles to update
vector of positive integers

IDs of obstacles to update, specified as a vector of positive integers.

`geomStruct` — Geometry parameters for ego bodies
structure | structure array

Geometry parameters for ego bodies, specified as a structure or structure array, where each structure contains the fields of the structure in the `Geometry` field of the obstacle to be updated.. The fields of this structure depend on whether you are using a `dynamicCapsuleList` or `dynamicCapsuleList3D` object.

Data Types: `struct`

Output Arguments

`status` — Result of updating obstacles

N-element column vector

Result of updating obstacles, specified as *N*-element column vector of ones, zeros, and negative ones. *N* is the number of obstacles specified in the `obstacleIDs` argument. Each value indicates whether the obstacle is removed (`1`), not found (`0`), or a duplicate (`-1`). If you specify the same obstacle ID multiple times in the `obstacleIDs` input argument, then all entries besides the last are marked as a duplicate.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

Objects

`dynamicCapsuleList` | `dynamicCapsuleList3D`

Functions

`addEgo` | `addObstacle` | `checkCollision` | `egoGeometry` | `egoPose` | `obstacleGeometry` | `obstaclePose` | `removeEgo` | `removeObstacle` | `show` | `updateEgoGeometry` | `updateEgoPose` | `updateObstaclePose`

Topics

“Highway Trajectory Planning Using Frenet Reference Path”

Introduced in R2020b

updateObstaclePose

Update states of obstacles

Syntax

```
updateObstaclePose(capsuleListObj,obstacleIDs,poseStruct)
status = updateObstaclePose(capsuleListObj,obstacleIDs,poseStruct)
```

Description

`updateObstaclePose(capsuleListObj,obstacleIDs,poseStruct)` updates the states of the specified obstacles in the capsule list. If a specified obstacle ID does not already exist, the function adds a new ego body with that ID to the list.

`status = updateObstaclePose(capsuleListObj,obstacleIDs,poseStruct)` returns an indicator of whether an obstacle is added, updated, or a duplicate.

Examples

Build Ego Body Paths and Check for Collisions with Obstacles

Build an ego body path and maintain obstacle states using the `dynamicCapsuleList` object. Visualize the states of all objects in the environment at different timestamps. Validate the path of the ego body by checking for collisions with obstacles in the environment.

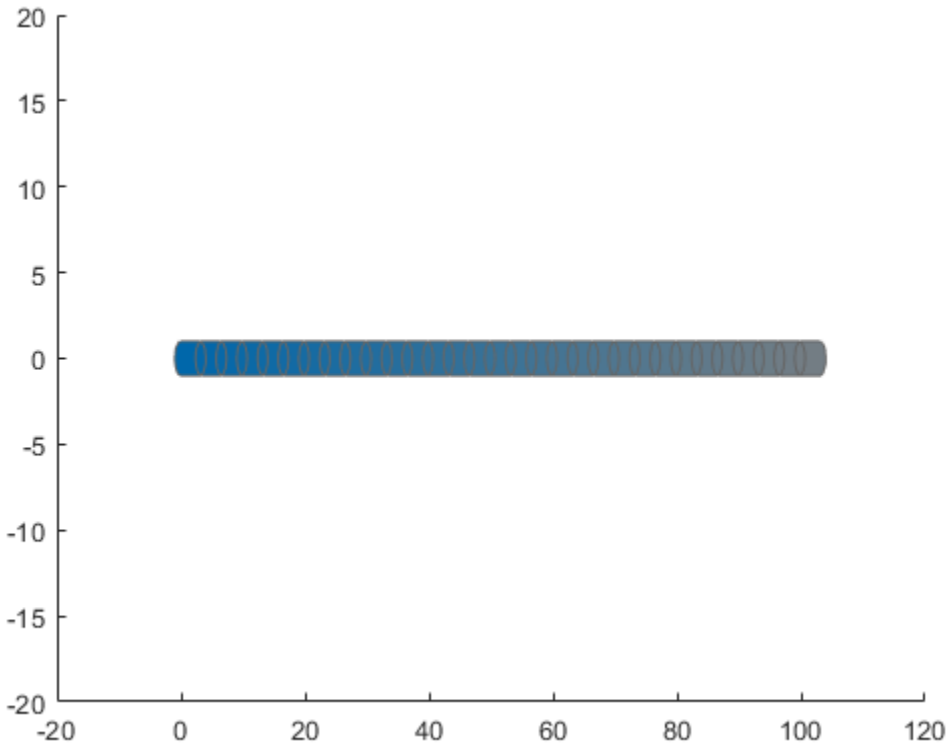
Create the `dynamicCapsuleList` object. Extract the maximum number of steps to use as the number of time stamps for your object paths.

```
obsList = dynamicCapsuleList;
numSteps = obsList.MaxNumSteps;
```

Add Ego Body

Define an ego body by specifying the ID, geometry, and state together in a structure. The capsule geometry has a length of 3 m and radius of 1 m. Specify the state as a linear path from x = 0m to x = 100m.

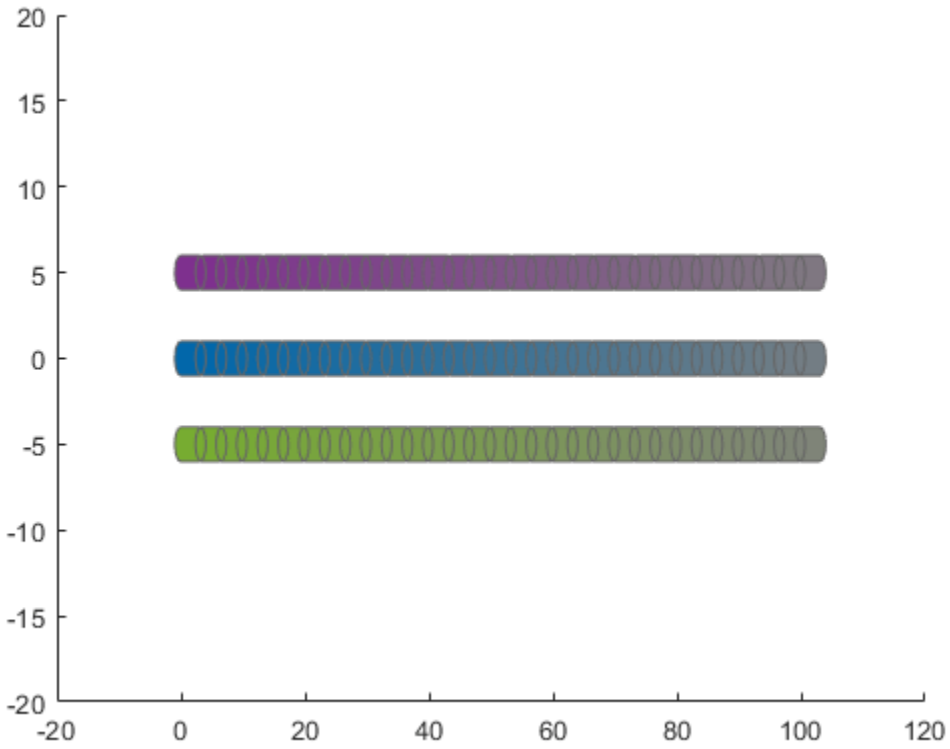
```
egoID1 = 1;
geom = struct("Length",3,"Radius",1,"FixedTransform",eye(3));
states = linspace(0,1,obsList.MaxNumSteps)'.*[100 0 0];

egoCapsule1 = struct('ID',egoID1,'States',states,'Geometry',geom);
addEgo(obsList,egoCapsule1);

show(obsList,"TimeStep",[1:numSteps]);
ylim([-20 20])
```



Add Obstacles

Specify states for two obstacles that are separated from the ego body by 5 m in opposite directions on the *y*-axis.. Assume the obstacles have the same geometry `geom` as the ego body.

```
obsState1 = states + [0 5 0];
obsState2 = states + [0 -5 0];

obsCapsule1 = struct('ID',1,'States',obsState1,'Geometry',geom);
obsCapsule2 = struct('ID',2,'States',obsState2,'Geometry',geom);

addObstacle(obsList,obsCapsule1);
addObstacle(obsList,obsCapsule2);

show(obsList,"TimeStep",[1:numSteps]);
ylim([-20 20])
```



Update Obstacles

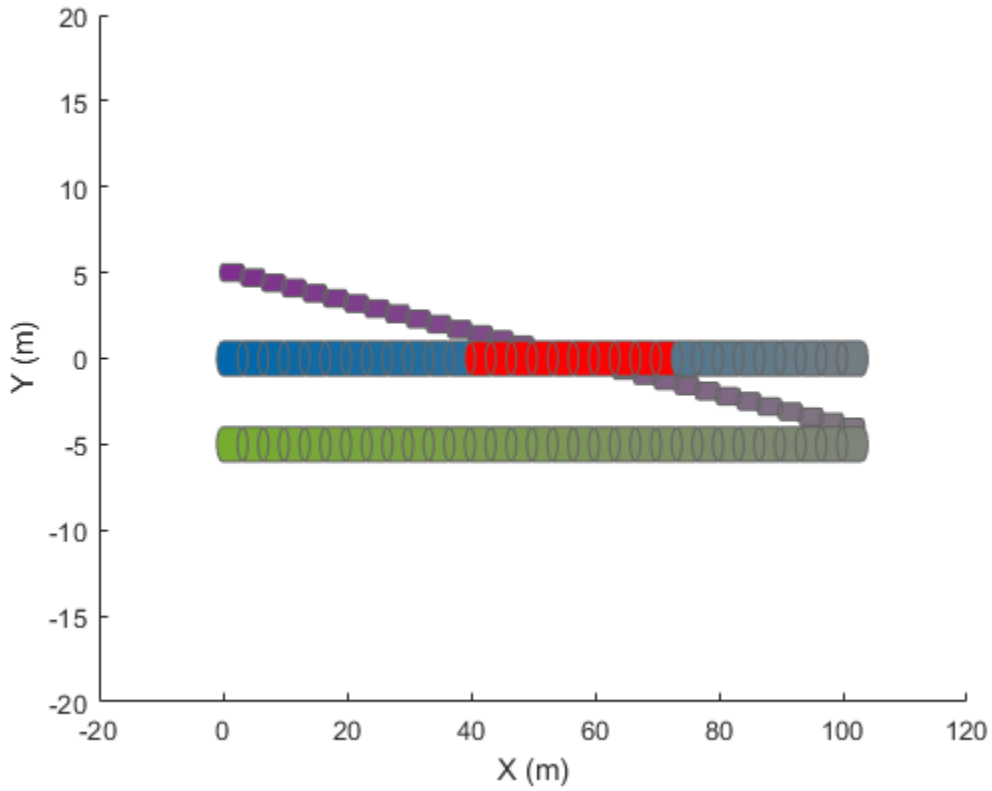
Alter your obstacle locations and geometry dimensions over time. Use the previously generated structure, modify the fields, and update the obstacles using the `updateObstacleGeometry` and `updateObstaclePose` object functions. Reduces the radius of the first obstacle to 0.5 m, and change the path to move it towards the ego body.

```
obsCapsule1.Geometry.Radius = 0.5;

obsCapsule1.States = ...
    [linspace(0,100,numSteps)' ... % x
     linspace(5,-4,numSteps)' ... % y
     zeros(numSteps,1)]; % theta

updateObstacleGeometry(obsList,1,obsCapsule1);
updateObstaclePose(obsList,1,obsCapsule1);
```

Check for Collisions

Visualize the new paths. Show where collisions between the ego body and an obstacle, which the display highlights in red. Notice that collisions between the obstacles are not checked.

```
show(obsList,"TimeStep",[1:numSteps],"ShowCollisions",1);
ylim([-20 20])
xlabel("X (m)")
ylabel("Y (m)")
```



Programmatically check for collisions by using the `checkCollision` object function. The function returns a vector of logical values that indicates the status of each time step. The vector is transposed for display purposes.

```
collisions = checkCollision(obsList)'
```

```
collisions = 1x31 logical array

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

To validate paths with a large number of steps, use the `any` function on the vector of collision values.

```
if any(collisions)
    disp("Collision detected.")
end
```

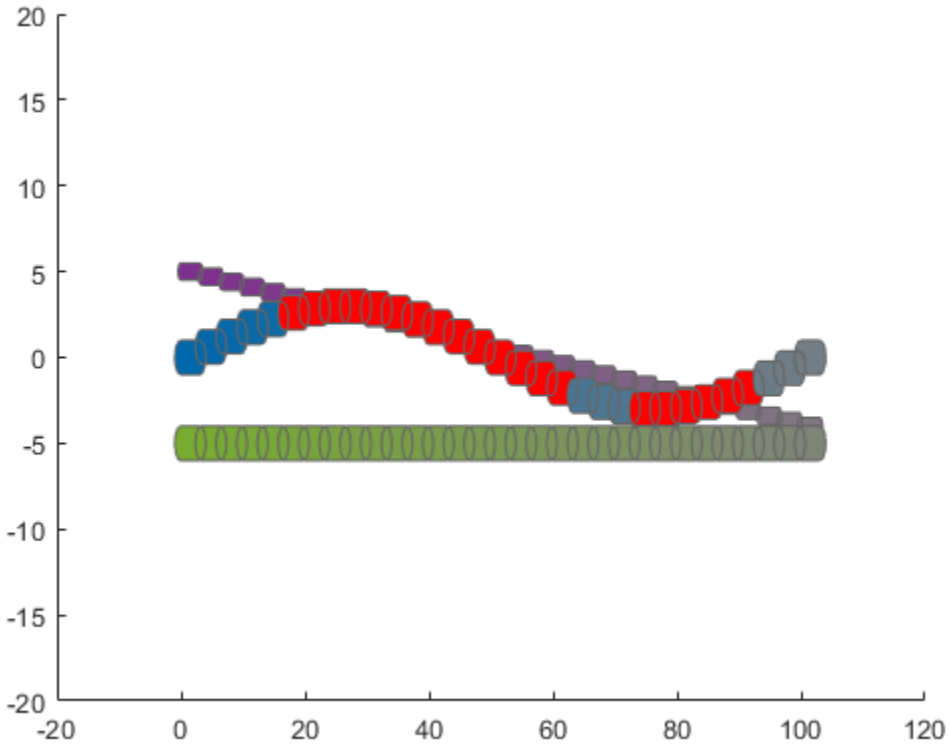
```
Collision detected.
```

Update Ego Path

Specify a new path for the ego body. Visualize the paths again, displaying collisions.

```
egoCapsule1.States = ...
    [linspace(0,100,numSteps)' ... % x
    3*sin(linspace(0,2*pi,numSteps))' ... % y
    zeros(numSteps,1)]; % theta
```

```
updateEgoPose(obsList,1,egoCapsule1);

show(obsList,"TimeStep",[1:numSteps],"ShowCollisions",1);
ylim([-20 20])
```



Input Arguments

`capsuleListObj` — Dynamic capsule list
`dynamicCapsuleList` object | `dynamicCapsuleList3D` object

Dynamic capsule list, specified as a `dynamicCapsuleList` or `dynamicCapsuleList3D` object.

`obstacleIDs` — IDs of obstacles to update
vector of positive integers

IDs of obstacles to update, specified as a vector of positive integers.

`poseStruct` — States for obstacles
structure | structure array

States for ego bodies, specified as a structure or structure array, where each structure contains the fields of the structure in the `Geometry` field of the obstacle to be updated. Each element of the structure array contains a matrix of states for each ego body. The state matrix size depends on whether you are using a `dynamicCapsuleList` or `dynamicCapsuleList3D` object.

Data Types: `struct`

Output Arguments

`status` — Result of updating obstacles

N-element column vector

Result of updating obstacles, specified as *N*-element column vector of ones, zeros, and negative ones. *N* is the number of obstacles specified in the `obstacleIDs` argument. Each value indicates whether the obstacle is removed (`1`), not found (`0`), or a duplicate (`-1`). If you specify the same obstacle ID multiple times in the `obstacleIDs` input argument, then all entries besides the last are marked as a duplicate.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

Objects

`dynamicCapsuleList` | `dynamicCapsuleList3D`

Functions

`addEgo` | `addObstacle` | `checkCollision` | `egoGeometry` | `egoPose` | `obstacleGeometry` | `obstaclePose` | `removeEgo` | `removeObstacle` | `show` | `updateEgoGeometry` | `updateEgoPose` | `updateObstacleGeometry`

Topics

“Highway Trajectory Planning Using Frenet Reference Path”

Introduced in R2020b

dynamicCapsuleList3D

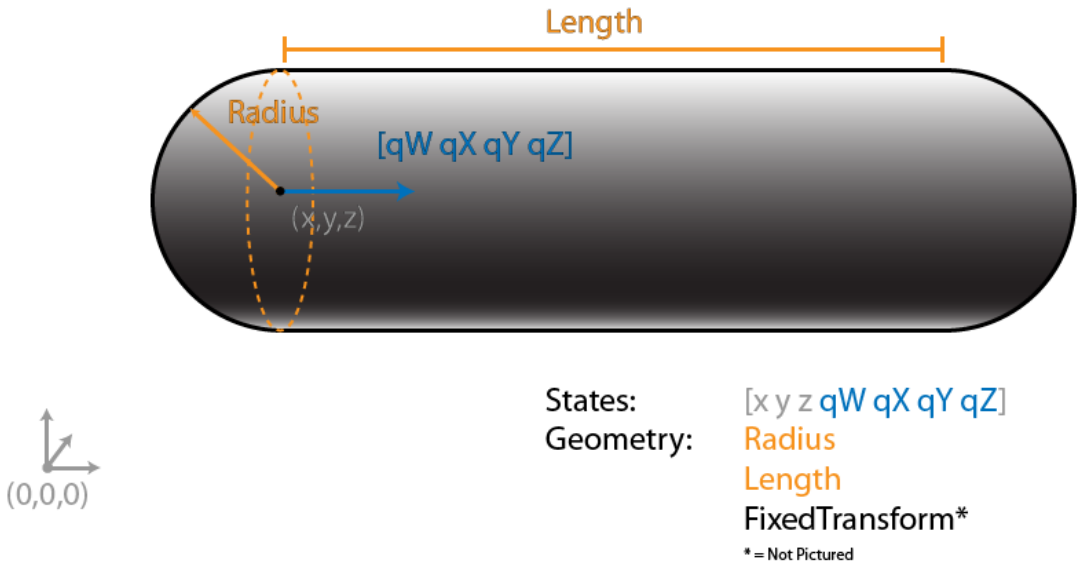
Dynamic capsule-based obstacle list

Description

The `dynamicCapsuleList3D` object manages two lists of capsule-based collision objects in 3-D space. Collision objects are separated into two lists, ego bodies and obstacles. For ego bodies and obstacles in 2-D, see the `dynamicCapsuleList` object.

Each collision object in the two lists has three key elements:

- **ID** –– Integer that identifies each object, stored in the `EgoIDs` property for ego bodies and the `ObstacleIDs` property for obstacles.
- **States** –– Location and orientation of the object as an *M*-by-6 matrix, where each row is of form `[x y z qW qX qY qZ]`, and *M* is the number of states along the path of the object in the world frame. The list of states assumes each state is separated by a fixed time interval. *xyz*-positions are in meters, and the orientation is a four-element quaternion vector. The default local origin is located at the center of the left hemisphere of the capsule.
- **Geometry** –– Size of the capsule-based object based on the specified length and radius. The radius applies to the spherical ends, and the length applies to the cylinder length. To shift the capsule geometry and local origin relative to the default origin point, specify a 4-by-4 transform relative to the local frame of the capsule. To keep the default transform, specify `eye(4)`.



Use the object functions to dynamically add, remove, and update the geometries and states of the various objects in your environment. To add an ego body, see the `addEgo` object function. To add an obstacle, see the `addObstacle` object function.

After specifying all the object paths, validate the ego-body paths and check for collisions with obstacles at every step using the `checkCollision` object function. The function only checks if an ego body collides with an obstacle, ignoring collisions between only obstacles or only ego bodies.

Creation

Syntax

```
obstacleList = dynamicCapsuleList3D
```

Description

`obstacleList = dynamicCapsuleList3D` creates a dynamic capsule-based obstacle list with no ego bodies or obstacles. To begin building an obstacle list, use the `addEgo` or `addObstacle` object functions.

Properties

`MaxNumSteps` — Maximum number of time steps in obstacle list
`31` (default) | positive integer

Maximum number of time steps in the obstacle list, specified as a positive integer. The number of steps determines to the maximum length of the `States` field for a specific ego body or obstacle.

Data Types: `double`

`EgoIDs` — List of IDs for ego bodies
vector of positive integers

This property is read-only.

List of identifiers for ego bodies, returned as a vector of positive integers.

Data Types: `double`

`ObstacleIDs` — IDs for obstacles
vector of positive integers

This property is read-only.

List of identifiers for obstacles, returned as a vector of positive integers.

Data Types: `double`

`NumObstacles` — Number of obstacles in list
integer

This property is read-only.

Number of obstacles in list, returned as an integer.

Data Types: `double`

`NumEgos` — Number of ego bodies in list
integer

This property is read-only.

Number of ego bodies in list, returned as an integer.

Data Types: `double`

Object Functions

addEgo	Add ego bodies to 3D capsule list
addObstacle	Add obstacles to 3-D capsule list
checkCollision	Check for collisions between ego bodies and obstacles
egoGeometry	Geometric properties of ego bodies
egoPose	Poses of ego bodies
obstacleGeometry	Geometric properties of obstacles
obstaclePose	Poses of obstacles
removeEgo	Remove ego bodies from capsule list
removeObstacle	Remove obstacles from capsule list
show	Display ego bodies and obstacles in environment
updateEgoGeometry	Update geometric properties of ego bodies
updateEgoPose	Update states of ego bodies
updateObstacleGeometry	Update geometric properties of obstacles
updateObstaclePose	Update states of obstacles

Examples

Build 3-D Ego Body Paths and Check for Collisions with 3-D Obstacles

Build an ego body path and maintain obstacle states using the `dynamicCapsuleList3D` object. Visualize the states of all objects in the environment at different timestamps. Validate the path of the ego body by checking for collisions with obstacles in the environment.

Create the `dynamicCapsuleList3D` object. Extract the maximum number of steps to use as the number of time stamps for your object paths.

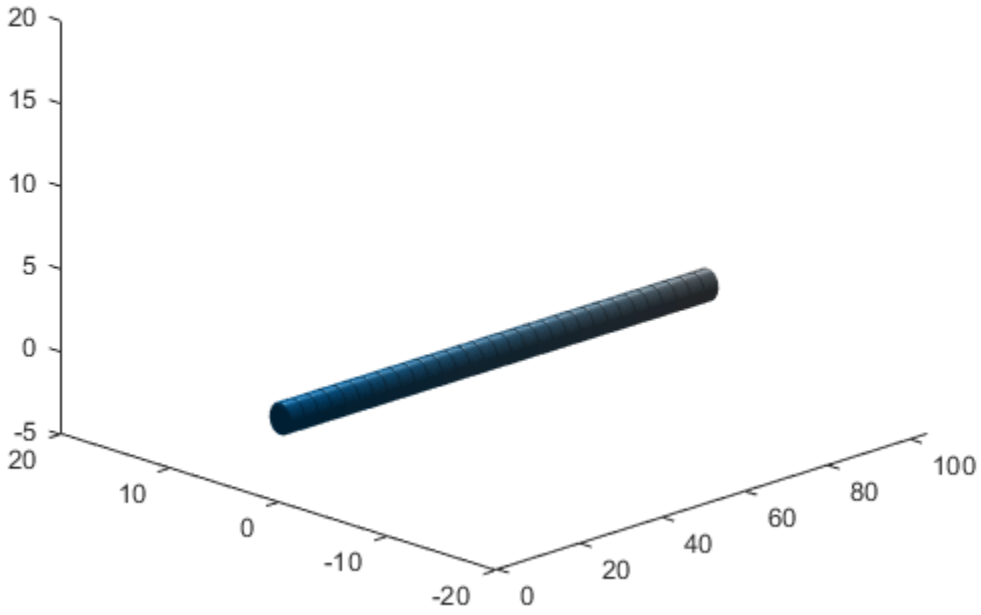
```
obsList = dynamicCapsuleList3D;
numSteps = obsList.MaxNumSteps;
```

Add Ego Body

Define an ego body by specifying the ID, geometry, and state together in a structure. The capsule geometry has a length of 3 m and radius of 1 m. Specify the state as a linear path from x = 0 m *to* x = 100 m.

```
egoID1 = 1;
geom = struct("Length",3,"Radius",1,"FixedTransform",eye(4));
states = linspace(0,1,obsList.MaxNumSteps)'.*[100 0 0];
states = [states ones(numSteps,2) zeros(numSteps,2)];

egoCapsule1 = struct('ID',egoID1,'States',states,'Geometry',geom);
addEgo(obsList,egoCapsule1);

show(obsList,"TimeStep",1:numSteps);
ylim([-20 20])
zlim([-5 20])
```

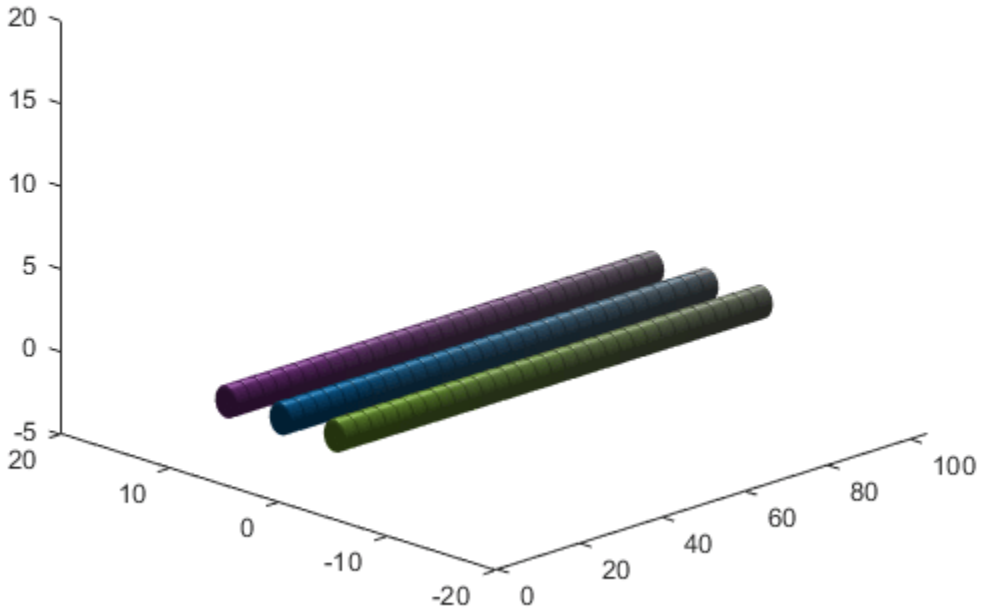
```
view(-45,25)
hold on
```



Add Obstacles

Specify states for two obstacles that are separated from the ego body by 5 m in opposite directions on the *y*-axis. Assume the obstacles have the same geometry `geom` as the ego body.

```
obsState1 = states + [0 5 0 0 0 0 0];
obsState2 = states + [0 -5 0 0 0 0 0];

obsCapsule1 = struct('ID',1,'States',obsState1,'Geometry',geom);
obsCapsule2 = struct('ID',2,'States',obsState2,'Geometry',geom);

addObstacle(obsList,obsCapsule1);
addObstacle(obsList,obsCapsule2);

cla
show(obsList,"TimeStep",1:numSteps);
```



Update Obstacles

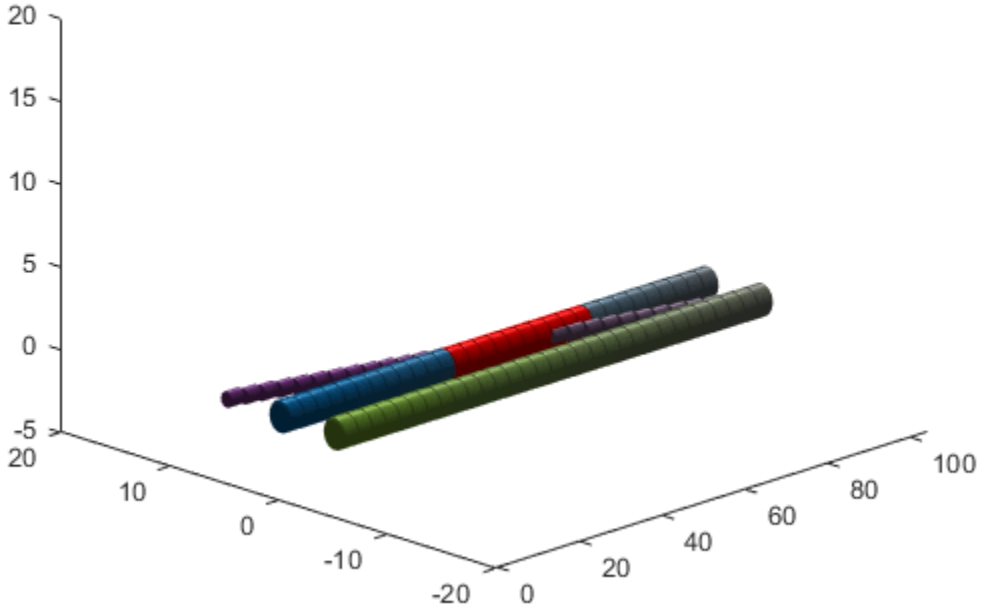
Alter your obstacle locations and geometry dimensions over time. Use the previously generated structure, modify the fields, and update the obstacles using the `updateObstacleGeometry` and `updateObstaclePose` object functions. Reduce the radius of the first obstacle to 0.5 m, and change the path to move it towards the ego body.

```
obsCapsule1.Geometry.Radius = 0.5;

obsCapsule1.States = ...
    [linspace(0,100,numSteps)' ... % x
     linspace(5,-4,numSteps)' ... % y
     zeros(numSteps,1) ... % z
     ones(numSteps,2) zeros(numSteps,2)]; % quaternion                                % quaternio

updateObstacleGeometry(obsList,1,obsCapsule1);
updateObstaclePose(obsList,1,obsCapsule1);
```

Check for Collisions

Visualize the new paths. Show where collisions between the ego body and an obstacle occur, which the display highlights in red. Notice that collisions between the obstacles are not checked.

```
cla
show(obsList,"TimeStep",1:numSteps,"ShowCollisions",1);
```



Programmatically check for collisions bu using the `checkCollision` object function. The function returns a vector of logical values that indicates the collision status at each time step. The vector is transposed for display purposes.

```
collisions = checkCollision(obsList)'
```

```
collisions = 1x31 logical array

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

To validate paths with a large number of steps, use the `any` function on the vector of collision values.

```
if any(collisions)
    disp("Collision detected.")
end
```

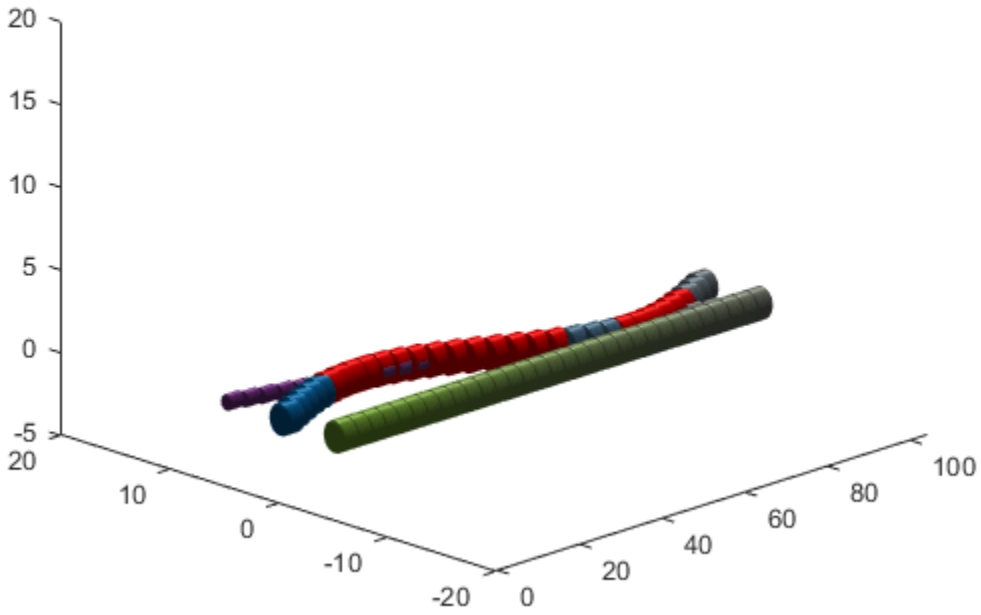
```
Collision detected.
```

Update Ego Path

Specify a new path for the ego body. Visualize the paths again, displaying collisions.

```
egoCapsule1.States = ...
    [linspace(0,100,numSteps)' ... % x
    3*sin(linspace(0,2*pi,numSteps))' ... % y
    zeros(numSteps,1)... % z
    ones(numSteps,2) zeros(numSteps,2)]; %quaternion                                % quaternio
```

```
updateEgoPose(obsList,1,egoCapsule1);

cla
show(obsList,"TimeStep",1:numSteps,"ShowCollisions",1);
```



Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

Objects

`dynamicCapsuleList`

Functions

`addEgo` | `addObstacle` | `checkCollision` | `egoGeometry` | `egoPose` | `obstacleGeometry` | `obstaclePose` | `removeEgo` | `removeObstacle` | `show` | `updateEgoGeometry` | `updateEgoPose` | `updateObstacleGeometry` | `updateObstaclePose`

Topics

“Highway Trajectory Planning Using Frenet Reference Path”

Introduced in R2020b

addEgo

Add ego bodies to 3D capsule list

Syntax

```
addEgo(capsuleListObj,egoStruct)
status = addEgo(capsuleListObj,egoStruct)
```

Description

`addEgo(capsuleListObj,egoStruct)` adds one or more ego bodies to the 3-D dynamic capsule list with the specified ID, state, and geometry values given in `egoStruct`.

`status = addEgo(capsuleListObj,egoStruct)` additionally returns an indicator of whether each specified ego body was added, updated, or a duplicate.

Examples

Build 3-D Ego Body Paths and Check for Collisions with 3-D Obstacles

Build an ego body path and maintain obstacle states using the `dynamicCapsuleList3D` object. Visualize the states of all objects in the environment at different timestamps. Validate the path of the ego body by checking for collisions with obstacles in the environment.

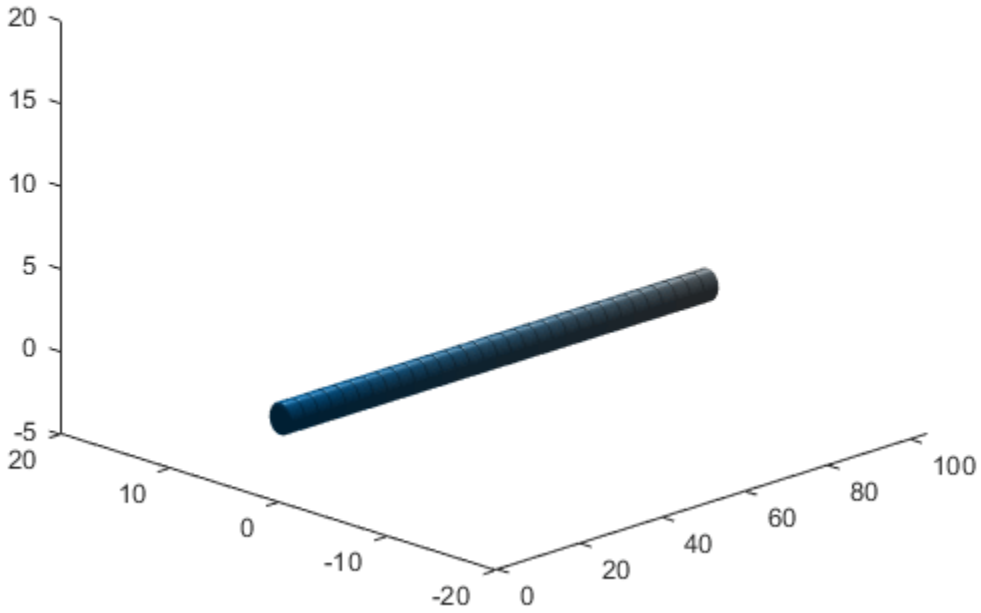
Create the `dynamicCapsuleList3D` object. Extract the maximum number of steps to use as the number of time stamps for your object paths.

```
obsList = dynamicCapsuleList3D;
numSteps = obsList.MaxNumSteps;
```

Add Ego Body

Define an ego body by specifying the ID, geometry, and state together in a structure. The capsule geometry has a length of 3 m and radius of 1 m. Specify the state as a linear path from $x = 0$ m *to* $x = 100$ m.

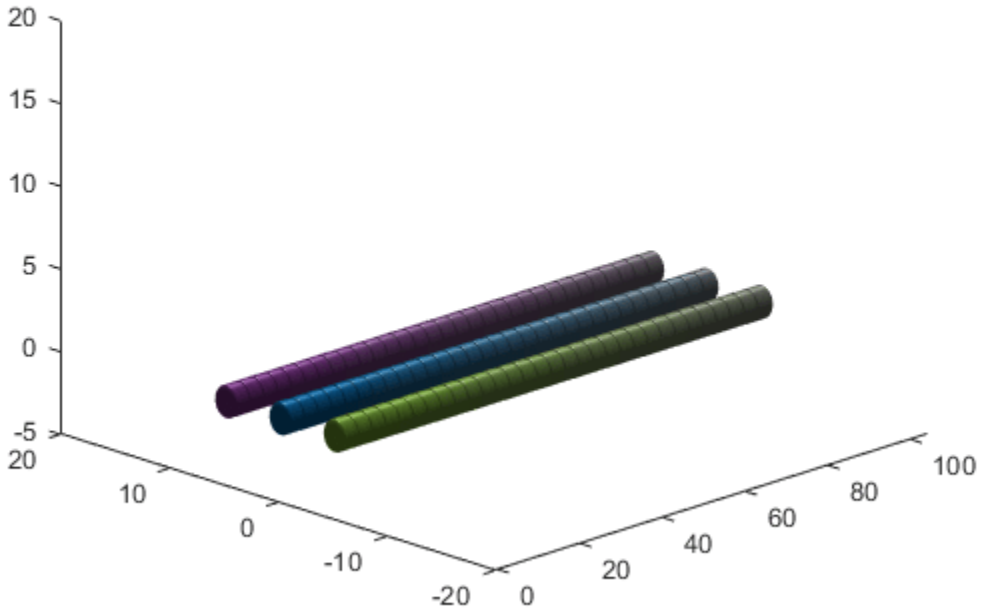
```
egoID1 = 1;
geom = struct("Length",3,"Radius",1,"FixedTransform",eye(4));
states = linspace(0,1,obsList.MaxNumSteps)'.*[100 0 0];
states = [states ones(numSteps,2) zeros(numSteps,2)];

egoCapsule1 = struct('ID',egoID1,'States',states,'Geometry',geom);
addEgo(obsList,egoCapsule1);

show(obsList,"TimeStep",1:numSteps);
ylim([-20 20])
zlim([-5 20])
view(-45,25)
hold on
```



Add Obstacles

Specify states for two obstacles that are separated from the ego body by 5 m in opposite directions on the *y*-axis. Assume the obstacles have the same geometry `geom` as the ego body.

```
obsState1 = states + [0 5 0 0 0 0 0];
obsState2 = states + [0 -5 0 0 0 0 0];

obsCapsule1 = struct('ID',1,'States',obsState1,'Geometry',geom);
obsCapsule2 = struct('ID',2,'States',obsState2,'Geometry',geom);

addObstacle(obsList,obsCapsule1);
addObstacle(obsList,obsCapsule2);

cla
show(obsList,"TimeStep",1:numSteps);
```



Update Obstacles

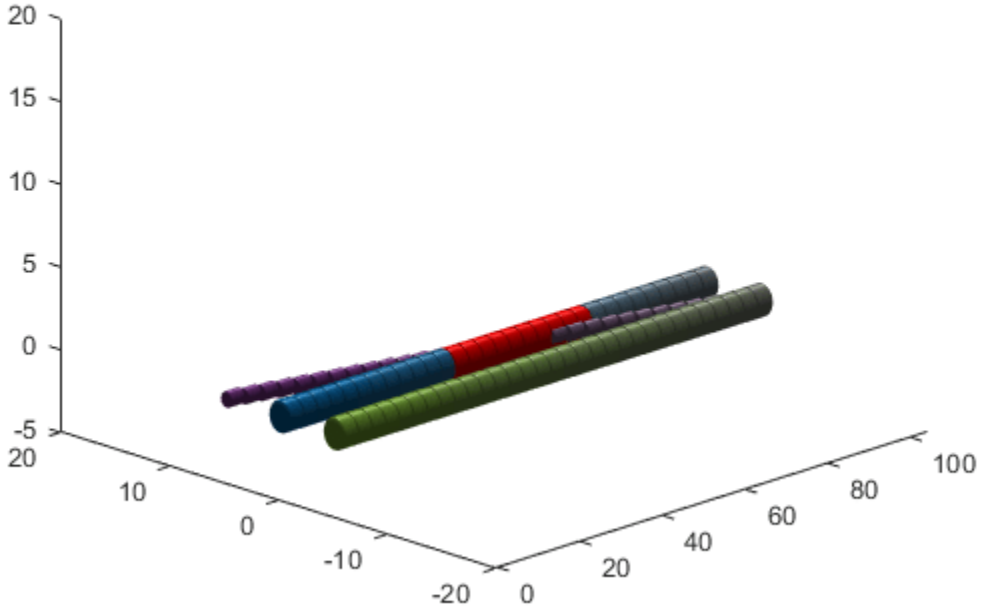
Alter your obstacle locations and geometry dimensions over time. Use the previously generated structure, modify the fields, and update the obstacles using the `updateObstacleGeometry` and `updateObstaclePose` object functions. Reduce the radius of the first obstacle to 0.5 m, and change the path to move it towards the ego body.

```
obsCapsule1.Geometry.Radius = 0.5;

obsCapsule1.States = ...
    [linspace(0,100,numSteps)' ... % x
     linspace(5,-4,numSteps)' ... % y
     zeros(numSteps,1) ... % z
     ones(numSteps,2) zeros(numSteps,2)]; % quaternion                                % quaternio

updateObstacleGeometry(obsList,1,obsCapsule1);
updateObstaclePose(obsList,1,obsCapsule1);
```

Check for Collisions

Visualize the new paths. Show where collisions between the ego body and an obstacle occur, which the display highlights in red. Notice that collisions between the obstacles are not checked.

```
cla
show(obsList,"TimeStep",1:numSteps,"ShowCollisions",1);
```



Programmatically check for collisions bu using the `checkCollision` object function. The function returns a vector of logical values that indicates the collision status at each time step. The vector is transposed for display purposes.

```
collisions = checkCollision(obsList)'
```

```
collisions = 1x31 logical array

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

To validate paths with a large number of steps, use the `any` function on the vector of collision values.

```
if any(collisions)
    disp("Collision detected.")
end
```

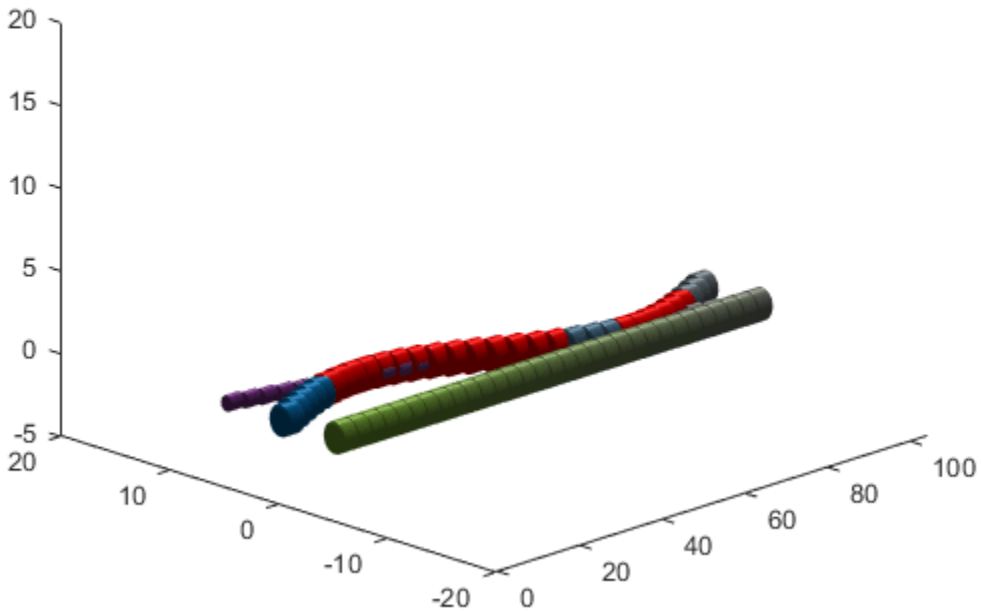
```
Collision detected.
```

Update Ego Path

Specify a new path for the ego body. Visualize the paths again, displaying collisions.

```
egoCapsule1.States = ...
    [linspace(0,100,numSteps)' ... % x
    3*sin(linspace(0,2*pi,numSteps))' ... % y
    zeros(numSteps,1)... % z
    ones(numSteps,2) zeros(numSteps,2)]; %quaternion                                % quaternio
```

```
updateEgoPose(obsList,1,egoCapsule1);

cla
show(obsList,"TimeStep",1:numSteps,"ShowCollisions",1);
```



Input Arguments

`capsuleListObj` — Dynamic capsule list

`dynamicCapsuleList3D` object

Dynamic capsule list, specified as a `dynamicCapsuleList3D` object.

`egoStruct` — Ego body parameters

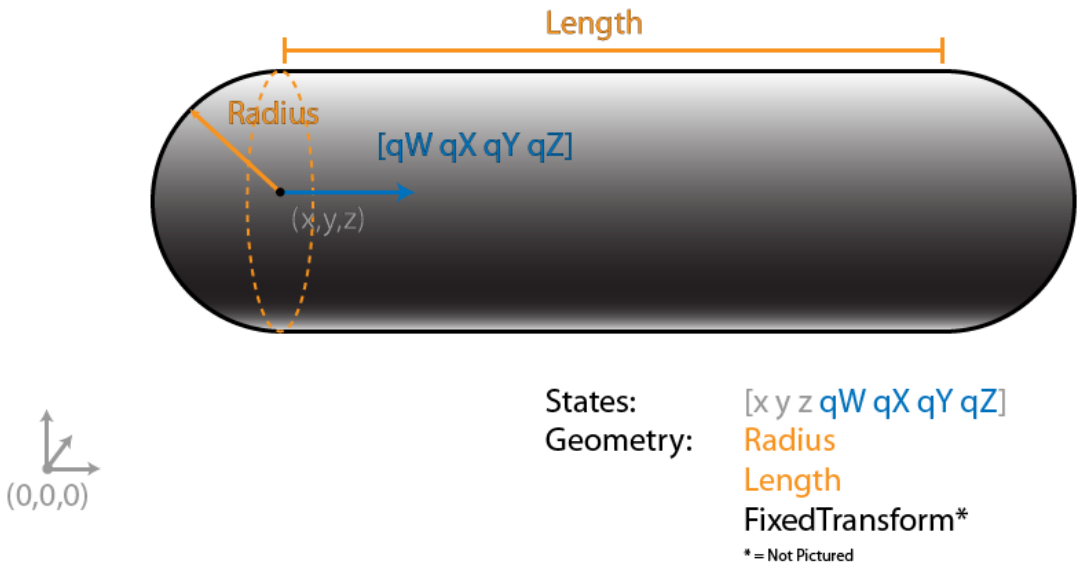
structure | structure array

Ego body parameters, specified as an *N*-element structure or a structure array, where *N* is the number of added ego bodies. The fields of each structure define the ID, geometry, and states of an ego body:

- `ID` -– Integer that identifies each object. Stored in the `EgoIDs` property of the `dynamicCapsuleList3D` object specified by the `capsuleListObj` argument.
- `States` -– Location and orientation of the object as an *M*-by-6 matrix, where each row is of form `[x y z qW qX qY qZ]`, and *M* is the number of states along the path of the object in the world frame. The list of states assumes each state is separated by a fixed time interval. *xyz*-positions are

in meters, and the orientation is a four-element quaternion vector. The default local origin is located at the center of the left hemisphere of the capsule.

- `Geometry` -– Structure with fields `Length`, `Radius`, and `FixedTransform`. These fields define the size of the capsule-based object using the specified length for the cylinder and hemisphere radius for the end caps. To shift the capsule geometry from the default origin, specify the `FixedTransform` field as a fixed transform relative to the local frame of the capsule. To keep the default capsule origin, specify the transform as `eye(4)`.



Output Arguments

`status` — Result of adding ego bodies

N-element column vector

Result of adding ego bodies, returned as a *N*-element column vector of ones, zeros, and negative ones. *N* is the number of ego bodies specified in the `egoStruct` argument. Each value indicates whether the associated body is added (`1`), updated (`0`), or a duplicate (`-1`). While adding ego bodies, if multiple structures with the same body ID are found in the structure array `egoStruct`, then the function marks the previous entry as duplicate and ignores it.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

Objects

`dynamicCapsuleList` | `dynamicCapsuleList3D`

Functions
`addObstacle` | `checkCollision` | `egoGeometry` | `egoPose` | `obstacleGeometry` | `obstaclePose` | `removeEgo` | `removeObstacle` | `show` | `updateEgoGeometry` | `updateEgoPose` | `updateObstacleGeometry` | `updateObstaclePose`

Topics
“Highway Trajectory Planning Using Frenet Reference Path”

Introduced in R2020b

addObstacle

Add obstacles to 3-D capsule list

Syntax

```
addObstacle(capsuleListObj,obstacleStruct)
status = addObstacle(capsuleListObj,obstacleStruct)
```

Description

`addObstacle(capsuleListObj,obstacleStruct)` adds one or more obstacles to the 3-D dynamic capsule list with the specified ID, state, and geometry values given in `obstacleStruct`.

`status = addObstacle(capsuleListObj,obstacleStruct)` additionally returns an indicator of whether each specified obstacle was added, updated, or a duplicate.

Examples

Build 3-D Ego Body Paths and Check for Collisions with 3-D Obstacles

Build an ego body path and maintain obstacle states using the `dynamicCapsuleList3D` object. Visualize the states of all objects in the environment at different timestamps. Validate the path of the ego body by checking for collisions with obstacles in the environment.

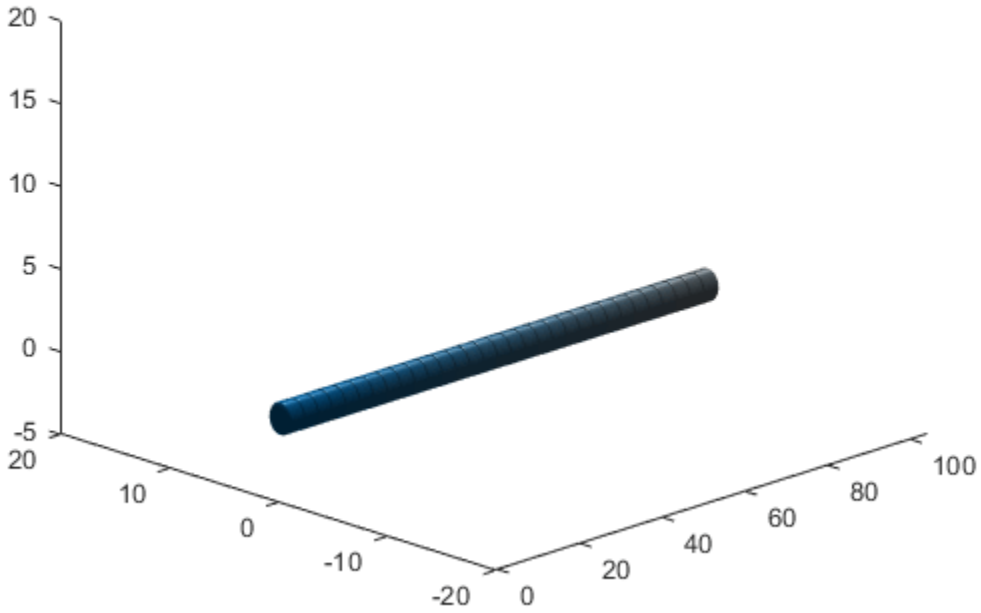
Create the `dynamicCapsuleList3D` object. Extract the maximum number of steps to use as the number of time stamps for your object paths.

```
obsList = dynamicCapsuleList3D;
numSteps = obsList.MaxNumSteps;
```

Add Ego Body

Define an ego body by specifying the ID, geometry, and state together in a structure. The capsule geometry has a length of 3 m and radius of 1 m. Specify the state as a linear path from $x = 0$ m *to* $x = 100$ m.

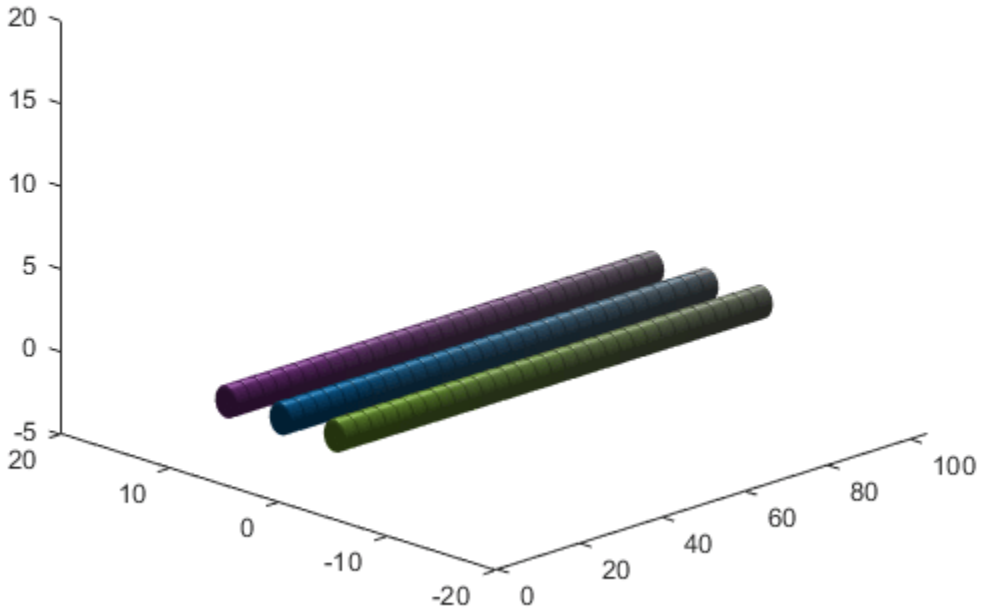
```
egoID1 = 1;
geom = struct("Length",3,"Radius",1,"FixedTransform",eye(4));
states = linspace(0,1,obsList.MaxNumSteps)'.*[100 0 0];
states = [states ones(numSteps,2) zeros(numSteps,2)];

egoCapsule1 = struct('ID',egoID1,'States',states,'Geometry',geom);
addEgo(obsList,egoCapsule1);

show(obsList,"TimeStep",1:numSteps);
ylim([-20 20])
zlim([-5 20])
view(-45,25)
hold on
```



Add Obstacles

Specify states for two obstacles that are separated from the ego body by 5 m in opposite directions on the *y*-axis. Assume the obstacles have the same geometry `geom` as the ego body.

```
obsState1 = states + [0 5 0 0 0 0 0];
obsState2 = states + [0 -5 0 0 0 0 0];

obsCapsule1 = struct('ID',1,'States',obsState1,'Geometry',geom);
obsCapsule2 = struct('ID',2,'States',obsState2,'Geometry',geom);

addObstacle(obsList,obsCapsule1);
addObstacle(obsList,obsCapsule2);

cla
show(obsList,"TimeStep",1:numSteps);
```



Update Obstacles

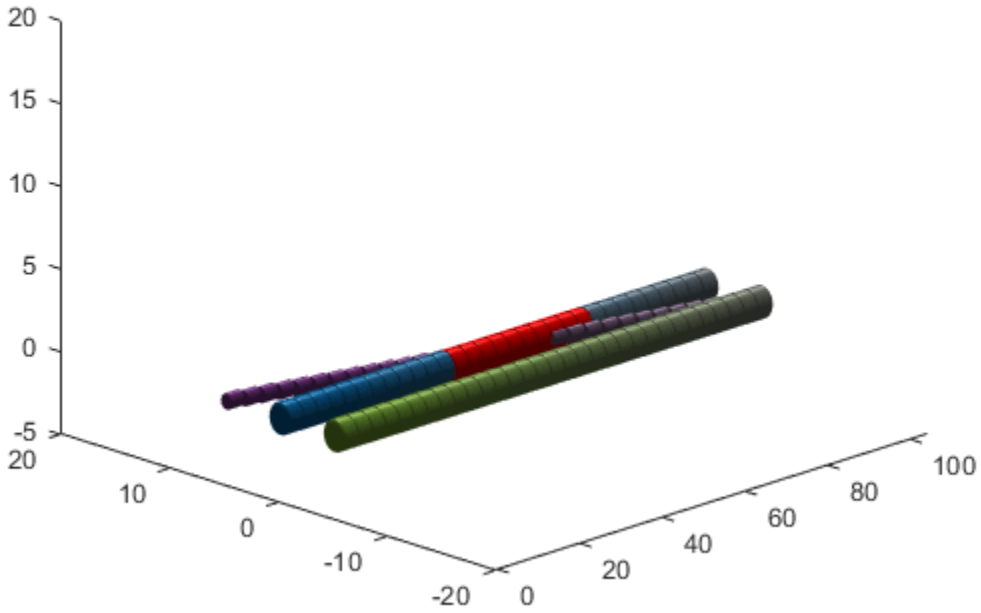
Alter your obstacle locations and geometry dimensions over time. Use the previously generated structure, modify the fields, and update the obstacles using the `updateObstacleGeometry` and `updateObstaclePose` object functions. Reduce the radius of the first obstacle to 0.5 m, and change the path to move it towards the ego body.

```
obsCapsule1.Geometry.Radius = 0.5;

obsCapsule1.States = ...
    [linspace(0,100,numSteps)' ... % x
     linspace(5,-4,numSteps)' ... % y
     zeros(numSteps,1) ... % z
     ones(numSteps,2) zeros(numSteps,2)]; % quaternion                                % quaternio

updateObstacleGeometry(obsList,1,obsCapsule1);
updateObstaclePose(obsList,1,obsCapsule1);
```

Check for Collisions

Visualize the new paths. Show where collisions between the ego body and an obstacle occur, which the display highlights in red. Notice that collisions between the obstacles are not checked.

```
cla
show(obsList,"TimeStep",1:numSteps,"ShowCollisions",1);
```



Programmatically check for collisions bu using the `checkCollision` object function. The function returns a vector of logical values that indicates the collision status at each time step. The vector is transposed for display purposes.

```
collisions = checkCollision(obsList)'
```

```
collisions = 1x31 logical array

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

To validate paths with a large number of steps, use the `any` function on the vector of collision values.

```
if any(collisions)
    disp("Collision detected.")
end
```

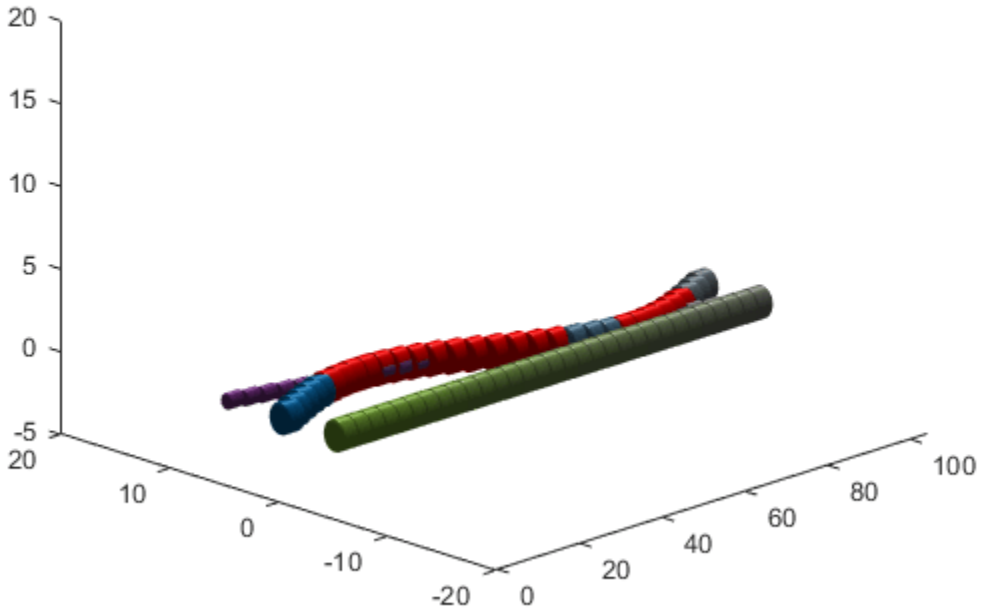
```
Collision detected.
```

Update Ego Path

Specify a new path for the ego body. Visualize the paths again, displaying collisions.

```
egoCapsule1.States = ...
    [linspace(0,100,numSteps)' ... % x
    3*sin(linspace(0,2*pi,numSteps))' ... % y
    zeros(numSteps,1)... % z
    ones(numSteps,2) zeros(numSteps,2)]; %quaternion                                    % quaternio
```

```
updateEgoPose(obsList,1,egoCapsule1);

cla
show(obsList,"TimeStep",1:numSteps,"ShowCollisions",1);
```



Input Arguments

`capsuleListObj` — Dynamic capsule-list

`dynamicCapsuleList3D` object

Dynamic capsule-list, specified as a `dynamicCapsuleList3D` object.

Output Arguments

`obstacleStruct` — Obstacle parameters

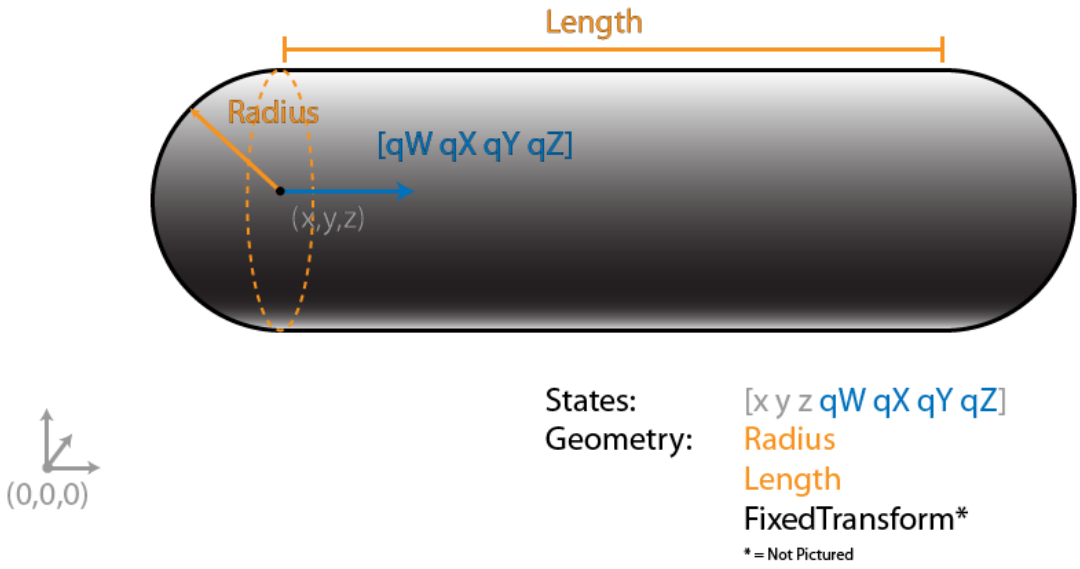
structure | structure array

Obstacle parameters, specified as an *N*-element structure or a structure array, where *N* is the number of added ego bodies. The fields of each structure define the ID, geometry, and states of an obstacle:

- `ID` –– Integer that identifies each object. Stored in the `ObstacleIDs` property of the `dynamicCapsuleList3D` object specified by the `capsuleListObj` argument.
- `States` –– Location and orientation of the object as an *M*-by-6 matrix, where each row is of form `[x y z qW qX qY qZ]`, and *M* is the number of states along the path of the object in the world

frame. The list of states assumes each state is separated by a fixed time interval. *xyz*-positions are in meters, and the orientation is a four-element quaternion vector. The default local origin is located at the center of the left hemisphere of the capsule.

- `Geometry` -- Structure with fields `Length`, `Radius`, and `FixedTransform`. These fields define the size of the capsule-based object using the specified length for the cylinder and hemisphere radius for the end caps. To shift the capsule geometry from the default origin, specify the `FixedTransform` field as a fixed transform relative to the local frame of the capsule. To keep the default capsule origin, specify the transform as `eye(4)`.



`status` — Result of adding obstacles
N-element column vector

Result of adding obstacles, returned as a *N*-element column vector of ones, zeros, and negative ones. *N* is the number of obstacles specified in the `obstacleStruct` argument. Each value indicates whether the associated body is added (`1`), updated (`0`), or a duplicate (`-1`). While adding obstacles, if multiple structures with the same body ID are found in the structure array `obstaclesStruct`, then the function marks the previous entry as duplicate and ignores it.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

Objects

`dynamicCapsuleList` | `dynamicCapsuleList3D`

Functions
`addEgo` | `checkCollision` | `egoGeometry` | `egoPose` | `obstacleGeometry` | `obstaclePose` | `removeEgo` | `removeObstacle` | `show` | `updateEgoGeometry` | `updateEgoPose` | `updateObstacleGeometry` | `updateObstaclePose`

Topics
“Highway Trajectory Planning Using Frenet Reference Path”

Introduced in R2020b

ekfSLAM

Perform simultaneous localization and mapping using extended Kalman filter

Description

The `ekfSLAM` object performs simultaneous localization and mapping (SLAM) using an extended Kalman filter (EKF). It takes in observed landmarks from the environment and compares them with known landmarks to find associations and new landmarks. Use the associations to correct the state and state covariance. The new landmarks are augmented in the state vector.

Creation

Syntax

```
slamObj = ekfSLAM
slamObj = ekfSLAM(Name,Value)
slamObj = ekfSLAM('MaxNumLandmark',N,Name,Value)
slamObj = ekfSLAM('MaxNumLandmark',N,‘MaxNumPoseStored’,M,Name,Value)
```

Description

`slamObj = ekfSLAM` creates an EKF SLAM object with default properties.

`slamObj = ekfSLAM(Name,Value)` sets properties using one or more name-value pair arguments in addition to any combination of input arguments from previous syntaxes. Any unspecified properties have default values.

`slamObj = ekfSLAM('MaxNumLandmark',N,Name,Value)` specifies an upper bound on the number of landmarks `N` allowed in the state vector when generating code. This limit on the number of landmarks applies only when generating code.

`slamObj = ekfSLAM('MaxNumLandmark',N,‘MaxNumPoseStored’,M,Name,Value)` specifies the maximum size of the pose history `M` along with the maximum number of landmarks `N` in the state vector while generating code. These limits apply only when generating code.

Properties

You cannot change the value of the properties State, StateCovariance, StateTransitionFcn, and MaxNumLandmark after the object is created. Set the value of these properties as a default or while creating the object.

`State` — State vector

`[0; 0; 0]` (default) | *M*-element column vector

State vector, specified as an *M*-element column vector.

Data Types: `single` | `double`

`StateCovariance` — State estimation error covariance
`eye(3)` (default) | *M*-by-*M* matrix

State estimation error covariance, specified as an *M*-by-*M* matrix. *M* is the number of states in the state vector.

Data Types: `single` | `double`

`StateTransitionFcn` — State transition function
`nav.algs.velocityMotionModel` (default) | function handle

State transition function, specified as a function handle. This function calculates the state vector at time step `k` from the state vector at time step `k-1`. The function can take additional input parameters, such as control inputs or time step size.

The function also calculates the Jacobians with respect to the current pose and controller input. If not specified, the Jacobians are computed using numerical differencing at each call to the `predict` function. This computation can increase processing time and numerical inaccuracy.

The function considers nonadditive process noise, and should have this signature:

```
[pose(k),jacPose,jacControl] =
StateTransitionFcn(pose(k-1),controlInput,parameters)
```

- `pose(k)` is the estimated pose at time `k`.
- `jacPose` is the Jacobian of `StateTransitionFcn` with respect to `pose(k-1)`.
- `jacControl` is the Jacobian of `StateTransitionFcn` with respect to `controlInput`.
- `controlInput` is the input for propagating the state.
- `parameters` are any additional arguments required by the state transition function.

Data Types: `function_handle`

`MeasurementFcn` — Measurement function
`nav.algs.rangeBearingMeasurement` (default) | function handle

Measurement function, specified as a function handle. This function calculates an *N*-element measurement vector for an *M*-element state vector.

The function also calculates the Jacobians with respect to the current pose and landmark position. If not specified, the Jacobians are computed using numerical differencing at each call to the `correct` function. This computation can increase processing time and numerical inaccuracy.

The function considers additive measurement noise, and should have this signature:

```
[measurements(k),jacPose,jacLandmarks] = MeasurementFcn(pose(k),landmarks)
```

- `pose(k)` is the estimated pose at time `k`.
- `measurements(k)` is the estimated measurement at time `k`.
- `landmarks` are the positions of the landmarks.
- `jacPose` is the Jacobian of `MeasurementFcn` with respect to `pose(k)`.
- `jacLandmarks` is the Jacobian of `MeasurementFcn` with respect to `landmarks`.

Data Types: `function_handle`

`InverseMeasurementFcn` — Inverse measurement function
`nav.algs.rangeBearingInverseMeasurement` (default) | function handle

Inverse measurement function, specified as a function handle. This function calculates the landmark position as an *M*-element state vector for an *N*-element measurement vector.

The function also calculates the Jacobians with respect to the current pose and measurement. If not specified, the Jacobians are computed using numerical differencing at each call to the `correct` function. This computation can increase processing time and numerical inaccuracy.

The function should have this signature:

```
[landmarks(k),jacPose,jacMeasurements] =
InverseMeasurementFcn(pose(k),measurements)
```

- `pose(k)` is the estimated pose at time `k`.
- `landmarks(k)` is the landmark position at time `k`.
- `measurements` are the observed landmarks at time `k`.
- `jacPose` is the Jacobian of `InverseMeasurementFcn` with respect to `pose(k)`.
- `jacMeasurements` is the Jacobian of `InverseMeasurementFcn` with respect to `measurements`.

Data Types: `function_handle`

`DataAssociationFcn` — Data association function
`nav.algs.associateMaxLikelihood` (default) | function handle

Data association function, specified as a function handle. This function associates the measurements with the landmarks already available in the state vector. The function may take additional input parameters.

The function should have this signature:

```
[associations,newLandmarks] =
DataAssociationFcn(knownLandmarks,knownLandmarksCovariance,observedLandmarks,
observedLandmarksCovariance,parameters)
```

- `knownLandmarks` are known landmarks in the map.
- `knownLandmarksCovariance` is the covariance of `knownLandmarks`.
- `observedLandmarks` are the observed landmarks in the environment.
- `observedLandmarksCovariance` is the covariance of `observedLandmarks`.
- `parameters` are any additional arguments required.
- `associations` is a list of associations from `knownLandmarks` to `observedLandmarks`.
- `newLandmarks` are the indices of `observedLandmarks` that qualify as new landmarks.

Data Types: `function_handle`

`ProcessNoise` — Process noise covariance
`eye(2)` (default) | *W*-by-*W* matrix

Process noise covariance, specified as a *W*-by-*W* matrix. *W* is the number of process noise terms.

Data Types: `single` | `double`

`MaxAssociationRange` — Maximum range for landmarks to be checked for association
`inf` (default) | positive integer

Maximum range for the landmarks to be checked for association, specified as a positive integer.

Data Types: `single` | `double`

`MaxNumLandmark` — Maximum number of landmarks in state vector
`inf` (default) | positive integer

Maximum number of landmarks in the state vector, specified as a positive integer.

Data Types: `single` | `double`

`MaxNumPoseStored` — Maximum size of pose history
`inf` (default) | positive integer

Maximum size of pose history, specified as a positive integer.

Data Types: `single` | `double`

Object Functions

copy	Create deep copy of EKF SLAM object
correct	Correct state and state error covariance
landmarkInfo	Retrieve landmark information
poseHistory	Retrieve corrected and predicted pose history
predict	Predict state and state error covariance
removeLandmark	Remove landmark from state vector
reset	Reset state and state estimation error covariance

Examples

Perform Landmark SLAM Using Extended Kalman Filter

Load a race track data set that contains the initial vehicle state, initial vehicle state covariance, process noise covariance, control input, time step size, measurement, measurement covariance, and validation gate values.

```
load("racetrackDataset.mat","initialState","initialStateCovariance", ...
     "processNoise","controllerInputs","timeStep", ...
     "measurements","measCovar","validationGate");
```

Create an `ekfSLAM` object with initial state, initial state covariance, and process noise.

```
ekfSlamObj = ekfSLAM("State",initialState, ...
                     "StateCovariance",initialStateCovariance, ...
                     "ProcessNoise",processNoise);
```

Initialize a variable to store the pose.

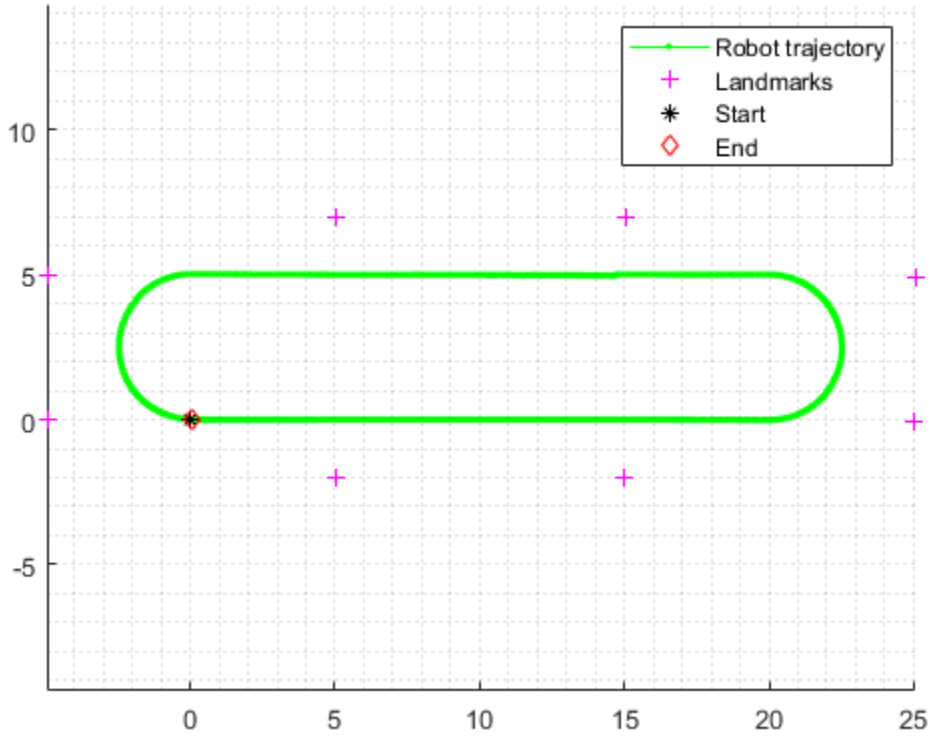
```
storedPose = nan(size(controllerInputs,1)+1,3);
storedPose(1,:) = ekfSlamObj.State(1:3);
```

Predict the state using the control input and time step size for the state transition function. Then, correct the state using the data of the observed landmarks, measurement covariance, and validation gate for the data association function.

```
for count = 1:size(controllerInputs,1)
    % Predict the state
    predict(ekfSlamObj,controllerInputs(count,:),timeStep);

    % Get the landmarks in the environment
    observedLandmarks = measurements{count};

    % Correct the state
    if ~isempty(observedLandmarks)
        correct(ekfSlamObj,observedLandmarks, ...
                measCovar,validationGate);
    end

    % Log the estimated pose
    storedPose(count+1,:) = ekfSlamObj.State(1:3);
end
```

Visualize the created map.

```
fig = figure;
figAx = axes(fig);
axis equal
grid minor
hold on
plot(figAx,storedPose(:,1),storedPose(:,2),"g.-")
landmarks = reshape(ekfSlamObj.State(4:end),2,[])';
plot(figAx,landmarks(:,1),landmarks(:,2),"m+")
plot(figAx,storedPose(1,1),storedPose(1,2),"k*")
plot(figAx,storedPose(end,1),storedPose(end,2),"rd")
legend("Robot trajectory","Landmarks","Start","End")
```



Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

Objects

`lidarSLAM`

Functions

`copy` | `correct` | `landmarkInfo` | `poseHistory` | `predict` | `removeLandmark` | `reset`

Topics

“EKF-Based Landmark SLAM”

Introduced in R2021b

copy

Create deep copy of EKF SLAM object

Syntax

```
newSlamObj = copy(slamObj)
```

Description

`newSlamObj = copy(slamObj)` creates a deep copy of `slamObj` with the same properties. Any changes made to `newSlamObj` are not reflected in `slamObj`.

Examples

Remove Landmark from ekfSLAM Object

Specify the initial vehicle state.

```
initialState = [1; -2; 0.1];
```

Specify the initial landmark positions.

```
landmarkPosition = [15.8495; -12.9496;
                    25.2455; -15.4705;
                    37.5880;   3.1023;
                    16.5690;   2.7466];
```

Specify the initial vehicle state covariance.

```
initialStateCovar = diag([0.1*ones(1,3) 1.1*ones(1,8)]);
```

Create an `ekfSLAM` object with initial state and initial state covariance.

```
ekfSlamObj = ekfSLAM('State',[initialState; landmarkPosition], ...
                     'StateCovariance',initialStateCovar);
landmarkInfo(ekfSlamObj)
```

```
ans=4×3 table
    landmark number    landmark state index    landmark position
    _______________    ____________________    _________________

           1                   4     5           15.85    -12.95
           2                   6     7          25.245    -15.47
           3                   8     9          37.588    3.1023
           4                  10    11          16.569    2.7466
```

Create a deep copy of the `ekfSLAM` object.

```
newEkfSlamObj = copy(ekfSlamObj);
```

Specify the landmark number to be removed.

```
removeLandmark(newEkfSlamObj,3);
landmarkInfo(newEkfSlamObj)
```

```
ans=3×3 table
    landmark number    landmark state index    landmark position
    _______________    ____________________    _________________

          1                  4    5             15.85    -12.95
          2                  6    7            25.245    -15.47
          3                  8    9            16.569    2.7466
```

Input Arguments

`slamObj` — EKF SLAM object

`ekfSLAM` object

EKF SLAM object, specified as an `ekfSLAM` object.

Output Arguments

`newSlamObj` — Copy of EKF SLAM object

`ekfSLAM` object

Copy of the EKF SLAM object, returned as an `ekfSLAM` object.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

Objects

`ekfSLAM` | `lidarSLAM`

Functions

`correct` | `landmarkInfo` | `poseHistory` | `predict` | `removeLandmark` | `reset`

Introduced in R2021b

correct

Correct state and state error covariance

Syntax

```
[associations,newLandmark] = correct(slamObj,measurement,
measurementCovariance)
[associations,newLandmark] = correct(slamObj,measurement,
measurementCovariance,varargin)
```

Description

`[associations,newLandmark] = correct(slamObj,measurement, measurementCovariance)` corrects the state and its associated state covariance based on the `measurement` and `measurementCovariance` at the current time step. `correct` uses the data association function specified in the DataAssociationFcn property of the `ekfSLAM` object, `slamObj`, to associate the measurement to landmarks and extract new landmarks from the measurement.

The `correct` function uses these associations to correct the state and associated state covariance, then augments the state with new landmarks.

`[associations,newLandmark] = correct(slamObj,measurement, measurementCovariance,varargin)` passes all additional arguments specified in `varargin` to the underlying DataAssociationFcn property of `slamObj`.

The first four inputs to the `DataAssociationFcn` property are the landmark position, landmark position covariance, measurement, and measurement covariance, followed by all arguments in `varargin`.

Examples

Perform Landmark SLAM Using Extended Kalman Filter

Load a race track data set that contains the initial vehicle state, initial vehicle state covariance, process noise covariance, control input, time step size, measurement, measurement covariance, and validation gate values.

```
load("racetrackDataset.mat","initialState","initialStateCovariance", ...
     "processNoise","controllerInputs","timeStep", ...
     "measurements","measCovar","validationGate");
```

Create an `ekfSLAM` object with initial state, initial state covariance, and process noise.

```
ekfSlamObj = ekfSLAM("State",initialState, ...
                     "StateCovariance",initialStateCovariance, ...
                     "ProcessNoise",processNoise);
```

Initialize a variable to store the pose.

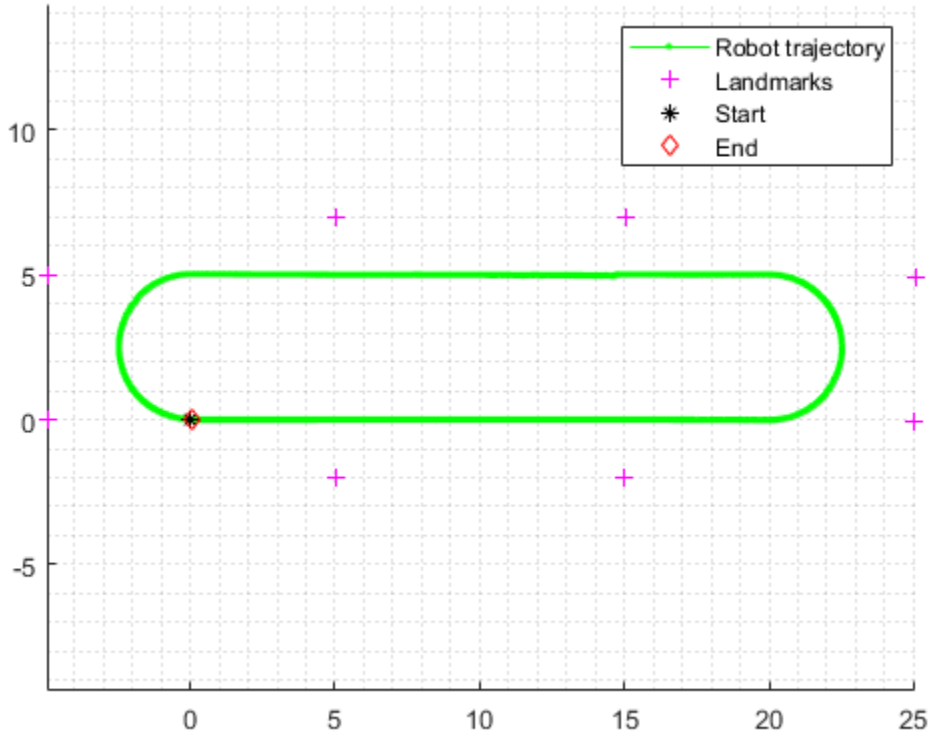
```
storedPose = nan(size(controllerInputs,1)+1,3);
storedPose(1,:) = ekfSlamObj.State(1:3);
```

Predict the state using the control input and time step size for the state transition function. Then, correct the state using the data of the observed landmarks, measurement covariance, and validation gate for the data association function.

```
for count = 1:size(controllerInputs,1)
    % Predict the state
    predict(ekfSlamObj,controllerInputs(count,:),timeStep);

    % Get the landmarks in the environment
    observedLandmarks = measurements{count};

    % Correct the state
    if ~isempty(observedLandmarks)
        correct(ekfSlamObj,observedLandmarks, ...
                measCovar,validationGate);
    end

    % Log the estimated pose
    storedPose(count+1,:) = ekfSlamObj.State(1:3);
end
```

Visualize the created map.

```
fig = figure;
figAx = axes(fig);
axis equal
grid minor
hold on
plot(figAx,storedPose(:,1),storedPose(:,2),"g.-")
landmarks = reshape(ekfSlamObj.State(4:end),2,[])';
plot(figAx,landmarks(:,1),landmarks(:,2),"m+")
plot(figAx,storedPose(1,1),storedPose(1,2),"k*")
plot(figAx,storedPose(end,1),storedPose(end,2),"rd")
legend("Robot trajectory","Landmarks","Start","End")
```



Input Arguments

`slamObj` — EKF SLAM object
`ekfSLAM` object

EKF SLAM object, specified as an `ekfSLAM` object.

`measurement` — Measurements of landmarks in environment
N-by-*K* matrix

Measurements of the landmarks in the environment, specified as an *N*-by-*K* matrix. *K* is the dimension of the measurement. *N* is the number of measurements.

Data Types: `single` | `double`

`measurementCovariance` — Covariance of measurements
K-element vector | *N***K*-by-*N***K* matrix

Covariance of the measurements, specified as a *K*-element vector or *N***K*-by-*N***K* matrix. *K* is the dimension of the measurement. *N* is the number of measurements. When specified as a vector, the same covariance value is used for all measurements.

Data Types: `single` | `double`

`varargin` — Variable-length input argument list
comma-separated list

Variable-length input argument list, specified as a comma-separated list. This input is passed directly into the DataAssociationFcn property of `slamObj`. When you call:

```
correct(slamObj,measurement,measurementCovariance,arg1,arg2)
```

MATLAB essentially calls the `dataAssociationFcn` as:

```
dataAssociationFcn(knownLandmarks,knownLandmarksCovariance, ...
measurement,measurementCovariance,arg1,arg2)
```

Output Arguments

`associations` — List of associations of landmarks to measurements

P-by-2 matrix

List of associations of landmarks to measurements, returned as a *P*-by-2 matrix. *P* is the number of associations. The first column of the matrix contains the indices of the associated landmarks, and the second column contains the associated measurement indices.

`newLandmark` — List of indices of measurements that qualify as new landmarks

Q-element vector

List of indices of the measurements that qualify as new landmarks, returned as a *Q*-element vector. *Q* is the number of measurements that qualify as new landmarks.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

Objects

`ekfSLAM`

Functions

`landmarkInfo` | `poseHistory` | `predict` | `removeLandmark` | `reset`

Introduced in R2021b

landmarkInfo

Retrieve landmark information

Syntax

```
Info = landmarkInfo(slamObj)
Info = landmarkInfo(slamObj,landmarkIndex)
```

Description

`Info = landmarkInfo(slamObj)` retrieves landmark information from the `ekfSLAM` object as a table that contains each landmark number along with its position and state index.

`Info = landmarkInfo(slamObj,landmarkIndex)` retrieves landmark information for only those landmarks specified by `landmarkIndex`.

Examples

Retrieve All Landmark Information from ekfSLAM Object

Specify the initial vehicle state.

```
initialState = [1; -2; 0.1];
```

Specify the initial landmark positions.

```
landmarkPosition = [15.8495; -12.9496;
                    25.2455; -15.4705;
                    37.5880;   3.1023;
                    16.5690;   2.7466];
```

Specify the initial vehicle state covariance.

```
initialStateCovar = diag([0.1*ones(1,3) 1.1*ones(1,8)]);
```

Create an `ekfSLAM` object with initial state and initial state covariance.

```
ekfSlamObj = ekfSLAM('State',[initialState; landmarkPosition], ...
                     'StateCovariance',initialStateCovar)
```

```
ekfSlamObj =
  ekfSLAM with properties:

                State: [11x1 double]
       StateCovariance: [11x11 double]
        MaxNumLandmark: Inf
    StateTransitionFcn: @nav.algs.velocityMotionModel
          ProcessNoise: [2x2 double]

           MeasurementFcn: @nav.algs.rangeBearingMeasurement
    InverseMeasurementFcn: @nav.algs.rangeBearingInverseMeasurement
```

```
     DataAssociationFcn: @nav.algs.associateMaxLikelihood
    MaxAssociationRange: Inf
```

Get the information for all the landmarks.

```
info = landmarkInfo(ekfSlamObj)
```

```
info=4×3 table
    landmark number    landmark state index    landmark position
    _______________    ____________________    _________________

          1                   4     5            15.85    -12.95
          2                   6     7           25.245    -15.47
          3                   8     9           37.588    3.1023
          4                  10    11           16.569    2.7466
```

Input Arguments

`slamObj` — EKF SLAM object

`ekfSLAM` object

EKF SLAM object, specified as an `ekfSLAM` object.

`landmarkIndex` — Indices of landmarks for which to retrieve information

N-element column vector | *N*-by-2 matrix

Indices of landmarks for which to retrieve information, specified as an *N*-element column vector of landmark numbers in the state vector or an *N*-by-2 matrix of exact positions of landmarks in the state vector. *N* is the number of landmarks.

Data Types: `single` | `double`

Output Arguments

`Info` — Landmark information

table

Landmark information, returned as a table.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

Objects

`ekfSLAM`

Functions

`correct` | `poseHistory` | `predict` | `removeLandmark` | `reset`

Introduced in R2021b

poseHistory

Retrieve corrected and predicted pose history

Syntax

```
[correctedPose,predictedPose] = poseHistory(slamObj)
```

Description

`[correctedPose,predictedPose] = poseHistory(slamObj)` retrieves the corrected and predicted pose history up to the current pose.

Note To use this function during code generation, you must specify the `MaxNumPoseStored` property of the `ekfSLAM` object. Otherwise, this function returns an error.

Examples

Retrieve Pose History from ekfSLAM Object

Specify the initial vehicle state.

```
initialState = [1; -2; 0.1];
```

Specify the initial landmark positions.

```
landmarkPosition = [15.8495; -12.9496;
                    25.2455; -15.4705;
                    37.5880;   3.1023;
                    16.5690;   2.7466];
```

Specify the initial vehicle state covariance.

```
initialStateCovar = diag([0.1*ones(1,3) 1.1*ones(1,8)]);
```

Create an `ekfSLAM` object with initial state and initial state covariance.

```
ekfSlamObj = ekfSLAM('State',[initialState; landmarkPosition], ...
                     'StateCovariance',initialStateCovar);
```

Specify the control input and time step size for the state transition function.

```
velocity = [1 0];
timeStep = 0.25;
```

Call the `predict` function.

```
predict(ekfSlamObj,velocity,timeStep);
```

Specify the measurement and measurement covariance for the data association function.

```
measurement = [18.4500 -0.7354;
               27.7362 -0.6071;
               36.9421  0.0386;
               16.2765  0.1959];
measureCovar = [0.1^2 (1.0*pi/180)^2];
```

Call the `correct` function.

```
validationGate = 5.991;
associations = correct(ekfSlamObj,measurement, ...
                       measureCovar,validationGate);
```

Get the pose history.

```
[corrPose,predPose] = poseHistory(ekfSlamObj)
```

```
corrPose = 1×3

    1.1609   -1.9736    0.0981


predPose = 1×3

    1.2488   -1.9750    0.1000
```

Input Arguments

`slamObj` — EKF SLAM object

`ekfSLAM` object

EKF SLAM object, specified as an `ekfSLAM` object.

Output Arguments

`correctedPose` — Corrected poses

M-by-3 matrix

Corrected poses, returned as an *M*-by-3 matrix with rows of the form [*X Y Yaw*]. *X* and *Y* specify the position in meters. *Yaw* specifies the orientation in radians.

`predictedPose` — Predicted poses

M-by-3 matrix

Predicted poses, returned as an *M*-by-3 matrix with rows of the form [*X Y Yaw*]. *X* and *Y* specify the position in meters. *Yaw* specifies the orientation in radians.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

Objects
`ekfSLAM`

Functions
`correct` | `landmarkInfo` | `predict` | `removeLandmark` | `reset`

Introduced in R2021b

predict

Predict state and state error covariance

Syntax

```
predict(slamObj,controlInput)
predict(slamObj,controlInput,varargin)
```

Description

`predict(slamObj,controlInput)` predicts the state and state error covariance. `predict` uses the StateTransitionFcn property of the `ekfSLAM` object, `slamObj`, and the controller input `controlInput` to predict the state.

`predict(slamObj,controlInput,varargin)` passes all additional arguments specified in `varargin` to the underlying StateTransitionFcn property of `slamObj`.

The first input to `StateTransitionFcn` is the pose from the previous time step, followed by all user-defined arguments in `varargin`.

Examples

Perform Landmark SLAM Using Extended Kalman Filter

Load a race track data set that contains the initial vehicle state, initial vehicle state covariance, process noise covariance, control input, time step size, measurement, measurement covariance, and validation gate values.

```
load("racetrackDataset.mat","initialState","initialStateCovariance", ...
     "processNoise","controllerInputs","timeStep", ...
     "measurements","measCovar","validationGate");
```

Create an `ekfSLAM` object with initial state, initial state covariance, and process noise.

```
ekfSlamObj = ekfSLAM("State",initialState, ...
                     "StateCovariance",initialStateCovariance, ...
                     "ProcessNoise",processNoise);
```

Initialize a variable to store the pose.

```
storedPose = nan(size(controllerInputs,1)+1,3);
storedPose(1,:) = ekfSlamObj.State(1:3);
```

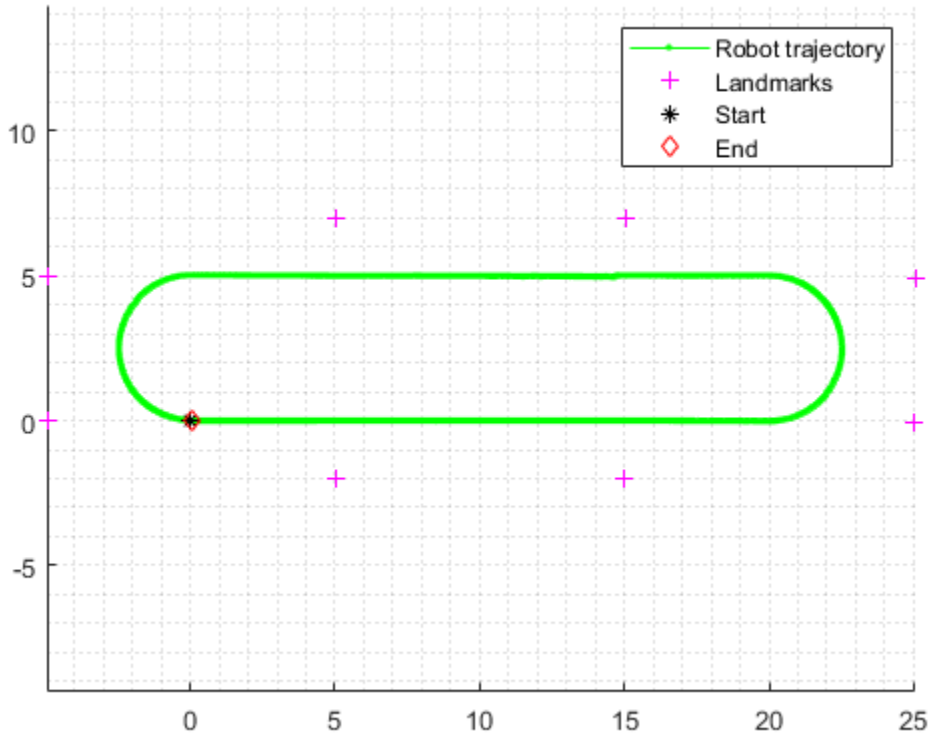
Predict the state using the control input and time step size for the state transition function. Then, correct the state using the data of the observed landmarks, measurement covariance, and validation gate for the data association function.

```
for count = 1:size(controllerInputs,1)
    % Predict the state
    predict(ekfSlamObj,controllerInputs(count,:),timeStep);
```

```
    % Get the landmarks in the environment
    observedLandmarks = measurements{count};

    % Correct the state
    if ~isempty(observedLandmarks)
        correct(ekfSlamObj,observedLandmarks, ...
                measCovar,validationGate);
    end

    % Log the estimated pose
    storedPose(count+1,:) = ekfSlamObj.State(1:3);
end
```

Visualize the created map.

```
fig = figure;
figAx = axes(fig);
axis equal
grid minor
hold on
plot(figAx,storedPose(:,1),storedPose(:,2),"g.-")
landmarks = reshape(ekfSlamObj.State(4:end),2,[])';
plot(figAx,landmarks(:,1),landmarks(:,2),"m+")
plot(figAx,storedPose(1,1),storedPose(1,2),"k*")
plot(figAx,storedPose(end,1),storedPose(end,2),"rd")
legend("Robot trajectory","Landmarks","Start","End")
```



Input Arguments

`slamObj` — EKF SLAM object
`ekfSLAM` object

EKF SLAM object, specified as a `ekfSLAM` object.

`controlInput` — Controller input required to propagate state
N-element vector

Controller input required to propagate the state from initial value to final value, specified as an *N*-element vector.

Note The dimension of the process noise must be equal to the number of elements in `controlInput`.

Data Types: `single` | `double`

`varargin` — Variable-length input argument list
comma-separated list

Variable-length input argument list, specified as a comma-separated list. This input is passed directly into the StateTransitionFcn property of `slamObj` to evolve the state. When you call:

```
predict(slamObj,controlInput,arg1,arg2)
```

MATLAB essentially calls the `stateTransitionFcn` as:

```
stateTransitionFcn(pose(k-1),controlInput,arg1,arg2)
```

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

Objects
`ekfSLAM`

Functions
`correct` | `landmarkInfo` | `poseHistory` | `removeLandmark` | `reset`

Introduced in R2021b

removeLandmark

Remove landmark from state vector

Syntax

```
removeLandmark(slamObj,landmarkIndex)
```

Description

`removeLandmark(slamObj,landmarkIndex)` removes the landmarks at the specified indices `landmarkIndex` from the state vector, along with associated covariances from the state covariance matrix.

Examples

Remove Landmark from ekfSLAM Object

Specify the initial vehicle state.

```
initialState = [1; -2; 0.1];
```

Specify the initial landmark positions.

```
landmarkPosition = [15.8495; -12.9496;
                    25.2455; -15.4705;
                    37.5880;   3.1023;
                    16.5690;   2.7466];
```

Specify the initial vehicle state covariance.

```
initialStateCovar = diag([0.1*ones(1,3) 1.1*ones(1,8)]);
```

Create an `ekfSLAM` object with initial state and initial state covariance.

```
ekfSlamObj = ekfSLAM('State',[initialState; landmarkPosition], ...
                     'StateCovariance',initialStateCovar);
landmarkInfo(ekfSlamObj)
```

```
ans=4×3 table
    landmark number    landmark state index    landmark position
    _______________    ____________________    _________________

           1                   4     5           15.85    -12.95
           2                   6     7          25.245    -15.47
           3                   8     9          37.588    3.1023
           4                  10    11          16.569    2.7466
```

Create a deep copy of the `ekfSLAM` object.

```
newEkfSlamObj = copy(ekfSlamObj);
```

Specify the landmark number to be removed.

```
removeLandmark(newEkfSlamObj,3);
landmarkInfo(newEkfSlamObj)
```

```
ans=3×3 table
    landmark number    landmark state index    landmark position
    _______________    ____________________    _________________

          1                  4    5             15.85    -12.95
          2                  6    7            25.245    -15.47
          3                  8    9            16.569    2.7466
```

Input Arguments

`slamObj` — EKF SLAM object

`ekfSLAM` object

EKF SLAM object, specified as an `ekfSLAM` object.

`landmarkIndex` — Indices of landmarks to remove

N-element column vector | *N*-by-2 matrix

Indices of the landmarks to remove, specified as an *N*-element column vector of landmark numbers in the state vector or an *N*-by-2 matrix of the exact positions of landmarks in the state vector. *N* is the number of landmarks to remove.

Data Types: `single` | `double`

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

Objects

`ekfSLAM`

Functions

`correct` | `landmarkInfo` | `poseHistory` | `predict` | `reset`

Introduced in R2021b

reset

Reset state and state estimation error covariance

Syntax

```
reset(slamObj)
```

Description

`reset(slamObj)` resets the state and state estimation error covariance to their default values, and resets the internal states.

Examples

Reset State and State Estimation Error Covariance in ekfSLAM Object

Specify the initial vehicle state.

```
initialState = [1; -2; 0.1];
```

Specify the initial vehicle state covariance.

```
initialStateCovar = 0.1*eye(3);
```

Create an `ekfSLAM` object with initial state and initial state covariance.

```
ekfSlamObj = ekfSLAM('State',initialState, ...
                     'StateCovariance',initialStateCovar);
```

Get the state and the state covariance from the `ekfSLAM` object.

```
ekfSlamObj.State
```

```
ans = 3×1

    1.0000
   -2.0000
    0.1000
```

```
ekfSlamObj.StateCovariance
```

```
ans = 3×3

    0.1000         0         0
         0    0.1000         0
         0         0    0.1000
```

Reset the state and state estimation error covariance to the default value.

```
reset(ekfSlamObj)
```

Get the state and the state covariance from the `ekfSLAM` object.

```
ekfSlamObj.State
```

```
ans = 3×1

     0
     0
     0
```

```
ekfSlamObj.StateCovariance
```

```
ans = 3×3

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

Input Arguments

`slamObj` — EKF SLAM object

`ekfSLAM` object

EKF SLAM object, specified as an `ekfSLAM` object.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

Objects

`ekfSLAM`

Functions

`correct` | `landmarkInfo` | `poseHistory` | `predict` | `removeLandmark`

Introduced in R2021b

insfilterErrorState

Estimate pose from IMU, GPS, and monocular visual odometry (MVO) data

Description

The `insfilterErrorState` object implements sensor fusion of IMU, GPS, and monocular visual odometry (MVO) data to estimate pose in the NED (or ENU) reference frame. The filter uses a 17-element state vector to track the orientation `quaternion`, velocity, position, IMU sensor biases, and the MVO scaling factor. The `insfilterErrorState` object uses an error-state Kalman filter to estimate these quantities.

Creation

Syntax

```
filter = insfilterErrorState
filter = insfilterErrorState('ReferenceFrame',RF)
filter = insfilterErrorState(___,Name,Value)
```

Description

`filter = insfilterErrorState` creates an `insfilterErrorState` object with default property values.

`filter = insfilterErrorState('ReferenceFrame',RF)` allows you to specify the reference frame, `RF`, of the `filter`. Specify `RF` as `'NED'` (North-East-Down) or `'ENU'` (East-North-Up). The default value is `'NED'`.

`filter = insfilterErrorState(___,Name,Value)` also allows you set properties of the created `filter` using one or more name-value pairs. Enclose each property name in single quotes.

Properties

`IMUSampleRate` — Sample rate of IMU (Hz)
`100` (default) | positive scalar

Sample rate of the inertial measurement unit (IMU) in Hz, specified as a positive scalar.

Data Types: `single` | `double`

`ReferenceLocation` — Reference location (deg, deg, meters)
`[0 0 0]` (default) | 3-element positive row vector

Reference location, specified as a 3-element row vector in geodetic coordinates (latitude, longitude, and altitude). Altitude is the height above the reference ellipsoid model, WGS84. The reference location units are [degrees degrees meters].

Data Types: `single` | `double`

`GyroscopeNoise` — Multiplicative process noise variance from gyroscope ((rad/s)²)
`[1e-6 1e-6 1e-6]` (default) | scalar | 3-element row vector

Multiplicative process noise variance from the gyroscope in $(rad/s)^2$, specified as a scalar or 3-element row vector of positive real finite numbers.

- If `GyroscopeNoise` is specified as a row vector, the elements correspond to the noise in the *x*, *y*, and *z* axes of the gyroscope, respectively.
- If `GyroscopeNoise` is specified as a scalar, the single element is applied to each axis.

Data Types: `single` | `double`

`GyroscopeBiasNoise` — Additive process noise variance from gyroscope bias ((rad/s)²)
`[1e-9 1e-9 1e-9]` (default) | scalar | 3-element row vector

Additive process noise variance from the gyroscope bias in $(rad/s)^2$, specified as a scalar or 3-element row vector of positive real finite numbers.

- If `GyroscopeBiasNoise` is specified as a row vector, the elements correspond to the noise in the *x*, *y*, and *z* axes of the gyroscope, respectively.
- If `GyroscopeBiasNoise` is specified as a scalar, the single element is applied to each axis

Data Types: `single` | `double`

`AccelerometerNoise` — Multiplicative process noise variance from accelerometer ((m/s²)²)
`[1e-4 1e-4 1e-4]` (default) | scalar | 3-element row vector

Multiplicative process noise variance from the accelerometer in $(m/s^2)^2$, specified as a scalar or 3-element row vector of positive real finite numbers.

- If `AccelerometerNoise` is specified as a row vector, the elements correspond to the noise in the *x*, *y*, and *z* axes of the accelerometer, respectively.
- If `AccelerometerNoise` is specified as a scalar, the single element is applied to each axis.

Data Types: `single` | `double`

`AccelerometerBiasNoise` — Additive process noise variance from accelerometer bias ((m/s²)²)
`[1e-4 1e-4 1e-4]` (default) | scalar | 3-element row vector

Additive process noise variance from accelerometer bias in $(m/s^2)^2$, specified as a scalar or 3-element row vector of positive real numbers.

- If `AccelerometerBiasNoise` is specified as a row vector, the elements correspond to the noise in the *x*, *y*, and *z* axes of the accelerometer, respectively.
- If `AccelerometerBiasNoise` is specified as a scalar, the single element is applied to each axis.

`State` — State vector of Kalman filter
`[1;zeros(15,1);1]` (default) | 17-element column vector

State vector of the extended Kalman filter, specified as a 17-element column vector. The state values represent:

State	Units	Index
Orientation (quaternion parts)	N/A	1:4
Position (NED or ENU)	m	5:7
Velocity (NED or ENU)	m/s	8:10
Gyroscope Bias (XYZ)	rad/s	11:13
Accelerometer Bias (XYZ)	m/s^2	14:16
Visual Odometry Scale (XYZ)	N/A	17

The default initial state corresponds to an object at rest located at `[0 0 0]` in geodetic LLA coordinates.

Data Types: `single` | `double`

`StateCovariance` — State error covariance for Kalman filter
`ones(16)` (default) | 16-by-16 matrix

State error covariance for the Kalman filter, specified as a 16-by-16-element matrix of real numbers. The state error covariance values represent:

State Covariance	Row/Column Index
δ Rotation Vector (XYZ)	1:3
δ Position (NED or ENU)	4:6
δ Velocity (NED or ENU)	7:9
δ Gyroscope Bias (XYZ)	10:12
δ Accelerometer Bias (XYZ)	13:15
δ Visual Odometry Scale (XYZ)	16

Note that because this is an error-state Kalman filter, it tracks the errors in the states. δ represents the error in the corresponding state.

Data Types: `single` | `double`

Object Functions

predict	Update states using accelerometer and gyroscope data for insfilterErrorState
correct	Correct states using direct state measurements for insfilterErrorState
residual	Residuals and residual covariances from direct state measurements for insfilterErrorState
fusegps	Correct states using GPS data for insfilterErrorState
residualgps	Residuals and residual covariance from GPS measurements for insfilterErrorState
fusemvo	Correct states using monocular visual odometry for insfilterErrorState
residualmvo	Residuals and residual covariance from monocular visual odometry measurements for insfilterErrorState
pose	Current orientation and position estimate for insfilterErrorState
reset	Reset internal states for insfilterErrorState
stateinfo	Display state vector information for insfilterErrorState
tune	Tune insfilterErrorState parameters to reduce estimation error
copy	Create copy of insfilterErrorState

Examples

Estimate Pose of Ground Vehicle

Load logged data of a ground vehicle following a circular trajectory. The `.mat` file contains IMU and GPS sensor measurements and ground truth orientation and position.

```
load('loggedGroundVehicleCircle.mat', ...
    'imuFs','localOrigin', ...
    'initialStateCovariance', ...
    'accelData','gyroData', ...
    'gpsFs','gpsLLA','Rpos','gpsVel','Rvel', ...
    'trueOrient','truePos');
```

Create an INS filter to fuse IMU and GPS data using an error-state Kalman filter.

```
initialState = [compact(trueOrient(1)),truePos(1,:),-6.8e-3,2.5002,0,zeros(1,6),1].';
filt = insfilterErrorState;
filt.IMUSampleRate = imuFs;
filt.ReferenceLocation = localOrigin;
filt.State = initialState;
filt.StateCovariance = initialStateCovariance;
```

Preallocate variables for position and orientation. Allocate a variable for indexing into the GPS data.

```
numIMUSamples = size(accelData,1);
estOrient = ones(numIMUSamples,1,'quaternion');
estPos = zeros(numIMUSamples,3);

gpsIdx = 1;
```

Fuse accelerometer, gyroscope, and GPS data. The outer loop predicts the filter forward at the fastest sample rate (the IMU sample rate).

```
for idx = 1:numIMUSamples

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

    % GPS data is collected at a lower sample rate than IMU data. Fuse GPS
    % data at the lower rate.
    if mod(idx, imuFs / gpsFs) == 0
        % Correct the filter states based on the GPS data.
        fusegps(filt,gpsLLA(gpsIdx,:),Rpos,gpsVel(gpsIdx,:),Rvel);
        gpsIdx = gpsIdx + 1;
    end

    % Log the current pose estimate
    [estPos(idx,:), estOrient(idx,:)] = pose(filt);
end
```

Calculate the RMS errors between the known true position and orientation and the output from the error-state filter.

```
pErr = truePos - estPos;
qErr = rad2deg(dist(estOrient,trueOrient));
```

```
pRMS = sqrt(mean(pErr.^2));
qRMS = sqrt(mean(qErr.^2));

fprintf('Position RMS Error\n');
```

```
Position RMS Error
```

```
fprintf('\tX: %.2f, Y: %.2f, Z: %.2f (meters)\n\n',pRMS(1),pRMS(2),pRMS(3));
```

```
    X: 0.40, Y: 0.24, Z: 0.05 (meters)
```

```
fprintf('Quaternion Distance RMS Error\n');
```

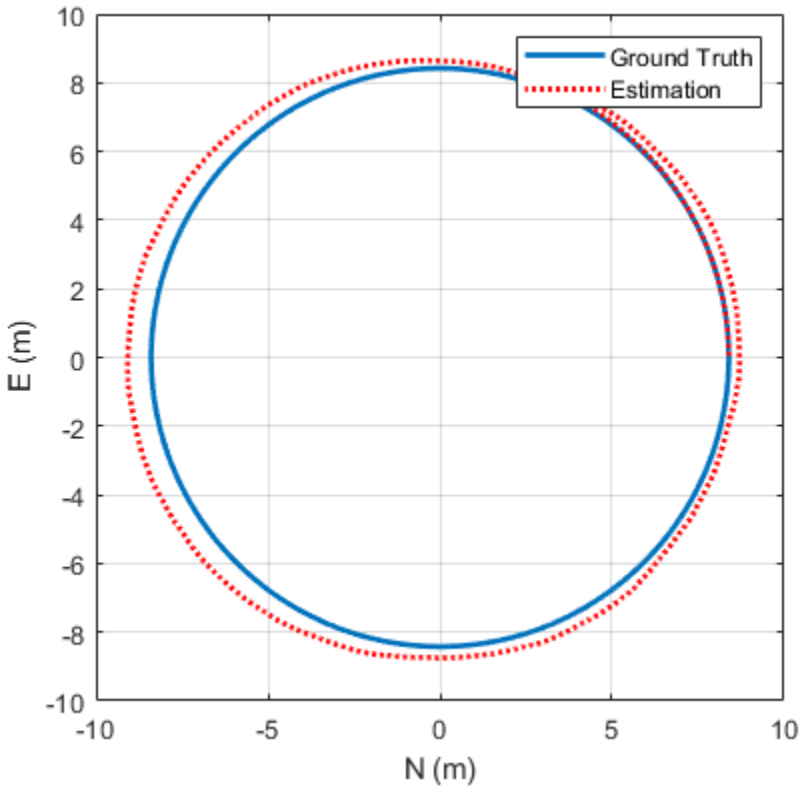
```
Quaternion Distance RMS Error
```

```
fprintf('\t%.2f (degrees)\n\n',qRMS);
```

```
    0.30 (degrees)
```

Visualize the true position and the estimated position.

```
plot(truePos(:,1),truePos(:,2),estPos(:,1),estPos(:,2),'r:','LineWidth',2)
grid on
axis square
xlabel('N (m)')
ylabel('E (m)')
legend('Ground Truth','Estimation')
```



Algorithms

Note: The following algorithm only applies to an NED reference frame.

`insfilterErrorState` uses a 17-axis error state Kalman filter structure to estimate pose in the NED reference frame. The state is defined as:

$$x = \begin{bmatrix} q_0 \\ q_1 \\ q_2 \\ q_3 \\ position_N \\ position_E \\ position_D \\ v_N \\ v_E \\ v_D \\ gyrobias_X \\ gyrobias_Y \\ gyrobias_Z \\ accelbias_X \\ accelbias_Y \\ accelbias_Z \\ scaleFactor \end{bmatrix}$$

where

- q_0, q_1, q_2, q_3 –– Parts of orientation quaternion. The orientation quaternion represents a frame rotation from the platform's current orientation to the local NED coordinate system.
- $position_N$, $position_E$, $position_D$ –– Position of the platform in the local NED coordinate system.
- $gyrobias_X$, $gyrobias_Y$, $gyrobias_Z$ –– Bias in the gyroscope reading.
- $accelbias_X$, $accelbias_Y$, $accelbias_Z$ –– Bias in the accelerometer reading.
- *scaleFactor* –– Scale factor of the pose estimate.

Given the conventional formulation of the state transition function,

$$x_{k|k-1} = f(\hat{x}_{k-1|k-1})$$

the predicted state estimate is:

$$x_{k|k-1} =$$

$$\begin{bmatrix}
q_0 + \Delta t * q_1(gyrobias_X/2 - gyro_X/2) + \Delta t * q_2 * (gyrobias_Y/2 - gyro_Y/2) + \Delta t * q_3 * (gyrobias_Z/2 - gyro_Z/2) \\
q_1 - \Delta t * q_0(gyrobias_X/2 - gyro_X/2) + \Delta t * q_3 * (gyrobias_Y/2 - gyro_Y/2) - \Delta t * q_2 * (gyrobias_Z/2 - gyro_Z/2) \\
q_2 - \Delta t * q_3(gyrobias_X/2 - gyro_X/2) - \Delta t * q_0 * (gyrobias_Y/2 - gyro_Y/2) + \Delta t * q_1 * (gyrobias_Z/2 - gyro_Z/2) \\
q_3 + \Delta t * q_2(gyrobias_X/2 - gyro_X/2) - \Delta t * q_1 * (gyrobias_Y/2 - gyro_Y/2) - \Delta t * q_0 * (gyrobias_Z/2 - gyro_Z/2) \\
position_N + \Delta t * v_N \\
position_E + \Delta t * v_E \\
position_D + \Delta t * v_D \\
v_N - \Delta t * \begin{Bmatrix}
q_0 * (q_0 * (accelbias_X - accel_X) - q_3 * (accelbias_Y - accel_Y) + q_2 * (accelbias_Z - accel_Z)) + g_N + \\
q_2 * (q_1 * (accelbias_Y - accel_Y) - q_2 * (accelbias_X - accel_X) + q_0 * (accelbias_Z - accel_Z)) + \\
q_1 * (q_1 * (accelbias_X - accel_X) + q_2 * (accelbias_Y - accel_Y) + q_3 * (accelbias_Z - accel_Z)) - \\
q_3 * (q_3 * (accelbias_X - accel_X) + q_0 * (accelbias_Y - accel_Y) - q_1 * (accelbias_Z - accel_Z))
\end{Bmatrix} \\
v_E - \Delta t * \begin{Bmatrix}
q_0 * (q_3 * (accelbias_X - accel_X) + q_0 * (accelbias_Y - accel_Y) - q_1 * (accelbias_Z - accel_Z)) + g_E - \\
q_1 * (q_1 * (accelbias_Y - accel_Y) - q_2 * (accelbias_X - accel_X) + q_0 * (accelbias_Z - accel_Z)) + \\
q_2 * (q_1 * (accelbias_X - accel_X) + q_2 * (accelbias_Y - accel_Y) + q_3 * (accelbias_Z - accel_Z)) + \\
q_3 * (q_0 * (accelbias_X - accel_X) - q_3 * (accelbias_Y - accel_Y) + q_2 * (accelbias_Z - accel_Z))
\end{Bmatrix} \\
\begin{Bmatrix}
q_0 * (q_1 * (accelbias_Y - accel_Y) - q_2 * (accelbias_X - accel_X) + q_0 * (accelbias_Z - accel_Z)) + g_D + \\
\end{Bmatrix}
\end{bmatrix}$$

where

- Δt –– IMU sample time.
- g_N, g_E, g_D –– Constant gravity vector in the NED frame.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`insfilterAsync` | `insfilterMARG` | `insfilterNonholonomic`

Introduced in R2019a

correct

Correct states using direct state measurements for `insfilterErrorState`

Syntax

```
correct(FUSE,idx,measurement,measurementCovariance)
```

Description

`correct(FUSE,idx,measurement,measurementCovariance)` corrects the state and state estimation error covariance of `FUSE`, an `insfilterErrorState` object, based on the measurement and measurement covariance. The measurement maps directly to the state specified by the indices `idx`.

Input Arguments

FUSE — INS filter object

`insfilterErrorState`

`insfilterErrorState`, specified as an object.

idx — State vector index of measurements to correct

N-element vector of increasing integers in the range [1, 17]

State vector index of measurements to correct, specified as an *N*-element vector of increasing integers in the range [1, 17].

The state values represent:

State	Units	Index
Orientation (quaternion parts)	N/A	1:4
Position (NED)	m	5:7
Velocity (NED)	m/s	8:10
Gyroscope Bias (XYZ)	rad/s	11:13
Accelerometer Bias (XYZ)	m/s^2	14:16
Visual Odometry Scale (XYZ)	N/A	17

Data Types: `single` | `double`

measurement — Direct measurement of state

N-element vector

Direct measurement of state, specified as a *N*-element vector. *N* is the number of elements of the index argument, `idx`.

Data Types: `single` | `double`

measurementCovariance — Covariance of measurement

scalar | *M*-element vector | *M*-by-*M* matrix

Covariance of measurement, specified as a scalar, *M*-element vector, or *M*-by-*M* matrix. If you correct orientation (state indices 1–4), then `M = numel(idx)-1`. If you do not correct orientation, then `M = numel(idx)`.

Data Types: `single` | `double`

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`insfilterErrorState` | `insfilter`

Introduced in R2019a

copy

Create copy of `insfilterErrorState`

Syntax

```
newFilter = copy(filter)
```

Description

`newFilter = copy(filter)` returns a copy of the `insfilterErrorState`, `filter`, with the exactly same property values.

Input Arguments

`filter` — Filter to be copied

`insfilterErrorState`

Filter to be copied, specified as an `insfilterErrorState` object.

Output Arguments

`newFilter` — New copied filter

`insfilterErrorState`

New copied filter, returned as an `insfilterErrorState` object.

See Also

`insfilterErrorState`

Introduced in R2020b

fusegps

Correct states using GPS data for `insfilterErrorState`

Syntax

```
[res,resCov] = fusegps(FUSE,position,positionCovariance)
[res,resCov] = fusegps(FUSE,position,positionCovariance,velocity,
velocityCovariance)
```

Description

`[res,resCov] = fusegps(FUSE,position,positionCovariance)` fuses GPS position data to correct the state estimate.

`[res,resCov] = fusegps(FUSE,position,positionCovariance,velocity, velocityCovariance)` fuses GPS position and velocity data to correct the state estimate.

Input Arguments

FUSE — INS filter object

`insfilterErrorState`

`insfilterErrorState`, specified as an object.

position — Position of GPS receiver (LLA)

3-element row vector

Position of GPS receiver in geodetic latitude, longitude, and altitude (LLA) specified as a real finite 3-element row vector. Latitude and longitude are in degrees with north and east being positive. Altitude is in meters.

Data Types: `single` | `double`

positionCovariance — Position measurement covariance of GPS receiver (m^2)

scalar | 3-element row vector | 3-by-3 matrix

Position measurement covariance of GPS receiver in m^2, specified as a 3-by-3 matrix.

Data Types: `single` | `double`

velocity — Velocity of GPS receiver in local NED coordinate system (m/s)

3-element row vector

Velocity of the GPS receiver in the local NED coordinate system in m/s, specified as a 3-element row vector.

Data Types: `single` | `double`

velocityCovariance — Velocity measurement covariance of GPS receiver $(m/s)^2$

scalar | 3-element row vector | 3-by-3 matrix

Velocity measurement covariance of the GPS receiver in the local NED coordinate system in $(m/s)^2$, specified as a 3-by-3 matrix.

Data Types: `single` | `double`

Output Arguments

`res` — Position and velocity residual

1-by-6 vector of real values

Position and velocity residual, returned as a 1-by-6 vector of real values in m and m/s, respectively.

`resCov` — Innovation residual

6-by-6 matrix of real values

Innovation residual, returned as a 6-by-6 matrix of real values.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`insfilterErrorState` | `insfilter`

Introduced in R2019a

fusemvo

Correct states using monocular visual odometry for `insfilterErrorState`

Syntax

```
[pResidual,oResidual,resCov] = fusemvo(FUSE,position,positionCovariance,ornt,
orntCovariance)
```

Description

`[pResidual,oResidual,resCov] = fusemvo(FUSE,position,positionCovariance,ornt,
orntCovariance)` fuses position and orientation data from monocular visual odometry (MVO) measurements to correct the state and state estimation error covariance.

Input Arguments

FUSE — INS filter object

`insfilterErrorState`

`insfilterErrorState`, specified as an object.

`position` — Position of camera in local NED coordinate system (m)

3-element row vector

Position of camera in the local NED coordinate system in meters, specified as a real finite 3-element row vector.

Data Types: `single` | `double`

`positionCovariance` — Position measurement covariance of MVO (m^2)

scalar | 3-element vector | 3-by-3 matrix

Position measurement covariance of MVO in m^2, specified as a scalar, 3-element vector, or 3-by-3 matrix.

Data Types: `single` | `double`

`ornt` — Orientation of camera with respect to local NED coordinate system

scalar quaternion | rotation matrix

Orientation of the camera with respect to the local NED coordinate system, specified as a scalar quaternion or 3-by-3 rotation matrix. The quaternion or rotation matrix is a frame rotation from the NED coordinate system to the current camera coordinate system.

Data Types: `quaternion` | `single` | `double`

`orntCovariance` — Orientation measurement covariance of monocular visual odometry (rad^2)

scalar | 3-element vector | 3-by-3 matrix

Orientation measurement covariance of monocular visual odometry in rad^2, specified as a scalar, 3-element vector, or 3-by-3 matrix.

Data Types: `single` | `double`

Output Arguments

`pResidual` — Position residual
1-by-3 vector of real values

Position residual, returned as a 1-by-3 vector of real values in m.

`oResidual` — Rotation vector residual
1-by-3 vector of real values

Rotation vector residual, returned as a 1-by-3 vector of real values in radians.

`resCov` — Residual covariance
6-by-6 matrix of real values

Residual covariance, returned as a 6-by-6 matrix of real values.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also
`insfilterErrorState` | `insfilter`

Introduced in R2019a

predict

Update states using accelerometer and gyroscope data for `insfilterErrorState`

Syntax

```
predict(FUSE,accelReadings,gyroReadings)
```

Description

`predict(FUSE,accelReadings,gyroReadings)` fuses accelerometer and gyroscope data to update the state estimate.

Input Arguments

FUSE — INS filter object
`insfilterErrorState`

`insfilterErrorState`, specified as an object.

`accelReadings` — Accelerometer readings in local sensor body coordinate system (m/s^2)
3-element row vector

Accelerometer readings in m/s^2, specified as a 3-element row vector.

Data Types: `single` | `double`

`gyroReadings` — Gyroscope readings in local sensor body coordinate system (rad/s)
3-element row vector

Gyroscope readings in rad/s, specified as a 3-element row vector.

Data Types: `single` | `double`

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

`insfilterErrorState` | `insfilter`

Introduced in R2019a

reset

Reset internal states for `insfilterErrorState`

Syntax

```
reset(FUSE)
```

Description

`reset(FUSE)` resets the `State`, `StateCovariance`, and internal integrators of `FUSE`, an `insfilterErrorState` object, to their default values.

Input Arguments

FUSE — INS filter object

`insfilterErrorState`

`insfilterErrorState`, specified as an object.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`insfilterErrorState` | `insfilter`

Introduced in R2019a

residual

Residuals and residual covariances from direct state measurements for `insfilterErrorState`

Syntax

```
[res, resCov]= residual(FUSE,idx,measurement,measurementCovariance)
```

Description

`[res, resCov]= residual(FUSE,idx,measurement,measurementCovariance)` computes the residual, `res`, and the residual covariance, `resCov`, based on the direct state measurement and measurement covariance. The `measurement` maps directly to the states specified by indices, `idx`.

Input Arguments

FUSE — INS filter object

`insfilterErrorState`

`insfilterErrorState`, specified as an object.

idx — State vector index of measurements to correct

N-element vector of increasing integers in the range [1, 17]

State vector index of measurements to correct, specified as an *N*-element vector of increasing integers in the range [1, 17].

The state values represent:

State	Units	Index
Orientation (quaternion parts)	N/A	1:4
Position (NED)	m	5:7
Velocity (NED)	m/s	8:10
Gyroscope Bias (XYZ)	rad/s	11:13
Accelerometer Bias (XYZ)	m/s^2	14:16
Visual Odometry Scale (XYZ)	N/A	17

Data Types: `single` | `double`

measurement — Direct measurement of state

N-element vector

Direct measurement of state, specified as a *N*-element vector. *N* is the number of elements of the index argument, `idx`.

measurementCovariance — Covariance of measurement

N-by-*N* matrix

Covariance of measurement, specified as an *N*-by-*N* matrix. *N* is the number of elements of the index argument, `idx`.

Output Arguments

`res` — Measurement residual

1-by-*N* vector of real values

Measurement residual, returned as a 1-by-*N* vector of real values.

`resCov` — Residual covariance

N-by-*N* matrix of real values

Residual covariance, returned as a *N*-by-*N* matrix of real values.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`insfilterErrorState`

Introduced in R2020a

residualgps

Residuals and residual covariance from GPS measurements for `insfilterErrorState`

Syntax

```
[res,resCov] = residualgps(FUSE,position,positionCovariance)
[res,resCov] = residualgps(FUSE,position,positionCovariance,velocity,
velocityCovariance)
```

Description

`[res,resCov] = residualgps(FUSE,position,positionCovariance)` computes the residual, `res`, and the residual covariance, `resCov`, based on the GPS position measurement and covariance.

`[res,resCov] = residualgps(FUSE,position,positionCovariance,velocity,velocityCovariance)` computes the residual, `res`, and the residual covariance, `resCov`, based on the GPS position measurement and covariance.

Input Arguments

FUSE — INS filter object
`insfilterErrorState`

`insfilterErrorState`, specified as an object.

`position` — Position of GPS receiver (LLA)
3-element row vector

Position of GPS receiver in geodetic latitude, longitude, and altitude (LLA) specified as a real finite 3-element row vector. Latitude and longitude are in degrees with north and east being positive. Altitude is in meters.

Data Types: `single` | `double`

`positionCovariance` — Position measurement covariance of GPS receiver (m^2)
3-by-3 matrix

Position measurement covariance of GPS receiver in m^2, specified as a 3-by-3 matrix.

Data Types: `single` | `double`

`velocity` — Velocity of GPS receiver in local NED coordinate system (m/s)
3-element row vector

Velocity of the GPS receiver in the local NED coordinate system in m/s, specified as a 3-element row vector.

Data Types: `single` | `double`

`velocityCovariance` — Velocity measurement covariance of GPS receiver (m/s^2)
3-by-3 matrix

Velocity measurement covariance of the GPS receiver in the local NED coordinate system in m/s^2, specified as a 3-by-3 matrix.

Data Types: `single` | `double`

Output Arguments

`res` — Position and velocity residual

1-by-3 vector of real values | 1-by-6 vector of real values

Position and velocity residual, returned as a 1-by-3 vector of real values if the inputs only contain position information, and returned as 1-by-6 vector of real values if the inputs also contain velocity information.

`resCov` — Residual covariance

3-by-3 matrix of real values | 6-by-6 matrix of real values

Residual covariance, returned as a 3-by-3 matrix of real values if the inputs only contain position information, and a 6-by-6 matrix of real values if the inputs also contain velocity information.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`insfilterErrorState`

Introduced in R2020a

residualmvo

Residuals and residual covariance from monocular visual odometry measurements for `insfilterErrorState`

Syntax

```
[pResidual,oResidual,resCov] = residualmvo(FUSE,position,positionCovariance,
ornt,orntCovariance)
```

Description

`[pResidual,oResidual,resCov] = residualmvo(FUSE,position,positionCovariance, ornt,orntCovariance)` computes the residual information based on the monocular visual odometry measurements and covariance.

Input Arguments

FUSE — INS filter object

`insfilterErrorState`

`insfilterErrorState`, specified as an object.

`position` — Position of camera in local NED coordinate system (m)

3-element row vector

Position of camera in the local NED coordinate system in meters, specified as a real finite 3-element row vector.

Data Types: `single` | `double`

`positionCovariance` — Position measurement covariance of MVO (m^2)

scalar | 3-element vector | 3-by-3 matrix

Position measurement covariance of MVO in m^2, specified as a scalar, 3-element vector, or 3-by-3 matrix.

Data Types: `single` | `double`

`ornt` — Orientation of camera with respect to local NED coordinate system

scalar quaternion | rotation matrix

Orientation of the camera with respect to the local NED coordinate system, specified as a scalar quaternion or 3-by-3 rotation matrix. The quaternion or rotation matrix is a frame rotation from the NED coordinate system to the current camera coordinate system.

Data Types: `quaternion` | `single` | `double`

`orntCovariance` — Orientation measurement covariance of monocular visual odometry (rad^2)

scalar | 3-element vector | 3-by-3 matrix

Orientation measurement covariance of monocular visual odometry in rad^2, specified as a scalar, 3-element vector, or 3-by-3 matrix.

Data Types: `single` | `double`

Output Arguments

`pResidual` — Position residual
1-by-3 vector of real values

Position residual, returned as a 1-by-3 vector of real values in meters.

`oResidual` — Rotation vector residual
1-by-3 vector of real values

Rotation vector residual, returned a 1-by-3 vector of real values in radians.

`resCov` — Residual covariance
6-by-6 matrix of real values

Residual covariance, returned as a 6-by-6 matrix of real values.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also
`insfilterErrorState` | `insfilter`

Introduced in R2020a

stateinfo

Display state vector information for `insfilterErrorState`

Syntax

```
stateinfo(FUSE)
info = stateinfo(FUSE)
```

Description

`stateinfo(FUSE)` displays the meaning of each index of the `State` property of `FUSE`, an `insfilterErrorState` object, and the associated units.

`info = stateinfo(FUSE)` returns a structure with fields containing descriptions of the elements of the state vector of the filter, `FUSE`.

Examples

State Information of `insfilterErrorState`

Create an `insfilterErrorState` object.

```
filter = insfilterErrorState;
```

Display the state information of the created filter.

```
stateinfo(filter)
```

```
States                            Units    Index
Orientation (quaternion parts)             1:4
Position (NAV)                    m        5:7
Velocity (NAV)                    m/s      8:10
Gyroscope Bias (XYZ)              rad/s    11:13
Accelerometer Bias (XYZ)          m/s²     14:16
Visual Odometry Scale                      17
```

Output the state information of the filter as a structure.

```
info = stateinfo(filter)
```

```
info = struct with fields:
            Orientation: [1 2 3 4]
               Position: [5 6 7]
               Velocity: [8 9 10]
          GyroscopeBias: [11 12 13]
      AccelerometerBias: [14 15 16]
    VisualOdometryScale: 17
```

Input Arguments

FUSE — INS filter object

`insfilterErrorState`

`insfilterErrorState`, specified as an object.

Output Arguments

`info` — State information

structure

State information, returned as a structure with fields containing descriptions of the elements of the state vector of the filter. The values of each field are the corresponding indices of the state vector.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`insfilterErrorState` | `insfilter`

Introduced in R2019a

pose

Current orientation and position estimate for `insfilterErrorState`

Syntax

```
[position,orientation,velocity] = pose(FUSE)
[position,orientation,velocity] = pose(FUSE,format)
```

Description

`[position,orientation,velocity] = pose(FUSE)` returns the current estimate of the pose of the object tracked by `FUSE`, an `insfilterErrorState` object.

`[position,orientation,velocity] = pose(FUSE,format)` returns the current estimate of the pose with orientation in the specified orientation format.

Input Arguments

FUSE — INS filter object

`insfilterErrorState`

`insfilterErrorState`, specified as an object.

`format` — Output orientation format

`'quaternion'` (default) | `'rotmat'`

Output orientation format, specified as either `'quaternion'` for a `quaternion` or `'rotmat'` for a rotation matrix.

Data Types: `char` | `string`

Output Arguments

`position` — Position estimate expressed in the local coordinate system (m)

3-element row vector

Position estimate expressed in the local coordinate system of the filter in meters, returned as a 3-element row vector.

Data Types: `single` | `double`

`orientation` — Orientation estimate expressed in the local coordinate system

quaternion (default) | 3-by-3 rotation matrix

Orientation estimate expressed in the local coordinate system of the filter, returned as a scalar quaternion or 3-by-3 rotation matrix. The quaternion or rotation matrix represents a frame rotation from the local reference frame of the filter to the body reference frame.

Data Types: `single` | `double` | `quaternion`

`velocity` — Velocity estimate expressed in local coordinate system (m/s)
3-element row vector

Velocity estimate expressed in the local coordinate system of the filter in m/s, returned as a 3-element row vector.

Data Types: `single` | `double`

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`insfilterErrorState` | `insfilter`

Introduced in R2019a

tune

Tune `insfilterErrorState` parameters to reduce estimation error

Syntax

```
tunedMeasureNoise = tune(filter,measureNoise,sensorData,groundTruth)
tunedMeasureNoise = tune( ___ ,config)
```

Description

`tunedMeasureNoise = tune(filter,measureNoise,sensorData,groundTruth)` adjusts the properties of the `insfilterErrorState` filter object, `filter`, and measurement noises to reduce the root-mean-squared (RMS) state estimation error between the fused sensor data and the ground truth. The function also returns the tuned measurement noise, `tunedMeasureNoise`. The function uses the property values in the filter and the measurement noise provided in the `measureNoise` structure as the initial estimate for the optimization algorithm.

`tunedMeasureNoise = tune( ___ ,config)` specifies the tuning configuration based on a `tunerconfig` object, `config`.

Examples

Tune `insfilterErrorState` to Optimize Pose Estimate

Load the recorded sensor data and ground truth data.

```
load('insfilterErrorStateTuneData.mat');
```

Create tables for the sensor data and the truth data.

```
sensorData = table(Accelerometer,Gyroscope, ...
    GPSPosition,GPSVelocity,MVOOrientation, ...
    MVOPosition);
groundTruth = table(Orientation,Position);
```

Create an `insfilterErrorState` filter object.

```
filter = insfilterErrorState('State',initialState, ...
    'StateCovariance',initialStateCovariance);
```

Create a tuner configuration object for the filter. Use the tuner noise function to obtain a set of initial sensor noises used in the filter.

```
cfg = tunerconfig('insfilterErrorState','MaxIterations',40);
measNoise = tunernoise('insfilterErrorState')
```

```
measNoise = struct with fields:
    MVOOrientationNoise: 1
       MVOPositionNoise: 1
       GPSPositionNoise: 1
```

```
       GPSVelocityNoise: 1
```

Tune the filter and obtain the tuned parameters.

```
tunedmn = tune(filter,measNoise,sensorData, ...
    groundTruth,cfg);
```

```
    Iteration    Parameter                Metric
    _________    _________                ______
    1            AccelerometerNoise       4.1291
    1            GyroscopeNoise           4.1291
    1            AccelerometerBiasNoise   4.1290
    1            GyroscopeBiasNoise       4.1290
    1            GPSPositionNoise         4.0213
    1            GPSVelocityNoise         4.0051
    1            MVOPositionNoise         3.9949
    1            MVOOrientationNoise      3.9886
    2            AccelerometerNoise       3.9886
    2            GyroscopeNoise           3.9886
    2            AccelerometerBiasNoise   3.9886
    2            GyroscopeBiasNoise       3.9886
    2            GPSPositionNoise         3.8381
    2            GPSVelocityNoise         3.8268
    2            MVOPositionNoise         3.8219
    2            MVOOrientationNoise      3.8035
    3            AccelerometerNoise       3.8035
    3            GyroscopeNoise           3.8035
    3            AccelerometerBiasNoise   3.8035
    3            GyroscopeBiasNoise       3.8035
    3            GPSPositionNoise         3.6299
    3            GPSVelocityNoise         3.6276
    3            MVOPositionNoise         3.6241
    3            MVOOrientationNoise      3.5911
    4            AccelerometerNoise       3.5911
    4            GyroscopeNoise           3.5911
    4            AccelerometerBiasNoise   3.5911
    4            GyroscopeBiasNoise       3.5911
    4            GPSPositionNoise         3.1728
    4            GPSVelocityNoise         3.1401
    4            MVOPositionNoise         2.7686
    4            MVOOrientationNoise      2.6632
    5            AccelerometerNoise       2.6632
    5            GyroscopeNoise           2.6632
    5            AccelerometerBiasNoise   2.6632
    5            GyroscopeBiasNoise       2.6632
    5            GPSPositionNoise         2.3242
    5            GPSVelocityNoise         2.2291
    5            MVOPositionNoise         2.2291
    5            MVOOrientationNoise      2.0904
    6            AccelerometerNoise       2.0903
    6            GyroscopeNoise           2.0903
    6            AccelerometerBiasNoise   2.0903
    6            GyroscopeBiasNoise       2.0903
    6            GPSPositionNoise         2.0903
    6            GPSVelocityNoise         2.0141
    6            MVOPositionNoise         1.9952
    6            MVOOrientationNoise      1.8497
```

```
7            AccelerometerNoise        1.8497
7            GyroscopeNoise            1.8496
7            AccelerometerBiasNoise    1.8496
7            GyroscopeBiasNoise        1.8496
7            GPSPositionNoise          1.8398
7            GPSVelocityNoise          1.7528
7            MVOPositionNoise          1.7362
7            MVOOrientationNoise       1.5762
8            AccelerometerNoise        1.5762
8            GyroscopeNoise            1.5762
8            AccelerometerBiasNoise    1.5762
8            GyroscopeBiasNoise        1.5762
8            GPSPositionNoise          1.5762
8            GPSVelocityNoise          1.5107
8            MVOPositionNoise          1.4786
8            MVOOrientationNoise       1.3308
9            AccelerometerNoise        1.3308
9            GyroscopeNoise            1.3308
9            AccelerometerBiasNoise    1.3308
9            GyroscopeBiasNoise        1.3308
9            GPSPositionNoise          1.3308
9            GPSVelocityNoise          1.2934
9            MVOPositionNoise          1.2525
9            MVOOrientationNoise       1.1462
10           AccelerometerNoise        1.1462
10           GyroscopeNoise            1.1462
10           AccelerometerBiasNoise    1.1462
10           GyroscopeBiasNoise        1.1462
10           GPSPositionNoise          1.1443
10           GPSVelocityNoise          1.1332
10           MVOPositionNoise          1.0964
10           MVOOrientationNoise       1.0382
11           AccelerometerNoise        1.0382
11           GyroscopeNoise            1.0382
11           AccelerometerBiasNoise    1.0382
11           GyroscopeBiasNoise        1.0382
11           GPSPositionNoise          1.0348
11           GPSVelocityNoise          1.0348
11           MVOPositionNoise          1.0081
11           MVOOrientationNoise       0.9734
12           AccelerometerNoise        0.9734
12           GyroscopeNoise            0.9734
12           AccelerometerBiasNoise    0.9734
12           GyroscopeBiasNoise        0.9734
12           GPSPositionNoise          0.9693
12           GPSVelocityNoise          0.9682
12           MVOPositionNoise          0.9488
12           MVOOrientationNoise       0.9244
13           AccelerometerNoise        0.9244
13           GyroscopeNoise            0.9244
13           AccelerometerBiasNoise    0.9244
13           GyroscopeBiasNoise        0.9244
13           GPSPositionNoise          0.9203
13           GPSVelocityNoise          0.9199
13           MVOPositionNoise          0.9045
13           MVOOrientationNoise       0.8846
14           AccelerometerNoise        0.8846
14           GyroscopeNoise            0.8846
```

```
14          AccelerometerBiasNoise    0.8845
14          GyroscopeBiasNoise        0.8845
14          GPSPositionNoise          0.8807
14          GPSVelocityNoise          0.8807
14          MVOPositionNoise          0.8659
14          MVOOrientationNoise       0.8501
15          AccelerometerNoise        0.8501
15          GyroscopeNoise            0.8501
15          AccelerometerBiasNoise    0.8500
15          GyroscopeBiasNoise        0.8500
15          GPSPositionNoise          0.8457
15          GPSVelocityNoise          0.8453
15          MVOPositionNoise          0.8299
15          MVOOrientationNoise       0.8173
16          AccelerometerNoise        0.8173
16          GyroscopeNoise            0.8173
16          AccelerometerBiasNoise    0.8172
16          GyroscopeBiasNoise        0.8172
16          GPSPositionNoise          0.8122
16          GPSVelocityNoise          0.8116
16          MVOPositionNoise          0.7961
16          MVOOrientationNoise       0.7858
17          AccelerometerNoise        0.7858
17          GyroscopeNoise            0.7858
17          AccelerometerBiasNoise    0.7857
17          GyroscopeBiasNoise        0.7857
17          GPSPositionNoise          0.7807
17          GPSVelocityNoise          0.7800
17          MVOPositionNoise          0.7655
17          MVOOrientationNoise       0.7572
18          AccelerometerNoise        0.7572
18          GyroscopeNoise            0.7572
18          AccelerometerBiasNoise    0.7570
18          GyroscopeBiasNoise        0.7570
18          GPSPositionNoise          0.7525
18          GPSVelocityNoise          0.7520
18          MVOPositionNoise          0.7401
18          MVOOrientationNoise       0.7338
19          AccelerometerNoise        0.7337
19          GyroscopeNoise            0.7337
19          AccelerometerBiasNoise    0.7335
19          GyroscopeBiasNoise        0.7335
19          GPSPositionNoise          0.7293
19          GPSVelocityNoise          0.7290
19          MVOPositionNoise          0.7185
19          MVOOrientationNoise       0.7140
20          AccelerometerNoise        0.7138
20          GyroscopeNoise            0.7138
20          AccelerometerBiasNoise    0.7134
20          GyroscopeBiasNoise        0.7134
20          GPSPositionNoise          0.7086
20          GPSVelocityNoise          0.7068
20          MVOPositionNoise          0.6956
20          MVOOrientationNoise       0.6926
21          AccelerometerNoise        0.6922
21          GyroscopeNoise            0.6922
21          AccelerometerBiasNoise    0.6916
21          GyroscopeBiasNoise        0.6916
```

```
21          GPSPositionNoise          0.6862
21          GPSVelocityNoise          0.6822
21          MVOPositionNoise          0.6682
21          MVOOrientationNoise       0.6667
22          AccelerometerNoise        0.6660
22          GyroscopeNoise            0.6660
22          AccelerometerBiasNoise    0.6650
22          GyroscopeBiasNoise        0.6650
22          GPSPositionNoise          0.6605
22          GPSVelocityNoise          0.6541
22          MVOPositionNoise          0.6372
22          MVOOrientationNoise       0.6368
23          AccelerometerNoise        0.6356
23          GyroscopeNoise            0.6356
23          AccelerometerBiasNoise    0.6344
23          GyroscopeBiasNoise        0.6344
23          GPSPositionNoise          0.6324
23          GPSVelocityNoise          0.6252
23          MVOPositionNoise          0.6087
23          MVOOrientationNoise       0.6087
24          AccelerometerNoise        0.6075
24          GyroscopeNoise            0.6075
24          AccelerometerBiasNoise    0.6068
24          GyroscopeBiasNoise        0.6068
24          GPSPositionNoise          0.6061
24          GPSVelocityNoise          0.6032
24          MVOPositionNoise          0.6032
24          MVOOrientationNoise       0.6032
25          AccelerometerNoise        0.6017
25          GyroscopeNoise            0.6017
25          AccelerometerBiasNoise    0.6012
25          GyroscopeBiasNoise        0.6012
25          GPSPositionNoise          0.6010
25          GPSVelocityNoise          0.6005
25          MVOPositionNoise          0.6005
25          MVOOrientationNoise       0.6005
26          AccelerometerNoise        0.5992
26          GyroscopeNoise            0.5992
26          AccelerometerBiasNoise    0.5987
26          GyroscopeBiasNoise        0.5987
26          GPSPositionNoise          0.5983
26          GPSVelocityNoise          0.5983
26          MVOPositionNoise          0.5983
26          MVOOrientationNoise       0.5980
27          AccelerometerNoise        0.5973
27          GyroscopeNoise            0.5973
27          AccelerometerBiasNoise    0.5972
27          GyroscopeBiasNoise        0.5972
27          GPSPositionNoise          0.5970
27          GPSVelocityNoise          0.5970
27          MVOPositionNoise          0.5970
27          MVOOrientationNoise       0.5970
28          AccelerometerNoise        0.5970
28          GyroscopeNoise            0.5970
28          AccelerometerBiasNoise    0.5970
28          GyroscopeBiasNoise        0.5970
28          GPSPositionNoise          0.5970
28          GPSVelocityNoise          0.5970
```

```
28          MVOPositionNoise          0.5969
28          MVOOrientationNoise       0.5966
29          AccelerometerNoise        0.5966
29          GyroscopeNoise            0.5966
29          AccelerometerBiasNoise    0.5966
29          GyroscopeBiasNoise        0.5966
29          GPSPositionNoise          0.5966
29          GPSVelocityNoise          0.5966
29          MVOPositionNoise          0.5965
29          MVOOrientationNoise       0.5965
30          AccelerometerNoise        0.5965
30          GyroscopeNoise            0.5965
30          AccelerometerBiasNoise    0.5965
30          GyroscopeBiasNoise        0.5965
30          GPSPositionNoise          0.5964
30          GPSVelocityNoise          0.5964
30          MVOPositionNoise          0.5964
30          MVOOrientationNoise       0.5961
31          AccelerometerNoise        0.5961
31          GyroscopeNoise            0.5961
31          AccelerometerBiasNoise    0.5960
31          GyroscopeBiasNoise        0.5960
31          GPSPositionNoise          0.5960
31          GPSVelocityNoise          0.5960
31          MVOPositionNoise          0.5960
31          MVOOrientationNoise       0.5958
32          AccelerometerNoise        0.5956
32          GyroscopeNoise            0.5956
32          AccelerometerBiasNoise    0.5956
32          GyroscopeBiasNoise        0.5956
32          GPSPositionNoise          0.5955
32          GPSVelocityNoise          0.5955
32          MVOPositionNoise          0.5954
32          MVOOrientationNoise       0.5954
33          AccelerometerNoise        0.5954
33          GyroscopeNoise            0.5954
33          AccelerometerBiasNoise    0.5953
33          GyroscopeBiasNoise        0.5953
33          GPSPositionNoise          0.5953
33          GPSVelocityNoise          0.5953
33          MVOPositionNoise          0.5952
33          MVOOrientationNoise       0.5950
34          AccelerometerNoise        0.5949
34          GyroscopeNoise            0.5949
34          AccelerometerBiasNoise    0.5949
34          GyroscopeBiasNoise        0.5949
34          GPSPositionNoise          0.5948
34          GPSVelocityNoise          0.5947
34          MVOPositionNoise          0.5946
34          MVOOrientationNoise       0.5946
35          AccelerometerNoise        0.5946
35          GyroscopeNoise            0.5946
35          AccelerometerBiasNoise    0.5946
35          GyroscopeBiasNoise        0.5946
35          GPSPositionNoise          0.5946
35          GPSVelocityNoise          0.5945
35          MVOPositionNoise          0.5944
35          MVOOrientationNoise       0.5943
```

```
    36          AccelerometerNoise        0.5942
    36          GyroscopeNoise            0.5942
    36          AccelerometerBiasNoise    0.5942
    36          GyroscopeBiasNoise        0.5942
    36          GPSPositionNoise          0.5941
    36          GPSVelocityNoise          0.5941
    36          MVOPositionNoise          0.5939
    36          MVOOrientationNoise       0.5939
    37          AccelerometerNoise        0.5939
    37          GyroscopeNoise            0.5939
    37          AccelerometerBiasNoise    0.5939
    37          GyroscopeBiasNoise        0.5939
    37          GPSPositionNoise          0.5939
    37          GPSVelocityNoise          0.5939
    37          MVOPositionNoise          0.5937
    37          MVOOrientationNoise       0.5937
    38          AccelerometerNoise        0.5937
    38          GyroscopeNoise            0.5937
    38          AccelerometerBiasNoise    0.5937
    38          GyroscopeBiasNoise        0.5937
    38          GPSPositionNoise          0.5936
    38          GPSVelocityNoise          0.5935
    38          MVOPositionNoise          0.5935
    38          MVOOrientationNoise       0.5935
    39          AccelerometerNoise        0.5935
    39          GyroscopeNoise            0.5935
    39          AccelerometerBiasNoise    0.5935
    39          GyroscopeBiasNoise        0.5935
    39          GPSPositionNoise          0.5934
    39          GPSVelocityNoise          0.5934
    39          MVOPositionNoise          0.5934
    39          MVOOrientationNoise       0.5934
    40          AccelerometerNoise        0.5934
    40          GyroscopeNoise            0.5934
    40          AccelerometerBiasNoise    0.5934
    40          GyroscopeBiasNoise        0.5934
    40          GPSPositionNoise          0.5933
    40          GPSVelocityNoise          0.5933
    40          MVOPositionNoise          0.5933
    40          MVOOrientationNoise       0.5933
```

Fuse the sensor data using the tuned filter.

```
N = size(sensorData,1);
qEstTuned = quaternion.zeros(N,1);
posEstTuned = zeros(N,3);
for ii=1:N
    predict(filter, Accelerometer(ii,:),Gyroscope(ii,:));
    if all(~isnan(GPSPosition(ii,1)))
        fusegps(filter,GPSPosition(ii,:), ...
            tunedmn.GPSPositionNoise,GPSVelocity(ii,:), ...
            tunedmn.GPSVelocityNoise);
    end
    if all(~isnan(MVOPosition(ii,1)))
        fusemvo(filter,MVOPosition(ii,:),tunedmn.MVOPositionNoise, ...
            MVOOrientation{ii},tunedmn.MVOOrientationNoise);
    end
```

```
    [posEstTuned(ii,:),qEstTuned(ii,:)] = pose(filter);
end
```

Compute the RMS errors.

```
orientationErrorTuned = rad2deg(dist(qEstTuned,Orientation));
rmsOrientationErrorTuned = sqrt(mean(orientationErrorTuned.^2))
```

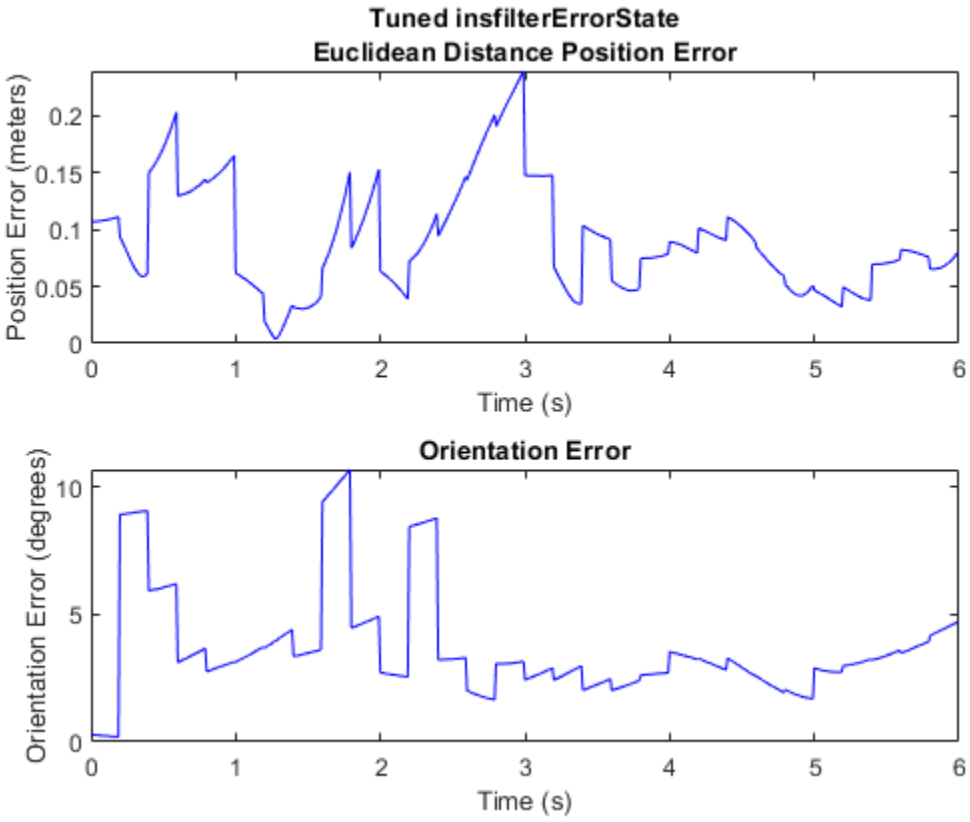
```
rmsOrientationErrorTuned = 4.2335
```

```
positionErrorTuned = sqrt(sum((posEstTuned - Position).^2,2));
rmsPositionErrorTuned = sqrt(mean( positionErrorTuned.^2))
```

```
rmsPositionErrorTuned = 0.1024
```

Visualize the results.

```
figure;
t = (0:N-1)./filter.IMUSampleRate;
subplot(2,1,1)
plot(t, positionErrorTuned,'b');
title("Tuned insfilterErrorState" + newline + ...
    "Euclidean Distance Position Error")
xlabel('Time (s)');
ylabel('Position Error (meters)')
subplot(2,1,2)
plot(t, orientationErrorTuned,'b');
title("Orientation Error")
xlabel('Time (s)');
ylabel('Orientation Error (degrees)');
```



Input Arguments

`filter` — Filter object

`insfilterErrorState` object

Filter object, specified as an `insfilterErrorState` object.

`measureNoise` — Measurement noise

structure

Measurement noise, specified as a structure. The function uses the measurement noise input as the initial guess for tuning the measurement noise. The structure must contain these fields:

Field name	Description
`MVOOrientationNoise`	Orientation measurement covariance of monocular visual odometry, specified as a scalar, 3-element vector, or 3-by-3 matrix in rad^2
`MVOPositionNoise`	Position measurement covariance of MVO, specified as a scalar, 3-element vector, or 3-by-3 matrix in m^2
`GPSPositionNoise`	Variance of GPS position noise, specified as a scalar in m^2

Field name	Description
`GPSVelocityNoise`	Variance of GPS velocity noise, specified as a scalar in $(m/s)^2$

`sensorData` — Sensor data
`table`

Sensor data, specified as a table. In each row, the sensor data is specified as:

- `Accelerometer` — Accelerometer data, specified as a 1-by-3 vector of scalars in m^2/s.
- `Gyroscope`— Gyroscope data, specified as a 1-by-3 vector of scalars in rad/s.
- `MVOOrienation` — Orientation of the camera with respect to the local navigation frame, specified as a scalar quaternion or 3-by-3 rotation matrix. The quaternion or rotation matrix is a frame rotation from the local navigation frame to the current camera coordinate system.
- `MVOPosition` — Position of camera in the local navigation frame, specified as a real 3-element row vector in meters.
- `GPSPosition` — GPS position data, specified as a 1-by-3 vector of scalars in meters.
- `GPSVelocity` — GPS velocity data, specified as a 1-by-3 vector of scalars in m/s.

If the GPS sensor does not produce complete measurements, specify the corresponding entry for `GPSPosition` and/or `GPSVelocity` as `NaN`. If you set the `Cost` property of the tuner configuration input, `config`, to `Custom`, then you can use other data types for the `sensorData` input based on your choice.

`groundTruth` — Ground truth data
`table`

Ground truth data, specified as a `table`. In each row, the table can optionally contain any of these variables:

- `Orientation` — Orientation from the navigation frame to the body frame, specified as a `quaternion` or a 3-by-3 rotation matrix.
- `Position` — Position in navigation frame, specified as a 1-by-3 vector of scalars in meters.
- `Velocity` — Velocity in navigation frame, specified as a 1-by-3 vector of scalars in m/s.
- `AccelerometerBias` — Accelerometer delta angle bias in body frame, specified as a 1-by-3 vector of scalars in m^2/s.
- `VisualOdometryScale` — Visual odometry scale factor, specified as a scalar.

The function processes each row of the `sensorData` and `groundTruth` tables sequentially to calculate the state estimate and RMS error from the ground truth. State variables not present in `groundTruth` input are ignored for the comparison. The `sensorData` and the `groundTruth` tables must have the same number of rows.

If you set the `Cost` property of the tuner configuration input, `config`, to `Custom`, then you can use other data types for the `groundTruth` input based on your choice.

`config` — Tuner configuration
`tunerconfig` object

Tuner configuration, specified as a `tunerconfig` object.

Output Arguments

`tunedMeasureNoise` — Tuned measurement noise

structure

Tuned measurement noise, returned as a structure. The structure contains these fields.

Field name	Description
`MVOOrientationNoise`	Orientation measurement covariance of monocular visual odometry, specified as a scalar, 3-element vector, or 3-by-3 matrix in rad^2
`MVOPositionNoise`	Position measurement covariance of MVO, specified as a scalar, 3-element vector, or 3-by-3 matrix in m^2
`GPSPositionNoise`	Variance of GPS position noise, specified as a scalar in m^2
`GPSVelocityNoise`	Variance of GPS velocity noise, specified as a scalar in $(m/s)^2$

References

[1] Abbeel, P., Coates, A., Montemerlo, M., Ng, A.Y. and Thrun, S. Discriminative Training of Kalman Filters. In *Robotics: Science and systems*, Vol. 2, pp. 1, 2005.

See Also

`tunerconfig` | `tunernoise`

Introduced in R2021a

gpsSensor

GPS receiver simulation model

Description

The `gpsSensor` System object models data output from a Global Positioning System (GPS) receiver.

To model a GPS receiver:

1. Create the `gpsSensor` object and set its properties.
2. Call the object with arguments, as if it were a function.

To learn more about how System objects work, see What Are System Objects?

Creation

Syntax

```
GPS = gpsSensor
GPS = gpsSensor('ReferenceFrame',RF)
GPS = gpsSensor(___,Name,Value)
```

Description

`GPS = gpsSensor` returns a `gpsSensor` System object that computes a Global Positioning System receiver reading based on a local position and velocity input signal. The default reference position in geodetic coordinates is

- latitude: 0° N
- longitude: 0° E
- altitude: 0 m

`GPS = gpsSensor('ReferenceFrame',RF)` returns a `gpsSensor` System object that computes a global positioning system receiver reading relative to the reference frame `RF`. Specify `RF` as `'NED'` (North-East-Down) or `'ENU'`(East-North-Up). The default value is `'NED'`.

`GPS = gpsSensor(___,Name,Value)` sets each property `Name` to the specified `Value`. Unspecified properties have default values.

Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see System Design in MATLAB Using System Objects.

`SampleRate` — Update rate of receiver (Hz)
`1` (default) | positive real scalar

Update rate of the receiver in Hz, specified as a positive real scalar.

Data Types: `single` | `double`

`ReferenceLocation` — Origin of local navigation reference frame
`[0 0 0]` (default) | [degrees degrees meters]

Reference location, specified as a 3-element row vector in geodetic coordinates (latitude, longitude, and altitude). Altitude is the height above the reference ellipsoid model, WGS84. The reference location is in [degrees degrees meters]. The degree format is decimal degrees (DD).

Data Types: `single` | `double`

`PositionInputFormat` — Position coordinate input format
`'Local'` (default) | `'Geodetic'`

Position coordinate input format, specified as `'Local'` or `'Geodetic'`.

- If you set the property as `'Local'`, then you need to specify the `truePosition` input as Cartesian coordinates with respect to the local navigation frame whose origin is fixed and defined by the `ReferenceLcation` property. Additionally, when you specify the `trueVelocity` input, you need to specify it with respect to this local navigation frame.
- If you set the property as `'Geodetic'`, then you need to specify the `truePosition` input as geodetic coordinates in latitude, longitude, and altitude. Additionally, when you specify the `trueVelocity` input, you need to specify it with respect to the navigation frame (NED or ENU) whose origin corresponds to the `truePosition` input. When setting the property as `'Geodetic'`, the `gpsSensor` object neglects the `ReferenceLocation` property.

Data Types: `character vector`

`HorizontalPositionAccuracy` — Horizontal position accuracy (m)
`1.6` (default) | nonnegative real scalar

Horizontal position accuracy in meters, specified as a nonnegative real scalar. The horizontal position accuracy specifies the standard deviation of the noise in the horizontal position measurement.

Tunable: Yes

Data Types: `single` | `double`

`VerticalPositionAccuracy` — Vertical position accuracy (m)
`3` (default) | nonnegative real scalar

Vertical position accuracy in meters, specified as a nonnegative real scalar. The vertical position accuracy specifies the standard deviation of the noise in the vertical position measurement.

Tunable: Yes

Data Types: `single` | `double`

`VelocityAccuracy` — Velocity accuracy (m/s)

`0.1` (default) | nonnegative real scalar

Velocity accuracy in meters per second, specified as a nonnegative real scalar. The velocity accuracy specifies the standard deviation of the noise in the velocity measurement.

Tunable: Yes

Data Types: `single` | `double`

`DecayFactor` — Global position noise decay factor

`0.999` (default) | scalar in the range [0,1]

Global position noise decay factor, specified as a scalar in the range [0,1].

A decay factor of 0 models the global position noise as a white noise process. A decay factor of 1 models the global position noise as a random walk process.

Tunable: Yes

Data Types: `single` | `double`

`RandomStream` — Random number source

`'Global stream'` (default) | `'mt19937ar with seed'`

Random number source, specified as a character vector or string:

- `'Global stream'` –– Random numbers are generated using the current global random number stream.
- `'mt19937ar with seed'` –– Random numbers are generated using the mt19937ar algorithm with the seed specified by the `Seed` property.

Data Types: `char` | `string`

`Seed` — Initial seed

`67` (default) | nonnegative integer scalar

Initial seed of an mt19937ar random number generator algorithm, specified as a nonnegative integer scalar.

Dependencies

To enable this property, set `RandomStream` to `'mt19937ar with seed'`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Usage

Syntax

```
[position,velocity,groundspeed,course] = GPS(truePosition,trueVelocity)
```

Description

`[position,velocity,groundspeed,course] = GPS(truePosition,trueVelocity)` computes global navigation satellite system receiver readings from the position and velocity inputs.

Input Arguments

`truePosition` — Position of GPS receiver in navigation coordinate system

N-by-3 matrix

Position of the GPS receiver in the navigation coordinate system, specified as a real finite *N*-by-3 matrix. *N* is the number of samples in the current frame.

- When the `PositionInputFormat` property is specified as `'Local'`, specify `turePosition` as Cartesian coordinates with respect to the local navigation frame whose origin is fixed at `ReferenceLocation`.
- When the `PositionInputFormat` property is specified as `'Geodetic'`, specify `turePosition` as geodetic coordinates in `[latitude,longitude,altitude]`. `Latitude` and `longitude` are in meters. `altitude` is the height above the WGS84 ellipsoid model in meters.

Data Types: `single` | `double`

`trueVelocity` — Velocity of GPS receiver in navigation coordinate system (m/s)

N-by-3 matrix

Velocity of GPS receiver in the navigation coordinate system in meters per second, specified as a real finite *N*-by-3 matrix. *N* is the number of samples in the current frame.

- When the `PositionInputFormat` property is specified as `'Local'`, specify `trueVelocity` with respect to the local navigation frame (NED or ENU) whose origin is fixed at `ReferenceLocation`.
- When the `PositionInputFormat` property is specified as `'Geodetic'`, specify `trueVelocity` with respect to the navigation frame (NED or ENU) whose origin corresponds to the `truePosition` input.

Data Types: `single` | `double`

Output Arguments

`position` — Position in LLA coordinate system

N-by-3 matrix

Position of the GPS receiver in the geodetic latitude, longitude, and altitude (LLA) coordinate system, returned as a real finite *N*-by-3 array. Latitude and longitude are in degrees with North and East being positive. Altitude is in meters.

N is the number of samples in the current frame.

Data Types: `single` | `double`

`velocity` — Velocity in local navigation coordinate system (m/s)

N-by-3 matrix

Velocity of the GPS receiver in the local navigation coordinate system in meters per second, returned as a real finite *N*-by-3 array. *N* is the number of samples in the current frame.

- When the `PositionInputFormat` property is specified as `'Local'`, the returned velocity is with respect to the local navigation frame whose origin is fixed at `ReferenceLocation`.

- When the `PositionInputFormat` property is specified as `'Geodetic'`, the returned velocity is with respect to the navigation frame (NED or ENU) whose origin corresponds to the `position` output.

Data Types: `single` | `double`

`groundspeed` — Magnitude of horizontal velocity in local navigation coordinate system (m/s)
N-by-1 column vector

Magnitude of the horizontal velocity of the GPS receiver in the local navigation coordinate system in meters per second, returned as a real finite *N*-by-1 column vector.

N is the number of samples in the current frame.

Data Types: `single` | `double`

`course` — Direction of horizontal velocity in local navigation coordinate system (°)
N-by-1 column vector

Direction of the horizontal velocity of the GPS receiver in the local navigation coordinate system in degrees, returned as a real finite *N*-by-1 column of values between 0 and 360. North corresponds to 360 degrees and East corresponds to 90 degrees.

N is the number of samples in the current frame.

Data Types: `single` | `double`

Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

Common to All System Objects

step	Run System object algorithm
release	Release resources and allow changes to System object property values and input characteristics
reset	Reset internal states of System object

Examples

Generate GPS Position Measurements From Stationary Input

Create a `gpsSensor` System object™ to model GPS receiver data. Assume a typical one Hz sample rate and a 1000-second simulation time. Define the reference location in terms of latitude, longitude, and altitude (LLA) of Natick, MA (USA). Define the sensor as stationary by specifying the true position and velocity with zeros.

```
fs = 1;
duration = 1000;
numSamples = duration*fs;
```

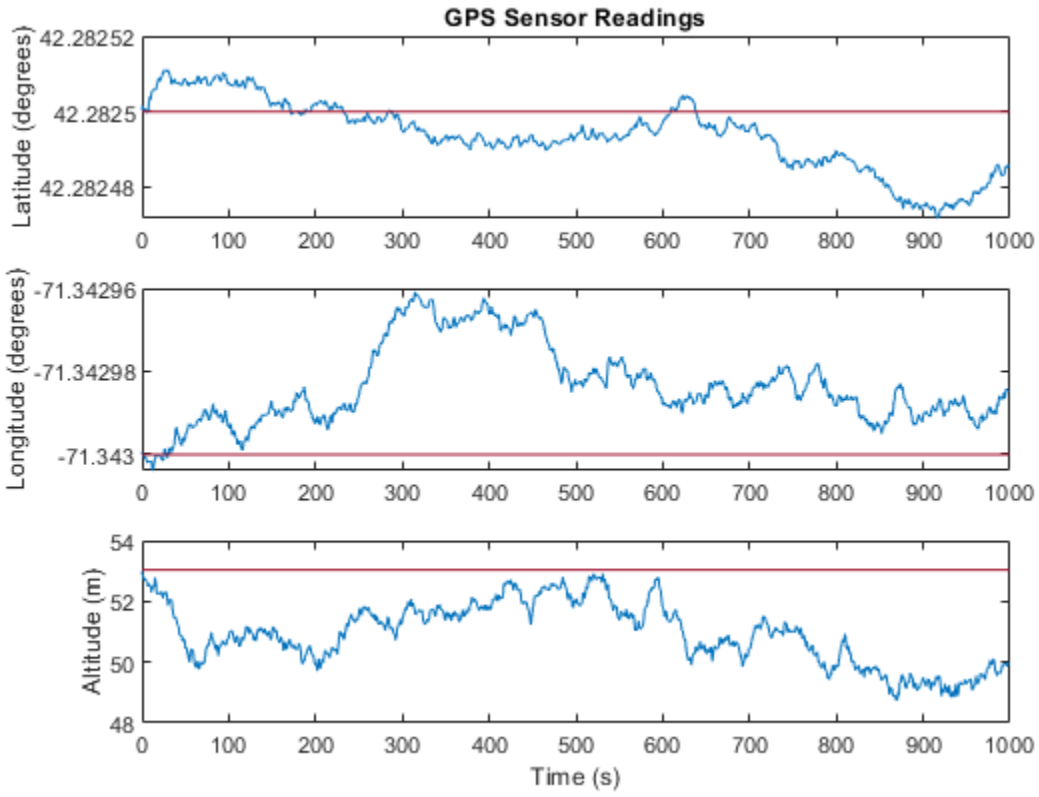
```
refLoc = [42.2825 -71.343 53.0352];

truePosition = zeros(numSamples,3);
trueVelocity = zeros(numSamples,3);

gps = gpsSensor('SampleRate',fs,'ReferenceLocation',refLoc);
```

Call `gps` with the specified `truePosition` and `trueVelocity` to simulate receiving GPS data for a stationary platform.

```
position = gps(truePosition,trueVelocity);
```

Plot the true position and the GPS sensor readings for position.

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

subplot(3, 1, 1)
plot(t, position(:,1), ...
     t, ones(numSamples)*refLoc(1))
title('GPS Sensor Readings')
ylabel('Latitude (degrees)')

subplot(3, 1, 2)
plot(t, position(:,2), ...
     t, ones(numSamples)*refLoc(2))
ylabel('Longitude (degrees)')

subplot(3, 1, 3)
plot(t, position(:,3), ...
     t, ones(numSamples)*refLoc(3))
ylabel('Altitude (m)')
xlabel('Time (s)')
```



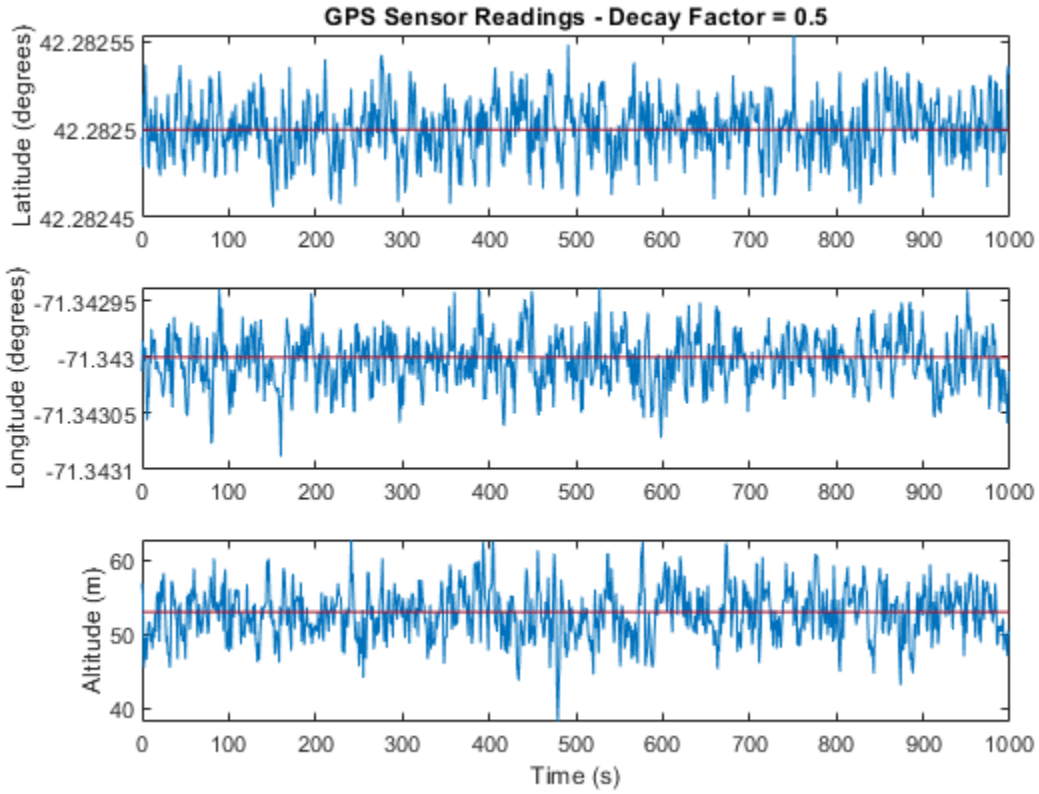
The position readings have noise controlled by `HorizontalPositionAccuracy`, `VerticalPositionAccuracy`, `VelocityAccuracy`, and `DecayFactor`. The `DecayFactor` property controls the drift in the noise model. By default, `DecayFactor` is set to `0.999`, which approaches a random walk process. To observe the effect of the `DecayFactor` property:

1. Reset the `gps` object.
2. Set `DecayFactor` to `0.5`.
3. Call `gps` with variables specifying a stationary position.
4. Plot the results.

The GPS position readings now oscillate around the true position.

```
reset(gps)
gps.DecayFactor = 0.5;
position = gps(truePosition,trueVelocity);

subplot(3, 1, 1)
plot(t, position(:,1), ...
     t, ones(numSamples)*refLoc(1))
title('GPS Sensor Readings - Decay Factor = 0.5')
ylabel('Latitude (degrees)')

subplot(3, 1, 2)
plot(t, position(:,2), ...
     t, ones(numSamples)*refLoc(2))
ylabel('Longitude (degrees)')
```

```
subplot(3, 1, 3)
plot(t, position(:,3), ...
     t, ones(numSamples)*refLoc(3))
ylabel('Altitude (m)')
xlabel('Time (s)')
```



Relationship Between Groundspeed and Course Accuracy

GPS receivers achieve greater course accuracy as groundspeed increases. In this example, you create a GPS receiver simulation object and simulate the data received from a platform that is accelerating from a stationary position.

Create a default `gpsSensor` System object™ to model data returned by a GPS receiver.

```
GPS = gpsSensor
```

```
GPS =
  gpsSensor with properties:

                    SampleRate: 1                   Hz
           PositionInputFormat: 'Local'
             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
```

Create matrices to describe the position and velocity of a platform in the NED coordinate system. The platform begins from a stationary position and accelerates to 60 m/s North-East over 60 seconds, then has a vertical acceleration to 2 m/s over 2 seconds, followed by a 2 m/s rate of climb for another 8 seconds. Assume a constant velocity, such that the velocity is the simple derivative of the position.

```
duration = 70;
numSamples = duration*GPS.SampleRate;

course = 45*ones(duration,1);
groundspeed = [(1:60)';60*ones(10,1)];

Nvelocity   = groundspeed.*sind(course);
Evelocity   = groundspeed.*cosd(course);
Dvelocity   = [zeros(60,1);-1;-2*ones(9,1)];
NEDvelocity = [Nvelocity,Evelocity,Dvelocity];

Ndistance   = cumsum(Nvelocity);
Edistance   = cumsum(Evelocity);
Ddistance   = cumsum(Dvelocity);
NEDposition = [Ndistance,Edistance,Ddistance];
```

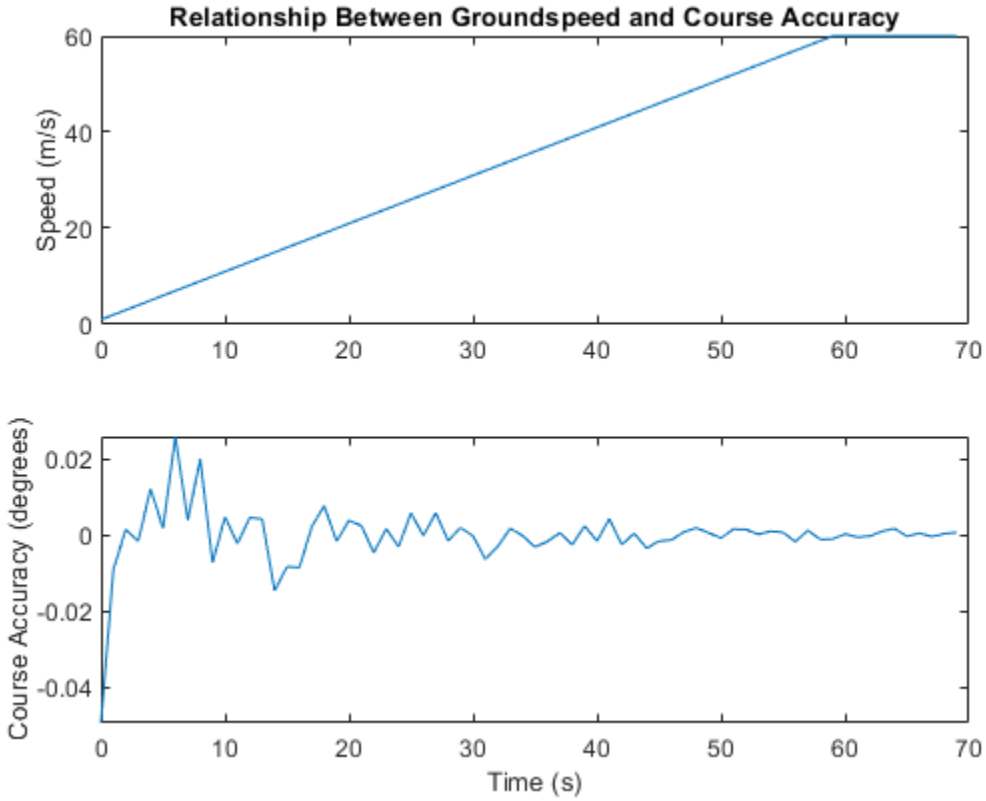
Model GPS measurement data by calling the GPS object with your velocity and position matrices.

```
[~,~,groundspeedMeasurement,courseMeasurement] = GPS(NEDposition,NEDvelocity);
```

Plot the groundspeed and the difference between the true course and the course returned by the GPS simulator.

As groundspeed increases, the accuracy of the course increases. Note that the velocity increase during the last ten seconds has no effect, because the additional velocity is not in the ground plane.

```
t = (0:numSamples-1)/GPS.SampleRate;

subplot(2,1,1)
plot(t,groundspeed);
ylabel('Speed (m/s)')
title('Relationship Between Groundspeed and Course Accuracy')

subplot(2,1,2)
courseAccuracy = courseMeasurement - course;
plot(t,courseAccuracy)
xlabel('Time (s)');
ylabel('Course Accuracy (degrees)')
```



Model GPS Receiver Data

Simulate GPS data received during a trajectory from the city of Natick, MA, to Boston, MA.

Define the decimal degree latitude and longitude for the city of Natick, MA USA, and Boston, MA USA. For simplicity, set the altitude for both locations to zero.

```
NatickLLA = [42.27752809999999, -71.34680909999997, 0];
BostonLLA = [42.3600825, -71.05888010000001, 0];
```

Define a motion that can take a platform from Natick to Boston in 20 minutes. Set the origin of the local NED coordinate system as Natick. Create a `waypointTrajectory` object to output the trajectory 10 samples at a time.

```
fs = 1;
duration = 60*20;

bearing = 68; % degrees
distance = 25.39e3; % meters
distanceEast = distance*sind(bearing);
distanceNorth = distance*cosd(bearing);

NatickNED = [0,0,0];
BostonNED = [distanceNorth,distanceEast,0];
```

```
trajectory = waypointTrajectory( ...
    'Waypoints', [NatickNED;BostonNED], ...
    'TimeOfArrival',[0;duration], ...
    'SamplesPerFrame',10, ...
    'SampleRate',fs);
```

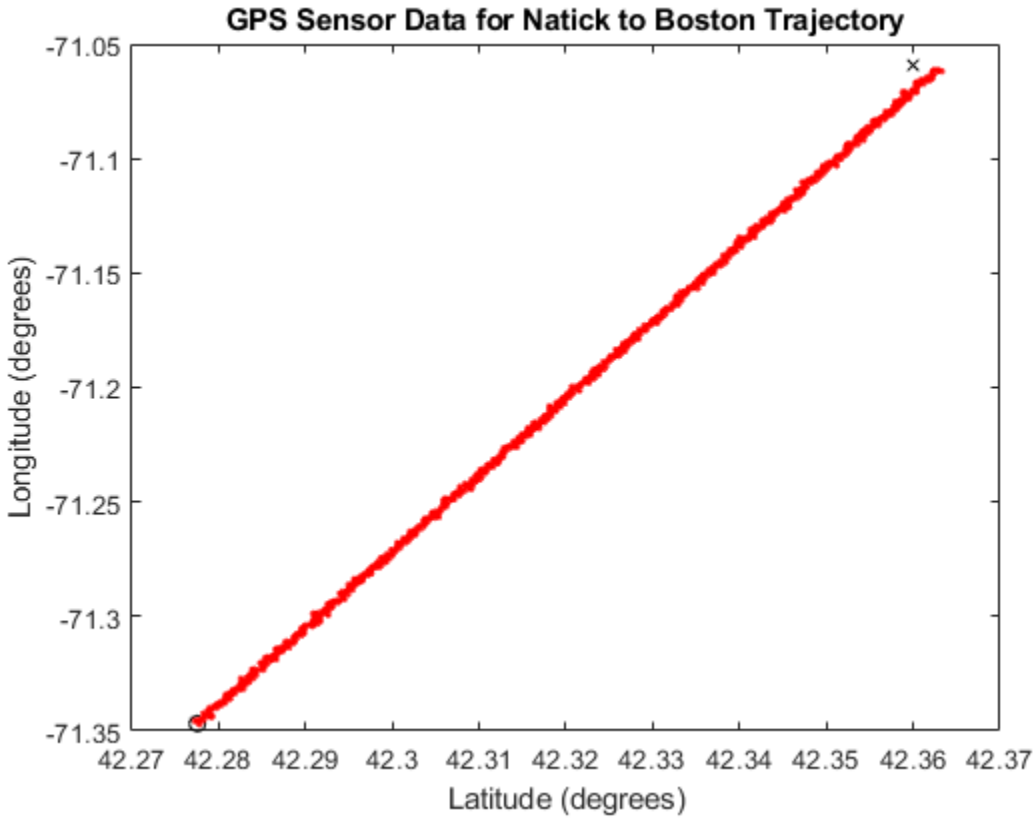
Create a `gpsSensor` object to model receiving GPS data for the platform. Set the `HorizontalPositionalAccuracy` to `25` and the `DecayFactor` to `0.25` to emphasize the noise. Set the `ReferenceLocation` to the Natick coordinates in LLA.

```
GPS = gpsSensor( ...
    'HorizontalPositionAccuracy',25, ...
    'DecayFactor',0.25, ...
    'SampleRate',fs, ...
    'ReferenceLocation',NatickLLA);
```

Open a figure and plot the position of Natick and Boston in LLA. Ignore altitude for simplicity.

In a loop, call the `gpsSensor` object with the ground-truth trajectory to simulate the received GPS data. Plot the ground-truth trajectory and the model of received GPS data.

```
figure(1)
plot(NatickLLA(1),NatickLLA(2),'ko', ...
     BostonLLA(1),BostonLLA(2),'kx')
xlabel('Latitude (degrees)')
ylabel('Longitude (degrees)')
title('GPS Sensor Data for Natick to Boston Trajectory')
hold on

while ~isDone(trajectory)
    [truePositionNED,~,trueVelocityNED] = trajectory();
    reportedPositionLLA = GPS(truePositionNED,trueVelocityNED);

    figure(1)
    plot(reportedPositionLLA(:,1),reportedPositionLLA(:,2),'r.')
end
```



As a best practice, release System objects when complete.

```
release(GPS)
release(trajectory)
```

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

Objects

`insSensor` | `imuSensor`

Introduced in R2019b

gyroparams

Gyroscope sensor parameters

Description

The `gyroparams` class creates a gyroscope sensor parameters object. You can use this object to model a gyroscope when simulating an IMU with `imuSensor`. See the “Algorithms” on page 2-438 section of `imuSensor` for details of `gyroparams` modeling.

Creation

Syntax

```
params = gyroparams
params = gyroparams(Name,Value)
```

Description

`params = gyroparams` returns an ideal gyroscope sensor parameters object with default values.

`params = gyroparams(Name,Value)` configures `gyroparams` object properties using one or more `Name,Value` pair arguments. `Name` is a property name and `Value` is the corresponding value. `Name` must appear inside single quotes (`''`). You can specify several name-value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`. Any unspecified properties take default values.

Properties

`MeasurementRange` — Maximum sensor reading (rad/s)
`Inf` (default) | real positive scalar

Maximum sensor reading in rad/s, specified as a real positive scalar.

Data Types: `single` | `double`

`Resolution` — Resolution of sensor measurements ((rad/s)/LSB)
`0` (default) | real nonnegative scalar

Resolution of sensor measurements in (rad/s)/LSB, specified as a real nonnegative scalar. Here, LSB is the acronym for least significant bit.

Data Types: `single` | `double`

`ConstantBias` — Constant sensor offset bias (rad/s)
`[0 0 0]` (default) | real scalar | real 3-element row vector

Constant sensor offset bias in rad/s, specified as a real scalar or 3-element row vector. Any scalar input is converted into a real 3-element row vector where each element has the input scalar value.

Data Types: `single` | `double`

`AxesMisalignment` — Sensor axes skew (%)
`diag([100 100 100])` (default) | scalar in the range [0,100] | 3-element row vector in the range [0,100] | 3-by-3 matrix in the range [0,100]

Sensor axes skew in percentage, specified as a scalar, a 3-element row vector, or a 3-by-3 matrix with values ranging from `0` to `100`. The diagonal elements of the matrix account for the misalignment effects for each axes. The off-diagonal elements account for the cross-axes misalignment effects. The measured state $v_{measure}$ is obtained from the true state $v_{measure}$ via the misalignment matrix as:

$$v_{measure} = \frac{1}{100} M v_{true} = \frac{1}{100} \begin{bmatrix} m_{11} & m_{12} & m_{13} \\ m_{21} & m_{22} & m_{23} \\ m_{31} & m_{32} & m_{33} \end{bmatrix} v_{true}$$

- If you specify the property as a scalar, then all the off-diagonal elements of the matrix take the value of the specified scalar and all the diagonal elements are 100.
- If you specify the property as a vector [*a b c*], then $m_{21} = m_{31} = a$, $m_{12} = m_{32} = b$, and $m_{13} = m_{23} = c$. All the diagonal elements are 100.

Data Types: `single` | `double`

`NoiseDensity` — Power spectral density of sensor noise ((rad/s)/√Hz)
`[0 0 0]` (default) | real scalar | real 3-element row vector

Power spectral density of sensor noise in (rad/s)/√Hz, specified as a real scalar or 3-element row vector. This property corresponds to the angle random walk (ARW). Any scalar input is converted into a real 3-element row vector where each element has the input scalar value.

Data Types: `single` | `double`

`BiasInstability` — Instability of the bias offset (rad/s)
`[0 0 0]` (default) | real scalar | real 3-element row vector

Instability of the bias offset in rad/s, specified as a real scalar or 3-element row vector. Any scalar input is converted into a real 3-element row vector where each element has the input scalar value.

Data Types: `single` | `double`

`RandomWalk` — Integrated white noise of sensor ((rad/s)(√Hz))
`[0 0 0]` (default) | real scalar | real 3-element row vector

Integrated white noise of sensor in (rad/s)(√Hz), specified as a real scalar or 3-element row vector. Any scalar input is converted into a real 3-element row vector where each element has the input scalar value.

Data Types: `single` | `double`

`TemperatureBias` — Sensor bias from temperature ((rad/s)/°C)
`[0 0 0]` (default) | real scalar | real 3-element row vector

Sensor bias from temperature in ((rad/s)/°C), specified as a real scalar or 3-element row vector. Any scalar input is converted into a real 3-element row vector where each element has the input scalar value.

Data Types: `single` | `double`

`TemperatureScaleFactor` — Scale factor error from temperature (%/°C)
`[0 0 0]` (default) | real scalar in the range [0,100] | real 3-element row vector in the range [0,100]

Scale factor error from temperature in (%/°C), specified as a real scalar or 3-element row vector with values ranging from 0 to 100. Any scalar input is converted into a real 3-element row vector where each element has the input scalar value.

Data Types: `single` | `double`

`AccelerationBias` — Sensor bias from linear acceleration (rad/s)/(m/s^2)
`[0 0 0]` (default) | real scalar | real 3-element row vector

Sensor bias from linear acceleration in (rad/s)/(m/s^2), specified as a real scalar or 3-element row vector. Any scalar input is converted into a real 3-element row vector where each element has the input scalar value.

Data Types: `single` | `double`

Examples

Generate Gyroscope Data from Stationary Inputs

Generate gyroscope data for an imuSensor object from stationary inputs.

Generate a gyroscope parameter object with a maximum sensor reading of 4.363 rad/s and a resolution of 1.332e-4 (rad/s)/LSB. The constant offset bias is 0.349 rad/s. The sensor has a power spectral density of 8.727e-4 $\frac{\text{rad/s}}{\sqrt{\text{Hz}}}$. The bias from temperature is 0.349 rad/s/^0C. The bias from temperature is 0.349 $\left(\text{rad/s}^2\right)/^0\text{C}$. The scale factor error from temperature is 0.2%/^{0}C. The sensor axes are skewed by 2%. The sensor bias from linear acceleration is 0.178e-3 (rad/s)/(m/s^2)

```
params = gyroparams('MeasurementRange',4.363,'Resolution',1.332e-04,'ConstantBias',0.349,'NoiseDe
```

Use a sample rate of 100 Hz spaced out over 1000 samples. Create the imuSensor object using the gyroscope parameter object.

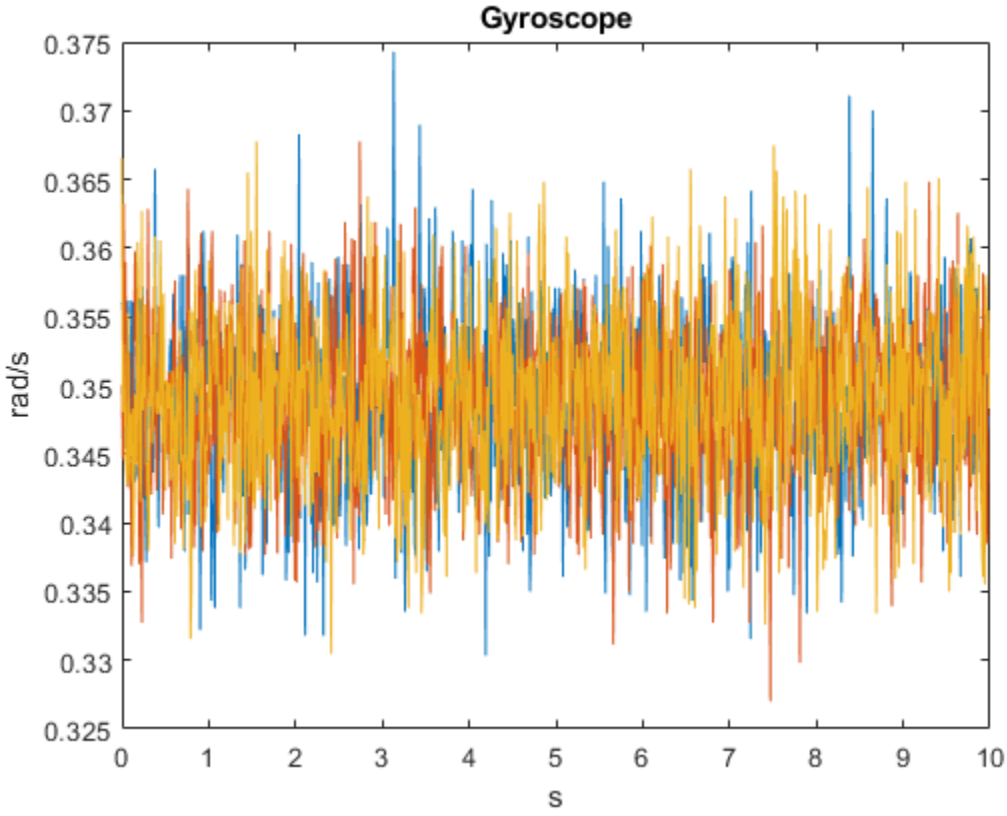
```
Fs = 100;
numSamples = 1000;
t = 0:1/Fs:(numSamples-1)/Fs;

imu = imuSensor('accel-gyro','SampleRate', Fs, 'Gyroscope', params);
```

Generate gyroscope data from the imuSensor object.

```
orient = quaternion.ones(numSamples, 1);
acc = zeros(numSamples, 3);
angvel = zeros(numSamples, 3);

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

Plot the resultant gyroscope data.

```
plot(t, gyroData)
title('Gyroscope')
xlabel('s')
ylabel('rad/s')
```



Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`accelparams` | `magparams` | `imuSensor`

Introduced in R2018b

imufilter

Orientation from accelerometer and gyroscope readings

Description

The `imufilter` System object fuses accelerometer and gyroscope sensor data to estimate device orientation.

To estimate device orientation:

1. Create the `imufilter` object and set its properties.
2. Call the object with arguments, as if it were a function.

To learn more about how System objects work, see What Are System Objects?

Creation

Syntax

```
FUSE = imufilter
FUSE = imufilter('ReferenceFrame',RF)
FUSE = imufilter( ___ ,Name,Value)
```

Description

`FUSE = imufilter` returns an indirect Kalman filter System object, `FUSE`, for fusion of accelerometer and gyroscope data to estimate device orientation. The filter uses a nine-element state vector to track error in the orientation estimate, the gyroscope bias estimate, and the linear acceleration estimate.

`FUSE = imufilter('ReferenceFrame',RF)` returns an `imufilter` filter System object that fuses accelerometer and gyroscope data to estimate device orientation relative to the reference frame `RF`. Specify `RF` as `'NED'` (North-East-Down) or `'ENU'` (East-North-Up). The default value is `'NED'`.

`FUSE = imufilter( ___ ,Name,Value)` sets each property `Name` to the specified `Value`. Unspecified properties have default values.

Example: `FUSE = imufilter('SampleRate',200,'GyroscopeNoise',1e-6)` creates a System object, `FUSE`, with a 200 Hz sample rate and gyroscope noise set to 1e-6 radians per second squared.

Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see System Design in MATLAB Using System Objects.

`SampleRate` — Sample rate of input sensor data (Hz)
`100` (default) | positive finite scalar

Sample rate of the input sensor data in Hz, specified as a positive finite scalar.

Tunable: No

Data Types: `single` | `double` | `uint8` | `uint16` | `uint32` | `uint64` | `int8` | `int16` | `int32` | `int64`

`DecimationFactor` — Decimation factor
`1` (default) | positive integer scalar

Decimation factor by which to reduce the sample rate of the input sensor data, specified as a positive integer scalar.

The number of rows of the inputs, `accelReadings` and `gyroReadings`, must be a multiple of the decimation factor.

Tunable: No

Data Types: `single` | `double` | `uint8` | `uint16` | `uint32` | `uint64` | `int8` | `int16` | `int32` | `int64`

`AccelerometerNoise` — Variance of accelerometer signal noise ($(m/s^2)^2$)
`0.00019247` (default) | positive real scalar

Variance of accelerometer signal noise in $(m/s^2)^2$, specified as a positive real scalar.

Tunable: Yes

Data Types: `single` | `double` | `uint8` | `uint16` | `uint32` | `uint64` | `int8` | `int16` | `int32` | `int64`

`GyroscopeNoise` — Variance of gyroscope signal noise ($(rad/s)^2$)
`9.1385e-5` (default) | positive real scalar

Variance of gyroscope signal noise in $(rad/s)^2$, specified as a positive real scalar.

Tunable: Yes

Data Types: `single` | `double` | `uint8` | `uint16` | `uint32` | `uint64` | `int8` | `int16` | `int32` | `int64`

`GyroscopeDriftNoise` — Variance of gyroscope offset drift ($(rad/s)^2$)
`3.0462e-13` (default) | positive real scalar

Variance of gyroscope offset drift in $(rad/s)^2$, specified as a positive real scalar.

Tunable: Yes

Data Types: `single` | `double` | `uint8` | `uint16` | `uint32` | `uint64` | `int8` | `int16` | `int32` | `int64`

`LinearAccelerationNoise` — Variance of linear acceleration noise ($(m/s^2)^2$)
`0.0096236` (default) | positive real scalar

Variance of linear acceleration noise in $(m/s^2)^2$, specified as a positive real scalar. Linear acceleration is modeled as a lowpass filtered white noise process.

Tunable: Yes

Data Types: `single` | `double` | `uint8` | `uint16` | `uint32` | `uint64` | `int8` | `int16` | `int32` | `int64`

`LinearAcclerationDecayFactor` — Decay factor for linear acceleration drift

`0.5` (default) | scalar in the range [0,1]

Decay factor for linear acceleration drift, specified as a scalar in the range [0,1]. If linear acceleration is changing quickly, set `LinearAccelerationDecayFactor` to a lower value. If linear acceleration changes slowly, set `LinearAccelerationDecayFactor` to a higher value. Linear acceleration drift is modeled as a lowpass-filtered white noise process.

Tunable: Yes

Data Types: `single` | `double` | `uint8` | `uint16` | `uint32` | `uint64` | `int8` | `int16` | `int32` | `int64`

`InitialProcessNoise` — Covariance matrix for process noise

9-by-9 matrix

Covariance matrix for process noise, specified as a 9-by-9 matrix. The default is:

```
  Columns 1 through 6

   0.000006092348396                   0                   0                   0                   0                   0
                   0   0.000006092348396                   0                   0                   0                   0
                   0                   0   0.000006092348396                   0                   0                   0
                   0                   0                   0   0.000076154354947                   0                   0
                   0                   0                   0                   0   0.000076154354947                   0
                   0                   0                   0                   0                   0   0.000076154354947
                   0                   0                   0                   0                   0                   0
                   0                   0                   0                   0                   0                   0
                   0                   0                   0                   0                   0                   0

  Columns 7 through 9

                   0                   0                   0
                   0                   0                   0
                   0                   0                   0
                   0                   0                   0
                   0                   0                   0
                   0                   0                   0
   0.009623610000000                   0                   0
                   0   0.009623610000000                   0
                   0                   0   0.009623610000000
```

The initial process covariance matrix accounts for the error in the process model.

Data Types: `single` | `double` | `uint8` | `uint16` | `uint32` | `uint64` | `int8` | `int16` | `int32` | `int64`

`OrientationFormat` — Output orientation format

`'quaternion'` (default) | `'Rotation matrix'`

Output orientation format, specified as `'quaternion'` or `'Rotation matrix'`. The size of the output depends on the input size, *N*, and the output orientation format:

- `'quaternion'` –– Output is an *N*-by-1 `quaternion`.
- `'Rotation matrix'` –– Output is a 3-by-3-by-*N* rotation matrix.

Data Types: `char` | `string`

Usage

Syntax

```
[orientation,angularVelocity] = FUSE(accelReadings,gyroReadings)
```

Description

`[orientation,angularVelocity] = FUSE(accelReadings,gyroReadings)` fuses accelerometer and gyroscope readings to compute orientation and angular velocity measurements. The algorithm assumes that the device is stationary before the first call.

Input Arguments

`accelReadings` — Accelerometer readings in sensor body coordinate system (m/s²)

N-by-3 matrix

Accelerometer readings in the sensor body coordinate system in m/s^2, specified as an *N*-by-3 matrix. *N* is the number of samples, and the three columns of `accelReadings` represent the [*x y z*] measurements. Accelerometer readings are assumed to correspond to the sample rate specified by the SampleRate property.

Data Types: `single` | `double`

`gyroReadings` — Gyroscope readings in sensor body coordinate system (rad/s)

N-by-3 matrix

Gyroscope readings in the sensor body coordinate system in rad/s, specified as an *N*-by-3 matrix. *N* is the number of samples, and the three columns of `gyroReadings` represent the [*x y z*] measurements. Gyroscope readings are assumed to correspond to the sample rate specified by the SampleRate property.

Data Types: `single` | `double`

Output Arguments

`orientation` — Orientation that rotates quantities from global coordinate system to sensor body coordinate system

M-by-1 vector of quaternions (default) | 3-by-3-by-*M* array

Orientation that can rotate quantities from a global coordinate system to a body coordinate system, returned as quaternions or an array. The size and type of `orientation` depends on whether the OrienationFormat property is set to `'quaternion'` or `'Rotation matrix'`:

- `'quaternion'` –– The output is an *M*-by-1 vector of quaternions, with the same underlying data type as the inputs.
- `'Rotation matrix'` –– The output is a 3-by-3-by-*M* array of rotation matrices the same data type as the inputs.

The number of input samples, *N*, and the DecimationFactor property determine *M*.

You can use `orientation` in a `rotateframe` function to rotate quantities from a global coordinate system to a sensor body coordinate system.

Data Types: `quaternion` | `single` | `double`

`angularVelocity` — Angular velocity in sensor body coordinate system (rad/s)
M-by-3 array (default)

Angular velocity with gyroscope bias removed in the sensor body coordinate system in rad/s, returned as an *M*-by-3 array. The number of input samples, *N*, and the DecimationFactor property determine *M*.

Data Types: `single` | `double`

Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

Specific to `imufilter`

tune	Tune imufilter parameters to reduce estimation error

Common to All System Objects

step	Run System object algorithm
release	Release resources and allow changes to System object property values and input characteristics
reset	Reset internal states of System object

Examples

Estimate Orientation from IMU data

Load the `rpy_9axis` file, which contains recorded accelerometer, gyroscope, and magnetometer sensor data from a device oscillating in pitch (around *y*-axis), then yaw (around *z*-axis), and then roll (around *x*-axis). The file also contains the sample rate of the recording.

```
load 'rpy_9axis.mat' sensorData Fs

accelerometerReadings = sensorData.Acceleration;
gyroscopeReadings = sensorData.AngularVelocity;
```

Create an `imufilter` System object™ with sample rate set to the sample rate of the sensor data. Specify a decimation factor of two to reduce the computational cost of the algorithm.

```
decim = 2;
fuse = imufilter('SampleRate',Fs,'DecimationFactor',decim);
```

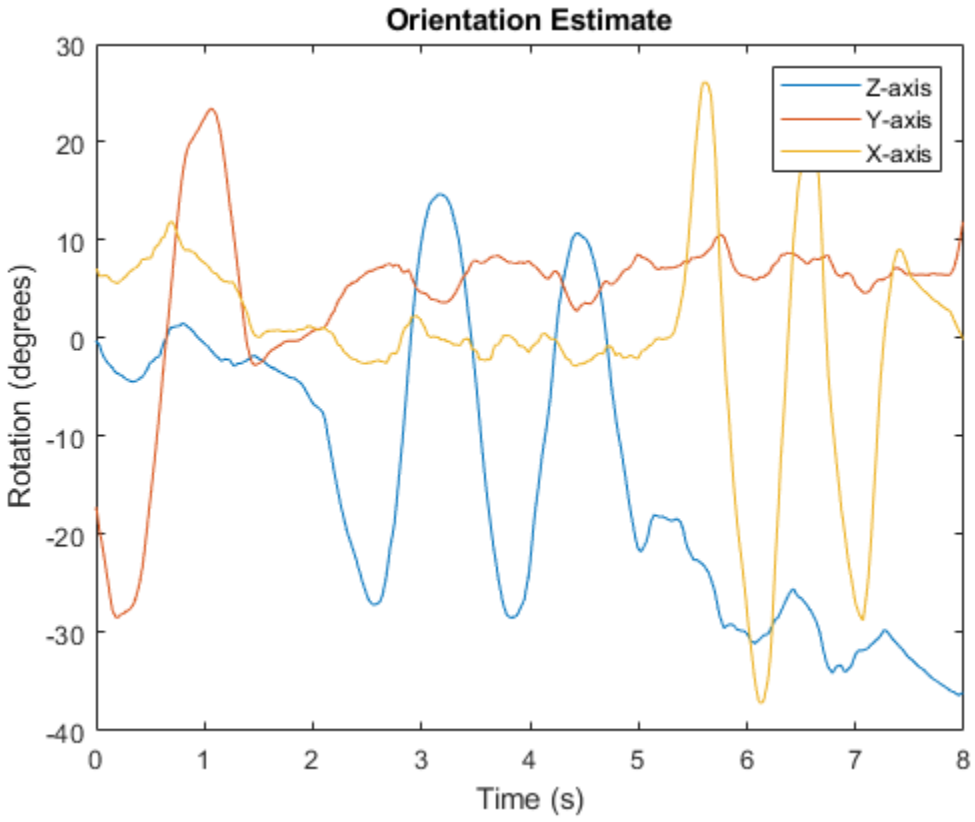
Pass the accelerometer readings and gyroscope readings to the `imufilter` object, `fuse`, to output an estimate of the sensor body orientation over time. By default, the orientation is output as a vector of quaternions.

```
q = fuse(accelerometerReadings,gyroscopeReadings);
```

Orientation is defined by the angular displacement required to rotate a parent coordinate system to a child coordinate system. Plot the orientation in Euler angles in degrees over time.

`imufilter` fusion correctly estimates the change in orientation from an assumed north-facing initial orientation. However, the device's *x*-axis was pointing southward when recorded. To correctly estimate the orientation relative to the true initial orientation or relative to NED, use `ahrsfilter`.

```
time = (0:decim:size(accelerometerReadings,1)-1)/Fs;

plot(time,eulerd(q,'ZYX','frame'))
title('Orientation Estimate')
legend('Z-axis', 'Y-axis', 'X-axis')
xlabel('Time (s)')
ylabel('Rotation (degrees)')
```



Model Tilt Using Gyroscope and Accelerometer Readings

Model a tilting IMU that contains an accelerometer and gyroscope using the `imuSensor` System object™. Use ideal and realistic models to compare the results of orientation tracking using the `imufilter` System object.

Load a struct describing ground-truth motion and a sample rate. The motion struct describes sequential rotations:

1. yaw: 120 degrees over two seconds
2. pitch: 60 degrees over one second

3. roll: 30 degrees over one-half second
4. roll: -30 degrees over one-half second
5. pitch: -60 degrees over one second
6. yaw: -120 degrees over two seconds

In the last stage, the motion struct combines the 1st, 2nd, and 3rd rotations into a single-axis rotation. The acceleration, angular velocity, and orientation are defined in the local NED coordinate system.

```
load y120p60r30.mat motion fs
accNED = motion.Acceleration;
angVelNED = motion.AngularVelocity;
orientationNED = motion.Orientation;

numSamples = size(motion.Orientation,1);
t = (0:(numSamples-1))'/fs;
```

Create an ideal IMU sensor object and a default IMU filter object.

```
IMU = imuSensor('accel-gyro','SampleRate',fs);

aFilter = imufilter('SampleRate',fs);
```

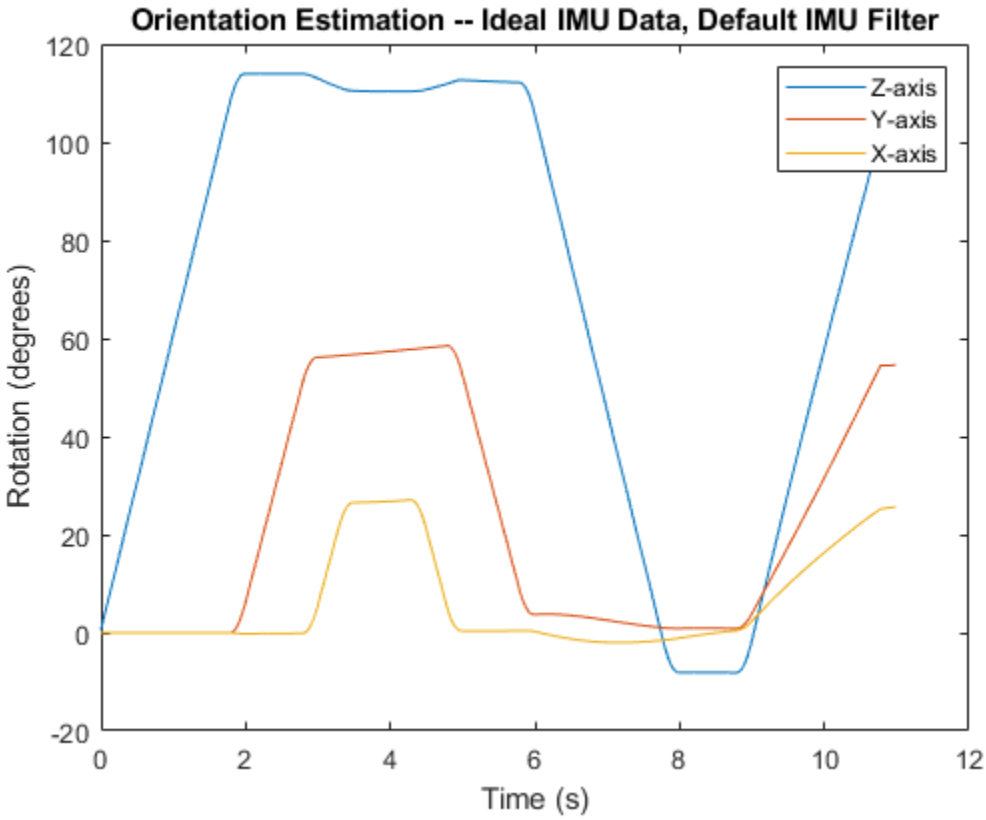
In a loop:

1. Simulate IMU output by feeding the ground-truth motion to the IMU sensor object.
2. Filter the IMU output using the default IMU filter object.

```
orientation = zeros(numSamples,1,'quaternion');
for i = 1:numSamples

    [accelBody,gyroBody] = IMU(accNED(i,:),angVelNED(i,:),orientationNED(i,:));

    orientation(i) = aFilter(accelBody,gyroBody);

end
release(aFilter)
```

Plot the orientation over time.

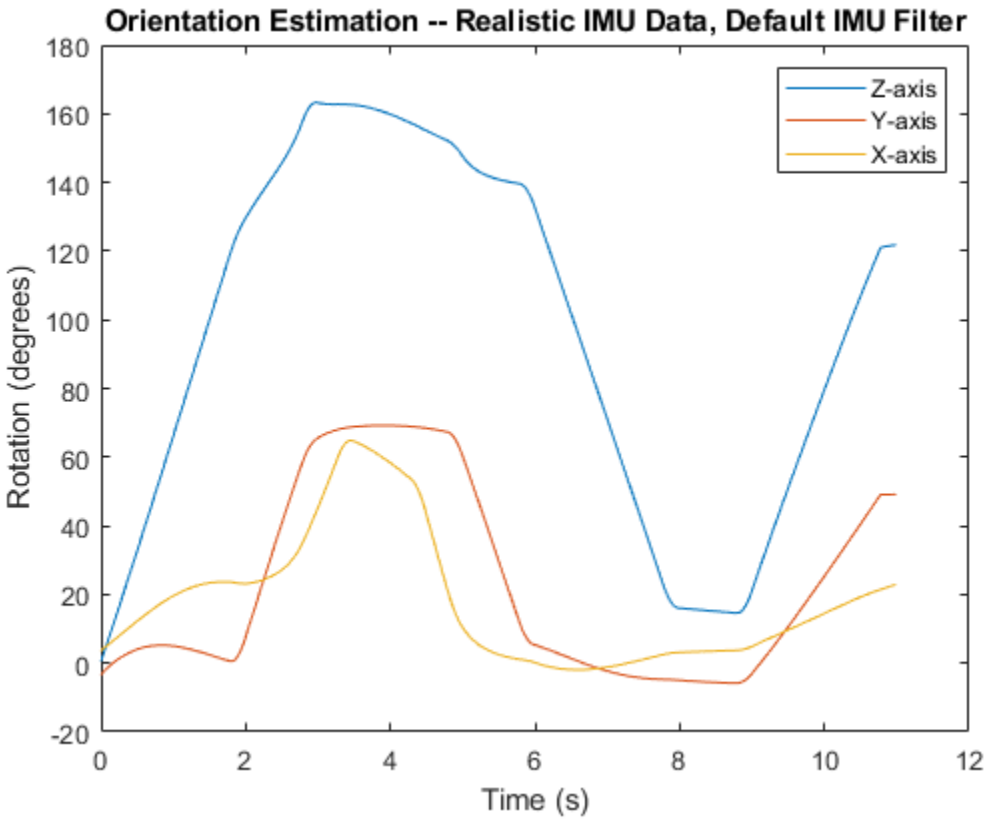
```
figure(1)
plot(t,eulerd(orientation,'ZYX','frame'))
xlabel('Time (s)')
ylabel('Rotation (degrees)')
title('Orientation Estimation -- Ideal IMU Data, Default IMU Filter')
legend('Z-axis','Y-axis','X-axis')
```



Modify properties of your `imuSensor` to model real-world sensors. Run the loop again and plot the orientation estimate over time.

```
IMU.Accelerometer = accelparams( ...
    'MeasurementRange',19.62, ...
    'Resolution',0.00059875, ...
    'ConstantBias',0.4905, ...
    'AxesMisalignment',2, ...
    'NoiseDensity',0.003924, ...
    'BiasInstability',0, ...
    'TemperatureBias', [0.34335 0.34335 0.5886], ...
    'TemperatureScaleFactor',0.02);
IMU.Gyroscope = gyroparams( ...
    'MeasurementRange',4.3633, ...
    'Resolution',0.00013323, ...
    'AxesMisalignment',2, ...
    'NoiseDensity',8.7266e-05, ...
    'TemperatureBias',0.34907, ...
    'TemperatureScaleFactor',0.02, ...
    'AccelerationBias',0.00017809, ...
    'ConstantBias',[0.3491,0.5,0]);

orientationDefault = zeros(numSamples,1,'quaternion');
for i = 1:numSamples

    [accelBody,gyroBody] = IMU(accNED(i,:),angVelNED(i,:),orientationNED(i,:));
```

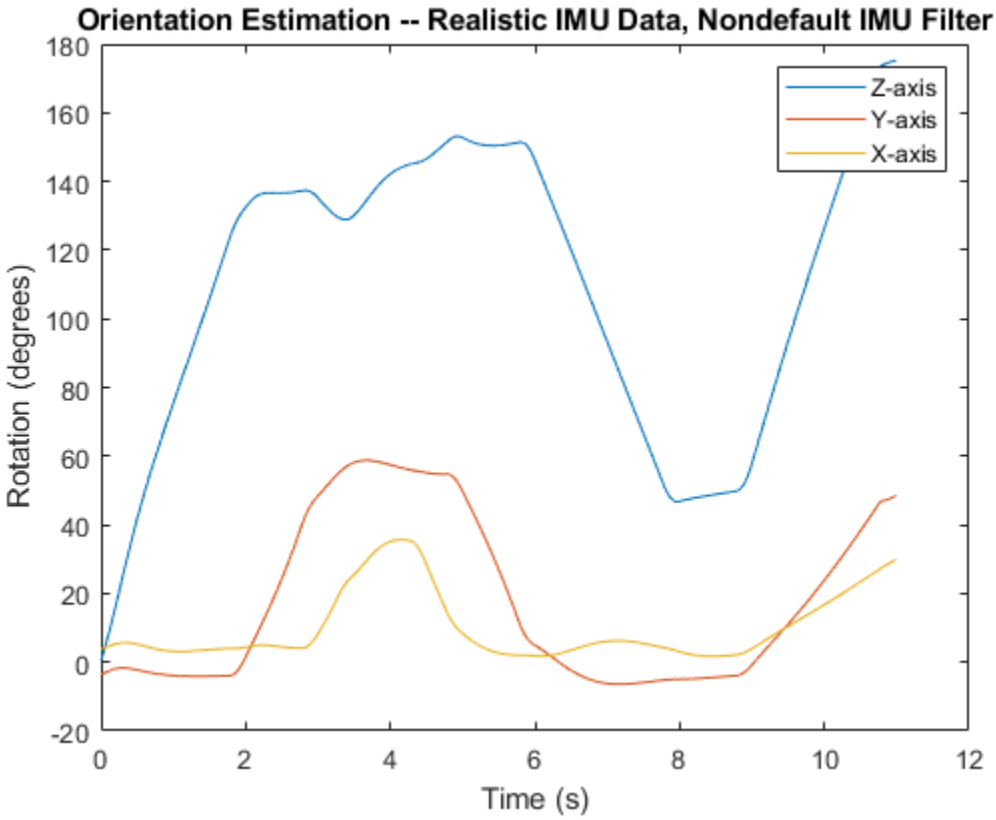
```
    orientationDefault(i) = aFilter(accelBody,gyroBody);

end
release(aFilter)

figure(2)
plot(t,eulerd(orientationDefault,'ZYX','frame'))
xlabel('Time (s)')
ylabel('Rotation (degrees)')
title('Orientation Estimation -- Realistic IMU Data, Default IMU Filter')
legend('Z-axis','Y-axis','X-axis')
```



The ability of the `imufilter` to track the ground-truth data is significantly reduced when modeling a realistic IMU. To improve performance, modify properties of your `imufilter` object. These values were determined empirically. Run the loop again and plot the orientation estimate over time.

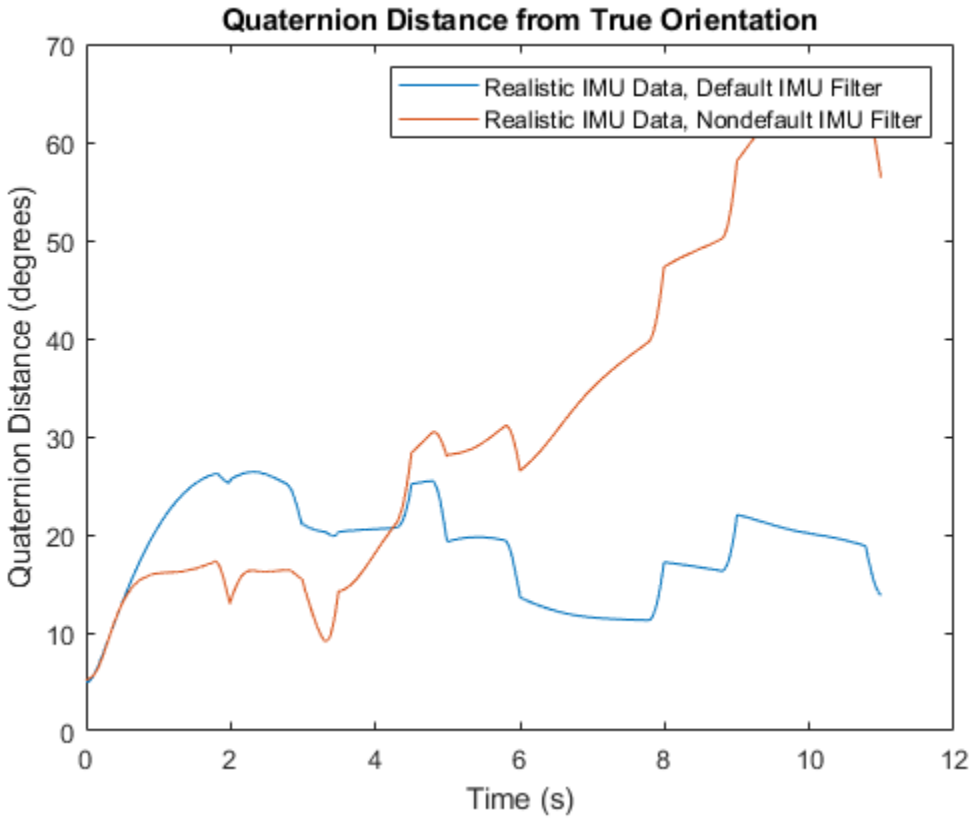
```
aFilter.GyroscopeNoise          = 7.6154e-7;
aFilter.AccelerometerNoise      = 0.0015398;
aFilter.GyroscopeDriftNoise     = 3.0462e-12;
aFilter.LinearAccelerationNoise = 0.00096236;
aFilter.InitialProcessNoise     = aFilter.InitialProcessNoise*10;

orientationNondefault = zeros(numSamples,1,'quaternion');
for i = 1:numSamples
    [accelBody,gyroBody] = IMU(accNED(i,:),angVelNED(i,:),orientationNED(i,:));

    orientationNondefault(i) = aFilter(accelBody,gyroBody);
```

```
end
release(aFilter)

figure(3)
plot(t,eulerd(orientationNondefault,'ZYX','frame'))
xlabel('Time (s)')
ylabel('Rotation (degrees)')
title('Orientation Estimation -- Realistic IMU Data, Nondefault IMU Filter')
legend('Z-axis','Y-axis','X-axis')
```



To quantify the improved performance of the modified `imufilter`, plot the quaternion distance between the ground-truth motion and the orientation as returned by the `imufilter` with default and nondefault properties.

```
qDistDefault = rad2deg(dist(orientationNED,orientationDefault));
qDistNondefault = rad2deg(dist(orientationNED,orientationNondefault));

figure(4)
plot(t,[qDistDefault,qDistNondefault])
title('Quaternion Distance from True Orientation')
legend('Realistic IMU Data, Default IMU Filter', ...
       'Realistic IMU Data, Nondefault IMU Filter')
xlabel('Time (s)')
ylabel('Quaternion Distance (degrees)')
```



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);
```

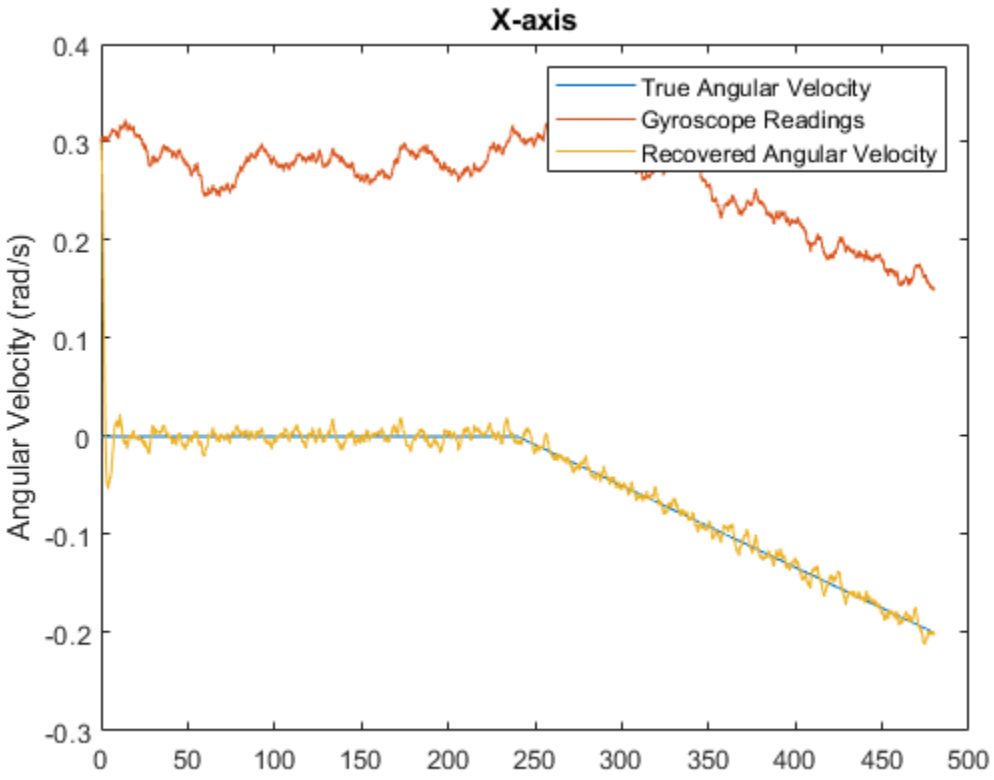
Create an `imufilter` System object, `fuse`. Call `fuse` with the modeled accelerometer readings and gyroscope readings.

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

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

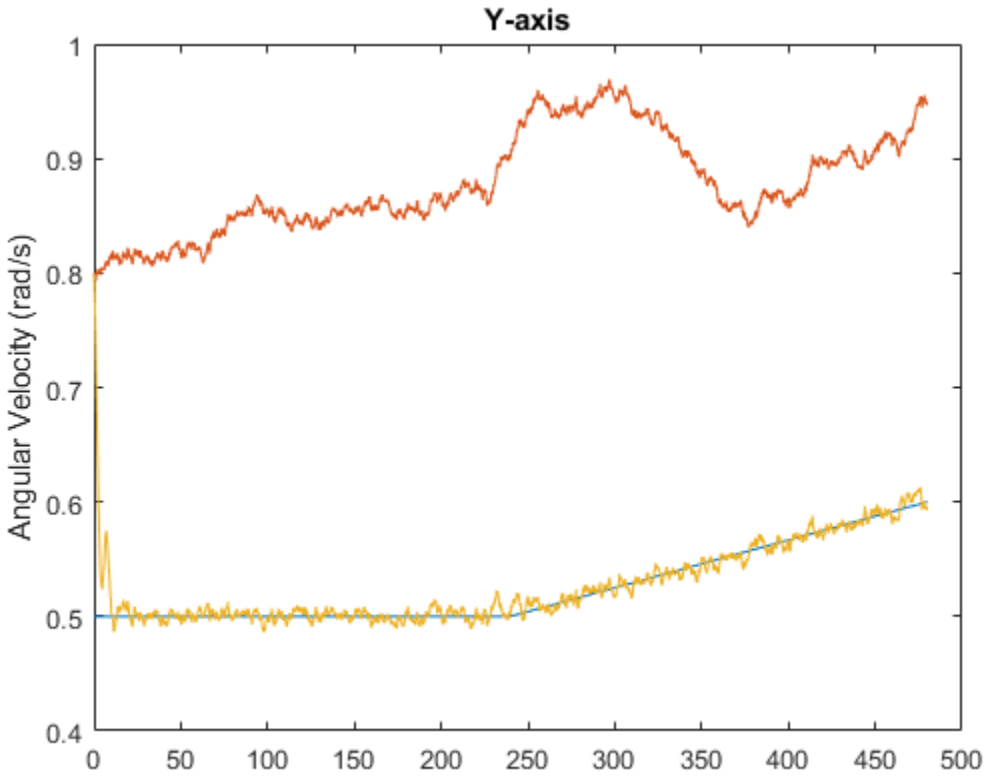
Plot the ground-truth angular velocity, the gyroscope readings, and the recovered angular velocity for each axis.

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

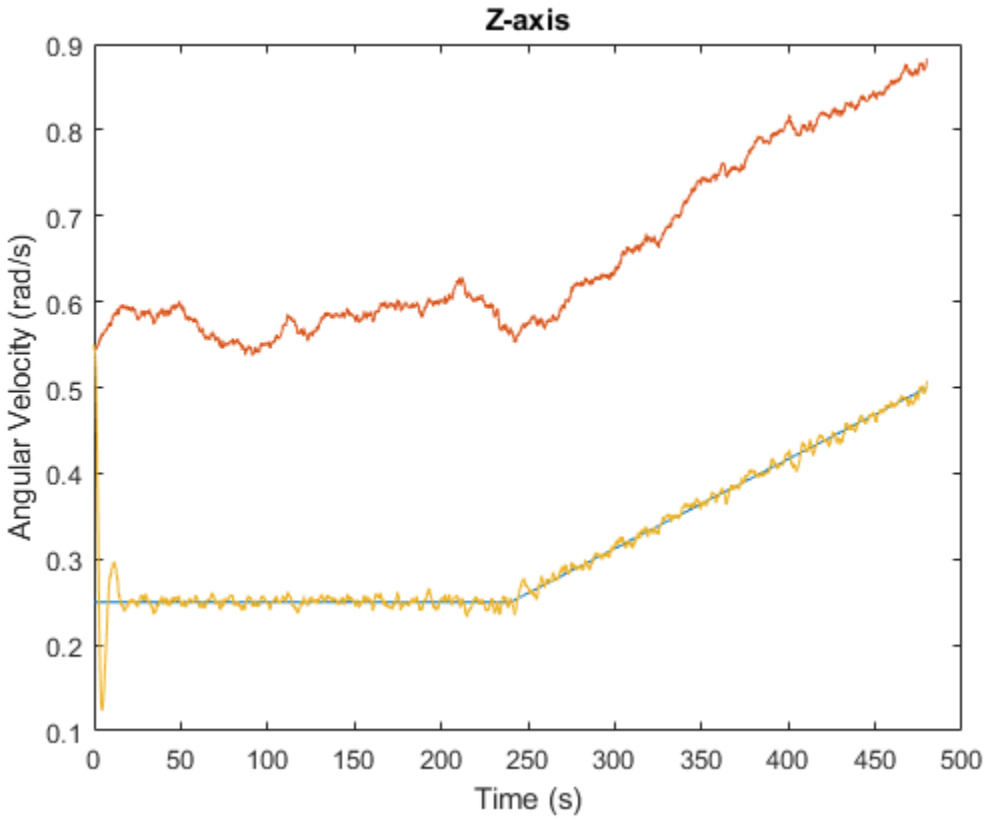
```
time = (0:numSamples-1)'/fs;

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)')
```



Algorithms

Note: The following algorithm only applies to an NED reference frame.

The `imufilter` uses the six-axis Kalman filter structure described in [1]. The algorithm attempts to track the errors in orientation, gyroscope offset, and linear acceleration to output the final orientation and angular velocity. Instead of tracking the orientation directly, the indirect Kalman filter models the error process, x, with a recursive update:

$$x_k = \begin{bmatrix} \theta_k \\ b_k \\ a_k \end{bmatrix} = F_k \begin{bmatrix} \theta_{k-1} \\ b_{k-1} \\ a_{k-1} \end{bmatrix} + w_k$$

where x_k is a 9-by-1 vector consisting of:

- θ_k –- 3-by-1 orientation error vector, in degrees, at time k
- b_k –- 3-by-1 gyroscope zero angular rate bias vector, in deg/s, at time k
- a_k –- 3-by-1 acceleration error vector measured in the sensor frame, in g, at time k
- w_k –- 9-by-1 additive noise vector
- F_k –- state transition model

Because x_k is defined as the error process, the *a priori* estimate is always zero, and therefore the state transition model, F_k, is zero. This insight results in the following reduction of the standard Kalman equations:

Standard Kalman equations:

$$x_k^- = F_k x_{k-1}^+$$
$$P_k^- = F_k P_{k-1}^+ F_k^T + Q_k$$

$$y_k = z_k - H_k x_k^-$$
$$S_k = R_k + H_k P_k^- H_k^T$$
$$K_k = P_k^- H_k^T (S_k)^{-1}$$
$$x_k^+ = x_k^- + K_k y_k$$
$$P_k^+ = P_k^- - K_k H_k P_k^-$$

Kalman equations used in this algorithm:

$$x_k^- = 0$$
$$P_k^- = Q_k$$

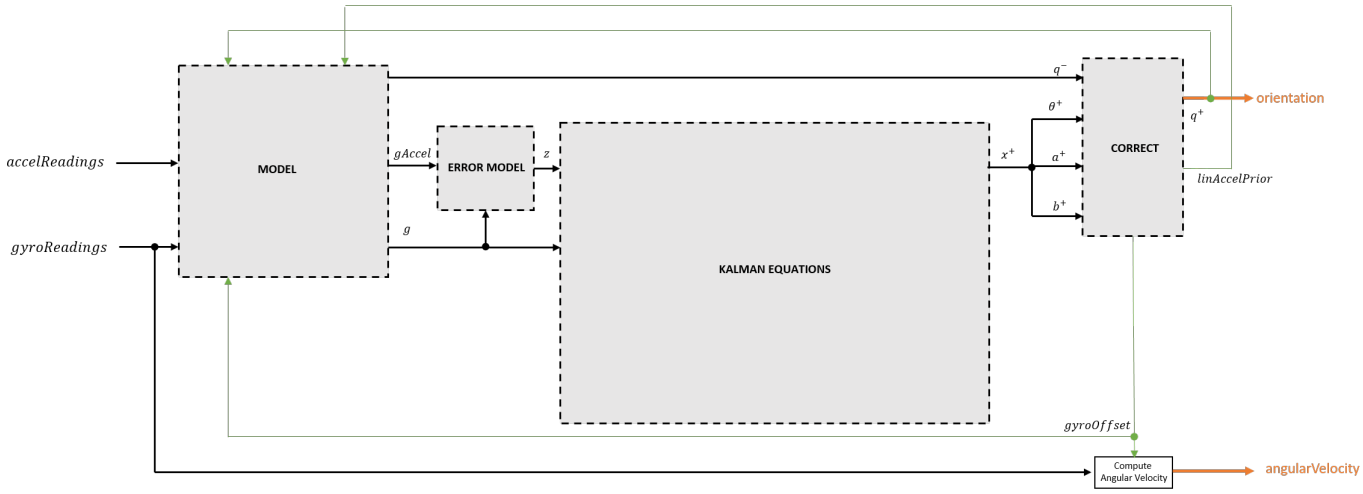
$$y_k = z_k$$
$$S_k = R_k + H_k P_k^- H_k^T$$
$$K_k = P_k^- H_k^T (S_k)^{-1}$$
$$x_k^+ = K_k y_k$$
$$P_k^+ = P_k^- - K_k H_k P_k^-$$

where

- x_k^- –– predicted (*a priori*) state estimate; the error process
- P_k^- –– predicted (*a priori*) estimate covariance
- y_k –– innovation
- S_k –– innovation covariance
- K_k –– Kalman gain
- x_k^+ –– updated (*a posteriori*) state estimate
- P_k^+ –– updated (*a posteriori*) estimate covariance

k represents the iteration, the superscript $^+$ represents an *a posteriori* estimate, and the superscript $^-$ represents an *a priori* estimate.

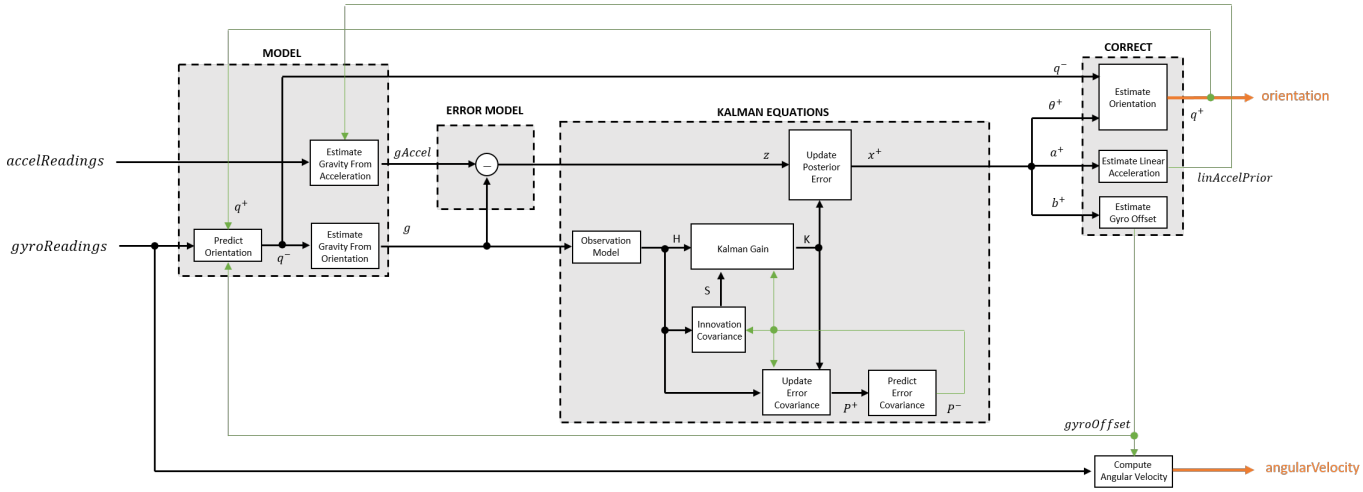
The graphic and following steps describe a single frame-based iteration through the algorithm.



Before the first iteration, the `accelReadings` and `gyroReadings` inputs are chunked into 1-by-3 frames and `DecimationFactor`-by-3 frames, respectively. The algorithm uses the most current accelerometer readings corresponding to the chunk of gyroscope readings.

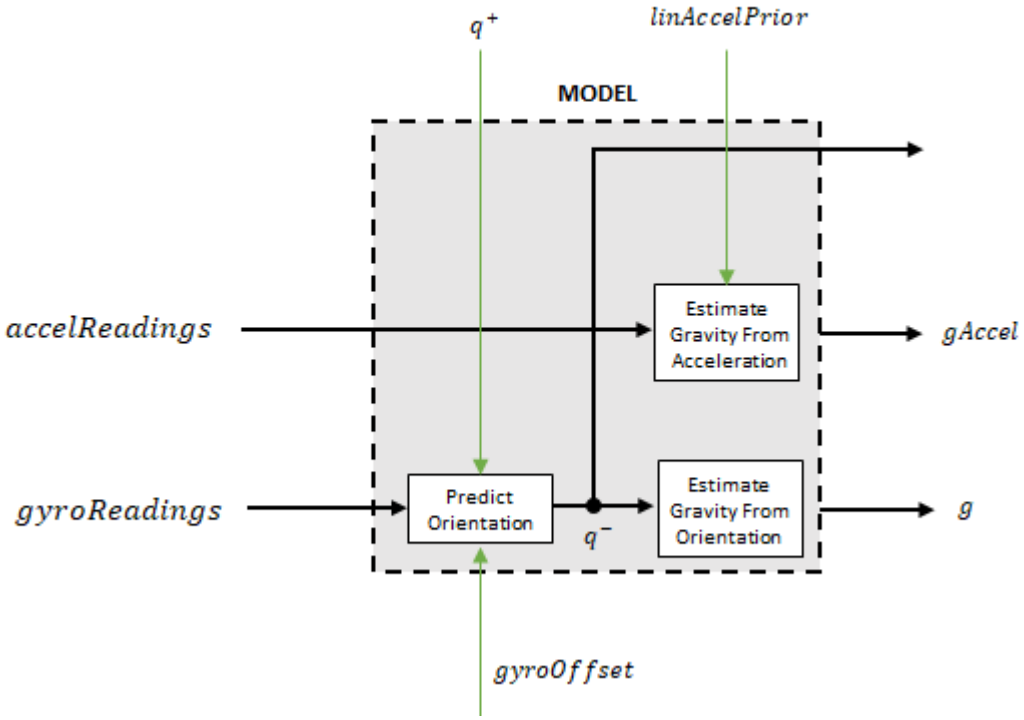
Detailed Overview

Step through the algorithm for an explanation of each stage of the detailed overview.



Model

The algorithm models acceleration and angular change as linear processes.



Predict Orientation

The orientation for the current frame is predicted by first estimating the angular change from the previous frame:

$$\Delta\varphi_{N\times 3} = \frac{(gyroReadings_{N\times 3} - gyroOffset_{1\times 3})}{fs}$$

where N is the decimation factor specified by the `DecimationFactor` property, and fs is the sample rate specified by the `SampleRate` property.

The angular change is converted into quaternions using the `rotvec` `quaternion` construction syntax:

$$\Delta Q_{N\times 1} = \text{quaternion}(\Delta\varphi_{N\times 3}, 'rotvec')$$

The previous orientation estimate is updated by rotating it by ΔQ:

$$q_{1\times 1}^{-} = \left(q_{1\times 1}^{+}\right)\left(\prod_{n=1}^{N}\Delta Q_n\right)$$

During the first iteration, the orientation estimate, q^-, is initialized by `ecompass` with an assumption that the x-axis points north.

Estimate Gravity from Orientation

The gravity vector is interpreted as the third column of the quaternion, q^-, in rotation matrix form:

$$g_{1 \times 3} = (rPrior(:,3))^T$$

See `ecompass` for an explanation of why the third column of *rPrior* can be interpreted as the gravity vector.

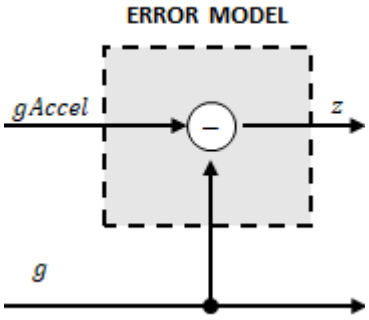
Estimate Gravity from Acceleration

A second gravity vector estimation is made by subtracting the decayed linear acceleration estimate of the previous iteration from the accelerometer readings:

$$gAccel_{1 \times 3} = accelReadings_{1 \times 3} - linAccelprior_{1 \times 3}$$

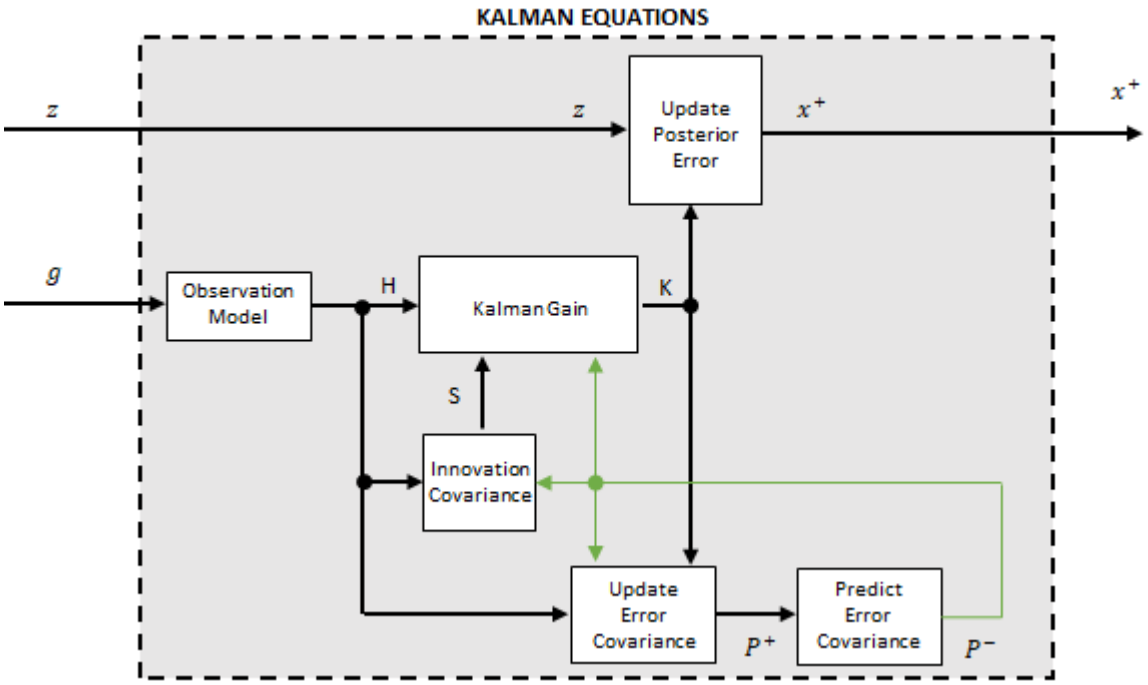
Error Model

The error model is the difference between the gravity estimate from the accelerometer readings and the gravity estimate from the gyroscope readings: $z = g - gAccel$.



Kalman Equations

The Kalman equations use the gravity estimate derived from the gyroscope readings, g, and the observation of the error process, z, to update the Kalman gain and intermediary covariance matrices. The Kalman gain is applied to the error signal, z, to output an *a posteriori* error estimate, x^+.



Observation Model

The observation model maps the 1-by-3 observed state, g, into the 3-by-9 true state, H.

The observation model is constructed as:

$$H_{3\times 9} = \begin{bmatrix} 0 & g_z & -g_y & 0 & -\kappa g_z & \kappa g_y & 1 & 0 & 0 \\ -g_z & 0 & g_x & \kappa g_z & 0 & -\kappa g_x & 0 & 1 & 0 \\ g_y & -g_x & 0 & -\kappa g_y & \kappa g_x & 0 & 0 & 0 & 1 \end{bmatrix}$$

where g_x, g_y, and g_z are the x-, y-, and z-elements of the gravity vector estimated from the orientation, respectively. κ is a constant determined by the SampleRate and DecimationFactor properties: κ = `DecimationFactor/SampleRate`.

See sections 7.3 and 7.4 of [1] for a derivation of the observation model.

Innovation Covariance

The innovation covariance is a 3-by-3 matrix used to track the variability in the measurements. The innovation covariance matrix is calculated as:

$$S_{3x3} = R_{3x3} + (H_{3x9})\left(P^{-}_{9x9}\right)(H_{3x9})^{T}$$

where

- H is the observation model matrix
- P^{-} is the predicted (*a priori*) estimate of the covariance of the observation model calculated in the previous iteration

- R is the covariance of the observation model noise, calculated as:

$$R_{3\times 3} = (\lambda + \xi + \kappa(\beta + \eta))\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$

  The following properties define the observation model noise variance:

  - κ -- (DecimationFactor/SampleRate)^2
  - β -- GyroscopeDriftNoise
  - η -- GyroscopeNoise
  - λ -- AccelerometerNoise
  - ξ -- LinearAccelerationNoise

Update Error Estimate Covariance

The error estimate covariance is a 9-by-9 matrix used to track the variability in the state.

The error estimate covariance matrix is updated as:

$$P^{+}_{9\times 9} = P^{-}_{9\times 9} - (K_{9\times 3})(H_{3\times 9})\left(P^{-}_{9\times 9}\right)$$

where K is the Kalman gain, H is the measurement matrix, and P^{-} is the error estimate covariance calculated during the previous iteration.

Predict Error Estimate Covariance

The error estimate covariance is a 9-by-9 matrix used to track the variability in the state. The *a priori* error estimate covariance, P^{-}, is set to the process noise covariance, Q, determined during the previous iteration. Q is calculated as a function of the *a posteriori* error estimate covariance, P^{+}. When calculating Q, the cross-correlation terms are assumed to be negligible compared to the autocorrelation terms, and are set to zero:

$$Q = \begin{bmatrix} P^+(1) + \kappa^2 P^+(31) + \beta + \eta & 0 & 0 & -\kappa\left(P^+(31) + \beta\right) & 0 \\ 0 & P^+(11) + \kappa^2 P^+(41) + \beta + \eta & 0 & 0 & -\kappa(P^+(41) + \\ 0 & 0 & P^+(21) + \kappa^2 P^+(51) + \beta + \eta & 0 & 0 \\ -\kappa\left(P^+(31) + \beta\right) & 0 & 0 & P^+(31) + \beta & 0 \\ 0 & -\kappa(P^+(41) + \beta) & 0 & 0 & P^+(41) + \beta \\ 0 & 0 & -\kappa(P^+(51) + \beta) & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

where

- P^+ –– is the updated (*a posteriori*) error estimate covariance
- κ –– DecimationFactor/SampleRate
- β –– GyroscopeDriftNoise
- η –– GyroscopeNoise
- ν –– LinearAcclerationDecayFactor
- ξ –– LinearAccelerationNoise

See section 10.1 of [1] for a derivation of the terms of the process error matrix.

Kalman Gain

The Kalman gain matrix is a 9-by-3 matrix used to weight the innovation. In this algorithm, the innovation is interpreted as the error process, z.

The Kalman gain matrix is constructed as:

$$K_{9\times 3} = \left(P^{-}_{9\times 9}\right)\left(H_{3\times 9}\right)^{T}\left(\left(S_{3\times 3}\right)^{T}\right)^{-1}$$

where

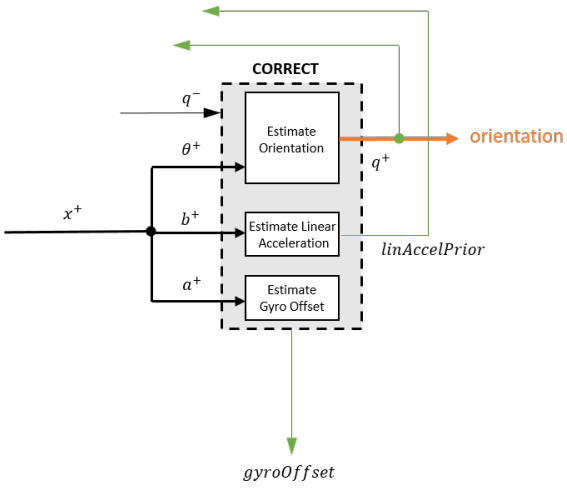
- P^{-} –– predicted error covariance
- H –– observation model
- S –– innovation covariance

Update a Posteriori Error

The *a posterior* error estimate is determined by combining the Kalman gain matrix with the error in the gravity vector estimations:

$$x_{9\times 1} = \left(K_{9\times 3}\right)\left(z_{1\times 3}\right)^{T}$$

Correct



Estimate Orientation

The orientation estimate is updated by multiplying the previous estimation by the error:

$$q^{+} = \left(q^{-}\right)\left(\theta^{+}\right)$$

Estimate Linear Acceleration

The linear acceleration estimation is updated by decaying the linear acceleration estimation from the previous iteration and subtracting the error:

$$linAccelPrior = (linAccelPrior_{k-1})\nu - b^{+}$$

where

- ν –- LinearAcclerationDecayFactor

Estimate Gyroscope Offset

The gyroscope offset estimation is updated by subtracting the gyroscope offset error from the gyroscope offset from the previous iteration:

$$gyroOffset = gyroOffset_{k-1} - a^{+}$$

Compute Angular Velocity

To estimate angular velocity, the frame of `gyroReadings` are averaged and the gyroscope offset computed in the previous iteration is subtracted:

$$angularVelocity_{1\times 3} = \frac{\sum gyroReadings_{N\times 3}}{N} - gyroOffset_{1\times 3}$$

where N is the decimation factor specified by the `DecimationFactor` property.

The gyroscope offset estimation is initialized to zeros for the first iteration.

References

[1] Open Source Sensor Fusion. https://github.com/memsindustrygroup/Open-Source-Sensor-Fusion/tree/master/docs

[2] Roetenberg, D., H.J. Luinge, C.T.M. Baten, and P.H. Veltink. "Compensation of Magnetic Disturbances Improves Inertial and Magnetic Sensing of Human Body Segment Orientation." *IEEE Transactions on Neural Systems and Rehabilitation Engineering*. Vol. 13. Issue 3, 2005, pp. 395-405.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

`ecompass` | `ahrsfilter` | `imuSensor` | `gpsSensor`

Introduced in R2018b

tune

Tune `imufilter` parameters to reduce estimation error

Syntax

```
tune(filter,sensorData,groundTruth)
tune( ___ ,config)
```

Description

`tune(filter,sensorData,groundTruth)` adjusts the properties of the `imufilter` filter object, `filter`, to reduce the root-mean-squared (RMS) quaternion distance error between the fused sensor data and the ground truth. The function fuses the sensor data to estimate the orientation, which is compared to the orientation in the ground truth. The function uses the property values in the filter as the initial estimate for the optimization algorithm.

`tune( ___ ,config)` specifies the tuning configuration based on a `tunerconfig` object, `config`.

Examples

Tune `imufilter` to Optimize Orientation Estimate

Load recorded sensor data and ground truth data.

```
ld = load('imufilterTuneData.mat');
qTrue = ld.groundTruth.Orientation; % true orientation
```

Create an `imufilter` object and fuse the filter with the sensor data.

```
fuse = imufilter;
qEstUntuned = fuse(ld.sensorData.Accelerometer, ...
    ld.sensorData.Gyroscope);
```

Create a `tunerconfig` object and tune the imufilter to improve the orientation estimate.

```
cfg = tunerconfig('imufilter');
tune(fuse, ld.sensorData, ld.groundTruth, cfg);
```

```
    Iteration    Parameter                       Metric
    _________    _________                       ______
    1            AccelerometerNoise              0.1149
    1            GyroscopeNoise                  0.1146
    1            GyroscopeDriftNoise             0.1146
    1            LinearAccelerationNoise         0.1122
    1            LinearAccelerationDecayFactor   0.1103
    2            AccelerometerNoise              0.1102
    2            GyroscopeNoise                  0.1098
    2            GyroscopeDriftNoise             0.1098
    2            LinearAccelerationNoise         0.1070
    2            LinearAccelerationDecayFactor   0.1053
    3            AccelerometerNoise              0.1053
```

```
3            GyroscopeNoise                 0.1048
3            GyroscopeDriftNoise            0.1048
3            LinearAccelerationNoise        0.1016
3            LinearAccelerationDecayFactor  0.1002
4            AccelerometerNoise             0.1001
4            GyroscopeNoise                 0.0996
4            GyroscopeDriftNoise            0.0996
4            LinearAccelerationNoise        0.0962
4            LinearAccelerationDecayFactor  0.0950
5            AccelerometerNoise             0.0950
5            GyroscopeNoise                 0.0943
5            GyroscopeDriftNoise            0.0943
5            LinearAccelerationNoise        0.0910
5            LinearAccelerationDecayFactor  0.0901
6            AccelerometerNoise             0.0900
6            GyroscopeNoise                 0.0893
6            GyroscopeDriftNoise            0.0893
6            LinearAccelerationNoise        0.0862
6            LinearAccelerationDecayFactor  0.0855
7            AccelerometerNoise             0.0855
7            GyroscopeNoise                 0.0848
7            GyroscopeDriftNoise            0.0848
7            LinearAccelerationNoise        0.0822
7            LinearAccelerationDecayFactor  0.0818
8            AccelerometerNoise             0.0817
8            GyroscopeNoise                 0.0811
8            GyroscopeDriftNoise            0.0811
8            LinearAccelerationNoise        0.0791
8            LinearAccelerationDecayFactor  0.0789
9            AccelerometerNoise             0.0788
9            GyroscopeNoise                 0.0782
9            GyroscopeDriftNoise            0.0782
9            LinearAccelerationNoise        0.0769
9            LinearAccelerationDecayFactor  0.0768
10           AccelerometerNoise             0.0768
10           GyroscopeNoise                 0.0762
10           GyroscopeDriftNoise            0.0762
10           LinearAccelerationNoise        0.0754
10           LinearAccelerationDecayFactor  0.0753
11           AccelerometerNoise             0.0753
11           GyroscopeNoise                 0.0747
11           GyroscopeDriftNoise            0.0747
11           LinearAccelerationNoise        0.0741
11           LinearAccelerationDecayFactor  0.0740
12           AccelerometerNoise             0.0740
12           GyroscopeNoise                 0.0734
12           GyroscopeDriftNoise            0.0734
12           LinearAccelerationNoise        0.0728
12           LinearAccelerationDecayFactor  0.0728
13           AccelerometerNoise             0.0728
13           GyroscopeNoise                 0.0721
13           GyroscopeDriftNoise            0.0721
13           LinearAccelerationNoise        0.0715
13           LinearAccelerationDecayFactor  0.0715
14           AccelerometerNoise             0.0715
14           GyroscopeNoise                 0.0706
14           GyroscopeDriftNoise            0.0706
14           LinearAccelerationNoise        0.0700
```

```
    14          LinearAccelerationDecayFactor    0.0700
    15          AccelerometerNoise               0.0700
    15          GyroscopeNoise                   0.0690
    15          GyroscopeDriftNoise              0.0690
    15          LinearAccelerationNoise          0.0684
    15          LinearAccelerationDecayFactor    0.0684
    16          AccelerometerNoise               0.0684
    16          GyroscopeNoise                   0.0672
    16          GyroscopeDriftNoise              0.0672
    16          LinearAccelerationNoise          0.0668
    16          LinearAccelerationDecayFactor    0.0667
    17          AccelerometerNoise               0.0667
    17          GyroscopeNoise                   0.0655
    17          GyroscopeDriftNoise              0.0655
    17          LinearAccelerationNoise          0.0654
    17          LinearAccelerationDecayFactor    0.0654
    18          AccelerometerNoise               0.0654
    18          GyroscopeNoise                   0.0641
    18          GyroscopeDriftNoise              0.0641
    18          LinearAccelerationNoise          0.0640
    18          LinearAccelerationDecayFactor    0.0639
    19          AccelerometerNoise               0.0639
    19          GyroscopeNoise                   0.0627
    19          GyroscopeDriftNoise              0.0627
    19          LinearAccelerationNoise          0.0627
    19          LinearAccelerationDecayFactor    0.0624
    20          AccelerometerNoise               0.0624
    20          GyroscopeNoise                   0.0614
    20          GyroscopeDriftNoise              0.0614
    20          LinearAccelerationNoise          0.0613
    20          LinearAccelerationDecayFactor    0.0613
```

Fuse the sensor data again using the tuned filter.

```
qEstTuned = fuse(ld.sensorData.Accelerometer, ...
    ld.sensorData.Gyroscope);
```

Compare the tuned and untuned filter RMS error performances.

```
dUntuned = rad2deg(dist(qEstUntuned, qTrue));
dTuned = rad2deg(dist(qEstTuned, qTrue));
rmsUntuned = sqrt(mean(dUntuned.^2))
```

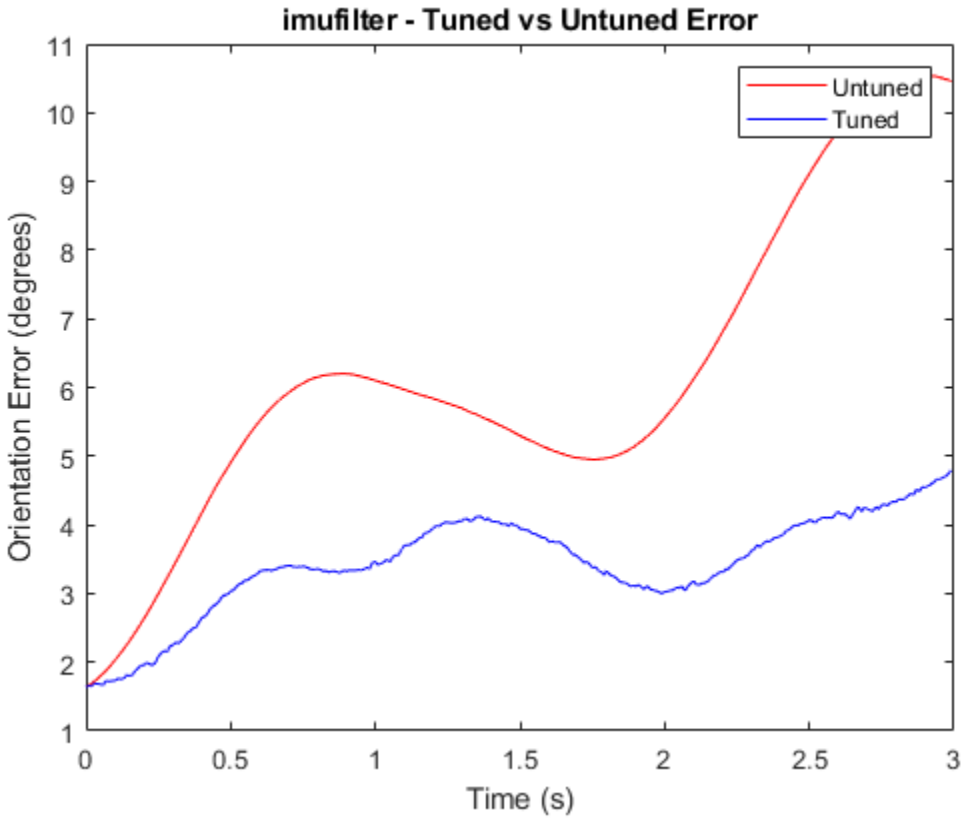
```
rmsUntuned = 6.5864
```

```
rmsTuned = sqrt(mean(dTuned.^2))
```

```
rmsTuned = 3.5098
```

Visualize the results.

```
N = numel(dUntuned);
t = (0:N-1)./ fuse.SampleRate;
plot(t, dUntuned, 'r', t, dTuned, 'b');
legend('Untuned', 'Tuned');
title('imufilter - Tuned vs Untuned Error')
xlabel('Time (s)');
ylabel('Orientation Error (degrees)');
```



Input Arguments

`filter` — Filter object

`imufilter` object

Filter object, specified as an `imufilter` object.

`sensorData` — Sensor data

`table`

Sensor data, specified as a `table`. In each row, the sensor data is specified as:

- `Accelerometer` — Accelerometer data, specified as a 1-by-3 vector of scalars in m^2/s.
- `Gyroscope` — Gyroscope data, specified as a 1-by-3 vector of scalars in rad/s.

If you set the `Cost` property of the tuner configuration input, `config`, to `Custom`, then you can use other data types for the `sensorData` input based on your choice.

`groundTruth` — Ground truth data

`timetable`

Ground truth data, specified as a `table`. The table has only one column of `Orientation` data. In each row, the orientation is specified as a `quaternion` object or a 3-by-3 rotation matrix.

The function processes each row of the `sensorData` and `groundTruth` tables sequentially to calculate the state estimate and RMS error from the ground truth. Each row of the `sensorData` and the `groundTruth` tables must correspond to each other.

If you set the `Cost` property of the tuner configuration input, `config`, to `Custom`, then you can use other data types for the `groundTruth` input based on your choice.

`config` — Tuner configuration
`tunerconfig` object

Tuner configuration, specified as a `tunerconfig` object.

References

[1] Abbeel, P., Coates, A., Montemerlo, M., Ng, A.Y. and Thrun, S. Discriminative Training of Kalman Filters. In *Robotics: Science and systems*, Vol. 2, pp. 1, 2005.

See Also

Introduced in R2020b

imuSensor

IMU simulation model

Description

The `imuSensor` System object models receiving data from an inertial measurement unit (IMU).

To model an IMU:

1. Create the `imuSensor` object and set its properties.
2. Call the object with arguments, as if it were a function.

To learn more about how System objects work, see What Are System Objects?

Creation

Syntax

```
IMU = imuSensor
IMU = imuSensor('accel-gyro')
IMU = imuSensor('accel-mag')
IMU = imuSensor('accel-gyro-mag')
IMU = imuSensor(___,'ReferenceFrame',RF)
IMU = imuSensor(___,Name,Value)
```

Description

`IMU = imuSensor` returns a System object, `IMU`, that computes an inertial measurement unit reading based on an inertial input signal. `IMU` has an ideal accelerometer and gyroscope.

`IMU = imuSensor('accel-gyro')` returns an `imuSensor` System object with an ideal accelerometer and gyroscope. `imuSensor` and `imuSensor('accel-gyro')` are equivalent creation syntaxes.

`IMU = imuSensor('accel-mag')` returns an `imuSensor` System object with an ideal accelerometer and magnetometer.

`IMU = imuSensor('accel-gyro-mag')` returns an `imuSensor` System object with an ideal accelerometer, gyroscope, and magnetometer.

`IMU = imuSensor(___,'ReferenceFrame',RF)` returns an `imuSensor` System object that computes an inertial measurement unit reading relative to the reference frame `RF`. Specify `RF` as `'NED'` (North-East-Down) or `'ENU'` (East-North-Up). The default value is `'NED'`.

`IMU = imuSensor(___,Name,Value)` sets each property `Name` to the specified `Value`. Unspecified properties have default values. This syntax can be used in combination with any of the previous input arguments.

Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see System Design in MATLAB Using System Objects.

`IMUType` — Type of inertial measurement unit

`'accel-gyro'` (default) | `'accel-mag'` | `'accel-gyro-mag'`

Type of inertial measurement unit, specified as a `'accel-gyro'`, `'accel-mag'`, or `'accel-gyro-mag'`.

The type of inertial measurement unit specifies which sensor readings to model:

- `'accel-gyro'` –– Accelerometer and gyroscope
- `'accel-mag'` –– Accelerometer and magnetometer
- `'accel-gyro-mag'` –– Accelerometer, gyroscope, and magnetometer

You can specify `IMUType` as a value-only argument during creation or as a `Name,Value` pair.

Data Types: `char` | `string`

`SampleRate` — Sample rate of sensor (Hz)

`100` (default) | positive scalar

Sample rate of the sensor model in Hz, specified as a positive scalar.

Data Types: `single` | `double`

`Temperature` — Temperature of IMU (ºC)

`25` (default) | real scalar

Operating temperature of the IMU in degrees Celsius, specified as a real scalar.

When the object calculates temperature scale factors and environmental drift noises, 25 ºC is used as the nominal temperature.

Tunable: Yes

Data Types: `single` | `double`

`MagneticField` — Magnetic field vector in local navigation coordinate system (µT)

`[27.5550 -2.4169 -16.0849]` (default) | real scalar

Magnetic field vector in microtesla, specified as a three-element row vector in the local navigation coordinate system.

The default magnetic field corresponds to the magnetic field at latitude zero, longitude zero, and altitude zero.

Tunable: Yes

Data Types: `single` | `double`

`Accelerometer` — Accelerometer sensor parameters
`accelparams` object (default)

Accelerometer sensor parameters, specified by an `accelparams` object.

Tunable: Yes

`Gyroscope` — Gyroscope sensor parameters
`gyroparams` object (default)

Gyroscope sensor parameters, specified by a `gyroparams` object.

Tunable: Yes

`Magnetometer` — Magnetometer sensor parameters
`magparams` object (default)

Magnetometer sensor parameters, specified by a `magparams` object.

Tunable: Yes

`RandomStream` — Random number source
`'Global stream'` (default) | `'mt19937ar with seed'`

Random number source, specified as a character vector or string:

- `'Global stream'` –– Random numbers are generated using the current global random number stream.
- `'mt19937ar with seed'` –– Random numbers are generated using the mt19937ar algorithm with the seed specified by the `Seed` property.

Data Types: `char` | `string`

`Seed` — Initial seed
`67` (default) | nonnegative integer scalar

Initial seed of an mt19937ar random number generator algorithm, specified as a real, nonnegative integer scalar.

Dependencies

To enable this property, set `RandomStream` to `'mt19937ar with seed'`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Usage

Syntax

```
[accelReadings,gyroReadings] = IMU(acc,angVel)
[accelReadings,gyroReadings] = IMU(acc,angVel,orientation)

[accelReadings,magReadings] = IMU(acc,angVel)
[accelReadings,magReadings] = IMU(acc,angVel,orientation)
```

```
[accelReadings,gyroReadings,magReadings] = IMU(acc,angVel)
[accelReadings,gyroReadings,magReadings] = IMU(acc,angVel,orientation)
```

Description

`[accelReadings,gyroReadings] = IMU(acc,angVel)` generates accelerometer and gyroscope readings from the acceleration and angular velocity inputs.

This syntax is only valid if `IMUType` is set to `'accel-gyro'` or `'accel-gyro-mag'`.

`[accelReadings,gyroReadings] = IMU(acc,angVel,orientation)` generates accelerometer and gyroscope readings from the acceleration, angular velocity, and orientation inputs.

This syntax is only valid if `IMUType` is set to `'accel-gyro'` or `'accel-gyro-mag'`.

`[accelReadings,magReadings] = IMU(acc,angVel)` generates accelerometer and magnetometer readings from the acceleration and angular velocity inputs.

This syntax is only valid if `IMUType` is set to `'accel-mag'`.

`[accelReadings,magReadings] = IMU(acc,angVel,orientation)` generates accelerometer and magnetometer readings from the acceleration, angular velocity, and orientation inputs.

This syntax is only valid if `IMUType` is set to `'accel-mag'`.

`[accelReadings,gyroReadings,magReadings] = IMU(acc,angVel)` generates accelerometer, gyroscope, and magnetometer readings from the acceleration and angular velocity inputs.

This syntax is only valid if `IMUType` is set to `'accel-gyro-mag'`.

`[accelReadings,gyroReadings,magReadings] = IMU(acc,angVel,orientation)` generates accelerometer, gyroscope, and magnetometer readings from the acceleration, angular velocity, and orientation inputs.

This syntax is only valid if `IMUType` is set to `'accel-gyro-mag'`.

Input Arguments

`acc` — Acceleration of IMU in local navigation coordinate system (m/s^2)

N-by-3 matrix

Acceleration of the IMU in the local navigation coordinate system, specified as a real, finite *N*-by-3 array in meters per second squared. *N* is the number of samples in the current frame.

Data Types: `single` | `double`

`angVel` — Angular velocity of IMU in local navigation coordinate system (rad/s)

N-by-3 matrix

Angular velocity of the IMU in the local navigation coordinate system, specified as a real, finite *N*-by-3 array in radians per second. *N* is the number of samples in the current frame.

Data Types: `single` | `double`

`orientation` — Orientation of IMU in local navigation coordinate system

N-element quaternion column vector | 3-by-3-by-*N*-element rotation matrix

Orientation of the IMU with respect to the local navigation coordinate system, specified as a `quaternion` *N*-element column vector or a 3-by-3-by-*N* rotation matrix. Each `quaternion` or rotation matrix represents a frame rotation from the local navigation coordinate system to the current IMU sensor body coordinate system. *N* is the number of samples in the current frame.

Data Types: `single` | `double` | `quaternion`

Output Arguments

`accelReadings` — Accelerometer measurement of IMU in sensor body coordinate system (m/s^2)
N-by-3 matrix

Accelerometer measurement of the IMU in the sensor body coordinate system, specified as a real, finite *N*-by-3 array in meters per second squared. *N* is the number of samples in the current frame.

Data Types: `single` | `double`

`gyroReadings` — Gyroscope measurement of IMU in sensor body coordinate system (rad/s)
N-by-3 matrix

Gyroscope measurement of the IMU in the sensor body coordinate system, specified as a real, finite *N*-by-3 array in radians per second. *N* is the number of samples in the current frame.

Data Types: `single` | `double`

`magReadings` — Magnetometer measurement of IMU in sensor body coordinate system (µT)
N-by-3 matrix (default)

Magnetometer measurement of the IMU in the sensor body coordinate system, specified as a real, finite *N*-by-3 array in microtelsa. *N* is the number of samples in the current frame.

Data Types: `single` | `double`

Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

Specific to `imuSensor`

loadparams	Load sensor parameters from JSON file
perturbations	Perturbation defined on object
perturb	Apply perturbations to object

Common to All System Objects

step	Run System object algorithm
release	Release resources and allow changes to System object property values and input characteristics
reset	Reset internal states of System object

Examples

Create Default `imuSensor` System object

The `imuSensor` System object™ enables you to model the data received from an inertial measurement unit consisting of a combination of gyroscope, accelerometer, and magnetometer.

Create a default `imuSensor` object.

```
IMU = imuSensor
```

```
IMU =
  imuSensor with properties:

          IMUType: 'accel-gyro'
       SampleRate: 100
      Temperature: 25
    Accelerometer: [1x1 accelparams]
        Gyroscope: [1x1 gyroparams]
     RandomStream: 'Global stream'
```

The `imuSensor` object, `IMU`, contains an idealized gyroscope and accelerometer. Use dot notation to view properties of the gyroscope.

```
IMU.Gyroscope
```

```
ans =
  gyroparams with properties:

    MeasurementRange: Inf             rad/s
          Resolution: 0               (rad/s)/LSB
        ConstantBias: [0 0 0]         rad/s
    AxesMisalignment: [3x3 double]    %

       NoiseDensity: [0 0 0]    (rad/s)/√Hz
    BiasInstability: [0 0 0]    rad/s
         RandomWalk: [0 0 0]    (rad/s)*√Hz

           TemperatureBias: [0 0 0]    (rad/s)/°C
    TemperatureScaleFactor: [0 0 0]    %/°C
          AccelerationBias: [0 0 0]    (rad/s)/(m/s²)
```

Sensor properties are defined by corresponding parameter objects. For example, the gyroscope model used by the `imuSensor` is defined by an instance of the `gyroparams` class. You can modify properties of the gyroscope model using dot notation. Set the gyroscope measurement range to 4.3 rad/s.

```
IMU.Gyroscope.MeasurementRange = 4.3;
```

You can also set sensor properties to preset parameter objects. Create an `accelparams` object to mimic specific hardware, and then set the IMU `Accelerometer` property to the `accelparams` object. Display the `Accelerometer` property to verify the properties are correctly set.

```
SpecSheet1 = accelparams( ...
    'MeasurementRange',19.62, ...
    'Resolution',0.00059875, ...
    'ConstantBias',0.4905, ...
    'AxesMisalignment',2, ...
```

```
    'NoiseDensity',0.003924, ...
    'BiasInstability',0, ...
    'TemperatureBias', [0.34335 0.34335 0.5886], ...
    'TemperatureScaleFactor', 0.02);

IMU.Accelerometer = SpecSheet1;

IMU.Accelerometer

ans =
  accelparams with properties:

    MeasurementRange: 19.62                     m/s²
          Resolution: 0.00059875                (m/s²)/LSB
      ConstantBias: [0.4905 0.4905 0.4905]    m/s²
    AxesMisalignment: [3x3 double]              %

       NoiseDensity: [0.003924 0.003924 0.003924]    (m/s²)/√Hz
    BiasInstability: [0 0 0]                         m/s²
         RandomWalk: [0 0 0]                         (m/s²)*√Hz

           TemperatureBias: [0.34335 0.34335 0.5886]    (m/s²)/°C
    TemperatureScaleFactor: [0.02 0.02 0.02]            %/°C
```

Generate IMU Data from Stationary Input

Use the `imuSensor` System object™ to model receiving data from a stationary ideal IMU containing an accelerometer, gyroscope, and magnetometer.

Create an ideal IMU sensor model that contains an accelerometer, gyroscope, and magnetometer.

```
IMU = imuSensor('accel-gyro-mag')

IMU =
  imuSensor with properties:

          IMUType: 'accel-gyro-mag'
       SampleRate: 100
      Temperature: 25
    MagneticField: [27.5550 -2.4169 -16.0849]
    Accelerometer: [1x1 accelparams]
        Gyroscope: [1x1 gyroparams]
     Magnetometer: [1x1 magparams]
     RandomStream: 'Global stream'
```

Define the ground-truth, underlying motion of the IMU you are modeling. The acceleration and angular velocity are defined relative to the local NED coordinate system.

```
numSamples = 1000;
acceleration = zeros(numSamples,3);
angularVelocity = zeros(numSamples,3);
```

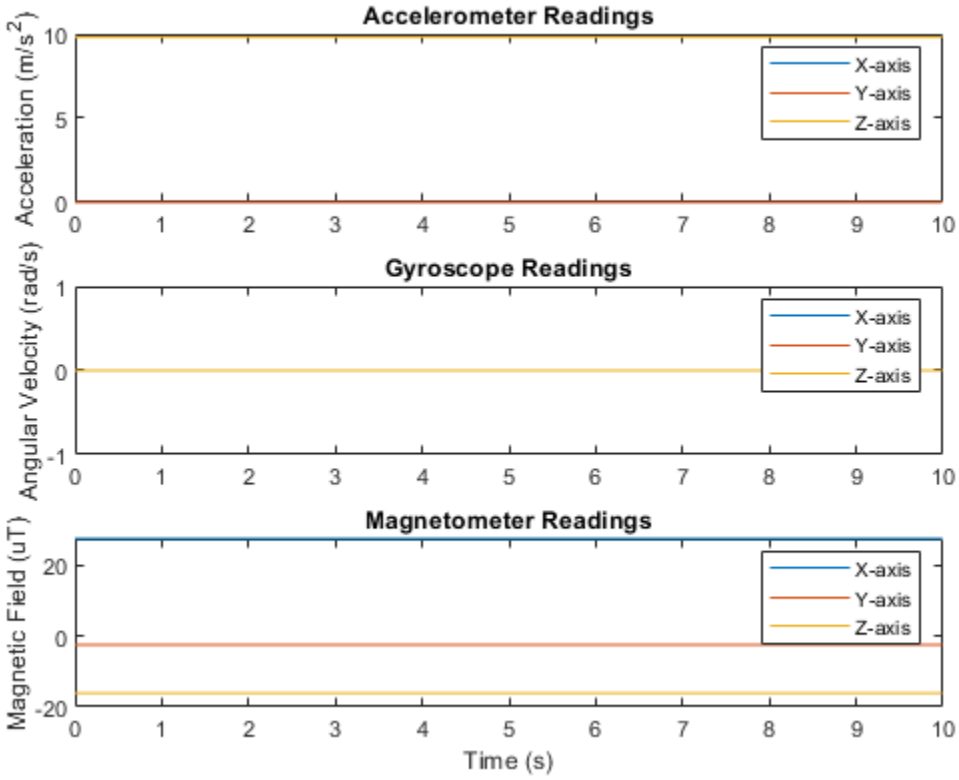
Call `IMU` with the ground-truth acceleration and angular velocity. The object outputs accelerometer readings, gyroscope readings, and magnetometer readings, as modeled by the properties of the

`imuSensor` System object. The accelerometer readings, gyroscope readings, and magnetometer readings are relative to the IMU sensor body coordinate system.

```
[accelReading,gyroReading,magReading] = IMU(acceleration,angularVelocity);
```

Plot the accelerometer readings, gyroscope readings, and magnetometer readings.

```
t = (0:(numSamples-1))/IMU.SampleRate;
subplot(3,1,1)
plot(t,accelReading)
legend('X-axis','Y-axis','Z-axis')
title('Accelerometer Readings')
ylabel('Acceleration (m/s^2)')

subplot(3,1,2)
plot(t,gyroReading)
legend('X-axis','Y-axis','Z-axis')
title('Gyroscope Readings')
ylabel('Angular Velocity (rad/s)')

subplot(3,1,3)
plot(t,magReading)
legend('X-axis','Y-axis','Z-axis')
title('Magnetometer Readings')
xlabel('Time (s)')
ylabel('Magnetic Field (uT)')
```



Orientation is not specified and the ground-truth motion is stationary, so the IMU sensor body coordinate system and the local NED coordinate system overlap for the entire simulation.

- Accelerometer readings: The *z*-axis of the sensor body corresponds to the Down-axis. The 9.8 m/s^2 acceleration along the *z*-axis is due to gravity.
- Gyroscope readings: The gyroscope readings are zero along each axis, as expected.
- Magnetometer readings: Because the sensor body coordinate system is aligned with the local NED coordinate system, the magnetometer readings correspond to the `MagneticField` property of `imuSensor`. The `MagneticField` property is defined in the local NED coordinate system.

Model Rotating Six-Axis IMU Data

Use `imuSensor` to model data obtained from a rotating IMU containing an ideal accelerometer and an ideal magnetometer. Use `kinematicTrajectory` to define the ground-truth motion. Fuse the `imuSensor` model output using the `ecompass` function to determine orientation over time.

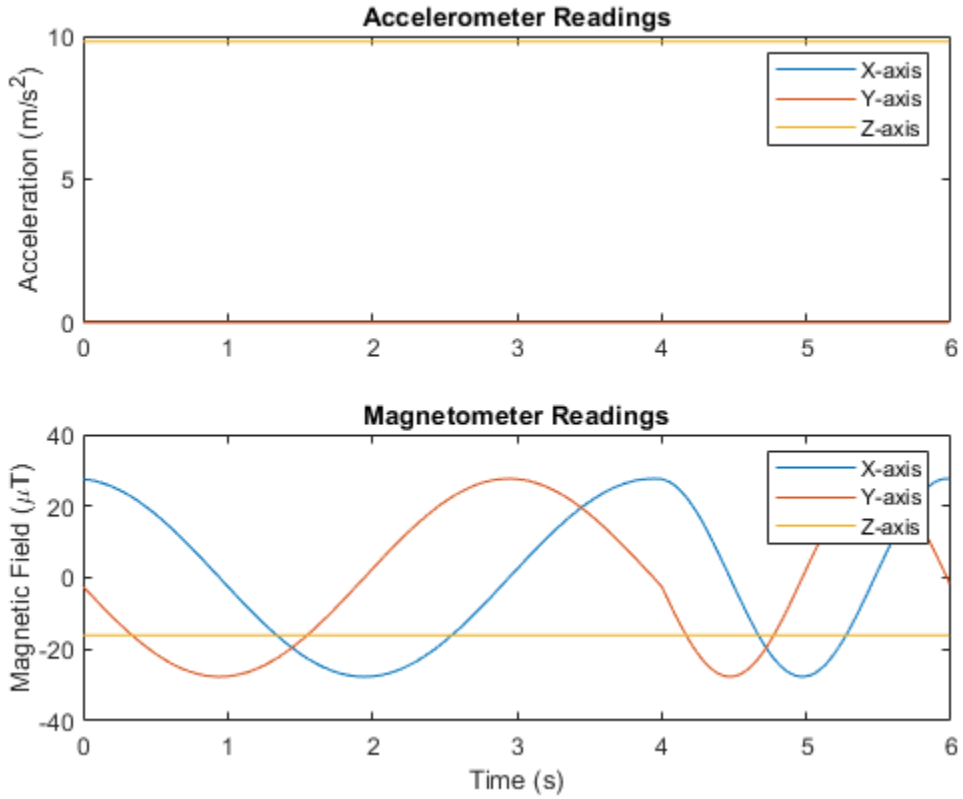
Define the ground-truth motion for a platform that rotates 360 degrees in four seconds, and then another 360 degrees in two seconds. Use `kinematicTrajectory` to output the orientation, acceleration, and angular velocity in the NED coordinate system.

```
fs = 100;
firstLoopNumSamples = fs*4;
secondLoopNumSamples = fs*2;
totalNumSamples = firstLoopNumSamples + secondLoopNumSamples;

traj = kinematicTrajectory('SampleRate',fs);

accBody = zeros(totalNumSamples,3);
angVelBody = zeros(totalNumSamples,3);
angVelBody(1:firstLoopNumSamples,3) = (2*pi)/4;
angVelBody(firstLoopNumSamples+1:end,3) = (2*pi)/2;

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

Create an `imuSensor` object with an ideal accelerometer and an ideal magnetometer. Call `IMU` with the ground-truth acceleration, angular velocity, and orientation to output accelerometer readings and magnetometer readings. Plot the results.

```
IMU = imuSensor('accel-mag','SampleRate',fs);

[accelReadings,magReadings] = IMU(accNED,angVelNED,orientationNED);

figure(1)
t = (0:(totalNumSamples-1))/fs;
subplot(2,1,1)
plot(t,accelReadings)
legend('X-axis','Y-axis','Z-axis')
ylabel('Acceleration (m/s^2)')
title('Accelerometer Readings')

subplot(2,1,2)
plot(t,magReadings)
legend('X-axis','Y-axis','Z-axis')
```

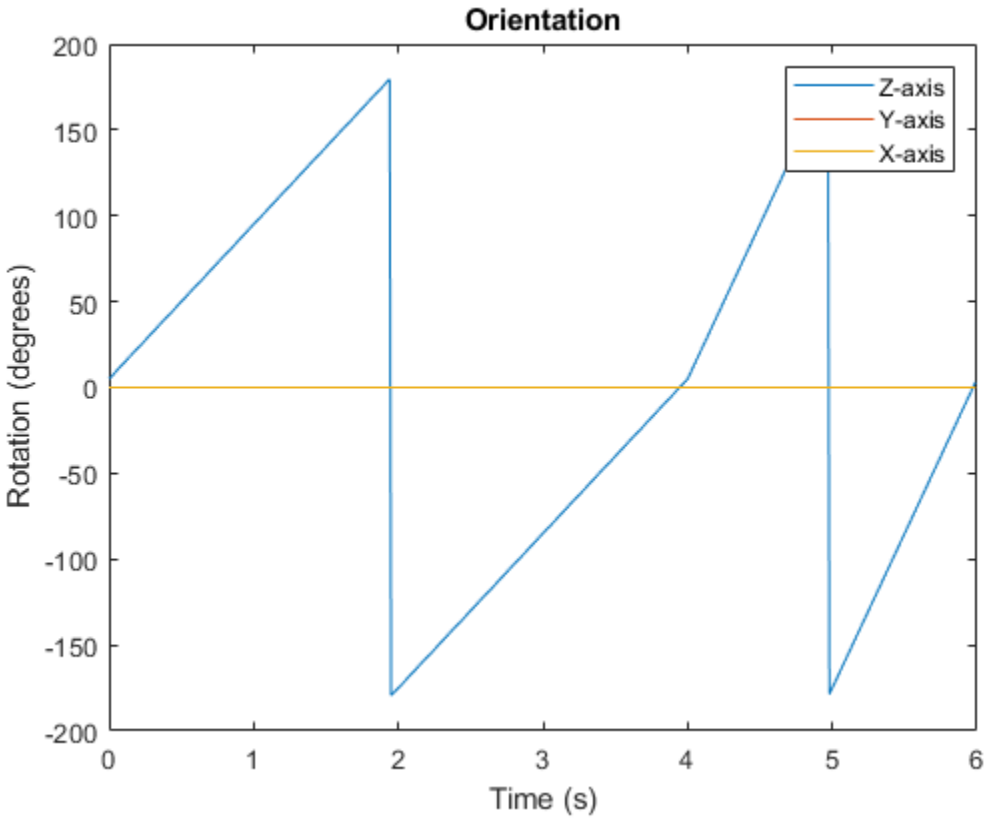
```
ylabel('Magnetic Field (\muT)')
xlabel('Time (s)')
title('Magnetometer Readings')
```



The accelerometer readings indicate that the platform has no translation. The magnetometer readings indicate that the platform is rotating around the *z*-axis.

Feed the accelerometer and magnetometer readings into the `ecompass` function to estimate the orientation over time. The `ecompass` function returns orientation in quaternion format. Convert orientation to Euler angles and plot the results. The orientation plot indicates that the platform rotates about the *z*-axis only.

```
orientation = ecompass(accelReadings,magReadings);

orientationEuler = eulerd(orientation,'ZYX','frame');

figure(2)
plot(t,orientationEuler)
legend('Z-axis','Y-axis','X-axis')
xlabel('Time (s)')
ylabel('Rotation (degrees)')
title('Orientation')
```



Model Rotating Six-Axis IMU Data with Noise

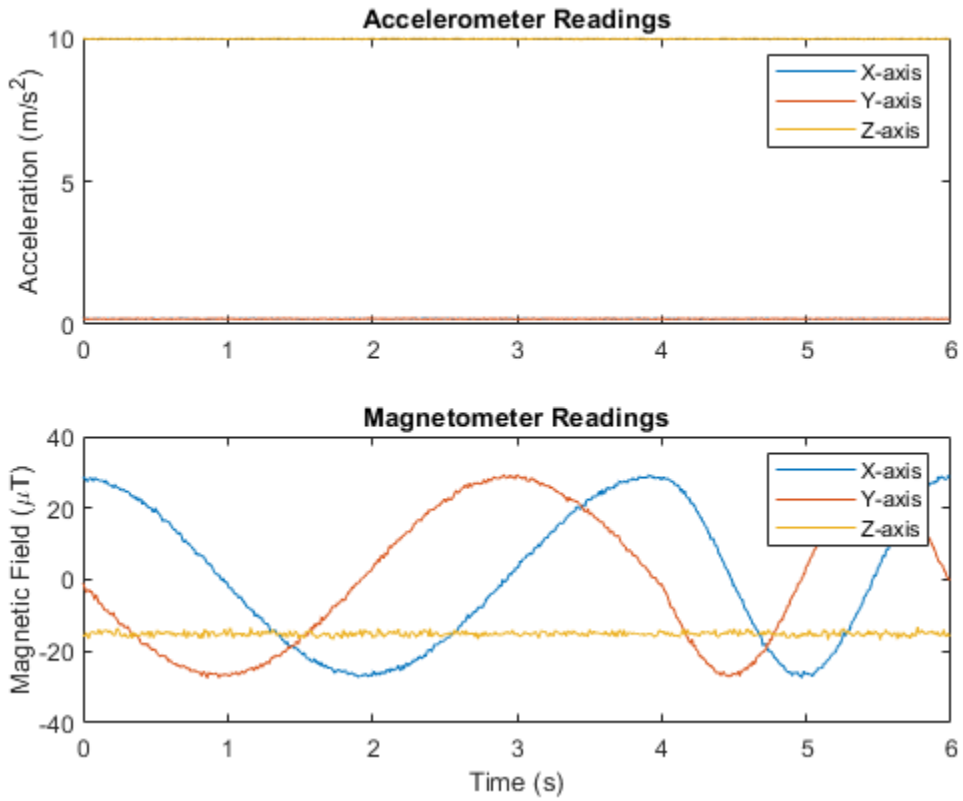
Use `imuSensor` to model data obtained from a rotating IMU containing a realistic accelerometer and a realistic magnetometer. Use `kinematicTrajectory` to define the ground-truth motion. Fuse the `imuSensor` model output using the `ecompass` function to determine orientation over time.

Define the ground-truth motion for a platform that rotates 360 degrees in four seconds, and then another 360 degrees in two seconds. Use `kinematicTrajectory` to output the orientation, acceleration, and angular velocity in the NED coordinate system.

```
fs = 100;
firstLoopNumSamples = fs*4;
secondLoopNumSamples = fs*2;
totalNumSamples = firstLoopNumSamples + secondLoopNumSamples;

traj = kinematicTrajectory('SampleRate',fs);

accBody = zeros(totalNumSamples,3);
angVelBody = zeros(totalNumSamples,3);
angVelBody(1:firstLoopNumSamples,3) = (2*pi)/4;
angVelBody(firstLoopNumSamples+1:end,3) = (2*pi)/2;

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

Create an `imuSensor` object with a realistic accelerometer and a realistic magnetometer. Call `IMU` with the ground-truth acceleration, angular velocity, and orientation to output accelerometer readings and magnetometer readings. Plot the results.

