

Navigation Toolbox™

Reference



MATLAB® & SIMULINK®

R2020a



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

© COPYRIGHT 2019–2020 by The MathWorks, Inc.

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

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

Trademarks

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

Patents

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

Revision History

September 2019	Online only	New for Version 1.0 (R2019b)
March 2020	Online only	Rereleased for Version 1.1 (R2020a)

1	Functions
2	Classes
3	Methods
4	Blocks
5	Apps

Functions

addRelativePose

Add relative pose to pose graph

Syntax

```
addRelativePose(poseGraph, relPose)
addRelativePose(poseGraph, relPose, infoMatrix)
addRelativePose(poseGraph, relPose, infoMatrix, fromNodeID)
addRelativePose(poseGraph, relPose, infoMatrix, fromNodeID, toNodeID)
[edge, edgeID] = addRelativePose( ___ )
```

Description

`addRelativePose(poseGraph, relPose)` creates a pose node and uses an edge specified by `relPose` to connect it to the last node in the pose graph.

`addRelativePose(poseGraph, relPose, infoMatrix)` also specifies the information matrix as part of the edge constraint, which represents the uncertainty of the pose measurement.

`addRelativePose(poseGraph, relPose, infoMatrix, fromNodeID)` creates a new pose node and connects it to the specific node specified by `fromNodeID`.

`addRelativePose(poseGraph, relPose, infoMatrix, fromNodeID, toNodeID)` creates an edge by specifying a relative pose between existing nodes specified by `fromNodeID` and `toNodeID`. This edge is called a loop closure.

`[edge, 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.

relPose — 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.

infoMatrix — Information matrix

6-element vector | 21-element vector

Information matrix, specified as a 6-element or 21-element vector. This vector contains the elements of the upper triangle of the square information matrix (compact form). The information matrix is the inverse of the covariance of the pose and represents the uncertainty of the measurement. If the pose vector is $[x \ y \ \theta]$, the covariance is a 3-by-3 matrix of pairwise covariance calculations. Typically, the uncertainty is determined by the sensor model.

For poseGraph (2-D), the information matrix is a six-element vector. The default is $[1 \ 0 \ 0 \ 1 \ 0 \ 1]$.

For poseGraph3D, the information matrix is a 21-element vector. The default is $[1 \ 0 \ 0 \ 0 \ 0 \ 0 \ 1 \ 0 \ 0 \ 0 \ 0 \ 1 \ 0 \ 0 \ 0 \ 1 \ 0 \ 0 \ 0 \ 1 \ 0 \ 1]$.

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**edge — Added edge**

two-element vector

Added edge, returned as a two-element vector. An edge is defined by the IDs of the two nodes that it connects with a relative pose.

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

edgeConstraints | edges | findEdgeID | nodes | optimizePoseGraph | removeEdges

Objects

lidarSLAM | poseGraph | poseGraph3D

Topics

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

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

Introduced in R2019b

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 $(1, 2, \dots, 2^{\lfloor \log_2[(N-1)/2] \rfloor})$, 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

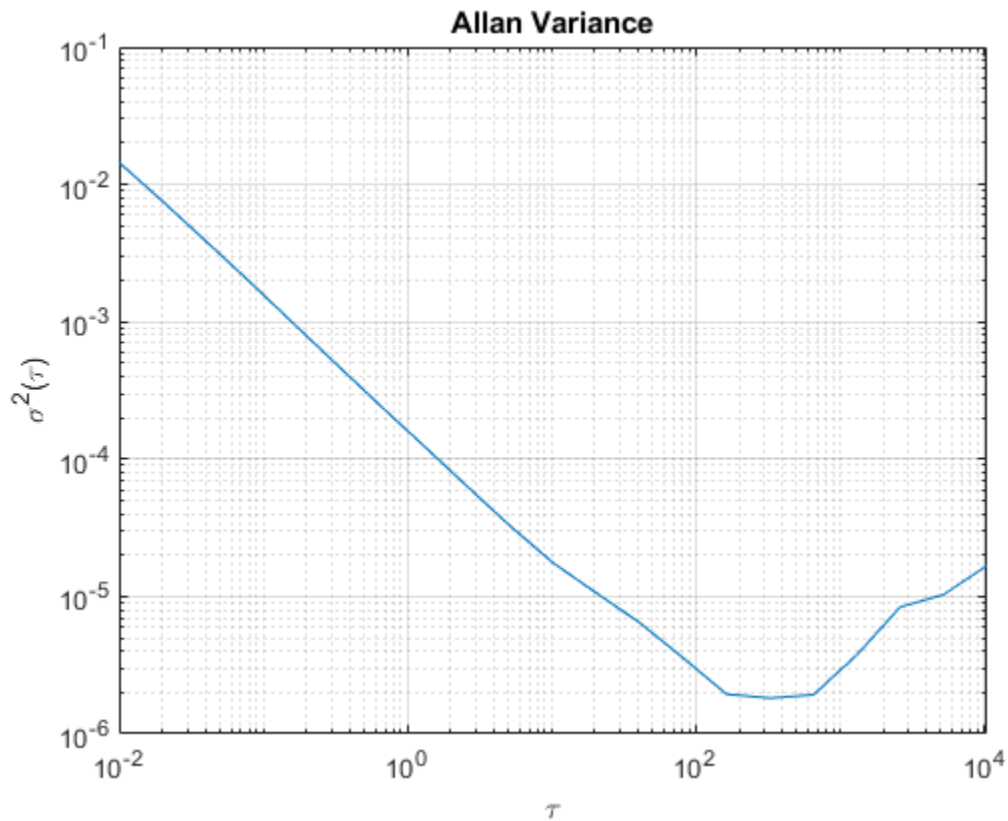
Determine Allan Variance of Single Axis Gyroscope

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 log log 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;
```

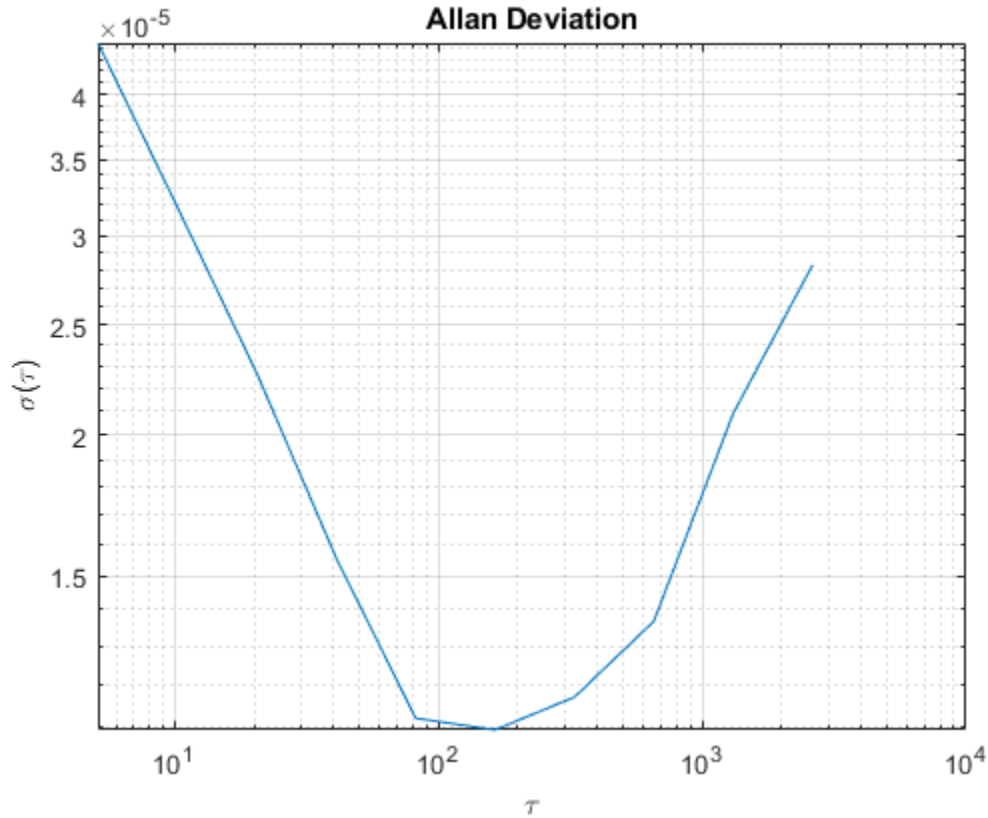
Calculate the Allan deviation at specific values of $m = \tau$. The Allan deviation is the square root of the Allan variance.

```
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:

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

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

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

N is the number of samples in Ω .

fs — Basic frequency of input data in Hz

scalar

Basic frequency of the input data, Ω , 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 .

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.

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]`.

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

Topics

“Coordinate Transformations in Robotics” (Robotics System Toolbox)

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

Topics

“Coordinate Transformations in Robotics” (Robotics System Toolbox)

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`

Topics

“Coordinate Transformations in Robotics” (Robotics System Toolbox)

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`.

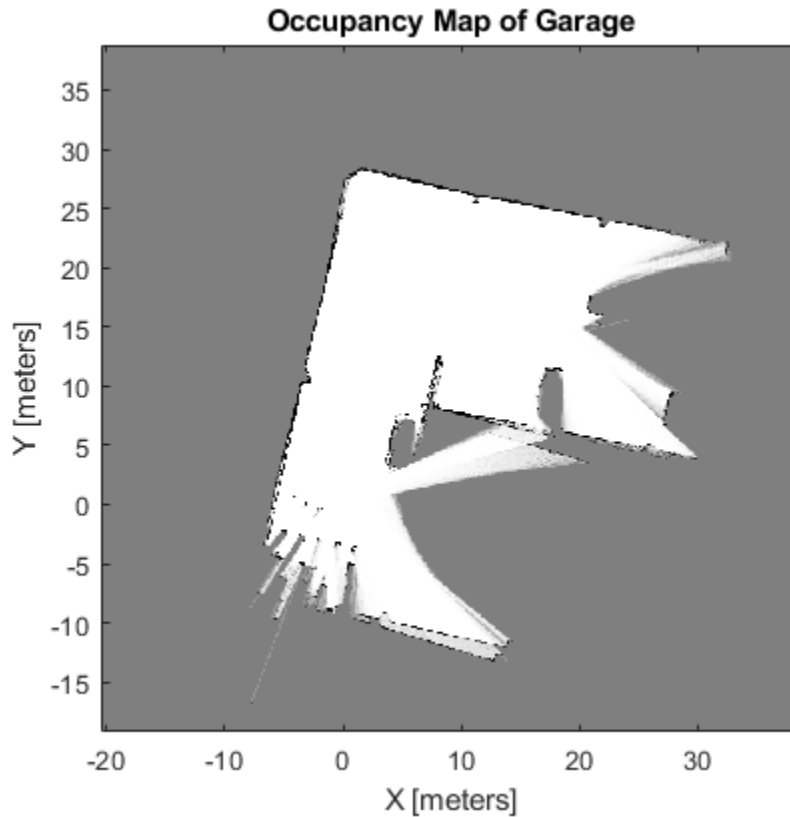
Load scan and pose estimates collected from sensors on a robot in a parking garage. The data collected is correlated using a `lidarSLAM` algorithm, which performs scan matching to associate scans and adjust poses over the full robot trajectory. Check to make sure scans and poses are the same length.

```
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.

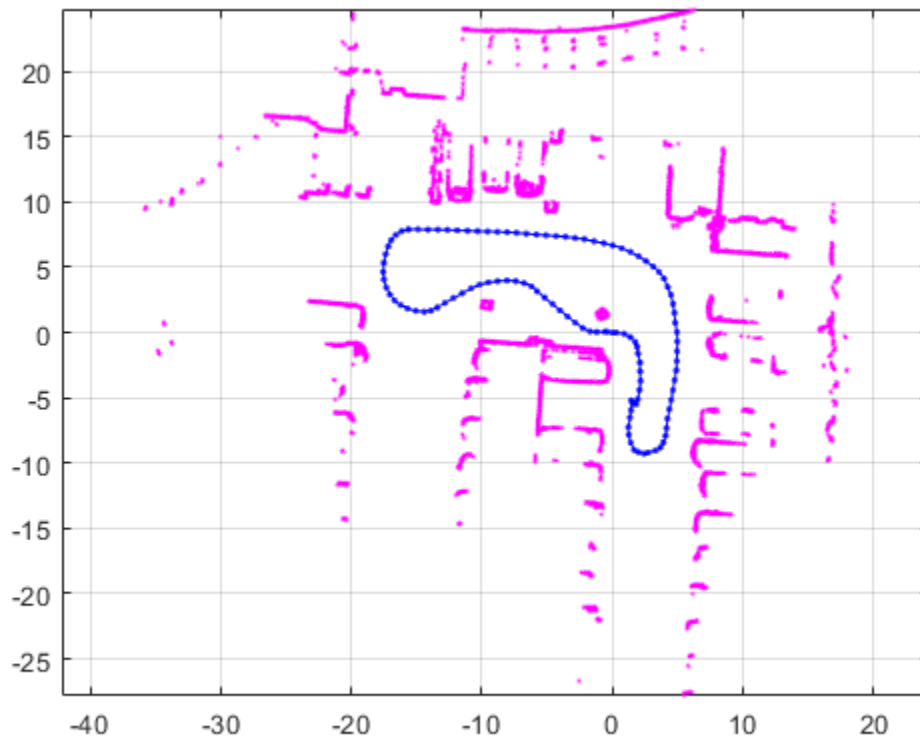
```
maxRange = 19.2; % meters
resolution = 10; % cells per meter

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.

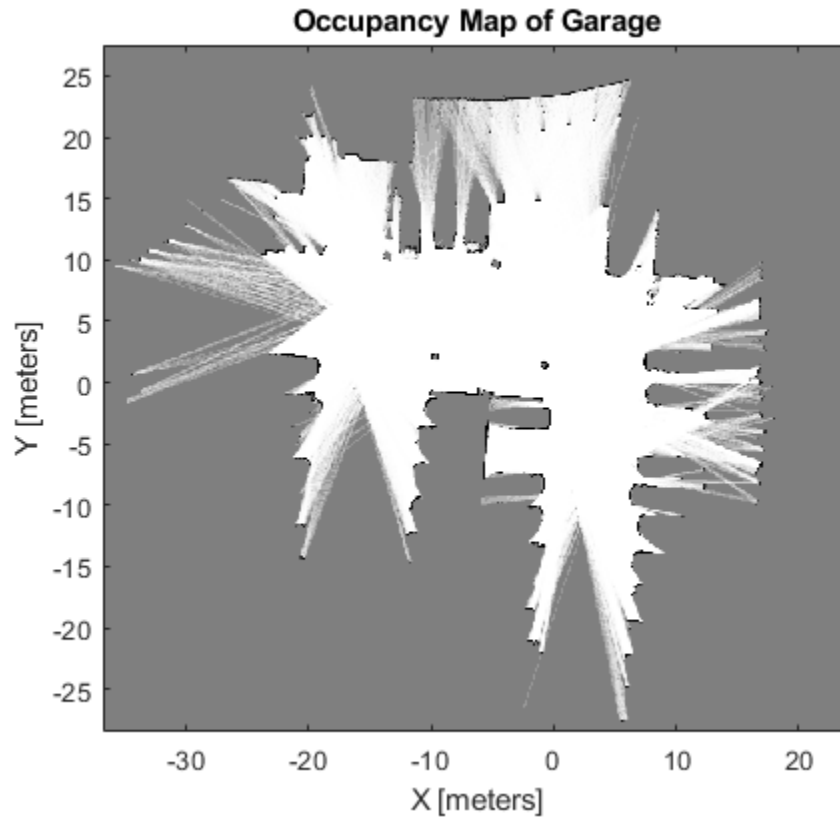
```
for i = 1:numel(scans)
    addScan(slamObj,scans{i});
    if rem(i,10) == 0
        show(slamObj);
    end
end
```



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

`lidarScan` | `matchScans` | `matchScansGrid` | `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

Topics

“Coordinate Transformations in Robotics” (Robotics System Toolbox)

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.

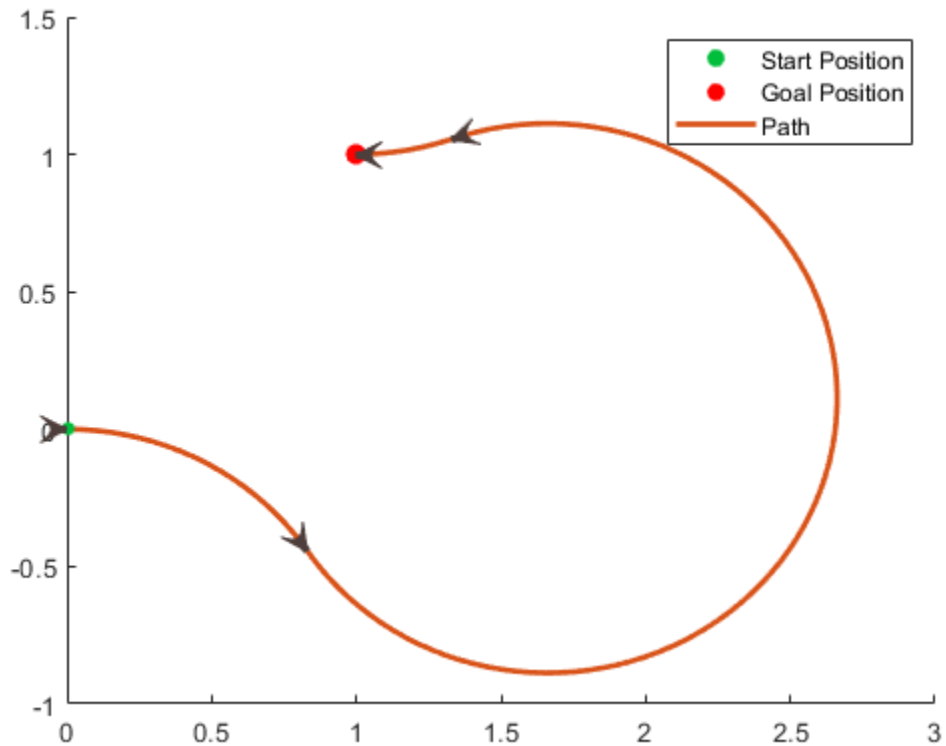
```
startPose = [0 0 0];
goalPose = [1 1 pi];
```

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.

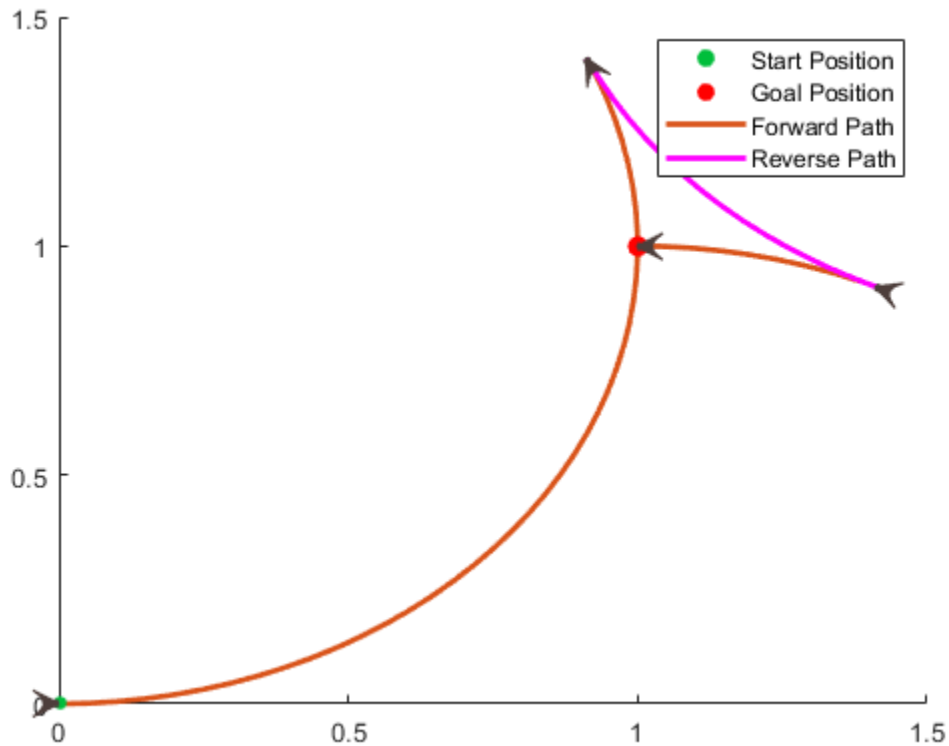
```
startPose = [0 0 0];
goalPose = [1 1 pi];
```

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

$[x, y, \theta]$ vector or matrix

This property is read-only.

Initial pose of the robot at the start of the path segment, specified as an $[x, y, \theta]$ 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(numStateVariables);
    obj.UniformDistribution = matlabshared.tracking.internal.UniformDistribution(numStateVariables);
    % 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, "enforce");
    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, varargin{:});

    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, "dis
    nav.internal.validation.validateStateMatrix(state2, size(state1,1), obj.NumStateVari

    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.

```

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

        if nargin == 0
            space = stateSpaceSE2;
        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.NumStateVariables,
        "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 invalid");
    end
end

```

```
% Interpolate at a fixed interval between states and check state validity
numInterpPoints = 100;
interpStates = obj.StateSpace.interpolate(state1, state2, linspace(0,1,numInterpPoints));
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);  
quaternionEulerAngles = eulerd(q, 'ZYX', 'frame')
```

```
quaternionEulerAngles = 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 = 1x3
```

```
    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 `kinematicTrajectorySystem` 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,angularVelocityBody);
```

Model Receiving IMU Data

Use an `imuSensorSystem` 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');
```

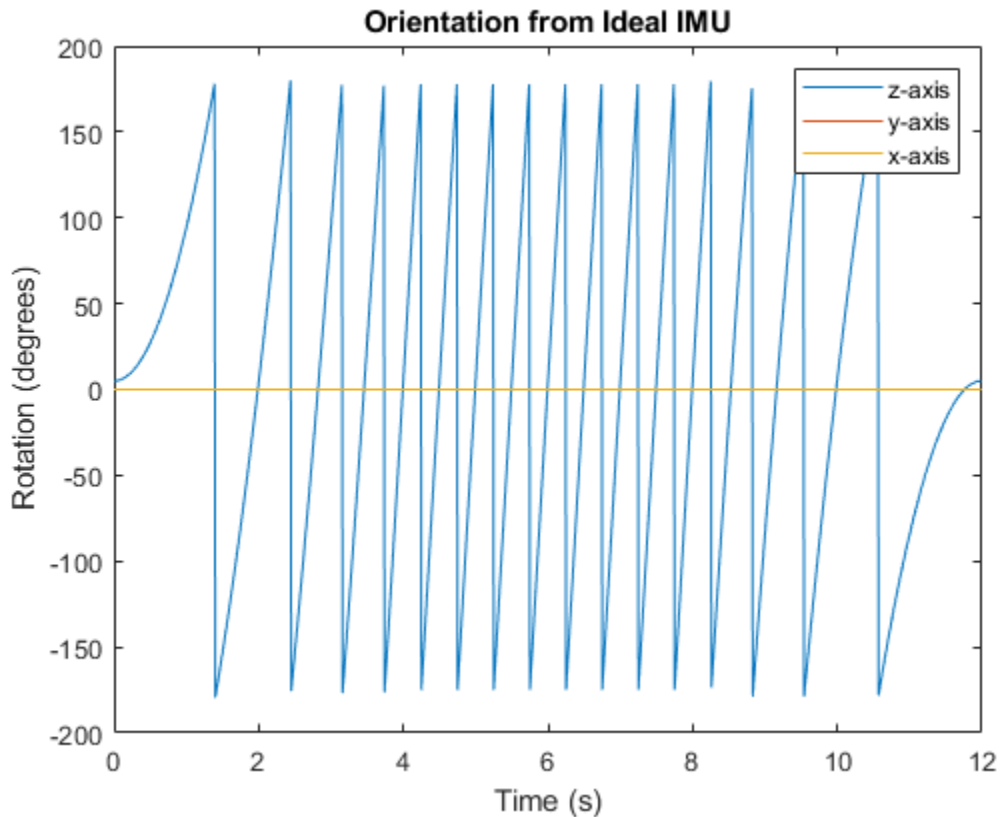
```
timeVector = (0:numSamples-1).'/fs;
```

```
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)

```

```

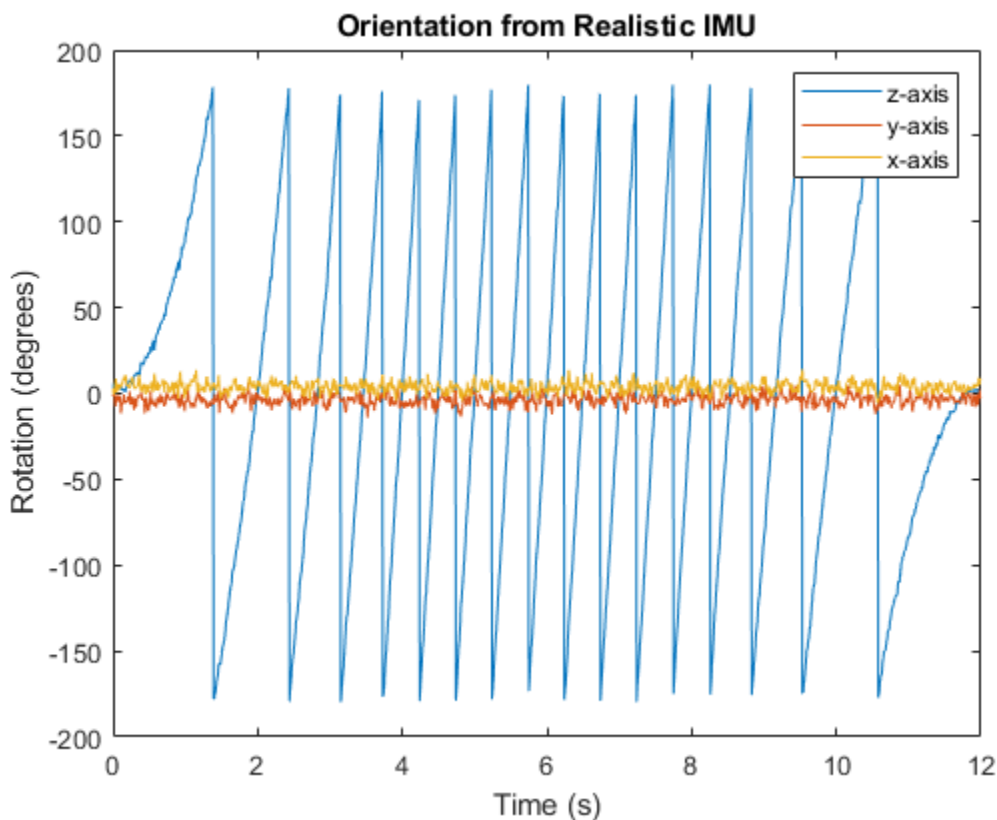
[accelerometerData,magnetometerData] = IMU(accelerationNED,angularVelocityNED,orientationNED);
orientation = ecompass(accelerometerData,magnetometerData);
orientationEuler = eulerd(orientation,'ZYX','frame');

```

```

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/s^2)

N -by-3 matrix

Accelerometer readings in sensor body coordinate system in m/s^2 , 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

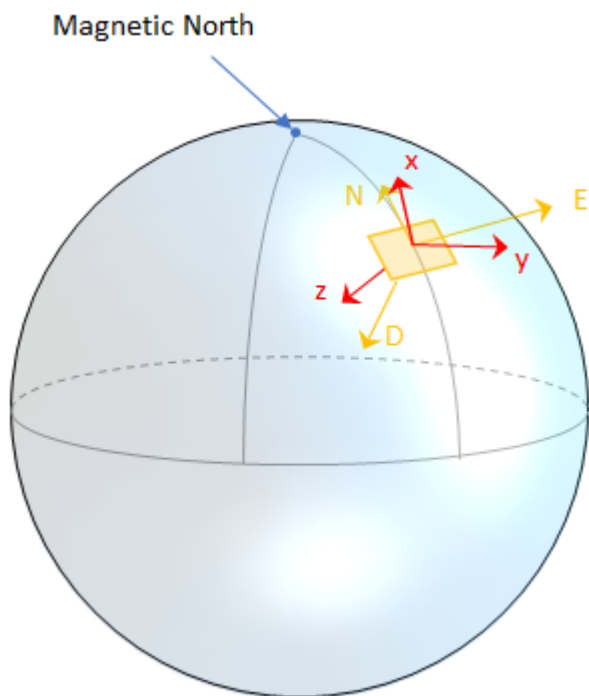
The `ecompass` function returns a quaternion or rotation matrix that can rotate quantities from a parent (NED for example) frame to a child (sensor) frame. For both output orientation formats, the rotation operator is determined by computing the rotation matrix.

The rotation matrix is first calculated with an intermediary:

$$R = \begin{bmatrix} (a \times m) \times a & a \times m & a \end{bmatrix}$$

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.



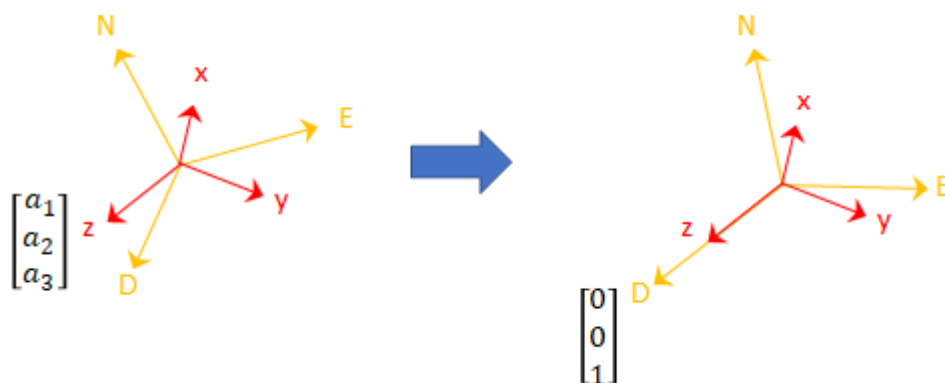
Recall that orientation of a sensor body is defined as the rotation operator (rotation matrix or quaternion) required to rotate a quantity from a parent (NED) frame to a child (sensor body) frame:

$$\begin{bmatrix} R \end{bmatrix} \begin{bmatrix} p_{\text{parent}} \end{bmatrix} = \begin{bmatrix} p_{\text{child}} \end{bmatrix}$$

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.

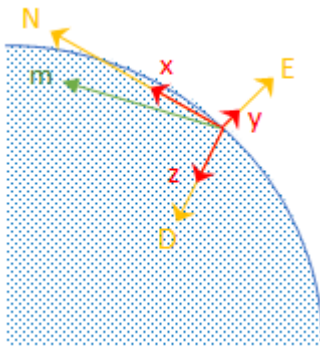


$$\begin{bmatrix} r_{11} & r_{21} & r_{31} \\ r_{12} & r_{22} & r_{32} \\ r_{13} & r_{23} & r_{33} \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix}$$

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

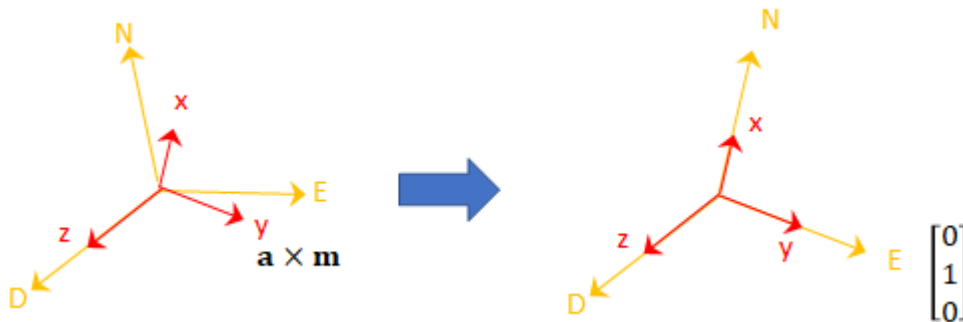
$$\begin{bmatrix} r_{31} \\ r_{32} \\ r_{33} \end{bmatrix} = \begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix}$$

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:

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

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

edgeConstraints

Edge constraints in pose graph

Syntax

```
relPoses = edgeConstraints(poseGraph)
[relPoses,infoMatrices] = edgeConstraints(poseGraph)
[relPoses,infoMatrices] = edgeConstraints(poseGraph,edgeIDs)
```

Description

`relPoses = edgeConstraints(poseGraph)` lists all edge constraints in the specified pose graph as a relative pose.

`[relPoses,infoMatrices] = edgeConstraints(poseGraph)` also returns the information matrices for each edge. The information matrix is the inverse of the covariance of the pose measurement.

`[relPoses,infoMatrices] = 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

relPoses — Relative poses measured between nodes

n-by-3 matrix | *n*-by-7 matrix

Relative poses measured between nodes, returned as an *n*-by-3 matrix or *n*-by-7 matrix.

For poseGraph (2-D), each row is an $[x \ y \ \text{theta}]$ vector, which defines the relative *xy*-position and orientation angle, *theta*, of a pose in the graph.

For poseGraph3D, each row is an $[x \ y \ z \ q_w \ q_x \ q_y \ q_z]$ vector, which defines the relative *xyz*-position and quaternion orientation, $[q_w \ q_x \ q_y \ q_z]$, 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, $[q_x \ q_y \ q_z \ q_w]$. Check the source of your pose graph data before adding nodes to your poseGraph3D object.

infoMatrices — 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 a vector that contains the elements of the upper triangle of the square information matrix. The information matrix is the inverse of the covariance of the pose and represents the uncertainty of the measurement. If the pose vector is `[x y theta]`, the covariance is a 3-by-3 matrix 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 `poseGraph3D`, each information matrix is a 21-element vector. The default is `[1 0 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**

`addRelativePose` | `edges` | `findEdgeID` | `nodes` | `optimizePoseGraph` | `removeEdges`

Objects

`lidarSLAM` | `poseGraph` | `poseGraph3D`

Topics

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

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

Introduced in R2019b

edges

Edges in pose graph

Syntax

```
edges = edges(poseGraph)
edges = edges(poseGraph, edgeIDs)
```

Description

`edges = edges(poseGraph)` returns all edges in the specified pose graph as a list of node ID pairs.

`edges = edges(poseGraph, edgeIDs)` returns edges corresponding to 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

edges — Edges in pose graph

n-by-2 matrix

Edges in pose graph, returned as *n*-by-2 matrix that lists the IDs of the two nodes that each edge connects.

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

`addRelativePose` | `edgeConstraints` | `findEdgeID` | `nodes` | `optimizePoseGraph` | `removeEdges`

Objects

`lidarSLAM` | `poseGraph` | `poseGraph3D`

Topics

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

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

Introduced in R2019b

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) | "YZZ" | "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.
- "YZZ" - 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

Topics

"Coordinate Transformations in Robotics" (Robotics System Toolbox)

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`

Topics

"Coordinate Transformations in Robotics" (Robotics System Toolbox)

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) | "YZZ" | "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.
- "YZZ" - 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`

Topics

"Coordinate Transformations in Robotics" (Robotics System Toolbox)

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`

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

`lidarSLAM` | `occupancyMap` | `occupancyMap3D`

Functions

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

Introduced in R2020a

findEdgeID

Find edge ID of edge

Syntax

```
edgeID = findEdgeID(poseGraph,edge)
```

Description

`edgeID = findEdgeID(poseGraph,edge)` 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.

edge — Edge in pose graph

two-element vector

Edge 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

Edge IDs, 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.

See Also

Functions

`addRelativePose` | `edgeConstraints` | `edges` | `nodes` | `optimizePoseGraph` | `removeEdges`

Objects

lidarSLAM | poseGraph | poseGraph3D

Topics

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

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

Introduced in R2019b

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

Topics

“Coordinate Transformations in Robotics” (Robotics System Toolbox)

Introduced in R2015a

importOccupancyMap3D

Import an octree file as 3D occupancy map

Syntax

```
map3D = importatOccupancyMap3D(mapPath)
```

Description

`map3D = importatOccupancyMap3D(mapPath)` imports the octree file (.ot/bt) specified at the relative or absolute file path, `mapPath`

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

lidarSLAM | occupancyMap | occupancyMap3D

Functions

inflate | insertPointCloud | setOccupancy | show

Introduced in R2020a

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

[ahrsfilter](#) | [imufilter](#) | [insfilterAsync](#) | [insfilterErrorState](#) | [insfilterMARG](#) | [insfilterNonholonomic](#)

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.

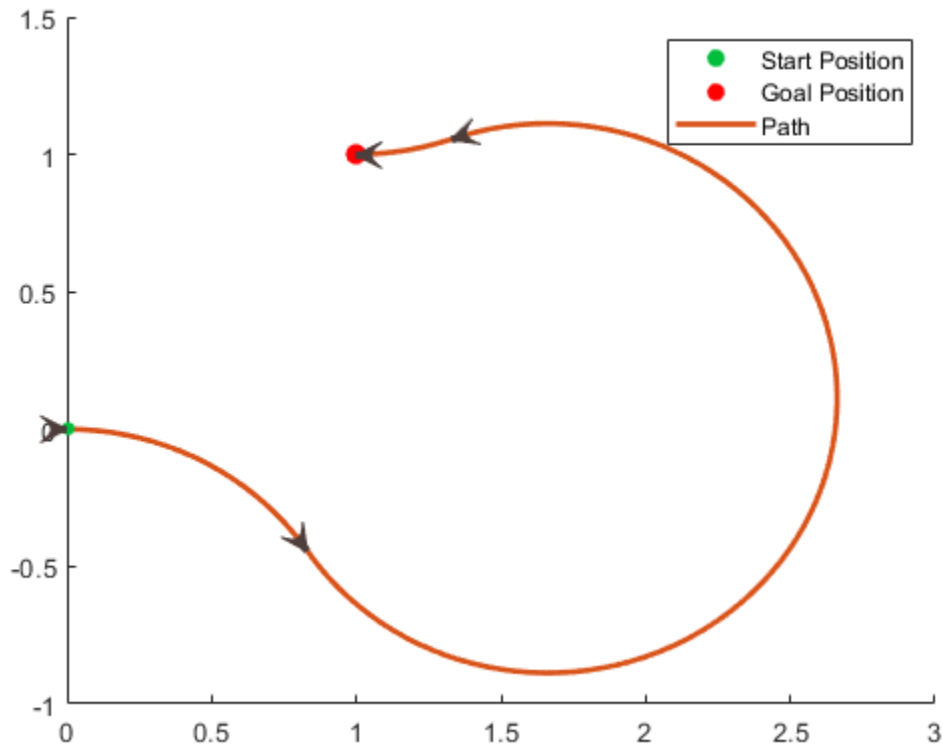
```
startPose = [0 0 0];
goalPose = [1 1 pi];
```

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.

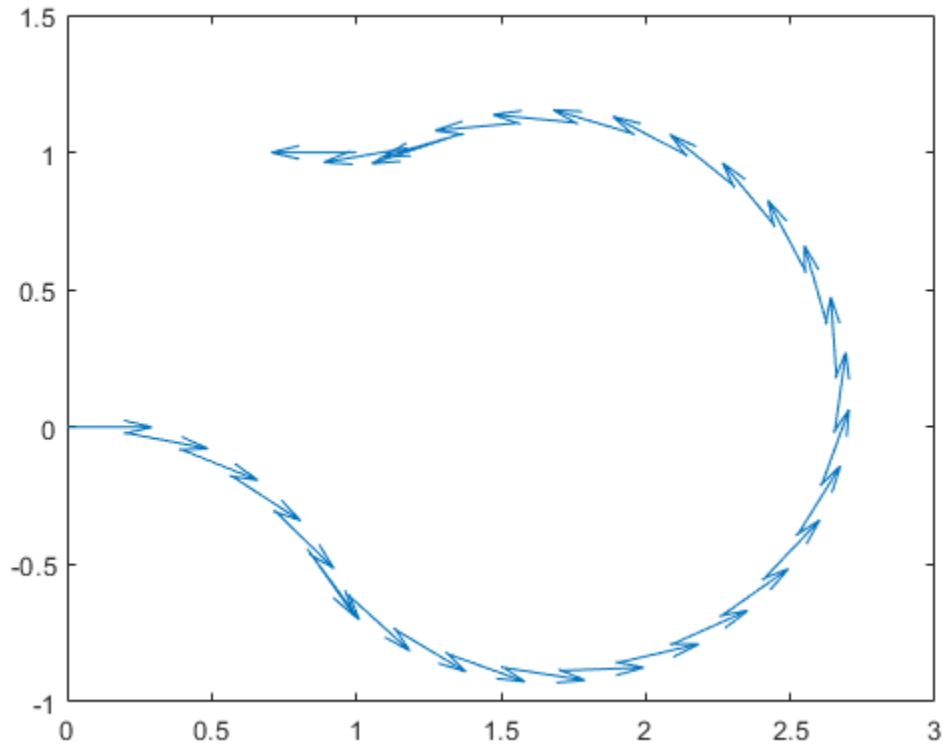
```
length = pathSegObj{1}.Length;
poses = interpolate(pathSegObj{1},0:0.2:length)
```

```
poses = 32x3
```

	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)
```



Error using matlab.internal.editor.figure.SerializedFigureState/serialize
Error while evaluating Figure SizeChangedFcn.

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

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
```

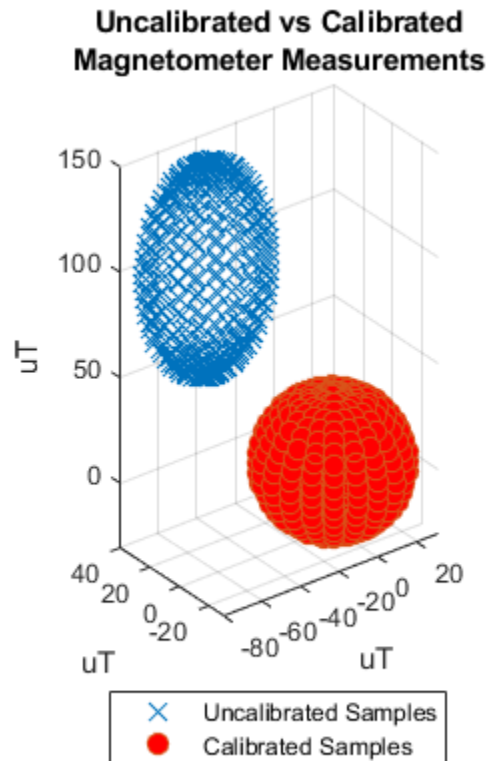
Visualize the uncalibrated and calibrated magnetometer data.

```
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

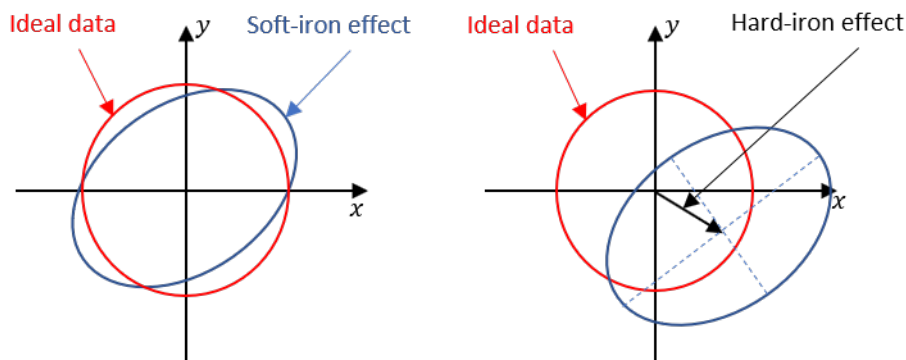
Expected magnetic field strength, returned as a scalar.

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

System Objects

imuSensor

Introduced in R2019a

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` 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);
```

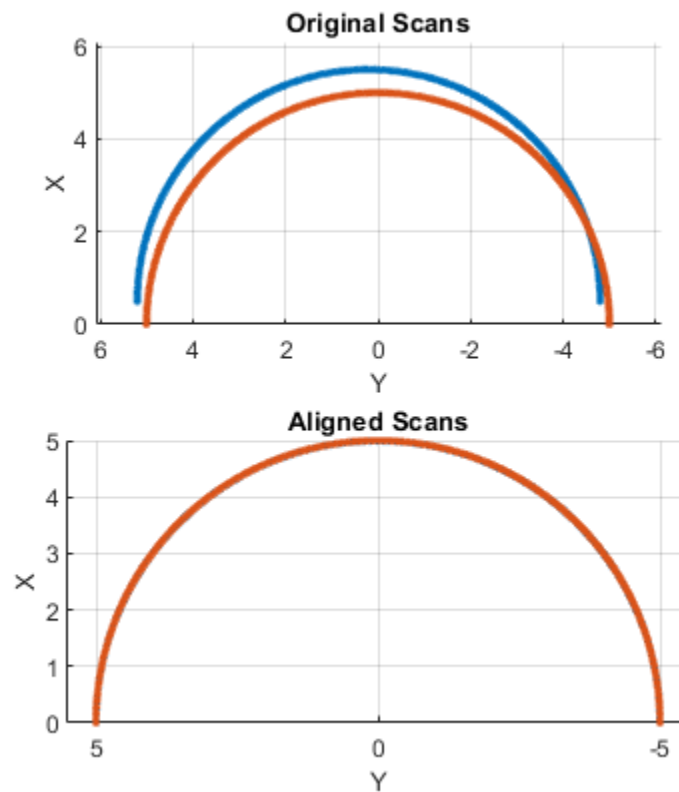
```
subplot(2,1,1);
hold on
plot(currScan)
```

```

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 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.

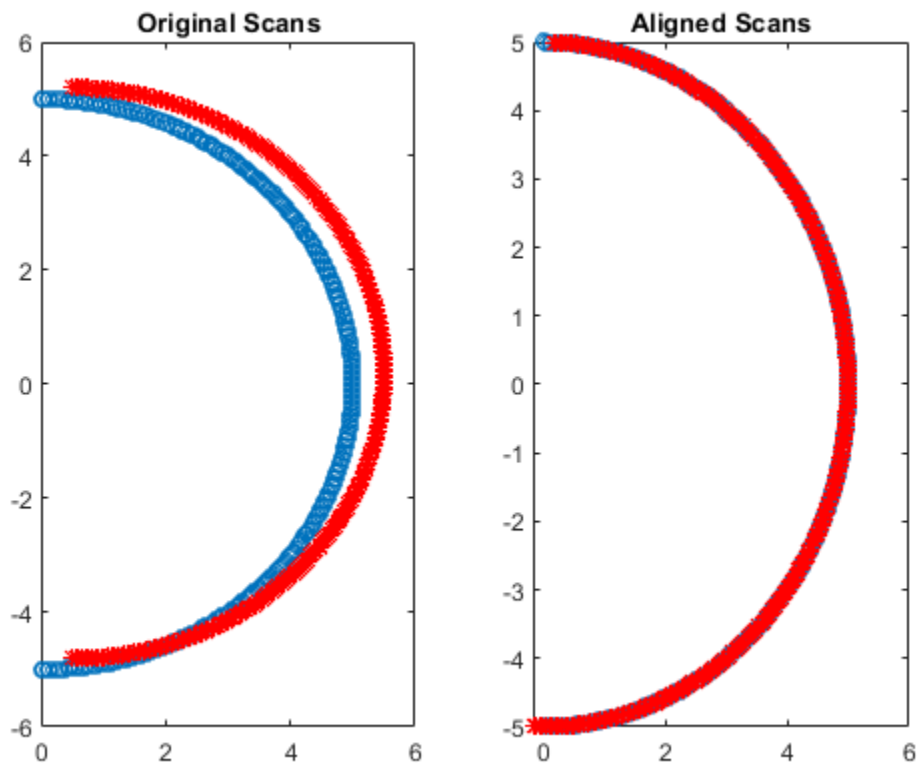
```
pose = matchScans(currRanges,currAngles,refRanges,refAngles,...
    'SolverAlgorithm','fminunc','InitialPose',[-0.4 -0.1 0]);
```

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

lidarScan | matchScansGrid | matchScansLine | transformScan

Classes

monteCarloLocalization | occupancyMap

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.

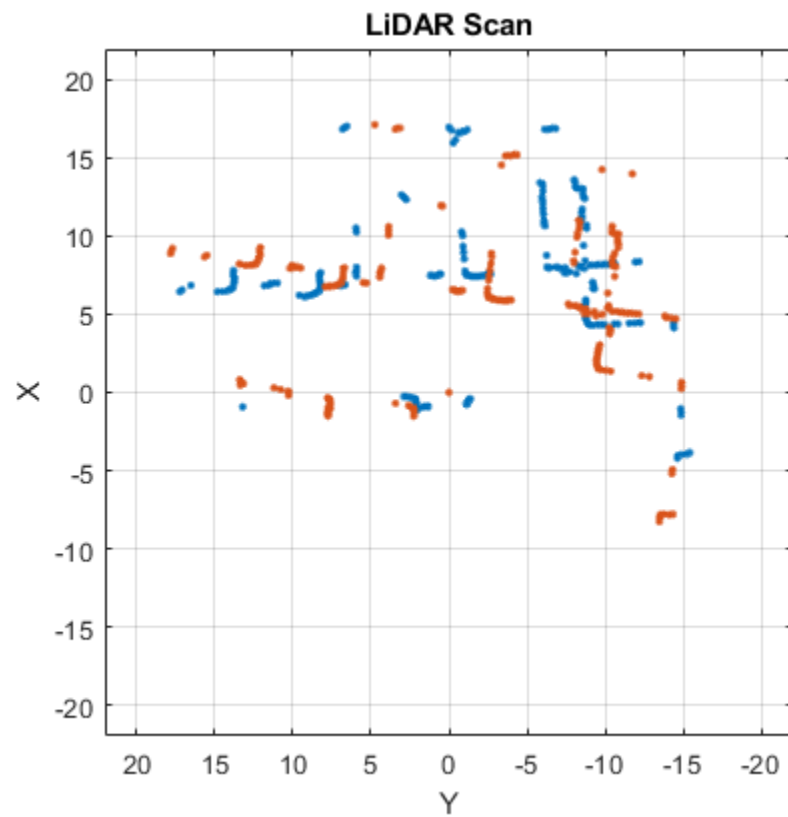
Examples

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
```



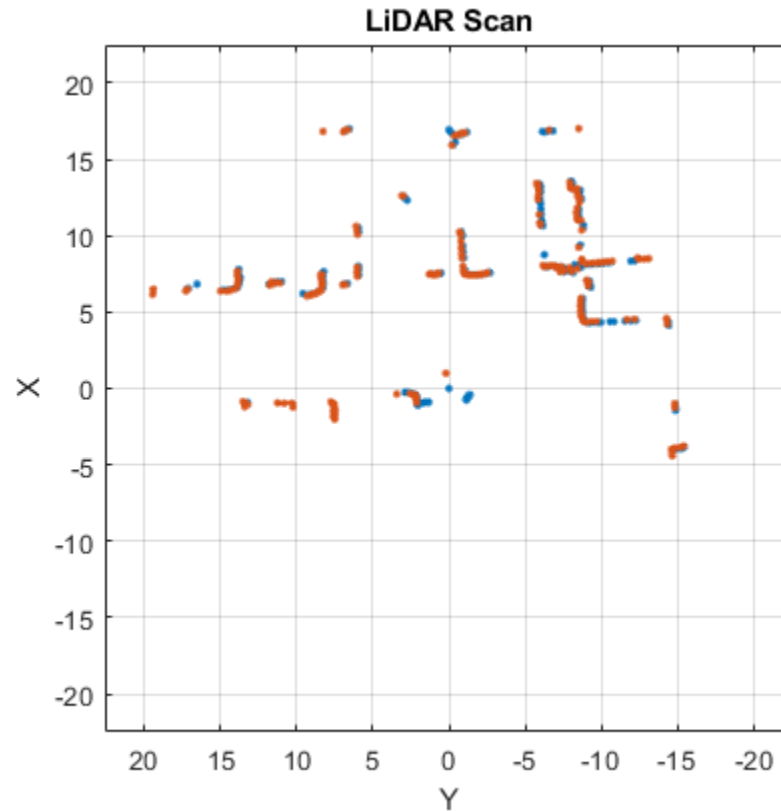
Use `matchScansGrid` to estimate the pose between the two scans.

```
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

lidarScan | matchScans | matchScansLine | 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.

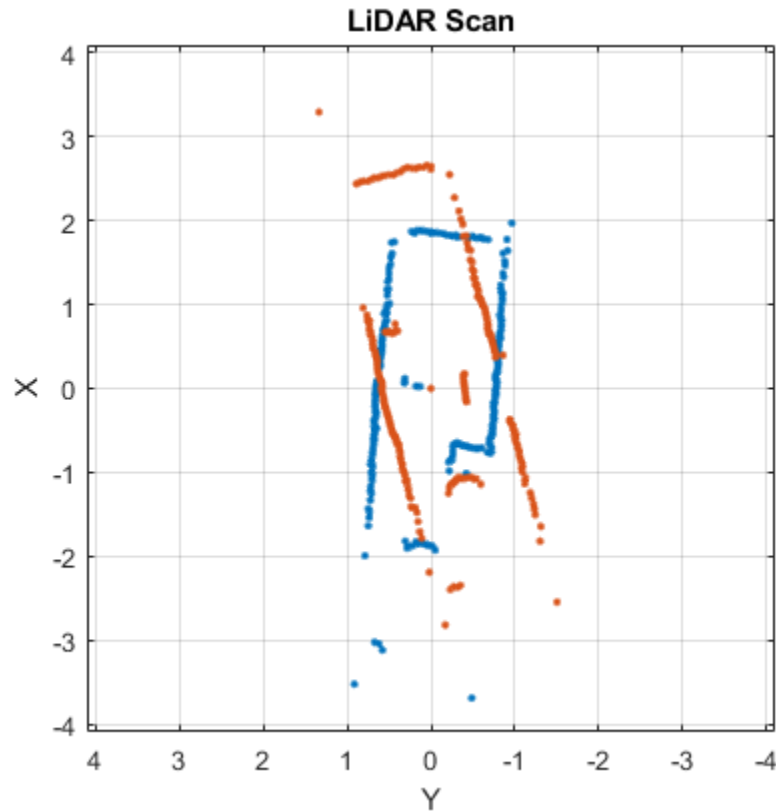
Examples

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`.

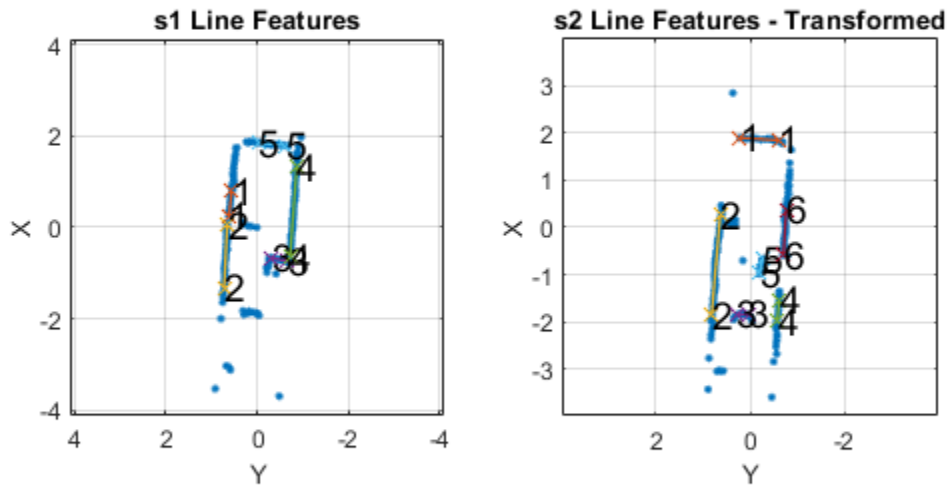
```
debugInfo.MatchHypothesis
```

```
ans = 1x6
```

```
    5    2    0    0    0    4
```

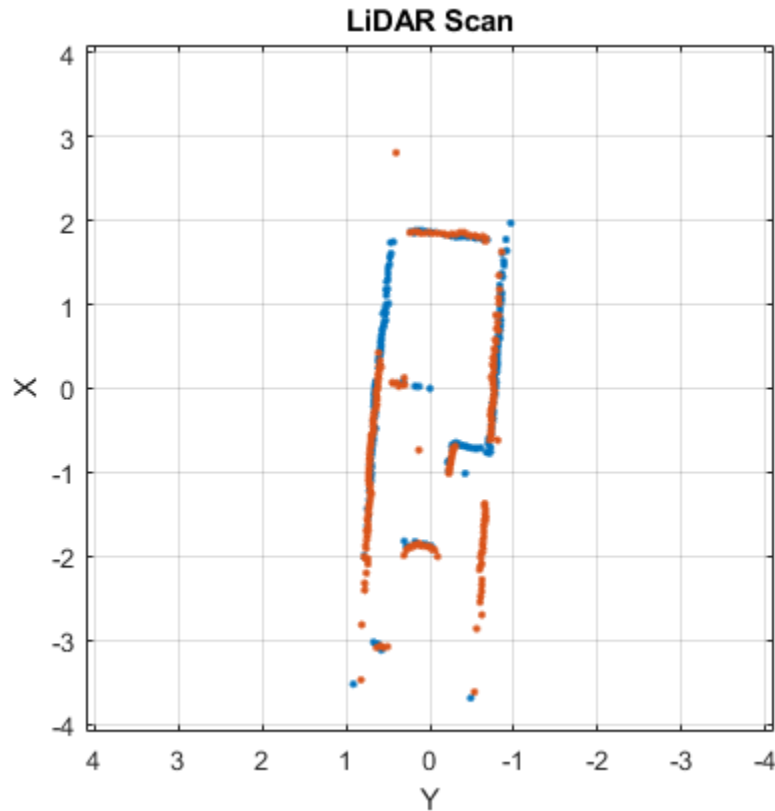
The provided helper function plots these two scans and the features extracted with labels. s2 is transformed to be in the same frame based on the initial guess for relative pose.

```
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.

MinCornerPromenace — 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, $\rho = x \cdot \cos(\alpha) + y \cdot \sin(\alpha)$.
- **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, $\rho = x \cdot \cos(\alpha) + y \cdot \sin(\alpha)$.
- **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

`lidarSLAM` | `matchScans` | `matchScansGrid`

Topics

"Estimate Robot Pose with Scan Matching"

"Implement Simultaneous Localization And Mapping (SLAM) with Lidar Scans"

Introduced in R2020a

nodes

Poses of nodes in pose graph

Syntax

```
nodes = nodes(poseGraph)
nodes = nodes(poseGraph, nodeIDs)
```

Description

`nodes = nodes(poseGraph)` lists all poses in the specified pose graph.

`nodes = nodes(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

nodes — Pose of each node

n -by-3 matrix | n -by-7 matrix

Pose of each node, specified as an n -by-3 or n -by-7 matrix. These poses are given in global coordinates for the whole pose graph.

For poseGraph (2-D), each row is an $[x \ y \ \text{theta}]$ vector, which defines the relative xy -position and orientation angle, theta , of a pose in the graph.

For poseGraph3D, each row is an $[x \ y \ z \ q_w \ q_x \ q_y \ q_z]$ vector, which defines the relative xyz -position and quaternion orientation, $[q_w \ q_x \ q_y \ q_z]$, 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, $[q_x \ q_y \ q_z \ q_w]$. Check the source of your pose graph data before adding nodes to your poseGraph3D object.

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

`addRelativePose` | `edgeConstraints` | `edges` | `findEdgeID` | `optimizePoseGraph` | `removeEdges`

Objects

`lidarSLAM` | `poseGraph` | `poseGraph3D`

Topics

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

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

Introduced in R2019b

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.

`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.

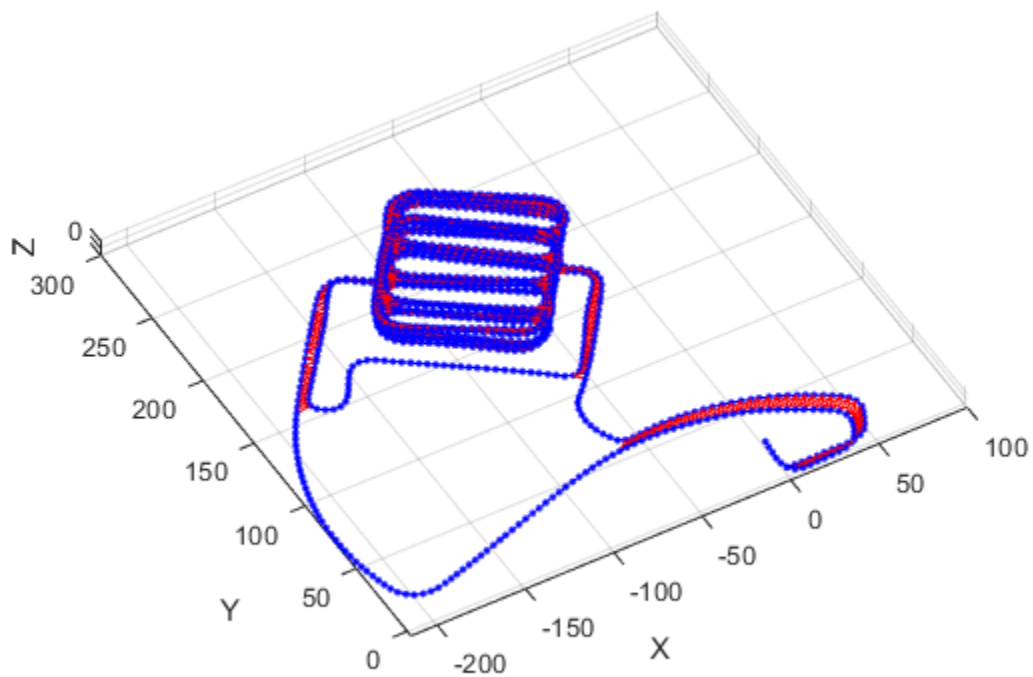
```
load parking-garage-posegraph.mat pg
disp(pg);

poseGraph3D with properties:

    NumNodes: 1661
    NumEdges: 6275
 NumLoopClosureEdges: 4615
 LoopClosureEdgeIDs: [1x4615 double]
```

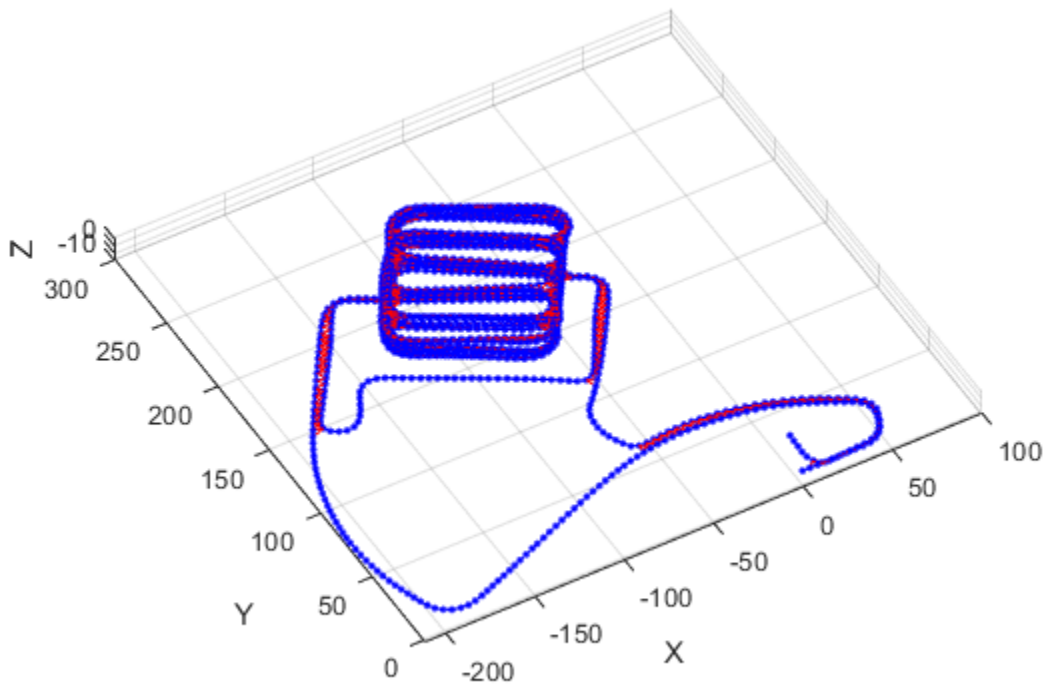
Plot the pose graph with IDs off. Red lines indicate loop closures identified in the dataset.

```
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.

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

addRelativePose | edgeConstraints | edges | findEdgeID | nodes | removeEdges

Objects

lidarSLAM | poseGraph | poseGraph3D

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 quaternion 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 UIAxesobject. See `axes` or `uiaxes`.

See Also

`eul2quat` | `hom2cart` | `quaternion` | `rotm2quat` | `tform2quat`

Introduced in R2018b

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

Topics

“Coordinate Transformations in Robotics” (Robotics System Toolbox)

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

Topics

“Coordinate Transformations in Robotics” (Robotics System Toolbox)

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 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

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

[quaternion](#) | [rotm2quat](#)

Topics

“Coordinate Transformations in Robotics” (Robotics System Toolbox)

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

[quaternion](#) | [tform2quat](#)

Topics

“Coordinate Transformations in Robotics” (Robotics System Toolbox)

Introduced in R2015a

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.

`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`.

Examples

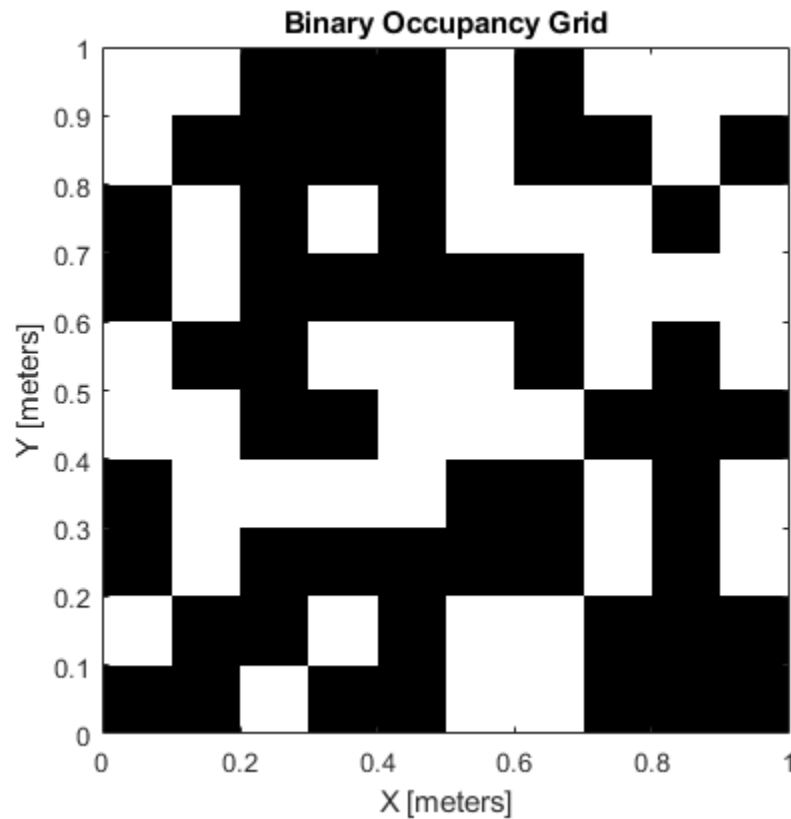
Read Binary Occupancy Data from ROS Message

Create a occupancy grid message and populate it with data.

```
msg = rosmesssage('nav_msgs/OccupancyGrid');
msg.Info.Height = 10;
msg.Info.Width = 10;
msg.Info.Resolution = 0.1;
msg.Data = 100*rand(100,1);
```

Read data from message. Show the map.

```
map = readBinaryOccupancyGrid(msg);
show(map)
```

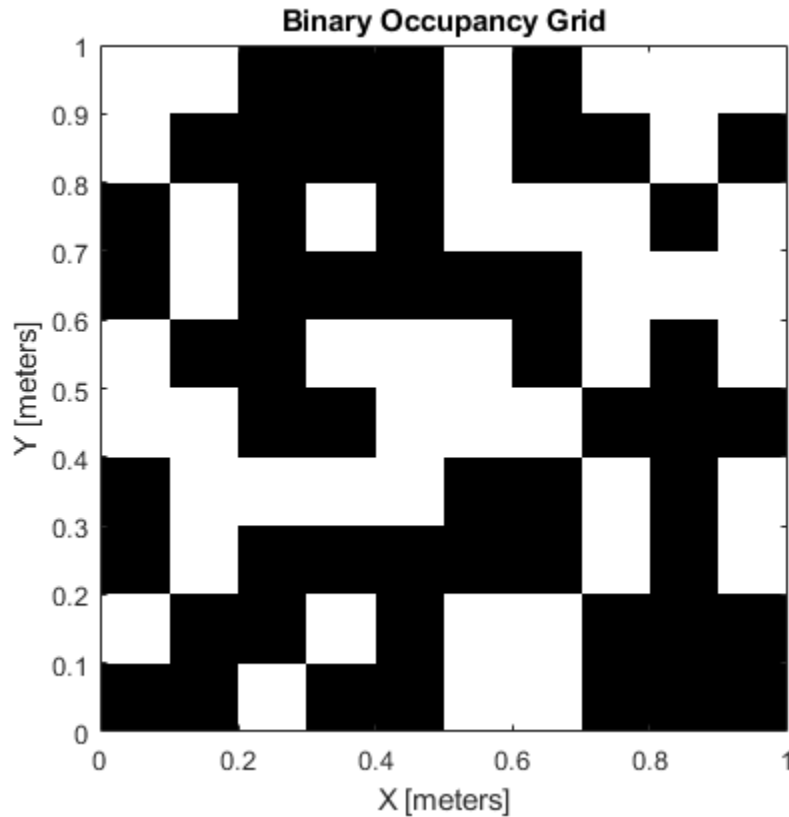
Read Binary Occupancy Data from ROS Message

Create a occupancy grid message and populate it with data.

```
msg = rosmesssage('nav_msgs/OccupancyGrid');  
msg.Info.Height = 10;  
msg.Info.Width = 10;  
msg.Info.Resolution = 0.1;  
msg.Data = 100*rand(100,1);
```

Read data from message. Show the map.

```
map = readBinaryOccupancyGrid(msg);  
show(map)
```



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. It is an object with a grid of binary values, where 1 indicates an occupied location and 0 indicates an unoccupied location.

See Also

`binaryOccupancyMap` | `controllerVFH` | `occupancyMap`

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.

Examples

Read An OccupancyGrid Message from ROS

Create a `nav_msgs/OccupancyGrid` ROS message.

```
msg = rosmessage('nav_msgs/OccupancyGrid');
```

Populate the ROS occupancy grid message with data.

```
msg.Info.Height = 10;  
msg.Info.Width = 10;  
msg.Info.Resolution = 0.1;  
msg.Data = 100*rand(100,1);
```

Read the `msg` data and convert to an `OccupancyGrid` object.

```
map = readOccupancyGrid(msg);
```

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

[OccupancyGrid](#) | [binaryOccupancyMap](#) | [occupancyMap](#) | [readBinaryOccupancyGrid](#) | [writeBinaryOccupancyGrid](#)

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.

Examples

Read Octomap ROS Messages

Load Octomap ROS messages and read them into MATLAB® as an `OccupancyMap3D` object.

Load the Octomap ROS messages. The Octomap map messages were previously recorded in a rosbag and read into MATLAB® as ROS message objects. You could also get these ROS messages live on a network.

```
load octomap_msgs
disp(octomapMsgs{1})
```

```
ROS Octomap message with properties:
```

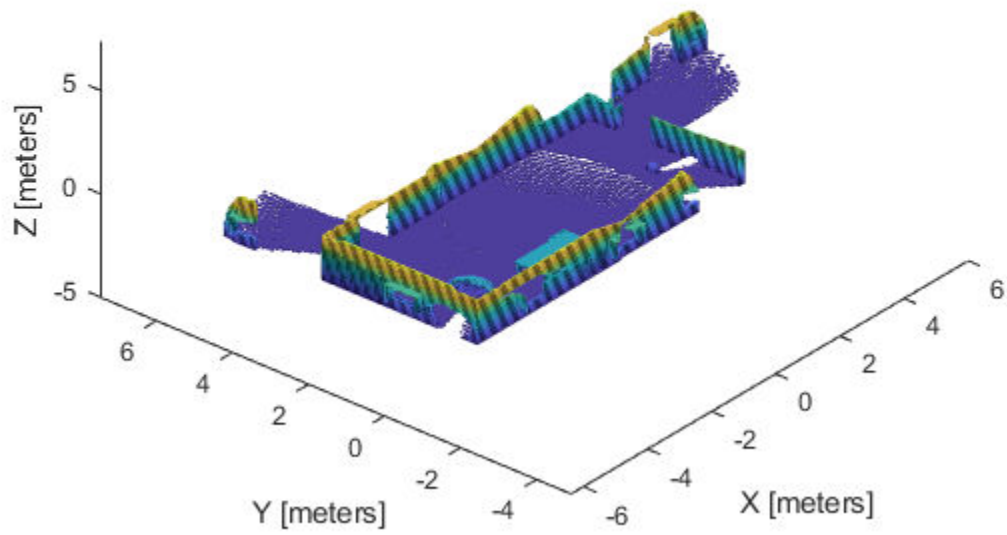
```
MessageType: 'octomap_msgs/Octomap'
Header: [1x1 Header]
Binary: 0
  Id: 'OcTree'
Resolution: 0.0500
  Data: [1175340x1 int8]
```

```
Use showdetails to show the contents of the message
```

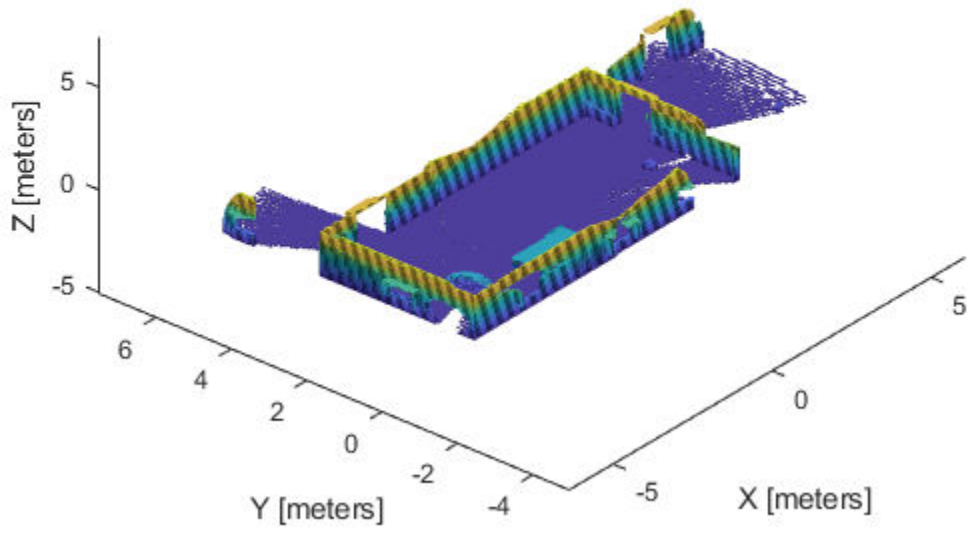
Read the data from the ROS messages into an `occupancyMap3D` object. Display each map.

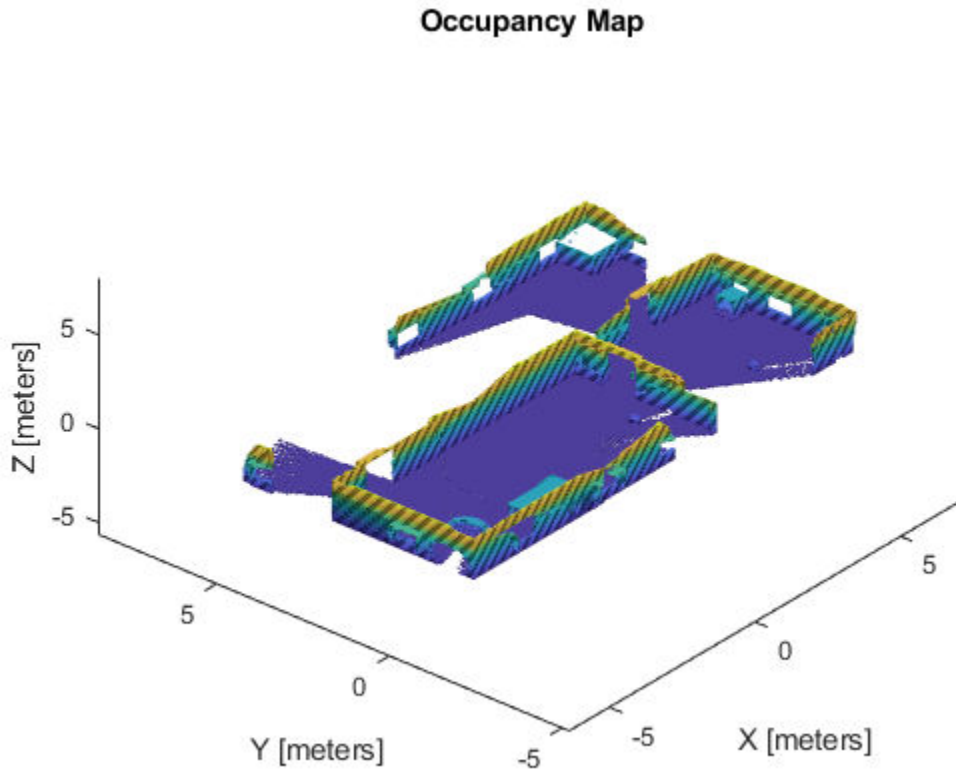
```
for i = 1:length(octomapMsgs)
    msg = octomapMsgs{i};
    map{i} = readOccupancyMap3D(msg);
    figure
    show(map{i});
end
```

Occupancy Map



Occupancy Map





Input Arguments

msg — 'octomap_msgs/Octomap' ROS message

Octomap object handle

'octomap_msgs/Octomap' ROS message, specified as an Octomap object handle. 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 `rosmesssage`.

Output Arguments

map — 3-D occupancy map

occupancyMap3D object handle

3-D occupancy map, returned as an occupancyMap3D object handle.

See Also

occupancyMap3D | rosmesssage | rossubscriber

Introduced in R2018a

removeEdges

Remove loop closure edges from graph

Syntax

```
removeEdges(poseGraph, edgeIDs)
```

Description

`removeEdges(poseGraph, edgeIDs)` removes loop closure edges from the pose graph. Edges that are not loop closures cannot be removed.

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.

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

`addRelativePose` | `edgeConstraints` | `edges` | `findEdgeID` | `nodes` | `optimizePoseGraph`

Objects

`lidarSLAM` | `poseGraph` | `poseGraph3D`

Topics

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

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

Introduced in R2019b

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

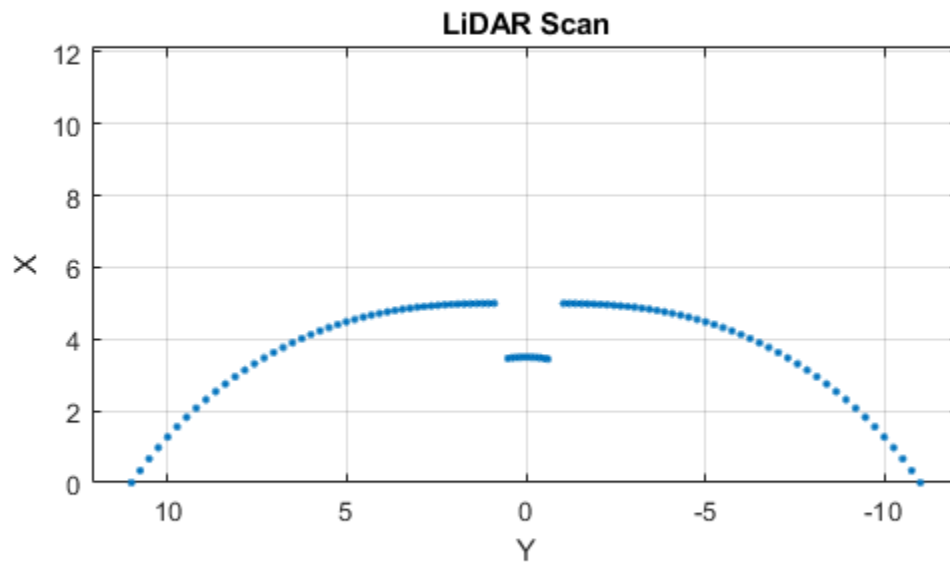
Plot Lidar Scan and Remove Invalid Points

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));
```

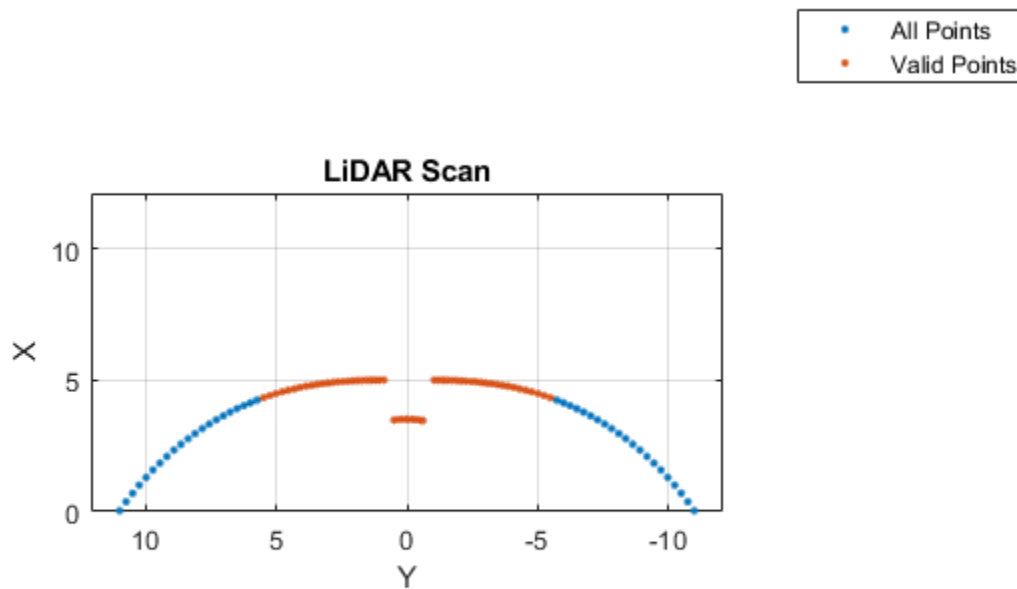
Create a lidar scan by specifying the ranges and angles. Plot all points of the lidar scan.

```
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` | `matchScans` | `transformScan`

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 = 1x4

    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`

Topics

“Coordinate Transformations in Robotics” (Robotics System Toolbox)

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) | "YZ" | "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.
- "YZ" - 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`

Topics

"Coordinate Transformations in Robotics" (Robotics System Toolbox)

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

Topics

“Coordinate Transformations in Robotics” (Robotics System Toolbox)

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 = 4x4
```

```

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

Topics

“Coordinate Transformations in Robotics” (Robotics System Toolbox)

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.

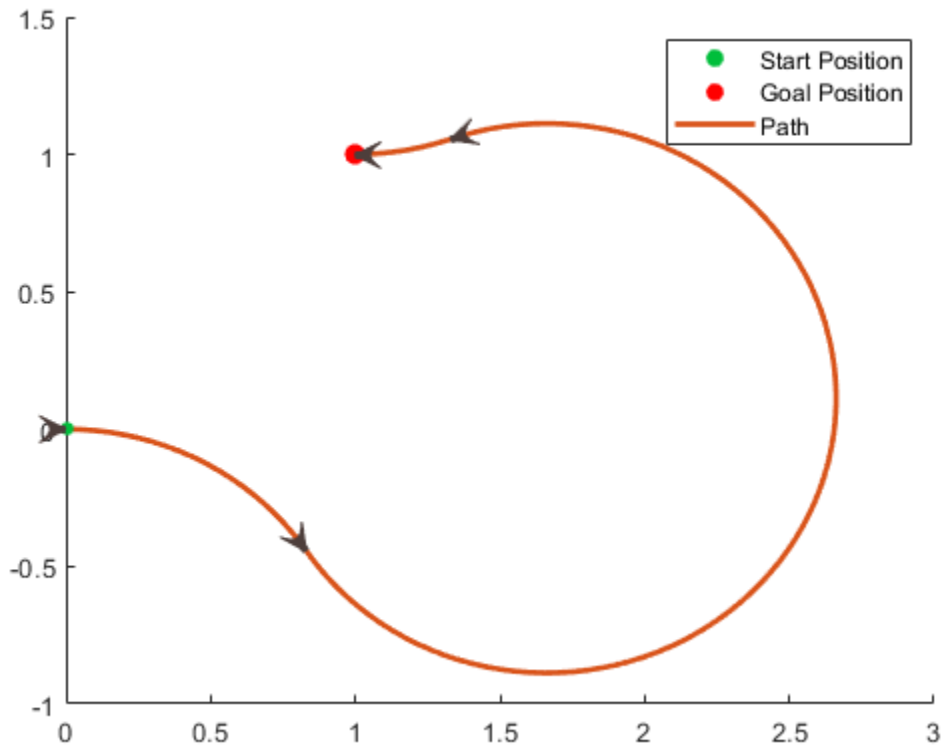
```
startPose = [0 0 0];
goalPose = [1 1 pi];
```

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.

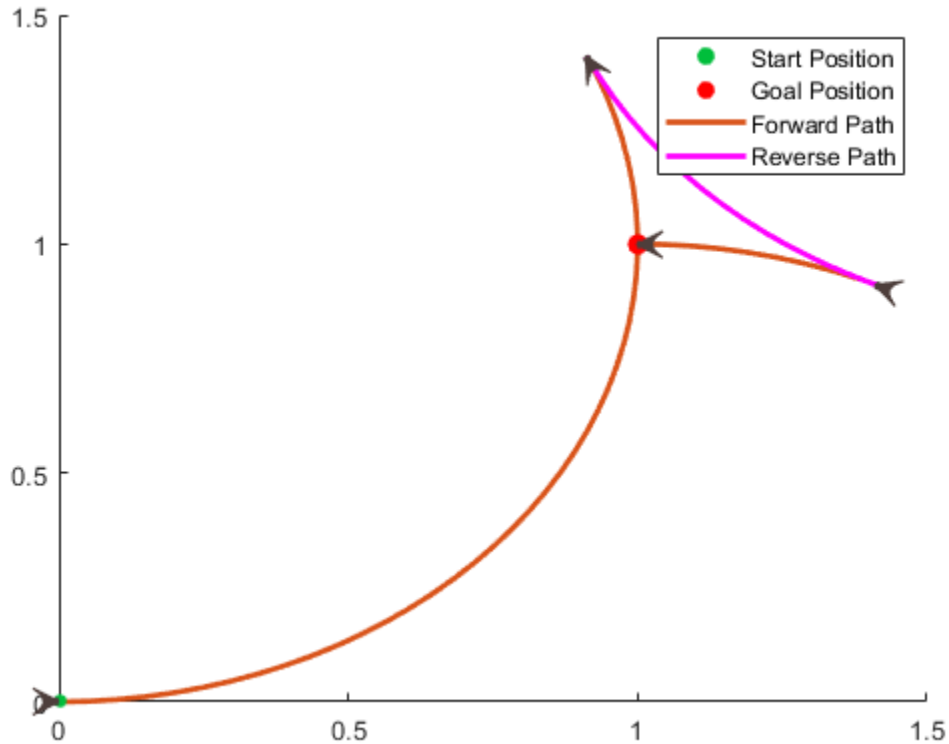
```
startPose = [0 0 0];
goalPose = [1 1 pi];
```

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 = 1x5
      1   -1   1   1   1
```

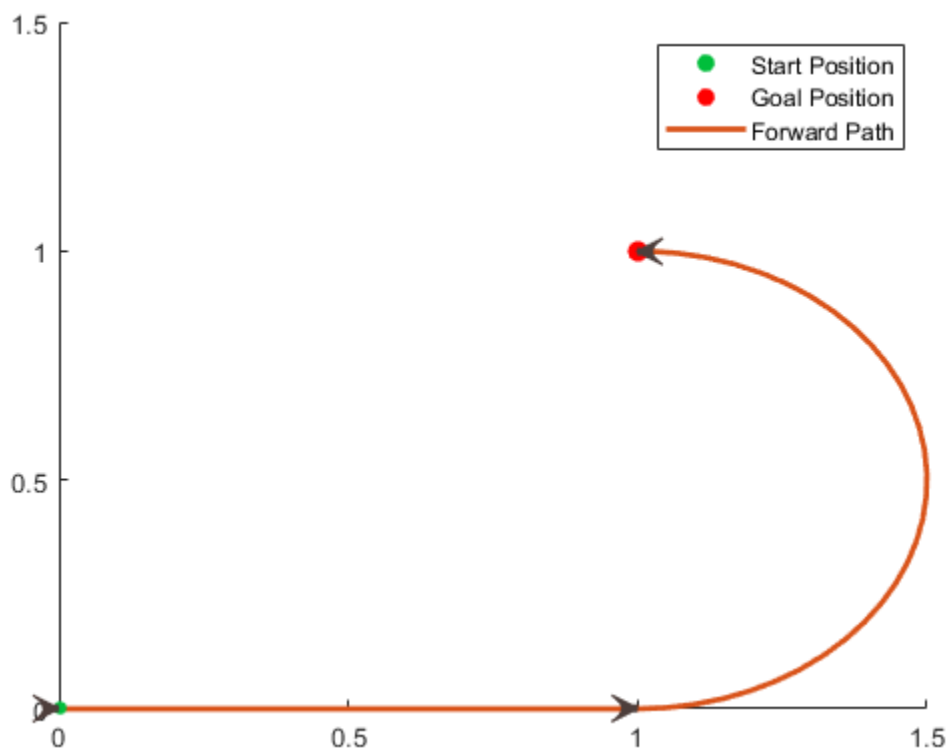
Disable this specific motion sequence in a new connection object. Reduce the `MinTurningRadius` if the robot is more maneuverable. Increase the reverse cost to reduce the likelihood of reverse directions being used. Connect the poses again to get a different path.

```
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.

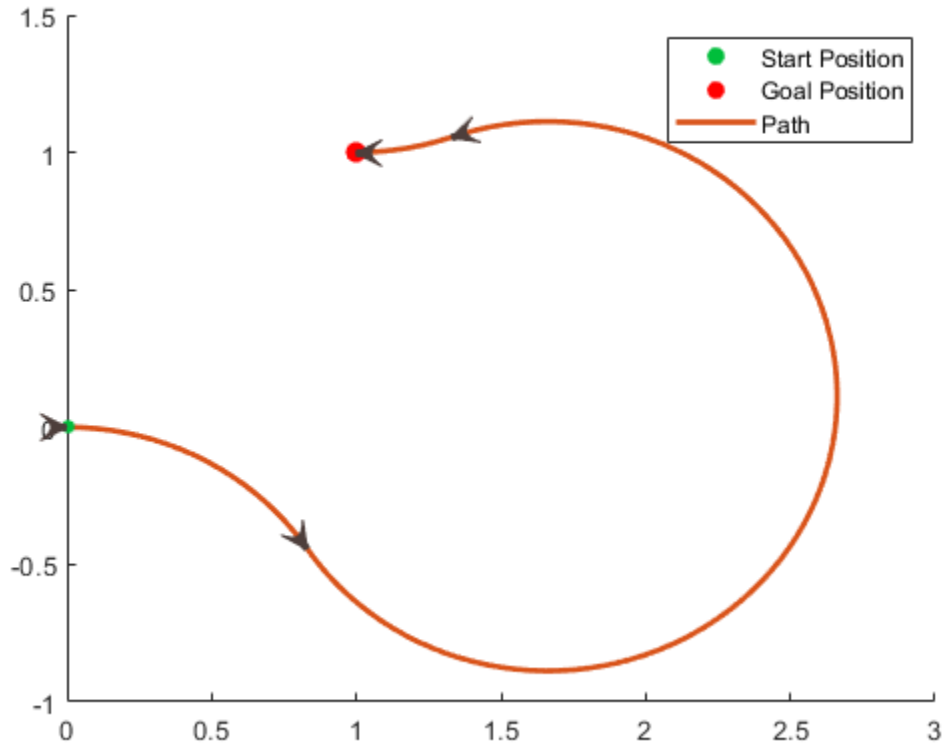
```
startPose = [0 0 0];
goalPose = [1 1 pi];
```

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.

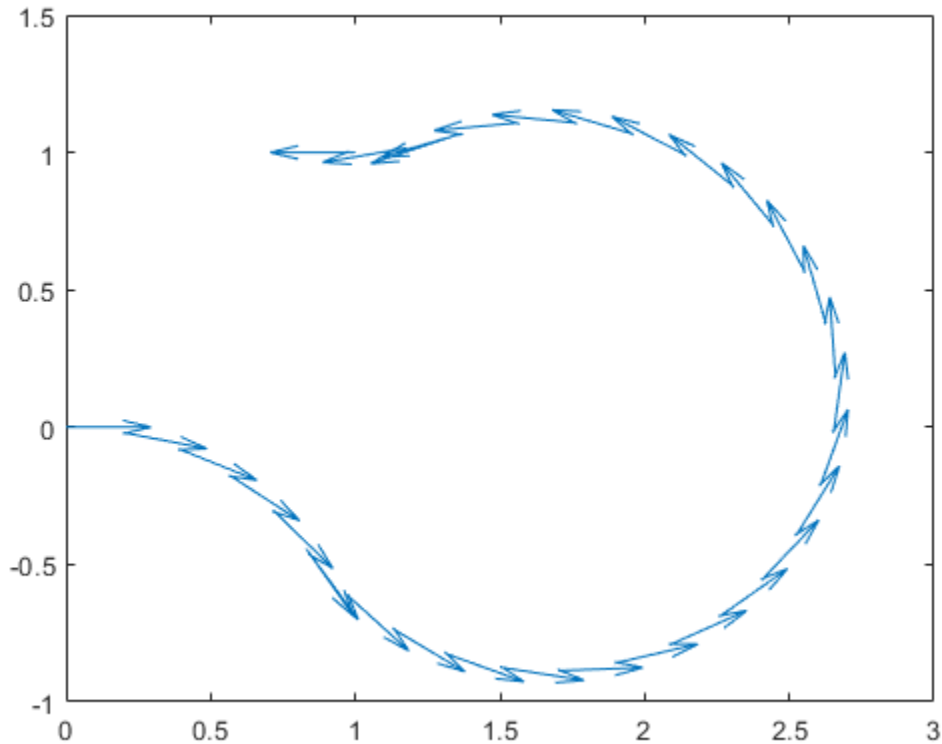
```
length = pathSegObj{1}.Length;
poses = interpolate(pathSegObj{1},0:0.2:length)
```

```
poses = 32x3
```

	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)
```



Error using matlab.internal.editor.figure.SerializedFigureState/serialize
Error while evaluating Figure SizeChangedFcn.

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.

See Also**Functions**

connect | interpolate

Objects

dubinsConnection | dubinsPathSegment | reedsSheppConnection | reedsSheppPathSegment

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.

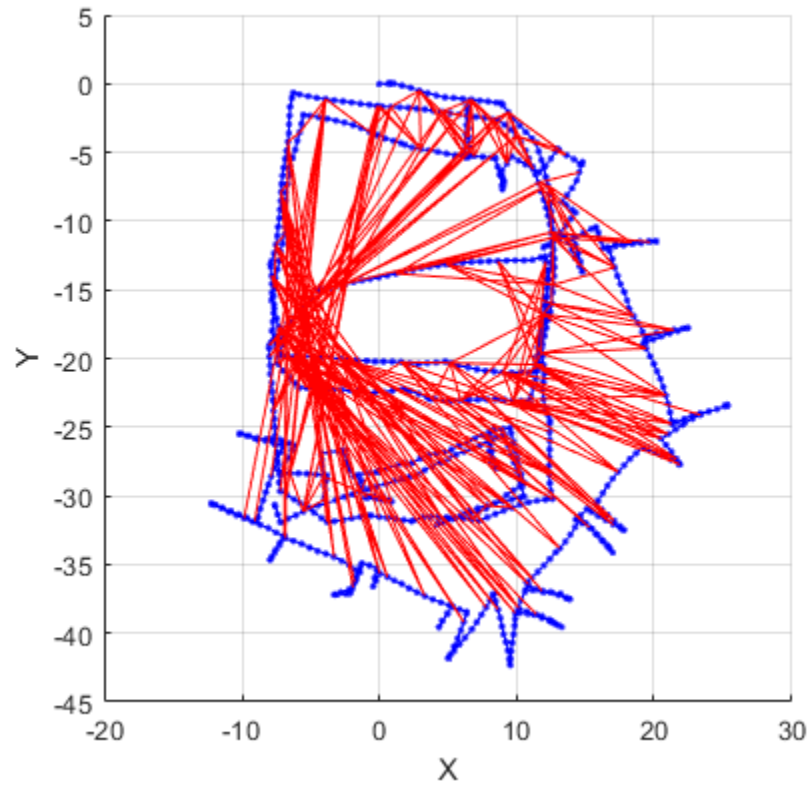
Load the Intel data set that contains a 2-D pose graph. Inspect the `poseGraph` object to view the number of nodes and loop closures.

```
load intel-2d-posegraph.mat pg
disp(pg)

poseGraph with properties:
    NumNodes: 1228
    NumEdges: 1483
    NumLoopClosureEdges: 256
    LoopClosureEdgeIDs: [1x256 double]
```

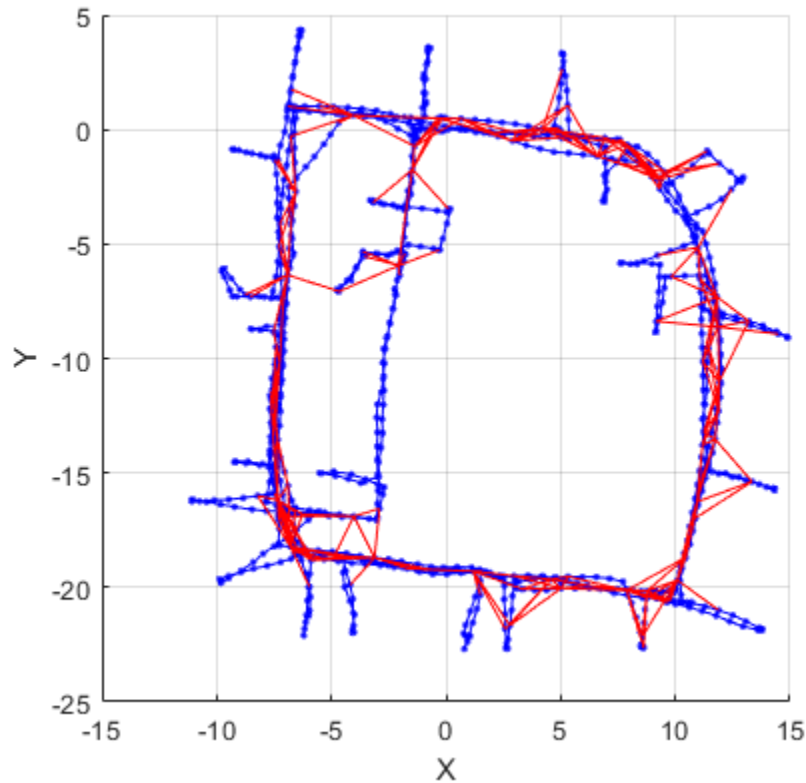
Plot the pose graph with IDs off. Red lines indicate loop closures identified in the dataset.

```
title('Original Pose Graph')
show(pg,'IDs','off');
```



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');
```



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.

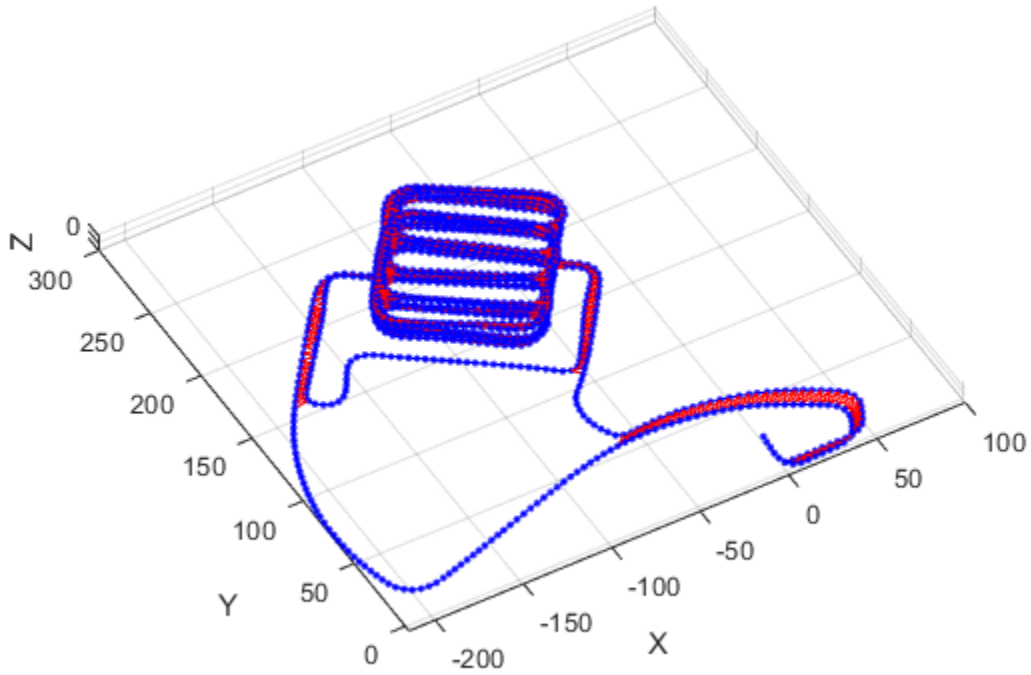
Load the pose graph from the MIT dataset. Inspect the `poseGraph3D` object to view the number of nodes and loop closures.

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

poseGraph3D with properties:
    NumNodes: 1661
    NumEdges: 6275
    NumLoopClosureEdges: 4615
    LoopClosureEdgeIDs: [1x4615 double]
```

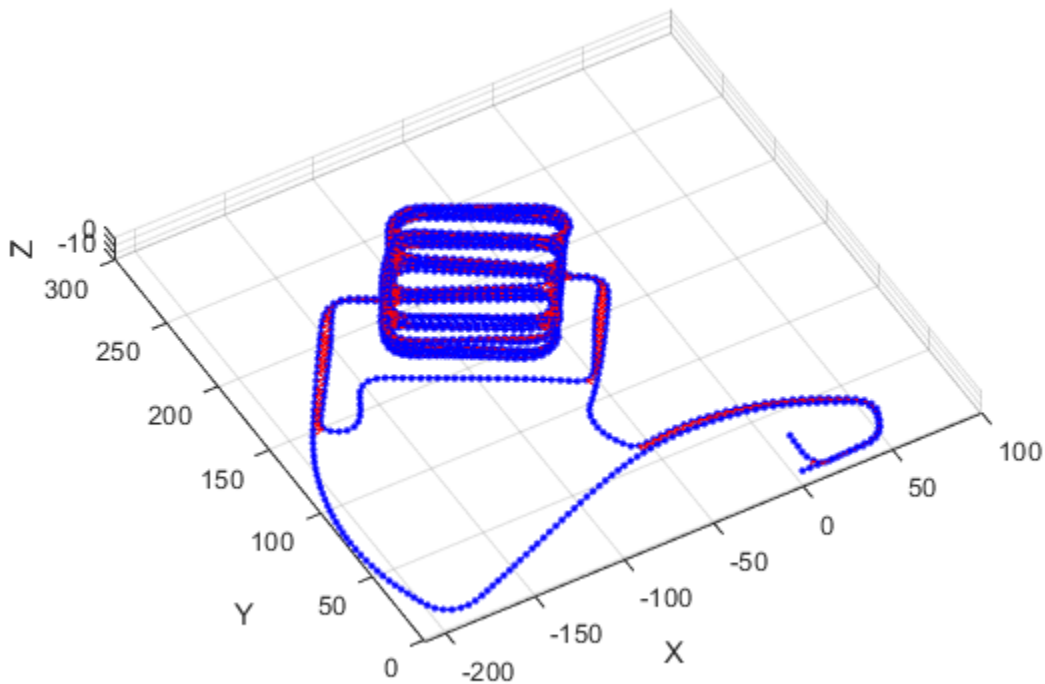
Plot the pose graph with IDs off. Red lines indicate loop closures identified in the dataset.

```
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' | 'off'

Display of IDs on pose graph, specified as the comma-separated pair consisting of 'IDs' and one of the following:

- 'all' — Plot node and edge IDs.
- 'nodes' — Plot node IDs.
- 'loopclosures' — Plot loop closure 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

addRelativePose | optimizePoseGraph

Objects

lidarSLAM | poseGraph | poseGraph3D

Topics

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

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

Introduced in R2019b

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

Topics

“Coordinate Transformations in Robotics” (Robotics System Toolbox)

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) | "YZZ" | "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.
- "YZZ" - 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`

Topics

“Coordinate Transformations in Robotics” (Robotics System Toolbox)

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

Topics

“Coordinate Transformations in Robotics” (Robotics System Toolbox)

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`

Topics

“Coordinate Transformations in Robotics” (Robotics System Toolbox)

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

Topics

“Coordinate Transformations in Robotics” (Robotics System Toolbox)

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

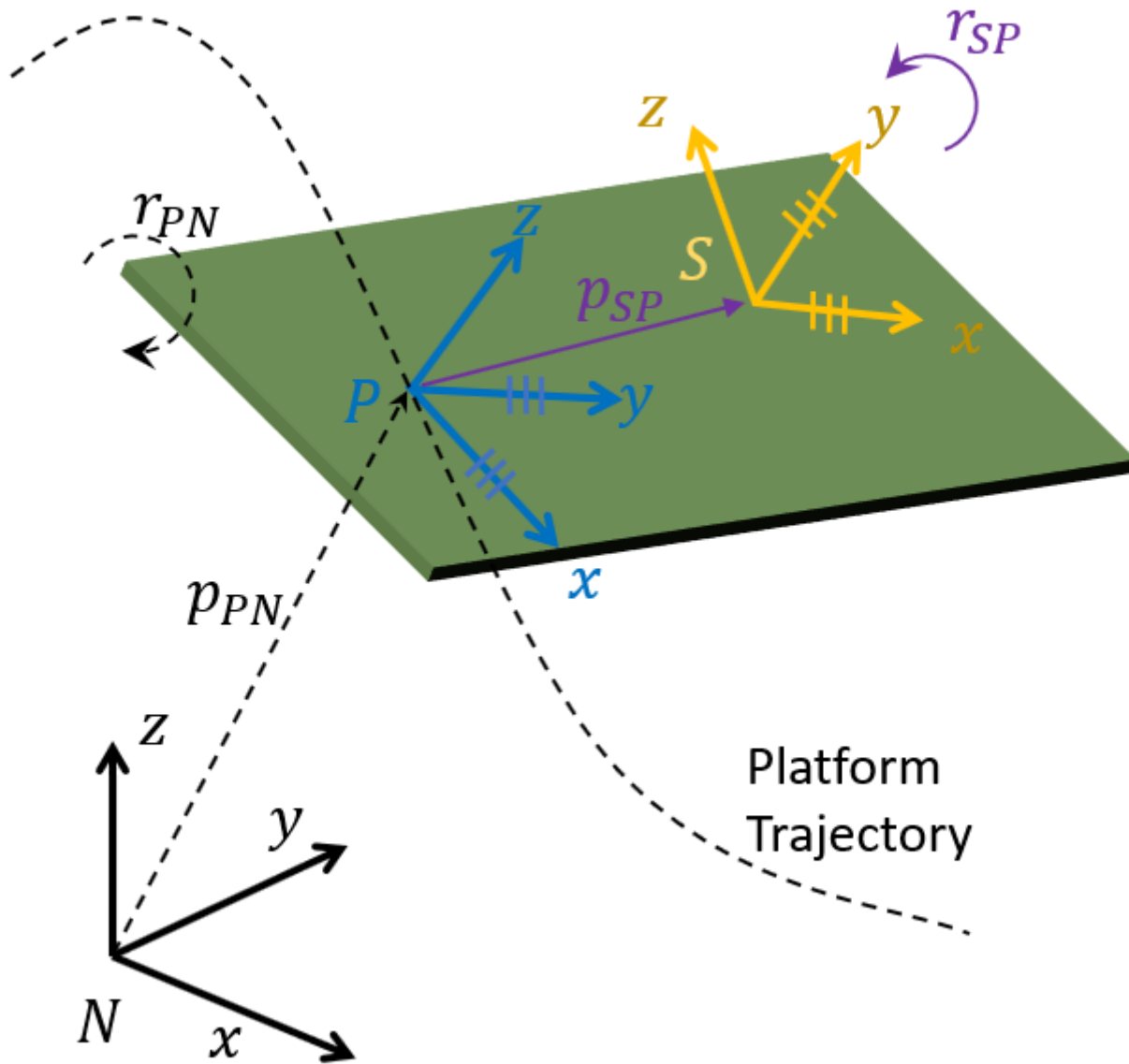
Angular velocity of sensor frame relative to navigation frame, returned as an *N*-by-3 matrix of real scalars. *N* is the number of position quantities specified by the angvelP input.

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.

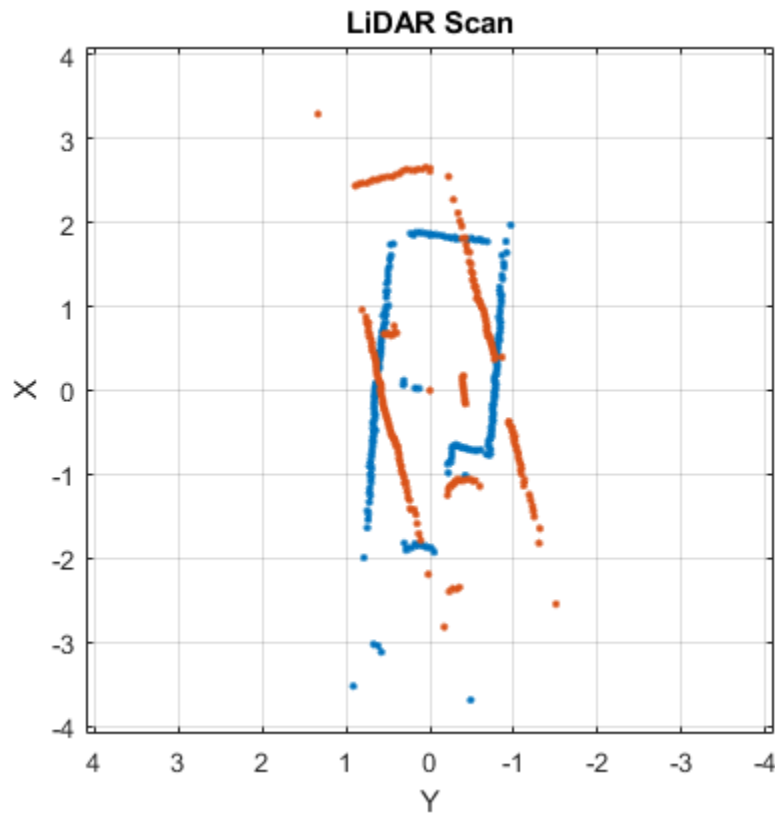
```
rotateScan = transformScan(refScan,[0,0,deg2rad(20)]);
```

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
```

```
-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);
```

```
pose = 1x3
```

```
-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.

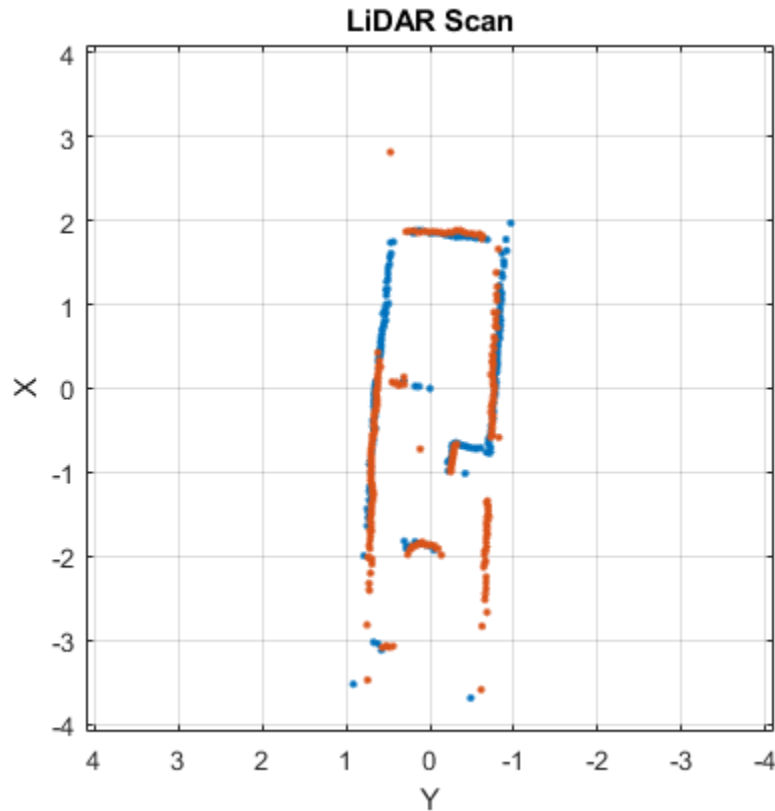
```
s2Transformed = transformScan(s2,pose);
```

```
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

[controllerVFH](#) | [matchScans](#) | [monteCarloLocalization](#) | [transformScan](#)

Topics

“Estimate Robot Pose with Scan Matching”

Introduced in R2017a

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

tform2tvec

Topics

“Coordinate Transformations in Robotics” (Robotics System Toolbox)

Introduced in R2015a

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`.

Examples

Write Binary Occupancy Map Information to ROS Message

Create occupancy grid and message. Write the map onto the message.

```
map = binaryOccupancyMap(randi([0,1], 10));  
msg = rosmesssage('nav_msgs/OccupancyGrid');  
writeBinaryOccupancyGrid(msg, 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 indicates 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

`binaryOccupancyMap` | `controllerVFH` | `occupancyMap`

Introduced in R2015a

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

[OccupancyGrid](#) | [binaryOccupancyMap](#) | [occupancyMap](#) | [readBinaryOccupancyGrid](#) | [writeBinaryOccupancyGrid](#)

Introduced in R2016b

Classes

accelparams class

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`.

Construction

`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/s²)

`inf` (default) | real positive scalar

Maximum sensor reading in m/s², specified as a real positive scalar.

Data Types: `single` | `double`

Resolution — Resolution of sensor measurements ((m/s²)/LSB)

`0` (default) | real nonnegative scalar

Resolution of sensor measurements in (m/s²)/LSB, specified as a real nonnegative scalar.

Data Types: `single` | `double`

ConstantBias — Constant sensor offset bias (m/s²)

`[0 0 0]` (default) | real scalar | real 3-element row vector

Constant sensor offset bias in m/s², 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 (%)

`[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 %, 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`

NoiseDensity — Power spectral density of sensor noise (m/s²/√Hz)

[0 0 0] (default) | real scalar | real 3-element row vector

Power spectral density of sensor noise in (m/s²/√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/s²)

[0 0 0] (default) | real scalar | real 3-element row vector

Instability of the bias offset in m/s², 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/s²)(√Hz))

[0 0 0] (default) | real scalar | real 3-element row vector

Integrated white noise of sensor in (m/s²)(√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/s²)/°C)

[0 0 0] (default) | real scalar | real 3-element row vector

Sensor bias from temperature in (m/s²)/°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/s² and a resolution of 0.598 (mm/s²)/LSB. The constant offset bias is 0.49 m/s². The sensor has a power spectral density of 3920 (μm/s²)/√Hz. The bias from temperature is 0.294 (m/s²)/°C. The scale factor error from temperature is 0.02%/°C. The sensor axes are skewed by 2%.

