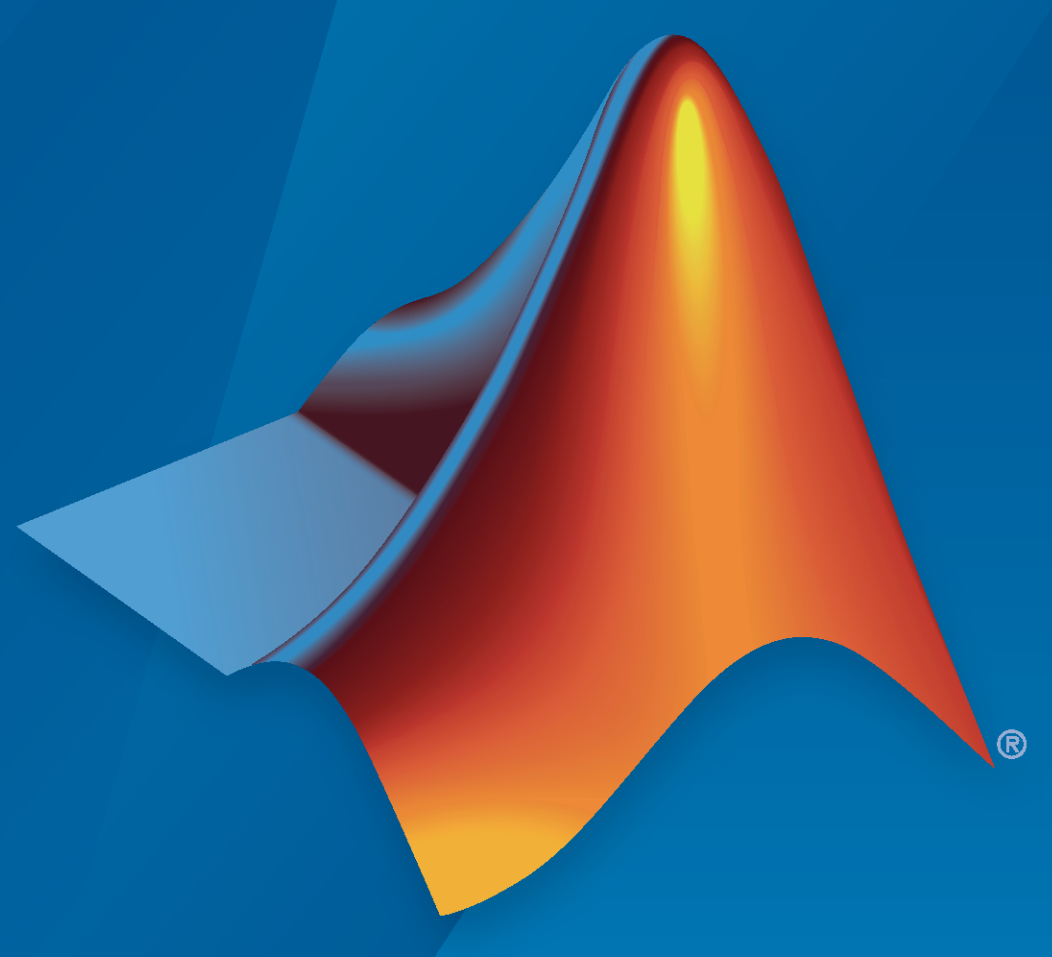
UAV Toolbox

Reference



MATLAB® & SIMULINK®

R2023a

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

UAV Toolbox Reference

© COPYRIGHT 2020–2023 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 2020	Online only	New for Version 1.0 (R2020b)
March 2021	Online only	Revised for Version 1.1 (R2021a)
September 2021	Online only	Revised for Version 1.2 (R2021b)
March 2022	Online only	Revised for Version 1.3 (R2022a)
September 2022	Online only	Revised for Version 1.4 (R2022b)
March 2023	Online only	Revised for Version 1.5 (R2023a)

Contents

1 Classes

2 Methods

3 Functions

4 Blocks

5 Apps

6 Scenes

7 Vehicles

1

Classes

controllerVFH3D

Avoid obstacles using 3D vector field histogram

Description

The `controllerVFH3D` System object™ enables a UAV to avoid obstacles, based on sensor data, by using 3D vector field histograms (3DVFH). The object computes an obstacle-free direction using the sensor-data-based positions of obstacles, the UAV position, the UAV orientation, and a target position.

`controllerVFH3D` uses the 3DVFH+ algorithm to compute an obstacle-free direction.

To find an obstacle-free direction:

1. Create the `controllerVFH3D` object and set its properties.
2. Call the object with arguments, as if it were a function.

To learn more about how System objects work, see What Are System Objects?

Creation

Syntax

```
vfh3D = controllerVFH3D
vfh3D = controllerVFH3D(Name=Value)
```

Description

`vfh3D = controllerVFH3D` returns a 3-D vector field histogram object, `vfh3D`, that computes a desired direction, a desired yaw, and a status output using the 3DVFH+ algorithm.

`vfh3D = controllerVFH3D(Name=Value)` returns a vector field histogram object with properties specified by one or more name-value arguments. Properties not specified retain their 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.

`HistogramResolution` — Histogram grid resolution
5 (default) | 1 | 3 | 6 | 10 | 15 | 18 | 30 | 45 | 60

Histogram grid resolution, specified as `1`, `3`, `5`, `6`, `10`, `15`, `18`, `30`, `45`, or `60`. All values are in degrees.

Example: `HistogramResolution=10`

Data Types: `single` | `double`

`WindowSize` — Histogram window size
`1` (default) | positive odd integer

Histogram window size, specified as a positive odd integer. The histogram window size determines the angular width of an obstacle-free opening in the azimuth and elevation directions. This value is unitless.

Example: `WindowSize=3`

Data Types: `single` | `double`

`HistogramThreshold` — Threshold for computing histogram
`1` (default) | positive integer

Threshold for computing the histogram, specified as a positive integer. This value specifies the minimum number of obstacle points that must be in a histogram cell for the cell to be considered as an obstacle. If a cell contains fewer than this number of obstacle points, the cell is considered as obstacle-free.

Example: `HistogramThreshold=2`

Data Types: `single` | `double`

`MaxAge` — Maximum age of remembered obstacle points
`0` (default) | numeric scalar

Maximum age of remembered obstacle points, specified as a numeric scalar. This value specifies the number of time steps prior to the current step from which the object remembers the obstacle points.

Example: `MaxAge=1`

Data Types: `single` | `double`

`DistanceLimits` — Limits of range sensor
`[0.2 10]` (default) | vector of form `[min max]`

Limits of the range sensor, specified as a vector of the form `[min max]`, with values in meters.

Example: `DistanceLimits=[0.3 15]`

Data Types: `single` | `double`

`HorizontalSensorFOV` — Horizontal field of view limits of range sensor
`[-60 60]` (default) | vector of form `[min max]`

Horizontal field of view limits of the range sensor, specified as a vector of the form `[min max]`, with values in degrees.

Example: `HorizontalSensorFOV=[-50 50]`

Data Types: `single` | `double`

`VerticalSensorFOV` — Vertical field of view limits of range sensor
`[-30 30]` (default) | vector of form `[min max]`

Vertical field of view limits of the range sensor, specified as a vector of the form [*min max*], with values in degrees.

Example: `VerticalSensorFOV=[-20 20]`

Data Types: `single` | `double`

`SensorLocation` — Sensor mounting location on UAV
`[0 0 0]` (default) | vector of form [*x y z*]

Sensor mounting location on the UAV, specified as a vector of the form [*x y z*], with values in meters.

Example: `SensorLocation=[0 0 0]`

Data Types: `single` | `double`

`SensorOrientation` — Orientation of sensor mounted on UAV
`[0 0 0]` (default) | vector of form [*roll pitch yaw*]

Orientation of the sensor mounted on the UAV, specified as a vector of the form [*roll pitch yaw*], with values in degrees.

Example: `SensorOrientation=[0 0 0]`

Data Types: `single` | `double`

`VehicleRadius` — Radius of UAV
`1` (default) | numeric scalar

Radius of the UAV, specified as a numeric scalar in meters. This value specifies the radius of the smallest circle that can circumscribe the UAV. The object uses the vehicle radius to account for vehicle size when computing the obstacle-free direction.

Example: `VehicleRadius=0.5`

Data Types: `single` | `double`

`SafetyDistance` — Safety distance between UAV and obstacle
`1` (default) | numeric scalar

Safety distance between UAV and obstacle, specified as a numeric scalar in meters. This value specifies the space to account for between the UAV and obstacles in addition to the vehicle radius. The object uses both the vehicle radius and safety distance to compute the obstacle-free direction.

Example: `SafetyDistance=2`

Data Types: `single` | `double`

`TargetDirectionWeight` — Cost function weight for moving in target direction
`5` (default) | numeric scalar

Cost function weight for moving in the target direction, specified as a numeric scalar. To prioritize following a target direction, set this weight to a value greater than the sum of `CurrentDirectionWeight` and `PreviousDirectionWeight`. To remove the target direction from consideration in the cost function, set this weight to `0`.

Example: `TargetDirectionWeight=0`

Data Types: `single` | `double`

`CurrentDirectionWeight` — Cost function weight for moving in current direction
2 (default) | numeric scalar

Cost function weight for moving in the current direction, specified as a numeric scalar. Higher values of this weight can produce more efficient paths. To remove the current direction from consideration in the cost function, set this weight to `0`.

Example: `CurrentDirectionWeight=0`

Data Types: `single` | `double`

`PreviousDirectionWeight` — Cost function weight for moving in previous direction
2 (default) | numeric scalar

Cost function weight for moving in the previous direction, specified as a numeric scalar. Higher values of this weight can produce smoother paths. To remove the previous direction from consideration in the cost function, set this weight to `0`.

Example: `PreviousDirectionWeight=1`

Data Types: `single` | `double`

Usage

Syntax

```
[desiredDirection,desiredYaw,status] = vhf3D(position,orientation,
obstaclePoints,targetPosition)
```

Description

`[desiredDirection,desiredYaw,status] = vhf3D(position,orientation, obstaclePoints,targetPosition)` finds an obstacle-free direction and yaw, using the 3DVFH+ algorithm, for the input UAV position, UAV orientation, sensor-data-based positions of obstacles, and target position. This syntax also returns the status `status` of the obstacle-free direction.

Input Arguments

`position` — Position of UAV
vector of form `[x; y; z]`

Position of the UAV, specified as a vector of the form `[x; y; z]`, in meters.

Example: `[1; 1; 1]`

Data Types: `single` | `double`

`orientation` — Orientation of UAV
quaternion vector of form `[w; x; y; z]`

Orientation of the UAV, specified as a quaternion vector of the form `[w; x; y; z]`.

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

Data Types: `single` | `double`

`obstaclePoints` — Positions of obstacles
N-by-3 matrix

Positions of the obstacles, specified as an *N*-by-3 matrix with rows of the form `[x y z]`, in meters. *N* is the number of obstacle points.

Example: `[1 1 1; 2 2 2]`

Data Types: `single` | `double`

`targetPosition` — Target position
vector of form `[x; y; z]`

Target position, specified as a vector of the form `[x; y; z]`, in meters.

Example: `[2; 3; 4]`

Data Types: `single` | `double`

Output Arguments

`desiredDirection` — Desired direction
vector of form `[x; y; z]`

Desired direction, returned as a vector of the form `[x; y; z]`, in meters.

Data Types: `single` | `double`

`desiredYaw` — Desired yaw
numeric scalar in range `[-pi, pi]`

Desired yaw, returned as numeric scalar in the range `[-pi, pi]`, in radians.

Data Types: `single` | `double`

`status` — Status of obstacle-free direction
`0` | `1` | `2` | `3`

Status of the obstacle-free direction, returned as `0`, `1`, `2`, or `3`.

- `0` — The object finds an obstacle-free direction.
- `1` — The object does not find an obstacle-free direction.
- `2` — The object finds an obstacle-free direction, but the direction is close to an obstacle.
- `3` — The object does not find an obstacle-free direction, and the direction is close to an obstacle.

Data Types: `uint8`

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 `controllerVFH3D`

show   Display 3D vector field histogram

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 3D Vector Field Histogram Object and Visualize Data

Create a `controllerVFH3D` object.

```
vfh3D = controllerVFH3D;
```

Create obstacles.

```
az = [-60:-20 20:60]*(pi/180);
el = (-30:30)*(pi/180);
[El,Az] = meshgrid(el,az);
```

Specify the distances of the obstacles from the sensor, and convert to Cartesian coordinates.

```
obstacleDist = linspace(15,20,numel(El(:)));
[xSensor,ySensor,zSensor] = sph2cart(Az(:),El(:),obstacleDist');
```

Align the sensor and histogram frames.

```
vfh3D.SensorOrientation = [-180 0 0];
```

Specify the sensor range limits.

```
vfh3D.DistanceLimits = [0.2 25];
```

Specify the current UAV position and orientation, the locations of obstacles, and the target position for the UAV.

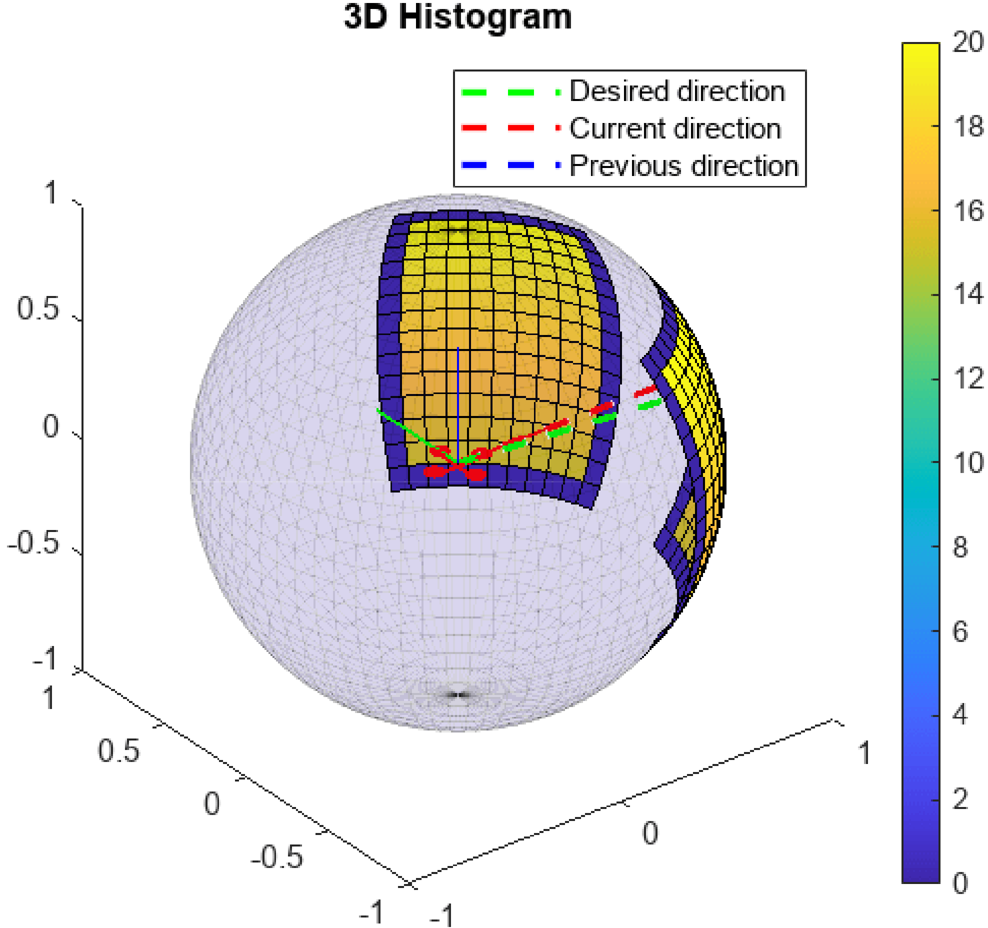
```
uavPosition = [0; 0; 0];
uavOrientation = [1; 0; 0; 0];
sensorPoints = [xSensor ySensor zSensor];
targetPosition = [20; 0; 0];
```

Compute an obstacle-free direction and desired yaw for the UAV, and return the status of the obstacle-free direction.

```
[desiredDirection,desiredYaw,status] = vfh3D(uavPosition, ...
                                             uavOrientation, ...
                                             sensorPoints, ...
                                             targetPosition);
```

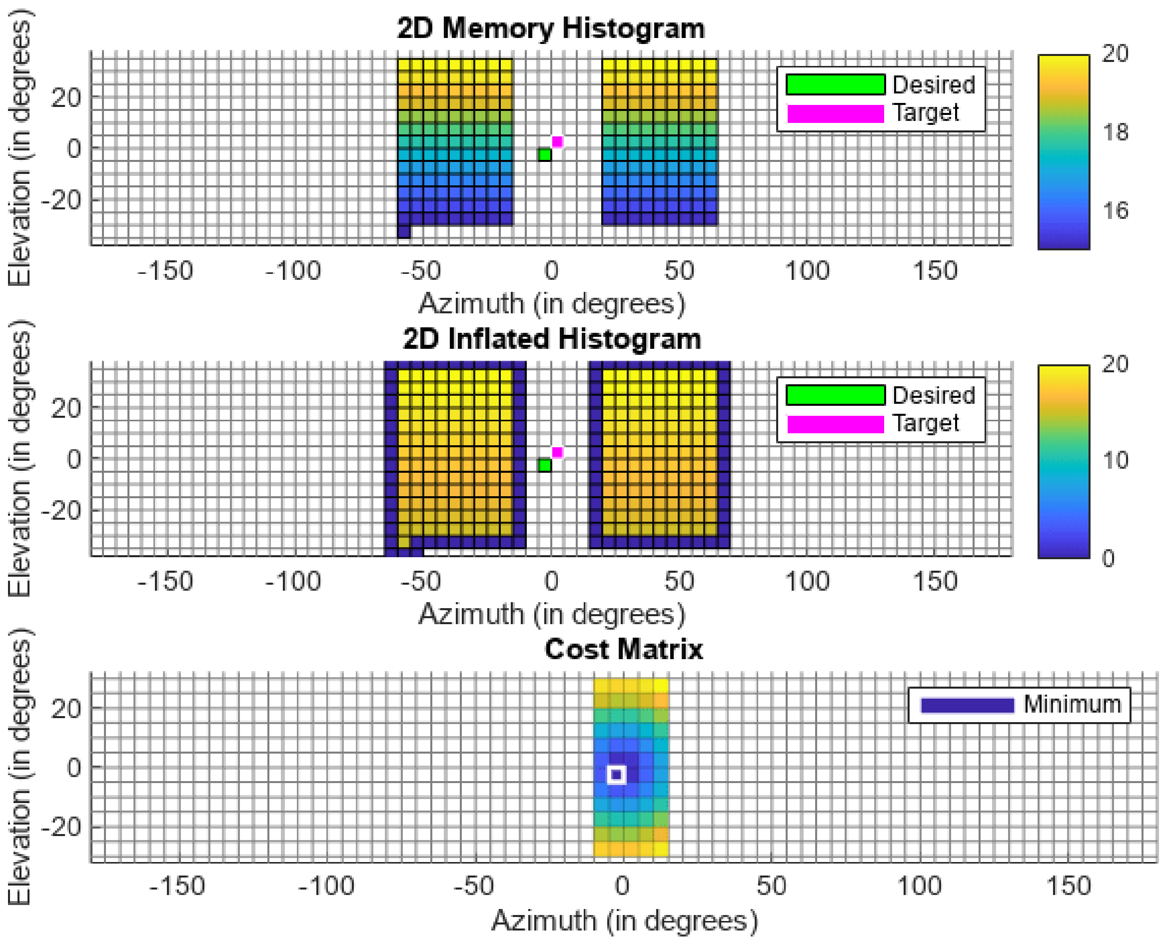
Visualize the default histogram of the calculated direction.

```
show(vfh3D)
axis equal
```



Visualize the 2D memory histogram, 2D inflated histogram, and cost matrix.

```
figure
ax(1) = subplot(3,1,1);
ax(2) = subplot(3,1,2);
ax(3) = subplot(3,1,3);
show(vfh3D, ...
     Parent=ax, ...
     PlotsToShow=["2D Memory Histogram","2D Inflated Histogram","Cost Matrix"])
axis(ax,"equal")
xlim(ax,"tight")
```



Version History

Introduced in R2022b

Extended Capabilities

C/C++ Code Generation

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

See Also

Objects

`uavScenario` | `uavMission`

Functions

`show`

extendedObjectMesh

Mesh representation of extended object

Description

The `extendedObjectMesh` represents the 3-D geometry of an object. The 3-D geometry is represented by faces and vertices. Use these object meshes to specify the geometry of an `uavPlatform` for simulating lidar sensor data using `uavLidarPointCloudGenerator`.

Creation

Syntax

```
mesh = extendedObjectMesh('cuboid')
mesh = extendedObjectMesh('cylinder')
mesh = extendedObjectMesh('cylinder',n)
mesh = extendedObjectMesh('sphere')
mesh = extendedObjectMesh('sphere',n)
mesh = extendedObjectMesh(vertices,faces)
```

Description

`mesh = extendedObjectMesh('cuboid')` returns an `extendedObjectMesh` object, that defines a cuboid with unit dimensions. The origin of the cuboid is located at its geometric center.

`mesh = extendedObjectMesh('cylinder')` returns a hollow cylinder mesh with unit dimensions. The cylinder mesh has 20 equally spaced vertices around its circumference. The origin of the cylinder is located at its geometric center. The height is aligned with the *z*-axis.

`mesh = extendedObjectMesh('cylinder',n)` returns a cylinder mesh with `n` equally spaced vertices around its circumference.

`mesh = extendedObjectMesh('sphere')` returns a sphere mesh with unit dimensions. The sphere mesh has 119 vertices and 180 faces. The origin of the sphere is located at its center.

`mesh = extendedObjectMesh('sphere',n)` additionally allows you to specify the resolution, `n`, of the spherical mesh. The sphere mesh has $(n + 1)^2 - 2$ vertices and $2n(n - 1)$ faces.

`mesh = extendedObjectMesh(vertices,faces)` returns a mesh from faces and vertices. `vertices` and `faces` set the `Vertices` and `Faces` properties respectively.

Properties

`Vertices` — Vertices of defined object
N-by-3 matrix of real scalar

Vertices of the defined object, specified as an *N*-by-3 matrix of real scalars. *N* is the number of vertices. The first, second, and third element of each row represents the *x*-, *y*-, and *z*-position of each vertex, respectively.