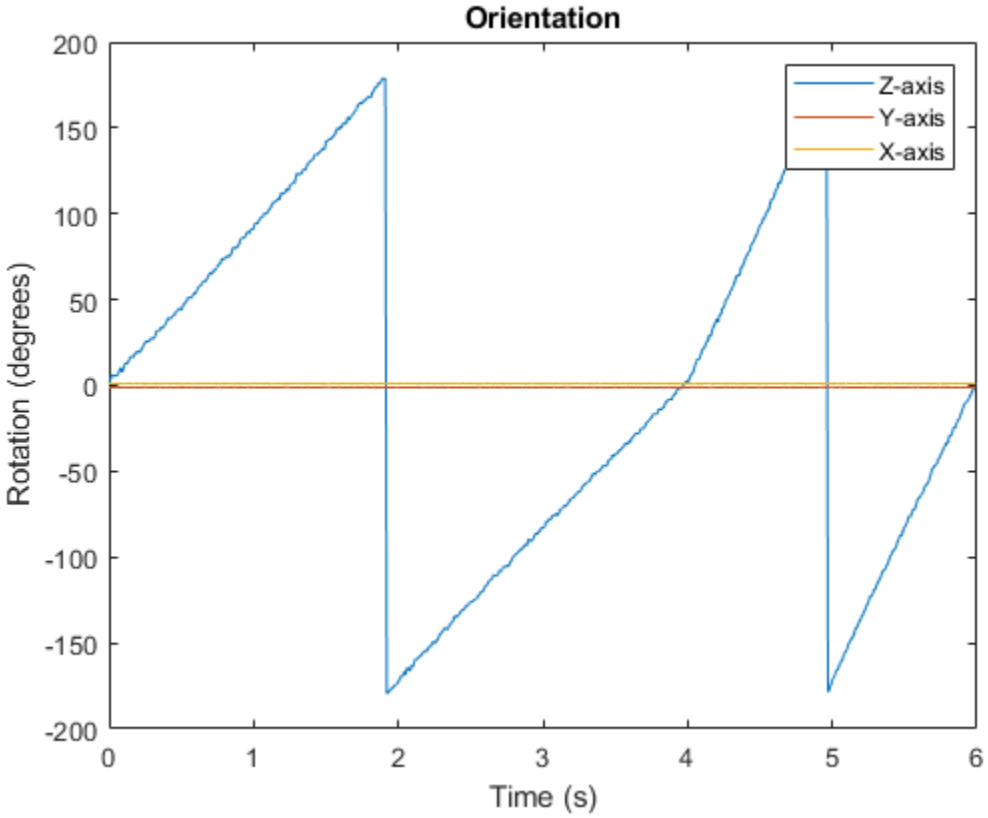
```
IMU = imuSensor('accel-mag','SampleRate',fs);

IMU.Accelerometer = accelparams( ...
    'MeasurementRange',19.62, ...             % m/s^2
    'Resolution',0.0023936, ...               % m/s^2 / LSB
    'TemperatureScaleFactor',0.008, ...       % % / degree C
    'ConstantBias',0.1962, ...                % m/s^2
    'TemperatureBias',0.0014715, ...          % m/s^2 / degree C
    'NoiseDensity',0.0012361);                % m/s^2 / Hz^(1/2)

IMU.Magnetometer = magparams( ...
    'MeasurementRange',1200, ...              % uT
    'Resolution',0.1, ...                     % uT / LSB
    'TemperatureScaleFactor',0.1, ...         % % / degree C
    'ConstantBias',1, ...                     % uT
    'TemperatureBias',[0.8 0.8 2.4], ...      % uT / degree C
    'NoiseDensity',[0.6 0.6 0.9]/sqrt(100)); % uT / Hz^(1/2)

[accelReadings,magReadings] = IMU(accNED,angVelNED,orientationNED);

figure(1)
t = (0:(totalNumSamples-1))/fs;
subplot(2,1,1)
plot(t,accelReadings)
legend('X-axis','Y-axis','Z-axis')
ylabel('Acceleration (m/s^2)')
title('Accelerometer Readings')

subplot(2,1,2)
plot(t,magReadings)
legend('X-axis','Y-axis','Z-axis')
ylabel('Magnetic Field (\muT)')
xlabel('Time (s)')
title('Magnetometer Readings')
```



The accelerometer readings indicate that the platform has no translation. The magnetometer readings indicate that the platform is rotating around the *z*-axis.

Feed the accelerometer and magnetometer readings into the `ecompass` function to estimate the orientation over time. The `ecompass` function returns orientation in quaternion format. Convert orientation to Euler angles and plot the results. The orientation plot indicates that the platform rotates about the *z*-axis only.

```
orientation = ecompass(accelReadings,magReadings);

orientationEuler = eulerd(orientation,'ZYX','frame');

figure(2)
plot(t,orientationEuler)
legend('Z-axis','Y-axis','X-axis')
xlabel('Time (s)')
ylabel('Rotation (degrees)')
title('Orientation')
```



%

Model Tilt Using Gyroscope and Accelerometer Readings

Model a tilting IMU that contains an accelerometer and gyroscope using the `imuSensor` System object™. Use ideal and realistic models to compare the results of orientation tracking using the `imufilter` System object.

Load a struct describing ground-truth motion and a sample rate. The motion struct describes sequential rotations:

1. yaw: 120 degrees over two seconds
2. pitch: 60 degrees over one second
3. roll: 30 degrees over one-half second
4. roll: -30 degrees over one-half second
5. pitch: -60 degrees over one second
6. yaw: -120 degrees over two seconds

In the last stage, the motion struct combines the 1st, 2nd, and 3rd rotations into a single-axis rotation. The acceleration, angular velocity, and orientation are defined in the local NED coordinate system.

```
load y120p60r30.mat motion fs
accNED = motion.Acceleration;
angVelNED = motion.AngularVelocity;
orientationNED = motion.Orientation;

numSamples = size(motion.Orientation,1);
t = (0:(numSamples-1))'/fs;
```

Create an ideal IMU sensor object and a default IMU filter object.

```
IMU = imuSensor('accel-gyro','SampleRate',fs);

aFilter = imufilter('SampleRate',fs);
```

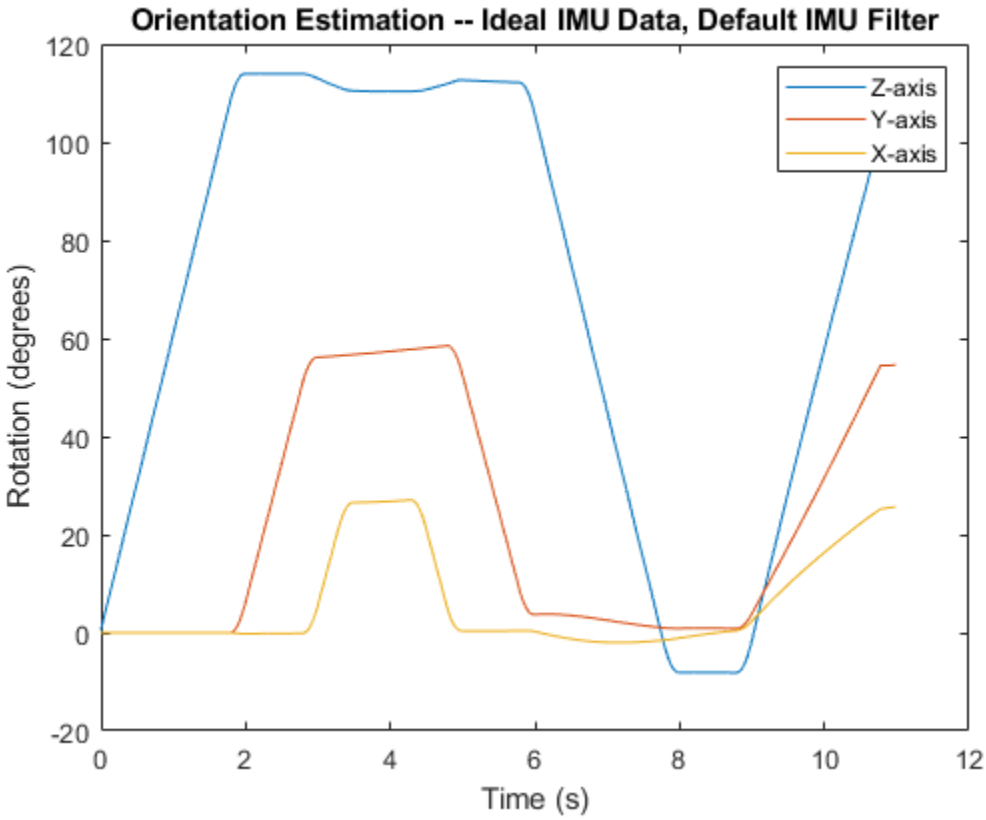
In a loop:

1. Simulate IMU output by feeding the ground-truth motion to the IMU sensor object.
2. Filter the IMU output using the default IMU filter object.

```
orientation = zeros(numSamples,1,'quaternion');
for i = 1:numSamples

    [accelBody,gyroBody] = IMU(accNED(i,:),angVelNED(i,:),orientationNED(i,:));

    orientation(i) = aFilter(accelBody,gyroBody);

end
release(aFilter)
```

Plot the orientation over time.

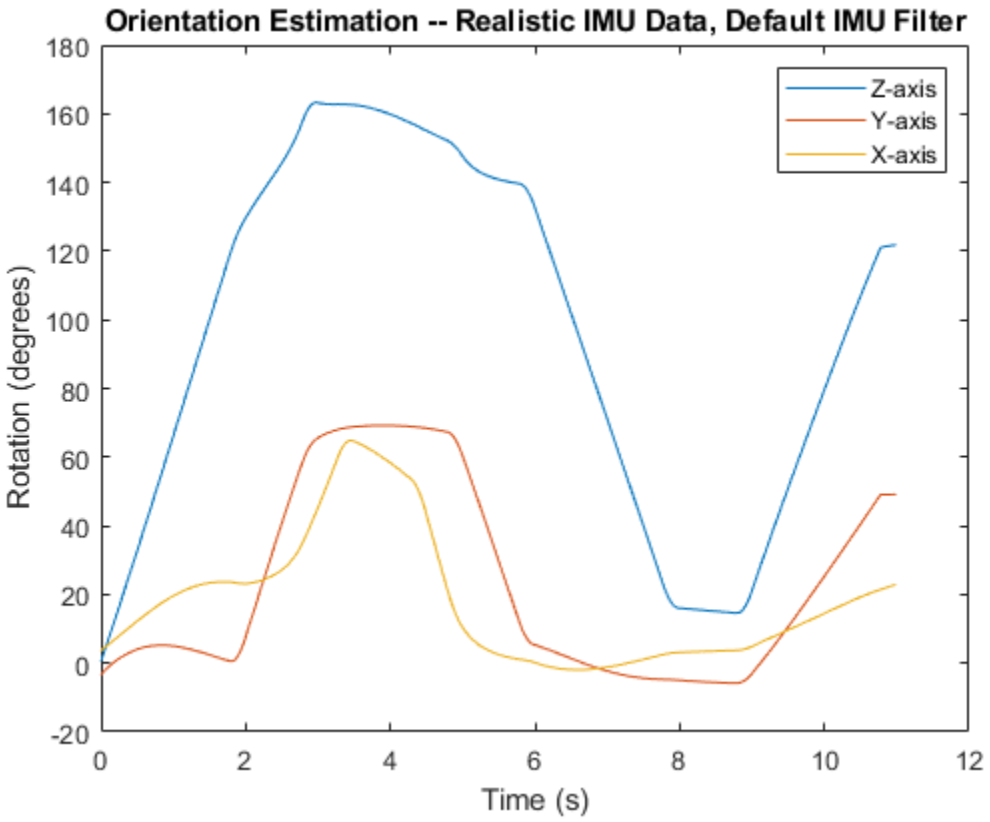
```
figure(1)
plot(t,eulerd(orientation,'ZYX','frame'))
xlabel('Time (s)')
ylabel('Rotation (degrees)')
title('Orientation Estimation -- Ideal IMU Data, Default IMU Filter')
legend('Z-axis','Y-axis','X-axis')
```



Modify properties of your `imuSensor` to model real-world sensors. Run the loop again and plot the orientation estimate over time.

```
IMU.Accelerometer = accelparams( ...
    'MeasurementRange',19.62, ...
    'Resolution',0.00059875, ...
    'ConstantBias',0.4905, ...
    'AxesMisalignment',2, ...
    'NoiseDensity',0.003924, ...
    'BiasInstability',0, ...
    'TemperatureBias', [0.34335 0.34335 0.5886], ...
    'TemperatureScaleFactor',0.02);
IMU.Gyroscope = gyroparams( ...
    'MeasurementRange',4.3633, ...
    'Resolution',0.00013323, ...
    'AxesMisalignment',2, ...
    'NoiseDensity',8.7266e-05, ...
    'TemperatureBias',0.34907, ...
    'TemperatureScaleFactor',0.02, ...
    'AccelerationBias',0.00017809, ...
    'ConstantBias',[0.3491,0.5,0]);

orientationDefault = zeros(numSamples,1,'quaternion');
for i = 1:numSamples

    [accelBody,gyroBody] = IMU(accNED(i,:),angVelNED(i,:),orientationNED(i,:));
```

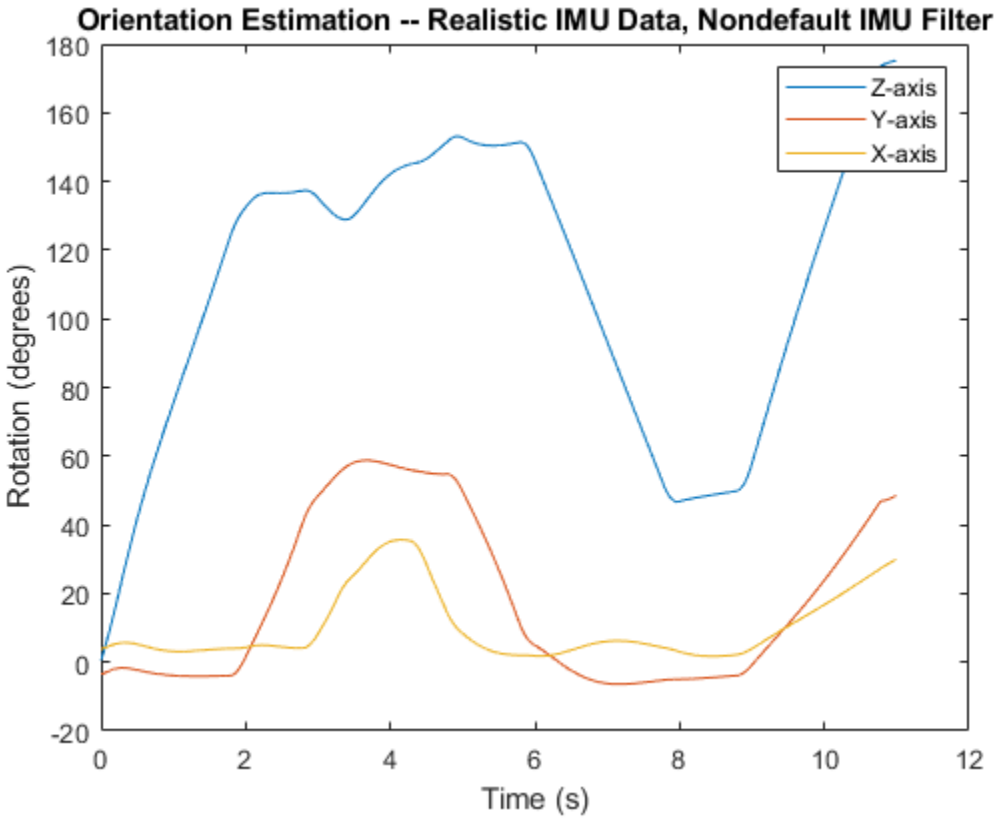
```
    orientationDefault(i) = aFilter(accelBody,gyroBody);

end
release(aFilter)

figure(2)
plot(t,eulerd(orientationDefault,'ZYX','frame'))
xlabel('Time (s)')
ylabel('Rotation (degrees)')
title('Orientation Estimation -- Realistic IMU Data, Default IMU Filter')
legend('Z-axis','Y-axis','X-axis')
```



The ability of the `imufilter` to track the ground-truth data is significantly reduced when modeling a realistic IMU. To improve performance, modify properties of your `imufilter` object. These values were determined empirically. Run the loop again and plot the orientation estimate over time.

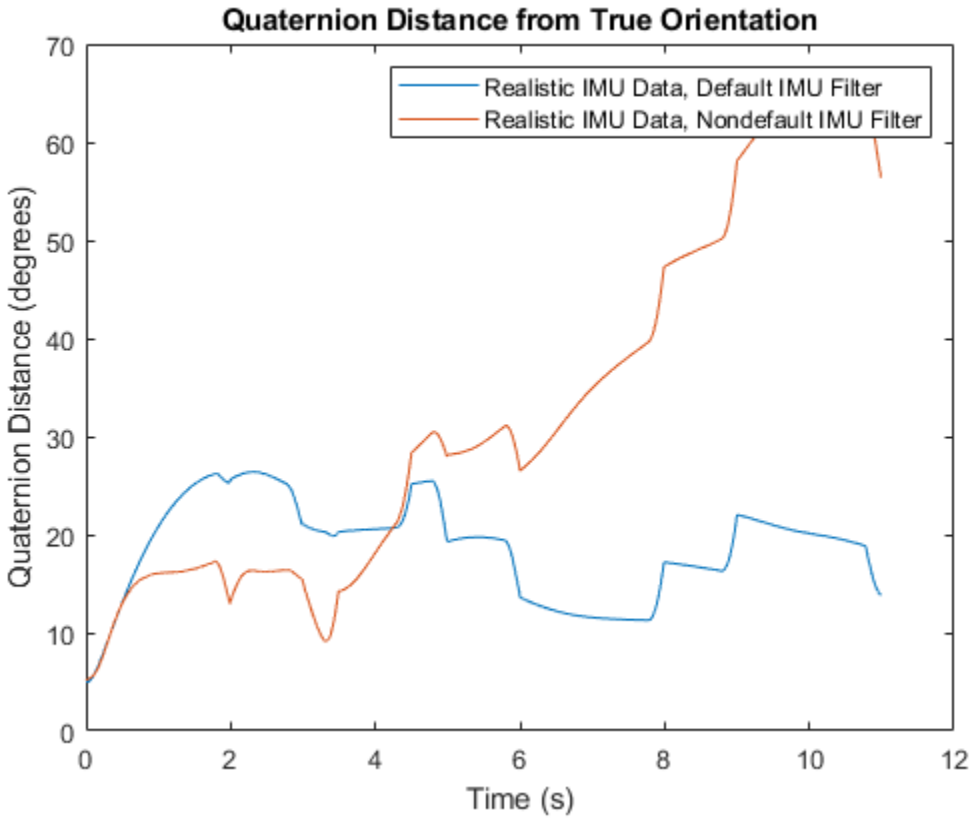
```
aFilter.GyroscopeNoise          = 7.6154e-7;
aFilter.AccelerometerNoise      = 0.0015398;
aFilter.GyroscopeDriftNoise     = 3.0462e-12;
aFilter.LinearAccelerationNoise = 0.00096236;
aFilter.InitialProcessNoise     = aFilter.InitialProcessNoise*10;

orientationNondefault = zeros(numSamples,1,'quaternion');
for i = 1:numSamples
    [accelBody,gyroBody] = IMU(accNED(i,:),angVelNED(i,:),orientationNED(i,:));

    orientationNondefault(i) = aFilter(accelBody,gyroBody);
```

```
end
release(aFilter)

figure(3)
plot(t,eulerd(orientationNondefault,'ZYX','frame'))
xlabel('Time (s)')
ylabel('Rotation (degrees)')
title('Orientation Estimation -- Realistic IMU Data, Nondefault IMU Filter')
legend('Z-axis','Y-axis','X-axis')
```



To quantify the improved performance of the modified `imufilter`, plot the quaternion distance between the ground-truth motion and the orientation as returned by the `imufilter` with default and nondefault properties.

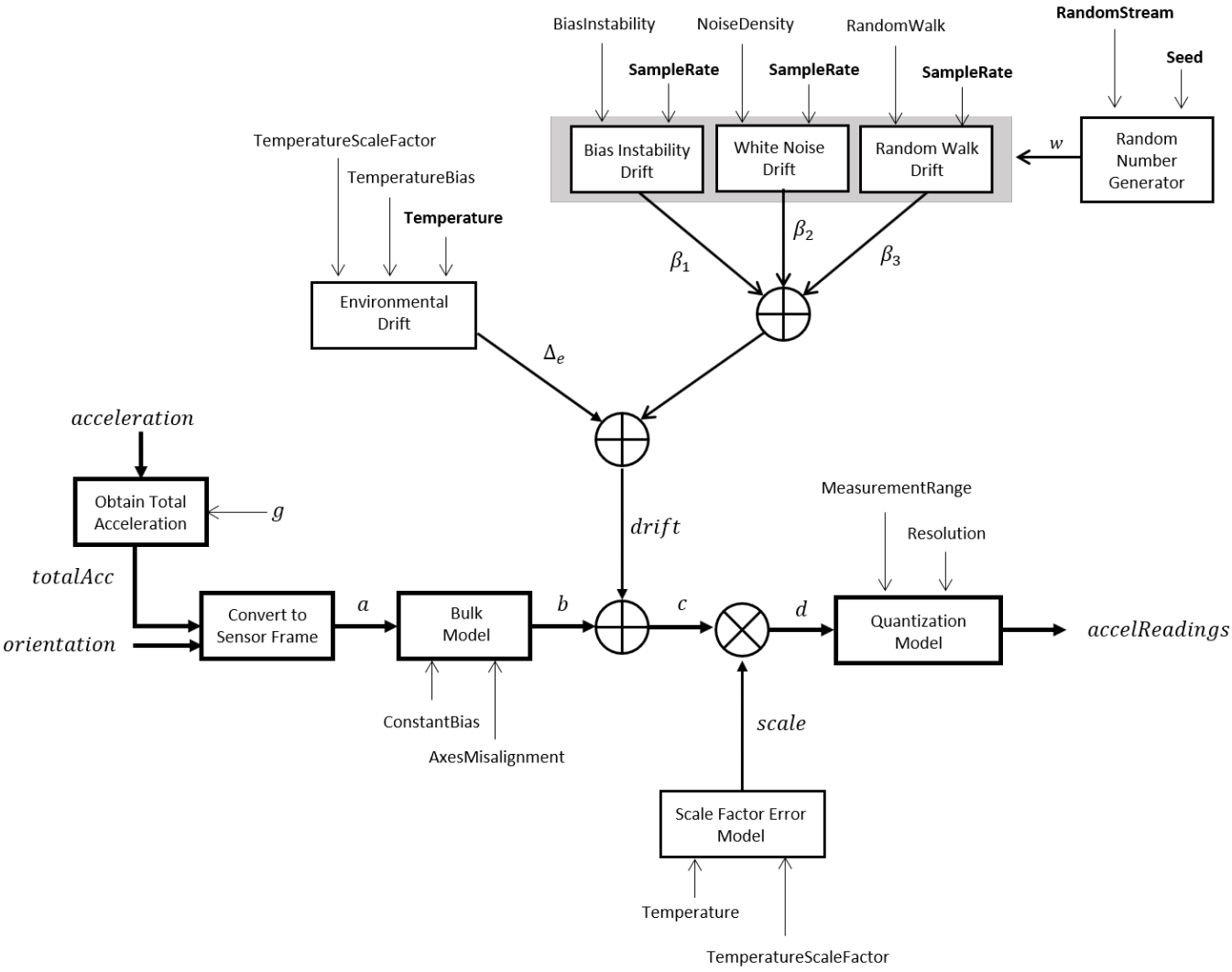
```
qDistDefault = rad2deg(dist(orientationNED,orientationDefault));
qDistNondefault = rad2deg(dist(orientationNED,orientationNondefault));

figure(4)
plot(t,[qDistDefault,qDistNondefault])
title('Quaternion Distance from True Orientation')
legend('Realistic IMU Data, Default IMU Filter', ...
       'Realistic IMU Data, Nondefault IMU Filter')
xlabel('Time (s)')
ylabel('Quaternion Distance (degrees)')
```



Algorithms

Accelerometer

The following algorithm description assumes an NED navigation frame. The accelerometer model uses the ground-truth orientation and acceleration inputs and the `imuSensor` and `accelparams` properties to model accelerometer readings.



Obtain Total Acceleration

To obtain the total acceleration (*totalAcc*), the acceleration is preprocessed by negating and adding the gravity constant vector (g= [0; 0; 9.8] m/s^2 assuming an NED frame) as:

$$totalAcc = -acceleration + g$$

The `acceleration` term is negated to obtain zero total acceleration readings when the accelerometer is in a free fall. The `acceleration` term is also known as the specific force.

Convert to Sensor Frame

Then the total acceleration is converted from the local navigation frame to the sensor frame using:

$$a = (orientation)(totalAcc)^T$$

If the orientation is input in quaternion form, it is converted to a rotation matrix before processing.

Bulk Model

The ground-truth acceleration in the sensor frame, a, passes through the bulk model, which adds axes misalignment and bias:

$$b = \left(\begin{bmatrix} 1 & \frac{\alpha_2}{100} & \frac{\alpha_3}{100} \\ \frac{\alpha_1}{100} & 1 & \frac{\alpha_3}{100} \\ \frac{\alpha_1}{100} & \frac{\alpha_2}{100} & 1 \end{bmatrix} \left(a^T\right) \right)^T + \text{ConstantBias}$$

where ConstantBias is a property of `accelparams`, and α_1, α_2, and α_3 are given by the first, second, and third elements of the AxesMisalignment property of `accelparams`.

Bias Instability Drift

The bias instability drift is modeled as white noise biased and then filtered:

$$\beta_1 = h_1 * (w)(\text{BiasInstability})$$

where BiasInstability is a property of `accelparams`, and h_1 is a filter defined by the SampleRate property:

$$H_1(z) = \frac{1}{1 - \frac{1}{2}z^{-1}}$$

White Noise Drift

White noise drift is modeled by multiplying elements of the white noise random stream by the standard deviation:

$$\beta_2 = (w)\left(\sqrt{\frac{\text{SampleRate}}{2}}\right)(\text{NoiseDensity})$$

where SampleRate is an `imuSensor` property, and NoiseDensity is an `accelparams` property. Elements of w are random numbers given by settings of the `imuSensor` random stream.

Random Walk Drift

The random walk drift is modeled by biasing elements of the white noise random stream and then filtering:

$$\beta_3 = h_2 * (w)\left(\frac{\text{RandomWalk}}{\sqrt{\frac{\text{SampleRate}}{2}}}\right)$$

where RandomWalk is a property of `accelparams`, SampleRate is a property of `imuSensor`, and h_2 is a filter defined as:

$$H_2(z) = \frac{1}{1 - z^{-1}}$$

Environmental Drift Noise

The environmental drift noise is modeled by multiplying the temperature difference from a standard with the temperature bias:

$$\Delta_e = (\text{Temperature} - 25)(\text{TemperatureBias})$$

where Temperature is a property of `imuSensor`, and TemperatureBias is a property of `accelparams`. The constant 25 corresponds to a standard temperature.

Scale Factor Error Model

The temperature scale factor error is modeled as:

$$scaleFactorError = 1 + \left(\frac{\text{Temperature} - 25}{100}\right)(\text{TemperatureScaleFactor})$$

where Temperature is a property of `imuSensor`, and TemperatureScaleFactor is a property of `accelparams`. The constant 25 corresponds to a standard temperature.

Quantization Model

The quantization is modeled by first saturating the continuous signal model:

$$e = \begin{cases} \text{MeasurementRange} & \text{if } d > \text{MeasurementRange} \\ -\text{MeasurementRange} & \text{if } -d > \text{MeasurementRange} \\ d & \text{else} \end{cases}$$

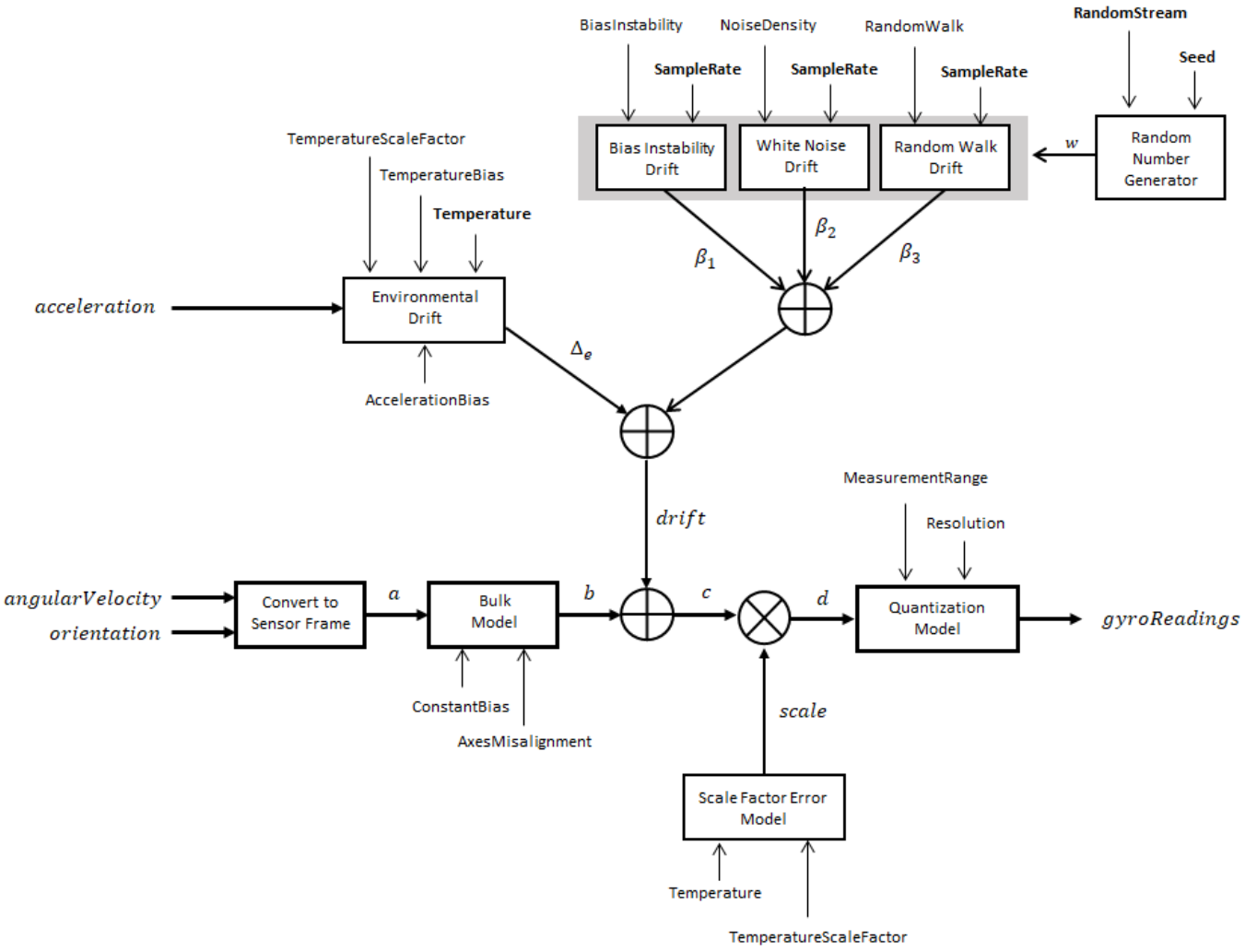
and then setting the resolution:

$$accelReadings = (\text{Resolution})\left(\text{round}\left(\frac{e}{\text{Resolution}}\right)\right)$$

where MeasurementRange is a property of `accelparams`.

Gyroscope

The following algorithm description assumes an NED navigation frame. The gyroscope model uses the ground-truth orientation, acceleration, and angular velocity inputs, and the `imuSensor` and `gyroparams` properties to model accelerometer readings.



Convert to Sensor Frame

The ground-truth angular velocity is converted from the local frame to the sensor frame using the ground-truth orientation:

$$a = (orientation)(angularVelocity)^T$$

If the orientation is input in quaternion form, it is converted to a rotation matrix before processing.

Bulk Model

The ground-truth angular velocity in the sensor frame, a, passes through the bulk model, which adds axes misalignment and bias:

$$b = \left(\begin{bmatrix} 1 & \frac{\alpha_2}{100} & \frac{\alpha_3}{100} \\ \frac{\alpha_1}{100} & 1 & \frac{\alpha_3}{100} \\ \frac{\alpha_1}{100} & \frac{\alpha_2}{100} & 1 \end{bmatrix} \left(a^T\right) \right)^T + \text{ConstantBias}$$

where ConstantBias is a property of `gyroparams`, and α_1, α_2, and α_3 are given by the first, second, and third elements of the AxesMisalignment property of `gyroparams`.

Bias Instability Drift

The bias instability drift is modeled as white noise biased and then filtered:

$$\beta_1 = h_1 * (w)(\text{BiasInstability})$$

where BiasInstability is a property of `gyroparams` and h_1 is a filter defined by the SampleRate property:

$$H_1(z) = \frac{1}{1 - \frac{1}{2}z^{-1}}$$

White Noise Drift

White noise drift is modeled by multiplying elements of the white noise random stream by the standard deviation:

$$\beta_2 = (w)\left(\sqrt{\frac{\text{SampleRate}}{2}}\right)(\text{NoiseDensity})$$

where SampleRate is an `imuSensor` property, and NoiseDensity is an `gyroparams` property. The elements of w are random numbers given by settings of the `imuSensor` random stream.

Random Walk Drift

The random walk drift is modeled by biasing elements of the white noise random stream and then filtering:

$$\beta_3 = h_2 * (w)\left(\frac{\text{RandomWalk}}{\sqrt{\frac{\text{SampleRate}}{2}}}\right)$$

where RandomWalk is a property of `gyroparams`, SampleRate is a property of `imuSensor`, and h_2 is a filter defined as:

$$H_2(z) = \frac{1}{1 - z^{-1}}$$

Environmental Drift Noise

The environmental drift noise is modeled by multiplying the temperature difference from a standard with the temperature bias:

$$\Delta_e = (\text{Temperature} - 25)(\text{TemperatureBias})$$

where Temperature is a property of `imuSensor`, and TemperatureBias is a property of `gyroparams`. The constant 25 corresponds to a standard temperature.

Scale Factor Error Model

The temperature scale factor error is modeled as:

$$scaleFactorError = 1 + \left(\frac{\text{Temperature} - 25}{100}\right)(\text{TemperatureScaleFactor})$$

where Temperature is a property of `imuSensor`, and TemperatureScaleFactor is a property of `gyroparams`. The constant 25 corresponds to a standard temperature.

Quantization Model

The quantization is modeled by first saturating the continuous signal model:

$$e = \begin{cases} \text{MeasurementRange} & \text{if } d > \text{MeasurementRange} \\ -\text{MeasurementRange} & \text{if } -d > \text{MeasurementRange} \\ d & \text{else} \end{cases}$$

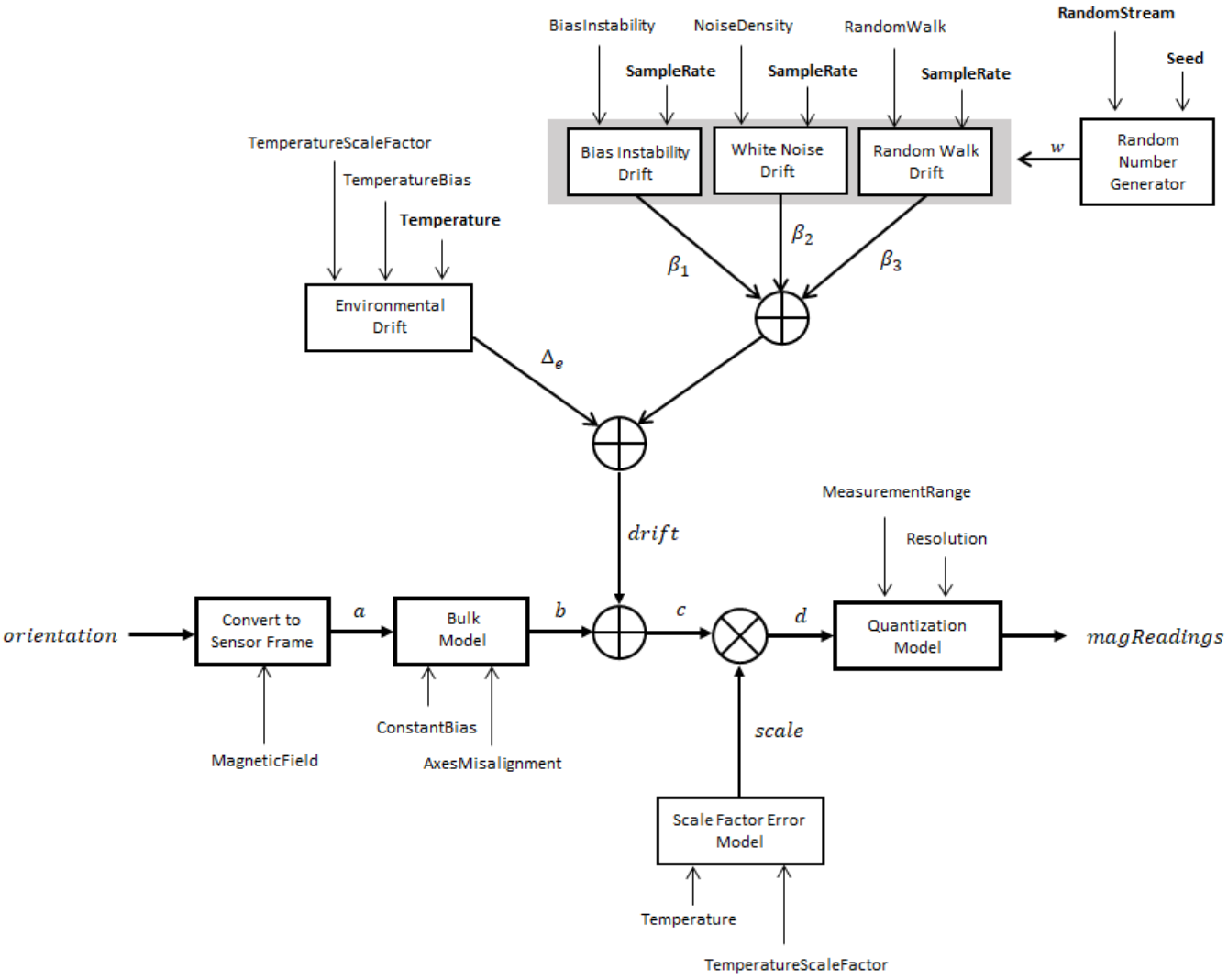
and then setting the resolution:

$$gyroReadings = (\text{Resolution})\left(\text{round}\left(\frac{e}{\text{Resolution}}\right)\right)$$

where MeasurementRange is a property of `gyroparams`.

Magnetometer

The following algorithm description assumes an NED navigation frame. The magnetometer model uses the ground-truth orientation and acceleration inputs, and the `imuSensor` and `magparams` properties to model magnetometer readings.



Convert to Sensor Frame

The ground-truth acceleration is converted from the local frame to the sensor frame using the ground-truth orientation:

$$a = (orientation)(totalAcc)^T$$

If the orientation is input in quaternion form, it is converted to a rotation matrix before processing.

Bulk Model

The ground-truth acceleration in the sensor frame, a, passes through the bulk model, which adds axes misalignment and bias:

$$b = \left(\begin{bmatrix} 1 & \frac{\alpha_2}{100} & \frac{\alpha_3}{100} \\ \frac{\alpha_1}{100} & 1 & \frac{\alpha_3}{100} \\ \frac{\alpha_1}{100} & \frac{\alpha_2}{100} & 1 \end{bmatrix} \left(a^T\right) \right)^T + \text{ConstantBias}$$

where ConstantBias is a property of `magparams`, and α_1, α_2, and α_3 are given by the first, second, and third elements of the AxesMisalignment property of `magparams`.

Bias Instability Drift

The bias instability drift is modeled as white noise biased and then filtered:

$$\beta_1 = h_1 * (w)(\text{BiasInstability})$$

where BiasInstability is a property of `magparams` and h_1 is a filter defined by the SampleRate property:

$$H_1(z) = \frac{1}{1 - \frac{1}{2}z^{-1}}$$

White Noise Drift

White noise drift is modeled by multiplying elements of the white noise random stream by the standard deviation:

$$\beta_2 = (w)\left(\sqrt{\frac{\text{SampleRate}}{2}}\right)(\text{NoiseDensity})$$

where SampleRate is an `imuSensor` property, and NoiseDensity is an `magparams` property. The elements of w are random numbers given by settings of the `imuSensor` random stream.

Random Walk Drift

The random walk drift is modeled by biasing elements of the white noise random stream and then filtering:

$$\beta_3 = h_2 * (w)\left(\frac{\text{RandomWalk}}{\sqrt{\frac{\text{SampleRate}}{2}}}\right)$$

where RandomWalk is a property of `magparams`, SampleRate is a property of `imuSensor`, and h_2 is a filter defined as:

$$H_2(z) = \frac{1}{1 - z^{-1}}$$

Environmental Drift Noise

The environmental drift noise is modeled by multiplying the temperature difference from a standard with the temperature bias:

$$\Delta_e = (\text{Temperature} - 25)(\text{TemperatureBias})$$

where Temperature is a property of `imuSensor`, and TemperatureBias is a property of `magparams`. The constant 25 corresponds to a standard temperature.

Scale Factor Error Model

The temperature scale factor error is modeled as:

$$scaleFactorError = 1 + \left(\frac{\text{Temperature} - 25}{100}\right)(\text{TemperatureScaleFactor})$$

where Temperature is a property of `imuSensor`, and TemperatureScaleFactor is a property of `magparams`. The constant 25 corresponds to a standard temperature.

Quantization Model

The quantization is modeled by first saturating the continuous signal model:

$$e = \begin{cases} \text{MeasurementRange} & \text{if } d > \text{MeasurementRange} \\ -\text{MeasurementRange} & \text{if } -d > \text{MeasurementRange} \\ d & \text{else} \end{cases}$$

and then setting the resolution:

$$magReadings = (\text{Resolution})\left(\text{round}\left(\frac{e}{\text{Resolution}}\right)\right)$$

where MeasurementRange is a property of `magparams`.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

The object functions, `perturbations` and `perturb`, do not support code generation.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

Classes

`accelparams` | `gyroparams` | `magparams`

Objects

`gpsSensor`

Introduced in R2019b

insSensor

Inertial navigation system and GNSS/GPS simulation model

Description

The `insSensor` System object models a device that fuses measurements from an inertial navigation system (INS) and global navigation satellite system (GNSS) such as a GPS, and outputs the fused measurements.

To output fused INS and GNSS measurements:

1. Create the `insSensor` object and set its properties.
2. Call the object with arguments, as if it were a function.

To learn more about how System objects work, see What Are System Objects?

Creation

Syntax

```
INS = insSensor
INS = insSensor(Name,Value)
```

Description

`INS = insSensor` returns a System object, `INS`, that models a device that outputs measurements from an INS and GNSS.

`INS = insSensor(Name,Value)` sets properties on page 2-448 using one or more name-value pairs. Unspecified properties have default values. Enclose each property name in quotes.

Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see System Design in MATLAB Using System Objects.

`MountingLocation` — Location of sensor on platform (m)
`[0 0 0]` (default) | three-element real-valued vector of form [*x y z*]

Location of the sensor on the platform, in meters, specified as a three-element real-valued vector of the form [*x y z*]. The vector defines the offset of the sensor origin from the origin of the platform.

Tunable: Yes

Data Types: `single` | `double`

`RollAccuracy` — Accuracy of roll measurement (deg)
`0.2` (default) | nonnegative real scalar

Accuracy of the roll measurement of the sensor body, in degrees, specified as a nonnegative real scalar.

Roll is the rotation around the *x*-axis of the sensor body. Roll noise is modeled as a white noise process. `RollAccuracy` sets the standard deviation of the roll measurement noise.

Tunable: Yes

Data Types: `single` | `double`

`PitchAccuracy` — Accuracy of pitch measurement (deg)
`0.2` (default) | nonnegative real scalar

Accuracy of the pitch measurement of the sensor body, in degrees, specified as a nonnegative real scalar.

Pitch is the rotation around the *y*-axis of the sensor body. Pitch noise is modeled as a white noise process. `PitchAccuracy` defines the standard deviation of the pitch measurement noise.

Tunable: Yes

Data Types: `single` | `double`

`YawAccuracy` — Accuracy of yaw measurement (deg)
`1` (default) | nonnegative real scalar

Accuracy of the yaw measurement of the sensor body, in degrees, specified as a nonnegative real scalar.

Yaw is the rotation around the *z*-axis of the sensor body. Yaw noise is modeled as a white noise process. `YawAccuracy` defines the standard deviation of the yaw measurement noise.

Tunable: Yes

Data Types: `single` | `double`

`PositionAccuracy` — Accuracy of position measurement (m)
`[1 1 1]` (default) | nonnegative real scalar | three-element real-valued vector

Accuracy of the position measurement of the sensor body, in meters, specified as a nonnegative real scalar or a three-element real-valued vector. The elements of the vector set the accuracy of the *x*-, *y*-, and *z*-position measurements, respectively. If you specify `PositionAccuracy` as a scalar value, then the object sets the accuracy of all three positions to this value.

Position noise is modeled as a white noise process. `PositionAccuracy` defines the standard deviation of the position measurement noise.

Tunable: Yes

Data Types: `single` | `double`

`VelocityAccuracy` — Accuracy of velocity measurement (m/s)
`0.05` (default) | nonnegative real scalar

Accuracy of the velocity measurement of the sensor body, in meters per second, specified as a nonnegative real scalar.

Velocity noise is modeled as a white noise process. `VelocityAccuracy` defines the standard deviation of the velocity measurement noise.

Tunable: Yes

Data Types: `single` | `double`

`AccelerationAccuracy` — Accuracy of acceleration measurement (m/s^2)

`0` (default) | nonnegative real scalar

Accuracy of the acceleration measurement of the sensor body, in meters per second, specified as a nonnegative real scalar.

Acceleration noise is modeled as a white noise process. `AccelerationAccuracy` defines the standard deviation of the acceleration measurement noise.

Tunable: Yes

Data Types: `single` | `double`

`AngularVelocityAccuracy` — Accuracy of angular velocity measurement (deg/s)

`0` (default) | nonnegative real scalar

Accuracy of the angular velocity measurement of the sensor body, in meters per second, specified as a nonnegative real scalar.

Angular velocity is modeled as a white noise process. `AngularVelocityAccuracy` defines the standard deviation of the acceleration measurement noise.

Tunable: Yes

Data Types: `single` | `double`

`TimeInput` — Enable input of simulation time

`false` or `0` (default) | `true` or `1`

Enable input of simulation time, specified as a logical `0` (`false`) or `1` (`true`). Set this property to `true` to input the simulation time by using the `simTime` argument.

Tunable: No

Data Types: `logical`

`HasGNSSFix` — Enable GNSS fix

`true` or `1` (default) | `false` or `0`

Enable GNSS fix, specified as a logical `1` (`true`) or `0` (`false`). Set this property to `false` to simulate the loss of a GNSS receiver fix. When a GNSS receiver fix is lost, position measurements drift at a rate specified by the `PositionErrorFactor` property.

Tunable: Yes

Dependencies

To enable this property, set `TimeInput` to `true`.

Data Types: `logical`

`PositionErrorFactor` — Position error factor without GNSS fix

`[0 0 0]` (default) | nonnegative scalar | 1-by-3 vector of scalars

Position error factor without GNSS fix, specified as a scalar or a 1-by-3 vector of scalars.

When the `HasGNSSFix` property is set to `false`, the position error grows at a quadratic rate due to constant bias in the accelerometer. The position error for a position component $E(t)$ can be expressed as $E(t) = 1/2\alpha t^2$, where α is the position error factor for the corresponding component and t is the time since the GNSS fix is lost. While running, the object computes t based on the `simTime` input. The computed $E(t)$ values for the x, y, and z components are added to the corresponding position components of the `gTruth` input.

Tunable: Yes

Dependencies

To enable this property, set `TimeInput` to `true` and `HasGNSSFix` to `false`.

Data Types: `single` | `double`

`RandomStream` — Random number source

`'Global stream'` (default) | `'mt19937ar with seed'`

Random number source, specified as one of these options:

- `'Global stream'` –– Generate random numbers using the current global random number stream.
- `'mt19937ar with seed'` –– Generate random numbers using the mt19937ar algorithm, with the seed specified by the `Seed` property.

Data Types: `char` | `string`

Seed — Initial seed

67 (default) | nonnegative integer

Initial seed of the mt19937ar random number generator algorithm, specified as a nonnegative integer.

Dependencies

To enable this property, set `RandomStream` to `'mt19937ar with seed'`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Usage

Syntax

```
measurement = INS(gTruth)
measurement = INS(gTruth,simTime)
```

Description

`measurement = INS(gTruth)` models the data received from an INS sensor reading and GNSS sensor reading. The output measurement is based on the inertial ground-truth state of the sensor body, `gTruth`.

`measurement = INS(gTruth,simTime)` additionally specifies the time of simulation, `simTime`. To enable this syntax, set the `TimeInput` property to `true`.

Input Arguments

`gTruth` — Inertial ground-truth state of sensor body

structure

Inertial ground-truth state of sensor body, in local Cartesian coordinates, specified as a structure containing these fields:

Field	Description
`'Position'`	Position, in meters, specified as a real, finite *N*-by-3 matrix of [*x y z*] vectors. *N* is the number of samples in the current frame.
`'Velocity'`	Velocity (*v*), in meters per second, specified as a real, finite *N*-by-3 matrix of [v_x v_y v_z] vector. *N* is the number of samples in the current frame.
`'Orientation'`	Orientation with respect to the local Cartesian coordinate system, specified as one of these options: • *N*-element column vector of `quaternion` objects • 3-by-3-by-*N* array of rotation matrices • *N*-by-3 matrix of [x_{roll} y_{pitch} z_{yaw}] angles in degrees Each quaternion or rotation matrix is a frame rotation from the local Cartesian coordinate system to the current sensor body coordinate system. *N* is the number of samples in the current frame.
`'Acceleration'`	Acceleration (*a*), in meters per second squared, specified as a real, finite *N*-by-3 matrix of [a_x a_y a_z] vectors. *N* is the number of samples in the current frame.
`'AngularVelocity'`	Angular velocity (ω), in degrees per second squared, specified as a real, finite *N*-by-3 matrix of [ω_x ω_y ω_z] vectors. *N* is the number of samples in the current frame.

The field values must be of type `double` or `single`.

The `Position`, `Velocity`, and `Orientation` fields are required. The other fields are optional.

Example: `struct('Position',[0 0 0],'Velocity',[0 0 0],'Orientation',quaternion([1 0 0 0]))`

`simTime` — Simulation time

nonnegative real scalar

Simulation time, in seconds, specified as a nonnegative real scalar.

Data Types: `single` | `double`

Output Arguments

`measurement` — Measurement of sensor body motion

structure

Measurement of the sensor body motion, in local Cartesian coordinates, returned as a structure containing these fields:

Field	Description
`'Position'`	Position, in meters, specified as a real, finite *N*-by-3 matrix of [*x* *y* *z*] vectors. *N* is the number of samples in the current frame.
`'Velocity'`	Velocity (*v*), in meters per second, specified as a real, finite *N*-by-3 matrix of $[v_x\ v_y\ v_z]$ vector. *N* is the number of samples in the current frame.
`'Orientation'`	Orientation with respect to the local Cartesian coordinate system, specified as one of these options: • *N*-element column vector of `quaternion` objects • 3-by-3-by-*N* array of rotation matrices • *N*-by-3 matrix of $[x_{roll}\ y_{pitch}\ z_{yaw}]$ angles in degrees Each quaternion or rotation matrix is a frame rotation from the local Cartesian coordinate system to the current sensor body coordinate system. *N* is the number of samples in the current frame.
`'Acceleration'`	Acceleration (*a*), in meters per second squared, specified as a real, finite *N*-by-3 matrix of $[a_x\ a_y\ a_z]$ vectors. *N* is the number of samples in the current frame.
`'AngularVelocity'`	Angular velocity (ω), in degrees per second squared, specified as a real, finite *N*-by-3 matrix of $[\omega_x\ \omega_y\ \omega_z]$ vectors. *N* is the number of samples in the current frame.

The returned field values are of type `double` or `single` and are of the same type as the corresponding field values in the `gTruth` input.

Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

Specific to `insSensor`

perturbations	Perturbation defined on object
perturb	Apply perturbations to object

Common to All System Objects

step	Run System object algorithm
clone	Create duplicate System object
isLocked	Determine if System object is in use
reset	Reset internal states of System object
release	Release resources and allow changes to System object property values and input characteristics

Examples

Generate INS Measurements from Stationary Input

Create a motion structure that defines a stationary position at the local north-east-down (NED) origin. Because the platform is stationary, you need to define only a single sample. Assume the ground-truth motion is sampled for 10 seconds with a 100 Hz sample rate. Create a default `insSensor` System object™. Preallocate variables to hold output from the `insSensor` object.

```
Fs = 100;
duration = 10;
numSamples = Fs*duration;

motion = struct( ...
    'Position',zeros(1,3), ...
    'Velocity',zeros(1,3), ...
    'Orientation',ones(1,1,'quaternion'));

INS = insSensor;

positionMeasurements = zeros(numSamples,3);
velocityMeasurements = zeros(numSamples,3);
orientationMeasurements = zeros(numSamples,1,'quaternion');
```

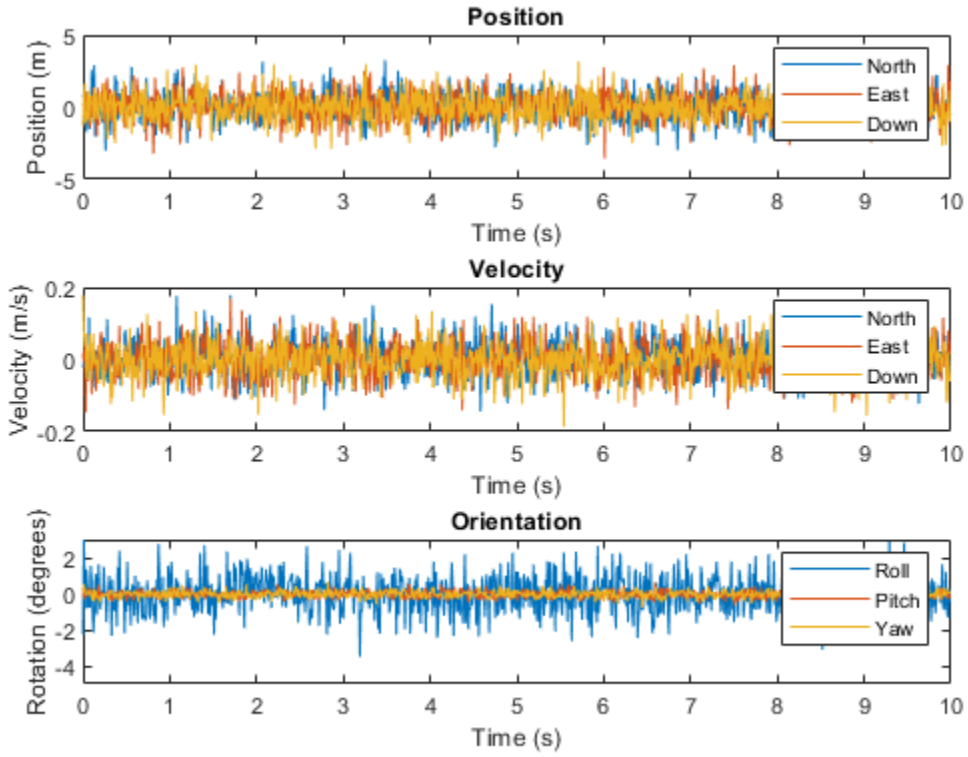
In a loop, call `INS` with the stationary motion structure to return the position, velocity, and orientation measurements in the local NED coordinate system. Log the position, velocity, and orientation measurements.

```
for i = 1:numSamples

    measurements = INS(motion);

    positionMeasurements(i,:) = measurements.Position;
    velocityMeasurements(i,:) = measurements.Velocity;
```

```
    orientationMeasurements(i) = measurements.Orientation;

end
```

Convert the orientation from quaternions to Euler angles for visualization purposes. Plot the position, velocity, and orientation measurements over time.

```
orientationMeasurements = eulerd(orientationMeasurements,'ZYX','frame');

t = (0:(numSamples-1))/Fs;

subplot(3,1,1)
plot(t,positionMeasurements)
title('Position')
xlabel('Time (s)')
ylabel('Position (m)')
legend('North','East','Down')

subplot(3,1,2)
plot(t,velocityMeasurements)
title('Velocity')
xlabel('Time (s)')
ylabel('Velocity (m/s)')
legend('North','East','Down')

subplot(3,1,3)
plot(t,orientationMeasurements)
title('Orientation')
xlabel('Time (s)')
ylabel('Rotation (degrees)')
legend('Roll', 'Pitch', 'Yaw')
```



Generate INS Measurements for a Turning Platform

Generate INS measurements using the `insSensor` System object™. Use `waypointTrajectory` to generate the ground-truth path.

Specify a ground-truth orientation that begins with the sensor body *x*-axis aligned with North and ends with the sensor body *x*-axis aligned with East. Specify waypoints for an arc trajectory and a time-of-arrival vector for the corresponding waypoints. Use a 100 Hz sample rate. Create a `waypointTrajectory` System object with the waypoint constraints, and set `SamplesPerFrame` so that the entire trajectory is output with one call.

```
eulerAngles = [0,0,0; ...
               0,0,0; ...
               90,0,0; ...
               90,0,0];
orientation = quaternion(eulerAngles,'eulerd','ZYX','frame');

r = 20;
waypoints = [0,0,0; ...
             100,0,0; ...
             100+r,r,0; ...
             100+r,100+r,0];

toa = [0,10,10+(2*pi*r/4),20+(2*pi*r/4)];
```

```
Fs = 100;
numSamples = floor(Fs*toa(end));

path = waypointTrajectory('Waypoints',waypoints, ...
    'TimeOfArrival',toa, ...
    'Orientation',orientation, ...
    'SampleRate',Fs, ...
    'SamplesPerFrame',numSamples);
```

Create an `insSensor` System object to model receiving INS data. Set the `PositionAccuracy` to `0.1`.

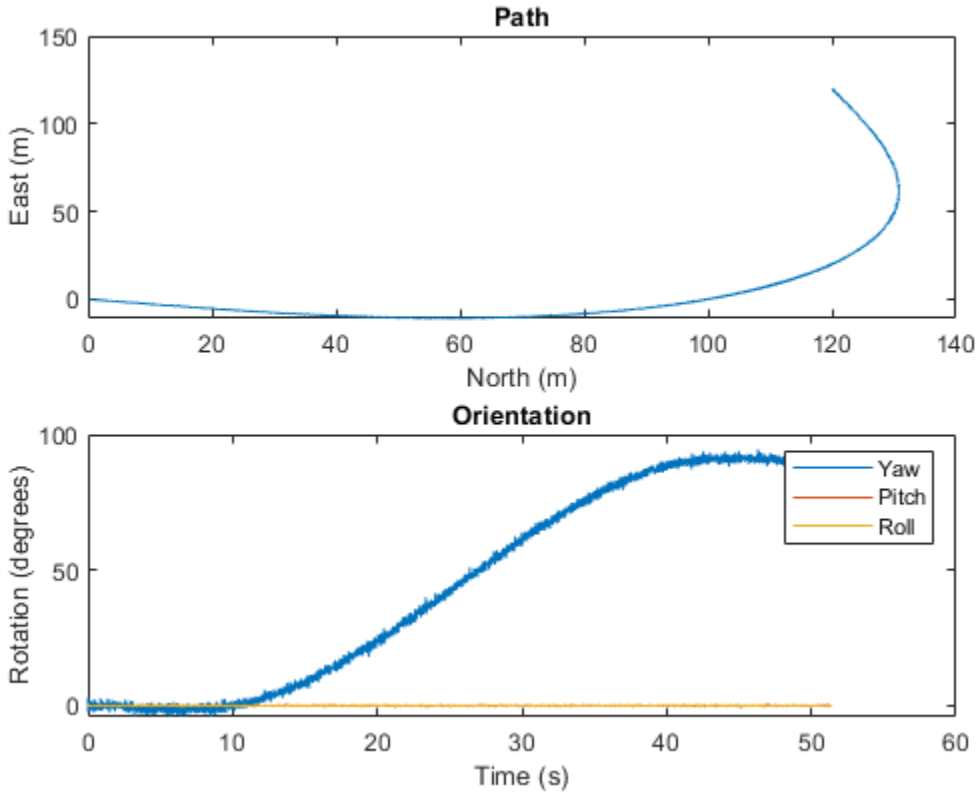
```
ins = insSensor('PositionAccuracy',0.1);
```

Call the waypoint trajectory object, `path`, to generate the ground-truth motion. Call the INS simulator, `ins`, with the ground-truth motion to generate INS measurements.

```
[motion.Position,motion.Orientation,motion.Velocity] = path();
insMeas = ins(motion);
```

Convert the orientation returned by `ins` to Euler angles in degrees for visualization purposes. Plot the full path and orientation over time.

```
orientationMeasurementEuler = eulerd(insMeas.Orientation,'ZYX','frame');

subplot(2,1,1)
plot(insMeas.Position(:,1),insMeas.Position(:,2));
title('Path')
xlabel('North (m)')
ylabel('East (m)')

subplot(2,1,2)
t = (0:(numSamples-1)).'/Fs;
plot(t,orientationMeasurementEuler(:,1), ...
     t,orientationMeasurementEuler(:,2), ...
     t,orientationMeasurementEuler(:,3));
title('Orientation')
legend('Yaw','Pitch','Roll')
xlabel('Time (s)')
ylabel('Rotation (degrees)')
```



Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

The object functions, `perturbations` and `perturb`, do not support code generation.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

Objects

`imuSensor` | `gpsSensor`

Introduced in R2020b

loadparams

Load sensor parameters from JSON file

Syntax

`loadparams(sensor,file,PN)`

Description

`loadparams(sensor,file,PN)` configures the `imuSensor` object, `sensor`, to match the parameters in the `PN` part of a JSON file, `File`.

Examples

Load Pre-defined Parameters in `imuSensor`

Create an `imuSensor` system object.

```
s = imuSensor;
```

Load a JSON file.

```
fn = fullfile(matlabroot,'toolbox','shared',...
    'positioning','positioningdata','generic.json');
```

Here is a screen shot of the JSON file with some parts collapsed.

Configure the object as a 6-axis sensor.

```
loadparams(s,fn,'GenericLowCost6Axis')
s
```

```
s =
  imuSensor with properties:

          IMUType: 'accel-gyro'
       SampleRate: 100
      Temperature: 25
    Accelerometer: [1x1 accelparams]
        Gyroscope: [1x1 gyroparams]
     RandomStream: 'Global stream'
```

Configure the object as a 9-axis sensor.

```
loadparams(s,fn,'GenericLowCost9Axis')
s
```

```
s =
  imuSensor with properties:

          IMUType: 'accel-gyro-mag'
       SampleRate: 100
      Temperature: 25
    MagneticField: [27.5550 -2.4169 -16.0849]
    Accelerometer: [1x1 accelparams]
        Gyroscope: [1x1 gyroparams]
     Magnetometer: [1x1 magparams]
     RandomStream: 'Global stream'
```

Input Arguments

`sensor` — IMU sensor

`imuSensor` object

IMU sensor, specified as an `imuSensor` system object.

`file` — JSON file

`.json` file

JavaScript Object Notation (JSON) format file, specified as a `.json` file.

`PN` — Part name

`string`

Part name in a JSON file, specified as a string.

See Also

`imuSensor`

Introduced in R2020a

kinematicTrajectory

Rate-driven trajectory generator

Description

The `kinematicTrajectory` System object generates trajectories using specified acceleration and angular velocity.

To generate a trajectory from rates:

1. Create the `kinematicTrajectory` object and set its properties.
2. Call the object with arguments, as if it were a function.

To learn more about how System objects work, see What Are System Objects?

Creation

Syntax

```
trajectory = kinematicTrajectory
trajectory = kinematicTrajectory(Name,Value)
```

Description

`trajectory = kinematicTrajectory` returns a System object, `trajectory`, that generates a trajectory based on acceleration and angular velocity.

`trajectory = kinematicTrajectory(Name,Value)` sets each property `Name` to the specified `Value`. Unspecified properties have default values.

Example: `trajectory = kinematicTrajectory('SampleRate',200,'Position',[0,1,10])` creates a kinematic trajectory System object, `trajectory`, with a sample rate of 200 Hz and the initial position set to [0,1,10].

Properties

If a property is *tunable*, you can change its value at any time.

`SampleRate` — Sample rate of trajectory (Hz)
`100` (default) | positive scalar

Sample rate of trajectory in Hz, specified as a positive scalar.

Tunable: Yes

Data Types: `single` | `double`

`Position` — Position state in local navigation coordinate system (m)
`[0 0 0]` (default) | 3-element row vector

Position state in the local navigation coordinate system in meters, specified as a three-element row vector.

Tunable: Yes

Data Types: `single` | `double`

`Velocity` — Velocity state in local navigation coordinate system (m/s)

`[0 0 0]` (default) | 3-element row vector

Velocity state in the local navigation coordinate system in m/s, specified as a three-element row vector.

Tunable: Yes

Data Types: `single` | `double`

`Orientation` — Orientation state in local navigation coordinate system

`quaternion(1,0,0,0)` (default) | scalar quaternion | 3-by-3 real matrix

Orientation state in the local navigation coordinate system, specified as a scalar quaternion or 3-by-3 real matrix. The orientation is a frame rotation from the local navigation coordinate system to the current body frame.

Tunable: Yes

Data Types: `quaternion` | `single` | `double`

`AccelerationSource` — Source of acceleration state

`'Input'` (default) | `'Property'`

Source of acceleration state, specified as `'Input'` or `'Property'`.

- `'Input'` –– specify acceleration state as an input argument to the kinematic trajectory object
- `'Property'` –– specify acceleration state by setting the `Acceleration` property

Tunable: No

Data Types: `char` | `string`

`Acceleration` — Acceleration state (m/s^2)

`[0 0 0]` (default) | three-element row vector

Acceleration state in m/s^2, specified as a three-element row vector.

Tunable: Yes

Dependencies

To enable this property, set AccelerationSource to `'Property'`.

Data Types: `single` | `double`

`AngularVelocitySource` — Source of angular velocity state

`'Input'` (default) | `'Property'`

Source of angular velocity state, specified as `'Input'` or `'Property'`.

- `'Input'` –– specify angular velocity state as an input argument to the kinematic trajectory object
- `'Property'` –– specify angular velocity state by setting the `AngularVelocity` property

Tunable: No

Data Types: `char` | `string`

`AngularVelocity` — Angular velocity state (rad/s)
`[0 0 0]` (default) | three-element row vector

Angular velocity state in rad/s, specified as a three-element row vector.

Tunable: Yes

Dependencies

To enable this property, set AngularVelocitySource to `'Property'`.

Data Types: `single` | `double`

`SamplesPerFrame` — Number of samples per output frame
1 (default) | positive integer

Number of samples per output frame, specified as a positive integer.

Tunable: No

Dependencies

To enable this property, set AngularVelocitySource to `'Property'` and AccelerationSource to `'Property'`.

Data Types: `single` | `double`

Usage

Syntax

```
[position,orientation,velocity,acceleration,angularVelocity] = trajectory(
bodyAcceleration,bodyAngularVelocity)
[position,orientation,velocity,acceleration,angularVelocity] = trajectory(
bodyAngularVelocity)
[position,orientation,velocity,acceleration,angularVelocity] = trajectory(
bodyAcceleration)
[position,orientation,velocity,acceleration,angularVelocity] = trajectory()
```

Description

`[position,orientation,velocity,acceleration,angularVelocity] = trajectory(bodyAcceleration,bodyAngularVelocity)` outputs the trajectory state and then updates the trajectory state based on `bodyAcceleration` and `bodyAngularVelocity`.

This syntax is only valid if `AngularVelocitySource` is set to `'Input'` and `AccelerationSource` is set to `'Input'`.

`[position,orientation,velocity,acceleration,angularVelocity] = trajectory(bodyAngularVelocity)` outputs the trajectory state and then updates the trajectory state based on `bodyAngularAcceleration`.

This syntax is only valid if `AngularVelocitySource` is set to `'Input'` and `AccelerationSource` is set to `'Property'`.

`[position,orientation,velocity,acceleration,angularVelocity] = trajectory(bodyAcceleration)` outputs the trajectory state and then updates the trajectory state based on `bodyAcceleration`.

This syntax is only valid if `AngularVelocitySource` is set to `'Property'` and `AccelerationSource` is set to `'Input'`.

`[position,orientation,velocity,acceleration,angularVelocity] = trajectory()` outputs the trajectory state and then updates the trajectory state.

This syntax is only valid if `AngularVelocitySource` is set to `'Property'` and `AccelerationSource` is set to `'Property'`.

Input Arguments

`bodyAcceleration` — Acceleration in body coordinate system (m/s²)

N-by-3 matrix

Acceleration in the body coordinate system in meters per second squared, specified as an *N*-by-3 matrix.

N is the number of samples in the current frame.

`bodyAngularVelocity` — Angular velocity in body coordinate system (rad/s)

N-by-3 matrix

Angular velocity in the body coordinate system in radians per second, specified as an *N*-by-3 matrix.

N is the number of samples in the current frame.

Output Arguments

`position` — Position in local navigation coordinate system (m)

N-by-3 matrix

Position in the local navigation coordinate system in meters, returned as an *N*-by-3 matrix.

N is the number of samples in the current frame.

Data Types: `single` | `double`

`orientation` — Orientation in local navigation coordinate system

N-element quaternion column vector | 3-by-3-by-*N* real array

Orientation in the local navigation coordinate system, returned as an *N*-by-1 quaternion column vector or a 3-by-3-by-*N* real array. Each quaternion or 3-by-3 rotation matrix is a frame rotation from the local navigation coordinate system to the current body coordinate system.

N is the number of samples in the current frame.

Data Types: `single` | `double`

`velocity` — Velocity in local navigation coordinate system (m/s)
N-by-3 matrix

Velocity in the local navigation coordinate system in meters per second, returned as an *N*-by-3 matrix.

N is the number of samples in the current frame.

Data Types: `single` | `double`

`acceleration` — Acceleration in local navigation coordinate system (m/s²)
N-by-3 matrix

Acceleration in the local navigation coordinate system in meters per second squared, returned as an *N*-by-3 matrix.

N is the number of samples in the current frame.

Data Types: `single` | `double`

`angularVelocity` — Angular velocity in local navigation coordinate system (rad/s)
N-by-3 matrix

Angular velocity in the local navigation coordinate system in radians per second, returned as an *N*-by-3 matrix.

N is the number of samples in the current frame.

Data Types: `single` | `double`

Object Functions

Specific to `kinematicTrajectory`

perturbations	Perturbation defined on object
perturb	Apply perturbations to object

Common to All System Objects

step	Run System object algorithm

Examples

Create Default `kinematicTrajectory`

Create a default `kinematicTrajectory` System object™ and explore the relationship between input, properties, and the generated trajectories.

```
trajectory = kinematicTrajectory
```

```
trajectory =
  kinematicTrajectory with properties:

               SampleRate: 100
```

```
                Position: [0 0 0]
             Orientation: [1x1 quaternion]
                Velocity: [0 0 0]
       AccelerationSource: 'Input'
    AngularVelocitySource: 'Input'
```

By default, the `kinematicTrajectory` object has an initial position of [0 0 0] and an initial velocity of [0 0 0]. Orientation is described by a quaternion one (1 + 0i + 0j + 0k).

The `kinematicTrajectory` object maintains a visible and writable state in the properties `Position`, `Velocity`, and `Orientation`. When you call the object, the state is output and then updated.

For example, call the object by specifying an acceleration and angular velocity relative to the body coordinate system.

```
bodyAcceleration = [5,5,0];
bodyAngularVelocity = [0,0,1];
[position,orientation,velocity,acceleration,angularVelocity] = trajectory(bodyAcceleration,bodyAn
```

```
position = 1×3

     0     0     0


orientation = quaternion
     1 + 0i + 0j + 0k


velocity = 1×3

     0     0     0


acceleration = 1×3

     5     5     0


angularVelocity = 1×3

     0     0     1
```

The position, orientation, and velocity output from the `trajectory` object correspond to the state reported by the properties before calling the object. The `trajectory` state is updated after being called and is observable from the properties:

```
trajectory
```

```
trajectory =
  kinematicTrajectory with properties:

               SampleRate: 100
                 Position: [2.5000e-04 2.5000e-04 0]
              Orientation: [1x1 quaternion]
                 Velocity: [0.0500 0.0500 0]
       AccelerationSource: 'Input'
```

```
    AngularVelocitySource: 'Input'
```

The `acceleration` and `angularVelocity` output from the `trajectory` object correspond to the `bodyAcceleration` and `bodyAngularVelocity`, except that they are returned in the navigation coordinate system. Use the `orientation` output to rotate `acceleration` and `angularVelocity` to the body coordinate system and verify they are approximately equivalent to `bodyAcceleration` and `bodyAngularVelocity`.

```
rotatedAcceleration = rotatepoint(orientation,acceleration)
```

```
rotatedAcceleration = 1×3

     5     5     0
```

```
rotatedAngularVelocity = rotatepoint(orientation,angularVelocity)
```

```
rotatedAngularVelocity = 1×3

     0     0     1
```

The `kinematicTrajectory` System object™ enables you to modify the trajectory state through the properties. Set the position to [0,0,0] and then call the object with a specified acceleration and angular velocity in the body coordinate system. For illustrative purposes, clone the `trajectory` object before modifying the `Position` property. Call both objects and observe that the positions diverge.

```
trajectoryClone = clone(trajectory);
trajectory.Position = [0,0,0];
```

```
position = trajectory(bodyAcceleration,bodyAngularVelocity)
```

```
position = 1×3

     0     0     0
```

```
clonePosition = trajectoryClone(bodyAcceleration,bodyAngularVelocity)
```

```
clonePosition = 1×3
10^-3 ×

    0.2500    0.2500         0
```

Create Oscillating Trajectory

This example shows how to create a trajectory oscillating along the North axis of a local NED coordinate system using the `kinematicTrajectory` System object™.

Create a default `kinematicTrajectory` object. The default initial orientation is aligned with the local NED coordinate system.

```
traj = kinematicTrajectory
```

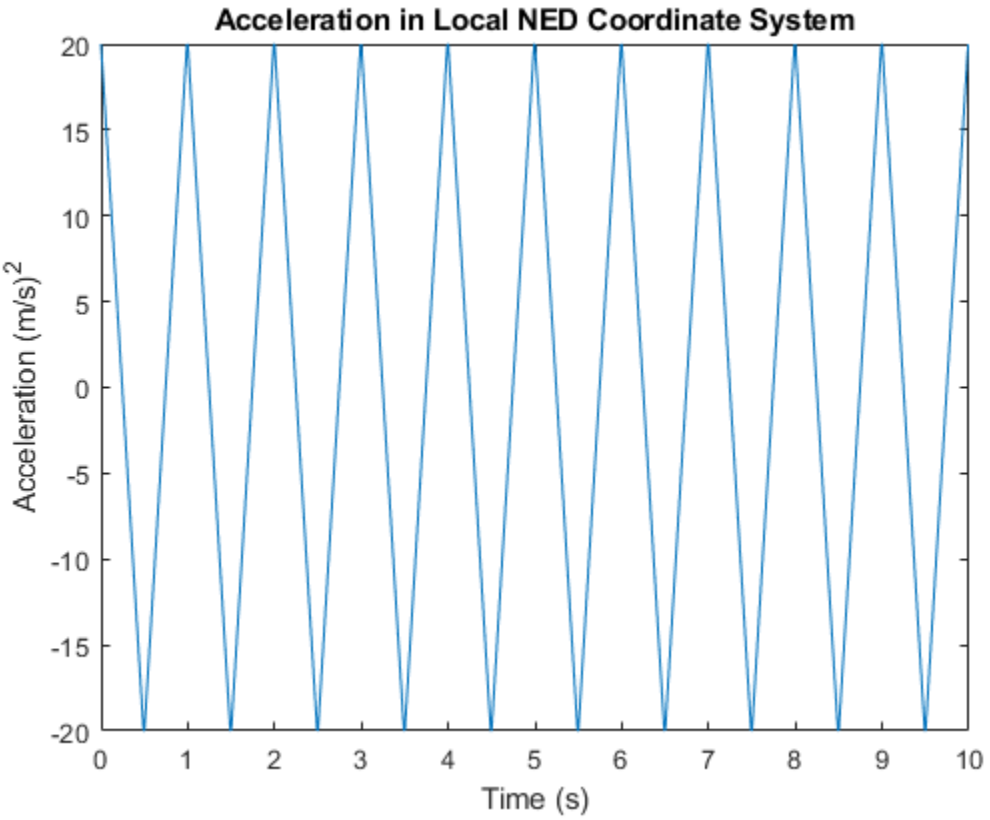
```
traj =

  kinematicTrajectory with properties:

              SampleRate: 100
                Position: [0 0 0]
             Orientation: [1x1 quaternion]
                Velocity: [0 0 0]
       AccelerationSource: 'Input'
    AngularVelocitySource: 'Input'
```

Define a trajectory for a duration of 10 seconds consisting of rotation around the East axis (pitch) and an oscillation along North axis of the local NED coordinate system. Use the default `kinematicTrajectory` sample rate.

```
fs = traj.SampleRate;
duration = 10;

numSamples = duration*fs;

cyclesPerSecond = 1;
samplesPerCycle = fs/cyclesPerSecond;
numCycles = ceil(numSamples/samplesPerCycle);
maxAccel = 20;

triangle = [linspace(maxAccel,1/fs-maxAccel,samplesPerCycle/2), ...
    linspace(-maxAccel,maxAccel-(1/fs),samplesPerCycle/2)]';
oscillation = repmat(triangle,numCycles,1);
oscillation = oscillation(1:numSamples);

accNED = [zeros(numSamples,2),oscillation];

angVelNED = zeros(numSamples,3);
angVelNED(:,2) = 2*pi;
```

Plot the acceleration control signal.

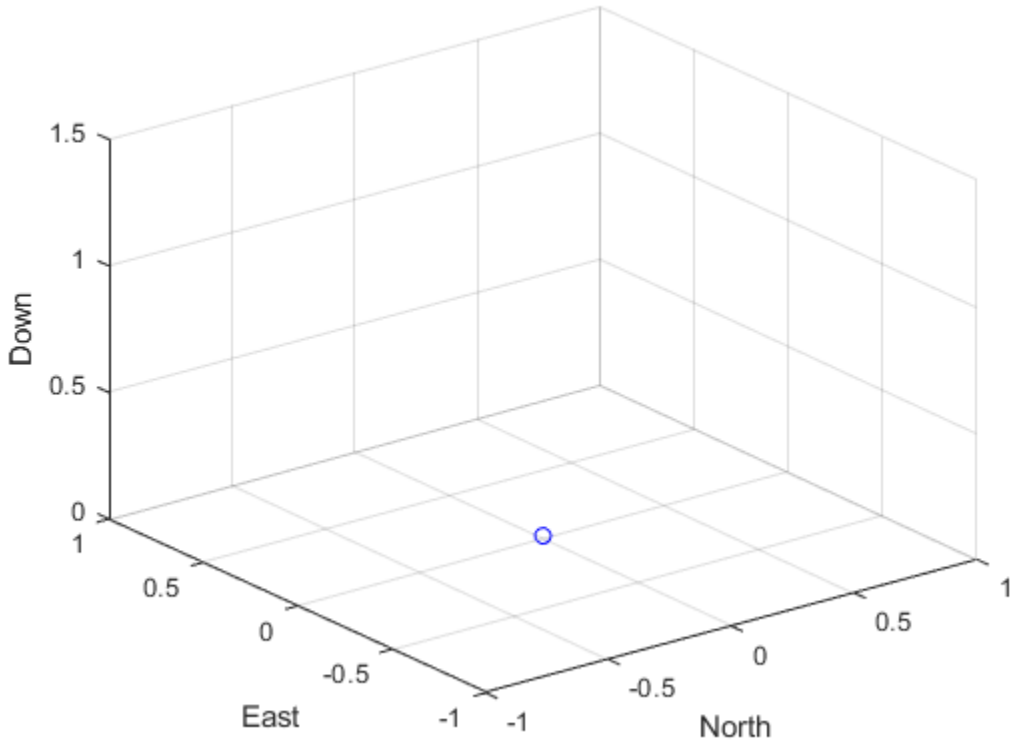
```
timeVector = 0:1/fs:(duration-1/fs);

figure(1)
plot(timeVector,oscillation)
xlabel('Time (s)')
ylabel('Acceleration (m/s)^2')
title('Acceleration in Local NED Coordinate System')
```

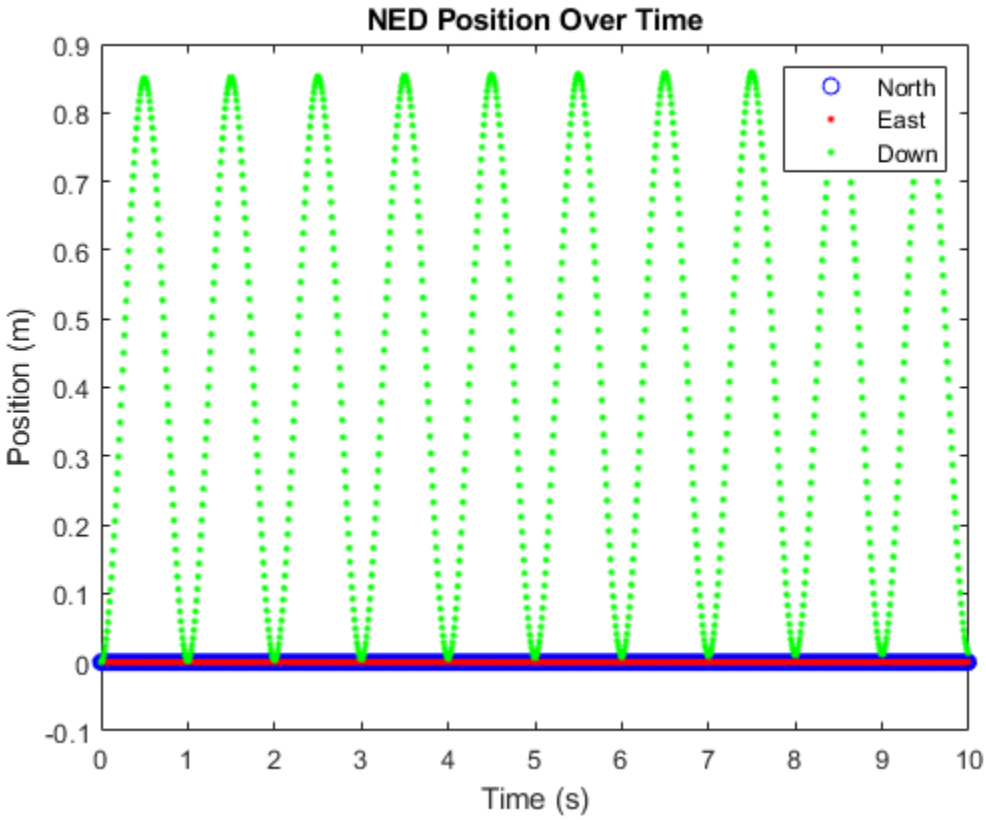


Generate the trajectory sample-by-sample in a loop. The `kinematicTrajectory` System object assumes the acceleration and angular velocity inputs are in the local sensor body coordinate system. Rotate the acceleration and angular velocity control signals from the NED coordinate system to the sensor body coordinate system using `rotateframe` and the `Orientation` state. Update a 3-D plot of the position at each time. Add `pause` to mimic real-time processing. Once the loop is complete, plot the position over time. Rotating the `accNED` and `angVelNED` control signals to the local body coordinate system assures the motion stays along the Down axis.

```
figure(2)
plotHandle = plot3(traj.Position(1),traj.Position(2),traj.Position(3),'bo');
grid on
xlabel('North')
ylabel('East')
zlabel('Down')
axis([-1 1 -1 1 0 1.5])
hold on

q = ones(numSamples,1,'quaternion');
for ii = 1:numSamples
     accBody = rotateframe(traj.Orientation,accNED(ii,:));
     angVelBody = rotateframe(traj.Orientation,angVelNED(ii,:));

    [pos(ii,:),q(ii),vel,ac] = traj(accBody,angVelBody);

    set(plotHandle,'XData',pos(ii,1),'YData',pos(ii,2),'ZData',pos(ii,3))

    pause(1/fs)
```

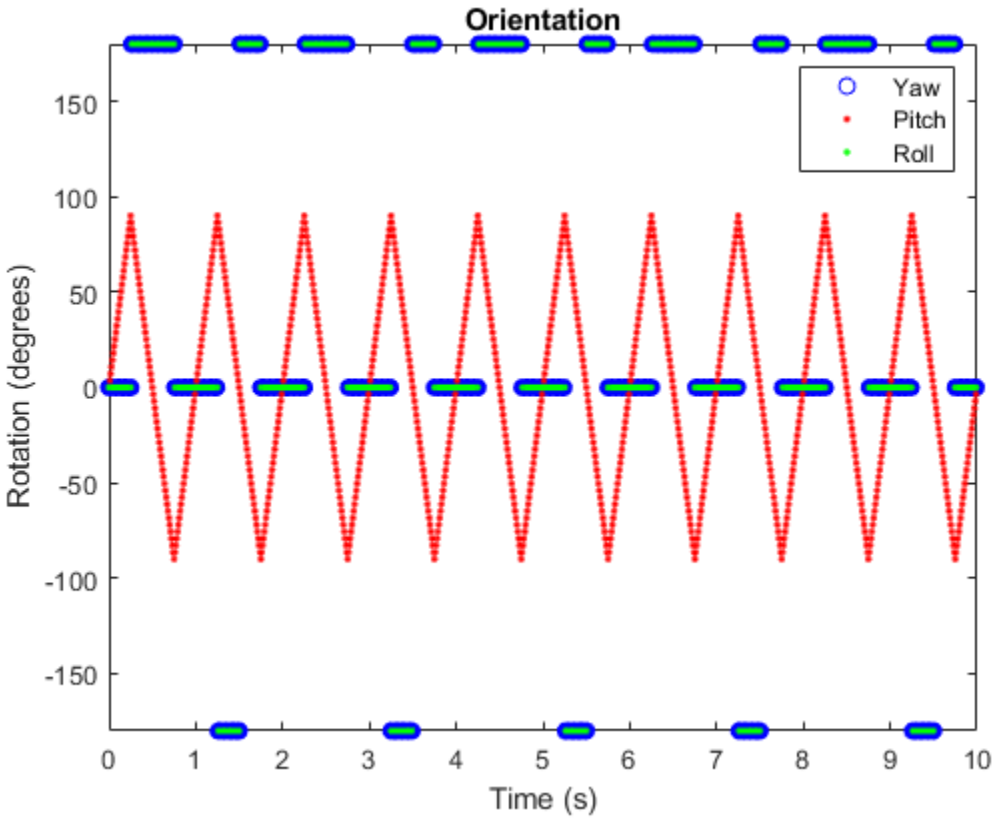
```
end

figure(3)
plot(timeVector,pos(:,1),'bo',...
     timeVector,pos(:,2),'r.',...
     timeVector,pos(:,3),'g.')
xlabel('Time (s)')
ylabel('Position (m)')
title('NED Position Over Time')
legend('North','East','Down')
```





Convert the recorded orientation to Euler angles and plot. Although the orientation of the platform changed over time, the acceleration always acted along the North axis.

```
figure(4)
eulerAngles = eulerd(q,'ZYX','frame');
plot(timeVector,eulerAngles(:,1),'bo',...
     timeVector,eulerAngles(:,2),'r.',...
     timeVector,eulerAngles(:,3),'g.')
axis([0,duration,-180,180])
legend('Yaw','Pitch','Roll')
xlabel('Time (s)')
ylabel('Rotation (degrees)')
title('Orientation')
```



Generate a Coil Trajectory

This example shows how to generate a coil trajectory using the `kinematicTrajectory` System object™.

Create a circular trajectory for a 1000 second duration and a sample rate of 10 Hz. Set the radius of the circle to 5000 meters and the speed to 80 meters per second. Set the climb rate to 100 meters per second and the pitch to 15 degrees. Specify the initial orientation as pointed in the direction of motion.

```
duration = 1000; % seconds
fs = 10;         % Hz
N = duration*fs; % number of samples

radius = 5000;   % meters
speed = 80;      % meters per second
climbRate = 50;  % meters per second
initialYaw = 90; % degrees
pitch = 15;      % degrees

initPos = [radius, 0, 0];
initVel = [0, speed, climbRate];
initOrientation = quaternion([initialYaw,pitch,0],'eulerd','zyx','frame');

trajectory = kinematicTrajectory('SampleRate',fs, ...
```

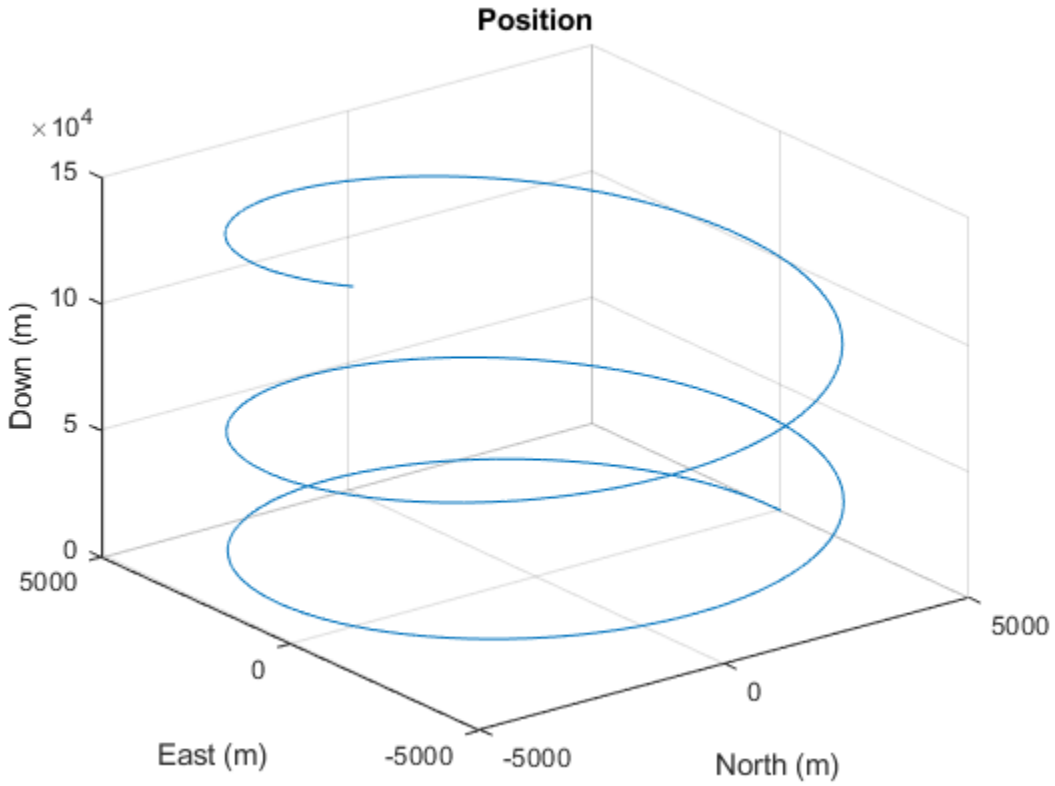
```
    'Velocity',initVel, ...
    'Position',initPos, ...
    'Orientation',initOrientation);
```

Specify a constant acceleration and angular velocity in the body coordinate system. Rotate the body frame to account for the pitch.

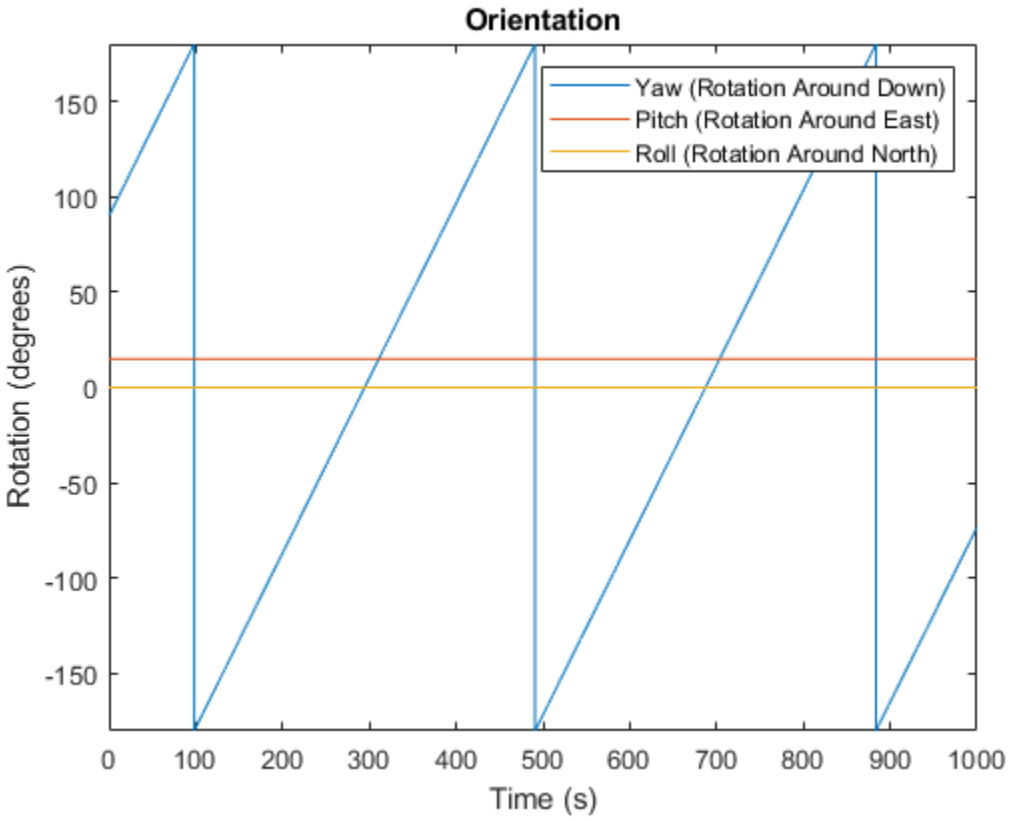
```
accBody = zeros(N,3);
accBody(:,2) = speed^2/radius;
accBody(:,3) = 0.2;

angVelBody = zeros(N,3);
angVelBody(:,3) = speed/radius;

pitchRotation = quaternion([0,pitch,0],'eulerd','zyx','frame');
angVelBody = rotateframe(pitchRotation,angVelBody);
accBody = rotateframe(pitchRotation,accBody);
```

Call `trajectory` with the specified acceleration and angular velocity in the body coordinate system. Plot the position, orientation, and speed over time.

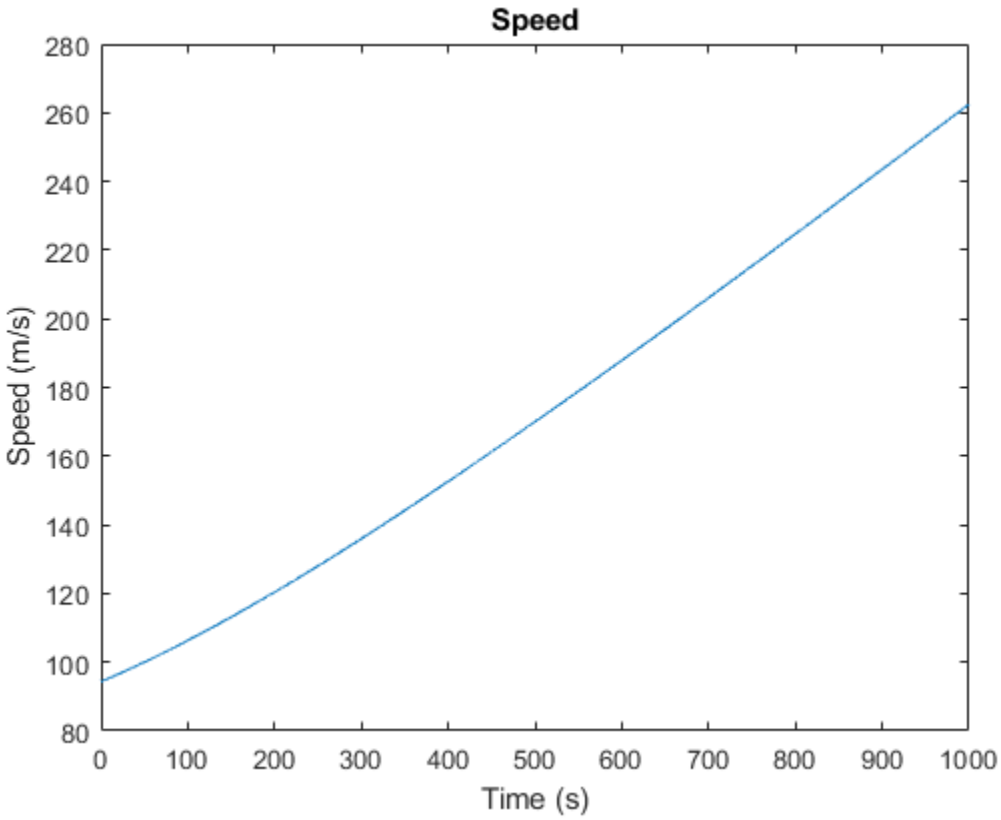
```
[position, orientation, velocity] = trajectory(accBody,angVelBody);

eulerAngles = eulerd(orientation,'ZYX','frame');
speed = sqrt(sum(velocity.^2,2));

timeVector = (0:(N-1))/fs;

figure(1)
plot3(position(:,1),position(:,2),position(:,3))
xlabel('North (m)')
ylabel('East (m)')
zlabel('Down (m)')
title('Position')
grid on
```




```
figure(2)
plot(timeVector,eulerAngles(:,1),...
     timeVector,eulerAngles(:,2),...
     timeVector,eulerAngles(:,3))
axis([0,duration,-180,180])
legend('Yaw (Rotation Around Down)','Pitch (Rotation Around East)','Roll (Rotation Around North)
xlabel('Time (s)')
ylabel('Rotation (degrees)')
title('Orientation')
```




```
figure(3)
plot(timeVector,speed)
xlabel('Time (s)')
ylabel('Speed (m/s)')
title('Speed')
```



Generate Spiraling Circular Trajectory with No Inputs

Define a constant angular velocity and constant acceleration that describe a spiraling circular trajectory.

```
Fs = 100;
r = 10;
speed = 2.5;
initialYaw = 90;

initPos = [r 0 0];
initVel = [0 speed 0];
initOrient = quaternion([initialYaw 0 0], 'eulerd', 'ZYX', 'frame');

accBody = [0 speed^2/r 0.01];
angVelBody = [0 0 speed/r];
```

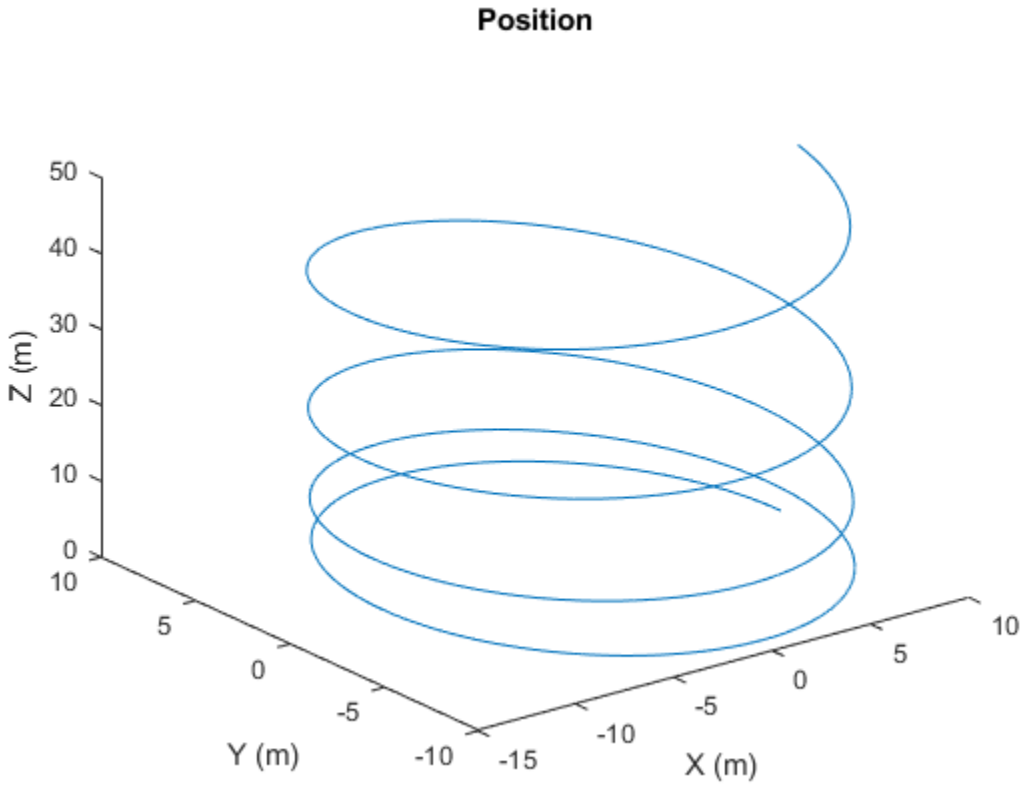
Create a kinematic trajectory object.

```
traj = kinematicTrajectory('SampleRate',Fs, ...
    'Position',initPos, ...
    'Velocity',initVel, ...
    'Orientation',initOrient, ...
    'AccelerationSource','Property', ...
    'Acceleration',accBody, ...
```

```
    'AngularVelocitySource','Property', ...
    'AngularVelocity',angVelBody);
```

Call the kinematic trajectory object in a loop and log the position output. Plot the position over time.

```
N = 10000;
pos = zeros(N, 3);
for i = 1:N
    pos(i,:) = traj();
end

plot3(pos(:,1), pos(:,2), pos(:,3))
title('Position')
xlabel('X (m)')
ylabel('Y (m)')
zlabel('Z (m)')
```



Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

The object functions, `perturbations` and `perturb`, do not support code generation.

Usage notes and limitations:

“System Objects in MATLAB Code Generation” (MATLAB Coder)

See Also

`waypointTrajectory`

Introduced in R2019b

lidarScan

Create object for storing 2-D lidar scan

Description

A `lidarScan` object contains data for a single 2-D lidar (light detection and ranging) scan. The lidar scan is a laser scan for a 2-D plane with distances (`Ranges`) measured from the sensor to obstacles in the environment at specific angles (`Angles`). Use this laser scan object as an input to other robotics algorithms such as `matchScans`, `controllerVFH`, or `monteCarloLocalization`.

Creation

Syntax

```
scan = lidarScan(ranges,angles)
scan = lidarScan(cart)
```

Description

`scan = lidarScan(ranges,angles)` creates a `lidarScan` object from the `ranges` and `angles`, that represent the data collected from a lidar sensor. The `ranges` and `angles` inputs are vectors of the same length and are set directly to the `Ranges` and `Angles` properties.

`scan = lidarScan(cart)` creates a `lidarScan` object using the input Cartesian coordinates as an *n*-by-2 matrix. The `Cartesian` property is set directly from this input.

`scan = lidarScan(scanMsg)` creates a `lidarScan` object from a `LaserScan` ROS message object.

Properties

Ranges — Range readings from lidar in meters

vector

Range readings from lidar, specified as a vector in meters. This vector is the same length as `Angles`, and the vector elements are measured in meters.

Data Types: `single` | `double`

Angles — Angle of readings from lidar in radians

vector

Angle of range readings from lidar, specified as a vector. This vector is the same length as `Ranges`, and the vector elements are measured in radians. Angles are measured counter-clockwise around the positive *z*-axis.

Data Types: `single` | `double`

`Cartesian` — Cartesian coordinates of lidar readings in meters

`[x y]` matrix

Cartesian coordinates of lidar readings, returned as an `[x y]` matrix. In the lidar coordinate frame, positive *x* is forward and positive *y* is to the left.

Data Types: `single` | `double`

`Count` — Number of lidar readings

scalar

Number of lidar readings, returned as a scalar. This scalar is also equal to the length of the `Ranges` and `Angles` vectors or the number of rows in `Cartesian`.

Data Types: `double`

Object Functions

plot	Display laser or lidar scan readings
removeInvalidData	Remove invalid range and angle data
transformScan	Transform laser scan based on relative pose

Examples

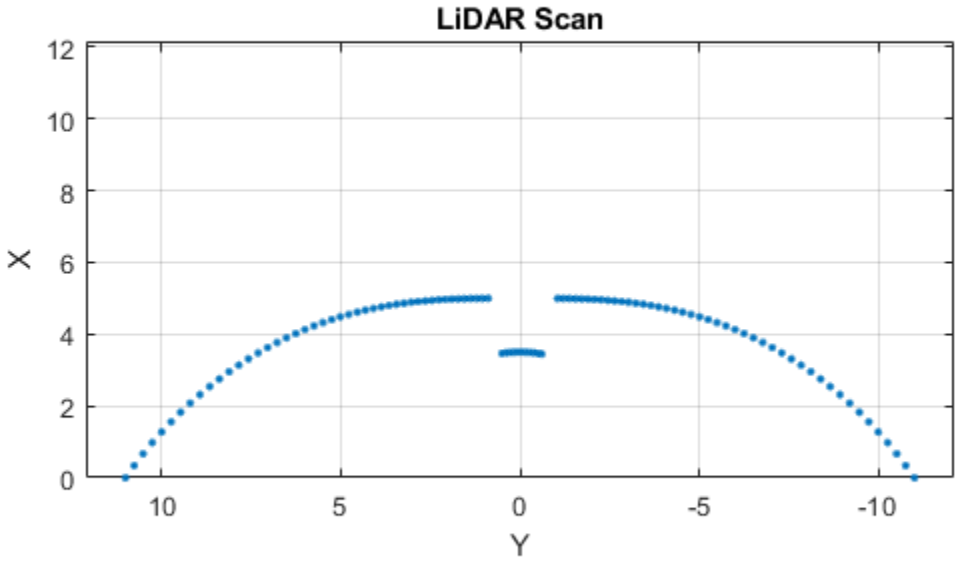
Plot Lidar Scan and Remove Invalid Points

Specify lidar data as vectors of ranges and angles. These values include readings outside of the sensors range.

```
x = linspace(-2,2);
ranges = abs((1.5).*x.^2 + 5);
ranges(45:55) = 3.5;
angles = linspace(-pi/2,pi/2,numel(ranges));
```

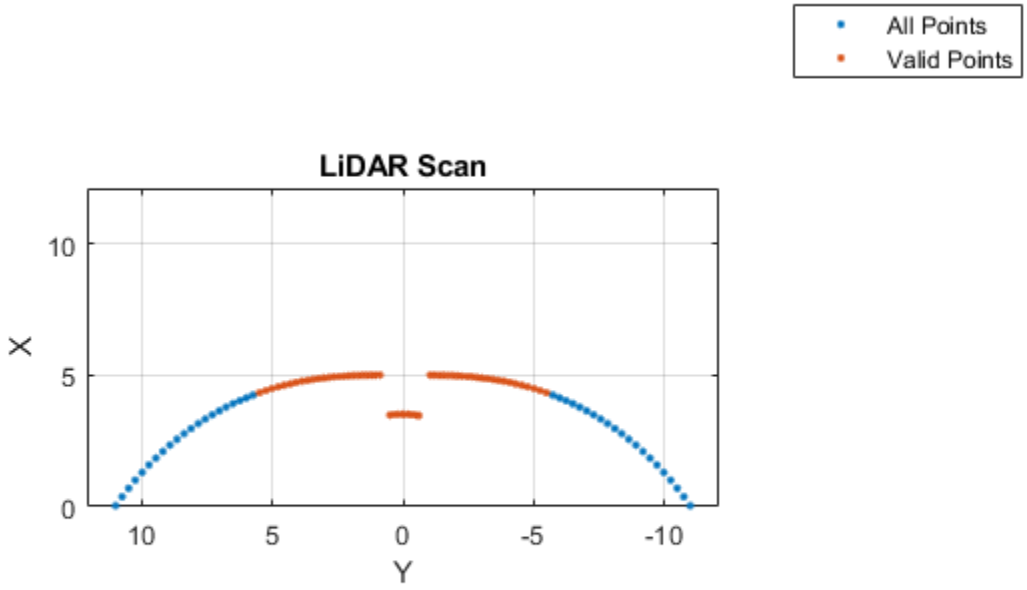
Create a lidar scan by specifying the ranges and angles. Plot all points of the lidar scan.

```
scan = lidarScan(ranges,angles);
plot(scan)
```



Remove invalid points based on a specified minimum and maximum range.

```
minRange = 0.1;
maxRange = 7;
scan2 = removeInvalidData(scan,'RangeLimits',[minRange maxRange]);
hold on
plot(scan2)
legend('All Points','Valid Points')
```



Transform Laser Scans

Create a `lidarScan` object. Specify the ranges and angles as vectors.

```
refRanges = 5*ones(1,300);
refAngles = linspace(-pi/2,pi/2,300);
refScan = lidarScan(refRanges,refAngles);
```

Translate the laser scan by an `[x y]` offset of `(0.5,0.2)`.

```
transformedScan = transformScan(refScan,[0.5 0.2 0]);
```

Rotate the laser scan by 20 degrees.

```
rotateScan = transformScan(refScan,[0,0,deg2rad(20)]);
```

Match Lidar Scans

Create a reference lidar scan using `lidarScan` (Robotics System Toolbox). Specify ranges and angles as vectors.

```
refRanges = 5*ones(1,300);
refAngles = linspace(-pi/2,pi/2,300);
refScan = lidarScan(refRanges,refAngles);
```

Using the `transformScan` (Robotics System Toolbox) function, generate a second lidar scan at an `x,y` offset of `(0.5,0.2)`.

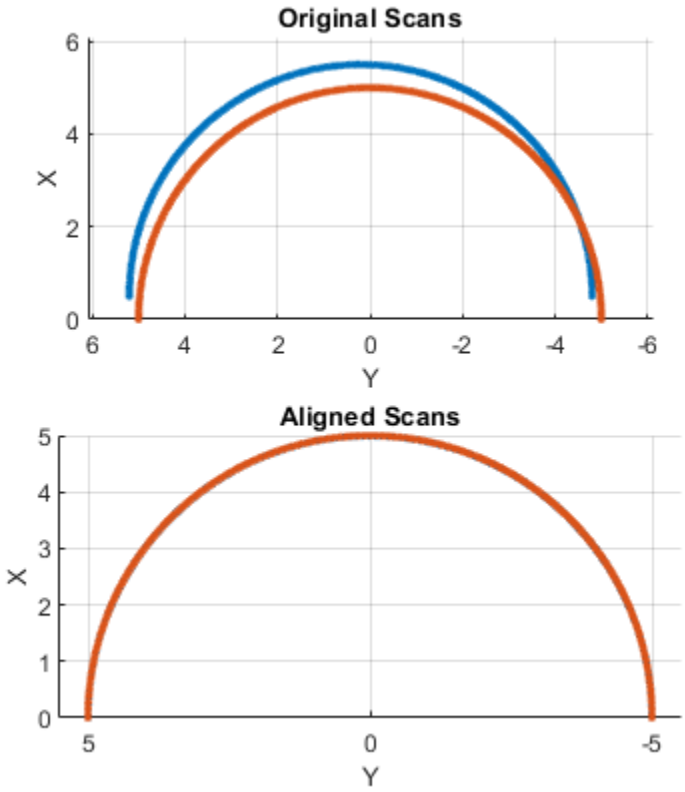
```
currScan = transformScan(refScan,[0.5 0.2 0]);
```

Match the reference scan and the second scan to estimate the pose difference between them.

```
pose = matchScans(currScan,refScan);
```

Use the `transformScan` function to align the scans by transforming the second scan into the frame of the first scan using the relative pose difference. Plot both the original scans and the aligned scans.

```
currScan2 = transformScan(currScan,pose);

subplot(2,1,1);
hold on
plot(currScan)
plot(refScan)
title('Original Scans')
hold off

subplot(2,1,2);
hold on
plot(currScan2)
plot(refScan)
title('Aligned Scans')
xlim([0 5])
hold off
```



Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

Lidar scans require a limited size in code generation. The lidar scans are limited to 4000 points (range and angles) as a maximum.

See Also

`matchScans` | `transformScan` | `controllerVFH` | `monteCarloLocalization`

Introduced in R2019b

plot

Display laser or lidar scan readings

Syntax

```
plot(scanObj)
plot( ___ ,Name,Value)
linehandle = plot( ___ )
```

Description

`plot(scanObj)` plots the lidar scan readings specified in `scanObj`.

`plot( ___ ,Name,Value)` provides additional options specified by one or more `Name,Value` pair arguments.

`linehandle = plot( ___ )` returns a column vector of line series handles, using any of the arguments from previous syntaxes. Use `linehandle` to modify properties of the line series after it is created.

Examples

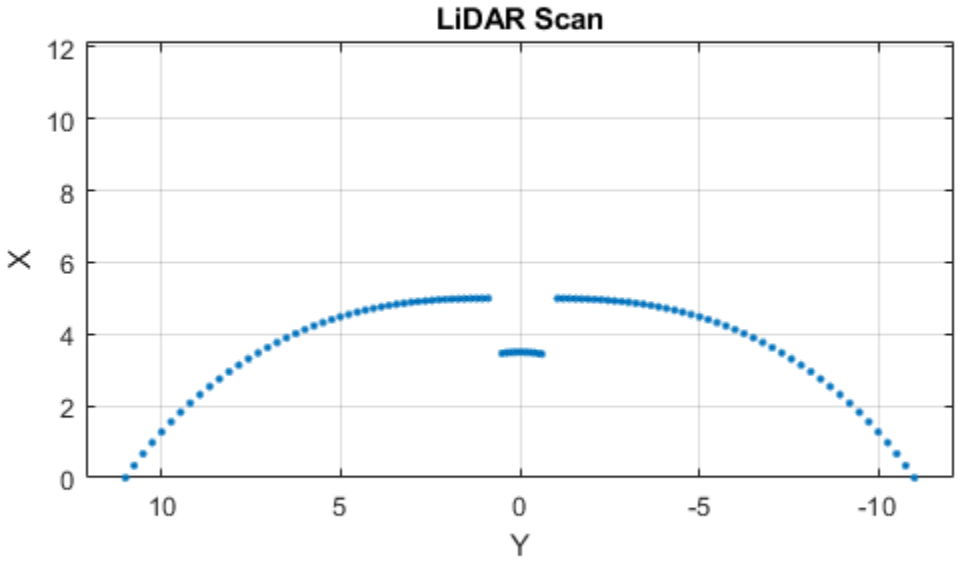
Plot Lidar Scan and Remove Invalid Points

Specify lidar data as vectors of ranges and angles. These values include readings outside of the sensors range.

```
x = linspace(-2,2);
ranges = abs((1.5).*x.^2 + 5);
ranges(45:55) = 3.5;
angles = linspace(-pi/2,pi/2,numel(ranges));
```

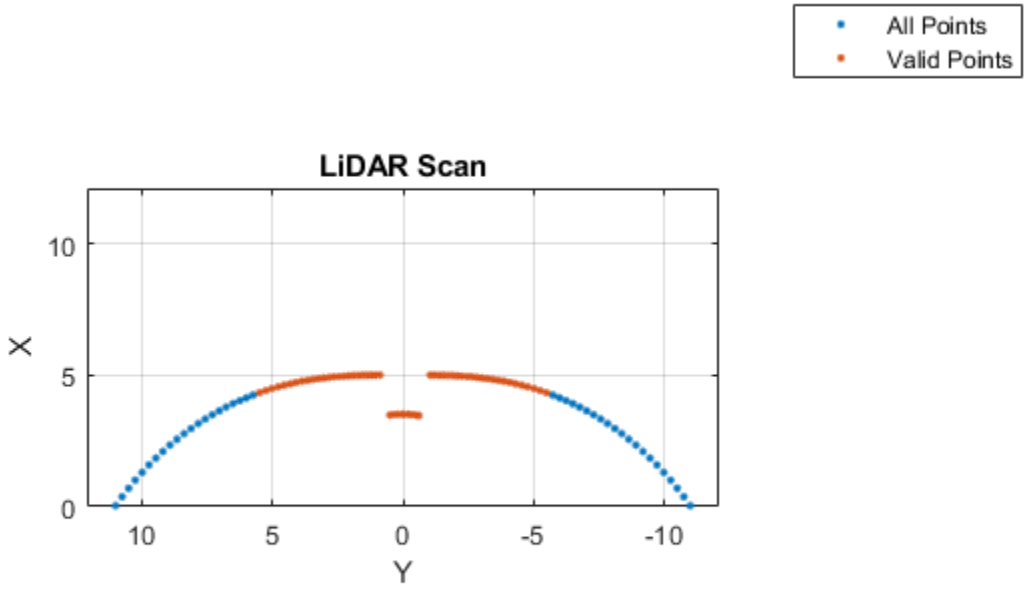
Create a lidar scan by specifying the ranges and angles. Plot all points of the lidar scan.

```
scan = lidarScan(ranges,angles);
plot(scan)
```



Remove invalid points based on a specified minimum and maximum range.

```
minRange = 0.1;
maxRange = 7;
scan2 = removeInvalidData(scan,'RangeLimits',[minRange maxRange]);
hold on
plot(scan2)
legend('All Points','Valid Points')
```



Input Arguments

`scanObj` — Lidar scan readings

`lidarScan` object

Lidar scan readings, specified as a `lidarScan` object.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `"MaximumRange",5`

`Parent` — Parent of axes

axes object

Parent of axes, specified as the comma-separated pair consisting of `"Parent"` and an axes object in which the laser scan is drawn. By default, the laser scan is plotted in the currently active axes.

`MaximumRange` — Range of laser scan

`scan.RangeMax` (default) | scalar

Range of laser scan, specified as the comma-separated pair consisting of `"MaximumRange"` and a scalar. When you specify this name-value pair argument, the minimum and maximum *x*-axis and the

maximum *y*-axis limits are set based on specified value. The minimum *y*-axis limit is automatically determined by the opening angle of the laser scanner.

This name-value pair only works when you input `scanMsg` as the laser scan.

Outputs

`linehandle` — One or more chart line objects

scalar | vector

One or more chart line objects, returned as a scalar or a vector. These are unique identifiers, which you can use to query and modify properties of a specific chart line.

See Also

`matchScans` | `transformScan` | `controllerVFH` | `monteCarloLocalization`

Topics

“Estimate Robot Pose with Scan Matching”

Introduced in R2015a

lidarSLAM

Perform localization and mapping using lidar scans

Description

The `lidarSLAM` class performs simultaneous localization and mapping (SLAM) for lidar scan sensor inputs. The SLAM algorithm takes in lidar scans and attaches them to a node in an underlying pose graph. The algorithm then correlates the scans using scan matching. It also searches for loop closures, where scans overlap previously mapped regions, and optimizes the node poses in the pose graph.

Creation

Syntax

```
slamObj = lidarSLAM
slamObj = lidarSLAM(mapResolution,maxLidarRange)
slamObj = lidarSLAM(mapResolution,maxLidarRange,maxNumScans)
```

Description

`slamObj = lidarSLAM` creates a lidar SLAM object. The default occupancy map size is 20 cells per meter. The maximum range for each lidar scan is 8 meters.

`slamObj = lidarSLAM(mapResolution,maxLidarRange)` creates a lidar SLAM object and sets the `MapResolution` and `MaxLidarRange` properties based on the inputs.

`slamObj = lidarSLAM(mapResolution,maxLidarRange,maxNumScans)` specifies the upper bound on the number of accepted scans allowed when generating code. `maxNumScans` is a positive integer. This scan limit is only required when generating code.

Properties

`PoseGraph` — Underlying pose graph that connects scans
`poseGraph` object

Underlying pose graph that connects scans, specified as a `poseGraph` object. Adding scans to `lidarSLAM` updates this pose graph. When loop closures are found, the pose graph is optimized using `OptimizationFcn`.

`MapResolution` — Resolution of occupancy grid map
`20` cells per meter (default) | positive integer

Resolution of the occupancy grid map, specified as a positive integer in cells per meter. Specify the map resolution on construction.

`MaxLidarRange` — Maximum range of lidar sensor
8 meters (default) | positive scalar

Maximum range of the lidar sensor, specified as a positive scalar in meters. Specify the maximum range on construction.

`OptimizationFcn` — Pose graph optimization function
`optimizePoseGraph` (default) | function handle

Pose graph optimization function, specified as a function handle. By default, the algorithm calls the `optimizePoseGraph` function. To specify your own optimization method, the class requires the function signature to be:

```
[updatedPose,stat] = myOptimizationFcn(poseGraph)
```

`poseGraph` is a `poseGraph` object. `updatedPose` is an *n*-by-3 vector of `[x y theta]` poses listed in sequential node ID order. `stat` is a structure containing a `ResidualError` field as a positive scalar. Use the `stat` structure to include other information relevant to your optimization.

`LoopClosureThreshold` — Threshold for accepting loop closures
`100` (default) | positive scalar

Threshold on the score from the scan matching algorithm for accepting loop closures, specified as a positive scalar. Higher thresholds correspond to a better match, but scores vary based on sensor data.

`LoopClosureSearchRadius` — Search radius for loop closure detection
8 meters (default) | positive scalar

Search radius for loop closure detection, specified as a positive scalar. Increasing this radius affects performance by increasing search time. Tune this distance based on your environment and the expected vehicle trajectory.

`LoopClosureMaxAttempts` — Number of attempts at finding loop closures
`1` (default) | positive integer

Number of attempts at finding looping closures, specified as a positive integer. Increasing the number of attempts affects performance by increasing search time.

`LoopClosureAutoRollback` — Allow automatic rollback of added loop closures
`true` (default) | `false`

Allow automatic rollback of added loop closures, specified as `true` or `false`. The SLAM object tracks the residual error returned by the `OptimizationFcn`. If it detects a sudden change in the residual error and this property is `true`, it rejects (rolls back) the loop closure.

`OptimizationInterval` — Number of loop closures accepted to trigger optimization
`1` (default) | positive integer

Number of loop closures accepted to trigger optimization, specified as a positive integer. By default, the `PoseGraph` is optimized every time `lidarSLAM` adds a loop closure.

`MovementThreshold` — Minimum change in pose required to process scans
`[0 0]` (default) | `[translation rotation]`

Minimum change in pose required to process scans, specified as a `[translation rotation]` vector. A relative pose change for a newly added scan is calculated as `[x y theta]`. If the translation in *xy*-position or rotation of `theta` exceeds these thresholds, the `lidarSLAM` object accepts the scan and adds a pose is added to the `PoseGraph`.

ScanRegistrationMethod — Scan registration method
`'BranchAndBound'` (default) | `'PhaseCorrelation'`

Scan registration method, specified as a character vector.

Note Image Processing Toolbox™ is required for using Phase Correlation method.

Object Functions

addScan	Add scan to lidar SLAM map
copy	Copy lidar SLAM object
removeLoopClosures	Remove loop closures from pose graph
scansAndPoses	Extract scans and corresponding poses
show	Plot scans and robot poses

Examples

Perform SLAM Using Lidar Scans

Use a `lidarSLAM` object to iteratively add and compare lidar scans and build an optimized pose graph of the robot trajectory. To get an occupancy map from the associated poses and scans, use the `buildMap` function.

Load Data and Set Up SLAM Algorithm

Load a cell array of `lidarScan` objects. The lidar scans were collected in a parking garage on a Husky® robot from ClearPath Robotics®. Typically, lidar scans are taken at a high frequency and each scan is not needed for SLAM. Therefore, down sample the scans by selecting only every 40th scan.

```
load garage_fl1_southend.mat scans
scans = scans(1:40:end);
```

To set up the SLAM algorithm, specify the lidar range, map resolution, loop closure threshold, and search radius. Tune these parameters for your specific robot and environment. Create the `lidarSLAM` object with these parameters.

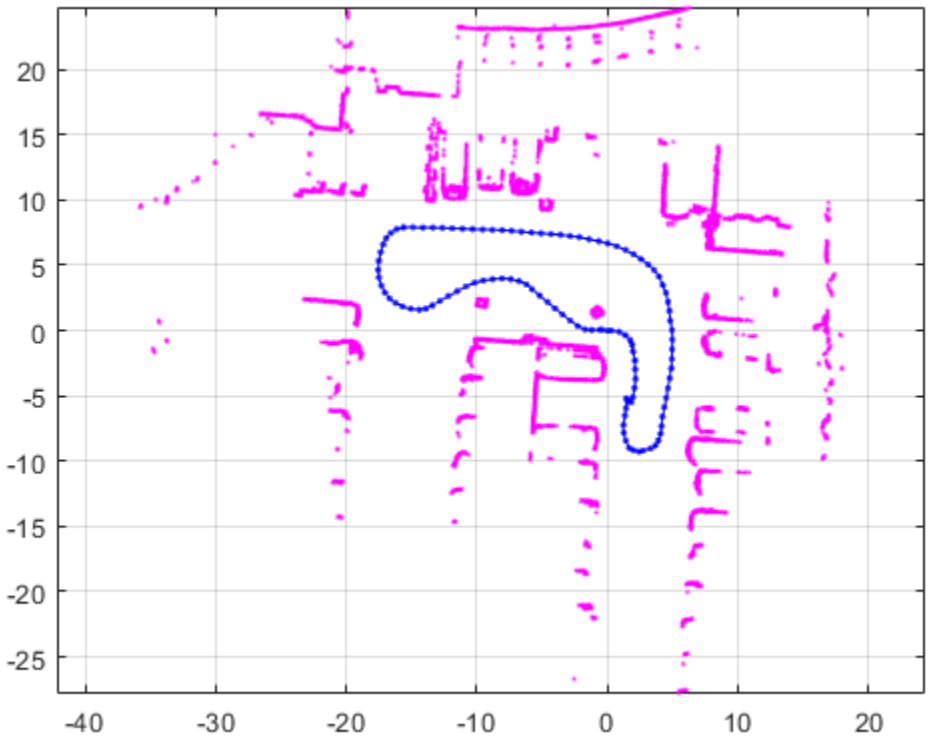
```
maxRange = 19.2; % meters
resolution = 10; % cells per meter

slamObj = lidarSLAM(resolution,maxRange);
slamObj.LoopClosureThreshold = 360;
slamObj.LoopClosureSearchRadius = 8;
```

Add Scans Iteratively

Using a `for` loop, add scans to the SLAM object. The object uses scan matching to compare each added scan to previously added ones. To improve the map, the object optimizes the pose graph whenever it detects a loop closure. Every 10 scans, display the stored poses and scans.

```
for i = 1:numel(scans)

    addScan(slamObj,scans{i});
```

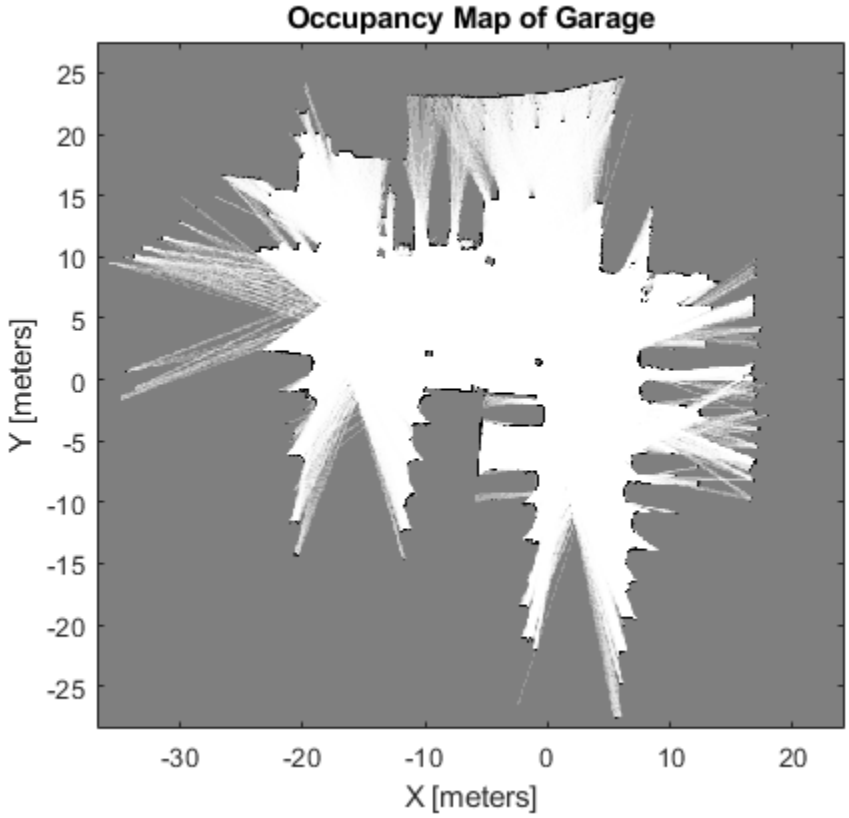
```
    if rem(i,10) == 0
        show(slamObj);
    end
end
```



View Occupancy Map

After adding all the scans to the SLAM object, build an `occupancyMap` map by calling `buildMap` with the scans and poses. Use the same map resolution and max range you used with the SLAM object.

```
[scansSLAM,poses] = scansAndPoses(slamObj);
occMap = buildMap(scansSLAM,poses,resolution,maxRange);
figure
show(occMap)
title('Occupancy Map of Garage')
```



More About

SLAM

Simultaneous localization and mapping (SLAM) is a general concept for algorithms correlating different sensor readings to build a map of a vehicle environment and track pose estimates. Different algorithms use different types of sensors and methods for correlating data.

The `lidarSLAM` algorithm uses lidar scans and odometry information as sensor inputs. The lidar scans map the environment and are correlated between each other to build an underlying pose graph of the vehicle trajectory. Odometry information is an optional input that gives an initial pose estimate for the scans to aid in the correlation. Scan matching algorithms correlate scans to previously added scans to estimate the relative pose between them and add them to an underlying pose graph.

The pose graph contains nodes connected by edges that represent the relative poses of the vehicle. Edges specify constraints on the node as an information matrix. To correct for drifting pose estimates, the algorithm optimizes over the whole pose graph whenever it detects loop closures.

The algorithm assumes that data comes from a vehicle navigating an environment and incrementally getting laser scans along its path. Therefore, scans are first compared to the most recent scan to identify relative poses and are added to the pose graph incrementally. However, the algorithm also searches for loop closures, which identify when the vehicle scans an area that was previously visited.

When working with SLAM algorithms, the environment and vehicle sensors affect the performance and data correlation quality. Tune your parameters properly for your expected environment or dataset.

References

[1] Hess, Wolfgang, Damon Kohler, Holger Rapp, and Daniel Andor. "Real-Time Loop Closure in 2D LIDAR SLAM." *2016 IEEE International Conference on Robotics and Automation (ICRA)*. 2016.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Use this syntax when constructing `lidarSLAM` objects for code generation:

`slamObj= lidarSLAM(mapResolution,maxLidarRange,maxNumScans)` specifies the upper bound on the number of accepted scans allowed when generating code. `maxNumScans` is a positive integer. This scan limit is only required when generating code.

See Also

`poseGraph` | `optimizePoseGraph`

Topics

“Implement Simultaneous Localization And Mapping (SLAM) with Lidar Scans”
“Implement Online Simultaneous Localization And Mapping (SLAM) with Lidar Scans”

Introduced in R2019b

addScan

Add scan to lidar SLAM map

Syntax

```
addScan(slamObj,currScan)
addScan(slamObj,currScan,relPoseEst)
[isAccepted,loopClosureInfo,optimInfo] = addScan( ___ )
```

Description

`addScan(slamObj,currScan)` adds a lidar scan, `currScan`, to the lidar SLAM object, `slamObj`. The function uses scan matching to correlate this scan to the most recent one, then adds it to the pose graph defined in `slamObj`. If the scan is accepted, `addScan` detects loop closures and optimizes based on settings in `slamObj`.

`addScan(slamObj,currScan,relPoseEst)` also specifies a relative pose to the latest lidar scan pose in `slamObj`. This relative pose improves the scan matching.

Note The `relPoseEst` input is ignored when the `ScanRegistrationMethod` property of `lidarSLAM` object is set to `'PhaseCorrelation'`.

`[isAccepted,loopClosureInfo,optimInfo] = addScan( ___ )` outputs detailed information about adding the scan to the SLAM object. `isAccepted` indicates if the scan is added or rejected. `loopClosureInfo` and `optimInfo` indicate if a loop closure is detected or the pose graph is optimized.

Examples

Perform SLAM Using Lidar Scans

Use a `lidarSLAM` object to iteratively add and compare lidar scans and build an optimized pose graph of the robot trajectory. To get an occupancy map from the associated poses and scans, use the `buildMap` function.

Load Data and Set Up SLAM Algorithm

Load a cell array of `lidarScan` objects. The lidar scans were collected in a parking garage on a Husky® robot from ClearPath Robotics®. Typically, lidar scans are taken at a high frequency and each scan is not needed for SLAM. Therefore, down sample the scans by selecting only every 40th scan.

```
load garage_fl1_southend.mat scans
scans = scans(1:40:end);
```

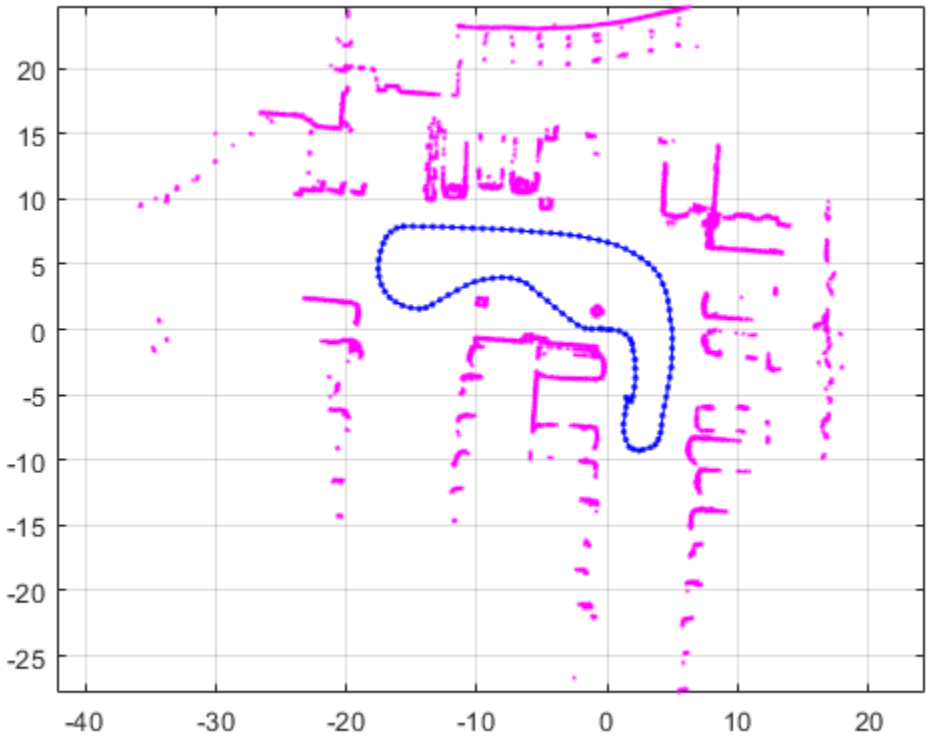
To set up the SLAM algorithm, specify the lidar range, map resolution, loop closure threshold, and search radius. Tune these parameters for your specific robot and environment. Create the `lidarSLAM` object with these parameters.

```
maxRange = 19.2; % meters
resolution = 10; % cells per meter

slamObj = lidarSLAM(resolution,maxRange);
slamObj.LoopClosureThreshold = 360;
slamObj.LoopClosureSearchRadius = 8;
```

Add Scans Iteratively

Using a `for` loop, add scans to the SLAM object. The object uses scan matching to compare each added scan to previously added ones. To improve the map, the object optimizes the pose graph whenever it detects a loop closure. Every 10 scans, display the stored poses and scans.

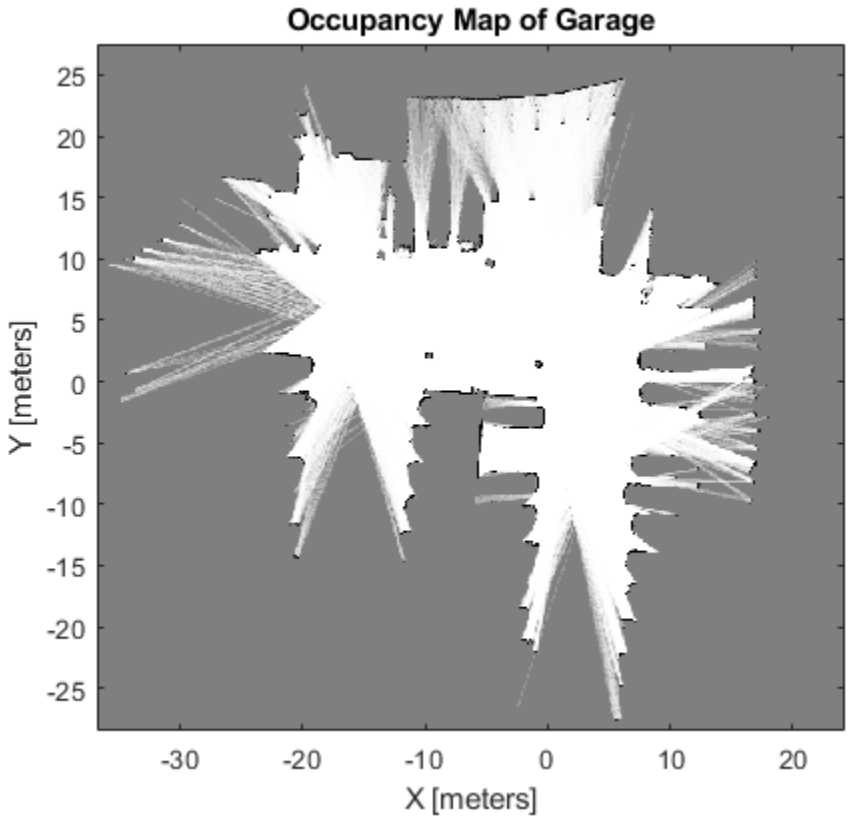
```
for i = 1:numel(scans)

    addScan(slamObj,scans{i});

    if rem(i,10) == 0
        show(slamObj);
    end
end
```



View Occupancy Map

After adding all the scans to the SLAM object, build an `occupancyMap` map by calling `buildMap` with the scans and poses. Use the same map resolution and max range you used with the SLAM object.

```
[scansSLAM,poses] = scansAndPoses(slamObj);
occMap = buildMap(scansSLAM,poses,resolution,maxRange);
figure
show(occMap)
title('Occupancy Map of Garage')
```



Input Arguments

`slamObj` — Lidar SLAM object

`lidarSLAM` object

Lidar SLAM object, specified as a `lidarSLAM` object. The object contains the SLAM algorithm parameters, sensor data, and underlying pose graph used to build the map.

`currScan` — Lidar scan reading

`lidarScan` object

Lidar scan reading, specified as a `lidarScan` object. This scan is correlated to the most recent scan in `slamObj` using scan matching.

`relPoseEst` — Relative pose estimate of scan

`[x y theta]` vector

Relative pose estimate of scan, specified as an `[x y theta]` vector. This relative pose improves scan matching.

Output Arguments

`isAccepted` — Indicates if scan is accepted

`true` | `false`

Indicates if scan is accepted, returned as `true` or `false`. If the relative pose between scans is below the `MovementThreshold` property of `slamObj`, the scan is rejected. By default, all scans are accepted.

`loopClosureInfo` — Loop closure details

structure

Loop closure details, returned as a structure with these fields:

- `EdgeIDs` –– IDs of newly connected edges in the pose graph, returned as a vector.
- `Edges` –– Newly added loop closure edges, returned as an *n*-by-2 matrix of node IDs that each edge connects.
- `Scores` –– Scores of newly connected edges in the pose graph returned from scan matching, returned as a vector.

Note If the `LoopClosureAutoRollback` property is set to `true` in `slamObj`, loop closure edges can be removed from the pose graph. This property rejects loops closures if the residual error changes drastically after optimization. Therefore, some of the edge IDs listed in this structure may not exist in the actual pose graph.

`optimInfo` — Pose graph optimization details

structure

Pose graph optimization details, returned as a structure with these fields:

- `IsPerformed` –– Boolean indicating if optimization is performed when adding this scan. Optimization performance depends on the `OptimizationInterval` property in `slamObj`.
- `IsAccepted` –– Boolean indicating if optimization was accepted based on `ResidualError`.
- `ResidualError` –– Error associated with optimization, returned as a scalar.
- `LoopClosureRemoved` –– List of IDs of loop closure edges removed during optimization, returned as a vector.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Use this syntax when constructing `lidarSLAM` objects for code generation:

`slamObj= lidarSLAM(mapResolution,maxLidarRange,maxNumScans)` specifies the upper bound on the number of accepted scans allowed when generating code. `maxNumScans` is a positive integer. This scan limit is only required when generating code.

See Also

`poseGraph` | `optimizePoseGraph`

Topics
“Implement Simultaneous Localization And Mapping (SLAM) with Lidar Scans”
“Implement Online Simultaneous Localization And Mapping (SLAM) with Lidar Scans”

Introduced in R2019b

copy

Copy lidar SLAM object

Syntax

```
newSlamObj = copy(slamObj)
```

Description

`newSlamObj = copy(slamObj)` creates a deep copy of `slamObj` with the same properties. Any changes made to `newSlamObj` are not reflected in `slamObj`.

Input Arguments

`slamObj` — Lidar SLAM object

`lidarSLAM` object

Lidar SLAM object, specified as a `lidarSLAM` object. The object contains the SLAM algorithm parameters, sensor data, and underlying pose graph used to build the map.

Output Arguments

`newSlamObj` — Lidar SLAM object

`lidarSLAM` object

Lidar SLAM object, returned as a `lidarSLAM` object.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Use this syntax when constructing `lidarSLAM` objects for code generation:

`slamObj= lidarSLAM(mapResolution,maxLidarRange,maxNumScans)` specifies the upper bound on the number of accepted scans allowed when generating code. `maxNumScans` is a positive integer. This scan limit is only required when generating code.

See Also

`poseGraph` | `optimizePoseGraph`

Topics

“Implement Simultaneous Localization And Mapping (SLAM) with Lidar Scans”
“Implement Online Simultaneous Localization And Mapping (SLAM) with Lidar Scans”

Introduced in R2019b

removeLoopClosures

Remove loop closures from pose graph

Syntax

```
removeLoopClosures(slamObj)
removeLoopClosures(slamObj,lcEdgeIDs)
```

Description

`removeLoopClosures(slamObj)` removes all loop closures from the underlying pose graph in `slamObj`.

`removeLoopClosures(slamObj,lcEdgeIDs)` removes the loop closure edges with the specified IDs from the underlying pose graph in `slamObj`.

Input Arguments

`slamObj` — Lidar SLAM object

`lidarSLAM` object

Lidar SLAM object, specified as a `lidarSLAM` object. The object contains the SLAM algorithm parameters, sensor data, and underlying pose graph used to build the map

`lcEdgeIDs` — Loop closure edge IDs

vector of positive integers

Loop closure edge IDs, specified as a vector of positive integers. To find specific edge IDs, use `findEdgeID` on the underlying `poseGraph` object defined in `slamObj`.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Use this syntax when constructing `lidarSLAM` objects for code generation:

`slamObj= lidarSLAM(mapResolution,maxLidarRange,maxNumScans)` specifies the upper bound on the number of accepted scans allowed when generating code. `maxNumScans` is a positive integer. This scan limit is only required when generating code.

See Also

`poseGraph` | `optimizePoseGraph`

Topics

“Implement Simultaneous Localization And Mapping (SLAM) with Lidar Scans”
“Implement Online Simultaneous Localization And Mapping (SLAM) with Lidar Scans”

Introduced in R2019b

scansAndPoses

Extract scans and corresponding poses

Syntax

```
[scans,poses] = scansAndPoses(slamObj)
[scans,poses] = scansAndPoses(slamObj,nodeIDs)
```

Description

`[scans,poses] = scansAndPoses(slamObj)` returns the scans used by the `lidarSLAM` object as `lidarScan` objects, along with their associated `[x y theta]` poses from the underlying pose graph of `slamObj`.

`[scans,poses] = scansAndPoses(slamObj,nodeIDs)` returns the scans and poses for the specific node IDs. To get the node IDs, see the underlying `poseGraph` object in `slamObj` for the node IDs.

Examples

Perform SLAM Using Lidar Scans

Use a `lidarSLAM` object to iteratively add and compare lidar scans and build an optimized pose graph of the robot trajectory. To get an occupancy map from the associated poses and scans, use the `buildMap` function.

Load Data and Set Up SLAM Algorithm

Load a cell array of `lidarScan` objects. The lidar scans were collected in a parking garage on a Husky® robot from ClearPath Robotics®. Typically, lidar scans are taken at a high frequency and each scan is not needed for SLAM. Therefore, down sample the scans by selecting only every 40th scan.

```
load garage_fl1_southend.mat scans
scans = scans(1:40:end);
```

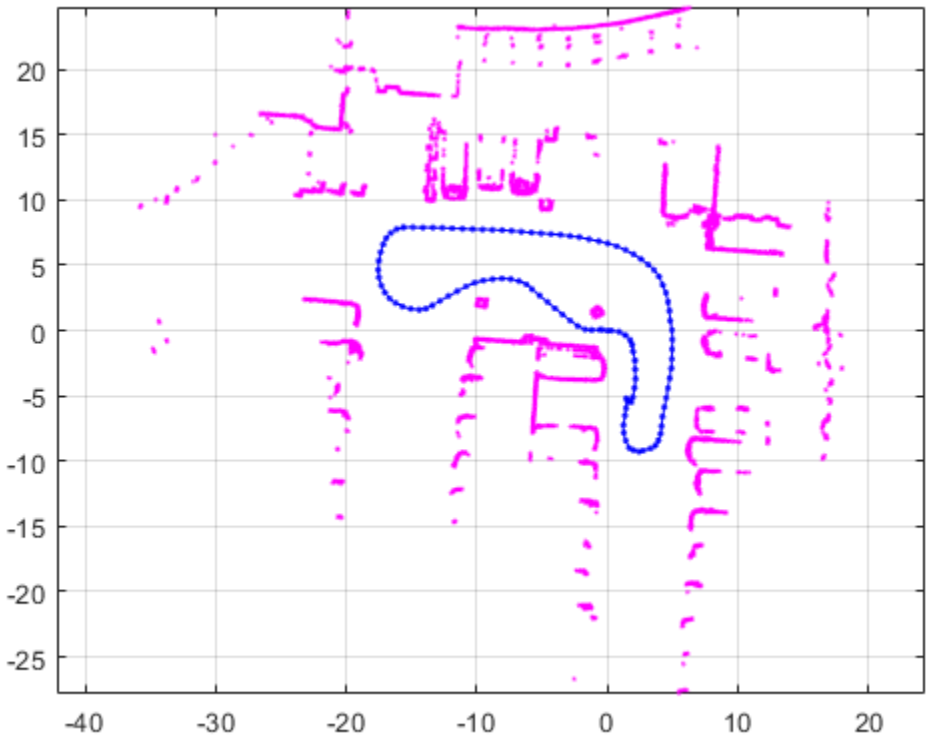
To set up the SLAM algorithm, specify the lidar range, map resolution, loop closure threshold, and search radius. Tune these parameters for your specific robot and environment. Create the `lidarSLAM` object with these parameters.

```
maxRange = 19.2; % meters
resolution = 10; % cells per meter

slamObj = lidarSLAM(resolution,maxRange);
slamObj.LoopClosureThreshold = 360;
slamObj.LoopClosureSearchRadius = 8;
```

Add Scans Iteratively

Using a `for` loop, add scans to the SLAM object. The object uses scan matching to compare each added scan to previously added ones. To improve the map, the object optimizes the pose graph whenever it detects a loop closure. Every 10 scans, display the stored poses and scans.

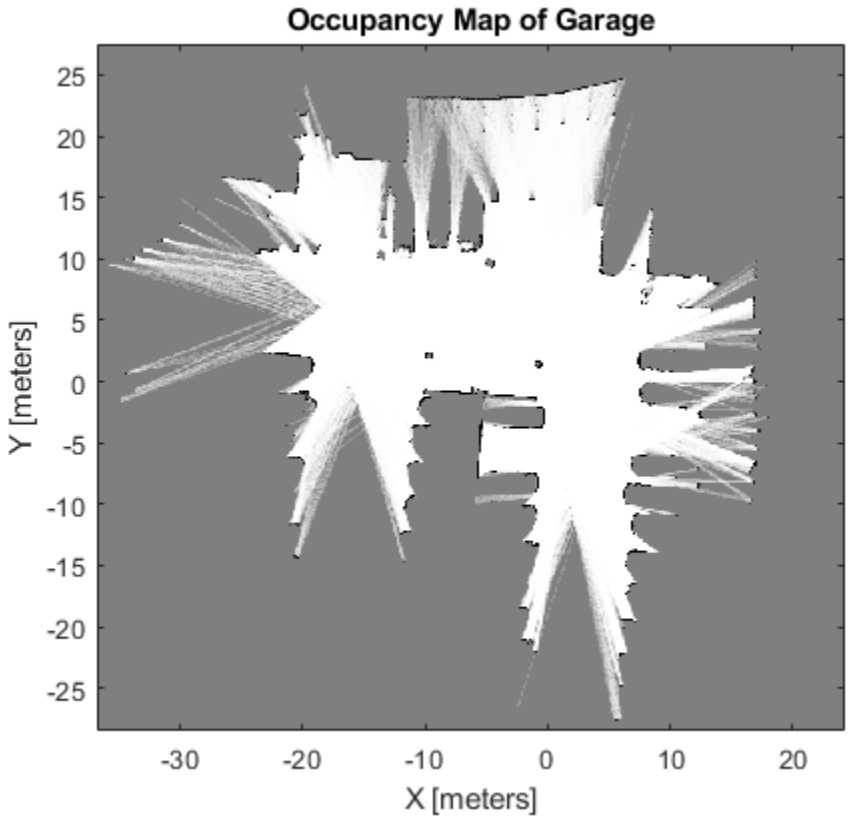
```
for i = 1:numel(scans)

    addScan(slamObj,scans{i});

    if rem(i,10) == 0
        show(slamObj);
    end
end
```



View Occupancy Map

After adding all the scans to the SLAM object, build an `occupancyMap` map by calling `buildMap` with the scans and poses. Use the same map resolution and max range you used with the SLAM object.

```
[scansSLAM,poses] = scansAndPoses(slamObj);
occMap = buildMap(scansSLAM,poses,resolution,maxRange);
figure
show(occMap)
title('Occupancy Map of Garage')
```



Input Arguments

`slamObj` — Lidar SLAM object
`lidarSLAM` object

Lidar SLAM object, specified as a `lidarSLAM` object. The object contains the SLAM algorithm parameters, sensor data, and underlying pose graph used to build the map.

`nodeIDs` — Node IDs from pose graph
positive integer

Node IDs from pose graph, specified as a positive integer. Nodes are added to the pose graph with sequential ID numbers. To get the node IDs, see the underlying `poseGraph` object in `slamObj` for the node IDs.

Output Arguments

`scans` — Lidar scan readings
`lidarScan` object

Lidar scan readings, returned as a `lidarScan` object.

`poses` — Pose for each scan
n-by-3 matrix | `[x y theta]` vectors

Pose for each scan, returned as an *n*-by-3 matrix of `[x y theta]` vectors. Each row is a pose that corresponds to a scan in `scans`.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Use this syntax when constructing `lidarSLAM` objects for code generation:

`slamObj= lidarSLAM(mapResolution,maxLidarRange,maxNumScans)` specifies the upper bound on the number of accepted scans allowed when generating code. `maxNumScans` is a positive integer. This scan limit is only required when generating code.

See Also

`poseGraph` | `optimizePoseGraph`

Topics

“Implement Simultaneous Localization And Mapping (SLAM) with Lidar Scans”
“Implement Online Simultaneous Localization And Mapping (SLAM) with Lidar Scans”

Introduced in R2019b

show

Plot scans and robot poses

Syntax

```
show(slamObj)
show(slamObj,Name,Value)
axes = show( ___ )
```

Description

`show(slamObj)` plots all the scans added to the `input lidarSLAM` object overlaid with the lidar poses in its underlying pose graph.

`show(slamObj,Name,Value)` specifies options using `Name,Value` pair arguments. For example, `"Poses","off"` turns off display of the underlying pose graph in `slamObj`.

`axes = show( ___ )` returns the axes handle that the lidar SLAM data is plotted to using any of the previous syntaxes.

Examples

Perform SLAM Using Lidar Scans

Use a `lidarSLAM` object to iteratively add and compare lidar scans and build an optimized pose graph of the robot trajectory. To get an occupancy map from the associated poses and scans, use the `buildMap` function.

Load Data and Set Up SLAM Algorithm

Load a cell array of `lidarScan` objects. The lidar scans were collected in a parking garage on a Husky® robot from ClearPath Robotics®. Typically, lidar scans are taken at a high frequency and each scan is not needed for SLAM. Therefore, down sample the scans by selecting only every 40th scan.

```
load garage_fl1_southend.mat scans
scans = scans(1:40:end);
```

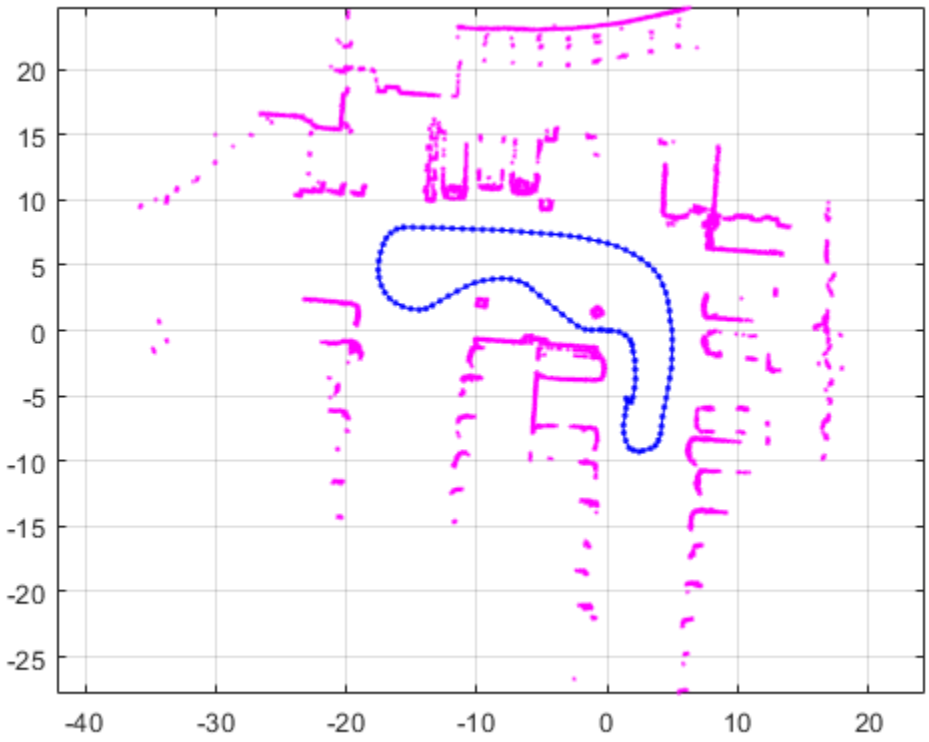
To set up the SLAM algorithm, specify the lidar range, map resolution, loop closure threshold, and search radius. Tune these parameters for your specific robot and environment. Create the `lidarSLAM` object with these parameters.

```
maxRange = 19.2; % meters
resolution = 10; % cells per meter

slamObj = lidarSLAM(resolution,maxRange);
slamObj.LoopClosureThreshold = 360;
slamObj.LoopClosureSearchRadius = 8;
```

Add Scans Iteratively

Using a `for` loop, add scans to the SLAM object. The object uses scan matching to compare each added scan to previously added ones. To improve the map, the object optimizes the pose graph whenever it detects a loop closure. Every 10 scans, display the stored poses and scans.

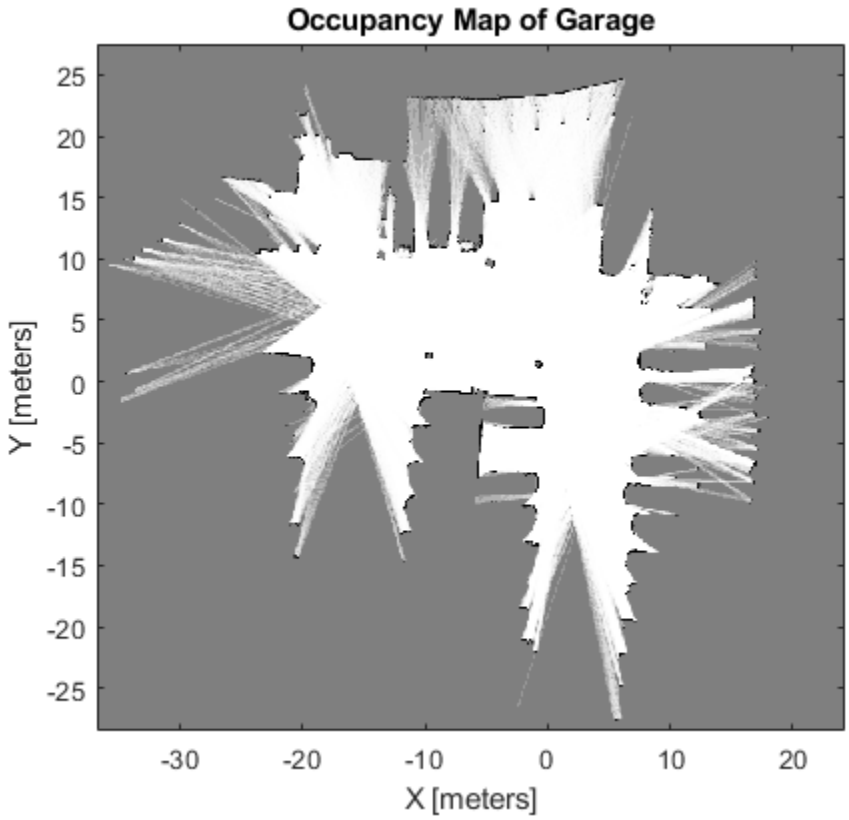
```
for i = 1:numel(scans)

    addScan(slamObj,scans{i});

    if rem(i,10) == 0
        show(slamObj);
    end
end
```



View Occupancy Map

After adding all the scans to the SLAM object, build an `occupancyMap` map by calling `buildMap` with the scans and poses. Use the same map resolution and max range you used with the SLAM object.

```
[scansSLAM,poses] = scansAndPoses(slamObj);
occMap = buildMap(scansSLAM,poses,resolution,maxRange);
figure
show(occMap)
title('Occupancy Map of Garage')
```



Input Arguments

`slamObj` — Lidar SLAM object

`lidarSLAM` object

Lidar SLAM object, specified as a `lidarSLAM` object. The object contains the SLAM algorithm parameters, sensor data, and underlying pose graph used to build the map.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `"Poses","off"`

`Parent` — Axes used to plot pose graph

`Axes` object | `UIAxes` object

Axes used to plot the pose graph, specified as the comma-separated pair consisting of `"Parent"` and either an `Axes` or `UIAxes` object. See `axes` or `uiaxes`.

Poses — Display lidar poses

`"on"` (default) | `"off"`

Display lidar poses, specified as the comma-separated pair consisting of `"Poses"` and `"on"` or `"off"`.

Output Arguments

axes — Axes used to plot the map

`Axes` object | `UIAxes` object

Axes used to plot the map, returned as either an `Axes` or `UIAxes` object. See `axes` or `uiaxes`.

See Also

`poseGraph` | `optimizePoseGraph`

Topics

“Implement Simultaneous Localization And Mapping (SLAM) with Lidar Scans”
“Implement Online Simultaneous Localization And Mapping (SLAM) with Lidar Scans”

Introduced in R2019b

likelihoodFieldSensorModel

Create a likelihood field range sensor model

Description

The `likelihoodFieldSensor` object creates a likelihood field sensor model object for range sensors. This object contains specific sensor model parameters. You can use this object to specify the model parameters in a `monteCarloLocalization` object.

Creation

Syntax

```
lf = likelihoodFieldSensorModel
```

Description

`lf = likelihoodFieldSensorModel` creates a likelihood field sensor model object for range sensors.

Properties

`Map` — Occupancy grid representing the map
`binaryOccupancyMap` object (default)

Occupancy grid representing the map, specified as a `binaryOccupancyMap` object. This object represents the environment of the vehicle as a grid with binary values indicating obstacles as `true` (`1`) and free locations as `false` (`0`).

`SensorPose` — Pose of the range sensor relative to the vehicle
[0 0 0] (default) | three-element vector

Pose of the range sensor relative to the coordinate frame of the vehicle, specified as a three-element vector, `[x y theta]`.

`SensorLimits` — Minimum and maximum range of sensor
[0 12] (default) | two-element vector

Minimum and maximum range of sensor, specified as a two-element vector in meters.

`NumBeams` — Number of beams used for likelihood computation
60 (default) | scalar

Number of beams used for likelihood computation, specified as a scalar. The computation efficiency can be improved by specifying a smaller number of beams than the actual number available from the sensor.

`MeasurementNoise` — Standard deviation for measurement noise
0.2 (default) | scalar

Standard deviation for measurement noise, specified as a scalar.

`RandomMeasurementWeight` — Weight for probability of random measurement
0.05 (default) | scalar

Weight for probability of random measurement, specified as a scalar. This scalar is the probability that the measurement is not accurate due to random interference.

`ExpectedMeasurementWeight` — Weight for probability of expected measurement
0.95 (default) | scalar

Weight for probability of expected measurement, specified as a scalar. The weight is the probability of getting a correct range measurement within the noise limits specified in `MeasurementNoise` property.

`MaxLikelihoodDistance` — Maximum distance to find nearest obstacles
2.0 (default) | scalar

Maximum distance to find nearest obstacles, specified as a scalar in meters.

Limitations

If you change your sensor model after using it with the `monteCarloLocalization` object, call `release` on that object beforehand. For example:

```
mcl = monteCarloLocalization;
[isUpdated,pose,covariance] = mcl(ranges,angles);
release(mcl)
mcl.SensorModel.NumBeams = 120;
```

See Also

`monteCarloLocalization` | `odometryMotionModel`

Topics
“Localize TurtleBot Using Monte Carlo Localization”
“Monte Carlo Localization Algorithm”

Introduced in R2019b

magparams

Magnetometer sensor parameters

Description

The `magparams` class creates a magnetometer sensor parameters object. You can use this object to model a magnetometer when simulating an IMU with `imuSensor`. See the “Algorithms” on page 2-438 section of `imuSensor` for details of `magparams` modeling.

Creation

Syntax

```
params = magarams
params = magparams(Name,Value)
```

Description

`params = magarams` returns an ideal magnetometer sensor parameters object with default values.

`params = magparams(Name,Value)` configures `magparams` object properties using one or more `Name,Value` pair arguments. `Name` is a property name and `Value` is the corresponding value. `Name` must appear inside single quotes (`''`). You can specify several name-value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`. Any unspecified properties take default values.

Properties

`MeasurementRange` — Maximum sensor reading (µT)
`Inf` (default) | real positive scalar

Maximum sensor reading in µT, specified as a real positive scalar.

Data Types: `single` | `double`

`Resolution` — Resolution of sensor measurements (µT/LSB)
`0` (default) | real nonnegative scalar

Resolution of sensor measurements in µT/LSB, specified as a real nonnegative scalar. Here, LSB is the acronym for least significant bit.

Data Types: `single` | `double`

`ConstantBias` — Constant sensor offset bias (µT)
`[0 0 0]` (default) | real scalar | real 3-element row vector

Constant sensor offset bias in µT, specified as a real scalar or 3-element row vector. Any scalar input is converted into a real 3-element row vector where each element has the input scalar value.

Data Types: `single` | `double`

`AxesMisalignment` — Sensor axes skew (%)
`diag([100 100 100])` (default) | scalar in the range [0,100] | 3-element row vector in the range [0,100] | 3-by-3 matrix in the range [0,100]

Sensor axes skew in percentage, specified as a scalar, a 3-element row vector, or a 3-by-3 matrix with values ranging from `0` to `100`. The diagonal elements of the matrix account for the misalignment effects for each axes. The off-diagonal elements account for the cross-axes misalignment effects. The measured state $v_{measure}$ is obtained from the true state $v_{measure}$ via the misalignment matrix as:

$$v_{measure} = \frac{1}{100} M v_{true} = \frac{1}{100} \begin{bmatrix} m_{11} & m_{12} & m_{13} \\ m_{21} & m_{22} & m_{23} \\ m_{31} & m_{32} & m_{33} \end{bmatrix} v_{true}$$

- If you specify the property as a scalar, then all the off-diagonal elements of the matrix take the value of the specified scalar and all the diagonal elements are 100.
- If you specify the property as a vector [*a b c*], then $m_{21} = m_{31} = a$, $m_{12} = m_{32} = b$, and $m_{13} = m_{23} = c$. All the diagonal elements are 100.

Data Types: `single` | `double`

`NoiseDensity` — Power spectral density of sensor noise (µT/√Hz)
`[0 0 0]` (default) | real scalar | real 3-element row vector

Power spectral density of sensor noise in µT/√Hz, specified as a real scalar or 3-element row vector. Any scalar input is converted into a real 3-element row vector where each element has the input scalar value.

Data Types: `single` | `double`

`BiasInstability` — Instability of the bias offset (µT)
`[0 0 0]` (default) | real scalar | real 3-element row vector

Instability of the bias offset in µT, specified as a real scalar or 3-element row vector. Any scalar input is converted into a real 3-element row vector where each element has the input scalar value.

Data Types: `single` | `double`

`RandomWalk` — Integrated white noise of sensor (µT/√Hz)
`[0 0 0]` (default) | real scalar | real 3-element row vector

Integrated white noise of sensor in (µT/√Hz), specified as a real scalar or 3-element row vector. Any scalar input is converted into a real 3-element row vector where each element has the input scalar value.

Data Types: `single` | `double`

`TemperatureBias` — Sensor bias from temperature (µT/°C)
`[0 0 0]` (default) | real scalar | real 3-element row vector

Sensor bias from temperature in (µT/°C), specified as a real scalar or 3-element row vector. Any scalar input is converted into a real 3-element row vector where each element has the input scalar value.

Data Types: `single` | `double`

`TemperatureScaleFactor` — Scale factor error from temperature (%/°C)
`[0 0 0]` (default) | real scalar in the range [0,100] | real 3-element row vector in the range [0,100]

Scale factor error from temperature in (%/℃), specified as a real scalar or 3-element row vector with values ranging from 0 to 100. Any scalar input is converted into a real 3-element row vector where each element has the input scalar value.

Data Types: `single` | `double`

Examples

Generate Magnetometer Data from Stationary Inputs

Generate magnetometer data for an imuSensor object from stationary inputs.

Generate a magnetometer parameter object with a maximum sensor reading of 1200 µT and a resolution of 0.1 µT/LSB. The constant offset bias is 1 µT. The sensor has a power spectral density of $\left(\frac{[0.6\ 0.6\ 0.9]}{\sqrt{100}}\right)$ µT/$\sqrt{\text{Hz}}$. The bias from temperature is [0.8 0.8 2.4] µT/^{0}C. The scale factor error from temperature is 0.1 %/^{0}C.

```
params = magparams('MeasurementRange',1200,'Resolution',0.1,'ConstantBias',1,'NoiseDensity',[0.6
```

Use a sample rate of 100 Hz spaced out over 1000 samples. Create the imuSensor object using the magnetometer parameter object.

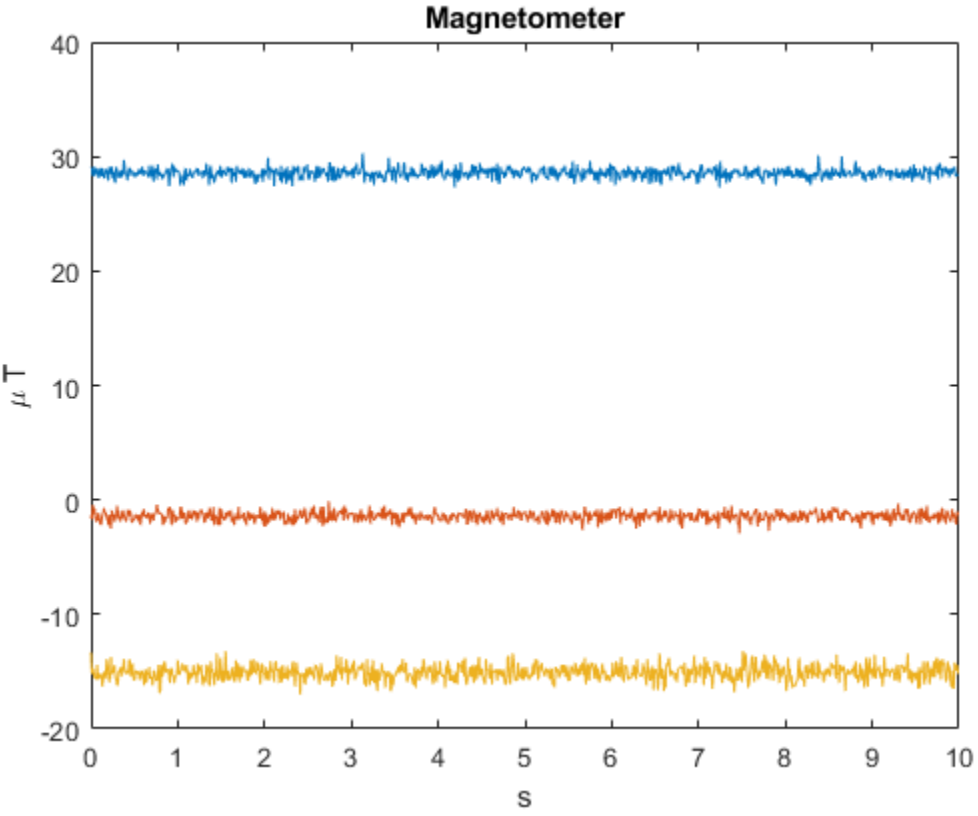
```
Fs = 100;
numSamples = 1000;
t = 0:1/Fs:(numSamples-1)/Fs;

imu = imuSensor('accel-mag','SampleRate', Fs, 'Magnetometer', params);
```

Generate magnetometer data from the imuSensor object.

```
orient = quaternion.ones(numSamples, 1);
acc = zeros(numSamples, 3);
angvel = zeros(numSamples, 3);

[~, magData] = imu(acc, angvel, orient);
```

Plot the resultant magnetometer data.

```
plot(t, magData)
title('Magnetometer')
xlabel('s')
ylabel('\mu T')
```



Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`accelparams` | `gyroparams` | `imuSensor`

Introduced in R2018b

mapLayer

Create map layer for *N*-dimensional data

Description

The `mapLayer` object creates an *N*-dimensional grid map, where the first two dimensions determine the footprint of the map, and all subsequent dimensions dictate the size and layout of the data stored in each cell. For storing scalar binary or probability values for a grid map, use the `binaryOccupancyMap` or `occupancyMap` objects instead.

A map layer stores data for grid cells that represent a discretized region of space. To query and update data using world, local, or grid coordinates, use the `getMapData` and `setMapData` object functions. Each grid cell in the map can store data of any size from a single a value to a multi-dimensional array. For more information, see the DataSize property.

Layer behavior can also be customized by providing function handles during creation using the `GetTransformFcn` and `SetTransformFcn` properties.

Creation

Syntax

```
map = mapLayer
map = mapLayer(p)
map = mapLayer(width,height)
map = mapLayer(rows,cols,'grid')
map = mapLayer(width,height,cellDims)
map = mapLayer(rows,cols,cellDims,'grid')
map = mapLayer(sourceMap)
map = mapLayer(___,Name,Value)
```

Description

`map = mapLayer` creates an empty map object occupying 10-by-10 meters of space with a resolution of 1 cell per meter.

`map = mapLayer(p)` creates a map from the values in the matrix or matrix array `p`. For 3-D matrix arrays, each cell in the map is filled with the vector of values at each grid location along the third dimension of the array. For an *N*-by-*D* matrix array, each cell contains a matrix (*N*=4) or a matrix array (*N*>4) of data for that grid location.

`map = mapLayer(width,height)` creates a map covering the specified width and height with a resolution of 1 cell per meter.

`map = mapLayer(rows,cols,'grid')` creates a map with a grid size of `rows`,`cols` with a resolution of 1 cell per meter.

`map = mapLayer(width,height,cellDims)` creates the map where the size of the data stored in each cell is defined by the array of integers `cellDims`.

`map = mapLayer(rows,cols,cellDims,'grid')` creates a map with a grid size of `rowscols` where the size of the data stored in each cell is defined by the array of integers `cellDims`.

`map = mapLayer(sourceMap)` creates a new object using the layers copied from another `mapLayer` object.

`map = mapLayer(___,Name,Value)` specifies property values using name-value pairs.

For example, `mapLayer(__,'LocalOriginInWorld',[15 20])` sets the local origin to a specific world location.

Properties

`DataSize` — Size of the *N*-dimensional data matrix

`[10 10]` (default) | vector of integers

Size of the *N*-dimensional data matrix, specified as vector of integers. The first two dimensions define the footprint of the map, and all subsequent dimensions dictate the size and layout of the data stored in each cell. The default value assumes a single value is stored for each cell in a 10-by-10 grid.

If the map stores an *n*-element vector of values in each cell, this property would be `[width height n]`.

If the map stores a 10-by-10 grid with each cell containing a 3-by-3-by-3 matrix array, the data size would be `[10 10 3 3 3]`.

This property is set when you create the object based on the dimensions of the input matrix `p` or the inputs `width`, `height`, and `cellDims`.

After you create the object, this property is read-only.

Data Types: `double`

`DataType` — Data type of the values stored

`'double'` (default) | character vector

Data type of the values stored in the map, specified as a character vector.

This property is set based on the data type of the input `p` or the data type of DefaultValue. After you create the object, this property is read-only.

Data Types: `char`

`DefaultValue` — Default value for unspecified map locations

`0` (default) | numeric scalar

Default value for unspecified map locations including areas outside the map, specified as a numeric scalar.

If you specify the GetTransformFcn or SetTransformFcn property when creating the object, the default value is updated based on that transformation function. If you create the map with a matrix of values `p`, the transform function modifies the values before storing.

Data Types: `double`

`GetTransformFcn` — Applies transformations to retrieved values
function handle

Applies transformations to values retrieved by the `getMapData` function, specified as a function handle.

This function handle is called inside the `getMapData` object function. It can be used to apply a transformation to values retrieved from the map layer. The function definition must have the following format:

```
modifiedValues = getTransformFcnHandle(map,values,varargin)
```

The size of the output `modifiedValues` must match the size of the input `values`. The function provides all map data accessed from the `getMapData` object function to this transform function through the `varargin` inputs.

You can set this property when you create the object. After you create the object, this property is read-only.

Data Types: `function_handle`

`GridLocationInWorld` — Location of the grid in local coordinates
`[0 0]` (default) | two-element vector | `[xWorld yWorld]`

Location of the bottom-left corner of the grid in world coordinates, specified as a two-element vector, `[xWorld yWorld]`.

You can set this property when you create the object.

Data Types: `double`

`GridOriginInLocal` — Location of the grid in local coordinates
`[0 0]` (default) | two-element vector | `[xLocal yLocal]`

Location of the bottom-left corner of the grid in local coordinates, specified as a two-element vector, `[xLocal yLocal]`.

You can set this property when you create the object.

Data Types: `double`

`GridSize` — Number of rows and columns in grid
two-element integer-valued vector

Number of rows and columns in grid, stored as a 1-by-2 real-valued vector representing the number of rows and columns, in that order.

This property is set when you create the object based on the first two dimensions of the input matrix `p`, the inputs `width` and `height`, or the inputs `row` and `col`.

Data Types: `double`

`LayerName` — Name of layer
`'mapLayer'` (default) | character vector | string scalar

Name of map layer, specified as a character vector or string scalar.

You can set this property when you create the object. After you create the object, this property is read-only.

Data Types: `double`

`LocalOriginInWorld` — Location of the local frame in world coordinates

`[0 0]` (default) | two-element vector | `[xWorld yWorld]`

Location of the origin of the local frame in world coordinates, specified as a two-element vector, `[xLocal yLocal]`. Use the `move` function to shift the local frame as your vehicle moves.

You can set this property when you create the object.

Data Types: `double`

`Resolution` — Grid resolution

`1` (default) | scalar

This property is read-only.

Grid resolution, stored as a scalar in cells per meter representing the number and size of grid locations.

You can set this property when you create the object. After you create the object, this property is read-only.

Data Types: `double`

`SetTransformFcn` — Applies transformations to set values

function handle

Applies transformations to values set by the `setMapData` function, specified as a function handle.

This function handle is called inside the `setMapData` object function. It can be used to apply a transformation to values set in the map layer. The function must have the following syntax:

```
modifiedValues = setTransformFcnHandle(map,values,varargin)
        if numel(varargin) == 0
            return; %
        else
          % Custom Code
        end
end
```

The size of the output, `modifiedValues`, must match the size of the input, `values`. The function provides all map data specified in the `setMapData` object function to this transform function. When creating this object without starting values, the function is called without additional input arguments, so specify an if-statement to return when the number of elements in `varagin` is zero.

You can set this property when you create the object. After you create the object, this property is read-only.

Data Types: `function_handle`

`XLocalLimits` — Minimum and maximum values of *x*-coordinates in local frame

two-element vector

This property is read-only.

Minimum and maximum values of *x*-coordinates in local frame, stored as a two-element horizontal vector of the form `[min max]`. Local frame is defined by `LocalOriginInWorld` property.

Data Types: `double`

`YLocalLimits` — Minimum and maximum values of *y*-coordinates in local frame
two-element vector

This property is read-only.

Minimum and maximum values of *y*-coordinates in local frame, stored as a two-element horizontal vector of the form `[min max]`. Local frame is defined by `LocalOriginInWorld` property.

Data Types: `double`

`XWorldLimits` — Minimum and maximum world range values of *x*-coordinates
two-element vector

This property is read-only.

Minimum and maximum world range values of *x*-coordinates, stored as a 1-by-2 vector representing the minimum and maximum values, in that order.

Data Types: `double`

`YWorldLimits` — Minimum and maximum world range values of *y*-coordinates
two-element vector

This property is read-only.

Minimum and maximum world range values of *y*-coordinates, stored as a 1-by-2 vector representing the minimum and maximum values, in that order.

Data Types: `double`

Object Functions

getMapData	Retrieve data from map layer
grid2local	Convert grid indices to local coordinates
grid2world	Convert grid indices to world coordinates
local2grid	Convert local coordinates to grid indices
local2world	Convert local coordinates to world coordinates
move	Move map in world frame
setMapData	Assign data to map layer
syncWith	Sync map with overlapping map
world2grid	Convert world coordinates to grid indices
world2local	Convert world coordinates to local coordinates

Examples

Store and Modify XY Velocities Using A Single Map Layer

Create a map layer that stores two values per grid location as *xy*-velocities.

Create an *m*-by-*n*-by-2 matrix of values. The first element in the third dimension is d*x* and the second is d*y* as velocities.

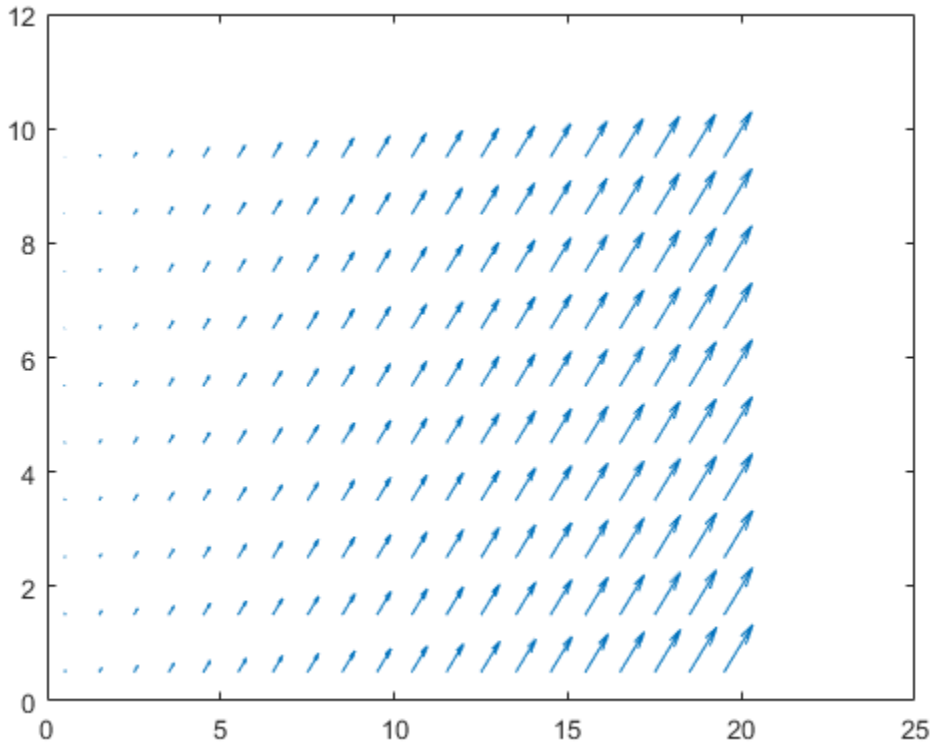
```
dXY = reshape(1:200,10,20);
dXY(:,:,2) = dXY;
```

Create a map layer from the matrix. Specify the resolution and layer name.

```
vLayer = mapLayer(dXY,'Resolution',1,'LayerName','dXY');
```

Get all the map data out as a matrix. Get the *xy*-locations of the velocity values by creating arrays that cover the minimum and maximum *xy*-world limits and is shifted to the grid-center locations. The *y*-locations are flipped when converting between matrix to world coordinates. Visualize the velocities corresponding to those grid-center locations using the `quiver` function.

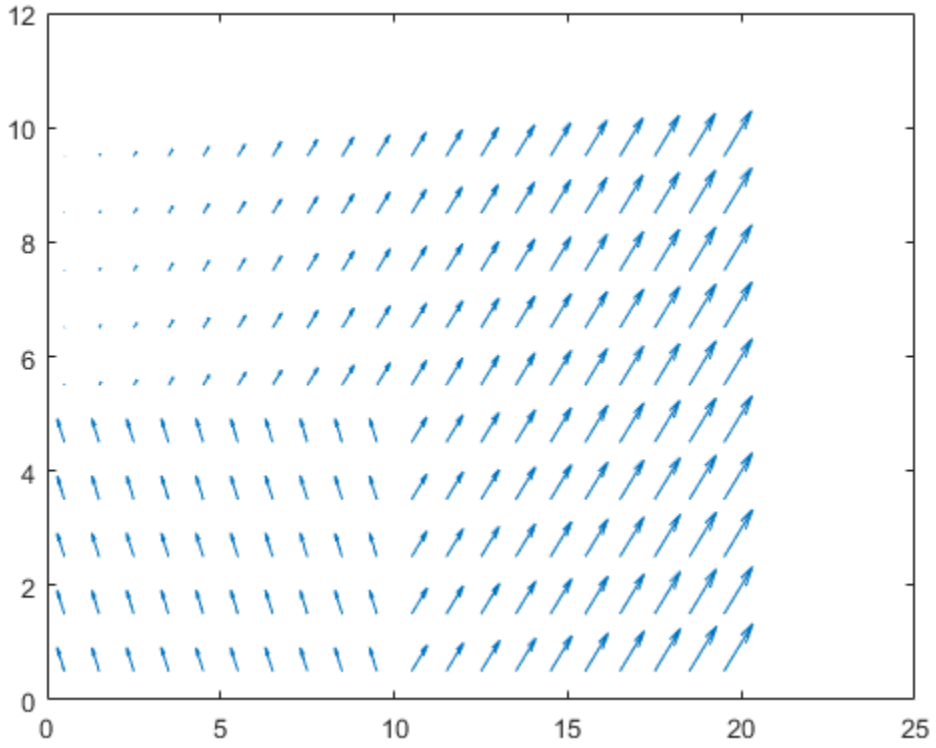
```
v = getMapData(vLayer);

R = 1/(2*vLayer.Resolution);
xLim = vLayer.XWorldLimits;
yLim = vLayer.YWorldLimits;
xLoc = (xLim(1)+R):R*2:(xLim(2)-R);
yLoc = (yLim(2)-R):-R*2:(yLim(1)+R);

quiver(xLoc,yLoc,v(:,:,1),v(:,:,2))
```



Set the bottom-left quadrant to new updated values. Create the values as a matrix and specify the bottom-left corner (0,0) in map coordinates to the `setData` function.

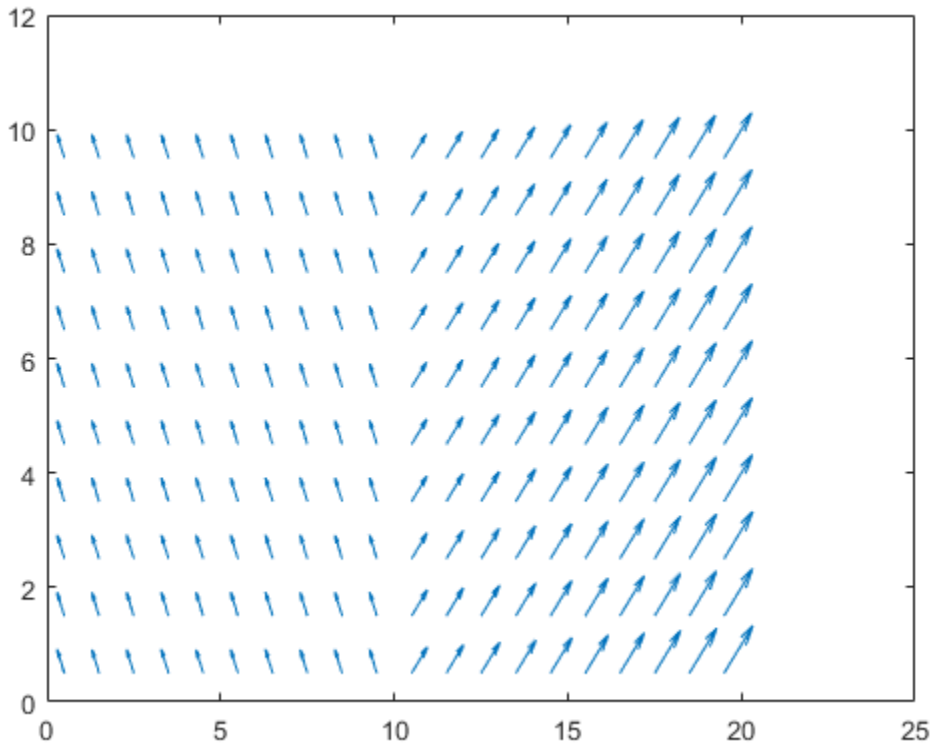
```
updateValues = repmat(reshape([-50,100],[1 1 2]),5,10);

setMapData(vLayer,[0 0],updateValues)
```

```
v = getMapData(vLayer);
quiver(xLoc,yLoc,v(:,:,1),v(:,:,2))
```



Set new values for the top-left quadrant using grid coordinates. For maps, the top-left grid location is (1,1).

```
setMapData(vLayer,[1 1],updateValues,'grid')
v = getMapData(vLayer);
quiver(xLoc,yLoc,v(:,:,1),v(:,:,2))
```



Write Custom Transform Functions for Map Layers

The `mapLayer` object enables you to apply custom element-wise transformations when setting and getting data in the map. To transform data you set or get from the map, specify function handles for the `GetTransfomFcn` and `SetTransformFcn` properties. This example shows how to implement a log-odds probabilitistic map layer by creating a lookup table for probability and log-odds values. The transform functions use these lookup tables to convert between these values when setting or getting data.

Create Lookup Tables

Generate a full lookup table of values that map the probability values to the minimum and maximum limits of `int16` values.

Create an array of `int16` values from `intmin` to `intmax`. Define the probablilty limits.

```
intType = 'int16';
intLinSpace = intmin(intType):intmax(intType);
numOfPoints = length(intLinSpace);
probLimits = [0.001 .999];
```

The `exampleHelperProbToLogodds` and `examplerHelperLogoddsToProb` functions covert between the log-odds and probability values. Use the helper functions to get the log-odds limits and

generate the array for looking up log-odds values. Create an interpolated grid for the entire lookup table.

```
logOddsLimits = exampleHelperProbToLogodds([0.001 .999]);
logOddsLookup = single(exampleHelperLogoddsToProb(linspace(logOddsLimits(1),logOddsLimits(2),num(
interpTable = griddedInterpolant(logOddsLookup,single(intLinSpace),'nearest');
```

Specify Transform Function Handles

The transform function handles utilize example helpers that define how to convert between log-odds integer values and the probability values with an applied saturation limit. The probability saturation limits are `[0.001 .999]` as previously specified. This behavior is similar to the `occupancyMap` object.

```
getXformFcn = @(obj,logodds,varargin)...
    exampleHelperIntLogoddsToProb(logodds,logOddsLookup(:),intLinSpace);

setXformFcn = @(obj,prob,varargin)...
    exampleHelperProbToIntLogodds(prob,interpTable,logOddsLookup(:),intLinSpace,probLimits);
```

Create Map Layer

Generate an occupancy map layer object from a matrix of probability values. Specify the get and set transform functions.

```
occupancyLayer = mapLayer(repmat(0.5,10,10),...
                          'LayerName','Occupancy',...
                          'GetTransformFcn',getXformFcn,...
                          'SetTransformFcn',setXformFcn);
```

Notice that when you create the map, the default value is 0.001 instead of 0. This difference is because the `SetTransformFcn` function has been applied to the default value of 0 when you create the object, which saturates the value to 0.001.

```
disp(occupancyLayer.DefaultValue)
```

```
    0.0010
```

Get and Set Map Data

The map data matches the matrix you set on creation.

```
extData = getMapData(occupancyLayer)
```

```
extData = 10×10

    0.5000    0.5000    0.5000    0.5000    0.5000    0.5000    0.5000    0.5000    0.5000    0.5
    0.5000    0.5000    0.5000    0.5000    0.5000    0.5000    0.5000    0.5000    0.5000    0.5
    0.5000    0.5000    0.5000    0.5000    0.5000    0.5000    0.5000    0.5000    0.5000    0.5
    0.5000    0.5000    0.5000    0.5000    0.5000    0.5000    0.5000    0.5000    0.5000    0.5
    0.5000    0.5000    0.5000    0.5000    0.5000    0.5000    0.5000    0.5000    0.5000    0.5
    0.5000    0.5000    0.5000    0.5000    0.5000    0.5000    0.5000    0.5000    0.5000    0.5
    0.5000    0.5000    0.5000    0.5000    0.5000    0.5000    0.5000    0.5000    0.5000    0.5
    0.5000    0.5000    0.5000    0.5000    0.5000    0.5000    0.5000    0.5000    0.5000    0.5
    0.5000    0.5000    0.5000    0.5000    0.5000    0.5000    0.5000    0.5000    0.5000    0.5
    0.5000    0.5000    0.5000    0.5000    0.5000    0.5000    0.5000    0.5000    0.5000    0.5
```

Set specific map locations to values that are:

- Outside of the probability saturation limits.
- Higher precision than the resolution of the lookup tables.

```
setMapData(occupancyLayer,[0 0],0.00001)
setMapData(occupancyLayer,[5 5],0.25999)
```

For the first location, the probability is bound to the saturation limits.

```
extData = getMapData(occupancyLayer,[0 0])
```

```
extData = 0.0010
```

The second location returns the value closest to the probability value in the lookup table.

```
extData2 = getMapData(occupancyLayer,[5 5])
```

```
extData2 = 0.2600
```

The generated map layer can now be used for updating a probability occupany map that are stored as `int16` values. To combine this map with other layers or map types, see the `multiLayerMap` object.

Limitations

- `mapLayer` objects can only belong to one `multiLayerMap` object at a time.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

- The LayerName property must be fixed at compile-time
- The DataType property must be known at compile-time.

See Also

Objects

`multiLayerMap` | `occupancyMap3D` | `occupancyMap` | `binaryOccupancyMap`

Functions

`getMapData` | `setMapData` | `move` | `syncWith`

Topics

“Motion Planning in Urban Environments Using Dynamic Occupancy Grid Map”
“Fuse Multiple Lidar Sensors Using Map Layers”

Introduced in R2021a

getMapData

Retrieve data from map layer

Syntax

```
mapData = getMapData(map)
mapData = getMapData(map,xy)
mapData = getMapData(map,xy,'local')
mapData = getMapData(map,ij,'grid')
[mapData,inBounds] = getMapData( ___ )

mapData = getMapData(map,bottomLeft,mapSize)
mapData = getMapData(map,bottomLeft,mapSize,'local')
mapData = getMapData(map,topLeft,gridSize,'grid')
```

Description

`mapData = getMapData(map)` returns a matrix of values that contains all the data for the given map layer `map`.

`mapData = getMapData(map,xy)` returns an array of values for the given *xy*-locations in world coordinates.

`mapData = getMapData(map,xy,'local')` returns an array of values for the given *xy*-locations in local coordinates.

`mapData = getMapData(map,ij,'grid')` returns an array of values for the given *ij*-locations in grid coordinates. Each row of `ij` refers to a grid cell index `[i j]`

`[mapData,inBounds] = getMapData( ___ )` also returns a vector of logical values indicating whether the corresponding input location `xy` or `ij` is valid using the previous syntaxes.

`mapData = getMapData(map,bottomLeft,mapSize)` returns a matrix of values in a subregion of the map layer, `map`. The subregion starts in the bottom-left *xy*-position `bottomLeft` in world coordinates with a given map size `mapSize` specified as `[width height]` in meters.

`mapData = getMapData(map,bottomLeft,mapSize,'local')` specifies the bottom-left corner of the subregion in local coordinates.

`mapData = getMapData(map,topLeft,gridSize,'grid')` specifies the top-left corner of the sub region in grid coordinates. The subregion size, `gridSize` is also given in grid coordinates as `[rows cols]`.

Examples

Store and Modify XY Velocities Using A Single Map Layer

Create a map layer that stores two values per grid location as *xy*-velocities.

Create an *m*-by-*n*-by-2 matrix of values. The first element in the third dimension is d*x* and the second is d*y* as velocities.

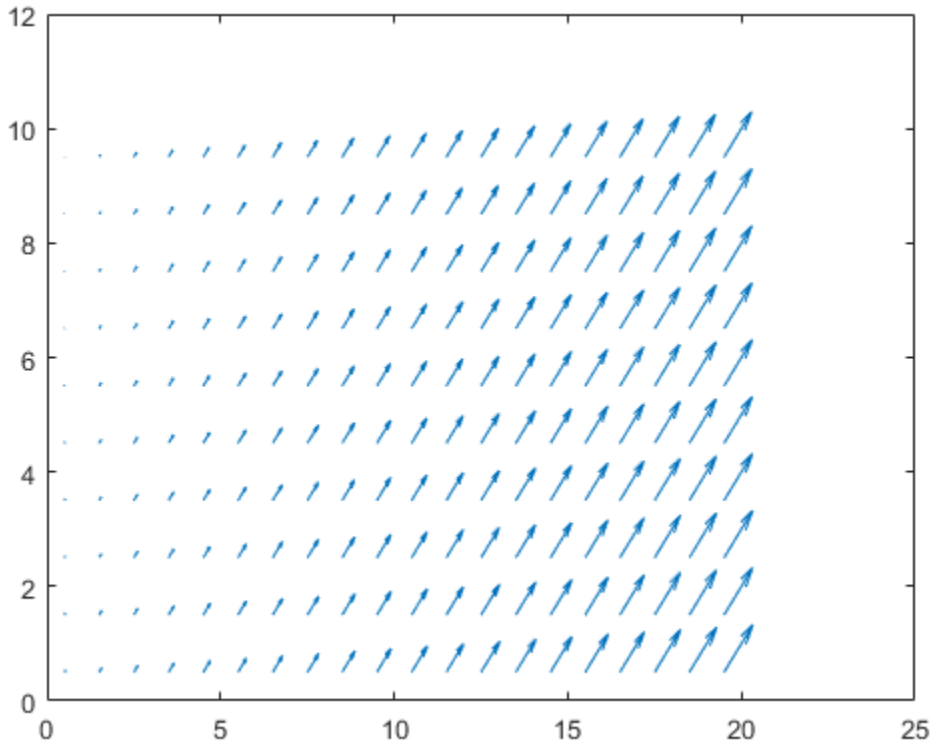
```
dXY = reshape(1:200,10,20);
dXY(:,:,2) = dXY;
```

Create a map layer from the matrix. Specify the resolution and layer name.

```
vLayer = mapLayer(dXY,'Resolution',1,'LayerName','dXY');
```

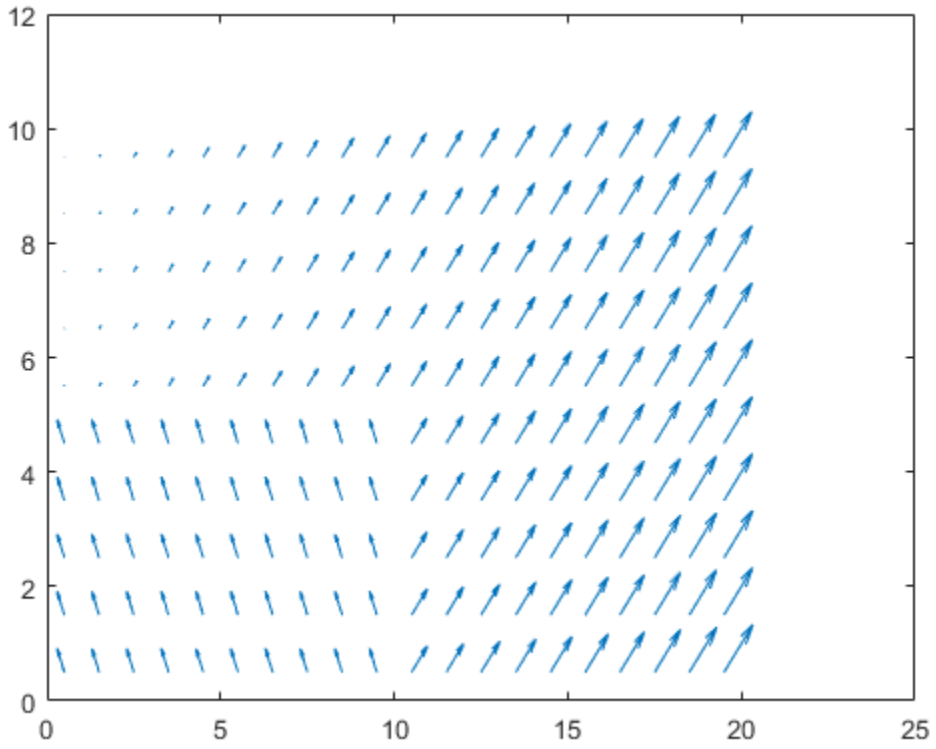
Get all the map data out as a matrix. Get the xy-locations of the velocity values by creating arrays that cover the minimum and maximum *xy*-world limits and is shifted to the grid-center locations. The *y*-locations are flipped when converting between matrix to world coordinates. Visualize the velocities corresponding to those grid-center locations using the `quiver` function.

```
v = getMapData(vLayer);

R = 1/(2*vLayer.Resolution);
xLim = vLayer.XWorldLimits;
yLim = vLayer.YWorldLimits;
xLoc = (xLim(1)+R):R*2:(xLim(2)-R);
yLoc = (yLim(2)-R):-R*2:(yLim(1)+R);

quiver(xLoc,yLoc,v(:,:,1),v(:,:,2))
```



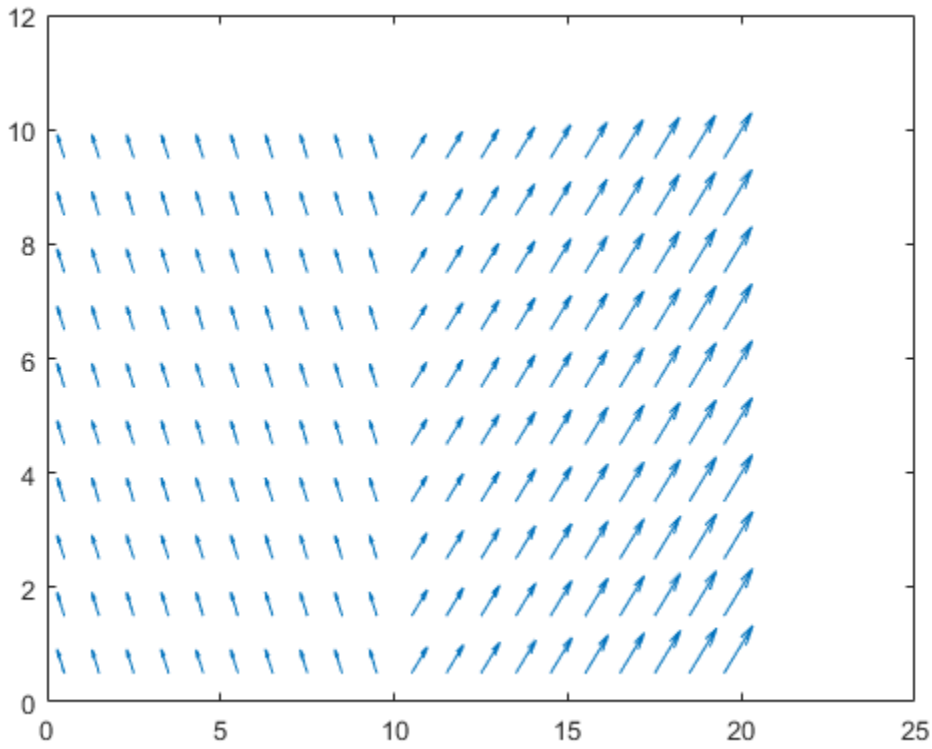
Set the bottom-left quadrant to new updated values. Create the values as a matrix and specify the bottom-left corner (0,0) in map coordinates to the `setData` function.

```
updateValues = repmat(reshape([-50,100],[1 1 2]),5,10);

setMapData(vLayer,[0 0],updateValues)
v = getMapData(vLayer);
quiver(xLoc,yLoc,v(:,:,1),v(:,:,2))
```



Set new values for the top-left quadrant using grid coordinates. For maps, the top-left grid location is (1,1).

```
setMapData(vLayer,[1 1],updateValues,'grid')
v = getMapData(vLayer);
quiver(xLoc,yLoc,v(:,:,1),v(:,:,2))
```



Write Custom Transform Functions for Map Layers

The `mapLayer` object enables you to apply custom element-wise transformations when setting and getting data in the map. To transform data you set or get from the map, specify function handles for the `GetTransfomFcn` and `SetTransformFcn` properties. This example shows how to implement a log-odds probabilitistic map layer by creating a lookup table for probability and log-odds values. The transform functions use these lookup tables to convert between these values when setting or getting data.

Create Lookup Tables

Generate a full lookup table of values that map the probability values to the minimum and maximum limits of `int16` values.

Create an array of `int16` values from `intmin` to `intmax`. Define the probablilty limits.

```
intType = 'int16';
intLinSpace = intmin(intType):intmax(intType);
numOfPoints = length(intLinSpace);
probLimits = [0.001 .999];
```

The `exampleHelperProbToLogodds` and `examplerHelperLogoddsToProb` functions covert between the log-odds and probability values. Use the helper functions to get the log-odds limits and

generate the array for looking up log-odds values. Create an interpolated grid for the entire lookup table.

```
logOddsLimits = exampleHelperProbToLogodds([0.001 .999]);
logOddsLookup = single(exampleHelperLogoddsToProb(linspace(logOddsLimits(1),logOddsLimits(2),num
interpTable = griddedInterpolant(logOddsLookup,single(intLinSpace),'nearest');
```

Specify Transform Function Handles

The transform function handles utilize example helpers that define how to convert between log-odds integer values and the probability values with an applied saturation limit. The probability saturation limits are `[0.001 .999]` as previously specified. This behavior is similar to the `occupancyMap` object.

```
getXformFcn = @(obj,logodds,varargin)...
    exampleHelperIntLogoddsToProb(logodds,logOddsLookup(:),intLinSpace);

setXformFcn = @(obj,prob,varargin)...
    exampleHelperProbToIntLogodds(prob,interpTable,logOddsLookup(:),intLinSpace,probLimits);
```

Create Map Layer

Generate an occupancy map layer object from a matrix of probability values. Specify the get and set transform functions.

```
occupancyLayer = mapLayer(repmat(0.5,10,10),...
                           'LayerName','Occupancy',...
                           'GetTransformFcn',getXformFcn,...
                           'SetTransformFcn',setXformFcn);
```

Notice that when you create the map, the default value is 0.001 instead of 0. This difference is because the `SetTransformFcn` function has been applied to the default value of 0 when you create the object, which saturates the value to 0.001.

```
disp(occupancyLayer.DefaultValue)

    0.0010
```

Get and Set Map Data

The map data matches the matrix you set on creation.

```
extData = getMapData(occupancyLayer)

extData = 10×10

    0.5000    0.5000    0.5000    0.5000    0.5000    0.5000    0.5000    0.5000    0.5000    0.5
    0.5000    0.5000    0.5000    0.5000    0.5000    0.5000    0.5000    0.5000    0.5000    0.5
    0.5000    0.5000    0.5000    0.5000    0.5000    0.5000    0.5000    0.5000    0.5000    0.5
    0.5000    0.5000    0.5000    0.5000    0.5000    0.5000    0.5000    0.5000    0.5000    0.5
    0.5000    0.5000    0.5000    0.5000    0.5000    0.5000    0.5000    0.5000    0.5000    0.5
    0.5000    0.5000    0.5000    0.5000    0.5000    0.5000    0.5000    0.5000    0.5000    0.5
    0.5000    0.5000    0.5000    0.5000    0.5000    0.5000    0.5000    0.5000    0.5000    0.5
    0.5000    0.5000    0.5000    0.5000    0.5000    0.5000    0.5000    0.5000    0.5000    0.5
    0.5000    0.5000    0.5000    0.5000    0.5000    0.5000    0.5000    0.5000    0.5000    0.5
    0.5000    0.5000    0.5000    0.5000    0.5000    0.5000    0.5000    0.5000    0.5000    0.5
```

Set specific map locations to values that are:

- Outside of the probability saturation limits.
- Higher precision than the resolution of the lookup tables.

```
setMapData(occupancyLayer,[0 0],0.00001)
setMapData(occupancyLayer,[5 5],0.25999)
```

For the first location, the probability is bound to the saturation limits.

```
extData = getMapData(occupancyLayer,[0 0])
```

```
extData = 0.0010
```

The second location returns the value closest to the probability value in the lookup table.

```
extData2 = getMapData(occupancyLayer,[5 5])
```

```
extData2 = 0.2600
```

The generated map layer can now be used for updating a probability occupany map that are stored as `int16` values. To combine this map with other layers or map types, see the `multiLayerMap` object.

Input Arguments

`map` — Map layer

`mapLayer` object

Map layer, specified as a `mapLayer` object.

`xy` — World or local coordinates

n-by-2 matrix

World or local coordinates, specified as an *n*-by-2 matrix of `[x y]` pairs, where *n* is the number of coordinates.

Data Types: `double`

`ij` — Grid positions

n-by-2 matrix

Grid positions, specified as an *n*-by-2 matrix of `[i j]` pairs in `[rows cols]` format, where *n* is the number of grid positions.

Data Types: `double`

`bottomLeft` — Location of output matrix in world or local coordinates

two-element vector | `[xCoord yCoord]`

Location of bottom left corner of output matrix in world or local coordinates, specified as a two-element vector, `[xCoord yCoord]`. Location is in world or local coordinates based on syntax.

Data Types: `double`

`mapSize` — Subregion map size

two-element vector | `[x y]`

Subregion map size, specified as a two-element vector `[x y]` in world or local coordinates.

Data Types: `double`

`gridSize` — Output grid size
two-element vector | `[row col]`

Output grid size, specified as a two-element vector `[row col]`.

Data Types: `double`

`topLeft` — Location of grid
two-element vector | `[iCoord jCoord]`

Location of top left corner of grid, specified as a two-element vector, `[iCoord jCoord]`.

Data Types: `double`

Output Arguments

`mapData` — Data values from map layer
matrix

Data values from map layer, returned as a matrix. By default, the function returns all data on the layer as an *M*-by-*N*-by-*DataDims* matrix. *M* and *N* are the grid rows and columns respectively. *DataDims* are the dimensions of the map data, `map.DataSize(3:end)`.

For other syntaxes, the map data may be returned as an array of values with size *N*-by-*DataDims* or as a subregion of the full matrix.

`inBounds` — Valid map locations
n-by-1 column vector

Valid map locations, returned as an *n*-by-1 column vector equal in length to `xy` or `ij`. Locations inside the map limits return a value of 1. Locations outside the map limits return a value of 0.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

Objects
`multiLayerMap` | `occupancyMap3D` | `occupancyMap` | `binaryOccupancyMap`

Functions
`setMapData` | `move` | `syncWith`

Introduced in R2021a

setMapData

Assign data to map layer

Syntax

```
setMapData(map,mapData)
setMapData(map,xy,mapData)
setMapData(map,xy,mapData,'local')
setMapData(map,ij,mapData,'grid')
inBounds = setMapData( ___ )

setMapData(map,bottomLeft,mapData)
setMapData(map,bottomLeft,mapData,'local')
setMapData(map,topLeft,mapData,'grid')
```

Description

`setMapData(map,mapData)` overwrites all values in the map layer using a matrix with dimensions that match the map layer data dimensions, `map.DataSize`.

`setMapData(map,xy,mapData)` specifies an array of values for the given *xy*-locations in world coordinates. The `mapData` input must be an *X*-by-1-by-*DataDims* array. *DataDims* are the dimensions of the map data, `map.DataSize(3:end)`. Locations outside the map boundaries are ignored.

`setMapData(map,xy,mapData,'local')` specifies locations in local coordinates.

`setMapData(map,ij,mapData,'grid')` specifies an array of values for the given *ij*-locations in grid coordinates. Each row of `ij` refers to a grid cell index `[i j]`

`inBounds = setMapData( ___ )` also returns a vector of logical values indicating whether the corresponding input location `xy` or `ij` is valid using the previous syntaxes.

`setMapData(map,bottomLeft,mapData)` specifies a matrix of values `mapData` for a subregion of the map layer, `map`. The subregion starts in the bottom-left *xy*-position `bottomLeft` and updates a subregion based on the size of `mapData`.

`setMapData(map,bottomLeft,mapData,'local')` specifies the bottom-left corner of the subregion in local coordinates.

`setMapData(map,topLeft,mapData,'grid')` specifies the top-left corner of a sub region in grid coordinates. The subregion is updated with values in `mapData`.

Examples

Store and Modify XY Velocities Using A Single Map Layer

Create a map layer that stores two values per grid location as *xy*-velocities.

Create an *m*-by-*n*-by-2 matrix of values. The first element in the third dimension is d*x* and the second is d*y* as velocities.

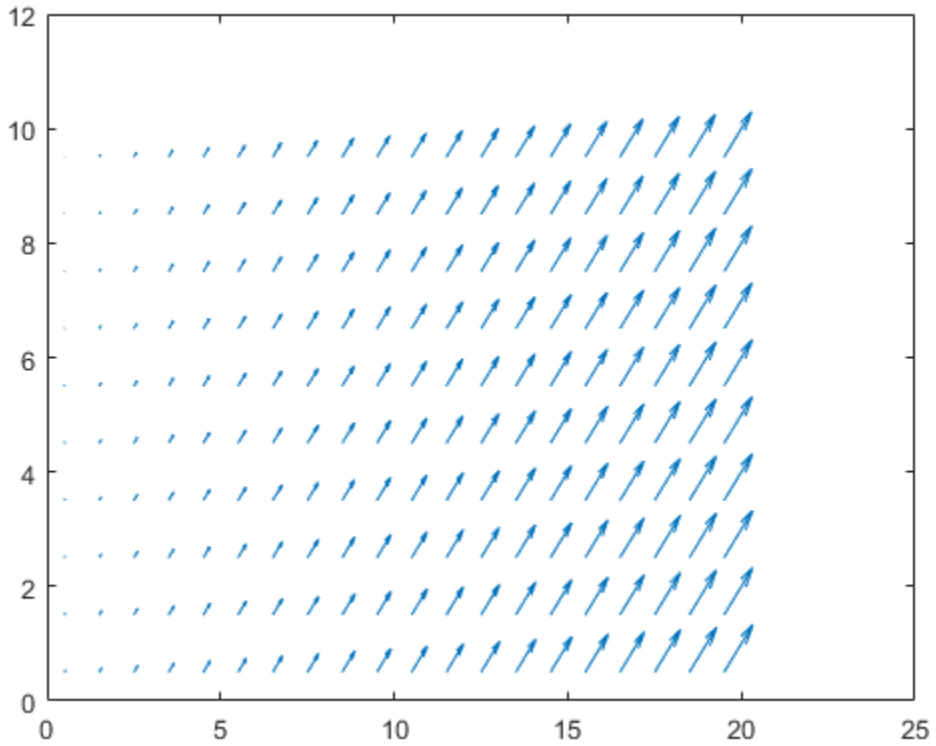
```
dXY = reshape(1:200,10,20);
dXY(:,:,2) = dXY;
```

Create a map layer from the matrix. Specify the resolution and layer name.

```
vLayer = mapLayer(dXY,'Resolution',1,'LayerName','dXY');
```

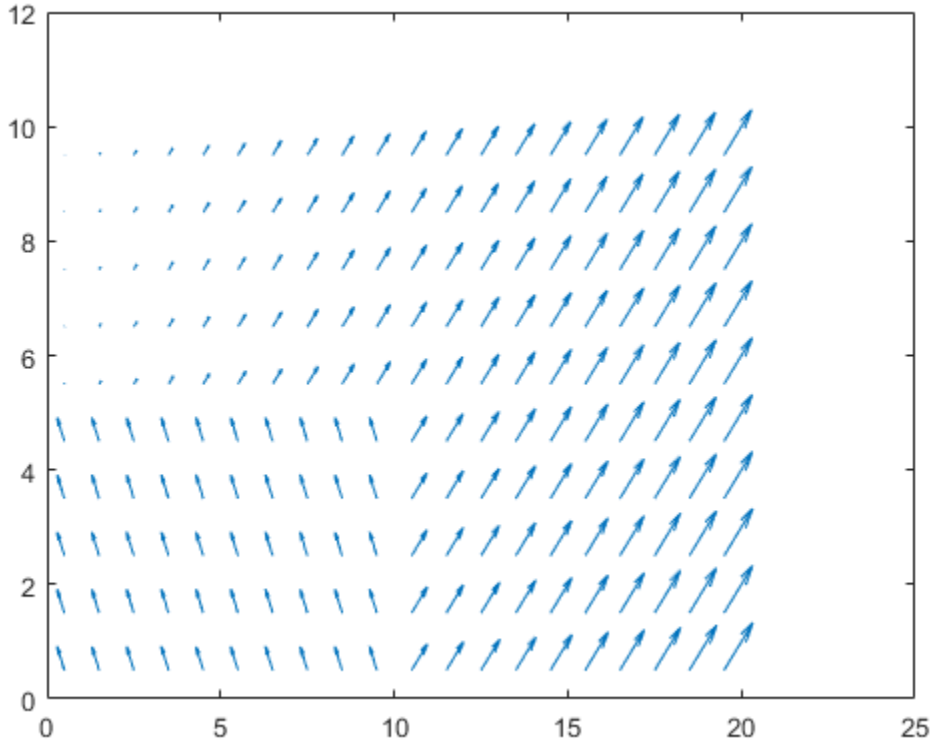
Get all the map data out as a matrix. Get the xy-locations of the velocity values by creating arrays that cover the minimum and maximum *xy*-world limits and is shifted to the grid-center locations. The *y*-locations are flipped when converting between matrix to world coordinates. Visualize the velocities corresponding to those grid-center locations using the `quiver` function.

```
v = getMapData(vLayer);

R = 1/(2*vLayer.Resolution);
xLim = vLayer.XWorldLimits;
yLim = vLayer.YWorldLimits;
xLoc = (xLim(1)+R):R*2:(xLim(2)-R);
yLoc = (yLim(2)-R):-R*2:(yLim(1)+R);

quiver(xLoc,yLoc,v(:,:,1),v(:,:,2))
```



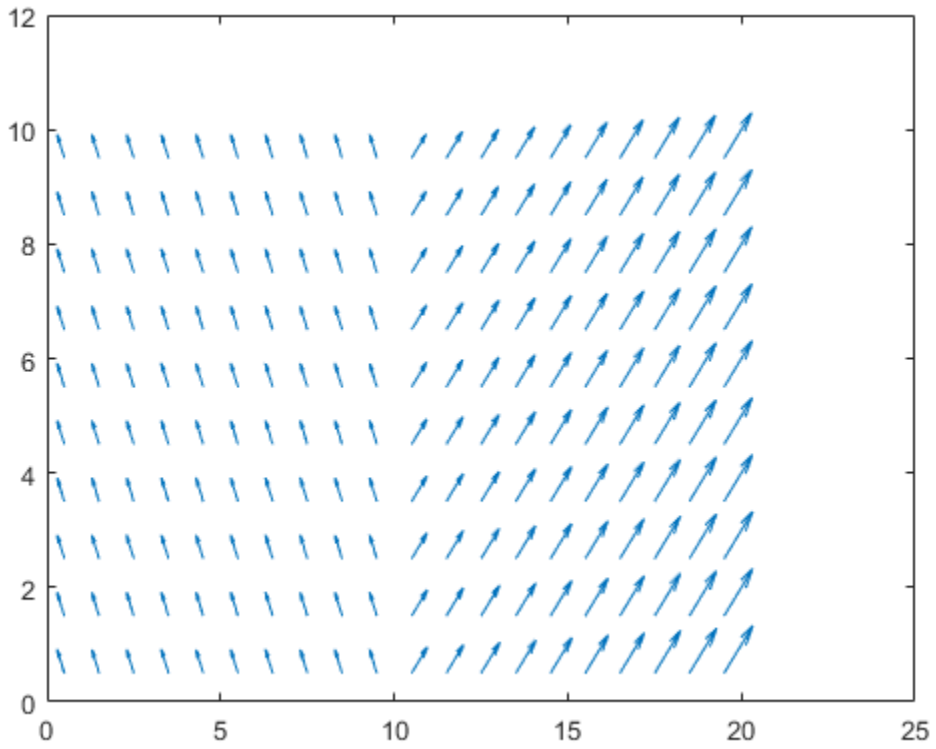
Set the bottom-left quadrant to new updated values. Create the values as a matrix and specify the bottom-left corner (0,0) in map coordinates to the `setData` function.

```
updateValues = repmat(reshape([-50,100],[1 1 2]),5,10);

setMapData(vLayer,[0 0],updateValues)
v = getMapData(vLayer);
quiver(xLoc,yLoc,v(:,:,1),v(:,:,2))
```



Set new values for the top-left quadrant using grid coordinates. For maps, the top-left grid location is (1,1).

```
setMapData(vLayer,[1 1],updateValues,'grid')
v = getMapData(vLayer);
quiver(xLoc,yLoc,v(:,:,1),v(:,:,2))
```



Write Custom Transform Functions for Map Layers

The `mapLayer` object enables you to apply custom element-wise transformations when setting and getting data in the map. To transform data you set or get from the map, specify function handles for the `GetTransfomFcn` and `SetTransformFcn` properties. This example shows how to implement a log-odds probabilitistic map layer by creating a lookup table for probability and log-odds values. The transform functions use these lookup tables to convert between these values when setting or getting data.

Create Lookup Tables

Generate a full lookup table of values that map the probability values to the minimum and maximum limits of `int16` values.

Create an array of `int16` values from `intmin` to `intmax`. Define the probablilty limits.

```
intType = 'int16';
intLinSpace = intmin(intType):intmax(intType);
numOfPoints = length(intLinSpace);
probLimits = [0.001 .999];
```

The `exampleHelperProbToLogodds` and `examplerHelperLogoddsToProb` functions covert between the log-odds and probability values. Use the helper functions to get the log-odds limits and

generate the array for looking up log-odds values. Create an interpolated grid for the entire lookup table.

```
logOddsLimits = exampleHelperProbToLogodds([0.001 .999]);
logOddsLookup = single(exampleHelperLogoddsToProb(linspace(logOddsLimits(1),logOddsLimits(2),num
interpTable = griddedInterpolant(logOddsLookup,single(intLinSpace),'nearest');
```

Specify Transform Function Handles

The transform function handles utilize example helpers that define how to convert between log-odds integer values and the probability values with an applied saturation limit. The probability saturation limits are `[0.001 .999]` as previously specified. This behavior is similar to the `occupancyMap` object.

```
getXformFcn = @(obj,logodds,varargin)...
    exampleHelperIntLogoddsToProb(logodds,logOddsLookup(:),intLinSpace);

setXformFcn = @(obj,prob,varargin)...
    exampleHelperProbToIntLogodds(prob,interpTable,logOddsLookup(:),intLinSpace,probLimits);
```

Create Map Layer

Generate an occupancy map layer object from a matrix of probability values. Specify the get and set transform functions.

```
occupancyLayer = mapLayer(repmat(0.5,10,10),...
                           'LayerName','Occupancy',...
                           'GetTransformFcn',getXformFcn,...
                           'SetTransformFcn',setXformFcn);
```

Notice that when you create the map, the default value is 0.001 instead of 0. This difference is because the `SetTransformFcn` function has been applied to the default value of 0 when you create the object, which saturates the value to 0.001.

```
disp(occupancyLayer.DefaultValue)
```

```
    0.0010
```

Get and Set Map Data

The map data matches the matrix you set on creation.

```
extData = getMapData(occupancyLayer)
```

```
extData = 10×10

    0.5000    0.5000    0.5000    0.5000    0.5000    0.5000    0.5000    0.5000    0.5000    0.5
    0.5000    0.5000    0.5000    0.5000    0.5000    0.5000    0.5000    0.5000    0.5000    0.5
    0.5000    0.5000    0.5000    0.5000    0.5000    0.5000    0.5000    0.5000    0.5000    0.5
    0.5000    0.5000    0.5000    0.5000    0.5000    0.5000    0.5000    0.5000    0.5000    0.5
    0.5000    0.5000    0.5000    0.5000    0.5000    0.5000    0.5000    0.5000    0.5000    0.5
    0.5000    0.5000    0.5000    0.5000    0.5000    0.5000    0.5000    0.5000    0.5000    0.5
    0.5000    0.5000    0.5000    0.5000    0.5000    0.5000    0.5000    0.5000    0.5000    0.5
    0.5000    0.5000    0.5000    0.5000    0.5000    0.5000    0.5000    0.5000    0.5000    0.5
    0.5000    0.5000    0.5000    0.5000    0.5000    0.5000    0.5000    0.5000    0.5000    0.5
    0.5000    0.5000    0.5000    0.5000    0.5000    0.5000    0.5000    0.5000    0.5000    0.5
```

Set specific map locations to values that are:

- Outside of the probability saturation limits.
- Higher precision than the resolution of the lookup tables.

```
setMapData(occupancyLayer,[0 0],0.00001)
setMapData(occupancyLayer,[5 5],0.25999)
```

For the first location, the probability is bound to the saturation limits.

```
extData = getMapData(occupancyLayer,[0 0])
```

```
extData = 0.0010
```

The second location returns the value closest to the probability value in the lookup table.

```
extData2 = getMapData(occupancyLayer,[5 5])
```

```
extData2 = 0.2600
```

The generated map layer can now be used for updating a probability occupany map that are stored as `int16` values. To combine this map with other layers or map types, see the `multiLayerMap` object.

Input Arguments

`map` — Map layer

`maLayer` object

Map layer, specified as a `maLayer` object.

`mapData` — Data values for setting map layer

matrix

Data values for setting map layer, specified as a matrix. By default, the function sets all data on the layer as an *M*-by-*N*-by-*DataDims* matrix. *M* and *N* are the grid height and width respectively. *DataDims* are the dimensions of the map data, `map.DataSize(3,:)`.

For other syntaxes, the map data may be specified as a matrix with size *N*-by-*DataDims*, where *N* is the number of elements in `xy` or `ij`, or as a subregion of the full matrix.

`topLeft` — Location of grid

two-element vector | `[iCoord jCoord]`

Location of top left corner of grid, specified as a two-element vector, `[iCoord jCoord]`.

Data Types: `double`

`bottomLeft` — Location of output matrix in world or local coordinates

two-element vector | `[xCoord yCoord]`

Location of bottom left corner of output matrix in world or local coordinates, specified as a two-element vector, `[xCoord yCoord]`. Location is in world or local coordinates based on syntax.

Data Types: `double`

`ij` — Grid positions

n-by-2 matrix

Grid positions, specified as an *n*-by-2 matrix of `[i j]` pairs in `[rows cols]` format, where *n* is the number of grid positions.

Data Types: `double`

`xy` — World or local coordinates
n-by-2 matrix

World or local coordinates, specified as an *n*-by-2 matrix of `[x y]` pairs, where *n* is the number of coordinates.

Data Types: `double`

Output Arguments

`inBounds` — Valid map locations
n-by-1 column vector

Valid map locations, returned as an *n*-by-1 column vector equal in length to `xy` or `ij`. Locations inside the map limits return a value of 1. Locations outside the map limits return a value of 0.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

Objects
`multiLayerMap` | `occupancyMap3D` | `occupancyMap` | `binaryOccupancyMap`

Functions
`getMapData` | `setMapData` | `move` | `syncWith`

Introduced in R2021a

insfilterMARG

Estimate pose from MARG and GPS data

Description

The `insfilterMARG` object implements sensor fusion of MARG and GPS data to estimate pose in the NED (or ENU) reference frame. MARG (magnetic, angular rate, gravity) data is typically derived from magnetometer, gyroscope, and accelerometer sensors. The filter uses a 22-element state vector to track the orientation quaternion, velocity, position, MARG sensor biases, and geomagnetic vector. The `insfilterMARG` object uses an extended Kalman filter to estimate these quantities.

Creation

Syntax

```
filter = insfilterMARG
filter = insfilterMARG('ReferenceFrame',RF)
filter = insfilterMARG(___,Name,Value)
```

Description

`filter = insfilterMARG` creates an `insfilterMARG` object with default property values.

`filter = insfilterMARG('ReferenceFrame',RF)` allows you to specify the reference frame, `RF`, of the `filter`. Specify `RF` as `'NED'` (North-East-Down) or `'ENU'` (East-North-Up). The default value is `'NED'`.

`filter = insfilterMARG(___,Name,Value)` also allows you set properties of the created `filter` using one or more name-value pairs. Enclose each property name in single quotes.

Properties

`IMUSampleRate` — Sample rate of the IMU (Hz)
`100` (default) | positive scalar

Sample rate of the inertial measurement unit (IMU) in Hz, specified as a positive scalar.

Data Types: `single` | `double`

`ReferenceLocation` — Reference location (deg, deg, meters)
`[0 0 0]` (default) | 3-element positive row vector

Reference location, specified as a 3-element row vector in geodetic coordinates (latitude, longitude, and altitude). Altitude is the height above the reference ellipsoid model, WGS84. The reference location units are [degrees degrees meters].

Data Types: `single` | `double`

`GyroscopeNoise` — Multiplicative process noise variance from gyroscope (rad/s)²
`1e-9` (default) | scalar | 3-element row vector

Multiplicative process noise variance from the gyroscope in $(rad/s)^2$, specified as a scalar or 3-element row vector of positive real finite numbers.

- If `GyroscopeNoise` is specified as a row vector, the elements correspond to the noise in the *x*, *y*, and *z* axes of the gyroscope, respectively.
- If `GyroscopeNoise` is specified as a scalar, the single element is applied to the *x*, *y*, and *z* axes of the gyroscope.

Data Types: `single` | `double`

`GyroscopeBiasNoise` — Multiplicative process noise variance from gyroscope bias (rad/s)²
`1e-10` (default) | positive scalar | 3-element row vector

Multiplicative process noise variance from the gyroscope bias in $(rad/s)^2$, specified as a scalar or 3-element row vector of positive real numbers.

- If `GyroscopeBiasNoise` is specified as a row vector, the elements correspond to the noise in the *x*, *y*, and *z* axes of the gyroscope bias, respectively.
- If `GyroscopeBiasNoise` is specified as a scalar, the single element is applied to each axis.

Data Types: `single` | `double`

`AccelerometerNoise` — Multiplicative process noise variance from accelerometer (m/s²)²
`1e-4` (default) | scalar | 3-element row vector

Multiplicative process noise variance from the accelerometer in $(m/s^2)^2$, specified as a scalar or 3-element row vector of positive real finite numbers.

- If `AccelerometerNoise` is specified as a row vector, the elements correspond to the noise in the *x*, *y*, and *z* axes of the accelerometer, respectively.
- If `AccelerometerNoise` is specified as a scalar, the single element is applied to each axis.

Data Types: `single` | `double`

`AccelerometerBiasNoise` — Multiplicative process noise variance from accelerometer bias (m/s²)²
`1e-4` (default) | positive scalar | 3-element row vector

Multiplicative process noise variance from the accelerometer bias in $(m/s^2)^2$, specified as a scalar or 3-element row vector of positive real numbers.

- If `AccelerometerBiasNoise` is specified as a row vector, the elements correspond to the noise in the *x*, *y*, and *z* axes of the accelerometer bias, respectively.
- If `AccelerometerBiasNoise` is specified as a scalar, the single element is applied to each axis.

Data Types: `single` | `double`

`GeomagneticVectorNoise` — Additive process noise for geomagnetic vector (μT²)
`1e-6` (default) | positive scalar | 3-element row vector

Additive process noise for geomagnetic vector in μT^2, specified as a scalar or 3-element row vector of positive real numbers.

- If `GeomagneticVectorNoise` is specified as a row vector, the elements correspond to the noise in the *x*, *y*, and *z* axes of the geomagnetic vector, respectively.
- If `GeomagneticVectorNoise` is specified as a scalar, the single element is applied to each axis.

Data Types: `single` | `double`

`MagnetometerBiasNoise` — Additive process noise for magnetometer bias (μT^2)
`0.1` (default) | positive scalar | 3-element row vector

Additive process noise for magnetometer bias in μT^2, specified as a scalar or 3-element row vector.

- If `MagnetometerBiasNoise` is specified as a row vector, the elements correspond to the noise in the *x*, *y*, and *z* axes of the magnetometer bias, respectively.
- If `MagnetometerBiasNoise` is specified as a scalar, the single element is applied to each axis.

Data Types: `single` | `double`

`State` — State vector of extended Kalman filter
22-element column vector

State vector of the extended Kalman filter. The state values represent:

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

Data Types: `single` | `double`

`StateCovariance` — State error covariance for extended Kalman filter
`eye(22)*1e-6` (default) | 22-by-22 matrix

State error covariance for the extended Kalman filter, specified as a 22-by-22-element matrix, or real numbers.

Data Types: `single` | `double`

Object Functions

correct	Correct states using direct state measurements for insfilterMARG
residual	Residuals and residual covariances from direct state measurements for insfilterMARG
fusegps	Correct states using GPS data for insfilterMARG
residualgps	Residuals and residual covariance from GPS measurements for insfilterMARG
fusemag	Correct states using magnetometer data for insfilterMARG
residualmag	Residuals and residual covariance from magnetometer measurements for insfilterMARG
pose	Current orientation and position estimate for insfilterMARG

predict	Update states using accelerometer and gyroscope data for insfilterMARG
reset	Reset internal states for insfilterMARG
stateinfo	Display state vector information for insfilterMARG
tune	Tune insfilterMARG parameters to reduce estimation error
copy	Create copy of insfitlerMARG

Examples

Estimate Pose of UAV

This example shows how to estimate the pose of an unmanned aerial vehicle (UAV) from logged sensor data and ground truth pose.

Load the logged sensor data and ground truth pose of an UAV.

```
load uavshort.mat
```

Initialize the `insfilterMARG` filter object.

```
f = insfilterMARG;
f.IMUSampleRate = imuFs;
f.ReferenceLocation = refloc;
f.AccelerometerBiasNoise = 2e-4;
f.AccelerometerNoise = 2;
f.GyroscopeBiasNoise = 1e-16;
f.GyroscopeNoise = 1e-5;
f.MagnetometerBiasNoise = 1e-10;
f.GeomagneticVectorNoise = 1e-12;
f.StateCovariance = 1e-9*ones(22);
f.State = initstate;

gpsidx = 1;
N = size(accel,1);
p = zeros(N,3);
q = zeros(N,1,'quaternion');
```

Fuse accelerometer, gyroscope, magnetometer, and GPS data.

```
for ii = 1:size(accel,1)                % Fuse IMU
   f.predict(accel(ii,:), gyro(ii,:));

   if ~mod(ii,fix(imuFs/2))             % Fuse magnetometer at 1/2 the IMU rate
       f.fusemag(mag(ii,:),Rmag);
   end

   if ~mod(ii,imuFs)                    % Fuse GPS once per second
       f.fusegps(lla(gpsidx,:),Rpos,gpsvel(gpsidx,:),Rvel);
       gpsidx = gpsidx + 1;
   end

   [p(ii,:),q(ii)] = pose(f);           %Log estimated pose
end
```

Calculate and display RMS errors.

```
posErr = truePos - p;
qErr = rad2deg(dist(trueOrient,q));
```

```
pRMS = sqrt(mean(posErr.^2));
qRMS = sqrt(mean(qErr.^2));
fprintf('Position RMS Error\n\tX: %.2f, Y: %.2f, Z: %.2f (meters)\n\n',pRMS(1),pRMS(2),pRMS(3));

Position RMS Error
    X: 0.57, Y: 0.53, Z: 0.68 (meters)


fprintf('Quaternion Distance RMS Error\n\t%.2f (degrees)\n\n',qRMS);

Quaternion Distance RMS Error
    0.28 (degrees)
```

Algorithms

Note: The following algorithm only applies to an NED reference frame.

`insfilterMARG` uses a 22-axis extended Kalman filter structure to estimate pose in the NED reference frame. The state is defined as:

$$x = \begin{bmatrix} q_0 \\ q_1 \\ q_2 \\ q_3 \\ position_{\mathrm{N}} \\ position_{\mathrm{E}} \\ position_{\mathrm{D}} \\ v_{\mathrm{N}} \\ v_{\mathrm{E}} \\ v_{\mathrm{D}} \\ \Delta\theta bias_{\mathrm{X}} \\ \Delta\theta bias_{\mathrm{Y}} \\ \Delta\theta bias_{\mathrm{Z}} \\ \Delta v bias_{\mathrm{X}} \\ \Delta v bias_{\mathrm{Y}} \\ \Delta v bias_{\mathrm{Z}} \\ geomagneticFieldVector_{\mathrm{N}} \\ geomagneticFieldVector_{\mathrm{E}} \\ geomagneticFieldVector_{\mathrm{D}} \\ magbias_{\mathrm{X}} \\ magbias_{\mathrm{Y}} \\ magbias_{\mathrm{Z}} \end{bmatrix}$$

where

- q_0, q_1, q_2, q_3 –– Parts of orientation quaternion. The orientation quaternion represents a frame rotation from the platform's current orientation to the local NED coordinate system.
- $position_N$, $position_E$, $position_D$ –– Position of the platform in the local NED coordinate system.
- ν_N, ν_E, ν_D –– Velocity of the platform in the local NED coordinate system.
- $\Delta\theta bias_X$, $\Delta\theta bias_Y$, $\Delta\theta bias_Z$ –– Bias in the integrated gyroscope reading.
- $\Delta\nu bias_X$, $\Delta\nu bias_Y$, $\Delta\nu bias_Z$ –– Bias in the integrated accelerometer reading.
- $geomagneticFieldVector_N$, $geomagneticFieldVector_E$, $geomagneticFieldVector_D$ –– Estimate of the geomagnetic field vector at the reference location.
- $magbias_X$, $magbias_Y$, $magbias_Z$ –– Bias in the magnetometer readings.

Given the conventional formation of the predicted state estimate,

$$x_{k|k-1} = f(\hat{x}_{k-1|k-1}, u_k)$$

u_k is controlled by accelerometer and gyroscope data that has been converted to delta velocity and delta angle through trapezoidal integration. The predicted state estimation is:

$$x_{k\,|\,k-1} =$$

$$\begin{bmatrix}
q_0 - q_1\left(\frac{\Delta\theta_X - \Delta\theta bias_X}{2}\right) - q_2\left(\frac{\Delta\theta_Y - \Delta\theta bias_Y}{2}\right) - q_3\left(\frac{\Delta\theta_Z - \Delta\theta bias_Z}{2}\right) \\
q_1 + q_0\left(\frac{\Delta\theta_X - \Delta\theta bias_X}{2}\right) - q_3\left(\frac{\Delta\theta_Y - \Delta\theta bias_Y}{2}\right) + q_2\left(\frac{\Delta\theta_Z - \Delta\theta bias_Z}{2}\right) \\
q_2 + q_3\left(\frac{\Delta\theta_X - \Delta\theta bias_X}{2}\right) + q_0\left(\frac{\Delta\theta_Y - \Delta\theta bias_Y}{2}\right) - q_1\left(\frac{\Delta\theta_Z - \Delta\theta bias_Z}{2}\right) \\
q_3 - q_2\left(\frac{\Delta\theta_X - \Delta\theta bias_X}{2}\right) + q_1\left(\frac{\Delta\theta_Y - \Delta\theta bias_Y}{2}\right) + q_0\left(\frac{\Delta\theta_Z - \Delta\theta bias_Z}{2}\right) \\
position_N + (\Delta t)(v_N) \\
position_E + (\Delta t)(v_E) \\
position_D + (\Delta t)(v_D) \\
v_N + (\Delta t)(g_N) + (\Delta v_X - \Delta v bias_X)\left(q_0^2 + q_1^2 - q_2^2 - q_3^2\right) - 2(\Delta v_Y - \Delta v bias_Y)(q_0 q_3 - q_1 q_2) + 2(\Delta v_Z - \Delta v bias_Z)(q_0 q_2 + \\
v_E + (\Delta t)(g_E) + (\Delta v_Y - \Delta v bias_Y)\left(q_0^2 - q_1^2 + q_2^2 - q_3^2\right) + 2(\Delta v_X - \Delta v bias_X)(q_0 q_3 + q_1 q_2) - 2(\Delta v_Z - \Delta v bias_Z)(q_0 q_1 - \\
v_D + (\Delta t)(g_D) + (\Delta v_Z - \Delta v bias_Z)\left(q_0^2 - q_1^2 - q_2^2 + q_3^2\right) - 2(\Delta v_X - \Delta v bias_X)(q_0 q_2 - q_1 q_3) + 2(\Delta v_Y - \Delta v bias_Y)(q_0 q_1 + \\
\Delta\theta bias_X \\
\Delta\theta bias_Y
\end{bmatrix}$$

where

- $\Delta\theta_X$, $\Delta\theta_Y$, $\Delta\theta_Z$ –– Integrated gyroscope reading.
- $\Delta\nu_X$, $\Delta\nu_Y$, $\Delta\nu_Z$ –– Integrated accelerometer readings.
- Δt –– IMU sample time.
- g_N, g_E, g_D –– Constant gravity vector in the NED frame.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`insfilterNonholonomic` | `insfilterErrorState` | `insfilterAsync`

Introduced in R2018b

correct

Correct states using direct state measurements for `insfilterMARG`

Syntax

```
correct(FUSE,idx,measurement,measurementCovariance)
```

Description

`correct(FUSE,idx,measurement,measurementCovariance)` corrects the state and state estimation error covariance based on the measurement and measurement covariance. The measurement maps directly to the state specified by the indices `idx`.

Input Arguments

`FUSE` — `insfilterMARG` object

object

`insfilterMARG`, specified as an object.

`idx` — State vector Index of measurement to correct

N-element vector of increasing integers in the range [1,22]

State vector index of measurement to correct, specified as an *N*-element vector of increasing integers in the range [1, 22].

The state values represent:

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

Data Types: `single` | `double`

`measurement` — Direct measurement of state

N-element vector

Direct measurement of state, specified as a *N*-element vector. *N* is the number of elements of the index argument, idx.

Data Types: `single` | `double`

`measurementCovariance` — Covariance of measurement
scalar | *N*-element vector | *N*-by-*N* matrix

Covariance of measurement, specified as a scalar, *N*-element vector, or *N*-by-*N* matrix. *N* is the number of elements of the index argument, idx.

Data Types: `single` | `double`

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`insfilterMARG` | `insfilter`

Introduced in R2018b

copy

Create copy of `insfitlerMARG`

Syntax

```
newFilter = copy(filter)
```

Description

`newFilter = copy(filter)` returns a copy of the `insfilterMARG`, `filter`, with the exactly same property values.

Input Arguments

`filter` — Filter to be copied

`insfitlerMARG`

Filter to be copied, specified as an `insfilterMARG` object.

Output Arguments

`newFilter` — New copied filter

`insfitlerMARG`

New copied filter, returned as an `insfilterMARG` object.

See Also

`insfilterMARG`

Introduced in R2020b

fusegps

Correct states using GPS data for `insfilterMARG`

Syntax

```
[res,resCov] = fusegps(FUSE,position,positionCovariance)
[res,resCov] = fusegps(FUSE,position,positionCovariance,velocity,
velocityCovariance)
```

Description

`[res,resCov] = fusegps(FUSE,position,positionCovariance)` fuses GPS position data to correct the state estimate.

`[res,resCov] = fusegps(FUSE,position,positionCovariance,velocity, velocityCovariance)` fuses GPS position and velocity data to correct the state estimate.

Input Arguments

`FUSE` — `insfilterMARG` object
object

`insfilterMARG`, specified as an object.

`position` — Position of GPS receiver (LLA)
3-element row vector

Position of GPS receiver in geodetic latitude, longitude, and altitude (LLA) specified as a real finite 3-element row vector. Latitude and longitude are in degrees with north and east being positive. Altitude is in meters.

Data Types: `single` | `double`

`positionCovariance` — Position measurement covariance of GPS receiver (m^2)
3-by-3 matrix

Position measurement covariance of GPS receiver in m^2, specified as a 3-by-3 matrix.

Data Types: `single` | `double`

`velocity` — Velocity of GPS receiver in local NED coordinate system (m/s)
3-element row vector

Velocity of the GPS receiver in the local NED coordinate system in m/s, specified as a 3-element row vector.

Data Types: `single` | `double`

`velocityCovariance` — Velocity measurement covariance of GPS receiver (m/s^2)
3-by-3 matrix

Velocity measurement covariance of the GPS receiver in the local NED coordinate system in m/s^2, specified as a 3-by-3 matrix.

Data Types: `single` | `double`

Output Arguments

`res` — Position and velocity residual

1-by-6 vector of real values

Position and velocity residual, returned as a 1-by-6 vector of real values in m and m/s, respectively.

`resCov` — Residual covariance

6-by-6 matrix of real values

Residual covariance, returned as a 6-by-6 matrix of real values.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`insfilter` | `insfilterMARG`

Introduced in R2018b

fusemag

Correct states using magnetometer data for `insfilterMARG`

Syntax

```
[res,resCov] = fusemag(FUSE,magReadings,magReadingsCovariance)
```

Description

`[res,resCov] = fusemag(FUSE,magReadings,magReadingsCovariance)` fuses magnetometer data to correct the state estimate.

Input Arguments

`FUSE` — `insfilterMARG` object
object

`insfilterMARG`, specified as an object.

`magReadings` — Magnetometer readings (µT)
3-element row vector

Magnetometer readings in µT, specified as a 3-element row vector of finite real numbers.

Data Types: `single` | `double`

`magReadingsCovariance` — Magnetometer readings error covariance (μT^2)
scalar | 3-element row vector | 3-by-3 matrix

Magnetometer readings error covariance in μT^2, specified as a scalar, 3-element row vector, or 3-by-3 matrix.

Data Types: `single` | `double`

Output Arguments

`res` — Residual
1-by-3 vector of real values

Residual, returned a 1-by-3 vector of real values in µT.

Data Types: `single` | `double`

`resCov` — Residual covariance
3-by-3 matrix of real values

Residual covariance, returned a 3-by-3 matrix of real values in $(\mu T)^2$.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`insfilterMARG` | `insfilter`

Introduced in R2018b

pose

Current orientation and position estimate for `insfilterMARG`

Syntax

```
[position,orientation,velocity] = pose(FUSE)
[position,orientation,velocity] = pose(FUSE,format)
```

Description

`[position,orientation,velocity] = pose(FUSE)` returns the current estimate of the pose and velocity.

`[position,orientation,velocity] = pose(FUSE,format)` returns the current estimate of the pose with orientation in the specified orientation format.

Input Arguments

`FUSE` — `insfilterMARG` object

object

`insfilterMARG`, specified as an object.

`format` — Output orientation format

`'quaternion'` (default) | `'rotmat'`

Output orientation format, specified as either `'quaternion'` for a `quaternion` or `'rotmat'` for a rotation matrix.

Data Types: `char` | `string`

Output Arguments

`position` — Position estimate expressed in the local coordinate system (m)

3-element row vector

Position estimate expressed in the local coordinate system of the filter in meters, returned as a 3-element row vector.

Data Types: `single` | `double`

`orientation` — Orientation estimate expressed in the local coordinate system

quaternion (default) | 3-by-3 rotation matrix

Orientation estimate expressed in the local coordinate system of the filter, returned as a scalar quaternion or 3-by-3 rotation matrix. The quaternion or rotation matrix represents a frame rotation from the local reference frame of the filter to the body reference frame.

Data Types: `single` | `double` | `quaternion`

`velocity` — Velocity estimate expressed in local coordinate system (m/s)
3-element row vector

Velocity estimate expressed in the local coordinate system of the filter in m/s, returned as a 3-element row vector.

Data Types: `single` | `double`

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also
`insfilterMARG` | `insfilter`

Introduced in R2018b

predict

Update states using accelerometer and gyroscope data for `insfilterMARG`

Syntax

```
predict(FUSE,accelReadings,gyroReadings)
```

Description

`predict(FUSE,accelReadings,gyroReadings)` fuses accelerometer and gyroscope data to update the state estimate.

Input Arguments

`FUSE` — `insfilterMARG` object
object

`insfilterMARG`, specified as an object.

`accelReadings` — Accelerometer readings in local sensor body coordinate system (m/s^2)
3-element row vector

Accelerometer readings in m/s^2, specified as a 3-element row vector.

Data Types: `single` | `double`

`gyroReadings` — Gyroscope readings in local sensor body coordinate system (rad/s)
3-element row vector

Gyroscope readings in rad/s, specified as a 3-element row vector.

Data Types: `single` | `double`

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

`insfilterMARG` | `insfilter`

Introduced in R2018b

reset

Reset internal states for `insfilterMARG`

Syntax

```
reset(FUSE)
```

Description

`reset(FUSE)` resets the State, StateCovariance, and internal integrators to their default values.

Input Arguments

`FUSE` — `insfilterMARG` object

object

`insfilterMARG`, specified as an object.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`insfilterMARG` | `insfilter`

Introduced in R2018b

residual

Residuals and residual covariances from direct state measurements for `insfilterMARG`

Syntax

```
[res, resCov]= residual(FUSE,idx,measurement,measurementCovariance)
```

Description

`[res, resCov]= residual(FUSE,idx,measurement,measurementCovariance)` computes the residual, `res`, and the residual covariance, `resCov`, based on the direct state measurement and measurement covariance. The `measurement` maps directly to the states specified by indices, `idx`.

Input Arguments

`FUSE` — `insfilterMARG` object

object

`insfilterMARG`, specified as an object.

`idx` — State vector index of measurement

N-element vector of increasing integers in the range [1,22]

State vector index of measurement, specified as an *N*-element vector of increasing integers in the range [1, 22].

The state values represent:

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

`measurement` — Direct measurement of state

N-element vector

Direct measurement of state, specified as a *N*-element vector. *N* is the number of elements of the index argument, `idx`.

`measurementCovariance` — Covariance of measurement

N-by-*N* matrix

Covariance of measurement, specified as an *N*-by-*N* matrix. *N* is the number of elements of the index argument, `idx`.

Output Arguments

`res` — Measurement residual
1-by-*N* vector of real values

Measurement residual, returned as a 1-by-*N* vector of real values.

`resCov` — Residual covariance
N-by-*N* matrix of real values

Residual covariance, returned as a *N*-by-*N* matrix of real values.

See Also

`insfilterMARG`

Introduced in R2020a

residualgps

Residuals and residual covariance from GPS measurements for `insfilterMARG`

Syntax

```
[res,resCov] = residualgps(FUSE,position,positionCovariance)
[res,resCov] = residualgps(FUSE,position,positionCovariance,velocity,
velocityCovariance)
```

Description

`[res,resCov] = residualgps(FUSE,position,positionCovariance)` computes the residual, `res`, and the residual covariance, `resCov`, based on the GPS position measurement and covariance.

`[res,resCov] = residualgps(FUSE,position,positionCovariance,velocity, velocityCovariance)` computes the residual, `res`, and the residual covariance, `resCov`, based on the GPS position measurement and covariance.

Input Arguments

`FUSE` — `insfilterMARG` object
object

`insfilterMARG`, specified as an object.

`position` — Position of GPS receiver (LLA)
3-element row vector

Position of GPS receiver in geodetic latitude, longitude, and altitude (LLA) specified as a real finite 3-element row vector. Latitude and longitude are in degrees with north and east being positive. Altitude is in meters.

Data Types: `single` | `double`

`positionCovariance` — Position measurement covariance of GPS receiver (m^2)
3-by-3 matrix

Position measurement covariance of GPS receiver in m^2, specified as a 3-by-3 matrix.

Data Types: `single` | `double`

`velocity` — Velocity of GPS receiver in local NED coordinate system (m/s)
3-element row vector

Velocity of the GPS receiver in the local NED coordinate system in m/s, specified as a 3-element row vector.

Data Types: `single` | `double`

`velocityCovariance` — Velocity measurement covariance of GPS receiver (m/s^2)
3-by-3 matrix

Velocity measurement covariance of the GPS receiver in the local NED coordinate system in m/s^2, specified as a 3-by-3 matrix.

Data Types: `single` | `double`

Output Arguments

`res` — Position and velocity residual

1-by-3 vector of real values | 1-by-6 vector of real values

Position and velocity residual, returned as a 1-by-3 vector of real values if the inputs only contain position information, and returned as 1-by-6 vector of real values if the inputs also contain velocity information.

`resCov` — Residual covariance

3-by-3 matrix of real values | 6-by-6 matrix of real values

Residual covariance, returned as a 3-by-3 matrix of real values if the inputs only contain position information, and a 6-by-6 matrix of real values if the inputs also contain velocity information.

See Also

`insfilter` | `insfilterMARG`

Introduced in R2020a

residualmag

Residuals and residual covariance from magnetometer measurements for `insfilterMARG`

Syntax

```
[res,resCov] = residualmag(FUSE,magReadings,magReadingsCovariance)
```

Description

`[res,resCov] = residualmag(FUSE,magReadings,magReadingsCovariance)` computes the residual, `residual`, and the residual covariance, `resCov`, based on the magnetometer readings and the corresponding covariance.

Input Arguments

`FUSE` — `insfilterMARG` object
object

`insfilterMARG`, specified as an object.

`magReadings` — Magnetometer readings (µT)
3-element row vector

Magnetometer readings in µT, specified as a 3-element row vector of finite real numbers.

Data Types: `single` | `double`

`magReadingsCovariance` — Magnetometer readings error covariance (μT^2)
scalar | 3-element row vector | 3-by-3 matrix

Magnetometer readings error covariance in μT^2, specified as a scalar, 3-element row vector, or 3-by-3 matrix.

Data Types: `single` | `double`

Output Arguments

`res` — Residual
1-by-3 vector of real values

Residual, returned as a 1-by-3 vector of real values in µT.

Data Types: `single` | `double`

`resCov` — Residual covariance
3-by-3 matrix of real values

Residual covariance, returned a 3-by-3 matrix of real values in $(\mu T)^2$.

See Also

`insfilterMARG` | `insfilter`

Introduced in R2020a

stateinfo

Display state vector information for `insfilterMARG`

Syntax

```
stateinfo(FUSE)
info = stateinfo(FUSE)
```

Description

`stateinfo(FUSE)` displays the meaning of each index of the State property and the associated units.

`info = stateinfo(FUSE)` returns a structure with fields containing descriptions of the elements of the state vector of the filter, `FUSE`.

Examples

State Information of `insfilterMARG`

Create an `insfilterMARG` object.

```
filter = insfilterMARG;
```

Display the state information of the created filter.

```
stateinfo(filter)
```

```
States                           Units    Index
Orientation (quaternion parts)            1:4
Position (NAV)                   m        5:7
Velocity (NAV)                   m/s      8:10
Delta Angle Bias (XYZ)           rad      11:13
Delta Velocity Bias (XYZ)        m/s      14:16
Geomagnetic Field Vector (NAV)   µT       17:19
Magnetometer Bias (XYZ)          µT       20:22
```

Output the state information of the filter as a structure.

```
info = stateinfo(filter)
```

```
info = struct with fields:
               Orientation: [1 2 3 4]
                  Position: [5 6 7]
                  Velocity: [8 9 10]
            DeltaAngleBias: [11 12 13]
         DeltaVelocityBias: [14 15 16]
    GeomagneticFieldVector: [17 18 19]
          MagnetometerBias: [20 21 22]
```

Input Arguments

`FUSE` — `insfilterMARG` object

object

`insfilterMARG`, specified as an object.

`info` — State information

structure

State information, returned as a structure with fields containing descriptions of the elements of the state vector of the filter. The values of each field are the corresponding indices of the state vector.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`insfilterMARG` | `insfilter`

Introduced in R2018b

tune

Tune `insfilterMARG` parameters to reduce estimation error

Syntax

```
tunedMeasureNoise = tune(filter,measureNoise,sensorData,groundTruth)
tunedMeasureNoise = tune( ___ ,config)
```

Description

`tunedMeasureNoise = tune(filter,measureNoise,sensorData,groundTruth)` adjusts the properties of the `insfilterMARG` filter object, `filter`, and measurement noises to reduce the root-mean-squared (RMS) state estimation error between the fused sensor data and the ground truth. The function also returns the tuned measurement noise, `tunedMeasureNoise`. The function uses the property values in the filter and the measurement noise provided in the `measureNoise` structure as the initial estimate for the optimization algorithm.

`tunedMeasureNoise = tune( ___ ,config)` specifies the tuning configuration based on a `tunerconfig` object, `config`.

Examples

Tune `insfilterMARG` to Optimize Pose Estimate

Load the recorded sensor data and ground truth data.

```
load('insfilterMARGTuneData.mat');
```

Create tables for the sensor data and the truth data.

```
sensorData = table(Accelerometer, Gyroscope, ...
    Magnetometer, GPSPosition, GPSVelocity);
groundTruth = table(Orientation, Position);
```

Create an `insfilterMARG` filter object that has a few noise properties.

```
filter = insfilterMARG('State',initialState,...
        'StateCovariance',initialStateCovariance,...
        'AccelerometerBiasNoise',1e-7,...
        'GyroscopeBiasNoise',1e-7,...
        'MagnetometerBiasNoise',1e-7,...
        'GeomagneticVectorNoise',1e-7);
```

Create a tuner configuration object for the filter. Set the maximum iterations to eight. Also, set the tunable parameters.

```
cfg = tunerconfig('insfilterMARG', 'MaxIterations', 8);
cfg.TunableParameters = setdiff(cfg.TunableParameters, ...
    {'GeomagneticFieldVector', 'AccelerometerBiasNoise', ...
    'GyroscopeBiasNoise', 'MagnetometerBiasNoise'});
```

Use the tuner noise function to obtain a set of initial sensor noises used in the filter.

```
measNoise = tunernoise('insfilterMARG')
```

```
measNoise = struct with fields:
    MagnetometerNoise: 1
     GPSPositionNoise: 1
     GPSVelocityNoise: 1
```

Tune the filter and obtain the tuned parameters.

```
tunedParams = tune(filter, measNoise, sensorData, ...
        groundTruth, cfg);
```

```
    Iteration    Parameter                Metric
    _________    _________                ______
    1            AccelerometerNoise       2.5701
    1            GPSPositionNoise         2.5446
    1            GPSVelocityNoise         2.5279
    1            GeomagneticVectorNoise   2.5268
    1            GyroscopeNoise           2.5268
    1            MagnetometerNoise        2.5204
    2            AccelerometerNoise       2.5203
    2            GPSPositionNoise         2.4908
    2            GPSVelocityNoise         2.4695
    2            GeomagneticVectorNoise   2.4684
    2            GyroscopeNoise           2.4684
    2            MagnetometerNoise        2.4615
    3            AccelerometerNoise       2.4615
    3            GPSPositionNoise         2.4265
    3            GPSVelocityNoise         2.4000
    3            GeomagneticVectorNoise   2.3988
    3            GyroscopeNoise           2.3988
    3            MagnetometerNoise        2.3911
    4            AccelerometerNoise       2.3911
    4            GPSPositionNoise         2.3500
    4            GPSVelocityNoise         2.3164
    4            GeomagneticVectorNoise   2.3153
    4            GyroscopeNoise           2.3153
    4            MagnetometerNoise        2.3068
    5            AccelerometerNoise       2.3068
    5            GPSPositionNoise         2.2587
    5            GPSVelocityNoise         2.2166
    5            GeomagneticVectorNoise   2.2154
    5            GyroscopeNoise           2.2154
    5            MagnetometerNoise        2.2063
    6            AccelerometerNoise       2.2063
    6            GPSPositionNoise         2.1505
    6            GPSVelocityNoise         2.0981
    6            GeomagneticVectorNoise   2.0971
    6            GyroscopeNoise           2.0971
    6            MagnetometerNoise        2.0875
    7            AccelerometerNoise       2.0874
    7            GPSPositionNoise         2.0240
    7            GPSVelocityNoise         1.9601
    7            GeomagneticVectorNoise   1.9594
    7            GyroscopeNoise           1.9594
    7            MagnetometerNoise        1.9499
```

```
    8            AccelerometerNoise        1.9499
    8            GPSPositionNoise          1.8802
    8            GPSVelocityNoise          1.8035
    8            GeomagneticVectorNoise    1.8032
    8            GyroscopeNoise            1.8032
    8            MagnetometerNoise         1.7959
```

Fuse the sensor data using the tuned filter.

```
N = size(sensorData,1);
qEstTuned = quaternion.zeros(N,1);
posEstTuned = zeros(N,3);
for ii=1:N
    predict(filter,Accelerometer(ii,:),Gyroscope(ii,:));
    if all(~isnan(Magnetometer(ii,1)))
        fusemag(filter,Magnetometer(ii,:),...
            tunedParams.MagnetometerNoise);
    end
    if all(~isnan(GPSPosition(ii,1)))
        fusegps(filter,GPSPosition(ii,:),...
            tunedParams.GPSPositionNoise,GPSVelocity(ii,:),...
            tunedParams.GPSVelocityNoise);
    end
    [posEstTuned(ii,:),qEstTuned(ii,:)] = pose(filter);
end
```

Compute the RMS errors.

```
orientationErrorTuned = rad2deg(dist(qEstTuned,Orientation));
rmsOrientationErrorTuned = sqrt(mean(orientationErrorTuned.^2))
```

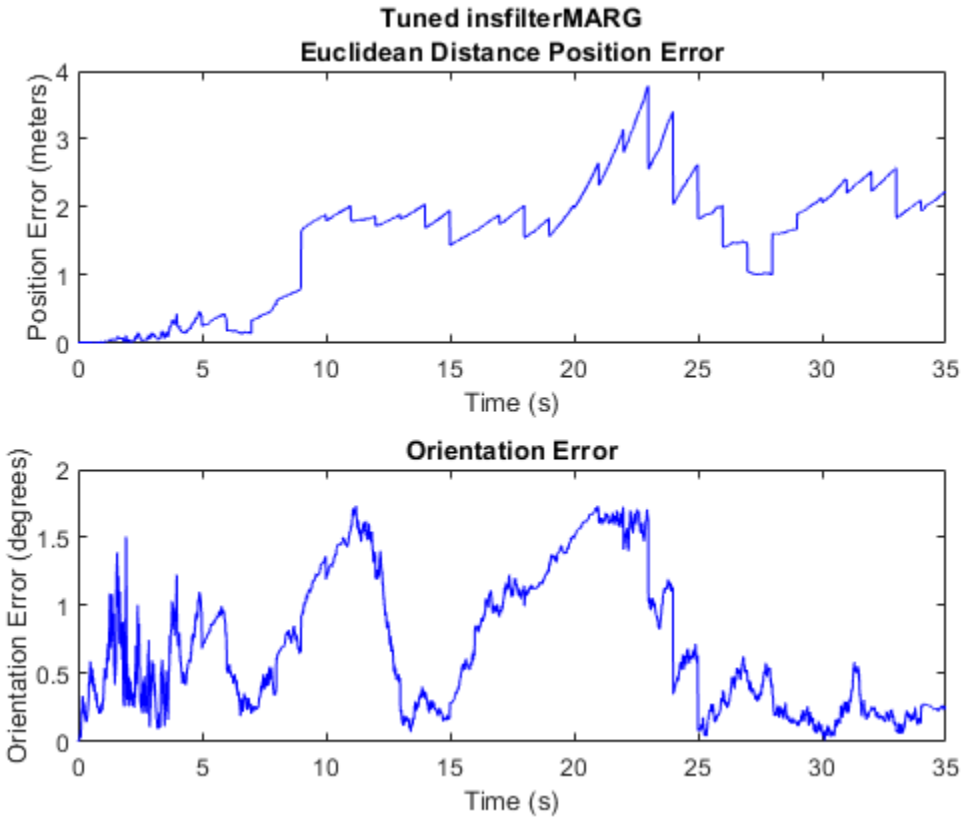
```
rmsOrientationErrorTuned = 0.8580
```

```
positionErrorTuned = sqrt(sum((posEstTuned - Position).^2,2));
rmsPositionErrorTuned = sqrt(mean(positionErrorTuned.^2))
```

```
rmsPositionErrorTuned = 1.7946
```

Visualize the results.

```
figure();
t = (0:N-1)./filter.IMUSampleRate;
subplot(2,1,1)
plot(t,positionErrorTuned,'b');
title("Tuned insfilterMARG" + newline + ...
    "Euclidean Distance Position Error")
xlabel('Time (s)');
ylabel('Position Error (meters)')
subplot(2,1,2)
plot(t, orientationErrorTuned,'b');
title("Orientation Error")
xlabel('Time (s)');
ylabel('Orientation Error (degrees)');
```



Input Arguments

`filter` — Filter object

`infilterMARG` object

Filter object, specified as an `insfilterMARG` object.

`measureNoise` — Measurement noise

structure

Measurement noise, specified as a structure. The function uses the measurement noise input as the initial guess for tuning the measurement noise. The structure must contain these fields:

Field name	Description
`MagnetometerNoise`	Variance of magnetometer noise, specified as a scalar in $(\mu T)^2$
`GPSPositionNoise`	Variance of GPS position noise, specified as a scalar in m^2
`GPSVelocityNoise`	Variance of GPS velocity noise, specified as a scalar in $(m/s)^2$

`sensorData` — Sensor data

`table`

Sensor data, specified as a `table`. In each row, the sensor data is specified as:

- `Accelerometer` — Accelerometer data, specified as a 1-by-3 vector of scalars in m^2/s.
- `Gyroscope` — Gyroscope data, specified as a 1-by-3 vector of scalars in rad/s.
- `Magnetometer` — Magnetometer data, specified as a 1-by-3 vector of scalars in µT.
- `GPSPosition` — GPS position data, specified as a 1-by-3 vector of scalars in [degrees, degrees, meters].
- `GPSVelocity` — GPS velocity data, specified as a 1-by-3 vector of scalars in m/s.

If the GPS sensor does not produce complete measurements, specify the corresponding entry for `GPSPosition` and/or `GPSVelocity` as `NaN`. If you set the `Cost` property of the tuner configuration input, `config`, to `Custom`, then you can use other data types for the `sensorData` input based on your choice.

`groundTruth` — Ground truth data
`table`

Ground truth data, specified as a `table`. In each row, the table can optionally contain any of these variables:

- `Orientation` — Orientation from the navigation frame to the body frame, specified as a `quaternion` or a 3-by-3 rotation matrix.
- `Position` — Position in navigation frame, specified as a 1-by-3 vector of scalars in meters.
- `Velocity` — Velocity in navigation frame, specified as a 1-by-3 vector of scalars in m/s.
- `DeltaAngleBias` — Delta angle bias, specified as a 1-by-3 vector of scalars in radians.
- `DeltaVelocityBias` — Delta velocity bias, specified as a 1-by-3 vector of scalars in m/s.
- `GeomagneticFieldVector` — Geomagnetic field vector in navigation frame, specified as a 1-by-3 vector of scalars.
- `MagnetometerBias` — Magnetometer bias in body frame, specified as a 1-by-3 vector of scalars in µT.

The function processes each row of the `sensorData` and `groundTruth` tables sequentially to calculate the state estimate and RMS error from the ground truth. State variables not present in `groundTruth` input are ignored for the comparison. The `sensorData` and the `groundTruth` tables must have the same number of rows.

If you set the `Cost` property of the tuner configuration input, `config`, to `Custom`, then you can use other data types for the `groundTruth` input based on your choice.

`config` — Tuner configuration
`tunerconfig` object

Tuner configuration, specified as a `tunerconfig` object.

Output Arguments

`tunedMeasureNoise` — Tuned measurement noise
structure

Tuned measurement noise, returned as a structure. The structure contains these fields.

Field name	Description
`MagnetometerNoise`	Variance of magnetometer noise, specified as a scalar in $(\mu T)^2$
`GPSPositionNoise`	Variance of GPS position noise, specified as a scalar in m^2
`GPSVelocityNoise`	Variance of GPS velocity noise, specified as a scalar in $(m/s)^2$

References

[1] Abbeel, P., Coates, A., Montemerlo, M., Ng, A.Y. and Thrun, S. Discriminative Training of Kalman Filters. In *Robotics: Science and systems*, Vol. 2, pp. 1, 2005.

See Also

`tunerconfig` | `tunernoise`

Introduced in R2021a

gnssSensor

Simulate GNSS to generate position and velocity readings

Description

The `gnssSensor` System object simulates a global navigation satellite system (GNSS) to generate position and velocity readings based on local position and velocity data. The object calculates satellite positions and velocities based on the sensor time and data that specifies the satellite orbital parameters on page 2-580. The object uses only the Global Positioning System (GPS) constellations for calculations. To set the starting positions of the satellites, set the InitialTime property.

To generate GNSS position and velocity readings:

1. Create the `gnssSensor` object and set its properties.
2. Call the object with arguments, as if it were a function.

To learn more about how System objects work, see What Are System Objects?

Calling the object increments the time of the sensor and propagates the satellite position and velocities based on the orbital parameters.

Creation

Syntax

```
GNSS = gnssSensor
GNSS = gnssSensor('ReferenceFrame',frame)
GNSS = gnssSensor( ___ ,Name,Value)
```

Description

`GNSS = gnssSensor` returns a `gnssSensor` System object `GNSS` that computes global navigation satellite system receiver readings based on local position and velocity input.

`GNSS = gnssSensor('ReferenceFrame',frame)` specifies the reference frame in which the GNSS readings are reported. Specify `frame` as `'NED'` (north-east-down) or `'ENU'` (east-north-up). The default value is `'NED'`.

`GNSS = gnssSensor( ___ ,Name,Value)` sets properties using one or more name-value pairs. For example, `gnssSensor('SampleRate',2)` creates a simulated GNSS with a sample rate of 2 Hz. Enclose each property name in quotes.

Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see System Design in MATLAB Using System Objects.

`SampleRate` — Sample rate of GNSS receiver (Hz)
`1` (default) | positive scalar

Sample rate of the GNSS receiver, specified as a positive scalar in Hz.

Data Types: `single` | `double`

`InitialTime` — Initial time of GNSS receiver
`datetime('now','TimeZone','UTC')` (default) | `datetime` object

Initial time of the GNSS receiver, specified as a `datetime` object. The object accounts for leap seconds in the conversion between the UTC and the GNSS time.

`ReferenceLocation` — Origin of local navigation reference frame
`[0 0 0]` (default) | three-element row vector of scalars

Origin of the local navigation reference frame, specified as a three-element row vector of scalars in geodetic coordinates (latitude in degrees, longitude in degrees, and altitude in meters). Altitude is the height above the reference ellipsoid model WGS84.

Data Types: `single` | `double`

`MaskAngle` — Elevation mask angle (deg)
`10` (default) | scalar in [0, 90]

Elevation mask angle, specified as a scalar in the range of [0, 90] in degrees. Satellites in view but below the mask angle are not used in estimating the position of the receiver.

Tunable: Yes

Data Types: `single` | `double`

`RangeAccuracy` — Standard deviation of measurement noise of pseudorange (m)
`1` (default) | nonnegative scalar

Standard deviation of the measurement noise of pseudorange, specified as a nonnegative scalar in meters.

Tunable: Yes

Data Types: `single` | `double`

`RangeRateAccuracy` — Standard deviation of measurement noise of pseudorange rate (m/s)
`0.02` (default) | nonnegative scalar

Standard deviation of the measurement noise of pseudorange rate, specified as a nonnegative scalar in meters per second.

Tunable: Yes

Data Types: `single` | `double`

`RandomStream` — Random number source
`'Global stream'` (default) | `'mt19937ar with seed'`

Random number source, specified as one of these options::

- `'Global stream'` –- Random numbers are generated using the current global random number stream.
- `'mt19937ar with seed'` –- Random numbers are generated using the mt19937ar algorithm with the seed specified by the `Seed` property.

Data Types: `char` | `string`

Seed — Initial seed of mt19937ar random number generator algorithm

67 (default) | nonnegative integer

Initial seed of an mt19937ar random number generator algorithm, specified as a nonnegative integer.

Dependencies

To enable this property, set `RandomStream` to `'mt19937ar with seed'`.

Data Types: `integer`

Usage

Syntax

```
[positionReadings,velocityReadings,status] = GNSS(position,velocity)
```

Description

`[positionReadings,velocityReadings,status] = GNSS(position,velocity)` computes global navigation satellite system receiver readings from the position and velocity inputs.

Input Arguments

`position` — Cartesian position of GNSS receiver in local navigation coordinate system (m)

N-by-3 matrix of scalar

Cartesian position of the GNSS receiver in the local navigation coordinate system, specified as an *N*-by-3 matrix of scalars in meters. *N* is the number of samples.

The default reference frame is NED (north-east-down). For ENU (east-north-up), set the `ReferenceFrame` name-value argument to `'ENU'` on creation.

Data Types: `single` | `double`

`velocity` — Velocity of GNSS receiver in local navigation coordinate system (m/s)

N-by-3 matrix of scalars

Velocity of the GNSS receiver in the local navigation coordinate system, specified as an *N*-by-3 matrix in meters per second. *N* is the number of samples.

Data Types: `single` | `double`

The default reference frame is NED (north-east-down). For ENU (east-north-up), set the `ReferenceFrame` name-value argument to `'ENU'` on creation.

Output Arguments

`positionReadings` — Position readings in LLA coordinate system

N-by-3 matrix of scalar

Position readings of the GNSS receiver in the geodetic latitude, longitude, and altitude (LLA) coordinate system, returned as an *N*-by-3 matrix of scalars. Altitude is the height above the reference ellipsoid model, WGS84. *N* is the number of samples in the input argument. Latitude and longitude are in degrees. Altitude is in meters.

Data Types: `single` | `double`

`velocityReadings` — Velocity readings in local navigation coordinate system (m/s)

N-by-3 matrix of scalar

Velocity reading of the GNSS receiver in the local navigation coordinate system in meters per second, returned as an *N*-by-3 matrix of scalars. *N* is the number of samples in the input arguments.

Data Types: `single` | `double`

`status` — Status information of visible satellites

N-element array of structure

Status information of visible satellites, returned as an *N*-element array of structures. *N* is the number of samples in the input arguments. Each structure contains these four fields:

Field Name	Description
`SatelliteAzimuth`	Azimuth angles of visible satellites, returned as an *M*-element vector of scalars in degrees. *M* is the number of visible satellites.
`SatelliteElevation`	Elevation angles of visible satellites, returned as an *M*-element vector of scalars in degrees. *M* is the number of visible satellites.
`HDOP`	Horizontal dilution of precision, returned as a scalar.
`VDOP`	Vertical dilution of precision, returned as a scalar.

To plot the satellite positions, see the `skyplot` function.

Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

Common to All System Objects

clone Create duplicate System object
step Run System object algorithm
release Release resources and allow changes to System object property values and input characteristics
reset Reset internal states of System object

isDone End-of-data status

Examples

Generate GNSS Position and Velocity Readings

Generate target positions and velocities based on a waypoint trajectory.

```
rng(2020) % For repeatable results
fs = 0.1;
tArrival = 50;
tspan = 0:1/fs:tArrival;
% Create a waypoint trajectory.
trajectory = waypointTrajectory([0,0,0;1,1,1]*500,[0,tArrival]);
[positions,~,velocities] = lookupPose(trajectory,tspan)
```

```
positions = 6×3

         0         0         0
  100.0000  100.0000  100.0000
  200.0000  200.0000  200.0000
  300.0000  300.0000  300.0000
  400.0000  400.0000  400.0000
  500.0000  500.0000  500.0000


velocities = 6×3

   10.0000   10.0000   10.0000
   10.0000   10.0000   10.0000
   10.0000   10.0000   10.0000
   10.0000   10.0000   10.0000
   10.0000   10.0000   10.0000
   10.0000   10.0000   10.0000
```

Create a GNSS System object. Use the LLA position for Natick, MA as the local reference frame origin of the trajectory.

```
refLocNatick = [42.2825 -71.343 53.0352];
gnss = gnssSensor('SampleRate',fs, ...
    'ReferenceLocation',refLocNatick);
```

Generate position and velocity readings based on the GNSS object.

```
[llaReadings,velocityReadings,status] = gnss(positions,velocities)
```

```
llaReadings = 6×3

   42.2825  -71.3430   52.0799
   42.2834  -71.3418  -47.4426
   42.2843  -71.3406 -146.6641
   42.2852  -71.3394 -246.2058
   42.2861  -71.3381 -345.0249
   42.2870  -71.3369 -444.7684
```

```
velocityReadings = 6×3

   10.0200    9.9996    9.9955
    9.9973   10.0153    9.9673
   10.0037   10.0046    9.9760
   10.0154    9.9928    9.9956
    9.9842   10.0177    9.9954
   10.0108    9.9955    9.9620


status=6×1 struct array with fields:
    SatelliteAzimuth
    SatelliteElevation
    HDOP
    VDOP
```

More About

Orbital Parameters

The satellite positions and velocities are defined by orbital parameters from IS-GPS-200K Interface Specification, and are given in the Earth-Centered Earth-Fixed (ECEF) coordinates.

Position calculations use equations from Table 30-II in the same IS-GPS-200K Interface Specification.

Velocity calculations use equations 8.21-8.27 in *Principles of GNSS, Inertial, and Multisensor Integrated Navigation Systems*[1].

References

[1] Groves, Paul D. *Principles of GNSS, Inertial, and Multisensor Integrated Navigation Systems*. 2nd ed, Artech House, 2013.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

Objects

`gpsSensor` | `imuSensor`

Functions

`skyplot` | `gnssconstellation` | `lookangles` | `pseudoranges` | `receiverposition`

Introduced in R2020b

mobileRobotPropagator

State propagator for wheeled robotic systems

Description

The `mobileRobotPropagator` object is a state propagator that propagates and validates the state of a mobile robot based on control commands, durations, and target states. The object supports different kinematic models, integrator types, and control policies.

Creation

Syntax

```
mobileProp = mobileRobotPropagator
mobileProp = mobileRobotPropagator(Name,Value)
```

Description

`mobileProp = mobileRobotPropagator` creates a mobile robot propagator for a bicycle kinematic model using a linear-pursuit control policy.

`mobileProp = mobileRobotPropagator(Name,Value)` specifies properties using name-value arguments. For example, `mobileRobotPropagator("ControlStepSize"=0.01)` creates a mobile robot propagator with a control step size of 0.01.

Properties

`StateSpace` — State space for sampling during planning
`stateSpaceSE2` object (default) | object of subclass of `nav.StateSpace` object

State space for sampling during planning, specified as an object of a subclass of `nav.StateSpace` object.

The state space is responsible for representing the configuration space of a system. The object should include all state information related to the propagated system. Systems employing multi-layer cascade controllers can append persistent low-level control information directly to the state vector, whereas the state propagator directly manages top-level control commands.

`Environment` — Environment for validating states
`[]` (default) | `binaryOccupancyMap` object | `occupancyMap` object | `vehicleCostmap` object

Environment for validating states, specified as a `binaryOccupancyMap`, `occupancyMap`, or `vehicleCostmap` object.

The `mobileRobotPropagator` object validates discrete states along the propagated motion. By default, the environment is empty, so the object only rejects states outside the state space bounds.

This property can only be set during construction.

`DistanceEstimator` — Distance metric for estimating propagation cost
`'euclidean'` (default) | `'dubins'` | `'reedsshepp'`

Distance metric for estimating propagation cost, specified as one of these options:

- `'euclidean'` — Standard Euclidean distance.
- `'dubins'` — Distance along a Dubins path that connects the two states. For more information, see `dubinsPathSegment`.
- `'reedsshepp'` — Distance along a Reeds Shepp path that connects the two states. For more information, see `reedsSheppPathSegment`.

This property can only be set during construction.

`GoalDistance` — Threshold of distance for reaching goal states
1 (default) | positive scalar

Threshold of distance for reaching goal states, specified as a positive scalar. When propagating states, a state is considered equal to the goal state when it is closer than this distance threshold.

This property can only be set during construction.

`KinematicModel` — Kinematic model for propagating state
`'bicycle'` (default) | `'ackermann'`

Kinematic model for propagating the state, which determines the state variables, size of the control inputs, and other system parameters that you can specify in the `SystemParameters` property.

Kinematic Model States and Controls

Type	State Vector	Control input
`'bicycle'`	`[x y theta]`	`[v psi]`
`'ackermann'`	`[x y theta psi]`	`[v psiDot]`

This property can only be set during construction and selecting the Ackermann kinematic model requires the Robotics System Toolbox™.

`Integrator` — Integration method when propagating state
`'rungekutta4'` (default) | `'euler'`

Integration method when propagating state. Integration step size can be updated through the `SystemParameters` property.

`'rungekutta4'` provides a more accurate integration result than `'euler'` at the cost of speed.

This property can only be set during construction.

`SystemParameters` — Parameters for kinematic model, integrator, and control policy
structure

Parameters for the kinematic model, integrator, and control policy, specified as a structure with these fields:

- `KinematicModel` — Parameters for the kinematic model type specified in the `KinematicModel` property.

  - `WheelBase` — Size of wheel base in meters
  - `SpeedLimit` — Velocity in the forward and backward directions in meters per second.
  - `SteerRatelimit` — Limits on steering rate in radians per second
- `Integrator` — Parameters for the integrator type specified in the `Integrator` property.
- `ControlPolicy` — Parameters for the control policy specified in the `ControlPolicy` property.

Control Parameters

`ControlPolicy` — Control command generation policy
`'linearpursuit'` (default) | `'arcpursuit'` | `'randomsamples'`

Control command generation policy, specified as one of these options:

- `'linearpursuit'` — Samples a random velocity and calculates a lookahead point along the vector that connects the initial state to the target state.
- `'arcpursuit'` — Samples a random velocity and calculates a lookahead point along an arc that is tangential to the target state and intersects the initial *xy*-position.
- `'randomsamples'` — Draws a finite set of random control samples from the control space and propagates to each. The propagator selects the sample that gets the closest to the goal and then performs a validation.

`ControlLimits` — Limits on control commands for each state
`[-1 1; -pi/4 pi/4]` (default) | *n*-by-2 matrix

Limits on control commands for each state, specified as an *n*-by-2 matrix. *n* is the number of control inputs for your system model.

`NumControlOutput` — Number of control outputs
`2` (default) | positive scalar

This property is read-only.

Number of control outputs, specified as a nonnegative scalar.

`ControlStepSize` — Duration of each control command
`0.1` (default) | positive scalar

Duration of each control command, specified as a positive scalar.

`MaxControlSteps` — Maximum number of control steps
`10` (default) | positive integer

Maximum number of times to propagate the system specified as positive integer.

Object Functions

distance	Estimate cost of propagating to target state
propagate	Propagate system without validation
propagateWhileValid	Propagate system and return valid motion
sampleControl	Generate control command and duration
setup	Set up the mobile robot state propagator

Examples

Plan Kinodynamic Path with Controls for Mobile Robot

Plan control paths for a bicycle kinematic model with the `mobileRobotPropagator` object. Specify a map for the environment, set state bounds, and define a start and goal location. Plan a path using the control-based RRT algorithm, which uses a state propagator for planning motion and the required control commands.

Set State and State Propagator Parameters

Load a ternary map matrix and create an `occupancyMap` object. Create the state propagator using the map. By default, the state propagator uses a bicycle kinematic model.

```
load('exampleMaps','ternaryMap')
map = occupancyMap(ternaryMap,10);

propagator = mobileRobotPropagator(Environment=map); % Bicycle model
```

Set the state bounds on the state space based on the map world limits.

```
propagator.StateSpace.StateBounds(1:2,:) = ...
                    [map.XWorldLimits; map.YWorldLimits];
```

Plan Path

Create the path planner from the state propagator.

```
planner = plannerControlRRT(propagator);
```

Specify the start and goal states.

```
start = [10 15 0];
goal  = [40 30 0];
```

Plan a path between the states. For repeatable results, reset the random number generator before planning. The `plan` function outputs a `navPathControl` object, which contains the states, control commands, and durations.

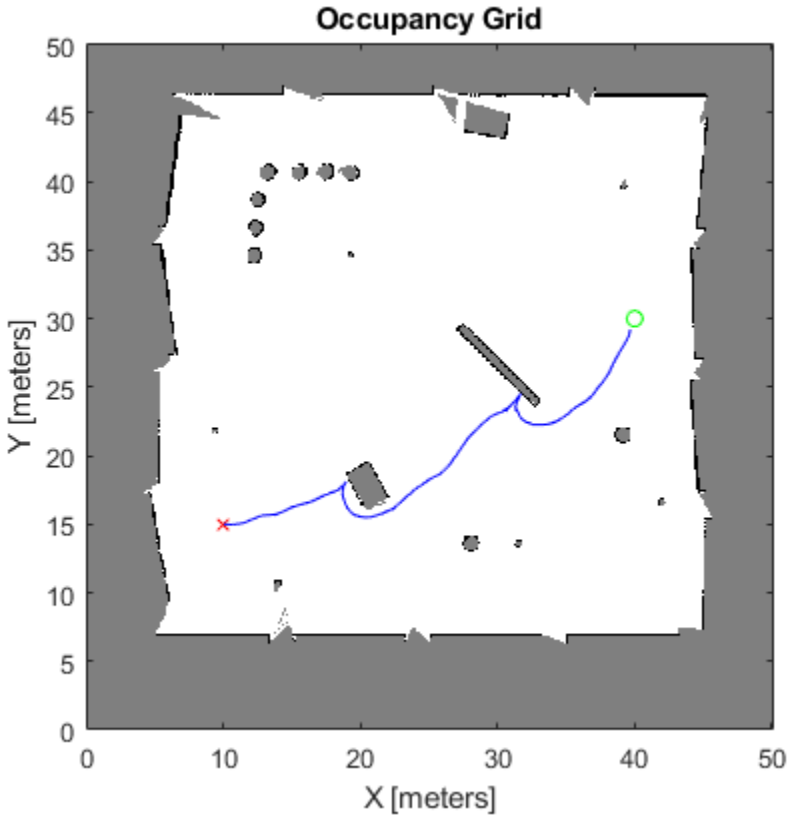
```
rng("default")
path = plan(planner,start,goal)
```

```
path =
  navPathControl with properties:

    StatePropagator: [1x1 mobileRobotPropagator]
             States: [192x3 double]
           Controls: [191x2 double]
          Durations: [191x1 double]
       TargetStates: [191x3 double]
          NumStates: 192
       NumSegments: 191
```

Visualize Results

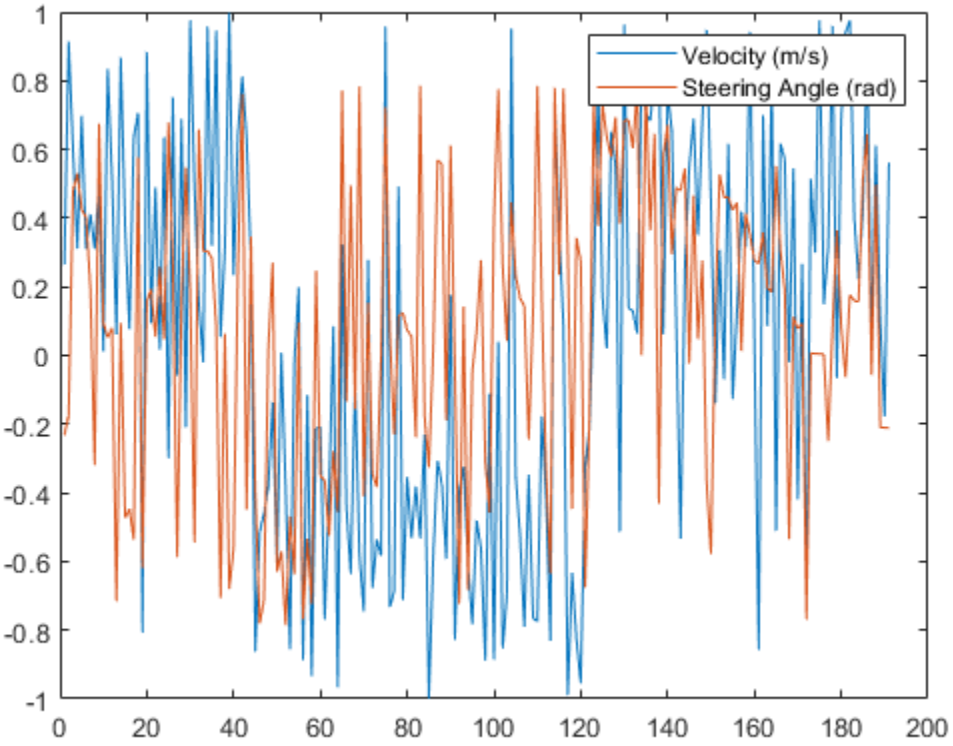
Visualize the map and plot the path states.

```
show(map)
hold on
plot(start(1),start(2),"rx")
plot(goal(1),goal(2),"go")
plot(path.States(:,1),path.States(:,2),"b")
hold off
```



Display the `[v psi]` control inputs of forward velocity and steering angle.

```
plot(path.Controls)
ylim([-1 1])
legend(["Velocity (m/s)","Steering Angle (rad)"])
```



See Also

Classes
`nav.StateSpace`

Objects
`stateSpaceSE2` | `stateSpaceDubins` | `stateSpaceReedsShepp`

Functions
`distance` | `propagate` | `propagateWhileValid` | `sampleControl` | `setup`

Introduced in R2021b

distance

Estimate cost of propagating to target state

Syntax

```
h = distance(mobileProp,q1,q2)
```

Description

`h = distance(mobileProp,q1,q2)` estimates the cost of propagating from one state to another. The `DistanceEstimator` property of the state propagator defines the distance metric for approximating cost.

Input Arguments

`mobileProp` — Mobile robot state propagator

`mobileRobotPropagator` object

Mobile robot state propagator, specified as a `mobileRobotPropagator` object.

`q1` — Initial states

n-by-*s* matrix

Initial states, specified as an *n*-by-*s* matrix. *n* is the number of states and *s* is the size of the state vector.

`q2` — Final states

n-by-*s* matrix

Final states, specified as an *n*-by-*s* matrix. *n* is the number of states and *s* is the size of the state vector.

Output Arguments

`h` — Cost values

n-element vector

Cost values, returned as an *n*-element vector. *n* is the number of `q1` and `q2` pairs.

You can use the cost values returned by this function to find the nearest neighbor for sampled target states when planning a path.

See Also

Objects

`mobileRobotPropagator`

Functions

`propagate` | `propagateWhileValid` | `sampleControl` | `setup`

Introduced in R2021b

propagate

Propagate system without validation

Syntax

```
[q,u,steps] = propagate(mobileProp,q0,u0,qTgt,maxSteps)
```

Description

`[q,u,steps] = propagate(mobileProp,q0,u0,qTgt,maxSteps)` iteratively propagates the system from the current state `q0` towards a target state `qTgt` with an initial control input `u0` for a maximum number of steps `maxSteps`.

At the end of each propagation step `i`, the system returns these values:

- `q(i,:)` — Current state of the system
- `u(i,:)` — Control input for step `i + 1`
- `steps(i)` — Number of steps between `i - 1` and `i`

Input Arguments

`mobileProp` — Mobile robot state propagator

`mobileRobotPropagator` object

Mobile robot state propagator, specified as a `mobileRobotPropagator` object.

`q0` — Initial state

s-element vector

Initial state of the system, specified as an *s*-element vector. *s* is the number of state variables in the state space.

`u0` — Initial control on initial state

c-element vector

Initial control on the initial state, specified as an *c*-element vector. *c* is the number of control inputs.

`qTgt` — Target state

s-element vector

Target state of the system, specified as an *s*-element vector. *s* is the number of state variables in the state space.

`maxSteps` — Maximum number of steps

positive scalar

Maximum number of steps, specified as a positive scalar.

Output Arguments

q — Propagated states
n-by-*s* matrix

Propagated states of the system, returned as an *s*-element vector. *s* is the number of state variables in the state space.

u — Control inputs for propagating states
n-by-*c* matrix

Control inputs for propagating states, returned as an *n*-by-*c* matrix. *c* is the number of control inputs

`steps` — Number of steps between each state and control input to next
n-element vector of positive integers

Number of steps from each state and control input to next, returned as an *n*-element vector of positive integers.