```
params = accelparams('MeasurementRange',19.6,'Resolution',0.598e-3,'ConstantBias',0.49,'NoiseDen
```

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);
```

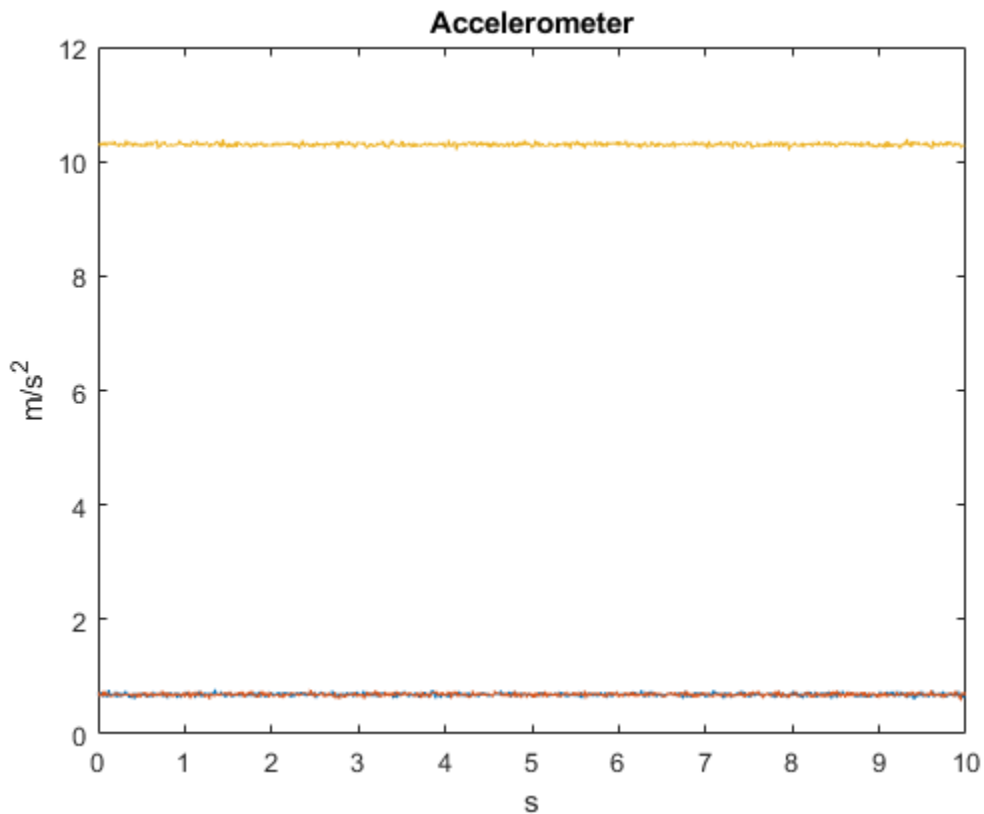
Generate accelerometer data from the imuSensor object.

```
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

System Objects

imuSensor

Classes

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/s²)²)

[1e-4, 1e-4, 1e-4] (default) | scalar | three-element row vector

Multiplicative process noise variance from the accelerometer in (m/s²)², specified as positive real finite numbers.

Data Types: single | double

GyroscopeBiasNoise — Multiplicative process noise variance from gyroscope bias ((rad/s²)²)

[1e-10, 1e-10, 1e-10] (default) | scalar | three-element row vector

Multiplicative process noise variance from the gyroscope bias in (rad/s²)², specified as positive real finite numbers.

Data Types: single | double

AccelerometerBiasNoise — Multiplicative process noise variance from accelerometer bias ((m/s²)²)

[1e-4, 1e-4, 1e-4] (default) | scalar | three-element row vector

Multiplicative process noise variance from the accelerometer bias in (m/s²)², specified as positive real finite numbers.

Data Types: single | double

GeomagneticVectorNoise — Additive process noise for geomagnetic vector (μT²)

[1e-6, 1e-6, 1e-6] (default) | scalar | three-element row vector

Additive process noise for geomagnetic vector in μT², specified as positive real finite numbers.

Data Types: single | double

MagnetometerBiasNoise — Additive process noise for magnetometer bias (μT²)

[0.1, 0.1, 0.1] (default) | scalar | three-element row vector

Additive process noise for magnetometer bias in μT², 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

<code>predict</code>	Update states using accelerometer and gyroscope data
<code>fusemag</code>	Correct states using magnetometer data
<code>fusealtimeter</code>	Correct states using altimeter data
<code>correct</code>	Correct states using direct state measurements
<code>residual</code>	Residuals and residual covariances from direct state measurements
<code>residualmag</code>	Residuals and residual covariance from magnetometer measurements
<code>residualaltimeter</code>	Residuals and residual covariance from altimeter measurements
<code>pose</code>	Current orientation and position estimate
<code>reset</code>	Reset internal states
<code>stateinfo</code>	Display state vector information

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 accelerometer 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)

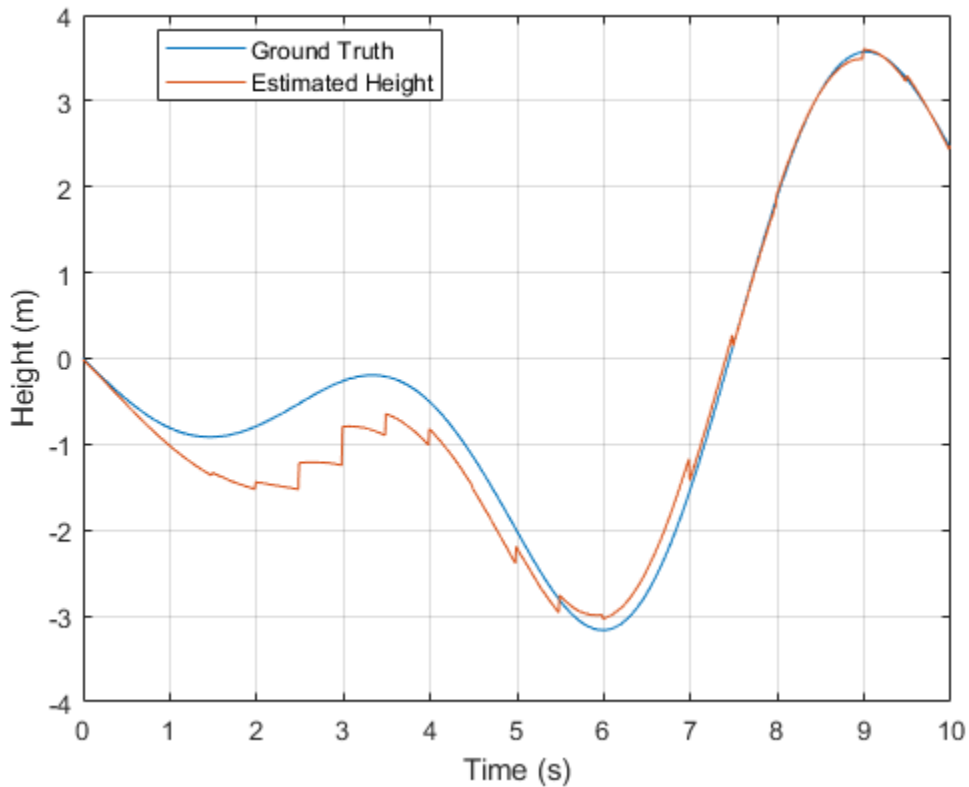
```

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

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 measurementscalar | 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**

fusealtimeter

Correct states using altimeter data

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 (m²)

real scalar

Altimeter readings error covariance in m², 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

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 $(\mu\text{T})^2$.

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

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 — Vertical position estimate in the local NED coordinate system (m)

scalar

Vertical position estimate in the local NED coordinate system in meters, returned as a scalar.

Data Types: single | double

orientation — Orientation estimate in the local NED coordinate system

quaternion (default) | 3-by-3 rotation matrix

Orientation estimate in the local NED coordinate system, returned as a scalar quaternion or 3-by-3 rotation matrix. The quaternion or rotation matrix represents a frame rotation from the local NED reference frame to the body reference frame.

Data Types: single | double | quaternion

velocity — Vertical velocity estimate in the local NED coordinate system (m/s)

scalar

Vertical velocity estimate in the local NED coordinate system in m/s, returned as a scalar.

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

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/s²)

N-by-3 matrix

Accelerometer readings in local sensor body coordinate system in m/s², 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

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

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

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 (m²)

real scalar

Altimeter readings error covariance in m², 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 m².

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

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 $(\mu\text{T})^2$.

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

Syntax

```
stateinfo(FUSE)
```

Description

`stateinfo(FUSE)` displays the meaning of each index of the State property and the associated units.

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

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? \(MATLAB\)](#).

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 \(MATLAB\)](#).

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/s²)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 μT², 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 $(\text{m/s}^2)^2$, 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 `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

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
0	0.000006092348396	0	0	0	0
0	0	0.000006092348396	0	0	0
0	0	0	0.000076154354947	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	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 0.600000000000
0 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/s², 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 `OrientationFormat` 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)
```

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.

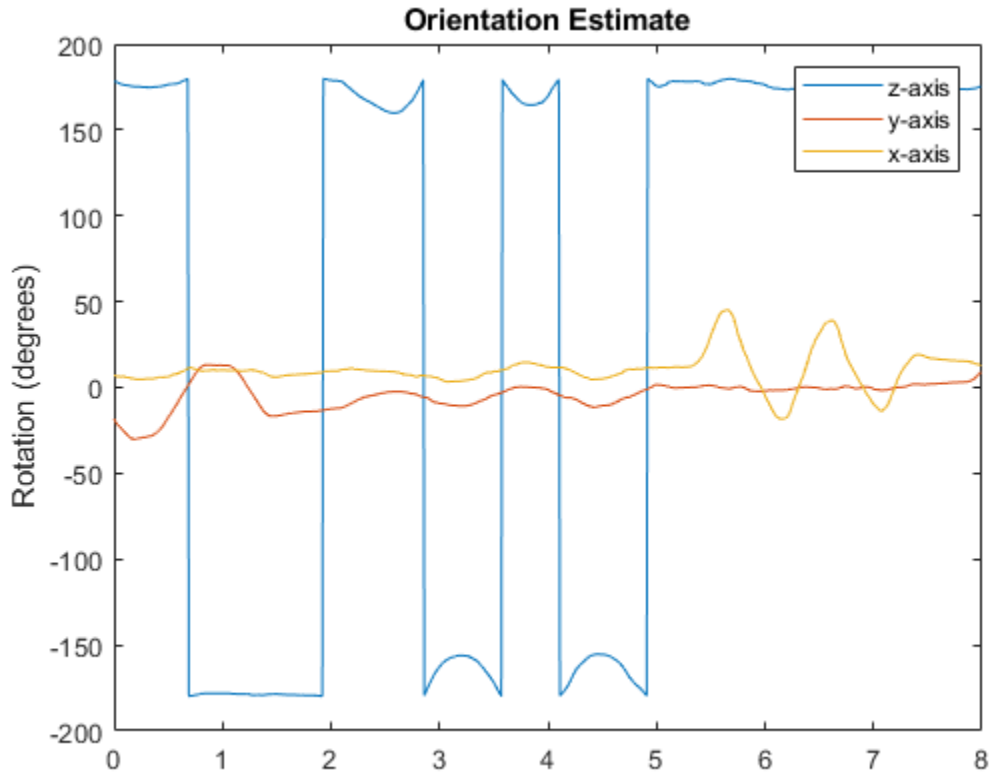
`ahrsfilter` correctly estimates the change in orientation over time, including the south-facing initial orientation.

```
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.

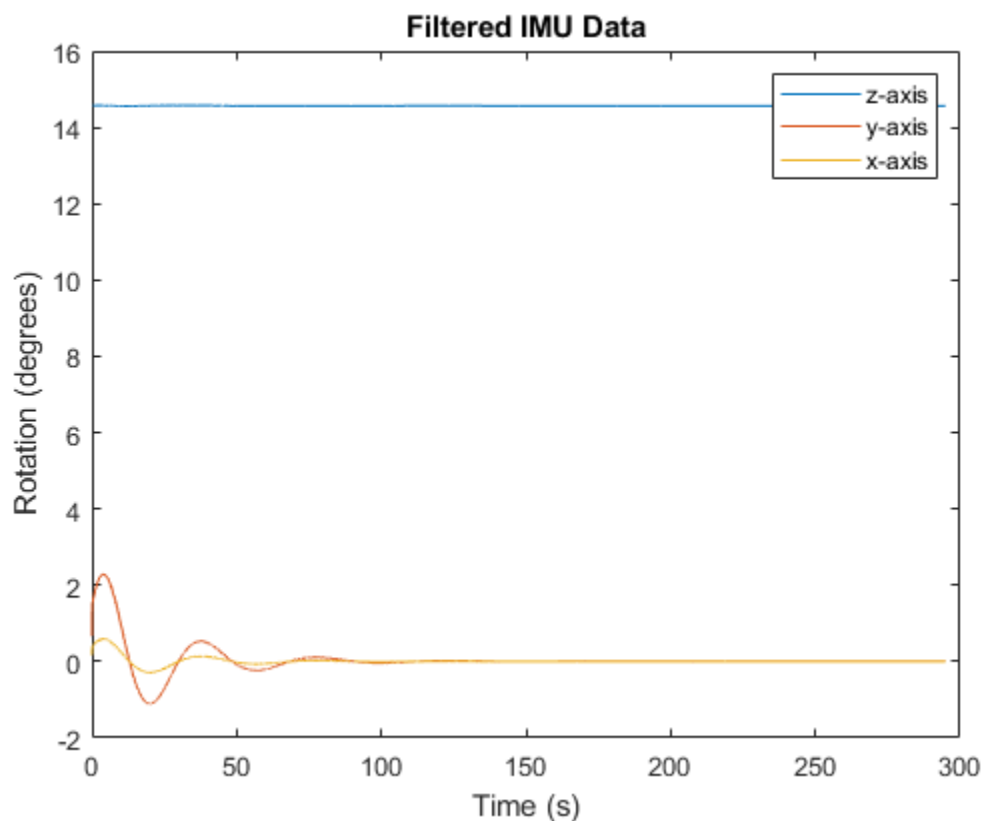
```
decim = 2;
FUSE = ahrsfilter('SampleRate',SampleRate,'DecimationFactor',decim);
```

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.

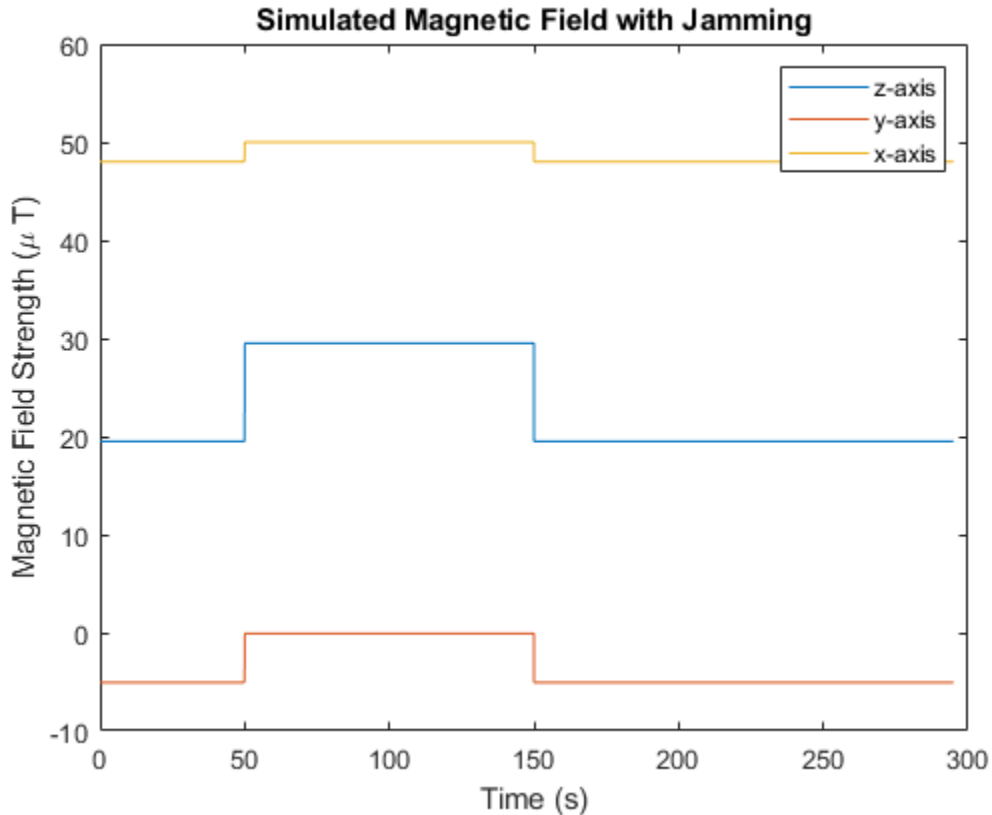
```
jamStrength = [10,5,2];
startStop = (50*SampleRate):(150*SampleRate);
jam = zeros(size(magReadings));
jam(startStop,:) = jamStrength.*ones(numel(startStop),3);
```

```

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.

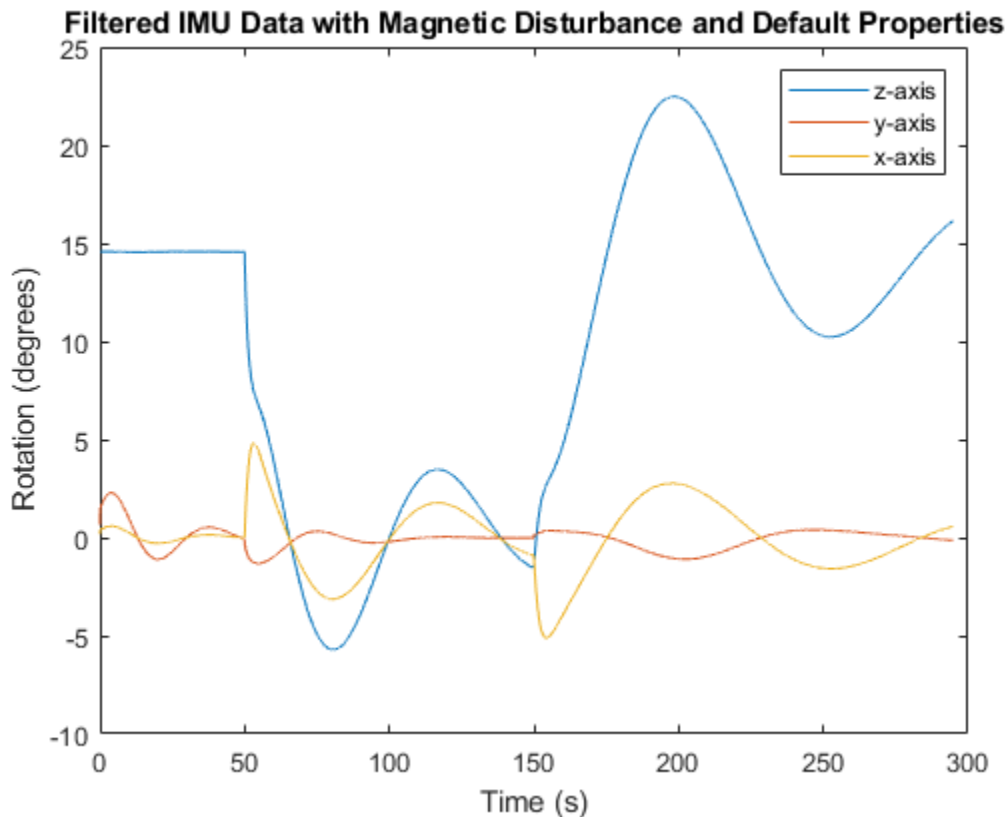
```

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.

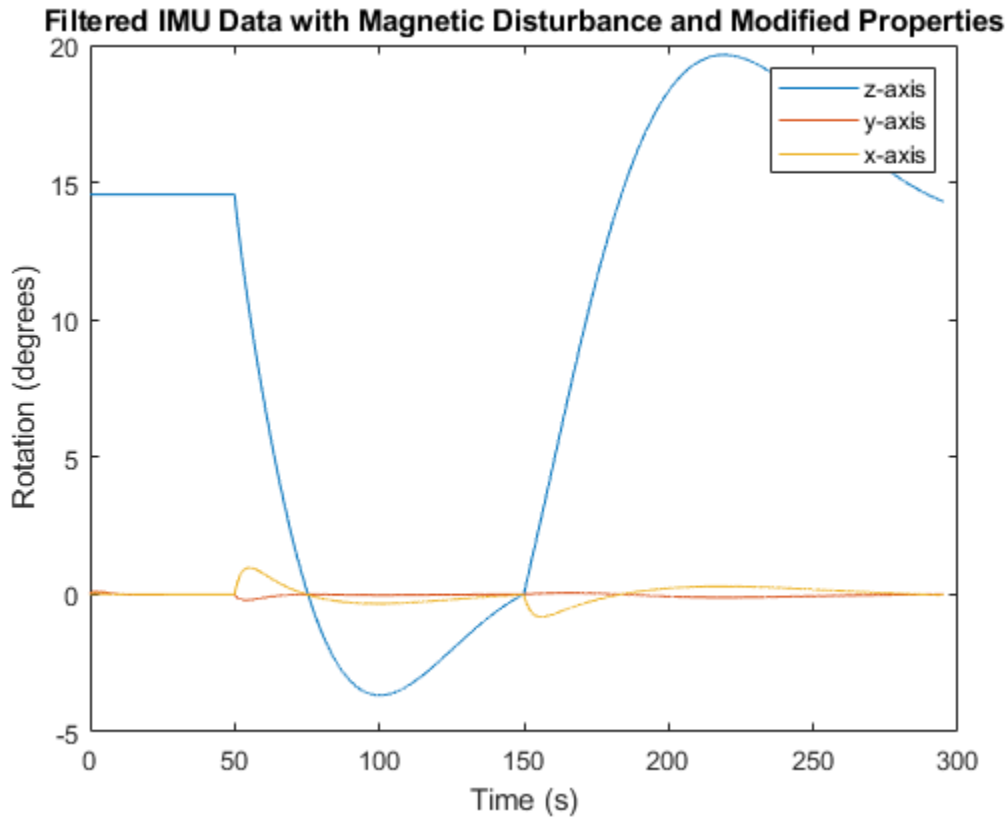
The orientation estimation output from `ahrsfilter` is more accurate and less affected by the magnetic transient. However, because the magnetometer signal is weighted less in the underlying fusion algorithm, the algorithm may take more time to restabilize.

```
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')
```

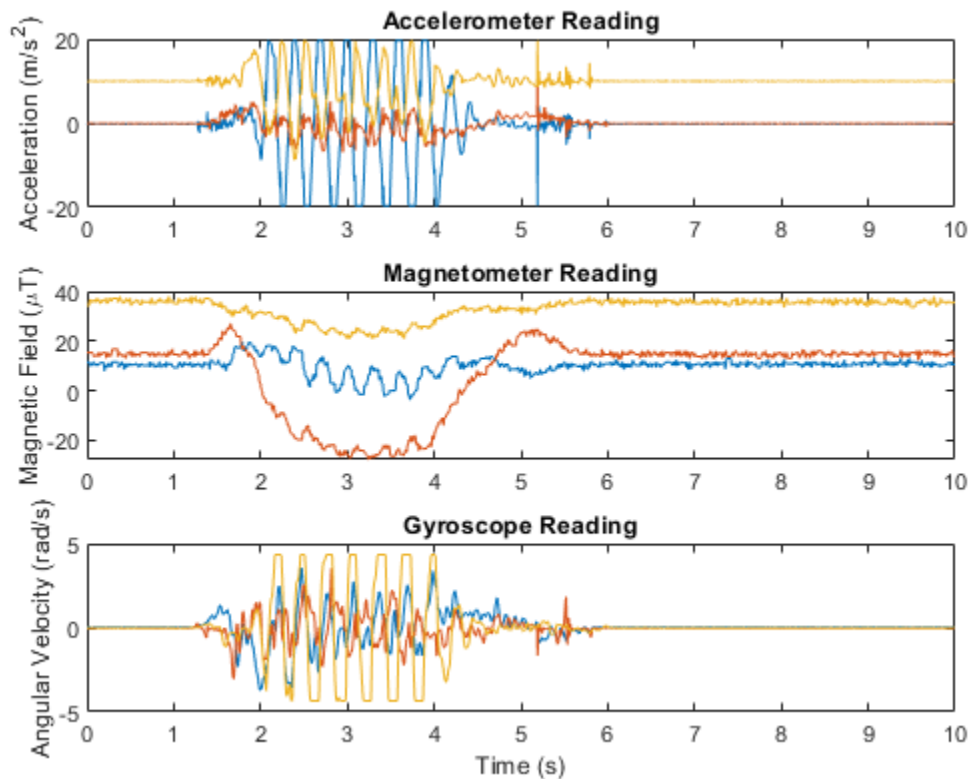


```

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.

```

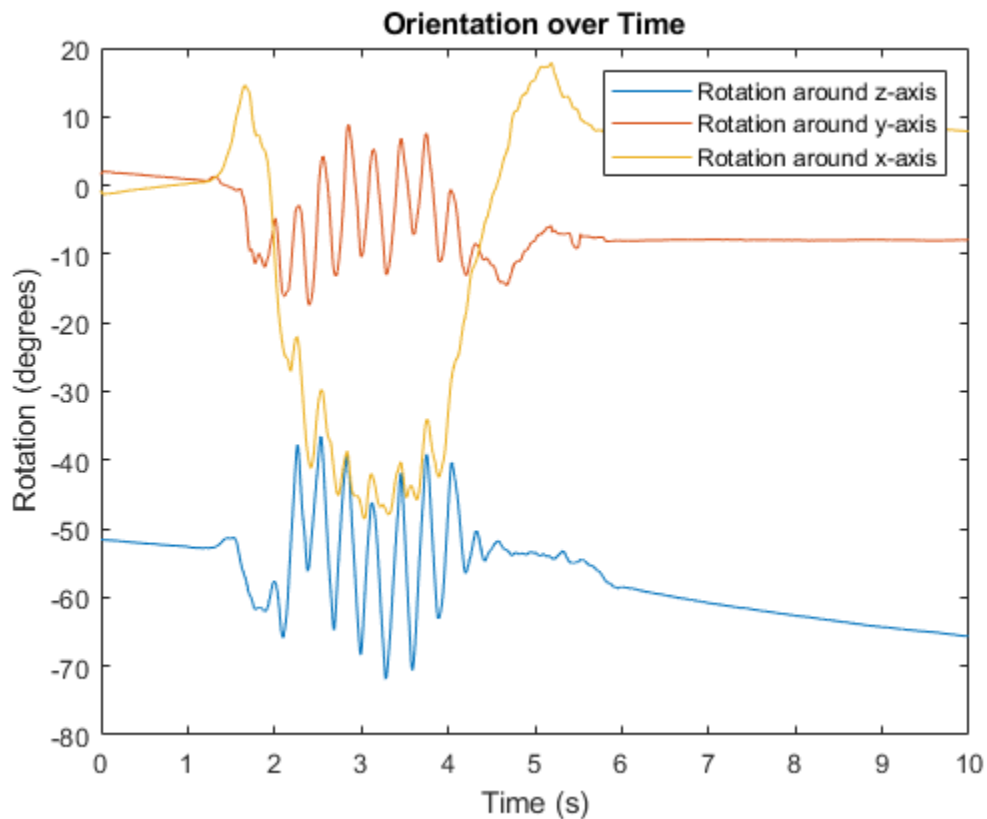
fuse = ahrsfilter('SampleRate',SampleRate);
orientation = fuse(accelReadings,gyroReadings,magReadings);

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')
```



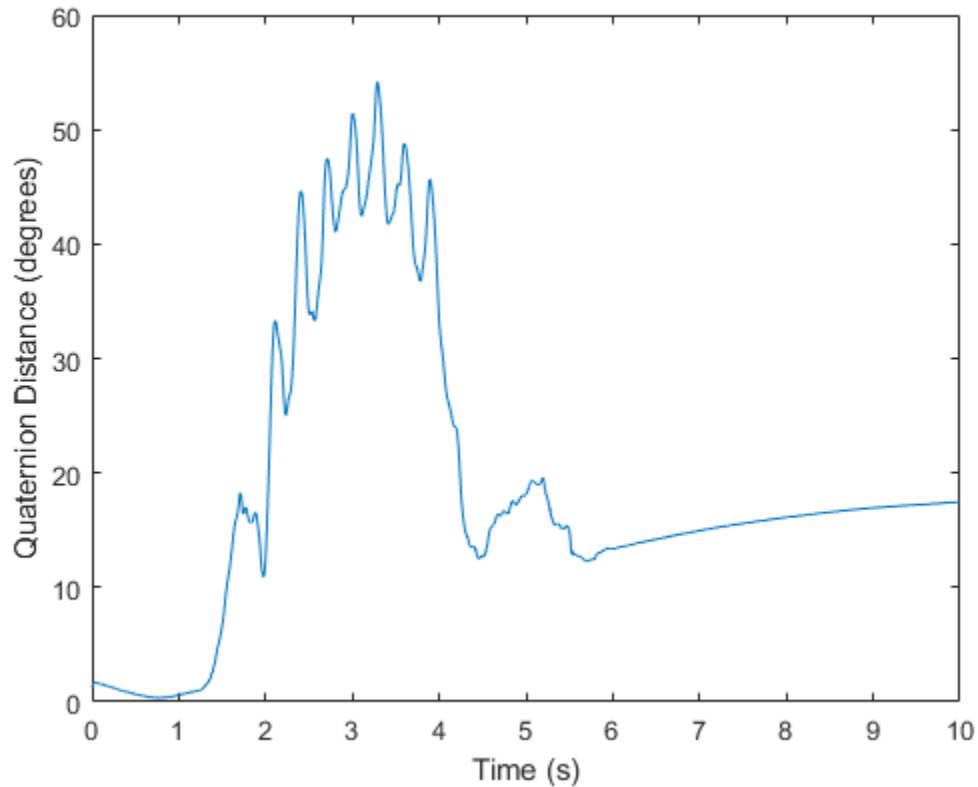
In the IMU recording, the shaking stops after approximately six seconds. Determine the resting orientation so that you can characterize how fast the `ahrsfilter` converges.

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)')
```



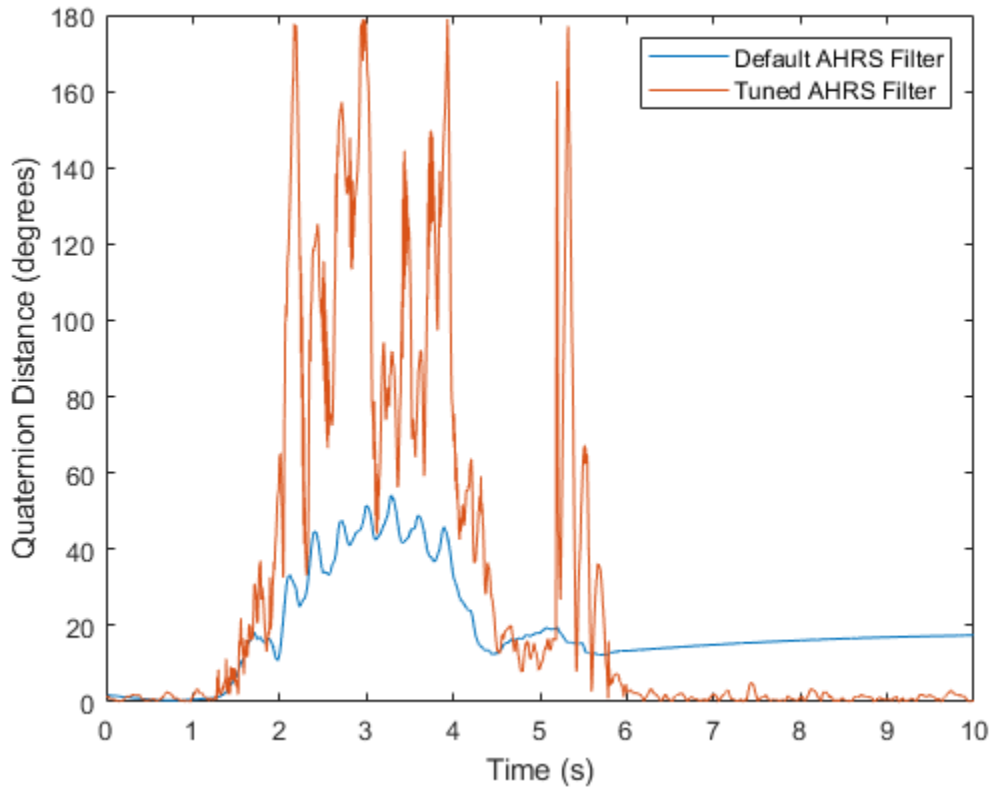
Modify the default `ahrsfilter` properties so that the filter converges to gravity more quickly. Increase the `GyroscopeDriftNoise` to $1e-2$ and decrease the `LinearAccelerationNoise` to $1e-4$. This instructs the `ahrsfilter` algorithm to weigh gyroscope data less and accelerometer data more. Because 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] (Sensor Fusion and Tracking Toolbox). 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:

$$\begin{aligned}x_k^- &= F_k x_{k-1}^+ \\P_k^- &= F_k P_{k-1}^+ F_k^T + Q_k \\y_k &= z_k - H_k x_k^- \\S_k &= R_k + H_k P_k^- H_k^T \\K_k &= P_k^- H_k^T (S_k)^{-1} \\x_k^+ &= x_k^- + K_k y_k \\P_k^+ &= P_k^- - K_k H_k P_k^-\end{aligned}$$

Kalman equations used in this algorithm:

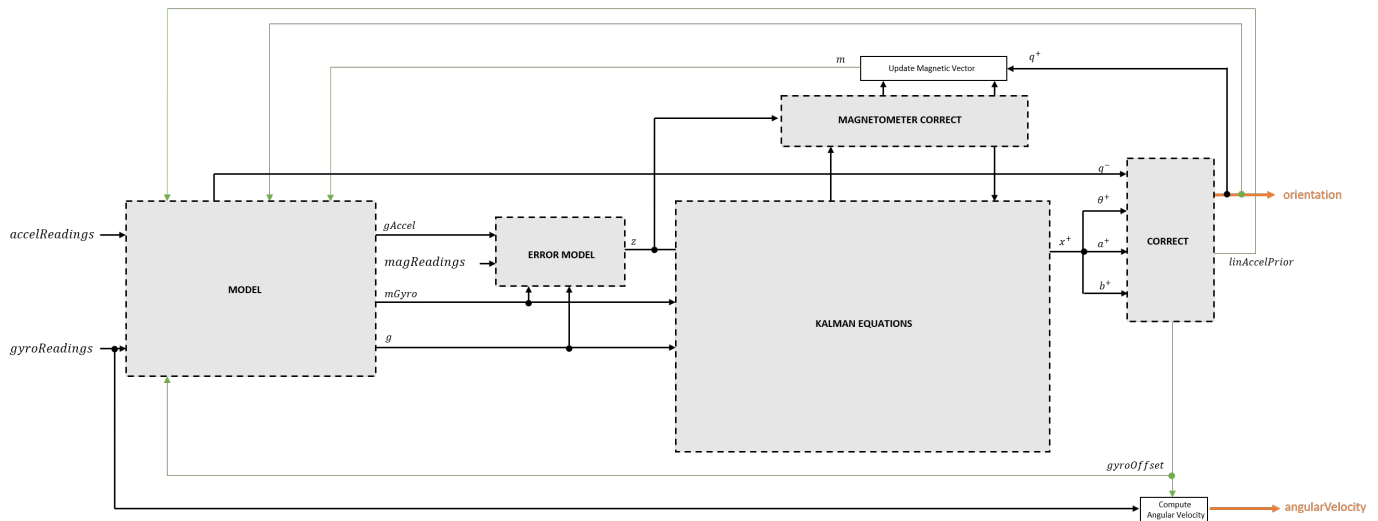
$$\begin{aligned}x_k^- &= 0 \\P_k^- &= Q_k \\y_k &= z_k \\S_k &= R_k + H_k P_k^- H_k^T \\K_k &= P_k^- H_k^T (S_k)^{-1} \\x_k^+ &= K_k y_k \\P_k^+ &= P_k^- - K_k H_k P_k^-\end{aligned}$$

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.

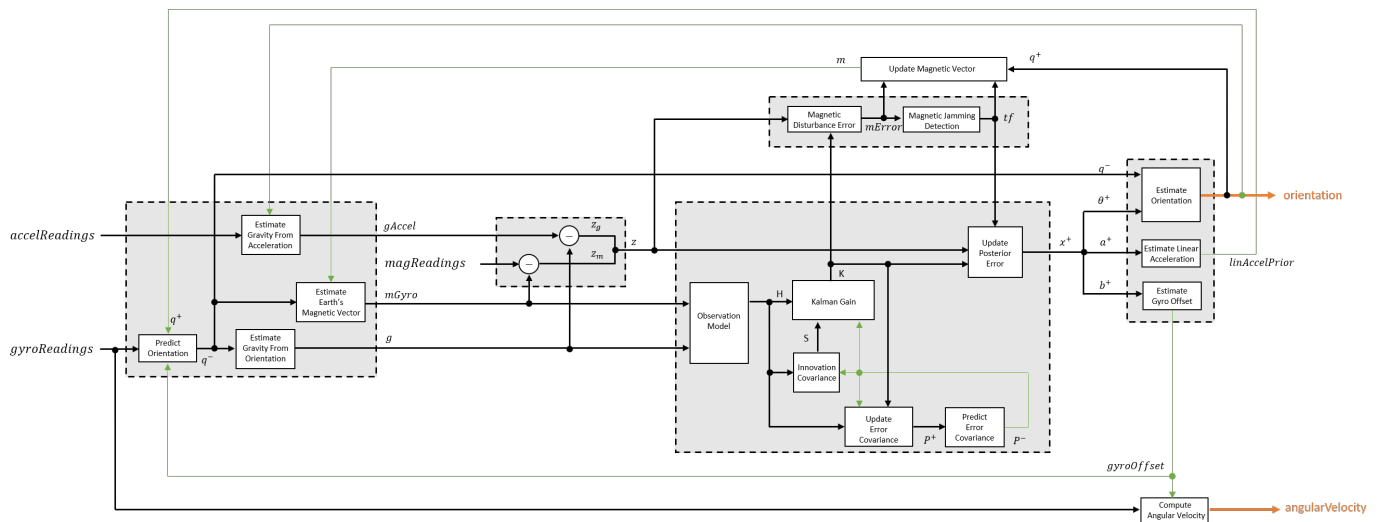
The graphic and following steps describe a single frame-based iteration through the algorithm.



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.

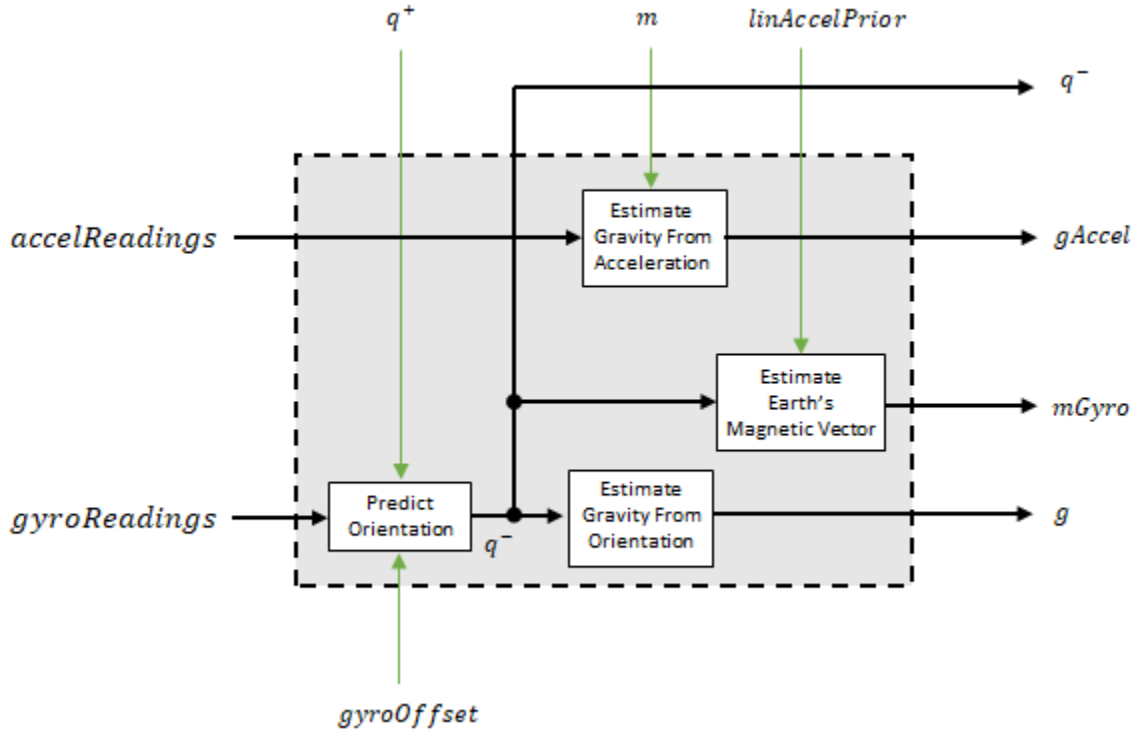
Detailed Overview

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:

$$\Delta\varphi_{N \times 3} = \frac{(\text{gyroReadings}_{N \times 3} - \text{gyroOffset}_{1 \times 3})}{fs}$$

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 :

$$q_{1 \times 1}^- = (q_{1 \times 1}^+) \left(\prod_{n=1}^N \Delta Q_n \right)$$

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] (Sensor Fusion and Tracking Toolbox) 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:

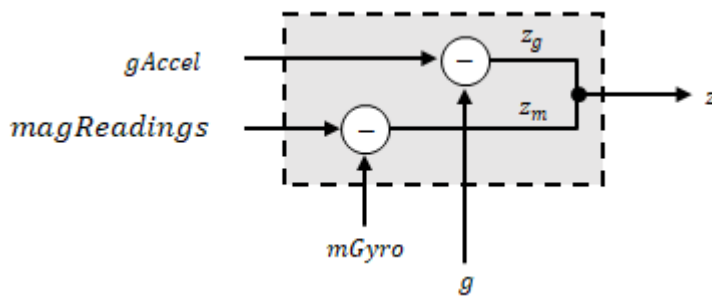
$$gAccel_{1 \times 3} = accelReadings_{1 \times 3} - linAccelPrior_{1 \times 3}$$

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:

$$mGyro_{1 \times 3} = ((rPrior)(m^T))^T$$

Error Model

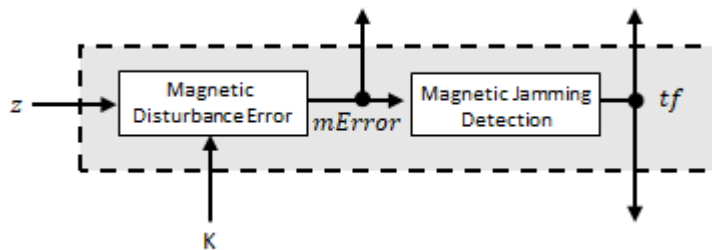


The error model combines two differences:

- 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

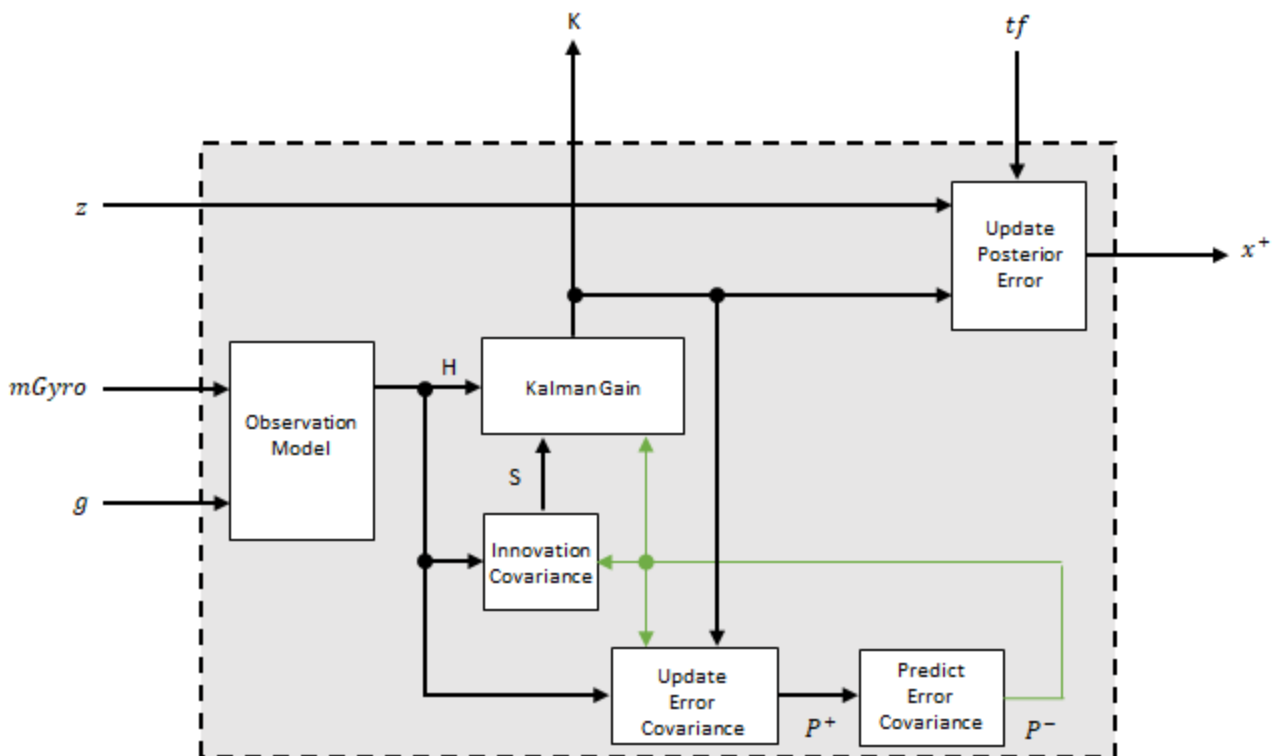
Magnetic jamming is determined by verifying that the power of the detected magnetic disturbance is less than or equal to four times the power of the expected magnetic field strength:

$$tf = \begin{cases} \text{true} & \text{if } \sum |mError|^2 > (4)(\text{ExpectedMagneticFieldStrength})^2 \\ \text{false} & \text{else} \end{cases}$$

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: $\kappa = \text{DecimationFactor}/\text{SampleRate}$.

See sections 7.3 and 7.4 of [1] (Sensor Fusion and Tracking Toolbox) 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:

$$S_{6 \times 6} = R_{6 \times 6} + (H_{6 \times 12})(P_{12 \times 12}^-)(H_{6 \times 12})^T$$

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

$$accel_{\text{noise}} = \text{AccelerometerNoise} + \text{LinearAccelerationNoise} + \kappa^2 (\text{GyroscopeDriftNoise} + \text{GyroscopeNoise})$$

and

$$mag_{\text{noise}} = \text{MagnetometerNoise} + \text{MagneticDisturbanceNoise} + \kappa^2 (\text{GyroscopeDriftNoise} + \text{GyroscopeNoise})$$

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:

$$P_{12 \times 12}^+ = P_{12 \times 12}^- - (K_{12 \times 6})(H_{6 \times 12})(P_{12 \times 12}^-)$$

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:

$Q =$

$$\begin{array}{cccccc}
 P^+(1) + \kappa^2 P^+(40) + \beta + \eta & 0 & 0 & -\kappa(P^+(40) + \beta) & 0 \\
 0 & P^+(14) + \kappa^2 P^+(53) + \beta + \eta & 0 & 0 & -\kappa(P^+(53) + \beta) \\
 0 & 0 & P^+(27) + \kappa^2 P^+(66) + \beta + \eta & 0 & 0 \\
 -\kappa(P^+(40) + \beta) & 0 & 0 & P^+(40) + \beta & 0 \\
 0 & -\kappa(P^+(53) + \beta) & 0 & 0 & P^+(53) + \beta \\
 0 & 0 & -\kappa(P^+(66) + \beta) & 0 & 0 \\
 0 & 0 & 0 & 0 & 0 \\
 0 & 0 & 0 & 0 & 0 \\
 0 & 0 & 0 & 0 & 0 \\
 0 & 0 & 0 & 0 & 0 \\
 0 & 0 & 0 & 0 & 0 \\
 0 & 0 & 0 & 0 & 0 \\
 0 & 0 & 0 & 0 & 0
 \end{array}$$

where

- P^+ -- is the updated (*a posteriori*) error estimate covariance
- κ -- DecimationFactor/SampleRate
- β -- GyroscopeDriftNoise
- η -- GyroscopeNoise
- ν -- LinearAccelerationDecayFactor
- ξ -- LinearAccelerationNoise
- σ -- MagneticDisturbanceDecayFactor
- γ -- MagneticDisturbanceNoise

See section 10.1 of [1] (Sensor Fusion and Tracking Toolbox) 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:

$$K_{12 \times 6} = (P_{12 \times 12}^-)(H_{6 \times 12})^T((S_{6 \times 6})^T)^{-1}$$

where

- P^- -- predicted error covariance
- H -- observation model
- S -- innovation covariance

Update a Posteriori Error

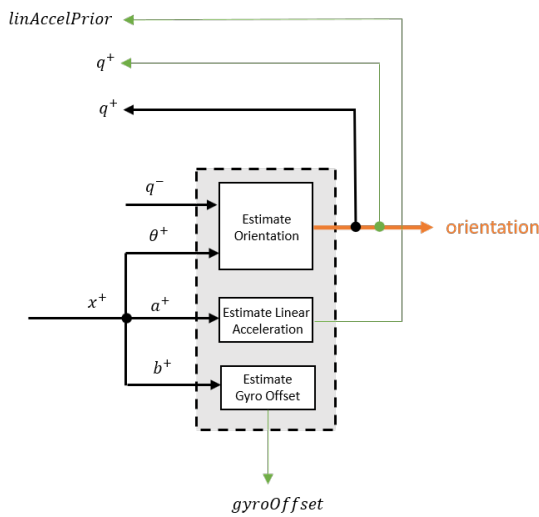
The *a posteriori* error estimate is determined by combining the Kalman gain matrix with the error in the gravity vector and magnetic vector estimations:

$$x_{12 \times 1} = (K_{12 \times 6})(z_{1 \times 6})^T$$

If magnetic jamming is detected in the current iteration, the magnetic vector error signal is ignored, and the *a posteriori* error estimate is calculated as:

$$x_{9 \times 1} = (K(1:9, 1:3))(z_g)^T$$

Correct



Estimate Orientation

The orientation estimate is updated by multiplying the previous estimation by the error:

$$q^+ = (q^-)(\theta^+)$$

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 -- LinearAccelerationDecayFactor

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:

$$m(1) = (\text{ExpectedMagneticFieldStrength})(\cos(\text{inclination}))$$

$$m(2) = 0$$

$$m(3) = (\text{ExpectedMagneticFieldStrength})(\sin(\text{inclination}))$$

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` | `gpsSensor` | `imuSensor` | `imufilter`

Introduced in R2018b

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? \(MATLAB\)](#).

Creation

Syntax

```
altimeter = altimeterSensor
altimeter = altimeterSensor('ReferenceFrame',RF)
altimeter = altimeterSensor( ____,Name,Value)
```

Description

`altimeter = altimeterSensor` returns an `altimeterSensorSystem` 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 \(MATLAB\)](#).

UpdateRate — 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

<code>step</code>	Run <code>System</code> object algorithm
<code>release</code>	Release resources and allow changes to <code>System</code> object property values and input characteristics
<code>reset</code>	Reset internal states of <code>System</code> 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('UpdateRate',Fs, ...  
                             'ConstantBias',0.01, ...  
                             'NoiseDensity',0.05, ...  
                             'BiasInstability',0.05, ...  
                             'DecayFactor',0.5);
```

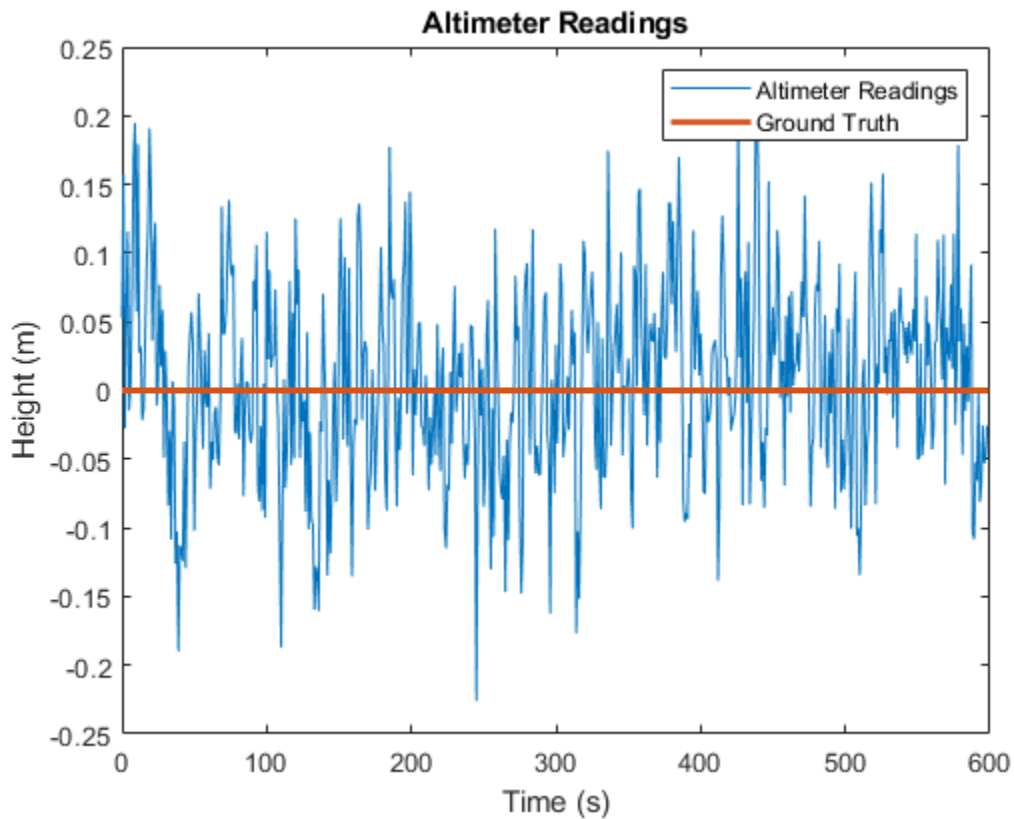
```
truePosition = zeros(numSamples,3);
```

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? \(MATLAB\)](#).

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 \(MATLAB\)](#).

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: NoData Types: `single` | `double`**AccelerometerGain — Accelerometer gain**

0.01 (default) | real scalar 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 scalar 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/s²)

N-by-3 matrix

Accelerometer readings in the sensor body coordinate system in m/s², 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/s².

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 `OrientationFormat` 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.

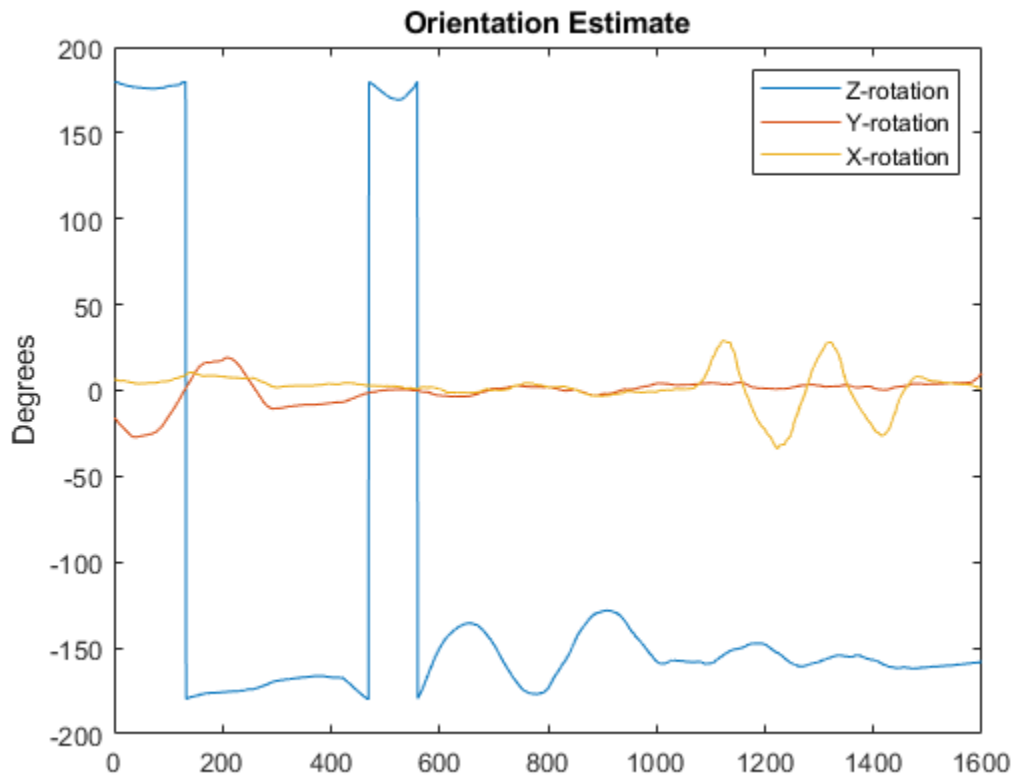
```
Fs = ld.Fs; % Hz  
fuse = complementaryFilter('SampleRate', Fs);
```

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 (m²)

[1e-6 1e-6 1e-6] (default) | scalar | three-element row vector

Additive position process noise in the local navigation coordinate system in m², 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/s²)²)

[50 50 50] (default) | scalar | three-element row vector

Additive acceleration process noise in (m/s²)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/s²)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 μT², 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 μT², 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

<code>predict</code>	Update states based on motion model
<code>fuseaccel</code>	Correct states using accelerometer data
<code>fusegyro</code>	Correct states using gyroscope data
<code>fusemag</code>	Correct states using magnetometer data
<code>fusegps</code>	Correct states using GPS data
<code>correct</code>	Correct states using direct state measurements
<code>residual</code>	Residuals and residual covariances from direct state measurements
<code>residualaccel</code>	Residuals and residual covariance from accelerometer measurements
<code>residualgps</code>	Residuals and residual covariance from GPS measurements
<code>residualmag</code>	Residuals and residual covariance from magnetometer measurements
<code>residualgyro</code>	Residuals and residual covariance from gyroscope measurements
<code>pose</code>	Current position, orientation, and velocity estimate
<code>reset</code>	Reset internal states
<code>stateinfo</code>	Display state vector information

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
```

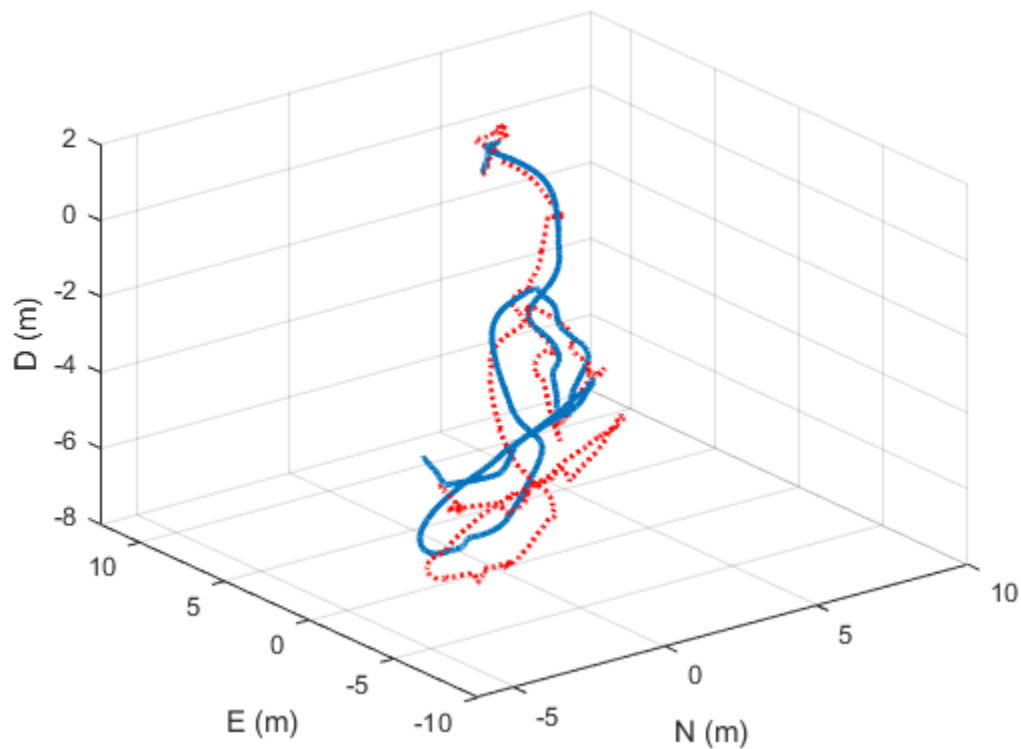
```
plot3(p(:,1), p(:,2), p(:,3), 'r:', 'LineWidth', 2)
```

```
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:

$$x = \begin{bmatrix} q_0 \\ q_1 \\ q_2 \\ q_3 \\ \text{angVel}_X \\ \text{angVel}_Y \\ \text{angVel}_Z \\ \text{position}_N \\ \text{position}_E \\ \text{position}_D \\ v_N \\ v_E \\ v_D \\ \text{accel}_N \\ \text{accel}_E \\ \text{accel}_D \\ \text{accelbias}_X \\ \text{accelbias}_Y \\ \text{accelbias}_Z \\ \text{gyrobias}_X \\ \text{gyrobias}_Y \\ \text{gyrobias}_Z \\ \text{geomagneticFieldVector}_N \\ \text{geomagneticFieldVector}_E \\ \text{geomagneticFieldVector}_D \\ \text{magbias}_X \\ \text{magbias}_Y \\ \text{magbias}_Z \end{bmatrix}$$

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.
- v_N, v_E, v_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:

correct

Correct states using direct state measurements

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

`insfilter` | `insfilterAsync`

Introduced in R2019a

fuseaccel

Correct states using accelerometer data

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/s²)

3-element row vector

Accelerometer readings in local sensor body coordinate system in m/s², specified as a 3-element row vector

Data Types: single | double

accelerationCovariance — Acceleration error covariance of accelerometer measurement ((m/s²)²)

scalar | 3-element row vector | 3-by-3 matrix

Acceleration error covariance of the accelerometer measurement in (m/s²)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/s².

resCov — Residual covariance

3-by-3 matrix of real values

Residual covariance, returned as a 3-by-3 matrix of real values in (m/s²)².

Extended Capabilities

C/C++ Code Generation

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

See Also

`insfilter` | `insfilterAsync`

Introduced in R2019a

fusegps

Correct states using GPS data

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 (m²)

3-by-3 matrix

Position measurement covariance of GPS receiver in m², 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/s²)

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` | `insfilterAsync` | `insfilterMARG`

Introduced in R2019a

fusegyro

Correct states using gyroscope data

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

`insfilter` | `insfilterAsync`

Introduced in R2019a

fusemag

Correct states using magnetometer data

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 $(\mu\text{T})^2$.

Extended Capabilities

C/C++ Code Generation

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

See Also

`insfilter` | `insfilterAsync`

Introduced in R2019a

pose

Current position, orientation, and velocity estimate

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 in the local NED coordinate system (m)**

3-element row vector

Position estimate in the local NED coordinate system in meters, returned as a 3-element row vector.

Data Types: single | double

orientation — **Orientation estimate in the local NED coordinate system**

quaternion (default) | 3-by-3 rotation matrix

Orientation estimate in the local NED coordinate system, returned as a scalar quaternion or 3-by-3 rotation matrix, depending on the specified orientation format. The quaternion or rotation matrix represents a frame rotation from the local NED reference frame to the body reference frame.

Data Types: single | double | quaternion

velocity — **Velocity estimate in the local NED coordinate system (m/s)**

3-element row vector

Velocity estimate in the local NED coordinate system in m/s, returned as a 3-element row vector.

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

Extended Capabilities

C/C++ Code Generation

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

See Also

`insfilter` | `insfilterAsync`

Introduced in R2019a

predict

Update states based on motion model

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

`insfilter` | `insfilterAsync`

Introduced in R2019a

reset

Reset internal states

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

insfilter | insfilterAsync

Introduced in R2019a

residual

Residuals and residual covariances from direct state measurements

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

Syntax

```
[res,resCov] = fuseaccel(FUSE,acceleration,accelerationCovariance)
```

Description

`[res,resCov] = fuseaccel(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/s²)

3-element row vector

Accelerometer readings in local sensor body coordinate system in m/s², specified as a 3-element row vector

Data Types: `single` | `double`

accelerationCovariance — Acceleration error covariance of accelerometer measurement ((m/s²)²)

scalar | 3-element row vector | 3-by-3 matrix

Acceleration error covariance of the accelerometer measurement in (m/s²)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/s².

resCov — Residual covariance

3-by-3 matrix of real values

Residual covariance, returned a 3-by-3 matrix of real values in (m/s²)².

Extended Capabilities

C/C++ Code Generation

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

See Also

`insfilter` | `insfilterAsync`

Introduced in R2020a

residualgps

Residuals and residual covariance from GPS measurements

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 (m²)

3-by-3 matrix

Position measurement covariance of GPS receiver in m², 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/s²)

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

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

`insfilter` | `insfilterAsync`

Introduced in R2020a

residualmag

Residuals and residual covariance from magnetometer measurements

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 $(\mu\text{T})^2$.

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

Syntax

```
stateinfo(FUSE)
```

Description

`stateinfo(FUSE)` displays the meaning of each index of the `State` property of the `insfilterAsync` object and the associated units.

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

`insfilter` | `insfilterAsync`

Introduced in R2019a

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**

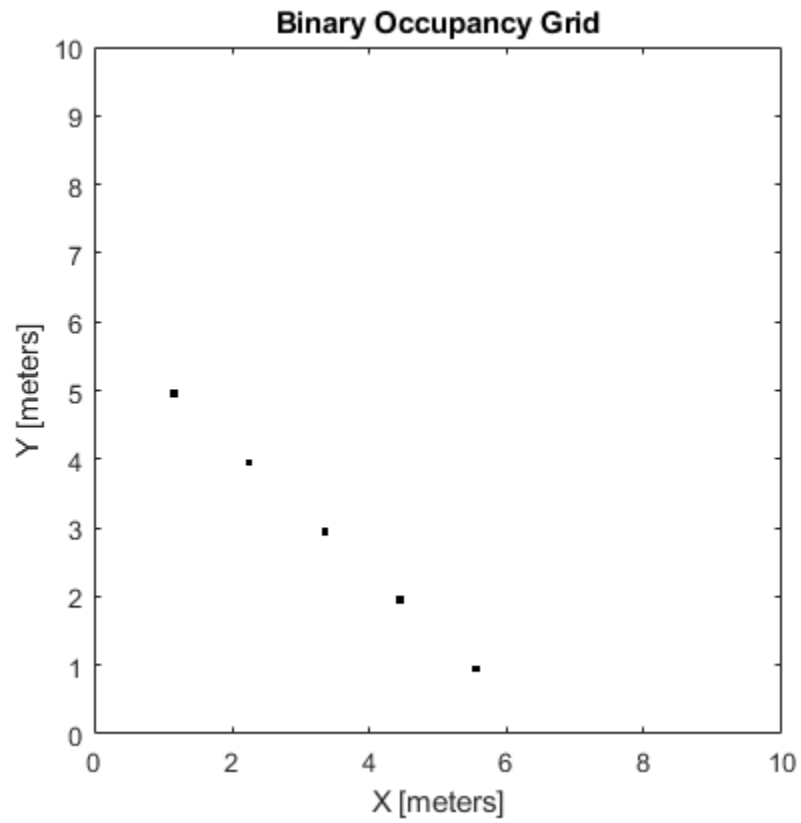
Create a 10m x 10m empty map.

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

Set occupancy of world locations and show map.

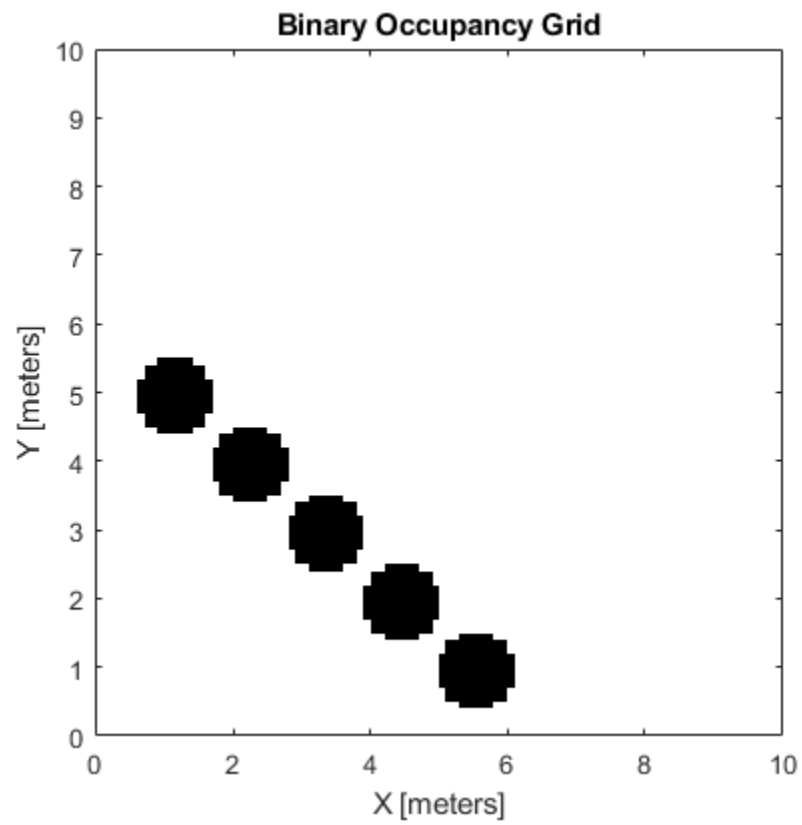
```
x = [1.2; 2.3; 3.4; 4.5; 5.6];
y = [5.0; 4.0; 3.0; 2.0; 1.0];
```

```
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)
```

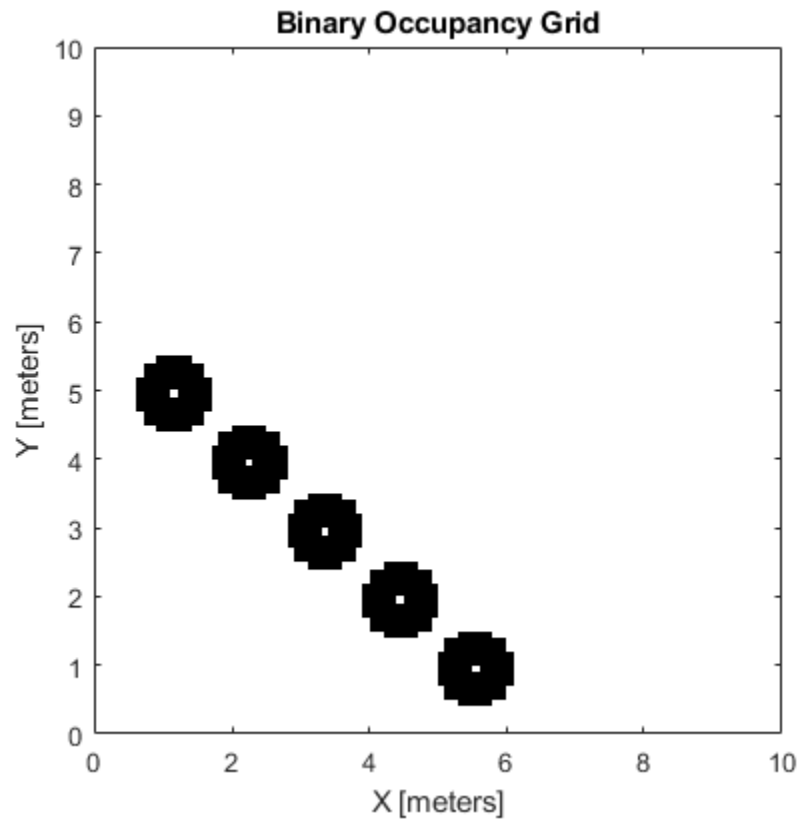


Image to Binary Occupancy Grid Example

This example shows how to convert an image to a binary occupancy grid for using with mapping and path planning.

Import image.

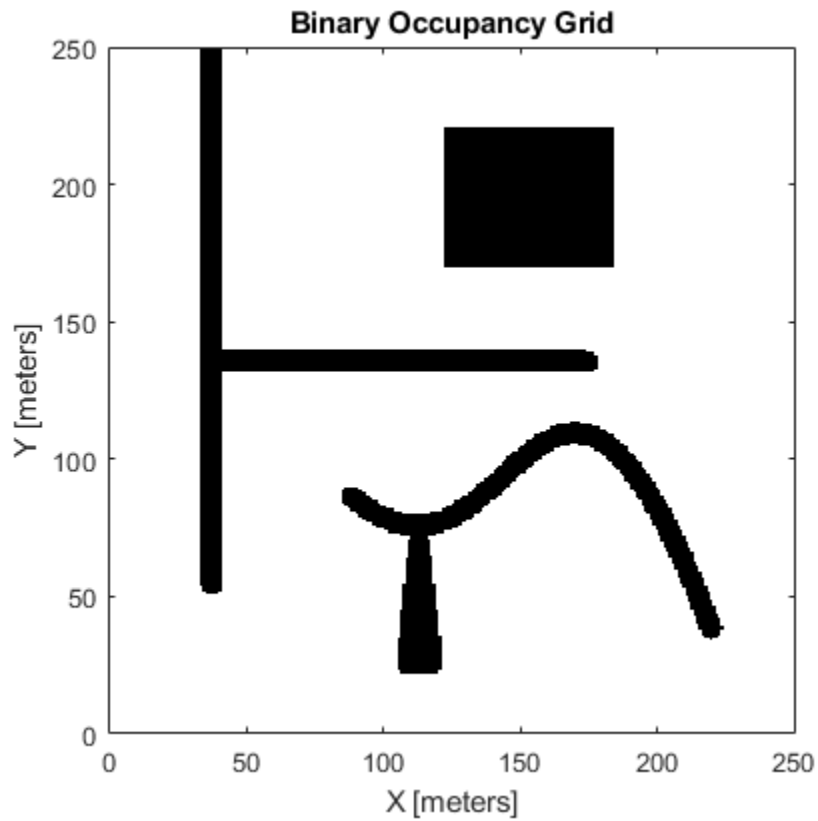
```
image = imread('imageMap.png');
```

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)
```

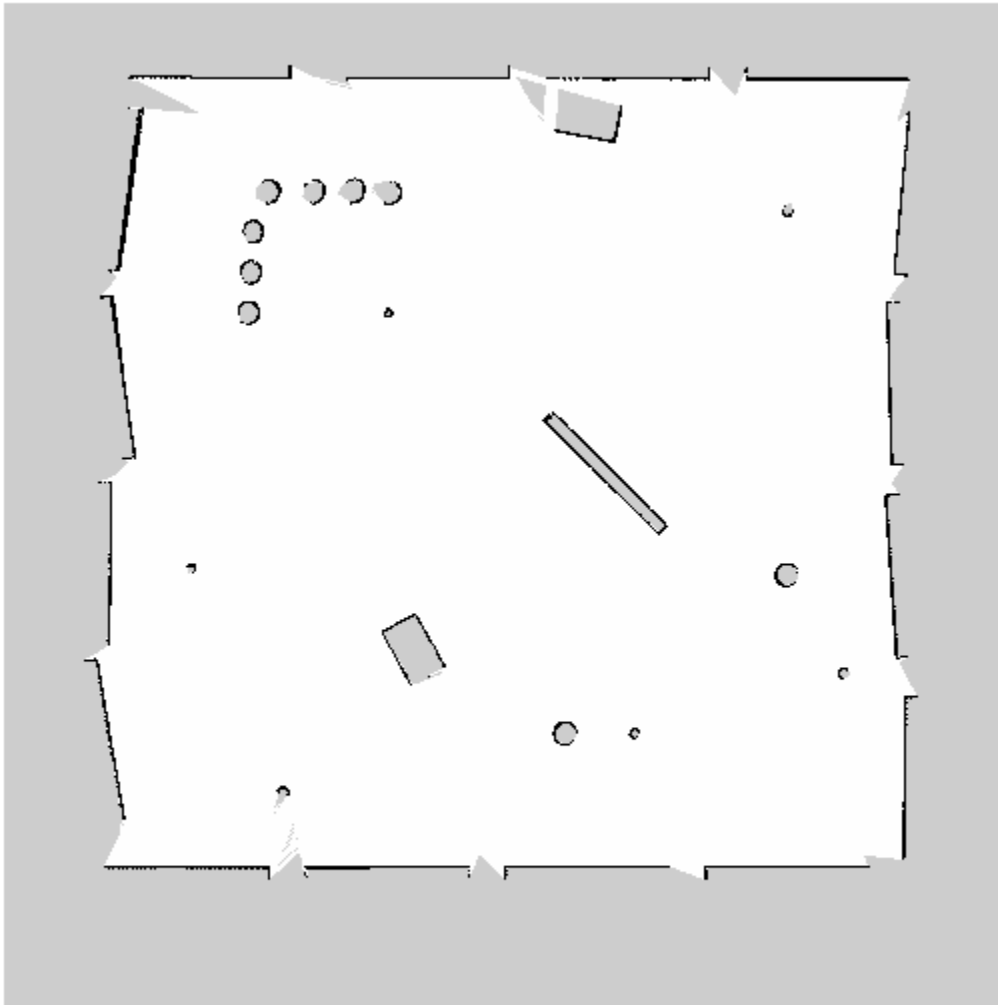


Convert PGM Image to Map

This example shows how to convert a .pgm file into a `binaryOccupancyMap` object for use in MATLAB.

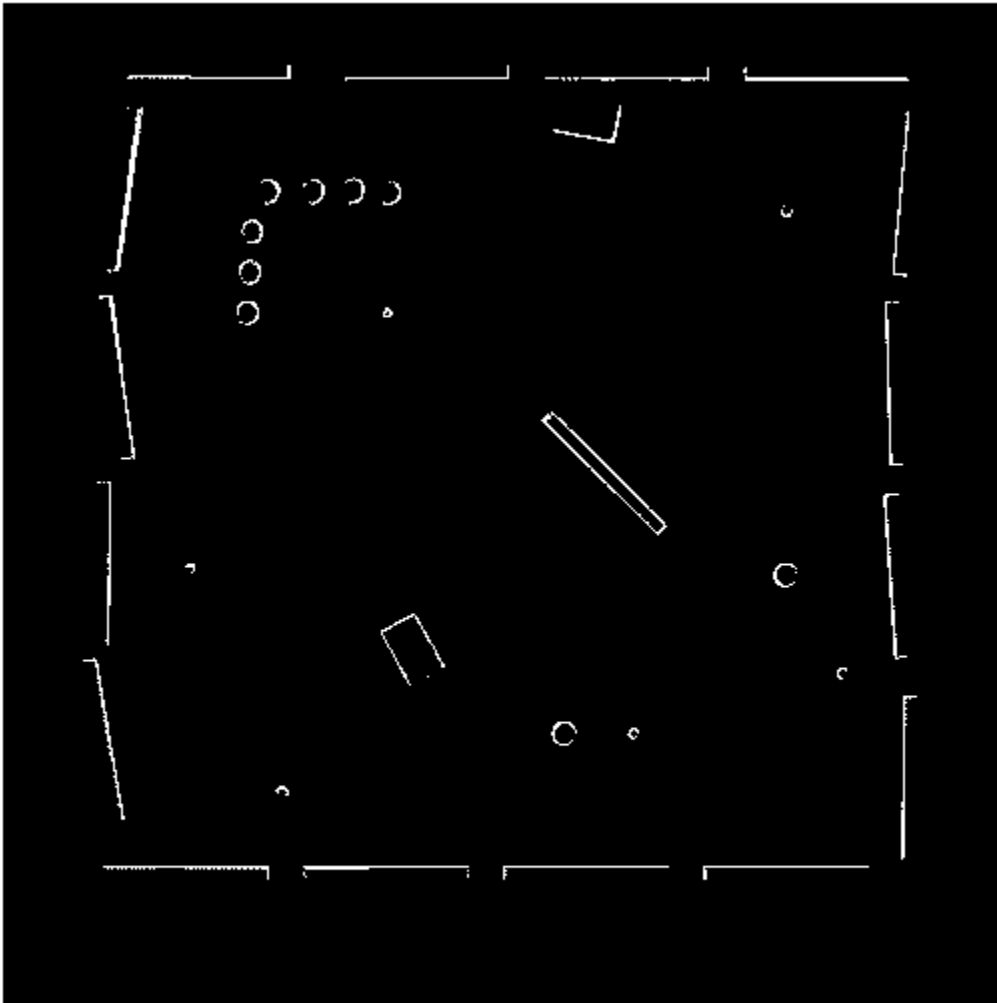
Import image using `imread`. The image is quite large and should be cropped to the relevant area.

```
image = imread('playpen_map.pgm');  
imageCropped = image(750:1250,750:1250);  
imshow(imageCropped)
```



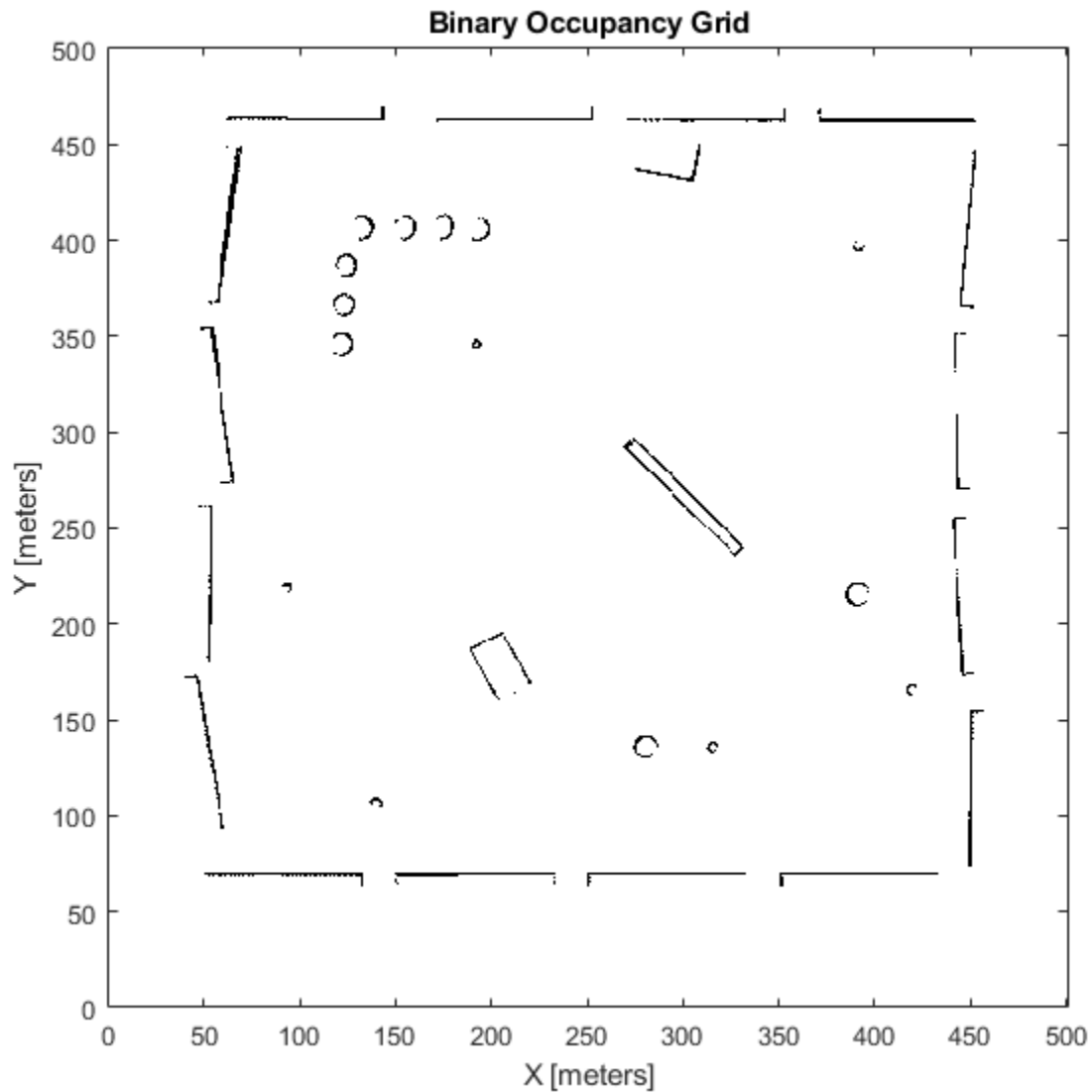
Unknown areas (gray) should be removed and treated as free space. Create a logical matrix based on a threshold. Depending on your image, this value could be different. Occupied space should be set as 1 (white in image).

```
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.

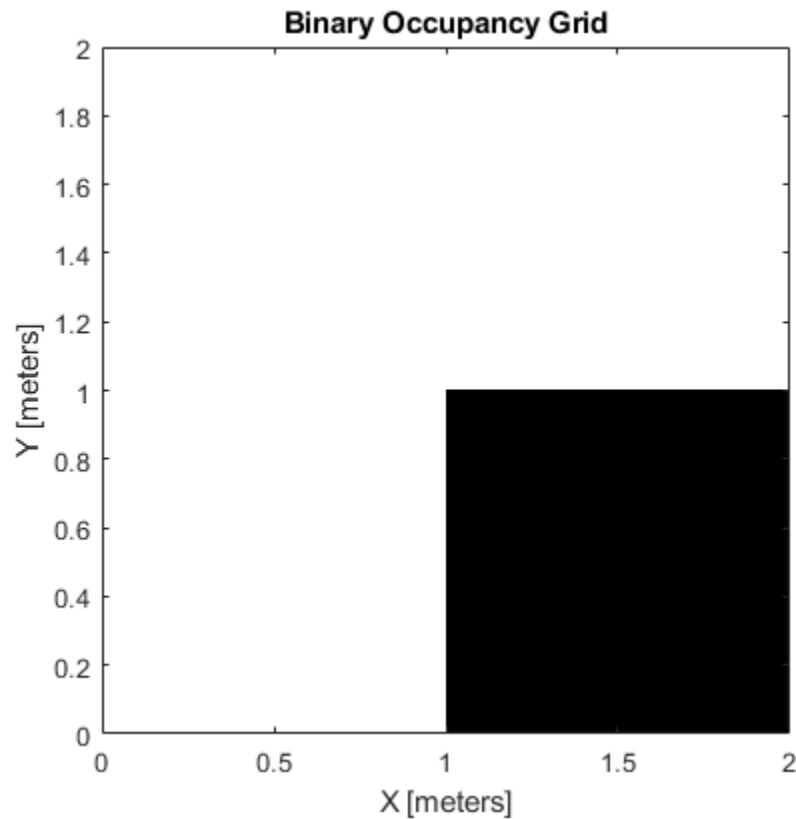
Examples

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 localtwo-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 sizetwo-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 gridtwo-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).

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.

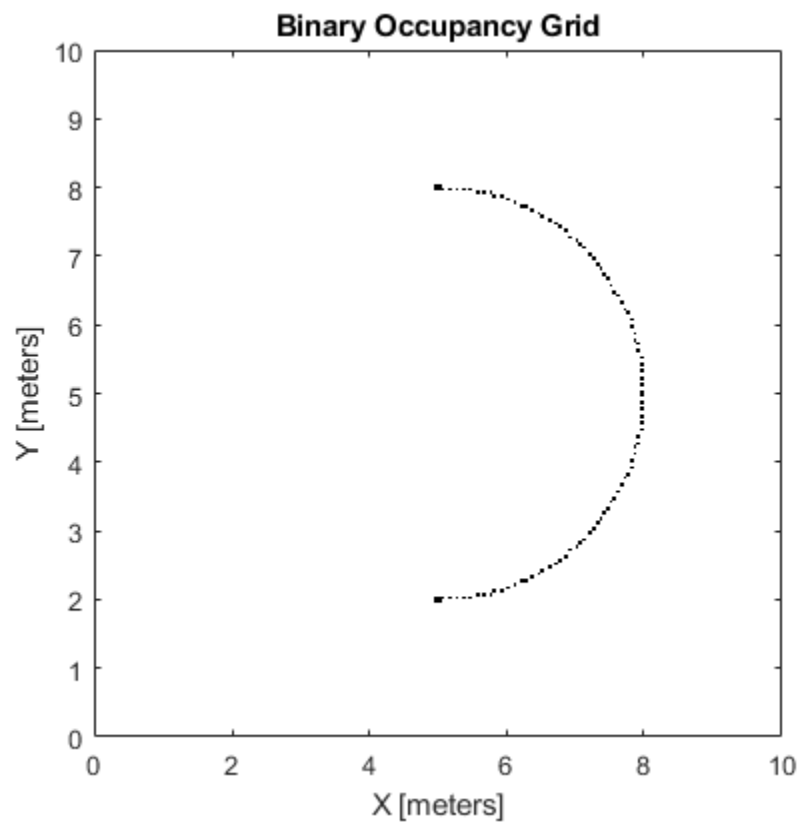
```
scan = lidarScan(ranges,angles);
```

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])
```

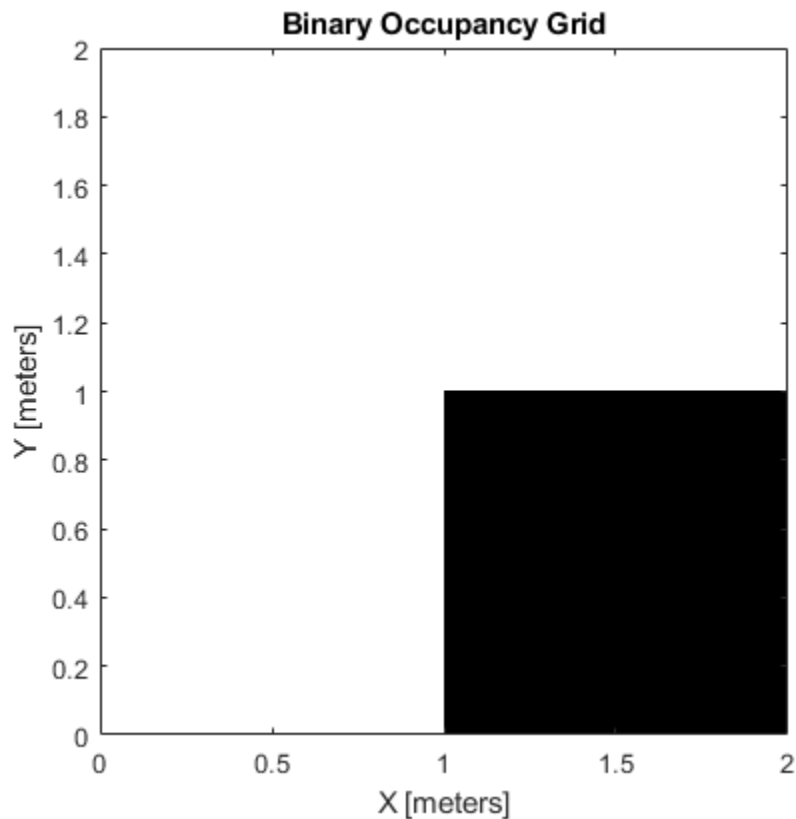
```
ans = logical
     1
```

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 localtwo-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 sizetwo-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 gridtwo-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.

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.

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.

See Also

`binaryOccupancyMap` | `grid2local` | `world2grid`

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

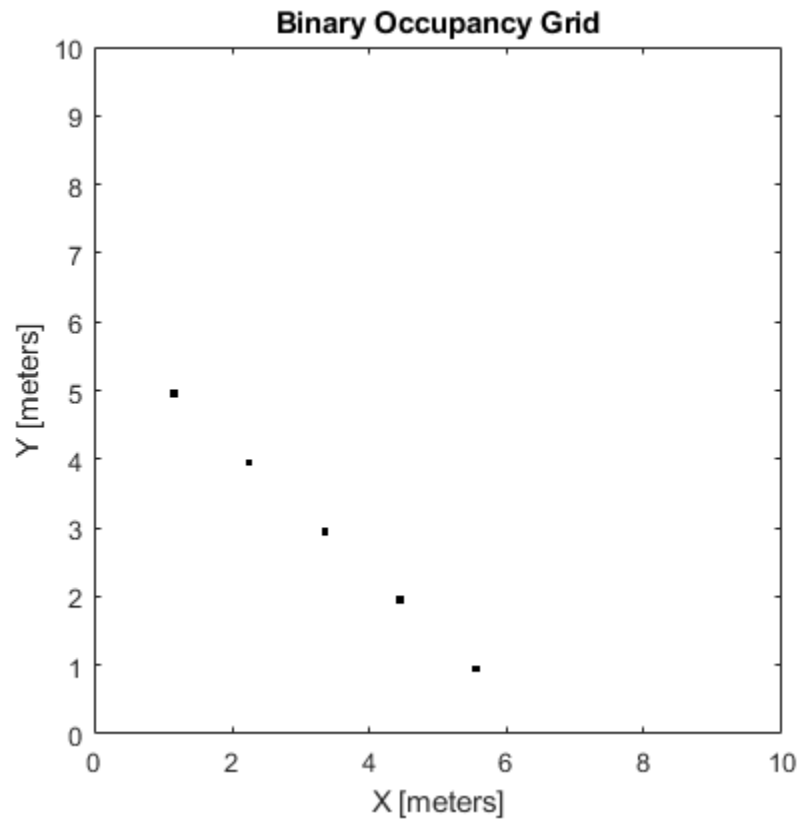
Create a 10m x 10m empty map.

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

Set occupancy of world locations and show map.

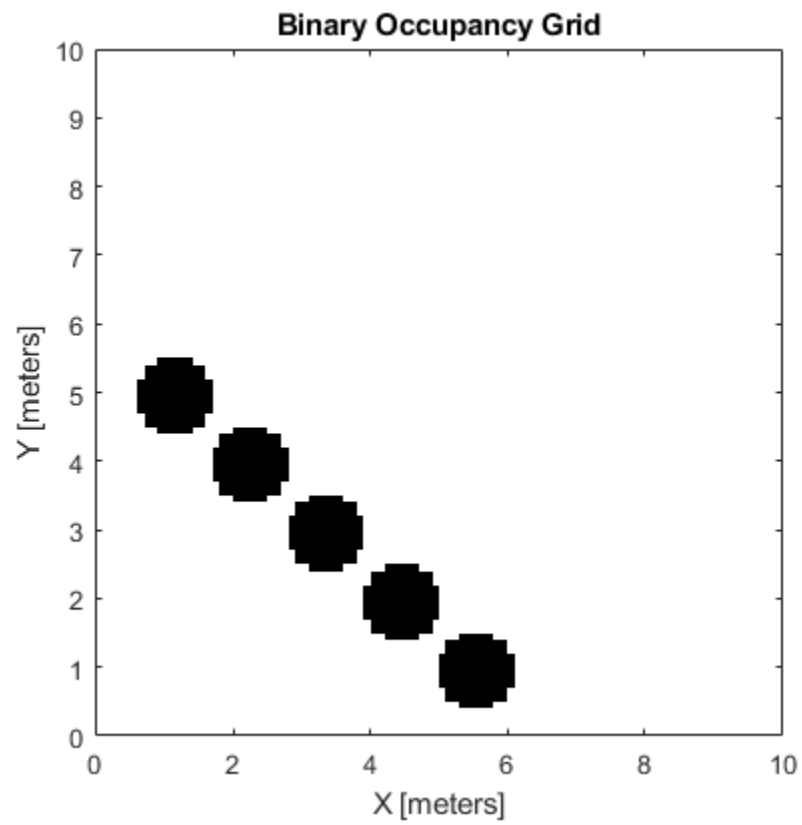
```
x = [1.2; 2.3; 3.4; 4.5; 5.6];
y = [5.0; 4.0; 3.0; 2.0; 1.0];
```

```
setOccupancy(map, [x y], ones(5,1))
figure
show(map)
```



Inflate occupied locations by a given radius.

```
inflate(map, 0.5)  
figure  
show(map)
```

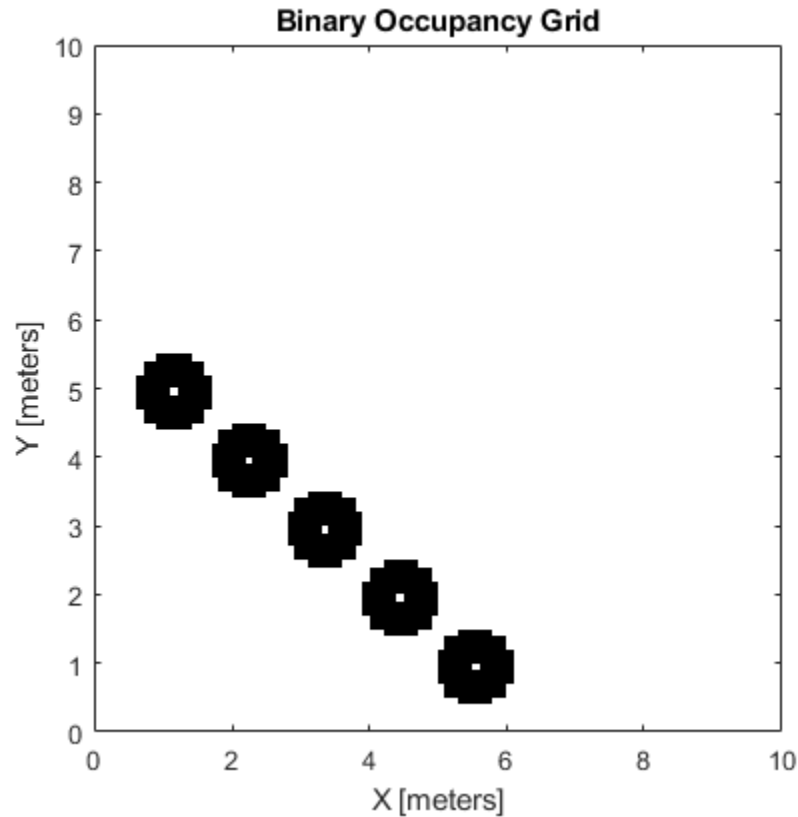


Get grid locations from world locations.

```
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`

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.

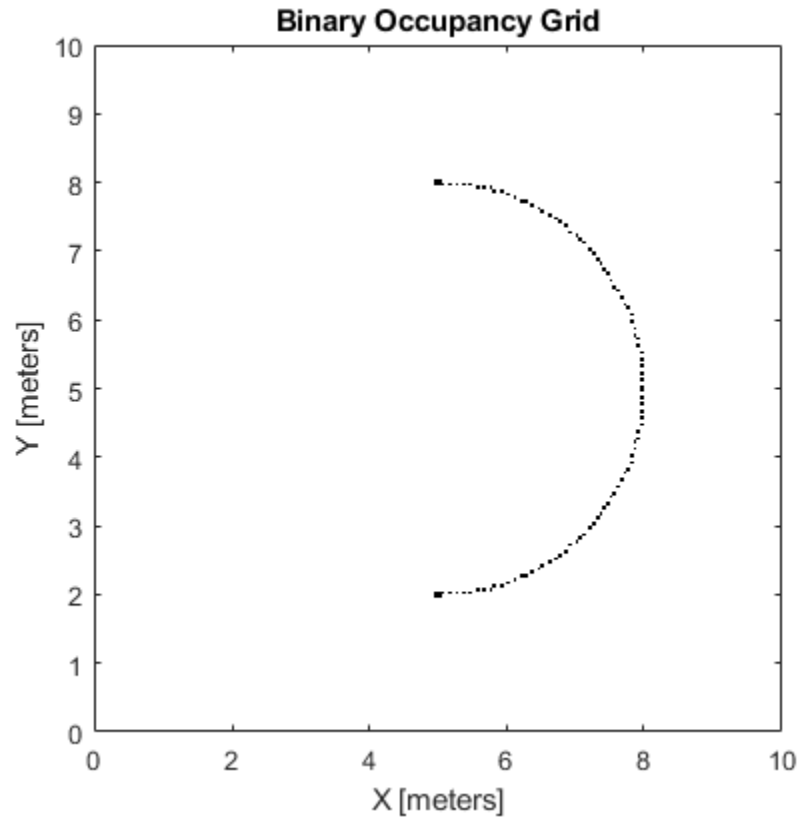
```
scan = lidarScan(ranges,angles);
```

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 θ (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`.

See Also

`binaryOccupancyMap` | `lidarScan` | `lidarScan` | `occupancyMap`

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`

See Also

`binaryOccupancyMap` | `grid2world` | `grid2world` | `occupancyMap`

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`

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.

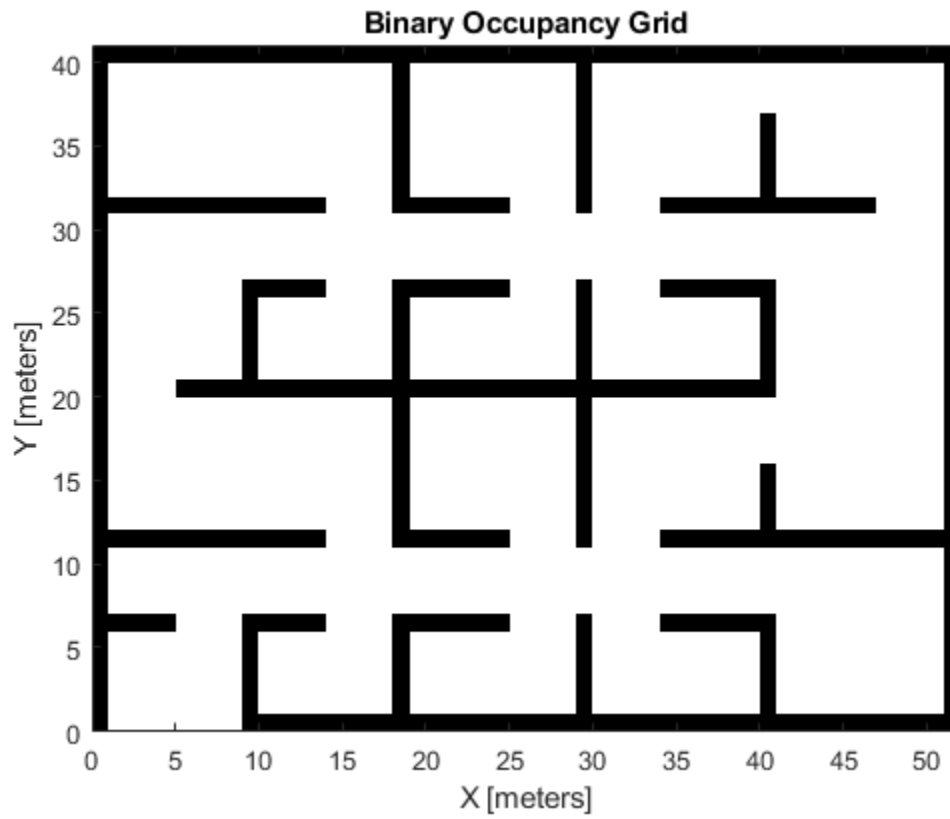
Examples

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.

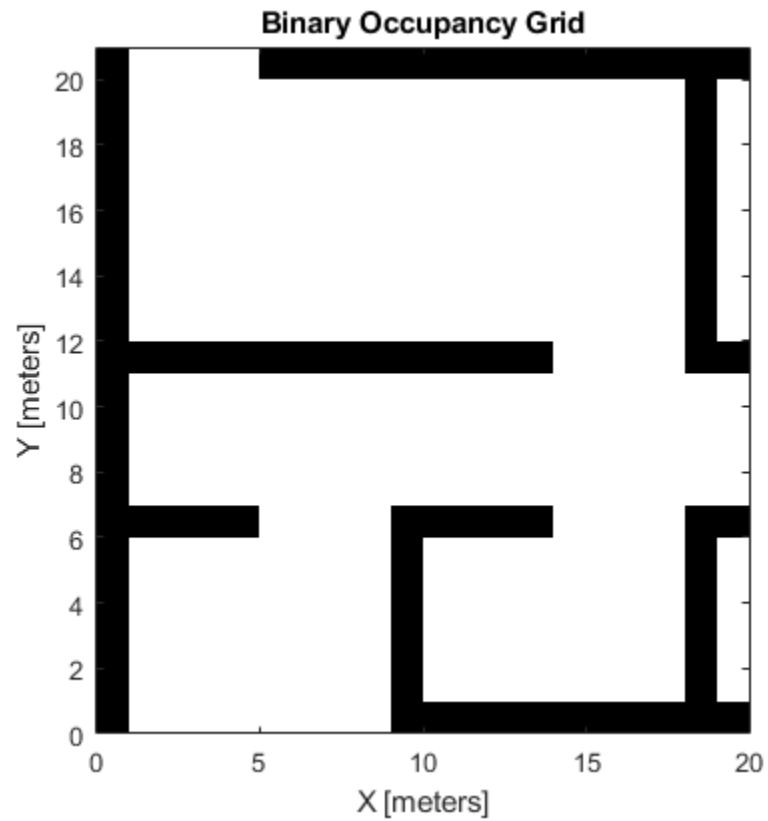
Load example maps. Create a binary occupancy map from the `complexMap`.

```
load exampleMaps.mat  
map = binaryOccupancyMap(complexMap);  
show(map)
```



Create a smaller local map.

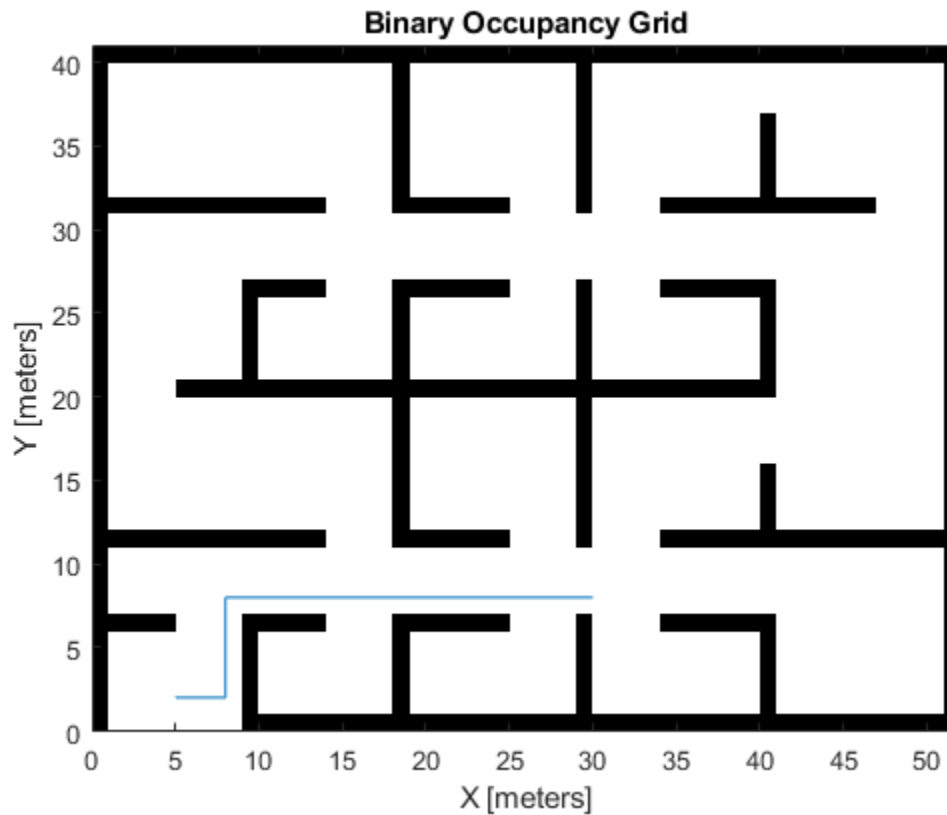
```
mapLocal = binaryOccupancyMap(complexMap(end-20:end,1:20));  
show(mapLocal)
```



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
```



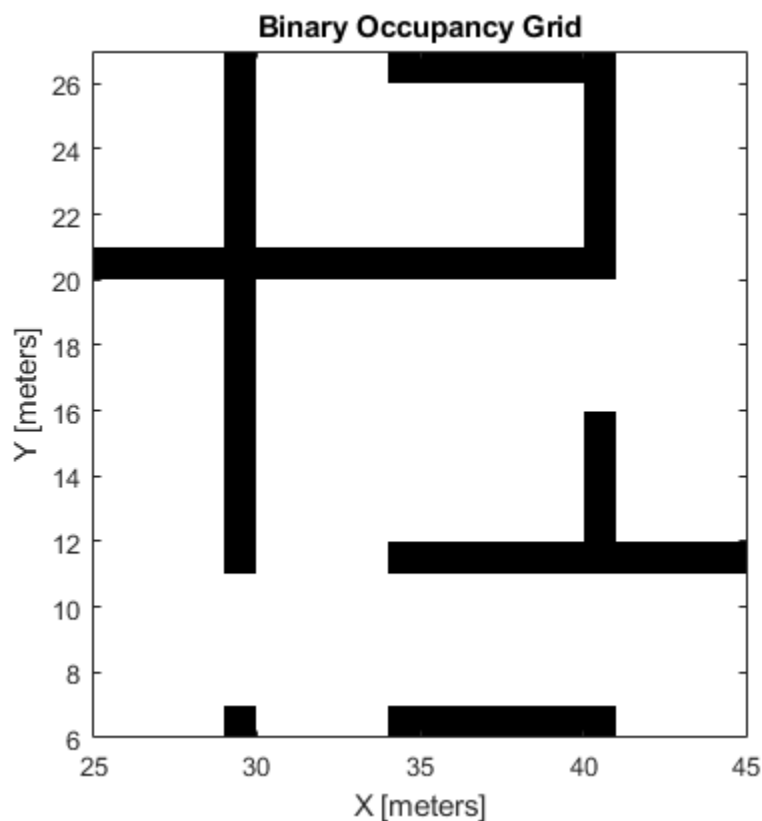
Create a loop for moving between points by the map resolution. Divide the difference between points by the map resolution to see how many incremental moves you can make.

```

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
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.

See Also

`binaryOccupancyMap` | `occupancyMap` | `occupancyMatrix`

Introduced in R2019b

occupancyMatrix

Convert occupancy grid to matrix

Syntax

```
mat = binaryOccupancyMatrix(map)
```

Description

`mat = binaryOccupancyMatrix(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`

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.

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

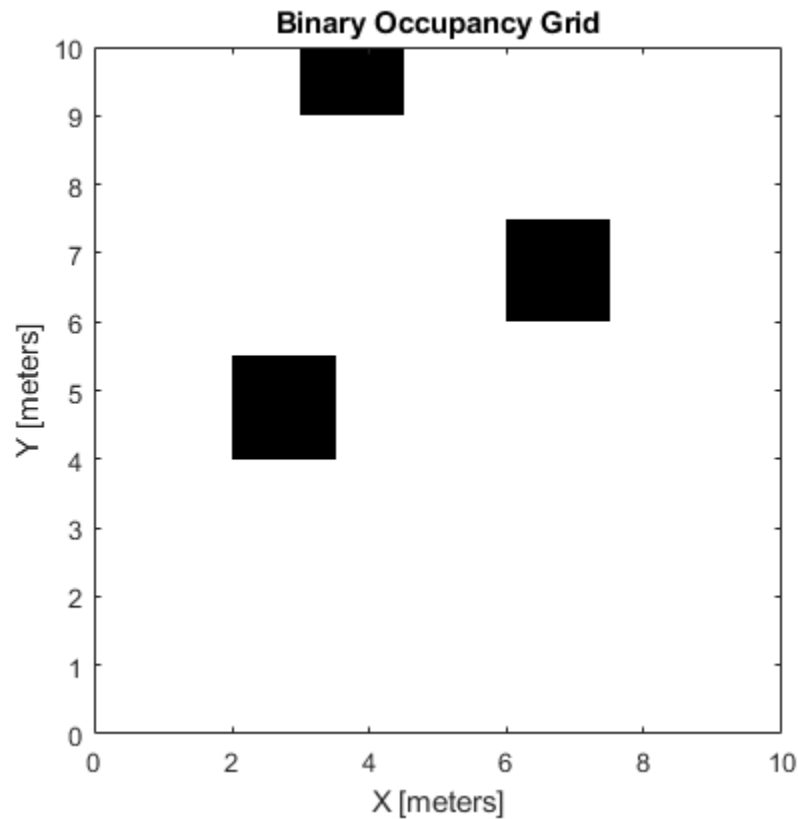
`intersectionPts = rayIntersection(map,pose,angles,maxrange)` returns intersection points of rays and occupied cells in the specified map. Rays emanate from the specified pose and angles. Intersection points are returned in the world coordinate frame. If there is no intersection up to the specified maxrange, [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.

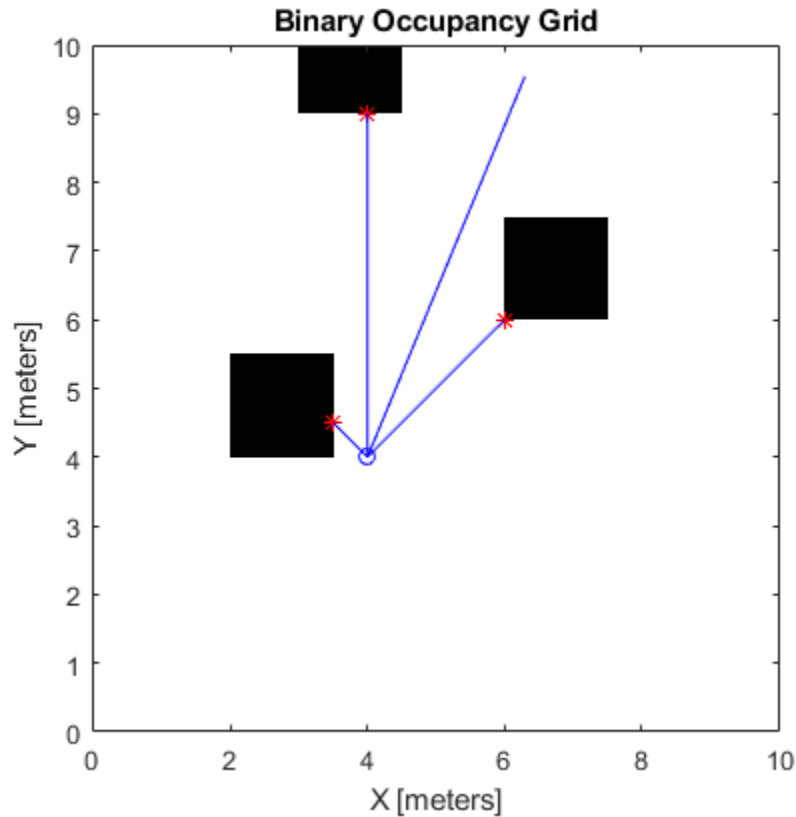
```
maxrange = 6;
angles = [pi/4, -pi/4, 0, -pi/8];
vehiclePose = [4, 4, pi/2];
intseccionPts = rayIntersection(map, vehiclePose, angles, maxrange)
```

```
intseccionPts = 4x2

    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(intseccionPts(:,1),intseccionPts(:,2),'*r') % Intersection points
plot(vehiclePose(1),vehiclePose(2),'ob') % Vehicle pose
for i = 1:3
    plot([vehiclePose(1),intseccionPts(i,1)],...
         [vehiclePose(2),intseccionPts(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 θ (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`.

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

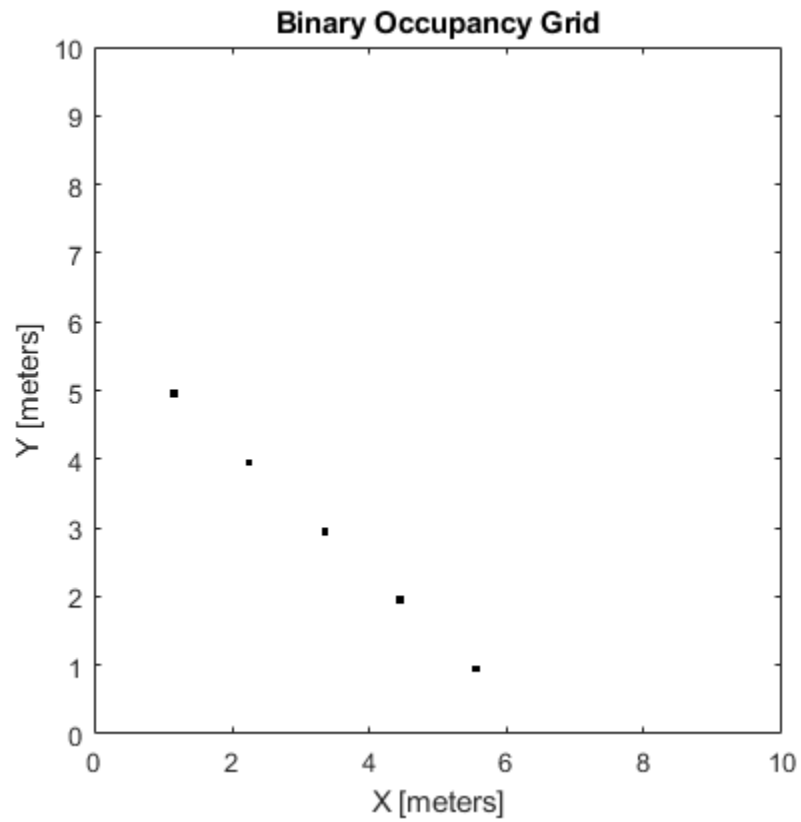
Create and Modify Binary Occupancy Grid

Create a 10m x 10m empty map.