Faces — Faces of defined object
M-by-3 matrix of positive integer

Faces of the defined object, specified as a *M*-by-3 array of positive integers. *M* is the number of faces. The three elements in each row are the vertex IDs of the three vertices forming the triangle face. The ID of the vertex is its corresponding row number specified in the `Vertices` property.

Object Functions

Use the object functions to develop new meshes.

translate	Translate mesh along coordinate axes
rotate	Rotate mesh about coordinate axes
scale	Scale mesh in each dimension
applyTransform	Apply forward transformation to mesh vertices
join	Join two object meshes
scaleToFit	Auto-scale object mesh to match specified cuboid dimensions
show	Display the mesh as a patch on the current axes

Examples

Create and Translate Cuboid Mesh

Create an `extendedObjectMesh` object and translate the object.

Construct a cuboid mesh.

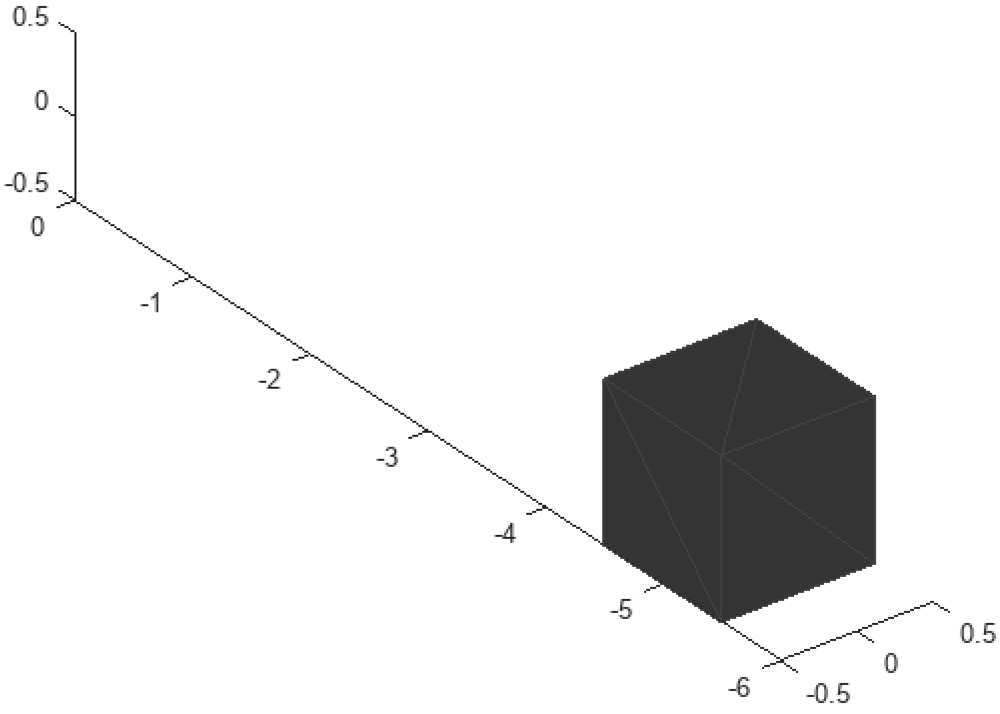
```
mesh = extendedObjectMesh('cuboid');
```

Translate the mesh by 5 units along the negative *y* axis.

```
mesh = translate(mesh,[0 -5 0]);
```

Visualize the mesh.

```
ax = show(mesh);
ax.YLim = [-6 0];
```



Create and Visualize Cylinder Mesh

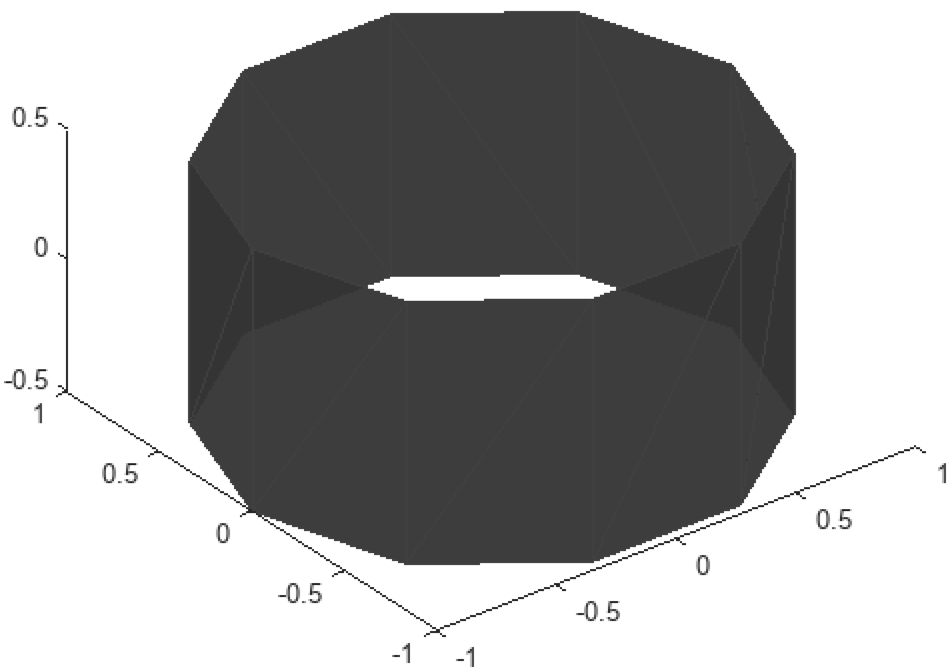
Create an `extendedObjectMesh` object and visualize the object.

Construct a cylinder mesh.

```
mesh = extendedObjectMesh('cylinder');
```

Visualize the mesh.

```
ax = show(mesh);
```



Create and Auto-Scale Sphere Mesh

Create an `extendedObjectMesh` object and auto-scale the object to the required dimensions.

Construct a sphere mesh of unit dimensions.

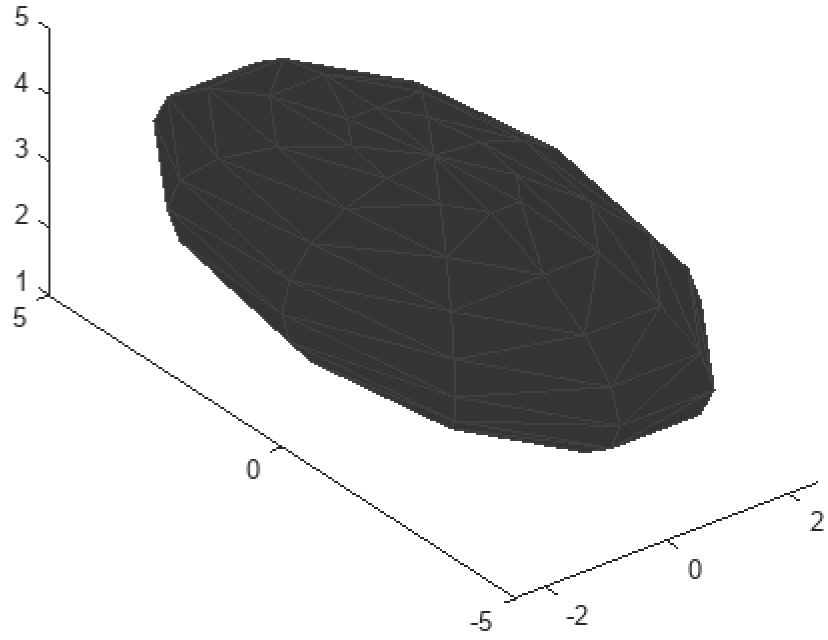
```
sph = extendedObjectMesh('sphere');
```

Auto-scale the mesh to the dimensions in `dims`.

```
dims = struct('Length',5,'Width',10,'Height',3,'OriginOffset',[0 0 -3]);
sph = scaleToFit(sph,dims);
```

Visualize the mesh.

```
show(sph);
```



Version History

Introduced in R2020b

See Also

Objects

`uavPlatform` | `uavLidarPointCloudGenerator`

Functions

`translate` | `rotate` | `scale` | `applyTransform` | `join` | `scaleToFit` | `show`

fixedwing

Guidance model for fixed-wing UAVs

Description

A `fixedwing` object represents a reduced-order guidance model for an unmanned aerial vehicle (UAV). The model approximates the behavior of a closed-loop system consisting of an autopilot controller and a fixed-wing kinematic model for 3-D motion.

For multirotor UAVs, see `multirotor`.

Creation

`model = fixedwing` creates a fixed-wing motion model with `double` precision values for inputs, outputs, and configuration parameters of the guidance model.

`model = fixedwing(DataType)` specifies the data type precision (`DataType` property) for the inputs, outputs, and configurations parameters of the guidance model.

Properties

Name — Name of UAV
`"Unnamed"` (default) | string scalar

Name of the UAV, used to differentiate it from other models in the workspace, specified as a string scalar.

Example: `"myUAV1"`

Data Types: `string`

`Configuration` — UAV controller configuration
structure

UAV controller configuration, specified as a structure of parameters. Specify these parameters to tune the internal control behavior of the UAV. Specify the proportional (P) and derivative (D) gains for the dynamic model and other UAV parameters. The structure for fixed-wing UAVs contains these fields with defaults listed:

- `'PDRoll'` - `[3402.97 116.67]`
- `'PHeight'` - `3.9`
- `'PFlightPathAngle'` - `39`
- `'PAirspeed'` - `0.39`
- `'FlightPathAngleLimits'` - `[-pi/2 pi/2]` (`[min max]` angle in radians)

Example: `struct('PDRoll',[3402.97,116.67],'PHeight',3.9,'PFlightPathAngle',39,'PAirSpeed',0.39,'FlightPathAngleLimits',[-pi/2 pi/2])`

Data Types: `struct`

`ModelType` — UAV guidance model type
`'FixedWingGuidance'` (default)

This property is read-only.

UAV guidance model type, specified as `'FixedWingGuidance'`.

`DataType` — Input and output numeric data types
`'double'` (default) | `'single'`

Input and output numeric data types, specified as either `'double'` or `'single'`. Choose the data type based on possible software or hardware limitations.

Object Functions

control	Control commands for UAV
derivative	Time derivative of UAV states
environment	Environmental inputs for UAV
state	UAV state vector

Examples

Simulate A Fixed-Wing Control Command

This example shows how to use the `fixedwing` guidance model to simulate the change in state of a UAV due to a command input.

Create the fixed-wing guidance model.

```
model = fixedwing;
```

Set the air speed of the vehicle by modifying the structure from the `state` function.

```
s = state(model);
s(4) = 5; % 5 m/s
```

Specify a control command, `u`, that maintains the air speed and gives a roll angle of `pi/12`.

```
u = control(model);
u.RollAngle = pi/12;
u.AirSpeed = 5;
```

Create a default environment without wind.

```
e = environment(model);
```

Compute the time derivative of the state given the current state, control command, and environment.

```
sdot = derivative(model,s,u,e);
```

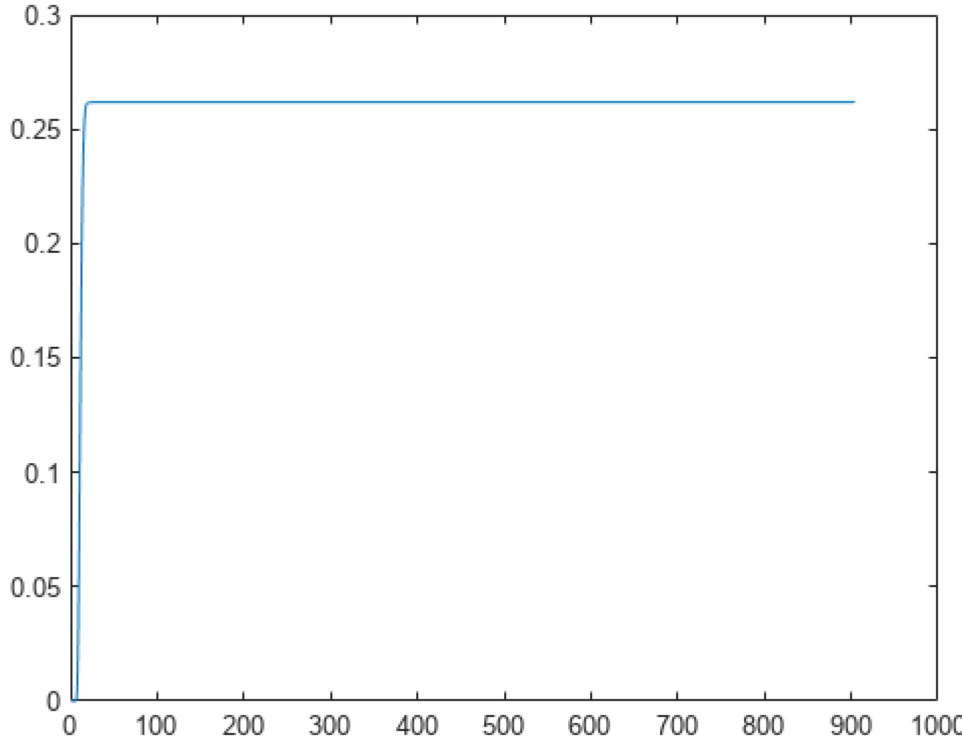
Simulate the UAV state using `ode45` integration. The `y` field outputs the fixed-wing UAV states based on this simulation.

```
simOut = ode45(@(~,x)derivative(model,x,u,e), [0 50], s);
size(simOut.y)
```

```
ans = 1×2

     8   904
```

Plot the change in roll angle based on the simulation output. The roll angle is the 7th row of the `simOut.y` output.

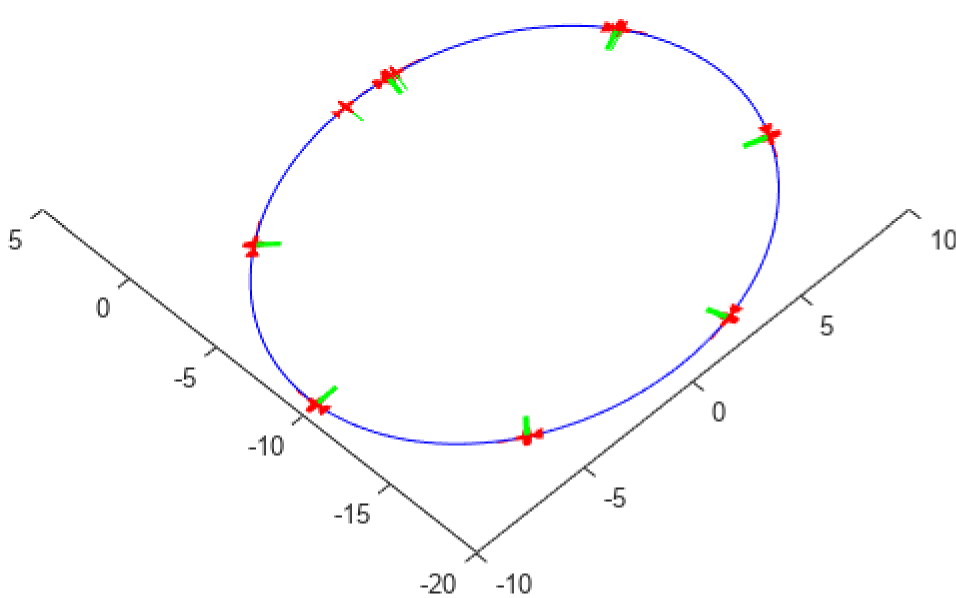
```
plot(simOut.y(7,:))
```



You can also plot the fixed-wing trajectory using `plotTransforms`. Create the translation and rotation vectors from the simulated state. Downsample (every 30th element) and transpose the `simOut` elements, and convert the Euler angles to quaternions. Specify the mesh as the `fixedwing.stl` file and the positive Z-direction as `"down"`. The displayed view shows the UAV making a constant turn based on the constant roll angle.

```
downsample = 1:30:size(simOut.y,2);
translations = simOut.y(1:3,downsample)'; % xyz-position
rotations = eul2quat([simOut.y(5,downsample)',simOut.y(6,downsample)',simOut.y(7,downsample)']);
plotTransforms(translations,rotations,...
    'MeshFilePath','fixedwing.stl','InertialZDirection',"down")
hold on
plot3(simOut.y(1,:),-simOut.y(2,:),simOut.y(3,:),'--b') % full path
xlim([-10.0 10.0])
```

```
ylim([-20.0 5.0])
zlim([-0.5 4.00])
view([-45 90])
hold off
```



More About

UAV Coordinate Systems

The UAV Toolbox uses the North-East-Down (NED) coordinate system convention, which is also sometimes called the local tangent plane (LTP). The UAV position vector consists of three numbers for position along the northern-axis, eastern-axis, and vertical position. The down element complies with the right-hand rule and results in negative values for altitude gain.

The ground plane, or earth frame (NE plane, D = 0), is assumed to be an inertial plane that is flat based on the operation region for small UAV control. The earth frame coordinates are $[x_e, y_e, z_e]$. The body frame of the UAV is attached to the center of mass with coordinates $[x_b, y_b, z_b]$. x_b is the preferred forward direction of the UAV, and z_b is perpendicular to the plane that points downwards when the UAV travels during perfect horizontal flight.

The orientation of the UAV (body frame) is specified in ZYX Euler angles. To convert from the earth frame to the body frame, we first rotate about the z_e-axis by the yaw angle, ψ. Then, rotate about the intermediate y-axis by the pitch angle, ϕ. Then, rotate about the intermediate x-axis by the roll angle, Θ.

The angular velocity of the UAV is represented by $[p,q,r]$ with respect to the body axes, $[x_b,y_b,z_b]$.

UAV Fixed-Wing Guidance Model Equations

For fixed-wing UAVs, the following equations are used to define the guidance model of the UAV. Use the `derivative` function to calculate the time-derivative of the UAV state using these governing equations. Specify the inputs using the `state`, `control`, and `environment` functions.

The UAV position in the earth frame is $[x_e, y_e, h]$ with orientation as heading angle, flight path angle, and roll angle, $[\chi, \gamma, \phi]$ in radians.

The model assumes that the UAV is flying under a coordinated-turn condition, with zero side-slip. The autopilot controls airspeed, altitude, and roll angle. The corresponding equations of motion are:

$$\dot{x}_e = V_g \cos\chi \cos\gamma$$

$$\dot{y}_e = V_g \sin\chi \cos\gamma$$

$$\dot{h} = V_g \sin\gamma$$

$$\dot{\chi} = \frac{g\cos(\chi - \psi)}{V_g}\tan\phi$$

$$V_g \sin(\gamma^c) = min\left(max\left(k_h(h^c - h), -V_g\right), V_g\right)$$

$$\dot{\gamma} = k_\gamma(\gamma^c - \gamma)$$

$$\dot{V}_a = k_{V_a}(V_a^c - V_a)$$

$$\frac{g\cos(\chi - \psi)}{V_g}\tan(\phi^c) = k_\chi(\chi^c - \chi)$$

$$\ddot{\phi} = k_{P\phi}(\phi^c - \phi) + k_{D\phi}(-\dot{\phi})$$

V_a and V_g denote the UAV air and ground speeds.

The wind speed is specified as $[V_{w_n},V_{w_e},V_{w_d}]$ for the north, east, and down directions. To generate the structure for these inputs, use the `environment` function.

k_* are controller gains. To specify these gains, use the `Configuration` property of the `fixedwing` object.

From these governing equations, the model gives the following variables:

$$[x_e \quad y_e \quad h \quad V_a \quad \chi \quad \gamma \quad \phi \quad \dot{\phi}]$$

These variables match the output of the `state` function.

Version History

Introduced in R2018b

References

[1] Randal W. Beard and Timothy W. McLain. "Chapter 9." *Small Unmanned Aircraft Theory and Practice*, NJ: Princeton University Press, 2012.

Extended Capabilities

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

See Also

Functions
`ode45` | `control` | `derivative` | `environment` | `state` | `plotTransforms`

Objects
`multirotor` | `uavWaypointFollower`

Blocks
UAV Guidance Model | Waypoint Follower

Topics
"Approximate High-Fidelity UAV Model with UAV Guidance Model Block"
"Tuning Waypoint Follower for Fixed-Wing UAV"

flightLogSignalMapping

Visualize UAV flight logs

Description

The `flightLogSignalMapping` provides visualization tools to analyze flight logs. To inspect UAV logs, first load your file using a file or log reader like `mavlinktlog` or `ulogreader`. Use preconfigured signal mapping and plots from ULOG or TLOG log files, or define your own signal mapping using `mapSignal`. Update or add new plots with `updatePlot`. Then, call `show` with a structure of data to display the list of configured plots defined in the AvailablePlots property.

For ease of use, specific Predefined Signals on page 1-22 and Predefined Plots on page 1-23 are provided. Details are listed below or can be viewed by calling `info` for your specific object.

Creation

Description

`mapper = flightLogSignalMapping` creates a flight log signal mapping object with no preset signal mapping. Before you can visualize signals, map signals using `mapSignal`.

`mapper = flightLogSignalMapping("tlog")` creates a flight log signal mapping object for the imported MAVLink TLOG message tables.

`mapper = flightLogSignalMapping("ulog")` creates a flight log signal mapping object for imported PX4® ULOG files.

Properties

`MappedSignals` — Names of all mapped signals
string array

Names of all mapped signals, specified as a string array.

Example: `["Accel" "Gyro" "Mag" "Barometer" "Gyro2"]`

Data Types: `string`

`AvailablePlots` — Names of plots that are available
string array

Names of plots that are available based on the mapped signals, specified as a string array. To add plots to this list, either map signals for the "Predefined Plots" on page 1-23 or call `updatePlot`.

Example: `["Accel" "Gyro" "Mag" "Barometer" "Gyro2"]`

Data Types: `string`

Object Functions

checkSignal	Check mapped signal
copy	Create deep copy of flight log signal mapping object
extract	Extract UAV flight log signals as timetables
info	Signal mapping and plot information for UAV log signal mapping
mapSignal	Map UAV flight log signal
show	Display plots for inspection of UAV logs
updatePlot	Update UAV flight log plot functions

More About

Predefined Signals

A set of predefined signals and plots are configured in the `flightLogSignalMapping` object. Depending on your log file type, you can map specific signals to the provided signal names using `mapSignal`. You can also call `info` to view the table for your log type and see whether you have already mapped a signal to that plot type.

Specify the `SignalName` as the input to `mapSignal`. Signals with the format `SignalName#` support mapping multiple signals of the same type. Replace # with incremental integers for each signal name when calling `mapSignal`.

The predefined signals have specific names and required fields when mapping the signal.