See Also

Objects
`mobileRobotPropagator`

Functions
`distance` | `propagateWhileValid` | `sampleControl` | `setup`

Introduced in R2021b

propagateWhileValid

Propagate system and return valid motion

Syntax

```
[q,u,steps] = propagate(mobileProp,q0,u0,qTgt,maxSteps)
```

Description

`[q,u,steps] = propagate(mobileProp,q0,u0,qTgt,maxSteps)` iteratively propagates the system from the current state `q0` towards a target state `qTgt` with an initial control input `u0` for a maximum number of steps `maxSteps`.

At the end of each propagation step `i`, the system returns these values:

- `q(i,:)` — Current state of the system
- `u(i,:)` — Control input for step `i + 1`
- `steps(i)` — Number of steps between `i - 1` and `i`

The function validates all propagations and returns system information between `q0` and the last valid state.

Input Arguments

`mobileProp` — Mobile robot state propagator
`mobileRobotPropagator` object

Mobile robot state propagator, specified as a `mobileRobotPropagator` object.

`q0` — Initial state
s-element vector

Initial state of the system, specified as an *s*-element vector. *s* is the number of state variables in the state space.

`u0` — Initial control on initial state
c-element vector

Initial control on the initial state, specified as an *c*-element vector. *c* is the number of control inputs.

`qTgt` — Target state
s-element vector

Target state of the system, specified as an *s*-element vector. *s* is the number of state variables in the state space.

`maxSteps` — Maximum number of steps
positive scalar

Maximum number of steps, specified as a positive scalar.

Output Arguments

q — Propagated states
n-by-*s* matrix

Propagated states of the system, returned as an *s*-element vector. *s* is the number of state variables in the state space.

u — Control inputs for propagating states
n-by-*c* matrix

Control inputs for propagating states, returned as an *n*-by-*c* matrix. *c* is the number of control inputs

`steps` — Number of steps from each state and control input to next
n-element vector of positive integers

Number of steps from each state and control input to next, returned as an *n*-element vector of positive integers.

See Also

Objects
`mobileRobotPropagator`

Functions
`distance` | `propagate` | `sampleControl` | `setup`

Introduced in R2021b

sampleControl

Generate control command and duration

Syntax

```
[u,steps] = sampleControl(mobileProp,q0,u0,qTgt)
```

Description

`[u,steps] = sampleControl(mobileProp,q0,u0,qTgt)` generates a series of control commands and number of steps to move from the current state `q0` with control command `u0` toward the target state `qTgt`.

Input Arguments

`mobileProp` — Mobile robot state propagator

`mobileRobotPropagator` object

Mobile robot state propagator, specified as a `mobileRobotPropagator` object.

`q0` — Initial state

s-element vector

Initial state of the system, specified as an *s*-element vector. *s* is the number of state variables in the state space.

`u0` — Initial control on the initial state

c-element vector

Initial control input, specified as an *c*-element vector. *c* is the number of control inputs.

`qTgt` — Target state

s-element vector

Target state of the system, specified as an *s*-element vector. *s* is the number of state variables in the state space.

Output Arguments

`u` — Control inputs for propagating states

n-by-*c* matrix

Control inputs for propagating states, returned as an *c*-element vector. *c* is the number of control inputs.

`steps` — Number of steps from each state and control input to next

n-element vector of positive integers

Number of steps from each state and control input to next, returned as an *n*-element vector of positive integers.

See Also

Objects
`mobileRobotPropagator`

Functions
`distance` | `propagate` | `propagateWhileValid` | `setup`

Introduced in R2021b

setup

Set up the mobile robot state propagator

Syntax

```
setup(mobileProp)
```

Description

`setup(mobileProp)` sets up the `mobileRobotPropagator` object based on the properties of the object. If you change the properties of the object, use this object function before you use the object to sample controls, propagate the system, or calculate distances.

Note Override this function to implement custom functionality to run in the setup.

Input Arguments

`mobileProp` — Mobile robot state propagator

`mobileRobotPropagator` object

Mobile robot state propagator, specified as a `mobileRobotPropagator` object.

See Also

Objects

`mobileRobotPropagator`

Functions

`distance` | `propagate` | `propagateWhileValid` | `sampleControl`

Introduced in R2021b

monteCarloLocalization

Localize robot using range sensor data and map

Description

The `monteCarloLocalization` System object creates a Monte Carlo localization (MCL) object. The MCL algorithm is used to estimate the position and orientation of a vehicle in its environment using a known map of the environment, lidar scan data, and odometry sensor data.

To localize the vehicle, the MCL algorithm uses a particle filter to estimate the vehicle's position. The particles represent the distribution of likely states for the vehicle, where each particle represents a possible vehicle state. The particles converge around a single location as the vehicle moves in the environment and senses different parts of the environment using a range sensor. An odometry sensor measures the vehicle's motion.

A `monteCarloLocalization` object takes the pose and lidar scan data as inputs. The input lidar scan sensor data is given in its own coordinate frame, and the algorithm transforms the data according to the `SensorModel.SensorPose` property that you must specify. The input pose is computed by integrating the odometry sensor data over time. If the change in pose is greater than any of the specified update thresholds, `UpdateThresholds`, then the particles are updated and the algorithm computes a new state estimate from the particle filter. The particles are updated using this process:

1. The 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. These likelihood weights are based on the sensor model you specify in `SensorModel`.
3. Based on the `ResamplingInterval` property, the particles are resampled from the posterior distribution, and the particles of low weight are eliminated. For example, a resampling interval of 2 means that the particles are resampled after every other update.

The outputs of the object are the estimated pose and covariance, and the value of `isUpdated`. This estimated state is the mean and covariance of the highest weighted cluster of particles. The output pose is given in the map's coordinate frame that is specified in the `SensorModel.Map` property. If the change in pose is greater than any of the update thresholds, then the state estimate has been updated and `isUpdated` is `true`. Otherwise, `isUpdated` is `false` and the estimate remains the same. For continuous tracking the best estimate of a robot's state, repeat this process of propagating particles, evaluating their likelihood, and resampling.

To estimate robot pose and covariance using lidar scan data:

1. Create the `monteCarloLocalization` object and set its properties.
2. Call the object with arguments, as if it were a function.

To learn more about how System objects work, see What Are System Objects?

Creation

Syntax

```
mcl = monteCarloLocalization
mcl = monteCarloLocalization(Name,Value)
```

Description

`mcl = monteCarloLocalization` returns an MCL object that estimates the pose of a vehicle using a map, a range sensor, and odometry data. By default, an empty map is assigned, so a valid map assignment is required before using the object.

`mcl = monteCarloLocalization(Name,Value)` creates an MCL object with additional options specified by one or more `Name,Value` pair arguments.

`Name` is a property name and `Value` is the corresponding value. `Name` must appear inside single quotes (`''`). You can specify several name-value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Properties

`InitialPose` — Initial pose of vehicle
`[0 0 0]` (default) | three-element vector

Initial pose of the vehicle used to start localization, specified as a three-element vector, `[x y theta]`, that indicates the position and heading of the vehicle. Initializing the MCL object with an initial pose estimate enables you to use a smaller value for the maximum number of particles and still converge on a location.

`InitialCovariance` — Covariance of initial pose
`diag([1 1 1])` (default) | diagonal matrix | three-element vector | scalar

Covariance of the Gaussian distribution for the initial pose, specified as a diagonal matrix. Three-element vector and scalar inputs are converted to a diagonal matrix. This matrix gives an estimate of the uncertainty of the `InitialPose`.

`GlobalLocalization` — Flag to start global localization
`false` (default) | `true`

Flag indicating whether to perform global localization, specified as `false` or `true`. The default value, `false`, initializes particles using the `InitialPose` and `InitialCovariance` properties. A `true` value initializes uniformly distributed particles in the entire map and ignores the `InitialPose` and `InitialCovariance` properties. Global localization requires a large number of particles to cover the entire workspace. Use global localization only when the initial estimate of vehicle location and orientation is not available.

`ParticleLimits` — Minimum and maximum number of particles
`[500 5000]` (default) | two-element vector

Minimum and maximum number of particles, specified as a two-element vector, `[min max]`.

`SensorModel` — Likelihood field sensor model
`likelihoodFieldSensor` object

Likelihood field sensor model, specified as a `likelihoodFieldSensor` object. The default value uses the default `likelihoodFieldSensorModel` object. After using the object to get output, call `release` on the object to make changes to `SensorModel`. For example:

```
mcl = monteCarloLocalization;
[isUpdated,pose,covariance] = mcl(ranges,angles);
release(mcl)
mcl.SensorModel.NumBeams = 120;
```

`MotionModel` — Odometry motion model for differential drive
`odometryMotionModel` object

Odometry motion model for differential drive, specified as an `odometryMotionModel` object. The default value uses the default `odometryMotionModel` object. After using the object to get output, call `release` on the object to make changes to `MotionModel`. For example:

```
mcl = monteCarloLocalization;
[isUpdated,pose,covariance] = mcl(ranges,angles);
release(mcl)
mcl.MotionModel.Noise = [0.25 0.25 0.4 0.4];
```

`UpdateThresholds` — Minimum change in states required to trigger update
`[0.2 0.2 0.2]` (default) | three-element vector

Minimum change in states required to trigger update, specified as a three-element vector. The localization updates the particles if the minimum change in any of the `[x y theta]` states is met. The pose estimate updates only if the particle filter is updated.

`ResamplingInterval` — Number of filter updates between resampling of particles
1 (default) | positive integer

Number of filter updates between resampling of particles, specified as a positive integer.

`UseLidarScan` — Use `lidarScan` object as scan input
`false` (default) | `true`

Use a `lidarScan` object as scan input, specified as either `false` or `true`.

Usage

Syntax

```
[isUpdated,pose,covariance] = mcl(odomPose,scan)

[isUpdated,pose,covariance] = mcl(odomPose,ranges,angles)
```

Description

`[isUpdated,pose,covariance] = mcl(odomPose,scan)` estimates the pose and covariance of a vehicle using the MCL algorithm. The estimates are based on the pose calculated from the specified vehicle odometry, `odomPose`, and the specified lidar scan sensor data, `scan`. `mcl` is the

`monteCarloLocalization` object. `isUpdated` indicates whether the estimate is updated based on the `UpdateThreshold` property.

To enable this syntax, you must set the `UseLidarScan` property to `true`. For example:

```
mcl = monteCarloLocalization('UseLidarScan',true);
...
[isUpdated,pose,covariance] = mcl(odomPose,scan);
```

`[isUpdated,pose,covariance] = mcl(odomPose,ranges,angles)` specifies the lidar scan data as `ranges` and `angles`.

Input Arguments

`odomPose` — Pose based on odometry

three-element vector

Pose based on odometry, specified as a three-element vector, `[x y theta]`. This pose is calculated by integrating the odometry over time.

`scan` — Lidar scan readings

`lidarScan` object

Lidar scan readings, specified as a `lidarScan` object.

Dependencies

To use this argument, you must set the `UseLidarScan` property to `true`.

```
mcl.UseLidarScan = true;
```

`ranges` — Range values from scan data

vector

Range values from scan data, specified as a vector with elements measured in meters. These range values are distances from a laser scan sensor at the specified `angles`. The `ranges` vector must have the same number of elements as the corresponding `angles` vector.

`angles` — Angle values from scan data

vector

Angle values from scan data, specified as a vector with elements measured in radians. These angle values are the angles at which the specified `ranges` were measured. The `angles` vector must be the same length as the corresponding `ranges` vector.

Output Arguments

`isUpdated` — Flag for pose update

logical

Flag for pose update, returned as a logical. If the change in pose is more than any of the update thresholds, then the output is `true`. Otherwise, it is `false`. A `true` output means that updated pose and covariance are returned. A `false` output means that pose and covariance are not updated and are the same as at the last update.

`pose` — Current pose estimate

three-element vector

Current pose estimate, returned as a three-element vector, `[x y theta]`. The pose is computed as the mean of the highest-weighted cluster of particles.

`covariance` — Covariance estimate for current pose
matrix

Covariance estimate for current pose, returned as a matrix. This matrix gives an estimate of the uncertainty of the current pose. The covariance is computed as the covariance of the highest-weighted cluster of particles.

Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

Specific to monteCarloLocalization

getParticles	Get particles from localization algorithm

Common to All System Objects

step	Run System object algorithm
release	Release resources and allow changes to System object property values and input characteristics
reset	Reset internal states of System object

Examples

Estimate Vehicle Pose from Range Sensor Data

Create a `monteCarloLocalization` object, assign a sensor model, and calculate a pose estimate using the `step` method.

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.

Create a `monteCarloLocalization` object. Set the `UseLidarScan` property to `true`.

```
mcl = monteCarloLocalization;
mcl.UseLidarScan = true;
```

Assign a sensor model with an occupancy grid map to the object.

```
sm = likelihoodFieldSensorModel;
p = zeros(200,200);
sm.Map = occupancyMap(p,20);
mcl.SensorModel = sm;
```

Create sample laser scan data input.

```
ranges = 10*ones(1,300);
ranges(1,130:170) = 1.0;
```

```
angles = linspace(-pi/2,pi/2,300);
odometryPose = [0 0 0];
```

Create a `lidarScan` object by specifying the ranges and angles.

```
scan = lidarScan(ranges,angles);
```

Estimate vehicle pose and covariance.

```
[isUpdated,estimatedPose,covariance] = mcl(odometryPose,scan)
```

```
isUpdated = logical
   1


estimatedPose = 1×3

   -0.0034   -0.0423   -0.0275


covariance = 3×3

    0.9379   -0.0365         0
   -0.0365    0.9656         0
         0         0    0.9870
```

References

[1] Thrun, Sebatian, Wolfram Burgard, and Dieter Fox. *Probabilistic Robotics*. MIT Press, 2005.

[2] Dellaert, F., D. Fox, W. Burgard, and S. Thrun. "Monte Carlo Localization for Mobile Robots." *Proceedings 1999 IEEE International Conference on Robotics and Automation.*

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`lidarScan` | `likelihoodFieldSensorModel` | `odometryMotionModel`

Topics

“Localize TurtleBot Using Monte Carlo Localization”
“Monte Carlo Localization Algorithm”
Class Attributes
Property Attributes

Introduced in R2019b

getParticles

Get particles from localization algorithm

Syntax

```
[particles,weights] = getParticles(mcl)
```

Description

`[particles,weights] = getParticles(mcl)` returns the current particles used by the `monteCarloLocalization` object. `particles` is an *n*-by-3 matrix that contains the location and orientation of each particle. Each row has a corresponding weight value specified in `weights`. The number of rows can change with each iteration of the MCL algorithm. Use this method to extract the particles and analyze them separately from the algorithm.

Examples

Get Particles from Monte Carlo Localization Algorithm

Get particles from the particle filter used in the Monte Carlo Localization object.

Create a map and a Monte Carlo localization object.

```
map = binaryOccupancyMap(10,10,20);
mcl = monteCarloLocalization(map);
```

Create robot data for the range sensor and pose.

```
ranges = 10*ones(1,300);
ranges(1,130:170) = 1.0;
angles = linspace(-pi/2,pi/2,300);
odometryPose = [0 0 0];
```

Initialize particles using `step`.

```
[isUpdated,estimatedPose,covariance] = step(mcl,odometryPose,ranges,angles);
```

Get particles from the updated object.

```
[particles,weights] = getParticles(mcl);
```

Input Arguments

`mcl` — `monteCarloLocalization` object
handle

`monteCarloLocalization` object, specified as an object handle.

Output Arguments

`particles` — Estimation particles

n-by-3 vector

Estimation particles, returned as an *n*-by-3 vector, `[x y theta]`. Each row corresponds to the position and orientation of a single particle. The length can change with each iteration of the algorithm.

`weights` — Weights of particles

n-by-1 vector

Weights of particles, returned as a *n*-by-1 vector. Each row corresponds to the weight of the particle in the matching row of `particles`. These weights are used in the final estimate of the pose of the vehicle. The length can change with each iteration of the algorithm.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`monteCarloLocalization`

Topics

“Monte Carlo Localization Algorithm”

Introduced in R2019b

multiLayerMap

Manage multiple map layers

Description

The `multiLayerMap` object groups and stores multiple map layers as `mapLayer`, `occupancyMap`, or `binaryOccupancyMap` objects.

Once added to this object, map layers can be modified by either using the `multiLayerMap` object functions or by performing actions on individual map layers using their object functions or the layer name as input. Any modification to common properties on the `multiLayerMap` object are reflected across all associated layers.

Creation

Syntax

```
map = multiLayerMap
map = multiLayerMap(maps)
map = multiLayerMap(names,mapData)
map = multiLayerMap(names,width,height)
map = multiLayerMap(names,width,height,cellDims)
map = multiLayerMap(names,rows,cols,'grid')
map = multiLayerMap(names,rows,cols,cellDims,'grid')
map = multiLayerMap(sourceMap)
map = multiLayerMap( ___ ,Name,Value)
```

Description

`map = multiLayerMap` creates an empty map object occupying 10-by-10 meters of space with a resolution of 1 cell per meter.

`map = multiLayerMap(maps)` creates a multilayer map from a cell array of `mapLayer`, `occupancyMap`, or `binaryOccupancyMap` objects. Objects combined into a multilayer map must be defined with the same resolution and cover the same region in space, but can represent different categories of information over the shared region.

`map = multiLayerMap(names,mapData)` creates a multilayer map from the cell array of layer names and associated cell array of map matrices. Matrices must have the same first two dimensions to cover the same shared region. Default resolution is 1 cell per meter.

`map = multiLayerMap(names,width,height)` creates a multilayer map with the cell array of layer names covering the specified width and height as scalars in meters.

`map = multiLayerMap(names,width,height,cellDims)` creates a multilayer map where the size of the data stored in each cell of the map is defined by the array of integers, `cellDims`. For multiple layers, `cellDims` is a cell array of integer arrays.

`map = multiLayerMap(names,rows,cols,'grid')` specifies the map width and height as a grid size specified in the `rows` and `cols` inputs.

`map = multiLayerMap(names,rows,cols,cellDims,'grid')` creates a map with a specified grid size and the size of the data stored in each cell is defined by the array of integers or cell array of integer arrays `cellDims`.

`map = multiLayerMap(sourceMap)` creates a new object using the layers copied from another `multiLayerMap` object.

`map = multiLayerMap( ___ ,Name,Value)` specifies property values using name-value pairs.

For example, `multiLayerMap(__,'LocalOriginInWorld',[15 20])` sets the local origin to a specific world location.

Properties

`DataSize` — Size of the data in each map layer data array

cell array of integer vectors

Size of the data in each map layer data array, specified as a cell array of integer vectors. In each vector, the first two dimensions define the footprint of the map layer, and all subsequent dimensions dictate the size and layout of the data stored in each cell.

If the map stores an *n*-element vector of values in each cell, this property would be `[width height n]`.

If the map stores a 10-by-10 grid with each cell containing a 3-by-3-by-3 matrix array, the data size would be `[10 10 3 3 3]`.

After you create the object, this property is read-only.

Data Types: `cell` | `double`

`DataType` — Data type of the values stored

cell array of string scalars

Data type of the values stored in each layer, specified as a cell array of character vectors.

When you create this object, the specified map layers determine each data type. After you create the object, this property is read-only.

Data Types: `cell` | `char`

`DefaultValue` — Default value for unspecified map locations

`{0}` (default) | cell array of numeric scalars

Default value for unspecified map locations for each layer, specified as a cell array of numeric scalars. This default value is returned for areas outside the map as well.

Data Types: `cell` | `double`

`GridLocationInWorld` — Location of the grid in local coordinates

`[0 0]` (default) | two-element vector | `[xWorld yWorld]`

Location of the bottom-left corner of the grid in world coordinates, specified as a two-element vector, `[xWorld yWorld]`.

You can set this property when you create the object.

Data Types: `double`

`GridOriginInLocal` — Location of the grid in local coordinates
`[0 0]` (default) | two-element vector | `[xLocal yLocal]`

Location of the bottom-left corner of the grid in local coordinates, specified as a two-element vector, `[xLocal yLocal]`.

You can set this property when you create the object.

Data Types: `double`

`GridSize` — Number of rows and columns in grid
two-element integer-valued vector

This property is read-only.

Number of rows and columns in grid, stored as a 1-by-2 real-valued vector representing the number of rows and columns, in that order.

Data Types: `double`

`LayerNames` — Name of each layer
cell array of string scalars

Name of each layer, specified as a cell array of string scalars. The order of these names are associated with the order of other properties that are cell arrays.

You can set this property when you create the object. After you create the object, this property is read-only.

Data Types: `cell` | `string`

`LocalOriginInWorld` — Location of the local frame in world coordinates
`[0 0]` (default) | two-element vector | `[xWorld yWorld]`

Location of the origin of the local frame in world coordinates, specified as a two-element vector, `[xLocal yLocal]`. Use the `move` function to shift the local frame as your vehicle moves.

You can set this property when you create the object.

Data Types: `double`

`NumLayers` — Number of map layers
`1` (default) | positive integer

This property is read-only.

Number of map layers, stored as a positive integer.

Data Types: `double`

`Resolution` — Grid resolution
`1` (default) | scalar

This property is read-only.

Grid resolution, stored as a scalar in cells per meter representing the number and size of grid locations.

You can set this property when you create the object. After you create the object, this property is read-only.

Data Types: `double`

`XLocalLimits` — Minimum and maximum values of *x*-coordinates in local frame
two-element vector

This property is read-only.

Minimum and maximum values of *x*-coordinates in local frame, stored as a two-element horizontal vector of the form `[min max]`. Local frame is defined by `LocalOriginInWorld` property.

Data Types: `double`

`YLocalLimits` — Minimum and maximum values of *y*-coordinates in local frame
two-element vector

This property is read-only.

Minimum and maximum values of *y*-coordinates in local frame, stored as a two-element horizontal vector of the form `[min max]`. Local frame is defined by `LocalOriginInWorld` property.

Data Types: `double`

`XWorldLimits` — Minimum and maximum world range values of *x*-coordinates
two-element vector

This property is read-only.

Minimum and maximum world range values of *x*-coordinates, stored as a 1-by-2 vector representing the minimum and maximum values, in that order.

Data Types: `double`

`YWorldLimits` — Minimum and maximum world range values of *y*-coordinates
two-element vector

This property is read-only.

Minimum and maximum world range values of *y*-coordinates, stored as a 1-by-2 vector representing the minimum and maximum values, in that order.

Data Types: `double`

Object Functions

getLayer	Return individual layers from multilayer map
getMapData	Retrieve data from map layers
grid2local	Convert grid indices to local coordinates
grid2world	Convert grid indices to world coordinates
local2grid	Convert local coordinates to grid indices

local2world	Convert local coordinates to world coordinates
move	Move map in world frame
setMapData	Assign data to map layers
syncWith	Sync map with overlapping map
world2grid	Convert world coordinates to grid indices
world2local	Convert world coordinates to local coordinates

Examples

Create Listeners Using Dependent Map Layers

The `multiLayerMap` object enables you to group multiple map layers and define behavior for those layers when setting and getting data. Using separate map layers, you can store various map data and specify different behaviors for each. You can also define the `SetTransformFcn` and `GetTransformFcn` function handles for a map layer so that dependencies are created between layers. This example shows how to store data in a map layer and implement event listeners which update other maps. These maps store how many times the data is updated or accessed.

Dependent Layers

Create two independent map layers.

```
mapAccessed = mapLayer(zeros(10,10),'LayerName','GetListener');
mapModified = mapLayer(zeros(10,10),'LayerName','SetListener');
```

Specify function handles for the get and set transform functions used in the main map layer. These functions increment the value of a grid location when you get or set map data in the input map `mainMap`. See Listener Function Handles on page 2-0 for the function implementation.

```
setHookFcn = @(mainMap,values,varargin)exampleHelperSetHookFcn(mapModified,mainMap,values,varargi
getHookFcn = @(mainMap,values,varargin)exampleHelperGetHookFcn(mapAccessed,mainMap,values,varargi
```

Create the main map layer with default values of 0.5. Specify the function handles to create the layer depencies.

```
map = mapLayer(repmat(0.5,10,10),...
                'GetTransformFcn',getHookFcn,...
                'SetTransformFcn',setHookFcn);
```

Add all maps to the same `multiLayerMap` object.

```
mapLayers = multiLayerMap({map,mapAccessed,mapModified});
```

Set the (0,0) map location with a value of zero using the `setMapData` object function of `map`.

```
setMapData(map,[0 0],0)
```

Check that `mapModified` incremented their value.

```
getMapData(mapModified,[0 0])
```

```
ans = 1
```

Get the data you just set to the main map layer. The expected value of zero is returned.

```
getMapData(map,[0 0])
```

```
ans = 0
```

Check that `mapAccessed` incremented their value.

```
getMapData(mapAccessed,[0 0])
```

```
ans = 1
```

Update the entire map with a matrix of values. Access the data as well.

```
setMapData(map,rand(10,10))
getMapData(map)
```

```
ans = 10×10

    0.8147    0.1576    0.6557    0.7060    0.4387    0.2760    0.7513    0.8407    0.3517    0.0
    0.9058    0.9706    0.0357    0.0318    0.3816    0.6797    0.2551    0.2543    0.8308    0.0
    0.1270    0.9572    0.8491    0.2769    0.7655    0.6551    0.5060    0.8143    0.5853    0.5
    0.9134    0.4854    0.9340    0.0462    0.7952    0.1626    0.6991    0.2435    0.5497    0.7
    0.6324    0.8003    0.6787    0.0971    0.1869    0.1190    0.8909    0.9293    0.9172    0.9
    0.0975    0.1419    0.7577    0.8235    0.4898    0.4984    0.9593    0.3500    0.2858    0.1
    0.2785    0.4218    0.7431    0.6948    0.4456    0.9597    0.5472    0.1966    0.7572    0.5
    0.5469    0.9157    0.3922    0.3171    0.6463    0.3404    0.1386    0.2511    0.7537    0.4
    0.9575    0.7922    0.6555    0.9502    0.7094    0.5853    0.1493    0.6160    0.3804    0.0
    0.9649    0.9595    0.1712    0.0344    0.7547    0.2238    0.2575    0.4733    0.5678    0.3
```

Check that `mapModified` and `mapAccessed` incremented their values.

```
getMapData(mapModified)
```

```
ans = 10×10

     1     1     1     1     1     1     1     1     1     1
     1     1     1     1     1     1     1     1     1     1
     1     1     1     1     1     1     1     1     1     1
     1     1     1     1     1     1     1     1     1     1
     1     1     1     1     1     1     1     1     1     1
     1     1     1     1     1     1     1     1     1     1
     1     1     1     1     1     1     1     1     1     1
     1     1     1     1     1     1     1     1     1     1
     1     1     1     1     1     1     1     1     1     1
     2     1     1     1     1     1     1     1     1     1
```

```
getMapData(mapAccessed)
```

```
ans = 10×10

     1     1     1     1     1     1     1     1     1     1
     1     1     1     1     1     1     1     1     1     1
     1     1     1     1     1     1     1     1     1     1
     1     1     1     1     1     1     1     1     1     1
     1     1     1     1     1     1     1     1     1     1
     1     1     1     1     1     1     1     1     1     1
     1     1     1     1     1     1     1     1     1     1
     1     1     1     1     1     1     1     1     1     1
     1     1     1     1     1     1     1     1     1     1
     2     1     1     1     1     1     1     1     1     1
```

The bottom-left location returns two and all other values are one. This confirms the listener functions are working as intended.

Listener Function Handles

These functions implement the get and set example helper functions that update the other map layers.

```
function valuesOut = exampleHelperSetHookFcn(modifiedMap,sourceLayer,valueIn,varargin)
    % Pass output through
    valuesOut = valueIn;

    % If no additional inputs are passed, return immediately.
    if numel(varargin) == 0
        return;
    else
        % Otherwise, increment the value in the modifiedMap.
        if numel(varargin) == 1
            currentValue = getMapData(modifiedMap);
            setMapData(modifiedMap,currentValue+1);
        else
            currentValue = getMapData(modifiedMap,varargin{1},varargin{3:end});
            % setMapData syntax <<<<>>>>
            setMapData(modifiedMap,varargin{1},currentValue+1,varargin{3:end});
        end
    end
end

function data = exampleHelperGetHookFcn(accessedMap,sourceLayer,valuesIn,varargin)

    data = valuesIn;

    % If no additional inputs are passed, check if the values in
    if numel(varargin) == 0
        if isequal(size(valuesIn),sourceLayer.DataSize)
            % Increment the depedent map.
            currentValue = getMapData(accessedMap);
            setMapData(accessedMap,currentValue+1);
        end
    else
        currentValue = getMapData(accessedMap,varargin{:});
        setMapData(accessedMap,varargin{1},currentValue+1,varargin{3:end});
    end
end
```

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

Objects

`mapLayer` | `occupancyMap3D` | `occupancyMap` | `binaryOccupancyMap`

Functions
`getMapData` | `setMapData` | `move` | `syncWith`

Topics
“Fuse Multiple Lidar Sensors Using Map Layers”

Introduced in R2021a

getLayer

Return individual layers from multilayer map

Syntax

```
mapLayer = getLayer(map,layerName)
```

Description

`mapLayer = getLayer(map,layerName)` returns the individual map layer, specified by the layer name `layerName`. For a list of all layer names, see the LayerNames property of the `multiLayerMap` object `map`.

.

Input Arguments

`map` — Multilayer map
`multiLayerMap` object

Multilayer map, specified as a `multiLayerMap` object.

`layerName` — Name of individual map layer
string scalar | character vector

Name of individual map layer, specified as a string scalar or character vector.

Data Types: `char` | `string`

Output Arguments

`mapLayer` — Individual map layer
`binaryOccupancyMap` object | `occupancyMap` object | `mapLayer` object

Individual map layer, returned as a `binaryOccupancyMap`, `occupancyMap`, or `mapLayer` object as a handle. For more information, see “Handle Object Behavior”.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

Objects
`mapLayer` | `occupancyMap3D` | `occupancyMap` | `binaryOccupancyMap`

Functions
`getMapData` | `setMapData` | `move` | `syncWith`

Topics
“Fuse Multiple Lidar Sensors Using Map Layers”

Introduced in R2021a

getMapData

Retrieve data from map layers

Syntax

```
mapData = getMapData(map)
mapData = getMapData(map,layername)
mapData = getMapData(map,layername,layerinputs)
```

Description

`mapData = getMapData(map)` returns a cell array of matrices for the data in each layer of the specified `multiLayerMap` object. For binary or occupancy map layers, the values of this function are passed to the `getOccupancy` function. `mapData` is returned as an cell-array of matrices for each layer.

`mapData = getMapData(map,layername)` returns all the map data for the specified layer name. `mapData` is returned as a matrix equal to the DataSize of the specified layer.

`mapData = getMapData(map,layername,layerinputs)` takes the `layerinputs` arguments and passes them to the `getMapData` object function for the specified map layer name. To access individual cells or blocks of data in the world, local, or grid coordinates, see the syntaxes of `getMapData`.

Examples

Create Listeners Using Dependent Map Layers

The `multiLayerMap` object enables you to group multiple map layers and define behavior for those layers when setting and getting data. Using separate map layers, you can store various map data and specify different behaviors for each. You can also define the `SetTransformFcn` and `GetTransformFcn` function handles for a map layer so that dependencies are created between layers. This example shows how to store data in a map layer and implement event listeners which update other maps. These maps store how many times the data is updated or accessed.

Dependent Layers

Create two independent map layers.

```
mapAccessed = mapLayer(zeros(10,10),'LayerName','GetListener');
mapModified = mapLayer(zeros(10,10),'LayerName','SetListener');
```

Specify function handles for the get and set transform functions used in the main map layer. These functions increment the value of a grid location when you get or set map data in the input map `mainMap`. See Listener Function Handles on page 2-0 for the function implementation.

```
setHookFcn = @(mainMap,values,varargin)exampleHelperSetHookFcn(mapModified,mainMap,values,varargi
getHookFcn = @(mainMap,values,varargin)exampleHelperGetHookFcn(mapAccessed,mainMap,values,varargi
```

Create the main map layer with default values of 0.5. Specify the function handles to create the layer depencies.

```
map = mapLayer(repmat(0.5,10,10),...
                'GetTransformFcn',getHookFcn,...
                'SetTransformFcn',setHookFcn);
```

Add all maps to the same `multiLayerMap` object.

```
mapLayers = multiLayerMap({map,mapAccessed,mapModified});
```

Set the (0,0) map location with a value of zero using the `setMapData` object function of `map`.

```
setMapData(map,[0 0],0)
```

Check that `mapModified` incremented their value.

```
getMapData(mapModified,[0 0])
```

```
ans = 1
```

Get the data you just set to the main map layer. The expected value of zero is returned.

```
getMapData(map,[0 0])
```

```
ans = 0
```

Check that `mapAccessed` incremented their value.

```
getMapData(mapAccessed,[0 0])
```

```
ans = 1
```

Update the entire map with a matrix of values. Access the data as well.

```
setMapData(map,rand(10,10))
getMapData(map)
```

```
ans = 10×10

    0.8147    0.1576    0.6557    0.7060    0.4387    0.2760    0.7513    0.8407    0.3517    0.0
    0.9058    0.9706    0.0357    0.0318    0.3816    0.6797    0.2551    0.2543    0.8308    0.0
    0.1270    0.9572    0.8491    0.2769    0.7655    0.6551    0.5060    0.8143    0.5853    0.5
    0.9134    0.4854    0.9340    0.0462    0.7952    0.1626    0.6991    0.2435    0.5497    0.7
    0.6324    0.8003    0.6787    0.0971    0.1869    0.1190    0.8909    0.9293    0.9172    0.9
    0.0975    0.1419    0.7577    0.8235    0.4898    0.4984    0.9593    0.3500    0.2858    0.1
    0.2785    0.4218    0.7431    0.6948    0.4456    0.9597    0.5472    0.1966    0.7572    0.5
    0.5469    0.9157    0.3922    0.3171    0.6463    0.3404    0.1386    0.2511    0.7537    0.4
    0.9575    0.7922    0.6555    0.9502    0.7094    0.5853    0.1493    0.6160    0.3804    0.0
    0.9649    0.9595    0.1712    0.0344    0.7547    0.2238    0.2575    0.4733    0.5678    0.3
```

Check that `mapModified` and `mapAccessed` incremented their values.

```
getMapData(mapModified)
```

```
ans = 10×10

     1     1     1     1     1     1     1     1     1     1
     1     1     1     1     1     1     1     1     1     1
```

```
     1     1     1     1     1     1     1     1     1     1
     1     1     1     1     1     1     1     1     1     1
     1     1     1     1     1     1     1     1     1     1
     1     1     1     1     1     1     1     1     1     1
     1     1     1     1     1     1     1     1     1     1
     1     1     1     1     1     1     1     1     1     1
     1     1     1     1     1     1     1     1     1     1
     2     1     1     1     1     1     1     1     1     1
```

```
getMapData(mapAccessed)
```

```
ans = 10×10
```

```
     1     1     1     1     1     1     1     1     1     1
     1     1     1     1     1     1     1     1     1     1
     1     1     1     1     1     1     1     1     1     1
     1     1     1     1     1     1     1     1     1     1
     1     1     1     1     1     1     1     1     1     1
     1     1     1     1     1     1     1     1     1     1
     1     1     1     1     1     1     1     1     1     1
     1     1     1     1     1     1     1     1     1     1
     1     1     1     1     1     1     1     1     1     1
     2     1     1     1     1     1     1     1     1     1
```

The bottom-left location returns two and all other values are one. This confirms the listener functions are working as intended.

Listener Function Handles

These functions implement the get and set example helper functions that update the other map layers.

```
function valuesOut = exampleHelperSetHookFcn(modifiedMap,sourceLayer,valueIn,varargin)
    % Pass output through
    valuesOut = valueIn;

    % If no additional inputs are passed, return immediately.
    if numel(varargin) == 0
        return;
    else
        % Otherwise, increment the value in the modifiedMap.
        if numel(varargin) == 1
            currentValue = getMapData(modifiedMap);
            setMapData(modifiedMap,currentValue+1);
        else
            currentValue = getMapData(modifiedMap,varargin{1},varargin{3:end});
            % setMapData syntax <<<<>>>>
            setMapData(modifiedMap,varargin{1},currentValue+1,varargin{3:end});
        end
    end
end

function data = exampleHelperGetHookFcn(accessedMap,sourceLayer,valuesIn,varargin)

    data = valuesIn;
```

```
    % If no additional inputs are passed, check if the values in
    if numel(varargin) == 0
        if isequal(size(valuesIn),sourceLayer.DataSize)
            % Increment the depedent map.
            currentValue = getMapData(accessedMap);
            setMapData(accessedMap,currentValue+1);
        end
    else
        currentValue = getMapData(accessedMap,varargin{:});
        setMapData(accessedMap,varargin{1},currentValue+1,varargin{3:end});
    end
end
```

Input Arguments

`map` — Multilayer map

`multiLayerMap` object

Multilayer map, specified as a `multiLayerMap` object.

`layername` — Map layer name

string scalar | character array

Map layer name, specified as a string scalar or character array. Map layers have their name specified when creating the `multiLayerMap` object.

`layerinputs` — Variable-length inputs to map layer

`varargin`

Variable-length inputs to `getMapData` function of map layer, specified as `varargin`. To specify individual cells or blocks of data in the world, local, or grid coordinates, see the syntaxes of `getMapData`.

Output Arguments

`mapData` — Data values from map layer

matrix

Data values from map layer, returned as a matrix. By default, the function returns all data on the layer as an *M*-by-*N*-by-*DataDims* matrix. *M* and *N* are the grid height and width respectively. *DataDims* are the dimensions of the map data, `map.DataSize(3,:)`.

For other syntaxes, the map data may be given as an array of values with size *N*-by-*DataDims* or as a subregion of the full matrix.

See Also

Objects

`multiLayerMap` | `mapLayer` | `occupancyMap3D` | `occupancyMap` | `binaryOccupancyMap`

Functions

`getMapData` | `setMapData` | `move` | `syncWith`

Introduced in R2021a

setMapData

Assign data to map layers

Syntax

```
setMapData(map,layername,layerinputs)
inBounds = setMapData(map,layername,layerinputs)
```

Description

`setMapData(map,layername,layerinputs)` takes the `layerinputs` arguments and passes them to the `setMapData` object function for the specified map layer name. To specify individual cells or blocks of data in the world, local, or grid coordinates, see the syntaxes of `setMapData`.

`inBounds = setMapData(map,layername,layerinputs)` returns an array of values for the given locations in the `layerinputs` input argument.

Examples

Create Listeners Using Dependent Map Layers

The `multiLayerMap` object enables you to group multiple map layers and define behavior for those layers when setting and getting data. Using separate map layers, you can store various map data and specify different behaviors for each. You can also define the `SetTransformFcn` and `GetTransformFcn` function handles for a map layer so that dependencies are created between layers. This example shows how to store data in a map layer and implement event listeners which update other maps. These maps store how many times the data is updated or accessed.

Dependent Layers

Create two independent map layers.

```
mapAccessed = mapLayer(zeros(10,10),'LayerName','GetListener');
mapModified = mapLayer(zeros(10,10),'LayerName','SetListener');
```

Specify function handles for the get and set transform functions used in the main map layer. These functions increment the value of a grid location when you get or set map data in the input map `mainMap`. See Listener Function Handles on page 2-0 for the function implementation.

```
setHookFcn = @(mainMap,values,varargin)exampleHelperSetHookFcn(mapModified,mainMap,values,varargi
getHookFcn = @(mainMap,values,varargin)exampleHelperGetHookFcn(mapAccessed,mainMap,values,varargi
```

Create the main map layer with default values of 0.5. Specify the function handles to create the layer depencies.

```
map = mapLayer(repmat(0.5,10,10),...
                'GetTransformFcn',getHookFcn,...
                'SetTransformFcn',setHookFcn);
```

Add all maps to the same `multiLayerMap` object.

```
mapLayers = multiLayerMap({map,mapAccessed,mapModified});
```

Set the (0,0) map location with a value of zero using the `setMapData` object function of `map`.

```
setMapData(map,[0 0],0)
```

Check that `mapModified` incremented their value.

```
getMapData(mapModified,[0 0])
```

```
ans = 1
```

Get the data you just set to the main map layer. The expected value of zero is returned.

```
getMapData(map,[0 0])
```

```
ans = 0
```

Check that `mapAccessed` incremented their value.

```
getMapData(mapAccessed,[0 0])
```

```
ans = 1
```

Update the entire map with a matrix of values. Access the data as well.

```
setMapData(map,rand(10,10))
getMapData(map)
```

```
ans = 10×10

    0.8147    0.1576    0.6557    0.7060    0.4387    0.2760    0.7513    0.8407    0.3517    0.0
    0.9058    0.9706    0.0357    0.0318    0.3816    0.6797    0.2551    0.2543    0.8308    0.0
    0.1270    0.9572    0.8491    0.2769    0.7655    0.6551    0.5060    0.8143    0.5853    0.5
    0.9134    0.4854    0.9340    0.0462    0.7952    0.1626    0.6991    0.2435    0.5497    0.7
    0.6324    0.8003    0.6787    0.0971    0.1869    0.1190    0.8909    0.9293    0.9172    0.9
    0.0975    0.1419    0.7577    0.8235    0.4898    0.4984    0.9593    0.3500    0.2858    0.1
    0.2785    0.4218    0.7431    0.6948    0.4456    0.9597    0.5472    0.1966    0.7572    0.5
    0.5469    0.9157    0.3922    0.3171    0.6463    0.3404    0.1386    0.2511    0.7537    0.4
    0.9575    0.7922    0.6555    0.9502    0.7094    0.5853    0.1493    0.6160    0.3804    0.0
    0.9649    0.9595    0.1712    0.0344    0.7547    0.2238    0.2575    0.4733    0.5678    0.3
```

Check that `mapModified` and `mapAccessed` incremented their values.

```
getMapData(mapModified)
```

```
ans = 10×10

     1     1     1     1     1     1     1     1     1     1
     1     1     1     1     1     1     1     1     1     1
     1     1     1     1     1     1     1     1     1     1
     1     1     1     1     1     1     1     1     1     1
     1     1     1     1     1     1     1     1     1     1
     1     1     1     1     1     1     1     1     1     1
     1     1     1     1     1     1     1     1     1     1
     1     1     1     1     1     1     1     1     1     1
     1     1     1     1     1     1     1     1     1     1
     2     1     1     1     1     1     1     1     1     1
```

```
getMapData(mapAccessed)
```

```
ans = 10×10

     1     1     1     1     1     1     1     1     1     1
     1     1     1     1     1     1     1     1     1     1
     1     1     1     1     1     1     1     1     1     1
     1     1     1     1     1     1     1     1     1     1
     1     1     1     1     1     1     1     1     1     1
     1     1     1     1     1     1     1     1     1     1
     1     1     1     1     1     1     1     1     1     1
     1     1     1     1     1     1     1     1     1     1
     1     1     1     1     1     1     1     1     1     1
     2     1     1     1     1     1     1     1     1     1
```

The bottom-left location returns two and all other values are one. This confirms the listener functions are working as intended.

Listener Function Handles

These functions implement the get and set example helper functions that update the other map layers.

```
function valuesOut = exampleHelperSetHookFcn(modifiedMap,sourceLayer,valueIn,varargin)
    % Pass output through
    valuesOut = valueIn;

    % If no additional inputs are passed, return immediately.
    if numel(varargin) == 0
        return;
    else
        % Otherwise, increment the value in the modifiedMap.
        if numel(varargin) == 1
            currentValue = getMapData(modifiedMap);
            setMapData(modifiedMap,currentValue+1);
        else
            currentValue = getMapData(modifiedMap,varargin{1},varargin{3:end});
            % setMapData syntax <<<<>>>>
            setMapData(modifiedMap,varargin{1},currentValue+1,varargin{3:end});
        end
    end
end

function data = exampleHelperGetHookFcn(accessedMap,sourceLayer,valuesIn,varargin)

    data = valuesIn;

    % If no additional inputs are passed, check if the values in
    if numel(varargin) == 0
        if isequal(size(valuesIn),sourceLayer.DataSize)
            % Increment the depedent map.
            currentValue = getMapData(accessedMap);
            setMapData(accessedMap,currentValue+1);
        end
    else
        currentValue = getMapData(accessedMap,varargin{:});
        setMapData(accessedMap,varargin{1},currentValue+1,varargin{3:end});
```

```
    end
end
```

Input Arguments

`map` — Multilayer map

`multiLayerMap` object

Multilayer map, specified as a `multiLayerMap` object.

`layername` — Map layer name

string scalar | character array

Map layer name, specified as a string scalar or character array. Map layers have their name specified when creating the `multiLayerMap` object.

`layerinputs` — Variable-length inputs to map layer

`varargin`

Variable-length inputs to `setMapData` function of the map layer, specified as `varargin`. To specify individual cells or blocks of data in the world, local, or grid coordinates, see the syntaxes of `setMapData`.

Output Arguments

`inBounds` — Valid map locations

n-by-1 column vector

Valid map locations, returned as an *n*-by-1 column vector equal in length to `xy` or `ij`. Locations inside the map limits return a value of 1. Locations outside the map limits return a value of 0.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

Objects

`multiLayerMap` | `mapLayer` | `occupancyMap3D` | `occupancyMap` | `binaryOccupancyMap`

Functions

`getMapData` | `setMapData` | `move` | `syncWith`

Introduced in R2021a

gpsdev

Connect to a GPS receiver connected to host computer

Description

The `gpsdev` System object connects to a GPS receiver connected to the host computer.

To connect to a GPS receiver:

1. Create the `gpsdev` object and set its properties.
2. Call the object with arguments, as if it were a function.

To learn more about how System objects work, see What Are System Objects?

Creation

Syntax

```
gps = gpsdev(port)
gps = gpsdev(serialobj)
gps = gpsdev( ___ ,Name,Value)
```

Description

`gps = gpsdev(port)` connects to a GPS Receiver at the specified serial port of host computer.

`gps = gpsdev(serialobj)` connects to a GPS Receiver specified by a serial object.

`gps = gpsdev( ___ ,Name,Value)` connects to a GPS Receiver on the specified port or specified through a serial object, using one or more name-value pairs.

Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see System Design in MATLAB Using System Objects.

`BaudRate` — Baud rate

`9600`

This property is read-only.

The baud rate for serial communication. The baud rate is set at 9600 bits/sec. The GPS receiver must be configured to work at 9600 bits/sec . If your GPS receiver is configured to some other baud rate, reconfigure it to 9600 bits/sec to use `gpsdev` function.

`ReadMode` — Specify which data samples to be returned
`'latest'` (default) | `'oldest'`

Specify whether to return the latest or the oldest data samples. The number of samples depends on the `SamplesPerRead` value. The data read from the GPS receiver is stored in the MATLAB buffer.

- `latest` — Provides the latest data samples available in the buffer. All previous data samples in the buffer are discarded. For example, if `SamplesPerRead` = 3, the latest three data samples read by the GPS receiver are returned.
- `oldest` — Provides the oldest data samples available in the buffer. In this case, no data samples are discarded. For example, if `SamplesPerRead` = 3, the first three data samples read are returned for the first read, the next three data samples are returned for the second read, and so on.

Tunable: No

Data Types: `character vector` | `string`

`SamplesRead` — Samples read
double

This property is read-only.

Number of samples read from the GPS receiver using the `read` function, after the object is locked. The `gpsdev` object gets locked either at the first call of the `read` function after the object creation or at the first call of the read function after the execution of the `release` function.

Data Types: `double`

`SamplesAvailable` — Samples in the host buffer
double

This property is read-only.

Samples available in the host buffer. When you `release` the object, `SamplesAvailable` is set to 0.

Data Types: `double`

`SamplesPerRead` — Samples per read
1 (default)

Samples read from the first `read`, specified as a positive integer in the range [1 10].

Tunable: No

Data Types: `double`

`OutputFormat` — Set output format
`'timetable'` (default) | `'matrix'`

Set the output format of the data returned by executing the `read` function.

When the `OutputFormat` is set to `timetable`, the `timetable` returned has the following fields:

- LLA (Latitude, Longitude, Altitude)
- Ground Speed
- Course over ground
- Dilution of Precisions(DOPs), VDOP,HDOP,PDOP
- GPS Receiver Time
- Time — System time when the data is read, in `datetime` or `duration` format

When the `OutputFormat` is set to `matrix`, the data is returned as matrices of Time, LLA, Ground Speed, Course over ground, DOPs, and GPS receiver time. The units for the GPS receiver readings are the same as the `timetable` format.

Tunable: Yes

Data Types: `character vector` | `string`

`TimeFormat` — Set the format of the time displayed when the GPS data is read
`'datetime'` (default) | `'duration'`

Set the format of the time displayed when the GPS data is read.

- `datetime` — Displays the date and time at which the data is read.
- `duration` — Displays the time elapsed in seconds after the GPS object is locked. The `gpsdev` object gets locked either at the first call of the `read` function after the object creation or at the first call of the read function after the execution of the `release` function.

Tunable: Yes

Data Types: `character vector` | `string`

Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

flush	Flush all GPS data accumulated in the buffers and reset properties
info	Read update rate, GPS lock information and number of satellites in view for the GPS receiver
read	Read data from GPS receiver
release	Release the GPS object
writeBytes	Write raw commands to the GPS receiver

Examples

Plot Geographic Position Using GPS Connected to Host Computer

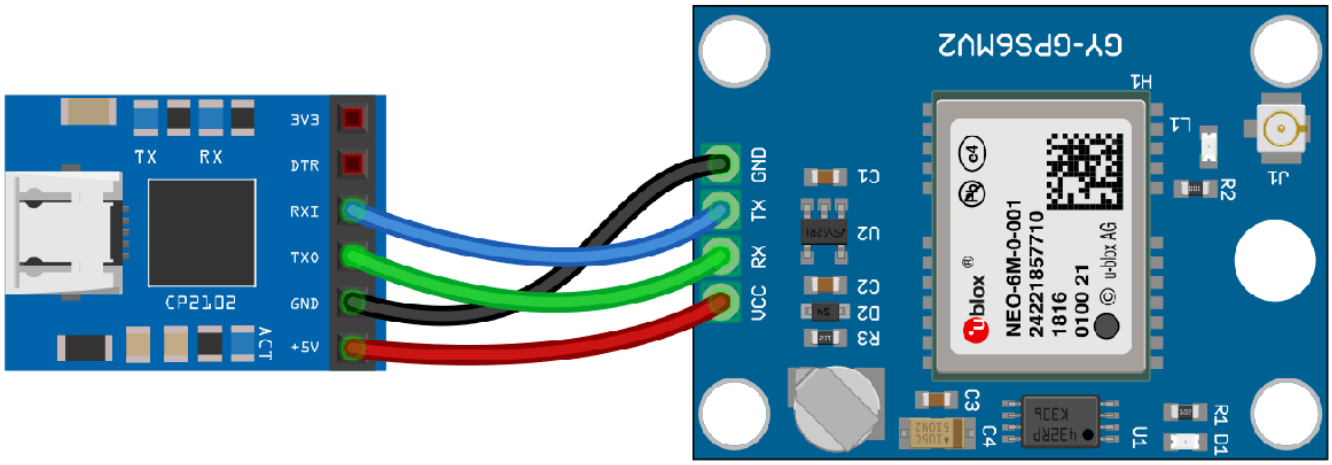
Get the geographic location using the GPS receiver connected to the host computer on a specific serial port and plot the location in a map.

Required Hardware

To run this example, you need:

- UBlox Neo-6M GPS module
- GPS antenna
- USB to UART module
- USB cable
- Connecting wires

Hardware Connection



Connect the pins on the UBlox Neo-6M GPS module to the pins on your USB to UART module. The connections are:

- VCC - +5V
- RX - TXO
- TX - RXI
- GND - GND

Connect the GPS antenna to the GPS module. Connect the USB to UART module to the host computer with a USB cable. GPS Fix can be easily acquired in locations that have a clear view of the sky. Wait for the GPS module to acquire satelite signals (Fix).This can be verified by checking the Fix LED (D1) of your GPS module.

Create GPS Object

Create a `gpsdev` object for the GPS module connected to a specific port.

```
gps = gpsdev('COM4')
```

```
gps =
  gpsdev with properties:

                      SerialPort: COM4
                        BaudRate: 9600 (bits/s)
```

```
                    SamplesPerRead: 1
                          ReadMode: "latest"
                       SamplesRead: 0
Show all properties all functions
```

Read the GPS data

Read the GPS data and extract latitude, longitude, and time from it. GPS returns UTC datetime. Convert it to system time zone.

```
[gpsData,~] = read(gps);
latitude = gpsData.LLA(1);
longitude = gpsData.LLA(2);
gpsTime = gpsData.GPSReceiverTime;
gpsTime.TimeZone = 'local';
```

Plot the position in a map along with the timestamp

Plot the position in geographic axes with the data obtained from the GPS module. GPS should have fix to get valid values for latitude, longitude and `gpsTime`.

If the GPS module does not have fix, the above commands give `NaN`s for latitude and longitude and `NaT` for `gpsTime`. In this case, make sure the antenna is exposed to clear sky and wait for some time and try the above steps again.

```
if(~isnan(latitude) && ~isnan(longitude))
 % plot the position in geographic coordinates
 fig = geoplot(latitude,longitude,'Marker',"o",'MarkerSize',6,'Color','red','MarkerFaceColor','re

 % Sets the latitude and longitude limits of the base Map
 geolimits([latitude-0.05 latitude+0.05],[longitude-0.05 longitude+0.05]) ;

 % Selects the basemap
 geobasemap streets;
 timeString = strcat("Timestamp: ",string(gpsTime));

 % Create annotation and display time received from GPS
 annotation('textbox',[0.005 0.98 0.6 0.01],'FitBoxToText','on','string',timeString,'Color','blue
end
```

Clean Up

When the connection is no longer needed, clear the associated object.

```
delete(gps);
clear gps;
```

Write Configuration Commands to GPS Receiver

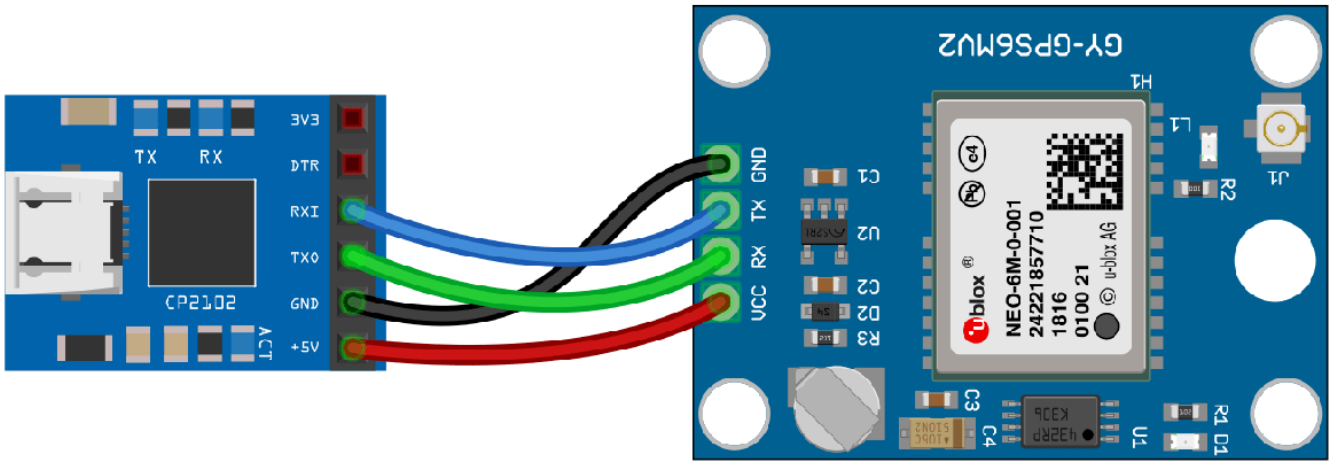
Write configuration commands to the GPS receiver connected to the host computer using `serialport` object.

Required Hardware

To run this example, you need:

- UBlox Neo-6M GPS module
- GPS antenna
- USB to UART module
- USB cable
- Connecting wires

Hardware Connection



Connect the pins on the UBlox Neo-6M GPS module to the pins on your USB to UART module. The connections are:

- VCC - +5V
- RX - TXO
- TX - RXI
- GND - GND

Connect the GPS antenna to the GPS module. Connect the USB to UART module to the host computer with a USB cable. GPS Fix can be easily acquired in locations that have a clear view of the sky. Wait for the GPS module to acquire satelite signals (Fix).This can be verified by checking the Fix LED (D1) of your GPS module.

Create GPS Object

Connect to the GPS receiver using `serialport` object. Specify the port name and the baud rate.

```
s = serialport('COM4',9600)
```

```
s =
  Serialport with properties:

                Port: "COM4"
            BaudRate: 9600
    NumBytesAvailable: 0

  Show all properties, functions
```

```
gps = gpsdev(s)

gps =
  gpsdev with properties:

                        SerialPort: COM4
                          BaudRate: 9600 (bits/s)

                     SamplesPerRead: 1
                           ReadMode: "latest"
                        SamplesRead: 0
Show all properties all functions
```

Write Configuration Commands

In the default configuration the GPS receiver returns the following NMEA messages: GPRMC, GPVTG, GPGGA, GPGSA, GPGSV, and GPGLL. The receiver can be configured to have a user defined set of output messages.

Read few lines of default messages from the serial port the GPS receiver is connected.

```
for i = 1:10
data = readline(s);
disp(data);
end
```

```
$GPRMC,,V,,,,,,,,,N*53
$GPVTG,,,,,,,,,N*30
$GPGGA,,,,,,0,00,99.99,,,,,,*48
$GPGSA,A,1,,,,,,,,,,,,,99.99,99.99,99.99*30
$GPGSV,2,1,08,01,,,18,08,,,12,09,,,12,15,,,19*77
$GPGSV,2,2,08,23,,,13,24,,,09,25,,,10,27,,,25*79
$GPGLL,,,,,,V,N*64
$GPRMC,,V,,,,,,,,,N*53
$GPVTG,,,,,,,,,N*30
$GPGGA,,,,,,0,00,99.99,,,,,,*48
```

Write the version monitor command to the GPS receiver to return the software and hardware version of the GPS receiver.

```
configCMD = [0xB5 0x62 0x0A 0x04 0x00 0x00 0x0E 0x34];
% writeBytes(gps,cfg)
write(s,configCMD,'uint8')
```

Read few lines of messages again to verify the version message.

```
for i = 1:10
data = readline(s);
disp(data);
end
```

```
$GPGSA,A,1,,,,,,,,,,,,,99.99,99.99,99.99*30
$GPGSV,2,1,05,01,,,13,09,,,11,15,,,16,23,,,12*74
$GPGSV,2,2,05,25,,,10*7A
$GPGLL,,,,,,V,N*64
µb
( 7.03 (45969)                  00040007  °$GPRMC,,V,,,,,,,,,N*53
$GPVTG,,,,,,,,,N*30
```

```
$GPGGA,,,,,,0,00,99.99,,,,,,*48
$GPSA,A,1,,,,,,,,,,,,,99.99,99.99,99.99*30
$GPGSV,2,1,06,01,,,11,09,,,11,23,,,14,24,,,21*75
```

It can be observed from the output, `7.03 (45969)` is the software version and `00040007` is the hardware version.

Clean Up

When the connection is no longer needed, clear the associated object.

```
delete(gps);
clear gps;
clear s;
```

Read Data from GPS Receiver as Timetable

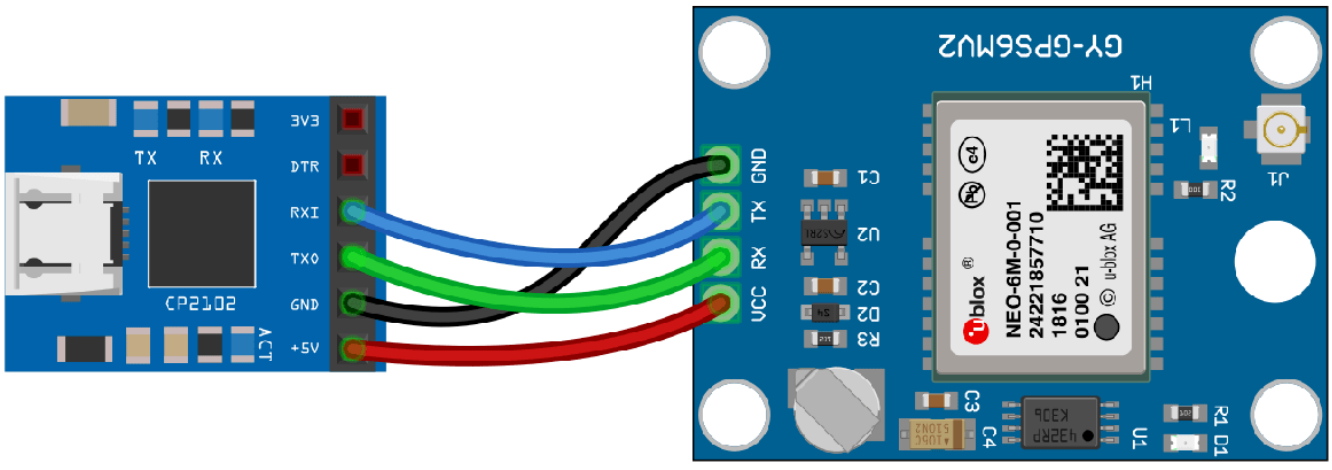
Read data from the GPS receiver connected to the host computer on a specific serial port.

Required Hardware

To run this example, you need:

- UBlox Neo-6M GPS module
- GPS antenna
- USB to UART module
- USB cable
- Connecting wires

Hardware Connection



Connect the pins on the UBlox Neo-6M GPS module to the pins on your USB to UART module. The connections are:

- VCC - +5V

- RX - TXO
- TX - RXI
- GND - GND

Connect the GPS antenna to the GPS module. Connect the USB to UART module to the host computer with a USB cable. GPS Fix can be easily acquired in locations that have a clear view of the sky. Wait for the GPS module to acquire satelite signals (Fix).This can be verified by checking the Fix LED (D1) of your GPS module.

Create GPS Object

Create a `gpsdev` object for the GPS receiver connected to a specific port. Specify the output format of the data as a timetable.

```
gps = gpsdev('COM4','OutputFormat',"timetable")
```

```
gps = 
  gpsdev with properties:

                         SerialPort: COM4
                           BaudRate: 9600 (bits/s)

                     SamplesPerRead: 1
                           ReadMode: "latest"
                        SamplesRead: 0
Show all properties all functions
```

Read the GPS data

Read the GPS data and return them as a timetable.

```
[tt,overruns] = read(gps)
```

```
tt=1×5 timetable
              Time                           LLA                  GroundSpeed    Course         DO
    ________________________    _________________________    ___________    ______    __________

    22-Mar-2021 15:31:15.190    17.47    78.343      449.6      0.25619        NaN       9.31    1.
```

```
overruns = 0
```

Display number of samples read and the samples available in the host buffer.

```
gps.SamplesRead
```

```
ans = 1
```

```
gps.SamplesAvailable
```

```
ans = 0
```

Release the GPS object to configure the non tunable properties. The release function also clears the buffer and resets the `SamplesRead` and `SamplesAvailable` properties.

```
release(gps)
```

Specify the number of samples per read to 2. Read the GPS data.

```
gps.SamplesPerRead = 2;
read(gps)
```

```
ans=2×5 timetable
            Time                            LLA                GroundSpeed    Course         DOI
    ________________________    ________________________    ___________    ______    __________

    22-Mar-2021 15:31:17.178    17.47    78.343      450     0.063791       NaN      9.32    1.4
    22-Mar-2021 15:31:17.178    17.47    78.343      450     0.063791       NaN      9.32    1.4
```

Display number of samples read and the samples available in the host buffer.

```
gps.SamplesRead
```

```
ans = 1
```

```
gps.SamplesAvailable
```

```
ans = 0
```

Clean Up

When the connection is no longer needed, clear the associated object.

```
delete(gps);
clear gps;
```

More About

GPS Modules

To verify the functionality, the following GPS modules were used:

- Adafruit Ultimate GPS
- Ublox NEO 6M
- Ublox NEO 7M

See Also

`nmeaParser`

Introduced in R2020b

nmeaParser

Parse data from standard and manufacturer-specific NMEA sentences sent from marine electronic devices

Description

The `nmeaParser` System object parses data from NMEA (National Marine Electronics Association) sentences. The sentences that need parsing of data can be standard sentences compliant with the NMEA 0183® specifications (which are sent from a GNSS (Global Navigation Satellite System) receiver), or other manufacturer-specific sentences approved by the NMEA (which are sent from other marine electronic devices).

The `nmeaParser` System object provides:

- Built-in support to parse data sent from GNSS receivers and identified by these nine NMEA message types: RMC, GGA, GSA, VTG, GLL, GST, ZDA, GSV, and HDT
- Additional configuration using the `CustomSentence` name-value pair to parse NMEA data from multiple device categories, including manufacturer-specific sentences from different hardware manufacturers

To parse data from NMEA sentences:

1. Create the `nmeaParser` object and set its properties.
2. Call the object with arguments, as if it were a function.

To learn more about how System objects work, see What Are System Objects?

The `nmeaParser` System object outputs an array of structures corresponding to the values extracted from the specified NMEA sentences.

Creation

Syntax

```
pnmea = nmeaParser
pnmea = nmeaParser("MessageIDs", 'msgID')
pnmea = nmeaParser("CustomSentence",
{['CustomMessageId1','parserFunctionName1'],
['CustomMessageId2','parserFunctionName2']})
pnmea = nmeaParser("MessageIDs", {'msgID1','msgID2'},"CustomSentence",
{['CustomMessageId1','parserFunctionName1'],
['CustomMessageId2','parserFunctionName2']})
```

Description

`pnmea = nmeaParser` returns a `nmeaParser` System object, `pnmea`, with default properties, that extracts data from these standard NMEA messages: RMC, GGA, and GSA. The order of structure arrays in the extracted output data is also: RMC, GGA, and GSA.

`pnmea = nmeaParser("MessageIDs", 'msgID')` returns a `nmeaParser` System object, `pnmea`, that extracts data from one of the nine standard NMEA messages with built-in support, specified using the Message IDs. Specify `msgID` as `"RMC"`, `"GGA"`, `"GSA"`, `"GSV"`, `"VTG"`, `"GLL"`, `"GST"`, `"ZDA"`, and `"HDT"`, or a combination of these IDs (for example: `["VTG","GLL","HDT"]`). The order in which you specify the Message IDs determines the order of the structure arrays in the extracted output data. The default value is `["RMC","GGA","GSA"]`.

`pnmea = nmeaParser("CustomSentence", {['CustomMessageId1','parserFunctionName1'], ['CustomMessageId2','parserFunctionName2']})` sets properties using the `CustomSentence` name-value pair and returns a `nmeaParser` System object, `pnmea`, that extracts data from custom NMEA message (either standard NMEA message or manufacturer-specific NMEA message), specified using the message IDs.

The `CustomSentence` name-value pair accepts a nested cell array where each element is a pair of message ID name (either standard NMEA message ID name or manufacturer-specific message ID) and the corresponding user-defined parser function, which is created by including the `extractNMEASentence` function in a function file. The order in which you specify the message IDs determines the order of the structure arrays in the extracted output data.

`pnmea = nmeaParser("MessageIDs", {'msgID1','msgID2'},"CustomSentence", {['CustomMessageId1','parserFunctionName1'], ['CustomMessageId2','parserFunctionName2']})` returns a `nmeaParser` System object, `pnmea`, that extracts data from two of the nine standard NMEA messages with built-in support and also from custom NMEA messages that you specified using the `CustomSentence` name-value pair.

Input Arguments

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `pnmea = nmeaParser("CustomSentence", {['CustomMessageId1','parserFunctionName1'], ['CustomMessageId2','parserFunctionName2']});`

`CustomSentence` — Specify message ID of sentence and the name of its parser function
cell array of character vectors

Specify the message ID of NMEA sentence from which you want to extract data and the name of the parser function. You can specify multiple message IDs and parser functions as a cell array of character vectors. The parser function is defined in a function file, which can optionally include the `extractNMEASentence` function.

Note The function file for the parser function must be present in the current directory or on MATLAB path.

`CustomSentence` accepts function name or function handle. For example, both these formats are valid:

- `pnmea = nmeaParser('CustomSentence', {['standardnmeaMessageId1','parserFunctionName1']}`

- `parserFunctionHandle=@parserFunctionName1`
  `pnmea = nmeaParser('CustomSentence', {['standardnmeaMessageId1',parserFunctionHandle]}`

Note Using `CustomSentence` name-value pair to parse data instead of the built-in parser function results in override of the default fields when data is parsed. For example, `nmeaParser('CustomSentence',{["RMC","parserRMC"]})` overrides the default format of parsed data as RMC is one of the nine sentences with built-in support for parsing data.

Data Types: `char` | `string`

Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see System Design in MATLAB Using System Objects.

`MessageIDs` — Message IDs of nine standard NMEA sentences with built-in support to extract data

`["RMC","GGA","GSA"]` (default) | `RMC` | `GGA` | `GSA` | `GSV` | `VTG` | `GLL` | `GST` | `ZDA` | `HDT`

Message IDs of nine NMEA sentences with built-in support, which are compliant with the NMEA 0183 Standard, from which you want to extract data. You can specify multiple message IDs as an array of strings to extract data from NMEA sentences.

Data Types: `char` | `string`

Usage

Syntax

```
[rmcData,ggaData,gsaData,vtgData,gllData,gstData,gsvData,zdaData,hdtData] =
pnmea(rawData)
[customNmeaData1,customNmeaData2] = pnmea(rawData)
```

Description

`[rmcData,ggaData,gsaData,vtgData,gllData,gstData,gsvData,zdaData,hdtData] = pnmea(rawData)` parses data from nine standard NMEA sentences with built-in support, and returns an array of structures, where each structure corresponds to a single Message ID. The sequence that you specify for the output arguments must be the same sequence that you specified for the Message IDs when creating the `nmeaParser` System object.

`[customNmeaData1,customNmeaData2] = pnmea(rawData)` parses data from two custom NMEA sentences (either standard NMEA sentence or manufacturer-specific NMEA sentence), and returns an array of structures, where each structure corresponds to a single Message ID. The sequence that you specify for the output arguments must be the same sequence that you specified in the `CustomSentence` name-value pair when creating the `nmeaParser` System object.

Input Arguments

`rawData` — NMEA data as obtained from a marine electronic device

string array

NMEA data, which is compliant with NMEA standard, as obtained from a marine electronic device.

Data Types: `string` | `char`

Output Arguments

`rmcData` — Data extracted from RMC sentence

structure

Data extracted from an RMC sentence. The output structure contains the information parsed from an RMC sentence along with the parsing status. If multiple RMC sentences are found in the input data, then an array of structures is returned. For more details, see “RMC Sentences” on page 2-647.

`ggaData` — Data extracted from GGA sentence

structure

Data extracted from a GGA sentence. The output structure contains the information parsed from a GGA sentence along with the parsing status. If multiple GGA sentences are found in the input data, then an array of structures is returned. For more details, see “GGA Sentences” on page 2-648.

`gsaData` — Data extracted from GSA sentence

structure

Data extracted from a GSA sentence. The output structure contains the information parsed from a GSA sentence along with the parsing status. If multiple GSA sentences are found in the input data, then an array of structures is returned. For more details, see “GSA Sentences” on page 2-650.

`vtgData` — Data extracted from VTG sentence

structure

Data extracted from a VTG sentence. The output structure contains the information parsed from a VTG sentence along with the parsing status. If multiple VTG sentences are found in the input data, then an array of structures is returned. For more details, see “VTG Sentences” on page 2-654.

`gllData` — Data extracted from GLL sentence

structure

Data extracted from a GLL sentence. The output structure contains the information parsed from a GLL sentence along with the parsing status. If multiple GLL sentences are found in the input data, then an array of structures is returned. For more details, see “GLL Sentences” on page 2-653.

`gstData` — Data extracted from GST sentence

structure

Data extracted from a GST sentence. The output structure contains the information parsed from a GST sentence along with the parsing status. If multiple GST sentences are found in the input data, then an array of structures is returned. For more details, see “GST Sentences” on page 2-655.

`gsvData` — Data extracted from GSV sentence

structure

Data extracted from a GSV sentence. The output structure contains the information parsed from a GSV sentence along with the parsing status. The complete satellite information is available in multiple gsvData structures. Each gsvData structure can have a maximum of four satellite information. For more details, see “GSV Sentences” on page 2-651.

`zdaData` — Data extracted from ZDA sentence
structure

Data extracted from a ZDA sentence. The output structure contains the information parsed from a ZDA sentence along with the parsing status. If multiple ZDA sentences are found in the input data, then an array of structures is returned. For more details, see “ZDA Sentences” on page 2-653.

`hdtData` — Data extracted from HDT sentence
structure

Data extracted from an HDT sentence. The output structure contains the information parsed from an HDT sentence along with the parsing status. If multiple HDT sentences are found in the input data, then an array of structures is returned. For more details, see “HDT Sentences” on page 2-656.

`customNmeaData1` — Data extracted from standard or manufacturer-specific NMEA sentence
structure

Data extracted from standard or manufacturer-specific NMEA sentence. The output structure contains the information parsed from the custom sentence along with the parsing status. If multiple sentences of the same NMEA message type are found in the input data, then an array of structures is returned.

Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

Examples

Extract Data from NMEA Sentences Using MessageID Property

Extract data from any of the nine standard NMEA sentences as part of the built-in support using the MessageID property. The NMEA data is obtained from a GNSS receiver.

Extract Data from RMC Sentence

Create an nmeaParser System Object by specifying the Message ID as "RMC".

```
pnmea = nmeaParser("MessageID","RMC");
```

Provide the RMC sentence obtained from the GNSS receiver as the input and extract data.

```
unparsedRMCLine='$GNRMC,143909.00,A,5107.0020216,N,11402.3294835,W,0.036,348.3,210307,0.0,E,A*31
rmcData = pnmea(unparsedRMCLine)
```

```
rmcData = struct with fields:
             TalkerID: "GN"
```

```
            MessageID: "RMC"
            FixStatus: 'A'
             Latitude: 51.1167
            Longitude: -114.0388
          GroundSpeed: 0.0185
       TrueCourseAngle: 348.3000
           UTCDateTime: 21-Mar-2007 14:39:09.000
     MagneticVariation: 0
         ModeIndicator: 'A'
      NavigationStatus: "NA"
                Status: 0
```

Extract Data from Multiple NMEA Message Types

Provide GGA, GSA, and RMC sentences as the input.

```
unparsedGGALine = ['$GPGGA,111357.771,5231.364,N,01324.240,E,1,12,1.0,0.0,M,0.0,M,,*69'];
unparsedGSALine = ['$GPGSA,A,3,01,02,03,04,05,06,07,08,09,10,11,12,1.0,1.0,1.0*30'];
unparsedRMCLine = ['$GPRMC,111357.771,A,5231.364,N,01324.240,E,10903,221.5,020620,000.0,W*44'];
```

Create a string array to include the three sentences

```
rawNMEAData = [unparsedGGALine ,newline,  unparsedGSALine ,newline,  unparsedRMCLine]
```

```
rawNMEAData =
    '$GPGGA,111357.771,5231.364,N,01324.240,E,1,12,1.0,0.0,M,0.0,M,,*69
     $GPGSA,A,3,01,02,03,04,05,06,07,08,09,10,11,12,1.0,1.0,1.0*30
     $GPRMC,111357.771,A,5231.364,N,01324.240,E,10903,221.5,020620,000.0,W*44'
```

However, consider that you need to extract data only from GGA and GSA sentences. So create the nmeaParser System Object 'pnmea', and specify the 'GGA' and 'GSA' Message IDs as a string array.

```
pnmea=nmeaParser("MessageIDs",["GGA","GSA"]);
```

Specify the output arguments for all the three sentences to extract the data as structures.

```
[ggaData,gsaData] =  pnmea(rawNMEAData)
```

```
ggaData = struct with fields:
                          TalkerID: "GP"
                         MessageID: "GGA"
                           UTCTime: 11:13:57.771
                          Latitude: 52.5227
                         Longitude: 13.4040
                  QualityIndicator: 1
                NumSatellitesInUse: 12
                              HDOP: 1
                          Altitude: 0
                   GeoidSeparation: 0
              AgeOfDifferentialData: NaN
    DifferentialReferenceStationID: NaN
                            Status: 0


gsaData = struct with fields:
              TalkerID: "GP"
             MessageID: "GSA"
```

```
                Mode: "A"
             FixType: 3
    SatellitesIDNumber: [1 2 3 4 5 6 7 8 9 10 11 12]
                PDOP: 1
                VDOP: 1
                HDOP: 1
            SystemID: NaN
              Status: 0
```

The above output shows that only GGA and GSA sentences are extracted based on the Message IDs specified as input.

Provide another GGA sentence as an additional input, and extract data. In this case, you need not modify the System object as the Message ID has not changed.

```
unparsedGGALine1='$GNGGA,001043.00,4404.14036,N,12118.85961,W,1,12,0.98,1113.0,M,-21.3,M,,*47'
```

```
unparsedGGALine1 =
'$GNGGA,001043.00,4404.14036,N,12118.85961,W,1,12,0.98,1113.0,M,-21.3,M,,*47'
```

```
rawNMEAData = [unparsedGGALine ,newline,  unparsedGSALine ,newline,  unparsedGGALine1]
```

```
rawNMEAData =
    '$GPGGA,111357.771,5231.364,N,01324.240,E,1,12,1.0,0.0,M,0.0,M,,*69
     $GPGSA,A,3,01,02,03,04,05,06,07,08,09,10,11,12,1.0,1.0,1.0*30
     $GNGGA,001043.00,4404.14036,N,12118.85961,W,1,12,0.98,1113.0,M,-21.3,M,,*47'


[ggaData,gsaData] =  pnmea(rawNMEAData)
```

```
ggaData=2×1 struct array with fields:
    TalkerID
    MessageID
    UTCTime
    Latitude
    Longitude
    QualityIndicator
    NumSatellitesInUse
    HDOP
    Altitude
    GeoidSeparation
    AgeOfDifferentialData
    DifferentialReferenceStationID
    Status


gsaData = struct with fields:
             TalkerID: "GP"
            MessageID: "GSA"
                Mode: "A"
              FixType: 3
    SatellitesIDNumber: [1 2 3 4 5 6 7 8 9 10 11 12]
                PDOP: 1
                VDOP: 1
                HDOP: 1
            SystemID: NaN
              Status: 0
```

A status of **0** indicates that the data was parsed successfully.

Extract Data from GSV Sentence

Create an nmeaParser System Object by specifying the Message ID as "GSV".

```
pnmea = nmeaParser("MessageID","GSV");
```

Provide the GSV sentence obtained from the GNSS receiver as the input and extract data.

```
unparsedGSVLine='$GPGSV,3,3,10,32,69,205,41,46,47,215,39*79';
gsvData = pnmea(unparsedGSVLine)
```

```
gsvData = struct with fields:
            TalkerID: "GP"
           MessageID: "GSV"
        NumSentences: 3
      SentenceNumber: 3
    SatellitesInView: 10
         SatelliteID: [32 46]
           Elevation: [69 47]
             Azimuth: [205 215]
                 SNR: [41 39]
            SignalID: NaN
              Status: 0
```

Extract Data from Multiple GSV Sentences

Provide multiple GSV sentences as the input.

```
unparsedGSVLine1 = '$GPGSV,3,1,10,01,,,31,03,28,325,40,10,,,33,12,20,047,30*70';
unparsedGSVLine2 = '$GPGSV,3,2,10,14,88,028,42,22,39,299,48,25,,,25,31,79,289,46*49';
unparsedGSVLine3 = '$GPGSV,3,3,10,32,69,205,41,46,47,215,39*79';
```

Create a string array to include the three sentences.

```
CRLF = [char(13),newline];
unparsedGSVLines = [unparsedGSVLine1,CRLF, unparsedGSVLine2, CRLF, unparsedGSVLine3];
```

Create the nmeaParser System Object 'pnmea', specify the 'GSV' Message ID, and extract data.

```
pnmea = nmeaParser("MessageIDs","GSV");
gsvData = pnmea(unparsedGSVLines)
```

```
gsvData=3×1 struct array with fields:
    TalkerID
    MessageID
    NumSentences
    SentenceNumber
    SatellitesInView
    SatelliteID
    Elevation
    Azimuth
    SNR
    SignalID
    Status
```

Read Data from NMEA Log

Read data from a sample NMEA log, so that the data can be parsed using the nmeaParser System Object.

The sample log file is `nmeaLog.nmea`, which is included in this example.

```
f = fopen('nmeaLog.nmea');
unParsedNMEAdata = fread(f);
pnmea = nmeaParser("MessageIDs",["RMC","GGA"]);
[rmcStruct, ggaStruct] = pnmea(unParsedNMEAdata)
```

```
rmcStruct=9×1 struct array with fields:
    TalkerID
    MessageID
    FixStatus
    Latitude
    Longitude
    GroundSpeed
    TrueCourseAngle
    UTCDateTime
    MagneticVariation
    ModeIndicator
    NavigationStatus
    Status


ggaStruct=9×1 struct array with fields:
    TalkerID
    MessageID
    UTCTime
    Latitude
    Longitude
    QualityIndicator
    NumSatellitesInUse
    HDOP
    Altitude
    GeoidSeparation
    AgeOfDifferentialData
    DifferentialReferenceStationID
    Status
```

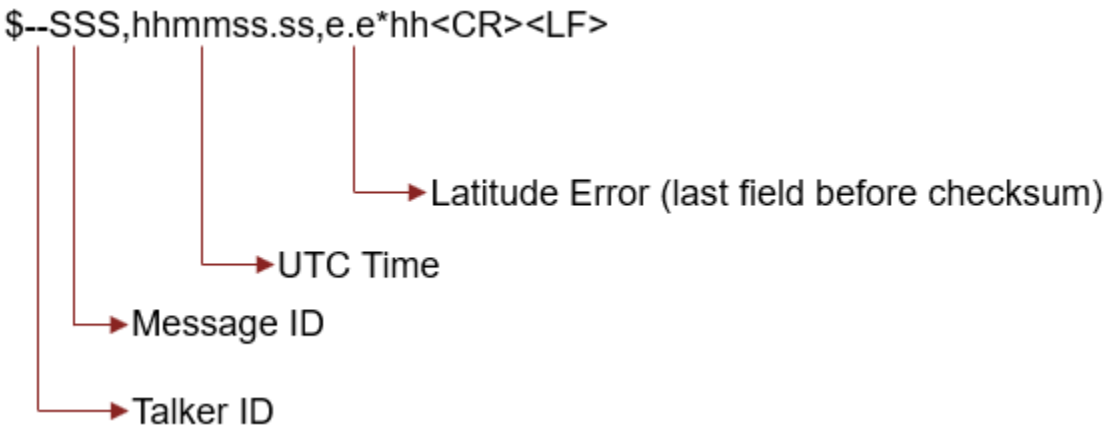
Extract Data Using CustomSentence Name-Value Pair

You can extract data from NMEA sentences using the `CustomSentence` name-value pair. The NMEA data to be parsed is obtained from marine electronic devices.

Identify the Structure of NMEA Sentence and Create the Parser Function

You need to identify the structure of the NMEA sentence, as defined in the specification, and use that information to define the structure of output data to be used in the nmeaParser System object.

For example, let us consider an example sentence with Message ID, `SSS`.



After identifying the structure, you create a function file that defines the parser function, **fsssParser.** In the function file, you define the output data as a structure array with its fields matching the sequence as it appears in the specification.

The Navigation Toolbox™ provides an optional pre-configured function, `extractNMEASentence`, that checks if the sentence is valid and convert the fields in the sentence into string array. You can call `extractNMEASentence` inside the function file. You can also use any other function instead (which outputs a string array from unparsed data), and then call it inside the function file.

The below image shows the function file with the code, with the assumption that the fields available in `SSS` sentence are TalkerID, MessageID, UTC, and LatitudeError. Refer the additional comments for details.

fsssParser.m

```
function OutputData = fsssParser(unparsedData, MessageId)
% fsssParser custom function to parse SSS sentence

% Define OutputData as a structure array with fields in the same sequence as they appear in
% SSS sentence structure
OutputData = struct('TalkerID',"",'MessageID',"",'UTC',nan,'LatitudeError',nan);

% Call another function to convert the unparsed data into string array.
% To do this, use either extractNMEASentence function
% or any other function that outputs a string array.
[valid, splitString] = extractNMEASentence(unparsedData,MessageId);

% Map each field of the structure to the corresponding string element
if(valid)
    OutputData.TalkerID = splitString(1);
    OutputData.MessageID = splitString(2);
    temp = char(splitString(3));
    utcOutput = [temp(1:2), ':', temp(3:4), ':', temp(5:9)];
    OutputData.UTC = utcOutput;
    OutputData.LatitudeError = str2double(splitString(4));

end

end
```

In the above sample file, you define the mapping between the fields in the structure array and the elements of string array. For certain fields (for example, UTC time), you may need to define a character array to map the fields.

After you include the MATLAB code as mentioned above, save the function file (**fsssParser.m**) in the MATLAB path, so that you can call it to obtain parsed data using the `CustomSentence` name-value pair of nmeaParser System object.

To download another sample function file **parserRMB.m** that is used this example, click **Open Live Script**. This is a function file specific to the fields of an RMB sentence (mentioned in NMEA Standard, Version 4.1).

Extract Data from RMB Sentence

Create an nmeaParser System Object by using the CustomSentence name-value pair and specifying the message ID as "RMB" and the function as "parserRMB" (downloaded in the previous step).

```
pnmea = nmeaParser("CustomSentence",{["RMB","parserRMB"]});
```

Provide the RMB sentence obtained from the GNSS receiver as the input and extract data.

```
unparsedRMBLine='$GPRMB,A,4.08,L,EGLL,EGLM,5130.02,N,00046.34,W,004.6,213.9,122.9,A*3D';
rmbData = pnmea(unparsedRMBLine)
```

```
rmbData = struct with fields:
                      TalkerID: "GP"
```

```
                    MessageID: "RMB"
                   DataStatus: 'A'
               CrossTrackError: 4.0800
              DirectionToSteer: NaN
              OriginWaypointID: NaN
         DestinationWaypointID: NaN
  DestinationWaypointLatitude: '5130.02 N'
 DestinationWaypointLongitude: '00046.34 W'
           RangeToDestination: 4.6000
          BearingToDestination: 213.9000
   DestinationClosingVelocity: 122.9000
                ArrivalStatus: 'A'
                ModeIndicator: "NA"
                       Status: 0
```

Extract Data from Multiple RMB Sentences

Provide multiple RMB sentences as the input.

```
unparsedRMBLine1 = ['$GPRMB,A,0.66,L,003,004,4917.24,N,12309.57,W,001.3,052.5,000.5,V*20'];
unparsedRMBLine2 = ['$GPRMB,A,4.08,L,EGLL,EGLM,5130.02,N,00046.34,W,004.6,213.9,122.9,A*3D'];
```

Create a character array to include the two sentences

```
rawNMEAData = [unparsedRMBLine1 ,newline,  unparsedRMBLine2]
```

```
rawNMEAData =
    '$GPRMB,A,0.66,L,003,004,4917.24,N,12309.57,W,001.3,052.5,000.5,V*20
     $GPRMB,A,4.08,L,EGLL,EGLM,5130.02,N,00046.34,W,004.6,213.9,122.9,A*3D'
```

Specify the output argument for the RMB sentence to extract the data.

```
[rmbData] =  pnmea(rawNMEAData)
```

```
rmbData=2×1 struct array with fields:
    TalkerID
    MessageID
    DataStatus
    CrossTrackError
    DirectionToSteer
    OriginWaypointID
    DestinationWaypointID
    DestinationWaypointLatitude
    DestinationWaypointLongitude
    RangeToDestination
    BearingToDestination
    DestinationClosingVelocity
    ArrivalStatus
    ModeIndicator
    Status
```

Extract Data from a Sentence with Built-in Support (RMC) and RMB Sentence

Create an nmeaParser System Object by using the MessageID property (to parse a sentence with built-in support - RMC) and also using the CustomSentence name-value pair (specifying the message ID as "RMB" and the function as "parserRMB" (created in a previous step)).

```
pnmea = nmeaParser("MessageID","RMC","CustomSentence",{["RMB","parserRMB"]});
```

Provide RMC and RMB sentences as the input.

```
unparsedRMCLine1 = ['$GNRMC,143909.00,A,5107.0020216,N,11402.3294835,W,0.036,348.3,210307,0.0,E,A
unparsedRMBLine2 = ['$GPRMB,A,4.08,L,EGLL,EGLM,5130.02,N,00046.34,W,004.6,213.9,122.9,A*3D'];
```

Create a string array to include the two sentences

```
rawNMEAData = [unparsedRMCLine1 ,newline,  unparsedRMBLine2]
```

```
rawNMEAData =
    '$GNRMC,143909.00,A,5107.0020216,N,11402.3294835,W,0.036,348.3,210307,0.0,E,A*31
     $GPRMB,A,4.08,L,EGLL,EGLM,5130.02,N,00046.34,W,004.6,213.9,122.9,A*3D'
```

Specify the output argument for the RMB sentence to extract the data.

```
[rmcdata,rmbData] =  pnmea(rawNMEAData)
```

```
rmcdata = struct with fields:
            TalkerID: "GN"
           MessageID: "RMC"
           FixStatus: 'A'
            Latitude: 51.1167
           Longitude: -114.0388
         GroundSpeed: 0.0185
      TrueCourseAngle: 348.3000
          UTCDateTime: 21-Mar-2007 14:39:09.000
    MagneticVariation: 0
        ModeIndicator: 'A'
     NavigationStatus: "NA"
               Status: 0


rmbData = struct with fields:
                       TalkerID: "GP"
                      MessageID: "RMB"
                     DataStatus: 'A'
                CrossTrackError: 4.0800
               DirectionToSteer: NaN
               OriginWaypointID: NaN
          DestinationWaypointID: NaN
     DestinationWaypointLatitude: '5130.02 N'
    DestinationWaypointLongitude: '00046.34 W'
             RangeToDestination: 4.6000
            BearingToDestination: 213.9000
      DestinationClosingVelocity: 122.9000
                  ArrivalStatus: 'A'
                  ModeIndicator: "NA"
                         Status: 0
```

Extract Data from a Manufacturer-specific Sentence Using CustomSentence Name-Value Pair

1 Identify Structure of Manufacturer-specific Sentence and Create Function File with Parser Function

The structure of NMEA sentence to be parsed is available in the specification of the device from the manufacturer. You need to identify the structure and use the information to define the structure of output data to be used in the nmeaParser System object.

For example, the structure of the NMEA sentence from a hardware manufacturer may look like this:

`$PMMCZ,hhmmss.ss,Latitude,N,Longitude,E,NavSatellite,DR*hh<CR><LF>`

Here, P denotes that the sentence is manufacturer-specific, MMC is the manufacturer mnemonic code, and Z is the sentence type. Each field thereafter indicates a specific data (position, velocity, time, and so on). Some manufacturers use two characters for the sentence type, followed by the data fields.

After identifying the structure, create the parser function, `parserMMCZ`, using the optional `extractNMEASentence` function, as shown below (you can also use other functions to extract the unparsed data to strings, instead of `extractNMEASentence`).

```
function OutputData = parserMMCZ(unparsedData, MessageID)

  OutputData = struct("MessageID",MessageID,...
  "UTC","NA",...
  "Latitude",NaN,...
  "Longitude",NaN,...
  "NavigationSatellites",NaN,...
  "Status",uint8(1));

  [isValid, splitString] = extractNMEASentence(unparsedData, MessageID);

if(isValid)
    OutputData.MessageID = splitString(1);
    temp = char(splitString(2));
    utcOutput = [temp(1:2), ':', temp(3:4), ':', temp(5:6)];
    OutputData.UTC = utcOutput;
    OutputData.Latitude = str2double(splitString{3});
    OutputData.Longitude = str2double(splitString{5});
    OutputData.NavigationSatellites = str2double(splitString{7});
    OutputData.Status = uint8(0);
  end
end
```

Save `parserMMCZ.m` in the MATLAB path.

2 Extract Data from Manufacturer-specific Sentence

Create an `nmeaParser` System Object by using the `CustomSentence` name-value pair and specifying the message ID as "MMCZ" and the function as "parserMMCZ" (created in the previous step).

```
pnmea = nmeaParser("CustomSentence",{["MMCZ","parserMMCZ"]});
```

Provide an MMC sentence obtained from the device as the input and extract data:

```
unparsedMMCLine='$PMMCZ,225444,4917.24,N,00046.34,E,3,DR*7C';
mmcData = pnmea(unparsedMMCLine)

mmcData =

  struct with fields:

              MessageID: "MMCZ"
                    UTC: '22:54:44'
               Latitude: 4.9172e+03
              Longitude: 46.3400
    NavigationSatellites: 3
                 Status: 0
```

More About

Status Field

The status field displayed along with the extracted values in each output structure can be used to determine the parsing status:

- `Status: 0` — Sentence is valid (checksum validation is successful and the extracted data is as per the requested Message ID)
- `Status: 1` — Checksum of the sentence to be parsed is invalid
- `Status: 2` — The requested sentence is not found in the input data

Note If a value is not available in the input sentence, the corresponding output value is displayed as “NA” for string values and "NaN" for numeric values.

RMC Sentences

Name of field displayed in the output structure after parsing is complete	Description
TalkerID	Identify the type of device that transmits data. For example, for a GPS receiver, the TalkerID is `GP` Data type: `string`
MessageID	Type of NMEA message – `RMC` Data type: `string`
FixStatus	Possible values: • `A` – Data is valid • `V` – Navigation receiver warning Data type: `string`
Latitude	Latitude in degrees. North is considered positive. Data type: `double`

Name of field displayed in the output structure after parsing is complete	Description
Longitude	Longitude in degrees. East is considered positive. Data type: `double`
GroundSpeed	Speed over ground in meters per second (m/s) Data type: `double`
TrueCourseAngle	Course over ground in degrees. Data type: `double`
UTCDateTime	UTC date and time Data type: `datetime`
MagneticVariation	Magnetic variation value. Direction W is considered as negative Data type: `double`
ModeIndicator	Possible values: • `N` – No fix • `E` – Estimated/Dead reckoning fix • `A` – Autonomous GNSS fix • `D` – Differential GNSS fix • `F` – RTK float • `M` – Manual input mode • `P` – Precision mode Data type: `string`
NavigationStatus	Possible values: • `S` – Safe • `C` – Caution • `U` = Unsafe • `V` = Navigational Status not valid Data type: `string`

GGA Sentences

Name of field displayed in the output structure after parsing is complete	Description
TalkerID	Identify the type of device that transmits data. For example, for a GPS receiver, the TalkerID is `GP` Data type: `string`

Name of field displayed in the output structure after parsing is complete	Description
MessageID	Type of NMEA message – `GGA` Data type: `string`
UTCTime	UTC Time (`hhmmss.ss`) Data type: `datetime`
Latitude	Latitude in degrees. North is considered positive. Data type: `double`
Longitude	Longitude in degrees. East is considered positive. Data type: `double`
QualityIndicator	Possible values: • `0` – No fix • `1` – Fix Valid • `2` – Differential GPS, SPS mode fix • `4` – RTK fix • `5` – RTK float • `6` – Estimated/Dead reckoning fix • `7` – Manual input mode • `8` – Simulator mode Data type: `uint8`
NumSatellitesInUse	Number of satellites used. This could be different from number of satellites in view. Data type: `uint8`
HDOP	Horizontal dilution of precision Data type: `double`
Altitude	Altitude above mean sea level in meters Data type: `double`
GeoidSeparation	Difference between ellipsoid and mean sea level in meters Data type: `double`
AgeOfDifferentialData	Age of differential corrections Data type: `double`
DifferentialReferenceStationID	ID of station providing differential corrections Data type: `uint16`

GSA Sentences

Name of field displayed in the output structure after parsing is complete	Description
TalkerID	Identify the type of device that transmits data. For example, for a GPS receiver, the TalkerID is `GP` Data type: `string`
MessageID	Type of NMEA message – `GSA` Data type: `string`
Mode	Possible values: • `M` – Manually set to operate in 2-D or 3-D mode • `A` – Automatic switching between 2-D or 3-D mode Data type: `string`
FixType	Possible values: • `1` – No fix • `2` – 2-D fix • `3` – 3-D fix Data type: `uint8`
SatellitesIDNumber	Satellite numbers (array of 12 bytes). Empty fields will be displayed as `Nan`. Data type: `uint8`
PDOP	Position dilution of precision Data type: `double`
VDOP	Vertical dilution of precision Data type: `double`
HDOP	Horizontal Dilution of Precision Data type: `double`
SystemID	NMEA defined GNSS System ID Data type: `uint8`

GSV Sentences

Name of field displayed in the output structure after parsing is complete	Description
TalkerID	Identify the type of device that transmits data. For example, for a GPS receiver, the TalkerID is `GP` Data type: `string`
MessageID	Type of NMEA message – `GSV` Data type: `string`
NumSentences	Total number of sentences. The complete satellite information is available in multiple GSV sentences. This field indicates the total number of `gsvData` structures containing the complete information per update. Data type: `double`
SentenceNumber	Sentence number of the currently parsed GSV line Data type: `double`
SatellitesInView	Total number of satellites in view Data type: `double`
SatelliteID	Satellite ID numbers specified as a row vector of size 1-by-N, where N is the number of satellite information available in one sentence. The maximum allowed valued of N is 4. Data type: `double`
Elevation	Elevation in degrees, specified as a row vector of size 1-by-N, where N is the number of satellite information available in one sentence. The maximum allowed valued of N is 4. The maximum value of Elevation is 90 degrees. Data type: `double`
Azimuth	Azimuth in degrees, specified as a row vector of size 1-by-N, where N is the number of satellite information available in one sentence. The maximum allowed valued of N is 4. The range of Azimuth value is [0-359] degrees. Data type: `double`

Name of field displayed in the output structure after parsing is complete	Description
SNR	Signal-to-noise ratio in dB-Hz, specified as a row vector of size 1-by-N, where N is the number of satellite information available in one sentence. The maximum allowed valued of N is 4. The range of SNR value is [0,99] dB. Data type: `double`
SignalID	Signal ID corresponding to the SatelliteID. This value is displayed only if the sentences conform to NMEA 0183 Standard, Version 4.1. Otherwise, the value displayed is `NaN`. Data type: `double`

The possible values of SignalID and the corresponding Signal Channel are listed in this table.

System	TalkerID displayed in the parsed data	SatelliteID	SignalID	Signal Channel
GPS	GP	• 1 – 32 (for GPS) • 33 – 64 (for SBAS)	0	All signals
			1	L1 C/A
			2	L1 P (Y)
			3	L1 M
			4	L2 P (Y)
			5	L2C-M
			6	L2C-L
			7	L5-I
			8	L5-Q
GLONASS	GL	• 33 – 64 (for SBAS) • 65 – 99 (for GLONASS)	0	All signals
			1	G1 C/A
			2	G1 P
			3	G2 C/A
			4	GLONASS (M) G2 P
GALILEO	GA	• 1 – 36 (for Galileo satellites) • 37 – 64 (for Galileo SBAS)	0	All signals
			1	E5a
			2	E5b
			3	E5a and E5b
			4	E6-A
			5	E6-BC

System	TalkerID displayed in the parsed data	SatelliteID	SignalID	Signal Channel
			6	L1-A
			7	L1-BC

ZDA Sentences

Name of field displayed in the output structure after parsing is complete	Description
TalkerID	Identify the type of device that transmits data. For example, for a GPS receiver, the TalkerID is `GP` Data type: `string`
MessageID	Type of NMEA message – `ZDA` Data type: `string`
UTCTime	UTC Time Data type: `datetime`
UTCDay	UTC Day Data type: `uint8`
UTCMonth	UTC Month Data type: `uint8`
UTCYear	UTC Year Data type: `uint8`
LocalZoneHours	Local zone hours ranging from `00` to `+/-13` Data type: `int8`
LocalZoneMinutes	Local zone minutes ranging from `00` to `59` Data type: `uint8`

GLL Sentences

Name of field displayed in the output structure after parsing is complete	Description
TalkerID	Identify the type of device that transmits data. For example, for a GPS receiver, the TalkerID is `GP` Data type: `string`

Name of field displayed in the output structure after parsing is complete	Description
MessageID	Type of NMEA message – `GLL` Data type: `string`
Latitude	Latitude in degrees. North is considered positive. Data type: `double`
Longitude	Longitude in degrees. East is considered positive. Data type: `double`
UTCTime	UTC Time Data type: `datetime`
DataValidity	Data validity status: • `A` – Data valid • `V` – Data invalid Data type: `string`
PositioningMode	Possible values: • `N` – Data not Valid • `E` – Estimated/Dead reckoning mode • `A` – Autonomous mode • `D` – Differential mode • `S` – Simulator Mode • `M` – Manual input mode Data type: `string`

VTG Sentences

Name of field displayed in the output structure after parsing is complete	Description
TalkerID	Identify the type of device that transmits data. For example, for a GPS receiver, the TalkerID is `GP` Data type: `string`
MessageID	Type of NMEA message – `VTG` Data type: `string`
TrueCourseAngle	Course over ground (true) in degrees Data type: `double`

Name of field displayed in the output structure after parsing is complete	Description
MagneticCourseAngle	Course over ground (magnetic) in degrees Data type: `double`
GroundSpeed	Speed over ground in meters per second (m/s) Data type: `double`
ModeIndicator	Possible values: • `N` – No fix • `E` – Estimated/Dead reckoning mode • `A` – Autonomous mode • `D` – Differential mode • `M` – Manual input mode • `N` – Data not valid • `P` – Precise • `S` – Simulator mode Data type: `string`

GST Sentences

Name of field displayed in the output structure after parsing is complete	Description
TalkerID	Identify the type of device that transmits data. For example, for a GPS receiver, the TalkerID is `GP` Data type: `string`
MessageID	Type of NMEA message – `GST` Data type: `string`
UTCTime	UTC Time Data type: `datetime`
RMSStdDeviationOfRanges	RMS value of the standard deviation of the ranges in meters. Data type: `double`
StdDeviationSemiMajorAxis	Standard deviation of semi-major axis in meters Data type: `double`
StdDeviationSemiMinorAxis	Standard deviation of semi-minor axis in meters Data type: `double`

Name of field displayed in the output structure after parsing is complete	Description
OrientationSemiMajorAxis	Orientation of semi-major axis, in degrees Data type: `double`
StdDeviationLatitudeError	Standard deviation of latitude error, in meters Data type: `double`
StdDeviationLongitudeError	Standard deviation of longitude error in meters Data type: `double`
StdDeviationAltitudeError	Standard deviation of altitude error in meters Data type: `double`

HDT Sentences

Name of field displayed in the output structure after parsing is complete	Description
TalkerID	Identify the type of device that transmits data. For example, for a GPS receiver, the TalkerID is `GP` Data type: `string`
MessageID	Type of NMEA message – `HDT` Data type: `string`
TrueHeadingAngle	Heading in degrees with respect to true north Data type: `double`

See Also

`skyplot` | `extractNMEASentence`

Topics

“Plot Position of GNSS Receiver Using Live NMEA Data or NMEA Log File”

Introduced in R2020b

navPath

Planned path

Description

The `navPath` object stores paths that are typically created by geometric path planners. Path points are stored as states in an associated state space.

Creation

Syntax

```
path = navPath
path = navPath(space)
path = navPath(space,states)
```

Description

`path = navPath` creates a path object, `path`, using the SE2 state space with default settings.

`path = navPath(space)` creates a path object with state space specified by `space`. The `space` input also sets the value of the StateSpace property.

`path = navPath(space,states)` allows you to initialize the path with state samples given by `states`. Specify `states` as a matrix of state samples. States that are outside of the `StateBounds` of the state `space` object are reduced to the bounds. The `states` input also sets the value of the States property.

Properties

StateSpace — State space for path

`stateSpaceSE2` (default) | state space object

State space for the path, specified as a state space object. Each state in the path is a sample from the specified state space. You can use objects such as `stateSpaceSE2`, `stateSpaceDubins`, or `stateSpaceReedsShepp` as a state space object. You can also customize a state space object using the `nav.StateSpace` object.

Data Types: `object`

States — States of path

`zeros(0,StateSpace.NumStateVariables)` (default) | real-valued *M*-by-*N* matrix

States of the path, specified as a real-valued *M*-by-*N* matrix. *M* is the number of states in the path, and *N* is the dimension of each state. You can only set this property during object creation or using the `append` function.

Data Types: `double`

NumStates — Number of state samples in path

0 (default) | nonnegative integer

Number of state samples in the path, specified as a nonnegative integer. The number is the same as the number of rows of the state matrix specified in the `States` property.

Data Types: `double`

Object Functions

append	Add states to end of path
copy	Create copy of path object
interpolate	Interpolate points along path
pathLength	Length of path

Examples

Create navPath Based on Multiple Waypoints

Create a navPath object based on multiple waypoints in a Dubins space.

```
dubinsSpace = stateSpaceDubins([0 25; 0 25; -pi pi])
```

```
dubinsSpace =
  stateSpaceDubins with properties:

   SE2 Properties
                Name: 'SE2 Dubins'
         StateBounds: [3×2 double]
    NumStateVariables: 3

   Dubins Vehicle Properties
     MinTurningRadius: 1
```

```
pathobj = navPath(dubinsSpace)
```

```
pathobj =
  navPath with properties:

    StateSpace: [1×1 stateSpaceDubins]
        States: [0×3 double]
     NumStates: 0
```

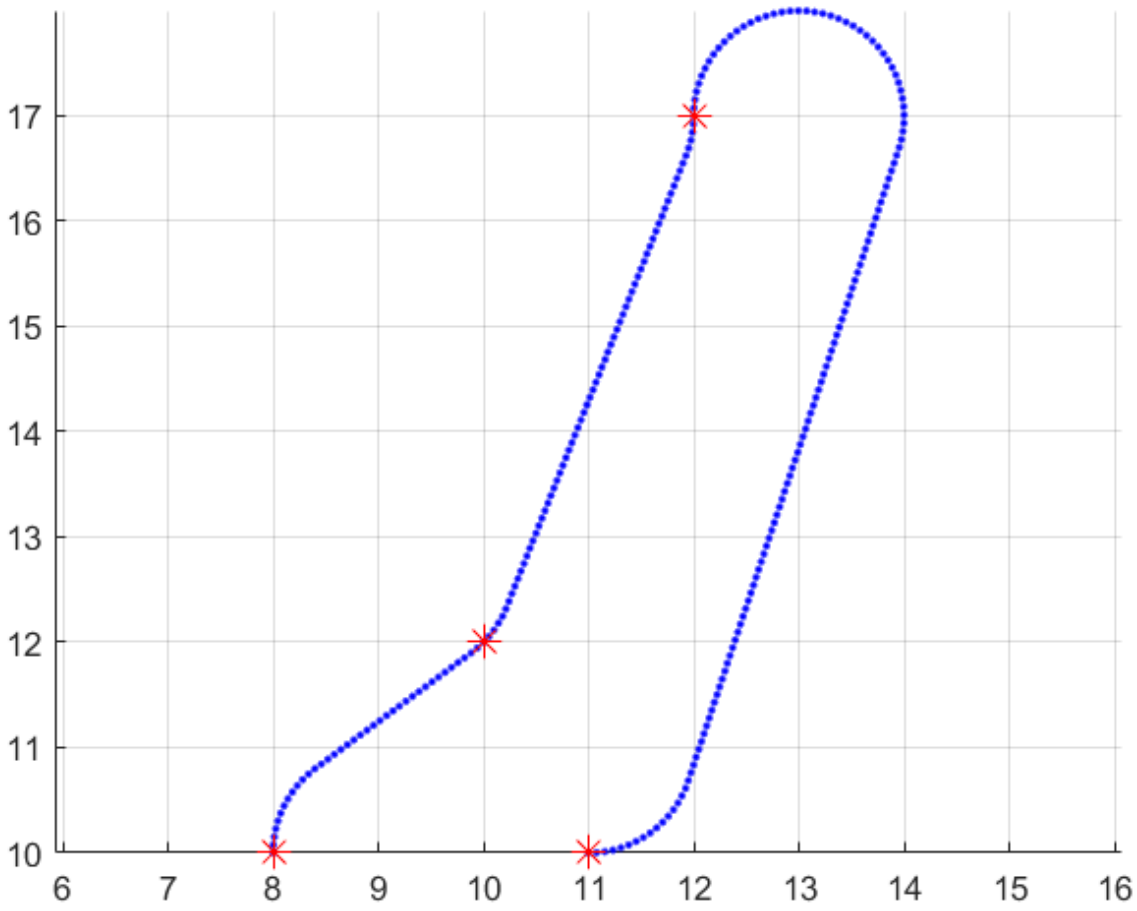
```
waypoints = [...
    8 10 pi/2;
    10 12 pi/4;
    12 17 pi/2;
    11 10 -pi];
append(pathobj, waypoints);
```

Interpolate that path so that it contains exactly 250 points.

```
interpolate(pathobj, 250)
```

Visualize the interpolated path and the original waypoints.

```
figure;
grid on;
axis equal;
hold on;
plot(pathobj.States(:,1), pathobj.States(:,2), ".b");
plot(waypoints(:,1), waypoints(:,2), "*r", "MarkerSize", 10)
```



Calculate length of path.

```
len = pathLength(pathobj);
disp("Path length = " + num2str(len))
```

```
Path length = 19.37
```

See Also

stateSpaceSE2 | stateSpaceReedsShepp | stateSpaceDubins | nav.StateSpace | pathmetrics

Introduced in R2019b

append

Add states to end of path

Syntax

```
append(path,states)
```

Description

`append(path,states)` appends the state samples, `states`, to the end of the `path`.

Examples

Create navPath Based on Multiple Waypoints

Create a navPath object based on multiple waypoints in a Dubins space.

```
dubinsSpace = stateSpaceDubins([0 25; 0 25; -pi pi])
```

```
dubinsSpace =
  stateSpaceDubins with properties:

   SE2 Properties
                Name: 'SE2 Dubins'
         StateBounds: [3×2 double]
    NumStateVariables: 3

   Dubins Vehicle Properties
     MinTurningRadius: 1
```

```
pathobj = navPath(dubinsSpace)
```

```
pathobj =
  navPath with properties:

    StateSpace: [1×1 stateSpaceDubins]
        States: [0×3 double]
     NumStates: 0
```

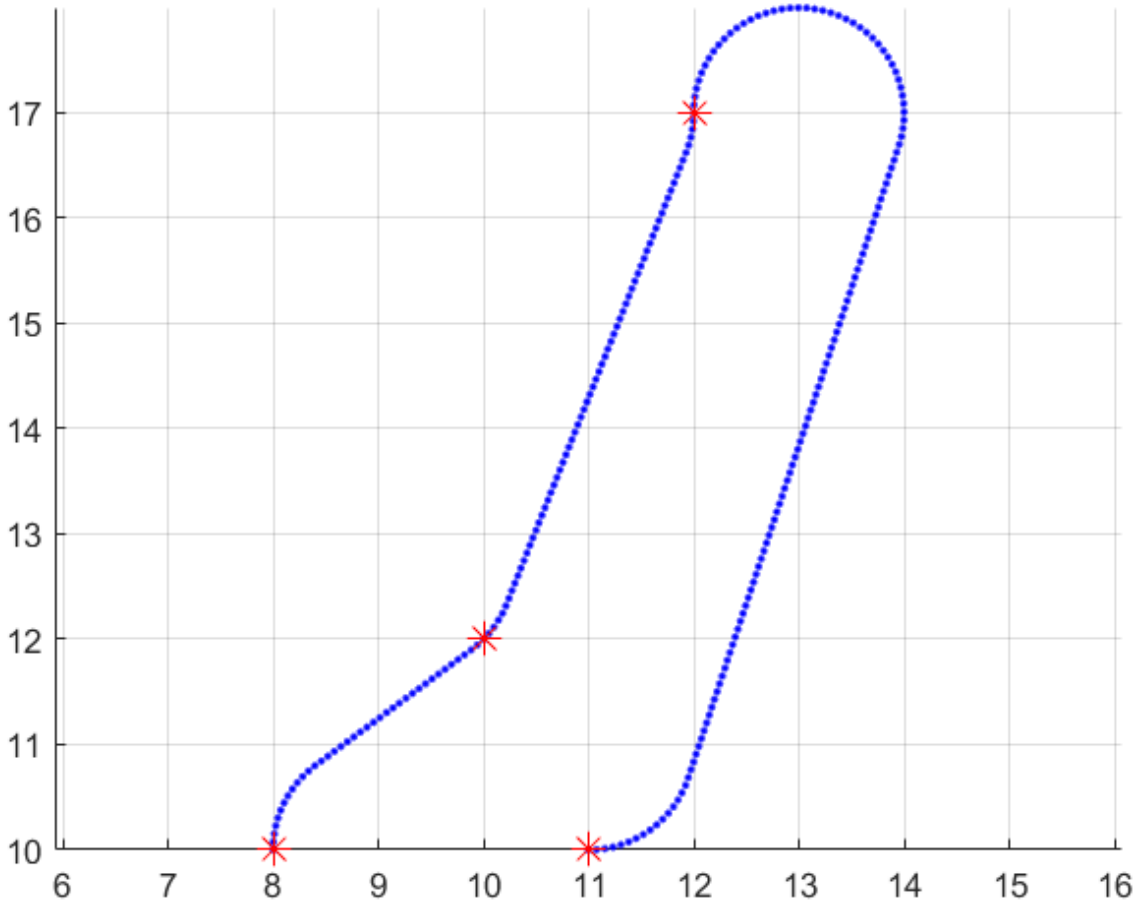
```
waypoints = [...
    8 10 pi/2;
    10 12 pi/4;
    12 17 pi/2;
    11 10 -pi];
append(pathobj, waypoints);
```

Interpolate that path so that it contains exactly 250 points.

```
interpolate(pathobj, 250)
```

Visualize the interpolated path and the original waypoints.

```
figure;
grid on;
axis equal;
hold on;
plot(pathobj.States(:,1), pathobj.States(:,2), ".b");
plot(waypoints(:,1), waypoints(:,2), "*r", "MarkerSize", 10)
```



Calculate length of path.

```
len = pathLength(pathobj);
disp("Path length = " + num2str(len))
```

```
Path length = 19.37
```

Input Arguments

path — path object
navPath object

Path object, specified as a `navPath` object.

Data Types: `object`

`states` — states of path
real-valued *M*-by-*N* matrix

States of the path, specified as a real-valued *M*-by-*N* matrix. *M* is the number of states appended to the path, and *N* is the dimension of each state. The dimension of each state is governed by the state space defined in the `StateSpace` property of `navPath`. States outside of the `StateBounds` of the state space of path are pruned to the bounds.

Example: `[ 0 0 0; 1 1 1]`

Data Types: `double`

See Also
`navPath`

Introduced in R2019b

interpolate

Interpolate points along path

Syntax

```
interpolate(path,numStates)
```

Description

`interpolate(path,numStates)` inserts a number of states in the path and ensures the distribution of all the points in the path to be uniform. The function preserves all the existing states in the path. The value of `numStates` must be greater than or equal to the number of existing states in the path.

Examples

Create navPath Based on Multiple Waypoints

Create a navPath object based on multiple waypoints in a Dubins space.

```
dubinsSpace = stateSpaceDubins([0 25; 0 25; -pi pi])
```

```
dubinsSpace =
  stateSpaceDubins with properties:

   SE2 Properties
                Name: 'SE2 Dubins'
         StateBounds: [3×2 double]
    NumStateVariables: 3

   Dubins Vehicle Properties
     MinTurningRadius: 1
```

```
pathobj = navPath(dubinsSpace)
```

```
pathobj =
  navPath with properties:

    StateSpace: [1×1 stateSpaceDubins]
        States: [0×3 double]
     NumStates: 0
```

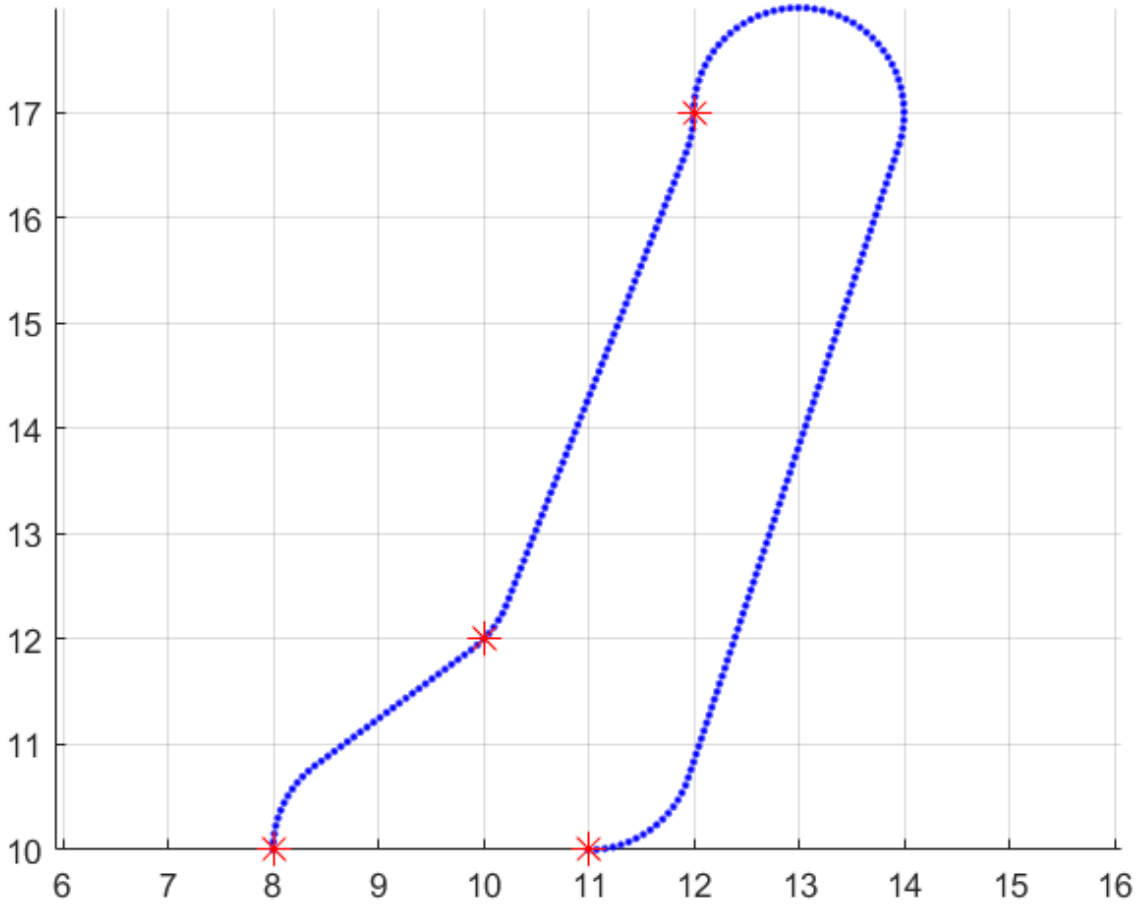
```
waypoints = [...
    8 10 pi/2;
    10 12 pi/4;
    12 17 pi/2;
    11 10 -pi];
append(pathobj, waypoints);
```

Interpolate that path so that it contains exactly 250 points.

```
interpolate(pathobj, 250)
```

Visualize the interpolated path and the original waypoints.

```
figure;
grid on;
axis equal;
hold on;
plot(pathobj.States(:,1), pathobj.States(:,2), ".b");
plot(waypoints(:,1), waypoints(:,2), "*r", "MarkerSize", 10)
```



Calculate length of path.

```
len = pathLength(pathobj);
disp("Path length = " + num2str(len))
```

```
Path length = 19.37
```

Input Arguments

`path` — Path object

`navpath` object

Path object, specified as a `navPath` object.

Data Types: `object`

`numStates` — Number of states

nonnegative integer

Number of states inserted in the path, specified as a nonnegative integer. Its value must be greater than or equal to the number of existing states in the path.

Data Types: `double`

See Also

`navPath`

Introduced in R2019b

pathLength

Length of path

Syntax

```
len = pathLength(path)
```

Description

`len = pathLength(path)` returns the total length of path by summing the distances between every sequential pair of states in the path. The function uses the state space object associated with `path` to calculate the distance between each state pair.

Examples

Create navPath Based on Multiple Waypoints

Create a navPath object based on multiple waypoints in a Dubins space.

```
dubinsSpace = stateSpaceDubins([0 25; 0 25; -pi pi])
```

```
dubinsSpace =
  stateSpaceDubins with properties:

   SE2 Properties
                Name: 'SE2 Dubins'
         StateBounds: [3×2 double]
    NumStateVariables: 3

   Dubins Vehicle Properties
     MinTurningRadius: 1
```

```
pathobj = navPath(dubinsSpace)
```

```
pathobj =
  navPath with properties:

    StateSpace: [1×1 stateSpaceDubins]
        States: [0×3 double]
     NumStates: 0
```

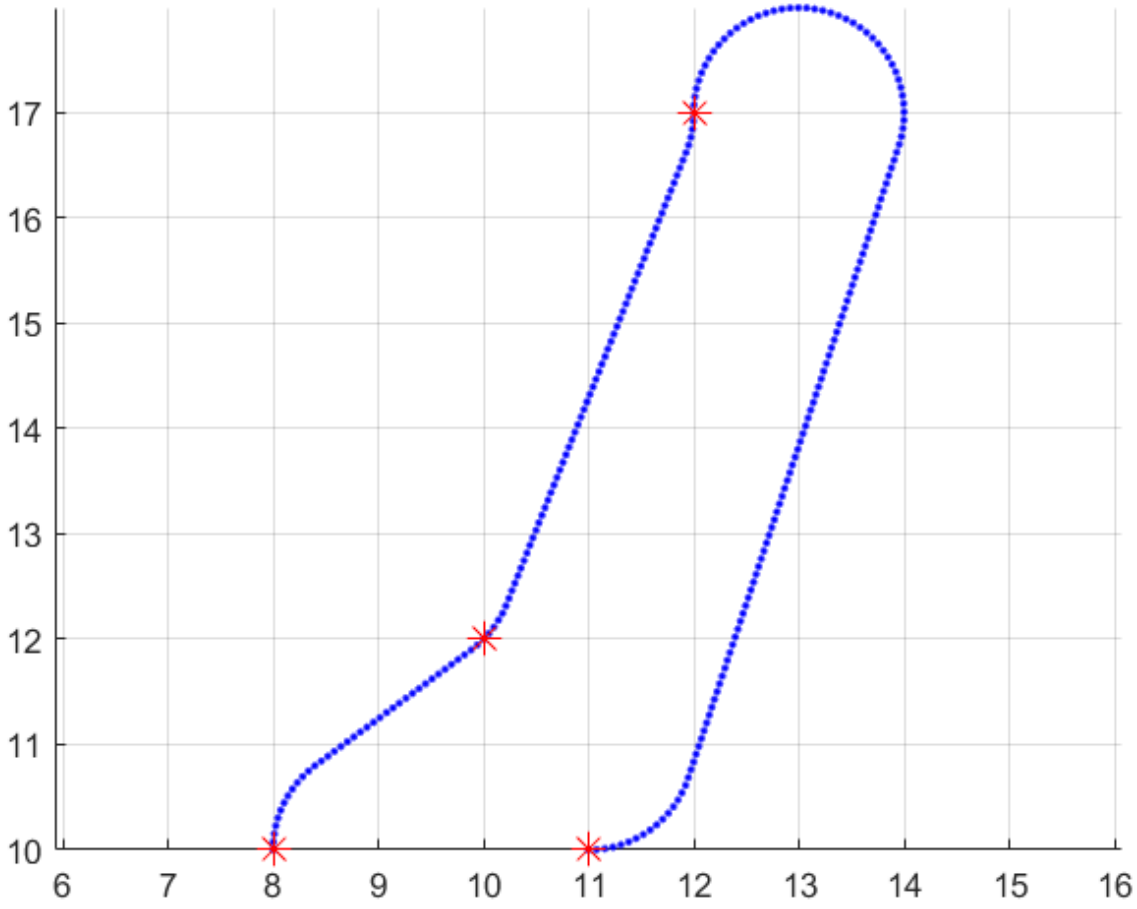
```
waypoints = [...
    8 10 pi/2;
    10 12 pi/4;
    12 17 pi/2;
    11 10 -pi];
append(pathobj, waypoints);
```

Interpolate that path so that it contains exactly 250 points.

```
interpolate(pathobj, 250)
```

Visualize the interpolated path and the original waypoints.

```
figure;
grid on;
axis equal;
hold on;
plot(pathobj.States(:,1), pathobj.States(:,2), ".b");
plot(waypoints(:,1), waypoints(:,2), "*r", "MarkerSize", 10)
```



Calculate length of path.

```
len = pathLength(pathobj);
disp("Path length = " + num2str(len))
```

```
Path length = 19.37
```

Input Arguments

`path` — Path object

`navpath` object

Path object, specified as a `navPath` object.

Data Types: `object`

Output Arguments

`len` — Length of path

nonnegative scalar

Length of the path, returned as a nonnegative scalar.

See Also

`navPath`

Introduced in R2019b

navPathControl

Path representing control-based kinematic trajectory

Description

The `navPathControl` object stores paths that are typically created by control-based path planners like the `plannerControlRRT` object. The `navPathControl` object represents paths as a sequence of states, controls, durations, and targets. This object associates each path with a specific state propagator, which propagates the control commands to determine the resulting states.

This object specifies states and targets in the path in the state space of propagator. Controls are outputs from a controller that are used to update your systems state during propagation. This object applies each control for an associated duration. Controls can be reference signals or direct inputs to an integrator depending on your system design.

Creation

Syntax

```
pathObj = navPathControl(propagator)
pathObj = navPathControl(propagator,states,controls,targets,durations)
```

Description

`pathObj = navPathControl(propagator)` creates a path object with the specified state propagator. The `propagator` argument specifies the `StatePropagator`

`pathObj = navPathControl(propagator,states,controls,targets,durations)` initializes the path with a sequence of specified states, controls, targets, and durations. The input states must have one more row than the other input vectors and matrices.

Properties

`StatePropagator` — State propagator
object of subclass of `nav.StatePropagator`

State propagator, specified as an object of a subclass of `nav.StatePropagator`. For example, the `mobileRobotPropagator` object represents the state space and kinematic control behavior for different mobile robot vehicle models.

Data Types: `double`

`States` — Series of states for path
`[ ]` (default) | *n*-by-*m* matrix

Series of states for the path, specified as an *n*-by-*m* matrix. *n* is the number of points in the path. *m* is the dimension of the state vector.

You can specify this property at object creation by using the `states` argument.

Data Types: `double`

`Controls` — Control input for each state
`[]` (default) | *(n–1)*-by-*m* matrix

Control input for each state, specified as an *(n–1)*-by-*m* matrix. *n* is the number of points in the path. *m* is the dimension of the state vector.

You can specify this property at object creation by using the `controls` argument.

Data Types: `double`

`TargetStates` — Target state for each state in path
`[]` (default) | *(n–1)*-by-*m* matrix

Target state for each state in the path, specified as an *(n–1)*-by-*m* matrix. *n* is the number of points in the path. *m* is the dimension of the state vector.

You can specify this property at object creation by using the `targets` argument.

Data Types: `double`

`Durations` — Duration of each control input
`[]` (default) | *(n–1)*-element vector in seconds

Duration of each control input, specified as an *(n–1)*-element vector in seconds. *n* is the number of points in the path.

You can specify this property at object creation by using the `durations` argument.

Data Types: `double`

`NumStates` — Number of states in path
`0` (default) | nonnegative integer

Number of states in the path, specified as a positive integer.

Data Types: `double`

`NumSegments` — Number of segments between states
`0` (default) | nonnegative integer

Number of segments between states in the path, specified as a positive integer, which must be one less than the number of states.

Data Types: `double`

Object Functions

append	Add states to end of path
interpolate	Interpolate path based on propagator step size
pathDuration	Total elapsed duration of control path

Examples

Plan Kinodynamic Path with Controls for Mobile Robot

Plan control paths for a bicycle kinematic model with the `mobileRobotPropagator` object. Specify a map for the environment, set state bounds, and define a start and goal location. Plan a path using the control-based RRT algorithm, which uses a state propagator for planning motion and the required control commands.

Set State and State Propagator Parameters

Load a ternary map matrix and create an `occupancyMap` object. Create the state propagator using the map. By default, the state propagator uses a bicycle kinematic model.

```
load('exampleMaps','ternaryMap')
map = occupancyMap(ternaryMap,10);

propagator = mobileRobotPropagator(Environment=map); % Bicycle model
```

Set the state bounds on the state space based on the map world limits.

```
propagator.StateSpace.StateBounds(1:2,:) = ...
                    [map.XWorldLimits; map.YWorldLimits];
```

Plan Path

Create the path planner from the state propagator.

```
planner = plannerControlRRT(propagator);
```

Specify the start and goal states.

```
start = [10 15 0];
goal  = [40 30 0];
```

Plan a path between the states. For repeatable results, reset the random number generator before planning. The `plan` function outputs a `navPathControl` object, which contains the states, control commands, and durations.

```
rng("default")
path = plan(planner,start,goal)
```

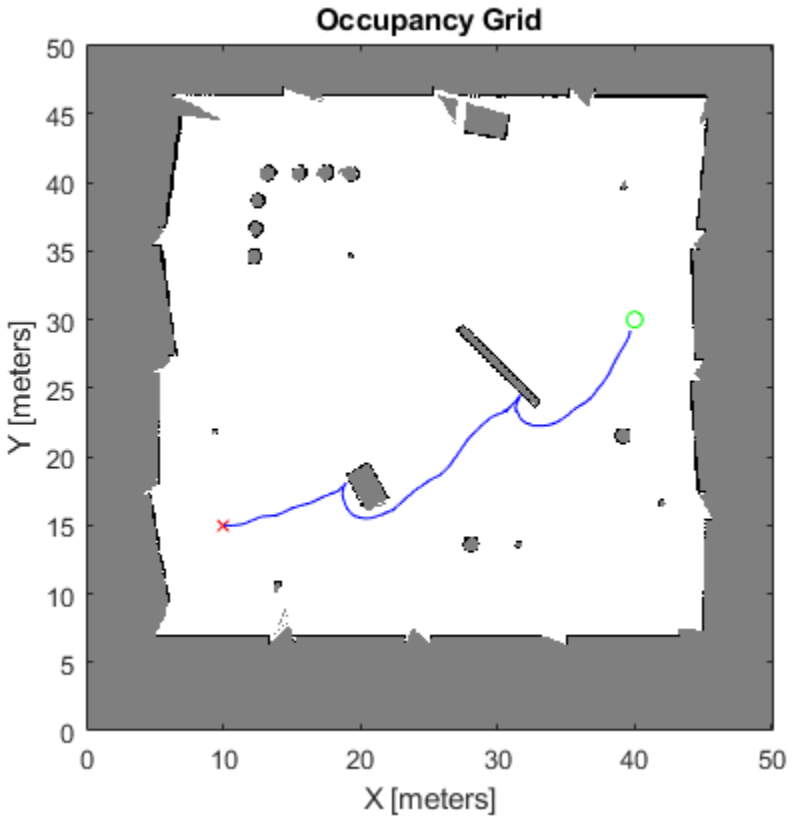
```
path =
  navPathControl with properties:

    StatePropagator: [1x1 mobileRobotPropagator]
             States: [192x3 double]
           Controls: [191x2 double]
          Durations: [191x1 double]
       TargetStates: [191x3 double]
          NumStates: 192
        NumSegments: 191
```

Visualize Results

Visualize the map and plot the path states.

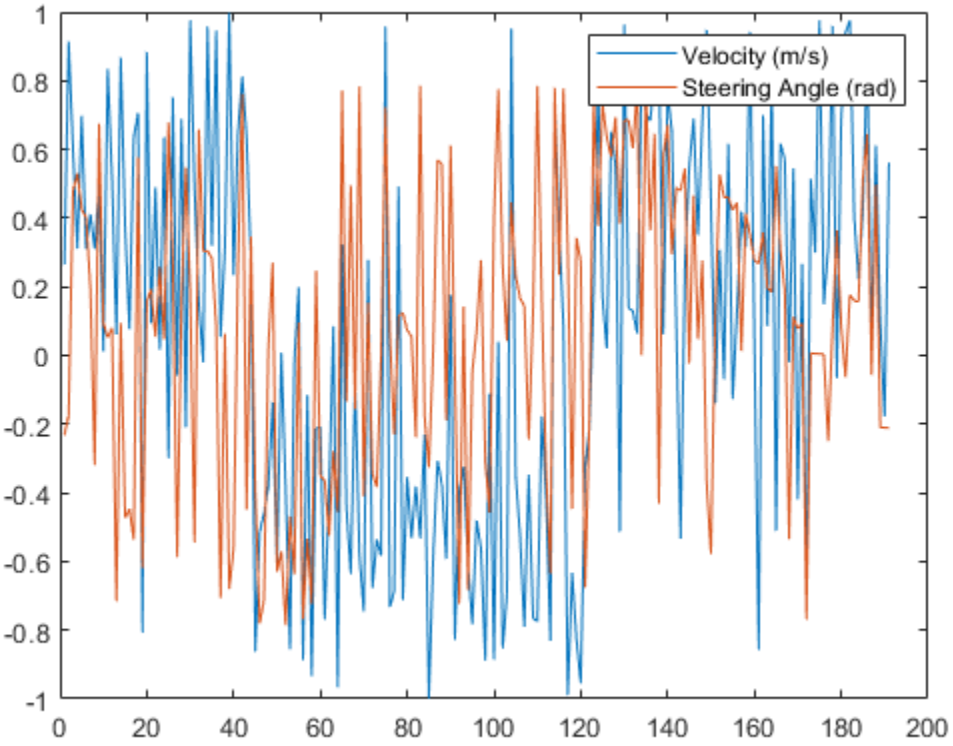
```
show(map)
hold on
```

```
plot(start(1),start(2),"rx")
plot(goal(1),goal(2),"go")
plot(path.States(:,1),path.States(:,2),"b")
hold off
```



Display the `[v psi]` control inputs of forward velocity and steering angle.

```
plot(path.Controls)
ylim([-1 1])
legend(["Velocity (m/s)","Steering Angle (rad)"])
```



See Also

Objects
`navPath` | `mobileRobotPropagator`

Functions
`append` | `interpolate` | `pathDuration`

Introduced in R2021b

append

Add states to end of path

Syntax

```
append(pathObj,states,controls,durations,targets)
```

Description

`append(pathObj,states,controls,durations,targets)` adds a sequence of states, controls, durations and targets to the end of the path. If the path is empty, the `states` input must have one more row than the other input vectors and matrices. If the path contains points already, the function applies the first control to the last state in the current path.

Input Arguments

`pathObj` — Control path

`navControlPath`

Control path, specified as a `navPathControl` object.

Data Types: `double`

`states` — Series of states for path

n-by-*m* matrix

Series of states for the path, specified as an *n*-by-*m* matrix. *n* is the number of points to add to the path. *m* is the dimension of the state vector.

Note If the path object is empty, the `states` input should be an *(n+1)*-by-*m* matrix.

Data Types: `double`

`controls` — Control input for each state

n-by-*m* matrix

Control input for each state, specified as an *n*-by-*m* matrix. *n* is the number of points to add to the path. *m* is the dimension of the state vector.

Data Types: `double`

`durations` — Duration of each control input

n-element vector in seconds

Duration of each control input, specified as an *n*-element vector in seconds. *n* is the number of points to add to the path.

Data Types: `double`

`targets` — Target state for each state in path
n-element vector in seconds

Target state for each state in the path,specified as an *n*-by-*m* matrix. *n* is the number of points to add to the path. *m* is the dimension of the state vector.

Data Types: `double`

See Also

Objects
`navPathControl` | `navPath` | `mobileRobotPropagator`

Functions
`interpolate` | `pathDuration`

Introduced in R2021b

interpolate

Interpolate path based on propagator step size

Syntax

```
interpolate(pathObj)
```

Description

`interpolate(pathObj)` evaluates the path based on the `ControlStepSize` property of `pathObj`, and adds all intermediate points to the path.

Examples

Create navPath Based on Multiple Waypoints

Create a navPath object based on multiple waypoints in a Dubins space.

```
dubinsSpace = stateSpaceDubins([0 25; 0 25; -pi pi])
```

```
dubinsSpace =
  stateSpaceDubins with properties:

   SE2 Properties
                Name: 'SE2 Dubins'
         StateBounds: [3×2 double]
    NumStateVariables: 3

   Dubins Vehicle Properties
     MinTurningRadius: 1
```

```
pathobj = navPath(dubinsSpace)
```

```
pathobj =
  navPath with properties:

    StateSpace: [1×1 stateSpaceDubins]
        States: [0×3 double]
     NumStates: 0
```

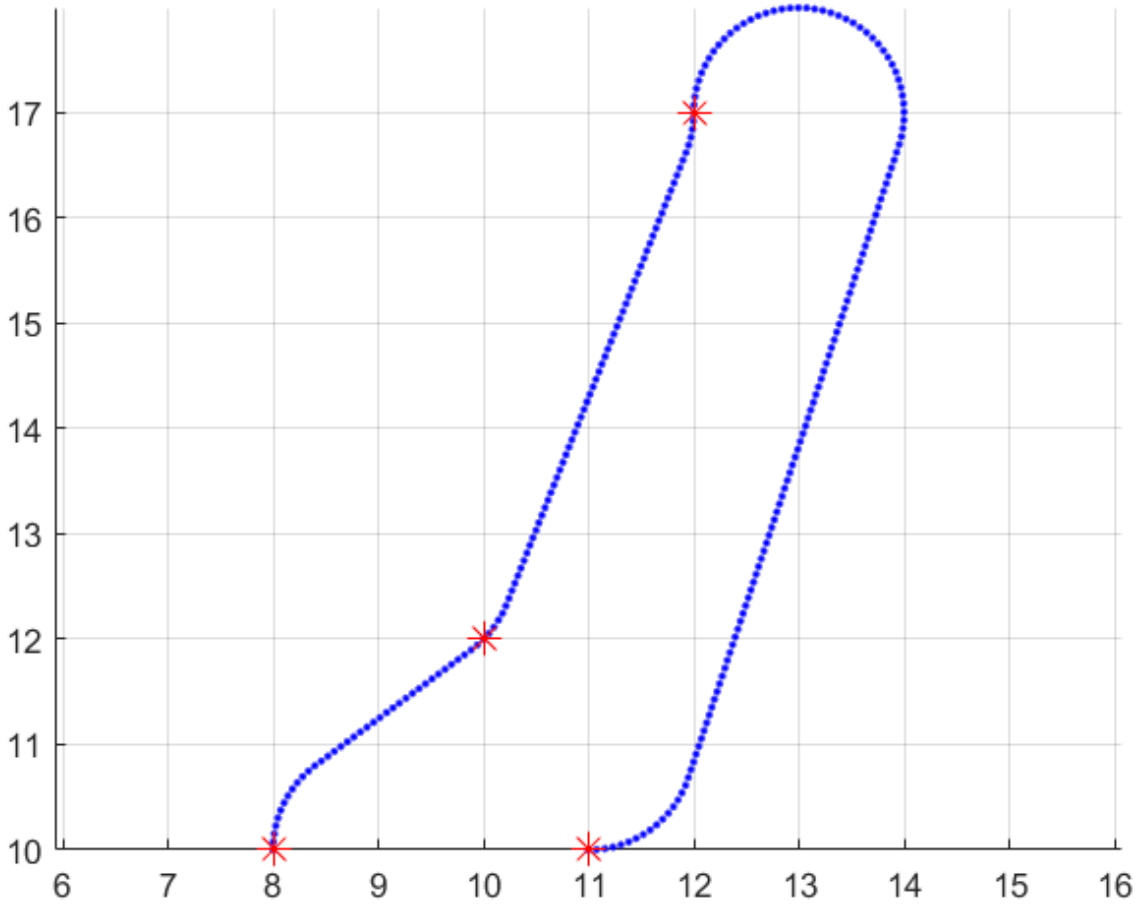
```
waypoints = [...
    8 10 pi/2;
    10 12 pi/4;
    12 17 pi/2;
    11 10 -pi];
append(pathobj, waypoints);
```

Interpolate that path so that it contains exactly 250 points.

```
interpolate(pathobj, 250)
```

Visualize the interpolated path and the original waypoints.

```
figure;
grid on;
axis equal;
hold on;
plot(pathobj.States(:,1), pathobj.States(:,2), ".b");
plot(waypoints(:,1), waypoints(:,2), "*r", "MarkerSize", 10)
```



Calculate length of path.

```
len = pathLength(pathobj);
disp("Path length = " + num2str(len))
```

```
Path length = 19.37
```

Input Arguments

`pathObj` — Control path object
`navControlPath` object

Control path, specified as a `navPathControl` object.

Data Types: `double`

See Also

Objects
`navPathControl` | `navPath` | `mobileRobotPropagator`

Functions
`append` | `pathDuration`

Introduced in R2021b

pathDuration

Total elapsed duration of control path

Syntax

```
totalTime = pathDuration(pathObj)
```

Description

`totalTime = pathDuration(pathObj)` returns the total elapsed duration of the control path.

Input Arguments

`pathObj` — Control path object

`navControlPath` object

Control path, specified as a `navPathControl` object.

Data Types: `double`

Output Arguments

`totalTime` — Total duration of control path

positive scalar

Total duration of the control path, returned as a positive scalar in seconds.

Data Types: `double`

See Also

Objects

`navPathControl` | `navPath` | `mobileRobotPropagator`

Functions

`append` | `interpolate`

Introduced in R2021b

nav.StatePropagator class

Package: nav

State propagator for control-based planning

Description

The `nav.StatePropagator` class is an interface for all state propagators used with the kinodynamic path planners derived from `nav.StateSpace`. Derive from this class if you are defining a propagator for your custom motion model or control system. For a concrete implementation for mobile robots, see the `mobileRobotPropagator` object.

This class generates controls, propagates state, and estimates cost or distance between states. Controlled systems utilize two main equations for two steps:

Initial Control

- `[u(i),steps] = controlFcn(q(i-1),u(i-1),qTgt)` generates the next control command from the previous state, control input, and target state.

Propagate the System

- `q(i) = q(i-1) + integrate(q,u(i),steps)` propagates the state `q(i-1)` using the generated command `u(i)` for the specified number of steps. The propagator uses a fixed step size, specified by the ControlStepSize property.

The `nav.StatePropagator.propagateWhileValid` method defines the integration and optionally, the control behavior, and also checks whether the generated states are valid within the state space. To skip state validation, use the `nav.StatePropagator.propagate` method.

When creating an instance of class, specify the StateSpace property, which defines the number of state variables, state bounds, and sampling behavior of the propagator. Also, specify the control limits on creation, which determines the value of the NumControlOutput property.

When you change properties, such as the state space, you may need to change the internal behavior of the propagator. To do this, implement the changes in the `nav.StatePropagator.setup` method and call `setup(obj)` before calling other methods again.

The `nav.StatePropagator` class is a `handle` class.

Creation

Syntax

```
propagatorObj = nav.StatePropagator(stateSpace,stepSize,numControlOutput)
```

Description

`propagatorObj = nav.StatePropagator(stateSpace,stepSize,numControlOutput)` creates a state propagator object for propagating a kinodynamic system.

Properties

Public Properties

`StateSpace` — State space for sampling during planning

object of subclass of `nav.StateSpace`

State space for sampling during planning, specified as an object of a subclass of `nav.StateSpace`.

The state space is responsible for representing configuration space of a system. The class should include all state information related to the propagated system. Systems employing multi-layer cascade controllers can append persistent low-level control information directly to the state vector, whereas the state propagator directly manages top-level control commands.

`ControlStepSize` — Duration of each control command

`0.1` (default) | positive scalar

Duration of each control command, specified as a positive scalar.

Protected Properties

`NumControlOutput` — Number of variables in control command

positive integer

Number of variables in the control command, specified as a positive integer.

Attributes:

`GetAccess`	`protected`
`SetAccess`	`immutable`

Methods

Public Methods

nav.StatePropagator.distance	Estimate cost of propagating to target state
nav.StatePropagator.propagate	Propagate system without validation
nav.StatePropagator.propagateWhileValid	Propagate system and return valid motion
nav.StatePropagator.sampleControl	Generate control command and step count
nav.StatePropagator.setup	Estimate cost of propagating to target state

See Also

Classes

`nav.StateSpace` | `nav.StateValidator`

Objects

`stateSpaceSE2` | `stateSpaceDubins` | `stateSpaceReedsShepp` | `validatorOccupancyMap` | `validatorVehicleCostmap`

Functions

`nav.StatePropagator.distance` | `nav.StatePropagator.propagate` | `nav.StatePropagator.propagateWhileValid` | `nav.StatePropagator.sampleControl` | `nav.StatePropagator.setup`

Introduced in R2021b

distance

Class: `nav.StatePropagator`
Package: `nav`

Estimate cost of propagating to target state

Syntax

```
h = distance(q1,q2)
```

Description

`h = distance(q1,q2)` estimates the cost of propagating from an initial set of states `q1` to final states `q2`. Each row in `q1` and `q2` represents a specific state in the system and the number of columns matches the number of state variables. The function outputs an *n*-element vector `h` for each `q1`-`q2` pair.

Input Arguments

q1 — Initial states

n-by-*s* matrix

Initial states, specified as an *n*-by-*s* matrix.

q2 — Final states

n-by-*s* matrix

Final states, specified as an *n*-by-*s* matrix.

Output Arguments

h — Cost values

n-element vector

Cost values, returned as an *n*-element vector, where *n* is the number of `q1`-`q2` pairs.

Cost values returned by this function are typically used to find the nearest neighbor for sampled target states when planning.

Attributes

`Abstract`	`true`

To learn about attributes of methods, see Method Attributes.

See Also

Classes
nav.StatePropagator | nav.StateSpace | nav.StateValidator

Functions
nav.StatePropagator.propagate | nav.StatePropagator.propagateWhileValid | nav.StatePropagator.sampleControl | nav.StatePropagator.setup

Introduced in R2021b

propagate

Class: `nav.StatePropagator`
Package: `nav`

Propagate system without validation

Syntax

```
[q,u,steps] = propagate(spObj,q0,u0,qTgt,maxSteps)
```

Description

`[q,u,steps] = propagate(spObj,q0,u0,qTgt,maxSteps)` iteratively propagates the system from the current state `q0` towards a target state `qTgt` with an initial control input `u0`for a maximum number of steps `maxSteps`. All propagations are validated and the function returns system information between `q0` and the last valid state.

At the end of each propagation step *i*, the system returns:

- `q(i,:)` — Current state of the system
- `u(i,:)` — Control input for step *i+1*
- `steps(i)` — Number of steps between *i-1* and *i*

Input Arguments

`spObj` — State propagator object

handle from child class of `nav.StatePropagator`

State propagator object, specified as a handle from a child class of `nav.StatePropagator`.

`q0` — Initial state

s-element vector

Initial state of the system, specified as an *s*-element vector, where *s* is the number of state variables in the state space.

`u0` — Initial control on the initial state

c-element vector

Initial control input, specified as an *c*-element vector, where *c* is the number of control inputs.

`qTgt` — Target state

s-element vector

Target state of the system, specified as an *s*-element vector, where *s* is the number of state variables in the state space.

`maxSteps` — Maximum number of steps

positive scalar

Maximum number of steps, specified as a positive scalar.

Output Arguments

q — Propagated states

n-by-*s* matrix

Initial state of the system, specified as an *s*-element vector, where *s* is the number of state variables in the state space.

u — Control inputs for propagating states

n-by-*c* matrix

Control inputs for propagating states, specified as an *s*-element vector, where *c* is the number of control inputs.

steps — Number of steps between each state and control input

n-element vector of positive integers

Number of steps between each state and control input, specified as an *n*-element vector of positive integers.

Attributes

`Abstract`	`true`

To learn about attributes of methods, see Method Attributes.

See Also

Classes

`nav.StatePropagator` | `nav.StateSpace` | `nav.StateValidator`

Functions

`nav.StatePropagator.distance` | `nav.StatePropagator.propagateWhileValid` | `nav.StatePropagator.sampleControl` | `nav.StatePropagator.setup`

Introduced in R2021b

propagateWhileValid

Class: `nav.StatePropagator`
Package: `nav`

Propagate system and return valid motion

Syntax

```
[q,u,steps] = propagateWhileValid(spObj,q0,u0,qTgt,maxSteps)
```

Description

`[q,u,steps] = propagateWhileValid(spObj,q0,u0,qTgt,maxSteps)` iteratively propagates the system from the current state `q0` towards a target state `qTgt` with an initial control input `u0`for a maximum number of steps `maxSteps`. All propagations are validated and the function returns system information between `q0` and the last valid state.

At the end of each propagation step *i*, the system returns:

- `q(i,:)` — Current state of the system
- `u(i,:)` — Control input for step *i+1*
- `steps(i)` — Number of steps between *i-1* and *i*

Input Arguments

`spObj` — State propagator object

handle from child class of `nav.StatePropagator`

State propagator object, specified as a handle from a child class of `nav.StatePropagator`.

`q0` — Initial state

s-element vector

Initial state of the system, specified as an *s*-element vector, where *s* is the number of state variables in the state space.

`u0` — Initial control on the initial state

c-element vector

Initial control input, specified as an *c*-element vector, where *c* is the number of control inputs.

`qTgt` — Target state

s-element vector

Target state of the system, specified as an *s*-element vector, where *s* is the number of state variables in the state space.

`maxSteps` — Maximum number of steps

positive scalar

Maximum number of steps, specified as a positive scalar.

Output Arguments

q — Propagated states

n-by-*s* matrix

Initial state of the system, specified as an *s*-element vector, where *s* is the number of state variables in the state space.

u — Control inputs for propagating states

n-by-*c* matrix

Control inputs for propagating states, specified as an *s*-element vector, where *c* is the number of control inputs.

steps — Number of steps between each state and control input

n-element vector of positive integers

Number of steps between each state and control input, specified as an *n*-element vector of positive integers.

Attributes

`Abstract`	`true`

To learn about attributes of methods, see Method Attributes.

See Also

Classes

`nav.StatePropagator` | `nav.StateSpace` | `nav.StateValidator`

Functions

`nav.StatePropagator.distance` | `nav.StatePropagator.propagate` | `nav.StatePropagator.sampleControl` | `nav.StatePropagator.setup`

Introduced in R2021b

sampleControl

Class: `nav.StatePropagator`
Package: `nav`

Generate control command and step count

Syntax

```
[u,steps] = sampleControl(spObj,q0,u0,qTgt)
```

Description

`[u,steps] = sampleControl(spObj,q0,u0,qTgt)` generates a series of control commands and number of steps to move from the current state `q0` with control command `u0` towards the target state `qTgt`

Input Arguments

`spObj` — State propagator object

handle from child class of `nav.StatePropagator`

State propagator object, specified as a handle from a child class of `nav.StatePropagator`.

`q0` — Initial state

s-element vector

Initial state of the system, specified as an *s*-element vector, where *s* is the number of state variables in the state space.

`u0` — Initial control on the initial state

c-element vector

Initial control input, specified as an *c*-element vector, where *c* is the number of control inputs.

`qTgt` — Target state

s-element vector

Target state of the system, specified as an *s*-element vector, where *s* is the number of state variables in the state space.

Output Arguments

`u` — Control inputs for propagating states

n-by-*c* matrix

Control inputs for propagating states, specified as an *s*-element vector, where *c* is the number of control inputs.

`steps` — Number of steps between each state and control input

n-element vector of positive integers

Number of steps between each state and control input, specified as an *n*-element vector of positive integers.

Attributes

`Abstract`	`true`

To learn about attributes of methods, see Method Attributes.

See Also

Classes
`nav.StatePropagator` | `nav.StateSpace` | `nav.StateValidator`

Functions
`nav.StatePropagator.distance` | `nav.StatePropagator.propagate` | `nav.StatePropagator.propagateWhileValid` | `nav.StatePropagator.setup`

Introduced in R2021b

setup

Class: nav.StatePropagator
Package: nav

Estimate cost of propagating to target state

Syntax

```
setup(mobileProp)
```

Description

`setup(mobileProp)` sets up the `nav.StatePropagator` object based on the specified parameters. If you change properties on the object, call this method before you sample controls, propagate the system, or calculate distances.

Input Arguments

`spObj` — State propagator object

handle from child class of `nav.StatePropagator`

State propagator object, specified as a handle from a child class of `nav.StatePropagator`.

See Also

Classes

`nav.StatePropagator` | `nav.StateSpace` | `nav.StateValidator`

Functions

`nav.StatePropagator.distance` | `nav.StatePropagator.propagate` | `nav.StatePropagator.propagateWhileValid` | `nav.StatePropagator.sampleControl` | `nav.StatePropagator.setup`

Introduced in R2021b

nav.StateSpace class

Package: nav

Create state space for path planning

Description

The nav.StateSpace class is an interface for state spaces used for path planning. Derive from this class if you want to define your own state space. This representation allows for sampling, interpolation, and calculating distances between spaces in the state space.

To create a sample template for generating your own state space class, call `createPlanningTemplate`. For specific implementations of the state validator class for general application, see **State Spaces** in “Motion Planning”.

The `nav.StateSpace` class is a `handle` class.

Class Attributes

`Abstract`	`true`

For information on class attributes, see “Class Attributes”.

Creation

Syntax

```
ssObj = nav.StateSpace(Name,NumStateVariables,StateBounds)
```

Description

`ssObj = nav.StateSpace(Name,NumStateVariables,StateBounds)` creates a state space object with a given name, number of state variables, and state bounds. This constructor can only be called from a derived class. Create your own class definition using `createPlanningTemplate`.

Properties

Public Properties

`NumStateVariables` — Number of variables in state space
positive numeric scalar

Number of variables in the state space, specified as a positive numeric scalar. This property is the dimension of the state space.

Example: 3

Attributes:

SetAccess	immutable

StateBounds — Minimum and maximum bounds of state variables
[min max] | *n*-by-2 matrix

Minimum and maximum bounds of state variables, specified as a [min max] *n*-by-2 matrix. This property depends on NumStateVariables, where *n* is the number of state variables. When specifying on construction, use the Bounds input.

Example: [-10 10; -10 10; -pi pi]

Attributes:

GetAccess	public
SetAccess	protected
Dependent	true

Data Types: double

Protected Properties

Name — Name of state space object
string scalar | character vector

Name of the state space object, specified as a string scalar or character vector.

Example: "customSE2StateSpace"

Attributes:

GetAccess	protected
SetAccess	protected

Methods

Public Methods

copy	Copy array of handle objects
distance	Distance between two states
enforceStateBounds	Limit state to state bounds
interpolate	Interpolate between states
sampleGaussian	Sample state using Gaussian distribution
sampleUniform	Sample state using uniform distribution

Examples

Create Custom State Space for Path Planning

This example shows how to use the createPlanningTemplate function to generate a template for customizing your own state space definition and sampler to use with path planning algorithms. A simple implementation is provided with the template.

Call the create template function. This function generates a class definition file for you to modify for your own implementation.

`createPlanningTemplate`

Class and Property Definition

The first part of the template specifies the class definition and any properties for the class. Derive from the `nav.StateSpace` class. For this example, create a property for the uniform and normal distributions. You can specify any additional user-defined properties here.

```
classdef MyCustomStateSpace < nav.StateSpace & ...
        matlabshared.planning.internal.EnforceScalarHandle
     properties
        UniformDistribution
        NormalDistribution
        % Specify additional properties here
end
```

Save your custom state space class and ensure your file name matches the class name.

Class Constructor

Use the constructor to set the name of the state space, the number of state variables, and define its boundaries. Alternatively, you can add input arguments to the function and pass the variables in when you create an object.

- For each state variable, define the `[min max]` values for the state bounds.
- Call the constructor of the base class.
- For this example, you specify the normal and uniform distribution property values using predefined `NormalDistribution` and `UniformDistribution` classes.
- Specify any other user-defined property values here.

```
methods
    function obj = MyCustomStateSpace
        spaceName = "MyCustomStateSpace";
        numStateVariables = 3;
        stateBounds = [-100 100;  % [min max]
                       -100 100;
                       -100 100];

        obj@nav.StateSpace(spaceName, numStateVariables, stateBounds);

        obj.NormalDistribution = matlabshared.tracking.internal.NormalDistribution(numStateVariab
        obj.UniformDistribution = matlabshared.tracking.internal.UniformDistribution(numStateVari
        % User-defined property values here
    end
```

Copy Semantics

Specify the `copy` method definition. Copy all the values of your user-defined variables into a new object, so `copyObj` is a deep copy. The default behavior given in this example creates a new copy of the object with the same name, state bounds, and distributions.

```
function copyObj = copy(obj)
    copyObj = feval(class(obj));
    copyObj.StateBounds = obj.StateBounds;
    copyObj.UniformDistribution = obj.UniformDistribution.copy;
    copyObj.NormalDistribution = obj.NormalDistribution.copy;
end
```

Enforce State Bounds

Specify how to ensure states are always within the state bounds. For this example, the state values get saturated at the minimum or maximum values for the state bounds.

```
function boundedState = enforceStateBounds(obj, state)
    nav.internal.validation.validateStateMatrix(state, nan, obj.NumStateVariables, "enforceStateB
    boundedState = state;
    boundedState = min(max(boundedState, obj.StateBounds(:,1)'), ...
        obj.StateBounds(:,2)');

end
```

Sample Uniformly

Specify the behavior for sampling across a uniform distribution. support multiple syntaxes to constrain the uniform distribution to a nearby state within a certain distance and sample multiple states.

```
STATE = sampleUniform(OBJ)
STATE = sampleUniform(OBJ,NUMSAMPLES)
STATE = sampleUniform(OBJ,NEARSTATE,DIST)
STATE = sampleUniform(OBJ,NEARSTATE,DIST,NUMSAMPLES)
```

For this example, use a validation function to process a `varargin` input that handles the varying input arguments.

```
 function state = sampleUniform(obj, varargin)
    narginchk(1,4);
    [numSamples, stateBounds] = obj.validateSampleUniformInput(varargin{:});

    obj.UniformDistribution.RandomVariableLimits = stateBounds;
    state = obj.UniformDistribution.sample(numSamples);
 end
```

Sample from Gaussian Distribution

Specify the behavior for sampling across a Gaussian distribution. Support multiple syntaxes for sampling a single state or multiple states.

```
STATE = sampleGaussian(OBJ, MEANSTATE, STDDEV)
STATE = sampleGaussian(OBJ, MEANSTATE, STDDEV, NUMSAMPLES)


function state = sampleGaussian(obj, meanState, stdDev, varargin)
    narginchk(3,4);

    [meanState, stdDev, numSamples] = obj.validateSampleGaussianInput(meanState, stdDev, varargi

    obj.NormalDistribution.Mean = meanState;
    obj.NormalDistribution.Covariance = diag(stdDev.^2);

    state = obj.NormalDistribution.sample(numSamples);
    state = obj.enforceStateBounds(state);

end
```

Interpolate Between States

Define how to interpolate between two states in your state space. Use an input, `fraction`, to determine how to sample along the path between two states. For this example, define a basic linear interpolation method using the difference between states.

```
function interpState = interpolate(obj, state1, state2, fraction)
    narginchk(4,4);
    [state1, state2, fraction] = obj.validateInterpolateInput(state1, state2, fraction);

    stateDiff = state2 - state1;
    interpState = state1 + fraction' * stateDiff;
end
```

Calculate Distance Between States

Specify how to calculate the distance between two states in your state space. Use the `state1` and `state2` inputs to define the start and end positions. Both inputs can be a single state (row vector) or multiple states (matrix of row vectors). For this example, calculate the distance based on the Euclidean distance between each pair of state positions.

```
function dist = distance(obj, state1, state2)

    narginchk(3,3);

    nav.internal.validation.validateStateMatrix(state1, nan, obj.NumStateVariables, "distance", '
    nav.internal.validation.validateStateMatrix(state2, size(state1,1), obj.NumStateVariables, "d

    stateDiff = bsxfun(@minus, state2, state1);
    dist = sqrt( sum( stateDiff.^2, 2 ) );
end
```

Terminate the methods and class sections.

```
    end
end
```

Save your state space class definition. You can now use the class constructor to create an object for your state space.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`nav.StateValidator` | `stateSpaceSE2` | `stateSpaceDubins` | `stateSpaceReedsShepp`

Introduced in R2019b

distance

Class: `nav.StateSpace`
Package: `nav`

Distance between two states

Syntax

```
dist = distance(ssObj,state1,state2)
```

Description

`dist = distance(ssObj,state1,state2)` calculates the distance between two states.

Input Arguments

`ssObj` — State space object

object of a subclass of `nav.StateSpace`

State space object, specified as an object of a subclass of `nav.StateSpace`.

`state1` — Initial state position

n-element vector | *m*-by-*n* matrix of row vectors

Initial state position, specified as a *n*-element vector or *m*-by-*n* matrix of row vectors. *n* is the dimension of the state space specified in the `NumStateVariables` property of `ssObj`. *m* is the number of state positions provided.

If specified as a matrix, `state1` and `state2` should have the same dimensions.

`state2` — Final state position

n-element vector | *m*-by-*n* matrix of row vectors

Final state position, specified as a *n*-element vector or *m*-by-*n* matrix of row vectors. *n* is the dimension of the state space specified in the `NumStateVariables` property of `ssObj`. *m* is the number of state positions provided.

If specified as a matrix, `state1` and `state2` should have the same dimensions.

Output Arguments

`dist` — Distance between two states

numeric scalar | *m*-element vector

Distance between two states, returned as a numeric scalar or *m*-element vector. This distance calculation is the main component of evaluating costs of paths.

Examples

Create Custom State Space for Path Planning

This example shows how to use the `createPlanningTemplate` function to generate a template for customizing your own state space definition and sampler to use with path planning algorithms. A simple implementation is provided with the template.

Call the create template function. This function generates a class definition file for you to modify for your own implementation.

```
createPlanningTemplate
```

Class and Property Definition

The first part of the template specifies the class definition and any properties for the class. Derive from the `nav.StateSpace` class. For this example, create a property for the uniform and normal distributions. You can specify any additional user-defined properties here.

```
classdef MyCustomStateSpace < nav.StateSpace & ...
        matlabshared.planning.internal.EnforceScalarHandle
     properties
        UniformDistribution
        NormalDistribution
        % Specify additional properties here
end
```

Save your custom state space class and ensure your file name matches the class name.

Class Constructor

Use the constructor to set the name of the state space, the number of state variables, and define its boundaries. Alternatively, you can add input arguments to the function and pass the variables in when you create an object.

- For each state variable, define the `[min max]` values for the state bounds.
- Call the constructor of the base class.
- For this example, you specify the normal and uniform distribution property values using predefined `NormalDistribution` and `UniformDistribution` classes.
- Specify any other user-defined property values here.

```
methods
    function obj = MyCustomStateSpace
        spaceName = "MyCustomStateSpace";
        numStateVariables = 3;
        stateBounds = [-100 100;  % [min max]
                       -100 100;
                       -100 100];

        obj@nav.StateSpace(spaceName, numStateVariables, stateBounds);

        obj.NormalDistribution = matlabshared.tracking.internal.NormalDistribution(numStateVariab
        obj.UniformDistribution = matlabshared.tracking.internal.UniformDistribution(numStateVari
        % User-defined property values here
    end
```

Copy Semantics

Specify the `copy` method definition. Copy all the values of your user-defined variables into a new object, so `copyObj` is a deep copy. The default behavior given in this example creates a new copy of the object with the same name, state bounds, and distributions.

```
function copyObj = copy(obj)
    copyObj = feval(class(obj));
    copyObj.StateBounds = obj.StateBounds;
    copyObj.UniformDistribution = obj.UniformDistribution.copy;
    copyObj.NormalDistribution = obj.NormalDistribution.copy;
end
```

Enforce State Bounds

Specify how to ensure states are always within the state bounds. For this example, the state values get saturated at the minimum or maximum values for the state bounds.

```
function boundedState = enforceStateBounds(obj, state)
    nav.internal.validation.validateStateMatrix(state, nan, obj.NumStateVariables, "enforceStateB
    boundedState = state;
    boundedState = min(max(boundedState, obj.StateBounds(:,1)'), ...
        obj.StateBounds(:,2)');

end
```

Sample Uniformly

Specify the behavior for sampling across a uniform distribution. support multiple syntaxes to constrain the uniform distribution to a nearby state within a certain distance and sample multiple states.

```
STATE = sampleUniform(OBJ)
STATE = sampleUniform(OBJ,NUMSAMPLES)
STATE = sampleUniform(OBJ,NEARSTATE,DIST)
STATE = sampleUniform(OBJ,NEARSTATE,DIST,NUMSAMPLES)
```

For this example, use a validation function to process a `varargin` input that handles the varying input arguments.

```
 function state = sampleUniform(obj, varargin)
    narginchk(1,4);
    [numSamples, stateBounds] = obj.validateSampleUniformInput(varargin{:});

    obj.UniformDistribution.RandomVariableLimits = stateBounds;
    state = obj.UniformDistribution.sample(numSamples);
 end
```

Sample from Gaussian Distribution

Specify the behavior for sampling across a Gaussian distribution. Support multiple syntaxes for sampling a single state or multiple states.

```
STATE = sampleGaussian(OBJ, MEANSTATE, STDDEV)
STATE = sampleGaussian(OBJ, MEANSTATE, STDDEV, NUMSAMPLES)


function state = sampleGaussian(obj, meanState, stdDev, varargin)
    narginchk(3,4);
```

```
    [meanState, stdDev, numSamples] = obj.validateSampleGaussianInput(meanState, stdDev, varargi

    obj.NormalDistribution.Mean = meanState;
    obj.NormalDistribution.Covariance = diag(stdDev.^2);

    state = obj.NormalDistribution.sample(numSamples);
    state = obj.enforceStateBounds(state);

end
```

Interpolate Between States

Define how to interpolate between two states in your state space. Use an input, `fraction`, to determine how to sample along the path between two states. For this example, define a basic linear interpolation method using the difference between states.

```
function interpState = interpolate(obj, state1, state2, fraction)
    narginchk(4,4);
    [state1, state2, fraction] = obj.validateInterpolateInput(state1, state2, fraction);

    stateDiff = state2 - state1;
    interpState = state1 + fraction' * stateDiff;
end
```

Calculate Distance Between States

Specify how to calculate the distance between two states in your state space. Use the `state1` and `state2` inputs to define the start and end positions. Both inputs can be a single state (row vector) or multiple states (matrix of row vectors). For this example, calculate the distance based on the Euclidean distance between each pair of state positions.

```
function dist = distance(obj, state1, state2)

    narginchk(3,3);

    nav.internal.validation.validateStateMatrix(state1, nan, obj.NumStateVariables, "distance", "
    nav.internal.validation.validateStateMatrix(state2, size(state1,1), obj.NumStateVariables, "

    stateDiff = bsxfun(@minus, state2, state1);
    dist = sqrt( sum( stateDiff.^2, 2 ) );
end
```

Terminate the methods and class sections.

```
    end
end
```

Save your state space class definition. You can now use the class constructor to create an object for your state space.

See Also

`nav.StateSpace` | `nav.StateValidator` | `stateSpaceSE2` | `stateSpaceDubins` | `stateSpaceReedsShepp`

Introduced in R2019b

enforceStateBounds

Class: `nav.StateSpace`
Package: `nav`

Limit state to state bounds

Syntax

```
boundedState = enforceStateBounds(ssObj,state)
```

Description

`boundedState = enforceStateBounds(ssObj,state)` returns a bounded state that lies inside the state bounds based on the given `state`. Use this method to define specific bounding behavior like wrapping angular states. The bounds are specified in the `StateBounds` property of `ssObj`.

Input Arguments

`ssObj` — State space object

object of a subclass of `nav.StateSpace`

State space object, specified as an object of a subclass of `nav.StateSpace`.

`state` — State position

n-element vector | *m*-by-*n* matrix of row vectors

State position, specified as a *n*-element vector or an *m*-by-*n* matrix of row vectors. *n* is the dimension of the state space specified in the `NumStateVariables` property of `ssObj`.

Output Arguments

`boundedState` — State position with enforced state bounds

n-element vector | *m*-by-*n* matrix of row vectors

State position with enforced state bounds, specified as a *n*-element vector or *m*-by-*n* matrix of row vectors. *n* is the dimension of the state space specified in the `NumStateVariables` property of `ssObj`.

Examples

Create Custom State Space for Path Planning

This example shows how to use the `createPlanningTemplate` function to generate a template for customizing your own state space definition and sampler to use with path planning algorithms. A simple implementation is provided with the template.

Call the create template function. This function generates a class definition file for you to modify for your own implementation.

`createPlanningTemplate`

Class and Property Definition

The first part of the template specifies the class definition and any properties for the class. Derive from the `nav.StateSpace` class. For this example, create a property for the uniform and normal distributions. You can specify any additional user-defined properties here.

```
classdef MyCustomStateSpace < nav.StateSpace & ...
        matlabshared.planning.internal.EnforceScalarHandle
     properties
        UniformDistribution
        NormalDistribution
        % Specify additional properties here
end
```

Save your custom state space class and ensure your file name matches the class name.

Class Constructor

Use the constructor to set the name of the state space, the number of state variables, and define its boundaries. Alternatively, you can add input arguments to the function and pass the variables in when you create an object.

- For each state variable, define the `[min max]` values for the state bounds.
- Call the constructor of the base class.
- For this example, you specify the normal and uniform distribution property values using predefined `NormalDistribution` and `UniformDistribution` classes.
- Specify any other user-defined property values here.

```
methods
    function obj = MyCustomStateSpace
        spaceName = "MyCustomStateSpace";
        numStateVariables = 3;
        stateBounds = [-100 100;  % [min max]
                       -100 100;
                       -100 100];

        obj@nav.StateSpace(spaceName, numStateVariables, stateBounds);

        obj.NormalDistribution = matlabshared.tracking.internal.NormalDistribution(numStateVariab
        obj.UniformDistribution = matlabshared.tracking.internal.UniformDistribution(numStateVari
        % User-defined property values here
    end
```

Copy Semantics

Specify the `copy` method definition. Copy all the values of your user-defined variables into a new object, so `copyObj` is a deep copy. The default behavior given in this example creates a new copy of the object with the same name, state bounds, and distributions.

```
function copyObj = copy(obj)
    copyObj = feval(class(obj));
    copyObj.StateBounds = obj.StateBounds;
    copyObj.UniformDistribution = obj.UniformDistribution.copy;
    copyObj.NormalDistribution = obj.NormalDistribution.copy;
end
```

Enforce State Bounds

Specify how to ensure states are always within the state bounds. For this example, the state values get saturated at the minimum or maximum values for the state bounds.

```
function boundedState = enforceStateBounds(obj, state)
    nav.internal.validation.validateStateMatrix(state, nan, obj.NumStateVariables, "enforceStateB
    boundedState = state;
    boundedState = min(max(boundedState, obj.StateBounds(:,1)'), ...
        obj.StateBounds(:,2)');

end
```

Sample Uniformly

Specify the behavior for sampling across a uniform distribution. support multiple syntaxes to constrain the uniform distribution to a nearby state within a certain distance and sample multiple states.

```
STATE = sampleUniform(OBJ)
STATE = sampleUniform(OBJ,NUMSAMPLES)
STATE = sampleUniform(OBJ,NEARSTATE,DIST)
STATE = sampleUniform(OBJ,NEARSTATE,DIST,NUMSAMPLES)
```

For this example, use a validation function to process a `varargin` input that handles the varying input arguments.

```
 function state = sampleUniform(obj, varargin)
    narginchk(1,4);
    [numSamples, stateBounds] = obj.validateSampleUniformInput(varargin{:});

    obj.UniformDistribution.RandomVariableLimits = stateBounds;
    state = obj.UniformDistribution.sample(numSamples);
 end
```

Sample from Gaussian Distribution

Specify the behavior for sampling across a Gaussian distribution. Support multiple syntaxes for sampling a single state or multiple states.

```
STATE = sampleGaussian(OBJ, MEANSTATE, STDDEV)
STATE = sampleGaussian(OBJ, MEANSTATE, STDDEV, NUMSAMPLES)


function state = sampleGaussian(obj, meanState, stdDev, varargin)
    narginchk(3,4);

    [meanState, stdDev, numSamples] = obj.validateSampleGaussianInput(meanState, stdDev, varargi

    obj.NormalDistribution.Mean = meanState;
    obj.NormalDistribution.Covariance = diag(stdDev.^2);

    state = obj.NormalDistribution.sample(numSamples);
    state = obj.enforceStateBounds(state);

end
```

Interpolate Between States

Define how to interpolate between two states in your state space. Use an input, `fraction`, to determine how to sample along the path between two states. For this example, define a basic linear interpolation method using the difference between states.

```
function interpState = interpolate(obj, state1, state2, fraction)
    narginchk(4,4);
    [state1, state2, fraction] = obj.validateInterpolateInput(state1, state2, fraction);

    stateDiff = state2 - state1;
    interpState = state1 + fraction' * stateDiff;
end
```

Calculate Distance Between States

Specify how to calculate the distance between two states in your state space. Use the `state1` and `state2` inputs to define the start and end positions. Both inputs can be a single state (row vector) or multiple states (matrix of row vectors). For this example, calculate the distance based on the Euclidean distance between each pair of state positions.

```
function dist = distance(obj, state1, state2)

    narginchk(3,3);

    nav.internal.validation.validateStateMatrix(state1, nan, obj.NumStateVariables, "distance", '
    nav.internal.validation.validateStateMatrix(state2, size(state1,1), obj.NumStateVariables, "d

    stateDiff = bsxfun(@minus, state2, state1);
    dist = sqrt( sum( stateDiff.^2, 2 ) );
end
```

Terminate the methods and class sections.

```
    end
end
```

Save your state space class definition. You can now use the class constructor to create an object for your state space.

See Also

`nav.StateSpace` | `nav.StateValidator` | `stateSpaceSE2` | `stateSpaceDubins` | `stateSpaceReedsShepp`

Introduced in R2019b

interpolate

Class: `nav.StateSpace`
Package: `nav`

Interpolate between states

Syntax

```
interpStates = interpolate(ssObj,state1,state2,ratios)
```

Description

`interpStates = interpolate(ssObj,state1,state2,ratios)` interpolates between two states in your state space based on the given `ratios`.

Input Arguments

`ssObj` — State space object

object of a subclass of `nav.StateSpace`

State space object, specified as an object of a subclass of `nav.StateSpace`.

`state1` — Initial state position

n-element vector

Initial state position, specified as a *n*-element vector. *n* is the dimension of the state space specified in the `NumStateVariables` property of `ssObj`.

`state2` — Final state position

n-element vector | *m*-by-*n* matrix of row vectors

Final state position, specified as a *n*-element vector. *n* is the dimension of the state space specified in the `NumStateVariables` property of `ssObj`.

`ratios` — Ratio values for interpolating along path

m-element vector

Ratio values for interpolating along path, specified as an *m*-element vector. These ratios determine how to sample between the two states.

Output Arguments

`interpStates` — Interpolated states

m-by-*n* matrix of row vectors

Interpolated states, returned as an *m*-by-*n* matrix of row vectors. *m* is the length of `ratios` and *n* is the dimension of the state space specified in the `NumStateVariables` property of `ssObj`.

Examples

Create Custom State Space for Path Planning

This example shows how to use the `createPlanningTemplate` function to generate a template for customizing your own state space definition and sampler to use with path planning algorithms. A simple implementation is provided with the template.

Call the create template function. This function generates a class definition file for you to modify for your own implementation.

```
createPlanningTemplate
```

Class and Property Definition

The first part of the template specifies the class definition and any properties for the class. Derive from the `nav.StateSpace` class. For this example, create a property for the uniform and normal distributions. You can specify any additional user-defined properties here.

```
classdef MyCustomStateSpace < nav.StateSpace & ...
        matlabshared.planning.internal.EnforceScalarHandle
     properties
        UniformDistribution
        NormalDistribution
        % Specify additional properties here
end
```

Save your custom state space class and ensure your file name matches the class name.

Class Constructor

Use the constructor to set the name of the state space, the number of state variables, and define its boundaries. Alternatively, you can add input arguments to the function and pass the variables in when you create an object.

- For each state variable, define the `[min max]` values for the state bounds.
- Call the constructor of the base class.
- For this example, you specify the normal and uniform distribution property values using predefined `NormalDistribution` and `UniformDistribution` classes.
- Specify any other user-defined property values here.

```
methods
    function obj = MyCustomStateSpace
        spaceName = "MyCustomStateSpace";
        numStateVariables = 3;
        stateBounds = [-100 100;  % [min max]
                       -100 100;
                       -100 100];

        obj@nav.StateSpace(spaceName, numStateVariables, stateBounds);

        obj.NormalDistribution = matlabshared.tracking.internal.NormalDistribution(numStateVariab
        obj.UniformDistribution = matlabshared.tracking.internal.UniformDistribution(numStateVari
        % User-defined property values here
    end
```

Copy Semantics

Specify the `copy` method definition. Copy all the values of your user-defined variables into a new object, so `copyObj` is a deep copy. The default behavior given in this example creates a new copy of the object with the same name, state bounds, and distributions.

```
function copyObj = copy(obj)
    copyObj = feval(class(obj));
    copyObj.StateBounds = obj.StateBounds;
    copyObj.UniformDistribution = obj.UniformDistribution.copy;
    copyObj.NormalDistribution = obj.NormalDistribution.copy;
end
```

Enforce State Bounds

Specify how to ensure states are always within the state bounds. For this example, the state values get saturated at the minimum or maximum values for the state bounds.

```
function boundedState = enforceStateBounds(obj, state)
    nav.internal.validation.validateStateMatrix(state, nan, obj.NumStateVariables, "enforceStateB
    boundedState = state;
    boundedState = min(max(boundedState, obj.StateBounds(:,1)'), ...
        obj.StateBounds(:,2)');

end
```

Sample Uniformly

Specify the behavior for sampling across a uniform distribution. support multiple syntaxes to constrain the uniform distribution to a nearby state within a certain distance and sample multiple states.

```
STATE = sampleUniform(OBJ)
STATE = sampleUniform(OBJ,NUMSAMPLES)
STATE = sampleUniform(OBJ,NEARSTATE,DIST)
STATE = sampleUniform(OBJ,NEARSTATE,DIST,NUMSAMPLES)
```

For this example, use a validation function to process a `varargin` input that handles the varying input arguments.

```
 function state = sampleUniform(obj, varargin)
    narginchk(1,4);
    [numSamples, stateBounds] = obj.validateSampleUniformInput(varargin{:});

    obj.UniformDistribution.RandomVariableLimits = stateBounds;
    state = obj.UniformDistribution.sample(numSamples);
 end
```

Sample from Gaussian Distribution

Specify the behavior for sampling across a Gaussian distribution. Support multiple syntaxes for sampling a single state or multiple states.

```
STATE = sampleGaussian(OBJ, MEANSTATE, STDDEV)
STATE = sampleGaussian(OBJ, MEANSTATE, STDDEV, NUMSAMPLES)


function state = sampleGaussian(obj, meanState, stdDev, varargin)
    narginchk(3,4);
```

```
    [meanState, stdDev, numSamples] = obj.validateSampleGaussianInput(meanState, stdDev, varargi

    obj.NormalDistribution.Mean = meanState;
    obj.NormalDistribution.Covariance = diag(stdDev.^2);

    state = obj.NormalDistribution.sample(numSamples);
    state = obj.enforceStateBounds(state);

end
```

Interpolate Between States

Define how to interpolate between two states in your state space. Use an input, `fraction`, to determine how to sample along the path between two states. For this example, define a basic linear interpolation method using the difference between states.

```
function interpState = interpolate(obj, state1, state2, fraction)
    narginchk(4,4);
    [state1, state2, fraction] = obj.validateInterpolateInput(state1, state2, fraction);

    stateDiff = state2 - state1;
    interpState = state1 + fraction' * stateDiff;
end
```

Calculate Distance Between States

Specify how to calculate the distance between two states in your state space. Use the `state1` and `state2` inputs to define the start and end positions. Both inputs can be a single state (row vector) or multiple states (matrix of row vectors). For this example, calculate the distance based on the Euclidean distance between each pair of state positions.

```
function dist = distance(obj, state1, state2)

    narginchk(3,3);

    nav.internal.validation.validateStateMatrix(state1, nan, obj.NumStateVariables, "distance", "
    nav.internal.validation.validateStateMatrix(state2, size(state1,1), obj.NumStateVariables, "

    stateDiff = bsxfun(@minus, state2, state1);
    dist = sqrt( sum( stateDiff.^2, 2 ) );
end
```

Terminate the methods and class sections.

```
    end
end
```

Save your state space class definition. You can now use the class constructor to create an object for your state space.

See Also

`nav.StateSpace` | `nav.StateValidator` | `stateSpaceSE2` | `stateSpaceDubins` | `stateSpaceReedsShepp`

Introduced in R2019b

sampleGaussian

Class: `nav.StateSpace`
Package: `nav`

Sample state using Gaussian distribution

Syntax

```
states = sampleGaussian(ssObj,meanState,stdDev)
states = sampleGaussian(ssObj,meanState,stdDev,numSamples)
```

Description

`states = sampleGaussian(ssObj,meanState,stdDev)` samples a single state in your state space from a Gaussian distribution centered on `meanState` with specified standard deviation.

`states = sampleGaussian(ssObj,meanState,stdDev,numSamples)` samples multiple states based on `numSamples`.

Input Arguments

`ssObj` — State space object

object of a subclass of `nav.StateSpace`

State space object, specified as an object of a subclass of `nav.StateSpace`.

`meanState` — Mean state position

n-element vector | *m*-by-*n* matrix of row vectors

Mean state position, specified as a *n*-element vector or *m*-by-*n* matrix of row vectors, where *n* is the dimension of the state space specified in the `NumStateVariables` property of `ssObj`. *m* is the number of samples specified in `numSamples`.

`stdDev` — Standard deviation around mean state

n-element vector | *m*-by-*n* matrix of row vectors

Standard deviation around mean state, specified as an *n*-element vector or *m*-by-*n* matrix of row vectors, where each element corresponds to an element in `meanState`.

`numSamples` — Number of samples

positive integer

Number of samples, specified as a positive integer. By default, the function assumes `numSamples` is 1.

Output Arguments

`states` — Sampled states from state space

n-element vector | *m*-by-*n* matrix of row vectors

Sampled states from state space, specified as a *n*-element vector or *m*-by-*n* matrix of row vectors. *n* is the dimension of the state space specified in the `NumStateVariables` property of `ssObj`. *m* is the number of samples specified in `numSamples`. All states are sampled within the `StateBounds` property of `ssObj`.

Examples

Create Custom State Space for Path Planning

This example shows how to use the `createPlanningTemplate` function to generate a template for customizing your own state space definition and sampler to use with path planning algorithms. A simple implementation is provided with the template.

Call the create template function. This function generates a class definition file for you to modify for your own implementation.

```
createPlanningTemplate
```

Class and Property Definition

The first part of the template specifies the class definition and any properties for the class. Derive from the `nav.StateSpace` class. For this example, create a property for the uniform and normal distributions. You can specify any additional user-defined properties here.

```
classdef MyCustomStateSpace < nav.StateSpace & ...
        matlabshared.planning.internal.EnforceScalarHandle
     properties
        UniformDistribution
        NormalDistribution
        % Specify additional properties here
end
```

Save your custom state space class and ensure your file name matches the class name.

Class Constructor

Use the constructor to set the name of the state space, the number of state variables, and define its boundaries. Alternatively, you can add input arguments to the function and pass the variables in when you create an object.

- For each state variable, define the `[min max]` values for the state bounds.
- Call the constructor of the base class.
- For this example, you specify the normal and uniform distribution property values using predefined `NormalDistribution` and `UniformDistribution` classes.
- Specify any other user-defined property values here.

```
methods
    function obj = MyCustomStateSpace
        spaceName = "MyCustomStateSpace";
        numStateVariables = 3;
        stateBounds = [-100 100;  % [min max]
                       -100 100;
                       -100 100];
```

```
        obj@nav.StateSpace(spaceName, numStateVariables, stateBounds);

        obj.NormalDistribution = matlabshared.tracking.internal.NormalDistribution(numStateVariab
        obj.UniformDistribution = matlabshared.tracking.internal.UniformDistribution(numStateVari
        % User-defined property values here
    end
```

Copy Semantics

Specify the `copy` method definition. Copy all the values of your user-defined variables into a new object, so `copyObj` is a deep copy. The default behavior given in this example creates a new copy of the object with the same name, state bounds, and distributions.

```
function copyObj = copy(obj)
    copyObj = feval(class(obj));
    copyObj.StateBounds = obj.StateBounds;
    copyObj.UniformDistribution = obj.UniformDistribution.copy;
    copyObj.NormalDistribution = obj.NormalDistribution.copy;
end
```

Enforce State Bounds

Specify how to ensure states are always within the state bounds. For this example, the state values get saturated at the minimum or maximum values for the state bounds.

```
function boundedState = enforceStateBounds(obj, state)
    nav.internal.validation.validateStateMatrix(state, nan, obj.NumStateVariables, "enforceStateB
    boundedState = state;
    boundedState = min(max(boundedState, obj.StateBounds(:,1)'), ...
        obj.StateBounds(:,2)');

end
```

Sample Uniformly

Specify the behavior for sampling across a uniform distribution. support multiple syntaxes to constrain the uniform distribution to a nearby state within a certain distance and sample multiple states.

```
STATE = sampleUniform(OBJ)
STATE = sampleUniform(OBJ,NUMSAMPLES)
STATE = sampleUniform(OBJ,NEARSTATE,DIST)
STATE = sampleUniform(OBJ,NEARSTATE,DIST,NUMSAMPLES)
```

For this example, use a validation function to process a `varargin` input that handles the varying input arguments.

```
 function state = sampleUniform(obj, varargin)
    narginchk(1,4);
    [numSamples, stateBounds] = obj.validateSampleUniformInput(varargin{:});

    obj.UniformDistribution.RandomVariableLimits = stateBounds;
    state = obj.UniformDistribution.sample(numSamples);
 end
```

Sample from Gaussian Distribution

Specify the behavior for sampling across a Gaussian distribution. Support multiple syntaxes for sampling a single state or multiple states.

```
STATE = sampleGaussian(OBJ, MEANSTATE, STDDEV)
STATE = sampleGaussian(OBJ, MEANSTATE, STDDEV, NUMSAMPLES)


function state = sampleGaussian(obj, meanState, stdDev, varargin)
    narginchk(3,4);

    [meanState, stdDev, numSamples] = obj.validateSampleGaussianInput(meanState, stdDev, varargi

    obj.NormalDistribution.Mean = meanState;
    obj.NormalDistribution.Covariance = diag(stdDev.^2);

    state = obj.NormalDistribution.sample(numSamples);
    state = obj.enforceStateBounds(state);

end
```

Interpolate Between States

Define how to interpolate between two states in your state space. Use an input, `fraction`, to determine how to sample along the path between two states. For this example, define a basic linear interpolation method using the difference between states.

```
function interpState = interpolate(obj, state1, state2, fraction)
    narginchk(4,4);
    [state1, state2, fraction] = obj.validateInterpolateInput(state1, state2, fraction);

    stateDiff = state2 - state1;
    interpState = state1 + fraction' * stateDiff;
end
```

Calculate Distance Between States

Specify how to calculate the distance between two states in your state space. Use the `state1` and `state2` inputs to define the start and end positions. Both inputs can be a single state (row vector) or multiple states (matrix of row vectors). For this example, calculate the distance based on the Euclidean distance between each pair of state positions.

```
function dist = distance(obj, state1, state2)

    narginchk(3,3);

    nav.internal.validation.validateStateMatrix(state1, nan, obj.NumStateVariables, "distance", "
    nav.internal.validation.validateStateMatrix(state2, size(state1,1), obj.NumStateVariables, "d

    stateDiff = bsxfun(@minus, state2, state1);
    dist = sqrt( sum( stateDiff.^2, 2 ) );
end
```

Terminate the methods and class sections.

```
    end
end
```

Save your state space class definition. You can now use the class constructor to create an object for your state space.

See Also

`nav.StateSpace` | `nav.StateValidator` | `stateSpaceSE2` | `stateSpaceDubins` | `stateSpaceReedsShepp`

Introduced in R2019b

sampleUniform

Class: `nav.StateSpace`
Package: `nav`

Sample state using uniform distribution

Syntax

```
states = sampleUniform(ssObj)
states = sampleUniform(ssObj,numSamples)
states = sampleUniform(ssObj,meanState,distance)
states = sampleUniform(ssObj,meanState,distance,numSamples)
```

Description

`states = sampleUniform(ssObj)` samples throughout your entire state space using a uniform distribution.

`states = sampleUniform(ssObj,numSamples)` samples multiple states based on `numSamples`.

`states = sampleUniform(ssObj,meanState,distance)` samples near a given mean state within a certain distance.

`states = sampleUniform(ssObj,meanState,distance,numSamples)` samples multiple states near a given mean state based on `numSamples`.

Input Arguments

`ssObj` — State space object

object of a subclass of `nav.StateSpace`

State space object, specified as an object of a subclass of `nav.StateSpace`.

`meanState` — Mean state position

n-element vector

Mean state position for sampling near, specified as a *n*-element vector, where *n* is the dimension of the state space specified in the `NumStateVariables` property of `ssObj`. *m* is the number of samples specified in `numSamples`.

`distance` — Max distance from mean state position

n-element vector

Max distance from mean state position, `nearState`, specified as a *n*-element vector, where `nearState` defines the center of the sampled region and `distance` is the maximum distance from `nearState` allowed in each dimension.

`numSamples` — Number of samples

positive integer

Number of samples, specified as a positive integer.

Output Arguments

`states` — Sampled states from state space

n-element vector | *m*-by-*n* matrix of row vectors

Sampled states from state space, specified as a *n*-element vector or *m*-by-*n* matrix of row vectors. *n* is the dimension of the state space specified in the `NumStateVariables` property of `ssObj`. *m* is the number of samples specified in `numSamples`. All states are sampled within the `StateBounds` property of `ssObj`.

Examples

Create Custom State Space for Path Planning

This example shows how to use the `createPlanningTemplate` function to generate a template for customizing your own state space definition and sampler to use with path planning algorithms. A simple implementation is provided with the template.

Call the create template function. This function generates a class definition file for you to modify for your own implementation.

```
createPlanningTemplate
```

Class and Property Definition

The first part of the template specifies the class definition and any properties for the class. Derive from the `nav.StateSpace` class. For this example, create a property for the uniform and normal distributions. You can specify any additional user-defined properties here.

```
classdef MyCustomStateSpace < nav.StateSpace & ...
        matlabshared.planning.internal.EnforceScalarHandle
     properties
        UniformDistribution
        NormalDistribution
        % Specify additional properties here
end
```

Save your custom state space class and ensure your file name matches the class name.

Class Constructor

Use the constructor to set the name of the state space, the number of state variables, and define its boundaries. Alternatively, you can add input arguments to the function and pass the variables in when you create an object.

- For each state variable, define the `[min max]` values for the state bounds.
- Call the constructor of the base class.
- For this example, you specify the normal and uniform distribution property values using predefined `NormalDistribution` and `UniformDistribution` classes.
- Specify any other user-defined property values here.

```
methods
    function obj = MyCustomStateSpace
```

```
        spaceName = "MyCustomStateSpace";
        numStateVariables = 3;
        stateBounds = [-100 100;  % [min max]
                       -100 100;
                       -100 100];

        obj@nav.StateSpace(spaceName, numStateVariables, stateBounds);

        obj.NormalDistribution = matlabshared.tracking.internal.NormalDistribution(numStateVariab
        obj.UniformDistribution = matlabshared.tracking.internal.UniformDistribution(numStateVar
        % User-defined property values here
    end
```

Copy Semantics

Specify the `copy` method definition. Copy all the values of your user-defined variables into a new object, so `copyObj` is a deep copy. The default behavior given in this example creates a new copy of the object with the same name, state bounds, and distributions.

```
function copyObj = copy(obj)
    copyObj = feval(class(obj));
    copyObj.StateBounds = obj.StateBounds;
    copyObj.UniformDistribution = obj.UniformDistribution.copy;
    copyObj.NormalDistribution = obj.NormalDistribution.copy;
end
```

Enforce State Bounds

Specify how to ensure states are always within the state bounds. For this example, the state values get saturated at the minimum or maximum values for the state bounds.

```
function boundedState = enforceStateBounds(obj, state)
    nav.internal.validation.validateStateMatrix(state, nan, obj.NumStateVariables, "enforceStateB
    boundedState = state;
    boundedState = min(max(boundedState, obj.StateBounds(:,1)'), ...
        obj.StateBounds(:,2)');

end
```

Sample Uniformly

Specify the behavior for sampling across a uniform distribution. support multiple syntaxes to constrain the uniform distribution to a nearby state within a certain distance and sample multiple states.

```
STATE = sampleUniform(OBJ)
STATE = sampleUniform(OBJ,NUMSAMPLES)
STATE = sampleUniform(OBJ,NEARSTATE,DIST)
STATE = sampleUniform(OBJ,NEARSTATE,DIST,NUMSAMPLES)
```

For this example, use a validation function to process a `varargin` input that handles the varying input arguments.

```
 function state = sampleUniform(obj, varargin)
    narginchk(1,4);
    [numSamples, stateBounds] = obj.validateSampleUniformInput(varargin{:});

    obj.UniformDistribution.RandomVariableLimits = stateBounds;
```

```
    state = obj.UniformDistribution.sample(numSamples);
 end
```

Sample from Gaussian Distribution

Specify the behavior for sampling across a Gaussian distribution. Support multiple syntaxes for sampling a single state or multiple states.

```
STATE = sampleGaussian(OBJ, MEANSTATE, STDDEV)
STATE = sampleGaussian(OBJ, MEANSTATE, STDDEV, NUMSAMPLES)


function state = sampleGaussian(obj, meanState, stdDev, varargin)
    narginchk(3,4);

    [meanState, stdDev, numSamples] = obj.validateSampleGaussianInput(meanState, stdDev, varargi

    obj.NormalDistribution.Mean = meanState;
    obj.NormalDistribution.Covariance = diag(stdDev.^2);

    state = obj.NormalDistribution.sample(numSamples);
    state = obj.enforceStateBounds(state);

end
```

Interpolate Between States

Define how to interpolate between two states in your state space. Use an input, `fraction`, to determine how to sample along the path between two states. For this example, define a basic linear interpolation method using the difference between states.

```
function interpState = interpolate(obj, state1, state2, fraction)
    narginchk(4,4);
    [state1, state2, fraction] = obj.validateInterpolateInput(state1, state2, fraction);

    stateDiff = state2 - state1;
    interpState = state1 + fraction' * stateDiff;
end
```

Calculate Distance Between States

Specify how to calculate the distance between two states in your state space. Use the `state1` and `state2` inputs to define the start and end positions. Both inputs can be a single state (row vector) or multiple states (matrix of row vectors). For this example, calculate the distance based on the Euclidean distance between each pair of state positions.

```
function dist = distance(obj, state1, state2)

    narginchk(3,3);

    nav.internal.validation.validateStateMatrix(state1, nan, obj.NumStateVariables, "distance", "
    nav.internal.validation.validateStateMatrix(state2, size(state1,1), obj.NumStateVariables, "d

    stateDiff = bsxfun(@minus, state2, state1);
    dist = sqrt( sum( stateDiff.^2, 2 ) );
end
```

Terminate the methods and class sections.

```
    end
end
```

Save your state space class definition. You can now use the class constructor to create an object for your state space.

See Also

`nav.StateSpace` | `nav.StateValidator` | `stateSpaceSE2` | `stateSpaceDubins` | `stateSpaceReedsShepp`

Introduced in R2019b

nav.StateValidator class

Package: nav

Create state validator for path planning

Description

nav.StateValidator is an interface for all state validators used for path planning. Derive from this class if you want to define your own state validator. This representation allows for state and motion validation.

To create a sample template for generating your own state space class, call createPlanningTemplate("StateValidator"). For specific implementations of the state validator class for general application, see **State Validation** in “Motion Planning”.

The nav.StateValidator class is a handle class.

Class Attributes

Abstract	true

For information on class attributes, see “Class Attributes”.

Creation

Syntax

```
ssObj = nav.StateValidator(stateSpace)
```

Description

ssObj = nav.StateValidator(stateSpace) creates a state validator object that validates states in the given state space. This constructor can only be called from a derived class. Create your own class definition using createPlanningTemplate.

Properties

StateSpace — State space definition

object of a subclass from nav.StateSpace

State space definition, specified as an object of a subclass from nav.StateSpace. Specify this property using the stateSpace input on construction. You can also specify any of our predefined objects in the **State Validation** section from “Motion Planning”.

Example: stateSpaceSE2

Attributes:

`GetAccess`	`public`
`SetAccess`	`immutable`

Methods

Public Methods

copy	Copy array of handle objects
isMotionValid	Check if path between states is valid
isStateValid	Check if state is valid

Examples

Create Custom State Space Validator for Path Planning

This example shows how to use the `createPlanningTemplate` function to generate a template for customizing your own state validation class. State validation is used with path planning algorithms to ensure valid paths. The template function provides a basic implementation for example purposes.

Call the create template function. This function generates a class definition file for you to modify for your own implementation. Save this file.

```
createPlanningTemplate("StateValidator")
```

Class and Property Definition

The first part of the template specifies the class definition and any properties for the class. Derive from the `nav.StateValidator` class. You can specify any additional user-defined properties here.

```
classdef MyCustomStateValidator < nav.StateValidator & ...
        matlabshared.planning.internal.EnforceScalarHandle
    properties
       % User-defined properties
    end
```

Save your custom state validator class and ensure your file name matches the class name.

Class Constructor

Use the constructor to set the name of the state space validator and specify the state space object. Set a default value for the state space if one is not provided. Call the constructor of the base class. Initialize any other user-defined properties. This example uses a default of `MyCustomStateSpace`, which was illustrated in the previous example.

```
methods
        function obj = MyCustomStateValidator(space)
            narginchk(0,1)

            if nargin == 0
                space = MyCustomStateSpace;
            end

            obj@nav.StateValidator(space);
```

```
        % Initialize user-defined properties
    end
```

Copy Semantics

Specify the `copy` method definition. Copy all the values of your user-defined variables into a new object, so `copyObj` is a deep copy. The default behavior given in this example creates a new copy of the object with the same type.

```
    function copyObj = copy(obj)
        copyObj = feval(class(obj), obj.StateSpace);
    end
```

Check State Validity

Define how a given state is validated. The `state` input can either be a single row vector, or a matrix of row vectors for multiple states. Customize this function for any special validation behavior for your state space like collision checking against obstacles.

```
    function isValid = isStateValid(obj, state)
        narginchk(2,2);
        nav.internal.validation.validateStateMatrix(state, nan, obj.StateSpace.NumStateVariab
            "isStateValid", "state");

        bounds = obj.StateSpace.StateBounds';
        inBounds = state >= bounds(1,:) & state <= bounds(2,:);
        isValid = all(inBounds, 2);

    end
```

Check Motion Validity

Define how to generate the motion between states and determine if it is valid. For this example, use `linspace` to evenly interpolate between states and check if these states are valid using `isStateValid`. Customize this function to sample between states or consider other analytical methods for determining if a vehicle can move between given states.

```
    function [isValid, lastValid] = isMotionValid(obj, state1, state2)
        narginchk(3,3);
        state1 = nav.internal.validation.validateStateVector(state1, ...
            obj.StateSpace.NumStateVariables, "isMotionValid", "state1");
        state2 = nav.internal.validation.validateStateVector(state2, ...
            obj.StateSpace.NumStateVariables, "isMotionValid", "state2");

        if (~obj.isStateValid(state1))
            error("statevalidator:StartStateInvalid", "The start state of the motion is inval
        end

        % Interpolate at a fixed interval between states and check state validity
        numInterpPoints = 100;
        interpStates = obj.StateSpace.interpolate(state1, state2, linspace(0,1,numInterpPoint
        interpValid = obj.isStateValid(interpStates);

        % Look for invalid states. Set lastValid state to index-1.
        firstInvalidIdx = find(~interpValid, 1);
        if isempty(firstInvalidIdx)
            isValid = true;
```

```
                lastValid = state2;
            else
                isValid = false;
                lastValid = interpStates(firstInvalidIdx-1,:);
            end

        end
```

Terminate the methods and class sections.

```
    end
end
```

Save your state space validator class definition. You can now use the class constructor to create an object for validation of states for a given state space.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`nav.StateSpace` | `validatorOccupancyMap` | `validatorVehicleCostmap`

Introduced in R2019b

isMotionValid

Class: `nav.StateValidator`
Package: `nav`

Check if path between states is valid

Syntax

```
[isValid,lastValid] = isMotionValid(validatorObj,state1,state2)
```

Description

`[isValid,lastValid] = isMotionValid(validatorObj,state1,state2)` determines if the motion between two states is valid by interpolating between states. The function also returns the last valid state along the path.

A default implementation for this method is provided when you call `createPlanningTemplate`.

Input Arguments

`validatorObj` — State validator object

object from a subclass of `nav.StateValidator`

State validator object, specified as an object from a subclass of `nav.StateValidator`. For provided state validator objects, see `validatorOccupancyMap` or `validatorVehicleCostmap`.

`state1` — Initial state position

n-element vector | *m*-by-*n* matrix of row vectors

Initial state position, specified as a *n*-element vector or *m*-by-*n* matrix of row vectors. *n* is the dimension of the state space specified in the state space property in `validatorObj`.

`state2` — Final state position

n-element vector | *m*-by-*n* matrix of row vectors

Final state position, specified as a *n*-element vector or *m*-by-*n* matrix of row vectors. *n* is the dimension of the state space specified in the state space property in `validatorObj`.

Output Arguments

`isValid` — Valid states

m-element vector of `1`s and `0`s

Valid states, specified as a *m*-element vector of `1`s and `0`s.

Data Types: `logical`

`lastValid` — Final valid state along path

n-element vector

Final valid state along path, specified as a *n*-element vector. *n* is the dimension of the state space specified in the state space property in `validatorObj`.

Examples

Create Custom State Space Validator for Path Planning

This example shows how to use the `createPlanningTemplate` function to generate a template for customizing your own state validation class. State validation is used with path planning algorithms to ensure valid paths. The template function provides a basic implementation for example purposes.

Call the create template function. This function generates a class definition file for you to modify for your own implementation. Save this file.

```
createPlanningTemplate("StateValidator")
```

Class and Property Definition

The first part of the template specifies the class definition and any properties for the class. Derive from the `nav.StateValidator` class. You can specify any additional user-defined properties here.

```
classdef MyCustomStateValidator < nav.StateValidator & ...
        matlabshared.planning.internal.EnforceScalarHandle
    properties
       % User-defined properties
    end
```

Save your custom state validator class and ensure your file name matches the class name.

Class Constructor

Use the constructor to set the name of the state space validator and specify the state space object. Set a default value for the state space if one is not provided. Call the constructor of the base class. Initialize any other user-defined properties. This example uses a default of `MyCustomStateSpace`, which was illustrated in the previous example.

```
methods
        function obj = MyCustomStateValidator(space)
            narginchk(0,1)

            if nargin == 0
                space = MyCustomStateSpace;
            end

            obj@nav.StateValidator(space);

           % Initialize user-defined properties
        end
```

Copy Semantics

Specify the `copy` method definition. Copy all the values of your user-defined variables into a new object, so `copyObj` is a deep copy. The default behavior given in this example creates a new copy of the object with the same type.

```
function copyObj = copy(obj)
    copyObj = feval(class(obj), obj.StateSpace);
end
```

Check State Validity

Define how a given state is validated. The `state` input can either be a single row vector, or a matrix of row vectors for multiple states. Customize this function for any special validation behavior for your state space like collision checking against obstacles.

```
function isValid = isStateValid(obj, state)
    narginchk(2,2);
    nav.internal.validation.validateStateMatrix(state, nan, obj.StateSpace.NumStateVariabl
        "isStateValid", "state");

    bounds = obj.StateSpace.StateBounds';
    inBounds = state >= bounds(1,:) & state <= bounds(2,:);
    isValid = all(inBounds, 2);

end
```

Check Motion Validity

Define how to generate the motion between states and determine if it is valid. For this example, use `linspace` to evenly interpolate between states and check if these states are valid using `isStateValid`. Customize this function to sample between states or consider other analytical methods for determining if a vehicle can move between given states.

```
function [isValid, lastValid] = isMotionValid(obj, state1, state2)
    narginchk(3,3);
    state1 = nav.internal.validation.validateStateVector(state1, ...
        obj.StateSpace.NumStateVariables, "isMotionValid", "state1");
    state2 = nav.internal.validation.validateStateVector(state2, ...
        obj.StateSpace.NumStateVariables, "isMotionValid", "state2");

    if (~obj.isStateValid(state1))
        error("statevalidator:StartStateInvalid", "The start state of the motion is inval
    end

    % Interpolate at a fixed interval between states and check state validity
    numInterpPoints = 100;
    interpStates = obj.StateSpace.interpolate(state1, state2, linspace(0,1,numInterpPoint
    interpValid = obj.isStateValid(interpStates);

    % Look for invalid states. Set lastValid state to index-1.
    firstInvalidIdx = find(~interpValid, 1);
    if isempty(firstInvalidIdx)
        isValid = true;
        lastValid = state2;
    else
        isValid = false;
        lastValid = interpStates(firstInvalidIdx-1,:);
    end

end
```

Terminate the methods and class sections.

```
    end
end
```

Save your state space validator class definition. You can now use the class constructor to create an object for validation of states for a given state space.

See Also

`nav.StateValidator` | `nav.StateSpace` | `validatorOccupancyMap` | `validatorVehicleCostmap`

Introduced in R2019b

isStateValid

Class: `nav.StateValidator`
Package: `nav`

Check if state is valid

Syntax

```
isValid = isStateValid(validatorObj,states)
```

Description

`isValid = isStateValid(validatorObj,states)` determines if the `states` are valid.

Input Arguments

`validatorObj` — State validator object

object from a subclass of `nav.StateValidator`

State validator object, specified as an object from a subclass of `nav.StateValidator`. For provided state validator objects, see `validatorOccupancyMap` or `validatorVehicleCostmap`.

`states` — State positions

n-element vector | *m*-by-*n* matrix of row vectors

Initial state position, specified as a *n*-element vector or *m*-by-*n* matrix of row vectors. *n* is the dimension of the state space specified in `validatorObj`. *m* is the number of states to validate.

Output Arguments

`isValid` — Valid states

m-element vector of `1`s and `0`s

Valid states, specified as a *m*-element vector of `1`s and `0`s.

Examples

Create Custom State Space Validator for Path Planning

This example shows how to use the `createPlanningTemplate` function to generate a template for customizing your own state validation class. State validation is used with path planning algorithms to ensure valid paths. The template function provides a basic implementation for example purposes.

Call the create template function. This function generates a class definition file for you to modify for your own implementation. Save this file.

```
createPlanningTemplate("StateValidator")
```

Class and Property Definition

The first part of the template specifies the class definition and any properties for the class. Derive from the `nav.StateValidator` class. You can specify any additional user-defined properties here.

```
classdef MyCustomStateValidator < nav.StateValidator & ...
        matlabshared.planning.internal.EnforceScalarHandle
    properties
       % User-defined properties
    end
```

Save your custom state validator class and ensure your file name matches the class name.

Class Constructor

Use the constructor to set the name of the state space validator and specify the state space object. Set a default value for the state space if one is not provided. Call the constructor of the base class. Initialize any other user-defined properties. This example uses a default of `MyCustomStateSpace`, which was illustrated in the previous example.

```
methods
        function obj = MyCustomStateValidator(space)
            narginchk(0,1)

            if nargin == 0
                space = MyCustomStateSpace;
            end

            obj@nav.StateValidator(space);

           % Initialize user-defined properties
        end
```

Copy Semantics

Specify the `copy` method definition. Copy all the values of your user-defined variables into a new object, so `copyObj` is a deep copy. The default behavior given in this example creates a new copy of the object with the same type.

```
        function copyObj = copy(obj)
            copyObj = feval(class(obj), obj.StateSpace);
        end
```

Check State Validity

Define how a given state is validated. The `state` input can either be a single row vector, or a matrix of row vectors for multiple states. Customize this function for any special validation behavior for your state space like collision checking against obstacles.

```
        function isValid = isStateValid(obj, state)
            narginchk(2,2);
            nav.internal.validation.validateStateMatrix(state, nan, obj.StateSpace.NumStateVariabl
                "isStateValid", "state");

            bounds = obj.StateSpace.StateBounds';
            inBounds = state >= bounds(1,:) & state <= bounds(2,:);
            isValid = all(inBounds, 2);
```

```
        end
```

Check Motion Validity

Define how to generate the motion between states and determine if it is valid. For this example, use `linspace` to evenly interpolate between states and check if these states are valid using `isStateValid`. Customize this function to sample between states or consider other analytical methods for determining if a vehicle can move between given states.

```
        function [isValid, lastValid] = isMotionValid(obj, state1, state2)
            narginchk(3,3);
            state1 = nav.internal.validation.validateStateVector(state1, ...
                obj.StateSpace.NumStateVariables, "isMotionValid", "state1");
            state2 = nav.internal.validation.validateStateVector(state2, ...
                obj.StateSpace.NumStateVariables, "isMotionValid", "state2");

            if (~obj.isStateValid(state1))
                error("statevalidator:StartStateInvalid", "The start state of the motion is inva
            end

            % Interpolate at a fixed interval between states and check state validity
            numInterpPoints = 100;
            interpStates = obj.StateSpace.interpolate(state1, state2, linspace(0,1,numInterpPoin
            interpValid = obj.isStateValid(interpStates);

            % Look for invalid states. Set lastValid state to index-1.
            firstInvalidIdx = find(~interpValid, 1);
            if isempty(firstInvalidIdx)
                isValid = true;
                lastValid = state2;
            else
                isValid = false;
                lastValid = interpStates(firstInvalidIdx-1,:);
            end

        end
```

Terminate the methods and class sections.

```
    end
end
```

Save your state space validator class definition. You can now use the class constructor to create an object for validation of states for a given state space.

See Also

`nav.StateValidator` | `nav.StateSpace` | `validatorOccupancyMap` | `validatorVehicleCostmap`

Introduced in R2019b

insfilterNonholonomic

Estimate pose with nonholonomic constraints

Description

The `insfilterNonholonomic` object implements sensor fusion of inertial measurement unit (IMU) and GPS data to estimate pose in the NED (or ENU) reference frame. IMU data is derived from gyroscope and accelerometer data. The filter uses a 16-element state vector to track the orientation quaternion, velocity, position, and IMU sensor biases. The `insfilterNonholonomic` object uses an extended Kalman filter to estimate these quantities.

Creation

Syntax

```
filter = insfilterNonholonomic
filter = insfilterNonholonomic('ReferenceFrame',RF)
filter = insfilterNonholonomic( ___ ,Name,Value)
```

Description

`filter = insfilterNonholonomic` creates an `insfilterErrorState` object with default property values.

`filter = insfilterNonholonomic('ReferenceFrame',RF)` allows you to specify the reference frame, `RF`, of the `filter`. Specify `RF` as `'NED'` (North-East-Down) or `'ENU'` (East-North-Up). The default value is `'NED'`.

`filter = insfilterNonholonomic( ___ ,Name,Value)` also allows you set properties of the created `filter` using one or more name-value pairs. Enclose each property name in single quotes.

Properties

`IMUSampleRate` — Sample rate of the IMU (Hz)
`100` (default) | positive scalar

Sample rate of the IMU in Hz, specified as a positive scalar.

Data Types: `single` | `double`

`ReferenceLocation` — Reference location (deg, deg, meters)
`[0 0 0]` (default) | 3-element positive row vector

Reference location, specified as a 3-element row vector in geodetic coordinates (latitude, longitude, and altitude). Altitude is the height above the reference ellipsoid model, WGS84. The reference location units are [degrees degrees meters].

Data Types: `single` | `double`

`DecimationFactor` — Decimation factor for kinematic constraint correction
2 (default) | positive integer scalar

Decimation factor for kinematic constraint correction, specified as a positive integer scalar.

Data Types: `single` | `double`

`GyroscopeNoise` — Multiplicative process noise variance from gyroscope $(rad/s)^2$
`[4.8e-6 4.8e-6 4.8e-6]` (default) | scalar | 3-element row vector

Multiplicative process noise variance from the gyroscope in $(rad/s)^2$, specified as a scalar or 3-element row vector of positive real finite numbers.

- If `GyroscopeNoise` is specified as a row vector, the elements correspond to the noise in the *x*, *y*, and *z* axes of the gyroscope, respectively.
- If `GyroscopeNoise` is specified as a scalar, the single element is applied to the *x*, *y*, and *z* axes of the gyroscope.

Data Types: `single` | `double`

`GyroscopeBiasNoise` — Multiplicative process noise variance from gyroscope bias $(rad/s)^2$
`[4e-14 4e-14 4e-14]` (default) | scalar | 3-element row vector

Multiplicative process noise variance from the gyroscope bias in $(rad/s)^2$, specified as a scalar or 3-element row vector of positive real finite numbers. Gyroscope bias is modeled as a lowpass filtered white noise process.

- If `GyroscopeBiasNoise` is specified as a row vector, the elements correspond to the noise in the *x*, *y*, and *z* axes of the gyroscope, respectively.
- If `GyroscopeBiasNoise` is specified as a scalar, the single element is applied to the *x*, *y*, and *z* axes of the gyroscope.

Data Types: `single` | `double`

`GyroscopeBiasDecayFactor` — Decay factor for gyroscope bias
`0.999` (default) | scalar in the range [0,1]

Decay factor for gyroscope bias, specified as a scalar in the range [0,1]. A decay factor of `0` models gyroscope bias as a white noise process. A decay factor of `1` models the gyroscope bias as a random walk process.

Data Types: `single` | `double`

`AccelerometerNoise` — Multiplicative process noise variance from accelerometer $(m/s^2)^2$
`[4.8e-2 4.8e-2 4.8e-2]` (default) | scalar | 3-element row vector

Multiplicative process noise variance from the accelerometer in $(m/s^2)^2$, specified as a scalar or 3-element row vector of positive real finite numbers.

- If `AccelerometerNoise` is specified as a row vector, the elements correspond to the noise in the *x*, *y*, and *z* axes of the accelerometer, respectively.
- If `AccelerometerNoise` is specified as a scalar, the single element is applied to each axis.

Data Types: `single` | `double`

`AccelerometerBiasNoise` — Multiplicative process noise variance from accelerometer bias $(m/s^2)^2$
`[4e-14 4e-14 4e-14]` (default) | positive scalar | 3-element row vector

Multiplicative process noise variance from the accelerometer bias in $(m/s^2)^2$, specified as a scalar or 3-element row vector of positive real numbers. Accelerometer bias is modeled as a lowpass filtered white noise process.

- If `AccelerometerBiasNoise` is specified as a row vector, the elements correspond to the noise in the *x*, *y*, and *z* axes of the accelerometer, respectively.
- If `AccelerometerBiasNoise` is specified as a scalar, the single element is applied to each axis.

`AccelerometerBiasDecayFactor` — Decay factor for accelerometer bias
`0.9999` (default) | scalar in the range [0,1]

Decay factor for accelerometer bias, specified as a scalar in the range [0,1]. A decay factor of `0` models accelerometer bias as a white noise process. A decay factor of `1` models the accelerometer bias as a random walk process.

Data Types: `single` | `double`

`State` — State vector of extended Kalman filter
`[1;zeros(15,1)]` | 16-element column vector

State vector of the extended Kalman filter. The state values represent:

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

Data Types: `single` | `double`

`StateCovariance` — State error covariance for extended Kalman filter
`eye(16)` (default) | 16-by-16 matrix

State error covariance for the extended Kalman filter, specified as a 16-by-16-element matrix, or real numbers.

Data Types: `single` | `double`

`ZeroVelocityConstraintNoise` — Velocity constraints noise $(m/s)^2$
`1e-2` (default) | nonnegative scalar

Velocity constraints noise in $(m/s)^2$, specified as a nonnegative scalar.

Data Types: `single` | `double`

Object Functions

correct Correct states using direct state measurements for insfilterNonholonomic

residual	Residuals and residual covariances from direct state measurements for insfilterNonholonomic
fusegps	Correct states using GPS data for insfilterNonholonomic
residualgps	Residuals and residual covariance from GPS measurements for insfilterNonholonomic
pose	Current orientation and position estimate for insfilterNonholonomic
predict	Update states using accelerometer and gyroscope data for insfilterNonholonomic
reset	Reset internal states for insfilterNonholonomic
stateinfo	Display state vector information for insfilterNonholonomic
tune	Tune insfilterNonholonomic parameters to reduce estimation error
copy	Create copy of insfitlerNonholonomic

Examples

Estimate Pose of Ground Vehicle

This example shows how to estimate the pose of a ground vehicle from logged IMU and GPS sensor measurements and ground truth orientation and position.

Load the logged data of a ground vehicle following a circular trajectory.

```
load('loggedGroundVehicleCircle.mat','imuFs','localOrigin','initialState','initialStateCovariance
      'gyroData','gpsFs','gpsLLA','Rpos','gpsVel','Rvel','trueOrient','truePos');
```

Initialize the `insfilterNonholonomic` object.

```
filt = insfilterNonholonomic;
filt.IMUSampleRate = imuFs;
filt.ReferenceLocation = localOrigin;
filt.State = initialState;
filt.StateCovariance = initialStateCovariance;

imuSamplesPerGPS = imuFs/gpsFs;
```

Log data for final metric computation. Use the `predict` object function to estimate filter state based on accelerometer and gyroscope data. Then correct the filter state according to GPS data.

```
numIMUSamples = size(accelData,1);
estOrient = quaternion.ones(numIMUSamples,1);
estPos = zeros(numIMUSamples,3);

gpsIdx = 1;

for idx = 1:numIMUSamples
    predict(filt,accelData(idx,:),gyroData(idx,:));       %Predict filter state

    if (mod(idx,imuSamplesPerGPS) == 0)                   %Correct filter state
        fusegps(filt,gpsLLA(gpsIdx,:),Rpos,gpsVel(gpsIdx,:),Rvel);
        gpsIdx = gpsIdx + 1;
    end

    [estPos(idx,:),estOrient(idx,:)] = pose(filt);        %Log estimated pose
end
```

Calculate and display RMS errors.

```
posd = estPos - truePos;
quatd = rad2deg(dist(estOrient,trueOrient));
msep = sqrt(mean(posd.^2));

fprintf('Position RMS Error\n\tX: %.2f, Y: %.2f, Z: %.2f (meters)\n\n',msep(1),msep(2),msep(3));
```

```
Position RMS Error
    X: 0.15, Y: 0.11, Z: 0.01 (meters)
```

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

```
Quaternion Distance RMS Error
    0.26 (degrees)
```

Algorithms

Note: The following algorithm only applies to an NED reference frame.

`insfilterNonholonomic` uses a 16-axis error state Kalman filter structure to estimate pose in the NED reference frame. The state is defined as:

$$x = \begin{bmatrix} q_0 \\ q_1 \\ q_2 \\ q_3 \\ gyrobias_X \\ gyrobias_Y \\ gyrobias_Z \\ position_N \\ position_E \\ position_D \\ v_N \\ v_E \\ v_D \\ accelbias_X \\ accelbias_Y \\ accelbias_Z \end{bmatrix}$$

where

- q_0, q_1, q_2, q_3 –– Parts of orientation quaternion. The orientation quaternion represents a frame rotation from the platform's current orientation to the local NED coordinate system.
- $gyrobias_X$, $gyrobias_Y$, $gyrobias_Z$ –– Bias in the gyroscope reading.
- $position_N$, $position_E$, $position_D$ –– Position of the platform in the local NED coordinate system.
- v_N, v_E, v_D –– Velocity of the platform in the local NED coordinate system.

- $accelbias_X$, $accelbias_Y$, $accelbias_Z$ –– Bias in the accelerometer reading.

Given the conventional formulation of the state transition function,

$$x_{k|k-1} = f(\widehat{x}_{k-1|k-1})$$

the predicted state estimate is:

$$x_{k|k-1} =$$

$$\begin{bmatrix}
q_0 + \Delta t * q_1(gyrobias_X/2 - gyro_X/2) + \Delta t * q_2 * (gyrobias_Y/2 - gyro_Y/2) + \Delta t * q_3 * (gyrobias_Z/2 - gyro_Z/2) \\
q_1 - \Delta t * q_0(gyrobias_X/2 - gyro_X/2) + \Delta t * q_3 * (gyrobias_Y/2 - gyro_Y/2) - \Delta t * q_2 * (gyrobias_Z/2 - gyro_Z/2) \\
q_2 - \Delta t * q_3(gyrobias_X/2 - gyro_X/2) - \Delta t * q_0 * (gyrobias_Y/2 - gyro_Y/2) + \Delta t * q_1 * (gyrobias_Z/2 - gyro_Z/2) \\
q_3 + \Delta t * q_2(gyrobias_X/2 - gyro_X/2) - \Delta t * q_1 * (gyrobias_Y/2 - gyro_Y/2) - \Delta t * q_0 * (gyrobias_Z/2 - gyro_Z/2) \\
-gryobias_X * (\Delta t * \lambda_{gyro} - 1) \\
-gryobias_Y * (\Delta t * \lambda_{gyro} - 1) \\
-gryobias_Z * (\Delta t * \lambda_{gyro} - 1) \\
position_N + \Delta t * v_N \\
position_E + \Delta t * v_E \\
position_D + \Delta t * v_D \\
v_N + \Delta t * \begin{Bmatrix}
q_0 * (q_0 * (accelbias_X - accel_X) - q_3 * (accelbias_Y - accel_Y) + q_2 * (accelbias_Z - accel_Z)) - g_N + \\
q_2 * (q_1 * (accelbias_Y - accel_Y) - q_2 * (accelbias_X - accel_X) + q_0 * (accelbias_Z - accel_Z)) + \\
q_1 * (q_1 * (accelbias_X - accel_X) + q_2 * (accelbias_Y - accel_Y) + q_3 * (accelbias_Z - accel_Z)) - \\
q_3 * (q_3 * (accelbias_X - accel_X) + q_0 * (accelbias_Y - accel_Y) - q_1 * (accelbias_Z - accel_Z))
\end{Bmatrix} \\
v_E + \Delta t * \begin{Bmatrix}
q_0 * (q_3 * (accelbias_X - accel_X) + q_0 * (accelbias_Y - accel_Y) - q_1 * (accelbias_Z - accel_Z)) - g_E - \\
q_1 * (q_1 * (accelbias_Y - accel_Y) - q_2 * (accelbias_X - accel_X) + q_0 * (accelbias_Z - accel_Z)) + \\
q_2 * (q_1 * (accelbias_X - accel_X) + q_2 * (accelbias_Y - accel_Y) + q_3 * (accelbias_Z - accel_Z)) + \\
q_3 * (q_3 * (accelbias_X - accel_X) + q_0 * (accelbias_Y - accel_Y) - q_1 * (accelbias_Z - accel_Z))
\end{Bmatrix}
\end{bmatrix}$$

where

- Δt –– IMU sample time.
- g_N, g_E, g_D –– Constant gravity vector in the NED frame.
- $accel_X$, $accel_Y$, $accel_Z$ –– Acceleration vector in the body frame.
- λ_{accel} –– Accelerometer bias decay factor.
- λ_{gyro} –– Gyroscope bias decay factor.

References

[1] Munguía, R. "A GPS-Aided Inertial Navigation System in Direct Configuration." *Journal of applied research and technology.* Vol. 12, Number 4, 2014, pp. 803 – 814.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`insfilterMARG` | `insfilterErrorState` | `insfilterAsync`

Introduced in R2018b

correct

Correct states using direct state measurements for `insfilterNonholonomic`

Syntax

```
correct(FUSE,idx,measurement,measurementCovariance)
```

Description

`correct(FUSE,idx,measurement,measurementCovariance)` corrects the state and state estimation error covariance based on the measurement and measurement covariance. The measurement maps directly to the state specified by the indices `idx`.

Input Arguments

`FUSE` — `insfilterNonholonomic` object

object

`insfilterNonholonomic`, specified as an object.

`idx` — State vector Index of measurement to correct

N-element vector of increasing integers in the range [1,16]

State vector index of measurement to correct, specified as an *N*-element vector of increasing integers in the range [1,16].

The state values represent:

State	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

Data Types: `single` | `double`

`measurement` — Direct measurement of state

N-element vector

Direct measurement of state, specified as a *N*-element vector. *N* is the number of elements of the index argument, idx.

Data Types: `single` | `double`

`measurementCovariance` — Covariance of measurement

scalar | *N*-element vector | *N*-by-*N* matrix

Covariance of measurement, specified as a scalar, *N*-element vector, or *N*-by-*N* matrix. *N* is the number of elements of the index argument, idx.

Data Types: `single` | `double`

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`insfilter` | `insfilterNonholonomic`

Introduced in R2018b

copy

Create copy of `insfitlerNonholonomic`

Syntax

```
newFilter = copy(filter)
```

Description

`newFilter = copy(filter)` returns a copy of the `insfilterNonholonomic`, `filter`, with the exactly same property values.

Input Arguments

`filter` — Filter to be copied

`insfitlerNonholonomic`

Filter to be copied, specified as an `insfilterNonholonomic` object.

Output Arguments

`newFilter` — New copied filter

`insfitlerNonholonomic`

New copied filter, returned as an `insfilterNonholonomic` object.

See Also

`insfilterNonholonomic`

Introduced in R2020b

fusegps

Correct states using GPS data for `insfilterNonholonomic`

Syntax

```
[res,resCov] = fusegps(FUSE,position,positionCovariance)
[res,resCov] = fusegps(FUSE,position,positionCovariance,velocity,
velocityCovariance)
```

Description

`[res,resCov] = fusegps(FUSE,position,positionCovariance)` fuses GPS position data to correct the state estimate.

`[res,resCov] = fusegps(FUSE,position,positionCovariance,velocity, velocityCovariance)` fuses GPS position and velocity data to correct the state estimate.

Input Arguments

`FUSE` — `insfilterNonholonomic` object
object

`insfilterNonholonomic`, specified as an object.

`position` — Position of GPS receiver (LLA)
3-element row vector

Position of GPS receiver in geodetic latitude, longitude, and altitude (LLA) specified as a real finite 3-element row vector. Latitude and longitude are in degrees with north and east being positive. Altitude is in meters.

Data Types: `single` | `double`

`positionCovariance` — Position measurement covariance of GPS receiver (m^2)
3-by-3 matrix

Position measurement covariance of GPS receiver in m^2, specified as a 3-by-3 matrix.

Data Types: `single` | `double`

`velocity` — Velocity of GPS receiver in local NED coordinate system (m/s)
3-element row vector

Velocity of the GPS receiver in the local NED coordinate system in m/s, specified as a 3-element row vector.

Data Types: `single` | `double`

`velocityCovariance` — Velocity measurement covariance of GPS receiver (m/s^2)
3-by-3 matrix

Velocity measurement covariance of the GPS receiver in the local NED coordinate system in m/s^2, specified as a 3-by-3 matrix.

Data Types: `single` | `double`

Output Arguments

`res` — Position and course residual

1-by-4 vector of real values

Position and course residual, returned as a 1-by-6 vector of real values in m and rad/s, respectively.

`resCov` — Residual covariance

4-by-4 matrix of real values

Residual covariance, returned as a 4-by-4 matrix of real values.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`insfilter` | `insfilterNonholonomic`

Introduced in R2018b

pose

Current orientation and position estimate for `insfilterNonholonomic`

Syntax

```
[position,orientation,velocity] = pose(FUSE)
[position,orientation,velocity] = pose(FUSE,format)
```

Description

`[position,orientation,velocity] = pose(FUSE)` returns the current estimate of the pose.

`[position,orientation,velocity] = pose(FUSE,format)` returns the current estimate of the pose with orientation in the specified orientation format.

Input Arguments

`FUSE` — `NHConstrainedIMUGPSFuser` object
object

`insfilterNonholonomic`, specified as an object.

`format` — Output orientation format
`'quaternion'` (default) | `'rotmat'`

Output orientation format, specified as either `'quaternion'` for a `quaternion` or `'rotmat'` for a rotation matrix.

Data Types: `char` | `string`

Output Arguments

`position` — Position estimate expressed in the local coordinate system (m)
3-element row vector

Position estimate expressed in the local coordinate system of the filter in meters, returned as a 3-element row vector.

Data Types: `single` | `double`

`orientation` — Orientation estimate expressed in the local coordinate system
quaternion (default) | 3-by-3 rotation matrix

Orientation estimate expressed in the local coordinate system of the filter, returned as a scalar quaternion or 3-by-3 rotation matrix. The quaternion or rotation matrix represents a frame rotation from the local reference frame of the filter to the body reference frame.

Data Types: `single` | `double` | `quaternion`

`velocity` — Velocity estimate expressed in local coordinate system (m/s)
3-element row vector

Velocity estimate expressed in the local coordinate system of the filter in m/s, returned as a 3-element row vector.

Data Types: `single` | `double`

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`insfilter` | `insfilterNonholonomic`

Introduced in R2018b

predict

Update states using accelerometer and gyroscope data for `insfilterNonholonomic`

Syntax

```
predict(FUSE,accelReadings,gyroReadings)
```

Description

`predict(FUSE,accelReadings,gyroReadings)` fuses accelerometer and gyroscope data to update the state estimate.

Input Arguments

`FUSE` — `insfilterNonholonomic` object
object

`insfilterNonholonomic`, specified as an object.

`accelReadings` — Accelerometer readings in local sensor body coordinate system (m/s^2)
3-element row vector

Accelerometer readings in m/s^2, specified as a 3-element row vector.

Data Types: `single` | `double`

`gyroReadings` — Gyroscope readings in local sensor body coordinate system (rad/s)
3-element row vector

Gyroscope readings in rad/s, specified as a 3-element row vector.

Data Types: `single` | `double`

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

`insfilter` | `insfilterNonholonomic`

Introduced in R2018b

reset

Reset internal states for `insfilterNonholonomic`

Syntax

```
reset(FUSE)
```

Description

`reset(FUSE)` resets the State, StateCovariance, and internal integrators to their default values.

Input Arguments

`FUSE` — `insfilterNonholonomic` object

object

`insfilterNonholonomic`, specified as an object.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`insfilter` | `insfilterNonholonomic`

Introduced in R2018b

residual

Residuals and residual covariances from direct state measurements for `insfilterNonholonomic`

Syntax

```
[res, resCov]= residual(FUSE,idx,measurement,measurementCovariance)
```

Description

`[res, resCov]= residual(FUSE,idx,measurement,measurementCovariance)` computes the residual, `res`, and the residual covariance, `resCov`, based on the direct state measurement and measurement covariance. The `measurement` maps directly to the states specified by indices, `idx`.

Input Arguments

`FUSE` — `insfilterNonholonomic` object
object

`insfilterNonholonomic`, specified as an object.

`idx` — State vector Index of measurement to correct
N-element vector of increasing integers in the range [1,16]

State vector index of measurement to correct, specified as an *N*-element vector of increasing integers in the range [1,16].

The state values represent:

State	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

Data Types: `single` | `double`

`measurement` — Direct measurement of state
N-element vector

Direct measurement of state, specified as a *N*-element vector. *N* is the number of elements of the index argument, `idx`.

`measurementCovariance` — Covariance of measurement
N-by-*N* matrix

Covariance of measurement, specified as an *N*-by-*N* matrix. *N* is the number of elements of the index argument, `idx`.

Output Arguments

`res` — Measurement residual

1-by-*N* vector of real values

Measurement residual, returned as a 1-by-*N* vector of real values.

`resCov` — Residual covariance

N-by-*N* matrix of real values

Residual covariance, returned as a *N*-by-*N* matrix of real values.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`insfilter` | `insfilterNonholonomic`

Introduced in R2020a

residualgps

Residuals and residual covariance from GPS measurements for `insfilterNonholonomic`

Syntax

```
[res,resCov] = residualgps(FUSE,position,positionCovariance)
[res,resCov] = residualgps(FUSE,position,positionCovariance,velocity,
velocityCovariance)
```

Description

`[res,resCov] = residualgps(FUSE,position,positionCovariance)` computes the residual, `res`, and the residual covariance, `resCov`, based on the GPS position measurement and covariance.

`[res,resCov] = residualgps(FUSE,position,positionCovariance,velocity,velocityCovariance)` computes the residual, `res`, and the residual covariance, `resCov`, based on the GPS position measurement and covariance.

Input Arguments

`FUSE` — `insfilterNonholonomic` object
object

`insfilterNonholonomic`, specified as an object.

`position` — Position of GPS receiver (LLA)
3-element row vector

Position of GPS receiver in geodetic latitude, longitude, and altitude (LLA) specified as a real finite 3-element row vector. Latitude and longitude are in degrees with north and east being positive. Altitude is in meters.

Data Types: `single` | `double`

`positionCovariance` — Position measurement covariance of GPS receiver (m^2)
3-by-3 matrix

Position measurement covariance of GPS receiver in m^2, specified as a 3-by-3 matrix.

Data Types: `single` | `double`

`velocity` — Velocity of GPS receiver in local NED coordinate system (m/s)
3-element row vector

Velocity of the GPS receiver in the local NED coordinate system in m/s, specified as a 3-element row vector.

Data Types: `single` | `double`

`velocityCovariance` — Velocity measurement covariance of GPS receiver (m/s^2)
3-by-3 matrix

Velocity measurement covariance of the GPS receiver in the local NED coordinate system in m/s^2, specified as a 3-by-3 matrix.

Data Types: `single` | `double`

Output Arguments

`res` — Position and course residual

1-by-3 vector of real values | 1-by-4 vector of real values

Position and course residual, returned as a 1-by-3 vector of real values the inputs only contain position information, and returned as a 1-by-4 vector of real values if the inputs also contain velocity information.

`resCov` — Residual covariance

3-by-3 matrix of real values | 4-by-4 matrix of real values

Residual covariance, returned as a 3-by-3 matrix of real values if the inputs only contain position information, and a 4-by-4 vector of real values if the inputs also contain velocity information.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`insfilterNonholonomic`

Introduced in R2020a

stateinfo

Display state vector information for `insfilterNonholonomic`

Syntax

```
stateinfo(FUSE)
info = stateinfo(FUSE)
```

Description

`stateinfo(FUSE)` displays the meaning of each index of the State property and the associated units.

`info = stateinfo(FUSE)` returns a structure with fields containing descriptions of the elements of the state vector of the filter, `FUSE`.

Examples

State information of `insfilterNonholonomic`

Create an `insfilterNonholonomic` object.

```
filter = insfilterErrorState;
```

Display the state information of the created filter.

```
stateinfo(filter)
```

```
States                            Units    Index
Orientation (quaternion parts)             1:4
Position (NAV)                    m        5:7
Velocity (NAV)                    m/s      8:10
Gyroscope Bias (XYZ)              rad/s    11:13
Accelerometer Bias (XYZ)          m/s²     14:16
Visual Odometry Scale                      17
```

Output the state information of the filter as a structure.

```
info = stateinfo(filter)
```

```
info = struct with fields:
            Orientation: [1 2 3 4]
               Position: [5 6 7]
               Velocity: [8 9 10]
          GyroscopeBias: [11 12 13]
      AccelerometerBias: [14 15 16]
    VisualOdometryScale: 17
```

Input Arguments

`FUSE` — `insfilterNonholonomic` object

object

`insfilterNonholonomic`, specified as an object.

Output Arguments

`info` — State information

structure

State information, returned as a structure with fields containing descriptions of the elements of the state vector of the filter. The values of each field are the corresponding indices of the state vector.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`insfilter` | `insfilterNonholonomic`

Introduced in R2018b

tune

Tune `insfilterNonholonomic` parameters to reduce estimation error

Syntax

```
tunedMeasureNoise = tune(filter,measureNoise,sensorData,groundTruth)
tunedMeasureNoise = tune( ___ ,config)
```

Description

`tunedMeasureNoise = tune(filter,measureNoise,sensorData,groundTruth)` adjusts the properties of the `insfilterNonholonomic` filter object, `filter`, and measurement noises to reduce the root-mean-squared (RMS) state estimation error between the fused sensor data and the ground truth. The function also returns the tuned measurement noise, `tunedMeasureNoise`. The function uses the property values in the filter and the measurement noise provided in the `measureNoise` structure as the initial estimate for the optimization algorithm.

`tunedMeasureNoise = tune( ___ ,config)` specifies the tuning configuration based on a `tunerconfig` object, `config`.

Examples

Tune `insfilterNonholonomic` to Optimize Pose Estimate

Load the recorded sensor data and ground truth data.

```
load('insfilterNonholonomicTuneData.mat');
```

Create tables for the sensor data and the truth data.

```
sensorData = table(Accelerometer, Gyroscope, ...
        GPSPosition, GPSVelocity);
groundTruth = table(Orientation, Position);
```

Create an `insfilterNonholonimic` filter object.

```
filter = insfilterNonholonomic('State', initialState, ...
        'StateCovariance', initialStateCovariance, ...
        'DecimationFactor', 1);
```

Create a tuner configuration object for the filter. Set the maximum number of iterations to 30.

```
config = tunerconfig('insfilterNonholonomic','MaxIterations',30);
```

Use the `tunernoise` function to obtain a set of initial sensor noises used in the filter.

```
measNoise = tunernoise('insfilterNonholonomic')
```

```
measNoise = struct with fields:
    GPSPositionNoise: 1
```

```
    GPSVelocityNoise: 1
```

Tune the filter and obtain the tuned measurement noise.

```
tunedNoise = tune(filter,measNoise,sensorData,groundTruth,config);
```

```
    Iteration    Parameter                       Metric
    _________    _________                       ______
    1            GyroscopeNoise                  3.4877
    1            AccelerometerNoise              3.3961
    1            GyroscopeBiasNoise              3.3961
    1            GyroscopeBiasDecayFactor        3.3961
    1            AccelerometerBiasNoise          3.3961
    1            AccelerometerBiasDecayFactor    3.3961
    1            ZeroVelocityConstraintNoise     3.3935
    1            GPSPositionNoise                3.2848
    1            GPSVelocityNoise                3.2798
    2            GyroscopeNoise                  3.2641
    2            AccelerometerNoise              3.1715
    2            GyroscopeBiasNoise              3.1715
    2            GyroscopeBiasDecayFactor        2.9661
    2            AccelerometerBiasNoise          2.9661
    2            AccelerometerBiasDecayFactor    2.9661
    2            ZeroVelocityConstraintNoise     2.9617
    2            GPSPositionNoise                2.8438
    2            GPSVelocityNoise                2.8384
    3            GyroscopeNoise                  2.8373
    3            AccelerometerNoise              2.7382
    3            GyroscopeBiasNoise              2.7382
    3            GyroscopeBiasDecayFactor        2.7382
    3            AccelerometerBiasNoise          2.7382
    3            AccelerometerBiasDecayFactor    2.7382
    3            ZeroVelocityConstraintNoise     2.7335
    3            GPSPositionNoise                2.6105
    3            GPSVelocityNoise                2.6045
    4            GyroscopeNoise                  2.6023
    4            AccelerometerNoise              2.5001
    4            GyroscopeBiasNoise              2.5001
    4            GyroscopeBiasDecayFactor        2.5001
    4            AccelerometerBiasNoise          2.5001
    4            AccelerometerBiasDecayFactor    2.5001
    4            ZeroVelocityConstraintNoise     2.4953
    4            GPSPositionNoise                2.3692
    4            GPSVelocityNoise                2.3626
    5            GyroscopeNoise                  2.3595
    5            AccelerometerNoise              2.2561
    5            GyroscopeBiasNoise              2.2561
    5            GyroscopeBiasDecayFactor        2.2508
    5            AccelerometerBiasNoise          2.2508
    5            AccelerometerBiasDecayFactor    2.2508
    5            ZeroVelocityConstraintNoise     2.2469
    5            GPSPositionNoise                2.1265
    5            GPSVelocityNoise                2.1191
    6            GyroscopeNoise                  2.1148
    6            AccelerometerNoise              2.0150
    6            GyroscopeBiasNoise              2.0150
    6            GyroscopeBiasDecayFactor        2.0150
```

```
6           AccelerometerBiasNoise          2.0150
6           AccelerometerBiasDecayFactor    2.0150
6           ZeroVelocityConstraintNoise     2.0116
6           GPSPositionNoise                1.8970
6           GPSVelocityNoise                1.8888
7           GyroscopeNoise                  1.8847
7           AccelerometerNoise              1.7921
7           GyroscopeBiasNoise              1.7921
7           GyroscopeBiasDecayFactor        1.7845
7           AccelerometerBiasNoise          1.7845
7           AccelerometerBiasDecayFactor    1.7845
7           ZeroVelocityConstraintNoise     1.7815
7           GPSPositionNoise                1.6794
7           GPSVelocityNoise                1.6708
8           GyroscopeNoise                  1.6679
8           AccelerometerNoise              1.5886
8           GyroscopeBiasNoise              1.5886
8           GyroscopeBiasDecayFactor        1.5866
8           AccelerometerBiasNoise          1.5866
8           AccelerometerBiasDecayFactor    1.5866
8           ZeroVelocityConstraintNoise     1.5850
8           GPSPositionNoise                1.5057
8           GPSVelocityNoise                1.4965
9           GyroscopeNoise                  1.4950
9           AccelerometerNoise              1.4364
9           GyroscopeBiasNoise              1.4364
9           GyroscopeBiasDecayFactor        1.4364
9           AccelerometerBiasNoise          1.4364
9           AccelerometerBiasDecayFactor    1.4364
9           ZeroVelocityConstraintNoise     1.4355
9           GPSPositionNoise                1.3894
9           GPSVelocityNoise                1.3790
10          GyroscopeNoise                  1.3773
10          AccelerometerNoise              1.3422
10          GyroscopeBiasNoise              1.3422
10          GyroscopeBiasDecayFactor        1.3421
10          AccelerometerBiasNoise          1.3421
10          AccelerometerBiasDecayFactor    1.3421
10          ZeroVelocityConstraintNoise     1.3399
10          GPSPositionNoise                1.3319
10          GPSVelocityNoise                1.3190
11          GyroscopeNoise                  1.3159
11          AccelerometerNoise              1.3102
11          GyroscopeBiasNoise              1.3102
11          GyroscopeBiasDecayFactor        1.3100
11          AccelerometerBiasNoise          1.3100
11          AccelerometerBiasDecayFactor    1.3100
11          ZeroVelocityConstraintNoise     1.3069
11          GPSPositionNoise                1.2964
11          GPSVelocityNoise                1.2762
12          GyroscopeNoise                  1.2740
12          AccelerometerNoise              1.2655
12          GyroscopeBiasNoise              1.2655
12          GyroscopeBiasDecayFactor        1.2641
12          AccelerometerBiasNoise          1.2641
12          AccelerometerBiasDecayFactor    1.2641
12          ZeroVelocityConstraintNoise     1.2631
12          GPSPositionNoise                1.2511
```

```
12          GPSVelocityNoise               1.2198
13          GyroscopeNoise                 1.2184
13          AccelerometerNoise             1.2058
13          GyroscopeBiasNoise             1.2058
13          GyroscopeBiasDecayFactor       1.2029
13          AccelerometerBiasNoise         1.2029
13          AccelerometerBiasDecayFactor   1.2029
13          ZeroVelocityConstraintNoise    1.2029
13          GPSPositionNoise               1.1874
13          GPSVelocityNoise               1.1408
14          GyroscopeNoise                 1.1403
14          AccelerometerNoise             1.1236
14          GyroscopeBiasNoise             1.1236
14          GyroscopeBiasDecayFactor       1.1186
14          AccelerometerBiasNoise         1.1186
14          AccelerometerBiasDecayFactor   1.1186
14          ZeroVelocityConstraintNoise    1.1183
14          GPSPositionNoise               1.0975
14          GPSVelocityNoise               1.0348
15          GyroscopeNoise                 1.0347
15          AccelerometerNoise             1.0155
15          GyroscopeBiasNoise             1.0155
15          GyroscopeBiasDecayFactor       1.0081
15          AccelerometerBiasNoise         1.0081
15          AccelerometerBiasDecayFactor   1.0081
15          ZeroVelocityConstraintNoise    1.0076
15          GPSPositionNoise               0.9813
15          GPSVelocityNoise               0.9078
16          GyroscopeNoise                 0.9074
16          AccelerometerNoise             0.8926
16          GyroscopeBiasNoise             0.8926
16          GyroscopeBiasDecayFactor       0.8823
16          AccelerometerBiasNoise         0.8823
16          AccelerometerBiasDecayFactor   0.8823
16          ZeroVelocityConstraintNoise    0.8815
16          GPSPositionNoise               0.8526
16          GPSVelocityNoise               0.7926
17          GyroscopeNoise                 0.7920
17          AccelerometerNoise             0.7870
17          GyroscopeBiasNoise             0.7870
17          GyroscopeBiasDecayFactor       0.7742
17          AccelerometerBiasNoise         0.7742
17          AccelerometerBiasDecayFactor   0.7742
17          ZeroVelocityConstraintNoise    0.7730
17          GPSPositionNoise               0.7665
17          GPSVelocityNoise               0.7665
18          GyroscopeNoise                 0.7662
18          AccelerometerNoise             0.7638
18          GyroscopeBiasNoise             0.7638
18          GyroscopeBiasDecayFactor       0.7495
18          AccelerometerBiasNoise         0.7495
18          AccelerometerBiasDecayFactor   0.7495
18          ZeroVelocityConstraintNoise    0.7482
18          GPSPositionNoise               0.7482
18          GPSVelocityNoise               0.7475
19          GyroscopeNoise                 0.7474
19          AccelerometerNoise             0.7474
19          GyroscopeBiasNoise             0.7474
```

```
19          GyroscopeBiasDecayFactor       0.7474
19          AccelerometerBiasNoise         0.7474
19          AccelerometerBiasDecayFactor   0.7474
19          ZeroVelocityConstraintNoise    0.7453
19          GPSPositionNoise               0.7416
19          GPSVelocityNoise               0.7382
20          GyroscopeNoise                 0.7378
20          AccelerometerNoise             0.7370
20          GyroscopeBiasNoise             0.7370
20          GyroscopeBiasDecayFactor       0.7370
20          AccelerometerBiasNoise         0.7370
20          AccelerometerBiasDecayFactor   0.7370
20          ZeroVelocityConstraintNoise    0.7345
20          GPSPositionNoise               0.7345
20          GPSVelocityNoise               0.7345
21          GyroscopeNoise                 0.7334
21          AccelerometerNoise             0.7334
21          GyroscopeBiasNoise             0.7334
21          GyroscopeBiasDecayFactor       0.7334
21          AccelerometerBiasNoise         0.7334
21          AccelerometerBiasDecayFactor   0.7334
21          ZeroVelocityConstraintNoise    0.7306
21          GPSPositionNoise               0.7279
21          GPSVelocityNoise               0.7268
22          GyroscopeNoise                 0.7248
22          AccelerometerNoise             0.7247
22          GyroscopeBiasNoise             0.7247
22          GyroscopeBiasDecayFactor       0.7234
22          AccelerometerBiasNoise         0.7234
22          AccelerometerBiasDecayFactor   0.7234
22          ZeroVelocityConstraintNoise    0.7207
22          GPSPositionNoise               0.7206
22          GPSVelocityNoise               0.7170
23          GyroscopeNoise                 0.7138
23          AccelerometerNoise             0.7134
23          GyroscopeBiasNoise             0.7134
23          GyroscopeBiasDecayFactor       0.7134
23          AccelerometerBiasNoise         0.7134
23          AccelerometerBiasDecayFactor   0.7134
23          ZeroVelocityConstraintNoise    0.7122
23          GPSPositionNoise               0.7122
23          GPSVelocityNoise               0.7122
24          GyroscopeNoise                 0.7081
24          AccelerometerNoise             0.7080
24          GyroscopeBiasNoise             0.7080
24          GyroscopeBiasDecayFactor       0.7080
24          AccelerometerBiasNoise         0.7080
24          AccelerometerBiasDecayFactor   0.7080
24          ZeroVelocityConstraintNoise    0.7080
24          GPSPositionNoise               0.7080
24          GPSVelocityNoise               0.7072
25          GyroscopeNoise                 0.7009
25          AccelerometerNoise             0.7009
25          GyroscopeBiasNoise             0.7009
25          GyroscopeBiasDecayFactor       0.7007
25          AccelerometerBiasNoise         0.7007
25          AccelerometerBiasDecayFactor   0.7007
25          ZeroVelocityConstraintNoise    0.7005
```

```
    25          GPSPositionNoise               0.6997
    25          GPSVelocityNoise               0.6997
    26          GyroscopeNoise                 0.6912
    26          AccelerometerNoise             0.6906
    26          GyroscopeBiasNoise             0.6906
    26          GyroscopeBiasDecayFactor       0.6906
    26          AccelerometerBiasNoise         0.6906
    26          AccelerometerBiasDecayFactor   0.6906
    26          ZeroVelocityConstraintNoise    0.6896
    26          GPSPositionNoise               0.6896
    26          GPSVelocityNoise               0.6896
    27          GyroscopeNoise                 0.6840
    27          AccelerometerNoise             0.6831
    27          GyroscopeBiasNoise             0.6831
    27          GyroscopeBiasDecayFactor       0.6831
    27          AccelerometerBiasNoise         0.6831
    27          AccelerometerBiasDecayFactor   0.6831
    27          ZeroVelocityConstraintNoise    0.6818
    27          GPSPositionNoise               0.6816
    27          GPSVelocityNoise               0.6816
    28          GyroscopeNoise                 0.6816
    28          AccelerometerNoise             0.6809
    28          GyroscopeBiasNoise             0.6809
    28          GyroscopeBiasDecayFactor       0.6809
    28          AccelerometerBiasNoise         0.6809
    28          AccelerometerBiasDecayFactor   0.6809
    28          ZeroVelocityConstraintNoise    0.6804
    28          GPSPositionNoise               0.6802
    28          GPSVelocityNoise               0.6802
    29          GyroscopeNoise                 0.6793
    29          AccelerometerNoise             0.6785
    29          GyroscopeBiasNoise             0.6785
    29          GyroscopeBiasDecayFactor       0.6785
    29          AccelerometerBiasNoise         0.6785
    29          AccelerometerBiasDecayFactor   0.6785
    29          ZeroVelocityConstraintNoise    0.6778
    29          GPSPositionNoise               0.6773
    29          GPSVelocityNoise               0.6773
    30          GyroscopeNoise                 0.6773
    30          AccelerometerNoise             0.6769
    30          GyroscopeBiasNoise             0.6769
    30          GyroscopeBiasDecayFactor       0.6769
    30          AccelerometerBiasNoise         0.6769
    30          AccelerometerBiasDecayFactor   0.6769
    30          ZeroVelocityConstraintNoise    0.6769
    30          GPSPositionNoise               0.6769
    30          GPSVelocityNoise               0.6769
```

Fuse the sensor data using the tuned filter. Obtain estimated pose and orientation.

```
N = size(sensorData,1);
qEstTuned = quaternion.zeros(N,1);
posEstTuned = zeros(N,3);
for ii=1:N
    predict(filter,Accelerometer(ii,:),Gyroscope(ii,:));
    if all(~isnan(GPSPosition(ii,1)))
        fusegps(filter, GPSPosition(ii,:), ...
            tunedNoise.GPSPositionNoise,GPSVelocity(ii,:), ...
```

```
            tunedNoise.GPSVelocityNoise);
    end
    [posEstTuned(ii,:),qEstTuned(ii,:)] = pose(filter);
end
```

Compute the RMS errors.

```
orientationErrorTuned = rad2deg(dist(qEstTuned,Orientation));
rmsOrientationErrorTuned = sqrt(mean(orientationErrorTuned.^2))
```

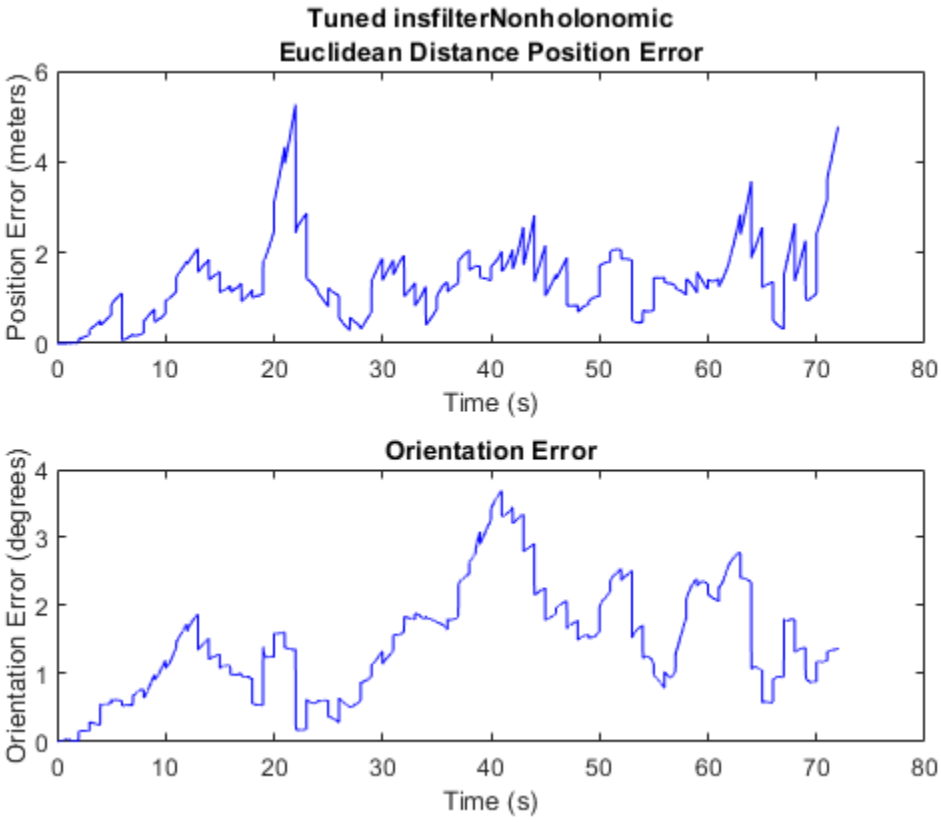
```
rmsOrientationErrorTuned = 1.6857
```

```
positionErrorTuned = sqrt(sum((posEstTuned-Position).^2,2));
rmsPositionErrorTuned = sqrt(mean(positionErrorTuned.^2))
```

```
rmsPositionErrorTuned = 1.6667
```

Visualize the results.

```
figure;
t = (0:N-1)./filter.IMUSampleRate;
subplot(2,1,1)
plot(t,positionErrorTuned,'b');
title("Tuned insfilterNonholonomic" + newline + ...
    "Euclidean Distance Position Error")
xlabel('Time (s)');
ylabel('Position Error (meters)')
subplot(2,1,2)
plot(t,orientationErrorTuned,'b');
title("Orientation Error")
xlabel('Time (s)');
ylabel('Orientation Error (degrees)');
```



Input Arguments

`filter` — Filter object

`infilterAsync` object

Filter object, specified as an `insfilterNonholonomic` object.

`measureNoise` — Measurement noise

structure

Measurement noise, specified as a structure. The function uses the measurement noise input as the initial guess for tuning the measurement noise. The structure must contain these fields:

Field name	Description
`GPSPositionNoise`	Variance of GPS position noise, specified as a scalar in m^2
`GPSVelocityNoise`	Variance of GPS velocity noise, specified as a scalar in $(m/s)^2$

Data Types: `struct`

`sensorData` — Sensor data

`table`

Sensor data, specified as a `table`. In each row, the sensor data is specified as:

- `Accelerometer` — Accelerometer data, specified as a 1-by-3 vector of scalars in m^2/s.
- `Gyroscope` — Gyroscope data, specified as a 1-by-3 vector of scalars in rad/s.
- `GPSPosition` — GPS position data, specified as a 1-by-3 vector of scalars in meters.
- `GPSVelocity` — GPS velocity data, specified as a 1-by-3 vector of scalars in m/s.

If the GPS sensor does not produce complete measurements, specify the corresponding entry for `GPSPosition` and/or `GPSVelocity` as `NaN`. If you set the `Cost` property of the tuner configuration input, `config`, to `Custom`, then you can use other data types for the `sensorData` input based on your choice.

Data Types: `table`

`groundTruth` — Ground truth data
`table`

Ground truth data, specified as a table. In each row, the table can optionally contain any of these variables:

- `Orientation` — Orientation from the navigation frame to the body frame, specified as a `quaternion` or a 3-by-3 rotation matrix.
- `Position` — Position in navigation frame, specified as a 1-by-3 vector of scalars in meters.
- `Velocity` — Velocity in navigation frame, specified as a 1-by-3 vector of scalars in m/s.
- `GyroscopeBias` — Gyroscope delta angle bias in body frame, specified as a 1-by-3 vector of scalars in rad/s.
- `AccelerometerBias` — Accelerometer delta angle bias in body frame, specified as a 1-by-3 vector of scalars in m^2/s.

The function processes each row of the `sensorData` and `groundTruth` tables sequentially to calculate the state estimate and RMS error from the ground truth. State variables not present in `groundTruth` input are ignored for the comparison. The `sensorData` and the `groundTruth` tables must have the same number of rows.

If you set the `Cost` property of the tuner configuration input, `config`, to `Custom`, then you can use other data types for the `groundTruth` input based on your choice.

Data Types: `table`

`config` — Tuner configuration
`tunerconfig` object

Tuner configuration, specified as a `tunerconfig` object.

Output Arguments

`tunedMeasureNoise` — Tuned measurement noise
structure

Tuned measurement noise, returned as a structure. The structure contains these fields.

Field name	Description
`GPSPositionNoise`	Variance of GPS position noise, specified as a scalar in m^2
`GPSVelocityNoise`	Variance of GPS velocity noise, specified as a scalar in $(m/s)^2$

Data Types: `struct`

References

[1] Abbeel, Pieter, et al. “Discriminative Training of Kalman Filters.” Robotics: Science and Systems I, Robotics: Science and Systems Foundation, 2005. DOI.org (Crossref), doi:10.15607/RSS.2005.I.038.

See Also

`tunerconfig` | `tunernoise` | `insfilterNonholonomic`

Introduced in R2020b

occupancyMap

Create occupancy map with probabilistic values

Description

`occupancyMap` creates a 2-D occupancy grid map object. 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 probability that the cell contains an obstacle. Values close to 0 represent a high probability that the cell is not occupied and obstacle free.

Occupancy maps are used in navigation algorithms such as path planning (see `plannerRRT`). They are also used in mapping applications for finding collision-free paths, performing collision avoidance, and calculating localization (see `monteCarloLocalization`). You can modify your occupancy map to fit your specific application.

The `occupancyMap` objects support local coordinates, world coordinates, and grid indices. The first grid location with index `(1,1)` begins in the top-left corner of the grid.

Use the `occupancyMap` class to create 2-D maps of an environment with probability values representing different obstacles in your world. You can specify exact probability values of cells or include observations from sensors such as laser scanners.

Probability values are stored using a binary Bayes filter to estimate the occupancy of each grid cell. A log-odds representation is used, with values stored as `int16` to reduce the map storage size and allow for real-time applications.

Creation

Syntax

```
map = occupancyMap(width,height)
map = occupancyMap(width,height,resolution)
map = occupancyMap(rows,cols,resolution,'grid')
map = occupancyMap(p)
map = occupancyMap(p,resolution)
map = occupancyMap(sourcemap)
map = occupancyMap(sourcemap,resolution)
```

Description

`map = occupancyMap(width,height)` creates a 2-D occupancy map object representing a world space of `width` and `height` in meters. The default grid resolution is 1 cell per meter.

`map = occupancyMap(width,height,resolution)` creates an occupancy map with a specified grid resolution in cells per meter. `resolution` sets the “Resolution” on page 2-0 property.

`map = occupancyMap(rows,cols,resolution,'grid')` creates an occupancy map with the specified number of rows and columns and with the resolution in cells per meter. The values of `rows` and `cols` sets the "GridSize" on page 2-0 property.

`map = occupancyMap(p)` creates an occupancy map from the values in matrix `p`. The grid size matches the size of the matrix, with each cell probability value interpreted from the matrix location.

`map = occupancyMap(p,resolution)` creates an occupancy map from the specified matrix and resolution in cells per meter.

`map = occupancyMap(sourcemap)` creates an object using values from another `occupancyMap` object.

`map = occupancyMap(sourcemap,resolution)` creates an object using values from another `occupancyMap` object, but resamples the matrix to have the specified resolution.

Input Arguments

`width` — Map width
scalar

Map width, specified as a scalar in meters.

`height` — Map height
scalar

Map height, specified as a scalar in meters.

`resolution` — Grid resolution
`1` (default) | scalar

Grid resolution, specified as a scalar in cells per meter.

`rows` — Number of rows in grid
positive scalar integer

Number of rows in grid, specified as a positive scalar integer.

`cols` — Number of columns in grid
positive scalar integer

Number of columns in grid, specified as a positive scalar integer.

`p` — Input occupancy grid
matrix of probability values from `0` to `1`

Input occupancy grid, specified as a matrix of probability values from `0` to `1`. The size of the grid matches the size of the matrix. Each matrix element corresponds to the probability of the grid cell location being occupied. 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.

`sourcemap` — Occupancy map object
`occupancyMap` object

Occupancy map object, specified as a `occupancyMap` object.

Properties

`FreeThreshold` — Threshold below which cells are considered obstacle-free
scalar between `0` and `1`

Threshold below which cells are considered obstacle-free, specified as a scalar between `0` and `1` inclusive. Cells with probability values below this threshold are considered obstacle free. This property also defines the free locations for path planning when using objects like `plannerRRT`.

Data Types: `double`

`OccupiedThreshold` — Threshold above which cells are considered occupied
scalar

Threshold above which cells are considered occupied, specified as a scalar. Cells with probability values above this threshold are considered occupied.

Data Types: `double`

`ProbabilitySaturation` — Saturation limits for probability
`[0.001 0.999]` (default) | two-element real-valued vector

Saturation limits for probability, specified as a 1-by-2 real-valued vector representing the minimum and maximum values, in that order. Values above or below these saturation values are set to the minimum and maximum values. This property reduces oversaturating of cells when incorporating multiple observations.

Data Types: `double`

`GridSize` — Number of rows and columns in grid
two-element integer-valued vector

This property is read-only.

Number of rows and columns in grid, stored as a 1-by-2 real-valued vector representing the number of rows and columns, in that order.

Data Types: `double`

`Resolution` — Grid resolution
`1` (default) | scalar

This property is read-only.

Grid resolution, stored as a scalar in cells per meter representing the number and size of grid locations.

Data Types: `double`

`XLocalLimits` — Minimum and maximum values of *x*-coordinates in local frame
two-element vector

This property is read-only.

Minimum and maximum values of *x*-coordinates in local frame, stored as a two-element horizontal vector of the form `[min max]`. Local frame is defined by `LocalOriginInWorld` property.

Data Types: `double`

`YLocalLimits` — Minimum and maximum values of *y*-coordinates in local frame
two-element vector

This property is read-only.

Minimum and maximum values of *y*-coordinates in local frame, stored as a two-element horizontal vector of the form `[min max]`. Local frame is defined by `LocalOriginInWorld` property.

Data Types: `double`

`XWorldLimits` — Minimum and maximum world range values of *x*-coordinates
two-element vector

This property is read-only.

Minimum and maximum world range values of *x*-coordinates, stored as a 1-by-2 vector representing the minimum and maximum values, in that order.

Data Types: `double`

`YWorldLimits` — Minimum and maximum world range values of *y*-coordinates
two-element vector

This property is read-only.

Minimum and maximum world range values of *y*-coordinates, stored as a 1-by-2 vector representing the minimum and maximum values, in that order.

Data Types: `double`

`GridLocationInWorld` — `[x y]` world coordinates of grid
`[0 0]` (default) | two-element vector

[*x*,*y*] world coordinates of the bottom-left corner of the grid, specified as a 1-by-2 vector.

Data Types: `double`

`LocalOriginInWorld` — Location of the local frame in world coordinates
`[0 0]` (default) | two-element vector | `[xWorld yWorld]`

Location of the origin of the local frame in world coordinates, specified as a two-element vector, `[xLocal yLocal]`. Use the `move` function to shift the local frame as your vehicle moves.

Data Types: `double`

`GridOriginInLocal` — Location of the grid in local coordinates
`[0 0]` (default) | two-element vector | `[xLocal yLocal]`

Location of the bottom-left corner of the grid in local coordinates, specified as a two-element vector, `[xLocal yLocal]`.

Data Types: `double`

`DefaultValue` — Default value for unspecified map locations
`0.5` (default) | scalar between `0` and `1`

Default value for unspecified map locations including areas outside the map, specified as a scalar between `0` and `1` inclusive.

Data Types: `double`

Object Functions

checkOccupancy	Check locations for free, occupied, or unknown values
copy	Create copy of occupancy grid
getOccupancy	Get occupancy value of locations
grid2local	Convert grid indices to local coordinates
grid2world	Convert grid indices to world coordinates
inflate	Inflate each occupied grid location
insertRay	Insert ray from laser scan observation
local2grid	Convert local coordinates to grid indices
local2world	Convert local coordinates to world coordinates
move	Move map in world frame
occupancyMatrix	Convert occupancy grid to double matrix
raycast	Compute cell indices along a ray
rayIntersection	Find intersection points of rays and occupied map cells
setOccupancy	Set occupancy value of locations
show	Show grid values in a figure
syncWith	Sync map with overlapping map
updateOccupancy	Integrate probability observations at locations
world2grid	Convert world coordinates to grid indices
world2local	Convert world coordinates to local coordinates

Examples

Insert Laser Scans into Occupancy Map

Create an empty occupancy grid map.

```
map = occupancyMap(10,10,20);
```

Input pose of the vehicle, ranges, angles, and the maximum range of the laser scan.

```
pose = [5,5,0];
ranges = 3*ones(100,1);
angles = linspace(-pi/2,pi/2,100);
maxrange = 20;
```

Create a `lidarScan` object with the specified ranges and angles.

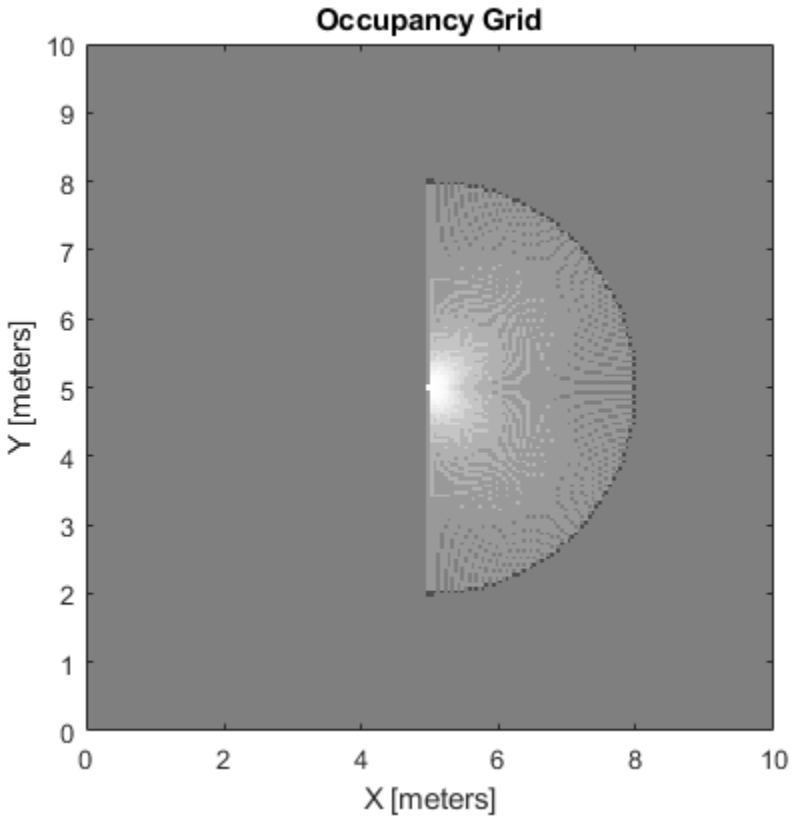
```
scan = lidarScan(ranges,angles);
```

Insert the laser scan data into the occupancy map.

```
insertRay(map,pose,scan,maxrange);
```

Show the map to see the results of inserting the laser scan.

```
show(map)
```



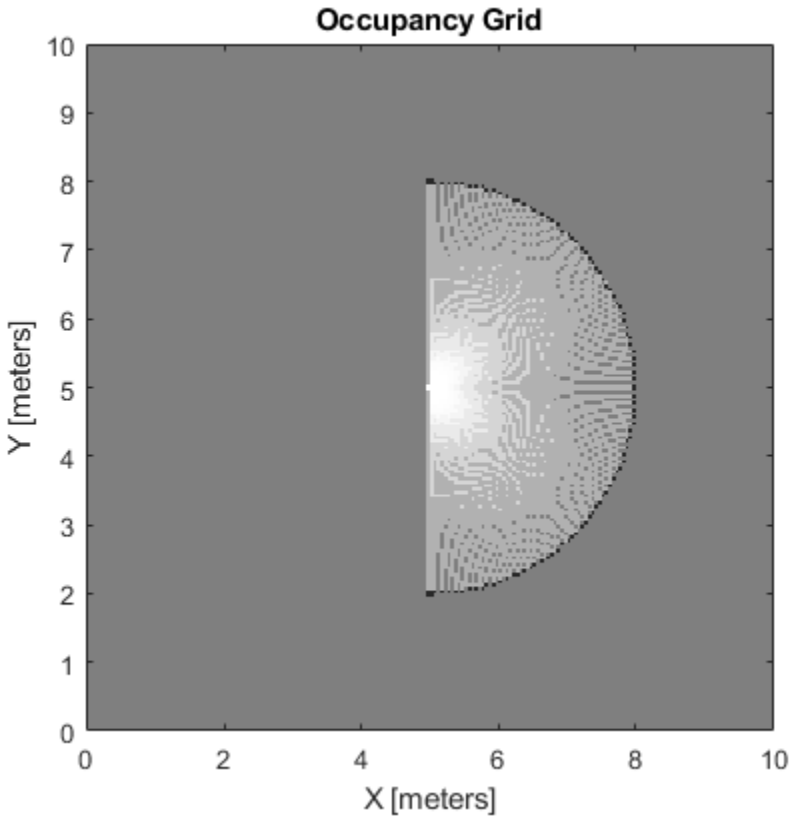
Check the occupancy of the spot directly in front of the vehicle.

```
getOccupancy(map,[8 5])
```

```
ans = 0.7000
```

Add a second reading and view the update to the occupancy values. The additional reading increases the confidence in the readings. The free and occupied values become more distinct.

```
insertRay(map,pose,scan,maxrange);
show(map)
```




```
getOccupancy(map,[8 5])
```

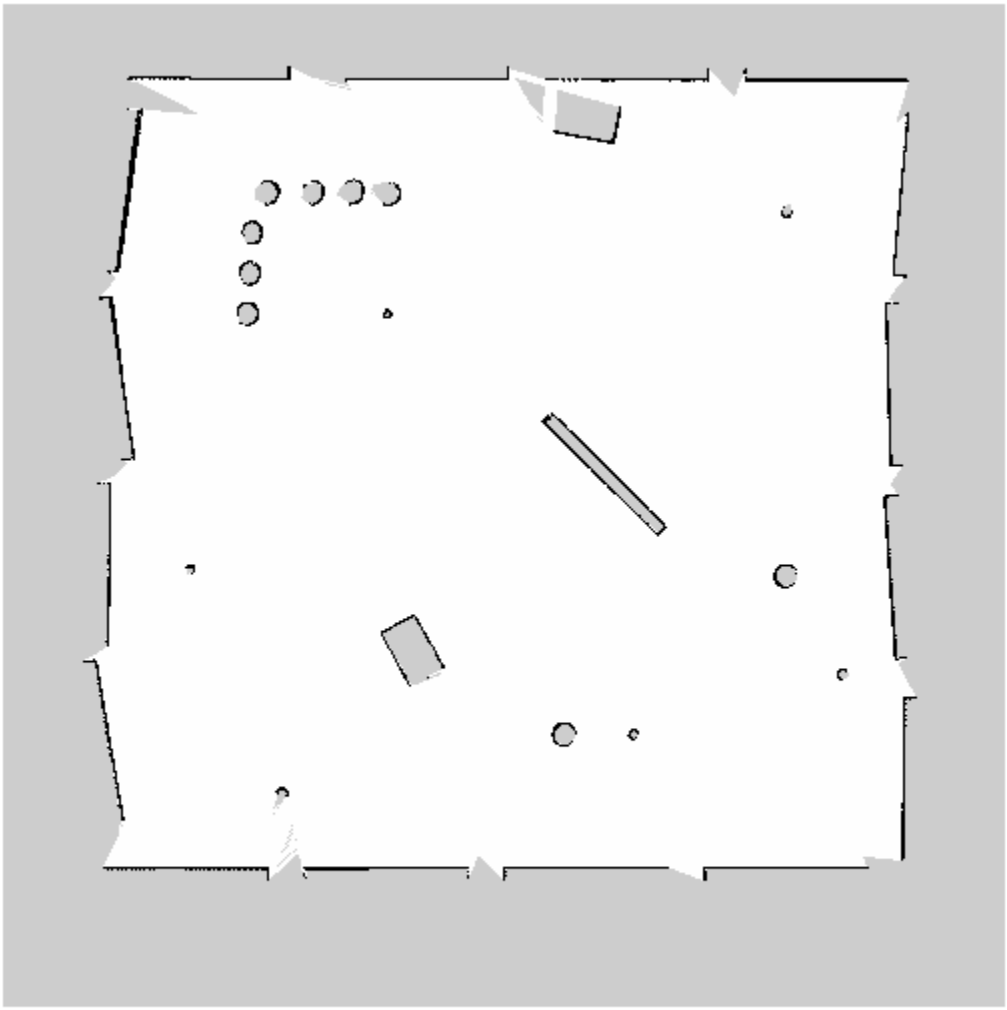
```
ans = 0.8448
```

Convert PGM Image to Map

Convert a portable graymap (PGM) file containing a ROS map into an `occupancyMap` for use in MATLAB.

Import the image using `imread`. Crop the image to the playpen area.

```
image = imread('playpen_map.pgm');
imageCropped = image(750:1250,750:1250);
imshow(imageCropped)
```

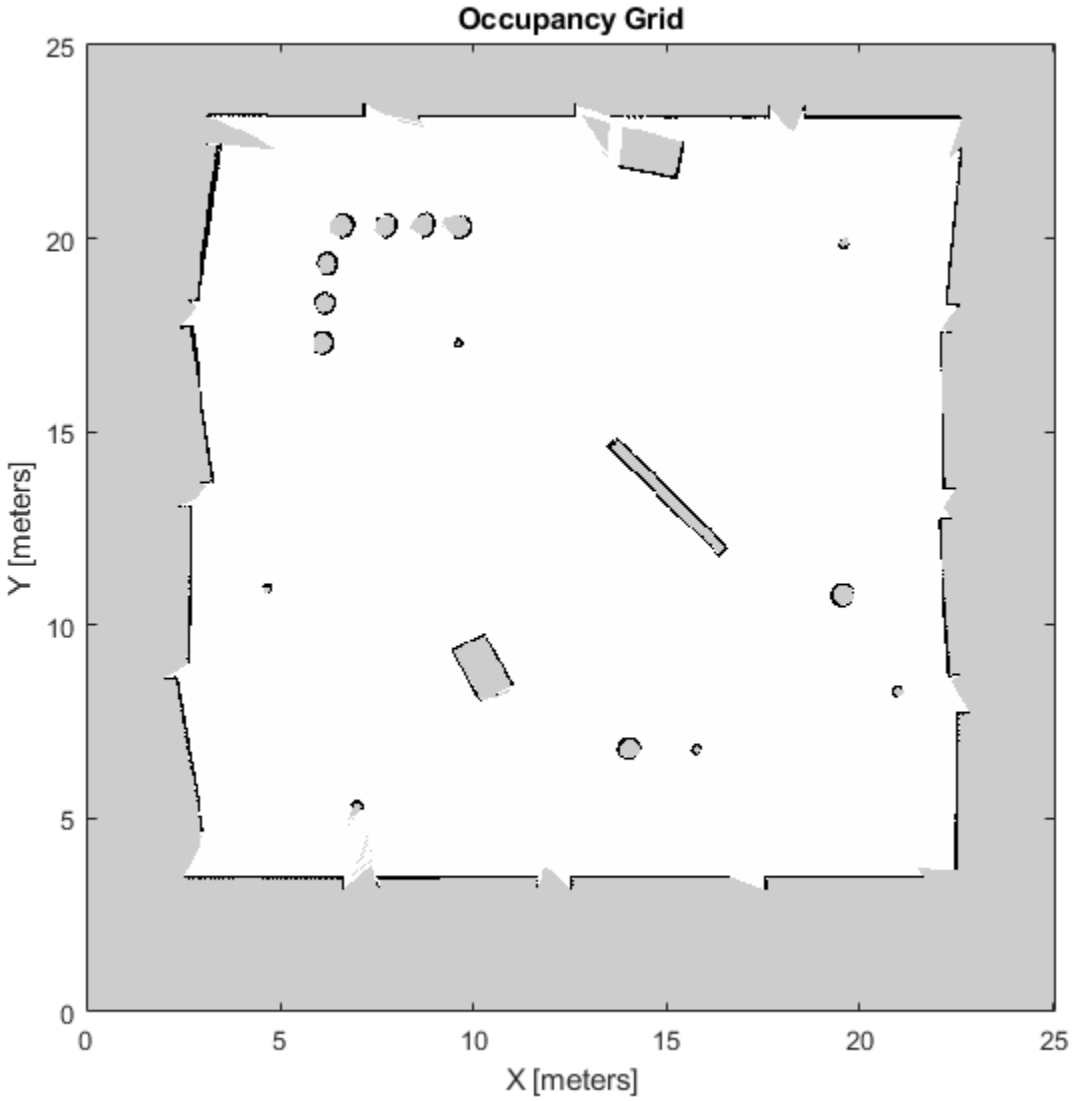


PGM values are expressed from 0 to 255 as `uint8`. Normalize these values by converting the cropped image to `double` and dividing each cell by 255. This image shows obstacles as values close to 0. Subtract the normalized image from 1 to get occupancy values with 1 representing occupied space.

```
imageNorm = double(imageCropped)/255;
imageOccupancy = 1 - imageNorm;
```

Create the `occupancyMap` object using an adjusted map image. The imported map resolution is 20 cells per meter.

```
map = occupancyMap(imageOccupancy,20);
show(map)
```



Limitations

Occupancy values have a limited resolution of ±0.001. The values are stored as `int16` using a log-odds representation. This data type limits resolution, but saves memory when storing large maps in MATLAB. When calling `setOccupancy` and then `getOccupancy`, the value returned might not equal the value you set. For more information, see the log-odds representations section in “Occupancy Grids”.

If memory size is a limitation, consider using `binaryOccupancyMap` instead. The binary occupancy map uses less memory with binary values, but still works with Navigation Toolbox™ algorithms and other applications.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`binaryOccupancyMap` | `mobileRobotPRM` | `controllerPurePursuit` | `rosReadOccupancyGrid` | `rosWriteOccupancyGrid`

Topics

“Create Egocentric Occupancy Maps Using Range Sensors”
“Build Occupancy Map from Lidar Scans and Poses”
“Occupancy Grids”

Introduced in R2019b

checkOccupancy

Check locations for free, occupied, or unknown values

Syntax

```
iOccval = checkOccupancy(map,xy)
iOccval = checkOccupancy(map,xy,'local')
iOccval = checkOccupancy(map,ij,'grid')
[iOccval,validPts] = checkOccupancy( ___ )

occMatrix = checkOccupancy(map)
occMatrix = checkOccupancy(map,bottomLeft,matSize)
occMatrix = checkOccupancy(map,bottomLeft,matSize,'local')
occMatrix = checkOccupancy(map,topLeft,matSize,'grid')
```

Description

`iOccval = checkOccupancy(map,xy)` returns an array of occupancy values at the `xy` locations. Each row is a separate `xy` location in the grid. Occupancy values can be obstacle free (`0`), occupied (`1`), or unknown (`–1`) based on the `OccupiedThreshold` and `FreeThreshold` properties of the `map` object.

`iOccval = checkOccupancy(map,xy,'local')` returns an array of occupancy values at the `xy` locations in the local frame. The local frame is based on the `LocalOriginInWorld` property of the `map`.

`iOccval = checkOccupancy(map,ij,'grid')` specifies `ij` grid cell indices instead of `xy` locations.

`[iOccval,validPts] = checkOccupancy( ___ )` also outputs an `n`-element vector of logical values indicating whether input coordinates are within the map limits.

`occMatrix = checkOccupancy(map)` returns a matrix that contains the occupancy status of each location. Obstacle-free cells return `0`, occupied cells return `1`. Unknown locations, including outside the map, return `-1`.

`occMatrix = checkOccupancy(map,bottomLeft,matSize)` returns a matrix of occupancy values by specifying the bottom-left corner location in world coordinates and the matrix size in meters.

`occMatrix = checkOccupancy(map,bottomLeft,matSize,'local')` returns a matrix of occupancy values by specifying the bottom-left corner location in local coordinates and the matrix size in meters.

`occMatrix = checkOccupancy(map,topLeft,matSize,'grid')` returns a matrix of occupancy values by specifying the top-left corner location in grid coordinates and the grid size.

Examples

Get Occupancy Values and Check Occupancy Status

Access occupancy values and check their occupancy status based on the occupied and free thresholds of the `occupancyMap` object.

Create a matrix and populate it with values. Use this matrix to create an occupancy map.

```
p = 0.5*ones(20,20);
p(11:20,11:20) = 0.75*ones(10,10);
map = occupancyMap(p,10);
```

Get the occupancy of different locations and check their occupancy statuses. The occupancy status returns 0 for free space and 1 for occupied space. Unknown values return –1.

```
pocc = getOccupancy(map,[1.5 1])
```

```
pocc = 0.7500
```

```
occupied = checkOccupancy(map,[1.5 1])
```

```
occupied = 1
```

```
pocc2 = getOccupancy(map,[5 5],'grid')
```

```
pocc2 = 0.5000
```

```
occupied2 = checkOccupancy(map,[5 5],'grid')
```

```
occupied2 = -1
```

Input Arguments

`map` — Map representation

`occupancyMap` object

Map representation, specified as a `occupancyMap` object. This object represents the environment of the vehicle. The object contains a matrix grid with values 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.

`xy` — World coordinates

n-by-2 matrix

World coordinates, specified as an *n*-by-2 matrix of `[x y]` pairs, where *n* is the number of world coordinates.

Data Types: `double`

`ij` — Grid positions

n-by-2 matrix

Grid positions, specified as an *n*-by-2 matrix of `[i j]` pairs in `[rows cols]` format, where *n* is the number of grid positions.

Data Types: `double`

`bottomLeft` — Location of output matrix in world or local
two-element vector | `[xCoord yCoord]`

Location of bottom left corner of output matrix in world or local coordinates, specified as a two-element vector, `[xCoord yCoord]`. Location is in world or local coordinates based on syntax.

Data Types: `double`

`matSize` — Output matrix size
two-element vector | `[xLength yLength]` | `[gridRow gridCol]`

Output matrix size, specified as a two-element vector, `[xLength yLength]`, or `[gridRow gridCol]`. Size is in world, local, or grid coordinates based on syntax.

Data Types: `double`

`topLeft` — Location of grid
two-element vector | `[iCoord jCoord]`

Location of top left corner of grid, specified as a two-element vector, `[iCoord jCoord]`.

Data Types: `double`

Output Arguments

`iOccval` — Interpreted occupancy values
n-by-1 column vector

Interpreted occupancy values, returned as an *n*-by-1 column vector equal in length to `xy` or `ij`.

Occupancy values can be obstacle free (0), occupied (1), or unknown (–1). These values are determined from the actual probability values and the `OccupiedThreshold` and `FreeThreshold` properties of the `map` object.

`validPts` — Valid map locations
n-by-1 column vector

Valid map locations, returned as an *n*-by-1 column vector equal in length to `xy` or `ij`. Locations inside the map return a value of 1. Locations outside the map limits return a value of 0.

`occMatrix` — Matrix of occupancy values
matrix

Matrix of occupancy values, returned as matrix with size equal to `matSize` or the size of your `map`. Occupancy values can be obstacle free (0), occupied (1), or unknown (-1).

See Also
`occupancyMap` | `getOccupancy` | `binaryOccupancyMap`

Introduced in R2019b

copy

Create copy of occupancy grid

Syntax

```
copyMap = copy(map)
```

Description

`copyMap = copy(map)` creates a deep copy of the `occupancyMap` object with the same properties.

Examples

Copy Occupancy Grid Map

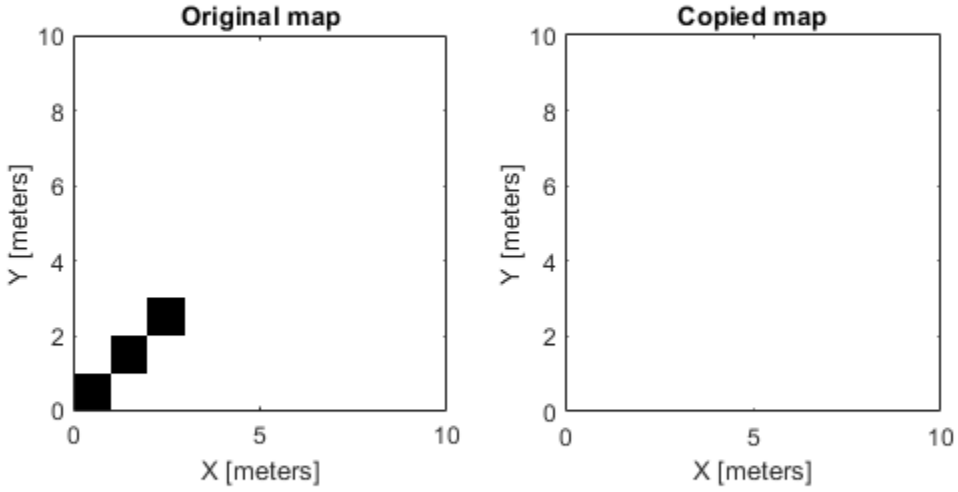
Copy an occupancy grid map object. Once copied, the original object can be modified without affecting the copied map.

Create an occupancy grid with zeros for an empty map.

```
p = zeros(10);
map = occupancyMap(p);
```

Copy the occupancy grid map. Modify the original map. The copied map is not modified. Plot the two maps side by side.

```
mapCopy = copy(map);
setOccupancy(map,[1:3;1:3]',ones(3,1));
subplot(1,2,1)
show(map)
title('Original map')
subplot(1,2,2)
show(mapCopy)
title('Copied map')
```



Input Arguments

`map` — Map representation

`occupancyMap` object

Map representation, specified as a `occupancyMap` object. This object represents the environment of the vehicle. The object contains a matrix grid with values representing the probability of the occupancy of that cell. Values close to 1 represent a high probability that the cell contains an obstacle. Values close to 0 represent a high probability that the cell is not occupied and obstacle free.

Output Arguments

`copyMap` — Copied map representation

`occupancyMap` object

Map representation, specified as a `occupancyMap` object. The properties are the same as the input object, `map`, but the copy has a different object handle.

See Also

`binaryOccupancyMap` | `occupancyMap` | `occupancyMatrix` | `getOccupancy`

Topics

“Occupancy Grids”

Introduced in R2019b

getOccupancy

Get occupancy value of locations

Syntax

```
occVal = getOccupancy(map,xy)
occVal = getOccupancy(map,xy,'local')
occVal = getOccupancy(map,ij,'grid')
[occVal,validPts] = getOccupancy( ___ )

occMatrix = getOccupancy(map)
occMatrix = getOccupancy(map,bottomLeft,matSize)
occMatrix = getOccupancy(map,bottomLeft,matSize,'local')
occMatrix = getOccupancy(map,topLeft,matSize,'grid')
```

Description

`occVal = getOccupancy(map,xy)` returns an array of probability occupancy values at the `xy` locations in the world frame. Values close to 1 represent a high probability that the cell contains an obstacle. Values close to 0 represent a high probability that the cell is not occupied and obstacle free. Unknown locations, including outside the map, return `map.DefaultValue`.

`occVal = getOccupancy(map,xy,'local')` returns an array of occupancy values at the `xy` locations in the local frame.

`occVal = getOccupancy(map,ij,'grid')` specifies `ij` grid cell indices instead of `xy` locations.

`[occVal,validPts] = getOccupancy( ___ )` additionally outputs an `n`-element vector of logical values indicating whether input coordinates are within the map limits.

`occMatrix = getOccupancy(map)` returns all occupancy values in the map as a matrix.

`occMatrix = getOccupancy(map,bottomLeft,matSize)` returns a matrix of occupancy values by specifying the bottom-left corner location in world coordinates and the matrix size in meters.

`occMatrix = getOccupancy(map,bottomLeft,matSize,'local')` returns a matrix of occupancy values by specifying the bottom-left corner location in local coordinates and the matrix size in meters.

`occMatrix = getOccupancy(map,topLeft,matSize,'grid')` returns a matrix of occupancy values by specifying the top-left corner location in grid indices and the matrix size.