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

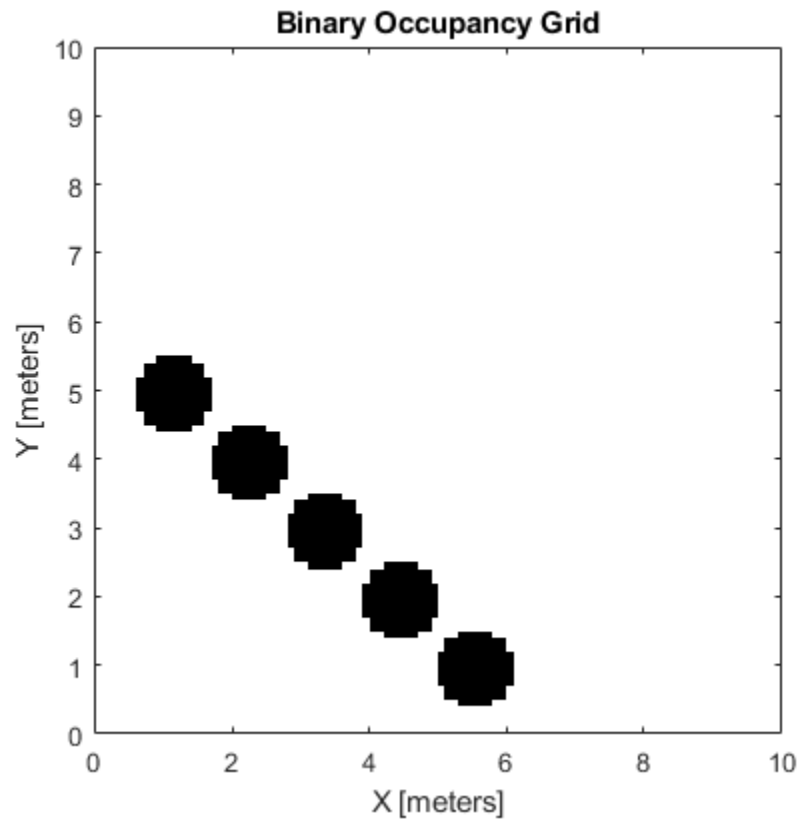
Set occupancy of world locations and show map.

```
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)
```

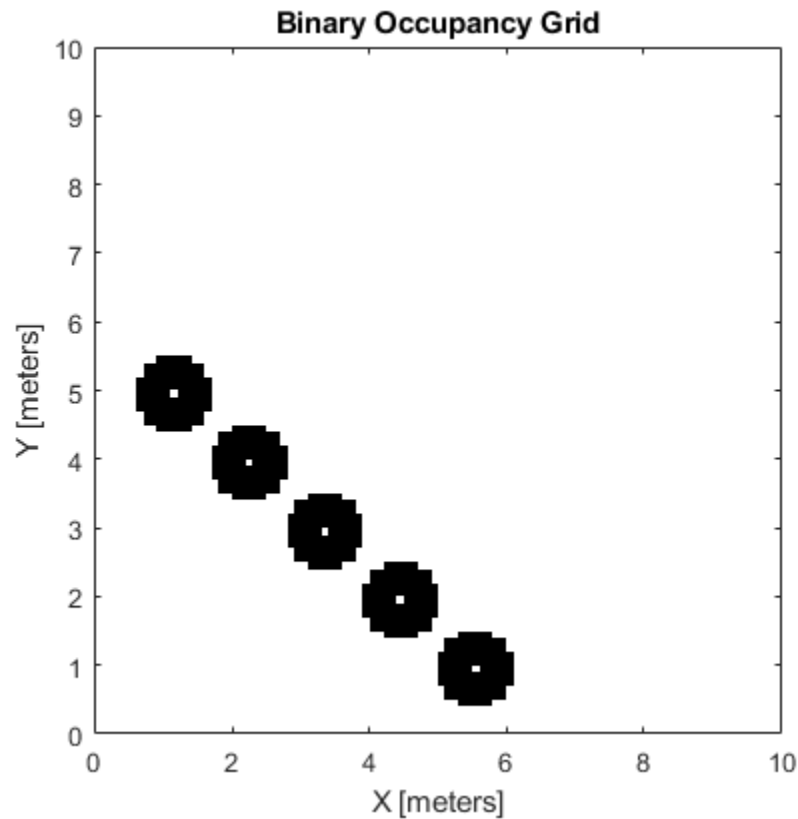


Get grid locations from world locations.

```
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 localtwo-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 gridtwo-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.

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`.

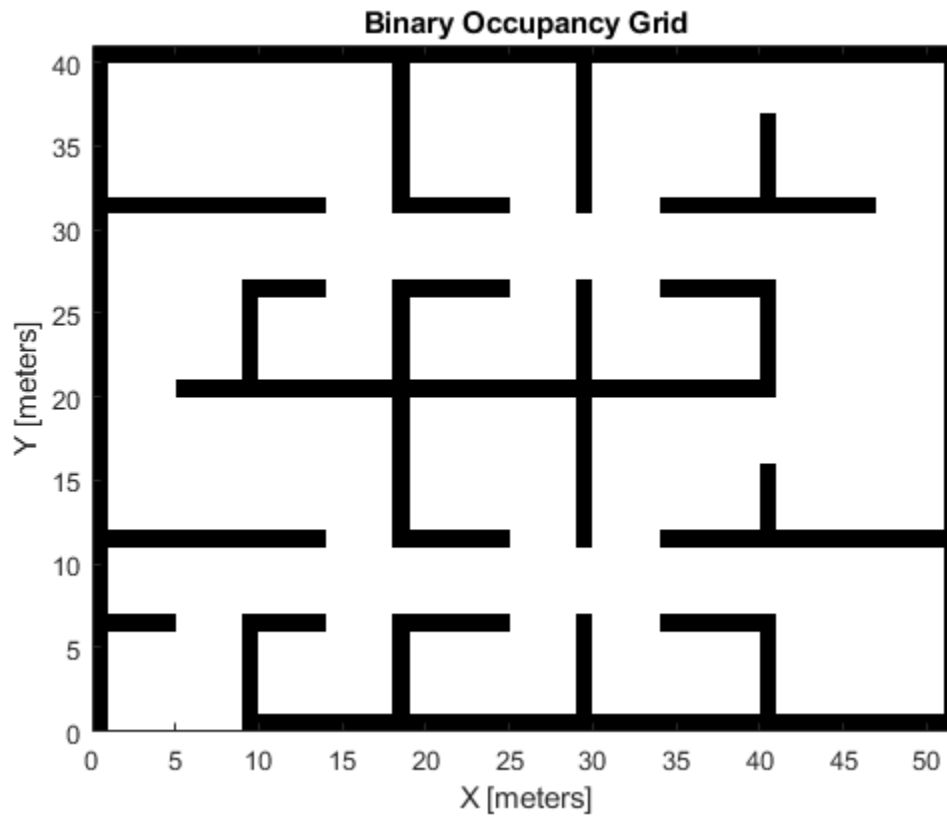
Examples

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.

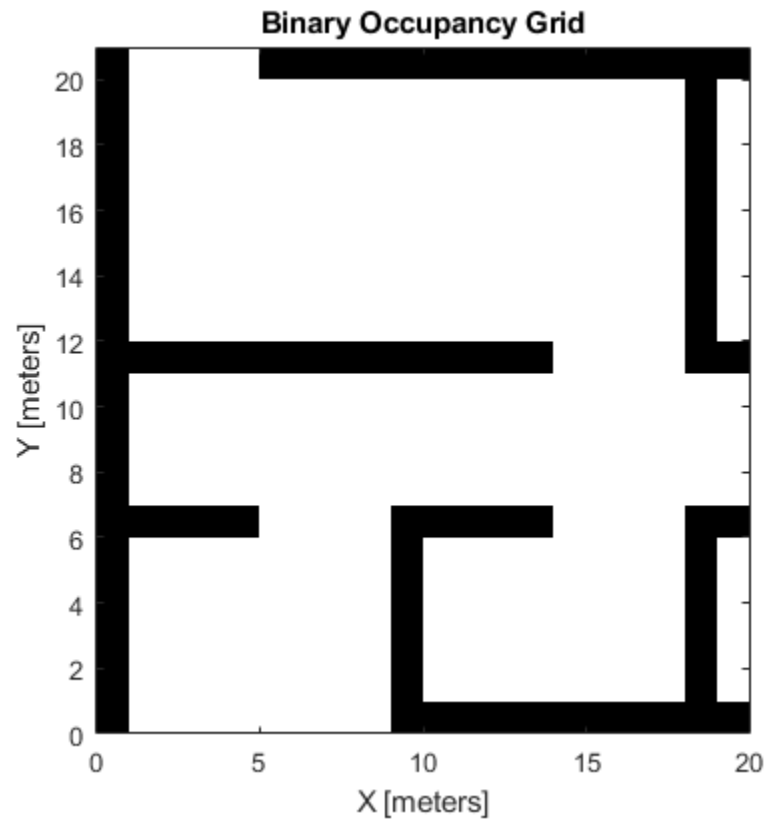
Load example maps. Create a binary occupancy map from the `complexMap`.

```
load exampleMaps.mat
map = binaryOccupancyMap(complexMap);
show(map)
```



Create a smaller local map.

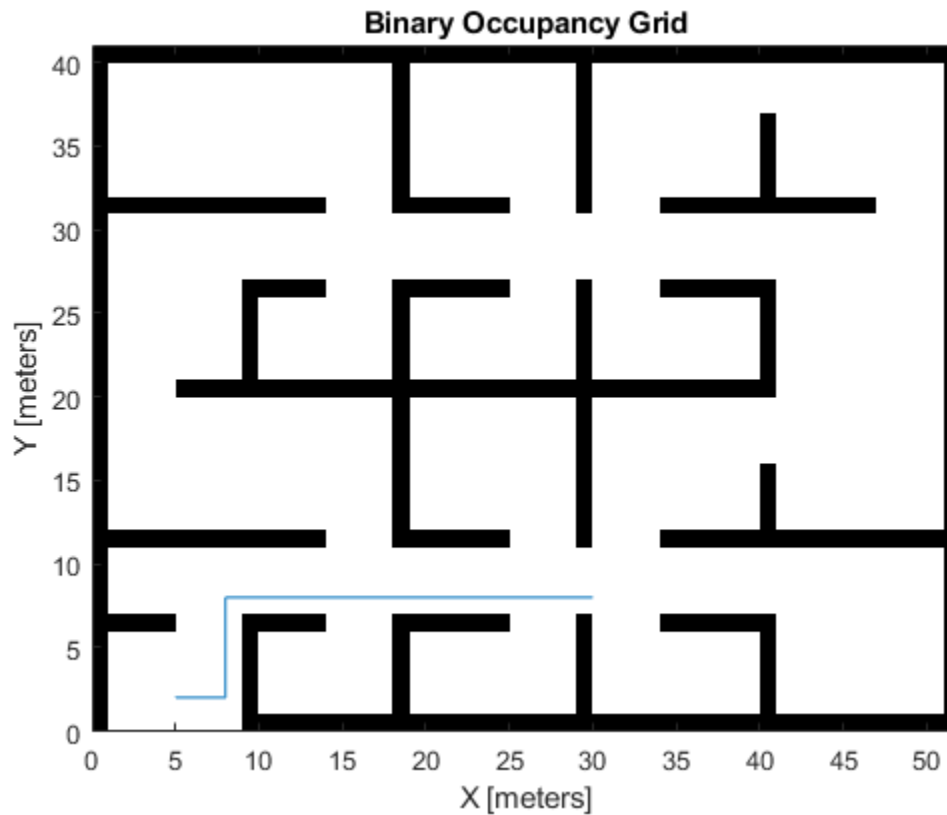
```
mapLocal = binaryOccupancyMap(complexMap(end-20:end,1:20));  
show(mapLocal)
```

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
```



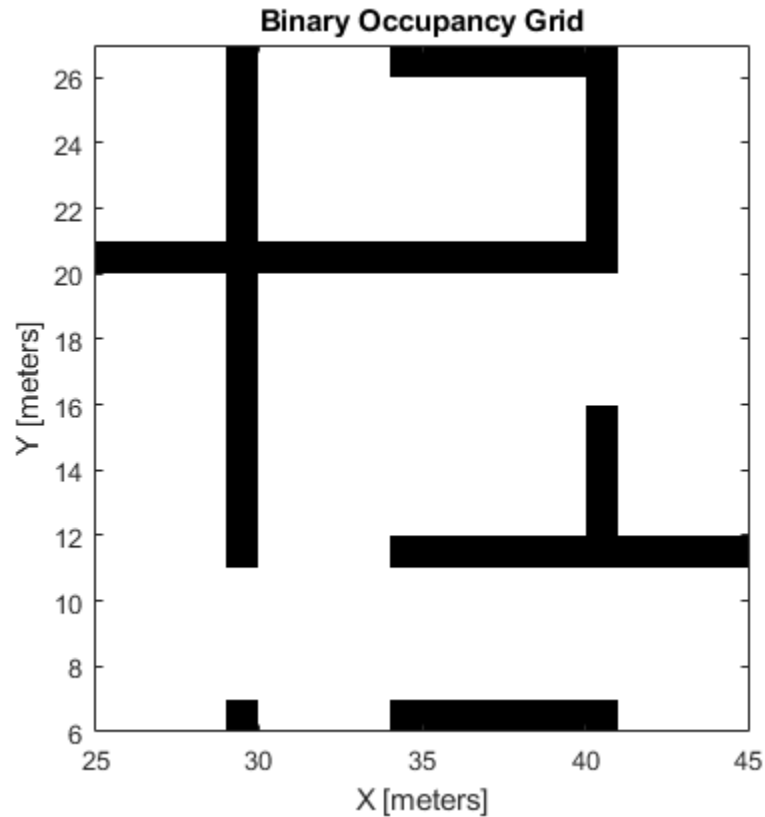
Create a loop for moving between points by the map resolution. Divide the difference between points by the map resolution to see how many incremental moves you can make.

```

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
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

occupancyMap | binaryOccupancyMap

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`.

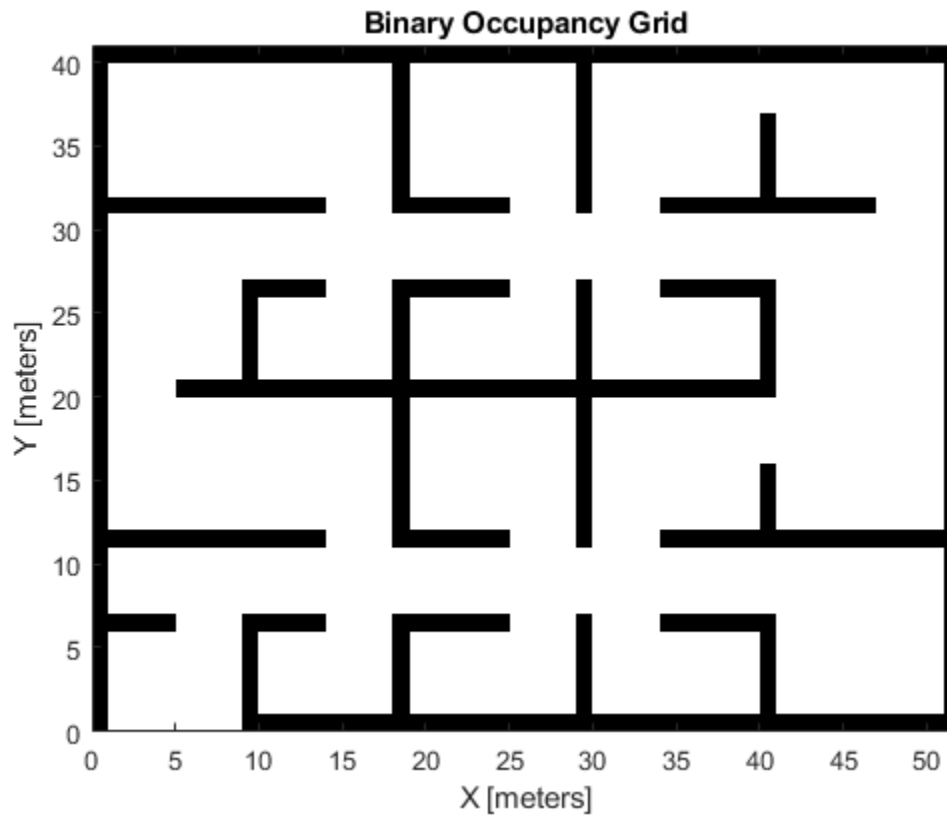
Examples

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.

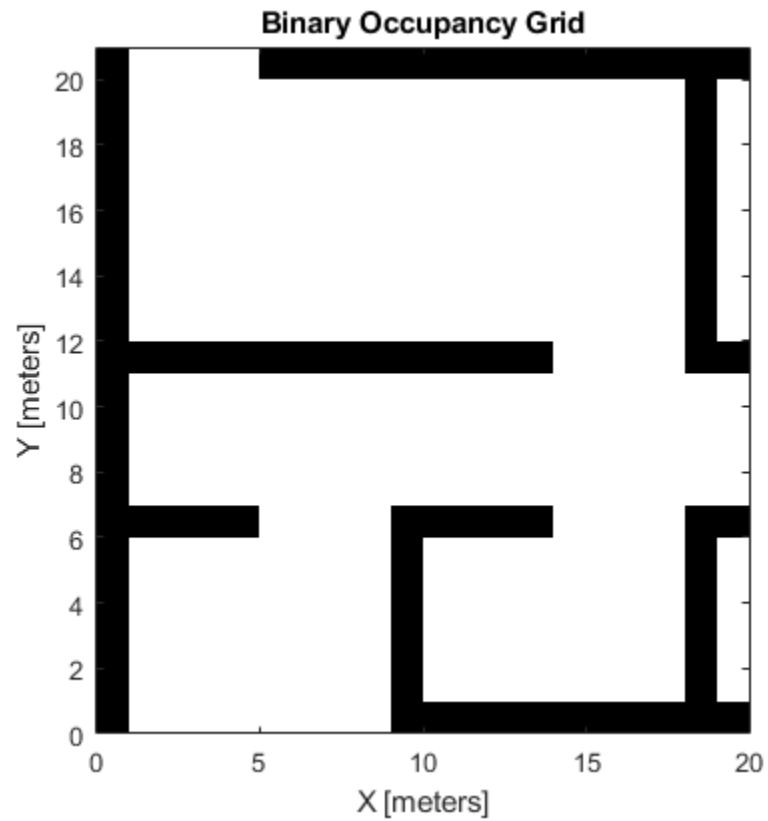
Load example maps. Create a binary occupancy map from the `complexMap`.

```
load exampleMaps.mat
map = binaryOccupancyMap(complexMap);
show(map)
```



Create a smaller local map.

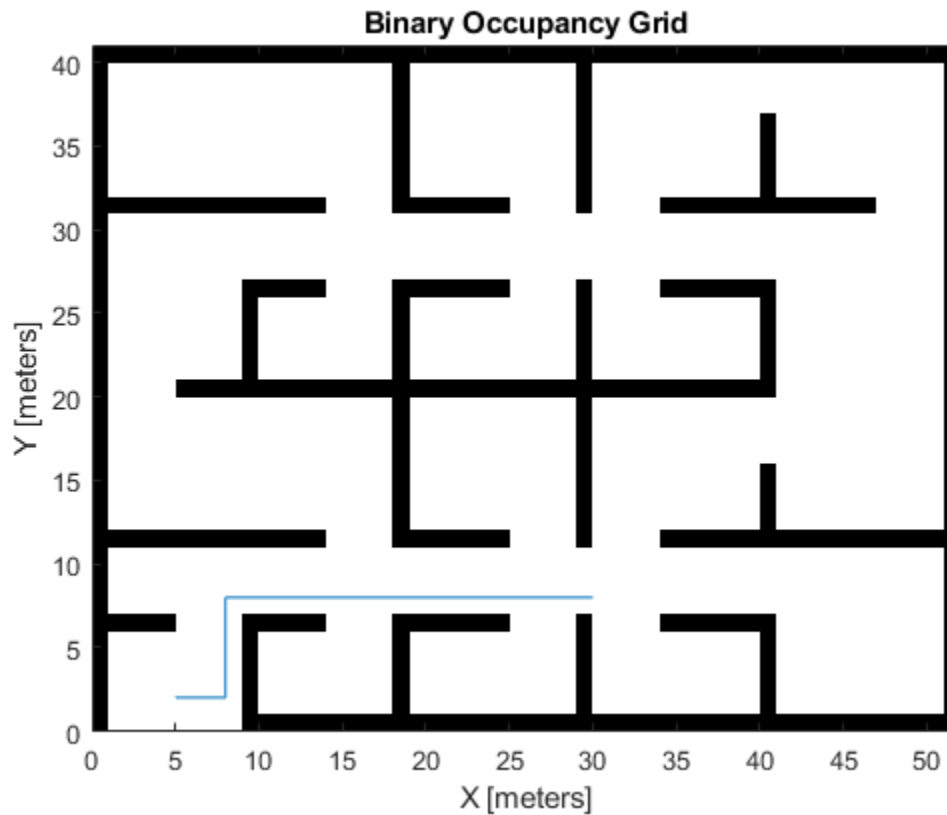
```
mapLocal = binaryOccupancyMap(complexMap(end-20:end,1:20));  
show(mapLocal)
```



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
```



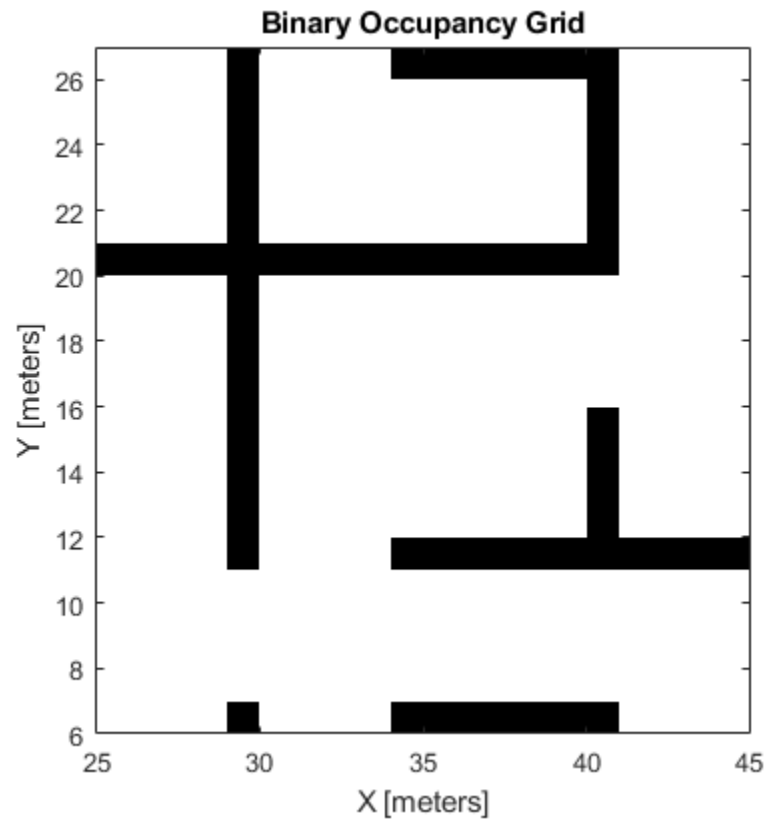
Create a loop for moving between points by the map resolution. Divide the difference between points by the map resolution to see how many incremental moves you can make.

```

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
end

```

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.

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.

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.

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? \(MATLAB\)](#).

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

<code>step</code>	Run System object algorithm
<code>release</code>	Release resources and allow changes to System object property values and input characteristics
<code>reset</code>	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.

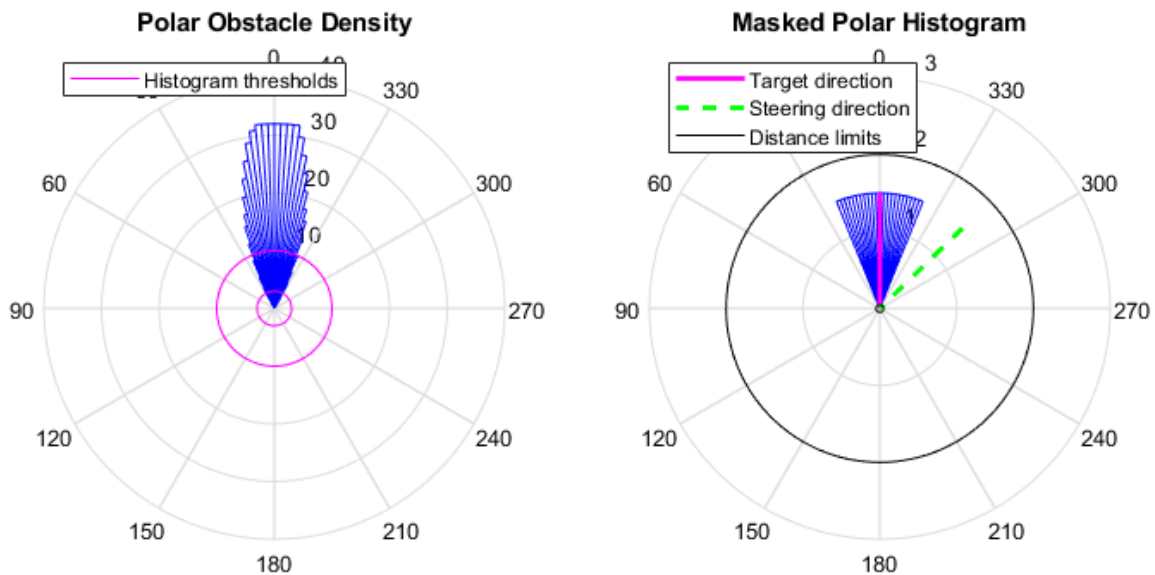
```
scan = lidarScan(ranges,angles);
```

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.

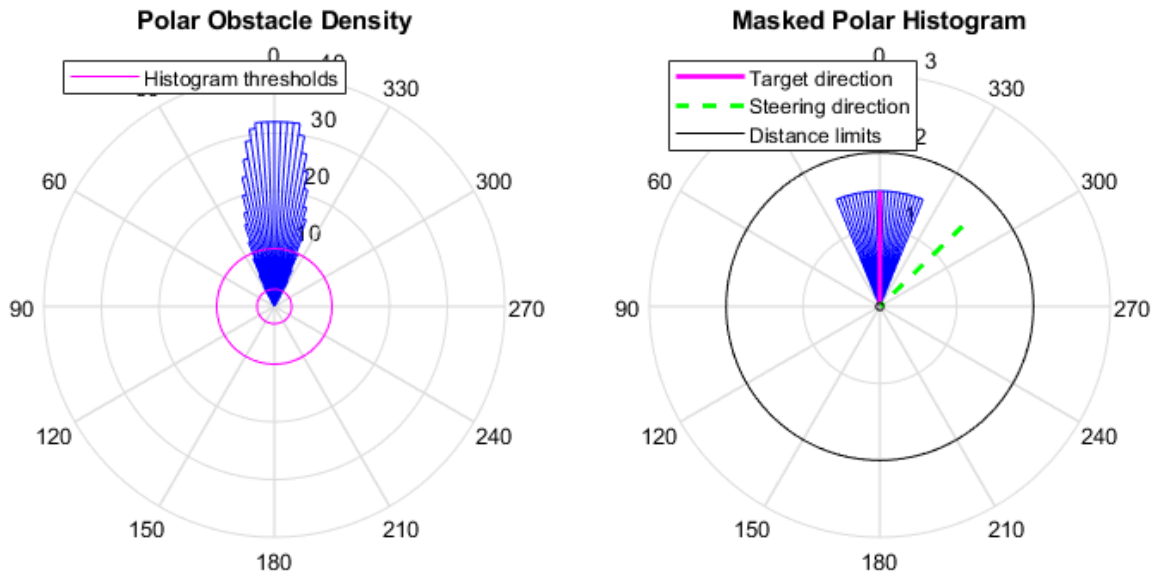
```
scan = lidarScan(ranges,angles);
```

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? \(MATLAB\)](#).

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

<code>step</code>	Run System object algorithm
<code>release</code>	Release resources and allow changes to System object property values and input characteristics
<code>reset</code>	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` | `binaryOccupancyMap` | `controllerVFH` | `occupancyMap` | `occupancyMap`

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.

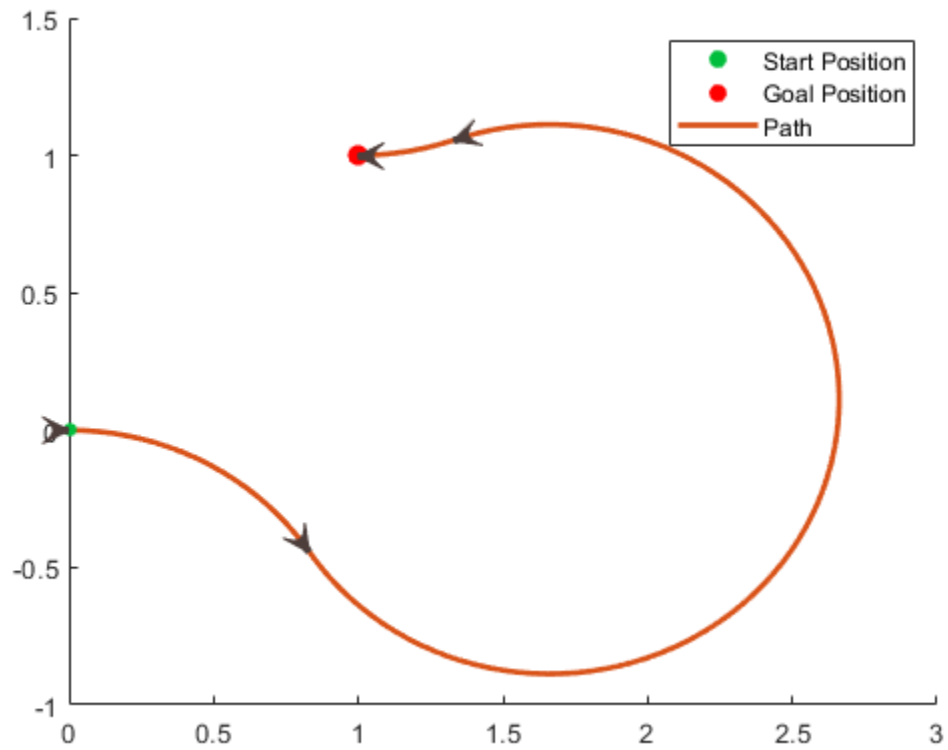
```
startPose = [0 0 0];
goalPose = [1 1 pi];
```

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.

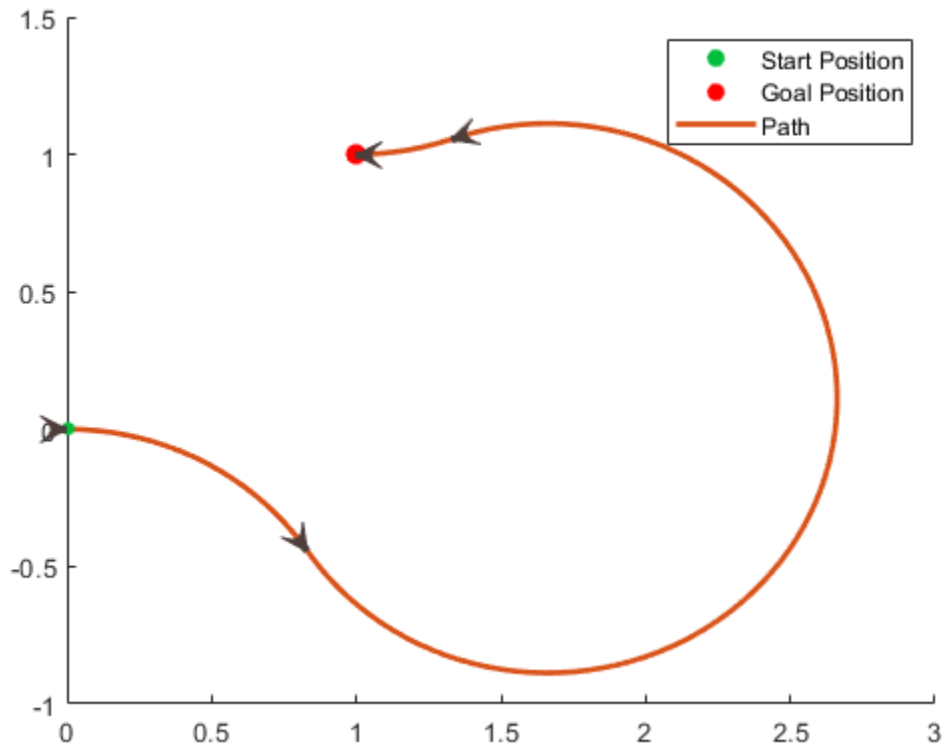
```
startPose = [0 0 0];  
goalPose = [1 1 pi];
```

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'}
```

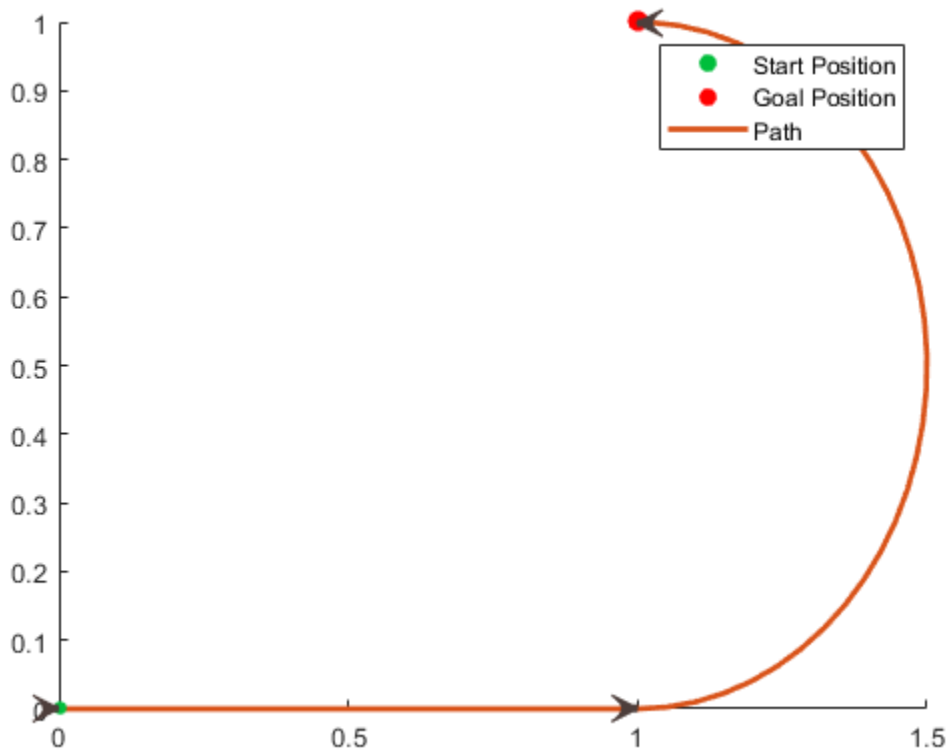
Disable this specific motion sequence in a new connection object. Reduce the `MinTurningRadius` if the robot is more maneuverable. Connect the poses again to get a different path.

```
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 $[x \ y \ \text{theta}]$ vectors.

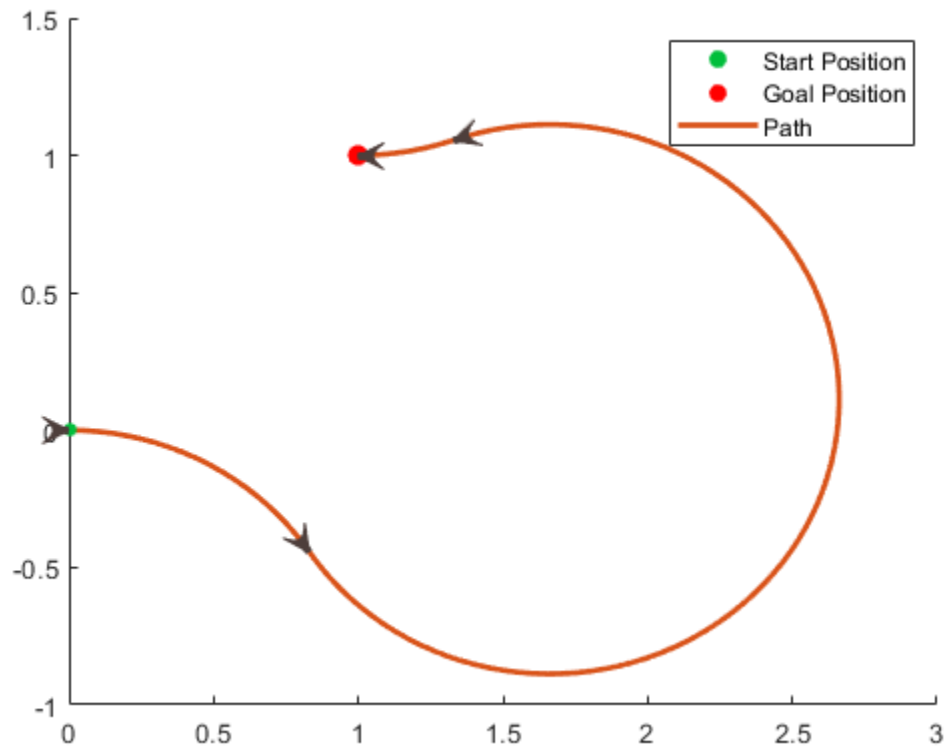
```
startPose = [0 0 0];  
goalPose = [1 1 pi];
```

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

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/s²)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/s²)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

<code>predict</code>	Update states using accelerometer and gyroscope data
<code>correct</code>	Correct states using direct state measurements
<code>residual</code>	Residuals and residual covariances from direct state measurements
<code>fusegps</code>	Correct states using GPS data
<code>residualgps</code>	Residuals and residual covariance from GPS measurements
<code>fusemvo</code>	Correct states using monocular visual odometry
<code>residualmvo</code>	Residuals and residual covariance from monocular visual odometry measurements
<code>pose</code>	Current orientation and position estimate
<code>reset</code>	Reset internal states
<code>stateinfo</code>	Display state vector information

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');
```

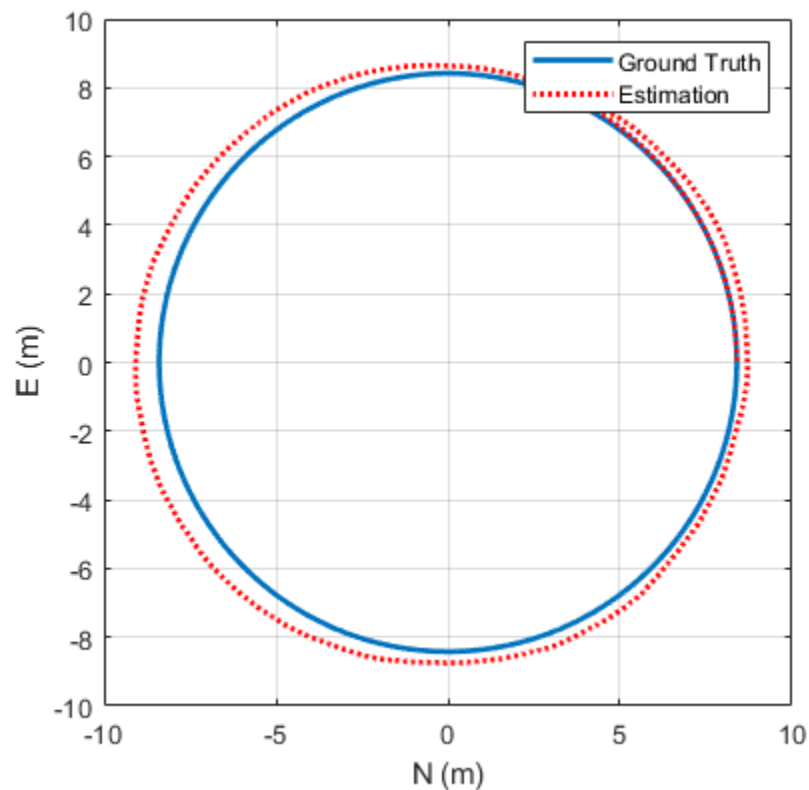
```
Quaternion Distance RMS Error
```

```
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{array}{l}
 \left[\begin{array}{l}
 q_0 + \Delta t * q_1(\text{gyrobias}_X/2 - \text{gyro}_X/2) + \Delta t * q_2 * (\text{gyrobias}_Y/2 - \text{gyro}_Y/2) + \Delta t * q_3 * (\text{gyrobias}_Z/2 - \text{gyro}_Z/2) \\
 q_1 - \Delta t * q_0(\text{gyrobias}_X/2 - \text{gyro}_X/2) + \Delta t * q_3 * (\text{gyrobias}_Y/2 - \text{gyro}_Y/2) - \Delta t * q_2 * (\text{gyrobias}_Z/2 - \text{gyro}_Z/2) \\
 q_2 - \Delta t * q_3(\text{gyrobias}_X/2 - \text{gyro}_X/2) - \Delta t * q_0 * (\text{gyrobias}_Y/2 - \text{gyro}_Y/2) + \Delta t * q_1 * (\text{gyrobias}_Z/2 - \text{gyro}_Z/2) \\
 q_3 + \Delta t * q_2(\text{gyrobias}_X/2 - \text{gyro}_X/2) - \Delta t * q_1 * (\text{gyrobias}_Y/2 - \text{gyro}_Y/2) - \Delta t * q_0 * (\text{gyrobias}_Z/2 - \text{gyro}_Z/2)
 \end{array} \right. \\
 \quad \text{position}_N + \Delta t * v_N \\
 \quad \text{position}_E + \Delta t * v_E \\
 \quad \text{position}_D + \Delta t * v_D \\
 v_N - \Delta t * \left[\begin{array}{l}
 q_0 * (q_0 * (\text{accelbias}_X - \text{accel}_X) - q_3 * (\text{accelbias}_Y - \text{accel}_Y) + q_2 * (\text{accelbias}_Z - \text{accel}_Z)) + g_N + \\
 q_2 * (q_1 * (\text{accelbias}_Y - \text{accel}_Y) - q_2 * (\text{accelbias}_X - \text{accel}_X) + q_0 * (\text{accelbias}_Z - \text{accel}_Z)) + \\
 q_1 * (q_1 * (\text{accelbias}_X - \text{accel}_X) + q_2 * (\text{accelbias}_Y - \text{accel}_Y) + q_3 * (\text{accelbias}_Z - \text{accel}_Z)) - \\
 q_3 * (q_3 * (\text{accelbias}_X - \text{accel}_X) + q_0 * (\text{accelbias}_Y - \text{accel}_Y) - q_1 * (\text{accelbias}_Z - \text{accel}_Z))
 \end{array} \right] \\
 v_E - \Delta t * \left[\begin{array}{l}
 q_0 * (q_3 * (\text{accelbias}_X - \text{accel}_X) + q_0 * (\text{accelbias}_Y - \text{accel}_Y) - q_1 * (\text{accelbias}_Z - \text{accel}_Z)) + g_E - \\
 q_1 * (q_1 * (\text{accelbias}_Y - \text{accel}_Y) - q_2 * (\text{accelbias}_X - \text{accel}_X) + q_0 * (\text{accelbias}_Z - \text{accel}_Z)) + \\
 q_2 * (q_1 * (\text{accelbias}_X - \text{accel}_X) + q_2 * (\text{accelbias}_Y - \text{accel}_Y) + q_3 * (\text{accelbias}_Z - \text{accel}_Z)) + \\
 q_3 * (q_0 * (\text{accelbias}_X - \text{accel}_X) - q_3 * (\text{accelbias}_Y - \text{accel}_Y) + q_2 * (\text{accelbias}_Z - \text{accel}_Z))
 \end{array} \right] \\
 \left[\begin{array}{l}
 q_0 * (q_1 * (\text{accelbias}_Y - \text{accel}_Y) - q_2 * (\text{accelbias}_X - \text{accel}_X) + q_0 * (\text{accelbias}_Z - \text{accel}_Z)) + g_D + \\
 q_1 * (q_2 * (\text{accelbias}_X - \text{accel}_X) - q_3 * (\text{accelbias}_Y - \text{accel}_Y) + q_2 * (\text{accelbias}_Z - \text{accel}_Z)) - \\
 q_2 * (q_3 * (\text{accelbias}_X - \text{accel}_X) + q_0 * (\text{accelbias}_Y - \text{accel}_Y) - q_1 * (\text{accelbias}_Z - \text{accel}_Z)) - \\
 q_3 * (q_0 * (\text{accelbias}_X - \text{accel}_X) - q_3 * (\text{accelbias}_Y - \text{accel}_Y) + q_2 * (\text{accelbias}_Z - \text{accel}_Z))
 \end{array} \right]
 \end{array}$$

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

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 $M = \text{numel}(\text{idx}) - 1$. If you do not correct orientation, then $M = \text{numel}(\text{idx})$.

Data Types: `single` | `double`

Extended Capabilities

C/C++ Code Generation

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

See Also

`insfilter` | `insfilterErrorState`

Introduced in R2019a

fusegps

Correct states using GPS data

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 (m²)

scalar | 3-element row vector | 3-by-3 matrix

Position measurement covariance of GPS receiver in m², 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

`insfilter` | `insfilterErrorState`

Introduced in R2019a

fusemvo

Correct states using monocular visual odometry

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 (m²)

scalar | 3-element vector | 3-by-3 matrix

Position measurement covariance of MVO in m², 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 (rad²)

scalar | 3-element vector | 3-by-3 matrix

Orientation measurement covariance of monocular visual odometry in rad², 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

`insfilter` | `insfilterErrorState`

Introduced in R2019a

predict

Update states using accelerometer and gyroscope data

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/s²)

3-element row vector

Accelerometer readings in m/s², 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` | `insfilterErrorState`

Introduced in R2019a

reset

Reset internal states

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

insfilter | insfilterErrorState

Introduced in R2019a

residual

Residuals and residual covariances from direct state measurements

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

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 (m²)

3-by-3 matrix

Position measurement covariance of GPS receiver in m², 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/s²)

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

Syntax

```
[pResidual,oResidual,resCov] = fusemvo(FUSE,position,positionCovariance,ornt,orntCovariance)
```

Description

[pResidual,oResidual,resCov] = fusemvo(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 (m²)

scalar | 3-element vector | 3-by-3 matrix

Position measurement covariance of MVO in m², 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 (rad²)

scalar | 3-element vector | 3-by-3 matrix

Orientation measurement covariance of monocular visual odometry in rad², 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 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

`insfilter` | `insfilterErrorState`

Introduced in R2020a

stateinfo

Display state vector information

Syntax

```
stateinfo(FUSE)
```

Description

`stateinfo(FUSE)` displays the meaning of each index of the `State` property of `FUSE`, an `insfilterErrorState` object, and the associated units.

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

`insfilter` | `insfilterErrorState`

Introduced in R2019a

pose

Current orientation and position estimate

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 in local NED coordinate system (m)

3-element row vector

Position estimate in the local NED coordinate system in meters, returned as a 3-element row vector.

Data Types: `single` | `double`

orientation — Orientation estimate in local NED coordinate system

quaternion (default) | 3-by-3 rotation matrix

Orientation estimate in the local NED coordinate system, returned as a scalar quaternion or 3-by-3 rotation matrix, depending on the specified orientation format. The quaternion or rotation matrix represents a frame rotation from the local NED reference frame to the body reference frame.

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

velocity — Velocity estimate in local NED coordinate system (m/s)

3-element row vector

Velocity estimate in the local NED coordinate system 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` | `insfilterErrorState`

Introduced in R2019a

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? \(MATLAB\)](#).

Creation

Syntax

```
GPS = gpsSensor
GPS = gpsSensor('ReferenceFrame',RF)
GPS = gpsSensor( ____,Name,Value)
```

Description

`GPS = gpsSensor` returns a `gpsSensorSystem` 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 `gpsSensorSystem` 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 (MATLAB).

UpdateRate — 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`

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 local navigation coordinate system (m)**

N-by-3 matrix

Position of the GPS receiver in the local navigation coordinate system in meters, specified as a real finite *N*-by-3 matrix.

N is the number of samples in the current frame.

Data Types: `single` | `double`

trueVelocity — Velocity of GPS receiver in local navigation coordinate system (m/s)

N-by-3 matrix

Velocity of GPS receiver in the local 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.

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.

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('UpdateRate', fs, 'ReferenceLocation', refLoc);
```

Call `gps` with the specified `truePosition` and `trueVelocity` to simulate receiving GPS data for a stationary platform.

```
position = gps(truePosition, trueVelocity);
```

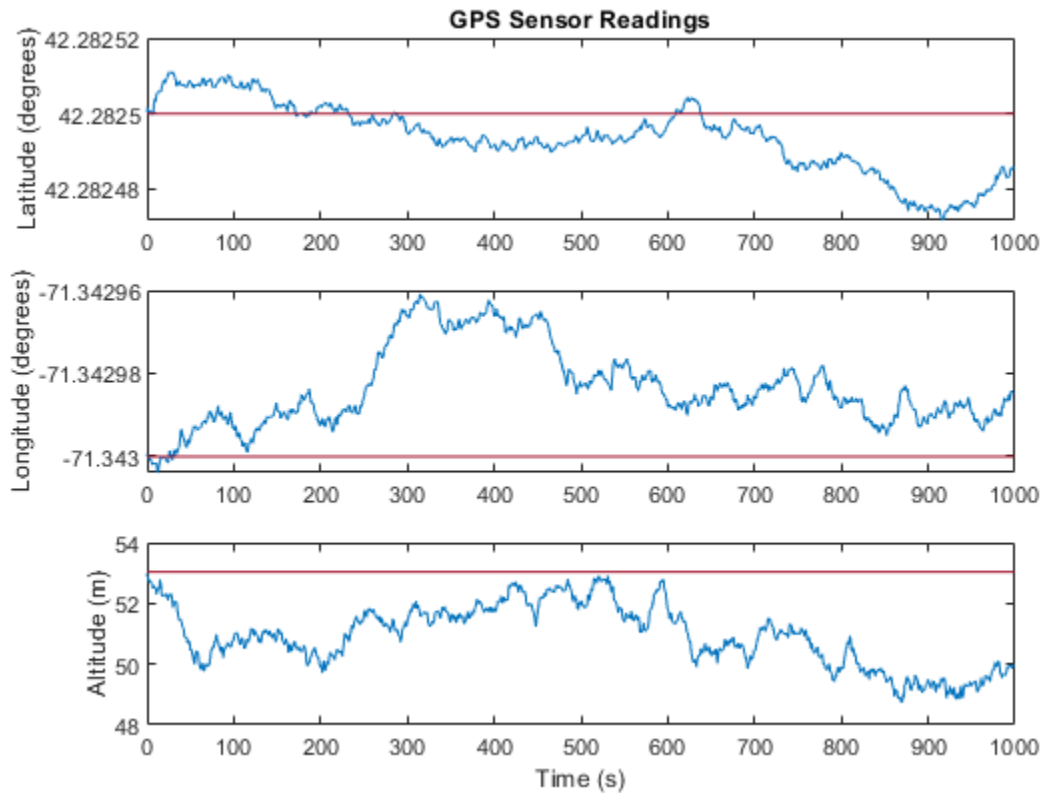
Plot the true position and the GPS sensor readings for position.

```
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);

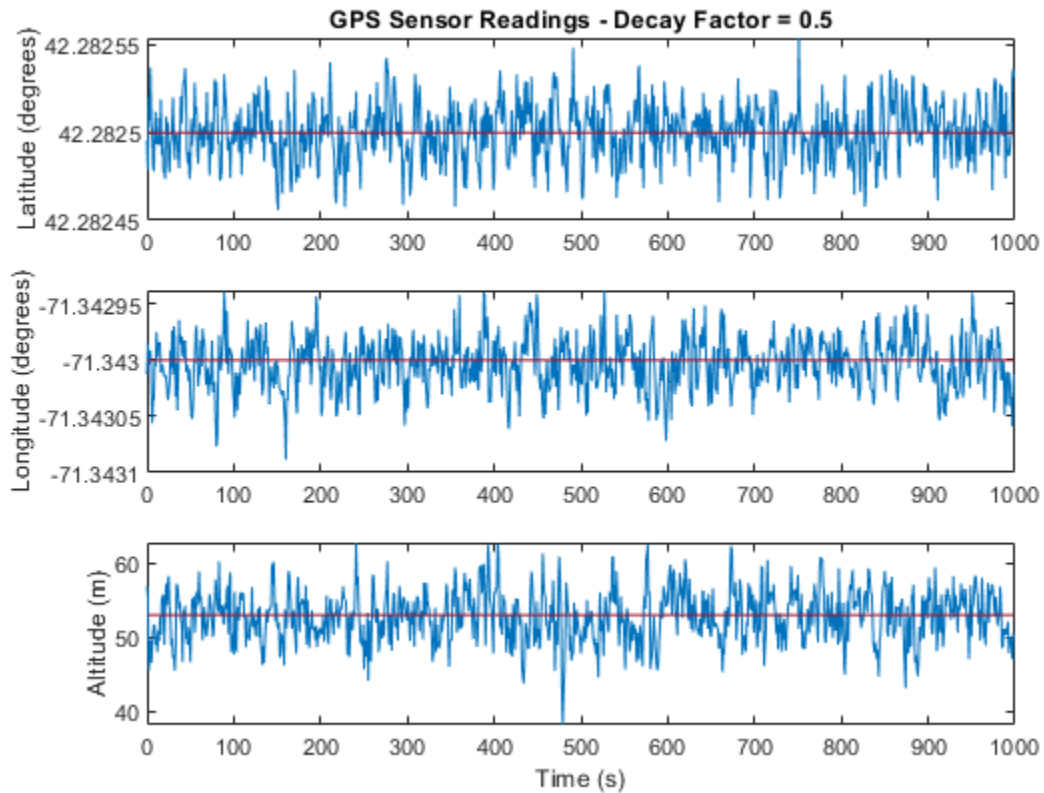
subplot(3, 1, 1)
plot(t, position(:,1), ...
     t, ones(numSamples)*refLoc(1))
title('GPS Sensor Readings - Decay Factor = 0.5')
ylabel('Latitude (degrees)')

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:
```

```

          UpdateRate: 1                Hz
      ReferenceLocation: [0 0 0]        [deg deg m]
HorizontalPositionAccuracy: 1.6         m
  VerticalPositionAccuracy: 3           m
          VelocityAccuracy: 0.1         m/s

```

```
RandomStream: 'Global stream'
DecayFactor: 0.999
```

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.UpdateRate;

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);
```

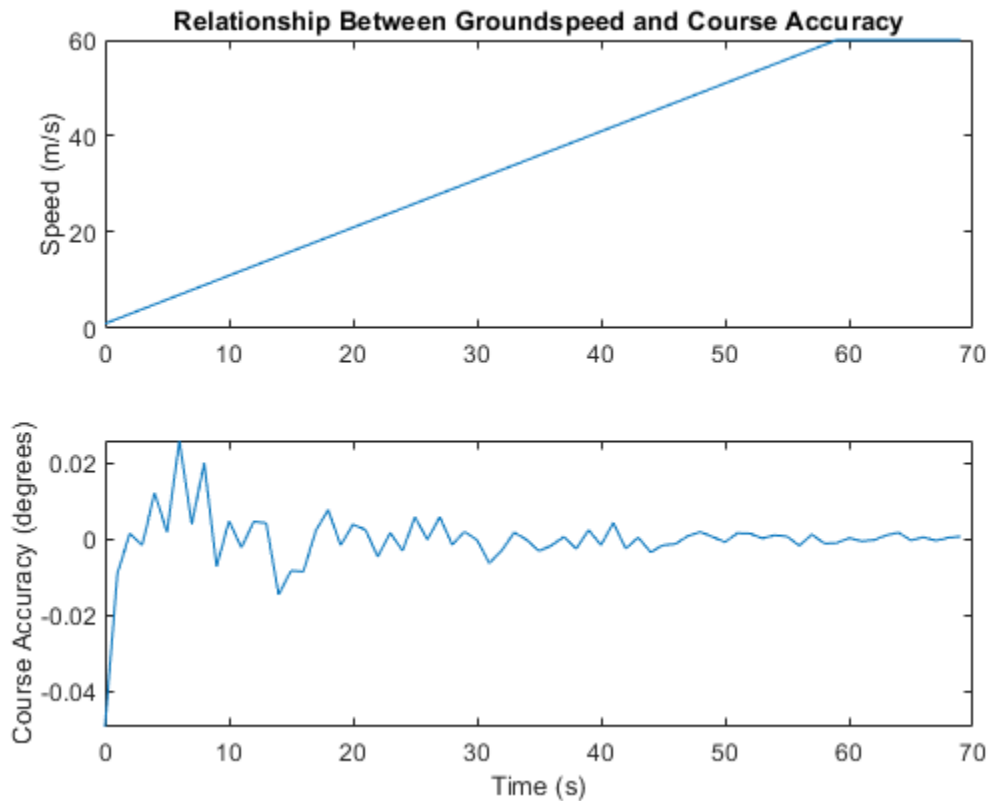
Plot the groundspeed and the difference between the true course and the course returned by the GPS simulator.

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.UpdateRate;

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( ...
    'HorizontalPositionalAccuracy',25, ...
    'DecayFactor',0.25, ...
    'UpdateRate',fs, ...
    'ReferenceLocation',NatickLLA);
```

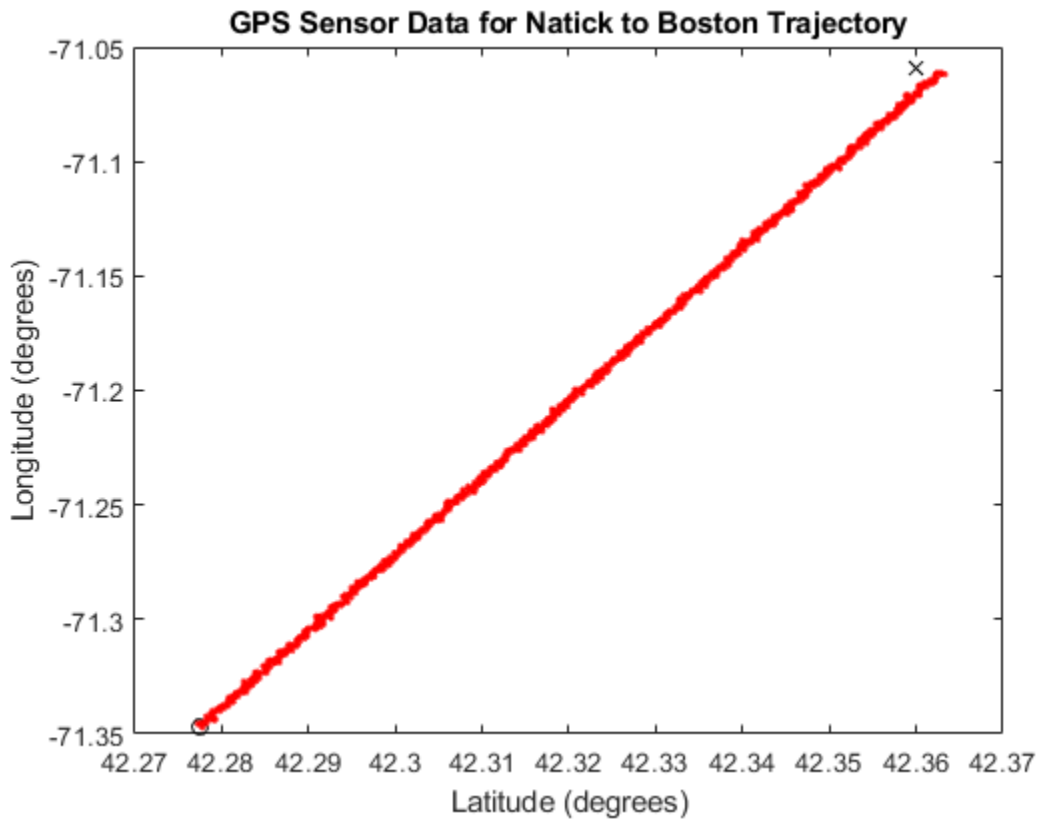
Open a figure and plot the position of Natick and Boston in LLA. Ignore altitude for simplicity.

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(trajjectory)
```

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

System Objects

imuSensor

Introduced in R2018b

gyroparams class

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.

Construction

`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

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 (%)

[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 %, 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

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/s²)

[0 0 0] (default) | real scalar | real 3-element row vector

Sensor bias from linear acceleration in (rad/s)/(m/s²), 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 1.332×10^{-4} (rad/s)/LSB. The constant offset bias is 0.349 rad/s. The sensor has a power spectral density of 8.727×10^{-4} rad/s/ $\sqrt{\text{Hz}}$. The bias from temperature is 0.349 rad/s/ $^{\circ}\text{C}$. The bias from temperature is 0.349 (rad/s²)/ $^{\circ}\text{C}$. The scale factor error from temperature is 0.2%/ $^{\circ}\text{C}$. The sensor axes are skewed by 2%. The sensor bias from linear acceleration is 0.178×10^{-3} (rad/s)/(m/s²)

```
params = gyroparams('MeasurementRange',4.363,'Resolution',1.332e-04,'ConstantBias',0.349,'NoiseD
```

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);
```

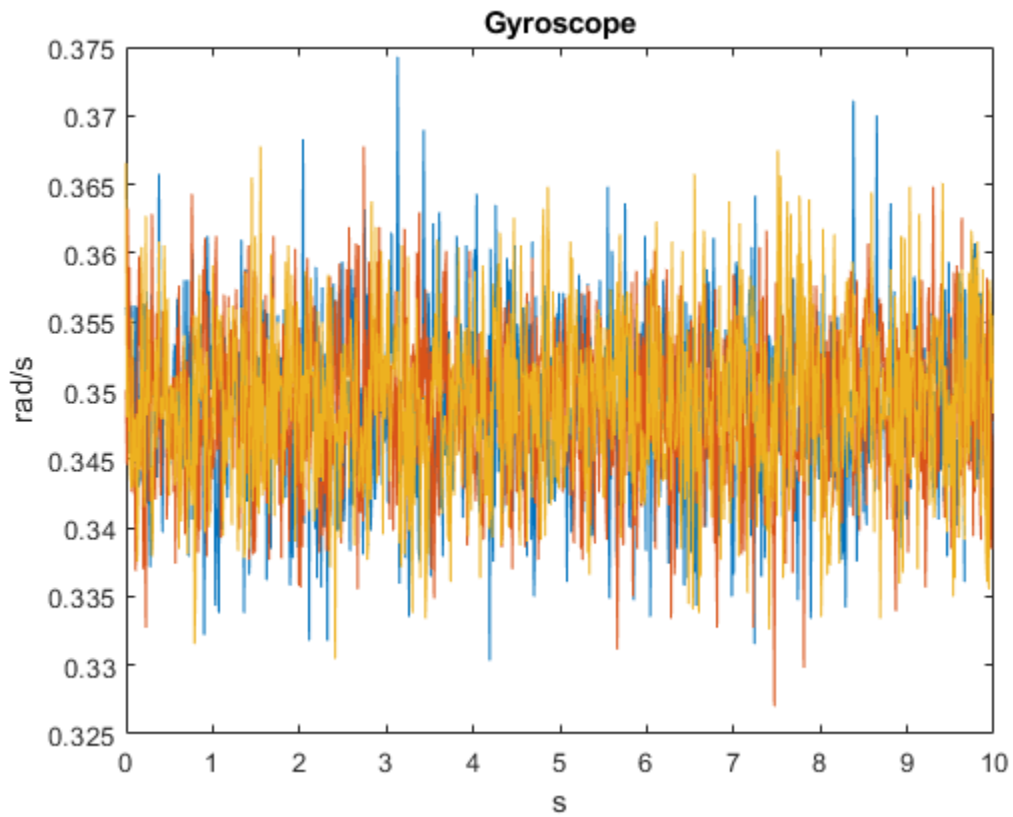
Generate gyroscope data from the imuSensor object.

```
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

Classes

[accelparams](#) | [magparams](#)

System Objects

[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? \(MATLAB\)](#).

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 (MATLAB).

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/s²)²)

0.00019247 (default) | positive real scalar

Variance of accelerometer signal noise in (m/s²)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/s²)²)

0.0096236 (default) | positive real scalar

Variance of linear acceleration noise in (m/s²)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

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 `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	0
0	0.000006092348396	0	0	0	0	0	0	0
0	0	0.000006092348396	0	0	0	0	0	0
0	0	0	0.000076154354947	0	0	0	0	0
0	0	0	0	0.000076154354947	0	0	0	0
0	0	0	0	0	0.000076154354947	0	0	0
0	0	0	0	0	0	0.000076154354947	0	0
0	0	0	0	0	0	0	0.000076154354947	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/s², 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 `OrientationFormat` 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)
```

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.

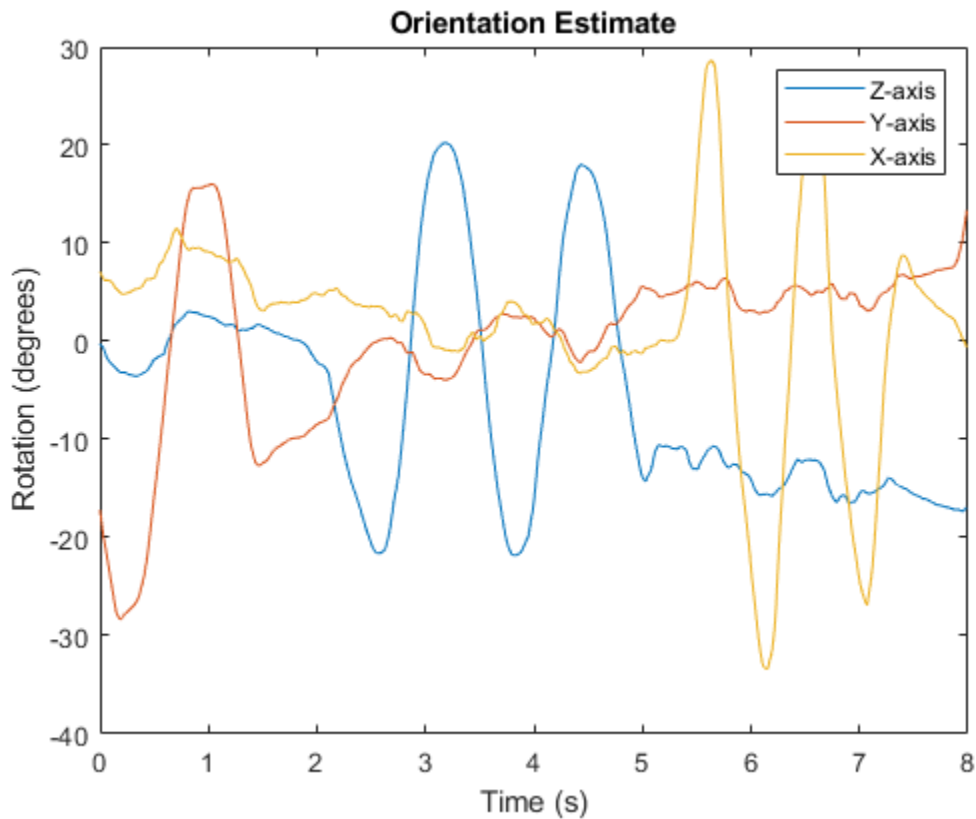
`imufilter` fusion correctly estimates the change in orientation from an assumed north-facing initial orientation. However, the device's *x*-axis was pointing southward when recorded. To correctly estimate the orientation relative to the true initial orientation or relative to NED, use `ahrsfilter`.

```

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);
```

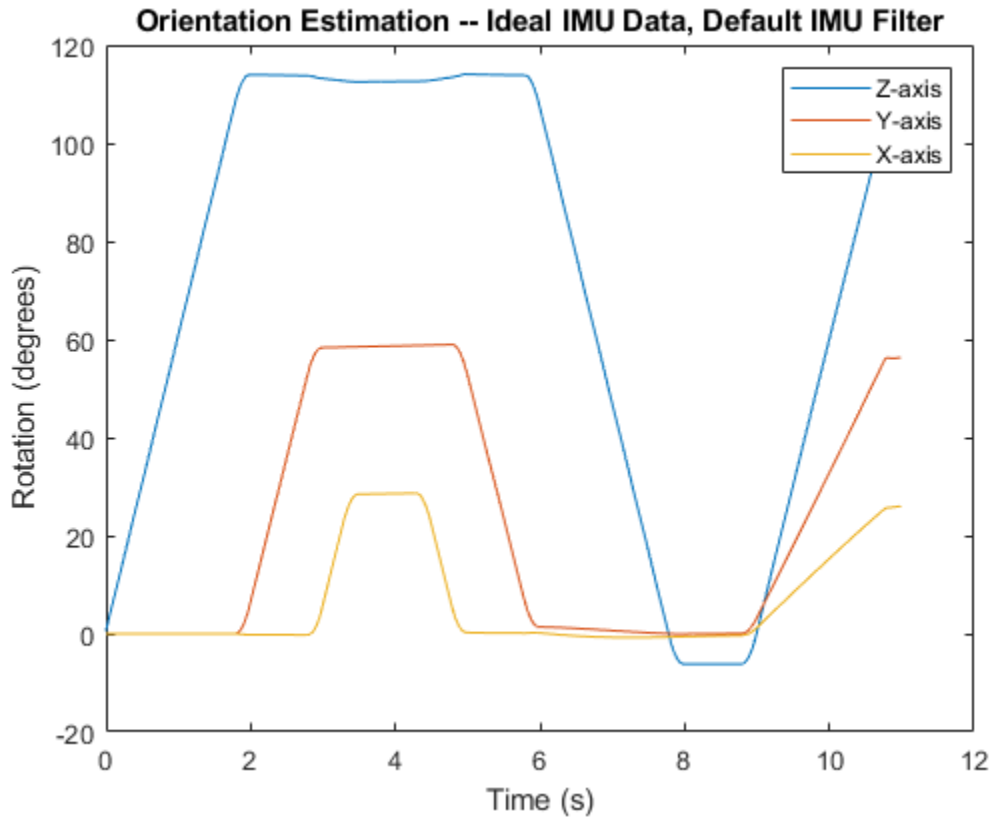
In a loop:

- 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,:));

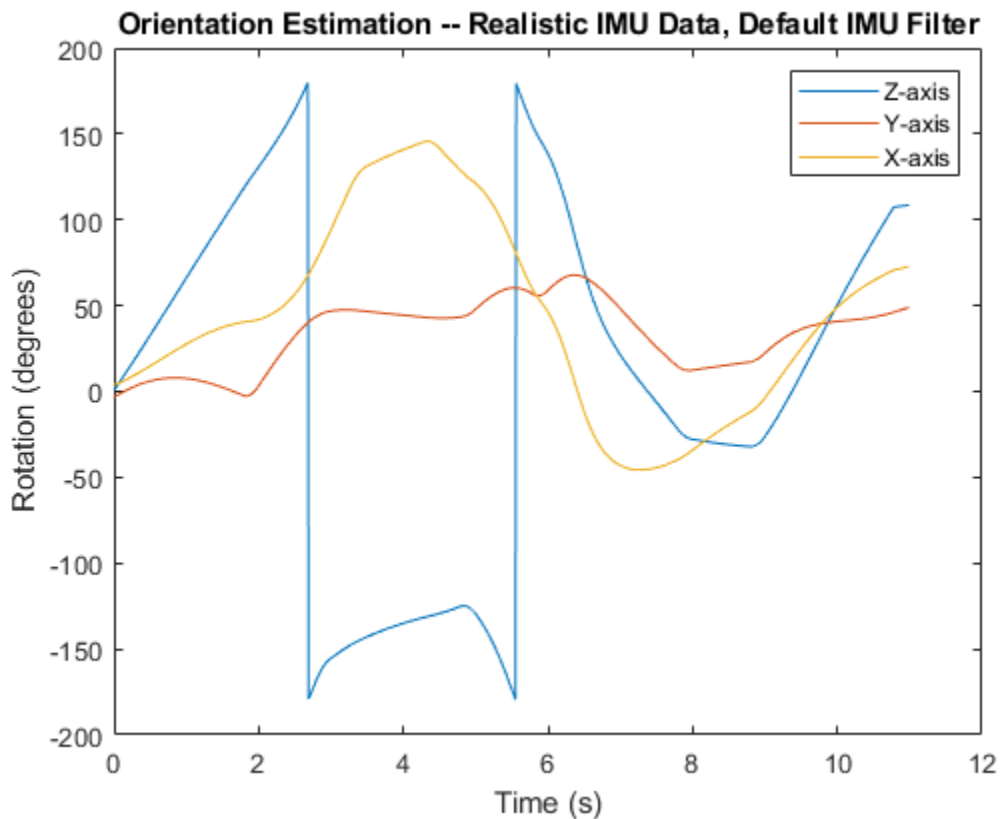
```

```

    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

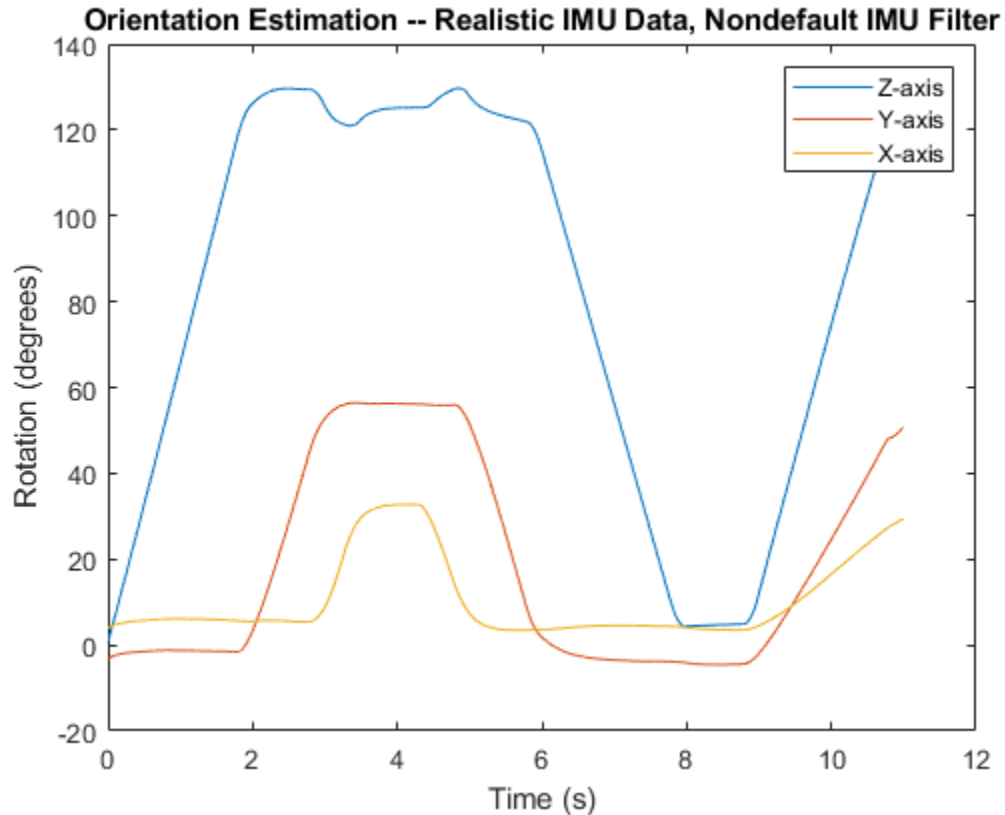
```

```

end
release(aFilter)

figure(3)
plot(t,eulder(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')

```



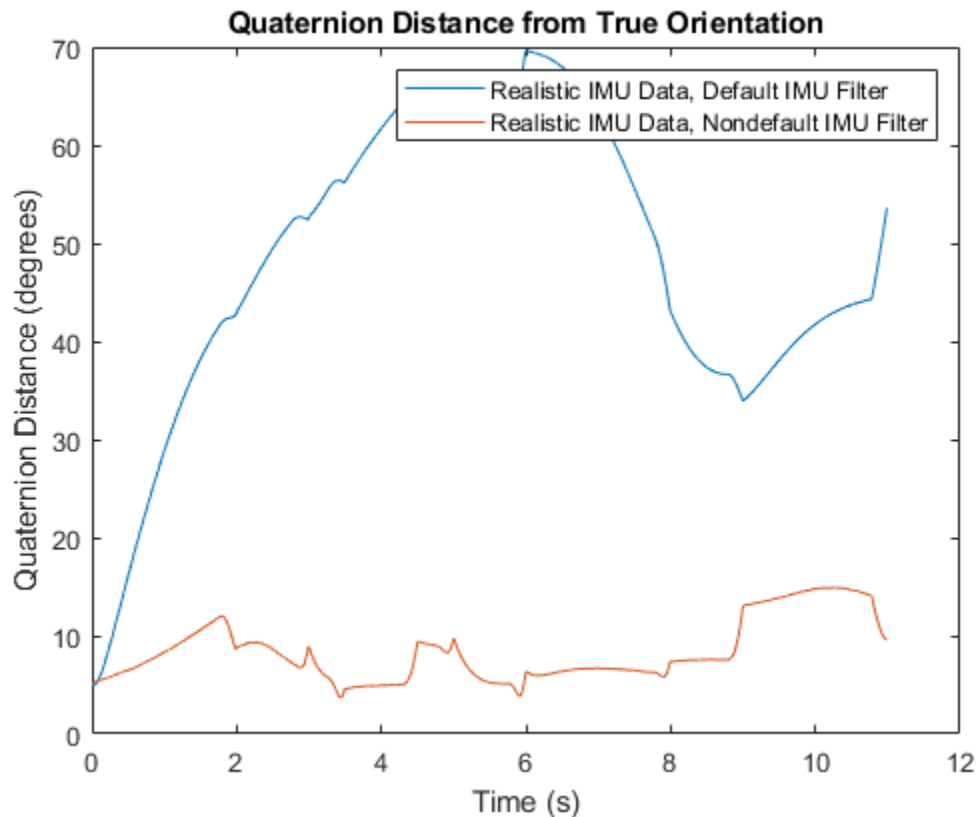
To quantify the improved performance of the modified `imufilter`, plot the quaternion distance between the ground-truth motion and the orientation as returned by the `imufilter` with default and nondefault properties.

```

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.

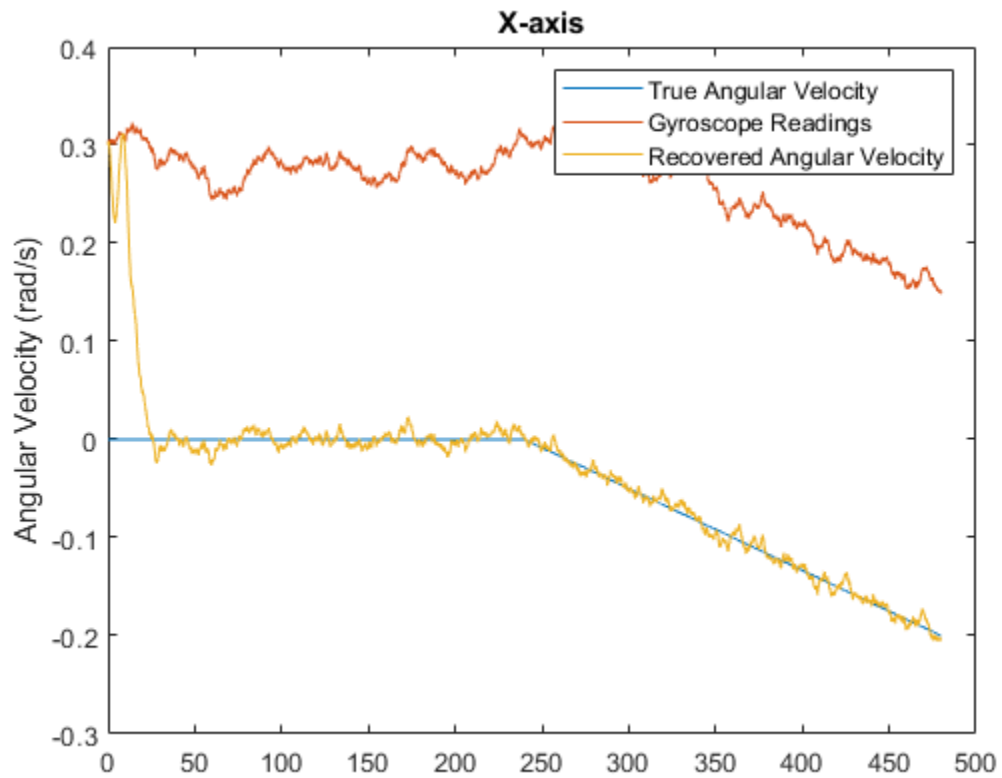
```
fuse = imufilter('SampleRate',fs, 'GyroscopeDriftNoise', 1e-6);
```

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

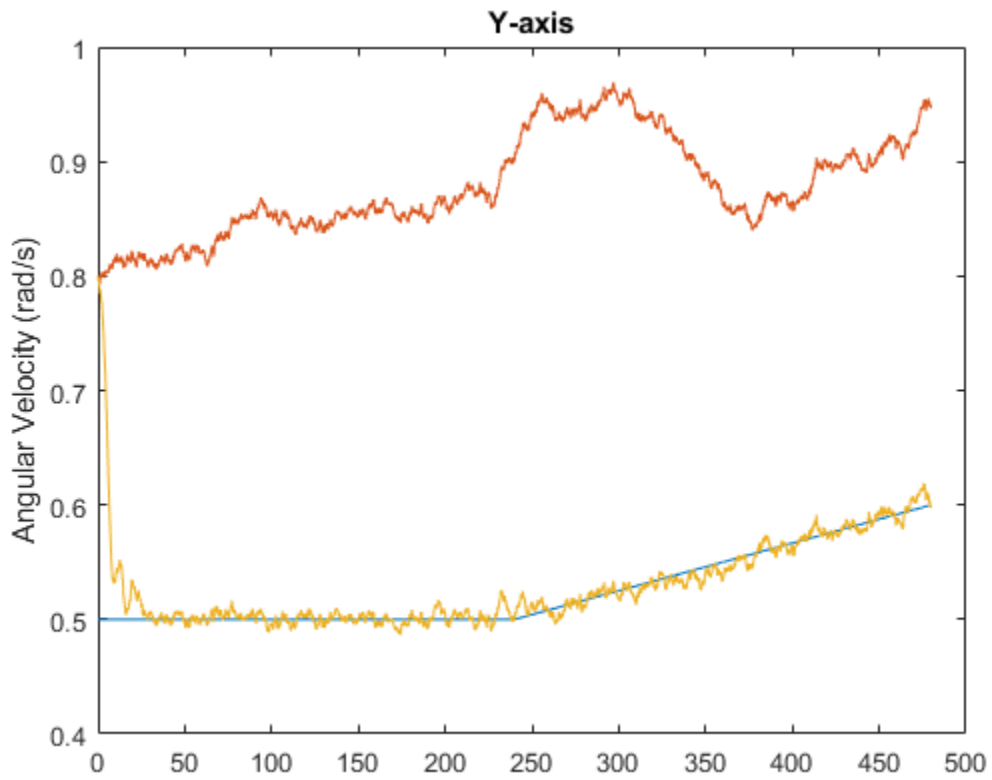
Plot the ground-truth angular velocity, the gyroscope readings, and the recovered angular velocity for each axis.

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

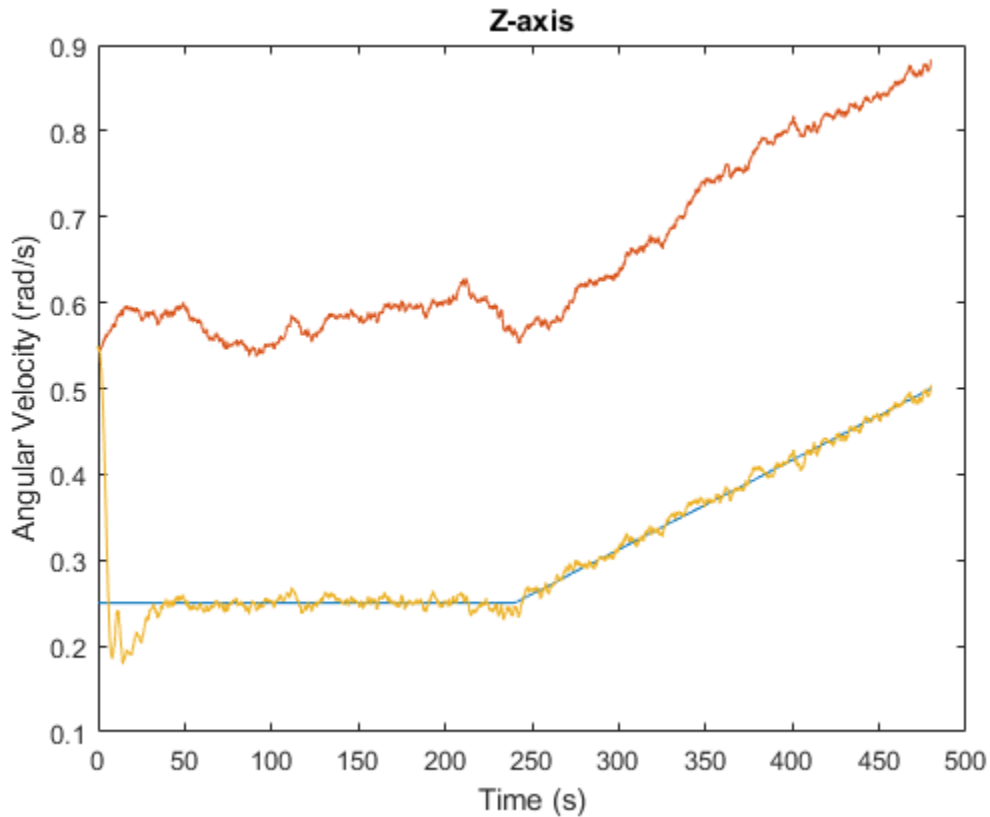
```
time = (0:numSamples-1)/fs;  
  
figure(1)  
plot(time,angVelBody(:,1), ...  
    time,gyroReadingsBody(:,1), ...  
    time,angVelBodyRecovered(:,1))  
title('X-axis')  
legend('True Angular Velocity', ...  
    'Gyroscope Readings', ...  
    'Recovered Angular Velocity')  
ylabel('Angular Velocity (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] (Sensor Fusion and Tracking Toolbox). 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:

$$\begin{aligned}x_k^- &= F_k x_{k-1}^+ \\P_k^- &= F_k P_{k-1}^+ F_k^T + Q_k \\y_k &= z_k - H_k x_k^- \\S_k &= R_k + H_k P_k^- H_k^T \\K_k &= P_k^- H_k^T (S_k)^{-1} \\x_k^+ &= x_k^- + K_k y_k \\P_k^+ &= P_k^- - K_k H_k P_k^-\end{aligned}$$

Kalman equations used in this algorithm:

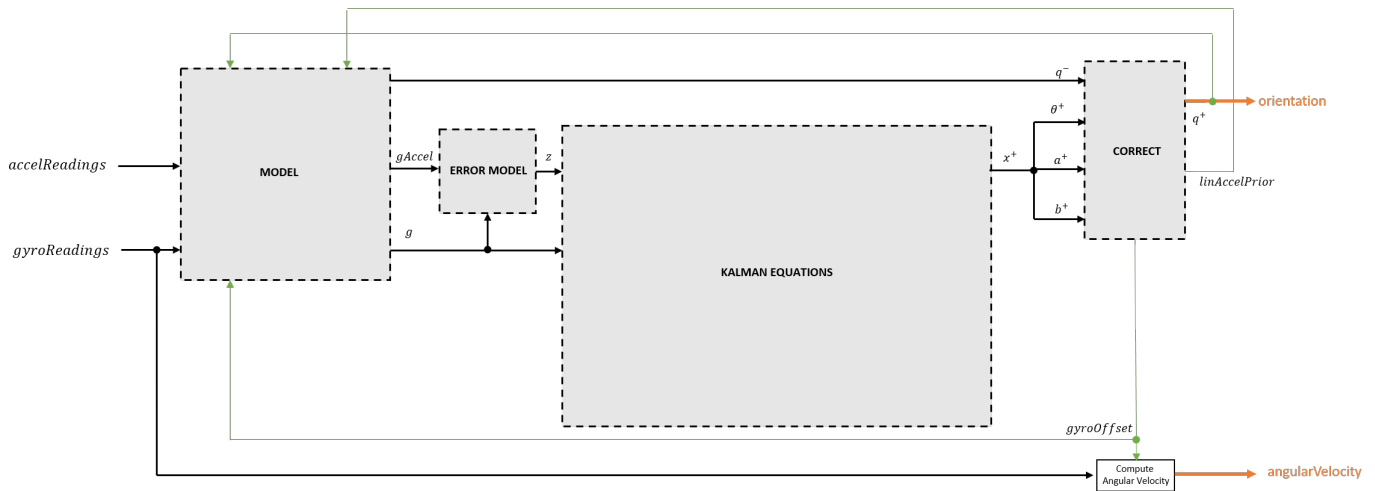
$$\begin{aligned}x_k^- &= 0 \\P_k^- &= Q_k \\y_k &= z_k \\S_k &= R_k + H_k P_k^- H_k^T \\K_k &= P_k^- H_k^T (S_k)^{-1} \\x_k^+ &= K_k y_k \\P_k^+ &= P_k^- - K_k H_k P_k^-\end{aligned}$$

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.

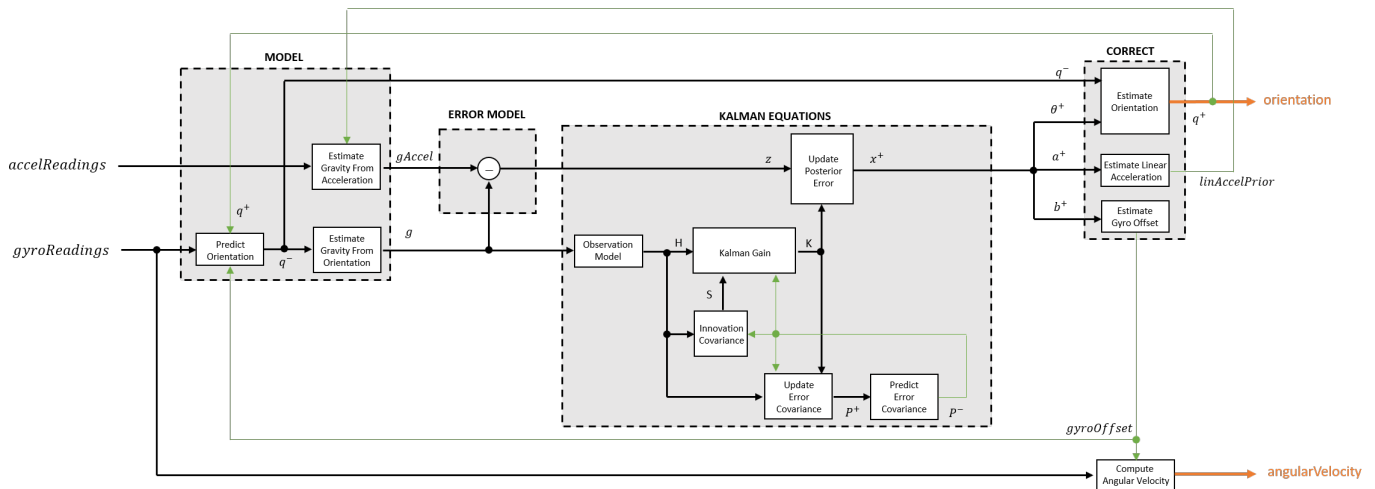
The graphic and following steps describe a single frame-based iteration through the algorithm.



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.

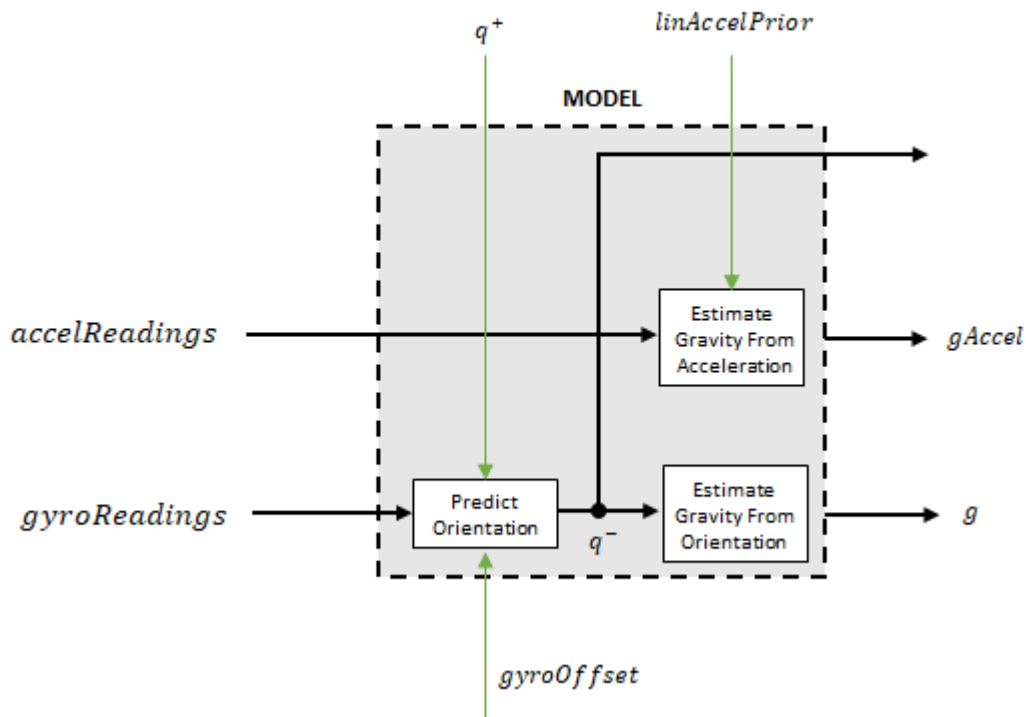
Detailed Overview

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 :

$$q_{1 \times 1}^- = (q_{1 \times 1}^+) \left(\prod_{n=1}^N \Delta Q_n \right)$$

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.

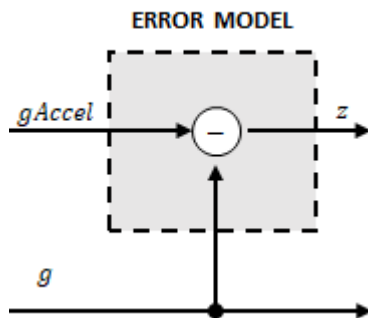
Estimate Gravity from Acceleration

A second gravity vector estimation is made by subtracting the decayed linear acceleration estimate of the previous iteration from the accelerometer readings:

$$g_{Accel}_{1 \times 3} = accelReadings_{1 \times 3} - linAccel_{prior}_{1 \times 3}$$

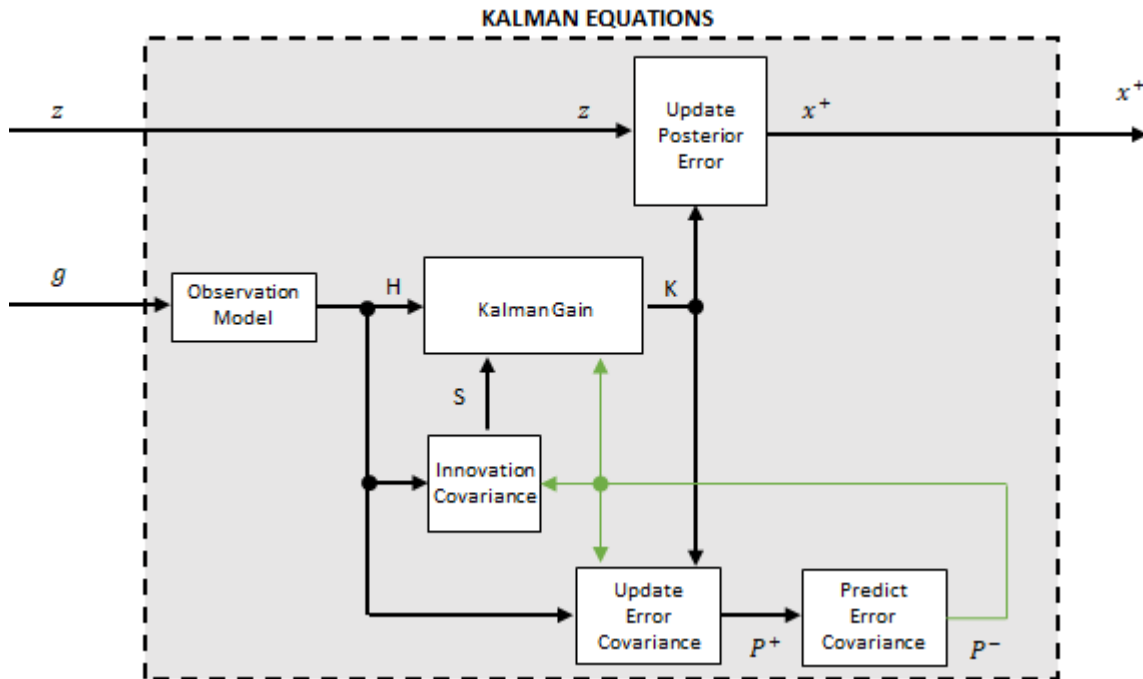
Error Model

The error model is the difference between the gravity estimate from the accelerometer readings and the gravity estimate from the gyroscope readings: $z = g - g_{Accel}$.



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: $\kappa = \text{DecimationFactor}/\text{SampleRate}$.

See sections 7.3 and 7.4 of [1] (Sensor Fusion and Tracking Toolbox) 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:

$$S_{3 \times 3} = R_{3 \times 3} + (H_{3 \times 9})(P_{9 \times 9}^-)(H_{3 \times 9})^T$$

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:

$$P_{9 \times 9}^+ = P_{9 \times 9}^- - (K_{9 \times 3})(H_{3 \times 9})(P_{9 \times 9}^-)$$

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:

$Q =$

$$\begin{bmatrix}
 P^+(1) + \kappa^2 P^+(31) + \beta + \eta & 0 & 0 & -\kappa(P^+(31) + \beta) & 0 \\
 0 & P^+(11) + \kappa^2 P^+(41) + \beta + \eta & 0 & 0 & P^+(41) + \beta \\
 0 & 0 & P^+(21) + \kappa^2 P^+(51) + \beta + \eta & 0 & 0 \\
 -\kappa(P^+(31) + \beta) & 0 & 0 & P^+(31) + \beta & 0 \\
 0 & P^+(41) + \beta & 0 & 0 & P^+(41) + \beta \\
 0 & 0 & P^+(51) + \beta & 0 & 0 \\
 0 & 0 & 0 & 0 & 0 \\
 0 & 0 & 0 & 0 & 0 \\
 0 & 0 & 0 & 0 & 0
 \end{bmatrix}$$

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] (Sensor Fusion and Tracking Toolbox) 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:

$$K_{9 \times 3} = (P_{9 \times 9}^-)(H_{3 \times 9})^T((S_{3 \times 3})^T)^{-1}$$

where

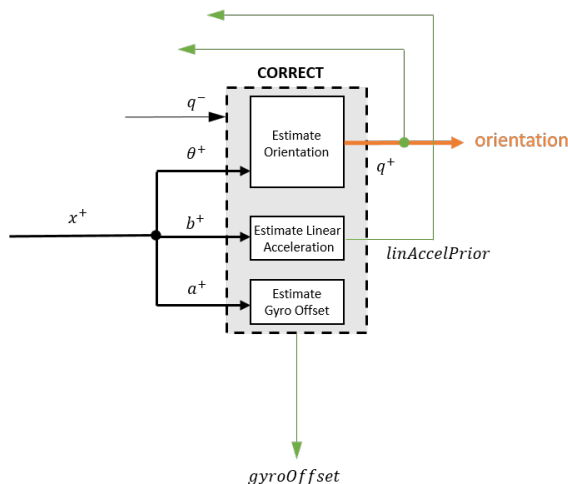
- P -- predicted error covariance
- H -- observation model
- S -- innovation covariance

Update a Posteriori Error

The *a posteriori* error estimate is determined by combining the Kalman gain matrix with the error in the gravity vector estimations:

$$x_{9 \times 1} = (K_{9 \times 3})(z_{1 \times 3})^T$$

Correct



Estimate Orientation

The orientation estimate is updated by multiplying the previous estimation by the error:

$$q^+ = (q^-)(\theta^+)$$

Estimate Linear Acceleration

The linear acceleration estimation is updated by decaying the linear acceleration estimation from the previous iteration and subtracting the error:

$$\text{linAccelPrior} = (\text{linAccelPrior}_{k-1})\nu - b^+$$

where

- ν -- LinearAccelerationDecayFactor

Estimate Gyroscope Offset

The gyroscope offset estimation is updated by subtracting the gyroscope offset error from the gyroscope offset from the previous iteration:

$$\text{gyroOffset} = \text{gyroOffset}_{k-1} - a^+$$

Compute Angular Velocity

To estimate angular velocity, the frame of `gyroReadings` are averaged and the gyroscope offset computed in the previous iteration is subtracted:

$$\text{angularVelocity}_{1 \times 3} = \frac{\sum \text{gyroReadings}_{N \times 3}}{N} - \text{gyroOffset}_{1 \times 3}$$

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

`ahrsfilter` | `ecompass` | `gpsSensor` | `imuSensor`

Introduced in R2018b

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? \(MATLAB\)](#).

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 (MATLAB)*.

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 (°C)

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 °C 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/s²)

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 microtesla. 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

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: [0 0 0]    %

    NoiseDensity: [0 0 0]       (rad/s)/√Hz
    BiasInstability: [0 0 0]    rad/s
    RandomWalk: [0 0 0]         (rad/s)*√Hz

    TemperatureBias: [0 0 0]    (rad/s)/°C
    TemperatureScaleFactor: [0 0 0] %/°C
    AccelerationBias: [0 0 0]   (rad/s)/(m/s²)
```

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: [2 2 2]              %

    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 Ideal 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.

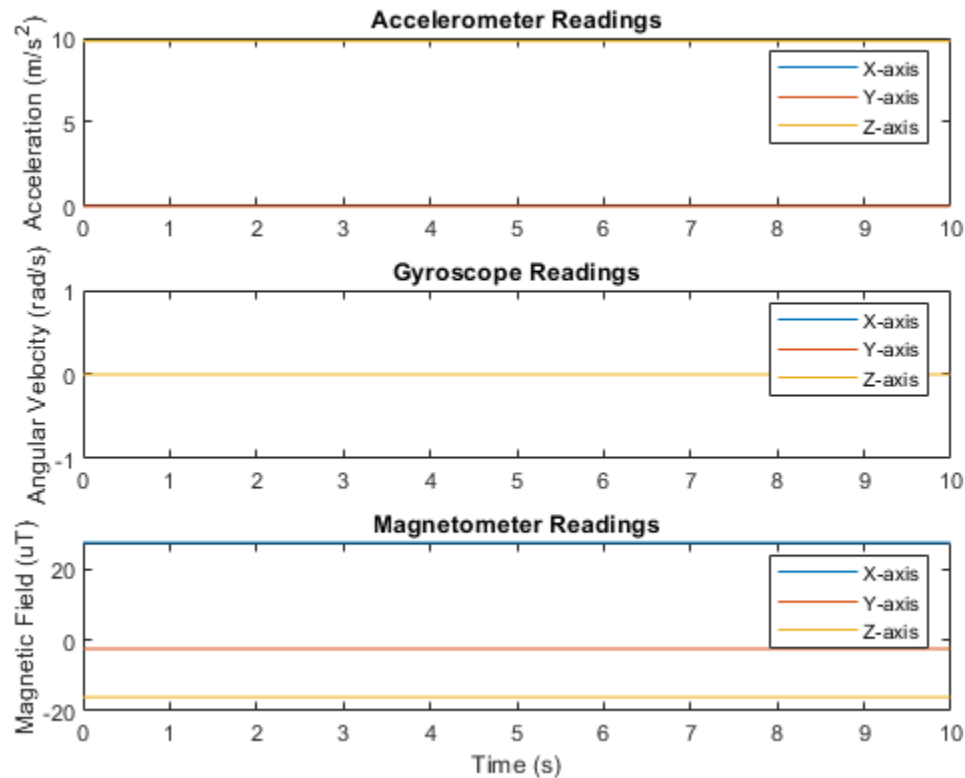
```
[accelReading,gyroReading,magReading] = IMU(acceleration,angularVelocity);
```

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.

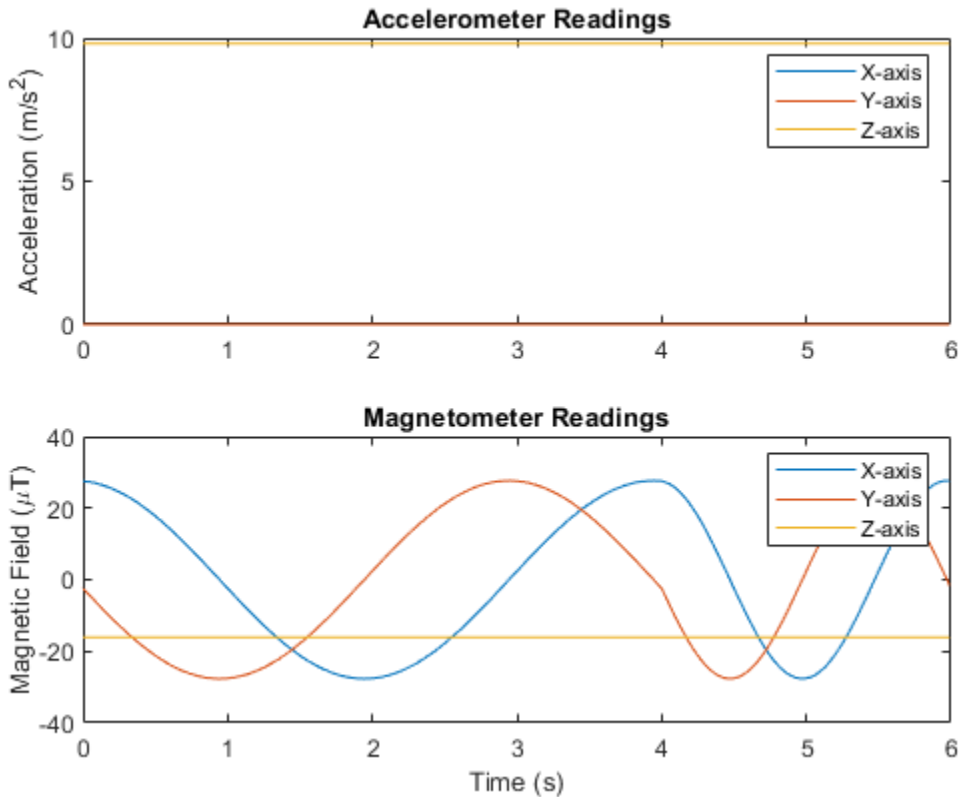
```
IMU = imuSensor('accel-mag','SampleRate',fs);

[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 (\u03bcT)')
xlabel('Time (s)')
title('Magnetometer Readings')
```

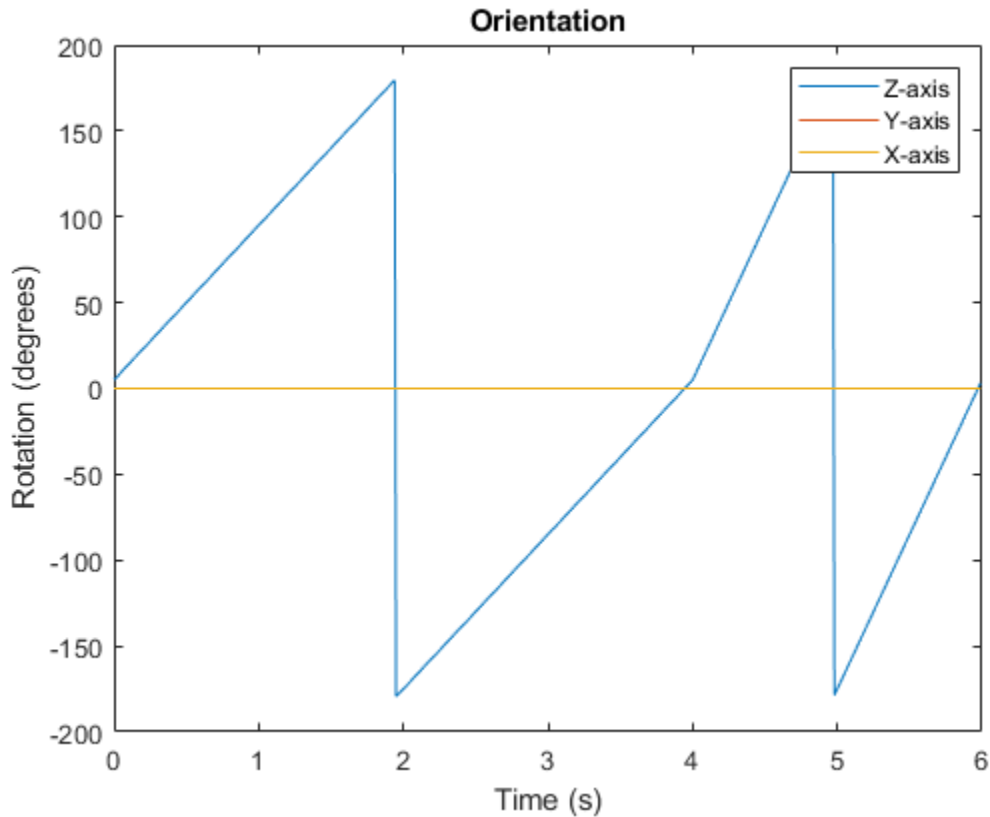


The accelerometer readings indicate that the platform has no translation. The magnetometer readings indicate that the platform is rotating around the z-axis.

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);

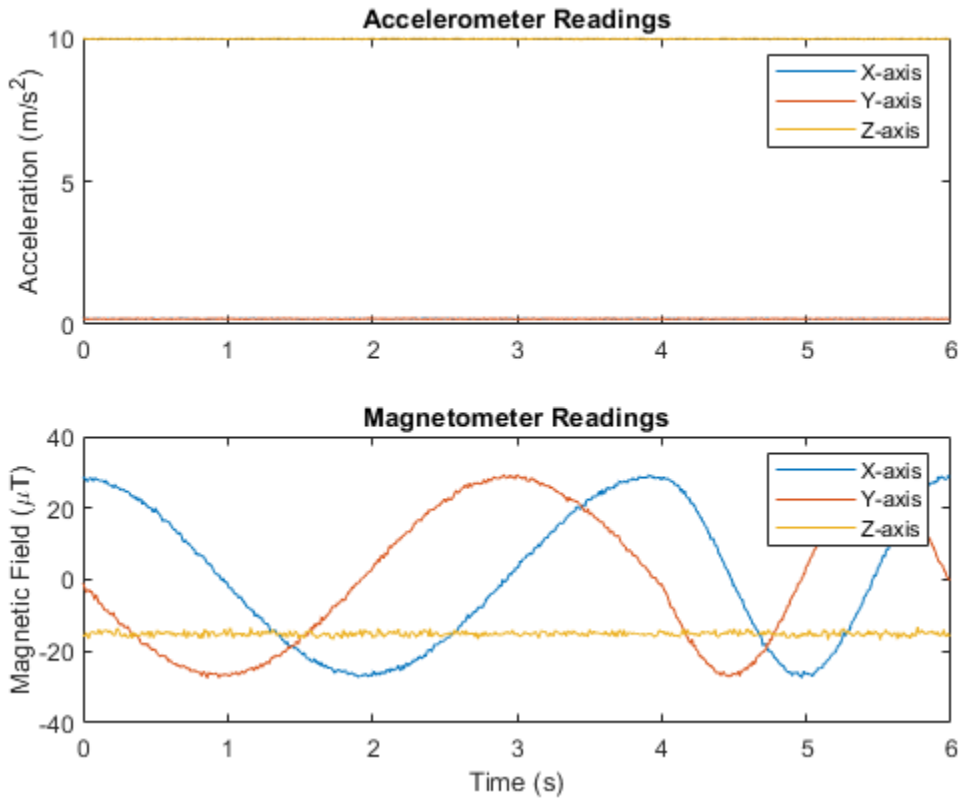
IMU.Accelerometer = accelparams( ...
    'MeasurementRange',19.62, ...           % m/s^2
    'Resolution',0.0023936, ...           % m/s^2 / LSB
    'TemperatureScaleFactor',0.008, ...    % % / degree C
    'ConstantBias',0.1962, ...           % m/s^2
    'TemperatureBias',0.0014715, ...      % m/s^2 / 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')
```



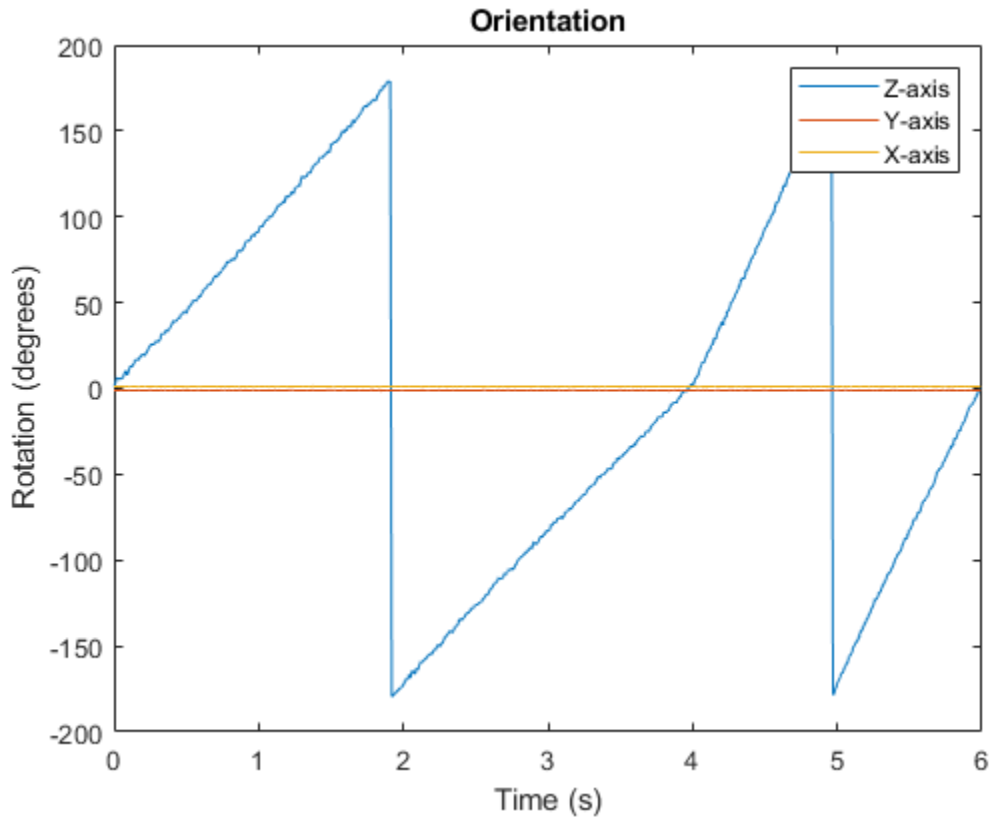
The accelerometer readings indicate that the platform has no translation. The magnetometer readings indicate that the platform is rotating around the z-axis.