Predefined Signals

Signal Name	Description	Fields	Unit
Accel#	Raw magnetometer reading from IMU sensor	[ax ay az]	m/s²
Airspeed#	Airspeed reading of pressure differential, indicated air speed, and temperature	[PressDiff, AirSpeed, Temp]	Pa, r
AttitudeEuler	Attitude of UAV in Euler (ZYX) form	[Roll, Pitch, Yaw]	radi
AttitudeRate	Angular velocity along each body axis	[xRotRate, yRotRate, zRotRate]	rad/
AttitudeTargetEuler	Target attitude of UAV in Euler (ZYX) form	[TargetRoll, TargetPitch, TargetYaw]	radi
Barometer#	Barometer readings for absolute pressure, relative pressure, and temperature	[PressAbs, PressAltitude, Temp]	Pa, r
Battery	Voltage readings for battery and remaining battery capacity (%)	[Volt1,Volt2, ... Volt16, RemainingCapacity	V, %
GPS#	GPS readings for latitude, longitude, altitude, ground speed, course angle, and number of satellites visible	[lat, long, alt, groundspeed, courseAngle, satellites]	degr degr
Gyro#	Raw body angular velocity readings from IMU sensor	[GyroX, GyroY, GyroZ]	rad/
LocalNED	Local NED coordinates estimated by the UAV	[xNED, yNED, zNED]	met
LocalNEDTarget	Target location in local NED coordinates	[xTarget, yTarget, zTarget]	met
LocalNEDVel	Local NED velocity estimated by the UAV	[vx vy vz]	m/s
LocalNEDVelTarget	Target velocity in NED in local NED	[vxTarget, vyTarget, vzTarget]	m/s
Mag#	Raw magnetometer reading from IMU sensor	[x y z]	Gs

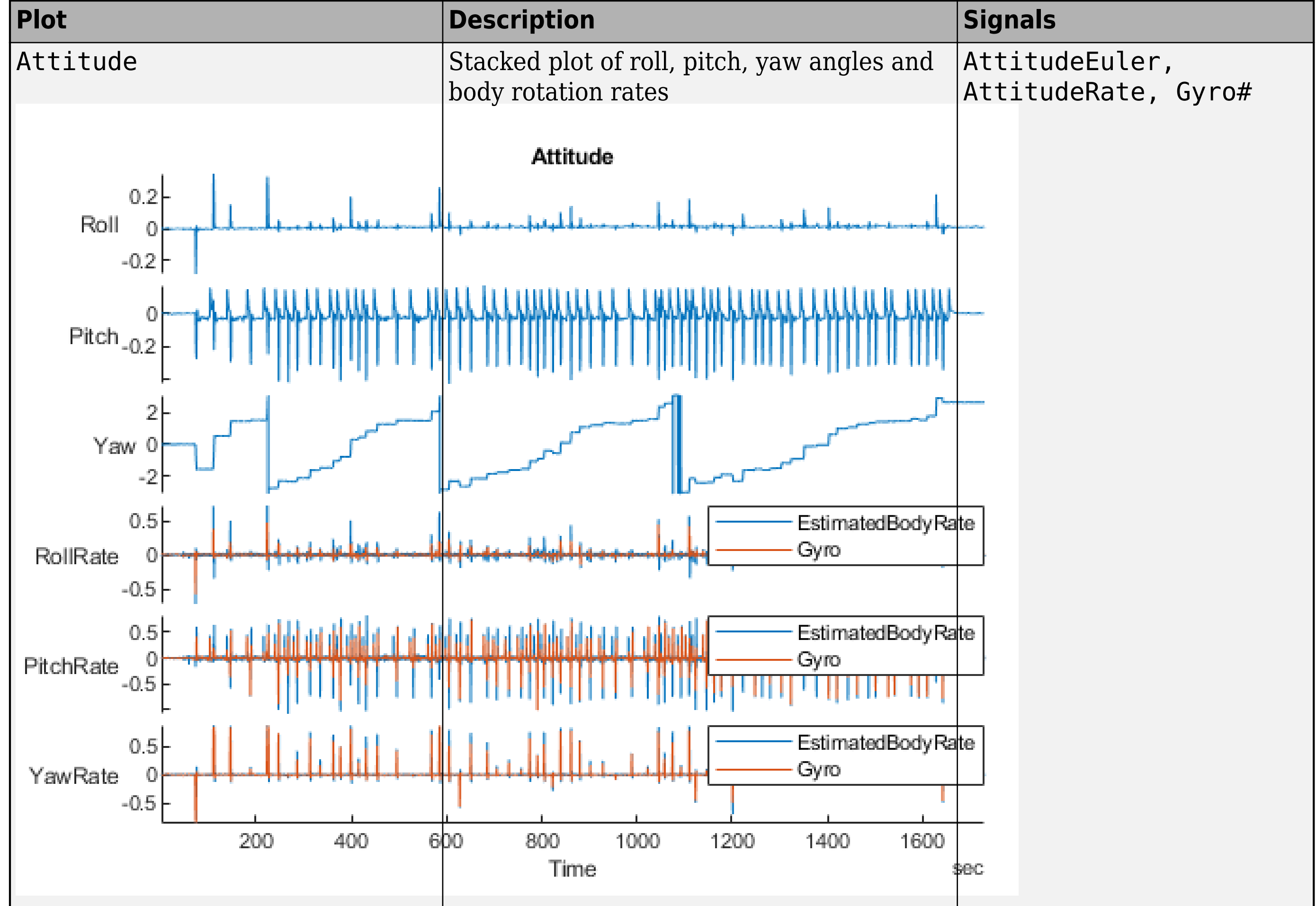
Predefined Plots

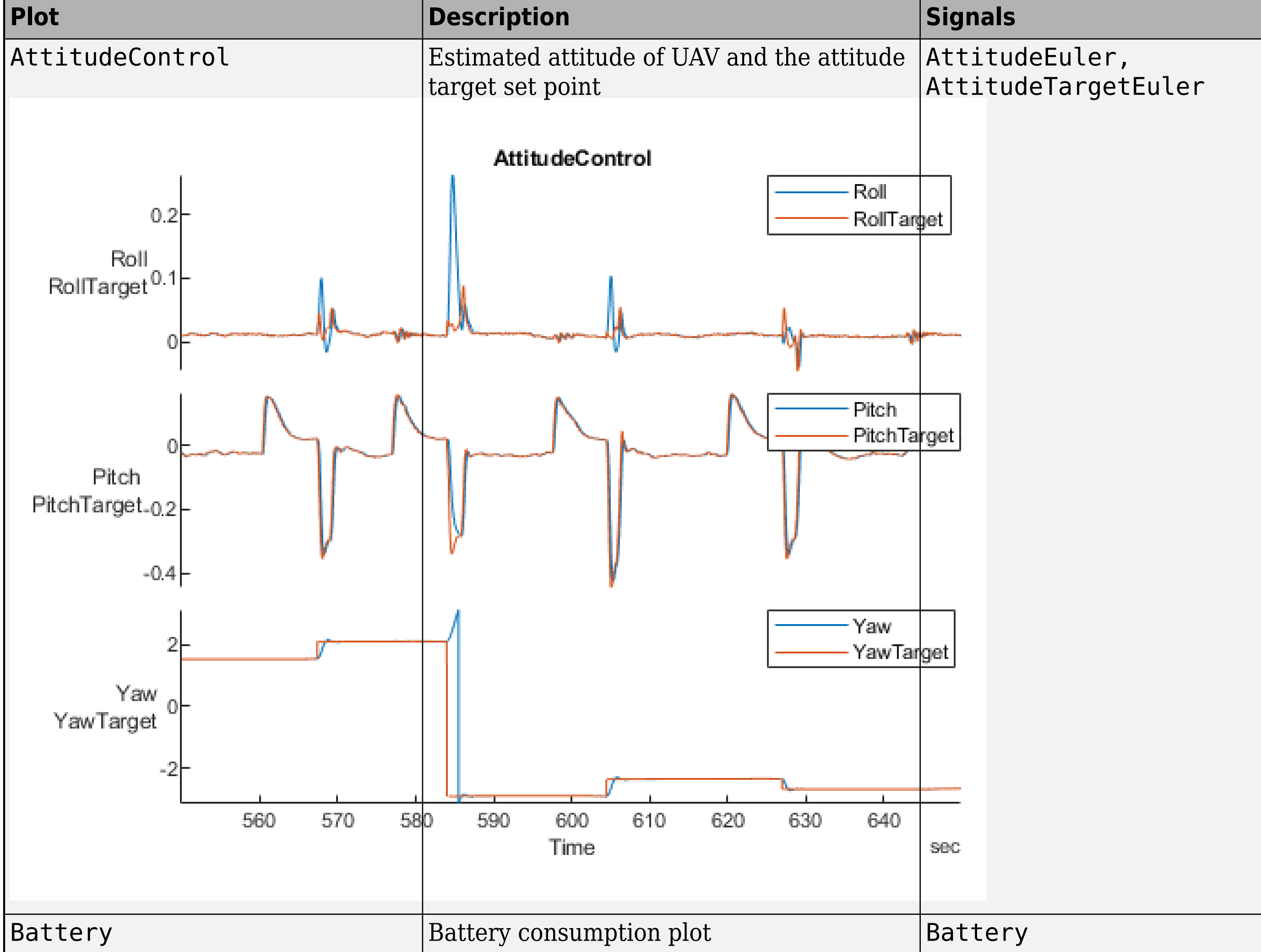
After mapping signals to the list of predefined signals using `mapSignal`, specific plots are made available when calling `show`. To view a list of available plots and their associated signals for your specific object, call `info(mapper,"Plot")`. If you want to define custom plots based on signals, use `updatePlot`.

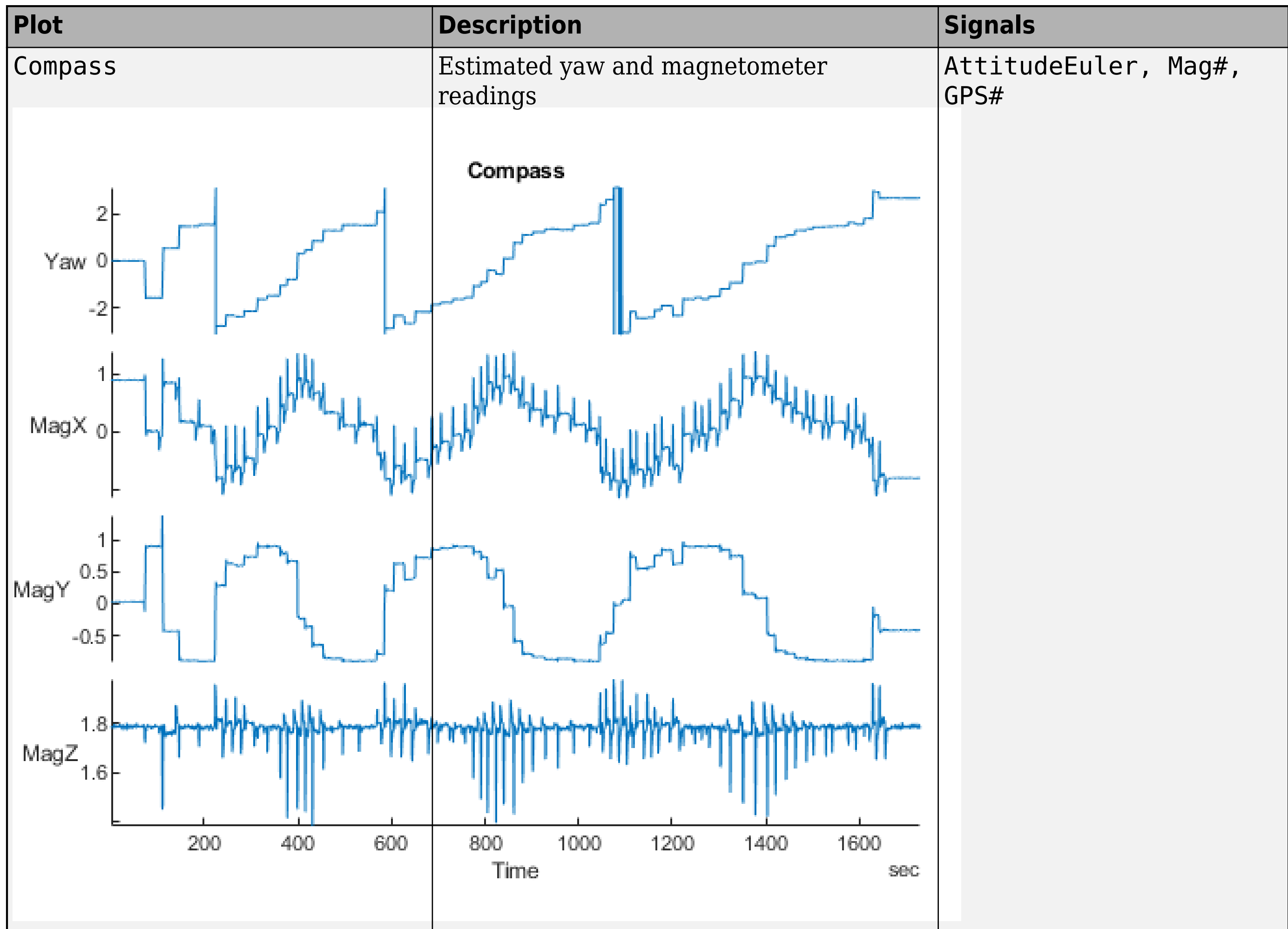
Each predefined plot has a set of required signals that must be mapped.

Predefined Plots

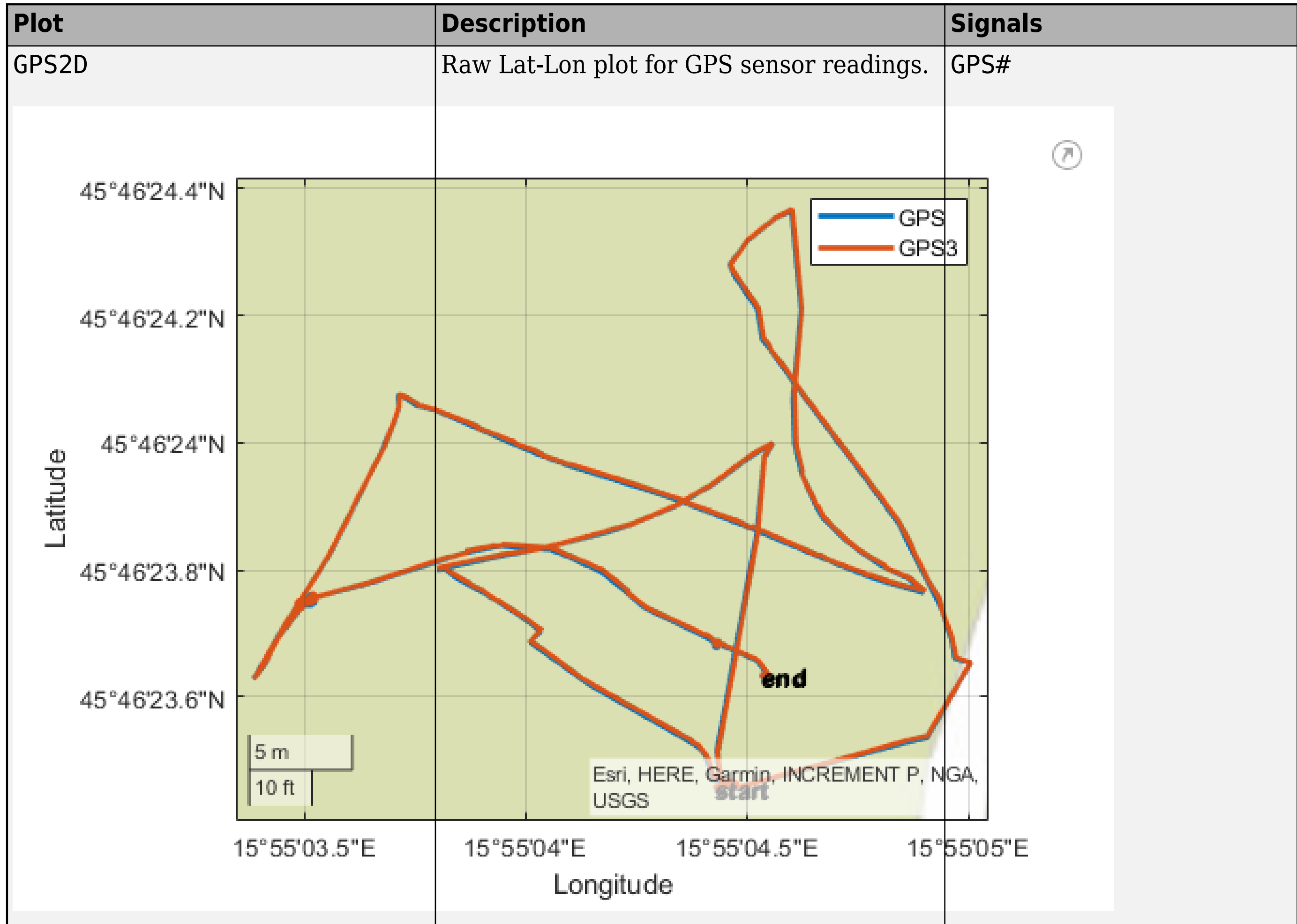
Plot	Description	Signals
`Attitude`	Stacked plot of roll, pitch, yaw angles and body rotation rates	`AttitudeEuler, AttitudeRate, Gyro#`

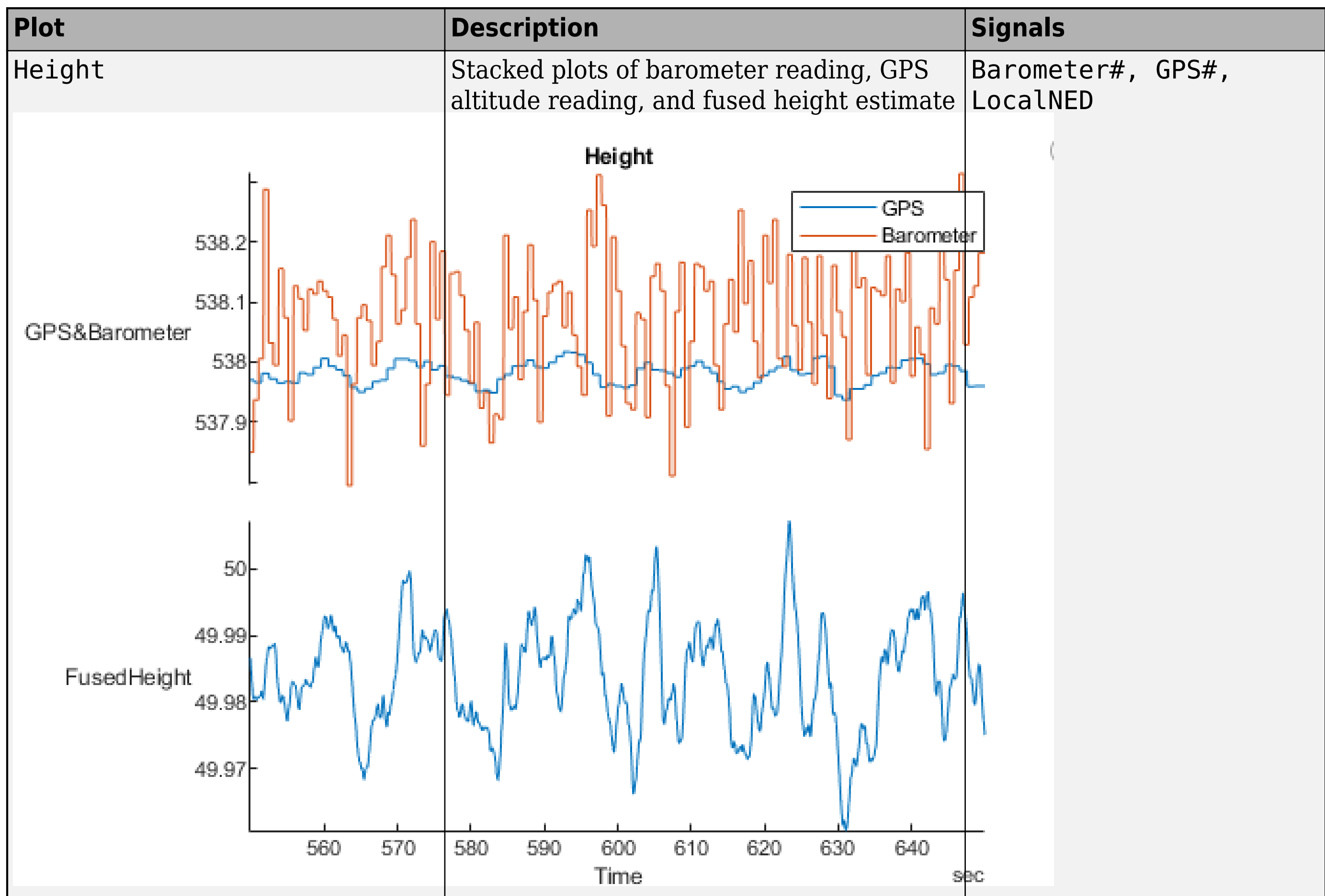


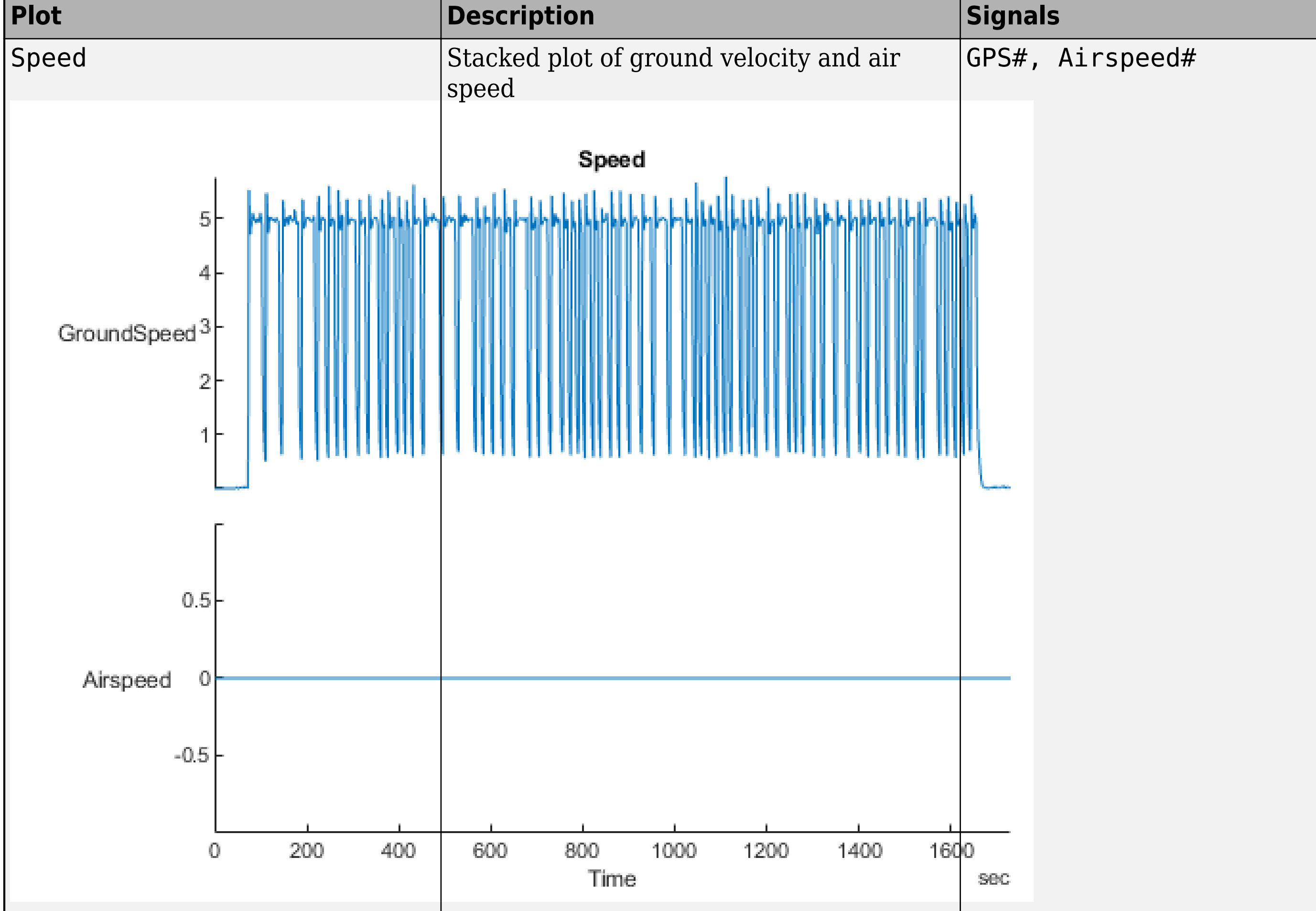
Plot	Description	Signals
AttitudeControl  	Estimated attitude of UAV and the attitude target set point	AttitudeEuler, AttitudeTargetEuler
Battery	Battery consumption plot	Battery