Examples

Insert Laser Scans into Occupancy Map

Create an empty occupancy grid map.

```
map = occupancyMap(10,10,20);
```

Input pose of the vehicle, ranges, angles, and the maximum range of the laser scan.

```
pose = [5,5,0];
ranges = 3*ones(100,1);
angles = linspace(-pi/2,pi/2,100);
maxrange = 20;
```

Create a `lidarScan` object with the specified ranges and angles.

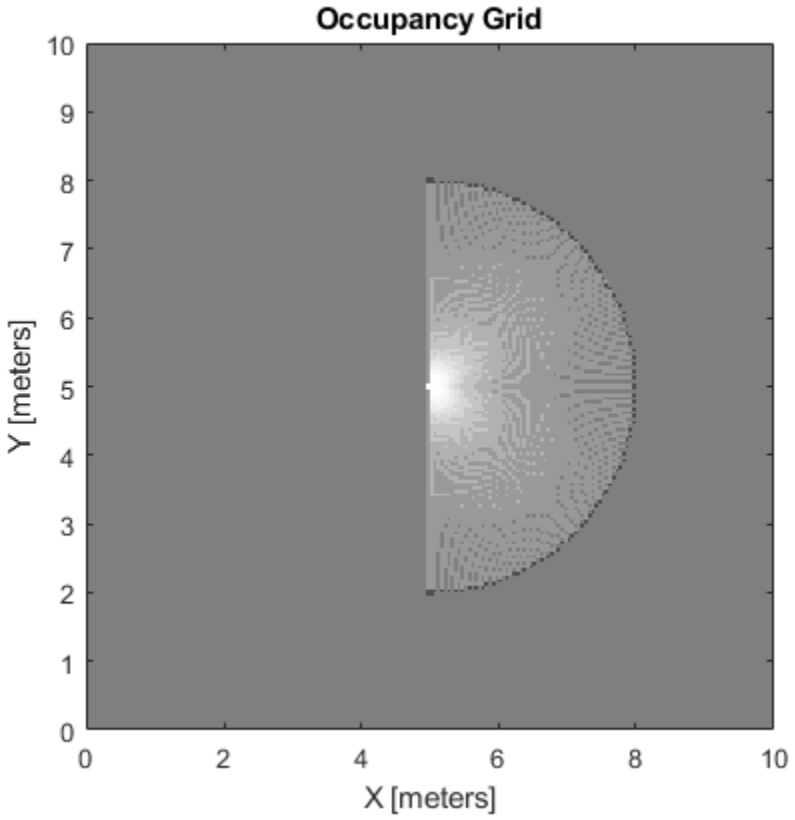
```
scan = lidarScan(ranges,angles);
```

Insert the laser scan data into the occupancy map.

```
insertRay(map,pose,scan,maxrange);
```

Show the map to see the results of inserting the laser scan.

```
show(map)
```



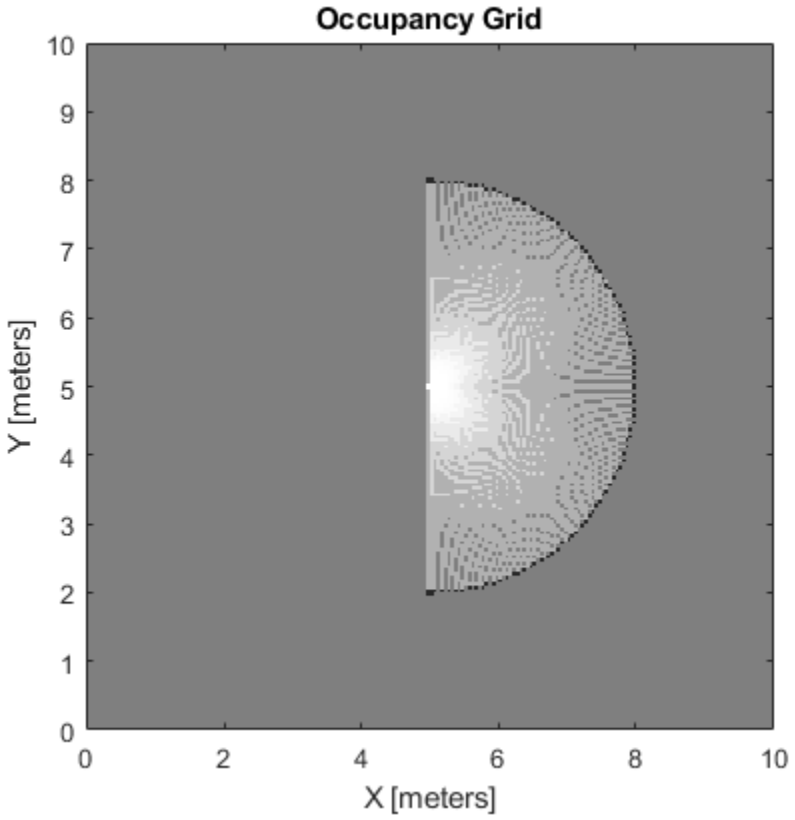
Check the occupancy of the spot directly in front of the vehicle.

```
getOccupancy(map,[8 5])
```

```
ans = 0.7000
```

Add a second reading and view the update to the occupancy values. The additional reading increases the confidence in the readings. The free and occupied values become more distinct.

```
insertRay(map,pose,scan,maxrange);
show(map)
```




```
getOccupancy(map,[8 5])
```

```
ans = 0.8448
```

Get Occupancy Values and Check Occupancy Status

Access occupancy values and check their occupancy status based on the occupied and free thresholds of the `occupancyMap` object.

Create a matrix and populate it with values. Use this matrix to create an occupancy map.

```
p = 0.5*ones(20,20);
p(11:20,11:20) = 0.75*ones(10,10);
map = occupancyMap(p,10);
```

Get the occupancy of different locations and check their occupancy statuses. The occupancy status returns 0 for free space and 1 for occupied space. Unknown values return –1.

```
pocc = getOccupancy(map,[1.5 1])
```

```
pocc = 0.7500
```

```
occupied = checkOccupancy(map,[1.5 1])
```

```
occupied = 1


pocc2 = getOccupancy(map,[5 5],'grid')

pocc2 = 0.5000

occupied2 = checkOccupancy(map,[5 5],'grid')

occupied2 = -1
```

Input Arguments

`map` — Map representation

`occupancyMap` object

Map representation, specified as a `occupancyMap` object. This object represents the environment of the vehicle. The object contains a matrix grid with values representing the probability of the occupancy of that cell. Values close to 1 represent a high probability that the cell contains an obstacle. Values close to 0 represent a high probability that the cell is not occupied and obstacle free.

`xy` — World coordinates

n-by-2 matrix

World coordinates, specified as an *n*-by-2 matrix of `[x y]` pairs, where *n* is the number of world coordinates.

Data Types: `double`

`ij` — Grid positions

n-by-2 matrix

Grid positions, specified as an *n*-by-2 matrix of `[i j]` pairs in `[rows cols]` format, where *n* is the number of grid positions.

Data Types: `double`

`bottomLeft` — Location of output matrix in world or local

two-element vector | `[xCoord yCoord]`

Location of bottom left corner of output matrix in world or local coordinates, specified as a two-element vector, `[xCoord yCoord]`. Location is in world or local coordinates based on syntax.

Data Types: `double`

`matSize` — Output matrix size

two-element vector | `[xLength yLength]` | `[gridRow gridCol]`

Output matrix size, specified as a two-element vector, `[xLength yLength]` or `[gridRow gridCol]`. Size is in world, local, or grid coordinates based on syntax.

Data Types: `double`

`topLeft` — Location of grid

two-element vector | `[iCoord jCoord]`

Location of top left corner of grid, specified as a two-element vector, `[iCoord jCoord]`.

Data Types: `double`

Output Arguments

`occVal` — Probability occupancy values

column vector

Probability occupancy values, returned as a column vector the same length as either `xy` or `ij`.

Values close to 0 represent a high probability that the cell is not occupied and obstacle free.

`validPts` — Valid map locations

n-by-1 column vector

Valid map locations, returned as an *n*-by-1 column vector equal in length to `xy` or `ij`. Locations inside the map return a value of 1. Locations outside the map limits return a value of 0.

`occMatrix` — Matrix of occupancy values

matrix

Matrix of occupancy values, returned as matrix with size equal to `matSize` or the size of `map`.

Values close to 0 represent a high probability that the cell is not occupied and obstacle free.

Limitations

Occupancy values have a limited resolution of ±0.001. The values are stored as `int16` using a log-odds representation. This data type limits resolution, but saves memory when storing large maps in MATLAB. When calling `setOccupancy` and then `getOccupancy`, the value returned might not equal the value you set. For more information, see the log-odds representations section in “Occupancy Grids”.

See Also

`occupancyMap` | `checkOccupancy`

Topics

“Occupancy Grids” (Robotics System Toolbox)

Introduced in R2019b

grid2local

Convert grid indices to local coordinates

Syntax

```
xy = grid2local(map,ij)
```

Description

`xy = grid2local(map,ij)` converts a `[row col]` array of grid indices, `ij`, to an array of local coordinates, `xy`.

Input Arguments

`map` — Map representation

`occupancyMap` object | `mapLayer` object | `multiLayerMap` object

Map representation, specified as a `occupancyMap`, `mapLayer`, or `multiLayerMap` object.

`ij` — Grid positions

n-by-2 vertical array

Grid positions, specified as an *n*-by-2 vertical array of `[i j]` pairs in `[rows cols]` format, where *n* is the number of grid positions.

Output Arguments

`xy` — Local coordinates

n-by-2 vertical array

Local coordinates, specified as an *n*-by-2 vertical array of `[x y]` pairs, where *n* is the number of local coordinates.

See Also

`grid2world` | `occupancyMap`

Introduced in R2019b

grid2world

Convert grid indices to world coordinates

Syntax

```
xy = grid2world(map,ij)
```

Description

`xy = grid2world(map,ij)` converts a `[row col]` array of grid indices, `ij`, to an array of world coordinates, `xy`.

Input Arguments

`map` — Map representation

`occupancyMap` object | `mapLayer` object | `multiLayerMap` object

Map representation, specified as a `occupancyMap`, `mapLayer`, or `multiLayerMap` object.

`ij` — Grid positions

n-by-2 matrix

Grid positions, specified as an *n*-by-2 matrix of `[i j]` pairs in `[rows cols]` format, where *n* is the number of grid positions. The grid cell locations are counted from the top left corner of the grid.

Data Types: `double`

Output Arguments

`xy` — World coordinates

n-by-2 matrix

World coordinates, returned as an *n*-by-2 matrix of `[x y]` pairs, where *n* is the number of world coordinates.

Data Types: `double`

See Also

`world2grid` | `grid2local` | `binaryOccupancyMap` | `occupancyMap`

Topics

“Occupancy Grids”

Introduced in R2019b

inflate

Inflate each occupied grid location

Syntax

```
inflate(map,radius)
inflate(map,gridradius,'grid')
```

Description

`inflate(map,radius)` inflates each occupied position of the specified `map` by the `radius`, specified in meters. Occupied location values are based on the `map.OccupiedThreshold` property. `radius` is rounded up to the nearest equivalent cell based on the resolution of the map. Values are modified using *grayscale inflation* to inflate higher probability values across the grid. This inflation increases the size of the occupied locations in the map.

`inflate(map,gridradius,'grid')` inflates each occupied position by the `gridradius`, specified in number of cells.

Examples

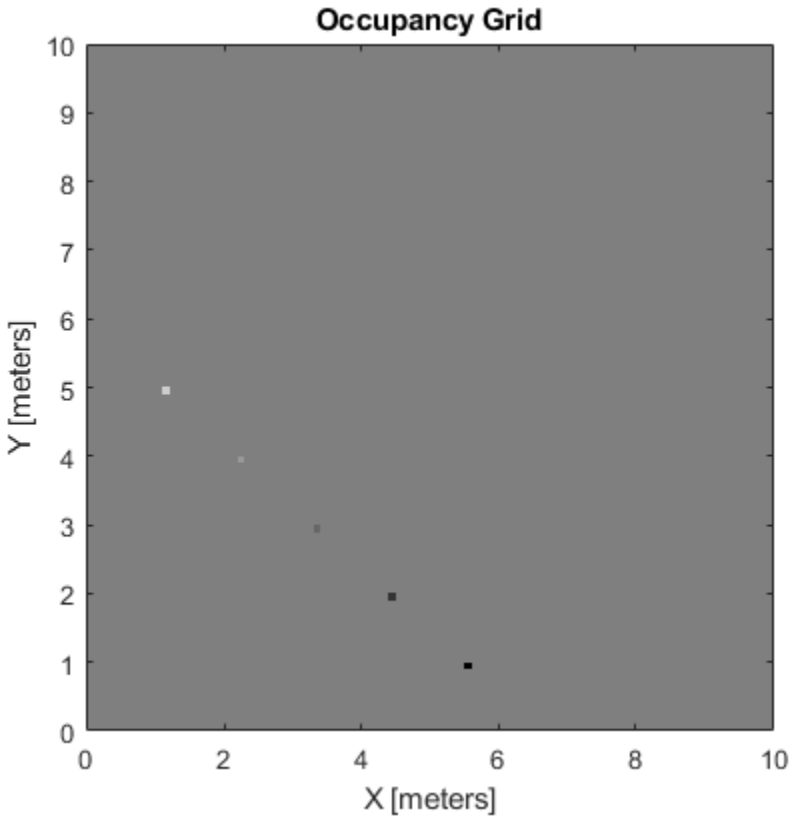
Create and Modify Occupancy Map

Create an empty map of 10-by-10 meters in size.

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

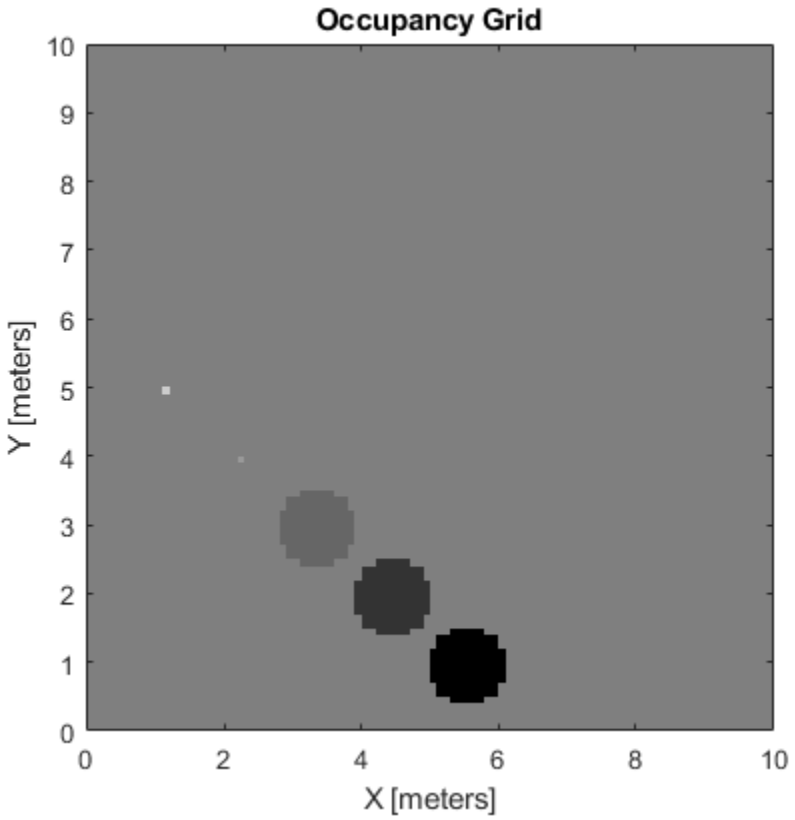
Update the occupancy of specific world locations with new probability values and display the map.

```
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 the occupied areas by a radius of 0.5 m. The larger occupancy values overwrite the smaller values.

```
inflate(map,0.5)
figure
show(map)
```

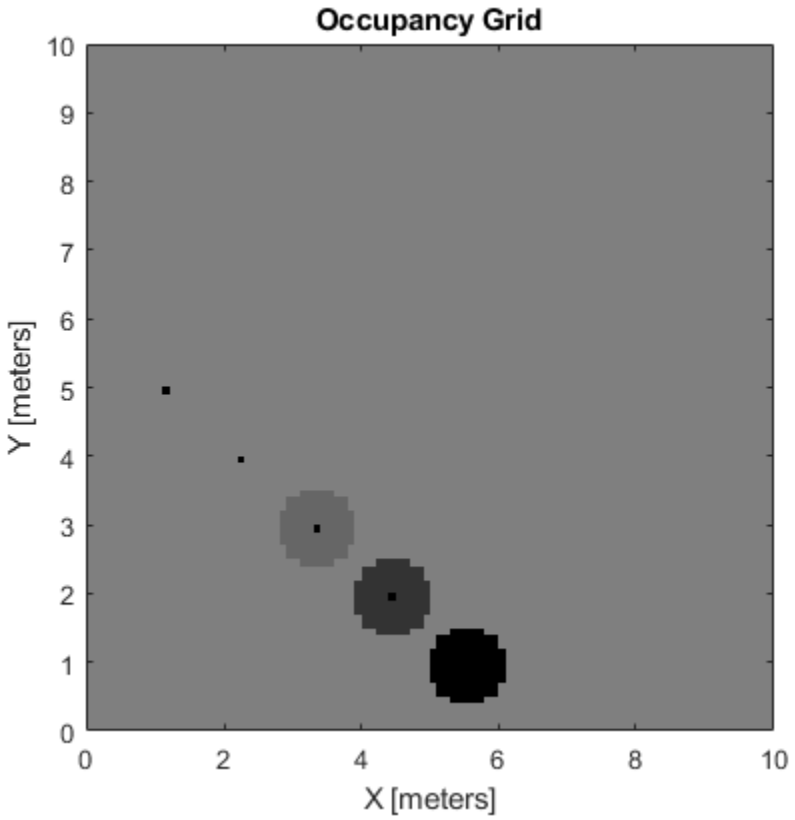


Get the grid locations from the world locations.

```
ij = world2grid(map,[x y]);
```

Set occupancy values for the grid locations.

```
setOccupancy(map,ij,ones(5,1),'grid')
figure
show(map)
```



Input Arguments

`map` — Map representation
`occupancyMap` object

Map representation, specified as a `occupancyMap` object. This object represents the environment of the vehicle. The object contains a matrix grid with values representing the probability of the occupancy of that cell. Values close to 1 represent a high probability that the cell contains an obstacle. Values close to 0 represent a high probability that the cell is not occupied and obstacle free.

`radius` — Dimension that defines by how much to inflate occupied locations
scalar

Dimension that defines by how much to inflate occupied locations, specified as a scalar in meters. `radius` is rounded up to the nearest equivalent cell value.

Data Types: `double`

`gridradius` — Number of cells by which to inflate the occupied locations
positive integer scalar

Number of cells by which to inflate the occupied locations, specified as a positive integer scalar.

Data Types: `double`

More About

Grayscale Inflation

In *grayscale inflation*, the `strel` function creates a circular structuring element using the inflation radius. The grayscale inflation of $A(x, y)$ by $B(x, y)$ is defined as:

$(A \oplus B)(x, y) = \max \{A(x - x', y' - y') + B(x', y') \mid (x', y') \in D_B\}$.

D_B is the domain of the probability values in the structuring element B. $A(x,y)$ is assumed to be $+\infty$ outside the domain of the grid.

Grayscale inflation acts as a local maximum operator and finds the highest probability values for nearby cells. The `inflate` method 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 vehicles to use as they navigate.

See Also

`binaryOccupancyMap` | `occupancyMap` | `getOccupancy`

Topics

“Occupancy Grids”

Introduced in R2019b

insertRay

Insert ray from laser scan observation

Syntax

```
insertRay(map,pose,scan,maxrange)
insertRay(map,pose,ranges,angles,maxrange)
insertRay(map,startpt,endpoints)
insertRay( ___ ,invModel)
```

Description

`insertRay(map,pose,scan,maxrange)` inserts one or more lidar scan sensor observations in the occupancy grid, `map`, using the input `lidarScan` object, `scan`, to get ray endpoints. The ray endpoints are considered free space if the input scan ranges are below `maxrange`. Cells observed as occupied are updated with an observation of 0.7. All other points along the ray are treated as obstacle free and updated with an observation of 0.4. Endpoints above `maxrange` are not updated. `NaN` values are ignored. This behavior correlates to the inverse sensor model.

`insertRay(map,pose,ranges,angles,maxrange)` specifies the range readings as vectors defined by the input `ranges` and `angles`.

`insertRay(map,startpt,endpoints)` inserts observations between the line segments from the start point to the end points. The endpoints are updated with a probability observation of 0.7. Cells along the line segments are updated with an observation of 0.4.

`insertRay( ___ ,invModel)` inserts rays with updated probabilities given in the two-element vector, `invModel`, that corresponds to obstacle-free and occupied observations. Use any of the previous syntaxes to input the rays.

Examples

Insert Laser Scans into Occupancy Map

Create an empty occupancy grid map.

```
map = occupancyMap(10,10,20);
```

Input pose of the vehicle, ranges, angles, and the maximum range of the laser scan.

```
pose = [5,5,0];
ranges = 3*ones(100,1);
angles = linspace(-pi/2,pi/2,100);
maxrange = 20;
```

Create a `lidarScan` object with the specified ranges and angles.

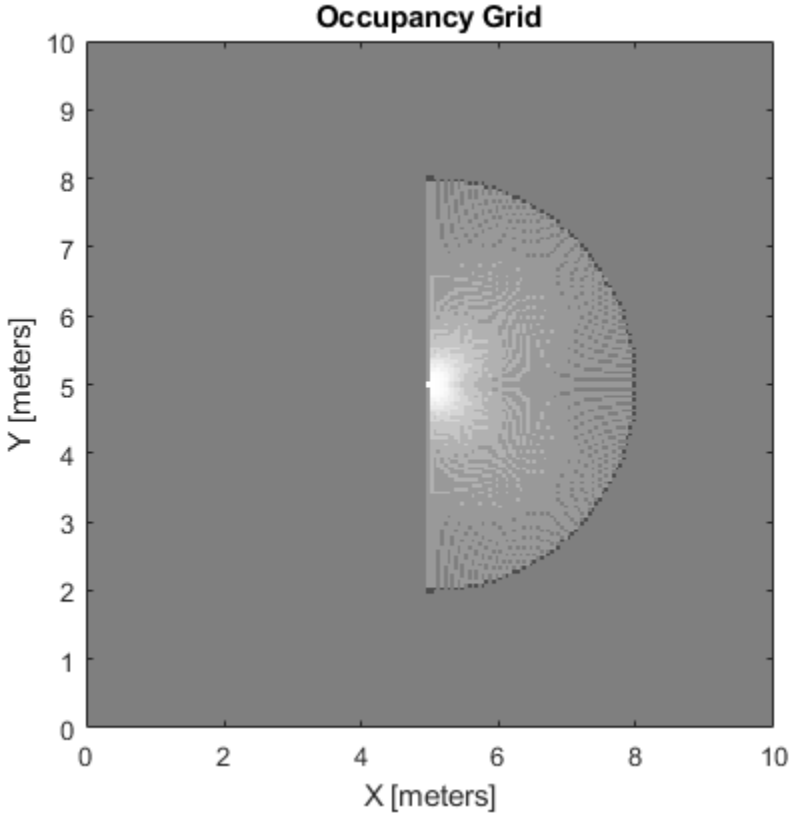
```
scan = lidarScan(ranges,angles);
```

Insert the laser scan data into the occupancy map.

```
insertRay(map,pose,scan,maxrange);
```

Show the map to see the results of inserting the laser scan.

```
show(map)
```



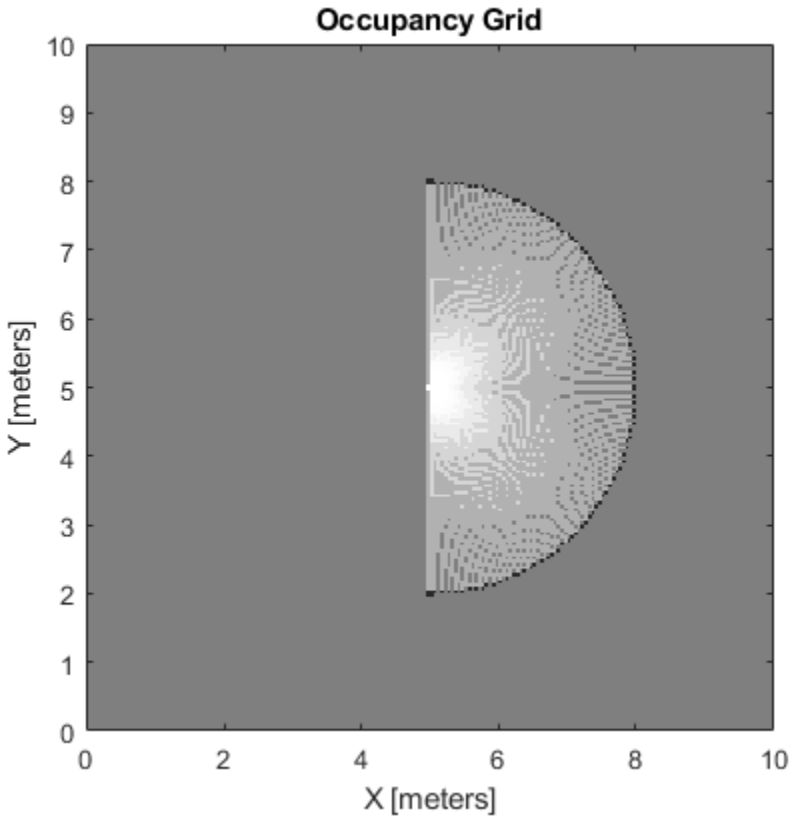
Check the occupancy of the spot directly in front of the vehicle.

```
getOccupancy(map,[8 5])
```

```
ans = 0.7000
```

Add a second reading and view the update to the occupancy values. The additional reading increases the confidence in the readings. The free and occupied values become more distinct.

```
insertRay(map,pose,scan,maxrange);
show(map)
```




```
getOccupancy(map,[8 5])
```

```
ans = 0.8448
```

Input Arguments

`map` — Map representation

`occupancyMap` object

Map representation, specified as a `occupancyMap` object. This object represents the environment of the vehicle. The object contains a matrix grid with values representing the probability of the occupancy of that cell. Values close to 1 represent a high probability that the cell contains an obstacle. Values close to 0 represent a high probability that the cell is not occupied and obstacle free.

`pose` — Position and orientation of vehicle

three-element vector

Position and orientation of vehicle, specified as an `[x y theta]` vector. The vehicle pose is an *x* and *y* position with angular orientation *theta* (in radians) measured from the *x*-axis.

`scan` — Lidar scan readings

`lidarScan` object

Lidar scan readings, specified as a `lidarScan` object.

`ranges` — Range values from scan data
vector

Range values from scan data, specified as a vector of elements measured in meters. These range values are distances from a sensor at given `angles`. The vector must be the same length as the corresponding `angles` vector.

`angles` — Angle values from scan data
vector

Angle values from scan data, specified as a vector of elements measured in radians. These angle values correspond to the given `ranges`. The vector must be the same length as the corresponding `ranges` vector.

`maxrange` — Maximum range of sensor
scalar

Maximum range of laser range sensor, specified as a scalar in meters. Range values greater than or equal to `maxrange` are considered free along the whole length of the ray, up to `maxrange`.

`startpt` — Start point for rays
two-element vector

Start point for rays, specified as a two-element vector, `[x y]`, in the world coordinate frame. All rays are line segments that originate at this point.

`endpoints` — Endpoints for rays
n-by-2 matrix

Endpoints for rays, specified as an *n*-by-2 matrix of `[x y]` pairs in the world coordinate frame, where *n* is the length of `ranges` or `angles`. All rays are line segments that originate at `startpt`.

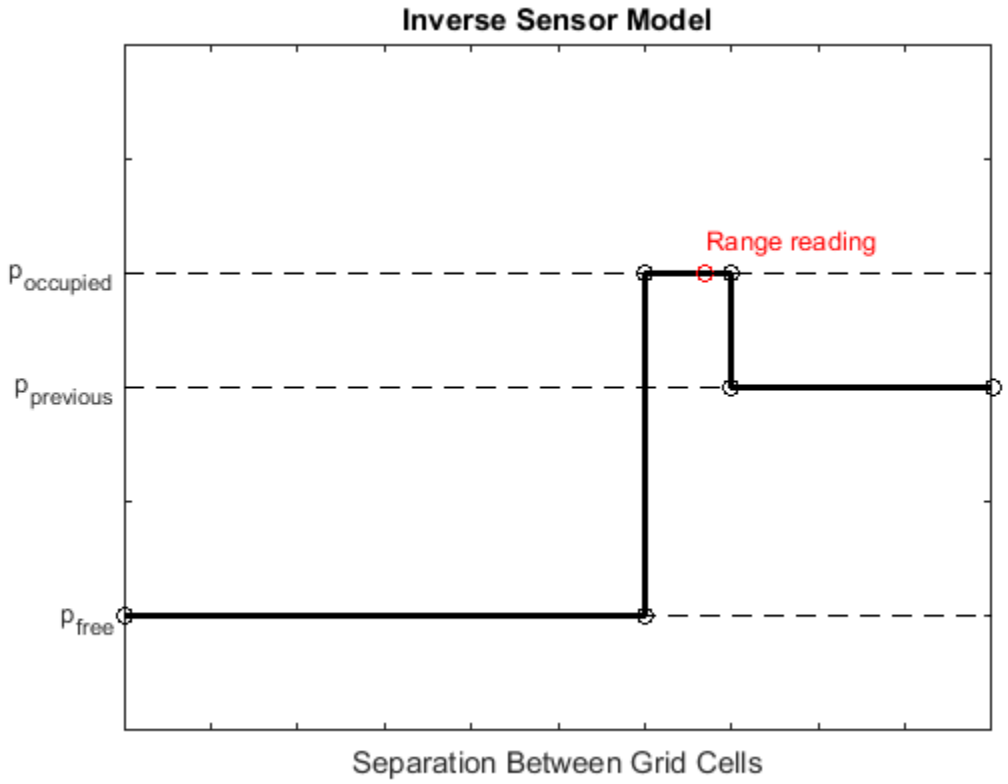
`invModel` — Inverse sensor model values
two-element vector

Inverse sensor model values, specified as a two-element vector corresponding to obstacle-free and occupied probabilities. Points along the ray are updated according to the inverse sensor model and the specified range readings. `NaN` range values are ignored. Range values greater than `maxrange` are not updated. See “Inverse Sensor Model” on page 2-795.

More About

Inverse Sensor Model

The inverse sensor model determines how values are set along a ray from a range sensor reading to the obstacles in the map. You can customize this model by specifying different probabilities for free and occupied locations in the `invModel` argument. `NaN` range values are ignored. Range values greater than `maxrange` are not updated.



Grid locations that contain range readings are updated with the occupied probability. Locations before the reading are updated with the free probability. All locations after the reading are not updated.

See Also

`occupancyMap` | `raycast` | `binaryOccupancyMap` | `lidarScan`

Topics

“Occupancy Grids”

Introduced in R2019b

local2grid

Convert local coordinates to grid indices

Syntax

```
ij = local2grid(map,xy)
```

Description

`ij = local2grid(map,xy)` converts an array of local coordinates, `xy`, to an array of grid indices, `ij` in `[row col]` format.

Input Arguments

`map` — Map representation

`occupancyMap` object | `mapLayer` object | `multiLayerMap` object

Map representation, specified as a `occupancyMap`, `mapLayer`, or `multiLayerMap` object.

`xy` — Local coordinates

n-by-2 matrix

Local coordinates, specified as an *n*-by-2 matrix of `[x y]` pairs, where *n* is the number of local coordinates.

Output Arguments

`ij` — Grid positions

n-by-2 matrix

Grid positions, returned as an *n*-by-2 matrix of `[i j]` pairs in `[row col]` format, where *n* is the number of grid positions. The grid cell locations start at (1,1) and are counted from the top left corner of the grid.

See Also

`grid2world` | `occupancyMap` | `binaryOccupancyMap`

Topics

“Occupancy Grids”

Introduced in R2019b

local2world

Convert local coordinates to world coordinates

Syntax

```
xyWorld = local2world(map,xy)
```

Description

`xyWorld = local2world(map,xy)` converts an array of local coordinates to world coordinates

Input Arguments

`map` — Map representation

`occupancyMap` object | `mapLayer` object | `multiLayerMap` object

Map representation, specified as a `occupancyMap`, `mapLayer`, or `multiLayerMap` object.

`xy` — Local coordinates

n-by-2 matrix

Local coordinates, specified as an *n*-by-2 matrix of `[x y]` pairs, where *n* is the number of world coordinates.

Data Types: `double`

Output Arguments

`xyWorld` — World coordinates

n-by-2 matrix

World coordinates, specified as an *n*-by-2 matrix of `[x y]` pairs, where *n* is the number of world coordinates.

Data Types: `double`

See Also

`grid2world` | `world2local` | `occupancyMap`

Topics

“Occupancy Grids”

Introduced in R2019b

move

Move map in world frame

Syntax

```
move(map,moveValue)
move(map,moveValue,Name,Value)
```

Description

`move(map,moveValue)` moves the local origin of the map to an absolute location, `moveValue`, in the world frame, and updates the map limits. Move values are truncated based on the resolution of the map. By default, newly revealed regions are set to `map.DefaultValue`.

`move(map,moveValue,Name,Value)` specifies additional options specified by one or more name-value pair arguments.

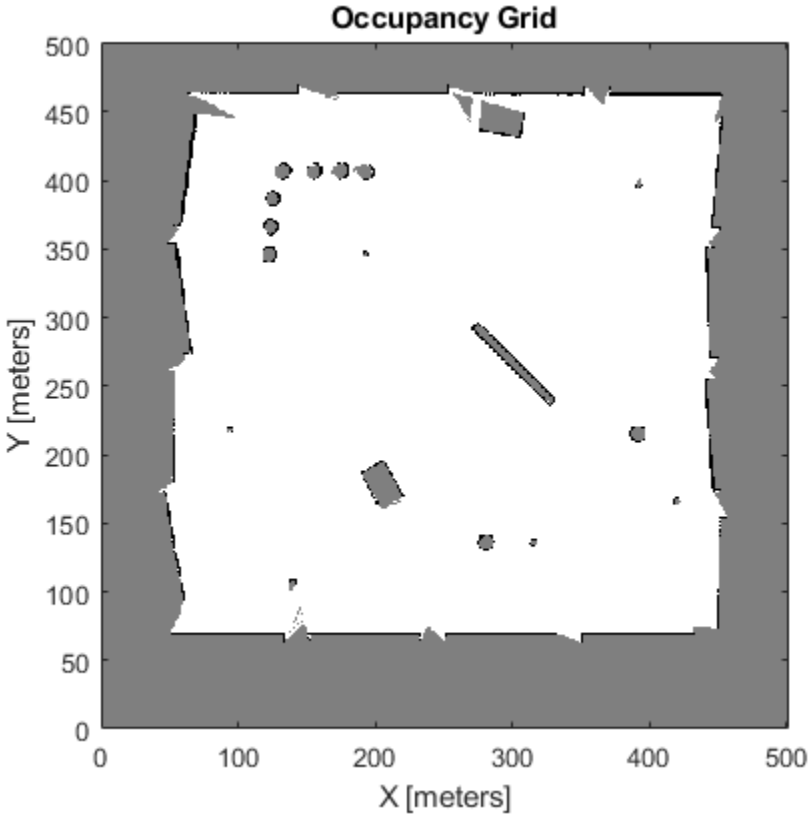
Examples

Move Local Map and Sync with World Map

This example shows how to move a local egocentric map and sync it with a larger world map. This process emulates a vehicle driving in an environment and getting updates on obstacles in the new areas.

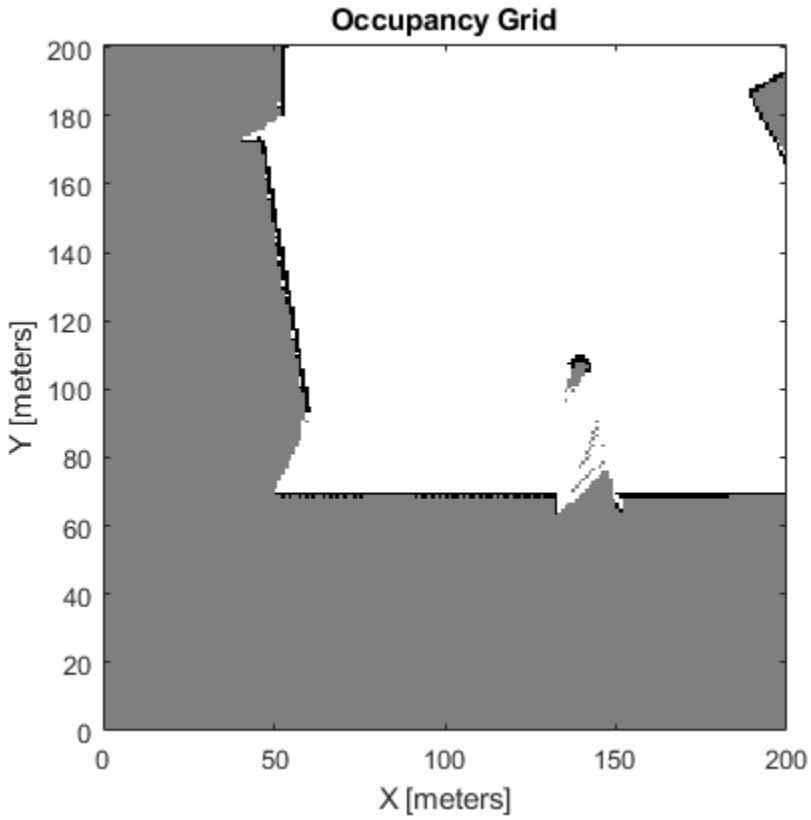
Load example maps. Create an occupancy map from the `ternaryMap`.

```
load exampleMaps.mat
map = occupancyMap(ternaryMap);
show(map)
```



Create a smaller local map.

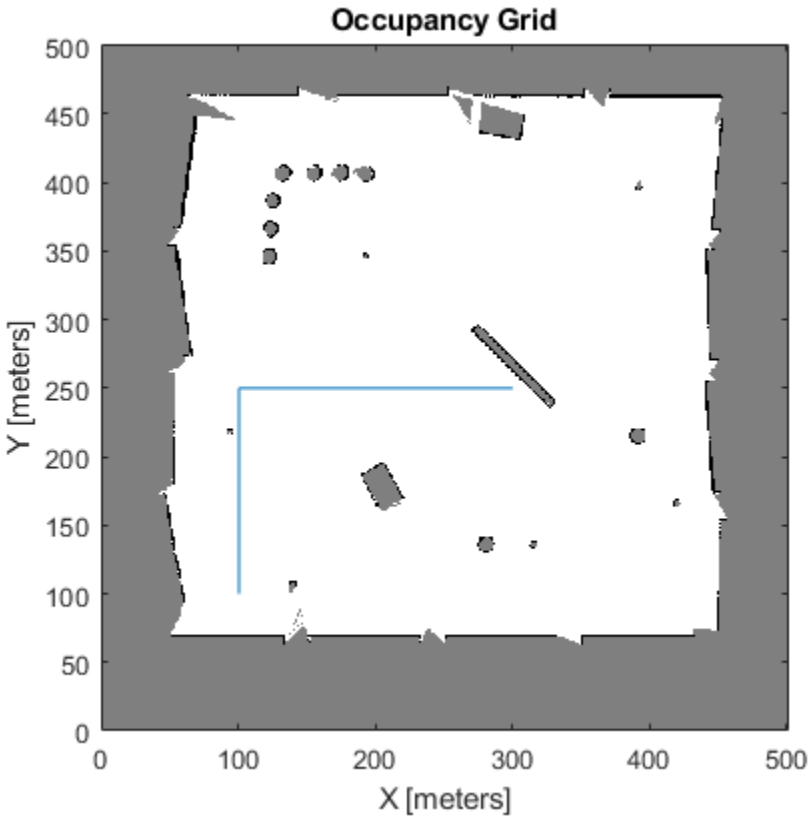
```
mapLocal = occupancyMap(ternaryMap(end-200:end,1:200));
show(mapLocal)
```



Follow a path planned in the world map and update the local map as you move your local frame.

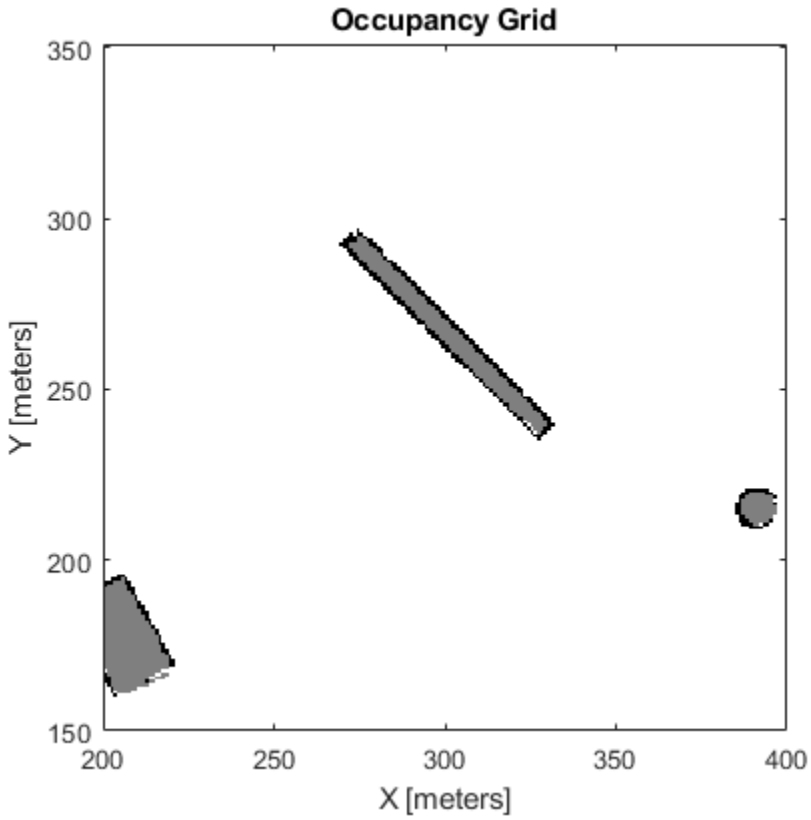
Specify path locations and plot on the map.

```
path = [100 100
        100 250
        200 250
        300 250];
show(map)
hold on
plot(path(:,1),path(:,2))
hold off
```



Create a loop for moving between points by the map resolution. Divide the difference between points by the map resolution to see how many incremental moves you can make.

```
for i = 1:length(path)-1
    moveAmount = (path(i+1,:)-path(i,:))/map.Resolution;
    for j = 1:abs(moveAmount(1)+moveAmount(2))
        moveValue = sign(moveAmount).*map.Resolution;
        move(mapLocal,moveValue, ...
            "MoveType","relative","SyncWith",map)

        show(mapLocal)
        drawnow limitrate
    end
end
```



Input Arguments

`map` — Map representation

`occupancyMap` object | `mapLayer` object | `multiLayerMap` object

Map representation, specified as a `occupancyMap`, `mapLayer`, or `multiLayerMap` object.

`moveValue` — Local map origin move value

`[x y]` vector

Local map origin move value, specified as an `[x y]` vector. By default, the value is an absolute location to move the local origin to in the world frame. Use the `MoveType` name-value pair to specify a relative move.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'MoveType','relative'`

`MoveType` — Type of move

`'absolute'` (default) | `'relative'`

Type of move, specified as `'absolute'` or `'relative'`. For relative moves, specify a relative `[x y]` vector for `moveValue` based on your current local frame.

`FillValue` — Fill value for revealed locations
`0` (default) | `1`

Fill value for revealed locations because of the shifted map limits, specified as `0` or `1`.

`SyncWith` — Secondary map to sync with
`occupancyMap` object

Secondary map to sync with, specified as a `occupancyMap` object. Any revealed locations based on the move are updated with values in this map using the world coordinates.

See Also

Objects
`multiLayerMap` | `mapLayer` | `occupancyMap` | `binaryOccupancyMap`

Functions
`occupancyMatrix`

Introduced in R2019b

occupancyMatrix

Convert occupancy grid to double matrix

Syntax

```
mat = occupancyMatrix(map)
mat = occupancyMatrix(map,'ternary')
```

Description

`mat = occupancyMatrix(map)` returns probability values stored in the occupancy grid object as a matrix.

`mat = occupancyMatrix(map,'ternary')` returns the occupancy status of each grid cell as a matrix. The `OccupiedThreshold` and `FreeThreshold` properties on the occupancy grid determine the obstacle free cells (0) and occupied cells (1). Unknown values are returned as –1.

Input Arguments

`map` — Map representation

`occupancyMap` object

Map representation, specified as a `occupancyMap` object. This object represents the environment of the vehicle. The object contains a matrix grid with values representing the probability of the occupancy of that cell. Values close to 1 represent a high probability that the cell contains an obstacle. Values close to 0 represent a high probability that the cell is not occupied and obstacle free.

Output Arguments

`mat` — Occupancy grid values

matrix

Occupancy grid values, returned as an *h*-by-*w* matrix, where *h* and *w* are defined by the two elements of the `GridSize` property of the occupancy grid object.

Data Types: `double`

See Also

`occupancyMap` | `getOccupancy` | `show` | `binaryOccupancyMap`

Topics

“Occupancy Grids”

Introduced in R2019b

raycast

Compute cell indices along a ray

Syntax

```
[endpoints,midpoints] = raycast(map,pose,range,angle)
[endpoints,midpoints] = raycast(map,p1,p2)
```

Description

`[endpoints,midpoints] = raycast(map,pose,range,angle)` returns cell indices of the specified `map` for all cells traversed by a ray originating from the specified `pose` at the specified `angle` and `range` values. `endpoints` contains all indices touched by the end of the ray, with all other points included in `midpoints`.

`[endpoints,midpoints] = raycast(map,p1,p2)` returns the cell indices of the line segment between the two specified points.

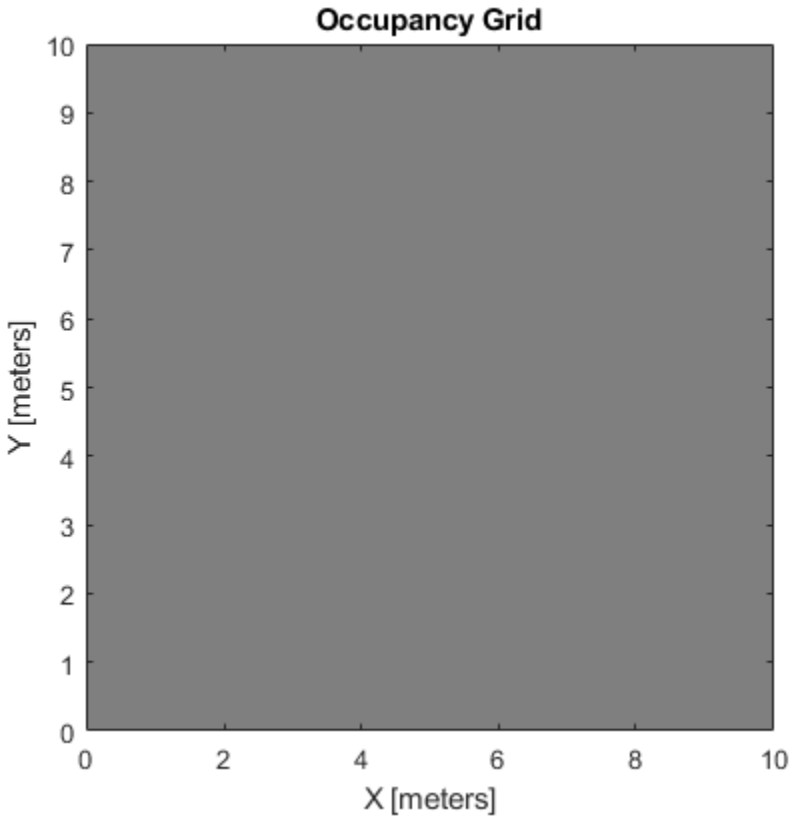
Examples

Get Grid Cells Along A Ray

Use the `raycast` method to generate cell indices for all cells traversed by a ray.

Create an empty map. A low-resolution map is used to illustrate the effected grid locations.

```
map = occupancyMap(10,10,1);
show(map)
```

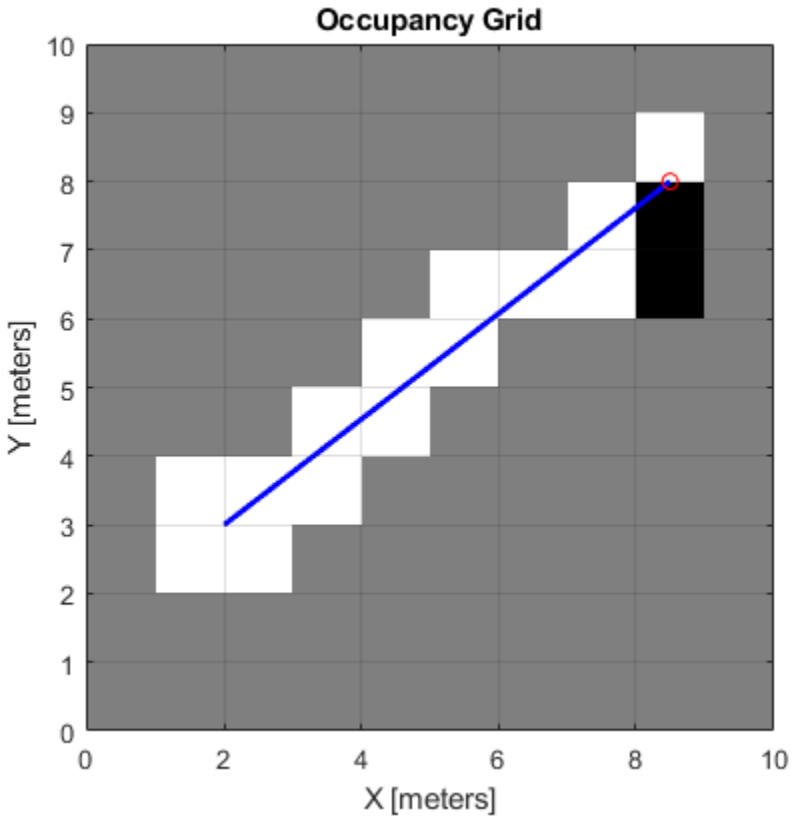


Get the grid indices of the midpoints and end points of a ray from `[2 3]` to `[8.5 8]`. Set occupancy values for these grid indices. Midpoints are treated as open space. Update endpoints with an occupied observation.

```
p1 = [2 3];
p2 = [8.5 8];
[endPts,midPts] = raycast(map,p1,p2);
setOccupancy(map,midPts,zeros(length(midPts),1),'grid');
setOccupancy(map,endPts,ones(length(endPts),1),'grid');
```

Plot the original ray over the map. Each grid cell touched by the line is updated. The starting point overlaps multiple cells, and the line touches the edge of certain cells, but all the cells are still updated.

```
show(map)
hold on
plot([p1(1) p2(1)],[p1(2) p2(2)],'-b','LineWidth',2)
plot(p2(1),p2(2),'or')
grid on
```



Input Arguments

`map` — Map representation
`occupancyMap` object

Map representation, specified as a `occupancyMap` object. This object represents the environment of the sensor. The object contains a matrix grid with values representing the probability of the occupancy of that cell. Values close to 1 represent a high probability that the cell contains an obstacle. Values close to 0 represent a high probability that the cell is not occupied and obstacle free.

`pose` — Position and orientation of sensor
three-element vector

Position and orientation of sensor, specified as an `[x y theta]` vector. The sensor pose is an *x* and *y* position with angular orientation *theta* (in radians) measured from the *x*-axis.

`range` — Range of ray
scalar

Range of ray, specified as a scalar in meters.

`angle` — Angle of ray
scalar

Angle of ray, specified as a scalar in radians. The angle value is for the corresponding `range`.

p1 — Starting point of ray
two-element vector

Starting point of ray, specified as an `[x y]` two-element vector. The point is defined in the world frame.

p2 — Endpoint of ray
two-element vector

Endpoint of ray, specified as an `[x y]` two-element vector. The point is defined in the world frame.

Output Arguments

`endpoints` — Endpoint grid indices
n-by-2 matrix

Endpoint indices, returned as an *n*-by-2 matrix of `[i j]` pairs, where *n* is the number of grid indices. The endpoints are where the `range` value hits at the specified `angle`. Multiple indices are returned when the endpoint lies on the boundary of multiple cells.

`midpoints` — Midpoint grid indices
n-by-2 matrix

Midpoint indices, returned as an *n*-by-2 matrix of `[i j]` pairs, where *n* is the number of grid indices. This argument includes all grid indices the ray intersects, excluding the endpoint.

See Also
`occupancyMap`

Topics
“Occupancy Grids”

Introduced in R2019b

rayIntersection

Find intersection points of rays and occupied map cells

Syntax

```
intersectionPts = rayIntersection(map,pose,angles,maxrange)
intersectionPts = rayIntersection(map,pose,angles,maxrange,threshold)
```

Description

`intersectionPts = rayIntersection(map,pose,angles,maxrange)` returns intersection points of rays and occupied cells in the specified `map`. Rays emanate from the specified `pose` and `angles`. Intersection points are returned in the world coordinate frame. If there is no intersection up to the specified `maxrange`, `[NaN NaN]` is returned. By default, the `OccupiedThreshold` property is used to determine occupied cells.

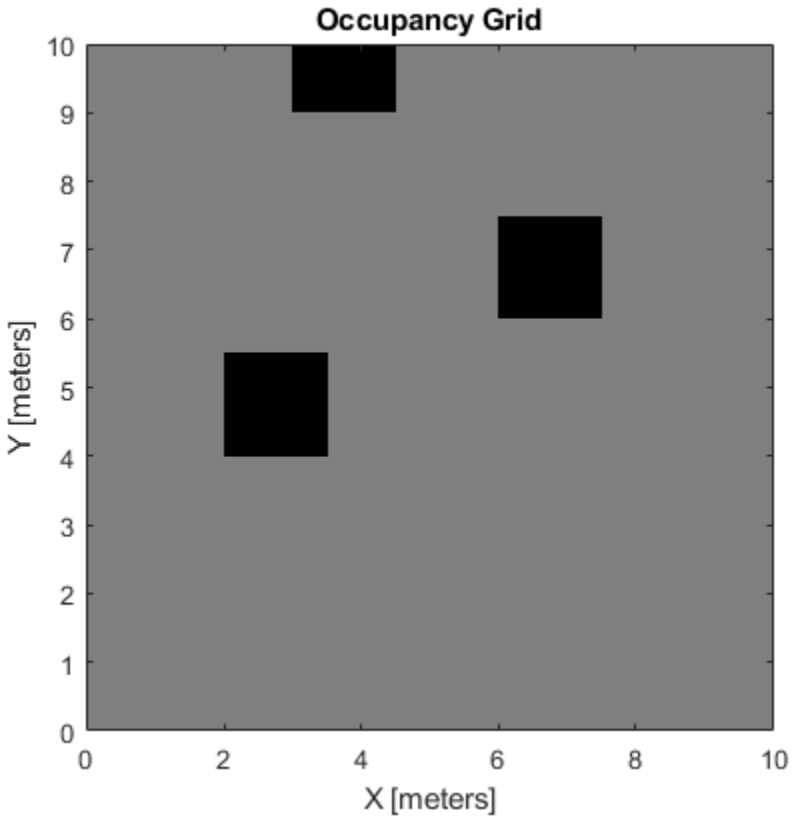
`intersectionPts = rayIntersection(map,pose,angles,maxrange,threshold)` returns intersection points based on the specified `threshold` for the occupancy values. Values greater than or equal to the threshold are considered occupied.

Examples

Get Ray Intersection Points on Occupancy Map

Create an occupancy grid map. Add obstacles and inflate them. A lower resolution map is used to illustrate the importance of using grid cells. Show the map.

```
map = occupancyMap(10,10,2);
obstacles = [4 10; 3 5; 7 7];
setOccupancy(map,obstacles,ones(length(obstacles),1))
inflate(map,0.25)
show(map)
```



Find the intersection points of occupied cells and rays that emit from the given vehicle pose. Specify the max range and angles for these rays. The last ray does not intersect with an obstacle within the max range, so it has no collision point.

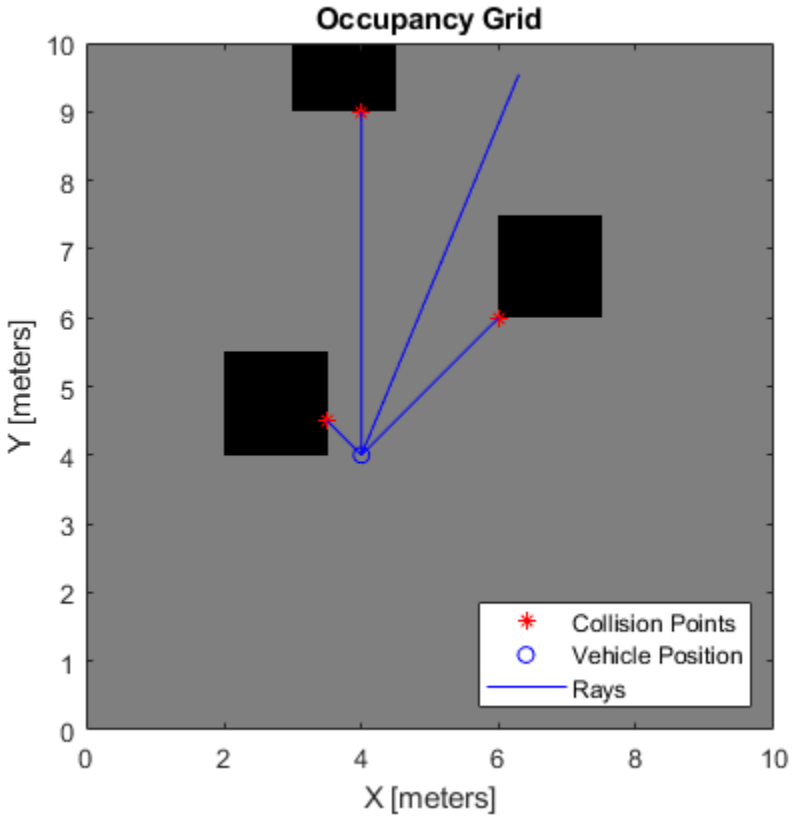
```
maxrange = 6;
angles = [pi/4,-pi/4,0,-pi/8];
vehiclePose = [4,4,pi/2];
intsectionPts = rayIntersection(map,vehiclePose,angles,maxrange,0.7)
```

```
intsectionPts = 4×2

    3.5000    4.5000
    6.0000    6.0000
    4.0000    9.0000
       NaN       NaN
```

Plot the intersection points and rays from the pose.

```
hold on
plot(intsectionPts(:,1),intsectionPts(:,2),'*r') % Intersection points
plot(vehiclePose(1),vehiclePose(2),'ob') % Vehicle pose
for i = 1:3
    plot([vehiclePose(1),intsectionPts(i,1)],...
        [vehiclePose(2),intsectionPts(i,2)],'-b') % Plot intersecting rays
end
plot([vehiclePose(1),vehiclePose(1)-6*sin(angles(4))],...
    [vehiclePose(2),vehiclePose(2)+6*cos(angles(4))],'-b') % No intersection ray
```

```
legend('Collision Points','Vehicle Position','Rays','Location','SouthEast')
```



Input Arguments

`map` — Map representation

`occupancyMap` object

Map representation, specified as a `occupancyMap` object. This object represents the environment of the sensor. The object contains a matrix grid with values representing the probability of the occupancy of that cell. Values close to 1 represent a high probability that the cell contains an obstacle. Values close to 0 represent a high probability that the cell is not occupied and obstacle free.

`pose` — Position and orientation of sensor

three-element vector

Position and orientation of sensor, specified as an `[x y theta]` vector. The sensor pose is an *x* and *y* position with angular orientation *theta* (in radians) measured from the *x*-axis.

`angles` — Ray angles emanating from sensor

vector

Ray angles emanating from the sensor, specified as a vector with elements in radians. These angles are relative to the specified sensor `pose`.

`maxrange` — Maximum range of sensor
scalar

Maximum range of laser range sensor, specified as a scalar in meters. Range values greater than or equal to `maxrange` are considered free along the whole length of the ray, up to `maxrange`.

`threshold` — Threshold for occupied cells
scalar from 0 to 1

Threshold for occupied cells, specified as a scalar from 0 to 1. Occupancy values greater than or equal to the threshold are treated as occupied cells to trigger intersections.

Output Arguments

`intersectionPts` — Intersection points
n-by-2 matrix

Intersection points, returned as *n*-by-2 matrix of `[x y]` pairs in the world frame, where *n* is the length of `angles`.

See Also

`occupancyMap` | `raycast` | `updateOccupancy` | `binaryOccupancyMap`

Topics
“Occupancy Grids”

Introduced in R2019b

setOccupancy

Set occupancy value of locations

Syntax

```
setOccupancy(map,xy,occval)

setOccupancy(map,xy,occval,'local')
setOccupancy(map,ij,occval,'grid')
validPts = setOccupancy( ___ )

setOccupancy(map,bottomLeft,inputMatrix)
setOccupancy(map,bottomLeft,inputMatrix,'local')
setOccupancy(map,topLeft,inputMatrix,'grid')
```

Description

`setOccupancy(map,xy,occval)` assigns the occupancy values to each coordinate specified in `xy`. `occval` can be a column vector the same size of `xy` or a scalar, which is applied to all coordinates.

`setOccupancy(map,xy,occval,'local')` assigns occupancy values, `occval`, to the input array of local coordinates, `xy`, as local coordinates.

`setOccupancy(map,ij,occval,'grid')` assigns occupancy values, `occval`, to the input array of grid indices, `ij`, as `[rows cols]`.

`validPts = setOccupancy( ___ )` outputs an `n`-element vector of logical values indicating whether input coordinates are within the map limits.

`setOccupancy(map,bottomLeft,inputMatrix)` assigns a matrix of occupancy values by specifying the bottom-left corner location in world coordinates.

`setOccupancy(map,bottomLeft,inputMatrix,'local')` assigns a matrix of occupancy values by specifying the bottom-left corner location in local coordinates.

`setOccupancy(map,topLeft,inputMatrix,'grid')` assigns a matrix of occupancy values by specifying the top-left cell index in grid indices and the matrix size.

Examples

Create and Modify Occupancy Map

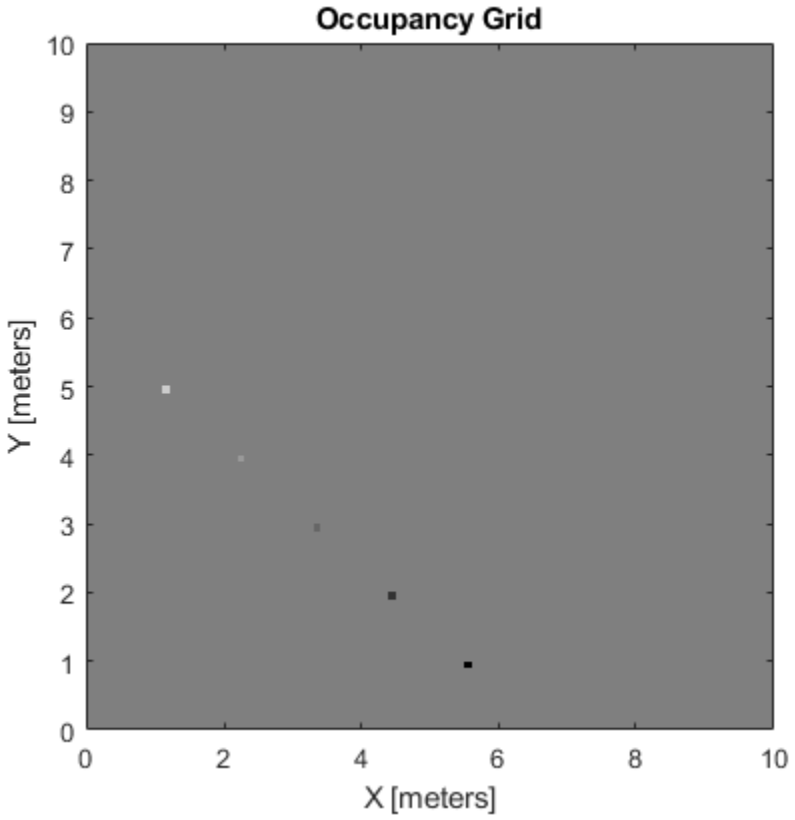
Create an empty map of 10-by-10 meters in size.

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

Update the occupancy of specific world locations with new probability values and display the map.

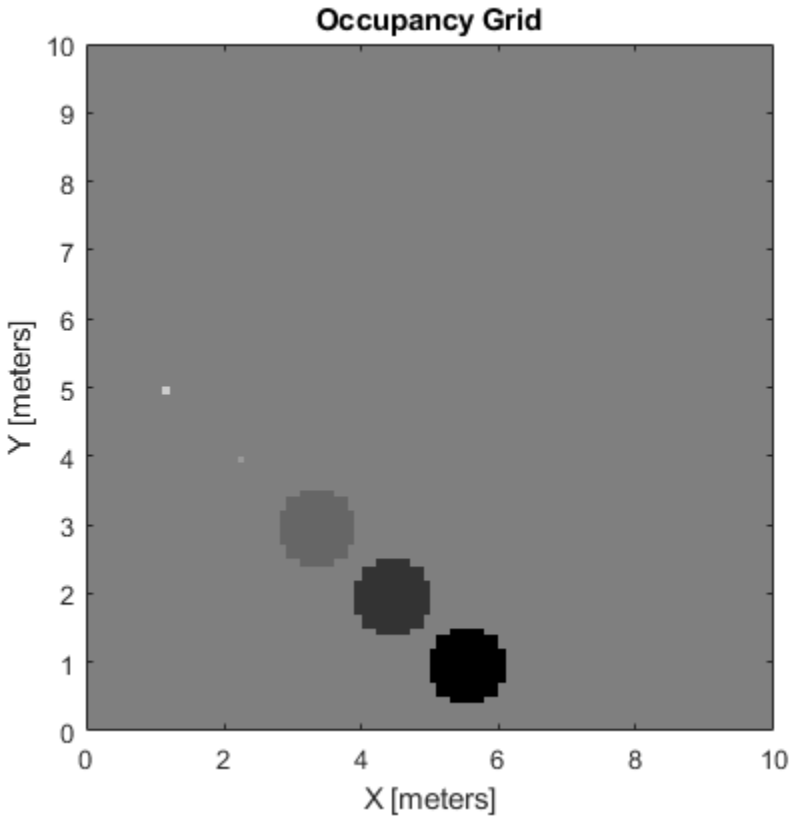
```
x = [1.2; 2.3; 3.4; 4.5; 5.6];
y = [5.0; 4.0; 3.0; 2.0; 1.0];
```

```
pvalues = [0.2; 0.4; 0.6; 0.8; 1];

updateOccupancy(map,[x y],pvalues)
figure
show(map)
```



Inflate the occupied areas by a radius of 0.5 m. The larger occupancy values overwrite the smaller values.

```
inflate(map,0.5)
figure
show(map)
```

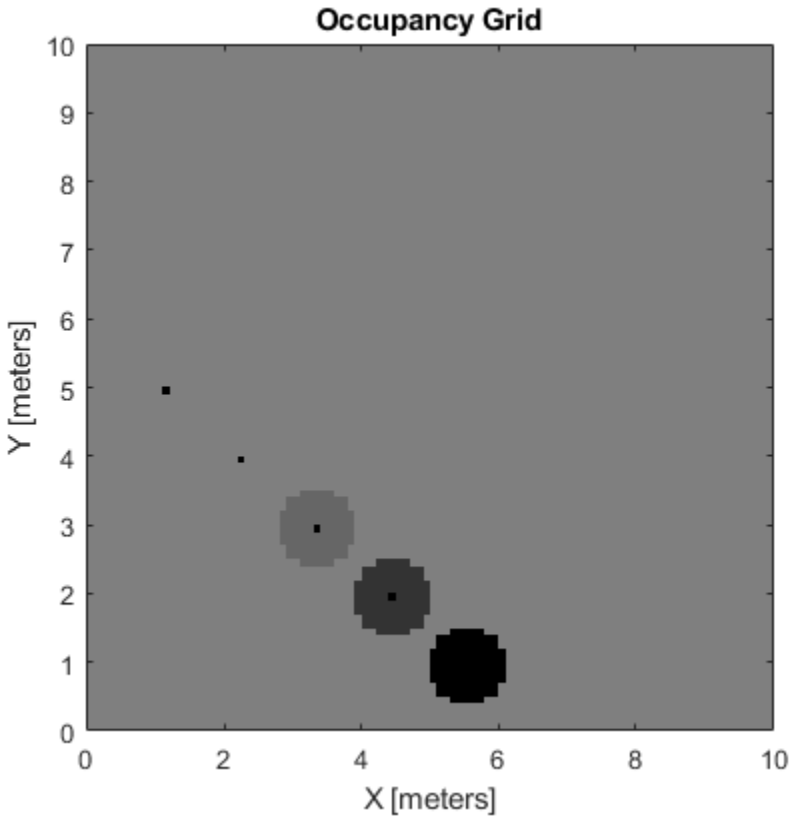


Get the grid locations from the world locations.

```
ij = world2grid(map,[x y]);
```

Set occupancy values for the grid locations.

```
setOccupancy(map,ij,ones(5,1),'grid')
figure
show(map)
```



Input Arguments

`map` — Map representation

`occupancyMap` object

Map representation, specified as a `occupancyMap` object. This object represents the environment of the vehicle. The object contains a matrix grid with values representing the probability of the occupancy of that cell. Values close to 1 represent a high probability that the cell contains an obstacle. Values close to 0 represent a high probability that the cell is not occupied and obstacle free.

`xy` — World coordinates

n-by-2 matrix

World coordinates, specified as an *n*-by-2 matrix of `[x y]` pairs, where *n* is the number of world coordinates.

Data Types: `double`

`ij` — Grid positions

n-by-2 matrix

Grid positions, specified as an *n*-by-2 matrix of `[i j]` pairs in `[rows cols]` format, where *n* is the number of grid positions.

Data Types: `double`

`occval` — Probability occupancy values
scalar | column vector

Probability occupancy values, specified as a scalar or a column vector the same size as either `xy` or `ij`. A scalar input is applied to all coordinates in either `xy` or `ij`.

Values close to 0 represent a high probability that the cell is not occupied and obstacle free.

`inputMatrix` — Occupancy values
matrix

Occupancy values, specified as a matrix. Values are given between 0 and 1 inclusively.

`bottomLeft` — Location of output matrix in world or local
two-element vector | `[xCoord yCoord]`

Location of bottom left corner of output matrix in world or local coordinates, specified as a two-element vector, `[xCoord yCoord]`. Location is in world or local coordinates based on syntax.

Data Types: `double`

`topLeft` — Location of grid
two-element vector | `[iCoord jCoord]`

Location of top left corner of grid, specified as a two-element vector, `[iCoord jCoord]`.

Data Types: `double`

Output Arguments

`validPts` — Valid map locations
n-by-1 column vector

Valid map locations, returned as an *n*-by-1 column vector equal in length to `xy` or `ij`. Locations inside the map return a value of 1. Locations outside the map limits return a value of 0.

Limitations

Occupancy values have a limited resolution of ±0.001. The values are stored as `int16` using a log-odds representation. This data type limits resolution, but saves memory when storing large maps in MATLAB. When calling `setOccupancy` and then `getOccupancy`, the value returned might not equal the value you set. For more information, see the log-odds representations section in “Occupancy Grids”.

See Also
`occupancyMap` | `getOccupancy` | `binaryOccupancyMap`

Topics
“Occupancy Grids”

Introduced in R2019b

show

Show grid values in a figure

Syntax

```
show(map)
show(map,'local')
show(map,'grid')
show( ___ ,Name,Value)
mapImage = show( ___ )
```

Description

`show(map)` displays the occupancy grid `map` in the current axes, with the axes labels representing the world coordinates.

`show(map,'local')` displays the occupancy grid `map` in the current axes, with the axes labels representing the local coordinates instead of world coordinates.

`show(map,'grid')` displays the occupancy grid `map` in the current axes, with the axes labels representing the grid coordinates.

`show( ___ ,Name,Value)` specifies additional options specified by one or more name-value pair arguments.

`mapImage = show( ___ )` returns the handle to the image object created by `show`.

Examples

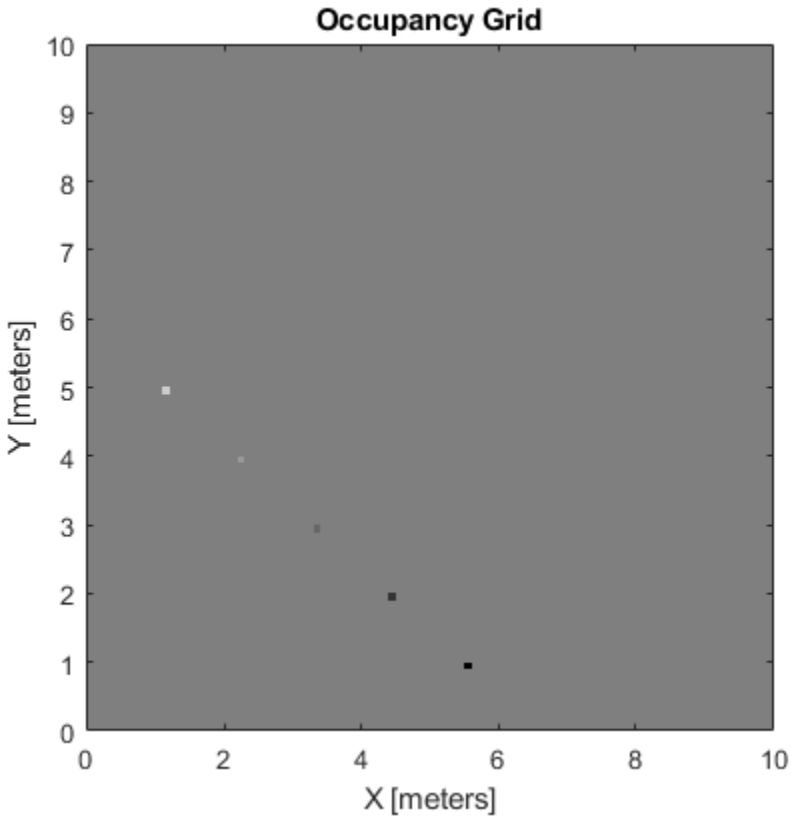
Create and Modify Occupancy Map

Create an empty map of 10-by-10 meters in size.

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

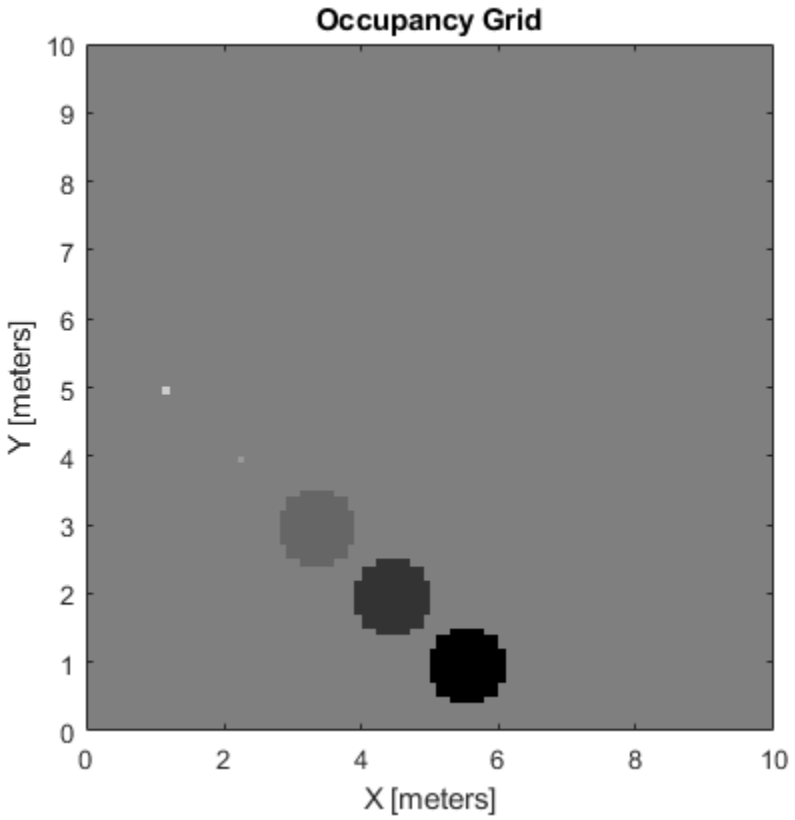
Update the occupancy of specific world locations with new probability values and display the map.

```
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 the occupied areas by a radius of 0.5 m. The larger occupancy values overwrite the smaller values.

```
inflate(map,0.5)
figure
show(map)
```

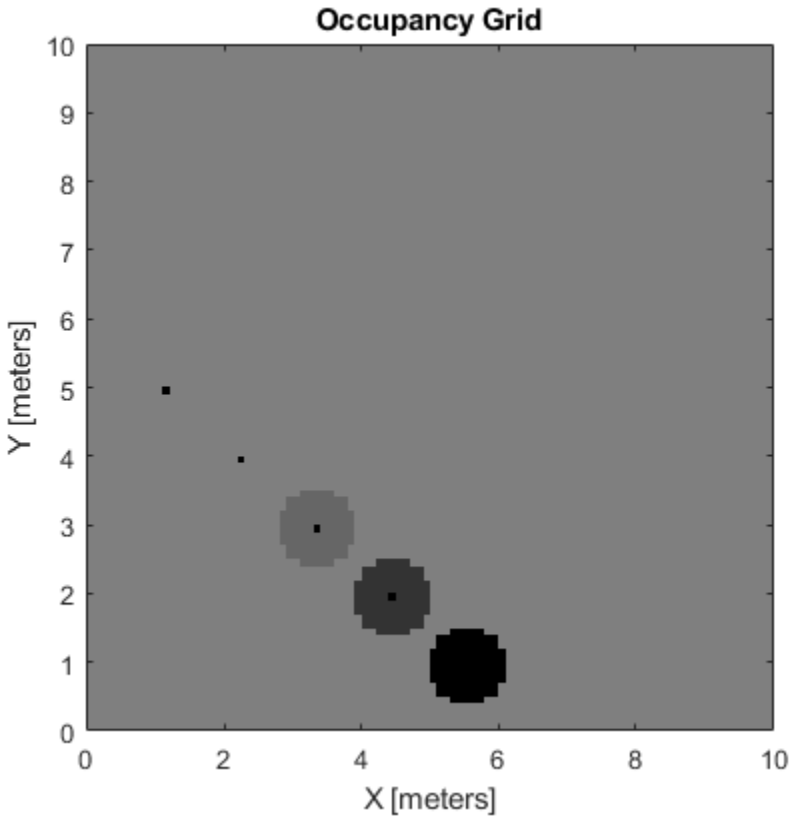


Get the grid locations from the world locations.

```
ij = world2grid(map,[x y]);
```

Set occupancy values for the grid locations.

```
setOccupancy(map,ij,ones(5,1),'grid')
figure
show(map)
```

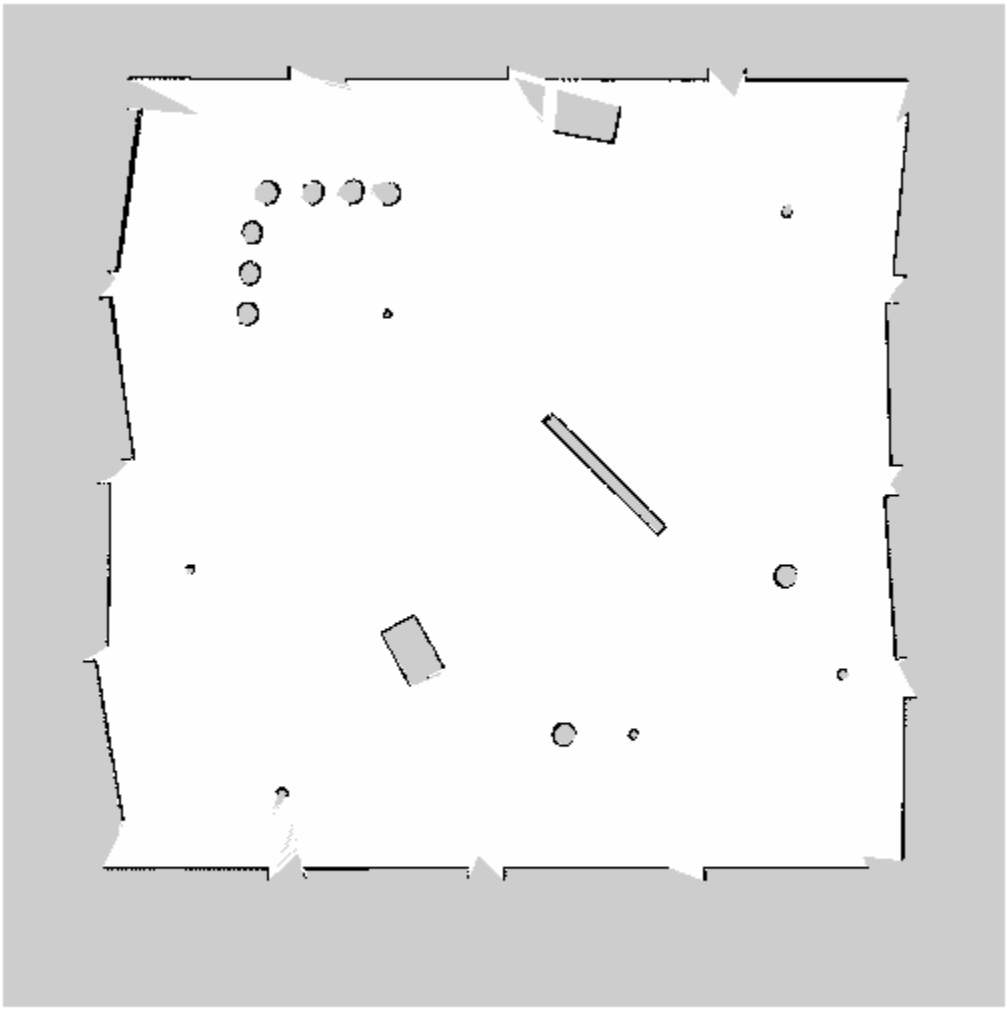


Convert PGM Image to Map

Convert a portable graymap (PGM) file containing a ROS map into an `occupancyMap` for use in MATLAB.

Import the image using `imread`. Crop the image to the playpen area.

```
image = imread('playpen_map.pgm');
imageCropped = image(750:1250,750:1250);
imshow(imageCropped)
```

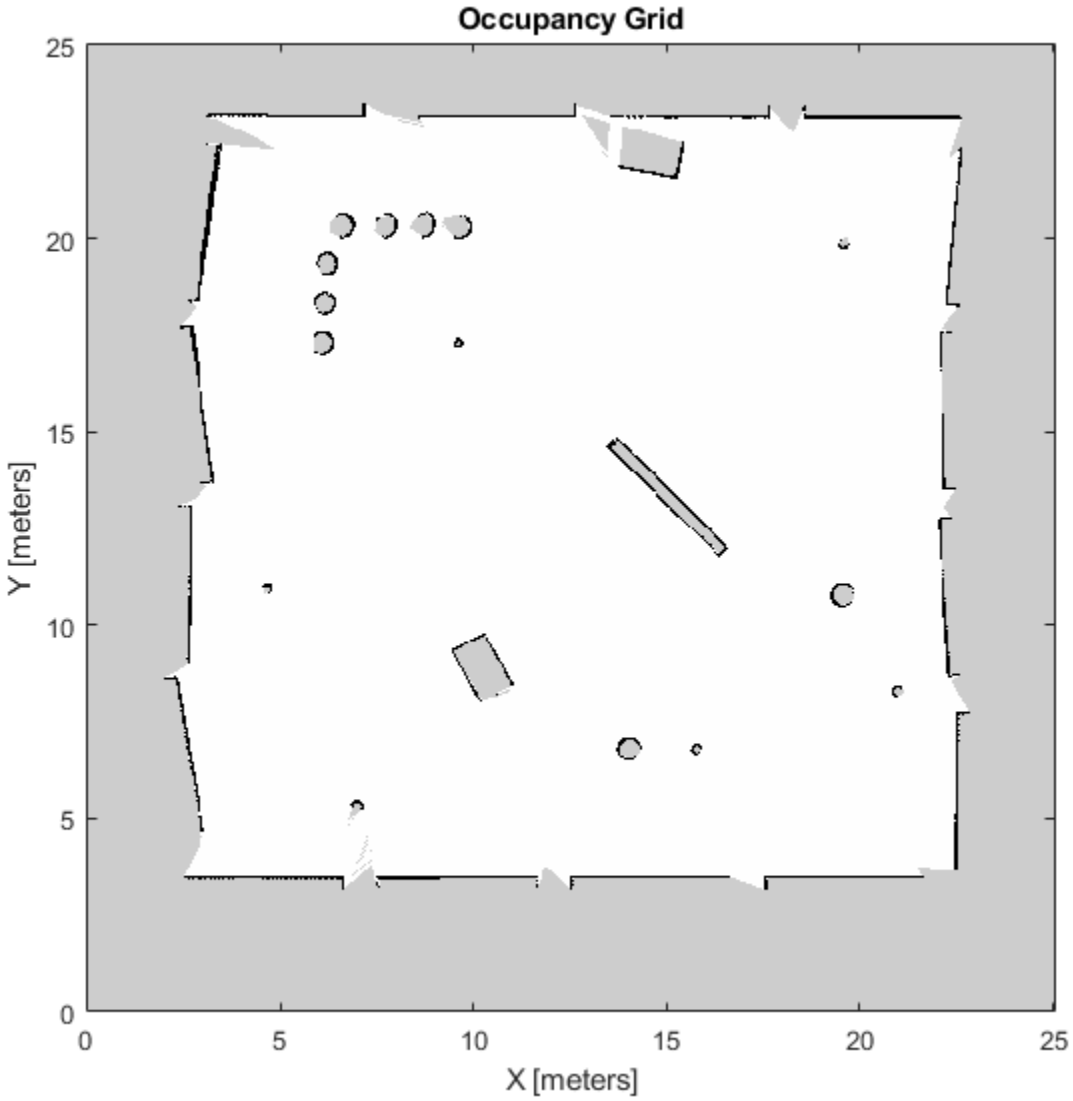


PGM values are expressed from 0 to 255 as `uint8`. Normalize these values by converting the cropped image to `double` and dividing each cell by 255. This image shows obstacles as values close to 0. Subtract the normalized image from 1 to get occupancy values with 1 representing occupied space.

```
imageNorm = double(imageCropped)/255;
imageOccupancy = 1 - imageNorm;
```

Create the `occupancyMap` object using an adjusted map image. The imported map resolution is 20 cells per meter.

```
map = occupancyMap(imageOccupancy,20);
show(map)
```



Input Arguments

`map` — Map representation

`occupancyMap` object

Map representation, specified as a `occupancyMap` object. This object represents the environment of the vehicle. The object contains a matrix grid with values representing the probability of the occupancy of that cell. Values close to 1 represent a high probability that the cell contains an obstacle. Values close to 0 represent a high probability that the cell is not occupied and obstacle free.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'Parent',axHandle`

`Parent` — Axes to plot the map

`Axes` object | `UIAxes` object

Axes to plot the map specified as either an `Axes` or `UIAxes` object. See `axes` or `uiaxes`.

`FastUpdate` — Update existing map plot

`0` (default) | `1`

Update existing map plot, specified as `0` or `1`. If you previously plotted your map on your figure, set to `1` for a faster update to the figure. This is useful for updating the figure in a loop for fast animations.

Outputs

`mapImage` — Map image

object handle

Map image, specified as an object handle.

See Also

`axes` | `occupancyMap` | `occupancyMatrix` | `binaryOccupancyMap`

Introduced in R2019b

syncWith

Sync map with overlapping map

Syntax

```
mat = syncWith(map,sourcemap)
```

Description

`mat = syncWith(map,sourcemap)` updates `map` with data from another `occupancyMap` object, `sourcemap`. Locations in `map` that are also found in `sourcemap` are updated. All other cells in `map` retain their current values.

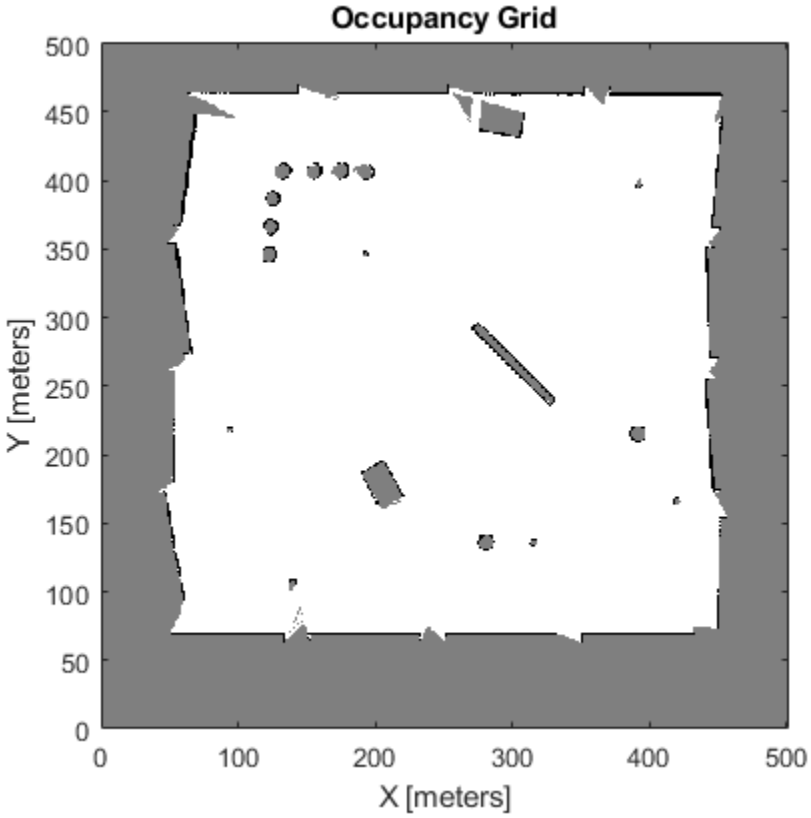
Examples

Move Local Map and Sync with World Map

This example shows how to move a local egocentric map and sync it with a larger world map. This process emulates a vehicle driving in an environment and getting updates on obstacles in the new areas.

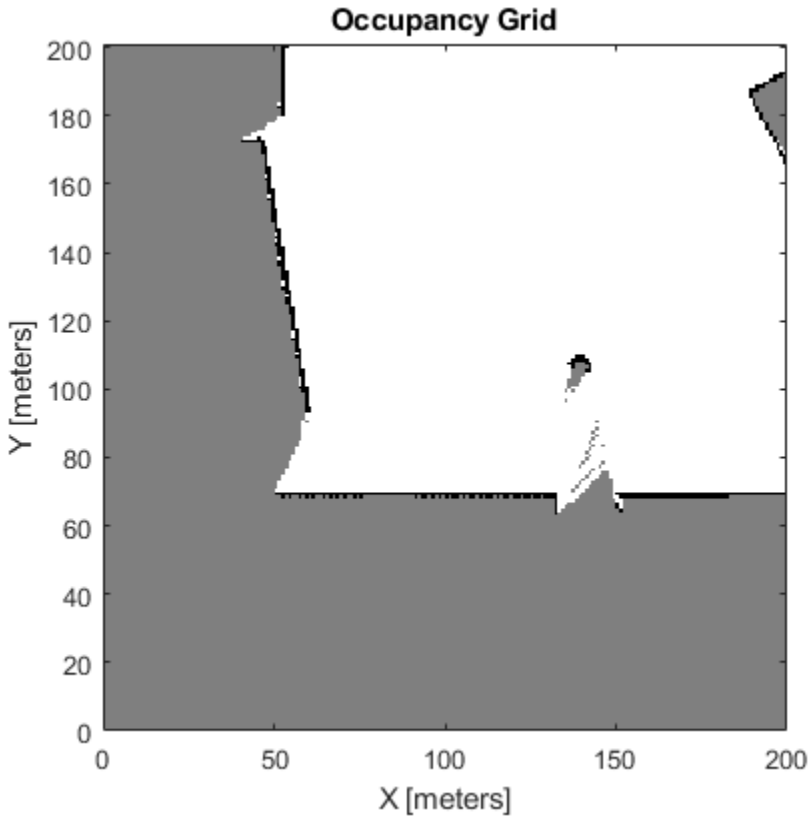
Load example maps. Create an occupancy map from the `ternaryMap`.

```
load exampleMaps.mat
map = occupancyMap(ternaryMap);
show(map)
```



Create a smaller local map.

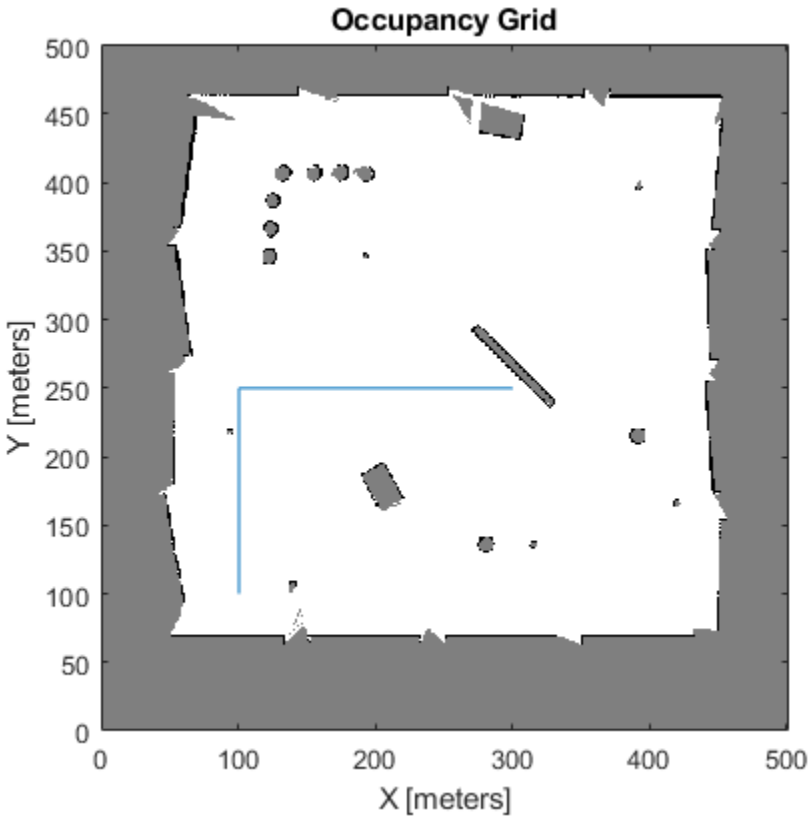
```
mapLocal = occupancyMap(ternaryMap(end-200:end,1:200));
show(mapLocal)
```



Follow a path planned in the world map and update the local map as you move your local frame.

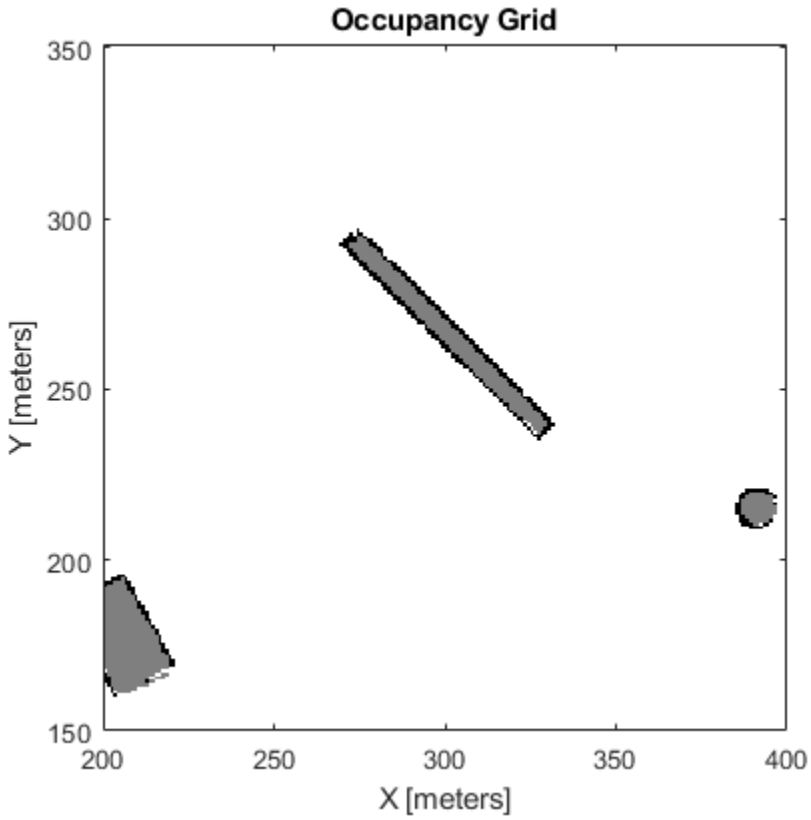
Specify path locations and plot on the map.

```
path = [100 100
        100 250
        200 250
        300 250];
show(map)
hold on
plot(path(:,1),path(:,2))
hold off
```



Create a loop for moving between points by the map resolution. Divide the difference between points by the map resolution to see how many incremental moves you can make.

```
for i = 1:length(path)-1
    moveAmount = (path(i+1,:)-path(i,:))/map.Resolution;
    for j = 1:abs(moveAmount(1)+moveAmount(2))
        moveValue = sign(moveAmount).*map.Resolution;
        move(mapLocal,moveValue, ...
            "MoveType","relative","SyncWith",map)

        show(mapLocal)
        drawnow limitrate
    end
end
```



Input Arguments

`map` — Map representation
`occupancyMap` object | `mapLayer` object | `multiLayerMap` object

Map representation, specified as a `occupancyMap`, `mapLayer`, or `multiLayerMap` object.

`sourcemap` — Source map data
`occupancyMap` object | `mapLayer` object | `multiLayerMap` object

Source map data, specified as a `occupancyMap`, `mapLayer`, or `multiLayerMap` object.

See Also
`occupancyMap` | `binaryOccupancyMap`

Topics
“Occupancy Grids”

Introduced in R2019b

updateOccupancy

Integrate probability observations at locations

Syntax

```
updateOccupancy(map,occMatrix)

updateOccupancy(map,locations,obs)
updateOccupancy(map,xy,obs,'world')
updateOccupancy(map,xy,obs,'local')
updateOccupancy(map,ij,obs,'grid')

updateOccupancy(map,bottomLeft,obsMatrix)
updateOccupancy(map,bottomLeft,obsMatrix,'world')
updateOccupancy(map,bottomLeft,obsMatrix,'local')
updateOccupancy(map,topLeft,obsMatrix,'grid')
```

Description

`updateOccupancy(map,occMatrix)` probabilistically integrates a matrix of occupancy values, `occMatrix`, with the current occupancy matrix of the `occupancyMap` object `map`. The size of the matrix must be equal to the `GridSize` property of map.

`updateOccupancy(map,locations,obs)` probabilistically integrates observation values, `obs`, into the occupancy map cells corresponding to the *n*-by-2 matrix of world coordinates `locations`. Observation values are determined based on the “Inverse Sensor Model” on page 2-836.

`updateOccupancy(map,xy,obs,'world')` probabilistically integrates observation values, `obs`, into the cells corresponding to the *n*-by-2 matrix of world coordinates `xy`.

`updateOccupancy(map,xy,obs,'local')` probabilistically integrates observation values, `obs`, into the cells corresponding to the *n*-by-2 matrix of local coordinates `xy`.

`updateOccupancy(map,ij,obs,'grid')` probabilistically integrates observation values, `obs`, into the cells corresponding to the *n*-by-2 matrix of grid indices `ij`.

`updateOccupancy(map,bottomLeft,obsMatrix)` probabilistically integrates an *m*-by-*n* matrix of observation values, `obsMatrix`, into a subregion in the map. Specify the bottom-left corner of the subregion as a world position, `bottomLeft`. The subregion extends *m* rows up and *n* columns to the right from the specified position.

`updateOccupancy(map,bottomLeft,obsMatrix,'world')` probabilistically integrates an *m*-by-*n* matrix of observation values, `obsMatrix`, into a subregion in the map. Specify the bottom-left corner of the subregion as a world position, `bottomLeft`. The subregion extends *m* rows up and *n* columns to the right from the specified position.

`updateOccupancy(map,bottomLeft,obsMatrix,'local')` probabilistically integrates an *m*-by-*n* matrix of observation values, `obsMatrix`, into a subregion in the map. Specify the bottom-left corner of the subregion as a local position, `bottomLeft`. The subregion extends *m* rows up and *n* columns to the right from the specified position.

`updateOccupancy(map,topLeft,obsMatrix,'grid')` probabilistically integrates an *m*-by-*n* matrix of observation values, `obsMatrix`, into a subregion in the map. Specify the top-left corner of the subregion as a grid index, `topLeft`. The subregion extends *m* rows down and *n* columns to the right from the specified index.

Examples

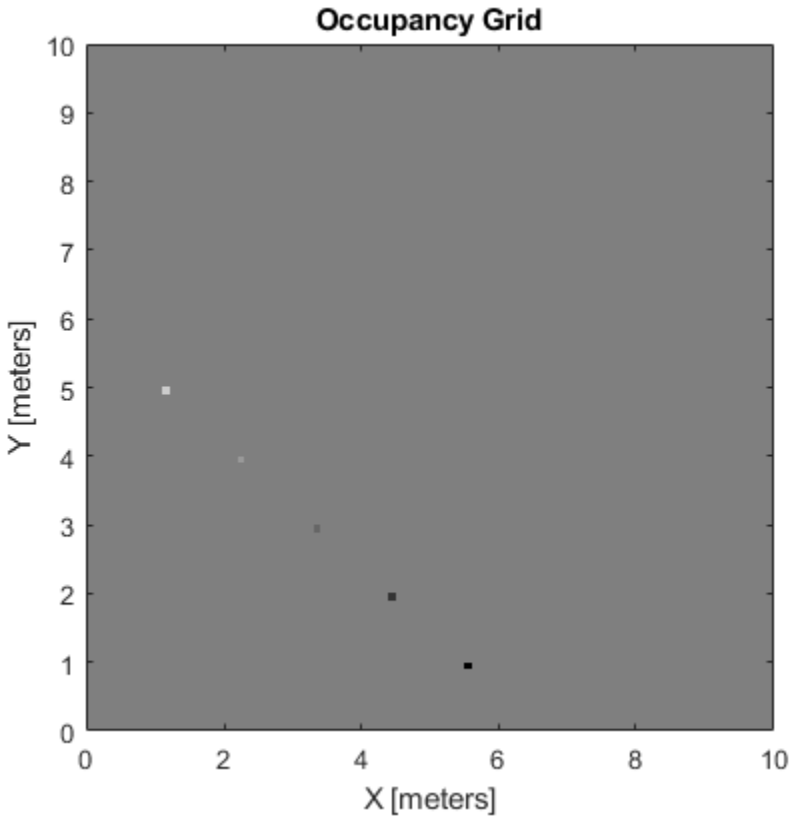
Create and Modify Occupancy Map

Create an empty map of 10-by-10 meters in size.

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

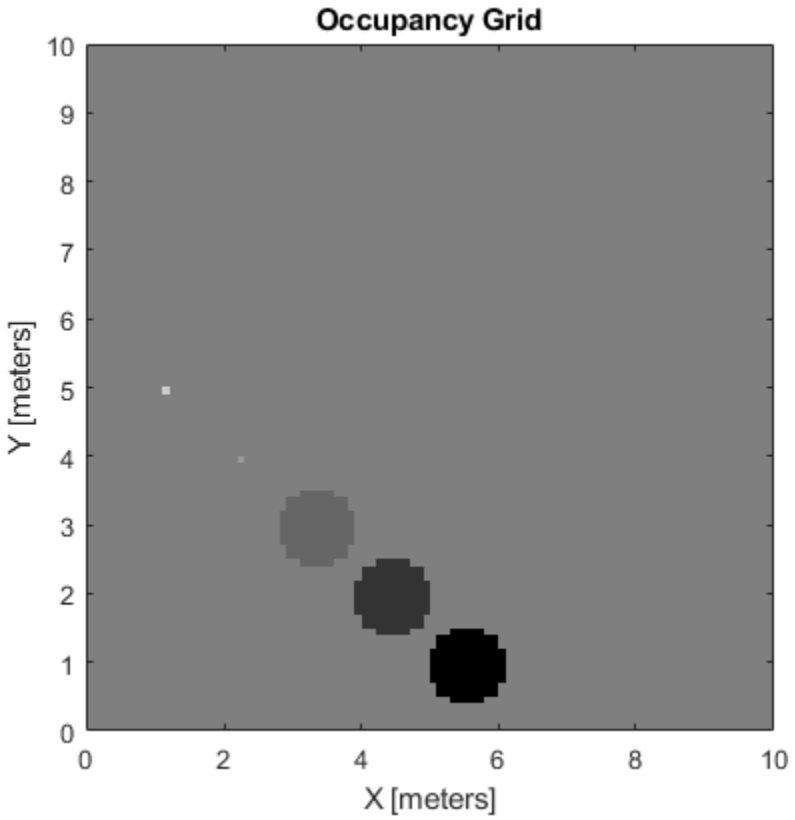
Update the occupancy of specific world locations with new probability values and display the map.

```
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 the occupied areas by a radius of 0.5 m. The larger occupancy values overwrite the smaller values.

```
inflate(map,0.5)
figure
show(map)
```

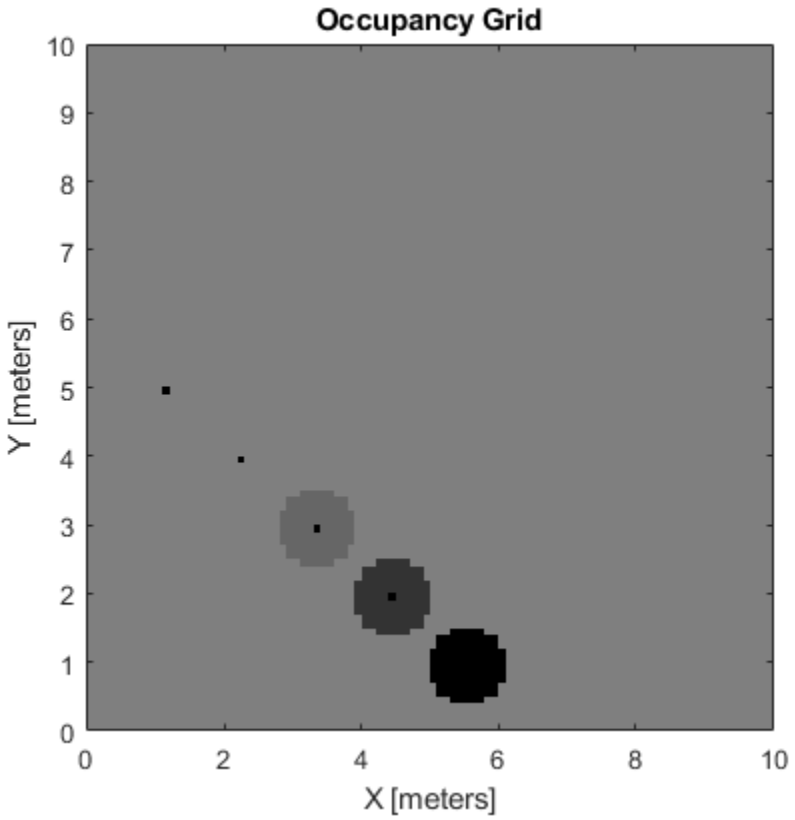


Get the grid locations from the world locations.

```
ij = world2grid(map,[x y]);
```

Set occupancy values for the grid locations.

```
setOccupancy(map,ij,ones(5,1),'grid')
figure
show(map)
```



Input Arguments

`map` — Map representation
`occupancyMap` object

Map representation, specified as an `occupancyMap` object. This object represents the environment of the vehicle. The object contains a matrix grid with each value representing the probability of the occupancy of that cell. Values close to `1` represent a high probability that the cell contains an obstacle. Values close to `0` represent a high probability that the cell is not occupied and contains no obstacles.

`occMatrix` — Matrix of occupancy values
numeric matrix | logical matrix

Matrix of occupancy values, specified as a matrix. The size of the matrix must be equal to the `GridSize` property of map.

The occupancy values can be of any numeric type, with values between 0 and 1. If the matrix is logical, the default occupancy values of 0.7 (`true`) and 0.4 (`false`) are used.

Example: `updateOccupancy(map,ones(map.GridSize)*0.6)`

Data Types: `single` | `double` | `logical`

`locations` — Cell locations in world coordinates
n-by-2 matrix

Cell locations in world coordinates, specified as an *n*-by-2 matrix with rows of the form [*x y*], where *n* is the number of world coordinates. The function ignores locations outside of the map boundaries.

Example: `updateOccupancy(map,[1 1; 3 3; 5 5],false)`

Data Types: `single` | `double`

`bottomLeft` — Location of bottom-left corner of observation matrix
two-element vector

Location of the bottom-left corner of the observation matrix, specified as a two-element vector of the form `[xCoord yCoord]`. The location is in world or local coordinates, based on the syntax.

Example: `updateOccupancy(map,[2 2],[0.2 0.4; 0.6 0.8],'world')`

Data Types: `single` | `double`

`topLeft` — Location of top-left corner of grid
two-element vector

Location of the top-left corner of the grid, specified as a two-element vector of form `[iCoord jCoord]`.

Example: `updateOccupancy(map,[2 2],[0.2 0.4; 0.6 0.8],'grid')`

Data Types: `single` | `double`

`xy` — World or local coordinates
n-by-2 matrix

World or local coordinates, specified as an *n*-by-2 matrix with rows of the form `[x y]`, where *n* is the number of coordinates.

Example: `updateOccupancy(map,[2 2; 4 4; 6 6],[0.2; 0.4; 0.6],'world')`

Data Types: `single` | `double`

`ij` — Grid positions
n-by-2 matrix

Grid positions, specified as an *n*-by-2 matrix with rows of the form `[i j]` in `[rows cols]` format, where *n* is the number of grid positions.

Example: `updateOccupancy(map,[2 2; 4 4; 6 6],[0.2; 0.4; 0.6],'grid')`

Data Types: `single` | `double`

`obs` — Probability observation values
n-element numeric column vector | *n*-element logical column vector | numeric scalar | logical scalar

Probability observation values, specified as a numeric or logical scalar or a numeric or logical *n*-element column vector the same size as either `locations`, `xy`, or `ij`.

`obs` values can be any value from 0 to 1, but if `obs` is a logical vector, the default observation values of 0.7 (`true`) and 0.4 (`false`) are used. If `obs` is a numeric or a logical scalar, the value is applied to all coordinates in `locations`, `xy`, or `ij`. These values correlate to the “Inverse Sensor Model” on page 2-836 for ray casting.

Example: `updateOccupancy(map,[2 2; 4 4; 6 6],[0.2; 0.4; 0.6],'local')`

Data Types: `single` | `double` | `logical`

`obsMatrix` — Matrix of probability observation values
m-by-*n* numeric matrix | *m*-by-*n* logical matrix

Matrix of probability observation values, specified as an *m*-by-*n* numeric or logical matrix.

The observation values can be of any numeric type with value between 0 and 1. If the matrix is logical, the default observation values of 0.7 (`true`) and 0.4 (`false`) are used.

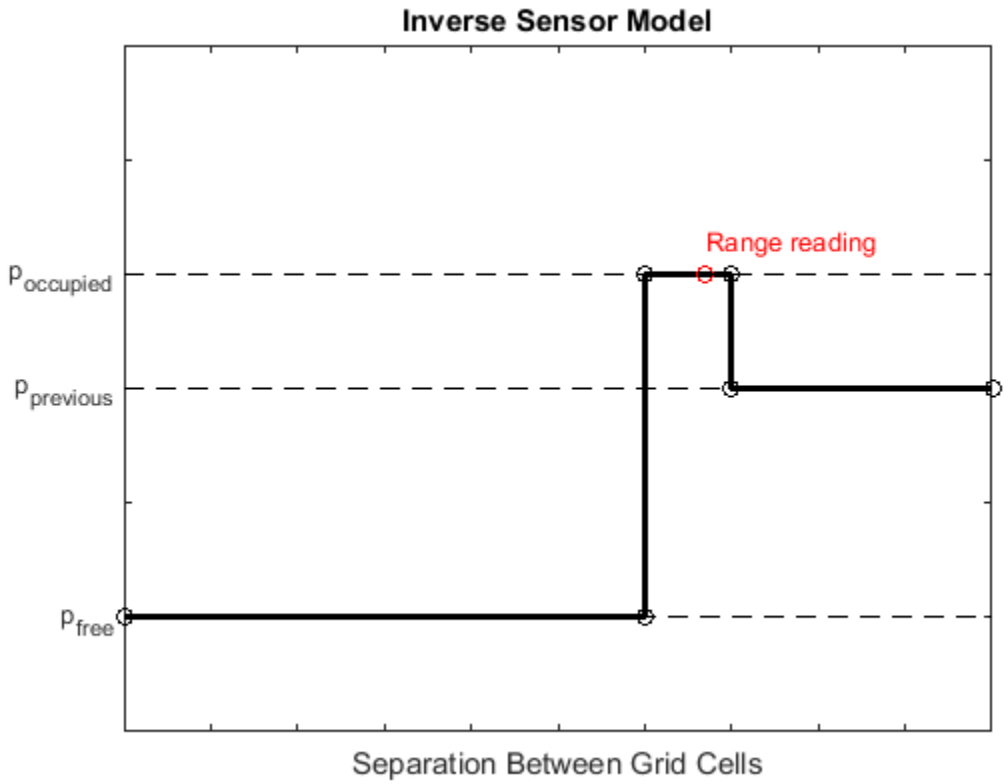
Example: `updateOccupancy(map,[2 2],[0.2 0.4; 0.6 0.8])`

Data Types: `single` | `double` | `logical`

More About

Inverse Sensor Model

The inverse sensor model determines how values are set along a ray from a range sensor reading to the obstacles in the map. `NaN` range values are ignored. Range values greater than `maxrange` are not updated.



Grid locations that contain range readings are updated with the occupied probability. Locations before the reading are updated with the free probability. All locations after the reading are not updated.

See Also

`occupancyMap` | `setOccupancy` | `binaryOccupancyMap`

Topics

“Occupancy Grids”

Introduced in R2019b

world2grid

Convert world coordinates to grid indices

Syntax

```
ij = world2grid(map,xy)
```

Description

`ij = world2grid(map,xy)` converts an array of world coordinates, `xy`, to an array of grid indices, `ij` in [*row col*] format.

Examples

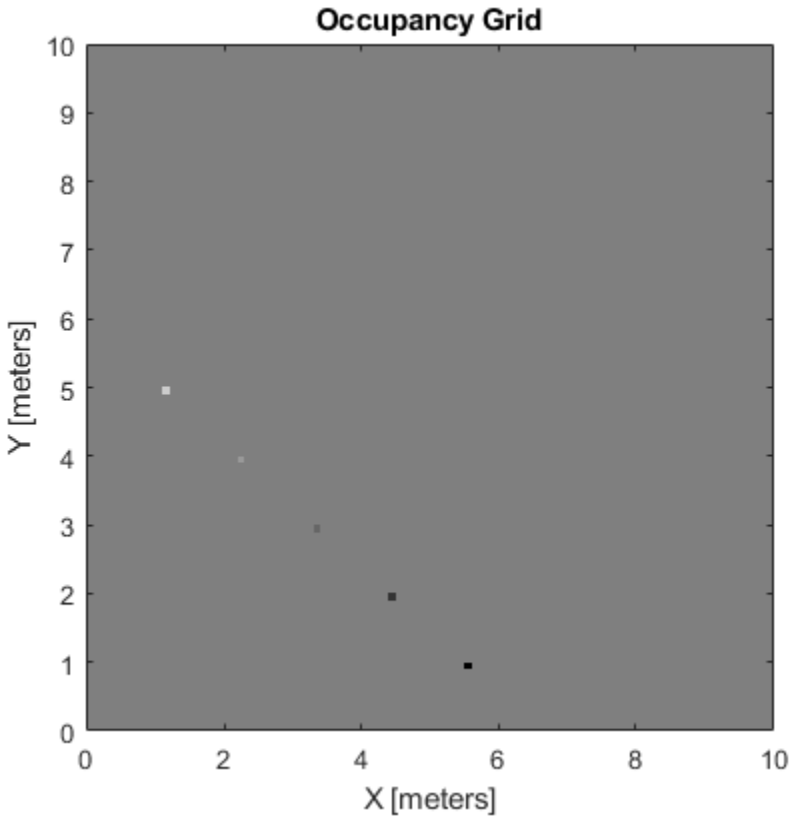
Create and Modify Occupancy Map

Create an empty map of 10-by-10 meters in size.

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

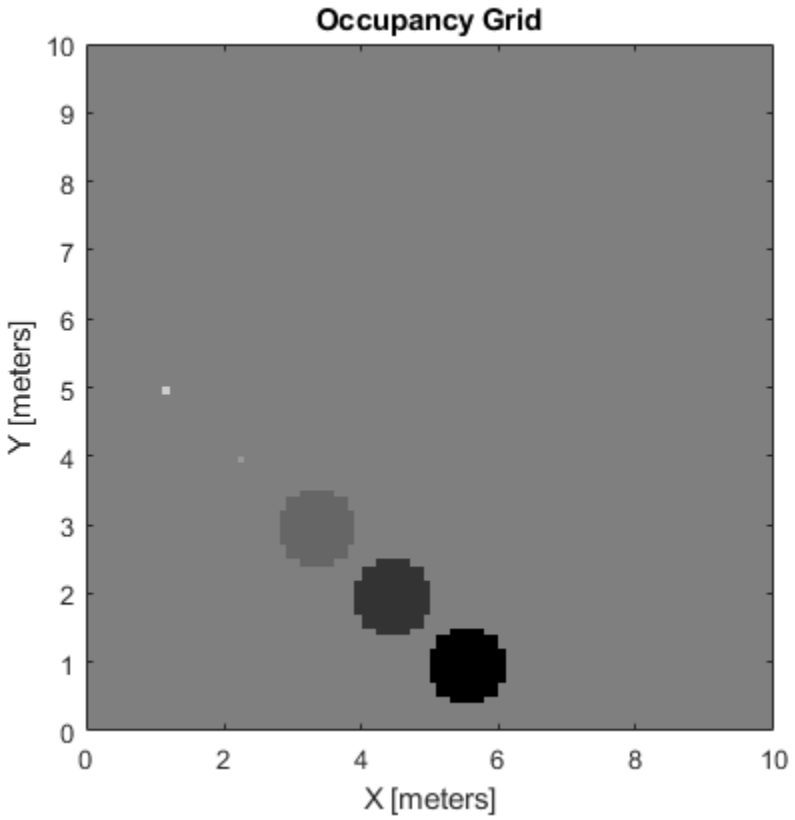
Update the occupancy of specific world locations with new probability values and display the map.

```
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 the occupied areas by a radius of 0.5 m. The larger occupancy values overwrite the smaller values.

```
inflate(map,0.5)
figure
show(map)
```

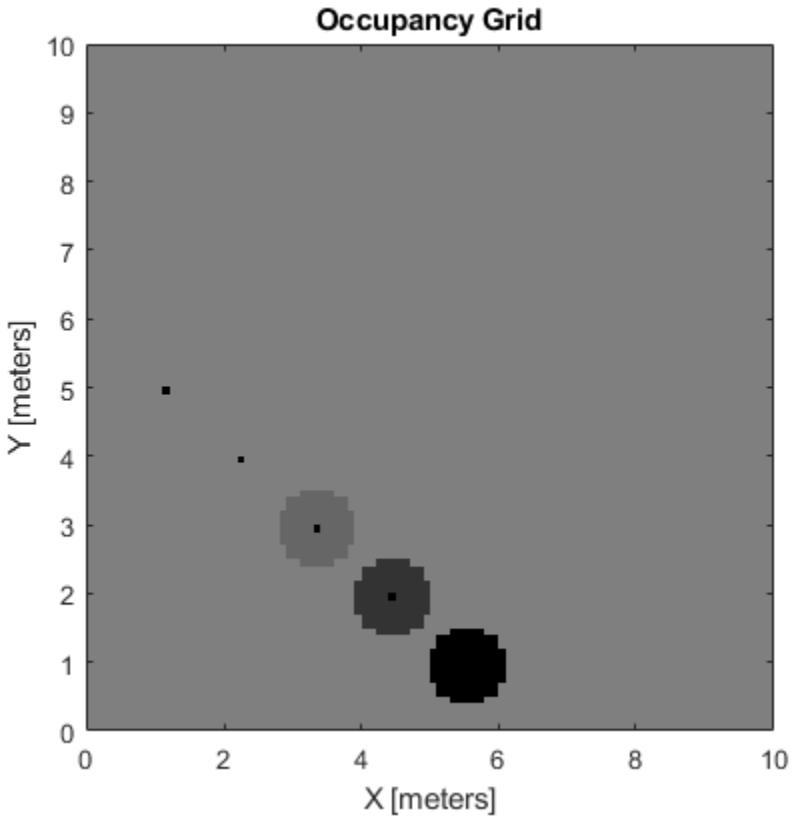


Get the grid locations from the world locations.

```
ij = world2grid(map,[x y]);
```

Set occupancy values for the grid locations.

```
setOccupancy(map,ij,ones(5,1),'grid')
figure
show(map)
```



Input Arguments

`map` — Map representation

`occupancyMap` object | `mapLayer` object | `multiLayerMap` object

Map representation, specified as a `occupancyMap`, `mapLayer`, or `multiLayerMap` object.

`xy` — World coordinates

n-by-2 matrix

World coordinates, specified as an *n*-by-2 matrix of `[x y]` pairs, where *n* is the number of world coordinates.

Data Types: `double`

Output Arguments

`ij` — Grid indices

n-by-2 matrix

Grid indices, returned as an *n*-by-2 matrix of `[i j]` pairs in `[row col]` format, where *n* is the number of grid positions. The grid cell locations are counted from the top left corner of the grid.

Data Types: `double`

See Also

Objects
`multiLayerMap` | `mapLayer` | `occupancyMap` | `binaryOccupancyMap`

Functions
`grid2world`

Topics
“Occupancy Grids”

Introduced in R2019b

rangeSensor

Simulate range-bearing sensor readings

Description

The `rangeSensor` System object is a range-bearing sensor that is capable of outputting range and angle measurements based on the given sensor pose and occupancy map. The range-bearing readings are based on the obstacles in the occupancy map.

To simulate a range-bearing sensor using this object:

1. Create the `rangeSensor` object and set its properties.
2. Call the object with arguments, as if it were a function.

To learn more about how System objects work, see What Are System Objects?

Creation

Syntax

```
rbsensor = rangeSensor
rbsensor = rangeSensor(Name,Value)
```

Description

`rbsensor = rangeSensor` returns a `rangeSensor` System object, `rbsensor`. The sensor is capable of outputting range and angle measurements based on the sensor pose and an occupancy map.

`rbsensor = rangeSensor(Name,Value)` sets properties for the sensor using one or more name-value pairs. Unspecified properties have default values. Enclose each property name in quotes.

Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see System Design in MATLAB Using System Objects.

Range — Minimum and maximum detectable range

`[0 20]` (default) | 1-by-2 positive real-valued vector

The minimum and maximum detectable range, specified as a 1-by-2 positive real-valued vector. Units are in meters.

Example: `[1 15]`

Tunable: Yes

Data Types: `single` | `double`

`HorizontalAngle` — Minimum and maximum horizontal detection angle

`[-pi pi]` (default) | 1-by-2 real-valued vector

Minimum and maximum horizontal detection angle, specified as a 1-by-2 real-valued vector. Units are in radians.

Example: `[-pi/3 pi/3]`

Data Types: `single` | `double`

`HorizontalAngleResolution` — Resolution of horizontal angle readings

`0.0244` (default) | positive scalar

Resolution of horizontal angle readings, specified as a positive scalar. The resolution defines the angular interval between two consecutive sensor readings. Units are in radians.

Example: `0.01`

Data Types: `single` | `double`

`RangeNoise` — Standard deviation of range noise

`0` (default) | positive scalar

The standard deviation of range noise, specified as a positive scalar. The range noise is modeled as a zero-mean white noise process with the specified standard deviation. Units are in meters.

Example: `0.01`

Tunable: Yes

Data Types: `single` | `double`

`HorizontalAngleNoise` — Standard deviation of horizontal angle noise

`0` (default) | positive scalar

The standard deviation of horizontal angle noise, specified as a positive scalar. The range noise is modeled as a zero-mean white noise process with the specified standard deviation. Units are in radians.

Example: `0.01`

Tunable: Yes

Data Types: `single` | `double`

`NumReadings` — Number of output readings

`258` (default) | positive integer

This property is read-only.

Number of output readings for each pose of the sensor, specified as a positive integer. This property depends on the `HorizonalAngle` and `HorizontalAngleResolution` properties.

Data Types: `single` | `double`

Usage

Syntax

```
[ranges,angles] = rbsensor(pose,map)
```

Description

`[ranges,angles] = rbsensor(pose,map)` returns the range and angle readings from the 2-D pose information and the ground-truth map.

Input Arguments

`pose` — Pose of sensor in map

N-by-3 real-valued matrix

Poses of the sensor in the 2-D map, specified as an *N*-by-3 real-valued matrix, where *N* is the number of poses to simulate the sensor. Each row of the matrix corresponds to a pose of the sensor in the order of [*x*, *y*, *θ*]. *x* and *y* represent the position of the sensor in the map frame. The units of *x* and *y* are in meters. *θ* is the heading angle of the sensor with respect to the positive *x*-direction of the map frame. The units of *θ* are in radians.

`map` — Ground-truth map

`occupancyMap` object | `binaryOccupancyMap` object

Ground-truth map, specified as an `occupancyMap` or a `binaryOccupancyMap` object. For the `occupancyMap` input, the range-bearing sensor considers a cell as occupied and returns a range reading if the occupancy probability of the cell is greater than the value specified by the `OccupiedThreshold` property of the occupancy map.

Output Arguments

`ranges` — Range readings

R-by-*N* real-valued matrix

Range readings, specified as an *R*-by-*N* real-valued matrix. *N* is the number of poses for which the sensor is simulated, and *R* is the number of sensor readings per pose of the sensor. *R* is same as the value of the `NumReadings` property.

`angles` — Angle readings

R-by-1 real-valued vector

Angle readings, specified as an *R*-by-1 real-valued vector. *R* is the number of sensor readings per pose of the sensor. *R* is same as the value of the `NumReadings` property.

Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

Common to All System Objects

step Run System object algorithm
clone Create duplicate System object

Examples

Obtain Range and Bearing Readings

Create a range-bearing sensor.

```
rbsensor = rangeSensor;
```

Specify the pose of the sensor and the ground-truth map.

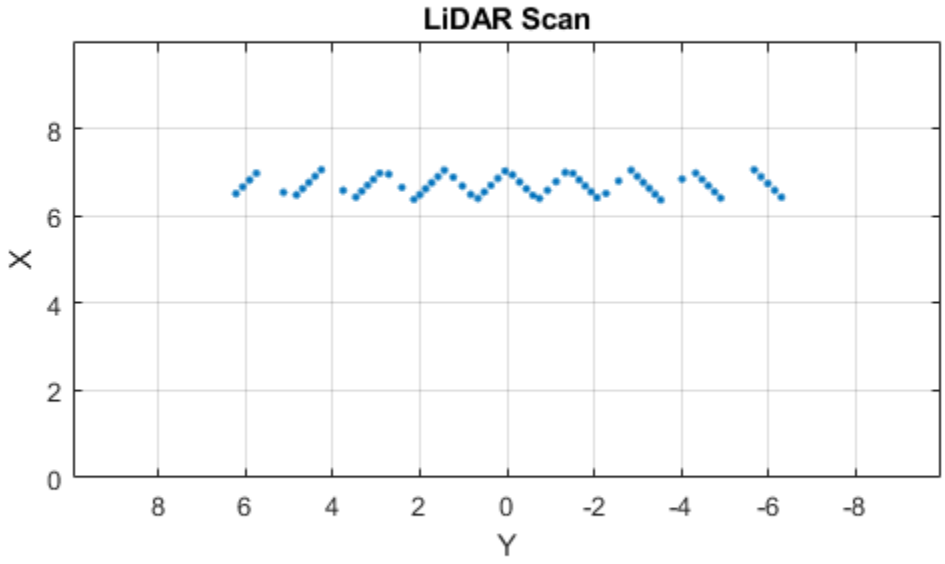
```
truePose = [0 0 pi/4];
trueMap = binaryOccupancyMap(eye(10));
```

Generate the sensor readings.

```
[ranges, angles] = rbsensor(truePose, trueMap);
```

Visualize the results using `lidarScan`.

```
scan = lidarScan(ranges, angles);
figure
plot(scan)
```



Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

`occupancyMap` | `binaryOccupancyMap` | `lidarScan`

Introduced in R2019b

world2local

Convert world coordinates to local coordinates

Syntax

```
xyLocal = world2local(map,xy)
```

Description

`xyLocal = world2local(map,xy)` converts an array of world coordinates to local coordinates.

Input Arguments

`map` — Map representation

`occupancyMap` object | `mapLayer` object | `multiLayerMap` object

Map representation, specified as a `occupancyMap`, `mapLayer`, or `multiLayerMap` object.

`xy` — World coordinates

n-by-2 vertical array

World coordinates, specified as an *n*-by-2 vertical array of `[x y]` pairs, where *n* is the number of world coordinates.

Output Arguments

`xyLocal` — Local coordinates

n-by-2 vertical array

Local coordinates, specified as an *n*-by-2 vertical array of `[x y]` pairs, where *n* is the number of local coordinates.

See Also

Objects

`multiLayerMap` | `mapLayer` | `occupancyMap` | `binaryOccupancyMap`

Functions

`grid2world` | `local2world`

Introduced in R2019b

occupancyMap3D

Create 3-D occupancy map

Description

The `occupancyMap3D` class stores a 3-D map and map information. The map is stored as probabilistic values in an octree data structure on page 2-853. The class handles arbitrary environments and expands its size dynamically based on observation inputs. You can add observations as point clouds or as specific *xyz* locations. These observations update the probability values. Probabilistic values represent the occupancy of locations. The octree data structure trims data appropriately to remain efficient both in memory and on disk.

Creation

Syntax

```
omap = occupancyMap3D
omap = occupancyMap3D(res)
omap = occupancyMap3D(res,Name,Value)
```

Description

`omap = occupancyMap3D` creates an empty 3-D occupancy map with no observations and default property values.

`omap = occupancyMap3D(res)` specifies a map resolution in cells/meter and sets the `Resolution` property.

`omap = occupancyMap3D(res,Name,Value)` creates an object with additional options specified by one or more `Name,Value` pair arguments. For example, `'FreeThreshold',0.25` sets the threshold to consider cells obstacle-free as a probability value of 0.25. Enclose each property name in quotes.

Properties

`Resolution` — Grid resolution

`1` (default) | positive scalar

Grid resolution in cells per meter, specified as a scalar. Specify resolution on construction. Inserting observations with precisions higher than this value are rounded down and applied at this resolution.

`FreeThreshold` — Threshold to consider cells as obstacle-free

`0.2` (default) | positive scalar

Threshold to consider cells as obstacle-free, specified as a positive scalar. Probability values below this threshold are considered obstacle-free.

`OccupiedThreshold` — Threshold to consider cells as occupied
`0.65` (default) | positive scalar

Threshold to consider cells as occupied, specified as a positive scalar. Probability values above this threshold are considered occupied.

`ProbabilitySaturation` — Saturation limits on probability values
`[0.001 0.999]` (default) | `[min max]` vector

Saturation limits on probability values, specified as a `[min max]` vector. Values above or below these saturation values are set to the `min` or `max` values. This property reduces oversaturating of cells when incorporating multiple observations.

Object Functions

checkOccupancy	Check if locations are free or occupied
getOccupancy	Get occupancy probability of locations
inflate	Inflate map
insertPointCloud	Insert 3-D points or point cloud observation into map
rayIntersection	Find intersection points of rays and occupied map cells
setOccupancy	Set occupancy probability of locations
show	Show occupancy map
updateOccupancy	Update occupancy probability at locations

Examples

Create 3-D Occupancy Map and Inflate Points

The `occupancyMap3D` object stores obstacles in 3-D space, using sensor observations to map an environment. Create a map and add points from a point cloud to identify obstacles. Then inflate the obstacles in the map to ensure safe operating space around obstacles.

Create an `occupancyMap3D` object with a map resolution of 10 cells/meter.

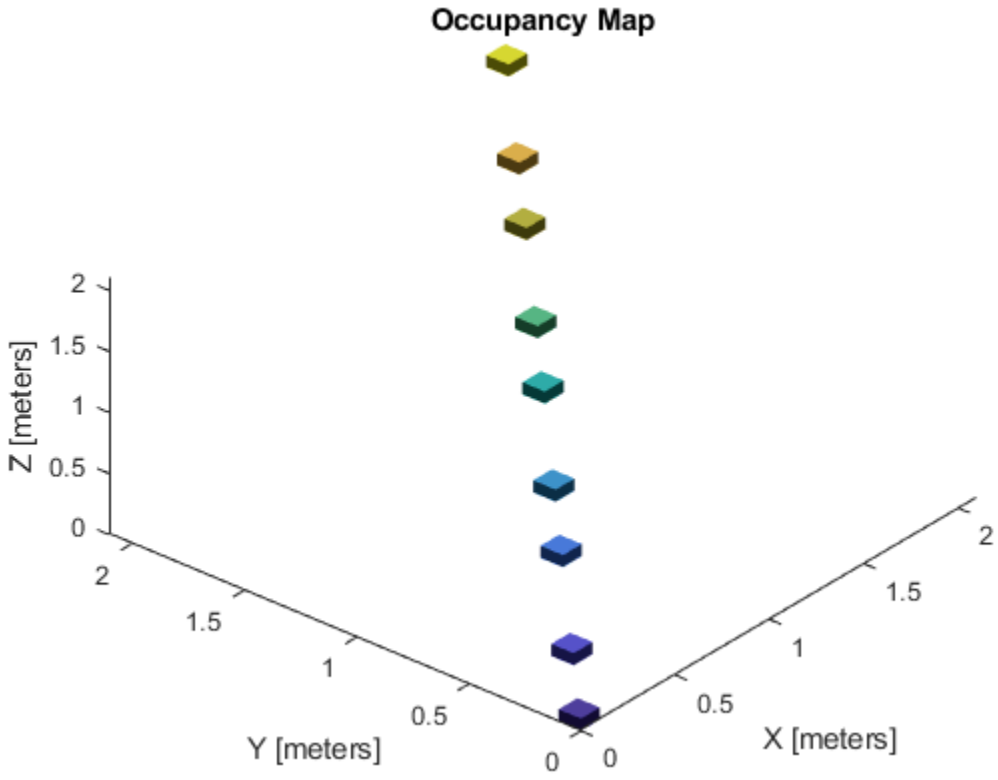
```
map3D = occupancyMap3D(10);
```

Define a set of 3-D points as an observation from a pose `[x y z qw qx qy qz]`. This pose is for the sensor that observes these points and is centered on the origin. Define two sets of points to insert multiple observations.

```
pose = [ 0 0 0 1 0 0 0];

points = repmat((0:0.25:2)', 1, 3);
points2 = [(0:0.25:2)' (2:-0.25:0)' (0:0.25:2)'];
maxRange = 5;
```

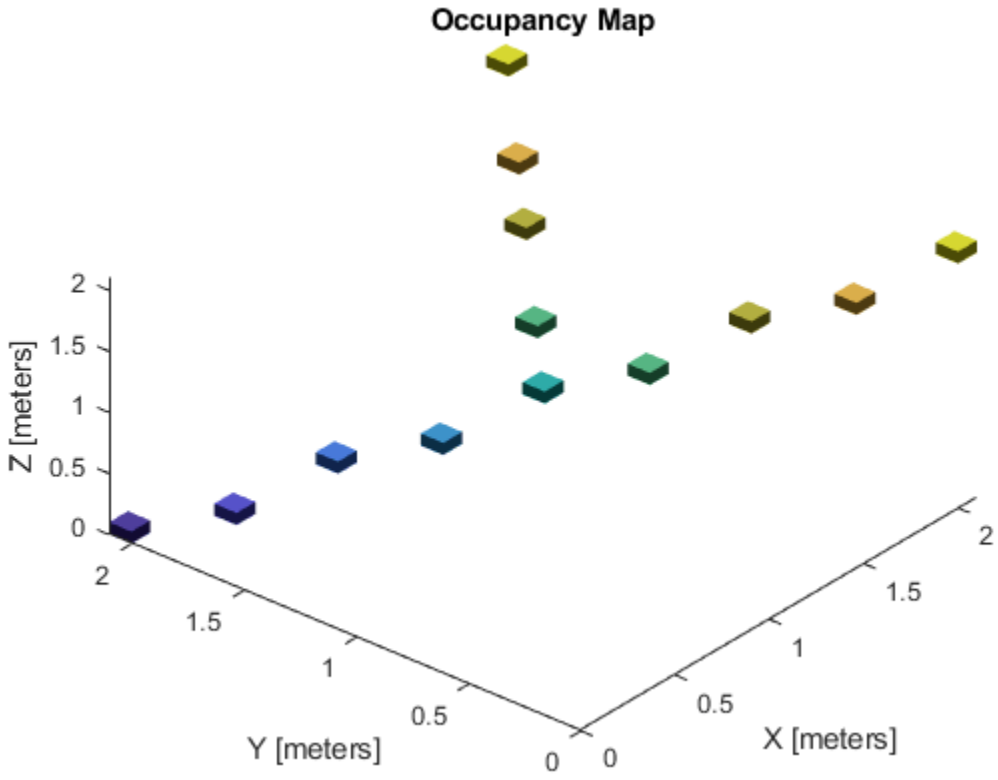
Insert the first set of points using `insertPointCloud`. The function uses the sensor pose and the given points to insert observations into the map. The colors displayed correlate to the height of the point only for illustrative purposes.

```
insertPointCloud(map3D,pose,points,maxRange)
show(map3D)
```



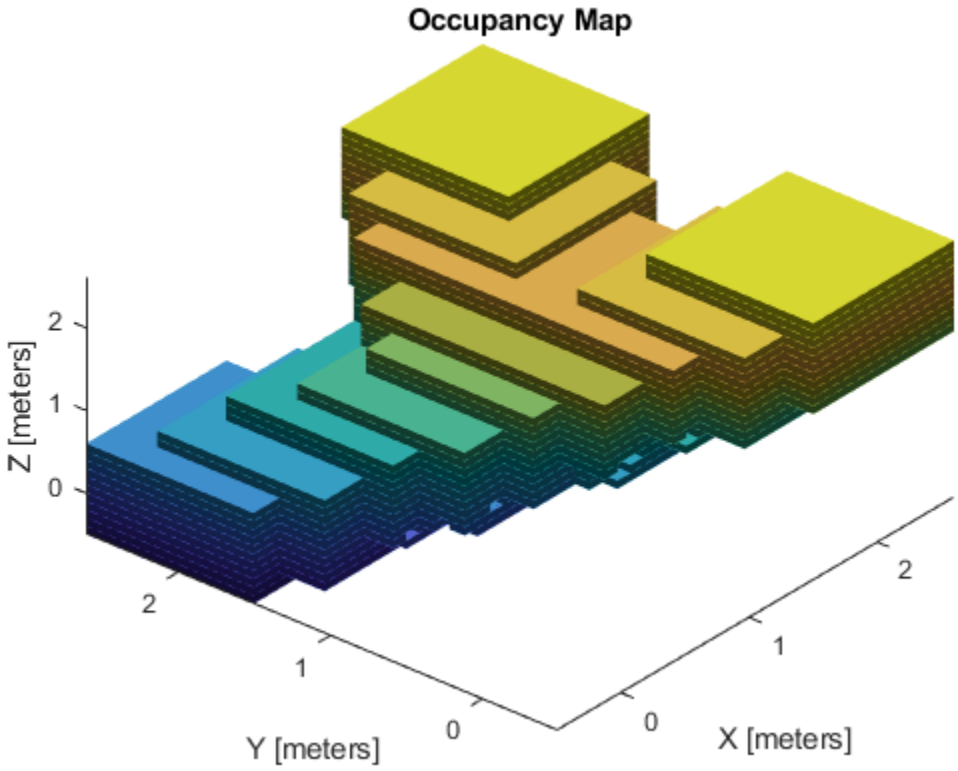
Insert the second set of points. The ray between the sensor pose (origin) and these points overlap points from the previous insertion. Therefore, the free space between the sensor and the new points are updated and marked as free space.

```
insertPointCloud(map3D,pose,points2,maxRange)
show(map3D)
```



Inflate the map to add a buffer zone for safe operation around obstacles. Define the vehicle radius and safety distance and use the sum of these values to define the inflation radius for the map.

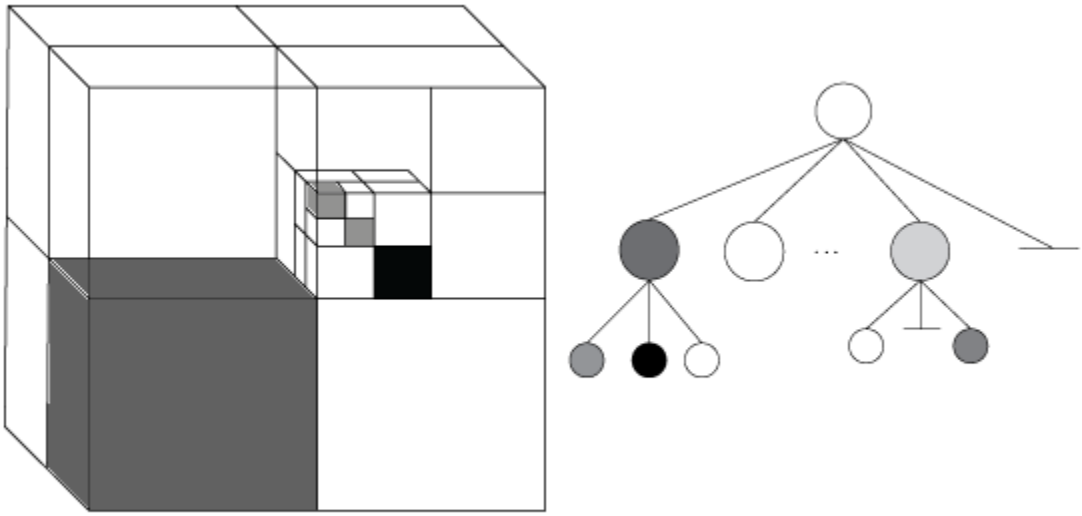
```
vehicleRadius = 0.2;
safetyRadius = 0.3;
inflationRadius = vehicleRadius + safetyRadius;
inflate(map3D, inflationRadius);

show(map3D)
```



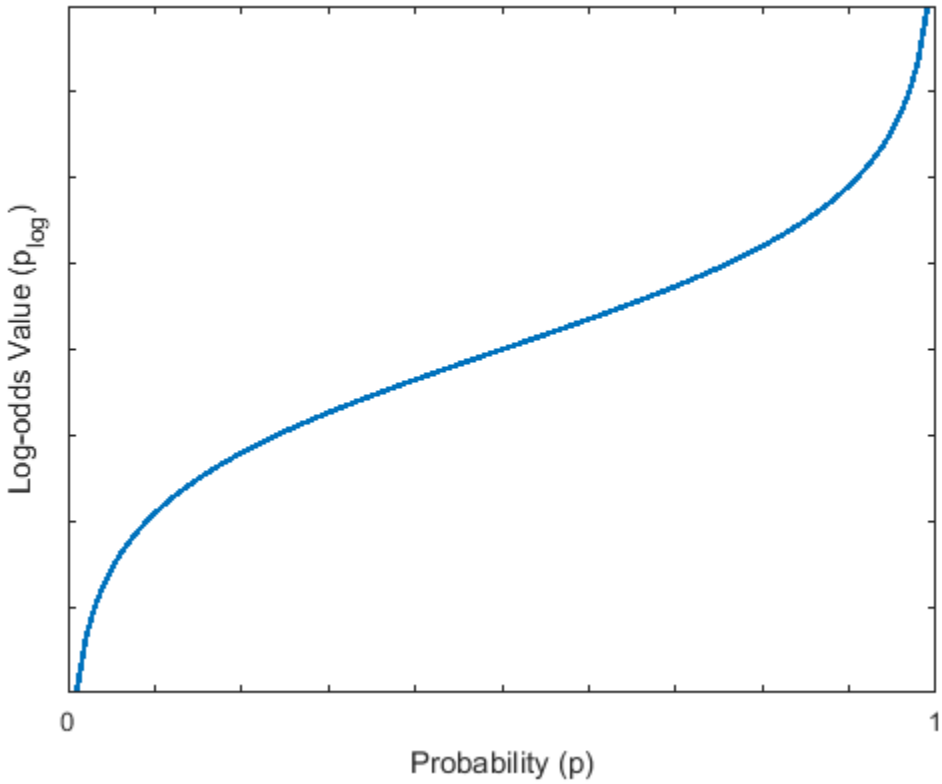
Algorithms

Octree Data Structure

The octree data structure is a hierarchical structure used for subdivision of an environment into cubic volumes called voxels. For a given map volume, the space is recursively subdivided into eight voxels until achieving a desired map resolution (voxel size) is achieved. This subdivision can be represented as a tree, which stores probability values for locations in the map.



The probability values in the tree have a log-odds representation. Using this representation, locations easily recover from dynamic observations and numerical errors due to small probabilities are reduced. To remain efficient in memory, lower branches of the tree are pruned in the structure if they share the same occupancy values using this log-odds representation.



The class internally handles the organization of this data structure, including the pruning of branches. Specify all observations as spatial coordinates when using functions such as `setOccupancy`, `getOccupancy`, or `insertPointCloud`. Insertions into the tree, and navigation through the tree, is determined based on the spatial coordinates and the resolution of the map.

References

[1] Hornung, Armin, Kai M. Wurm, Maren Bennewitz, Cyrill Stachniss, and Wolfram Burgard. "OctoMap: an efficient probabilistic 3D mapping framework based on octrees." *Autonomous Robots*, Vol. 34, No. 3, 2013, pp. 189–206.. doi:10.1007/s10514-012-9321-0.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

Classes

`occupancyMap` | `binaryOccupancyMap`

Functions

`insertPointCloud` | `inflate` | `setOccupancy` | `show` | `rosReadOccupancyMap3D`

Introduced in R2019b

checkOccupancy

Check if locations are free or occupied

Syntax

```
iOccval = checkOccupancy(map3D,xyz)
```

Description

`iOccval = checkOccupancy(map3D,xyz)` returns an array of occupancy values specified at the *xyz* locations using the `OccupiedThreshold` and `FreeThreshold` properties of the input `occupancyMap3D` object. Each row is a separate *xyz* location in the map to check the occupancy of. Occupancy values can be obstacle-free (0), occupied (1), or unknown (–1).

Examples

Check Occupancy Status and Get Occupancy Values in 3-D Occupancy Map

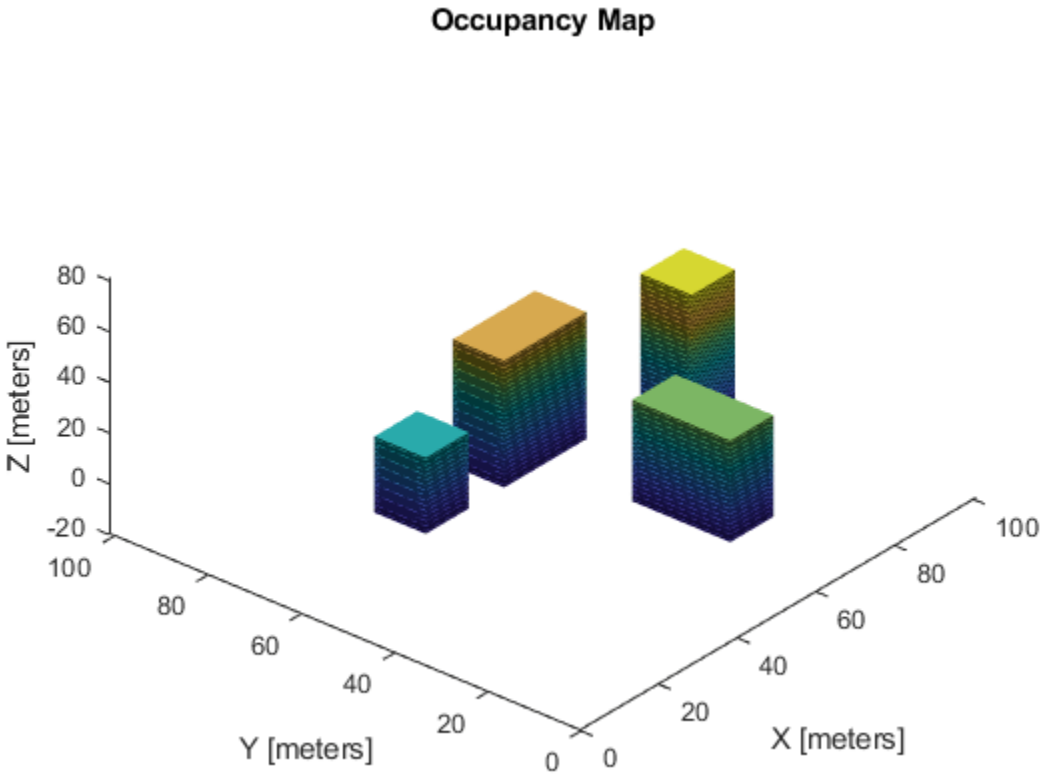
Import a 3-D occupancy map.

```
map3D = importOccupancyMap3D("citymap.ot")
```

```
map3D =
  occupancyMap3D with properties:

    ProbabilitySaturation: [1.0000e-03 0.9990]
               Resolution: 1
        OccupiedThreshold: 0.6500
            FreeThreshold: 0.2000
```

Display the map.

```
show(map3D)
```



Check the occupancy statuses of different locations and get their occupancy values.

```
iOccVal1 = checkOccupancy(map3D,[50 15 0])
```

```
iOccVal1 = 0
```

```
OccVal1 = getOccupancy(map3D,[50 15 0])
```

```
OccVal1 = 0.0019
```

```
iOccVal2 = checkOccupancy(map3D,[50 15 15])
```

```
iOccVal2 = 1
```

```
OccVal2 = getOccupancy(map3D,[50 15 15])
```

```
OccVal2 = 0.6500
```

```
iOccVal3 = checkOccupancy(map3D,[50 15 45])
```

```
iOccVal3 = -1
```

```
OccVal3 = getOccupancy(map3D,[50 15 45])
```

```
OccVal3 = 0.5000
```

Input Arguments

`map3D` — 3-D occupancy map

`occupancyMap3D` object

3-D occupancy map, specified as an `occupancyMap3D` object.

`xyz` — World coordinates

n-by-3 matrix

World coordinates, specified as an *n*-by-3 matrix of `[x y z]` points, where *n* is the number of world coordinates.

Output Arguments

`iOccval` — Interpreted occupancy values

column vector

Interpreted occupancy values, returned as a column vector with the same length as `xyz`.

Occupancy values can be obstacle-free (0), occupied (1), or unknown (–1). These values are determined from the actual probability values and the `OccupiedThreshold` and `FreeThreshold` properties of the `map3D` object.

See Also

Classes

`occupancyMap3D` | `lidarSLAM` | `occupancyMap`

Functions

`insertPointCloud` | `inflate` | `setOccupancy` | `show`

Introduced in R2019b

getOccupancy

Get occupancy probability of locations

Syntax

```
occval = getOccupancy(map3D,xyz)
```

Description

`occval = getOccupancy(map3D,xyz)` returns an array of probability occupancy values at the specified *xyz* locations in the `occupancyMap3D` object. 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.

Examples

Check Occupancy Status and Get Occupancy Values in 3-D Occupancy Map

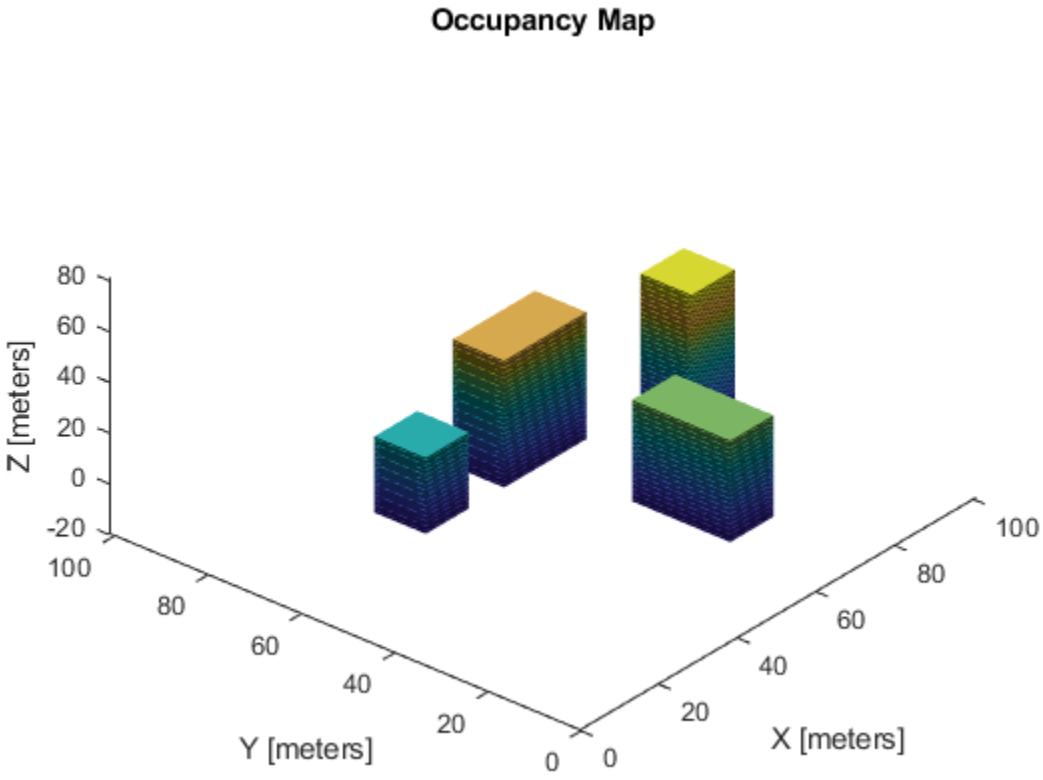
Import a 3-D occupancy map.

```
map3D = importOccupancyMap3D("citymap.ot")
```

```
map3D =
  occupancyMap3D with properties:

    ProbabilitySaturation: [1.0000e-03 0.9990]
               Resolution: 1
        OccupiedThreshold: 0.6500
            FreeThreshold: 0.2000
```

Display the map.

```
show(map3D)
```



Check the occupancy statuses of different locations and get their occupancy values.

```
iOccVal1 = checkOccupancy(map3D,[50 15 0])
```

```
iOccVal1 = 0
```

```
OccVal1 = getOccupancy(map3D,[50 15 0])
```

```
OccVal1 = 0.0019
```

```
iOccVal2 = checkOccupancy(map3D,[50 15 15])
```

```
iOccVal2 = 1
```

```
OccVal2 = getOccupancy(map3D,[50 15 15])
```

```
OccVal2 = 0.6500
```

```
iOccVal3 = checkOccupancy(map3D,[50 15 45])
```

```
iOccVal3 = -1
```

```
OccVal3 = getOccupancy(map3D,[50 15 45])
```

```
OccVal3 = 0.5000
```

Input Arguments

`map3D` — 3-D occupancy map

`occupancyMap3D` object

3-D occupancy map, specified as an `occupancyMap3D` object.

`xyz` — World coordinates

n-by-3 matrix

World coordinates, specified as an *n*-by-3 matrix of `[x y z]` points, where *n* is the number of world coordinates.

Output Arguments

`occval` — Probability occupancy values

column vector

Probability occupancy values, returned as a column vector with the same length as `xyz`.

Values close to 0 represent certainty that the cell is not occupied and obstacle-free.

See Also

Classes

`occupancyMap3D` | `lidarSLAM` | `occupancyMap`

Functions

`insertPointCloud` | `inflate` | `setOccupancy` | `show`

Introduced in R2019b

inflate

Inflate map

Syntax

```
inflate(map3D,radius)
```

Description

`inflate(map3D,radius)` inflates each occupied position of the specified in the input `occupancyMap3D` object by the `radius` specified in meters. `radius` is rounded up to the nearest equivalent cell based on the resolution of the map. This inflation increases the size of the occupied locations in the map.

Examples

Get Ray Intersection Points on 3-D Occupancy Map

Import a 3-D occupancy map.

```
map3D = importOccupancyMap3D("citymap.ot")
```

```
map3D =
  occupancyMap3D with properties:

    ProbabilitySaturation: [1.0000e-03 0.9990]
               Resolution: 1
        OccupiedThreshold: 0.6500
            FreeThreshold: 0.2000
```

Inflate the occupied areas by a radius of 1 m. Display the map.

```
inflate(map3D,1)
show(map3D)
```

Find the intersection points of rays and occupied map cells.

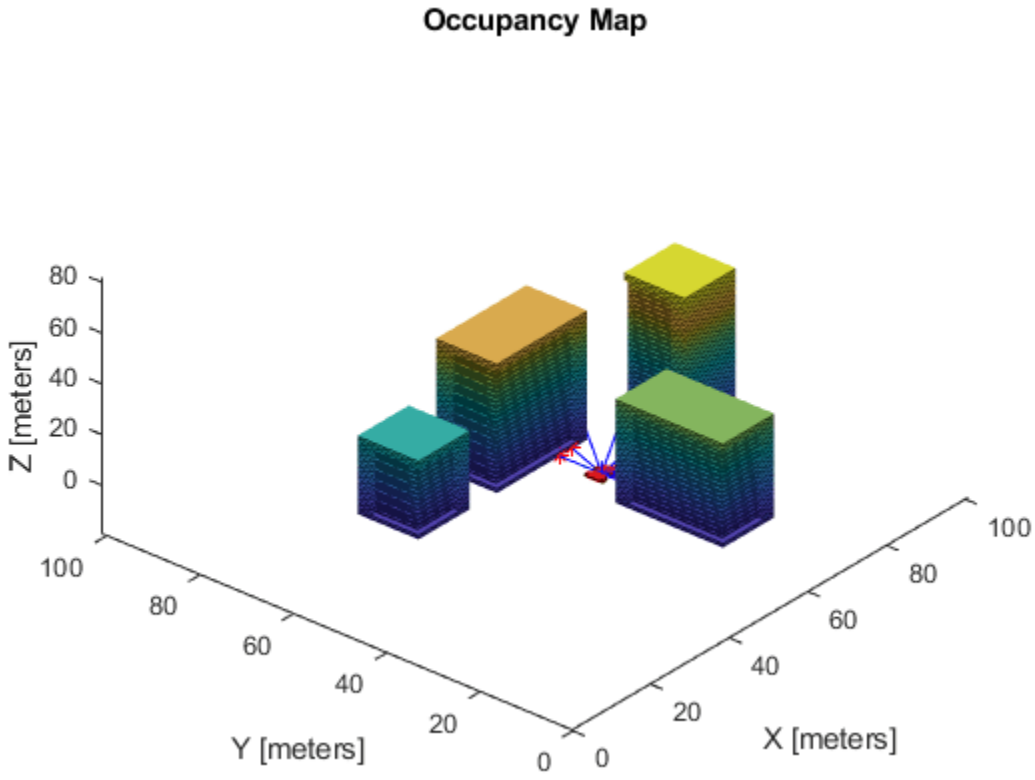
```
numRays = 10;
angles = linspace(-pi/2,pi/2,numRays);
directions = [cos(angles); sin(angles); zeros(1,numRays)]';
sensorPose = [55 40 1 1 0 0 0];
maxrange = 15;
[intersectionPts,isOccupied] = rayIntersection(map3D,sensorPose,directions,maxrange)
```

```
intersectionPts = 10×3

   55.0000   32.0000    1.0000
   57.9118   32.0000    1.0000
   61.7128   32.0000    1.0000
   67.9904   32.5000    1.0000
```

```
   69.0000   37.5314    1.0000
   69.0000   42.4686    1.0000
   67.9904   47.5000    1.0000
   64.6418   51.4907    1.0000
   58.2757   49.0000    1.0000
   55.0000   49.0000    1.0000


isOccupied = 10×1

     1
     1
     1
    -1
     1
     1
    -1
    -1
     1
     1
```

Plot the intersection points and rays from the pose.

```
hold on
plotTransforms(sensorPose(1:3),sensorPose(4:end),...
               'FrameSize',5,'MeshFilePath','groundvehicle.stl') % Vehicle sensor pose
for i = 1:numRays
    plot3([sensorPose(1),intersectionPts(i,1)],...
          [sensorPose(2),intersectionPts(i,2)],...
          [sensorPose(3),intersectionPts(i,3)],'-b') % Plot rays
    if isOccupied(i) == 1
        plot3(intersectionPts(i,1),intersectionPts(i,2),intersectionPts(i,3),'*r') % Intersectio
    end
end
```



Input Arguments

`map3D` — 3-D occupancy map
`occupancyMap3D` object

3-D occupancy map, specified as an `occupancyMap3D` object.

`radius` — Amount to inflate occupied locations
scalar

Amount to inflate occupied locations, specified as a scalar. `radius` is rounded up to the nearest cell value.

See Also

Classes
`occupancyMap3D` | `lidarSLAM` | `occupancyMap`

Functions
`insertPointCloud` | `setOccupancy` | `show`

Introduced in R2019b

insertPointCloud

Insert 3-D points or point cloud observation into map

Syntax

```
insertPointCloud(map3D,pose,points,maxrange)
insertPointCloud(map3D,pose,ptcloud,maxrange)
```

Description

`insertPointCloud(map3D,pose,points,maxrange)` inserts one or more sensor observations at the given `points` in the occupancy map, `map3D`. Occupied points are updated with an observation of 0.7. All other points between the sensor `pose` and `points` are treated as obstacle-free and updated with an observation of 0.4. Points outside `maxrange` are not updated. `NaN` values are ignored.

`insertPointCloud(map3D,pose,ptcloud,maxrange)` inserts a `ptcloud` object into the map.

Examples

Create 3-D Occupancy Map and Inflate Points

The `occupancyMap3D` object stores obstacles in 3-D space, using sensor observations to map an environment. Create a map and add points from a point cloud to identify obstacles. Then inflate the obstacles in the map to ensure safe operating space around obstacles.

Create an `occupancyMap3D` object with a map resolution of 10 cells/meter.

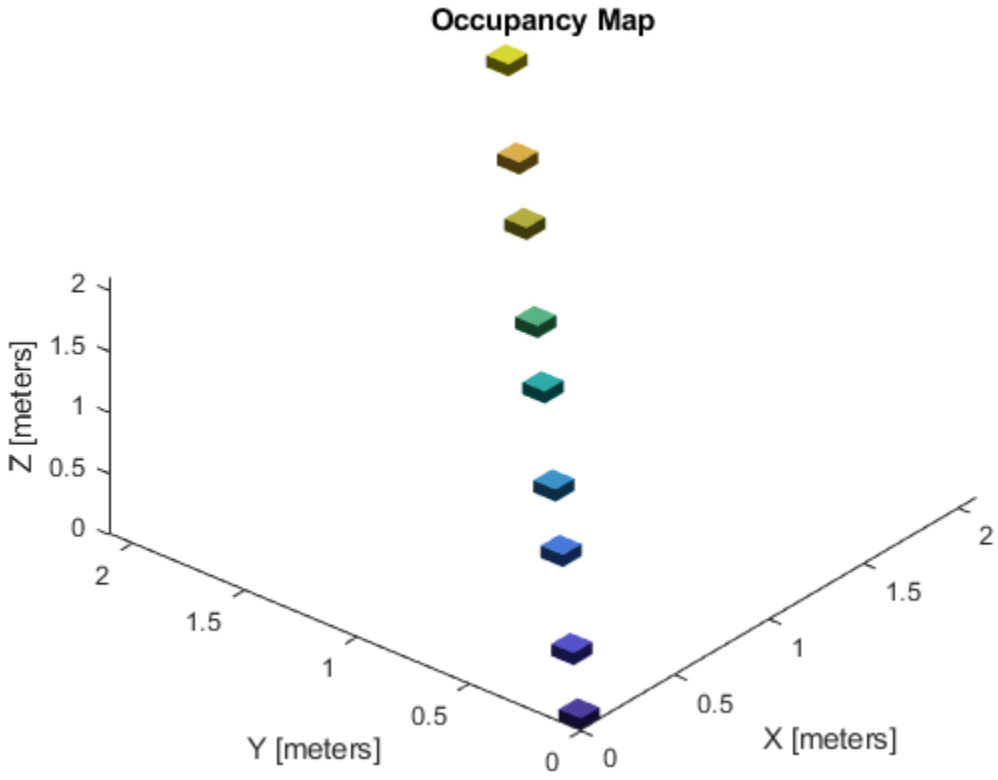
```
map3D = occupancyMap3D(10);
```

Define a set of 3-D points as an observation from a pose `[x y z qw qx qy qz]`. This pose is for the sensor that observes these points and is centered on the origin. Define two sets of points to insert multiple observations.

```
pose = [ 0 0 0 1 0 0 0];

points = repmat((0:0.25:2)', 1, 3);
points2 = [(0:0.25:2)' (2:-0.25:0)' (0:0.25:2)'];
maxRange = 5;
```

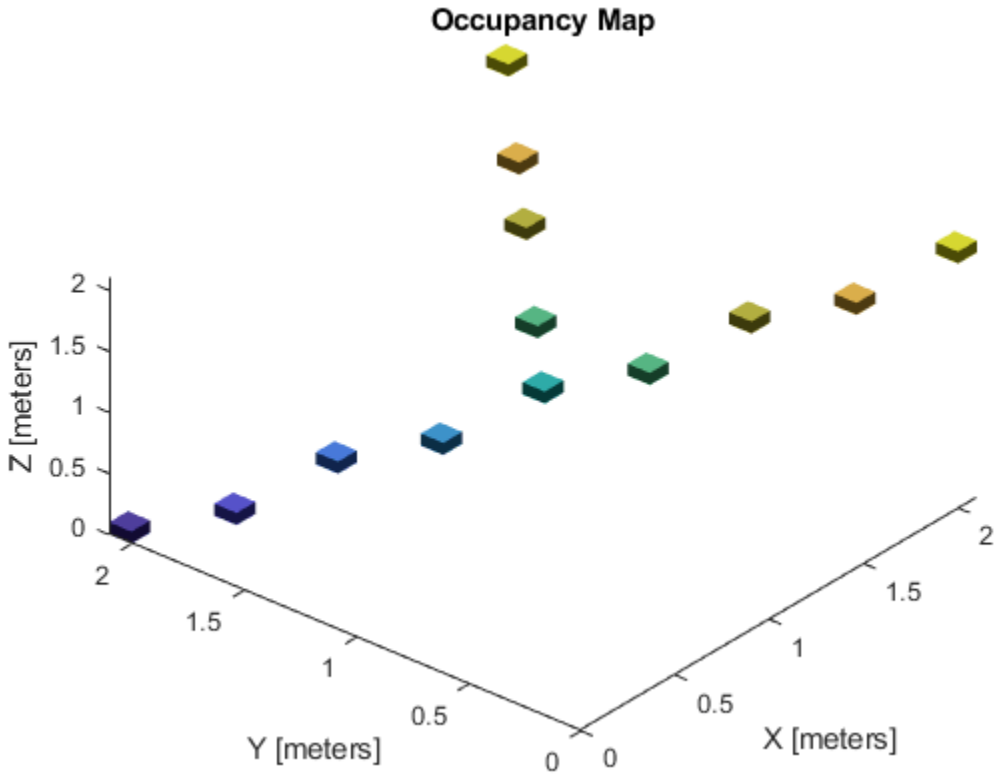
Insert the first set of points using `insertPointCloud`. The function uses the sensor pose and the given points to insert observations into the map. The colors displayed correlate to the height of the point only for illustrative purposes.

```
insertPointCloud(map3D,pose,points,maxRange)
show(map3D)
```



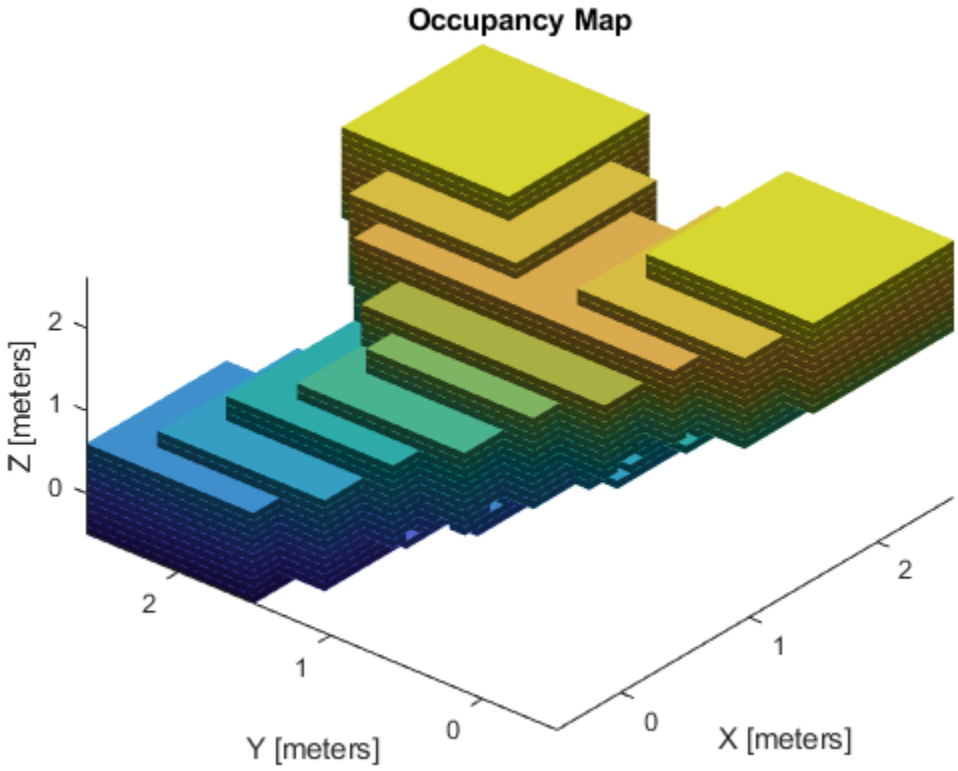
Insert the second set of points. The ray between the sensor pose (origin) and these points overlap points from the previous insertion. Therefore, the free space between the sensor and the new points are updated and marked as free space.

```
insertPointCloud(map3D,pose,points2,maxRange)
show(map3D)
```



Inflate the map to add a buffer zone for safe operation around obstacles. Define the vehicle radius and safety distance and use the sum of these values to define the inflation radius for the map.

```
vehicleRadius = 0.2;
safetyRadius = 0.3;
inflationRadius = vehicleRadius + safetyRadius;
inflate(map3D, inflationRadius);

show(map3D)
```



Input Arguments

`map3D` — 3-D occupancy map
`occupancyMap3D` object

3-D occupancy map, specified as a `occupancyMap3D` object.

`points` — Points of point cloud
n-by-3 matrix

Points of point cloud in sensor coordinates, specified as an *n*-by-3 matrix of `[x y z]` points, where *n* is the number of points in the point cloud.

`ptcloud` — Point cloud reading
`pointCloud` object

Point cloud reading, specified as a `pointCloud` object.

Note Using `pointCloud` objects requires Computer Vision Toolbox.

`pose` — Position and orientation of vehicle
`[x y z qw qx qy qz]` vector

Position and orientation of vehicle, specified as an `[x y z qw qx qy qz]` vector. The vehicle pose is an *xyz*-position vector with a quaternion orientation vector specified as `[qw qx qy qz]`.

`maxrange` — Maximum range of sensor
scalar

Maximum range of point cloud sensor, specified as a scalar. Points outside this range are ignored.

See Also

Classes
`occupancyMap3D` | `lidarSLAM` | `occupancyMap`

Functions
`inflate` | `setOccupancy` | `show`

Introduced in R2019b

rayIntersection

Find intersection points of rays and occupied map cells

Syntax

```
[intersectionPts,isOccupied] = rayIntersection(map3D,sensorPose,directions,
maxrange)
[intersectionPts,isOccupied] = rayIntersection(map3D,sensorPose,directions,
maxrange,ignoreUnknown)
```

Description

`[intersectionPts,isOccupied] = rayIntersection(map3D,sensorPose,directions, maxrange)` returns intersection points of rays in the specified map, `map3D`. Rays emanate from the specified `sensorPose` at the given orientations, `directions`. Intersection points are returned in the world coordinate frame. Use `isOccupied` to determine if the intersection point is at the sensor max range or if it intersects an obstacle.

`[intersectionPts,isOccupied] = rayIntersection(map3D,sensorPose,directions, maxrange,ignoreUnknown)` additionally accepts optional arguments for the sensors max range and whether to ignore unknown values. By default, the rays extend to the map boundary and unknown values are ignored.

Examples

Get Ray Intersection Points on 3-D Occupancy Map

Import a 3-D occupancy map.

```
map3D = importOccupancyMap3D("citymap.ot")
```

```
map3D =
  occupancyMap3D with properties:

    ProbabilitySaturation: [1.0000e-03 0.9990]
               Resolution: 1
        OccupiedThreshold: 0.6500
            FreeThreshold: 0.2000
```

Inflate the occupied areas by a radius of 1 m. Display the map.

```
inflate(map3D,1)
show(map3D)
```

Find the intersection points of rays and occupied map cells.

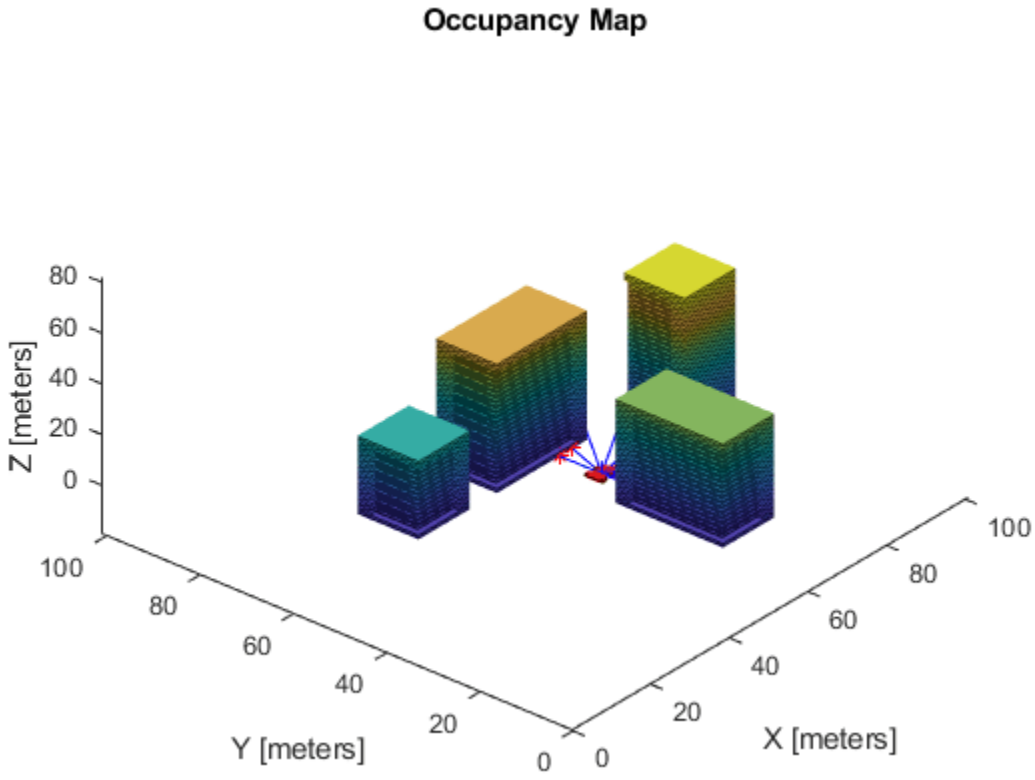
```
numRays = 10;
angles = linspace(-pi/2,pi/2,numRays);
directions = [cos(angles); sin(angles); zeros(1,numRays)]';
```

```
sensorPose = [55 40 1 1 0 0 0];
maxrange = 15;
[intersectionPts,isOccupied] = rayIntersection(map3D,sensorPose,directions,maxrange)
```

```
intersectionPts = 10×3

   55.0000   32.0000    1.0000
   57.9118   32.0000    1.0000
   61.7128   32.0000    1.0000
   67.9904   32.5000    1.0000
   69.0000   37.5314    1.0000
   69.0000   42.4686    1.0000
   67.9904   47.5000    1.0000
   64.6418   51.4907    1.0000
   58.2757   49.0000    1.0000
   55.0000   49.0000    1.0000


isOccupied = 10×1

     1
     1
     1
    -1
     1
     1
    -1
    -1
     1
     1
```

Plot the intersection points and rays from the pose.

```
hold on
plotTransforms(sensorPose(1:3),sensorPose(4:end),...
               'FrameSize',5,'MeshFilePath','groundvehicle.stl') % Vehicle sensor pose
for i = 1:numRays
    plot3([sensorPose(1),intersectionPts(i,1)],...
          [sensorPose(2),intersectionPts(i,2)],...
          [sensorPose(3),intersectionPts(i,3)],'-b') % Plot rays
    if isOccupied(i) == 1
        plot3(intersectionPts(i,1),intersectionPts(i,2),intersectionPts(i,3),'*r') % Intersectio
    end
end
```



Input Arguments

`map3D` — 3-D occupancy map

`occupancyMap3D` object

3-D occupancy map, specified as a `occupancyMap3D` object.

`sensorPose` — Position and orientation of sensor

`[x y z qw qx qy qz]` vector

Position and orientation of sensor, specified as an `[x y z qw qx qy qz]` vector. The vehicle pose is an *xyz*-position vector with a quaternion orientation vector specified as `[qw qx qy qz]`.

`directions` — Orientation of rays emanating from sensor

n-by-3 `[dx dy dz]` matrix | *n*-by-2 `[az el]` matrix

Orientation of rays emanating from the sensor relative to the sensor coordinate frame, specified as an *n*-by-3 `[dx dy dz]` matrix or *n*-by-2 `[az el]` matrix.

- `[dx dy dz]` is a directional vector in *xyz*-coordinates.
- `[az el]` is a vector with azimuth angle, `az`, measured from the positive *x* direction to the positive *y* direction, and elevation angle from the *xy*-plane to the positive *z*-direction in sensor coordinate frame.

`maxrange` — Maximum range of sensor
scalar

Maximum range of laser range sensor, specified as a scalar in meters. Range values greater than or equal to `maxrange` are considered free along the whole length of the ray, up to `maxrange`.

`ignoreUnknown` — Interpret unknown values as free or occupied
`1` (default) | `0`

Interpret unknown values in the map as free or occupied specified as `1` or `0`. Set this value to `0` to assume unknown values are occupied.

Output Arguments

`intersectionPts` — Intersection points
n-by-3 matrix

Intersection points, returned as *n*-by-3 matrix of `[x y z]` points in the world frame, where *n* is the length of `directions`.

`isOccupied` — Occupancy status of ray end points
vector of zeroes and ones

Occupancy status of ray end points, returned as a vector of zeroes and ones. Use `isOccupied` to determine if the intersection point is at the sensor max range or if it intersects an obstacle.

See Also

Classes
`occupancyMap3D` | `lidarSLAM` | `occupancyMap`

Functions
`insertPointCloud` | `inflate` | `setOccupancy` | `show`

Introduced in R2020a

setOccupancy

Set occupancy probability of locations

Syntax

```
setOccupancy(map3D,xyz,occval)
```

Description

`setOccupancy(map3D,xyz,occval)` assigns the occupancy values to each specified `xyz` coordinate in the 3-D occupancy map.

Examples

Create and Export 3-D Occupancy Map

Create an `occupancyMap3D` object.

```
map3D = occupancyMap3D;
```

Create a ground plane and set occupancy values to 0.

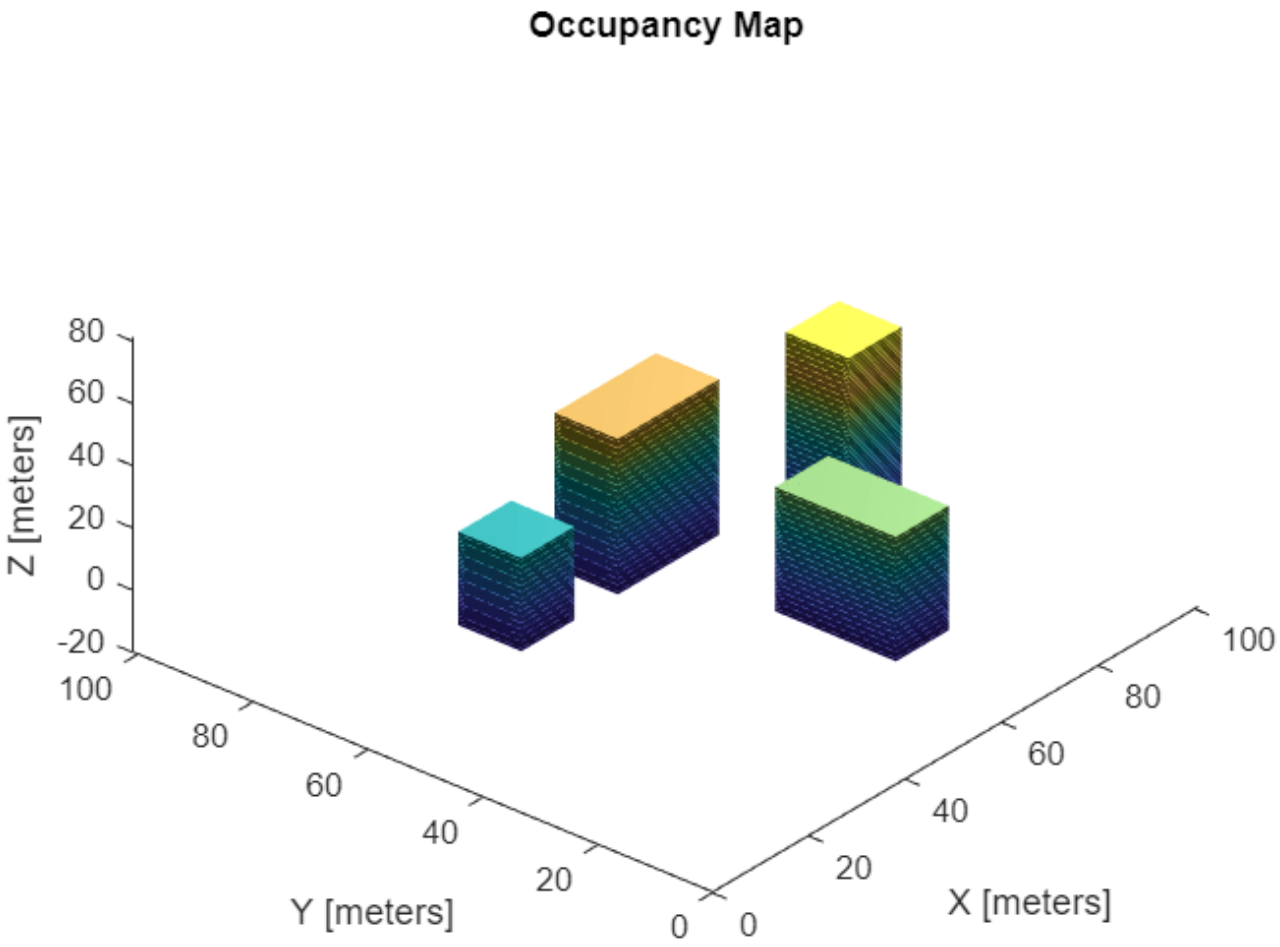
```
[xGround,yGround,zGround] = meshgrid(0:100,0:100,0);
xyzGround = [xGround(:) yGround(:) zGround(:)];
occval = 0;
setOccupancy(map3D,xyzGround,occval)
```

Create obstacles in specific world locations of the map.

```
[xBuilding1,yBuilding1,zBuilding1] = meshgrid(20:30,50:60,0:30);
[xBuilding2,yBuilding2,zBuilding2] = meshgrid(50:60,10:30,0:40);
[xBuilding3,yBuilding3,zBuilding3] = meshgrid(40:60,50:60,0:50);
[xBuilding4,yBuilding4,zBuilding4] = meshgrid(70:80,35:45,0:60);

xyzBuildings = [xBuilding1(:) yBuilding1(:) zBuilding1(:);...
                xBuilding2(:) yBuilding2(:) zBuilding2(:);...
                xBuilding3(:) yBuilding3(:) zBuilding3(:);...
                xBuilding4(:) yBuilding4(:) zBuilding4(:)];
```

Update the obstacles with new probability values and display the map.

```
obs = 0.65;
updateOccupancy(map3D,xyzBuildings,obs)
show(map3D)
```



Check if the map file named `citymap.ot` already exist in the current directory and delete it before creating the map file.

```
if exist("citymap.ot",'file')
    delete("citymap.ot")
end
```

Export the map as an octree file.

```
filePath = fullfile(pwd,"citymap.ot");
exportOccupancyMap3D(map3D,filePath)
```

Input Arguments

`map3D` — 3-D occupancy map
`occupancyMap3D` object

3-D occupancy map, specified as an `occupancyMap3D` object.

`xyz` — World coordinates
n-by-3 matrix

World coordinates, specified as an *n*-by-3 matrix of `[x y z]` points, where *n* is the number of world coordinates.

`occval` — Probability occupancy values
scalar | column vector

Probability occupancy values, specified as a scalar or a column vector with the same length as `xyz`. A scalar input is applied to all coordinates in `xyz`.

Values close to 0 represent certainty that the cell is not occupied and obstacle-free.

See Also

Classes
`occupancyMap3D` | `lidarSLAM` | `occupancyMap`

Functions
`insertPointCloud` | `inflate` | `setOccupancy` | `show`

Introduced in R2019b

show

Show occupancy map

Syntax

```
axes = show(map3D)
show(map3D,"Parent",parent)
```

Description

`axes = show(map3D)` displays the occupancy map, `map3D`, in the current axes, with the axes labels representing the world coordinates.

The function displays the 3-D environment using 3-D voxels for areas with occupancy values greater than the `OccupiedThreshold` property value specified in `map3D`. The color of the 3-D plot is strictly height-based.

`show(map3D,"Parent",parent)` displays the occupancy map in the axes handle specified by `parent`.

Examples

Get Ray Intersection Points on 3-D Occupancy Map

Import a 3-D occupancy map.

```
map3D = importOccupancyMap3D("citymap.ot")
```

```
map3D =
  occupancyMap3D with properties:

    ProbabilitySaturation: [1.0000e-03 0.9990]
               Resolution: 1
        OccupiedThreshold: 0.6500
            FreeThreshold: 0.2000
```

Inflate the occupied areas by a radius of 1 m. Display the map.

```
inflate(map3D,1)
show(map3D)
```

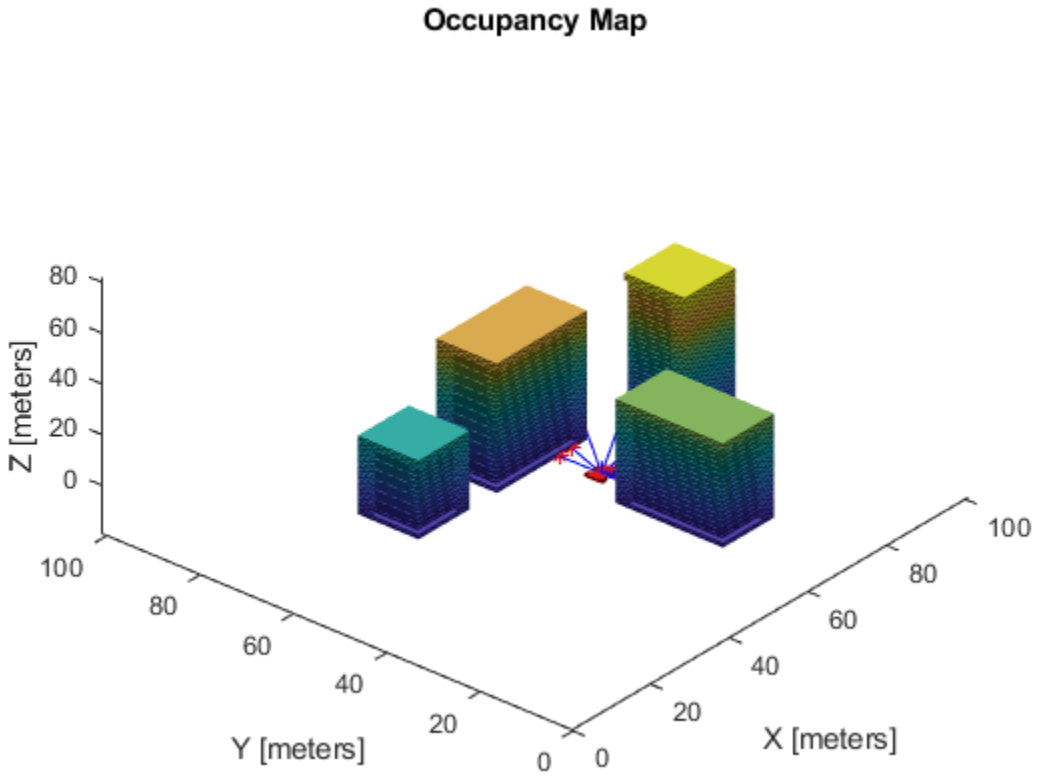
Find the intersection points of rays and occupied map cells.

```
numRays = 10;
angles = linspace(-pi/2,pi/2,numRays);
directions = [cos(angles); sin(angles); zeros(1,numRays)]';
sensorPose = [55 40 1 1 0 0 0];
maxrange = 15;
[intersectionPts,isOccupied] = rayIntersection(map3D,sensorPose,directions,maxrange)
```

```
intersectionPts = 10×3

   55.0000   32.0000    1.0000
   57.9118   32.0000    1.0000
   61.7128   32.0000    1.0000
   67.9904   32.5000    1.0000
   69.0000   37.5314    1.0000
   69.0000   42.4686    1.0000
   67.9904   47.5000    1.0000
   64.6418   51.4907    1.0000
   58.2757   49.0000    1.0000
   55.0000   49.0000    1.0000


isOccupied = 10×1

     1
     1
     1
    -1
     1
     1
    -1
    -1
     1
     1
```

Plot the intersection points and rays from the pose.

```
hold on
plotTransforms(sensorPose(1:3),sensorPose(4:end),...
               'FrameSize',5,'MeshFilePath','groundvehicle.stl') % Vehicle sensor pose
for i = 1:numRays
    plot3([sensorPose(1),intersectionPts(i,1)],...
          [sensorPose(2),intersectionPts(i,2)],...
          [sensorPose(3),intersectionPts(i,3)],'-b') % Plot rays
    if isOccupied(i) == 1
        plot3(intersectionPts(i,1),intersectionPts(i,2),intersectionPts(i,3),'*r') % Intersectio
    end
end
```



Input Arguments

`map3D` — 3-D occupancy map
`occupancyMap3D` object

3-D occupancy map, specified as an `occupancyMap3D` object.

`parent` — Axes used to plot the map
`Axes` object | `UIAxes` object

Axes used to plot the map, specified as either an `Axes` or `UIAxes` object. See `axes` or `uiaxes`.

Output Arguments

`axes` — Axes handle for map
`Axes` object | `UIAxes` object

Axes handle for map, returned as either an `Axes` or `UIAxes`object. See `axes` or `uiaxes`.

See Also

Classes
`occupancyMap3D` | `lidarSLAM` | `occupancyMap`

Functions
`insertPointCloud` | `setOccupancy` | `show`

Introduced in R2019b

updateOccupancy

Update occupancy probability at locations

Syntax

```
updateOccupancy(map3D,xyz,obs)
```

Description

`updateOccupancy(map3D,xyz,obs)` probabilistically integrates the observation values, `obs`, to each specified `xyz` coordinate in the `occupancyMap3D` object, `map3D`.

Examples

Create and Export 3-D Occupancy Map

Create an `occupancyMap3D` object.

```
map3D = occupancyMap3D;
```

Create a ground plane and set occupancy values to 0.

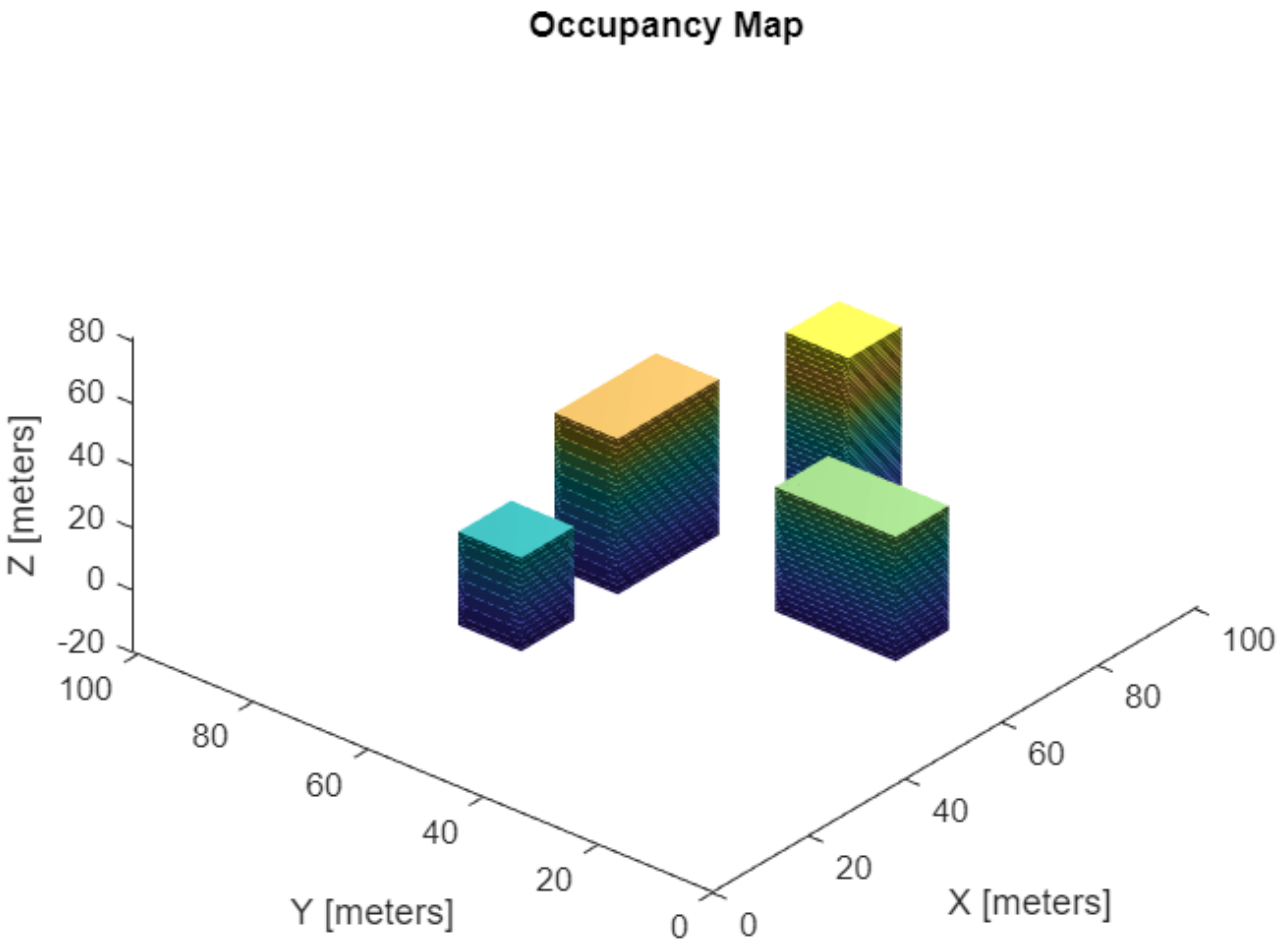
```
[xGround,yGround,zGround] = meshgrid(0:100,0:100,0);
xyzGround = [xGround(:) yGround(:) zGround(:)];
occval = 0;
setOccupancy(map3D,xyzGround,occval)
```

Create obstacles in specific world locations of the map.

```
[xBuilding1,yBuilding1,zBuilding1] = meshgrid(20:30,50:60,0:30);
[xBuilding2,yBuilding2,zBuilding2] = meshgrid(50:60,10:30,0:40);
[xBuilding3,yBuilding3,zBuilding3] = meshgrid(40:60,50:60,0:50);
[xBuilding4,yBuilding4,zBuilding4] = meshgrid(70:80,35:45,0:60);

xyzBuildings = [xBuilding1(:) yBuilding1(:) zBuilding1(:);...
                xBuilding2(:) yBuilding2(:) zBuilding2(:);...
                xBuilding3(:) yBuilding3(:) zBuilding3(:);...
                xBuilding4(:) yBuilding4(:) zBuilding4(:)];
```

Update the obstacles with new probability values and display the map.

```
obs = 0.65;
updateOccupancy(map3D,xyzBuildings,obs)
show(map3D)
```



Check if the map file named `citymap.ot` already exist in the current directory and delete it before creating the map file.

```
if exist("citymap.ot",'file')
    delete("citymap.ot")
end
```

Export the map as an octree file.

```
filePath = fullfile(pwd,"citymap.ot");
exportOccupancyMap3D(map3D,filePath)
```

Input Arguments

`map3D` — 3-D occupancy map
`occupancyMap3D` object

3-D occupancy map, specified as an `occupancyMap3D` object.

xyz — World coordinates
n-by-3 matrix

World coordinates, specified as an *n*-by-3 matrix of `[x y z]` points, where *n* is the number of world coordinates.

obs — Probability observation values
numeric scalar | logical scalar | *n*-by-1 column vector

Probability observation values, specified as a numeric or logical scalar, or as an *n*-by-1 column vector with the same size as `xyz`.

`obs` values can be from 0 to 1, but if `obs` is a logical array, the function uses the default observation values of 0.7 (`true`) and 0.4 (`false`). If `obs` is a numeric or logical scalar, the value is applied to all coordinates in `xyz`.

See Also

Classes
`occupancyMap3D` | `lidarSLAM` | `occupancyMap`

Functions
`insertPointCloud` | `inflate` | `setOccupancy` | `show`

Introduced in R2019b

odometryMotionModel

Create an odometry motion model

Description

`odometryMotionModel` creates an odometry motion model object for differential drive vehicles. This object contains specific motion model parameters. You can use this object to specify the motion model parameters in the `monteCarloLocalization` object.

This motion model assumes that the vehicle makes pure rotation and translation motions to travel from one location to the other. The model propagates points for either forward or backwards motion based on these motion patterns. The elements of the `Noise` property refer to the variance in the motion. To see the effect of changing the noise parameters, use `showNoiseDistribution`.

Creation

Syntax

```
omm = odometryMotionModel
```

Description

`omm = odometryMotionModel` creates an odometry motion model object for differential drive vehicles.

Properties

`Noise` — Gaussian noise for vehicle motion

`[0.2 0.2 0.2 0.2]` (default) | 4-element vector

Gaussian noise for vehicle motion, specified as a 4-element vector. This property represents the variance parameters for Gaussian noise applied to vehicle motion. The elements of the vector correspond to the following errors in order:

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

Type — Type of the odometry motion model

`'DifferentialDrive'` (default)

This property is read-only.

Type of the odometry motion model, returned as `'DifferentialDrive'`. This read-only property indicates the type of odometry motion model being used by the object.

Object Functions

showNoiseDistribution Display noise parameter effects

Examples

Predict Poses Based On An Odometry Motion Model

This example shows how to use the `odometryMotionModel` class to predict the pose of a vehicle. An `odometryMotionModel` object contains the motion model parameters for a differential drive vehicle. Use the object to predict the pose of a vehicle based on its current and previous poses and the motion model parameters.

Create odometry motion model object.

```
motionModel = odometryMotionModel;
```

Define previous poses and the current odometry reading. Each pose prediction corresponds to a row in `previousPoses` vector.

```
previousPoses =  rand(10,3);
currentOdom = [0.1 0.1 0.1];
```

The first call to the object initializes values and returns the previous poses as the current poses.

```
currentPoses = motionModel(previousPoses, currentOdom);
```

Subsequent calls to the object with updated odometry poses returns the predicted poses based on the motion model.

```
currentOdom = currentOdom + [0.1 0.1 0.05];
predPoses = motionModel(previousPoses, currentOdom);
```

Show Noise Distribution Effects for Odometry Motion Model

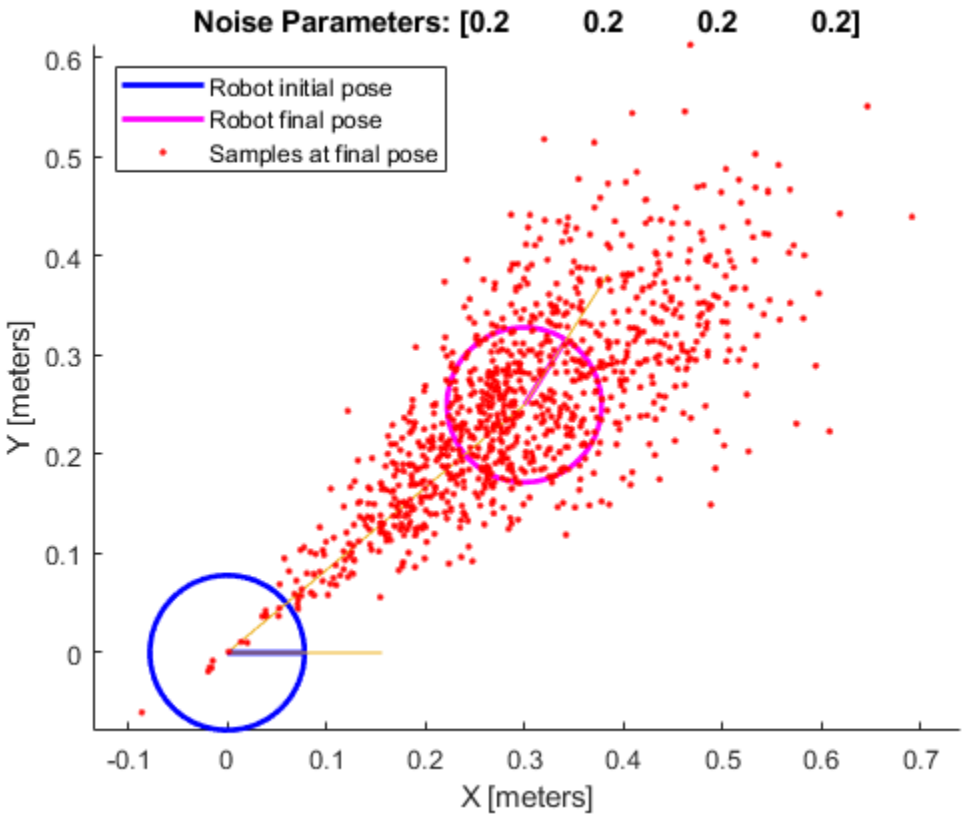
This example shows how to visualize the effect of different noise parameters on the `odometryMotionModel` class. An `odometryMotionModel` object contains the motion model noise parameters for a differential drive vehicle. Use `showNoiseDistribution` to visualize how changing these values affect the distribution of predicted poses.

Create a motion model object.

```
motionModel = odometryMotionModel;
```

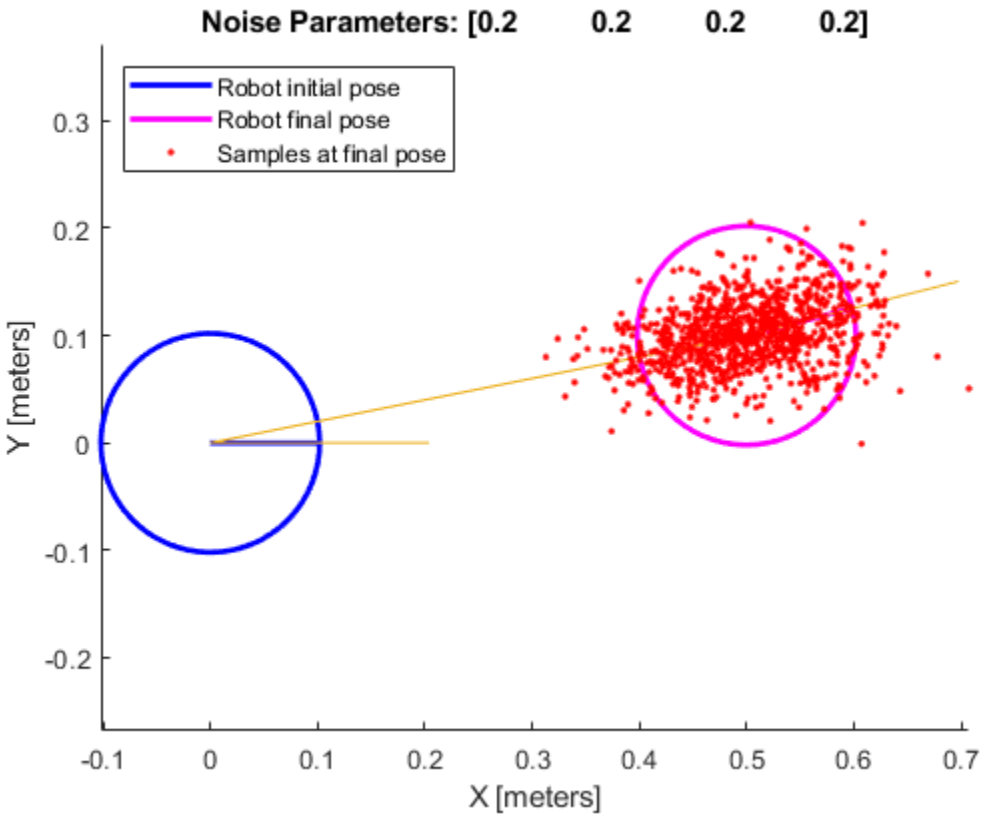
Show the distribution of particles with the existing noise parameters. Each particle is a hypothesis for the predicted pose.

```
showNoiseDistribution(motionModel);
```



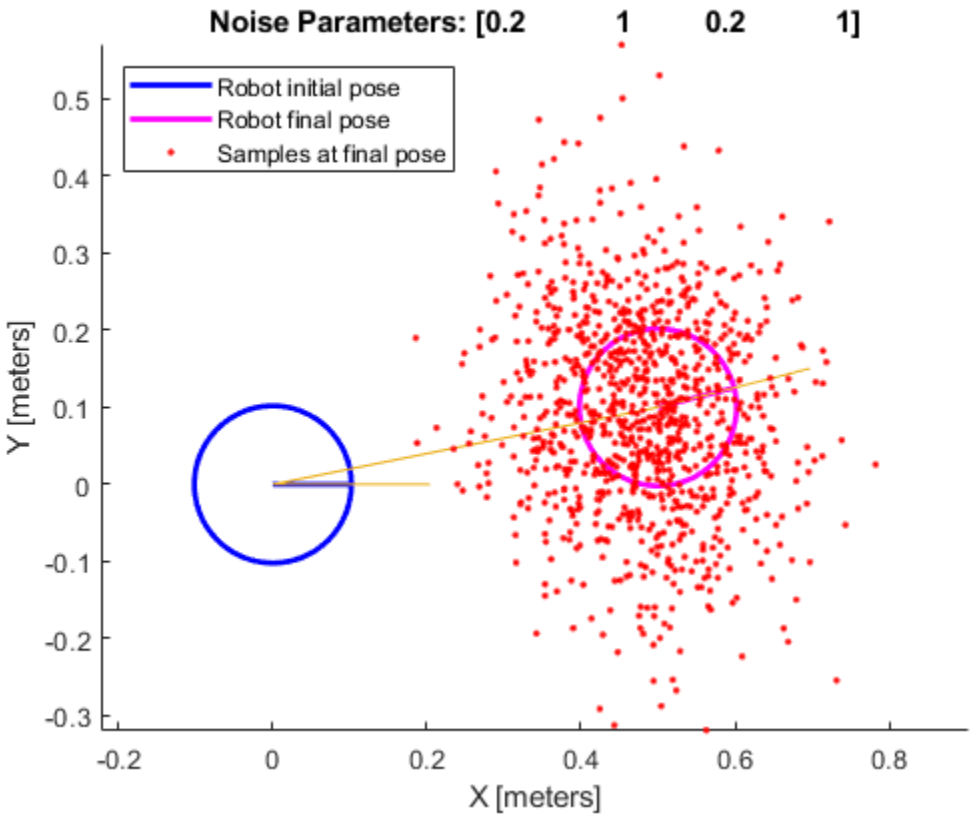
Show the distribution with a specified odometry pose change and number of samples. The change in odometry is used as the final pose with hypotheses distributed around based on the `Noise` parameters.

```
showNoiseDistribution(motionModel, ...
            'OdometryPoseChange', [0.5 0.1 0.25], ...
            'NumSamples', 1000);
```



Change the `Noise` parameters and visualize the effects. Use the same odometry pose change and number of samples.

```
motionModel.Noise = [0.2 1 0.2 1];

showNoiseDistribution(motionModel, ...
           'OdometryPoseChange', [0.5 0.1 0.25], ...
           'NumSamples', 1000);
```



Limitations

If you make changes to your motion model after using it with the `monteCarloLocalization` object, call `release` on that object beforehand. For example:

```
mcl = monteCarloLocalization;
[isUpdated,pose,covariance] = mcl(ranges,angles);
release(mcl)
mcl.MotionModel.Noise = [0.25 0.25 0.4 0.4];
```

References

[1] Thrun, Sebatian, Wolfram Burgard, and Dieter Fox. *Probabilistic Robotics*. MIT Press, 2005.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`monteCarloLocalization` | `likelihoodFieldSensorModel`

Topics
“Localize TurtleBot Using Monte Carlo Localization”

Introduced in R2019b

showNoiseDistribution

Display noise parameter effects

Syntax

```
showNoiseDistribution(ommObj)
showNoiseDistribution(ommObj)
showNoiseDistribution(ommObj,Name,Value)
```

Description

`showNoiseDistribution(ommObj)` shows the noise distribution for a default odometry pose update, number of samples and the current noise parameters on the input object.

`axes = showNoiseDistribution(ommObj)` shows the noise distribution and returns the axes handle.

`showNoiseDistribution(ommObj,Name,Value)` provides additional options specified by one or more `Name,Value` pairs. Name is the property name and Value is the corresponding value. Name must appear inside single quotes (`' '`). You can specify several name-value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`. Properties not specified retain their default values.

Examples

Show Noise Distribution Effects for Odometry Motion Model

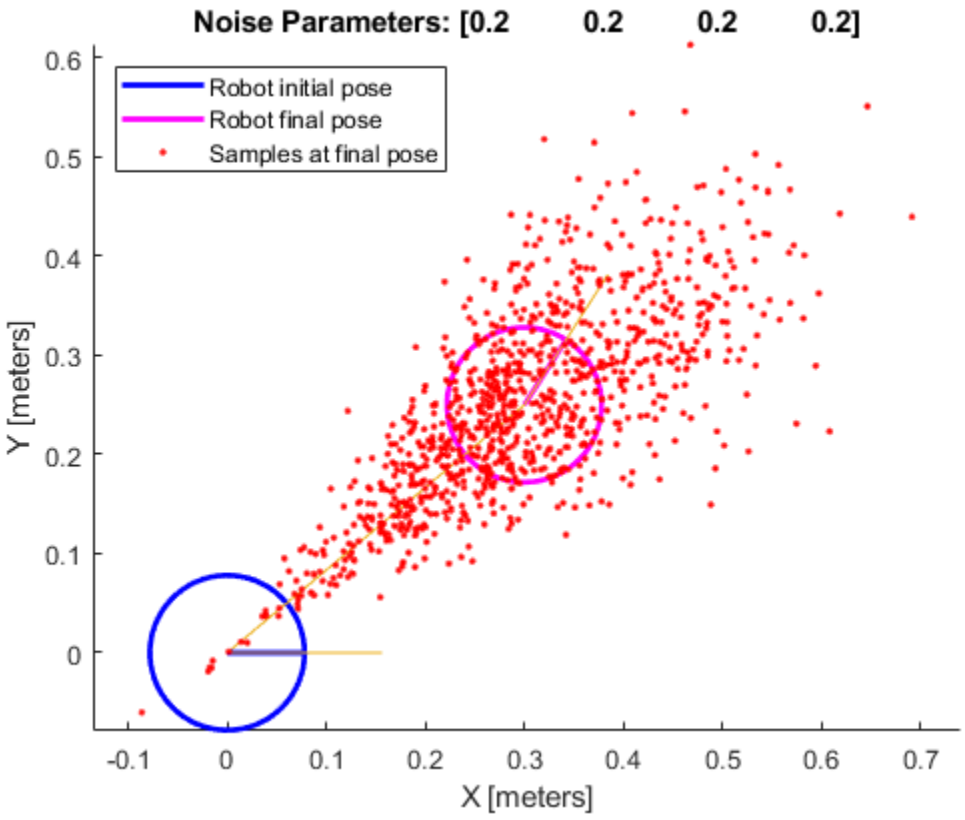
This example shows how to visualize the effect of different noise parameters on the `odometryMotionModel` class. An `odometryMotionModel` object contains the motion model noise parameters for a differential drive vehicle. Use `showNoiseDistribution` to visualize how changing these values affect the distribution of predicted poses.

Create a motion model object.

```
motionModel = odometryMotionModel;
```

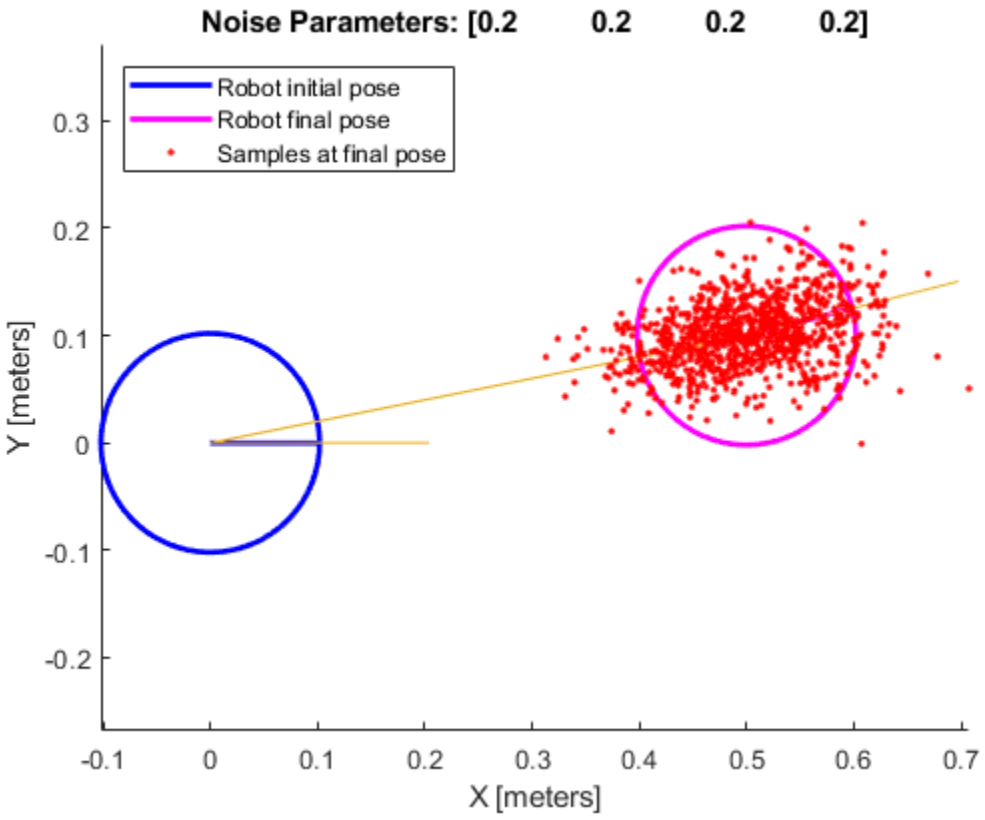
Show the distribution of particles with the existing noise parameters. Each particle is a hypothesis for the predicted pose.

```
showNoiseDistribution(motionModel);
```



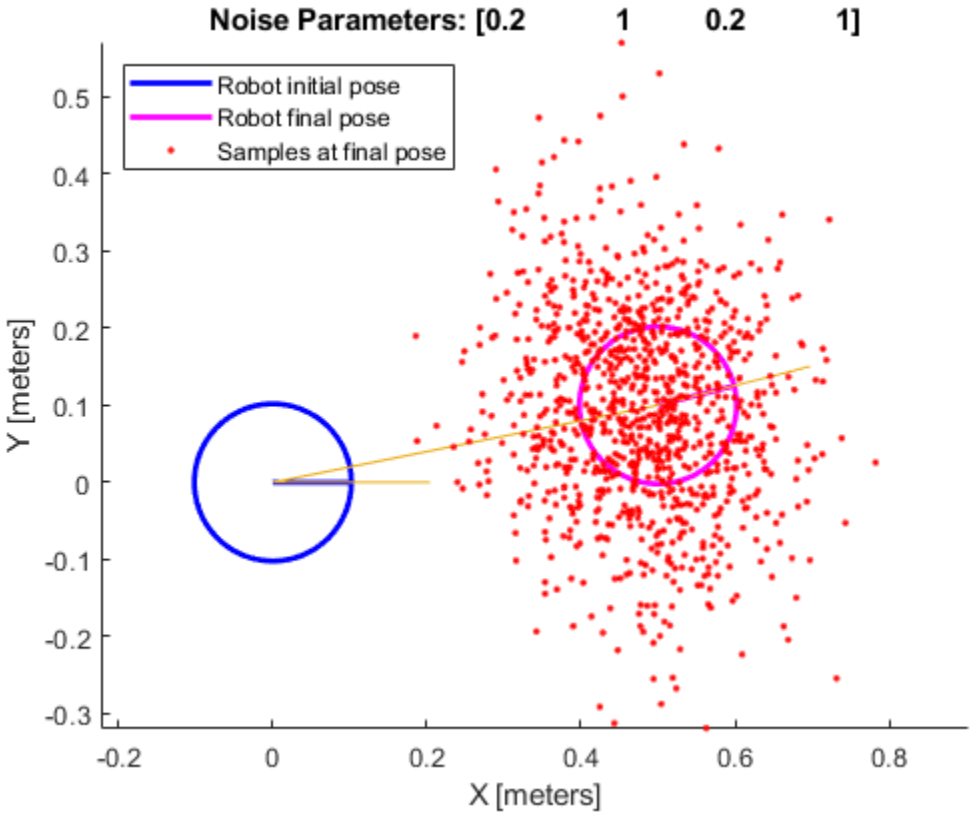
Show the distribution with a specified odometry pose change and number of samples. The change in odometry is used as the final pose with hypotheses distributed around based on the `Noise` parameters.

```
showNoiseDistribution(motionModel, ...
            'OdometryPoseChange', [0.5 0.1 0.25], ...
            'NumSamples', 1000);
```



Change the `Noise` parameters and visualize the effects. Use the same odometry pose change and number of samples.

```
motionModel.Noise = [0.2 1 0.2 1];

showNoiseDistribution(motionModel, ...
           'OdometryPoseChange', [0.5 0.1 0.25], ...
           'NumSamples', 1000);
```



Input Arguments

`ommObj` — `odometryMotionModel` object

handle

`odometryMotionModel` object, specified as a handle. Create this object using `odometryMotionModel`.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'OdometryPoseChange',[1 1 pi]`

`OdometryPoseChange` — Change in odometry

three-element vector

Change in odometry of the robot, specified as a comma-separated pair consisting of `'OdometryPoseChange'` and a three-element vector, `[x y theta]`.

`NumSamples` — Number of particles to display

scalar

Number of particles to display, specified as a comma-separated pair consisting of `'NumSamples'` and a scalar.

`Parent` — Axes to plot the map
`Axes` object | `UIAxes` object

Axes to plot the map, specified as a comma-separated pair consisting of `'Parent'` and either an `Axes` or `UIAxes` object. See `axes` or `uiaxes`.

See Also
`monteCarloLocalization` | `odometryMotionModel` | `likelihoodFieldSensorModel`

Introduced in R2019b

pathmetrics

Information for path metrics

Description

The `pathmetrics` object holds information for computing path metrics. Use object functions to calculate smoothness, clearance, and path validity based on a set of poses and the associated map environment.

Creation

Syntax

```
pathMetricsObj = pathmetrics(path)
pathMetricsObj = pathmetrics(path,validator)
```

Description

`pathMetricsObj = pathmetrics(path)` creates an object based on the input `navPath` object. The state validator is assumed to be a `validatorOccupancyMap` object. The `path` input sets the value of the "Path" on page 2-0 property.

`pathMetricsObj = pathmetrics(path,validator)` creates an object based on the input `navPath` object and associated state validator for checking the path validity. The `validator` input sets the value of the "StateValidator" on page 2-0 property.

Properties

`Path` — Path data structure
`navPath` object

Path data structure, specified as a `navPath` object is the path whose metric is to be calculated.

`StateValidator` — Validator for states on path
`validatorOccupancyMap(stateSpaceSE2,binaryOccupancyMap(10))` (default) | `validatorOccupancyMap` object | `validatorVehicleCostmap` object

Validator for states on path, specified either as a `validatorOccupancyMap` or `validatorVehicleCostmap` object.

Object Functions

clearance	Minimum clearance of path
isPathValid	Determine if planned path is obstacle free
show	Visualize path metrics in map environment
smoothness	Smoothness of path

Examples

Compute Path Metrics

Compute smoothness, clearance, and validity of a planned path based on a set of poses and the associated map environment.

Load and Assign Map to State Validator

Create an occupancy map from an example map and set the map resolution.

```
load exampleMaps.mat; % simpleMap
mapResolution = 1; % cells/meter
map = occupancyMap(simpleMap,mapResolution);
```

Create a Dubins state space.

```
statespace = stateSpaceDubins;
```

Create a state validator based on occupancy map to store the parameters and states in the Dubins state space.

```
statevalidator = validatorOccupancyMap(statespace);
```

Assign the map to the validator.

```
statevalidator.Map = map;
```

Set the validation distance for the validator.

```
statevalidator.ValidationDistance = 0.01;
```

Update the state space bounds to be the same as the map limits.

```
statespace.StateBounds = [map.XWorldLimits;map.YWorldLimits;[-pi pi]];
```

Plan Path

Create an RRT* path planner and allow further optimization.

```
planner = plannerRRTStar(statespace,statevalidator);
planner.ContinueAfterGoalReached = true;
```

Reduce the maximum number of iterations and increase the maximum connection distance.

```
planner.MaxIterations = 2500;
planner.MaxConnectionDistance = 0.3;
```

Define start and goal states for the path planner as `[`*x, y, theta*`]` vectors. *x* and *y* are the Cartesian coordinates, and *theta* is the orientation angle.

```
start = [2.5, 2.5, 0]; % [meters, meters, radians]
goal = [22.5, 8.75, 0];
```

Plan a path from the start state to the goal state. The plan function returns a `navPath` object.

```
rng(100,'twister') % repeatable result
[path,solutionInfo] = plan(planner,start,goal);
```

Compute and Visualize Path Metrics

Create a path metrics object.

```
pathMetricsObj = pathmetrics(path,statevalidator);
```

Check path validity. The result is `1 (true)` if the planned path is obstacle free. `0 (false)` indicates an invalid path.

```
isPathValid(pathMetricsObj)
```

```
ans = logical
   1
```

Calculate the minimum clearance of the path.

```
clearance(pathMetricsObj)
```

```
ans = 1.4142
```

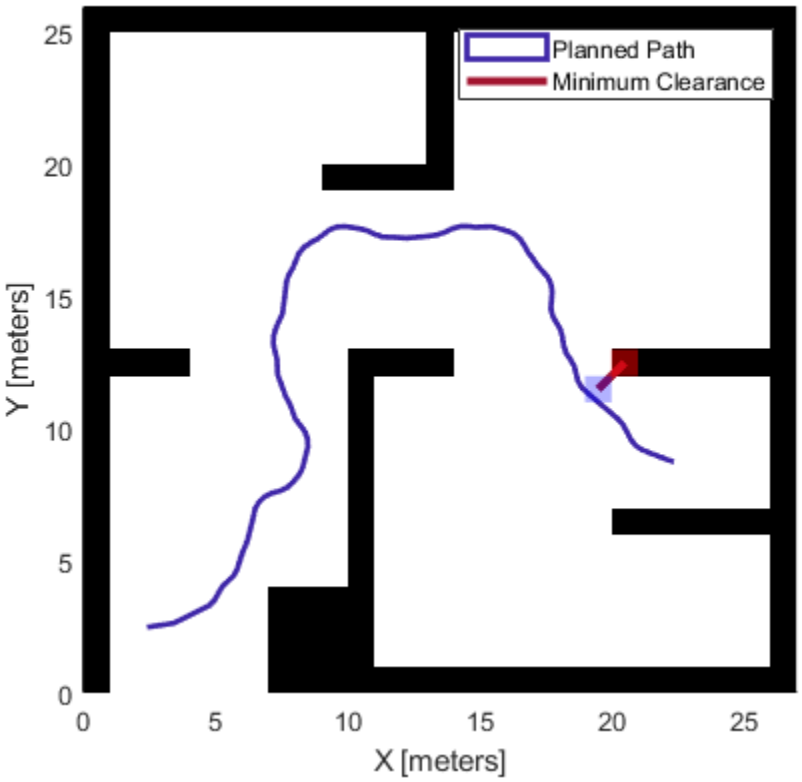
Evaluate the smoothness of the path. Values close to `0` indicate a smoother path. Straight-line paths return a value of `0`.

```
smoothness(pathMetricsObj)
```

```
ans = 1.7318
```

Visualize the minimum clearance of the path.

```
show(pathMetricsObj)
legend('Planned Path','Minimum Clearance')
```



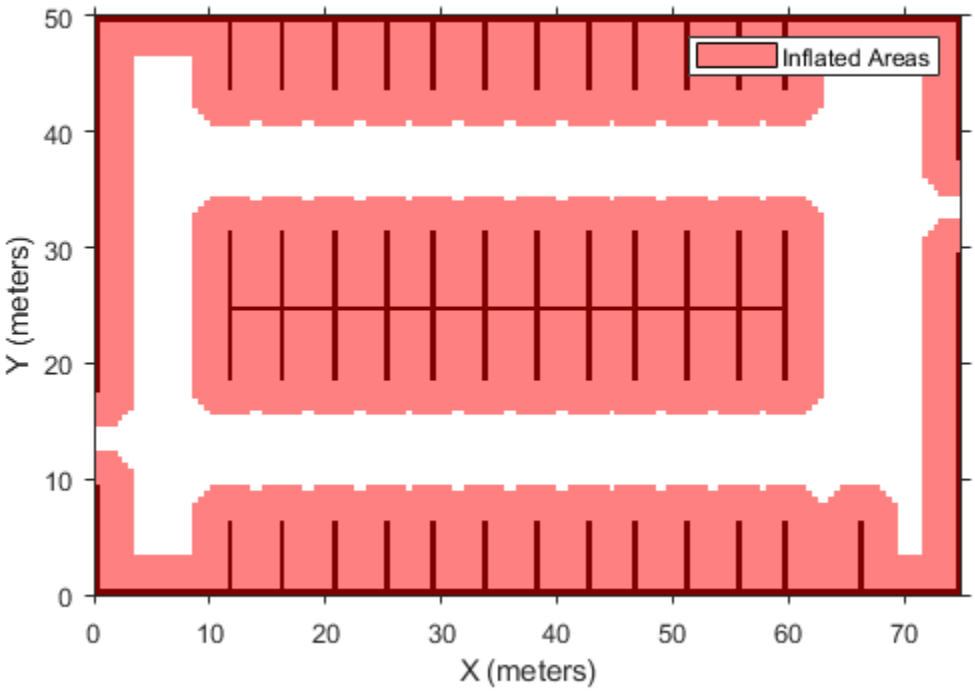
Vehicle Path Planning and Metrics Computation in a 2-D Costmap Environment

Plan a vehicle path through a parking lot using the RRT* algorithm. Compute and visualize the smoothness, clearance, and validity of the planned path.

Load and Assign Map to State Validator

Load a costmap of a parking lot. Plot the costmap to see the parking lot and the inflated areas that the vehicle should avoid.

```
load parkingLotCostmap.mat;
costmap = parkingLotCostmap;
plot(costmap)
xlabel('X (meters)')
ylabel('Y (meters)')
```



Create a `stateSpaceDubins` object and increase the minimum turing radius to 4 meters.

```
statespace = stateSpaceDubins;
statespace.MinTurningRadius = 4; % meters
```

Create a `validatorVehicleCostmap` object using the created state space.

```
statevalidator = validatorVehicleCostmap(statespace);
```

Assign the parking lot costmap to the state validator object.

```
statevalidator.Map = costmap;
```

Plan Path

Define start and goal poses for the vehicle as [*x*, *y*, `0`] vectors. World units for the (*x*,*y*) locations are in meters. World units for the orientation angles `0` are in degrees.

```
startPose = [5, 5, 90]; % [meters, meters, degrees]
goalPose = [40, 38, 180]; % [meters, meters, degrees]
```

Use a `pathPlannerRRT` (Automated Driving Toolbox) object and the `plan` (Automated Driving Toolbox) function to plan the vehicle path from the start pose to the goal pose.

```
planner = pathPlannerRRT(costmap);
refPath = plan(planner,startPose,goalPose);
```

Interpolate along the path at every one meter. Convert the orientation angles from degrees to radians.

```
poses = zeros(size(refPath.PathSegments,2)+1,3);
poses(1,:) = refPath.StartPose;
for i = 1:size(refPath.PathSegments,2)
    poses(i+1,:) = refPath.PathSegments(i).GoalPose;
end
poses(:,3) = deg2rad(poses(:,3));
```

Create a `navPath` object using the Dubins state space object and the states specified by `poses`.

```
path = navPath(statespace,poses);
```

Compute and Visualize Path Metrics

Create a `pathmetrics` object.

```
pathMetricsObj = pathmetrics(path,statevalidator);
```

Check path validity. The result is `1` (`true`) if the planned path is obstacle free. `0` (`false`) indicates an invalid path.

```
isPathValid(pathMetricsObj)
```

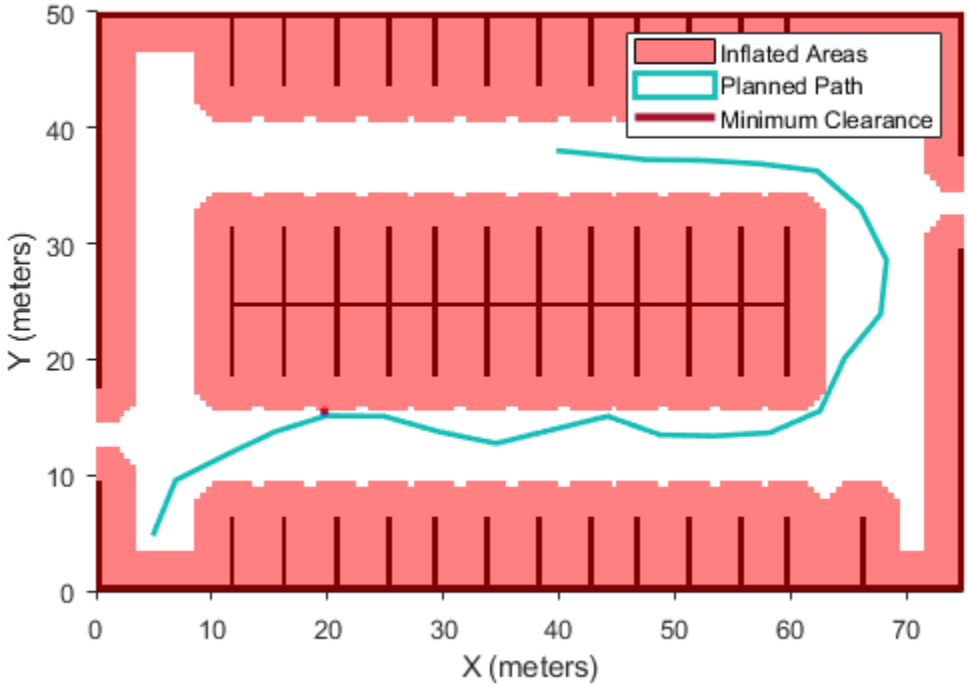
```
ans = logical
   1
```

Compute and visualize the minimum `clearance` of the path.

```
clearance(pathMetricsObj)
```

```
ans = 0.5000
```

```
show(pathMetricsObj)
legend('Inflated Areas','Planned Path','Minimum Clearance')
xlabel('X (meters)')
ylabel('Y (meters)')
```

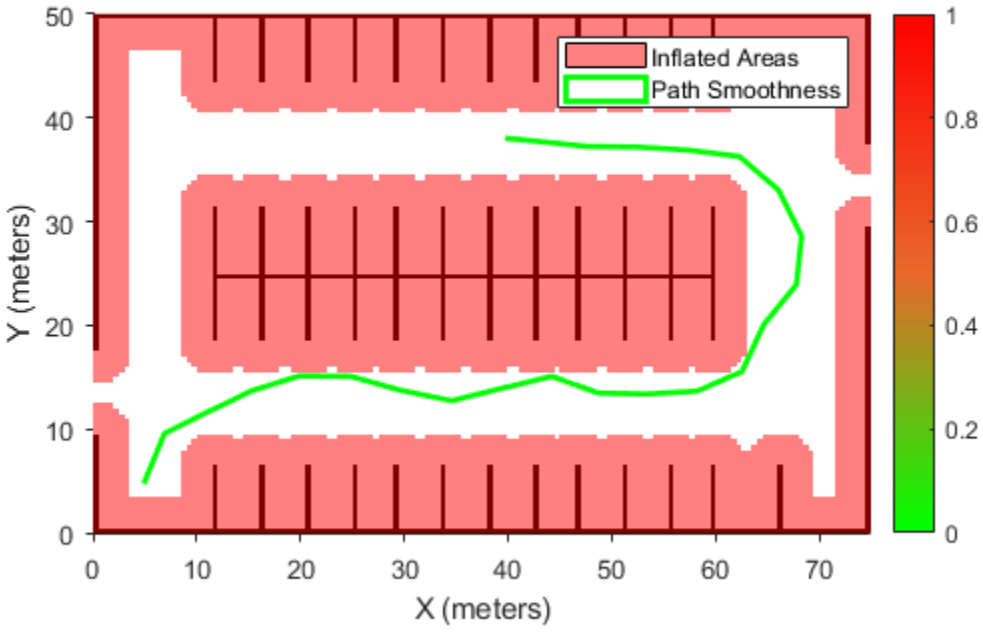


Compute and visualize the `smoothness` of the path. Values close to `0` indicate a smoother path. Straight-line paths return a value of `0`.

```
smoothness(pathMetricsObj)
```

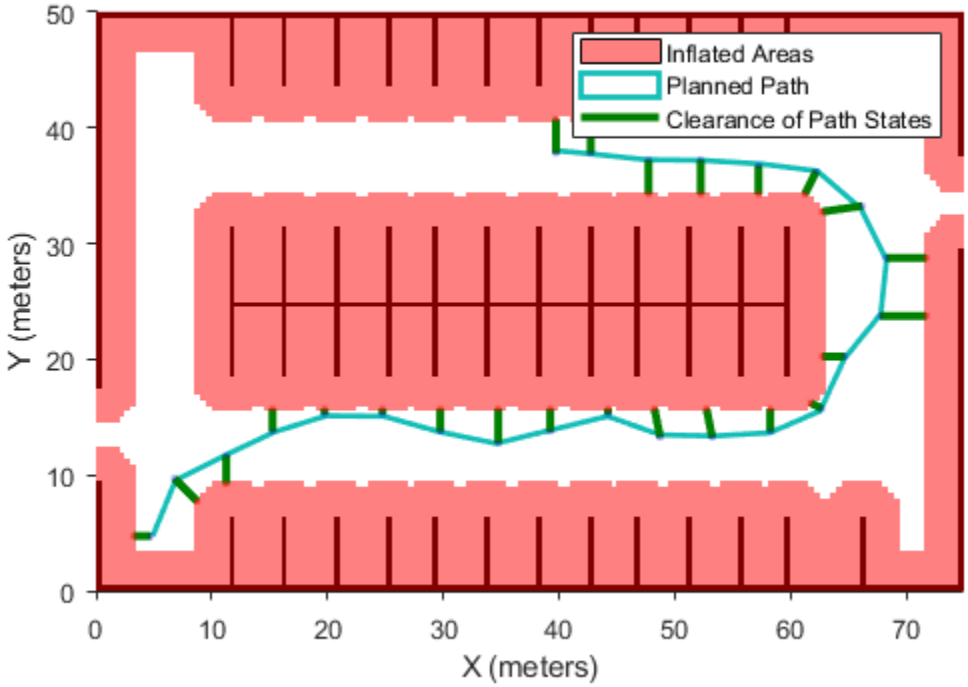
```
ans = 0.0842
```

```
show(pathMetricsObj,'Metrics',{'Smoothness'})
legend('Inflated Areas','Path Smoothness')
xlabel('X (meters)')
ylabel('Y (meters)')
```



Visualize the clearance for each state of the path.

```
show(pathMetricsObj,'Metrics',{'StatesClearance'})
legend('Inflated Areas','Planned Path','Clearance of Path States')
xlabel('X (meters)')
ylabel('Y (meters)')
```



See Also

`occupancyMap` | `occupancyMap3D` | `plannerRRTStar`

Introduced in R2019b

clearance

Minimum clearance of path

Syntax

```
clearance(pathMetricsObj)
clearance(pathMetricsObj,'Type','states')
```

Description

`clearance(pathMetricsObj)` returns the minimum clearance of the path. Clearance is measured as the minimum distance between grid cell centers of states on the path and obstacles in the specified map environment.

Note The computed clearance is accurate up to `sqrt(2)` times grid map cell size.

`clearance(pathMetricsObj,'Type','states')` returns the set of minimum distances for each state of the path, in the form of an *n*-by-1 vector, where *n* is the number of states.

Examples

Compute Path Metrics

Compute smoothness, clearance, and validity of a planned path based on a set of poses and the associated map environment.

Load and Assign Map to State Validator

Create an occupancy map from an example map and set the map resolution.

```
load exampleMaps.mat; % simpleMap
mapResolution = 1; % cells/meter
map = occupancyMap(simpleMap,mapResolution);
```

Create a Dubins state space.

```
statespace = stateSpaceDubins;
```

Create a state validator based on occupancy map to store the parameters and states in the Dubins state space.

```
statevalidator = validatorOccupancyMap(statespace);
```

Assign the map to the validator.

```
statevalidator.Map = map;
```

Set the validation distance for the validator.

```
statevalidator.ValidationDistance = 0.01;
```

Update the state space bounds to be the same as the map limits.

```
statespace.StateBounds = [map.XWorldLimits;map.YWorldLimits;[-pi pi]];
```

Plan Path

Create an RRT* path planner and allow further optimization.

```
planner = plannerRRTStar(statespace,statevalidator);
planner.ContinueAfterGoalReached = true;
```

Reduce the maximum number of iterations and increase the maximum connection distance.

```
planner.MaxIterations = 2500;
planner.MaxConnectionDistance = 0.3;
```

Define start and goal states for the path planner as `[`*x, y, theta*`]` vectors. *x* and *y* are the Cartesian coordinates, and *theta* is the orientation angle.

```
start = [2.5, 2.5, 0]; % [meters, meters, radians]
goal = [22.5, 8.75, 0];
```

Plan a path from the start state to the goal state. The plan function returns a `navPath` object.

```
rng(100,'twister') % repeatable result
[path,solutionInfo] = plan(planner,start,goal);
```

Compute and Visualize Path Metrics

Create a path metrics object.

```
pathMetricsObj = pathmetrics(path,statevalidator);
```

Check path validity. The result is `1 (true)` if the planned path is obstacle free. `0 (false)` indicates an invalid path.

```
isPathValid(pathMetricsObj)
```

```
ans = logical
   1
```

Calculate the minimum clearance of the path.

```
clearance(pathMetricsObj)
```

```
ans = 1.4142
```

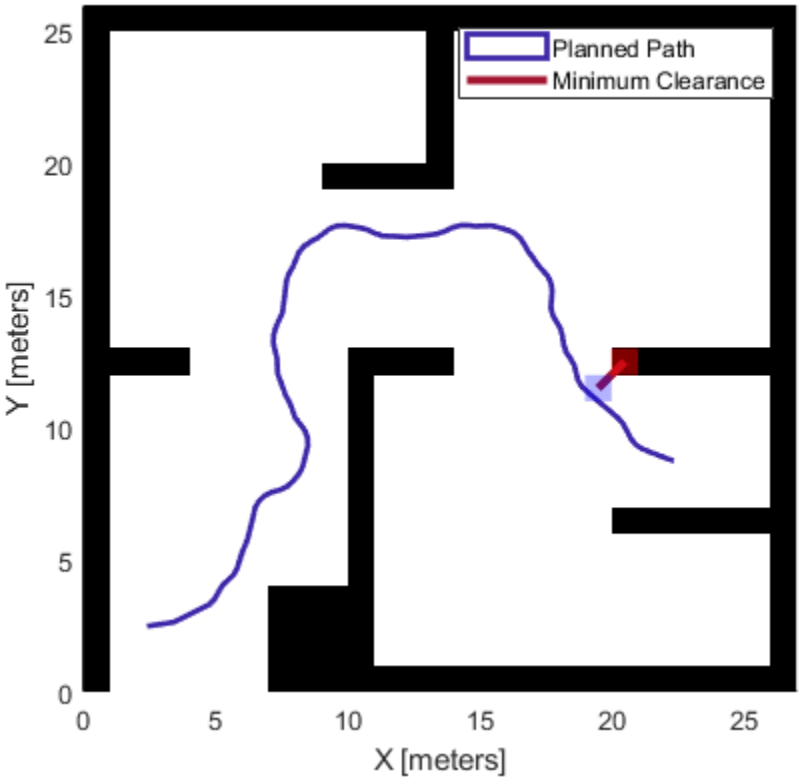
Evaluate the smoothness of the path. Values close to `0` indicate a smoother path. Straight-line paths return a value of `0`.

```
smoothness(pathMetricsObj)
```

```
ans = 1.7318
```

Visualize the minimum clearance of the path.

```
show(pathMetricsObj)
legend('Planned Path','Minimum Clearance')
```



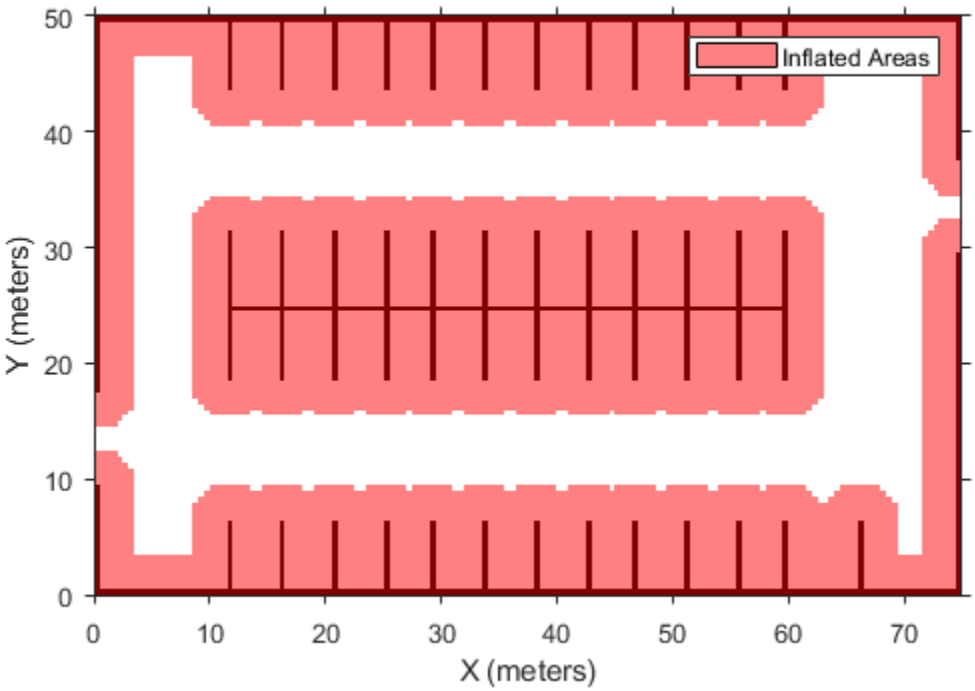
Vehicle Path Planning and Metrics Computation in a 2-D Costmap Environment

Plan a vehicle path through a parking lot using the RRT* algorithm. Compute and visualize the smoothness, clearance, and validity of the planned path.

Load and Assign Map to State Validator

Load a costmap of a parking lot. Plot the costmap to see the parking lot and the inflated areas that the vehicle should avoid.

```
load parkingLotCostmap.mat;
costmap = parkingLotCostmap;
plot(costmap)
xlabel('X (meters)')
ylabel('Y (meters)')
```



Create a `stateSpaceDubins` object and increase the minimum turing radius to 4 meters.

```
statespace = stateSpaceDubins;
statespace.MinTurningRadius = 4; % meters
```

Create a `validatorVehicleCostmap` object using the created state space.

```
statevalidator = validatorVehicleCostmap(statespace);
```

Assign the parking lot costmap to the state validator object.

```
statevalidator.Map = costmap;
```

Plan Path

Define start and goal poses for the vehicle as [*x*, *y*, `θ`] vectors. World units for the (*x*,*y*) locations are in meters. World units for the orientation angles `θ` are in degrees.

```
startPose = [5, 5, 90]; % [meters, meters, degrees]
goalPose = [40, 38, 180]; % [meters, meters, degrees]
```

Use a `pathPlannerRRT` (Automated Driving Toolbox) object and the `plan` (Automated Driving Toolbox) function to plan the vehicle path from the start pose to the goal pose.

```
planner = pathPlannerRRT(costmap);
refPath = plan(planner,startPose,goalPose);
```

Interpolate along the path at every one meter. Convert the orientation angles from degrees to radians.

```
poses = zeros(size(refPath.PathSegments,2)+1,3);
poses(1,:) = refPath.StartPose;
for i = 1:size(refPath.PathSegments,2)
    poses(i+1,:) = refPath.PathSegments(i).GoalPose;
end
poses(:,3) = deg2rad(poses(:,3));
```

Create a `navPath` object using the Dubins state space object and the states specified by `poses`.

```
path = navPath(statespace,poses);
```

Compute and Visualize Path Metrics

Create a `pathmetrics` object.

```
pathMetricsObj = pathmetrics(path,statevalidator);
```

Check path validity. The result is `1` (`true`) if the planned path is obstacle free. `0` (`false`) indicates an invalid path.

```
isPathValid(pathMetricsObj)
```

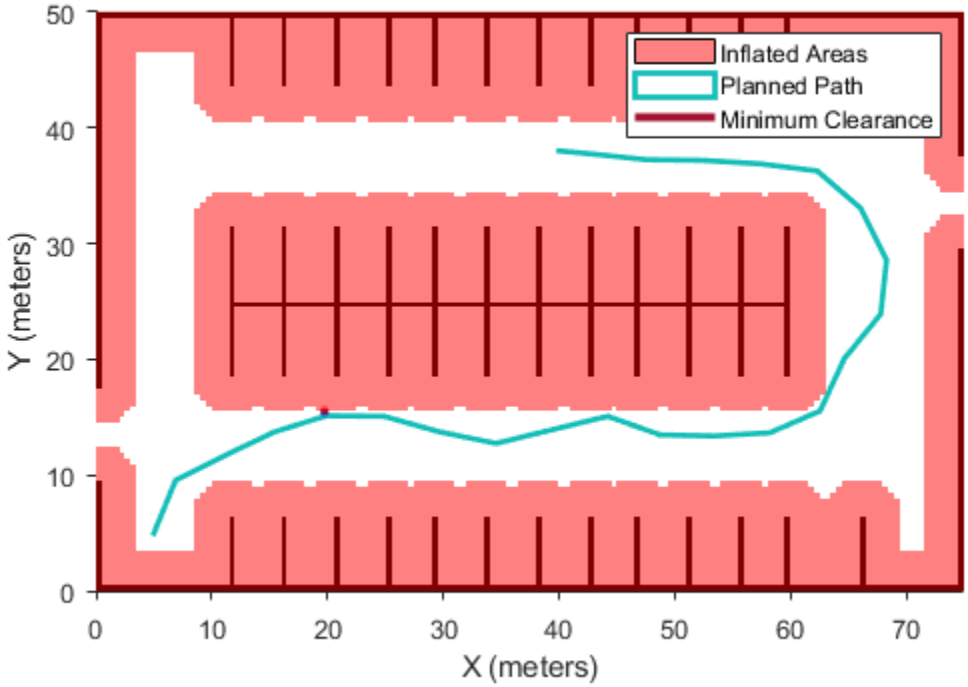
```
ans = logical
   1
```

Compute and visualize the minimum `clearance` of the path.

```
clearance(pathMetricsObj)
```

```
ans = 0.5000
```

```
show(pathMetricsObj)
legend('Inflated Areas','Planned Path','Minimum Clearance')
xlabel('X (meters)')
ylabel('Y (meters)')
```

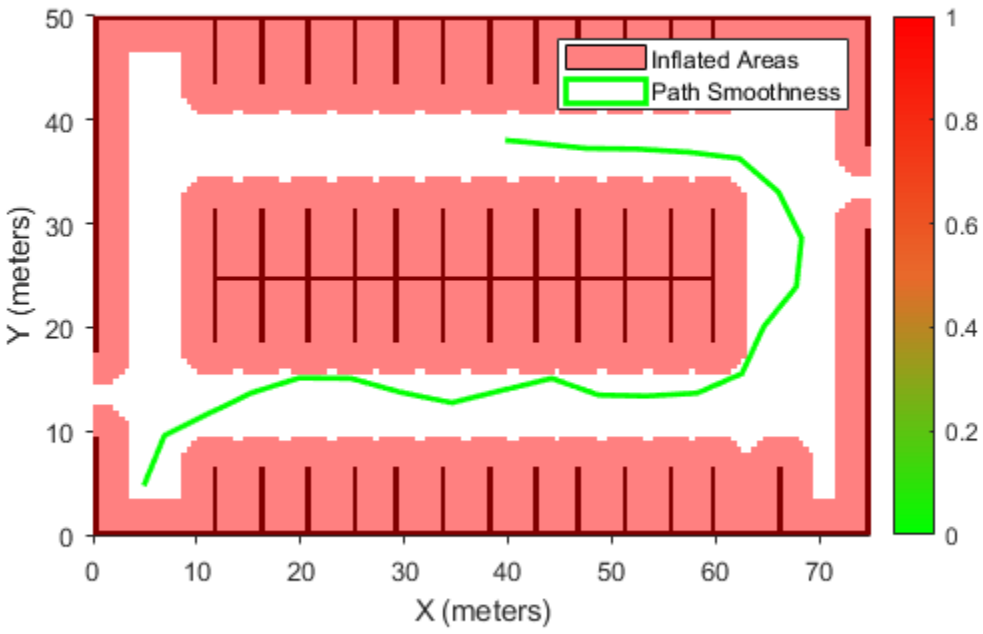


Compute and visualize the `smoothness` of the path. Values close to `0` indicate a smoother path. Straight-line paths return a value of `0`.

```
smoothness(pathMetricsObj)
```

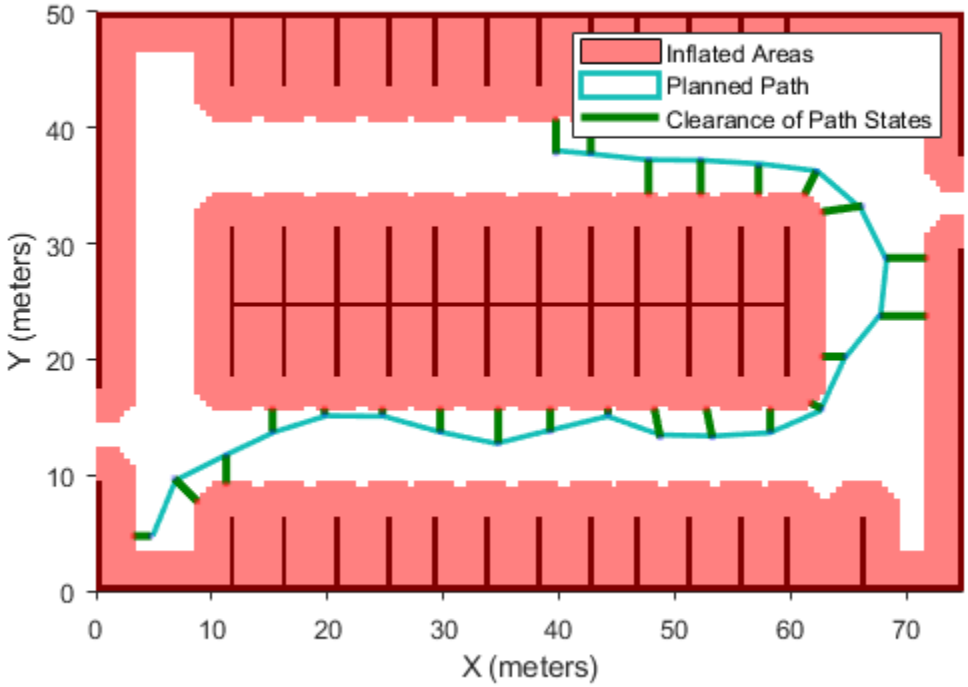
```
ans = 0.0842
```

```
show(pathMetricsObj,'Metrics',{'Smoothness'})
legend('Inflated Areas','Path Smoothness')
xlabel('X (meters)')
ylabel('Y (meters)')
```



Visualize the clearance for each state of the path.

```
show(pathMetricsObj,'Metrics',{'StatesClearance'})
legend('Inflated Areas','Planned Path','Clearance of Path States')
xlabel('X (meters)')
ylabel('Y (meters)')
```



Input Arguments

`pathMetricsObj` — Information for path metrics

`pathmetrics` object

Information for path metrics, specified as a `pathmetrics` object.

See Also

Objects

`pathmetrics`

Functions

`isPathValid` | `show` | `smoothness`

Introduced in R2019b

isPathValid

Determine if planned path is obstacle free

Syntax

```
isPathValid(pathMetricsObj)
```

Description

`isPathValid(pathMetricsObj)` returns either a logical `1 (true)` if the planned path is obstacle free or a logical `0 (false)` if the path is invalid.

Examples

Compute Path Metrics

Compute smoothness, clearance, and validity of a planned path based on a set of poses and the associated map environment.

Load and Assign Map to State Validator

Create an occupancy map from an example map and set the map resolution.

```
load exampleMaps.mat; % simpleMap
mapResolution = 1; % cells/meter
map = occupancyMap(simpleMap,mapResolution);
```

Create a Dubins state space.

```
statespace = stateSpaceDubins;
```

Create a state validator based on occupancy map to store the parameters and states in the Dubins state space.

```
statevalidator = validatorOccupancyMap(statespace);
```

Assign the map to the validator.

```
statevalidator.Map = map;
```

Set the validation distance for the validator.

```
statevalidator.ValidationDistance = 0.01;
```

Update the state space bounds to be the same as the map limits.

```
statespace.StateBounds = [map.XWorldLimits;map.YWorldLimits;[-pi pi]];
```

Plan Path

Create an RRT* path planner and allow further optimization.

```
planner = plannerRRTStar(statespace,statevalidator);
planner.ContinueAfterGoalReached = true;
```

Reduce the maximum number of iterations and increase the maximum connection distance.

```
planner.MaxIterations = 2500;
planner.MaxConnectionDistance = 0.3;
```

Define start and goal states for the path planner as `[x, y, theta]` vectors. *x* and *y* are the Cartesian coordinates, and *theta* is the orientation angle.

```
start = [2.5, 2.5, 0]; % [meters, meters, radians]
goal = [22.5, 8.75, 0];
```

Plan a path from the start state to the goal state. The plan function returns a `navPath` object.

```
rng(100,'twister') % repeatable result
[path,solutionInfo] = plan(planner,start,goal);
```

Compute and Visualize Path Metrics

Create a path metrics object.

```
pathMetricsObj = pathmetrics(path,statevalidator);
```

Check path validity. The result is `1 (true)` if the planned path is obstacle free. `0 (false)` indicates an invalid path.

```
isPathValid(pathMetricsObj)
```

```
ans = logical
   1
```

Calculate the minimum clearance of the path.

```
clearance(pathMetricsObj)
```

```
ans = 1.4142
```

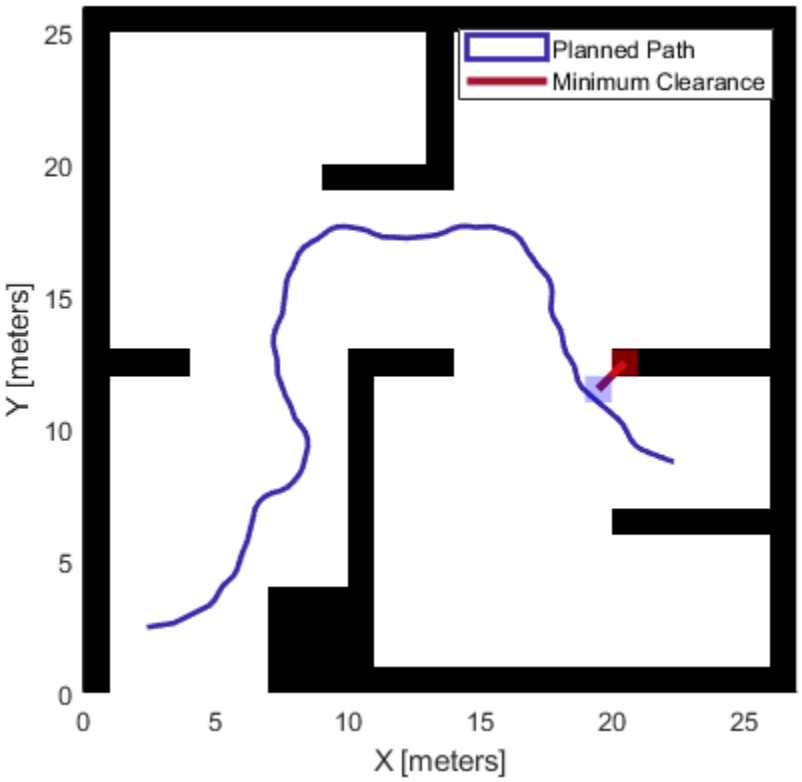
Evaluate the smoothness of the path. Values close to `0` indicate a smoother path. Straight-line paths return a value of `0`.

```
smoothness(pathMetricsObj)
```

```
ans = 1.7318
```

Visualize the minimum clearance of the path.

```
show(pathMetricsObj)
legend('Planned Path','Minimum Clearance')
```



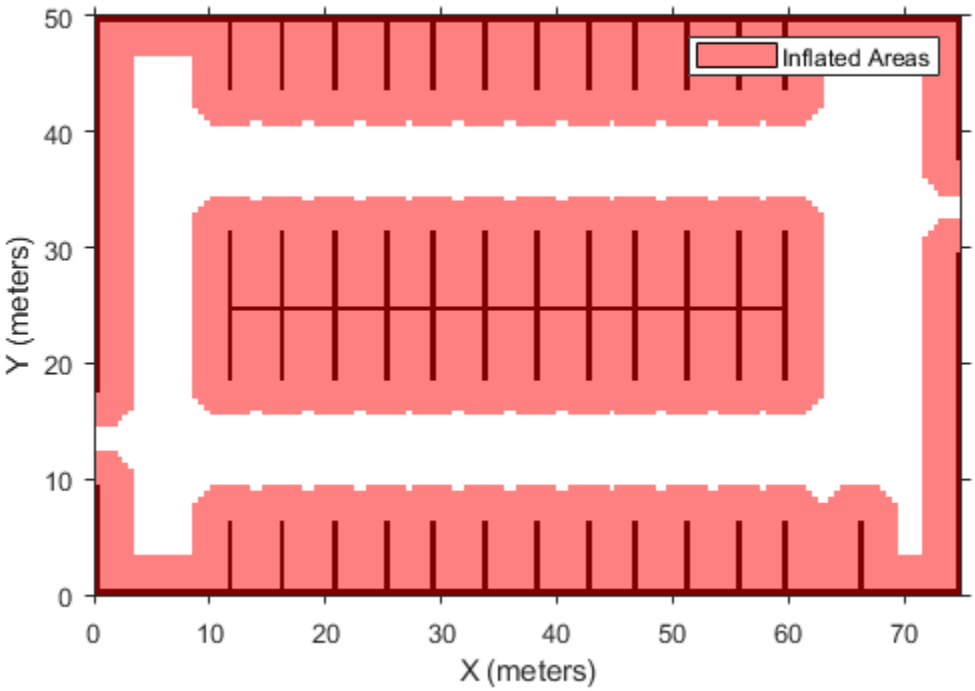
Vehicle Path Planning and Metrics Computation in a 2-D Costmap Environment

Plan a vehicle path through a parking lot using the RRT* algorithm. Compute and visualize the smoothness, clearance, and validity of the planned path.

Load and Assign Map to State Validator

Load a costmap of a parking lot. Plot the costmap to see the parking lot and the inflated areas that the vehicle should avoid.

```
load parkingLotCostmap.mat;
costmap = parkingLotCostmap;
plot(costmap)
xlabel('X (meters)')
ylabel('Y (meters)')
```



Create a `stateSpaceDubins` object and increase the minimum turing radius to 4 meters.

```
statespace = stateSpaceDubins;
statespace.MinTurningRadius = 4; % meters
```

Create a `validatorVehicleCostmap` object using the created state space.

```
statevalidator = validatorVehicleCostmap(statespace);
```

Assign the parking lot costmap to the state validator object.

```
statevalidator.Map = costmap;
```

Plan Path

Define start and goal poses for the vehicle as [*x*, *y*, `θ`] vectors. World units for the (*x*,*y*) locations are in meters. World units for the orientation angles `θ` are in degrees.

```
startPose = [5, 5, 90]; % [meters, meters, degrees]
goalPose = [40, 38, 180]; % [meters, meters, degrees]
```

Use a `pathPlannerRRT` (Automated Driving Toolbox) object and the `plan` (Automated Driving Toolbox) function to plan the vehicle path from the start pose to the goal pose.

```
planner = pathPlannerRRT(costmap);
refPath = plan(planner,startPose,goalPose);
```

Interpolate along the path at every one meter. Convert the orientation angles from degrees to radians.

```
poses = zeros(size(refPath.PathSegments,2)+1,3);
poses(1,:) = refPath.StartPose;
for i = 1:size(refPath.PathSegments,2)
    poses(i+1,:) = refPath.PathSegments(i).GoalPose;
end
poses(:,3) = deg2rad(poses(:,3));
```

Create a `navPath` object using the Dubins state space object and the states specified by `poses`.

```
path = navPath(statespace,poses);
```

Compute and Visualize Path Metrics

Create a `pathmetrics` object.

```
pathMetricsObj = pathmetrics(path,statevalidator);
```

Check path validity. The result is `1` (`true`) if the planned path is obstacle free. `0` (`false`) indicates an invalid path.

```
isPathValid(pathMetricsObj)
```

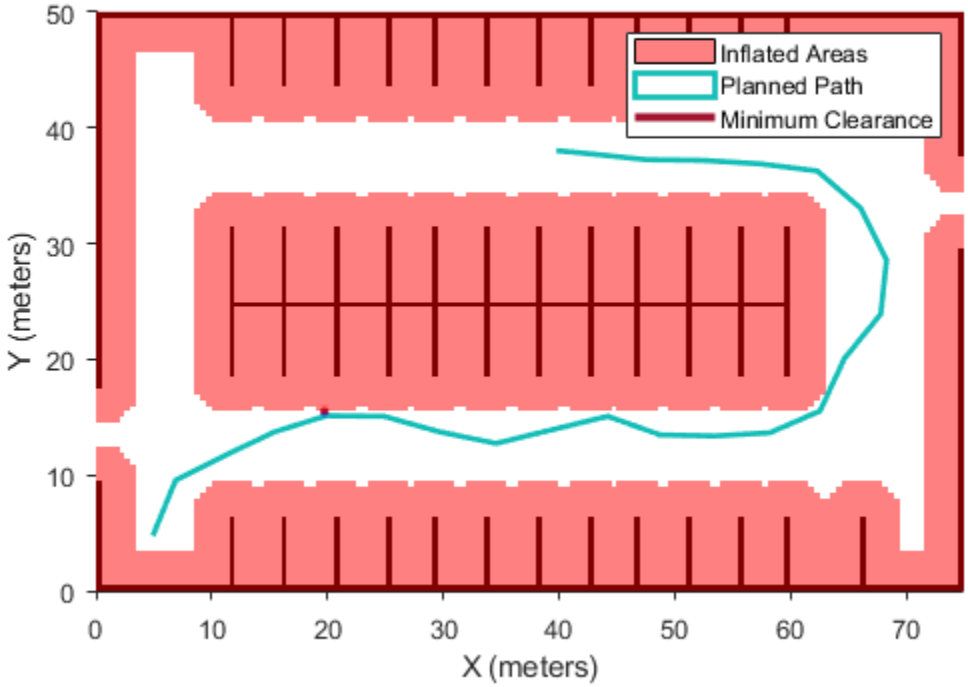
```
ans = logical
   1
```

Compute and visualize the minimum `clearance` of the path.

```
clearance(pathMetricsObj)
```

```
ans = 0.5000
```

```
show(pathMetricsObj)
legend('Inflated Areas','Planned Path','Minimum Clearance')
xlabel('X (meters)')
ylabel('Y (meters)')
```

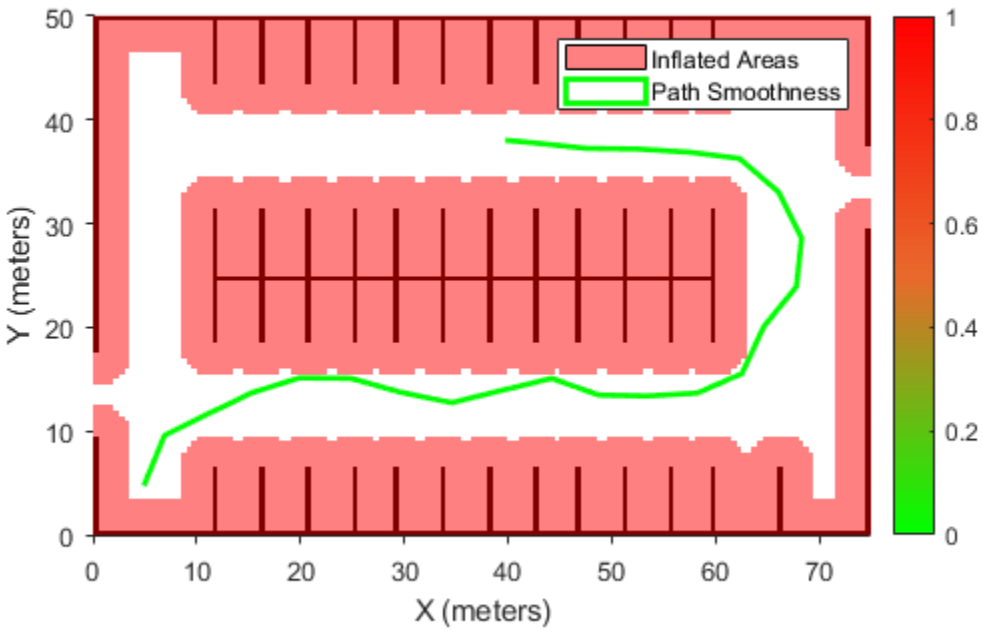


Compute and visualize the `smoothness` of the path. Values close to `0` indicate a smoother path. Straight-line paths return a value of `0`.

```
smoothness(pathMetricsObj)
```

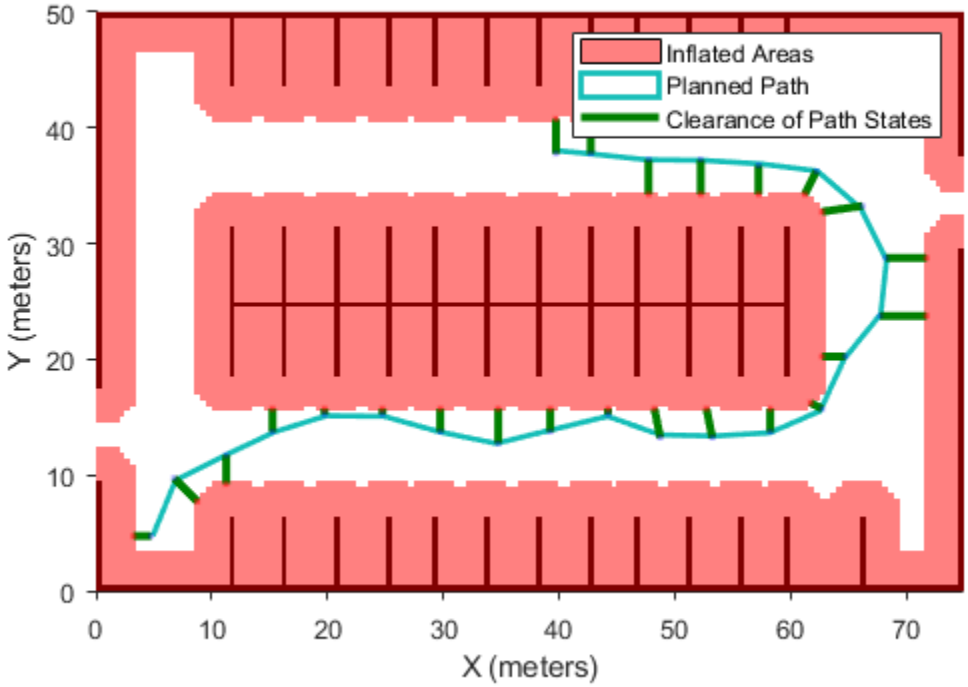
```
ans = 0.0842
```

```
show(pathMetricsObj,'Metrics',{'Smoothness'})
legend('Inflated Areas','Path Smoothness')
xlabel('X (meters)')
ylabel('Y (meters)')
```



Visualize the clearance for each state of the path.

```
show(pathMetricsObj,'Metrics',{'StatesClearance'})
legend('Inflated Areas','Planned Path','Clearance of Path States')
xlabel('X (meters)')
ylabel('Y (meters)')
```



Input Arguments

`pathMetricsObj` — Information for path metrics

`pathmetrics` object

Information for path metrics, specified as a `pathmetrics` object.

See Also

Objects

`pathmetrics`

Functions

`clearance` | `show` | `smoothness`

Introduced in R2019b

show

Visualize path metrics in map environment

Syntax

```
show(pathMetricsObj)
show(pathMetricsObj,Name,Value)
axHandle = show(pathMetricsObj)
```

Description

`show(pathMetricsObj)` plots the path in the map environment with the minimum clearance.

`show(pathMetricsObj,Name,Value)` specifies additional options using one or more name-value pair arguments.

`axHandle = show(pathMetricsObj)` outputs the axes handle of the figure used to plot the path.

Examples

Compute Path Metrics

Compute smoothness, clearance, and validity of a planned path based on a set of poses and the associated map environment.

Load and Assign Map to State Validator

Create an occupancy map from an example map and set the map resolution.

```
load exampleMaps.mat; % simpleMap
mapResolution = 1; % cells/meter
map = occupancyMap(simpleMap,mapResolution);
```

Create a Dubins state space.

```
statespace = stateSpaceDubins;
```

Create a state validator based on occupancy map to store the parameters and states in the Dubins state space.

```
statevalidator = validatorOccupancyMap(statespace);
```

Assign the map to the validator.

```
statevalidator.Map = map;
```

Set the validation distance for the validator.

```
statevalidator.ValidationDistance = 0.01;
```

Update the state space bounds to be the same as the map limits.

```
statespace.StateBounds = [map.XWorldLimits;map.YWorldLimits;[-pi pi]];
```

Plan Path

Create an RRT* path planner and allow further optimization.

```
planner = plannerRRTStar(statespace,statevalidator);
planner.ContinueAfterGoalReached = true;
```

Reduce the maximum number of iterations and increase the maximum connection distance.

```
planner.MaxIterations = 2500;
planner.MaxConnectionDistance = 0.3;
```

Define start and goal states for the path planner as `[`*x, y, theta*`]` vectors. *x* and *y* are the Cartesian coordinates, and *theta* is the orientation angle.

```
start = [2.5, 2.5, 0]; % [meters, meters, radians]
goal = [22.5, 8.75, 0];
```

Plan a path from the start state to the goal state. The plan function returns a `navPath` object.

```
rng(100,'twister') % repeatable result
[path,solutionInfo] = plan(planner,start,goal);
```

Compute and Visualize Path Metrics

Create a path metrics object.

```
pathMetricsObj = pathmetrics(path,statevalidator);
```

Check path validity. The result is `1 (true)` if the planned path is obstacle free. `0 (false)` indicates an invalid path.

```
isPathValid(pathMetricsObj)
```

```
ans = logical
   1
```

Calculate the minimum clearance of the path.

```
clearance(pathMetricsObj)
```

```
ans = 1.4142
```

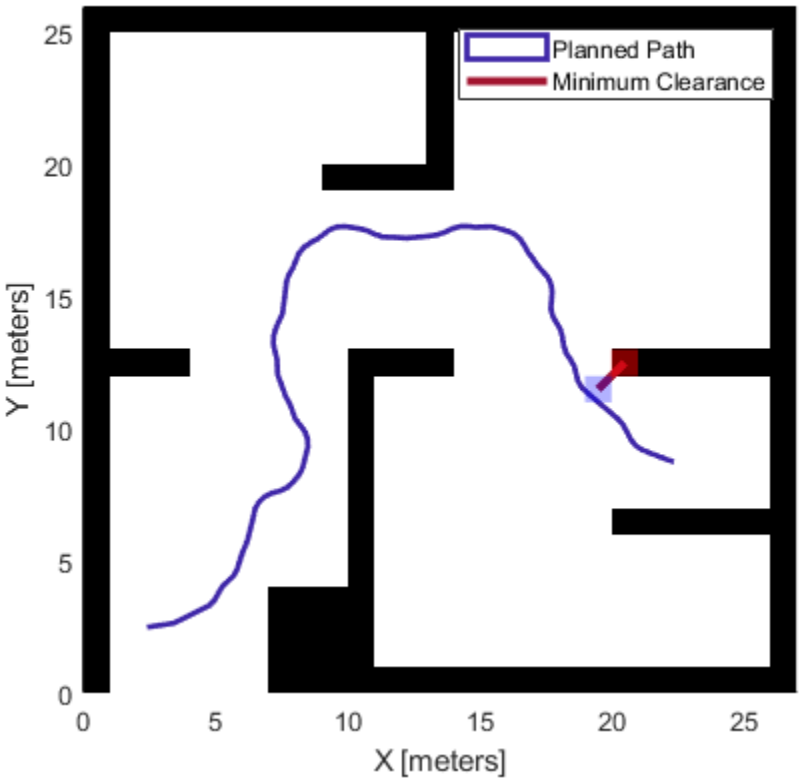
Evaluate the smoothness of the path. Values close to `0` indicate a smoother path. Straight-line paths return a value of `0`.

```
smoothness(pathMetricsObj)
```

```
ans = 1.7318
```

Visualize the minimum clearance of the path.

```
show(pathMetricsObj)
legend('Planned Path','Minimum Clearance')
```



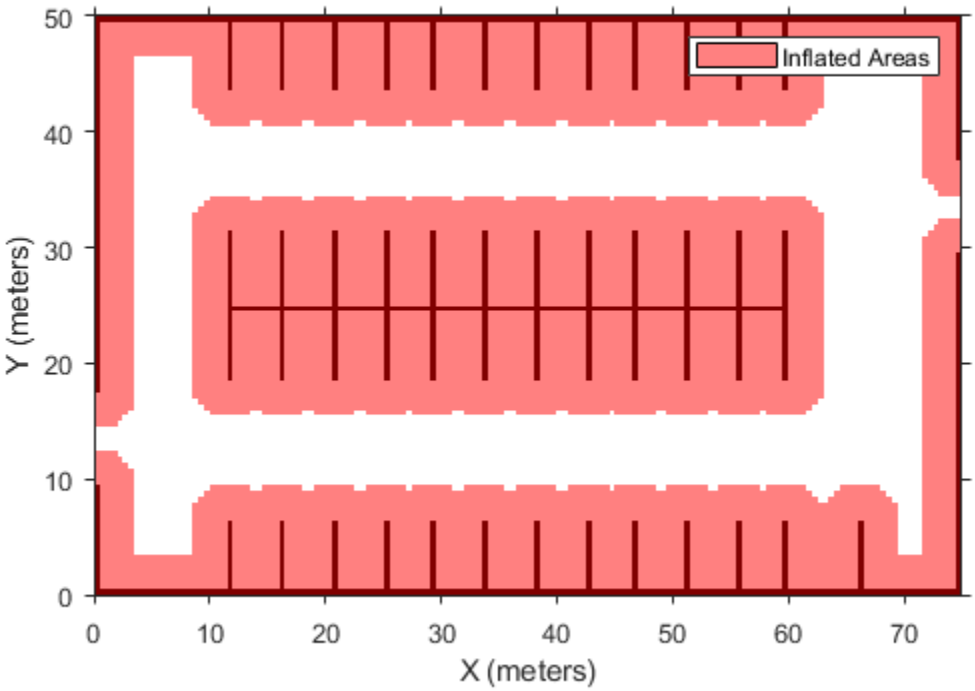
Vehicle Path Planning and Metrics Computation in a 2-D Costmap Environment

Plan a vehicle path through a parking lot using the RRT* algorithm. Compute and visualize the smoothness, clearance, and validity of the planned path.

Load and Assign Map to State Validator

Load a costmap of a parking lot. Plot the costmap to see the parking lot and the inflated areas that the vehicle should avoid.

```
load parkingLotCostmap.mat;
costmap = parkingLotCostmap;
plot(costmap)
xlabel('X (meters)')
ylabel('Y (meters)')
```



Create a `stateSpaceDubins` object and increase the minimum turing radius to 4 meters.

```
statespace = stateSpaceDubins;
statespace.MinTurningRadius = 4; % meters
```

Create a `validatorVehicleCostmap` object using the created state space.

```
statevalidator = validatorVehicleCostmap(statespace);
```

Assign the parking lot costmap to the state validator object.

```
statevalidator.Map = costmap;
```

Plan Path

Define start and goal poses for the vehicle as [*x*, *y*, `θ`] vectors. World units for the (*x*,*y*) locations are in meters. World units for the orientation angles `θ` are in degrees.

```
startPose = [5, 5, 90]; % [meters, meters, degrees]
goalPose = [40, 38, 180]; % [meters, meters, degrees]
```

Use a `pathPlannerRRT` (Automated Driving Toolbox) object and the `plan` (Automated Driving Toolbox) function to plan the vehicle path from the start pose to the goal pose.

```
planner = pathPlannerRRT(costmap);
refPath = plan(planner,startPose,goalPose);
```

Interpolate along the path at every one meter. Convert the orientation angles from degrees to radians.

```
poses = zeros(size(refPath.PathSegments,2)+1,3);
poses(1,:) = refPath.StartPose;
for i = 1:size(refPath.PathSegments,2)
    poses(i+1,:) = refPath.PathSegments(i).GoalPose;
end
poses(:,3) = deg2rad(poses(:,3));
```

Create a `navPath` object using the Dubins state space object and the states specified by `poses`.

```
path = navPath(statespace,poses);
```

Compute and Visualize Path Metrics

Create a `pathmetrics` object.

```
pathMetricsObj = pathmetrics(path,statevalidator);
```

Check path validity. The result is `1` (`true`) if the planned path is obstacle free. `0` (`false`) indicates an invalid path.

```
isPathValid(pathMetricsObj)
```

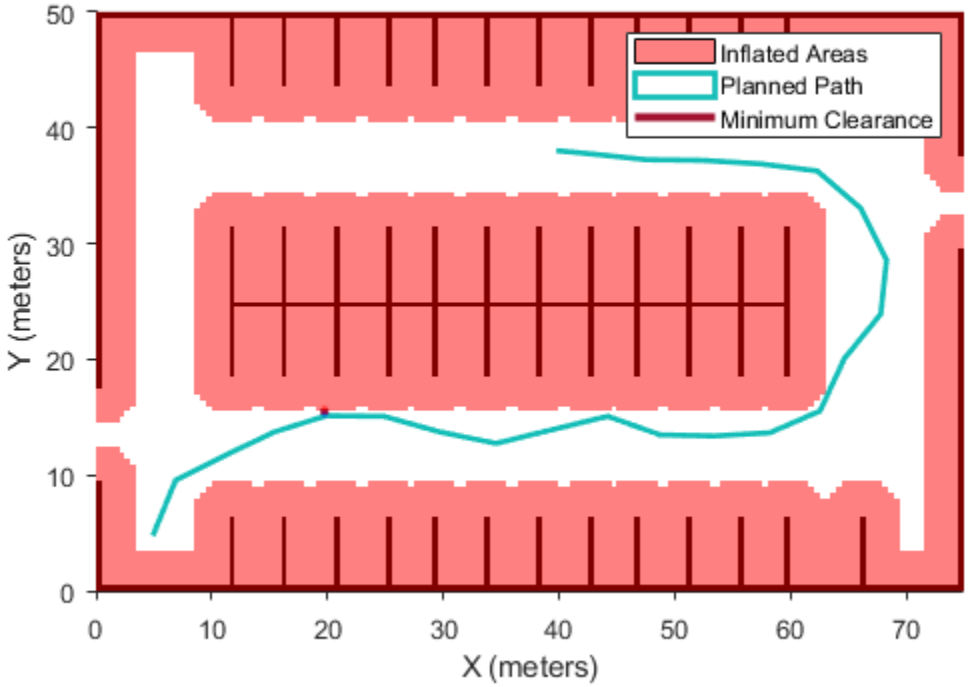
```
ans = logical
   1
```

Compute and visualize the minimum `clearance` of the path.

```
clearance(pathMetricsObj)
```

```
ans = 0.5000
```

```
show(pathMetricsObj)
legend('Inflated Areas','Planned Path','Minimum Clearance')
xlabel('X (meters)')
ylabel('Y (meters)')
```

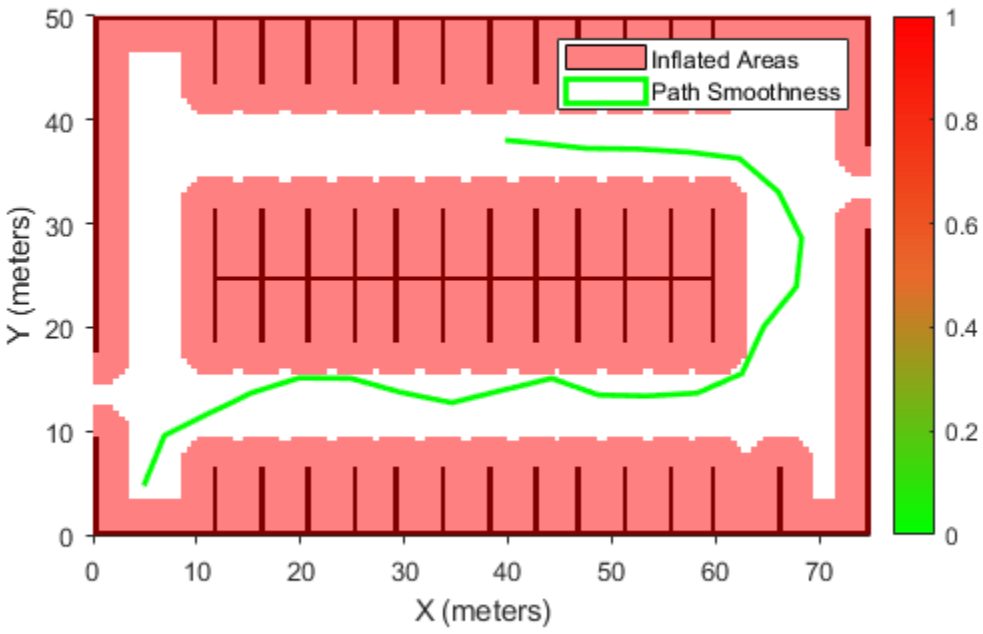


Compute and visualize the `smoothness` of the path. Values close to `0` indicate a smoother path. Straight-line paths return a value of `0`.

```
smoothness(pathMetricsObj)
```

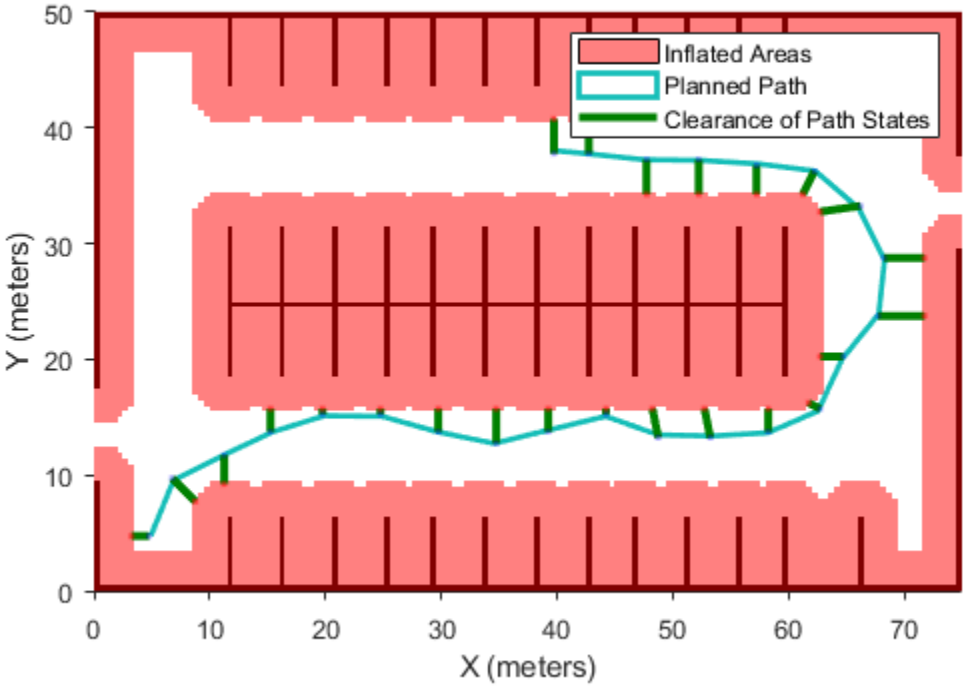
```
ans = 0.0842
```

```
show(pathMetricsObj,'Metrics',{'Smoothness'})
legend('Inflated Areas','Path Smoothness')
xlabel('X (meters)')
ylabel('Y (meters)')
```



Visualize the clearance for each state of the path.

```
show(pathMetricsObj,'Metrics',{'StatesClearance'})
legend('Inflated Areas','Planned Path','Clearance of Path States')
xlabel('X (meters)')
ylabel('Y (meters)')
```



Input Arguments

`pathMetricsObj` — Information for path metrics

`pathmetrics` object

Information for path metrics, specified as a `pathmetrics` object.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'Parent',axHandle`

`Parent` — Axes used to plot path

`Axes` object | `UIAxes` object

Axes used to plot path, specified as the comma-separated pair consisting of `'Parent'` and either an `axes` or `uiaxes` object. If you do not specify `Parent`, a new figure is created.

Example: `show(pathMetricsObj,'Parent',axHandle)`

`Metrics` — Display metrics option

cell array of strings

Display metrics option, specified as the comma-separated pair consisting of `'Metrics'` and a cell array with any combination of these values:

- `'MinClearance'` — Display minimum clearance of path.
- `'StatesClearance'` — Display clearance of path states.
- `'Smoothness'` — Display path smoothness.

Example: `show(pathMetricsObj,'Metrics',{'Smoothness','StatesClearance'})`

Data Types: `cell`

Output Arguments

`axHandle` — Axes used to plot path
`Axes` object | `UIAxes` object

Axes used to plot path, returned as either an `axes` or `uiaxes` object.

See Also

Objects
`pathmetrics`

Functions
`clearance` | `isPathValid` | `smoothness`

Introduced in R2019b

smoothness

Smoothness of path

Syntax

```
smoothness(pathMetricsObj)
smoothness(pathMetricsObj,'Type','segments')
```

Description

`smoothness(pathMetricsObj)` evaluates the smoothness of the planned path. Values close to `0` indicate a smoother path. Straight-line paths return a value of `0`.

`smoothness(pathMetricsObj,'Type','segments')` returns individual smoothness calculations between each set of three poses on the path, in the form of a (*n*–2)-element vector, where *n* is the number of poses.

Examples

Compute Path Metrics

Compute smoothness, clearance, and validity of a planned path based on a set of poses and the associated map environment.