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);
```

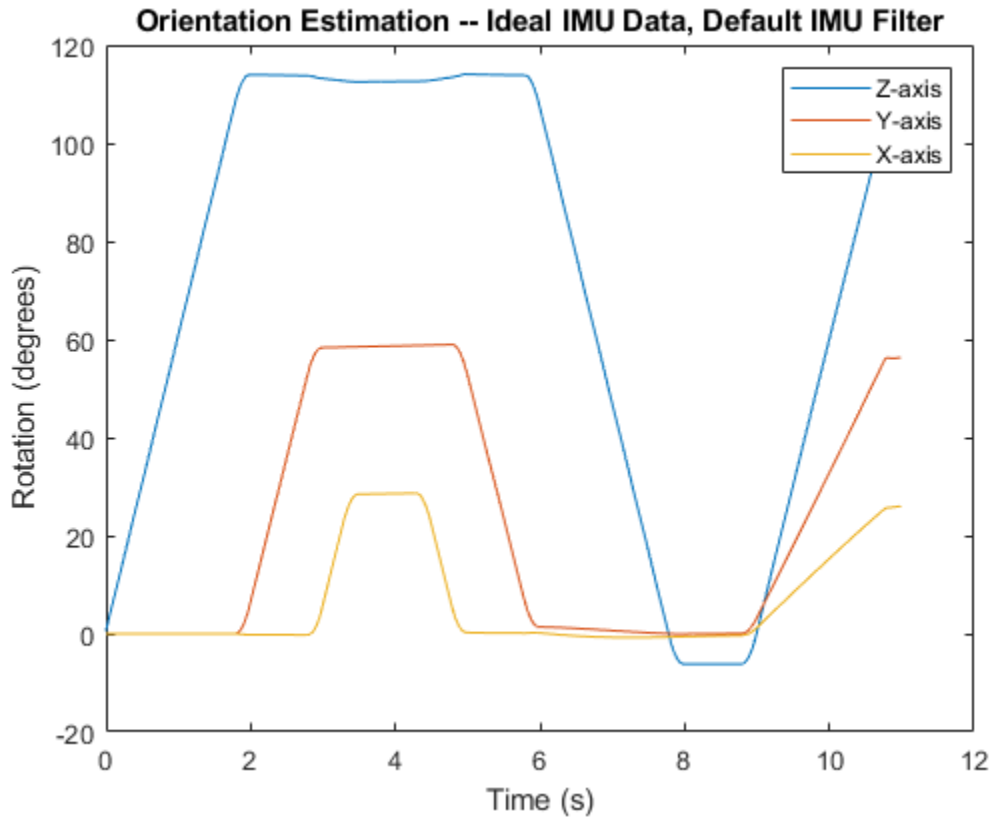
In a loop:

- 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,:));

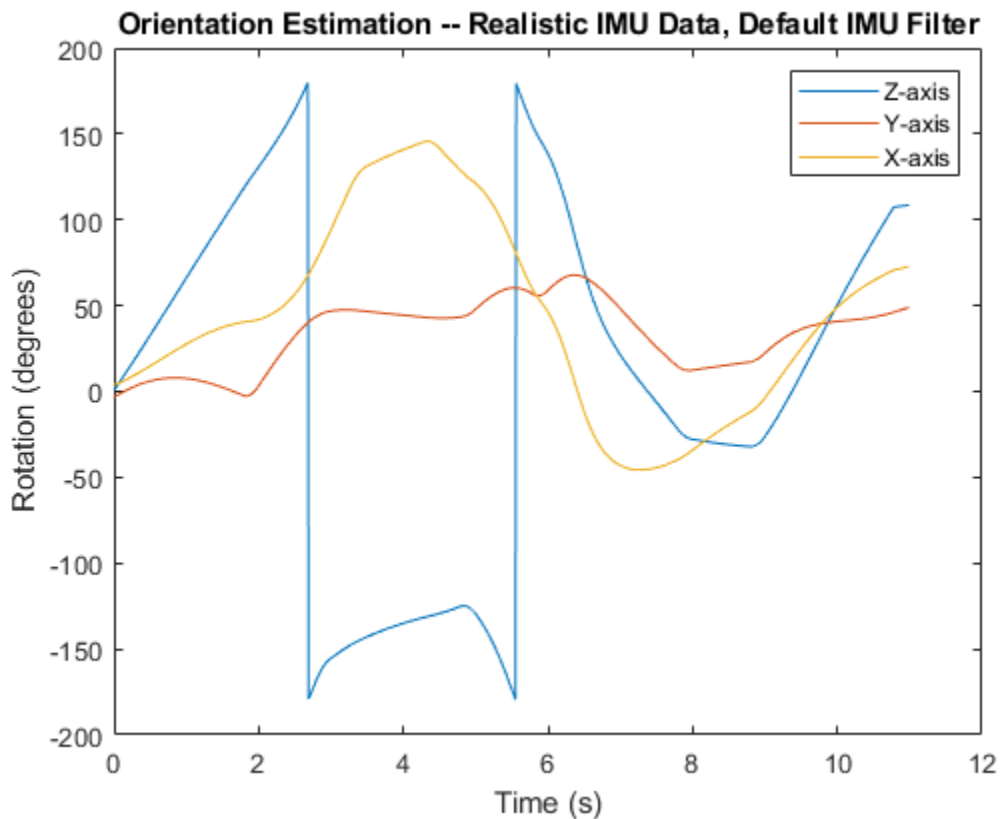
```

```

    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

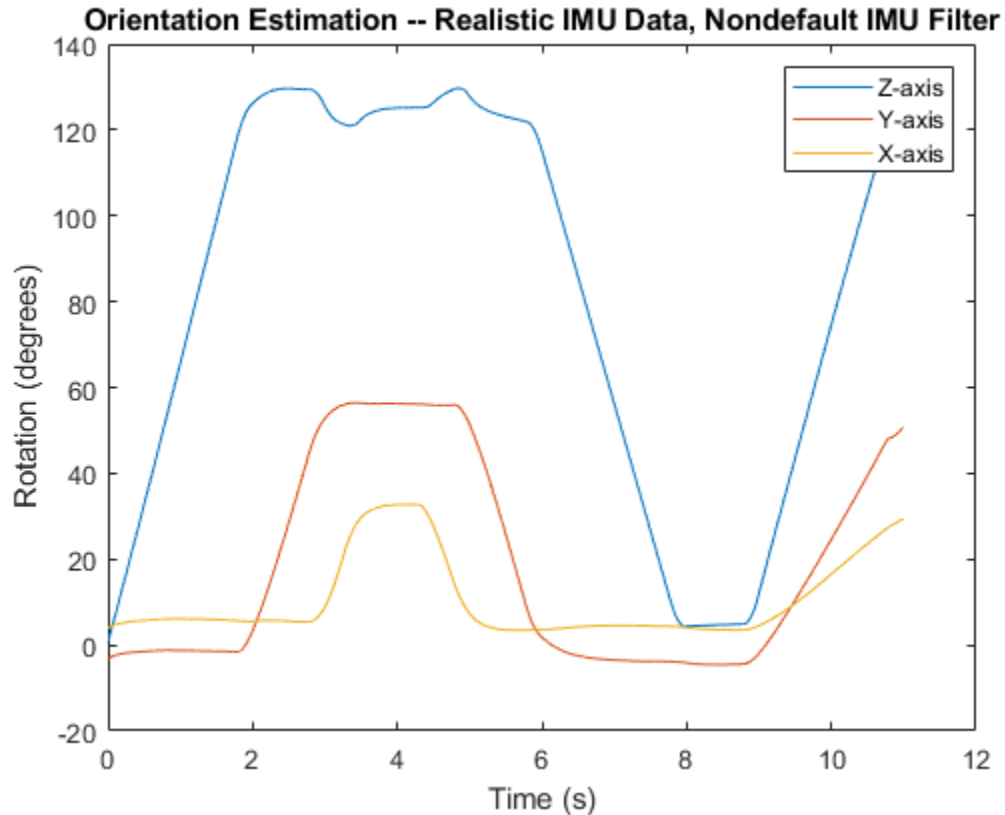
```

```

end
release(aFilter)

figure(3)
plot(t,eulder(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.

```

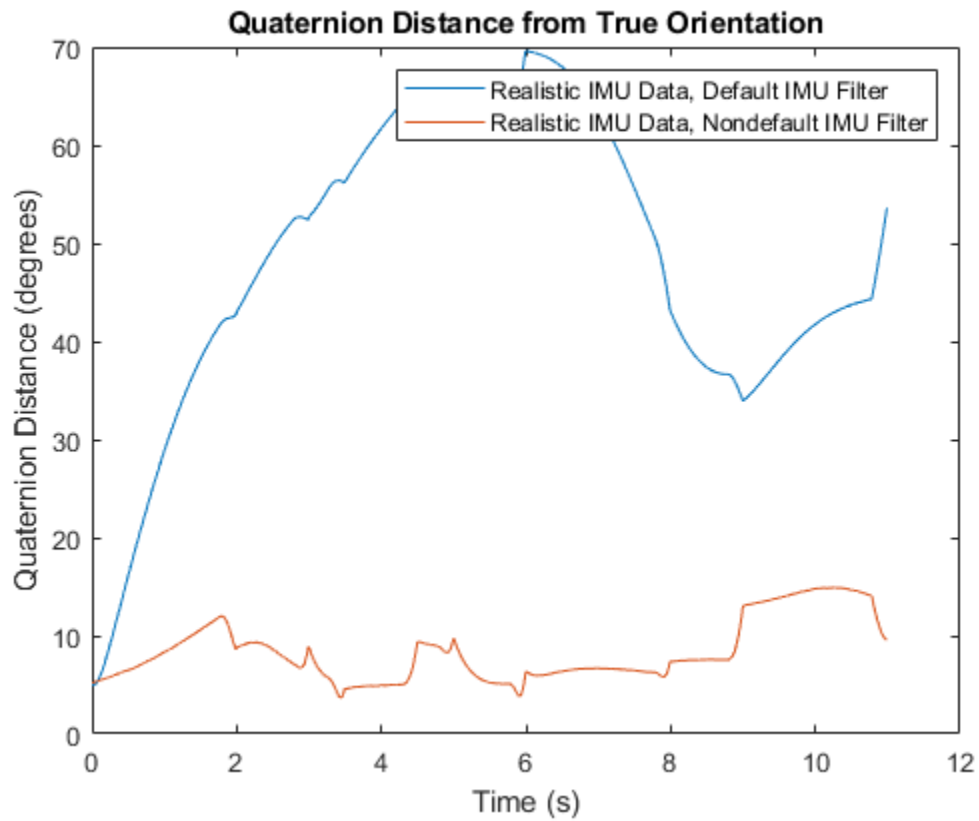
qDistDefault = rad2deg(dist(orientationNED,orientationDefault));
qDistNondefault = rad2deg(dist(orientationNED,orientationNondefault));

```

```

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)')

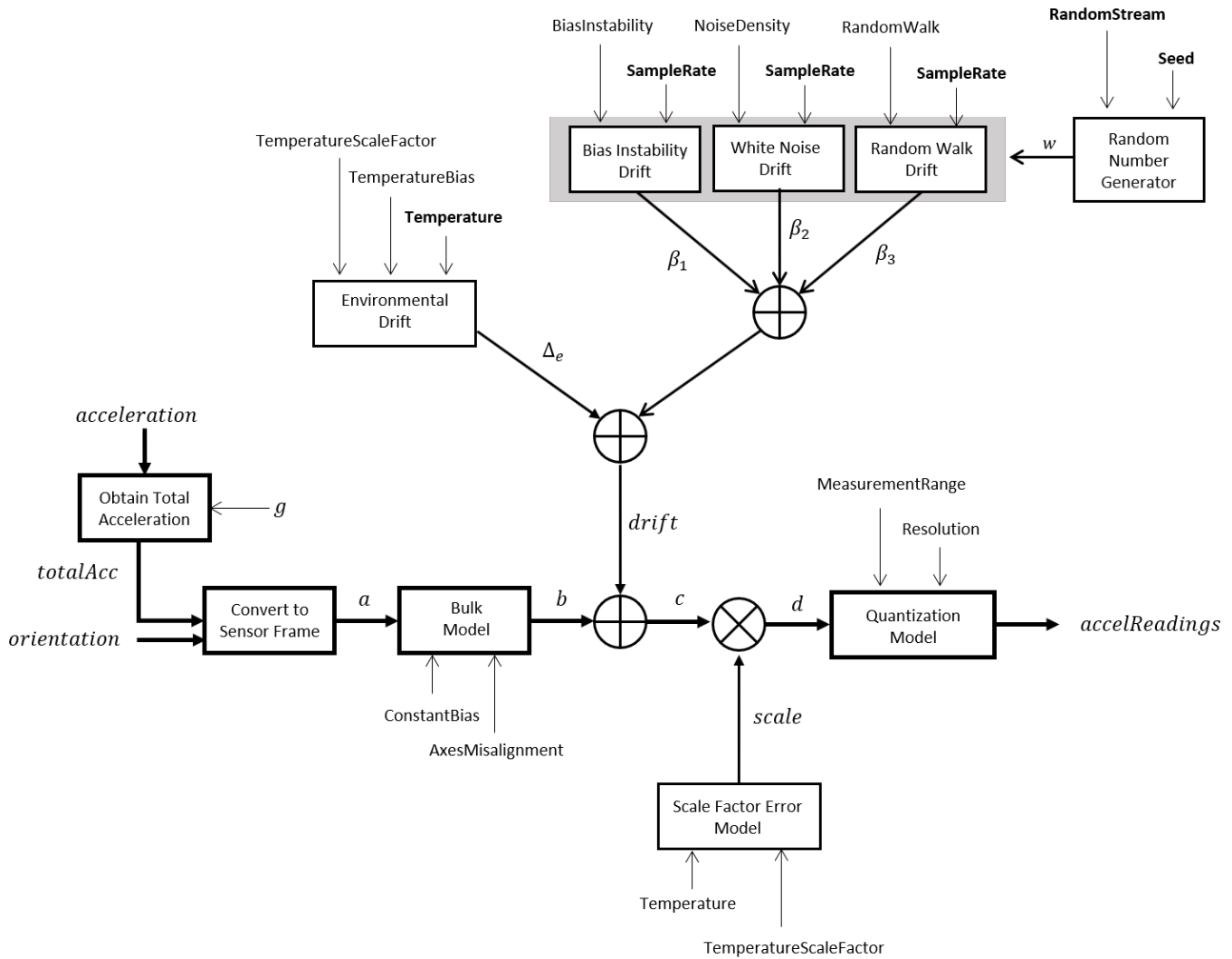
```



Algorithms

Accelerometer

The 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/s²) as:

$$totalAcc = - acceleration + g$$

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:

$$b = \left(\begin{bmatrix} 1 & \frac{\alpha_2}{100} & \frac{\alpha_3}{100} \\ \frac{\alpha_1}{100} & 1 & \frac{\alpha_3}{100} \\ \frac{\alpha_1}{100} & \frac{\alpha_2}{100} & 1 \end{bmatrix} (a^T) \right)^T + \text{ConstantBias}$$

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:

$$H_1(z) = \frac{1}{1 + \left(\frac{2}{\text{SampleRate}} - 1 \right) z^{-1}}$$

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:

$$scaleFactorError = 1 + \left(\frac{Temperature - 25}{100} \right) (TemperatureScaleFactor)$$

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:

$$e = \begin{cases} \text{MeasurementRange} & \text{if } d > \text{MeasurementRange} \\ -\text{MeasurementRange} & \text{if } -d > \text{MeasurementRange} \\ d & \text{else} \end{cases}$$

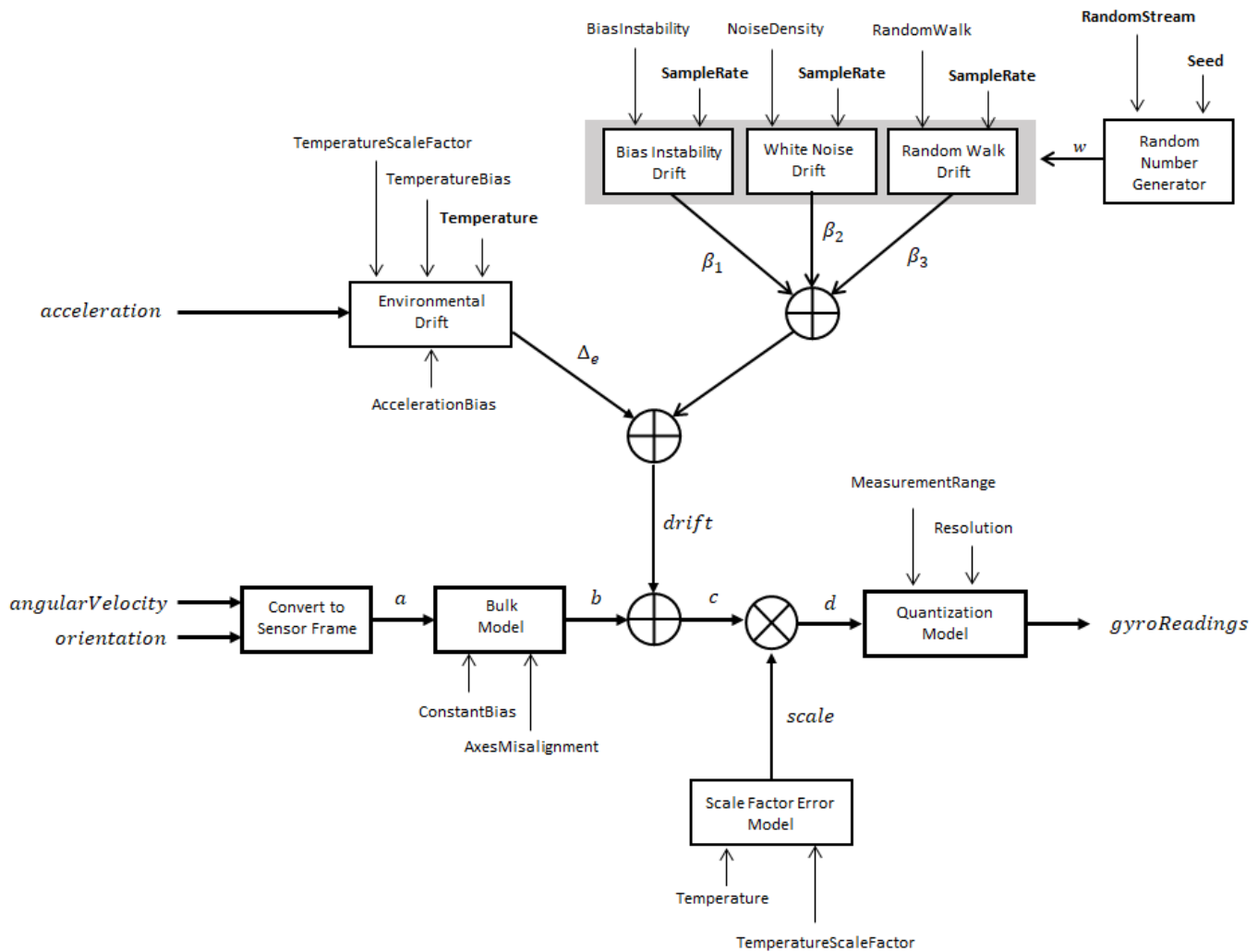
and then setting the resolution:

$$accelReadings = (\text{Resolution}) \left(\text{round} \left(\frac{e}{\text{Resolution}} \right) \right)$$

where `MeasurementRange` is a property of `accelParams`.

Gyroscope

The 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:

$$a = (\textit{orientation})(\textit{angularVelocity})^T$$

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:

$$b = \left(\begin{bmatrix} 1 & \frac{\alpha_2}{100} & \frac{\alpha_3}{100} \\ \frac{\alpha_1}{100} & 1 & \frac{\alpha_3}{100} \\ \frac{\alpha_1}{100} & \frac{\alpha_2}{100} & 1 \end{bmatrix} (a^T) \right)^T + \text{ConstantBias}$$

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:

$$H_1(z) = \frac{1}{1 + \left(\frac{2}{\text{SampleRate}} - 1 \right) z^{-1}}$$

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:

$$scaleFactorError = 1 + \left(\frac{Temperature - 25}{100} \right) (TemperatureScaleFactor)$$

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:

$$e = \begin{cases} \text{MeasurementRange} & \text{if } d > \text{MeasurementRange} \\ -\text{MeasurementRange} & \text{if } -d > \text{MeasurementRange} \\ d & \text{else} \end{cases}$$

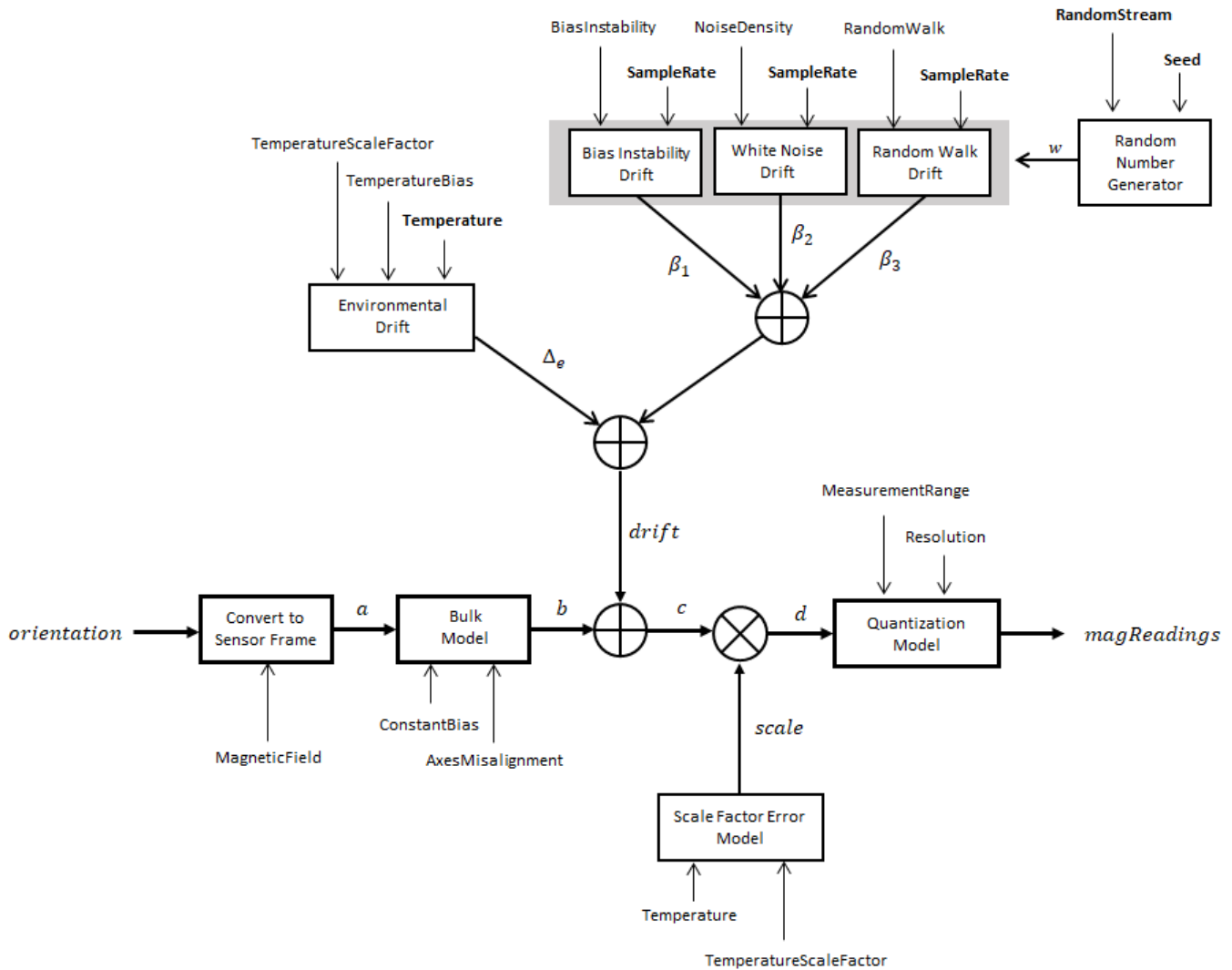
and then setting the resolution:

$$gyroReadings = (\text{Resolution}) \left(\text{round} \left(\frac{e}{\text{Resolution}} \right) \right)$$

where `MeasurementRange` is a property of `gyroParams`.

Magnetometer

The 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:

$$b = \left(\begin{bmatrix} 1 & \frac{\alpha_2}{100} & \frac{\alpha_3}{100} \\ \frac{\alpha_1}{100} & 1 & \frac{\alpha_3}{100} \\ \frac{\alpha_1}{100} & \frac{\alpha_2}{100} & 1 \end{bmatrix} (a^T) \right)^T + \text{ConstantBias}$$

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:

$$H_1(z) = \frac{1}{1 + \left(\frac{2}{\text{SampleRate}} - 1 \right) z^{-1}}$$

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:

$$scaleFactorError = 1 + \left(\frac{Temperature - 25}{100} \right) (TemperatureScaleFactor)$$

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™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

Classes

`accelparams` | `gyroparams` | `magparams`

System Objects

`gpsSensor`

Introduced in R2018b

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.

```
{
  "GenericLowCost9Axis":
  {
    "Accelerometer":
    {
      "MeasurementRange": 19.6133,
      "Resolution": 0.0023928,
      "ConstantBias": [0.19,0.19,0.19],
      "AxesMisalignment": [0,0,0],
      "NoiseDensity": [0.0012356,0.0012356,0.0012356],
      "BiasInstability": [0,0,0],
      "RandomWalk": [0,0,0],
      "TemperatureBias": [0,0,0],
      "TemperatureScaleFactor": [0,0,0]
    },
    "Gyroscope":
    { [collapsed] },
    "Magnetometer":
    { [collapsed] }
  },
  "GenericLowCost6Axis":
  { [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? \(MATLAB\)](#).

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/s²)

`[0 0 0]` (default) | three-element row vector

Acceleration state in m/s², 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

`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,bodyAngularVelocity)
```

```
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: [1×1 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];

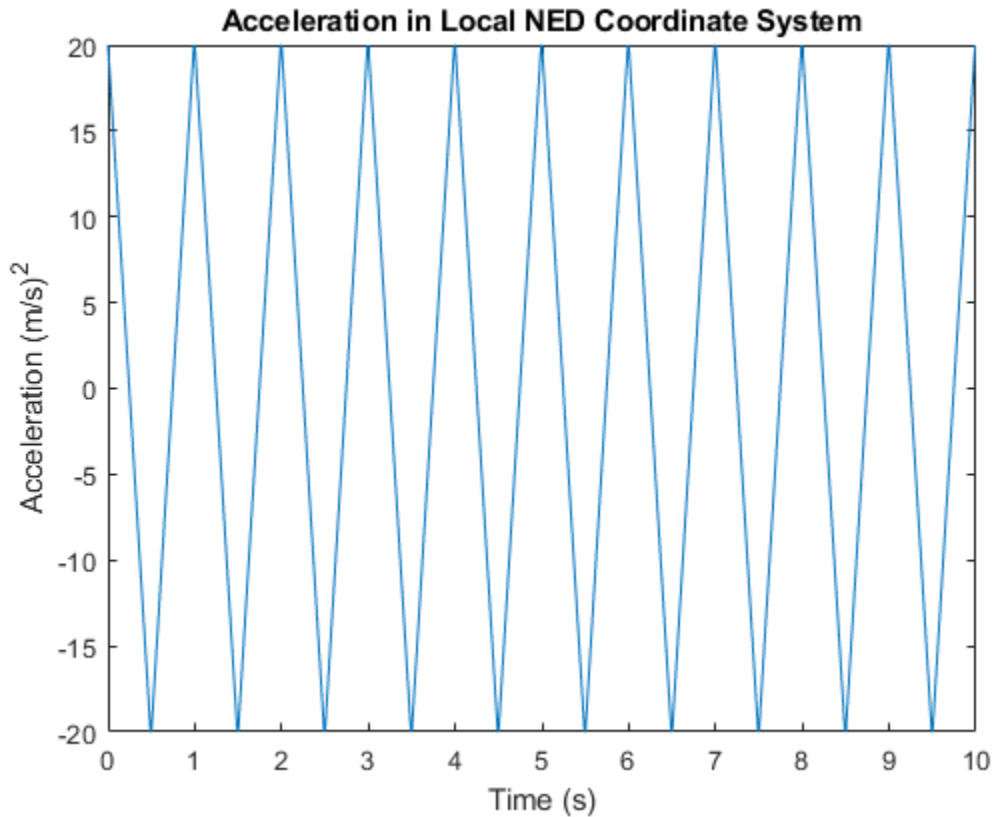
angVelNED = zeros(numSamples,3);
angVelNED(:,2) = 2*pi;

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

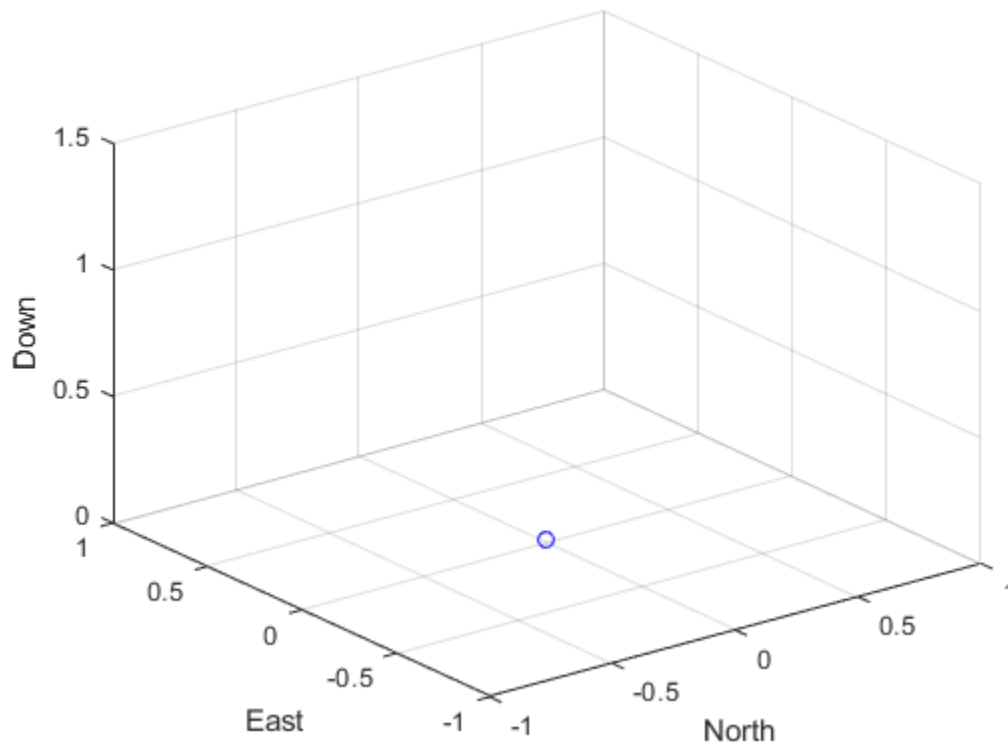
q = ones(numSamples,1,'quaternion');
for ii = 1:numSamples
    accBody = rotateframe(traj.Orientation,accNED(ii,:));
    angVelBody = rotateframe(traj.Orientation,angVelNED(ii,:));

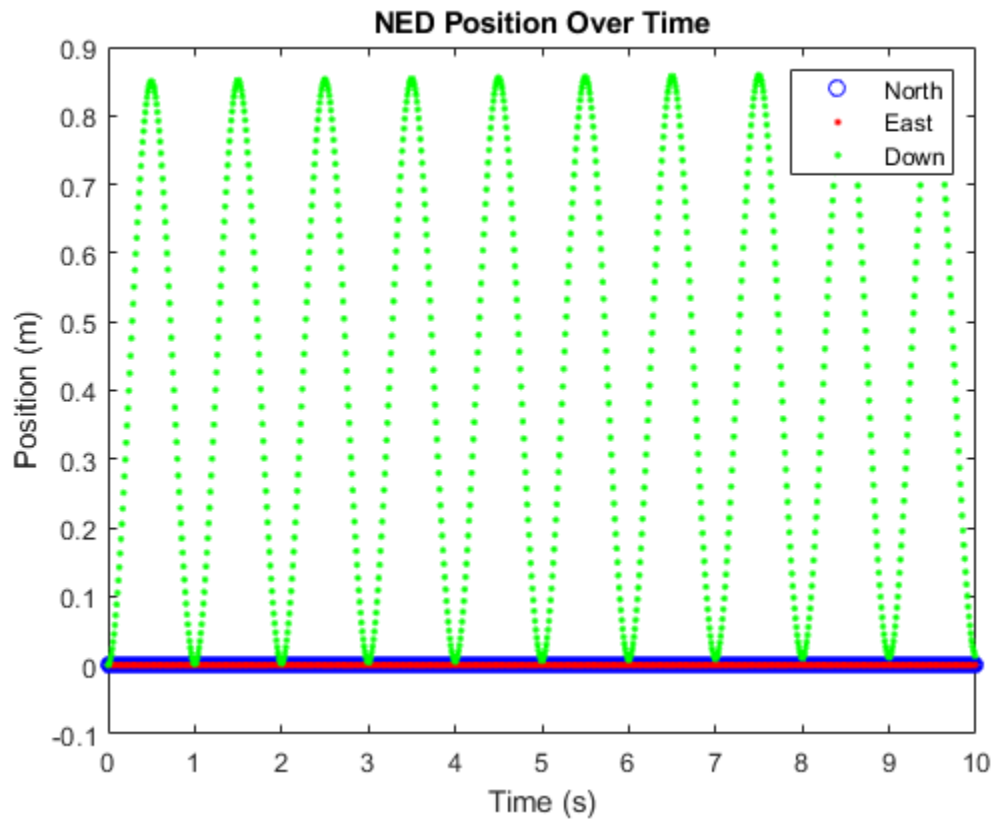
    [pos(ii,:),q(ii),vel,ac] = traj(accBody,angVelBody);

    set(plotHandle,'XData',pos(ii,1),'YData',pos(ii,2),'ZData',pos(ii,3))

    pause(1/fs)
```

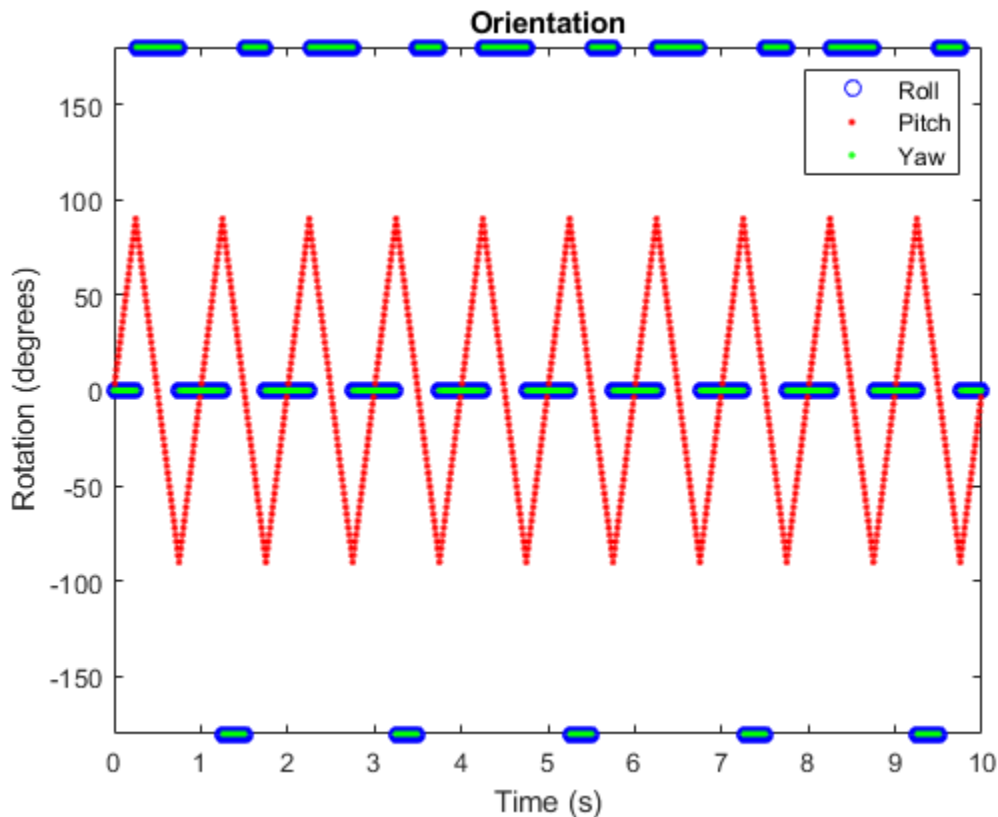
```
end  
  
figure(3)  
plot(timeVector,pos(:,1),'bo',...  
      timeVector,pos(:,2),'r.',...  
      timeVector,pos(:,3),'g.')  
xlabel('Time (s)')  
ylabel('Position (m)')  
title('NED Position Over Time')  
legend('North','East','Down')
```





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('Roll', 'Pitch', 'Yaw')
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);
```

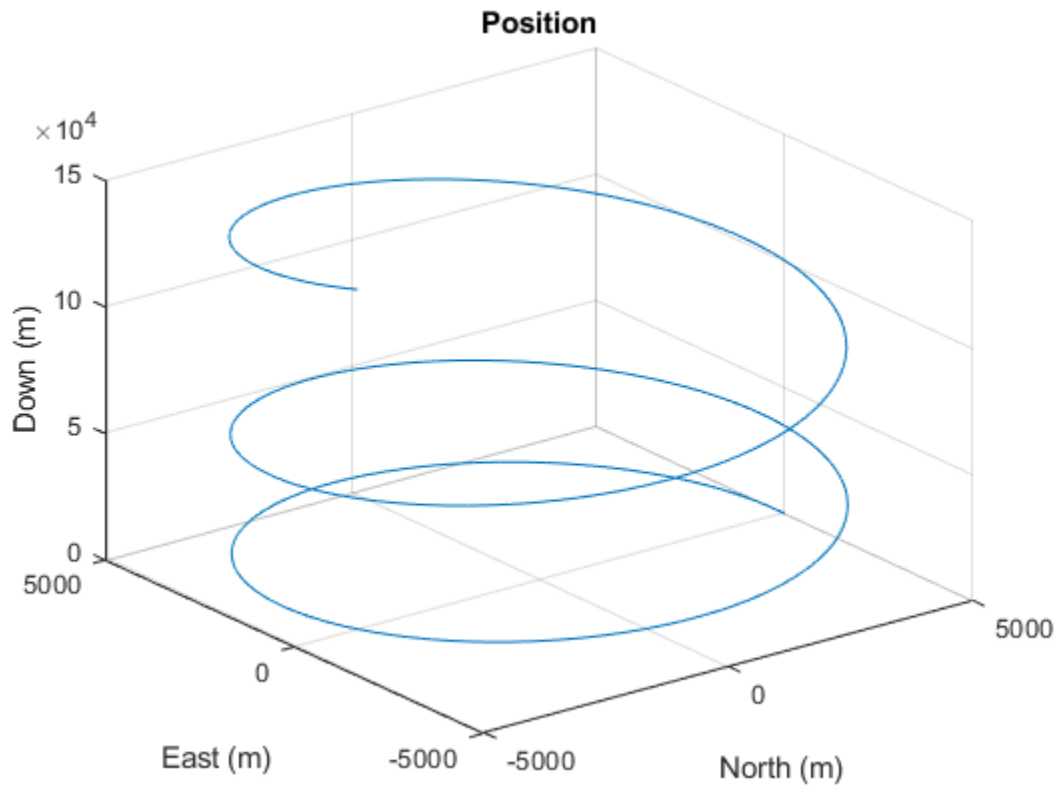
Call `trajectory` with the specified acceleration and angular velocity in the body coordinate system. Plot the position, orientation, and speed over time.

```
[position, orientation, velocity] = trajectory(accBody,angVelBody);
```

```
eulerAngles = eulerd(orientation, 'ZYX', 'frame');  
speed = sqrt(sum(velocity.^2,2));
```

```
timeVector = (0:(N-1))/fs;
```

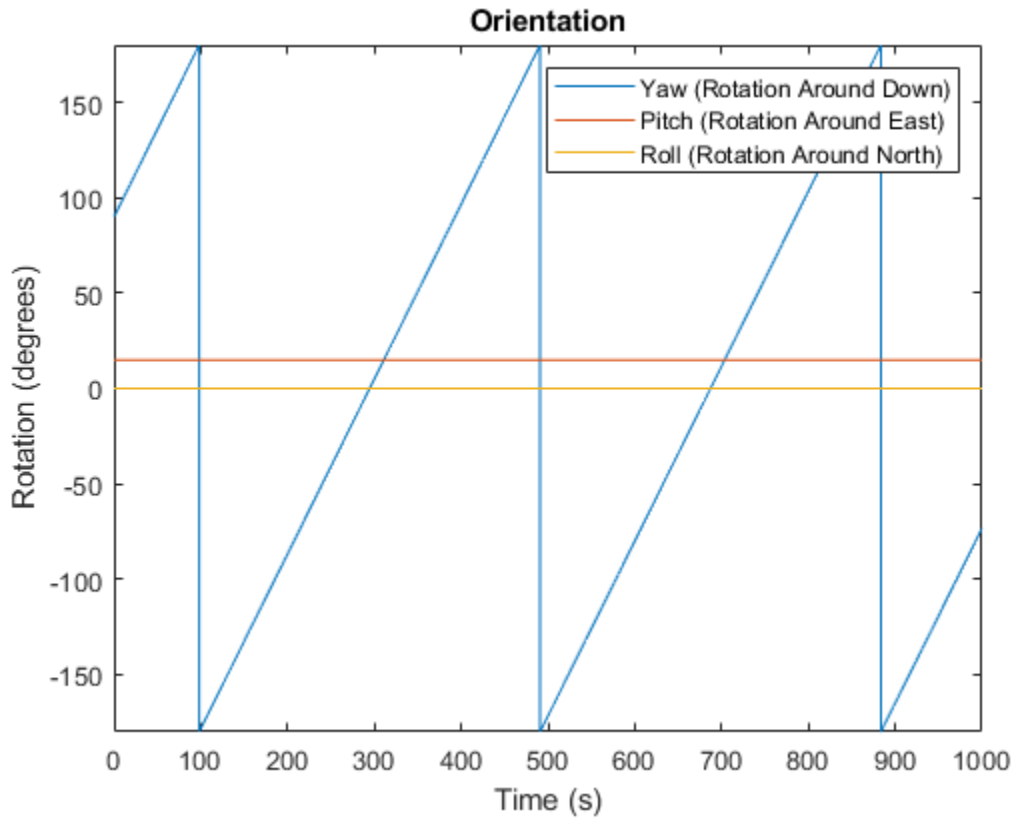
```
figure(1)  
plot3(position(:,1),position(:,2),position(:,3))  
xlabel('North (m)')  
ylabel('East (m)')  
zlabel('Down (m)')  
title('Position')  
grid on
```



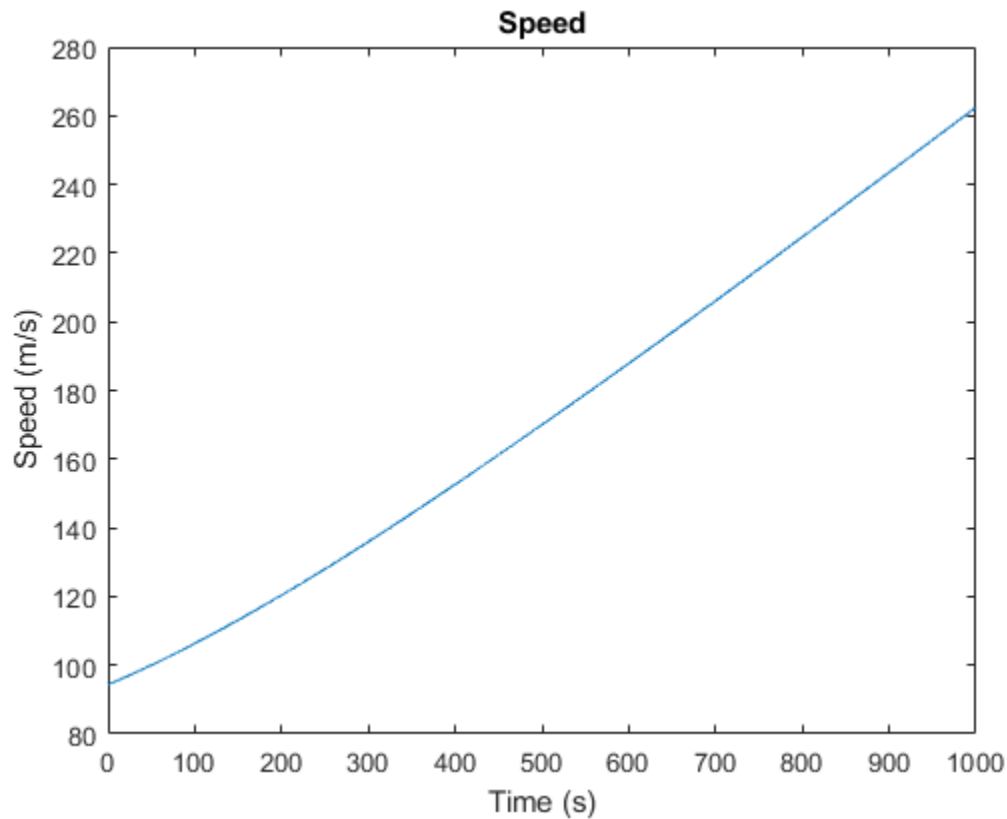
```

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.

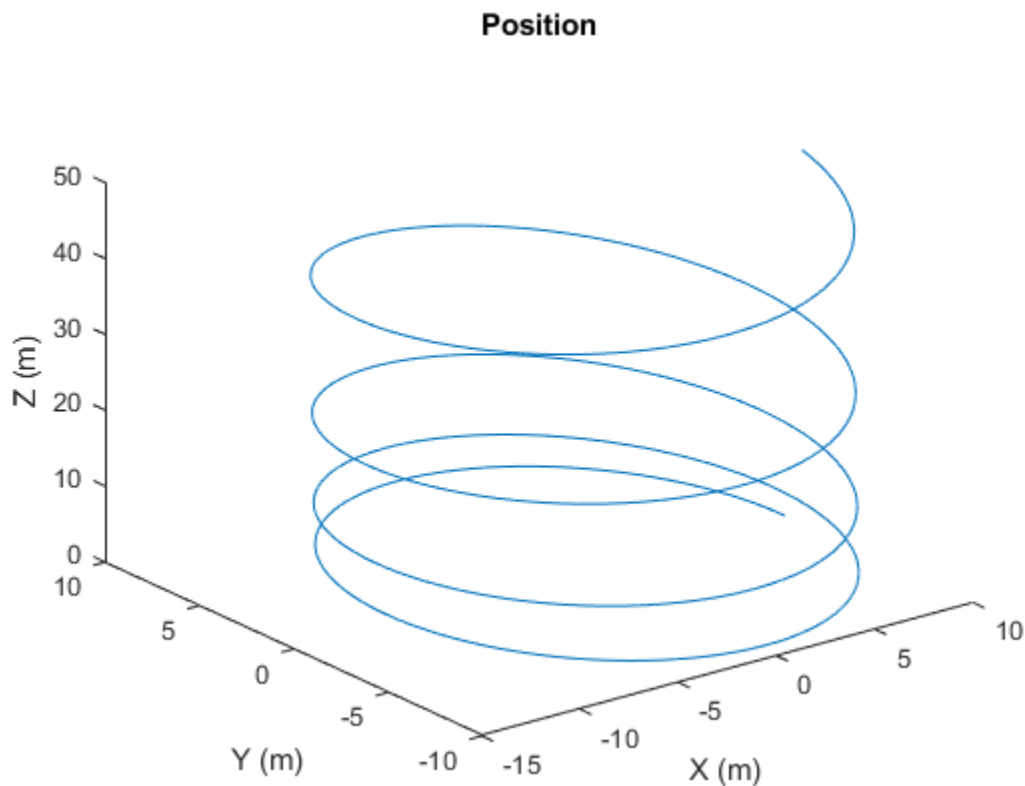
```
traj = kinematicTrajectory('SampleRate',Fs, ...
    'Position',initPos, ...
    'Velocity',initVel, ...
    'Orientation',initOrient, ...
    'AccelerationSource','Property', ...
    'Acceleration',accBody, ...
```

```
'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™.

Usage notes and limitations:

“System Objects in MATLAB Code Generation” (MATLAB Coder)

See Also

waypointTrajectory

Introduced in R2018b

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

vector

Range readings from lidar, specified as a vector. 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

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

[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
plot	Display laser or lidar scan readings
removeInvalidData	Remove invalid range and angle data
transformScan	Transform laser scan based on relative pose

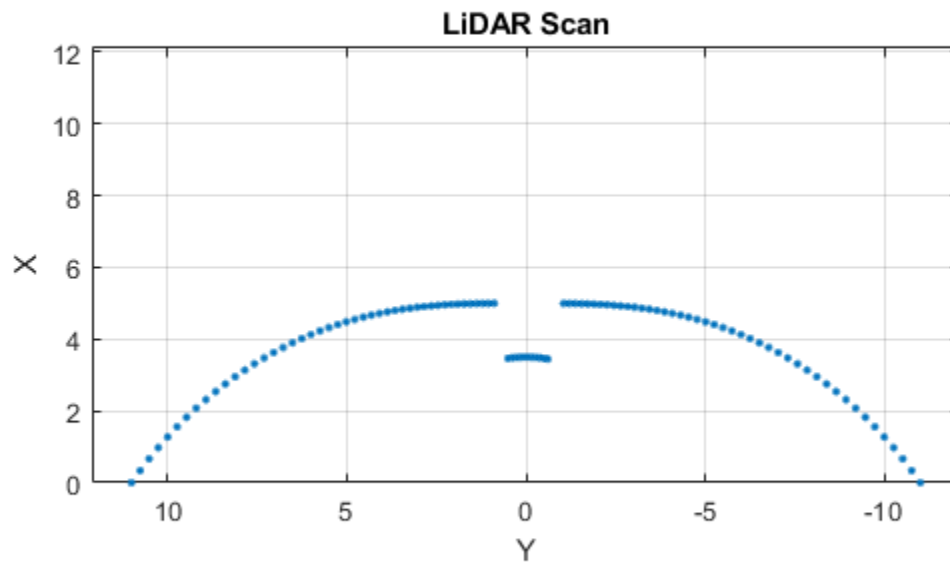
Examples**Plot Lidar Scan and Remove Invalid Points**

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));
```

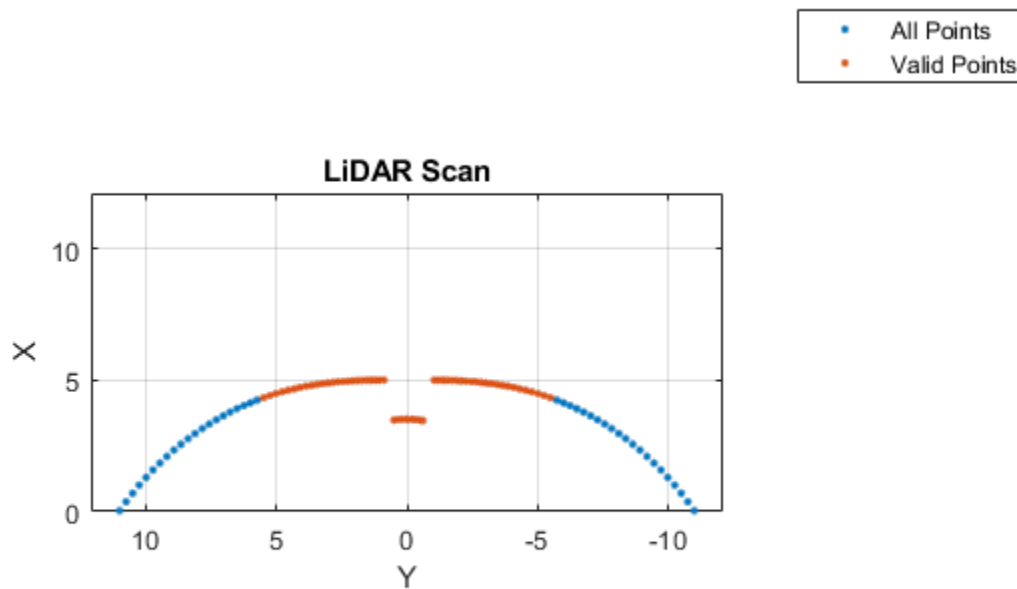
Create a lidar scan by specifying the ranges and angles. Plot all points of the lidar scan.

```
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`. 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` 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.

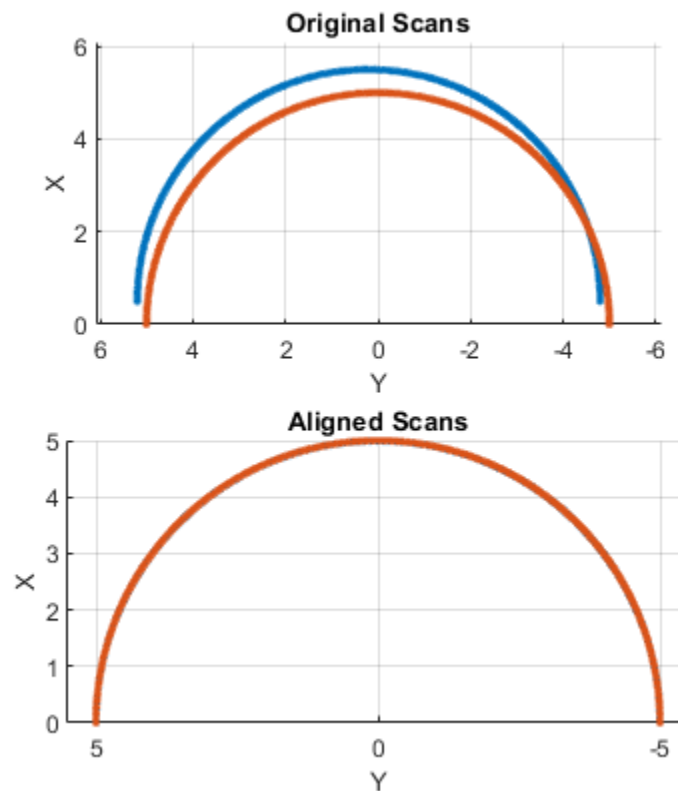
```
pose = matchScans(currScan,refScan);
```

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

[controllerVFH](#) | [matchScans](#) | [monteCarloLocalization](#) | [transformScan](#)

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

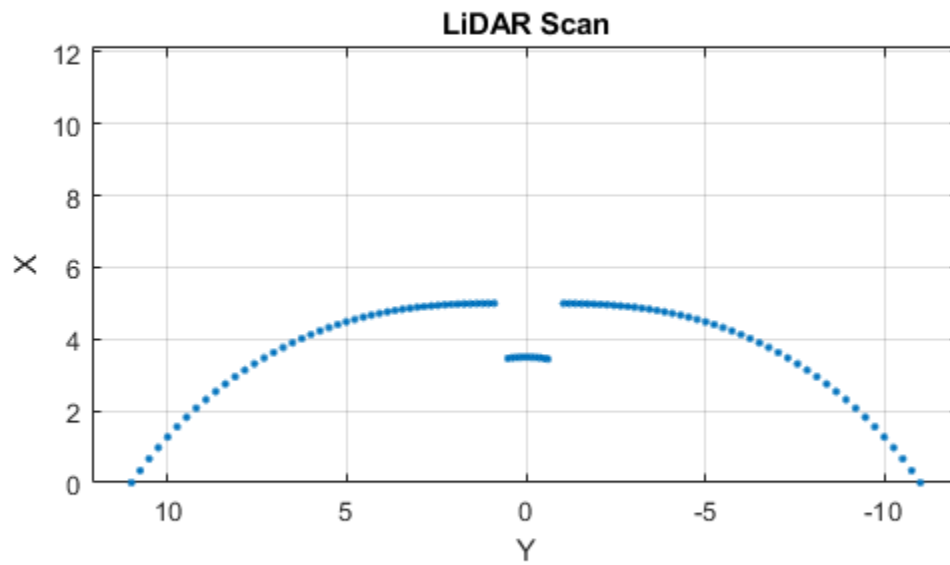
Plot Lidar Scan and Remove Invalid Points

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));
```

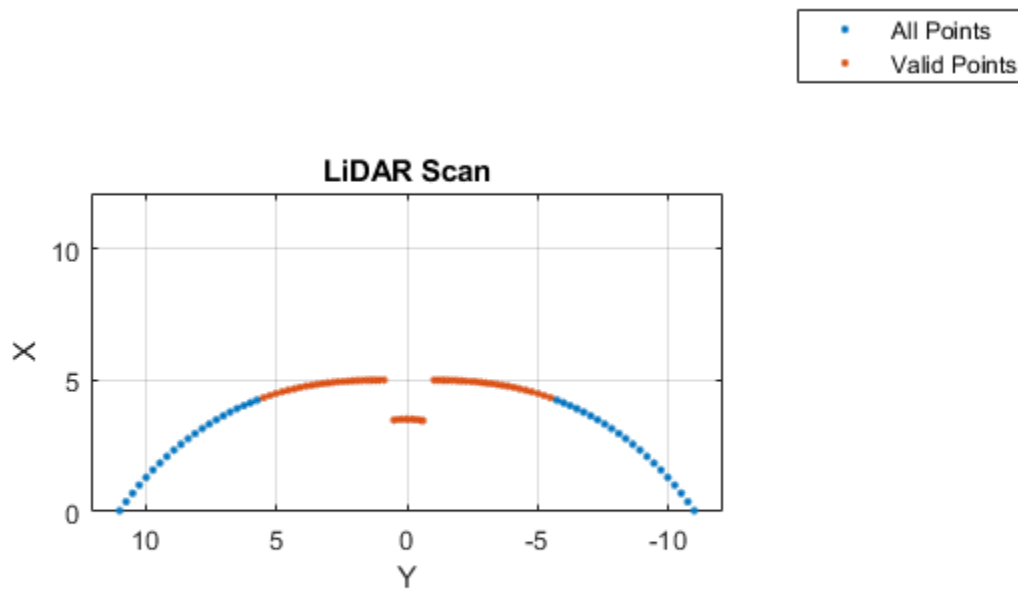
Create a lidar scan by specifying the ranges and angles. Plot all points of the lidar scan.

```
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

`controllerVFH` | `matchScans` | `monteCarloLocalization` | `transformScan`

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`.

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

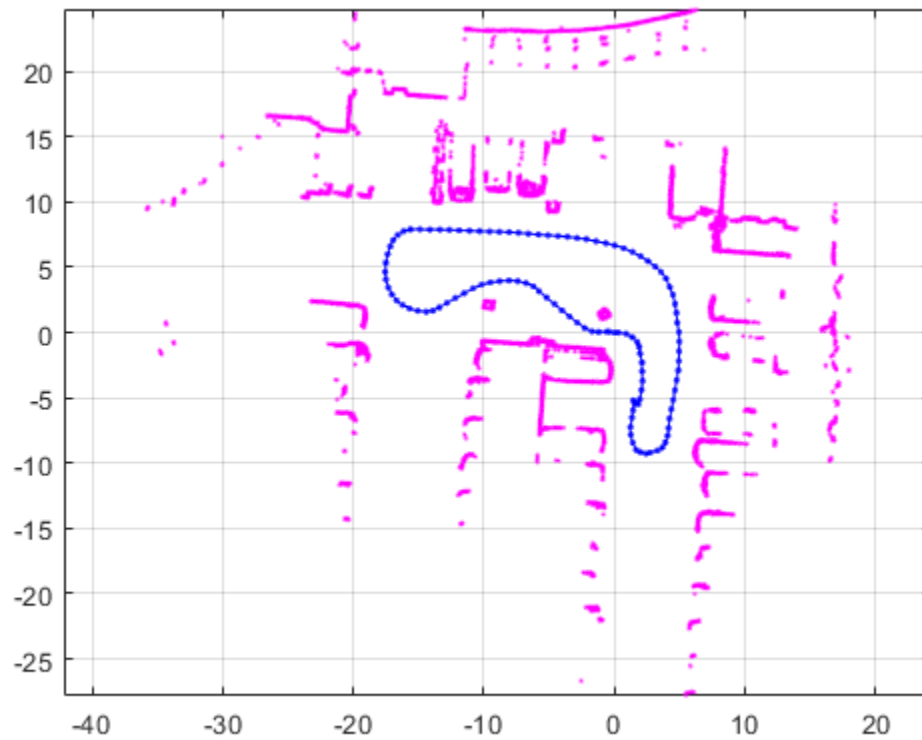
slamObj = lidarSLAM(resolution,maxRange);
slamObj.LoopClosureThreshold = 360;
slamObj.LoopClosureSearchRadius = 8;
```

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});

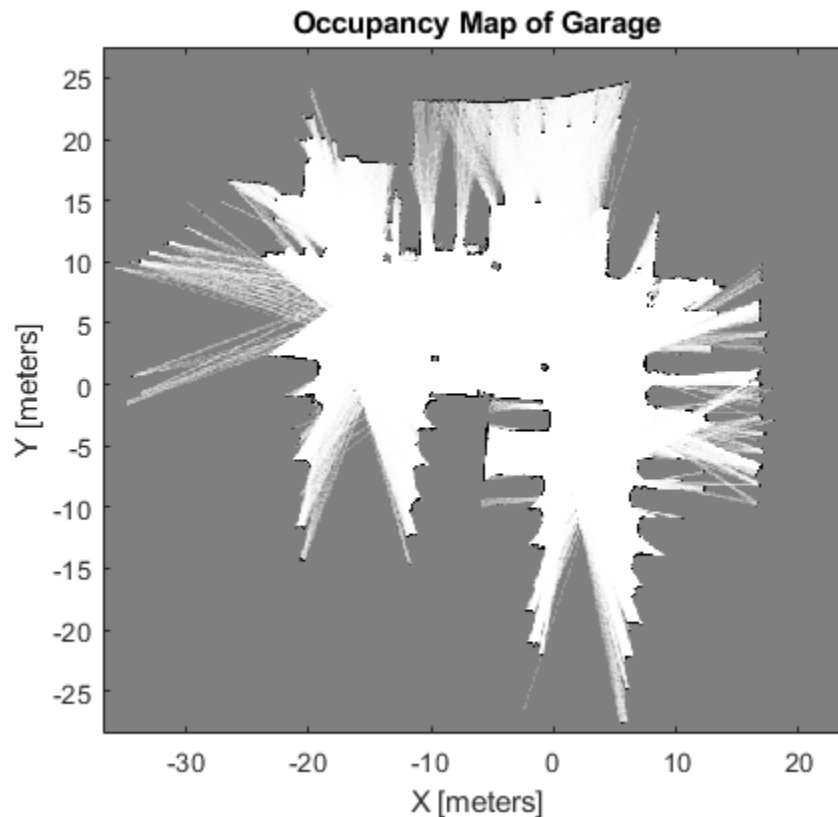
    if rem(i,10) == 0
        show(slamObj);
    end
end
```

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

`optimizePoseGraph` | `poseGraph`

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.

`[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.

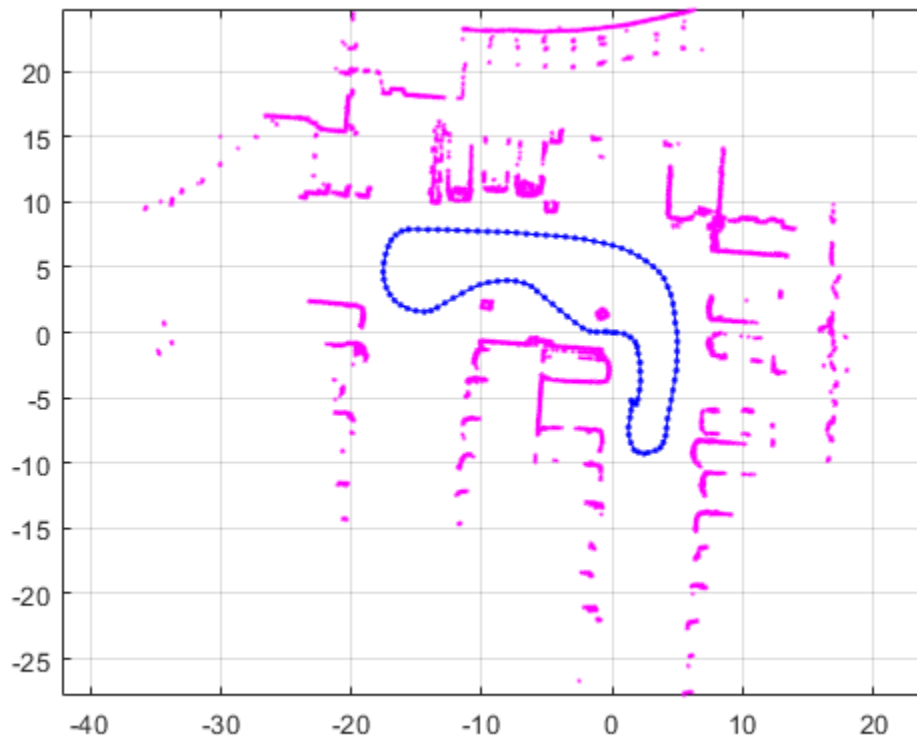
```
maxRange = 19.2; % meters
resolution = 10; % cells per meter
```

```
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
```

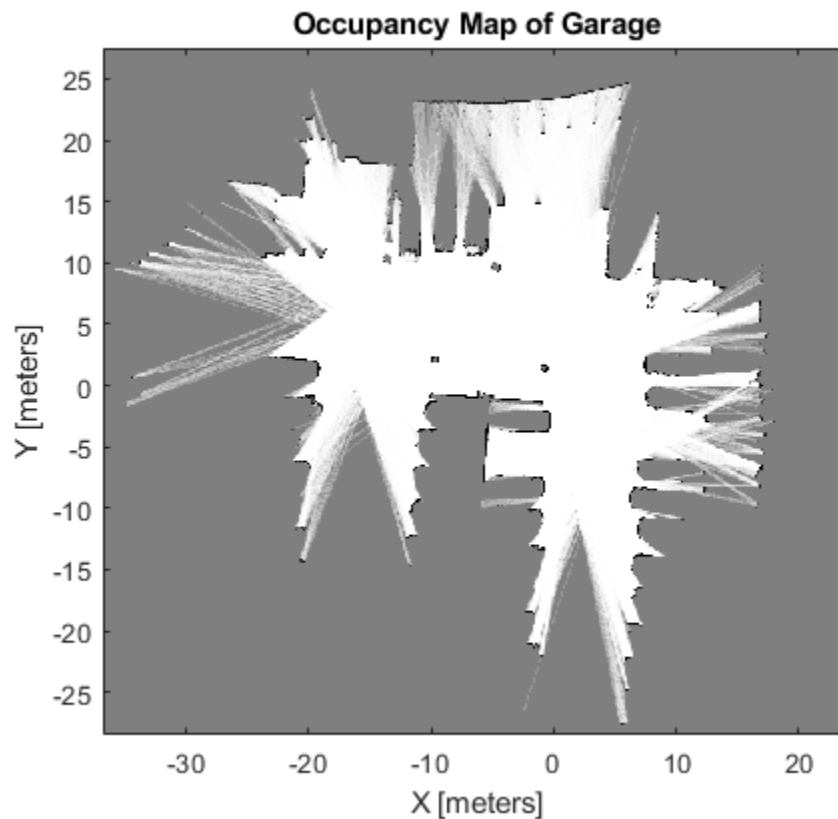


View Occupancy Map

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

`optimizePoseGraph` | `poseGraph`

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

`optimizePoseGraph` | `poseGraph`

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

`optimizePoseGraph` | `poseGraph`

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.

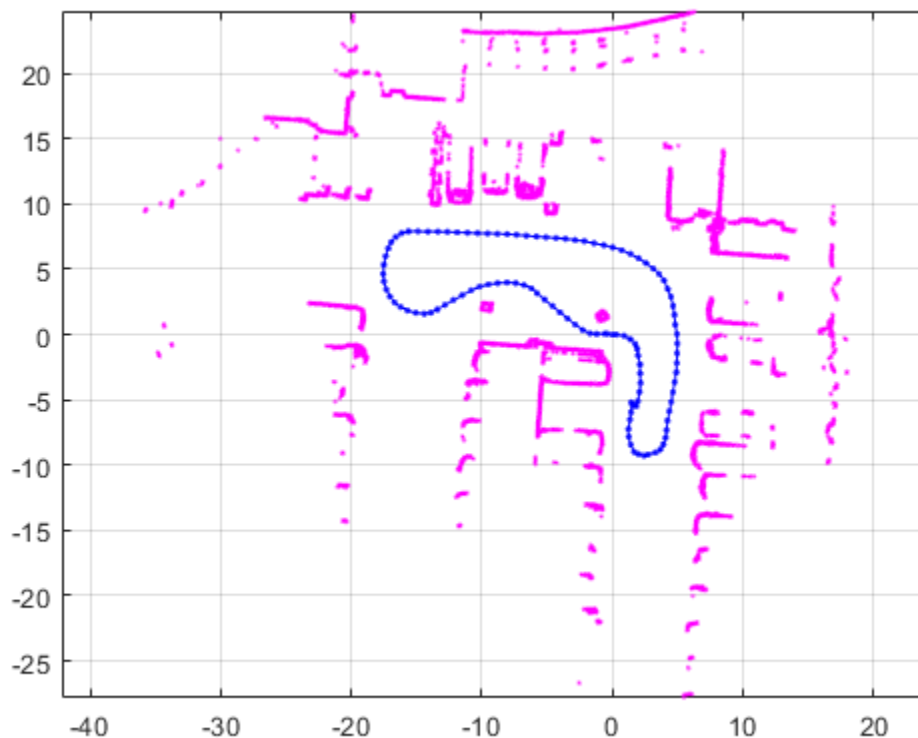
```
maxRange = 19.2; % meters
resolution = 10; % cells per meter

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.

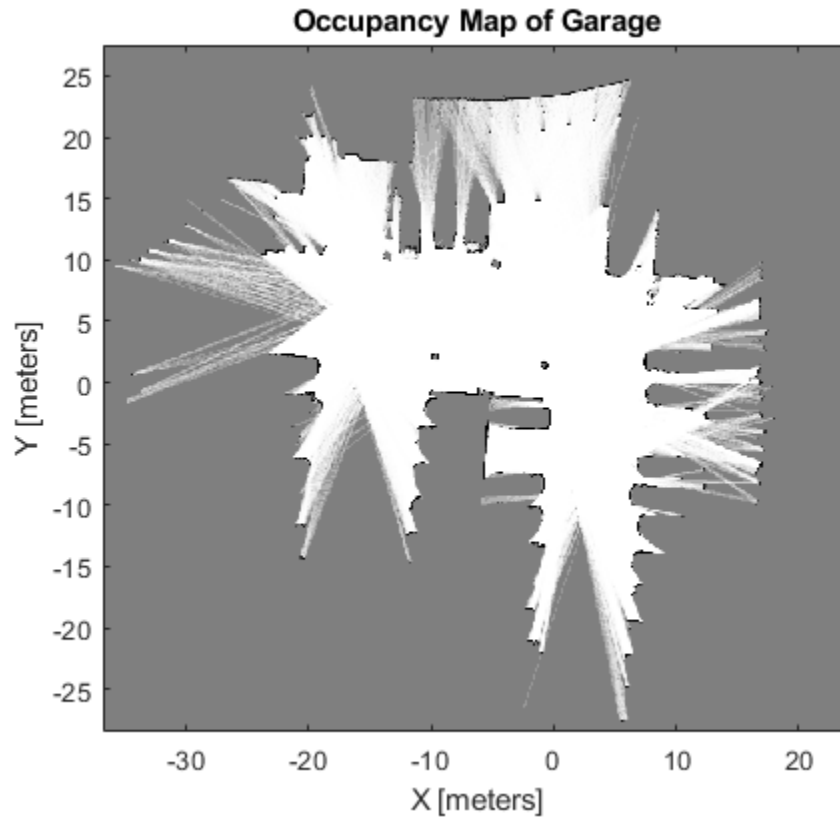
```
for i = 1:numel(scans)
    addScan(slamObj,scans{i});
    if rem(i,10) == 0
        show(slamObj);
    end
end
```



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

`optimizePoseGraph` | `poseGraph`

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.

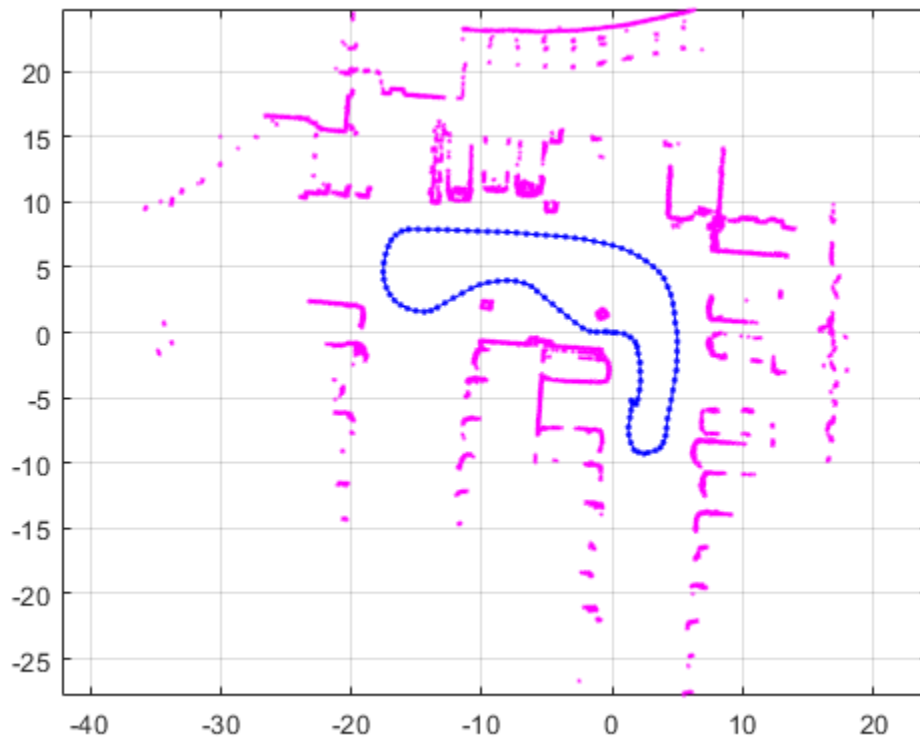
```
maxRange = 19.2; % meters
resolution = 10; % cells per meter

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.

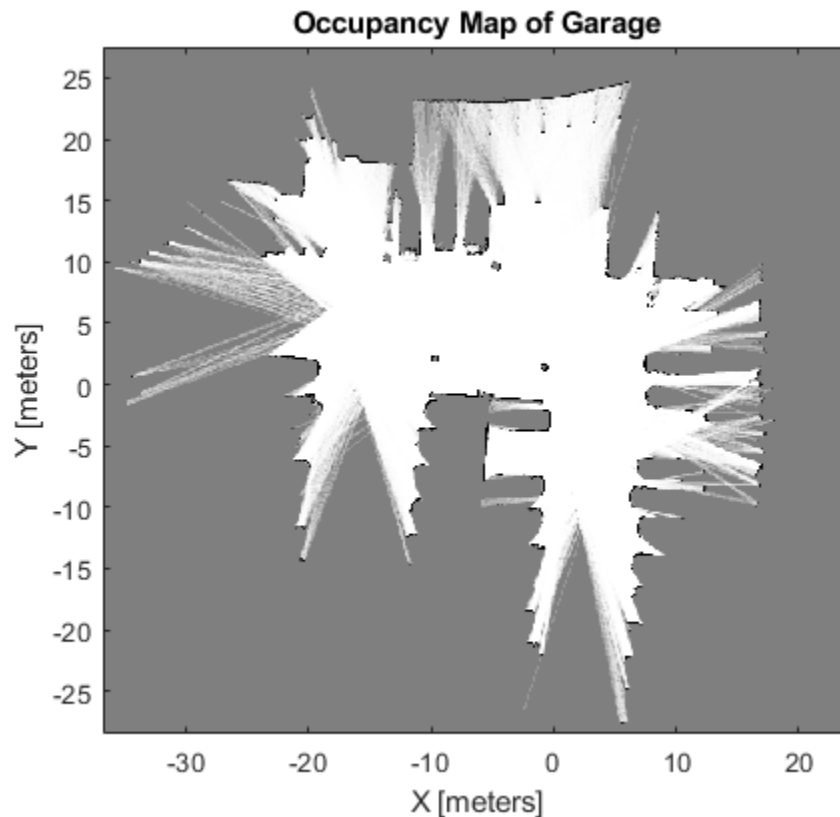
```
for i = 1:numel(scans)
    addScan(slamObj,scans{i});
    if rem(i,10) == 0
        show(slamObj);
    end
end
```



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

optimizePoseGraph | poseGraph

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

`likelihoodFieldSensor` 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(...);
release(mcl)
mcl.SensorModel.PropName = value;
```

See Also

`monteCarloLocalization` | `odometryMotionModel`

Topics

“Localize TurtleBot Using Monte Carlo Localization”

“Monte Carlo Localization Algorithm”

Introduced in R2019b

magparams class

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`.

Construction

`params = magparams` 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 ($\mu\text{T}/\text{LSB}$)

`0` (default) | real nonnegative scalar

Resolution of sensor measurements in $\mu\text{T}/\text{LSB}$, specified as a real nonnegative scalar

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 (%)

`[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 %, 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`

NoiseDensity — Power spectral density of sensor noise ($\mu\text{T}/\sqrt{\text{Hz}}$)

[0 0 0] (default) | real scalar | real 3-element row vector

Power spectral density of sensor noise in $\mu\text{T}/\sqrt{\text{Hz}}$, 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 ($\mu\text{T}/\sqrt{\text{Hz}}$)

[0 0 0] (default) | real scalar | real 3-element row vector

Integrated white noise of sensor in ($\mu\text{T}/\sqrt{\text{Hz}}$), 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 ($\mu\text{T}/^\circ\text{C}$)

[0 0 0] (default) | real scalar | real 3-element row vector

Sensor bias from temperature in ($\mu\text{T}/^\circ\text{C}$), 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 ($\%/^\circ\text{C}$)

[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 ($\%/^\circ\text{C}$), 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

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 $\mu\text{T}/\text{LSB}$. The constant offset bias is 1 μT . The sensor has a power spectral density of $\left(\frac{[0.6 \ 0.6 \ 0.9]}{\sqrt{100}}\right) \mu\text{T}/\sqrt{\text{Hz}}$. The bias from temperature is [0.8 0.8 2.4] $\mu\text{T}/^\circ\text{C}$. The scale factor error from temperature is 0.1 $\%/^\circ\text{C}$.

```
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);
```

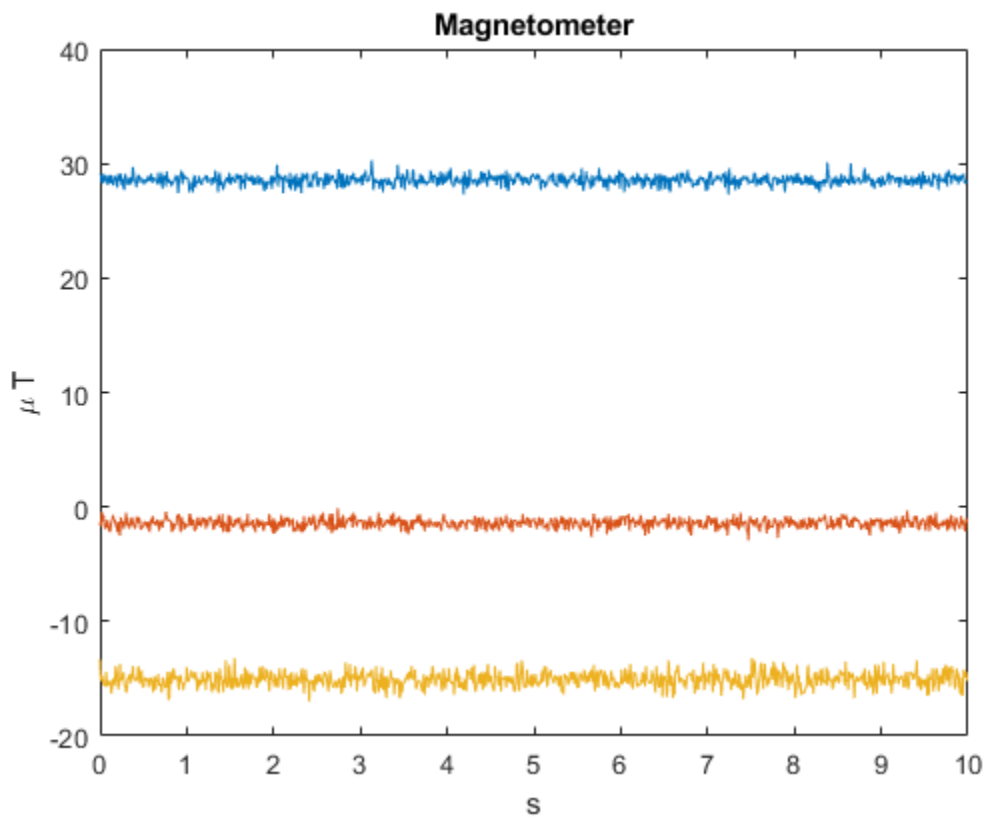
Generate magnetometer data from the imuSensor object.

```
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

Classes

accelparams | gyroparams

System Objects

imuSensor

Introduced in R2018b

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/s²)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/s²)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 μT², 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
- `residual` Residuals and residual covariances from direct state measurements
- `fusegps` Correct states using GPS data
- `residualgps` Residuals and residual covariance from GPS measurements
- `fusemag` Correct states using magnetometer data
- `residualmag` Residuals and residual covariance from magnetometer measurements
- `pose` Current orientation and position estimate
- `predict` Update states using accelerometer and gyroscope data

reset Reset internal states
stateinfo Display state vector information

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:

$$x = \begin{bmatrix} q_0 \\ q_1 \\ q_2 \\ q_3 \\ position_N \\ position_E \\ position_D \\ v_N \\ v_E \\ v_D \\ \Delta\theta_{bias_X} \\ \Delta\theta_{bias_Y} \\ \Delta\theta_{bias_Z} \\ \Delta v_{bias_X} \\ \Delta v_{bias_Y} \\ \Delta v_{bias_Z} \\ geomagneticFieldVector_N \\ geomagneticFieldVector_E \\ geomagneticFieldVector_D \\ mag_{bias_X} \\ mag_{bias_Y} \\ mag_{bias_Z} \end{bmatrix}$$

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.
- $mag_{bias_X}, mag_{bias_Y}, mag_{bias_Z}$ -- 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:

$x_{k|k-1} =$

$$\begin{aligned}
 & q_0 - q_1 \left(\frac{\Delta\theta_X - \Delta\theta_{biasX}}{2} \right) - q_2 \left(\frac{\Delta\theta_Y - \Delta\theta_{biasY}}{2} \right) - q_3 \left(\frac{\Delta\theta_Z - \Delta\theta_{biasZ}}{2} \right) \\
 & q_1 + q_0 \left(\frac{\Delta\theta_X - \Delta\theta_{biasX}}{2} \right) - q_3 \left(\frac{\Delta\theta_Y - \Delta\theta_{biasY}}{2} \right) + q_2 \left(\frac{\Delta\theta_Z - \Delta\theta_{biasZ}}{2} \right) \\
 & q_2 + q_3 \left(\frac{\Delta\theta_X - \Delta\theta_{biasX}}{2} \right) + q_0 \left(\frac{\Delta\theta_Y - \Delta\theta_{biasY}}{2} \right) - q_1 \left(\frac{\Delta\theta_Z - \Delta\theta_{biasZ}}{2} \right) \\
 & q_3 - q_2 \left(\frac{\Delta\theta_X - \Delta\theta_{biasX}}{2} \right) + q_1 \left(\frac{\Delta\theta_Y - \Delta\theta_{biasY}}{2} \right) + q_0 \left(\frac{\Delta\theta_Z - \Delta\theta_{biasZ}}{2} \right)
 \end{aligned}$$

$position_N + (\Delta t)(v_N)$

$position_E + (\Delta t)(v_E)$

$position_D + (\Delta t)(v_D)$

$$\begin{aligned}
 & v_N + (\Delta t)(g_N) + (\Delta v_X - \Delta v_{biasX})(q_0^2 + q_1^2 - q_2^2 - q_3^2) - 2(\Delta v_Y - \Delta v_{biasY})(q_0 q_3 - q_1 q_2) + 2(\Delta v_Z - \Delta v_{biasZ})(q_0 q_2 + \\
 & v_E + (\Delta t)(g_E) + (\Delta v_Y - \Delta v_{biasY})(q_0^2 - q_1^2 + q_2^2 - q_3^2) + 2(\Delta v_X - \Delta v_{biasX})(q_0 q_3 + q_1 q_2) - 2(\Delta v_Z - \Delta v_{biasZ})(q_0 q_1 - \\
 & v_D + (\Delta t)(g_D) + (\Delta v_Z - \Delta v_{biasZ})(q_0^2 - q_1^2 - q_2^2 + q_3^2) - 2(\Delta v_X - \Delta v_{biasX})(q_0 q_2 - q_1 q_3) + 2(\Delta v_Y - \Delta v_{biasY})(q_0 q_1 +
 \end{aligned}$$

2-342

$\Delta\theta_{biasX}$

$\Delta\theta_{biasY}$

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

[insfilterAsync](#) | [insfilterErrorState](#) | [insfilterNonholonomic](#)

Introduced in R2018b

correct

Correct states using direct state measurements

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 measurementscalar | 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` | `insfilterMARG`**Introduced in R2018b**

fusegps

Correct states using GPS data

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 (m²)

3-by-3 matrix

Position measurement covariance of GPS receiver in m², 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/s²)

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

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 $(\mu\text{T})^2$.

Extended Capabilities

C/C++ Code Generation

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

See Also

`insfilter` | `insfilterMARG`

Introduced in R2018b

pose

Current orientation and position estimate

Syntax

```
[position,orientation ] = pose(FUSE)
[position,orientation ] = pose(FUSE,format)
```

Description

[position,orientation] = pose(FUSE) returns the current estimate of the pose.

[position,orientation] = 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 in the local NED coordinate system (m)

3-element row vector

Position estimate in the local NED coordinate system in meters, returned as a 3-element row vector.

Data Types: single | double

orientation — Orientation estimate in the local NED coordinate system

quaternion (default) | 3-by-3 rotation matrix

Orientation estimate in the local NED coordinate system, specified as a scalar quaternion or 3-by-3 rotation matrix. The quaternion or rotation matrix represents a frame rotation from the local NED reference frame to the body reference frame.

Data Types: single | double | quaternion

Extended Capabilities

C/C++ Code Generation

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

See Also

`insfilter` | `insfilterMARG`

Introduced in R2018b

predict

Update states using accelerometer and gyroscope data

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/s²)

3-element row vector

Accelerometer readings in m/s², 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` | `insfilterMARG`

Introduced in R2018b

reset

Reset internal states

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

`insfilter` | `insfilterMARG`

Introduced in R2018b

residual

Residuals and residual covariances from direct state measurements

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

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 (m²)

3-by-3 matrix

Position measurement covariance of GPS receiver in m², 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/s²)

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

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 $(\mu\text{T})^2$.

See Also

`insfilter` | `insfilterMARG`

Introduced in R2020a

stateinfo

Display state vector information

Syntax

```
stateinfo(FUSE)
```

Description

`stateinfo(FUSE)` displays the meaning of each index of the State property and the associated units.

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

`insfilter` | `insfilterMARG`

Introduced in R2018b

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? \(MATLAB\)](#).

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 likelihoodFieldSensor 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(_);
release(mcl)
mcl.SensorModel.PropName = value;
```

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(_);
release(mcl)
mcl.MotionModel.PropName = value;
```

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, Sebastian, 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` | `likelihoodFieldSensor` | `odometryMotionModel`

Topics

"Localize TurtleBot Using Monte Carlo Localization"

"Monte Carlo Localization Algorithm"

Class Attributes (MATLAB)

Property Attributes (MATLAB)

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

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: [3x2 double]  
    NumStateVariables: 3
```

```
Dubins Vehicle Properties  
    MinTurningRadius: 1
```

```
pathobj = navPath(dubinsSpace)
```

```
pathobj =  
navPath with properties:  
  
    StateSpace: [1x1 stateSpaceDubins]  
    States: [0x3 double]  
    NumStates: 0
```

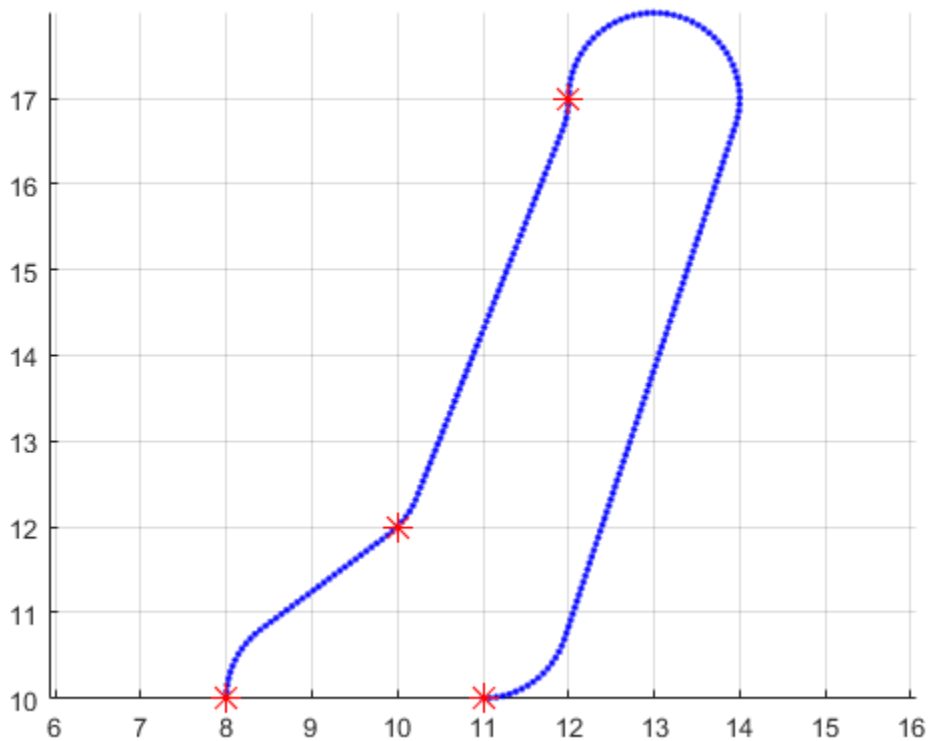
```
waypoints = [...  
    8 10 pi/2;  
    10 12 pi/4;  
    12 17 pi/2;  
    11 10 -pi];  
append(pathobj, waypoints);
```

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

[nav.StateSpace](#) | [pathmetrics](#) | [stateSpaceDubins](#) | [stateSpaceReedsShepp](#) | [stateSpaceSE2](#)

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.

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.

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.

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

nav.StateSpace class

Package: nav

Create state space for path planning

Description

`nav.StateSpace` 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 our implementations of the state space class, see “State Spaces”.

The `nav.StateSpace` class is a handle class.

Class Attributes

Abstract true

For information on class attributes, see “Class Attributes” (MATLAB).

Creation

Syntax

```
ssObj = nav.StateSpace(Name, NumStateVariables, Bounds)
```

Description

`ssObj = nav.StateSpace(Name, NumStateVariables, Bounds)` creates a state space object with a given name and state bounds. `NumStateVariables` specifies the number of state variables. 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 state space, specified as a positive numeric scalar. This property is the dimension of the state space.

Example: 3

Attributes:

SetAccess - immutable

StateBounds — Min and max bounds of state variables[min max] | n -by-2 matrix

Min and max 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(numStateVariables);
        obj.UniformDistribution = matlabshared.tracking.internal.UniformDistribution(numStateVariables);
        % 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, "enfo
    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,

    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, "dist");
    nav.internal.validation.validateStateMatrix(state2, size(state1,1), obj.NumStateVariables, "dist");

    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](#) | [stateSpaceDubins](#) | [stateSpaceReedsShepp](#) | [stateSpaceSE2](#)

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`.

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`. 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(numStateVariables);
        obj.UniformDistribution = matlabshared.tracking.internal.UniformDistribution(numStateVariables);
        % 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, "enfo
    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,
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, "dist");
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](#) | [stateSpaceDubins](#) | [stateSpaceReedsShepp](#) | [stateSpaceSE2](#)

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(numStateVariables);
        obj.UniformDistribution = matlabshared.tracking.internal.UniformDistribution(numStateVariables);
        % 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, "enfo
    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,

    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, "dist");
    nav.internal.validation.validateStateMatrix(state2, size(state1,1), obj.NumStateVariables, "dist");

    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](#) | [stateSpaceDubins](#) | [stateSpaceReedsShepp](#) | [stateSpaceSE2](#)

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(numStateVariables);
        obj.UniformDistribution = matlabshared.tracking.internal.UniformDistribution(numStateVariables);
        % 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, "enfo
    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,
    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, "dist");
    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](#) | [stateSpaceDubins](#) | [stateSpaceReedsShepp](#) | [stateSpaceSE2](#)

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

Mean state position, 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`.

stdDev — Standard deviation around mean state

n-element vector

Standard deviation around mean state, specified as an *n*-element vector, 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(numStateV
obj.UniformDistribution = matlabshared.tracking.internal.UniformDistribution(numState
% 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, "enfo
    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, varargin);

    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, "dist");
    nav.internal.validation.validateStateMatrix(state2, size(state1,1), obj.NumStateVariables, "dist");

    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` | `stateSpaceDubins` | `stateSpaceReedsShepp` | `stateSpaceSE2`

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, nearState, distance)
states = sampleUniform(ssObj, nearState, 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, nearState, distance)` samples near a given state within a certain distance.

`states = sampleUniform(ssObj, nearState, distance, numSamples)` samples multiple states near a given 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`.

nearState — Mean state position

n-element vector

Mean state position, 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(numStateVariables);
obj.UniformDistribution = matlabshared.tracking.internal.UniformDistribution(numStateVariables);
% 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, "enforce");
    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;
end

```

```

        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, varargin);

    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, "dist");
    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` | `stateSpaceDubins` | `stateSpaceReedsShepp` | `stateSpaceSE2`

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 our implementations of the state validator class, see “State Validation”.

The `nav.StateValidator` class is a handle class.

Class Attributes

Abstract true

For information on class attributes, see “Class Attributes” (MATLAB).

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 “State Spaces”.

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.

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

        if nargin == 0
            space = stateSpaceSE2;
        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.NumStateVariables,
        "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 invalid");
    end

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

```

```
        else
            isValid = false;
            lastValid = interpStates(firstInvalidIdx-1,:);
        end
    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 1s and 0s

Valid states, specified as a m -element vector of 1s and 0s.

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.

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

        if nargin == 0
            space = stateSpaceSE2;
        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.NumStateVariables,
        "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 invalid");
    end

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

Valid states, specified as a m -element vector of 1s and 0s.

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.

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

        if nargin == 0
            space = stateSpaceSE2;
        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.NumStateVariables,
        "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 invalid");
    end

    % Interpolate at a fixed interval between states and check state validity
    numInterpPoints = 100;
    interpStates = obj.StateSpace.interpolate(state1, state2, linspace(0,1,numInterpPoints));
    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` | `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/s²)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/s²)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

[4e-14 4e-14 4e-14] (default) | positive scalar | 3-element row vector

Multiplicative process noise variance from the accelerometer bias in (m/s²)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
 residual Residuals and residual covariances from direct state measurements

fusegps	Correct states using GPS data
residualgps	Residuals and residual covariance from GPS measurements
pose	Current orientation and position estimate
predict	Update states using accelerometer and gyroscope data
reset	Reset internal states
stateinfo	Display state vector information

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));
mseps = 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,

$$x_{k|k-1} = f(\hat{x}_{k-1|k-1})$$

the predicted state estimate is:

$$x_k | k-1 =$$

$$\begin{array}{l}
 \left. \begin{array}{l}
 q_0 + \Delta t * q_1(\text{gyrobias}_X/2 - \text{gyro}_X/2) + \Delta t * q_2 * (\text{gyrobias}_Y/2 - \text{gyro}_Y/2) + \Delta t * q_3 * (\text{gyrobias}_Z/2 - \text{gyro}_Z/2) \\
 q_1 - \Delta t * q_0(\text{gyrobias}_X/2 - \text{gyro}_X/2) + \Delta t * q_3 * (\text{gyrobias}_Y/2 - \text{gyro}_Y/2) - \Delta t * q_2 * (\text{gyrobias}_Z/2 - \text{gyro}_Z/2) \\
 q_2 - \Delta t * q_3(\text{gyrobias}_X/2 - \text{gyro}_X/2) - \Delta t * q_0 * (\text{gyrobias}_Y/2 - \text{gyro}_Y/2) + \Delta t * q_1 * (\text{gyrobias}_Z/2 - \text{gyro}_Z/2) \\
 q_3 + \Delta t * q_2(\text{gyrobias}_X/2 - \text{gyro}_X/2) - \Delta t * q_1 * (\text{gyrobias}_Y/2 - \text{gyro}_Y/2) - \Delta t * q_0 * (\text{gyrobias}_Z/2 - \text{gyro}_Z/2) \\
 \quad - \text{gryobias}_X * (\Delta t * \lambda_{\text{gyro}} - 1) \\
 \quad - \text{gryobias}_Y * (\Delta t * \lambda_{\text{gyro}} - 1) \\
 \quad - \text{gryobias}_Z * (\Delta t * \lambda_{\text{gyro}} - 1) \\
 \quad \text{position}_N + \Delta t * v_N \\
 \quad \text{position}_E + \Delta t * v_E \\
 \quad \text{position}_D + \Delta t * v_D
 \end{array} \right\} \\
 v_N + \Delta t * \left\{ \begin{array}{l}
 q_0 * (q_0 * (\text{accelbias}_X - \text{accel}_X) - q_3 * (\text{accelbias}_Y - \text{accel}_Y) + q_2 * (\text{accelbias}_Z - \text{accel}_Z)) - g_N + \\
 q_2 * (q_1 * (\text{accelbias}_Y - \text{accel}_Y) - q_2 * (\text{accelbias}_X - \text{accel}_X) + q_0 * (\text{accelbias}_Z - \text{accel}_Z)) + \\
 q_1 * (q_1 * (\text{accelbias}_X - \text{accel}_X) + q_2 * (\text{accelbias}_Y - \text{accel}_Y) + q_3 * (\text{accelbias}_Z - \text{accel}_Z)) - \\
 q_3 * (q_3 * (\text{accelbias}_X - \text{accel}_X) + q_0 * (\text{accelbias}_Y - \text{accel}_Y) - q_1 * (\text{accelbias}_Z - \text{accel}_Z))
 \end{array} \right\} \\
 v_E + \Delta t * \left\{ \begin{array}{l}
 q_0 * (q_3 * (\text{accelbias}_X - \text{accel}_X) + q_0 * (\text{accelbias}_Y - \text{accel}_Y) - q_1 * (\text{accelbias}_Z - \text{accel}_Z)) - g_E - \\
 q_1 * (q_1 * (\text{accelbias}_Y - \text{accel}_Y) - q_2 * (\text{accelbias}_X - \text{accel}_X) + q_0 * (\text{accelbias}_Z - \text{accel}_Z)) + \\
 q_2 * (q_1 * (\text{accelbias}_X - \text{accel}_X) + q_2 * (\text{accelbias}_Y - \text{accel}_Y) + q_3 * (\text{accelbias}_Z - \text{accel}_Z)) + \\
 q_3 * (q_3 * (\text{accelbias}_X - \text{accel}_X) + q_0 * (\text{accelbias}_Y - \text{accel}_Y) - q_1 * (\text{accelbias}_Z - \text{accel}_Z))
 \end{array} \right\}
 \end{array}$$

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

`insfilterAsync` | `insfilterErrorState` | `insfilterMARG`

Introduced in R2018b

correct

Correct states using direct state measurements

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

fusegps

Correct states using GPS data

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 (m²)

3-by-3 matrix

Position measurement covariance of GPS receiver in m², 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/s²)

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

Syntax

```
[orientation, position, velocity] = pose(FUSE)
[orientation, position, velocity] = pose(FUSE, format)
```

Description

[orientation, position, velocity] = pose(FUSE) returns the current estimate of the pose.

[orientation, position, velocity] = pose(FUSE, format) returns the current estimate of the pose with orientation in the specified orientation format.

Input Arguments

FUSE — NHConstrainedIMUGPSFuser object

object

Object of NHConstrainedIMUGPSFuser, created by the insfilter function.

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

orientation — Orientation estimate in the local NED coordinate system

quaternion (default) | 3-by-3 rotation matrix

Orientation estimate in the local NED coordinate system, specified as a scalar quaternion or 3-by-3 rotation matrix. The quaternion or rotation matrix represents a frame rotation from the local NED reference frame to the body reference frame.

Data Types: single | double | quaternion

position — Position estimate in the local NED coordinate system (m)

3-element row vector

Position estimate in the local NED coordinate system in meters, returned as a 3-element row vector.

Data Types: single | double

velocity — Velocity estimate in local NED coordinate system (m/s)

3-element row vector

Velocity estimate in the local NED coordinate system 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

`NHConstrainedIMUGPSFuser` | `insfilter`

Introduced in R2018b

predict

Update states using accelerometer and gyroscope data

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/s²)

3-element row vector

Accelerometer readings in m/s², 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

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

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

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 (m²)

3-by-3 matrix

Position measurement covariance of GPS receiver in m², 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/s²)

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 if 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 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

`insfilterNonholonomic`

Introduced in R2020a

stateinfo

Display state vector information

Syntax

```
stateinfo(FUSE)
```

Description

`stateinfo(FUSE)` displays the meaning of each index of the State property and the associated units.

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

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.

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 zero 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

<code>getOccupancy</code>	Get occupancy value of locations
<code>grid2local</code>	Convert grid indices to local coordinates
<code>grid2world</code>	Convert grid indices to world coordinates
<code>inflate</code>	Inflate each occupied grid location
<code>insertRay</code>	Insert ray from laser scan observation
<code>local2grid</code>	Convert local coordinates to grid indices
<code>local2world</code>	Convert local coordinates to world coordinates
<code>move</code>	Move map in world frame
<code>occupancyMatrix</code>	Convert occupancy grid to double matrix
<code>raycast</code>	Compute cell indices along a ray
<code>rayIntersection</code>	Find intersection points of rays and occupied map cells
<code>setOccupancy</code>	Set occupancy value of locations
<code>show</code>	Show grid values in a figure
<code>syncWith</code>	Sync map with overlapping map
<code>updateOccupancy</code>	Integrate probability observations at locations
<code>world2grid</code>	Convert world coordinates to grid indices
<code>world2local</code>	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.

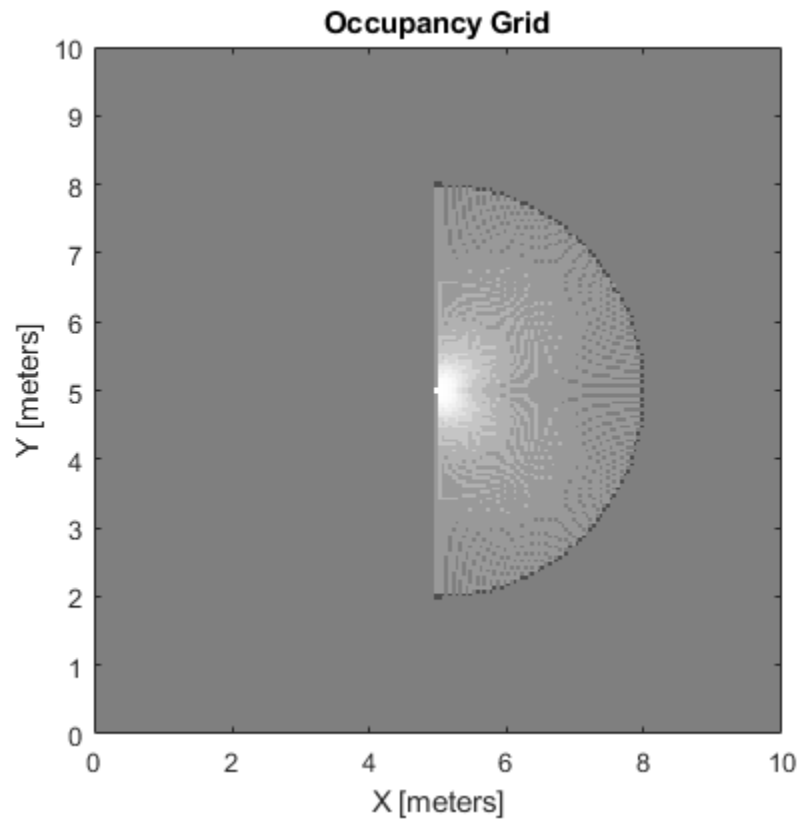
```
scan = lidarScan(ranges,angles);
```

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)
```

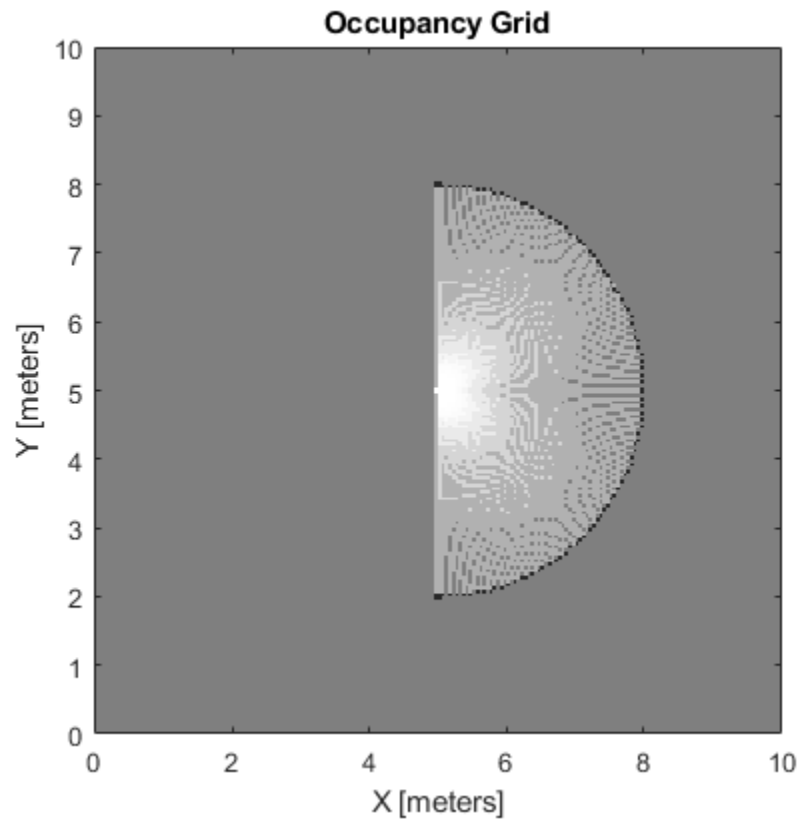
Check the occupancy of the spot directly in front of the vehicle.

```
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])
```

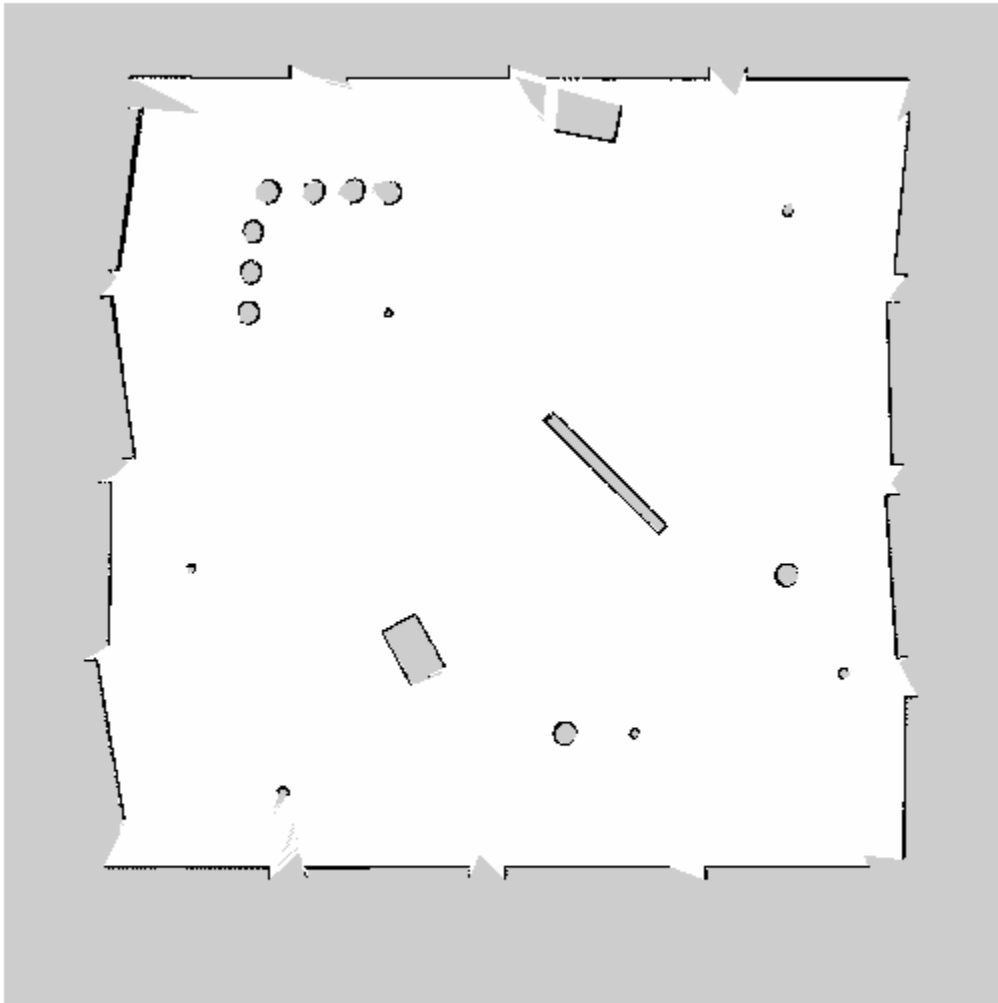
```
ans = 0.8448
```

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)
```

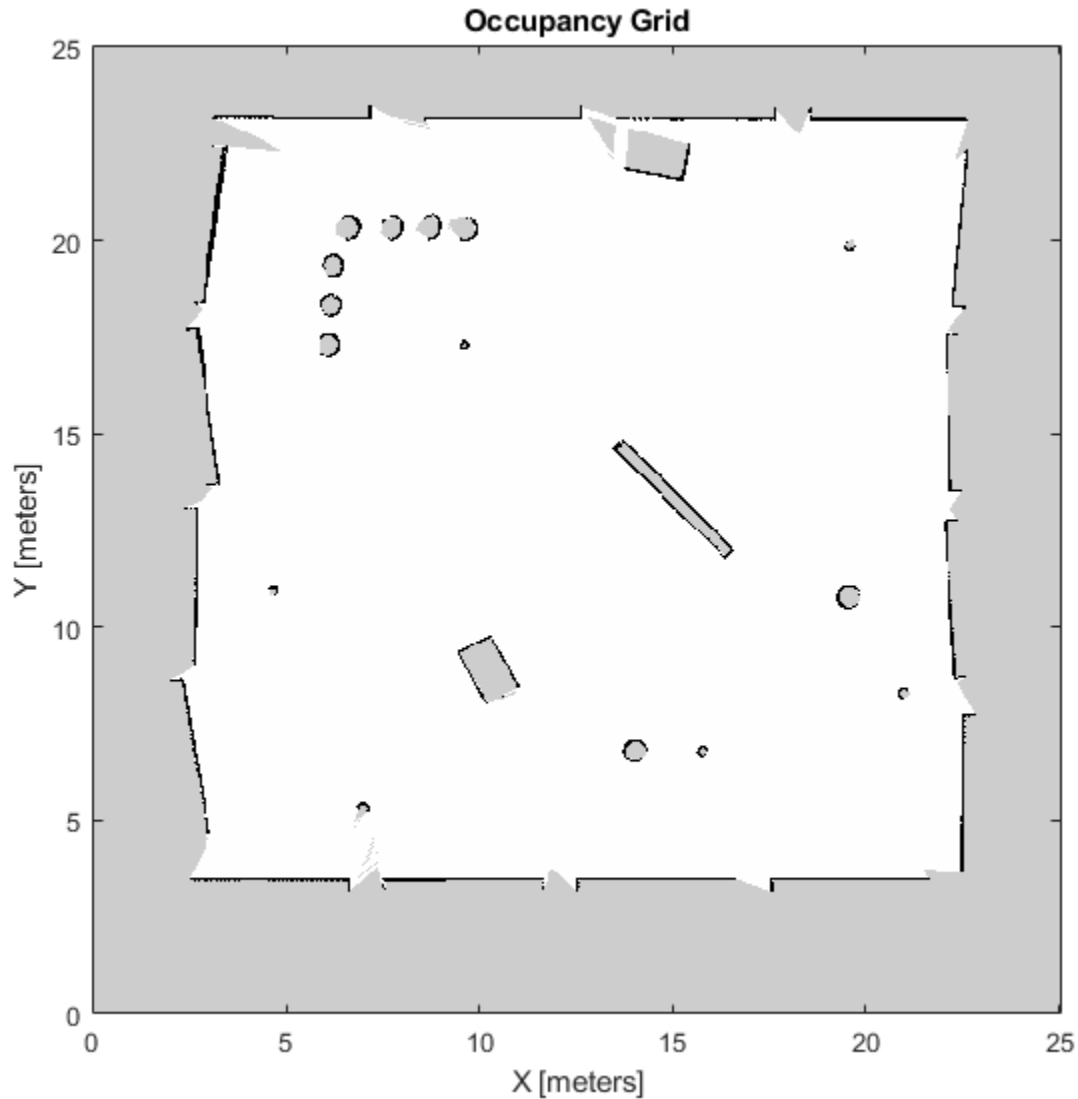


PGM values are expressed from 0 to 255 as `uint8`. Normalize these values by converting the cropped image to `double` and dividing each cell by 255. This image shows obstacles as values close to 0. Subtract the normalized image from 1 to get occupancy values with 1 representing occupied space.

```
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` | `controllerPurePursuit` | `mobileRobotPRM` | `readOccupancyGrid` | `writeOccupancyGrid`

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)
occVal = 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.

`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.

`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

binaryOccupancyMap | getOccupancy | occupancyMap

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

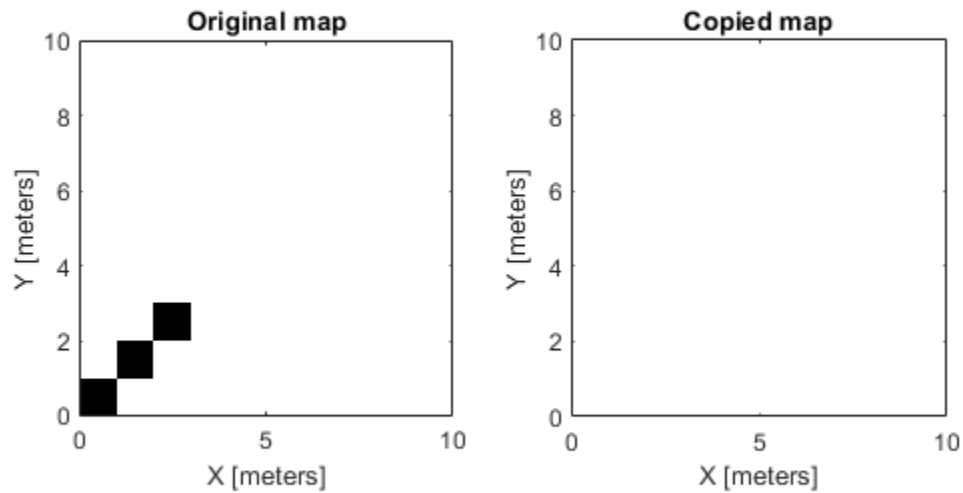
Copy an occupancy grid map object. Once copied, the original object can be modified without affecting the copied map.

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` | `getOccupancy` | `occupancyMap` | `occupancyMatrix`

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( ___ )

[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.

`[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.

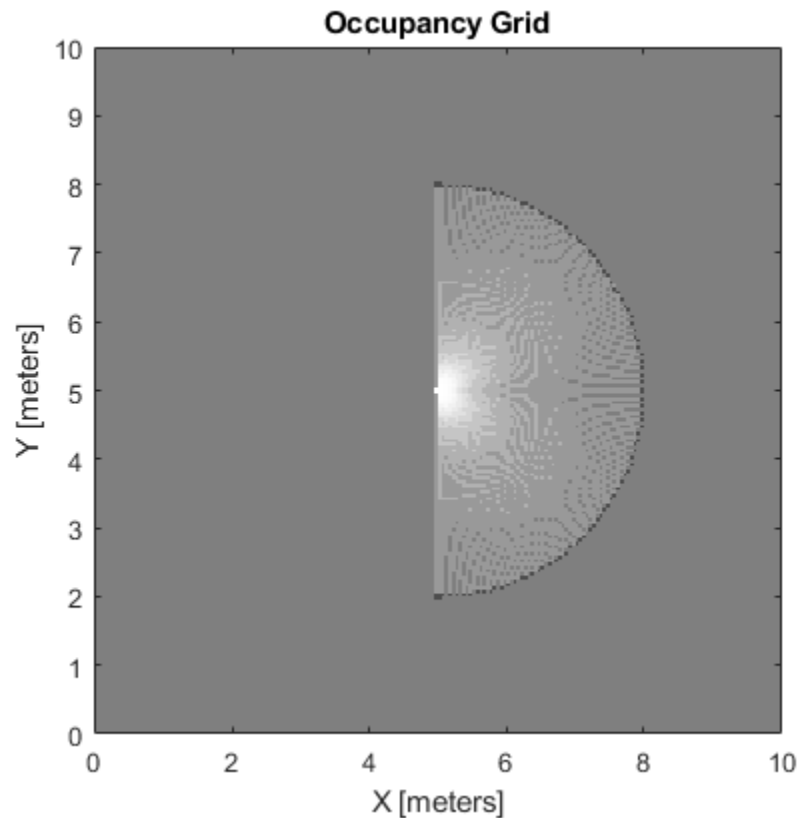
```
scan = lidarScan(ranges,angles);
```

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)
```



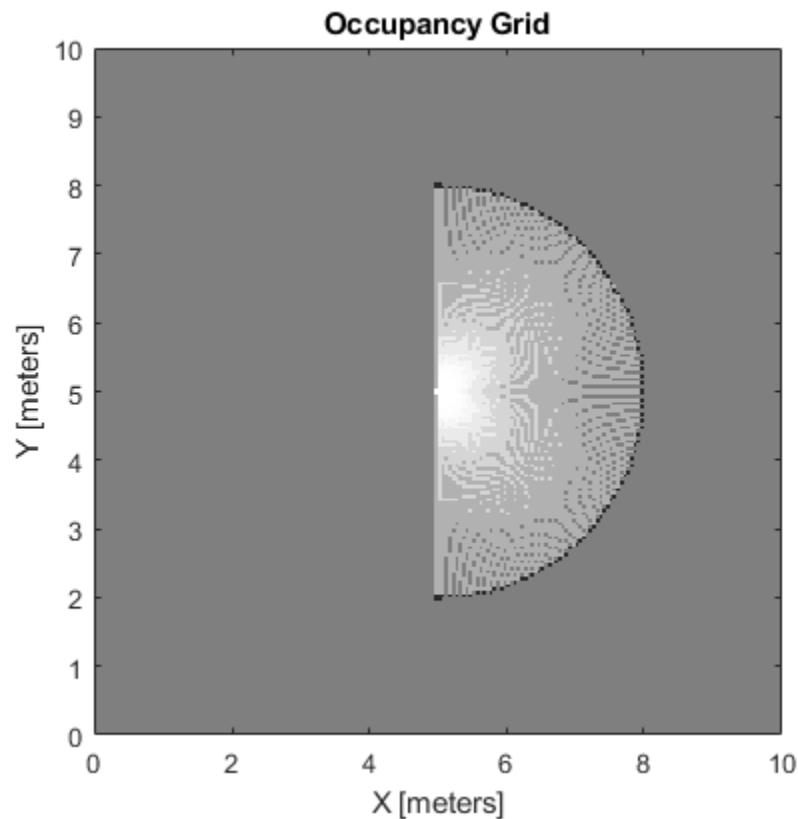
Check the occupancy of the spot directly in front of the vehicle.