Plot	Description	Signals
`Compass` 	Estimated yaw and magnetometer readings	`AttitudeEuler, Mag#, GPS#`

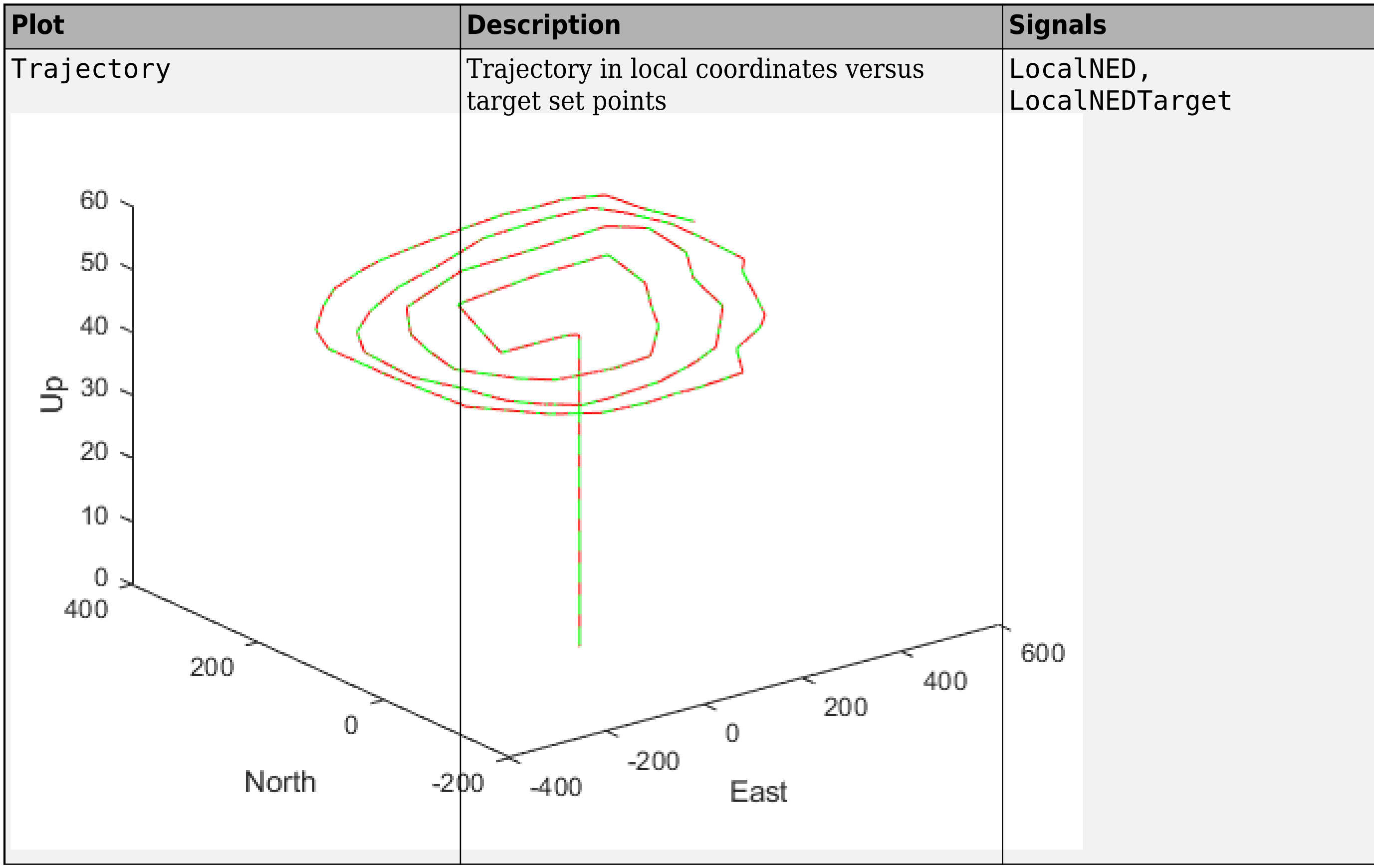
Plot	Description	Signals
GPS2D	Raw Lat-Lon plot for GPS sensor readings.	GPS#



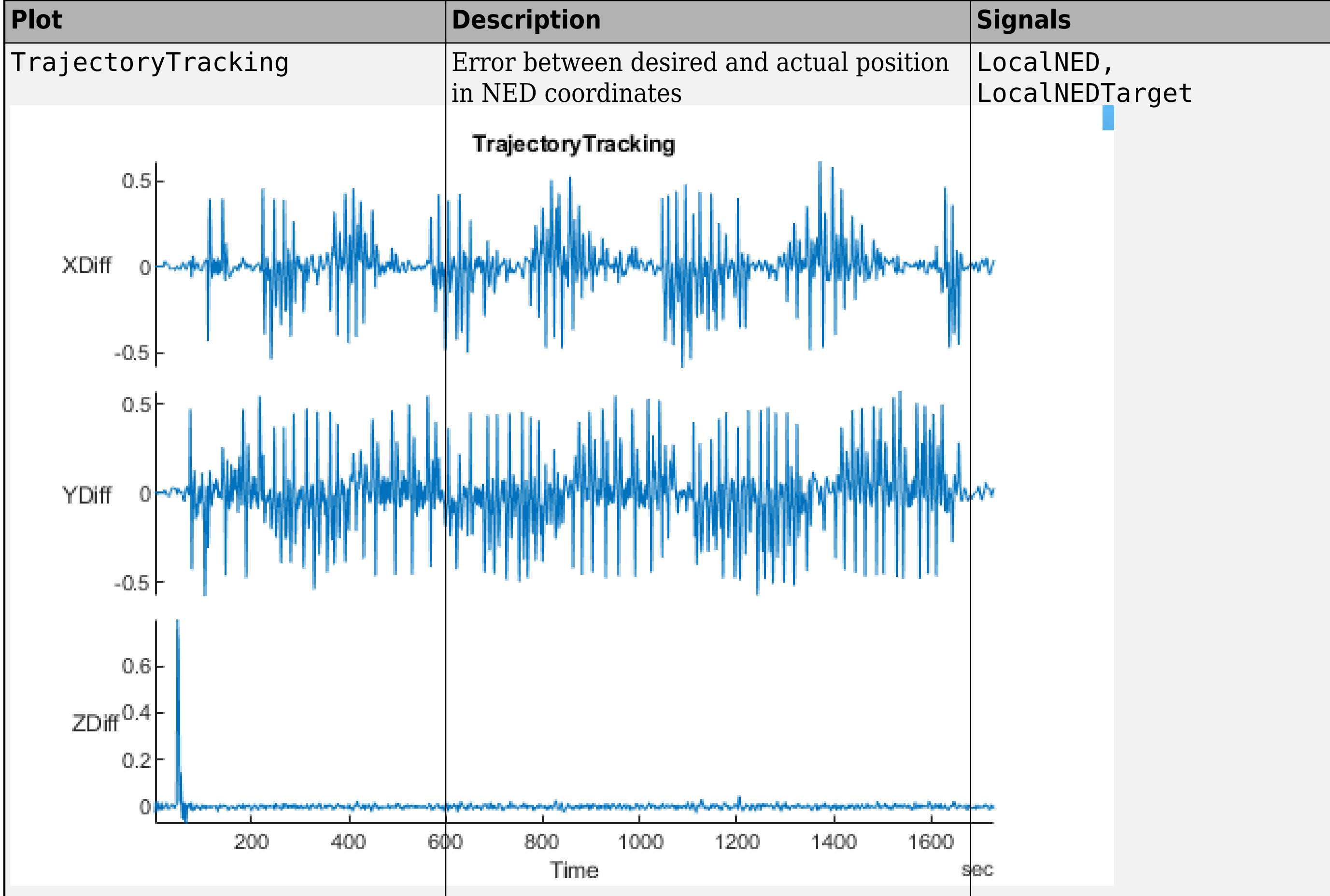
Plot	Description	Signals
`Height` 	Stacked plots of barometer reading, GPS altitude reading, and fused height estimate	`Barometer#, GPS#, LocalNED`

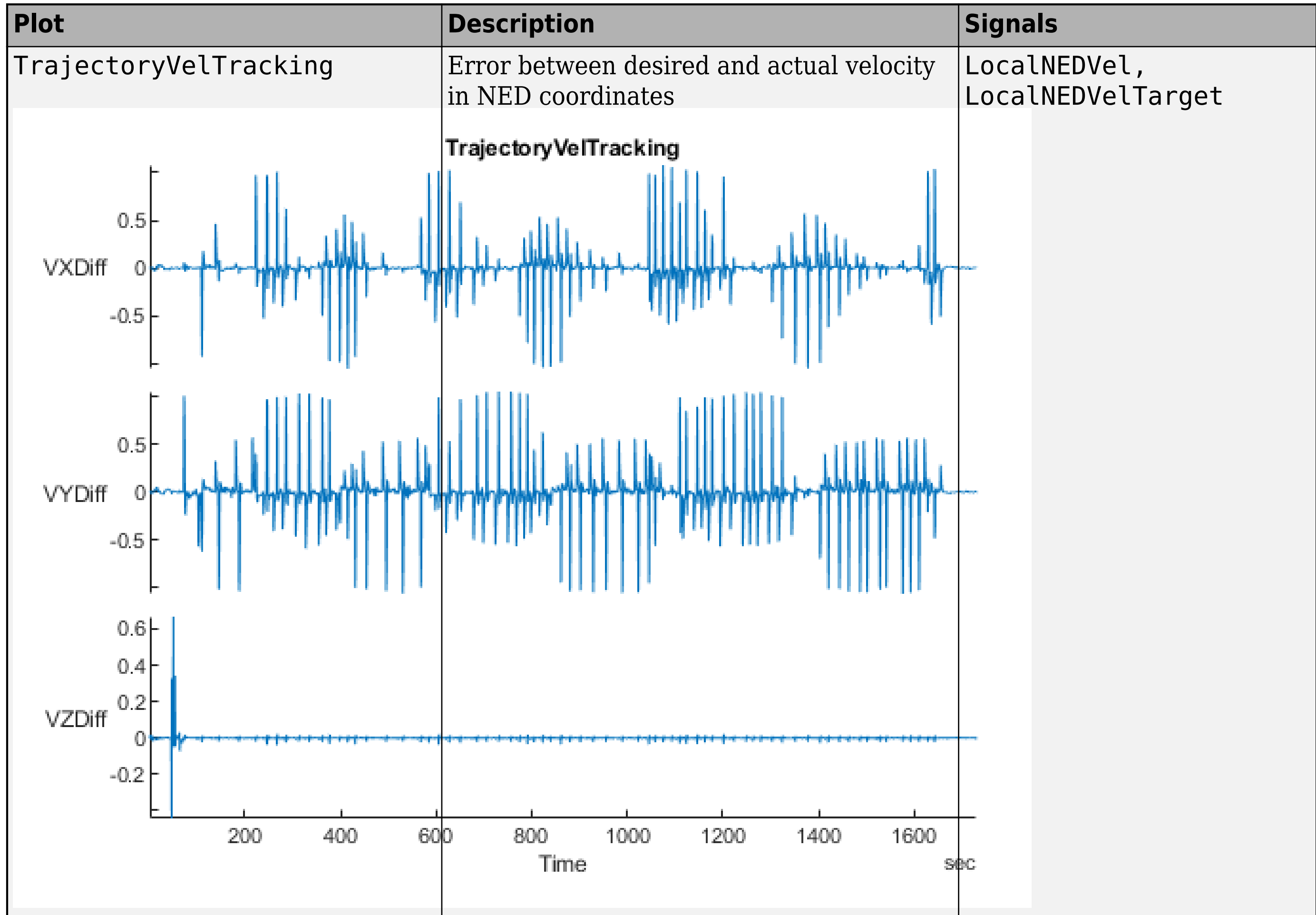
Plot	Description	Signals
`Speed` 	Stacked plot of ground velocity and air speed	`GPS#, Airspeed#`

Plot	Description	Signals
`Trajectory`	Trajectory in local coordinates versus target set points	`LocalNED`, `LocalNEDTarget`



Plot	Description	Signals
TrajectoryTracking	Error between desired and actual position in NED coordinates	LocalNED, LocalNEDTarget



Plot	Description	Signals
`TrajectoryVelTracking`	Error between desired and actual velocity in NED coordinates	`LocalNEDVel`, `LocalNEDVelTarget`
		

Version History

Introduced in R2020b

See Also

`mavlinktlog`

gpsSensor

GPS receiver simulation model

Description

The `gpsSensor` System object models data output from a Global Positioning System (GPS) receiver. The object models the position noise as a first order Gauss Markov process, in which the sigma values are specified in the `HorizontalPositionAccuracy` and the `VerticalPositionAccuracy` properties. The object models the velocity noise as Gaussian noise with its sigma value specified in the `VelocityAccuracy` property.

To model a GPS receiver:

1. Create the `gpsSensor` object and set its properties.
2. Call the object with arguments, as if it were a function.

To learn more about how System objects work, see What Are System Objects?

Creation

Syntax

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

Description

`GPS = gpsSensor` returns a `gpsSensor` System object that computes a Global Positioning System receiver reading based on a local position and velocity input signal. The default reference position in geodetic coordinates is

- latitude: 0º N
- longitude: 0º E
- altitude: 0 m

`GPS = gpsSensor('ReferenceFrame',RF)` returns a `gpsSensor` System object that computes a global positioning system receiver reading relative to the reference frame `RF`. Specify `RF` as `'NED'` (North-East-Down) or `'ENU'` (East-North-Up). The default value is `'NED'`.

`GPS = gpsSensor( ___ ,Name,Value)` sets each property `Name` to the specified `Value`. Unspecified properties have default values.

Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see System Design in MATLAB Using System Objects.

`SampleRate` — Update rate of receiver (Hz)
`1` (default) | positive real scalar

Update rate of the receiver in Hz, specified as a positive real scalar.

Data Types: `single` | `double`

`ReferenceLocation` — Origin of local navigation reference frame
`[0 0 0]` (default) | `[latitude longitude altitude]`

Reference location, specified as a 3-element row vector in geodetic coordinates (latitude, longitude, and altitude). Altitude is the height above the reference ellipsoid model, WGS84. The reference location is in `[degrees degrees meters]`. The degree format is decimal degrees (DD).

Data Types: `single` | `double`

`PositionInputFormat` — Position coordinate input format
`'Local'` (default) | `'Geodetic'`

Position coordinate input format, specified as `'Local'` or `'Geodetic'`.

- If you set the property as `'Local'`, then you need to specify the `truePosition` input as Cartesian coordinates with respect to the local navigation frame whose origin is fixed and defined by the `ReferenceLcation` property. Additionally, when you specify the `trueVelocity` input, you need to specify it with respect to this local navigation frame.
- If you set the property as `'Geodetic'`, then you need to specify the `truePosition` input as geodetic coordinates in latitude, longitude, and altitude. Additionally, when you specify the `trueVelocity` input, you need to specify it with respect to the navigation frame (NED or ENU) whose origin corresponds to the `truePosition` input. When setting the property as `'Geodetic'`, the `gpsSensor` object neglects the `ReferenceLocation` property.

Data Types: `character vector`

`HorizontalPositionAccuracy` — Horizontal position accuracy (m)
`1.6` (default) | nonnegative real scalar

Horizontal position accuracy in meters, specified as a nonnegative real scalar. The horizontal position accuracy specifies the standard deviation of the noise in the horizontal position measurement.

Tunable: Yes

Data Types: `single` | `double`

`VerticalPositionAccuracy` — Vertical position accuracy (m)
`3` (default) | nonnegative real scalar

Vertical position accuracy in meters, specified as a nonnegative real scalar. The vertical position accuracy specifies the standard deviation of the noise in the vertical position measurement.

Tunable: Yes

Data Types: `single` | `double`

`VelocityAccuracy` — Velocity accuracy (m/s)
`0.1` (default) | nonnegative real scalar

Velocity accuracy in meters per second, specified as a nonnegative real scalar. The velocity accuracy specifies the standard deviation of the noise in the velocity measurement.

Tunable: Yes

Data Types: `single` | `double`

`DecayFactor` — Global position noise decay factor
`0.999` (default) | scalar in the range [0,1]

Global position noise decay factor, specified as a scalar in the range [0,1].

A decay factor of 0 models the global position noise as a white noise process. A decay factor of 1 models the global position noise as a random walk process.

Tunable: Yes

Data Types: `single` | `double`

`RandomStream` — Random number source
`'Global stream'` (default) | `'mt19937ar with seed'`

Random number source, specified as a character vector or string:

- `'Global stream'` –– Random numbers are generated using the current global random number stream.
- `'mt19937ar with seed'` –– Random numbers are generated using the mt19937ar algorithm with the seed specified by the `Seed` property.

Data Types: `char` | `string`

Seed — Initial seed
`67` (default) | nonnegative integer scalar

Initial seed of an mt19937ar random number generator algorithm, specified as a nonnegative integer scalar.

Dependencies

To enable this property, set `RandomStream` to `'mt19937ar with seed'`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Usage

Syntax

```
[position,velocity,groundspeed,course] = GPS(truePosition,trueVelocity)
```

Description

`[position,velocity,groundspeed,course] = GPS(truePosition,trueVelocity)` computes global navigation satellite system receiver readings from the position and velocity inputs.

Input Arguments

`truePosition` — Position of GPS receiver in navigation coordinate system
N-by-3 matrix

Position of the GPS receiver in the navigation coordinate system, specified as a real finite *N*-by-3 matrix. *N* is the number of samples in the current frame.

- When the `PositionInputFormat` property is specified as `'Local'`, specify `truePosition` as Cartesian coordinates with respect to the local navigation frame whose origin is fixed at `ReferenceLocation`.
- When the `PositionInputFormat` property is specified as `'Geodetic'`, specify `truePosition` as geodetic coordinates in `[latitude longitude altitude]`. `Latitude` and `longitude` are in meters. `altitude` is the height above the WGS84 ellipsoid model in meters.

Data Types: `single` | `double`

`trueVelocity` — Velocity of GPS receiver in navigation coordinate system (m/s)
N-by-3 matrix

Velocity of GPS receiver in the navigation coordinate system in meters per second, specified as a real finite *N*-by-3 matrix. *N* is the number of samples in the current frame.

- When the `PositionInputFormat` property is specified as `'Local'`, specify `trueVelocity` with respect to the local navigation frame (NED or ENU) whose origin is fixed at `ReferenceLocation`.
- When the `PositionInputFormat` property is specified as `'Geodetic'`, specify `trueVelocity` with respect to the navigation frame (NED or ENU) whose origin corresponds to the `truePosition` input.

Data Types: `single` | `double`

Output Arguments

`position` — Position in LLA coordinate system
N-by-3 matrix

Position of the GPS receiver in the geodetic latitude, longitude, and altitude (LLA) coordinate system, returned as a real finite *N*-by-3 array. Latitude and longitude are in degrees with North and East being positive. Altitude is in meters.

N is the number of samples in the current frame.

Data Types: `single` | `double`

`velocity` — Velocity in local navigation coordinate system (m/s)
N-by-3 matrix

Velocity of the GPS receiver in the local navigation coordinate system in meters per second, returned as a real finite *N*-by-3 array. *N* is the number of samples in the current frame.

- When the `PositionInputFormat` property is specified as `'Local'`, the returned velocity is with respect to the local navigation frame whose origin is fixed at `ReferenceLocation`.

- When the `PositionInputFormat` property is specified as `'Geodetic'`, the returned velocity is with respect to the navigation frame (NED or ENU) whose origin corresponds to the `position` output.

Data Types: `single` | `double`

`groundspeed` — Magnitude of horizontal velocity in local navigation coordinate system (m/s)
N-by-1 column vector

Magnitude of the horizontal velocity of the GPS receiver in the local navigation coordinate system in meters per second, returned as a real finite *N*-by-1 column vector.

N is the number of samples in the current frame.

Data Types: `single` | `double`

`course` — Direction of horizontal velocity in local navigation coordinate system (°)
N-by-1 column vector

Direction of the horizontal velocity of the GPS receiver in the local navigation coordinate system in degrees, returned as a real finite *N*-by-1 column of values between 0 and 360. North corresponds to 360 degrees and East corresponds to 90 degrees.