Load and Assign Map to State Validator

Create an occupancy map from an example map and set the map resolution.

```
load exampleMaps.mat; % simpleMap
mapResolution = 1; % cells/meter
map = occupancyMap(simpleMap,mapResolution);
```

Create a Dubins state space.

```
statespace = stateSpaceDubins;
```

Create a state validator based on occupancy map to store the parameters and states in the Dubins state space.

```
statevalidator = validatorOccupancyMap(statespace);
```

Assign the map to the validator.

```
statevalidator.Map = map;
```

Set the validation distance for the validator.

```
statevalidator.ValidationDistance = 0.01;
```

Update the state space bounds to be the same as the map limits.

```
statespace.StateBounds = [map.XWorldLimits;map.YWorldLimits;[-pi pi]];
```

Plan Path

Create an RRT* path planner and allow further optimization.

```
planner = plannerRRTStar(statespace,statevalidator);
planner.ContinueAfterGoalReached = true;
```

Reduce the maximum number of iterations and increase the maximum connection distance.

```
planner.MaxIterations = 2500;
planner.MaxConnectionDistance = 0.3;
```

Define start and goal states for the path planner as `[`*x, y, theta*`]` vectors. *x* and *y* are the Cartesian coordinates, and *theta* is the orientation angle.

```
start = [2.5, 2.5, 0]; % [meters, meters, radians]
goal = [22.5, 8.75, 0];
```

Plan a path from the start state to the goal state. The plan function returns a `navPath` object.

```
rng(100,'twister') % repeatable result
[path,solutionInfo] = plan(planner,start,goal);
```

Compute and Visualize Path Metrics

Create a path metrics object.

```
pathMetricsObj = pathmetrics(path,statevalidator);
```

Check path validity. The result is `1 (true)` if the planned path is obstacle free. `0 (false)` indicates an invalid path.

```
isPathValid(pathMetricsObj)
```

```
ans = logical
   1
```

Calculate the minimum clearance of the path.

```
clearance(pathMetricsObj)
```

```
ans = 1.4142
```

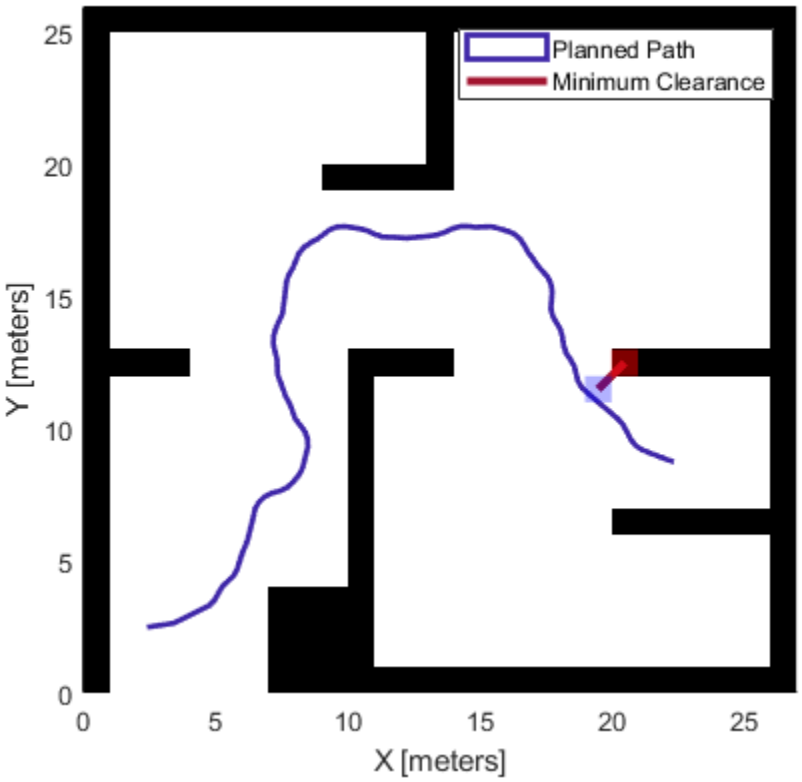
Evaluate the smoothness of the path. Values close to `0` indicate a smoother path. Straight-line paths return a value of `0`.

```
smoothness(pathMetricsObj)
```

```
ans = 1.7318
```

Visualize the minimum clearance of the path.

```
show(pathMetricsObj)
legend('Planned Path','Minimum Clearance')
```



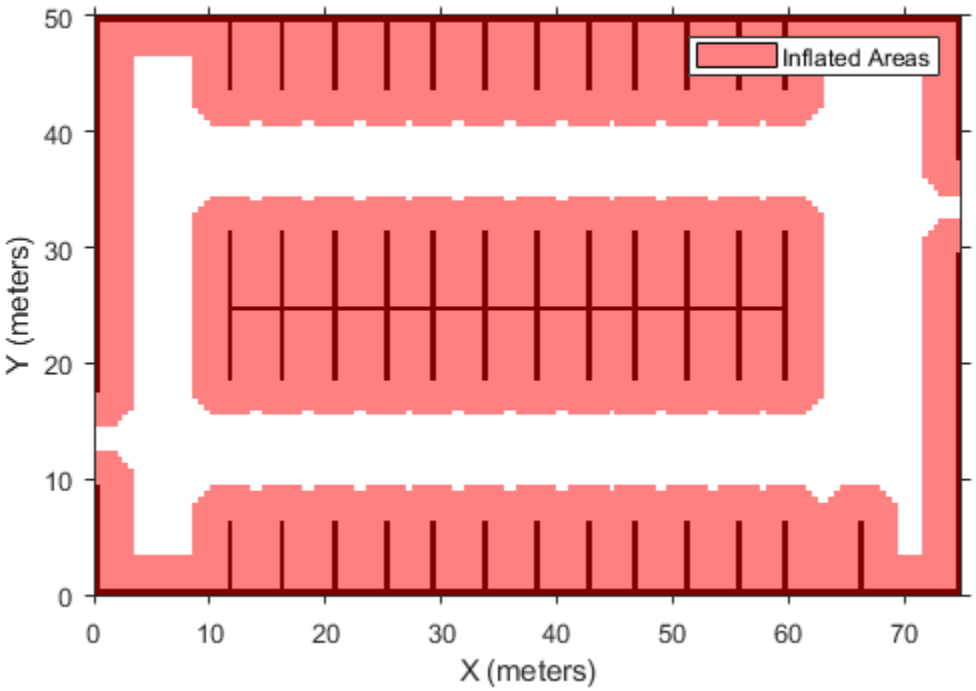
Vehicle Path Planning and Metrics Computation in a 2-D Costmap Environment

Plan a vehicle path through a parking lot using the RRT* algorithm. Compute and visualize the smoothness, clearance, and validity of the planned path.

Load and Assign Map to State Validator

Load a costmap of a parking lot. Plot the costmap to see the parking lot and the inflated areas that the vehicle should avoid.

```
load parkingLotCostmap.mat;
costmap = parkingLotCostmap;
plot(costmap)
xlabel('X (meters)')
ylabel('Y (meters)')
```



Create a `stateSpaceDubins` object and increase the minimum turing radius to 4 meters.

```
statespace = stateSpaceDubins;
statespace.MinTurningRadius = 4; % meters
```

Create a `validatorVehicleCostmap` object using the created state space.

```
statevalidator = validatorVehicleCostmap(statespace);
```

Assign the parking lot costmap to the state validator object.

```
statevalidator.Map = costmap;
```

Plan Path

Define start and goal poses for the vehicle as [*x*, *y*, `θ`] vectors. World units for the (*x*,*y*) locations are in meters. World units for the orientation angles `θ` are in degrees.

```
startPose = [5, 5, 90]; % [meters, meters, degrees]
goalPose = [40, 38, 180]; % [meters, meters, degrees]
```

Use a `pathPlannerRRT` (Automated Driving Toolbox) object and the `plan` (Automated Driving Toolbox) function to plan the vehicle path from the start pose to the goal pose.

```
planner = pathPlannerRRT(costmap);
refPath = plan(planner,startPose,goalPose);
```

Interpolate along the path at every one meter. Convert the orientation angles from degrees to radians.

```
poses = zeros(size(refPath.PathSegments,2)+1,3);
poses(1,:) = refPath.StartPose;
for i = 1:size(refPath.PathSegments,2)
    poses(i+1,:) = refPath.PathSegments(i).GoalPose;
end
poses(:,3) = deg2rad(poses(:,3));
```

Create a `navPath` object using the Dubins state space object and the states specified by `poses`.

```
path = navPath(statespace,poses);
```

Compute and Visualize Path Metrics

Create a `pathmetrics` object.

```
pathMetricsObj = pathmetrics(path,statevalidator);
```

Check path validity. The result is `1` (`true`) if the planned path is obstacle free. `0` (`false`) indicates an invalid path.

```
isPathValid(pathMetricsObj)
```

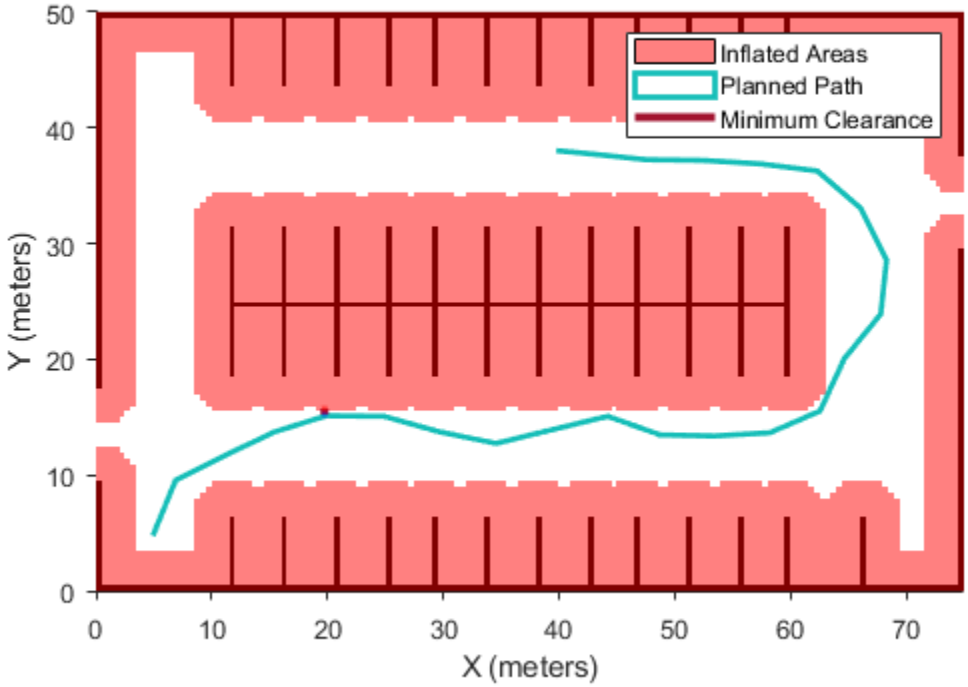
```
ans = logical
   1
```

Compute and visualize the minimum `clearance` of the path.

```
clearance(pathMetricsObj)
```

```
ans = 0.5000
```

```
show(pathMetricsObj)
legend('Inflated Areas','Planned Path','Minimum Clearance')
xlabel('X (meters)')
ylabel('Y (meters)')
```

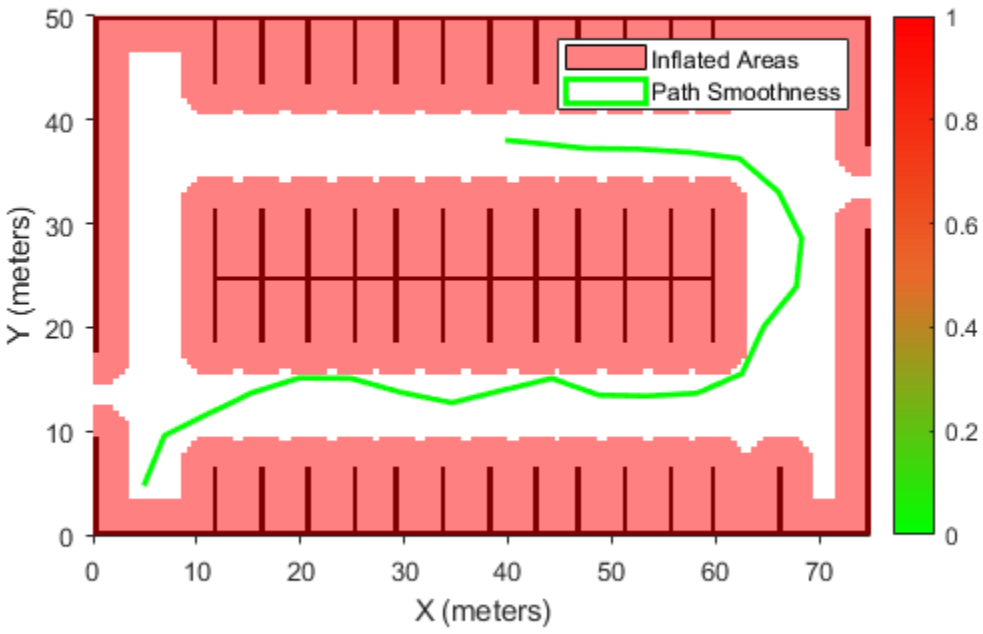


Compute and visualize the `smoothness` of the path. Values close to `0` indicate a smoother path. Straight-line paths return a value of `0`.

```
smoothness(pathMetricsObj)
```

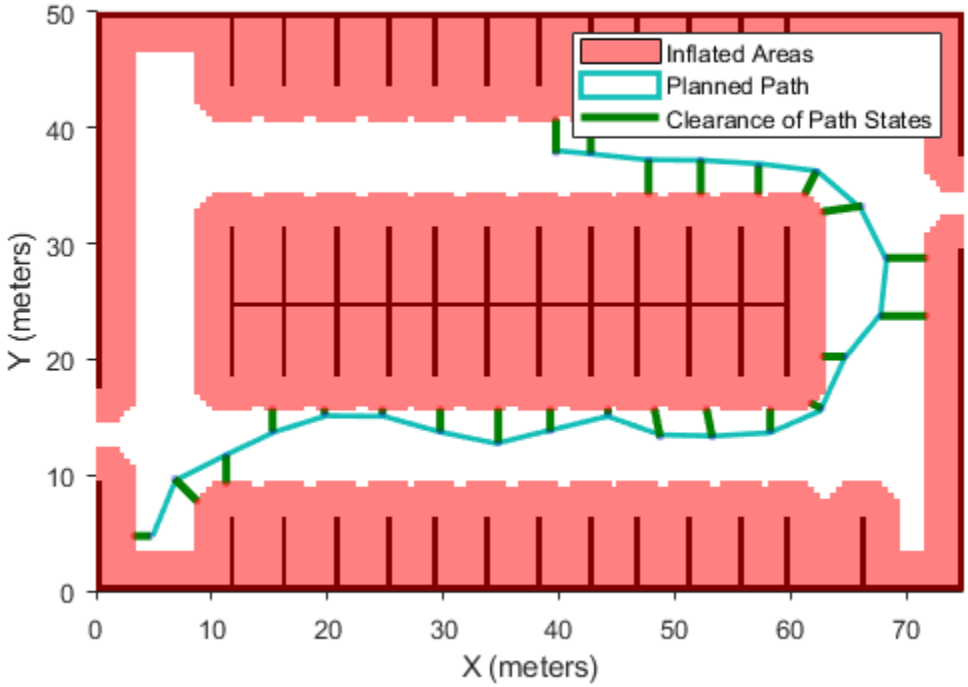
```
ans = 0.0842
```

```
show(pathMetricsObj,'Metrics',{'Smoothness'})
legend('Inflated Areas','Path Smoothness')
xlabel('X (meters)')
ylabel('Y (meters)')
```



Visualize the clearance for each state of the path.

```
show(pathMetricsObj,'Metrics',{'StatesClearance'})
legend('Inflated Areas','Planned Path','Clearance of Path States')
xlabel('X (meters)')
ylabel('Y (meters)')
```



Input Arguments

`pathMetricsObj` — Information for path metrics

`pathmetrics` object

Information for path metrics, specified as a `pathmetrics` object.

References

[1] Lindemann, Stephen R., and Steven M. LaValle. "Simple and efficient algorithms for computing smooth, collision-free feedback laws over given cell decompositions." *The International Journal of Robotics Research 28, no. 5*. 2009, pp. 600-621.

See Also

Objects

`pathmetrics`

Functions

`clearance` | `isPathValid` | `show`

Introduced in R2019b

plannerAStarGrid

A* path planner for grid map

Description

The `plannerAStarGrid` object creates an A* path planner. The planner performs an A* search on an occupancy map and finds shortest obstacle-free path between the specified start and goal grid locations as determined by heuristic cost.

Creation

Syntax

```
planner = plannerAStarGrid
planner = plannerAStarGrid(map)
planner = plannerAStarGrid( ___ ,Name,Value)
```

Description

`planner = plannerAStarGrid` creates a `plannerAStarGrid` object with a `binaryOccupancyMap` object using a width and height of 10 meters and grid resolution of 1 cell per meter.

`planner = plannerAStarGrid(map)` creates a `plannerAStarGrid` object using the specified map object `map`. Specify `map` as either a `binaryOccupancyMap` or `occupancyMap` object. The `map` input sets the value of the Map property.

`planner = plannerAStarGrid( ___ ,Name,Value)` sets properties using one or more name-value pairs. Unspecified properties have default values. Enclose each property name in quotes.

For example, `plannerAStarGrid(map,'GCost','Manhattan')` creates an A* path planner object using the Manhattan cost function.

Properties

`Map` — Map representation

`binaryOccupancyMap` object (default) | `occupancyMap` object

Map representation, specified as either a `binaryOccupancyMap` or `occupancyMap` object. This object represents the environment of the robot as an occupancy grid. The value of each grid cell indicates the occupancy of the associated location in the map.

Example: `planner.Map = binaryOccupancyMap(zeros(50,50));`

`GCost` — General cost of moving between any two points in grid

`'Euclidean'` (default) | `'Chebyshev'` | `'EulideanSquared'` | `'Manhattan'`

The general cost of moving between any two points in a grid, specified as one of the following predefined cost functions `'Chebyshev'`, `'Euclidean'`, `'EulideanSquared'`, or `'Manhattan'`.

Note You can either use the predefined cost functions or a custom cost function. To use a custom cost function, see GCostFcn property.

Example: `planner = plannerAStarGrid(map,'GCost','Manhattan');`

Example: `planner.GCost = 'Chebyshev';`

Data Types: `string` | `char`

`GCostFcn` — Custom GCost function

function handle

Custom GCost function, specified as a function handle. The function handle must accept two pose inputs as a [*row column*] vectors and return a scalar of type double.

Note You can either use the predefined cost functions or a custom cost function. To use the predefined cost functions, see GCost property.

Example: `planner = plannerAStarGrid(map,'GCostFcn',@(pose1,pose2)sum(abs(pose1-pose2),2));`

Example: `planner.GCostFcn = @(pose1,pose2)sum(abs(pose1-pose2),2);`

Data Types: `function_handle`

`HCost` — Heuristic cost between point and goal in grid

`'Euclidean'` (default) | `'Chebyshev'` | `'EulideanSquared'` | `'Manhattan'`

The heuristic cost between a point and the goal in a grid, specified as one of the following predefined cost functions `'Chebyshev'`, `'Euclidean'`, `'EulideanSquared'`, or `'Manhattan'`.

Note You can either use the predefined cost functions or a custom cost function. To use a custom cost function, see HCostFcn property.

Example: `planner = plannerAStarGrid(map,'HCost','Manhattan');`

Example: `planner.HCost = 'Chebyshev';`

Data Types: `string` | `char`

`HCostFcn` — Custom HCost function

function handle

Custom HCost function, specified as a function handle. The function handle must accept two pose inputs as a [*row column*] vectors and return a scalar of type double.

Note You can either use the predefined cost functions or a custom cost function. To use the predefined cost functions, see HCost property.

Example: `planner = plannerAStarGrid(map,'HCostFcn',@(pose1,pose2)sum(abs(pose1-pose2),2));`

Example: `planner.HCostFcn = @(pose1,pose2)sum(abs(pose1-pose2),2);`

Data Types: `function_handle`

`TieBreaker` — Toggle tiebreaker mode
`'off'` (default) | `'on'`

Toggle tiebreaker mode, specified as either `'on'` or `'off'`.

When you enable the `TieBreaker` property, the A* path planner chooses between multiple paths of the same length by adjusting the heuristic cost value.

Example: `planner = plannerAStarGrid(map,'TieBreaker','on');`

Example: `planner.TieBreaker = 'off';`

Data Types: `string` | `char`

Object Functions

plan Find shortest obstacle-free path between two points
show Plot and visualize A* explored nodes and planned path

Examples

Plan Obstacle-Free Path in Grid Map Using A-Star Path Planner

Plan the shortest collision-free path through an obstacle grid map using the A* path planning algorithm.

Generate a `binaryOccupancyMap` object with randomly scattered obstacles using the `mapClutter` function.

```
rng('default');
map = mapClutter;
```

Use the map to create a `plannerAStarGrid` object.

```
planner = plannerAStarGrid(map);
```

Define the start and goal points.

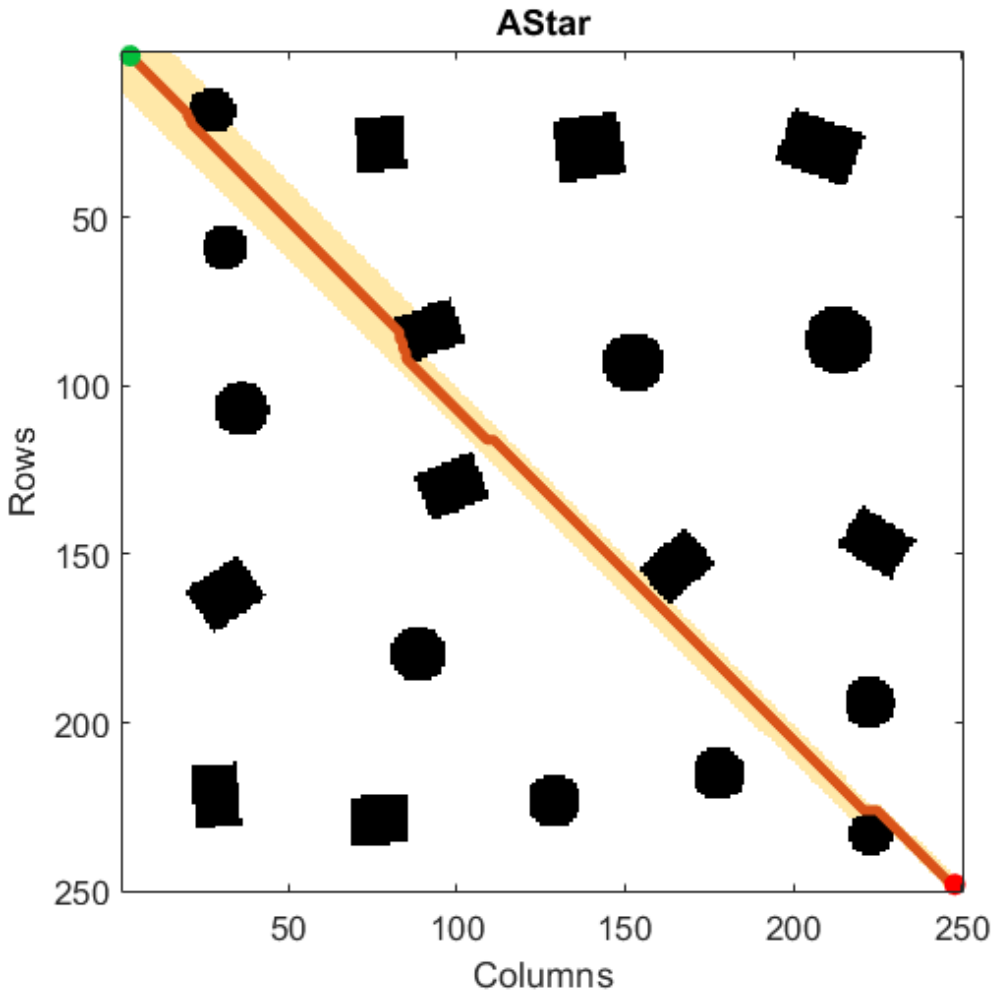
```
start = [2 3];
goal = [248 248];
```

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

```
plan(planner,start,goal);
```

Visualize the path and the explored nodes using the `show` object function.

```
show(planner)
```



Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`binaryOccupancyMap` | `occupancyMap` | `plannerRRT` | `plannerRRTStar` | `plannerHybridAStar`

Introduced in R2020b

plan

Find shortest obstacle-free path between two points

Syntax

```
path = plan(planner,start,goal)
path = plan(planner,start,goal,'world')
[path,debugInfo] = plan( ___ )
```

Description

`path = plan(planner,start,goal)` finds the shortest obstacle-free path, `path`, between a specified start point, `start`, and goal point, `goal`, specified as `[row column]` in grid frame with origin at top-left corner, using the specified A* path planner `planner`.

`path = plan(planner,start,goal,'world')` finds the shortest obstacle-free path, `path`, between a specified start point, `start`, and goal point, `goal`, specified as `[x y]` in world coordinate frame with origin at bottom-left corner, using the specified A* path planner `planner`.

`[path,debugInfo] = plan( ___ )` also returns `debugInfo` that contains the path cost, number of nodes explored, and GCost for each explored node.

Examples

Plan Obstacle-Free Path in Grid Map Using A-Star Path Planner

Plan the shortest collision-free path through an obstacle grid map using the A* path planning algorithm.

Generate a `binaryOccupancyMap` object with randomly scattered obstacles using the `mapClutter` function.

```
rng('default');
map = mapClutter;
```

Use the map to create a `plannerAStarGrid` object.

```
planner = plannerAStarGrid(map);
```

Define the start and goal points.

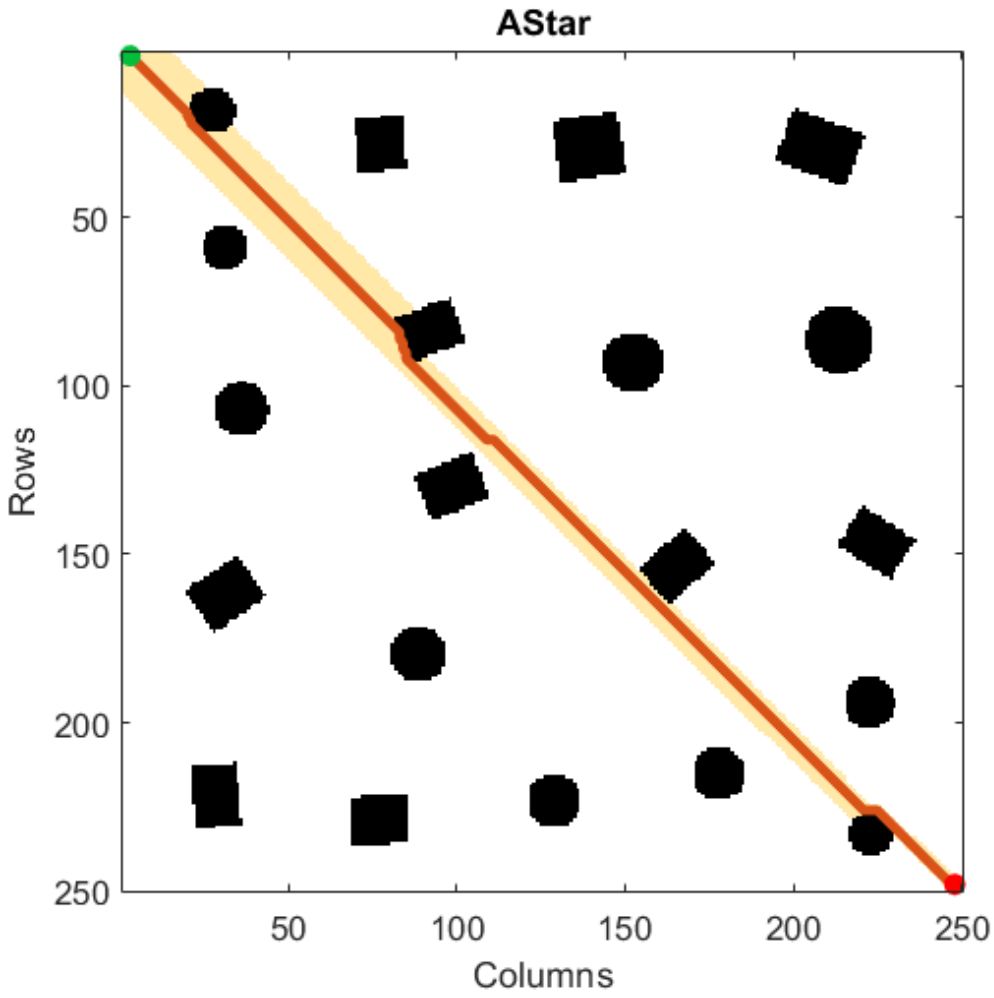
```
start = [2 3];
goal = [248 248];
```

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

```
plan(planner,start,goal);
```

Visualize the path and the explored nodes using the `show` object function.

```
show(planner)
```



Input Arguments

`planner` — A* path planner for grid map
`plannerAStarGrid` object

A* path planner for a grid map, specified as a `plannerAStarGrid` object.

`start` — Start position in grid or world
two-element vector

Start position in the grid or world, specified as a two-element vector of the form [*row column*], or [*x y*]. The location is in grid positions or world coordinates based on syntax.

Example: `[2 3]`

Data Types: `double`

`goal` — Goal position in grid or world
two-element vector

Goal position in the grid or world, specified as a two-element vector of the form [*row column*], or [*x y*]. The location is in grid positions or world coordinates based on syntax.

Example: `[28 46]`

Data Types: `double`

Output Arguments

`path` — Shortest obstacle-free path

n-by-2 matrix

Shortest obstacle-free path, returned as an *n*-by-2 matrix. *n* is the number of waypoints in the path. Each row represents the [*row column*], or [*x y*] location of a waypoint along the solved path from the start location to the goal. The location is in grid positions or world coordinates based on syntax.

Data Types: `double`

`debugInfo` — Debugging information for path result

structure

Debugging information for the path result, returned as a structure with these fields:

- `PathCost` — Cost of the path
- `NumNodesExplored` — Number of nodes explored
- `GCostMatrix` — GCost for each explored node

Data Types: `struct`

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`plannerAStarGrid` | `show`

Introduced in R2020b

show

Plot and visualize A* explored nodes and planned path

Syntax

```
show(planner)
axHandle = show(planner)
[___] = show(___,Name,Value)
```

Description

`show(planner)` plots and visualizes the A* explored nodes and the planned path in the associated map.

`axHandle = show(planner)` returns the axes handle of the figure used to plot the path.

`[___] = show(___,Name,Value)` specifies options using one or more name-value pair arguments in addition to any of the arguments from previous syntaxes. Enclose argument name inside single quotes (' ').

For example, `'ExploredNodes','off'` plots and visualizes the planned path without displaying the explored nodes.

Examples

Plan Obstacle-Free Path in Grid Map Using A-Star Path Planner

Plan the shortest collision-free path through an obstacle grid map using the A* path planning algorithm.

Generate a `binaryOccupancyMap` object with randomly scattered obstacles using the `mapClutter` function.

```
rng('default');
map = mapClutter;
```

Use the map to create a `plannerAStarGrid` object.

```
planner = plannerAStarGrid(map);
```

Define the start and goal points.

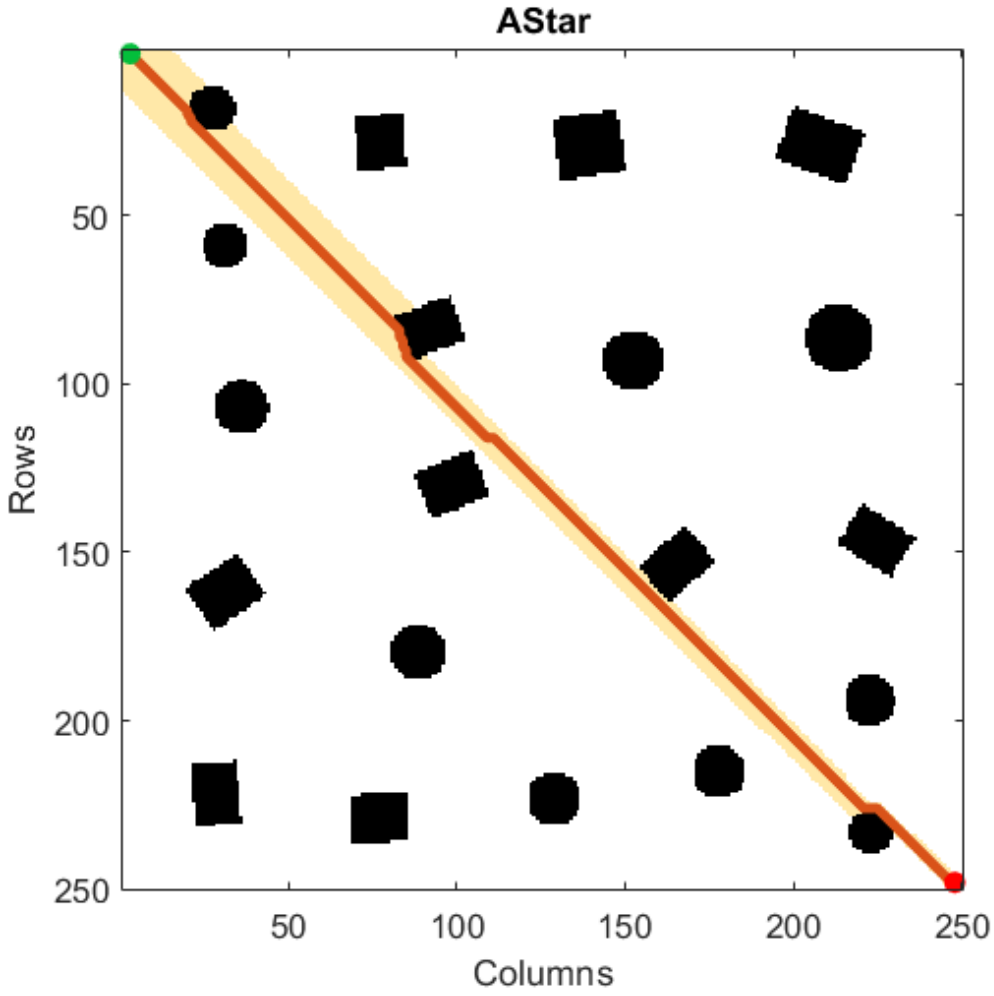
```
start = [2 3];
goal = [248 248];
```

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

```
plan(planner,start,goal);
```

Visualize the path and the explored nodes using the `show` object function.

```
show(planner)
```



Input Arguments

`planner` — A* path planner for grid map

`plannerAStarGrid` object

A* path planner for a grid map, specified as a `plannerAStarGrid` object.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'ExploredNodes','off'` plots and visualizes the planned path without displaying the explored nodes.

`Parent` — Axes used to plot path
`Axes` object | `UIAxes` object

Axes used to plot path, specified as the comma-separated pair consisting of `'Parent'` and either an Axes Properties or UIAxes Properties object. If you do not specify `Parent`, a new figure is created.

Example: `'Parent',axHandle`

`ExploredNodes` — Display explored nodes
`'on'` (default) | `'off'`

Display the explored nodes, specified as the comma-separated pair consisting of `'ExploredNodes'` and either `'on'` or `'off'`.

Example: `'ExploredNodes','off'`

Data Types: `string` | `char`

Output Arguments

`axHandle` — Axes used to plot path
`Axes` object | `UIAxes` object

Axes used to plot the path, returned as an Axes Properties or UIAxes Properties object.

See Also
`plannerAStarGrid` | `plan`

Introduced in R2020b

plannerBiRRT

Create bidirectional RRT planner for geometric planning

Description

The `plannerBiRRT` object is a single-query planner that uses the bidirectional rapidly exploring random tree (RRT) algorithm with an optional connect heuristic for increased speed.

The bidirectional RRT planner creates one tree with a root node at the specified start state and another tree with a root node at the specified goal state. To extend each tree, the planner generates a random state and, if valid, takes a step from the nearest node based on the MaxConnectionDistance property. The start and goal trees alternate this extension process until both trees are connected. If the EnableConnectHeuristic property is enabled, the extension process ignores the MaxConnectionDistance property. Invalid states or connections that collide with the environment are not added to the tree.

Creation

Syntax

```
planner = plannerBiRRT(stateSpace,stateVal)
```

Description

`planner = plannerBiRRT(stateSpace,stateVal)` creates a bidirectional RRT planner from a state space object, `stateSpace`, and a state validator object, `stateVal`. The state space of `stateVal` must be the same as `stateSpace`. The `stateSpace` and `stateVal` arguments also set the `StateSpace` and `StateValidator` properties, respectively, of the planner.

Properties

`StateSpace` — State space for planner
state space object

State space for the planner, specified as a state space object. You can use state space objects such as `stateSpaceSE2`, `stateSpaceDubins`, and `stateSpaceReedsShepp`. You can also customize a state space object using the `nav.StateSpace` class.

`StateValidator` — State validator for planner
state validator object

State validator for the planner, specified as a state validator object. You can use state validator objects such as `validatorOccupancyMap` and `validatorVehicleCostmap`.

`MaxConnectionDistance` — Maximum length between planned configurations
`0.1` (default) | positive scalar

Maximum length between planned configurations, specified as a positive scalar.

If the EnableConnectHeuristic property is set to `true`, the object ignores this distance when connecting the two trees during the connect stage.

Data Types: `single` | `double`

`MaxIterations` — Maximum number of iterations
`1e4` (default) | positive integer

Maximum number of iterations, specified as a positive integer.

Data Types: `single` | `double`

`MaxNumTreeNodes` — Maximum number of nodes in search tree
`1e4` (default) | positive integer

Maximum number of nodes in the search tree, specified as a positive integer.

Data Types: `single` | `double`

`EnableConnectHeuristic` — Directly join trees during connect phase
`false` or `0` (default) | `true` or `1`

Directly join trees during the connect phase of the planner, specified as a logical `0` (`false`) or `1` (`true`).

Setting this property to `true` causes the object to ignore the MaxConnectionDistance property when attempting to connect the two trees together.

Data Types: `logical`

Object Functions

plan	Plan path between two states
copy	Create deep copy of planner object

Examples

Plan Path Between Two States Using Bidirectional RRT

Use the `plannerBiRRT` object to plan a path between two states in an environment with obstacles. Visualize the planned path with interpolated states.

Create a state space.

```
ss = stateSpaceSE2;
```

Create an `occupancyMap`-based state validator using the created state space.

```
sv = validatorOccupancyMap(ss);
```

Create an occupancy map from an example map and set map resolution as 10 cells per meter.

```
load exampleMaps
map = occupancyMap(ternaryMap,10);
sv.Map = map;
```

Set the validation distance for the validator.

```
sv.ValidationDistance = 0.01;
```

Update the state space bounds to be the same as the map limits.

```
ss.StateBounds = [map.XWorldLimits; map.YWorldLimits; [-pi pi]];
```

Create the path planner and increase the maximum connection distance.

```
planner = plannerBiRRT(ss,sv);
planner.MaxConnectionDistance = 0.3;
```

Specify the start and goal states.

```
start = [20 10 0];
goal = [40 40 0];
```

Plan a path. Due to the randomness of the RRT algorithm, set the `rng` seed for repeatability.

```
rng(100,'twister')
[pthObj,solnInfo] = plan(planner,start,goal);
```

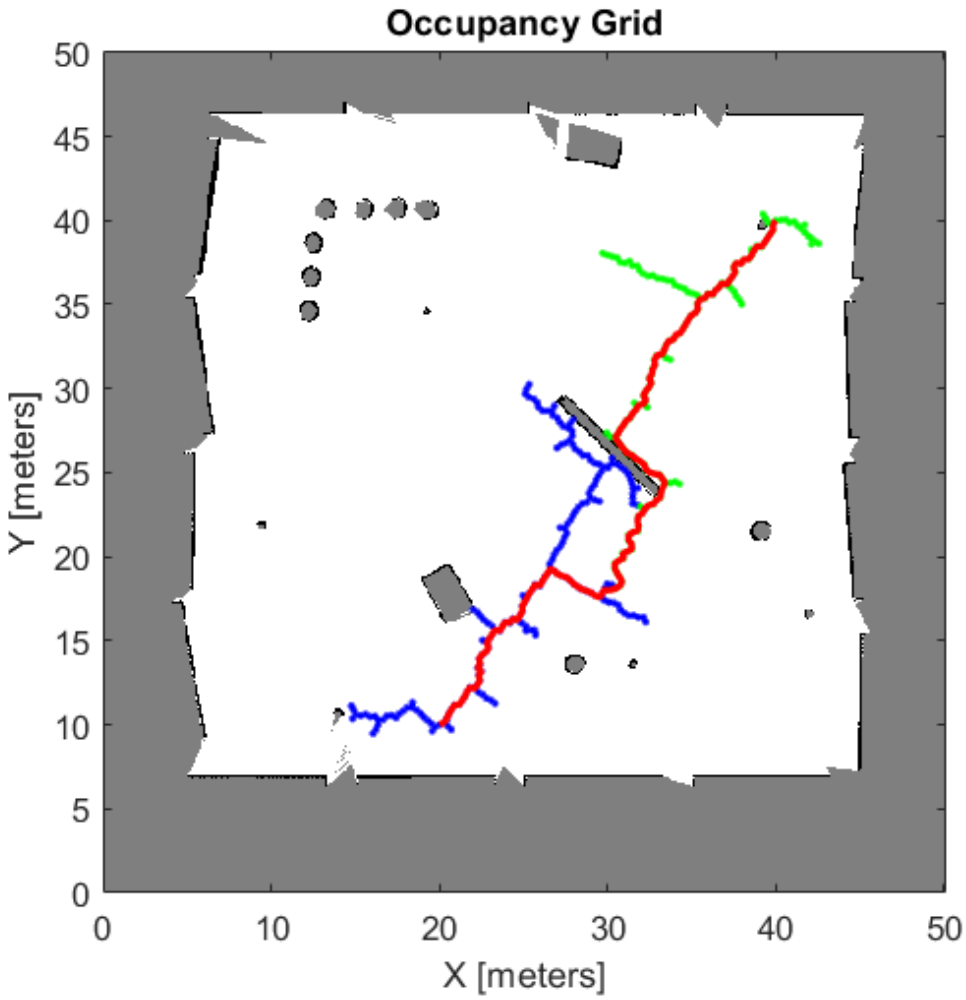
Display the number of iterations taken for the tree to converge.

```
fprintf('Number of iterations: %d\n',solnInfo.NumIterations)
```

```
Number of iterations: 346
```

Visualize the results.

```
show(map)
hold on
plot(solnInfo.StartTreeData(:,1),solnInfo.StartTreeData(:,2),'.-','color','b') % Start tree expa
plot(solnInfo.GoalTreeData(:,1),solnInfo.GoalTreeData(:,2),'.-','color','g') % Goal tree expansi
plot(pthObj.States(:,1),pthObj.States(:,2),'r-','LineWidth',2) % draw path
hold off
```



Replan the path with the `EnableConnectHeuristic` property set to true.

```
planner.EnableConnectHeuristic = true;
[pthObj,solnInfo] = plan(planner,start,goal);
```

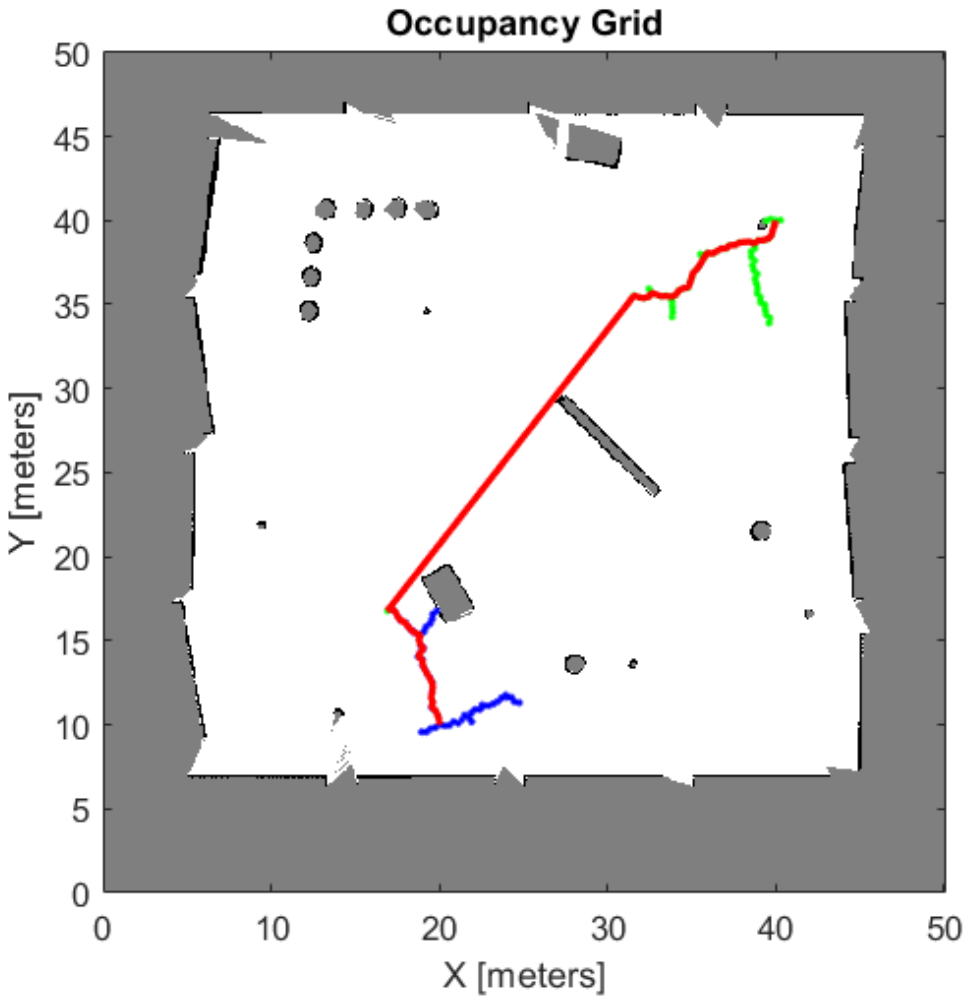
Display the number of iterations taken for the tree to converge. Observe that the planner requires significantly fewer iterations compared to when the `EnableConnectHeuristic` property is set to false.

```
fprintf('Number of iterations: %d\n',solnInfo.NumIterations)
```

```
Number of iterations: 135
```

Visualize the results.

```
figure
show(map)
hold on
plot(solnInfo.StartTreeData(:,1),solnInfo.StartTreeData(:,2),'.-','color','b') % Start tree expan
plot(solnInfo.GoalTreeData(:,1),solnInfo.GoalTreeData(:,2),'.-','color','g') % Goal tree expansio
plot(pthObj.States(:,1),pthObj.States(:,2),'r-','LineWidth',2) % draw path
```



References

[1] Kuffner, J. J., and S. M. LaValle. “RRT-Connect: An Efficient Approach to Single-Query Path Planning.” In *Proceedings 2000 ICRA. Millennium Conference. IEEE International Conference on Robotics and Automation. Symposia Proceedings (Cat. No.00CH37065)*, 2:995–1001. San Francisco, CA, USA: IEEE, 2000. https://doi:10.1109/ROBOT.2000.844730.

See Also

Objects
`plannerRRT` | `plannerRRTStar`

Functions
`plan` | `copy`

Introduced in R2021a

copy

Create deep copy of planner object

Syntax

```
plannerCopy = copy(planner)
```

Description

`plannerCopy = copy(planner)` creates a deep copy of the planner object with the same properties.

Input Arguments

`planner` — Path planner

`plannerBiRRT` object

Path planner, specified as a `plannerBiRRT` object.

Output Arguments

`plannerCopy` — Copy of path planner

`plannerBiRRT` object

Copy of path planner, returned as a `plannerBiRRT` object.

See Also

Objects

`plannerBiRRT`

Functions

`plan`

Introduced in R2021a

plan

Plan path between two states

Syntax

```
path = plan(planner,startState,goalState)
[path,solnInfo] = plan(planner,startState,goalState)
```

Description

`path = plan(planner,startState,goalState)` returns a bidirectional rapidly exploring random tree (RRT) path from the start state to the goal state as a `navPath` object.

`[path,solnInfo] = plan(planner,startState,goalState)` also returns the solution information from path planning.

Examples

Plan Path Between Two States Using Bidirectional RRT

Use the `plannerBiRRT` object to plan a path between two states in an environment with obstacles. Visualize the planned path with interpolated states.

Create a state space.

```
ss = stateSpaceSE2;
```

Create an `occupancyMap`-based state validator using the created state space.

```
sv = validatorOccupancyMap(ss);
```

Create an occupancy map from an example map and set map resolution as 10 cells per meter.

```
load exampleMaps
map = occupancyMap(ternaryMap,10);
sv.Map = map;
```

Set the validation distance for the validator.

```
sv.ValidationDistance = 0.01;
```

Update the state space bounds to be the same as the map limits.

```
ss.StateBounds = [map.XWorldLimits; map.YWorldLimits; [-pi pi]];
```

Create the path planner and increase the maximum connection distance.

```
planner = plannerBiRRT(ss,sv);
planner.MaxConnectionDistance = 0.3;
```

Specify the start and goal states.

```
start = [20 10 0];
goal = [40 40 0];
```

Plan a path. Due to the randomness of the RRT algorithm, set the `rng` seed for repeatability.

```
rng(100,'twister')
[pthObj,solnInfo] = plan(planner,start,goal);
```

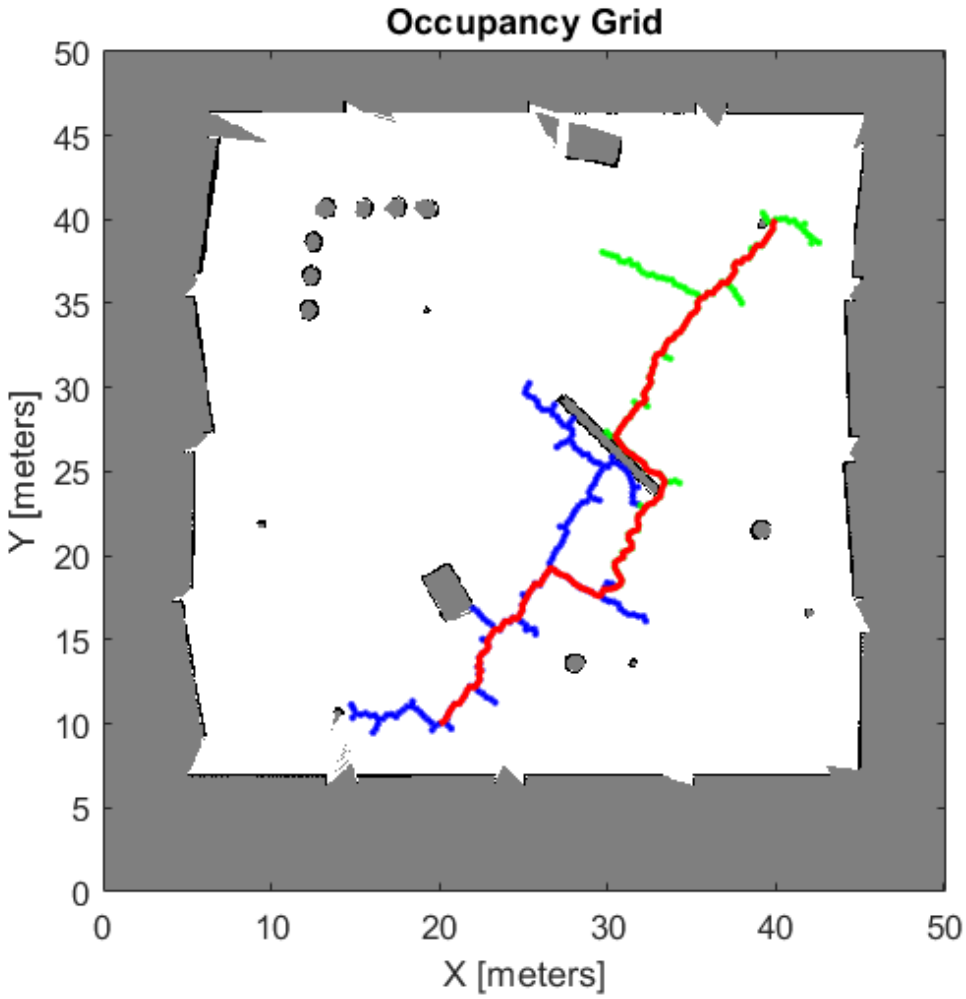
Display the number of iterations taken for the tree to converge.

```
fprintf('Number of iterations: %d\n',solnInfo.NumIterations)
```

```
Number of iterations: 346
```

Visualize the results.

```
show(map)
hold on
plot(solnInfo.StartTreeData(:,1),solnInfo.StartTreeData(:,2),'.-','color','b') % Start tree expa
plot(solnInfo.GoalTreeData(:,1),solnInfo.GoalTreeData(:,2),'.-','color','g') % Goal tree expansi
plot(pthObj.States(:,1),pthObj.States(:,2),'r-','LineWidth',2) % draw path
hold off
```



Replan the path with the `EnableConnectHeuristic` property set to true.

```
planner.EnableConnectHeuristic = true;
[pthObj,solnInfo] = plan(planner,start,goal);
```

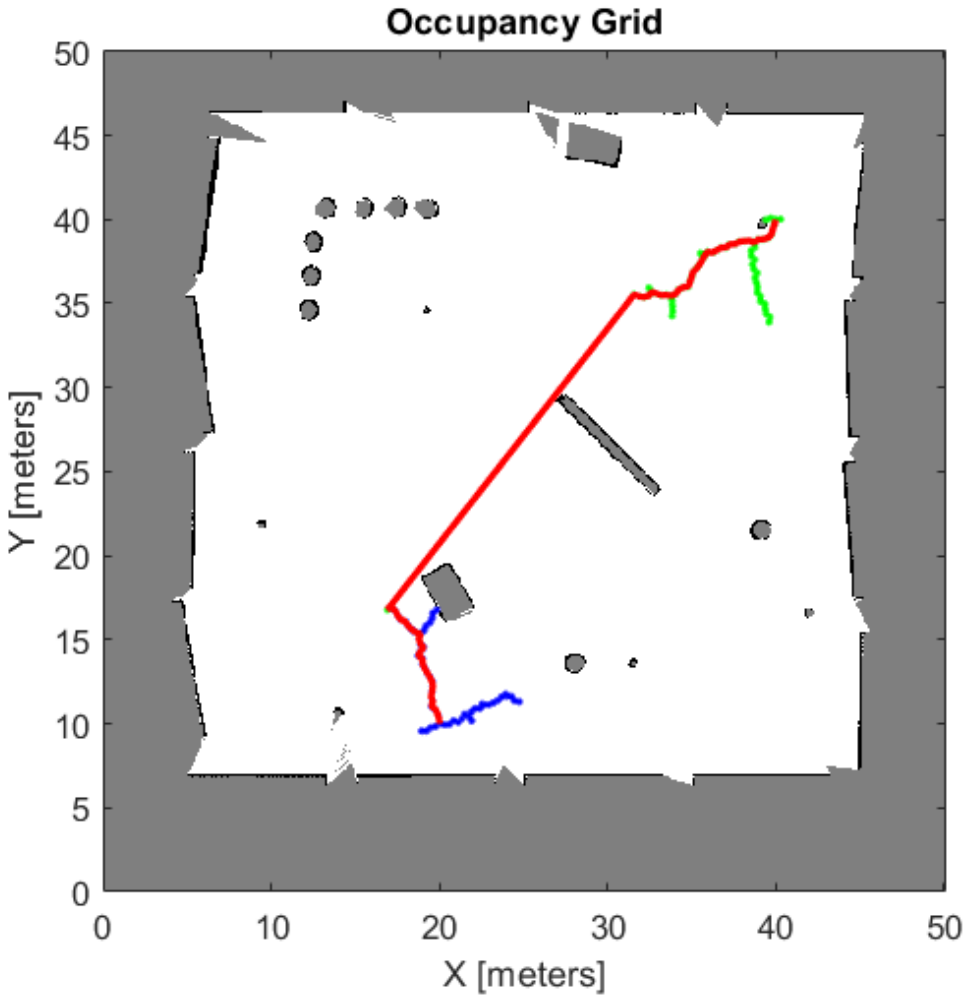
Display the number of iterations taken for the tree to converge. Observe that the planner requires significantly fewer iterations compared to when the `EnableConnectHeuristic` property is set to false.

```
fprintf('Number of iterations: %d\n',solnInfo.NumIterations)
```

```
Number of iterations: 135
```

Visualize the results.

```
figure
show(map)
hold on
plot(solnInfo.StartTreeData(:,1),solnInfo.StartTreeData(:,2),'.-','color','b') % Start tree expan
plot(solnInfo.GoalTreeData(:,1),solnInfo.GoalTreeData(:,2),'.-','color','g') % Goal tree expansio
plot(pthObj.States(:,1),pthObj.States(:,2),'r-','LineWidth',2) % draw path
```



Input Arguments

`planner` — Path planner

`plannerBiRRT` object

Path planner, specified as a `plannerBiRRT` object.

`startState` — Start state of path

N-element real-valued vector

Start state of the path, specified as an *N*-element real-valued vector. *N* is the number of dimensions in the state space.

Example: `[1 1 pi/6]`

Data Types: `single` | `double`

`goalState` — Goal state of path

N-element real-valued vector

Goal state of the path, specified as an *N*-element real-valued vector. *N* is the number of dimensions in the state space.

Example: `[2 2 pi/3]`

Data Types: `single` | `double`

Output Arguments

`path` — Planned path information

`navPath` object

Planned path information, returned as a `navPath` object.

`solnInfo` — Solution Information

structure

Solution Information, returned as a structure. The structure contains these fields:

Field	Description
`IsPathFound`	Indicates whether a path is found. It returns `1` (`true`) if a path is found. Otherwise, it returns `0` (`false`).
`ExitFlag`	Indicates the termination cause of the planner, returned as: • `1` — The planner reaches the goal. • `2` — The planner reaches the maximum number of iterations. • `3` — The planner reaches the maximum number of nodes.

Field	Description
`StartTreeNumNodes`	Number of nodes in the start search tree when the planner terminates, excluding the root node.
`GoalTreeNumNodes`	Number of nodes in the goal search tree when the planner terminates, excluding the root node.
`NumIterations`	Number of combined iterations by both the start tree and goal tree.
`StartTreeData`	Collection of explored states that reflect the status of the start search tree when the planner terminates. Note that `NaN` values are inserted as delimiters to separate each individual edge.
`GoalTreeData`	Collection of explored states that reflect the status of the goal search tree when the planner terminates. Note that `NaN` values are inserted as delimiters to separate each individual edge.

Data Types: `structure`

See Also

Objects
`plannerBiRRT` | `navPath`

Functions
`copy`

Introduced in R2021a

plannerControlRRT

Control-based RRT planner

Description

The `plannerControlRRT` object is a rapidly exploring random tree (RRT) planner for solving kinematic and dynamic (kinodynamic) planning problems using controls. The RRT algorithm is a tree-based motion planning routine that incrementally grows a search tree. In kinematic planners, the tree grows by randomly sampling states in system configuration space, and then attempts to propagate the nearest node toward that state. The state propagator samples controls for reaching the state based on the kinematic model and control policies. As the tree adds nodes, the sampled states span the search space and eventually connect the start and goal states.

These are the control-based RRT algorithm steps:

- Planner, `plannerControlRRT`, requests a state from the state space.
- Planner finds the nearest state in the search tree based on cost.
- State propagator, `mobileRobotPropagator`, samples control commands and durations to propagate toward the target state.
- State propagator propagates toward the target state.
- If the propagator returns a valid trajectory to the state, then add the state to the tree.
- **Optional**: Attempt to direct trajectory toward final goal based on NumGoalExtension and GoalBias properties.
- Continue searching until the search tree reaches the goal or satisfies other exit criteria.

The benefit of a kinodynamic planner like `plannerControlRRT` is that it is guaranteed to return a sequence of states, controls, and references which comprise a kinematically or dynamically feasibly path. The drawback to a kinodynamic planner is that the kinematic propagations cannot guarantee that new states are exactly equal to the target states unless there exists and analytic representation for a sequence of controls that drive the system between two configurations with zero residual error. This means that kinodynamic planners are typically asymptotically complete and guarantee kinematic feasibility, but often can not guarantee asymptotic optimality.

Creation

Syntax

```
controlPlanner = plannerControlRRT(propagator)
controlPlanner = plannerControlRRT(propagator,Name=Value)
```

Description

`controlPlanner = plannerControlRRT(propagator)` creates a kinodynamic RRT planner from a state propagator object and sets the `StatePropagator` property.

`controlPlanner = plannerControlRRT(propagator,Name=Value)` specifies additional properties using name-value arguments. For example, `plannerControlRRT(propagator,ContinueAfterGoalReached=1)` continues to search for alternative paths after the tree first reaches the goal.

Properties

`StatePropagator` — State propagator
`mobileRobotPropagator` object (default) | object of subclass of `nav.StatePropagator`

Mobile robot state propagator, specified as a `mobileRobotPropagator` object or an object of a subclass of `nav.StatePropagator`.

`ContinueAfterGoalReached` — Optimization after reaching goal
`false` or `0` (default) | `true` or `1`

Optimization after reaching the goal, specified as a logical `0` (`false`) or `1` (`true`). If specified as true, the planner continues to search for alternative paths after it first reaches the goal. The planner terminates regardless of the value of this property if it reaches the maximum number of iterations or maximum number of tree nodes.

Data Types: `logical`

`MaxPlanningTime` — Maximum time allowed for planning
`Inf` (default) | positive scalar in seconds

Maximum time allowed for planning, specified as a positive scalar in seconds.

Data Types: `single` | `double`

`MaxNumTreeNodes` — Maximum number of nodes in search tree
`1e4` (default) | positive integer

Maximum number of nodes in the search tree, excluding the root node, specified as a positive integer.

Data Types: `single` | `double`

`MaxNumIteration` — Maximum number of iterations
`1e4` (default) | positive integer

Maximum number of iterations, specified as a positive integer.

Data Types: `single` | `double`

`NumGoalExtension` — Number of times to propagate towards goal
`1` (default) | positive integer

The maximum number of times the planner can propagate towards the goal, specified as a positive integer. After successfully adding a new node to the tree, the planner attempts to propagate the new node toward the goal using the `propagateWhileValid` object function of the state propagator. The planner continues propagating until the function returns an empty state vector indicating that no valid control is found, the planner reaches the goal, or the function has been called `NumGoalExtension` times.

To turn this behavior off, set the property to `0`. Turning this behavior off will result in propagating randomly instead of toward the goal.

Data Types: `single` | `double`

`GoalBias` — Probability of choosing goal state during state sampling
`0.1` (default) | real scalar in range [0, 1]

Probability of choosing the goal state during state sampling, specified as a real scalar in the range [0, 1]. This property determines the likelihood of the planner choosing the actual goal state when randomly selecting states from the state space. You can start by setting the probability to a small value, such as `0.05`.

Data Types: `single` | `double`

`GoalReachedFcn` — Callback function to determine whether goal is reached
`@plannerControlRRT.GoalReachedDefault` | function handle

Callback function to determine whether the goal is reached, specified as a function handle. You can create your own goal-reached function. The function must follow this syntax:

```
function isReached = myGoalReachedFcn(planner,currentState,goalState),
```

where:

- `planner` — is the created planner object, specified as a `plannerControlRRT` object.
- `currentState` — is the current state, specified as a *s*-element real vector. *s* is the number of state variables in the state space.
- `goalState` — is the goal state, specified as a *s*-element real vector. *s* is the number of state variables in the state space.
- `isReached` — is a boolean that indicates whether the current state has reached the goal state, returned as `true` or `false`.

Data Types: `function handle`

Object Functions

plan Plan path from start to goal state

Examples

Plan Kinodynamic Path with Controls for Mobile Robot

Plan control paths for a bicycle kinematic model with the `mobileRobotPropagator` object. Specify a map for the environment, set state bounds, and define a start and goal location. Plan a path using the control-based RRT algorithm, which uses a state propagator for planning motion and the required control commands.

Set State and State Propagator Parameters

Load a ternary map matrix and create an `occupancyMap` object. Create the state propagator using the map. By default, the state propagator uses a bicycle kinematic model.

```
load('exampleMaps','ternaryMap')
map = occupancyMap(ternaryMap,10);

propagator = mobileRobotPropagator(Environment=map); % Bicycle model
```

Set the state bounds on the state space based on the map world limits.

```
propagator.StateSpace.StateBounds(1:2,:) = ...
                    [map.XWorldLimits; map.YWorldLimits];
```

Plan Path

Create the path planner from the state propagator.

```
planner = plannerControlRRT(propagator);
```

Specify the start and goal states.

```
start = [10 15 0];
goal  = [40 30 0];
```

Plan a path between the states. For repeatable results, reset the random number generator before planning. The `plan` function outputs a `navPathControl` object, which contains the states, control commands, and durations.

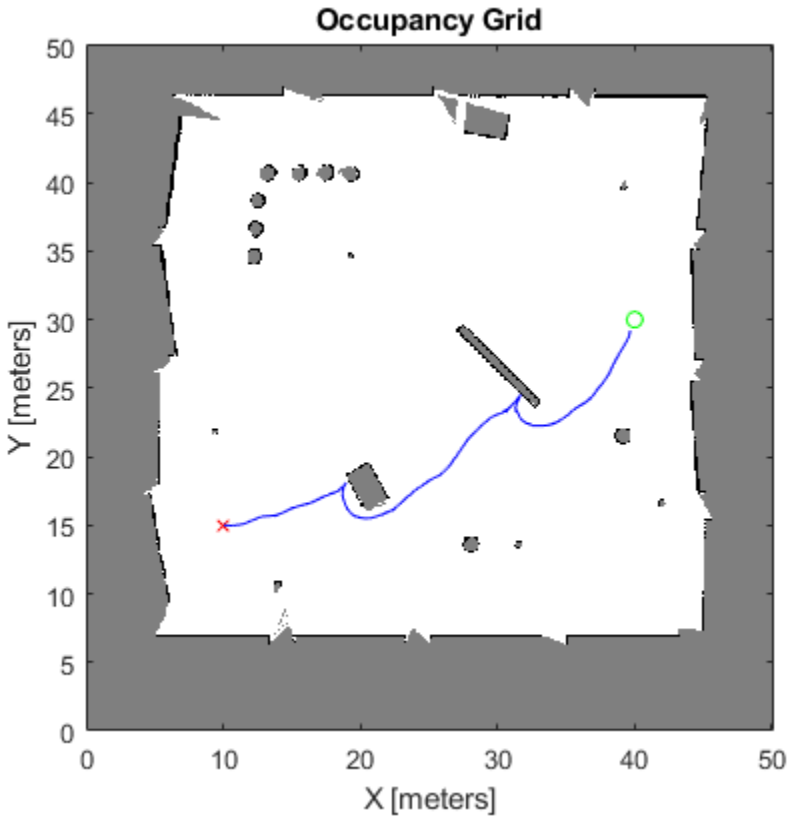
```
rng("default")
path = plan(planner,start,goal)
```

```
path =
  navPathControl with properties:

    StatePropagator: [1x1 mobileRobotPropagator]
             States: [192x3 double]
           Controls: [191x2 double]
          Durations: [191x1 double]
       TargetStates: [191x3 double]
          NumStates: 192
         NumSegments: 191
```

Visualize Results

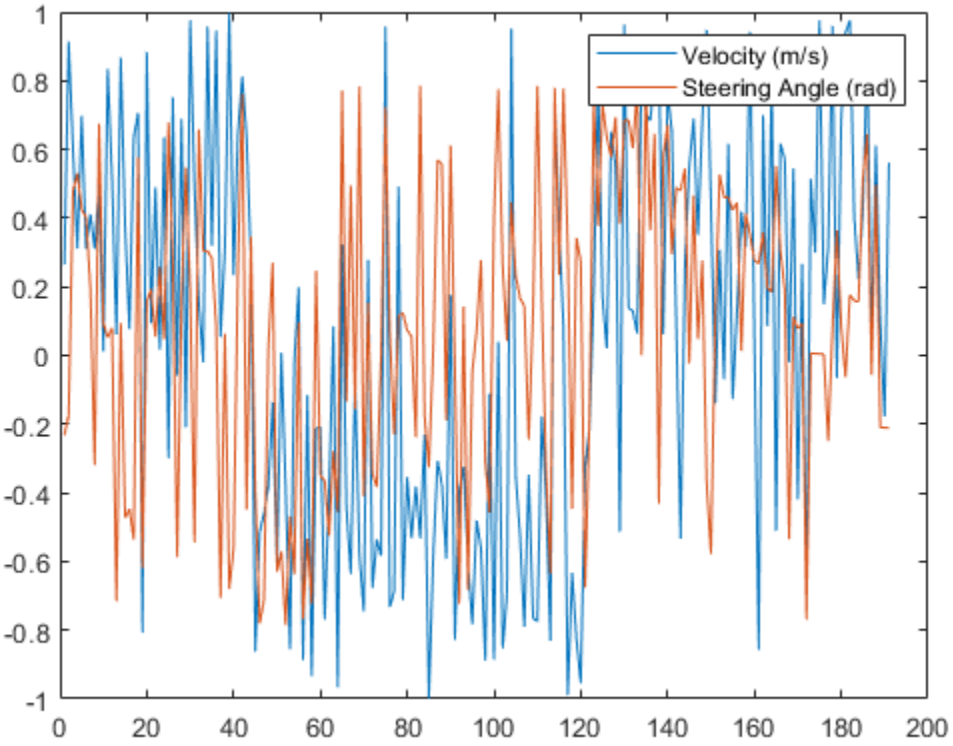
Visualize the map and plot the path states.

```
show(map)
hold on
plot(start(1),start(2),"rx")
plot(goal(1),goal(2),"go")
plot(path.States(:,1),path.States(:,2),"b")
hold off
```



Display the `[v psi]` control inputs of forward velocity and steering angle.

```
plot(path.Controls)
ylim([-1 1])
legend(["Velocity (m/s)","Steering Angle (rad)"])
```



References

[1] S.M. Lavalle, J.J. Kuffner, "Randomized kinodynamic planning", *International Journal of Robotics Research*, vol. 20, no. 5, pp. 378-400, May 2001

[2] Kavraki, L. and S. LaValle. "Chapter 5 Motion Planning", 1st ed., B. Siciliano et O. Khatib, Ed. New York: *Springer-Verlag Berlin Heidelberg*, 2008, pp. 109-131.

See Also

Functions

`plan`

Objects

`plannerAStarGrid` | `plannerBiRRT` | `plannerHybridAStar` | `plannerRRT` | `plannerRRTStar`

Topics

“Reverse-Capable Motion Planning for Tractor-Trailer Model Using plannerControlRRT”

Introduced in R2021b

plan

Plan path from start to goal state

Syntax

```
path = plan(planner,startState,goalState)
[path,solutionInfo] = plan(planner,startState,goalState)
```

Description

`path = plan(planner,startState,goalState)` plans a path, `path`, from the start state to the goal state.

`[path,solutionInfo] = plan(planner,startState,goalState)` also returns the solution information `solutionInfo` of the path planning.

Examples

Plan Path Between Two States

Create a state space.

```
ss = stateSpaceSE2;
```

Create an `occupanyMap`-based state validator using the created state space.

```
sv = validatorOccupancyMap(ss);
```

Create an occupany map from an example map and set map resolution as 10 cells/meter.

```
load exampleMaps
map = occupancyMap(simpleMap,10);
sv.Map = map;
```

Set validation distance for the validator.

```
sv.ValidationDistance = 0.01;
```

Update state space bounds to be the same as map limits.

```
ss.StateBounds = [map.XWorldLimits;map.YWorldLimits; [-pi pi]];
```

Create the path planner and increase max connection distance.

```
planner = plannerRRT(ss,sv);
planner.MaxConnectionDistance = 0.3;
```

Set the start and goal states.

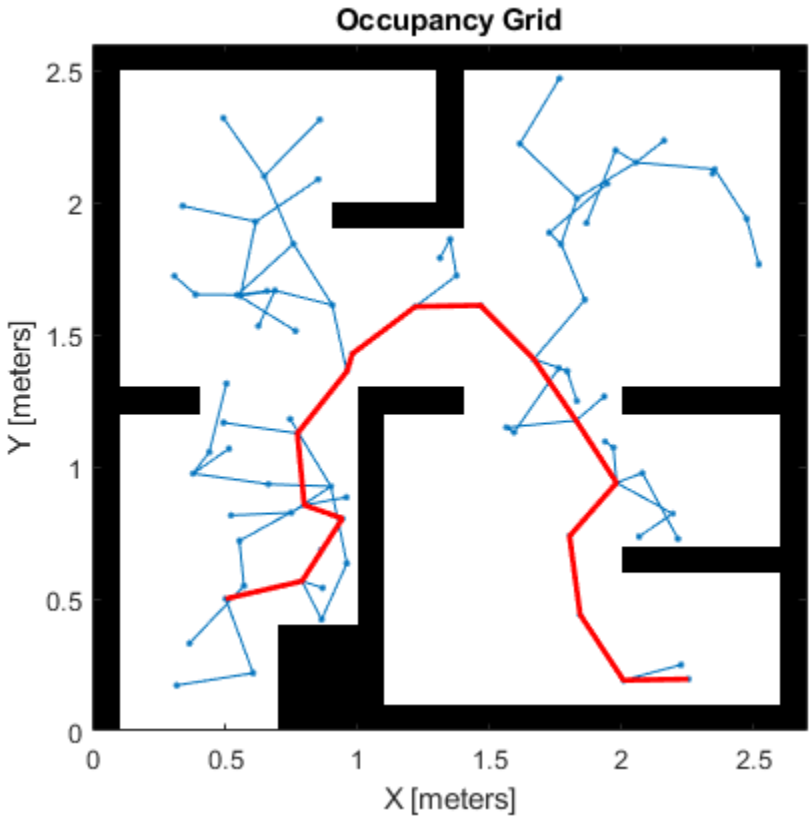
```
start = [0.5,0.5,0];
goal = [2.5,0.2,0];
```

Plan a path with default settings.

```
rng(100,'twister'); % for repeatable result
[pthObj,solnInfo] = plan(planner,start,goal);
```

Visualize the results.

```
show(map)
hold on
plot(solnInfo.TreeData(:,1),solnInfo.TreeData(:,2),'.-'); % tree expansion
plot(pthObj.States(:,1),pthObj.States(:,2),'r-','LineWidth',2) % draw path
```



Input Arguments

`planner` — Path planner

`plannerRRT` object | `plannerRRTStar` object

Path planner, specified as a `plannerRRT` object or a `plannerRRTStar` object.

`startState` — Start state of path

s-element vector

Initial state of the path, specified as an *s*-element vector. *s* is the number of state variables in the state space. For example, a robot in the SE(2) space has a state vector of `[x y theta]`.

Example: `[1 1 pi/6]`

Data Types: `single` | `double`

`goalState` — Goal state of path
s-element vector

Goal state of the path, specified as an *s*-element vector. *s* is the number of state variables in the state space. For example, a robot in the SE(2) space has a state vector of `[x y theta]`.

Example: `[2 2 pi/3]`

Data Types: `single` | `double`

Output Arguments

`path` — Planned path information
`navPathControl` object

Planned path information, returned as a `navPathControl` object.

`solutionInfo` — Solution Information
structure

Solution Information, returned as a structure. The structure contains these fields:

Field	Description
`IsPathFound`	Indicates whether a path is found. It returns as `1` if a path is found. Otherwise, it returns as `0`.
`ExitFlag`	Indicates the terminate status of the planner, returned as one of these options: • `1` — Goal successfully reached • `2` — Exceeded maximum number of iterations • `3` — Exceeded maximum number of nodes • `4` — Exceeded maximum planning time
`NumTreeNode`	Number of nodes in the search tree when the planner terminates excluding the root node.
`NumIterations`	Number of extension routines executed.
`PlanningTime`	Elapsed time while planning, returned as a scalar in seconds.
`TreeInfo`	Collection of explored states that reflects the status of the search tree when the planner terminates. Note that the planner inserts `NaN` values as delimiters to separate each individual edge.

Data Types: `structure`

See Also

Objects
`plannerControlRRT` | `navPathControl` | `plannerRRT`

Introduced in R2021b

plannerHybridAStar

Hybrid A* path planner

Description

The Hybrid A* path planner object generates a smooth path in a given 2-D space for vehicles with nonholonomic constraints.

Note The Hybrid A* planner checks for collisions in the map by interpolating the motion primitives and analytic expansion based on the `ValidationDistance` property of the `stateValidator` object. If the `ValidationDistance` property is set to `Inf`, the object interpolates based on the cell size of the map specified in the state validator. Inflate the occupancy map before assigning it to the planner to account for the vehicle size.

Creation

Syntax

```
planner = plannerHybridAStar(validator)
planner = plannerHybridAStar(validator,Name,Value)
```

Description

`planner = plannerHybridAStar(validator)` creates a path planner object using the Hybrid A* algorithm. Specify the `validator` input as a `validatorOccupancyMap` or `validatorVehicleCostmap` object. The `validator` input sets the value of the "StateValidator" on page 2-0 property.

`planner = plannerHybridAStar(validator,Name,Value)` sets "Properties" on page 2-969 of the path planner by using one or more name-value pair arguments. Enclose each property name inside single quotes (' ').

Properties

`StateValidator` — State validator for planning
`validatorOccupancyMap` object | `validatorVehicleCostmap` object

State validator for planning, specified either as a `validatorOccupancyMap` or `validatorVehicleCostmap` object based on SE(2) state space.

`MotionPrimitiveLength` — Length of motion primitives to be generated
`ceil(sqrt(2)*map_CellSize)` (default) | positive scalar

Length of motion primitives to be generated, specified as the comma-separated pair consisting of `'MotionPrimitiveLength'` and a positive scalar in meters. Increase the length for large maps or sparse environments. Decrease the length for dense environments.

Note `'MotionPrimitiveLength'` cannot exceed one-fourth the length of the circumference of a circle based on the `'MinTurningRadius'`.

Data Types: `double`

`MinTurningRadius` — Minimum turning radius of vehicle
`(2*motion_primitive_length)/pi` (default) | positive scalar

Minimum turning radius of vehicle, specified as the comma-separated pair consisting of `'MinTurningRadius'` and a positive scalar in meters.

Note The value of `'MinTurningRadius'` is set such that the `'MotionPrimitiveLength'` cannot exceed one-fourth the length of the circumference of a circle based on it.

Data Types: `double`

`NumMotionPrimitives` — Number of motion primitives to be generated
5 (default) | positive odd integer scalar greater than or equal to 3

Number of motion primitives to be generated, specified as the comma-separated pair consisting of `'NumMotionPrimitives'` and a positive odd integer scalar greater than or equal to 3.

`ForwardCost` — Cost multiplier to travel in forward direction
1 (default) | positive scalar

Cost multiplier to travel in forward direction, specified as the comma-separated pair consisting of `'ForwardCost'` and a positive scalar. Increase the cost value to penalize forward motion.

Data Types: `double`

`ReverseCost` — Cost multiplier to travel in reverse direction
3 (default) | positive scalar

Cost multiplier to travel in reverse direction, specified as the comma-separated pair consisting of `'ReverseCost'` and a positive scalar. Increase the cost value to penalize reverse motion.

Data Types: `double`

`DirectionSwitchingCost` — Additive cost for switching direction of motion
0 (default) | positive scalar

Additive cost for switching direction of motion, specified as the comma-separated pair consisting of `'DirectionSwitchingCost'` and a positive scalar. Increase the cost value to penalize direction switching.

Data Types: `double`

`AnalyticExpansionInterval` — Interval for attempting analytic expansion from lowest cost node available
5 (default) | positive integer scalar

Interval for attempting analytic expansion from the lowest cost node available at that instance, specified as the comma-separated pair consisting of `'AnalyticExpansionInterval'` and a positive integer scalar.

The Hybrid A* path planner expands the motion primitives from the nodes with the lowest cost available at that instance:

- The number of nodes to be expanded depends upon the number of primitives to be generated in both the direction and their validity, the cycle repeats until `'AnalyticExpansionInterval'` is reached.
- The planner then attempts an analytic expansion to reach the goal pose from the tree using a Reeds-Shepp model. If the attempt fails, the planner repeats the cycle.

Improve the algorithm performance by reducing the interval to increase the number of checks for a Reeds-Shepp connection to the final goal.

`InterpolationDistance` — Distance between interpolated poses in output path
1 (default) | positive scalar

Distance between interpolated poses in output path, specified as the comma-separated pair consisting of `'InterpolationDistance'` and a positive scalar in meters.

Data Types: `double`

Object Functions

plan	Find obstacle-free path between two poses
show	Visualize the planned path

Examples

Obstacle-Free Path Planning Using Hybrid A Star

Plan a collision-free path for a vehicle through a parking lot by using the Hybrid A* algorithm.

Create and Assign Map to State Validator

Load the cost values of cells in the vehicle costmap of a parking lot.

```
load parkingLotCostVal.mat % costVal
```

Create a `binaryOccupancyMap` with cost values.

```
map = binaryOccupancyMap(costVal);
```

Create a state validator object for collision checking.

```
validator = validatorOccupancyMap;
```

Assign the map to the state validator object.

```
validator.Map = map;
```

Plan and Visualize Path

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

```
planner = plannerHybridAStar(validator,'MinTurningRadius',4,'MotionPrimitiveLength',6);
```

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

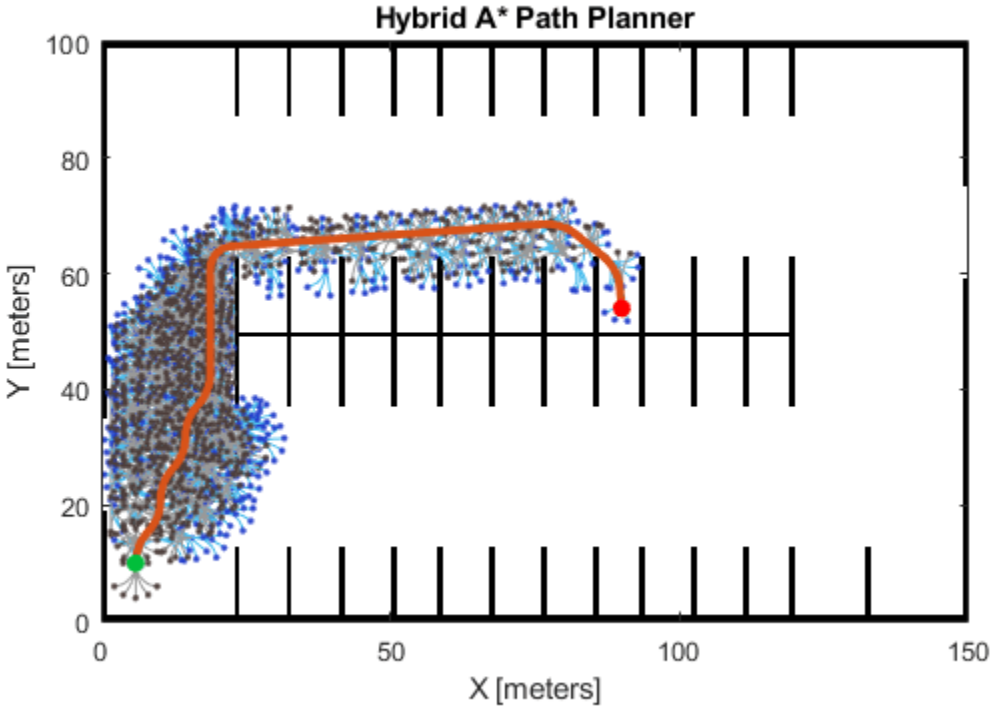
```
startPose = [6 10 pi/2]; % [meters, meters, radians]
goalPose = [90 54 -pi/2];
```

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

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

Visualize the path using show function.

```
show(planner)
```



References

[1] Dolgov, Dmitri, Sebastian Thrun, Michael Montemerlo, and James Diebel. *Practical Search Techniques in Path Planning for Autonomous Driving*. American Association for Artificial Intelligence, 2008.

[2] Petereit, Janko, Thomas Emter, Christian W. Frey, Thomas Kopfstedt, and Andreas Beutel. "Application of Hybrid A* to an Autonomous Mobile Robot for Path Planning in Unstructured Outdoor Environments." *ROBOTIK 2012: 7th German Conference on Robotics*. 2012, pp. 1-6.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`validatorOccupancyMap` | `validatorVehicleCostmap` | `navPath`

Topics

Introduced in R2019b

plan

Find obstacle-free path between two poses

Syntax

```
path = plan(planner,start,goal)
[path,directions] = plan(planner,start,goal)
[path,directions,solutionInfo] = plan(planner,start,goal)
```

Description

`path = plan(planner,start,goal)` computes an obstacle-free path between start and goal poses, specified as `[x y theta]` vectors, using the input `plannerHybridAStar` object.

`[path,directions] = plan(planner,start,goal)` also returns the direction of motion for each pose along the path, `directions`, as a column vector. A value of `1` indicates forward direction and a value of `-1` indicates reverse direction. The function returns an empty column vector when the planner is unable to find a path.

`[path,directions,solutionInfo] = plan(planner,start,goal)` also returns `solutionInfo` that contains the solution information of the path planning as a structure.

Examples

Obstacle-Free Path Planning Using Hybrid A Star

Plan a collision-free path for a vehicle through a parking lot by using the Hybrid A* algorithm.

Create and Assign Map to State Validator

Load the cost values of cells in the vehicle costmap of a parking lot.

```
load parkingLotCostVal.mat % costVal
```

Create a `binaryOccupancyMap` with cost values.

```
map = binaryOccupancyMap(costVal);
```

Create a state validator object for collision checking.

```
validator = validatorOccupancyMap;
```

Assign the map to the state validator object.

```
validator.Map = map;
```

Plan and Visualize Path

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

```
planner = plannerHybridAStar(validator,'MinTurningRadius',4,'MotionPrimitiveLength',6);
```

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

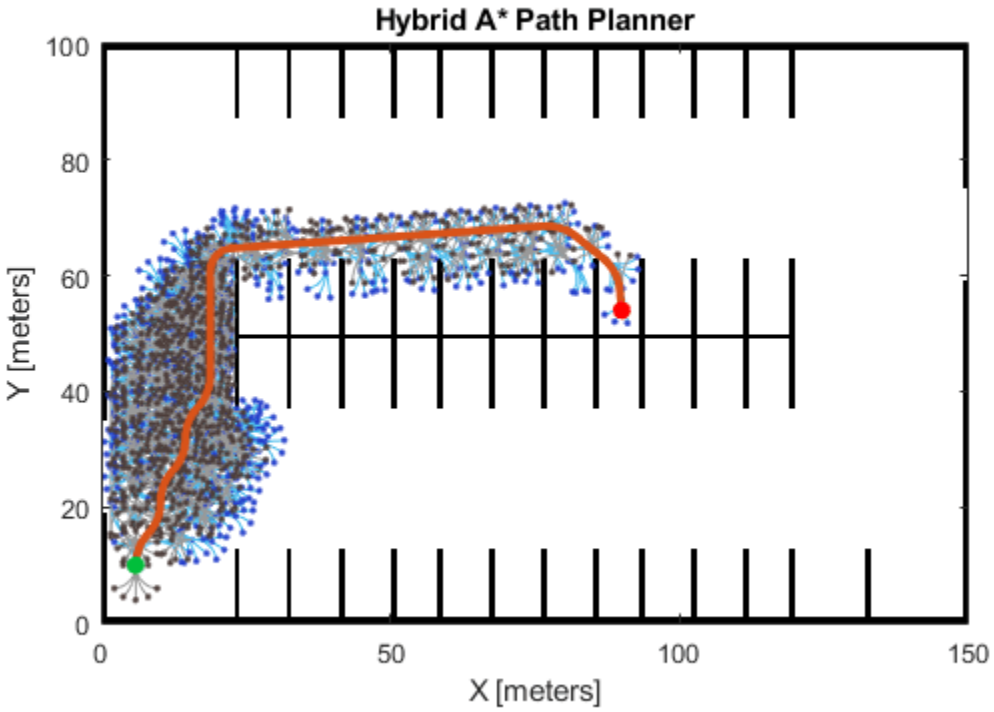
```
startPose = [6 10 pi/2]; % [meters, meters, radians]
goalPose = [90 54 -pi/2];
```

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

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

Visualize the path using show function.

```
show(planner)
```



Input Arguments

`planner` — Hybrid A* path planner
`plannerHybridAStar` object

Hybrid A* path planner, specified as a `plannerHybridAStar` object.

`start` — Start location of path
three-element vector

Start location of path, specified as a 1-by-3 vector in the form `[x y theta]`. *x* and *y* specify the position in meters, and *theta* specifies the orientation angle in radians.

Example: `[5 5 pi/2]`

Data Types: `double`

`goal` — Final location of path

three-element vector

Final location of path, specified as a 1-by-3 vector in the form `[x y theta]`. *x* and *y* specify the position in meters, and *theta* specifies the orientation angle in radians.

Example: `[45 45 pi/4]`

Data Types: `double`

Output Arguments

`path` — Obstacle-free path

`navPath` object

Obstacle-free path, returned as a `navPath` object.

`directions` — Directions of motion

column vector of 1s (forward) and −1s (reverse)

Direction of motion for each pose along the path, returned as a column vector of 1s (forward) and −1s (reverse).

Data Types: `double`

`solutionInfo` — Solution Information

structure

Solution Information, returned as a structure. The fields of the structure are:

Fields of `solutionInfo`

Fields	Description
`IsPathFound`	Indicates whether a path is found. It returns as `1` if a path is found. Otherwise, it returns `0`.
`NumNodes`	Number of nodes in the search tree when the planner terminates (excluding the root node).
`NumIterations`	Number of planning iterations executed.

Data Types: `struct`

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`validatorOccupancyMap` | `validatorVehicleCostmap` | `navPath`

Introduced in R2019b

show

Visualize the planned path

Syntax

```
show(planner)
show(planner,Name,Value)
axHandle = show(planner)
```

Description

`show(planner)` plots the Hybrid A* expansion tree and the planned path in the map.

`show(planner,Name,Value)` specifies additional options using one or more name-value pair arguments.

`axHandle = show(planner)` outputs the axes handle of the figure used to plot the path.

Examples

Obstacle-Free Path Planning Using Hybrid A Star

Plan a collision-free path for a vehicle through a parking lot by using the Hybrid A* algorithm.

Create and Assign Map to State Validator

Load the cost values of cells in the vehicle costmap of a parking lot.

```
load parkingLotCostVal.mat % costVal
```

Create a `binaryOccupancyMap` with cost values.

```
map = binaryOccupancyMap(costVal);
```

Create a state validator object for collision checking.

```
validator = validatorOccupancyMap;
```

Assign the map to the state validator object.

```
validator.Map = map;
```

Plan and Visualize Path

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

```
planner = plannerHybridAStar(validator,'MinTurningRadius',4,'MotionPrimitiveLength',6);
```

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

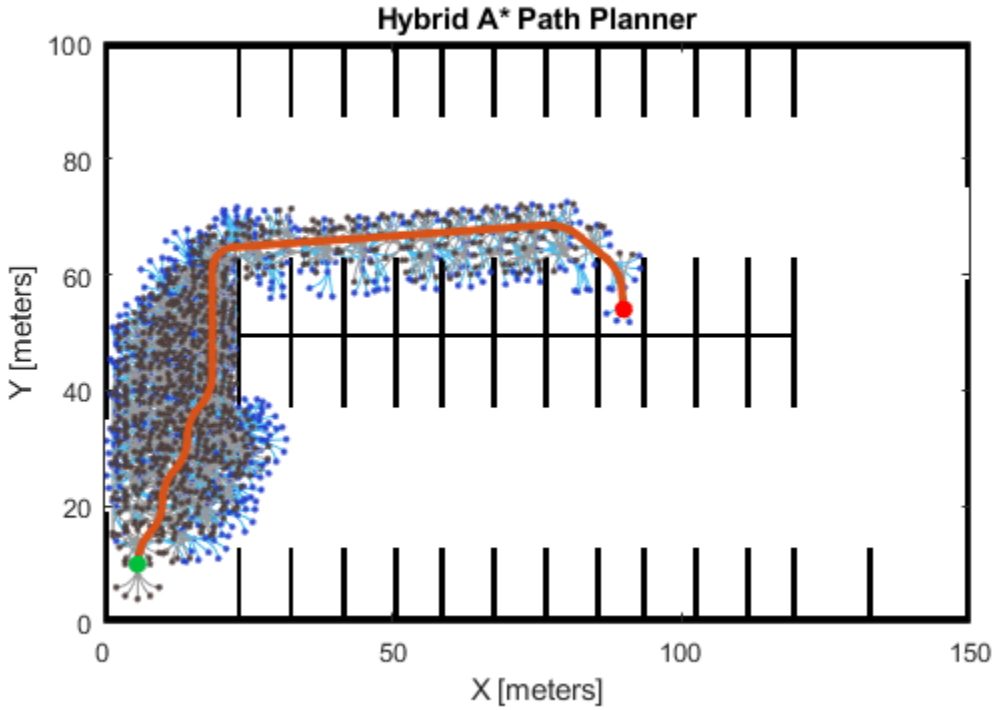
```
startPose = [6 10 pi/2]; % [meters, meters, radians]
goalPose = [90 54 -pi/2];
```

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

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

Visualize the path using show function.

```
show(planner)
```



Input Arguments

`planner` — Hybrid A* path planner

`plannerHybridAStar` object

Hybrid A* path planner, specified as a `plannerHybridAStar` object.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'Positions','none'`

`Parent` — Axes used to plot path
`Axes` object | `UIAxes` object

Axes used to plot path, specified as the comma-separated pair consisting of `'Parent'` and either an `axes` or `uiaxes` object. If you do not specify `'Parent'`, a new figure is created.

`Tree` — Display expansion tree
`'on'` (default) | `'off'`

Display expansion tree option, specified as the comma-separated pair consisting of `'Tree'` and either `'on'` or `'off'`.

Example: `show(planner,'Tree','off')`

Data Types: `string`

`Path` — Display planned path
`'on'` (default) | `'off'`

Display planned path option, specified as the comma-separated pair consisting of `'Path'` and either `'on'` or `'off'`.

Example: `show(planner,'Path','off')`

Data Types: `string`

`Positions` — Display start and goal points
`'both'` (default) | `'start'` | `'goal'` | `'none'`

Display the start and goal points, specified as the comma-separated pair consisting of `'Positions'` and one of the following:

- `'start'` — Display the start point.
- `'goal'` — Display the goal point.
- `'both'` — Display the start and goal points.
- `'none'` — Do not display any points.

Example: `show(planner,'Positions','start')`

Data Types: `string`

Output Arguments

`axHandle` — Axes used to plot path
`Axes` object | `UIAxes` object

Axes used to plot path, returned as either an `axes` or `uiaxes` object.

See Also
`validatorOccupancyMap` | `validatorVehicleCostmap` | `navPath`

Introduced in R2019b

plannerRRT

Create an RRT planner for geometric planning

Description

The `plannerRRT` object creates a rapidly-exploring random tree (RRT) planner for solving geometric planning problems. RRT is a tree-based motion planner that builds a search tree incrementally from samples randomly drawn from a given state space. The tree eventually spans the search space and connects the start state to the goal state. The general tree growing process is as follows:

1. The planner samples a random state x_{rand} in the state space.
2. The planner finds a state x_{near} that is already in the search tree and is closest (based on the distance definition in the state space) to x_{rand}.
3. The planner expands from x_{near} towards x_{rand}, until a state x_{new} is reached.
4. Then new state x_{new} is added to the search tree.

For geometric RRT, the expansion and connection between two states can be found analytically without violating the constraints specified in the state space of the planner object.

Creation

Syntax

```
planner = plannerRRT(stateSpace,stateVal)
```

Description

`planner = plannerRRT(stateSpace,stateVal)` creates an RRT planner from a state space object, `stateSpace`, and a state validator object, `stateVal`. The state space of `stateVal` must be the same as `stateSpace`. `stateSpace` and `stateVal` also sets the `StateSpace` and `StateValidator` properties of the `planner`.

Properties

`StateSpace` — State space for the planner

state space object

State space for the planner, specified as a state space object. You can use state space objects such as `stateSpaceSE2`, `stateSpaceDubins`, and `stateSpaceReedsShepp`. You can also customize a state space object using the `nav.StateSpace` object.

`StateValidator` — State validator for the planner

state validator object

State validator for the planner, specified as a state validator object. You can use state validator objects such as `validatorOccupancyMap` and `validatorVehicleCostmap`.

`MaxNumTreeNodes` — Maximum number of nodes in the search tree
`1e4` (default) | positive integer

Maximum number of nodes in the search tree (excluding the root node), specified as a positive integer.

Data Types: `single` | `double`

`MaxIterations` — Maximum number of iterations
`1e4` (default) | positive integer

Maximum number of iterations, specified as a positive integer.

Data Types: `single` | `double`

`MaxConnectionDistance` — Maximum length of motion
`0.1` (default) | positive scalar

Maximum length of a motion allowed in the tree, specified as a scalar.

Data Types: `single` | `double`

`GoalReachedFcn` — Callback function to evaluate whether goal is reached
`@nav.algs.checkIfGoalIsReached` | function handle

Callback function to evaluate whether the goal is reached, specified as a function handle. You can create your own goal reached function. The function must follow this syntax:

```
function isReached = myGoalReachedFcn(planner,currentState,goalState)
```

where:

- `planner` — The created planner object, specified as `plannerRRT` object.
- `currentState` — The current state, specified as a three element real vector.
- `goalState` — The goal state, specified as a three element real vector.
- `isReached` — A boolean variable to indicate whether the current state has reached the goal state, returned as `true` or `false`.

To use custom `GoalReachedFcn` in code generation workflow, this property must be set to a custom function handle before calling the plan function and it cannot be changed after initialization.

Data Types: `function handle`

`GoalBias` — Probability of choosing goal state during state sampling
`0.05` (default) | real scalar in [0,1]

Probability of choosing the goal state during state sampling, specified as a real scalar in [0,1]. The property defines the probability of choosing the actual goal state during the process of randomly selecting states from the state space. You can start by setting the probability to a small value such as `0.05`.

Data Types: `single` | `double`

Object Functions

plan Plan path between two states

copy Create copy of planner object

Examples

Plan Path Between Two States

Create a state space.

```
ss = stateSpaceSE2;
```

Create an `occupanyMap`-based state validator using the created state space.

```
sv = validatorOccupancyMap(ss);
```

Create an occupany map from an example map and set map resolution as 10 cells/meter.

```
load exampleMaps
map = occupancyMap(simpleMap,10);
sv.Map = map;
```

Set validation distance for the validator.

```
sv.ValidationDistance = 0.01;
```

Update state space bounds to be the same as map limits.

```
ss.StateBounds = [map.XWorldLimits;map.YWorldLimits; [-pi pi]];
```

Create the path planner and increase max connection distance.

```
planner = plannerRRT(ss,sv);
planner.MaxConnectionDistance = 0.3;
```

Set the start and goal states.

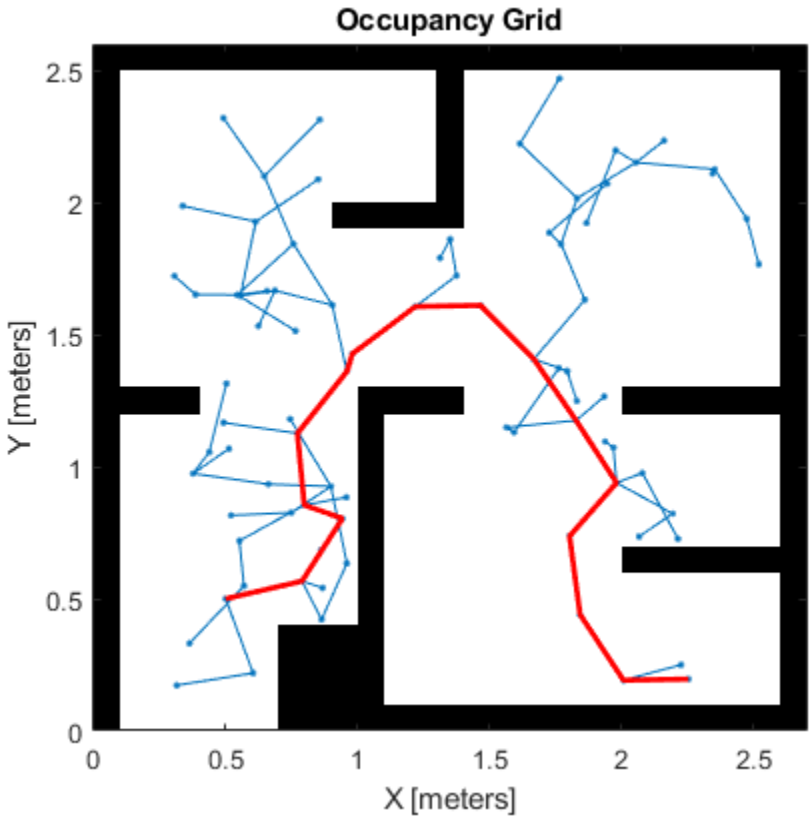
```
start = [0.5,0.5,0];
goal = [2.5,0.2,0];
```

Plan a path with default settings.

```
rng(100,'twister'); % for repeatable result
[pthObj,solnInfo] = plan(planner,start,goal);
```

Visualize the results.

```
show(map)
hold on
plot(solnInfo.TreeData(:,1),solnInfo.TreeData(:,2),'.-'); % tree expansion
plot(pthObj.States(:,1),pthObj.States(:,2),'r-','LineWidth',2) % draw path
```



References

[1] S.M. Lavalle and J.J. Kuffner. "Randomized Kinodynamic Planning." *The International Journal of Robotics Research*. Vol. 20, Number 5, 2001, pp. 378 – 400.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- To use custom GoalReachedFcn in code generation workflow, this property must be set to a custom function handle before calling the plan function and it cannot be changed after initialization.

See Also

`navPath` | `plannerRRTStar` | `stateSpaceReedsShepp` | `stateSpaceDubins` | `stateSpaceSE2`

Introduced in R2019b

plannerRRTStar

Create an optimal RRT path planner (RRT*)

Description

The `plannerRRTStar` object creates an asymptotically-optimal RRT planner, RRT*. The RRT* algorithm converges to an optimal solution in terms of the state space distance. Also, its runtime is a constant factor of the runtime of the RRT algorithm. RRT* is used to solve geometric planning problems. A geometric planning problem requires that any two random states drawn from the state space can be connected.

Creation

Syntax

```
planner = plannerRRTStar(stateSpace,stateVal)
```

Description

`planner = plannerRRTStar(stateSpace,stateVal)` creates an RRT* planner from a state space object, `stateSpace`, and a state validator object, `stateVal`. The state space of `stateVal` must be the same as `stateSpace`. `stateSpace` and `stateVal` also sets the `StateSpace` and `StateValidator` properties of the `planner` object.

Properties

`BallRadiusConstant` — Constant used to estimate the near neighbors search radius
`100` (default) | positive scalar

Constant used to estimate the near neighbors search radius, specified as a positive scalar. With a larger ball radius, the searching radius reduces slower as the number of nodes in the tree increases. The radius is estimated as following:

$$r = \min\left(\left\{\frac{\gamma \ln(n)}{nV_d}\right\}^{1/d}, \eta\right)$$

where:

- d — Dimension of the state space
- n — Number of nodes in the search tree
- η — The value of the `MaxConnectionDistance` property
- V_d — Volume of the unit ball in the dth dimension

Data Types: `single` | `double`

`ContinueAfterGoalReached` — Continue to optimize after goal is reached
`false` (default) | `true`

Decide if the planner continues to optimize after the goal is reached, specified as `false` or `true`. The planner also terminates regardless of the value of this property if the maximum number of iterations or maximum number of tree nodes is reached.

Data Types: `logical`

`StateSpace` — State space for the planner
state space object

State space for the planner, specified as a state space object. You can use state space objects such as `stateSpaceSE2`, `stateSpaceDubins`, and `stateSpaceReedsShepp`. You can also customize a state space object using the `nav.StateSpace` object.

`StateValidator` — State validator for the planner
state validator object

State validator for the planner, specified as a state validator object. You can use state validator objects such as `validatorOccupancyMap` and `validatorVehicleCostmap`.

`MaxNumTreeNodes` — Maximum number of nodes in the search tree
`1e4` (default) | positive integer

Maximum number of nodes in the search tree (excluding the root node), specified as a positive integer.

Data Types: `single` | `double`

`MaxIterations` — Maximum number of iterations
`1e4` (default) | positive integer

Maximum number of iterations, specified as a positive integer.

Data Types: `single` | `double`

`MaxConnectionDistance` — Maximum length of motion
`0.1` (default) | positive scalar

Maximum length of a motion allowed in the tree, specified as a scalar.

Data Types: `single` | `double`

`GoalReachedFcn` — Callback function to determine whether goal is reached
`@nav.algs.checkIfGoalIsReached` | function handle

Callback function to determine whether the goal is reached, specified as a function handle. You can create your own goal reached function. The function must follow this syntax:

```
function isReached = myGoalReachedFcn(planner,currentState,goalState)
```

where:

- `planner` — The created planner object, specified as `plannerRRTStar` object.
- `currentState` — The current state, specified as a three element real vector.
- `goalState` — The goal state, specified as a three element real vector.
- `isReached` — A boolean variable to indicate whether the current state has reached the goal state, returned as `true` or `false`.

To use custom `GoalReachedFcn` in code generation workflow, this property must be set to a custom function handle before calling the plan function and it cannot be changed after initialization.

Data Types: `function handle`

`GoalBias` — Probability of choosing goal state during state sampling
`0.05` (default) | real scalar in [0,1]

Probability of choosing the goal state during state sampling, specified as a real scalar in [0,1]. The property defines the probability of choosing the actual goal state during the process of randomly selecting states from the state space. You can start by setting the probability to a small value such as `0.05`.

Data Types: `single` | `double`

Object Functions

plan Plan path between two states
copy Create copy of planner object

Examples

Plan Optimal Path Between Two States

Create a state space.

```
ss = stateSpaceSE2;
```

Create a `occupancyMap`-based state validator using the created state space.

```
sv = validatorOccupancyMap(ss);
```

Create an occupany map from an example map and set map resolution as 10 cells/meter.

```
load exampleMaps.mat
map = occupancyMap(simpleMap, 10);
sv.Map = map;
```

Set validation distance for the validator.

```
sv.ValidationDistance = 0.01;
```

Update state space bounds to be the same as map limits.

```
ss.StateBounds = [map.XWorldLimits; map.YWorldLimits; [-pi pi]];
```

Create RRT* path planner and allow further optimization after goal is reached.

```
planner = plannerRRTStar(ss,sv);
planner.ContinueAfterGoalReached = true;
```

Reduce max iterations and increase max connection distance.

```
planner.MaxIterations = 2500;
planner.MaxConnectionDistance = 0.3;
```

Set the start and goal states.

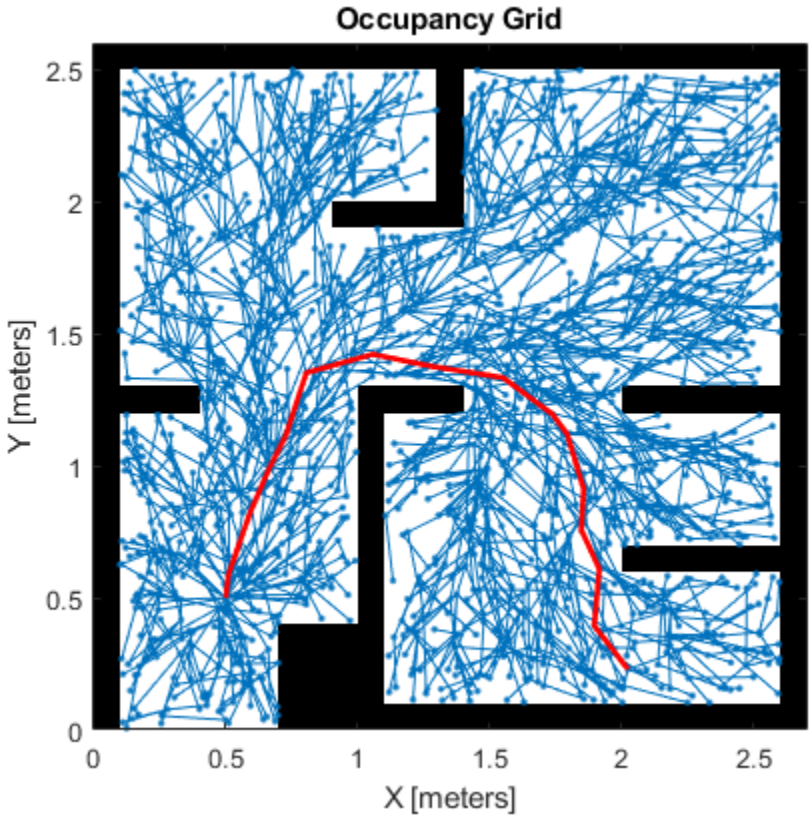
```
start = [0.5, 0.5 0];
goal = [2.5, 0.2, 0];
```

Plan a path with default settings.

```
rng(100, 'twister') % repeatable result
[pthObj, solnInfo] = plan(planner,start,goal);
```

Visualize the results.

```
map.show;
hold on;
plot(solnInfo.TreeData(:,1),solnInfo.TreeData(:,2), '.-'); % tree expansion
plot(pthObj.States(:,1),pthObj.States(:,2),'r-','LineWidth',2); % draw path
```



References

[1] Karaman, S. and E. Frazzoli. "Sampling-Based Algorithms for Optimal Motion Planning." *The International Journal of Robotics Research*. Vol. 30, Number 7, 2011, pp 846 – 894.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- To use custom GoalReachedFcn in code generation workflow, this property must be set to a custom function handle before calling the plan function and it cannot be changed after initialization.

See Also

`navPath` | `plannerRRT` | `stateSpaceSE2` | `stateSpaceReedsShepp` | `stateSpaceDubins`

Introduced in R2019b

plan

Plan path between two states

Syntax

```
path = plan(planner,startState,goalState)
[path,solutionInfo] = plan(planner,startState,goalState)
```

Description

`path = plan(planner,startState,goalState)` returns a `path` from the start state to the goal state.

`[path,solutionInfo] = plan(planner,startState,goalState)` also returns `solInfo` that contains the solution information of the path planning.

Examples

Plan Path Between Two States

Create a state space.

```
ss = stateSpaceSE2;
```

Create an `occupanyMap`-based state validator using the created state space.

```
sv = validatorOccupancyMap(ss);
```

Create an occupany map from an example map and set map resolution as 10 cells/meter.

```
load exampleMaps
map = occupancyMap(simpleMap,10);
sv.Map = map;
```

Set validation distance for the validator.

```
sv.ValidationDistance = 0.01;
```

Update state space bounds to be the same as map limits.

```
ss.StateBounds = [map.XWorldLimits;map.YWorldLimits; [-pi pi]];
```

Create the path planner and increase max connection distance.

```
planner = plannerRRT(ss,sv);
planner.MaxConnectionDistance = 0.3;
```

Set the start and goal states.

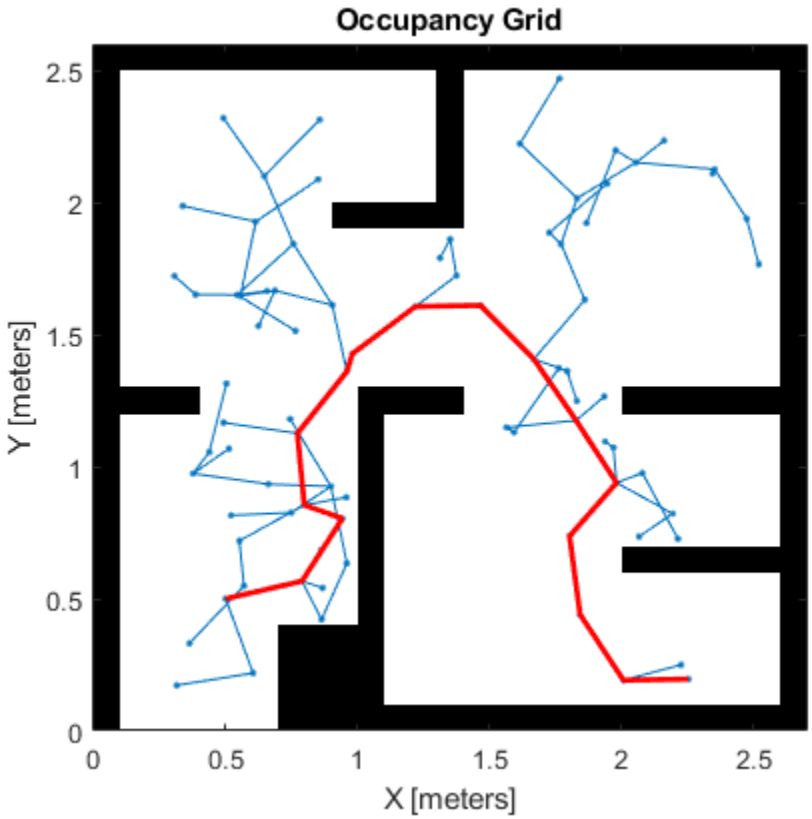
```
start = [0.5,0.5,0];
goal = [2.5,0.2,0];
```

Plan a path with default settings.

```
rng(100,'twister'); % for repeatable result
[pthObj,solnInfo] = plan(planner,start,goal);
```

Visualize the results.

```
show(map)
hold on
plot(solnInfo.TreeData(:,1),solnInfo.TreeData(:,2),'.-'); % tree expansion
plot(pthObj.States(:,1),pthObj.States(:,2),'r-','LineWidth',2) % draw path
```



Input Arguments

`planner` — Path planner

`plannerRRT` object | `plannerRRTStar` object

Path planner, specified as a `plannerRRT` object or a `plannerRRTStar` object.

`startState` — Start state of the path

N-element real-valued vector

Start state of the path, specified as an *N*-element real-valued vector. *N* is the dimension of the state space.

Example: `[1 1 pi/6]`

Data Types: `single` | `double`

`goalState` — Goal state of the path
N-element real-valued vector

Goal state of the path, specified as an *N*-element real-valued vector. *N* is the dimension of the state space.

Example: `[2 2 pi/3]`

Data Types: `single` | `double`

Output Arguments

`path` — Object that holds planned path information
`navPath` object

An object that holds the planned path information, returned as a `navPath` object.

`solutionInfo` — Solution Information
structure

Solution Information, returned as a structure. The fields of the structure are:

Fields of `solutionInfo`

Fields	Description
`IsPathFound`	Indicates whether a path is found. It returns as `1` if a path is found. Otherwise, it returns `0`.
`ExitFlag`	Indicates the terminate status of the planner, returned as • 1 — if the goal is reached • 2 — if the maximum number of iterations is reached • 3 — if the maximum number of nodes is reached
`NumNodes`	Number of nodes in the search tree when the planner terminates (excluding the root node).
`NumIterations`	Number of "extend" routines executed.
`TreeData`	A collection of explored states that reflects the status of the search tree when planner terminates. Note that `NaN` values are inserted as delimiters to separate each individual edge.
`PathCosts`	Contains the cost of the path at each iteration. Value for iterations when path has not reached the goal is denoted by a `NaN`. Size of the array is `NumIterations`-by-1. Last element contains the cost of the final path. **Note** This field is applicable only for `plannerRRTStar` object.

Data Types: `structure`

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`navPath` | `plannerRRT` | `plannerRRTStar` | `stateSpaceReedsShepp` | `stateSpaceDubins` | `stateSpaceSE2`

Introduced in R2019b

copy

Create copy of planner object

Syntax

```
planner2 = copy(planner1)
```

Description

`planner2 = copy(planner1)` creates a planner object, `planner2`, form a planner object, `planner1`.

Input Arguments

`planner1` — Path planner

`plannerRRT` object | `plannerRRTStar` object

Path planner, specified as a `plannerRRT` object or a `plannerRRTStar` object.

Output Arguments

`planner2` — Path planner

`plannerRRT` object | `plannerRRTStar` object

Path planner, returned as a `plannerRRT` object or a `plannerRRTStar` object.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`navPath` | `plannerRRT` | `plannerRRTStar` | `stateSpaceReedsShepp` | `stateSpaceDubins` | `stateSpaceSE2`

Introduced in R2018b

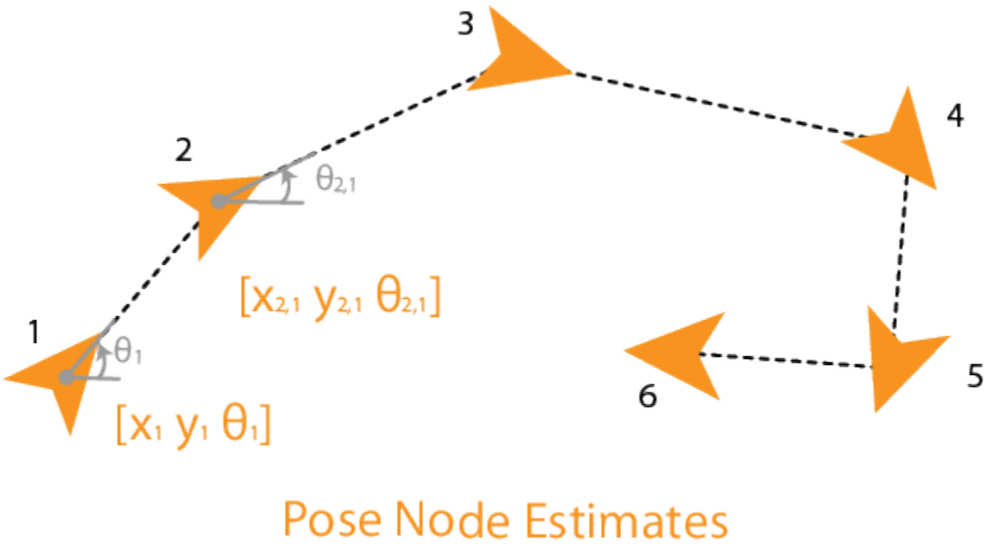
poseGraph

Create 2-D pose graph

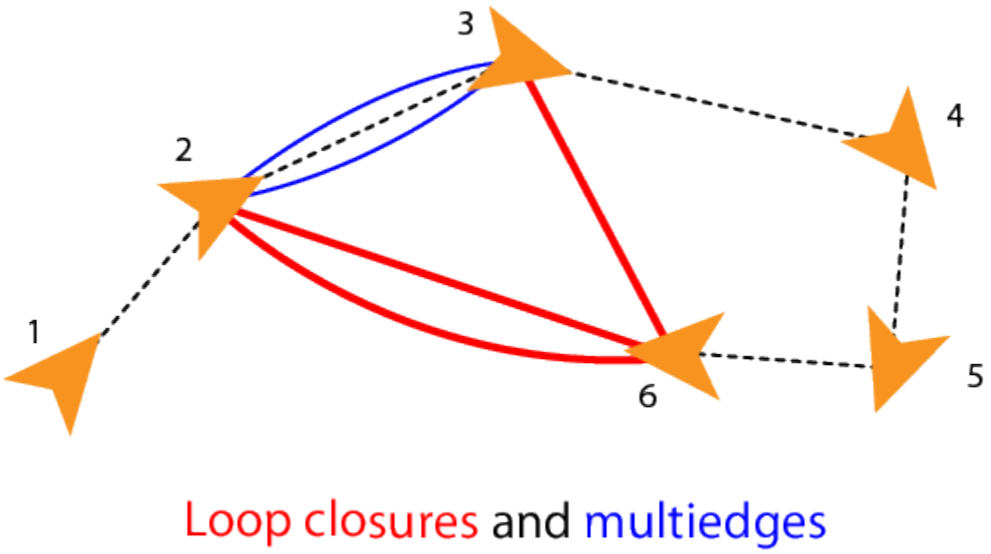
Description

A `poseGraph` object stores information for a 2-D pose graph representation. A pose graph contains nodes connected by edges. Each node estimate is connected to the graph by edge constraints that define the relative pose between nodes and the uncertainty on that measurement.

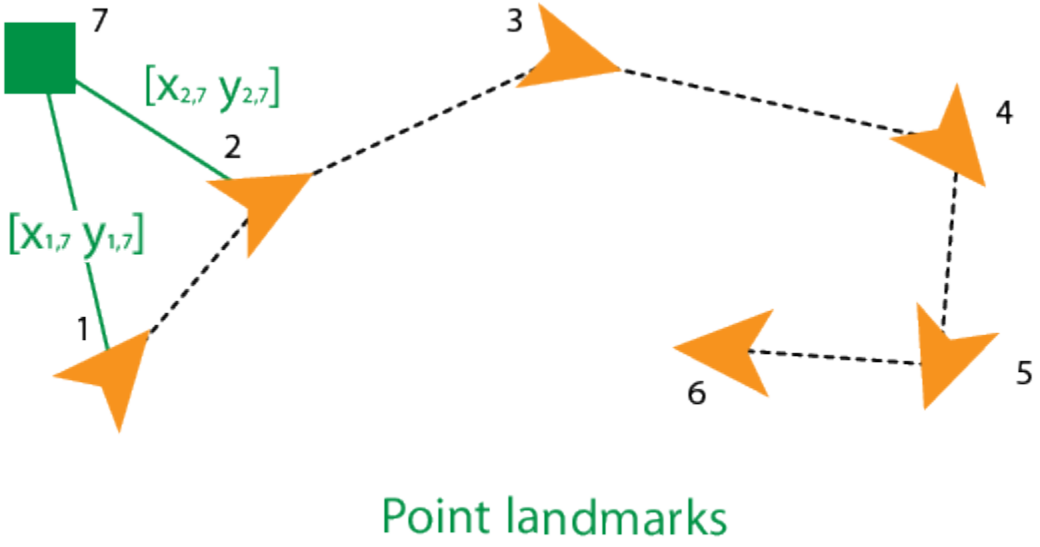
To construct a pose graph iteratively, use the `addRelativePose` function to add relative pose estimates and connect them to an existing node with specified edge constraints. Pose nodes must be specified relative to a pose node. Specify the uncertainty of the measurement using an information matrix.



Adding an edge between two nonsequential nodes creates a *loop closure* in the graph. Multiple edges or *multiedges* between node pairs are also supported, which includes loop closures. To add additional edge constraints or loop closures, specify the node IDs using the `addRelativePose` function. When optimizing the pose graph, the `optimizePoseGraph` function finds a solution to satisfy all these edge constraints.



To add landmark point nodes, use the `addPointLandmark` function. This function specifies nodes as *xy*-points without orientation estimates. Landmarks must be specified relative to a pose node.



The `lidarSLAM` object performs lidar-based simultaneous localization and mapping, which is based around the optimization of a 2-D pose graph.

For 3-D pose graphs, see the `poseGraph3D` object or the “Landmark SLAM Using AprilTag Markers” example.

Creation

Syntax

```
poseGraph = poseGraph
poseGraph = poseGraph('MaxNumEdges',maxEdges,'MaxNumNodes',maxNodes)
```

Description

`poseGraph = poseGraph` creates a 2-D pose graph object. Add poses using `addRelativePose` to construct a pose graph iteratively.

`poseGraph = poseGraph('MaxNumEdges',maxEdges,'MaxNumNodes',maxNodes)` specifies an upper bound on the number of edges and nodes allowed in the pose graph when generating code. This syntax is only required when generating code.

Properties

`NumNodes` — Number of nodes in pose graph
`1` (default) | positive integer

This property is read-only.

Number of nodes in pose graph, specified as a positive integer. Each node represents a pose measurement or a point landmark measurement. To specify relative poses between nodes, use `addRelativePose`. To specify a landmark pose, use `addLandmarkPose`. To get a list of all nodes, use `edgeNodePairs`.

`NumEdges` — Number of edges in pose graph
`0` (default) | nonnegative integer

This property is read-only.

Number of edges in pose graph, specified as a nonnegative integer. Each edge connects two nodes in the pose graph. Loop closure edges and landmark edges are included.

`NumLoopClosureEdges` — Number of loop closures
`0` (default) | nonnegative integer

This property is read-only.

Number of loop closures in pose graph, specified as a nonnegative integer. To get the edge IDs of the loop closures, use the `LoopClosureEdgeIDs` property.

`LoopClosureEdgeIDs` — Loop closure edge IDs
vector

This property is read-only.

Loop closure edges IDs, specified as a vector of edge IDs.

`LandmarkNodeIDs` — Landmark node IDs
vector

This property is read-only.

Landmark node IDs, specified as a vector of IDs for each node.

Object Functions

addPointLandmark	Add landmark point node to pose graph
addRelativePose	Add relative pose to pose graph
copy	Create copy of pose graph
edgeNodePairs	Edge node pairs in pose graph
edgeConstraints	Edge constraints in pose graph
edgeResidualErrors	Compute pose graph edge residual errors
findEdgeID	Find edge ID of edge
nodeEstimates	Poses of nodes in pose graph
removeEdges	Remove loop closure edges from graph
show	Plot pose graph

Examples

Optimize a 2-D Pose Graph

Optimize a pose graph based on the nodes and edge constraints. The pose graph used in this example is from the Intel Research Lab Dataset and was generated from collecting wheel odometry and a laser range finder sensor information in an indoor lab.

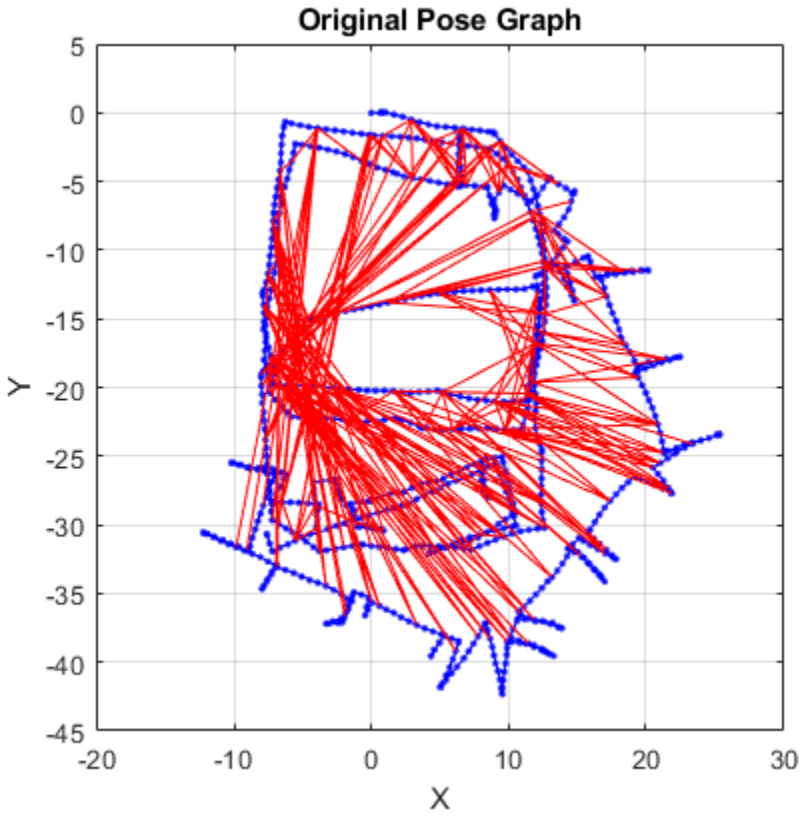
Load the Intel data set that contains a 2-D pose graph. Inspect the `poseGraph` object to view the number of nodes and loop closures.

```
load intel-2d-posegraph.mat pg
disp(pg)
```

```
  poseGraph with properties:

              NumNodes: 1228
              NumEdges: 1483
    NumLoopClosureEdges: 256
     LoopClosureEdgeIDs: [1228 1229 1230 1231 1232 1233 1234 1235 1236 ... ]
        LandmarkNodeIDs: [1x0 double]
```

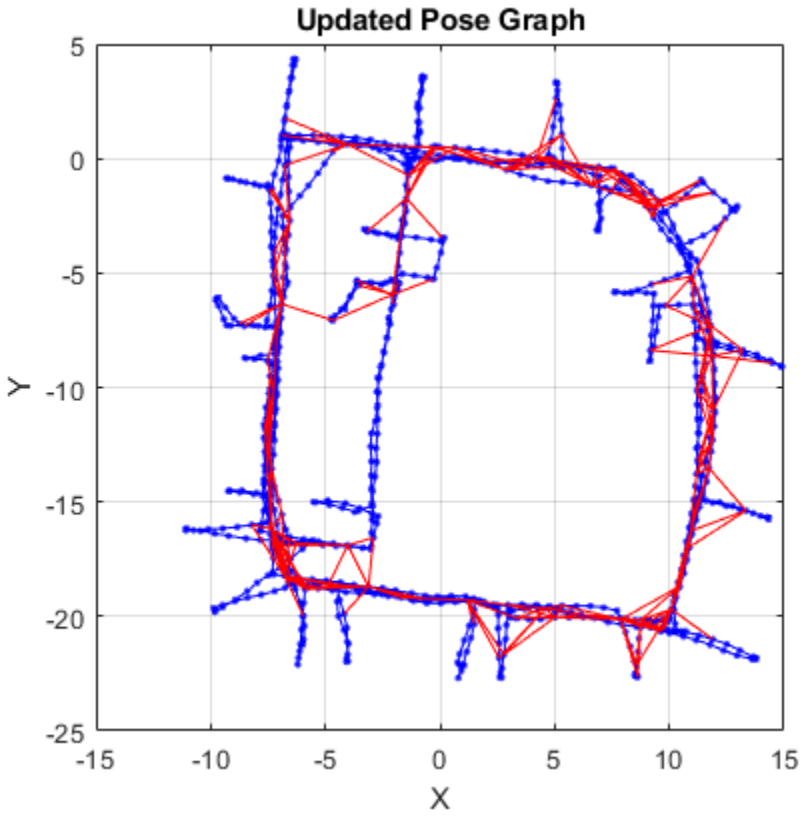
Plot the pose graph with IDs off. Red lines indicate loop closures identified in the dataset.

```
show(pg,'IDs','off');
title('Original Pose Graph')
```



Optimize the pose graph. Nodes are adjusted based on the edge constraints and loop closures. Plot the optimized pose graph to see the adjustment of the nodes with loop closures.

```
updatedPG = optimizePoseGraph(pg);
figure
show(updatedPG,'IDs','off');
title('Updated Pose Graph')
```



References

[1] Grisetti, G., R. Kummerle, C. Stachniss, and W. Burgard. "A Tutorial on Graph-Based SLAM." *IEEE Intelligent Transportation Systems Magazine*. Vol. 2, No. 4, 2010, pp. 31–43. doi:10.1109/mits.2010.939925.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Use this syntax when constructing `poseGraph` objects for code generation: `poseGraph = poseGraph('MaxNumEdges',maxEdges,'MaxNumNodes',maxNodes)` specifies an upper bound on the number of edges and nodes allowed in the pose graph when generating code. This limit is only required when generating code.

See Also

Functions

`optimizePoseGraph` | `addRelativePose` | `addPointLandmark` | `show`

Objects

`lidarSLAM` | `poseGraph3D`

Topics
“Implement Simultaneous Localization And Mapping (SLAM) with Lidar Scans”
“Implement Online Simultaneous Localization And Mapping (SLAM) with Lidar Scans”
“Landmark SLAM Using AprilTag Markers”

Introduced in R2019b

addPointLandmark

Add landmark point node to pose graph

Syntax

```
addPointLandmark(poseGraph,measurement)
addPointLandmark(poseGraph,measurement,infoMat)
addPointLandmark(poseGraph,measurement,infoMat,poseNodeID)
addPointLandmark(poseGraph,measurement,infoMat,poseNodeID,pointNodeID)
[nodePair,edgeID] = addPointLandmark( ___ )
```

Description

`addPointLandmark(poseGraph,measurement)` adds a landmark point node, based on the input position `measurement` that connects to the last pose node in the pose graph. To add pose measurement nodes, see the `addRelativePose` function.

`addPointLandmark(poseGraph,measurement,infoMat)` also specifies the information matrix as part of the edge constraint, which represents the uncertainty of the landmark measurement.

`addPointLandmark(poseGraph,measurement,infoMat,poseNodeID)` adds a new landmark point node and connects it to the pose node specified by `poseNodeID`.

`addPointLandmark(poseGraph,measurement,infoMat,poseNodeID,pointNodeID)` creates an edge by specifying a point measurement between existing nodes, specified by `poseNodeID` and `pointNodeID`. If the node pair already exists, the function appends the new measurement.

`[nodePair,edgeID] = addPointLandmark( ___ )` returns the newly added edge and edge ID using any combination of inputs from the previous syntaxes.

Input Arguments

`poseGraph` — Pose graph

`poseGraph` object | `poseGraph3D` object

Pose graph, specified as a `poseGraph` or `poseGraph3D` object.

`measurement` — Position of landmark point

two-element vector of form `[x y]` | three-element vector of form `[x y z]`

Position of landmark point, specified as one of the following:

For `poseGraph` (2-D), the pose is a two-element vector of form of the form `[x y]`, which defines an *xy*-position for the landmark.

For `poseGraph3D`, the pose is a three-element vector of the form `[x y z]`, which defines an *xyz*-position for the landmark.

`infoMat` — Information matrix for landmark

three-element vector | six-element vector

Information matrix for the landmark, specified as a three-element or six-element vector.

Each vector is the compact form of the upper triangle of the square information matrix. An information matrix represents the uncertainty of the measurement. The matrix is calculated as the inverse of the covariance. If the measurement is an `[x y]` vector, the covariance matrix is a 2-by-2 matrix of pairwise covariance calculations. Typically, the uncertainty is determined by the sensor model.

For `poseGraph` (2-D), each information matrix is a three-element vector. The default is `[1 1 0]`.

For `poseGraph3D`, each information matrix is a six-element vector. The default is `[1 0 0 1 0 1]`.

`poseNodeID` — Pose node to attach from
positive integer

Pose node to attach from, specified as a positive integer. This integer corresponds to the node ID of a pose node in `poseGraph`. When specified without the `pointNodeID` input, `addPointLandmark` creates a new landmark point node and adds an edge between the new node and the `poseNodeID` node.

`pointNodeID` — Landmark point node to attach to
positive integer

Landmark point node to attach to, specified as a positive integer. This integer corresponds to the ID of a landmark node in the pose graph. See the LandmarkNodeIDs property of the pose graph.

Output Arguments

`nodePair` — Edge node pair in pose graph
two-element vector

Edge node pair in the pose graph, returned as a two-element vector that lists the IDs of the two nodes that the edge connects. Multiple edges may exist between the same pair of nodes.

`edgeID` — ID of added edge
positive integer

ID of added edge, returned as a positive integer.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

Use this syntax when constructing `poseGraph` or `poseGraph3D` objects for code generation:

`poseGraph = poseGraph('MaxNumEdges',maxEdges,'MaxNumNodes',maxNodes)` specifies an upper bound on the number of edges and nodes allowed in the pose graph when generating code. This limit is only required when generating code.

```
poseGraph = poseGraph3D('MaxNumEdges',maxEdges,'MaxNumNodes',maxNodes)
```

See Also

Functions
`optimizePoseGraph` | `findEdgeID` | `edgeNodePairs` | `edgeConstraints` | `nodeEstimates` | `removeEdges`

Objects
`poseGraph` | `poseGraph3D` | `lidarSLAM`

Topics
“Implement Simultaneous Localization And Mapping (SLAM) with Lidar Scans”
“Implement Online Simultaneous Localization And Mapping (SLAM) with Lidar Scans”
“Landmark SLAM Using AprilTag Markers”

Introduced in R2021a

addRelativePose

Add relative pose to pose graph

Syntax

```
addRelativePose(poseGraph,measurement)
addRelativePose(poseGraph,measurement,infoMat)
addRelativePose(poseGraph,measurement,infoMat,fromNodeID)
addRelativePose(poseGraph,measurement,infoMat,fromNodeID,toNodeID)
[nodePair,edgeID] = addRelativePose( ___ )
```

Description

`addRelativePose(poseGraph,measurement)` creates a node based on the input `measurement` that connects to the last pose node in the pose graph. To add landmark nodes, see the `addPointLandmark` function.

`addRelativePose(poseGraph,measurement,infoMat)` also specifies the information matrix as part of the edge constraint, which represents the uncertainty of the pose measurement.

`addRelativePose(poseGraph,measurement,infoMat,fromNodeID)` creates a new pose node and connects it to the specific node specified by `fromNodeID`.

`addRelativePose(poseGraph,measurement,infoMat,fromNodeID,toNodeID)` creates an edge by specifying a relative pose measurement between existing nodes specified by `fromNodeID` and `toNodeID`. This edge is called a loop closure. If a loop closure already exists, the function appends the new measurement. Calling the `optimizePoseGraph` function combines multiple appended measurements into a single edge. This syntax does not support adding edges to a landmark node.

`[nodePair,edgeID] = addRelativePose( ___ )` returns the newly added edge and edge ID using any of the previous syntaxes.

Input Arguments

`poseGraph` — Pose graph

`poseGraph` object | `poseGraph3D` object

Pose graph, specified as a `poseGraph` or `poseGraph3D` object.

`measurement` — Relative pose between nodes

`[x y theta]` vector | `[x y z qw qx qy qz]` vector

Relative pose between nodes, specified as one of the following:

For `poseGraph` (2-D), the pose is a `[x y theta]` vector, which defines a *xy*-position and orientation angle, `theta`.

For `poseGraph3D`, the pose is a `[x y z qw qx qy qz]` vector, which defines by an *xyz*-position and quaternion orientation, `[qw qx qy qz]`

Note Many other sources for 3-D pose graphs, including `.g2o` formats, specify the quaternion orientation in a different order, for example, `[qx qy qz qw]`. Check the source of your pose graph data before adding nodes to your `poseGraph3D` object.

`infoMat` — Information matrix
6-element vector | 21-element vector

Information matrices, specified in compact form as a 6-element vector or 21-element vector.

Each row is the upper triangle of the square information matrix. An information matrix represents the uncertainty of the measurement. The matrix is calculated as the inverse of the covariance. If the measurement is an `[x y theta]` vector, the covariance matrix is a 3-by-3 of pairwise covariance calculations. Typically, the uncertainty is determined by the sensor model.

For `poseGraph` (2-D), each information matrix is a six-element vector. The default is `[1 0 0 1 0 1]`. For landmark nodes, the last three elements are returned as `NaN`.

For `poseGraph3D`, each information matrix is a 21-element vector. The default is `[1 0 0 0 0 0 1 0 0 0 0 1 0 0 0 1 0 0 1 0 1]`.

`fromNodeID` — Node to attach from
positive integer

Node to attach from, specified as a positive integer. This integer corresponds to the node ID of a node in `poseGraph`. When specified without `toNodeID`, `addRelativePose` creates a new node and adds an edge between the new node and the `fromNodeID` node.

`toNodeID` — Node to attach to
positive integer

Node to attach to, specified as a positive integer. This integer corresponds to the node ID of a node in `poseGraph`. `addRelativePose` adds an edge between this node and the `fromNodeID` node.

Output Arguments

`nodePair` — Edge node pair in pose graph
two-element vector

Edge node pairs in pose graph, returned as two-element vector that lists the IDs of the two nodes that each edge connects. Multiple edges may exist between the same pair of nodes.

`edgeID` — ID of added edge
positive integer

ID of added edge, returned as a positive integer.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

Use this syntax when constructing `poseGraph` or `poseGraph3D` objects for code generation:

`poseGraph = poseGraph('MaxNumEdges',maxEdges,'MaxNumNodes',maxNodes)` specifies an upper bound on the number of edges and nodes allowed in the pose graph when generating code. This limit is only required when generating code.

`poseGraph = poseGraph3D('MaxNumEdges',maxEdges,'MaxNumNodes',maxNodes)`

See Also

Functions
`optimizePoseGraph` | `findEdgeID` | `edgeNodePairs` | `edgeConstraints` | `nodeEstimates` | `removeEdges`

Objects
`poseGraph` | `poseGraph3D` | `lidarSLAM`

Topics
“Implement Simultaneous Localization And Mapping (SLAM) with Lidar Scans”
“Implement Online Simultaneous Localization And Mapping (SLAM) with Lidar Scans”
“Landmark SLAM Using AprilTag Markers”

Introduced in R2019b

copy

Create copy of pose graph

Syntax

```
poseGraph2 = copy(poseGraph1)
```

Description

`poseGraph2 = copy(poseGraph1)` creates a deep copy of the pose graph object with the same properties.

Examples

Input Arguments

poseGraph1 — Pose graph

`poseGraph` object | `poseGraph3D` object

Pose graph, specified as a `poseGraph` or `poseGraph3D` object.

Output Arguments

poseGraph2 — Copy of pose graph

`poseGraph` object | `poseGraph3D` object

Copy of pose graph, returned as a `poseGraph` or `poseGraph3D` object.

See Also

`poseGraph` | `poseGraph3D`

Introduced in R2019b

edgeConstraints

Edge constraints in pose graph

Syntax

```
measurements = edgeConstraints(poseGraph)
[measurements,infoMats] = edgeConstraints(poseGraph)
[measurements,infoMats] = edgeConstraints(poseGraph,edgeIDs)
```

Description

`measurements = edgeConstraints(poseGraph)` lists all edge constraints in the specified pose graph as a relative pose.

`[measurements,infoMats] = edgeConstraints(poseGraph)` also returns the information matrices for each edge. The information matrix is the inverse of the covariance of the pose measurement.

`[measurements,infoMats] = edgeConstraints(poseGraph,edgeIDs)` returns edge constraints for the specified edge IDs.

Input Arguments

`poseGraph` — Pose graph
`poseGraph` object | `poseGraph3D` object

Pose graph, specified as a `poseGraph` or `poseGraph3D` object.

`edgeIDs` — Edge IDs
vector of positive integers

Edge IDs, specified as a vector of positive integers.

Output Arguments

`measurements` — Measurements between nodes
n-by-3 matrix | *n*-by-7 matrix

Measurements between nodes, returned as an *n*-by-3 matrix or *n*-by-7 matrix.

For `poseGraph` (2-D), each row is an `[x y theta]` vector, which defines the relative *xy*-position and orientation angle, `theta`, of a pose in the graph. For landmark positions, `theta` is returned as `NaN`.

For `poseGraph3D`, each row is an `[x y z qw qx qy qz]` vector, which defines the relative *xyz*-position and quaternion orientation, `[qw qx qy qz]`, of a pose in the graph.

Note Many other sources for 3-D pose graphs, including `.g2o` formats, specify the quaternion orientation in a different order, for example, `[qx qy qz qw]`. Check the source of your pose graph data before adding nodes to your `poseGraph3D` object.

`infoMats` — Information matrices
n-by-6 matrix | *n*-by-21 matrix

Information matrices, specified in compact form as a *n*-by-6 or *n*-by-21 matrix, where *n* is the number of poses in the pose graph.

Each row is the upper triangle of the square information matrix. An information matrix represents the uncertainty of the measurement. The matrix is calculated as the inverse of the covariance. If the measurement is an `[x y theta]` vector, the covariance matrix is a 3-by-3 of pairwise covariance calculations. Typically, the uncertainty is determined by the sensor model.

For `poseGraph` (2-D), each information matrix is a six-element vector. The default is `[1 0 0 1 0 1]`. For landmark nodes, the last three elements are returned as `NaN`.

For `poseGraph3D`, each information matrix is a 21-element vector. The default is `[1 0 0 0 0 0 1 0 0 0 0 1 0 0 0 1 0 0 1 0 1]`.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

Use this syntax when constructing `poseGraph` or `poseGraph3D` objects for code generation:

`poseGraph = poseGraph('MaxNumEdges',maxEdges,'MaxNumNodes',maxNodes)` specifies an upper bound on the number of edges and nodes allowed in the pose graph when generating code. This limit is only required when generating code.

See Also

Functions
`edgeNodePairs` | `optimizePoseGraph` | `addRelativePose` | `findEdgeID` | `nodeEstimates` | `removeEdges`

Objects
`poseGraph` | `poseGraph3D` | `lidarSLAM`

Topics
“Implement Simultaneous Localization And Mapping (SLAM) with Lidar Scans”
“Implement Online Simultaneous Localization And Mapping (SLAM) with Lidar Scans”

Introduced in R2019b

edgeNodePairs

Edge node pairs in pose graph

Syntax

```
nodePairs = edgeNodePairs(poseGraph)
nodePairs = edgeNodePairs(poseGraph,edgeIDs)
```

Description

`nodePairs = edgeNodePairs(poseGraph)` returns all edges in the specified pose graph as a list of node ID pairs. Each row of the `edges` output is a pair of nodes that form an edge. Multiple edges may exist between the same pair of nodes.

`nodePairs = edgeNodePairs(poseGraph,edgeIDs)` returns edges corresponding to the specified edge IDs. Each edge in the pose graph has a unique ID even if the node pairs are the same.

Input Arguments

`poseGraph` — Pose graph

`poseGraph` object | `poseGraph3D` object

Pose graph, specified as a `poseGraph` or `poseGraph3D` object.

`edgeIDs` — Edge IDs

vector of positive integers

Edge IDs, specified as a vector of positive integers.

Output Arguments

`nodePairs` — Edge node pairs in pose graph

n-by-2 matrix

Edge node pairs in pose graph, returned as *n*-by-2 matrix that lists the IDs of the two nodes that each edge connects. Each row is a pair of nodes that form an edge. Multiple edges may exist between the same pair of nodes, so the matrix may contain duplicate entries.

Compatibility Considerations

`edgeNodePairs` was renamed

Behavior change in future release

The `edgeNodePairs` object function was renamed from `edges`. Use `edgeNodePairs` when calling the function.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Use this syntax when constructing `poseGraph` or `poseGraph3D` objects for code generation:

`poseGraph = poseGraph('MaxNumEdges',maxEdges,'MaxNumNodes',maxNodes)` specifies an upper bound on the number of edges and nodes allowed in the pose graph when generating code. This limit is only required when generating code.

See Also

Functions

`optimizePoseGraph` | `addRelativePose` | `findEdgeID` | `edgeConstraints` | `nodeEstimates` | `removeEdges`

Objects

`poseGraph` | `poseGraph3D` | `lidarSLAM`

Topics

“Implement Simultaneous Localization And Mapping (SLAM) with Lidar Scans”
“Implement Online Simultaneous Localization And Mapping (SLAM) with Lidar Scans”

Introduced in R2019b

edgeResidualErrors

Compute pose graph edge residual errors

Syntax

```
resErrorVec = edgeResidualErrors(poseGraphObj)
```

Description

`resErrorVec = edgeResidualErrors(poseGraphObj)` returns the residual errors for each edge in the pose graph with the current pose node estimates. The residual errors order matches the order of edge IDs in `poseGraph`.

Examples

Optimize and Trim Loop Closures For 2-D Pose Graphs

Optimize a pose graph based on the nodes and edge constraints. Trim loop closed based on their edge residual errors.

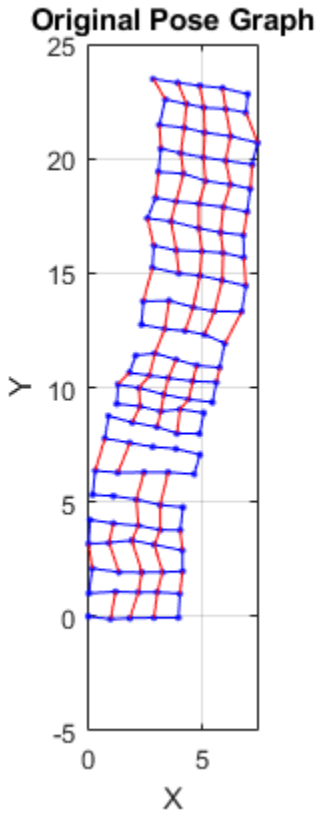
Load the data set that contains a 2-D pose graph. Inspect the `poseGraph` object to view the number of nodes and loop closures.

```
load grid-2d-posegraph.mat pg
disp(pg)
```

```
  poseGraph with properties:

              NumNodes: 120
              NumEdges: 193
    NumLoopClosureEdges: 74
     LoopClosureEdgeIDs: [120 121 122 123 124 125 126 127 128 129 130 ... ]
        LandmarkNodeIDs: [1x0 double]
```

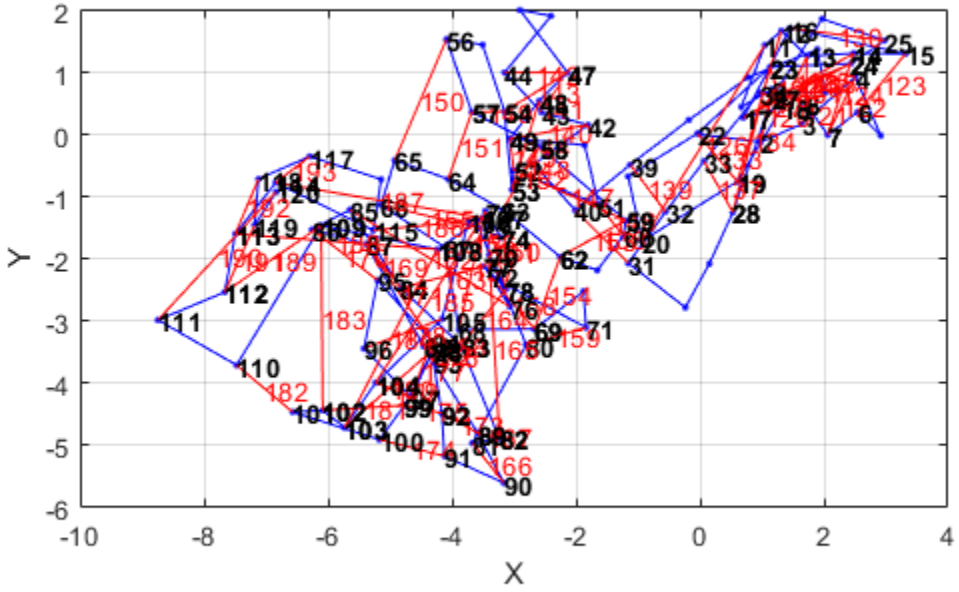
Plot the pose graph with IDs off. Red lines indicate loop closures identified in the dataset. The poses in the graph should follow a grid pattern, but show evidence of drift over time.

```
show(pg,'IDs','off');
title('Original Pose Graph')
```



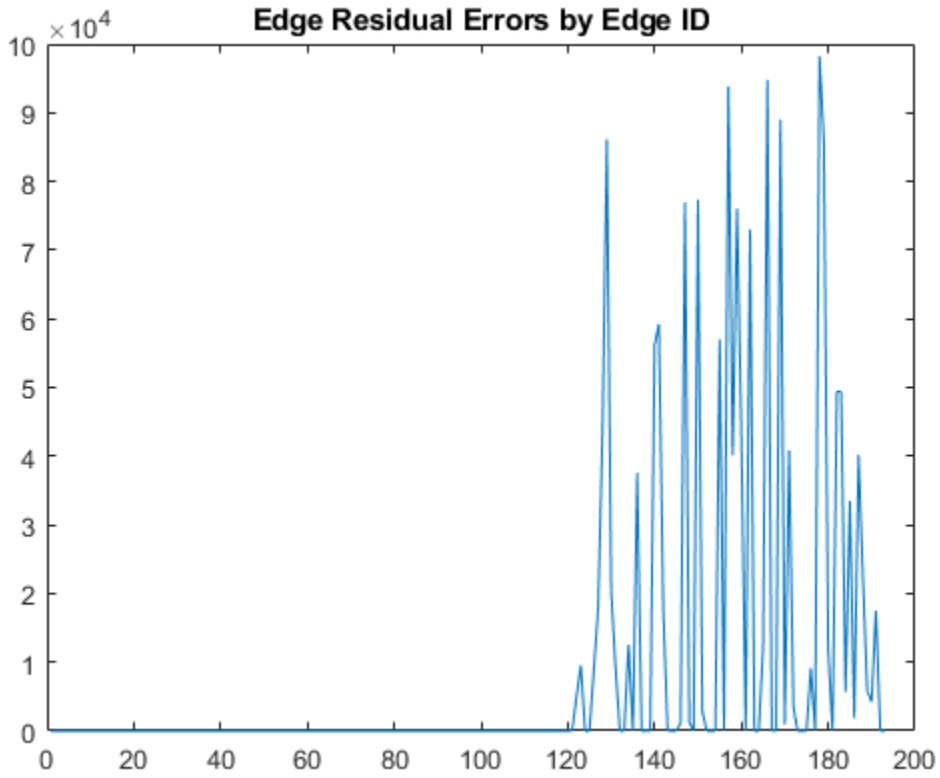
Optimize the pose grap using the `optimizePoseGraph` function. By default, this function uses the `"builtin-trust-region"` solver. Because the pose graph contains some bad loop closures, the resulting pose graph is actual not desirable.

```
pgOptim = optimizePoseGraph(pg);
figure;
show(pgOptim);
```



Look at the edge residual errors for the original pose graph. Large outlier error values at the end indicate bad loop closures.

```
resErrorVec = edgeResidualErrors(pg);
plot(resErrorVec);
title('Edge Residual Errors by Edge ID')
```



Certain loop closures should be trimmed from the pose graph based on their residual error. Use the `trimLoopClosures` function to trim therse bad loop closures. Set the maximum and truncation threshold for the trimmer parameters. This threshold is set based on the measurement accuracy and should be tuned for your system.

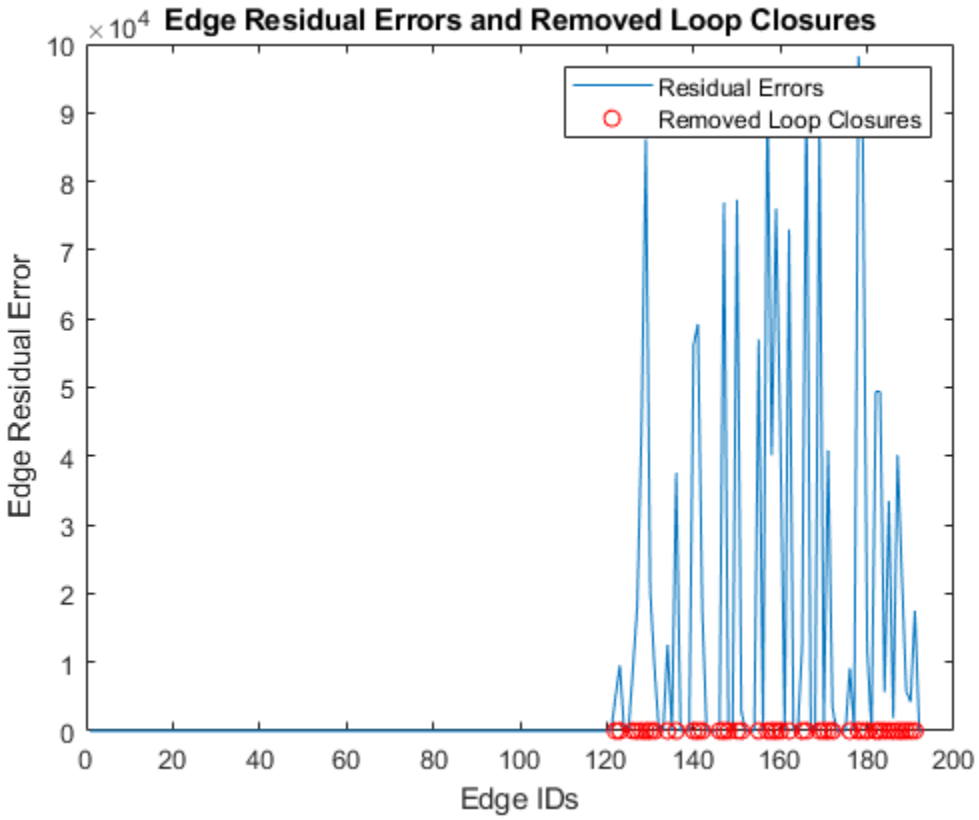
```
trimParams.MaxIterations = 100;
trimParams.TruncationThreshold = 25;

solverOptions = poseGraphSolverOptions;
```

Use the `trimLoopClosures` function with the trimmer parameters and solver options.

```
[pgNew, trimInfo, debugInfo] = trimLoopClosures(pg,trimParams,solverOptions);
```

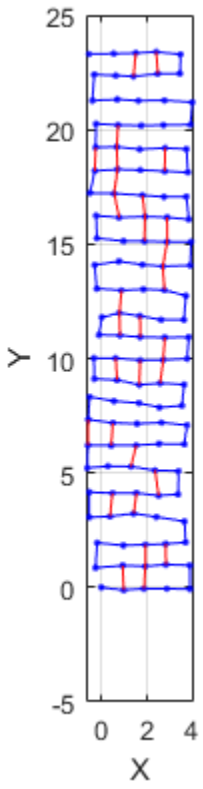
From the `trimInfo` output, plot the loop closures removed from the optimized pose graph. By plotting with the residual errors plot before, you can see the large error loop closures were removed.

```
removedLCs = trimInfo.LoopClosuresToRemove;

hold on
plot(removedLCs,zeros(length(removedLCs)),'or')
title('Edge Residual Errors and Removed Loop Closures')
legend('Residual Errors', 'Removed Loop Closures')
xlabel('Edge IDs')
ylabel('Edge Residual Error')
hold off
```



Show the new pose graph with the bad loop closures trimmed.

```
show(pgNew,"IDs","off");
```



Input Arguments

`poseGraphObj` — Pose graph
`poseGraph` object | `poseGraph3D` object

Pose graph, specified as a `poseGraph` or `poseGraph3D` object.

Output Arguments

`resErrorVec` — Edge residual errors for pose graph
vector of positive scalars

Edge residual errors for pose graph, specified as a vector of positive scalars.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

Use this syntax when constructing `poseGraph` or `poseGraph3D` objects for code generation:

`poseGraph = poseGraph('MaxNumEdges',maxEdges,'MaxNumNodes',maxNodes)` specifies an upper bound on the number of edges and nodes allowed in the pose graph when generating code. This limit is only required when generating code.

See Also

Functions
edgeNodePairs | optimizePoseGraph | addRelativePose | findEdgeID | nodeEstimates | removeEdges

Objects
poseGraph | poseGraph3D | lidarSLAM

Introduced in R2020b

findEdgeID

Find edge ID of edge

Syntax

```
edgeID = findEdgeID(poseGraph,nodePairs)
```

Description

`edgeID = findEdgeID(poseGraph,nodePairs)` finds the edge ID for a specified edge. Edges are defined by the IDs of the two nodes that connect them.

Input Arguments

`poseGraph` — Pose graph
`poseGraph` object | `poseGraph3D` object

Pose graph, specified as a `poseGraph` or `poseGraph3D` object.

`nodePairs` — Edge node pairs in pose graph
two-element vector

Edge node pairs in pose graph, specified as a two-element vector that lists the IDs of the two nodes that the edge connects.

Output Arguments

`edgeID` — Edge ID
positive integer | vector

Edge IDs, returned as a positive integer or vector of positive integers. The pose graph can contain multiple edges between each node pair, so multiple edge IDs may be returned for a single edge ID.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

Use this syntax when constructing `poseGraph` or `poseGraph3D` objects for code generation:

`poseGraph = poseGraph('MaxNumEdges',maxEdges,'MaxNumNodes',maxNodes)` specifies an upper bound on the number of edges and nodes allowed in the pose graph when generating code. This limit is only required when generating code.

See Also

Functions
`optimizePoseGraph` | `addRelativePose` | `edgeNodePairs` | `edgeConstraints` | `nodeEstimates` | `removeEdges`

Objects
`poseGraph` | `poseGraph3D` | `lidarSLAM`

Topics
“Implement Simultaneous Localization And Mapping (SLAM) with Lidar Scans”
“Implement Online Simultaneous Localization And Mapping (SLAM) with Lidar Scans”

Introduced in R2019b

nodeEstimates

Poses of nodes in pose graph

Syntax

```
measurements = nodeEstimates(poseGraph)
measurements = nodeEstimates(poseGraph,nodeIDs)
```

Description

`measurements = nodeEstimates(poseGraph)` lists all poses in the specified pose graph.

`measurements = nodeEstimates(poseGraph,nodeIDs)` lists the poses with the specified node IDs.

Input Arguments

`poseGraph` — Pose graph
`poseGraph` object | `poseGraph3D` object

Pose graph, specified as a `poseGraph` or `poseGraph3D` object.

`nodeIDs` — Node IDs
positive integer | vector of positive integers

Node IDs, specified as a positive integer or vector of positive integers. Each node added gets an ID sequentially in the graph.

Output Arguments

`measurements` — Measurements between nodes
n-by-3 matrix | *n*-by-7 matrix

Measurements between nodes, returned as an *n*-by-3 matrix or *n*-by-7 matrix.

For `poseGraph` (2-D), each row is an `[x y theta]` vector, which defines the relative *xy*-position and orientation angle, `theta`, of a pose in the graph. For landmark positions, `theta` is returned as `NaN`.

For `poseGraph3D`, each row is an `[x y z qw qx qy qz]` vector, which defines the relative *xyz*-position and quaternion orientation, `[qw qx qy qz]`, of a pose in the graph.

Note Many other sources for 3-D pose graphs, including `.g2o` formats, specify the quaternion orientation in a different order, for example, `[qx qy qz qw]`. Check the source of your pose graph data before adding nodes to your `poseGraph3D` object.

Compatibility Considerations

`nodeEstimates` was renamed
Behavior change in future release

The `nodeEstimates` object function was renamed from `nodes`. Use `nodeEstimates` when calling the function.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

Use this syntax when constructing `poseGraph` or `poseGraph3D` objects for code generation:

`poseGraph = poseGraph('MaxNumEdges',maxEdges,'MaxNumNodes',maxNodes)` specifies an upper bound on the number of edges and nodes allowed in the pose graph when generating code. This limit is only required when generating code.

See Also

Functions
`optimizePoseGraph` | `addRelativePose` | `edgeNodePairs` | `findEdgeID` | `removeEdges` | `edgeConstraints`

Objects
`poseGraph` | `poseGraph3D` | `lidarSLAM`

Topics
“Implement Simultaneous Localization And Mapping (SLAM) with Lidar Scans”
“Implement Online Simultaneous Localization And Mapping (SLAM) with Lidar Scans”

Introduced in R2019b

removeEdges

Remove loop closure edges from graph

Syntax

```
removeEdges(poseGraph,edgeIDs)
```

Description

`removeEdges(poseGraph,edgeIDs)` removes loop closure edges, landmark edges, or duplicate incremental edges from the pose graph.

Input Arguments

`poseGraph` — Pose graph
`poseGraph` object | `poseGraph3D` object

Pose graph, specified as a `poseGraph` or `poseGraph3D` object.

`edgeIDs` — Edge IDs
vector of positive integers

Edge IDs, specified as a vector of positive integers. To get edge IDs based on node pairs, see the `findEdgeID` function.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

Use this syntax when constructing `poseGraph` or `poseGraph3D` objects for code generation:

`poseGraph = poseGraph('MaxNumEdges',maxEdges,'MaxNumNodes',maxNodes)` specifies an upper bound on the number of edges and nodes allowed in the pose graph when generating code. This limit is only required when generating code.

See Also

Functions
`optimizePoseGraph` | `findEdgeID` | `addRelativePose` | `edgeNodePairs` | `edgeConstraints` | `nodeEstimates`

Objects
`poseGraph` | `poseGraph3D` | `lidarSLAM`

Topics
“Implement Simultaneous Localization And Mapping (SLAM) with Lidar Scans”
“Implement Online Simultaneous Localization And Mapping (SLAM) with Lidar Scans”

Introduced in R2019b

show

Plot pose graph

Syntax

```
show(poseGraph)
show(poseGraph,Name,Value)
axes = show( ___ )
```

Description

`show(poseGraph)` plots the specified pose graph in a figure.

`show(poseGraph,Name,Value)` specifies options using `Name,Value` pair arguments. For example, `'IDs','on'` plots all node and edge IDs of the pose graph.

`axes = show( ___ )` returns the axes handle that the pose graph is plotted to using any of previous syntaxes.

Examples

Optimize a 2-D Pose Graph

Optimize a pose graph based on the nodes and edge constraints. The pose graph used in this example is from the Intel Research Lab Dataset and was generated from collecting wheel odometry and a laser range finder sensor information in an indoor lab.

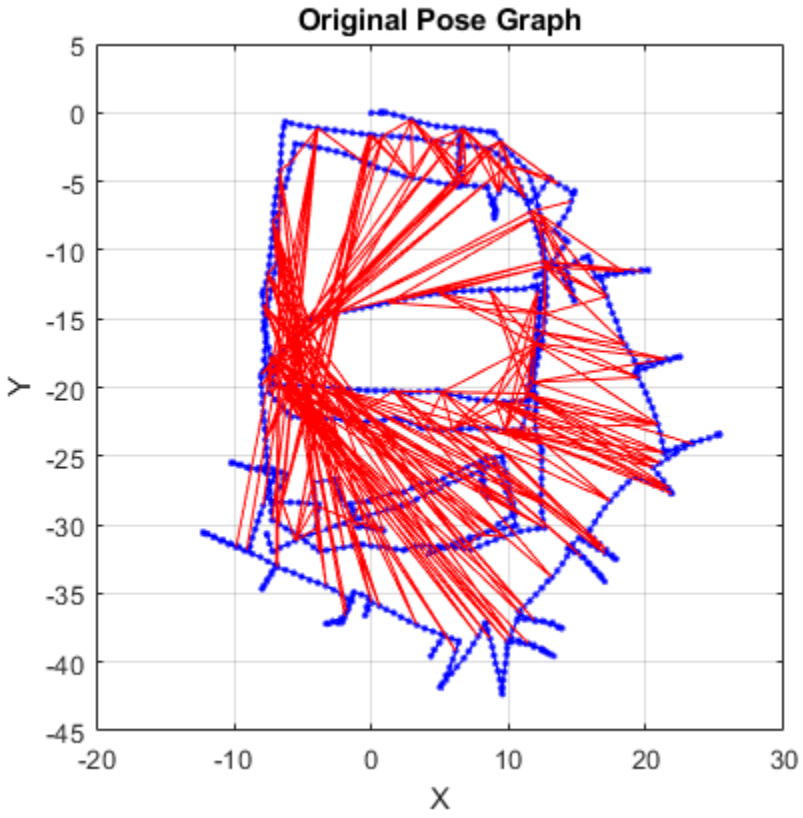
Load the Intel data set that contains a 2-D pose graph. Inspect the `poseGraph` object to view the number of nodes and loop closures.

```
load intel-2d-posegraph.mat pg
disp(pg)
```

```
  poseGraph with properties:

              NumNodes: 1228
              NumEdges: 1483
    NumLoopClosureEdges: 256
     LoopClosureEdgeIDs: [1228 1229 1230 1231 1232 1233 1234 1235 1236 ... ]
        LandmarkNodeIDs: [1x0 double]
```

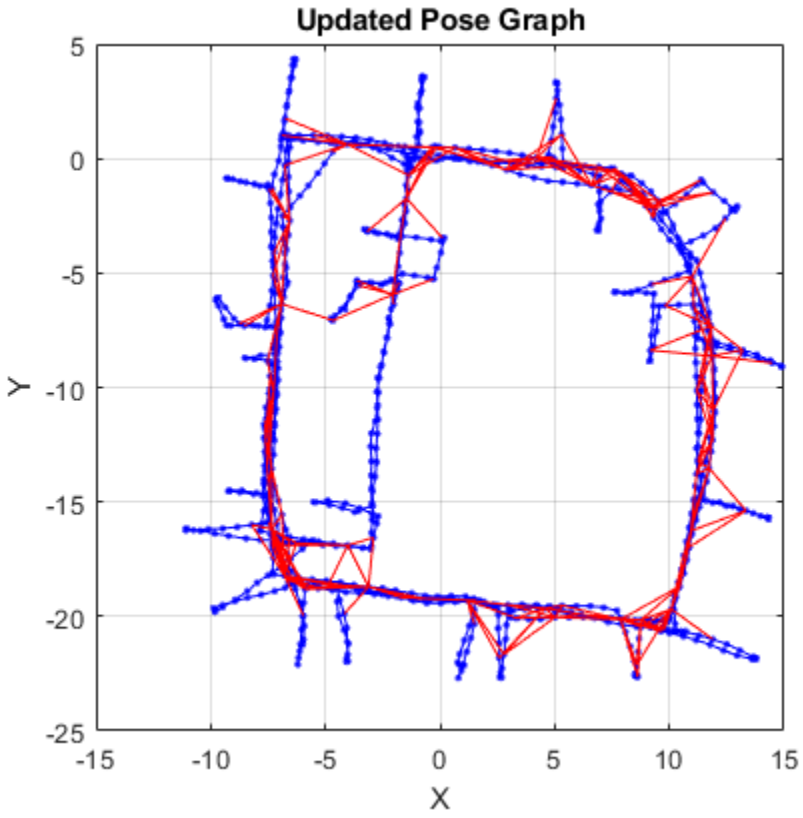
Plot the pose graph with IDs off. Red lines indicate loop closures identified in the dataset.

```
show(pg,'IDs','off');
title('Original Pose Graph')
```



Optimize the pose graph. Nodes are adjusted based on the edge constraints and loop closures. Plot the optimized pose graph to see the adjustment of the nodes with loop closures.

```
updatedPG = optimizePoseGraph(pg);
figure
show(updatedPG,'IDs','off');
title('Updated Pose Graph')
```



Optimize a 3-D Pose Graph

Optimize a pose graph based on the nodes and edge constraints. The pose graph used in this example is taken from the MIT Dataset and was generated using information extracted from a parking garage.

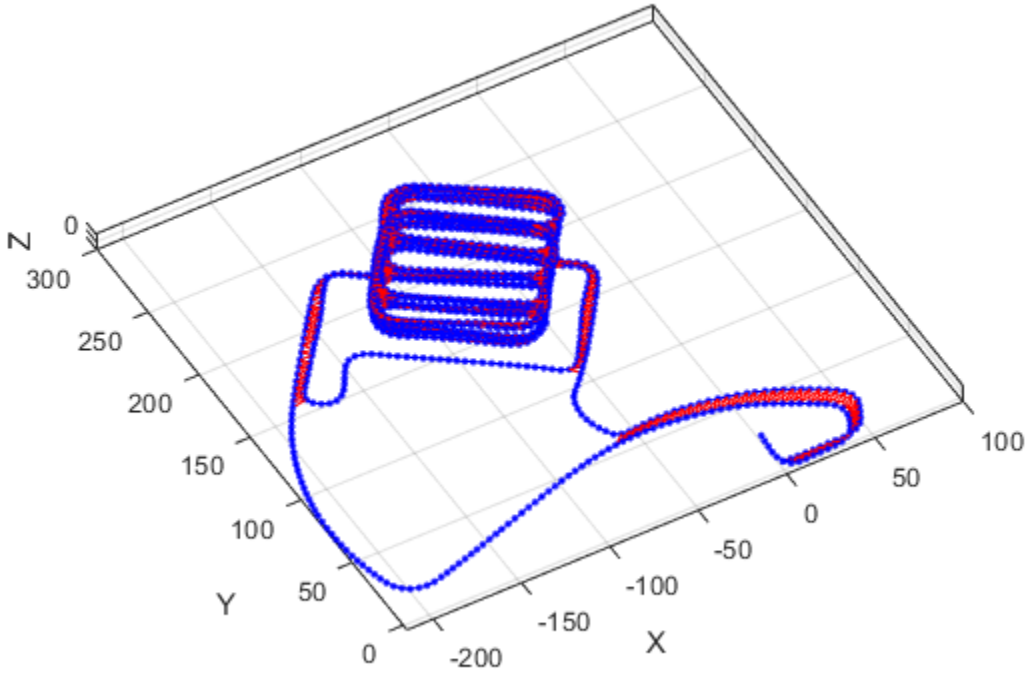
Load the pose graph from the MIT dataset. Inspect the `poseGraph3D` object to view the number of nodes and loop closures.

```
load parking-garage-posegraph.mat pg
disp(pg);
```

```
  poseGraph3D with properties:

              NumNodes: 1661
              NumEdges: 6275
    NumLoopClosureEdges: 4615
     LoopClosureEdgeIDs: [128 129 130 132 133 134 135 137 138 139 140 ... ]
        LandmarkNodeIDs: [1x0 double]
```

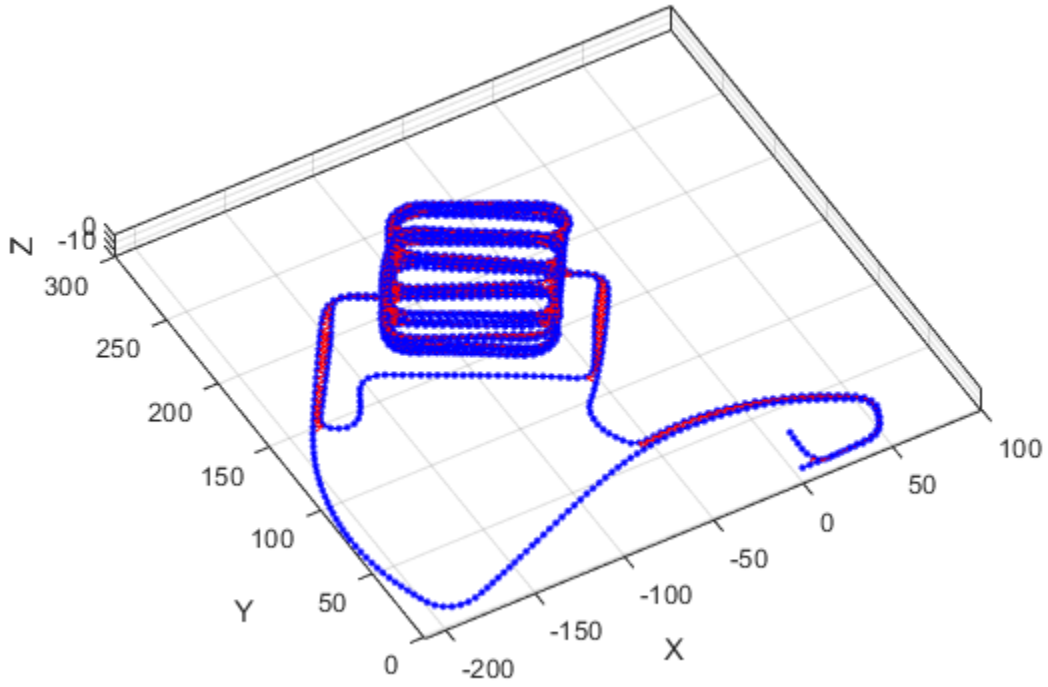
Plot the pose graph with IDs off. Red lines indicate loop closures identified in the dataset.

```
title('Original Pose Graph')
show(pg,'IDs','off');
view(-30,45)
```



Optimize the pose graph. Nodes are adjusted based on the edge constraints and loop closures. Plot the optimized pose graph to see the adjustment of the nodes with loop closures.

```
updatedPG = optimizePoseGraph(pg);
figure
title('Updated Pose Graph')
show(updatedPG,'IDs','off');
view(-30,45)
```



Input Arguments

poseGraph — Pose graph

`poseGraph` object | `poseGraph3D` object

Pose graph, specified as a `poseGraph` or `poseGraph3D` object.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'IDs','off'`

Parent — Axes used to plot pose graph

`Axes` object | `UIAxes` object

Axes used to plot the pose graph, specified as the comma-separated pair consisting of `'Parent'` and either an `Axes` or `UIAxes`object. See `axes` or `uiaxes`.

IDs — Display of IDs on pose graph

`'loopclosures'` (default) | `'all'` | `'nodes'` | `'landmarks'` | `'off'`

Display of IDs on pose graph, specified as the comma-separated pair consisting of `'IDs'` and one of the following:

- `'all'` — Plot all node and edge IDs.
- `'nodes'` — Plot all node IDs and loop closure IDs.
- `'loopclosures'` — Plot only loop closure edge IDs.
- `'landmarks'` — Plot landmark edge IDs.
- `'off'` — Do not plot any IDs.

Output Arguments

axes — Axes used to plot the map

`Axes` object | `UIAxes` object

Axes used to plot the map, returned as either an `Axes` or `UIAxes` object. See `axes` or `uiaxes`.

See Also

Functions

`optimizePoseGraph` | `addRelativePose`

Objects

`poseGraph` | `poseGraph3D` | `lidarSLAM`

Topics

“Implement Simultaneous Localization And Mapping (SLAM) with Lidar Scans”
“Implement Online Simultaneous Localization And Mapping (SLAM) with Lidar Scans”
“Landmark SLAM Using AprilTag Markers”

Introduced in R2019b

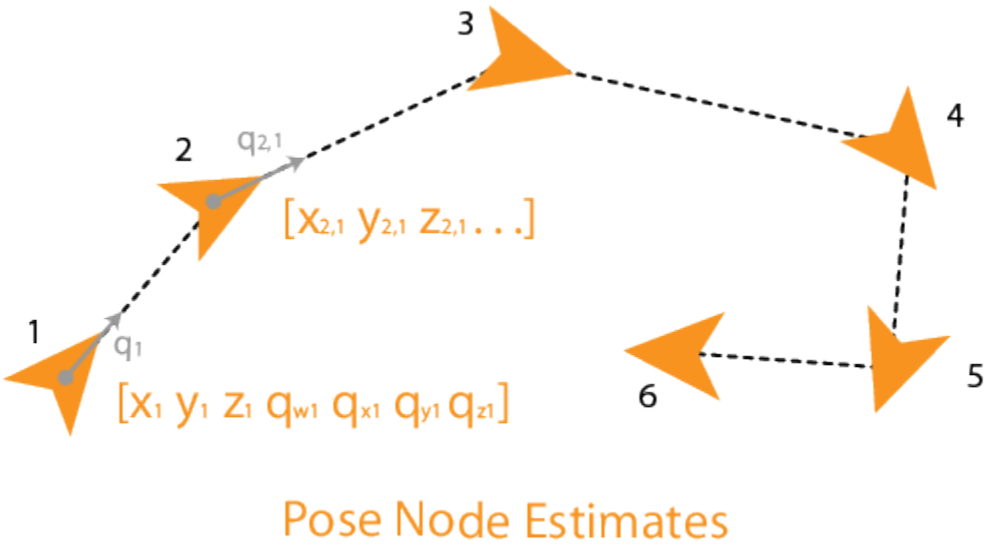
poseGraph3D

Create 3-D pose graph

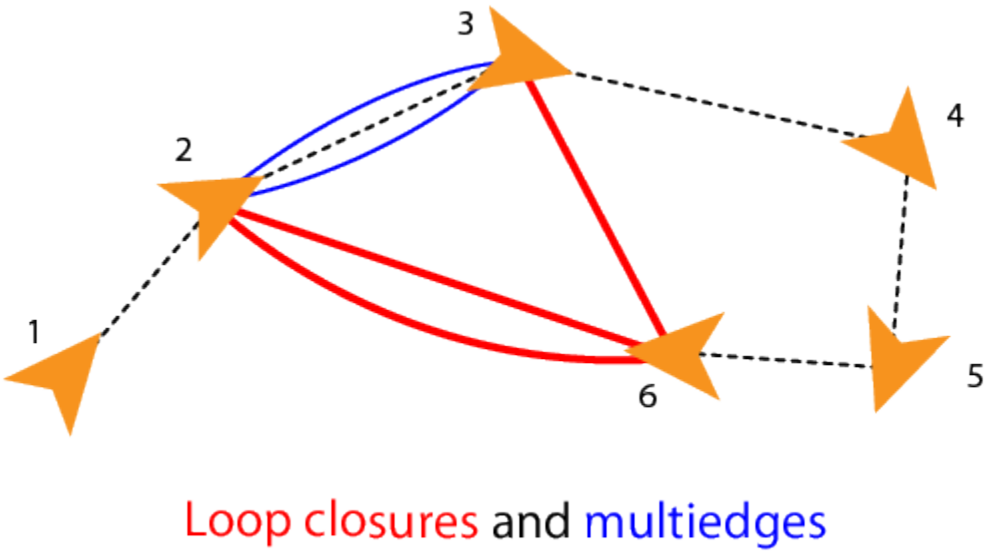
Description

A `poseGraph3D` object stores information for a 3-D pose graph representation. A pose graph contains nodes connected by edges. Each node estimate is connected to the graph by edge constraints that define the relative pose between nodes and the uncertainty on that measurement.

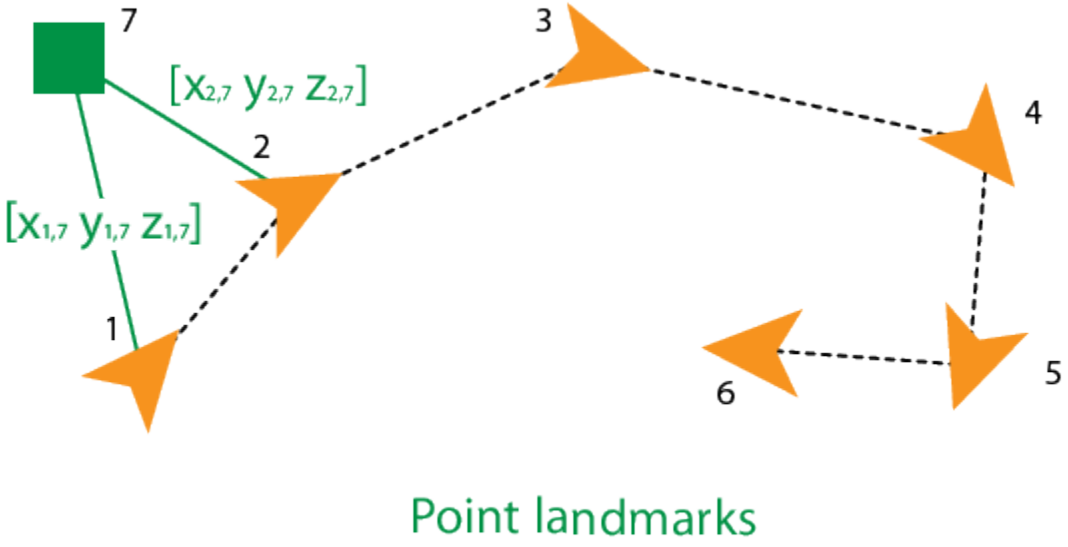
To construct a pose graph iteratively, use the `addRelativePose` function to add relative pose estimates and connect them to an existing node with specified edge constraints. Pose nodes must be specified relative to a pose node. Specify the uncertainty of the measurement using an information matrix.



Adding an edge between two nonsequential nodes creates a *loop closure* in the graph. Multiple edges or *multiedges* between node pairs are also supported, which includes loop closures. To add additional edge constraints or loop closures, specify the node IDs using the `addRelativePose` function. When optimizing the pose graph, the `optimizePoseGraph` function finds a solution to satisfy all these edge constraints.



To add landmark point nodes, use the `addPointLandmark` function. This function specifies nodes as *xyz*-points without orientation estimates. Landmarks must be specified relative to a pose node.



For 2-D pose graphs, see `poseGraph`.

For an example that builds and optimizes a 3-D pose graph from real-world sensor data, see "Landmark SLAM Using AprilTag Markers".

Creation

Syntax

```
poseGraph = poseGraph3D
poseGraph = poseGraph3D('MaxNumEdges',maxEdges,'MaxNumNodes',maxNodes)
```

Description

`poseGraph = poseGraph3D` creates a 3-D pose graph object. Add poses using `addRelativePose` to construct a pose graph iteratively.

`poseGraph = poseGraph3D('MaxNumEdges',maxEdges,'MaxNumNodes',maxNodes)` specifies an upper bound on the number of edges and nodes allowed in the pose graph when generating code. This limit is only required when generating code.

Properties

`NumNodes` — Number of nodes in pose graph
`1` (default) | positive integer

This property is read-only.

Number of nodes in pose graph, specified as a positive integer. Each node represents a pose measurement or a point landmark measurement. To specify relative poses between nodes, use `addRelativePose`. To specify a landmark pose, use `addLandmarkPose`. To get a list of all nodes, use `edgeNodePairs`.

`NumEdges` — Number of edges in pose graph
`0` (default) | nonnegative integer

This property is read-only.

Number of edges in pose graph, specified as a nonnegative integer. Each edge connects two nodes in the pose graph. Loop closure edges and landmark edges are included.

`NumLoopClosureEdges` — Number of loop closures
`0` (default) | nonnegative integer

This property is read-only.

Number of loop closures in pose graph, specified as a nonnegative integer. To get the edge IDs of the loop closures, use the `LoopClosureEdgeIDs` property.

`LoopClosureEdgeIDs` — Loop closure edge IDs
vector

This property is read-only.

Loop closure edges IDs, specified as a vector of edge IDs.

`LandmarkNodeIDs` — Landmark node IDs
vector

This property is read-only.

Landmark node IDs, specified as a vector of IDs for each node.

Object Functions

addPointLandmark	Add landmark point node to pose graph
addRelativePose	Add relative pose to pose graph
copy	Create copy of pose graph
edgeNodePairs	Edge node pairs in pose graph
edgeConstraints	Edge constraints in pose graph
edgeResidualErrors	Compute pose graph edge residual errors
findEdgeID	Find edge ID of edge
nodeEstimates	Poses of nodes in pose graph
removeEdges	Remove loop closure edges from graph
show	Plot pose graph

Examples

Optimize a 3-D Pose Graph

Optimize a pose graph based on the nodes and edge constraints. The pose graph used in this example is taken from the MIT Dataset and was generated using information extracted from a parking garage.

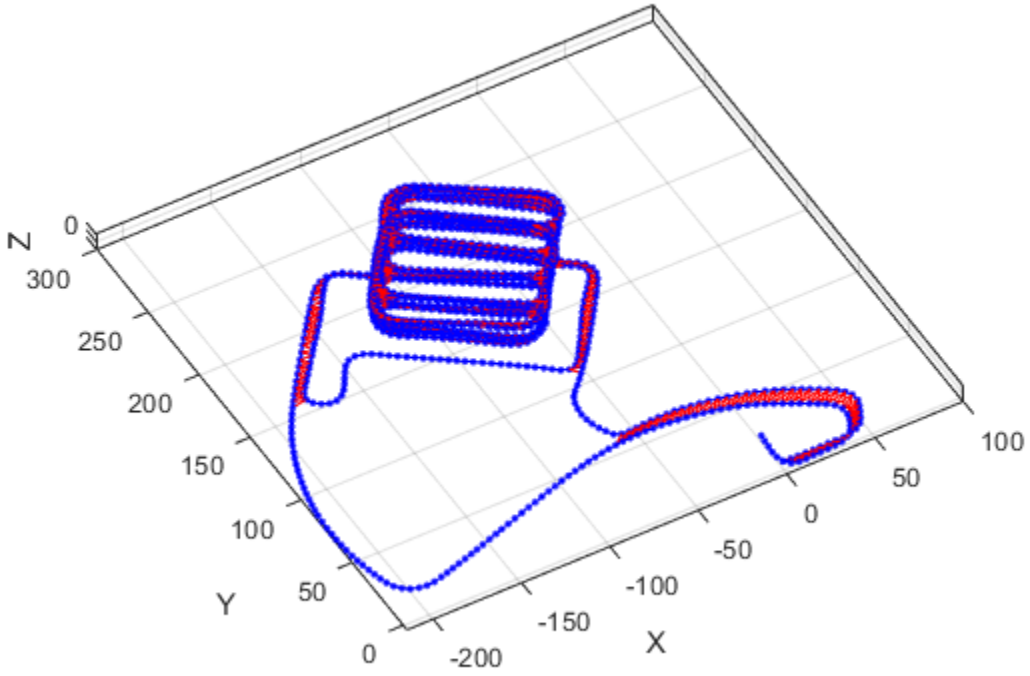
Load the pose graph from the MIT dataset. Inspect the `poseGraph3D` object to view the number of nodes and loop closures.

```
load parking-garage-posegraph.mat pg
disp(pg);
```

```
  poseGraph3D with properties:

              NumNodes: 1661
              NumEdges: 6275
    NumLoopClosureEdges: 4615
     LoopClosureEdgeIDs: [128 129 130 132 133 134 135 137 138 139 140 ... ]
        LandmarkNodeIDs: [1x0 double]
```

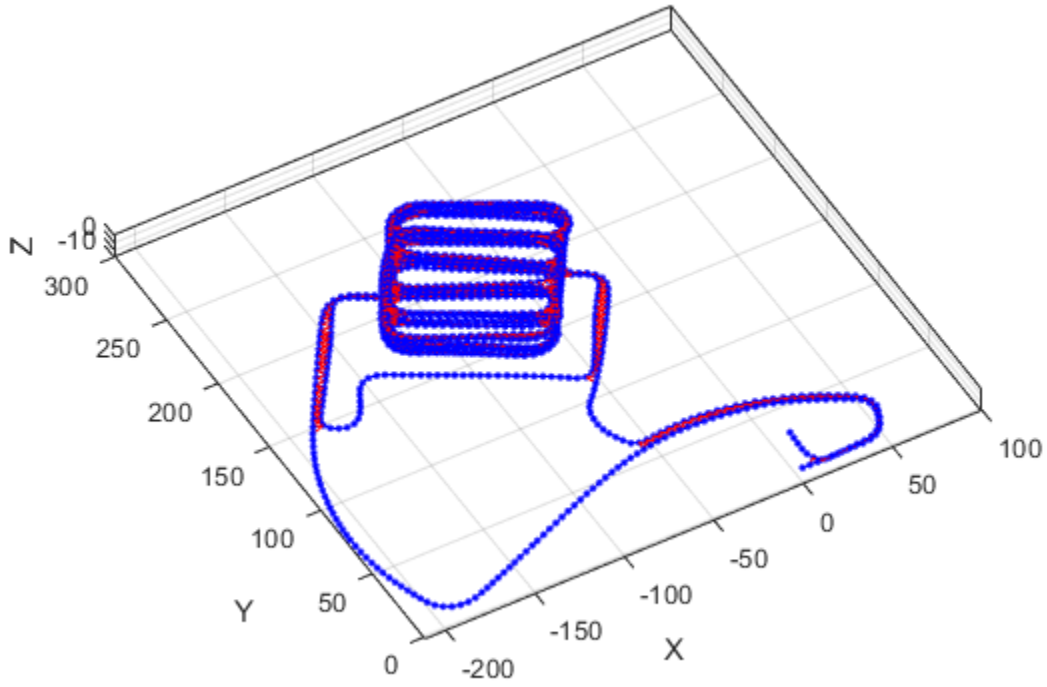
Plot the pose graph with IDs off. Red lines indicate loop closures identified in the dataset.

```
title('Original Pose Graph')
show(pg,'IDs','off');
view(-30,45)
```



Optimize the pose graph. Nodes are adjusted based on the edge constraints and loop closures. Plot the optimized pose graph to see the adjustment of the nodes with loop closures.

```
updatedPG = optimizePoseGraph(pg);
figure
title('Updated Pose Graph')
show(updatedPG,'IDs','off');
view(-30,45)
```



References

[1] Carlone, Luca, Roberto Tron, Kostas Daniilidis, and Frank Dellaert. "Initialization Techniques for 3D SLAM: a Survey on Rotation Estimation and its Use in Pose Graph Optimization." *2015 IEEE International Conference on Robotics and Automation (ICRA)*. 2015, pp. 4597–4604.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Use this syntax when constructing `poseGraph3D` objects for code generation:

`poseGraph = poseGraph3D('MaxNumEdges',maxEdges,'MaxNumNodes',maxNodes)` specifies an upper bound on the number of edges and nodes allowed in the pose graph when generating code. This limit is only required when generating code.

See Also

Functions

`optimizePoseGraph` | `addRelativePose` | `addPointLandmark`

Objects

`poseGraph` | `lidarSLAM`

Topics
“Landmark SLAM Using AprilTag Markers”

Introduced in R2019b

quaternion

Create a quaternion array

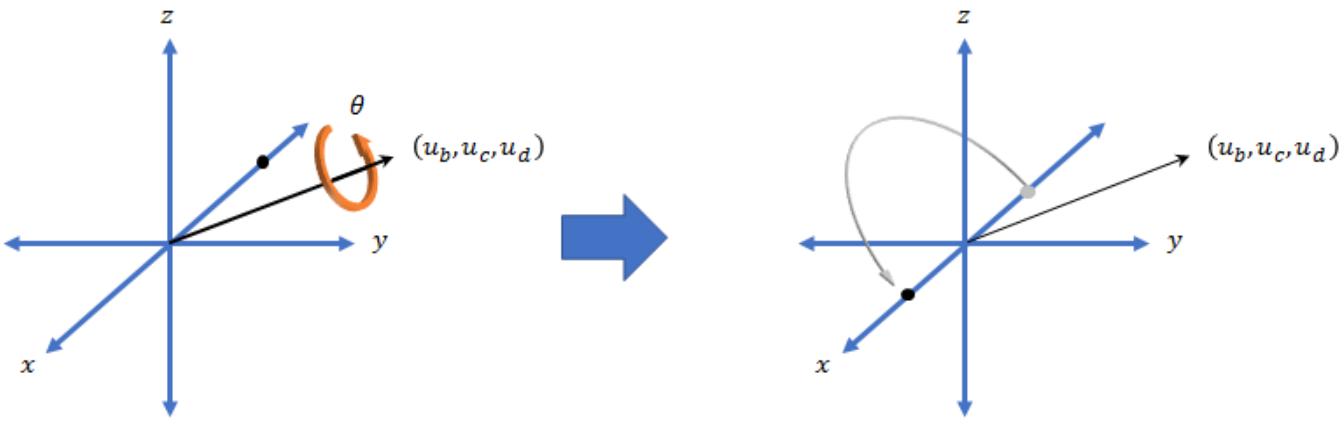
Description

A quaternion is a four-part hyper-complex number used in three-dimensional rotations and orientations.

A quaternion number is represented in the form $a + b\mathrm{i} + c\mathrm{j} + d\mathrm{k}$, where a, b, c, and d parts are real numbers, and i, j, and k are the basis elements, satisfying the equation: $\mathrm{i}^2 = \mathrm{j}^2 = \mathrm{k}^2 = \mathrm{ijk} = -1$.

The set of quaternions, denoted by **H**, is defined within a four-dimensional vector space over the real numbers, $\mathbf{R}^4$. Every element of **H** has a unique representation based on a linear combination of the basis elements, i, j, and k.

All rotations in 3-D can be described by an axis of rotation and angle about that axis. An advantage of quaternions over rotation matrices is that the axis and angle of rotation is easy to interpret. For example, consider a point in $\mathbf{R}^3$. To rotate the point, you define an axis of rotation and an angle of rotation.



The quaternion representation of the rotation may be expressed as $q = \cos\left(\theta/2\right) + \sin\left(\theta/2\right)(u_b\mathrm{i} + u_c\mathrm{j} + u_d\mathrm{k})$, where θ is the angle of rotation and [u_b, u_c, and u_d] is the axis of rotation.

Creation

Syntax

```
quat = quaternion()
quat = quaternion(A,B,C,D)
quat = quaternion(matrix)
quat = quaternion(RV,'rotvec')
```

```
quat = quaternion(RV,'rotvecd')
quat = quaternion(RM,'rotmat',PF)
quat = quaternion(E,'euler',RS,PF)
quat = quaternion(E,'eulerd',RS,PF)
```

Description

`quat = quaternion()` creates an empty quaternion.

`quat = quaternion(A,B,C,D)` creates a quaternion array where the four quaternion parts are taken from the arrays `A`, `B`, `C`, and `D`. All the inputs must have the same size and be of the same data type.

`quat = quaternion(matrix)` creates an *N*-by-1 quaternion array from an *N*-by-4 matrix, where each column becomes one part of the quaternion.

`quat = quaternion(RV,'rotvec')` creates an *N*-by-1 quaternion array from an *N*-by-3 matrix of rotation vectors, `RV`. Each row of `RV` represents a rotation vector in radians.

`quat = quaternion(RV,'rotvecd')` creates an *N*-by-1 quaternion array from an *N*-by-3 matrix of rotation vectors, `RV`. Each row of `RV` represents a rotation vector in degrees.

`quat = quaternion(RM,'rotmat',PF)` creates an *N*-by-1 quaternion array from the 3-by-3-by-*N* array of rotation matrices, `RM`. `PF` can be either `'point'` if the Euler angles represent point rotations or `'frame'` for frame rotations.

`quat = quaternion(E,'euler',RS,PF)` creates an *N*-by-1 quaternion array from the *N*-by-3 matrix, `E`. Each row of `E` represents a set of Euler angles in radians. The angles in `E` are rotations about the axes in sequence `RS`.

`quat = quaternion(E,'eulerd',RS,PF)` creates an *N*-by-1 quaternion array from the *N*-by-3 matrix, `E`. Each row of `E` represents a set of Euler angles in degrees. The angles in `E` are rotations about the axes in sequence `RS`.

Input Arguments

`A,B,C,D` — Quaternion parts

comma-separated arrays of the same size

Parts of a quaternion, specified as four comma-separated scalars, matrices, or multi-dimensional arrays of the same size.

Example: `quat = quaternion(1,2,3,4)` creates a quaternion of the form 1 + 2i + 3j + 4k.

Example: `quat = quaternion([1,5],[2,6],[3,7],[4,8])` creates a 1-by-2 quaternion array where `quat(1,1) = 1 + 2i + 3j + 4k` and `quat(1,2) = 5 + 6i + 7j + 8k`

Data Types: `single` | `double`

`matrix` — Matrix of quaternion parts

N-by-4 matrix

Matrix of quaternion parts, specified as an *N*-by-4 matrix. Each row represents a separate quaternion. Each column represents a separate quaternion part.

Example: `quat = quaternion(rand(10,4))` creates a 10-by-1 quaternion array.

Data Types: `single` | `double`

`RV` — Matrix of rotation vectors
N-by-3 matrix

Matrix of rotation vectors, specified as an *N*-by-3 matrix. Each row of `RV` represents the [X Y Z] elements of a rotation vector. A rotation vector is a unit vector representing the axis of rotation scaled by the angle of rotation in radians or degrees.

To use this syntax, specify the first argument as a matrix of rotation vectors and the second argument as the `'rotvec'` or `'rotvecd'`.

Example: `quat = quaternion(rand(10,3),'rotvec')` creates a 10-by-1 quaternion array.

Data Types: `single` | `double`

`RM` — Rotation matrices
3-by-3 matrix | 3-by-3-by-*N* array

Array of rotation matrices, specified by a 3-by-3 matrix or 3-by-3-by-*N* array. Each page of the array represents a separate rotation matrix.

Example: `quat = quaternion(rand(3),'rotmat','point')`

Example: `quat = quaternion(rand(3),'rotmat','frame')`

Data Types: `single` | `double`

`PF` — Type of rotation matrix
`'point'` | `'frame'`

Type of rotation matrix, specified by `'point'` or `'frame'`.

Example: `quat = quaternion(rand(3),'rotmat','point')`

Example: `quat = quaternion(rand(3),'rotmat','frame')`

Data Types: `char` | `string`

`E` — Matrix of Euler angles
N-by-3 matrix

Matrix of Euler angles, specified by an *N*-by-3 matrix. If using the `'euler'` syntax, specify `E` in radians. If using the `'eulerd'` syntax, specify `E` in degrees.

Example: `quat = quaternion(E,'euler','YZY','point')`

Example: `quat = quaternion(E,'euler','XYZ','frame')`

Data Types: `single` | `double`

`RS` — Rotation sequence
character vector | scalar string

Rotation sequence, specified as a three-element character vector:

- `'YZY'`
- `'YXY'`
- `'ZYZ'`

- 'ZXZ'
- 'XYX'
- 'XZX'
- 'XYZ'
- 'YZX'
- 'ZXY'
- 'XZY'
- 'ZYX'
- 'YXZ'

Assume you want to determine the new coordinates of a point when its coordinate system is rotated using frame rotation. The point is defined in the original coordinate system as:

```
point = [sqrt(2)/2,sqrt(2)/2,0];
```

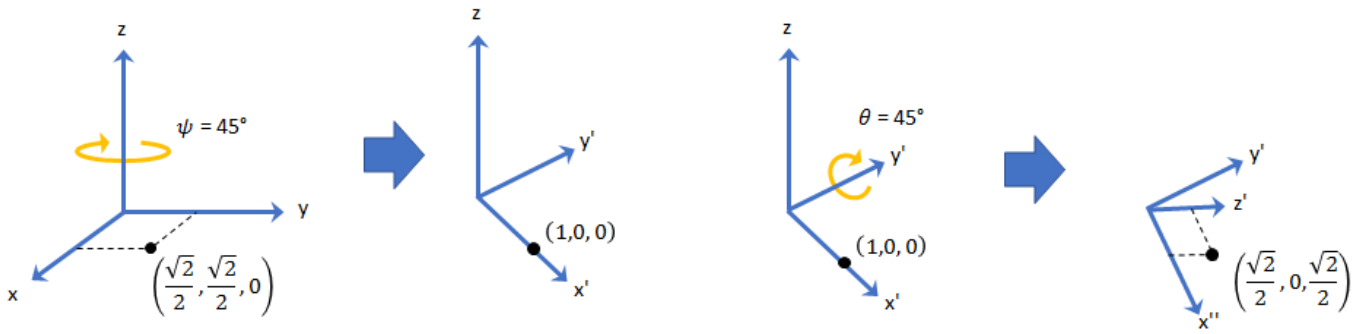
In this representation, the first column represents the *x*-axis, the second column represents the *y*-axis, and the third column represents the *z*-axis.

You want to rotate the point using the Euler angle representation [45,45,0]. Rotate the point using two different rotation sequences:

- If you create a quaternion rotator and specify the 'ZYX' sequence, the frame is first rotated 45° around the *z*-axis, then 45° around the new *y*-axis.

  ```
  quatRotator = quaternion([45,45,0],'eulerd','ZYX','frame');
  newPointCoordinate = rotateframe(quatRotator,point)
  ```

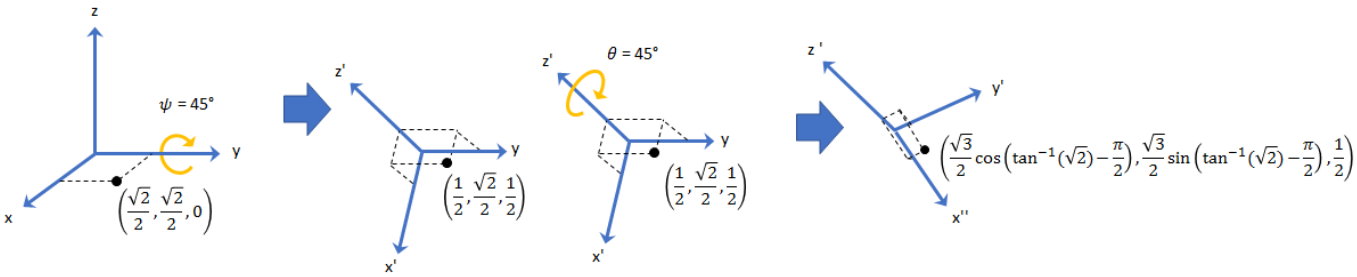
  ```
  newPointCoordinate =

      0.7071   -0.0000    0.7071
  ```



- If you create a quaternion rotator and specify the 'YZX' sequence, the frame is first rotated 45° around the *y*-axis, then 45° around the new *z*-axis.

  ```
  quatRotator = quaternion([45,45,0],'eulerd','YZX','frame');
  newPointCoordinate = rotateframe(quatRotator,point)
  ```

  ```
  newPointCoordinate =

      0.8536    0.1464    0.5000
  ```



Data Types: `char` | `string`

Object Functions

angvel	Angular velocity from quaternion array
classUnderlying	Class of parts within quaternion
compact	Convert quaternion array to N-by-4 matrix
conj	Complex conjugate of quaternion
'	Complex conjugate transpose of quaternion array
dist	Angular distance in radians
euler	Convert quaternion to Euler angles (radians)
eulerd	Convert quaternion to Euler angles (degrees)
exp	Exponential of quaternion array
.\,ldivide	Element-wise quaternion left division
log	Natural logarithm of quaternion array
meanrot	Quaternion mean rotation
-	Quaternion subtraction
*	Quaternion multiplication
norm	Quaternion norm
normalize	Quaternion normalization
ones	Create quaternion array with real parts set to one and imaginary parts set to zero
parts	Extract quaternion parts
.^,power	Element-wise quaternion power
prod	Product of a quaternion array
randrot	Uniformly distributed random rotations
./,rdivide	Element-wise quaternion right division
rotateframe	Quaternion frame rotation
rotatepoint	Quaternion point rotation
rotmat	Convert quaternion to rotation matrix
rotvec	Convert quaternion to rotation vector (radians)
rotvecd	Convert quaternion to rotation vector (degrees)
slerp	Spherical linear interpolation
.*,times	Element-wise quaternion multiplication
'	Transpose a quaternion array
-	Quaternion unary minus
zeros	Create quaternion array with all parts set to zero

Examples

Create Empty Quaternion

```
quat = quaternion()
```

```
quat =

  0x0 empty quaternion array
```

By default, the underlying class of the quaternion is a double.

```
classUnderlying(quat)
```

```
ans =
'double'
```

Create Quaternion by Specifying Individual Quaternion Parts

You can create a quaternion array by specifying the four parts as comma-separated scalars, matrices, or multidimensional arrays of the same size.

Define quaternion parts as scalars.

```
A = 1.1;
B = 2.1;
C = 3.1;
D = 4.1;
quatScalar = quaternion(A,B,C,D)
```

```
quatScalar = quaternion
     1.1 + 2.1i + 3.1j + 4.1k
```

Define quaternion parts as column vectors.

```
A = [1.1;1.2];
B = [2.1;2.2];
C = [3.1;3.2];
D = [4.1;4.2];
quatVector = quaternion(A,B,C,D)
```

```
quatVector = 2x1 quaternion array
     1.1 + 2.1i + 3.1j + 4.1k
     1.2 + 2.2i + 3.2j + 4.2k
```

Define quaternion parts as matrices.

```
A = [1.1,1.3; ...
     1.2,1.4];
B = [2.1,2.3; ...
     2.2,2.4];
C = [3.1,3.3; ...
     3.2,3.4];
D = [4.1,4.3; ...
     4.2,4.4];
quatMatrix = quaternion(A,B,C,D)
```

```
quatMatrix = 2x2 quaternion array
     1.1 + 2.1i + 3.1j + 4.1k     1.3 + 2.3i + 3.3j + 4.3k
     1.2 + 2.2i + 3.2j + 4.2k     1.4 + 2.4i + 3.4j + 4.4k
```

Define quaternion parts as three dimensional arrays.

```
A = randn(2,2,2);
B = zeros(2,2,2);
C = zeros(2,2,2);
D = zeros(2,2,2);
quatMultiDimArray = quaternion(A,B,C,D)
```

```
quatMultiDimArray = 2x2x2 quaternion array
quatMultiDimArray(:,:,1) =

     0.53767 +       0i +       0j +       0k     -2.2588 +       0i +       0j +       0k
      1.8339 +       0i +       0j +       0k     0.86217 +       0i +       0j +       0k


quatMultiDimArray(:,:,2) =

     0.31877 +       0i +       0j +       0k    -0.43359 +       0i +       0j +       0k
     -1.3077 +       0i +       0j +       0k     0.34262 +       0i +       0j +       0k
```

Create Quaternion by Specifying Quaternion Parts Matrix

You can create a scalar or column vector of quaternions by specify an *N*-by-4 matrix of quaternion parts, where columns correspond to the quaternion parts A, B, C, and D.

Create a column vector of random quaternions.

```
quatParts = rand(3,4)
```

```
quatParts = 3×4

    0.8147    0.9134    0.2785    0.9649
    0.9058    0.6324    0.5469    0.1576
    0.1270    0.0975    0.9575    0.9706
```

```
quat = quaternion(quatParts)
```

```
quat = 3x1 quaternion array
     0.81472 + 0.91338i +  0.2785j + 0.96489k
     0.90579 + 0.63236i + 0.54688j + 0.15761k
     0.12699 + 0.09754i + 0.95751j + 0.97059k
```

To retrieve the `quatParts` matrix from quaternion representation, use `compact`.

```
retrievedquatParts = compact(quat)
```

```
retrievedquatParts = 3×4

    0.8147    0.9134    0.2785    0.9649
    0.9058    0.6324    0.5469    0.1576
    0.1270    0.0975    0.9575    0.9706
```

Create Quaternion by Specifying Rotation Vectors

You can create an *N*-by-1 quaternion array by specifying an *N*-by-3 matrix of rotation vectors in radians or degrees. Rotation vectors are compact spatial representations that have a one-to-one relationship with normalized quaternions.

Rotation Vectors in Radians

Create a scalar quaternion using a rotation vector and verify the resulting quaternion is normalized.

```
rotationVector = [0.3491,0.6283,0.3491];
quat = quaternion(rotationVector,'rotvec')
```

```
quat = quaternion
     0.92124 + 0.16994i + 0.30586j + 0.16994k
```

```
norm(quat)
```

```
ans = 1.0000
```

You can convert from quaternions to rotation vectors in radians using the `rotvec` function. Recover the `rotationVector` from the quaternion, `quat`.

```
rotvec(quat)
```

```
ans = 1×3

    0.3491    0.6283    0.3491
```

Rotation Vectors in Degrees

Create a scalar quaternion using a rotation vector and verify the resulting quaternion is normalized.

```
rotationVector = [20,36,20];
quat = quaternion(rotationVector,'rotvecd')
```

```
quat = quaternion
     0.92125 + 0.16993i + 0.30587j + 0.16993k
```

```
norm(quat)
```

```
ans = 1
```

You can convert from quaternions to rotation vectors in degrees using the `rotvecd` function. Recover the `rotationVector` from the quaternion, `quat`.

```
rotvecd(quat)
```

```
ans = 1×3

   20.0000   36.0000   20.0000
```

Create Quaternion by Specifying Rotation Matrices

You can create an N-by-1 quaternion array by specifying a 3-by-3-by-N array of rotation matrices. Each page of the rotation matrix array corresponds to one element of the quaternion array.

Create a scalar quaternion using a 3-by-3 rotation matrix. Specify whether the rotation matrix should be interpreted as a frame or point rotation.

```
rotationMatrix = [1 0         0; ...
                  0 sqrt(3)/2 0.5; ...
                  0 -0.5      sqrt(3)/2];
quat = quaternion(rotationMatrix,'rotmat','frame')
```

```
quat = quaternion
     0.96593 + 0.25882i +       0j +       0k
```

You can convert from quaternions to rotation matrices using the `rotmat` function. Recover the `rotationMatrix` from the quaternion, `quat`.

```
rotmat(quat,'frame')
```

```
ans = 3×3

    1.0000         0         0
         0    0.8660    0.5000
         0   -0.5000    0.8660
```

Create Quaternion by Specifying Euler Angles

You can create an *N*-by-1 quaternion array by specifying an *N*-by-3 array of Euler angles in radians or degrees.

Euler Angles in Radians

Use the `euler` syntax to create a scalar quaternion using a 1-by-3 vector of Euler angles in radians. Specify the rotation sequence of the Euler angles and whether the angles represent a frame or point rotation.

```
E = [pi/2,0,pi/4];
quat = quaternion(E,'euler','ZYX','frame')
```

```
quat = quaternion
     0.65328 +  0.2706i +  0.2706j + 0.65328k
```

You can convert from quaternions to Euler angles using the `euler` function. Recover the Euler angles, `E`, from the quaternion, `quat`.

```
euler(quat,'ZYX','frame')
```

```
ans = 1×3

    1.5708         0    0.7854
```

Euler Angles in Degrees

Use the `eulerd` syntax to create a scalar quaternion using a 1-by-3 vector of Euler angles in degrees. Specify the rotation sequence of the Euler angles and whether the angles represent a frame or point rotation.

```
E = [90,0,45];
quat = quaternion(E,'eulerd','ZYX','frame')
```

```
quat = quaternion
     0.65328 +  0.2706i +  0.2706j + 0.65328k
```

You can convert from quaternions to Euler angles in degrees using the `eulerd` function. Recover the Euler angles, `E`, from the quaternion, `quat`.

```
eulerd(quat,'ZYX','frame')
```

```
ans = 1×3

   90.0000         0   45.0000
```

Quaternion Algebra

Quaternions form a noncommutative associative algebra over the real numbers. This example illustrates the rules of quaternion algebra.

Addition and Subtraction

Quaternion addition and subtraction occur part-by-part, and are commutative:

```
Q1 = quaternion(1,2,3,4)
```

```
Q1 = quaternion
     1 + 2i + 3j + 4k
```

```
Q2 = quaternion(9,8,7,6)
```

```
Q2 = quaternion
     9 + 8i + 7j + 6k
```

```
Q1plusQ2 = Q1 + Q2
```

```
Q1plusQ2 = quaternion
     10 + 10i + 10j + 10k
```

```
Q2plusQ1 = Q2 + Q1
```

```
Q2plusQ1 = quaternion
     10 + 10i + 10j + 10k
```

```
Q1minusQ2 = Q1 - Q2
```

```
Q1minusQ2 = quaternion
    -8 - 6i - 4j - 2k
```

```
Q2minusQ1 = Q2 - Q1
```

```
Q2minusQ1 = quaternion
     8 + 6i + 4j + 2k
```

You can also perform addition and subtraction of real numbers and quaternions. The first part of a quaternion is referred to as the *real* part, while the second, third, and fourth parts are referred to as the *vector*. Addition and subtraction with real numbers affect only the real part of the quaternion.

```
Q1plusRealNumber = Q1 + 5
```

```
Q1plusRealNumber = quaternion
     6 + 2i + 3j + 4k
```

```
Q1minusRealNumber = Q1 - 5
```

```
Q1minusRealNumber = quaternion
    -4 + 2i + 3j + 4k
```

Multiplication

Quaternion multiplication is determined by the products of the basis elements and the distributive law. Recall that multiplication of the basis elements, *i*, *j*, and *k*, are not commutative, and therefore quaternion multiplication is not commutative.

```
Q1timesQ2 = Q1 * Q2
```

```
Q1timesQ2 = quaternion
    -52 + 16i + 54j + 32k
```

```
Q2timesQ1 = Q2 * Q1
```

```
Q2timesQ1 = quaternion
    -52 + 36i + 14j + 52k
```

```
isequal(Q1timesQ2,Q2timesQ1)
```

```
ans = logical
   0
```

You can also multiply a quaternion by a real number. If you multiply a quaternion by a real number, each part of the quaternion is multiplied by the real number individually:

```
Q1times5 = Q1*5
```

```
Q1times5 = quaternion
      5 + 10i + 15j + 20k
```

Multiplying a quaternion by a real number is commutative.

```
isequal(Q1*5,5*Q1)
```

```
ans = logical
   1
```

Conjugation

The complex conjugate of a quaternion is defined such that each element of the vector portion of the quaternion is negated.

```
Q1
```

```
Q1 = quaternion
     1 + 2i + 3j + 4k
```

```
conj(Q1)
```

```
ans = quaternion
     1 - 2i - 3j - 4k
```

Multiplication between a quaternion and its conjugate is commutative:

```
isequal(Q1*conj(Q1),conj(Q1)*Q1)
```

```
ans = logical
   1
```

Quaternion Array Manipulation

You can organize quaternions into vectors, matrices, and multidimensional arrays. Built-in MATLAB® functions have been enhanced to work with quaternions.

Concatenate

Quaternions are treated as individual objects during concatenation and follow MATLAB rules for array manipulation.

```
Q1 = quaternion(1,2,3,4);
Q2 = quaternion(9,8,7,6);

qVector = [Q1,Q2]
```

```
qVector = 1x2 quaternion array
     1 + 2i + 3j + 4k     9 + 8i + 7j + 6k
```

```
Q3 = quaternion(-1,-2,-3,-4);
Q4 = quaternion(-9,-8,-7,-6);

qMatrix = [qVector;Q3,Q4]
```

```
qMatrix = 2x2 quaternion array
     1 + 2i + 3j + 4k     9 + 8i + 7j + 6k
```

```
    -1 - 2i - 3j - 4k    -9 - 8i - 7j - 6k
```

```
qMultiDimensionalArray(:,:,1) = qMatrix;
qMultiDimensionalArray(:,:,2) = qMatrix
```

```
qMultiDimensionalArray = 2x2x2 quaternion array
qMultiDimensionalArray(:,:,1) =

     1 + 2i + 3j + 4k     9 + 8i + 7j + 6k
    -1 - 2i - 3j - 4k    -9 - 8i - 7j - 6k


qMultiDimensionalArray(:,:,2) =

     1 + 2i + 3j + 4k     9 + 8i + 7j + 6k
    -1 - 2i - 3j - 4k    -9 - 8i - 7j - 6k
```

Indexing

To access or assign elements in a quaternion array, use indexing.

```
qLoc2 = qMultiDimensionalArray(2)
```

```
qLoc2 = quaternion
    -1 - 2i - 3j - 4k
```

Replace the quaternion at index two with a quaternion one.

```
qMultiDimensionalArray(2) = ones('quaternion')
```

```
qMultiDimensionalArray = 2x2x2 quaternion array
qMultiDimensionalArray(:,:,1) =

     1 + 2i + 3j + 4k     9 + 8i + 7j + 6k
     1 + 0i + 0j + 0k    -9 - 8i - 7j - 6k


qMultiDimensionalArray(:,:,2) =

     1 + 2i + 3j + 4k     9 + 8i + 7j + 6k
    -1 - 2i - 3j - 4k    -9 - 8i - 7j - 6k
```

Reshape

To reshape quaternion arrays, use the `reshape` function.

```
qMatReshaped = reshape(qMatrix,4,1)
```

```
qMatReshaped = 4x1 quaternion array
     1 + 2i + 3j + 4k
    -1 - 2i - 3j - 4k
     9 + 8i + 7j + 6k
    -9 - 8i - 7j - 6k
```

Transpose

To transpose quaternion vectors and matrices, use the `transpose` function.

```
qMatTransposed = transpose(qMatrix)

qMatTransposed = 2x2 quaternion array
     1 + 2i + 3j + 4k    -1 - 2i - 3j - 4k
     9 + 8i + 7j + 6k    -9 - 8i - 7j - 6k
```

Permute

To permute quaternion vectors, matrices, and multidimensional arrays, use the `permute` function.

```
qMultiDimensionalArray

qMultiDimensionalArray = 2x2x2 quaternion array
qMultiDimensionalArray(:,:,1) =

     1 + 2i + 3j + 4k     9 + 8i + 7j + 6k
     1 + 0i + 0j + 0k    -9 - 8i - 7j - 6k


qMultiDimensionalArray(:,:,2) =

     1 + 2i + 3j + 4k     9 + 8i + 7j + 6k
    -1 - 2i - 3j - 4k    -9 - 8i - 7j - 6k


qMatPermute = permute(qMultiDimensionalArray,[3,1,2])

qMatPermute = 2x2x2 quaternion array
qMatPermute(:,:,1) =

     1 + 2i + 3j + 4k     1 + 0i + 0j + 0k
     1 + 2i + 3j + 4k    -1 - 2i - 3j - 4k


qMatPermute(:,:,2) =

     9 + 8i + 7j + 6k    -9 - 8i - 7j - 6k
     9 + 8i + 7j + 6k    -9 - 8i - 7j - 6k
```

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

Topics

“Rotations, Orientation, and Quaternions”
“Lowpass Filter Orientation Using Quaternion SLERP”

Introduced in R2019b

angvel

Angular velocity from quaternion array

Syntax

```
AV = angvel(Q,dt,'frame')
AV = angvel(Q,dt,'point')
[AV,qf] = angvel(Q,dt,fp,qi)
```

Description

`AV = angvel(Q,dt,'frame')` returns the angular velocity array from an array of quaternions, `Q`. The quaternions in `Q` correspond to frame rotation. The initial quaternion is assumed to represent zero rotation.

`AV = angvel(Q,dt,'point')` returns the angular velocity array from an array of quaternions, `Q`. The quaternions in `Q` correspond to point rotation. The initial quaternion is assumed to represent zero rotation.

`[AV,qf] = angvel(Q,dt,fp,qi)` allows you to specify the initial quaternion, `qi`, and the type of rotation, `fp`. It also returns the final quaternion, `qf`.

Examples

Generate Angular Velocity From Quaternion Array

Create an array of quaternions.

```
eulerAngles = [(0:10:90).',zeros(numel(0:10:90),2)];
q = quaternion(eulerAngles,'eulerd','ZYX','frame');
```

Specify the time step and generate the angular velocity array.

```
dt = 1;
av = angvel(q,dt,'frame') % units in rad/s
```

```
av = 10×3

         0         0         0
         0         0    0.1743
         0         0    0.1743
         0         0    0.1743
         0         0    0.1743
         0         0    0.1743
         0         0    0.1743
         0         0    0.1743
         0         0    0.1743
         0         0    0.1743
```

Input Arguments

`Q` — Quaternions
N-by-1 vector of quaternions

Quaternions, specified as an *N*-by-1 vector of quaternions.

Data Types: `quaternion`

`dt` — Time step
nonnegative scalar

Time step, specified as a nonnegative scalar.

Data Types: `single` | `double`

`fp` — Type of rotation
`'frame'` | `'point'`

Type of rotation, specified as `'frame'` or `'point'`.

`qi` — Initial quaternion
quaternion

Initial quaternion, specified as a quaternion.

Data Types: `quaternion`

Output Arguments

`AV` — Angular velocity
N-by-3 real matrix

Angular velocity, returned as an *N*-by-3 real matrix. *N* is the number of quaternions given in the input *Q*. Each row of the matrix corresponds to an angular velocity vector.

`qf` — Final quaternion
quaternion

Final quaternion, returned as a quaternion. `qf` is the same as the last quaternion in the *Q* input.

Data Types: `quaternion`

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also
`quaternion`

Topics
“Rotations, Orientation, and Quaternions”

Introduced in R2020a

classUnderlying

Class of parts within quaternion

Syntax

```
underlyingClass = classUnderlying(quat)
```

Description

`underlyingClass = classUnderlying(quat)` returns the name of the class of the parts of the quaternion `quat`.

Examples

Get Underlying Class of Quaternion

A quaternion is a four-part hyper-complex number used in three-dimensional representations. The four parts of the quaternion are of data type `single` or `double`.

Create two quaternions, one with an underlying data type of `single`, and one with an underlying data type of `double`. Verify the underlying data types by calling `classUnderlying` on the quaternions.

```
qSingle = quaternion(single([1,2,3,4]))
```

```
qSingle = quaternion
     1 + 2i + 3j + 4k
```

```
classUnderlying(qSingle)
```

```
ans =
'single'
```

```
qDouble = quaternion([1,2,3,4])
```

```
qDouble = quaternion
     1 + 2i + 3j + 4k
```

```
classUnderlying(qDouble)
```

```
ans =
'double'
```

You can separate quaternions into their parts using the `parts` function. Verify the parts of each quaternion are the correct data type. Recall that `double` is the default MATLAB® type.

```
[aS,bS,cS,dS] = parts(qSingle)
```

```
aS = single
    1
```

```
bS = single
     2

cS = single
     3

dS = single
     4

[aD,bD,cD,dD] = parts(qDouble)

aD = 1

bD = 2

cD = 3

dD = 4
```

Quaternions follow the same implicit casting rules as other data types in MATLAB. That is, a quaternion with underlying data type `single` that is combined with a quaternion with underlying data type `double` results in a quaternion with underlying data type `single`. Multiply `qDouble` and `qSingle` and verify the resulting underlying data type is `single`.

```
q = qDouble*qSingle;
classUnderlying(q)

ans =
'single'
```

Input Arguments

`quat` — Quaternion to investigate

scalar | vector | matrix | multi-dimensional array

Quaternion to investigate, specified as a quaternion or array of quaternions.

Data Types: `quaternion`

Output Arguments

`underlyingClass` — Underlying class of quaternion object

`'single'` | `'double'`

Underlying class of quaternion, returned as the character vector `'single'` or `'double'`.

Data Types: `char`

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

Functions
`compact` | `parts`

Objects
`quaternion`

Topics
“Rotations, Orientation, and Quaternions”

Introduced in R2019b

compact

Convert quaternion array to *N*-by-4 matrix

Syntax

```
matrix = compact(quat)
```

Description

`matrix = compact(quat)` converts the quaternion array, `quat`, to an *N*-by-4 matrix. The columns are made from the four quaternion parts. The i^{th} row of the matrix corresponds to `quat(i)`.

Examples

Convert Quaternion Array to Compact Representation of Parts

Create a scalar quaternion with random parts. Convert the parts to a 1-by-4 vector using `compact`.

```
randomParts = randn(1,4)
```

```
randomParts = 1×4

    0.5377    1.8339   -2.2588    0.8622
```

```
quat = quaternion(randomParts)
```

```
quat = quaternion
     0.53767 +  1.8339i -  2.2588j + 0.86217k
```

```
quatParts = compact(quat)
```

```
quatParts = 1×4

    0.5377    1.8339   -2.2588    0.8622
```

Create a 2-by-2 array of quaternions, then convert the representation to a matrix of quaternion parts. The output rows correspond to the linear indices of the quaternion array.

```
quatArray = [quaternion([1:4;5:8]),quaternion([9:12;13:16])]
```

```
quatArray = 2x2 quaternion array
      1 +  2i +  3j +  4k      9 + 10i + 11j + 12k
      5 +  6i +  7j +  8k     13 + 14i + 15j + 16k
```

```
quatArrayParts = compact(quatArray)
```

```
quatArrayParts = 4×4
```

```
     1     2     3     4
     5     6     7     8
     9    10    11    12
    13    14    15    16
```

Input Arguments

`quat` — Quaternion to convert

scalar | vector | matrix | multidimensional array

Quaternion to convert, specified as scalar, vector, matrix, or multidimensional array of quaternions.

Data Types: `quaternion`

Output Arguments

`matrix` — Quaternion in matrix form

N-by-4 matrix

Quaternion in matrix form, returned as an *N*-by-4 matrix, where N = `numel(quat)`.

Data Types: `single` | `double`

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

Functions

`parts` | `classUnderlying`

Objects

`quaternion`

Topics

“Rotations, Orientation, and Quaternions”

Introduced in R2019b

conj

Complex conjugate of quaternion

Syntax

```
quatConjugate = conj(quat)
```

Description

`quatConjugate = conj(quat)` returns the complex conjugate of the quaternion, `quat`.

If $q = a + b\mathrm{i} + c\mathrm{j} + d\mathrm{k}$, the complex conjugate of q is $q^* = a - b\mathrm{i} - c\mathrm{j} - d\mathrm{k}$. Considered as a rotation operator, the conjugate performs the opposite rotation. For example,

```
q = quaternion(deg2rad([16 45 30]),'rotvec');
a = q*conj(q);
rotatepoint(a,[0,1,0])
```

```
ans =

     0     1     0
```

Examples

Complex Conjugate of Quaternion

Create a quaternion scalar and get the complex conjugate.

```
q = normalize(quaternion([0.9 0.3 0.3 0.25]))
```

```
q = quaternion
     0.87727 + 0.29242i + 0.29242j + 0.24369k
```

```
qConj = conj(q)
```

```
qConj = quaternion
     0.87727 - 0.29242i - 0.29242j - 0.24369k
```

Verify that a quaternion multiplied by its conjugate returns a quaternion one.

```
q*qConj
```

```
ans = quaternion
     1 + 0i + 0j + 0k
```

Input Arguments

`quat` — Quaternion

scalar | vector | matrix | multidimensional array

Quaternion to conjugate, specified as a scalar, vector, matrix, or array of quaternions.

Data Types: `quaternion`

Output Arguments

`quatConjugate` — Quaternion conjugate

scalar | vector | matrix | multidimensional array

Quaternion conjugate, returned as a quaternion or array of quaternions the same size as `quat`.

Data Types: `quaternion`

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

Functions

`norm` | `.*,times`

Objects

`quaternion`

Topics

“Rotations, Orientation, and Quaternions”

Introduced in R2019b

ctranspose, '

Complex conjugate transpose of quaternion array

Syntax

```
quatTransposed = quat'
```

Description

`quatTransposed = quat'` returns the complex conjugate transpose of the quaternion, `quat`.

Examples

Vector Complex Conjugate Transpose

Create a vector of quaternions and compute its complex conjugate transpose.

```
quat = quaternion(randn(4,4))
```

```
quat = 4x1 quaternion array
      0.53767 +  0.31877i +   3.5784j +   0.7254k
       1.8339 -   1.3077i +   2.7694j - 0.063055k
      -2.2588 -  0.43359i -   1.3499j +  0.71474k
      0.86217 +  0.34262i +   3.0349j -  0.20497k
```

```
quatTransposed = quat'
```

```
quatTransposed = 1x4 quaternion array
      0.53767 -  0.31877i -   3.5784j -   0.7254k       1.8339 +   1.3077i -   2.7694j + 0.06305
```

Matrix Complex Conjugate Transpose

Create a matrix of quaternions and compute its complex conjugate transpose.

```
quat = [quaternion(randn(2,4)),quaternion(randn(2,4))]
```

```
quat = 2x2 quaternion array
      0.53767 -   2.2588i +  0.31877j -  0.43359k       3.5784 -   1.3499i +   0.7254j +  0.7147
       1.8339 +  0.86217i -   1.3077j +  0.34262k       2.7694 +   3.0349i - 0.063055j -  0.2049
```

```
quatTransposed = quat'
```

```
quatTransposed = 2x2 quaternion array
      0.53767 +   2.2588i -  0.31877j +  0.43359k       1.8339 -  0.86217i +   1.3077j -  0.3426
       3.5784 +   1.3499i -   0.7254j -  0.71474k       2.7694 -   3.0349i + 0.063055j +  0.2049
```

Input Arguments

`quat` — Quaternion to transpose

scalar | vector | matrix

Quaternion to transpose, specified as a vector or matrix or quaternions. The complex conjugate transpose is defined for 1-D and 2-D arrays.

Data Types: `quaternion`

Output Arguments

`quatTransposed` — Conjugate transposed quaternion

scalar | vector | matrix

Conjugate transposed quaternion, returned as an *N*-by-*M* array, where `quat` was specified as an *M*-by-*N* array.

Data Types: `quaternion`

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

Functions

`transpose, '`

Objects

`quaternion`

Topics

“Rotations, Orientation, and Quaternions”

Introduced in R2019b

dist

Angular distance in radians

Syntax

```
distance = dist(quatA,quatB)
```

Description

`distance = dist(quatA,quatB)` returns the angular distance in radians between two quaternions, `quatA` and `quatB`.

Examples

Calculate Quaternion Distance

Calculate the quaternion distance between a single quaternion and each element of a vector of quaternions. Define the quaternions using Euler angles.

```
q = quaternion([0,0,0],'eulerd','zyx','frame')
```

```
q = quaternion
     1 + 0i + 0j + 0k
```

```
qArray = quaternion([0,45,0;0,90,0;0,180,0;0,-90,0;0,-45,0],'eulerd','zyx','frame')
```

```
qArray = 5x1 quaternion array
       0.92388 +         0i +   0.38268j +         0k
       0.70711 +         0i +   0.70711j +         0k
    6.1232e-17 +         0i +         1j +         0k
       0.70711 +         0i -   0.70711j +         0k
       0.92388 +         0i -   0.38268j +         0k
```

```
quaternionDistance = rad2deg(dist(q,qArray))
```

```
quaternionDistance = 5×1

   45.0000
   90.0000
  180.0000
   90.0000
   45.0000
```

If both arguments to `dist` are vectors, the quaternion distance is calculated between corresponding elements. Calculate the quaternion distance between two quaternion vectors.

```
angles1 = [30,0,15; ...
           30,5,15; ...
```

```
           30,10,15; ...
           30,15,15];
angles2 = [30,6,15; ...
           31,11,15; ...
           30,16,14; ...
           30.5,21,15.5];

qVector1 = quaternion(angles1,'eulerd','zyx','frame');
qVector2 = quaternion(angles2,'eulerd','zyx','frame');

rad2deg(dist(qVector1,qVector2))

ans = 4×1

    6.0000
    6.0827
    6.0827
    6.0287
```

Note that a quaternion represents the same rotation as its negative. Calculate a quaternion and its negative.

```
qPositive = quaternion([30,45,-60],'eulerd','zyx','frame')

qPositive = quaternion
     0.72332 - 0.53198i + 0.20056j +  0.3919k


qNegative = -qPositive

qNegative = quaternion
    -0.72332 + 0.53198i - 0.20056j -  0.3919k
```

Find the distance between the quaternion and its negative.

```
dist(qPositive,qNegative)

ans = 0
```

The components of a quaternion may look different from the components of its negative, but both expressions represent the same rotation.

Input Arguments

`quatA,quatB` — Quaternions to calculate distance between

scalar | vector | matrix | multidimensional array

Quaternions to calculate distance between, specified as comma-separated quaternions or arrays of quaternions. `quatA` and `quatB` must have compatible sizes:

- `size(quatA) == size(quatB)`, or
- `numel(quatA) == 1`, or
- `numel(quatB) == 1`, or

- if `[Adim1,…,AdimN] = size(quatA)` and `[Bdim1,…,BdimN] = size(quatB)`, then for `i = 1:N`, either `Adimi==Bdimi` or `Adim==1` or `Bdim==1`.

  If one of the quaternion arguments contains only one quaternion, then this function returns the distances between that quaternion and every quaternion in the other argument.

Data Types: `quaternion`

Output Arguments

`distance` — Angular distance (radians)

scalar | vector | matrix | multidimensional array

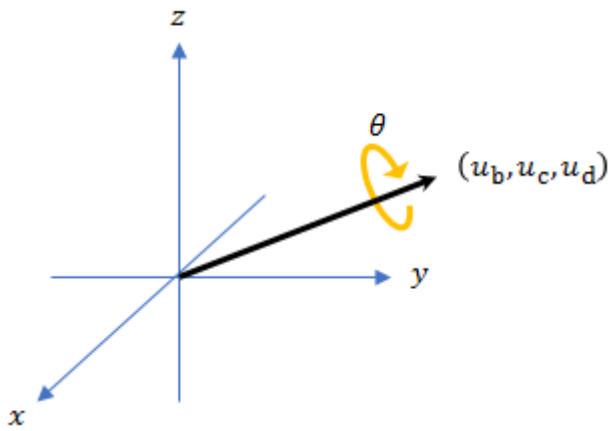
Angular distance in radians, returned as an array. The dimensions are the maximum of the union of `size(quatA)` and `size(quatB)`.

Data Types: `single` | `double`

Algorithms

The `dist` function returns the angular distance between two quaternions.

A quaternion may be defined by an axis (u_b,u_c,u_d) and angle of rotation θ_q:
$q = \cos\left(\theta_q/2\right) + \sin\left(\theta_q/2\right)(u_b\mathrm{i} + u_c\mathrm{j} + u_d\mathrm{k})$.



Given a quaternion in the form, $q = a + b\mathrm{i} + c\mathrm{j} + d\mathrm{k}$, where a is the real part, you can solve for the angle of q as $\theta_q = 2\cos^{-1}(a)$.

Consider two quaternions, p and q, and the product $z = p*\text{conjugate}(q)$. As p approaches q, the angle of z goes to 0, and z approaches the unit quaternion.

The angular distance between two quaternions can be expressed as $\theta_z = 2\cos^{-1}(\text{real}(z))$.

Using the `quaternion` data type syntax, the angular distance is calculated as:

```
angularDistance = 2*acos(abs(parts(p*conj(q))));
```

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

Functions

`parts` | `conj`

Objects

`quaternion`

Topics

“Rotations, Orientation, and Quaternions”

Introduced in R2019b

euler

Convert quaternion to Euler angles (radians)

Syntax

```
eulerAngles = euler(quat,rotationSequence,rotationType)
```

Description

`eulerAngles = euler(quat,rotationSequence,rotationType)` converts the quaternion, `quat`, to an *N*-by-3 matrix of Euler angles.

Examples

Convert Quaternion to Euler Angles in Radians

Convert a quaternion frame rotation to Euler angles in radians using the 'ZYX' rotation sequence.

```
quat = quaternion([0.7071 0.7071 0 0]);
eulerAnglesRandians = euler(quat,'ZYX','frame')
```

```
eulerAnglesRandians = 1×3

         0         0    1.5708
```

Input Arguments

`quat` — Quaternion to convert to Euler angles

scalar | vector | matrix | multidimensional array

Quaternion to convert to Euler angles, specified as a scalar, vector, matrix, or multidimensional array of quaternions.

Data Types: `quaternion`

`rotationSequence` — Rotation sequence

`'ZYX'` | `'ZYZ'` | `'ZXY'` | `'ZXZ'` | `'YXZ'` | `'YXY'` | `'YZX'` | `'XYZ'` | `'XYX'` | `'XZY'` | `'XZX'`

Rotation sequence of Euler representation, specified as a character vector or string.

The rotation sequence defines the order of rotations about the axes. For example, if you specify a rotation sequence of `'YZX'`:

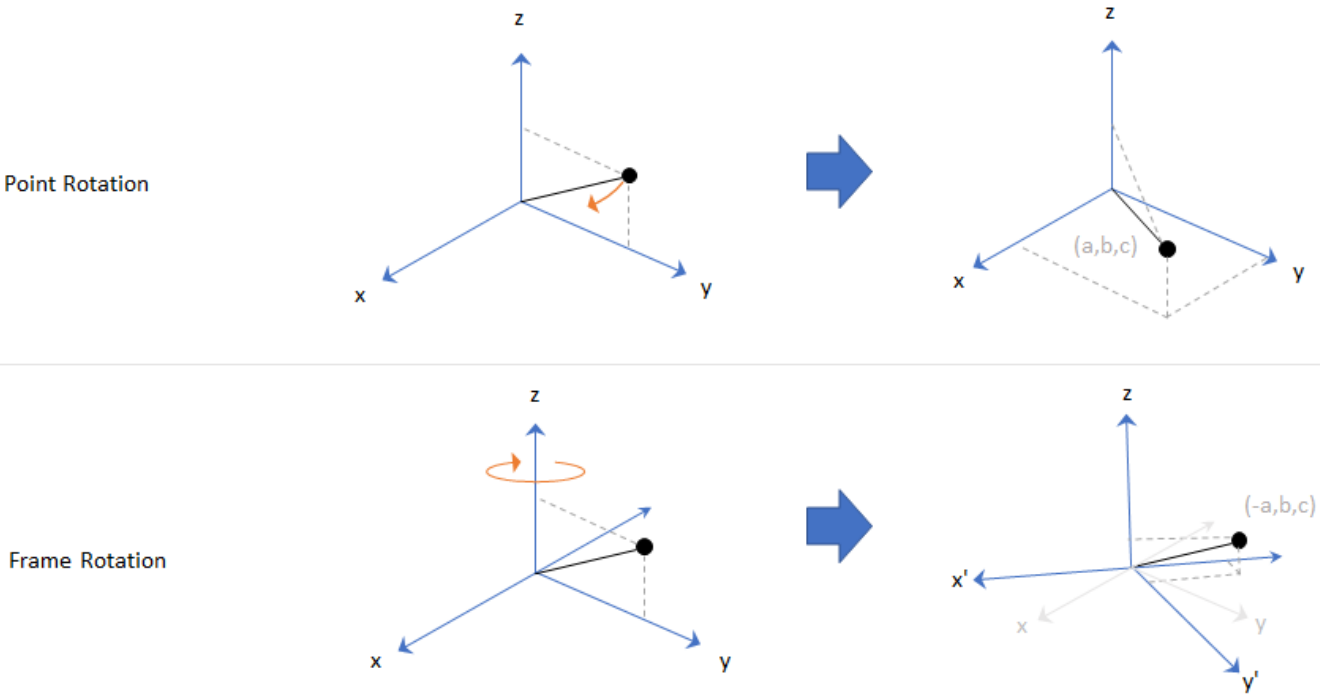
1. The first rotation is about the y-axis.
2. The second rotation is about the new z-axis.
3. The third rotation is about the new x-axis.

Data Types: `char` | `string`

`rotationType` — Type of rotation
`'point'` | `'frame'`

Type of rotation, specified as `'point'` or `'frame'`.

In a point rotation, the frame is static and the point moves. In a frame rotation, the point is static and the frame moves. Point rotation and frame rotation define equivalent angular displacements but in opposite directions.



Data Types: `char` | `string`

Output Arguments

`eulerAngles` — Euler angle representation (radians)
N-by-3 matrix

Euler angle representation in radians, returned as a *N*-by-3 matrix. *N* is the number of quaternions in the `quat` argument.

For each row of `eulerAngles`, the first element corresponds to the first axis in the rotation sequence, the second element corresponds to the second axis in the rotation sequence, and the third element corresponds to the third axis in the rotation sequence.

The data type of the Euler angles representation is the same as the underlying data type of `quat`.

Data Types: `single` | `double`

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

Functions

`eulerd` | `rotateframe` | `rotatepoint`

Objects

`quaternion`

Topics

“Rotations, Orientation, and Quaternions”

Introduced in R2019b

eulerd

Convert quaternion to Euler angles (degrees)

Syntax

```
eulerAngles = eulerd(quat,rotationSequence,rotationType)
```

Description

`eulerAngles = eulerd(quat,rotationSequence,rotationType)` converts the quaternion, `quat`, to an *N*-by-3 matrix of Euler angles in degrees.

Examples

Convert Quaternion to Euler Angles in Degrees

Convert a quaternion frame rotation to Euler angles in degrees using the 'ZYX' rotation sequence.

```
quat = quaternion([0.7071 0.7071 0 0]);
eulerAnglesDegrees = eulerd(quat,'ZYX','frame')
```

```
eulerAnglesDegrees = 1×3

         0         0   90.0000
```

Input Arguments

`quat` — Quaternion to convert to Euler angles

scalar | vector | matrix | multidimensional array

Quaternion to convert to Euler angles, specified as a scalar, vector, matrix, or multidimensional array of quaternions.

Data Types: `quaternion`

`rotationSequence` — Rotation sequence

`'ZYX'` | `'ZYZ'` | `'ZXY'` | `'ZXZ'` | `'YXZ'` | `'YXY'` | `'YZX'` | `'XYZ'` | `'XYX'` | `'XZY'` | `'XZX'`

Rotation sequence of Euler angle representation, specified as a character vector or string.

The rotation sequence defines the order of rotations about the axes. For example, if you specify a rotation sequence of `'YZX'`:

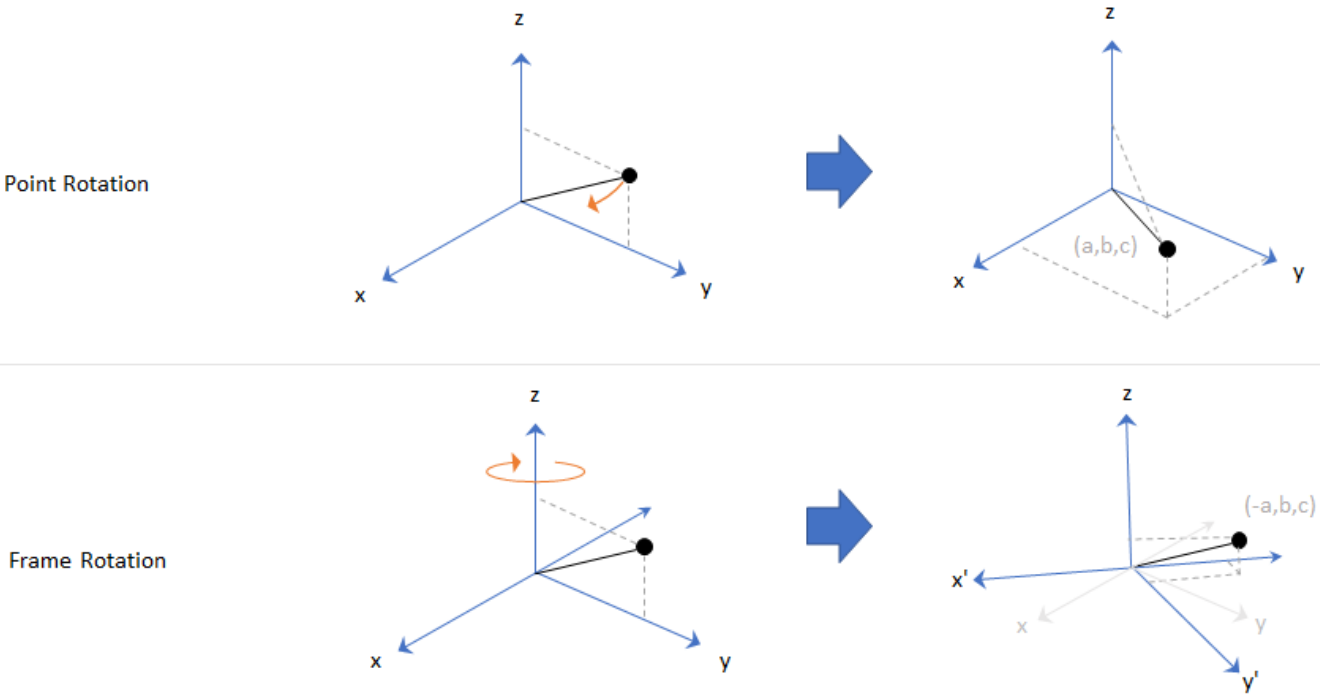
1. The first rotation is about the *y*-axis.
2. The second rotation is about the new *z*-axis.
3. The third rotation is about the new *x*-axis.

Data Types: `char` | `string`

`rotationType` — Type of rotation
`'point'` | `'frame'`

Type of rotation, specified as `'point'` or `'frame'`.

In a point rotation, the frame is static and the point moves. In a frame rotation, the point is static and the frame moves. Point rotation and frame rotation define equivalent angular displacements but in opposite directions.



Data Types: `char` | `string`

Output Arguments

`eulerAngles` — Euler angle representation (degrees)
N-by-3 matrix

Euler angle representation in degrees, returned as a *N*-by-3 matrix. *N* is the number of quaternions in the `quat` argument.

For each row of `eulerAngles`, the first column corresponds to the first axis in the rotation sequence, the second column corresponds to the second axis in the rotation sequence, and the third column corresponds to the third axis in the rotation sequence.

The data type of the Euler angles representation is the same as the underlying data type of `quat`.

Data Types: `single` | `double`

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

Functions

`euler` | `rotateframe` | `rotatepoint`

Objects

`quaternion`

Topics

“Rotations, Orientation, and Quaternions”

Introduced in R2019b

exp

Exponential of quaternion array

Syntax

```
B = exp(A)
```

Description

`B = exp(A)` computes the exponential of the elements of the quaternion array `A`.

Examples

Exponential of Quaternion Array

Create a 4-by-1 quaternion array A.

```
A = quaternion(magic(4))
```

```
A = 4x1 quaternion array
     16 +  2i +  3j + 13k
      5 + 11i + 10j +  8k
      9 +  7i +  6j + 12k
      4 + 14i + 15j +  1k
```

Compute the exponential of A.

```
B = exp(A)
```

```
B = 4x1 quaternion array
     5.3525e+06 + 1.0516e+06i + 1.5774e+06j + 6.8352e+06k
        -57.359 -     89.189i -     81.081j -     64.865k
        -6799.1 +     2039.1i +     1747.8j +     3495.6k
          -6.66 +     36.931i +     39.569j +     2.6379k
```

Input Arguments

A — Input quaternion

scalar | vector | matrix | multidimensional array

Input quaternion, specified as a scalar, vector, matrix, or multidimensional array.

Data Types: `quaternion`

Output Arguments

B — Result

scalar | vector | matrix | multidimensional array

Result of quaternion exponential, returned as a scalar, vector, matrix, or multidimensional array.

Data Types: `quaternion`

Algorithms

Given a quaternion $A = a + b\mathrm{i} + c\mathrm{j} + d\mathrm{k} = a + \bar{v}$, the exponential is computed by

$$\exp(A) = e^{a}\left(\cos\|\bar{v}\| + \frac{\bar{v}}{\|\bar{v}\|}\sin\|\bar{v}\|\right)$$

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

Functions

`.^,power` | `log`

Objects

`quaternion`

Topics

“Rotations, Orientation, and Quaternions”

Introduced in R2019b

ldivide, .\

Element-wise quaternion left division

Syntax

```
C = A.\B
```

Description

`C = A.\B` performs quaternion element-wise division by dividing each element of quaternion `B` by the corresponding element of quaternion `A`.

Examples

Divide a Quaternion Array by a Real Scalar

Create a 2-by-1 quaternion array, and divide it element-by-element by a real scalar.

```
A = quaternion([1:4;5:8])
```

```
A = 2x1 quaternion array
     1 + 2i + 3j + 4k
     5 + 6i + 7j + 8k
```

```
B = 2;
C = A.\B
```

```
C = 2x1 quaternion array
     0.066667 -  0.13333i -      0.2j -  0.26667k
     0.057471 - 0.068966i -  0.08046j - 0.091954k
```

Divide a Quaternion Array by Another Quaternion Array

Create a 2-by-2 quaternion array, and divide it element-by-element by another 2-by-2 quaternion array.

```
q1 = quaternion([1:4;2:5;4:7;5:8]);
A = reshape(q1,2,2)
```

```
A = 2x2 quaternion array
     1 + 2i + 3j + 4k     4 + 5i + 6j + 7k
     2 + 3i + 4j + 5k     5 + 6i + 7j + 8k
```

```
q2 = quaternion(magic(4));
B = reshape(q2,2,2)
```

```
B = 2x2 quaternion array
     16 +  2i +  3j + 13k      9 +  7i +  6j + 12k
      5 + 11i + 10j +  8k      4 + 14i + 15j +  1k


C = A.\B

C = 2x2 quaternion array
          2.7 -      1.9i -      0.9j -      1.7k       1.5159 -  0.37302i -  0.15079j -  0.0238
       2.2778 +  0.46296i -  0.57407j + 0.092593k       1.2471 +  0.91379i -  0.33908j -   0.109
```

Input Arguments

A — Divisor

scalar | vector | matrix | multidimensional array

Divisor, specified as a quaternion, an array of quaternions, a real scalar, or an array of real numbers.

`A` and `B` must have compatible sizes. In the simplest cases, they can be the same size or one can be a scalar. Two inputs have compatible sizes if, for every dimension, the dimension sizes of the inputs are the same or one of the dimensions is 1.

Data Types: `quaternion` | `single` | `double`

B — Dividend

scalar | vector | matrix | multidimensional array

Dividend, specified as a quaternion, an array of quaternions, a real scalar, or an array of real numbers.

`A` and `B` must have compatible sizes. In the simplest cases, they can be the same size or one can be a scalar. Two inputs have compatible sizes if, for every dimension, the dimension sizes of the inputs are the same or one of the dimensions is 1.

Data Types: `quaternion` | `single` | `double`

Output Arguments

C — Result

scalar | vector | matrix | multidimensional array

Result of quaternion division, returned as a scalar, vector, matrix, or multidimensional array.

Data Types: `quaternion`

Algorithms

Quaternion Division

Given a quaternion $A = a_1 + a_2\mathrm{i} + a_3\mathrm{j} + a_4\mathrm{k}$ and a real scalar p,

$$C = p.\backslash A = \frac{a_1}{p} + \frac{a_2}{p}\mathrm{i} + \frac{a_3}{p}\mathrm{j} + \frac{a_4}{p}\mathrm{k}$$

Note For a real scalar p, $A./p = A.\backslash p$.

Quaternion Division by a Quaternion Scalar

Given two quaternions A and B of compatible sizes, then

$$C = A.\backslash B = A^{-1}.*B = \left(\frac{conj(A)}{norm(A)^2}\right).*B$$

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

Functions

`.*,times` | `conj` | `norm` | `./,ldivide`

Objects

`quaternion`

Topics

“Rotations, Orientation, and Quaternions”

Introduced in R2019b

log

Natural logarithm of quaternion array

Syntax

```
B = log(A)
```

Description

`B = log(A)` computes the natural logarithm of the elements of the quaternion array `A`.

Examples

Logarithmic Values of Quaternion Array

Create a 3-by-1 quaternion array A.

```
A = quaternion(randn(3,4))
```

```
A = 3x1 quaternion array
     0.53767 + 0.86217i - 0.43359j +  2.7694k
      1.8339 + 0.31877i + 0.34262j -  1.3499k
     -2.2588 -  1.3077i +  3.5784j +  3.0349k
```

Compute the logarithmic values of A.

```
B = log(A)
```

```
B = 3x1 quaternion array
       1.0925 + 0.40848i - 0.20543j +  1.3121k
       0.8436 + 0.14767i + 0.15872j - 0.62533k
       1.6807 - 0.53829i +   1.473j +  1.2493k
```

Input Arguments

`A` — Input array

scalar | vector | matrix | multidimensional array

Input array, specified as a scalar, vector, matrix, or multidimensional array.

Data Types: `quaternion`

Output Arguments

`B` — Logarithm values

scalar | vector | matrix | multidimensional array

Quaternion natural logarithm values, returned as a scalar, vector, matrix, or multidimensional array.

Data Types: `quaternion`

Algorithms

Given a quaternion $A = a + \bar{v} = a + b\mathrm{i} + c\mathrm{j} + d\mathrm{k}$, the logarithm is computed by

$$\log(A) = \log\|A\| + \frac{\bar{v}}{\|\bar{v}\|}\arccos\frac{a}{\|A\|}$$

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

Functions

`exp` | `.^,power`

Objects

`quaternion`

Topics

“Rotations, Orientation, and Quaternions”

Introduced in R2019b

meanrot

Quaternion mean rotation

Syntax

```
quatAverage = meanrot(quat)
quatAverage = meanrot(quat,dim)
quatAverage = meanrot( ___ ,nanflag)
```

Description

`quatAverage = meanrot(quat)` returns the average rotation of the elements of `quat` along the first array dimension whose size not does equal 1.

- If `quat` is a vector, `meanrot(quat)` returns the average rotation of the elements.
- If `quat` is a matrix, `meanrot(quat)` returns a row vector containing the average rotation of each column.
- If `quat` is a multidimensional array, then `mearot(quat)` operates along the first array dimension whose size does not equal 1, treating the elements as vectors. This dimension becomes 1 while the sizes of all other dimensions remain the same.

The `meanrot` function normalizes the input quaternions, `quat`, before calculating the mean.

`quatAverage = meanrot(quat,dim)` return the average rotation along dimension `dim`. For example, if `quat` is a matrix, then `meanrot(quat,2)` is a column vector containing the mean of each row.

`quatAverage = meanrot( ___ ,nanflag)` specifies whether to include or omit `NaN` values from the calculation for any of the previous syntaxes. `meanrot(quat,'includenan')` includes all `NaN` values in the calculation while `mean(quat,'omitnan')` ignores them.

Examples

Quaternion Mean Rotation

Create a matrix of quaternions corresponding to three sets of Euler angles.

```
eulerAngles = [40 20 10; ...
               50 10 5; ...
               45 70 1];

quat = quaternion(eulerAngles,'eulerd','ZYX','frame');
```

Determine the average rotation represented by the quaternions. Convert the average rotation to Euler angles in degrees for readability.

```
quatAverage = meanrot(quat)
```

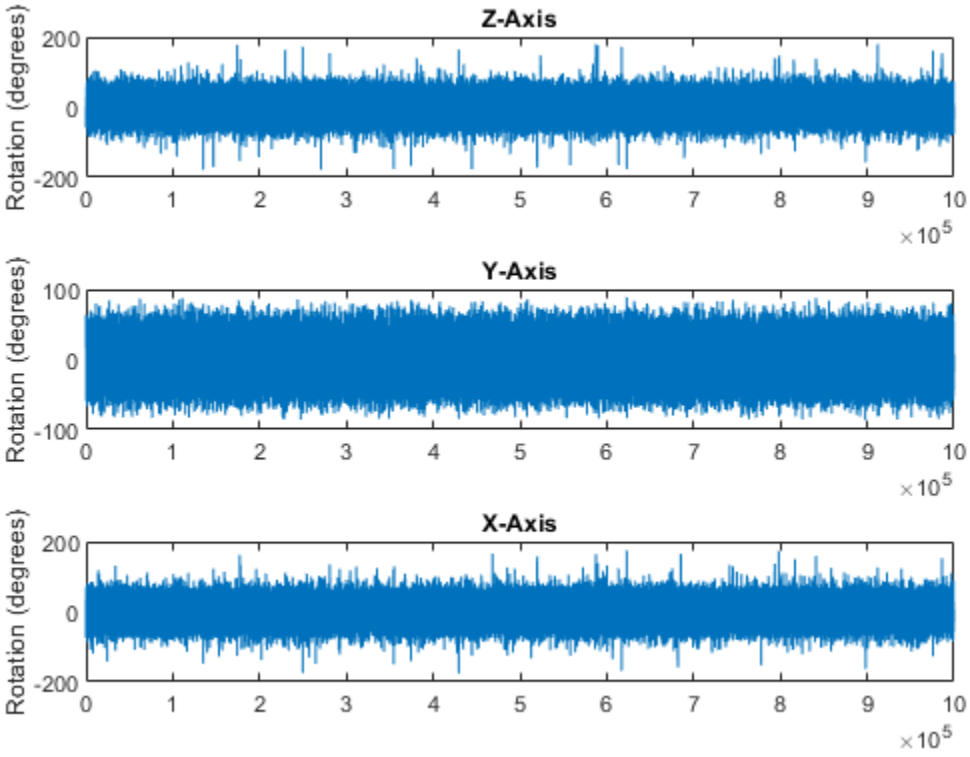
```
quatAverage = quaternion
      0.88863 - 0.062598i +  0.27822j +  0.35918k


eulerAverage = eulerd(quatAverage,'ZYX','frame')

eulerAverage = 1×3

   45.7876   32.6452    6.0407
```

Average Out Rotational Noise

Use `meanrot` over a sequence of quaternions to average out additive noise.

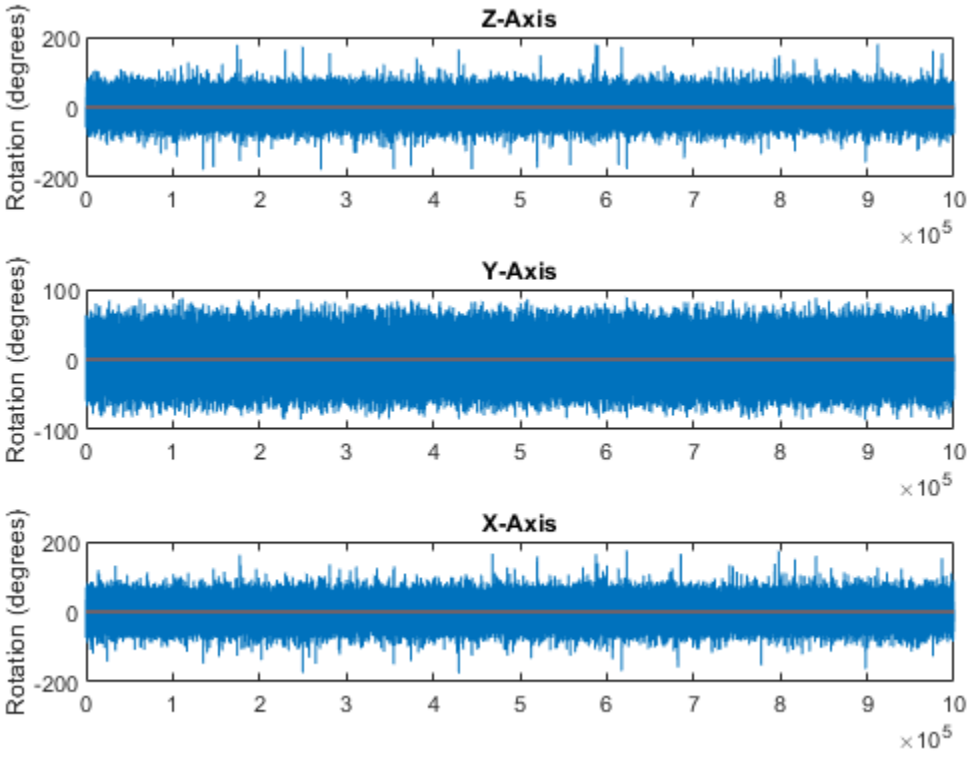
Create a vector of 1e6 quaternions whose distance, as defined by the `dist` function, from quaternion(1,0,0,0) is normally distributed. Plot the Euler angles corresponding to the noisy quaternion vector.

```
nrows = 1e6;
ax = 2*rand(nrows,3) - 1;
ax = ax./sqrt(sum(ax.^2,2));
ang = 0.5*randn(size(ax,1),1);
q = quaternion(ax.*ang ,'rotvec');

noisyEulerAngles = eulerd(q,'ZYX','frame');

figure(1)

subplot(3,1,1)
plot(noisyEulerAngles(:,1))
title('Z-Axis')
ylabel('Rotation (degrees)')
hold on

subplot(3,1,2)
plot(noisyEulerAngles(:,2))
title('Y-Axis')
ylabel('Rotation (degrees)')
hold on

subplot(3,1,3)
plot(noisyEulerAngles(:,3))
title('X-Axis')
ylabel('Rotation (degrees)')
hold on
```



Use `meanrot` to determine the average quaternion given the vector of quaternions. Convert to Euler angles and plot the results.

```
qAverage = meanrot(q);

qAverageInEulerAngles = eulerd(qAverage,'ZYX','frame');

figure(1)

subplot(3,1,1)
plot(ones(nrows,1)*qAverageInEulerAngles(:,1))
title('Z-Axis')

subplot(3,1,2)
plot(ones(nrows,1)*qAverageInEulerAngles(:,2))
title('Y-Axis')

subplot(3,1,3)
plot(ones(nrows,1)*qAverageInEulerAngles(:,3))
title('X-Axis')
```



The `meanrot` Algorithm and Limitations

The `meanrot` Algorithm

The `meanrot` function outputs a quaternion that minimizes the squared Frobenius norm of the difference between rotation matrices. Consider two quaternions:

- `q0` represents no rotation.
- `q90` represents a 90 degree rotation about the *x*-axis.

```
q0 = quaternion([0 0 0],'eulerd','ZYX','frame');
q90 = quaternion([0 0 90],'eulerd','ZYX','frame');
```

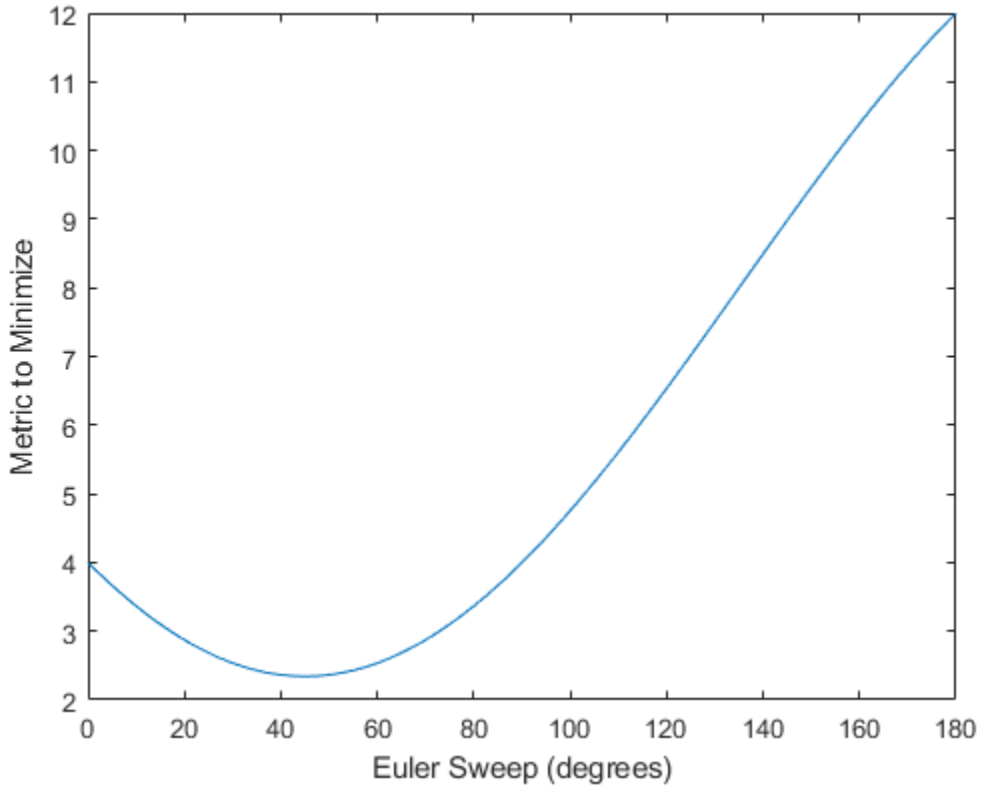
Create a quaternion sweep, `qSweep`, that represents rotations from 0 to 180 degrees about the *x*-axis.

```
eulerSweep = (0:1:180)';
qSweep = quaternion([zeros(numel(eulerSweep),2),eulerSweep], ...
    'eulerd','ZYX','frame');
```

Convert `q0`, `q90`, and `qSweep` to rotation matrices. In a loop, calculate the metric to minimize for each member of the quaternion sweep. Plot the results and return the value of the Euler sweep that corresponds to the minimum of the metric.

```
r0     = rotmat(q0,'frame');
r90    = rotmat(q90,'frame');
```

```
rSweep = rotmat(qSweep,'frame');

metricToMinimize = zeros(size(rSweep,3),1);
for i = 1:numel(qSweep)
    metricToMinimize(i) = norm((rSweep(:,:,i) - r0),'fro').^2 + ...
                          norm((rSweep(:,:,i) - r90),'fro').^2;
end

plot(eulerSweep,metricToMinimize)
xlabel('Euler Sweep (degrees)')
ylabel('Metric to Minimize')
```




```
[~,eulerIndex] = min(metricToMinimize);
eulerSweep(eulerIndex)
```

```
ans = 45
```

The minimum of the metric corresponds to the Euler angle sweep at 45 degrees. That is, `meanrot` defines the average between `quaterion([0 0 0],'ZYX','frame')` and `quaternion([0 0 90],'ZYX','frame')` as `quaternion([0 0 45],'ZYX','frame')`. Call `meanrot` with `q0` and `q90` to verify the same result.

```
eulerd(meanrot([q0,q90]),'ZYX','frame')
```

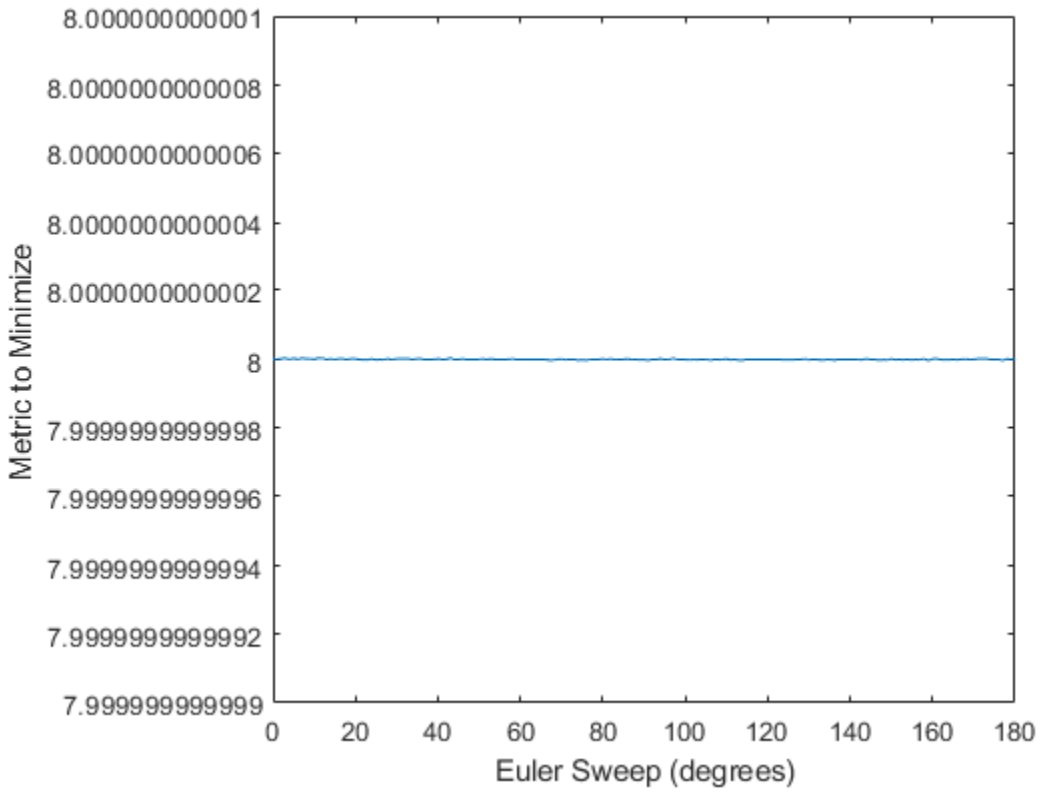
```
ans = 1×3
```

```
         0         0   45.0000
```

Limitations

The metric that `meanrot` uses to determine the mean rotation is not unique for quaternions significantly far apart. Repeat the experiment above for quaternions that are separated by 180 degrees.

```
q180 = quaternion([0 0 180],'eulerd','ZYX','frame');
r180 = rotmat(q180,'frame');

for i = 1:numel(qSweep)
    metricToMinimize(i) = norm((rSweep(:,:,i) - r0),'fro').^2 + ...
                          norm((rSweep(:,:,i) - r180),'fro').^2;
end

plot(eulerSweep,metricToMinimize)
xlabel('Euler Sweep (degrees)')
ylabel('Metric to Minimize')
```




```
[~,eulerIndex] = min(metricToMinimize);
eulerSweep(eulerIndex)
```

```
ans = 159
```

Quaternion means are usually calculated for rotations that are close to each other, which makes the edge case shown in this example unlikely in real-world applications. To average two quaternions that are significantly far apart, use the `slerp` function. Repeat the experiment using `slerp` and verify that the quaternion mean returned is more intuitive for large distances.

```
qMean = slerp(q0,q180,0.5);
q0_q180 = eulerd(qMean,'ZYX','frame')
```

```
q0_q180 = 1×3

         0         0   90.0000
```

Input Arguments

`quat` — Quaternion

scalar | vector | matrix | multidimensional array

Quaternion for which to calculate the mean, specified as a scalar, vector, matrix, or multidimensional array of quaternions.

Data Types: `quaternion`

`dim` — Dimension to operate along

positive integer scalar

Dimension to operate along, specified as a positive integer scalar. If no value is specified, then the default is the first array dimension whose size does not equal 1.

Dimension `dim` indicates the dimension whose length reduces to 1. The `size(quatAverage,dim)` is `1`, while the sizes of all other dimensions remain the same.

Data Types: `double` | `single`

`nanflag` — `NaN` condition

`'includenan'` (default) | `'omitnan'`

`NaN` condition, specified as one of these values:

- `'includenan'` -- Include `NaN` values when computing the mean rotation, resulting in `NaN`.
- `'omitnan'` -- Ignore all `NaN` values in the input.

Data Types: `char` | `string`

Output Arguments

`quatAverage` — Quaternion average rotation

scalar | vector | matrix | multidimensional array

Quaternion average rotation, returned as a scalar, vector, matrix, or multidimensional array.

Data Types: `single` | `double`

Algorithms

`meanrot` determines a quaternion mean, $\bar{q}$, according to [1]. $\bar{q}$ is the quaternion that minimizes the squared Frobenius norm of the difference between rotation matrices:

$$\bar{q} = \arg \min_{q \in S^3} \sum_{i=1}^{n} \|A(q) - A(q_i)\|_F^2$$

References

[1] Markley, F. Landis, Yang Chen, John Lucas Crassidis, and Yaakov Oshman. "Average Quaternions." *Journal of Guidance, Control, and Dynamics*. Vol. 30, Issue 4, 2007, pp. 1193-1197.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

Functions

`dist` | `slerp`

Objects

`quaternion`

Topics

“Rotations, Orientation, and Quaternions”

Introduced in R2019b

minus, -

Quaternion subtraction

Syntax

```
C = A - B
```

Description

`C = A - B` subtracts quaternion `B` from quaternion `A` using quaternion subtraction. Either `A` or `B` may be a real number, in which case subtraction is performed with the real part of the quaternion argument.

Examples

Subtract a Quaternion from a Quaternion

Quaternion subtraction is defined as the subtraction of the corresponding parts of each quaternion. Create two quaternions and perform subtraction.

```
Q1 = quaternion([1,0,-2,7]);
Q2 = quaternion([1,2,3,4]);

Q1minusQ2 = Q1 - Q2
```

```
Q1minusQ2 = quaternion
     0 - 2i - 5j + 3k
```

Subtract a Real Number from a Quaternion

Addition and subtraction of real numbers is defined for quaternions as acting on the real part of the quaternion. Create a quaternion and then subtract 1 from the real part.

```
Q = quaternion([1,1,1,1])
```

```
Q = quaternion
     1 + 1i + 1j + 1k
```

```
Qminus1 = Q - 1
```

```
Qminus1 = quaternion
     0 + 1i + 1j + 1k
```

Input Arguments

A — Input

scalar | vector | matrix | multidimensional array

Input, specified as a quaternion, array of quaternions, real number, or array of real numbers.

Data Types: `quaternion` | `single` | `double`

B — Input

scalar | vector | matrix | multidimensional array

Input, specified as a quaternion, array of quaternions, real number, or array of real numbers.

Data Types: `quaternion` | `single` | `double`

Output Arguments

C — Result

scalar | vector | matrix | multidimensional array

Result of quaternion subtraction, returned as a scalar, vector, matrix, or multidimensional array of quaternions.

Data Types: `quaternion`

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

Functions

`-,uminus` | `.*,times` | `*,mtimes`

Objects

`quaternion`

Topics

“Rotations, Orientation, and Quaternions”

Introduced in R2019b

mtimes, *

Quaternion multiplication

Syntax

```
quatC = A*B
```

Description

`quatC = A*B` implements quaternion multiplication if either `A` or `B` is a quaternion. Either `A` or `B` must be a scalar.

You can use quaternion multiplication to compose rotation operators:

- To compose a sequence of frame rotations, multiply the quaternions in the order of the desired sequence of rotations. For example, to apply a p quaternion followed by a q quaternion, multiply in the order pq. The rotation operator becomes $(pq)^{*}v(pq)$, where v represents the object to rotate specified in quaternion form. * represents conjugation.
- To compose a sequence of point rotations, multiply the quaternions in the reverse order of the desired sequence of rotations. For example, to apply a p quaternion followed by a q quaternion, multiply in the reverse order, qp. The rotation operator becomes $(qp)v(qp)^{*}$.

Examples

Multiply Quaternion Scalar and Quaternion Vector

Create a 4-by-1 column vector, `A`, and a scalar, `b`. Multiply `A` times `b`.

```
A = quaternion(randn(4,4))
```

```
A = 4x1 quaternion array
      0.53767 +  0.31877i +   3.5784j +   0.7254k
       1.8339 -   1.3077i +   2.7694j - 0.063055k
      -2.2588 -  0.43359i -   1.3499j +  0.71474k
      0.86217 +  0.34262i +   3.0349j -  0.20497k
```

```
b = quaternion(randn(1,4))
```

```
b = quaternion
    -0.12414 +  1.4897i +   1.409j +  1.4172k
```

```
C = A*b
```

```
C = 4x1 quaternion array
      -6.6117 +   4.8105i +  0.94224j -   4.2097k
      -2.0925 +   6.9079i +   3.9995j -   3.3614k
       1.8155 -   6.2313i -    1.336j -     1.89k
      -4.6033 +   5.8317i + 0.047161j -    2.791k
```

Input Arguments

A — Input

scalar | vector | matrix | multidimensional array

Input to multiply, specified as a quaternion, array of quaternions, real scalar, or array of real scalars.

If B is nonscalar, then A must be scalar.

Data Types: `quaternion` | `single` | `double`

B — Input

scalar | vector | matrix | multidimensional array

Input to multiply, specified as a quaternion, array of quaternions, real scalar, or array of real scalars.

If A is nonscalar, then B must be scalar.

Data Types: `quaternion` | `single` | `double`

Output Arguments

quatC — Quaternion product

scalar | vector | matrix | multidimensional array

Quaternion product, returned as a quaternion or array of quaternions.

Data Types: `quaternion`

Algorithms

Quaternion Multiplication by a Real Scalar

Given a quaternion

$$q = a_q + b_q\mathrm{i} + c_q\mathrm{j} + d_q\mathrm{k},$$

the product of q and a real scalar β is

$$\beta q = \beta a_q + \beta b_q\mathrm{i} + \beta c_q\mathrm{j} + \beta d_q\mathrm{k}$$

Quaternion Multiplication by a Quaternion Scalar

The definition of the basis elements for quaternions,

$$\mathrm{i}^2 = \mathrm{j}^2 = \mathrm{k}^2 = \mathrm{ijk} = -1\ ,$$

can be expanded to populate a table summarizing quaternion basis element multiplication:

	1	**i**	**j**	**k**
1	1	i	j	k
i	i	−1	k	−j

j	j	−k	−1	i
k	k	j	−i	−1

When reading the table, the rows are read first, for example: ij = k and ji = −k.

Given two quaternions, $q = a_q + b_q i + c_q j + d_q k$, and $p = a_p + b_p i + c_p j + d_p k$, the multiplication can be expanded as:

$$\begin{aligned} z = pq &= \left(a_p + b_p i + c_p j + d_p k\right)\left(a_q + b_q i + c_q j + d_q k\right) \\ &= a_p a_q + a_p b_q i + a_p c_q j + a_p d_q k \\ &+ b_p a_q i + b_p b_q i^2 + b_p c_q ij + b_p d_q ik \\ &+ c_p a_q j + c_p b_q ji + c_p c_q j^2 + c_p d_q jk \\ &+ d_p a_q k + d_p b_q ki + d_p c_q kj + d_p d_q k^2 \end{aligned}$$

You can simplify the equation using the quaternion multiplication table:

$$\begin{aligned} z = pq &= a_p a_q + a_p b_q i + a_p c_q j + a_p d_q k \\ &+ b_p a_q i - b_p b_q + b_p c_q k - b_p d_q j \\ &+ c_p a_q j - c_p b_q k - c_p c_q + c_p d_q i \\ &+ d_p a_q k + d_p b_q j - d_p c_q i - d_p d_q \end{aligned}$$

References

[1] Kuipers, Jack B. *Quaternions and Rotation Sequences: A Primer with Applications to Orbits, Aerospace, and Virtual Reality*. Princeton, NJ: Princeton University Press, 2007.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

Functions

`.*,times`

Objects

`quaternion`

Topics

“Rotations, Orientation, and Quaternions”

Introduced in R2019b

norm

Quaternion norm

Syntax

```
N = norm(quat)
```

Description

`N = norm(quat)` returns the norm of the quaternion, `quat`.

Given a quaternion of the form $Q = a + b\mathrm{i} + c\mathrm{j} + d\mathrm{k}$, the norm of the quaternion is defined as $\mathrm{norm}(Q) = \sqrt{a^2 + b^2 + c^2 + d^2}$.

Examples

Calculate Quaternion Norm

Create a scalar quaternion and calculate its norm.

```
quat = quaternion(1,2,3,4);
norm(quat)
```

```
ans = 5.4772
```

The quaternion norm is defined as the square root of the sum of the quaternion parts squared. Calculate the quaternion norm explicitly to verify the result of the `norm` function.

```
[a,b,c,d] = parts(quat);
sqrt(a^2+b^2+c^2+d^2)
```

```
ans = 5.4772
```

Input Arguments

quat — Quaternion

scalar | vector | matrix | multidimensional array

Quaternion for which to calculate the norm, specified as a scalar, vector, matrix, or multidimensional array of quaternions.

Data Types: `quaternion`

Output Arguments

N — Quaternion norm

scalar | vector | matrix | multidimensional array

Quaternion norm. If the input `quat` is an array, the output is returned as an array the same size as `quat`. Elements of the array are real numbers with the same data type as the underlying data type of the quaternion, `quat`.

Data Types: `single` | `double`

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

Functions

`normalize` | `parts` | `conj`

Objects

`quaternion`

Topics

“Rotations, Orientation, and Quaternions”

Introduced in R2019b

normalize

Quaternion normalization

Syntax

```
quatNormalized = normalize(quat)
```

Description

`quatNormalized = normalize(quat)` normalizes the quaternion.

Given a quaternion of the form $Q = a + b\mathrm{i} + c\mathrm{j} + d\mathrm{k}$, the normalized quaternion is defined as $Q/\sqrt{a^2 + b^2 + c^2 + d^2}$.

Examples

Normalize Elements of Quaternion Vector

Quaternions can represent rotations when normalized. You can use `normalize` to normalize a scalar, elements of a matrix, or elements of a multi-dimensional array of quaternions. Create a column vector of quaternions, then normalize them.

```
quatArray = quaternion([1,2,3,4; ...
                        2,3,4,1; ...
                        3,4,1,2]);
quatArrayNormalized = normalize(quatArray)
```

```
quatArrayNormalized = 3x1 quaternion array
     0.18257 + 0.36515i + 0.54772j +  0.7303k
     0.36515 + 0.54772i +  0.7303j + 0.18257k
     0.54772 +  0.7303i + 0.18257j + 0.36515k
```

Input Arguments

quat — Quaternion to normalize

scalar | vector | matrix | multidimensional array

Quaternion to normalize, specified as a scalar, vector, matrix, or multidimensional array of quaternions.

Data Types: `quaternion`

Output Arguments

quatNormalized — Normalized quaternion

scalar | vector | matrix | multidimensional array

Normalized quaternion, returned as a quaternion or array of quaternions the same size as `quat`.

Data Types: `quaternion`

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

Functions

`norm` | `.*,times` | `conj`

Objects

`quaternion`

Topics

“Rotations, Orientation, and Quaternions”

Introduced in R2019b

ones

Create quaternion array with real parts set to one and imaginary parts set to zero

Syntax

```
quatOnes = ones('quaternion')
quatOnes = ones(n,'quaternion')
quatOnes = ones(sz,'quaternion')
quatOnes = ones(sz1,...,szN,'quaternion')

quatOnes = ones( ___ ,'like',prototype,'quaternion')
```

Description

`quatOnes = ones('quaternion')` returns a scalar quaternion with the real part set to 1 and the imaginary parts set to 0.

Given a quaternion of the form $Q = a + b\mathrm{i} + c\mathrm{j} + d\mathrm{k}$, a quaternion one is defined as $Q = 1 + 0\mathrm{i} + 0\mathrm{j} + 0\mathrm{k}$.

`quatOnes = ones(n,'quaternion')` returns an `n`-by-`n` quaternion matrix with the real parts set to 1 and the imaginary parts set to 0.

`quatOnes = ones(sz,'quaternion')` returns an array of quaternion ones where the size vector, `sz`, defines `size(qOnes)`.

Example: `ones([1,4,2],'quaternion')` returns a 1-by-4-by-2 array of quaternions with the real parts set to `1` and the imaginary parts set to `0`.

`quatOnes = ones(sz1,...,szN,'quaternion')` returns a `sz1`-by-...-by-`szN` array of ones where `sz1,…,szN` indicates the size of each dimension.

`quatOnes = ones( ___ ,'like',prototype,'quaternion')` specifies the underlying class of the returned quaternion array to be the same as the underlying class of the quaternion prototype.

Examples

Quaternion Scalar One

Create a quaternion scalar one.

```
quatOnes = ones('quaternion')
```

```
quatOnes = quaternion
     1 + 0i + 0j + 0k
```

Square Matrix of Quaternion Ones

Create an n-by-n matrix of quaternion ones.

```
n = 3;
quatOnes = ones(n,'quaternion')
```

```
quatOnes = 3x3 quaternion array
     1 + 0i + 0j + 0k     1 + 0i + 0j + 0k     1 + 0i + 0j + 0k
     1 + 0i + 0j + 0k     1 + 0i + 0j + 0k     1 + 0i + 0j + 0k
     1 + 0i + 0j + 0k     1 + 0i + 0j + 0k     1 + 0i + 0j + 0k
```

Multidimensional Array of Quaternion Ones

Create a multidimensional array of quaternion ones by defining array dimensions in order. In this example, you create a 3-by-1-by-2 array. You can specify dimensions using a row vector or comma-separated integers. Specify the dimensions using a row vector and display the results:

```
dims = [3,1,2];
quatOnesSyntax1 = ones(dims,'quaternion')
```

```
quatOnesSyntax1 = 3x1x2 quaternion array
quatOnesSyntax1(:,:,1) =

     1 + 0i + 0j + 0k
     1 + 0i + 0j + 0k
     1 + 0i + 0j + 0k


quatOnesSyntax1(:,:,2) =

     1 + 0i + 0j + 0k
     1 + 0i + 0j + 0k
     1 + 0i + 0j + 0k
```

Specify the dimensions using comma-separated integers, and then verify the equivalency of the two syntaxes:

```
quatOnesSyntax2 = ones(3,1,2,'quaternion');
isequal(quatOnesSyntax1,quatOnesSyntax2)
```

```
ans = logical
   1
```

Underlying Class of Quaternion Ones

A quaternion is a four-part hyper-complex number used in three-dimensional rotations and orientations. You can specify the underlying data type of the parts as `single` or `double`. The default is `double`.

Create a quaternion array of ones with the underlying data type set to `single`.

```
quatOnes = ones(2,'like',single(1),'quaternion')
```

```
quatOnes = 2x2 quaternion array
     1 + 0i + 0j + 0k     1 + 0i + 0j + 0k
     1 + 0i + 0j + 0k     1 + 0i + 0j + 0k
```

Verify the underlying class using the `classUnderlying` function.

```
classUnderlying(quatOnes)
```

```
ans =
'single'
```

Input Arguments

`n` — Size of square quaternion matrix

integer value

Size of square quaternion matrix, specified as an integer value.

If `n` is zero or negative, then `quatOnes` is returned as an empty matrix.

Example: `ones(4,'quaternion')` returns a 4-by-4 matrix of quaternions with the real parts set to `1` and the imaginary parts set to `0`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`sz` — Output size

row vector of integer values

Output size, specified as a row vector of integer values. Each element of `sz` indicates the size of the corresponding dimension in `quatOnes`. If the size of any dimension is `0` or negative, then `quatOnes` is returned as an empty array.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`prototype` — Quaternion prototype

variable

Quaternion prototype, specified as a variable.

Example: `ones(2,'like',quat,'quaternion')` returns a 2-by-2 matrix of quaternions with the same underlying class as the prototype quaternion, `quat`.

Data Types: `quaternion`

`sz1,...,szN` — Size of each dimension

two or more integer values

Size of each dimension, specified as two or more integers. If the size of any dimension is `0` or negative, then `quatOnes` is returned as an empty array.

Example: `ones(2,3,'quaternion')` returns a 2-by-3 matrix of quaternions with the real parts set to `1` and the imaginary parts set to `0`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Output Arguments

`quat0nes` — Quaternion ones

scalar | vector | matrix | multidimensional array

Quaternion ones, returned as a scalar, vector, matrix, or multidimensional array of quaternions.

Given a quaternion of the form $Q = a + b\mathrm{i} + c\mathrm{j} + d\mathrm{k}$, a quaternion one is defined as $Q = 1 + 0\mathrm{i} + 0\mathrm{j} + 0\mathrm{k}$.

Data Types: `quaternion`

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

Functions

`zeros`

Objects

`quaternion`

Topics

“Rotations, Orientation, and Quaternions”

Introduced in R2019b

parts

Extract quaternion parts

Syntax

```
[a,b,c,d] = parts(quat)
```

Description

`[a,b,c,d] = parts(quat)` returns the parts of the quaternion array as arrays, each the same size as `quat`.

Examples

Convert Quaternion to Matrix of Quaternion Parts

Convert a quaternion representation to parts using the `parts` function.

Create a two-element column vector of quaternions by specifying the parts.

```
quat = quaternion([1:4;5:8])
```

```
quat = 2x1 quaternion array
     1 + 2i + 3j + 4k
     5 + 6i + 7j + 8k
```

Recover the parts from the quaternion matrix using the `parts` function. The parts are returned as separate output arguments, each the same size as the input 2-by-1 column vector of quaternions.

```
[qA,qB,qC,qD] = parts(quat)
```

```
qA = 2×1

     1
     5


qB = 2×1

     2
     6


qC = 2×1

     3
     7


qD = 2×1
```

```
4
8
```

Input Arguments

`quat` — Quaternion

scalar | vector | matrix | multidimensional array

Quaternion, specified as a quaternion or array of quaternions.

Data Types: `quaternion`

Output Arguments

`[a,b,c,d]` — Quaternion parts

scalar | vector | matrix | multidimensional array

Quaternion parts, returned as four arrays: a, b, c, and d. Each part is the same size as `quat`.

Data Types: `single` | `double`

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

Functions

`classUnderlying` | `compact`

Objects

`quaternion`

Topics

“Rotations, Orientation, and Quaternions”

Introduced in R2019b

power, .^

Element-wise quaternion power

Syntax

```
C = A.^b
```

Description

`C = A.^b` raises each element of `A` to the corresponding power in `b`.