```
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

`checkOccupancy` | `occupancyMap`

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

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.

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

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.

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

`binaryOccupancyMap` | `grid2local` | `occupancyMap` | `world2grid`

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 a 10 m-by-10 m empty map.

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

Update the occupancy of world locations with specific probability values and display the map.

```
x = [1.2; 2.3; 3.4; 4.5; 5.6];
```

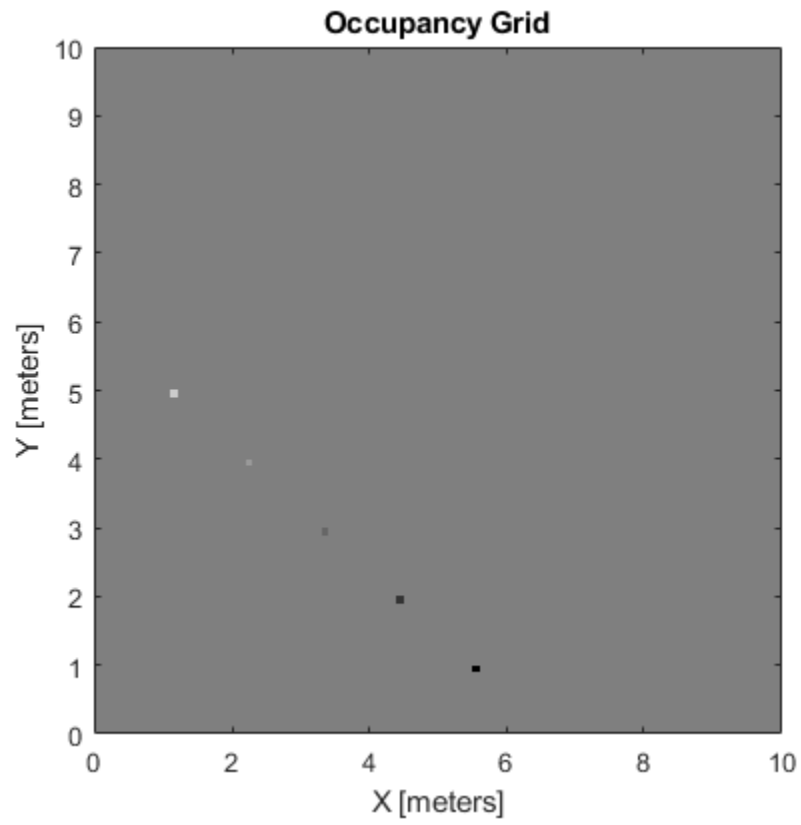
```
y = [5.0; 4.0; 3.0; 2.0; 1.0];
```

```
pvalues = [0.2 0.4 0.6 0.8 1];
```

```
updateOccupancy(map,[x y],pvalues)
```

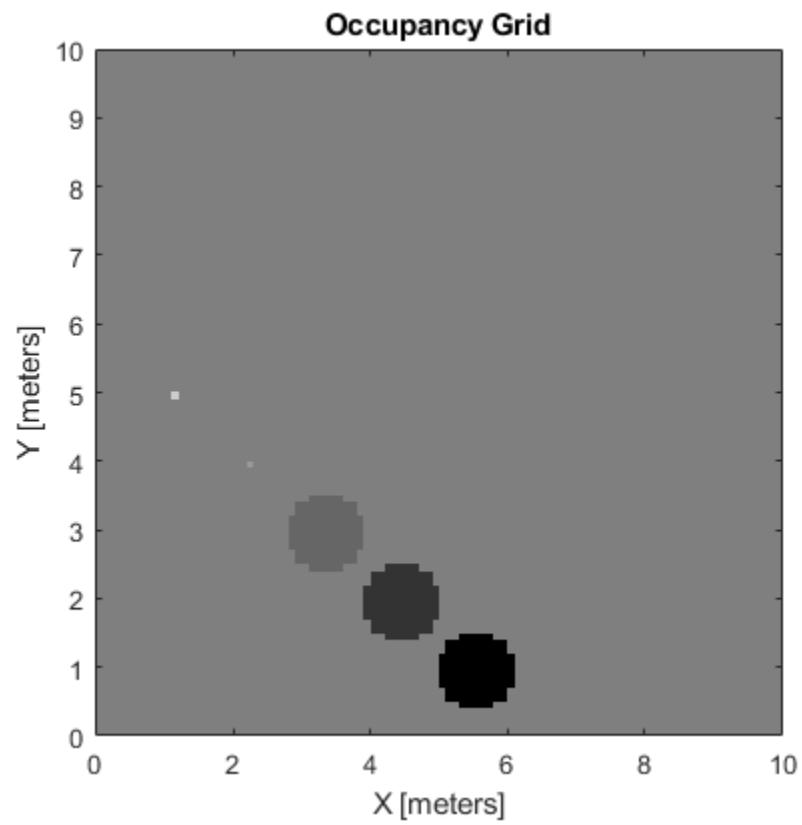
```
figure
```

```
show(map)
```



Inflate occupied areas by a radius of 0.5 m. Larger occupancy values overwrite the smaller values.

```
inflate(map,0.5)  
figure  
show(map)
```

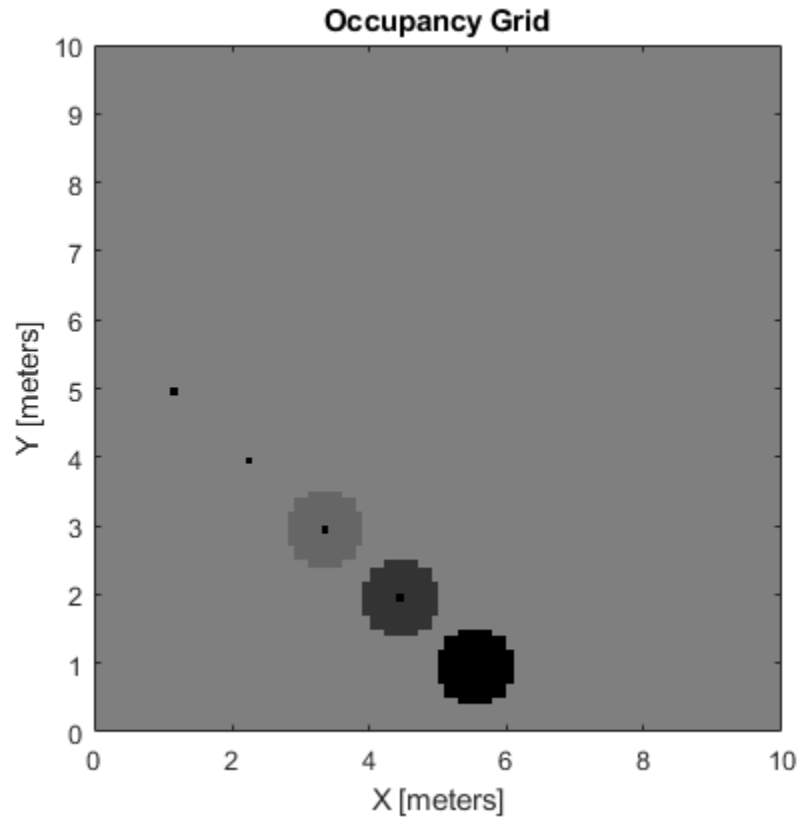


Get grid locations from world locations.

```
ij = world2grid(map,[x y]);
```

Set grid locations to occupied 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` | `getOccupancy` | `occupancyMap`

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.

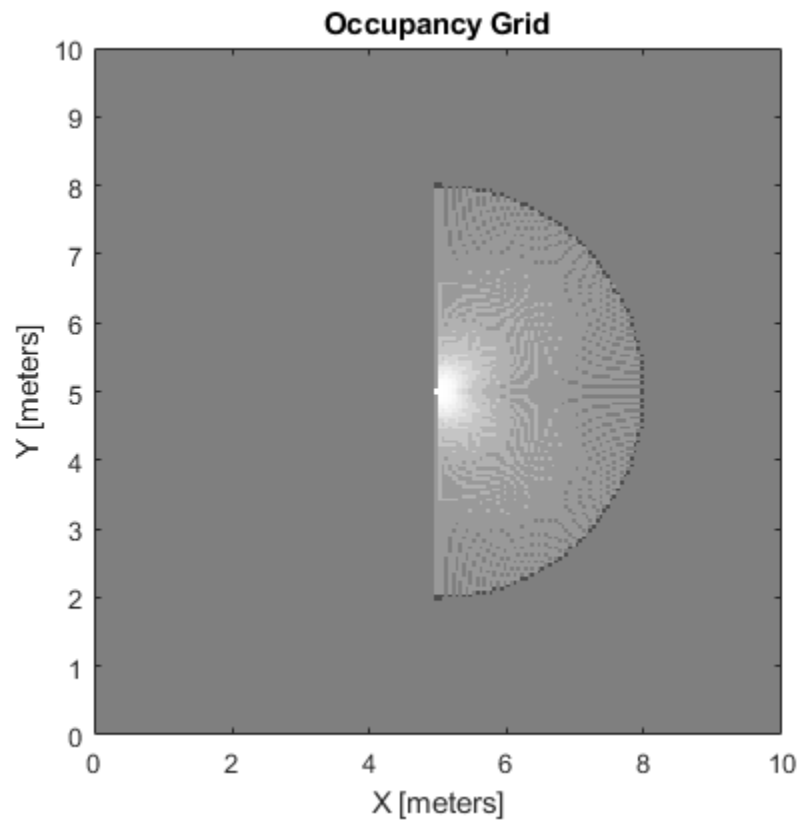
```
scan = lidarScan(ranges,angles);
```

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)
```



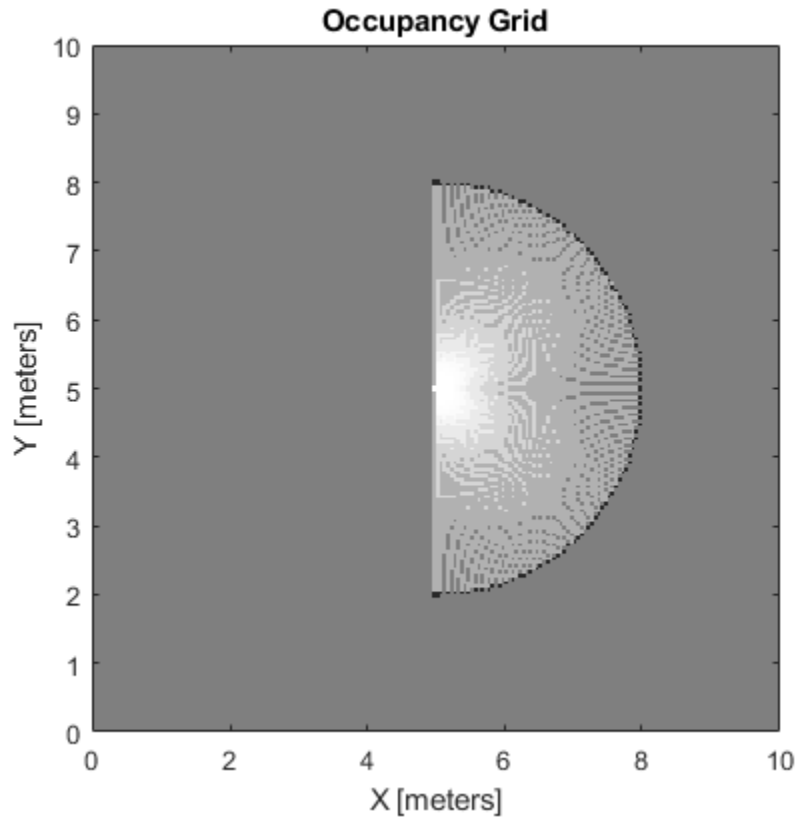
Check the occupancy of the spot directly in front of the vehicle.

```
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

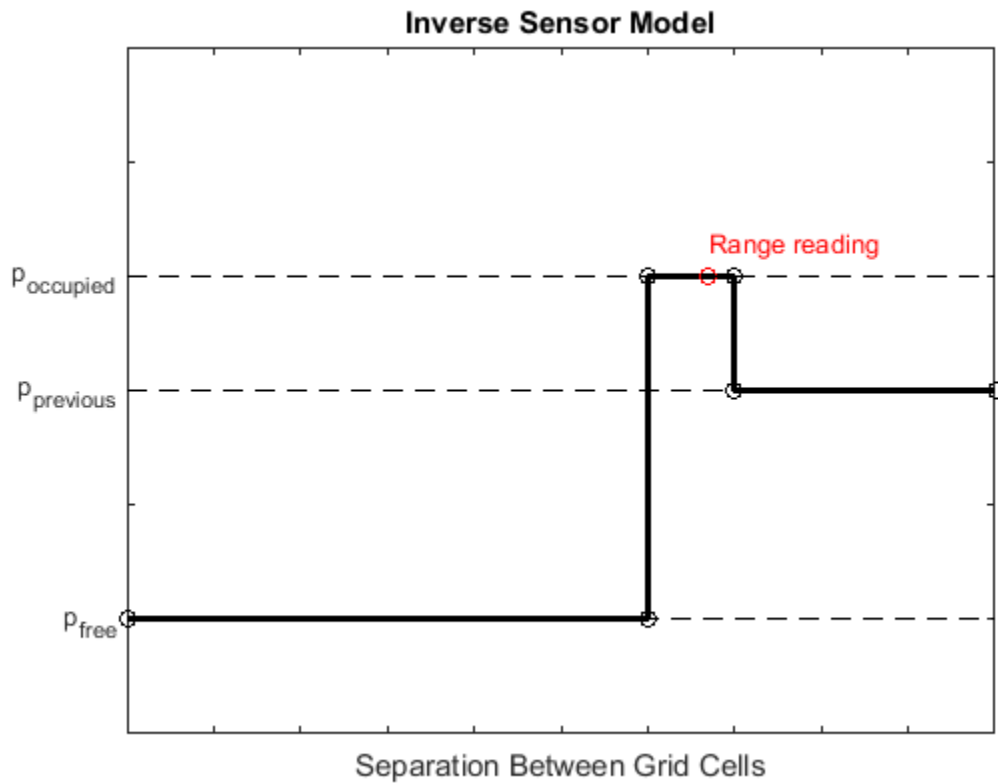
two-element vector

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 2-465.

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

[binaryOccupancyMap](#) | [lidarScan](#) | [occupancyMap](#) | [raycast](#)

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

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 — 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

`binaryOccupancyMap` | `grid2world` | `occupancyMap`

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

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 — 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` | `occupancyMap` | `world2local`

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.

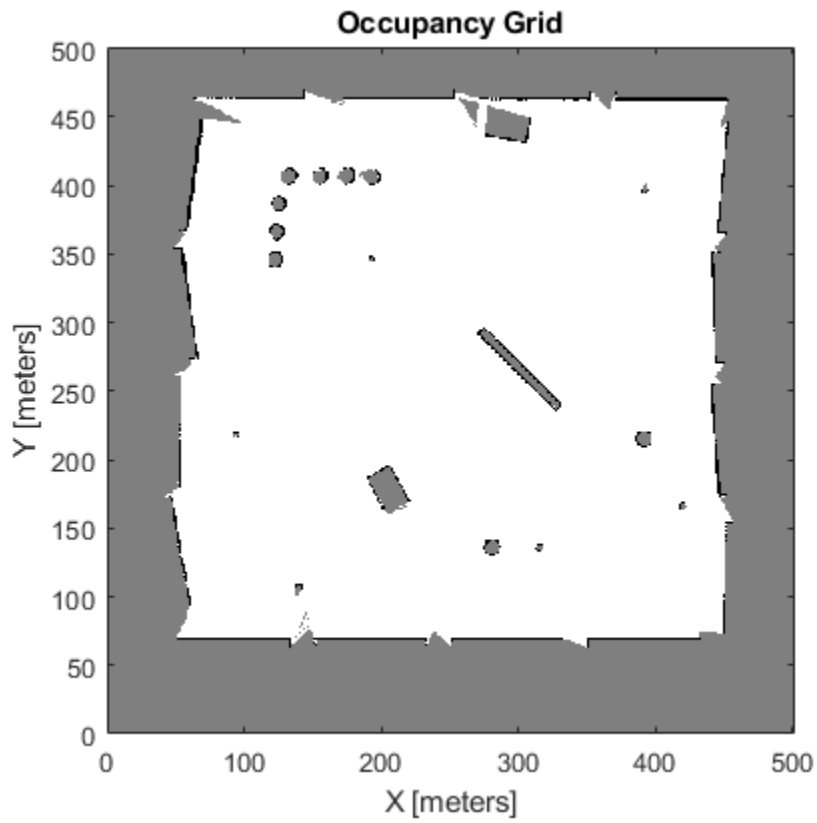
Examples

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.

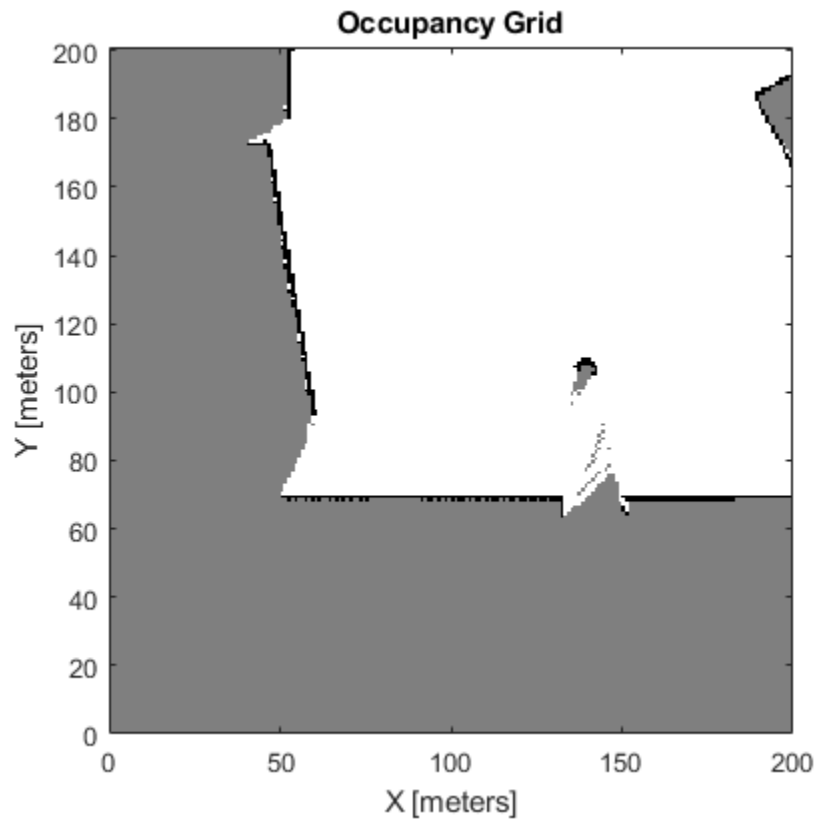
Load example maps. Create an occupancy map from the `ternaryMap`.

```
load exampleMaps.mat
map = occupancyMap(ternaryMap);
show(map)
```



Create a smaller local map.

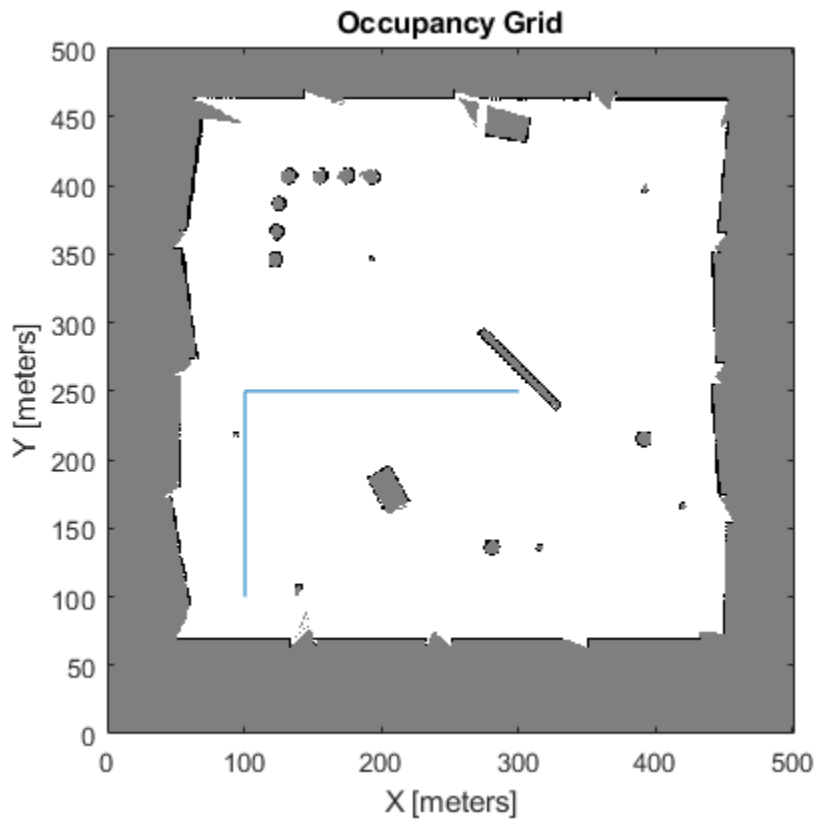
```
mapLocal = occupancyMap(ternaryMap(end-200:end,1:200));  
show(mapLocal)
```

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
        200 200];
show(map)
hold on
plot(path(:,1),path(:,2))
hold off
```



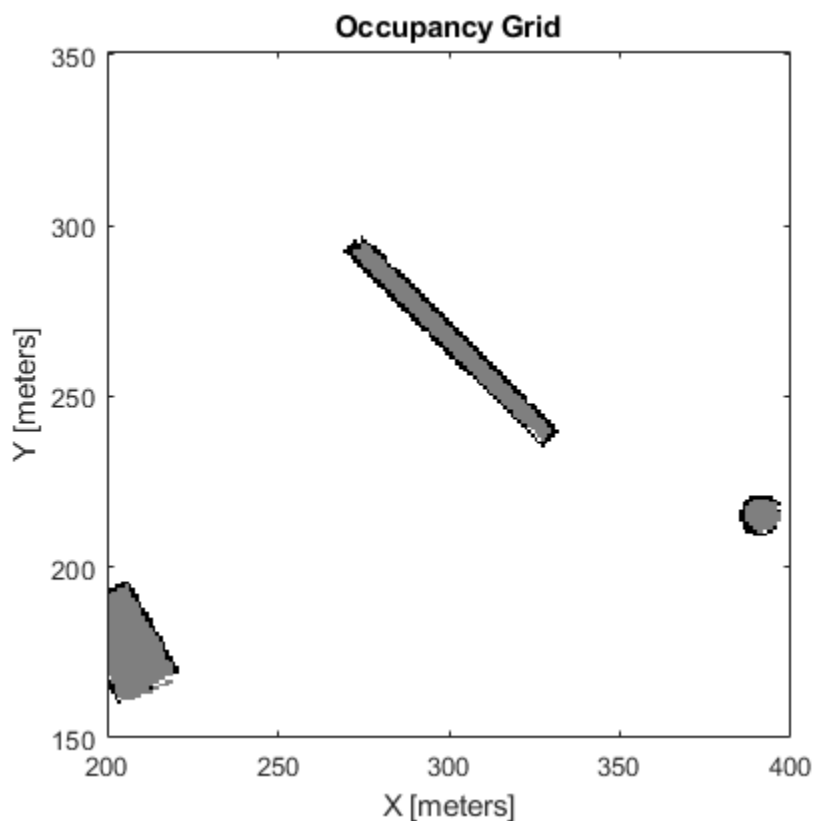
Create a loop for moving between points by the map resolution. Divide the difference between points by the map resolution to see how many incremental moves you can make.

```

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
end

```



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.

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`.

SynWith — 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`binaryOccupancyMap` | `occupancyMap` | `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

`binaryOccupancyMap` | `getOccupancy` | `occupancyMap` | `show`

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.

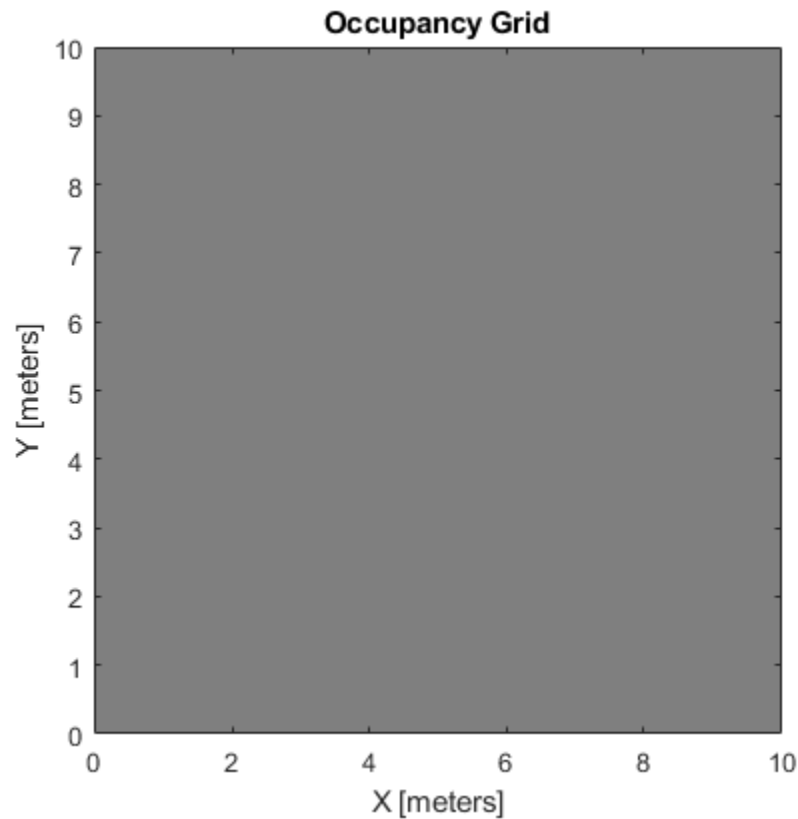
Examples

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)
```

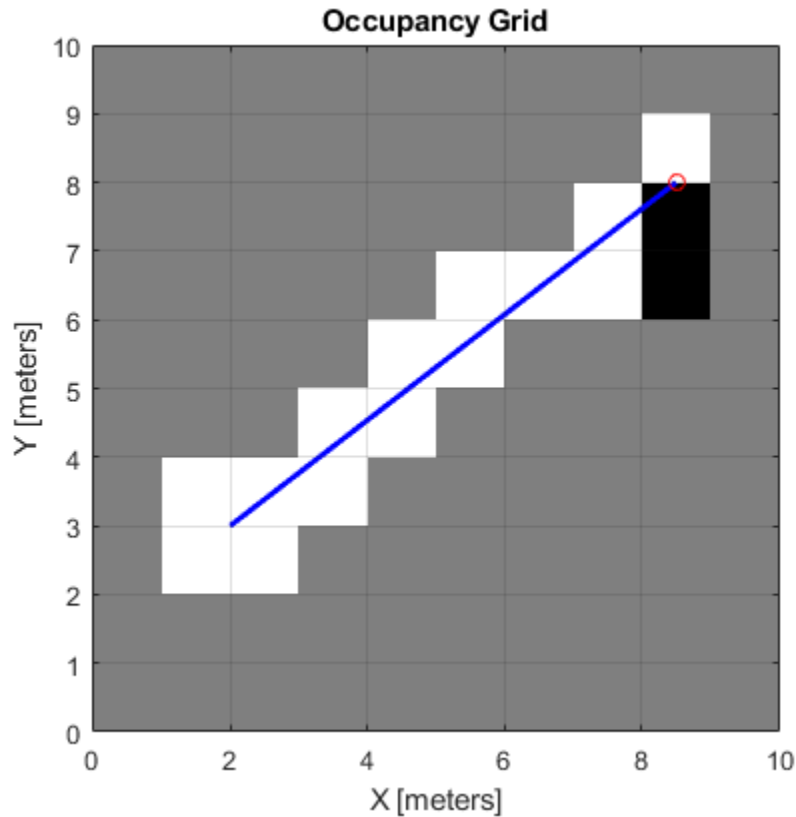


Get the grid indices of the midpoints and end points of a ray from [2 3] to [8.5 8]. Set occupancy values for these grid indices. Midpoints are treated as open space. Update endpoints with an occupied observation.

```
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 θ (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 only given if the point intersect grid locations.

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.

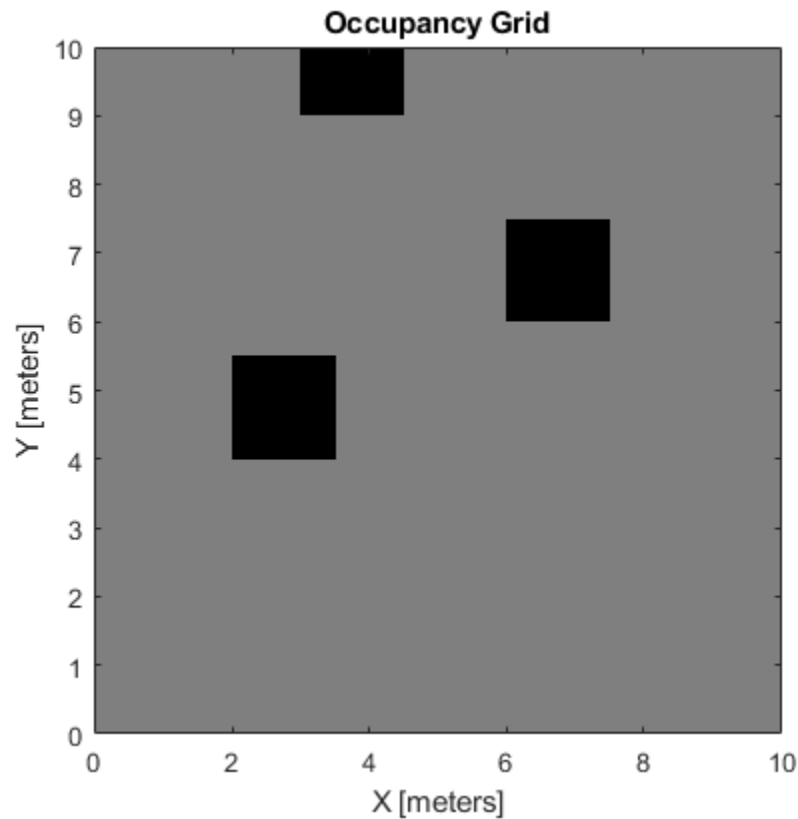
`intersectionPts = rayIntersection(map,pose,angles,maxrange,threshold)` returns intersection points based on the specified threshold for the occupancy values. Values greater than or equal to the threshold are considered occupied.

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.

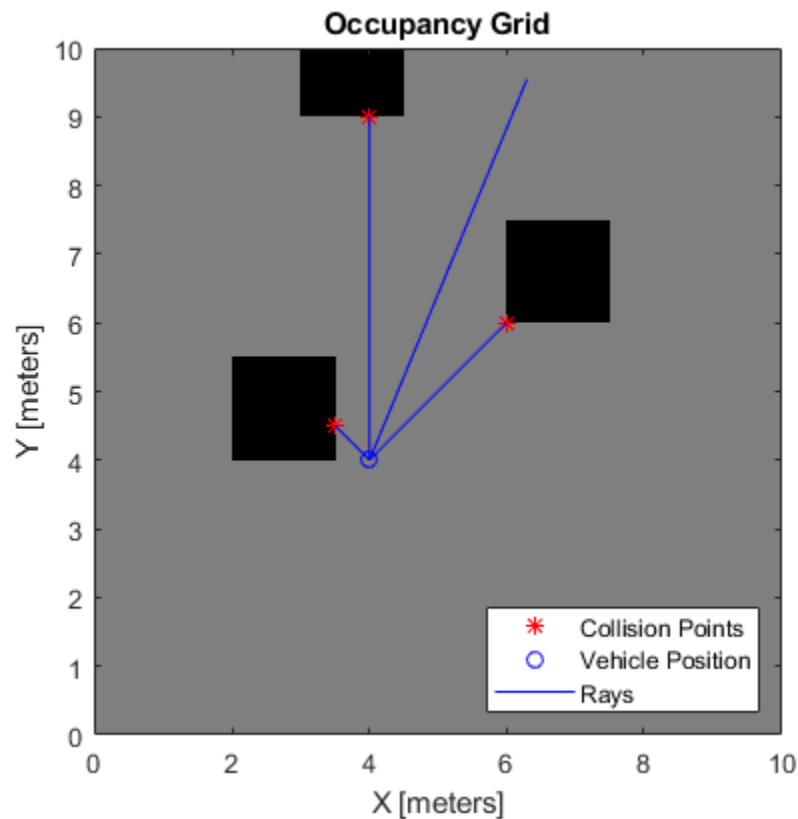
```
maxrange = 6;
angles = [pi/4, -pi/4, 0, -pi/8];
vehiclePose = [4, 4, pi/2];
intsectionPts = rayIntersection(map, vehiclePose, angles, maxrange, 0.7)
```

```
intsectionPts = 4x2
    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 θ (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

binaryOccupancyMap | occupancyMap | raycast | updateOccupancy

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

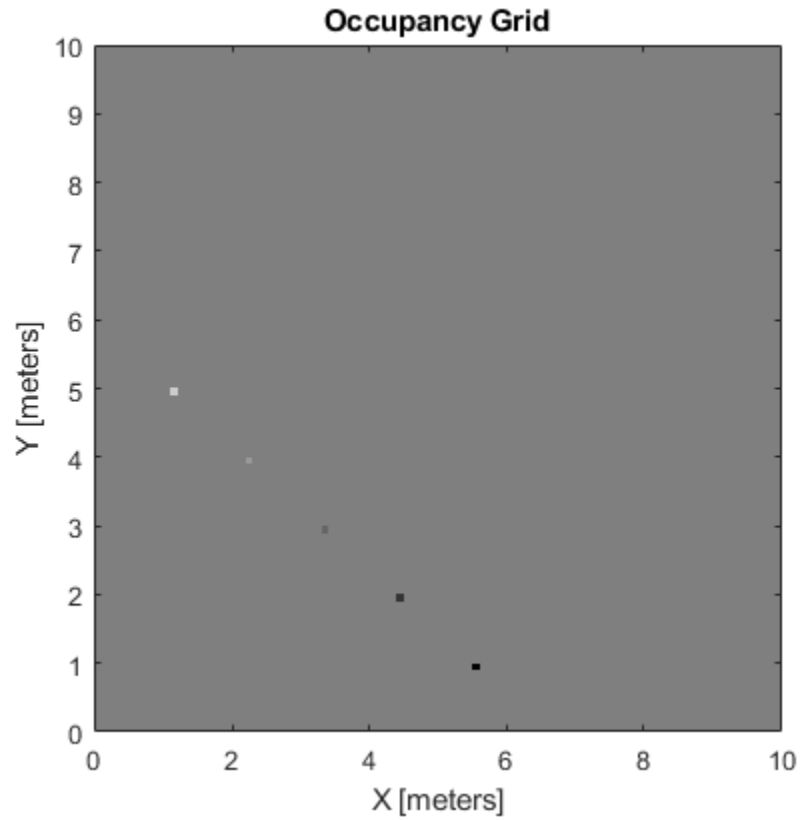
Create a 10 m-by-10 m empty map.

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

Update the occupancy of world locations with specific probability values and display the map.

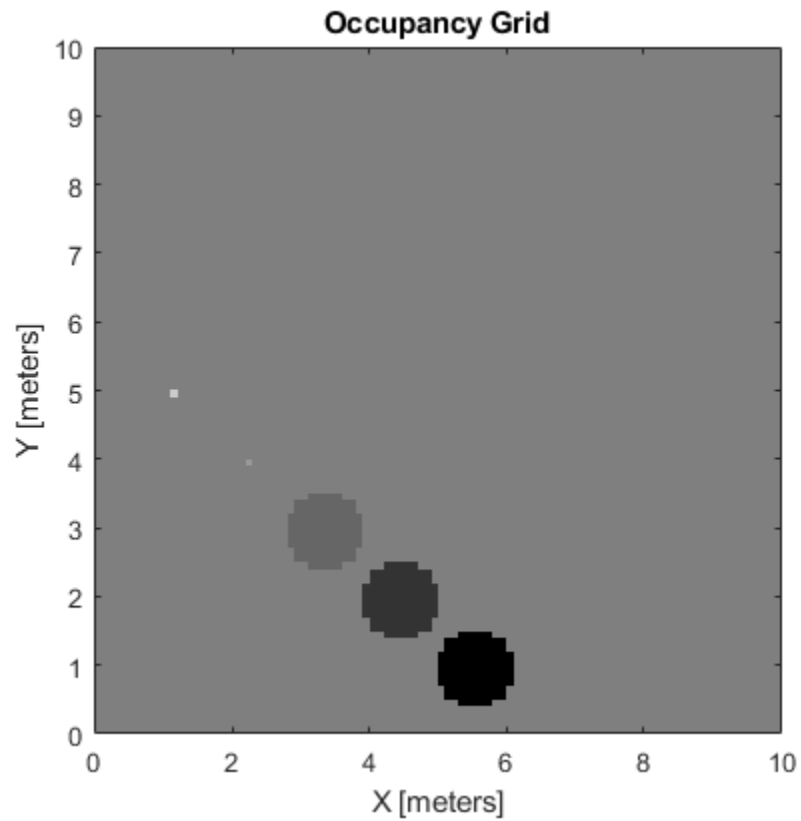
```
x = [1.2; 2.3; 3.4; 4.5; 5.6];
y = [5.0; 4.0; 3.0; 2.0; 1.0];
```

```
pvalues = [0.2 0.4 0.6 0.8 1];  
updateOccupancy(map,[x y],pvalues)  
figure  
show(map)
```



Inflate occupied areas by a radius of 0.5 m. Larger occupancy values overwrite the smaller values.

```
inflate(map,0.5)  
figure  
show(map)
```

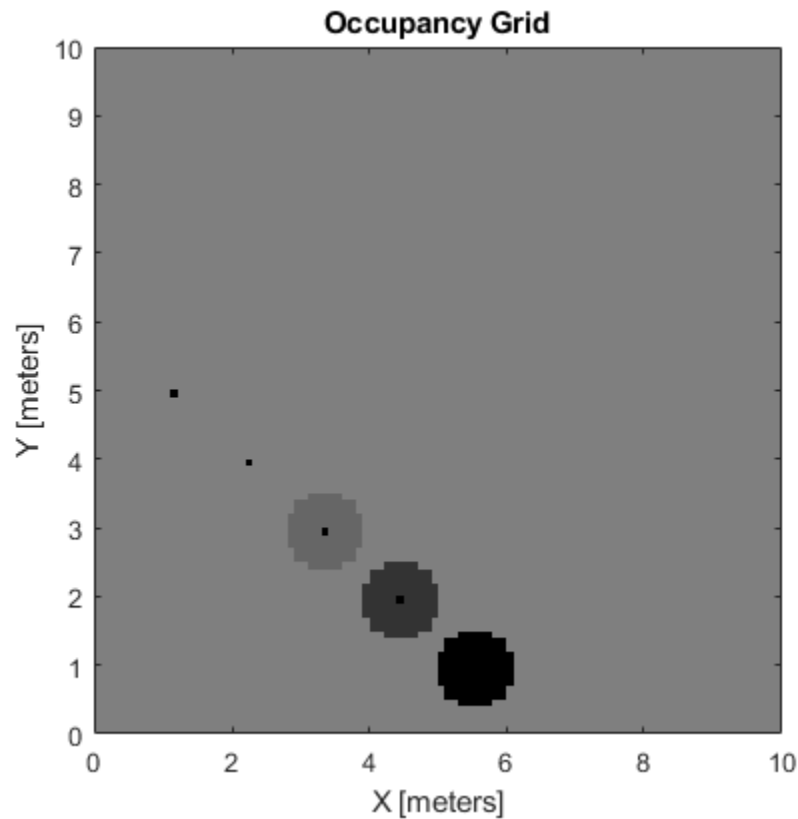


Get grid locations from world locations.

```
ij = world2grid(map,[x y]);
```

Set grid locations to occupied 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 localtwo-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 gridtwo-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

`binaryOccupancyMap` | `getOccupancy` | `occupancyMap`

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 a 10 m-by-10 m empty map.

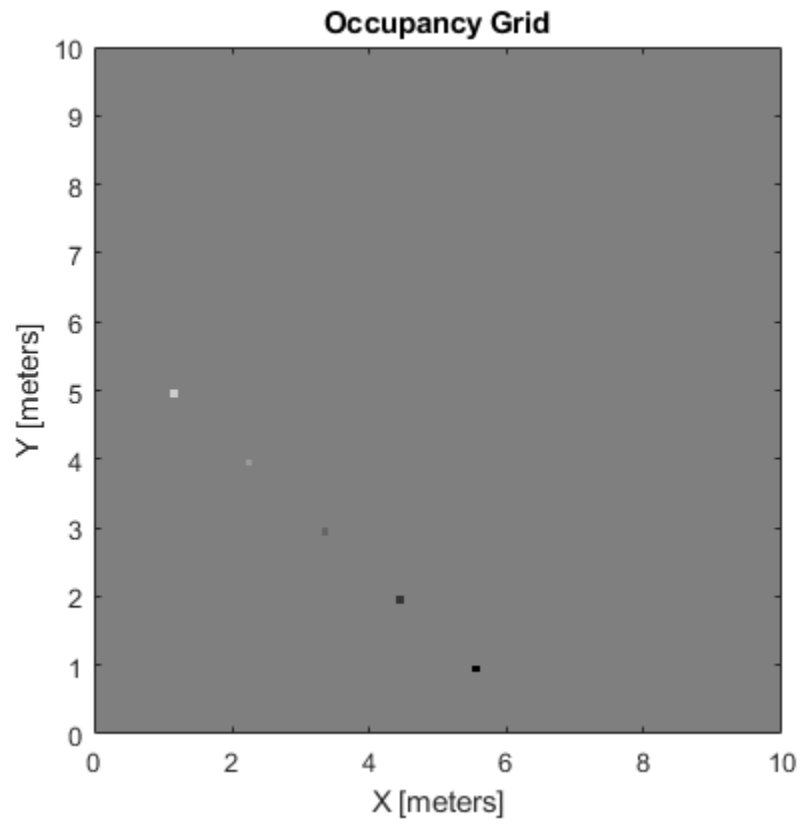
```
map = occupancyMap(10,10,10);
```

Update the occupancy of world locations with specific 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];

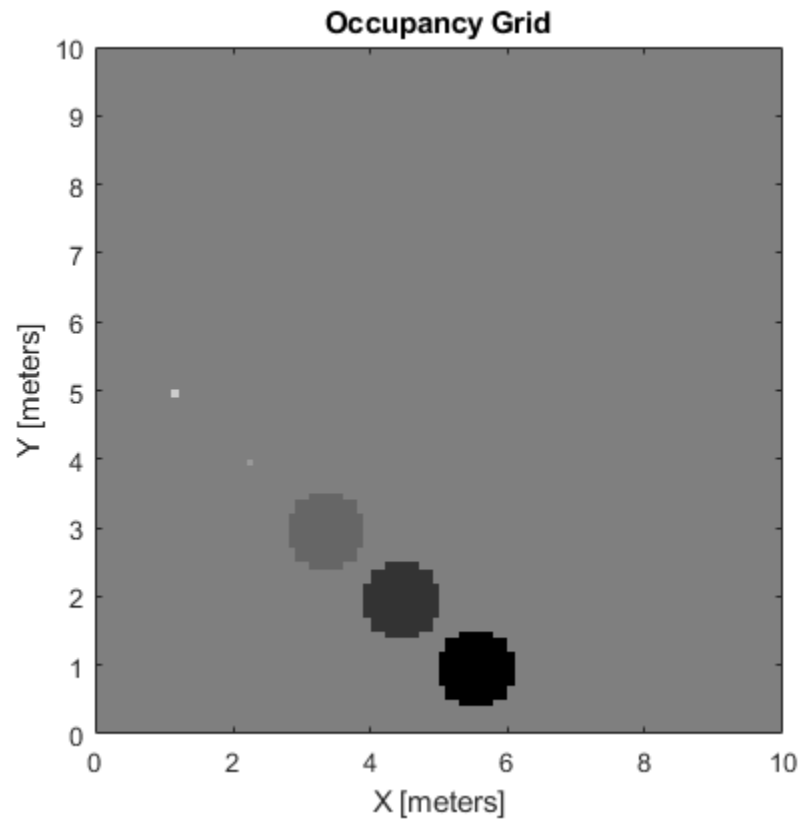
pvalues = [0.2 0.4 0.6 0.8 1];

updateOccupancy(map,[x y],pvalues)
figure
show(map)
```



Inflate occupied areas by a radius of 0.5 m. Larger occupancy values overwrite the smaller values.

```
inflate(map,0.5)  
figure  
show(map)
```

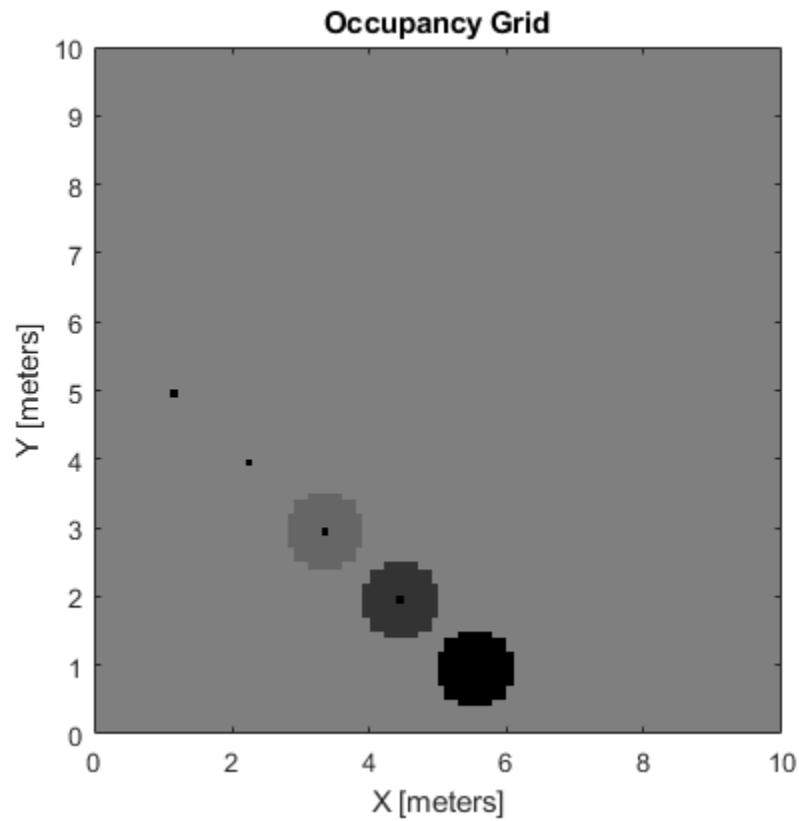


Get grid locations from world locations.

```
ij = world2grid(map,[x y]);
```

Set grid locations to occupied locations.

```
setOccupancy(map,ij,ones(5,1),'grid')  
figure  
show(map)
```

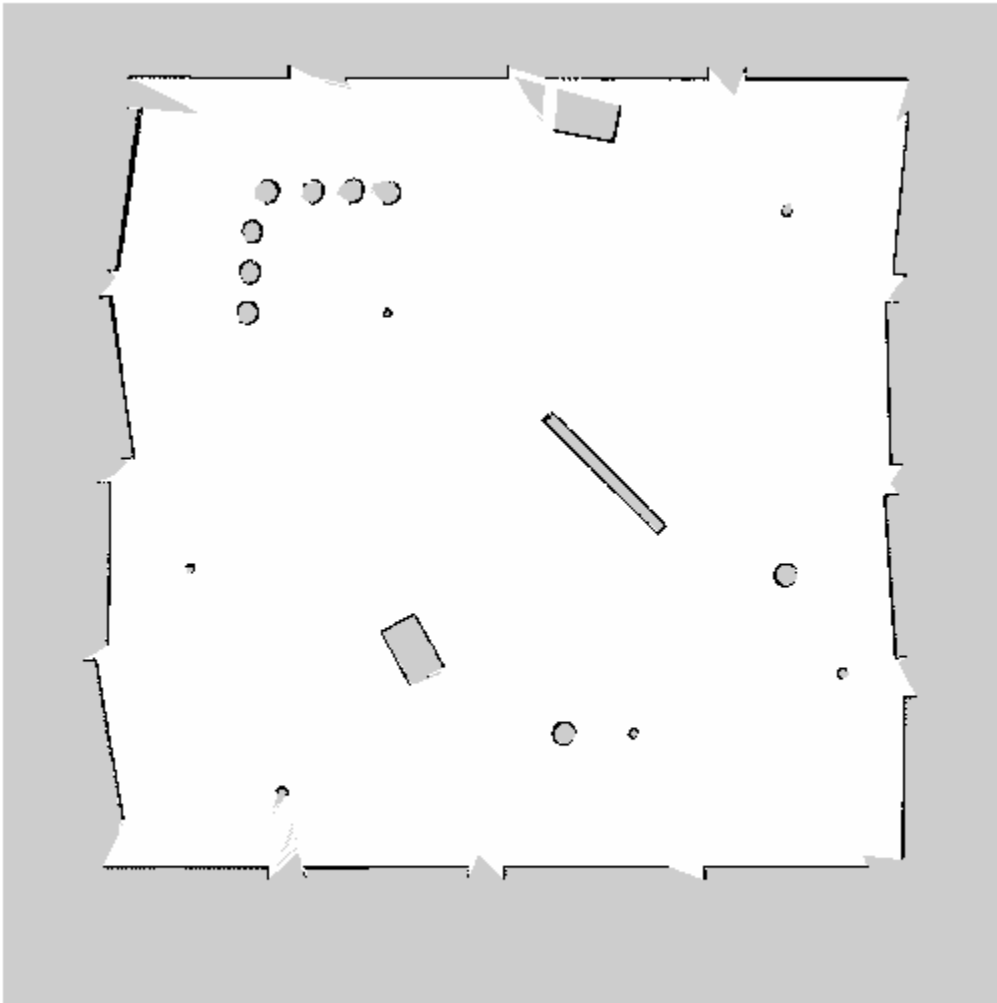


Convert PGM Image to Map

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)
```

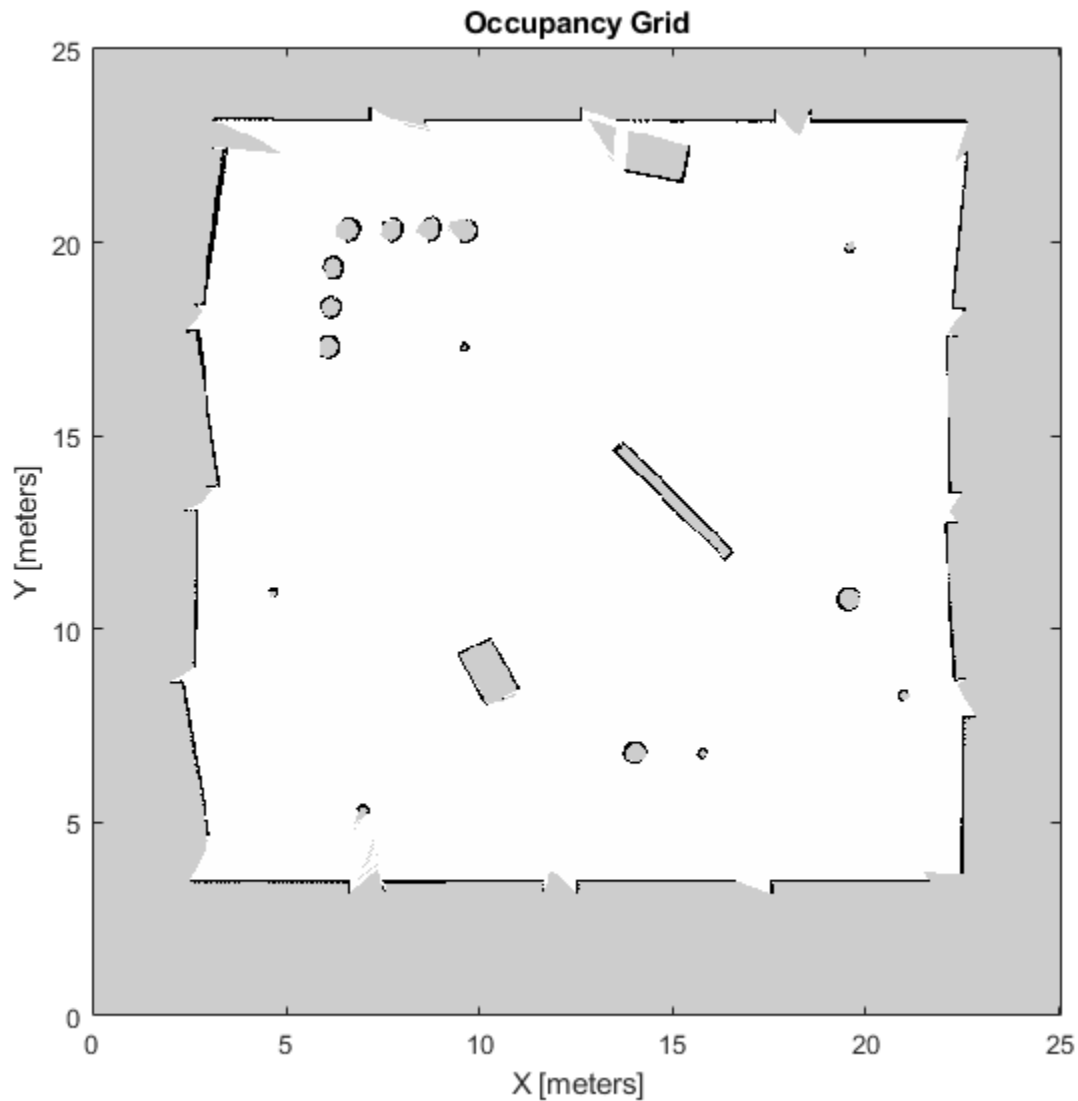


PGM values are expressed from 0 to 255 as `uint8`. Normalize these values by converting the cropped image to `double` and dividing each cell by 255. This image shows obstacles as values close to 0. Subtract the normalized image from 1 to get occupancy values with 1 representing occupied space.

```
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` | `binaryOccupancyMap` | `occupancyMap` | `occupancyMatrix`

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` are set to `map.DefaultValue`.

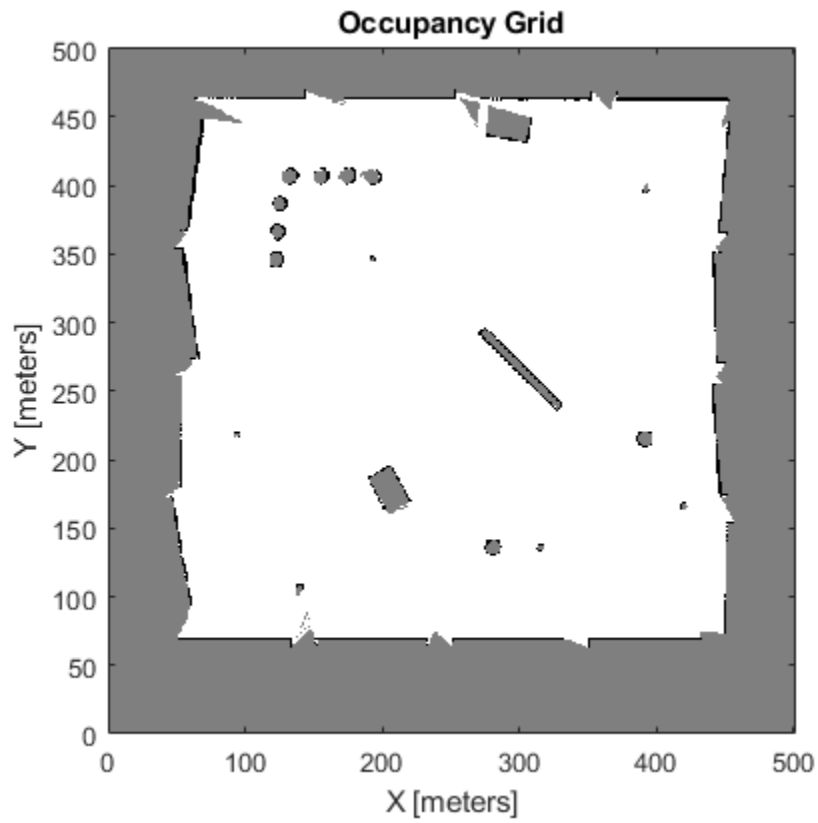
Examples

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.

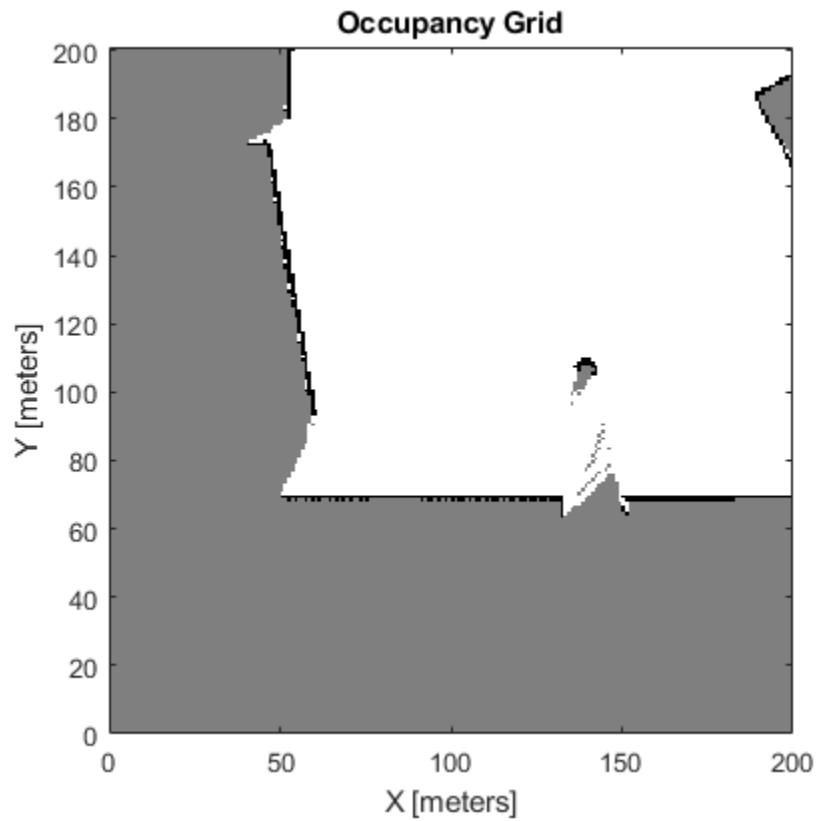
Load example maps. Create an occupancy map from the `ternaryMap`.

```
load exampleMaps.mat  
map = occupancyMap(ternaryMap);  
show(map)
```



Create a smaller local map.

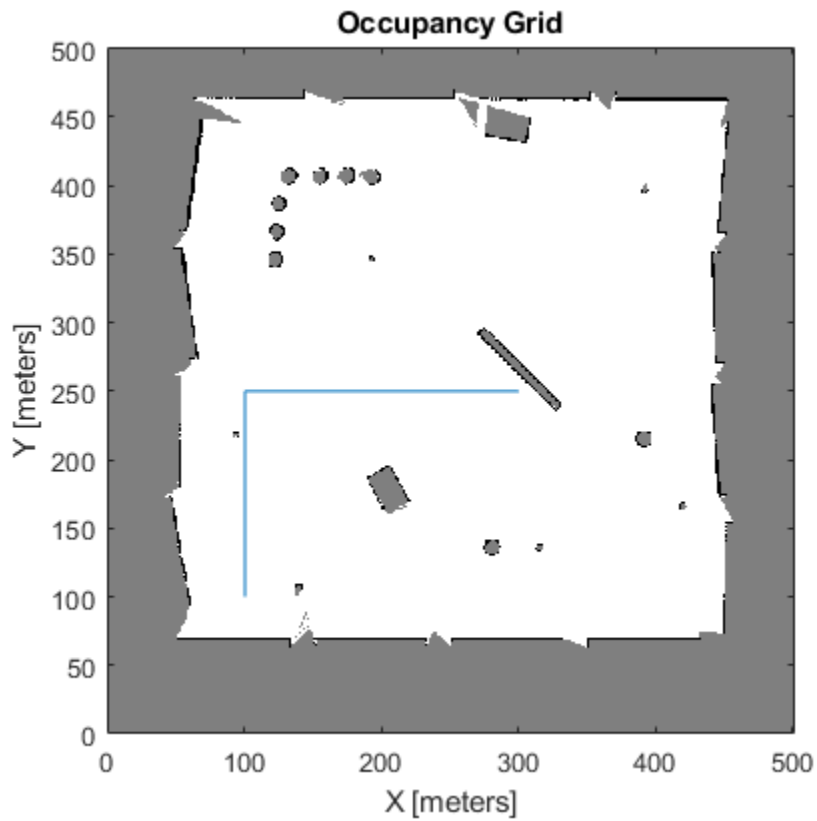
```
mapLocal = occupancyMap(ternaryMap(end-200:end,1:200));  
show(mapLocal)
```



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
        200 100];
show(map)
hold on
plot(path(:,1),path(:,2))
hold off
```



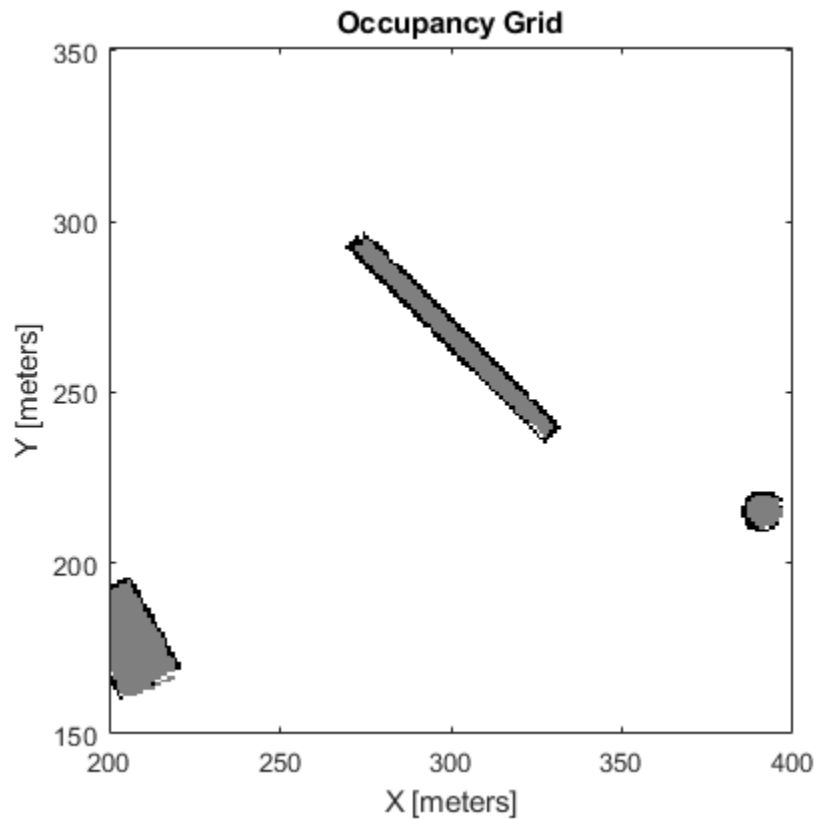
Create a loop for moving between points by the map resolution. Divide the difference between points by the map resolution to see how many incremental moves you can make.

```

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
end

```



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.

sourcemap — 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.

See Also

`binaryOccupancyMap` | `occupancyMap`

Topics

"Occupancy Grids"

Introduced in R2019b

updateOccupancy

Integrate probability observations at locations

Syntax

```
updateOccupancy(map,xy,obs)
updateOccupancy(map,ij,occval,'grid')
```

Description

`updateOccupancy(map,xy,obs)` probabilistically integrates the observation values, `obs`, to each coordinate specified in `xy`. Observation values are determined based on the “Inverse Sensor Model” on page 2-506.

`updateOccupancy(map,ij,occval,'grid')` probabilistically integrates the occupancy values, `occval`, to the specified grid locations, `ij`, instead of world coordinates.

Examples

Create and Modify Occupancy Map

Create a 10 m-by-10 m empty map.

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

Update the occupancy of world locations with specific probability values and display the map.

```
x = [1.2; 2.3; 3.4; 4.5; 5.6];
```

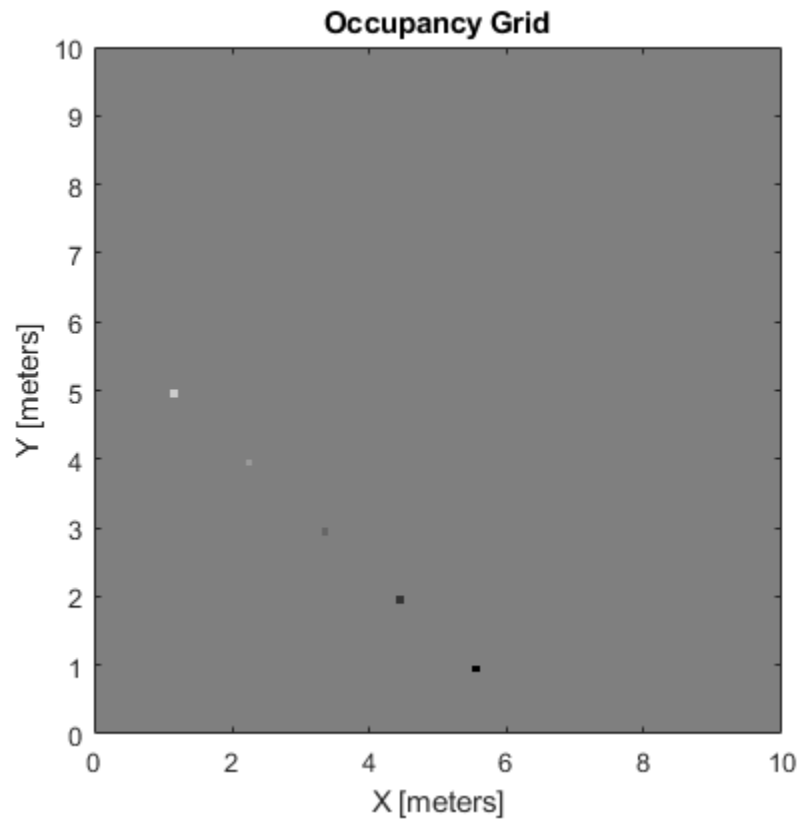
```
y = [5.0; 4.0; 3.0; 2.0; 1.0];
```

```
pvalues = [0.2 0.4 0.6 0.8 1];
```

```
updateOccupancy(map,[x y],pvalues)
```

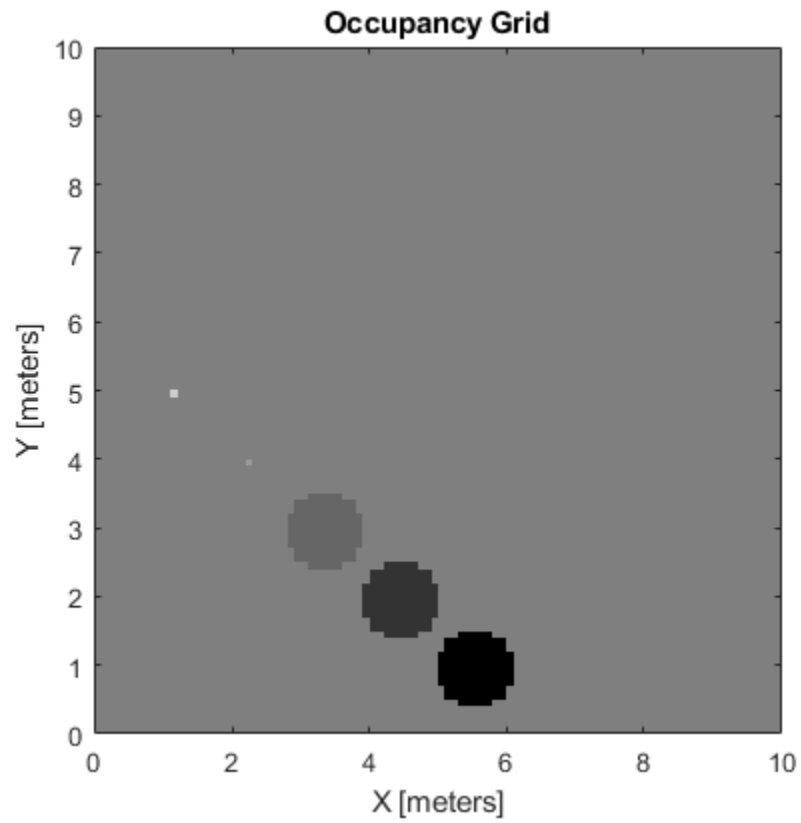
```
figure
```

```
show(map)
```

Inflate occupied areas by a radius of 0.5 m. Larger occupancy values overwrite the smaller values.

```
inflate(map,0.5)  
figure  
show(map)
```

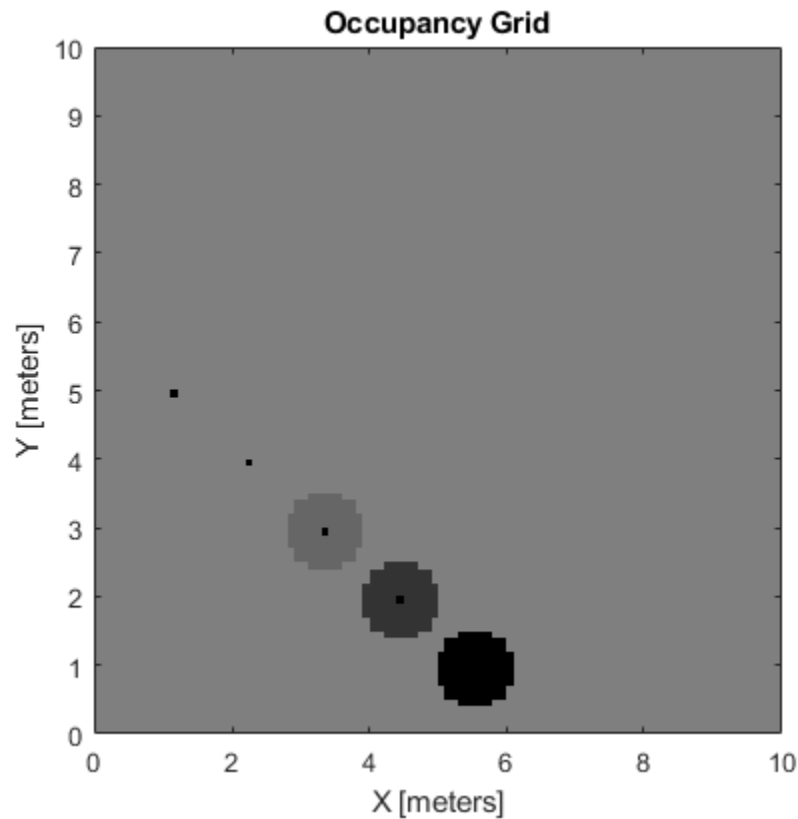


Get grid locations from world locations.

```
ij = world2grid(map,[x y]);
```

Set grid locations to occupied 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 vertical 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`

obs — Probability observation values

n -by-1 numeric column vector | n -by-1 logical column vector | numeric scalar | logical scalar

Probability observation values, specified as a scalar or an n -by-1 column vector the same size as either `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. These values correlate to the inverse sensor model for ray casting. If `obs` is a numeric or a logical scalar, the value is applied to all coordinates in `xy` or `ij`.

occval — Probability occupancy values

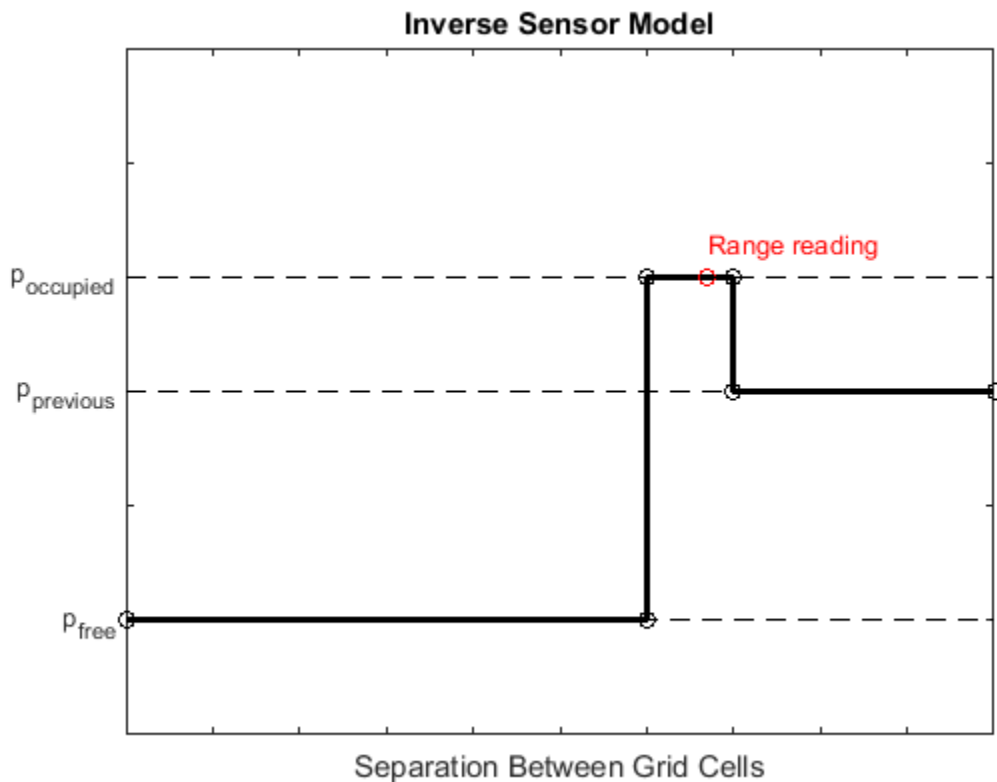
scalar | column vector

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.

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

[binaryOccupancyMap](#) | [occupancyMap](#) | [setOccupancy](#)

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 a 10 m-by-10 m empty map.

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

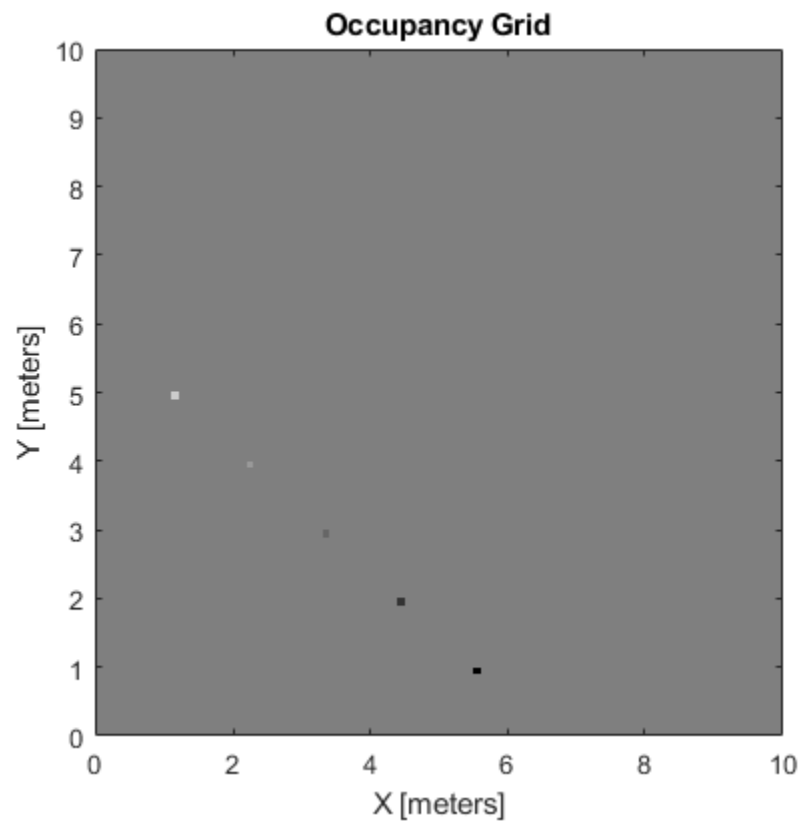
Update the occupancy of world locations with specific probability values and display the map.

```
x = [1.2; 2.3; 3.4; 4.5; 5.6];
```

```
y = [5.0; 4.0; 3.0; 2.0; 1.0];
```

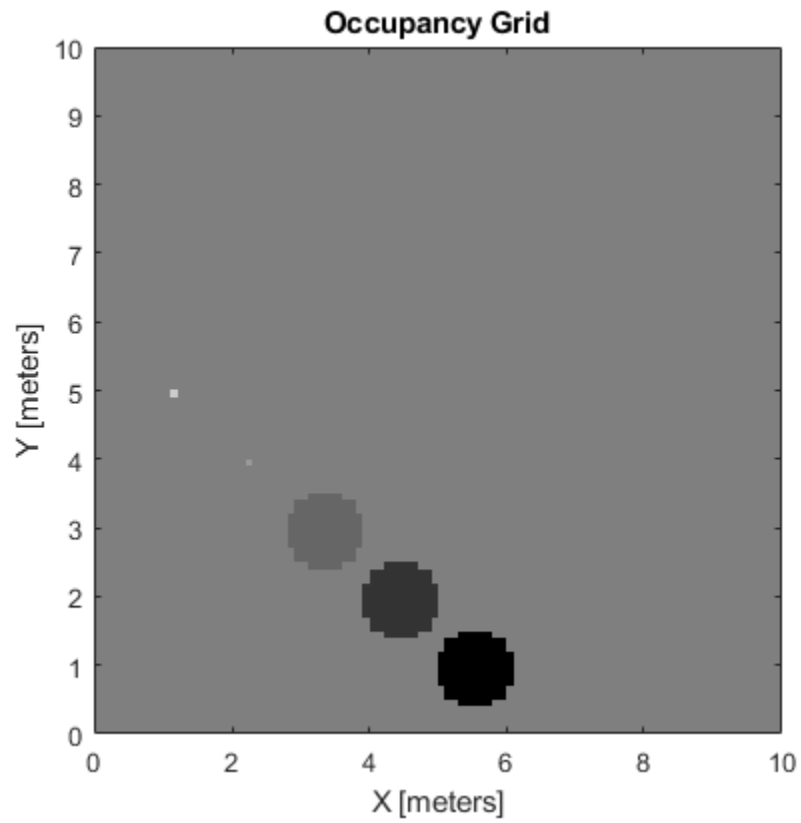
```
pvalues = [0.2 0.4 0.6 0.8 1];
```

```
updateOccupancy(map,[x y],pvalues)  
figure  
show(map)
```



Inflate occupied areas by a radius of 0.5 m. Larger occupancy values overwrite the smaller values.

```
inflate(map,0.5)  
figure  
show(map)
```

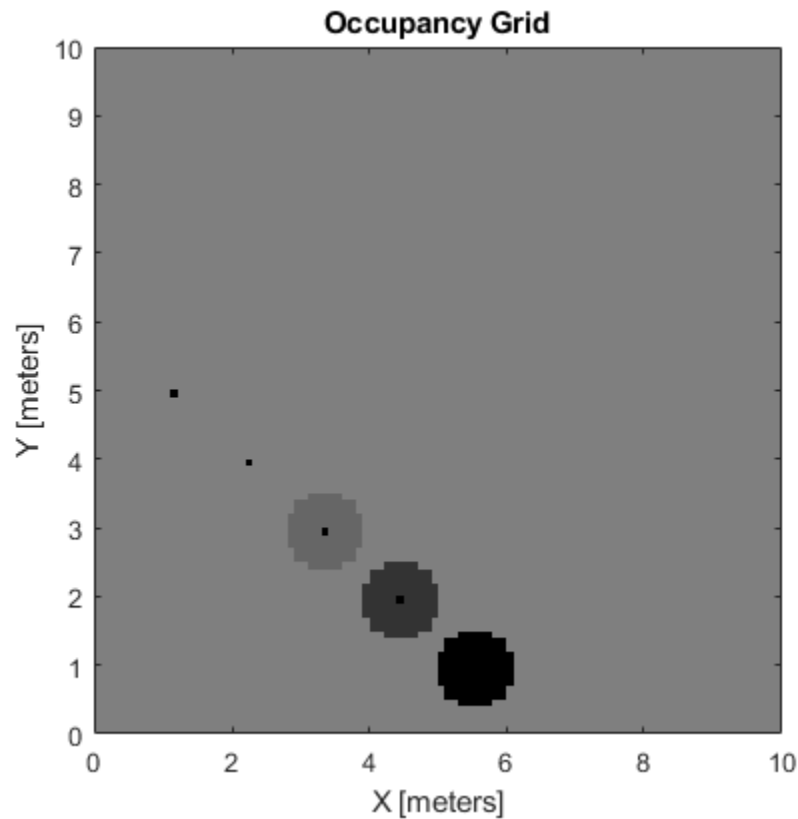


Get grid locations from world locations.

```
ij = world2grid(map,[x y]);
```

Set grid locations to occupied 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 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.

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

[binaryOccupancyMap](#) | [grid2world](#) | [occupancyMap](#)

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? \(MATLAB\)](#).

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 \(MATLAB\)](#).

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 `HorizontalAngle` 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.

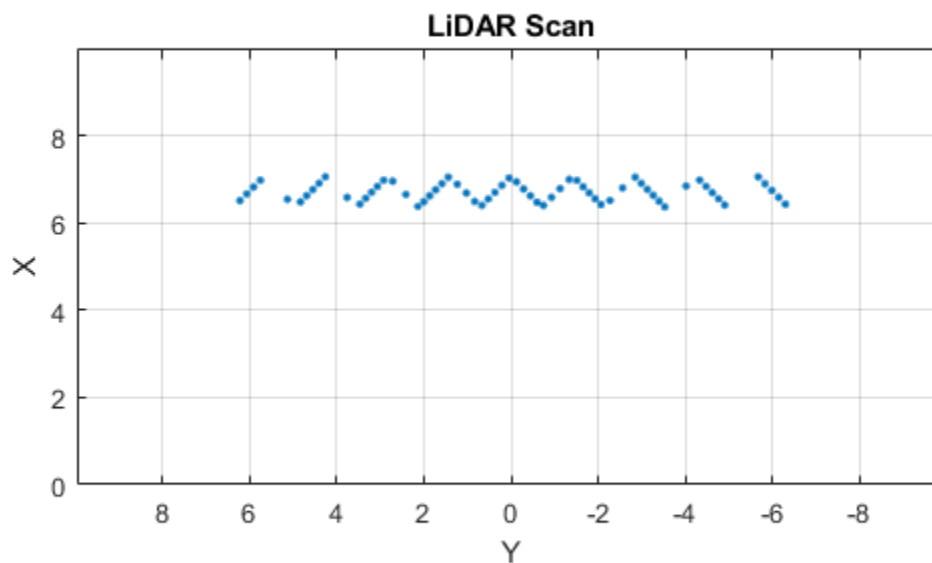
```
truePose = [0 0 pi/4];  
trueMap = binaryOccupancyMap(eye(10));
```

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

[binaryOccupancyMap](#) | [lidarScan](#) | [occupancyMap](#)

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

Map representation, specified as a `occupancyMap` 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

`binaryOccupancyMap` | `grid2world` | `local2world` | `occupancyMap`

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 2-523. 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);
```

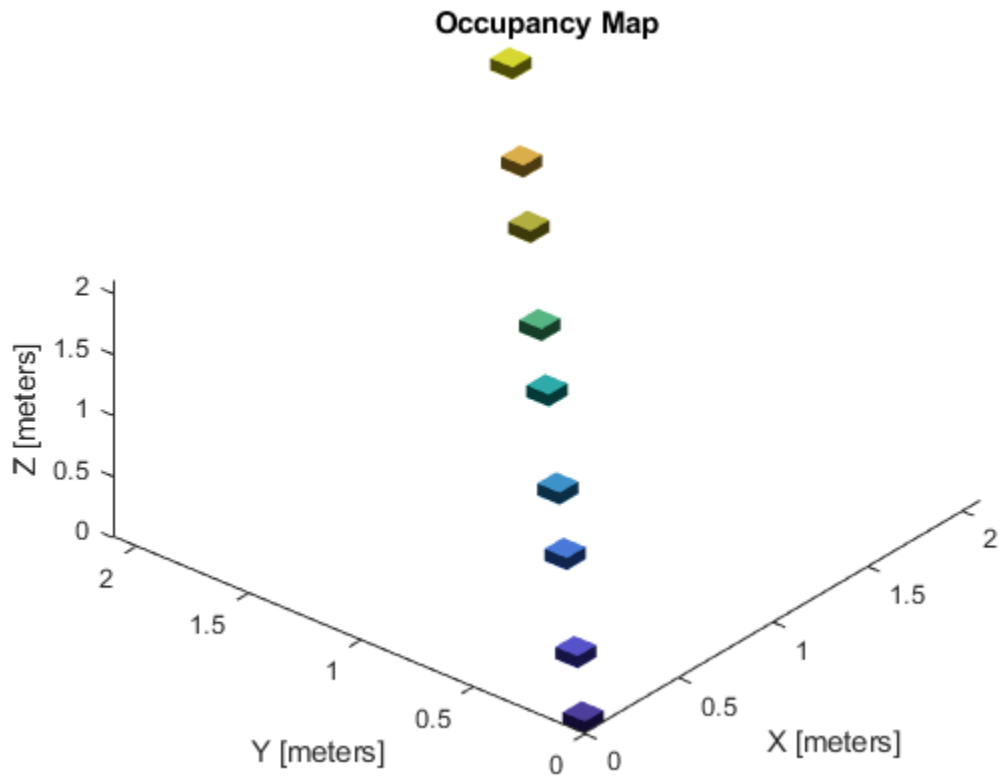
Define a set of 3-D points as an observation from a pose [x y z qw qx qy qz]. This pose is for the sensor that observes these points and is centered on the origin. Define two sets of points to insert multiple observations.

```
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;
```

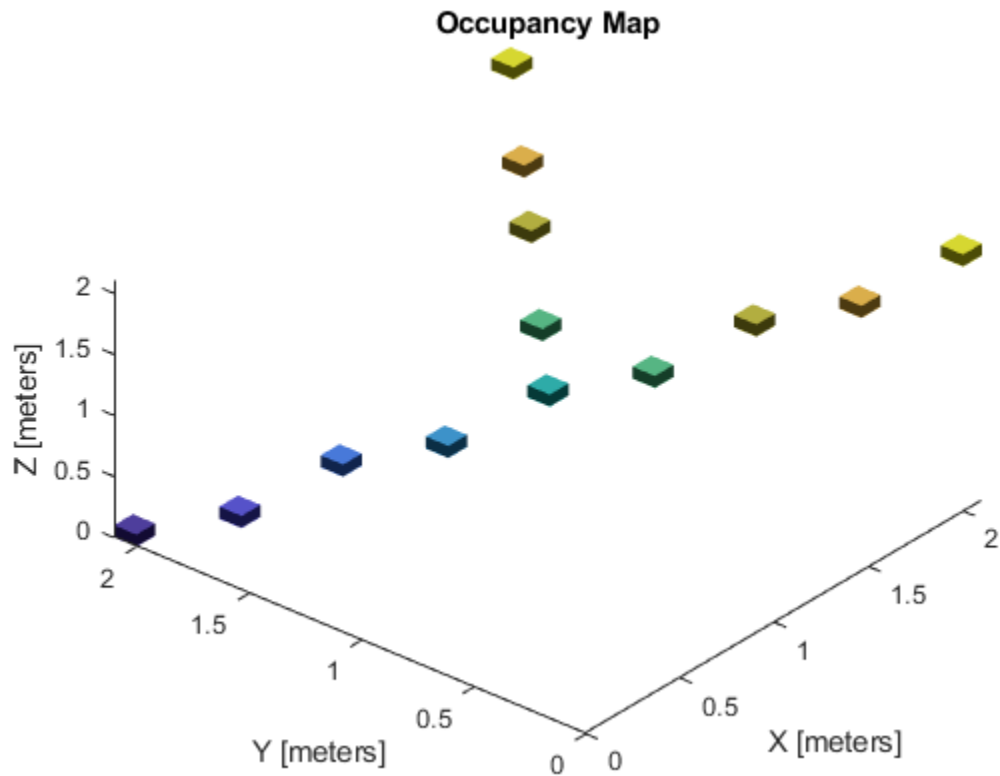
Insert the first set of points using `insertPointCloud`. The function uses the sensor pose and the given points to insert observations into the map. The colors displayed correlate to the height of the point only for illustrative purposes.

```
insertPointCloud(map3D,pose,points,maxRange)
show(map3D)
```



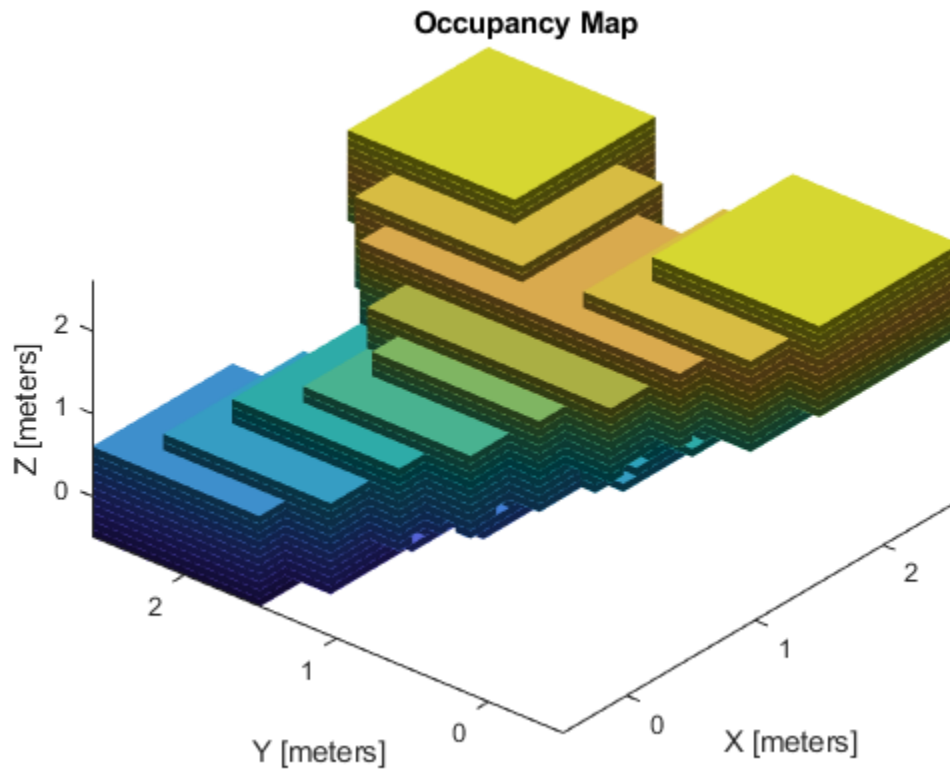
Insert the second set of points. The ray between the sensor pose (origin) and these points overlap points from the previous insertion. Therefore, the free space between the sensor and the new points are updated and marked as free space.

```
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.

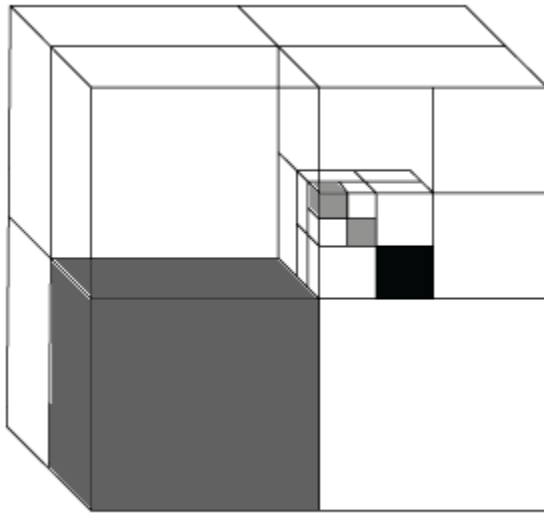
```
vehicleRadius = 0.2;  
safetyRadius = 0.3;  
inflationRadius = vehicleRadius + safetyRadius;  
inflate(map3D, inflationRadius);  
  
show(map3D)
```



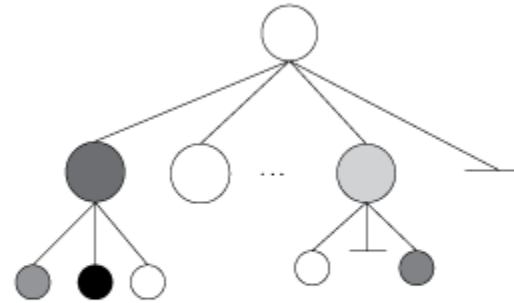
Algorithms

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.

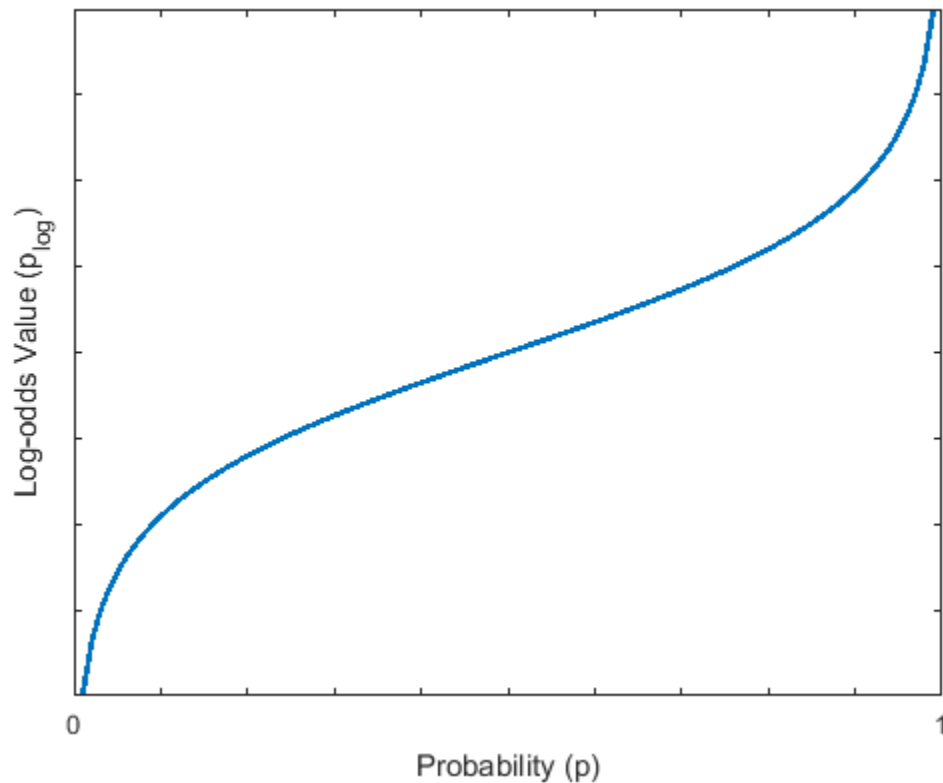


3-D Voxels



Octree Branching Structure

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.

See Also

Classes

`binaryOccupancyMap` | `occupancyMap`

Functions

`inflate` | `insertPointCloud` | `readOccupancyMap3D` | `setOccupancy` | `show`

Introduced in R2019b

checkOccupancy

Check if locations are free or occupied

Syntax

```
i0ccval = checkOccupancy(map3D, xyz)
```

Description

`i0ccval = 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).

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

i0ccval — 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

`lidarSLAM` | `occupancyMap` | `occupancyMap3D`

Functions

`inflate` | `insertPointCloud` | `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.

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

`lidarSLAM` | `occupancyMap` | `occupancyMap3D`

Functions

`inflate` | `insertPointCloud` | `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.

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

`lidarSLAM` | `occupancyMap` | `occupancyMap3D`

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.

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

lidarSLAM | occupancyMap | occupancyMap3D

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.

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**

`lidarSLAM` | `occupancyMap` | `occupancyMap3D`

Functions

`inflate` | `insertPointCloud` | `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.

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

`lidarSLAM` | `occupancyMap` | `occupancyMap3D`

Functions

`inflate` | `insertPointCloud` | `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`.

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

`lidarSLAM` | `occupancyMap` | `occupancyMap3D`

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`.

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

`lidarSLAM` | `occupancyMap` | `occupancyMap3D`

Functions

`inflate` | `insertPointCloud` | `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

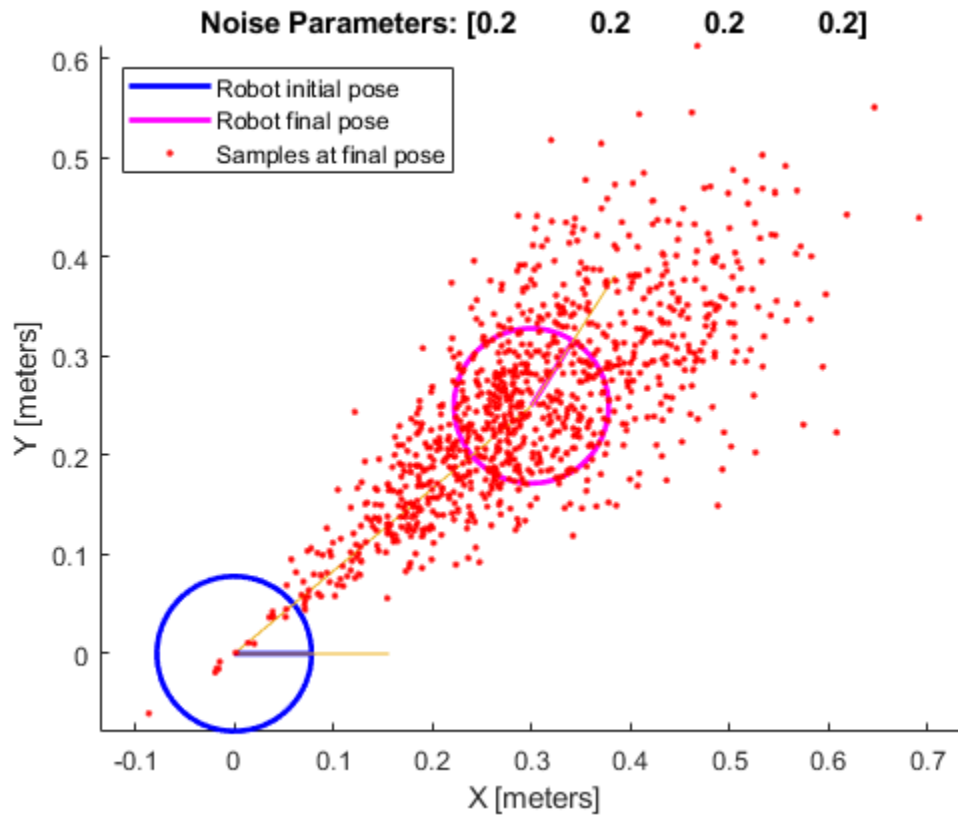
This example shows how to visualize the effect of different noise parameters on the `odometryMotionModel` class. An `odometryMotionModel` object contains the motion model noise parameters for a differential drive vehicle. Use `showNoiseDistribution` to visualize how changing these values affect the distribution of predicted poses.

Create a motion model object.

```
motionModel = odometryMotionModel;
```

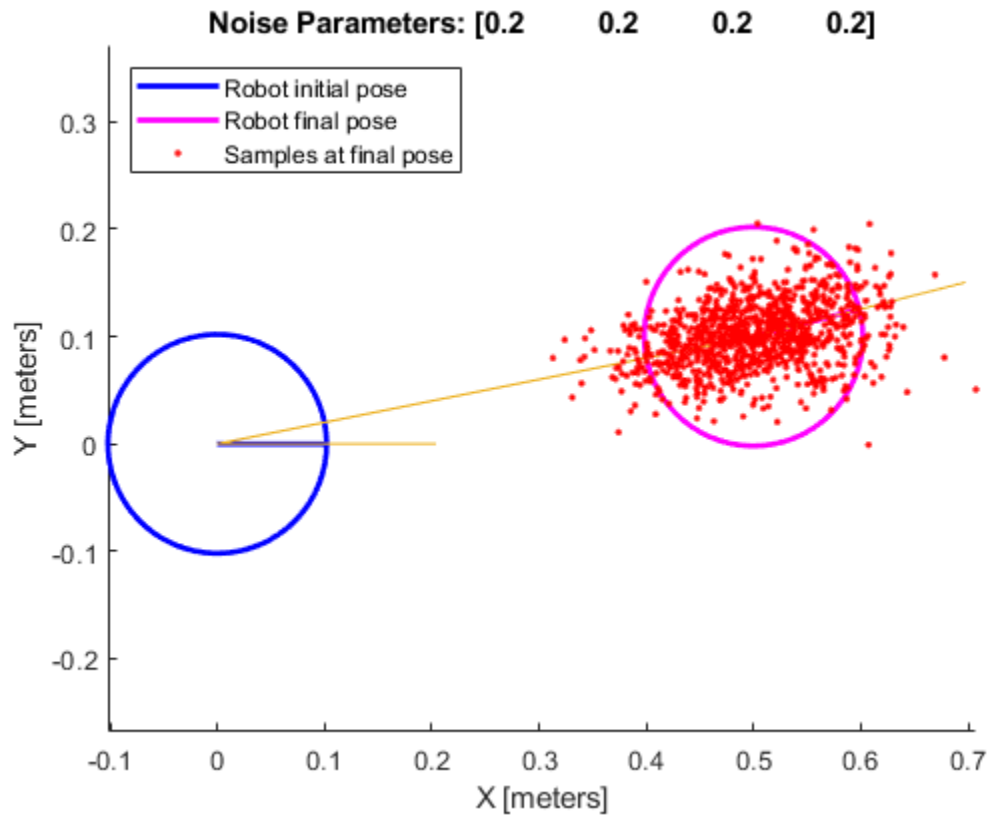
Show the distribution of particles with the existing noise parameters. Each particle is a hypothesis for the predicted pose.

```
showNoiseDistribution(motionModel);
```



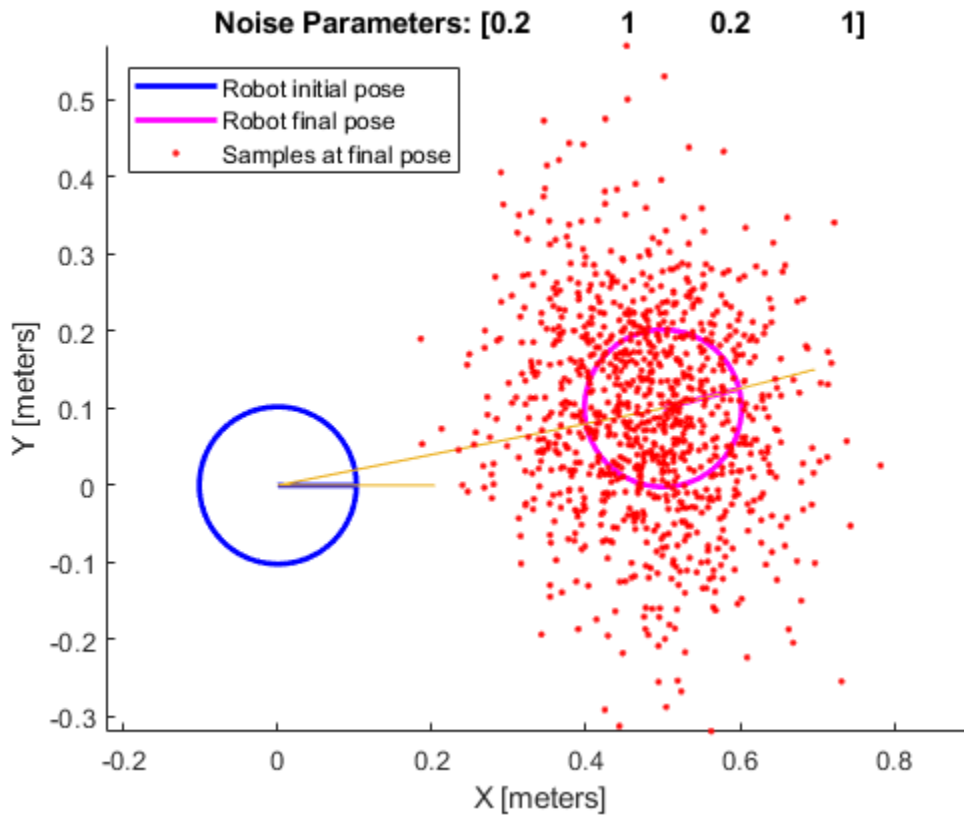
Show the distribution with a specified odometry pose change and number of samples. The change in odometry is used as the final pose with hypotheses distributed around based on the Noise parameters.

```
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(...);
release(mcl)
mcl.MotionModel.PropName = value;
```

References

[1] Thrun, Sebastian, 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

`likelihoodFieldSensor` | `monteCarloLocalization`

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

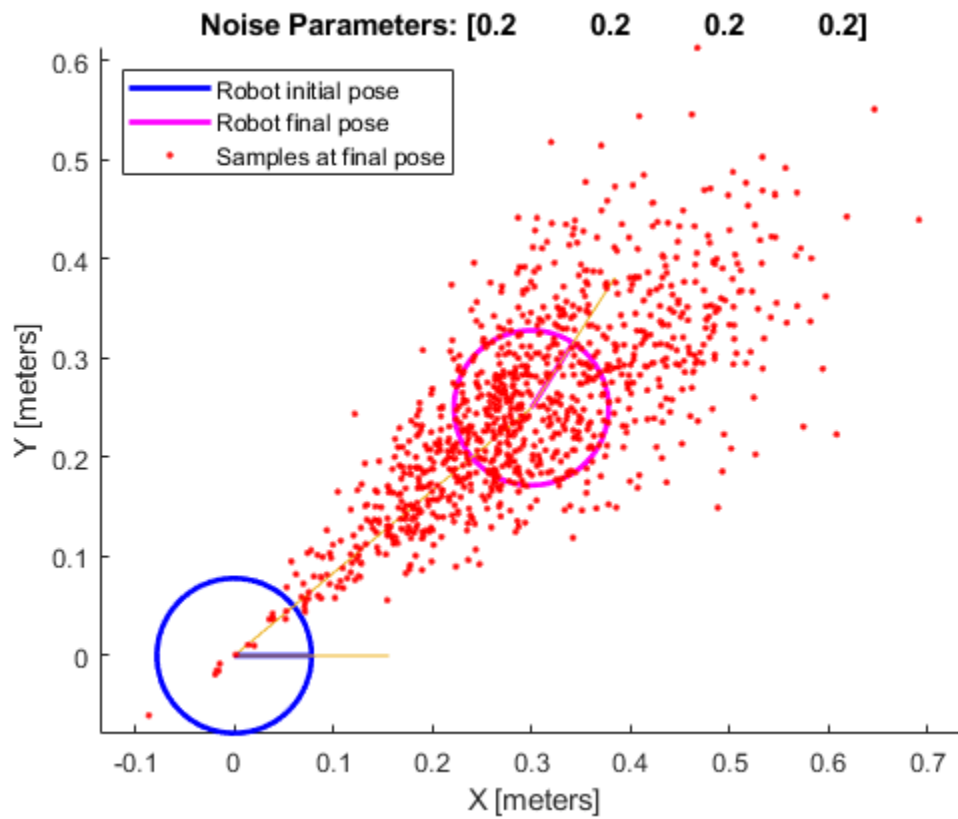
This example shows how to visualize the effect of different noise parameters on the `odometryMotionModel` class. An `odometryMotionModel` object contains the motion model noise parameters for a differential drive vehicle. Use `showNoiseDistribution` to visualize how changing these values affect the distribution of predicted poses.

Create a motion model object.

```
motionModel = odometryMotionModel;
```

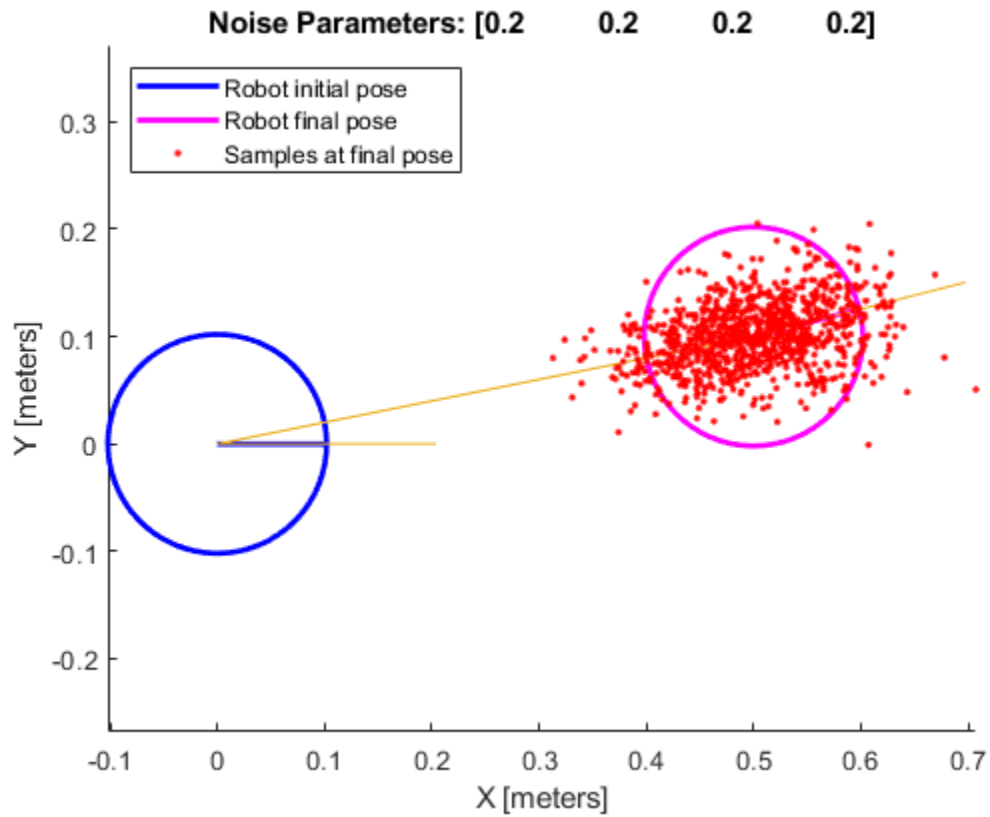
Show the distribution of particles with the existing noise parameters. Each particle is a hypothesis for the predicted pose.

```
showNoiseDistribution(motionModel);
```

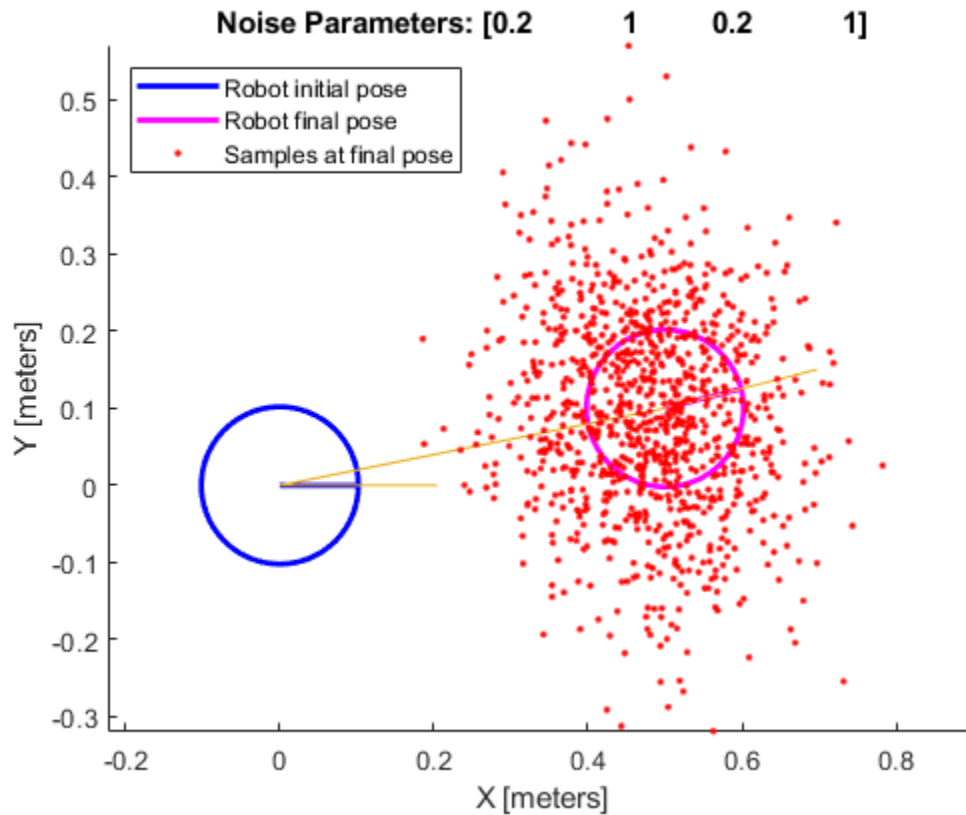
Show the distribution with a specified odometry pose change and number of samples. The change in odometry is used as the final pose with hypotheses distributed around based on the Noise parameters.

```
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 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

`likelihoodFieldSensor` | `monteCarloLocalization` | `odometryMotionModel`

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

<code>clearance</code>	Minimum clearance of path
<code>isPathValid</code>	Determine if planned path is obstacle free
<code>show</code>	Visualize path metrics in map environment
<code>smoothness</code>	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 θ 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 = 0.0257
```

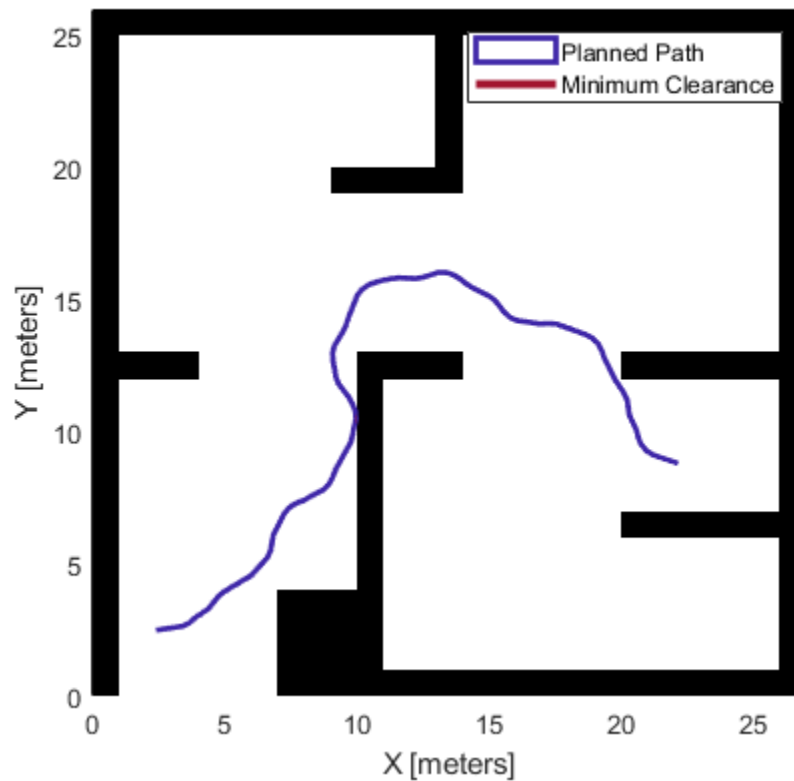
Evaluate the smoothness of the path. Values close to 0 indicate a smoother path. Straight-line paths return a value of 0.

```
smoothness(pathMetricsObj)
```

```
ans = 1.7453
```

Visualize the minimum clearance of the path.

```
show(pathMetricsObj)  
legend('Planned Path','Minimum Clearance')
```



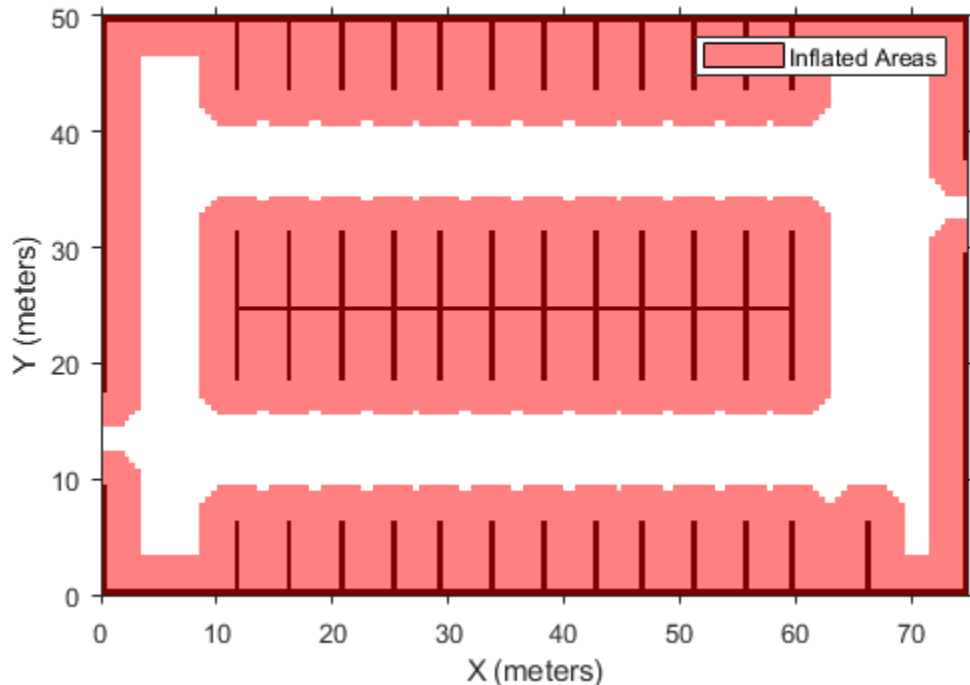
Vehicle Path Planning and Metrics Computation in a 2-D Costmap Environment

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 turning 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 $[x, y, \theta]$ 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` object and the `plan` 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)
```

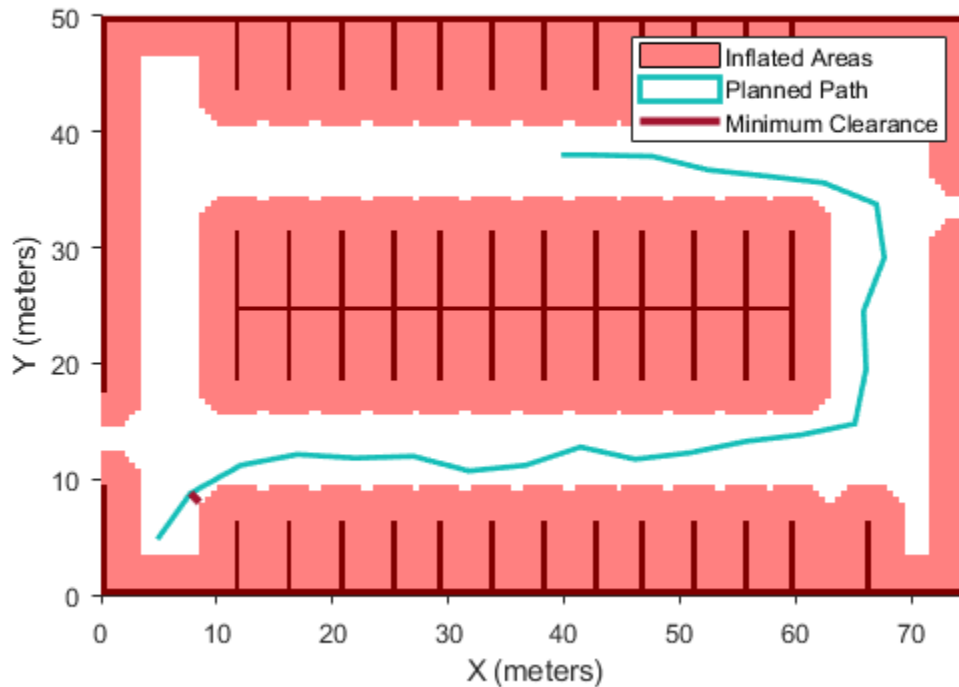
```
ans = logical
      1
```

Compute and visualize the minimum clearance of the path.