N is the number of samples in the current frame.

Data Types: `single` | `double`

Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

Common to All System Objects

step	Run System object algorithm
release	Release resources and allow changes to System object property values and input characteristics
reset	Reset internal states of System object

Examples

Generate GPS Position Measurements From Stationary Input

Create a `gpsSensor` System object™ to model GPS receiver data. Assume a typical one Hz sample rate and a 1000-second simulation time. Define the reference location in terms of latitude, longitude, and altitude (LLA) of Natick, MA (USA). Define the sensor as stationary by specifying the true position and velocity with zeros.

```
fs = 1;
duration = 1000;
numSamples = duration*fs;
```

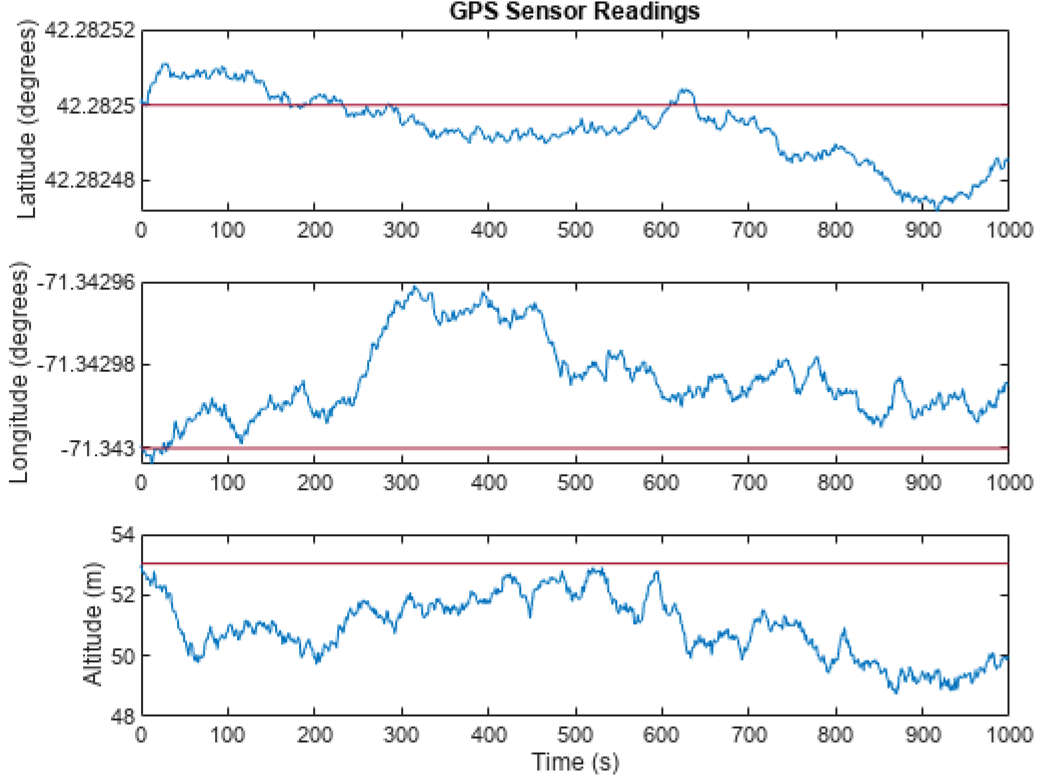
```
refLoc = [42.2825 -71.343 53.0352];

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

gps = gpsSensor('SampleRate',fs,'ReferenceLocation',refLoc);
```

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

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

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

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

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

subplot(3, 1, 2)
plot(t, position(:,2), ...
     t, ones(numSamples)*refLoc(2))
ylabel('Longitude (degrees)')

subplot(3, 1, 3)
plot(t, position(:,3), ...
     t, ones(numSamples)*refLoc(3))
ylabel('Altitude (m)')
xlabel('Time (s)')
```



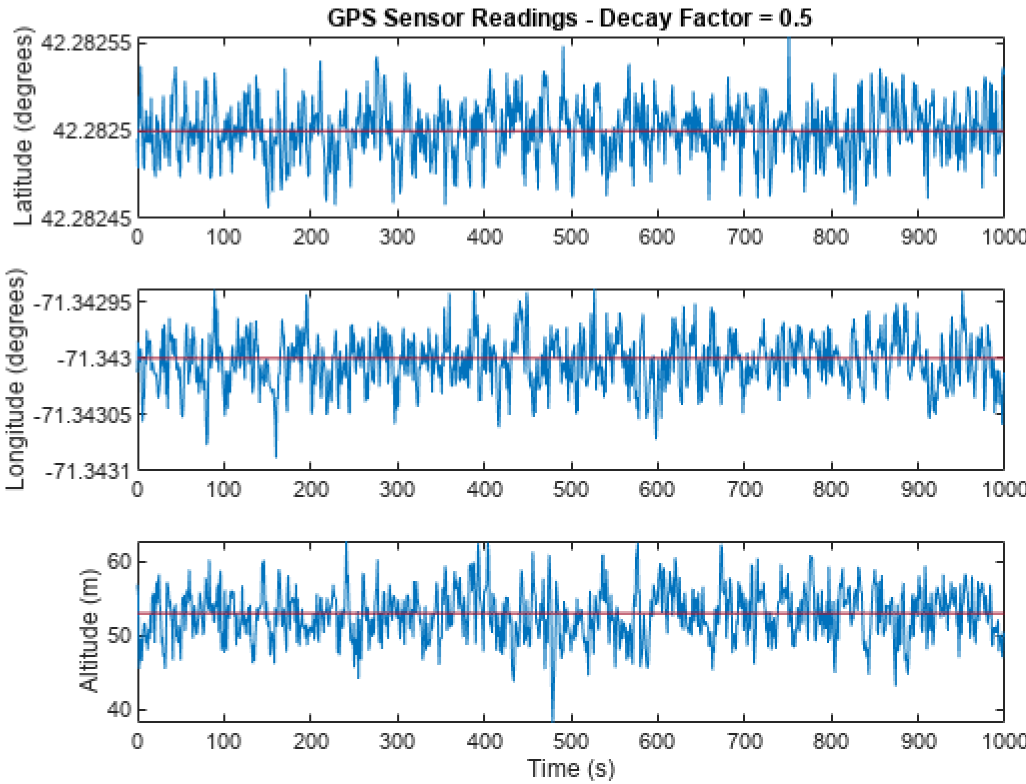
The position readings have noise controlled by `HorizontalPositionAccuracy`, `VerticalPositionAccuracy`, `VelocityAccuracy`, and `DecayFactor`. The `DecayFactor` property controls the drift in the noise model. By default, `DecayFactor` is set to `0.999`, which approaches a random walk process. To observe the effect of the `DecayFactor` property:

1. Reset the `gps` object.
2. Set `DecayFactor` to `0.5`.
3. Call `gps` with variables specifying a stationary position.
4. Plot the results.

The `GPS` position readings now oscillate around the true position.

```
reset(gps)
gps.DecayFactor = 0.5;
position = gps(truePosition,trueVelocity);

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

subplot(3, 1, 2)
plot(t, position(:,2), ...
     t, ones(numSamples)*refLoc(2))
ylabel('Longitude (degrees)')
```

```
subplot(3, 1, 3)
plot(t, position(:,3), ...
     t, ones(numSamples)*refLoc(3))
ylabel('Altitude (m)')
xlabel('Time (s)')
```



Relationship Between Groundspeed and Course Accuracy

GPS receivers achieve greater course accuracy as groundspeed increases. In this example, you create a GPS receiver simulation object and simulate the data received from a platform that is accelerating from a stationary position.

Create a default `gpsSensor` System object™ to model data returned by a GPS receiver.

```
GPS = gpsSensor
```

```
GPS = 
  gpsSensor with properties:

                   SampleRate: 1                      Hz       
          PositionInputFormat: 'Local'                         
            ReferenceLocation: [0 0 0]                [deg deg m]
    HorizontalPositionAccuracy: 1.6                   m        
      VerticalPositionAccuracy: 3                     m        
```

```
      VelocityAccuracy: 0.1                m/s
          RandomStream: 'Global stream'
           DecayFactor: 0.999
```

Create matrices to describe the position and velocity of a platform in the NED coordinate system. The platform begins from a stationary position and accelerates to 60 m/s North-East over 60 seconds, then has a vertical acceleration to 2 m/s over 2 seconds, followed by a 2 m/s rate of climb for another 8 seconds. Assume a constant velocity, such that the velocity is the simple derivative of the position.

```
duration = 70;
numSamples = duration*GPS.SampleRate;

course = 45*ones(duration,1);
groundspeed = [(1:60)';60*ones(10,1)];

Nvelocity   = groundspeed.*sind(course);
Evelocity   = groundspeed.*cosd(course);
Dvelocity   = [zeros(60,1);-1;-2*ones(9,1)];
NEDvelocity = [Nvelocity,Evelocity,Dvelocity];

Ndistance   = cumsum(Nvelocity);
Edistance   = cumsum(Evelocity);
Ddistance   = cumsum(Dvelocity);
NEDposition = [Ndistance,Edistance,Ddistance];
```

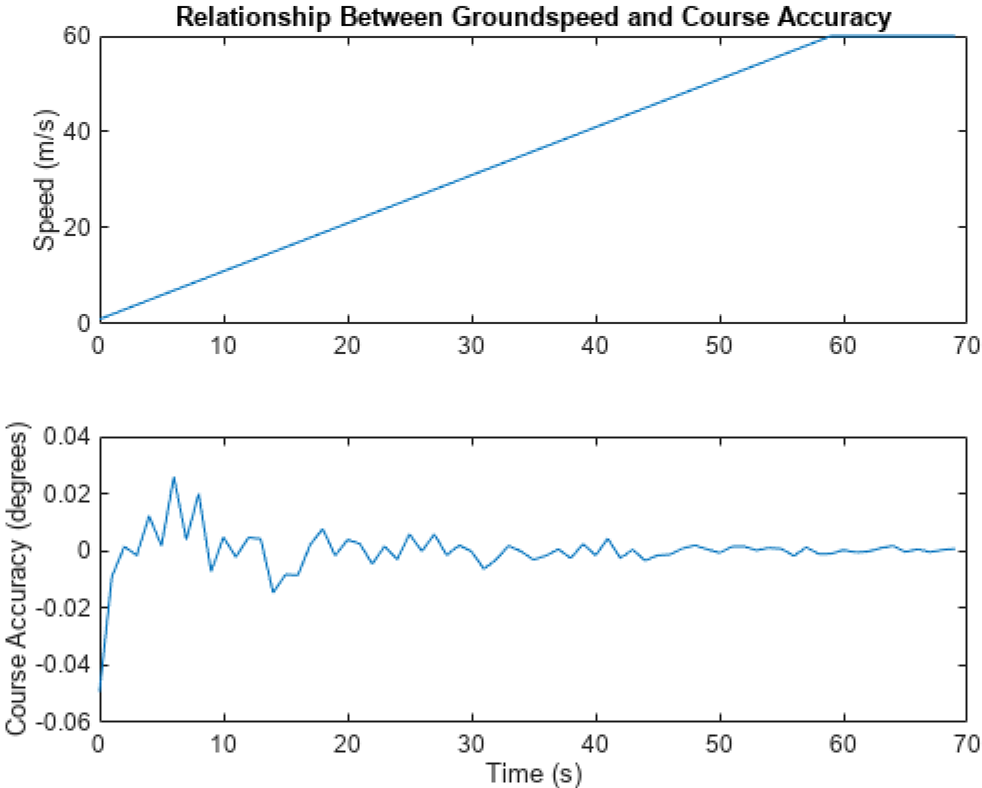
Model GPS measurement data by calling the `GPS` object with your velocity and position matrices.

```
[~,~,groundspeedMeasurement,courseMeasurement] = GPS(NEDposition,NEDvelocity);
```

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

As groundspeed increases, the accuracy of the course increases. Note that the velocity increase during the last ten seconds has no effect, because the additional velocity is not in the ground plane.

```
t = (0:numSamples-1)/GPS.SampleRate;

subplot(2,1,1)
plot(t,groundspeed);
ylabel('Speed (m/s)')
title('Relationship Between Groundspeed and Course Accuracy')

subplot(2,1,2)
courseAccuracy = courseMeasurement - course;
plot(t,courseAccuracy)
xlabel('Time (s)');
ylabel('Course Accuracy (degrees)')
```



Model GPS Receiver Data

Simulate GPS data received during a trajectory from the city of Natick, MA, to Boston, MA.

Define the decimal degree latitude and longitude for the city of Natick, MA USA, and Boston, MA USA. For simplicity, set the altitude for both locations to zero.

```
NatickLLA = [42.27752809999999, -71.34680909999997, 0];
BostonLLA = [42.3600825, -71.05888010000001, 0];
```

Define a motion that can take a platform from Natick to Boston in 20 minutes. Set the origin of the local NED coordinate system as Natick. Create a `waypointTrajectory` object to output the trajectory 10 samples at a time.

```
fs = 1;
duration = 60*20;

bearing = 68; % degrees
distance = 25.39e3; % meters
distanceEast = distance*sind(bearing);
distanceNorth = distance*cosd(bearing);

NatickNED = [0,0,0];
BostonNED = [distanceNorth,distanceEast,0];
```

```
trajectory = waypointTrajectory( ...
    'Waypoints', [NatickNED;BostonNED], ...
    'TimeOfArrival',[0;duration], ...
    'SamplesPerFrame',10, ...
    'SampleRate',fs);
```

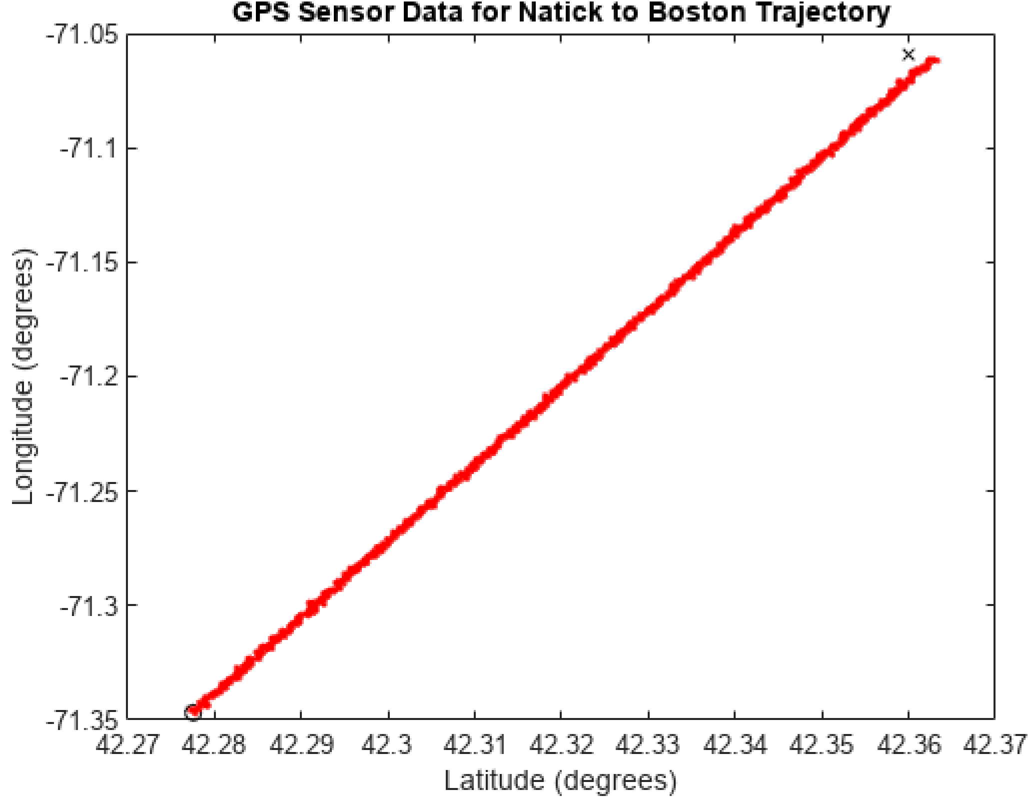
Create a `gpsSensor` object to model receiving GPS data for the platform. Set the `HorizontalPositionalAccuracy` to `25` and the `DecayFactor` to `0.25` to emphasize the noise. Set the `ReferenceLocation` to the Natick coordinates in LLA.

```
GPS = gpsSensor( ...
    'HorizontalPositionAccuracy',25, ...
    'DecayFactor',0.25, ...
    'SampleRate',fs, ...
    'ReferenceLocation',NatickLLA);
```

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

In a loop, call the `gpsSensor` object with the ground-truth trajectory to simulate the received GPS data. Plot the ground-truth trajectory and the model of received GPS data.

```
figure(1)
plot(NatickLLA(1),NatickLLA(2),'ko', ...
     BostonLLA(1),BostonLLA(2),'kx')
xlabel('Latitude (degrees)')
ylabel('Longitude (degrees)')
title('GPS Sensor Data for Natick to Boston Trajectory')
hold on

while ~isDone(trajectory)
    [truePositionNED,~,trueVelocityNED] = trajectory();
    reportedPositionLLA = GPS(truePositionNED,trueVelocityNED);

    figure(1)
    plot(reportedPositionLLA(:,1),reportedPositionLLA(:,2),'r.')
end
```



As a best practice, release System objects when complete.

```
release(GPS)
release(trajectory)
```

Version History

Introduced in R2020b

Extended Capabilities

C/C++ Code Generation

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

Usage notes and limitations:

See "System Objects in MATLAB Code Generation" (MATLAB Coder).

See Also

Objects

insSensor | uavSensor

insSensor

Inertial navigation system and GNSS/GPS simulation model

Description

The `insSensor` System object models a device that fuses measurements from an inertial navigation system (INS) and global navigation satellite system (GNSS) such as a GPS, and outputs the fused measurements.

To output fused INS and GNSS measurements:

1. Create the `insSensor` object and set its properties.
2. Call the object with arguments, as if it were a function.

To learn more about how System objects work, see What Are System Objects?

Creation

Syntax

```
INS = insSensor
INS = insSensor(Name,Value)
```

Description

`INS = insSensor` returns a System object, `INS`, that models a device that outputs measurements from an INS and GNSS.

`INS = insSensor(Name,Value)` sets properties on page 1-45 using one or more name-value pairs. Unspecified properties have default values. Enclose each property name in quotes.

Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see System Design in MATLAB Using System Objects.

`MountingLocation` — Location of sensor on platform (m)
`[0 0 0]` (default) | three-element real-valued vector of form [*x y z*]

Location of the sensor on the platform, in meters, specified as a three-element real-valued vector of the form [*x y z*]. The vector defines the offset of the sensor origin from the origin of the platform.

Tunable: Yes

Data Types: `single` | `double`

`RollAccuracy` — Accuracy of roll measurement (deg)
`0.2` (default) | nonnegative real scalar

Accuracy of the roll measurement of the sensor body, in degrees, specified as a nonnegative real scalar.

Roll is the rotation around the *x*-axis of the sensor body. Roll noise is modeled as a white noise process. `RollAccuracy` sets the standard deviation of the roll measurement noise.

Tunable: Yes

Data Types: `single` | `double`

`PitchAccuracy` — Accuracy of pitch measurement (deg)
`0.2` (default) | nonnegative real scalar

Accuracy of the pitch measurement of the sensor body, in degrees, specified as a nonnegative real scalar.

Pitch is the rotation around the *y*-axis of the sensor body. Pitch noise is modeled as a white noise process. `PitchAccuracy` defines the standard deviation of the pitch measurement noise.

Tunable: Yes

Data Types: `single` | `double`

`YawAccuracy` — Accuracy of yaw measurement (deg)
`1` (default) | nonnegative real scalar

Accuracy of the yaw measurement of the sensor body, in degrees, specified as a nonnegative real scalar.

Yaw is the rotation around the *z*-axis of the sensor body. Yaw noise is modeled as a white noise process. `YawAccuracy` defines the standard deviation of the yaw measurement noise.

Tunable: Yes

Data Types: `single` | `double`

`PositionAccuracy` — Accuracy of position measurement (m)
`[1 1 1]` (default) | nonnegative real scalar | three-element real-valued vector

Accuracy of the position measurement of the sensor body, in meters, specified as a nonnegative real scalar or a three-element real-valued vector. The elements of the vector set the accuracy of the *x*-, *y*-, and *z*-position measurements, respectively. If you specify `PositionAccuracy` as a scalar value, then the object sets the accuracy of all three positions to this value.

Position noise is modeled as a white noise process. `PositionAccuracy` defines the standard deviation of the position measurement noise.

Tunable: Yes

Data Types: `single` | `double`

`VelocityAccuracy` — Accuracy of velocity measurement (m/s)
`0.05` (default) | nonnegative real scalar

Accuracy of the velocity measurement of the sensor body, in meters per second, specified as a nonnegative real scalar.

Velocity noise is modeled as a white noise process. `VelocityAccuracy` defines the standard deviation of the velocity measurement noise.

Tunable: Yes

Data Types: `single` | `double`

`AccelerationAccuracy` — Accuracy of acceleration measurement (m/s^2)

`0` (default) | nonnegative real scalar

Accuracy of the acceleration measurement of the sensor body, in meters per second, specified as a nonnegative real scalar.

Acceleration noise is modeled as a white noise process. `AccelerationAccuracy` defines the standard deviation of the acceleration measurement noise.

Tunable: Yes

Data Types: `single` | `double`

`AngularVelocityAccuracy` — Accuracy of angular velocity measurement (deg/s)

`0` (default) | nonnegative real scalar

Accuracy of the angular velocity measurement of the sensor body, in meters per second, specified as a nonnegative real scalar.

Angular velocity is modeled as a white noise process. `AngularVelocityAccuracy` defines the standard deviation of the acceleration measurement noise.

Tunable: Yes

Data Types: `single` | `double`

`TimeInput` — Enable input of simulation time

`false` or `0` (default) | `true` or `1`

Enable input of simulation time, specified as a logical `0` (`false`) or `1` (`true`). Set this property to `true` to input the simulation time by using the `simTime` argument.

Tunable: No

Data Types: `logical`

`HasGNSSFix` — Enable GNSS fix

`true` or `1` (default) | `false` or `0`

Enable GNSS fix, specified as a logical `1` (`true`) or `0` (`false`). Set this property to `false` to simulate the loss of a GNSS receiver fix. When a GNSS receiver fix is lost, position measurements drift at a rate specified by the `PositionErrorFactor` property.

Tunable: Yes

Dependencies

To enable this property, set `TimeInput` to `true`.

Data Types: `logical`

`PositionErrorFactor` — Position error factor without GNSS fix

`[0 0 0]` (default) | nonnegative scalar | 1-by-3 vector of scalars

Position error factor without GNSS fix, specified as a scalar or a 1-by-3 vector of scalars.

When the `HasGNSSFix` property is set to `false`, the position error grows at a quadratic rate due to constant bias in the accelerometer. The position error for a position component $E(t)$ can be expressed as $E(t) = 1/2\alpha t^2$, where α is the position error factor for the corresponding component and t is the time since the GNSS fix is lost. While running, the object computes t based on the `simTime` input. The computed $E(t)$ values for the x, y, and z components are added to the corresponding position components of the `gTruth` input.

Tunable: Yes

Dependencies

To enable this property, set `TimeInput` to `true` and `HasGNSSFix` to `false`.

Data Types: `single` | `double`

`RandomStream` — Random number source

`'Global stream'` (default) | `'mt19937ar with seed'`

Random number source, specified as one of these options:

- `'Global stream'` –– Generate random numbers using the current global random number stream.
- `'mt19937ar with seed'` –– Generate random numbers using the mt19937ar algorithm, with the seed specified by the `Seed` property.

Data Types: `char` | `string`

`Seed` — Initial seed

`67` (default) | nonnegative integer

Initial seed of the mt19937ar random number generator algorithm, specified as a nonnegative integer.

Dependencies

To enable this property, set `RandomStream` to `'mt19937ar with seed'`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Usage

Syntax