Examples

Raise a Quaternion to a Real Scalar Power

Create a quaternion and raise it to a real scalar power.

```
A = quaternion(1,2,3,4)
```

```
A = quaternion
     1 + 2i + 3j + 4k
```

```
b = 3;
C = A.^b
```

```
C = quaternion
     -86 -  52i -  78j - 104k
```

Raise a Quaternion Array to Powers from a Multidimensional Array

Create a 2-by-1 quaternion array and raise it to powers from a 2-D array.

```
A = quaternion([1:4;5:8])
```

```
A = 2x1 quaternion array
     1 + 2i + 3j + 4k
     5 + 6i + 7j + 8k
```

```
b = [1 0 2; 3 2 1]
```

```
b = 2×3

     1     0     2
     3     2     1
```

```
C = A.^b
```

```
C = 2x3 quaternion array
        1 +    2i +    3j +    4k        1 +    0i +    0j +    0k      -28 +    4i +    6j +
    -2110 -  444i -  518j -  592k     -124 +   60i +   70j +   80k        5 +    6i +    7j +
```

Input Arguments

A — Base

scalar | vector | matrix | multidimensional array

Base, specified as a scalar, vector, matrix, or multidimensional array.

Data Types: `quaternion` | `single` | `double`

b — Exponent

scalar | vector | matrix | multidimensional array

Exponent, specified as a real scalar, vector, matrix, or multidimensional array.

Data Types: `single` | `double`

Output Arguments

C — Result

scalar | vector | matrix | multidimensional array

Each element of quaternion A raised to the corresponding power in b, returned as a scalar, vector, matrix, or multidimensional array.

Data Types: `quaternion`

Algorithms

The polar representation of a quaternion $A = a + b\mathrm{i} + c\mathrm{j} + d\mathrm{k}$ is given by

$$A = \|A\|\left(\cos\theta + \widehat{u}\sin\theta\right)$$

where θ is the angle of rotation, and $\hat{u}$ is the unit quaternion.

Quaternion A raised by a real exponent b is given by

$$P = A.\hat{}\,b = \|A\|^{b}\left(\cos(b\theta) + \widehat{u}\sin(b\theta)\right)$$

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

Functions

`log` | `exp`

Objects
`quaternion`

Topics
“Rotations, Orientation, and Quaternions”

Introduced in R2019b

prod

Product of a quaternion array

Syntax

```
quatProd = prod(quat)
quatProd = prod(quat,dim)
```

Description

`quatProd = prod(quat)` returns the quaternion product of the elements of the array.

`quatProd = prod(quat,dim)` calculates the quaternion product along dimension `dim`.

Examples

Product of Quaternions in Each Column

Create a 3-by-3 array whose elements correspond to their linear indices.

```
A = reshape(quaternion(randn(9,4)),3,3)
```

```
A = 3x3 quaternion array
      0.53767 +   2.7694i +    1.409j -  0.30344k      0.86217 +   0.7254i -   1.2075j +   0.888
       1.8339 -   1.3499i +   1.4172j +  0.29387k      0.31877 - 0.063055i +  0.71724j -   1.147
      -2.2588 +   3.0349i +   0.6715j -  0.78728k      -1.3077 +  0.71474i +   1.6302j -   1.068
```

Find the product of the quaternions in each column. The length of the first dimension is `1`, and the length of the second dimension matches `size(A,2)`.

```
B = prod(A)
```

```
B = 1x3 quaternion array
     -19.837 -  9.1521i +  15.813j -  19.918k     -5.4708 - 0.28535i +   3.077j -  1.2295k
```

Product of Specified Dimension of Quaternion Array

You can specify which dimension of a quaternion array to take the product of.

Create a 2-by-2-by-2 quaternion array.

```
A = reshape(quaternion(randn(8,4)),2,2,2);
```

Find the product of the elements in each page of the array. The length of the first dimension matches `size(A,1)`, the length of the second dimension matches `size(A,2)`, and the length of the third dimension is `1`.

```
dim = 3;
B = prod(A,dim)

B = 2x2 quaternion array
     -2.4847 +  1.1659i - 0.37547j +  2.8068k     0.28786 - 0.29876i - 0.51231j -  4.2972k
     0.38986 -  3.6606i -  2.0474j -   6.047k      -1.741 - 0.26782i +  5.4346j +  4.1452k
```

Input Arguments

`quat` — Quaternion

scalar | vector | matrix | multidimensional array

Quaternion, specified as scalar, vector, matrix, or multidimensional array of quaternions.

Example: `qProd = prod(quat)` calculates the quaternion product along the first non-singleton dimension of `quat`.

Data Types: `quaternion`

`dim` — Dimension

first non-singleton dimension (default) | positive integer

Dimension along which to calculate the quaternion product, specified as a positive integer. If `dim` is not specified, `prod` operates along the first non-singleton dimension of `quat`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Output Arguments

`quatProd` — Quaternion product

positive integer

Quaternion product, returned as quaternion array with one less non-singleton dimension than `quat`.

For example, if `quat` is a 2-by-2-by-5 array,

- `prod(quat,1)` returns a 1-by-2-by-5 array.
- `prod(quat,2)` returns a 2-by-1-by-5 array.
- `prod(quat,3)` returns a 2-by-2 array.

Data Types: `quaternion`

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

Functions

`mtimes` | `.*,times`

Objects
`quaternion`

Topics
“Rotations, Orientation, and Quaternions”

Introduced in R2019b

rdivide, ./

Element-wise quaternion right division

Syntax

```
C = A./B
```

Description

`C = A./B` performs quaternion element-wise division by dividing each element of quaternion `A` by the corresponding element of quaternion `B`.

Examples

Divide a Quaternion Array by a Real Scalar

Create a 2-by-1 quaternion array, and divide it element-by-element by a real scalar.

```
A = quaternion([1:4;5:8])
```

```
A = 2x1 quaternion array
     1 + 2i + 3j + 4k
     5 + 6i + 7j + 8k
```

```
B = 2;
C = A./B
```

```
C = 2x1 quaternion array
     0.5 +   1i + 1.5j +   2k
     2.5 +   3i + 3.5j +   4k
```

Divide a Quaternion Array by Another Quaternion Array

Create a 2-by-2 quaternion array, and divide it element-by-element by another 2-by-2 quaternion array.

```
q1 = quaternion(magic(4));
A = reshape(q1,2,2)
```

```
A = 2x2 quaternion array
     16 +  2i +  3j + 13k      9 +  7i +  6j + 12k
      5 + 11i + 10j +  8k      4 + 14i + 15j +  1k
```

```
q2 = quaternion([1:4;3:6;2:5;4:7]);
B = reshape(q2,2,2)
```

```
B = 2x2 quaternion array
     1 + 2i + 3j + 4k     2 + 3i + 4j + 5k
     3 + 4i + 5j + 6k     4 + 5i + 6j + 7k


C = A./B

C = 2x2 quaternion array
          2.7 -      0.1i -      2.1j -      1.7k       2.2778 + 0.092593i -  0.46296j -  0.5740
       1.8256 - 0.081395i +  0.45349j -  0.24419k       1.4524 -      0.5i +   1.0238j -   0.261
```

Input Arguments

A — Dividend

scalar | vector | matrix | multidimensional array

Dividend, specified as a quaternion, an array of quaternions, a real scalar, or an array of real numbers.

A and B must have compatible sizes. In the simplest cases, they can be the same size or one can be a scalar. Two inputs have compatible sizes if, for every dimension, the dimension sizes of the inputs are the same or one of the dimensions is 1.

Data Types: `quaternion` | `single` | `double`

B — Divisor

scalar | vector | matrix | multidimensional array

Divisor, specified as a quaternion, an array of quaternions, a real scalar, or an array of real numbers.

A and B must have compatible sizes. In the simplest cases, they can be the same size or one can be a scalar. Two inputs have compatible sizes if, for every dimension, the dimension sizes of the inputs are the same or one of the dimensions is 1.

Data Types: `quaternion` | `single` | `double`

Output Arguments

C — Result

scalar | vector | matrix | multidimensional array

Result of quaternion division, returned as a scalar, vector, matrix, or multidimensional array.

Data Types: `quaternion`

Algorithms

Quaternion Division

Given a quaternion $A = a_1 + a_2\mathrm{i} + a_3\mathrm{j} + a_4\mathrm{k}$ and a real scalar p,

$$C = A\,./p = \frac{a_1}{p} + \frac{a_2}{p}\mathrm{i} + \frac{a_3}{p}\mathrm{j} + \frac{a_4}{p}\mathrm{k}$$

Note For a real scalar p, $A./p = A.\backslash p$.

Quaternion Division by a Quaternion Scalar

Given two quaternions A and B of compatible sizes,

$$C = A./B = A.*B^{-1} = A.*\left(\frac{conj(B)}{norm(B)^2}\right)$$

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

Functions

`conj` | `./,ldivide` | `norm` | `.*,times`

Objects

`quaternion`

Topics

“Rotations, Orientation, and Quaternions”

Introduced in R2019b

randrot

Uniformly distributed random rotations

Syntax

```
R = randrot
R = randrot(m)
R = randrot(m1,...,mN)
R = randrot([m1,...,mN])
```

Description

`R = randrot` returns a unit quaternion drawn from a uniform distribution of random rotations.

`R = randrot(m)` returns an `m`-by-`m` matrix of unit quaternions drawn from a uniform distribution of random rotations.

`R = randrot(m1,...,mN)` returns an `m1`-by-...-by-`mN` array of random unit quaternions, where `m1`, ..., `mN` indicate the size of each dimension. For example, `randrot(3,4)` returns a 3-by-4 matrix of random unit quaternions.

`R = randrot([m1,...,mN])` returns an `m1`-by-...-by-`mN` array of random unit quaternions, where `m1`,..., `mN` indicate the size of each dimension. For example, `randrot([3,4])` returns a 3-by-4 matrix of random unit quaternions.

Examples

Matrix of Random Rotations

Generate a 3-by-3 matrix of uniformly distributed random rotations.

```
r = randrot(3)
```

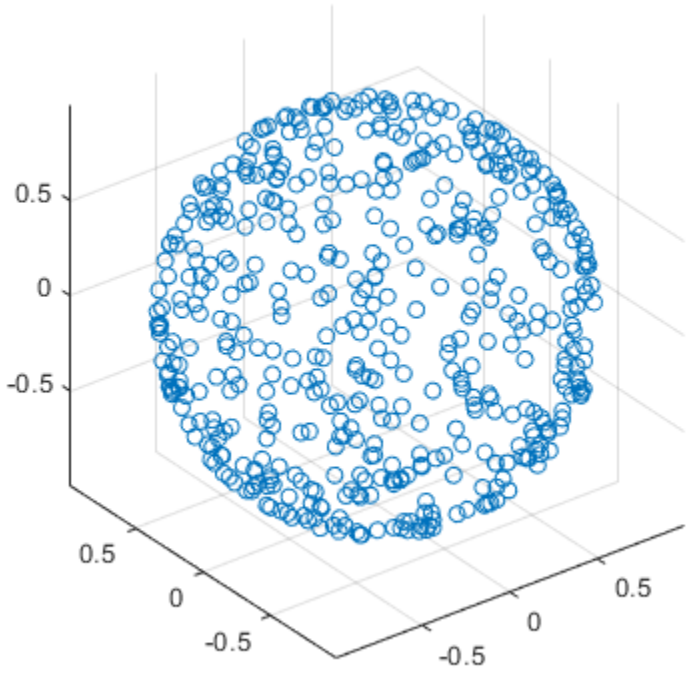
```
r = 3x3 quaternion array
      0.17446 +  0.59506i -  0.73295j +  0.27976k      0.69704 - 0.060589i +  0.68679j -  0.1969
      0.21908 -  0.89875i -    0.298j +  0.23548k    -0.049744 +  0.59691i +  0.56459j +  0.5678
       0.6375 +  0.49338i -  0.24049j +  0.54068k       0.2979 -  0.53568i +  0.31819j +  0.7232
```

Create Uniform Distribution of Random Rotations

Create a vector of 500 random quaternions. Use rotatepoint on page 2-1124 to visualize the distribution of the random rotations applied to point (1, 0, 0).

```
q = randrot(500,1);

pt = rotatepoint(q, [1 0 0]);
```

```
figure
scatter3(pt(:,1), pt(:,2), pt(:,3))
axis equal
```



Input Arguments

`m` — Size of square matrix

integer

Size of square quaternion matrix, specified as an integer value. If `m` is `0` or negative, then `R` is returned as an empty matrix.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`m1,...,mN` — Size of each dimension

two or more integer values

Size of each dimension, specified as two or more integer values. If the size of any dimension is `0` or negative, then `R` is returned as an empty array.

Example: `randrot(2,3)` returns a 2-by-3 matrix of random quaternions.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`[m1,...,mN]` — Vector of size of each dimension

row vector of integer values

Vector of size of each dimension, specified as a row vector of two or more integer values. If the size of any dimension is `0` or negative, then `R` is returned as an empty array.

Example: `randrot([2,3])` returns a 2-by-3 matrix of random quaternions.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Output Arguments

R — Random quaternions

scalar | vector | matrix | multidimensional array

Random quaternions, returned as a quaternion or array of quaternions.

Data Types: `quaternion`

References

[1] Shoemake, K. "Uniform Random Rotations." *Graphics Gems III* (K. David, ed.). New York: Academic Press, 1992.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`quaternion`

Topics

“Rotations, Orientation, and Quaternions”

Introduced in R2019b

rotateframe

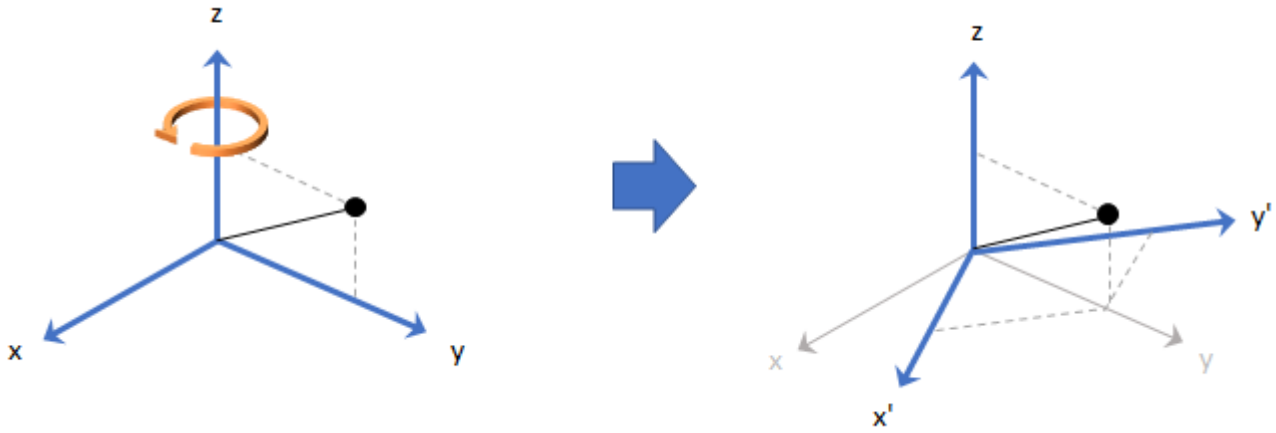
Quaternion frame rotation

Syntax

```
rotationResult = rotateframe(quat,cartesianPoints)
```

Description

`rotationResult = rotateframe(quat,cartesianPoints)` rotates the frame of reference for the Cartesian points using the quaternion, `quat`. The elements of the quaternion are normalized before use in the rotation.

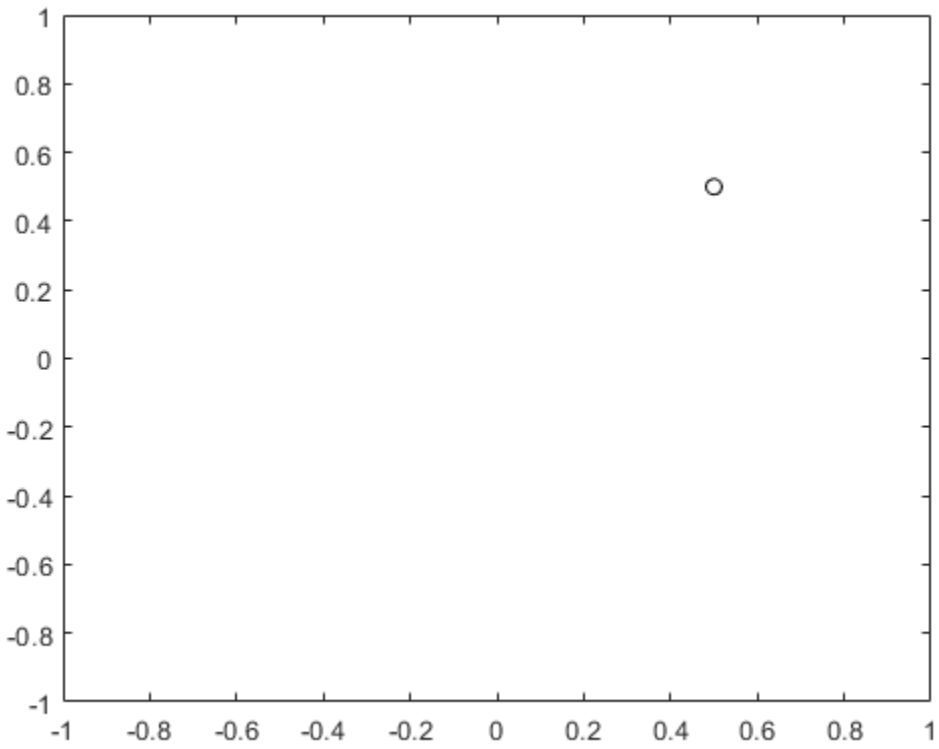


Examples

Rotate Frame Using Quaternion Vector

Define a point in three dimensions. The coordinates of a point are always specified in the order *x*, *y*, and *z*. For convenient visualization, define the point on the *x*-*y* plane.

```
x = 0.5;
y = 0.5;
z = 0;
plot(x,y,'ko')
hold on
axis([-1 1 -1 1])
```



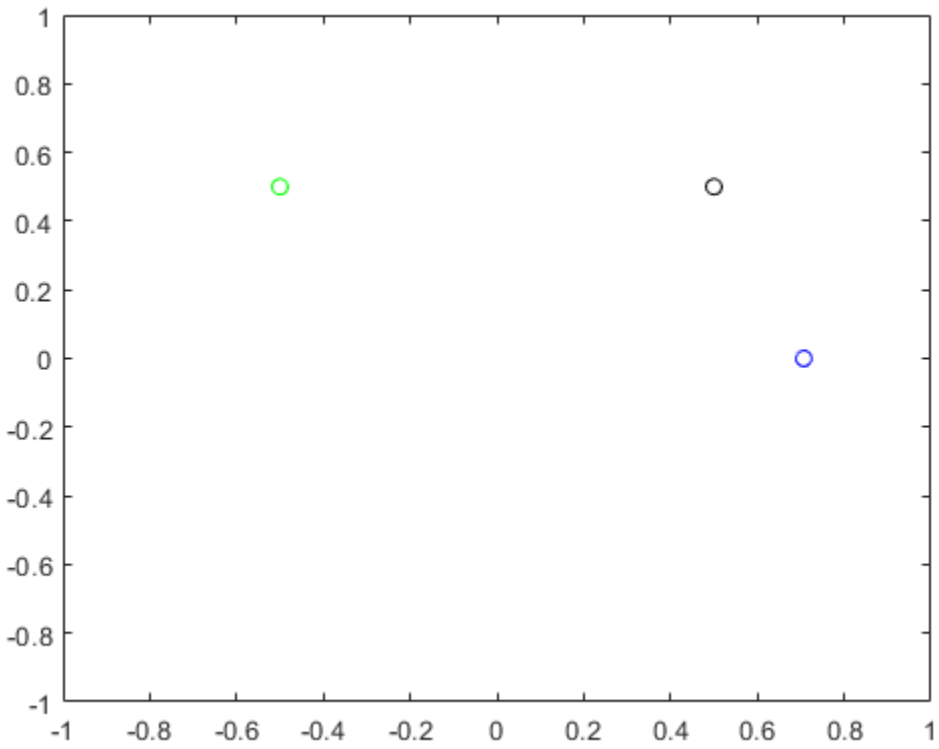
Create a quaternion vector specifying two separate rotations, one to rotate the frame 45 degrees and another to rotate the point -90 degrees about the *z*-axis. Use `rotateframe` to perform the rotations.

```
quat = quaternion([0,0,pi/4; ...
                   0,0,-pi/2],'euler','XYZ','frame');

rereferencedPoint = rotateframe(quat,[x,y,z])
```

```
rereferencedPoint = 2×3

    0.7071   -0.0000         0
   -0.5000    0.5000         0
```

Plot the rereferenced points.

```
plot(rereferencedPoint(1,1),rereferencedPoint(1,2),'bo')
plot(rereferencedPoint(2,1),rereferencedPoint(2,2),'go')
```



Rereference Group of Points using Quaternion

Define two points in three-dimensional space. Define a quaternion to rereference the points by first rotating the reference frame about the *z*-axis 30 degrees and then about the new *y*-axis 45 degrees.

```
a = [1,0,0];
b = [0,1,0];
quat = quaternion([30,45,0],'eulerd','ZYX','point');
```

Use `rotateframe` to reference both points using the quaternion rotation operator. Display the result.

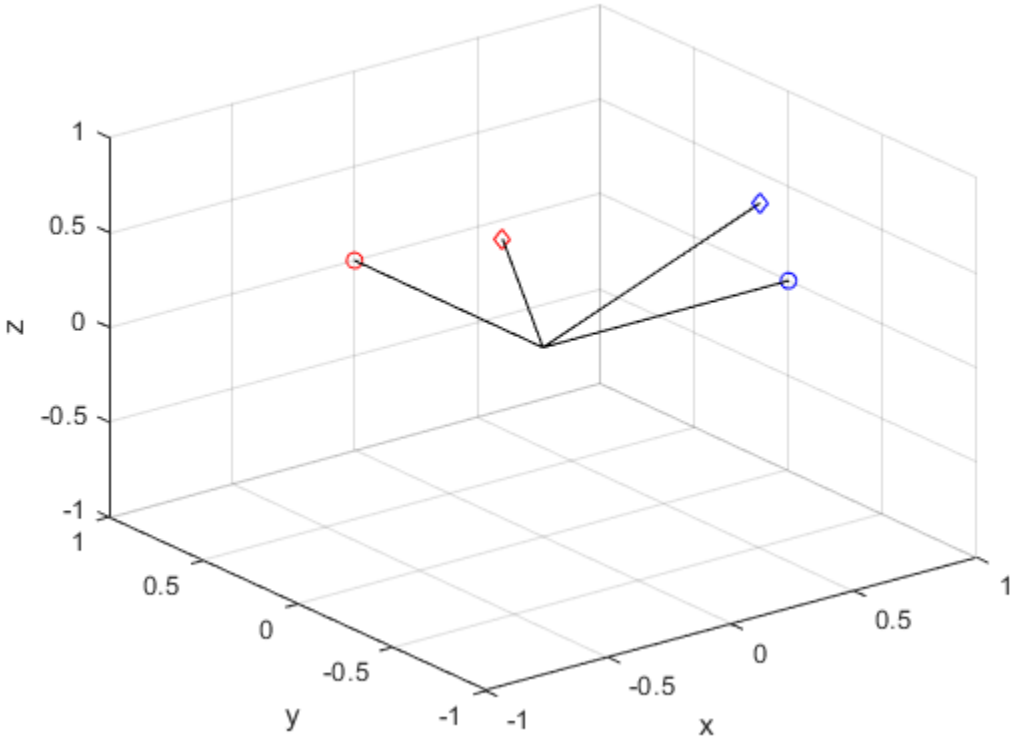
```
rP = rotateframe(quat,[a;b])
```

```
rP = 2×3

    0.6124   -0.3536    0.7071
    0.5000    0.8660   -0.0000
```

Visualize the original orientation and the rotated orientation of the points. Draw lines from the origin to each of the points for visualization purposes.

```
plot3(a(1),a(2),a(3),'bo');

hold on
```

```
grid on
axis([-1 1 -1 1 -1 1])
xlabel('x')
ylabel('y')
zlabel('z')

plot3(b(1),b(2),b(3),'ro');
plot3(rP(1,1),rP(1,2),rP(1,3),'bd')
plot3(rP(2,1),rP(2,2),rP(2,3),'rd')

plot3([0;rP(1,1)],[0;rP(1,2)],[0;rP(1,3)],'k')
plot3([0;rP(2,1)],[0;rP(2,2)],[0;rP(2,3)],'k')
plot3([0;a(1)],[0;a(2)],[0;a(3)],'k')
plot3([0;b(1)],[0;b(2)],[0;b(3)],'k')
```



Input Arguments

quat — Quaternion that defines rotation

scalar | vector

Quaternion that defines rotation, specified as a scalar quaternion or vector of quaternions.

Data Types: `quaternion`

cartesianPoints — Three-dimensional Cartesian points

1-by-3 vector | *N*-by-3 matrix

Three-dimensional Cartesian points, specified as a 1-by-3 vector or *N*-by-3 matrix.

Data Types: `single` | `double`

Output Arguments

`rotationResult` — Re-referenced Cartesian points

vector | matrix

Cartesian points defined in reference to rotated reference frame, returned as a vector or matrix the same size as `cartesianPoints`.

The data type of the re-referenced Cartesian points is the same as the underlying data type of `quat`.

Data Types: `single` | `double`

Algorithms

Quaternion frame rotation re-references a point specified in $\mathbf{R}^3$ by rotating the original frame of reference according to a specified quaternion:

$$L_q(u) = q^*uq$$

where q is the quaternion, * represents conjugation, and u is the point to rotate, specified as a quaternion.

For convenience, the `rotateframe` function takes a point in $\mathbf{R}^3$ and returns a point in $\mathbf{R}^3$. Given a function call with some arbitrary quaternion, $q = a + b\mathrm{i} + c\mathrm{j} + d\mathrm{k}$, and arbitrary coordinate, $[x,y,z]$,

```
point = [x,y,z];
rereferencedPoint = rotateframe(q,point)
```

the `rotateframe` function performs the following operations:

1. Converts point $[x,y,z]$ to a quaternion:

   $$u_q = 0 + xi + yj + zk$$

2. Normalizes the quaternion, q:

   $$q_n = \frac{q}{\sqrt{a^2 + b^2 + c^2 + d^2}}$$

3. Applies the rotation:

   $$v_q = q^*u_qq$$

4. Converts the quaternion output, v_q, back to $\mathbf{R}^3$

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

Functions
`rotatepoint`

Objects
`quaternion`

Topics
“Rotations, Orientation, and Quaternions”

Introduced in R2019b

rotatepoint

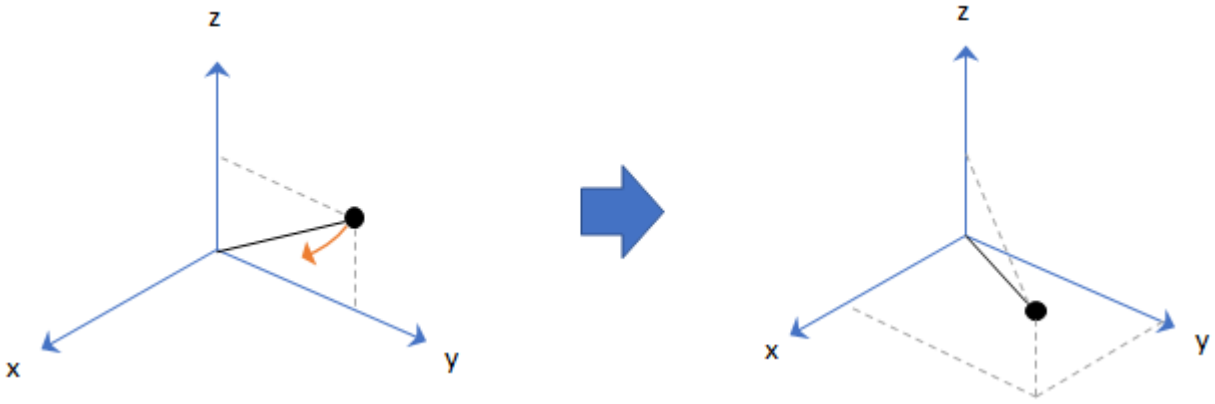
Quaternion point rotation

Syntax

```
rotationResult = rotatepoint(quat,cartesianPoints)
```

Description

`rotationResult = rotatepoint(quat,cartesianPoints)` rotates the Cartesian points using the quaternion, `quat`. The elements of the quaternion are normalized before use in the rotation.

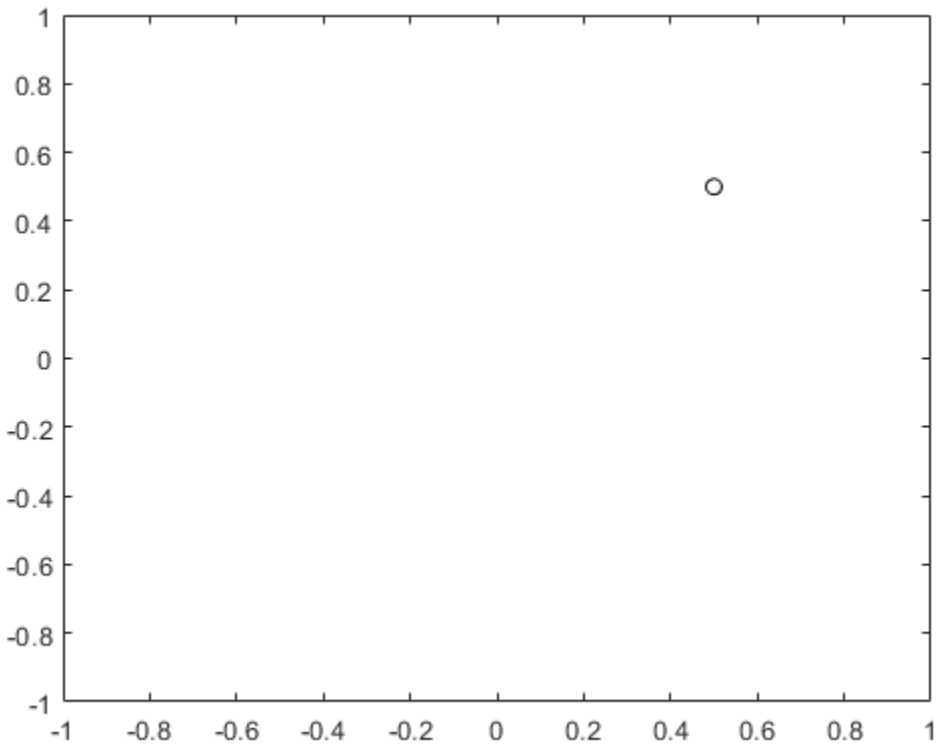


Examples

Rotate Point Using Quaternion Vector

Define a point in three dimensions. The coordinates of a point are always specified in order *x*, *y*, *z*. For convenient visualization, define the point on the *x*-*y* plane.

```
x = 0.5;
y = 0.5;
z = 0;

plot(x,y,'ko')
hold on
axis([-1 1 -1 1])
```



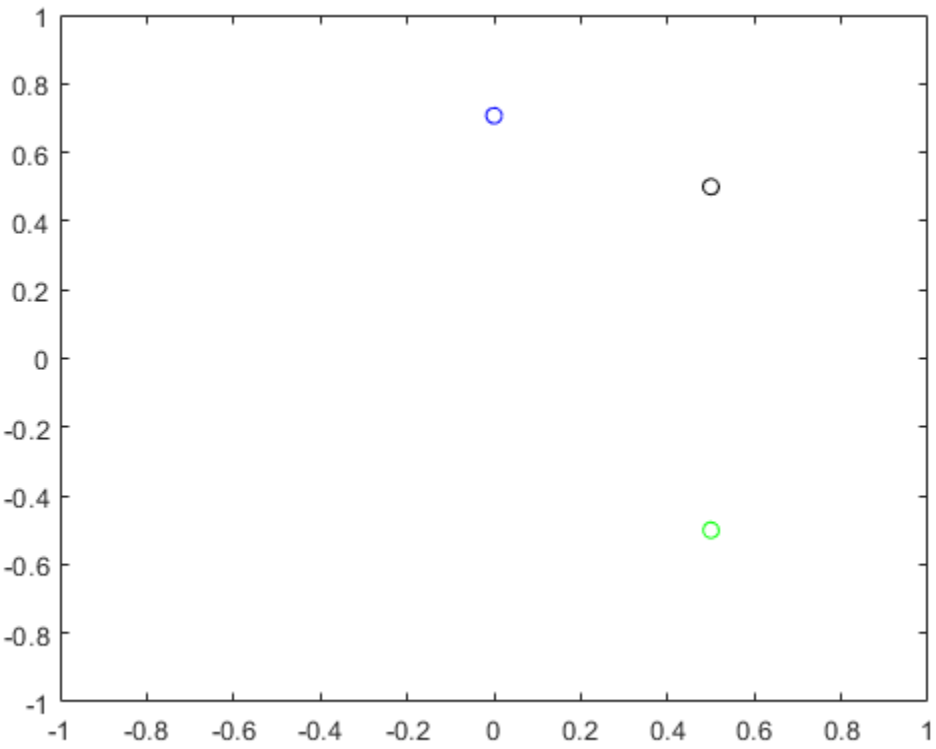
Create a quaternion vector specifying two separate rotations, one to rotate the point 45 and another to rotate the point -90 degrees about the *z*-axis. Use `rotatepoint` to perform the rotation.

```
quat = quaternion([0,0,pi/4; ...
                   0,0,-pi/2],'euler','XYZ','point');

rotatedPoint = rotatepoint(quat,[x,y,z])
```

```
rotatedPoint = 2×3

   -0.0000    0.7071         0
    0.5000   -0.5000         0
```

Plot the rotated points.

```
plot(rotatedPoint(1,1),rotatedPoint(1,2),'bo')
plot(rotatedPoint(2,1),rotatedPoint(2,2),'go')
```



Rotate Group of Points Using Quaternion

Define two points in three-dimensional space. Define a quaternion to rotate the point by first rotating about the *z*-axis 30 degrees and then about the new *y*-axis 45 degrees.

```
a = [1,0,0];
b = [0,1,0];
quat = quaternion([30,45,0],'eulerd','ZYX','point');
```

Use `rotatepoint` to rotate both points using the quaternion rotation operator. Display the result.

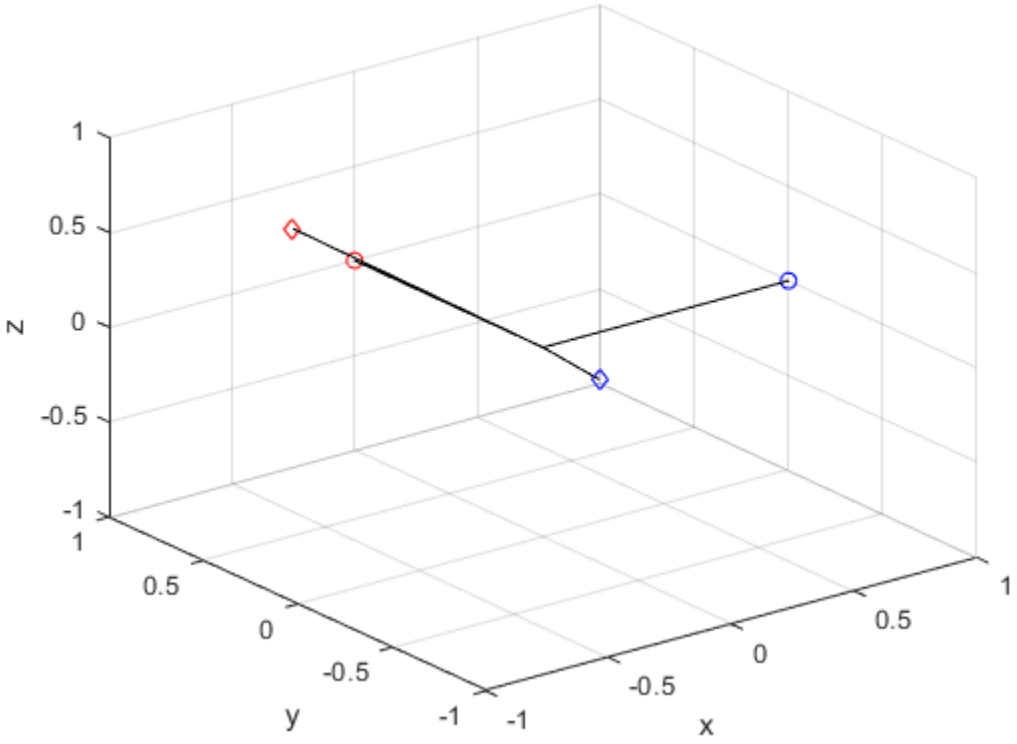
```
rP = rotatepoint(quat,[a;b])
```

```
rP = 2×3

    0.6124    0.5000   -0.6124
   -0.3536    0.8660    0.3536
```

Visualize the original orientation and the rotated orientation of the points. Draw lines from the origin to each of the points for visualization purposes.

```
plot3(a(1),a(2),a(3),'bo');

hold on
```

```
grid on
axis([-1 1 -1 1 -1 1])
xlabel('x')
ylabel('y')
zlabel('z')

plot3(b(1),b(2),b(3),'ro');
plot3(rP(1,1),rP(1,2),rP(1,3),'bd')
plot3(rP(2,1),rP(2,2),rP(2,3),'rd')

plot3([0;rP(1,1)],[0;rP(1,2)],[0;rP(1,3)],'k')
plot3([0;rP(2,1)],[0;rP(2,2)],[0;rP(2,3)],'k')
plot3([0;a(1)],[0;a(2)],[0;a(3)],'k')
plot3([0;b(1)],[0;b(2)],[0;b(3)],'k')
```



Input Arguments

quat — Quaternion that defines rotation

scalar | vector

Quaternion that defines rotation, specified as a scalar quaternion, row vector of quaternions, or column vector of quaternions.

Data Types: `quaternion`

`cartesianPoints` — Three-dimensional Cartesian points
1-by-3 vector | *N*-by-3 matrix

Three-dimensional Cartesian points, specified as a 1-by-3 vector or *N*-by-3 matrix.

Data Types: `single` | `double`

Output Arguments

`rotationResult` — Repositioned Cartesian points
vector | matrix

Rotated Cartesian points defined using the quaternion rotation, returned as a vector or matrix the same size as `cartesianPoints`.

Data Types: `single` | `double`

Algorithms

Quaternion point rotation rotates a point specified in $\mathbf{R}^3$ according to a specified quaternion:

$$L_q(u) = quq^*$$

where q is the quaternion, * represents conjugation, and u is the point to rotate, specified as a quaternion.

For convenience, the `rotatepoint` function takes in a point in $\mathbf{R}^3$ and returns a point in $\mathbf{R}^3$. Given a function call with some arbitrary quaternion, $q = a + b\mathrm{i} + c\mathrm{j} + d\mathrm{k}$, and arbitrary coordinate, $[x,y,z]$, for example,

```
rereferencedPoint = rotatepoint(q,[x,y,z])
```

the `rotatepoint` function performs the following operations:

1. Converts point $[x,y,z]$ to a quaternion:

   $$u_q = 0 + xi + yj + zk$$

2. Normalizes the quaternion, q:

   $$q_n = \frac{q}{\sqrt{a^2 + b^2 + c^2 + d^2}}$$

3. Applies the rotation:

   $$v_q = qu_qq^*$$

4. Converts the quaternion output, v_q, back to $\mathbf{R}^3$

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

Functions
`rotateframe`

Objects
`quaternion`

Topics
“Rotations, Orientation, and Quaternions”

Introduced in R2019b

rotmat

Convert quaternion to rotation matrix

Syntax

```
rotationMatrix = rotmat(quat,rotationType)
```

Description

`rotationMatrix = rotmat(quat,rotationType)` converts the quaternion, `quat`, to an equivalent rotation matrix representation.

Examples

Convert Quaternion to Rotation Matrix for Point Rotation

Define a quaternion for use in point rotation.

```
theta = 45;
gamma = 30;
quat = quaternion([0,theta,gamma],'eulerd','ZYX','point')
```

```
quat = quaternion
       0.8924 +  0.23912i +  0.36964j + 0.099046k
```

Convert the quaternion to a rotation matrix.

```
rotationMatrix = rotmat(quat,'point')
```

```
rotationMatrix = 3×3

    0.7071   -0.0000    0.7071
    0.3536    0.8660   -0.3536
   -0.6124    0.5000    0.6124
```

To verify the rotation matrix, directly create two rotation matrices corresponding to the rotations about the *y*- and *x*-axes. Multiply the rotation matrices and compare to the output of `rotmat`.

```
theta = 45;
gamma = 30;

ry = [cosd(theta)   0           sind(theta) ; ...
      0             1           0           ; ...
     -sind(theta)   0           cosd(theta)];

rx = [1             0           0           ;      ...
      0             cosd(gamma) -sind(gamma) ;     ...
      0             sind(gamma) cosd(gamma)];

rotationMatrixVerification = rx*ry
```

```
rotationMatrixVerification = 3×3

    0.7071         0    0.7071
    0.3536    0.8660   -0.3536
   -0.6124    0.5000    0.6124
```

Convert Quaternion to Rotation Matrix for Frame Rotation

Define a quaternion for use in frame rotation.

```
theta = 45;
gamma = 30;
quat = quaternion([0,theta,gamma],'eulerd','ZYX','frame')
```

```
quat = quaternion
       0.8924 +  0.23912i +  0.36964j - 0.099046k
```

Convert the quaternion to a rotation matrix.

```
rotationMatrix = rotmat(quat,'frame')
```

```
rotationMatrix = 3×3

    0.7071   -0.0000   -0.7071
    0.3536    0.8660    0.3536
    0.6124   -0.5000    0.6124
```

To verify the rotation matrix, directly create two rotation matrices corresponding to the rotations about the *y*- and *x*-axes. Multiply the rotation matrices and compare to the output of `rotmat`.

```
theta = 45;
gamma = 30;

ry = [cosd(theta)   0           -sind(theta) ; ...
      0             1           0            ; ...
     sind(theta)   0           cosd(theta)];

rx = [1             0           0            ;      ...
      0             cosd(gamma) sind(gamma) ;     ...
      0             -sind(gamma) cosd(gamma)];

rotationMatrixVerification = rx*ry
```

```
rotationMatrixVerification = 3×3

    0.7071         0   -0.7071
    0.3536    0.8660    0.3536
    0.6124   -0.5000    0.6124
```

Convert Quaternion Vector to Rotation Matrices

Create a 3-by-1 normalized quaternion vector.

```
qVec = normalize(quaternion(randn(3,4)));
```

Convert the quaternion array to rotation matrices. The pages of `rotmatArray` correspond to the linear index of `qVec`.

```
rotmatArray = rotmat(qVec,'frame');
```

Assume `qVec` and `rotmatArray` correspond to a sequence of rotations. Combine the quaternion rotations into a single representation, then apply the quaternion rotation to arbitrarily initialized Cartesian points.

```
loc = normalize(randn(1,3));
quat = prod(qVec);
rotateframe(quat,loc)
```

```
ans = 1×3

    0.9524    0.5297    0.9013
```

Combine the rotation matrices into a single representation, then apply the rotation matrix to the same initial Cartesian points. Verify the quaternion rotation and rotation matrix result in the same orientation.

```
totalRotMat = eye(3);
for i = 1:size(rotmatArray,3)
    totalRotMat = rotmatArray(:,:,i)*totalRotMat;
end
totalRotMat*loc'
```

```
ans = 3×1

    0.9524
    0.5297
    0.9013
```

Input Arguments

`quat` — Quaternion to convert

scalar | vector | matrix | multidimensional array

Quaternion to convert, specified as a scalar, vector, matrix, or multidimensional array.

Data Types: `quaternion`

`rotationType` — Type or rotation

`'frame'` | `'point'`

Type of rotation represented by the `rotationMatrix` output, specified as `'frame'` or `'point'`.

Data Types: `char` | `string`

Output Arguments

`rotationMatrix` — Rotation matrix representation
3-by-3 matrix | 3-by-3-by-*N* multidimensional array

Rotation matrix representation, returned as a 3-by-3 matrix or 3-by-3-by-*N* multidimensional array.

- If `quat` is a scalar, `rotationMatrix` is returned as a 3-by-3 matrix.
- If `quat` is non-scalar, `rotationMatrix` is returned as a 3-by-3-by-*N* multidimensional array, where `rotationMatrix(:,:,i)` is the rotation matrix corresponding to `quat(i)`.

The data type of the rotation matrix is the same as the underlying data type of `quat`.

Data Types: `single` | `double`

Algorithms

Given a quaternion of the form

$$q = a + bi + cj + dk,$$

the equivalent rotation matrix for frame rotation is defined as

$$\begin{bmatrix} 2a^2 - 1 + 2b^2 & 2bc + 2ad & 2bd - 2ac \\ 2bc - 2ad & 2a^2 - 1 + 2c^2 & 2cd + 2ab \\ 2bd + 2ac & 2cd - 2ab & 2a^2 - 1 + 2d^2 \end{bmatrix}.$$

The equivalent rotation matrix for point rotation is the transpose of the frame rotation matrix:

$$\begin{bmatrix} 2a^2 - 1 + 2b^2 & 2bc - 2ad & 2bd + 2ac \\ 2bc + 2ad & 2a^2 - 1 + 2c^2 & 2cd - 2ab \\ 2bd - 2ac & 2cd + 2ab & 2a^2 - 1 + 2d^2 \end{bmatrix}.$$

References

[1] Kuipers, Jack B. *Quaternions and Rotation Sequences: A Primer with Applications to Orbits, Aerospace, and Virtual Reality*. Princeton, NJ: Princeton University Press, 2007.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

Functions
`rotvec` | `rotvecd` | `euler` | `eulerd`

Objects
`quaternion`

Topics
“Rotations, Orientation, and Quaternions”

Introduced in R2019b

rotvec

Convert quaternion to rotation vector (radians)

Syntax

```
rotationVector = rotvec(quat)
```

Description

`rotationVector = rotvec(quat)` converts the quaternion array, `quat`, to an *N*-by-3 matrix of equivalent rotation vectors in radians. The elements of `quat` are normalized before conversion.

Examples

Convert Quaternion to Rotation Vector in Radians

Convert a random quaternion scalar to a rotation vector in radians

```
quat = quaternion(randn(1,4));
rotvec(quat)
```

```
ans = 1×3

    1.6866   -2.0774    0.7929
```

Input Arguments

`quat` — Quaternion to convert

scalar | vector | matrix | multidimensional array

Quaternion to convert, specified as scalar quaternion, vector, matrix, or multidimensional array of quaternions.

Data Types: `quaternion`

Output Arguments

`rotationVector` — Rotation vector (radians)

N-by-3 matrix

Rotation vector representation, returned as an *N*-by-3 matrix of rotations vectors, where each row represents the [X Y Z] angles of the rotation vectors in radians. The *i*th row of `rotationVector` corresponds to the element `quat(i)`.

The data type of the rotation vector is the same as the underlying data type of `quat`.

Data Types: `single` | `double`

Algorithms

All rotations in 3-D can be represented by a three-element axis of rotation and a rotation angle, for a total of four elements. If the rotation axis is constrained to be unit length, the rotation angle can be distributed over the vector elements to reduce the representation to three elements.

Recall that a quaternion can be represented in axis-angle form

$$q = \cos\left(\theta/2\right) + \sin\left(\theta/2\right)(\mathrm{xi} + y\mathrm{j} + z\mathrm{k}),$$

where θ is the angle of rotation and $[x,y,z]$ represent the axis of rotation.

Given a quaternion of the form

$$q = a + bi + cj + dk\,,$$

you can solve for the rotation angle using the axis-angle form of quaternions:

$$\theta = 2\cos^{-1}(a)\,.$$

Assuming a normalized axis, you can rewrite the quaternion as a rotation vector without loss of information by distributing θ over the parts b, c, and d. The rotation vector representation of q is

$$q_{\mathrm{rv}} = \frac{\theta}{\sin\left(\theta/2\right)}[b, c, d]\,.$$

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

Functions

`rotvecd` | `euler` | `eulerd`

Objects

`quaternion`

Topics

“Rotations, Orientation, and Quaternions”

Introduced in R2019b

rotvecd

Convert quaternion to rotation vector (degrees)

Syntax

```
rotationVector = rotvecd(quat)
```

Description

`rotationVector = rotvecd(quat)` converts the quaternion array, `quat`, to an *N*-by-3 matrix of equivalent rotation vectors in degrees. The elements of `quat` are normalized before conversion.

Examples

Convert Quaternion to Rotation Vector in Degrees

Convert a random quaternion scalar to a rotation vector in degrees.

```
quat = quaternion(randn(1,4));
rotvecd(quat)
```

```
ans = 1×3

   96.6345 -119.0274   45.4312
```

Input Arguments

`quat` — Quaternion to convert

scalar | vector | matrix | multidimensional array

Quaternion to convert, specified as scalar, vector, matrix, or multidimensional array of quaternions.

Data Types: `quaternion`

Output Arguments

`rotationVector` — Rotation vector (degrees)

N-by-3 matrix

Rotation vector representation, returned as an *N*-by-3 matrix of rotation vectors, where each row represents the [*x y z*] angles of the rotation vectors in degrees. The *i*th row of `rotationVector` corresponds to the element `quat(i)`.

The data type of the rotation vector is the same as the underlying data type of `quat`.

Data Types: `single` | `double`

Algorithms

All rotations in 3-D can be represented by four elements: a three-element axis of rotation and a rotation angle. If the rotation axis is constrained to be unit length, the rotation angle can be distributed over the vector elements to reduce the representation to three elements.

Recall that a quaternion can be represented in axis-angle form

$$q = \cos\left(\theta/2\right) + \sin\left(\theta/2\right)(\mathrm{xi} + y\mathrm{j} + z\mathrm{k}),$$

where θ is the angle of rotation in degrees, and $[x,y,z]$ represent the axis of rotation.

Given a quaternion of the form

$$q = a + bi + cj + dk\,,$$

you can solve for the rotation angle using the axis-angle form of quaternions:

$$\theta = 2\cos^{-1}(a)\,.$$

Assuming a normalized axis, you can rewrite the quaternion as a rotation vector without loss of information by distributing θ over the parts b, c, and d. The rotation vector representation of q is

$$q_{\mathrm{rv}} = \frac{\theta}{\sin\left(\theta/2\right)}[b, c, d]\,.$$

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

Functions

`rotvec` | `euler` | `eulerd`

Objects

`quaternion`

Topics

“Rotations, Orientation, and Quaternions”

Introduced in R2019b

slerp

Spherical linear interpolation

Syntax

```
q0 = slerp(q1,q2,T)
```

Description

`q0 = slerp(q1,q2,T)` spherically interpolates between `q1` and `q2` by the interpolation coefficient `T`. The function always chooses the shorter interpolation path between `q1` and `q2`.

Examples

Interpolate Between Two Quaternions

Create two quaternions with the following interpretation:

1. `a` = 45 degree rotation around the *z*-axis
2. `c` = -45 degree rotation around the *z*-axis

```
a = quaternion([45,0,0],'eulerd','ZYX','frame');
c = quaternion([-45,0,0],'eulerd','ZYX','frame');
```

Call `slerp` with the quaternions `a` and `c` and specify an interpolation coefficient of 0.5.

```
interpolationCoefficient = 0.5;

b = slerp(a,c,interpolationCoefficient);
```

The output of `slerp`, `b`, represents an average rotation of `a` and `c`. To verify, convert `b` to Euler angles in degrees.

```
averageRotation = eulerd(b,'ZYX','frame')
```

```
averageRotation = 1×3

     0     0     0
```

The interpolation coefficient is specified as a normalized value between `0` and `1`, inclusive. An interpolation coefficient of `0` corresponds to the `a` quaternion, and an interpolation coefficient of `1` corresponds to the `c` quaternion. Call `slerp` with coefficients `0` and `1` to confirm.

```
b = slerp(a,c,[0,1]);
eulerd(b,'ZYX','frame')
```

```
ans = 2×3

   45.0000         0         0
```

```
  -45.0000         0         0
```

You can create smooth paths between quaternions by specifying arrays of equally spaced interpolation coefficients.

```
path = 0:0.1:1;

interpolatedQuaternions = slerp(a,c,path);
```

For quaternions that represent rotation only about a single axis, specifying interpolation coefficients as equally spaced results in quaternions equally spaced in Euler angles. Convert `interpolatedQuaternions` to Euler angles and verify that the difference between the angles in the path is constant.

```
k = eulerd(interpolatedQuaternions,'ZYX','frame');
abc = abs(diff(k))
```

```
abc = 10×3

    9.0000         0         0
    9.0000         0         0
    9.0000         0         0
    9.0000         0         0
    9.0000         0         0
    9.0000         0         0
    9.0000         0         0
    9.0000         0         0
    9.0000         0         0
    9.0000         0         0
```

Alternatively, you can use the `dist` function to verify that the distance between the interpolated quaternions is consistent. The `dist` function returns angular distance in radians; convert to degrees for easy comparison.

```
def = rad2deg(dist(interpolatedQuaternions(2:end),interpolatedQuaternions(1:end-1)))
```

```
def = 1×10

    9.0000    9.0000    9.0000    9.0000    9.0000    9.0000    9.0000    9.0000    9.0000    9.0
```

SLERP Minimizes Great Circle Path

The SLERP algorithm interpolates along a great circle path connecting two quaternions. This example shows how the SLERP algorithm minimizes the great circle path.

Define three quaternions:

1. `q0` - quaternion indicating no rotation from the global frame
2. `q179` - quaternion indicating a 179 degree rotation about the *z*-axis
3. `q180` - quaternion indicating a 180 degree rotation about the *z*-axis

4 `q181` - quaternion indicating a 181 degree rotation about the *z*-axis

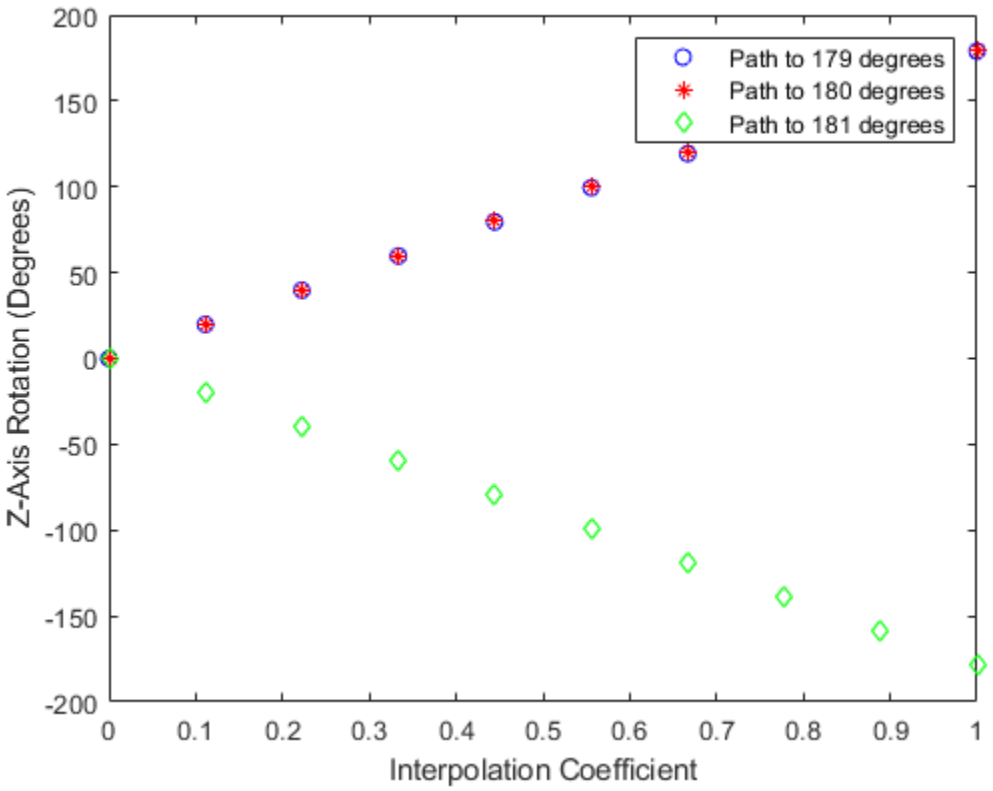
```
q0 = ones(1,'quaternion');

q179 = quaternion([179,0,0],'eulerd','ZYX','frame');

q180 = quaternion([180,0,0],'eulerd','ZYX','frame');

q181 = quaternion([181,0,0],'eulerd','ZYX','frame');
```

Use `slerp` to interpolate between `q0` and the three quaternion rotations. Specify that the paths are traveled in 10 steps.

```
T = linspace(0,1,10);

q179path = slerp(q0,q179,T);
q180path = slerp(q0,q180,T);
q181path = slerp(q0,q181,T);
```

Plot each path in terms of Euler angles in degrees.

```
q179pathEuler = eulerd(q179path,'ZYX','frame');
q180pathEuler = eulerd(q180path,'ZYX','frame');
q181pathEuler = eulerd(q181path,'ZYX','frame');

plot(T,q179pathEuler(:,1),'bo', ...
     T,q180pathEuler(:,1),'r*', ...
     T,q181pathEuler(:,1),'gd');
legend('Path to 179 degrees', ...
       'Path to 180 degrees', ...
       'Path to 181 degrees')
xlabel('Interpolation Coefficient')
ylabel('Z-Axis Rotation (Degrees)')
```



The path between `q0` and `q179` is clockwise to minimize the great circle distance. The path between `q0` and `q181` is counterclockwise to minimize the great circle distance. The path between `q0` and `q180` can be either clockwise or counterclockwise, depending on numerical rounding.

Show Interpolated Quaternions on Sphere

Create two quaternions.

```
q1 = quaternion([75,-20,-10],'eulerd','ZYX','frame');
q2 = quaternion([-45,20,30],'eulerd','ZYX','frame');
```

Define the interpolation coefficient.

```
T = 0:0.01:1;
```

Obtain the interpolated quaternions.

```
quats = slerp(q1,q2,T);
```

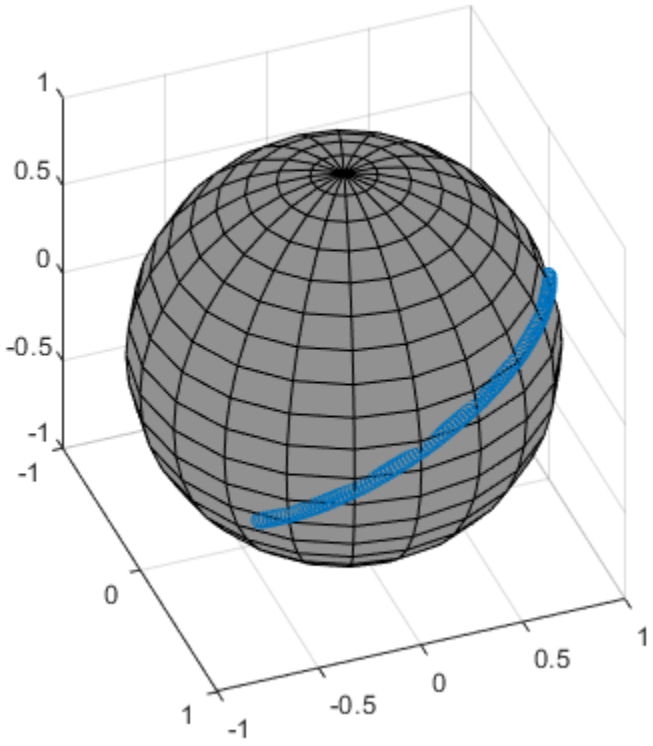
Obtain the corresponding rotate points.

```
pts = rotatepoint(quats,[1 0 0]);
```

Show the interpolated quaternions on a unit sphere.

```
figure
[X,Y,Z] = sphere;
```

```
surf(X,Y,Z,'FaceColor',[0.57 0.57 0.57])
hold on;

scatter3(pts(:,1),pts(:,2),pts(:,3))
view([69.23 36.60])
axis equal
```



Note that the interpolated quaternions follow the shorter path from `q1` to `q2`.

Input Arguments

q1 — Quaternion

scalar | vector | matrix | multidimensional array

Quaternion to interpolate, specified as a scalar, vector, matrix, or multidimensional array of quaternions.

`q1`, `q2`, and `T` must have compatible sizes. In the simplest cases, they can be the same size or any one can be a scalar. Two inputs have compatible sizes if, for every dimension, the dimension sizes of the inputs are either the same or one of them is 1.

Data Types: `quaternion`

q2 — Quaternion

scalar | vector | matrix | multidimensional array

Quaternion to interpolate, specified as a scalar, vector, matrix, or multidimensional array of quaternions.

`q1`, `q2`, and `T` must have compatible sizes. In the simplest cases, they can be the same size or any one can be a scalar. Two inputs have compatible sizes if, for every dimension, the dimension sizes of the inputs are either the same or one of the dimension sizes is 1.

Data Types: `quaternion`

`T` — Interpolation coefficient
scalar | vector | matrix | multidimensional array

Interpolation coefficient, specified as a scalar, vector, matrix, or multidimensional array of numbers with each element in the range [0,1].

`q1`, `q2`, and `T` must have compatible sizes. In the simplest cases, they can be the same size or any one can be a scalar. Two inputs have compatible sizes if, for every dimension, the dimension sizes of the inputs are either the same or one of the dimension sizes is 1.

Data Types: `single` | `double`

Output Arguments

`q0` — Interpolated quaternion
scalar | vector | matrix | multidimensional array

Interpolated quaternion, returned as a scalar, vector, matrix, or multidimensional array.

Data Types: `quaternion`

Algorithms

Quaternion **s**pherical **l**inear int**erp**olation (SLERP) is an extension of linear interpolation along a plane to spherical interpolation in three dimensions. The algorithm was first proposed in [1]. Given two quaternions, q_1 and q_2, SLERP interpolates a new quaternion, q_0, along the great circle that connects q_1 and q_2. The interpolation coefficient, T, determines how close the output quaternion is to either q_1 and q_2.

The SLERP algorithm can be described in terms of sinusoids:

$$q_0 = \frac{\sin((1-T)\theta)}{\sin(\theta)}q_1 + \frac{\sin(T\theta)}{\sin(\theta)}q_2$$

where q_1 and q_2 are normalized quaternions, and θ is half the angular distance between q_1 and q_2.

References

[1] Shoemake, Ken. "Animating Rotation with Quaternion Curves." *ACM SIGGRAPH Computer Graphics* Vol. 19, Issue 3, 1985, pp. 345–354.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

Functions
`dist` | `meanrot`

Objects
`quaternion`

Topics
“Lowpass Filter Orientation Using Quaternion SLERP”
“Rotations, Orientation, and Quaternions”

Introduced in R2019b

times, .*

Element-wise quaternion multiplication

Syntax

```
quatC = A.*B
```

Description

`quatC = A.*B` returns the element-by-element quaternion multiplication of quaternion arrays.

You can use quaternion multiplication to compose rotation operators:

- To compose a sequence of frame rotations, multiply the quaternions in the same order as the desired sequence of rotations. For example, to apply a p quaternion followed by a q quaternion, multiply in the order pq. The rotation operator becomes $(pq)^*v(pq)$, where v represents the object to rotate in quaternion form. * represents conjugation.
- To compose a sequence of point rotations, multiply the quaternions in the reverse order of the desired sequence of rotations. For example, to apply a p quaternion followed by a q quaternion, multiply in the reverse order, qp. The rotation operator becomes $(qp)v(qp)^*$.

Examples

Multiply Two Quaternion Vectors

Create two vectors, `A` and `B`, and multiply them element by element.

```
A = quaternion([1:4;5:8]);
B = A;
C = A.*B
```

```
C = 2x1 quaternion array
     -28 +   4i +   6j +   8k
    -124 +  60i +  70j +  80k
```

Multiply Two Quaternion Arrays

Create two 3-by-3 arrays, `A` and `B`, and multiply them element by element.

```
A = reshape(quaternion(randn(9,4)),3,3);
B = reshape(quaternion(randn(9,4)),3,3);
C = A.*B
```

```
C = 3x3 quaternion array
     0.60169 +  2.4332i -  2.5844j + 0.51646k    -0.49513 +  1.1722i +  4.4401j -   1.217k      2
     -4.2329 +  2.4547i +  3.7768j + 0.77484k    -0.65232 - 0.43112i -  1.4645j - 0.90073k     -1
```

```
     -4.4159 +  2.1926i +  1.9037j -  4.0303k     -2.0232 +  0.4205i - 0.17288j +  3.8529k     -2
```

Note that quaternion multiplication is not commutative:

```
isequal(C,B.*A)
```

```
ans = logical
   0
```

Multiply Quaternion Row and Column Vectors

Create a row vector `a` and a column vector `b`, then multiply them. The 1-by-3 row vector and 4-by-1 column vector combine to produce a 4-by-3 matrix with all combinations of elements multiplied.

```
a = [zeros('quaternion'),ones('quaternion'),quaternion(randn(1,4))]
```

```
a = 1x3 quaternion array
          0 +       0i +       0j +       0k           1 +       0i +       0j +       0k     0
```

```
b = quaternion(randn(4,4))
```

```
b = 4x1 quaternion array
      0.31877 +   3.5784i +   0.7254j -  0.12414k
      -1.3077 +   2.7694i - 0.063055j +   1.4897k
     -0.43359 -   1.3499i +  0.71474j +    1.409k
      0.34262 +   3.0349i -  0.20497j +   1.4172k
```

```
a.*b
```

```
ans = 4x3 quaternion array
            0 +        0i +        0j +        0k      0.31877 +   3.5784i +   0.7254j -  0.1241
            0 +        0i +        0j +        0k      -1.3077 +   2.7694i - 0.063055j +   1.489
            0 +        0i +        0j +        0k     -0.43359 -   1.3499i +  0.71474j +    1.40
            0 +        0i +        0j +        0k      0.34262 +   3.0349i -  0.20497j +   1.417
```

Input Arguments

A — Array to multiply

scalar | vector | matrix | multidimensional array

Array to multiply, specified as a quaternion, an array of quaternions, a real scalar, or an array of real numbers.

`A` and `B` must have compatible sizes. In the simplest cases, they can be the same size or one can be a scalar. Two inputs have compatible sizes if, for every dimension, the dimension sizes of the inputs are the same or one of them is 1.

Data Types: `quaternion` | `single` | `double`

B — Array to multiply

scalar | vector | matrix | multidimensional array

Array to multiply, specified as a quaternion, an array of quaternions, a real scalar, or an array of real numbers.

A and B must have compatible sizes. In the simplest cases, they can be the same size or one can be a scalar. Two inputs have compatible sizes if, for every dimension, the dimension sizes of the inputs are the same or one of them is 1.

Data Types: `quaternion` | `single` | `double`

Output Arguments

quatC — Quaternion product

scalar | vector | matrix | multidimensional array

Quaternion product, returned as a scalar, vector, matrix, or multidimensional array.

Data Types: `quaternion`

Algorithms

Quaternion Multiplication by a Real Scalar

Given a quaternion,

$$q = a_q + b_q i + c_q j + d_q k,$$

the product of q and a real scalar β is

$$\beta q = \beta a_q + \beta b_q i + \beta c_q j + \beta d_q k$$

Quaternion Multiplication by a Quaternion Scalar

The definition of the basis elements for quaternions,

$$i^2 = j^2 = k^2 = ijk = -1,$$

can be expanded to populate a table summarizing quaternion basis element multiplication:

	1	**i**	**j**	**k**
1	1	i	j	k
i	i	−1	k	−j
j	j	−k	−1	i
k	k	j	−i	−1

When reading the table, the rows are read first, for example: ij = k and ji = −k.

Given two quaternions, $q = a_q + b_q i + c_q j + d_q k$, and $p = a_p + b_p i + c_p j + d_p k$, the multiplication can be expanded as:

$$
\begin{aligned}
z = pq &= \left(a_p + b_p\mathrm{i} + c_p\mathrm{j} + d_p\mathrm{k}\right)\left(a_q + b_q\mathrm{i} + c_q\mathrm{j} + d_q\mathrm{k}\right) \\
&= a_p a_q + a_p b_q\mathrm{i} + a_p c_q\mathrm{j} + a_p d_q\mathrm{k} \\
&+ b_p a_q\mathrm{i} + b_p b_q\mathrm{i}^2 + b_p c_q\mathrm{ij} + b_p d_q\mathrm{ik} \\
&+ c_p a_q\mathrm{j} + c_p b_q\mathrm{ji} + c_p c_q\mathrm{j}^2 + c_p d_q\mathrm{jk} \\
&+ d_p a_q k + d_p b_q\mathrm{ki} + d_p c_q\mathrm{kj} + d_p d_q\mathrm{k}^2
\end{aligned}
$$

You can simplify the equation using the quaternion multiplication table.

$$
\begin{aligned}
z = pq &= a_p a_q + a_p b_q\mathrm{i} + a_p c_q\mathrm{j} + a_p d_q\mathrm{k} \\
&+ b_p a_q\mathrm{i} - b_p b_q + b_p c_q\mathrm{k} - b_p d_q\mathrm{j} \\
&+ c_p a_q\mathrm{j} - c_p b_q\mathrm{k} - c_p c_q + c_p d_q\mathrm{i} \\
&+ d_p a_q k + d_p b_q\mathrm{j} - d_p c_q\mathrm{i} - d_p d_q
\end{aligned}
$$

References

[1] Kuipers, Jack B. *Quaternions and Rotation Sequences: A Primer with Applications to Orbits, Aerospace, and Virtual Reality*. Princeton, NJ: Princeton University Press, 2007.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

Functions

`prod` | `mtimes, *`

Objects

`quaternion`

Topics

“Rotations, Orientation, and Quaternions”

Introduced in R2019b

transpose, .'

Transpose a quaternion array

Syntax

```
Y = quat.'
```

Description

`Y = quat.'` returns the non-conjugate transpose of the quaternion array, `quat`.

Examples

Vector Transpose

Create a vector of quaternions and compute its nonconjugate transpose.

```
quat = quaternion(randn(4,4))
```

```
quat = 4x1 quaternion array
      0.53767 +  0.31877i +   3.5784j +   0.7254k
       1.8339 -   1.3077i +   2.7694j - 0.063055k
      -2.2588 -  0.43359i -   1.3499j +  0.71474k
      0.86217 +  0.34262i +   3.0349j -  0.20497k
```

```
quatTransposed = quat.'
```

```
quatTransposed = 1x4 quaternion array
      0.53767 +  0.31877i +   3.5784j +   0.7254k       1.8339 -   1.3077i +   2.7694j - 0.06305
```

Matrix Transpose

Create a matrix of quaternions and compute its nonconjugate transpose.

```
quat = [quaternion(randn(2,4)),quaternion(randn(2,4))]
```

```
quat = 2x2 quaternion array
      0.53767 -   2.2588i +  0.31877j -  0.43359k       3.5784 -   1.3499i +   0.7254j +  0.7147
       1.8339 +  0.86217i -   1.3077j +  0.34262k       2.7694 +   3.0349i - 0.063055j -  0.2049
```

```
quatTransposed = quat.'
```

```
quatTransposed = 2x2 quaternion array
      0.53767 -   2.2588i +  0.31877j -  0.43359k       1.8339 +  0.86217i -   1.3077j +  0.3426
       3.5784 -   1.3499i +   0.7254j +  0.71474k       2.7694 +   3.0349i - 0.063055j -  0.2049
```

Input Arguments

quat — Quaternion array to transpose

vector | matrix

Quaternion array to transpose, specified as a vector or matrix of quaternions. `transpose` is defined for 1-D and 2-D arrays. For higher-order arrays, use `permute`.

Data Types: `quaternion`

Output Arguments

Y — Transposed quaternion array

vector | matrix

Transposed quaternion array, returned as an *N*-by-*M* array, where `quat` was specified as an *M*-by-*N* array.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

Functions

`ctranspose, '`

Objects

`quaternion`

Topics

“Rotations, Orientation, and Quaternions”

Introduced in R2019b

uminus, -

Quaternion unary minus

Syntax

```
mQuat = -quat
```

Description

`mQuat = -quat` negates the elements of `quat` and stores the result in `mQuat`.

Examples

Negate Elements of Quaternion Matrix

Unary minus negates each part of a the quaternion. Create a 2-by-2 matrix, `Q`.

```
Q = quaternion(randn(2),randn(2),randn(2),randn(2))
```

```
Q = 2x2 quaternion array
      0.53767 +  0.31877i +   3.5784j +   0.7254k      -2.2588 -  0.43359i -   1.3499j +  0.71474
       1.8339 -   1.3077i +   2.7694j - 0.063055k      0.86217 +  0.34262i +   3.0349j -  0.20497
```

Negate the parts of each quaternion in `Q`.

```
R = -Q
```

```
R = 2x2 quaternion array
     -0.53767 -  0.31877i -   3.5784j -   0.7254k       2.2588 +  0.43359i +   1.3499j -  0.71474
      -1.8339 +   1.3077i -   2.7694j + 0.063055k     -0.86217 -  0.34262i -   3.0349j +  0.20497
```

Input Arguments

`quat` — Quaternion array

scalar | vector | matrix | multidimensional array

Quaternion array, specified as a scalar, vector, matrix, or multidimensional array.

Data Types: `quaternion`

Output Arguments

`mQuat` — Negated quaternion array

scalar | vector | matrix | multidimensional array

Negated quaternion array, returned as the same size as `quat`.

Data Types: `quaternion`

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

Functions

`minus, -`

Objects

`quaternion`

Topics

“Rotations, Orientation, and Quaternions”

Introduced in R2019b

zeros

Create quaternion array with all parts set to zero

Syntax

```
quatZeros = zeros('quaternion')
quatZeros = zeros(n,'quaternion')
quatZeros = zeros(sz,'quaternion')
quatZeros = zeros(sz1,...,szN,'quaternion')

quatZeros = zeros(___,'like',prototype,'quaternion')
```

Description

`quatZeros = zeros('quaternion')` returns a scalar quaternion with all parts set to zero.

`quatZeros = zeros(n,'quaternion')` returns an `n`-by-`n` matrix of quaternions.

`quatZeros = zeros(sz,'quaternion')` returns an array of quaternions where the size vector, `sz`, defines `size(quatZeros)`.

`quatZeros = zeros(sz1,...,szN,'quaternion')` returns a `sz1`-by-...-by-`szN` array of quaternions where `sz1,…,szN` indicates the size of each dimension.

`quatZeros = zeros(___,'like',prototype,'quaternion')` specifies the underlying class of the returned quaternion array to be the same as the underlying class of the quaternion prototype.

Examples

Quaternion Scalar Zero

Create a quaternion scalar zero.

```
quatZeros = zeros('quaternion')
```

```
quatZeros = quaternion
     0 + 0i + 0j + 0k
```

Square Matrix of Quaternions

Create an n-by-n array of quaternion zeros.

```
n = 3;
quatZeros = zeros(n,'quaternion')
```

```
quatZeros = 3x3 quaternion array
     0 + 0i + 0j + 0k     0 + 0i + 0j + 0k     0 + 0i + 0j + 0k
```

```
     0 + 0i + 0j + 0k     0 + 0i + 0j + 0k     0 + 0i + 0j + 0k
     0 + 0i + 0j + 0k     0 + 0i + 0j + 0k     0 + 0i + 0j + 0k
```

Multidimensional Array of Quaternion Zeros

Create a multidimensional array of quaternion zeros by defining array dimensions in order. In this example, you create a 3-by-1-by-2 array. You can specify dimensions using a row vector or comma-separated integers.

Specify the dimensions using a row vector and display the results:

```
dims = [3,1,2];
quatZerosSyntax1 = zeros(dims,'quaternion')
```

```
quatZerosSyntax1 = 3x1x2 quaternion array
quatZerosSyntax1(:,:,1) =

     0 + 0i + 0j + 0k
     0 + 0i + 0j + 0k
     0 + 0i + 0j + 0k


quatZerosSyntax1(:,:,2) =

     0 + 0i + 0j + 0k
     0 + 0i + 0j + 0k
     0 + 0i + 0j + 0k
```

Specify the dimensions using comma-separated integers, and then verify the equivalence of the two syntaxes:

```
quatZerosSyntax2 = zeros(3,1,2,'quaternion');
isequal(quatZerosSyntax1,quatZerosSyntax2)
```

```
ans = logical
   1
```

Underlying Class of Quaternion Zeros

A quaternion is a four-part hyper-complex number used in three-dimensional representations. You can specify the underlying data type of the parts as `single` or `double`. The default is `double`.

Create a quaternion array of zeros with the underlying data type set to `single`.

```
quatZeros = zeros(2,'like',single(1),'quaternion')
```

```
quatZeros = 2x2 quaternion array
     0 + 0i + 0j + 0k     0 + 0i + 0j + 0k
     0 + 0i + 0j + 0k     0 + 0i + 0j + 0k
```

Verify the underlying class using the `classUnderlying` function.

```
classUnderlying(quatZeros)
```

```
ans =
'single'
```

Input Arguments

`n` — Size of square quaternion matrix

integer value

Size of square quaternion matrix, specified as an integer value. If `n` is `0` or negative, then `quatZeros` is returned as an empty matrix.

Example: `zeros(4,'quaternion')` returns a 4-by-4 matrix of quaternion zeros.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`sz` — Output size

row vector of integer values

Output size, specified as a row vector of integer values. Each element of `sz` indicates the size of the corresponding dimension in `quatZeros`. If the size of any dimension is `0` or negative, then `quatZeros` is returned as an empty array.

Example: `zeros([1,4,2],'quaternion')` returns a 1-by-4-by-2 array of quaternion zeros.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`prototype` — Quaternion prototype

variable

Quaternion prototype, specified as a variable.

Example: `zeros(2,'like',quat,'quaternion')` returns a 2-by-2 matrix of quaternions with the same underlying class as the prototype quaternion, `quat`.

Data Types: `quaternion`

`sz1,...,szN` — Size of each dimension

two or more integer values

Size of each dimension, specified as two or more integers.

- If the size of any dimension is `0`, then `quatZeros` is returned as an empty array.
- If the size of any dimension is negative, then it is treated as `0`.

Example: `zeros(2,3,'quaternion')` returns a 2-by-3 matrix of quaternion zeros.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Output Arguments

`quatZeros` — Quaternion zeros

scalar | vector | matrix | multidimensional array

Quaternion zeros, returned as a quaternion or array of quaternions.

Given a quaternion of the form $Q = a + b\mathrm{i} + c\mathrm{j} + d\mathrm{k}$, a quaternion zero is defined as $Q = 0 + 0\mathrm{i} + 0\mathrm{j} + 0\mathrm{k}$.

Data Types: `quaternion`

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

Functions

`ones`

Objects

`quaternion`

Topics

“Rotations, Orientation, and Quaternions”

Introduced in R2019b

rateControl

Execute loop at fixed frequency

Description

The `rateControl` object enables you to run a loop at a fixed frequency. It also collects statistics about the timing of the loop iterations. Use `waitfor` in the loop to pause code execution until the next time step. The loop operates every `DesiredPeriod` seconds, unless the enclosed code takes longer to operate. The object uses the `OverrunAction` property to determine how it handles longer loop operation times. The default setting, `'slip'`, immediately executes the loop if `LastPeriod` is greater than `DesiredPeriod`. Using `'drop'` causes the `waitfor` method to wait until the next multiple of `DesiredPeriod` is reached to execute the next loop.

Tip The scheduling resolution of your operating system and the level of other system activity can affect rate execution accuracy. As a result, accurate rate timing is limited to 100 Hz for execution of MATLAB code. To improve performance and execution speeds, use code generation.

Creation

Syntax

```
rateObj = rateControl(desiredRate)
```

Description

`rateObj = rateControl(desiredRate)` creates an object that operates loops at a fixed-rate based on your system time and directly sets the `DesireRate` property.

Properties

`DesiredRate` — Desired execution rate

scalar

Desired execution rate of loop, specified as a scalar in Hz. When using `waitfor`, the loop operates every `DesiredRate` seconds, unless the loop takes longer. It then begins the next loop based on the specified `OverrunAction`.

`DesiredPeriod` — Desired time period between executions

scalar

Desired time period between executions, specified as a scalar in seconds. This property is equal to the inverse of `DesiredRate`.

`TotalElapsedTime` — Elapsed time since construction or reset

scalar

Elapsed time since construction or reset, specified as a scalar in seconds.

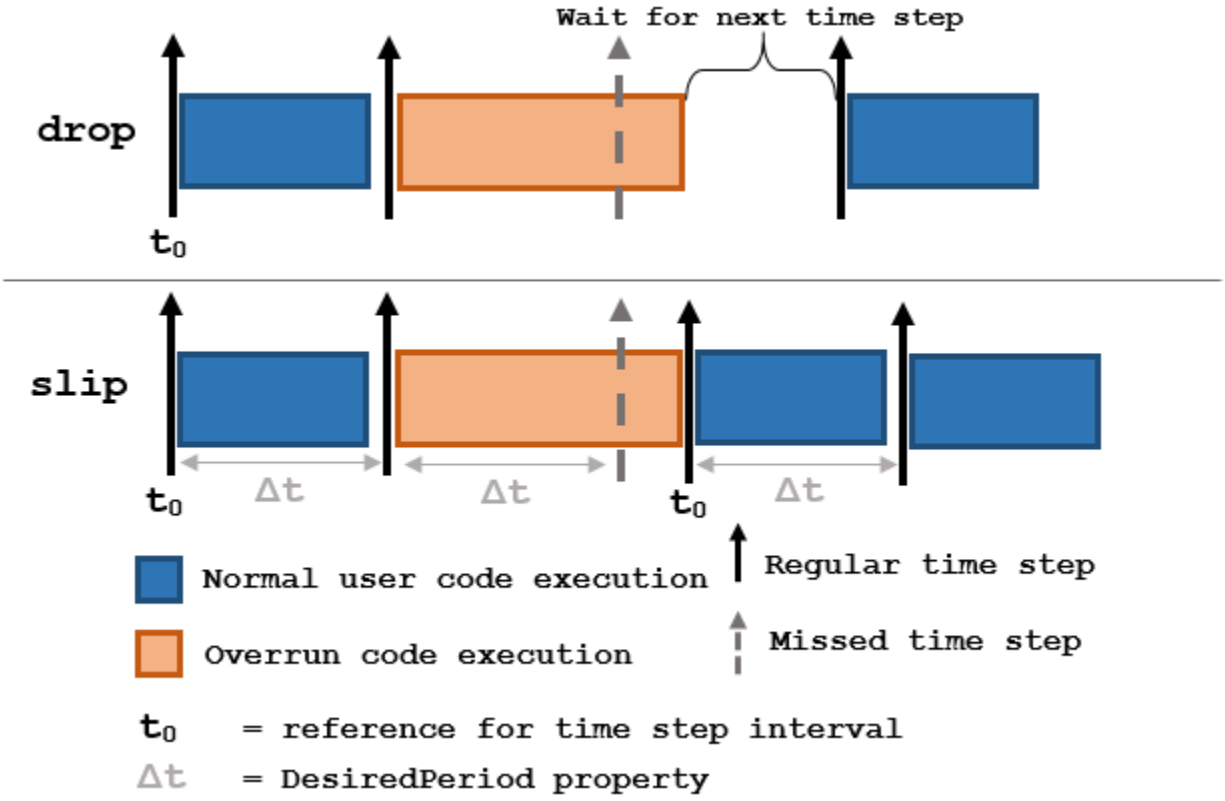
`LastPeriod` — Elapsed time between last two calls to `waitfor`
`NaN` (default) | scalar

Elapsed time between last two calls to `waitfor`, specified as a scalar. By default, `LastPeriod` is set to `NaN` until `waitfor` is called for the first time. After the first call, `LastPeriod` equals `TotalElapsedTime`.

`OverrunAction` — Method for handling overruns
`'slip'` (default) | `'drop'`

Method for handling overruns, specified as one of these character vectors:

- `'drop'` — waits until the next time interval equal to a multiple of `DesiredPeriod`
- `'slip'` — immediately executes the loop again



Each code section calls `waitfor` at the end of execution.

Object Functions

waitfor	Pause code execution to achieve desired execution rate
statistics	Statistics of past execution periods
reset	Reset Rate object

Examples

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: 0.004098
Iteration: 2 - Time Elapsed: 1.004890
Iteration: 3 - Time Elapsed: 2.009797
Iteration: 4 - Time Elapsed: 3.024798
Iteration: 5 - Time Elapsed: 4.004382
Iteration: 6 - Time Elapsed: 5.009306
Iteration: 7 - Time Elapsed: 6.002341
Iteration: 8 - Time Elapsed: 7.013342
Iteration: 9 - Time Elapsed: 8.004305
Iteration: 10 - Time Elapsed: 9.000212
```

Each iteration executes at a 1-second interval.

Get Statistics From Rate Object Execution

Create a `rateControl` object for running at 20 Hz.

```
r = rateControl(20);
```

Start a loop and control operation using the `rateControl` object.

```
for i = 1:30
    % Your code goes here
    waitfor(r);
end
```

Get `Rate` object statistics after loop operation.

```
stats = statistics(r)
```

```
stats = struct with fields:
              Periods: [0.0558 0.0610 0.0355 0.0551 0.0611 0.0494 0.2308 ... ]
           NumPeriods: 30
        AveragePeriod: 0.0551
    StandardDeviation: 0.0352
          NumOverruns: 1
```

Run Loop At Fixed Rate and Reset Rate Object

Create a `rateControl` object for running at 20 Hz.

```
r = rateControl(2);
```

Start a loop and control operation using the `Rate` object.

```
for i = 1:30
    % Your code goes here
    waitfor(r);
end
```

Display the `rateControl` object properties after loop operation.

```
disp(r)
```

```
  rateControl with properties:

         DesiredRate: 2
      DesiredPeriod: 0.5000
      OverrunAction: 'slip'
    TotalElapsedTime: 15.0135
          LastPeriod: 0.4916
```

Reset the object to restart the time statistics.

```
reset(r);
disp(r)
```

```
  rateControl with properties:

         DesiredRate: 2
      DesiredPeriod: 0.5000
      OverrunAction: 'slip'
    TotalElapsedTime: 0.0040
          LastPeriod: NaN
```

Compatibility Considerations

`rateControl` was renamed

Behavior change in future release

The `rateControl` object was renamed from `robotics.Rate`. Use `rateControl` for all object creation.

See Also

`rosrate` | `waitfor` | `statistics` | `reset`

Topics

“Execute Code at a Fixed-Rate”

Introduced in R2016a

reset

Reset `Rate` object

Syntax

```
reset(rate)
```

Description

`reset(rate)` resets the state of the `Rate` object, including the elapsed time and all statistics about previous periods. `reset` is useful if you want to run multiple successive loops at the same rate, or if the object is created before the loop is executed.

Input Arguments

`rate` — Rate object

handle

`Rate` object, specified as a handle. This object contains the information for the desired rate and other information about the execution. See `rateControl` for more information.

Examples

Run Loop At Fixed Rate and Reset Rate Object

Create a `rateControl` object for running at 20 Hz.

```
r = rateControl(2);
```

Start a loop and control operation using the `Rate` object.

```
for i = 1:30
    % Your code goes here
    waitfor(r);
end
```

Display the `rateControl` object properties after loop operation.

```
disp(r)
```

```
  rateControl with properties:

         DesiredRate: 2
      DesiredPeriod: 0.5000
      OverrunAction: 'slip'
    TotalElapsedTime: 15.0135
          LastPeriod: 0.4916
```

Reset the object to restart the time statistics.

```
reset(r);
disp(r)
```

```
  rateControl with properties:

        DesiredRate: 2
      DesiredPeriod: 0.5000
      OverrunAction: 'slip'
    TotalElapsedTime: 0.0040
         LastPeriod: NaN
```

See Also

`rateControl` | `waitfor`

Topics

“Execute Code at a Fixed-Rate”

Introduced in R2016a

statistics

Statistics of past execution periods

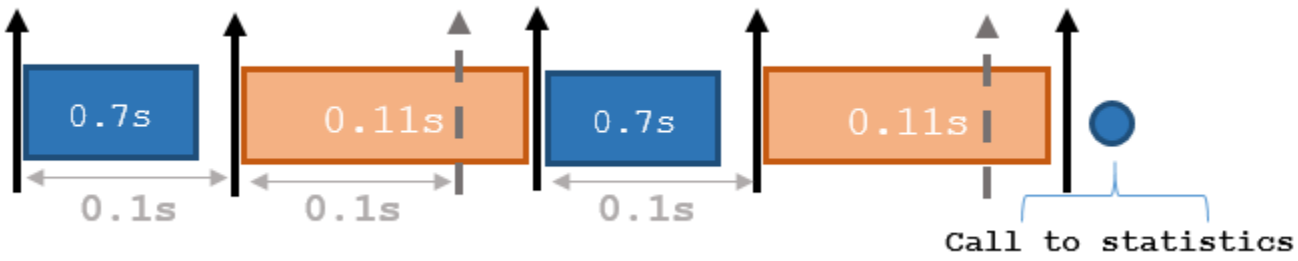
Syntax

```
stats = statistics(rate)
```

Description

`stats = statistics(rate)` returns statistics of previous periods of code execution. `stats` is a struct with these fields: `Periods`, `NumPeriods`, `AveragePeriod`, `StandardDeviation`, and `NumOverruns`.

Here is a sample execution graphic using the default setting, `'slip'`, for the `OverrunAction` property in the `Rate` object. See `OverrunAction` for more information on overrun code execution.



The output of `statistics` is:

```
stats =

              Periods: [0.7 0.11 0.7 0.11]
           NumPeriods: 4
        AveragePeriod: 0.09
    StandardDeviation: 0.0231
          NumOverruns: 2
```

Input Arguments

`rate` — Rate object

handle

`Rate` object, specified as an object handle. This object contains the information for the `DesiredRate` and other info about the execution. See `rateControl` for more information.

Output Arguments

`stats` — Time execution statistics

structure

Time execution statistics, returned as a structure. This structure contains the following fields:

- `Period` — All time periods (returned in seconds) used to calculate statistics as an indexed array. `stats.Period(end)` is the most recent period.
- `NumPeriods` — Number of elements in `Periods`
- `AveragePeriod` — Average time in seconds
- `StandardDeviation` — Standard deviation of all periods in seconds, centered around the mean stored in `AveragePeriod`
- `NumOverruns` — Number of periods with overrun

Examples

Get Statistics From Rate Object Execution

Create a `rateControl` object for running at 20 Hz.

```
r = rateControl(20);
```

Start a loop and control operation using the `rateControl` object.

```
for i = 1:30
    % Your code goes here
    waitfor(r);
end
```

Get `Rate` object statistics after loop operation.

```
stats = statistics(r)
```

```
stats = struct with fields:
              Periods: [0.0558 0.0610 0.0355 0.0551 0.0611 0.0494 0.2308 ... ]
           NumPeriods: 30
        AveragePeriod: 0.0551
    StandardDeviation: 0.0352
          NumOverruns: 1
```

See Also

`rateControl` | `waitfor`

Topics

“Execute Code at a Fixed-Rate”

Introduced in R2016a

waitfor

Package: `robotics`

Pause code execution to achieve desired execution rate

Syntax

```
waitfor(rate)
numMisses = waitfor(rate)
```

Description

`waitfor(rate)` pauses execution until the code reaches the desired execution rate. The function accounts for the time that is spent executing code between `waitfor` calls.

`numMisses = waitfor(rate)` returns the number of iterations missed while executing code between calls.

Examples

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: 0.004098
Iteration: 2 - Time Elapsed: 1.004890
Iteration: 3 - Time Elapsed: 2.009797
Iteration: 4 - Time Elapsed: 3.024798
Iteration: 5 - Time Elapsed: 4.004382
Iteration: 6 - Time Elapsed: 5.009306
Iteration: 7 - Time Elapsed: 6.002341
Iteration: 8 - Time Elapsed: 7.013342
Iteration: 9 - Time Elapsed: 8.004305
Iteration: 10 - Time Elapsed: 9.000212
```

Each iteration executes at a 1-second interval.

Input Arguments

`rate` — `Rate` object

handle

`Rate` object, specified as a handle. This object contains the information for the desired rate and other information about the execution. See `rateControl` for more information.

Output Arguments

`numMisses` — Number of missed task executions

scalar

Number of missed task executions, returned as a scalar. `waitfor` returns the number of times the task was missed in the `Rate` object based on the `LastPeriod` time. For example, if the desired rate is 1 Hz and the last period was 3.2 seconds, `numMisses` returns 3.

See Also

`rateControl`

Topics

“Execute Code at a Fixed-Rate”

Introduced in R2016a

reedsSheppConnection

Reeds-Shepp path connection type

Description

The `reedSheppConnection` object holds information for computing a `reedsSheppPathSegment` object to connect between poses. A Reeds-Shepp path segment connects two poses as a sequence of five motions. The motion options are:

- Straight
- Left turn at maximum steer
- Right turn at maximum steer
- No movement

A Reeds-Shepp path segment supports both forward and backward motion.

Use this connection object to define parameters for a vehicle motion model, including the minimum turning radius and options for path types. To generate a path segment between poses using this connection type, call the `connect` function.

Creation

Syntax

```
reedsConnObj = reedsSheppConnection
reedsConnObj = reedsSheppConnection(Name,Value)
```

Description

`reedsConnObj = reedsSheppConnection` creates an object using default property values.

`reedsConnObj = reedsSheppConnection(Name,Value)` specifies property values using name-value pairs. To set multiple properties, specify multiple name-value pairs.

Properties

`MinTurningRadius` — Minimum turning radius
1 (default) | positive scalar in meters

Minimum turning radius for the vehicle, specified as a positive scalar in meters. The minimum turning radius is for the smallest circle the vehicle can make with maximum steer in a single direction.

Data Types: `double`

`DisabledPathTypes` — Path types to disable
`{}` (default) | vector of string scalars | cell array of character vectors

Path types to disable, specified as a vector of string scalars or cell array of character vectors.

Motion Type	Description
`"Sp","Sn"`	Straight (`p` = forward, `n` = reverse)
`"Lp","Ln"`	Left turn at the maximum steering angle of the vehicle (`p` = forward, `n` = reverse)
`"Rp","Rn"`	Right turn at the maximum steering angle of the vehicle (`p` = forward, `n` = reverse)
`"N"`	No motion

If a path segment has fewer than five motion types, the remaining elements are `"N"` (no motion).

To see all available path types, see the `AllPathTypes` property.

Example: `["LpSnLp","LnSnRpSn","LnSnRpSnLp"]`

Data Types: `cell`

`AllPathTypes` — All possible path types
cell array of character vectors

This property is read-only.

All possible path types, specified as a cell array of character vectors. This property lists all types. To disable certain types, specify types from this list in `DisabledPathTypes`.

For Reeds-Shepp connections, there are 44 possible combinations of motion types.

Data Types: `cell`

`ForwardCost` — Cost multiplier to travel forward
1 (default) | positive numeric scalar

Cost multiple to travel forward, specified as a positive numeric scalar. Increase this property to penalize forward motion.

Data Types: `double`

`ReverseCost` — Cost multiplier to travel in reverse
1 (default) | positive numeric scalar

Cost multiple to travel in reverse, specified as a positive numeric scalar. Increase this property to penalize reverse motion.

Data Types: `double`

Object Functions

connect Connect poses for given connection type

Examples

Connect Poses Using ReedsShepp Connection Path

Create a `reedsSheppConnection` object.

```
reedsConnObj = reedsSheppConnection;
```

Define start and goal poses as `[x y theta]` vectors.

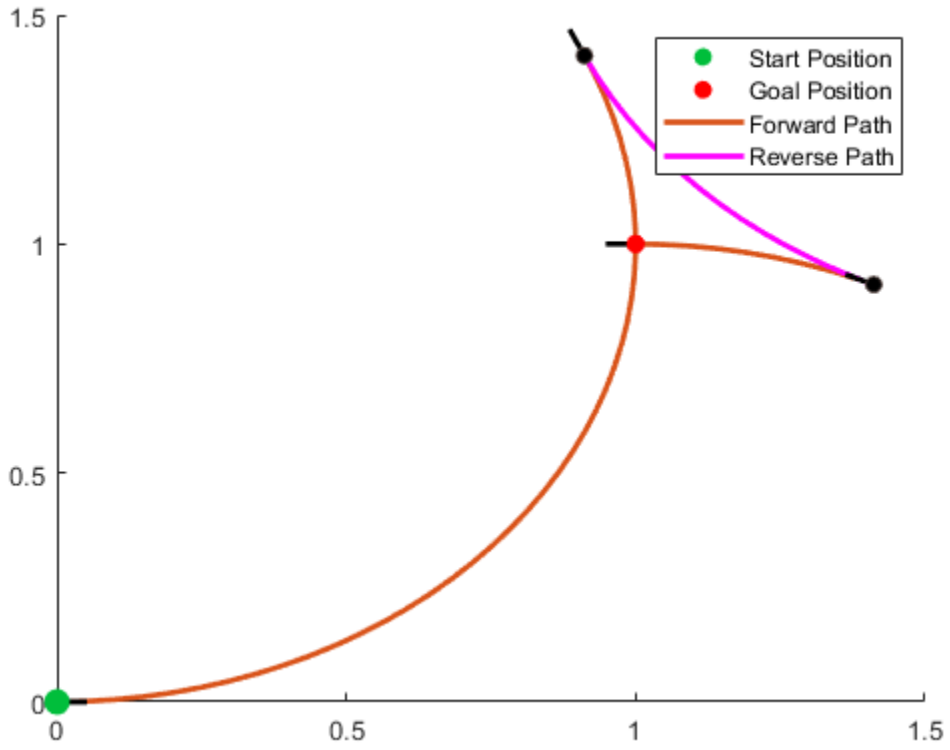
```
startPose = [0 0 0];
goalPose = [1 1 pi];
```

Calculate a valid path segment to connect the poses.

```
[pathSegObj,pathCosts] = connect(reedsConnObj,startPose,goalPose);
```

Show the generated path.

```
show(pathSegObj{1})
```



Modify Connection Types for Reeds-Shepp Path

Create a `reedsSheppConnection` object.

```
reedsConnObj = reedsSheppConnection;
```

Define start and goal poses as `[x y theta]` vectors.

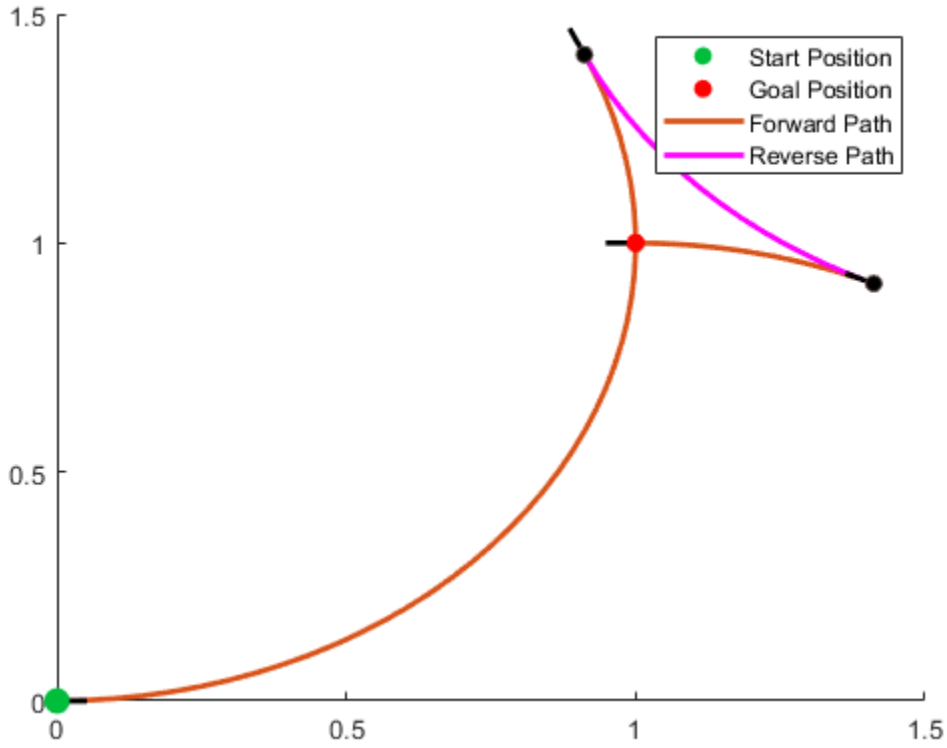
```
startPose = [0 0 0];
goalPose = [1 1 pi];
```

Calculate a valid path segment to connect the poses.

```
[pathSegObj,pathCosts] = connect(reedsConnObj,startPose,goalPose);
```

Show the generated path. Notice the direction of the turns.

```
show(pathSegObj{1})
```




```
pathSegObj{1}.MotionTypes
```

```
ans = 1x5 cell
    {'L'}    {'R'}    {'L'}    {'N'}    {'N'}
```

```
pathSegObj{1}.MotionDirections
```

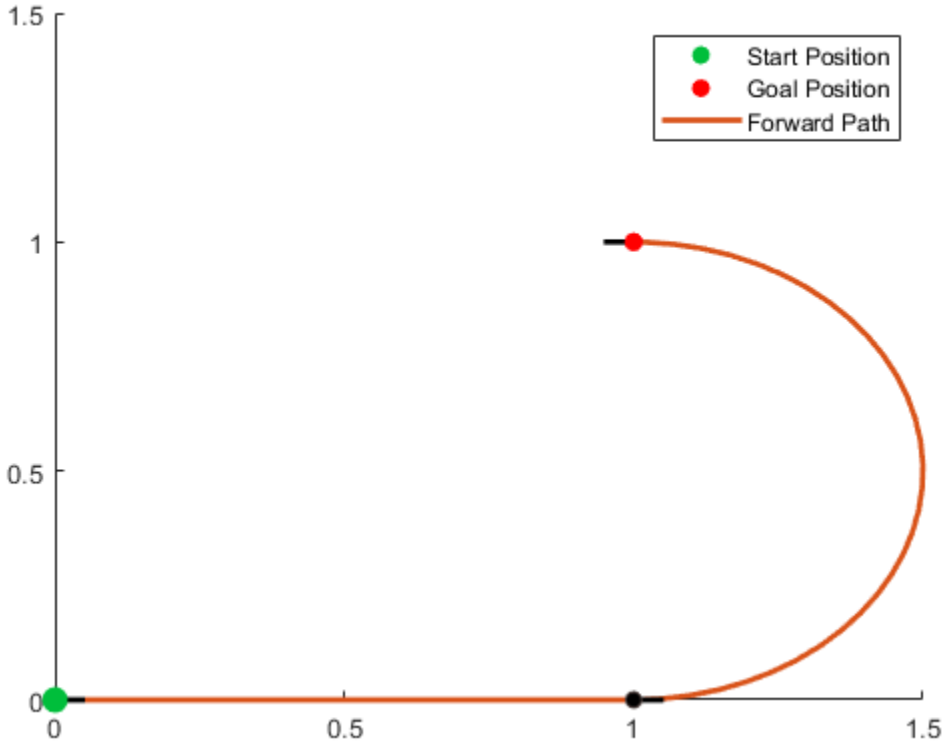
```
ans = 1×5

     1    -1     1     1     1
```

Disable this specific motion sequence in a new connection object. Reduce the `MinTurningRadius` if the robot is more maneuverable. Increase the reverse cost to reduce the likelihood of reverse directions being used. Connect the poses again to get a different path.

```
reedsConnObj = reedsSheppConnection('DisabledPathTypes',{'LpRnLp'});
reedsConnObj.MinTurningRadius = 0.5;
reedsConnObj.ReverseCost = 5;

[pathSegObj,pathCosts] = connect(reedsConnObj,startPose,goalPose);
pathSegObj{1}.MotionTypes
```

```
ans = 1x5 cell
    {'L'}    {'S'}    {'L'}    {'N'}    {'N'}
```

```
show(pathSegObj{1})
xlim([0 1.5])
ylim([0 1.5])
```



Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

Objects

`reedsSheppPathSegment` | `dubinsConnection` | `dubinsPathSegment`

Functions
`connect` | `interpolate` | `show`

Introduced in R2019b

reedsSheppPathSegment

Reeds-Shepp path segment connecting two poses

Description

The `reedSheppPathSegment` object holds information for a Reeds-Shepp path segment to connect between poses. A Reeds-Shepp path segment connects two poses as a sequence of five motion types. The motion options are:

- Straight
- Left turn at maximum steer
- Right turn at maximum steer
- No movement

Creation

To generate a `reedSheppPathSegment` object, use the `connect` function with a `reedsSheppConnection` object:

`reedsPathSegObj = connect(connectionObj,start,goal)` connects the start and goal poses using the specified connection type object.

To specifically define a path segment:

`reedsPathSegObj = reedsSheppPathSegment(connectionObj,start,goal,motionLengths,motionTypes)` specifies the Reeds-Shepp connection type, the start and goal poses, and the corresponding motion lengths and types. These values are set to the corresponding properties in the object.

Properties

`MinTurningRadius` — Minimum turning radius of vehicle
positive scalar

This property is read-only.

Minimum turning radius of the vehicle, specified as a positive scalar in meters. This value corresponds to the radius of the turning circle at the maximum steering angle of the vehicle.

Data Types: `double`

`StartPose` — Initial pose of vehicle
$[x, y, \Theta]$ vector

This property is read-only.

Initial pose of the vehicle at the start of the path segment, specified as an $[x, y, \Theta]$ vector. x and y are in meters. Θ is in radians.

Data Types: `double`

`GoalPose` — Goal pose of vehicle

[x, y, Θ] vector

This property is read-only.

Goal pose of the vehicle at the end of the path segment, specified as an [x, y, Θ] vector. x and y are in meters. Θ is in radians.

Data Types: `double`

`MotionLengths` — Length of each motion

five-element numeric vector

This property is read-only.

Length of each motion in the path segment, specified as a five-element numeric vector in meters. Each motion length corresponds to a motion type specified in `MotionTypes`.

Data Types: `double`

`MotionTypes` — Type of each motion

five-element string cell array

This property is read-only.

Type of each motion in the path segment, specified as a five-element string cell array.

Motion Type	Description
`"S"`	Straight (forward, `p` or reverse, `n`)
`"L"`	Left turn at the maximum steering angle of the vehicle (forward, `p` or reverse, `n`)
`"R"`	Right turn at the maximum steering angle of the vehicle (forward, `p` or reverse, `n`)
`"N"`	No motion

If a path segment has fewer than five motion types, the remaining elements are `"N"` (no motion).

Example: `{"L","S","R","L","R"}`

Data Types: `cell`

`MotionDirections` — Direction of each motion

five-element vector of 1s (forward motion) and –1s (reverse motion)

This property is read-only.

Direction of each motion in the path segment, specified as a five-element vector of 1s (forward motion) and –1s (reverse motion). Each motion direction corresponds to a motion length specified in `MotionLengths` and a motion type specified in `MotionTypes`.

When no motion occurs, that is, when a `MotionTypes` value is `"N"`, then the corresponding `MotionDirections` element is `1`.

Example: `[-1 1 -1 1 1]`

Data Types: `double`

`Length` — Length of path segment
positive scalar

This property is read-only.

Length of the path segment, specified as a positive scalar in meters. This length is just a sum of the elements in `MotionLengths`.

Data Types: `double`

Object Functions

interpolate	Interpolate poses along path segment
show	Visualize path segment

Examples

Connect Poses Using ReedsShepp Connection Path

Create a `reedsSheppConnection` object.

```
reedsConnObj = reedsSheppConnection;
```

Define start and goal poses as `[x y theta]` vectors.

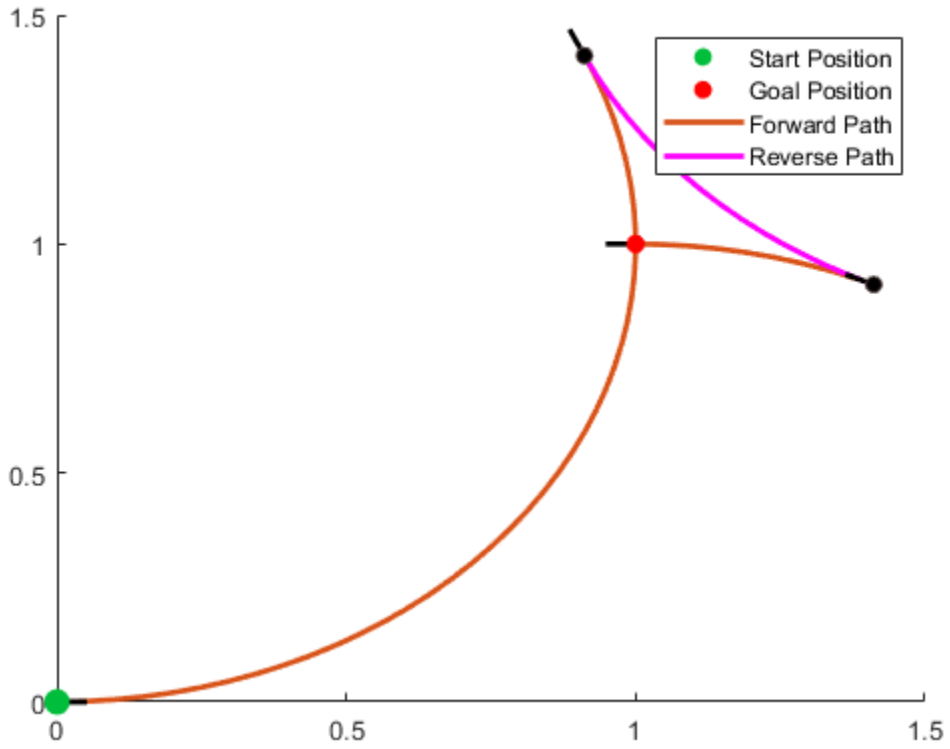
```
startPose = [0 0 0];
goalPose = [1 1 pi];
```

Calculate a valid path segment to connect the poses.

```
[pathSegObj,pathCosts] = connect(reedsConnObj,startPose,goalPose);
```

Show the generated path.

```
show(pathSegObj{1})
```



References

[1] Reeds, J. A., and L. A. Shepp. "Optimal Paths for a Car That Goes Both Forwards and Backwards." *Pacific Journal of Mathematics*. Vol. 145, Number 2, 1990, pp. 367–393.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

Objects

`reedsSheppConnection` | `dubinsConnection` | `dubinsPathSegment`

Functions

`interpolate` | `show` | `connect`

Introduced in R2019b

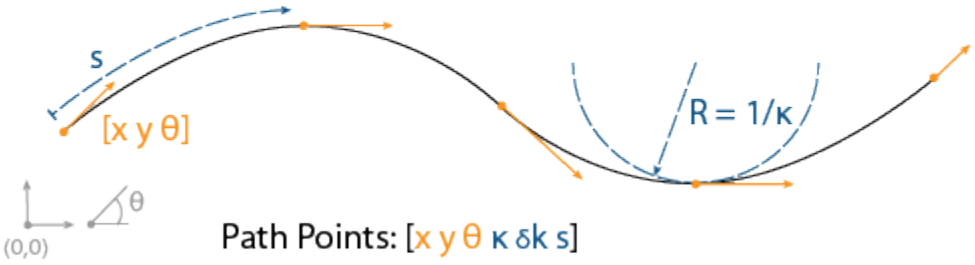
referencePathFrenet

Smooth reference path fit to waypoints

Description

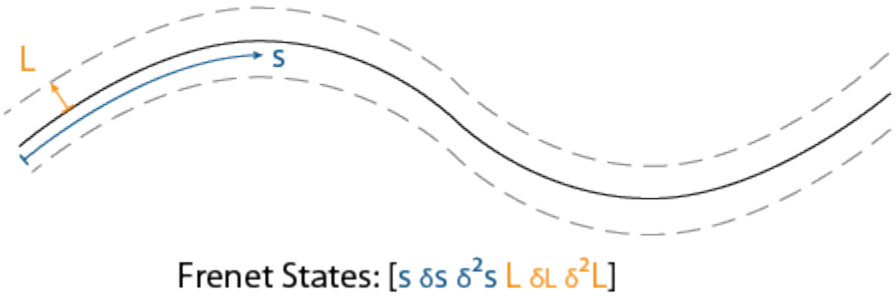
The `referencePathFrenet` object fits a smooth, piecewise, continuous curve to a set of waypoints given as `[x y]` or `[x y theta]`. After fitting, points along the curve, the path points are expressed as `[x y theta kappa dkappa s]`, where:

- `x` `y` and `theta`— SE(2) state expressed in global coordinates, with `x` and `y` in meters and `theta` in radians
- `kappa` — Curvature, or inverse of the radius, in meters
- `dkappa` — Derivative of curvature with respect to arc length in meters per second
- `s` — Arc length, or distance along path from path origin, in meters



Using this object, convert trajectories between global and Frenet coordinate systems, interpolate states along the path based on arc length, and query the closest point on a path from a global state. The absolute difference between `theta` and the orientation of the closest point on the path from a global state must not exceed pi/2 when converting between global and Frenet coordinate systems.

The object expresses Frenet states as a vector of form `[S dS ddS L dL ddL]`, where `S` is the arc length and `L` is the perpendicular deviation from the direction of the reference path. Derivatives of `S` are relative to time. Derivatives of `L` are relative to the arc length, `S`.



Creation

Syntax

```
refPathObj = referencePathFrenet(waypoints)
refPathObj = referencePathFrenet(
waypoints,'DiscretizationDistance',discretionDist)
```

Description

`refPathObj = referencePathFrenet(waypoints)` fits a piecewise, continuous set of curves to the specified waypoints. The `waypoints` argument sets the `Waypoints` property.

`refPathObj = referencePathFrenet(
waypoints,'DiscretizationDistance',discretionDist)` fits a piecewise, continuous set of curves to waypoints using the specified distance between interpolated path points. The `discretionDist` argument sets the `DiscretizationDistance` property.

Properties

`DiscretizationDistance` — Arc length between interpolated path points

`0.05` (default) | positive scalar

Arc length between interpolated path points, specified as a positive scalar in meters. The object uses interpolated path points to accelerate performance of the transformation functions `frenet2global` and `global2frenet`. A smaller discretization distance can improve accuracy at the expense of memory and computational efficiency.

Data Types: `single` | `double`

`Waypoints` — Presampled points along path

P-by-2 numeric matrix | *P*-by-3 numeric matrix

Presampled points along the path, specified as a *P*-by-2 matrix with rows of form `[x y]` or *P*-by-3 matrix with rows of form `[x y theta]`. Specify `x` and `y` in meters and `theta` in radians. *P* is the number of presampled points, and must be greater than or equal to two

Data Types: `single` | `double`

Object Functions

closestPoint	Find closest point on reference path to global point
frenet2global	Convert Frenet states to global states
global2frenet	Convert global states to Frenet states
interpolate	Interpolate reference path at provided arc lengths
show	Display reference path in figure

Examples

Generate Alternative Trajectories for Reference Path

Generate alternative trajectories for a reference path using Frenet coordinates. Specify different initial and terminal states for your trajectories. Tune your states based on the generated trajectories.

Generate a reference path from a set of waypoints. Create a `trajectoryGeneratorFrenet` object from the reference path.

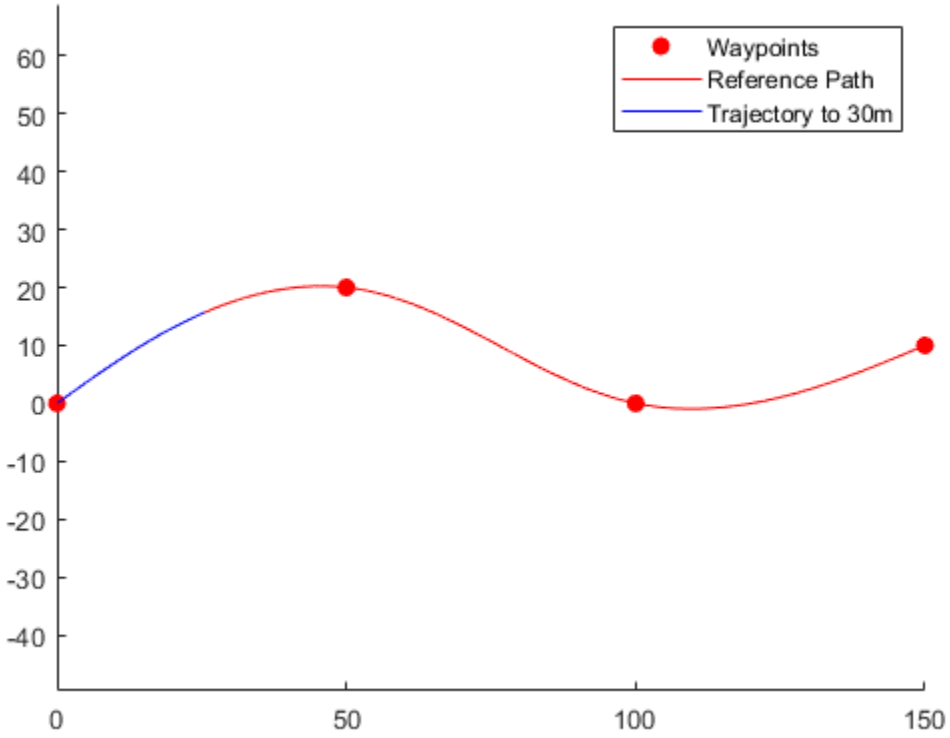
```
waypoints = [0 0; ...
    50 20; ...
    100 0; ...
    150 10];
refPath = referencePathFrenet(waypoints);
connector = trajectoryGeneratorFrenet(refPath);
```

Generate a five-second trajectory between the path origin and a point 30 m down the path as Frenet states.

```
initState = [0 0 0 0 0 0];  % [S ds ddS L dL ddL]
termState = [30 0 0 0 0 0]; % [S ds ddS L dL ddL]
[~,trajGlobal] = connect(connector,initState,termState,5);
```

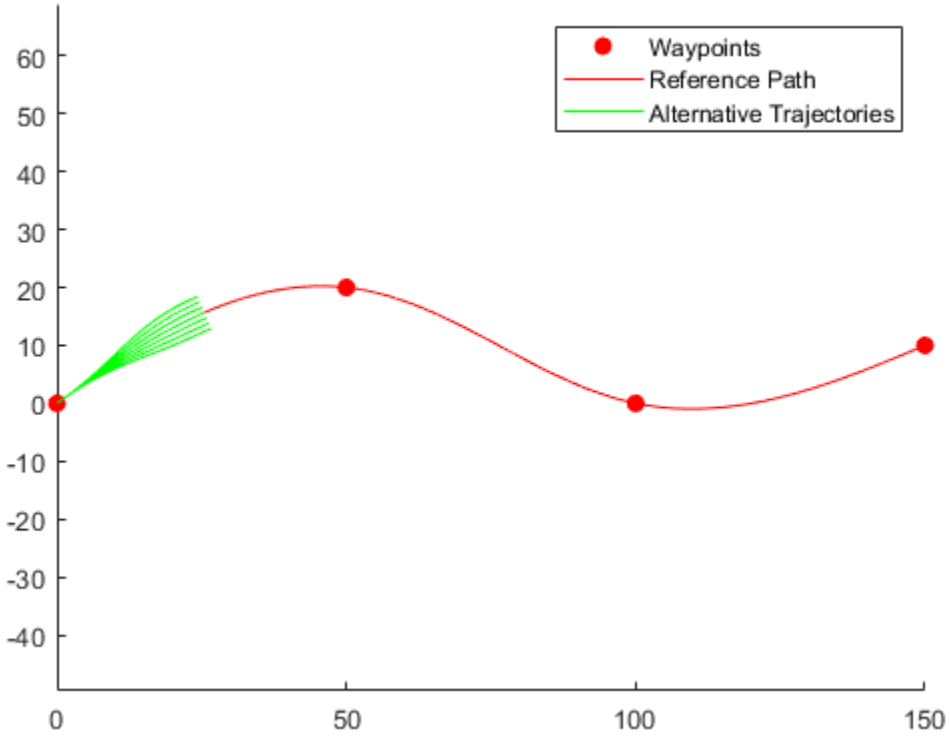
Display the trajectory in global coordinates.

```
show(refPath);
hold on
axis equal
plot(trajGlobal.Trajectory(:,1),trajGlobal.Trajectory(:,2),'b')
legend(["Waypoints","Reference Path","Trajectory to 30m"])
```



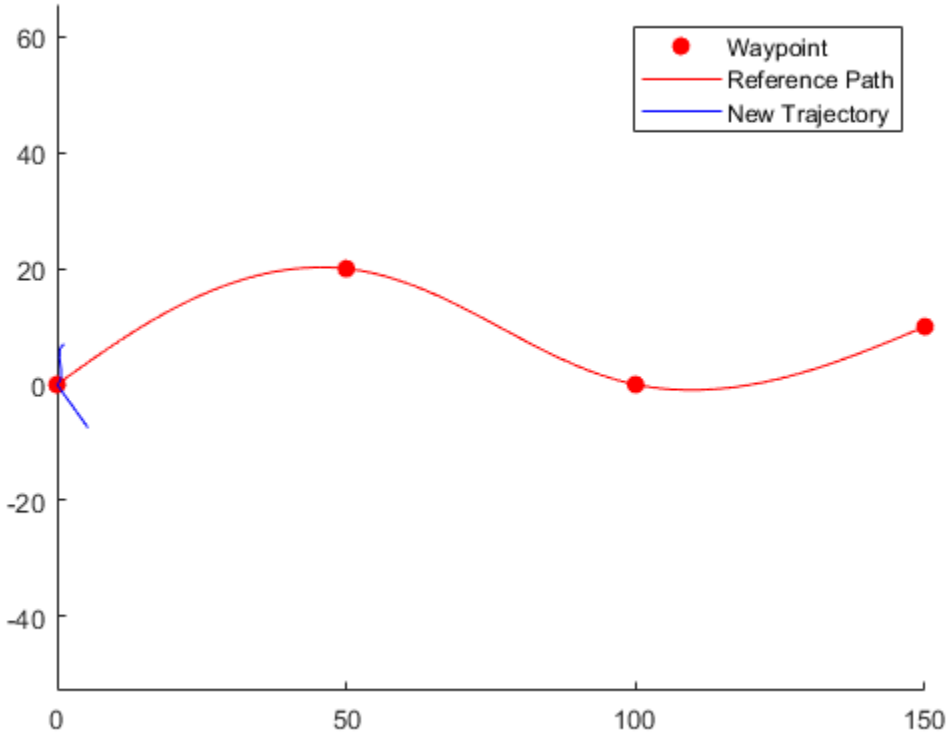
Create a matrix of terminal states with lateral deviations between –3 m and 3 m. Generate trajectories that cover the same arc length in 10 seconds, but deviate laterally from the reference path. Display the new alternative paths.

```
termStateDeviated = termState + ([-3:3]' * [0 0 0 1 0 0]);
[~,trajGlobal] = connect(connector,initState,termStateDeviated,10);

clf
show(refPath);
hold on
axis equal
for i = 1:length(trajGlobal)
    plot(trajGlobal(i).Trajectory(:,1),trajGlobal(i).Trajectory(:,2),'g')
end
legend(["Waypoints","Reference Path","Alternative Trajectories"])
hold off
```



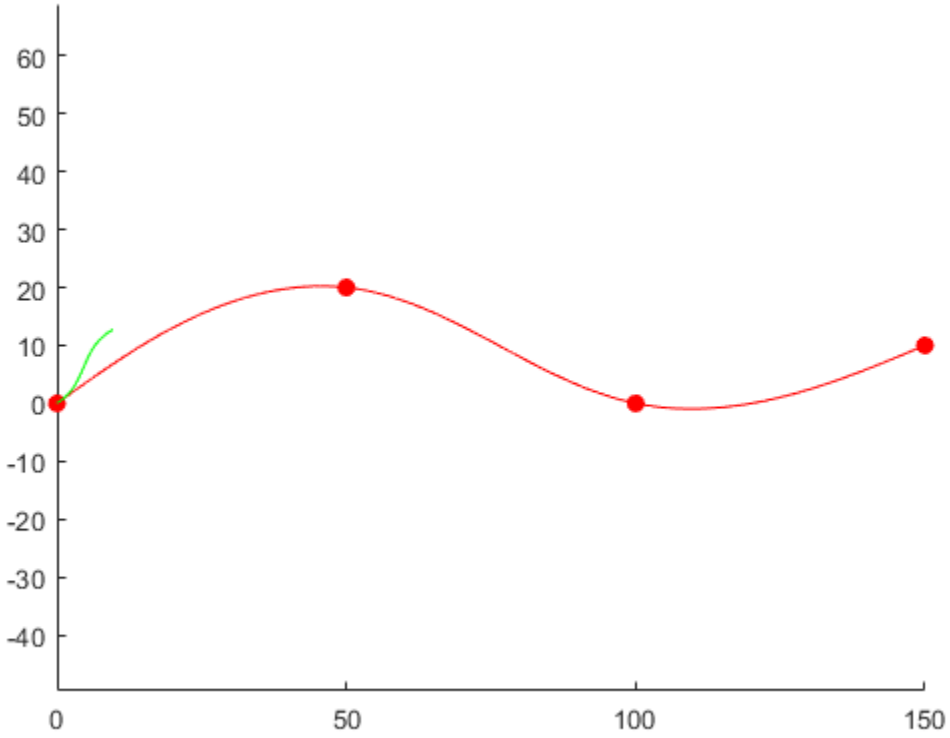
Specify a new terminal state to generate a new trajectory. This trajectory is not desirable because it requires reverse motion to achieve a lateral velocity of 10 m/s.

```
newTermState = [5 10 0 5 0 0];
[~,newTrajGlobal] = connect(connector,initState,newTermState,3);

clf
show(refPath);
hold on
axis equal
plot(newTrajGlobal.Trajectory(:,1),newTrajGlobal.Trajectory(:,2),'b');
legend(["Waypoint","Reference Path","New Trajectory"])
hold off
```



Relax the restriction on the longitudinal state by specifying an arc length of `NaN`. Generate and display the trajectory again. The new position shows a good alternative trajectory that deviates off the reference path.

```
relaxedTermState = [NaN 10 0 5 0 0];
[~,trajGlobalRelaxed] = connect(connector,initState,relaxedTermState,3);

clf
show(refPath);
hold on
axis equal
plot(trajGlobalRelaxed.Trajectory(:,1),trajGlobalRelaxed.Trajectory(:,2),'g');
hold off
```



Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

Objects

`trajectoryGeneratorFrenet` | `navPath`

Functions

`closestPoint` | `frenet2global` | `global2frenet` | `interpolate` | `show`

Topics

“Highway Trajectory Planning Using Frenet Reference Path”

Introduced in R2020b

closestPoint

Find closest point on reference path to global point

Syntax

```
pathPoints = closestPoint(refPath,points)
```

Description

`pathPoints = closestPoint(refPath,points)` finds the closest point on the reference path to each of the specified (*x*,*y*)-positions `points`.

Examples

Generate Trajectory from Reference Path

Generate a reference path from a set of waypoints.

```
waypoints = [0 0; 50 20; 100 0; 150 10];
refPath = referencePathFrenet(waypoints);
```

Create a trajectoryGeneratorFrenet object from the reference path.

```
connector = trajectoryGeneratorFrenet(refPath);
```

Generate a five-second trajectory between the path origin and a point 30 m down the path as Frenet states.

```
initState = [0 0 0 0 0 0];  % [S dS ddS L dL ddL]
termState = [30 0 0 0 0 0]; % [S dS ddS L dL ddL]
frenetTraj = connect(connector,initState,termState,5);
```

Convert the trajectory to the global states.

```
globalTraj = frenet2global(refPath,frenetTraj.Trajectory);
```

Display the reference path and the trajectory.

```
show(refPath);
axis equal
hold on
plot(globalTraj(:,1),globalTraj(:,2),'b')
```

Specify global points and find the closest points on reference path.

```
globalPoints = waypoints(2:end,:) + [20 -50];
nearestPathPoint = closestPoint(refPath,globalPoints);
```

Display the global points and the closest points on reference path.

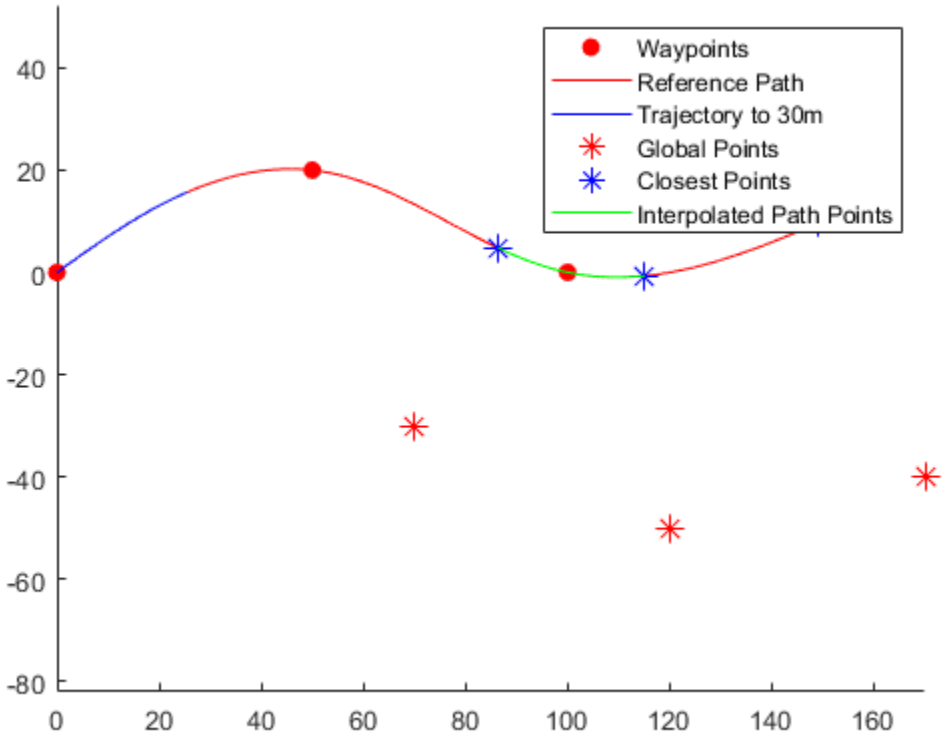
```
plot(globalPoints(:,1),globalPoints(:,2),'r*','MarkerSize',10)
plot(nearestPathPoint(:,1),nearestPathPoint(:,2),'b*','MarkerSize',10)
```

Interpolate between the arc lengths of the first two closest points along the reference path.

```
arclengths = linspace(nearestPathPoint(1,6),nearestPathPoint(2,6),10);
pathStates = interpolate(refPath,arclengths);
```

Display the interpolated path points.

```
plot(pathStates(:,1),pathStates(:,2),'g')
legend(["Waypoints","Reference Path","Trajectory to 30m",...
        "Global Points","Closest Points","Interpolated Path Points"])
```



Input Arguments

`refPath` — Reference path

`referencePathFrenet` object

Reference path, specified as a `referencePathFrenet` object.

`points` — Global points

P-by-2 numeric matrix

Global points, specified as a *P*-by-2 numeric matrix with rows of the form `[x y]`. *P* is the number of points. Positions are in meters.

Output Arguments

`pathPoints` — Closest points on reference path

N-by-6 numeric matrix

Closest points on the reference path , returned as an *N*-by-6 numeric matrix with rows of form `[x y theta kappa dkappa s]`, where:

- `x` `y` and `theta`— SE(2) state expressed in global coordinates, with `x` and `y` in meters and `theta` in radians
- `kappa` — Curvature, or inverse of the radius, in meters
- `dkappa` — Derivative of curvature with respect to arc length in meters per second
- `s` — Arc length, or distance along path from path origin, in meters

N is the number of points sampled along the reference path.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

Objects

`referencePathFrenet` | `trajectoryGeneratorFrenet` | `navPath`

Functions

`frenet2global` | `global2frenet` | `interpolate` | `show`

Topics

“Highway Trajectory Planning Using Frenet Reference Path”

Introduced in R2020b

frenet2global

Convert Frenet states to global states

Syntax

```
globalState = frenet2global(refPath,frenetState)
```

Description

`globalState = frenet2global(refPath,frenetState)` converts Frenet trajectory states to global states.

Examples

Generate Trajectory from Reference Path

Generate a reference path from a set of waypoints.

```
waypoints = [0 0; 50 20; 100 0; 150 10];
refPath = referencePathFrenet(waypoints);
```

Create a trajectoryGeneratorFrenet object from the reference path.

```
connector = trajectoryGeneratorFrenet(refPath);
```

Generate a five-second trajectory between the path origin and a point 30 m down the path as Frenet states.

```
initState = [0 0 0 0 0 0];  % [S dS ddS L dL ddL]
termState = [30 0 0 0 0 0]; % [S dS ddS L dL ddL]
frenetTraj = connect(connector,initState,termState,5);
```

Convert the trajectory to the global states.

```
globalTraj = frenet2global(refPath,frenetTraj.Trajectory);
```

Display the reference path and the trajectory.

```
show(refPath);
axis equal
hold on
plot(globalTraj(:,1),globalTraj(:,2),'b')
```

Specify global points and find the closest points on reference path.

```
globalPoints = waypoints(2:end,:) + [20 -50];
nearestPathPoint = closestPoint(refPath,globalPoints);
```

Display the global points and the closest points on reference path.

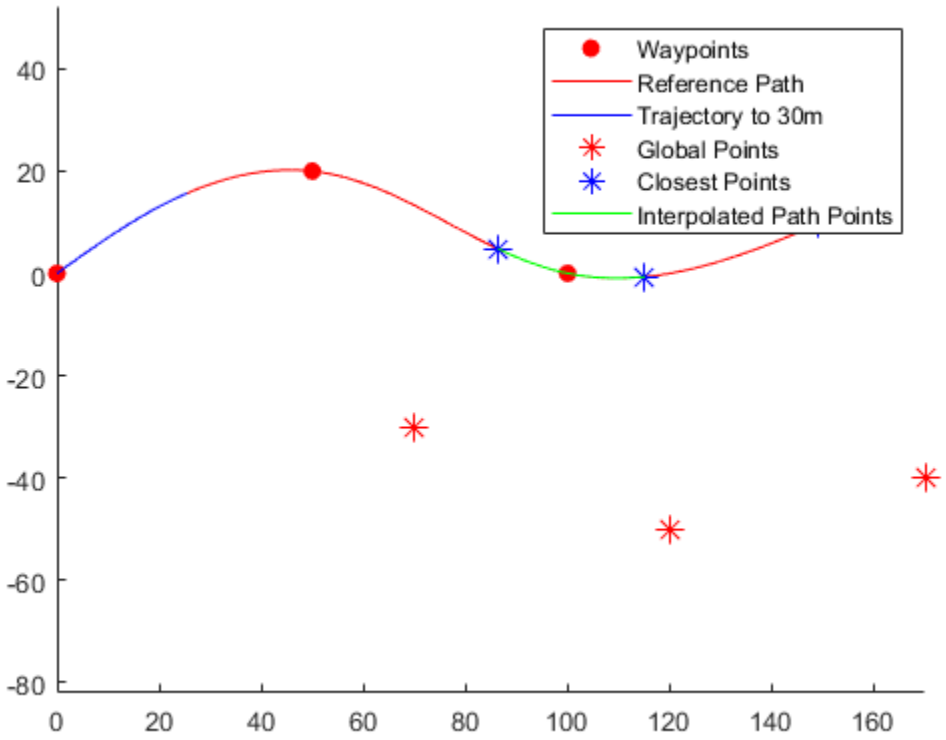
```
plot(globalPoints(:,1),globalPoints(:,2),'r*','MarkerSize',10)
plot(nearestPathPoint(:,1),nearestPathPoint(:,2),'b*','MarkerSize',10)
```

Interpolate between the arc lengths of the first two closest points along the reference path.

```
arclengths = linspace(nearestPathPoint(1,6),nearestPathPoint(2,6),10);
pathStates = interpolate(refPath,arclengths);
```

Display the interpolated path points.

```
plot(pathStates(:,1),pathStates(:,2),'g')
legend(["Waypoints","Reference Path","Trajectory to 30m",...
        "Global Points","Closest Points","Interpolated Path Points"])
```



Input Arguments

`refPath` — Reference path

`referencePathFrenet` object

Reference path, specified as a `referencePathFrenet` object.

`frenetState` — Trajectory in Frenet coordinate frame

P-by-6 numeric matrix

Trajectory in the Frenet coordinate frame, specified as a *P*-by-6 numeric matrix with rows of form `[S dS ddS L dL ddL]`, where `S` is the arc length and `L` is the perpendicular deviation from the direction of the reference path. Derivatives of `S` are relative to time. Derivatives of `L` are relative to the arc length, `S`. *P* is the number of Frenet states specified.

Output Arguments

`globalState` — Trajectory in global coordinate frame

P-by-6 numeric matrix

Trajectories in the global coordinate frame, returned as a *P*-by-6 numeric matrix with rows of form `[x y theta kappa speed accel]`, where:

- `x` `y` and `theta` –– SE(2) state expressed in global coordinates, with `x` and `y` in meters and `theta` in radians
- `kappa` –– Curvature, or inverse of the radius, in meters
- `speed` –– Speed in the `theta` direction in m/s
- `accel` –– Acceleration in the `theta` direction in m/s^2

P is the number of Frenet trajectories converted to global trajectories.

The absolute difference between `theta` and the orientation of the closest point on the path from a global state must not exceed pi/2 when converting between global and Frenet coordinate systems.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

Objects

`referencePathFrenet` | `trajectoryGeneratorFrenet` | `navPath`

Functions

`closestPoint` | `global2frenet` | `interpolate` | `show`

Topics

“Highway Trajectory Planning Using Frenet Reference Path”

Introduced in R2020b

global2frenet

Convert global states to Frenet states

Syntax

```
frenetState = global2frenet(refPath,globalState)
```

Description

`frenetState = global2frenet(refPath,globalState)` converts global states to Frenet trajectory states.

Input Arguments

`refPath` — Reference path

`referencePathFrenet` object

Reference path, specified as a `referencePathFrenet` object.

`globalState` — Trajectory in global coordinate frame

P-by-6 numeric matrix

Trajectories in the global coordinate frame, specified as a *P*-by-6 numeric matrix with rows of form `[x y theta kappa speed accel]`, where:

- `x` `y` and `theta` –- SE(2) state expressed in global coordinates, with `x` and `y` in meters and `theta` in radians
- `kappa` –- Curvature, or inverse of the radius, in meters
- `speed` –- Speed in the `theta` direction in m/s
- `accel` –- Acceleration in the `theta` direction in m/s^2

P is the number of Frenet trajectories converted to global trajectories.

The absolute difference between `theta` and the orientation of the closest point on the path from a global state must not exceed pi/2 when converting between global and Frenet coordinate systems.

Output Arguments

`frenetState` — Trajectory in Frenet coordinate frame

P-by-6 numeric matrix

Trajectory in the Frenet coordinate frame, returned as a *P*-by-6 numeric matrix with rows of form `[S dS ddS L dL ddL]`, where `S` is the arc length and `L` is the perpendicular deviation from the direction of the reference path. Derivatives of `S` are relative to time. Derivatives of `L` are relative to the arc length, `S`. *P* is the number of Frenet states specified.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

Objects

`referencePathFrenet` | `trajectoryGeneratorFrenet` | `navPath`

Functions

`closestPoint` | `frenet2global` | `global2frenet` | `interpolate` | `show`

Topics

“Highway Trajectory Planning Using Frenet Reference Path”

Introduced in R2020b

interpolate

Interpolate reference path at provided arc lengths

Syntax

```
pathPoints = interpolate(refPath,arclengths)
```

Description

`pathPoints = interpolate(refPath,arclengths)` interpolates the reference path at the provided arc lengths and returns the interpolated points on the path in global coordinates.

Examples

Generate Trajectory from Reference Path

Generate a reference path from a set of waypoints.

```
waypoints = [0 0; 50 20; 100 0; 150 10];
refPath = referencePathFrenet(waypoints);
```

Create a trajectoryGeneratorFrenet object from the reference path.

```
connector = trajectoryGeneratorFrenet(refPath);
```

Generate a five-second trajectory between the path origin and a point 30 m down the path as Frenet states.

```
initState = [0 0 0 0 0 0];  % [S dS ddS L dL ddL]
termState = [30 0 0 0 0 0]; % [S dS ddS L dL ddL]
frenetTraj = connect(connector,initState,termState,5);
```

Convert the trajectory to the global states.

```
globalTraj = frenet2global(refPath,frenetTraj.Trajectory);
```

Display the reference path and the trajectory.

```
show(refPath);
axis equal
hold on
plot(globalTraj(:,1),globalTraj(:,2),'b')
```

Specify global points and find the closest points on reference path.

```
globalPoints = waypoints(2:end,:) + [20 -50];
nearestPathPoint = closestPoint(refPath,globalPoints);
```

Display the global points and the closest points on reference path.

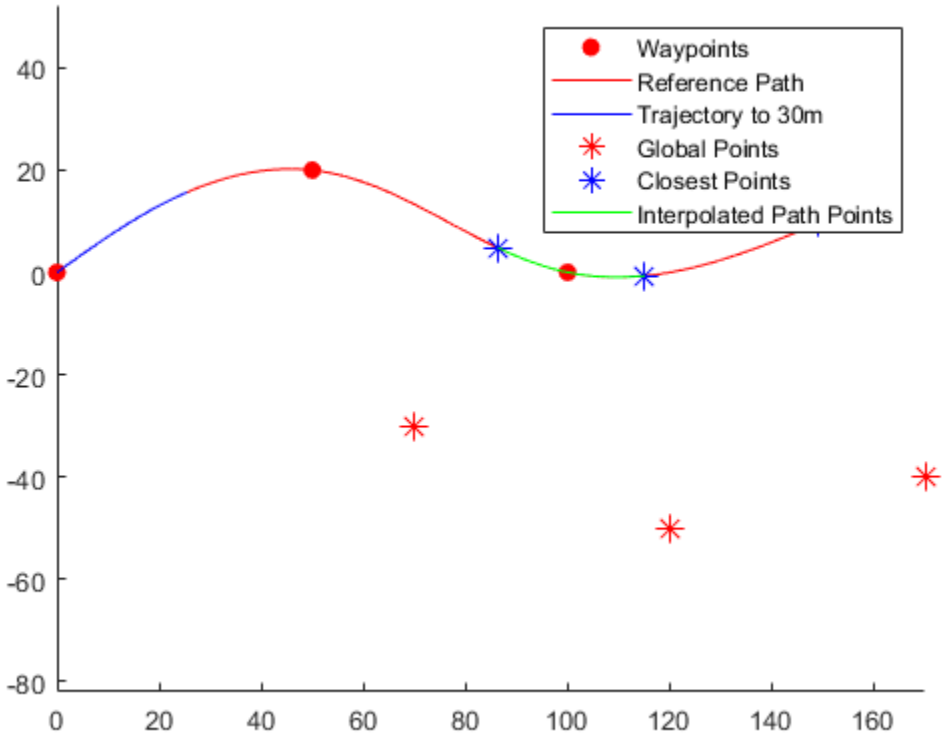
```
plot(globalPoints(:,1),globalPoints(:,2),'r*','MarkerSize',10)
plot(nearestPathPoint(:,1),nearestPathPoint(:,2),'b*','MarkerSize',10)
```

Interpolate between the arc lengths of the first two closest points along the reference path.

```
arclengths = linspace(nearestPathPoint(1,6),nearestPathPoint(2,6),10);
pathStates = interpolate(refPath,arclengths);
```

Display the interpolated path points.

```
plot(pathStates(:,1),pathStates(:,2),'g')
legend(["Waypoints","Reference Path","Trajectory to 30m",...
        "Global Points","Closest Points","Interpolated Path Points"])
```



Input Arguments

`refPath` — Reference path

`referencePathFrenet` object

Reference path, specified as a `referencePathFrenet` object.

`arclengths` — Distances along reference path

N-element vector

Distances along the reference path, specified as an *N*-element vector. *N* is the number of arc lengths at which to sample points. Each distance is in meters.

Output Arguments

`pathPoints` — Points on reference path

N-by-6 numeric matrix

Points on the reference path , returned as an *N*-by-6 numeric matrix with rows of form `[x y theta kappa dkappa s]`, where:

- `x` `y` and `theta`— SE(2) state expressed in global coordinates, with `x` and `y` in meters and `theta` in radians
- `kappa` — Curvature, or inverse of the radius, in meters
- `dkappa` — Derivative of curvature with respect to arc length in meters per second
- `s` — Arc length, or distance along path from path origin, in meters

N is the number of points sampled along the reference path.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

Objects

`referencePathFrenet` | `trajectoryGeneratorFrenet` | `navPath`

Functions

`closestPoint` | `frenet2global` | `global2frenet` | `show`

Topics

“Highway Trajectory Planning Using Frenet Reference Path”

Introduced in R2020b

show

Display reference path in figure

Syntax

```
show(refPath)
show(refPath,'Parent',parentAx)
ax = show( ___ )
```

Description

`show(refPath)` displays the reference path and its lateral states in the current figure.

`show(refPath,'Parent',parentAx)` displays the reference path on the specified axes. `parentAx` is specified as an `Axes` handle.

`ax = show( ___ )` displays the reference path using any of the previous input combinations and returns the axes handle on which the reference path is plotted.

Examples

Generate Alternative Trajectories for Reference Path

Generate alternative trajectories for a reference path using Frenet coordinates. Specify different initial and terminal states for your trajectories. Tune your states based on the generated trajectories.

Generate a reference path from a set of waypoints. Create a `trajectoryGeneratorFrenet` object from the reference path.

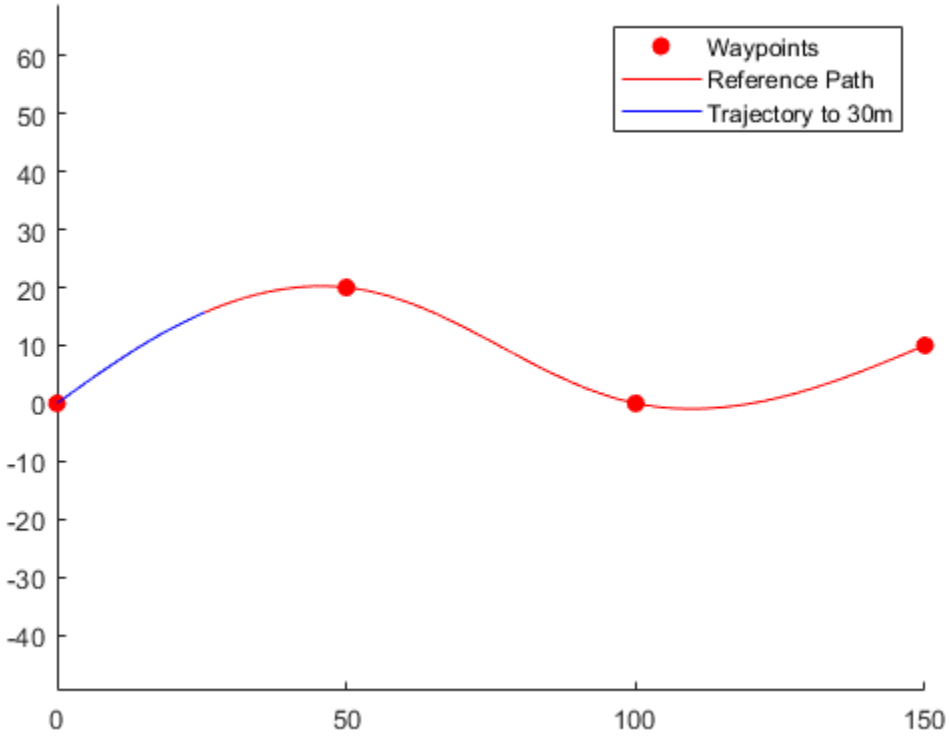
```
waypoints = [0 0; ...
    50 20; ...
    100 0; ...
    150 10];
refPath = referencePathFrenet(waypoints);
connector = trajectoryGeneratorFrenet(refPath);
```

Generate a five-second trajectory between the path origin and a point 30 m down the path as Frenet states.

```
initState = [0 0 0 0 0 0];  % [S ds ddS L dL ddL]
termState = [30 0 0 0 0 0]; % [S ds ddS L dL ddL]
[~,trajGlobal] = connect(connector,initState,termState,5);
```

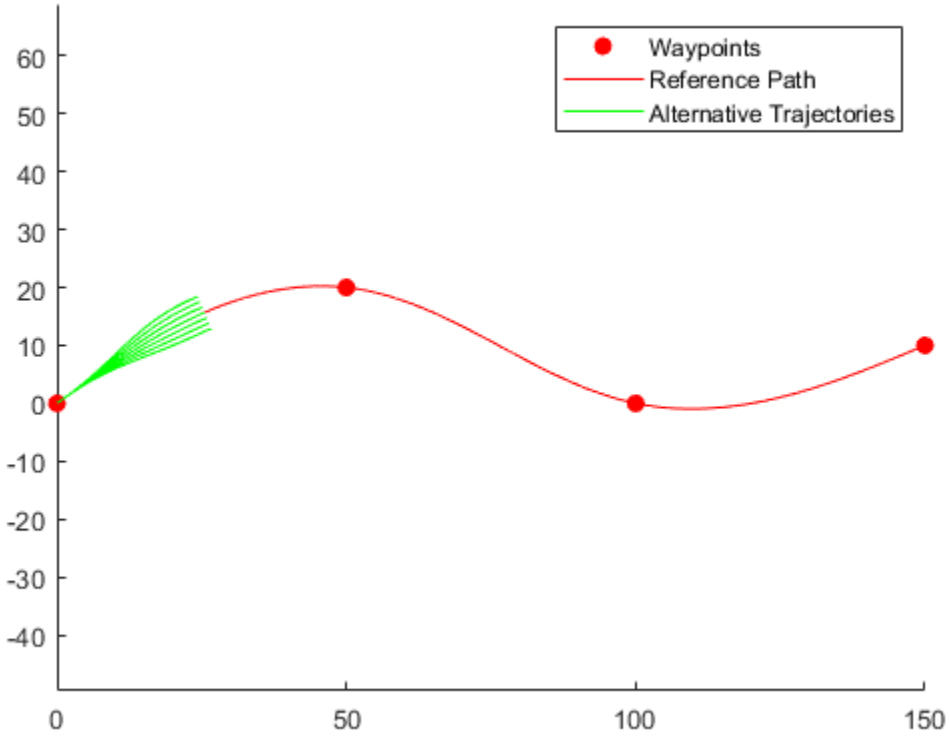
Display the trajectory in global coordinates.

```
show(refPath);
hold on
axis equal
plot(trajGlobal.Trajectory(:,1),trajGlobal.Trajectory(:,2),'b')
legend(["Waypoints","Reference Path","Trajectory to 30m"])
```



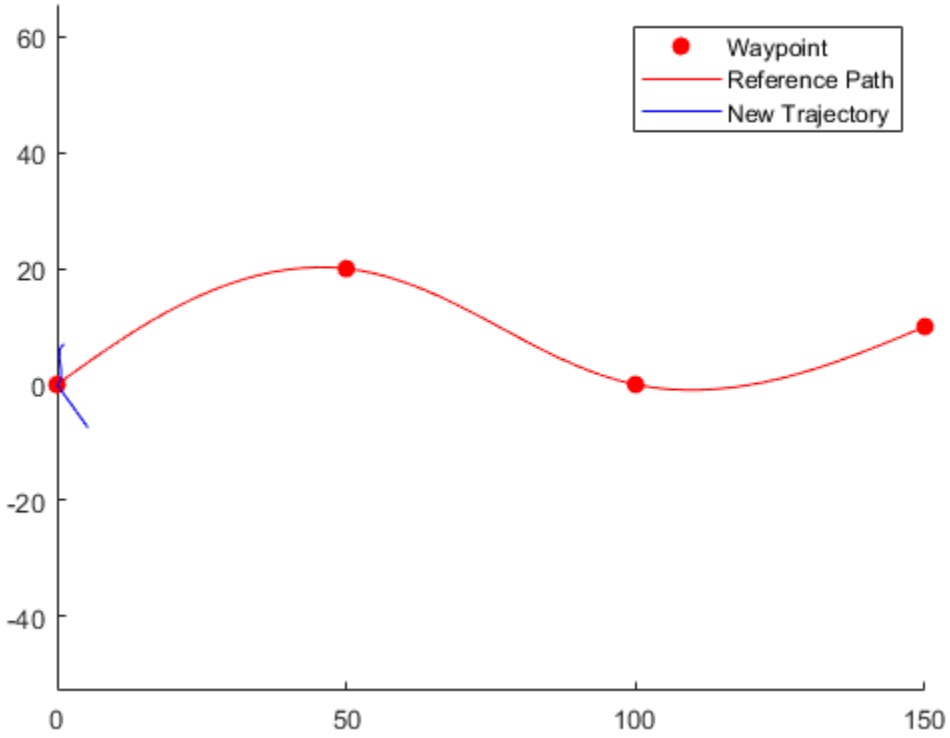
Create a matrix of terminal states with lateral deviations between –3 m and 3 m. Generate trajectories that cover the same arc length in 10 seconds, but deviate laterally from the reference path. Display the new alternative paths.

```
termStateDeviated = termState + ([-3:3]' * [0 0 0 1 0 0]);
[~,trajGlobal] = connect(connector,initState,termStateDeviated,10);

clf
show(refPath);
hold on
axis equal
for i = 1:length(trajGlobal)
    plot(trajGlobal(i).Trajectory(:,1),trajGlobal(i).Trajectory(:,2),'g')
end
legend(["Waypoints","Reference Path","Alternative Trajectories"])
hold off
```



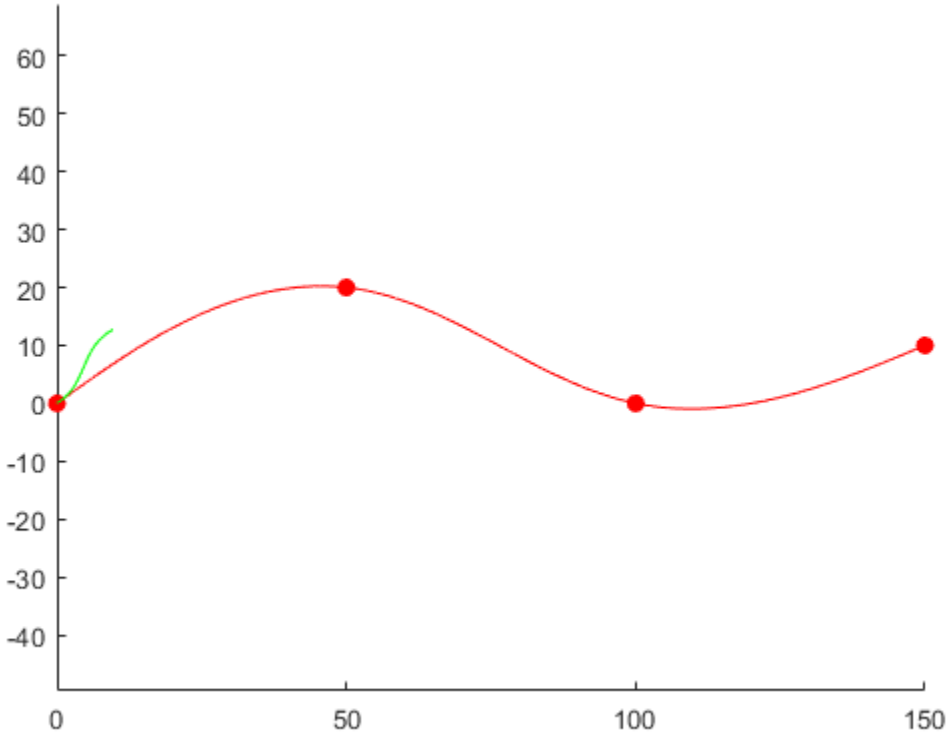
Specify a new terminal state to generate a new trajectory. This trajectory is not desirable because it requires reverse motion to achieve a lateral velocity of 10 m/s.

```
newTermState = [5 10 0 5 0 0];
[~,newTrajGlobal] = connect(connector,initState,newTermState,3);

clf
show(refPath);
hold on
axis equal
plot(newTrajGlobal.Trajectory(:,1),newTrajGlobal.Trajectory(:,2),'b');
legend(["Waypoint","Reference Path","New Trajectory"])
hold off
```



Relax the restriction on the longitudinal state by specifying an arc length of `NaN`. Generate and display the trajectory again. The new position shows a good alternative trajectory that deviates off the reference path.

```
relaxedTermState = [NaN 10 0 5 0 0];
[~,trajGlobalRelaxed] = connect(connector,initState,relaxedTermState,3);

clf
show(refPath);
hold on
axis equal
plot(trajGlobalRelaxed.Trajectory(:,1),trajGlobalRelaxed.Trajectory(:,2),'g');
hold off
```



Input Arguments

`refPath` — Reference path
`referencePathFrenet` object

Reference path, specified as a `referencePathFrenet` object.

Output Arguments

`ax` — Axes on which the reference path is plotted
`Axes` handle

Axes on which the reference path is plotted, returned as an `Axes` handle.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

Objects
`referencePathFrenet` | `trajectoryGeneratorFrenet` | `navPath`

Functions
`closestPoint` | `frenet2global` | `global2frenet` | `interpolate`

Topics
“Highway Trajectory Planning Using Frenet Reference Path”

Introduced in R2020b

resamplingPolicyPF

Create resampling policy object with resampling settings

Description

The `resamplingPolicyPF` object stores settings for when resampling should occur when using a particle filter for state estimation. The object contains the method that triggers resampling and the relevant threshold for this resampling. Use this object as the `ResamplingPolicy` property of the `stateEstimatorPF` object.

Creation

Syntax

```
policy = resamplingPolicyPF
```

Description

`policy = resamplingPolicyPF` creates a `resamplingPolicyPF` object `policy`, which contains properties to be modified to control when resampling should be triggered. Use this object as the `ResamplingPolicy` property of the `stateEstimatorPF` object.

Properties

`TriggerMethod` — Method for determining if resampling should occur
`'ratio'` (default) | character vector

Method for determining if resampling should occur, specified as a character vector. Possible choices are `'ratio'` and `'interval'`. The `'interval'` method triggers resampling at regular intervals of operating the particle filter. The `'ratio'` method triggers resampling based on the ratio of effective total particles.

`SamplingInterval` — Fixed interval between resampling
1 (default) | scalar

Fixed interval between resampling, specified as a scalar. This interval determines during which correction steps the resampling is executed. For example, a value of 2 means the resampling is executed every second correction step. A value of `inf` means that resampling is never executed.

This property only applies with the `TriggerMethod` is set to `'interval'`.

`MinEffectiveParticleRatio` — Minimum desired ratio of effective to total particles
0.5 (default) | scalar

Minimum desired ratio of effective to total particles, specified as a scalar. The effective number of particles is a measure of how well the current set of particles approximates the posterior distribution. A lower effective particle ratio means less particles are contributing to the estimation and resampling

might be required. If the ratio of effective particles to total particles falls below the `MinEffectiveParticleRatio`, a resampling step is triggered.

See Also

`stateEstimatorPF` | `correct`

Introduced in R2019b

SkyPlotChart Properties

Sky plot chart appearance and behavior

Description

The `SkyPlotChart` properties control the appearance of a sky plot chart generated using the `skyplot` function. To modify the chart appearance, use dot notation on the `SkyPlotChart` object:

```
h = skyplot;
h.AzimuthData = [45 120 295];
h.ElevationData = [10 45 60];
h.Labels = ["G1" "G4" "G11"];
```

Properties

Sky Plot Properties

`AzimuthData` — Azimuth angles for visible satellite positions

n-element vector of angles

Azimuth angles for visible satellite positions, specified as an *n*-element vector of angles. *n* is the number of visible satellite positions in the plot. Angles are measured in degrees, clockwise-positive from the North direction.

Example: `[25 45 182 356]`

Data Types: `double`

`ElevationData` — Elevation angles for visible satellite positions

n-element vector of angles

Elevation angles for visible satellite positions, specified as an *n*-element vector of angles. *n* is the number of visible satellite positions in the plot. Angles are measured from the horizon line with 90 degrees being directly up.

Example: `[45 90 27 74]`

Data Types: `double`

`LabelData` — Labels for visible satellite positions

n-element string array

Labels for visible satellite positions, specified as an *n*-element string array. *n* is the number of visible satellite positions in the plot.

Example: `["G1" "G11" "G7" "G3"]`

Data Types: `string`

`GroupData` — Group for each satellite position

`categorical` array

Group for each satellite position, specified as a `categorical` array. Each group has a different color label defined by the ColorOrder property.

Example: `[GPS GPS Galileo Galileo]`

Data Types: `double`

`ColorOrder` — Color order

seven predefined colors (default) | three-column matrix of RGB triplets

Color order, specified as a three-column matrix of RGB triplets. This property defines the palette of colors MATLAB uses to create plot objects such as `Line`, `Scatter`, and `Bar` objects. Each row of the array is an RGB triplet. An RGB triplet is a three-element vector whose elements specify the intensities of the red, green, and blue components of a color. The intensities must be in the range [0, 1]. This table lists the default colors.

Colors	ColorOrder Matrix

	`[ 0 0.4470 0.7410`
	`0.8500 0.3250 0.0980`
	`0.9290 0.6940 0.1250`
	`0.4940 0.1840 0.5560`
	`0.4660 0.6740 0.1880`
	`0.3010 0.7450 0.9330`
	`0.6350 0.0780 0.1840]`

MATLAB assigns colors to objects according to their order of creation. For example, when plotting lines, the first line uses the first color, the second line uses the second color, and so on. If there are more lines than colors, then the cycle repeats.

You can also set the color order using the `colororder` function.

Label Properties

`LabelFontSize` — Font size of labels

scalar numeric value

Font size of labels, specified as a scalar numeric value. The default font depends on the specific operating system and locale.

Example: `h = skyplot(__,'LabelFontSize',12)`

Example: `h.LabelFontSize = 12`

`LabelFontSizeMode` — Selection mode for font size of labels

`'auto'` (default) | `'manual'`

Selection mode for the font size of labels, specified as one of these values:

- `'auto'` — Font size specified by MATLAB. If you resize the axes to be smaller than the default size, the font size can scale down to improve readability and layout.
- `'manual'` — Font size specified manually. MATLAB does not scale the font size as the axes size changes. To specify the font size, set the `LabelFontSize` property.

Chart Properties

`HandleVisibility` — Visibility of object handle

`'on'` (default) | `'off'` | `'callback'`

Visibility of the `SkyPlotChart` object handle in the `Children` property of the parent, specified as one of these values:

- `'on'` — Object handle is always visible.
- `'off'` — Object handle is invisible at all times. This option is useful for preventing unintended changes to the UI by another function. To temporarily hide the handle during the execution of that function, set the `HandleVisibility` to `'off'`.
- `'callback'` — Object handle is visible from within callbacks or functions invoked by callbacks, but not from within functions invoked from the command line. This option blocks access to the object at the command line, but allows callback functions to access it.

If the object is not listed in the `Children` property of the parent, then functions that obtain object handles by searching the object hierarchy or querying handle properties cannot return it. This includes `get`, `findobj`, `gca`, `gcf`, `gco`, `newplot`, `cla`, `clf`, and `close`.

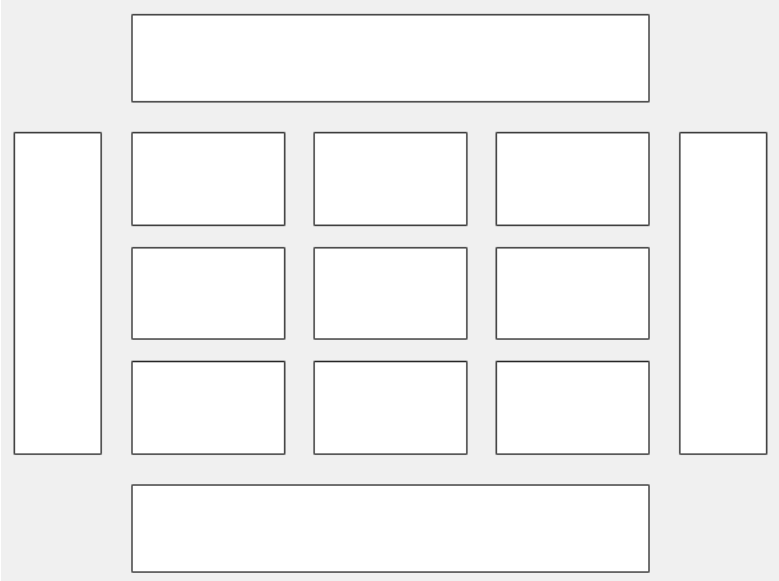
Hidden object handles are still valid. Set the root `ShowHiddenHandles` property to `'on'` to list all object handles, regardless of their `HandleVisibility` property setting.

`Layout` — Layout options

empty `LayoutOptions` array (default) | `TiledChartLayoutOptions` object | `GridLayoutOptions` object

Layout options, specified as a `TiledChartLayoutOptions` or `GridLayoutOptions` object. This property is useful when the chart is either in a tiled chart layout or a grid layout.

To position the chart within the grid of a tiled chart layout, set the `Tile` and `TileSpan` properties on the `TiledChartLayoutOptions` object. For example, consider a 3-by-3 tiled chart layout. The layout has a grid of tiles in the center, and four tiles along the outer edges. In practice, the grid is invisible and the outer tiles do not take up space until you populate them with axes or charts.



This code places the chart `c` in the third tile of the grid..

```
c.Layout.Tile = 3;
```

To make the chart span multiple tiles, specify the `TileSpan` property as a two-element vector. For example, this chart spans 2 rows and 3 columns of tiles.

```
c.Layout.TileSpan = [2 3];
```

To place the chart in one of the surrounding tiles, specify the `Tile` property as `'north'`, `'south'`, `'east'`, or `'west'`. For example, setting the value to `'east'` places the chart in the tile to the right of the grid.

```
c.Layout.Tile = 'east';
```

To place the chart into a layout within an app, specify this property as a `GridLayoutOptions` object. For more information about working with grid layouts in apps, see `uigridlayout`.

If the chart is not a child of either a tiled chart layout or a grid layout (for example, if it is a child of a figure or panel) then this property is empty and has no effect.

`Parent` — Parent container

`Figure` object | `Panel` object | `Tab` object | `TiledChartLayout` object | `GridLayout` object

Parent container, specified as a `Figure`, `Panel`, `Tab`, `TiledChartLayout`, or `GridLayout` object.

Marker Properties

`MarkerEdgeAlpha` — Marker edge transparency

1 (default) | scalar in range `[0,1]` | `'flat'`

Marker edge transparency, specified as a scalar in the range `[0,1]` or `'flat'`. A value of 1 is opaque and 0 is completely transparent. Values between 0 and 1 are semitransparent.

To set the edge transparency to a different value for each point in the plot, set the `AlphaData` property to a vector the same size as the `XData` property, and set the `MarkerEdgeAlpha` property to `'flat'`.

`MarkerEdgeColor` — Marker outline color

`'flat'` (default) | `'auto'` | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Marker outline color, specified as `'auto'`, an RGB triplet, a hexadecimal color code, a color name, or a short name. The value of `'auto'` uses the same color as the `Color` property.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`. For example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (`#`) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and the hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
'red'	'r'	[1 0 0]	'#FF0000'	
'green'	'g'	[0 1 0]	'#00FF00'	
'blue'	'b'	[0 0 1]	'#0000FF'	
'cyan'	'c'	[0 1 1]	'#00FFFF'	
'magenta'	'm'	[1 0 1]	'#FF00FF'	
'yellow'	'y'	[1 1 0]	'#FFFF00'	
'black'	'k'	[0 0 0]	'#000000'	
'white'	'w'	[1 1 1]	'#FFFFFF'	
'none'	Not applicable	Not applicable	Not applicable	No color

This table shows the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
[0 0.4470 0.7410]	'#0072BD'	
[0.8500 0.3250 0.0980]	'#D95319'	
[0.9290 0.6940 0.1250]	'#EDB120'	
[0.4940 0.1840 0.5560]	'#7E2F8E'	
[0.4660 0.6740 0.1880]	'#77AC30'	
[0.3010 0.7450 0.9330]	'#4DBEEE'	
[0.6350 0.0780 0.1840]	'#A2142F'	

MarkerFaceAlpha — Marker face transparency

0.6 (default) | scalar in range [0,1] | 'flat'

Marker face transparency, specified as a scalar in the range [0,1] or 'flat'. A value of 1 is opaque and 0 is completely transparent. Values between 0 and 1 are partially transparent.

To set the marker face transparency to a different value for each point, set the AlphaData property to a vector the same size as the XData property, and set the MarkerFaceAlpha property to 'flat'.

MarkerFaceColor — Marker fill color

'flat' (default) | 'auto' | 'none' | RGB triplet | hexadecimal color code | 'r' | 'g' | 'b' | ...

Marker fill color, specified as 'flat', 'auto', an RGB triplet, a hexadecimal color code, a color name, or a short name. The 'flat' option uses the CData values. The 'auto' option uses the same color as the Color property for the axes.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range [0,1]; for example, [0.4 0.6 0.7].

- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (#) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	
`'yellow'`	`'y'`	`[1 1 0]`	`'#FFFF00'`	
`'black'`	`'k'`	`[0 0 0]`	`'#000000'`	
`'white'`	`'w'`	`[1 1 1]`	`'#FFFFFF'`	
`'none'`	Not applicable	Not applicable	Not applicable	No color

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
`[0 0.4470 0.7410]`	`'#0072BD'`	
`[0.8500 0.3250 0.0980]`	`'#D95319'`	
`[0.9290 0.6940 0.1250]`	`'#EDB120'`	
`[0.4940 0.1840 0.5560]`	`'#7E2F8E'`	
`[0.4660 0.6740 0.1880]`	`'#77AC30'`	
`[0.3010 0.7450 0.9330]`	`'#4DBEEE'`	
`[0.6350 0.0780 0.1840]`	`'#A2142F'`	

Example: `[0.3 0.2 0.1]`

Example: `'green'`

Example: `'#D2F9A7'`

`MarkerSizeData` — Marker size

`100` (default) | positive scalar | vector of positive values

Marker size, specified as a positive scalar or vector of positive values in points, where one point = 1/72 of an inch. If specified as a vector, the vector must be of the same length as `AzimuthData`.

Position

`PositionConstraint` — Position to hold constant

`'outerposition'` | `'innerposition'`

Position property to hold constant when adding, removing, or changing decorations, specified as one of the following values:

- `'outerposition'` — The `OuterPosition` property remains constant when you add, remove, or change decorations such as a title or an axis label. If any positional adjustments are needed, MATLAB adjusts the `InnerPosition` property.
- `'innerposition'` — The `InnerPosition` property remains constant when you add, remove, or change decorations such as a title or an axis label. If any positional adjustments are needed, MATLAB adjusts the `OuterPosition` property.

Note Setting this property has no effect when the parent container is a `TiledChartLayout`.

`OuterPosition` — Outer size and location

`[0 0 1 1]` (default) | four-element vector

Outer size and location of the skyplot within the parent container (typically a figure, panel, or tab), specified as a four-element vector of the form `[left bottom width height]`. The outer position includes the colorbar, title, and axis labels.

- The `left` and `bottom` elements define the distance from the lower-left corner of the container to the lower-left corner of the skyplot.
- The `width` and `height` elements are the skyplot dimensions, which include the skyplot cells, plus a margin for the surrounding text and colorbar.

The default value of `[0 0 1 1]` covers the whole interior of the container. The units are normalized relative to the size of the container. To change the units, set the `Units` property.

Note Setting this property has no effect when the parent container is a `TiledChartLayout`.

`InnerPosition` — Inner size and location

`[0.1300 0.1100 0.7750 0.8114]` (default) | four-element vector

Inner size and location of the skyplot within the parent container (typically a figure, panel, or tab), specified as a four-element vector of the form `[left bottom width height]`. The inner position does not include the colorbar, title, or axis labels.

- The `left` and `bottom` elements define the distance from the lower-left corner of the container to the lower-left corner of the skyplot.
- The `width` and `height` elements are the skyplot dimensions, which include only the skyplot cells.

Note Setting this property has no effect when the parent container is a `TiledChartLayout`.

`Position` — Inner size and location

four-element vector

Inner size and location of the skyplot within the parent container (typically a figure, panel, or tab), specified as a four-element vector of the form `[left bottom width height]`. This property is equivalent to the `InnerPosition` property.

Note Setting this property has no effect when the parent container is a `TiledChartLayout`.

`Units` — Position units

`'normalized'` (default) | `'inches'` | `'centimeters'` | `'points'` | `'pixels'` | `'characters'`

Position units, specified as one of these values.

Units	Description
`'normalized'` (default)	Normalized with respect to the container, which is typically the figure or a panel. The lower left corner of the container maps to `(0,0)`, and the upper right corner maps to `(1,1)`.
`'inches'`	Inches.
`'centimeters'`	Centimeters.
`'characters'`	Based on the default `uicontrol` font of the graphics root object: • Character width = width of letter `x`. • Character height = distance between the baselines of two lines of text.
`'points'`	Typography points. One point equals 1/72 inch.
`'pixels'`	Pixels. Starting in R2015b, distances in pixels are independent of your system resolution on Windows® and Macintosh systems: • On Windows systems, a pixel is 1/96th of an inch. • On Macintosh systems, a pixel is 1/72nd of an inch. On Linux® systems, the size of a pixel is determined by your system resolution.

When specifying the units as a name-value argument during object creation, you must set the `Units` property before specifying the properties that you want to use these units, such as `OuterPosition`.

`Visible` — State of visibility

`'on'` (default) | on/off logical value

State of visibility, specified as `'on'` or `'off'`, or as numeric or logical `1` (`true`) or `0` (`false`). A value of `'on'` is equivalent to `true`, and `'off'` is equivalent to `false`. Thus, you can use the value of this property as a logical value. The value is stored as an on/off logical value of type `matlab.lang.OnOffSwitchState`.

- `'on'` — Display the skyplot.
- `'off'` — Hide the skyplot without deleting it. You can still access the properties of an invisible `SkyPlotChart` object.

See Also

Functions
`skyplot` | `polarscatter`

Objects
`gnssSensor` | `nmeaParser`

Introduced in R2021a

stateEstimatorPF

Create particle filter state estimator

Description

The `stateEstimatorPF` object 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 and correction. The prediction step uses the previous state to predict the current state based on a given system model. The correction step uses the current sensor measurement to correct the state estimate. The algorithm periodically redistributes, or resamples, the particles in the state space to match the posterior distribution of the estimated state.

The estimated state consists of state variables. Each particle represents a discrete state hypothesis of these state variables. 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.

For more information on the particle filter workflow and setting specific parameters, see:

- “Particle Filter Workflow”
- “Particle Filter Parameters”

Creation

Syntax

```
pf = stateEstimatorPF
```

Description

`pf = stateEstimatorPF` creates an object that enables the state estimation for a simple system with three state variables. Use the `initialize` method to initialize the particles with a known mean and covariance or uniformly distributed particles within defined bounds. To customize the particle filter’s system and measurement models, modify the `StateTransitionFcn` and `MeasurementLikelihoodFcn` properties.

After you create the object, use `initialize` to initialize the `NumStateVariables` and `NumParticles` properties. The `initialize` function sets these two properties based on your inputs.

Properties

`NumStateVariables` — Number of state variables
3 (default) | scalar

This property is read-only.

Number of state variables, specified as a scalar. This property is set based on the inputs to the `initialize` method. The number of states is implicit based on the specified matrices for initial state and covariance.

`NumParticles` — Number of particles used in the filter

1000 (default) | scalar

This property is read-only.

Number of particles using in the filter, specified as a scalar. You can specify this property only by calling the `initialize` method.

`StateTransitionFcn` — Callback function for determining the state transition between particle filter steps

function handle

Callback function for determining the state transition between particle filter steps, specified as a function handle. The state transition function evolves the system state for each particle. The function signature is:

```
function predictParticles = stateTransitionFcn(pf,prevParticles,varargin)
```

The callback function accepts at least two input arguments: the `stateEstimatorPF` object, `pf`, and the particles at the previous time step, `prevParticles`. These specified particles are the `predictParticles` returned from the previous call of the object. `predictParticles` and `prevParticles` are the same size: `NumParticles`-by-`NumStateVariables`.

You can also use `varargin` to pass in a variable number of arguments from the `predict` function. When you call:

```
predict(pf,arg1,arg2)
```

MATLAB essentially calls `stateTranstionFcn` as:

```
stateTransitionFcn(pf,prevParticles,arg1,arg2)
```

`MeasurementLikelihoodFcn` — Callback function calculating the likelihood of sensor measurements

function handle

Callback function calculating the likelihood of sensor measurements, specified as a function handle. Once a sensor measurement is available, this callback function calculates the likelihood that the measurement is consistent with the state hypothesis of each particle. You must implement this function based on your measurement model. The function signature is:

```
function likelihood = measurementLikelihoodFcn(PF,predictParticles,measurement,varargin)
```

The callback function accepts at least three input arguments:

1. `pf` – The associated `stateEstimatorPF` object
2. `predictParticles` – The particles that represent the predicted system state at the current time step as an array of size `NumParticles`-by-`NumStateVariables`
3. `measurement` – The state measurement at the current time step

You can also use `varargin` to pass in a variable number of arguments. These arguments are passed by the `correct` function. When you call:

```
correct(pf,measurement,arg1,arg2)
```

MATLAB essentially calls `measurementLikelihoodFcn` as:

```
measurementLikelihoodFcn(pf,predictParticles,measurement,arg1,arg2)
```

The callback needs to return exactly one output, `likelihood`, which is the likelihood of the given `measurement` for each particle state hypothesis.

`IsStateVariableCircular` — Indicator if state variables have a circular distribution
[0 0 0] (default) | logical array

Indicator if state variables have a circular distribution, specified as a logical array. Circular (or angular) distributions use a probability density function with a range of `[-pi,pi]`. If the object has multiple state variables, then `IsStateVariableCircular` is a row vector. Each vector element indicates if the associated state variable is circular. If the object has only one state variable, then `IsStateVariableCircular` is a scalar.

`ResamplingPolicy` — Policy settings that determine when to trigger resampling
object

Policy settings that determine when to trigger resampling, specified as an object. You can trigger resampling either at fixed intervals, or you can trigger it dynamically, based on the number of effective particles. See `resamplingPolicyPF` for more information.

`ResamplingMethod` — Method used for particle resampling
`'multinomial'` (default) | `'residual'` | `'stratified'` | `'systematic'`

Method used for particle resampling, specified as `'multinomial'`, `'residual'`, `'stratified'`, and `'systematic'`.

`StateEstimationMethod` — Method used for state estimation
`'mean'` (default) | `'maxweight'`

Method used for state estimation, specified as `'mean'` and `'maxweight'`.

`Particles` — Array of particle values
`NumParticles`-by-`NumStateVariables` matrix

Array of particle values, specified as a `NumParticles`-by-`NumStateVariables` matrix. Each row corresponds to the state hypothesis of a single particle.

`Weights` — Particle weights
`NumParticles`-by-1 vector

Particle weights, specified as a `NumParticles`-by-1 vector. Each weight is associated with the particle in the same row in the `Particles` property.

`State` — Best state estimate
vector

This property is read-only.

Best state estimate, returned as a vector with length `NumStateVariables`. The estimate is extracted based on the `StateEstimationMethod` property.

`State Covariance` — Corrected system covariance
N-by-*N* matrix | `[]`

This property is read-only.

Corrected system variance, returned as an *N*-by-*N* matrix, where *N* is equal to the `NumStateVariables` property. The corrected state is calculated based on the `StateEstimationMethod` property and the `MeasurementLikelihoodFcn`. If you specify a state estimate method that does not support covariance, then the property is set to `[]`.

Object Functions

initialize	Initialize the state of the particle filter
getStateEstimate	Extract best state estimate and covariance from particles
predict	Predict state of robot in next time step
correct	Adjust state estimate based on sensor measurement

Examples

Particle Filter Prediction and Correction

Create a `stateEstimatorPF` object, and execute a prediction and correction step for state estimation. The particle filter gives a predicted state estimate based on the return value of `StateTransitionFcn`. It then corrects the state based on a given measurement and the return value of `MeasurementLikelihoodFcn`.

Create a particle filter with the default three states.

```
pf = stateEstimatorPF
```

```
pf =
  stateEstimatorPF with properties:

          NumStateVariables: 3
               NumParticles: 1000
         StateTransitionFcn: @nav.algs.gaussianMotion
    MeasurementLikelihoodFcn: @nav.algs.fullStateMeasurement
     IsStateVariableCircular: [0 0 0]
            ResamplingPolicy: [1x1 resamplingPolicyPF]
            ResamplingMethod: 'multinomial'
       StateEstimationMethod: 'mean'
            StateOrientation: 'row'
                   Particles: [1000x3 double]
                     Weights: [1000x1 double]
                       State: 'Use the getStateEstimate function to see the value.'
             StateCovariance: 'Use the getStateEstimate function to see the value.'
```

Specify the mean state estimation method and systematic resampling method.

```
pf.StateEstimationMethod = 'mean';
pf.ResamplingMethod = 'systematic';
```

Initialize the particle filter at state [4 1 9] with unit covariance (`eye(3)`). Use 5000 particles.

```
initialize(pf,5000,[4 1 9],eye(3));
```

Assuming a measurement [4.2 0.9 9], run one predict and one correct step.

```
[statePredicted,stateCov] = predict(pf);
[stateCorrected,stateCov] = correct(pf,[4.2 0.9 9]);
```

Get the best state estimate based on the `StateEstimationMethod` algorithm.

```
stateEst = getStateEstimate(pf)
```

```
stateEst = 1×3

    4.1562    0.9185    9.0202
```

Estimate Robot Position in a Loop Using Particle Filter

Use the `stateEstimatorPF` object to track a robot as it moves in a 2-D space. The measured position has random noise added. Using `predict` and `correct`, track the robot based on the measurement and on an assumed motion model.

Initialize the particle filter and specify the default state transition function, the measurement likelihood function, and the resampling policy.

```
pf = stateEstimatorPF;
pf.StateEstimationMethod = 'mean';
pf.ResamplingMethod = 'systematic';
```

Sample 1000 particles with an initial position of [0 0] and unit covariance.

```
initialize(pf,1000,[0 0],eye(2));
```

Prior to estimation, define a sine wave path for the dot to follow. Create an array to store the predicted and estimated position. Define the amplitude of noise.

```
t = 0:0.1:4*pi;
dot = [t; sin(t)]';
robotPred = zeros(length(t),2);
robotCorrected = zeros(length(t),2);
noise = 0.1;
```

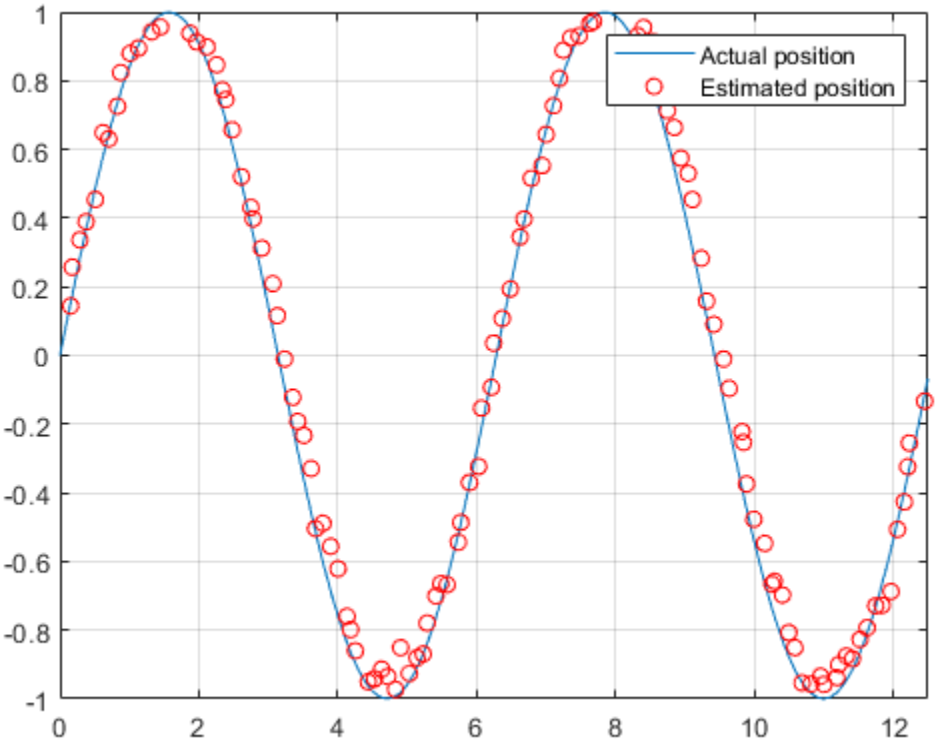
Begin the loop for predicting and correcting the estimated position based on measurements. The resampling of particles occurs based on the`ResamplingPolicy` property. The robot moves based on a sine wave function with random noise added to the measurement.

```
for i = 1:length(t)
    % Predict next position. Resample particles if necessary.
    [robotPred(i,:),robotCov] = predict(pf);
    % Generate dot measurement with random noise. This is
    % equivalent to the observation step.
    measurement(i,:) = dot(i,:) + noise*(rand([1 2])-noise/2);
    % Correct position based on the given measurement to get best estimation.
```

```
    % Actual dot position is not used. Store corrected position in data array.
    [robotCorrected(i,:),robotCov] = correct(pf,measurement(i,:));
end
```

Plot the actual path versus the estimated position. Actual results may vary due to the randomness of particle distributions.

```
plot(dot(:,1),dot(:,2),robotCorrected(:,1),robotCorrected(:,2),'or')
xlim([0 t(end)])
ylim([-1 1])
legend('Actual position','Estimated position')
grid on
```



The figure shows how close the estimate state matches the actual position of the robot. Try tuning the number of particles or specifying a different initial position and covariance to see how it affects tracking over time.

Compatibility Considerations

`stateEstimatorPF` was renamed
Behavior change in future release

The `stateEstimatorPF` object was renamed from `robotics.ParticleFilter`. Use `stateEstimatorPF` for all object creation.

References

[1] Arulampalam, M.S., S. Maskell, N. Gordon, and T. Clapp. "A Tutorial on Particle Filters for Online Nonlinear/Non-Gaussian Bayesian Tracking." *IEEE Transactions on Signal Processing*. Vol. 50, No. 2, Feb 2002, pp. 174-188.

[2] Chen, Z. "Bayesian Filtering: From Kalman Filters to Particle Filters, and Beyond." *Statistics*. Vol. 182, No. 1, 2003, pp. 1-69.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`resamplingPolicyPF` | `initialize` | `getStateEstimate` | `predict` | `correct`

Topics

“Track a Car-Like Robot Using Particle Filter” (Robotics System Toolbox)
“Particle Filter Parameters”
“Particle Filter Workflow”

Introduced in R2016a

copy

Create copy of particle filter

Syntax

```
b = copy(a)
```

Description

`b = copy(a)` copies each element in the array of handles, `a`, to the new array of handles, `b`.

The `copy` method does not copy dependent properties. MATLAB does not call `copy` recursively on any handles contained in property values. MATLAB also does not call the class constructor or property-set methods during the copy operation.

Input Arguments

a — Object array

handle

Object array, specified as a handle.

Output Arguments

b — Object array containing copies of the objects in a

handle

Object array containing copies of the object in `a`, specified as a handle.

`b` has the same number of elements and is the same size and class of `a`. `b` is the same class as `a`. If `a` is empty, `b` is also empty. If `a` is heterogeneous, `b` is also heterogeneous. If `a` contains deleted handles, then `copy` creates deleted handles of the same class in `b`. Dynamic properties and listeners associated with objects in `a` are not copied to objects in `b`.

See Also

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

Topics

“Track a Car-Like Robot Using Particle Filter” (Robotics System Toolbox)
“Particle Filter Parameters”
“Particle Filter Workflow”

Introduced in R2016a

correct

Adjust state estimate based on sensor measurement

Syntax

```
[stateCorr,stateCov] = correct(pf,measurement)
[stateCorr,stateCov] = correct(pf,measurement,varargin)
```

Description

`[stateCorr,stateCov] = correct(pf,measurement)` calculates the corrected system state and its associated uncertainty covariance based on a sensor `measurement` at the current time step. `correct` uses the `MeasurementLikelihoodFcn` property from the particle filter object, `pf`, as a function to calculate the likelihood of the sensor measurement for each particle. The two inputs to the `MeasurementLikelihoodFcn` function are:

1. `pf` – The `stateEstimatorPF` object, which contains the particles of the current iteration
2. `measurement` – The sensor measurements used to correct the state estimate

The `MeasurementLikelihoodFcn` function then extracts the best state estimate and covariance based on the setting in the `StateEstimationMethod` property.

`[stateCorr,stateCov] = correct(pf,measurement,varargin)` passes all additional arguments in `varargin` to the underlying `MeasurementLikelihoodFcn` after the first three required inputs.

Examples

Particle Filter Prediction and Correction

Create a `stateEstimatorPF` object, and execute a prediction and correction step for state estimation. The particle filter gives a predicted state estimate based on the return value of `StateTransitionFcn`. It then corrects the state based on a given measurement and the return value of `MeasurementLikelihoodFcn`.

Create a particle filter with the default three states.

```
pf = stateEstimatorPF
```

```
pf =
  stateEstimatorPF with properties:

          NumStateVariables: 3
               NumParticles: 1000
         StateTransitionFcn: @nav.algs.gaussianMotion
    MeasurementLikelihoodFcn: @nav.algs.fullStateMeasurement
     IsStateVariableCircular: [0 0 0]
            ResamplingPolicy: [1x1 resamplingPolicyPF]
            ResamplingMethod: 'multinomial'
       StateEstimationMethod: 'mean'
```

```
    StateOrientation: 'row'
           Particles: [1000x3 double]
             Weights: [1000x1 double]
               State: 'Use the getStateEstimate function to see the value.'
     StateCovariance: 'Use the getStateEstimate function to see the value.'
```

Specify the mean state estimation method and systematic resampling method.

```
pf.StateEstimationMethod = 'mean';
pf.ResamplingMethod = 'systematic';
```

Initialize the particle filter at state [4 1 9] with unit covariance (`eye(3)`). Use 5000 particles.

```
initialize(pf,5000,[4 1 9],eye(3));
```

Assuming a measurement [4.2 0.9 9], run one predict and one correct step.

```
[statePredicted,stateCov] = predict(pf);
[stateCorrected,stateCov] = correct(pf,[4.2 0.9 9]);
```

Get the best state estimate based on the `StateEstimationMethod` algorithm.

```
stateEst = getStateEstimate(pf)
```

```
stateEst = 1×3

    4.1562    0.9185    9.0202
```

Input Arguments

`pf` — `stateEstimatorPF` object

handle

`stateEstimatorPF` object, specified as a handle. See `stateEstimatorPF` for more information.

`measurement` — Sensor measurements

array

Sensor measurements, specified as an array. This input is passed directly into the `MeasurementLikelihoodFcn` property of `pf`. It is used to calculate the likelihood of the sensor measurement for each particle.

`varargin` — Variable-length input argument list

comma-separated list

Variable-length input argument list, specified as a comma-separated list. This input is passed directly into the `MeasurementLikelihoodFcn` property of `pf`. It is used to calculate the likelihood of the sensor measurement for each particle. When you call:

```
correct(pf,measurement,arg1,arg2)
```

MATLAB essentially calls `measurementLikelihoodFcn` as:

```
measurementLikelihoodFcn(pf,measurement,arg1,arg2)
```

Output Arguments

`stateCorr` — Corrected system state

vector with length `NumStateVariables`

Corrected system state, returned as a row vector with length `NumStateVariables`. The corrected state is calculated based on the `StateEstimationMethod` algorithm and the `MeasurementLikelihoodFcn`.

`stateCov` — Corrected system covariance

N-by-*N* matrix | `[]`

Corrected system variance, returned as an *N*-by-*N* matrix, where *N* is the value of `NumStateVariables` property from `pf`. The corrected state is calculated based on the `StateEstimationMethod` algorithm and the `MeasurementLikelihoodFcn`. If you specify a state estimate method that does not support covariance, then the function returns `stateCov` as `[]`.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

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

Topics

“Track a Car-Like Robot Using Particle Filter” (Robotics System Toolbox)
“Particle Filter Parameters”
“Particle Filter Workflow”

Introduced in R2016a

getStateEstimate

Extract best state estimate and covariance from particles

Syntax

```
stateEst = getStateEstimate(pf)
[stateEst,stateCov] = getStateEstimate(pf)
```

Description

`stateEst = getStateEstimate(pf)` returns the best state estimate based on the current set of particles. The estimate is extracted based on the `StateEstimationMethod` property from the `stateEstimatorPF` object, `pf`.

`[stateEst,stateCov] = getStateEstimate(pf)` also returns the covariance around the state estimate. The covariance is a measure of the uncertainty of the state estimate. Not all state estimate methods support covariance output. In this case, `getStateEstimate` returns `stateCov` as `[]`.

Examples

Particle Filter Prediction and Correction

Create a `stateEstimatorPF` object, and execute a prediction and correction step for state estimation. The particle filter gives a predicted state estimate based on the return value of `StateTransitionFcn`. It then corrects the state based on a given measurement and the return value of `MeasurementLikelihoodFcn`.

Create a particle filter with the default three states.

```
pf = stateEstimatorPF
```

```
pf =
  stateEstimatorPF with properties:

          NumStateVariables: 3
               NumParticles: 1000
         StateTransitionFcn: @nav.algs.gaussianMotion
    MeasurementLikelihoodFcn: @nav.algs.fullStateMeasurement
     IsStateVariableCircular: [0 0 0]
            ResamplingPolicy: [1x1 resamplingPolicyPF]
            ResamplingMethod: 'multinomial'
       StateEstimationMethod: 'mean'
            StateOrientation: 'row'
                   Particles: [1000x3 double]
                     Weights: [1000x1 double]
                       State: 'Use the getStateEstimate function to see the value.'
            StateCovariance: 'Use the getStateEstimate function to see the value.'
```

Specify the mean state estimation method and systematic resampling method.

```
pf.StateEstimationMethod = 'mean';
pf.ResamplingMethod = 'systematic';
```

Initialize the particle filter at state [4 1 9] with unit covariance (`eye(3)`). Use 5000 particles.

```
initialize(pf,5000,[4 1 9],eye(3));
```

Assuming a measurement [4.2 0.9 9], run one predict and one correct step.

```
[statePredicted,stateCov] = predict(pf);
[stateCorrected,stateCov] = correct(pf,[4.2 0.9 9]);
```

Get the best state estimate based on the `StateEstimationMethod` algorithm.

```
stateEst = getStateEstimate(pf)
```

```
stateEst = 1×3

    4.1562    0.9185    9.0202
```

Input Arguments

`pf` — `stateEstimatorPF` object

handle

`stateEstimatorPF` object, specified as a handle. See `stateEstimatorPF` for more information.

Output Arguments

`stateEst` — Best state estimate

vector

Best state estimate, returned as a row vector with length `NumStateVariables`. The estimate is extracted based on the `StateEstimationMethod` algorithm specified in `pf`.

`stateCov` — Corrected system covariance

N-by-*N* matrix | `[]`

Corrected system variance, returned as an *N*-by-*N* matrix, where *N* is the value of `NumStateVariables` property from `pf`. The corrected state is calculated based on the `StateEstimationMethod` algorithm and the `MeasurementLikelihoodFcn`. If you specify a state estimate method that does not support covariance, then the function returns `stateCov` as `[]`.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

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

Topics
“Track a Car-Like Robot Using Particle Filter” (Robotics System Toolbox)
“Particle Filter Parameters”
“Particle Filter Workflow”

Introduced in R2016a

initialize

Initialize the state of the particle filter

Syntax

```
initialize(pf,numParticles,mean,covariance)
initialize(pf,numParticles,stateBounds)
initialize( ___ ,Name,Value)
```

Description

`initialize(pf,numParticles,mean,covariance)` initializes the particle filter object, `pf`, with a specified number of particles, `numParticles`. The initial states of the particles in the state space are determined by sampling from the multivariate normal distribution with the specified `mean` and `covariance`.

`initialize(pf,numParticles,stateBounds)` determines the initial location of the particles by sample from the multivariate uniform distribution within the specified `stateBounds`.

`initialize( ___ ,Name,Value)` initializes the particles with additional options specified by one or more `Name,Value` pair arguments.

Examples

Particle Filter Prediction and Correction

Create a `stateEstimatorPF` object, and execute a prediction and correction step for state estimation. The particle filter gives a predicted state estimate based on the return value of `StateTransitionFcn`. It then corrects the state based on a given measurement and the return value of `MeasurementLikelihoodFcn`.

Create a particle filter with the default three states.

```
pf = stateEstimatorPF
```

```
pf =
  stateEstimatorPF with properties:

          NumStateVariables: 3
               NumParticles: 1000
         StateTransitionFcn: @nav.algs.gaussianMotion
    MeasurementLikelihoodFcn: @nav.algs.fullStateMeasurement
     IsStateVariableCircular: [0 0 0]
            ResamplingPolicy: [1x1 resamplingPolicyPF]
            ResamplingMethod: 'multinomial'
       StateEstimationMethod: 'mean'
            StateOrientation: 'row'
                   Particles: [1000x3 double]
                     Weights: [1000x1 double]
                       State: 'Use the getStateEstimate function to see the value.'
```

```
    StateCovariance: 'Use the getStateEstimate function to see the value.'
```

Specify the mean state estimation method and systematic resampling method.

```
pf.StateEstimationMethod = 'mean';
pf.ResamplingMethod = 'systematic';
```

Initialize the particle filter at state [4 1 9] with unit covariance (`eye(3)`). Use 5000 particles.

```
initialize(pf,5000,[4 1 9],eye(3));
```

Assuming a measurement [4.2 0.9 9], run one predict and one correct step.

```
[statePredicted,stateCov] = predict(pf);
[stateCorrected,stateCov] = correct(pf,[4.2 0.9 9]);
```

Get the best state estimate based on the `StateEstimationMethod` algorithm.

```
stateEst = getStateEstimate(pf)
```

```
stateEst = 1×3

    4.1562    0.9185    9.0202
```

Input Arguments

`pf` — `stateEstimatorPF` object

handle

`stateEstimatorPF` object, specified as a handle. See `stateEstimatorPF` for more information.

`numParticles` — Number of particles used in the filter

scalar

Number of particles used in the filter, specified as a scalar.

`mean` — Mean of particle distribution

vector

Mean of particle distribution, specified as a vector. The `NumStateVariables` property of `pf` is set based on the length of this vector.

`covariance` — Covariance of particle distribution

N-by-*N* matrix

Covariance of particle distribution, specified as an *N*-by-*N* matrix, where *N* is the value of `NumStateVariables` property from `pf`.

`stateBounds` — Bounds of state variables

n-by-2 matrix

Bounds of state variables, specified as an *n*-by-2 matrix. The `NumStateVariables` property of `pf` is set based on the value of *n*. Each row corresponds to the lower and upper limit of the corresponding state variable.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `"CircularVariables",[0 0 1]`

`CircularVariables` — Circular variables

logical vector

Circular variables, specified as a logical vector. Each state variable that uses circular or angular coordinates is indicated with a 1. The length of the vector is equal to the `NumStateVariables` property of `pf`.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

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

Topics

“Track a Car-Like Robot Using Particle Filter” (Robotics System Toolbox)
“Particle Filter Parameters”
“Particle Filter Workflow”

Introduced in R2016a

predict

Predict state of robot in next time step

Syntax

```
[statePred,stateCov] = predict(pf)
[statePred,stateCov] = predict(pf,varargin)
```

Description

`[statePred,stateCov] = predict(pf)` calculates the predicted system state and its associated uncertainty covariance. `predict` uses the `StateTransitionFcn` property of `stateEstimatorPF` object, `pf`, to evolve the state of all particles. It then extracts the best state estimate and covariance based on the setting in the `StateEstimationMethod` property.

`[statePred,stateCov] = predict(pf,varargin)` passes all additional arguments specified in `varargin` to the underlying `StateTransitionFcn` property of `pf`. The first input to `StateTransitionFcn` is the set of particles from the previous time step, followed by all arguments in `varargin`.

Examples

Particle Filter Prediction and Correction

Create a `stateEstimatorPF` object, and execute a prediction and correction step for state estimation. The particle filter gives a predicted state estimate based on the return value of `StateTransitionFcn`. It then corrects the state based on a given measurement and the return value of `MeasurementLikelihoodFcn`.

Create a particle filter with the default three states.

```
pf = stateEstimatorPF
```

```
pf =
  stateEstimatorPF with properties:

          NumStateVariables: 3
               NumParticles: 1000
         StateTransitionFcn: @nav.algs.gaussianMotion
    MeasurementLikelihoodFcn: @nav.algs.fullStateMeasurement
     IsStateVariableCircular: [0 0 0]
            ResamplingPolicy: [1x1 resamplingPolicyPF]
            ResamplingMethod: 'multinomial'
       StateEstimationMethod: 'mean'
            StateOrientation: 'row'
                   Particles: [1000x3 double]
                     Weights: [1000x1 double]
                       State: 'Use the getStateEstimate function to see the value.'
             StateCovariance: 'Use the getStateEstimate function to see the value.'
```

Specify the mean state estimation method and systematic resampling method.

```
pf.StateEstimationMethod = 'mean';
pf.ResamplingMethod = 'systematic';
```

Initialize the particle filter at state [4 1 9] with unit covariance (`eye(3)`). Use 5000 particles.

```
initialize(pf,5000,[4 1 9],eye(3));
```

Assuming a measurement [4.2 0.9 9], run one predict and one correct step.

```
[statePredicted,stateCov] = predict(pf);
[stateCorrected,stateCov] = correct(pf,[4.2 0.9 9]);
```

Get the best state estimate based on the `StateEstimationMethod` algorithm.

```
stateEst = getStateEstimate(pf)
```

```
stateEst = 1×3

    4.1562    0.9185    9.0202
```

Input Arguments

`pf` — `stateEstimatorPF` object

handle

`stateEstimatorPF` object, specified as a handle. See `stateEstimatorPF` for more information.

`varargin` — Variable-length input argument list

comma-separated list

Variable-length input argument list, specified as a comma-separated list. This input is passed directly into the `StateTransitionFcn` property of `pf` to evolve the system state for each particle. When you call:

```
predict(pf,arg1,arg2)
```

MATLAB essentially calls the `stateTranstionFcn` as:

```
stateTransitionFcn(pf,prevParticles,arg1,arg2)
```

Output Arguments

`statePred` — Predicted system state

vector

Predicted system state, returned as a vector with length `NumStateVariables`. The predicted state is calculated based on the `StateEstimationMethod` algorithm.

`stateCov` — Corrected system covariance

N-by-*N* matrix | [ ]

Corrected system variance, returned as an *N*-by-*N* matrix, where *N* is the value of `NumStateVariables` property from `pf`. The corrected state is calculated based on the

`StateEstimationMethod` algorithm and the `MeasurementLikelihoodFcn`. If you specify a state estimate method that does not support covariance, then the function returns `stateCov` as `[]`.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

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

Topics

“Track a Car-Like Robot Using Particle Filter” (Robotics System Toolbox)
“Particle Filter Parameters”
“Particle Filter Workflow”

Introduced in R2016a

stateSpaceSE2

SE(2) state space

Description

The `stateSpaceSE2` object stores parameters and states in the SE(2) state space, which is composed of state vectors represented by [x, y, θ]. x and y are Cartesian coordinates, and θ is the orientation angle. The object uses Euclidean distance to calculate distance and uses linear interpolation to calculate translation and rotation of the state.

Creation

Syntax

```
space = stateSpaceSE2
space = stateSpaceSE2(bounds)
```

Description

`space = stateSpaceSE2` creates an SE(2) state space object with default state bounds for x, y, and θ.

`space = stateSpaceSE2(bounds)` specifies the `bounds` for x, y, and θ. The state values beyond the bounds are truncated to the bounds. The input, `bounds`, allows you to set the value of the StateBounds property.

Properties

`Name` — Name of state space
`'SE2'` (default) | string

Name of state space, specified as a string.

`NumStateVariables` — Dimension of the state space
3 (default) | positive integer

This property is read-only.

Dimension of the state space, specified as a positive integer.

`StateBounds` — Bounds of state variables
`[-100 100; -100 100; -3.1416 3.1416]` (default) | 3-by-2 real-valued matrix

Bounds of state variables, specified as a 3-by-2 real-valued matrix.

- The first row specifies the lower and upper bounds of the x state in meters.
- The second row specifies the lower and upper bounds of the y state in meters.

- The third row specifies the lower and upper bounds of the θ state in radians.

Data Types: `double`

`WeightXY` — Weight applied to x and y distance calculation
1 (default) | nonnegative real scalar

Weight applied to x and y distance calculation, specified as a nonnegative real scalar.

In the object, the distance calculated as:

$$d = \sqrt{\left(w_{xy}(d_x^2 + d_y^2)\right) + w_\theta d_\theta^2}$$

w_{xy} is weight applied to x and y coordinates, and w_θ is the weight applied to the θ coordinate. d_x, d_y, and d_θ are the distances in the x, y, and θ direction, respectively.

Data Types: `double`

`WeightTheta` — Weight applied to theta distance calculation
0.1 (default) | nonnegative real scalar

Weight applied to θ distance calculation, specified as a nonnegative real scalar.

In the object, the distance calculated as:

$$d = \sqrt{\left(w_{xy}(d_x^2 + d_y^2)\right) + w_\theta d_\theta^2}$$

w_{xy} is weight applied to x and y coordinates, and w_θ is the weight applied to the θ coordinate. d_x, d_y, and d_θ are the distances in the x, y, and θ direction, respectively.

Data Types: `double`

Object Functions

copy	Create deep copy of state space object
distance	Distance between two states
enforceStateBounds	Reduce state to state bounds
interpolate	Interpolate between states
sampleGaussian	Sample state using Gaussian distribution
sampleUniform	Sample state using uniform distribution

Examples

Plan Path Between Two SE(2) States

Create an SE(2) state space.

```
ss = stateSpaceSE2;
```

Create an `occupancyMap`-based state validator using the created state space.

```
sv = validatorOccupancyMap(ss);
```

Create an occupancy map from an example map and set map resolution as 10 cells/meter.

```
load exampleMaps
map = occupancyMap(simpleMap,10);
sv.Map = map;
```

Set validation distance for the validator.

```
sv.ValidationDistance = 0.01;
```

Update state space bounds to be the same as map limits.

```
ss.StateBounds = [map.XWorldLimits;map.YWorldLimits; [-pi pi]];
```

Create the path planner and increase maximum connection distance.

```
planner = plannerRRT(ss,sv);
planner.MaxConnectionDistance = 0.3;
```

Set the start and goal states.

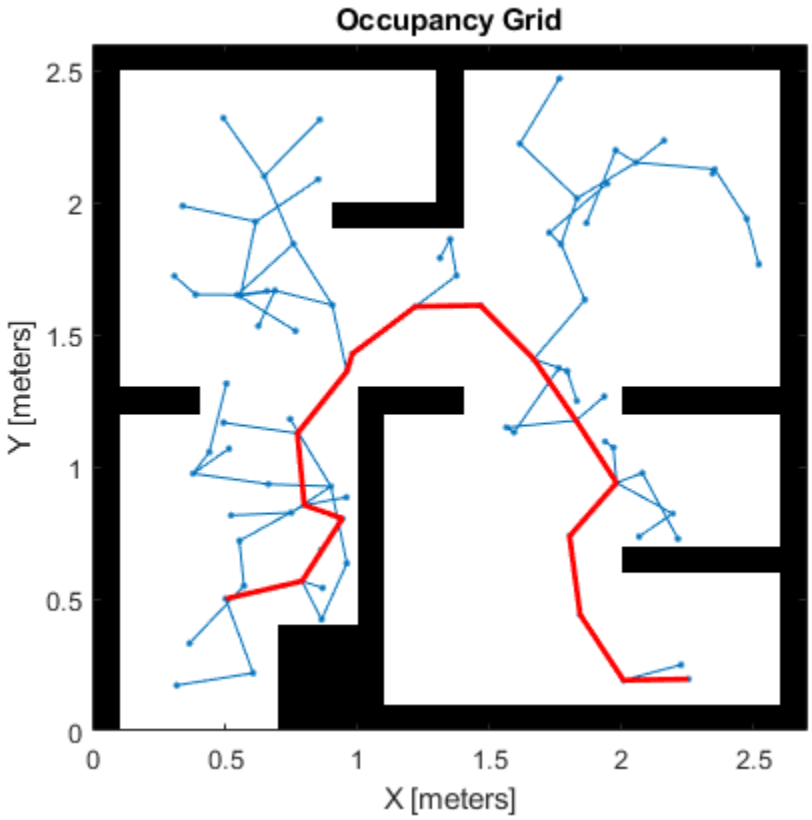
```
start = [0.5,0.5,0];
goal = [2.5,0.2,0];
```

Plan a path with default settings.

```
rng(100,'twister'); % for repeatable result
[pthObj,solnInfo] = planner.plan(start,goal);
```

Visualize the results.

```
map.show; hold on;
plot(solnInfo.TreeData(:,1),solnInfo.TreeData(:,2),'.-'); % tree expansion
plot(pthObj.States(:,1), pthObj.States(:,2),'r-','LineWidth',2) % draw path
```



Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`stateSpaceDubins` | `stateSpaceReedsShepp`

Introduced in R2019b

stateSpaceSE3

SE(3) state space

Description

The `stateSpaceSE3` object stores parameters and states in the SE(3) state space, which is composed of state vectors represented by [*x*, *y*, *z*, *qw*, *qx*, *qy*, *qz*]. *x*, *y*, and *z* are Cartesian coordinates. *qw*, *qx*, *qy*, and *qz* represent the orientation in a quaternion. The object uses Euclidean distance calculation and linear interpolation for the translation component of the state. The object uses quaternion distance calculation and spherical linear interpolation for the rotation component of the state.

Creation

Syntax

```
space = stateSpaceSE3
space = stateSpaceSE3(bounds)
```

Description

`space = stateSpaceSE3` creates an SE(3) state space object with default state bounds for *x*, *y*, and *z*. The state variables *qw*, *qx*, *qy*, and *qz* corresponding to orientation are not bounded.

`space = stateSpaceSE3(bounds)` creates an SE(3) state space object with state bounds specified as a 7-by-2 matrix. Each row specifies the minimum and maximum value for a dimension of the state in the order *x*, *y*, *z*, *qw*, *qx*, *qy*, and *qz*. The input `bounds` sets the `StateBounds` property.

Properties

`Name` — Name of state space

`'SE3'` (default) | character vector

This property is read-only.

Name of state space, specified as a character vector.

Data Types: `char`

`NumStateVariables` — Number of state space dimensions

7 (default) | positive integer

This property is read-only.

Number of state space dimensions, specified dimensions, returned as a positive integer.

Data Types: `double`

`StateBounds` — Bounds of state variables

`[-100 100; -100 100; -100 100; Inf Inf; Inf Inf; Inf Inf; Inf Inf]` (default) | 7-by-2 matrix of real values

Bounds of state variables, specified as a 7-by-2 matrix of real values.

- The first row specifies the lower and upper bounds of the *x* state in meters.
- The second row specifies the lower and upper bounds of the *y* state in meters.
- The third row specifies the lower and upper bounds of the *z* state in meters.
- The fourth through the seventh rows specify the lower and upper bounds of the state variables *qw*, *qx*, *qy*, and *qz* respectively, corresponding to orientation as a quaternion.

Note The `StateBounds` property only affect the Cartesian components of the state. The state variables corresponding to orientation are not bounded.

Example: `stateSpaceSE3([-10 10; -10 10; -10 10; Inf Inf; Inf Inf; Inf Inf; Inf Inf])`

Example: `space.StateBounds = [-10 10; -10 10; -10 10; Inf Inf; Inf Inf; Inf Inf; Inf Inf]`

Data Types: `double`

`WeightXYZ` — Weight applied to *x*, *y*, and *z* distance calculation

1 (default) | positive real scalar

Weight applied to the *x*, *y*, and *z* distance calculation, specified as a positive real scalar. By default, the weight for translation is chosen to be greater than the weight for rotation.

The object calculates distance as:

$$d = \sqrt{w_{xyz}(d_x^2 + d_y^2 + d_z^2) + w_q d_q^2}$$

,

where w_{xyz} is the weight applied to *x*, *y*, and *z* coordinates, and w_q is the weight applied to the orientation in quaternion. d_x, d_y, and d_z are the distances in the *x*, *y*, and *z* directions, respectively. d_q is the quaternion distance.

Example: `space.WeightXYZ = 2`

Data Types: `double`

`WeightQuaternion` — Weight applied to quaternion distance calculation

`0.1` (default) | positive real scalar

Weight applied to quaternion distance calculation, specified as a positive real scalar. By default, the weight for rotation is chosen to be less than the weight for translation.

The object calculates distance as:

$$d = \sqrt{w_{xyz}(d_x^2 + d_y^2 + d_z^2) + w_q d_q^2}$$

,

where w_{xyz} is weight applied to x, y, and z coordinates, and w_q is the weight applied to the orientation in quaternion. d_x, d_y, and d_z are the distances in the x, y, and z direction, respectively. d_q is the quaternion distance.

Example: `space.WeightQuaternion = 0.5`

Data Types: `double`

Object Functions

copy	Create deep copy of state space object
distance	Distance between two states
enforceStateBounds	Reduce state to state bounds
interpolate	Interpolate between states
sampleUniform	Sample state using uniform distribution

Examples

Validate Path Through 3-D Occupancy Map Environment

Create a 3-D occupancy map and associated state validator. Plan, validate, and visualize a path through the occupancy map.

Load and Assign Map to State Validator

Load a 3-D occupancy map of a city block into the workspace. Specify a threshold for which cells to consider as obstacle-free.

```
mapData = load('dMapCityBlock.mat');
omap = mapData.omap;
omap.FreeThreshold = 0.5;
```

Inflate the occupancy map to add a buffer zone for safe operation around the obstacles.

```
inflate(omap,1)
```

Create an SE(3) state space object with bounds for state variables.

```
ss = stateSpaceSE3([-20 220;
    -20 220;
    -10 100;
    inf inf;
    inf inf;
    inf inf;
    inf inf]);
```

Create a 3-D occupancy map state validator using the created state space.

```
sv = validatorOccupancyMap3D(ss);
```

Assign the occupancy map to the state validator object. Specify the sampling distance interval.

```
sv.Map = omap;
sv.ValidationDistance = 0.1;
```

Plan and Visualize Path

Create a path planner with increased maximum connection distance. Reduce the maximum number of iterations.

```
planner = plannerRRT(ss,sv);
planner.MaxConnectionDistance = 50;
planner.MaxIterations = 1000;
```

Create a user-defined evaluation function for determining whether the path reaches the goal. Specify the probability of choosing the goal state during sampling.

```
planner.GoalReachedFcn = @(~,x,y)(norm(x(1:3)-y(1:3))<5);
planner.GoalBias = 0.1;
```

Set the start and goal states.

```
start = [40 180 25 0.7 0.2 0 0.1];
goal = [150 33 35 0.3 0 0.1 0.6];
```

Plan a path using the specified start, goal, and planner.

```
[pthObj,solnInfo] = plan(planner,start,goal);
```

Check that the points of the path are valid states.

```
isValid = isStateValid(sv,pthObj.States)
```

```
isValid = 7x1 logical array

   1
   1
   1
   1
   1
   1
   1
```

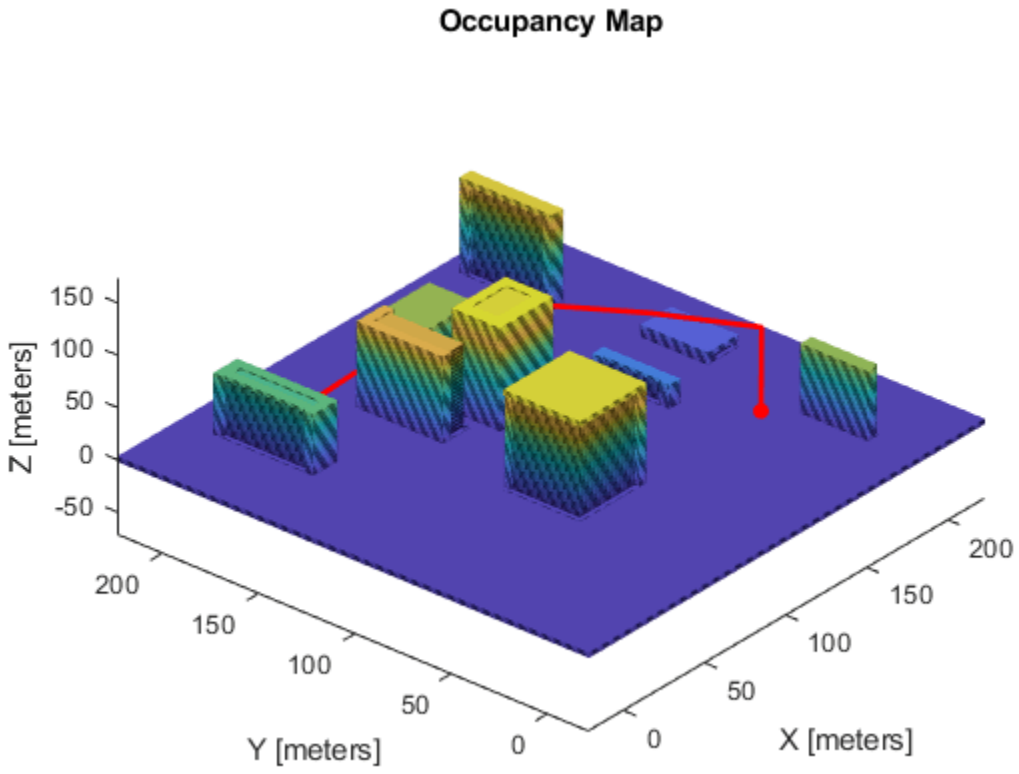
Check that the motion between each sequential path state is valid.

```
isPathValid = zeros(size(pthObj.States,1)-1,1,'logical');
for i = 1:size(pthObj.States,1)-1
    [isPathValid(i),~] = isMotionValid(sv,pthObj.States(i,:),...
        pthObj.States(i+1,:));
end
isPathValid
```

```
isPathValid = 6x1 logical array

   1
   1
   1
   1
   1
   1
```

Visualize the results.

```
show(omap)
hold on
scatter3(start(1,1),start(1,2),start(1,3),'g','filled') % draw start state
scatter3(goal(1,1),goal(1,2),goal(1,3),'r','filled')    % draw goal state
plot3(pthObj.States(:,1),pthObj.States(:,2),pthObj.States(:,3),...
    'r-','LineWidth',2) % draw path
```



Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`stateSpaceSE2` | `validatorOccupancyMap3D`

Introduced in R2020b

stateSpaceDubins

State space for Dubins vehicles

Description

The `stateSpaceDubins` object stores parameters and states in the Dubins state space, which is composed of state vectors represented by [x, y, θ]. x and y are Cartesian coordinates, and θ is the orientation angle. The Dubins state space has a lower limit on the turning radius (specified by the `MinTurningRadius` property in the object) for navigating between states and uses the shortest feasible curve to connect states.

Creation

Syntax

```
space = stateSpaceDubins
space = stateSpaceDubins(bounds)
```

Description

`space = stateSpaceDubins` creates a Dubins state space object with default state bounds for x, y, and θ.

`space = stateSpaceDubins(bounds)` specifies the `bounds` for x, y, and θ. The state values beyond the bounds are truncated to the bounds. The input, `bounds`, allows you to set the value of the StateBounds property.

Properties

`Name` — Name of state space
`'SE2 Dubins'` (default) | string

Name of state space, specified as a string.

`NumStateVariables` — Dimension of the state space
3 (default) | positive integer

This property is read-only.

Dimension of the state space, specified as a positive integer.

`StateBounds` — Bounds of state variables
`[-100 100; -100 100; -3.1416 3.1416]` (default) | 3-by-2 real-valued matrix

Bounds of state variables, specified as a 3-by-2 real-valued matrix.

- The first row specifies the lower and upper bounds for the x state in meters.

- The second row specifies the lower and upper bounds for the y state in meters.
- The third row specifies the lower and upper bounds for the θ state in radians.

Data Types: `double`

`MinTurningRadius` — Minimum turning radius
1 (default) | positive scalar

Minimum turning radius in meters, specified as a positive scalar. The minimum turning radius is for the smallest circle the vehicle can make with maximum steer in a single direction.

Object Functions

copy	Create deep copy of state space object
distance	Distance between two states
enforceStateBounds	Reduce state to state bounds
interpolate	Interpolate between states
sampleGaussian	Sample state using Gaussian distribution
sampleUniform	Sample state using uniform distribution

Examples

Plan Path Between Two States in Dubins State Space

Create a Dubins state space and set the minimum turing radius to `0.2`.

```
ss = stateSpaceDubins;
ss.MinTurningRadius = 0.2;
```

Create an `occupancyMap`-based state validator using the created state space.

```
sv = validatorOccupancyMap(ss);
```

Create an occupancy map from an example map and set map resolution as 10 cells/meter.

```
load exampleMaps
map = occupancyMap(simpleMap,10);
sv.Map = map;
```

Set validation distance for the validator.

```
sv.ValidationDistance = 0.01;
```

Update state space bounds to be the same as map limits.

```
ss.StateBounds = [map.XWorldLimits;map.YWorldLimits; [-pi pi]];
```

Create the path planner and increase max connection distance.

```
planner = plannerRRT(ss,sv);
planner.MaxConnectionDistance = 0.3;
```

Set the start and goal states.

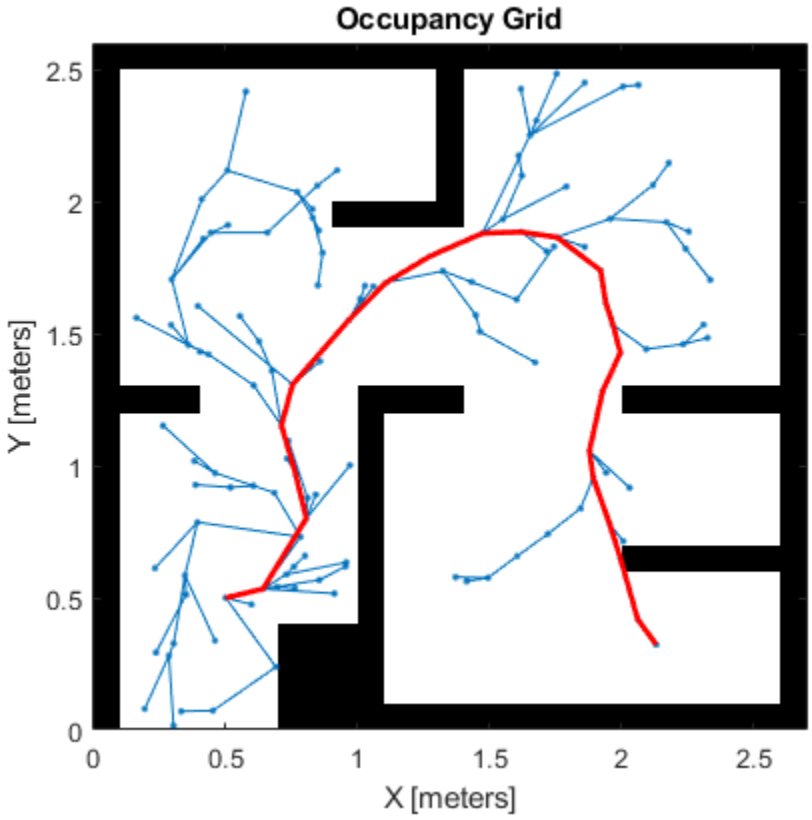
```
start = [0.5,0.5,0];
goal = [2.5,0.2,0];
```

Plan a path with default settings.

```
rng(100,'twister'); % repeatable result
[pthObj,solnInfo] = planner.plan(start,goal);
```

Visualize the results.

```
show(map);
hold on;
plot(solnInfo.TreeData(:,1),solnInfo.TreeData(:,2),'.-'); % tree expansion
plot(pthObj.States(:,1), pthObj.States(:,2),'r-','LineWidth',2) % draw path
```



Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`stateSpaceSE2` | `stateSpaceReedsShepp` | `dubinsConnection`

Introduced in R2019b

stateSpaceReedsShepp

State space for Reeds-Shepp vehicles

Description

The `stateSpaceReedsShepp` object stores parameters and states in the Reeds-Shepp state space, which is composed of state vectors represented by [x, y, θ]. x and y are Cartesian coordinates, and θ is the orientation angle. The Reeds-Shepp state space has a lower limit on the turning radius (specified by the `MinTurningRadius` property in the object) and forward and reverse costs (specified by the `ForwardCost` and `ReverseCost` properties in the object) for navigating between states.

Creation

Syntax

```
space = stateSpaceReedsShepp
sapce = stateSpaceReedsShepp(bounds)
```

Description

`space = stateSpaceReedsShepp` creates a Reeds-Shepp state space object with default state bounds for x, y, and θ.

`sapce = stateSpaceReedsShepp(bounds)` specifies the `bounds` for x, y, and θ. The state values beyond the bounds are truncated to the bounds. The input, `bounds`, sets the value of the StateBounds property.

Properties

`Name` — Name of state space
`'SE2 Dubins'` (default) | string

Name of state space, specified as a string.

`NumStateVariables` — Dimension of the state space
3 (default) | positive integer

This property is read-only.

Dimension of the state space, specified as a positive integer.

`StateBounds` — Bounds of state variables
`[-100 100; -100 100; -3.1416 3.1416]` (default) | 3-by-2 real-valued matrix

Bounds of state variables, specified as a 3-by-2 real-valued matrix.

- The first row specifies the lower and upper bounds for the x state in meters.
- The second row specifies the lower and upper bounds for the y state in meters.
- The third row specifies the lower and upper bounds for the θ state in radians.

Data Types: `double`

`MinTurningRadius` — Minimum turning radius
1 (default) | positive scalar

Minimum turning radius in meters, specified as a positive scalar. The minimum turning radius is for the smallest circle the vehicle can make with maximum steer in a single direction.

`ForwardCost` — Cost multiplier for forward motion
1 (default) | positive scalar

Cost multiplier for forward motion, specified as a positive scalar. Increase the cost to penalize forward motion.

`ReverseCost` — Cost multiplier for reverse motion
1 (default) | positive scalar

Cost multiplier for reverse motion, specified as a positive scalar. Increase the cost to penalize reverse motion.

Object Functions

copy	Create deep copy of state space object
distance	Distance between two states
enforceStateBounds	Reduce state to state bounds
interpolate	Interpolate between states
sampleGaussian	Sample state using Gaussian distribution
sampleUniform	Sample state using uniform distribution

Examples

Plan Path Between Two States in ReedsShepp State Space

Create a ReedsShepp state space.

```
ss = stateSpaceReedsShepp;
```

Create an `occupanyMap`-based state validator using the created state space.

```
sv = validatorOccupancyMap(ss);
```

Create an occupany map from an example map and set map resolution as 10 cells/meter.

```
load exampleMaps
map = occupancyMap(simpleMap,10);
sv.Map = map;
```

Set validation distance for the validator.

```
sv.ValidationDistance = 0.01;
```

Update state space bounds to be the same as map limits.

```
ss.StateBounds = [map.XWorldLimits;map.YWorldLimits; [-pi pi]];
```

Create the path planner and increase max connection distance.

```
planner = plannerRRT(ss,sv);
planner.MaxConnectionDistance = 0.3;
```

Set the start and goal states.

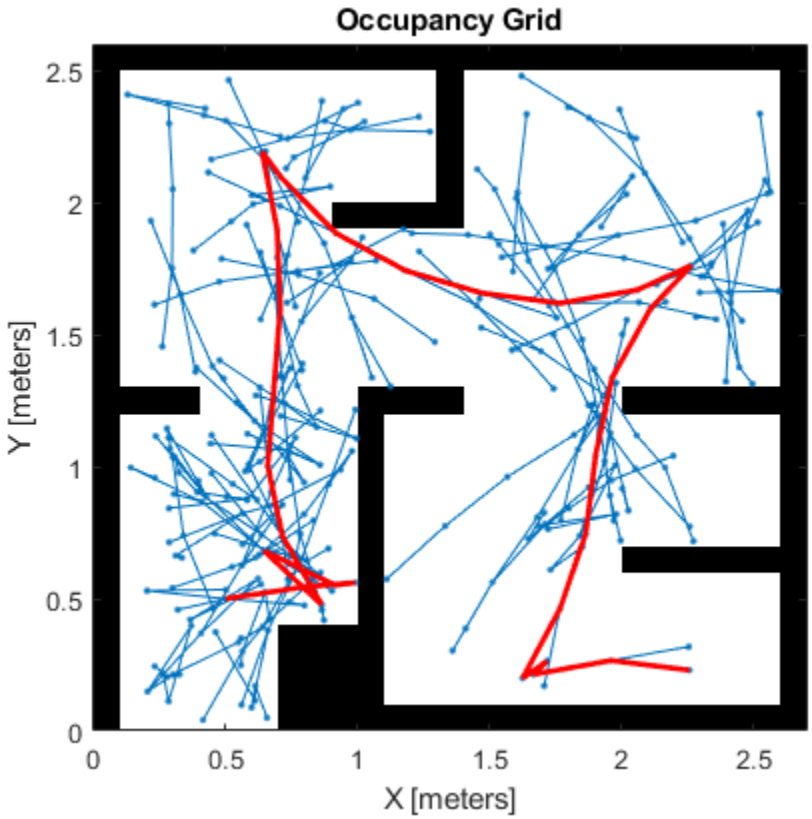
```
start = [0.5,0.5,0];
goal = [2.5,0.2,0];
```

Plan a path with default settings.

```
rng(100,'twister'); % repeatable result
[pthObj,solnInfo] = planner.plan(start,goal);
```

Visualize the results.

```
show(map);
hold on;
plot(solnInfo.TreeData(:,1),solnInfo.TreeData(:,2),'.-'); % tree expansion
plot(pthObj.States(:,1), pthObj.States(:,2),'r-','LineWidth',2) % draw path
```



Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`stateSpaceDubins` | `stateSpaceSE2` | `reedsSheppConnection`

Introduced in R2019b

distance

Distance between two states

Syntax

```
dist = distance(space,states1,states2)
```

Description

`dist = distance(space,states1,states2)` returns the distance between `states1` and `states2` in the specified state space `space`.

Examples

Calculate Distance Between Two States in SE3

Create an SE(3) state space.

```
space = stateSpaceSE3
```

```
space =
  stateSpaceSE3 with properties:

                Name: 'SE3'
         StateBounds: [7x2 double]
    NumStateVariables: 7
             WeightXYZ: 1
      WeightQuaternion: 0.1000
```

Calculate distance between two states.

```
dist = distance(space,[2 10 3 0.2 0 0 0.8],[0 -2.5 4 0.7 0.3 0 0])
```

```
dist = 12.7269
```

Calculate Euclidean distance between two states.

```
space.WeightQuaternion = 0;
distEuc = distance(space,[2 10 3 0.2 0 0 0.8; 4 5 2 1 2 4 2],[62 5 33 0.2 0 0 0.8; 9 9 3 3 1 3.1
```

```
distEuc = 2×1

   67.2681
    6.4807
```

Input Arguments

space — State space object

`stateSpaceSE2` object | `stateSpaceSE3` object | `stateSpaceDubins` object | `stateSpaceReedsShepp` object

State space object, specified as a `stateSpaceSE2`, `stateSpaceSE3`, `stateSpaceDubins`, or `stateSpaceReedsShepp` object.

states1 — Initial states for distance calculation

n-by-3 matrix of real values | *n*-by-7 matrix of real values

Initial states for distance calculation, specified as an *n*-by-3 or *n*-by-7 matrix of real values. *n* is the number of specified states.

For the 2-D state space objects `stateSpaceSE2`, `stateSpaceDubins`, and `stateSpaceReedsShepp`, each row is of form `[x y theta]`, which defines the *xy*-position and orientation angle `theta` of a state in the state space.

For the 3-D state space object `stateSpaceSE3`, each row is of form `[x y z qw qx qy qz]`, which defines the *xyz*-position and quaternion orientation `[qw qx qy qz]` of a state in the state space.

The function supports following combinations for distance calculation:

- *n*-to-*n* — *n* number of states in `states1` and *n* number of states in `states2`.

  For example, `distance(space,rand(10,7),rand(10,7))`
- 1-to-*n* — 1 state in `states1` and *n* number of states in `states2`.

  For example, `distance(space,rand(1,7),rand(10,7))`
- *n*-to-1 — *n* number of states in `states1` and 1 state in `states2`.

  For example, `distance(space,rand(10,7),rand(1,7))`

Data Types: `single` | `double`

states2 — Final states for distance calculation

n-by-3 matrix of real values | *n*-by-7 matrix of real values

Final states for distance calculation, specified as an *n*-by-3 or *n*-by-7 matrix of real values. *n* is the number of specified states.

For the 2-D state space objects `stateSpaceSE2`, `stateSpaceDubins`, and `stateSpaceReedsShepp`, each row is of form `[x y theta]`, which defines the *xy*-position and orientation angle `theta` of a state in the state space.

For the 3-D state space object `stateSpaceSE3`, each row is of form `[x y z qw qx qy qz]`, which defines the *xyz*-position and quaternion orientation `[qw qx qy qz]` of a state in the state space.

The function supports following combinations for distance calculation:

- *n*-to-*n* — *n* number of states in `states1` and *n* number of states in `states2`.

  For example, `distance(space,rand(10,7),rand(10,7))`

- 1-to-*n* — 1 state in `states1` and *n* number of states in `states2`.

  For example, `distance(space,rand(1,7),rand(10,7))`
- *n*-to-1 — *n* number of states in `states1` and 1 state in `states2`.

  For example, `distance(space,rand(10,7),rand(1,7))`

Data Types: `single` | `double`

Output Arguments

`dist` — Distance between states

n-element column vector

Distance between states, returned as an *n*-element column vector. *n* is the number of specified states.

The function supports following combinations for distance calculation:

- *n*-to-*n* — *n* number of states in `states1` and *n* number of states in `states2`.
- 1-to-*n* — 1 state in `states1` and *n* number of states in `states2`.
- *n*-to-1 — *n* number of states in `states1` and 1 state in `states2`.

Data Types: `single` | `double`

See Also

`stateSpaceSE2` | `stateSpaceSE3` | `stateSpaceDubins` | `stateSpaceReedsShepp`

Introduced in R2019b

interpolate

Interpolate between states

Syntax

```
interpStates = interpolate(space,state1,state2,ratio)
```

Description

`interpStates = interpolate(space,state1,state2,ratio)` interpolates states between the specified start state `state1` and end state `state2` based on the specified interpolation ratio `ratio`.

Examples

Interpolate Between States in SE(2)

Create an SE(2) state space with default properties.

```
space = stateSpaceSE2
```

```
space =
  stateSpaceSE2 with properties:

                Name: 'SE2'
         StateBounds: [3×2 double]
    NumStateVariables: 3
             WeightXY: 1
         WeightTheta: 0.1000
```

Create a pair of states in 2-D space.

```
state1 = [2 10 -pi];
state2 = [0 -2.5 -pi/4];
```

Interpolate halfway between two states.

```
state = interpolate(space,state1,state2,0.5)
```

```
state = 1×3

    1.0000    3.7500   -1.9635
```

Interpolate multiple points with a fixed interval.

```
states = interpolate(space,state1,state2,0:0.02:1)
```

```
states = 51×3

    2.0000   10.0000   -3.1416
    1.9600    9.7500   -3.0945
```

```
    1.9200    9.5000   -3.0473
    1.8800    9.2500   -3.0002
    1.8400    9.0000   -2.9531
    1.8000    8.7500   -2.9060
    1.7600    8.5000   -2.8588
    1.7200    8.2500   -2.8117
    1.6800    8.0000   -2.7646
    1.6400    7.7500   -2.7175
      ⋮
```

Interpolate Between States in SE(3)

Create an SE(3) state space with default properties.

```
space = stateSpaceSE3
```

```
space =
  stateSpaceSE3 with properties:

                Name: 'SE3'
         StateBounds: [7×2 double]
    NumStateVariables: 7
            WeightXYZ: 1
     WeightQuaternion: 0.1
```

Create a pair of states in 3-D space.

```
state1 = [2 10 3 0.2 0 0 0.8];
state2 = [0 -2.5 4 0.7 0.3 0 0];
```

Interpolate halfway between two states.

```
state = interpolate(space,state1,state2,0.5)
```

```
state = 1×7

            1         3.75          3.5       0.7428      0.25188            0      0.62033
```

Interpolate multiple points with a fixed interval.

```
states = interpolate(space,state1,state2,0:0.02:1)
```

```
states = 51×7

            2           10            3      0.24254            0            0      0.97014
         1.96         9.75         3.02      0.26633     0.010877            0      0.96382
         1.92          9.5         3.04      0.28994     0.021745            0       0.9568
         1.88         9.25         3.06      0.31333     0.032598            0      0.94908
         1.84            9         3.08       0.3365     0.043428            0      0.94068
          1.8         8.75          3.1      0.35943     0.054225            0       0.9316
         1.76          8.5         3.12      0.38209     0.064984            0      0.92184
         1.72         8.25         3.14      0.40448     0.075695            0      0.91141
         1.68            8         3.16      0.42657     0.086352            0      0.90032
```

```
    1.64         7.75         3.18      0.44835     0.096946            0      0.88858
⋮
```

Input Arguments

`space` — State space object

`stateSpaceSE2` object | `stateSpaceSE3` object | `stateSpaceDubins` object | `stateSpaceReedsShepp` object

State space object, specified as a `stateSpaceSE2`, `stateSpaceSE3`, `stateSpaceDubins`, or `stateSpaceReedsShepp` object.

`state1` — Start state for interpolation

three-element vector of real values | seven-element vector of real values

Start state for interpolation, specified as a three-element or seven-element vector of real values.

For the 2-D state space objects `stateSpaceSE2`, `stateSpaceDubins`, and `stateSpaceReedsShepp`, the state is a vector of form `[x y theta]`, which defines the *xy*-position and orientation angle `theta` of a state in the state space.

For the 3-D state space object `stateSpaceSE3`, the state is a vector of form `[x y z qw qx qy qz]`, which defines the *xyz*-position and quaternion orientation `[qw qx qy qz]` of a state in the state space.

Data Types: `single` | `double`

`state2` — End state for interpolation

three-element vector of real values | seven-element vector of real values

End state for interpolation, specified as a three-element or seven-element vector of real values.

For the 2-D state space objects `stateSpaceSE2`, `stateSpaceDubins`, and `stateSpaceReedsShepp`, the state is a vector of form `[x y theta]`, which defines the *xy*-position and orientation angle `theta` of a state in the state space.

For the 3-D state space object `stateSpaceSE3`, the state is a vector of form `[x y z qw qx qy qz]`, which defines the *xyz*-position and quaternion orientation `[qw qx qy qz]` of a state in the state space.

Data Types: `single` | `double`

`ratio` — Interpolation ratio

scalar in range [0, 1] | *n*-element column vector of values in the range [0, 1]

Interpolation ratio, specified as a scalar in the range of [0, 1], or an *n*-element column vector of values in the range [0, 1] . *n* is the number of desired interpolation points.

Data Types: `single` | `double`

Output Arguments

`interpStates` — Interpolated states

n-by-3 matrix of real values | *n*-by-7 matrix of real values

Interpolated states, returned as an *n*-by-3 or *n*-by-7 matrix of real values. *n* is the number of interpolation points specified by the `ratio` input argument.

For the 2-D state space objects `stateSpaceSE2`, `stateSpaceDubins`, and `stateSpaceReedsShepp`, each row is of form `[x y theta]`, which defines the *xy*-position and orientation angle `theta` of the interpolated states.

For the 3-D state space object `stateSpaceSE3`, each row is of form `[x y z qw qx qy qz]`, which defines the *xyz*-position and quaternion orientation `[qw qx qy qz]` of the interpolated states.

Data Types: `single` | `double`

See Also

`stateSpaceSE2` | `stateSpaceSE3` | `stateSpaceDubins` | `stateSpaceReedsShepp`

Introduced in R2019b

enforceStateBounds

Reduce state to state bounds

Syntax

```
boundedStates = enforceStateBounds(space,states)
```

Description

`boundedStates = enforceStateBounds(space,states)` reduces the specified states `states` to the state bounds in the `StateBounds` property of the specified state space object `space`.

Examples

Enforce State Bounds for SE(3) States

Create an SE(3) state space object.

```
space = stateSpaceSE3([-1 1; ...
    -2 2; ...
    -10 10; ...
    -inf inf; ...
    -inf inf; ...
    -inf inf; ...
    -inf inf])
```

```
space =
  stateSpaceSE3 with properties:

                Name: 'SE3'
         StateBounds: [7x2 double]
    NumStateVariables: 7
             WeightXYZ: 1
      WeightQuaternion: 0.1000
```

Create a pair of states in 3-D space.

```
state1 = [2 10 3 2 0 0 0.8];
state2 = [223 100 3 2 2 12 5];
```

Enforce state bounds for a single state.

```
boundedState = enforceStateBounds(space,state1)
```

```
boundedState = 1×7

    1.0000    2.0000    3.0000    2.0000         0         0    0.8000
```

Enforce state bounds for multiple states.

```
boundedStates = enforceStateBounds(space,[state1; state2])

boundedStates = 2×7

    1.0000    2.0000    3.0000    2.0000         0         0    0.8000
    1.0000    2.0000    3.0000    2.0000    2.0000   12.0000    5.0000
```

Input Arguments

space — State space object

`stateSpaceSE2` object | `stateSpaceSE3` object | `stateSpaceDubins` object | `stateSpaceReedsShepp` object

State space object, specified as a `stateSpaceSE2`, `stateSpaceSE3`, `stateSpaceDubins`, or `stateSpaceReedsShepp` object.

states — Unbounded states

n-by-3 matrix of real values | *n*-by-7 matrix of real values

Unbounded states, specified as an *n*-by-3 or *n*-by-7 matrix of real values.

For the 2-D state space objects `stateSpaceSE2`, `stateSpaceDubins`, and `stateSpaceReedsShepp`, each row is of form `[x y theta]`, which defines the *xy*-position and orientation angle `theta` of a state in the state space.

For the 3-D state space object `stateSpaceSE3`, each row is of form `[x y z qw qx qy qz]`, which defines the *xyz*-position and quaternion orientation `[qw qx qy qz]` of a state in the state space.

Data Types: `single` | `double`

Output Arguments

boundedStates — Bounded states

n-by-3 matrix of real values | *n*-by-7 matrix of real values

Bounded states, returned as an *n*-by-3 or *n*-by-7 matrix of real values. The value of *n* is same as for `states` input argument.

The function truncates each of the specified unbounded states to the bounds specified in the `StateBounds` property of the state space object `space`.

For the 2-D state space objects `stateSpaceSE2`, `stateSpaceDubins`, and `stateSpaceReedsShepp`, each row is of form `[x y theta]`, which defines the *xy*-position and orientation angle `theta` of the bounded states.

For the 3-D state space object `stateSpaceSE3`, each row is of form `[x y z qw qx qy qz]`, which defines the *xyz*-position and quaternion orientation `[qw qx qy qz]` of the bounded states.

Data Types: `single` | `double`

See Also

`stateSpaceSE2` | `stateSpaceSE3` | `stateSpaceDubins` | `stateSpaceReedsShepp`

Introduced in R2019b

copy

Create deep copy of state space object

Syntax

```
space2 = copy(space1)
```

Description

`space2 = copy(space1)` creates a deep copy of the specified state space object.

Examples

Create Deep Copy of SE(3) State Space Object

Create a default SE(3) state space object.

```
space = stateSpaceSE3;
```

Specify weight for the quaternion distance in state space calculation.

```
space.WeightQuaternion = 2/3
```

```
space =
  stateSpaceSE3 with properties:

                Name: 'SE3'
         StateBounds: [7x2 double]
    NumStateVariables: 7
             WeightXYZ: 1
      WeightQuaternion: 0.6667
```

Create a deep copy of the state space object.

```
space2 = copy(space)
```

```
space2 =
  stateSpaceSE3 with properties:

                Name: 'SE3'
         StateBounds: [7x2 double]
    NumStateVariables: 7
             WeightXYZ: 1
      WeightQuaternion: 0.6667
```

Verify that the `WeightQuaternion` property values of the two state space objects are equal.

```
isequal(space.WeightQuaternion,space2.WeightQuaternion)
```

```
ans = logical
   1
```

Input Arguments

space1 — State space object
`stateSpaceSE2` object | `stateSpaceSE3` object | `stateSpaceDubins` object | `stateSpaceReedsShepp` object

State space object, specified as a `stateSpaceSE2`, `stateSpaceSE3`, `stateSpaceDubins`, or `stateSpaceReedsShepp` object.

Output Arguments

space2 — State space object
`stateSpaceSE2` object | `stateSpaceSE3` object | `stateSpaceDubins` object | `stateSpaceReedsShepp` object

State space object, returned as a `stateSpaceSE2`, `stateSpaceSE3`, `stateSpaceDubins`, or `stateSpaceReedsShepp` object.

See Also
`stateSpaceSE2` | `stateSpaceSE3` | `stateSpaceDubins` | `stateSpaceReedsShepp`

Introduced in R2019b

sampleGaussian

Sample state using Gaussian distribution

Syntax

```
state = sampleGaussian(space,meanState,stdDev)
state = sampleGaussian(space,meanState,stdDev,numSamples)
```

Description

`state = sampleGaussian(space,meanState,stdDev)` returns a sample state of the state space based on a Gaussian (normal) distribution with specified mean, `meanState`, and standard deviation, `stdDev`.

`state = sampleGaussian(space,meanState,stdDev,numSamples)` returns a number of state samples. The number is equal to `numSamples`.

Input Arguments

`space` — State space object

`spaceSE2` object | `spaceDubins` object | `spaceReedsShepp` object

State space object, specified as a `stateSpaceSE2`, a `stateSpaceDubins`, or a `stateSpaceReedsShepp` object.

Data Types: `object`

`meanState` — Mean state

3-element vector of real values

Mean state of the Gaussian distribution for sampling, specified as a 3-element vector of real values.

Example: `[5 5 pi/3]`

Data Types: `single` | `double`

`stdDev` — Standard deviation

3-element vector of nonnegative values

Standard deviation of the Gaussian distribution for sampling, specified as a 3-element vector of nonnegative values.

Example: `[0.1 0.1 pi/18]`

Data Types: `single` | `double`

`numSamples` — Number of samples

positive integer

Number of samples, specified as a positive integer.

Data Types: `single` | `double`

Output Arguments

`state` — State samples

N-by-3 real-valued matrix

State samples, returned as an *N*-by-3 real-valued matrix. *N* is the number of samples. Each row of the matrix corresponds to one incidence of state in the state space.

Data Types: `single` | `double`

See Also

`stateSpaceSE2` | `stateSpaceDubins` | `stateSpaceReedsShepp`

Introduced in R2019b

sampleUniform

Sample state using uniform distribution

Syntax

```
state = sampleUniform(space)
state = sampleUniform(space,numSamples)
state = sampleUniform(space,nearState,distVector,numSamples)
```

Description

`state = sampleUniform(space)` samples a state within the bounds in the `StateBounds` property of the specified state space object `space` using a uniform probability distribution. For a `stateSpaceSE3` object, the state variables corresponding to orientation are bound to a unit quaternion using a uniform distribution of random rotations.

`state = sampleUniform(space,numSamples)` returns a specified number of state samples `numSamples` within the bounds of the state space object.

`state = sampleUniform(space,nearState,distVector,numSamples)` samples states in a specified subregion of the bounds of the state space object. Specify the center of the sampling region `nearState` and the distance from the center of the sampling region to its boundaries `distVector`.

Note The `stateSpaceSE3` object does not support this syntax.

Examples

Sample State Using Uniform Distribution in SE(3)

Create an SE(3) state space.

```
space = stateSpaceSE3([-10 10; -10 10; -10 10; inf inf; inf inf; inf inf; inf inf])
```

```
space =
  stateSpaceSE3 with properties:

                Name: 'SE3'
         StateBounds: [7x2 double]
    NumStateVariables: 7
            WeightXYZ: 1
     WeightQuaternion: 0.1000
```

Sample 3 states within full state bounds.

```
state = sampleUniform(space,3)
```

```
state = 3×7
```

```
    6.2945    8.1158   -7.4603    0.6316    0.3078   -0.6921   -0.1654
    8.2675    2.6472   -8.0492   -0.0834   -0.9448    0.2709    0.1641
   -4.4300    0.9376    9.1501    0.5771   -0.5458   -0.5490   -0.2601
```

Input Arguments

`space` — State space object

`stateSpaceSE2` object | `stateSpaceSE3` object | `stateSpaceDubins` object | `stateSpaceReedsShepp` object

State space object, specified as a `stateSpaceSE2`, `stateSpaceSE3`, `stateSpaceDubins`, or `stateSpaceReedsShepp` object.

`numSamples` — Number of samples

positive integer

Number of samples, specified as a positive integer.

Data Types: `single` | `double`

`nearState` — Center of sampling region

three-element vector of real values

Center of the sampling region, specified as a three-element vector of real values.

For the 2-D state space objects `stateSpaceSE2`, `stateSpaceDubins`, and `stateSpaceReedsShepp`, the state is a vector of form `[x y theta]`, which defines the *x*,*y*-position and orientation angle `theta` of a state in the state space.

Note The `stateSpaceSE3` object does not support this argument.

Data Types: `single` | `double`

`distVector` — Distance of sampling region boundary from center

three-element vector of positive numbers

Distance of sampling region boundary from the center, specified as a three-element vector of positive numbers.

For the 2-D state space objects `stateSpaceSE2`, `stateSpaceDubins`, and `stateSpaceReedsShepp`, the state is a vector of form `[x y theta]`, which defines the *x*,*y*-position and orientation angle `theta` of a state in the state space.

Note The `stateSpaceSE3` object does not support this argument.

Data Types: `single` | `double`

Output Arguments

`state` — State samples

n-by-3 matrix of real values | *n*-by-7 matrix of real values

State samples, returned as an *n*-by-3 or *n*-by-7 matrix of real values. *n* is the number of samples.

For the 2-D state space objects `stateSpaceSE2`, `stateSpaceDubins`, and `stateSpaceReedsShepp`, each row is of form `[x y theta]`, which defines the *xy*-position and orientation angle `theta` of the sampled states.

For the 3-D state space object `stateSpaceSE3`, each row is of form `[x y z qw qx qy qz]`, which defines the *xyz*-position and quaternion orientation `[qw qx qy qz]` of the sampled states.

Data Types: `single` | `double`

See Also

`stateSpaceSE2` | `stateSpaceSE3` | `stateSpaceDubins` | `stateSpaceReedsShepp`

Introduced in R2019b

trajectoryGeneratorFrenet

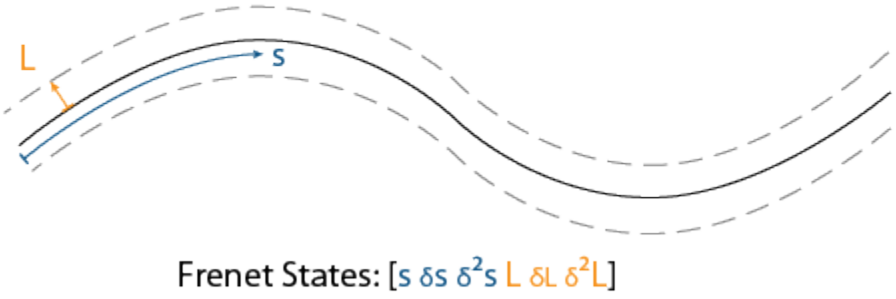
Find optimal trajectory along reference path

Description

The `trajectoryGeneratorFrenet` object generates alternate trajectories using fourth or fifth-order polynomials relative to a given reference path. Each trajectory defines a motion between Frenet states over a specified time span.

Frenet states describe their position, velocity and acceleration relative to a static reference path, specified as a `referencePathFrenet` object.

The object expresses Frenet states as a vector of form `[S dS ddS L dL ddL]`, where `S` is the arc length and `L` is the perpendicular deviation from the direction of the reference path. Derivatives of `S` are relative to time. Derivatives of `L` are relative to the arc length, `S`.



To generate alternative trajectories, specify the initial and terminal frenet states with a given time span to the `connect` object function.

Creation

Syntax

```
connectorFrenet = trajectoryGeneratorFrenet(refPath)
connectorFrenet = trajectoryGeneratorFrenet(
refPath,'TimeResolution',timeValue)
```

Description

`connectorFrenet = trajectoryGeneratorFrenet(refPath)` generates trajectories between initial and terminal states relative to a reference path `refPath` specified as a `referencePathFrenet` object. The `refPath` input argument sets the ReferencePath property.

`connectorFrenet = trajectoryGeneratorFrenet(refPath,'TimeResolution',timeValue)` specifies the time interval for discretization. The `timeValue` argument sets the TimeResolution property.

Properties

`TimeResolution` — Discretization time interval between sampled Frenet states

`0.1` (default) | positive scalar in seconds

Discretization time interval between sampled Frenet states, specified as a positive scalar in seconds. When using the `connect` object function, this property determines the resolution of the `Times` field of the generated trajectory structures, `frenetTrajectory` and `globalTrajectory`.

Data Types: `double`

`ReferencePath` — Reference path in Frenet coordinates

`referencePathFrenet`

Reference path in Frenet coordinates, specified as a `referencePathFrenet` object.

Object Functions

connect Connect initial and terminal Frenet states

Examples

Generate Alternative Trajectories for Reference Path

Generate alternative trajectories for a reference path using Frenet coordinates. Specify different initial and terminal states for your trajectories. Tune your states based on the generated trajectories.

Generate a reference path from a set of waypoints. Create a `trajectoryGeneratorFrenet` object from the reference path.

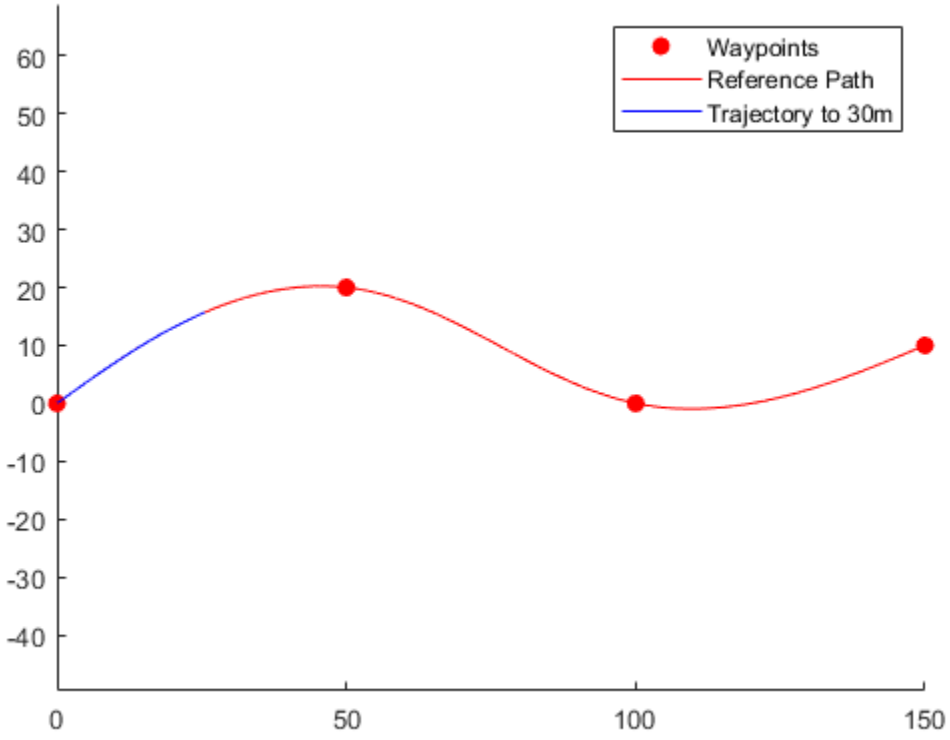
```
waypoints = [0 0; ...
    50 20; ...
    100 0; ...
    150 10];
refPath = referencePathFrenet(waypoints);
connector = trajectoryGeneratorFrenet(refPath);
```

Generate a five-second trajectory between the path origin and a point 30 m down the path as Frenet states.

```
initState = [0 0 0 0 0 0];  % [S ds ddS L dL ddL]
termState = [30 0 0 0 0 0]; % [S ds ddS L dL ddL]
[~,trajGlobal] = connect(connector,initState,termState,5);
```

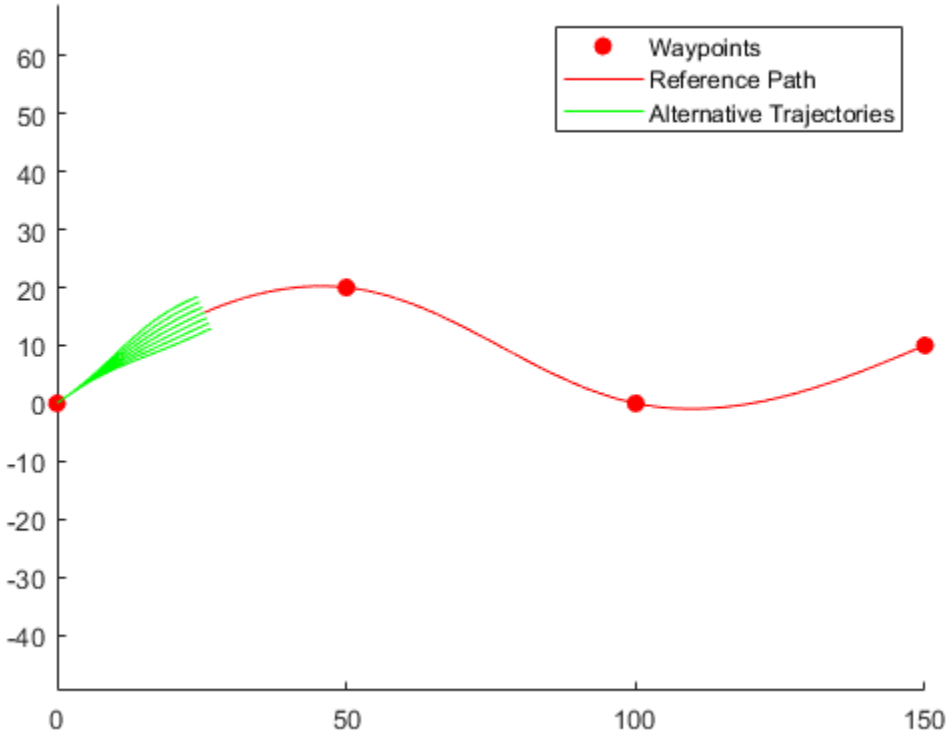
Display the trajectory in global coordinates.

```
show(refPath);
hold on
axis equal
plot(trajGlobal.Trajectory(:,1),trajGlobal.Trajectory(:,2),'b')
legend(["Waypoints","Reference Path","Trajectory to 30m"])
```



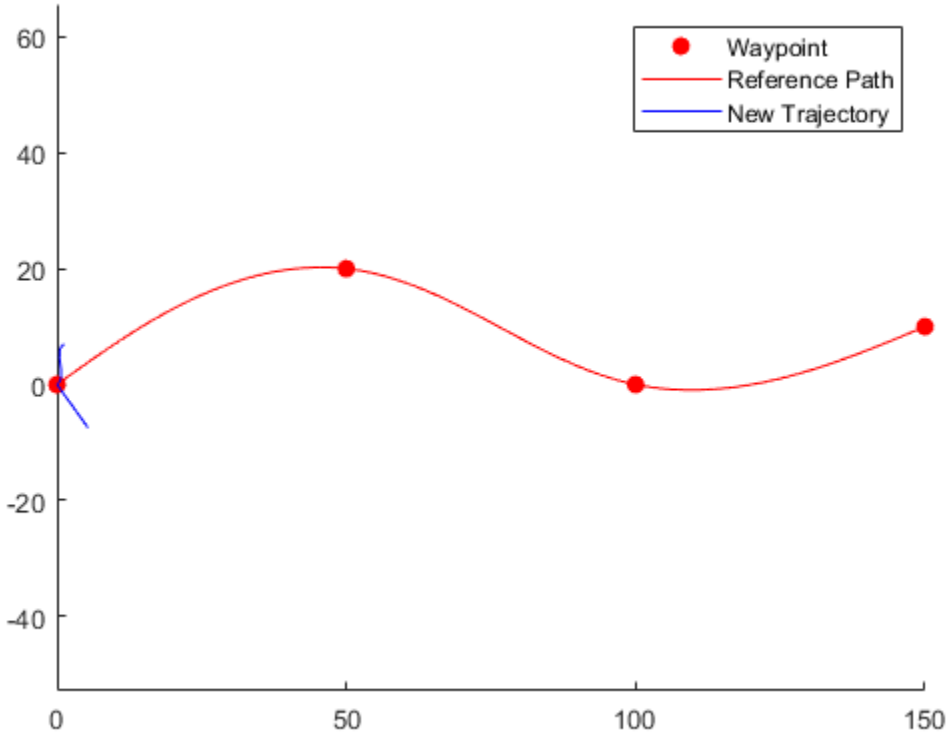
Create a matrix of terminal states with lateral deviations between –3 m and 3 m. Generate trajectories that cover the same arc length in 10 seconds, but deviate laterally from the reference path. Display the new alternative paths.

```
termStateDeviated = termState + ([-3:3]' * [0 0 0 1 0 0]);
[~,trajGlobal] = connect(connector,initState,termStateDeviated,10);

clf
show(refPath);
hold on
axis equal
for i = 1:length(trajGlobal)
    plot(trajGlobal(i).Trajectory(:,1),trajGlobal(i).Trajectory(:,2),'g')
end
legend(["Waypoints","Reference Path","Alternative Trajectories"])
hold off
```



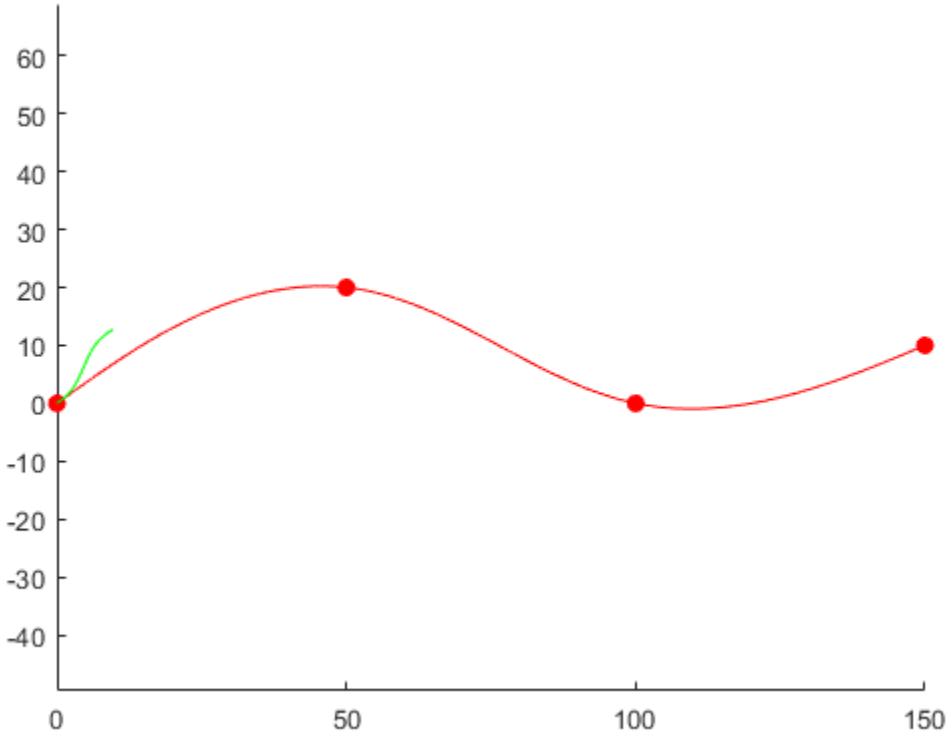
Specify a new terminal state to generate a new trajectory. This trajectory is not desirable because it requires reverse motion to achieve a lateral velocity of 10 m/s.

```
newTermState = [5 10 0 5 0 0];
[~,newTrajGlobal] = connect(connector,initState,newTermState,3);

clf
show(refPath);
hold on
axis equal
plot(newTrajGlobal.Trajectory(:,1),newTrajGlobal.Trajectory(:,2),'b');
legend(["Waypoint","Reference Path","New Trajectory"])
hold off
```



Relax the restriction on the longitudinal state by specifying an arc length of `NaN`. Generate and display the trajectory again. The new position shows a good alternative trajectory that deviates off the reference path.

```
relaxedTermState = [NaN 10 0 5 0 0];
[~,trajGlobalRelaxed] = connect(connector,initState,relaxedTermState,3);

clf
show(refPath);
hold on
axis equal
plot(trajGlobalRelaxed.Trajectory(:,1),trajGlobalRelaxed.Trajectory(:,2),'g');
hold off
```



Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

Objects

`referencePathFrenet`

Functions

`connect` | `closestPoint` | `frenet2global` | `global2frenet` | `interpolate` | `show`

Topics

“Highway Trajectory Planning Using Frenet Reference Path”

Introduced in R2020b

connect

Connect initial and terminal Frenet states

Syntax

```
frenetTrajectory = connect(connectorFrenet,initialState,terminalState,
timeSpan)
[___,globalTrajectory] = connect(___)
```

Description

`frenetTrajectory = connect(connectorFrenet,initialState,terminalState, timeSpan)` connects the specified initial Frenet states to the specified terminal states over a span of time in seconds. This object function supports 1-to-*n*, *n*-to-1, or *n*-to-*n* pairwise trajectory connections.

`[___,globalTrajectory] = connect(___)` returns the trajectories in global coordinates in addition to all arguments in the previous syntax.

Examples

Generate Alternative Trajectories for Reference Path

Generate alternative trajectories for a reference path using Frenet coordinates. Specify different initial and terminal states for your trajectories. Tune your states based on the generated trajectories.

Generate a reference path from a set of waypoints. Create a `trajectoryGeneratorFrenet` object from the reference path.

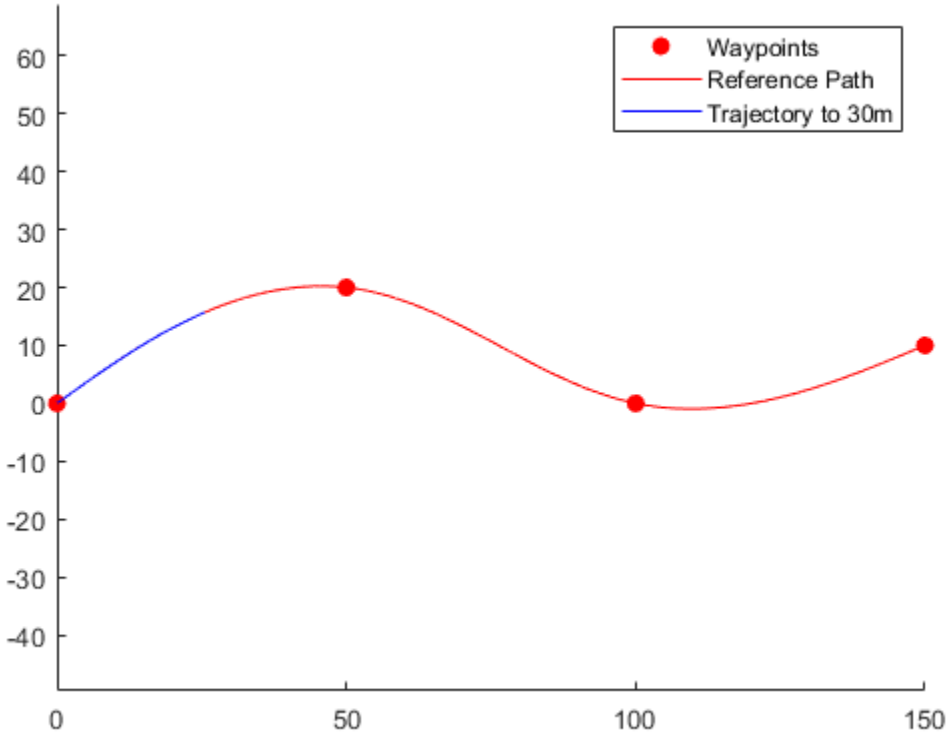
```
waypoints = [0 0; ...
    50 20; ...
    100 0; ...
    150 10];
refPath = referencePathFrenet(waypoints);
connector = trajectoryGeneratorFrenet(refPath);
```

Generate a five-second trajectory between the path origin and a point 30 m down the path as Frenet states.

```
initState = [0 0 0 0 0 0];  % [S ds ddS L dL ddL]
termState = [30 0 0 0 0 0]; % [S ds ddS L dL ddL]
[~,trajGlobal] = connect(connector,initState,termState,5);
```

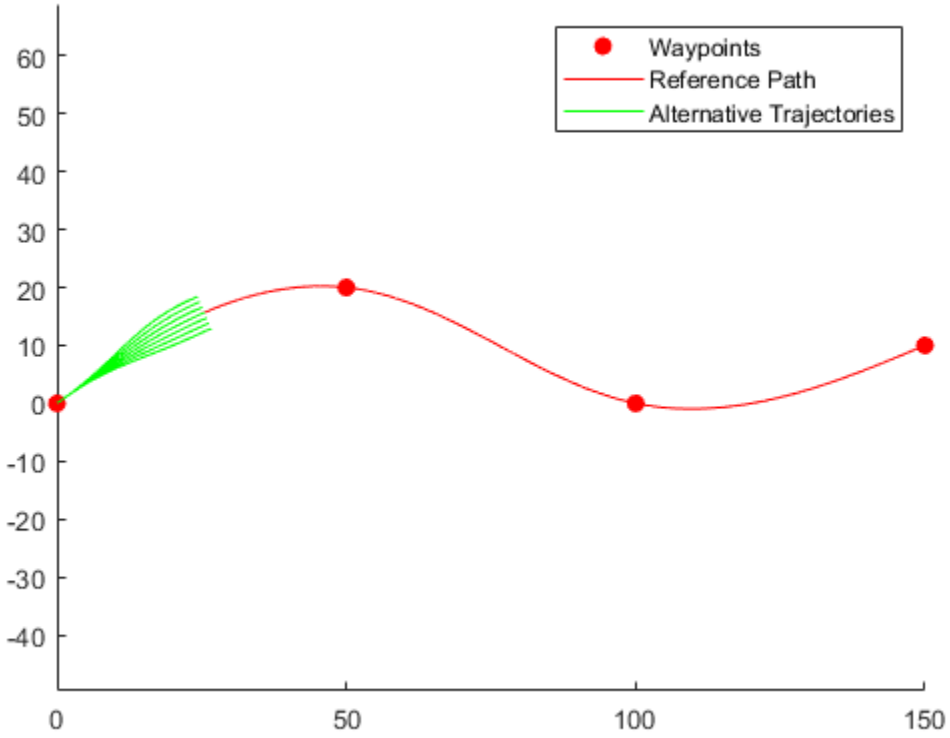
Display the trajectory in global coordinates.

```
show(refPath);
hold on
axis equal
plot(trajGlobal.Trajectory(:,1),trajGlobal.Trajectory(:,2),'b')
legend(["Waypoints","Reference Path","Trajectory to 30m"])
```



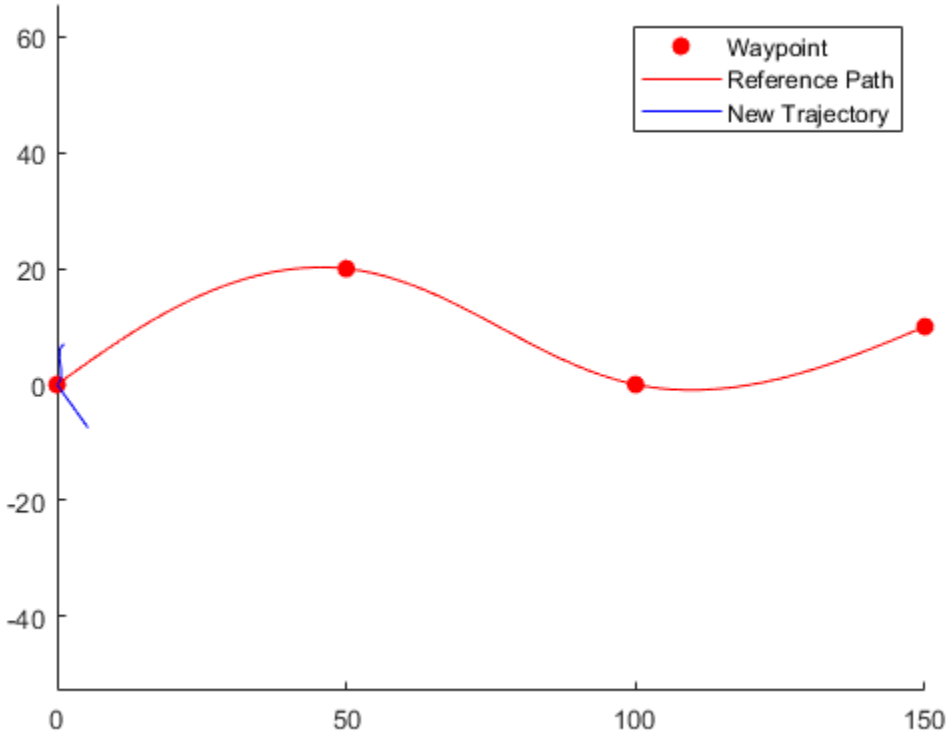
Create a matrix of terminal states with lateral deviations between –3 m and 3 m. Generate trajectories that cover the same arc length in 10 seconds, but deviate laterally from the reference path. Display the new alternative paths.

```
termStateDeviated = termState + ([-3:3]' * [0 0 0 1 0 0]);
[~,trajGlobal] = connect(connector,initState,termStateDeviated,10);

clf
show(refPath);
hold on
axis equal
for i = 1:length(trajGlobal)
    plot(trajGlobal(i).Trajectory(:,1),trajGlobal(i).Trajectory(:,2),'g')
end
legend(["Waypoints","Reference Path","Alternative Trajectories"])
hold off
```



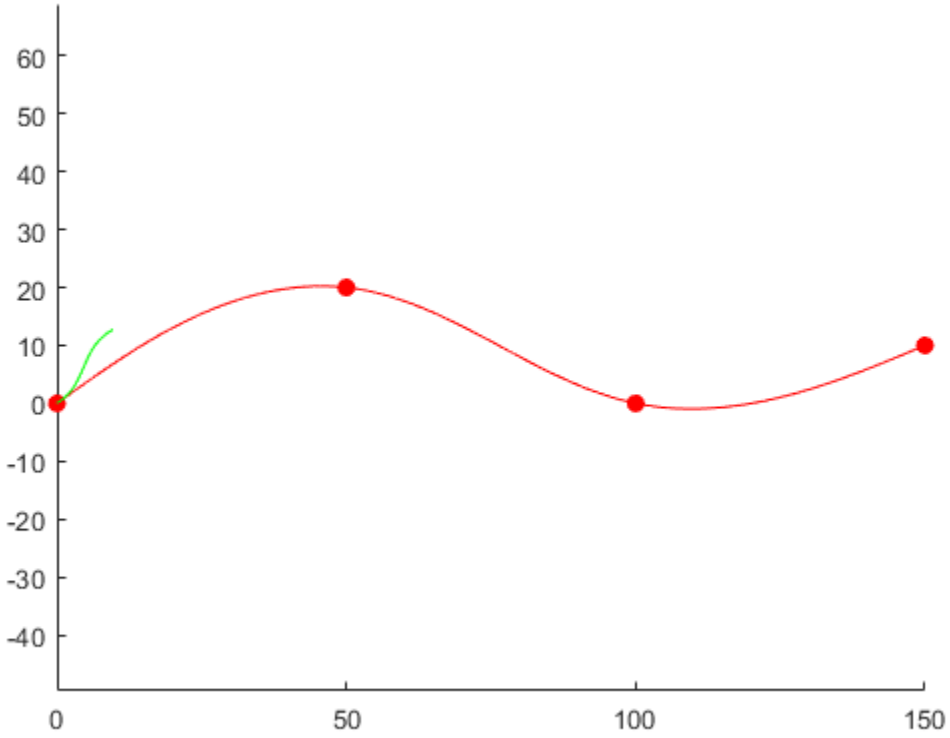
Specify a new terminal state to generate a new trajectory. This trajectory is not desirable because it requires reverse motion to achieve a lateral velocity of 10 m/s.

```
newTermState = [5 10 0 5 0 0];
[~,newTrajGlobal] = connect(connector,initState,newTermState,3);

clf
show(refPath);
hold on
axis equal
plot(newTrajGlobal.Trajectory(:,1),newTrajGlobal.Trajectory(:,2),'b');
legend(["Waypoint","Reference Path","New Trajectory"])
hold off
```



Relax the restriction on the longitudinal state by specifying an arc length of `NaN`. Generate and display the trajectory again. The new position shows a good alternative trajectory that deviates off the reference path.

```
relaxedTermState = [NaN 10 0 5 0 0];
[~,trajGlobalRelaxed] = connect(connector,initState,relaxedTermState,3);

clf
show(refPath);
hold on
axis equal
plot(trajGlobalRelaxed.Trajectory(:,1),trajGlobalRelaxed.Trajectory(:,2),'g');
hold off
```



Input Arguments

`connectorFrenet` — Frenet trajectory generator
`trajectoryGeneratorFrenet` object

Frenet trajectory generator, specified as a `trajectoryGeneratorFrenet` object.

`initialState` — Initial Frenet states
n-by-6 numeric matrix

Initial Frenet states, specified as an *n*-by-6 numeric matrix. Each row of the matrix is a set of Frenet coordinates for the initial state of a trajectory in the form `[S dS ddS L dL ddL]`. The value of *n* must be equal to the number of rows in the `terminalState` argument or `1`.

`terminalState` — Final Frenet states
n-by-6 numeric matrix

Final Frenet states, specified as an *n*-by-6 numeric matrix. Each row of the matrix is a set of Frenet coordinates for the initial state of a trajectory in the form `[S dS ddS L dL ddL]`. The value of *n* must be equal to the number of rows in the `initialState` argument or `1`.

`timeSpan` — Time horizon for all trajectories
positive scalar in seconds

Time horizon for all trajectories, specified as a positive scalar in seconds. The generated trajectories are sampled evenly across this time span based on the TimeResolution property of the `trajectoryGeneratorFrenet` object specified in the `connectorFrenet` argument.

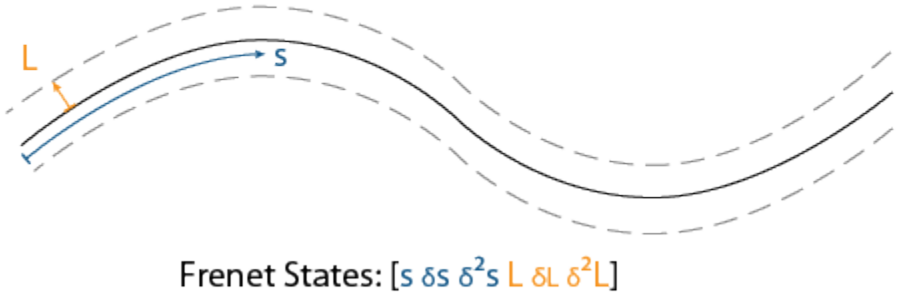
Output Arguments

`frenetTrajectory` — Frenet trajectories

structure | structure array

Frenet trajectories between all initial and final states, returned as a structure array with these fields:

- `Trajectory` — *n*-by-6 numeric matrix. Each row of the matrix is a set of Frenet coordinates for the initial state of a trajectory in the form `[S dS ddS L dL ddL]`.
- `Time` — Vector of positive scalars from `0` to `timeSpan` in seconds.



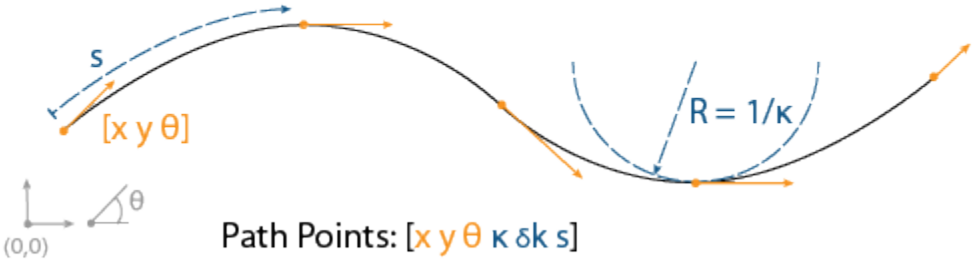
This function supports 1-to-*n*, *n*-to-1, or *n*-to-*n* pairwise trajectory connections based on the number of rows of `initialState` and `terminalState`.

`globalTrajectory` — Global trajectories

structure | structure array

Global trajectories between all initial and final states, returned as structure or structure array with fields:

- `Trajectory` — *n*-by-6 numeric matrix. Each row of the matrix is a set of global sates of the form `[x y theta kappa dkappa s]`.
- `Time` — Vector of positive scalars from `0` to `timeSpan` in seconds.



This function supports 1-to-*n*, *n*-to-1, or *n*-to-*n* pairwise trajectory connections based on the number of rows of `initialState` and `terminalState`.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

Objects

`referencePathFrenet` | `trajectoryGeneratorFrenet`

Functions

`closestPoint` | `frenet2global` | `global2frenet` | `interpolate` | `show`

Topics

“Highway Trajectory Planning Using Frenet Reference Path”

Introduced in R2020b

trajectoryOptimalFrenet

Find optimal trajectory along reference path

Description

The `trajectoryOptimalFrenet` object is a path planner which samples and evaluates local trajectories based on a reference path. The planner generates a set of terminal states based on the reference path and other parameters in the object. The planner then connects the state to each terminal state using 4^{th} or 5^{th} order polynomials. To choose an optimal path, sampled trajectories are evaluated for kinematic feasibility, collision, and cost.

Creation

Syntax

```
trajectoryOptimalFrenet(refPath,validator)
planner = trajectoryOptimalFrenet( ___ ,Name,Value)
```

Description

`trajectoryOptimalFrenet(refPath,validator)` creates a `trajectoryOptimalFrenet` object with reference path, `refPath`, in the form of an *n*-by-2 array of `[x y]` waypoints and a state validator, `validator`, specified as a `validatorOccupancyMap` object.

`planner = trajectoryOptimalFrenet( ___ ,Name,Value)` sets additional properties using one or more name-value pairs in any order.

Input Arguments

`refPath` — Reference path
n-by-2 matrix

Reference path, specified as an *n*-by-2 matrix of `[x y]` pairs, where *n* is the number of waypoints.

Example: [100,100;400,400]

Data Types: `double`

`validator` — State validator object
`validatorOccupancyMap` object

State validator object, specified as a `validatorOccupancyMap` object.

Properties

Note For the `'Weights'` and `'FeasibilityParameters'` properties, you cannot specify the entire structures at once. Instead, set their fields individually as name-value pairs. For example,

`trajectoryOptimalFrenet(refPath,validator,'Deviation',0)` sets the `'Deviation'` field of the structure `'Weights'`.

`Weights` — Weights for all trajectory costs

structure

The weights for all trajectory costs, specified as a structure containing scalars for the cost multipliers of the corresponding trajectory attributes. The total trajectory cost is a sum of all attributes multiplied by their weights. The structure has the these fields.

`Time` — Weight for time cost

`0` (default) | positive scalar

The cost function multiplies the weight by the total time taken to reach the terminal state. Specify this value as the comma-separated pair of `'Time'` and a positive scalar in seconds.

Data Types: `double`

`ArcLength` — Weight for arc length cost

`0` (default) | positive scalar

The cost function multiplies the weight by the total length of the generated trajectories. Specify this value as the comma-separated pair of `'ArcLength'` and a positive scalar in meters.

Data Types: `double`

`LateralSmoothness` — Weight for lateral jerk cost

`0` (default) | positive scalar

The cost function multiplies the weight by the integral of lateral jerk squared. This value determines the aggressiveness of the trajectory in the lateral direction (perpendicular to the reference path). Specify this value as the comma-separated pair of `'LateralSmoothness'` and a positive scalar. To penalize lateral jerk in the planned trajectory increase this cost value.

Data Types: `double`

`LongitudinalSmoothness` — Weight for longitudinal jerk cost

`0` (default) | positive scalar

The cost function multiplies the weight by the integral of longitudinal jerk squared. This value determines the aggressiveness of the trajectories in the longitudinal direction (direction of the reference path). Specify this value as the comma-separated pair of `'LongitudinalSmoothness'` and a positive scalar. To penalize large change in forward and backward acceleration increase this cost value.

Data Types: `double`

`Deviation` — Weight for deviation from reference path

`1` (default) | positive scalar

The cost function multiplies the weight by the perpendicular distance from the reference path at the end of the trajectory in meters. Specify this value as the comma-separated pair of `'Deviation'` and a positive scalar in meters.

Data Types: `double`

Data Types: `struct`

`FeasibilityParameters` — Structure containing feasibility parameters
structure

Feasibility parameters, specified as a structure containing scalar values to check the validity of a trajectory. The structure has the these fields.

`MaxCurvature` — Maximum curvature that vehicle can execute
`0.1` (default) | positive real scalar

Maximum curvature that the vehicle can execute. Specify this value as the comma-separated pair of `'MaxCurvature'` and a positive real scalar in m^{-1}. This value determines the kinematic feasibility of the trajectory.

Data Types: `double`

`MaxAcceleration` — Maximum acceleration in direction of motion of vehicle
`2.5` (default) | positive real scalar

Maximum acceleration in the direction of motion of the vehicle. Specify this value as the comma-separated pair of `'MaxAcceleration'` and a positive real scalar in m/s^2. To lower the limit on the acceleration of the vehicle in the forward or reverse direction decrease this value.

Data Types: `double`

Data Types: `struct`

`TimeResolution` — Trajectory discretization interval
`0.1` (default) | positive real scalar

Time interval between discretized states of the trajectory. Specify this value as the comma-separated pair of `'TimeResolution'` and a positive real scalar in seconds. These discretized states determine state validity and cost function.

Data Types: `double`

`CostFunction` — User-defined cost function
nullCost (default) | function handle

The user-defined cost function, specified as a function handle. The function must accept a matrix of *n*-by-7 states, `TRAJSTATES`, for each trajectory and return a cost value as a scalar. The `plan` function returns the path with the lowest cost.

For example, `leftLaneChangeCost = @(states)((states(end,2) < refPath(end,2))*10)` creates a cost function handle to prioritize left lane changes.

Data Types: `function handle`

`TrajectoryList` — List of all possible trajectories
structure array

This property is read-only.

The `'TrajectoryList'` property, returned as a structure array of all the candidate trajectories and their corresponding parameters. Each structure has the these fields:

- `Trajectory` — An *n*-by-7 matrix of [*x*, *y*, *theta*, *kappa*, *speed*, *acceleration*, *time*], where *n* is the number of trajectory waypoints.
- `Cost` — Cost of the trajectory.
- `MaxAcceleration` — Maximum acceleration of the trajectory.
- `MaxCurvature` — Maximum curvature of the trajectory.
- `Feasible` — A four-element vector [*velocity*, *acceleration*, *curvature*, *collision*] indicating the validity of the trajectory.

  The value of the elements can be either,

  - `1` — The trajectory is valid.
  - `0` — The trajectory is invalid.
  - `-1` — The trajectory is not checked.

Data Types: `struct`

`TerminalStates` — Structure of all goal states
structure

A structure that contains a list of goal states relative to the reference path. These parameters define the sampling behavior for generating alternative trajectory segments between start and each goal state. The structure has the these fields.

`Longitudinal` — Lengths of the trajectory segment
`30:15:90` (default) | vector

Lengths of the trajectory segment, specified as a vector in meters.

Data Types: `double`

`Lateral` — Array of deviations from reference path in perpendicular direction at goal state
`-2:1:2` (default) | vector

Array of deviations from reference path in perpendicular direction at goal state, specified as a vector in meters.

Data Types: `double`

Speed — Velocity at goal state in direction of motion
`10` (default) | positive scalar

Velocity at the goal state in the direction of motion, specified as a positive scalar in m/s.

Data Types: `double`

`Acceleration` — Acceleration at goal state in direction of motion
`0` (default) | positive scalar

Acceleration at the goal state in the direction of motion, specified as a positive scalar in m/s^2.

Data Types: `double`

`Time` — Array of end-times for executing trajectory segment
`7` (default) | positive vector

Array of end-times for executing the trajectory segment, specified as a positive vector in seconds.

Data Types: `double`

Data Types: `struct`

`Waypoints` — Waypoints of reference path

[ ] (default) | *n*-by-2 matrix

Waypoints of reference path, specified as an *n*-by-2 matrix of `[x y]` pairs, where *n* is the number of waypoints. Waypoints act as a reference for planning alternative trajectories optimized by this planner.

Data Types: `double`

`NumSegments` — Number of longitudinal segments for each trajectory

`1` (default) | positive scalar

Number of longitudinal segments for each trajectory. Specify this value as the comma-separated pair of `'NumSegments'` and a positive scalar. This property generates intermediate longitudinal terminal states to which all lateral terminal states are combined with for generating more motion primitives to each terminal state.

For example, `'NumSegments',2` creates two partitions between each longitudinal terminal state. Trajectories are generated to reach the intermediate longitudinal states with all the available lateral terminal states.

Data Types: `double`

`DeviationOffset` — Deviation offset from reference path in lateral direction

`0` (default) | scalar

Deviation offset from the reference path in the lateral direction. Specify this value as the comma-separated pair of `'DeviationOffset'` and a scalar. A negative value offset the deviation to the right, and a positive value offset the deviation to the left of the reference path in the lateral direction. Set this property to bias your solution to a certain turn direction when avoiding obstacles in the reference path.