```
clearance(pathMetricsObj)
```

```
ans = 1.1014
```

```
show(pathMetricsObj)
legend('Inflated Areas','Planned Path','Minimum Clearance')
xlabel('X (meters)')
ylabel('Y (meters)')
```

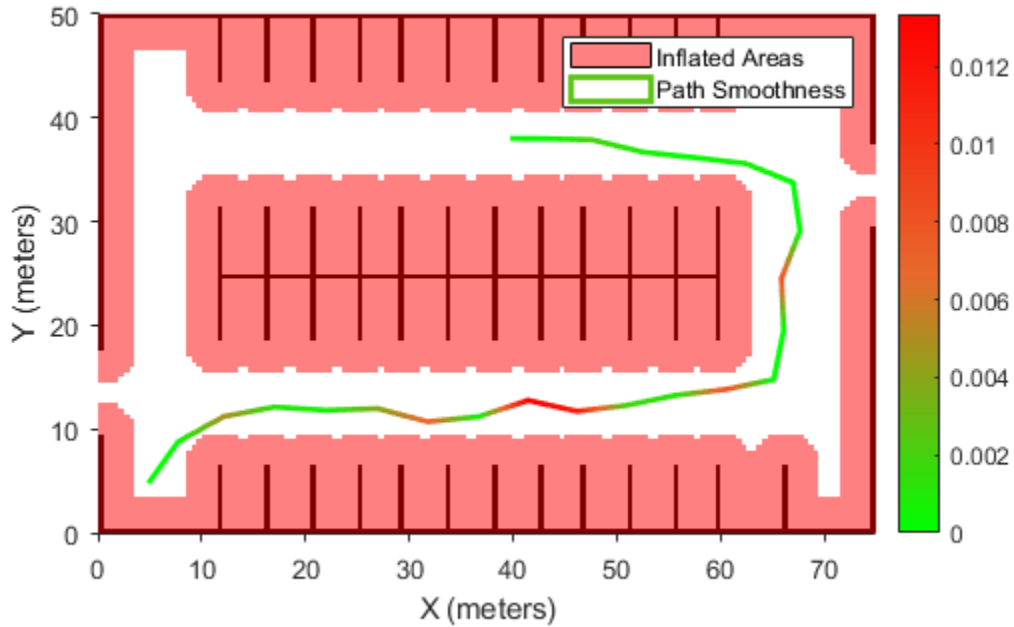


Compute and visualize the smoothness of the path. Values close to 0 indicate a smoother path. Straight-line paths return a value of 0.

```
smoothness(pathMetricsObj)
```

```
ans = 0.0658
```

```
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)')
```


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 poses on the path and obstacles in the specified map environment.

`clearance(pathMetricsObj, 'Type', 'states')` returns the set of minimum distances for each state of the path, in the form of an $(n-1)$ -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 θ 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 = 0.0257
```

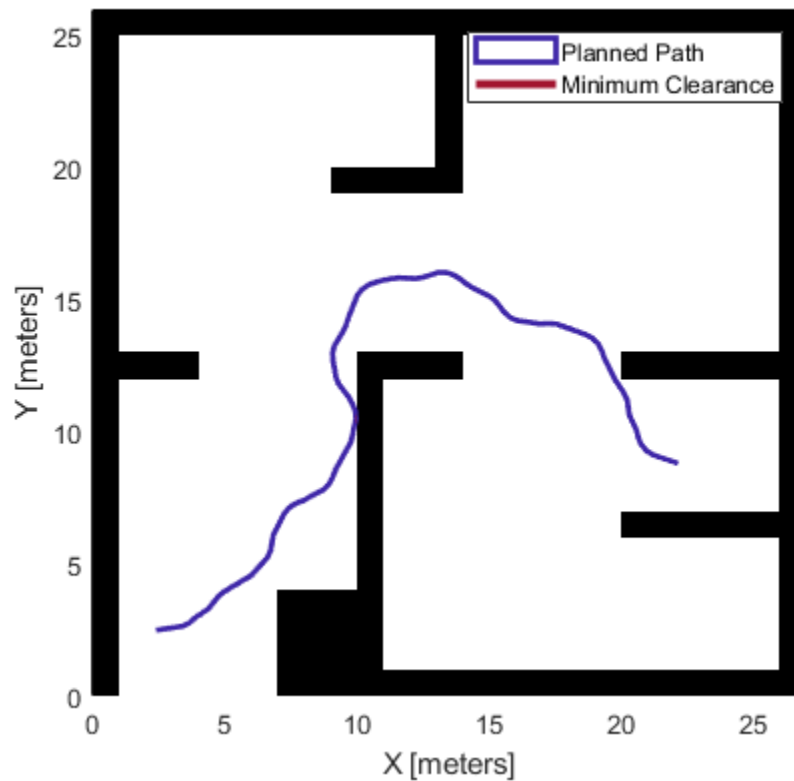
Evaluate the smoothness of the path. Values close to 0 indicate a smoother path. Straight-line paths return a value of 0.

```
smoothness(pathMetricsObj)
```

```
ans = 1.7453
```

Visualize the minimum clearance of the path.

```
show(pathMetricsObj)
legend('Planned Path', 'Minimum Clearance')
```



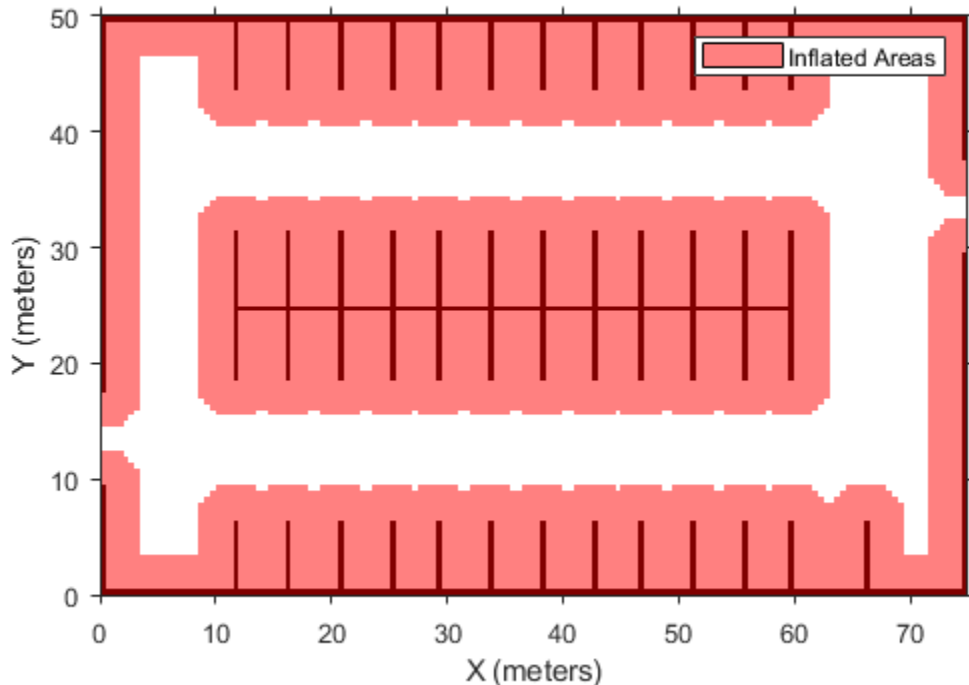
Vehicle Path Planning and Metrics Computation in a 2-D Costmap Environment

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 turning 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 $[x, y, \theta]$ 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` object and the `plan` 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)
```

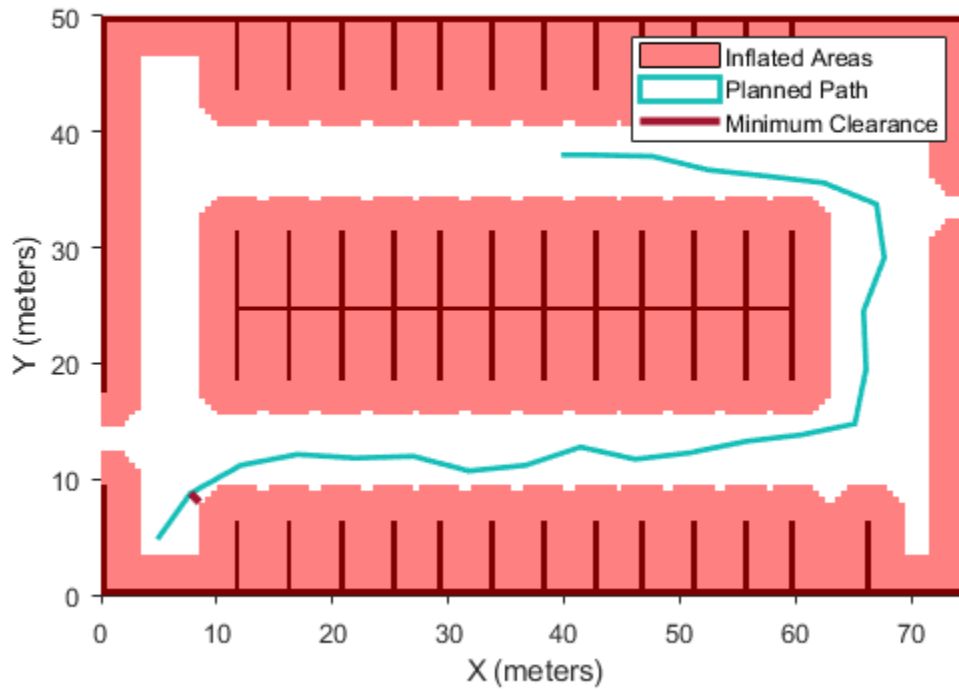
```
ans = logical
     1
```

Compute and visualize the minimum clearance of the path.

```
clearance(pathMetricsObj)
```

```
ans = 1.1014
```

```
show(pathMetricsObj)
legend('Inflated Areas','Planned Path','Minimum Clearance')
xlabel('X (meters)')
ylabel('Y (meters)')
```

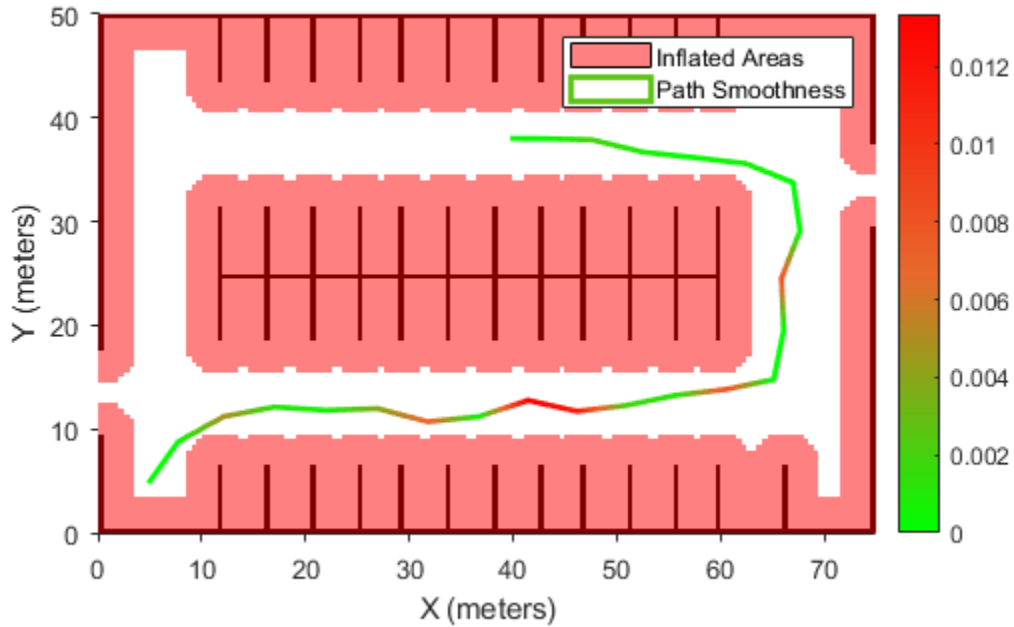


Compute and visualize the smoothness of the path. Values close to 0 indicate a smoother path. Straight-line paths return a value of 0.

```
smoothness(pathMetricsObj)
```

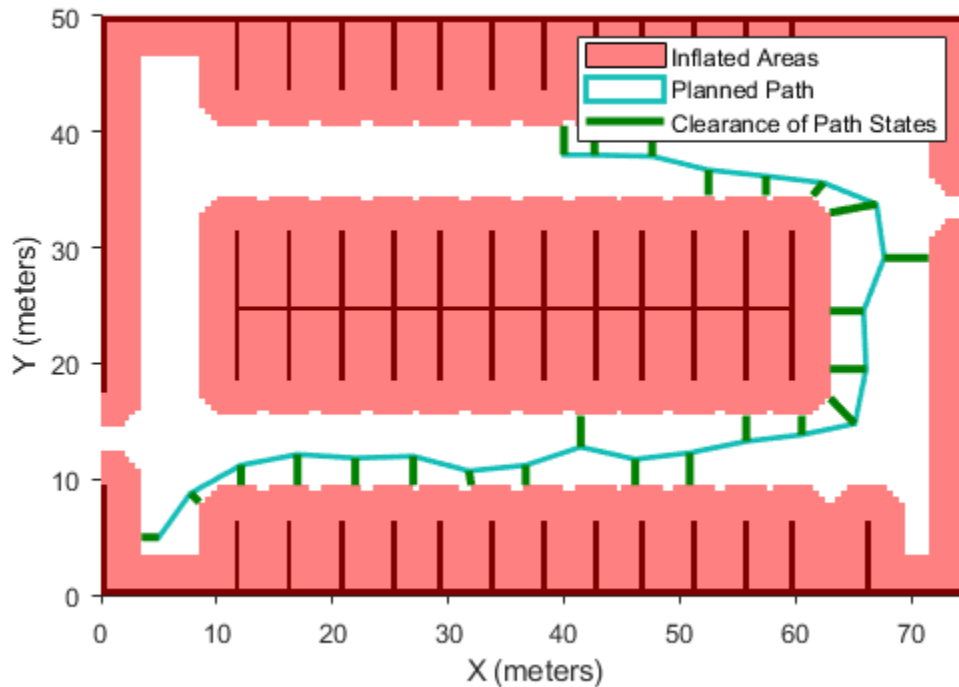
```
ans = 0.0658
```

```
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 θ 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 = 0.0257
```

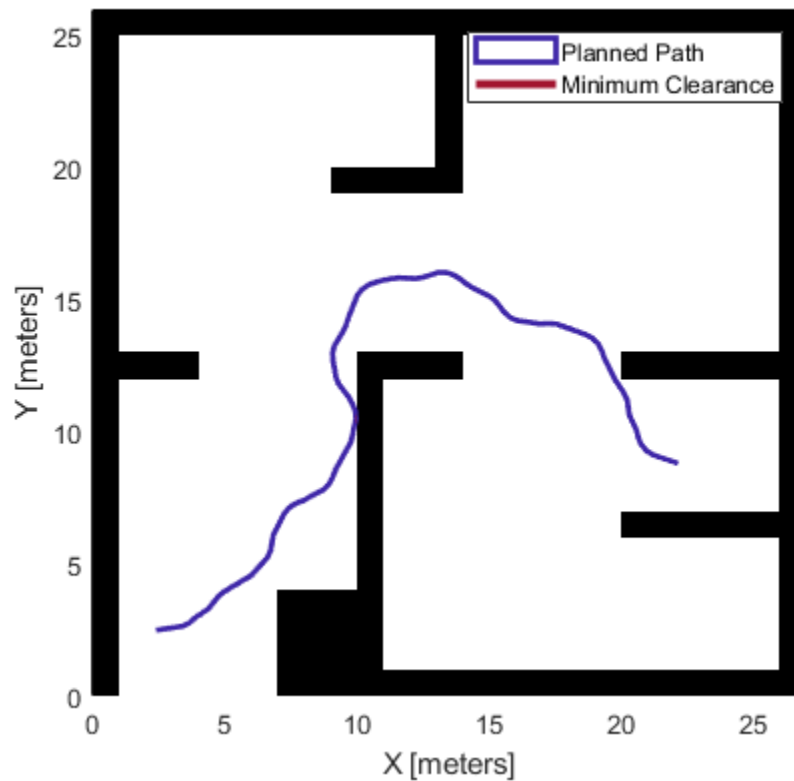
Evaluate the smoothness of the path. Values close to 0 indicate a smoother path. Straight-line paths return a value of 0.

```
smoothness(pathMetricsObj)
```

```
ans = 1.7453
```

Visualize the minimum clearance of the path.

```
show(pathMetricsObj)
legend('Planned Path','Minimum Clearance')
```



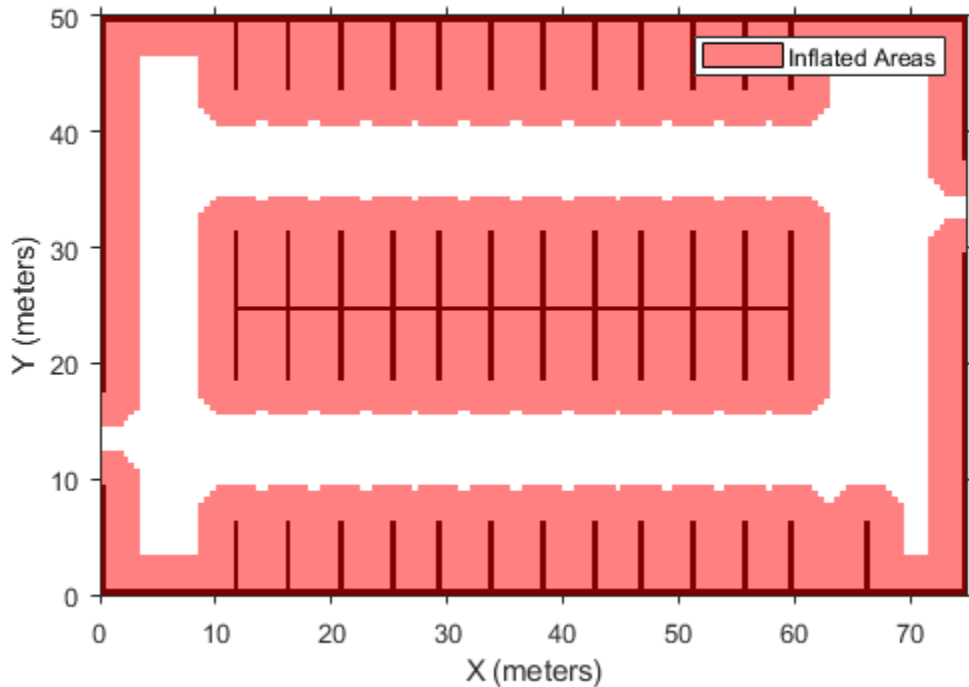
Vehicle Path Planning and Metrics Computation in a 2-D Costmap Environment

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 turning 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 $[x, y, \theta]$ 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` object and the `plan` 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)
```

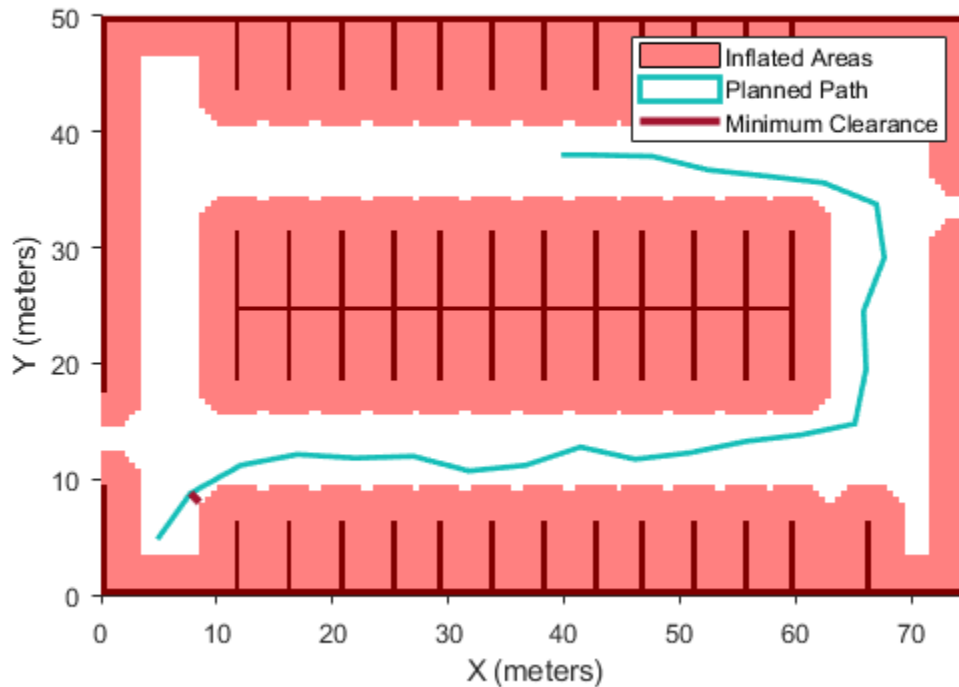
```
ans = logical
      1
```

Compute and visualize the minimum clearance of the path.

```
clearance(pathMetricsObj)
```

```
ans = 1.1014
```

```
show(pathMetricsObj)
legend('Inflated Areas','Planned Path','Minimum Clearance')
xlabel('X (meters)')
ylabel('Y (meters)')
```

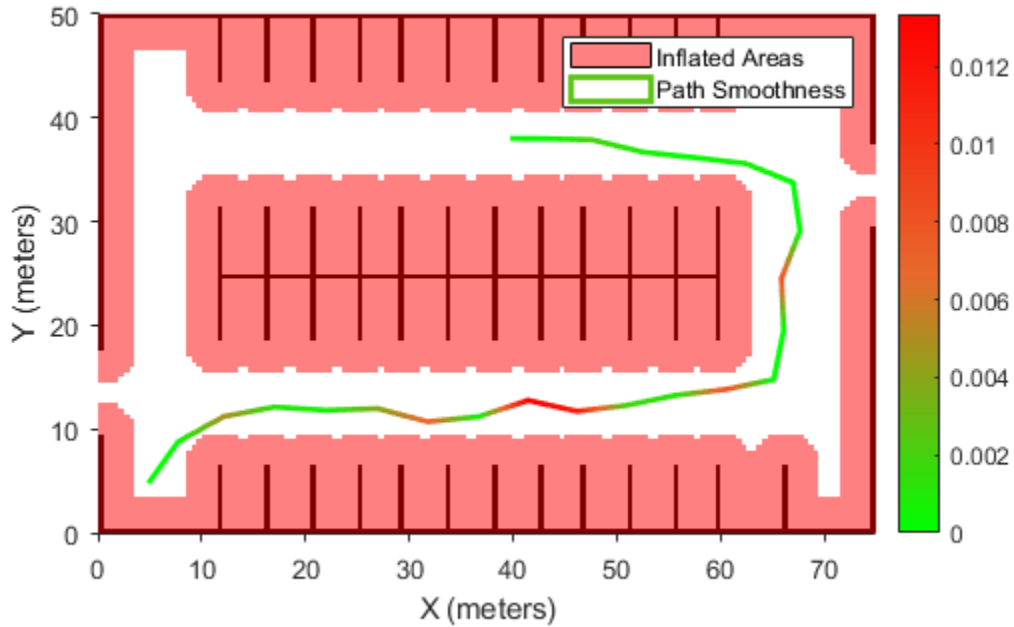


Compute and visualize the smoothness of the path. Values close to 0 indicate a smoother path. Straight-line paths return a value of 0.

```
smoothness(pathMetricsObj)
```

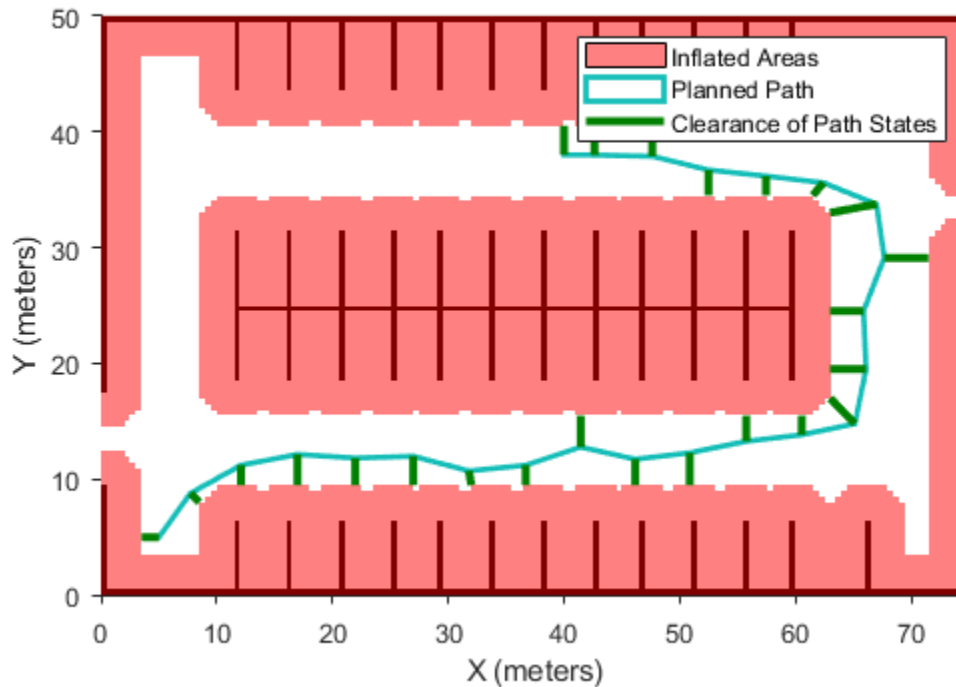
```
ans = 0.0658
```

```
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 θ 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 = 0.0257
```

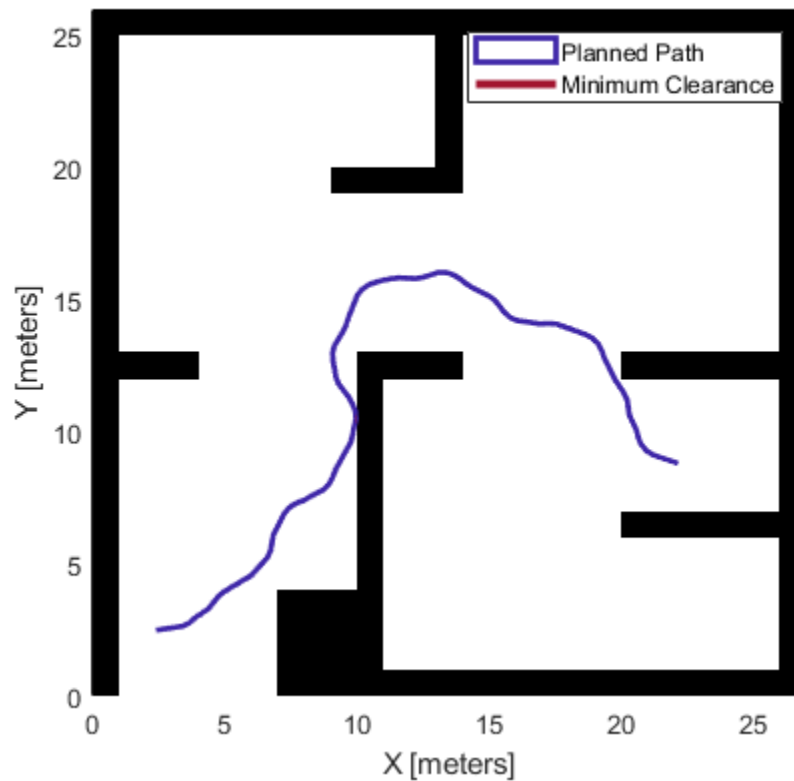
Evaluate the smoothness of the path. Values close to 0 indicate a smoother path. Straight-line paths return a value of 0.

```
smoothness(pathMetricsObj)
```

```
ans = 1.7453
```

Visualize the minimum clearance of the path.

```
show(pathMetricsObj)
legend('Planned Path','Minimum Clearance')
```



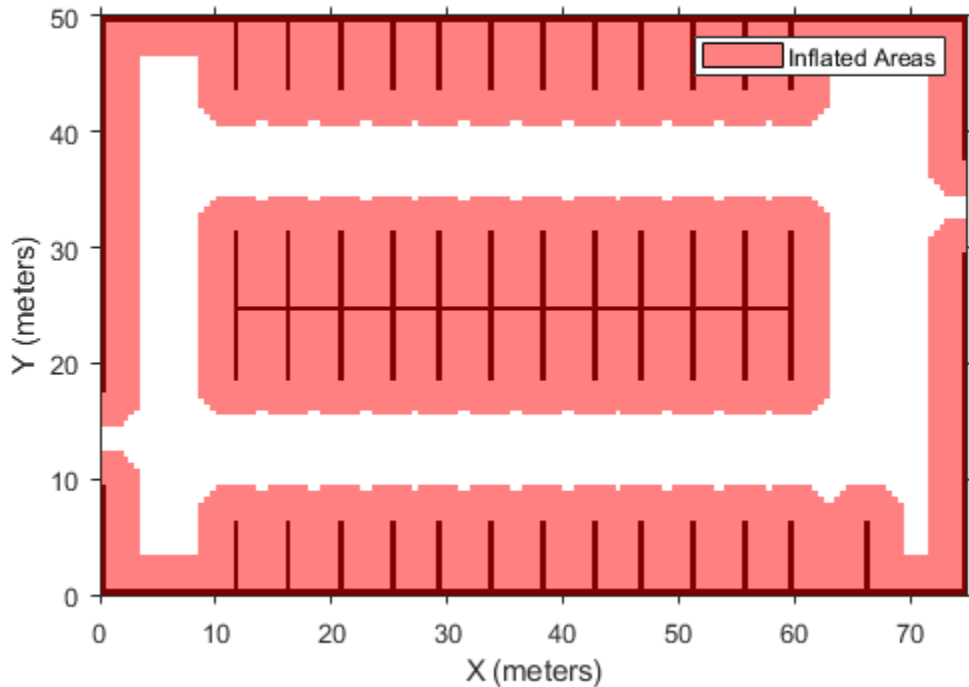
Vehicle Path Planning and Metrics Computation in a 2-D Costmap Environment

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 turning 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 $[x, y, \theta]$ 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` object and the `plan` 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)
```

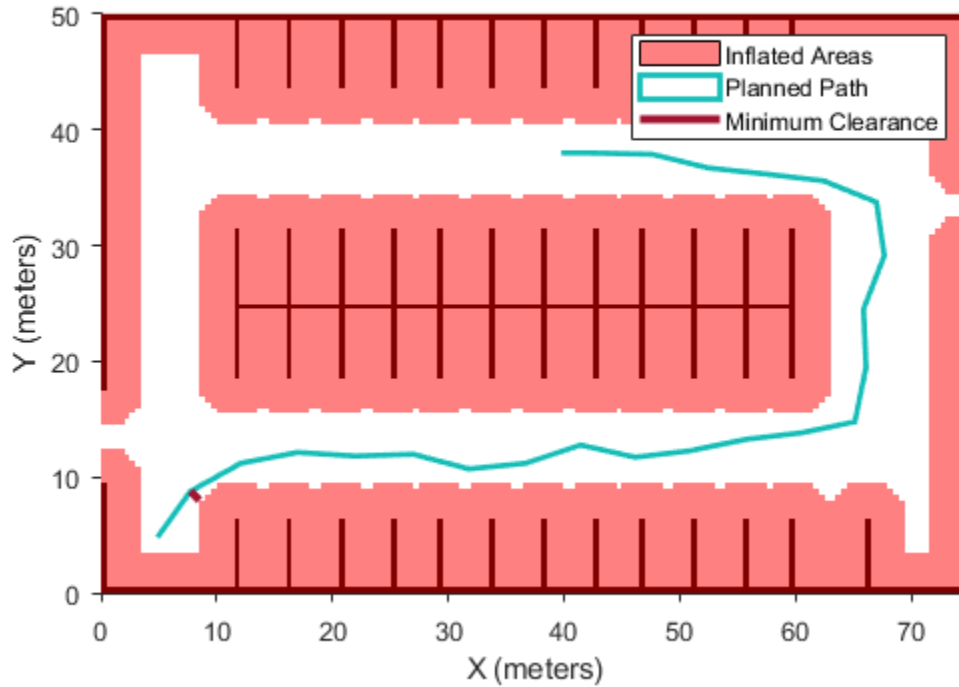
```
ans = logical
      1
```

Compute and visualize the minimum clearance of the path.

```
clearance(pathMetricsObj)
```

```
ans = 1.1014
```

```
show(pathMetricsObj)
legend('Inflated Areas','Planned Path','Minimum Clearance')
xlabel('X (meters)')
ylabel('Y (meters)')
```

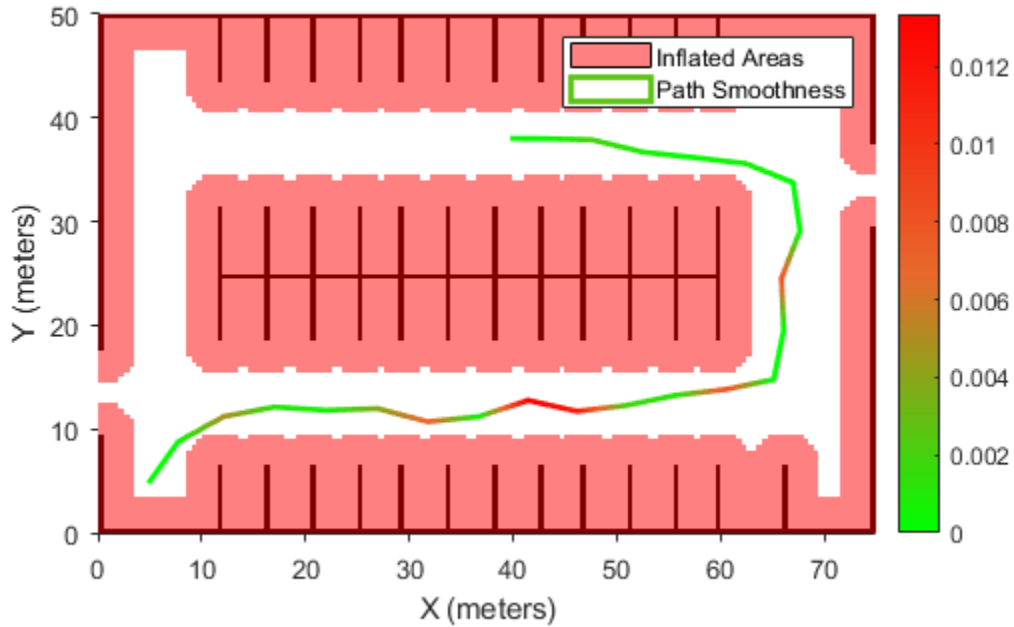


Compute and visualize the smoothness of the path. Values close to 0 indicate a smoother path. Straight-line paths return a value of 0.

```
smoothness(pathMetricsObj)
```

```
ans = 0.0658
```

```
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)')
```


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 θ 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 = 0.0257
```

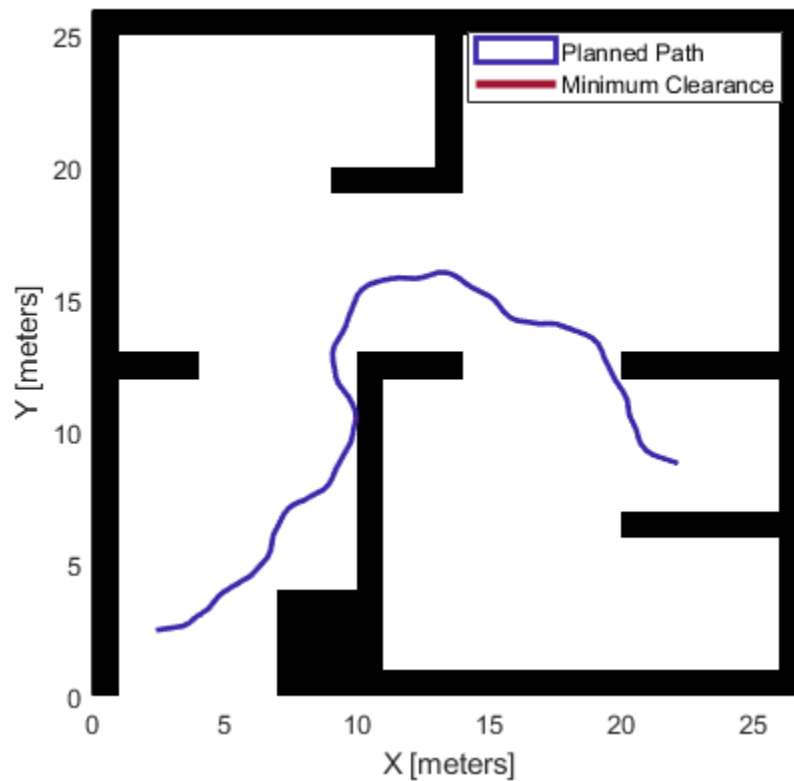
Evaluate the smoothness of the path. Values close to 0 indicate a smoother path. Straight-line paths return a value of 0.

```
smoothness(pathMetricsObj)
```

```
ans = 1.7453
```

Visualize the minimum clearance of the path.

```
show(pathMetricsObj)  
legend('Planned Path','Minimum Clearance')
```

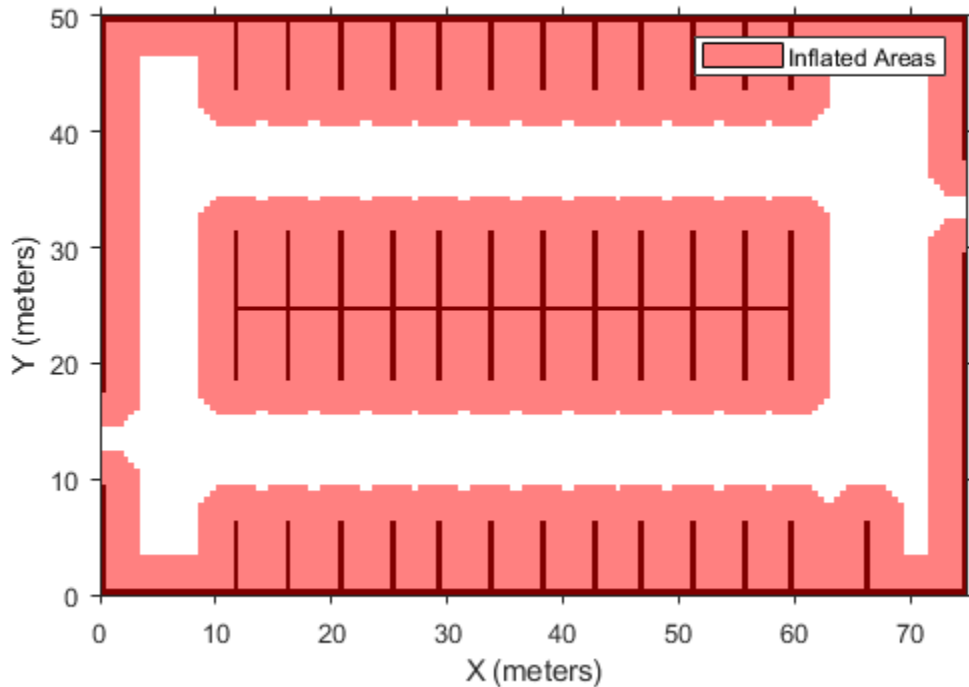
Vehicle Path Planning and Metrics Computation in a 2-D Costmap Environment

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 turning 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 $[x, y, \theta]$ 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` object and the `plan` 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)
```

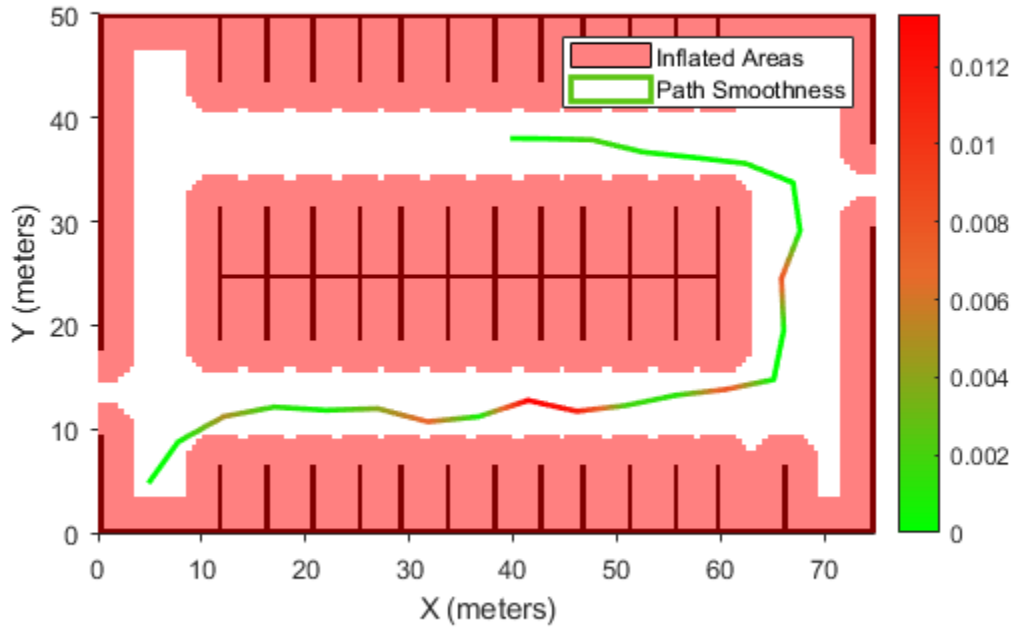
```
ans = logical
      1
```

Compute and visualize the minimum clearance of the path.

```
clearance(pathMetricsObj)
```

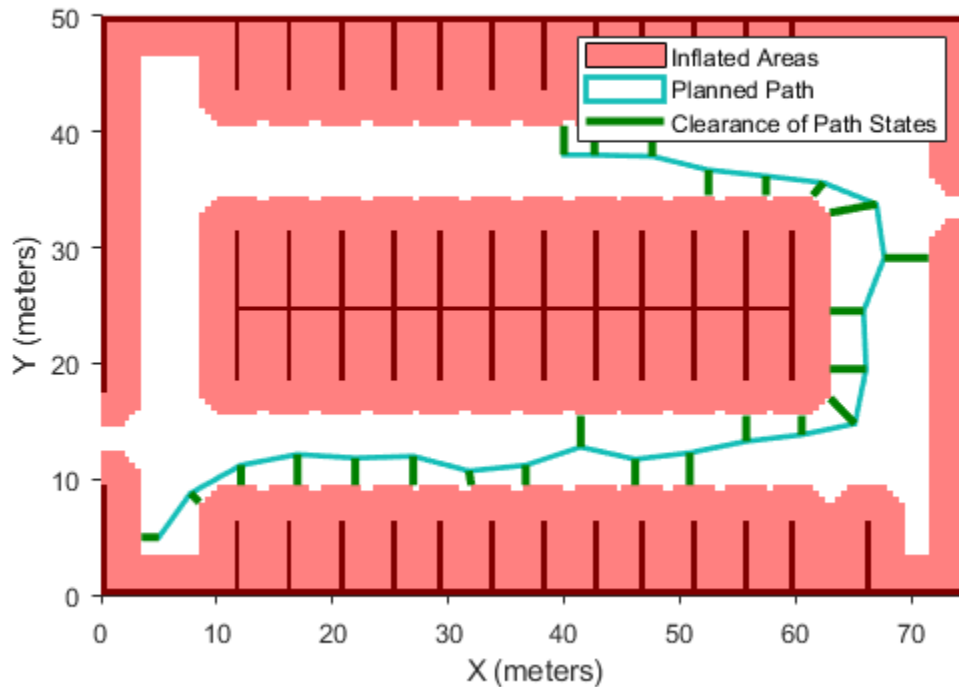
```
ans = 1.1014
```

```
show(pathMetricsObj)
legend('Inflated Areas','Planned Path','Minimum Clearance')
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

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 2-589 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

Number of motion primitives to be generated, specified as the comma-separated pair consisting of 'NumMotionPrimitives' and a positive odd integer scalar.

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 θ specifies the orientation angle in radians.

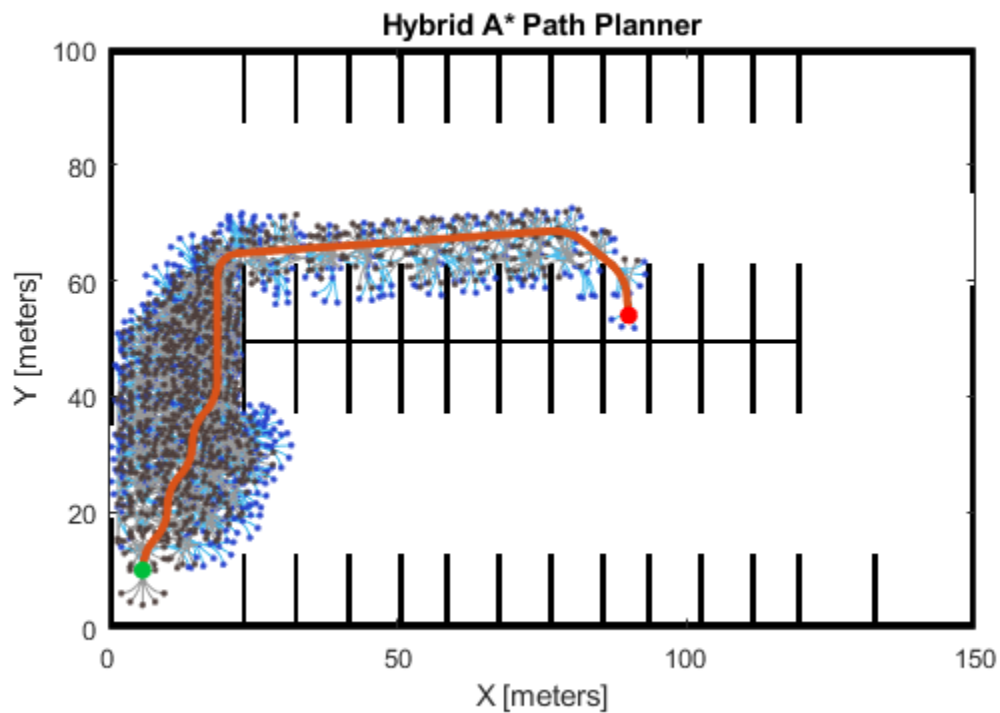
```
startPose = [6 10 pi/2]; % [meters, meters, radians]
goalPose = [90 54 -pi/2];
```

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

[navPath](#) | [validatorOccupancyMap](#) | [validatorVehicleCostmap](#)

Introduced in R2019b

plan

Find obstacle-free path between two poses

Syntax

```
path = 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.

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 θ specifies the orientation angle in radians.

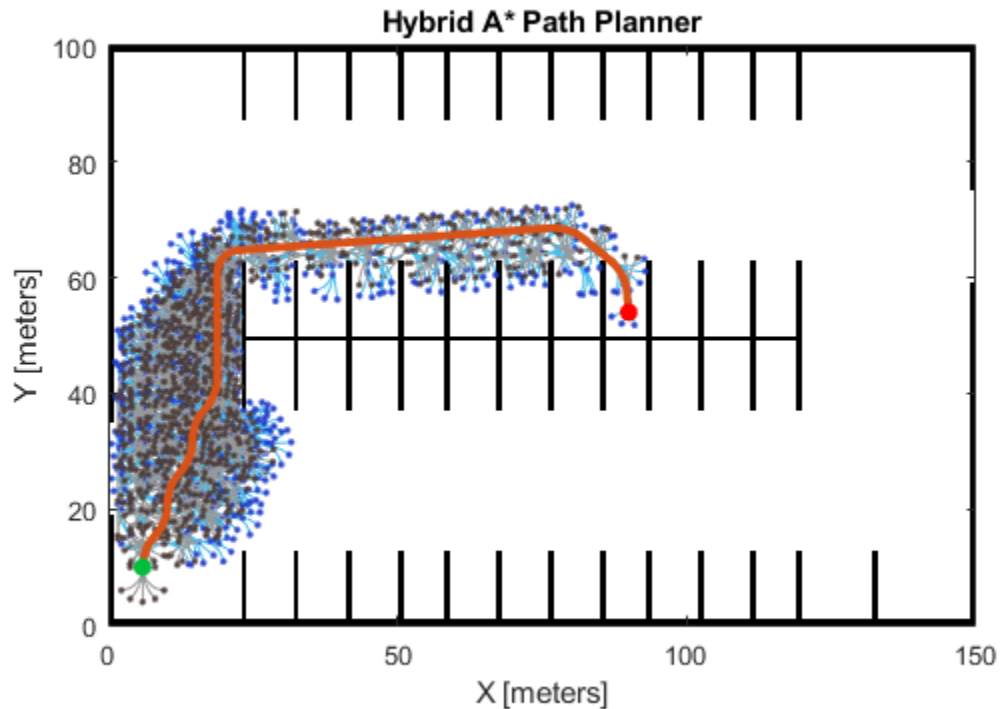
```
startPose = [6 10 pi/2]; % [meters, meters, radians]  
goalPose = [90 54 -pi/2];
```

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 θ 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 θ 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.

Extended Capabilities

C/C++ Code Generation

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

See Also

navPath | validatorOccupancyMap | validatorVehicleCostmap

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 θ specifies the orientation angle in radians.

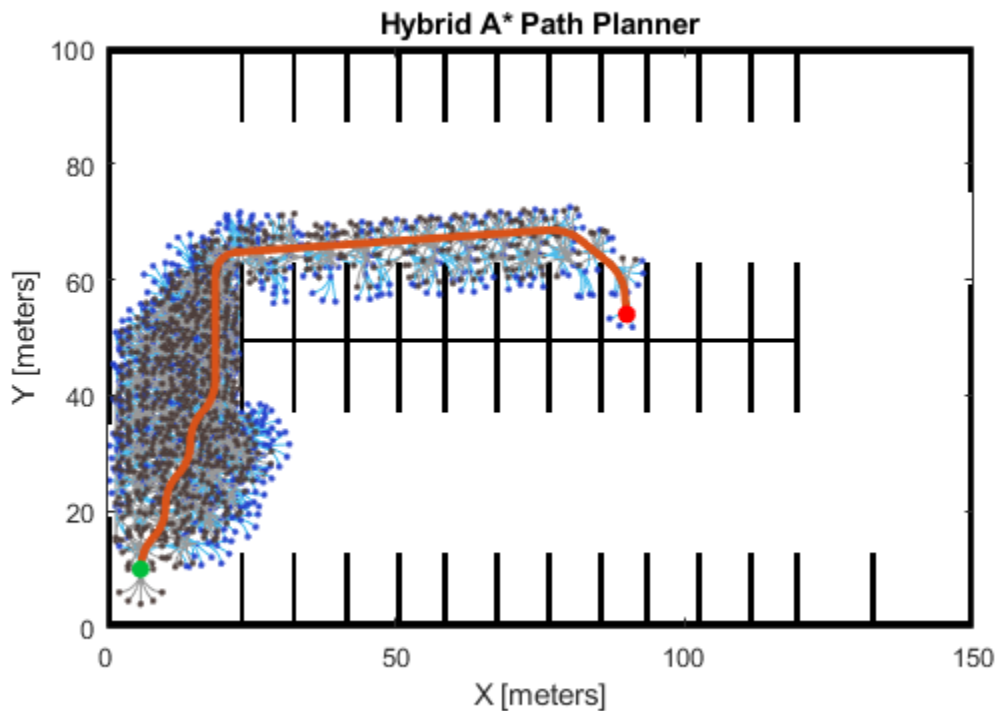
```
startPose = [6 10 pi/2]; % [meters, meters, radians]
goalPose = [90 54 -pi/2];
```

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

`navPath` | `validatorOccupancyMap` | `validatorVehicleCostmap`

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.

Data Types: 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`.

Data Types: `object`

MaxNumTreeNodees — Maximum number of nodes in the search tree

1e4 (default) | positive integer

Maximum number of nodes in the search tree, 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`.

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 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.

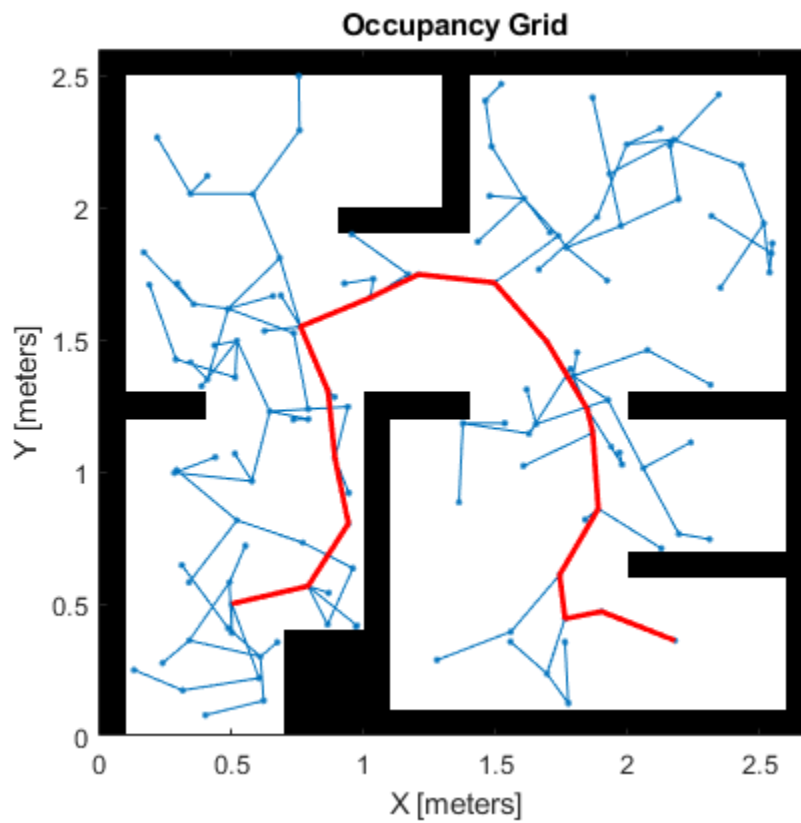
```
start = [0.5,0.5,0];  
goal = [2.5,0.2,0];
```

Plan a path with default settings.

```
rng(100,'twister'); % 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
```



Extended Capabilities

C/C++ Code Generation

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

See Also

[navPath](#) | [plannerRRTStar](#) | [stateSpaceDubins](#) | [stateSpaceReedsShepp](#) | [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

BallRadiusContant — 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 d th dimension

Data Types: object

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.

Data Types: `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`.

Data Types: `object`

MaxNumTreeNodes — Maximum number of nodes in the search tree

1e4 (default) | positive integer

Maximum number of nodes in the search tree, 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`.

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 occupancy 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.

```
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.

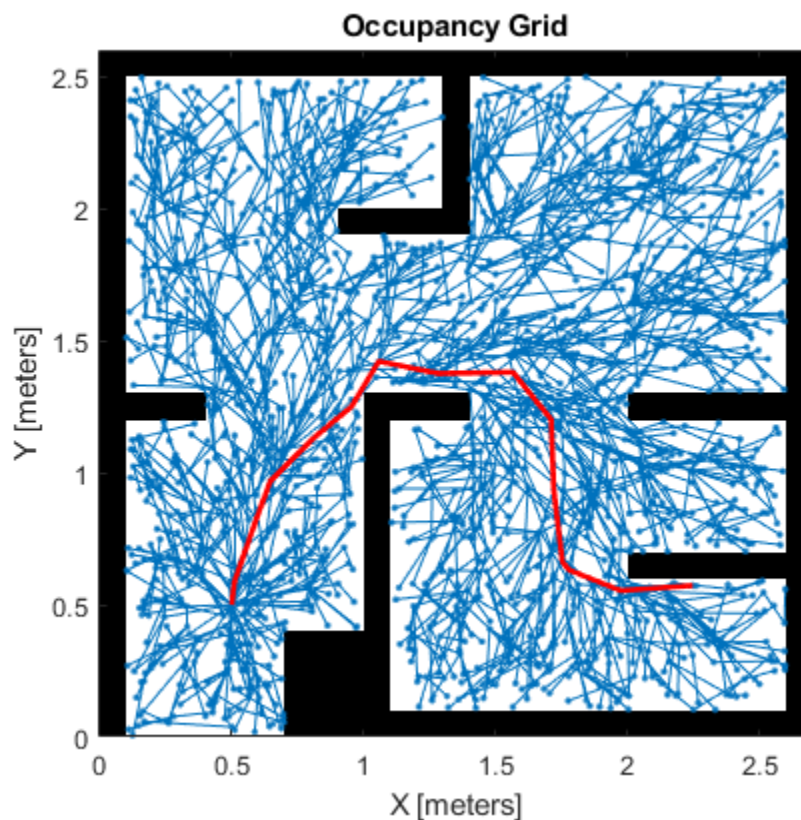

```
start = [0.5, 0.5 0];
goal = [2.5, 0.2, 0];
```

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." *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™.

See Also

[navPath](#) | [plannerRRT](#) | [stateSpaceDubins](#) | [stateSpaceReedsShepp](#) | [stateSpaceSE2](#)

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.

Input Arguments

planner — Path planner

`plannerRRT` object | `plannerRRTStar` object

Path planner, specified as a `plannerRRT` object or a `plannerRRTStar` object.

Data Types: `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.

Data Types: 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.
ExistFlag	Indicates the terminate status of the planner, returned as <ul style="list-style-type: none"> • 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.

Data Types: structure

Extended Capabilities

C/C++ Code Generation

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

See Also

navPath | plannerRRT | plannerRRTStar | stateSpaceDubins | stateSpaceReedsShepp | stateSpaceSE2

Introduced in R2019b

copy

Create copy of planner object

Syntax

```
planner2 = copy(planner1)
```

Description

`planner2 = copy(planner1)` creates a planner object, `planner2`, from a planner object, `planner1`.

Input Arguments

planner1 — Path planner

`plannerRRT` object | `plannerRRTStar` object

Path planner, specified as a `plannerRRT` object or a `plannerRRTStar` object.

Data Types: object

Output Arguments

planner2 — Path planner

`plannerRRT` object | `plannerRRTStar` object

Path planner, returned as a `plannerRRT` object or a `plannerRRTStar` object.

Data Types: object

Extended Capabilities

C/C++ Code Generation

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

See Also

`navPath` | `plannerRRT` | `plannerRRTStar` | `stateSpaceDubins` | `stateSpaceReedsShepp` | `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, with edge constraints that define the relative pose between nodes and the uncertainty on that measurement. The `optimizePoseGraph` function modifies the nodes to account for the uncertainty and improve the overall graph.

For 3-D pose graphs, see `poseGraph`.

To construct a pose graph iteratively, use `addRelativePose` to add a node and connect it to an existing node with specified edge constraints. Specify the uncertainty of the measurement using an information matrix. Adding an edge between two existing nodes creates a loop closure in the graph.

`LidarSLAM` (lidar-based simultaneous localization and mapping) is built around the optimization of a 2-D pose graph.

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 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 in the pose graph as an $[x \ y \ \theta]$ vector with an xy -position and orientation angle, `theta`. To specify relative poses between nodes, use `addRelativePose`. To get a list of all nodes, use `nodes`.

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 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.

Object Functions

<code>addRelativePose</code>	Add relative pose to pose graph
<code>edges</code>	Edges in pose graph
<code>edgeConstraints</code>	Edge constraints in pose graph
<code>findEdgeID</code>	Find edge ID of edge
<code>nodes</code>	Poses of nodes in pose graph
<code>optimizePoseGraph</code>	Optimize nodes in pose graph
<code>removeEdges</code>	Remove loop closure edges from graph
<code>show</code>	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.

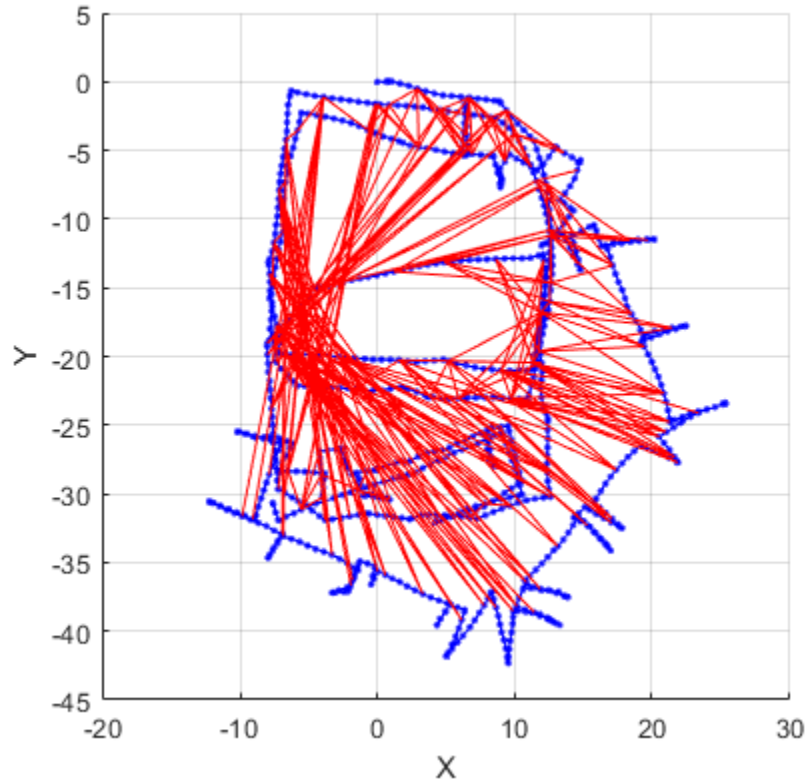
Load the Intel data set that contains a 2-D pose graph. Inspect the `poseGraph` object to view the number of nodes and loop closures.

```
load intel-2d-posegraph.mat pg
disp(pg)

poseGraph with properties:
    NumNodes: 1228
    NumEdges: 1483
    NumLoopClosureEdges: 256
    LoopClosureEdgeIDs: [1x256 double]
```

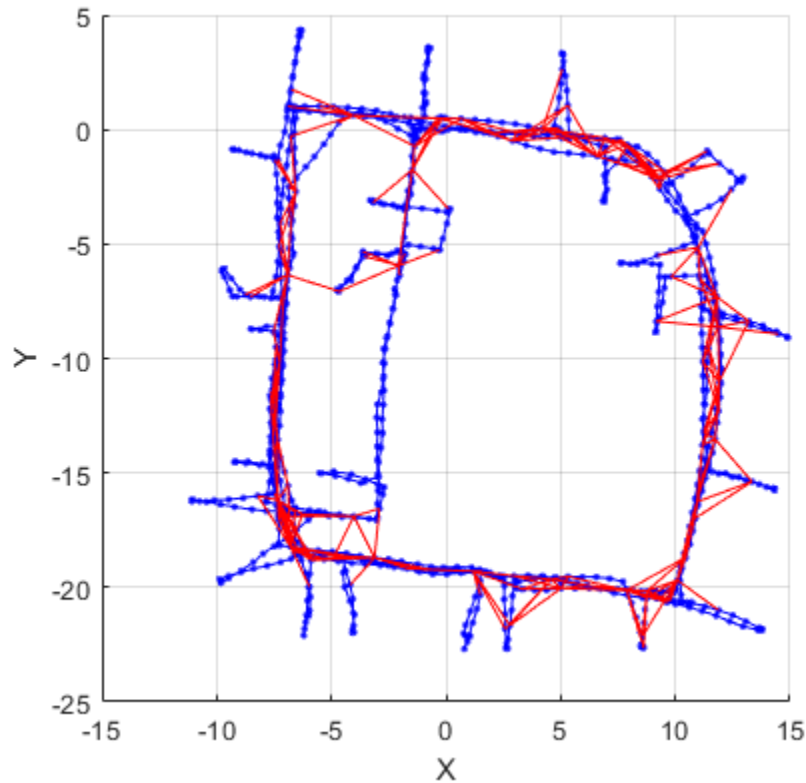
Plot the pose graph with IDs off. Red lines indicate loop closures identified in the dataset.

```
title('Original Pose Graph')  
show(pg, 'IDs', 'off');
```



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');
```

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

`addRelativePose` | `optimizePoseGraph` | `show`

Objects

`lidarSLAM` | `poseGraph3D`

Topics

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

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

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, with edge constraints that define the relative pose between nodes and the uncertainty on that measurement. The `optimizePoseGraph` function modifies the nodes to account for the uncertainty and improve the overall graph.

For 2-D pose graphs, see `poseGraph`.

To construct a pose graph iteratively, use `addRelativePose` to add poses and connect them to the existing graph. Specify the uncertainty associated using an information matrix. Specify loop closures by add extra edge constraints between existing nodes.

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 in the pose graph as an $[x \ y \ z \ q_w \ q_x \ q_y \ q_z]$ vector with an xyz-position and quaternion orientation, $[q_w \ q_x \ q_y \ q_z]$. To specify relative poses between nodes, use `addRelativePose`. To get a list of all nodes, use `nodes`.

Note The order of the quaternion $[q_w \ q_x \ q_y \ q_z]$ uses the standard convention. Some vehicle coordinate systems instead specify the order as $[q_x \ q_y \ q_z \ q_w]$. Check the source of your pose graph data before adding nodes to your `poseGraph3D` object.

NumEdges — Number of edges in pose graph \emptyset (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 are included.

NumLoopClosureEdges — Number of loop closures \emptyset (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.

Object Functions

<code>addRelativePose</code>	Add relative pose to pose graph
<code>edges</code>	Edges in pose graph
<code>edgeConstraints</code>	Edge constraints in pose graph
<code>findEdgeID</code>	Find edge ID of edge
<code>nodes</code>	Poses of nodes in pose graph
<code>optimizePoseGraph</code>	Optimize nodes in pose graph
<code>removeEdges</code>	Remove loop closure edges from graph
<code>show</code>	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.

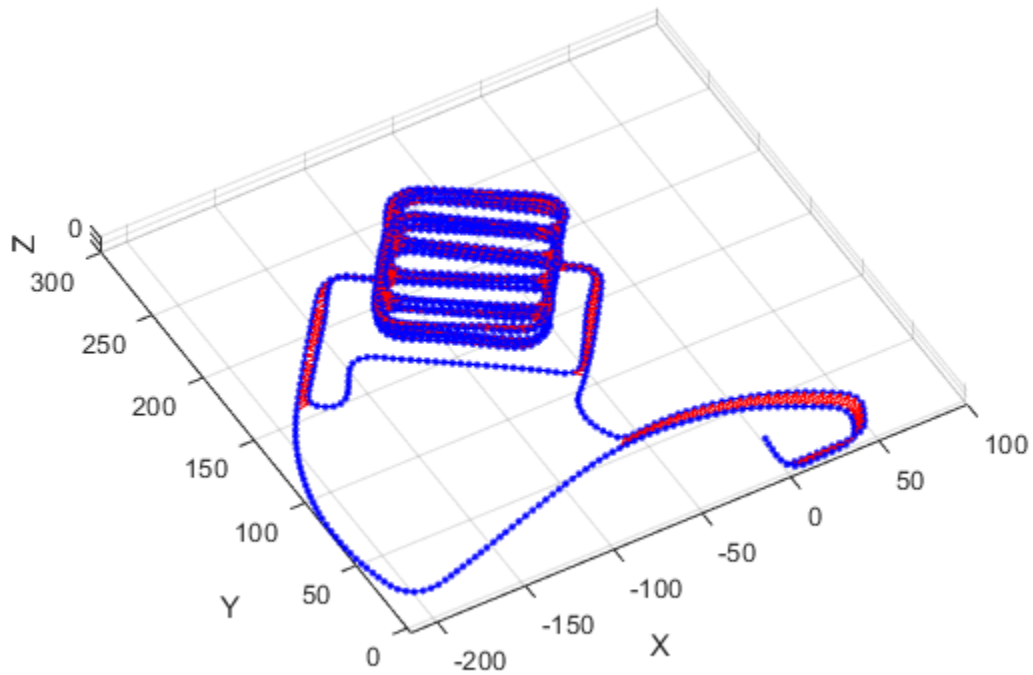
```
load parking-garage-posegraph.mat pg
disp(pg);
```

```
poseGraph3D with properties:
```

```
    NumNodes: 1661
    NumEdges: 6275
NumLoopClosureEdges: 4615
  LoopClosureEdgeIDs: [1x4615 double]
```

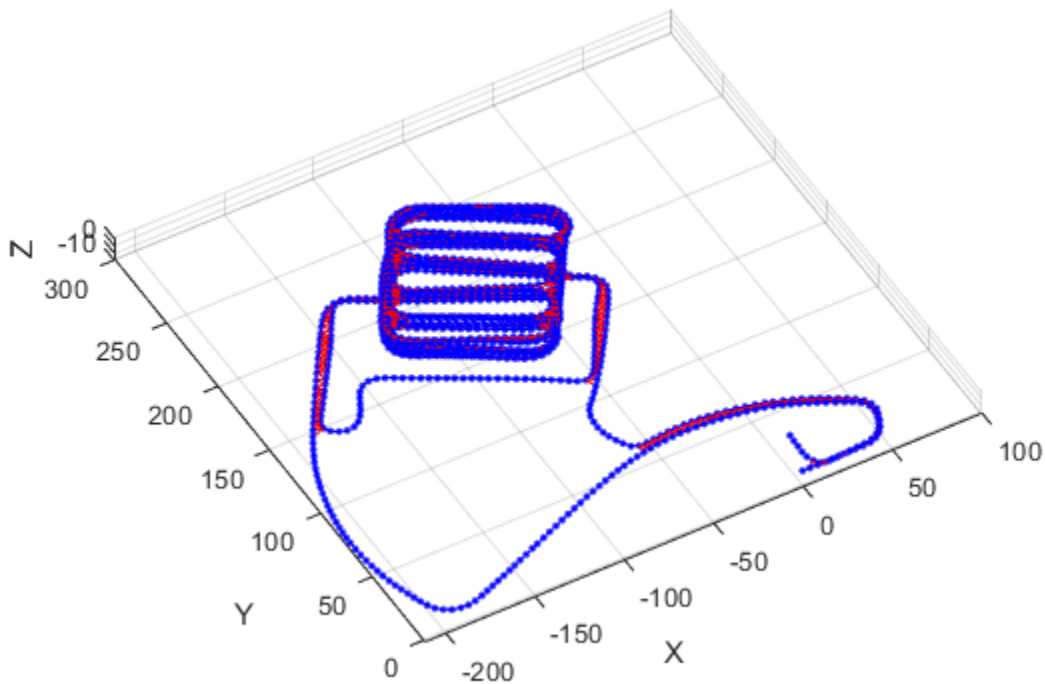
Plot the pose graph with IDs off. Red lines indicate loop closures identified in the dataset.

```
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

`addRelativePose` | `optimizePoseGraph`

Objects

`lidarSLAM` | `poseGraph`

Introduced in R2019b

quaternion

Create a quaternion array

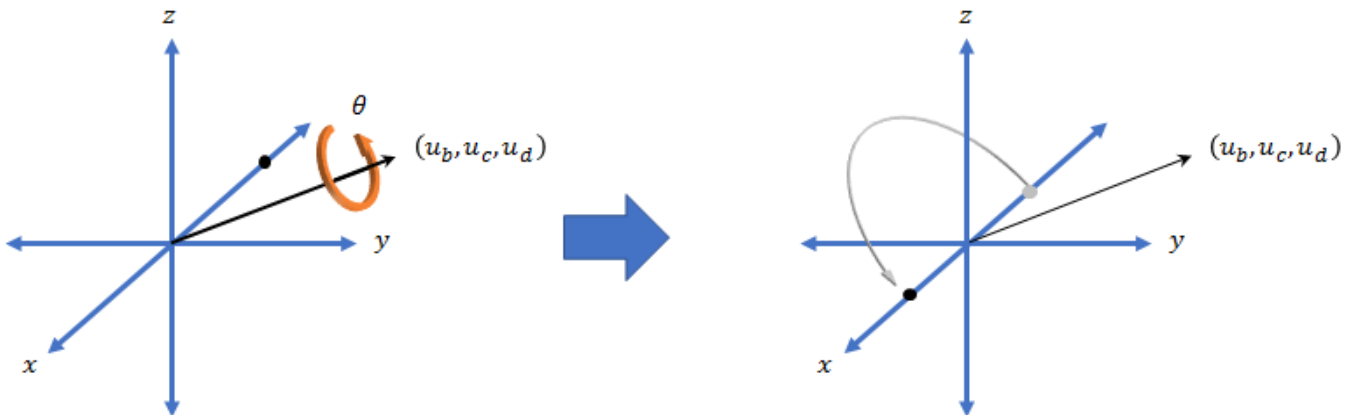
Description

A quaternion is a four-part hyper-complex number used in three-dimensional rotations and orientations.

A quaternion number is represented in the form $a + bi + cj + dk$, where a , b , c , and d parts are real numbers, and i , j , and k are the basis elements, satisfying the equation: $i^2 = j^2 = k^2 = ijk = -1$.

The set of quaternions, denoted by \mathbf{H} , is defined within a four-dimensional vector space over the real numbers, \mathbf{R}^4 . Every element of \mathbf{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 $q = \cos(\theta/2) + \sin(\theta/2)(u_b i + u_c j + u_d k)$, where θ is the angle of rotation and $[u_b, u_c, \text{ and } u_d]$ 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];
```

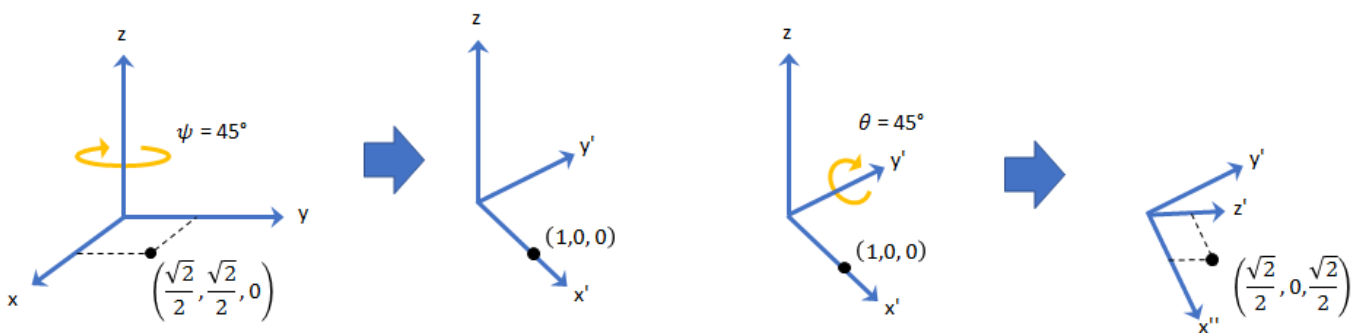
In this representation, the first column represents the x-axis, the second column represents the y-axis, and the third column represents the z-axis.

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)
```

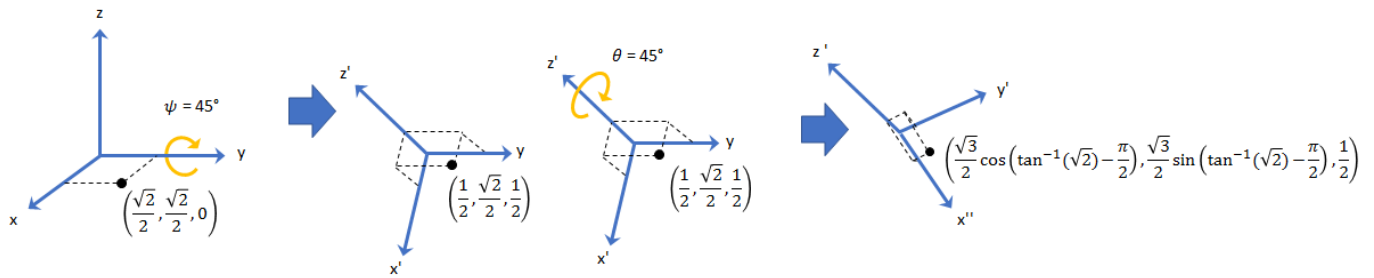
```
newPointCoordinate =
    0.7071    -0.0000    0.7071
```



- 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
ctranspose, '	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
minus, -	Quaternion subtraction
mtimes, *	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, .'	Transpose a quaternion array
uminus, -	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 = 3x4

    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 = 3x4

    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 = 10x3
```

```

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=2×2 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 $N = \text{numel}(\text{quat})$.

Data Types: single | double

Extended Capabilities

C/C++ Code Generation

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

See Also

Functions

classUnderlying | parts

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 $q = a + bi + cj + dk$, the complex conjugate of q is $q^* = a - bi - cj - dk$. 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 + 0.063055k
   -2.2588 + 0.43359i + 1.3499j - 0.71474k    0.86217 - 0.34262i - 3.0349j + 0.20497k
```

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 + 0.71474k
    1.8339 + 0.86217i - 1.3077j + 0.34262k    2.7694 + 3.0349i - 0.063055j - 0.20497k
```

```
quatTransposed = quat'
```

```
quatTransposed=2x2 quaternion array
    0.53767 + 2.2588i - 0.31877j + 0.43359k    1.8339 - 0.86217i + 1.3077j - 0.34262k
    3.5784 + 1.3499i - 0.7254j - 0.71474k    2.7694 - 3.0349i + 0.063055j + 0.20497k
```

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 the quaternion rotation operators for `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 = 5×1 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 $[Adim1, \dots, AdimN] = \text{size}(\text{quatA})$ and $[Bdim1, \dots, BdimN] = \text{size}(\text{quatB})$, 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 $\text{size}(\text{quatA})$ and $\text{size}(\text{quatB})$.

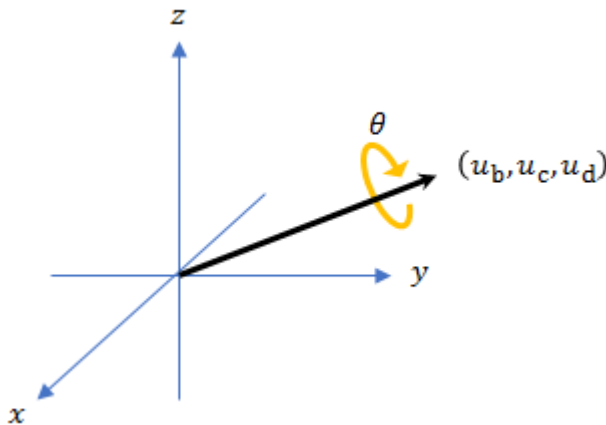
Data Types: single | double

Algorithms

The `dist` function returns the angular distance between two quaternion rotation operators.

A quaternion may be defined by an axis (u_b, u_c, u_d) and angle of rotation θ_q :

$$q = \cos\left(\frac{\theta_q}{2}\right) + \sin\left(\frac{\theta_q}{2}\right)(u_b i + u_c j + u_d k).$$



Given a quaternion in the form, $q = a + bi + cj + dk$, where a is the real part, you can solve for θ_q :
 $\theta_q = 2\cos^{-1}(a)$.

Consider two quaternions, p and q , and the product $z = p * \text{conjugate}(q)$. In a rotation operator, z rotates by p and derotates by q . As p approaches q , the angle of z goes to 0, and the product 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, angular distance is calculated as:

```
angularDistance = 2*acos(parts(p*conj(q)));
```

Extended Capabilities

C/C++ Code Generation

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

See Also

Functions

conj | parts

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]);
eulerAnglesRadians = euler(quat, 'ZYX', 'frame')
```

```
eulerAnglesRadians = 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' | 'YZX' | 'ZYX' | 'ZYX' | 'ZYX' | 'ZYX' | 'ZYX' | 'ZYX' | 'ZYX' | 'ZYX'

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.

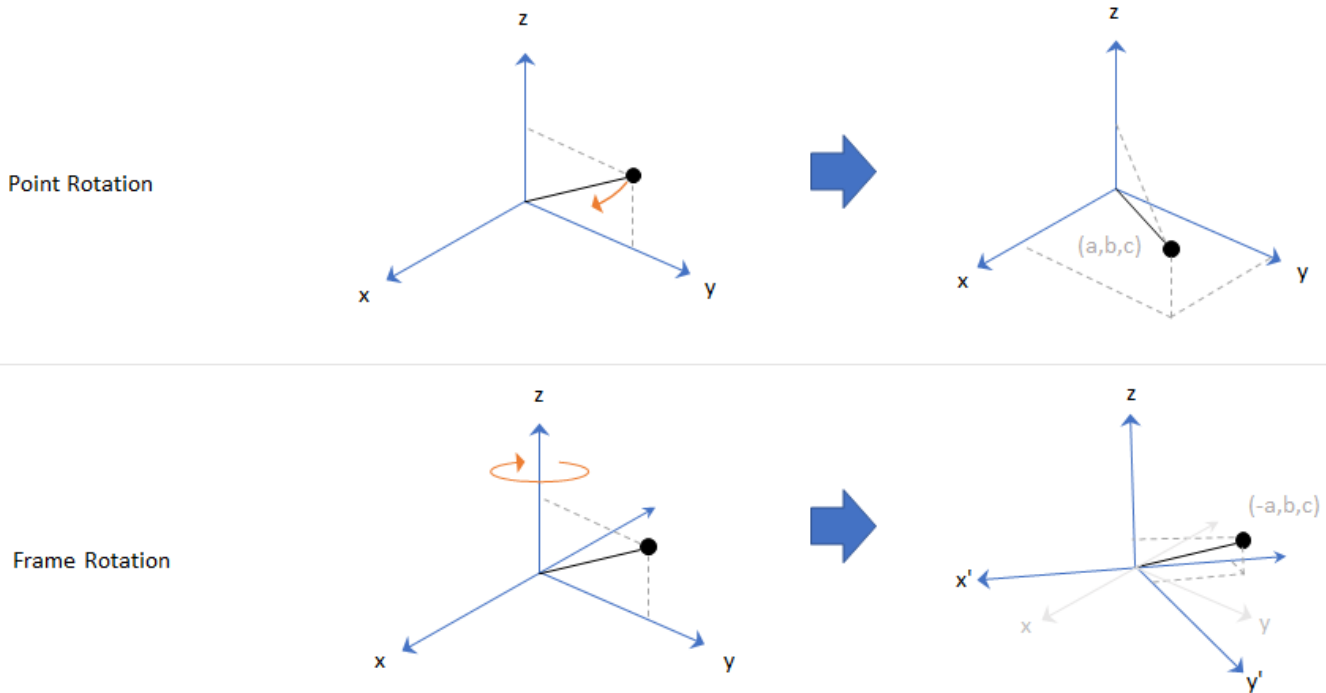
Data Types: char | string

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' | 'YZX' | 'ZXY' | 'XZY' | 'YZY' | 'ZYX' | 'XZY' | 'YZX' | '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.

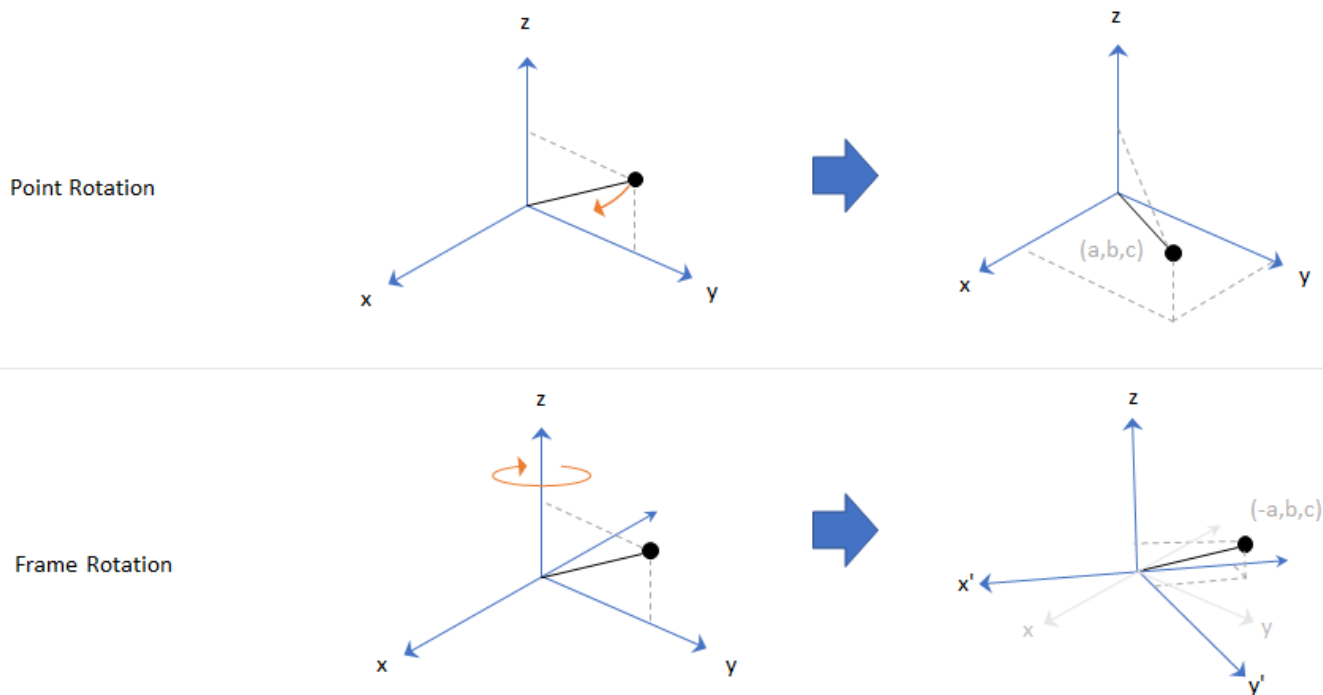
Data Types: char | string

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

$B = \text{exp}(A)$

Description

$B = \text{exp}(A)$ 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=4×1 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=4×1 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 $A = a + bi + cj + dk = a + \bar{v}$, 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

log | power, .^

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=2×1 quaternion array
    1 + 2i + 3j + 4k
    5 + 6i + 7j + 8k
```

```
B = 2;
C = A.\B
```

```
C=2×1 quaternion array
    0.066667 - 0.133333i - 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=2×2 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=2×2 quaternion array

```
16 + 2i + 3j + 13k      9 + 7i + 6j + 12k
5 + 11i + 10j + 8k      4 + 14i + 15j + 1k
```

C = A.\B

C=2×2 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 $A = a_1 + a_2i + a_3j + a_4k$ and a real scalar p ,

$$C = p.\backslash A = \frac{a_1}{p} + \frac{a_2}{p}i + \frac{a_3}{p}j + \frac{a_4}{p}k$$

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

$$C = A.\backslash B = A^{-1} .* B = \left(\frac{\text{conj}(A)}{\text{norm}(A)^2} \right) .* B$$

Extended Capabilities

C/C++ Code Generation

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

See Also

Functions

conj | norm | rdivide, ./ | times, .*

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=3×1 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=3×1 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 $A = a + \bar{v} = a + bi + cj + dk$, 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 `meanrot(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');

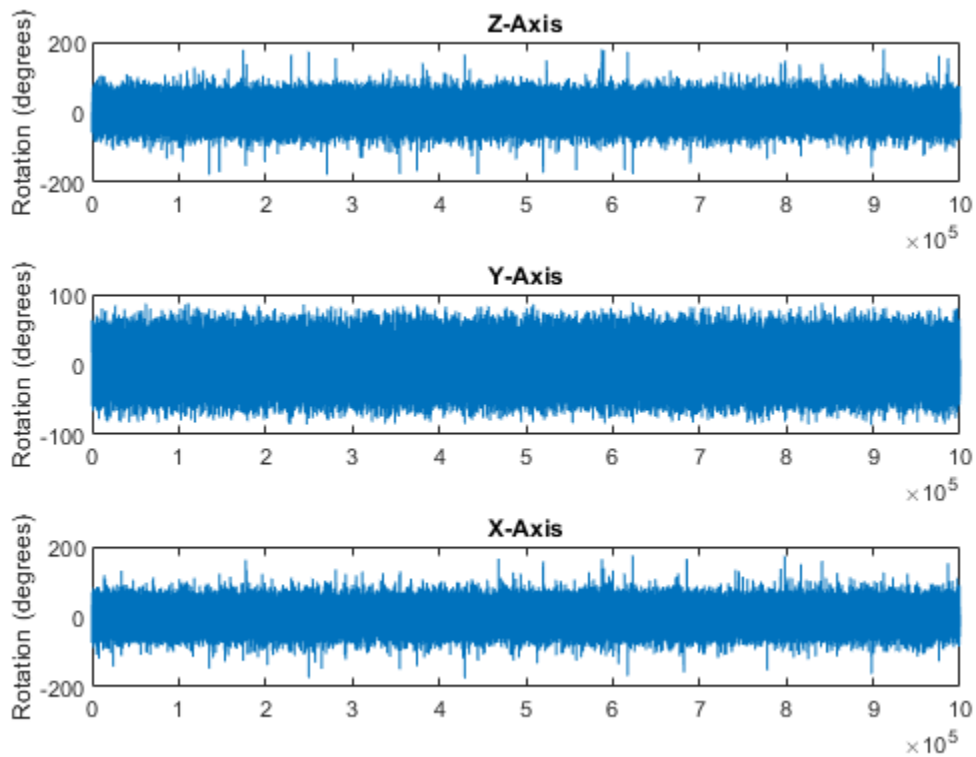
figure(1)

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

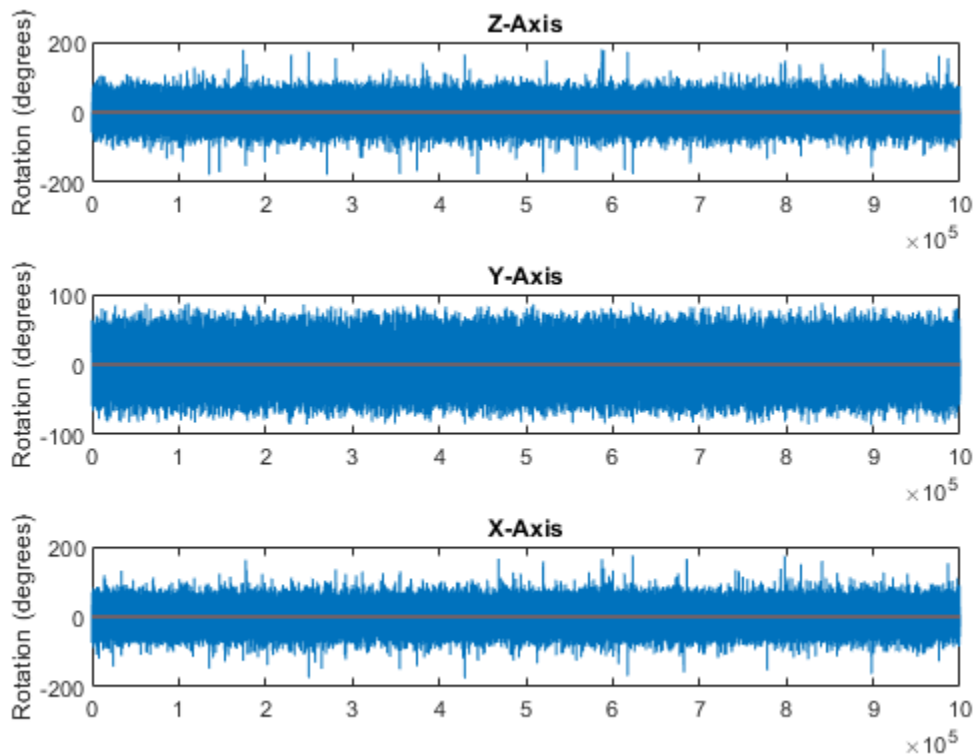
subplot(3,1,3)
plot(noisyEulerAngles(:,3))
title('X-Axis')
ylabel('Rotation (degrees)')
hold on

```



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:

- q_0 represents no rotation.
- q_{90} 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, q_{sweep} , 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 q_0 , q_{90} , and q_{sweep} 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');
```

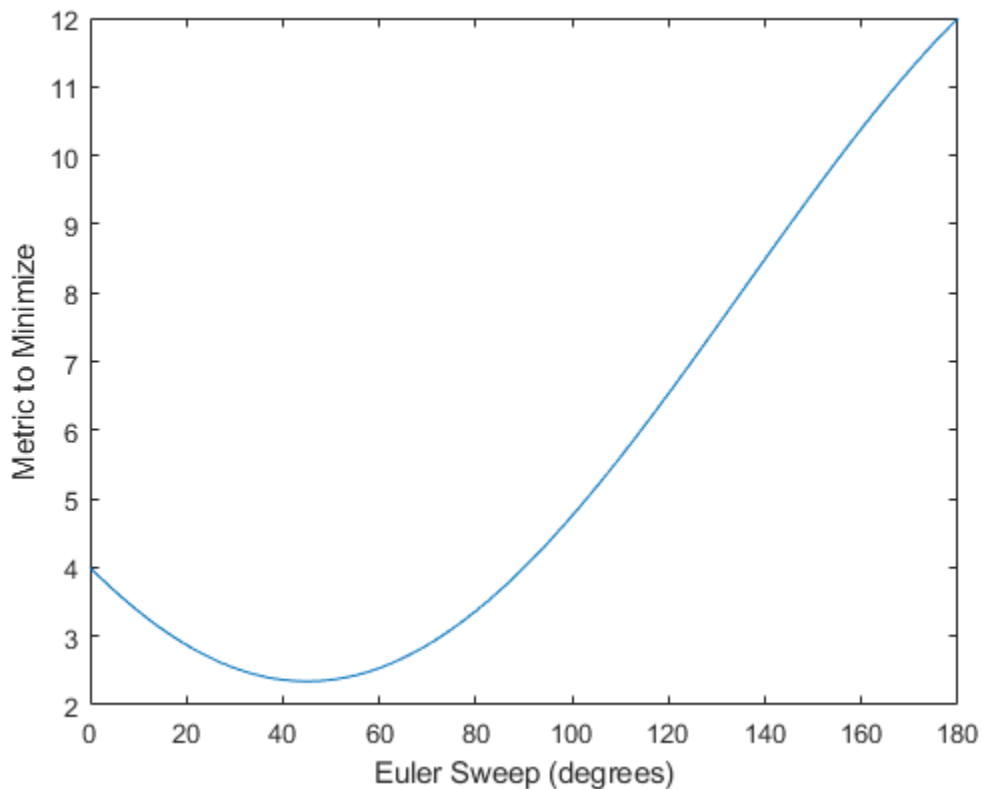
```

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 quaternion([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 = 1x3
```



```
0      0  45.0000
```

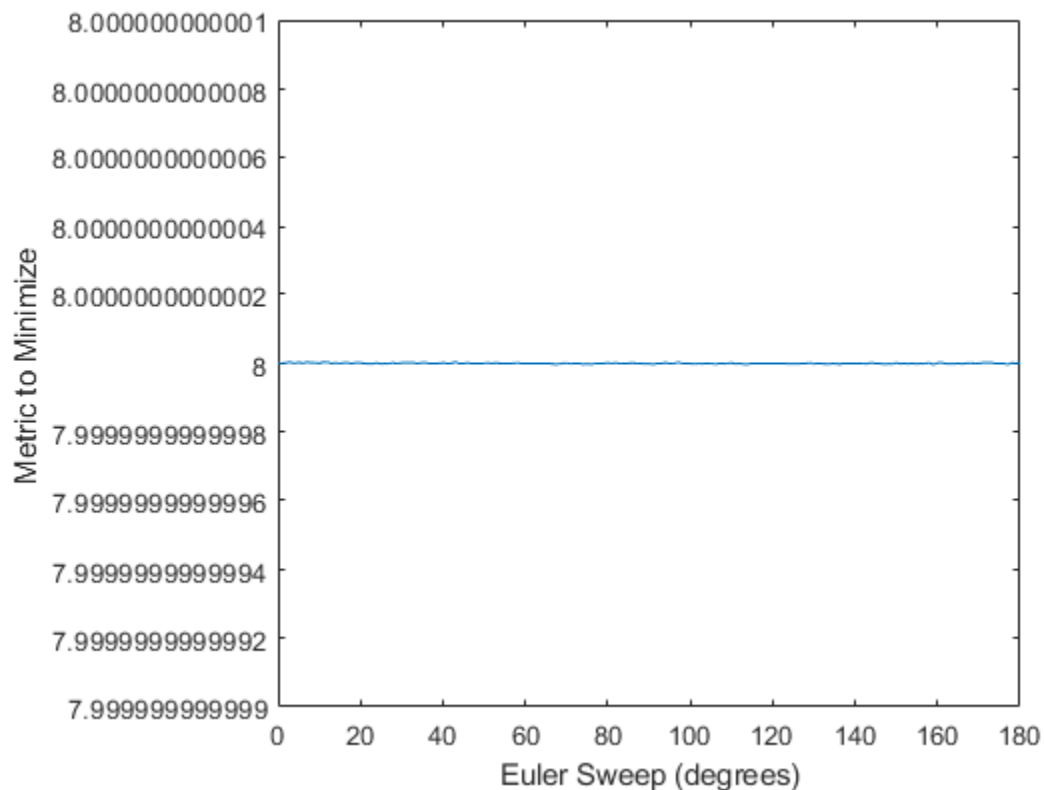
Limitations

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

mtimes, * | times, .* | uminus, -

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

$$q = a_q + b_q i + c_q j + d_q k,$$

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:

$$\begin{aligned}
 z = pq &= (a_p + b_p i + c_p j + d_p k)(a_q + b_q i + c_q j + d_q k) \\
 &= a_p a_q + a_p b_q i + a_p c_q j + a_p d_q k \\
 &\quad + b_p a_q i + b_p b_q i^2 + b_p c_q j + b_p d_q i k \\
 &\quad + c_p a_q j + c_p b_q j i + c_p c_q j^2 + c_p d_q j k \\
 &\quad + d_p a_q k + d_p b_q k i + d_p c_q k j + d_p d_q k^2
 \end{aligned}$$

You can simplify the equation using the quaternion multiplication table:

$$\begin{aligned}
 z = pq &= a_p a_q + a_p b_q i + a_p c_q j + a_p d_q k \\
 &\quad + b_p a_q i - b_p b_q + b_p c_q k - b_p d_q j \\
 &\quad + c_p a_q j - c_p b_q k - c_p c_q + c_p d_q i \\
 &\quad + d_p a_q k + d_p b_q j - d_p c_q i - d_p d_q
 \end{aligned}$$

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 $Q = a + bi + cj + dk$, the norm of the quaternion is defined as $\text{norm}(Q) = \sqrt{a^2 + b^2 + c^2 + d^2}$.

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

`conj` | `normalize` | `parts`

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 $Q = a + bi + cj + dk$, 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

`conj` | `norm` | `times`, `.*`

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

```
quat0nes = ones('quaternion')
quat0nes = ones(n,'quaternion')
quat0nes = ones(sz,'quaternion')
quat0nes = ones(sz1,...,szN,'quaternion')

quat0nes = ones( ____, 'like', prototype, 'quaternion')
```

Description

`quat0nes = 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 $Q = a + bi + cj + dk$, a quaternion one is defined as $Q = 1 + 0i + 0j + 0k$.

`quat0nes = ones(n,'quaternion')` returns an n-by-n quaternion matrix with the real parts set to 1 and the imaginary parts set to 0.

`quat0nes = ones(sz,'quaternion')` returns an array of quaternion ones where the size vector, `sz`, defines `size(q0nes)`.

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.

`quat0nes = ones(sz1,...,szN,'quaternion')` returns a `sz1`-by-...-by-`szN` array of ones where `sz1,...,szN` indicates the size of each dimension.

`quat0nes = 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.

```
quat0nes = ones('quaternion')

quat0nes = quaternion
          1 + 0i + 0j + 0k
```

Square Matrix of Quaternion Ones

Create an n-by-n matrix of quaternion ones.

```
n = 3;
quat0nes = ones(n, 'quaternion')

quat0nes=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];
quat0nesSyntax1 = ones(dims, 'quaternion')
```

```
quat0nesSyntax1 = 3x1x2 quaternion array
quat0nesSyntax1(:,:,1) =
```

```
    1 + 0i + 0j + 0k
    1 + 0i + 0j + 0k
    1 + 0i + 0j + 0k
```

```
quat0nesSyntax1(:,:,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:

```
quat0nesSyntax2 = ones(3,1,2, 'quaternion');
isequal(quat0nesSyntax1, quat0nesSyntax2)
```

```
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 $Q = a + bi + cj + dk$, a quaternion one is defined as $Q = 1 + 0i + 0j + 0k$.

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=2×1 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, d, 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=2×1 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
```

$C=2 \times 3$ quaternion array

$$\begin{array}{cccccccccccc} 1 + & 2i + & 3j + & 4k & 1 + & 0i + & 0j + & 0k & -28 + & 4i + & 6j + \\ -2110 - & 444i - & 518j - & 592k & -124 + & 60i + & 70j + & 80k & 5 + & 6i + & 7j + \end{array}$$

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 $A = a + bi + cj + dk$ is given by

$$A = \|A\|(\cos\theta + \hat{u}\sin\theta)$$

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

$$P = A.^b = \|A\|^b(\cos(b\theta) + \hat{u}\sin(b\theta))$$

Extended Capabilities

C/C++ Code Generation

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

See Also

Functions

exp | log

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=3×3 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=1×3 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=2×1 quaternion array
    1 + 2i + 3j + 4k
    5 + 6i + 7j + 8k
```

```
B = 2;
C = A./B
```

```
C=2×1 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=2×2 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=2×2 quaternion array

$$\begin{array}{cc} 1 + 2i + 3j + 4k & 2 + 3i + 4j + 5k \\ 3 + 4i + 5j + 6k & 4 + 5i + 6j + 7k \end{array}$$

C = A./B

C=2×2 quaternion array

$$\begin{array}{cccc} 2.7 - & 0.1i - & 2.1j - & 1.7k \\ 1.8256 - & 0.081395i + & 0.45349j - & 0.24419k \end{array} \quad \begin{array}{cccc} 2.2778 + & 0.092593i - & 0.46296j - & 0.5740 \\ 1.4524 - & 0.5i + & 1.0238j - & 0.261 \end{array}$$

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 $A = a_1 + a_2i + a_3j + a_4k$ and a real scalar p ,

$$C = A ./ p = \frac{a_1}{p} + \frac{a_2}{p}i + \frac{a_3}{p}j + \frac{a_4}{p}k$$

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,

$$C = A ./ B = A .* B^{-1} = A .* \left(\frac{\text{conj}(B)}{\text{norm}(B)^2} \right)$$

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

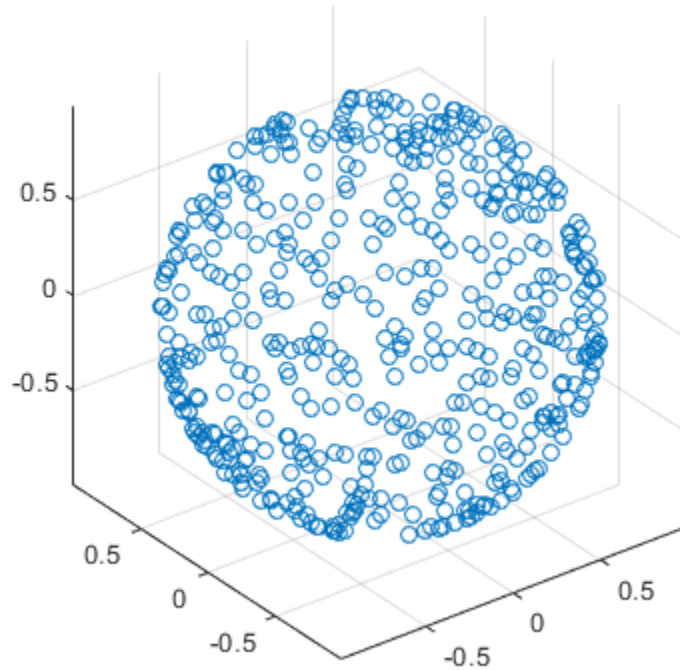
Generate a 3-by-3 matrix of uniformly distributed random rotations.