```
measurement = INS(gTruth)
measurement = INS(gTruth,simTime)
```

Description

`measurement = INS(gTruth)` models the data received from an INS sensor reading and GNSS sensor reading. The output measurement is based on the inertial ground-truth state of the sensor body, `gTruth`.

`measurement = INS(gTruth,simTime)` additionally specifies the time of simulation, `simTime`. To enable this syntax, set the `TimeInput` property to `true`.

Input Arguments

`gTruth` — Inertial ground-truth state of sensor body
structure

Inertial ground-truth state of sensor body, in local Cartesian coordinates, specified as a structure containing these fields:

Field	Description
`'Position'`	Position, in meters, specified as a real, finite *N*-by-3 matrix of [*x y z*] vectors. *N* is the number of samples in the current frame.
`'Velocity'`	Velocity (*v*), in meters per second, specified as a real, finite *N*-by-3 matrix of [v_x v_y v_z] vector. *N* is the number of samples in the current frame.
`'Orientation'`	Orientation with respect to the local Cartesian coordinate system, specified as one of these options: • *N*-element column vector of `quaternion` objects • 3-by-3-by-*N* array of rotation matrices • *N*-by-3 matrix of [x_{roll} y_{pitch} z_{yaw}] angles in degrees Each quaternion or rotation matrix is a frame rotation from the local Cartesian coordinate system to the current sensor body coordinate system. *N* is the number of samples in the current frame.
`'Acceleration'`	Acceleration (*a*), in meters per second squared, specified as a real, finite *N*-by-3 matrix of [a_x a_y a_z] vectors. *N* is the number of samples in the current frame.
`'AngularVelocity'`	Angular velocity (*ω*), in degrees per second squared, specified as a real, finite *N*-by-3 matrix of [ω_x ω_y ω_z] vectors. *N* is the number of samples in the current frame.

The field values must be of type `double` or `single`.

The `Position`, `Velocity`, and `Orientation` fields are required. The other fields are optional.

Example: `struct('Position',[0 0 0],'Velocity',[0 0 0],'Orientation',quaternion([1 0 0 0]))`

`simTime` — Simulation time
nonnegative real scalar

Simulation time, in seconds, specified as a nonnegative real scalar.

Data Types: `single` | `double`

Output Arguments

`measurement` — Measurement of sensor body motion
structure

Measurement of the sensor body motion, in local Cartesian coordinates, returned as a structure containing these fields:

Field	Description
`'Position'`	Position, in meters, specified as a real, finite *N*-by-3 matrix of [*x y z*] vectors. *N* is the number of samples in the current frame.
`'Velocity'`	Velocity (v), in meters per second, specified as a real, finite *N*-by-3 matrix of [v_x v_y v_z] vector. *N* is the number of samples in the current frame.
`'Orientation'`	Orientation with respect to the local Cartesian coordinate system, specified as one of these options: • *N*-element column vector of `quaternion` objects • 3-by-3-by-*N* array of rotation matrices • *N*-by-3 matrix of [x_{roll} y_{pitch} z_{yaw}] angles in degrees Each quaternion or rotation matrix is a frame rotation from the local Cartesian coordinate system to the current sensor body coordinate system. *N* is the number of samples in the current frame.
`'Acceleration'`	Acceleration (a), in meters per second squared, specified as a real, finite *N*-by-3 matrix of [a_x a_y a_z] vectors. *N* is the number of samples in the current frame.
`'AngularVelocity'`	Angular velocity (ω), in degrees per second squared, specified as a real, finite *N*-by-3 matrix of [ω_x ω_y ω_z] vectors. *N* is the number of samples in the current frame.

The returned field values are of type `double` or `single` and are of the same type as the corresponding field values in the `gTruth` input.

Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

Specific to `insSensor`

perturbations	Perturbation defined on object
perturb	Apply perturbations to object

Common to All System Objects

step	Run System object algorithm
clone	Create duplicate System object
isLocked	Determine if System object is in use
reset	Reset internal states of System object
release	Release resources and allow changes to System object property values and input characteristics

Examples

Generate INS Measurements from Stationary Input

Create a motion structure that defines a stationary position at the local north-east-down (NED) origin. Because the platform is stationary, you need to define only a single sample. Assume the ground-truth motion is sampled for 10 seconds with a 100 Hz sample rate. Create a default `insSensor` System object™. Preallocate variables to hold output from the `insSensor` object.

```
Fs = 100;
duration = 10;
numSamples = Fs*duration;

motion = struct( ...
    'Position',zeros(1,3), ...
    'Velocity',zeros(1,3), ...
    'Orientation',ones(1,1,'quaternion'));

INS = insSensor;

positionMeasurements = zeros(numSamples,3);
velocityMeasurements = zeros(numSamples,3);
orientationMeasurements = zeros(numSamples,1,'quaternion');
```

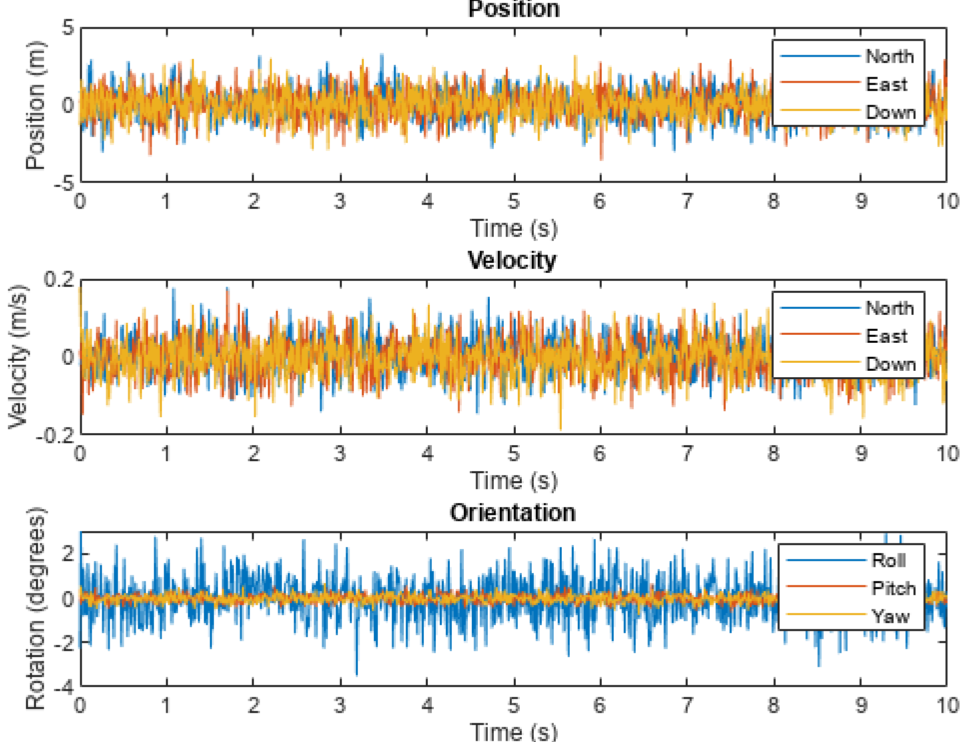
In a loop, call `INS` with the stationary motion structure to return the position, velocity, and orientation measurements in the local NED coordinate system. Log the position, velocity, and orientation measurements.

```
for i = 1:numSamples

    measurements = INS(motion);

    positionMeasurements(i,:) = measurements.Position;
    velocityMeasurements(i,:) = measurements.Velocity;
```

```
    orientationMeasurements(i) = measurements.Orientation;
end
```

Convert the orientation from quaternions to Euler angles for visualization purposes. Plot the position, velocity, and orientation measurements over time.

```
orientationMeasurements = eulerd(orientationMeasurements,'ZYX','frame');

t = (0:(numSamples-1))/Fs;

subplot(3,1,1)
plot(t,positionMeasurements)
title('Position')
xlabel('Time (s)')
ylabel('Position (m)')
legend('North','East','Down')

subplot(3,1,2)
plot(t,velocityMeasurements)
title('Velocity')
xlabel('Time (s)')
ylabel('Velocity (m/s)')
legend('North','East','Down')

subplot(3,1,3)
plot(t,orientationMeasurements)
title('Orientation')
xlabel('Time (s)')
ylabel('Rotation (degrees)')
legend('Roll', 'Pitch', 'Yaw')
```



Generate INS Measurements for a Turning Platform

Generate INS measurements using the `insSensor` System object™. Use `waypointTrajectory` to generate the ground-truth path.

Specify a ground-truth orientation that begins with the sensor body *x*-axis aligned with North and ends with the sensor body *x*-axis aligned with East. Specify waypoints for an arc trajectory and a time-of-arrival vector for the corresponding waypoints. Use a 100 Hz sample rate. Create a `waypointTrajectory` System object with the waypoint constraints, and set `SamplesPerFrame` so that the entire trajectory is output with one call.

```
eulerAngles = [0,0,0; ...
               0,0,0; ...
               90,0,0; ...
               90,0,0];
orientation = quaternion(eulerAngles,'eulerd','ZYX','frame');

r = 20;
waypoints = [0,0,0; ...
             100,0,0; ...
             100+r,r,0; ...
             100+r,100+r,0];

toa = [0,10,10+(2*pi*r/4),20+(2*pi*r/4)];
```

```
Fs = 100;
numSamples = floor(Fs*toa(end));

path = waypointTrajectory('Waypoints',waypoints, ...
    'TimeOfArrival',toa, ...
    'Orientation',orientation, ...
    'SampleRate',Fs, ...
    'SamplesPerFrame',numSamples);
```

Create an `insSensor` System object to model receiving INS data. Set the `PositionAccuracy` to `0.1`.

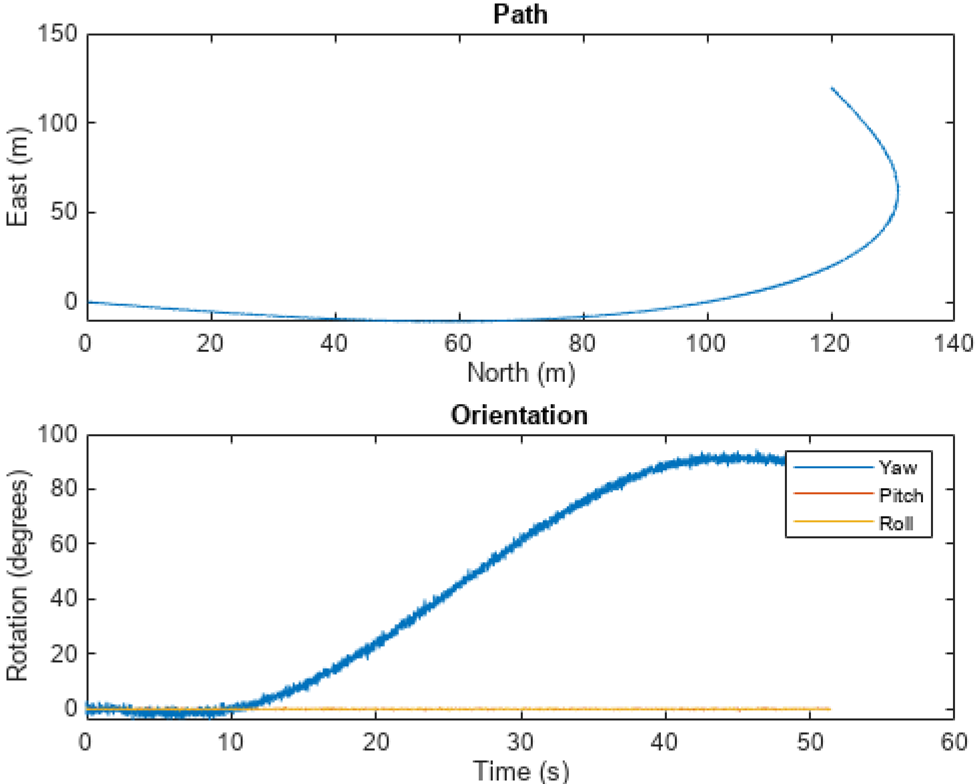
```
ins = insSensor('PositionAccuracy',0.1);
```

Call the waypoint trajectory object, `path`, to generate the ground-truth motion. Call the INS simulator, `ins`, with the ground-truth motion to generate INS measurements.

```
[motion.Position,motion.Orientation,motion.Velocity] = path();
insMeas = ins(motion);
```

Convert the orientation returned by `ins` to Euler angles in degrees for visualization purposes. Plot the full path and orientation over time.

```
orientationMeasurementEuler = eulerd(insMeas.Orientation,'ZYX','frame');

subplot(2,1,1)
plot(insMeas.Position(:,1),insMeas.Position(:,2));
title('Path')
xlabel('North (m)')
ylabel('East (m)')

subplot(2,1,2)
t = (0:(numSamples-1)).'/Fs;
plot(t,orientationMeasurementEuler(:,1), ...
     t,orientationMeasurementEuler(:,2), ...
     t,orientationMeasurementEuler(:,3));
title('Orientation')
legend('Yaw','Pitch','Roll')
xlabel('Time (s)')
ylabel('Rotation (degrees)')
```



Version History

Introduced in R2020b

Extended Capabilities

C/C++ Code Generation

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

The object functions, `perturbations` and `perturb`, do not support code generation.

Usage notes and limitations:

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

See Also

Objects

`gpsSensor` | `uavSensor`

Objects

mavlinkdialect

Parse and store MAVLink dialect XML

Description

The `mavlinkdialect` object parses and stores message and enum definitions extracted from a MAVLink message definition file (`.xml`). The message definition files define the messages supported for this specific dialect. The structure of the message definitions is defined by the MAVLink message protocol.

Creation

Syntax

```
dialect = mavlinkdialect("common.xml")
dialect = mavlinkdialect(dialectXML)
dialect = mavlinkdialect(dialectXML,version)
dialect = mavlinkdialect( ___ ,SigningChannel=channel)
```

Description

`dialect = mavlinkdialect("common.xml")` creates a MAVLink dialect using the `common.xml` file for standard MAVLink messages.

`dialect = mavlinkdialect(dialectXML)` specifies the XML file for parsing the message definitions. The input sets the `DialectXML` property.

`dialect = mavlinkdialect(dialectXML,version)` additionally specifies the MAVLink protocol version. The inputs set the `DialectXML` and `Version` properties, respectively.

`dialect = mavlinkdialect( ___ ,SigningChannel=channel)` uses the MAVLink signing channel `channel`.

Input Arguments

`channel` — MAVLink signing channel
structure

MAVLink signing channel, specified as a structure containing these fields:

- `Stream` — Signing stream, specified as a `mavlinksigning` object.
- `SystemID` — System ID, specified by an integer in the range [0, 255].
- `ComponentID` — Component ID, specified by an integer in the range [0, 255].
- `LinkID` — Link ID, specified by an integer in the range [0, 255].
- `Key` — Key, specified by as a string.
- `Timestamp` — Time passed since `2015-01-01 UTC`. Unit is 10 microseconds.

- `CreationTime` — Time of creation, specified as a `datetime` array.

You can create this structure by adding the signing channel using the `addChannel` object function of the `mavlinksigning` object.

Properties

`DialectXML` — MAVLink dialect name

string

MAVLink dialect name, specified as a string. This name is based on the XML file name.

Example: `"ardupilotmega"`

Data Types: `char` | `string`

`Version` — MAVLink protocol version

2 (default) | 1

MAVLink protocol version, specified as either 1 or 2.

Data Types: `double`

Object Functions

createcmd	Create MAVLink command message
createmsg	Create MAVLink message
deserializemsg	Deserialize MAVLink message from binary buffer
msginfo	Message definition for message ID
enuminfo	Enum definition for enum ID
enum2num	Enum value for given entry
num2enum	Enum entry for given value

Examples

Parse and Use MAVLink Dialect

This example shows how to parse a MAVLink XML file and create messages and commands from the definitions.

NOTE: This example requires you to install the UAV Library for Robotics System Toolbox®. Call `roboticsAddons` to open the Add-ons Explorer and install the library.

Parse and store the MAVLink dialect XML. Specify the XML path. The default `"common.xml"` dialect is provided. This XML file contains all the message and enum definitions.

```
dialect = mavlinkdialect("common.xml");
```

Create a MAVLink command from the `MAV_CMD` enum, which is an enum of MAVLink commands to send to the UAV. Specify the setting as `"int"` or `"long"`, and the type as an integer or string.

```
cmdMsg = createcmd(dialect,"long",22)
```

```
cmdMsg = struct with fields:
      MsgID: 76
```

```
    Payload: [1x1 struct]
```

Verify the command name using `num2enum`. Command 22 is a take-off command for the UAV. You can convert back to an ID using `enum2num`. Your dialect can contain many different enums with different names and IDs.

```
cmdName = num2enum(dialect,"MAV_CMD",22)
```

```
cmdName = 
"MAV_CMD_NAV_TAKEOFF"
```

```
cmdID = enum2num(dialect,"MAV_CMD",cmdName)
```

```
cmdID = 22
```

Use `enuminfo` to view the table of the `MAV_CMD` enum entries.

```
info = enuminfo(dialect,"MAV_CMD");
info.Entries{:}
```

```
ans=148×3 table
                    Name                    Value
    ____________________________________    _____    ______________________________________________

    "MAV_CMD_NAV_WAYPOINT"                    16     "Navigate to waypoint."
    "MAV_CMD_NAV_LOITER_UNLIM"                17     "Loiter around this waypoint an unlimited a
    "MAV_CMD_NAV_LOITER_TURNS"                18     "Loiter around this waypoint for X turns"
    "MAV_CMD_NAV_LOITER_TIME"                 19     "Loiter at the specified latitude, longitud
    "MAV_CMD_NAV_RETURN_TO_LAUNCH"            20     "Return to launch location"
    "MAV_CMD_NAV_LAND"                        21     "Land at location."
    "MAV_CMD_NAV_TAKEOFF"                     22     "Takeoff from ground / hand. Vehicles that
    "MAV_CMD_NAV_LAND_LOCAL"                  23     "Land at local position (local frame only)"
    "MAV_CMD_NAV_TAKEOFF_LOCAL"               24     "Takeoff from local position (local frame o
    "MAV_CMD_NAV_FOLLOW"                      25     "Vehicle following, i.e. this waypoint repr
    "MAV_CMD_NAV_CONTINUE_AND_CHANGE_ALT"     30     "Continue on the current course and climb/d
    "MAV_CMD_NAV_LOITER_TO_ALT"               31     "Begin loiter at the specified Latitude and
    "MAV_CMD_DO_FOLLOW"                       32     "Begin following a target"
    "MAV_CMD_DO_FOLLOW_REPOSITION"            33     "Reposition the MAV after a follow target c
    "MAV_CMD_DO_ORBIT"                        34     "Start orbiting on the circumference of a c
    "MAV_CMD_NAV_ROI"                         80     "Sets the region of interest (ROI) for a se
      ⋮
```

Query the dialect for a specific message ID. Create a blank MAVLink message using the message ID.

```
info = msginfo(dialect,"HEARTBEAT")
```

```
info=1×4 table
    MessageID    MessageName
    _________    ___________    ________________________________________________________________

        0        "HEARTBEAT"    "The heartbeat message shows that a system or component is presen
```

```
msg = createmsg(dialect,info.MessageID);
```

Version History

Introduced in R2019a

See Also

`mavlinkio` | `mavlinkclient` | `mavlinksub` | `mavlinksigning`

Topics

"Tune UAV Parameters Using MAVLink Parameter Protocol"

External Websites

MAVLink Developer Guide

mavlinkclient

MAVLink client information

Description

The `mavlinkclient` object stores MAVLink client information for connecting to UAVs (unmanned aerial vehicles) that utilize the MAVLink communication protocol. Connect with a MAVLink client using `mavlinkio` and use this object for saving the component and system information.

Creation

Syntax

`client = mavlinkclient(mavlink,sysID,compID)`

Description

`client = mavlinkclient(mavlink,sysID,compID)` creates a MAVLink client interface for a MAVLink component. Connect to a MAVLink client using `mavlinkio` and specify the object in `mavlink`. When a heartbeat is received by the client, the `ComponentType` and `AutoPilotType` properties are updated automatically. Specify the `SystemID` and `ComponentID` as integers.

Properties

`SystemID` — MAVLink system ID
positive integer between 1 and 255

MAVLink system ID, specified as a positive integer between 1 and 255. MAVLink protocol only supports up to 255 systems. Usually, each UAV has its own system ID, but multiple UAVs could be considered one system.

Example: `1`

Data Types: `uint8`

`ComponentID` — MAVLink component ID
positive integer between 1 and 255

MAVLink component ID, specified as a positive integer between 1 and 255.

Example: `2`

Data Types: `uint8`

`ComponentType` — MAVLink component type
`"Unknown"` (default) | string

MAVLink component type, specified as a string. This value is automatically updated to the correct type if a heartbeat message is received by the client with the matching system ID and component ID. You must be connected to a client using `mavlinkio`.

Example: "MAV_TYPE_GCS"

Data Types: string

AutoPilot — Autopilot type for UAV
"Unknown" (default) | string

Autopilot type for UAV, specified as a string. This value is automatically updated to the correct type if a heartbeat message is received by the client with the matching system ID and component ID. You must be connected to a client using mavlinkio.

Example: "MAV_AUTOPILOT_INVALID"

Data Types: string

Examples

Store MAVLink Client Information

Connect to a MAVLink client.