Data Types: `double`

Object Functions

cart2frenet	Convert Cartesian states to Frenet states
copy	Create deep copy of object
frenet2cart	Convert Frenet states to Cartesian states
plan	Plan optimal trajectory
show	Visualize trajectories

Examples

Optimal Trajectory Planning in Frenet Space

This example shows how to plan an optimal trajectory using a `trajectoryOptimalFrenet` object.

Create and Assign Map to State Validator

Create a state validator object for collision checking.

```
stateValidator = validatorOccupancyMap;
```

Create an obstacle grid map.

```
grid = zeros(50,100);
grid(24:26,48:53) = 1;
```

Create a `binaryOccupancyMap` with the grid map.

```
map = binaryOccupancyMap(grid);
```

Assign the map and the state bounds to the state validator.

```
stateValidator.Map = map;
stateValidator.StateSpace.StateBounds(1:2,:) = [map.XWorldLimits; map.YWorldLimits];
```

Plan and Visualize Trajectory

Create a reference path for the planner to follow.

```
refPath = [0,25;100,25];
```

Initialize the planner object with the reference path, and the state validator.

```
planner = trajectoryOptimalFrenet(refPath,stateValidator);
```

Assign longitudinal terminal state, lateral deviation, and maximum acceleration values.

```
planner.TerminalStates.Longitudinal = 100;
planner.TerminalStates.Lateral = -10:5:10;
planner.FeasibilityParameters.MaxAcceleration = 10;
```

Specify the deviation offset value close to the left lateral terminal state to prioritize left lane changes.

```
planner.DeviationOffset = 5;
```

Trajectory Planning

Initial cartesian state of vehicle.

```
initCartState = [0 25 pi/9 0 0 0];
```

Convert cartesian state of vehicle to Frenet state.

```
initFrenetState = cart2frenet(planner,initCartState);
```

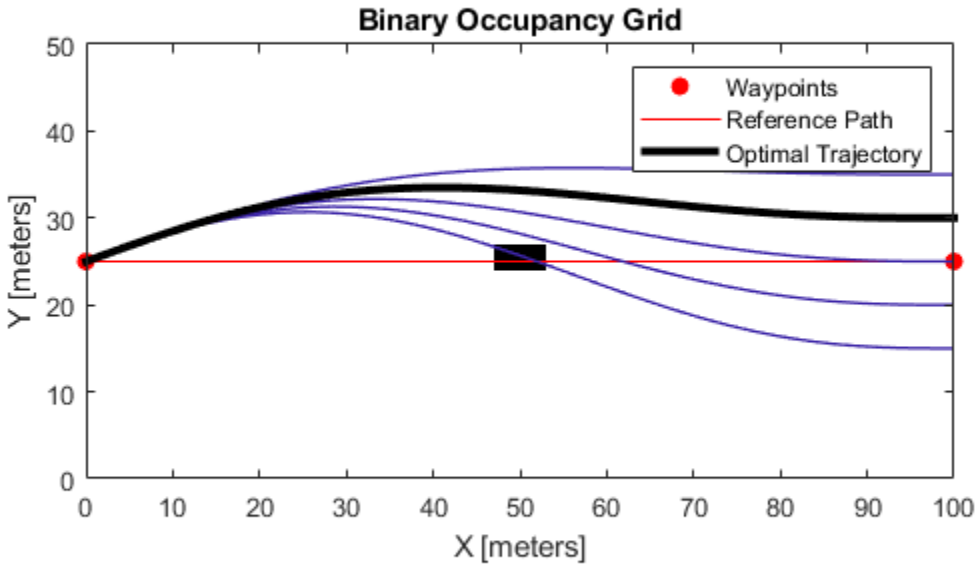
Plan a trajectory from initial Frenet state.

```
plan(planner,initFrenetState);
```

Trajectory Visualization

Visualize the map and the trajectories.

```
show(map)
hold on
show(planner,'Trajectory','all')
```



Partitioning Longitudinal Terminal States in Trajectory Generation

This example shows how to partition the longitudinal terminal states in optimal trajectory planning using a `trajectoryOptimalFrenet` object.

Create and Assign Map to State Validator

Create a state validator object for collision checking.

```
stateValidator = validatorOccupancyMap;
```

Create an obstacle grid map.

```
grid = zeros(50,100);
grid(25:27,28:33) = 1;
grid(16:18,37:42) = 1;
grid(29:31,72:77) = 1;
```

Create a `binaryOccupancyMap` with the grid map.

```
map = binaryOccupancyMap(grid);
```

Assign the map and the state bounds to the state validator.

```
stateValidator.Map = map;
stateValidator.StateSpace.StateBounds(1:2,:) = [map.XWorldLimits; map.YWorldLimits];
```

Plan and Visualize Trajectory

Create a reference path for the planner to follow.

```
refPath = [0,25;30,30;75,20;100,25];
```

Initialize the planner object with the reference path, and the state validator.

```
planner = trajectoryOptimalFrenet(refPath,stateValidator);
```

Assign longitudinal terminal state, lateral deviation, and maximum acceleration values.

```
planner.TerminalStates.Longitudinal = 100;
planner.TerminalStates.Lateral = -5:5:5;
planner.FeasibilityParameters.MaxAcceleration = 10;
```

Assign the number of partitions for the longitudinal terminal state.

```
planner.NumSegments = 3;
```

Trajectory Planning

Initial Frenet state of vehicle.

```
initFrenetState = zeros(1,6);
```

Plan a trajectory from initial Frenet state.

```
plan(planner,initFrenetState);
```

Trajectory Visualization

Visualize the map and the trajectories.

```
show(map)
hold on
show(planner,'Trajectory','all')
hold on
```

Generate Lane Boundaries

Calculate end of reference path as Frenet state.

```
refPathEnd = cart2frenet(planner,[planner.Waypoints(end,:) 0 0 0 0]);
```

Calculate lane offsets on both sides of the lateral terminal states with half lane width value.

```
laneOffsets = unique([planner.TerminalStates.Lateral+2.5 planner.TerminalStates.Lateral-2.5]);
```

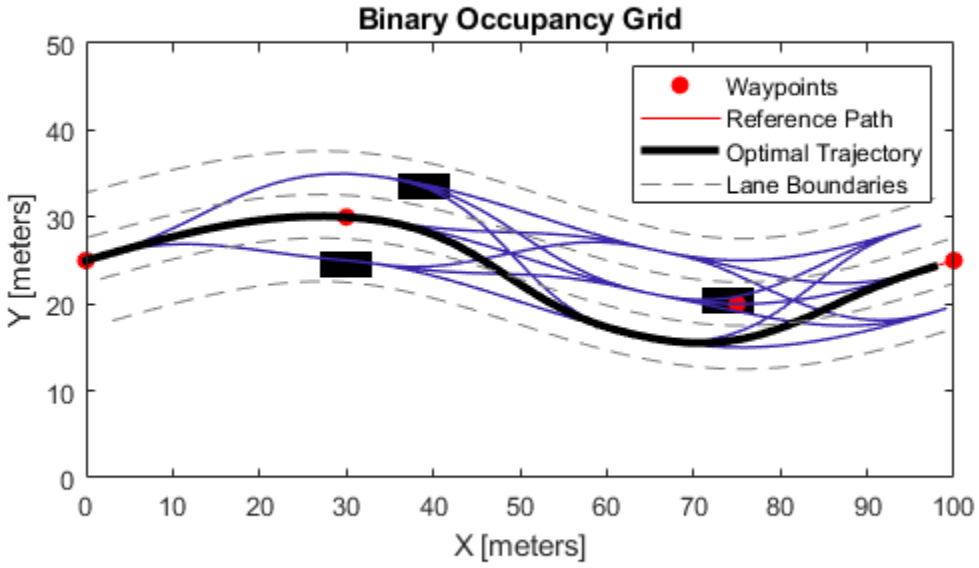
Calculate positions of lanes in Cartesian state.

```
numLaneOffsets = numel(laneOffsets);
xRefPathEnd = ceil(refPathEnd(1));
laneXY = zeros((numLaneOffsets*xRefPathEnd)+numLaneOffsets,2);
```

```
xIndex = 0;

for laneID = 1:numLaneOffsets
    for x = 1:xRefPathEnd
        laneCart = frenet2cart(planner,[x 0 0 laneOffsets(laneID) 0 0]);
        xIndex = xIndex + 1;
        laneXY(xIndex,:) = laneCart(1:2);
    end
    xIndex = xIndex + 1;
    laneXY(xIndex,:) = NaN(1,2);
end
```

Plot lane boundaries.

```
plot(laneXY(:,1),laneXY(:,2),'LineWidth',0.5,'Color',[0.5 0.5 0.5],'DisplayName','Lane Boundaries
```



Limitations

- Self-intersections in the reference path can lead to unexpected behavior.
- The planner does not support reverse driving.
- Initial orientation for planning should be within `-pi/2` and `pi/2` to the reference path.
- Limit the number of TerminalStates for real-time applications since computational complexity grows with it.

References

[1] Werling, Moritz, Julius Ziegler, Sören Kammel, and Sebastian Thrun. "Optimal Trajectory Generation for Dynamic Street Scenarios in a Frenet Frame." *2010 IEEE International Conference on Robotics and Automation*. 2010, pp. 987–993.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`validatorOccupancyMap` | `nav.StateValidator` | `plannerHybridAStar`

Introduced in R2019b

cart2frenet

Convert Cartesian states to Frenet states

Syntax

```
cart2frenet(planner,cartesianStates)
```

Description

`cart2frenet(planner,cartesianStates)` converts a six-element vector of `cartesianStates` `[`x`, `y`, `$theta$`, `$kappa$`, `$speed$`, `$acceleration$`]` to a six-element vector of Frenet states `[`s`, `ds/dt`, `d^2s/dt^2`, `l`, `dl/ds`, `d^2l/ds^2`]`, where s is arc length from the first point in reference path, and l is normal distance from the closest point at s on the reference path.

Examples

Optimal Trajectory Planning in Frenet Space

This example shows how to plan an optimal trajectory using a `trajectoryOptimalFrenet` object.

Create and Assign Map to State Validator

Create a state validator object for collision checking.

```
stateValidator = validatorOccupancyMap;
```

Create an obstacle grid map.

```
grid = zeros(50,100);
grid(24:26,48:53) = 1;
```

Create a `binaryOccupancyMap` with the grid map.

```
map = binaryOccupancyMap(grid);
```

Assign the map and the state bounds to the state validator.

```
stateValidator.Map = map;
stateValidator.StateSpace.StateBounds(1:2,:) = [map.XWorldLimits; map.YWorldLimits];
```

Plan and Visualize Trajectory

Create a reference path for the planner to follow.

```
refPath = [0,25;100,25];
```

Initialize the planner object with the reference path, and the state validator.

```
planner = trajectoryOptimalFrenet(refPath,stateValidator);
```

Assign longitudinal terminal state, lateral deviation, and maximum acceleration values.

```
planner.TerminalStates.Longitudinal = 100;
planner.TerminalStates.Lateral = -10:5:10;
planner.FeasibilityParameters.MaxAcceleration = 10;
```

Specify the deviation offset value close to the left lateral terminal state to prioritize left lane changes.

```
planner.DeviationOffset = 5;
```

Trajectory Planning

Initial cartesian state of vehicle.

```
initCartState = [0 25 pi/9 0 0 0];
```

Convert cartesian state of vehicle to Frenet state.

```
initFrenetState = cart2frenet(planner,initCartState);
```

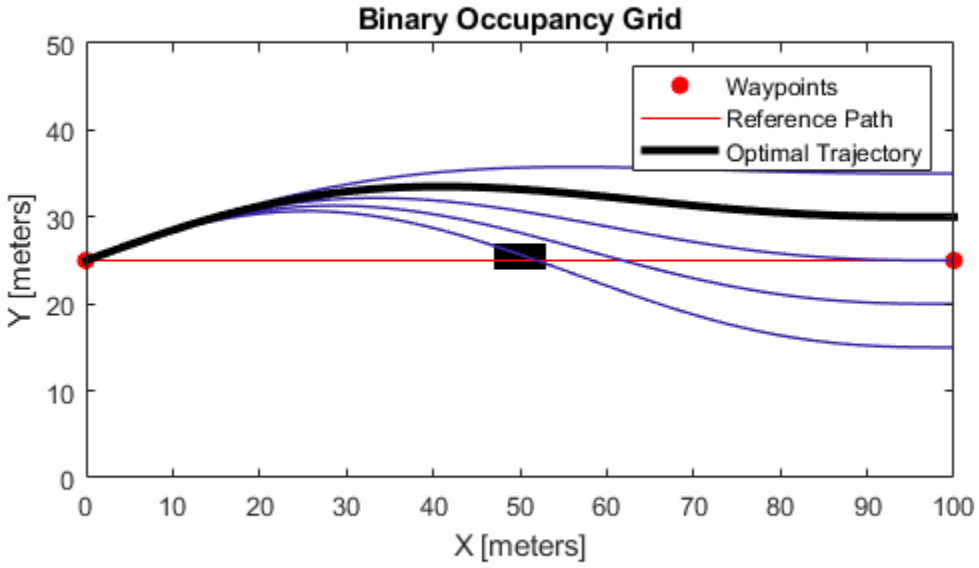
Plan a trajectory from initial Frenet state.

```
plan(planner,initFrenetState);
```

Trajectory Visualization

Visualize the map and the trajectories.

```
show(map)
hold on
show(planner,'Trajectory','all')
```



Partitioning Longitudinal Terminal States in Trajectory Generation

This example shows how to partition the longitudinal terminal states in optimal trajectory planning using a `trajectoryOptimalFrenet` object.

Create and Assign Map to State Validator

Create a state validator object for collision checking.

```
stateValidator = validatorOccupancyMap;
```

Create an obstacle grid map.

```
grid = zeros(50,100);
grid(25:27,28:33) = 1;
grid(16:18,37:42) = 1;
grid(29:31,72:77) = 1;
```

Create a `binaryOccupancyMap` with the grid map.

```
map = binaryOccupancyMap(grid);
```

Assign the map and the state bounds to the state validator.

```
stateValidator.Map = map;
stateValidator.StateSpace.StateBounds(1:2,:) = [map.XWorldLimits; map.YWorldLimits];
```

Plan and Visualize Trajectory

Create a reference path for the planner to follow.

```
refPath = [0,25;30,30;75,20;100,25];
```

Initialize the planner object with the reference path, and the state validator.

```
planner = trajectoryOptimalFrenet(refPath,stateValidator);
```

Assign longitudinal terminal state, lateral deviation, and maximum acceleration values.

```
planner.TerminalStates.Longitudinal = 100;
planner.TerminalStates.Lateral = -5:5:5;
planner.FeasibilityParameters.MaxAcceleration = 10;
```

Assign the number of partitions for the longitudinal terminal state.

```
planner.NumSegments = 3;
```

Trajectory Planning

Initial Frenet state of vehicle.

```
initFrenetState = zeros(1,6);
```

Plan a trajectory from initial Frenet state.

```
plan(planner,initFrenetState);
```

Trajectory Visualization

Visualize the map and the trajectories.

```
show(map)
hold on
show(planner,'Trajectory','all')
hold on
```

Generate Lane Boundaries

Calculate end of reference path as Frenet state.

```
refPathEnd = cart2frenet(planner,[planner.Waypoints(end,:) 0 0 0 0]);
```

Calculate lane offsets on both sides of the lateral terminal states with half lane width value.

```
laneOffsets = unique([planner.TerminalStates.Lateral+2.5 planner.TerminalStates.Lateral-2.5]);
```

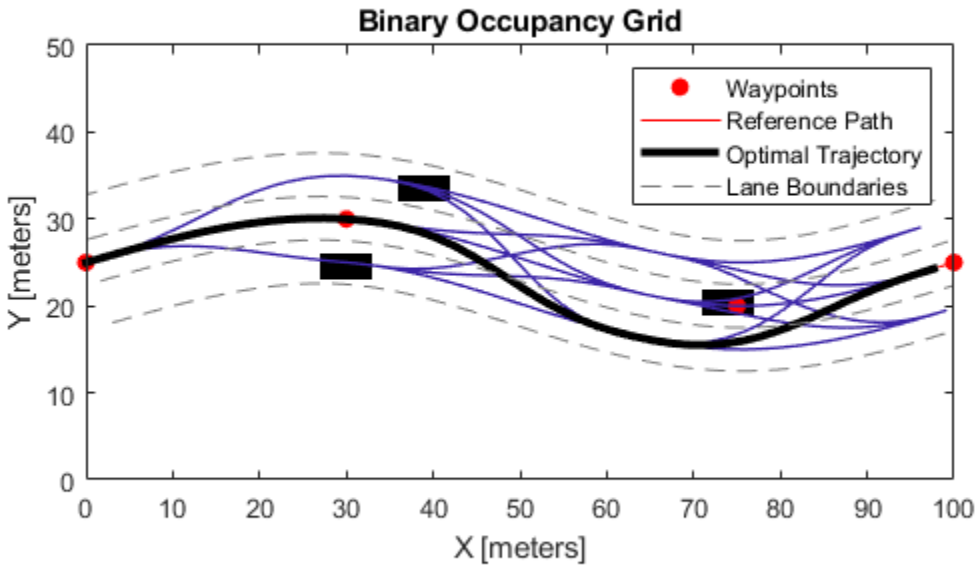
Calculate positions of lanes in Cartesian state.

```
numLaneOffsets = numel(laneOffsets);
xRefPathEnd = ceil(refPathEnd(1));
laneXY = zeros((numLaneOffsets*xRefPathEnd)+numLaneOffsets,2);
xIndex = 0;

for laneID = 1:numLaneOffsets
    for x = 1:xRefPathEnd
        laneCart = frenet2cart(planner,[x 0 0 laneOffsets(laneID) 0 0]);
```

```
        xIndex = xIndex + 1;
        laneXY(xIndex,:) = laneCart(1:2);
    end
    xIndex = xIndex + 1;
    laneXY(xIndex,:) = NaN(1,2);
end
```

Plot lane boundaries.

```
plot(laneXY(:,1),laneXY(:,2),'LineWidth',0.5,'Color',[0.5 0.5 0.5],'DisplayName','Lane Boundaries
```



Input Arguments

`planner` — Optimal trajectory planner in Frenet space

`trajectoryOptimalFrenet` object

Optimal trajectory planner in Frenet space, specified as a `trajectoryOptimalFrenet` object.

`cartesianStates` — Vector of Cartesian states

six-element vector

Vector of Cartesian states, specified as a 1-by-6 vector `[x, y, theta, kappa, speed, acceleration]`.

- *x* and *y* specify the position in meters.

- *theta* specifies the orientation angle in radians.
- *kappa* specifies the curvature in m^{-1}.
- *speed* specifies the velocity in `m/s`.
- *acceleration* specifies the acceleration in m/s^2.

Example: `[110 110 pi/4 0 0 0]`

Data Types: `double`

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`trajectoryOptimalFrenet` | `frenet2cart`

Introduced in R2019b

copy

Create deep copy of object

Syntax

```
plannerCopy = copy(planner)
```

Description

`plannerCopy = copy(planner)` creates a deep copy of the `trajectoryOptimalFrenet` object with the same properties.

Input Arguments

`planner` — Trajectory optimal Frenet object

`trajectoryOptimalFrenet` object

Trajectory optimal Frenet object, specified as a `trajectoryOptimalFrenet` object.

Output Arguments

`plannerCopy` — Copy of trajectory optimal Frenet object

`trajectoryOptimalFrenet` object

Copy of trajectory optimal Frenet object, returned as a `trajectoryOptimalFrenet` object with the same properties.

See Also

`trajectoryOptimalFrenet`

Introduced in R2020b

frenet2cart

Convert Frenet states to Cartesian states

Syntax

```
frenet2cart(planner,frenetStates)
```

Description

`frenet2cart(planner,frenetStates)` converts a six-element vector of `frenetStates` [s, ds/dt, d^2s/dt^2, l, dl/ds, d^2l/ds^2] to a six-element vector of Cartesian states [x, y, $theta$, $kappa$, $speed$, $acceleration$].

Examples

Optimal Trajectory Planning in Frenet Space

This example shows how to plan an optimal trajectory using a `trajectoryOptimalFrenet` object.

Create and Assign Map to State Validator

Create a state validator object for collision checking.

```
stateValidator = validatorOccupancyMap;
```

Create an obstacle grid map.

```
grid = zeros(50,100);
grid(24:26,48:53) = 1;
```

Create a `binaryOccupancyMap` with the grid map.

```
map = binaryOccupancyMap(grid);
```

Assign the map and the state bounds to the state validator.

```
stateValidator.Map = map;
stateValidator.StateSpace.StateBounds(1:2,:) = [map.XWorldLimits; map.YWorldLimits];
```

Plan and Visualize Trajectory

Create a reference path for the planner to follow.

```
refPath = [0,25;100,25];
```

Initialize the planner object with the reference path, and the state validator.

```
planner = trajectoryOptimalFrenet(refPath,stateValidator);
```

Assign longitudinal terminal state, lateral deviation, and maximum acceleration values.

```
planner.TerminalStates.Longitudinal = 100;
planner.TerminalStates.Lateral = -10:5:10;
planner.FeasibilityParameters.MaxAcceleration = 10;
```

Specify the deviation offset value close to the left lateral terminal state to prioritize left lane changes.

```
planner.DeviationOffset = 5;
```

Trajectory Planning

Initial cartesian state of vehicle.

```
initCartState = [0 25 pi/9 0 0 0];
```

Convert cartesian state of vehicle to Frenet state.

```
initFrenetState = cart2frenet(planner,initCartState);
```

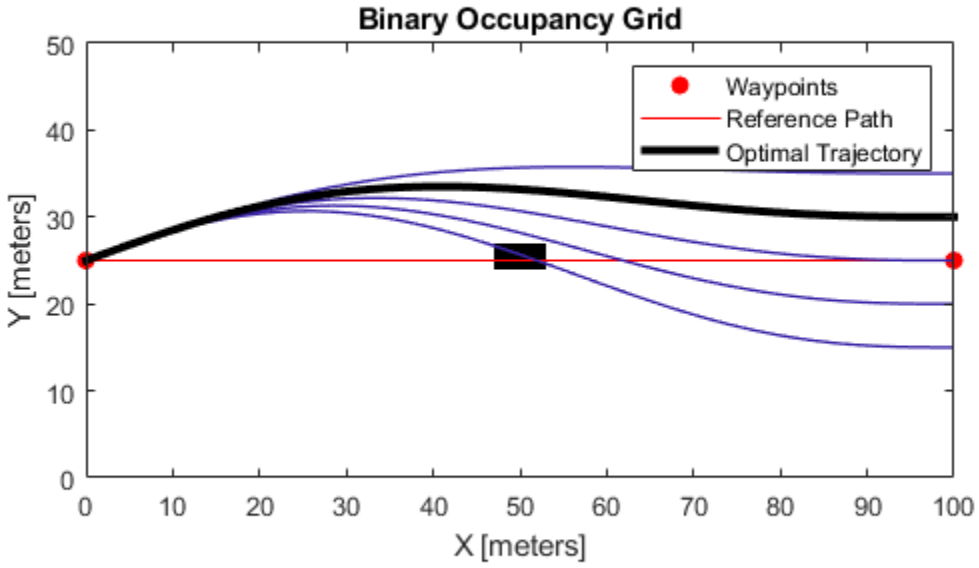
Plan a trajectory from initial Frenet state.

```
plan(planner,initFrenetState);
```

Trajectory Visualization

Visualize the map and the trajectories.

```
show(map)
hold on
show(planner,'Trajectory','all')
```



Partitioning Longitudinal Terminal States in Trajectory Generation

This example shows how to partition the longitudinal terminal states in optimal trajectory planning using a `trajectoryOptimalFrenet` object.

Create and Assign Map to State Validator

Create a state validator object for collision checking.

```
stateValidator = validatorOccupancyMap;
```

Create an obstacle grid map.

```
grid = zeros(50,100);
grid(25:27,28:33) = 1;
grid(16:18,37:42) = 1;
grid(29:31,72:77) = 1;
```

Create a `binaryOccupancyMap` with the grid map.

```
map = binaryOccupancyMap(grid);
```

Assign the map and the state bounds to the state validator.

```
stateValidator.Map = map;
stateValidator.StateSpace.StateBounds(1:2,:) = [map.XWorldLimits; map.YWorldLimits];
```

Plan and Visualize Trajectory

Create a reference path for the planner to follow.

```
refPath = [0,25;30,30;75,20;100,25];
```

Initialize the planner object with the reference path, and the state validator.

```
planner = trajectoryOptimalFrenet(refPath,stateValidator);
```

Assign longitudinal terminal state, lateral deviation, and maximum acceleration values.

```
planner.TerminalStates.Longitudinal = 100;
planner.TerminalStates.Lateral = -5:5:5;
planner.FeasibilityParameters.MaxAcceleration = 10;
```

Assign the number of partitions for the longitudinal terminal state.

```
planner.NumSegments = 3;
```

Trajectory Planning

Initial Frenet state of vehicle.

```
initFrenetState = zeros(1,6);
```

Plan a trajectory from initial Frenet state.

```
plan(planner,initFrenetState);
```

Trajectory Visualization

Visualize the map and the trajectories.

```
show(map)
hold on
show(planner,'Trajectory','all')
hold on
```

Generate Lane Boundaries

Calculate end of reference path as Frenet state.

```
refPathEnd = cart2frenet(planner,[planner.Waypoints(end,:) 0 0 0 0]);
```

Calculate lane offsets on both sides of the lateral terminal states with half lane width value.

```
laneOffsets = unique([planner.TerminalStates.Lateral+2.5 planner.TerminalStates.Lateral-2.5]);
```

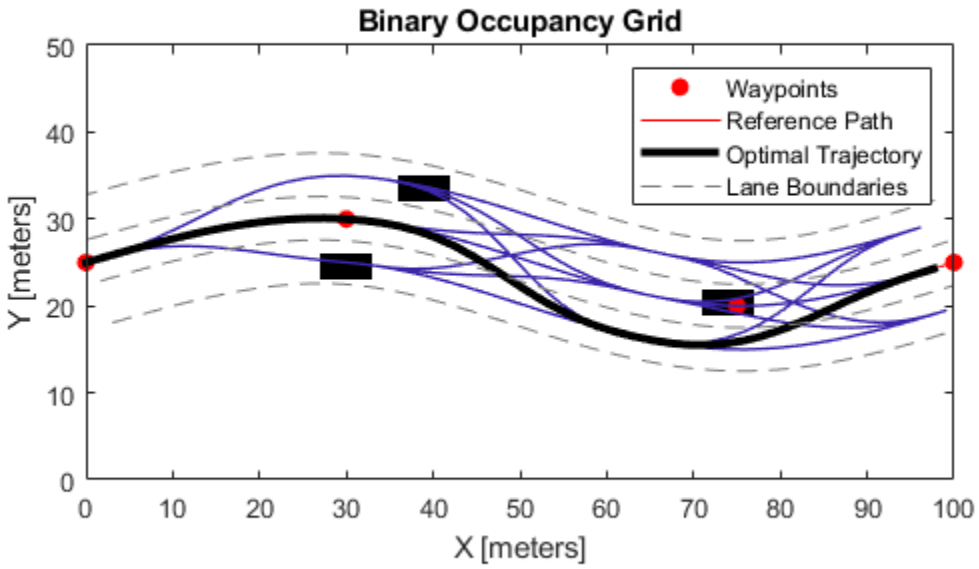
Calculate positions of lanes in Cartesian state.

```
numLaneOffsets = numel(laneOffsets);
xRefPathEnd = ceil(refPathEnd(1));
laneXY = zeros((numLaneOffsets*xRefPathEnd)+numLaneOffsets,2);
xIndex = 0;

for laneID = 1:numLaneOffsets
    for x = 1:xRefPathEnd
        laneCart = frenet2cart(planner,[x 0 0 laneOffsets(laneID) 0 0]);
```

```
        xIndex = xIndex + 1;
        laneXY(xIndex,:) = laneCart(1:2);
    end
    xIndex = xIndex + 1;
    laneXY(xIndex,:) = NaN(1,2);
end
```

Plot lane boundaries.

```
plot(laneXY(:,1),laneXY(:,2),'LineWidth',0.5,'Color',[0.5 0.5 0.5],'DisplayName','Lane Boundaries
```



Input Arguments

`planner` — Optimal trajectory planner in Frenet space

`trajectoryOptimalFrenet` object

Optimal trajectory planner in Frenet space, specified as a `trajectoryOptimalFrenet` object.

`frenetStates` — Vector of Frenet states

six-element vector

Vector of Frenet states, specified as a 1-by-6 vector, $[s,\ ds/dt,\ d^2s/dt^2,\ l,\ dl/ds,\ d^2l/ds^2]$.

- *s* specifies the arc length from the first point in reference path in meters.
- *ds/dt* specifies the first derivative of arc length.

- d^2s/dt^2 specifies the second derivative of arc length.
- l specifies the normal distance from the closest point in the reference path in meters.
- dl/ds specifies the first derivative of normal distance.
- d^2l/ds^2 specifies the second derivative of normal distance.

Example: `[10 1 0 3 0 0]`

Data Types: `double`

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`trajectoryOptimalFrenet` | `cart2frenet`

Introduced in R2019b

plan

Plan optimal trajectory

Syntax

```
[traj,index,cost,flag] = plan(planner,start)
```

Description

`[traj,index,cost,flag] = plan(planner,start)` computes a feasible trajectory, `traj`, from a list of candidate trajectories generated from the `trajectoryOptimalFrenet` object, `planner`. `start` is specified as a six-element vector `[`s`, `ds/dt`, `d^2s/dt^2`, `l`, `dl/ds`, `d^2l/ds^2`]`, where s is the arc length from the first point in the reference path, and l is normal distance from the closest point at s on the reference path.

The output trajectory, `traj`, also has an associated `cost` and `index` for the TrajectoryList property of the planner. `flag` is a numeric exit flag indicating status of the solution.

To improve the results of the planning output, modify the parameters on the `planner` object.

Examples

Optimal Trajectory Planning in Frenet Space

This example shows how to plan an optimal trajectory using a `trajectoryOptimalFrenet` object.

Create and Assign Map to State Validator

Create a state validator object for collision checking.

```
stateValidator = validatorOccupancyMap;
```

Create an obstacle grid map.

```
grid = zeros(50,100);
grid(24:26,48:53) = 1;
```

Create a `binaryOccupancyMap` with the grid map.

```
map = binaryOccupancyMap(grid);
```

Assign the map and the state bounds to the state validator.

```
stateValidator.Map = map;
stateValidator.StateSpace.StateBounds(1:2,:) = [map.XWorldLimits; map.YWorldLimits];
```

Plan and Visualize Trajectory

Create a reference path for the planner to follow.

```
refPath = [0,25;100,25];
```

Initialize the planner object with the reference path, and the state validator.

```
planner = trajectoryOptimalFrenet(refPath,stateValidator);
```

Assign longitudinal terminal state, lateral deviation, and maximum acceleration values.

```
planner.TerminalStates.Longitudinal = 100;
planner.TerminalStates.Lateral = -10:5:10;
planner.FeasibilityParameters.MaxAcceleration = 10;
```

Specify the deviation offset value close to the left lateral terminal state to prioritize left lane changes.

```
planner.DeviationOffset = 5;
```

Trajectory Planning

Initial cartesian state of vehicle.

```
initCartState = [0 25 pi/9 0 0 0];
```

Convert cartesian state of vehicle to Frenet state.

```
initFrenetState = cart2frenet(planner,initCartState);
```

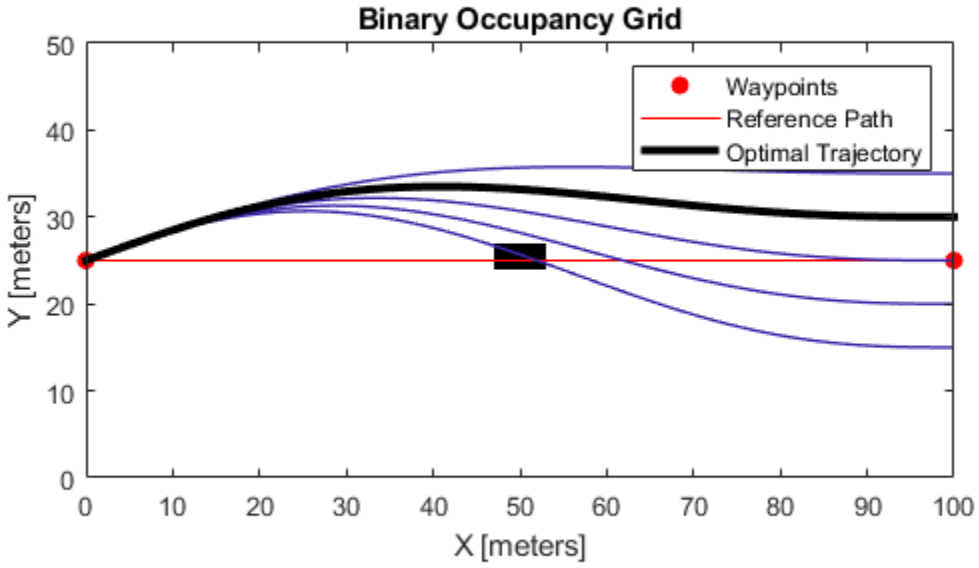
Plan a trajectory from initial Frenet state.

```
plan(planner,initFrenetState);
```

Trajectory Visualization

Visualize the map and the trajectories.

```
show(map)
hold on
show(planner,'Trajectory','all')
```



Partitioning Longitudinal Terminal States in Trajectory Generation

This example shows how to partition the longitudinal terminal states in optimal trajectory planning using a `trajectoryOptimalFrenet` object.

Create and Assign Map to State Validator

Create a state validator object for collision checking.

```
stateValidator = validatorOccupancyMap;
```

Create an obstacle grid map.

```
grid = zeros(50,100);
grid(25:27,28:33) = 1;
grid(16:18,37:42) = 1;
grid(29:31,72:77) = 1;
```

Create a `binaryOccupancyMap` with the grid map.

```
map = binaryOccupancyMap(grid);
```

Assign the map and the state bounds to the state validator.

```
stateValidator.Map = map;
stateValidator.StateSpace.StateBounds(1:2,:) = [map.XWorldLimits; map.YWorldLimits];
```

Plan and Visualize Trajectory

Create a reference path for the planner to follow.

```
refPath = [0,25;30,30;75,20;100,25];
```

Initialize the planner object with the reference path, and the state validator.

```
planner = trajectoryOptimalFrenet(refPath,stateValidator);
```

Assign longitudinal terminal state, lateral deviation, and maximum acceleration values.

```
planner.TerminalStates.Longitudinal = 100;
planner.TerminalStates.Lateral = -5:5:5;
planner.FeasibilityParameters.MaxAcceleration = 10;
```

Assign the number of partitions for the longitudinal terminal state.

```
planner.NumSegments = 3;
```

Trajectory Planning

Initial Frenet state of vehicle.

```
initFrenetState = zeros(1,6);
```

Plan a trajectory from initial Frenet state.

```
plan(planner,initFrenetState);
```

Trajectory Visualization

Visualize the map and the trajectories.

```
show(map)
hold on
show(planner,'Trajectory','all')
hold on
```

Generate Lane Boundaries

Calculate end of reference path as Frenet state.

```
refPathEnd = cart2frenet(planner,[planner.Waypoints(end,:) 0 0 0 0]);
```

Calculate lane offsets on both sides of the lateral terminal states with half lane width value.

```
laneOffsets = unique([planner.TerminalStates.Lateral+2.5 planner.TerminalStates.Lateral-2.5]);
```

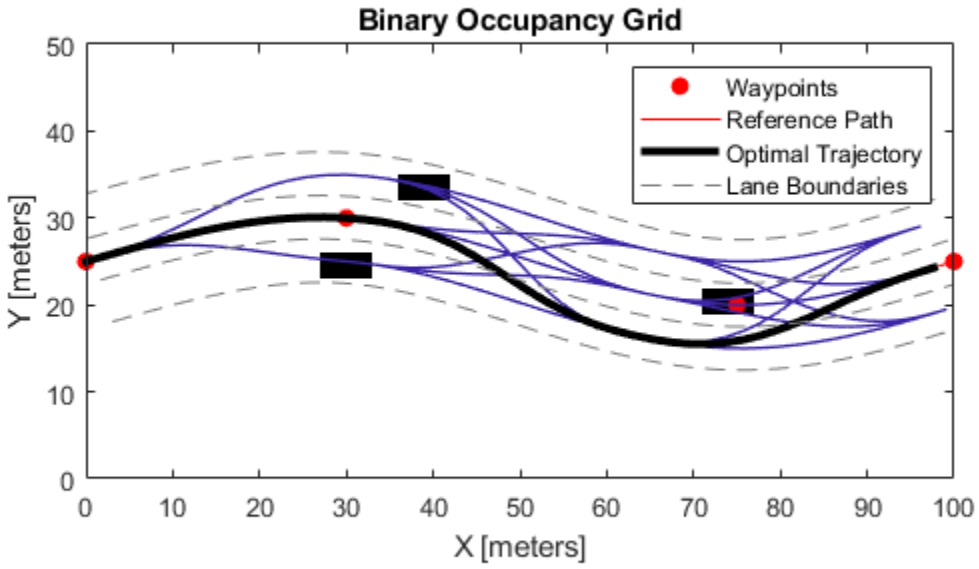
Calculate positions of lanes in Cartesian state.

```
numLaneOffsets = numel(laneOffsets);
xRefPathEnd = ceil(refPathEnd(1));
laneXY = zeros((numLaneOffsets*xRefPathEnd)+numLaneOffsets,2);
xIndex = 0;

for laneID = 1:numLaneOffsets
    for x = 1:xRefPathEnd
        laneCart = frenet2cart(planner,[x 0 0 laneOffsets(laneID) 0 0]);
```

```
        xIndex = xIndex + 1;
        laneXY(xIndex,:) = laneCart(1:2);
    end
    xIndex = xIndex + 1;
    laneXY(xIndex,:) = NaN(1,2);
end
```

Plot lane boundaries.

```
plot(laneXY(:,1),laneXY(:,2),'LineWidth',0.5,'Color',[0.5 0.5 0.5],'DisplayName','Lane Boundaries
```



Input Arguments

`planner` — Optimal trajectory planner in Frenet space

`trajectoryOptimalFrenet` object

Optimal trajectory planner in Frenet space, specified as a `trajectoryOptimalFrenet` object.

`start` — Initial Frenet state

six-element vector

Initial Frenet state, specified as a 1-by-6 vector `[s, ds/dt, d²s/dt², l, dl/ds, d²l/ds²]`.

- *s* specifies the arc length from the first point in reference path in meters.
- *ds/dt* specifies the first derivative of arc length.

- d^2s/dt^2 specifies the second derivative of arc length.
- l specifies the normal distance from the closest point in the reference path in meters.
- dl/ds specifies the first derivative of normal distance.
- d^2l/ds^2 specifies the second derivative of normal distance.

Output Arguments

`traj` — Feasible trajectory with minimum cost

n-by-7 matrix

Feasible trajectory with minimum cost, returned as an *n*-by-7 matrix of `[x, y, theta, kappa, speed, acceleration, time]`, where *n* is the number of trajectory waypoints.

- *x* and *y* specify the position in meters.
- *theta* specifies the orientation angle in radians.
- *kappa* specifies the curvature in m^{-1}.
- *speed* specifies the velocity in `m/s`.
- *acceleration* specifies the acceleration in m/s^2.
- *time* specifies the time in `s`.

`index` — Index of feasible trajectory with minimum cost

positive integer scalar

Index of feasible trajectory with minimum cost, returned as a positive integer scalar.

`cost` — Least cost of feasible trajectory

positive scalar

Least cost of feasible trajectory, returned as a positive scalar.

`flag` — Exit flag indicating solution status

`0` | `1`

Exit flag indicating the solution status, returned either as `0` or `1`.

- `0` — Optimal trajectory was found.
- `1` — No feasible trajectory exists.

When no feasible trajectory exists, the planner returns an empty trajectory.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`trajectoryOptimalFrenet` | `show`

Introduced in R2019b

show

Visualize trajectories

Syntax

```
show(planner)
show(planner,Name,Value)
axHandle = show(planner)
```

Description

`show(planner)` visualizes the reference path and trajectory from the candidates generated by the `plan` function. The trajectory is shown as a line plot. The plot also includes datatip mode, which can be used to visualize the feasibility vector and index of the trajectory from the TrajectoryList property.

`show(planner,Name,Value)` specifies additional options using one or more `Name,Value` pair arguments.

`axHandle = show(planner)` returns the axes handle of the figure used to plot the trajectory.

Examples

Optimal Trajectory Planning in Frenet Space

This example shows how to plan an optimal trajectory using a `trajectoryOptimalFrenet` object.

Create and Assign Map to State Validator

Create a state validator object for collision checking.

```
stateValidator = validatorOccupancyMap;
```

Create an obstacle grid map.

```
grid = zeros(50,100);
grid(24:26,48:53) = 1;
```

Create a `binaryOccupancyMap` with the grid map.

```
map = binaryOccupancyMap(grid);
```

Assign the map and the state bounds to the state validator.

```
stateValidator.Map = map;
stateValidator.StateSpace.StateBounds(1:2,:) = [map.XWorldLimits; map.YWorldLimits];
```

Plan and Visualize Trajectory

Create a reference path for the planner to follow.

```
refPath = [0,25;100,25];
```

Initialize the planner object with the reference path, and the state validator.

```
planner = trajectoryOptimalFrenet(refPath,stateValidator);
```

Assign longitudinal terminal state, lateral deviation, and maximum acceleration values.

```
planner.TerminalStates.Longitudinal = 100;
planner.TerminalStates.Lateral = -10:5:10;
planner.FeasibilityParameters.MaxAcceleration = 10;
```

Specify the deviation offset value close to the left lateral terminal state to prioritize left lane changes.

```
planner.DeviationOffset = 5;
```

Trajectory Planning

Initial cartesian state of vehicle.

```
initCartState = [0 25 pi/9 0 0 0];
```

Convert cartesian state of vehicle to Frenet state.

```
initFrenetState = cart2frenet(planner,initCartState);
```

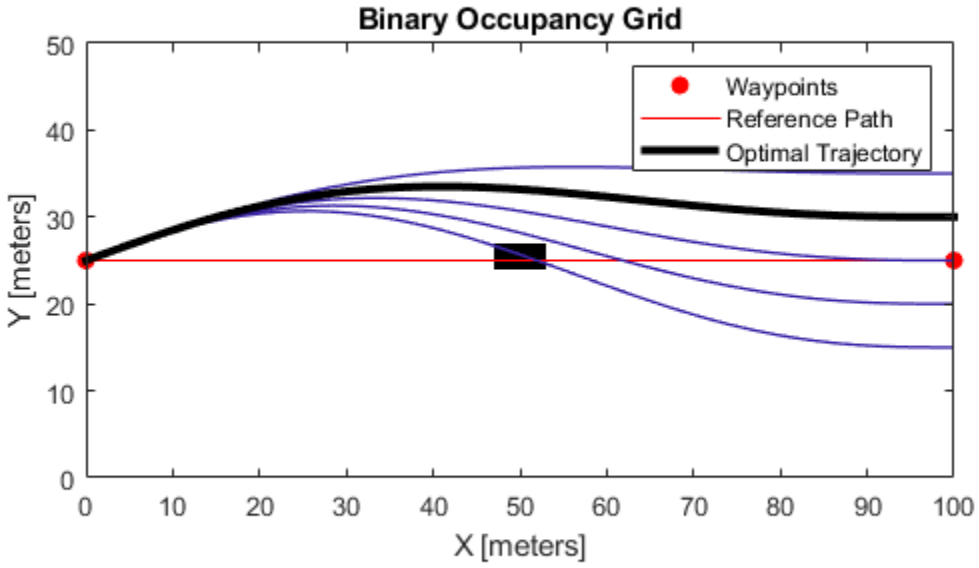
Plan a trajectory from initial Frenet state.

```
plan(planner,initFrenetState);
```

Trajectory Visualization

Visualize the map and the trajectories.

```
show(map)
hold on
show(planner,'Trajectory','all')
```



Partitioning Longitudinal Terminal States in Trajectory Generation

This example shows how to partition the longitudinal terminal states in optimal trajectory planning using a `trajectoryOptimalFrenet` object.

Create and Assign Map to State Validator

Create a state validator object for collision checking.

```
stateValidator = validatorOccupancyMap;
```

Create an obstacle grid map.

```
grid = zeros(50,100);
grid(25:27,28:33) = 1;
grid(16:18,37:42) = 1;
grid(29:31,72:77) = 1;
```

Create a `binaryOccupancyMap` with the grid map.

```
map = binaryOccupancyMap(grid);
```

Assign the map and the state bounds to the state validator.

```
stateValidator.Map = map;
stateValidator.StateSpace.StateBounds(1:2,:) = [map.XWorldLimits; map.YWorldLimits];
```

Plan and Visualize Trajectory

Create a reference path for the planner to follow.

```
refPath = [0,25;30,30;75,20;100,25];
```

Initialize the planner object with the reference path, and the state validator.

```
planner = trajectoryOptimalFrenet(refPath,stateValidator);
```

Assign longitudinal terminal state, lateral deviation, and maximum acceleration values.

```
planner.TerminalStates.Longitudinal = 100;
planner.TerminalStates.Lateral = -5:5:5;
planner.FeasibilityParameters.MaxAcceleration = 10;
```

Assign the number of partitions for the longitudinal terminal state.

```
planner.NumSegments = 3;
```

Trajectory Planning

Initial Frenet state of vehicle.

```
initFrenetState = zeros(1,6);
```

Plan a trajectory from initial Frenet state.

```
plan(planner,initFrenetState);
```

Trajectory Visualization

Visualize the map and the trajectories.

```
show(map)
hold on
show(planner,'Trajectory','all')
hold on
```

Generate Lane Boundaries

Calculate end of reference path as Frenet state.

```
refPathEnd = cart2frenet(planner,[planner.Waypoints(end,:) 0 0 0 0]);
```

Calculate lane offsets on both sides of the lateral terminal states with half lane width value.

```
laneOffsets = unique([planner.TerminalStates.Lateral+2.5 planner.TerminalStates.Lateral-2.5]);
```

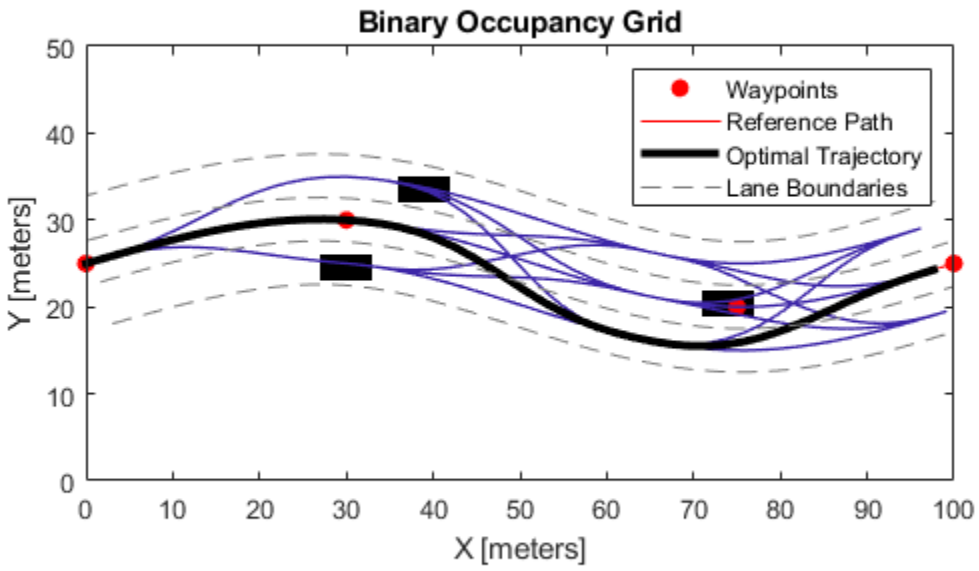
Calculate positions of lanes in Cartesian state.

```
numLaneOffsets = numel(laneOffsets);
xRefPathEnd = ceil(refPathEnd(1));
laneXY = zeros((numLaneOffsets*xRefPathEnd)+numLaneOffsets,2);
xIndex = 0;

for laneID = 1:numLaneOffsets
    for x = 1:xRefPathEnd
        laneCart = frenet2cart(planner,[x 0 0 laneOffsets(laneID) 0 0]);
```

```
            xIndex = xIndex + 1;
            laneXY(xIndex,:) = laneCart(1:2);
        end
        xIndex = xIndex + 1;
        laneXY(xIndex,:) = NaN(1,2);
    end
```

Plot lane boundaries.

```
plot(laneXY(:,1),laneXY(:,2),'LineWidth',0.5,'Color',[0.5 0.5 0.5],'DisplayName','Lane Boundaries
```



Input Arguments

`planner` — Optimal trajectory planner in Frenet space

`trajectoryOptimalFrenet` object

Optimal trajectory planner in Frenet space, specified as a `trajectoryOptimalFrenet` object.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'Trajectory','all'`

`Parent` — Axes to plot trajectory
`Axes` object | `UIAxes` object

Axes to plot trajectory, specified as the comma-separated pair consisting of `'Parent'` and either an `Axes` or `UIAxes` object. See `axes` or `uiaxes`.

`Trajectory` — Trajectory display option
`'optimal'` (default) | `'all'`

Trajectory display option, specified as the comma-separated pair consisting of `'Trajectory'` and either `'optimal'` or `'all'`.

`ReferencePath` — Reference path display option
`'on'` (default) | `'off'`

Reference path display option, specified as the comma-separated pair consisting of `'ReferencePath'` and either `'on'` or `'off'`.

`TrajectoryColor` — Trajectory color display option
`'velocity'` (default) | `'acceleration'` | `'cost'` | `'none'`

Trajectory color display option, specified as the comma-separated pair consisting of `'TrajectoryColor'` and one of the following:

- `'acceleration'`
- `'cost'`
- `'velocity'`
- `'none'`

Set this property to display the specified trajectory as a color-gradient along the specified path.

Output Arguments

`axHandle` — Axes handle used to plot trajectory
`Axes` object | `UIAxes` object

Axes handle used to plot trajectory, returned as either an `axes`, or `uiaxes` object.

See Also
`trajectoryOptimalFrenet` | `plan`

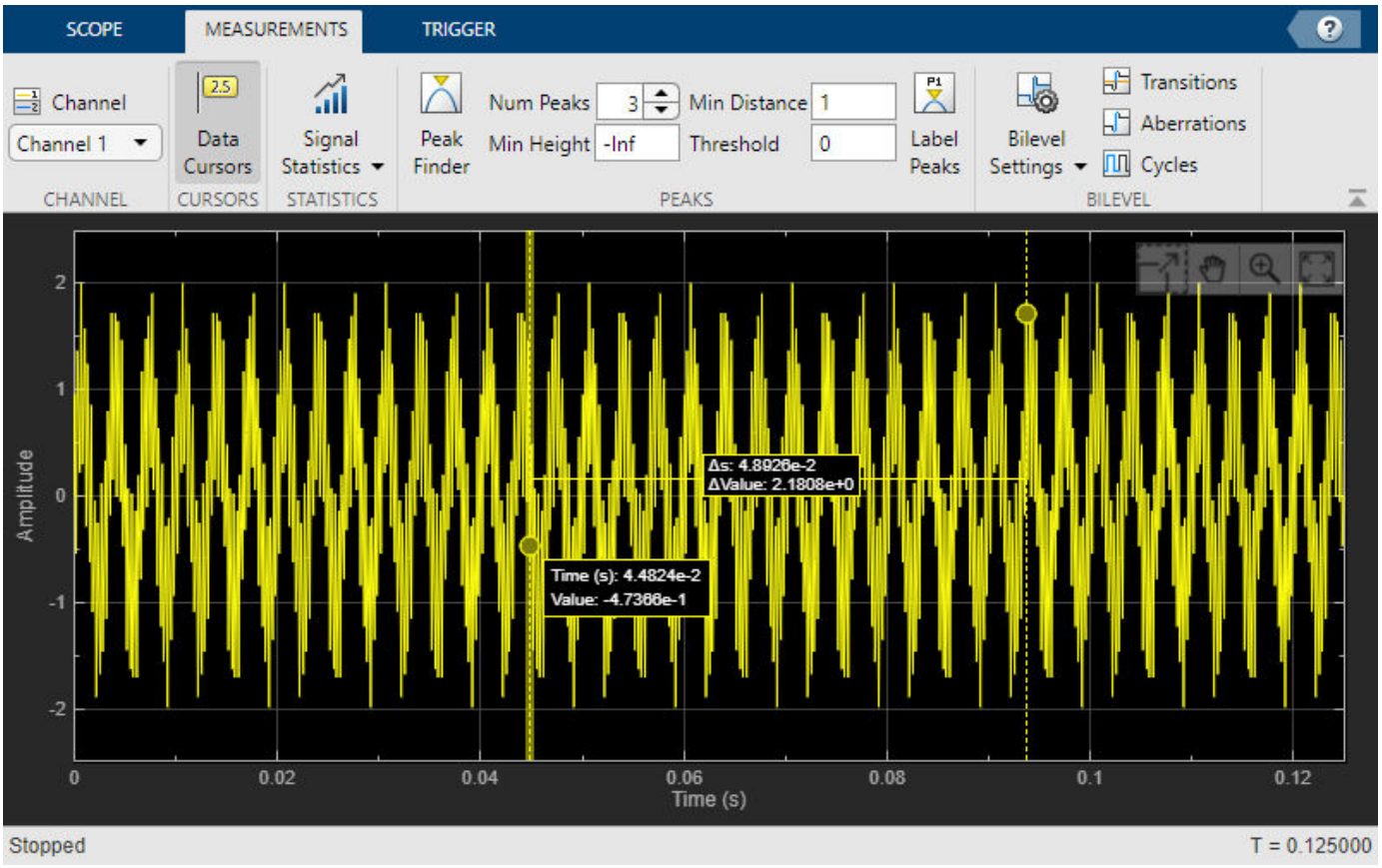
Introduced in R2019b

timescope

Display time-domain signals

Description

The `timescope` object displays signals in the time domain.



Scope features:

- “Data Cursors” — Measure signal values using vertical and horizontal cursors.
- “Signal Statistics” — Display the maximum, minimum, peak-to-peak difference, mean, median, and RMS values of a selected signal.
- “Peak Finder” — Find maxima, showing the *x*-axis values at which they occur.
- “Bilevel Measurements” — Measure transitions, overshoots, undershoots, and cycles.
- “Triggers” — Set triggers to sync repeating signals and pause the display when events occur.

Use “Object Functions” on page 2-1321 to show, hide, and determine visibility of the scope window.

Creation

Syntax

```
scope = timescope
scope = timescope(Name,Value)
```

Description

`scope = timescope` returns a `timescope` object, `scope`. This object displays real- and complex-valued floating and fixed-point signals in the time domain.

`scope = timescope(Name,Value)` returns a `timescope` object with properties set to the specified value. Specify properties and their values in quotes, separated by commas. You can specify name-value pair arguments in any order.

Properties

Most properties can be changed from the `timescope` UI.

Frequently Used

`SampleRate` — Sample rate of inputs

1 (default) | finite numeric scalar | vector

Sampling rate of the input signal, in hertz, specified as a finite numeric scalar or vector of scalars.

The inverse of the sample rate determines the *x*-axis (time axis) spacing between points in the displayed signal. When the value of `NumInputPorts` is greater than 1 and the sample rate is scalar, the object uses the same sample rate for all inputs. To specify different sample rates for each input, use a vector.

You can only set this property when creating the object or after calling `release`.

Scope Window Use

On the **Scope** tab, click **Settings**. Under **Data and Axes**, set **Sample Rate**.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`TimeSpanSource` — Source of time span

`'auto'` (default) | `'property'`

Source of the time span for frame-based input signals, specified as one of the following:

- `'property'` – The object derives the *x*-axis limits from the `TimeDisplayOffset` and `TimeSpan` properties.
- `'auto'` – The *x*-axis limits are derived from the `TimeDisplayOffset` property, `SampleRate` property, and the number of rows in each input signal (*FrameSize* in the equations below). The limits are calculated as:
  - Minimum time-axis limit = `TimeDisplayOffset`
  - Maximum time-axis limit = `TimeDisplayOffset + max(1/SampleRate.*`*`FrameSize`*`)`

Dependency

When you set the `TimeSpan` property, `TimeSpanSource` is automatically set to `'property'`.

Scope Window Use

On the **Scope** tab, click **Settings**. Under **Data and Axes**, set **Time Span**.

Data Types: `char` | `string`

`TimeSpan` — Time span

`10` (default) | positive scalar

Time span, in seconds, specified as a positive, numeric scalar value. The time-axis limits are calculated as:

- Minimum time-axis limit = `TimeDisplayOffset`
- Maximum time-axis limit = `TimeDisplayOffset` + `TimeSpan`

Dependencies

To enable this property, set `TimeSpanSource` to `'property'`.

Scope Window Use

On the **Scope** tab, click **Settings**. Under **Data and Axes**, edit **Time Span**.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`TimeSpanOverrunAction` — Data overrun behavior

`'scroll'` (default) | `'wrap'`

Specify how the scope displays new data beyond the visible time span as either:

- `'scroll'` — In this mode, the scope scrolls old data to the left to make room for new data on the right of the scope display. This mode is beneficial for debugging and monitoring time-varying signals.
- `'wrap'` — In this mode, the scope adds data to the left of the plot after overrunning the right of the plot.

Scope Window Use

On the **Scope** tab, click **Settings**. Under **Data and Axes**, set **Overrun Action**.

Data Types: `char` | `string`

`PlotType` — Type of plot

`'line'` (default) | `'stairs'`

Type of plot, specified as either:

- `'line'` — Line graph, similar to the `line` or `plot` function.
- `'stairs'` — Stair-step graph, similar to the `stairs` function. Stair-step graphs are useful for drawing time history graphs of digitally sampled data.

Scope Window Use

On the **Scope** tab, click **Settings**. Under **Data and Axes**, set **Plot Type**.

Data Types: `char` | `string`

`AxesScaling` — Axes scaling mode

`'onceatstop'` (default) | `'auto'` | `'manual'` | `'updates'`

When this property is set to:

- `'onceatstop'` –– The limits are updated once at the end of the simulation (when `release` is called).
- `'auto'` –– The scope attempts to always keep the data in the display while minimizing the number of updates to the axes limits.
- `'manual'` –– The scope takes no action unless specified by the user.
- `'updates'` –– The scope scales the axes once and only once after 100 updates to the visualization.

You can set this property only when creating the object.

Data Types: `char` | `string`

Advanced

`LayoutDimensions` — Display layout grid dimensions

`[1,1]` (default) | `[numberOfRows, numberOfColumns]`

Specify the layout grid dimensions as a two-element vector: `[numberOfRows,numberOfColumns]`. The grid can have a maximum of 4 rows and 4 columns.

If you create a grid of multiple axes, to modify the settings of individual axes, use the `ActiveDisplay`.

Example: `scope.LayoutDimensions = [2,4]`

Scope Window Use

On the **Scope** tab, click **Display Grid** () and select a specific number of rows and columns from the grid.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`TimeUnits` — Units of x-axis

`'seconds'` (default) | `'none'` | `'metric'`

Specify the units used to describe the x-axis (time axis). You can select one of the following options:

- `'seconds'` —The scope always displays the units on the x-axis as seconds. The scope shows the word `Time(s)` on the x-axis.
- `'none'` — The scope does not display any units on the x-axis. The scope only shows the word `Time` on the x-axis.
- `'metric'` — The scope displays the units on the x-axis as `Time (s)` changing the units to day, weeks, months, or years as you plot more data points.

Scope Window Use

On the **Scope** tab, click **Settings**. Under **Data and Axes**, set **Time Units**.

Data Types: `char` | `string`

`TimeDisplayOffset` — Offset *x*-axis limits

0 (default) | scalar | vector

Specify, in seconds, how far to move the data on the *x*-axis. The signal value does not change, only the limits displayed on the *x*-axis change.

If you specify this property as a scalar, then that value is the time display offset for all channels. If you specify this property as a vector, each input channel can be a different time display offset

Scope Window Use

On the **Scope** tab, click **Settings**. Under **Data and Axes**, set **Time Offset**.

`TimeAxisLabels` — Time-axis labels

`'all'` (default) | `'bottom` | `'none'`

Time-axis labels, specified as:

- `'all'` — Time-axis labels appear in all displays.
- `'bottom` — Time-axis labels appear in the bottom display of each column.
- `'none'` — No labels appear in any display.

Scope Window Use

On the **Scope** tab, click **Settings**. Under **Data and Axes**, set **Time Labels**.

Data Types: `char` | `string`

`MaximizeAxes` — Maximize axes control

`'auto'` (default) | `'on'` | `'off'`

Specify whether to display the scope in the maximized-axes mode. In this mode, the axes are expanded to fit into the entire display. To conserve space, labels do not appear in each display. Instead, the tick-marks and their values appear on top of the plotted data. You can select one of the following options:

- `'auto'` — The axes appear maximized in all displays only if the `Title` and `YLabel` properties are empty for every display. If you enter any value in any display for either of these properties, the axes are not maximized.
- `'on'` — The axes appear maximized in all displays. Any values entered into the `Title` and `YLabel` properties are hidden.
- `'off'` — None of the axes appear maximized.

Scope Window Use

On the scope window, click on  to maximize axes, hiding all labels and insetting the axes values.

Data Types: `char` | `string`

`BufferLength` — Buffer length

`50000` (default) | positive integer

Specify the length of the buffer used for each input signal as a positive integer.

You can set this property only when creating the object.

Scope Window Use

On the **Scope** tab, click **Settings**. Under **Data and Axes**, set **Buffer Length**.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Visualization

`Name` — Window name

`'Time Scope'` (default) | character vector | string scalar

Specify the name of the scope as a character vector or string scalar. This name appears as the title of the scope's figure window. To specify a title of a scope plot, use the `Title` property.

Data Types: `char` | `string`

`Position` — Window position

screen center (default) | `[left bottom width height]`

Scope window position in pixels, specified by the size and location of the scope window as a four-element vector of the form `[left bottom width height]`. You can place the scope window in a specific position on your screen by modifying the values of this property.

By default, the window appears in the center of your screen with a width of `800` pixels and height of `500` pixels. The exact values of the position depend on your screen resolution.

`ChannelNames` — Channel names

`{''}` (default) | cell array of character vectors

Specify the input channel names as a cell array of character vectors. The channel names appear in the legend, and on the **Measurements** tab under **Select Channel**. If you do not specify names, the channels are labeled as `Channel 1`, `Channel 2`, etc.

Dependency

To enable this property, set `ShowLegend` to `true`.

Data Types: `char`

`ActiveDisplay` — Active display for setting properties

`1` (default) | integer

Active display used to set properties, specified by the integer display number. The number of a display corresponds to the display's row-wise placement index. Setting this property controls which display is used for the following properties: `YLimits`, `YLabel`, `ShowLegend`, `ShowGrid`, `Title`, and `PlotAsMagnitudePhase`.

Scope Window Use

On the **Scope** tab, click **Settings**. Under **Display and Labels**, set **Active Display**.

`Title` — Display title

`''` (default) | character vector | string scalar

Specify the display title as a character vector or a string scalar.

Dependency

When you set this property, `ActiveDisplay` controls the display that is updated.

Scope Window Use

On the **Scope** tab, click **Settings**. Under **Display and Labels**, set **Title**.

Data Types: `char` | `string`

`YLabel` — *y*-axis label

`'Amplitude'` (default) | character vector | string scalar

Specify the text for the scope to display to the left of the *y*-axis.

Dependencies

This property applies only when `PlotAsMagnitudePhase` is `false`. When `PlotAsMagnitudePhase` is `true`, the two *y*-axis labels are read-only values `"Magnitude"` and `"Phase"`, for the magnitude plot and the phase plot, respectively.

When you set this property, `ActiveDisplay` controls the display that is updated.

Scope Window Use

On the **Scope** tab, click **Settings**. Under **Display and Labels**, set **YLabel**.

Data Types: `char` | `string`

`YLimits` — *y*-axis limits

`[-10,10]` (default) | `[ymin, ymax]`

Specify the *y*-axis limits as a two-element numeric vector, `[ymin, ymax]`.

- If `PlotAsMagnitudePhase` is `false`, the default is `[-10,10]`.
- If `PlotAsMagnitudePhase` is `true`, the default is `[0,10]`. This property specifies the *y*-axis limits of only the magnitude plot. The *y*-axis limits of the phase plot are always `[-180,180]`

Dependency

When you set this property, `ActiveDisplay` controls the display that is updated.

Scope Window Use

On the **Scope** tab, click **Settings**. Under **Display and Labels**, set **Y-Axis Limits**.

`ShowLegend` — Show legend

`false` (default) | `true`

To show a legend with the input names, set this property to `true`.

From the legend, you can control which signals are visible. In the scope legend, click a signal name to hide the signal in the scope. To show the signal, click the signal name again.

Scope Window Use

On the **Scope** tab, click **Settings**. Under **Display and Labels**, select **Show Legend**.

Data Types: `logical`

`ShowGrid` — Grid visibility
`true` (default) | `false`

Set this property to `true` to show grid lines on the plot.

Scope Window Use

On the **Scope** tab, click **Settings**. Under **Display and Labels**, select **Show Grid**.

`PlotAsMagnitudePhase` — Plot signal as magnitude and phase
`false` (default) | `true`

Plot signal as magnitude and phased, specified as either:

- `true` – The scope plots the magnitude and phase of the input signal on two separate axes within the same active display.
- `false` – The scope plots the real and imaginary parts of the input signal on two separate axes within the same active display.

This property is useful for complex-valued input signals. Turning on this property affects the phase for real-valued input signals. When the amplitude of the input signal is nonnegative, the phase is 0 degrees. When the amplitude of the input signal is negative, the phase is 180 degrees.

Scope Window Use

On the **Scope** tab, click **Settings**. Under **Display and Labels**, select **Magnitude Phase Plot**.

Object Functions

To use an object function, specify the object as the first input argument.

hide	Hide scope window
show	Display scope window
isVisible	Determine visibility of scope
generateScript	Generate MATLAB script to create scope with current settings
step	Run System object algorithm
release	Release resources and allow changes to System object property values and input characteristics
reset	Reset internal states of System object

If you want to restart the simulation from the beginning, call `reset` to clear the scope window displays. Do not call `reset` after calling `release`.

Examples

View Sine Wave on Time Scope

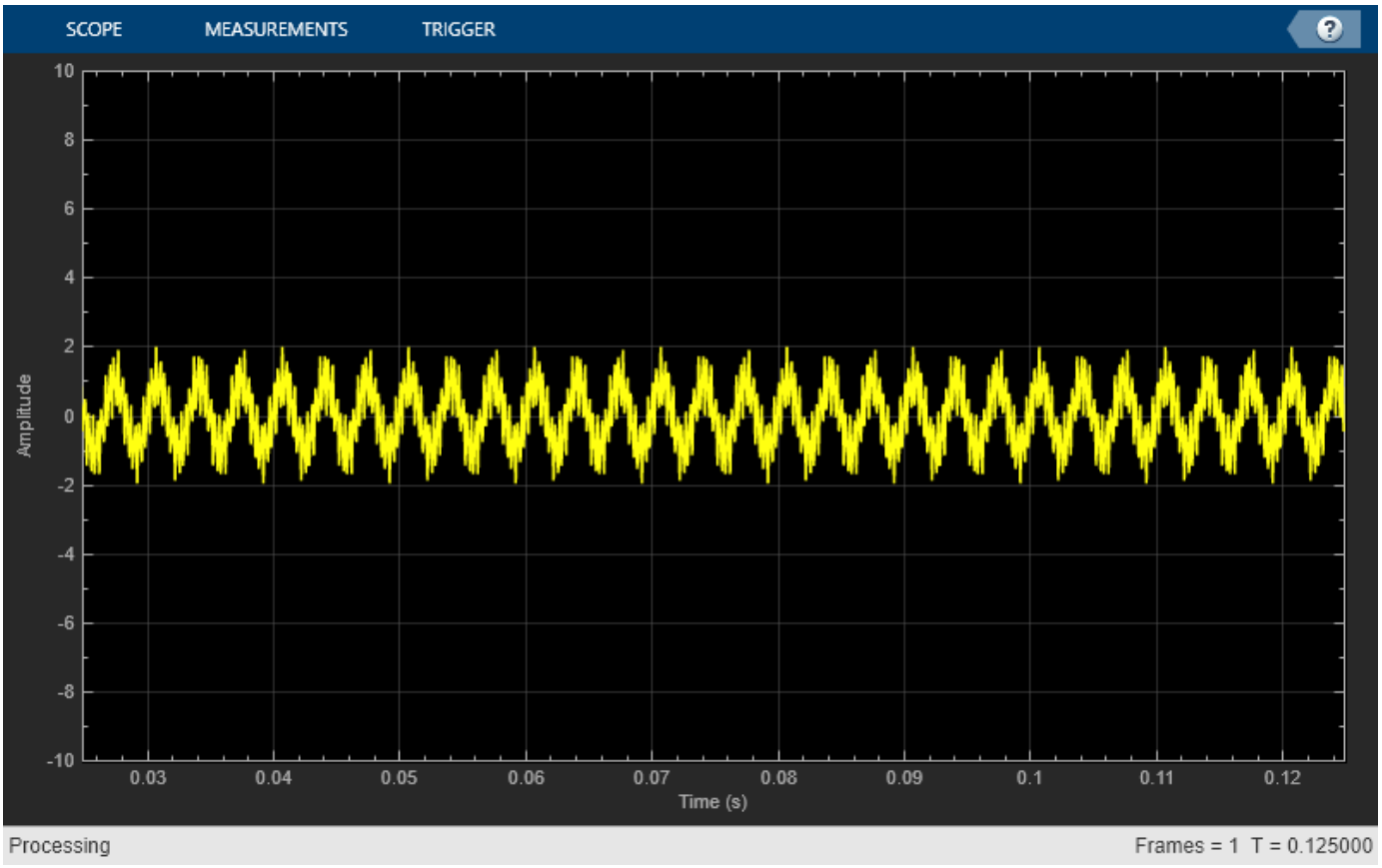
Create a time-domain sinusoidal signal. Display the signal by calling the time scope object.

Create a sinusoidal signal with two tones, one at 0.3 kHz and the other at 3 kHz.

```
t = (0:1000)'/8e3;
xin = sin(2*pi*0.3e3*t)+sin(2*pi*3e3*t);
```

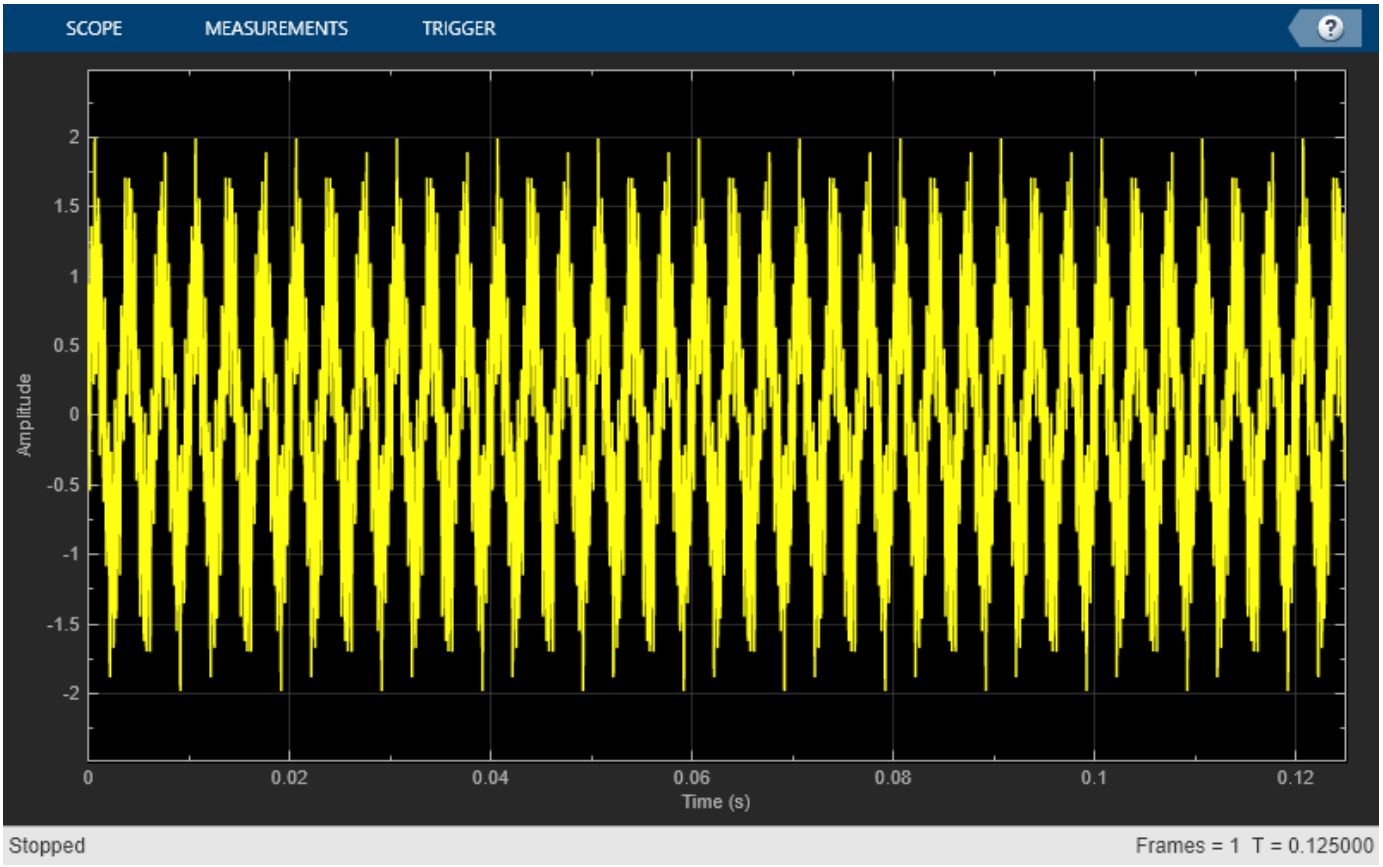
Create a `timescope` object and view the sinusoidal signal by calling the time scope object `scope`.

```
scope = timescope('SampleRate', 8e3,...
    'TimeSpanSource', 'property', ...
    'TimeSpan', 0.1);
scope(xin)
```



Run `release` to allow changes to property values and input characteristics. The scope automatically scales the axes.

```
release(scope);
```

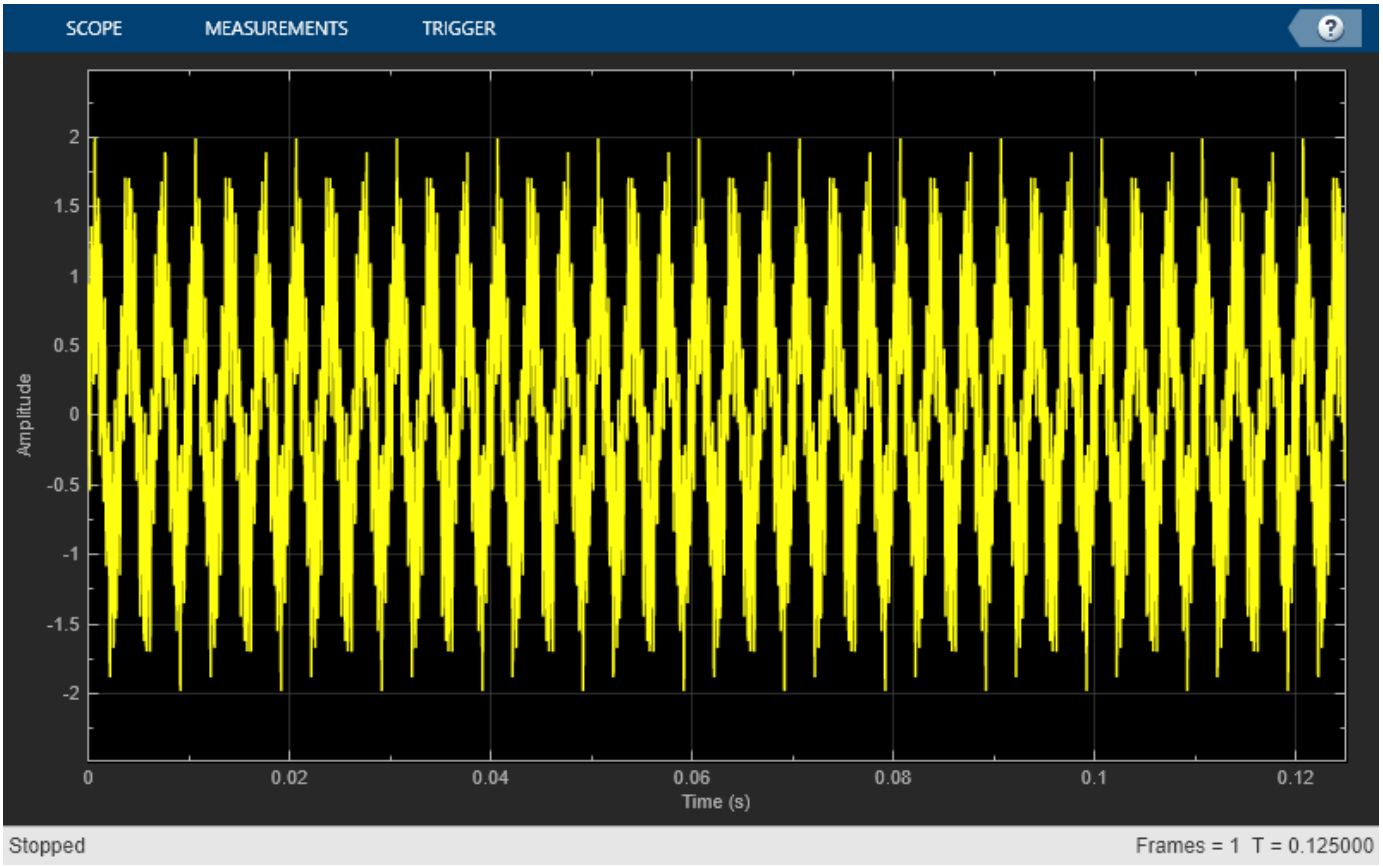


Hide the scope window.

```
if(isVisible(scope))
    hide(scope)
end
```

Show the scope window.

```
if(~isVisible(scope))
    show(scope)
end
```



Use Bilevel Measurements Panel with Clock Input Signal

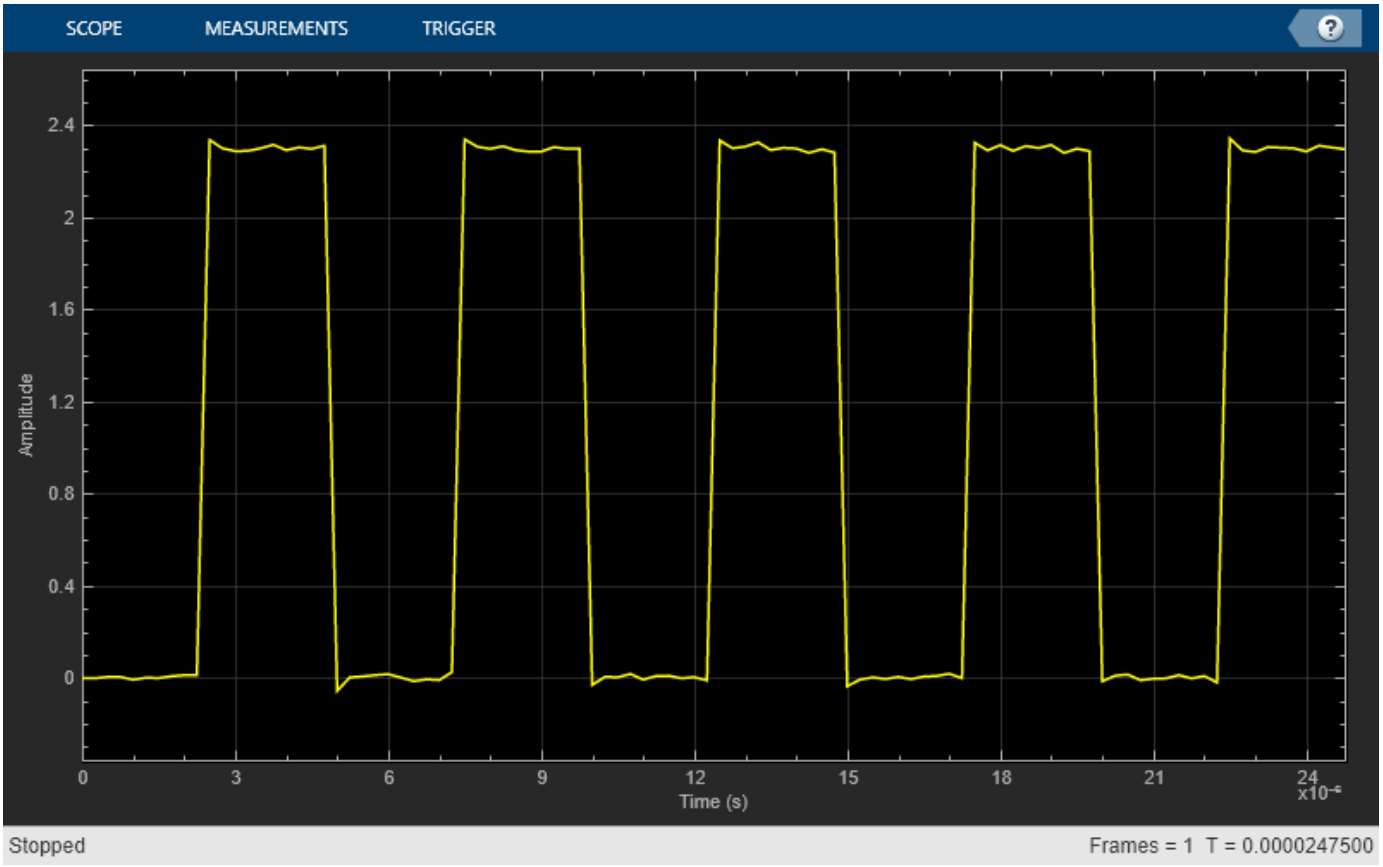
Create and Display Clock Input Signal

Load the clock data, `x` and `t`. Find the sample time, `ts`.

```
load clockex
ts = t(2)-t(1);
```

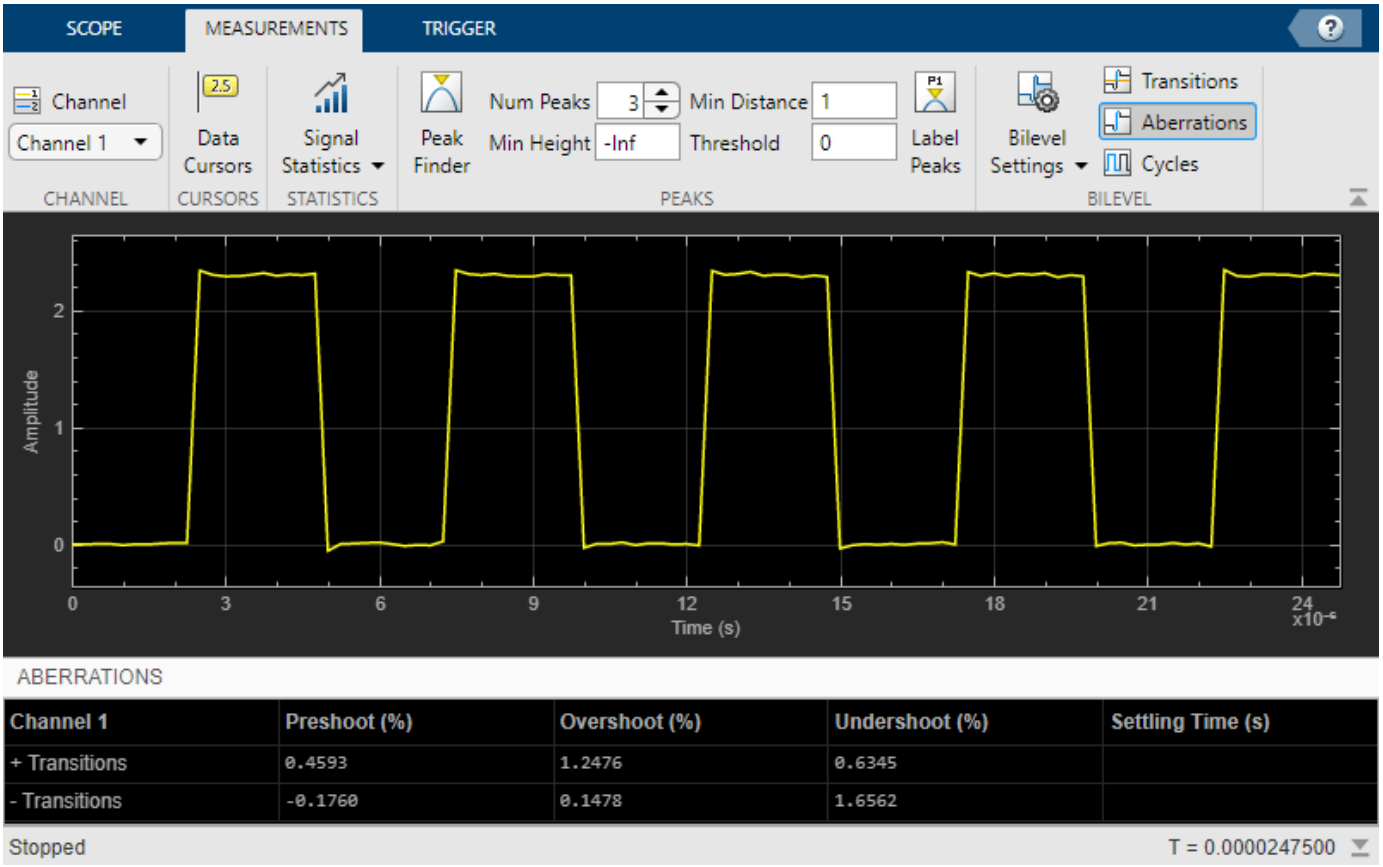
Create a `timescope` object and call the object to display the signal. To autoscale the axes and enable changes to property values and input characteristics, call `release`.

```
scope = timescope('SampleRate',1/ts,'TimeSpanSource','Auto');
scope(x);
release(scope);
```



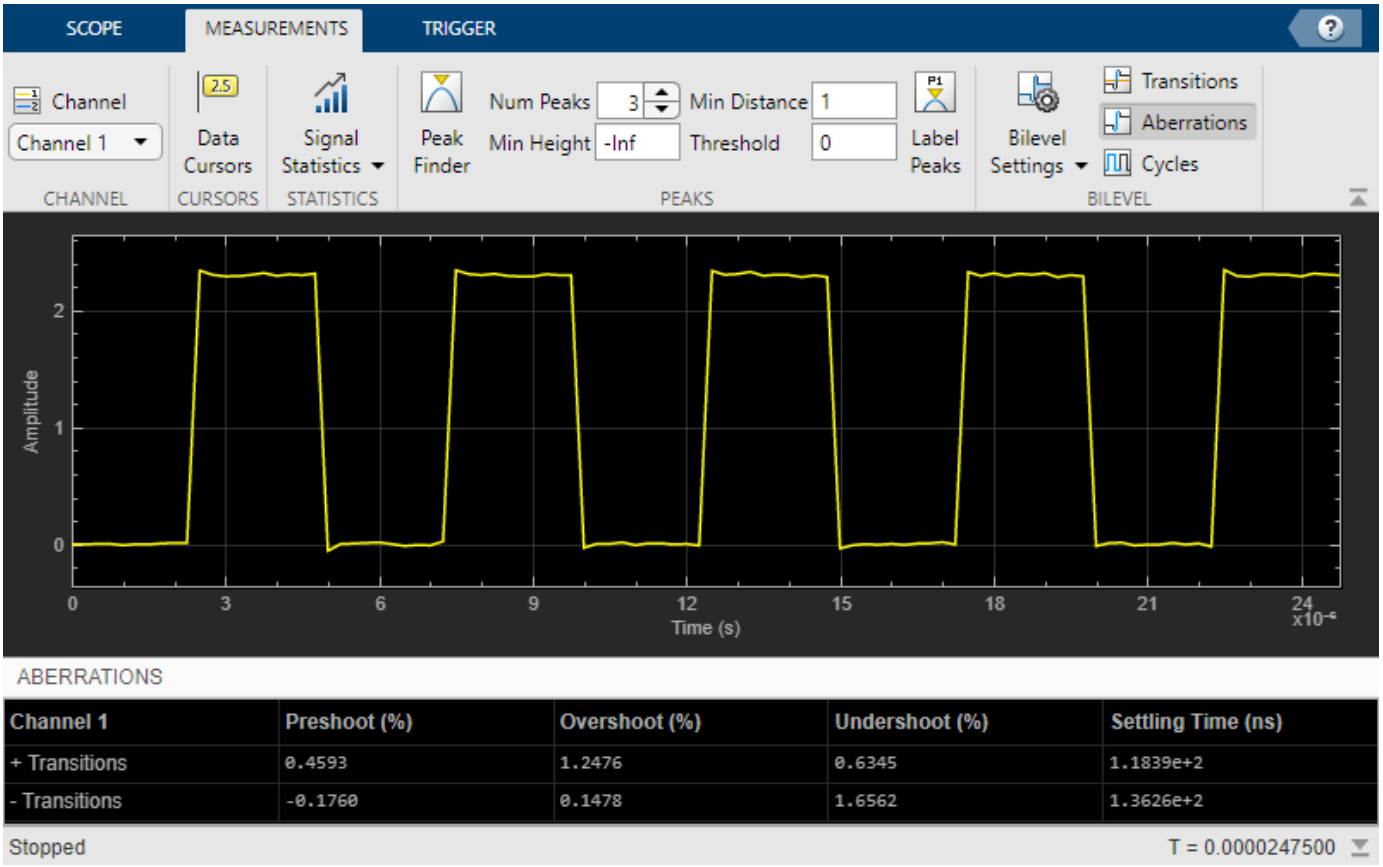
Use Bilevel Measurements Panel to Find Settling Time

1. From the **Measurements** tab, select **Aberrations**.



Initially, the Time Scope does not display the **Settling Time** measurement. This absence occurs because the default value of the **Settle Seek** parameter is longer than the entire simulation duration.

2. In the **Bilevel Settings > Settle Seek** box, enter `2e-6` and press **Enter**.

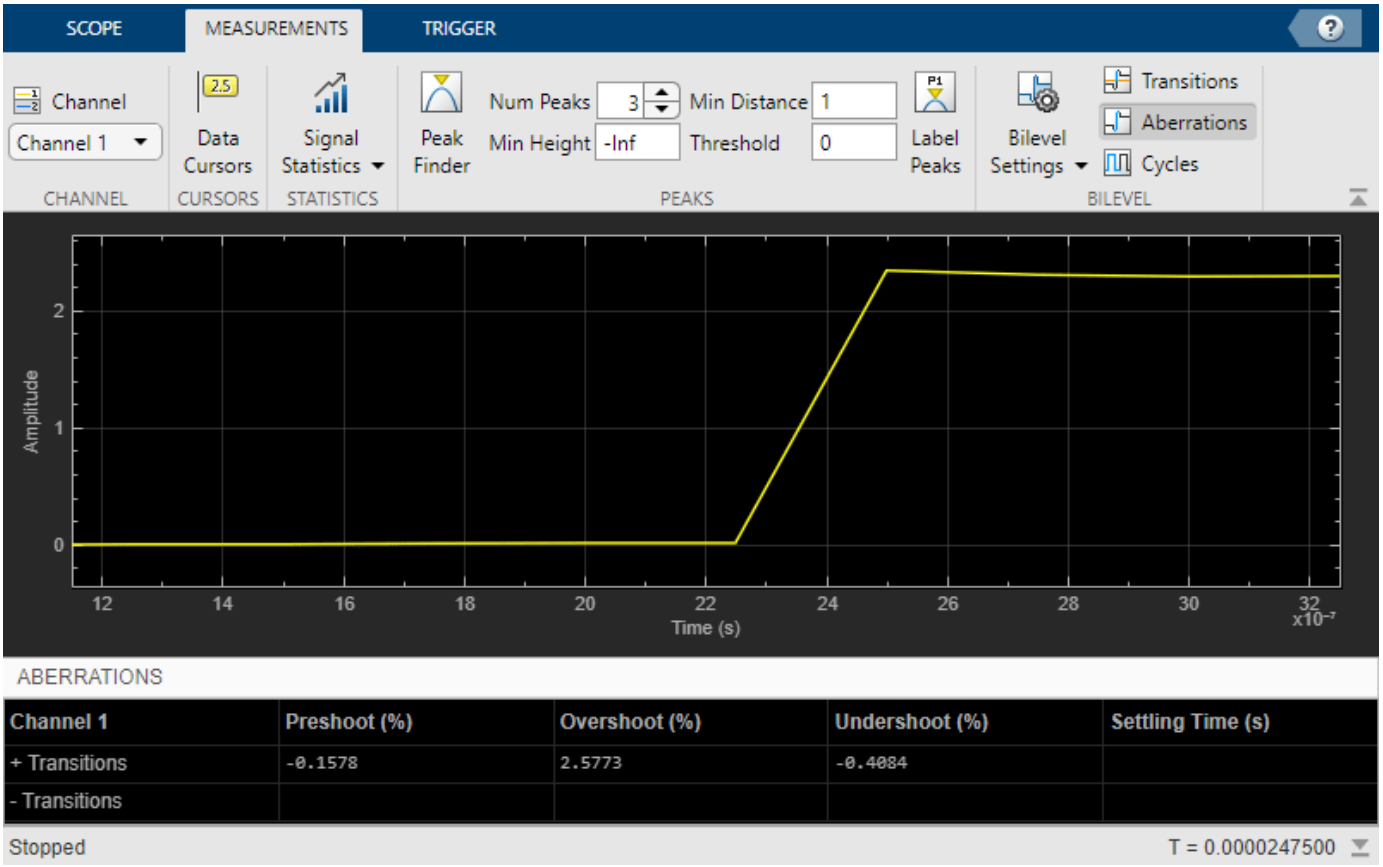


Time Scope now displays a rising edge **Settling Time** value of `118.392` ns.

This settling time value is actually the statistical average of the settling times for all five rising edges. To show the settling time for only one rising edge, you can zoom in on that transition.

3. Hover over the upper right corner of the scope axes, and click the zoom button.

4. Click and drag to zoom in on one of the transitions.



Time Scope updates the rising edge **Settling Time** value to reflect the new time window.

Visualize Multiple Inputs with Different Sample Rates

This example shows how to visualize multiple inputs with different sample rates and plot the signals on multiple axes.

Generate three different sine waves and plot them on the `timescope`.

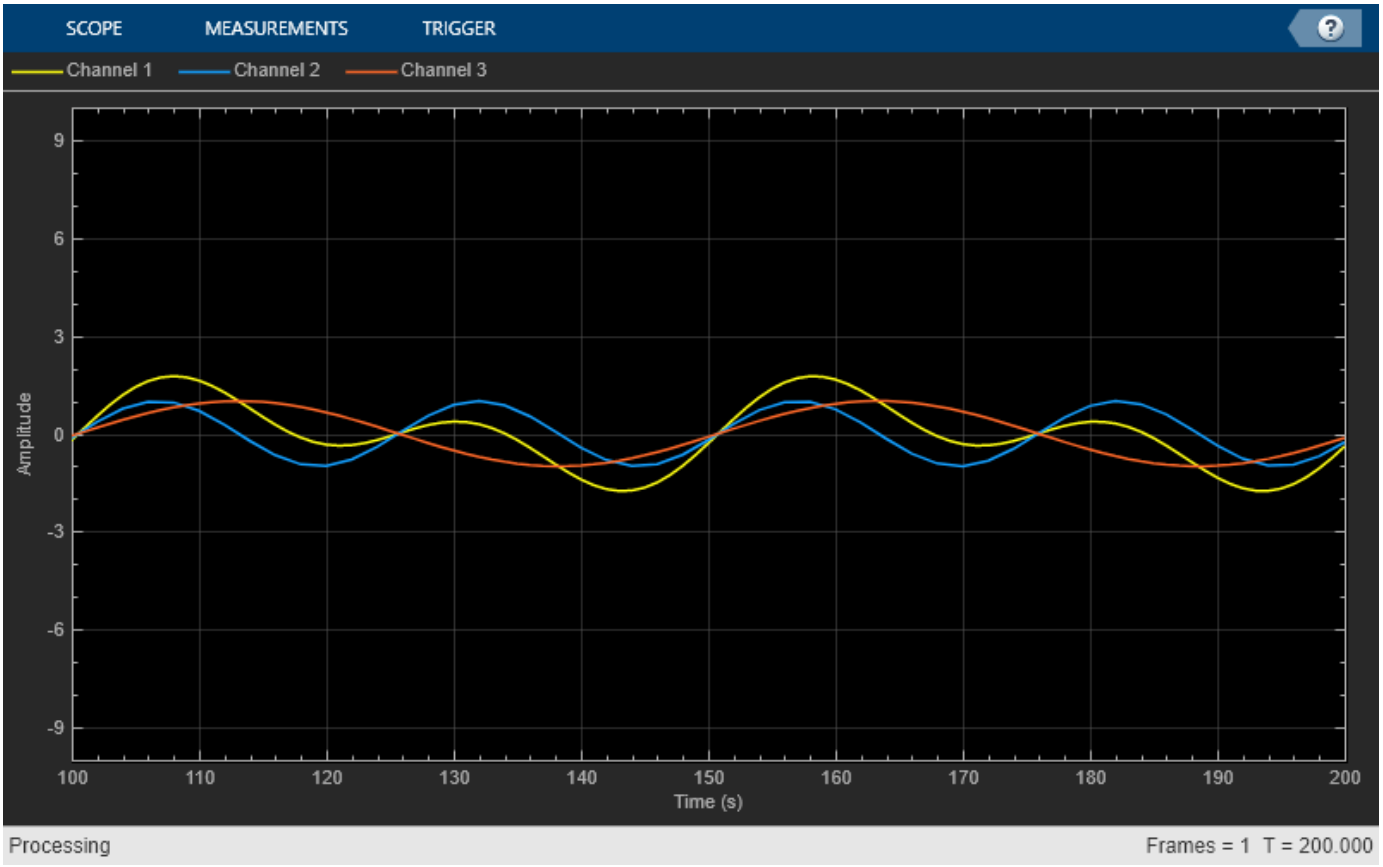
```
freq = 1/500;
t    = (0:100)'/freq;
t2   = (0:0.5:100)'/freq;
xin1 = sin(1/2*t);
xin2 = sin(1/4*t2);
xin  = sin(1/2*t2)+sin(1/4*t2);

scope = timescope('SampleRate', [freq freq/2 freq], ...
    'TimeSpanSource', 'property', ...
    'TimeSpan', 0.1,...
    'LayoutDimensions',[2,1]);
scope(xin, xin1, xin2)

release(scope)
```

Use Multiple Axes on Scope

This example show how to add titles, set *y*-axis limits, and modify properties when you have multiple axes on your `timescope` object.

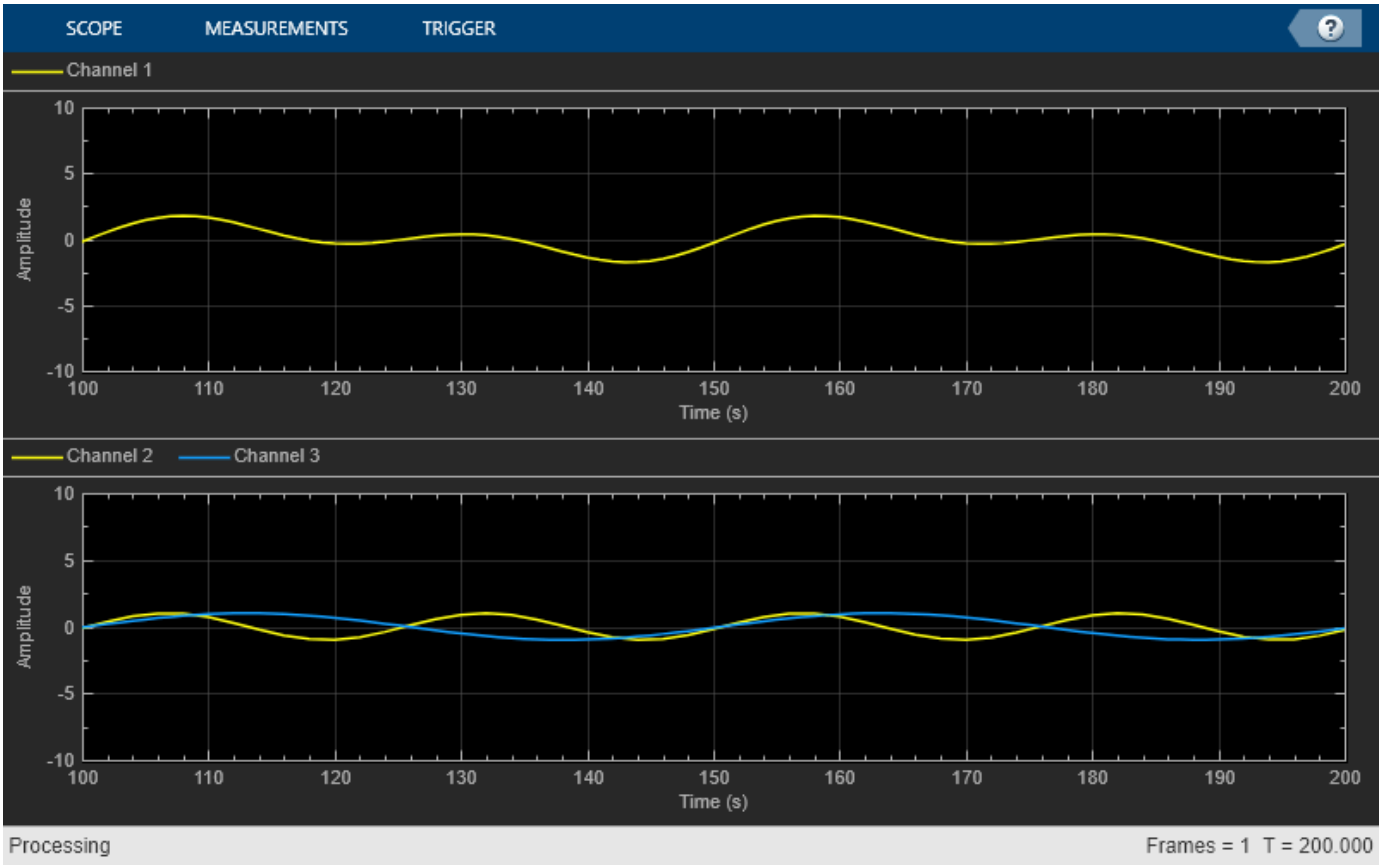
Use the `timescope` to visualize three sine waves with two different sample rates.

```
freq = 1;
t    = (0:100)'/freq;
t2   = (0:0.5:100)'/freq;
xin1 = sin(1/2*t);
xin2 = sin(1/4*t2);
xin  = sin(1/2*t2)+sin(1/4*t2);

scope = timescope('SampleRate', [freq freq/2 freq], ...
     'TimeSpanSource', 'property', ...
    'TimeSpan', 100);
scope(xin, xin1, xin2)
```



Change the layout to add a second axis. The second and third inputs automatically move to the new second axis.

```
scope.LayoutDimensions = [2,1];
```

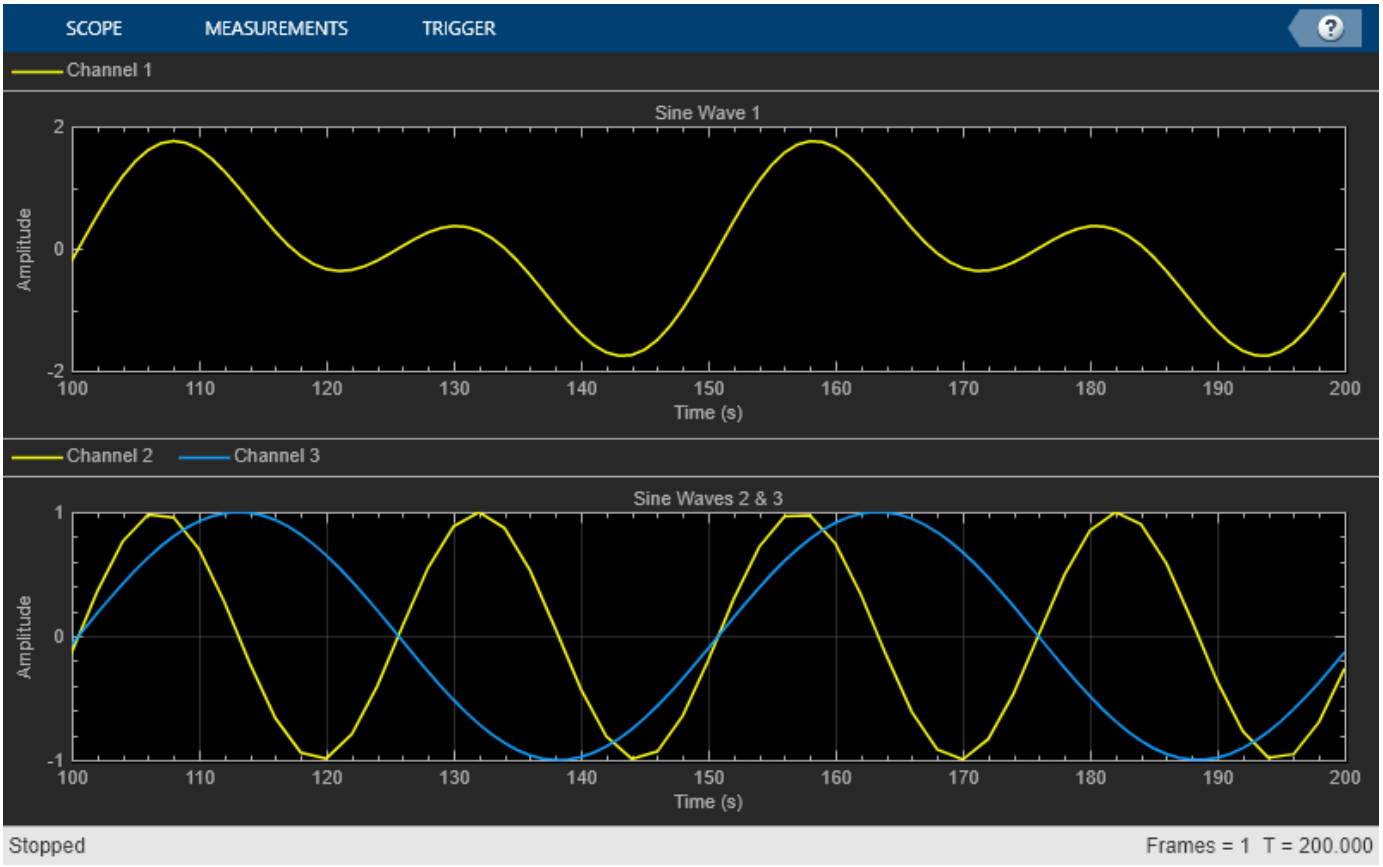


Modify the settings for the first axis by specifying the `ActiveDisplay` property to `1`, then changing some properties for that axis.

```
scope.ActiveDisplay = 1;
scope.ShowGrid = false;
scope.Title = "Sine Wave 1";
scope.YLimits = [-2,2];
```

Repeat this process to modify the second axis.

```
scope.ActiveDisplay = 2;
scope.Title = "Sine Waves 2 & 3";
scope.YLimits = [-1,1];
release(scope)
```



View Sine Wave Input Signals at Different Sample Rates and Offsets

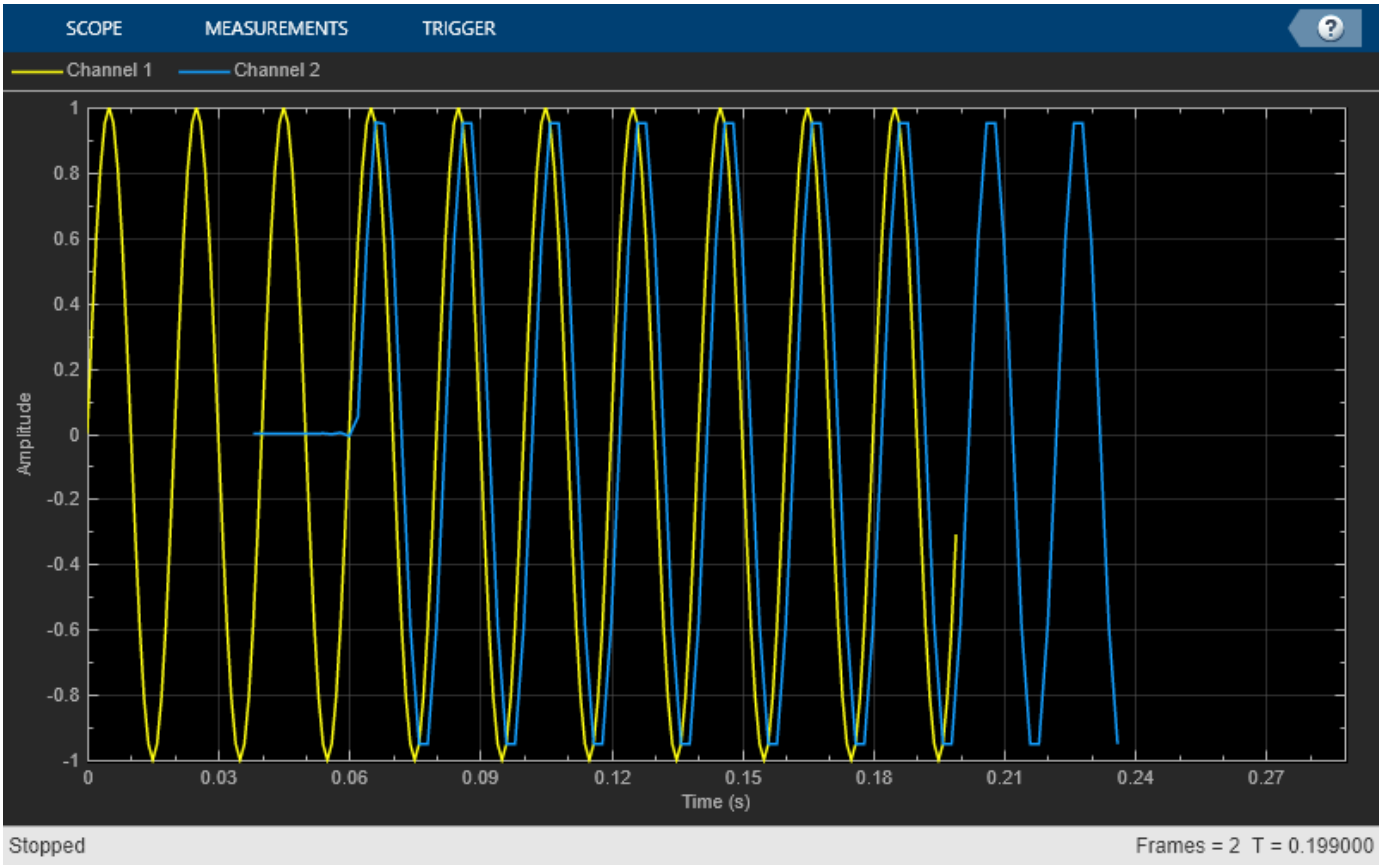
Create a `dsp.SineWave` with a 1000 Hz sampling frequency. Create a `dsp.FIRDecimator` object to decimate the sine wave by 2. Create a `timescope` object with two input ports.

```
Fs = 1000;  % Sampling frequency
sine = dsp.SineWave('Frequency',50,...
    'SampleRate',Fs, ...
    'SamplesPerFrame',100);
decimate = dsp.FIRDecimator; % To decimate sine by 2
scope = timescope('SampleRate',[Fs Fs/2], ...
    'TimeDisplayOffset',[0 38/Fs], ...
    'TimeSpanSource','Property',...
    'TimeSpan',0.25, ...
    'YLimits',[-1 1], ...
    'ShowLegend', true);
```

Call the `dsp.SineWave` object to create a sine wave signal. Use the `dsp.FIRDecimator` object to create a second signal that equals the original signal, but decimated by a factor of 2. Display the signals by calling the `timescope` object.

```
for ii = 1:2
     xsine = sine();
     xdec = decimate(xsine);
```

```
        scope(xsine,xdec)
    end
    release(scope)
```



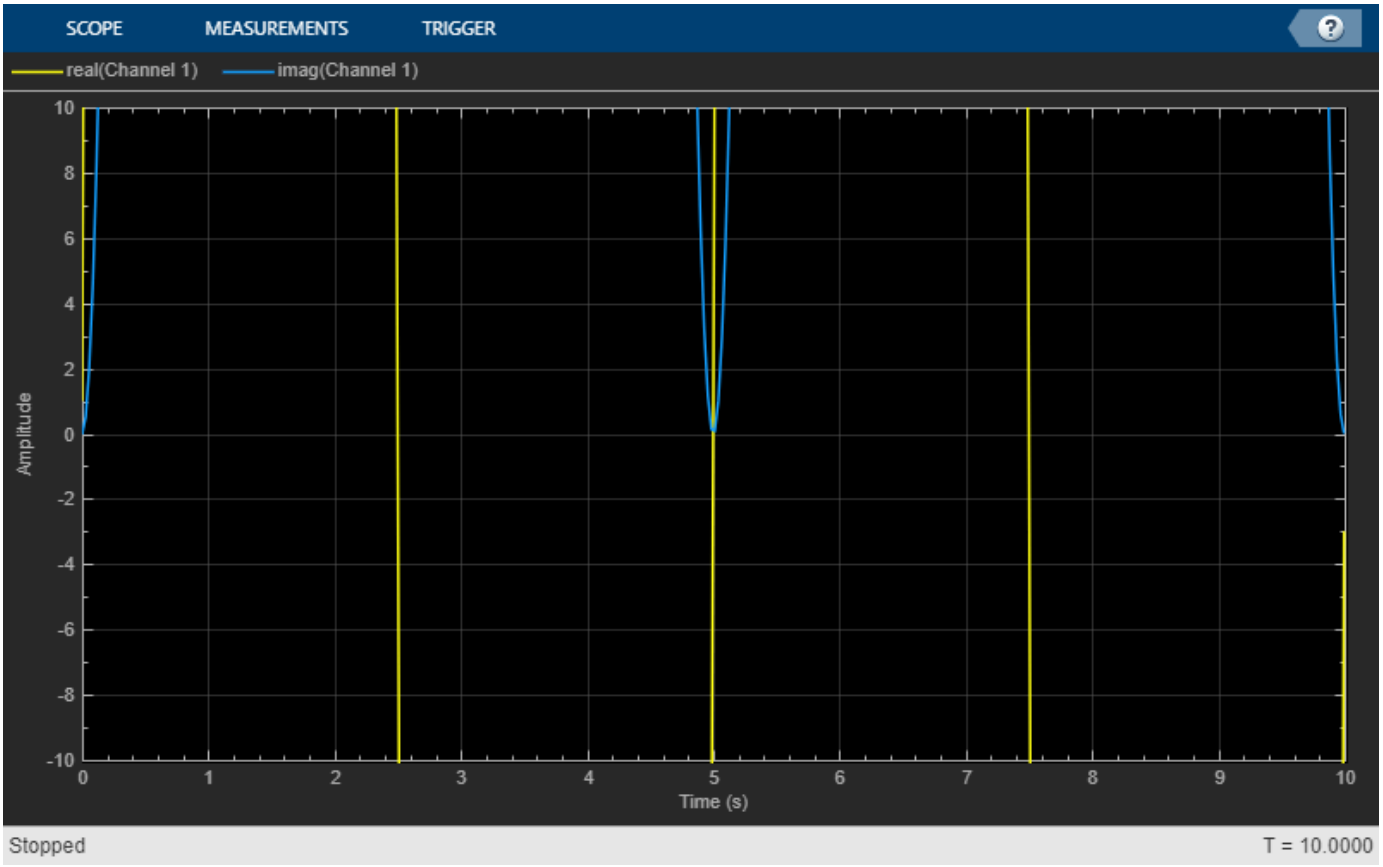
Close the Time Scope window and clear the variables.

```
clear scope Fs sine decimate ii xsine xdec
```

Display Complex-Valued Input Signal

Create a vector representing a complex-valued sinusoidal signal, and create a `timescope` object. Call the scope to display the signal.

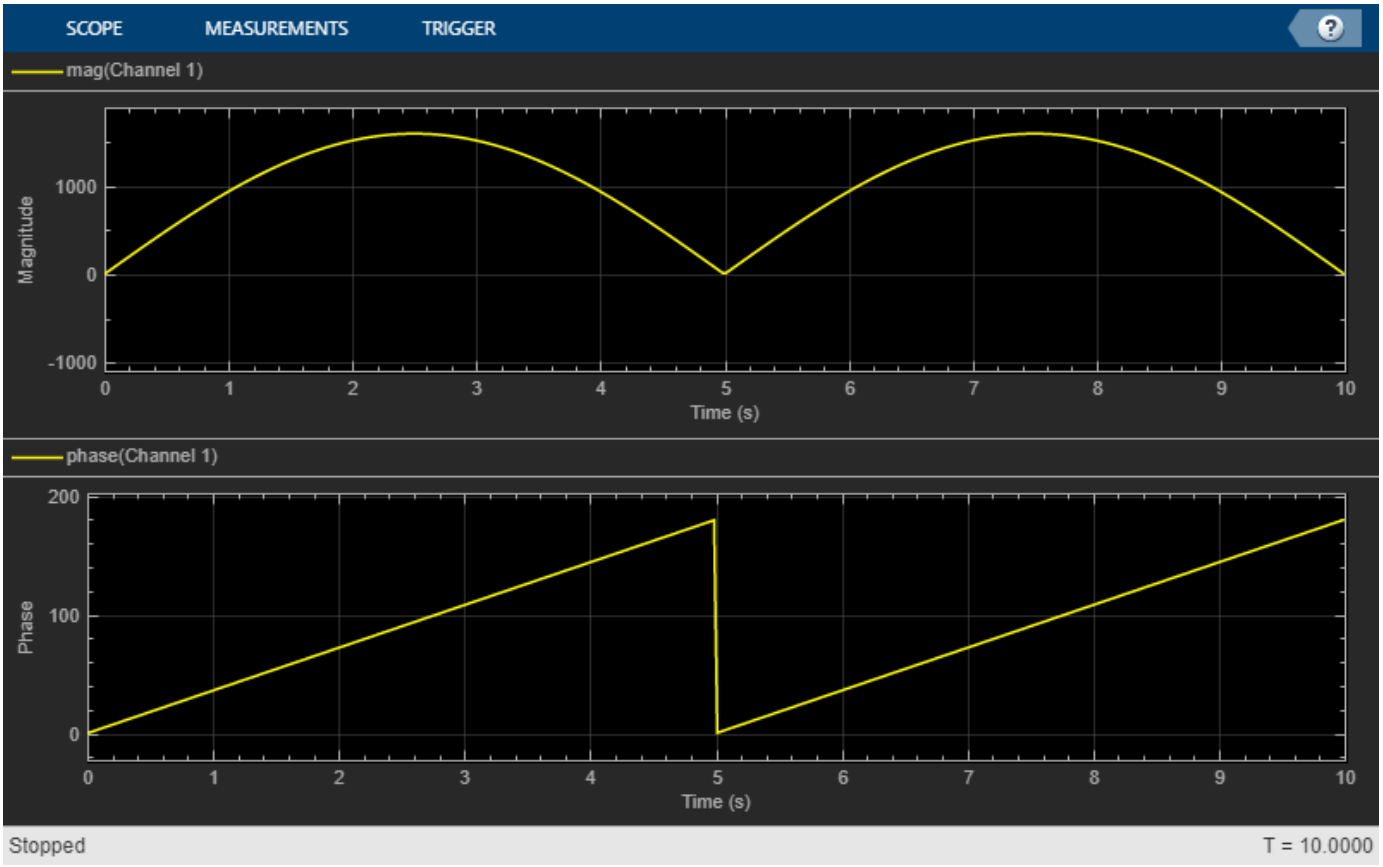
```
fs = 1000;
t = (0:1/fs:10)';
CxSine = cos(2*pi*0.2*t) + 1i*sin(2*pi*0.2*t);
CxSineSum = cumsum(CxSine);
scope = timescope('SampleRate',fs,'TimeSpanSource','Auto','ShowLegend',1);
scope(CxSineSum);
release(scope)
```



By default, when the input is a complex-valued signal, Time Scope plots the real and imaginary portions on the same axes. These real and imaginary portions appear as different-colored lines on the same axes within the same active display.

Change the `PlotAsMagnitudePhase` property to `true` and call `release`.

```
scope.PlotAsMagnitudePhase = true;
scope(CxSineSum);
release(scope)
```



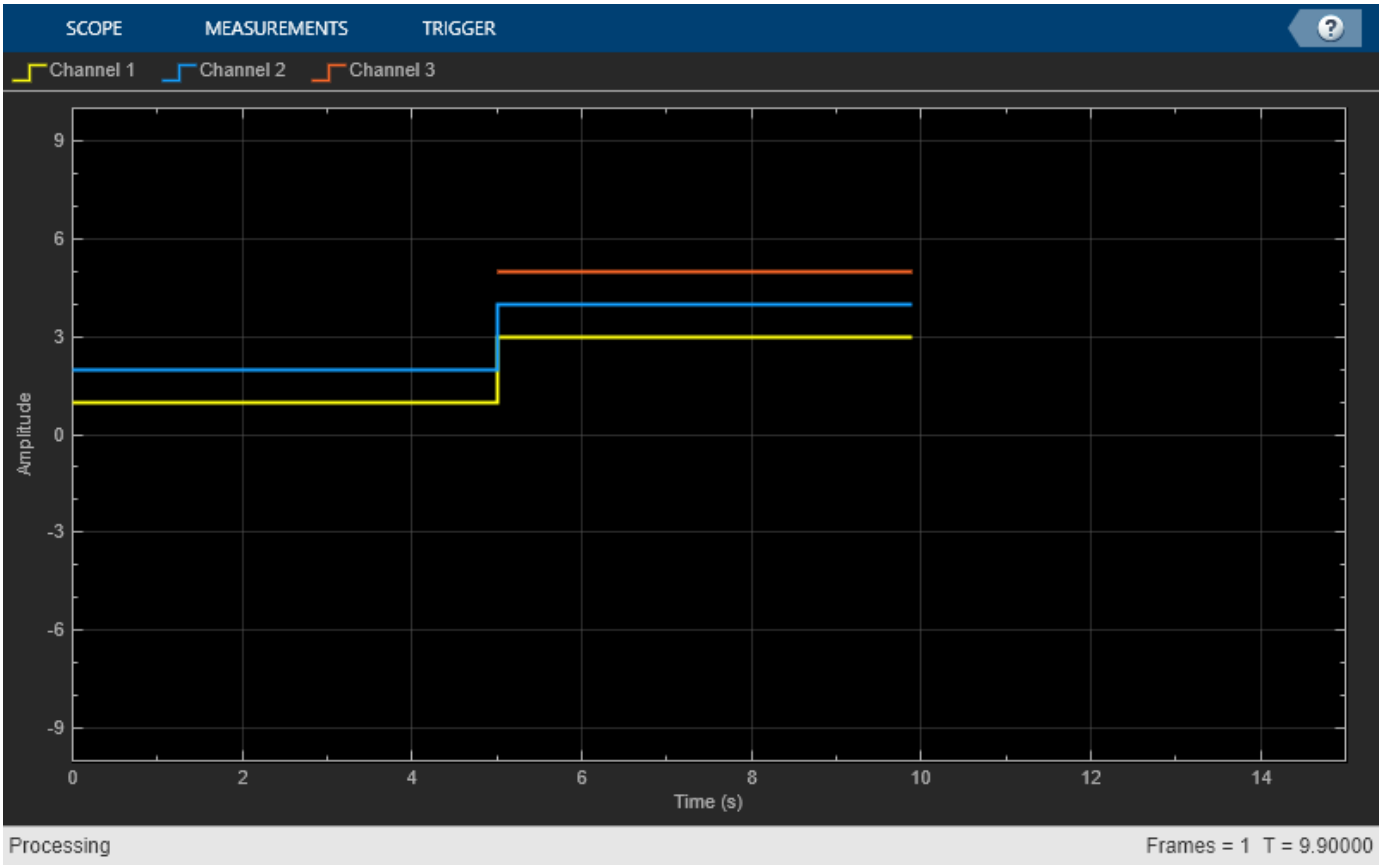
Time Scope now plots the magnitude and phase of the input signal on two separate axes within the same active display. The top axes display magnitude and the bottom axes display the phase, in degrees.

Display Input Signal of Changing Size

This example shows how the `timescope` object visualizes inputs that change dimensions halfway through.

Create a vector that represents a two-channel constant signal. Create another vector that represents a three-channel constant signal. Create a `timescope` object. Call the scope with two inputs to display the signal.

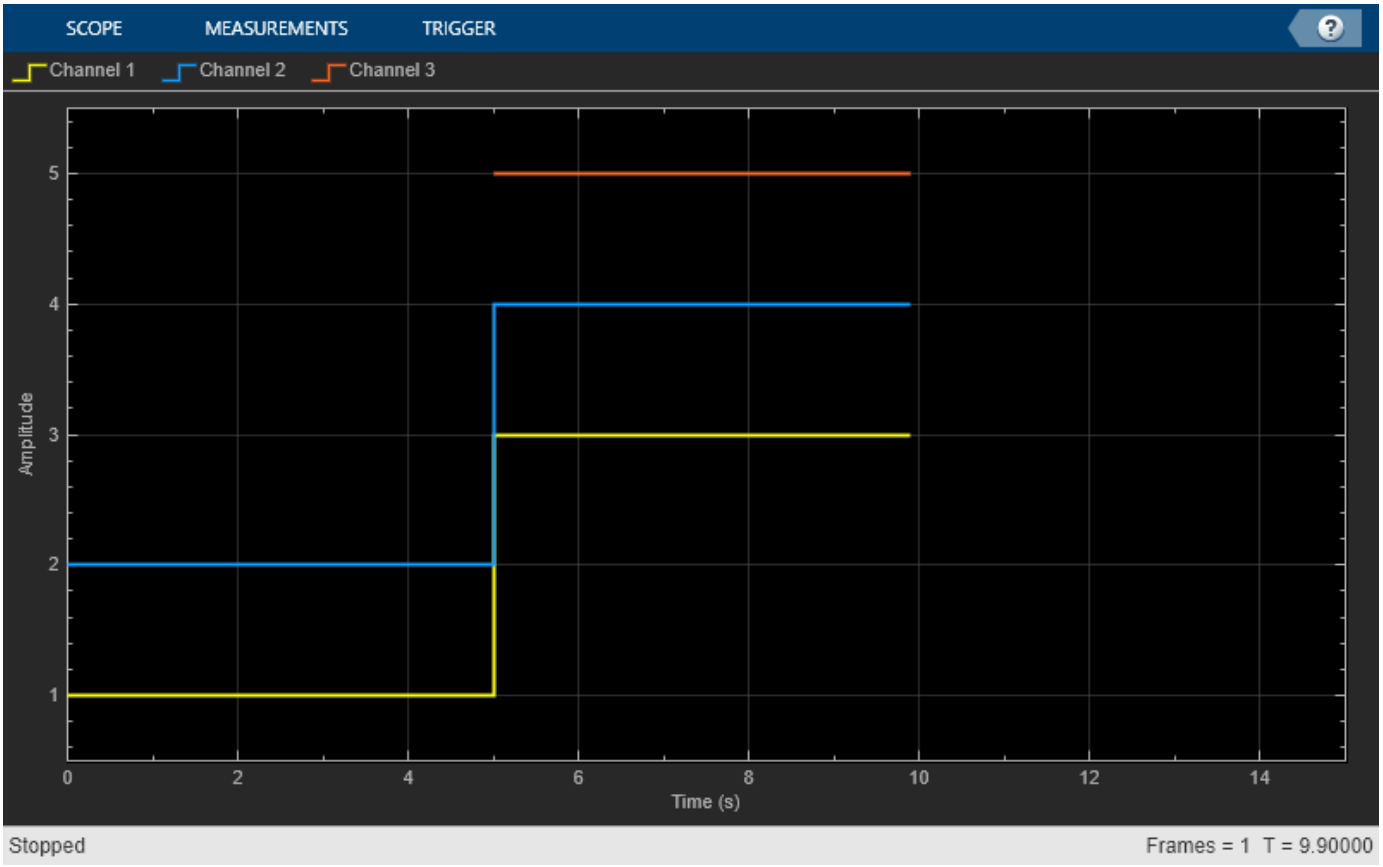
```
fs = 10;
sigdim2 = [ones(5*fs,1) 1+ones(5*fs,1)];                    % 2-dim 0-5 s
sigdim3 = [2+ones(5*fs,1) 3+ones(5*fs,1) 4+ones(5*fs,1)];  % 3-dim 5-10 s
scope = timescope('SampleRate',fs,'TimeSpanSource','Property');
scope.PlotType = 'Stairs';
scope.TimeSpanOverrunAction = 'Scroll';
scope.TimeDisplayOffset = [0 5];
scope([sigdim2; sigdim3(:,1:2)], sigdim3(:,3));
```



In this example, the size of the input signal to the Time Scope changes as the simulation progresses. When the simulation time is less than 5 seconds, Time Scope plots only the two-channel signal, `sigdim2`. After 5 seconds, Time Scope also plots the three-channel signal, `sigdim3`.

Run the `release` method to enable changes to property values and input characteristics. The scope automatically scales the axes.

```
release(scope)
```



Find Heart Rate Using Peak Finder Panel with ECG Input Signal

Use Peak Finder panel of the Time Scope to measure a heart rate.

Create and Display ECG Signal

Create the electrocardiogram (ECG) signal. The custom `ecg` function helps generate the heartbeat signal.

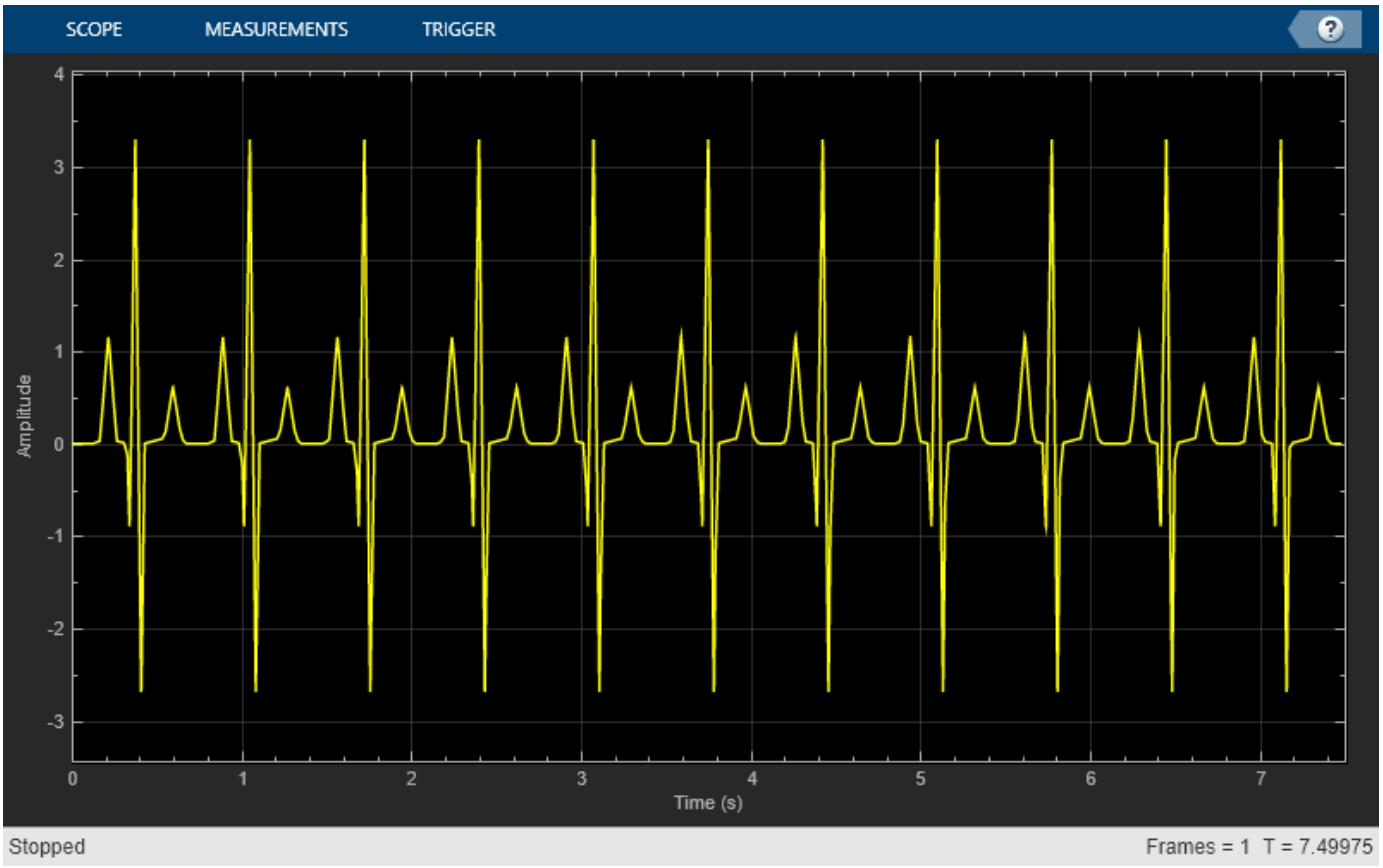
```
function x = ecg(L)
a0 = [0,  1, 40,  1,   0, -34, 118, -99,   0,   2,  21,   2,   0,   0,   0];
d0 = [0, 27, 59, 91, 131, 141, 163, 185, 195, 275, 307, 339, 357, 390, 440];
a = a0 / max(a0);
d = round(d0 * L / d0(15));
d(15) = L;
for i = 1:14
    m = d(i) : d(i+1) - 1;
    slope = (a(i+1) - a(i)) / (d(i+1) - d(i));
    x(m+1) = a(i) + slope * (m - d(i));
end
```

```
x1 = 3.5*ecg(2700).';
y1 = sgolayfilt(kron(ones(1,13),x1),0,21);
```

```
n = (1:30000)';
del = round(2700*rand(1));
mhb = y1(n + del);
ts = 0.00025;
```

Create a `timescope` object and call the object to display the signal. To autoscale the axes and enable changes to property values and input characteristics, call `release`.

```
scope = timescope('SampleRate',1/ts);
scope(mhb);
release(scope)
```



Find Heart Rate

Use the Peak Finder measurements to measure the time between heart beats.

1. On the **Measurements** tab, select **Peak Finder**.
2. For the **Num Peaks** property, enter `10`.

In the **Peaks** pane at the bottom of the window, the Time Scope displays a list of ten peak amplitude values and the times at which they occur.

The list of peak values shows a constant time difference of 0.675 second between each heartbeat. Based on the following equation, the heart rate of this ECG signal is about 89 beats per minute.

$$\frac{60\ \text{s/min}}{0.675\ \text{s/beat}} = 88.89\ \text{bpm}$$

Close the Time Scope window and remove the variables you created from the workspace.

```
clear scope x1 y1 n del mhb ts
```

Tips

- To close the scope window and clear its associated data, use the MATLAB `clear` function.
- To hide or show the scope window, use the `hide` and `show` functions.
- Use the MATLAB `mcc` function to compile code containing a scope. You cannot open scope configuration dialogs if you have more than one compiled component in your application.

See Also

Topics

“Configure Time Scope MATLAB Object”

Introduced in R2020a

generateScript

Generate MATLAB script to create scope with current settings

Syntax

```
generateScript(scope)
```

Description

`generateScript(scope)` generates a MATLAB script that can re-create a `timescope` object with the current settings in the scope.

Examples

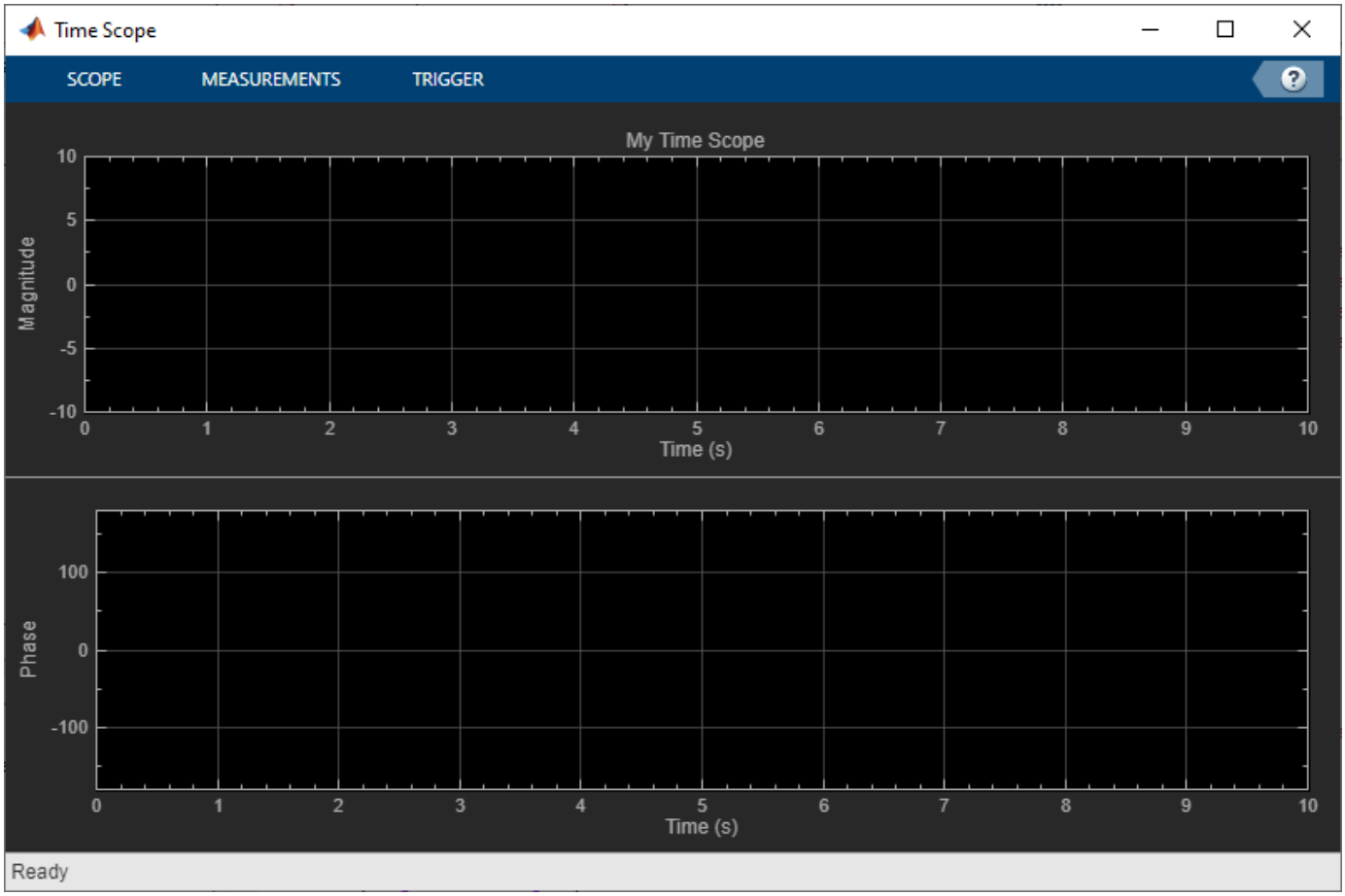
Generate Script from `timescope`

Generate MATLAB script after making changes to the `timescope` object in the scope window.

Note The script only generates commands for settings that are available from the command line, applicable to the current visualization, and changed from the default value.

1. Create a `timescope` object.

   ```
   scope = timescope;
   show(scope)
   ```
2. Set options in the Time Scope. For this example, on the **Scope** tab, click **Settings**. Under **Display and Labels**, select **Show Legend** and **Magnitude Phase Plot**. Set the **Title** as well.



3 Generate a script to recreate the `timescope` with the same modified settings. Either select **Generate Script** from the **Scope** tab, or enter:

```
generateScript(scope);
```

A new editor window opens with code to regenerate the same scope.

```
% Creation Code for 'timescope'.
% Generated by Time Scope on 8-Nov-2019 13:51:54 -0500.

timeScope = timescope('Position',[2286 355 800 500], ...
    'Title','My Time Scope', ...
    'ShowLegend',true, ...
    'PlotAsMagnitudePhase',true);
```

Input Arguments

`scope` — object

`timescope` object

Object whose settings you want to recreate with a script.

See Also

Functions

`hide` | `show` | `isVisible`

Objects
`timescope`

Introduced in R2020a

hide

Hide scope window

Syntax

`hide(scope)`

Description

`hide(scope)` hides the scope window.

Examples

View Sine Wave on Time Scope

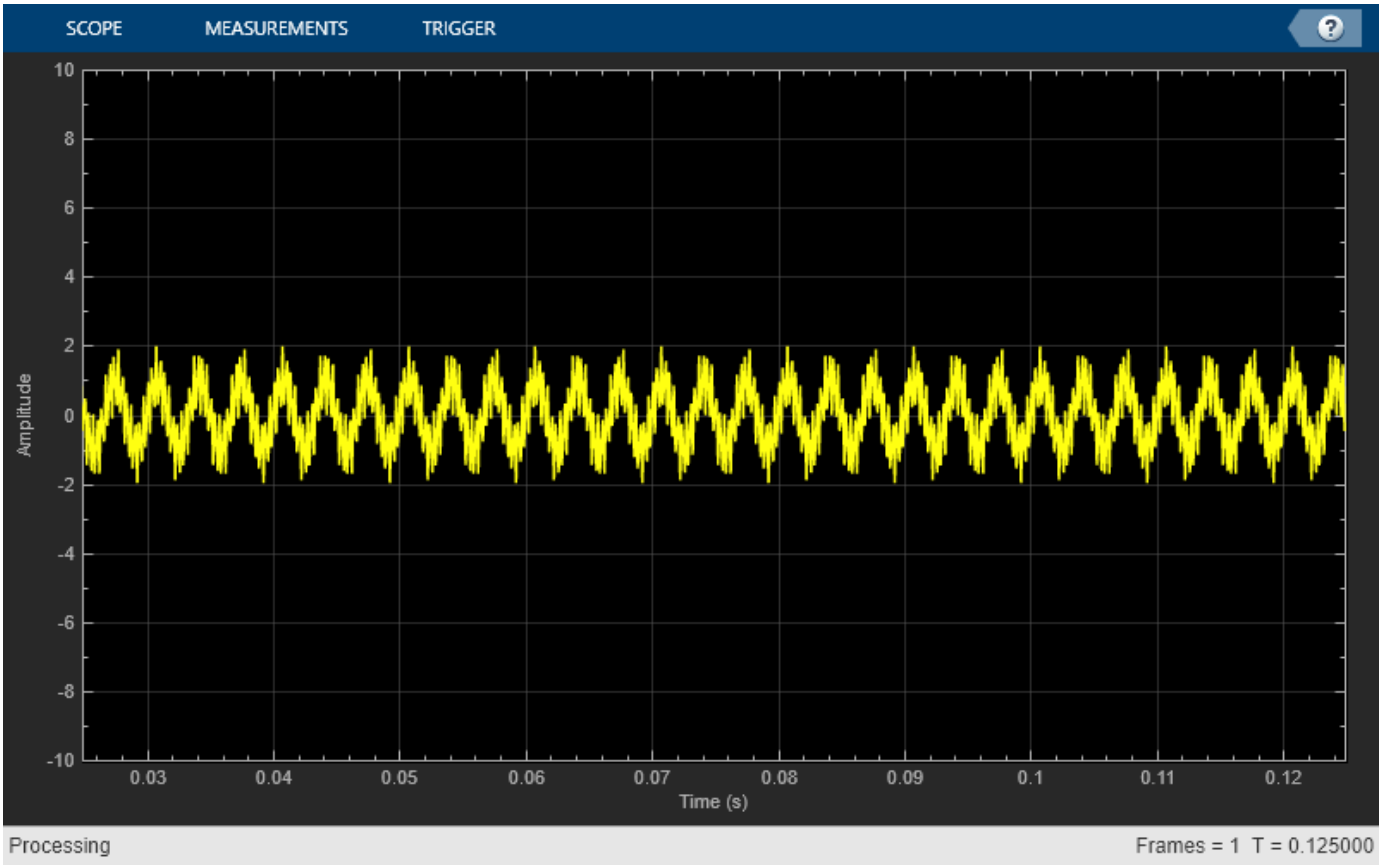
Create a time-domain sinusoidal signal. Display the signal by calling the time scope object.

Create a sinusoidal signal with two tones, one at 0.3 kHz and the other at 3 kHz.

```
t = (0:1000)'/8e3;
xin = sin(2*pi*0.3e3*t)+sin(2*pi*3e3*t);
```

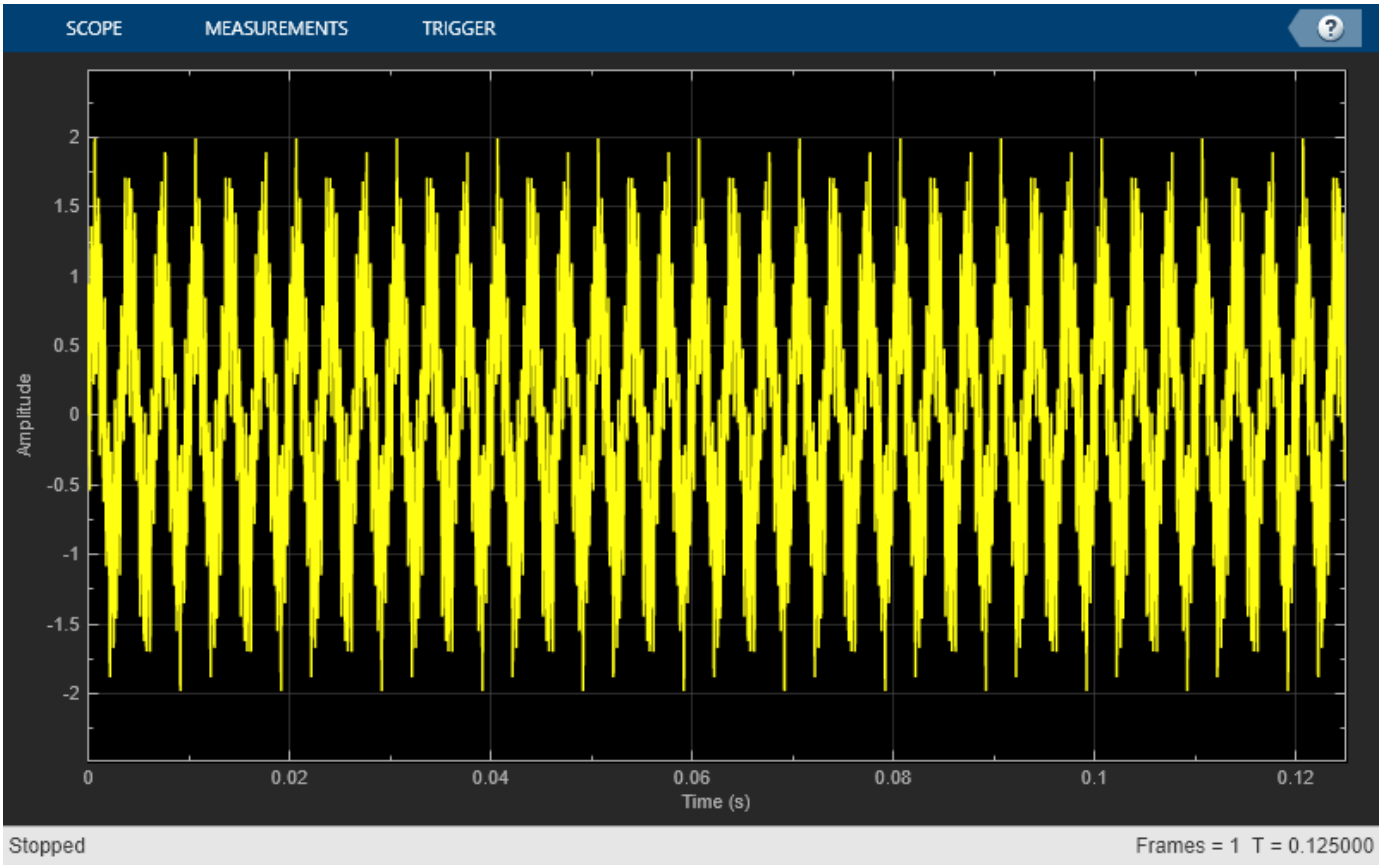
Create a `timescope` object and view the sinusoidal signal by calling the time scope object `scope`.

```
scope = timescope('SampleRate', 8e3,...
    'TimeSpanSource', 'property', ...
    'TimeSpan', 0.1);
scope(xin)
```



Run `release` to allow changes to property values and input characteristics. The scope automatically scales the axes.

```
release(scope);
```

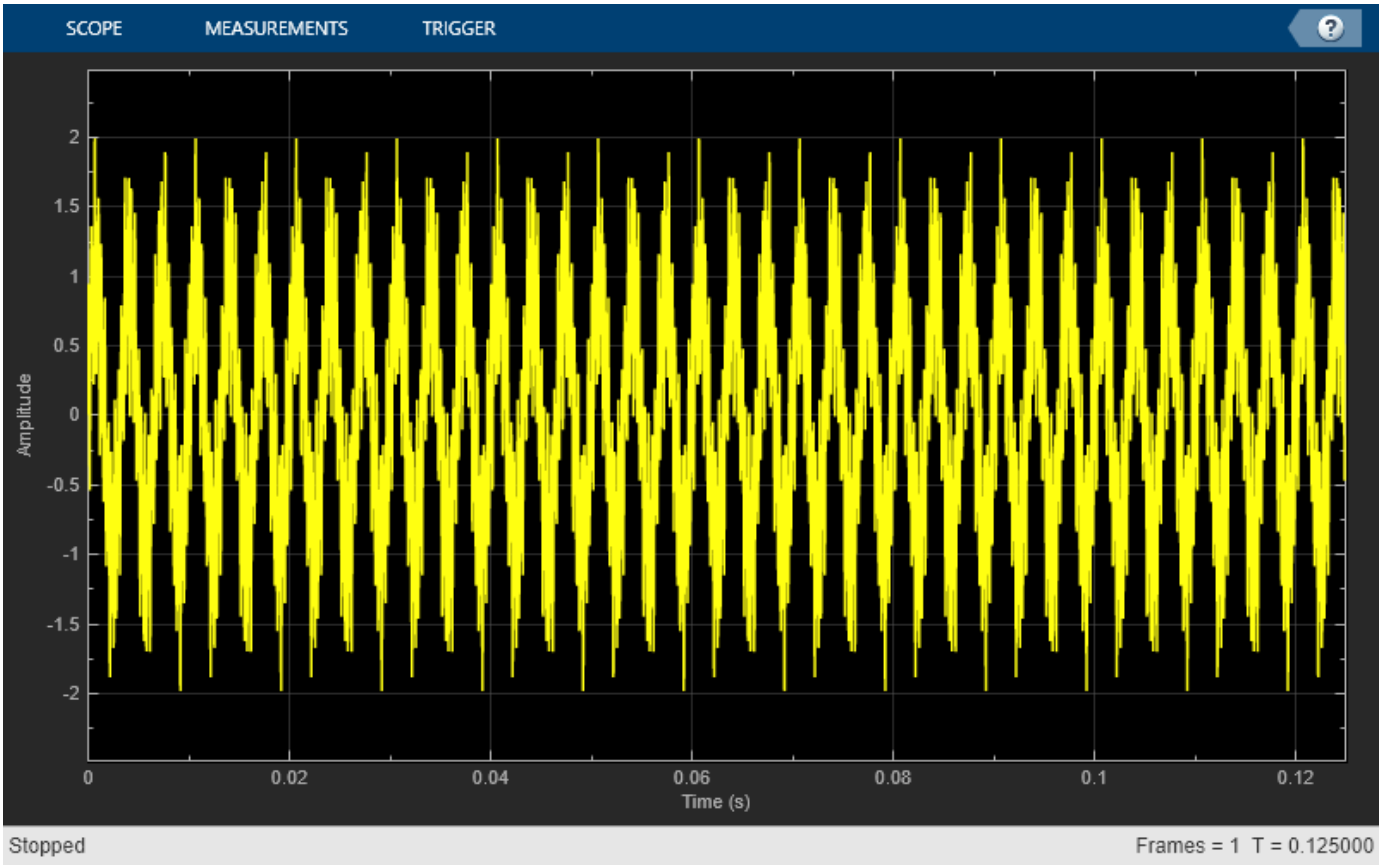


Hide the scope window.

```
if(isVisible(scope))
    hide(scope)
end
```

Show the scope window.

```
if(~isVisible(scope))
    show(scope)
end
```



Input Arguments

`scope` — Scope object

`timescope` object

Scope object whose window you want to hide, specified as a `timescope` object.

Example: `myScope = timescope; hide(myScope)`

See Also

Functions

`show` | `isVisible` | `generateScript`

Objects

`timescope`

Introduced in R2020a

isVisible

Determine visibility of scope

Syntax

```
visibility = isVisible(scope)
```

Description

`visibility = isVisible(scope)` returns the visibility of the scope as logical, with `1` (`true`) for visible.

Examples

View Sine Wave on Time Scope

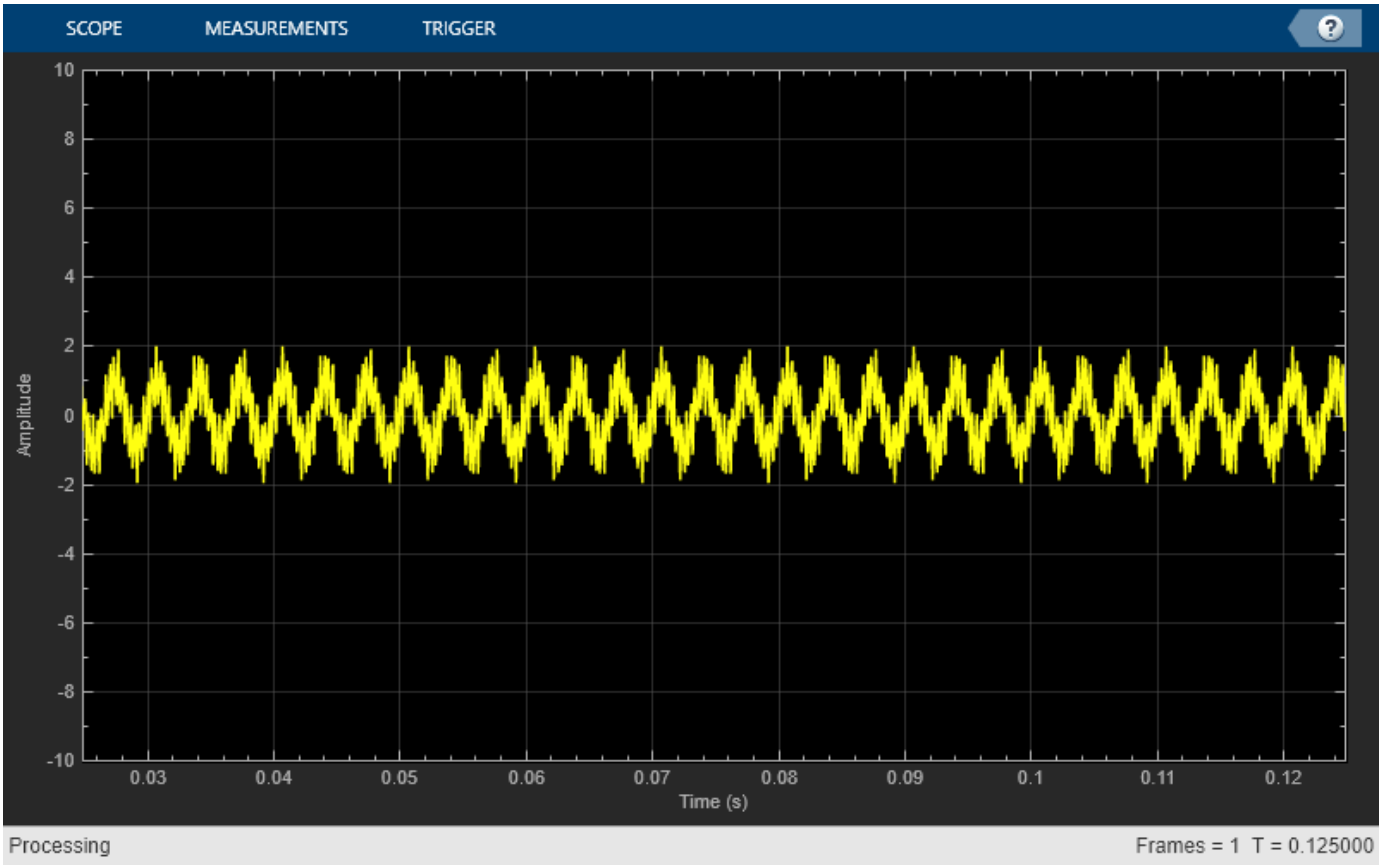
Create a time-domain sinusoidal signal. Display the signal by calling the time scope object.

Create a sinusoidal signal with two tones, one at 0.3 kHz and the other at 3 kHz.

```
t = (0:1000)'/8e3;
xin = sin(2*pi*0.3e3*t)+sin(2*pi*3e3*t);
```

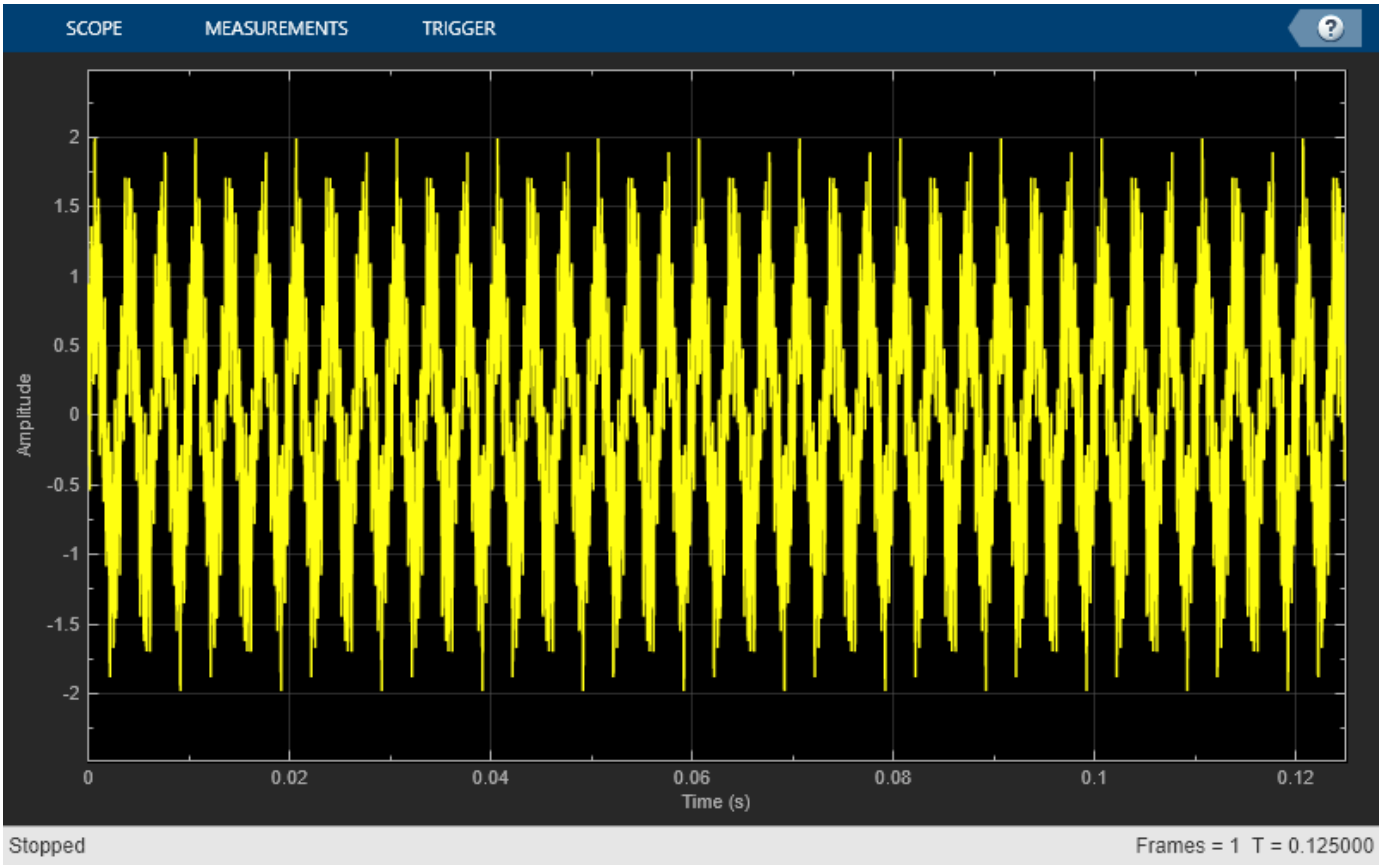
Create a `timescope` object and view the sinusoidal signal by calling the time scope object `scope`.

```
scope = timescope('SampleRate', 8e3,...
    'TimeSpanSource', 'property', ...
    'TimeSpan', 0.1);
scope(xin)
```



Run `release` to allow changes to property values and input characteristics. The scope automatically scales the axes.

```
release(scope);
```

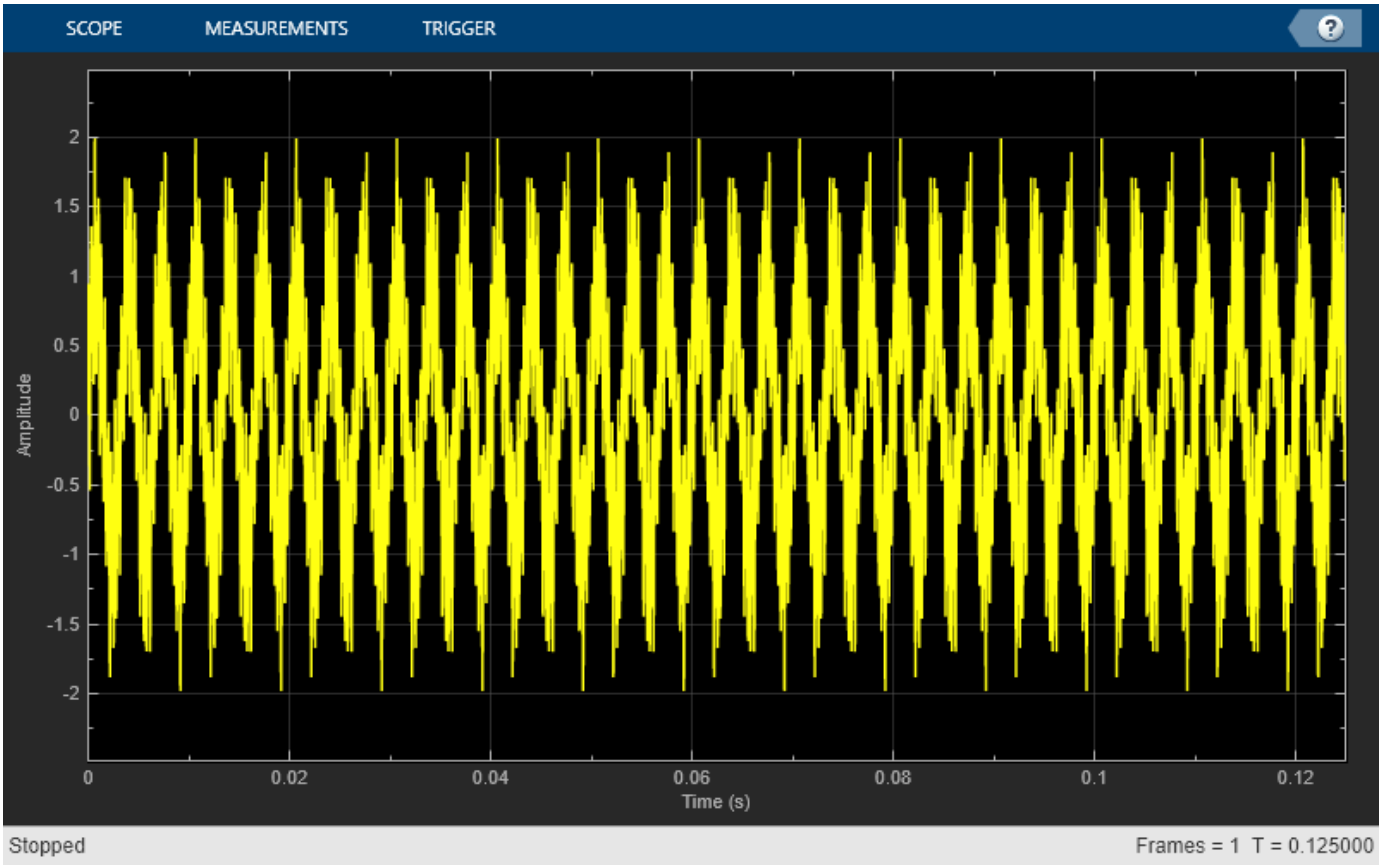


Hide the scope window.

```
if(isVisible(scope))
    hide(scope)
end
```

Show the scope window.

```
if(~isVisible(scope))
    show(scope)
end
```



Input Arguments

`scope` — Scope object

`timescope` object

Scope object whose visibility you want to query.

Example: `myScope = timescope; visibility = isVisible(myScope)`

Output Arguments

`visibility` — Scope visibility

`1` | `0`

If the scope window is open, the `isVisible` function returns `1` (`true`). Otherwise, the function returns `0` (`false`).

See Also

Functions

`hide` | `show` | `generateScript`

Objects
`timescope`

Introduced in R2020a

show

Display scope window

Syntax

`show(scope)`

Description

`show(scope)` shows the scope window.

Examples

View Sine Wave on Time Scope

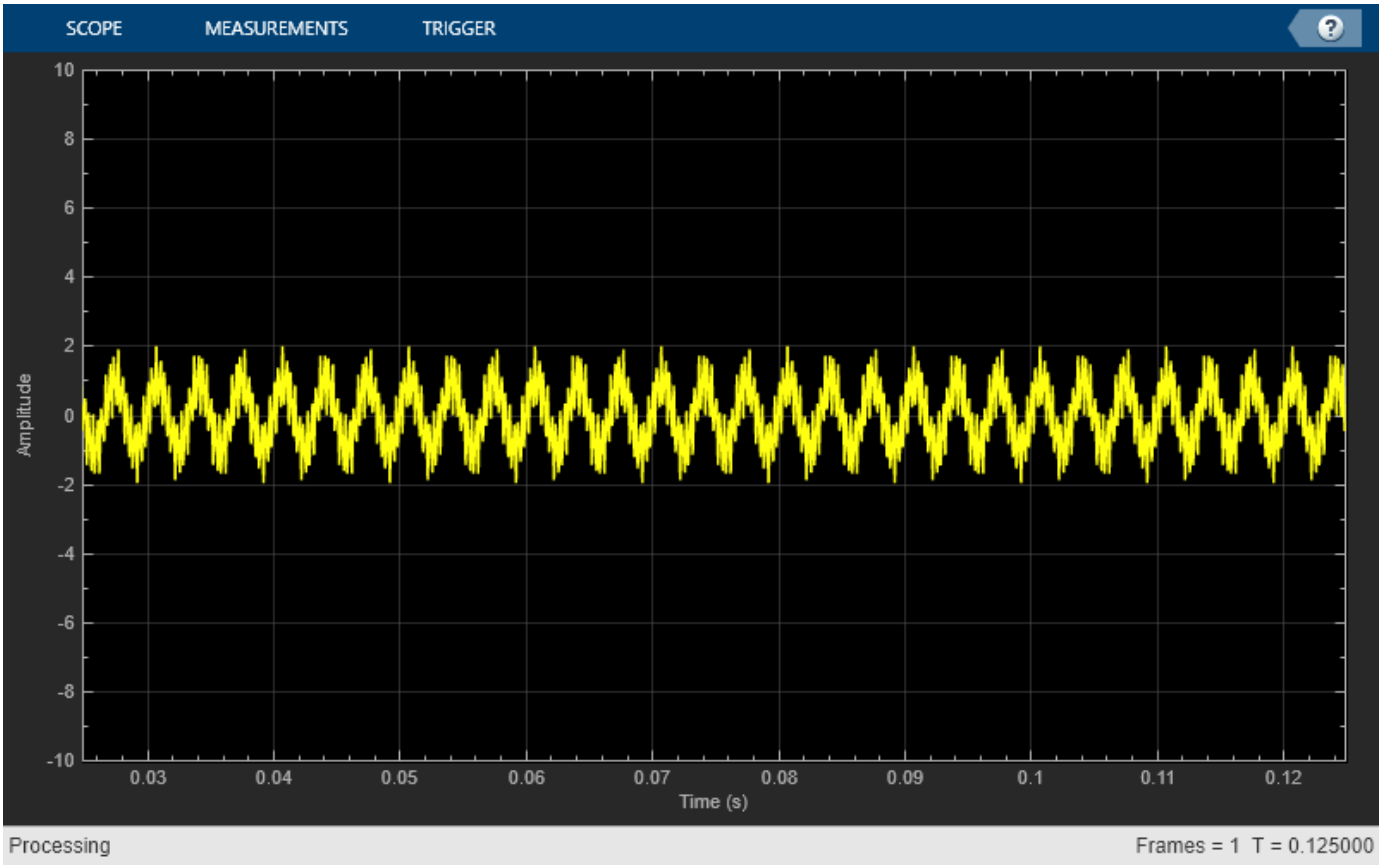
Create a time-domain sinusoidal signal. Display the signal by calling the time scope object.

Create a sinusoidal signal with two tones, one at 0.3 kHz and the other at 3 kHz.

```
t = (0:1000)'/8e3;
xin = sin(2*pi*0.3e3*t)+sin(2*pi*3e3*t);
```

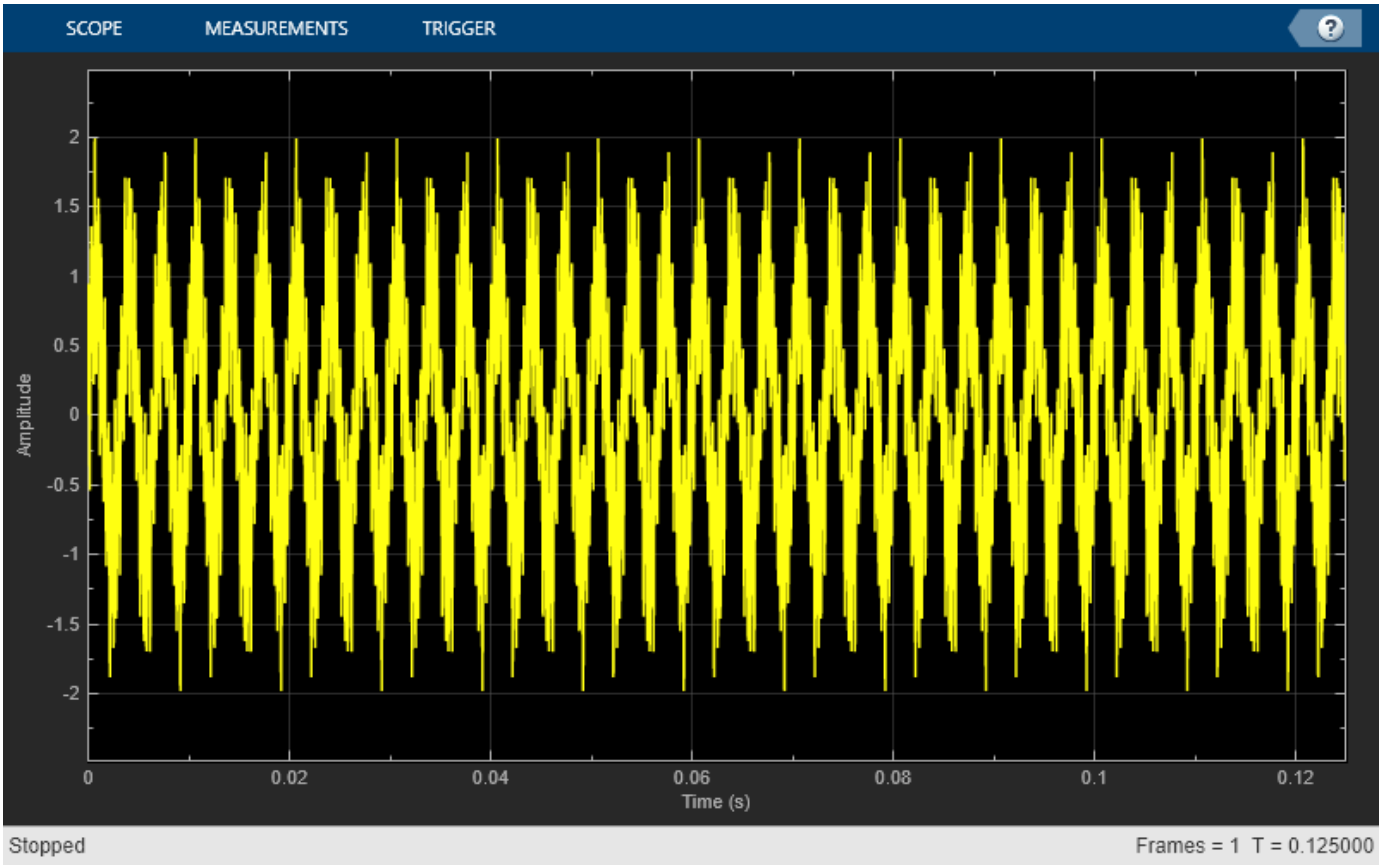
Create a `timescope` object and view the sinusoidal signal by calling the time scope object `scope`.

```
scope = timescope('SampleRate', 8e3,...
    'TimeSpanSource', 'property', ...
    'TimeSpan', 0.1);
scope(xin)
```



Run `release` to allow changes to property values and input characteristics. The scope automatically scales the axes.

```
release(scope);
```

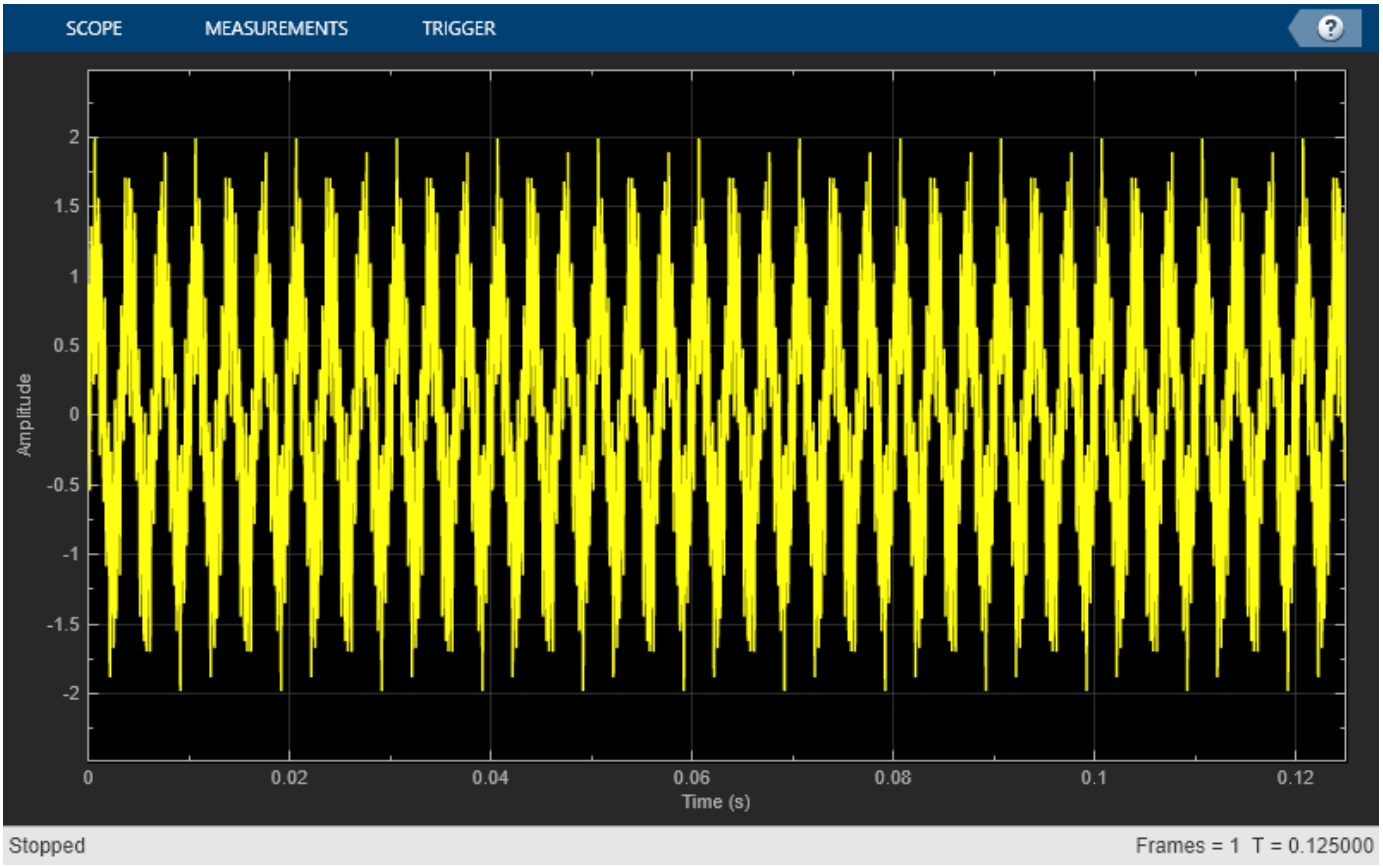


Hide the scope window.

```
if(isVisible(scope))
    hide(scope)
end
```

Show the scope window.

```
if(~isVisible(scope))
    show(scope)
end
```



Input Arguments

`scope` — Scope object

`timescope` object

Scope object whose window you want to show, specified as a `timescope` object.

Example: `myScope = timescope; show(myScope)`

See Also

Functions

`hide` | `isVisible` | `generateScript`

Objects

`timescope`

Introduced in R2020a

tunerconfig

Fusion filter tuner configuration options

Description

The `tunerconfig` object creates a tuner configuration for a fusion filter used to tune the filter for reduced estimation error.

Creation

Syntax

```
config = tunerconfig(filterName)
config = tunerconfig(filterName,Name,Value)
```

Description

`config = tunerconfig(filterName)` creates a `tunerconfig` object controlling the optimization algorithm of the tune function of the fusion filter.

`config = tunerconfig(filterName,Name,Value)` configures the created `tunerconfig` object properties using one or more name-value pair arguments. `Name` is a property name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name-value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`. Any unspecified properties take default values.

For example, `tunerconfig('imufilter','MaxIterations',3)` create a `tunerconfig` object for the `imufilter` object with the a maximum of three allowed iterations.

Inputs Arguments

`filterName` — Fusion filter name
`'imufilter'` | `'ahrsfilter'` | `'ahrs10filter'` | `'insfilterAsync'` | `'insfilterMARG'` | `'insfitlerErrorState'` | `'insfilterNonholonomic'`

Fusion filter name, specified as one of these options:

- `'imufilter'`
- `'ahrsfilter'`
- `'ahrs10filter'`
- `'insfilterAsync'`
- `'insfilterMARG'`
- `'insfitlerErrorState'`
- `'insfilterNonholonomic'`

Properties

`Filter` — Class name of filter

string

This property is read-only.

Class name of filter, specified as a string. Its value is one of these strings:

- `"imufilter"`
- `"ahrsfilter"`
- `"ahrs10filter"`
- `"insfilterAsync"`
- `"insfilterMARG"`
- `"insfitlerErrorState"`
- `"insfilterNonholonomic"`

`TunableParameters` — Tunable parameters

array of string

Tunable parameters, specified as an array of strings. Each string is a tunable property name of the fusion filter. By default, the property contains all the tunable parameters of a fusion filter.

Example: `["AccelerometerNoise" "GyroscopeNoise"]`

`StepForward` — Factor of forward step

`1.1` (default) | scalar larger than 1

Factor of a forward step, specified as a scalar larger than 1. During the tuning process, the tuner increases or decreases the noise parameters to achieve smaller estimation errors. This property specifies the ratio of parameter increase during a parameter increase step.

`StepBackward` — Factor of backward step

`0.5` (default) | scalar in range (0,1)

Factor of a backward step, specified as a scalar in the range of (0,1). During the tuning process, the tuner increases or decreases the noise parameters to achieve smaller estimation errors. This property specifies the factor of parameter decrease during a parameter decrease step.

`MaxIterations` — Maximum number of iterations

`20` (default) | positive integer

Maximum number of iterations allowed by the tuning algorithm, specified as a positive integer.

`ObjectiveLimit` — Cost at which to stop tuning process

`0.1` (default) | positive scalar

Cost at which to stop the tuning process, specified as a positive scalar.

`FunctionTolerance` — Minimum change in cost to continue tuning

`0` (default) | nonnegative scalar

Minimum change in cost to continue tuning, specified as a nonnegative scalar. If the change in cost is smaller than the specified tolerance, the tuning process stops.

`Display` — Enable showing the iteration details

`"iter"` (default) | `"none"`

Enable showing the iteration details, specified as `"iter"` or `"none"`. When specified as:

- `"iter"` — The program shows the tuned parameter details in each iteration in the Command Window.
- `"none"` — The program does not show any tuning information.

`Cost` — Metric for evaluating filter performance

`"RMS"` (default) | `"Custom"`

Metric for evaluating filter performance, specified as `"RMS"` or `"Custom"`. When specified as:

- `"RMS"` — The program optimizes the root-mean-squared (RMS) error between the estimate and the truth.
- `"Custom"` — The program optimizes the filter performance by using a customized cost function specified by the `CustomCostFcn` property.

`CustomCostFcn` — Customized cost function

`[]` (default) | function handle

Customized cost function, specified as a function handle.

Dependencies

To enable this property, set the `Cost` property to `'Custom'`.

`OutputFcn` — Output function called at each iteration

`[]` (default) | function handle

Output function called at each iteration, specified as a function handle. The function must use the following syntax:

```
stop = myOutputFcn(params,tunerValues)
```

`params` is a structure of the current best estimate of each parameter at the end of the current iteration. `tunerValues` is a structure containing information of the tuner configuration, sensor data, and truth data. It has these fields:

Field Name	Description
`Iteration`	Iteration count of the tuner, specified as a positive integer
`SensorData`	Sensor data input to the `tune` function
`GroundTruth`	Ground truth input to the `tune` function
`Configuration`	`tunerconfig` object used for tuning
`Cost`	Tuning cost at the end of the current iteration

Tip You can use the built-in function `tunerPlotPose` to visualize the truth data and the estimates for most of your tuning applications. See the “Visualize Tuning Results Using tunerPlotPose” on page 1-239 example for details.

Examples

Create Tunerconfig Object and Show Tunable Parameters

Create a tunerconfig object for the `insfilterAsync` object.

```
config = tunerconfig('insfilterAsync')
```

```
config =
  tunerconfig with properties:

      TunableParameters: [1×14 string]
            StepForward: 1.1000
           StepBackward: 0.5000
          MaxIterations: 20
    OptimalityTolerance: 0.1000
                Display: iter
                   Cost: RMS
```

Display the default tunable parameters.

```
config.TunableParameters
```

```
ans = 1×14 string
    "AccelerometerNoise"    "GyroscopeNoise"    "MagnetometerNoise"    "GPSPositionNoise"    "GPS
```

Tune `insfilterAsync` to Optimize Pose Estimate

Load the recorded sensor data and ground truth data.

```
load('insfilterAsyncTuneData.mat');
```

Create timetables for the sensor data and the truth data.

```
sensorData = timetable(Accelerometer, Gyroscope, ...
    Magnetometer, GPSPosition, GPSVelocity, 'SampleRate', 100);
groundTruth = timetable(Orientation, Position, ...
    'SampleRate', 100);
```

Create an `insfilterAsync` filter object that has a few noise properties.

```
filter = insfilterAsync('State', initialState, ...
    'StateCovariance', initialStateCovariance, ...
    'AccelerometerBiasNoise', 1e-7, ...
    'GyroscopeBiasNoise', 1e-7, ...
    'MagnetometerBiasNoise', 1e-7, ...
    'GeomagneticVectorNoise', 1e-7);
```

Create a tuner configuration object for the filter. Set the maximum iterations to two. Also, set the tunable parameters as the unspecified properties.

```
config = tunerconfig('insfilterAsync','MaxIterations',8);
config.TunableParameters = setdiff(config.TunableParameters, ...
```

```
    {'GeomagneticVectorNoise', 'AccelerometerBiasNoise', ...
    'GyroscopeBiasNoise', 'MagnetometerBiasNoise'});
config.TunableParameters
```

```
ans = 1×10 string
    "AccelerationNoise"    "AccelerometerNoise"    "AngularVelocityNoise"    "GPSPositionNoise"
```

Use the tuner noise function to obtain a set of initial sensor noises used in the filter.

```
measNoise = tunernoise('insfilterAsync')
```

```
measNoise = struct with fields:
    AccelerometerNoise: 1
        GyroscopeNoise: 1
     MagnetometerNoise: 1
      GPSPositionNoise: 1
      GPSVelocityNoise: 1
```

Tune the filter and obtain the tuned parameters.

```
tunedParams = tune(filter,measNoise,sensorData,groundTruth,config);
```

```
    Iteration    Parameter               Metric
    _________    _________               ______
    1            AccelerationNoise       2.1345
    1            AccelerometerNoise      2.1264
    1            AngularVelocityNoise    1.9659
    1            GPSPositionNoise        1.9341
    1            GPSVelocityNoise        1.8420
    1            GyroscopeNoise          1.7589
    1            MagnetometerNoise       1.7362
    1            PositionNoise           1.7362
    1            QuaternionNoise         1.7218
    1            VelocityNoise           1.7218
    2            AccelerationNoise       1.7190
    2            AccelerometerNoise      1.7170
    2            AngularVelocityNoise    1.6045
    2            GPSPositionNoise        1.5948
    2            GPSVelocityNoise        1.5323
    2            GyroscopeNoise          1.4803
    2            MagnetometerNoise       1.4703
    2            PositionNoise           1.4703
    2            QuaternionNoise         1.4632
    2            VelocityNoise           1.4632
    3            AccelerationNoise       1.4596
    3            AccelerometerNoise      1.4548
    3            AngularVelocityNoise    1.3923
    3            GPSPositionNoise        1.3810
    3            GPSVelocityNoise        1.3322
    3            GyroscopeNoise          1.2998
    3            MagnetometerNoise       1.2976
    3            PositionNoise           1.2976
    3            QuaternionNoise         1.2943
    3            VelocityNoise           1.2943
    4            AccelerationNoise       1.2906
    4            AccelerometerNoise      1.2836
    4            AngularVelocityNoise    1.2491
```

```
    4            GPSPositionNoise        1.2258
    4            GPSVelocityNoise        1.1880
    4            GyroscopeNoise          1.1701
    4            MagnetometerNoise       1.1698
    4            PositionNoise           1.1698
    4            QuaternionNoise         1.1688
    4            VelocityNoise           1.1688
    5            AccelerationNoise       1.1650
    5            AccelerometerNoise      1.1569
    5            AngularVelocityNoise    1.1454
    5            GPSPositionNoise        1.1100
    5            GPSVelocityNoise        1.0778
    5            GyroscopeNoise          1.0709
    5            MagnetometerNoise       1.0675
    5            PositionNoise           1.0675
    5            QuaternionNoise         1.0669
    5            VelocityNoise           1.0669
    6            AccelerationNoise       1.0634
    6            AccelerometerNoise      1.0549
    6            AngularVelocityNoise    1.0549
    6            GPSPositionNoise        1.0180
    6            GPSVelocityNoise        0.9866
    6            GyroscopeNoise          0.9810
    6            MagnetometerNoise       0.9775
    6            PositionNoise           0.9775
    6            QuaternionNoise         0.9768
    6            VelocityNoise           0.9768
    7            AccelerationNoise       0.9735
    7            AccelerometerNoise      0.9652
    7            AngularVelocityNoise    0.9652
    7            GPSPositionNoise        0.9283
    7            GPSVelocityNoise        0.8997
    7            GyroscopeNoise          0.8947
    7            MagnetometerNoise       0.8920
    7            PositionNoise           0.8920
    7            QuaternionNoise         0.8912
    7            VelocityNoise           0.8912
    8            AccelerationNoise       0.8885
    8            AccelerometerNoise      0.8811
    8            AngularVelocityNoise    0.8807
    8            GPSPositionNoise        0.8479
    8            GPSVelocityNoise        0.8238
    8            GyroscopeNoise          0.8165
    8            MagnetometerNoise       0.8165
    8            PositionNoise           0.8165
    8            QuaternionNoise         0.8159
    8            VelocityNoise           0.8159
```

Fuse the sensor data using the tuned filter.

```
dt = seconds(diff(groundTruth.Time));
N = size(sensorData,1);
qEst = quaternion.zeros(N,1);
posEst = zeros(N,3);
% Iterate the filter for prediction and correction using sensor data.
for ii=1:N
    if ii ~= 1
        predict(filter, dt(ii-1));
```

```
    end
    if all(~isnan(Accelerometer(ii,:)))
        fuseaccel(filter,Accelerometer(ii,:), ...
            tunedParams.AccelerometerNoise);
    end
    if all(~isnan(Gyroscope(ii,:)))
        fusegyro(filter, Gyroscope(ii,:), ...
            tunedParams.GyroscopeNoise);
    end
    if all(~isnan(Magnetometer(ii,1)))
        fusemag(filter, Magnetometer(ii,:), ...
            tunedParams.MagnetometerNoise);
    end
    if all(~isnan(GPSPosition(ii,1)))
        fusegps(filter, GPSPosition(ii,:), ...
            tunedParams.GPSPositionNoise, GPSVelocity(ii,:), ...
            tunedParams.GPSVelocityNoise);
    end
    [posEst(ii,:), qEst(ii,:)] = pose(filter);
end
```

Compute the RMS errors.

```
orientationError = rad2deg(dist(qEst, Orientation));
rmsorientationError = sqrt(mean(orientationError.^2))
```

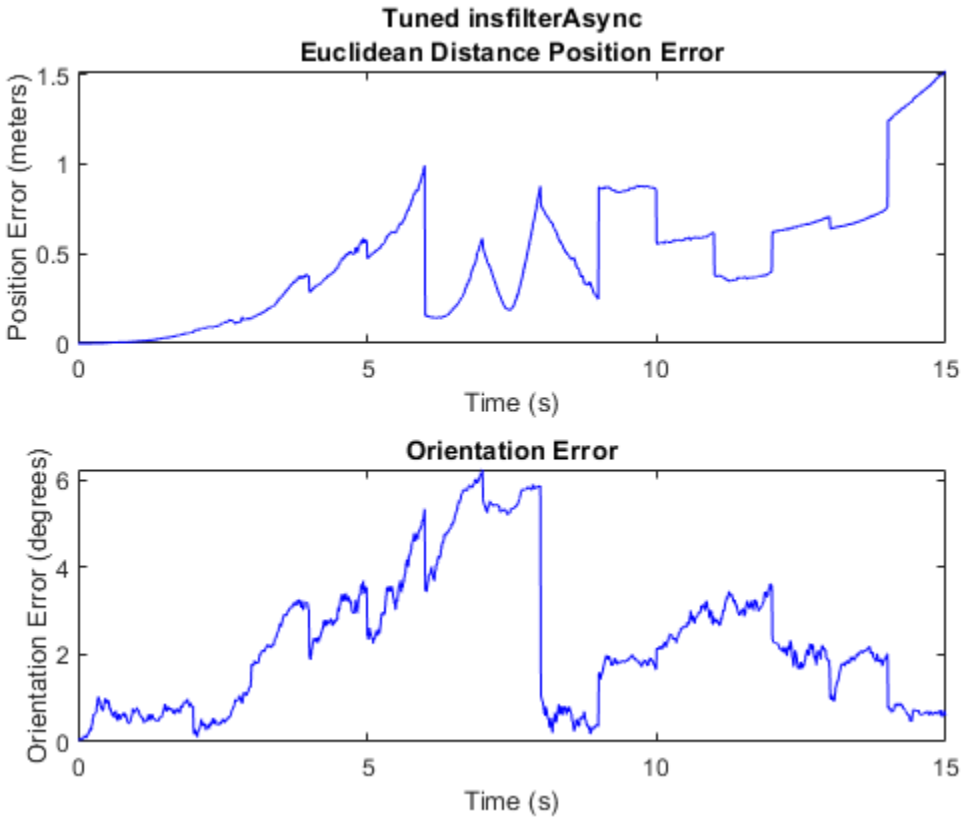
```
rmsorientationError = 2.7801
```

```
positionError = sqrt(sum((posEst - Position).^2, 2));
rmspositionError = sqrt(mean( positionError.^2))
```

```
rmspositionError = 0.5966
```

Visualize the results.

```
figure();
t = (0:N-1)./ groundTruth.Properties.SampleRate;
subplot(2,1,1)
plot(t, positionError, 'b');
title("Tuned insfilterAsync" + newline + "Euclidean Distance Position Error")
xlabel('Time (s)');
ylabel('Position Error (meters)')
subplot(2,1,2)
plot(t, orientationError, 'b');
title("Orientation Error")
xlabel('Time (s)');
ylabel('Orientation Error (degrees)');
```



Tune `imufilter` to Optimize Orientation Estimate

Load recorded sensor data and ground truth data.

```
ld = load('imufilterTuneData.mat');
qTrue = ld.groundTruth.Orientation; % true orientation
```

Create an `imufilter` object and fuse the filter with the sensor data.

```
fuse = imufilter;
qEstUntuned = fuse(ld.sensorData.Accelerometer, ...
    ld.sensorData.Gyroscope);
```

Create a `tunerconfig` object and tune the imufilter to improve the orientation estimate.

```
cfg = tunerconfig('imufilter');
tune(fuse, ld.sensorData, ld.groundTruth, cfg);
```

```
    Iteration    Parameter                       Metric
    _________    _________                       ______
    1            AccelerometerNoise              0.1149
    1            GyroscopeNoise                  0.1146
    1            GyroscopeDriftNoise             0.1146
    1            LinearAccelerationNoise         0.1122
    1            LinearAccelerationDecayFactor   0.1103
```

```
2            AccelerometerNoise              0.1102
2            GyroscopeNoise                  0.1098
2            GyroscopeDriftNoise             0.1098
2            LinearAccelerationNoise         0.1070
2            LinearAccelerationDecayFactor   0.1053
3            AccelerometerNoise              0.1053
3            GyroscopeNoise                  0.1048
3            GyroscopeDriftNoise             0.1048
3            LinearAccelerationNoise         0.1016
3            LinearAccelerationDecayFactor   0.1002
4            AccelerometerNoise              0.1001
4            GyroscopeNoise                  0.0996
4            GyroscopeDriftNoise             0.0996
4            LinearAccelerationNoise         0.0962
4            LinearAccelerationDecayFactor   0.0950
5            AccelerometerNoise              0.0950
5            GyroscopeNoise                  0.0943
5            GyroscopeDriftNoise             0.0943
5            LinearAccelerationNoise         0.0910
5            LinearAccelerationDecayFactor   0.0901
6            AccelerometerNoise              0.0900
6            GyroscopeNoise                  0.0893
6            GyroscopeDriftNoise             0.0893
6            LinearAccelerationNoise         0.0862
6            LinearAccelerationDecayFactor   0.0855
7            AccelerometerNoise              0.0855
7            GyroscopeNoise                  0.0848
7            GyroscopeDriftNoise             0.0848
7            LinearAccelerationNoise         0.0822
7            LinearAccelerationDecayFactor   0.0818
8            AccelerometerNoise              0.0817
8            GyroscopeNoise                  0.0811
8            GyroscopeDriftNoise             0.0811
8            LinearAccelerationNoise         0.0791
8            LinearAccelerationDecayFactor   0.0789
9            AccelerometerNoise              0.0788
9            GyroscopeNoise                  0.0782
9            GyroscopeDriftNoise             0.0782
9            LinearAccelerationNoise         0.0769
9            LinearAccelerationDecayFactor   0.0768
10           AccelerometerNoise              0.0768
10           GyroscopeNoise                  0.0762
10           GyroscopeDriftNoise             0.0762
10           LinearAccelerationNoise         0.0754
10           LinearAccelerationDecayFactor   0.0753
11           AccelerometerNoise              0.0753
11           GyroscopeNoise                  0.0747
11           GyroscopeDriftNoise             0.0747
11           LinearAccelerationNoise         0.0741
11           LinearAccelerationDecayFactor   0.0740
12           AccelerometerNoise              0.0740
12           GyroscopeNoise                  0.0734
12           GyroscopeDriftNoise             0.0734
12           LinearAccelerationNoise         0.0728
12           LinearAccelerationDecayFactor   0.0728
13           AccelerometerNoise              0.0728
13           GyroscopeNoise                  0.0721
13           GyroscopeDriftNoise             0.0721
```

```
    13          LinearAccelerationNoise          0.0715
    13          LinearAccelerationDecayFactor    0.0715
    14          AccelerometerNoise               0.0715
    14          GyroscopeNoise                   0.0706
    14          GyroscopeDriftNoise              0.0706
    14          LinearAccelerationNoise          0.0700
    14          LinearAccelerationDecayFactor    0.0700
    15          AccelerometerNoise               0.0700
    15          GyroscopeNoise                   0.0690
    15          GyroscopeDriftNoise              0.0690
    15          LinearAccelerationNoise          0.0684
    15          LinearAccelerationDecayFactor    0.0684
    16          AccelerometerNoise               0.0684
    16          GyroscopeNoise                   0.0672
    16          GyroscopeDriftNoise              0.0672
    16          LinearAccelerationNoise          0.0668
    16          LinearAccelerationDecayFactor    0.0667
    17          AccelerometerNoise               0.0667
    17          GyroscopeNoise                   0.0655
    17          GyroscopeDriftNoise              0.0655
    17          LinearAccelerationNoise          0.0654
    17          LinearAccelerationDecayFactor    0.0654
    18          AccelerometerNoise               0.0654
    18          GyroscopeNoise                   0.0641
    18          GyroscopeDriftNoise              0.0641
    18          LinearAccelerationNoise          0.0640
    18          LinearAccelerationDecayFactor    0.0639
    19          AccelerometerNoise               0.0639
    19          GyroscopeNoise                   0.0627
    19          GyroscopeDriftNoise              0.0627
    19          LinearAccelerationNoise          0.0627
    19          LinearAccelerationDecayFactor    0.0624
    20          AccelerometerNoise               0.0624
    20          GyroscopeNoise                   0.0614
    20          GyroscopeDriftNoise              0.0614
    20          LinearAccelerationNoise          0.0613
    20          LinearAccelerationDecayFactor    0.0613
```

Fuse the sensor data again using the tuned filter.

```
qEstTuned = fuse(ld.sensorData.Accelerometer, ...
    ld.sensorData.Gyroscope);
```

Compare the tuned and untuned filter RMS error performances.

```
dUntuned = rad2deg(dist(qEstUntuned, qTrue));
dTuned = rad2deg(dist(qEstTuned, qTrue));
rmsUntuned = sqrt(mean(dUntuned.^2))
```

```
rmsUntuned = 6.5864
```

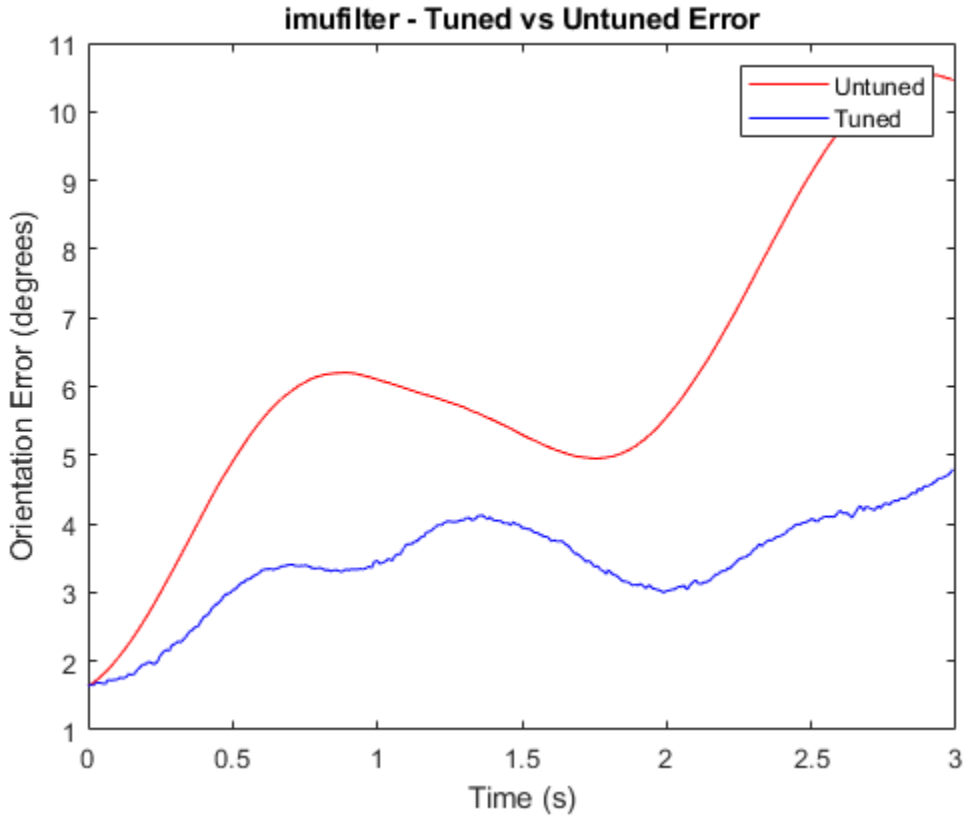
```
rmsTuned = sqrt(mean(dTuned.^2))
```

```
rmsTuned = 3.5098
```

Visualize the results.

```
N = numel(dUntuned);
t = (0:N-1)./ fuse.SampleRate;
```

```
plot(t, dUntuned, 'r', t, dTuned, 'b');
legend('Untuned', 'Tuned');
title('imufilter - Tuned vs Untuned Error')
xlabel('Time (s)');
ylabel('Orientation Error (degrees)');
```



Save Tuned Parameters in MAT File Using Output Function

Load the recorded sensor data and ground truth data.

```
load('insfilterAsyncTuneData.mat');
```

Create timetables for the sensor data and the truth data.

```
sensorData = timetable(Accelerometer, Gyroscope, ...
    Magnetometer, GPSPosition, GPSVelocity, 'SampleRate', 100);
groundTruth = timetable(Orientation, Position, ...
    'SampleRate', 100);
```

Create an `insfilterAsync` filter object that has a few noise properties.

```
filter = insfilterAsync('State', initialState, ...
    'StateCovariance', initialStateCovariance, ...
    'AccelerometerBiasNoise', 1e-7, ...
    'GyroscopeBiasNoise', 1e-7, ...
```

```
    'MagnetometerBiasNoise', 1e-7, ...
    'GeomagneticVectorNoise', 1e-7);
```

Create a tuner configuration object for the filter. Define the `OutputFcn` property as a customized function, `myOutputFcn`, which saves the latest tuned parameters in a MAT file.

```
config = tunerconfig('insfilterAsync', ...
    'MaxIterations',5, ...
    'Display','none', ...
    'OutputFcn', @myOutputFcn);
config.TunableParameters = setdiff(config.TunableParameters, ...
    {'GeomagneticVectorNoise', 'AccelerometerBiasNoise', ...
    'GyroscopeBiasNoise', 'MagnetometerBiasNoise'});
config.TunableParameters
```

```
ans = 1x10 string
  Columns 1 through 3

    "AccelerationNoise"    "AccelerometerNoise"    "AngularVelocityN..."

  Columns 4 through 6

    "GPSPositionNoise"    "GPSVelocityNoise"    "GyroscopeNoise"

  Columns 7 through 9

    "MagnetometerNoise"    "PositionNoise"    "QuaternionNoise"

  Column 10

    "VelocityNoise"
```

Use the tuner noise function to obtain a set of initial sensor noises used in the filter.

```
measNoise = tunernoise('insfilterAsync')
```

```
measNoise = struct with fields:
    AccelerometerNoise: 1
        GyroscopeNoise: 1
     MagnetometerNoise: 1
      GPSPositionNoise: 1
      GPSVelocityNoise: 1
```

Tune the filter and obtain the tuned parameters.

```
tunedParams = tune(filter,measNoise,sensorData,groundTruth,config);
```

Display the save parameters using the saved file.

```
fileObject = matfile('myfile.mat');
fileObject.params
```

```
ans = struct with fields:
         AccelerationNoise: [88.8995 88.8995 88.8995]
    AccelerometerBiasNoise: [1.0000e-07 1.0000e-07 1.0000e-07]
        AccelerometerNoise: 0.7942
      AngularVelocityNoise: [0.0089 0.0089 0.0089]
```

```
          GPSPositionNoise: 1.1664
          GPSVelocityNoise: 0.5210
    GeomagneticVectorNoise: [1.0000e-07 1.0000e-07 1.0000e-07]
        GyroscopeBiasNoise: [1.0000e-07 1.0000e-07 1.0000e-07]
           GyroscopeNoise: 0.5210
     MagnetometerBiasNoise: [1.0000e-07 1.0000e-07 1.0000e-07]
         MagnetometerNoise: 1.0128
             PositionNoise: [5.2100e-07 5.2100e-07 5.2100e-07]
           QuaternionNoise: [1.3239e-06 1.3239e-06 1.3239e-06 1.3239e-06]
         ReferenceLocation: [0 0 0]
                     State: [28x1 double]
           StateCovariance: [28x28 double]
             VelocityNoise: [6.3678e-07 6.3678e-07 6.3678e-07]
```

The output function

```
function stop = myOutputFcn(params, ~)
save('myfile.mat','params'); % overwrite the file with latest
stop = false;
end
```

See Also

`insfilterAsync` | `insfilterNonholonomic` | `insfilterMARG` | `insfilterErrorState` | `ahrsfilter` | `ahrs10filter` | `imufilter`

Introduced in R2020b

validatorOccupancyMap

State validator based on 2-D grid map

Description

The `validatorOccupancyMap` object validates states and discretized motions based on the value in a 2-D occupancy map. An occupied map location is interpreted as an invalid state.

Creation

Syntax

Description

`validator = validatorOccupancyMap` creates a 2-D occupancy map validator associated with an SE2 state space with default settings.

`validator = validatorOccupancyMap(stateSpace)` creates a validator in the given state space definition derived from `nav.StateSpace`.

`validator = validatorOccupancyMap(stateSpace,Name,Value)` specifies the `Map` or `XYIndices` properties using `Name,Value` pair arguments.

Properties

`StateSpace` — State space for validating states
`stateSpaceSE2` (default) | subclass of `nav.StateSpace`

State space for validating states, specified as a subclass of `nav.StateSpace`. Provided state space objects include:

- `stateSpaceSE2`
- `stateSpaceDubins`
- `stateSpaceReedsShepp`

`Map` — Map used for validating states
`binaryOccupancyMap(10,10)` (default) | `binaryOccupancyMap` object | `occupancyMap` object

Map used for validating states, specified as a `binaryOccupancyMap` or `occupancyMap` object.

`ValidationDistance` — Interval for checking state validity
`Inf` (default) | positive numeric scalar

Interval for sampling between states and checking state validity, specified as a positive numeric scalar.

`XYIndices` — State variable mapping for *xy*-coordinates
`[1 2]` (default) | `[xIdx yIdx]`

State variable mapping for *xy*-coordinates in state vector, specified as a two-element vector, `[xIdx yIdx]`. For example, if a state vector is given as `[r p y x y z]`, the *xy*-coordinates are `[4 5]`.

Object Functions

copy	Create deep copy of state validator object
isStateValid	Check if state is valid
isMotionValid	Check if path between states is valid

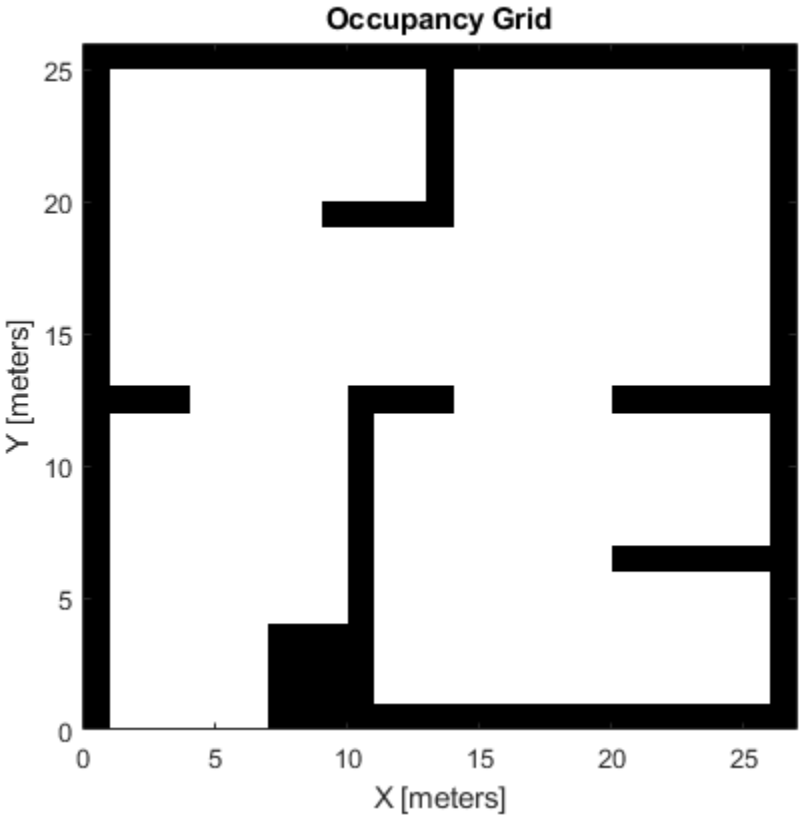
Examples

Validate Path Through Occupancy Map Environment

This example shows how to validate paths through an evironment.

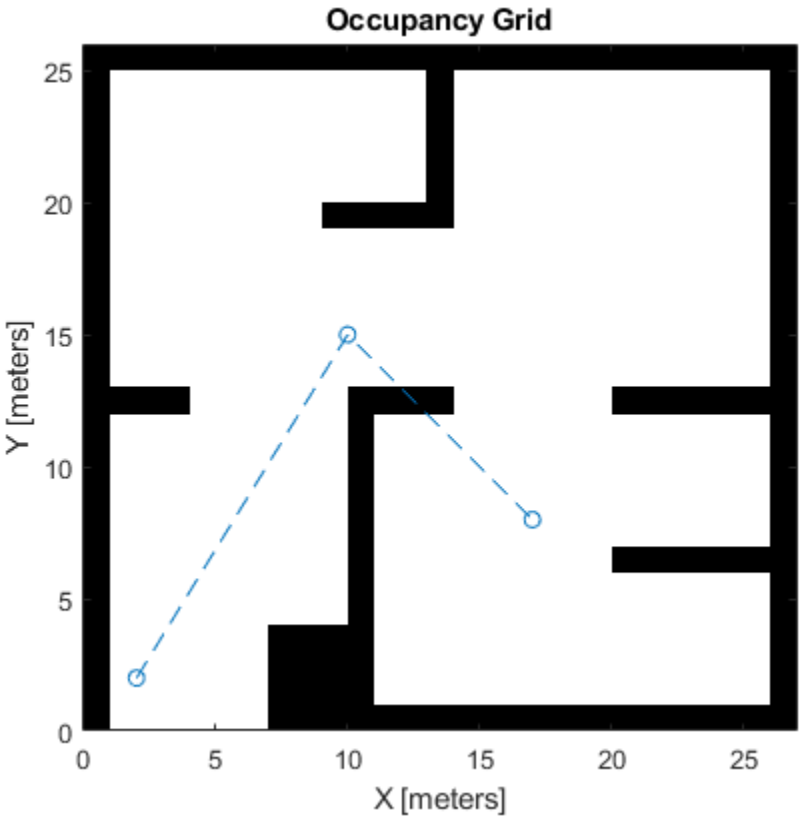
Load example maps. Use the simple map to create a binary occupancy map.

```
load exampleMaps.mat
map = occupancyMap(simpleMap);
show(map)
```



Specify a coarse path through the map.

```
path = [2 2 pi/2; 10 15 0; 17 8 -pi/2];
hold on
plot(path(:,1),path(:,2),"--o")
```



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

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

Check the points of the path are valid states. All three points are in free space, so are considered valid.

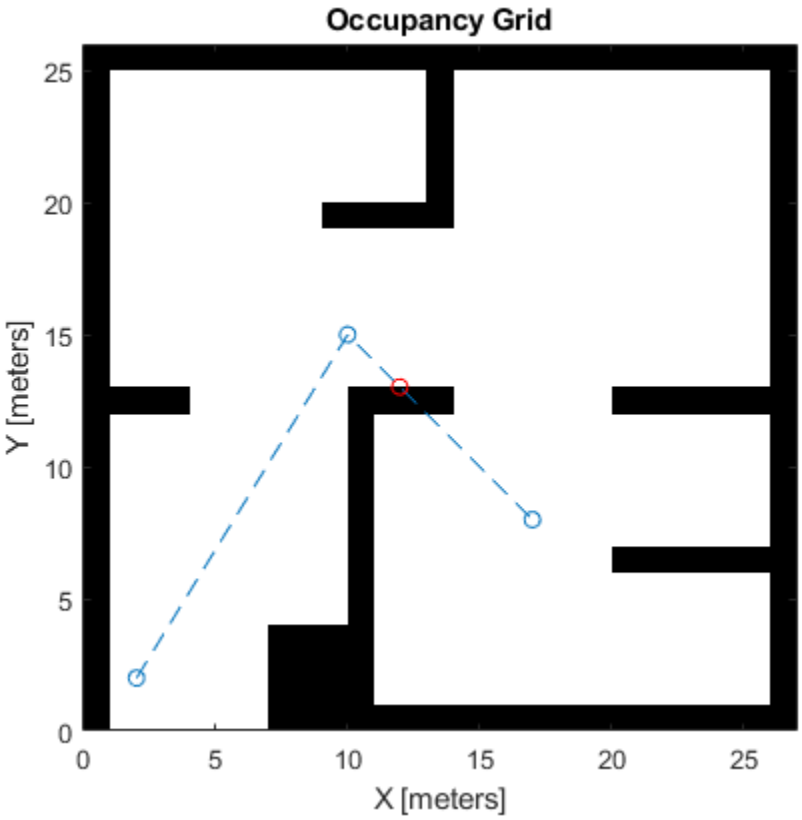
```
isValid = isStateValid(validator,path)
```

```
isValid = 3x1 logical array

   1
   1
   1
```

Check the motion between each sequential path states. The `isMotionValid` function interpolates along the path between states. If a path segment is invalid, plot the last valid point along the path.

```
startStates = [path(1,:);path(2,:)];
endStates = [path(2,:);path(3,:)];
    for i = 1:2
        [isPathValid, lastValid] = isMotionValid(validator,startStates(i,:),endStates(i,:));
        if ~isPathValid
            plot(lastValid(1),lastValid(2),'or')
```

```
        end
    end
hold off
```



Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Specify the `Map` and `XYIndices` properties when your create the object. For example:

```
validator = validatorOccupancyMap('Map',occMap,'XYIndices',[4 5])
```

See Also

`stateSpaceSE2` | `nav.StateSpace` | `nav.StateValidator`

Introduced in R2019b

validatorOccupancyMap3D

State validator based on 3-D grid map

Description

The `validatorOccupancyMap3D` object validates states and discretized motions based on occupancy values in a 3-D occupancy map. The object interprets obstacle-free map locations as valid states. The object interprets occupied and unknown map locations as invalid states.

Creation

Syntax

```
validator = validatorOccupancyMap3D
validator = validatorOccupancyMap3D(stateSpace)
validator = validatorOccupancyMap3D(stateSpace,Name,Value)
```

Description

`validator = validatorOccupancyMap3D` creates a 3-D occupancy map validator associated with an SE(3) state space with default settings.

`validator = validatorOccupancyMap3D(stateSpace)` creates a validator in the specified state space. The `stateSpace` input sets the value of the StateSpace property.

`validator = validatorOccupancyMap3D(stateSpace,Name,Value)` sets properties using one or more name-value pairs. Unspecified properties have default values. Enclose each property name in quotes.

For example, `validatorOccupancyMap3D('ValidationDistance',0.1)` creates a 3-D occupancy map validator with a sampling interval of 0.1.

Properties

StateSpace — State space for validating states
`stateSpaceSE3` object (default) | subclass of `nav.StateSpace`

This property is read-only.

State space for validating states, specified as a subclass of `nav.StateSpace`. These are the predefined state space objects:

- `stateSpaceSE3`
- `stateSpaceSE2`
- `stateSpaceDubins`
- `stateSpaceReedsShepp`

Example: `validatorOccupancyMap3D(stateSpaceSE3)`

`Map` — Map used for validating states
`occupancyMap3D` (default) | `occupancyMap3D` object

Map used for validating states, specified as an `occupancyMap3D` object.

Example: `validator.Map = occupancyMap3D(10)`

`ValidationDistance` — Interval for checking state validity
`Inf` (default) | positive numeric scalar

Interval for sampling between states and checking state validity, specified as a positive numeric scalar.

Example: `validator.ValidationDistance = 0.1`

Data Types: `double`

`XYZIndices` — State variable mapping for *xyz*-coordinates
`[1 2 3]` (default) | three-element vector

State variable mapping for *xyz*-coordinates in the state vector, specified as a three-element vector of form `[xIdx yIdx zIdx]`.

Data Types: `double`

Object Functions

copy	Create deep copy of state validator object
isMotionValid	Check if path between states is valid
isStateValid	Check if state is valid

Examples

Validate Path Through 3-D Occupancy Map Environment

Create a 3-D occupancy map and associated state validator. Plan, validate, and visualize a path through the occupancy map.

Load and Assign Map to State Validator

Load a 3-D occupancy map of a city block into the workspace. Specify a threshold for which cells to consider as obstacle-free.

```
mapData = load('dMapCityBlock.mat');
omap = mapData.omap;
omap.FreeThreshold = 0.5;
```

Inflate the occupancy map to add a buffer zone for safe operation around the obstacles.

```
inflate(omap,1)
```

Create an SE(3) state space object with bounds for state variables.

```
ss = stateSpaceSE3([-20 220;
    -20 220;
```

```
    -10 100;
    inf inf;
    inf inf;
    inf inf;
    inf inf]);
```

Create a 3-D occupancy map state validator using the created state space.

```
sv = validatorOccupancyMap3D(ss);
```

Assign the occupancy map to the state validator object. Specify the sampling distance interval.

```
sv.Map = omap;
sv.ValidationDistance = 0.1;
```

Plan and Visualize Path

Create a path planner with increased maximum connection distance. Reduce the maximum number of iterations.

```
planner = plannerRRT(ss,sv);
planner.MaxConnectionDistance = 50;
planner.MaxIterations = 1000;
```

Create a user-defined evaluation function for determining whether the path reaches the goal. Specify the probability of choosing the goal state during sampling.

```
planner.GoalReachedFcn = @(~,x,y)(norm(x(1:3)-y(1:3))<5);
planner.GoalBias = 0.1;
```

Set the start and goal states.

```
start = [40 180 25 0.7 0.2 0 0.1];
goal = [150 33 35 0.3 0 0.1 0.6];
```

Plan a path using the specified start, goal, and planner.

```
[pthObj,solnInfo] = plan(planner,start,goal);
```

Check that the points of the path are valid states.

```
isValid = isStateValid(sv,pthObj.States)
```

```
isValid = 7x1 logical array

   1
   1
   1
   1
   1
   1
   1
```

Check that the motion between each sequential path state is valid.

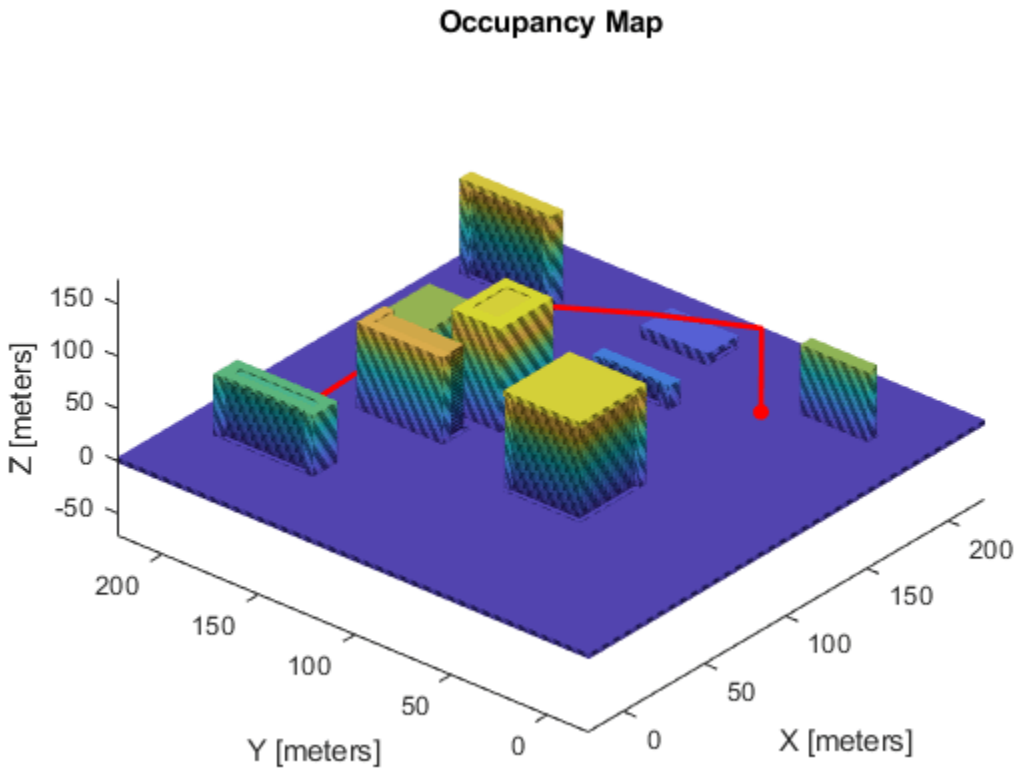
```
isPathValid = zeros(size(pthObj.States,1)-1,1,'logical');
for i = 1:size(pthObj.States,1)-1
    [isPathValid(i),~] = isMotionValid(sv,pthObj.States(i,:),...
```

```
        pthObj.States(i+1,:));
end
isPathValid
```

```
isPathValid = 6x1 logical array

   1
   1
   1
   1
   1
   1
```

Visualize the results.

```
show(omap)
hold on
scatter3(start(1,1),start(1,2),start(1,3),'g','filled') % draw start state
scatter3(goal(1,1),goal(1,2),goal(1,3),'r','filled')    % draw goal state
plot3(pthObj.States(:,1),pthObj.States(:,2),pthObj.States(:,3),...
    'r-','LineWidth',2) % draw path
```



Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`occupancyMap3D` | `stateSpaceSE3` | `validatorOccupancyMap`

Introduced in R2020b

validatorVehicleCostmap

State validator based on 2-D costmap

Description

The `validatorOccupancyMap` object validates states and discretized motions based on the value in a 2-D costmap. An occupied map location is interpreted as an invalid state.

Creation

Syntax

Description

`validator = validatorVehicleCostmap` creates a vehicle cost map validator associated with an SE2 state space with default settings.

`validator = validatorVehicleCostmap(stateSpace)` creates a validator in the given state space definition derived from `nav.StateSpace`.

`validator = validatorVehicleCostmap(stateSpace,xyIndices)`sets the `XYIndices` property to specify which variables in the state vector define the *xy*-coordinates.

`validator = validatorVehicleCostmap(stateSpace,Name,Value)` specifies the `Map` or `XYIndices` properties using `Name,Value` pair arguments.

Properties

`StateSpace` — State space for validating states
`stateSpaceSE2` (default) | subclass of `nav.StateSpace`

State space for validating states, specified as a subclass of `nav.StateSpace`. Provided state space objects include:

- `stateSpaceSE2`
- `stateSpaceDubins`
- `stateSpaceReedsShepp`

`Map` — Map used for validating states
`vehicleCostMap(10,10)` (default) | `vehicleCostMap` object

Map used for validating states, specified as a `vehicleCostMap` object.

`ValidationDistance` — Interval for checking state validity
`Inf` (default) | positive numeric scalar

Interval for sampling between states and checking state validity, specified as a positive numeric scalar.

`XYIndices` — State variable mapping for *xy*-coordinates
`[1 2]` (default) | `[xIdx yIdx]`

State variable mapping for *xy*-coordinates in state vector, specified as a two-element vector, `[xIdx yIdx]`. For example, if a state vector is given as `[r p y x y z]`, the *xy*-coordinates are `[4 5]`.

`ThetaIndex` — State variable mapping for *theta* coordinate
`NaN` (default) | positive integer

State variable mapping for *theta* coordinate in state vector, specified as a positive integer. For example, if a state vector is given as `[x y theta]`, the *theta* coordinate is `3`.

Object Functions

copy	Create deep copy of state validator object
isStateValid	Check if state is valid
isMotionValid	Check if path between states is valid

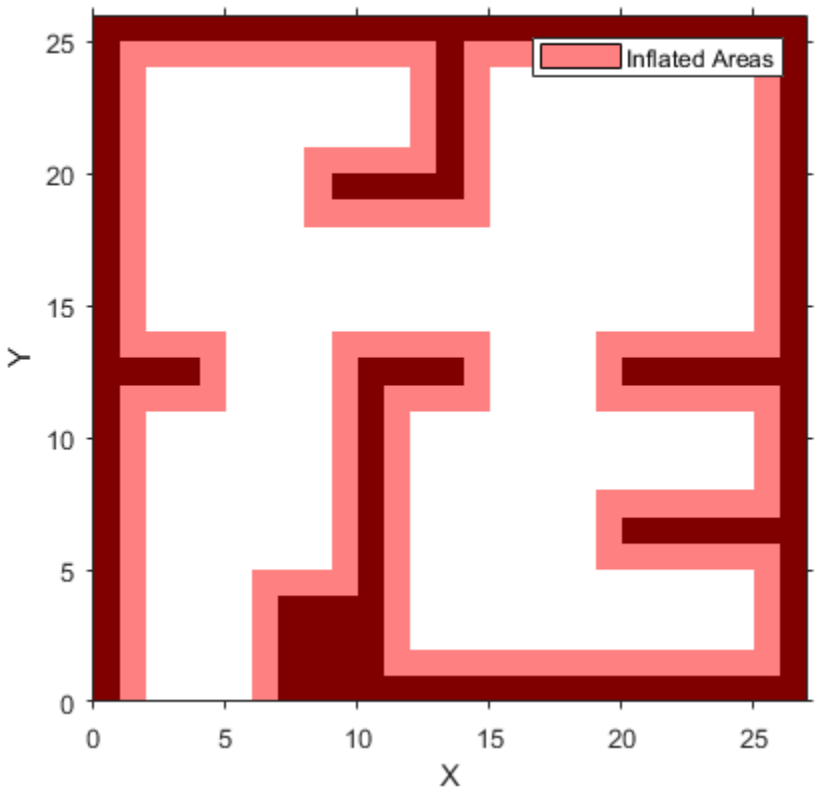
Examples

Validate Path Through Vehicle Costmap Environment

This example shows how to validate paths through an evironment.

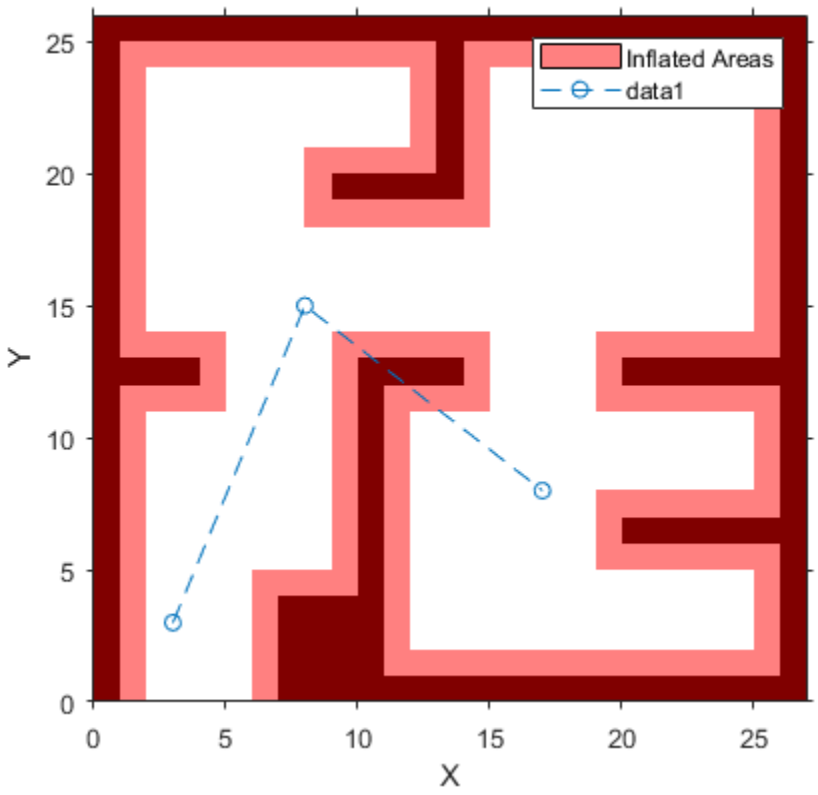
Load example maps. Use the simple map to create a vehicle cost map. Specify an inflation raidus of 1 meter.

```
load exampleMaps.mat
map = vehicleCostmap(double(simpleMap));
map.CollisionChecker = inflationCollisionChecker("InflationRadius",1);
plot(map)
```



Specify a coarse path through the map.

```
path = [3 3 pi/2; 8 15 0; 17 8 -pi/2];
hold on
plot(path(:,1),path(:,2),"--o")
```



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

```
validator = validatorVehicleCostmap(stateSpaceSE2);
validator.Map = map;
validator.ValidationDistance = 0.1;
```

Check the points of the path are valid states. All three points are in free space, so are considered valid.

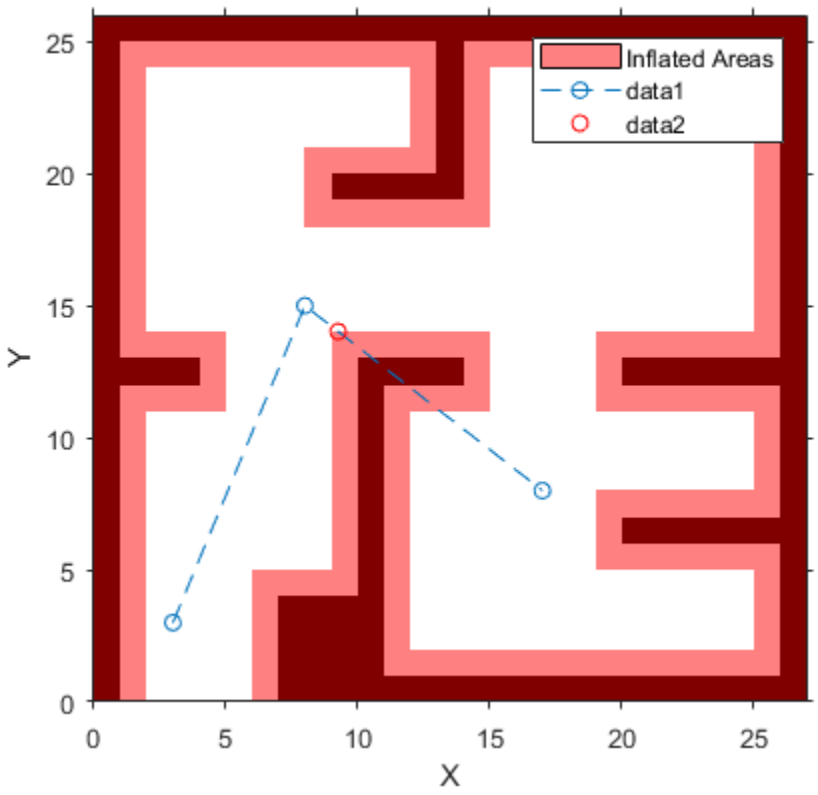
```
isValid = isStateValid(validator,path)
```

```
isValid = 3x1 logical array

   1
   1
   1
```

Check the motion between each sequential path states. The `isMotionValid` function interpolates along the path between states. If a path segment is invalid, plot the last valid point along the path.

```
startStates = [path(1,:);path(2,:)];
endStates = [path(2,:);path(3,:)];
    for i = 1:2
        [isPathValid, lastValid] = isMotionValid(validator,startStates(i,:),endStates(i,:));
        if ~isPathValid
            plot(lastValid(1),lastValid(2),'or')
```

```
        end
    end
hold off
```



Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

For code generation, the map used inside the object must remain constant.

See Also

`validatorOccupancyMap` | `stateSpaceSE2` | `nav.StateSpace` | `nav.StateValidator`

Introduced in R2019b

copy

Create deep copy of state validator object

Syntax

```
validator2 = copy(validator1)
```

Description

`validator2 = copy(validator1)` creates a deep copy of the specified state validator object.

Examples

Create Deep Copy of 3-D Occupancy Map State Validator Object

Create a validator object and set a custom validation distance.

```
validator = validatorOccupancyMap3D;
validator.ValidationDistance = 4.5
```

```
validator =
  validatorOccupancyMap3D with properties:

                  Map: [1x1 occupancyMap3D]
           StateSpace: [1x1 stateSpaceSE3]
           XYZIndices: [1 2 3]
    ValidationDistance: 4.5000
```

Create a deep copy of the state validator object.

```
validator2 = copy(validator)
```

```
validator2 =
  validatorOccupancyMap3D with properties:

                  Map: [1x1 occupancyMap3D]
           StateSpace: [1x1 stateSpaceSE3]
           XYZIndices: [1 2 3]
    ValidationDistance: 4.5000
```

Verify that the `ValidationDistance` property values of the two state validator objects are equal.

```
isequal(validator.ValidationDistance,validator2.ValidationDistance)
```

```
ans = logical
   1
```

Input Arguments

validator1 — State validator object
`validatorOccupancyMap` object | `validatorOccupancyMap3D` object | `validatorVehicleCostmap` object

State validator object, specified as a `validatorOccupancyMap`, `validatorOccupancyMap3D`, or `validatorVehicleCostmap` object.

Output Arguments

validator2 — State validator object
`validatorOccupancyMap` object | `validatorOccupancyMap3D` object | `validatorVehicleCostmap` object

State validator object, returned as a `validatorOccupancyMap`, `validatorOccupancyMap3D`, or `validatorVehicleCostmap` object.

See Also
`validatorOccupancyMap` | `validatorOccupancyMap3D` | `validatorVehicleCostmap`

Introduced in R2020b

isMotionValid

Check if path between states is valid

Syntax

```
[isValid,lastValid] = isMotionValid(validator,state1,state2)
```

Description

`[isValid,lastValid] = isMotionValid(validator,state1,state2)` checks if the path between two states is valid by interpolating between states. The function also returns the last valid state along the path.

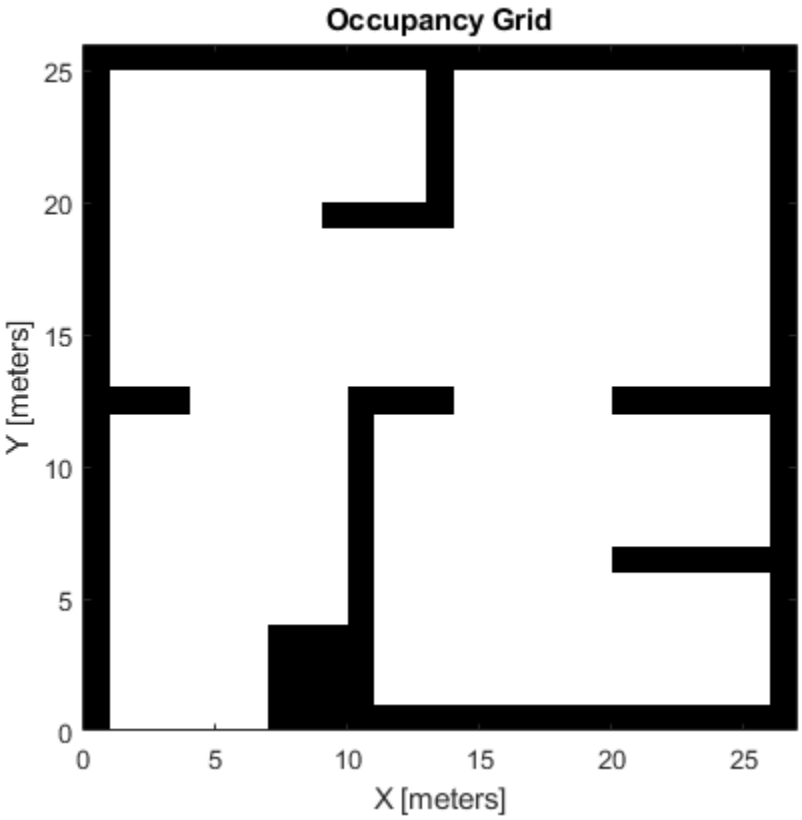
Examples

Validate Path Through Occupancy Map Environment

This example shows how to validate paths through an evironment.

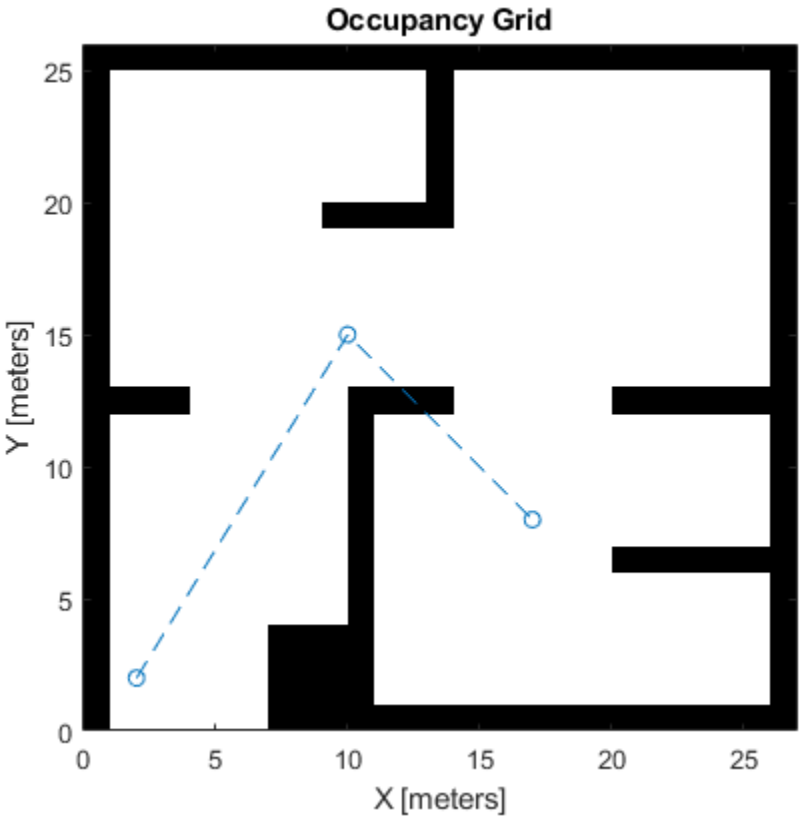
Load example maps. Use the simple map to create a binary occupancy map.

```
load exampleMaps.mat
map = occupancyMap(simpleMap);
show(map)
```



Specify a coarse path through the map.

```
path = [2 2 pi/2; 10 15 0; 17 8 -pi/2];
hold on
plot(path(:,1),path(:,2),"--o")
```



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

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

Check the points of the path are valid states. All three points are in free space, so are considered valid.

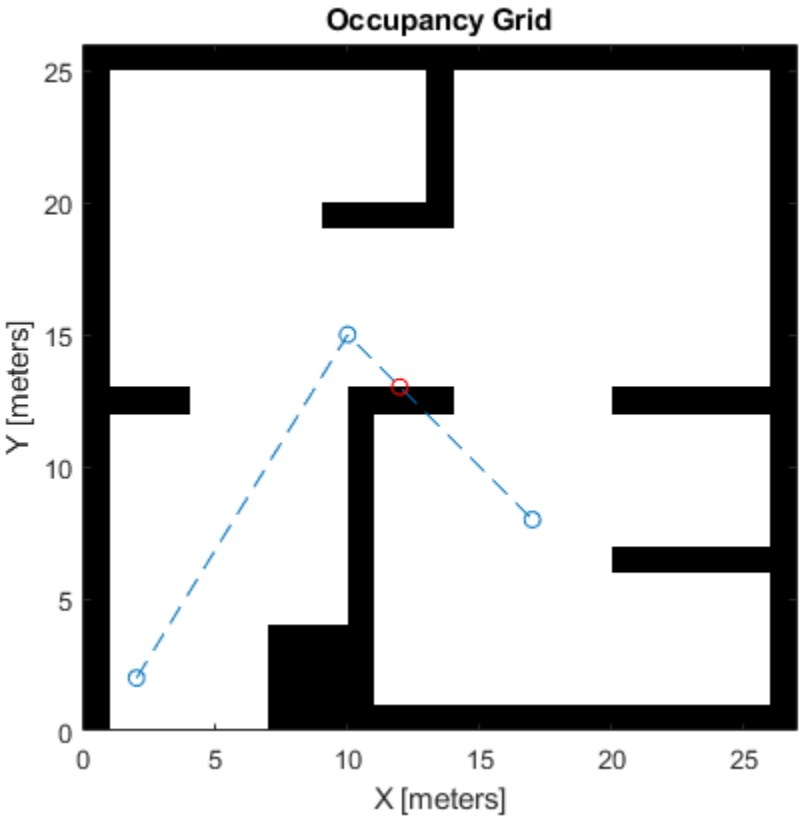
```
isValid = isStateValid(validator,path)
```

```
isValid = 3x1 logical array

   1
   1
   1
```

Check the motion between each sequential path states. The `isMotionValid` function interpolates along the path between states. If a path segment is invalid, plot the last valid point along the path.

```
startStates = [path(1,:);path(2,:)];
endStates = [path(2,:);path(3,:)];
    for i = 1:2
        [isPathValid, lastValid] = isMotionValid(validator,startStates(i,:),endStates(i,:));
        if ~isPathValid
            plot(lastValid(1),lastValid(2),'or')
```

```
        end
    end
hold off
```

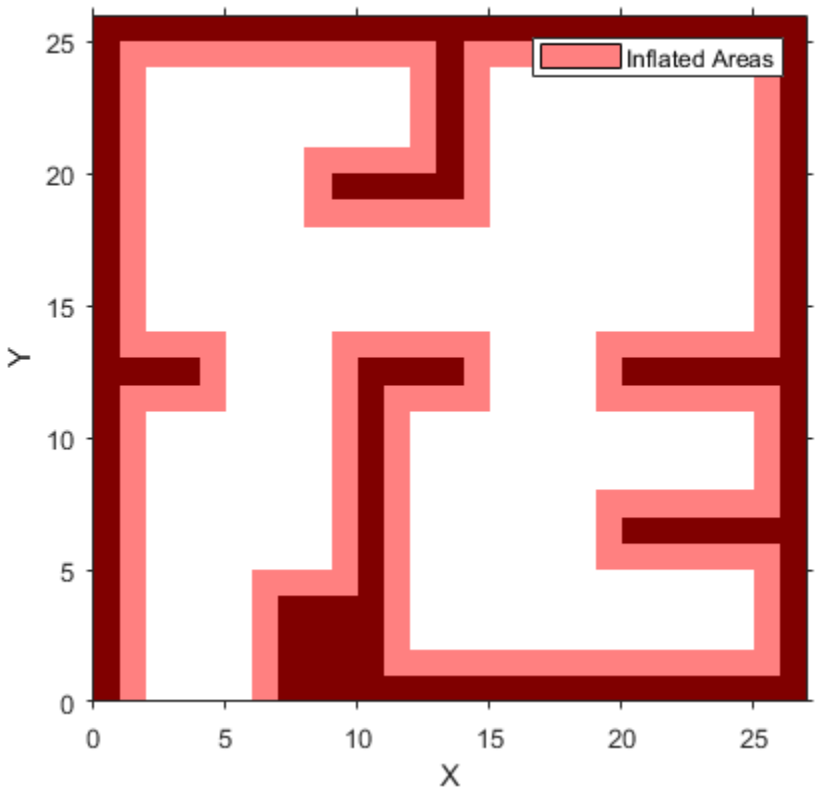


Validate Path Through Vehicle Costmap Environment

This example shows how to validate paths through an evironment.

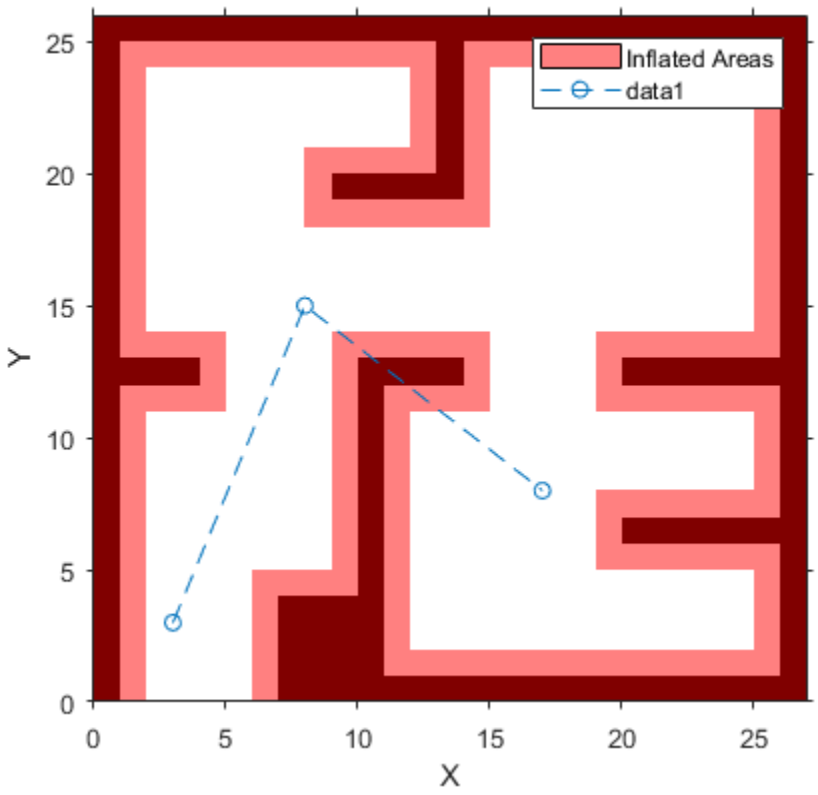
Load example maps. Use the simple map to create a vehicle cost map. Specify an inflation raidus of 1 meter.

```
load exampleMaps.mat
map = vehicleCostmap(double(simpleMap));
map.CollisionChecker = inflationCollisionChecker("InflationRadius",1);
plot(map)
```



Specify a coarse path through the map.

```
path = [3 3 pi/2; 8 15 0; 17 8 -pi/2];
hold on
plot(path(:,1),path(:,2),"--o")
```



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

```
validator = validatorVehicleCostmap(stateSpaceSE2);
validator.Map = map;
validator.ValidationDistance = 0.1;
```

Check the points of the path are valid states. All three points are in free space, so are considered valid.

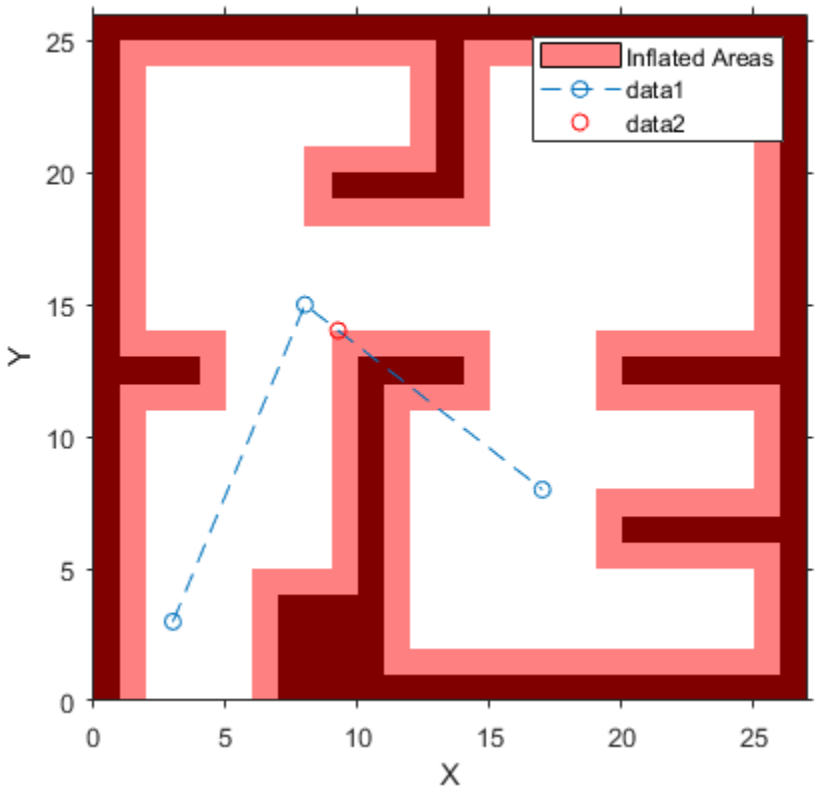
```
isValid = isStateValid(validator,path)
```

```
isValid = 3x1 logical array

   1
   1
   1
```

Check the motion between each sequential path states. The `isMotionValid` function interpolates along the path between states. If a path segment is invalid, plot the last valid point along the path.

```
startStates = [path(1,:);path(2,:)];
endStates = [path(2,:);path(3,:)];
    for i = 1:2
        [isPathValid, lastValid] = isMotionValid(validator,startStates(i,:),endStates(i,:));
        if ~isPathValid
            plot(lastValid(1),lastValid(2),'or')
```

```
        end
    end
hold off
```



Validate Path Through 3-D Occupancy Map Environment

Create a 3-D occupancy map and associated state validator. Plan, validate, and visualize a path through the occupancy map.

Load and Assign Map to State Validator

Load a 3-D occupancy map of a city block into the workspace. Specify a threshold for which cells to consider as obstacle-free.

```
mapData = load('dMapCityBlock.mat');
omap = mapData.omap;
omap.FreeThreshold = 0.5;
```

Inflate the occupancy map to add a buffer zone for safe operation around the obstacles.

```
inflate(omap,1)
```

Create an SE(3) state space object with bounds for state variables.

```
ss = stateSpaceSE3([-20 220;
    -20 220;
```

```
    -10 100;
    inf inf;
    inf inf;
    inf inf;
    inf inf]);
```

Create a 3-D occupancy map state validator using the created state space.

```
sv = validatorOccupancyMap3D(ss);
```

Assign the occupancy map to the state validator object. Specify the sampling distance interval.

```
sv.Map = omap;
sv.ValidationDistance = 0.1;
```

Plan and Visualize Path

Create a path planner with increased maximum connection distance. Reduce the maximum number of iterations.

```
planner = plannerRRT(ss,sv);
planner.MaxConnectionDistance = 50;
planner.MaxIterations = 1000;
```

Create a user-defined evaluation function for determining whether the path reaches the goal. Specify the probability of choosing the goal state during sampling.

```
planner.GoalReachedFcn = @(~,x,y)(norm(x(1:3)-y(1:3))<5);
planner.GoalBias = 0.1;
```

Set the start and goal states.

```
start = [40 180 25 0.7 0.2 0 0.1];
goal = [150 33 35 0.3 0 0.1 0.6];
```

Plan a path using the specified start, goal, and planner.

```
[pthObj,solnInfo] = plan(planner,start,goal);
```

Check that the points of the path are valid states.

```
isValid = isStateValid(sv,pthObj.States)
```

```
isValid = 7x1 logical array

   1
   1
   1
   1
   1
   1
   1
```

Check that the motion between each sequential path state is valid.

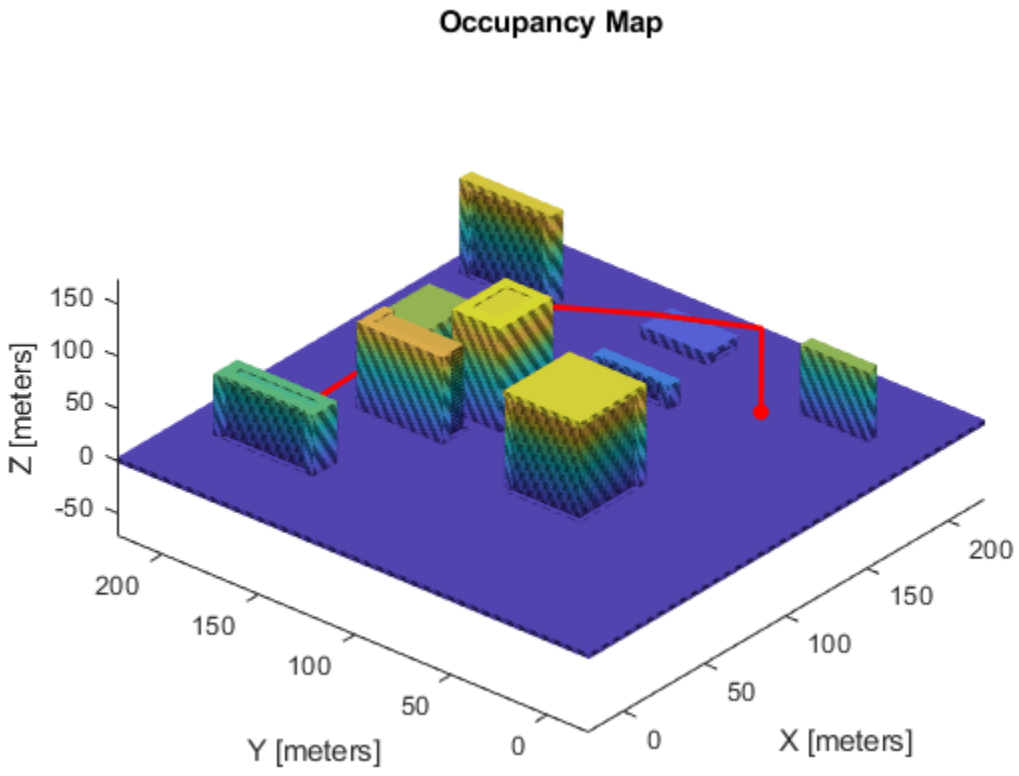
```
isPathValid = zeros(size(pthObj.States,1)-1,1,'logical');
for i = 1:size(pthObj.States,1)-1
    [isPathValid(i),~] = isMotionValid(sv,pthObj.States(i,:),...
```

```
        pthObj.States(i+1,:));
end
isPathValid
```

```
isPathValid = 6x1 logical array

   1
   1
   1
   1
   1
   1
```

Visualize the results.

```
show(omap)
hold on
scatter3(start(1,1),start(1,2),start(1,3),'g','filled') % draw start state
scatter3(goal(1,1),goal(1,2),goal(1,3),'r','filled')    % draw goal state
plot3(pthObj.States(:,1),pthObj.States(:,2),pthObj.States(:,3),...
    'r-','LineWidth',2) % draw path
```



Input Arguments

`validator` — State validator object
object of subclass of `nav.StateValidator`

State validator object, specified as an object of subclass of `nav.StateValidator`. These are the predefined state validator objects:

- `validatorOccupancyMap`
- `validatorVehicleCostmap`
- `validatorOccupancyMap3D`

`state1` — Initial state positions
n-element row vector | *m*-by-*n* matrix

Initial state positions, specified as an *n*-element row vector or *m*-by-*n* matrix. *n* is the dimension of the state space specified in `validator`. *m* is the number of states to validate.

Data Types: `single` | `double`

`state2` — Final state positions
n-element row vector | *m*-by-*n* matrix

Final state positions, specified as an *n*-element row vector or *m*-by-*n* matrix. *n* is the dimension of the state space specified in `validator`. *m* is the number of states to validate.

Data Types: `single` | `double`

Output Arguments

`isValid` — Valid states
m-element logical column vector

Valid states, returned as an *m*-element logical column vector.

Data Types: `logical`

`lastValid` — Final valid state along each path
n-element row vector | *m*-by-*n* matrix

Final valid state along each path, returned as an *n*-element row vector or *m*-by-*n* matrix. *n* is the dimension of the state space specified in the state space property in `validator`. *m* is the number of paths validated. Each row contains the final valid state along the associated path.

Data Types: `single` | `double`

See Also

`isStateValid` | `stateSpaceSE2` | `nav.StateSpace` | `nav.StateValidator`

Introduced in R2019b

isStateValid

Check if state is valid

Syntax

```
isValid = isStateValid(validator,states)
```

Description

`isValid = isStateValid(validator,states)` checks if a set of given states are valid.

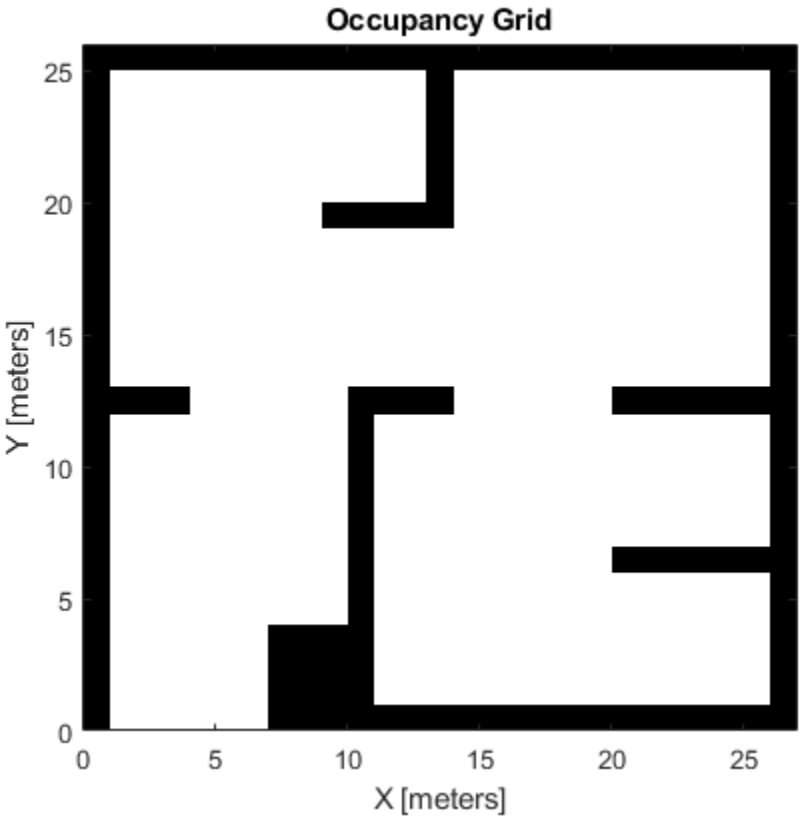
Examples

Validate Path Through Occupancy Map Environment

This example shows how to validate paths through an evironment.

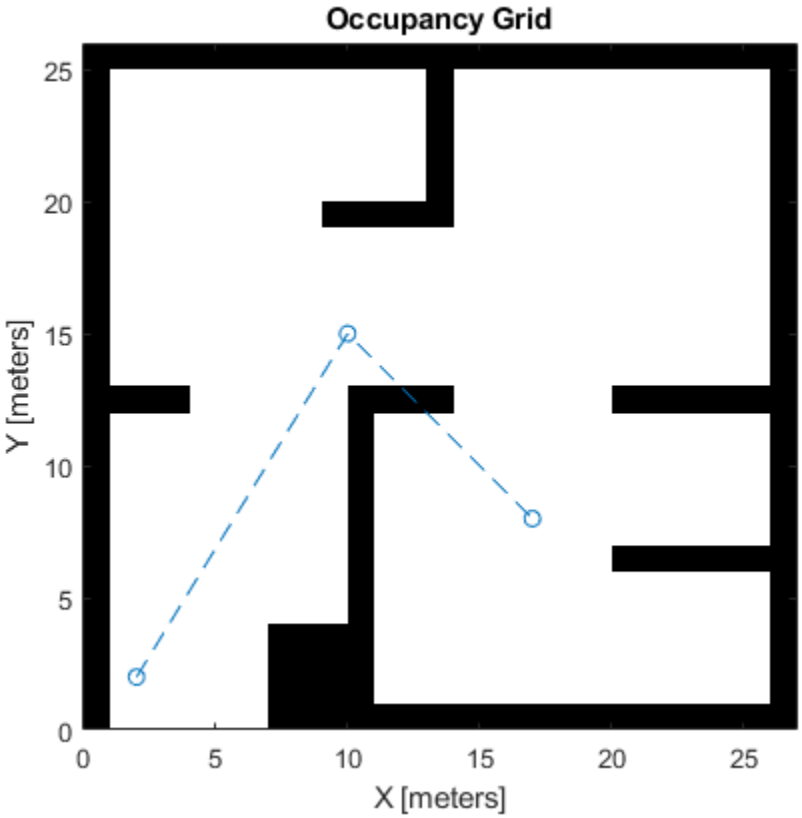
Load example maps. Use the simple map to create a binary occupancy map.

```
load exampleMaps.mat
map = occupancyMap(simpleMap);
show(map)
```



Specify a coarse path through the map.

```
path = [2 2 pi/2; 10 15 0; 17 8 -pi/2];
hold on
plot(path(:,1),path(:,2),"--o")
```



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

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

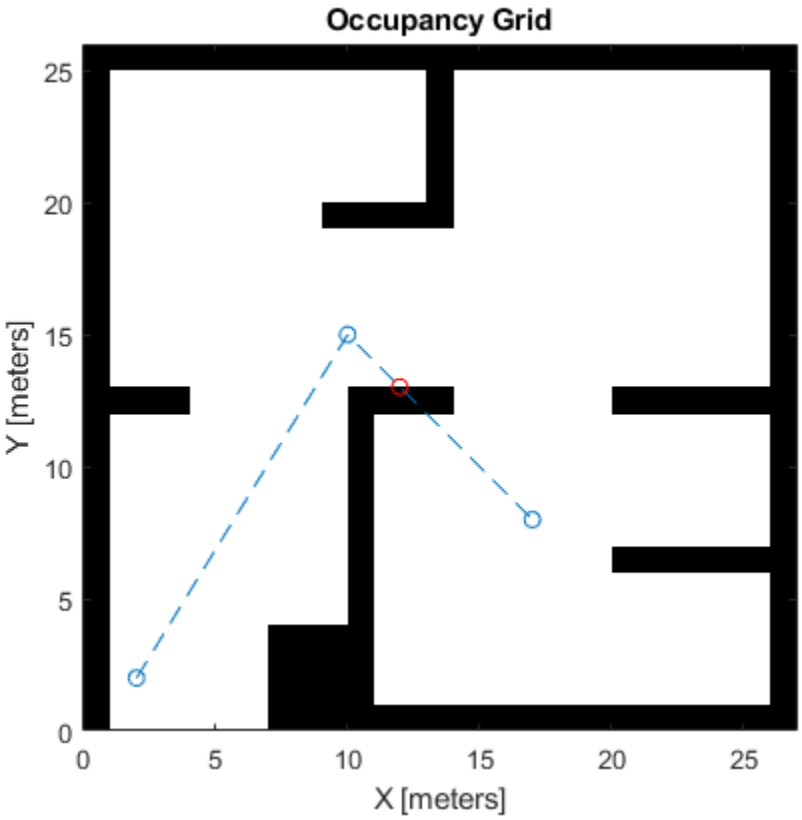
Check the points of the path are valid states. All three points are in free space, so are considered valid.

```
isValid = isStateValid(validator,path)
```

```
isValid = 3x1 logical array

   1
   1
   1
```

Check the motion between each sequential path states. The `isMotionValid` function interpolates along the path between states. If a path segment is invalid, plot the last valid point along the path.

```
startStates = [path(1,:);path(2,:)];
endStates = [path(2,:);path(3,:)];
    for i = 1:2
        [isPathValid, lastValid] = isMotionValid(validator,startStates(i,:),endStates(i,:));
        if ~isPathValid
            plot(lastValid(1),lastValid(2),'or')
        end
    end
hold off
```

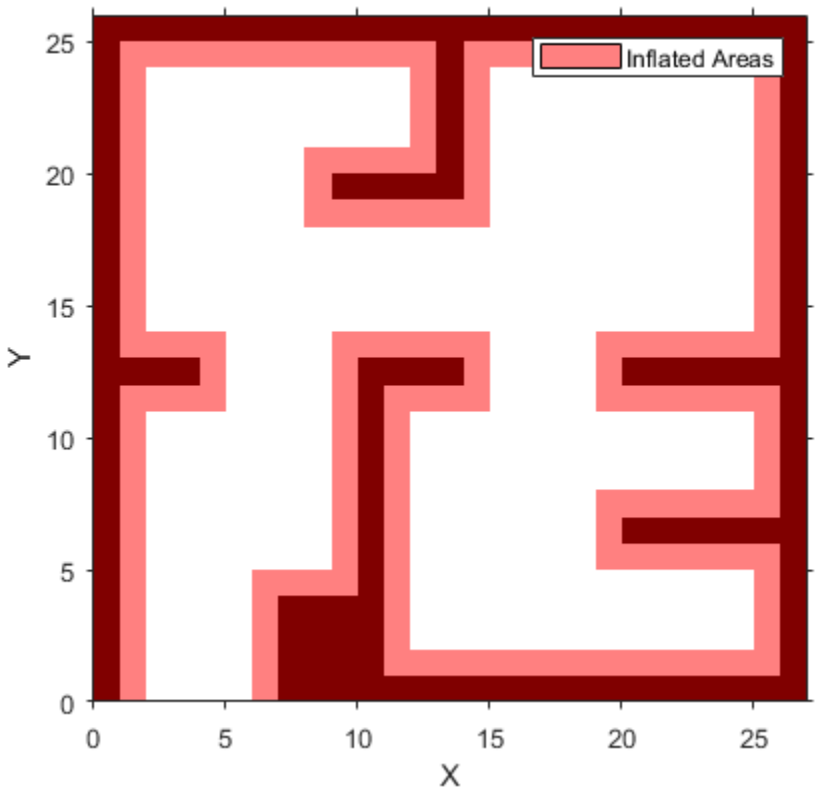


Validate Path Through Vehicle Costmap Environment

This example shows how to validate paths through an evironment.

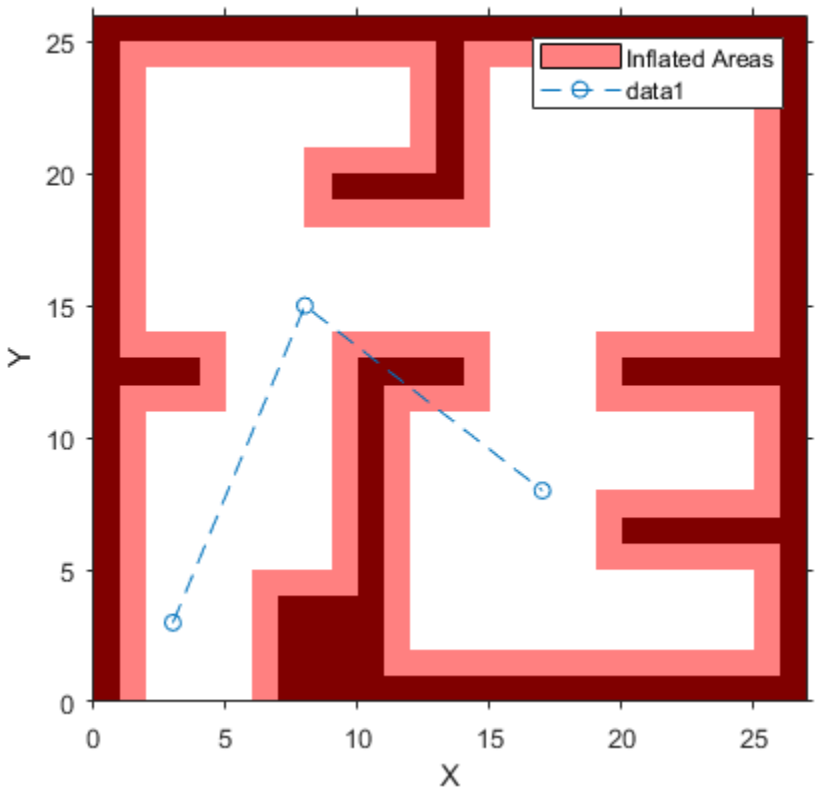
Load example maps. Use the simple map to create a vehicle cost map. Specify an inflation raidus of 1 meter.

```
load exampleMaps.mat
map = vehicleCostmap(double(simpleMap));
map.CollisionChecker = inflationCollisionChecker("InflationRadius",1);
plot(map)
```



Specify a coarse path through the map.

```
path = [3 3 pi/2; 8 15 0; 17 8 -pi/2];
hold on
plot(path(:,1),path(:,2),"--o")
```



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

```
validator = validatorVehicleCostmap(stateSpaceSE2);
validator.Map = map;
validator.ValidationDistance = 0.1;
```

Check the points of the path are valid states. All three points are in free space, so are considered valid.

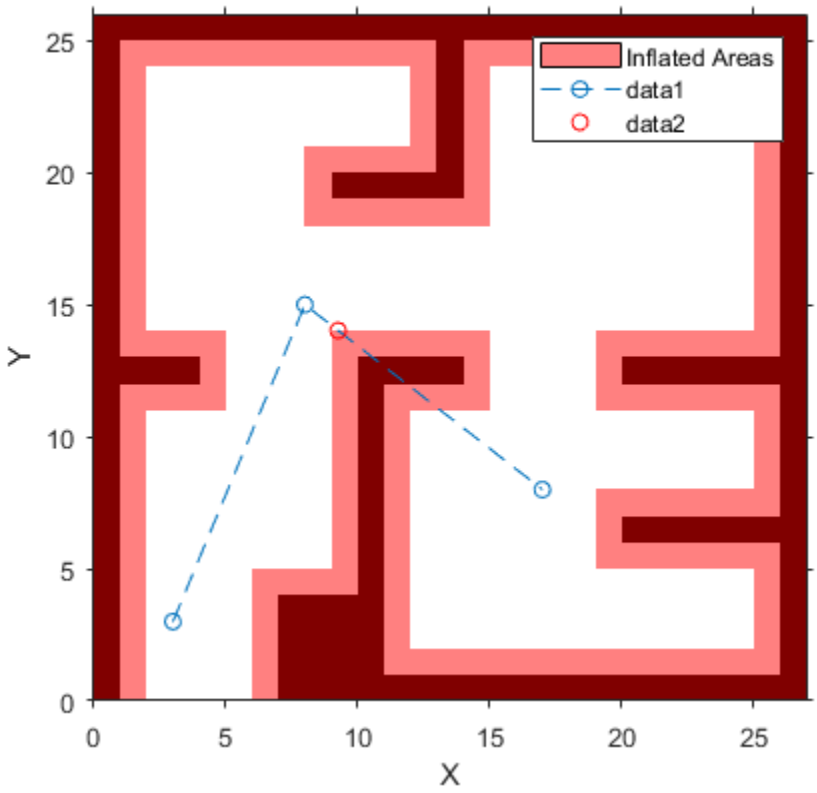
```
isValid = isStateValid(validator,path)
```

```
isValid = 3x1 logical array

   1
   1
   1
```

Check the motion between each sequential path states. The `isMotionValid` function interpolates along the path between states. If a path segment is invalid, plot the last valid point along the path.

```
startStates = [path(1,:);path(2,:)];
endStates = [path(2,:);path(3,:)];
    for i = 1:2
        [isPathValid, lastValid] = isMotionValid(validator,startStates(i,:),endStates(i,:));
        if ~isPathValid
            plot(lastValid(1),lastValid(2),'or')
```

```
        end
    end
hold off
```



Validate Path Through 3-D Occupancy Map Environment

Create a 3-D occupancy map and associated state validator. Plan, validate, and visualize a path through the occupancy map.

Load and Assign Map to State Validator

Load a 3-D occupancy map of a city block into the workspace. Specify a threshold for which cells to consider as obstacle-free.

```
mapData = load('dMapCityBlock.mat');
omap = mapData.omap;
omap.FreeThreshold = 0.5;
```

Inflate the occupancy map to add a buffer zone for safe operation around the obstacles.

```
inflate(omap,1)
```

Create an SE(3) state space object with bounds for state variables.

```
ss = stateSpaceSE3([-20 220;
    -20 220;
```

```
    -10 100;
    inf inf;
    inf inf;
    inf inf;
    inf inf]);
```

Create a 3-D occupancy map state validator using the created state space.

```
sv = validatorOccupancyMap3D(ss);
```

Assign the occupancy map to the state validator object. Specify the sampling distance interval.

```
sv.Map = omap;
sv.ValidationDistance = 0.1;
```

Plan and Visualize Path

Create a path planner with increased maximum connection distance. Reduce the maximum number of iterations.

```
planner = plannerRRT(ss,sv);
planner.MaxConnectionDistance = 50;
planner.MaxIterations = 1000;
```

Create a user-defined evaluation function for determining whether the path reaches the goal. Specify the probability of choosing the goal state during sampling.

```
planner.GoalReachedFcn = @(~,x,y)(norm(x(1:3)-y(1:3))<5);
planner.GoalBias = 0.1;
```

Set the start and goal states.

```
start = [40 180 25 0.7 0.2 0 0.1];
goal = [150 33 35 0.3 0 0.1 0.6];
```

Plan a path using the specified start, goal, and planner.

```
[pthObj,solnInfo] = plan(planner,start,goal);
```

Check that the points of the path are valid states.

```
isValid = isStateValid(sv,pthObj.States)
```

```
isValid = 7x1 logical array

   1
   1
   1
   1
   1
   1
   1
```

Check that the motion between each sequential path state is valid.

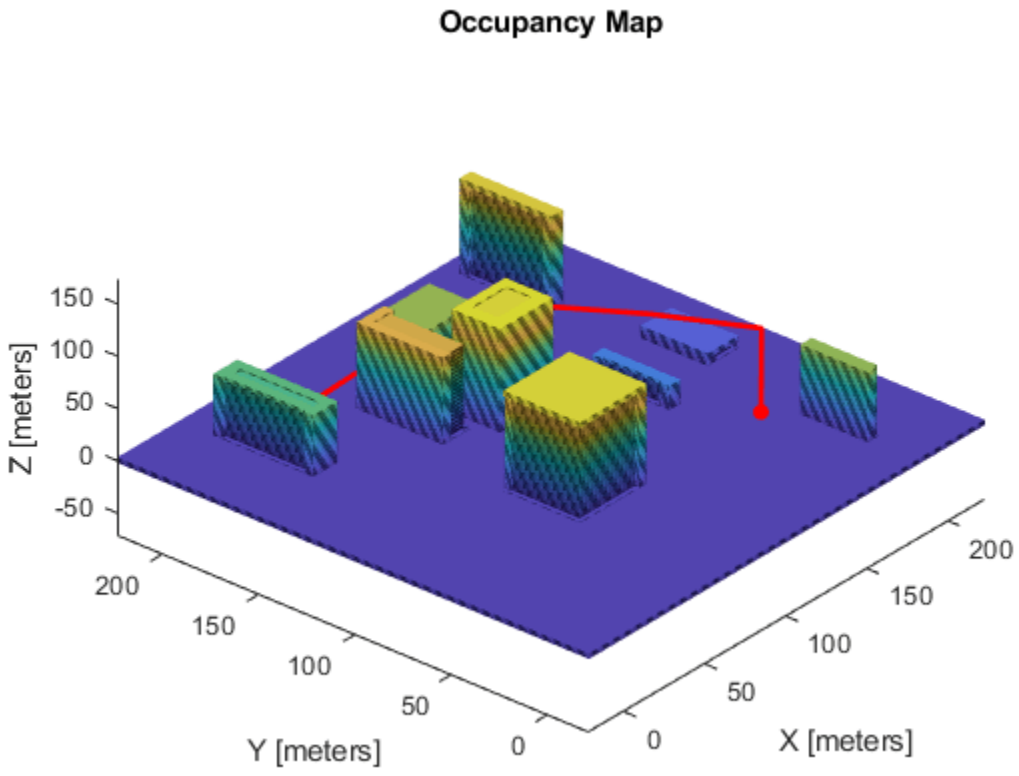
```
isPathValid = zeros(size(pthObj.States,1)-1,1,'logical');
for i = 1:size(pthObj.States,1)-1
    [isPathValid(i),~] = isMotionValid(sv,pthObj.States(i,:),...
```

```
        pthObj.States(i+1,:));
end
isPathValid
```

```
isPathValid = 6x1 logical array

   1
   1
   1
   1
   1
   1
```

Visualize the results.

```
show(omap)
hold on
scatter3(start(1,1),start(1,2),start(1,3),'g','filled') % draw start state
scatter3(goal(1,1),goal(1,2),goal(1,3),'r','filled')    % draw goal state
plot3(pthObj.States(:,1),pthObj.States(:,2),pthObj.States(:,3),...
    'r-','LineWidth',2) % draw path
```



Input Arguments

`validator` — State validator object
object of subclass of `nav.StateValidator`

State validator object, specified as an object of subclass of `nav.StateValidator`. These are the predefined state validator objects:

- `validatorOccupancyMap`
- `validatorVehicleCostmap`
- `validatorOccupancyMap3D`

`states` — State positions

n-element row vector | *m*-by-*n* matrix

State positions, specified as an *n*-element row vector or *m*-by-*n* matrix. *n* is the dimension of the state space specified in `validator`. *m* is the number of states to validate.

Data Types: `single` | `double`

Output Arguments

`isValid` — Valid states

m-element logical column vector

Valid states, returned as an *m*-element logical column vector.

Data Types: `logical`

See Also

`isMotionValid` | `stateSpaceSE2` | `nav.StateSpace` | `nav.StateValidator`

Introduced in R2019b

waypointTrajectory

Waypoint trajectory generator

Description

The `waypointTrajectory` System object generates trajectories using specified waypoints. When you create the System object, you can optionally specify the time of arrival, velocity, and orientation at each waypoint. See “Algorithms” on page 2-1431 for more details.

To generate a trajectory from waypoints:

1. Create the `waypointTrajectory` object and set its properties.
2. Call the object as if it were a function.

To learn more about how System objects work, see What Are System Objects?.

Creation

Syntax

```
trajectory = waypointTrajectory
trajectory = waypointTrajectory(Waypoints,TimeOfArrival)
trajectory = waypointTrajectory(Waypoints,TimeOfArrival,Name,Value)
```

Description

`trajectory = waypointTrajectory` returns a System object, `trajectory`, that generates a trajectory based on default stationary waypoints.

`trajectory = waypointTrajectory(Waypoints,TimeOfArrival)` specifies the `Waypoints` that the generated trajectory passes through and the `TimeOfArrival` at each waypoint.

`trajectory = waypointTrajectory(Waypoints,TimeOfArrival,Name,Value)` sets each creation argument or property `Name` to the specified `Value`. Unspecified properties and creation arguments have default or inferred values.

Example: `trajectory = waypointTrajectory([10,10,0;20,20,0;20,20,10],[0,0.5,10])` creates a waypoint trajectory System object, `trajectory`, that starts at waypoint `[10,10,0]`, and then passes through `[20,20,0]` after 0.5 seconds and `[20,20,10]` after 10 seconds.

Creation Arguments

Creation arguments are properties which are set during creation of the System object and cannot be modified later. If you do not explicitly set a creation argument value, the property value is inferred.

If you specify any creation argument, then you must specify both the Waypoints and TimeOfArrival creation arguments. You can specify `Waypoints` and `TimeOfArrival` as value-only arguments or name-value pairs.

Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see System Design in MATLAB Using System Objects.

`SampleRate` — Sample rate of trajectory (Hz)

`100` (default) | positive scalar

Sample rate of trajectory in Hz, specified as a positive scalar.

Tunable: Yes

Data Types: `double`

`SamplesPerFrame` — Number of samples per output frame

`1` (default) | positive scalar integer

Number of samples per output frame, specified as a positive scalar integer.

Tunable: Yes

Data Types: `double`

`Waypoints` — Positions in the navigation coordinate system (m)

N-by-3 matrix

Positions in the navigation coordinate system in meters, specified as an *N*-by-3 matrix. The columns of the matrix correspond to the first, second, and third axes, respectively. The rows of the matrix, *N*, correspond to individual waypoints.

Dependencies

To set this property, you must also set valid values for the TimeOfArrival property.

Data Types: `double`

`TimeOfArrival` — Time at each waypoint (s)

N-element column vector of nonnegative increasing numbers

Time corresponding to arrival at each waypoint in seconds, specified as an *N*-element column vector. The first element of `TimeOfArrival` must be `0`. The number of samples, *N*, must be the same as the number of samples (rows) defined by `Waypoints`.

Dependencies

To set this property, you must also set valid values for the Waypoints property.

Data Types: `double`

`Velocities` — Velocity in navigation coordinate system at each waypoint (m/s)

N-by-3 matrix

Velocity in the navigation coordinate system at each way point in meters per second, specified as an *N*-by-3 matrix. The columns of the matrix correspond to the first, second, and third axes, respectively.

The number of samples, *N*, must be the same as the number of samples (rows) defined by `Waypoints`.

If the velocity is specified as a non-zero value, the object automatically calculates the course of the trajectory. If the velocity is specified as zero, the object infers the course of the trajectory from adjacent waypoints.

Dependencies

To set this property, you must also set valid values for the Waypoints and TimeOfArrival properties.

Data Types: `double`

`Course` — Horizontal direction of travel (degree)

N-element real vector

Horizontal direction of travel, specified as an *N*-element real vector in degrees. The number of samples, *N*, must be the same as the number of samples (rows) defined by `Waypoints`. If neither `Velocities` nor `Course` is specified, course is inferred from the waypoints.

Dependencies

To set this property, the `Velocities` property must not be specified in object creation.

Data Types: `double`

`GroundSpeed` — Groundspeed at each waypoint (m/s)

N-element real vector

Groundspeed at each waypoint, specified as an *N*-element real vector in m/s. If the property is not specified, it is inferred from the waypoints. The number of samples, *N*, must be the same as the number of samples (rows) defined by `Waypoints`.

Dependencies

To set this property, the `Velocities` property must not be specified at object creation.

Data Types: `double`

`ClimbRate` — Climb rate at each waypoint (m/s)

N-element real vector

Climb Rate at each waypoint, specified as an *N*-element real vector in degrees. The number of samples, *N*, must be the same as the number of samples (rows) defined by `Waypoints`. If neither `Velocities` nor `Course` is specified, climbrate is inferred from the waypoints.

Dependencies

To set this property, the `Velocities` property must not be specified at object creation.

Data Types: `double`

`Orientation` — Orientation at each waypoint

N-element quaternion column vector | 3-by-3-by-*N* array of real numbers

Orientation at each waypoint, specified as an *N*-element `quaternion` column vector or 3-by-3-by-*N* array of real numbers. The number of quaternions or rotation matrices, *N*, must be the same as the number of samples (rows) defined by `Waypoints`.

If `Orientation` is specified by quaternions, the underlying class must be `double`.

Dependencies

To set this property, you must also set valid values for the Waypoints and TimeOfArrival properties.

Data Types: `quaternion` | `double`

`AutoPitch` — Align pitch angle with direction of motion
`false` (default) | `true`

Align pitch angle with the direction of motion, specified as `true` or `false`. When specified as `true`, the pitch angle automatically aligns with the direction of motion. If specified as `false`, the pitch angle is set to zero (level orientation).

Dependencies

To set this property, the `Orientation` property must not be specified at object creation.

`AutoBank` — Align roll angle to counteract centripetal force
`false` (default) | `true`

Align roll angle to counteract the centripetal force, specified as `true` or `false`. When specified as `true`, the roll angle automatically counteract the centripetal force. If specified as `false`, the roll angle is set to zero (flat orientation).

Dependencies

To set this property, the `Orientation` property must not be specified at object creation.

`ReferenceFrame` — Reference frame of trajectory
`'NED'` (default) | `'ENU'`

Reference frame of the trajectory, specified as `'NED'` (North-East-Down) or `'ENU'` (East-North-Up).

Usage

Syntax

```
[position,orientation,velocity,acceleration,angularVelocity] = trajectory()
```

Description

`[position,orientation,velocity,acceleration,angularVelocity] = trajectory()` outputs a frame of trajectory data based on specified creation arguments and properties.

Output Arguments

`position` — Position in local navigation coordinate system (m)
M-by-3 matrix

Position in the local navigation coordinate system in meters, returned as an *M*-by-3 matrix.

M is specified by the SamplesPerFrame property.

Data Types: `double`

`orientation` — Orientation in local navigation coordinate system
M-element quaternion column vector | 3-by-3-by-*M* real array

Orientation in the local navigation coordinate system, returned as an *M*-by-1 `quaternion` column vector or a 3-by-3-by-*M* real array.

Each quaternion or 3-by-3 rotation matrix is a frame rotation from the local navigation coordinate system to the current body coordinate system.

M is specified by the SamplesPerFrame property.

Data Types: `double`

`velocity` — Velocity in local navigation coordinate system (m/s)
M-by-3 matrix

Velocity in the local navigation coordinate system in meters per second, returned as an *M*-by-3 matrix.

M is specified by the SamplesPerFrame property.

Data Types: `double`

`acceleration` — Acceleration in local navigation coordinate system (m/s^2)
M-by-3 matrix

Acceleration in the local navigation coordinate system in meters per second squared, returned as an *M*-by-3 matrix.

M is specified by the SamplesPerFrame property.

Data Types: `double`

`angularVelocity` — Angular velocity in local navigation coordinate system (rad/s)
M-by-3 matrix

Angular velocity in the local navigation coordinate system in radians per second, returned as an *M*-by-3 matrix.

M is specified by the SamplesPerFrame property.

Data Types: `double`

Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

Specific to `waypointTrajectory`

waypointInfo	Get waypoint information table
lookupPose	Obtain pose information for certain time
perturbations	Perturbation defined on object
perturb	Apply perturbations to object

Common to All System Objects

clone	Create duplicate System object
step	Run System object algorithm
release	Release resources and allow changes to System object property values and input characteristics
reset	Reset internal states of System object
isDone	End-of-data status

Examples

Create Default `waypointTrajectory`

```
trajectory = waypointTrajectory
```

```
trajectory =
  waypointTrajectory with properties:

         SampleRate: 100
    SamplesPerFrame: 1
          Waypoints: [2x3 double]
      TimeOfArrival: [2x1 double]
         Velocities: [2x3 double]
             Course: [2x1 double]
        GroundSpeed: [2x1 double]
          ClimbRate: [2x1 double]
        Orientation: [2x1 quaternion]
          AutoPitch: 0
           AutoBank: 0
     ReferenceFrame: 'NED'
```

Inspect the default waypoints and times of arrival by calling `waypointInfo`. By default, the waypoints indicate a stationary position for one second.

```
waypointInfo(trajectory)
```

```
ans=2×2 table
    TimeOfArrival     Waypoints
    _____________    ___________

          0          0    0    0
          1          0    0    0
```

Create Square Trajectory

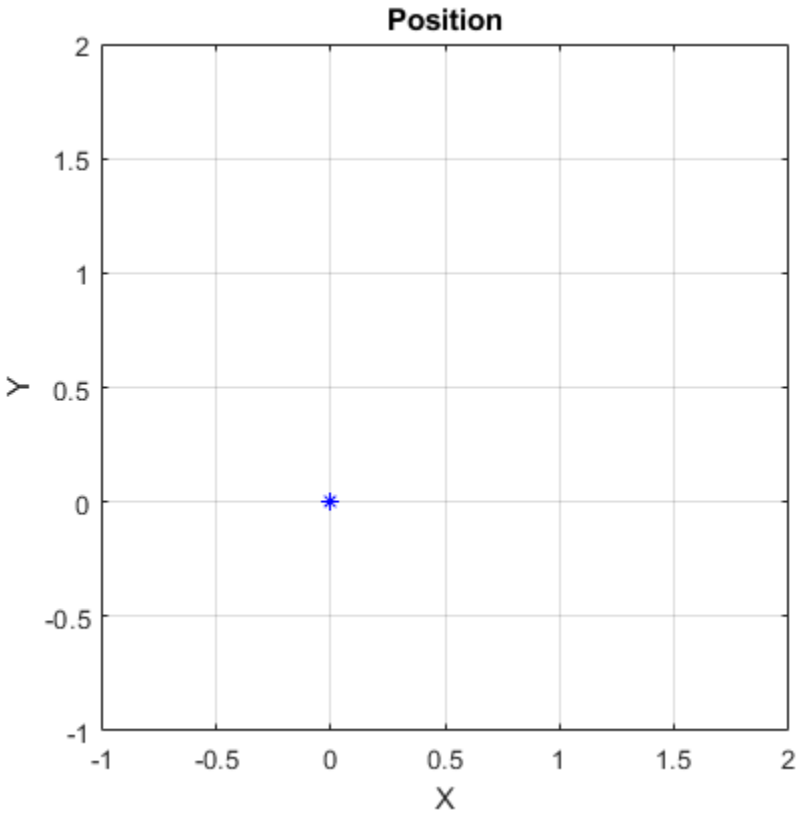
Create a square trajectory and examine the relationship between waypoint constraints, sample rate, and the generated trajectory.