```
r = randrot(3)
```

Create Uniform Distribution of Random Rotations

Create a vector of 500 random quaternions. Use `rotatepoint` on page 2-707 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

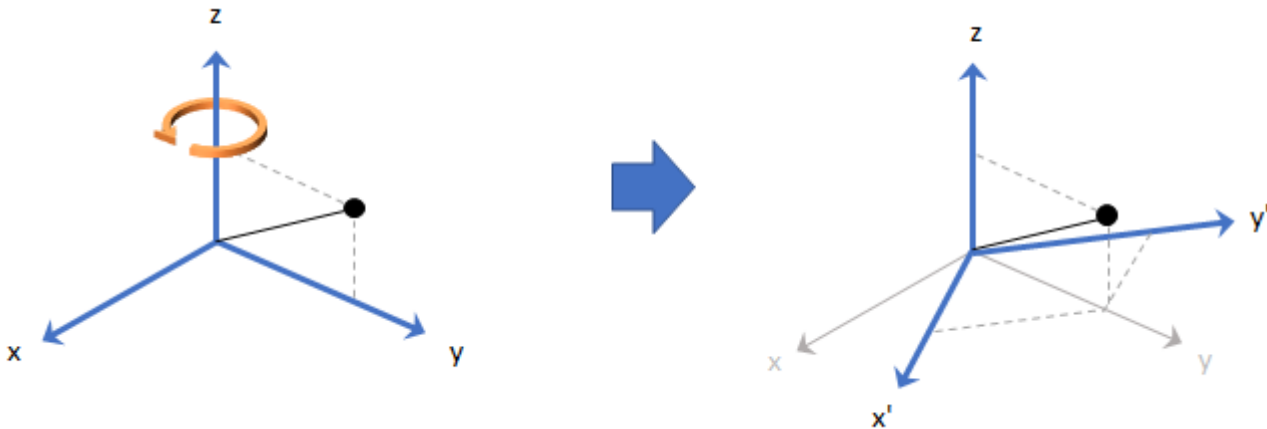
Quaternion frame rotation

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.

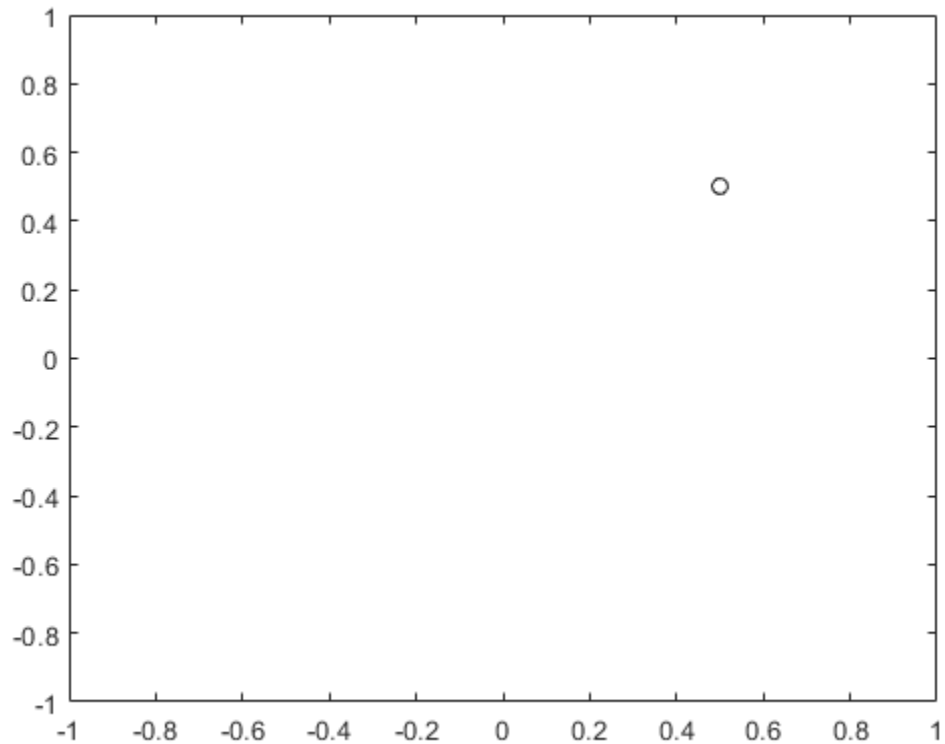


Examples

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');
```

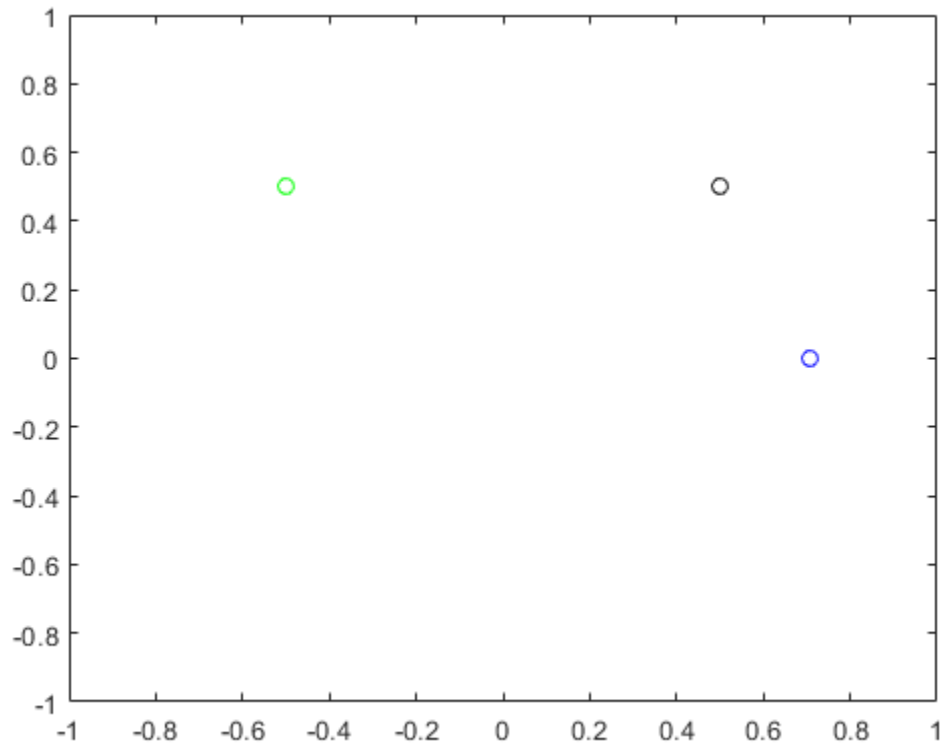
```
rereferencedPoint = rotateframe(quat, [x,y,z])
```

```
rereferencedPoint = 2x3
```

```
    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
```

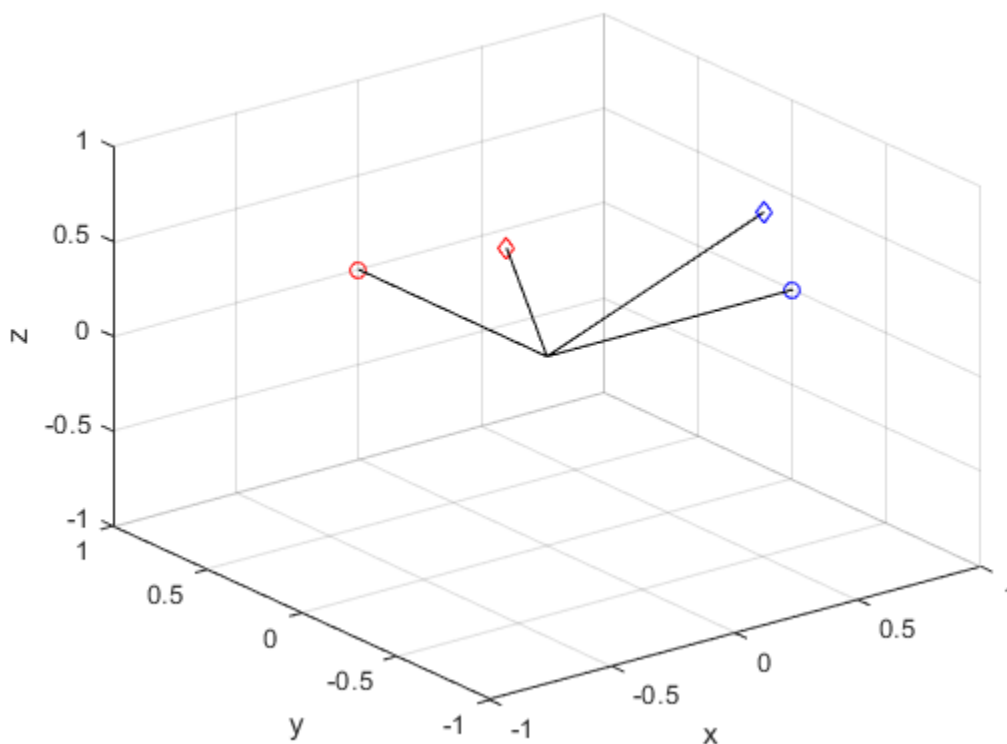
```

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, $q = a + bi + cj + dk$, 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

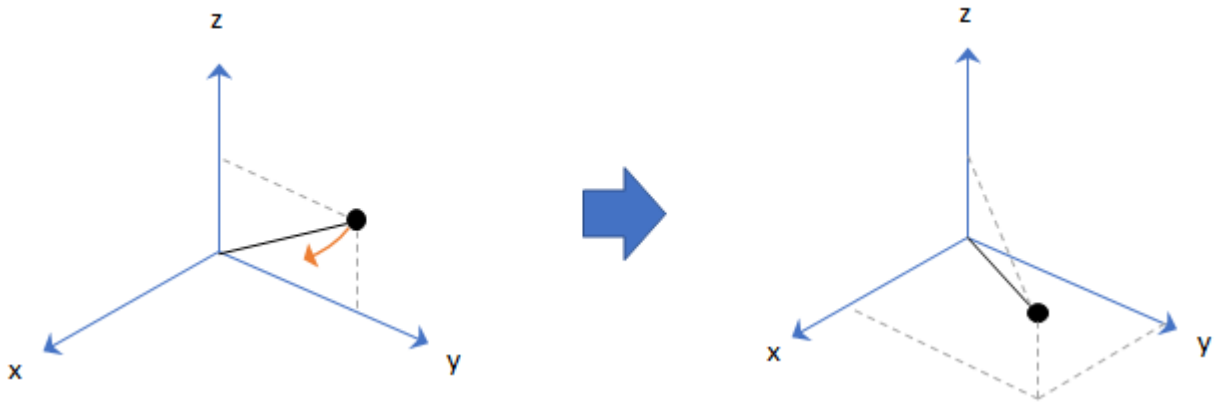
Quaternion point rotation

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.

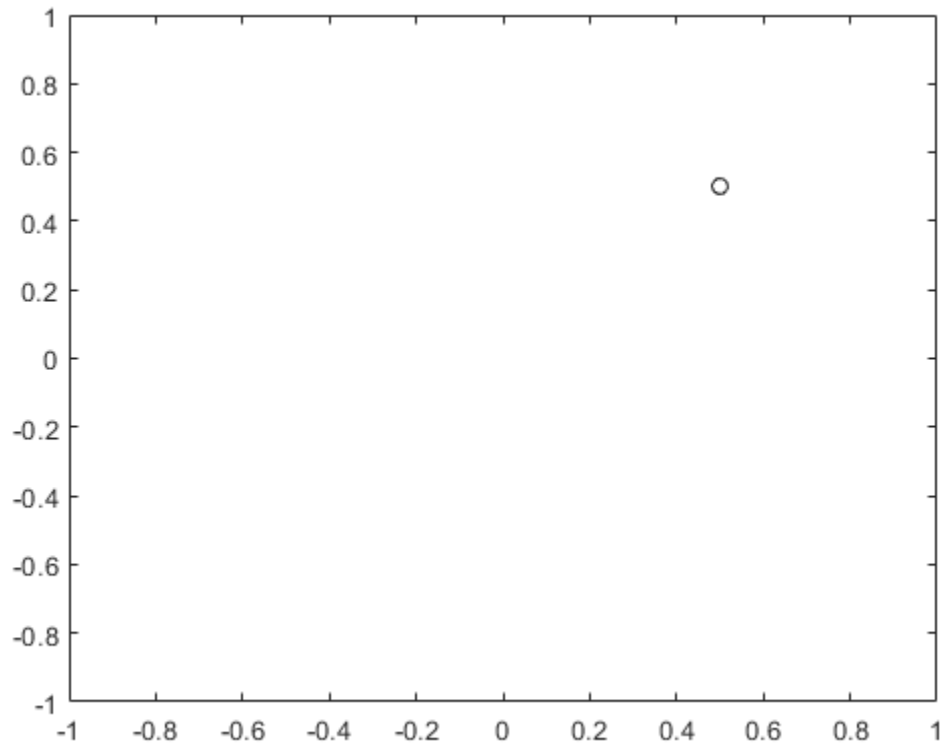


Examples

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');
```

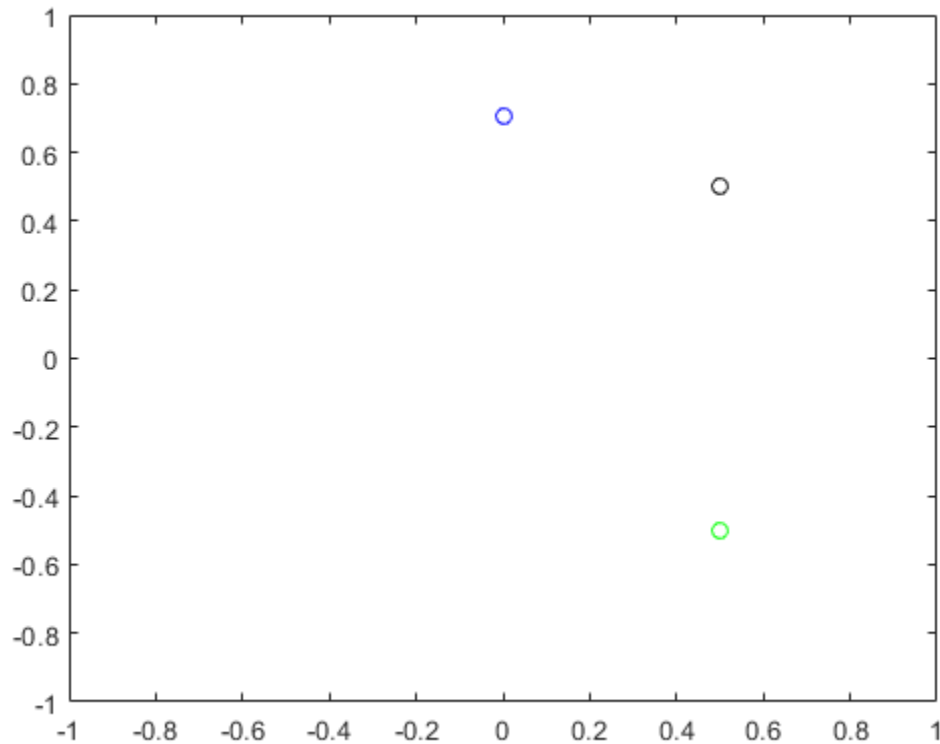
```
rotatedPoint = rotatepoint(quat,[x,y,z])
```

```
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
```

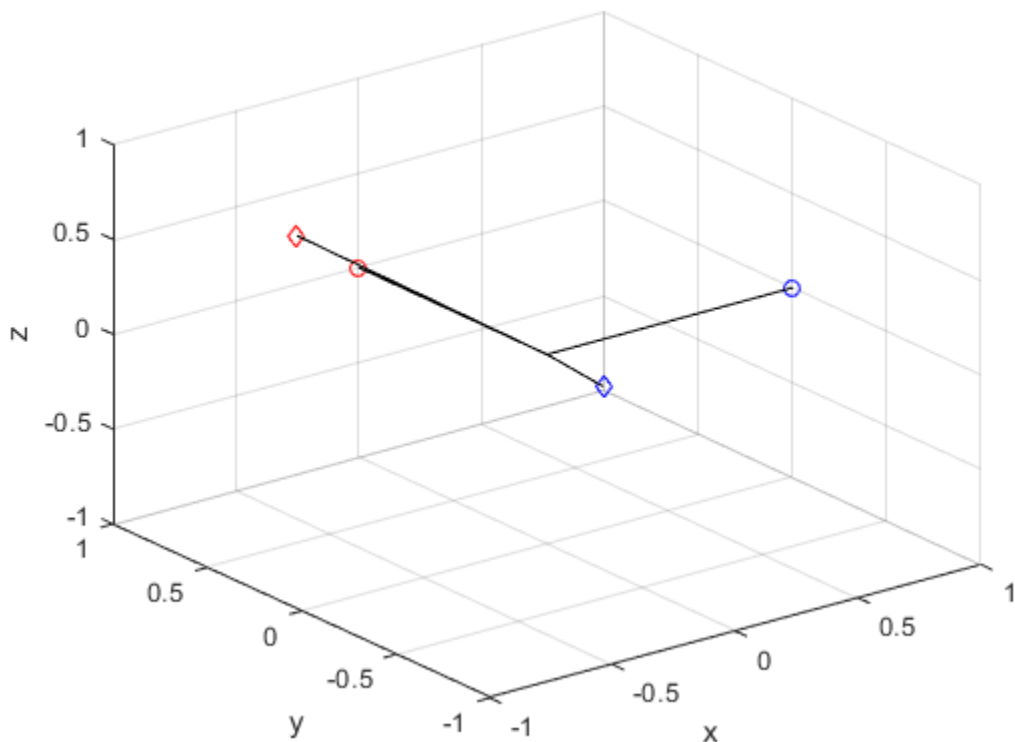
```

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 points1-by-3 vector | N -by-3 matrixThree-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, $q = a + bi + cj + dk$, 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 = 3x3

    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 = 3x3

    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 = 3x3

    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

`euler` | `eulerd` | `rotvec` | `rotvecd`

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

$$q = \cos(\theta/2) + \sin(\theta/2)(xi + yj + zk),$$

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

$$q_{rv} = \frac{\theta}{\sin(\theta/2)}[b, c, d].$$

Extended Capabilities

C/C++ Code Generation

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

See Also

Functions

euler | eulerd | rotvecd

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

$$q = \cos(\theta/2) + \sin(\theta/2)(xi + yj + zk),$$

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

$$q_{rv} = \frac{\theta}{\sin(\theta/2)}[b, c, d].$$

Extended Capabilities

C/C++ Code Generation

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

See Also

Functions

euler | eulerd | rotvec

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`.

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.
```

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

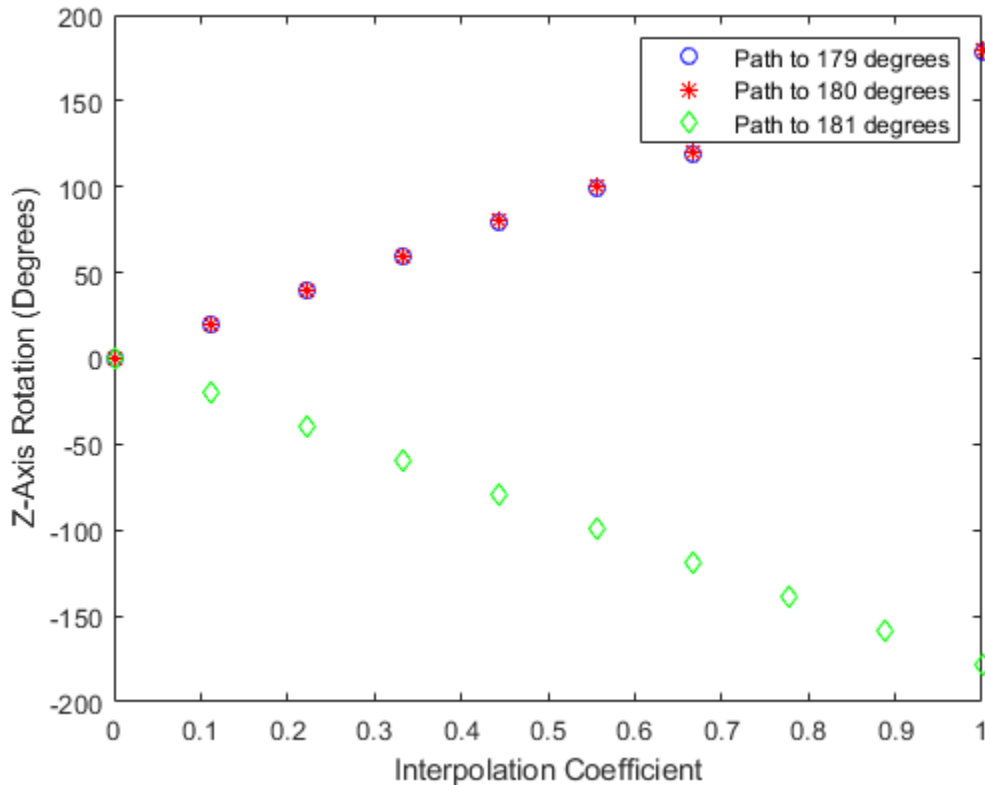
```
q0 = ones(1, 'quaternion');  
q179 = quaternion([179,0,0], 'eulerd', 'ZYX', 'frame');  
q180 = quaternion([180,0,0], 'eulerd', 'ZYX', 'frame');  
q181 = quaternion([181,0,0], 'eulerd', 'ZYX', 'frame');
```

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 q_0 and q_{179} is clockwise to minimize the great circle distance. The path between q_0 and q_{181} is counterclockwise to minimize the great circle distance. The path between q_0 and q_{180} can be either clockwise or counterclockwise, depending on numerical rounding.

Input Arguments

q1 – Quaternion

scalar | vector | matrix | multidimensional array

Quaternion to interpolate, specified as a scalar, vector, matrix, or multidimensional array of quaternions.

q_1 , q_2 , 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.

q_1 , q_2 , 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].

q_1 , q_2 , 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 **spherical linear interpolation** (SLERP) is an extension of linear interpolation along a plane to spherical interpolation in three dimensions. The algorithm was first proposed in [1] (Sensor Fusion and Tracking Toolbox). 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
   -4.2329 + 2.4547i + 3.7768j + 0.77484k   -0.65232 - 0.43112i - 1.4645j - 0.90073k
```

```
-4.4159 + 2.1926i + 1.9037j - 4.0303k -2.0232 + 0.4205i - 0.17288j + 3.8529k
```

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=1×3 quaternion array
      0 +      0i +      0j +      0k      1 +      0i +      0j +      0k      0
```

```
b = quaternion(randn(4,4))
```

```
b=4×1 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=4×3 quaternion array
      0 +      0i +      0j +      0k      0.31877 +      3.5784i +      0.7254j -      0.12414k
      0 +      0i +      0j +      0k      -1.3077 +      2.7694i -      0.063055j +      1.4897k
      0 +      0i +      0j +      0k      -0.43359 -      1.3499i +      0.71474j +      1.409k
      0 +      0i +      0j +      0k      0.34262 +      3.0349i -      0.20497j +      1.4172k
```

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:

$$\begin{aligned}
 z = pq &= (a_p + b_p i + c_p j + d_p k)(a_q + b_q i + c_q j + d_q k) \\
 &= a_p a_q + a_p b_q i + a_p c_q j + a_p d_q k \\
 &\quad + b_p a_q i + b_p b_q i^2 + b_p c_q j + b_p d_q i k \\
 &\quad + c_p a_q j + c_p b_q j i + c_p c_q j^2 + c_p d_q j k \\
 &\quad + d_p a_q k + d_p b_q k i + d_p c_q k j + d_p d_q k^2
 \end{aligned}$$

You can simplify the equation using the quaternion multiplication table.

$$\begin{aligned}
 z = pq &= a_p a_q + a_p b_q i + a_p c_q j + a_p d_q k \\
 &\quad + b_p a_q i - b_p b_q + b_p c_q k - b_p d_q j \\
 &\quad + c_p a_q j - c_p b_q k - c_p c_q + c_p d_q i \\
 &\quad + d_p a_q k + d_p b_q j - d_p c_q i - d_p d_q
 \end{aligned}$$

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

mtimes, * | prod

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=4×1 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=1×4 quaternion array
    0.53767 + 0.31877i + 3.5784j + 0.7254k    1.8339 - 1.3077i + 2.7694j - 0.063055k
```

Matrix Transpose

Create a matrix of quaternions and compute its nonconjugate transpose.

```
quat = [quaternion(randn(2,4)), quaternion(randn(2,4))]
```

```
quat=2×2 quaternion array
    0.53767 - 2.2588i + 0.31877j - 0.43359k    3.5784 - 1.3499i + 0.7254j + 0.71474k
    1.8339 + 0.86217i - 1.3077j + 0.34262k    2.7694 + 3.0349i - 0.063055j - 0.20497k
```

```
quatTransposed = quat.'
```

```
quatTransposed=2×2 quaternion array
    0.53767 - 2.2588i + 0.31877j - 0.43359k    1.8339 + 0.86217i - 1.3077j + 0.34262k
    3.5784 - 1.3499i + 0.7254j + 0.71474k    2.7694 + 3.0349i - 0.063055j - 0.20497k
```

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`, ' `quat`

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 + 0.7147k  
    1.8339 - 1.3077i + 2.7694j - 0.063055k    0.86217 + 0.34262i + 3.0349j - 0.2049k
```

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 - 0.7147k  
   -1.8339 + 1.3077i - 2.7694j + 0.063055k   -0.86217 - 0.34262i - 3.0349j + 0.2049k
```

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 $Q = a + bi + cj + dk$, a quaternion zero is defined as $Q = 0 + 0i + 0j + 0k$.

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

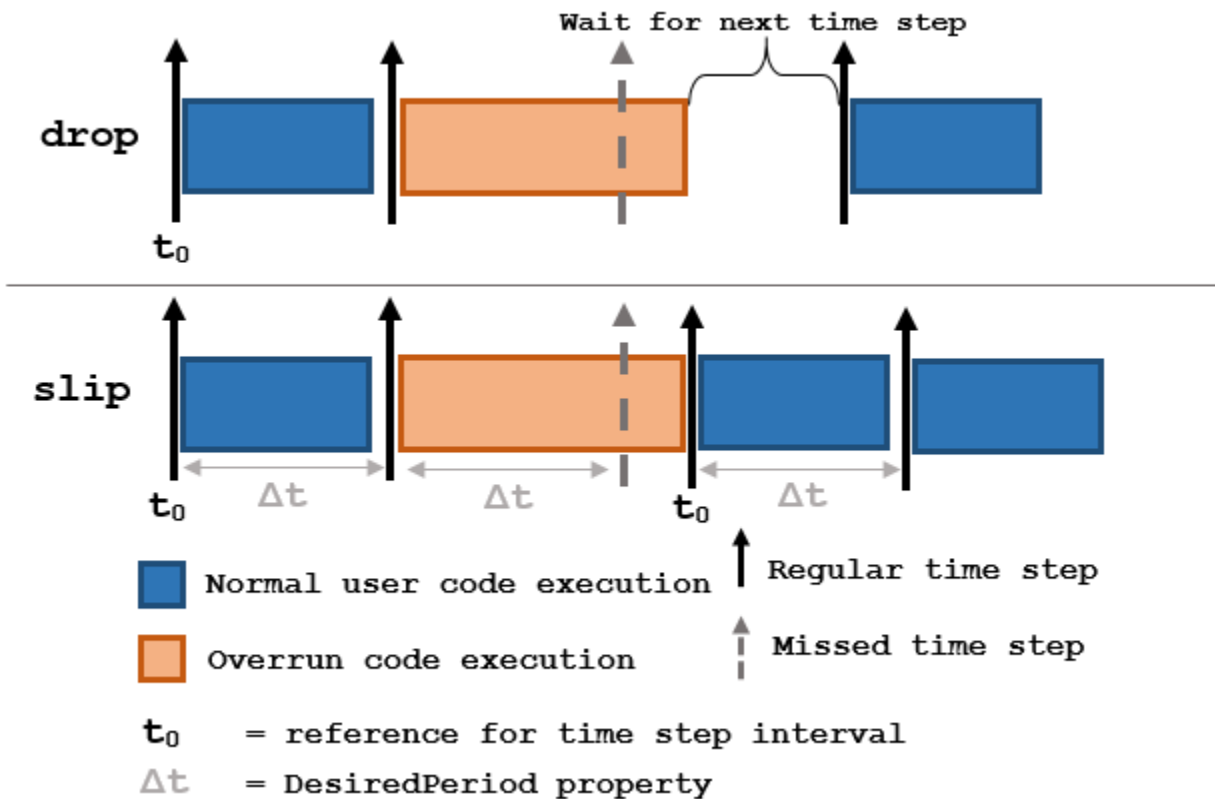
Elapsed time between last two calls to `waitfor`, specified as a scalar. By default, `LastPeriod` is set to NaN until `waitfor` is called for the first time. After the first call, `LastPeriod` equals `TotalElapsedTime`.

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: 0.005878
Iteration: 2 - Time Elapsed: 1.000761
Iteration: 3 - Time Elapsed: 2.002471
Iteration: 4 - Time Elapsed: 3.000673
Iteration: 5 - Time Elapsed: 4.001496
Iteration: 6 - Time Elapsed: 5.000765
Iteration: 7 - Time Elapsed: 6.000638
Iteration: 8 - Time Elapsed: 7.000956
Iteration: 9 - Time Elapsed: 8.001423
Iteration: 10 - Time Elapsed: 9.000287
```

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: [1x30 double]
    NumPeriods: 30
    AveragePeriod: 0.0501
    StandardDeviation: 0.0014
    NumOVERRUNS: 0
```

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: 15.0221
    LastPeriod: 0.5006
```

Reset the object to restart the time statistics.

```
reset(r);
disp(r)

rateControl with properties:

    DesiredRate: 2
    DesiredPeriod: 0.5000
    OverrunAction: 'slip'
    TotalElapsedTime: 0.0087
    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

`reset` | `rosclock` | `statistics` | `waitfor`

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: 15.0221
    LastPeriod: 0.5006
```

Reset the object to restart the time statistics.

```
reset(r);  
disp(r)
```

```
rateControl with properties:
```

```
    DesiredRate: 2  
    DesiredPeriod: 0.5000  
    OverrunAction: 'slip'  
    TotalElapsedTime: 0.0087  
    LastPeriod: NaN
```

See Also

[rateControl](#) | [waitfor](#)

Topics

“Execute Code at a Fixed-Rate”

Introduced in R2016a

statistics

Statistics of past execution periods

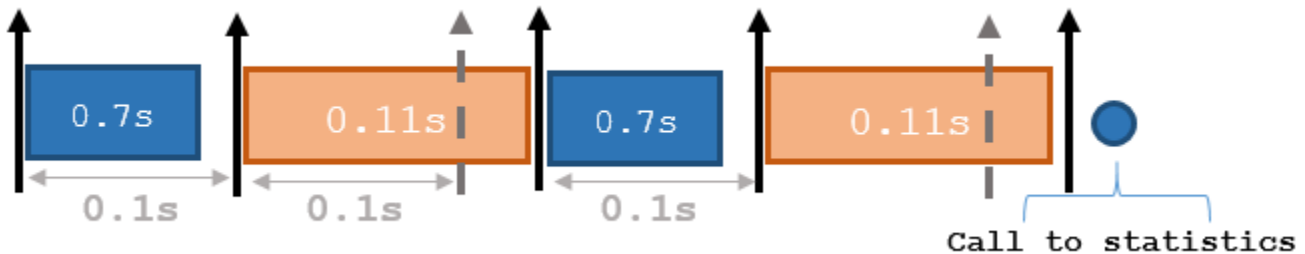
Syntax

```
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: [1x30 double]
    NumPeriods: 30
    AveragePeriod: 0.0501
    StandardDeviation: 0.0014
    NumOverruns: 0
```

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: 0.005878
Iteration: 2 - Time Elapsed: 1.000761
Iteration: 3 - Time Elapsed: 2.002471
Iteration: 4 - Time Elapsed: 3.000673
Iteration: 5 - Time Elapsed: 4.001496
Iteration: 6 - Time Elapsed: 5.000765
Iteration: 7 - Time Elapsed: 6.000638
Iteration: 8 - Time Elapsed: 7.000956
Iteration: 9 - Time Elapsed: 8.001423
Iteration: 10 - Time Elapsed: 9.000287
```

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 `robotics.Rate` 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 `reedsSheppConnection` 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];
```

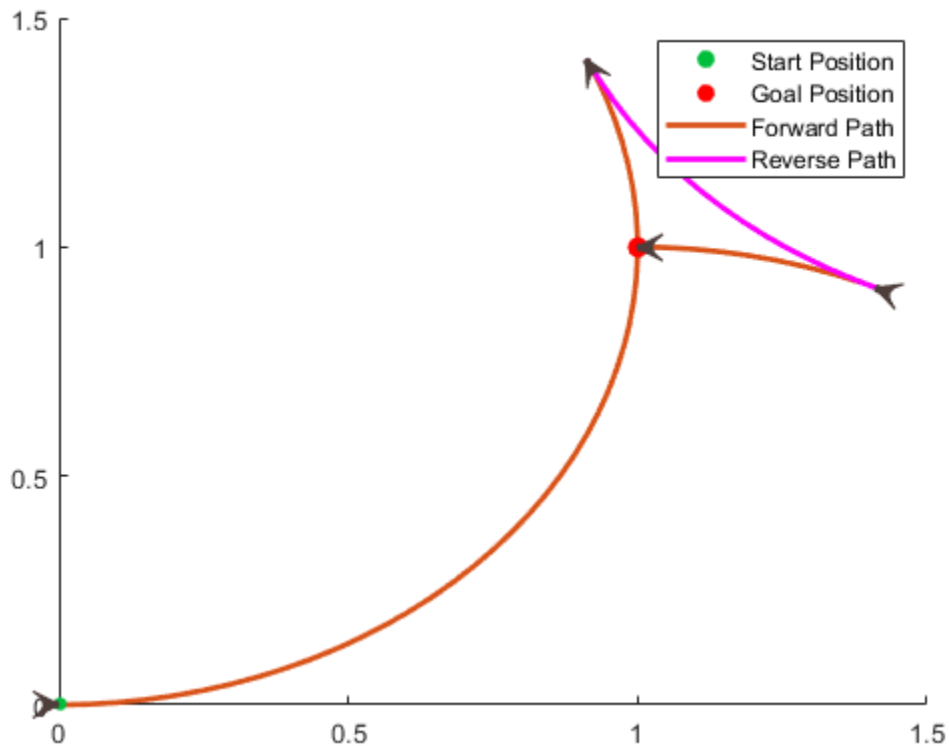
```
goalPose = [1 1 pi];
```

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.

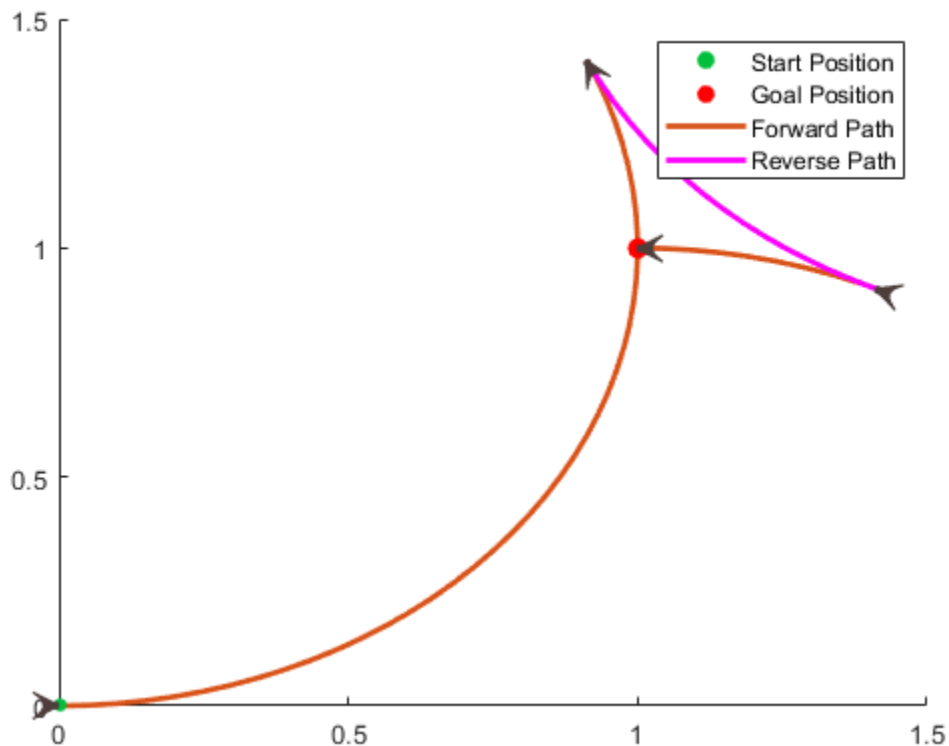
```
startPose = [0 0 0];
goalPose = [1 1 pi];
```

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 = 1x5
     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.

```

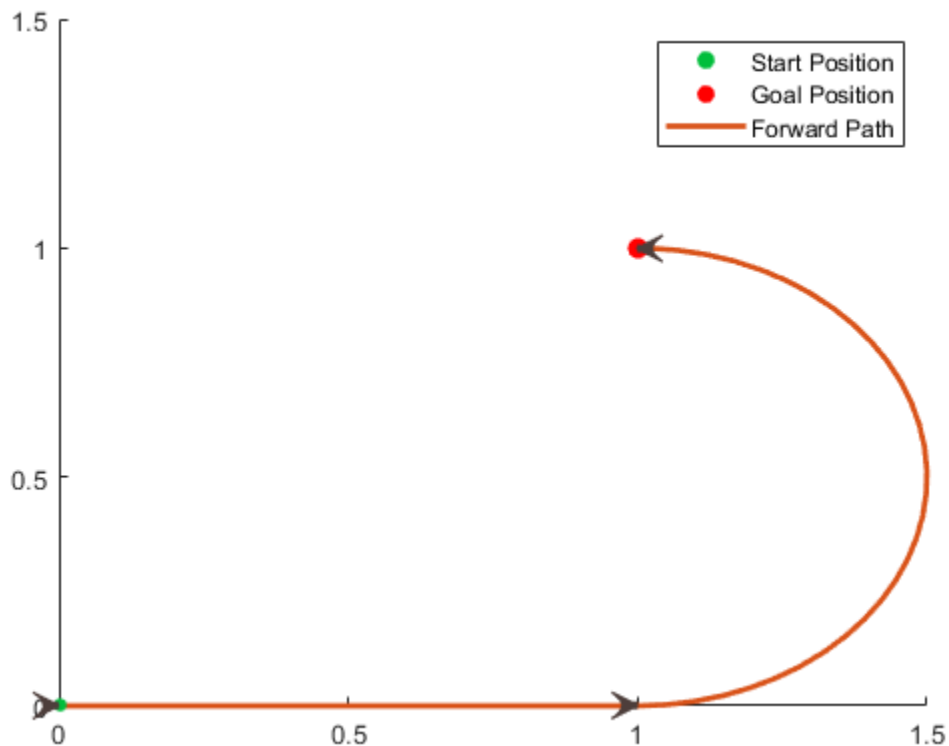
reedsConnObj = reedsSheppConnection('DisabledPathTypes',{'LpRnLp'});
reedsConnObj.MinTurningRadius = 0.5;
reedsConnObj.ReverseCost = 5;

[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

dubinsConnection | dubinsPathSegment | reedsSheppPathSegment

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];
```

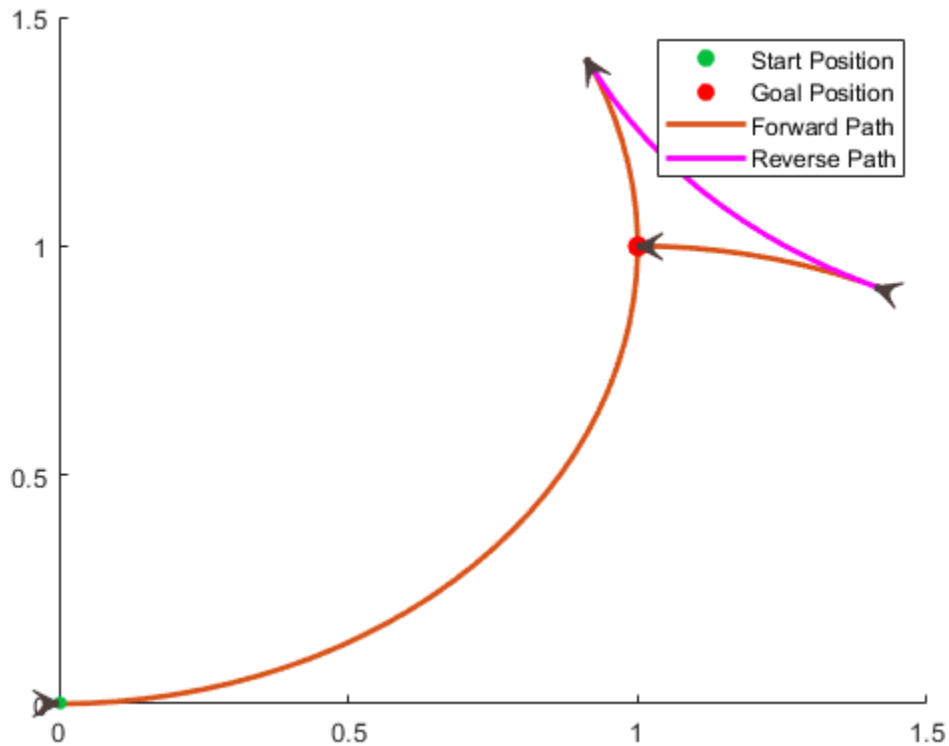
```
goalPose = [1 1 pi];
```

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

[dubinsConnection](#) | [dubinsPathSegment](#) | [reedsSheppConnection](#)

Functions

[connect](#) | [interpolate](#) | [show](#)

Introduced in R2019b

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 `navParticleResamplingPolicy` object 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

`correct` | `stateEstimatorPF`

Introduced in R2019b

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 `stateTransitionFcn` 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

<code>initialize</code>	Initialize the state of the particle filter
<code>getStateEstimate</code>	Extract best state estimate and covariance from particles
<code>predict</code>	Predict state of robot in next time step
<code>correct</code>	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

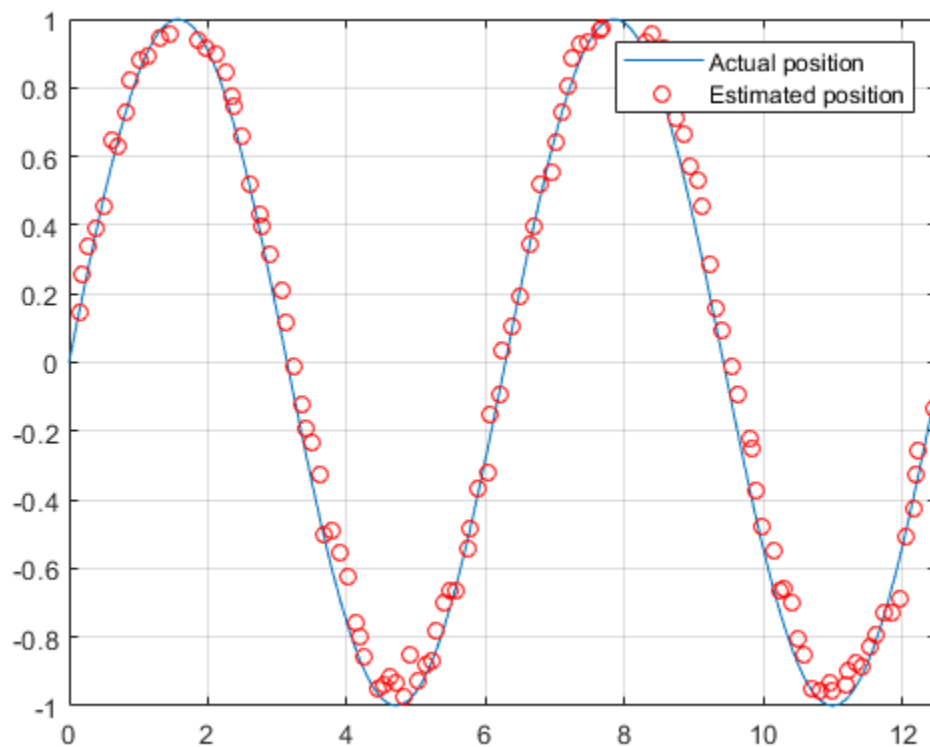
```

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

`correct` | `getStateEstimate` | `initialize` | `predict` | `resamplingPolicyPF`

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

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

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 = 1x3

```

```

    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 [].

See Also

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

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 [].

See Also

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

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`.

See Also

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

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 `stateTransitionFcn` 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 [].

See Also

correct | getStateEstimate | initialize | predict | resamplingPolicyPF | stateEstimatorPF

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 $[x, y, \theta]$. 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  
sapce = stateSpaceSE2(bounds)
```

Description

`space = stateSpaceSE2` creates an SE(2) state space object with default state bounds for x , y , and θ .

`sapce = 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 copy of state space object
distance	Distance between two states
enforceStateBounds	Ensure state lie within 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.

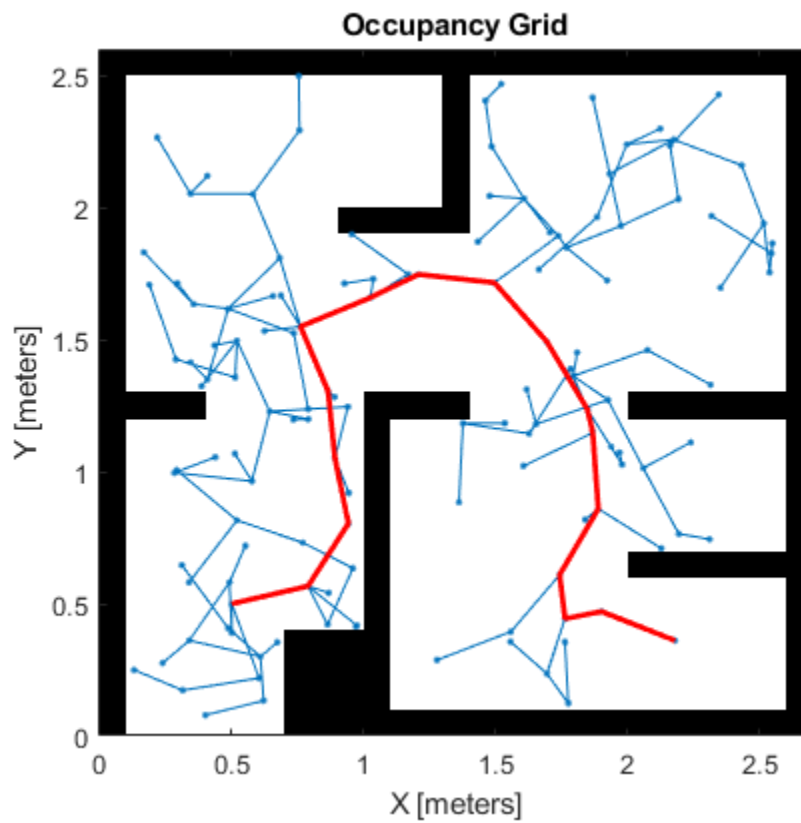
```
start = [0.5,0.5,0];
goal = [2.5,0.2,0];
```

Plan a path with default settings.

```
rng(100,'twister'); % 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

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 $[x, y, \theta]$. 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 copy of state space object
distance	Distance between two states
enforceStateBounds	Ensure state lie within 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 turning 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.

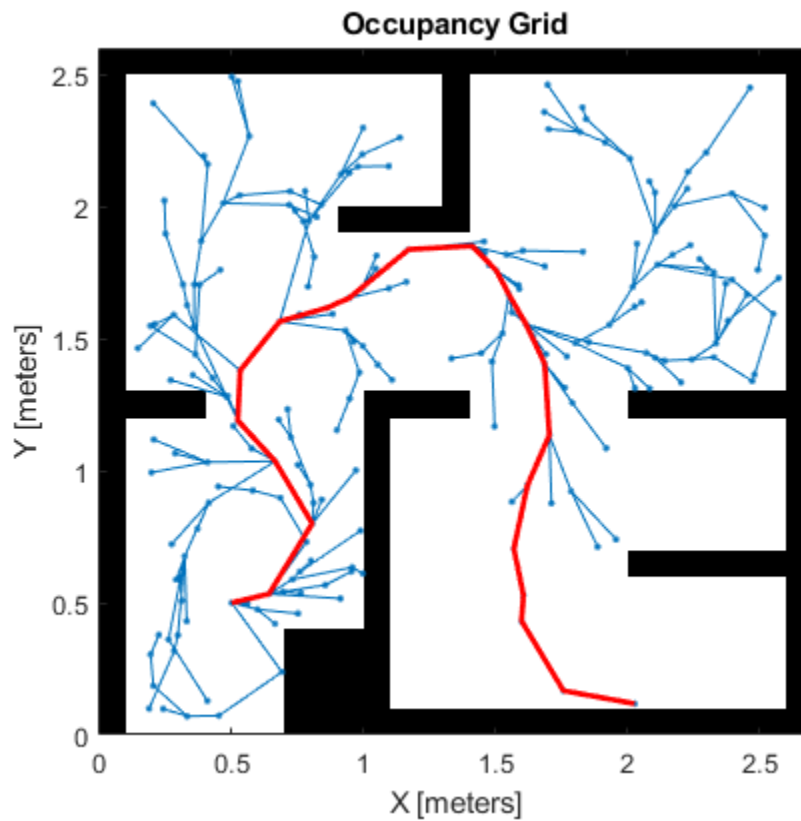
```
start = [0.5,0.5,0];
goal = [2.5,0.2,0];
```

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

[dubinsConnection](#) | [stateSpaceReedsShepp](#) | [stateSpaceSE2](#)

Introduced in R2019b

stateSpaceReedsShepp

State space for Reeds-Shepp vehicles

Description

The `stateSpaceReedsShepp` object stores parameters and states in the ReedsShepp state space, which is composed of state vectors represented by $[x, y, \theta]$. 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

<code>copy</code>	Create copy of state space object
<code>distance</code>	Distance between two states
<code>enforceStateBounds</code>	Ensure state lie within state bounds
<code>interpolate</code>	Interpolate between states
<code>sampleGaussian</code>	Sample state using Gaussian distribution
<code>sampleUniform</code>	Sample state using uniform distribution

Examples**Plan Path Between Two States in ReedsShepp State Space**

Create a ReedsShepp state space.

```
ss = stateSpaceReedsShepp;
```

Create an `occupancyMap`-based state validator using the created state space.

```
sv = validatorOccupancyMap(ss);
```

Create an occupancy map from an example map and 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.

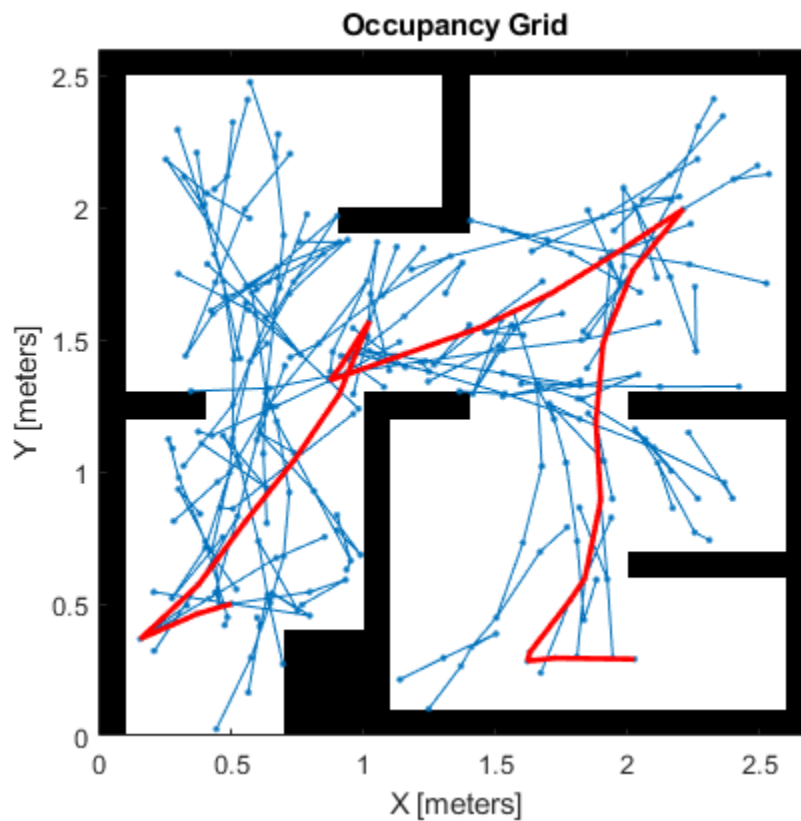
```
start = [0.5,0.5,0];
goal = [2.5,0.2,0];
```

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

[reedsSheppConnection](#) | [stateSpaceDubins](#) | [stateSpaceSE2](#)

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 state space specified by `space`.

Input Arguments

space — State space object

`stateSpaceSE2` object | `stateSpaceDubins` object | `stateSpaceReedsShepp` object

State space object, specified as a `stateSpaceSE2`, a `stateSpaceDubins`, or a `stateSpaceReedsShepp` object.

Data Types: object

states1 — States for distance calculation

M -by-3 real-valued matrix

States for distance calculation, specified as an M -by-3 real-valued matrix. Each row of the matrix corresponds to one incidence of state in the state space. The number of rows in `states1` must be exactly the same as that of `states2`.

Data Types: single | double

states2 — States for distance calculation

M -by-3 real-valued matrix

States for distance calculation, specified as an M -by-3 real-valued matrix. Each row of the matrix corresponds to one incidence of state in the state space. The number of rows in `states2` must be exactly the same as that of `states1`.

Data Types: single | double

Output Arguments

dist — Distance between states

M -by-1 real vector

Distance between states, returned as an M -by-1 real vector. The dimension M depends on the dimension of the input.

See Also

`stateSpaceDubins` | `stateSpaceReedsShepp` | `stateSpaceSE2`

Introduced in R2019b

interpolate

Interpolate between states

Syntax

```
interpStates = interpolate(space, state1, state2, ratio)
```

Description

`interpStates = interpolate(space, state1, state2, ratio)` returns the states interpolated from `state1` and `state2` based on the specified ratio.

Examples

Interpolate Between States in SE2

Create an SE2 state space.

```
space = stateSpaceSE2
```

```
space =
  stateSpaceSE2 with properties:
      Name: 'SE2'
      StateBounds: [3x2 double]
      NumStateVariables: 3
      WeightXY: 1
      WeightTheta: 0.1000
```

Interpolate half-way between 2 states.

```
state = interpolate(space, [2 10 -pi], [0 -2.5 -pi/4], 0.5)
```

```
state = 1x3
    1.0000    3.7500   -1.9635
```

Interpolate multiple points with a fixed interval.

```
states = interpolate(space, [2 10 -pi], [0 -2.5 -pi/4], [0:0.02:1])
```

```
states = 51x3
    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
:		

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

state1 — Start state for interpolation

3-element real vector

Start state for interpolation, specified as a 3-element real vector.

Data Types: single | double

state2 — End state for interpolation

3-element real vector

End state for interpolation, specified as a 3-element real vector.

Data Types: single | double

ratio — Interpolation ratio

scalar in $[0, 1]$ | N -by-1 array of scalars in $[0, 1]$

Interpolation ratio, specified as a scalar in $[0, 1]$, or an N -by-1 array of scalars in $[0, 1]$. N is the number of interpolation points.

Data Types: single | double

Output Arguments

interpStates — Interpolated states

N -by-3 real-valued matrix

Interpolated states, returned as an N -by-3 real-valued matrix. N is the number of interpolation points specified by the ratio input.

Data Types: single | double

See Also

stateSpaceDubins | stateSpaceReedsShepp | stateSpaceSE2

Introduced in R2019b

enforceStateBounds

Ensure state lie within state bounds

Syntax

```
boundedStates = enforceStateBounds(space, states)
```

Description

`boundedStates = enforceStateBounds(space, states)` reduces `states` to state bounds specified in the `StateBounds` property of the `space` object.

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`

states — Unbounded states

M -by-3 real-valued matrix

Unbounded states, specified as an M -by-3 real-valued matrix. Each row of the matrix corresponds to one incidence of state in the state space.

Data Types: `single` | `double`

Output Arguments

boundedStates — Bounded states

M -by-3 real-valued matrix

Bounded states truncated to the `StateBounds` specified in the state space, returned as an M -by-3 real-valued matrix. M is same as the dimension of the `states` input.

Data Types: `single` | `double`

See Also

`stateSpaceDubins` | `stateSpaceReedsShepp` | `stateSpaceSE2`

Introduced in R2019b

copy

Create copy of state space object

Syntax

```
space2 = copy(space1)
```

Description

`space2 = copy(space1)` creates a copy of the space object, `space2`, from the state space object, `space1`.

Input Arguments

space1 — State space object

`stateSpaceSE2` object | `stateSpaceDubins` object | `stateSpaceReedsShepp` object

State space object, specified as a `stateSpaceSE2`, a `stateSpaceDubins`, or a `stateSpaceReedsShepp` object.

Data Types: object

Output Arguments

space2 — State space object

`stateSpaceSE2` object | `stateSpaceDubins` object | `stateSpaceReedsShepp` object

State space object, returned as a `stateSpaceSE2`, a `stateSpaceDubins`, or a `stateSpaceReedsShepp` object.

Data Types: object

See Also

`stateSpaceDubins` | `stateSpaceReedsShepp` | `stateSpaceSE2`

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

`stateSpaceDubins` | `stateSpaceReedsShepp` | `stateSpaceSE2`

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)` returns a sample state within the `StateBounds` property of the `space` object using a uniform distribution.

`state = sampleUniform(space,numSamples)` returns a number of state samples. The number is equal to `numSamples`.

`state = sampleUniform(space,nearState,distVector,numSamples)` returns the state samples in a subregion defined by `nearState` and `distVector`. The center of the subregion is the `nearState`. `distVector` defines the distance from the two boundaries of the subregion to the center.

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`

numSamples — Number of samples

positive integer

Number of samples, specified as a positive integer.

Data Types: `single` | `double`

nearState — Center of sampling region

3-element real vector

Center of the sampling region, specified as a 3-element real vector.

Data Types: `single` | `double`

distVector — Distance of sampling region boundary form the center

3-element nonnegative vector

Distance of sampling region boundary form the center, specified as a 3-element nonnegative vector.

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

`stateSpaceDubins` | `stateSpaceReedsShepp` | `stateSpaceSE2`

Introduced in R2019b

trajectoryOptimalFrenet

Find optimal trajectory for reference path

Description

The `trajectoryOptimalFrenet` object generates an optimal, feasible, and collision-free trajectory for the reference path.

The `plan` function computes an optimal trajectory between the start and terminal states. The function samples multiple trajectories for each pair of states and chooses a feasible trajectory with the least cost.

Creation

Syntax

```
planner = trajectoryOptimalFrenet(refPath,validator)
planner = trajectoryOptimalFrenet(refPath,validator,Name,Value)
```

Description

`planner = 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(refPath,validator,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/s².

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

<code>cart2frenet</code>	Convert Cartesian states to Frenet states
<code>frenet2cart</code>	Convert Frenet states to Cartesian states
<code>plan</code>	Plan optimal trajectory
<code>show</code>	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 to the state validator.

```
stateValidator.Map = map;
```

Plan and Visualize Trajectory

Create a reference path for the planner to follow.

```
refPath = [0,25;100,25];
```

Declare the cost function handle to prioritize left lane changes.

```
leftLaneChangeCost = @(states)((states(end,2) < refPath(end,2))*10);
```

Initialize the planner object with the reference path, the state validator, and the custom cost function.

```
planner = trajectoryOptimalFrenet(refPath,stateValidator,'CostFunction',leftLaneChangeCost);
```

Assign longitudinal terminal state, lateral deviation, and maximum acceleration values.

```
planner.TerminalStates.Longitudinal = 100;
planner.TerminalStates.Lateral = -10:10:10;
planner.FeasibilityParameters.MaxAcceleration = 10;
```

Trajectory Planning

Initial Frenet state of vehicle.

```
initFrenetState = zeros(1,6);
```

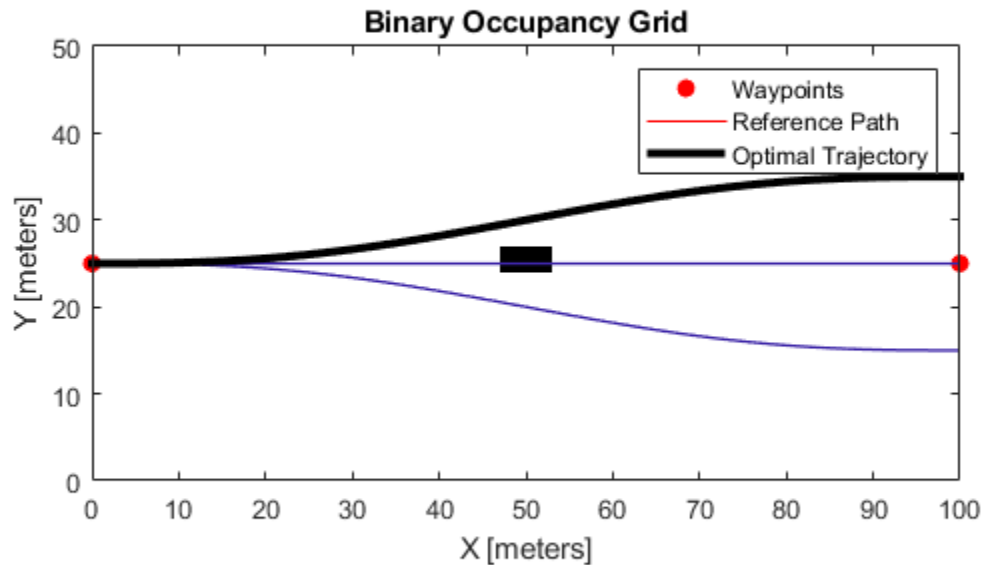
Plan a trajectory from initial Frenet state.

```
plan(planner,initFrenetState);
```

Trajectory Visualization

Visualize the map and the trajectories.

```
show(map)
hold on
show(planner,'Trajectory','all')
```



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(22:24,28:33) = 1;
grid(18:20,37:42) = 1;
grid(29:31,72:77) = 1;
```

Create a `binaryOccupancyMap` with the grid map.

```
map = binaryOccupancyMap(grid);
```

Assign the map to the state validator.

```
statevalidator.Map = map;
```

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, and maximum acceleration values.

```
planner.TerminalStates.Longitudinal = 100;  
planner.FeasibilityParameters.MaxAcceleration = 10;
```

Trajectory Planning

Initial Frenet state of vehicle.

```
initFrenetState = zeros(1,6);
```

Assign the number of partitions for the longitudinal terminal state.

```
planner.NumSegments = 3;
```

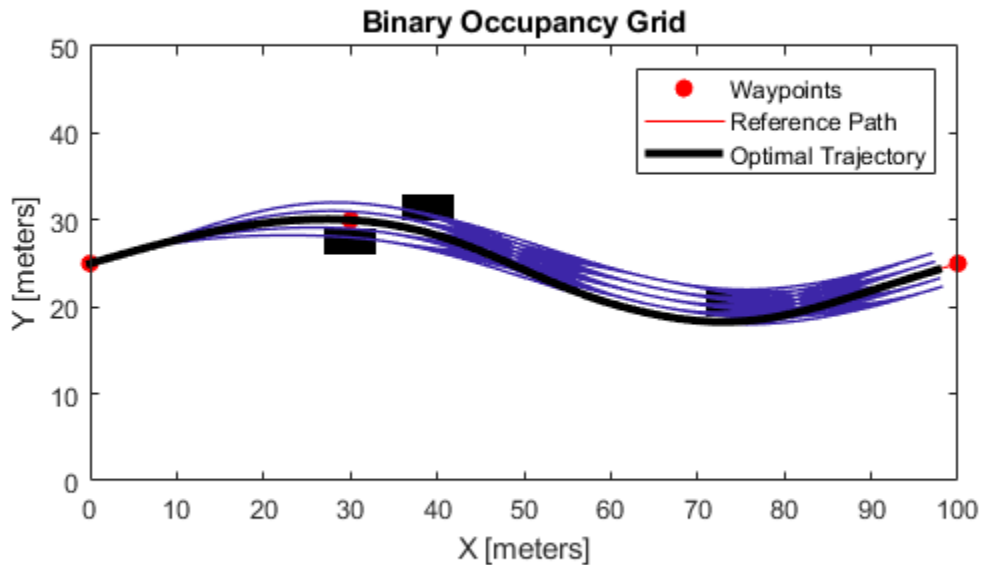
Plan a trajectory from initial Frenet state.

```
plan(planner,initFrenetState);
```

Trajectory Visualization

Visualize the map and the trajectories.

```
show(map)  
hold on  
show(planner,'Trajectory','all')
```



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

nav.StateValidator | plannerHybridAStar | validatorOccupancyMap

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 , θ , κ , v , a] 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 to the state validator.

```
stateValidator.Map = map;
```

Plan and Visualize Trajectory

Create a reference path for the planner to follow.

```
refPath = [0,25;100,25];
```

Declare the cost function handle to prioritize left lane changes.

```
leftLaneChangeCost = @(states)((states(end,2) < refPath(end,2))*10);
```

Initialize the planner object with the reference path, the state validator, and the custom cost function.

```
planner = trajectoryOptimalFrenet(refPath, stateValidator, 'CostFunction', leftLaneChangeCost);
```


Assign longitudinal terminal state, lateral deviation, and maximum acceleration values.

```
planner.TerminalStates.Longitudinal = 100;
planner.TerminalStates.Lateral = -10:10:10;
planner.FeasibilityParameters.MaxAcceleration = 10;
```

Trajectory Planning

Initial Frenet state of vehicle.

```
initFrenetState = zeros(1,6);
```

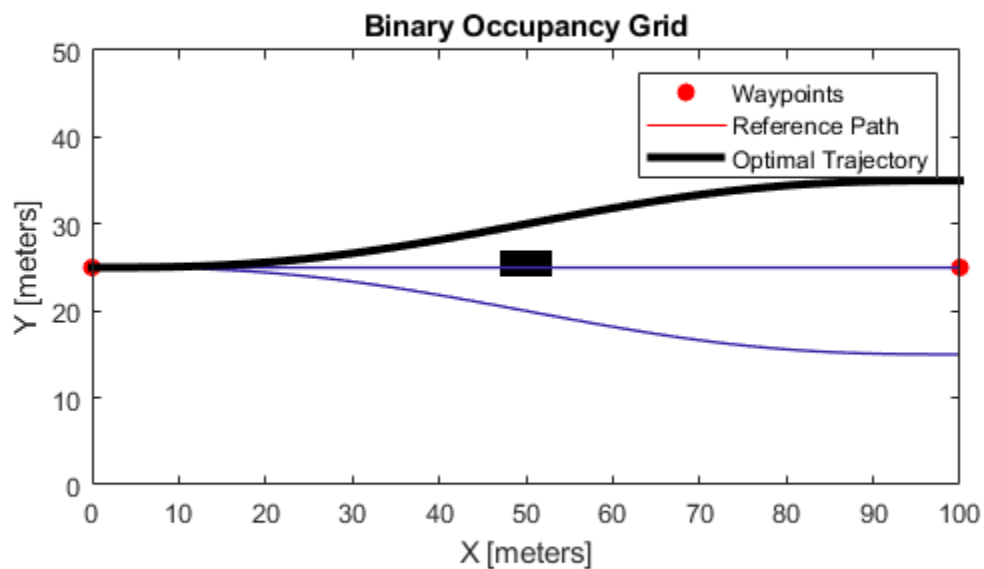
Plan a trajectory from initial Frenet state.

```
plan(planner,initFrenetState);
```

Trajectory Visualization

Visualize the map and the trajectories.

```
show(map)
hold on
show(planner,'Trajectory','all')
```



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(22:24,28:33) = 1;  
grid(18:20,37:42) = 1;  
grid(29:31,72:77) = 1;
```

Create a `binaryOccupancyMap` with the grid map.

```
map = binaryOccupancyMap(grid);
```

Assign the map to the state validator.

```
statevalidator.Map = map;
```

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, and maximum acceleration values.

```
planner.TerminalStates.Longitudinal = 100;  
planner.FeasibilityParameters.MaxAcceleration = 10;
```

Trajectory Planning

Initial Frenet state of vehicle.

```
initFrenetState = zeros(1,6);
```

Assign the number of partitions for the longitudinal terminal state.

```
planner.NumSegments = 3;
```

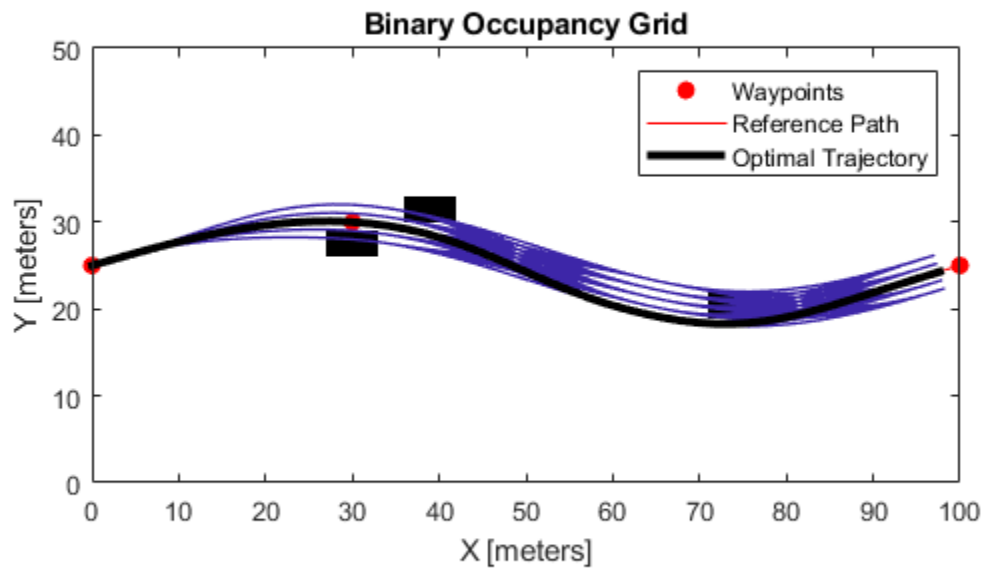
Plan a trajectory from initial Frenet state.

```
plan(planner,initFrenetState);
```

Trajectory Visualization

Visualize the map and the trajectories.

```
show(map)
hold on
show(planner, 'Trajectory', 'all')
```



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 $[x, y, \theta, \kappa, \text{speed}, \text{acceleration}]$.

- x and y specify the position in meters.
- θ specifies the orientation angle in radians.
- κ 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

frenet2cart | trajectoryOptimalFrenet

Introduced in R2019b

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 to the state validator.

```
stateValidator.Map = map;
```

Plan and Visualize Trajectory

Create a reference path for the planner to follow.

```
refPath = [0,25;100,25];
```

Declare the cost function handle to prioritize left lane changes.

```
leftLaneChangeCost = @(states)((states(end,2) < refPath(end,2))*10);
```

Initialize the planner object with the reference path, the state validator, and the custom cost function.

```
planner = trajectoryOptimalFrenet(refPath,stateValidator,'CostFunction',leftLaneChangeCost);
```

Assign longitudinal terminal state, lateral deviation, and maximum acceleration values.

```
planner.TerminalStates.Longitudinal = 100;  
planner.TerminalStates.Lateral = -10:10:10;  
planner.FeasibilityParameters.MaxAcceleration = 10;
```

Trajectory Planning

Initial Frenet state of vehicle.

```
initFrenetState = zeros(1,6);
```

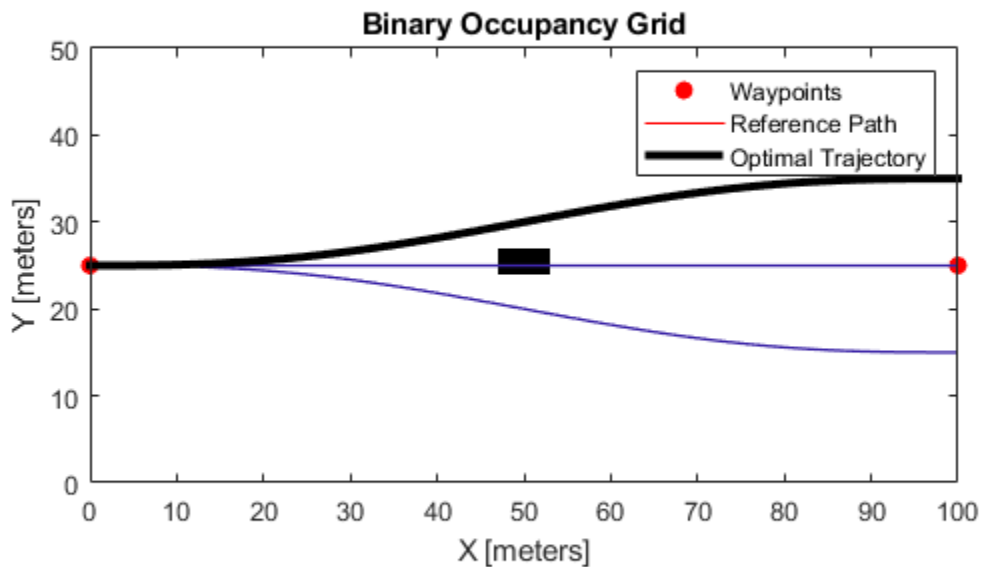
Plan a trajectory from initial Frenet state.

```
plan(planner,initFrenetState);
```

Trajectory Visualization

Visualize the map and the trajectories.

```
show(map)  
hold on  
show(planner,'Trajectory','all')
```



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(22:24,28:33) = 1;
grid(18:20,37:42) = 1;
grid(29:31,72:77) = 1;
```

Create a `binaryOccupancyMap` with the grid map.

```
map = binaryOccupancyMap(grid);
```

Assign the map to the state validator.

```
statevalidator.Map = map;
```

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, and maximum acceleration values.

```
planner.TerminalStates.Longitudinal = 100;
planner.FeasibilityParameters.MaxAcceleration = 10;
```

Trajectory Planning

Initial Frenet state of vehicle.

```
initFrenetState = zeros(1,6);
```

Assign the number of partitions for the longitudinal terminal state.

```
planner.NumSegments = 3;
```

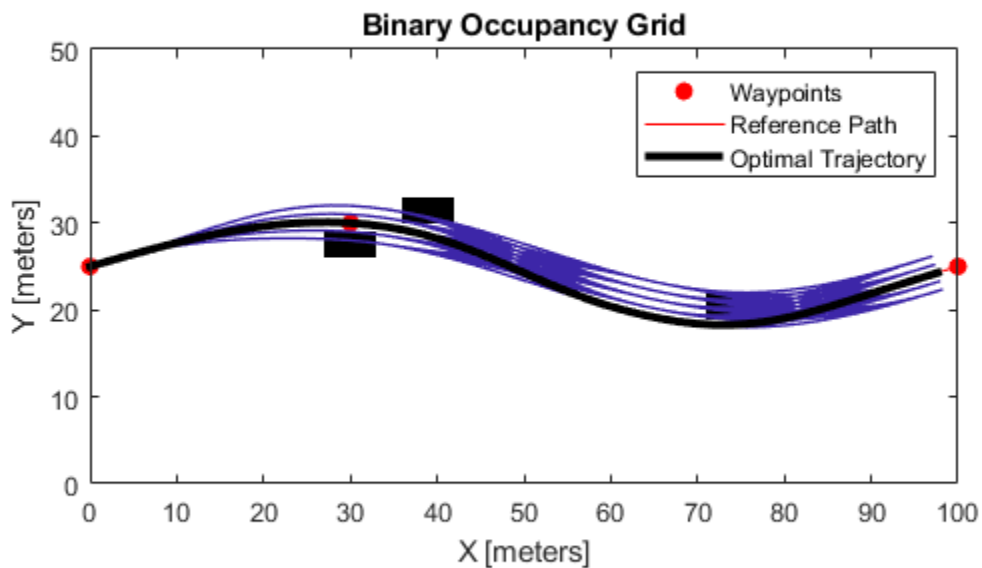
Plan a trajectory from initial Frenet state.

```
plan(planner,initFrenetState);
```

Trajectory Visualization

Visualize the map and the trajectories.

```
show(map)
hold on
show(planner, 'Trajectory', 'all')
```



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

[cart2frenet](#) | [trajectoryOptimalFrenet](#)

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 to the state validator.

```
stateValidator.Map = map;
```

Plan and Visualize Trajectory

Create a reference path for the planner to follow.

```
refPath = [0,25;100,25];
```

Declare the cost function handle to prioritize left lane changes.

```
leftLaneChangeCost = @(states)((states(end,2) < refPath(end,2))*10);
```

Initialize the planner object with the reference path, the state validator, and the custom cost function.

```
planner = trajectoryOptimalFrenet(refPath,stateValidator,'CostFunction',leftLaneChangeCost);
```

Assign longitudinal terminal state, lateral deviation, and maximum acceleration values.

```
planner.TerminalStates.Longitudinal = 100;  
planner.TerminalStates.Lateral = -10:10:10;  
planner.FeasibilityParameters.MaxAcceleration = 10;
```

Trajectory Planning

Initial Frenet state of vehicle.

```
initFrenetState = zeros(1,6);
```

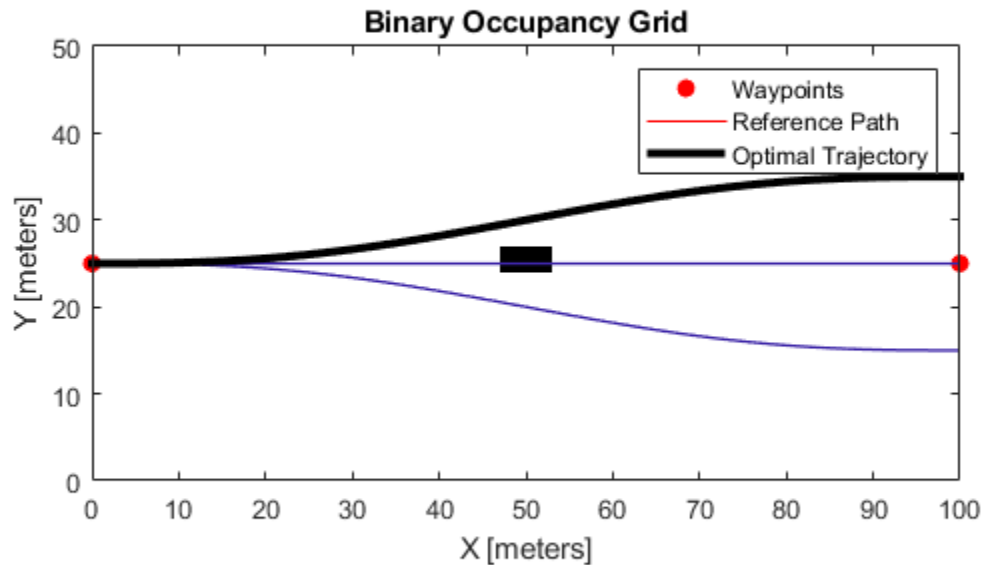
Plan a trajectory from initial Frenet state.

```
plan(planner,initFrenetState);
```

Trajectory Visualization

Visualize the map and the trajectories.

```
show(map)  
hold on  
show(planner,'Trajectory','all')
```



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(22:24,28:33) = 1;
grid(18:20,37:42) = 1;
grid(29:31,72:77) = 1;
```

Create a `binaryOccupancyMap` with the grid map.

```
map = binaryOccupancyMap(grid);
```

Assign the map to the state validator.

```
statevalidator.Map = map;
```

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, and maximum acceleration values.

```
planner.TerminalStates.Longitudinal = 100;  
planner.FeasibilityParameters.MaxAcceleration = 10;
```

Trajectory Planning

Initial Frenet state of vehicle.

```
initFrenetState = zeros(1,6);
```

Assign the number of partitions for the longitudinal terminal state.

```
planner.NumSegments = 3;
```

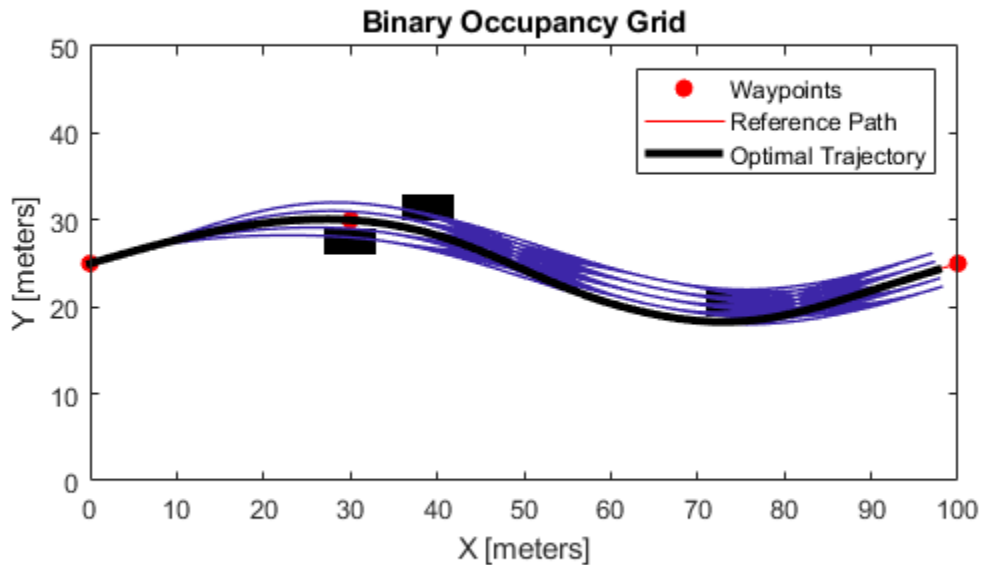
Plan a trajectory from initial Frenet state.

```
plan(planner,initFrenetState);
```

Trajectory Visualization

Visualize the map and the trajectories.

```
show(map)  
hold on  
show(planner,'Trajectory','all')
```



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 $[s, ds/dt, d^2s/dt^2, l, dl/ds, d^2l/ds^2]$.

- 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.

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

show | trajectoryOptimalFrenet

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 to the state validator.

```
stateValidator.Map = map;
```

Plan and Visualize Trajectory

Create a reference path for the planner to follow.

```
refPath = [0,25;100,25];
```

Declare the cost function handle to prioritize left lane changes.


```
leftLaneChangeCost = @(states)((states(end,2) < refPath(end,2))*10);
```

Initialize the planner object with the reference path, the state validator, and the custom cost function.

```
planner = trajectoryOptimalFrenet(refPath,stateValidator,'CostFunction',leftLaneChangeCost);
```

Assign longitudinal terminal state, lateral deviation, and maximum acceleration values.

```
planner.TerminalStates.Longitudinal = 100;  
planner.TerminalStates.Lateral = -10:10:10;  
planner.FeasibilityParameters.MaxAcceleration = 10;
```

Trajectory Planning

Initial Frenet state of vehicle.

```
initFrenetState = zeros(1,6);
```

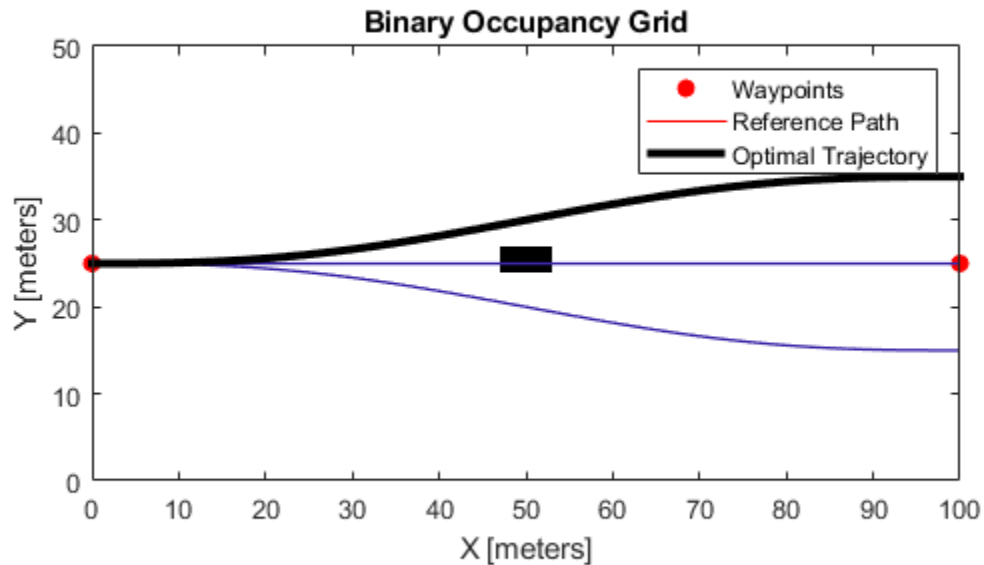
Plan a trajectory from initial Frenet state.

```
plan(planner,initFrenetState);
```

Trajectory Visualization

Visualize the map and the trajectories.

```
show(map)  
hold on  
show(planner,'Trajectory','all')
```



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(22:24,28:33) = 1;
grid(18:20,37:42) = 1;
grid(29:31,72:77) = 1;
```

Create a `binaryOccupancyMap` with the grid map.

```
map = binaryOccupancyMap(grid);
```

Assign the map to the state validator.

```
statevalidator.Map = map;
```

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, and maximum acceleration values.

```
planner.TerminalStates.Longitudinal = 100;  
planner.FeasibilityParameters.MaxAcceleration = 10;
```

Trajectory Planning

Initial Frenet state of vehicle.

```
initFrenetState = zeros(1,6);
```

Assign the number of partitions for the longitudinal terminal state.

```
planner.NumSegments = 3;
```

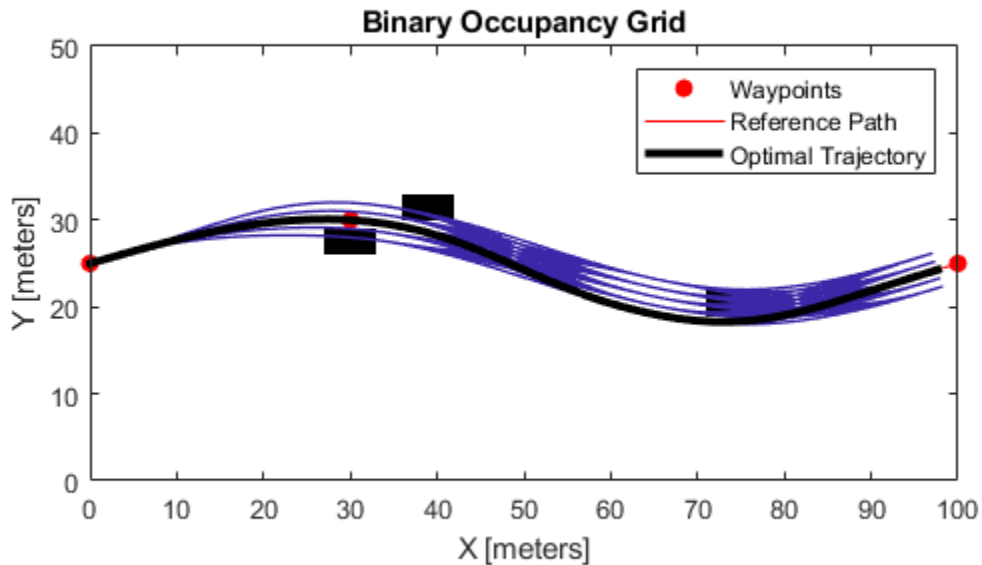
Plan a trajectory from initial Frenet state.

```
plan(planner,initFrenetState);
```

Trajectory Visualization

Visualize the map and the trajectories.

```
show(map)  
hold on  
show(planner,'Trajectory','all')
```



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.

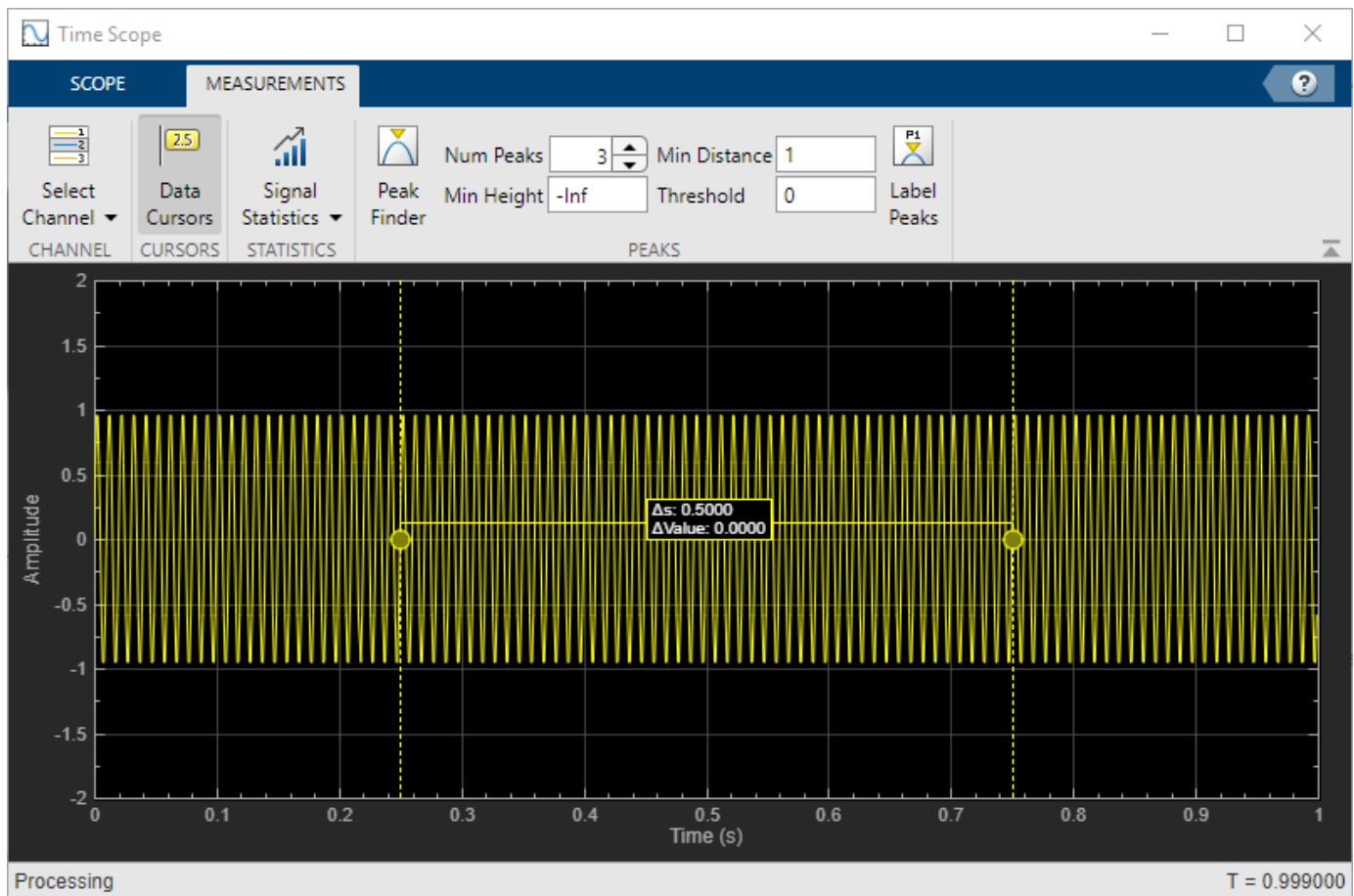
See Also`plan` | `trajectoryOptimalFrenet`**Introduced in R2019b**

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.

Use “Object Functions” on page 2-843 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

Sampling rate of the input signal, in hertz, specified as a finite numeric scalar.

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, the object uses the same sample rate for all inputs.

You can set this property only when creating the object.

UI 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)`

UI 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'.

UI 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.

UI 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.

UI 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.

Tunable: Yes


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.

Example: `scope.LayoutDimensions = [2,4]`

UI 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'`

Specify the units used to describe the x-axis (time axis). You can select one of the following options:

- `'seconds'` — In this mode, the scope always displays the units on the x-axis as seconds. The scope shows the word `Time(s)` on the x-axis.
- `'none'` — In this mode, the scope does not display any units on the x-axis. The scope only shows the word `Time` on the x-axis.

UI 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`

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.

UI 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.

UI 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.

UI Use

On the scope window, click on  to maximize axes, hiding all labels and inseting 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.

UI 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 410 pixels and height of 300 pixels. The exact values of the position depend on your screen resolution.

ChannelNames — Channel names

empty cell array (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`.

UI 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.

UI 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.

UI 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.

UI 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.

UI Use

On the **Scope** tab, click **Settings**. Under **Display and Labels**, select **Show Legend**.

Data Types: logical

ShowGrid — Grid visibility

false (default) | true

Set this property to `true` to show grid lines on the plot.

UI 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.

UI 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.

<code>hide</code>	Hide scope window
<code>show</code>	Display scope window
<code>isVisible</code>	Determine visibility of scope
<code>generateScript</code>	Generate MATLAB script to create scope with current settings
<code>step</code>	Run System object algorithm
<code>release</code>	Release resources and allow changes to System object property values and input characteristics
<code>reset</code>	Reset internal states of System object

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
```

See Also

Functions

`generateScript` | `hide` | `isVisible` | `show`

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`

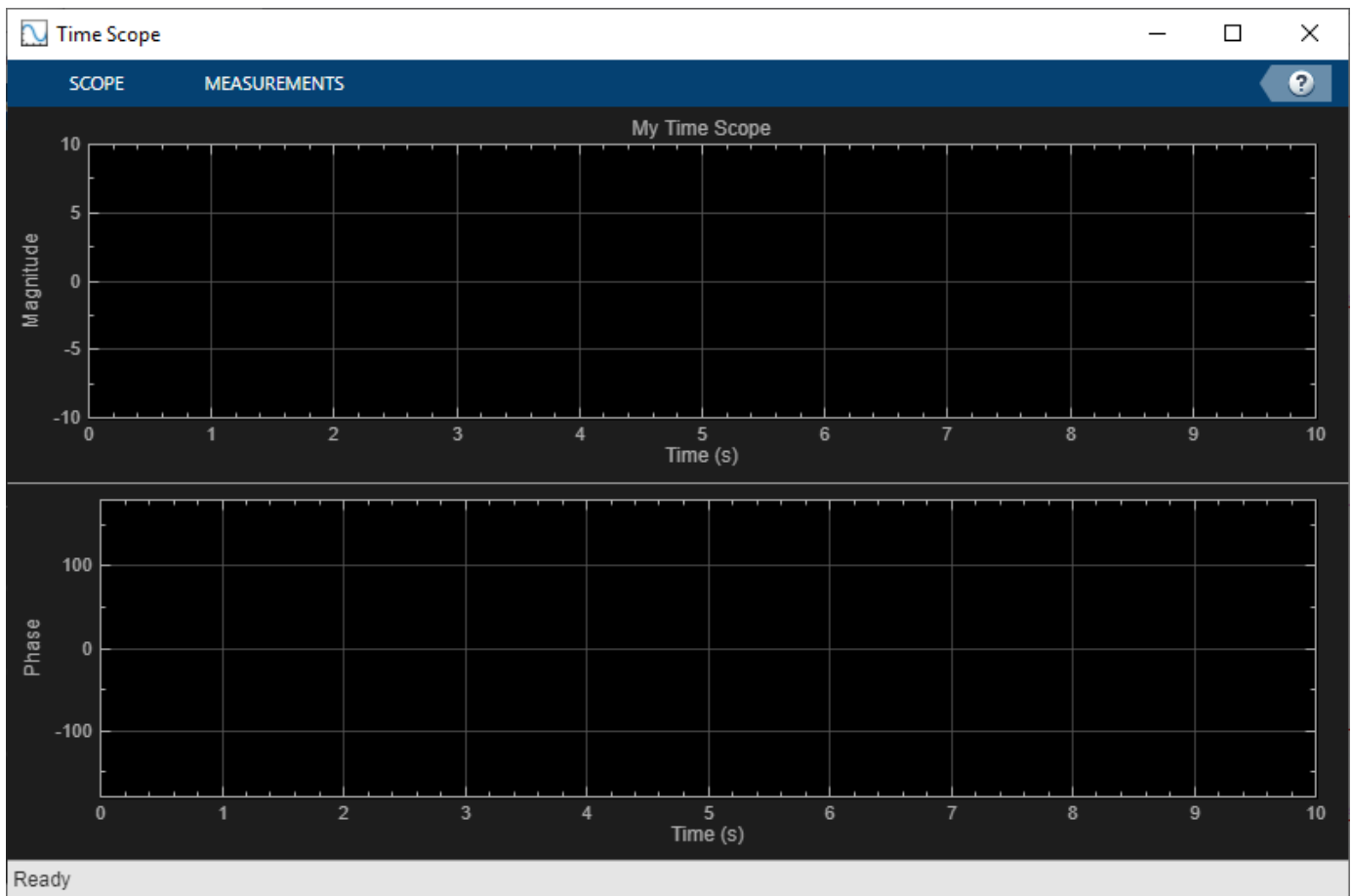
Generate MATLAB script after making changes to the `timescope` object in the scope window.

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 | isVisible | show

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

`generateScript` | `isVisible` | `show`

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

`generateScript` | `hide` | `show`

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

`generateScript` | `hide` | `isVisible`

Objects

`timescope`

Introduced in R2020a

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 vehicle cost 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

`isStateValid` Check if state is valid
`isMotionValid` Check if path between states is valid

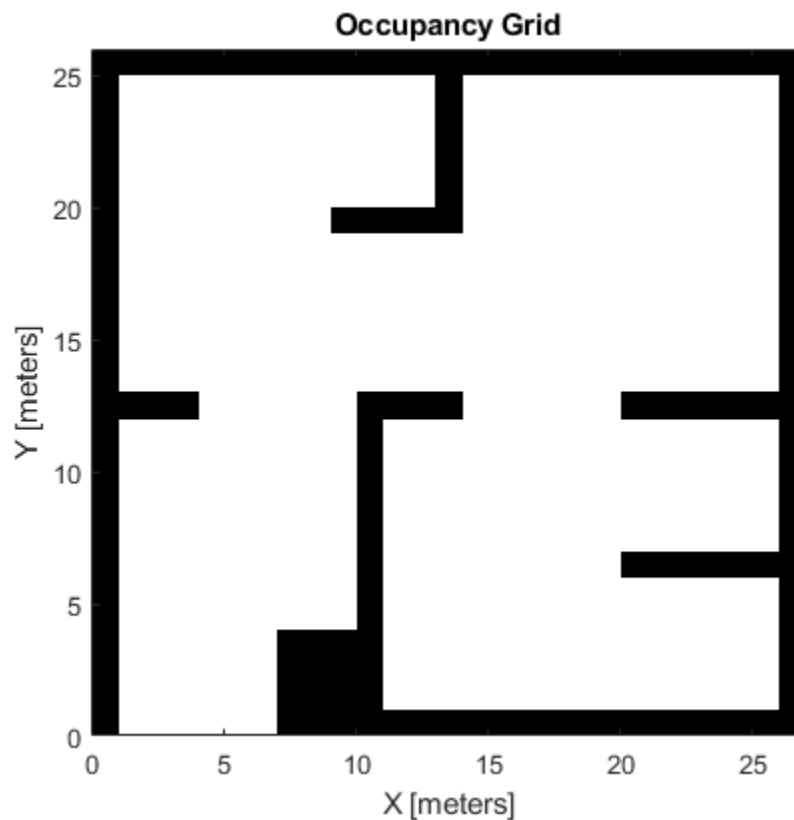
Examples

Validate Path Through Occupancy Map Environment

This example shows how to validate paths through an environment.

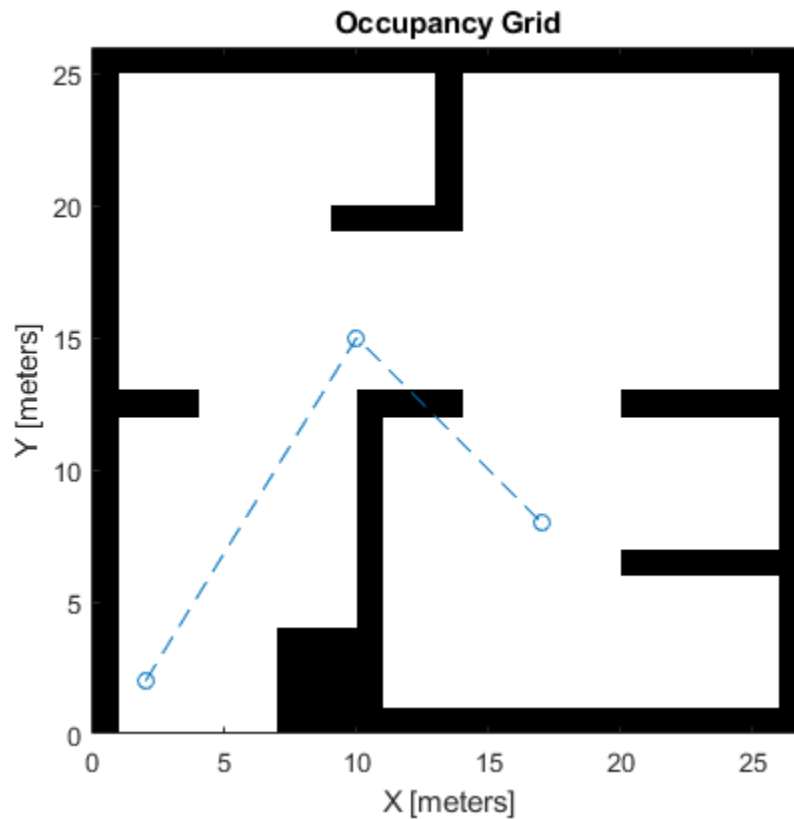
Load example maps. Use the simple map to create a binary occupancy map.

```
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
```

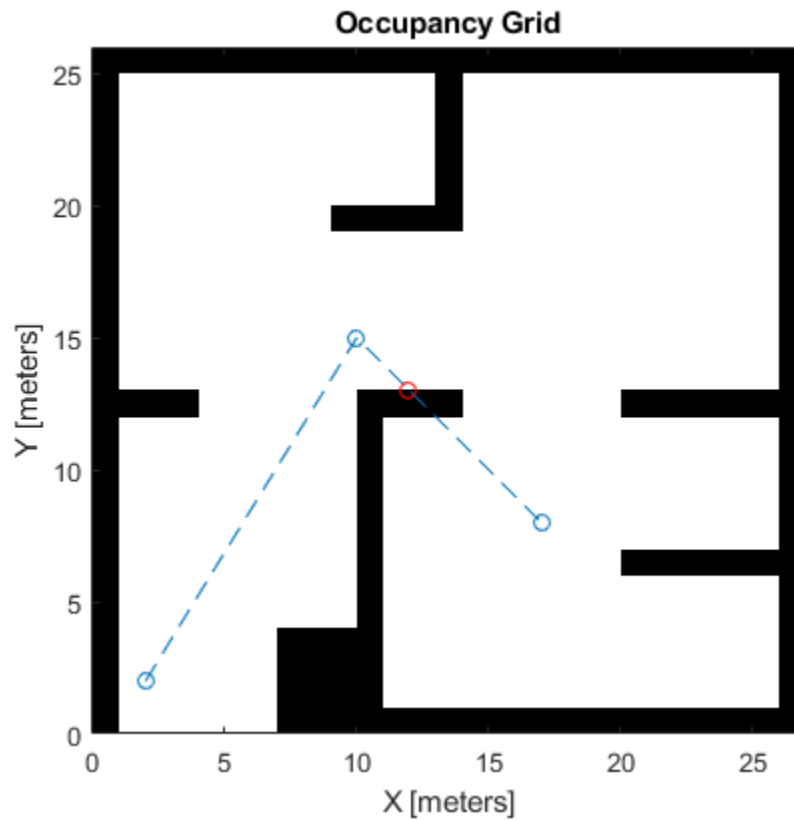
Check the motion between each sequential path states. The `isMotionValid` function interpolates along the path between states. If a path segment is invalid, plot the last valid point along the path.

```
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 you create the object. For example:

```
validator = validatorOccupancyMap('Map',occMap,'XYIndices',[4 5])
```

See Also

[nav.StateSpace](#) | [nav.StateValidator](#) | [stateSpaceSE2](#)

Introduced in R2019b

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

isStateValid Check if state is valid

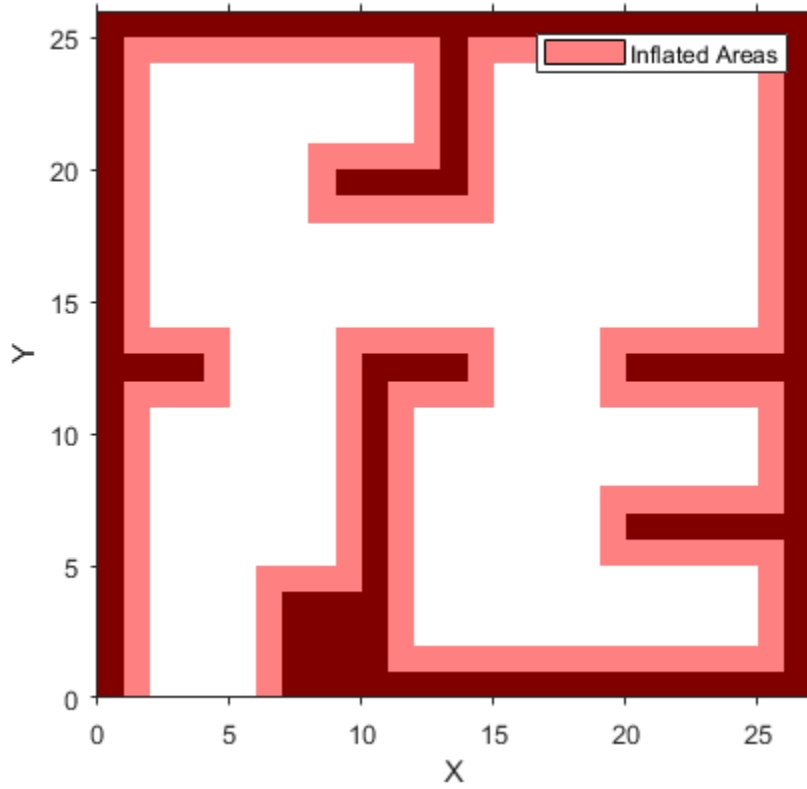
isMotionValid Check if path between states is valid

Examples**Validate Path Through Vehicle Costmap Environment**

This example shows how to validate paths through an environment.

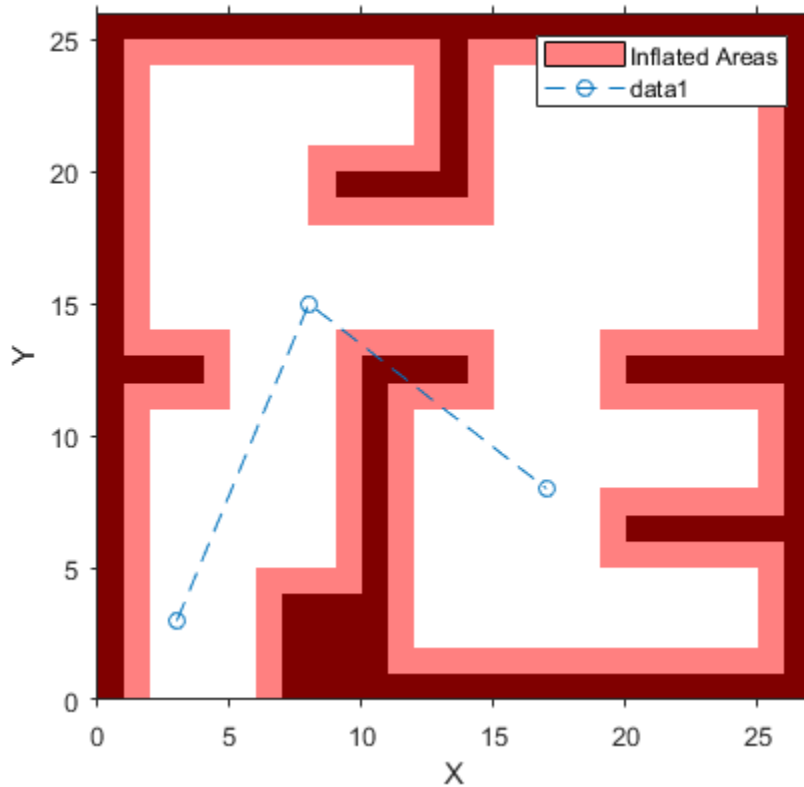
Load example maps. Use the simple map to create a vehicle cost map. Specify an inflation radius of 1 meter.

```
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)
```

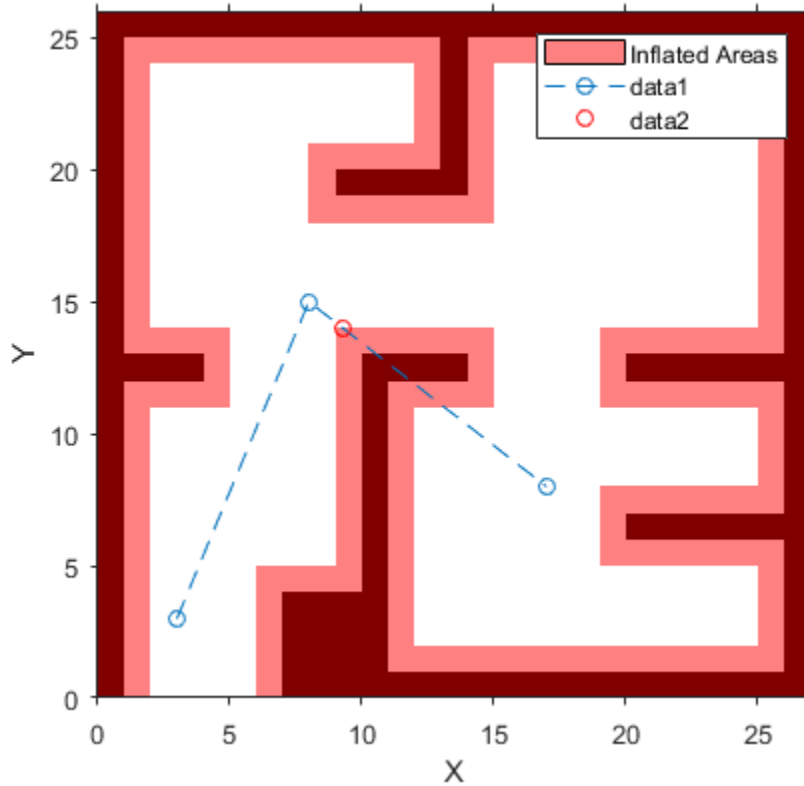
```
isValid = 3x1 logical array
```

```
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

[nav.StateSpace](#) | [nav.StateValidator](#) | [stateSpaceSE2](#) | [validatorOccupancyMap](#)

Introduced in R2019b

isMotionValid

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.

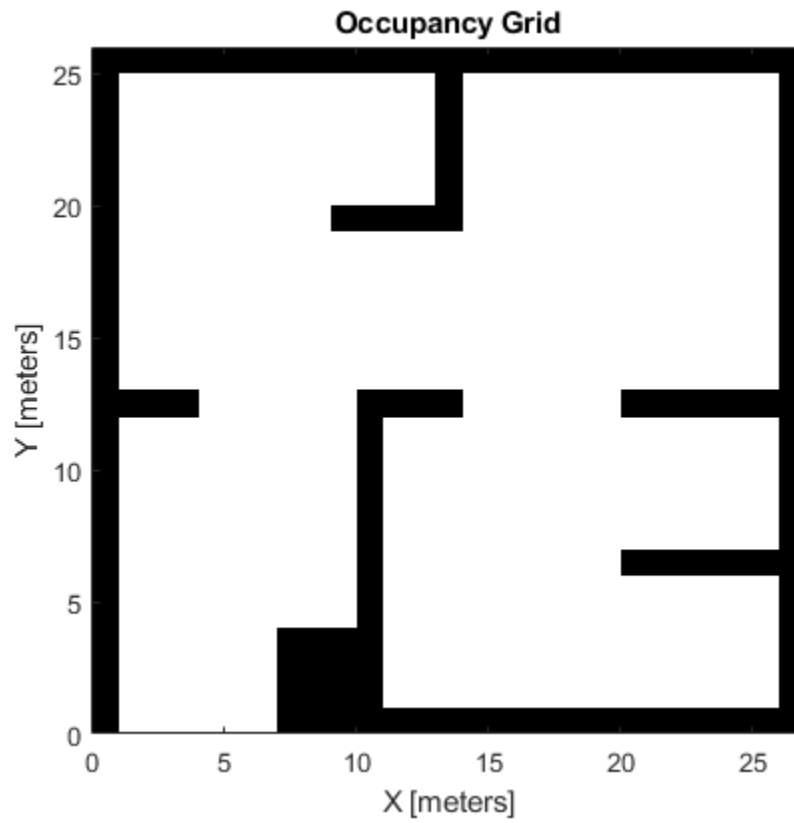
Examples

Validate Path Through Occupancy Map Environment

This example shows how to validate paths through an environment.

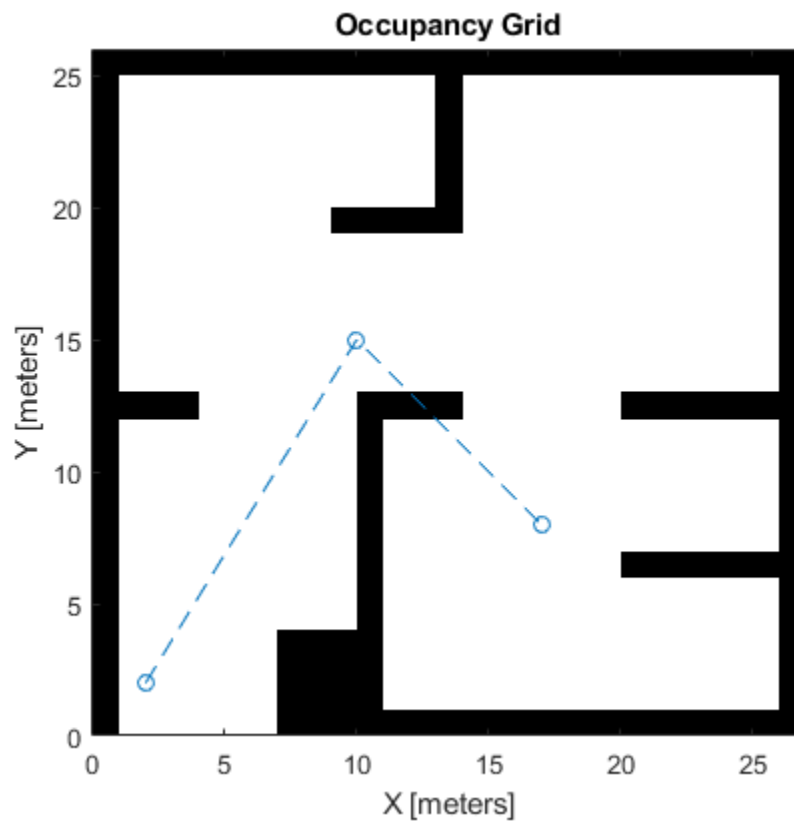
Load example maps. Use the simple map to create a binary occupancy map.

```
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)
```

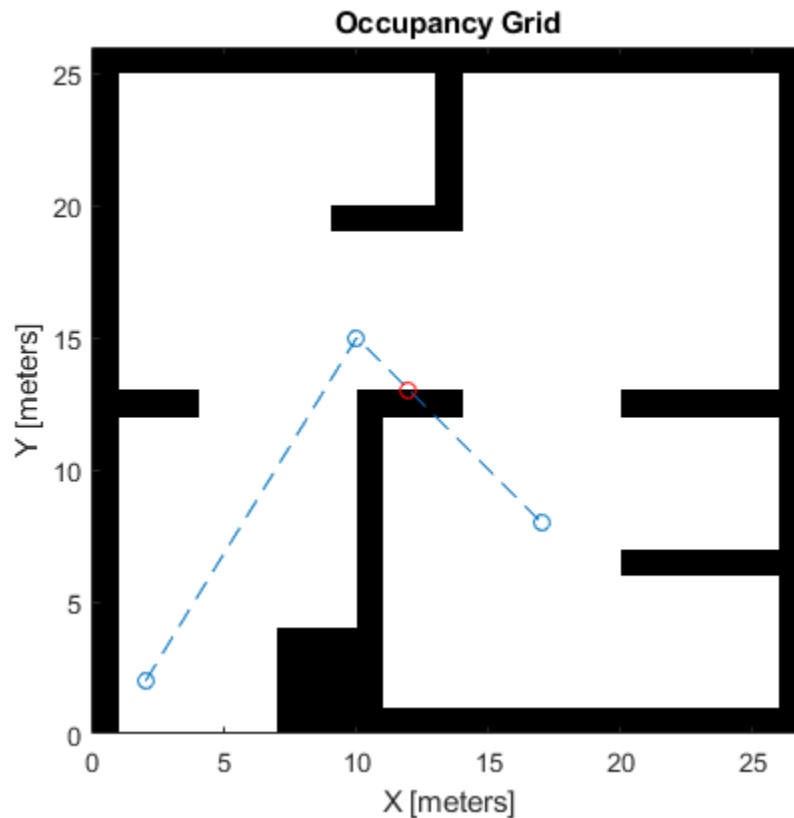
```
isValid = 3x1 logical array
```

```
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
```

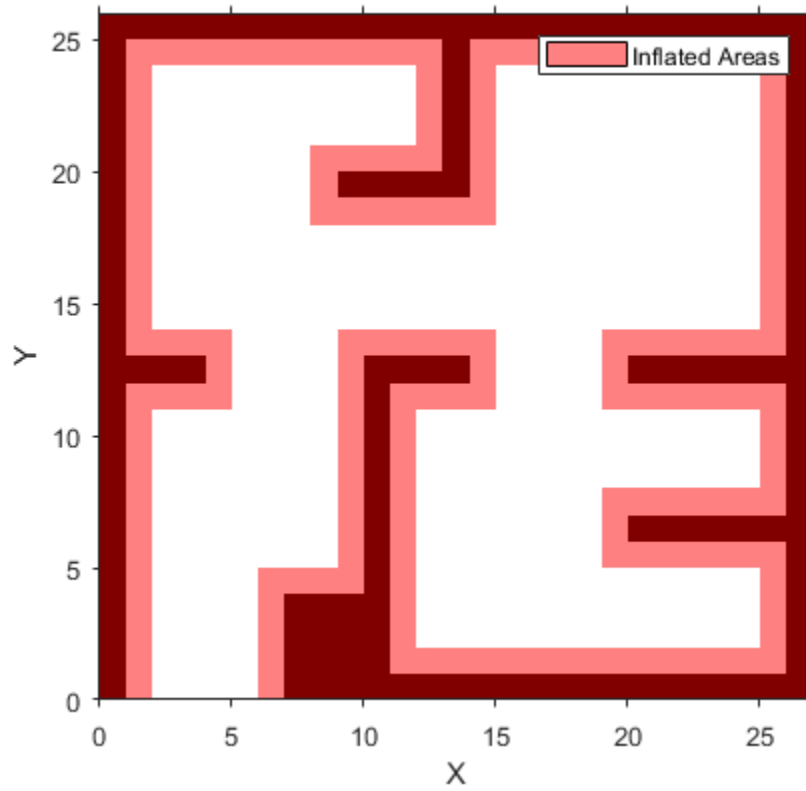


Validate Path Through Vehicle Costmap Environment

This example shows how to validate paths through an environment.

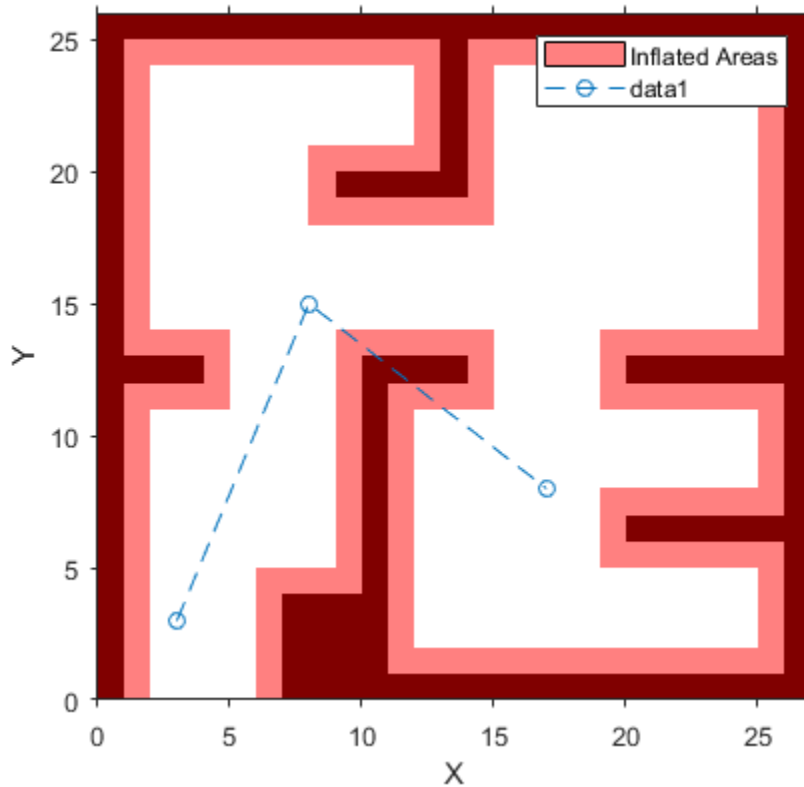
Load example maps. Use the simple map to create a vehicle cost map. Specify an inflation radius of 1 meter.

```
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
```

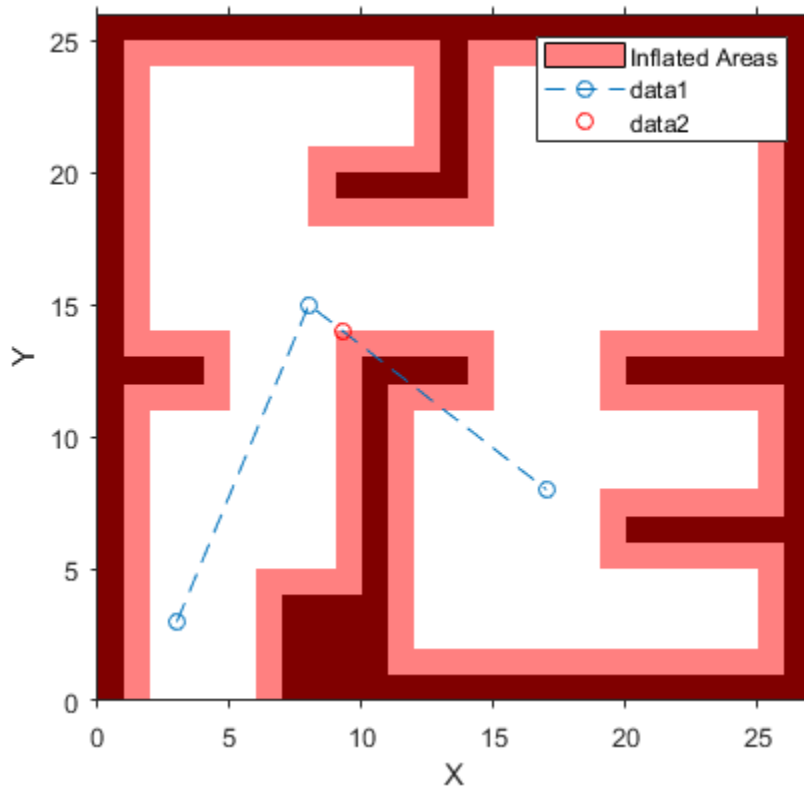
Check the motion between each sequential path states. The `isMotionValid` function interpolates along the path between states. If a path segment is invalid, plot the last valid point along the path.

```
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

```



Input Arguments

validatorObj — State validator object

object of subclass of `nav.StateValidator`

State validator object, specified as an object of subclass of `nav.StateValidator`. For provided state validator objects, see `validatorOccupancyMap` or `validatorVehicleCostmap`.

state1 — Initial state positions

n -element vector | m -by- n matrix of row vectors

Initial state positions, 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.

state2 — Final state positions

n -element vector | m -by- n matrix of row vectors

Final state positions, 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 1s and 0s

Valid states, specified as a *m*-element vector of 1s and 0s.

lastValid — Final valid state along path

n-element vector | *m*-by-*n* matrix of row vectors

Final valid state along path, 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`.