```
mavlink = mavlinkio("common.xml");
connect(mavlink,"UDP");
```

Create the object for storing the client information. Specify the system and component ID.

```
client = mavlinkclient(mavlink,1,1)
```

```
client =
  mavlinkclient with properties:

         SystemID: 1
      ComponentID: 1
    ComponentType: "Unknown"
    AutopilotType: "Unknown"
```

Disconnect from client.

```
disconnect(mavlink)
```

Version History

Introduced in R2019a

See Also

mavlinkio | mavlinkdialect | mavlinksub

Topics
"Tune UAV Parameters Using MAVLink Parameter Protocol"

External Websites
MAVLink Developer Guide

mavlinkio

Connect with MAVLink clients to exchange messages

Description

The `mavlinkio` object connects with MAVLink clients through UDP ports to exchange messages with UAVs (unmanned aerial vehicles) using the MAVLink communication protocols.

Creation

Syntax

```
mavlink = mavlinkio(msgDefinitions)
mavlink = mavlinkio(dialectXML)
mavlink = mavlinkio(dialectXML,version)
mavlink = mavlinkio( ___ ,Name,Value)
```

Description

`mavlink = mavlinkio(msgDefinitions)` creates an interface to connect with MAVLink clients using the input `mavlinkdialect` object, which defines the message definitions. This dialect object is set directly to the `Dialect` property.

`mavlink = mavlinkio(dialectXML)` directly specifies the XML file for the message definitions as a file name. A `mavlinkdialect` is created using this XML file and set to the `Dialect` property

`mavlink = mavlinkio(dialectXML,version)` additionally specifies the MAVLink protocol version as either `1` or `2`.

`mavlink = mavlinkio( ___ ,Name,Value)` additionally specifies arguments using the following name-value pairs.

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

The name-value pairs directly set the MAVLink client information in the `LocalClient` property. See LocalClient for more info on what values can be set.

Properties

`Dialect` — MAVLink dialect
`mavlinkdialect` object

MAVLink dialect, specified as a `mavlinkdialect` object. The dialect specifies the message structure for the MAVLink protocol.

`LocalClient` — Local client information
structure

This property is read-only.

Local client information, specified as a structure. The local client is setup in MATLAB® to communicate with other MAVLink clients. The structure contains the following fields:

- `SystemID`
- `ComponentID`
- `ComponentType`
- `AutopilotType`

To set these values when creating the `mavlinkio` object, use name-value pairs. For example:

```
mavlink = mavlinkio("common.xml","SystemID",1,"ComponentID",1)
```

This property is nontunable when you are connected to a MAVLink client. For more information, see `mavlinkclient`.

Data Types: `struct`

Object Functions

connect	Connect to MAVLink clients through UDP port
disconnect	Disconnect from MAVLink clients
sendmsg	Send MAVLink message
sendudpmsg	Send MAVLink message to UDP port
serializemsg	Serialize MAVLink message to binary buffer
listConnections	List all active MAVLink connections
listClients	List all connected MAVLink clients
listTopics	List all topics received by MAVLink client

Examples

Store MAVLink Client Information

Connect to a MAVLink client.

```
mavlink = mavlinkio("common.xml");
connect(mavlink,"UDP");
```

Create the object for storing the client information. Specify the system and component ID.

```
client = mavlinkclient(mavlink,1,1)
```

```
client = 
  mavlinkclient with properties:

         SystemID: 1
      ComponentID: 1
    ComponentType: "Unknown"
    AutopilotType: "Unknown"
```

Disconnect from client.

```
disconnect(mavlink)
```

Work with MAVLink Connection

This example shows how to connect to MAVLink clients, inspect the list of topics, connections, and clients, and send messages through UDP ports using the MAVLink communication protocol.

Connect to a MAVLink client using the `"common.xml"` dialect. This local client communicates with any other clients through a UDP port.

```
dialect = mavlinkdialect("common.xml");
mavlink = mavlinkio(dialect);
connect(mavlink,"UDP")
```

```
ans = 
"Connection1"
```

You can list all the active clients, connections, and topics for the MAVLink connection. Currently, there is only one client connection and no topics have received messages.

```
listClients(mavlink)
```

```
ans=1×4 table
    SystemID    ComponentID    ComponentType          AutopilotType      
    ________    ___________    ______________    ________________________

      255            1         "MAV_TYPE_GCS"    "MAV_AUTOPILOT_INVALID"
```

```
listConnections(mavlink)
```

```
ans=1×2 table
    ConnectionName       ConnectionInfo    
    ______________    ___________________

    "Connection1"     "UDP@0.0.0.0:53894"
```

```
listTopics(mavlink)
```

```
ans =

  0x5 empty table
```

Create a subscriber for receiving messages on the client. This subscriber listens for the `"HEARTBEAT"` message topic with ID equal to `0`.

```
sub = mavlinksub(mavlink,0);
```

Create a `"HEARTBEAT"` message using the `mavlinkdialect` object. Specify payload information and send the message over the MAVLink client.

```
msg = createmsg(dialect,"HEARTBEAT");
msg.Payload.type(:) = enum2num(dialect,'MAV_TYPE','MAV_TYPE_QUADROTOR');
sendmsg(mavlink,msg)
```

Disconnect from the client.

```
disconnect(mavlink)
```

Version History

Introduced in R2019a

See Also

`connect` | `mavlinkdialect` | `mavlinkclient` | `mavlinksub`

Topics

"Tune UAV Parameters Using MAVLink Parameter Protocol"

External Websites

MAVLink Developer Guide

mavlinksub

Receive MAVLink messages

Description

The `mavlinksub` object subscribes to topics from the connected MAVLink clients using a `mavlinkio` object. Use the `mavlinksub` object to obtain the most recently received messages and call functions to process newly received messages.

Creation

Syntax

```
sub = mavlinksub(mavlink)
sub = mavlinksub(mavlink,topic)
sub = mavlinksub(mavlink,client)
sub = mavlinksub(mavlink,client,topic)
sub = mavlinksub( ___ ,Name,Value)
```

Description

`sub = mavlinksub(mavlink)` subscribes to all topics from all clients connected via the `mavlinkio` object. This syntax sets the `Client` property to `"Any"`.

`sub = mavlinksub(mavlink,topic)` subscribes to a specific topic, specified as a string or integer, from all clients connected via the `mavlinkio` object. The function sets the `topic` input to the `Topic` property.

`sub = mavlinksub(mavlink,client)` subscribes to all topics from the client specified as a `mavlinkclient` object. The function sets the `Client` property to this input client.

`sub = mavlinksub(mavlink,client,topic)` subscribes to a specific topic on a specific client. The function sets the `Client` and `Topic` properties.

`sub = mavlinksub( ___ ,Name,Value)` additionally specifies the `BufferSize` or `NewMessageFcn` properties using name-value pairs and the previous syntaxes. The `Name` input is one of the property names.

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Properties

`Client` — Client information of received message

`"Any"` (default) | `mavlinkclient` object

Client information of the received message, specified as a `mavlinkclient` object. The default value of `"Any"` means the subscriber is listening to all clients connected via the `mavlinkio` object.

`Topic` — Topic name
`"Any"` (default) | string

Topic name the subscriber listens to, specified as a string. The default value of `"Any"` means the subscriber is listening to all topics on the client.

Example: `"HEARTBEAT"`

Data Types: `char` | `string`

`BufferSize` — Length of message buffer
`1` (default) | positive integer

Length of message buffer, specified as a positive integer. This value is the maximum number of messages that can be stored in this subscriber.

Data Types: `double`

`NewMessageFcn` — Callback function for new messages
`[]` (default) | function handle

Callback function for new messages, specified as a function handle. This function is called when a new message is received by the client. The function handle has the following syntax:

```
callback(sub,msg)
```

`sub` is a structure with fields for the `Client`, `Topic`, and `BufferSize` properties of the `mavlinksub` object. `msg` is the message received as a structure with the fields:

- `MsgID` –– Positive integer for message ID.
- `SystemID` –– System ID of MAVLink client that sent message.
- `ComponentID`–– Component ID of MAVLink client that sent message.
- `Payload` –– Structure containing fields based on the message definition.
- `Seq` –– Positive integer for sequence of message.

The `Payload` is a structure defined by the message definition for the MAVLink dialect.

Data Types: `function_handle`

Object Functions

latestmsgs   Received messages from MAVLink subscriber

Examples

Subscribe to MAVLink Topic

Connect to a MAVLink client.

```
mavlink = mavlinkio("common.xml")
```

```
mavlink = 
  mavlinkio with properties:
```

```
        Dialect: [1x1 mavlinkdialect]
    LocalClient: [1x1 struct]
```

```
connect(mavlink,"UDP")
```

```
ans = 
"Connection1"
```

Get the client information.

```
client = mavlinkclient(mavlink,1,1);
```

Subscribe to the "HEARTBEAT" topic.

```
heartbeat = mavlinksub(mavlink,client,'HEARTBEAT');
```

Get the latest message. You must wait for a message to be received. Currently, no heartbeat message has been received on the `mavlink` object.

```
latestmsgs(heartbeat,1)
```

```
ans = 

  1x0 empty struct array with fields:

    MsgID
    SystemID
    ComponentID
    Payload
    Seq
```

Disconnect from client.

```
disconnect(mavlink)
```

Version History

Introduced in R2019a

See Also

`latestmsgs` | `mavlinkclient` | `mavlinkio` | `mavlinkdialect`

Topics

"Tune UAV Parameters Using MAVLink Parameter Protocol"

External Websites

MAVLink Developer Guide

mavlinktlog

Read MAVLink message from TLOG file

Description

The `mavlinktlog` object reads all messages from a telemetry log or TLOG file (`.tlog`). The object gives you information about the file, including the start and end time, number of messages, available topics, and packet loss percentage. You can specify a MAVLink dialect for parsing the messages or use the `common.xml` dialect.

Creation

Syntax

```
tlogReader = mavlinktlog(filePath)
tlogReader = mavlinktlog(filePath,dialect)
```

Description

`tlogReader = mavlinktlog(filePath)` reads all messages from the tlog file at the given file path and returns an object summarizing the file. This syntax uses the `common.xml` dialect for the MAVLink protocol (Version 2.0) for parsing the messages. The information in `filePath` is used to set the `FileName` property.

`tlogReader = mavlinktlog(filePath,dialect)` reads the MAVLink messages based on the dialect specified as a `mavlinkdialect` object or string scalar specifying the XML file path. `dialect` sets the `Dialect` property.

Properties

`FileName` — Name of TLOG file
string scalar | character vector

This property is read-only.

Name of the TLOG file, specified as a string scalar or character vector. The name is the last part of the path given in the `filePath` input.

Example: `'flightlog.tlog'`

Data Types: `string` | `char`

`Dialect` — MAVLink dialect
`'common.xml'` (default) | `mavlinkdialect` object

This property is read-only.

MAVLink dialect used for parsing the message data, specified as a `mavlinkdialect` object.

`StartTime` — Time of first message recorded
`datetime` object

This property is read-only.

Time of the first message recorded in the TLOG file, specified as a `datetime` object.

Data Types: `datetime`

`EndTime` — Time of last message recorded
`datetime` object

This property is read-only.

Time of the last message recorded in the TLOG file, specified as a `datetime` object.

Data Types: `datetime`

`NumMessages` — Number of MAVLink messages in TLOG file
numeric scalar

This property is read-only.

Number of MAVLink messages in the TLOG file, specified as a numeric scalar.

Data Types: `double`

`AvailableTopics` — List of different message types
table

This property is read-only.

List of different messages, specified as a table that contains:

- `MessageID`
- `MessageName`
- `SystemID`
- `ComponentID`
- `NumMessages`

Data Types: `table`

`NumPacketsLost` — Percentage of packets lost
numeric scalar from 0 through 100

This property is read-only.

Percentage of packets lost, specified as a numeric scalar from 0 through 100.

Data Types: `double`

Object Functions

readmsg   Read specific messages from TLOG file

Examples

Read Messages from MAVLink TLOG File

Load the TLOG file. Specify the relative path of the file name.

```
tlogReader = mavlinktlog("mavlink_flightlog.tlog")
```

```
tlogReader = 
  mavlinktlog with properties:

           FileName: "mavlink_flightlog"
            Dialect: [1x1 mavlinkdialect]
          StartTime: '2018-05-18 17:53:28.893 (0.000 Seconds)'
            EndTime: '2018-05-18 18:04:52.524 (683.631 Seconds)'
        NumMessages: 20370
    AvailableTopics: [5x5 table]
     NumPacketsLost: 251
```

Read the `"HEARTBEAT"` messages from the file.

```
msgData = readmsg(tlogReader,"MessageName","HEARTBEAT")
```

```
msgData=1×6 table
    MessageID    MessageName    SystemID    ComponentID         Messages          Version
    _________    ___________    ________    ___________    __________________    _______

        0        "HEARTBEAT"       5             1         {3897x6 timetable}       2
```

Version History

Introduced in R2019a

See Also

`readmsg` | `mavlinkdialect` | `mavlinkclient` | `mavlinkio`

Topics

"Visualize and Play Back MAVLink Flight Log"

multirotor

Guidance model for multirotor UAVs

Description

A `multirotor` object represents a reduced-order guidance model for an unmanned aerial vehicle (UAV). The model approximates the behavior of a closed-loop system consisting of an autopilot controller and a multirotor kinematic model for 3-D motion.

For fixed-wing UAVs, see `fixedwing`.

Creation

`model = multirotor` creates a multirotor motion model with `double` precision values for inputs, outputs, and configuration parameters of the guidance model.

`model = multirotor(DataType)` specifies the data type precision (`DataType` property) for the inputs, outputs, and configurations parameters of the guidance model.

Properties

Name — Name of UAV
`"Unnamed"` (default) | string scalar

Name of the UAV, used to differentiate it from other models in the workspace, specified as a string scalar.

Example: `"myUAV1"`

Data Types: `string`

`Configuration` — UAV controller configuration
structure

UAV controller configuration, specified as a structure of parameters. Specify these parameters to tune the internal control behaviour of the UAV. Specify the proportional (P) and derivative (D) gains for the dynamic model and other UAV parameters. For multirotor UAVs, the structure contains these fields with defaults listed:

- `'PDRoll'` - `[3402.97 116.67]`
- `'PDPitch'` - `[3402.97 116.67]`
- `'PYawRate'` - `1950`
- `'PThrust'` - `3900`
- `'Mass'` - `0.1` (measured in kg)

Example: `struct('PDRoll',[3402.97,116.67],'PDPitch',[3402.97, 116.67],'PYawRate',1950,'PThrust',3900,'Mass',0.1)`

Data Types: `struct`

`ModelType` — UAV guidance model type
`'MultirotorGuidance'` (default)

This property is read-only.

UAV guidance model type, specified as `'MultirotorGuidance'`.

`DataType` — Input and output numeric data types
`'double'` (default) | `'single'`

Input and output numeric data types, specified as either `'double'` or `'single'`. Choose the data type based on possible software or hardware limitations. Specify `DataType` when first creating the object.

Object Functions

control	Control commands for UAV
derivative	Time derivative of UAV states
environment	Environmental inputs for UAV
state	UAV state vector

Examples

Simulate A Multirotor Control Command

This example shows how to use the `multirotor` guidance model to simulate the change in state of a UAV due to a command input.

Create the multirotor guidance model.

```
model = multirotor;
```

Create a state structure. Specify the location in world coordinates.

```
s = state(model);
s(1:3) = [3;2;1];
```

Specify a control command, `u`, that specified the roll and thrust of the multirotor.

```
u = control(model);
u.Roll = pi/12;
u.Thrust = 1;
```

Create a default environment without wind.

```
e = environment(model);
```

Compute the time derivative of the state given the current state, control command, and environment.

```
sdot = derivative(model,s,u,e);
```

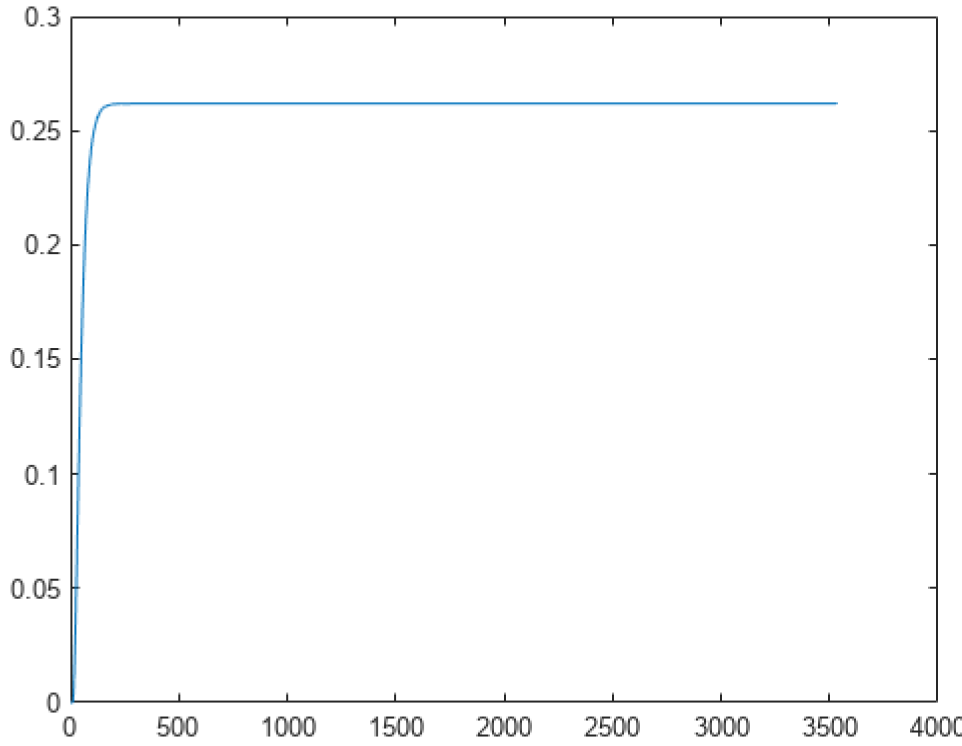
Simulate the UAV state using `ode45` integration. The `y` field outputs the multirotor UAV states as a 13-by-*n* matrix.

```
simOut = ode45(@(~,x)derivative(model,x,u,e), [0 3], s);
size(simOut.y)
```

```
ans = 1×2

          13        3536
```

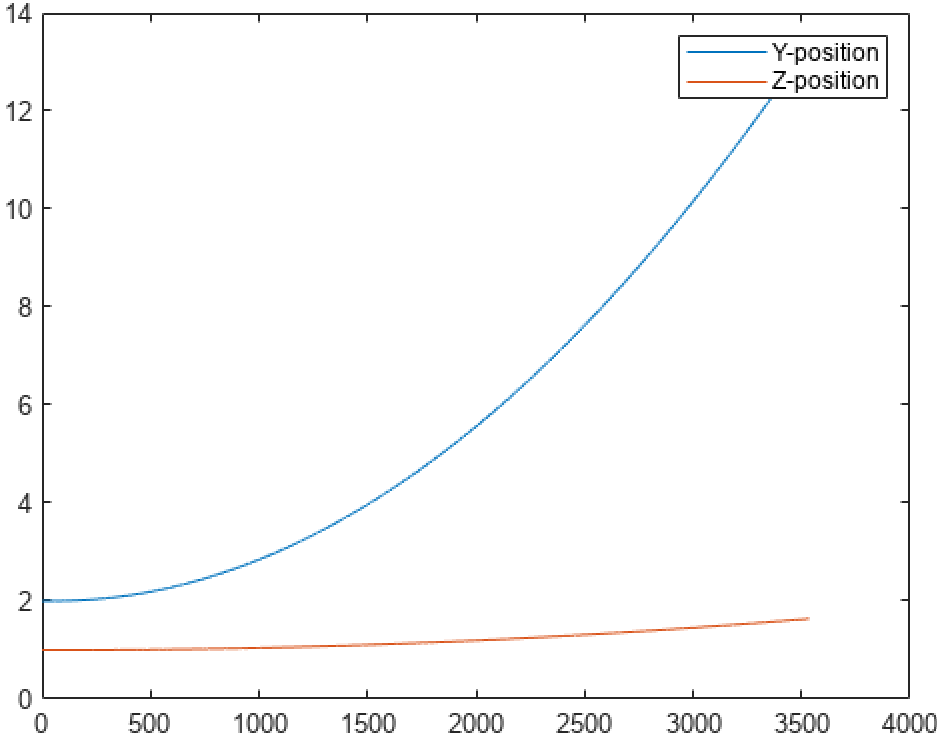
Plot the change in roll angle based on the simulation output. The roll angle (the X Euler angle) is the 9th row of the `simOut.y` output.

```
plot(simOut.y(9,:))
```



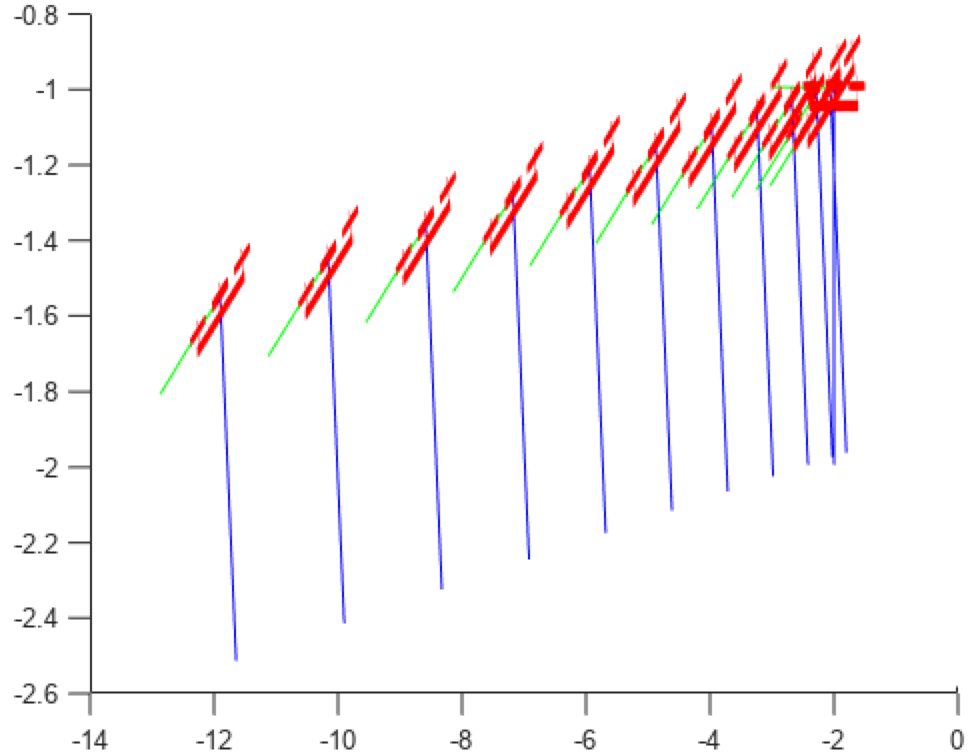
Plot the change in the Y and Z positions. With the specified thrust and roll angle, the multirotor should fly over and lose some altitude. A positive value for Z is expected as positive Z is down.