Create a square trajectory by defining the vertices of the square. Define the orientation at each waypoint as pointing in the direction of motion. Specify a 1 Hz sample rate and use the default `SamplesPerFrame` of 1.

```
waypoints = [0,0,0; ... % Initial position
             0,1,0; ...
             1,1,0; ...
             1,0,0; ...
             0,0,0];    % Final position

toa = 0:4; % time of arrival

orientation = quaternion([0,0,0; ...
                          45,0,0; ...
                          135,0,0; ...
                          225,0,0; ...
                          0,0,0], ...
                          'eulerd','ZYX','frame');

trajectory = waypointTrajectory(waypoints, ...
    'TimeOfArrival',toa, ...
    'Orientation',orientation, ...
    'SampleRate',1);
```

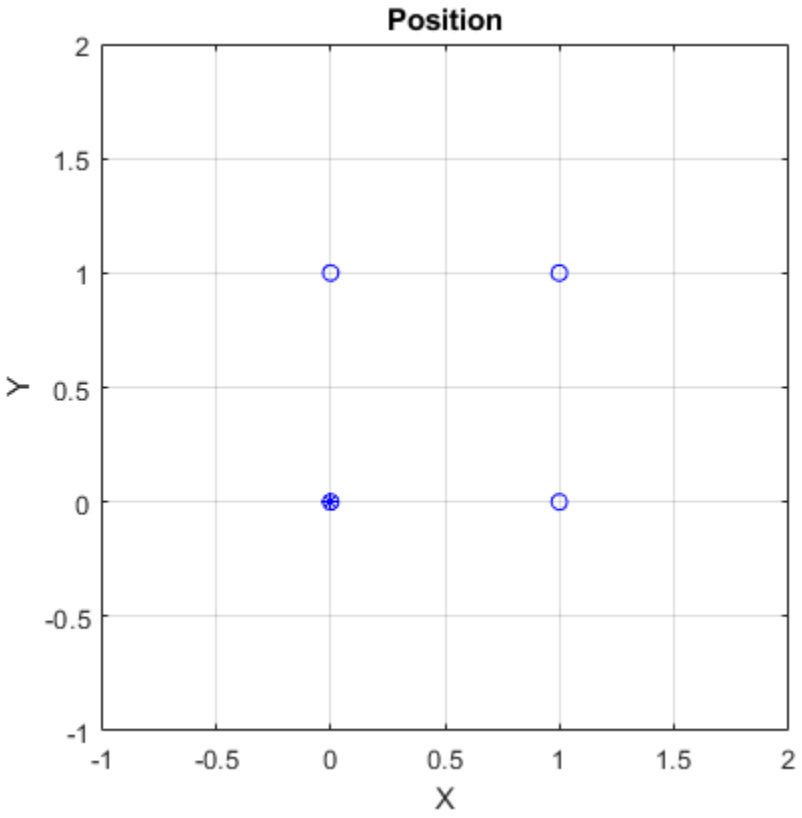
Create a figure and plot the initial position of the platform.

```
figure(1)
plot(waypoints(1,1),waypoints(1,2),'b*')
title('Position')
axis([-1,2,-1,2])
axis square
xlabel('X')
ylabel('Y')
grid on
hold on
```



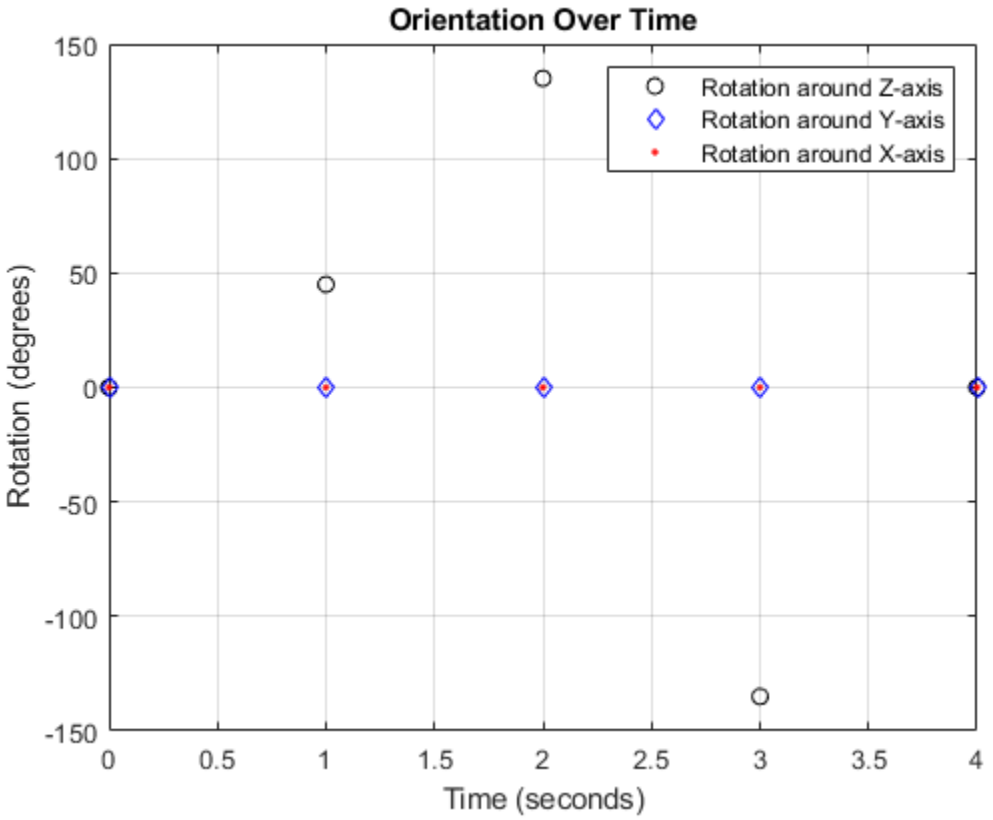
In a loop, step through the trajectory to output the current position and current orientation. Plot the current position and log the orientation. Use `pause` to mimic real-time processing.

```
orientationLog = zeros(toa(end)*trajectory.SampleRate,1,'quaternion');
count = 1;
while ~isDone(trajectory)
   [currentPosition,orientationLog(count)] = trajectory();

   plot(currentPosition(1),currentPosition(2),'bo')

   pause(trajectory.SamplesPerFrame/trajectory.SampleRate)
   count = count + 1;
end
hold off
```



Convert the orientation quaternions to Euler angles for easy interpretation, and then plot orientation over time.

```
figure(2)
eulerAngles = eulerd([orientation(1);orientationLog],'ZYX','frame');
plot(toa,eulerAngles(:,1),'ko', ...
     toa,eulerAngles(:,2),'bd', ...
     toa,eulerAngles(:,3),'r.');
title('Orientation Over Time')
legend('Rotation around Z-axis','Rotation around Y-axis','Rotation around X-axis')
xlabel('Time (seconds)')
ylabel('Rotation (degrees)')
grid on
```



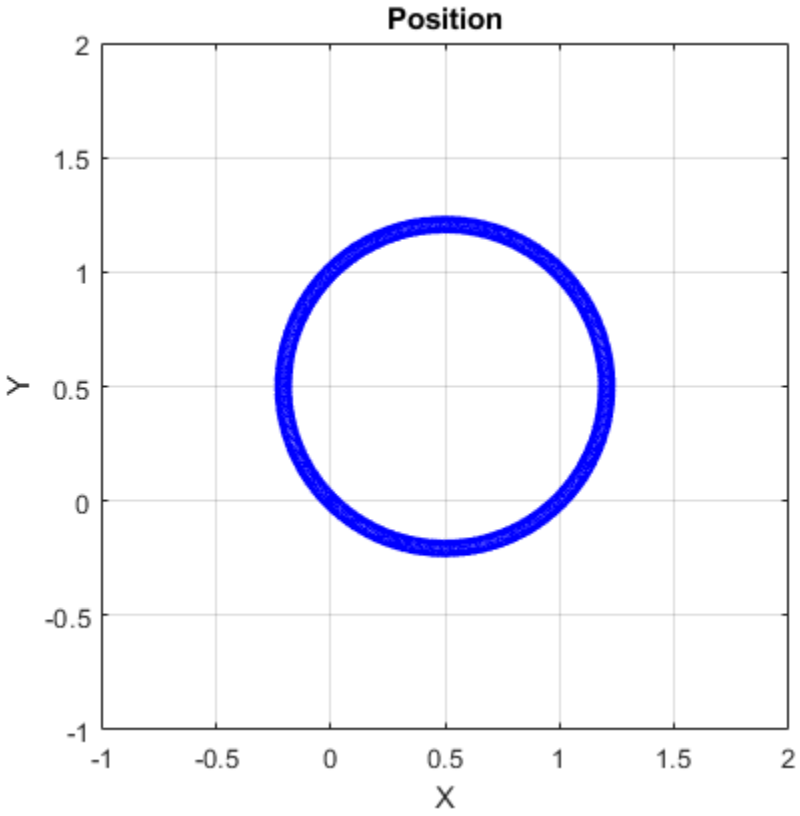
So far, the trajectory object has only output the waypoints that were specified during construction. To interpolate between waypoints, increase the sample rate to a rate faster than the time of arrivals of the waypoints. Set the `trajectory` sample rate to 100 Hz and call `reset`.

```
trajectory.SampleRate = 100;
reset(trajectory)
```

Create a figure and plot the initial position of the platform. In a loop, step through the trajectory to output the current position and current orientation. Plot the current position and log the orientation. Use `pause` to mimic real-time processing.

```
figure(1)
plot(waypoints(1,1),waypoints(1,2),'b*')
title('Position')
axis([-1,2,-1,2])
axis square
xlabel('X')
ylabel('Y')
grid on
hold on

orientationLog = zeros(toa(end)*trajectory.SampleRate,1,'quaternion');
count = 1;
while ~isDone(trajectory)
   [currentPosition,orientationLog(count)] = trajectory();

   plot(currentPosition(1),currentPosition(2),'bo')
```

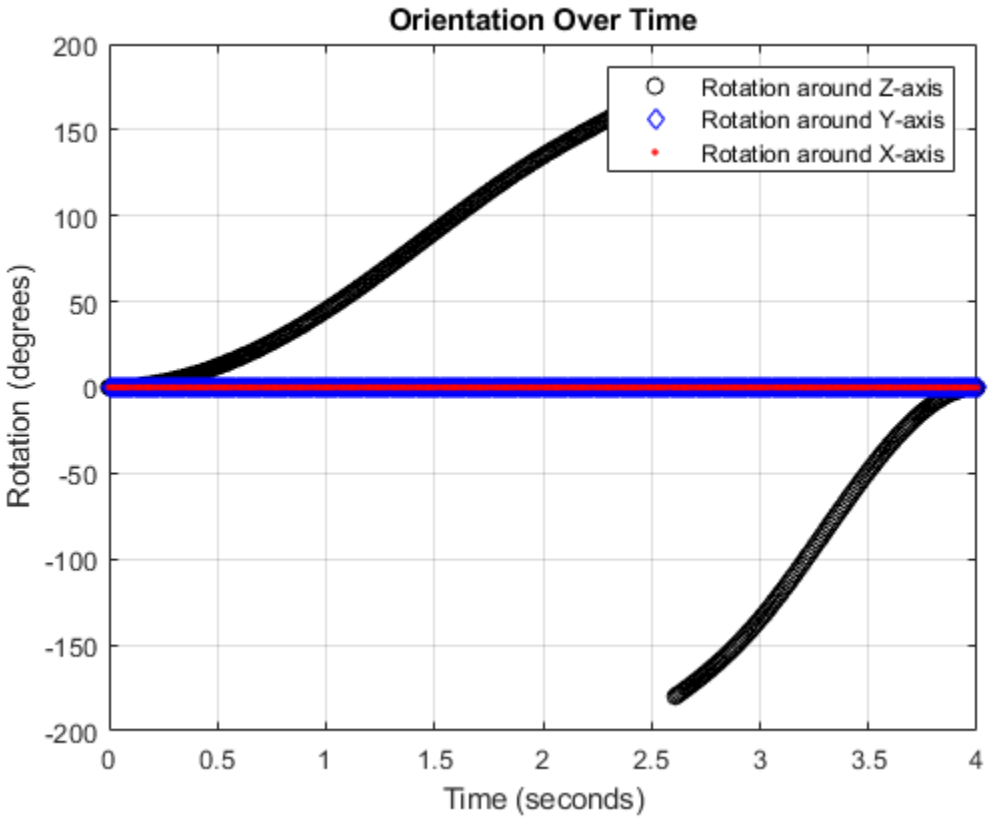
```
    pause(trajectory.SamplesPerFrame/trajectory.SampleRate)
    count = count + 1;
end
hold off
```



The trajectory output now appears circular. This is because the `waypointTrajectory` System object™ minimizes the acceleration and angular velocity when interpolating, which results in smoother, more realistic motions in most scenarios.

Convert the orientation quaternions to Euler angles for easy interpretation, and then plot orientation over time. The orientation is also interpolated.

```
figure(2)
eulerAngles = eulerd([orientation(1);orientationLog],'ZYX','frame');
t = 0:1/trajectory.SampleRate:4;
plot(t,eulerAngles(:,1),'ko', ...
     t,eulerAngles(:,2),'bd', ...
     t,eulerAngles(:,3),'r.');
title('Orientation Over Time')
legend('Rotation around Z-axis','Rotation around Y-axis','Rotation around X-axis')
xlabel('Time (seconds)')
ylabel('Rotation (degrees)')
grid on
```

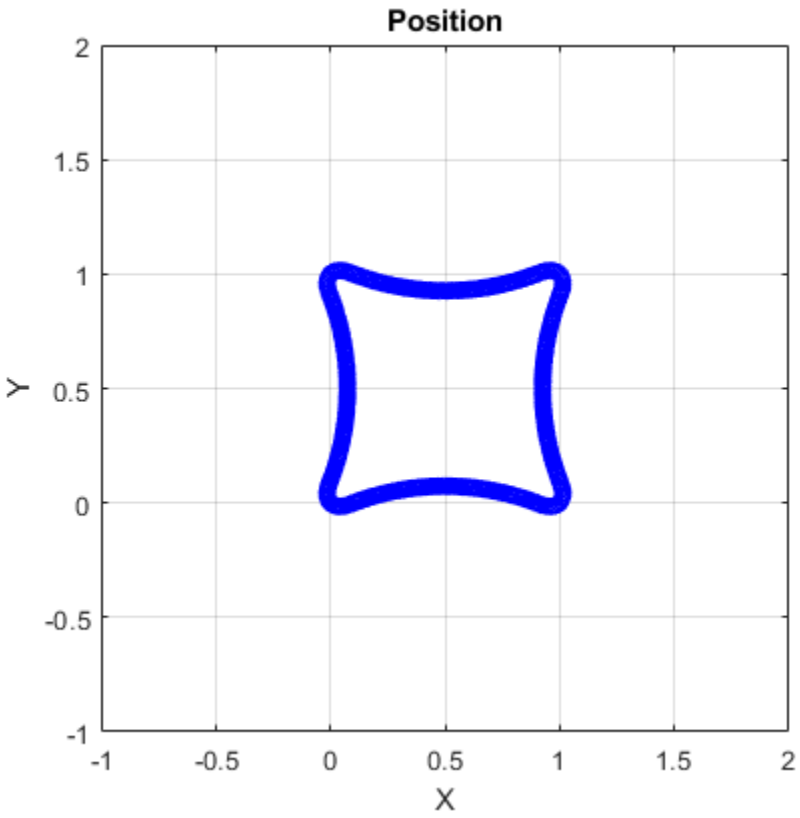


The `waypointTrajectory` algorithm interpolates the waypoints to create a smooth trajectory. To return to the square trajectory, provide more waypoints, especially around sharp changes. To track corresponding times, waypoints, and orientation, specify all the trajectory info in a single matrix.

```
               % Time, Waypoint, Orientation
trajectoryInfo = [0,   0,0,0,    0,0,0; ... % Initial position
                  0.1, 0,0.1,0,  0,0,0; ...

                  0.9, 0,0.9,0,  0,0,0; ...
                  1,   0,1,0,    45,0,0; ...
                  1.1, 0.1,1,0,  90,0,0; ...

                  1.9, 0.9,1,0,  90,0,0; ...
                  2,   1,1,0,    135,0,0; ...
                  2.1, 1,0.9,0,  180,0,0; ...

                  2.9, 1,0.1,0,  180,0,0; ...
                  3,   1,0,0,    225,0,0; ...
                  3.1, 0.9,0,0,  270,0,0; ...

                  3.9, 0.1,0,0,  270,0,0; ...
                  4,   0,0,0,    270,0,0];    % Final position

trajectory = waypointTrajectory(trajectoryInfo(:,2:4), ...
    'TimeOfArrival',trajectoryInfo(:,1), ...
    'Orientation',quaternion(trajectoryInfo(:,5:end),'eulerd','ZYX','frame'), ...
    'SampleRate',100);
```

Create a figure and plot the initial position of the platform. In a loop, step through the trajectory to output the current position and current orientation. Plot the current position and log the orientation. Use `pause` to mimic real-time processing.

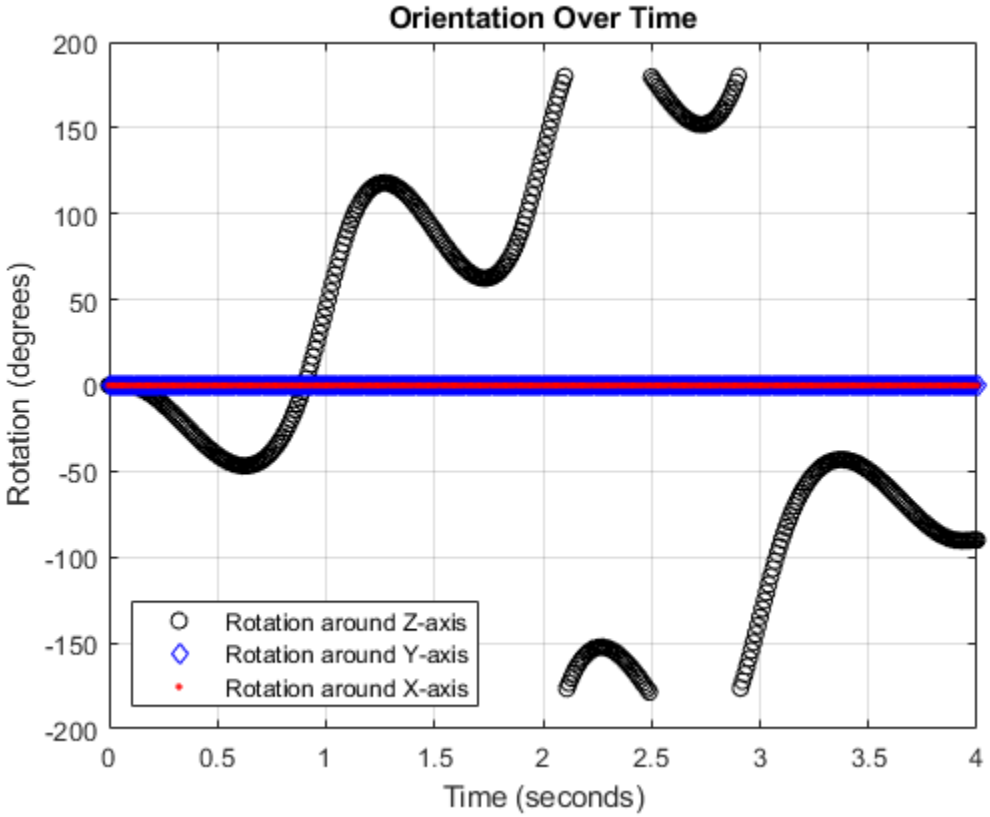
```
figure(1)
plot(waypoints(1,1),waypoints(1,2),'b*')
title('Position')
axis([-1,2,-1,2])
axis square
xlabel('X')
ylabel('Y')
grid on
hold on

orientationLog = zeros(toa(end)*trajectory.SampleRate,1,'quaternion');
count = 1;
while ~isDone(trajectory)
   [currentPosition,orientationLog(count)] = trajectory();

   plot(currentPosition(1),currentPosition(2),'bo')

   pause(trajectory.SamplesPerFrame/trajectory.SampleRate)
   count = count+1;
end
hold off
```



The trajectory output now appears more square-like, especially around the vertices with waypoints.

Convert the orientation quaternions to Euler angles for easy interpretation, and then plot orientation over time.

```
figure(2)
eulerAngles = eulerd([orientation(1);orientationLog],'ZYX','frame');
t = 0:1/trajectory.SampleRate:4;
eulerAngles = plot(t,eulerAngles(:,1),'ko', ...
                   t,eulerAngles(:,2),'bd', ...
                   t,eulerAngles(:,3),'r.');
title('Orientation Over Time')
legend('Rotation around Z-axis', ...
       'Rotation around Y-axis', ...
       'Rotation around X-axis', ...
       'Location', 'SouthWest')
xlabel('Time (seconds)')
ylabel('Rotation (degrees)')
grid on
```



Create Arc Trajectory

This example shows how to create an arc trajectory using the `waypointTrajectory` System object™. `waypointTrajectory` creates a path through specified waypoints that minimizes acceleration and angular velocity. After creating an arc trajectory, you restrict the trajectory to be within preset bounds.

Create an Arc Trajectory

Define a constraints matrix consisting of waypoints, times of arrival, and orientation for an arc trajectory. The generated trajectory passes through the waypoints at the specified times with the specified orientation. The `waypointTrajectory` System object requires orientation to be specified using quaternions or rotation matrices. Convert the Euler angles saved in the constrains matrix to quaternions when specifying the `Orientation` property.

```
          % Arrival, Waypoints, Orientation
constraints = [0,    20,20,0,    90,0,0;
               3,    50,20,0,    90,0,0;
               4,    58,15.5,0,  162,0,0;
               5.5,  59.5,0,0    180,0,0];

trajectory = waypointTrajectory(constraints(:,2:4), ...
    'TimeOfArrival',constraints(:,1), ...
    'Orientation',quaternion(constraints(:,5:7),'eulerd','ZYX','frame'));
```

Call `waypointInfo` on `trajectory` to return a table of your specified constraints. The creation properties `Waypoints`, `TimeOfArrival`, and `Orientation` are variables of the table. The table is convenient for indexing while plotting.

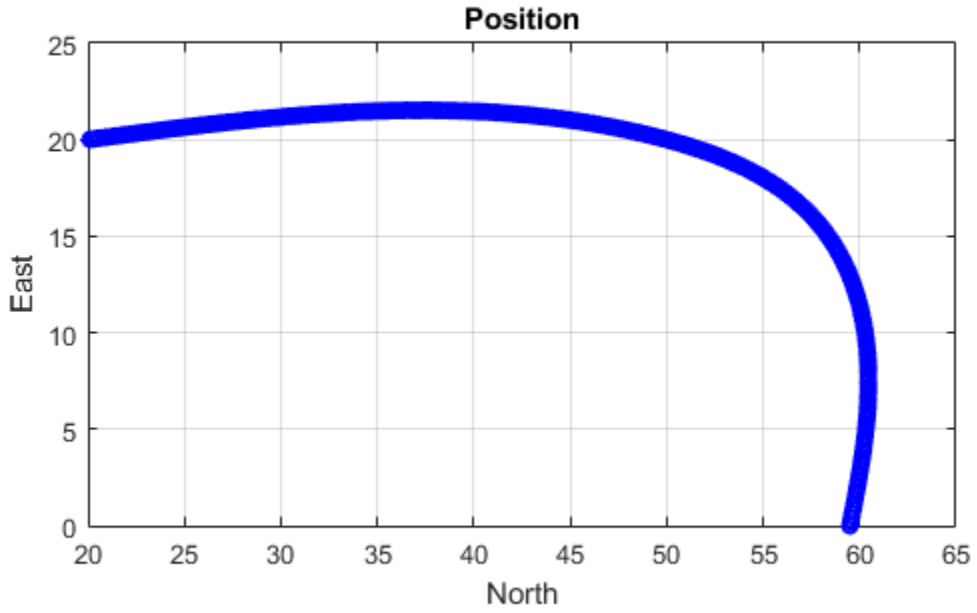
```
tInfo = waypointInfo(trajectory)
```

```
tInfo =

  4x3 table

    TimeOfArrival         Waypoints            Orientation
    _____________    ____________________    _______________

           0           20      20       0    {1x1 quaternion}
           3           50      20       0    {1x1 quaternion}
           4           58    15.5       0    {1x1 quaternion}
         5.5         59.5       0       0    {1x1 quaternion}
```

The trajectory object outputs the current position, velocity, acceleration, and angular velocity at each call. Call `trajectory` in a loop and plot the position over time. Cache the other outputs.

```
figure(1)
plot(tInfo.Waypoints(1,1),tInfo.Waypoints(1,2),'b*')
title('Position')
axis([20,65,0,25])
xlabel('North')
ylabel('East')
grid on
daspect([1 1 1])
hold on

orient = zeros(tInfo.TimeOfArrival(end)*trajectory.SampleRate,1,'quaternion');
vel = zeros(tInfo.TimeOfArrival(end)*trajectory.SampleRate,3);
acc = vel;
angVel = vel;

count = 1;
while ~isDone(trajectory)
```

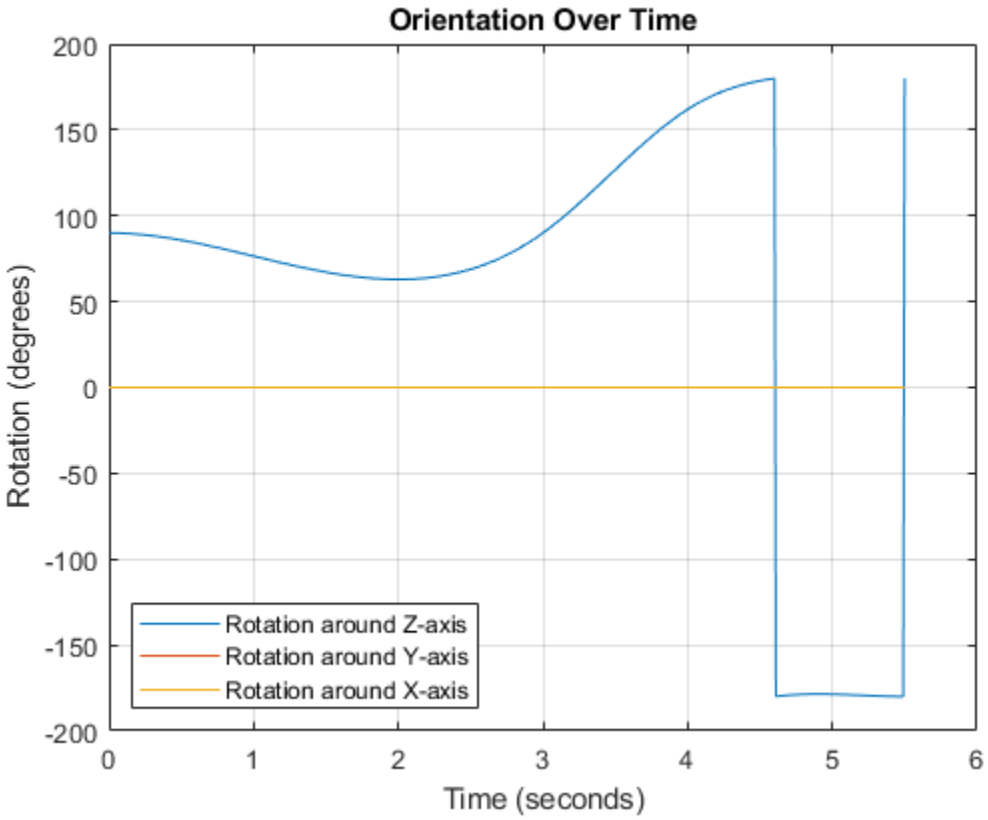
```
    [pos,orient(count),vel(count,:),acc(count,:),angVel(count,:)] = trajectory();

    plot(pos(1),pos(2),'bo')

    pause(trajectory.SamplesPerFrame/trajectory.SampleRate)
    count = count + 1;
end
```

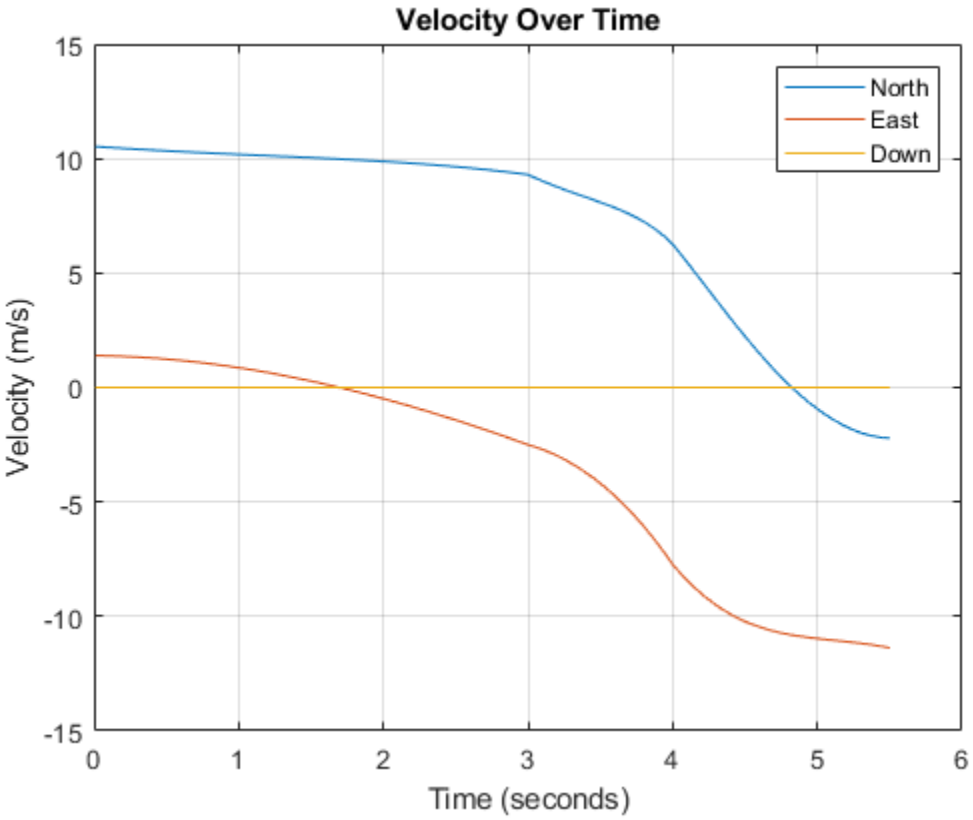


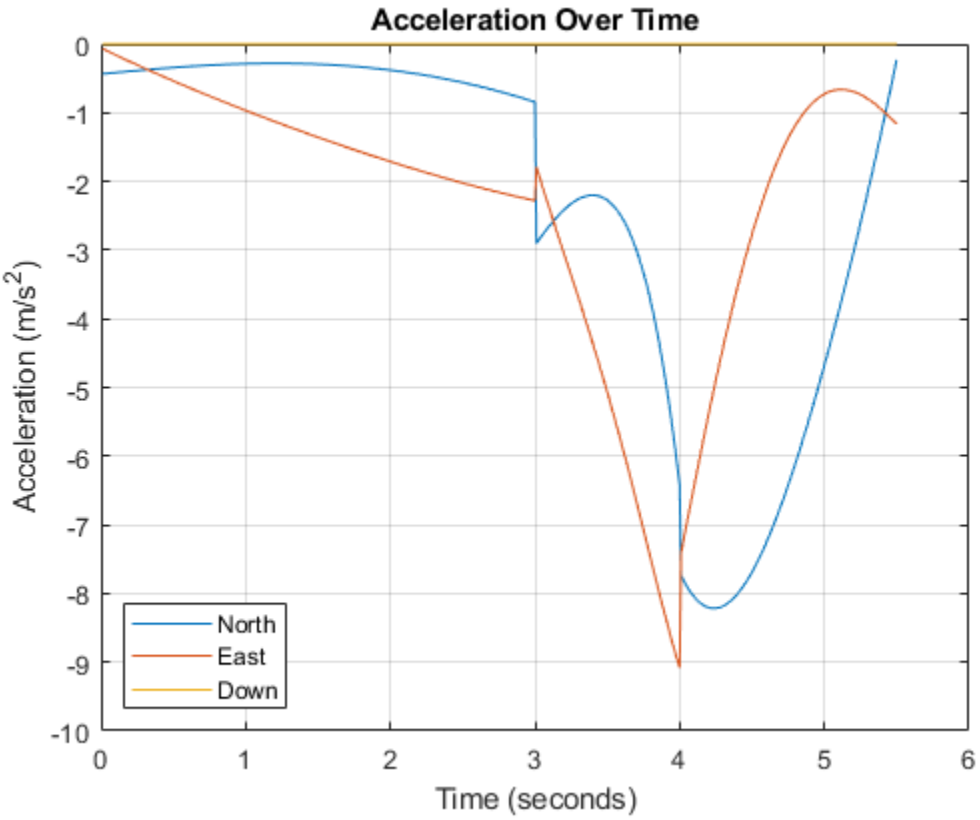
Inspect the orientation, velocity, acceleration, and angular velocity over time. The `waypointTrajectory` System object™ creates a path through the specified constraints that minimized acceleration and angular velocity.

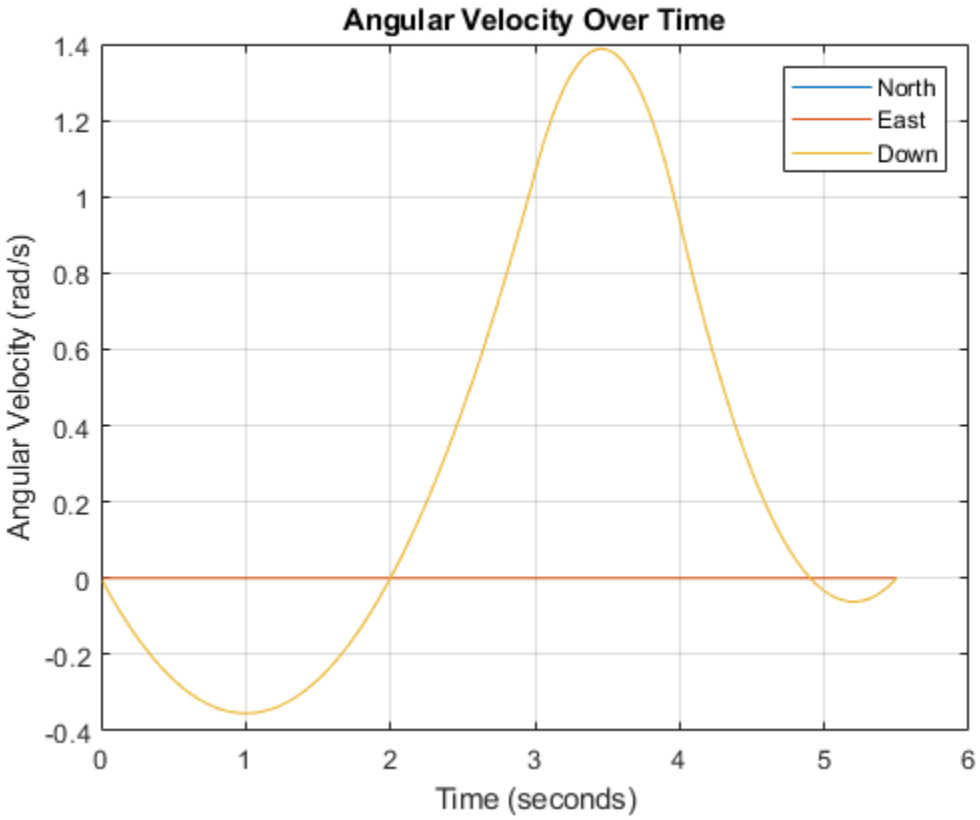
```
figure(2)
timeVector = 0:(1/trajectory.SampleRate):tInfo.TimeOfArrival(end);
eulerAngles = eulerd([tInfo.Orientation{1};orient],'ZYX','frame');
plot(timeVector,eulerAngles(:,1), ...
     timeVector,eulerAngles(:,2), ...
     timeVector,eulerAngles(:,3));
title('Orientation Over Time')
legend('Rotation around Z-axis', ...
       'Rotation around Y-axis', ...
       'Rotation around X-axis', ...
       'Location','southwest')
xlabel('Time (seconds)')
ylabel('Rotation (degrees)')
grid on
```

```
figure(3)
plot(timeVector(2:end),vel(:,1), ...
     timeVector(2:end),vel(:,2), ...
     timeVector(2:end),vel(:,3));
title('Velocity Over Time')
legend('North','East','Down')
xlabel('Time (seconds)')
ylabel('Velocity (m/s)')
grid on

figure(4)
plot(timeVector(2:end),acc(:,1), ...
     timeVector(2:end),acc(:,2), ...
     timeVector(2:end),acc(:,3));
title('Acceleration Over Time')
legend('North','East','Down','Location','southwest')
xlabel('Time (seconds)')
ylabel('Acceleration (m/s^2)')
grid on

figure(5)
plot(timeVector(2:end),angVel(:,1), ...
     timeVector(2:end),angVel(:,2), ...
     timeVector(2:end),angVel(:,3));
title('Angular Velocity Over Time')
legend('North','East','Down')
xlabel('Time (seconds)')
ylabel('Angular Velocity (rad/s)')
grid on
```


Orientation Over Time
Rotation (degrees)
Time (seconds)
Rotation around Z-axis
Rotation around Y-axis
Rotation around X-axis
200
150
100
50
0
-50
-100
-150
-200
0
1
2
3
4
5
6


Velocity Over Time
North
East
Down
Velocity (m/s)
Time (seconds)
15
10
5
0
-5
-10
-15
0
1
2
3
4
5
6

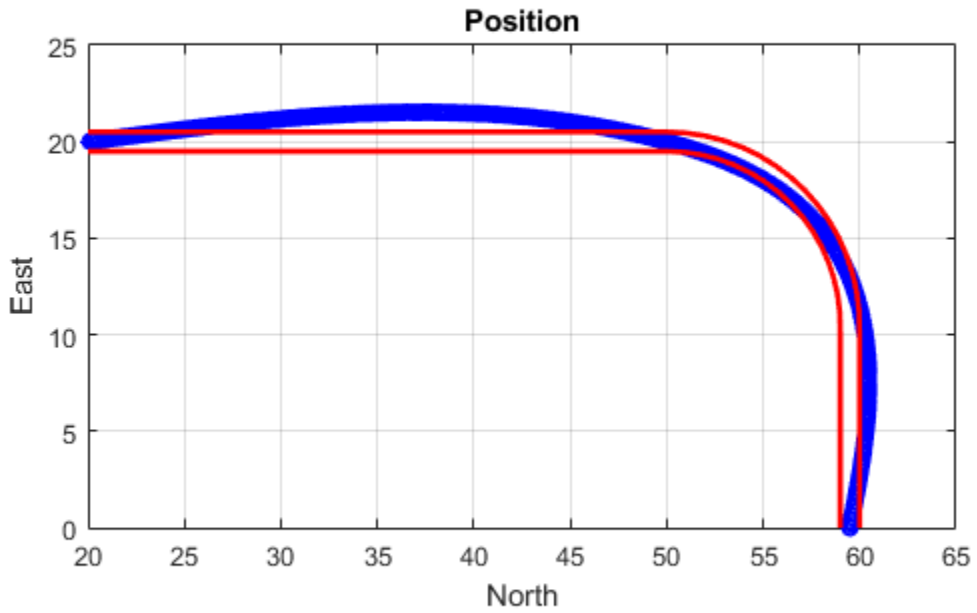

Acceleration Over Time
Acceleration (m/s^2)
Time (seconds)
North
East
Down
0
-1
-2
-3
-4
-5
-6
-7
-8
-9
-10
0
1
2
3
4
5
6



Restrict Arc Trajectory Within Preset Bounds

You can specify additional waypoints to create trajectories within given bounds. Create upper and lower bounds for the arc trajectory.

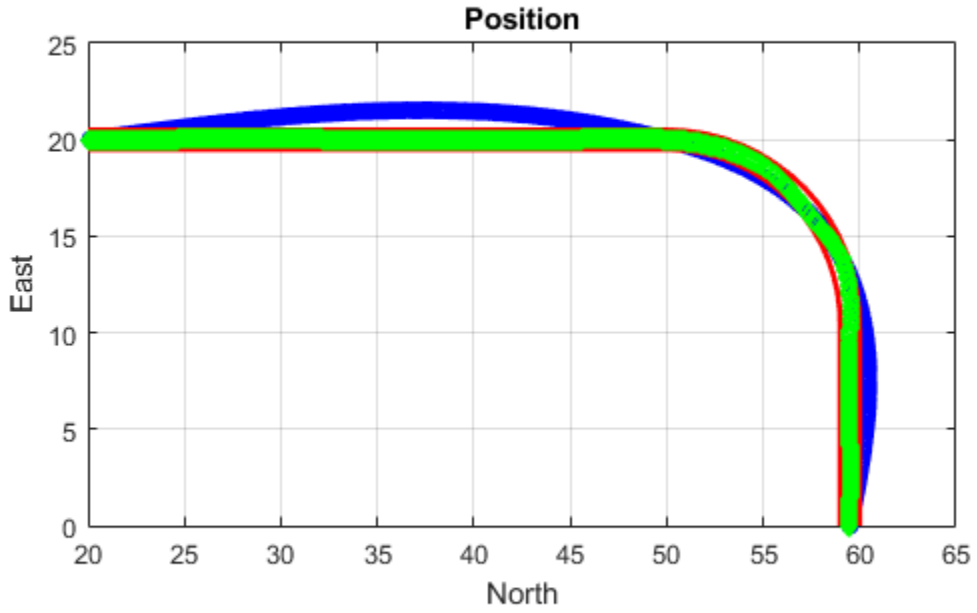
```
figure(1)
xUpperBound = [(20:50)';50+10*sin(0:0.1:pi/2)';60*ones(11,1)];
yUpperBound = [20.5.*ones(31,1);10.5+10*cos(0:0.1:pi/2)';(10:-1:0)'];

xLowerBound = [(20:49)';50+9*sin(0:0.1:pi/2)';59*ones(11,1)];
yLowerBound = [19.5.*ones(30,1);10.5+9*cos(0:0.1:pi/2)';(10:-1:0)'];

plot(xUpperBound,yUpperBound,'r','LineWidth',2);
plot(xLowerBound,yLowerBound,'r','LineWidth',2)
```



To create a trajectory within the bounds, add additional waypoints. Create a new `waypointTrajectory` System object™, and then call it in a loop to plot the generated trajectory. Cache the orientation, velocity, acceleration, and angular velocity output from the `trajectory` object.

```
            % Time,  Waypoint,     Orientation
constraints = [0,    20,20,0,      90,0,0;
               1.5,  35,20,0,      90,0,0;
               2.5   45,20,0,      90,0,0;
               3,    50,20,0,      90,0,0;
               3.3,  53,19.5,0,    108,0,0;
               3.6,  55.5,18.25,0, 126,0,0;
               3.9,  57.5,16,0,    144,0,0;
               4.2,  59,14,0,      162,0,0;
               4.5,  59.5,10,0     180,0,0;
               5,    59.5,5,0      180,0,0;
               5.5,  59.5,0,0      180,0,0];

trajectory = waypointTrajectory(constraints(:,2:4), ...
    'TimeOfArrival',constraints(:,1), ...
    'Orientation',quaternion(constraints(:,5:7),'eulerd','ZYX','frame'));
tInfo = waypointInfo(trajectory);

figure(1)
plot(tInfo.Waypoints(1,1),tInfo.Waypoints(1,2),'b*')

count = 1;
```

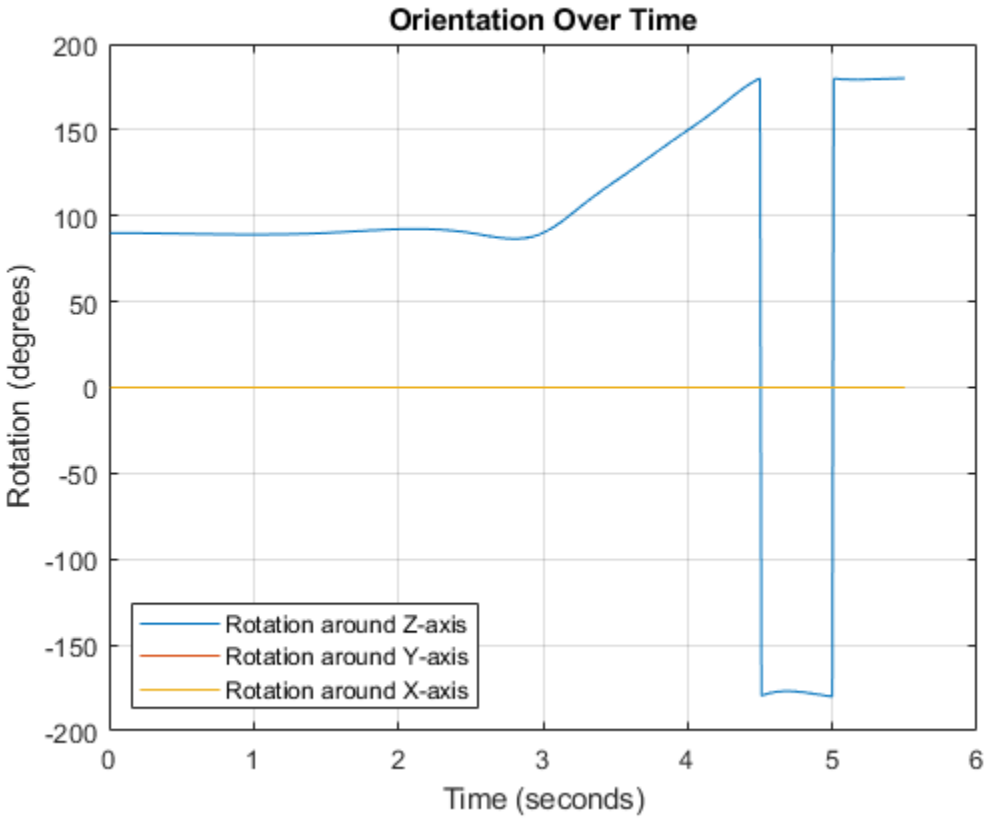
```
while ~isDone(trajectory)
   [pos,orient(count),vel(count,:),acc(count,:),angVel(count,:)] = trajectory();

   plot(pos(1),pos(2),'gd')

   pause(trajectory.SamplesPerFrame/trajectory.SampleRate)
   count = count + 1;
end
```

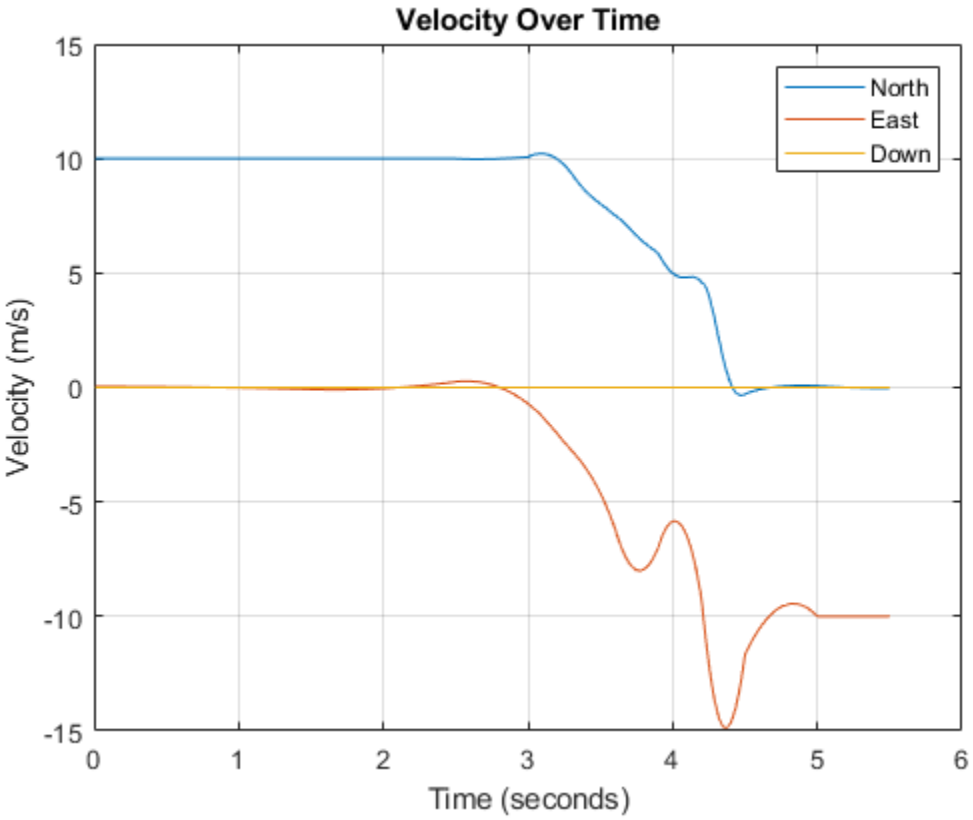


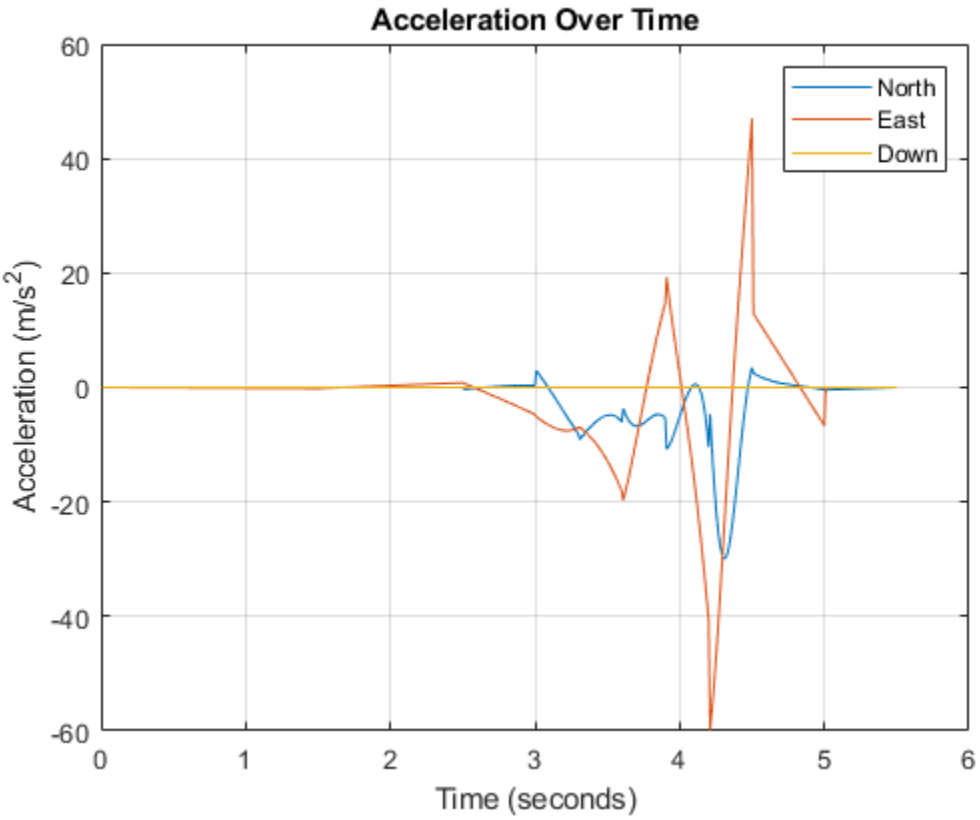
The generated trajectory now fits within the specified boundaries. Visualize the orientation, velocity, acceleration, and angular velocity of the generated trajectory.

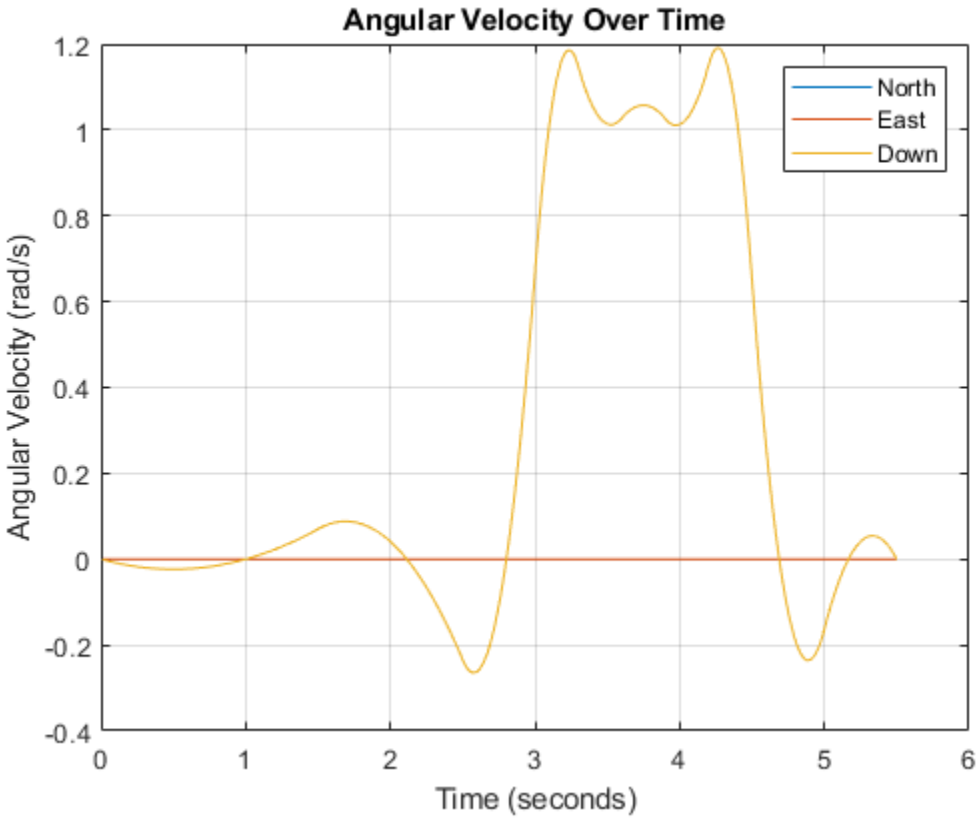
```
figure(2)
timeVector = 0:(1/trajectory.SampleRate):tInfo.TimeOfArrival(end);
eulerAngles = eulerd(orient,'ZYX','frame');
plot(timeVector(2:end),eulerAngles(:,1), ...
     timeVector(2:end),eulerAngles(:,2), ...
     timeVector(2:end),eulerAngles(:,3));
title('Orientation Over Time')
legend('Rotation around Z-axis', ...
       'Rotation around Y-axis', ...
       'Rotation around X-axis', ...
       'Location','southwest')
xlabel('Time (seconds)')
ylabel('Rotation (degrees)')
grid on

figure(3)
```

```
plot(timeVector(2:end),vel(:,1), ...
     timeVector(2:end),vel(:,2), ...
     timeVector(2:end),vel(:,3));
title('Velocity Over Time')
legend('North','East','Down')
xlabel('Time (seconds)')
ylabel('Velocity (m/s)')
grid on

figure(4)
plot(timeVector(2:end),acc(:,1), ...
     timeVector(2:end),acc(:,2), ...
     timeVector(2:end),acc(:,3));
title('Acceleration Over Time')
legend('North','East','Down')
xlabel('Time (seconds)')
ylabel('Acceleration (m/s^2)')
grid on

figure(5)
plot(timeVector(2:end),angVel(:,1), ...
     timeVector(2:end),angVel(:,2), ...
     timeVector(2:end),angVel(:,3));
title('Angular Velocity Over Time')
legend('North','East','Down')
xlabel('Time (seconds)')
ylabel('Angular Velocity (rad/s)')
grid on
```


Orientation Over Time
Rotation (degrees)
Time (seconds)
200
150
100
50
0
-50
-100
-150
-200
0
1
2
3
4
5
6
Rotation around Z-axis
Rotation around Y-axis
Rotation around X-axis


Velocity Over Time
North
East
Down
Velocity (m/s)
Time (seconds)
15
10
5
0
-5
-10
-15
0
1
2
3
4
5
6


Acceleration Over Time
North
East
Down
Acceleration (m/s^2)
Time (seconds)
60
40
20
0
-20
-40
-60
0
1
2
3
4
5
6



Note that while the generated trajectory now fits within the spatial boundaries, the acceleration and angular velocity of the trajectory are somewhat erratic. This is due to over-specifying waypoints.

Algorithms

The `waypointTrajectory` System object defines a trajectory that smoothly passes through waypoints. The trajectory connects the waypoints through an interpolation that assumes the gravity direction expressed in the trajectory reference frame is constant. Generally, you can use `waypointTrajectory` to model platform or vehicle trajectories within a hundreds of kilometers distance span.

The planar path of the trajectory (the *x*-*y* plane projection) consists of piecewise, clothoid curves. The curvature of the curve between two consecutive waypoints varies linearly with the curve length between them. The tangent direction of the path at each waypoint is chosen to minimize discontinuities in the curvature, unless the course is specified explicitly via the `Course` property or implicitly via the `Velocities` property. Once the path is established, the object uses cubic Hermite interpolation to compute the location of the vehicle throughout the path as a function of time and the planar distance travelled.

The normal component (*z*-component) of the trajectory is subsequently chosen to satisfy a shape-preserving piecewise spline (PCHIP) unless the climb rate is specified explicitly via the `ClimbRate` property or the third column of the `Velocities` property. Choose the sign of the climb rate based on the selected `ReferenceFrame`:

- When an 'ENU' reference frame is selected, specifying a positive climb rate results in an increasing value of z.
- When an 'NED' reference frame is selected, specifying a positive climb rate results in a decreasing value of z.

You can define the orientation of the vehicle through the path in two primary ways:

- If the `Orientation` property is specified, then the object uses a piecewise-cubic, quaternion spline to compute the orientation along the path as a function of time.
- If the `Orientation` property is not specified, then the yaw of the vehicle is always aligned with the path. The roll and pitch are then governed by the `AutoBank` and `AutoPitch` property values, respectively.

AutoBank	AutoPitch	Description
`false`	`false`	The vehicle is always level (zero pitch and roll). This is typically used for large marine vessels.
`false`	`true`	The vehicle pitch is aligned with the path, and its roll is always zero. This is typically used for ground vehicles.
`true`	`false`	The vehicle pitch and roll are chosen so that its local z-axis is aligned with the net acceleration (including gravity). This is typically used for rotary-wing craft.
`true`	`true`	The vehicle roll is chosen so that its local transverse plane aligns with the net acceleration (including gravity). The vehicle pitch is aligned with the path. This is typically used for two-wheeled vehicles and fixed-wing aircraft.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

The object function, `waypointInfo`, does not support code generation.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

`kinematicTrajectory`

Introduced in R2019b

lookupPose

Obtain pose information for certain time

Syntax

```
[position,orientation,velocity,acceleration,angularVelocity] = lookupPose(
traj,sampleTimes)
```

Description

`[position,orientation,velocity,acceleration,angularVelocity] = lookupPose(traj,sampleTimes)` returns the pose information of the waypoint trajectory at the specified sample times. If any sample time is beyond the duration of the trajectory, the corresponding pose information is returned as `NaN`.

Input Arguments

`traj` — Waypoint trajectory

`waypointTrajectory` object

Waypoint trajectory, specified as a `waypointTrajectory` object.

`sampleTimes` — Sample times

M-element vector of nonnegative scalar

Sample times in seconds, specified as an *M*-element vector of nonnegative scalars.

Output Arguments

`position` — Position in local navigation coordinate system (m)

M-by-3 matrix

Position in the local navigation coordinate system in meters, returned as an *M*-by-3 matrix.

M is specified by the `sampleTimes` input.

Data Types: `double`

`orientation` — Orientation in local navigation coordinate system

M-element quaternion column vector | 3-by-3-by-*M* real array

Orientation in the local navigation coordinate system, returned as an *M*-by-1 `quaternion` column vector or a 3-by-3-by-*M* real array.

Each quaternion or 3-by-3 rotation matrix is a frame rotation from the local navigation coordinate system to the current body coordinate system.

M is specified by the `sampleTimes` input.

Data Types: `double`

`velocity` — Velocity in local navigation coordinate system (m/s)
M-by-3 matrix

Velocity in the local navigation coordinate system in meters per second, returned as an *M*-by-3 matrix.

M is specified by the `sampleTimes` input.

Data Types: `double`

`acceleration` — Acceleration in local navigation coordinate system (m/s²)
M-by-3 matrix

Acceleration in the local navigation coordinate system in meters per second squared, returned as an *M*-by-3 matrix.

M is specified by the `sampleTimes` input.

Data Types: `double`

`angularVelocity` — Angular velocity in local navigation coordinate system (rad/s)
M-by-3 matrix

Angular velocity in the local navigation coordinate system in radians per second, returned as an *M*-by-3 matrix.

M is specified by the `sampleTimes` input.

Data Types: `double`

See Also

`waypointTrajectory`

Introduced in R2019b

waypointInfo

Get waypoint information table

Syntax

```
trajectoryInfo = waypointInfo(trajectory)
```

Description

`trajectoryInfo = waypointInfo(trajectory)` returns a table of waypoints, times of arrival, velocities, and orientation for the `trajectory` System object

Input Arguments

`trajectory` — Object of `waypointTrajectory`

object

Object of the `waypointTrajectory` System object.

Output Arguments

`trajectoryInfo` — Trajectory information

table

Trajectory information, returned as a table with variables corresponding to set creation properties: Waypoints, TimeOfArrival, Velocities, and Orientation.

The trajectory information table always has variables `Waypoints` and `TimeOfArrival`. If the `Velocities` property is set during construction, the trajectory information table additionally returns velocities. If the `Orientation` property is set during construction, the trajectory information table additionally returns orientation.

See Also

`waypointTrajectory`

Introduced in R2019b

perturb

Apply perturbations to object

Syntax

```
offsets = perturb(obj)
```

Description

`offsets = perturb(obj)` applies the perturbations defined on the object, `obj` and returns the offset values. You can define perturbations on the object by using the `perturbations` function.

Examples

Perturb Waypoint Trajectory

Define a waypoint trajectory. By default, this trajectory contains two waypoints.

```
traj = waypointTrajectory
```

```
traj =
  waypointTrajectory with properties:

         SampleRate: 100
    SamplesPerFrame: 1
          Waypoints: [2x3 double]
      TimeOfArrival: [2x1 double]
         Velocities: [2x3 double]
             Course: [2x1 double]
        GroundSpeed: [2x1 double]
          ClimbRate: [2x1 double]
        Orientation: [2x1 quaternion]
          AutoPitch: 0
           AutoBank: 0
     ReferenceFrame: 'NED'
```

Define perturbations on the `Waypoints` property and the `TimeOfArrival` property.

```
rng(2020);
perturbs1 = perturbations(traj,'Waypoints','Normal',1,1)
```

```
perturbs1=2×3 table
       Property          Type            Value
    _______________    ________    _________________

    "Waypoints"        "Normal"    {[  1]}    {[  1]}
    "TimeOfArrival"    "None"      {[NaN]}    {[NaN]}
```

```
perturbs2 = perturbations(traj,'TimeOfArrival','Selection',{[0;1],[0;2]})
```

```
perturbs2=2×3 table
       Property           Type                    Value
    _______________    ___________    ____________________________________

    "Waypoints"        "Normal"       {[     1]}    {[            1]}
    "TimeOfArrival"    "Selection"    {1x2 cell}    {[0.5000 0.5000]}
```

Perturb the trajectory.

```
offsets = perturb(traj)
```

```
offsets=2×1 struct array with fields:
    Property
    Offset
    PerturbedValue
```

The `Waypoints` property and the `TimeOfArrival` property have changed.

```
traj.Waypoints
```

```
ans = 2×3

    1.8674    1.0203    0.7032
    2.3154   -0.3207    0.0999
```

```
traj.TimeOfArrival
```

```
ans = 2×1

     0
     2
```

Perturb Accuracy of insSensor

Create an `insSensor` object.

```
sensor = insSensor
```

```
sensor =
  insSensor with properties:

          MountingLocation: [0 0 0]           m
              RollAccuracy: 0.2               deg
             PitchAccuracy: 0.2               deg
               YawAccuracy: 1                 deg
          PositionAccuracy: [1 1 1]           m
          VelocityAccuracy: 0.05              m/s
      AccelerationAccuracy: 0                 m/s²
    AngularVelocityAccuracy: 0                deg/s
                 TimeInput: 0
              RandomStream: 'Global stream'
```

Define the perturbation on the `RollAccuracy` property as three values with an equal possibility each.

```
values = {0.1 0.2 0.3}
```

```
values=1×3 cell array
    {[0.1000]}    {[0.2000]}    {[0.3000]}
```

```
probabilities = [1/3 1/3 1/3]
```

```
probabilities = 1×3

    0.3333    0.3333    0.3333
```

```
perturbations(sensor,'RollAccuracy','Selection',values,probabilities)
```

```
ans=7×3 table
            Property                Type                        Value
    ____________________________    ___________    ____________________________________

    "RollAccuracy"                  "Selection"    {1x3 cell}    {[0.3333 0.3333 0.3333]}
    "PitchAccuracy"                 "None"         {[   NaN]}    {[                  NaN]}
    "YawAccuracy"                   "None"         {[   NaN]}    {[                  NaN]}
    "PositionAccuracy"              "None"         {[   NaN]}    {[                  NaN]}
    "VelocityAccuracy"              "None"         {[   NaN]}    {[                  NaN]}
    "AccelerationAccuracy"          "None"         {[   NaN]}    {[                  NaN]}
    "AngularVelocityAccuracy"       "None"         {[   NaN]}    {[                  NaN]}
```

Perturb the `sensor` object using the perturb function.

```
rng(2020)
perturb(sensor);
sensor
```

```
sensor =
  insSensor with properties:

           MountingLocation: [0 0 0]            m
               RollAccuracy: 0.5                deg
              PitchAccuracy: 0.2                deg
                YawAccuracy: 1                  deg
           PositionAccuracy: [1 1 1]            m
           VelocityAccuracy: 0.05               m/s
       AccelerationAccuracy: 0                  m/s²
    AngularVelocityAccuracy: 0                  deg/s
                  TimeInput: 0
               RandomStream: 'Global stream'
```

The `RollAccuracy` is perturbed to `0.5` deg.

Perturb imuSensor Parameters

Create an `imuSensor` object and show its perturbable properties.

```
imu = imuSensor;
perturbations(imu)
```

```
ans=17×3 table
                 Property                   Type           Value
    _____________________________________  ______  _________________

    "Accelerometer.MeasurementRange"       "None"    {[NaN]}    {[NaN]}
    "Accelerometer.Resolution"             "None"    {[NaN]}    {[NaN]}
    "Accelerometer.ConstantBias"           "None"    {[NaN]}    {[NaN]}
    "Accelerometer.NoiseDensity"           "None"    {[NaN]}    {[NaN]}
    "Accelerometer.BiasInstability"        "None"    {[NaN]}    {[NaN]}
    "Accelerometer.RandomWalk"             "None"    {[NaN]}    {[NaN]}
    "Accelerometer.TemperatureBias"        "None"    {[NaN]}    {[NaN]}
    "Accelerometer.TemperatureScaleFactor" "None"    {[NaN]}    {[NaN]}
    "Gyroscope.MeasurementRange"           "None"    {[NaN]}    {[NaN]}
    "Gyroscope.Resolution"                 "None"    {[NaN]}    {[NaN]}
    "Gyroscope.ConstantBias"               "None"    {[NaN]}    {[NaN]}
    "Gyroscope.NoiseDensity"               "None"    {[NaN]}    {[NaN]}
    "Gyroscope.BiasInstability"            "None"    {[NaN]}    {[NaN]}
    "Gyroscope.RandomWalk"                 "None"    {[NaN]}    {[NaN]}
    "Gyroscope.TemperatureBias"            "None"    {[NaN]}    {[NaN]}
    "Gyroscope.TemperatureScaleFactor"     "None"    {[NaN]}    {[NaN]}
      ⋮
```

Specify the perturbation for the `NoiseDensity` property of the accelerometer as a uniform distribution.

```
perturbations(imu,'Accelerometer.NoiseDensity', ...
    'Uniform',1e-5,1e-3);
```

Specify the perturbation for the `RandomWalk` property of the gyroscope as a truncated normal distribution.

```
 perts = perturbations(imu,'Gyroscope.RandomWalk', ...
    'TruncatedNormal',2,1e-5,0,Inf);
```

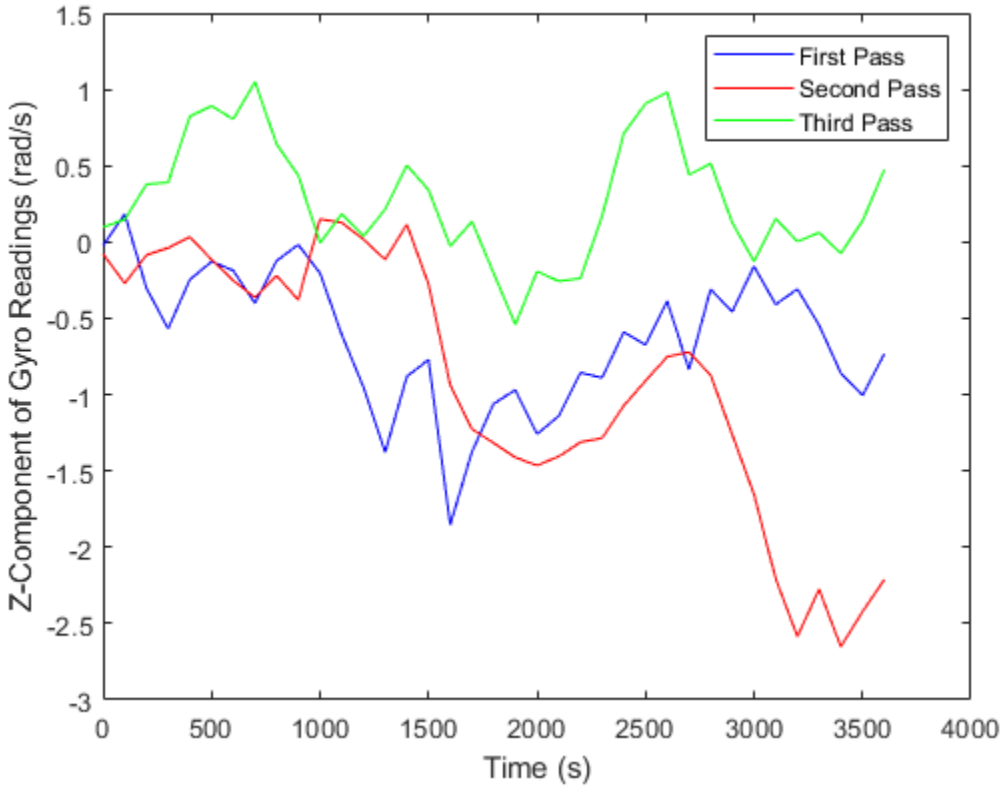
Load prerecorded IMU data.

```
load imuSensorData.mat
numSamples = size(orientations);
```

Simulate the `imuSensor` three times with different perturbation realizations.

```
rng(2021); % For repeatable results
numRuns = 3;
colors = ['b' 'r' 'g'];
for idx = 1:numRuns

    % Clone IMU to maintain original values
    imuCopy = clone(imu);

    % Perturb noise values
```

```
    offsets = perturb(imuCopy);

    % Obtain the measurements
    [accelReadings,gyroReadings] = imuCopy(accelerations,angularVelocities,orientations);

    % Plot the results
    plot(times,gyroReadings(:,3),colors(idx));
    hold on;
end
xlabel('Time (s)')
ylabel('Z-Component of Gyro Readings (rad/s)')
legend("First Pass","Second Pass","Third Pass");
hold off
```



Input Arguments

`obj` — Object for perturbation

objects

Object for perturbation, specified as an object. The objects that you can perturb include:

- `waypointTrajectory`
- `kinematicTrajectory`
- `insSensor`

- `imuSensor`

Output Arguments

`offsets` — Property offsets

array of structure

Property offsets, returned as an array of structures. Each structure contains these fields:

Field Name	Description
`Property`	Name of perturbed property
`Offset`	Offset values applied in the perturbation
`PerturbedValue`	Property values after the perturbation

See Also

`perturbations`

Introduced in R2020b

perturbations

Perturbation defined on object

Syntax

```
perturbs = perturbations(obj)
perturbs = perturbations(obj,property)
perturbs = perturbations(obj,property,'None')
perturbs = perturbations(obj,property,'Selection',values,probabilities)
perturbs = perturbations(obj,property,'Normal',mean,deviation)
perturbs = perturbations(obj,property,'TruncatedNormal',mean,deviation,
lowerLimit,upperLimit)
perturbs = perturbations(obj,property,'Uniform',minVal,maxVal)
perturbs = perturbations(obj,property,'Custom',perturbFcn)
```

Description

`perturbs = perturbations(obj)` returns the list of property perturbations, `perturbs`, defined on the object, `obj`. The returned `perturbs` lists all the perturbable properties. If any property is not perturbed, then its corresponding `Type` is returned as `"Null"` and its corresponding `Value` is returned as `{Null,Null}`.

`perturbs = perturbations(obj,property)` returns the current perturbation applied to the specified `property`.

`perturbs = perturbations(obj,property,'None')` defines a `property` that must not be perturbed.

`perturbs = perturbations(obj,property,'Selection',values,probabilities)` defines the `property` perturbation offset drawn from a set of `values` that have corresponding `probabilities`.

`perturbs = perturbations(obj,property,'Normal',mean,deviation)` defines the `property` perturbation offset drawn from a normal distribution with specified `mean` and standard `deviation`.

`perturbs = perturbations(obj,property,'TruncatedNormal',mean,deviation, lowerLimit,upperLimit)` defines the `property` perturbation offset drawn from a normal distribution with specified `mean`, standard `deviation`, lower limit, and upper limit.

`perturbs = perturbations(obj,property,'Uniform',minVal,maxVal)` defines the `property` perturbation offset drawn from a uniform distribution on an interval [`minVal`, `maxValue`].

`perturbs = perturbations(obj,property,'Custom',perturbFcn)` enables you to define a custom function, `perturbFcn`, that draws the perturbation offset value.

Examples

Default Perturbation Properties of `waypointTrajectory`

Create a `waypointTrajectory` object.

```
traj = waypointTrajectory;
```

Show the default perturbation properties using the `perturbations` method.

```
perturbs = perturbations(traj)
```

```
perturbs=2×3 table
       Property          Type            Value
    _______________    _______    ________________

    "Waypoints"        "None"    {[NaN]}    {[NaN]}
    "TimeOfArrival"    "None"    {[NaN]}    {[NaN]}
```

Perturb Accuracy of insSensor

Create an `insSensor` object.

```
sensor = insSensor
```

```
sensor =
  insSensor with properties:

           MountingLocation: [0 0 0]            m
               RollAccuracy: 0.2                deg
              PitchAccuracy: 0.2                deg
                YawAccuracy: 1                  deg
           PositionAccuracy: [1 1 1]            m
           VelocityAccuracy: 0.05               m/s
       AccelerationAccuracy: 0                  m/s²
    AngularVelocityAccuracy: 0                  deg/s
                  TimeInput: 0
               RandomStream: 'Global stream'
```

Define the perturbation on the `RollAccuracy` property as three values with an equal possibility each.

```
values = {0.1 0.2 0.3}
```

```
values=1×3 cell array
    {[0.1000]}    {[0.2000]}    {[0.3000]}
```

```
probabilities = [1/3 1/3 1/3]
```

```
probabilities = 1×3

    0.3333    0.3333    0.3333
```

```
perturbations(sensor,'RollAccuracy','Selection',values,probabilities)
```

```
ans=7×3 table
            Property                Type                        Value
    ____________________________    __________    ____________________________________

    "RollAccuracy"                  "Selection"    {1x3 cell}    {[0.3333 0.3333 0.3333]}
    "PitchAccuracy"                 "None"         {[   NaN]}    {[                  NaN]}
    "YawAccuracy"                   "None"         {[   NaN]}    {[                  NaN]}
    "PositionAccuracy"              "None"         {[   NaN]}    {[                  NaN]}
    "VelocityAccuracy"              "None"         {[   NaN]}    {[                  NaN]}
    "AccelerationAccuracy"          "None"         {[   NaN]}    {[                  NaN]}
    "AngularVelocityAccuracy"       "None"         {[   NaN]}    {[                  NaN]}
```

Perturb the `sensor` object using the perturb function.

```
rng(2020)
perturb(sensor);
sensor
```

```
sensor =
  insSensor with properties:

          MountingLocation: [0 0 0]           m
              RollAccuracy: 0.5               deg
             PitchAccuracy: 0.2               deg
               YawAccuracy: 1                 deg
          PositionAccuracy: [1 1 1]           m
          VelocityAccuracy: 0.05              m/s
       AccelerationAccuracy: 0                m/s²
    AngularVelocityAccuracy: 0                deg/s
                  TimeInput: 0
               RandomStream: 'Global stream'
```

The `RollAccuracy` is perturbed to `0.5` deg.

Perturb Waypoint Trajectory

Define a waypoint trajectory. By default, this trajectory contains two waypoints.

```
traj = waypointTrajectory
```

```
traj =
  waypointTrajectory with properties:

         SampleRate: 100
    SamplesPerFrame: 1
          Waypoints: [2x3 double]
      TimeOfArrival: [2x1 double]
         Velocities: [2x3 double]
             Course: [2x1 double]
        GroundSpeed: [2x1 double]
          ClimbRate: [2x1 double]
        Orientation: [2x1 quaternion]
          AutoPitch: 0
           AutoBank: 0
```

```
     ReferenceFrame: 'NED'
```

Define perturbations on the `Waypoints` property and the `TimeOfArrival` property.

```
rng(2020);
perturbs1 = perturbations(traj,'Waypoints','Normal',1,1)
```

```
perturbs1=2×3 table
       Property          Type            Value
    _______________    ________    ________________

    "Waypoints"        "Normal"    {[  1]}    {[  1]}
    "TimeOfArrival"    "None"      {[NaN]}    {[NaN]}
```

```
perturbs2 = perturbations(traj,'TimeOfArrival','Selection',{[0;1],[0;2]})
```

```
perturbs2=2×3 table
       Property          Type                    Value
    _______________    ___________    ____________________________

    "Waypoints"        "Normal"       {[     1]}    {[            1]}
    "TimeOfArrival"    "Selection"    {1x2 cell}    {[0.5000 0.5000]}
```

Perturb the trajectory.

```
offsets = perturb(traj)
```

```
offsets=2×1 struct array with fields:
    Property
    Offset
    PerturbedValue
```

The `Waypoints` property and the `TimeOfArrival` property have changed.

```
traj.Waypoints
```

```
ans = 2×3

    1.8674    1.0203    0.7032
    2.3154   -0.3207    0.0999
```

```
traj.TimeOfArrival
```

```
ans = 2×1

     0
     2
```

Perturb `imuSensor` Parameters

Create an `imuSensor` object and show its perturbable properties.

```
imu = imuSensor;
perturbations(imu)
```

```
ans=17×3 table
                  Property                     Type           Value
    ______________________________________    ______    __________________

    "Accelerometer.MeasurementRange"          "None"    {[NaN]}    {[NaN]}
    "Accelerometer.Resolution"                "None"    {[NaN]}    {[NaN]}
    "Accelerometer.ConstantBias"              "None"    {[NaN]}    {[NaN]}
    "Accelerometer.NoiseDensity"              "None"    {[NaN]}    {[NaN]}
    "Accelerometer.BiasInstability"           "None"    {[NaN]}    {[NaN]}
    "Accelerometer.RandomWalk"                "None"    {[NaN]}    {[NaN]}
    "Accelerometer.TemperatureBias"           "None"    {[NaN]}    {[NaN]}
    "Accelerometer.TemperatureScaleFactor"    "None"    {[NaN]}    {[NaN]}
    "Gyroscope.MeasurementRange"              "None"    {[NaN]}    {[NaN]}
    "Gyroscope.Resolution"                    "None"    {[NaN]}    {[NaN]}
    "Gyroscope.ConstantBias"                  "None"    {[NaN]}    {[NaN]}
    "Gyroscope.NoiseDensity"                  "None"    {[NaN]}    {[NaN]}
    "Gyroscope.BiasInstability"               "None"    {[NaN]}    {[NaN]}
    "Gyroscope.RandomWalk"                    "None"    {[NaN]}    {[NaN]}
    "Gyroscope.TemperatureBias"               "None"    {[NaN]}    {[NaN]}
    "Gyroscope.TemperatureScaleFactor"        "None"    {[NaN]}    {[NaN]}
      ⋮
```

Specify the perturbation for the `NoiseDensity` property of the accelerometer as a uniform distribution.

```
perturbations(imu,'Accelerometer.NoiseDensity', ...
    'Uniform',1e-5,1e-3);
```

Specify the perturbation for the `RandomWalk` property of the gyroscope as a truncated normal distribution.

```
 perts = perturbations(imu,'Gyroscope.RandomWalk', ...
    'TruncatedNormal',2,1e-5,0,Inf);
```

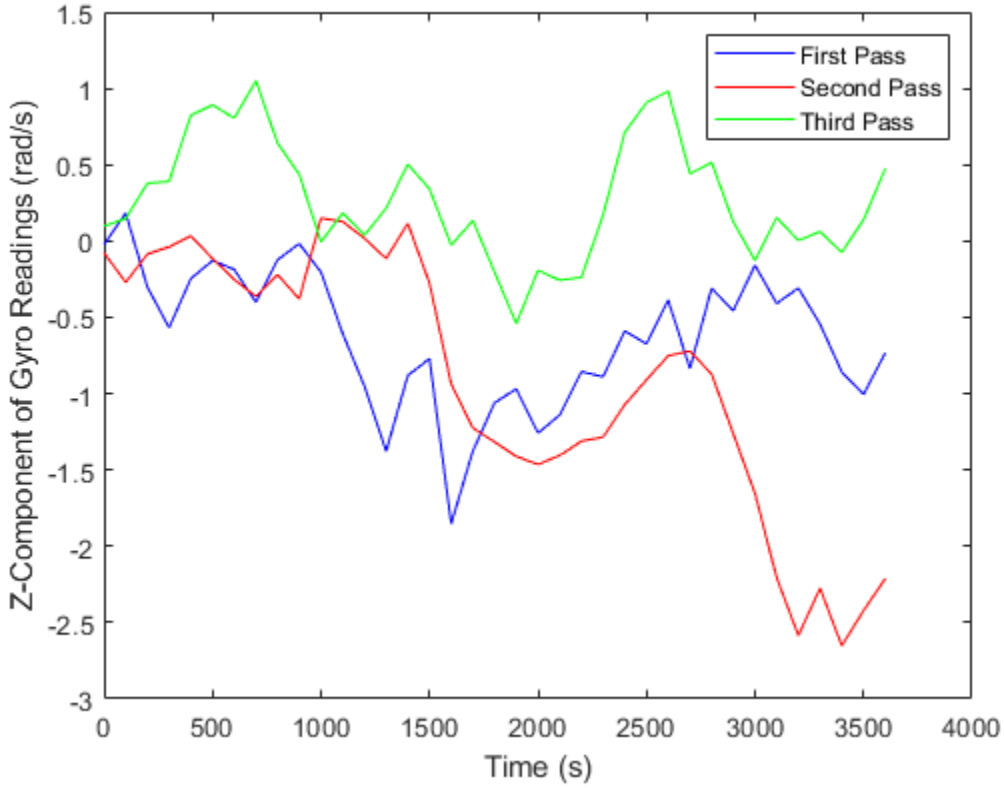
Load prerecorded IMU data.

```
load imuSensorData.mat
numSamples = size(orientations);
```

Simulate the `imuSensor` three times with different perturbation realizations.

```
rng(2021); % For repeatable results
numRuns = 3;
colors = ['b' 'r' 'g'];
for idx = 1:numRuns

    % Clone IMU to maintain original values
    imuCopy = clone(imu);

    % Perturb noise values
    offsets = perturb(imuCopy);

    % Obtain the measurements
    [accelReadings,gyroReadings] = imuCopy(accelerations,angularVelocities,orientations);
```

```
    % Plot the results
    plot(times,gyroReadings(:,3),colors(idx));
    hold on;
end
xlabel('Time (s)')
ylabel('Z-Component of Gyro Readings (rad/s)')
legend("First Pass","Second Pass","Third Pass");
hold off
```



Input Arguments

`obj` — Object to be perturbed

objects

Object to be perturbed, specified as an object. The objects that you can perturb include:

- `waypointTrajectory`
- `kinematicTrajectory`
- `insSensor`
- `imuSensor`

`property` — Perturbable property

property name

Perturbable property, specified as a property name. Use `perturbations` to obtain a full list of perturbable properties for the specified `obj`.

For the `imuSensor` System object, you can perturb properties of its accelerometer, gyroscope, and magnetometer components. For more details, see the “Perturb imuSensor Parameters” on page 2-1446 example.

`values` — Perturbation offset values
n-element cell array of property values

Perturbation offset values, specified as an *n*-element cell array of property values. The function randomly draws the perturbation value for the property from the cell array based on the values' corresponding probabilities specified in the `probabilities` input.

`probabilities` — Drawing probabilities for each perturbation value
n-element array of nonnegative scalar

Drawing probabilities for each perturbation value, specified as an *n*-element array of nonnegative scalars, where *n* is the number of perturbation values provided in the `values` input. The sum of all elements must be equal to one.

For example, you can specify a series of perturbation value-probability pair as `{x1,x2,...,xn}` and `{p1,p2,...,pn}`, where the probability of drawing `xi` is `pi` (`i` = 1, 2, ...,`n`).

`mean` — Mean of normal or truncated normal distribution
scalar | vector | matrix

Mean of normal or truncated normal distribution, specified as a scalar, vector, or matrix. The dimension of `mean` must be compatible with the corresponding property that you perturb.

`deviation` — Standard deviation of normal or truncated normal distribution
nonnegative scalar | vector of nonnegative scalar | matrix of nonnegative scalar

Standard deviation of normal or truncated normal distribution, specified as a nonnegative scalar, vector of nonnegative scalars, or matrix of nonnegative scalars. The dimension of `deviation` must be compatible with the corresponding property that you perturb.

`lowerLimit` — Lower limit of truncated normal distribution
scalar | vector | matrix

Lower limit of the truncated normal distribution, specified as a scalar, vector, or matrix. The dimension of `lowerLimit` must be compatible with the corresponding property that you perturb.

`upperLimit` — Upper limit of truncated normal distribution
scalar | vector | matrix

Upper limit of the truncated normal distribution, specified as a scalar, vector, or matrix. The dimension of `upperLimit` must be compatible with the corresponding property that you perturb.

`minVal` — Minimum value of uniform distribution interval
scalar | vector | matrix

Minimum value of the uniform distribution interval, specified as a scalar, vector, or matrix. The dimension of `minVal` must be compatible with the corresponding property that you perturb.

`maxVal` — Maximum value of uniform distribution interval
scalar | vector | matrix

Maximum value of the uniform distribution interval, specified as a scalar, vector, or matrix. The dimension of `maxVal` must be compatible with the corresponding property that you perturb.

`perturbFcn` — Perturbation function
function handle

Perturbation function, specified as a function handle. The function must have this syntax:

```
offset = myfun(propVal)
```

where `propVal` is the value of the `property` and `offset` is the perturbation offset for the property.

Output Arguments

`perturbs` — Perturbations defined on object
table of perturbation property

Perturbations defined on the object, returned as a table of perturbation properties. The table has three columns:

- `Property` — Property names.
- `Type` — Type of perturbations, returned as `"None"`, `"Selection"`, `"Normal"`, `"TruncatedNormal"`, `"Uniform"`, or `"Custom"`.
- `Value` — Perturbation values, returned as a cell array.

More About

Specify Perturbation Distributions

You can specify the distribution for the perturbation applied to a specific property.

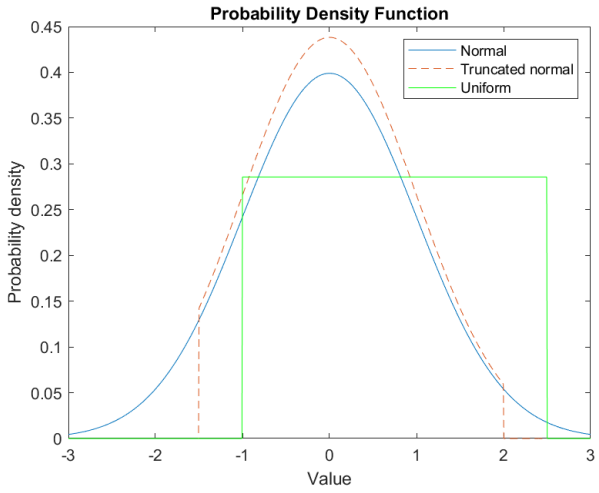
- Selection distribution — The function defines the perturbation offset as one of the specified values with the associated probability. For example, if you specify the values as `[1 2]` and specify the probabilities as `[0.7 0.3]`, then the `perturb` function adds an offset value of `1` to the property with a probability of `0.7` and add an offset value of `2` to the property with a probability of `0.3`. Use selection distribution when you only want to perturb the property with a number of discrete values.
- Normal distribution — The function defines the perturbation offset as a value drawn from a normal distribution with the specified mean and standard deviation (or covariance). Normal distribution is the most commonly used distribution since it mimics the natural perturbation of parameters in most cases.
- Truncated normal distribution — The function defines the perturbation offset as a value drawn from a truncated normal distribution with the specified mean, standard deviation (or covariance), lower limit, and upper limit. Different from the normal distribution, the values drawn from a truncated normal distribution are truncated by the lower and upper limit. Use truncated normal distribution when you want to apply a normal distribution, but the valid values of the property are confined in an interval.

- Uniform distribution — The function defines the perturbation offset as a value drawn from a uniform distribution with the specified minimum and maximum values. All the values in the interval (specified by the minimum and maximum values) have the same probability of realization.
- Custom distribution — Customize your own perturbation function. The function must have this syntax:

  ```
  offset = myfun(propVal)
  ```

  where `propVal` is the value of the `property` and `offset` is the perturbation offset for the property.

This figure shows probability density functions for a normal distribution, a truncated normal distribution, and a uniform distribution, respectively.



See Also

`perturb`

Introduced in R2020b

wheelEncoderAckermann

Simulate wheel encoder sensor readings for Ackermann vehicle

Description

The `wheelEncoderAckermann` System object computes wheel encoder tick readings based on the pose input of an Ackermann vehicle.

To obtain the encoder tick readings:

1. Create the `wheelEncoderAckermann` object and set its properties.
2. Call the object with arguments, as if it were a function.

To learn more about how System objects work, see What Are System Objects?

Creation

Syntax

```
encoder = wheelEncoderAckermann
encoder= wheelEncoderAckermann(Name,Value)
```

Description

`encoder = wheelEncoderAckermann` creates a `wheelEncoderAckermann` System object, `encoder`.

`encoder= wheelEncoderAckermann(Name,Value)` sets properties for the encoder using one or more name-value pairs. For example, `wheelAckermann('SampleRate',120)` sets the sample rate of the encoder to 120 Hz. Unspecified properties have default values. Enclose each property name in quotes.

Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see System Design in MATLAB Using System Objects.

`SampleRate` — Sample rate of encoder

`100` (default) | positive scalar

Sample rate of the encoder, specified as a positive scalar in Hz.

Example: `'SampleRate',100`

Data Types: `double`

`TicksPerRevolution` — Number of encoder ticks per wheel revolution
`[2048 2048 2048 2048]` (default) | four-element vector of positive integers

Number of encoder ticks per wheel revolution, specified as a four-element vector of positive integers. The first, second, third, and fourth elements are for the back-left, back-right, front-left, and front-right wheels, respectively.

Data Types: `double`

`WheelRadius` — Wheel radius
`[0.35 0.35 0.35 0.35]` (default) | four-element vector of positive scalars

Wheel radius, specified as a four-element vector of positive scalars in meters. The first, second, third, and fourth elements are for the back-left, back-right, front-left, and front-right wheels, respectively.

Data Types: `double`

`WheelRadiusBias` — Bias of wheel radius
`[0 0 0 0]` (default) | four-element vector of scalars

Bias of the wheel radius, specified as a four-element vector of scalars in meters. The first, second, third, and fourth elements are for the back-left, back-right, front-left, and front-right wheels, respectively.

Data Types: `double`

`WheelPositionAccuracy` — Standard deviation of wheel position error
`[0 0 0 0]` (default) | four-element vector of nonnegative scalars

Standard deviation of wheel position error, specified as a four-element vector of nonnegative scalars in radians. The first, second, third, and fourth elements are for the back-left, back-right, front-left, and front-right wheels, respectively.

Data Types: `double`

`SlipRatio` — Slip or skid ratio of wheel
`[0 0 0 0]` (default) | four-element vector of scalars

Slip or skid ratio of the wheel, specified as a four-element vector of scalars in which each scalar is larger than or equal to –1. The first, second, third, and fourth elements are for the back-left, back-right, front-left, and front-right wheels, respectively.

- For a wheel that slips (over rotation), specify it as a positive value. A higher value denotes more slipping.
- For a wheel that skids (under rotation), specify it as a negative value larger than or equal to –1. A lower value denotes more skidding. For a wheel that does not rotate, specify it as –1.

Data Types: `double`

`TrackWidth` — Distance between wheel axles
`[1.572 1.572]` (default) | two-element vector of positive scalars

Distance between the wheel axles, specified as a two-element vector of positive scalars in meters. The first element is for the back track, and the second element is for the front track.

Data Types: `double`

`TrackWidthBias` — Bias of track width
`0` (default) | two-element vector of scalars

Bias of track width, specified as a two-element vector of scalars in meters. The first element is for the back track, and the second element is for the front track.

Data Types: `double`

`WheelBase` — Distance between front and rear axles
`2.818` (default) | positive scalar

Distance between the front and the rear axles, specified as a positive scalar in meters.

Data Types: `double`

`RandomStream` — Random number source
`'Global stream'` (default) | `'mt19937ar with seed'`

Random number source, specified as a character vector or string:

- `'Global stream'` –– Random numbers are generated using the current global random number stream.
- `'mt19937ar with seed'` –– Random numbers are generated using the mt19937ar algorithm with the seed specified by the `Seed` property.

Data Types: `char` | `string`

`Seed` — Initial seed
`67` (default) | nonnegative integer

Initial seed of an mt19937ar random number generator algorithm, specified as a nonnegative integer.

Dependencies

To enable this property, set `RandomStream` to `'mt19937ar with seed'`.

Usage

Syntax

```
ticks = encoder(velocity,angularVelocity,orientation)
```

Description

`ticks = encoder(velocity,angularVelocity,orientation)` return the wheel tick readings, `ticks`, form velocity, angular velocity, and orientation information.

Input Arguments

`velocity` — Velocity of vehicle
N-by-3 matrix of scalars

Velocity of the vehicle in the local navigation frame, specified as an *N*-by-3 matrix of scalars in m/s. *N* is the number of samples.

`angularVelocity` — Angular velocity of vehicle
N-by-3 matrix of scalars

Angular velocity of the vehicle in the local navigation frame, specified as an *N*-by-3 matrix of scalars in rad/s. *N* is the number of samples.

`orientation` — orientation of vehicle
N-element vector of quaternion | 3-by-3-by-*N* array of rotation matrices

Orientation of the vehicle in the local navigation frame, specified as an *N*-element vector of `quaternion` or a 3-by-3-by-*N* array of rotation matrices. *N* is the number of samples. Each quaternion or rotation matrix is a frame rotation from the local navigation coordinate system to the current vehicle body coordinate system.

Output Arguments

`ticks` — Number of wheel ticks per time step
N-by-4 matrix of nonnegative integers

Number of wheel ticks the vehicle moved per time step, returned as an *N*-by-4 matrix of nonnegative integers. *N* is the number of samples. The first, second, third, and fourth columns are for the back-left, back-right, front-left, and front-right wheels, respectively.

Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

Common to All System Objects

clone	Create duplicate System object
step	Run System object algorithm
release	Release resources and allow changes to System object property values and input characteristics
reset	Reset internal states of System object
isLocked	Determine if System object is in use

Examples

Generate Wheel Ticks from Ackermann Vehicle Pose

Create the wheel encoder sensor.

```
encoder = wheelEncoderAckermann;
```

Define poses of the vehicle.

```
orient = [quaternion([60 0 0],'eulerd','ZYX','frame'); quaternion([45 0 0], 'eulerd', 'ZYX', 'fra
vel = [1 0 0; 0 1 0];
angvel = [0 0 0.2; 0 0 0.1];
```

Generate wheel ticks from the poses.

```
ticks = encoder(vel,angvel,orient)
```

```
ticks = 2×4

     3     6     6     8
     6     7     6     7
```

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

Introduced in R2020b

wheelEncoderBicycle

Simulate wheel encoder sensor readings for bicycle vehicle

Description

The `wheelEncoderBicycle` System object computes wheel encoder tick readings based on the pose input for a bicycle vehicle.

To obtain the encoder tick readings:

1. Create the `wheelEncoderBicycle` object and set its properties.
2. Call the object with arguments, as if it were a function.

To learn more about how System objects work, see What Are System Objects?

Creation

Syntax

```
encoder = wheelEncoderBicycle
encoder= wheelEncoderBicycle(Name,Value)
```

Description

`encoder = wheelEncoderBicycle` creates a `wheelEncoderBicycle` System object, `encoder`.

`encoder= wheelEncoderBicycle(Name,Value)` sets properties for the encoder using one or more name-value pairs. For example, `wheelEncoderBicycle('SampleRate',120)` sets the sample rate of the encoder to 120 Hz. Unspecified properties have default values. Enclose each property name in quotes.

Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see System Design in MATLAB Using System Objects.

`SampleRate` — Sample rate of encoder

`100` (default) | positive scalar

Sample rate of the encoder, specified as a positive scalar in Hz.

Data Types: `double`

`TicksPerRevolution` — Number of encoder ticks per wheel revolution
`[2048 2048]` (default) | two-element vector of positive integers

Number of encoder ticks per wheel revolution, specified as a two-element vector of positive integers. The first element is for the back wheel, and the second element is for the front wheel.

Data Types: `double`

`WheelRadius` — Wheel radius
`[0.35 0.35]` (default) | two-element vector of positive scalars

Wheel radius, specified as a two-element vector of positive scalars in meters. The first element is for the back wheel, and the second element is for the front wheel.

Data Types: `double`

`WheelRadiusBias` — Bias of wheel radius
`[0 0]` (default) | two-element vector of scalars

Bias of the wheel radius, specified as a two-element vector of scalars in meters. The first element is for the back wheel, and the second element is for the front wheel.

Data Types: `double`

`WheelPositionAccuracy` — Standard deviation of wheel position error
`[0 0]` (default) | two-element vector of nonnegative scalars

Standard deviation of wheel position error, specified as a two-element vector of nonnegative scalars in radians. The first element is for the back wheel, and the second element is for the front wheel.

Data Types: `double`

`SlipRatio` — Slip or skid ratio of wheel
`[0 0]` (default) | two-element vector of scalar

Slip or skid ratio of the wheel, specified as a two-element vector of scalars in which each scalar is larger than or equal to –1. The first element is for the back wheel, and the second element is for the front wheel.

- For a wheel that slips (over rotation), specify it as a positive value. A higher value denotes more slipping.
- For a wheel that skids (under rotation), specify it as a negative value larger than or equal to –1. A lower value denotes more skidding. For a wheel that does not rotate, specify it as –1.

Data Types: `double`

`WheelBase` — Distance between front and rear wheels
`2.818` (default) | positive scalar

Distance between the front and the rear wheels, specified as a positive scalar in meters.

Data Types: `double`

`RandomStream` — Random number source
`'Global stream'` (default) | `'mt19937ar with seed'`

Random number source, specified as a character vector or string:

- `'Global stream'` –– Random numbers are generated using the current global random number stream.
- `'mt19937ar with seed'` –– Random numbers are generated using the mt19937ar algorithm with the seed specified by the `Seed` property.

Data Types: `char` | `string`

Seed — Initial seed of mt19937ar random number generator
67 (default) | nonnegative integer

Initial seed of an mt19937ar random number generator algorithm, specified as a nonnegative integer.

Dependencies

To enable this property, set `RandomStream` to `'mt19937ar with seed'`.

Usage

Syntax

```
ticks = encoder(velocity,angularVelocity,orientation)
```

Description

`ticks = encoder(velocity,angularVelocity,orientation)` return the wheel tick readings, `ticks`, form velocity, angular velocity, and orientation information.

Input Arguments

`velocity` — Velocity of vehicle
N-by-3 matrix of scalars

Velocity of the vehicle in the local navigation frame, specified as an *N*-by-3 matrix of scalars in m/s. *N* is the number of samples.

`angularVelocity` — Angular velocity of vehicle
N-by-3 matrix of scalars

Angular velocity of the vehicle in the local navigation frame, specified as an *N*-by-3 matrix of scalars in rad/s. *N* is the number of samples.

`orientation` — orientation of vehicle
N-element vector of quaternion | 3-by-3-by-*N* array of rotation matrices

Orientation of the vehicle in the local navigation frame, specified as an *N*-element vector of `quaternion` or a 3-by-3-by-*N* array of rotation matrices. *N* is the number of samples. Each quaternion or rotation matrix is a frame rotation from the local navigation coordinate system to the current vehicle body coordinate system.

Output Arguments

`ticks` — Number of wheel ticks per time step
N-by-2 matrix of integer

Number of wheel ticks the vehicle moved per time step, returned as an *N*-by-2 matrix of integers. *N* is the number of samples. The first column is for the back wheel, and the second column is for the front wheel.

Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

Common to All System Objects

clone	Create duplicate System object
step	Run System object algorithm
release	Release resources and allow changes to System object property values and input characteristics
reset	Reset internal states of System object
isLocked	Determine if System object is in use

Examples

Generate Wheel Ticks from Bicycle Vehicle Pose

Create the wheel encoder sensor.

```
encoder = wheelEncoderBicycle;
```

Define poses of the vehicle.

```
orient = [quaternion([90 0 0],'eulerd','ZYX','frame'); quaternion([45 0 0], 'eulerd', 'ZYX', 'fra
vel = [1 0 0; 0 1 0];
angvel = [0 0 0.2; 0 0 0.1];
```

Generate wheel ticks from the poses.

```
ticks = encoder(vel,angvel,orient)
```

```
ticks = 2×2

     0     5
     6     7
```

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

Introduced in R2020b

wheelEncoderDifferentialDrive

Simulate wheel encoder sensor readings for differential drive vehicle

Description

The `wheelEncoderDifferentialDrive` System object computes wheel encoder tick readings based on the pose input of a differential drive vehicle.

To obtain the encoder tick readings:

1. Create the `wheelEncoderDifferentialDrive` object and set its properties.
2. Call the object with arguments, as if it were a function.

To learn more about how System objects work, see What Are System Objects?

Creation

Syntax

```
encoder = wheelEncoderDifferentialDrive
encoder= wheelEncoderDifferentialDrive(Name,Value)
```

Description

`encoder = wheelEncoderDifferentialDrive` creates a `wheelEncoderDifferentialDrive` System object, `encoder`.

`encoder= wheelEncoderDifferentialDrive(Name,Value)` sets properties for the encoder using one or more name-value pairs. For example, `wheelEncoderDifferentialDrive('SampleRate',120)` sets the sample rate of the encoder to 120 Hz. Unspecified properties have default values. Enclose each property name in quotes.

Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see System Design in MATLAB Using System Objects.

`SampleRate` — Sample rate of encoder

`100` (default) | positive scalar

Sample rate of the encoder, specified as a positive scalar in Hz.

Data Types: `double`

`TicksPerRevolution` — Number of encoder ticks per wheel revolution
`[2048 2048]` (default) | two-element vector of positive integers

Number of encoder ticks per wheel revolution, specified as a two-element vector of positive integers. The first element is for the left wheel, and the second element is for the right wheel.

Data Types: `double`

`WheelRadius` — Wheel radius
`[0.35 0.35]` (default) | two-element vector of positive scalars

Wheel radius, specified as a two-element vector of positive scalars in meters. The first element is for the left wheel, and the second element is for the right wheel.

Data Types: `double`

`WheelRadiusBias` — Bias of wheel radius
`[0 0]` (default) | two-element vector of scalars

Bias of the wheel radius, specified as a two-element vector of scalars in meters. The first element is for the left wheel, and the second element is for the right wheel.

Data Types: `double`

`WheelPositionAccuracy` — Standard deviation of wheel position error
`[0 0]` (default) | two-element vector of nonnegative scalar

Standard deviation of wheel position error, specified as a two-element vector of nonnegative scalars in radians. The first element is for the left wheel, and the second element is for the right wheel.

Data Types: `double`

`SlipRatio` — Slip or skid ratio of wheel
`[0 0]` (default) | two-element vector of scalar

Slip or skid ratio of the wheel, specified as a two-element vector of scalars in which each scalar is larger than or equal to –1. The first element is for the left wheel, and the second element is for the right wheel.

- For a wheel that slips (over rotation), specify it as a positive value. A higher value denotes more slipping.
- For a wheel that skids (under rotation), specify it as a negative value larger than or equal to –1. A lower value denotes more skidding. For a wheel that does not rotate, specify it as –1.

Data Types: `double`

`TrackWidth` — Distance between wheel axles
`1.572` (default) | positive scalar

Distance between the wheel axles, specified as a positive scalar in meters.

Data Types: `double`

`TrackWidthBias` — Bias of track width
`0` (default) | scalar

Bias of track width, specified as a scalar in meters.

Data Types: `double`

`RandomStream` — Random number source
`'Global stream'` (default) | `'mt19937ar with seed'`

Random number source, specified as a character vector or string:

- `'Global stream'` –– Random numbers are generated using the current global random number stream.
- `'mt19937ar with seed'` –– Random numbers are generated using the mt19937ar algorithm with the seed specified by the `Seed` property.

Data Types: `char` | `string`

Seed — Initial seed
67 (default) | nonnegative integer

Initial seed of an mt19937ar random number generator algorithm, specified as a nonnegative integer.

Dependencies

To enable this property, set `RandomStream` to `'mt19937ar with seed'`.

Usage

Syntax

```
ticks = encoder(velocity,angularVelocity,orientation)
```

Description

`ticks = encoder(velocity,angularVelocity,orientation)` return the wheel tick readings, `ticks`, form velocity, angular velocity, and orientation information.

Input Arguments

`velocity` — Velocity of vehicle
N-by-3 matrix of scalars

Velocity of the vehicle in the local navigation frame, specified as an *N*-by-3 matrix of scalars in m/s. *N* is the number of samples.

`angularVelocity` — Angular velocity of vehicle
N-by-3 matrix of scalars

Angular velocity of the vehicle in the local navigation frame, specified as an *N*-by-3 matrix of scalars in rad/s. *N* is the number of samples.

`orientation` — orientation of vehicle
N-element vector of quaternion | 3-by-3-by-*N* array of rotation matrices

Orientation of the vehicle in the local navigation frame, specified as an *N*-element vector of `quaternion` or a 3-by-3-by-*N* array of rotation matrices. *N* is the number of samples. Each quaternion or rotation matrix is a frame rotation from the local navigation coordinate system to the current vehicle body coordinate system.

Output Arguments

`ticks` — Number of wheel ticks per time step

N-by-2 matrix of nonnegative integers

Number of wheel ticks the vehicle moved per time step, returned as an *N*-by-2 matrix of integers. *N* is the number of samples. The first column is for the left wheel, and the second column is for the right wheel.

Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

Common to All System Objects

clone	Create duplicate System object
step	Run System object algorithm
release	Release resources and allow changes to System object property values and input characteristics
reset	Reset internal states of System object
isLocked	Determine if System object is in use

Examples

Generate Wheel Ticks from Differential Drive Vehicle Pose

Create the wheel encoder sensor.

```
encoder = wheelEncoderDifferentialDrive;
```

Define poses of the vehicle.

```
orient = [quaternion([60 0 0],'eulerd','ZYX','frame'); quaternion([45 0 0], 'eulerd', 'ZYX', 'fra
vel = [1 0 0; 0 1 0];
angvel = [0 0 0.2; 0 0 0.1];
```

Generate wheel ticks from the poses.

```
ticks = encoder(vel,angvel,orient)
```

```
ticks = 2×2

     3     6
     6     7
```

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

Introduced in R2020b

wheelEncoderUnicycle

Simulate wheel encoder sensor readings for unicycle vehicle

Description

The `wheelEncoderUnicycle` System object computes wheel encoder tick readings based on the pose input of a unicycle vehicle.

To obtain the encoder tick readings:

1. Create the `wheelEncoderUnicycle` object and set its properties.
2. Call the object with arguments, as if it were a function.

To learn more about how System objects work, see What Are System Objects?

Creation

Syntax

```
encoder = wheelEncoderUnicycle
encoder = wheelEncoderUnicycle(Name,Value)
```

Description

`encoder = wheelEncoderUnicycle` creates a `wheelEncoderUnicycle` System object `encoder`.

`encoder = wheelEncoderUnicycle(Name,Value)` sets properties for the encoder using one or more name-value pairs. For example, `wheelEncoderUnicycle('SampleRate',120)` sets the sample rate of the encoder to 120 Hz. Unspecified properties have default values. Enclose each property name in quotes.

Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see System Design in MATLAB Using System Objects.

`SampleRate` — Sample rate of encoder

`100` (default) | positive scalar

Sample rate of the encoder, specified as a positive scalar in Hz.

Data Types: `double`

`TicksPerRevolution` — Number of encoder ticks per wheel revolution
`2048` (default) | positive integer

Number of encoder ticks per wheel revolution, specified as a positive integer.

Data Types: `double`

`WheelRadius` — Wheel radius
`0.35` (default) | positive scalar

Wheel radius, specified as a positive scalar in meters.

Data Types: `double`

`WheelRadiusBias` — Bias of wheel radius
`0` (default) | scalar

Bias of the wheel radius, specified as a scalar in meters.

Data Types: `double`

`WheelPositionAccuracy` — Standard deviation of wheel position error
`0` (default) | nonnegative scalar

Standard deviation of wheel position error, specified as a nonnegative scalar in radians.

Data Types: `double`

`SlipRatio` — Slip or skid ratio of wheel
`0` (default) | scalar

Slip or skid ratio of the wheel, specified as a scalar larger than or equal to –1.

- For a wheel that slips (over rotation), specify it as a positive value. A higher value denotes more slipping.
- For a wheel that skids (under rotation), specify it as a negative value larger than or equal to –1. A lower value denotes more skidding. For a wheel that does not rotate, specify it as –1.

Data Types: `double`

`RandomStream` — Random number source
`'Global stream'` (default) | `'mt19937ar with seed'`

Random number source, specified as a character vector or string:

- `'Global stream'` –– Random numbers are generated using the current global random number stream.
- `'mt19937ar with seed'` –– Random numbers are generated using the mt19937ar algorithm with the seed specified by the `Seed` property.

Data Types: `char` | `string`

`Seed` — Initial seed of mt19937ar random number generator algorithm
`67` (default) | nonnegative integer

Initial seed of an mt19937ar random number generator algorithm, specified as a nonnegative integer.

Dependencies

To enable this property, set `RandomStream` to `'mt19937ar with seed'`.

Usage

Syntax

```
ticks = encoder(velocity,angularVelocity,orientation)
```

Description

`ticks = encoder(velocity,angularVelocity,orientation)` return the wheel tick readings `ticks` from the specified velocity, angular velocity, and orientation information.

Input Arguments

`velocity` — Velocity of vehicle
N-by-3 matrix of scalars

Velocity of the vehicle in the local navigation frame, specified as an *N*-by-3 matrix of scalars in m/s. *N* is the number of samples.

`angularVelocity` — Angular velocity of vehicle
N-by-3 matrix of scalars

Angular velocity of the vehicle in the local navigation frame, specified as an *N*-by-3 matrix of scalars in rad/s. *N* is the number of samples.

`orientation` — orientation of vehicle
N-element vector of quaternion | 3-by-3-by-*N* array of rotation matrices

Orientation of the vehicle in the local navigation frame, specified as an *N*-element vector of `quaternion` or a 3-by-3-by-*N* array of rotation matrices. *N* is the number of samples. Each quaternion or rotation matrix is a frame rotation from the local navigation coordinate system to the current vehicle body coordinate system.

Output Arguments

`ticks` — Number of wheel ticks per time step
N-element vector of nonnegative integers

Number of wheel ticks the vehicle moved per time step, returned as an *N*-element vector of nonnegative integers. *N* is the number of samples.

Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

Common to All System Objects

clone	Create duplicate System object
step	Run System object algorithm
release	Release resources and allow changes to System object property values and input characteristics
reset	Reset internal states of System object
isLocked	Determine if System object is in use

Examples

Generate Wheel Ticks from Unicycle Vehicle Pose

Create the wheel encoder sensor.

```
encoder = wheelEncoderUnicycle;
```

Define poses of the vehicle.

```
orient = [quaternion([90 0 0],'eulerd','ZYX','frame'); quaternion([45 0 0], 'eulerd', 'ZYX', 'fra
vel = [1 0 0; 0 1 0];
angvel = [0 0 0.2; 0 0 0.1];
```

Generate wheel ticks from the poses.

```
ticks = encoder(vel,angvel,orient)
```

```
ticks = 2×1

     0
     6
```

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

Introduced in R2020b

wheelEncoderOdometryAckermann

Compute Ackermann vehicle odometry using wheel encoder ticks and steering angle

Description

The `wheelEncoderOdometryAckermann` System object computes Ackermann vehicle odometry using the wheel encoder ticks and steering angle of the vehicle.

To compute Ackermann vehicle odometry:

1. Create the `wheelEncoderOdometryAckermann` object and set its properties.
2. Call the object with arguments, as if it were a function.

To learn more about how System objects work, see What Are System Objects?

Creation

Syntax

```
whlEncOdom = wheelEncoderOdometryAckermann
whlEncOdom = wheelEncoderOdometryAckermann(encoder)
whlEncOdom = wheelEncoderOdometryAckermann(Name,Value)
```

Description

`whlEncOdom = wheelEncoderOdometryAckermann` creates a `wheelEncoderOdometryAckermann` System object with default property values.

`whlEncOdom = wheelEncoderOdometryAckermann(encoder)` creates a `wheelEncoderOdometryAckermann` System object using the specified `wheelEncoderAckermann` System object, `encoder`, to set properties.

`whlEncOdom = wheelEncoderOdometryAckermann(Name,Value)` sets properties using one or more name-value pairs. Unspecified properties have default values. Enclose each property name in quotes.

For example, `whlEncOdom = wheelEncoderOdometryAckermann('SampleRate',100)` sets the sample rate of the sensor to 100 Hz.

Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see System Design in MATLAB Using System Objects.

`SampleRate` — Sample rate of sensor
`100` (default) | positive scalar

Sample rate of sensor, specified as a positive scalar in hertz.

Example: `'SampleRate',100`

Data Types: `double`

`TicksPerRevolution` — Number of encoder ticks per wheel revolution
`[2048 2048]` (default) | positive integer | two-element vector of positive integers

Number of encoder ticks per wheel revolution, specified as a positive integer or two-element vector of positive integers.

When specifying this value as a two-element vector, the first element corresponds to the back left wheel and the second to the back right wheel.

Example: `'TicksPerRevolution',[2048 2048]`

Data Types: `double`

`WheelRadius` — Wheel radius
`[0.35 0.35]` (default) | positive scalar | two-element vector of positive numbers

Wheel radius, specified as a positive scalar or two-element vector of positive numbers in meters.

When specifying this value as a two-element vector, the first element corresponds to the back left wheel and the second to the back right wheel.

Example: `'WheelRadius',[0.35 0.35]`

Data Types: `double`

`TrackWidth` — Distance between wheels on axle
`1.572` (default) | positive scalar

Distance between the wheels on the axle, specified as a positive scalar in meters.

Example: `'TrackWidth',1.572`

Data Types: `double`

`WheelBase` — Distance between front and rear axle
`2.818` (default) | positive scalar

Distance between the front and rear axle, specified as a positive scalar in meters.

Example: `'WheelBase',2.818`

Data Types: `double`

`InitialPose` — Initial pose of vehicle
`[0 0 0]` (default) | three-element vector

Initial pose of the vehicle, specified as three-element vector of the form [*X Y Yaw*]. *X* and *Y* specify the vehicle position in meters. *Yaw* specifies the vehicle orientation in radians. All values are in the local navigation coordinate system.

Example: `'InitialPose',[0 0 0]`

Data Types: `double`

Usage

Syntax

```
pose = whlEncOdom(ticks,steer)
[pose,velocity] = whlEncOdom(ticks,steer)
```

Description

`pose = whlEncOdom(ticks,steer)` computes the odometry of an Ackermann vehicle using the specified wheel encoder ticks `ticks` and steering angle `steer`, and returns the position and orientation of the vehicle in the local navigation coordinate system.

`[pose,velocity] = whlEncOdom(ticks,steer)` additionally returns the linear and angular velocity of the vehicle in the local navigation coordinate system.

Input Arguments

`ticks` — Number of wheel encoder ticks

n-by-2 matrix

Number of wheel encoder ticks, specified as an *n*-by-2 matrix. *n* is the number of samples in the current frame.

Each row of the matrix specifies wheel encoder ticks in the form [*ticksBackLeft ticksBackRight*], where *ticksBackLeft* and *ticksBackRight* specify the number of ticks for the back left and back right wheels, respectively.

Example: `[5 5; 2 2]`

Data Types: `single` | `double`

`steer` — Steering angle of vehicle

n-element column vector

Steering angle of the vehicle, specified as an *n*-element column vector in radians. *n* is the number of samples in the current frame.

Example: `[0.2; 0.2]`

Data Types: `single` | `double`

Output Arguments

`pose` — Position and orientation of vehicle

n-by-3 matrix

Position and orientation of the vehicle, returned as an *n*-by-3 matrix. *n* is the number of samples in the current frame. Each row of the matrix specifies the position and orientation of a sample in the form [*X Y Yaw*]. *X* and *Y* specify the vehicle position in meters. *Yaw* specifies the vehicle orientation in radians. All values are in the local navigation coordinate system.

Data Types: `single` | `double`

`velocity` — Linear and angular velocity of vehicle

n-by-3 matrix

Linear and angular velocity of the vehicle, returned as an *n*-by-3 matrix. *n* is the number of samples in the current frame. Each row of the matrix specifies the linear and angular velocity of a sample in the form [*velX velY yawRate*]. *velX* and *velY* specify the linear velocity of the vehicle in meters per second. *yawRate* specifies the angular velocity of the vehicle in radians per second. All values are in the local navigation coordinate system.

Data Types: `single` | `double`

Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

Common to All System Objects

clone	Create duplicate System object
step	Run System object algorithm
release	Release resources and allow changes to System object property values and input characteristics
reset	Reset internal states of System object
isLocked	Determine if System object is in use

Examples

Compute Ackermann Vehicle Odometry Using Wheel Encoder Ticks and Steering Angle

Create a `wheelEncoderOdometryAckermann` System object.

```
whlEncOdom = wheelEncoderOdometryAckermann;
```

Specify the number of wheel encoder ticks and the steering angle.

```
ticks = [5 5; 2 2];
steer = [0.2; 0.2];
```

Compute the Ackermann vehicle odometry.

```
[pose,vel] = whlEncOdom(ticks,steer)
```

```
pose = 2×3

    0.0053689   1.0368e-06   0.00038621
    0.0075165   2.0321e-06   0.00054069


vel = 2×3

      0.53689   0.00020735     0.038621
      0.21476   0.00011612     0.015448
```

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`wheelEncoderAckermann` | `wheelEncoderOdometryBicycle` | `wheelEncoderOdometryDifferentialDrive` | `wheelEncoderOdometryUnicycle`

Introduced in R2020b

wheelEncoderOdometryBicycle

Compute bicycle odometry using wheel encoder ticks and steering angle

Description

The `wheelEncoderOdometryBicycle` System object computes bicycle odometry using the wheel encoder ticks and steering angle of the vehicle.

To compute bicycle odometry:

1. Create the `wheelEncoderOdometryBicycle` object and set its properties.
2. Call the object with arguments, as if it were a function.

To learn more about how System objects work, see What Are System Objects?

Creation

Syntax

```
whlEncOdom = wheelEncoderOdometryBicycle
whlEncOdom = wheelEncoderOdometryBicycle(encoder)
whlEncOdom = wheelEncoderOdometryBicycle(Name,Value)
```

Description

`whlEncOdom = wheelEncoderOdometryBicycle` creates a `wheelEncoderOdometryBicycle` System object with default property values.

`whlEncOdom = wheelEncoderOdometryBicycle(encoder)` creates a `wheelEncoderOdometryBicycle` System object using the specified `wheelEncoderBicycle` System object, `encoder`, to set properties.

`whlEncOdom = wheelEncoderOdometryBicycle(Name,Value)` sets properties using one or more name-value pairs. Unspecified properties have default values. Enclose each property name in quotes.

For example, `whlEncOdom = wheelEncoderOdometryBicycle('SampleRate',100)` sets the sample rate of the sensor to 100 Hz.

Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see System Design in MATLAB Using System Objects.

`SampleRate` — Sample rate of sensor
`100` (default) | positive scalar

Sample rate of sensor, specified as a positive scalar in hertz.

Example: `'SampleRate',100`

Data Types: `double`

`TicksPerRevolution` — Number of encoder ticks per wheel revolution
`2048` (default) | positive integer

Number of encoder ticks per wheel revolution, specified as a positive integer. This value corresponds to the rear wheel of the bicycle.

Example: `'TicksPerRevolution',2048`

Data Types: `double`

`WheelRadius` — Wheel radius
`0.35` (default) | positive scalar

Wheel radius, specified as a positive scalar in meters.

Example: `'WheelRadius',0.35`

Data Types: `double`

`WheelBase` — Distance between front and rear axle
`2.818` (default) | positive scalar

Distance between front and rear axle, specified as a positive scalar in meters.

Example: `'WheelBase',2.818`

Data Types: `double`

`InitialPose` — Initial pose of vehicle
`[0 0 0]` (default) | three-element vector

Initial pose of the vehicle, specified as three-element vector of the form [*X Y Yaw*]. *X* and *Y* specify the vehicle position in meters. *Yaw* specifies the vehicle orientation in radians. All values are in the local navigation coordinate system.

Example: `'InitialPose',[0 0 0]`

Tunable: No

Data Types: `double`

Usage

Syntax

```
pose = whlEncOdom(ticks,steer)
[pose,velocity] = whlEncOdom(ticks,steer)
```

Description

`pose = whlEncOdom(ticks,steer)` computes the odometry of a bicycle using the specified wheel encoder ticks `ticks` and steering angle `steer`, and returns the position and orientation of the vehicle in the local navigation coordinate system.

`[pose,velocity] = whlEncOdom(ticks,steer)` additionally returns the linear and angular velocity of the vehicle in the local navigation coordinate system.

Input Arguments

`ticks` — Number of wheel encoder ticks

n-element column vector

Number of wheel encoder ticks, specified as an *n*-element column vector. *n* is the number of samples in the current frame. Each element is the number of ticks for the rear wheel of the bicycle in the corresponding sample.

Example: `[5; 2]`

Data Types: `single` | `double`

`steer` — Steering angle of vehicle

n-element column vector

Steering angle of the vehicle, specified as an *n*-element column vector in radians. *n* is the number of samples in the current frame.

Example: `[0.2; 0.2]`

Data Types: `single` | `double`

Output Arguments

`pose` — Position and orientation of vehicle

n-by-3 matrix

Position and orientation of the vehicle, returned as an *n*-by-3 matrix. *n* is the number of samples in the current frame. Each row of the matrix specifies the position and orientation of a sample in the form [*X Y Yaw*]. *X* and *Y* specify the vehicle position in meters. *Yaw* specifies the vehicle orientation in radians. All values are in the local navigation coordinate system.

Data Types: `single` | `double`

`velocity` — Linear and angular velocity of vehicle

n-by-3 matrix

Linear and angular velocity of the vehicle, returned as an *n*-by-3 matrix. *n* is the number of samples in the current frame. Each row of the matrix specifies the linear and angular velocity of a sample in the form [*velX velY yawRate*]. *velX* and *velY* specify the linear velocity of the vehicle in meters per second. *yawRate* specifies the angular velocity of the vehicle in radians per second. All values are in the local navigation coordinate system.

Data Types: `single` | `double`

Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

Common to All System Objects

clone	Create duplicate System object
step	Run System object algorithm
release	Release resources and allow changes to System object property values and input characteristics
reset	Reset internal states of System object
isLocked	Determine if System object is in use

Examples

Compute Bicycle Odometry Using Wheel Encoder Ticks and Steering Angle

Create a `wheelEncoderOdometryBicycle` System object.

```
whlEncOdom = wheelEncoderOdometryBicycle;
```

Specify the number of wheel encoder ticks and the steering angle.

```
ticks = [5; 2];
steer = [0.2; 0.2];
```

Compute the bicycle odometry.

```
[pose,vel] = whlEncOdom(ticks,steer)
```

```
pose = 2×3

    0.0054    0.0000    0.0004
    0.0075    0.0000    0.0005


vel = 2×3

    0.5369    0.0002    0.0386
    0.2148    0.0001    0.0154
```

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`wheelEncoderBicycle` | `wheelEncoderOdometryAckermann` | `wheelEncoderOdometryDifferentialDrive` | `wheelEncoderOdometryUnicycle`

Introduced in R2020b

wheelEncoderOdometryDifferentialDrive

Compute differential-drive vehicle odometry using wheel encoder ticks

Description

The `wheelEncoderOdometryDifferentialDrive` System object computes differential-drive vehicle odometry using the wheel encoder ticks.

To compute differential-drive vehicle odometry:

1. Create the `wheelEncoderOdometryDifferentialDrive` object and set its properties.
2. Call the object with arguments, as if it were a function.

To learn more about how System objects work, see What Are System Objects?

Creation

Syntax

```
whlEncOdom = wheelEncoderOdometryDifferentialDrive
whlEncOdom = wheelEncoderOdometryDifferentialDrive(encoder)
whlEncOdom = wheelEncoderOdometryDifferentialDrive(Name,Value)
```

Description

`whlEncOdom = wheelEncoderOdometryDifferentialDrive` creates a `wheelEncoderOdometryDifferentialDrive` System object with default property values.

`whlEncOdom = wheelEncoderOdometryDifferentialDrive(encoder)` creates a `wheelEncoderOdometryDifferentialDrive` System object using the specified `wheelEncoderDifferentialDrive` System object, `encoder`, to set properties.

`whlEncOdom = wheelEncoderOdometryDifferentialDrive(Name,Value)` sets properties using one or more name-value pairs. Unspecified properties have default values. Enclose each property name in quotes.

For example, `whlEncOdom = wheelEncoderOdometryDifferentialDrive('SampleRate',100)` sets the sample rate of the sensor to 100 Hz.

Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see System Design in MATLAB Using System Objects.

`SampleRate` — Sample rate of sensor
`100` (default) | positive scalar

Sample rate of sensor, specified as a positive scalar in hertz.

Example: `'SampleRate',100`

Data Types: `double`

`TicksPerRevolution` — Number of encoder ticks per wheel revolution
`[2048 2048]` (default) | positive integer | two-element vector of positive integers

Number of encoder ticks per wheel revolution, specified as a positive integer or two-element vector of positive integers.

When specifying this value as a two-element vector, the first element corresponds to the left wheel and the second to the right wheel.

Example: `'TicksPerRevolution',[2048 2048]`

Data Types: `double`

`WheelRadius` — Wheel radius
`[0.35 0.35]` (default) | positive scalar | two-element vector of positive numbers

Wheel radius, specified as a positive scalar or two-element vector of positive numbers in meters.

When specifying this value as a two-element vector, the first element corresponds to the left wheel and the second to the right wheel.

Example: `'WheelRadius',[0.35 0.35]`

Data Types: `double`

`TrackWidth` — Distance between wheels on axle
`1.572` (default) | positive scalar

Distance between the wheels on the axle, specified as a positive scalar in meters.

Example: `'TrackWidth',1.572`

Data Types: `double`

`InitialPose` — Initial pose of vehicle
`[0 0 0]` (default) | three-element vector

Initial pose of the vehicle, specified as three-element vector of the form [*X Y Yaw*]. *X* and *Y* specify the vehicle position in meters. *Yaw* specifies the vehicle orientation in radians. All values are in the local navigation coordinate system.

Example: `'InitialPose',[0 0 0]`

Data Types: `double`

Usage

Syntax

```
pose = whlEncOdom(ticks)
[pose,velocity] = whlEncOdom(ticks)
```

Description

`pose = whlEncOdom(ticks)` computes the odometry of a differential-drive vehicle using the specified wheel encoder ticks `ticks`, and returns the position and orientation of the vehicle in the local navigation coordinate system.

`[pose,velocity] = whlEncOdom(ticks)` additionally returns the linear and angular velocity of the vehicle in the local navigation coordinate system.

Input Arguments

`ticks` — Number of wheel encoder ticks

n-by-2 matrix

Number of wheel encoder ticks, specified as an *n*-by-2 matrix. *n* is the number of samples in the current frame.

Each row of the matrix specifies wheel encoder ticks in the form [*ticksLeft ticksRight*], where *ticksLeft* and *ticksRight* specify the number of ticks for the left and right wheels, respectively.

Example: `[5 5; 2 2]`

Data Types: `single` | `double`

Output Arguments

`pose` — Position and orientation of vehicle

n-by-3 matrix

Position and orientation of the vehicle, returned as an *n*-by-3 matrix. *n* is the number of samples in the current frame. Each row of the matrix specifies the position and orientation of a sample in the form [*X Y Yaw*]. *X* and *Y* specify the vehicle position in meters. *Yaw* specifies the vehicle orientation in radians. All values are in the local navigation coordinate system.

Data Types: `single` | `double`

`velocity` — Linear and angular velocity of vehicle

n-by-3 matrix

Linear and angular velocity of the vehicle, returned as an *n*-by-3 matrix. *n* is the number of samples in the current frame. Each row of the matrix specifies the linear and angular velocity of a sample in the form [*velX velY yawRate*]. *velX* and *velY* specify the linear velocity of the vehicle in meters per second. *yawRate* specifies the angular velocity of the vehicle in radians per second. All values are in the local navigation coordinate system.

Data Types: `single` | `double`

Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

Common to All System Objects

clone	Create duplicate System object
step	Run System object algorithm
release	Release resources and allow changes to System object property values and input characteristics
reset	Reset internal states of System object
isLocked	Determine if System object is in use

Examples

Compute Differential-Drive Vehicle Odometry Using Wheel Encoder Ticks

Create a `wheelEncoderOdometryDifferentialDrive` System object.

```
whlEncOdom = wheelEncoderOdometryDifferentialDrive;
```

Specify the number of wheel encoder ticks.

```
ticks = [5 5; 2 2];
```

Compute the differential-drive vehicle odometry.

```
[pose,vel] = whlEncOdom(ticks)
```

```
pose = 2×3

    0.0054         0         0
    0.0075         0         0


vel = 2×3

    0.5369         0         0
    0.2148         0         0
```

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`wheelEncoderDifferentialDrive` | `wheelEncoderOdometryAckermann` | `wheelEncoderOdometryBicycle` | `wheelEncoderOdometryUnicycle`

Introduced in R2020b

wheelEncoderOdometryUnicycle

Compute unicycle odometry using wheel encoder ticks and angular velocity

Description

The `wheelEncoderOdometryUnicycle` System object computes unicycle odometry using the wheel encoder ticks and angular velocity.

To compute unicycle odometry:

1. Create the `wheelEncoderOdometryUnicycle` object and set its properties.
2. Call the object with arguments, as if it were a function.

To learn more about how System objects work, see What Are System Objects?

Creation

Syntax

```
whlEncOdom = wheelEncoderOdometryUnicycle
whlEncOdom = wheelEncoderOdometryUnicycle(encoder)
whlEncOdom = wheelEncoderOdometryUnicycle(Name,Value)
```

Description

`whlEncOdom = wheelEncoderOdometryUnicycle` creates a `wheelEncoderOdometryUnicycle` System object with default property values.

`whlEncOdom = wheelEncoderOdometryUnicycle(encoder)` creates a `wheelEncoderOdometryUnicycle` System object using the specified `wheelEncoderUnicycle` System object, `encoder`, to set properties.

`whlEncOdom = wheelEncoderOdometryUnicycle(Name,Value)` sets properties using one or more name-value pairs. Unspecified properties have default values. Enclose each property name in quotes.

For example, `whlEncOdom = wheelEncoderOdometryUnicycle('SampleRate',100)` sets the sample rate of the sensor to 100 Hz.

Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see System Design in MATLAB Using System Objects.

`SampleRate` — Sample rate of sensor
`100` (default) | positive scalar

Sample rate of sensor, specified as a positive scalar in hertz.

Example: `'SampleRate',100`

Data Types: `double`

`TicksPerRevolution` — Number of encoder ticks per wheel revolution
`2048` (default) | positive integer

Number of encoder ticks per wheel revolution, specified as a positive integer.

Example: `'TicksPerRevolution',2048`

Data Types: `double`

`WheelRadius` — Wheel radius
`0.35` (default) | positive scalar

Wheel radius, specified as a positive scalar in meters.

Example: `'WheelRadius',0.35`

Data Types: `double`

`InitialPose` — Initial pose of vehicle
`[0 0 0]` (default) | three-element vector

Initial pose of the vehicle, specified as three-element vector of the form [*X Y Yaw*]. *X* and *Y* specify the vehicle position in meters. *Yaw* specifies the vehicle orientation in radians. All values are in the local navigation coordinate system.

Example: `'InitialPose',[0 0 0]`

Tunable: No

Data Types: `double`

Usage

Syntax

```
pose = whlEncOdom(ticks,angVel)
[pose,velocity] = whlEncOdom(ticks,angVel)
```

Description

`pose = whlEncOdom(ticks,angVel)` computes the odometry of a unicycle using the specified wheel encoder ticks `ticks` and angular velocity `angVel`, and returns the position and orientation of the vehicle in the local navigation coordinate system.

`[pose,velocity] = whlEncOdom(ticks,angVel)` additionally returns the linear and angular velocity of the vehicle in the local navigation coordinate system.

Input Arguments

`ticks` — Number of wheel encoder ticks

n-element column vector

Number of wheel encoder ticks, specified as an *n*-element column vector. *n* is the number of samples in the current frame.

Example: `[5; 2]`

Data Types: `single` | `double`

`angVel` — Angular velocity of vehicle in vehicle body coordinate system

n-element column vector

Angular velocity of the vehicle in the vehicle body coordinate system, specified as an *n*-element column vector in radians per second. *n* is the number of samples in the current frame.

Example: `[0.2; 0.2]`

Data Types: `single` | `double`

Output Arguments

`pose` — Position and orientation of vehicle

n-by-3 matrix

Position and orientation of the vehicle, returned as an *n*-by-3 matrix. *n* is the number of samples in the current frame. Each row of the matrix specifies the position and orientation of a sample in the form [*X Y Yaw*]. *X* and *Y* specify the vehicle position in meters. *Yaw* specifies the vehicle orientation in radians. All values are in the local navigation coordinate system.

Data Types: `single` | `double`

`velocity` — Linear and angular velocity of vehicle

n-by-3 matrix

Linear and angular velocity of the vehicle, returned as an *n*-by-3 matrix. *n* is the number of samples in the current frame. Each row of the matrix specifies the linear and angular velocity of a sample in the form [*velX velY yawRate*]. *velX* and *velY* specify the linear velocity of the vehicle in meters per second. *yawRate* specifies the angular velocity of the vehicle in radians per second. All values are in the local navigation coordinate system.

Data Types: `single` | `double`

Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

Common to All System Objects

clone	Create duplicate System object
step	Run System object algorithm

release	Release resources and allow changes to System object property values and input characteristics
reset	Reset internal states of System object
isLocked	Determine if System object is in use

Examples

Compute Unicycle Odometry Using Wheel Encoder Ticks and Angular Velocity

Create a `wheelEncoderOdometryUnicycle` System object.

```
whlEncOdom = wheelEncoderOdometryUnicycle;
```

Specify the number of wheel encoder ticks and angular velocity.

```
ticks = [5; 2];
angVel = [0.2; 0.2];
```

Compute the unicycle odometry.

```
[pose,vel] = whlEncOdom(ticks,angVel)
```

```
pose = 2×3

    0.0054    0.0000    0.0020
    0.0075    0.0000    0.0040


vel = 2×3

    0.5369    0.0011    0.2000
    0.2148    0.0009    0.2000
```

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`wheelEncoderUnicycle` | `wheelEncoderOdometryAckermann` | `wheelEncoderOdometryBicycle` | `wheelEncoderOdometryDifferentialDrive`

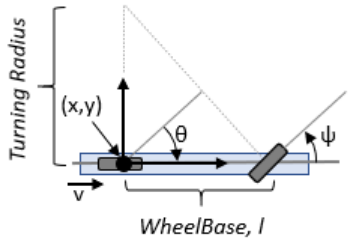
Introduced in R2020b

bicycleKinematics

Bicycle vehicle model

Description

`bicycleKinematics` creates a bicycle vehicle model to simulate simplified car-like vehicle dynamics. This model represents a vehicle with two axles separated by a distance, `WheelBase`. The state of the vehicle is defined as a three-element vector, [*x y theta*], with a global *xy*-position, specified in meters, and a vehicle heading angle, *theta*, specified in radians. The front wheel can be turned with steering angle *psi*. The vehicle heading, *theta*, is defined at the center of the rear axle. To compute the time derivative states of the model, use the `derivative` function with input commands and the current robot state.



Creation

Syntax

```
kinematicModel = bicycleKinematics
```

```
kinematicModel = bicycleKinematics(Name,Value)
```

Description

`kinematicModel = bicycleKinematics` creates a bicycle kinematic model object with default property values.

`kinematicModel = bicycleKinematics(Name,Value)` sets additional properties to the specified values. You can specify multiple properties in any order.

Properties

`WheelBase` — Distance between front and rear axles
1 (default) | positive numeric scalar

The wheel base refers to the distance between the front and rear vehicle axles, specified in meters.

`VehicleSpeedRange` — Range of vehicle speeds
`[-Inf Inf]` (default) | positive numeric scalar

The vehicle speed range is a two-element vector that provides the minimum and maximum vehicle speeds, [*MinSpeed MaxSpeed*], specified in meters per second.

`MaxSteeringAngle` — Maximum steering angle
`pi/4` (default) | numeric scalar

The maximum steering angle, *psi*, refers to the maximum angle the vehicle can be steered to the right or left, specified in radians. A value of `pi/2` provides the vehicle with a minimum turning radius of 0. This property is used to validate the user-provided state input.

`MinimumTurningRadius` — Minimum vehicle turning radius
`1.0000` (default) | numeric scalar

This read-only property returns the minimum vehicle turning radius in meters. The minimum radius is computed using the wheel base and the maximum steering angle.

`VehicleInputs` — Type of motion inputs for vehicle
`"VehicleSpeedSteeringAngle"` (default) | character vector | string scalar

The `VehicleInputs` property specifies the format of the model input commands when using the `derivative` function. The property has two valid options, specified as a string or character vector:

- `"VehicleSpeedSteeringAngle"` — Vehicle speed and steering angle
- `"VehicleSpeedHeadingRate"` — Vehicle speed and heading angular velocity

Object Functions

derivative Time derivative of bicycle vehicle model

Examples

Plot Path of Bicycle Kinematic Robot

Create a robot and set its initial starting position and orientation.

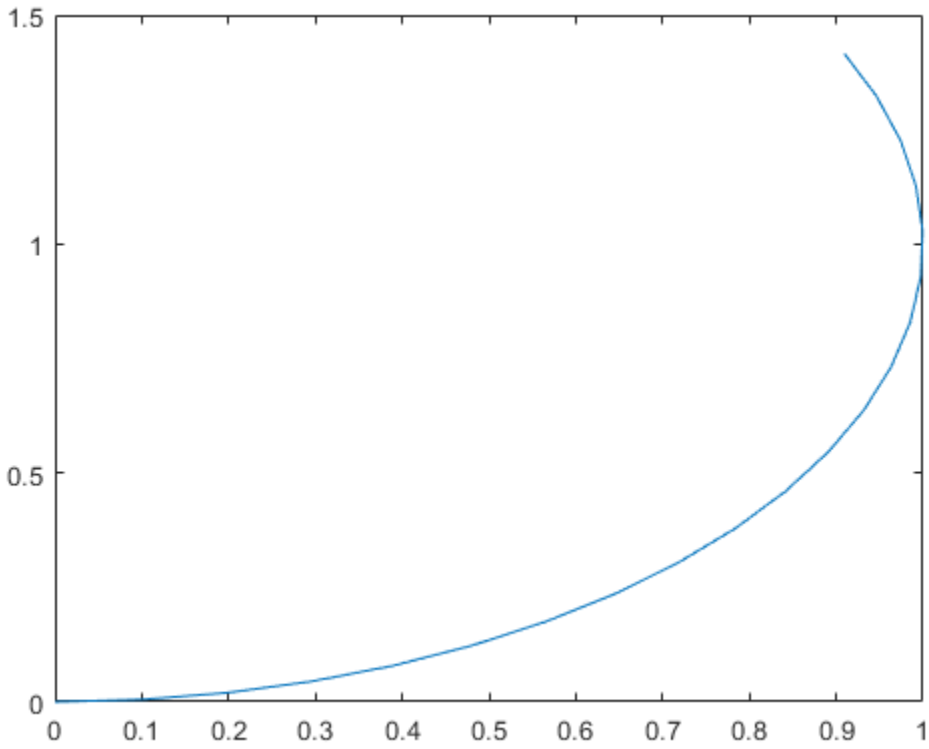
```
kinematicModel = bicycleKinematics;
initialState = [0 0 0];
```

Set the timespan of the simulation to 1 s with 0.05 s timesteps and the input commands to 2 m/s and left turn. Simulate the motion of the robot by using the `ode45` solver on the `derivative` function.

```
tspan = 0:0.05:1;
inputs = [2 pi/4]; %Turn left
[t,y] = ode45(@(t,y)derivative(kinematicModel,y,inputs),tspan,initialState);
```

Plot the path.

```
figure
plot(y(:,1),y(:,2))
```



References

[1] Lynch, Kevin M., and Frank C. Park. *Modern Robotics: Mechanics, Planning, and Control* 1st ed. Cambridge, MA: Cambridge University Press, 2017.

[2] Corke, Peter I. *Robotics, Vision and Control: Fundamental Algorithms in MATLAB*. Springer, 2011.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

Topics

“Mobile Robot Kinematics Equations” (Robotics System Toolbox)

Introduced in R2021b

derivative

Time derivative of bicycle vehicle model

Syntax

```
stateDot = derivative(kinematicModel,state,cmds)
```

Description

`stateDot = derivative(kinematicModel,state,cmds)` returns the current state derivative, `stateDot`, as a three-element vector [*xDot yDot thetaDot*] for a bicycle kinematics vehicle motion model, `kinematicModel`. *xDot* and *yDot* refer to the vehicle velocity, specified in meters per second. *thetaDot* is the angular velocity of the vehicle heading, specified in radians per second.

Examples

Plot Path of Bicycle Kinematic Robot

Create a robot and set its initial starting position and orientation.

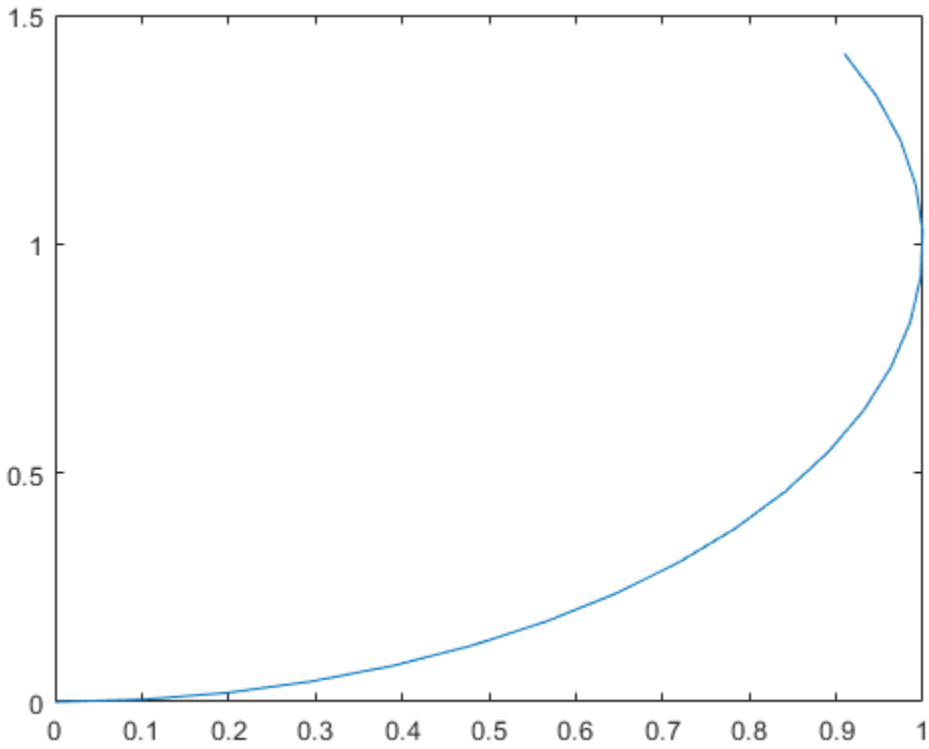
```
kinematicModel = bicycleKinematics;
initialState = [0 0 0];
```

Set the timespan of the simulation to 1 s with 0.05 s timesteps and the input commands to 2 m/s and left turn. Simulate the motion of the robot by using the `ode45` solver on the `derivative` function.

```
tspan = 0:0.05:1;
inputs = [2 pi/4]; %Turn left
[t,y] = ode45(@(t,y)derivative(kinematicModel,y,inputs),tspan,initialState);
```

Plot the path.

```
figure
plot(y(:,1),y(:,2))
```



Input Arguments

`kinematicModel` — Bicycle kinematic motion model

`bicycleKinematics` object

Bicycle kinematic motion model, specified as a `bicycleKinematics` object.

`state` — Current vehicle state

three-element vector | four-element vector

Current vehicle state returned as a three-element vector of the form `[x y theta]`.

x and *y* refer to the vehicle position, specified in meters per second. *theta* is the vehicle heading, specified in radians per second.

`cmds` — Input commands to motion model

two-element vector

Input commands to the motion model, specified as a two-element vector. The `VehicleInputs` property value of `motionModel` determines the format of this command vector. These are the valid `VehicleInputs` values for a `bicycleKinematics` object:

- `"VehicleSpeedSteeringAngle"` -- `[v psiDot]`
- `"VehicleSpeedHeadingRate"` -- `[v omegaDot]`

v is the vehicle velocity in the direction of motion in meters per second. *psiDot* is the steering angle rate in radians per second. *omegaDot* is the angular velocity at the rear axle.

Output Arguments

`stateDot` — Derivative of current state

three-element vector of form `[xDot yDot thetaDot]`

Derivative of current state, returned as a three-element vector of the form `[xDot yDot thetaDot]`. *xDot* and *yDot* refer to the vehicle velocity, returned in meters per second. *thetaDot* is the angular velocity of the vehicle heading, returned in radians per second.

References

[1] Lynch, Kevin M., and Frank C. Park. *Modern Robotics: Mechanics, Planning, and Control*. 1st ed. Cambridge, MA: Cambridge University Press, 2017.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`bicycleKinematics`

Introduced in R2021b

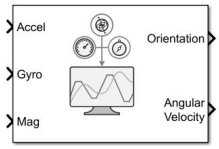
Methods

Blocks

AHRS

Orientation from accelerometer, gyroscope, and magnetometer readings

Library: Navigation Toolbox / Multisensor Positioning / Navigation Filters
Sensor Fusion and Tracking Toolbox / Multisensor Positioning / Navigation Filters



Description

The AHRS Simulink® block fuses accelerometer, magnetometer, and gyroscope sensor data to estimate device orientation.

Ports

Input

`Accel` — Accelerometer readings in sensor body coordinate system (m/s²)

N-by-3 matrix of real scalar

Accelerometer readings in the sensor body coordinate system in m/s^2, specified as an *N*-by-3 matrix of real scalars. *N* is the number of samples, and the three columns of `Accel` represent the [*x y z*] measurements, respectively.

Data Types: `single` | `double`

`Gyro` — Gyroscope readings in sensor body coordinate system (rad/s)

N-by-3 matrix of real scalar

Gyroscope readings in the sensor body coordinate system in rad/s, specified as an *N*-by-3 matrix of real scalars. *N* is the number of samples, and the three columns of `Gyro` represent the [*x y z*] measurements, respectively.

Data Types: `single` | `double`

`Mag` — Magnetometer readings in sensor body coordinate system (µT)

N-by-3 matrix of real scalar

Magnetometer readings in the sensor body coordinate system in µT, specified as an *N*-by-3 matrix of real scalars. *N* is the number of samples, and the three columns of `magReadings` represent the [*x y z*] measurements, respectively.

Data Types: `single` | `double`

Output

`Orientation` — Orientation of sensor body frame relative to navigation frame

M-by-4 array of scalar | 3-by-3-by-*M*-element rotation matrix

Orientation of the sensor body frame relative to the navigation frame, return as an *M*-by-4 array of scalars or a 3-by-3-by-*M* array of rotation matrices. Each row the of the *N*-by-4 array is assumed to be the four elements of a `quaternion`. The number of input samples, *N*, and the **Decimation Factor** parameter determine the output size *M*.

Data Types: `single` | `double`

`Angular Velocity` — Angular velocity in sensor body coordinate system (rad/s)
M-by-3 array of real scalar (default)

Angular velocity with gyroscope bias removed in the sensor body coordinate system in rad/s, returned as an *M*-by-3 array of real scalars. The number of input samples, *N*, and the **Decimation Factor** parameter determine the output size *M*.

Data Types: `single` | `double`

Parameters

Main

`Reference frame` — Navigation reference frame
`NED` (default) | `ENU`

Navigation reference frame, specified as `NED` (North-East-Down) or `ENU` (East-North-Up).

`Decimation factor` — Decimation factor
1 (default) | positive integer

Decimation factor by which to reduce the input sensor data rate, specified as a positive integer.

The number of rows of the inputs –– **Accel**, **Gyro** , and **Mag** –– must be a multiple of the decimation factor.

Data Types: `single` | `double`

`Initial process noise` — Initial process noise
`ahrsfilter.defaultProcessNoise` (default) | 12-by-12 matrix of real scalar

Initial process noise, specified as a 12-by-12 matrix of real scalars. The default value, `ahrsfilter.defaultProcessNoise`, is a 12-by-12 diagonal matrix as:

```
  Columns 1 through 6

   0.000006092348396                   0                   0                   0
                   0   0.000006092348396                   0                   0
                   0                   0   0.000006092348396                   0
                   0                   0                   0   0.000076154354947
                   0                   0                   0                   0   0.00007615435
                   0                   0                   0                   0
                   0                   0                   0                   0
                   0                   0                   0                   0
                   0                   0                   0                   0
                   0                   0                   0                   0
                   0                   0                   0                   0
                   0                   0                   0                   0

  Columns 7 through 12
```

```
                    0                    0                    0                    0
                    0                    0                    0                    0
                    0                    0                    0                    0
                    0                    0                    0                    0
                    0                    0                    0                    0
                    0                    0                    0                    0
   0.009623610000000                    0                    0                    0
                    0   0.009623610000000                    0                    0
                    0                    0   0.009623610000000                    0
                    0                    0                    0   0.600000000000000
                    0                    0                    0                    0   0.600000000000000
                    0                    0                    0                    0
```

Data Types: `single` | `double`

`Orientation format` — Orientation output format
`'quaternion'` (default) | `'Rotation matrix'`

Output orientation format, specified as `'quaternion'` or `'Rotation matrix'`:

- `'quaternion'` –– Output is an *M*-by-4 array of real scalars. Each row of the array represents the four components of a `quaternion`.
- `'Rotation matrix'` –– Output is a 3-by-3-by-*M* rotation matrix.

The output size *M* depends on the input dimension *N* and the **Decimation Factor** parameter.

Data Types: `char` | `string`

`Simulate using` — Type of simulation to run
`Interpreted Execution` (default) | `Code Generation`

- `Interpreted execution` — Simulate the model using the MATLAB interpreter. This option shortens startup time. In `Interpreted execution` mode, you can debug the source code of the block.
- `Code generation` — Simulate the model using generated C code. The first time you run a simulation, Simulink generates C code for the block. The C code is reused for subsequent simulations as long as the model does not change. This option requires additional startup time.

Measurement Noise

`Accelerometer noise ((m/s²)²)` — Variance of accelerometer signal noise ((m/s²)²)
`0.00019247` (default) | positive real scalar

Variance of accelerometer signal noise in $(m/s^2)^2$, specified as a positive real scalar.

Data Types: `single` | `double`

`Gyroscope noise ((rad/s)²)` — Variance of gyroscope signal noise ((rad/s)²)
`9.1385e-5` (default) | positive real scalar

Variance of gyroscope signal noise in $(rad/s)^2$, specified as a positive real scalar.

Data Types: `single` | `double`

`Magnetometer noise (μT²)` — Variance of magnetometer signal noise (μT²)
`0.1` (default) | positive real scalar

Variance of magnetometer signal noise in μT^2, specified as a positive real scalar.

Data Types: `single` | `double`

`Gyroscope drift noise (rad/s)` — Variance of gyroscope offset drift ((rad/s)²)
`3.0462e-13` (default) | positive real scalar

Variance of gyroscope offset drift in $(rad/s)^2$, specified as a positive real scalar.

Data Types: `single` | `double`

Environmental Noise

`Linear acceleration noise ((m/s²)²)` — Variance of linear acceleration noise (m/s²)²
`0.0096236` (default) | positive real scalar

Variance of linear acceleration noise in $(m/s^2)^2$, specified as a positive real scalar. Linear acceleration is modeled as a lowpass-filtered white noise process.

Data Types: `single` | `double`

`Magnetic disturbance noise (μT²)` — Variance of magnetic disturbance noise (μT²)
`0.5` (default) | real finite positive scalar

Variance of magnetic disturbance noise in μT^2, specified as a real finite positive scalar.

Data Types: `single` | `double`

`Linear acceleration decay factor` — Decay factor for linear acceleration drift
`0.5` (default) | scalar in the range [0,1)

Decay factor for linear acceleration drift, specified as a scalar in the range [0,1). If linear acceleration changes quickly, set this parameter to a lower value. If linear acceleration changes slowly, set this parameter to a higher value. Linear acceleration drift is modeled as a lowpass-filtered white noise process.

Data Types: `single` | `double`

`Magnetic disturbance decay factor` — Decay factor for magnetic disturbance
`0.5` (default) | positive scalar in the range [0,1]

Decay factor for magnetic disturbance, specified as a positive scalar in the range [0,1]. Magnetic disturbance is modeled as a first order Markov process.

Data Types: `single` | `double`

`Magnetic field strength (μT)` — Magnetic field strength (μT)
`50` (default) | real positive scalar

Magnetic field strength in μT, specified as a real positive scalar. The magnetic field strength is an estimate of the magnetic field strength of the Earth at the current location.

Data Types: `single` | `double`

Algorithms

Note: The following algorithm only applies to an NED reference frame.

The AHRS block uses the nine-axis Kalman filter structure described in [1]. The algorithm attempts to track the errors in orientation, gyroscope offset, linear acceleration, and magnetic disturbance to output the final orientation and angular velocity. Instead of tracking the orientation directly, the indirect Kalman filter models the error process, x, with a recursive update:

$$x_k = \begin{bmatrix} \theta_k \\ b_k \\ a_k \\ d_k \end{bmatrix} = F_k \begin{bmatrix} \theta_{k-1} \\ b_{k-1} \\ a_{k-1} \\ d_{k-1} \end{bmatrix} + w_k$$

where x_k is a 12-by-1 vector consisting of:

- θ_k –– 3-by-1 orientation error vector, in degrees, at time k
- b_k –– 3-by-1 gyroscope zero angular rate bias vector, in deg/s, at time k
- a_k –– 3-by-1 acceleration error vector measured in the sensor frame, in g, at time k
- d_k –– 3-by-1 magnetic disturbance error vector measured in the sensor frame, in µT, at time k

and where w_k is a 12-by-1 additive noise vector, and F_k is the state transition model.

Because x_k is defined as the error process, the *a priori* estimate is always zero, and therefore the state transition model, F_k, is zero. This insight results in the following reduction of the standard Kalman equations:

Standard Kalman equations:

$$x_k^- = F_k x_{k-1}^+$$
$$P_k^- = F_k P_{k-1}^+ F_k^T + Q_k$$

$$y_k = z_k - H_k x_k^-$$
$$S_k = R_k + H_k P_k^- H_k{}^T$$
$$K_k = P_k^- H_k^T (S_k)^{-1}$$
$$x_k^+ = x_k^- + K_k y_k$$
$$P_k^+ = P_k{}^- - K_k H_k P_k^-$$

Kalman equations used in this algorithm:

$$x_k^- = 0$$
$$P_k^- = Q_k$$

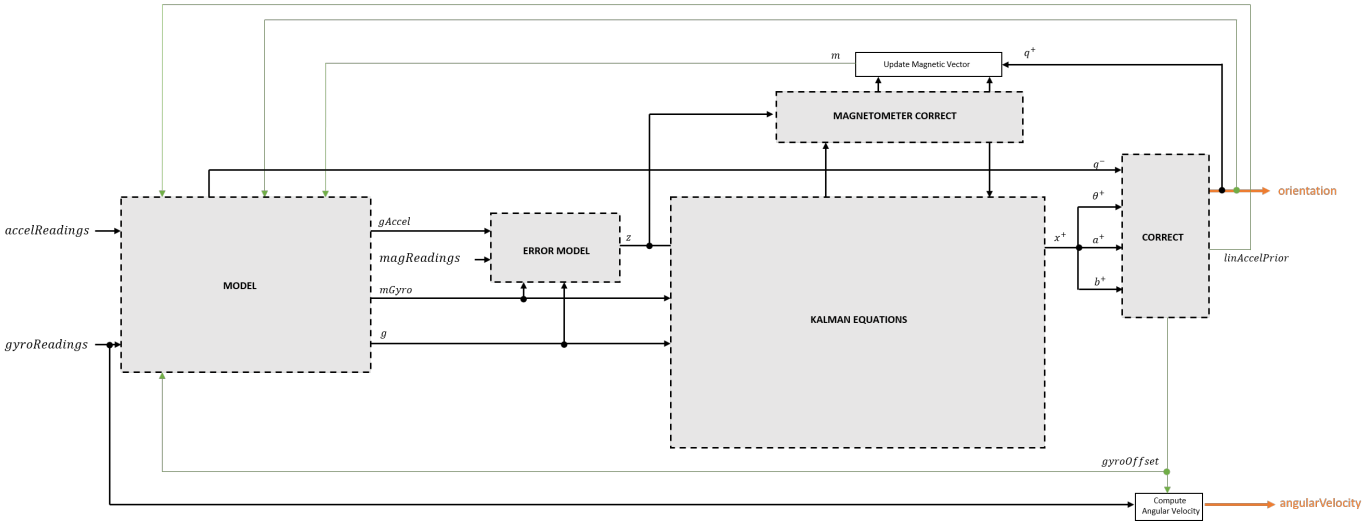
$$y_k = z_k$$
$$S_k = R_k + H_k P_k^- H_k{}^T$$
$$K_k = P_k^- H_k^T (S_k)^{-1}$$
$$x_k^+ = K_k y_k$$
$$P_k^+ = P_k{}^- - K_k H_k P_k^-$$

where:

- x_k^- –– predicted (*a priori*) state estimate; the error process
- P_k^- –– predicted (*a priori*) estimate covariance
- y_k –– innovation
- S_k –– innovation covariance
- K_k –– Kalman gain
- x_k^+ –– updated (*a posteriori*) state estimate
- P_k^+ –– updated (*a posteriori*) estimate covariance

k represents the iteration, the superscript $^+$ represents an *a posteriori* estimate, and the superscript $^-$ represents an *a priori* estimate.

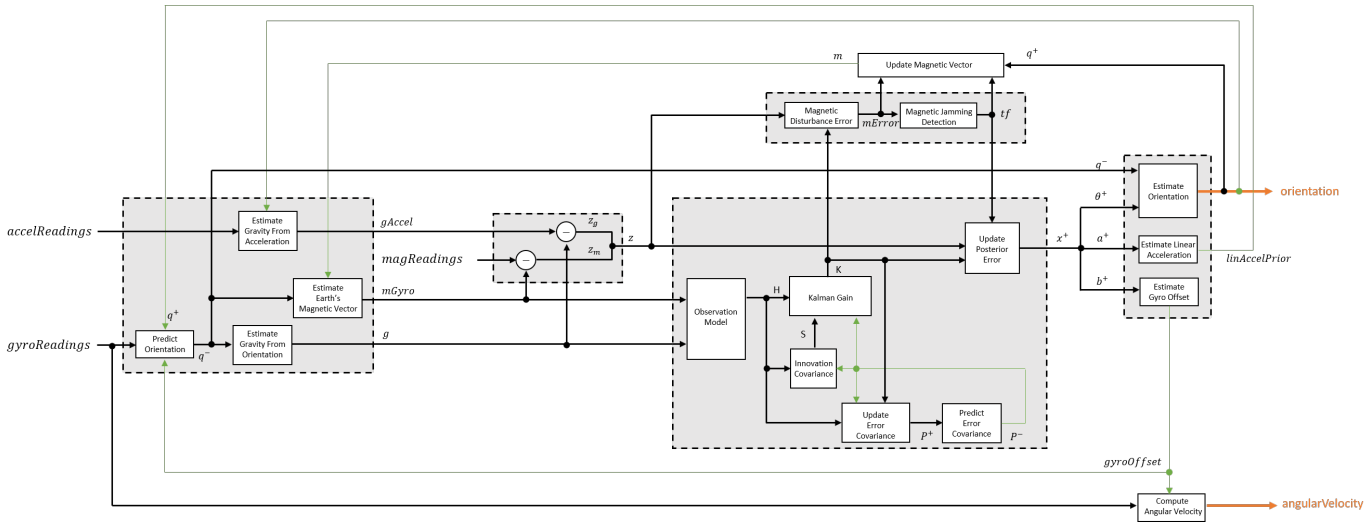
The graphic and following steps describe a single frame-based iteration through the algorithm.



Before the first iteration, the `accelReadings`, `gyroReadings`, and `magReadings` inputs are chunked into `DecimationFactor`-by-3 frames. For each chunk, the algorithm uses the most current accelerometer and magnetometer readings corresponding to the chunk of gyroscope readings.

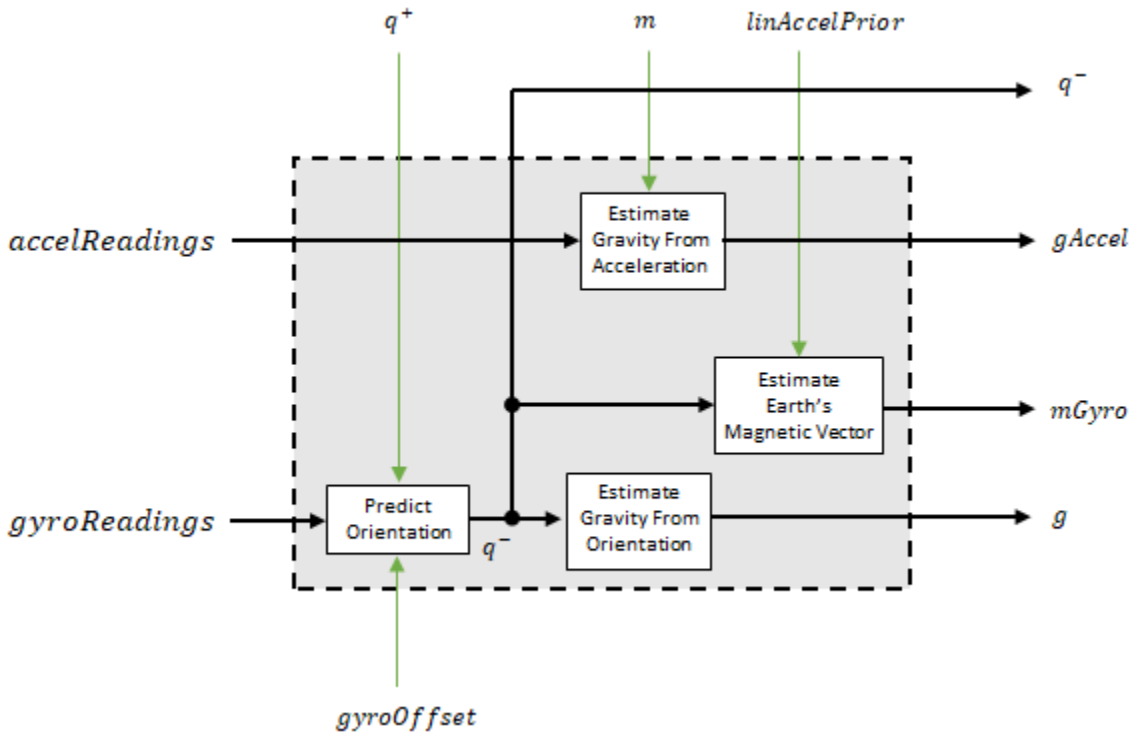
Detailed Overview

Walk through the algorithm for an explanation of each stage of the detailed overview.



Model

The algorithm models acceleration and angular change as linear processes.



Predict Orientation

The orientation for the current frame is predicted by first estimating the angular change from the previous frame:

$$\Delta\varphi_{N\times 3} = \frac{(gyroReadings_{N\times 3} - gyroOffset_{1\times 3})}{fs}$$

where N is the decimation factor specified by the Decimation factor and fs is the sample rate.

The angular change is converted into quaternions using the `rotvec` `quaternion` construction syntax:

$$\Delta Q_{N\times 1} = \text{quaternion}(\Delta\varphi_{N\times 3}, \text{'rotvec'})$$

The previous orientation estimate is updated by rotating it by ΔQ:

$$q_{1\times 1}^{-} = \left(q_{1\times 1}^{+}\right)\left(\prod_{n=1}^{N}\Delta Q_n\right)$$

During the first iteration, the orientation estimate, q^{-}, is initialized by `ecompass`.

Estimate Gravity from Orientation

The gravity vector is interpreted as the third column of the quaternion, q^{-}, in rotation matrix form:

$$g_{1\times 3} = (rPrior(:,3))^{T}$$

Estimate Gravity from Acceleration

A second gravity vector estimation is made by subtracting the decayed linear acceleration estimate of the previous iteration from the accelerometer readings:

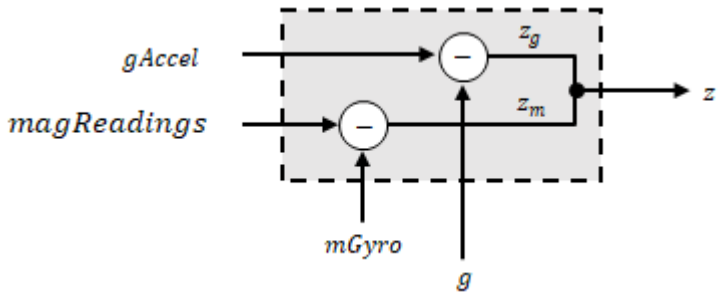
$$gAccel_{1\times 3} = accelReadings_{1\times 3} - linAccelprior_{1\times 3}$$

Estimate Earth's Magnetic Vector

Earth's magnetic vector is estimated by rotating the magnetic vector estimate from the previous iteration by the *a priori* orientation estimate, in rotation matrix form:

$$mGyro_{1\times 3} = \left((rPrior)\left(m^{T}\right)\right)^{T}$$

Error Model

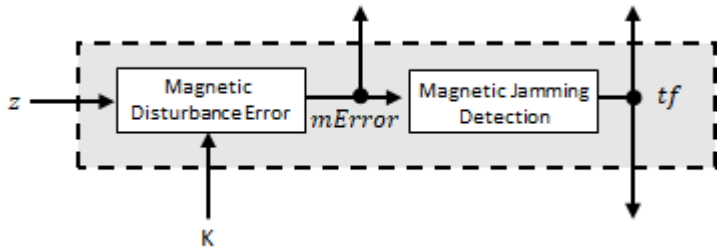


The error model combines two differences:

- The difference between the gravity estimate from the accelerometer readings and the gravity estimate from the gyroscope readings: $z_g = g - gAccel$
- The difference between the magnetic vector estimate from the gyroscope readings and the magnetic vector estimate from the magnetometer:$z_m = mGyro - magReadings$

Magnetometer Correct

The magnetometer correct estimates the error in the magnetic vector estimate and detects magnetic jamming.



Magnetometer Disturbance Error

The magnetic disturbance error is calculated by matrix multiplication of the Kalman gain associated with the magnetic vector with the error signal:

$$mError_{3\times 1} = \left((K(10{:}12,:)_{3\times 6})(z_{1\times 6})^T\right)^T$$

The Kalman gain, K, is the Kalman gain calculated in the current iteration.

Magnetic Jamming Detection

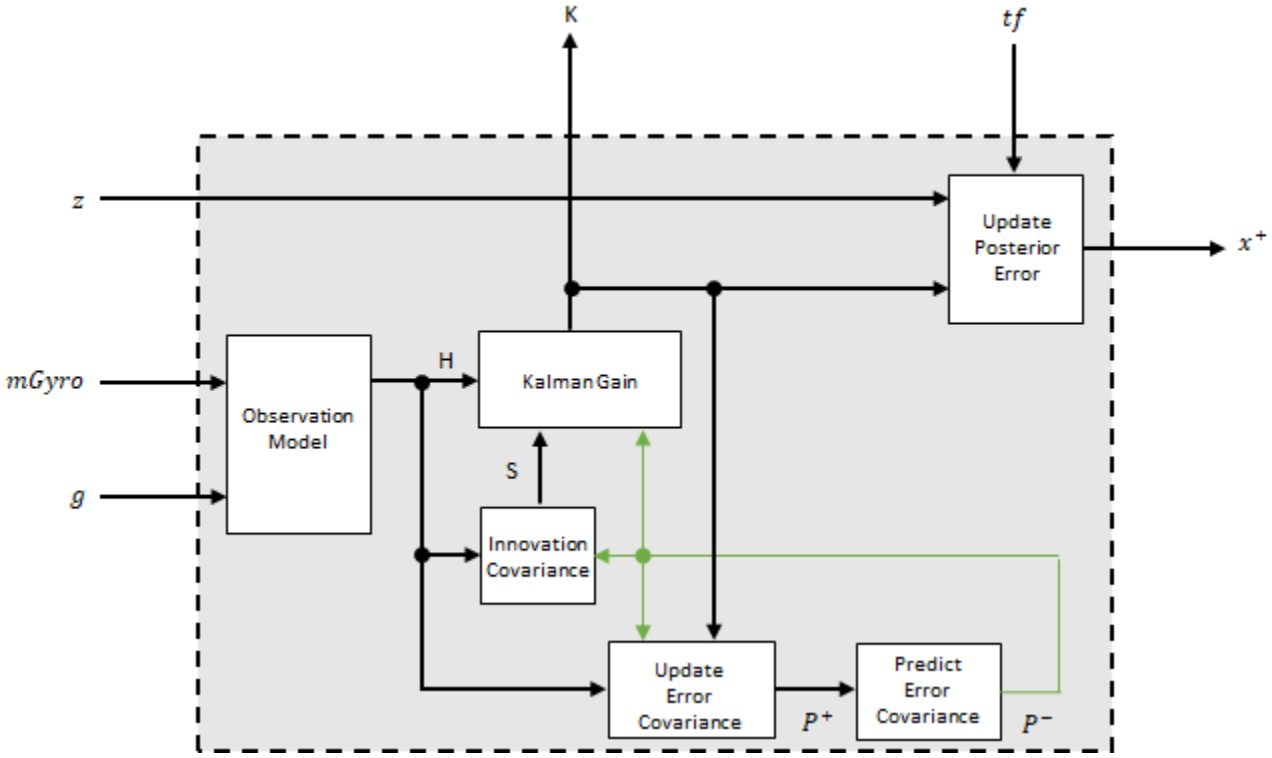
Magnetic jamming is determined by verifying that the power of the detected magnetic disturbance is less than or equal to four times the power of the expected magnetic field strength:

$$tf = \begin{cases} \text{true} & \text{if } \sum|mError|^2 > (4)(\text{ExpectedMagneticFieldStrength})^2 \\ \text{false} & \text{else} \end{cases}$$

ExpectedMagneticFieldStrength is a property of `ahrsfilter`.

Kalman Equations

The Kalman equations use the gravity estimate derived from the gyroscope readings, g, the magnetic vector estimate derived from the gyroscope readings, *mGyro*, and the observation of the error process, z, to update the Kalman gain and intermediary covariance matrices. The Kalman gain is applied to the error signal, z, to output an *a posteriori* error estimate, x^+.



Observation Model

The observation model maps the 1-by-3 observed states, g and *mGyro*, into the 6-by-12 true state, H.

The observation model is constructed as:

$$H_{3\times 9} = \begin{bmatrix} 0 & g_z & -g_y & 0 & -\kappa g_z & \kappa g_y & 1 & 0 & 0 & 0 & 0 & 0 \\ -g_z & 0 & g_x & \kappa g_z & 0 & -\kappa g_x & 0 & 1 & 0 & 0 & 0 & 0 \\ g_y & -g_x & 0 & -\kappa g_y & \kappa g_x & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & m_z & -m_y & 0 & -\kappa m_z & -\kappa m_y & 0 & 0 & 0 & -1 & 0 & 0 \\ -m_z & 0 & m_x & \kappa m_z & 0 & -\kappa m_x & 0 & 0 & 0 & 0 & -1 & 0 \\ m_y & -m_x & 0 & -\kappa m_y & \kappa m_x & 0 & 0 & 0 & 0 & 0 & 0 & -1 \end{bmatrix}$$

where g_x, g_y, and g_z are the *x*-, *y*-, and *z*-elements of the gravity vector estimated from the *a priori* orientation, respectively. m_x, m_y, and m_z are the *x*-, *y*-, and *z*-elements of the magnetic vector estimated from the *a priori* orientation, respectively. κ is a constant determined by the Sample rate and Decimation factor properties: κ = `Decimation factor/Sample rate`.

Innovation Covariance

The innovation covariance is a 6-by-6 matrix used to track the variability in the measurements. The innovation covariance matrix is calculated as:

$$S_{6x6} = R_{6x6} + (H_{6x12})\left(P^{-}_{12x12}\right)(H_{6x12})^{T}$$

where

- H is the observation model matrix
- P^- is the predicted (*a priori*) estimate of the covariance of the observation model calculated in the previous iteration
- R is the covariance of the observation model noise, calculated as:

$$R_{6\times 6} = \begin{bmatrix} accel_{\text{noise}} & 0 & 0 & 0 & 0 & 0 \\ 0 & accel_{\text{noise}} & 0 & 0 & 0 & 0 \\ 0 & 0 & accel_{\text{noise}} & 0 & 0 & 0 \\ 0 & 0 & 0 & mag_{\text{noise}} & 0 & 0 \\ 0 & 0 & 0 & 0 & mag_{\text{noise}} & 0 \\ 0 & 0 & 0 & 0 & 0 & mag_{\text{noise}} \end{bmatrix}$$

where

$accel_{\text{noise}}$ = AccelerometerNoise + LinearAccelerationNoise + κ^2 (GyroscopeDriftNoise + GyroscopeNoise)

and

mag_{noise} = MagnetometerNoise + MagneticDisturbanceNoise + κ^2 (GyroscopeDriftNoise + GyroscopeNoise)

Update Error Estimate Covariance

The error estimate covariance is a 12-by-12 matrix used to track the variability in the state.

The error estimate covariance matrix is updated as:

$$P^+_{12\times 12} = P^-_{12\times 12} - (K_{12\times 6})(H_{6\times 12})\left(P^-_{12\times 12}\right)$$

where K is the Kalman gain, H is the measurement matrix, and P^- is the error estimate covariance calculated during the previous iteration.

Predict Error Estimate Covariance

The error estimate covariance is a 12-by-12 matrix used to track the variability in the state. The *a priori* error estimate covariance, P^-, is set to the process noise covariance, Q, determined during the previous iteration. Q is calculated as a function of the *a posteriori* error estimate covariance, P^+. When calculating Q, it is assumed that the cross-correlation terms are negligible compared to the autocorrelation terms, and are set to zero:

$$Q =$$

$$\begin{bmatrix} P^+(1) + \kappa^2 P^+(40) + \beta + \eta & 0 & 0 & -\kappa\left(P^+(40) + \beta\right) & 0 \\ 0 & P^+(14) + \kappa^2 P^+(53) + \beta + \eta & 0 & 0 & -\kappa\left(P^+(53) + \\ 0 & 0 & P^+(27) + \kappa^2 P^+(66) + \beta + \eta & 0 & 0 \\ -\kappa\left(P^+(40) + \beta\right) & 0 & 0 & P^+(40) + \beta & 0 \\ 0 & -\kappa\left(P^+(53) + \beta\right) & 0 & 0 & P^+(53) + \beta \\ 0 & 0 & -\kappa\left(P^+(66) + \beta\right) & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

where

- P^+ –– is the updated (*a posteriori*) error estimate covariance
- κ –– Decimation factor divided by sample rate.
- β –– Gyroscope drift noise.
- η –– Gyroscope noise.
- ν –– Linear acceleration decay factor.
- ξ –– Linear acceleration noise.
- σ –– Magnetic disturbance decay factor.
- γ –– Magnetic disturbance noise.

Kalman Gain

The Kalman gain matrix is a 12-by-6 matrix used to weight the innovation. In this algorithm, the innovation is interpreted as the error process, z.

The Kalman gain matrix is constructed as:

$$K_{12\times 6} = \left(P^-_{12\times 12}\right)\left(H_{6\times 12}\right)^T\left(\left(S_{6\times 6}\right)^T\right)^{-1}$$

where

- P^- –– predicted error covariance
- H –– observation model
- S –– innovation covariance

Update a Posteriori Error

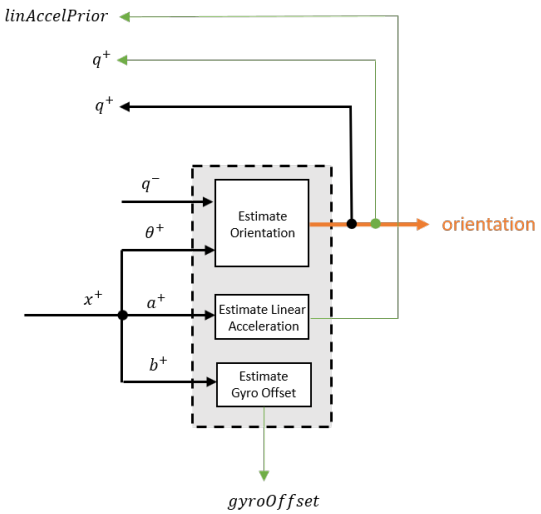
The *a posterior* error estimate is determined by combining the Kalman gain matrix with the error in the gravity vector and magnetic vector estimations:

$$x_{12\times 1} = \left(K_{12\times 6}\right)\left(z_{1\times 6}\right)^T$$

If magnetic jamming is detected in the current iteration, the magnetic vector error signal is ignored, and the *a posterior* error estimate is calculated as:

$$x_{9\times 1} = \left(K(1{:}9, 1{:}3)\right)\left(z_g\right)^T$$

Correct



Estimate Orientation

The orientation estimate is updated by multiplying the previous estimation by the error:

$$q^+ = (q^-)\left(\theta^+\right)$$

Estimate Linear Acceleration

The linear acceleration estimation is updated by decaying the linear acceleration estimation from the previous iteration and subtracting the error:

$$linAccelPrior = (linAccelPrior_{k-1})\nu - b^+$$

where

- ν –- Linear acceleration decay factor

Estimate Gyroscope Offset

The gyroscope offset estimation is updated by subtracting the gyroscope offset error from the gyroscope offset from the previous iteration:

$$gyroOffset = gyroOffset_{k-1} - a^+$$

Compute Angular Velocity

To estimate angular velocity, the frame of `gyroReadings` are averaged and the gyroscope offset computed in the previous iteration is subtracted:

$$angularVelocity_{1\times 3} = \frac{\sum gyroReadings_{N\times 3}}{N} - gyroOffset_{1\times 3}$$

where N is the decimation factor specified by the `DecimationFactor` property.

The gyroscope offset estimation is initialized to zeros for the first iteration.

Update Magnetic Vector

If magnetic jamming was not detected in the current iteration, the magnetic vector estimate, m, is updated using the *a posteriori* magnetic disturbance error and the *a posteriori* orientation.

The magnetic disturbance error is converted to the navigation frame:

$$mErrorNED_{1\times 3} = \left((rPost_{3\times 3})^T (mError_{1\times 3})^T\right)^T$$

The magnetic disturbance error in the navigation frame is subtracted from the previous magnetic vector estimate and then interpreted as inclination:

$$M = m - mErrorNED$$

$$inclination = \text{atan2}(M(3), M(1))$$

The inclination is converted to a constrained magnetic vector estimate for the next iteration:

$m(1)$ = (ExpectedMagneticFieldStrength)(cos(*inclination*))
$m(2) = 0$
$m(3)$ = (ExpectedMagneticFieldStrength)(sin(*inclination*))

References

[1] Open Source Sensor Fusion. https://github.com/memsindustrygroup/Open-Source-Sensor-Fusion/tree/master/docs

[2] Roetenberg, D., H.J. Luinge, C.T.M. Baten, and P.H. Veltink. "Compensation of Magnetic Disturbances Improves Inertial and Magnetic Sensing of Human Body Segment Orientation." *IEEE Transactions on Neural Systems and Rehabilitation Engineering*. Vol. 13. Issue 3, 2005, pp. 395-405.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

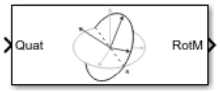
`ahrsfilter` | `ecompass` | `imufilter` | `imuSensor` | `gpsSensor`

Introduced in R2020a

Coordinate Transformation Conversion

Convert to a specified coordinate transformation representation

Library: Robotics System Toolbox / Utilities
Navigation Toolbox / Utilities
ROS Toolbox / Utilities
UAV Toolbox / Utilities



Description

The Coordinate Transformation Conversion block converts a coordinate transformation from the input representation to a specified output representation. The input and output representations use the following forms:

- Axis-Angle (`AxAng`) – `[x y z theta]`
- Euler Angles (`Eul`) – `[z y x]`, `[z y z]`, or `[x y z]`
- Homogeneous Transformation (`TForm`) – 4-by-4 matrix
- Quaternion (`Quat`) – `[w x y z]`
- Rotation Matrix (`RotM`) – 3-by-3 matrix
- Translation Vector (`TrVec`) – `[x y z]`

All vectors must be **column vectors**.

To accommodate representations that only contain position or orientation information (`TrVec` or `Eul`, for example), you can specify two inputs or outputs to handle all transformation information. When you select the Homogeneous Transformation as an input or output, an optional `Show TrVec input/output port` parameter can be selected on the block mask to toggle the multiple ports.

Ports

Input

`Input transformation` — Coordinate transformation

column vector | 3-by-3 matrix | 4-by-4 matrix

Input transformation, specified as a coordinate transformation. The following representations are supported:

- Axis-Angle (`AxAng`) – `[x y z theta]`
- Euler Angles (`Eul`) – `[z y x]`, `[z y z]`, or `[x y z]`
- Homogeneous Transformation (`TForm`) – 4-by-4 matrix
- Quaternion (`Quat`) – `[w x y z]`
- Rotation Matrix (`RotM`) – 3-by-3 matrix

- Translation Vector (`TrVec`) – `[x y z]`

All vectors must be **column vectors**.

To accommodate representations that only contain position or orientation information (`TrVec` or `Eul`, for example), you can specify two inputs or outputs to handle all transformation information. When you select the Homogeneous Transformation as an input or output, an optional `Show TrVec input/output port` parameter can be selected on the block mask to toggle the multiple ports.

`TrVec` — Translation vector

3-element column vector

Translation vector, specified as a 3-element column vector, `[x y z]`, which corresponds to a translation in the *x*, *y*, and *z* axes respectively. This port can be used to input or output the translation information separately from the rotation vector.

Dependencies

You must select Homogeneous Transformation (`TForm`) for the opposite transformation port to get the option to show the additional `TrVec` port. Enable the port by clicking `Show TrVec input/output port`.

Output Arguments

`Output transformation` — Coordinate transformation

column vector | 3-by-3 matrix | 4-by-4 matrix

Output transformation, specified as a coordinate transformation with the specified representation. The following representations are supported:

- Axis-Angle (`AxAng`) – `[x y z theta]`
- Euler Angles (`Eul`) – `[z y x]`, `[z y z]`, or `[x y z]`
- Homogeneous Transformation (`TForm`) – 4-by-4 matrix
- Quaternion (`Quat`) – `[w x y z]`
- Rotation Matrix (`RotM`) – 3-by-3 matrix
- Translation Vector (`TrVec`) – `[x y z]`

To accommodate representations that only contain position or orientation information (`TrVec` or `Eul`, for example), you can specify two inputs or outputs to handle all transformation information. When you select the Homogeneous Transformation as an input or output, an optional `Show TrVec input/output port` parameter can be selected on the block mask to toggle the multiple ports.

`TrVec` — Translation vector

three-element column vector

Translation vector, specified as a three-element column vector, `[x y z]`, which corresponds to a translation in the *x*, *y*, and *z* axes respectively. This port can be used to input or output the translation information separately from the rotation vector.

Dependencies

You must select Homogeneous Transformation (`TForm`) for the opposite transformation port to get the option to show the additional `TrVec` port. Enable the port by clicking `Show TrVec input/output port`.

Parameters

`Representation` — Input or output representation

`Axis-Angle` | `Euler Angles` | `Homogeneous Transformation` | `Rotation Matrix` | `Translation Vector` | `Quaternion`

Select the representation for both the input and output port for the block. If you are using a transformation with only orientation information, you can also select the `Show TrVec input/output port` when converting to or from a homogeneous transformation.

`Axis rotation sequence` — Order of Euler angle axis rotations

ZYX (default) | ZYZ | XYZ

Order of the Euler angle axis rotations, specified as `ZYX`, `ZYZ`, or `XYZ`. The order of the angles in the input or output port `Eul` must match this rotation sequence. The default order `ZYX` specifies an orientation by:

- Rotating about the initial *z*-axis
- Rotating about the intermediate *y*-axis
- Rotating about the second intermediate *x*-axis

Dependencies

You must select `Euler Angles` for the `Representation` input or output parameter. The axis rotation sequence only applies to Euler angle rotations.

`Show TrVec input/output port` — Toggle `TrVec` port

off (default) | on

Toggle the `TrVec` input or output port when you want to specify or receive a separate translation vector for position information along with an orientation representation.

Dependencies

You must select Homogeneous Transformation (`TForm`) for the opposite transformation port to get the option to show the additional `TrVec` port.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

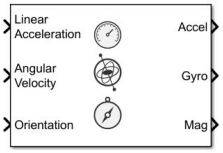
`axang2quat` | `eul2tform` | `trvec2tform`

Introduced in R2017b

IMU

IMU simulation model

Library: Sensor Fusion and Tracking Toolbox / Multisensor Positioning / Sensor Models
Navigation Toolbox / Multisensor Positioning / Sensor Models



Description

The IMU Simulink block models receiving data from an inertial measurement unit (IMU) composed of accelerometer, gyroscope, and magnetometer sensors.

Ports

Input

`Linear Acceleration` — Acceleration of IMU in local navigation coordinate system (m/s²)

N-by-3 matrix of real scalar

Acceleration of the IMU in the local navigation coordinate system, specified as an *N*-by-3 matrix of real scalars in meters per second squared. *N* is the number of samples in the current frame.

Data Types: `single` | `double`

`Angular Velocity` — Angular velocity of IMU in local navigation coordinate system (rad/s)

N-by-3 matrix of real scalar

Angular velocity of the IMU sensor body frame in the local navigation coordinate system, specified as an *N*-by-3 matrix of scalars in radians per second. *N* is the number of samples in the current frame.

Data Types: `single` | `double`

`Orientation` — Orientation of IMU in local navigation coordinate system

N-by-4 array of real scalar | 3-by-3-by-*N*-element rotation matrix

Orientation of the IMU sensor body frame with respect to the local navigation coordinate system, specified as an *N*-by-4 array of real scalars or a 3-by-3-by-*N* rotation matrix. Each row the of the *N*-by-4 array is assumed to be the four elements of a `quaternion`. *N* is the number of samples in the current frame.

Data Types: `single` | `double`

Output

`Accel` — Accelerometer measurement of IMU in sensor body coordinate system (m/s²)

N-by-3 matrix of real scalar

Accelerometer measurement of the IMU in the sensor body coordinate system, returned as an *N*-by-3 matrix of real scalars in meters per second squared. *N* is the number of samples in the current frame.

Data Types: `single` | `double`

`Gyro` — Gyroscope measurement of IMU in sensor body coordinate system (rad/s)
N-by-3 matrix of real scalar

Gyroscope measurement of the IMU in the sensor body coordinate system, returned as an *N*-by-3 matrix of real scalars in radians per second. *N* is the number of samples in the current frame.

Data Types: `single` | `double`

`Mag` — Magnetometer measurement of IMU in sensor body coordinate system (μT)
N-by-3 matrix of real scalar

Magnetometer measurement of the IMU in the sensor body coordinate system, returned as an *N*-by-3 matrix of real scalars in microtesla. *N* is the number of samples in the current frame.

Data Types: `single` | `double`

Parameters

Parameters

`Reference frame` — Navigation reference frame
`NED` (default) | `ENU`

Navigation reference frame, specified as `NED` (North-East-Down) or `ENU` (East-North-Up).

`Temperature (°C)` — Operating temperature of IMU (°C)
`25` (default) | real scalar

Operating temperature of the IMU in degrees Celsius, specified as a real scalar.

When the block calculates temperature scale factors and environmental drift noises, 25 °C is used as the nominal temperature.

Data Types: `single` | `double`

`Magnetic field (NED)` — Magnetic field vector expressed in NED navigation frame (μT)
`[27.5550, -2.4169, -16.0849]` (default) | 1-by-3 vector of scalar

Magnetic field vector expressed in the NED navigation frame, specified as a 1-by-3 vector of scalars.

The default magnetic field corresponds to the magnetic field at latitude zero, longitude zero, and altitude zero.

Dependencies

To enable this parameter, set **Reference frame** to `NED`.

Data Types: `single` | `double`

`MagneticField (ENU)` — Magnetic field vector expressed in ENU navigation frame (μT)
`[-2.4169, 27.5550, 16.0849]` (default) | 1-by-3 vector of scalar

Magnetic field vector expressed in the ENU navigation frame, specified as a 1-by-3 vector of scalars.

The default magnetic field corresponds to the magnetic field at latitude zero, longitude zero, and altitude zero.

Dependencies

To enable this parameter, set **Reference frame** to `ENU`.

Data Types: `single` | `double`

Seed — Initial seed for randomization

`67` (default) | nonnegative integer

Initial seed of a random number generator algorithm, specified as a nonnegative integer.

Data Types: `single` | `double`

`Simulate using` — Type of simulation to run

`Interpreted Execution` (default) | `Code Generation`

- `Interpreted execution` — Simulate the model using the MATLAB interpreter. This option shortens startup time. In `Interpreted execution` mode, you can debug the source code of the block.
- `Code generation` — Simulate the model using generated C code. The first time that you run a simulation, Simulink generates C code for the block. The C code is reused for subsequent simulations if the model does not change. This option requires additional startup time.

Accelerometer

`Maximum readings (m/s²)` — Maximum sensor reading (m/s²)

`inf` (default) | real positive scalar

Maximum sensor reading in m/s^2, specified as a real positive scalar.

Data Types: `single` | `double`

`Resolution ((m/s²)/LSB)` — Resolution of sensor measurements ((m/s²)/LSB)

`0` (default) | real nonnegative scalar

Resolution of sensor measurements in $(m/s^2)/LSB$, specified as a real nonnegative scalar.

Data Types: `single` | `double`

`Constant offset bias (m/s²)` — Constant sensor offset bias (m/s²)

`[0 0 0]` (default) | real scalar | real 3-element row vector

Constant sensor offset bias in m/s^2, specified as a real scalar or 3-element row vector. Any scalar input is converted into a real 3-element row vector where each element has the input scalar value.

Data Types: `single` | `double`

`Axis skew (%)` — Sensor axes skew (%)

`[0 0 0]` (default) | real scalar in the range [0,100] | real 3-element row vector in the range [0,100]

Sensor axes skew in a percentage, specified as a real scalar or 3-element row vector with values ranging from 0 to 100. Any scalar input is converted into a real 3-element row vector where each element has the input scalar value.

Data Types: `single` | `double`

`Velocity random walk (m/s²/√Hz)` — Velocity random walk (m/s²/√Hz)
`[0 0 0]` (default) | real scalar | real 3-element row vector

Velocity random walk in (m/s^2/√Hz), specified as a real scalar or 3-element row vector. This property corresponds to the power spectral density of sensor noise. Any scalar input is converted into a real 3-element row vector where each element has the input scalar value.

Data Types: `single` | `double`

`Bias Instability (m/s²)` — Instability of the bias offset (m/s²)
`[0 0 0]` (default) | real scalar | real 3-element row vector

Instability of the bias offset in m/s^2, specified as a real scalar or 3-element row vector. Any scalar input is converted into a real 3-element row vector where each element has the input scalar value.

Data Types: `single` | `double`

`Acceleration random walk ((m/s²)(√Hz))` — Acceleration random walk ((m/s²)(√Hz))
`[0 0 0]` (default) | real scalar | real 3-element row vector

Acceleration random walk of sensor in (m/s^2)(√Hz), specified as a real scalar or 3-element row vector. Any scalar input is converted into a real 3-element row vector where each element has the input scalar value.

Data Types: `single` | `double`

`Bias from temperature ((m/s²)/°C)` — Sensor bias from temperature ((m/s²)/°C)
`[0 0 0]` (default) | real scalar | real 3-element row vector

Sensor bias from temperature in (m/s^2)/°C, specified as a real scalar or 3-element row vector. Any scalar input is converted into a real 3-element row vector where each element has the input scalar value.

Data Types: `single` | `double`

`Temperature scale factor (%/°C)` — Scale factor error from temperature (%/°C)
`[0 0 0]` (default) | real scalar in the range [0,100] | real 3-element row vector in the range [0,100]

Scale factor error from temperature in %/°C, specified as a real scalar or real 3-element row vector with values ranging from 0 to 100. Any scalar input is converted into a real 3-element row vector where each element has the input scalar value.

Data Types: `single` | `double`

Gyroscope

`Maximum readings (rad/s)` — Maximum sensor reading (rad/s)
`inf` (default) | real positive scalar

Maximum sensor reading in rad/s, specified as a real positive scalar.

Data Types: `single` | `double`

`Resolution ((rad/s)/LSB)` — Resolution of sensor measurements ((rad/s)/LSB)
`0` (default) | real nonnegative scalar

Resolution of sensor measurements in (rad/s)/LSB, specified as a real nonnegative scalar.

Data Types: `single` | `double`

`Constant offset bias (rad/s)` — Constant sensor offset bias (rad/s)
`[0 0 0]` (default) | real scalar | real 3-element row vector

Constant sensor offset bias in rad/s, specified as a real scalar or 3-element row vector. Any scalar input is converted into a real 3-element row vector where each element has the input scalar value.

Data Types: `single` | `double`

`Axis skew (%)` — Sensor axes skew (%)
`[0 0 0]` (default) | real scalar in the range [0,100] | real 3-element row vector in the range [0,100]

Sensor axes skew in a percentage, specified as a real scalar or 3-element row vector with values ranging from 0 to 100. Any scalar input is converted into a real 3-element row vector where each element has the input scalar value.

Data Types: `single` | `double`

`Bias from acceleration ((rad/s)/(m/s²)` — Sensor bias from linear acceleration (rad/s)/(m/s²)
`[0 0 0]` (default) | real scalar | real 3-element row vector

Sensor bias from linear acceleration in (rad/s)/(m/s^2), specified as a real scalar or 3-element row vector. Any scalar input is converted into a real 3-element row vector where each element has the input scalar value.

Data Types: `single` | `double`

`Angle random walk ((rad/s)/(√Hz))` — Acceleration random walk ((rad/s)/(√Hz))
`[0 0 0]` (default) | real scalar | real 3-element row vector

Acceleration random walk of sensor in (rad/s)/(√Hz), specified as a real scalar or 3-element row vector. Any scalar input is converted into a real 3-element row vector where each element has the input scalar value.

Data Types: `single` | `double`

`Bias Instability (rad/s)` — Instability of the bias offset (rad/s)
`[0 0 0]` (default) | real scalar | real 3-element row vector

Instability of the bias offset in rad/s, specified as a real scalar or 3-element row vector. Any scalar input is converted into a real 3-element row vector where each element has the input scalar value.

Data Types: `single` | `double`

`Rate random walk ((rad/s)(√Hz))` — Integrated white noise of sensor ((rad/s)(√Hz))
`[0 0 0]` (default) | real scalar | real 3-element row vector

Integrated white noise of sensor in (rad/s)(√Hz), specified as a real scalar or 3-element row vector. Any scalar input is converted into a real 3-element row vector where each element has the input scalar value.

Data Types: `single` | `double`

`Bias from temperature ((rad/s)/°C)` — Sensor bias from temperature ((rad/s)/°C)
`[0 0 0]` (default) | real scalar | real 3-element row vector

Sensor bias from temperature in (rad/s)/℃, specified as a real scalar or 3-element row vector. Any scalar input is converted into a real 3-element row vector where each element has the input scalar value.

Data Types: `single` | `double`

`Temperature scale factor (%/℃)` — Scale factor error from temperature (%/°C)
`[0 0 0]` (default) | real scalar in the range [0,100] | real 3-element row vector in the range [0,100]

Scale factor error from temperature in %/℃, specified as a real scalar or real 3-element row vector with values ranging from 0 to 100. Any scalar input is converted into a real 3-element row vector where each element has the input scalar value.

Data Types: `single` | `double`

Magnetometer

`Maximum readings (μT)` — Maximum sensor reading (μT)
`inf` (default) | real positive scalar

Maximum sensor reading in μT, specified as a real positive scalar.

Data Types: `single` | `double`

`Resolution ((μT)/LSB)` — Resolution of sensor measurements ((μT)/LSB)
`0` (default) | real nonnegative scalar

Resolution of sensor measurements in (μT)/LSB, specified as a real nonnegative scalar.

Data Types: `single` | `double`

`Constant offset bias (μT)` — Constant sensor offset bias (μT)
`[0 0 0]` (default) | real scalar | real 3-element row vector

Constant sensor offset bias in μT, specified as a real scalar or 3-element row vector. Any scalar input is converted into a real 3-element row vector where each element has the input scalar value.

Data Types: `single` | `double`

`Axis skew (%)` — Sensor axes skew (%)
`[0 0 0]` (default) | real scalar in the range [0,100] | real 3-element row vector in the range [0,100]

Sensor axes skew in a percentage, specified as a real scalar or 3-element row vector with values ranging from 0 to 100. Any scalar input is converted into a real 3-element row vector where each element has the input scalar value.

Data Types: `single` | `double`

`White noise PSD (μT/√Hz)` — Power spectral density of sensor noise (μT/√Hz)
`[0 0 0]` (default) | real scalar | real 3-element row vector

Power spectral density of sensor noise in μT/√Hz, specified as a real scalar or 3-element row vector. Any scalar input is converted into a real 3-element row vector where each element has the input scalar value.

Data Types: `single` | `double`

`Bias Instability (μT)` — Instability of the bias offset (μT)
`[0 0 0]` (default) | real scalar | real 3-element row vector

Instability of the bias offset in µT, specified as a real scalar or 3-element row vector. Any scalar input is converted into a real 3-element row vector where each element has the input scalar value.

Data Types: `single` | `double`

`Random walk ((µT)*√Hz)` — Integrated white noise of sensor ((µT)*√Hz)
`[0 0 0]` (default) | real scalar | real 3-element row vector

Integrated white noise of sensor in (µT)*√Hz, specified as a real scalar or 3-element row vector. Any scalar input is converted into a real 3-element row vector where each element has the input scalar value.

Data Types: `single` | `double`

`Bias from temperature (µT/℃)` — Sensor bias from temperature (µT/°C)
`[0 0 0]` (default) | real scalar | real 3-element row vector

Sensor bias from temperature in µT/℃, specified as a real scalar or 3-element row vector. Any scalar input is converted into a real 3-element row vector where each element has the input scalar value.

Data Types: `single` | `double`

`Temperature scale factor (%/℃)` — Scale factor error from temperature (%/°C)
`[0 0 0]` (default) | real scalar in the range [0,100] | real 3-element row vector in the range [0,100]

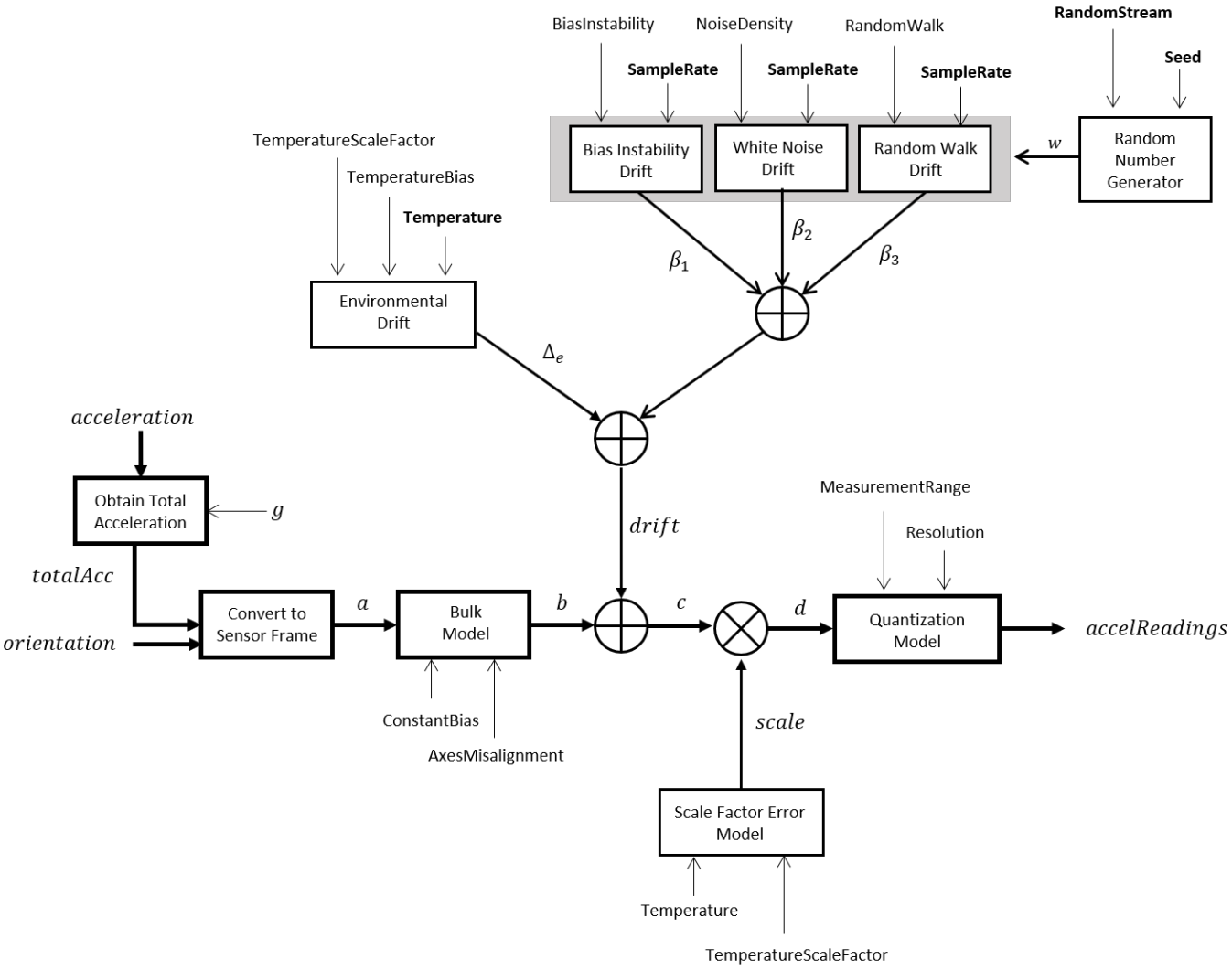
Scale factor error from temperature in %/℃, specified as a real scalar or real 3-element row vector with values ranging from 0 to 100. Any scalar input is converted into a real 3-element row vector where each element has the input scalar value.

Data Types: `single` | `double`

Algorithms

Accelerometer

The following algorithm description assumes an NED navigation frame. The accelerometer model uses the ground-truth orientation and acceleration inputs and the `imuSensor` and `accelparams` properties to model accelerometer readings.



Obtain Total Acceleration

To obtain the total acceleration (*totalAcc*), the acceleration is preprocessed by negating and adding the gravity constant vector (g= [0; 0; 9.8] m/s^2 assuming an NED frame) as:

$$totalAcc = -acceleration + g$$

The `acceleration` term is negated to obtain zero total acceleration readings when the accelerometer is in a free fall. The `acceleration` term is also known as the specific force.

Convert to Sensor Frame

Then the total acceleration is converted from the local navigation frame to the sensor frame using:

$$a = (orientation)(totalAcc)^T$$

If the orientation is input in quaternion form, it is converted to a rotation matrix before processing.

Bulk Model

The ground-truth acceleration in the sensor frame, a, passes through the bulk model, which adds axes misalignment and bias:

$$b = \left(\begin{bmatrix} 1 & \frac{\alpha_2}{100} & \frac{\alpha_3}{100} \\ \frac{\alpha_1}{100} & 1 & \frac{\alpha_3}{100} \\ \frac{\alpha_1}{100} & \frac{\alpha_2}{100} & 1 \end{bmatrix} \left(a^T\right) \right)^T + \text{ConstantBias}$$

where ConstantBias is a property of `accelparams`, and α_1, α_2, and α_3 are given by the first, second, and third elements of the AxesMisalignment property of `accelparams`.

Bias Instability Drift

The bias instability drift is modeled as white noise biased and then filtered:

$$\beta_1 = h_1 * (w)(\text{BiasInstability})$$

where BiasInstability is a property of `accelparams`, and h_1 is a filter defined by the SampleRate property:

$$H_1(z) = \frac{1}{1 - \frac{1}{2}z^{-1}}$$

White Noise Drift

White noise drift is modeled by multiplying elements of the white noise random stream by the standard deviation:

$$\beta_2 = (w)\left(\sqrt{\frac{\text{SampleRate}}{2}}\right)(\text{NoiseDensity})$$

where SampleRate is an `imuSensor` property, and NoiseDensity is an `accelparams` property. Elements of w are random numbers given by settings of the `imuSensor` random stream.

Random Walk Drift

The random walk drift is modeled by biasing elements of the white noise random stream and then filtering:

$$\beta_3 = h_2 * (w)\left(\frac{\text{RandomWalk}}{\sqrt{\frac{\text{SampleRate}}{2}}}\right)$$

where RandomWalk is a property of `accelparams`, SampleRate is a property of `imuSensor`, and h_2 is a filter defined as:

$$H_2(z) = \frac{1}{1 - z^{-1}}$$

Environmental Drift Noise

The environmental drift noise is modeled by multiplying the temperature difference from a standard with the temperature bias:

$$\Delta_e = (\text{Temperature} - 25)(\text{TemperatureBias})$$

where Temperature is a property of `imuSensor`, and TemperatureBias is a property of `accelparams`. The constant 25 corresponds to a standard temperature.

Scale Factor Error Model

The temperature scale factor error is modeled as:

$$scaleFactorError = 1 + \left(\frac{\text{Temperature} - 25}{100}\right)(\text{TemperatureScaleFactor})$$

where Temperature is a property of `imuSensor`, and TemperatureScaleFactor is a property of `accelparams`. The constant 25 corresponds to a standard temperature.

Quantization Model

The quantization is modeled by first saturating the continuous signal model:

$$e = \begin{cases} \text{MeasurementRange} & \text{if } d > \text{MeasurementRange} \\ -\text{MeasurementRange} & \text{if } -d > \text{MeasurementRange} \\ d & \text{else} \end{cases}$$

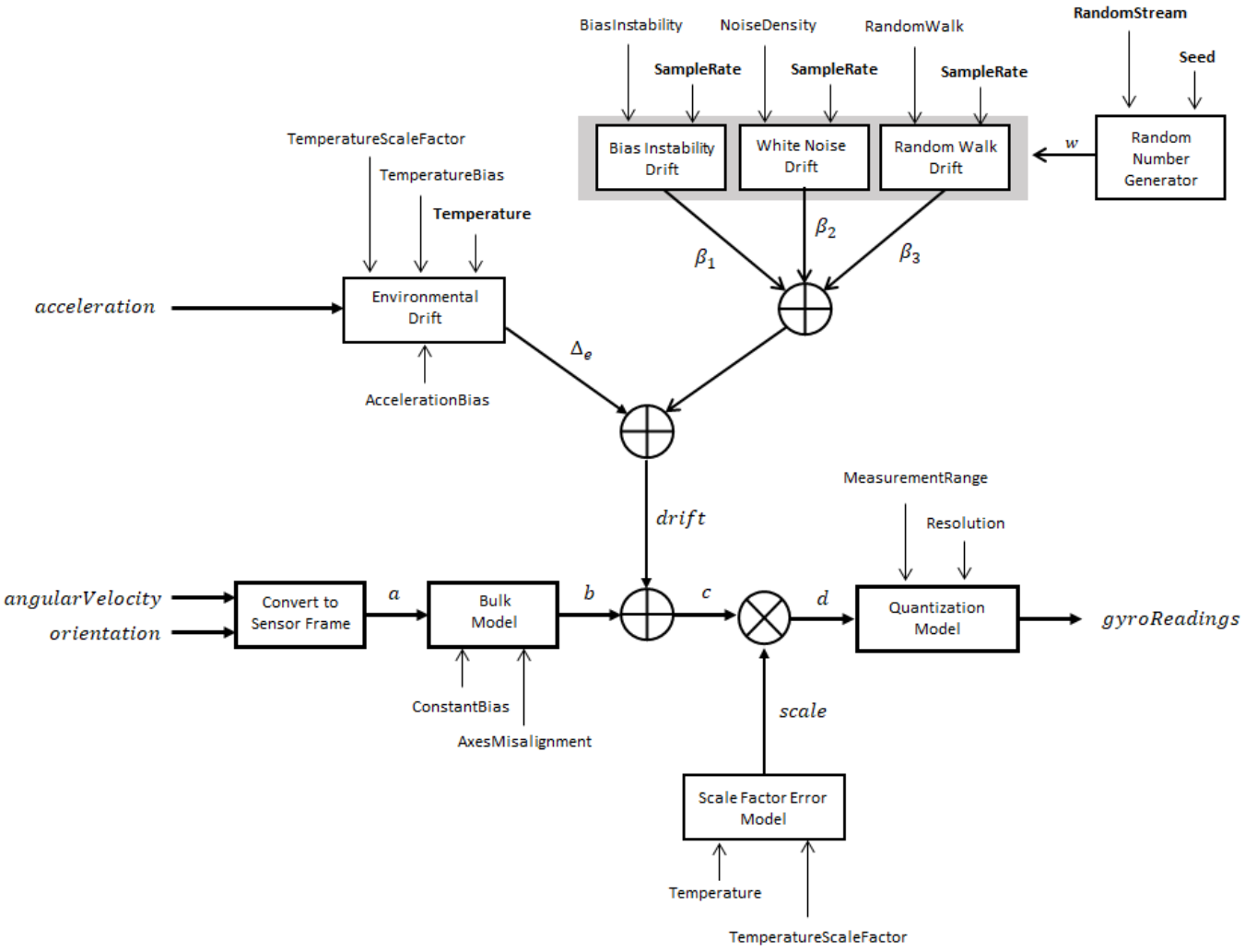
and then setting the resolution:

$$accelReadings = (\text{Resolution})\left(\text{round}\left(\frac{e}{\text{Resolution}}\right)\right)$$

where MeasurementRange is a property of `accelparams`.

Gyroscope

The following algorithm description assumes an NED navigation frame. The gyroscope model uses the ground-truth orientation, acceleration, and angular velocity inputs, and the `imuSensor` and `gyroparams` properties to model accelerometer readings.



Convert to Sensor Frame

The ground-truth angular velocity is converted from the local frame to the sensor frame using the ground-truth orientation:

$$a = (orientation)(angularVelocity)^T$$

If the orientation is input in quaternion form, it is converted to a rotation matrix before processing.

Bulk Model

The ground-truth angular velocity in the sensor frame, a, passes through the bulk model, which adds axes misalignment and bias:

$$b = \left(\begin{bmatrix} 1 & \frac{\alpha_2}{100} & \frac{\alpha_3}{100} \\ \frac{\alpha_1}{100} & 1 & \frac{\alpha_3}{100} \\ \frac{\alpha_1}{100} & \frac{\alpha_2}{100} & 1 \end{bmatrix} \left(a^T\right) \right)^T + \text{ConstantBias}$$

where ConstantBias is a property of `gyroparams`, and α_1, α_2, and α_3 are given by the first, second, and third elements of the AxesMisalignment property of `gyroparams`.

Bias Instability Drift

The bias instability drift is modeled as white noise biased and then filtered:

$$\beta_1 = h_1 * (w)(\text{BiasInstability})$$

where BiasInstability is a property of `gyroparams` and h_1 is a filter defined by the SampleRate property:

$$H_1(z) = \frac{1}{1 - \frac{1}{2}z^{-1}}$$

White Noise Drift

White noise drift is modeled by multiplying elements of the white noise random stream by the standard deviation:

$$\beta_2 = (w)\left(\sqrt{\frac{\text{SampleRate}}{2}}\right)(\text{NoiseDensity})$$

where SampleRate is an `imuSensor` property, and NoiseDensity is an `gyroparams` property. The elements of w are random numbers given by settings of the `imuSensor` random stream.

Random Walk Drift

The random walk drift is modeled by biasing elements of the white noise random stream and then filtering:

$$\beta_3 = h_2 * (w)\left(\frac{\text{RandomWalk}}{\sqrt{\frac{\text{SampleRate}}{2}}}\right)$$

where RandomWalk is a property of `gyroparams`, SampleRate is a property of `imuSensor`, and h_2 is a filter defined as:

$$H_2(z) = \frac{1}{1 - z^{-1}}$$

Environmental Drift Noise

The environmental drift noise is modeled by multiplying the temperature difference from a standard with the temperature bias:

$$\Delta_e = (\text{Temperature} - 25)(\text{TemperatureBias})$$

where Temperature is a property of `imuSensor`, and TemperatureBias is a property of `gyroparams`. The constant 25 corresponds to a standard temperature.

Scale Factor Error Model

The temperature scale factor error is modeled as:

$$scaleFactorError = 1 + \left(\frac{\text{Temperature} - 25}{100}\right)(\text{TemperatureScaleFactor})$$

where Temperature is a property of `imuSensor`, and TemperatureScaleFactor is a property of `gyroparams`. The constant 25 corresponds to a standard temperature.

Quantization Model

The quantization is modeled by first saturating the continuous signal model:

$$e = \begin{cases} \text{MeasurementRange} & \text{if } d > \text{MeasurementRange} \\ -\text{MeasurementRange} & \text{if } -d > \text{MeasurementRange} \\ d & \text{else} \end{cases}$$

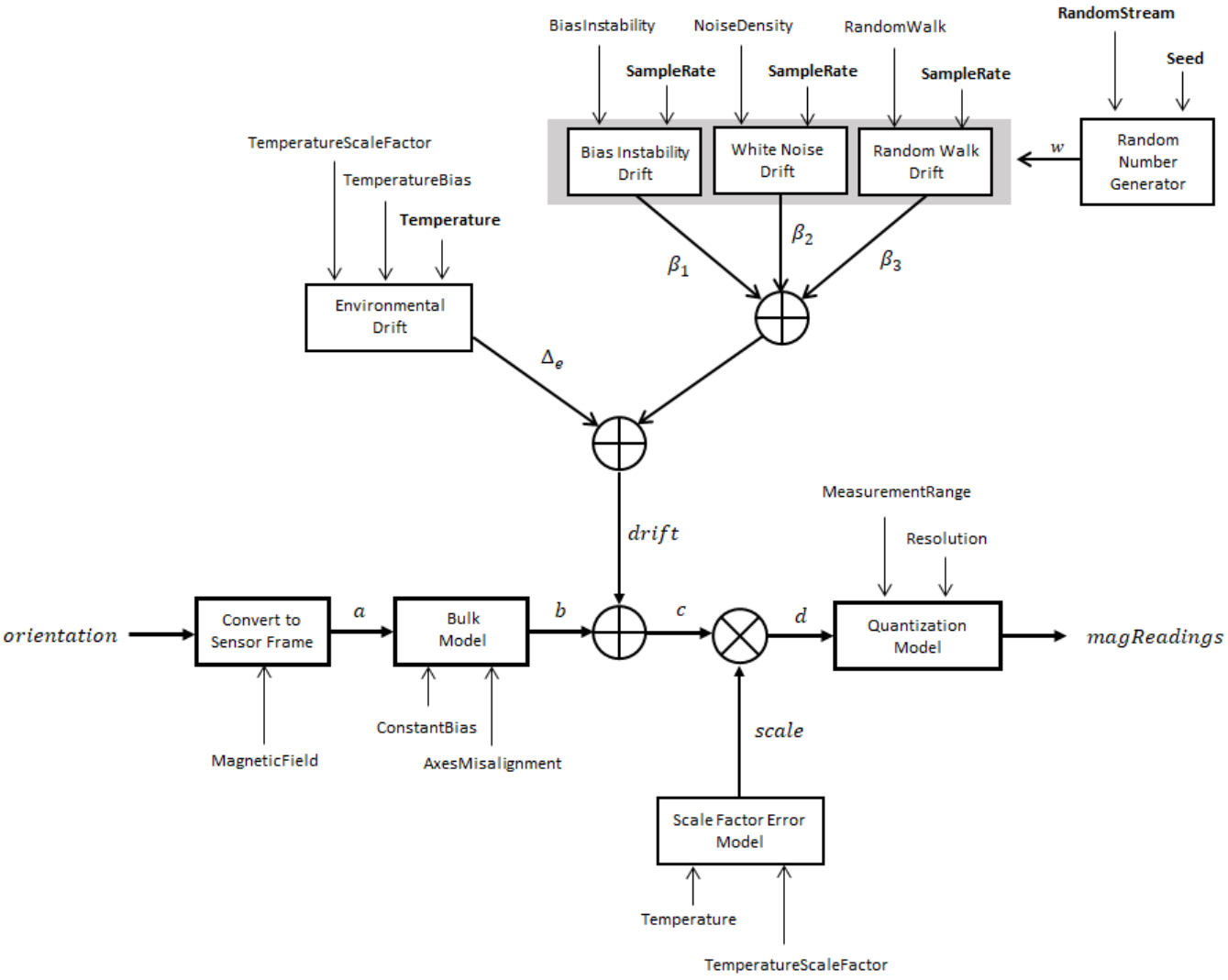
and then setting the resolution:

$$gyroReadings = (\text{Resolution})\left(\text{round}\left(\frac{e}{\text{Resolution}}\right)\right)$$

where MeasurementRange is a property of `gyroparams`.

Magnetometer

The following algorithm description assumes an NED navigation frame. The magnetometer model uses the ground-truth orientation and acceleration inputs, and the `imuSensor` and `magparams` properties to model magnetometer readings.



Convert to Sensor Frame

The ground-truth acceleration is converted from the local frame to the sensor frame using the ground-truth orientation:

$$a = (orientation)(totalAcc)^T$$

If the orientation is input in quaternion form, it is converted to a rotation matrix before processing.

Bulk Model

The ground-truth acceleration in the sensor frame, a, passes through the bulk model, which adds axes misalignment and bias:

$$b = \left(\begin{bmatrix} 1 & \frac{\alpha_2}{100} & \frac{\alpha_3}{100} \\ \frac{\alpha_1}{100} & 1 & \frac{\alpha_3}{100} \\ \frac{\alpha_1}{100} & \frac{\alpha_2}{100} & 1 \end{bmatrix} \left(a^T\right) \right)^T + \text{ConstantBias}$$

where ConstantBias is a property of `magparams`, and α_1, α_2, and α_3 are given by the first, second, and third elements of the AxesMisalignment property of `magparams`.

Bias Instability Drift

The bias instability drift is modeled as white noise biased and then filtered:

$$\beta_1 = h_1 * (w)(\text{BiasInstability})$$

where BiasInstability is a property of `magparams` and h_1 is a filter defined by the SampleRate property:

$$H_1(z) = \frac{1}{1 - \frac{1}{2}z^{-1}}$$

White Noise Drift

White noise drift is modeled by multiplying elements of the white noise random stream by the standard deviation:

$$\beta_2 = (w)\left(\sqrt{\frac{\text{SampleRate}}{2}}\right)(\text{NoiseDensity})$$

where SampleRate is an `imuSensor` property, and NoiseDensity is an `magparams` property. The elements of w are random numbers given by settings of the `imuSensor` random stream.

Random Walk Drift

The random walk drift is modeled by biasing elements of the white noise random stream and then filtering:

$$\beta_3 = h_2 * (w)\left(\frac{\text{RandomWalk}}{\sqrt{\frac{\text{SampleRate}}{2}}}\right)$$

where RandomWalk is a property of `magparams`, SampleRate is a property of `imuSensor`, and h_2 is a filter defined as:

$$H_2(z) = \frac{1}{1 - z^{-1}}$$

Environmental Drift Noise

The environmental drift noise is modeled by multiplying the temperature difference from a standard with the temperature bias:

$$\Delta_e = (\text{Temperature} - 25)(\text{TemperatureBias})$$

where Temperature is a property of `imuSensor`, and TemperatureBias is a property of `magparams`. The constant 25 corresponds to a standard temperature.

Scale Factor Error Model

The temperature scale factor error is modeled as:

$$scaleFactorError = 1 + \left(\frac{\text{Temperature} - 25}{100}\right)(\text{TemperatureScaleFactor})$$

where Temperature is a property of `imuSensor`, and TemperatureScaleFactor is a property of `magparams`. The constant 25 corresponds to a standard temperature.

Quantization Model

The quantization is modeled by first saturating the continuous signal model:

$$e = \begin{cases} \text{MeasurementRange} & \text{if } d > \text{MeasurementRange} \\ -\text{MeasurementRange} & \text{if } -d > \text{MeasurementRange} \\ d & \text{else} \end{cases}$$

and then setting the resolution:

$$magReadings = (\text{Resolution})\left(\text{round}\left(\frac{e}{\text{Resolution}}\right)\right)$$

where MeasurementRange is a property of `magparams`.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

Classes

`accelparams` | `gyroparams` | `magparams`

Objects

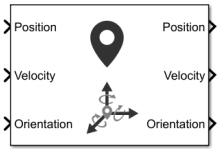
`imuSensor` | `gpsSensor`

Introduced in R2020a

INS

Simulate INS sensor

Library: Navigation Toolbox / Multisensor Positioning / Sensor Models
Automated Driving Toolbox / Driving Scenario and Sensor Modeling
Sensor Fusion and Tracking Toolbox / Multisensor Positioning / Sensor Models
UAV Toolbox / UAV Scenario and Sensor Modeling



Description

The block simulates an INS sensor, which outputs noise-corrupted position, velocity, and orientation based on the corresponding inputs. The block can also optionally output acceleration and angular velocity based on the corresponding inputs. To change the level of noise present in the output, you can vary the roll, pitch, yaw, position, velocity, acceleration, and angular velocity accuracies. The accuracy is defined as the standard deviation of the noise.

Ports

Input

`Position` — Position of INS sensor

N-by-3 real-valued matrix

Position of the INS sensor relative to the navigation frame, in meters, specified as an *N*-by-3 real-valued matrix. *N* is the number of samples.

Data Types: `single` | `double`

`Velocity` — Velocity of INS sensor

N-by-3 real-valued matrix of scalar

Velocity of the INS sensor relative to the navigation frame, in meters per second, specified as an *N*-by-3 real-valued matrix. *N* is the number of samples.

Data Types: `single` | `double`

`Orientation` — Orientation of INS sensor

3-by-3-by-*N* real-valued array | *N*-by-4 real-valued matrix | *N*-by-3 matrix of Euler angles

Orientation of the INS sensor relative to the navigation frame, specified as one of these formats:

- A 3-by-3-by-*N* real-valued array, where each page of the array (3-by-3 matrix) is a rotation matrix.
- An *N*-by-4 real-valued matrix, where each row of the matrix is the four elements of a quaternion.

- An *N*-by-3 matrix of Euler angles, where each row of the matrix is the three Euler angles corresponding to the z-y-x rotation convention.

N is the number of samples.

Data Types: `single` | `double`

`Acceleration` — Acceleration of INS sensor

N-by-3 real-valued matrix

Acceleration of the INS sensor relative to the navigation frame, in meters per second squared, specified as an *N*-by-3 real-valued matrix. *N* is the number of samples.

Dependencies

To enable this input port, select **Use acceleration and angular velocity**.

Data Types: `single` | `double`

`AngularVelocity` — Angular velocity of INS sensor

N-by-3 real-valued matrix

Angular velocity of the INS sensor relative to the navigation frame, in degrees per second, specified as an *N*-by-3 real-valued matrix. *N* is the number of samples.

Dependencies

To enable this input port, select **Use acceleration and angular velocity**.

Data Types: `single` | `double`

`HasGNSSFix` — Enable GNSS fix

N-by-1 logical vector

Enable GNNS fix, specified as an *N*-by-1 logical vector. *N* is the number of samples. Specify this input as `false` to simulate the loss of a GNSS receiver fix. When a GNSS receiver fix is lost, position measurements drift at a rate specified by the **Position error factor** parameter.

Dependencies

To enable this input port, select **Enable HasGNSSFix port**.

Data Types: `single` | `double`

Output

`Position` — Position of INS sensor

N-by-3 real-valued matrix

Position of the INS sensor relative to the navigation frame, in meters, returned as an *N*-by-3 real-valued matrix. *N* is the number of samples in the input.

Data Types: `single` | `double`

`Velocity` — Velocity of INS sensor

N-by-3 real-valued matrix

Velocity of the INS sensor relative to the navigation frame, in meters per second, returned as an *N*-by-3 real-valued matrix. *N* is the number of samples in the input.

Data Types: `single` | `double`

`Orientation` — Orientation of INS sensor

3-by-3-by-*N* real-valued array | *N*-by-4 real-valued matrix

Orientation of the INS sensor relative to the navigation frame, returned as one of the formats:

- A 3-by-3-by-*N* real-valued array, where each page of the array (3-by-3 matrix) is a rotation matrix.
- An *N*-by-4 real-valued matrix, where each row of the matrix is the four elements of a quaternion.
- An *N*-by-3 matrix of Euler angles, where each row of the matrix is the three Euler angles corresponding to the z-y-x rotation convention.

N is the number of samples in the input.

Data Types: `single` | `double`

`Acceleration` — Acceleration of INS sensor

N-by-3 real-valued matrix

Acceleration of the INS sensor relative to the navigation frame, in meters per second squared, returned as an *N*-by-3 real-valued matrix. *N* is the number of samples.

Dependencies

To enable this output port, select **Use acceleration and angular velocity**.

Data Types: `single` | `double`

`AngularVelocity` — Angular velocity of INS sensor

N-by-3 real-valued matrix

Angular velocity of the INS sensor relative to the navigation frame, in degrees per second, returned as an *N*-by-3 real-valued matrix. *N* is the number of samples.

Dependencies

To enable this output port, select **Use acceleration and angular velocity**.

Data Types: `single` | `double`

Parameters

`Mounting location (m)` — Location of sensor on platform (m)

`[0 0 0]` (default) | three-element real-valued vector of form [*x y z*]

Location of the sensor on the platform, in meters, specified as a three-element real-valued vector of the form [*x y z*]. The vector defines the offset of the sensor origin from the origin of the platform.

Data Types: `single` | `double`

`Roll (X-axis) accuracy (deg)` — Accuracy of roll measurement (deg)

`0.2` (default) | nonnegative real scalar

Accuracy of the roll measurement of the sensor body in degrees, specified as a nonnegative real scalar.

Roll is defined as rotation around the *x*-axis of the sensor body. Roll noise is modeled as white process noise with standard deviation equal to the specified **Roll accuracy** in degrees.

Data Types: `single` | `double`

`Pitch (Y-axis) accuracy (deg)` — Accuracy of pitch measurement (deg)
`0.2` (default) | nonnegative real scalar

Accuracy of the pitch measurement of the sensor body in degrees, specified as a nonnegative real scalar.

Pitch is defined as rotation around the *y*-axis of the sensor body. Pitch noise is modeled as white process noise with standard deviation equal to the specified **Pitch accuracy** in degrees.

Data Types: `single` | `double`

`Yaw (Z-axis) accuracy (deg)` — Accuracy of yaw measurement (deg)
`1` (default) | nonnegative real scalar

Accuracy of the yaw measurement of the sensor body in degrees, specified as a nonnegative real scalar.

Yaw is defined as rotation around the *z*-axis of the sensor body. Yaw noise is modeled as white process noise with standard deviation equal to the specified **Yaw accuracy** in degrees.

Data Types: `single` | `double`

`Position accuracy (m)` — Accuracy of position measurement (m)
`1` (default) | nonnegative real scalar | 1-by-3 vector of nonnegative values

Accuracy of the position measurement of the sensor body in meters, specified as a nonnegative real scalar or a 1-by-3 vector of nonnegative values. If you specify the parameter as a scalar value, then the block sets the accuracy of all three position components to this value.

Position noise is modeled as white process noise with a standard deviation equal to the specified **Position accuracy** in meters.

Data Types: `single` | `double`

`Velocity accuracy (m/s)` — Accuracy of velocity measurement (m/s)
`1` (default) | nonnegative real scalar

Accuracy of the velocity measurement of the sensor body in meters per second, specified as a nonnegative real scalar.

Velocity noise is modeled as white process noise with a standard deviation equal to the specified **Velocity accuracy** in meters per second.

Data Types: `single` | `double`

`Use acceleration and angular velocity` — Use acceleration and angular velocity
off (default) | on

Select this check box to enable the block inputs of acceleration and angular velocity through the **Acceleration** and **AngularVelocity** input ports, respectively. Meanwhile, the block outputs the acceleration and angular velocity measurements through the **Acceleration** and **AngularVelocity** output ports, respectively. Additionally, selecting this parameter enables you to specify the **Acceleration accuracy** and **Angular velocity accuracy** parameters.

`Acceleration accuracy (m/s²)` — Accuracy of acceleration measurement (m/s²)

`0` (default) | nonnegative real scalar

Accuracy of the acceleration measurement of the sensor body in meters, specified as a nonnegative real scalar.

Acceleration noise is modeled as white process noise with a standard deviation equal to the specified **Acceleration accuracy** in meters per second squared.

Dependencies

To enable this parameter, select **Use acceleration and angular velocity**.

Data Types: `single` | `double`

`Angular velocity accuracy (deg/s)` — Accuracy of angular velocity measurement (deg/s)

`0` (default) | nonnegative real scalar

Accuracy of the angular velocity measurement of the sensor body in meters, specified as a nonnegative real scalar.

Angular velocity noise is modeled as white process noise with a standard deviation equal to the specified **Angular velocity accuracy** in degrees per second.

Dependencies

To enable this parameter, select **Use acceleration and angular velocity**.

Data Types: `single` | `double`

`Enable HasGNSSFix port` — Enable `HasGNSSFix` input port

off (default) | on

Select this check box to enable the **HasGNSSFix** input port. When the **HasGNSSFix** input is specified as `false`, position measurements drift at a rate specified by the **Position error factor** parameter.

`Position error factor (m)` — Position error factor (m)

`[0 0 0]` (default) | nonnegative scalar | 1-by-3 real-valued vector

Position error factor without GNSS fix, specified as a scalar or a 1-by-3 real-valued vector. If you specify the parameter as a scalar value, then the block sets the position error factors of all three position components to this value.

When the **HasGNSSFix** input is specified as `false`, the position error grows at a quadratic rate due to constant bias in the accelerometer. The position error for a position component $E(t)$ can be expressed as $E(t) = 1/2\alpha t^2$, where α is the position error factor for the corresponding component and t is the time since the GNSS fix is lost. The computed $E(t)$ values for the x, y, and z components are added to the corresponding position components of the **Position** output.

Dependencies

To enable this parameter, select **Enable HasGNSSFix port**.

Data Types: `double`

`Intial Seed` — Initial seed for randomization
67 (default) | nonnegative integer

Initial seed of a random number generator algorithm, specified as a nonnegative integer.

Data Types: `single` | `double`

`Simulate using` — Type of simulation to run
`Interpreted Execution` (default) | `Code Generation`

- `Interpreted execution` — Simulate the model using the MATLAB interpreter. This option shortens startup time. In `Interpreted execution` mode, you can debug the source code of the block.
- `Code generation` — Simulate the model using generated C code. The first time that you run a simulation, Simulink generates C code for the block. The C code is reused for subsequent simulations if the model does not change. This option requires additional startup time.

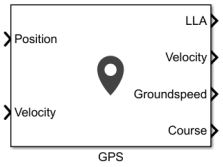
See Also
IMU | `insSensor`

Introduced in R2020b

GPS

Simulate GPS sensor readings with noise

Library: UAV Toolbox / UAV Scenario and Sensor Modeling
Navigation Toolbox / Multisensor Positioning / Sensor Models
Sensor Fusion and Tracking Toolbox / Multisensor Positioning / Sensor Models



Description

The block outputs noise-corrupted GPS measurements based on the input position and velocity in the local coordinate frame or geodetic frame. It uses the WGS84 earth model to convert local coordinates to latitude-longitude-altitude LLA coordinates.

Ports

Input

`Position` — Position of GPS receiver in navigation coordinate system

matrix

Specify the input position of the GPS receiver in the navigation coordinate system as a real, finite *N*-by-3 matrix. *N* is the number of samples in the current frame. The format of the matrix rows differs depending on the value of the **Position input format** parameter.

- If the value of the **Position input format** parameter is `Local`, specify each row of the **Position** as Cartesian coordinates in meters with respect to the local navigation reference frame, specified by the **Reference frame** parameter, with the origin specified by the **Reference location** parameter.
- If the value of the **Position input format** parameter is `Geodetic`, specify each row of the **Position** input as geodetic coordinates of the form `[latitude longitude altitude]`. The values of `latitude` and `longitude` are in degrees. `Altitude` is the height above the WGS84 ellipsoid model in meters.

Data Types: `single` | `double`

`Velocity` — Velocity in local navigation coordinate system (m/s)

matrix

Specify the input velocity of the GPS receiver in the navigation coordinate system in meters per second as a real, finite *N*-by-3 matrix. *N* is the number of samples in the current frame. The format of the matrix rows differs depending on the value of the **Position input format** parameter.

- If the value of the **Position input format** parameter is `Local`, specify each row of the **Velocity** with respect to the local navigation reference frame (NED or ENU), specified by the **Reference frame** parameter, with the origin specified by the **Reference location** parameter.
- If the value of the **Position input format** parameter is `Geodetic`, specify each row of the **Velocity** with respect to the navigation reference frame (NED or ENU), specified by the **Reference frame** parameter, with the origin specified by **Position**.

Data Types: `single` | `double`

Output

`LLA` — Position in LLA coordinate system

matrix

Position of the GPS receiver in the geodetic latitude, longitude, and altitude (LLA) coordinate system, returned as a real, finite *N*-by-3 array. Latitude and longitude are in degrees with North and East being positive. Altitude is in meters.

N is the number of samples in the current frame.

Data Types: `single` | `double`

`Velocity` — Velocity in local navigation coordinate system (m/s)

matrix

Velocity of the GPS receiver in the local navigation coordinate system in meters per second, returned as a real, finite *N*-by-3 matrix. *N* is the number of samples in the current frame. The format of the matrix rows differs depending on the value of the **Position input format** parameter.

- If the value of the **Position input format** parameter is `Local`, the **Velocity** output is with respect to the local navigation reference frame (NED or ENU), specified by the **Reference frame** parameter, with the origin specified by the **Reference location** parameter.
- If the value of the **Position input format** parameter is `Geodetic`, the **Velocity** output is with respect to the navigation reference frame (NED or ENU), specified by the **Reference frame** parameter, with the origin specified by **LLA**.

Data Types: `single` | `double`

`Groundspeed` — Magnitude of horizontal velocity in local navigation coordinate system (m/s)

vector

Magnitude of the horizontal velocity of the GPS receiver in the local navigation coordinate system in meters per second, returned as a real, finite *N*-element column vector.

N is the number of samples in the current frame.

Data Types: `single` | `double`

`Course` — Direction of horizontal velocity in local navigation coordinate system (°)

vector

Direction of the horizontal velocity of the GPS receiver in the local navigation coordinate system, in degrees, returned as a real, finite *N*-element column vector of values from 0 to 360. North corresponds to 0 degrees and East corresponds to 90 degrees.

N is the number of samples in the current frame.

Data Types: `single` | `double`

Parameters

`Reference frame` — Reference frame

`NED` (default) | `ENU`

Specify the reference frame as `NED` (North-East-Down) or `ENU`(East-North-Up).

`Position input format` — Position coordinate input format

`Local` (default) | `Geodetic`

Specify the position coordinate input format as `Local` or `Geodetic`.

- If you set this parameter to `Local`, then the input to the **Position** port must be in the form of Cartesian coordinates with respect to the local navigation frame, specified by the **Reference Frame** parameter, with the origin fixed and defined by the **Reference location** parameter. The input to the **Velocity** input port must also be with respect to this local navigation frame.
- If you set this parameter to `Geodetic`, then the input to the **Position** port must be geodetic coordinates in `[latitude longitude altitude]`. The input to the **Velocity** input port must also be with respect to the navigation frame specified by the **Reference frame** parameter, with the origin corresponding to the **Position** port.

`Reference location` — Origin of local navigation reference frame

`[0,0,0]` (default) | three-element vector

Specify the origin of the local reference frame as a three-element row vector in geodetic coordinates `[latitude longitude altitude]`, where `altitude` is the height above the reference ellipsoid model WGS84. The reference location values are in degrees, degrees, and meters, respectively. The degree format is decimal degrees (DD).

Dependencies

To enable this parameter, set the **Position input format** parameter to `Local`.

`Horizontal position accuracy` — Horizontal position accuracy (m)

`1.6` (default) | nonnegative real scaler

Specify horizontal position accuracy as a nonnegative real scalar in meters. The horizontal position accuracy specifies the standard deviation of the noise in the horizontal position measurement. Increasing this value adds noise to the measurement, decreasing its accuracy.

Tunable: Yes

`Vertical position accuracy` — Vertical position accuracy (m)

`3` (default) | nonnegative real scaler

Specify vertical position accuracy as a nonnegative real scalar in meters. The vertical position accuracy specifies the standard deviation of the noise in the vertical position measurement. Increasing this value adds noise to the measurement, decreasing its accuracy.

Tunable: Yes

`Velocity accuracy` — Velocity accuracy (m/s)
`0.1` (default) | nonnegative real scalar

Specify velocity accuracy per second as a nonnegative real scalar in meters. The velocity accuracy specifies the standard deviation of the noise in the velocity measurement. Increasing this value adds noise to the measurement, decreasing its accuracy.

Tunable: Yes

`Decay factor` — Global position noise decay factor
`0.999` (default) | scalar in range [0, 1]

Specify the global position noise decay factor as a numeric scalar in the range [0, 1]. A decay factor of `0` models the global position noise as a white noise process. A decay factor of 1 models the global position noise as a random walk process.

Tunable: Yes

Seed — Initial seed
67 (default) | nonnegative integer

Specify the initial seed of an mt19937ar random number generator algorithm as a nonnegative integer.

`Simulate using` — Type of simulation to run
`Interpreted execution` (default) | `Code generation`

Select the type of simulation to run from these options:

- `Interpreted execution` — Simulate the model using the MATLAB interpreter. For more information, see “Simulation Modes”.
- `Code generation` — Simulate the model using generated C code. The first time you run a simulation, Simulink generates C code for the block. The C code is reused for subsequent simulations, as long as the model does not change.

See Also

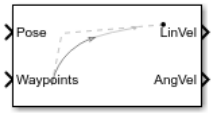
Objects
`gpsSensor`

Introduced in R2021b

Pure Pursuit

Linear and angular velocity control commands

Library: Robotics System Toolbox / Mobile Robot Algorithms
Navigation Toolbox / Control Algorithms



Description

The Pure Pursuit block computes linear and angular velocity commands for following a path using a set of waypoints and the current pose of a differential drive vehicle. The block takes updated poses to update velocity commands for the vehicle to follow a path along a desired set of waypoints. Use the **Max angular velocity** and **Desired linear velocity** parameters to update the velocities based on the performance of the vehicle.

The **Lookahead distance** parameter computes a look-ahead point on the path, which is an instantaneous local goal for the vehicle. The angular velocity command is computed based on this point. Changing **Lookahead distance** has a significant impact on the performance of the algorithm. A higher look-ahead distance results in a smoother trajectory for the vehicle, but can cause the vehicle to cut corners along the path. Too low of a look-ahead distance can result in oscillations in tracking the path, causing unstable behavior. For more information on the pure pursuit algorithm, see “Pure Pursuit Controller”.

Input/Output Ports

Input

`Pose` — Current vehicle pose

`[x y theta]` vector

Current vehicle pose, specified as an `[x y theta]` vector, which corresponds to the *x*-*y* position and orientation angle, *theta*. Positive angles are measured counterclockwise from the positive *x*-axis.

`Waypoints` — Waypoints

[ ] (default) | *n*-by-2 array

Waypoints, specified as an *n*-by-2 array of `[x y]` pairs, where *n* is the number of waypoints. You can generate the waypoints using path planners like `mobileRobotPRM` or specify them as an array in Simulink.

Output

`LinVel` — Linear velocity

scalar in meters per second

Linear velocity, specified as a scalar in meters per second.

Data Types: `double`

`AngVel` — Angular velocity

scalar in radians per second

Angular velocity, specified as a scalar in radians per second.

Data Types: `double`

`TargetDir` — Target direction for vehicle

scalar in radians

Target direction for the vehicle, specified as a scalar in radians. The forward direction of the vehicle is considered zero radians, with positive angles measured counterclockwise. This output can be used as the input to the **TargetDir** port for the Vector Field Histogram block.

Dependencies

To enable this port, select the **Show TargetDir output port** parameter.

Parameters

`Desired linear velocity (m/s)` — Linear velocity

`0.1` (default) | scalar

Desired linear velocity, specified as a scalar in meters per second. The controller assumes that the vehicle drives at a constant linear velocity and that the computed angular velocity is independent of the linear velocity.

`Maximum angular velocity (rad/s)` — Angular velocity

`1.0` (default) | scalar

Maximum angular velocity, specified as a scalar in radians per second. The controller saturates the absolute angular velocity output at the given value.

`Lookahead distance (m)` — Look-ahead distance

`1.0` (default) | scalar

Look-ahead distance, specified as a scalar in meters. The look-ahead distance changes the response of the controller. A vehicle with a higher look-ahead distance produces smooth paths but takes larger turns at corners. A vehicle with a smaller look-ahead distance follows the path closely and takes sharp turns, but oscillate along the path. For more information on the effects of look-ahead distance, see “Pure Pursuit Controller”.

`Show TargetDir output port` — Target direction indicator

off (default) | on

Select this parameter to enable the **TargetDir** out port. This port gives the target direction as an angle in radians from the forward position, with positive angles measured counterclockwise.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

Blocks
Vector Field Histogram

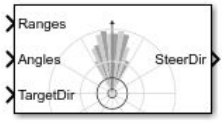
Classes
`binaryOccupancyMap` | `occupancyMap` | `binaryOccupancyMap` | `occupancyMap` | `controllerVFH`

Topics
“Plan Path for a Differential Drive Robot in Simulink” (Robotics System Toolbox)
“Path Following with Obstacle Avoidance in Simulink®”
“Pure Pursuit Controller”

Introduced in R2019b

Vector Field Histogram

Avoid obstacles using vector field histogram
Library: Navigation Toolbox / Control Algorithms



Description

The Vector Field Histogram (VFH) block enables your vehicle to avoid obstacles based on range sensor data. Given a range sensor reading in terms of ranges and angles, and a target direction to drive toward, the VFH controller computes an obstacle-free steering direction.

For more information on the algorithm details, see “Vector Field Histogram” on page 4-51 under Algorithms.

Limitations

- The `Ranges` and `Angles` inputs are limited to 4000 elements when generating code for models that use this block.

Input/Output Ports

Input

`Ranges` — Range values from scan data

vector of scalars

Range values from scan data, specified as a vector of scalars in meters. These range values are distances from a sensor at specified angles. The vector must be the same length as the corresponding **Angles** vector.

`Angles` — Angle values from scan data

vector of scalars

Angle values from scan data, specified as a vector of scalars in radians. These angle values are the specific angles of the specified ranges. The vector must be the same length as the corresponding **Ranges** vector.

`TargetDir` — Target direction for vehicle

scalar

Target direction for the vehicle, specified as a scalar in radians. The forward direction of the vehicle is considered zero radians, with positive angles measured counterclockwise. You can use the **TargetDir** output from the Pure Pursuit block when generating controls from a set of waypoints.

Output

`steeringDir` — Steering direction for vehicle
scalar

Steering direction for the vehicle, specified as a scalar in radians. This obstacle-free direction is calculated based on the VFH+ algorithm. The forward direction of the vehicle is considered zero radians, with positive angles measured counterclockwise.

Parameters

Main

`Number of angular sectors` — Number of bins used to create the histograms
`180` (default) | scalar

Number of bins used to create the histograms, specified as a scalar. This parameter is nontunable. You can set this parameter only when the object is initialized.

`Range distance limits (m)` — Limits for range readings
`[0.05 2]` (default) | two-element vector of scalars

Limits for range readings in meters, specified as a two-element vector of scalars. The range readings input are only considered if they fall within the distance limits. Use the lower distance limit to ignore false positives from poor sensor performance at lower ranges. Use the upper limit to ignore obstacles that are too far away from the vehicle.

`Histogram thresholds` — Thresholds for computing binary histogram
`[3 10]` (default) | two-element vector of scalars

Thresholds for computing binary histogram, specified as a two-element vector of scalars. The algorithm uses these thresholds to compute the binary histogram from the polar obstacle density. Polar obstacle density values higher than the upper threshold are represented as occupied space (1) in the binary histogram. Values smaller than the lower threshold are represented as free space (0). Values that fall between the limits are set to the values of a previous computed binary histogram if one exists from previous iterations. If a previous histogram does not exist, the value is set as free space (0).

`vehicle radius (m)` — Radius of the vehicle
`0.1` (default) | scalar

Radius of the vehicle, specified as a scalar in meters. This dimension defines the smallest circle that can circumscribe your vehicle. The vehicle radius is used to account for vehicle size when computing the obstacle-free direction.

`Safety distance (m)` — Safety distance around the vehicle
`0.1` (default) | scalar

Safety distance left around the vehicle position in addiction to **vehicle radius**, specified as a scalar in meters. The vehicle radius and safety distance are used to compute the obstacle-free direction.

`Minimum turning radius (m)` — Minimum turning radius at current speed
`0.1` (default) | scalar

Minimum turning radius for the vehicle moving at its current speed, specified as a scalar in meters.

`Simulate using` — Specify type of simulation to run
`Code generation` (default) | `Interpreted execution`

- `Code generation` — Simulate model using generated C code. The first time you run a simulation, Simulink generates C code for the block. The C code is reused for subsequent simulations, as long as the model does not change. This option requires additional startup time but the speed of the subsequent simulations is comparable to `Interpreted execution`.
- `Interpreted execution` — Simulate model using the MATLAB interpreter. This option shortens startup time but has a slower simulation speed than `Code generation`. In this mode, you can debug the source code of the block.

Tunable: No

Cost Function Weights

`Target direction weight` — Cost function weight for target direction
5 (default) | scalar

Cost function weight for moving toward the target direction, specified as a scalar. To follow a target direction, set this weight to be higher than the sum of **Current direction weight** and **Previous direction weight**. To ignore the target direction cost, set this weight to `0`.

`Current direction weight` — Cost function weight for current direction
2 (default) | scalar

Cost function weight for moving the vehicle in the current heading direction, specified as a scalar. Higher values of this weight produce efficient paths. To ignore the current direction cost, set this weight to `0`.

`Previous direction weight` — Cost function weight for previous direction
2 (default) | scalar

Cost function weight for moving in the previously selected steering direction, specified as a scalar. Higher values of this weight produce smoother paths. To ignore the previous direction cost, set this weight to `0`.

Algorithms

Vector Field Histogram

The block uses the VFH+ algorithm to compute the obstacle-free direction. First, the algorithm takes the ranges and angles from range sensor data and builds a polar histogram for obstacle locations. Then, it uses the input histogram thresholds to calculate a binary histogram that indicates occupied and free directions. Finally, the algorithm computes a masked histogram, which is computed from the binary histogram based on the minimum turning radius of the vehicle.

The algorithm selects multiple steering directions based on the open space and possible driving directions. A cost function, with weights corresponding to the previous, current, and target directions, calculates the cost of different possible directions. The algorithm then returns an obstacle-free direction with minimal cost. Using the obstacle-free direction, you can input commands to move your vehicle in that direction.

To use this block for your own application and environment, you must tune the algorithm parameters. Parameter values depend on the type of vehicle, the range sensor, and the hardware you use. For more information on the VFH algorithm, see `controllerVFH`.

See Also

Blocks
Pure Pursuit | Publish | Subscribe

Classes
`controllerVFH`

Topics
“Vector Field Histogram”

Introduced in R2019b

Apps

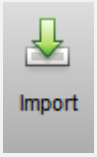
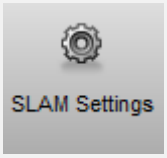
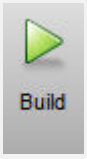
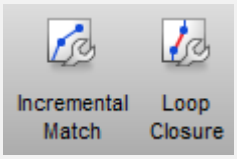
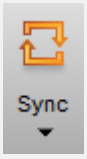
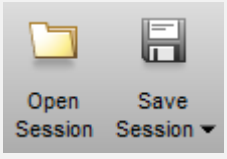
SLAM Map Builder

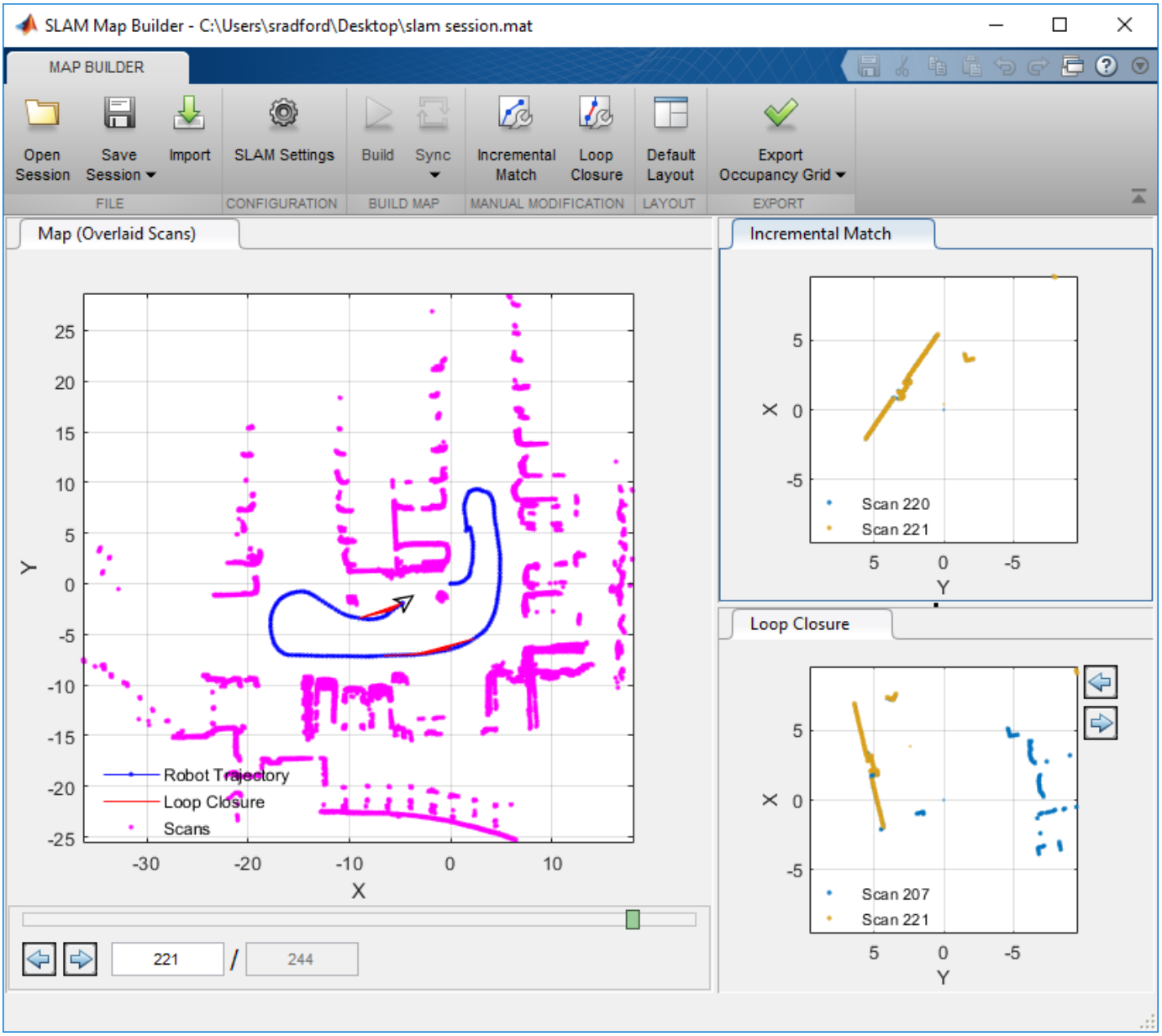
Build 2-D grid maps using lidar-based SLAM

Description

The **SLAM Map Builder** app loads recorded lidar scans and odometry sensor data to build a 2-D occupancy grid using simultaneous localization and mapping (SLAM) algorithms. Incremental scan matching aligns and overlays scans to build the map. Loop closure detection adjusts for drift of the vehicle odometry by detecting previously visited locations and adjusting the overall map. Sometimes, the scan matching algorithm and loop closure detection require manual adjustment. Use the app to manually align scans and modify loop closures to improve the overall map accuracy. You can also tune the SLAM algorithm settings to improve the automatic map building.

To use the app:

	To load rosbag log files, select **Import** > **Import from rosbag**. Select the rosbag file and click **Open**. This opens the **Import** tab. For more information, see Import and Filter a rosbag on page 5-13. To load data from the workspace, **Import** > **Import from workspace**. Select your **Scans** and **Poses** variables using the drop downs provided. You can also specify the variables in the `slamMapBuilder` function. See Programmatic Use on page 5-12.
	Use **SLAM Settings** to adjust the SLAM algorithm settings. Default values are provided, but your specific sensors and data may require tuning of these settings. The most important value to tune is the **Loop Closure Threshold**. For more information, see Tune SLAM Settings on page 5-13.
	Click **Build** to begin the SLAM map building process. The building process aligns scans in the map using incremental scan matching, identifies loop closures when visiting previous locations, and adjusts poses. Click **Pause** at any time during the map building process to manually align incremental scans or modify loop closures.
	Click **Incremental Match** to modify the relative pose of the currently selected frame and align the scan with the previous scan. Click **Loop Closure** to modify or ignore the detected loop closure for the current frame. Use the slider on the bottom to scroll back to areas where scan matching or loop closures are not accurate. You can modify any number of scans or loop closures. For more information, see Modify Increment Scans and Loop Closures on page 5-14.
	After modifying your map, click **Sync** to update all the poses in the scan map. The two options under **Sync** are **Sync**, which searches for new loop closures, or **Sync Fast**, which skips loop closure searching and just updates the scan map. For more information, see Sync the Map on page 5-15.
	When you are satisfied with how the map looks, click **Export to OccupancyGrid** to either export the map to an m-file or save the map in the workspace. The map is output as a 2-D probabilistic occupancy grid in an `occupancyMap` object.
	You can open existing app sessions you have saved using **Open Session**. When you are in the **Map Builder** tab, you can save your progress to an m-file using **Save Session**.



Open the SLAM Map Builder App

- MATLAB Toolstrip: On the **Apps** tab, under **Control System Design and Analysis**, click  **SLAM Map Builder**.
- MATLAB Command Window: Enter `slamMapBuilder`

Examples

Build and Tune a Map Using Lidar Scans with SLAM

The **SLAM Map Builder** app helps you build an occupancy grid from lidar scans using simultaneous localization and mapping (SLAM) algorithms. The map is built by estimating poses through scan matching and using loop closures for pose graph optimization. This example shows you the workflow for loading a rosbag of lidar scan data, filtering the data, and building the map. Tune the scan map by adjusting incremental scan matches and modifying loop closures.

Load Lidar Scan Data

Load the example `.mat` file into the workspace, which contains a variable, `scans`, as a cell array of `lidarScan` objects.

```
load slamLidarScans.mat
```

Open the App

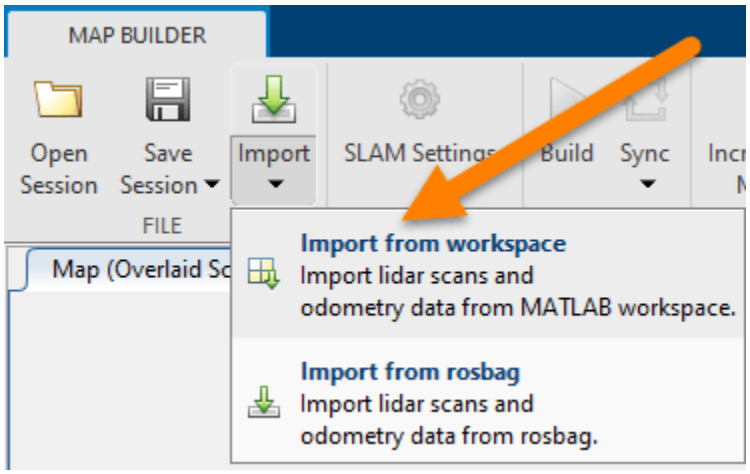
In the **Apps** tab, under **Control System Design and Analysis,** click **SLAM Map Builder**.

Also, you can call the `slamMapBuilder` function:

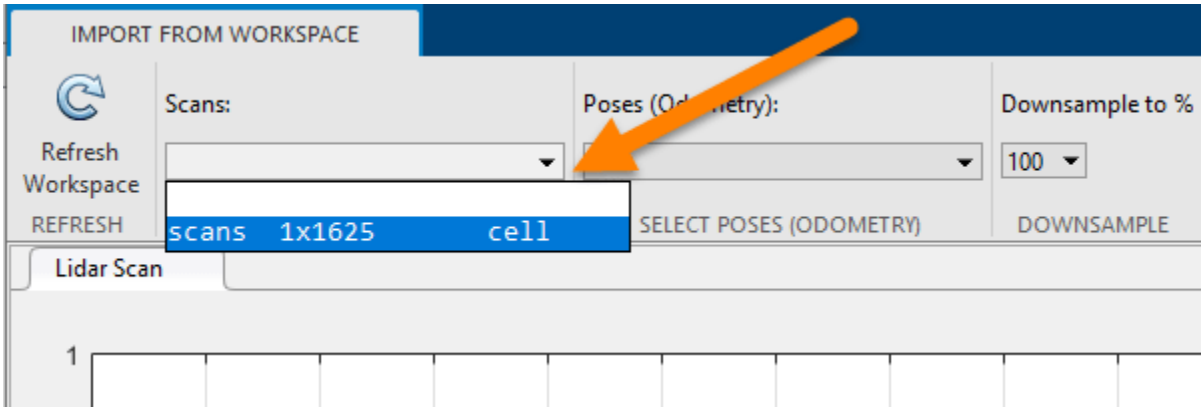
```
slamMapBuilder
```

Import Lidar Scans

Click **Import > Import from workspace** to load the scans. Data stored as a rosbag can be loaded with a ROS Toolbox license.



Select the `scans` variable in the **Scans** drop down.



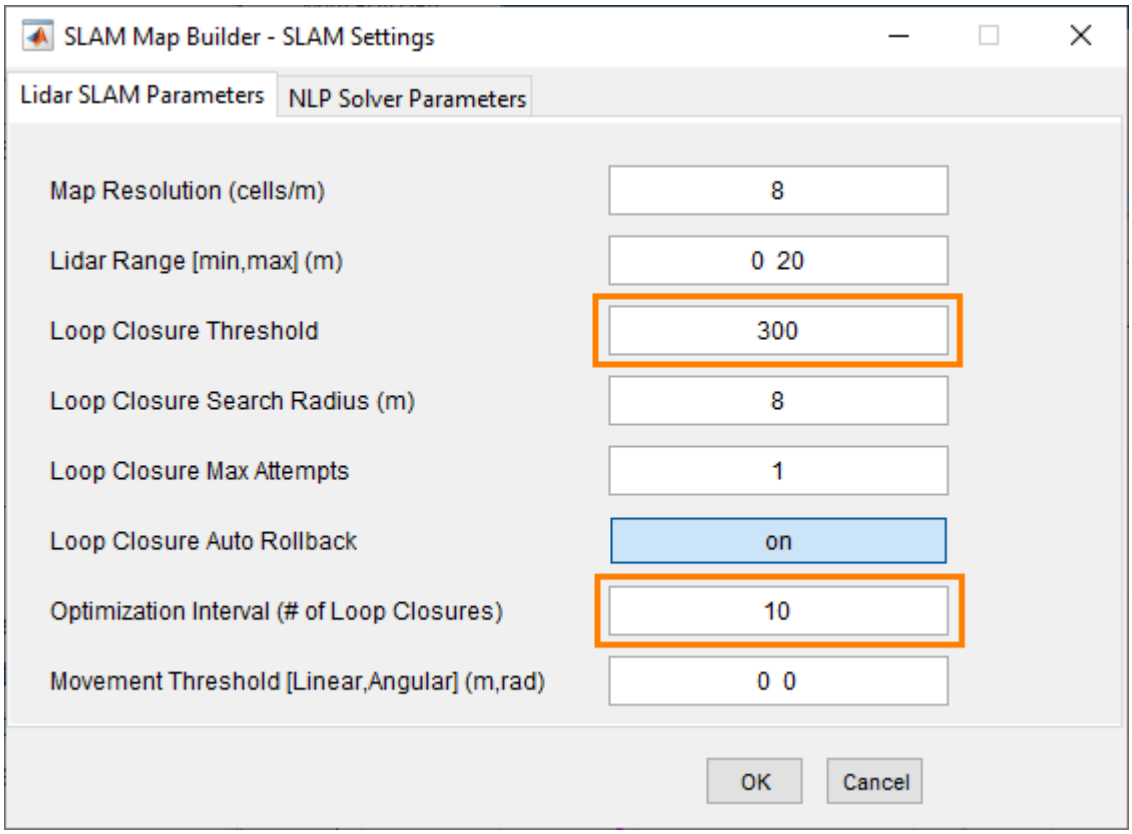
In the tool bar, set **Downsample to (%)** to 10. Downsampling evenly samples from the data to reduce computation time for the SLAM algorithm. For this example, 10% is every 5th scan. Click **Apply**.

Use the slider or arrow keys at the bottom to preview the scans.

Once you are done importing, click **Close.**

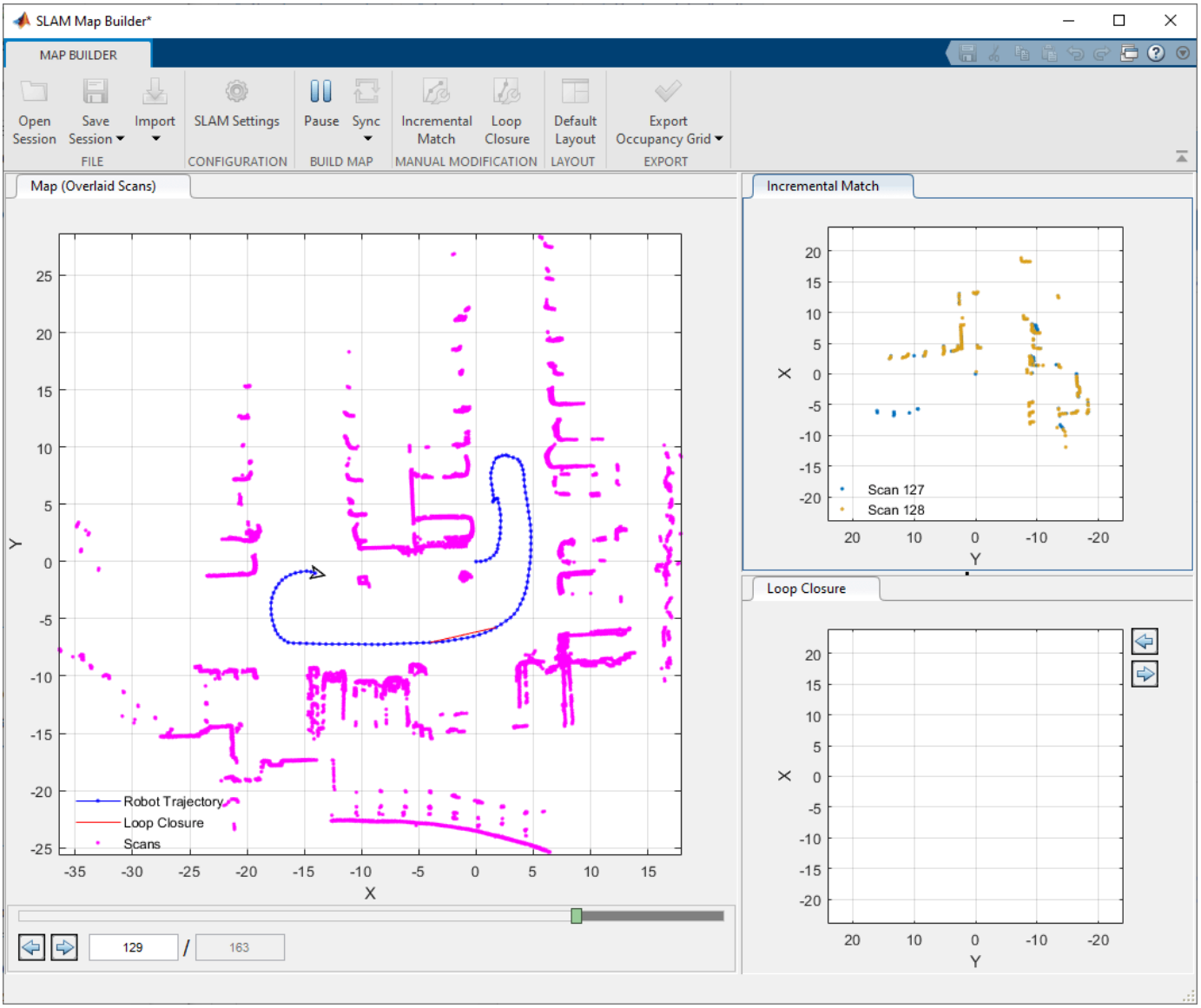
Tune SLAM Settings

The SLAM algorithm can be tuned using the **SLAM Settings** dialog. The parameters should be adjusted based on your sensor specifications, the environment, and your application. For this example, increase **Loop Closure Threshold** from `200` to `300`. This increased threshold decreases the likelihood of accepting and using a detected loop closure. Set the **Optimization Interval** to `10`. With every 10th loop closure accepted, the pose graph is optimized to account for drift.



Build the Map

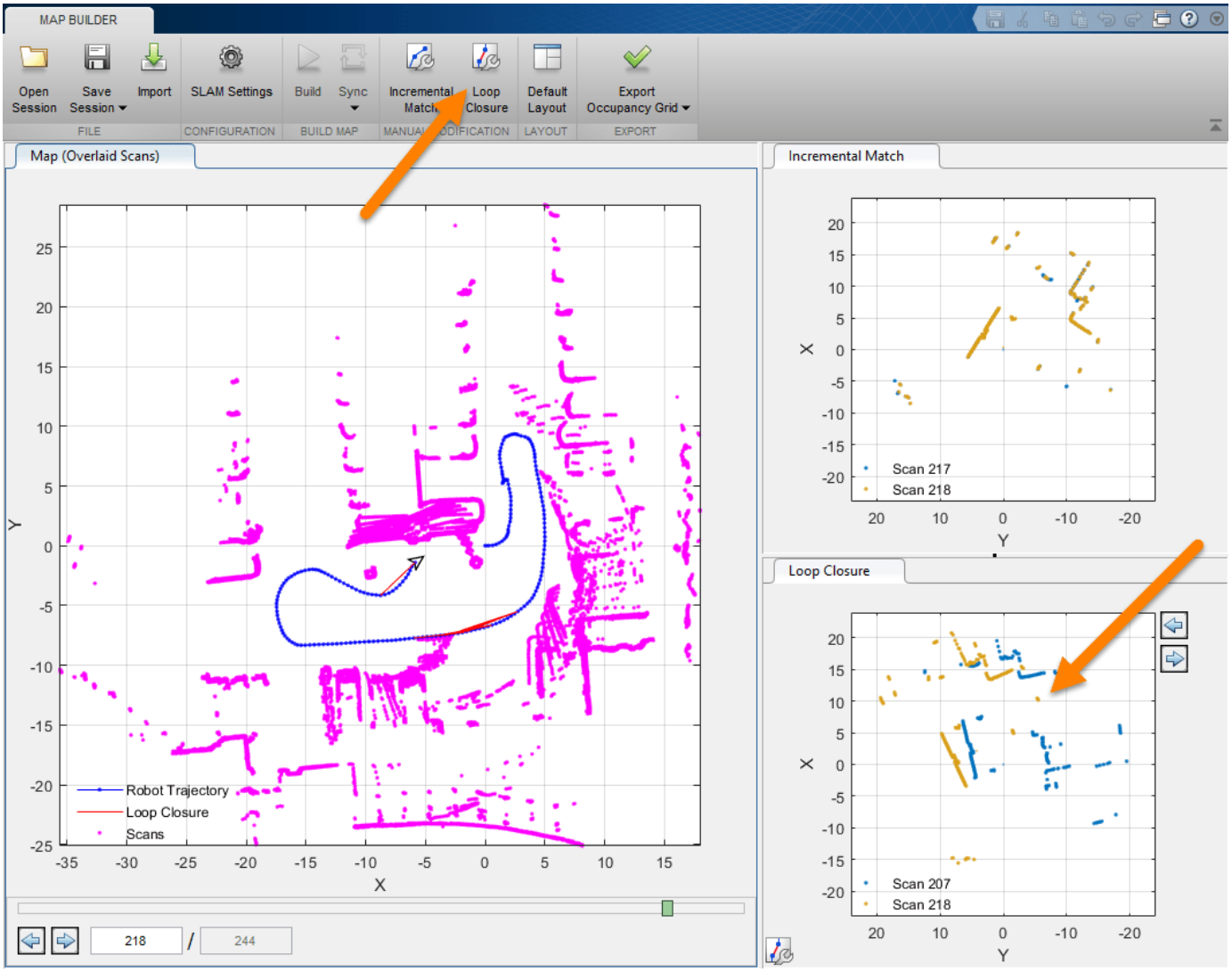
After filtering your data and setting the SLAM algorithm settings, click **Build**. The app begins processing scans to build the map. You should see the slider progressing and scans being overlaid in the map. The estimated robot trajectory is plotted on the same scan map. Incremental scan matches are shown in the **Incremental Match** pane. Whenever a loop closure is detected, the **Loop Closure** pane shows the two scans overlaid on each other.



Adjust Scan Matches or Loop Closures

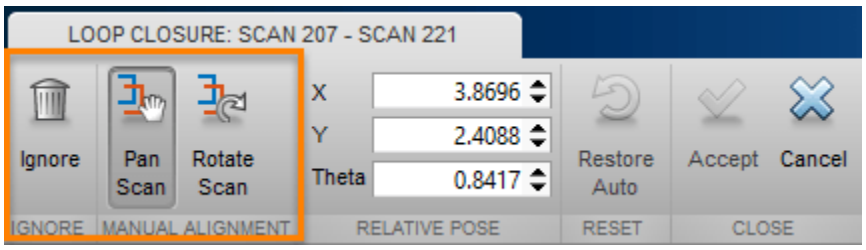
At any time during the build process, if you notice the map is distorted or an incremental match or loop closure looks off, click **Pause** to select scans for adjustment. You can modify scans at the end of the build process as well. Navigate using the arrow keys or slider to the point in the file where the distortion first occurs. Click the **Incremental Match** or **Loop Closure** buttons to adjust the currently displayed scan poses. In this section, the bad loop closure is artificial and only for illustration purposes.

Click the **Loop Closure** button. This opens a tab for modifying the loop closure relative pose.

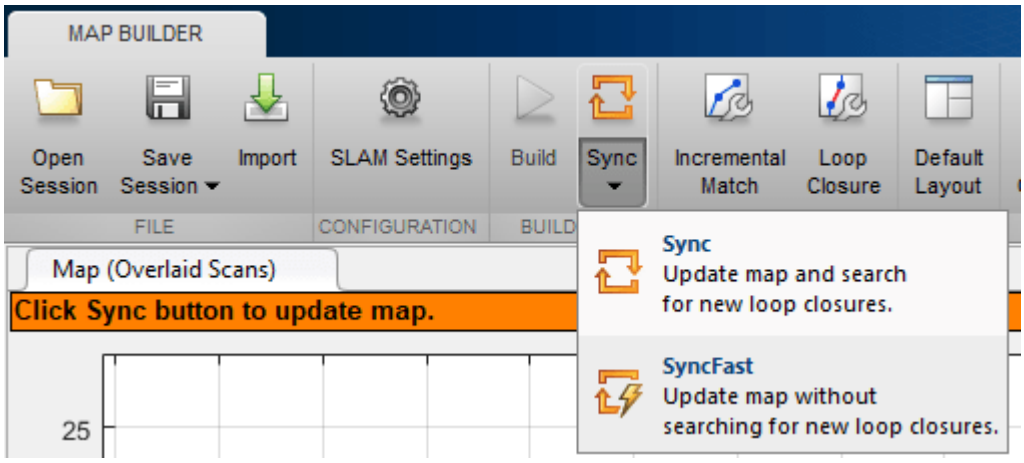


To ignore the loop closure completely, click **Ignore**. Otherwise, manually modify the relative scan pose until the scans line up.

Click **Pan Scan** or **Rotate Scan**, then click and drag in the figure to align the two scans. Click **Accept** when you are done. You can do this for multiple scans.

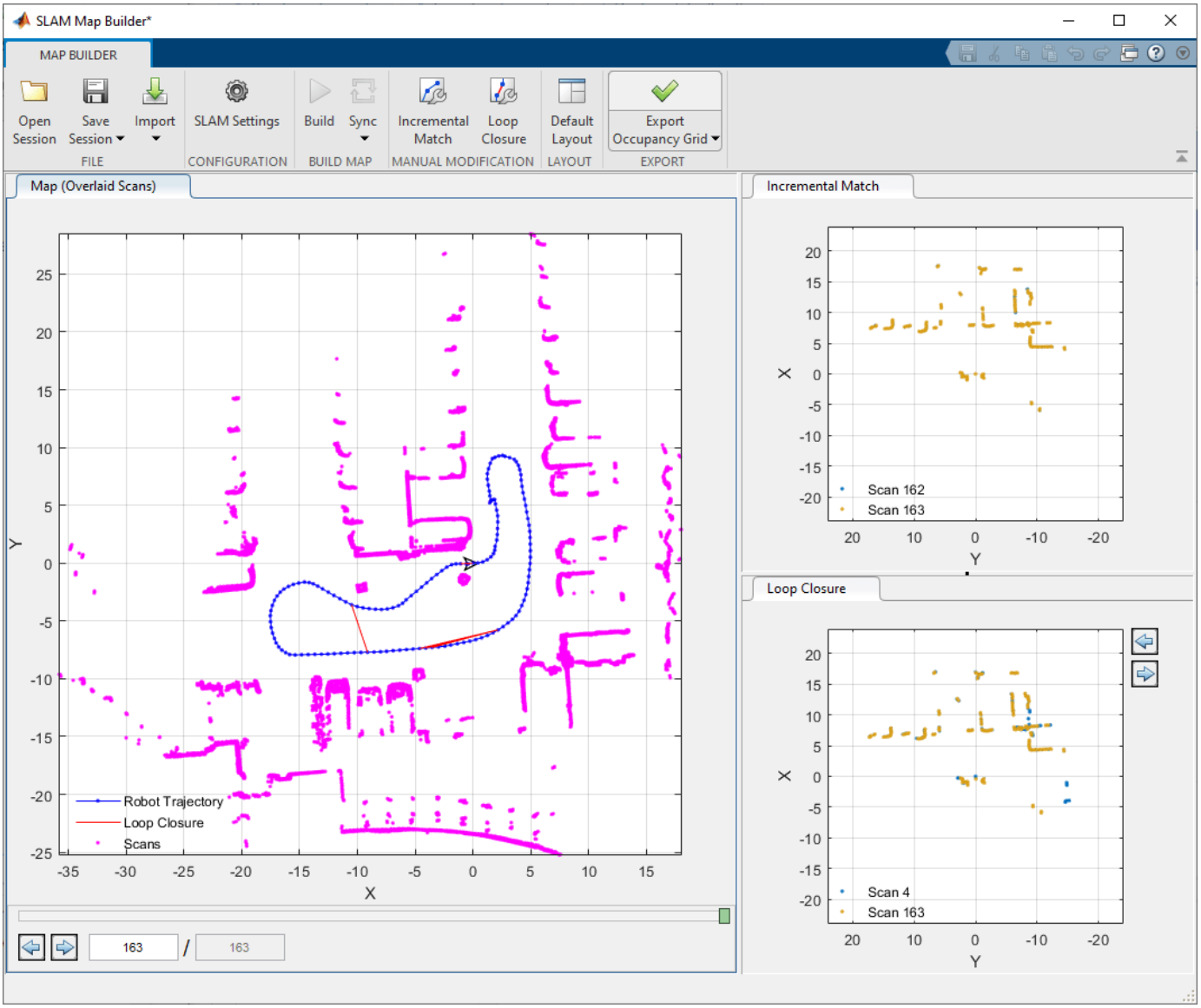


After you modify your scan poses for incremental matches and loop closures, click **Sync** to apply changes. **SyncFast** updates the map without searching for new loop closures and reduces computation time if you have already processed all the scans.



Export Occupancy Grid

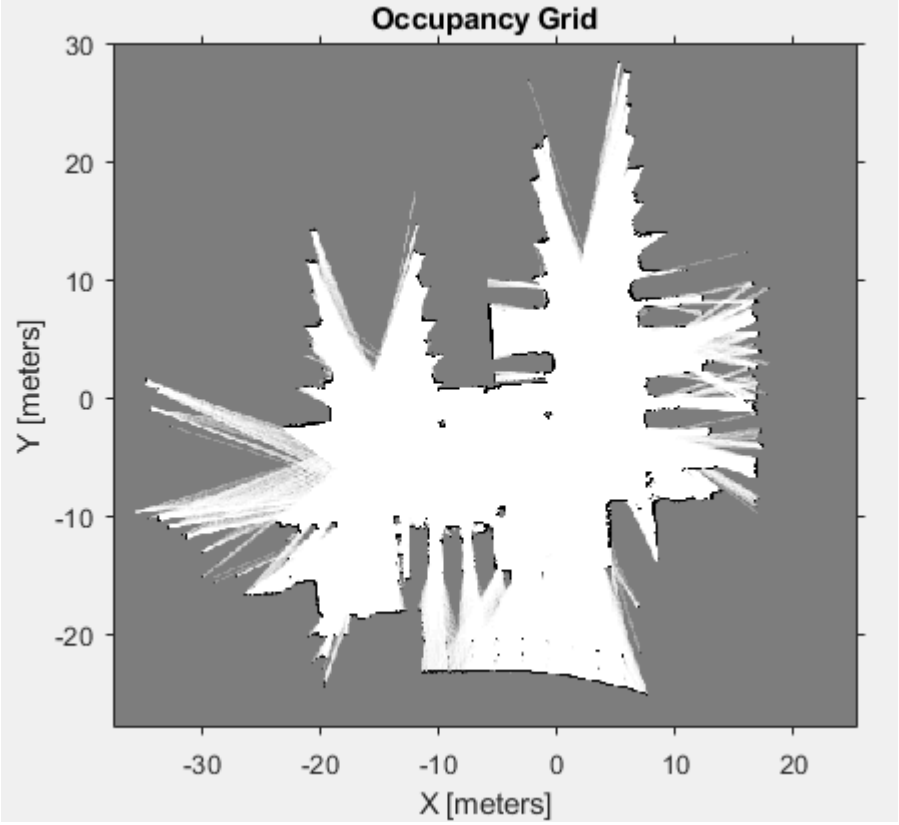
Once you have synced your changes and finished building the map, you should see a fully overlaid scan map with a robot trajectory.



Click **Export Occupancy Grid** to get a final occupancy map of your environment as a `occupancyMap` object. Specify the variable name to export the map to the workspace. You can create a map from a subset of scans by scrolling back to the desired frame before exporting and selecting **Up to currently selected scan**.

Call `show` on the stored map to visualize the occupancy map.

```
show(myOccMap)
```



You can also save a SLAM Map Builder app session using the **Save Session** button. The app writes the current state of the app to a `.mat` file that can be loaded later using **Open Session**.

- "Implement Simultaneous Localization And Mapping (SLAM) with Lidar Scans"
- "Implement Online Simultaneous Localization And Mapping (SLAM) with Lidar Scans"

Programmatic Use

`slamMapBuilder(bag)` opens the **SLAM Map Builder** app and imports the rosbag log file specified in `bag`, a `BagSelection` object created using the `rosbag` function. The app opens to the **Import** tab to filter the sensor data in your rosbag.

`slamMapBuilder(sessionFile)` opens the **SLAM Map Builder** app from a saved session file name, `sessionFile`. An app session file is created through the **Save Session** button in the app toolstrip.

`slamMapBuilder(scans)` opens the **SLAM Map Builder** app and imports the scans specified in `scans`, a cell array of `lidarScan` objects. The app assumes you have prefiltered your scans and skips the import process. Click **Build** to start building the map.

`slamMapBuilder(scans,poses)` opens the **SLAM Map Builder** app and imports the scans and poses. `scans` is specified as a cell array of `lidarScan` objects. `poses` is a matrix of `[x y theta]` vectors that correspond to the poses of `scans`. The app assumes you have prefiltered your scans and skips the import process. Click **Build** to start building the map.

More About

Import and Filter a rosbag

When you click the **Import** button, specify the parameters for your rosbag and how you want to filter the data in the toolstrip. You must **Apply** your settings to see the scans updated in the figures.

- Select the ROS topic for the lidar scans and odometry (if available).
- In **Odom Topic**, if you select `Use TF`, specify the frame of the lidar scan sensor, **Lidar Frame**, and the base fixed frame of the vehicle, **Fixed Frame**. The items in the drop down menu are generated based on the available frames in the tf transformation tree of the rosbag.
- Specify the **Start Time** and **End Time** if you want to trim data from rosbag. You can use the sliders or manually type in your time values.
- Select the desired downsample percentage of scans in **Downsample Scans**. This evenly downsamples the scans based on the percentage. For example, a value of 25% would only select every fourth scan.
- Click **Apply** to see the new filtered scans and apply all settings. **Close** the tab when you are done.

If you'd like more control over filtering scans in the rosbag, import your rosbag into MATLAB using `rosbag`. Filter the rosbag using `select`. To open the app using your custom filtered rosbag, see Programmatic Use on page 5-12.

Tune SLAM Settings

To improve the automatic map building process, the SLAM algorithm has tunable parameters. Click **SLAM Settings** to tune the parameters. Use **Lidar SLAM Parameters** to affect different aspects of the scan alignment and loop closure detection processes. Also, tune the **NLP Solver Parameters** to change how the map optimization algorithm improves the overall map based on loop closures.

Lidar SLAM Parameters:

- **Map Resolution (cells/m)** –– Resolution of the map. The resolution affects the location accuracy of the scan alignment and defines the output size of the occupancy grid.
- **Lidar Range [min,max] (m)** –– Range of lidar sensor readings. When processing the lidar scans, readings outside of the lidar range are ignored.
- **Loop Closure Threshold** –– Unitless threshold for accepting loop closures. Depending on your lidar scans, the average loop closure score varies. If the build process does not find loop closures and the vehicle revisits locations in the map, consider lowering this threshold.
- **Loop Closure Search Radius (m)** –– Radius to search for loop closures. Based on the odometry pose, the algorithm searches for loop closures in the existing map within the given radius in meters.
- **Loop Closure Max Attempts** –– Number of attempts at finding loop closures. When this number increases, the algorithm makes more attempts to find loop closures in the map but increases computation time.
- **Loop Closure Auto Rollback** –– Allow automatic rejection of loop closures. The algorithm tracks the residual error from the map optimization. If it detects a sudden change in the error and this parameter is set to `on`, the loop closure is rejected.
- **Optimization Interval (# of Loop Closures)** –– Number of detected loop closures accepted to trigger optimization. By default, the map is optimized with every loop closure found.

- **Movement Threshold [Linear,Angular] (m,rad)** –– Minimum change in pose required to accept a new scan. If the pose of the vehicle does not exceed this threshold, the next scan is discarded from the map building process.

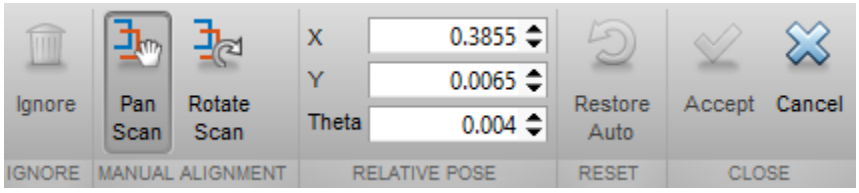
NLP Solver Parameters:

- **Max Iterations** –– Maximum number of iterations for map optimization. Increasing this value may improve map accuracy but increases computation time.
- **Max Time (s)** –– Maximum time allowed for map optimization specified in seconds. Increasing this value may improve map accuracy but increases computation time.
- **Gradient Tolerance** –– Lower bound on the norm of the gradient of the cost function for optimization. Lowering this value causes the optimization to run longer to search for a local minimum but increases the computation time.
- **Function Tolerance** –– Lower bound on the change in the cost function for optimization. Lowering this value causes the optimization to run longer to search for a local minimum but increases the computation time.
- **Step Tolerance** –– Lower bound on the step size for optimization. Lowering this value causes the optimization to run longer to search for a local minimum but increases the computation time.
- **First Node Pose [x,y,theta] (m,rad)** –– Pose of the first node in the graph. If you need to offset the position of the scans in the map, specify the position, `[x y]`, in meters and orientation, `theta`, in radians.

After changing any of these settings, the map building process must be restarted to rebuild the map with the new parameters.

Modify Incremental Matches and Loop Closures

This app allows you to manually modify incremental scans and adjust detected loop closures. If you notice scans are not properly aligned after you build the map, use the **Incremental Match** and **Loop Closure** buttons to open their modification tabs. Use the modification toolstrip buttons to adjust the relative pose between scans.



- **Ignore** –– When modifying loop closures, you can simply ignore loop closures if they are inaccurate. The algorithm always discards ignored loop closure if detected in the same app session. You cannot ignore incremental scan matches.
- **Pan Scan** –– Click this button to manually shift the pose. After selecting, click and drag inside the map to shift the scans and overlay them properly. Align all the points of the scans until you are satisfied. You can manually specify the **X**, **Y** location in the **Relative Pose** section as well.
- **Rotate Scan** –– Click this button to manually rotate the pose. After selecting, click and drag inside the map to rotate the scans and overlay them properly. Align all the points of the scans until you are satisfied. You can manually specify the **Theta** location in the **Relative Pose** section as well.

Sync the Map

After making modifications to the map building process using **Incremental Scans** and **Loop Closures**, you must sync the map to apply the changes. Based on the changes you make to properly align scans, the overall map shifts and alignments change for every scan after your modification. You have two options after making your modifications, **Sync** or **Sync Fast**. If you click **Sync Fast**, the changes to the poses are automatically applied and no other changes to the map occur. **Sync** restarts the entire map building and loop closure detection processes starting at the first modification. The specified modifications are applied, but the algorithm attempts to realign other scans and search for new loop closures as well.

See Also

Functions
`buildMap` | `matchScans` | `matchScansGrid` | `rosbag` | `optimizePoseGraph`

Objects
`lidarSLAM` | `lidarScan` | `occupancyMap` | `poseGraph`

Topics
“Implement Simultaneous Localization And Mapping (SLAM) with Lidar Scans”
“Implement Online Simultaneous Localization And Mapping (SLAM) with Lidar Scans”

Introduced in R2019b