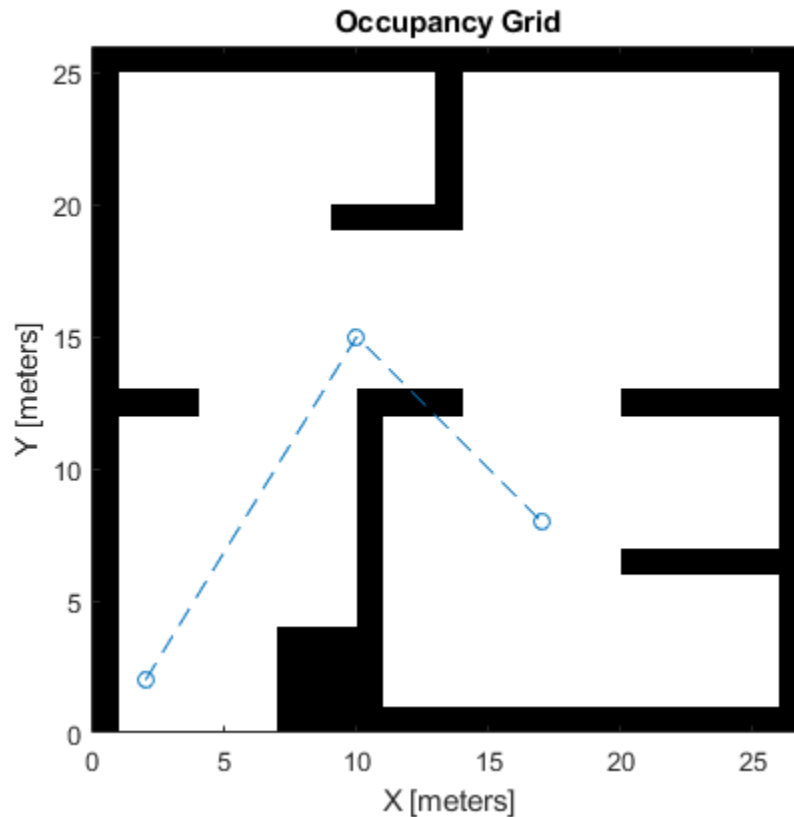
See Also

`isStateValid` | `nav.StateSpace` | `nav.StateValidator` | `stateSpaceSE2`

Introduced in R2019b

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
```

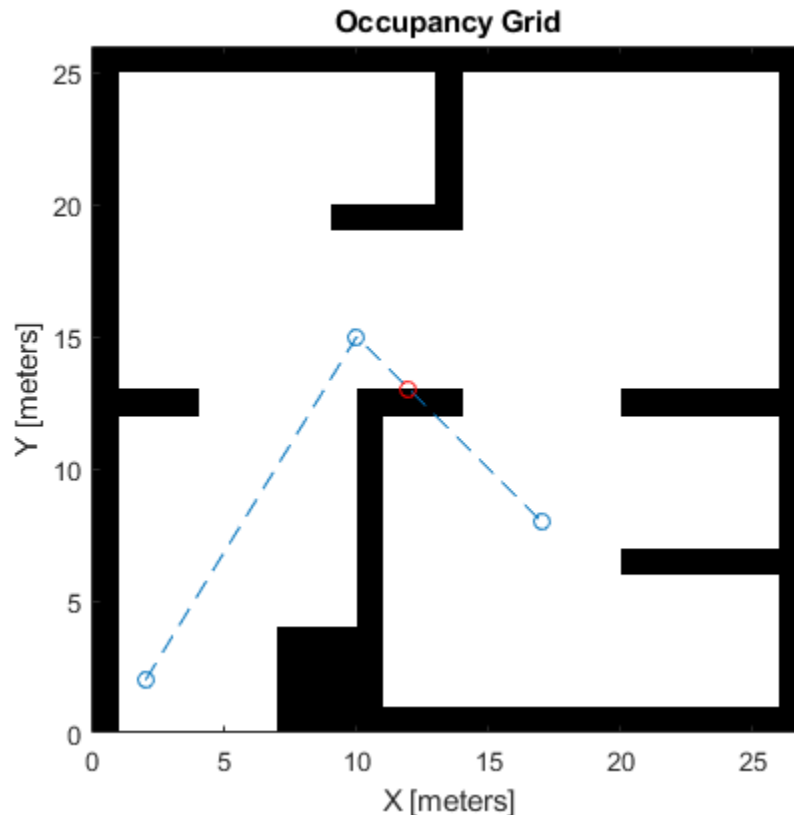
```
1
1
1
```

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
end
hold off

```



Validate Path Through Vehicle Costmap Environment

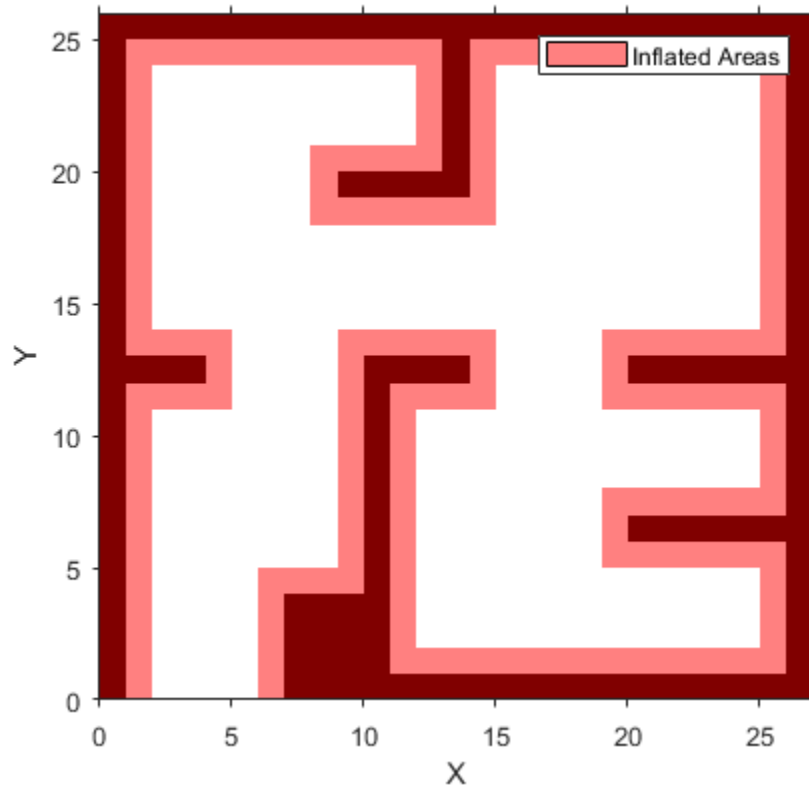
This example shows how to validate paths through an environment.

Load example maps. Use the simple map to create a vehicle cost map. Specify an inflation radius of 1 meter.

```

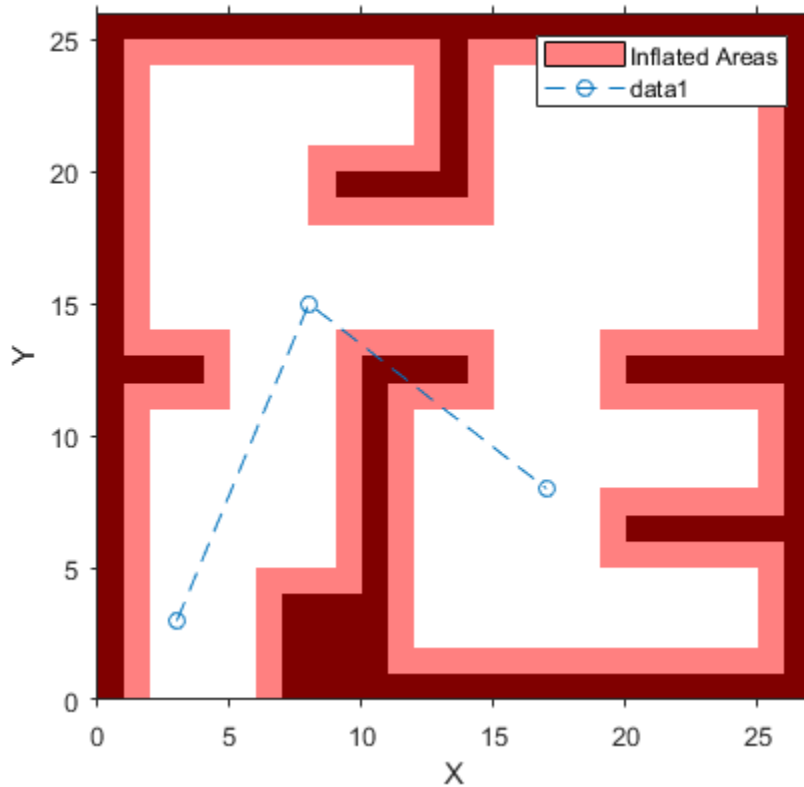
load exampleMaps.mat
map = vehicleCostmap(double(simpleMap));
map.CollisionChecker = inflationCollisionChecker("InflationRadius",1);
plot(map)

```



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
```

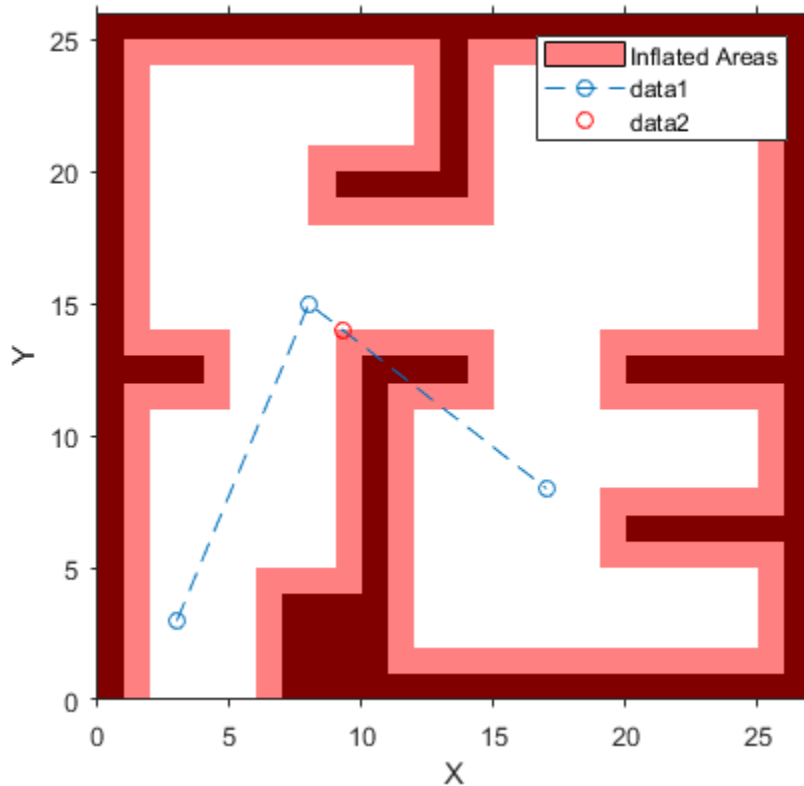
Check the motion between each sequential path states. The `isMotionValid` function interpolates along the path between states. If a path segment is invalid, plot the last valid point along the path.

```
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

```



Input Arguments

validatorObj — State validator object

object of subclass of `nav.StateValidator`

State validator object, specified as an object of 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 1s and 0s

Valid states, specified as a m -element vector of 1s and 0s.

See Also

[isMotionValid](#) | [nav.StateSpace](#) | [nav.StateValidator](#) | [stateSpaceSE2](#)

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.

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? \(MATLAB\)](#).

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 (MATLAB).

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 — Climbrate at each waypoint (m/s)

N -element real vector

Climbrate 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/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 `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

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=2x2 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

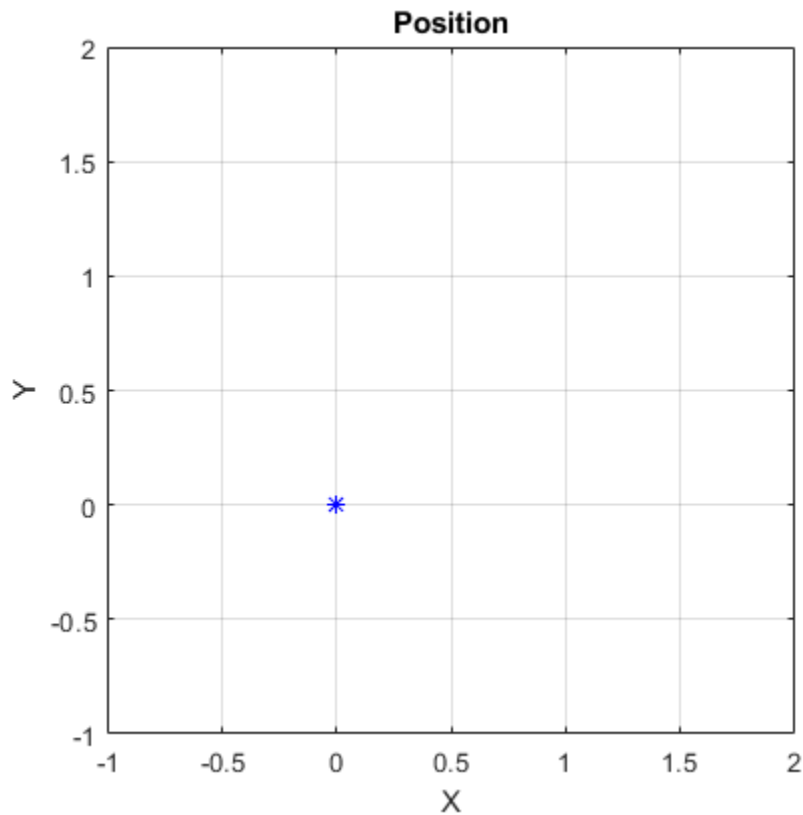
toa = 0:4; % time of arrival

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
```

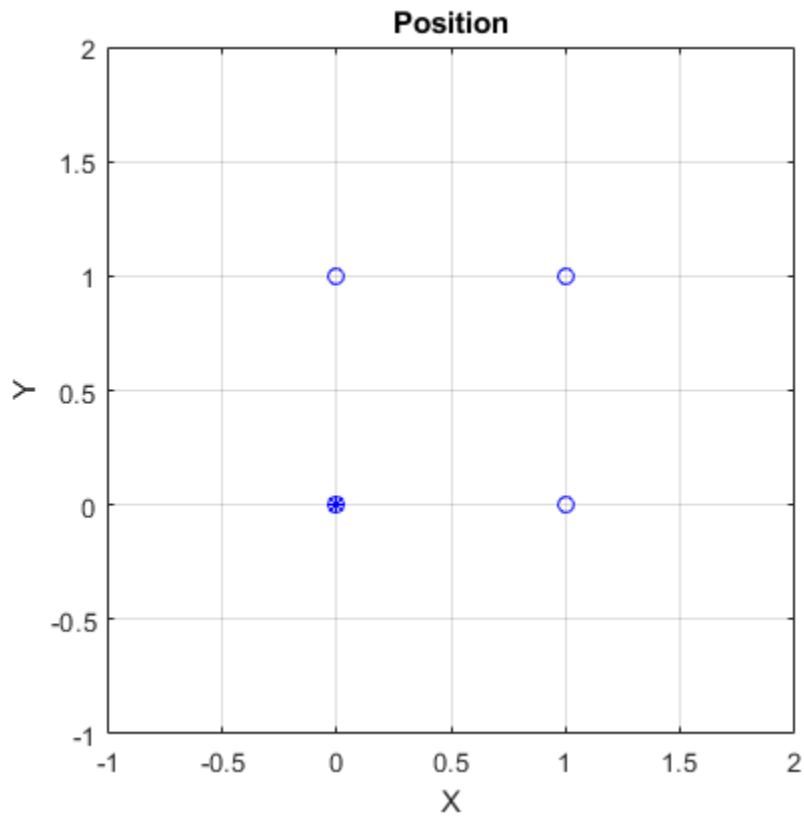


In a loop, step through the trajectory to output the current position and current orientation. Plot the current position and log the orientation. Use `pause` to mimic real-time processing.

```
orientationLog = zeros(toa(end)*trajectory.SampleRate,1,'quaternion');
count = 1;
while ~isDone(trajectory)
    [currentPosition,orientationLog(count)] = trajectory();

    plot(currentPosition(1),currentPosition(2),'bo')

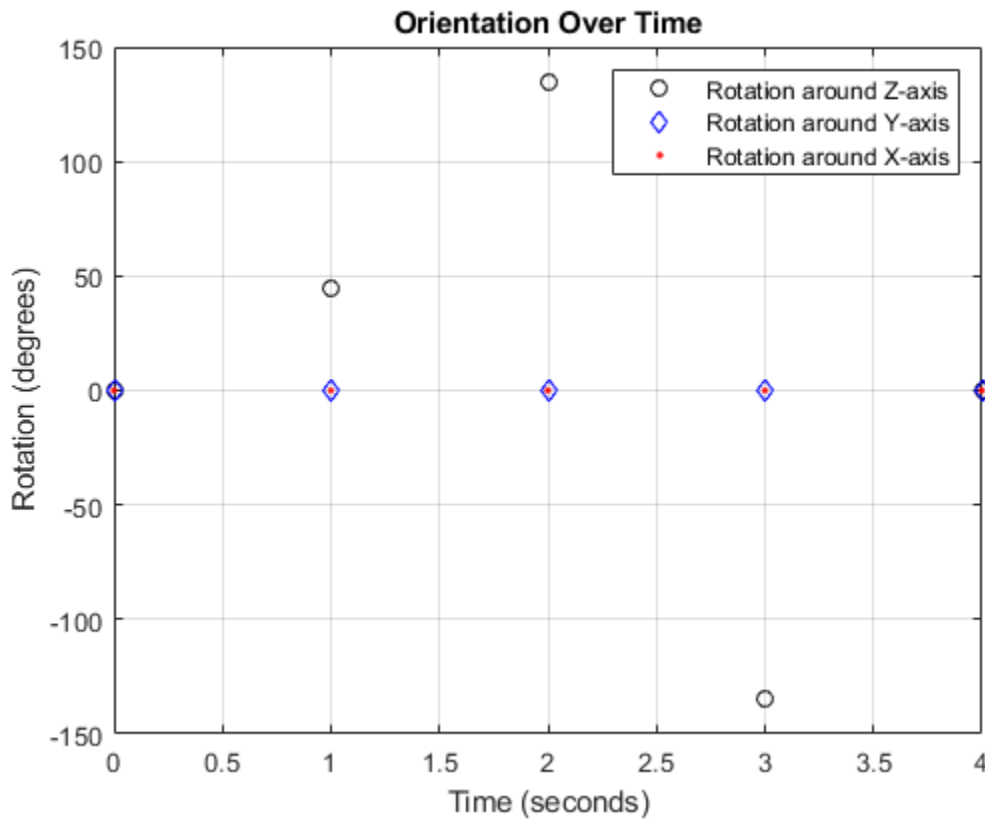
    pause(trajectory.SamplesPerFrame/trajectory.SampleRate)
    count = count + 1;
end
hold off
```



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

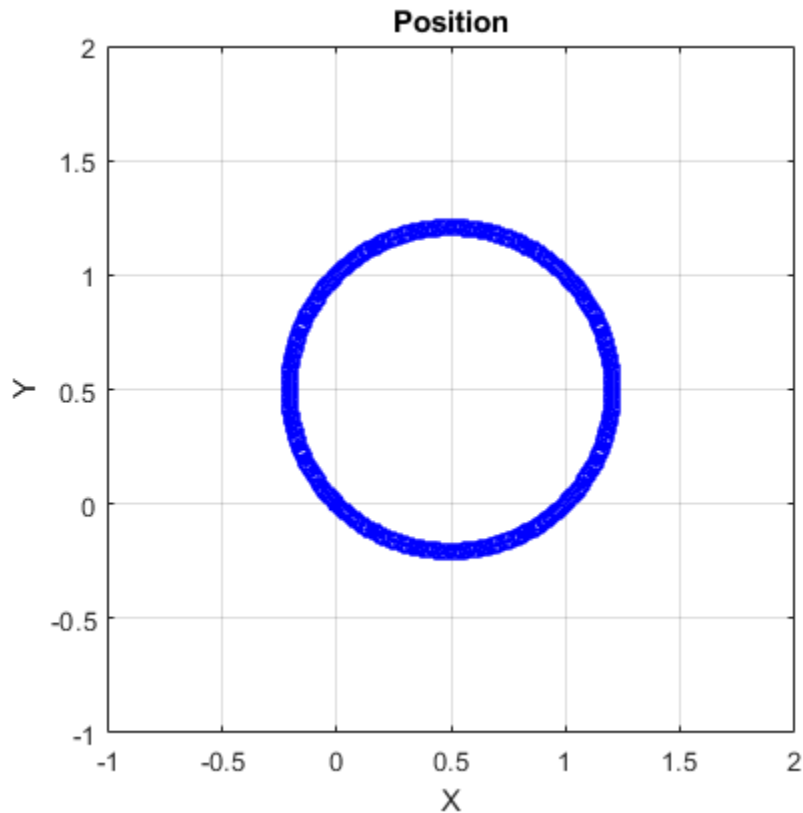
orientationLog = zeros(toa(end)*trajectory.SampleRate,1,'quaternion');
count = 1;
while ~isDone(trajectory)
    [currentPosition,orientationLog(count)] = trajectory();

    plot(currentPosition(1),currentPosition(2),'bo')
```

```

    pause(trajjectory.SamplesPerFrame/trajjectory.SampleRate)
    count = count + 1;
end
hold off

```



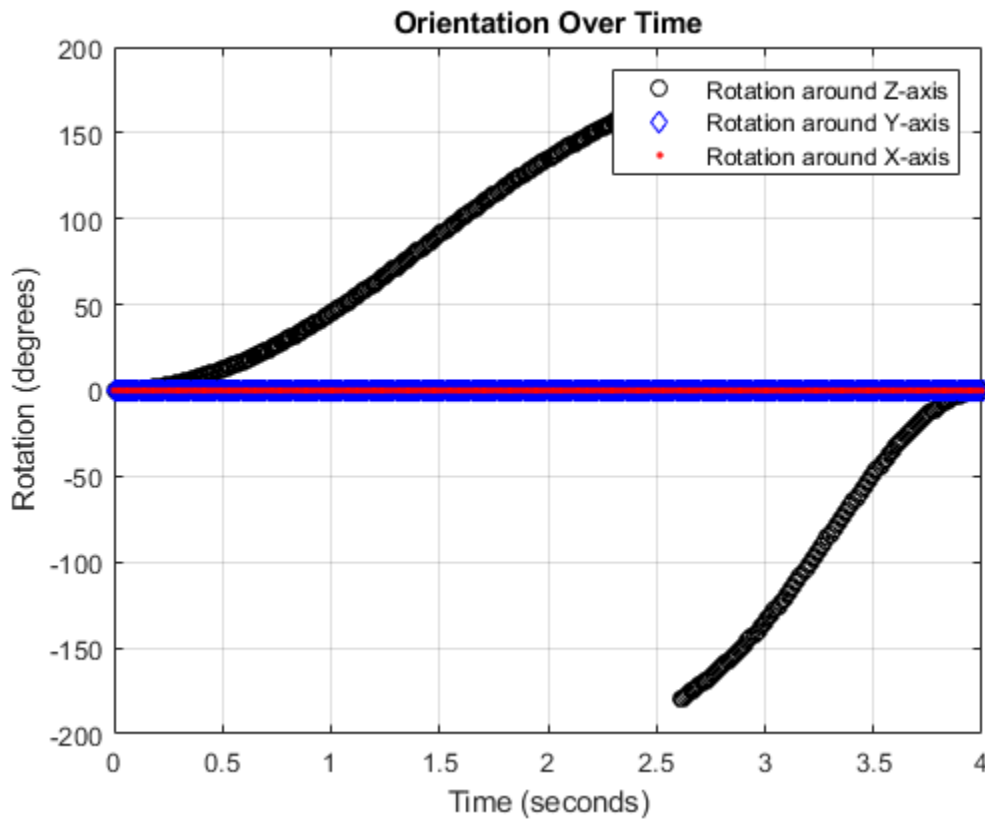
The trajectory output now appears circular. This is because the `waypointTrajectory System object™` minimizes the acceleration and angular velocity when interpolating, which results in smoother, more realistic motions in most scenarios.

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/trajjectory.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);

```

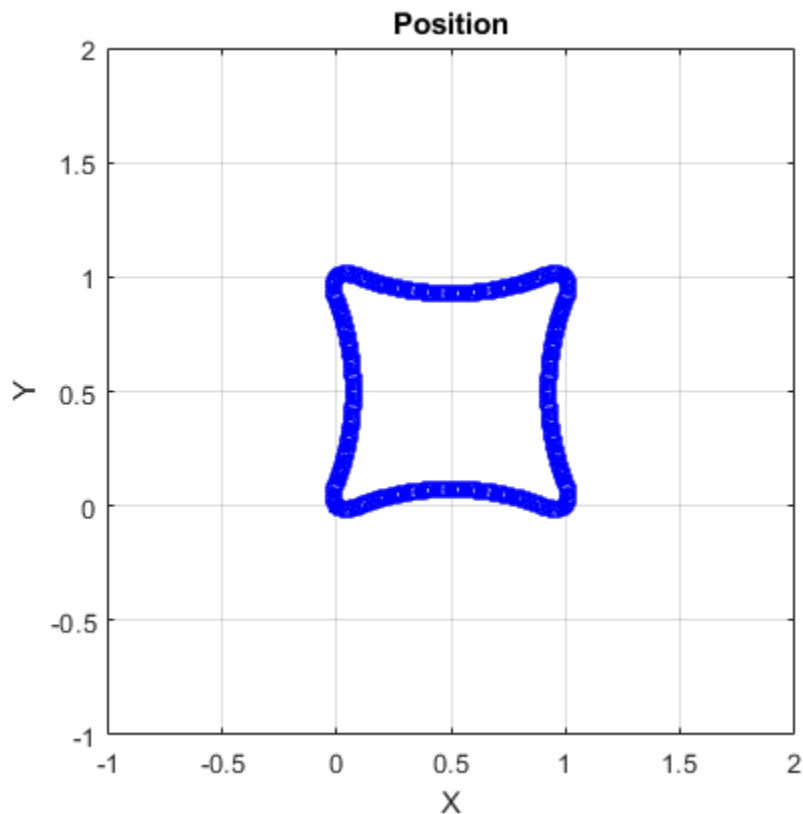
Create a figure and plot the initial position of the platform. In a loop, step through the trajectory to output the current position and current orientation. Plot the current position and log the orientation. Use `pause` to mimic real-time processing.

```
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
```

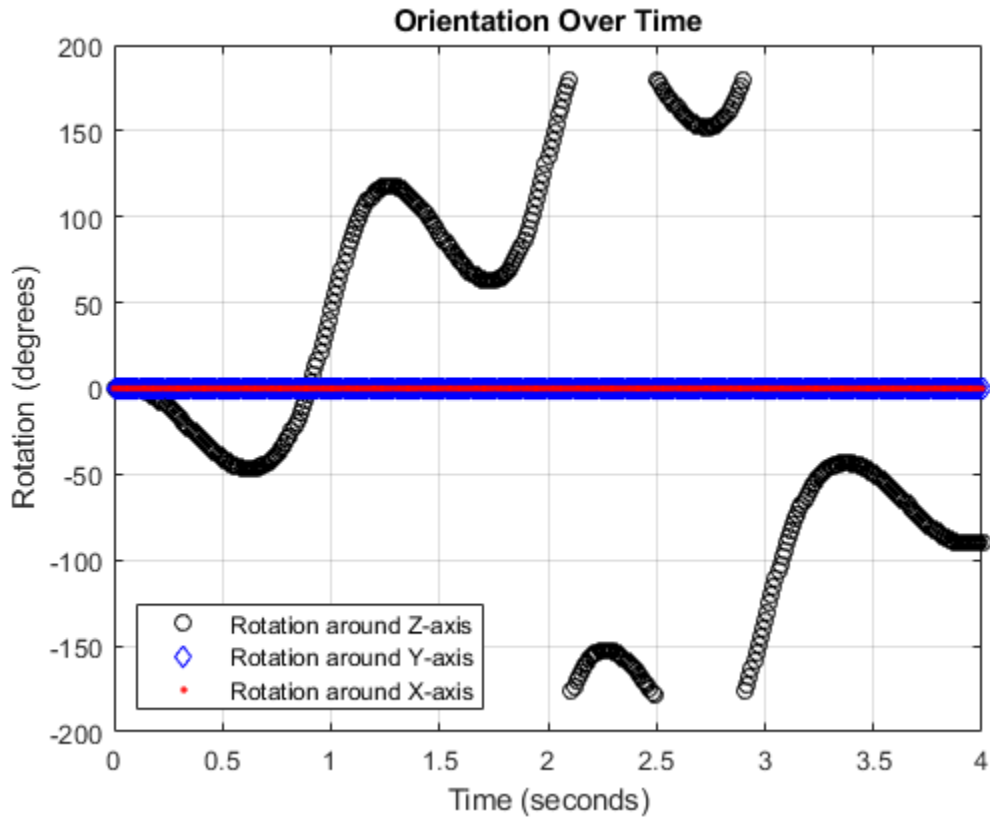


The trajectory output now appears more square-like, especially around the vertices with waypoints.

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 constraints 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;

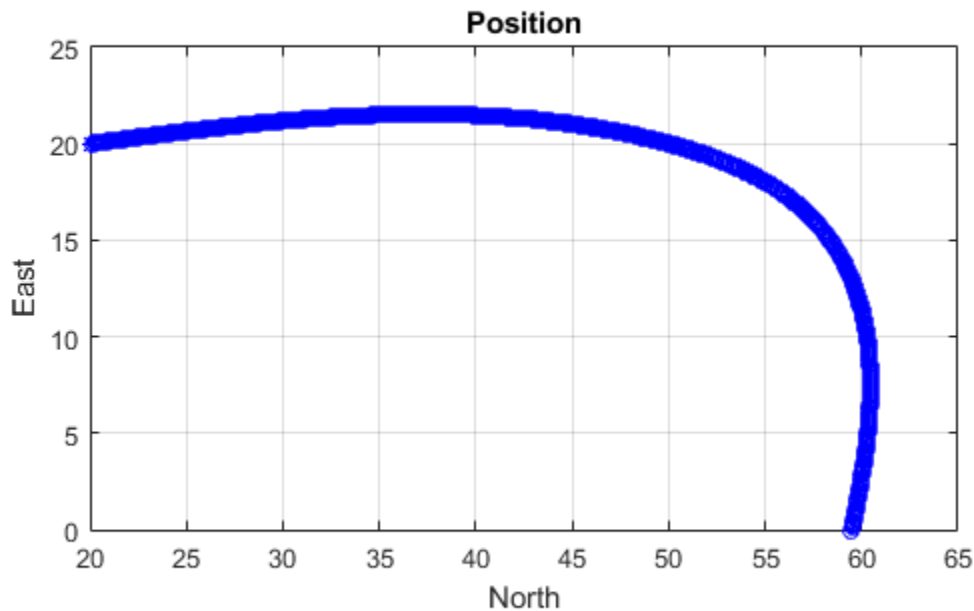
count = 1;
while ~isDone(trajectory)
```

```

[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.

```

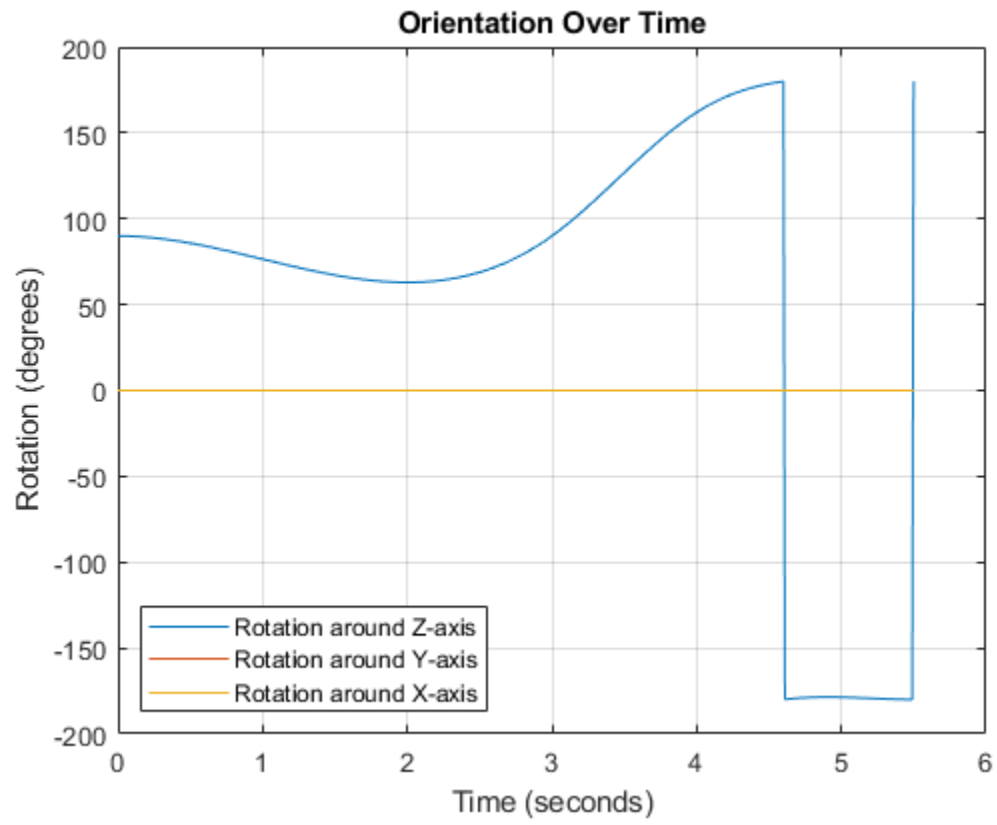
figure(2)
timeVector = 0:(1/trajectory.SampleRate):tInfo.TimeOfArrival(end);
eulerAngles = eulerd([tInfo.Orientation{1};orient],'ZYX','frame');
plot(timeVector,eulerAngles(:,1), ...
      timeVector,eulerAngles(:,2), ...
      timeVector,eulerAngles(:,3));
title('Orientation Over Time')
legend('Rotation around Z-axis', ...
       'Rotation around Y-axis', ...
       'Rotation around X-axis', ...
       'Location','southwest')
xlabel('Time (seconds)')
ylabel('Rotation (degrees)')
grid on

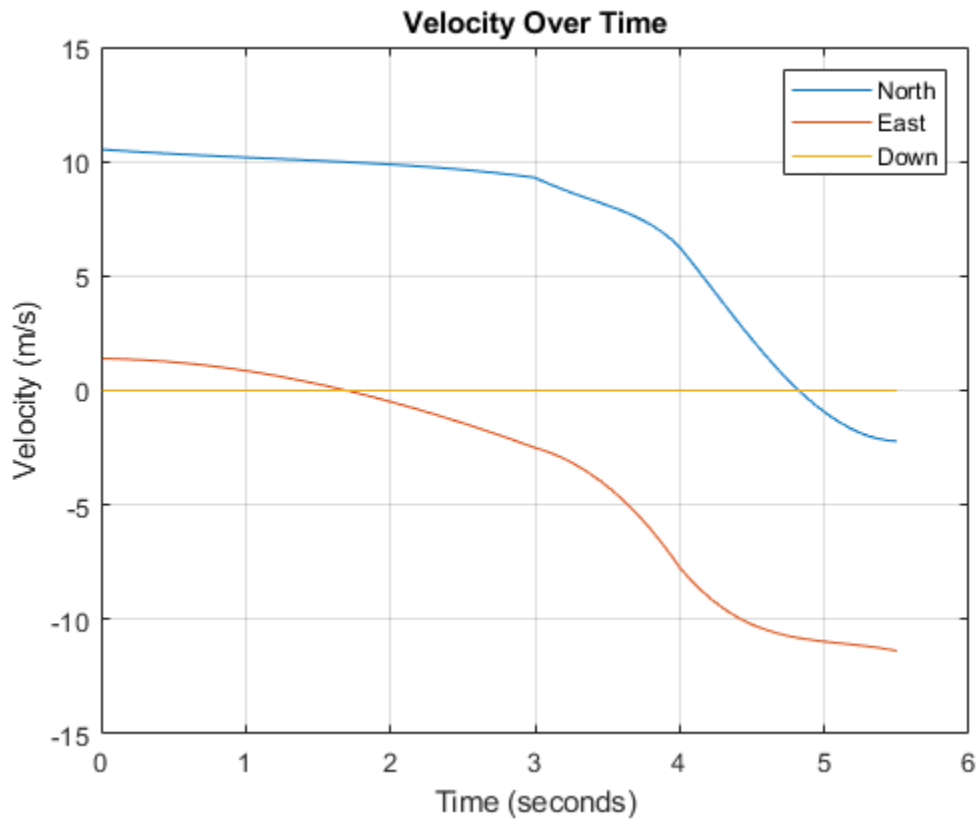
```

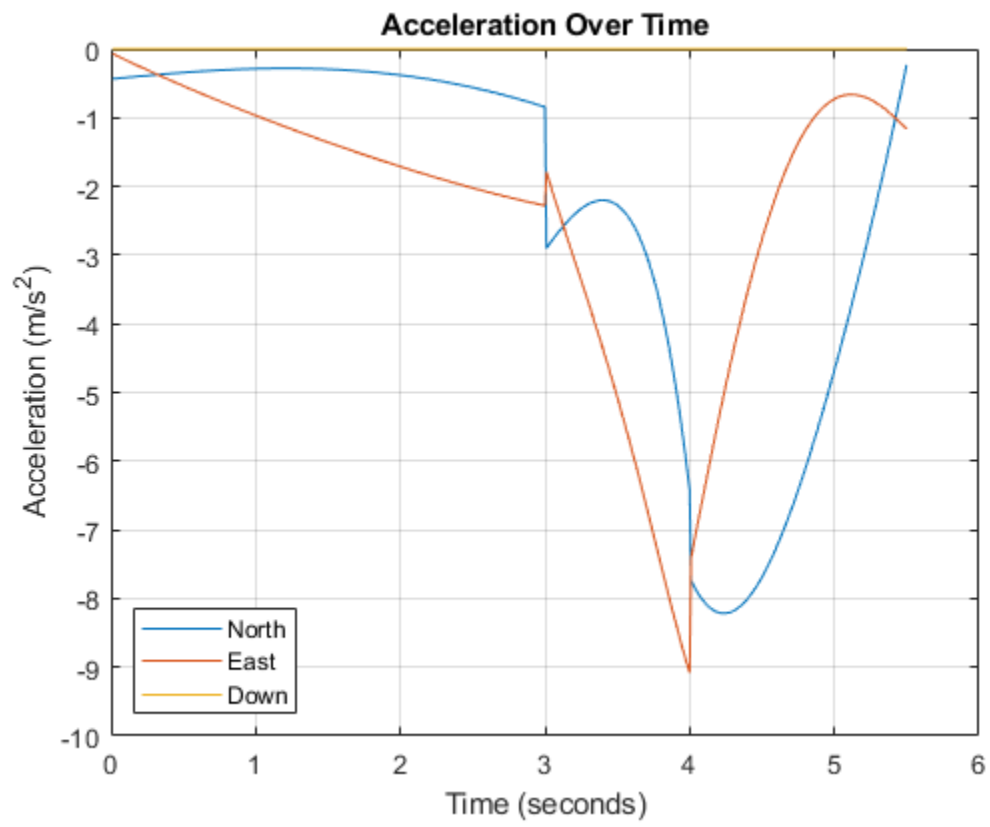
```
figure(3)
plot(timeVector(2:end),vel(:,1), ...
      timeVector(2:end),vel(:,2), ...
      timeVector(2:end),vel(:,3));
title('Velocity Over Time')
legend('North','East','Down')
xlabel('Time (seconds)')
ylabel('Velocity (m/s)')
grid on

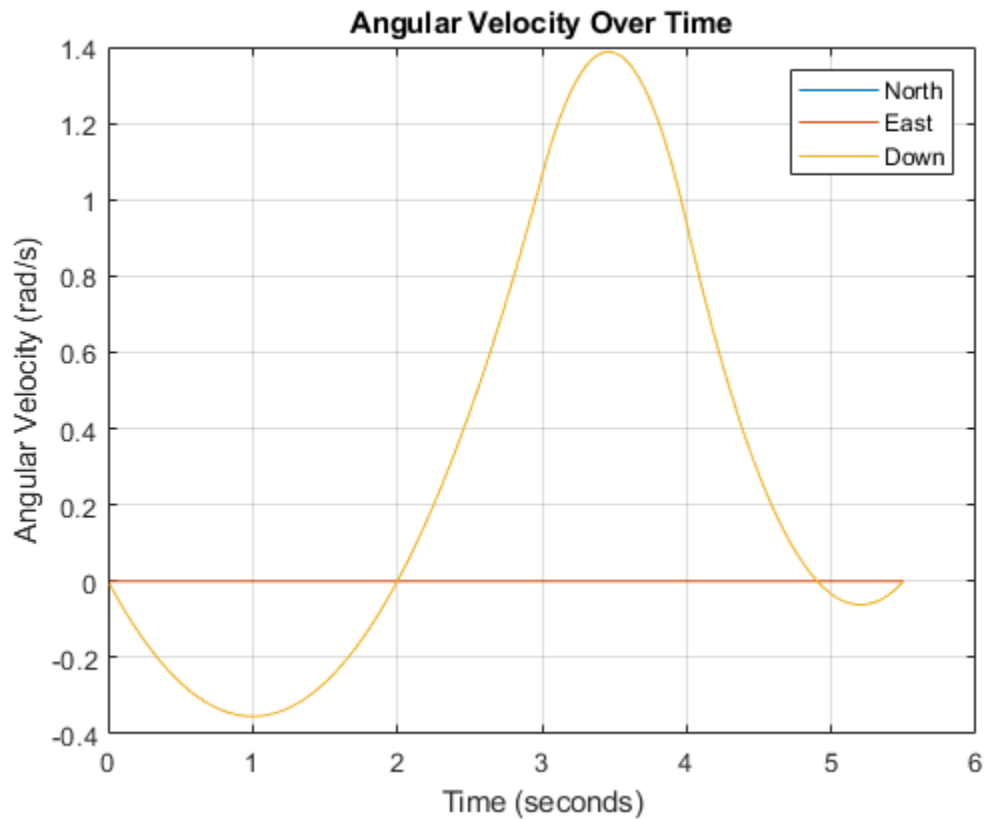
figure(4)
plot(timeVector(2:end),acc(:,1), ...
      timeVector(2:end),acc(:,2), ...
      timeVector(2:end),acc(:,3));
title('Acceleration Over Time')
legend('North','East','Down','Location','southwest')
xlabel('Time (seconds)')
ylabel('Acceleration (m/s^2)')
grid on

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
```







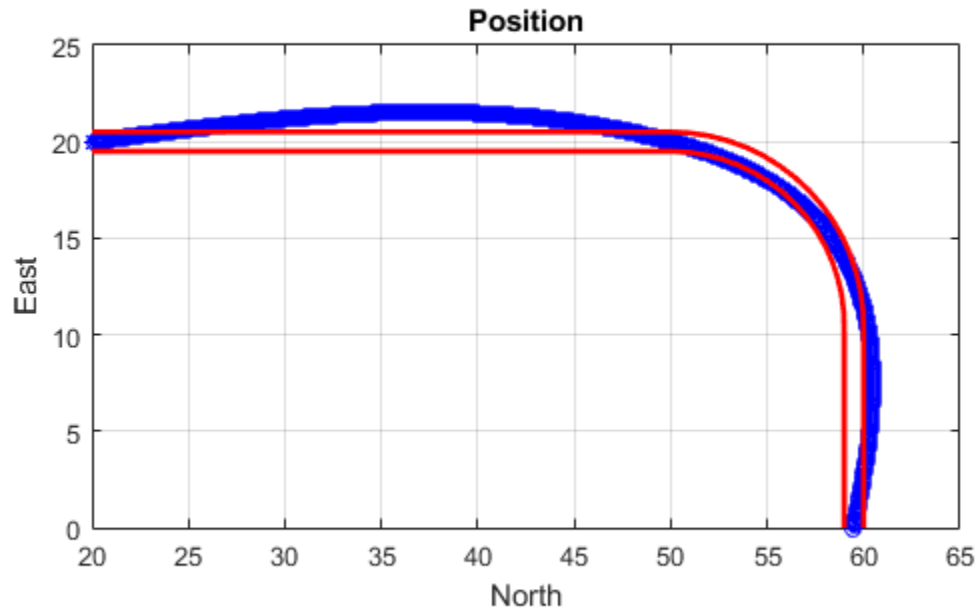
Restrict Arc Trajectory Within Preset Bounds

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;

```

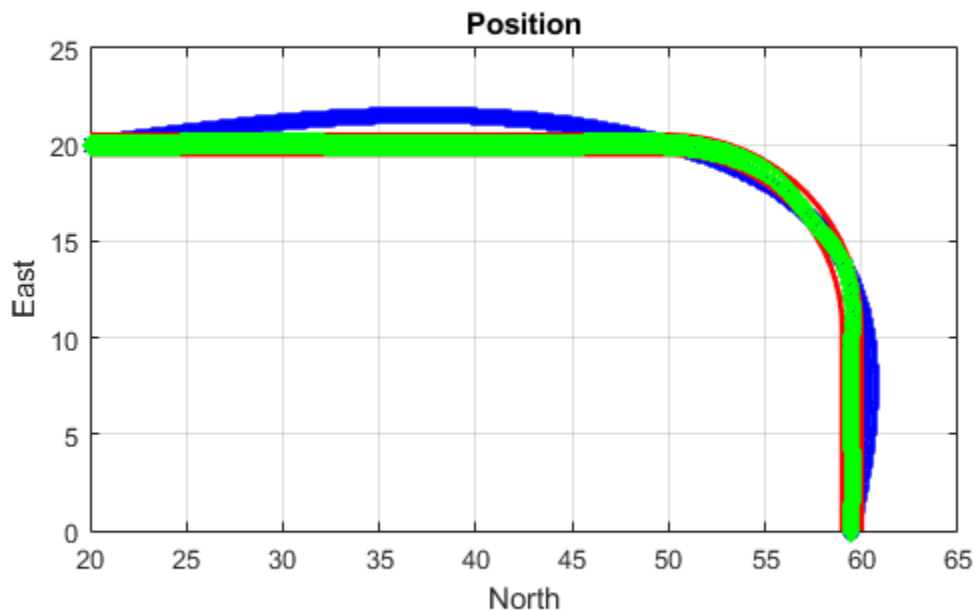
```

while ~isDone(trajjectory)
    [pos,orient(count),vel(count,:),acc(count,:),angVel(count,:)] = trajjectory();

    plot(pos(1),pos(2),'gd')

    pause(trajjectory.SamplesPerFrame/trajjectory.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/trajjectory.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

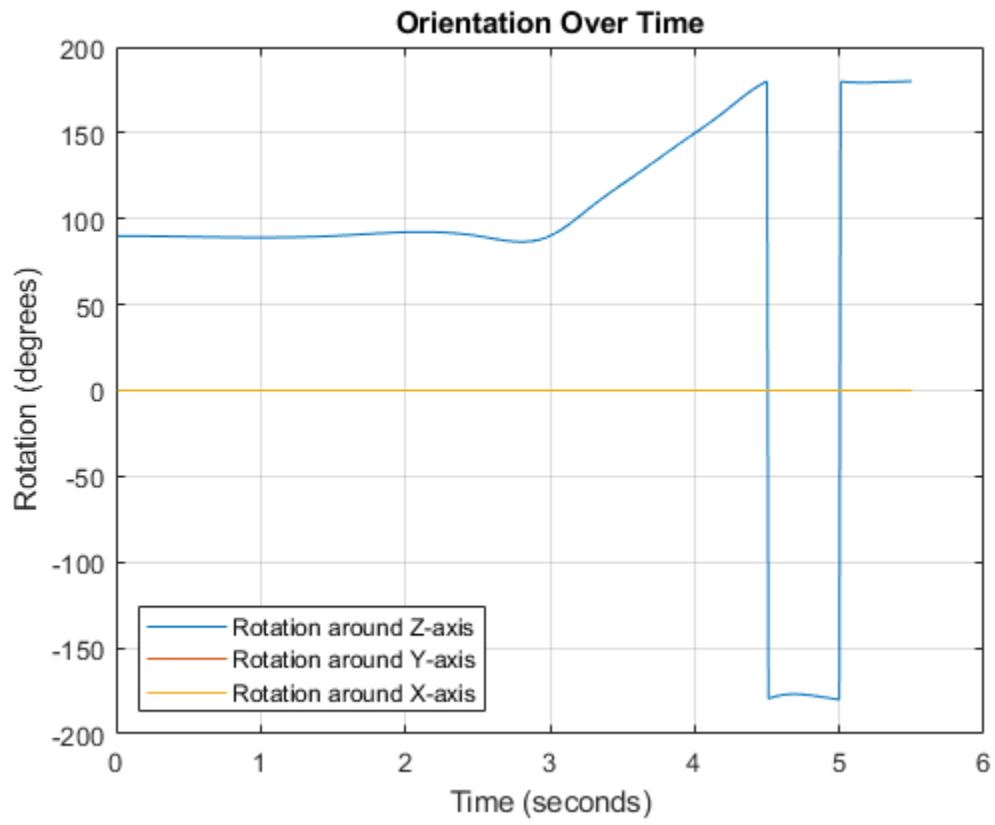
figure(3)

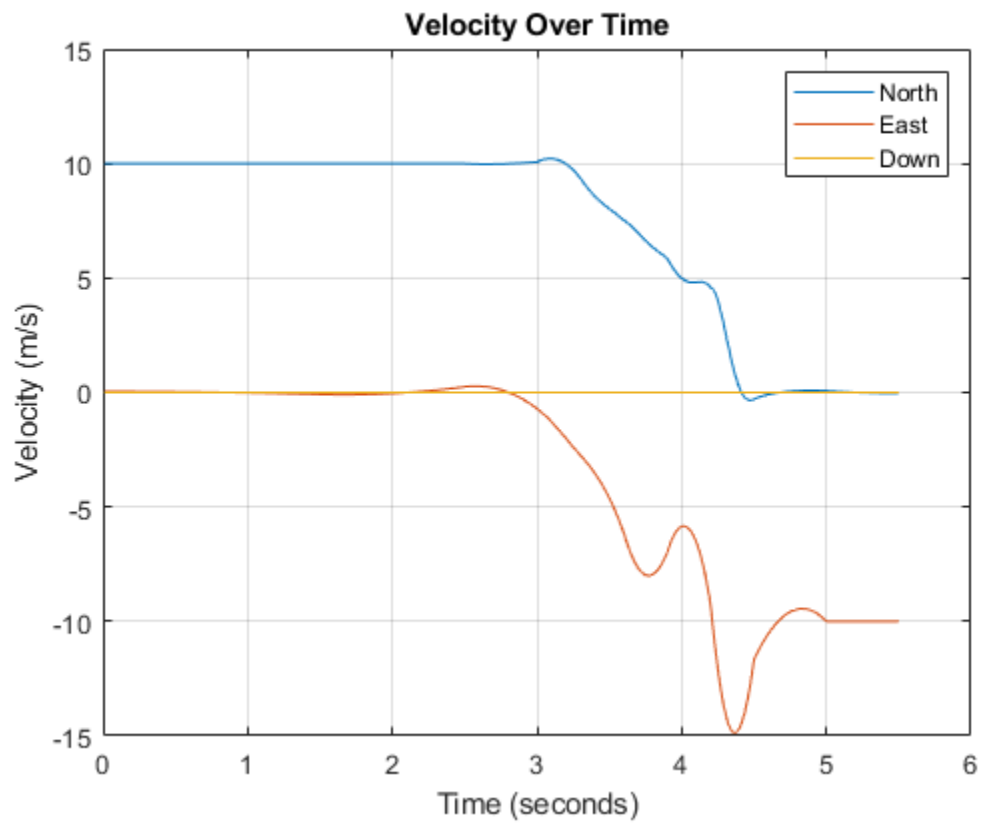
```

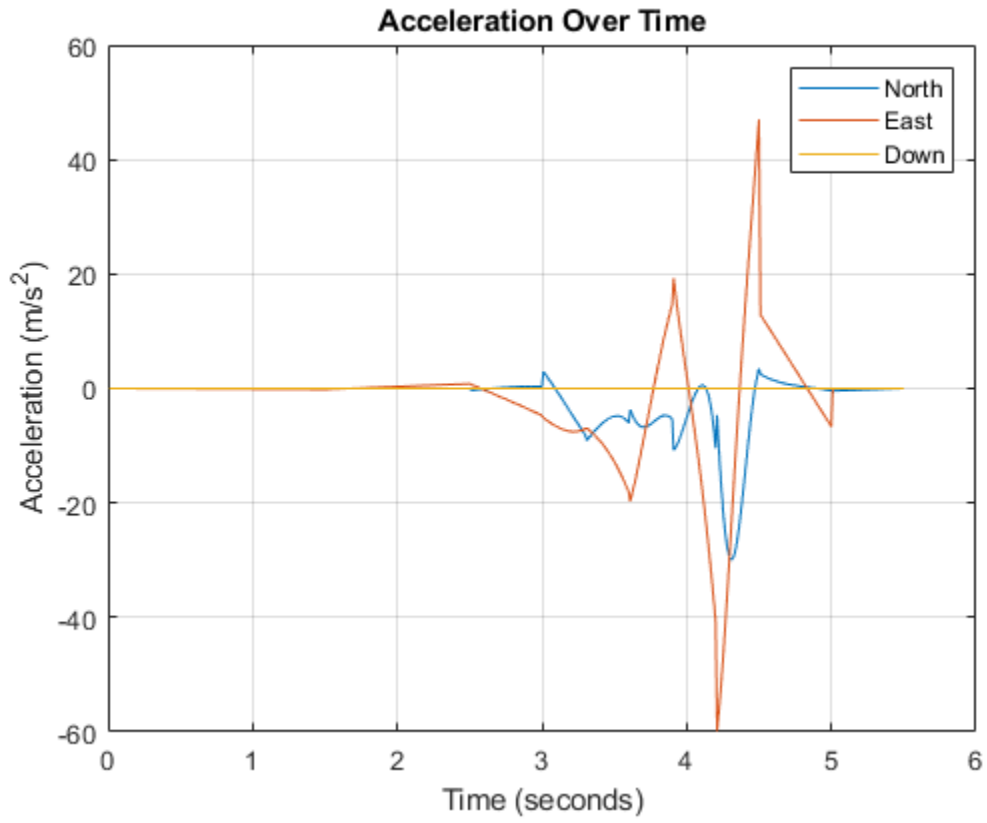
```
plot(timeVector(2:end),vel(:,1), ...
      timeVector(2:end),vel(:,2), ...
      timeVector(2:end),vel(:,3));
title('Velocity Over Time')
legend('North','East','Down')
xlabel('Time (seconds)')
ylabel('Velocity (m/s)')
grid on

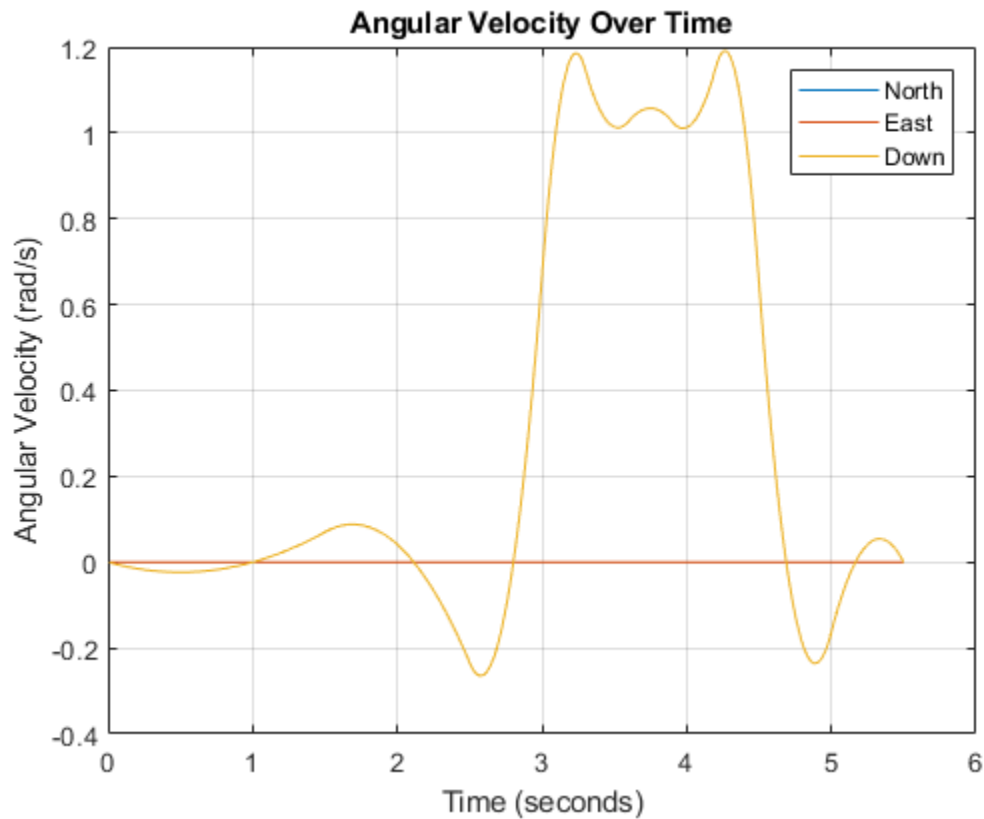
figure(4)
plot(timeVector(2:end),acc(:,1), ...
      timeVector(2:end),acc(:,2), ...
      timeVector(2:end),acc(:,3));
title('Acceleration Over Time')
legend('North','East','Down')
xlabel('Time (seconds)')
ylabel('Acceleration (m/s^2)')
grid on

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
```









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.

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 R2018b

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

`kinematicTrajectory` | `waypointTrajectory`

Introduced in R2018b

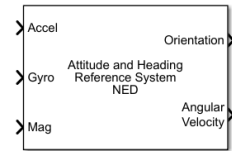
Methods

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/s², 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
0	0.000006092348396	0	0	0	0
0	0	0.000006092348396	0	0	0
0	0	0	0.000076154354947	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	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
0.009623610000000 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 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/s²)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 μT², 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/s²)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 μT², 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] (Sensor Fusion and Tracking Toolbox). 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:

$$\begin{aligned} x_k^- &= F_k x_{k-1}^+ \\ P_k^- &= F_k P_{k-1}^+ F_k^T + Q_k \\ y_k &= z_k - H_k x_k^- \\ S_k &= R_k + H_k P_k^- H_k^T \\ K_k &= P_k^- H_k^T (S_k)^{-1} \\ x_k^+ &= x_k^- + K_k y_k \\ P_k^+ &= P_k^- - K_k H_k P_k^- \end{aligned}$$

Kalman equations used in this algorithm:

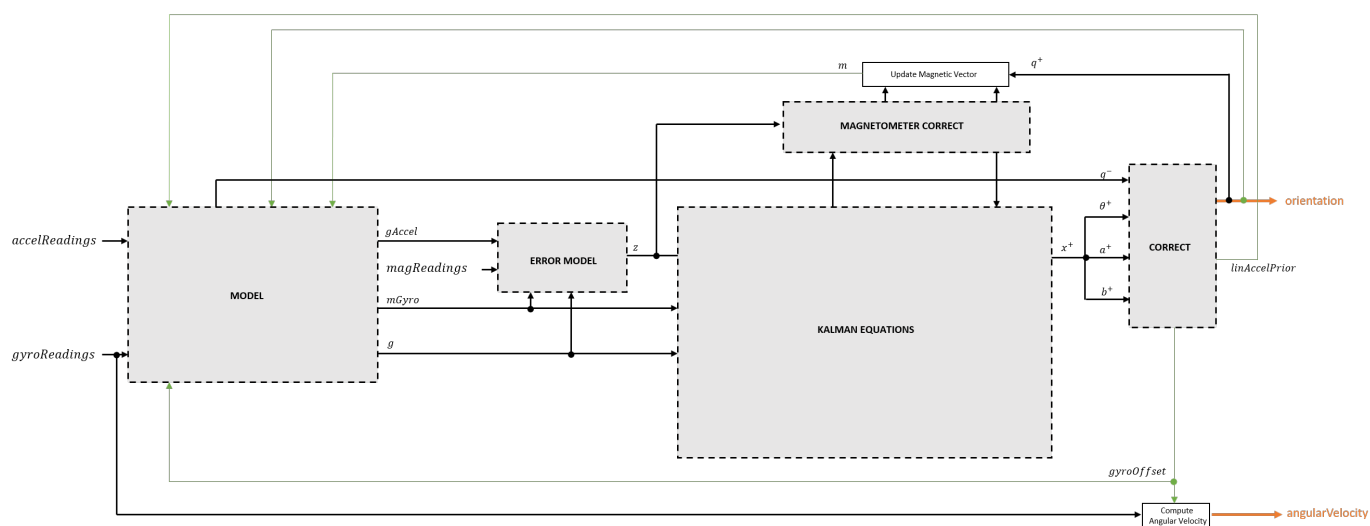
$$\begin{aligned} x_k^- &= 0 \\ P_k^- &= Q_k \\ y_k &= z_k \\ S_k &= R_k + H_k P_k^- H_k^T \\ K_k &= P_k^- H_k^T (S_k)^{-1} \\ x_k^+ &= K_k y_k \\ P_k^+ &= P_k^- - K_k H_k P_k^- \end{aligned}$$

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.

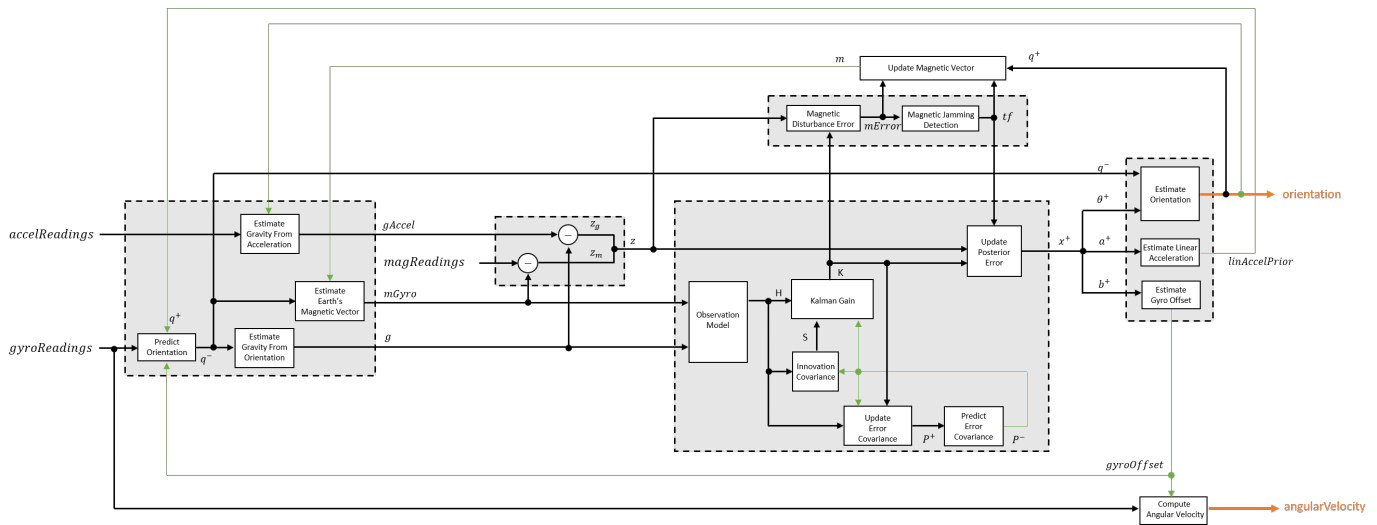
The graphic and following steps describe a single frame-based iteration through the algorithm.



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.

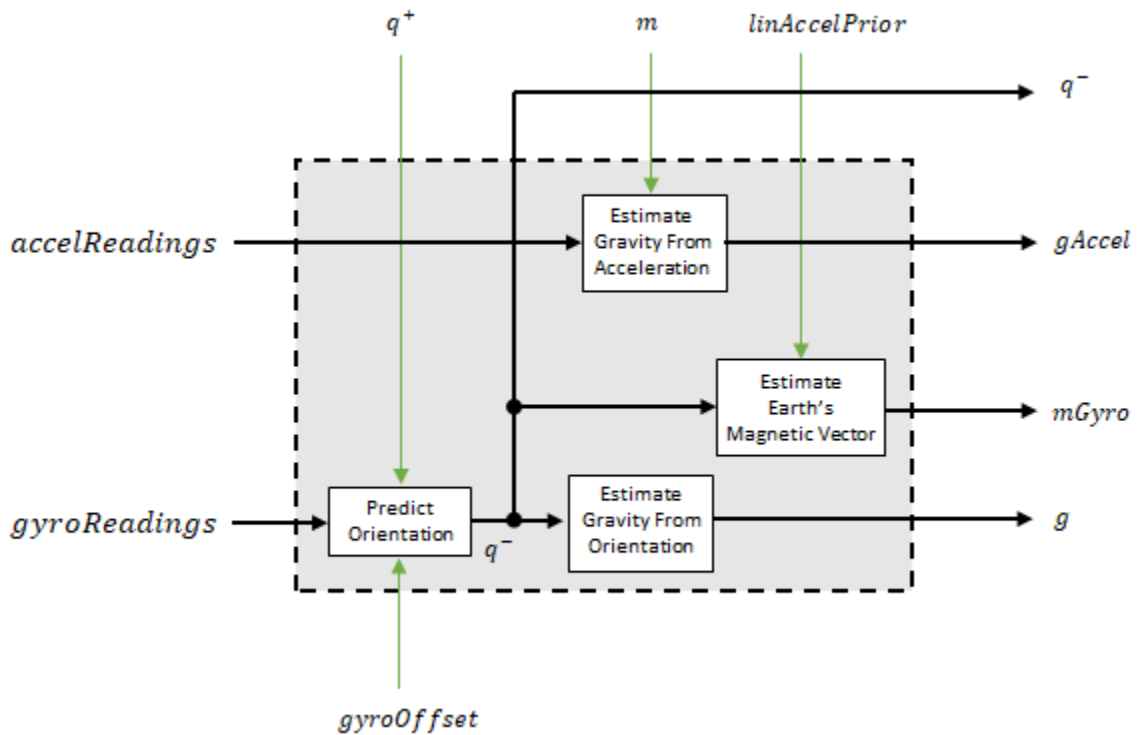
Detailed Overview

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:

$$\Delta\varphi_{N \times 3} = \frac{(gyroReadings_{N \times 3} - gyroOffset_1 \times 3)}{f_s}$$

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:

$$\Delta Q_{N \times 1} = \text{quaternion}(\Delta\varphi_{N \times 3}, 'rotvec')$$

The previous orientation estimate is updated by rotating it by ΔQ :

$$q_{1 \times 1}^- = (q_{1 \times 1}^+) \left(\prod_{n=1}^N \Delta Q_n \right)$$

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:

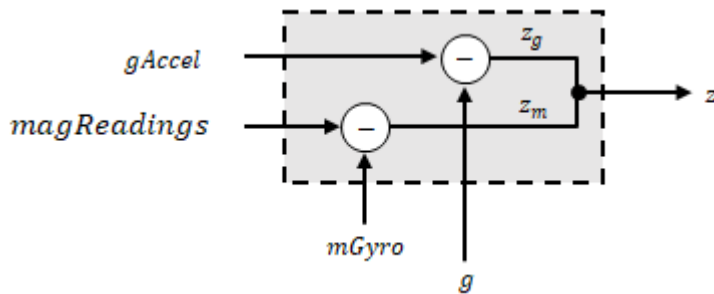
$$gAccel_{1 \times 3} = accelReadings_{1 \times 3} - linAccelPrior_{1 \times 3}$$

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:

$$mGyro_{1 \times 3} = ((rPrior)(m^T))^T$$

Error Model

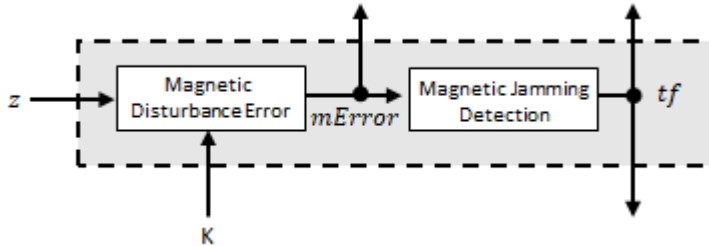


The error model combines two differences:

- 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

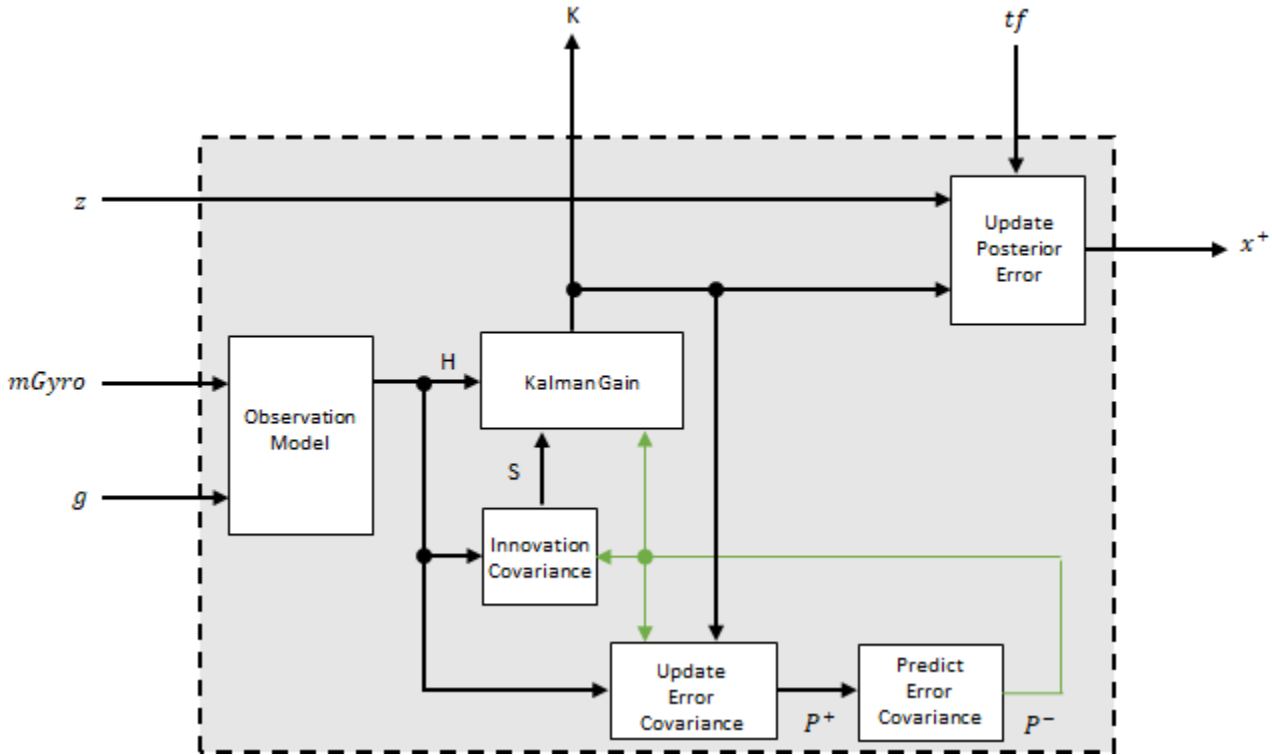
Magnetic jamming is determined by verifying that the power of the detected magnetic disturbance is less than or equal to four times the power of the expected magnetic field strength:

$$tf = \begin{cases} \text{true} & \text{if } \sum |mError|^2 > (4)(\text{ExpectedMagneticFieldStrength})^2 \\ \text{false} & \text{else} \end{cases}$$

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: $\kappa = \text{Decimation factor}/\text{Sample rate}$.

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:

$$S_{6 \times 6} = R_{6 \times 6} + (H_{6 \times 12})(P_{12 \times 12}^-)(H_{6 \times 12})^T$$

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:

$$R_{6 \times 6} = \begin{bmatrix} accel_{noise} & 0 & 0 & 0 & 0 & 0 \\ 0 & accel_{noise} & 0 & 0 & 0 & 0 \\ 0 & 0 & accel_{noise} & 0 & 0 & 0 \\ 0 & 0 & 0 & mag_{noise} & 0 & 0 \\ 0 & 0 & 0 & 0 & mag_{noise} & 0 \\ 0 & 0 & 0 & 0 & 0 & mag_{noise} \end{bmatrix}$$

where

$$accel_{noise} = \text{AccelerometerNoise} + \text{LinearAccelerationNoise} + \kappa^2 \\ (\text{GyroscopeDriftNoise} + \text{GyroscopeNoise})$$

and

$$mag_{noise} = \text{MagnetometerNoise} + \text{MagneticDisturbanceNoise} + \kappa^2 \\ (\text{GyroscopeDriftNoise} + \text{GyroscopeNoise})$$

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:

$$P_{12 \times 12}^+ = P_{12 \times 12}^- - (K_{12 \times 6})(H_{6 \times 12})(P_{12 \times 12}^-)$$

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 =

$$\begin{bmatrix}
 P^+(1) + \kappa^2 P^+(40) + \beta + \eta & 0 & 0 & -\kappa(P^+(40) + \beta) & 0 \\
 0 & P^+(14) + \kappa^2 P^+(53) + \beta + \eta & 0 & 0 & -\kappa(P^+(53) + \beta) \\
 0 & 0 & P^+(27) + \kappa^2 P^+(66) + \beta + \eta & 0 & 0 \\
 -\kappa(P^+(40) + \beta) & 0 & 0 & P^+(40) + \beta & 0 \\
 0 & -\kappa(P^+(53) + \beta) & 0 & 0 & P^+(53) + \beta \\
 0 & 0 & -\kappa(P^+(66) + \beta) & 0 & 0 \\
 0 & 0 & 0 & 0 & 0 \\
 0 & 0 & 0 & 0 & 0 \\
 0 & 0 & 0 & 0 & 0 \\
 0 & 0 & 0 & 0 & 0 \\
 0 & 0 & 0 & 0 & 0 \\
 0 & 0 & 0 & 0 & 0 \\
 0 & 0 & 0 & 0 & 0
 \end{bmatrix}$$

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:

$$K_{12 \times 6} = (P_{12 \times 12}^-)(H_{6 \times 12})^T((S_{6 \times 6})^T)^{-1}$$

where

- P^- -- predicted error covariance
- H -- observation model
- S -- innovation covariance

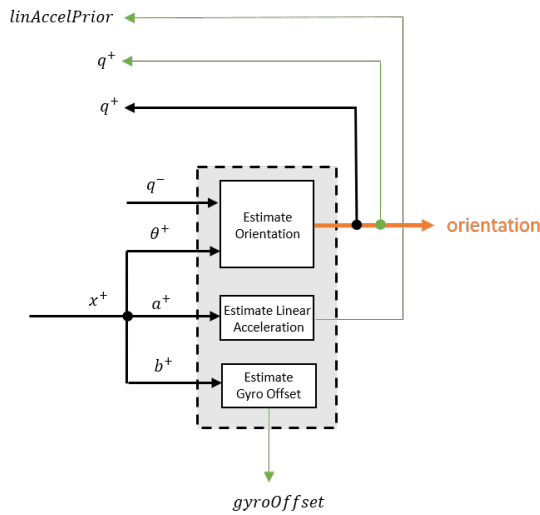
Update a Posteriori Error

The *a posteriori* error estimate is determined by combining the Kalman gain matrix with the error in the gravity vector and magnetic vector estimations:

$$x_{12 \times 1} = (K_{12 \times 6})(z_{1 \times 6})^T$$

If magnetic jamming is detected in the current iteration, the magnetic vector error signal is ignored, and the *a posteriori* error estimate is calculated as:

$$x_{9 \times 1} = (K(1:9, 1:3))(z_g)^T$$

Correct**Estimate Orientation**

The orientation estimate is updated by multiplying the previous estimation by the error:

$$q^+ = (q^-)(\theta^+)$$

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:

$$m(1) = (\text{ExpectedMagneticFieldStrength})(\cos(\text{inclination}))$$

$$m(2) = 0$$

$$m(3) = (\text{ExpectedMagneticFieldStrength})(\sin(\text{inclination}))$$

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

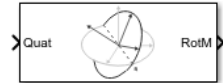
`ahrsfilter` | `ecompass` | `gpsSensor` | `imuSensor` | `imufilter`

Introduced in R2020a

Coordinate Transformation Conversion

Convert to a specified coordinate transformation representation

Library: Robotics System Toolbox / Utilities
 Navigation Toolbox / Utilities
 ROS 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) - $[x \ y \ z \ \text{theta}]$
- 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.

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

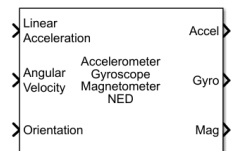
axang2quat | eul2tform | trvec2tform

Introduced in R2017b

IMU

IMU simulation model

Library: Navigation Toolbox / Multisensor Positioning / Sensor Models
Sensor Fusion and Tracking 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 ($^{\circ}\text{C}$) — Operating temperature of IMU ($^{\circ}\text{C}$)

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 $^{\circ}\text{C}$ 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/s², 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/s²)/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/s², 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 $(\text{m/s}^2/\sqrt{\text{Hz}})$, 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/s^2) – 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`

Acceleration random walk ($(\text{m/s}^2)(\sqrt{\text{Hz}})$) – Acceleration random walk ($(\text{m/s}^2)(\sqrt{\text{Hz}})$)

`[0 0 0]` (default) | real scalar | real 3-element row vector

Acceleration random walk of sensor in $(\text{m/s}^2)(\sqrt{\text{Hz}})$, 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 ($(\text{m/s}^2)/^\circ\text{C}$) – Sensor bias from temperature ($(\text{m/s}^2)/^\circ\text{C}$)

`[0 0 0]` (default) | real scalar | real 3-element row vector

Sensor bias from temperature in $(\text{m/s}^2)/^\circ\text{C}$, 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 ($\%/^\circ\text{C}$) – Scale factor error from temperature ($\%/^\circ\text{C}$)

`[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 $\%/^\circ\text{C}$, 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 ($(\text{rad/s})/\text{LSB}$) – Resolution of sensor measurements ($(\text{rad/s})/\text{LSB}$)

`0` (default) | real nonnegative scalar

Resolution of sensor measurements in $(\text{rad/s})/\text{LSB}$, 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/s²), 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)/°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

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 ($(\mu T)*\sqrt{\text{Hz}}$) — Integrated white noise of sensor ($(\mu T)*\sqrt{\text{Hz}}$)

[0 0 0] (default) | real scalar | real 3-element row vector

Integrated white noise of sensor in $(\mu T)*\sqrt{\text{Hz}}$, 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 ($\mu T/^\circ\text{C}$) — Sensor bias from temperature ($\mu T/^\circ\text{C}$)

[0 0 0] (default) | real scalar | real 3-element row vector

Sensor bias from temperature in $\mu T/^\circ\text{C}$, 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 ($\%/^\circ\text{C}$) — Scale factor error from temperature ($\%/^\circ\text{C}$)

[0 0 0] (default) | real scalar in the range [0,100] | real 3-element row vector in the range [0,100]

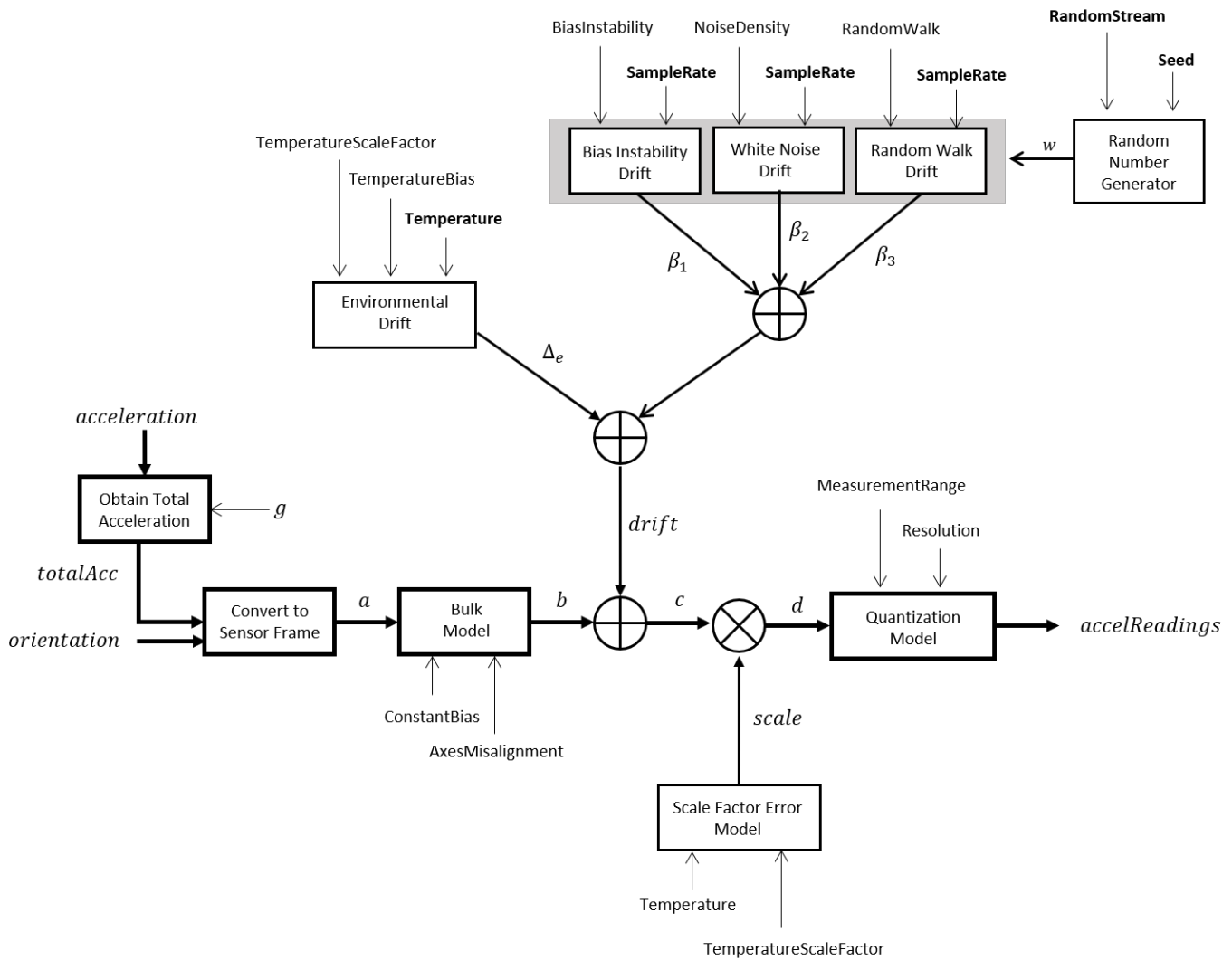
Scale factor error from temperature in $\%/^\circ\text{C}$, specified as a real scalar or real 3-element row vector with values ranging from 0 to 100. Any scalar input is converted into a real 3-element row vector where each element has the input scalar value.

Data Types: single | double

Algorithms

Accelerometer

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/s²) as:

$$totalAcc = -acceleration + g$$

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:

$$b = \left(\begin{bmatrix} 1 & \frac{\alpha_2}{100} & \frac{\alpha_3}{100} \\ \frac{\alpha_1}{100} & 1 & \frac{\alpha_3}{100} \\ \frac{\alpha_1}{100} & \frac{\alpha_2}{100} & 1 \end{bmatrix} (a^T) \right)^T + \text{ConstantBias}$$

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:

$$H_1(z) = \frac{1}{1 + \left(\frac{2}{\text{SampleRate}} - 1 \right) z^{-1}}$$

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:

$$scaleFactorError = 1 + \left(\frac{Temperature - 25}{100} \right) (TemperatureScaleFactor)$$

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:

$$e = \begin{cases} \text{MeasurementRange} & \text{if } d > \text{MeasurementRange} \\ -\text{MeasurementRange} & \text{if } -d > \text{MeasurementRange} \\ d & \text{else} \end{cases}$$

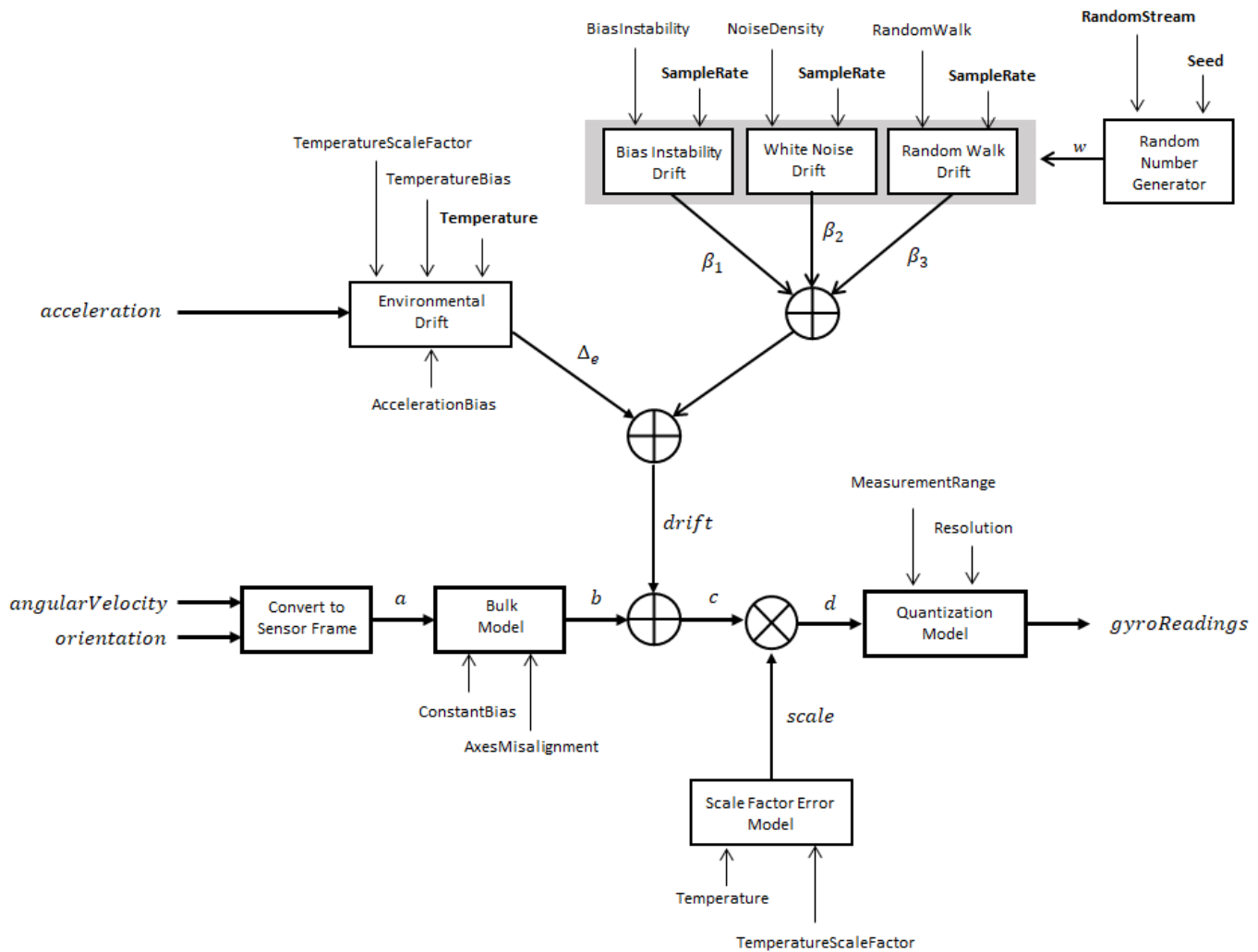
and then setting the resolution:

$$accelReadings = (\text{Resolution}) \left(\text{round} \left(\frac{e}{\text{Resolution}} \right) \right)$$

where `MeasurementRange` is a property of `accelParams`.

Gyroscope

The 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:

$$a = (\textit{orientation})(\textit{angularVelocity})^T$$

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:

$$b = \left(\begin{bmatrix} 1 & \frac{\alpha_2}{100} & \frac{\alpha_3}{100} \\ \frac{\alpha_1}{100} & 1 & \frac{\alpha_3}{100} \\ \frac{\alpha_1}{100} & \frac{\alpha_2}{100} & 1 \end{bmatrix} (a^T) \right)^T + \text{ConstantBias}$$

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:

$$H_1(z) = \frac{1}{1 + \left(\frac{2}{\text{SampleRate}} - 1 \right) z^{-1}}$$

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:

$$scaleFactorError = 1 + \left(\frac{Temperature - 25}{100} \right) (TemperatureScaleFactor)$$

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:

$$e = \begin{cases} \text{MeasurementRange} & \text{if } d > \text{MeasurementRange} \\ -\text{MeasurementRange} & \text{if } -d > \text{MeasurementRange} \\ d & \text{else} \end{cases}$$

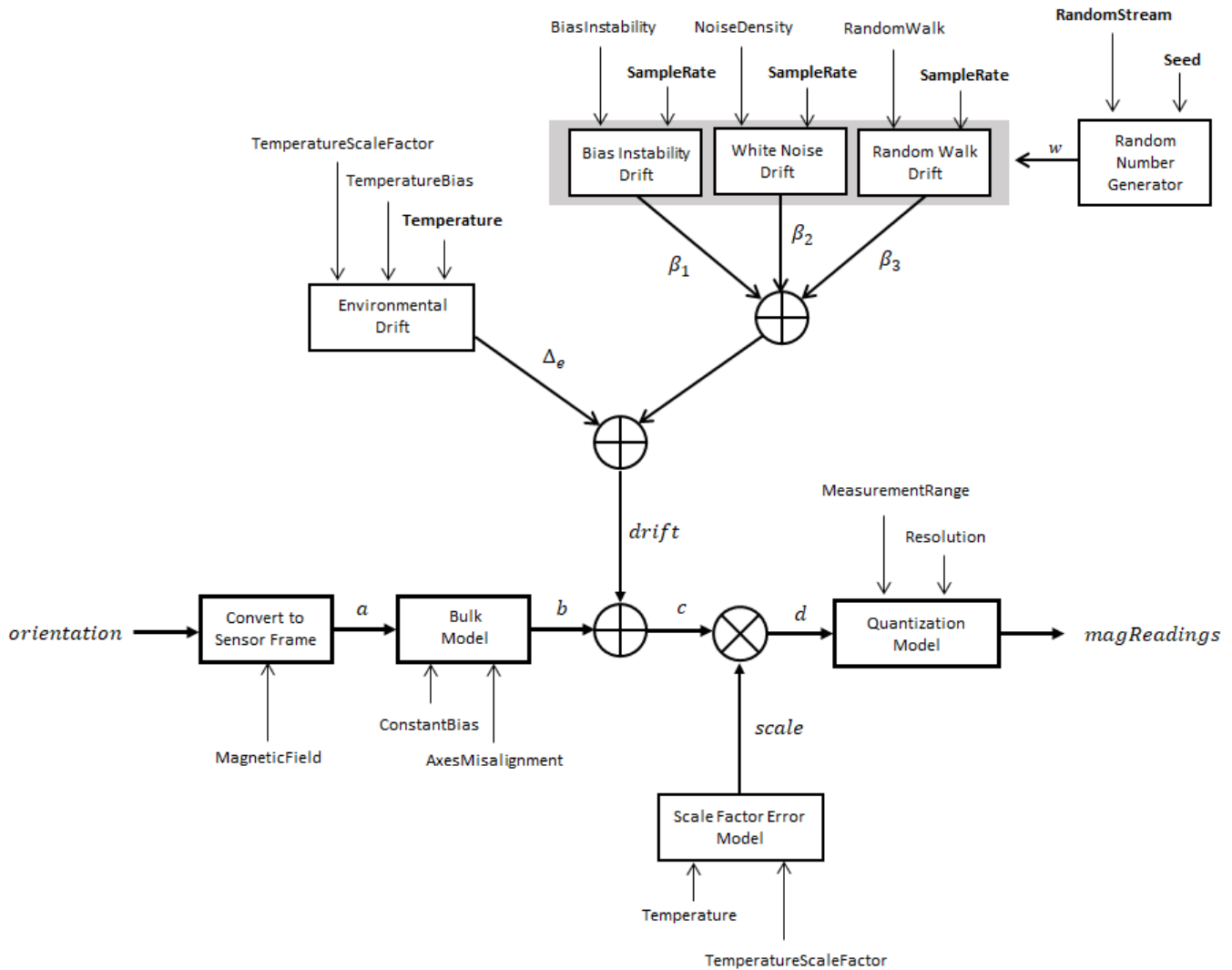
and then setting the resolution:

$$gyroReadings = (\text{Resolution}) \left(\text{round} \left(\frac{e}{\text{Resolution}} \right) \right)$$

where `MeasurementRange` is a property of `gyroParams`.

Magnetometer

The 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:

$$b = \left(\begin{bmatrix} 1 & \frac{\alpha_2}{100} & \frac{\alpha_3}{100} \\ \frac{\alpha_1}{100} & 1 & \frac{\alpha_3}{100} \\ \frac{\alpha_1}{100} & \frac{\alpha_2}{100} & 1 \end{bmatrix} (a^T) \right)^T + \text{ConstantBias}$$

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:

$$H_1(z) = \frac{1}{1 + \left(\frac{2}{\text{SampleRate}} - 1 \right) z^{-1}}$$

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:

$$scaleFactorError = 1 + \left(\frac{Temperature - 25}{100} \right) (TemperatureScaleFactor)$$

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`

System Objects

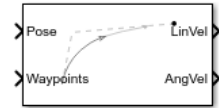
`gpsSensor` | `imuSensor`

Introduced in R2020a

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 | binaryOccupancyMap | controllerVFH | occupancyMap | occupancyMap

Topics

“Plan Path for a Differential Drive Robot in Simulink” (Robotics System Toolbox)

“Path Following with Obstacle Avoidance in Simulink®”

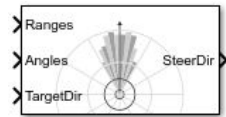
“Pure Pursuit Controller”

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 4-41 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 addition 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

Publish | Pure Pursuit | 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.

Note You must have the ROS Toolbox to use this app.

To use the app:

 Import	<p>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 5-12.</p> <p>You can also specify scans and odometry poses that are prefiltered in the workspace by calling <code>slamMapBuilder</code> with inputs. This skips the import process. See Programmatic Use on page 5-11.</p>
 SLAM Settings	<p>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 5-12.</p>
 Build	<p>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.</p>
 Incremental Match Loop Closure	<p>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 5-13.</p>
 Sync	<p>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 5-14.</p>
 Export Occupancy Grid	<p>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 <code>occupancyMap</code> object.</p>
 Open Session Save Session	<p>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.</p>

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.

Open the App

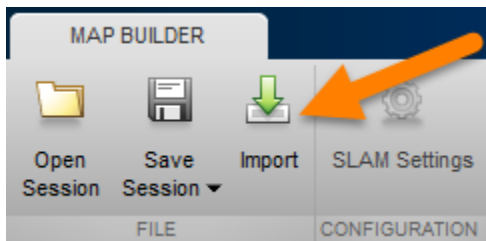
In the **Apps** tab, under **Control System Design and Analysis**, click **SLAM Map Builder**.

Also, you can call the function:

```
slamMapBuilder
```

Import Lidar Scans from rosbag

Click **Import > Import from rosbag** to load a rosbag. The provided rosbag, `southend.bag`, contains laser scan messages. Select the file and open. The scans are shown in the **Import** tab.

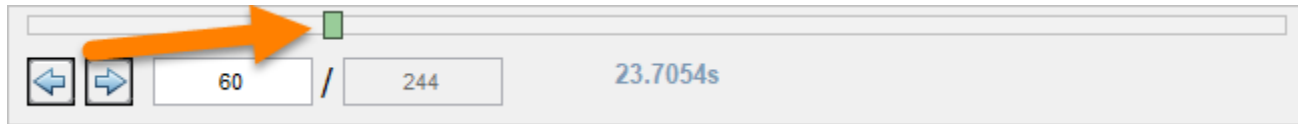


In the **Import** Tab, specify the import parameters. The **Lidar Topic** is preselected as `/scan` because that is the only `sensor_msgs/LaserScan` topic found. If odometry is available as a `tf` tree, select **Use TF**, and specify the **Lidar Frame** (sensor frame) and the **Fixed Frame** (world frame).

Select the desired **Start Time (s)** and **End Time (s)**. Because the scans are captured at a high frequency, downsample the scans to reduce data processing. Select the desired percentage of scans in **Downsample Scans to (%)**. Scans are evenly sampled. For example, 5% is every 20th scan.

Click **Apply** to apply filtering parameters.

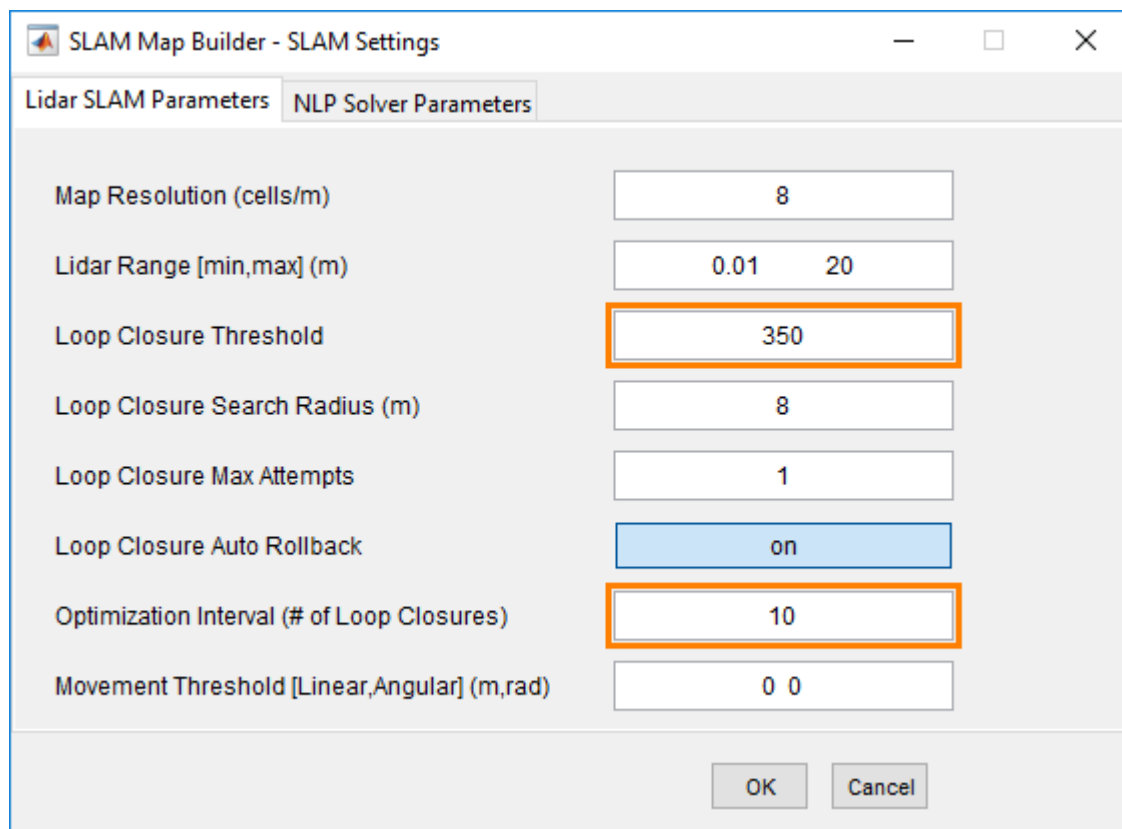
Use the slider or arrow keys as the bottom to preview the scans.



Once you are done filtering, click **Close**.

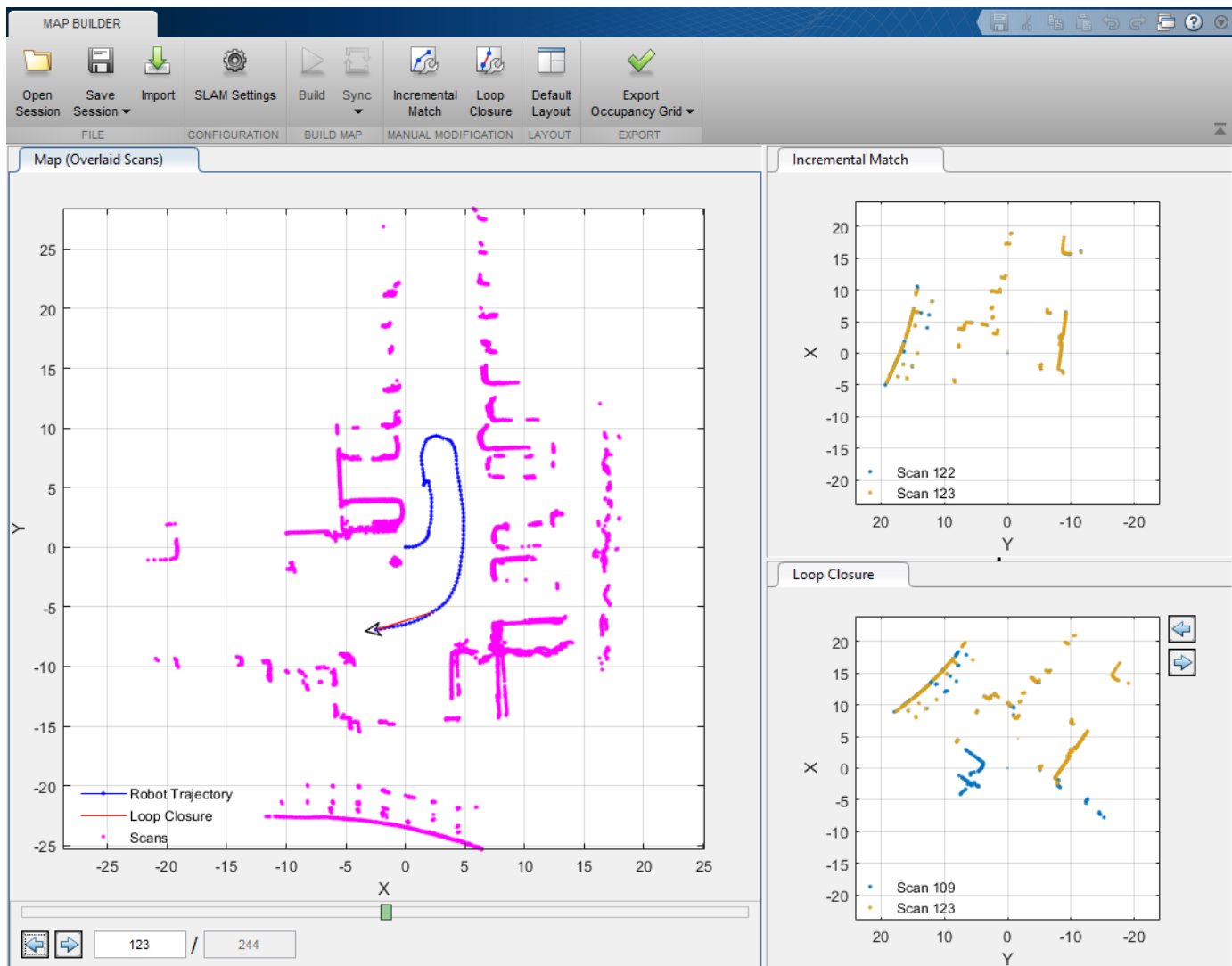
Tune SLAM Settings

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 robotic 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.



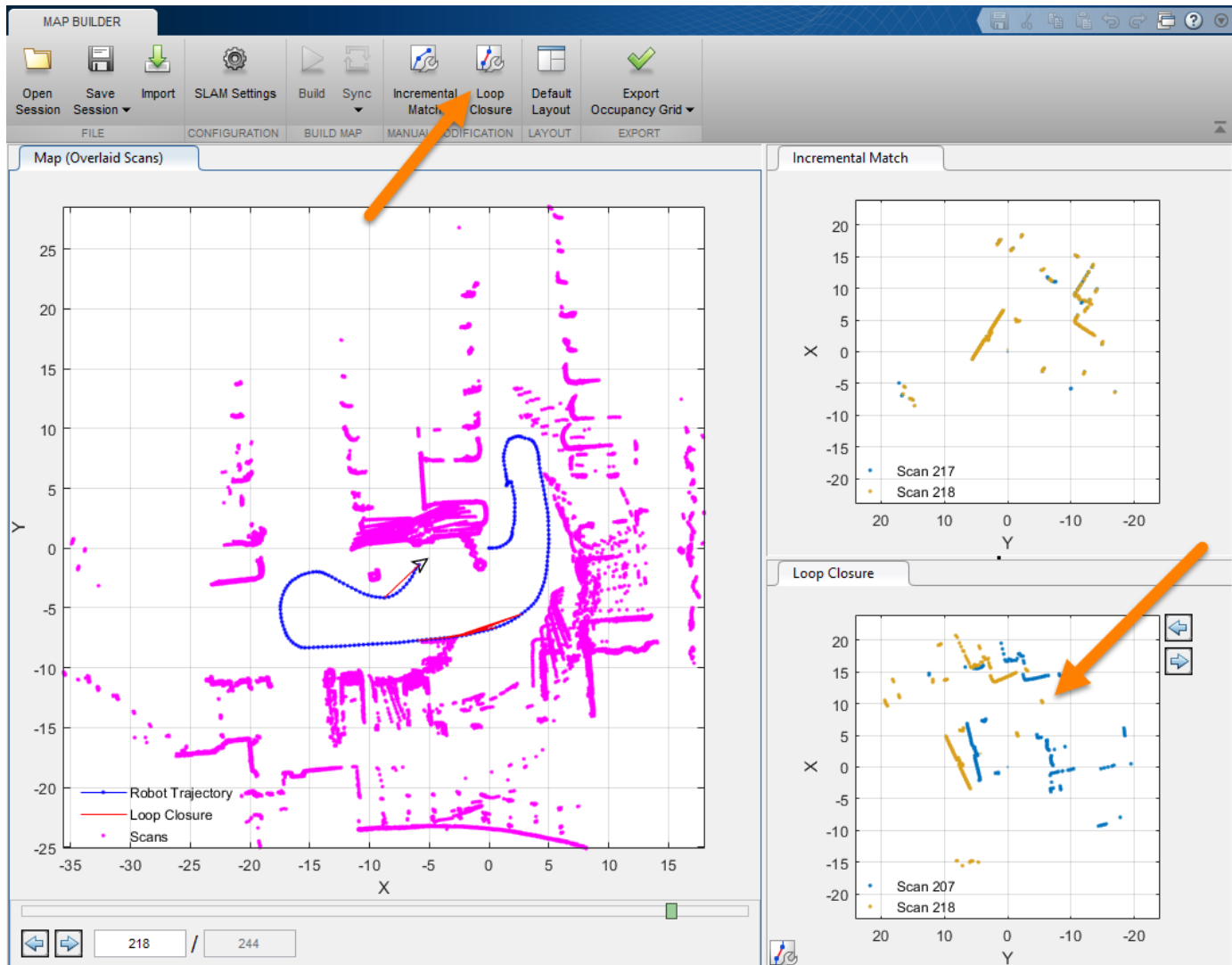
Build the Map

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.



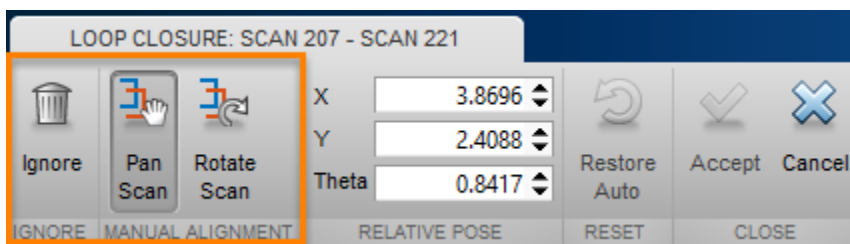
At any time during the build process, if you notice the map is distorted or an incremental match or loop closure looks off, click **Pause** to select scans for adjustment. You can modify scans at the end of the build process as well. Navigate using the arrow keys or slider to the point in the file where the distortion first occurs. Click the **Incremental Match** or **Loop Closure** buttons to adjust the currently displayed scan poses. In this example, we manually created a bad loop closure that does not normally occur with this data set at scan 218.

Click the **Loop Closure** button. This opens a tab for modifying the loop closure relative pose.

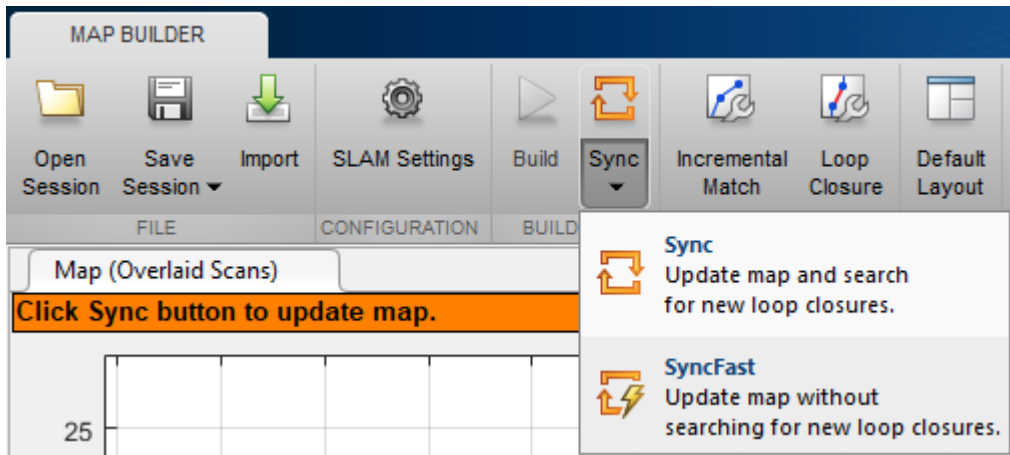


To ignore the loop closure completely, click **Ignore**. Otherwise, manually modify the relative scan pose until the scans line up.

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.

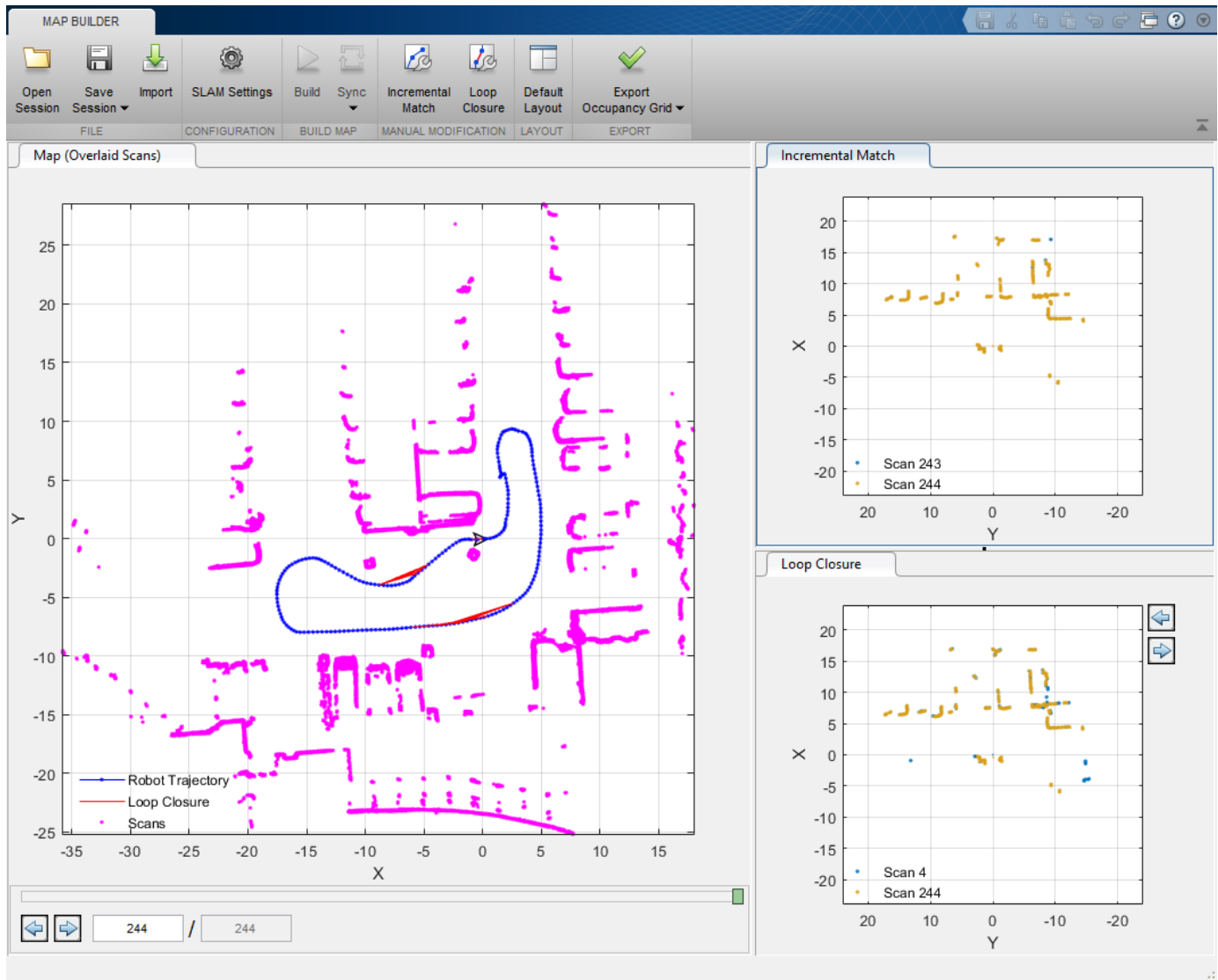


After you modify your scan poses for incremental matches and loop closures, click **Sync** to apply changes. **SyncFast** updates the map without searching for new loop closures and reduces computation time if you have already processed all the scans.

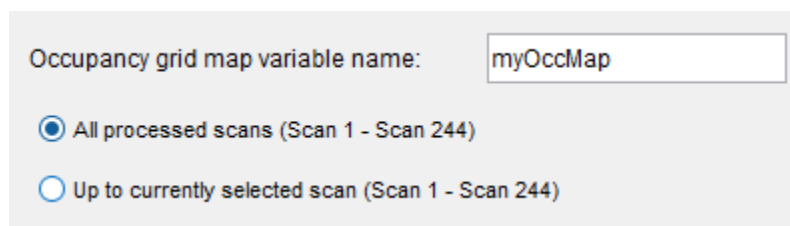


Export Occupancy Grid

Once you have synced your changes and finished building the map, you should see a fully overlaid scan map with a robot trajectory.

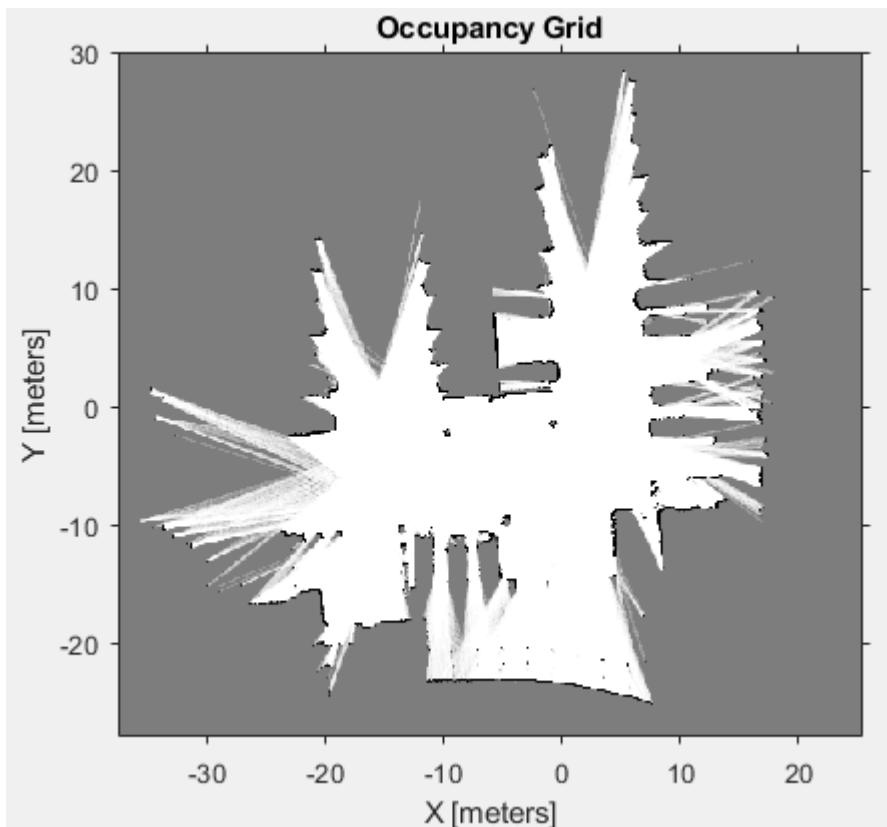


Click **Export Occupancy Grid** to get a final occupancy map of your environment as a `occupancyMap` object. Specify the variable name to export the map to the workspace. You can create a map from a subset of scans by scrolling back to the desired frame before exporting and selecting **Up to currently 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 toolbar.

`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 toolbar. 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 5-11.

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.

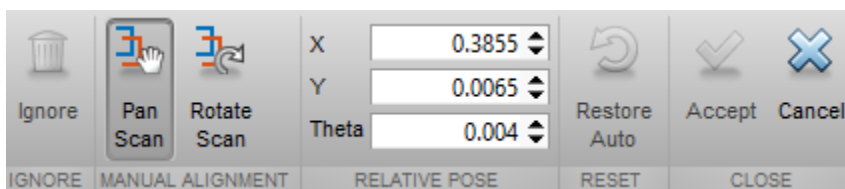
NLP Solver Parameters:

- **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, θ , 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](#) | [optimizePoseGraph](#) | [rosviz](#)

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