```
figure
plot(simOut.y(2,:));
hold on
plot(simOut.y(3,:));
legend('Y-position','Z-position')
hold off
```



You can also plot the multirotor trajectory using `plotTransforms`. Create the translation and rotation vectors from the simulated state. Downsample (every 300th element) and transpose the `simOut` elements, and convert the Euler angles to quaternions. Specify the mesh as the `multirotor.stl` file and the positive Z-direction as `"down"`. The displayed view shows the UAV translating in the Y-direction and losing altitude.

```
translations = simOut.y(1:3,1:300:end)'; % xyz position
rotations = eul2quat(simOut.y(7:9,1:300:end)'); % ZYX Euler
plotTransforms(translations,rotations,...
    'MeshFilePath','multirotor.stl','InertialZDirection',"down")
view([90.00 -0.60])
```



More About

UAV Coordinate Systems

The UAV Toolbox uses the North-East-Down (NED) coordinate system convention, which is also sometimes called the local tangent plane (LTP). The UAV position vector consists of three numbers for position along the northern-axis, eastern-axis, and vertical position. The down element complies with the right-hand rule and results in negative values for altitude gain.

The ground plane, or earth frame (NE plane, D = 0), is assumed to be an inertial plane that is flat based on the operation region for small UAV control. The earth frame coordinates are $[x_e,y_e,z_e]$. The body frame of the UAV is attached to the center of mass with coordinates $[x_b,y_b,z_b]$. x_b is the preferred forward direction of the UAV, and z_b is perpendicular to the plane that points downwards when the UAV travels during perfect horizontal flight.

The orientation of the UAV (body frame) is specified in ZYX Euler angles. To convert from the earth frame to the body frame, we first rotate about the z_e-axis by the yaw angle, ψ. Then, rotate about the intermediate y-axis by the pitch angle, ϕ. Then, rotate about the intermediate x-axis by the roll angle, Θ.

The angular velocity of the UAV is represented by $[p,q,r]$ with respect to the body axes, $[x_b,y_b,z_b]$.

UAV Multirotor Guidance Model Equations

For multirotors, the following equations are used to define the guidance model of the UAV. To calculate the time-derivative of the UAV state using these governing equations, use the `derivative` function. Specify the inputs using `state`, `control`, and `environment`.

The UAV position in the earth frame is $[x_e, y_e, z_e]$ with orientation as ZYX Euler angles, $[\psi, \Theta, \phi]$ in radians. Angular velocities are $[p, q, r]$ in radians per second.

The UAV body frame uses coordinates as $[x_b, y_b, z_b]$.

The rotation matrix that rotates vector from body frame to world frame is:

$$R_b^e = \begin{bmatrix} c_\theta c_\psi & c_\psi s_\phi s_\theta - c_\phi s_\psi & c_\phi c_\psi s_\theta + s_\phi s_\psi \\ c_\theta s_\psi & c_\phi c_\psi + s_\phi s_\theta s_\psi & -c_\psi s_\phi + c_\phi s_\theta s_\psi \\ -s_\theta & c_\theta s_\phi & c_\phi c_\theta \end{bmatrix}$$

The cos(x) and sin(x) are abbreviated as c_x and s_x.

The acceleration of the UAV center of mass in earth coordinates is governed by:

$$m \begin{bmatrix} \ddot{x}_e \\ \ddot{y}_e \\ \ddot{z}_e \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ mg \end{bmatrix} + R_b^e \begin{bmatrix} 0 \\ 0 \\ -F_{thrust} \end{bmatrix}$$

m is the UAV mass, g is gravity, and F_{thrust} is the total force created by the propellers applied to the multirotor along the $-z_b$ axis (points upwards in a horizontal pose).

The closed-loop roll-pitch attitude controller is approximated by the behavior of 2 independent PD controllers for the two rotation angles, and 2 independent P controllers for the yaw rate and thrust. The angular velocity, angular acceleration, and thrust are governed by:

$$J = \begin{bmatrix} 1 & \sin\phi\tan\theta & \cos\phi\tan\theta \\ 0 & \cos\phi & -\sin\phi \\ 0 & \dfrac{\sin\phi}{\cos\theta} & \dfrac{\cos\phi}{\cos\theta} \end{bmatrix}$$

$$\begin{bmatrix} \dot{\phi} \\ \dot{\theta} \\ \dot{\psi} \end{bmatrix} = J \begin{bmatrix} p \\ q \\ r \end{bmatrix}$$

$$\begin{bmatrix} \dot{p} \\ \dot{q} \\ \dot{r} \end{bmatrix} = \begin{bmatrix} 1 & 0 & -\sin\theta \\ 0 & \cos\phi & \sin\phi\cos\theta \\ 0 & -\sin\phi & \cos\phi\cos\theta \end{bmatrix} \begin{bmatrix} KP_{\phi}(\phi^{c} - \phi) + KD_{\phi}(-\dot{\phi}) \\ KP_{\theta}(\theta^{c} - \theta) + KD_{\theta}(-\dot{\theta}) \\ KP_{\psi}(\dot{\psi}^{c} - \dot{\psi}) \end{bmatrix}$$

$$\dot{F}_{thrust} = KP_{F}(F^{c}_{thrust} - F_{thrust})$$

This model assumes the autopilot takes in commanded roll, pitch, yaw rate, $[\phi^c \quad \theta^c \quad \dot{\psi}^c]$ and a commanded total thrust force, F^c_{thrust}. The structure to specify these inputs is generated from `control`.

The P and D gains for the control inputs are specified as KP_{α} and KD_{α}, where α is either the rotation angle or thrust. These gains along with the UAV mass, m, are specified in the `Configuration` property of the `multirotor` object.

From these governing equations, the model gives the following variables:

$$[x_e \quad y_e \quad z_e \quad \dot{x}_e \quad \dot{y}_e \quad \dot{z}_e \quad \psi \quad \theta \quad \phi \quad p \quad q \quad r \quad F_{thrust}]$$

These variables match the output of the `state` function.

Version History

Introduced in R2018b

References

[1] Mellinger, Daniel, and Nathan Michael. "Trajectory Generation and Control for Precise Aggressive Maneuvers with Quadrotors." *The International Journal of Robotics Research*. 2012, pp. 664-74.

Extended Capabilities

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

See Also

Functions
`ode45` | `control` | `derivative` | `environment` | `state` | `plotTransforms`

Objects
`fixedwing` | `uavWaypointFollower`

Blocks
UAV Guidance Model | Waypoint Follower

Topics
"Approximate High-Fidelity UAV Model with UAV Guidance Model Block"
"Tuning Waypoint Follower for Fixed-Wing UAV"

mavlinksigning

Store MAVLink signing channel information

Description

The `mavlinksigning` enables you protect UAVs from unauthorized communication using message signing. Use this object to store up to 16 MAVLink signing channels, and use the `addmavlinkkeys`, `lsmavlinkkeys`, `rmmavlinkkeys` functions to add, list, and remove MAVLink keys in the current MATLAB session for use with signing channels.

Note Note that message signing is not the same as message encryption. MAVLink does not provide message encryption. See MAVLink Message Signing (Authentication) for more details about MAVLink message signing.

Creation

Syntax

`stream = mavlinksigning`

Description

`stream = mavlinksigning` creates a `mavlinksigning` object to store signing channels.

Object Functions

addChannel	Add MAVLink signing channel
removeChannel	Remove MAVLink signing channel

Examples

Add and Remove MAVLink Signing Channels

Create a `mavlinksigning` object to store MAVLink signing channels.

```
stream = mavlinksigning;
```

Load and list the keys from the `keys.env` file.

```
addmavlinkkeys("keys.env");
lsmavlinkkeys
```

```
ans = 1x2 string
    "Key1"    "Key2"
```

Add channel with a system ID of 1, component ID of 2, link ID of 3.

```
addChannel(stream,1,2,3,"Key1")
```

```
ans = struct with fields:
          Stream: [1x1 mavlinksigning]
        SystemID: 1
     ComponentID: 2
          LinkID: 3
             Key: "Key1"
       Timestamp: 25782415012287
    CreationTime: 04-Mar-2023 01:49:10
```

Remove the same channel.

```
removeChannel(stream,1,2,3)
```

Send Signed Messages

Add MAVLink signing keys from the `keys.env` file to the MATLAB session.

```
addmavlinkkeys("keys.env")
lsmavlinkkeys
```

```
ans = 1x2 string
    "Key1"    "Key2"
```

Create MAVLink signing streams for UAV and the ground control system.

```
signingStream = mavlinksigning;
signingChannelUAV = signingStream.addChannel(1,1,1,"Key1");
signingChannelGCS =  signingStream.addChannel(255,1,1,"Key1");
```

Create signed dialect and MAVLink IO object.

```
dialectUAV = mavlinkdialect("common.xml",2,SigningChannel=signingChannelUAV);
dialectGCS = mavlinkdialect("common.xml",2,SigningChannel=signingChannelGCS);
ioUAV = mavlinkio(dialectUAV);
```

Create signed message and display the signature at the end of the buffer.

```
msg = dialectUAV.createmsg("HEARTBEAT");
buffer = ioUAV.serializemsg(msg)
```

```
buffer = 1x34 uint8 row vector

   253     9     1     0     0     1     1     0     0     0     0     0     0     0     0     0
```

Read the signed message.

```
[msgReceived,status] = dialectGCS.deserializemsg(buffer,OutputAllMessages=true)
```

```
msgReceived = struct with fields:
          MsgID: 0
       SystemID: 1
    ComponentID: 1
```

```
        Payload: [1x1 struct]
            Seq: 0

status = 0
```

Version History

Introduced in R2022a

See Also

`mavlinkdialect` | `mavlinkio` | `addmavlinkkeys` | `lsmavlinkkeys` | `rmmavlinkkeys`

polynomialTrajectory

Piecewise-polynomial trajectory generator

Description

The `polynomialTrajectory` System object generates trajectories using a specified piecewise polynomial.

You can create a piecewise-polynomial structure using trajectory generators like `minjerkpolytraj`, and `minsnappolytraj`, as well as any custom trajectory generator. You can then pass the structure to the `polynomialTrajectory` System object to create a trajectory interface for scenario simulation using the `uavScenario` object. You can use the `polynomialTrajectory` object to specify the trajectory for `uavPlatform` motion.

To generate a trajectory from a piecewise polynomial:

1. Create the `polynomialTrajectory` object and set its properties.
2. Call the object with arguments, as if it were a function.

To learn more about how System objects work, see What Are System Objects?

Creation

Syntax

```
trajectory = polynomialTrajectory(pp,Name=Value)
```

Description

`trajectory = polynomialTrajectory(pp,Name=Value)` returns a System object, `trajectory`, that generates a trajectory using the piecewise polynomial `pp`. Specify properties using one or more name-value arguments. Properties that you do not specify retain their default values.

Input Arguments

pp — Piecewise polynomial
structure

Piecewise polynomial, specified as a structure that defines the polynomial for each section of the piecewise trajectory.

Data Types: `struct`

Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see System Design in MATLAB Using System Objects.

`SampleRate` — Sample rate of trajectory (Hz)
`100` (default) | positive scalar

Sample rate of the trajectory in Hz, specified as a positive scalar.

Tunable: Yes

Data Types: `double`

`SamplesPerFrame` — Number of samples per output frame
`1` (default) | positive integer

Number of samples per output frame, specified as a positive integer.

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. *N* is the number of waypoints.

Each `quaternion` must have a norm of `1`. Each 3-by-3 rotation matrix must be an orthonormal matrix. Each quaternion or rotation matrix is a frame rotation from the local navigation coordinate system to the current body coordinate system at the corresponding waypoint.

If you do not specify this property, then the object sets yaw to the direction of travel at each waypoint, and pitch and roll are subject to the values of the AutoPitch and AutoBank properties, respectively.

Data Types: `double`

`Waypoints` — Positions in navigation coordinate system (m)
N-by-3 matrix

This property is read-only.

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.

The object infers this value from the piecewise polynomial `pp`.

Data Types: `double`

`TimeOfArrival` — Timestamp at each waypoint (s)
N-element column vector of nonnegative increasing numbers

This property is read-only.

Timestamp at each waypoint, in seconds, specified as an *N*-element column vector. The number of samples, *N*, is the same as the number of samples (rows) in Waypoints property. The value of each element of the vector must be greater than the value of the previous element.

The object infers this value from the piecewise polynomial pp.

Data Types: double

AutoPitch — Align pitch angle with direction of motion

falseor 0 (default) | trueor 1

Align the pitch angle with the direction of motion, specified as a logical 0 (false) or 1 (true). When specified as true, the pitch angle automatically aligns with the direction of motion. If specified as false, the object sets the pitch angle to 0 (level orientation).

Dependencies

To set this property, you must not specify the Orientation property.

Data Types: logical

AutoBank — Align roll angle to counteract centripetal force

falseor 0 (default) | trueor 1

Align the roll angle to counteract the centripetal force, specified as a logical 0 (false) or 1 (true). When specified as true, the roll angle automatically counteracts the centripetal force. If specified as false, the object sets the roll angle to 0 (flat orientation).

Dependencies

To set this property, you must not specify the Orientation property.

Data Types: logical

ReferenceFrame — Reference frame of trajectory

"NED" (default) | "ENU"

Reference frame of the trajectory, specified as "NED" (North-East-Down) or "ENU" (East-North-Up).

Data Types: char | string

Velocities — Velocity in navigation coordinate system at each waypoint (m/s)

N-by-3 matrix

This property is read-only.

Velocity in the navigation coordinate system at each waypoint, 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*, is the same as the number of samples (rows) in Waypoints property.

The object infers this value from the derivative of the piecewise polynomial pp.

Data Types: double

Course — Horizontal direction of travel (degrees)

N-element real vector

This property is read-only.

Horizontal direction of travel, in degrees, specified as an *N*-element real vector. The number of samples, *N*, is the same as the number of samples (rows) in Waypoints property.

The object infers this value from the derivative of the piecewise polynomial pp.

Data Types: `double`

`GroundSpeed` — Groundspeed at each waypoint (m/s)
N-element real vector

This property is read-only.

Groundspeed at each waypoint, in meters per second, specified as an *N*-element real vector. The number of samples, *N*, is the same as the number of samples (rows) in Waypoints property.

The object infers this value from the derivative of the piecewise polynomial `pp`.

Data Types: `double`

`ClimbRate` — Climb rate at each waypoint (m/s)
N-element real vector

This property is read-only.

Climb Rate at each waypoint, in meters per second, specified as an *N*-element real vector. The number of samples, *N*, is the same as the number of samples (rows) in Waypoints property.

The object infers this value from the derivative of the piecewise polynomial `pp`.

Data Types: `double`

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. The trajectory returns `NaN` for positions and orientations outside the range of the time of arrival.

Output Arguments

`position` — Position in local navigation coordinate system
M-by-3 matrix

Position in the local navigation coordinate system, returned as an *M*-by-3 matrix in meters.

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*-element `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 at the corresponding sample.

M is specified by the SamplesPerFrame property.

Data Types: `double`

`velocity` — Velocity in local navigation coordinate system
M-by-3 matrix

Velocity in the local navigation coordinate system, returned as an *M*-by-3 matrix, in meters per second.

M is specified by the SamplesPerFrame property.

Data Types: `double`

`acceleration` — Acceleration in local navigation coordinate system
M-by-3 matrix

Acceleration in the local navigation coordinate system, returned as an *M*-by-3 matrix, in meters per second squared.

M is specified by the SamplesPerFrame property.

Data Types: `double`

`angularVelocity` — Angular velocity in local navigation coordinate system
M-by-3 matrix

Angular velocity in the local navigation coordinate system, returned as an *M*-by-3 matrix, in radians per second.

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 `polynomialTrajectory`

lookupPose	Obtain pose information for certain time
waypointInfo	Get waypoint information table

Common to All System Objects

step	Run System object algorithm
release	Release resources and allow changes to System object property values and input characteristics
clone	Create duplicate System object
isLocked	Determine if System object is in use

reset	Reset internal states of System object
isDone	End-of-data status

Examples

Generate Trajectory from Piecewise Polynomial Using polynomialTrajectory

Use the `minjerkpolytraj` function to generate the piecewise polynomial and the time samples for the specified waypoints of a trajectory.

```
waypoints = [0 20 20 0 0; 0 0 5 5 0; 0 5 10 5 0];
timePoints = cumsum([0 10 1.25*pi 10 1.25*pi]);
numSamples = 100;
[~,~,~,~,pp,~,tsamples] = minjerkpolytraj(waypoints,timePoints,numSamples);
```

Use the `polynomialTrajectory` System object to generate a trajectory from the piecewise polynomial that a multirotor must follow. Specify the sample rate of the trajectory and the orientation at each waypoint.

```
eulerAngles = [0 0 0; 0 0 0; 180 0 0; 180 0 0; 0 0 0];
q = quaternion(eulerAngles,"eulerd","ZYX","frame");
traj = polynomialTrajectory(pp,SampleRate=100,Orientation=q);
```

Inspect the waypoints, times of arrival, and orientation by using `waypointInfo`.

```
waypointInfo(traj)
```

```
ans=5×3 table
    TimeOfArrival                     Waypoints                         Orientation
    _____________    _______________________________________    ________________

           0                 0              0              0    {1x1 quaternion}
          10                20              0              5    {1x1 quaternion}
      13.927                20              5             10    {1x1 quaternion}
      23.927                 0              5              5    {1x1 quaternion}
      27.854         6.409e-14    -1.1102e-13    -1.1902e-13    {1x1 quaternion}
```

Obtain pose information one buffer frame at a time.

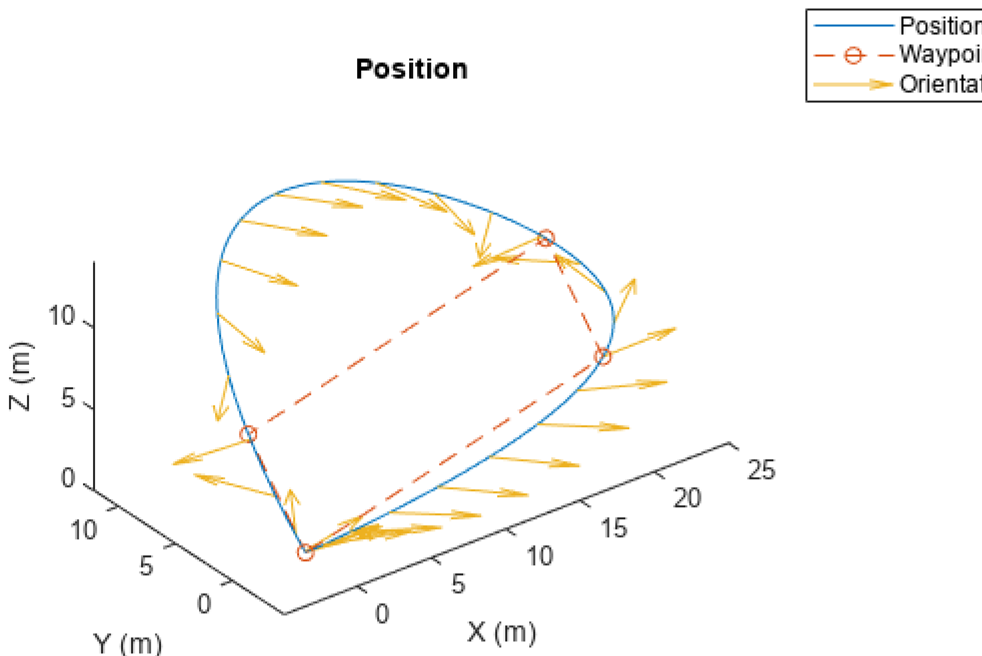
```
[pos,orient,vel,acc,angvel] = traj();
i = 1;
spf = traj.SamplesPerFrame;
while ~isDone(traj)
    idx = (i+1):(i+spf);
    [pos(idx,:),orient(idx,:), ...
     vel(idx,:),acc(idx,:),angvel(idx,:)] = traj();
    i = i + spf;
end
```

Get the yaw angle from the orientation.

```
eulOrientation = quat2eul(orient);
yawAngle = eulOrientation(:,1);
```

Plot the generated positions and orientations, as well as the specified waypoints.

```
plot3(pos(:,1),pos(:,2),pos(:,3), ...
      waypoints(1,:),waypoints(2,:),waypoints(3,:),"--o")
hold on
% Plot the yaw using quiver.
quiverIdx = 1:100:length(pos);
quiver3(pos(quiverIdx,1),pos(quiverIdx,2),pos(quiverIdx,3), ...
        cos(yawAngle(quiverIdx)),sin(yawAngle(quiverIdx)), ...
        zeros(numel(quiverIdx),1))
title("Position")
xlabel("X (m)")
ylabel("Y (m)")
zlabel("Z (m)")
legend({"Position","Waypoints","Orientation"})
axis equal
hold off
```



Obtain Pose Information of Polynomial Trajectory at Certain Time

Use the `minsnappolytraj` function to generate the piecewise polynomial and the time samples for the specified waypoints of a trajectory.

```
waypoints = [0 20 20 0 0; 0 0 5 5 0; 0 5 10 5 0];
timePoints = linspace(0,30,5);
numSamples = 100;
[~,~,~,~,~,pp,~,~] = minsnappolytraj(waypoints,timePoints,numSamples);
```

Use the `polynomialTrajectory` System object to generate a trajectory from the piecewise polynomial. Specify the sample rate of the trajectory.

```
traj = polynomialTrajectory(pp,SampleRate=100);
```

Inspect the waypoints and times of arrival by using `waypointInfo`.

```
waypointInfo(traj)
```

```
ans=5×2 table
    TimeOfArrival                    Waypoints
    _____________    ________________________________________

          0                  0              0               0
        7.5                 20              0               5
         15                 20              5              10
       22.5                  0              5               5
         30         3.3973e-13    -2.7018e-12     -2.6041e-12
```

Obtain the time of arrival between the second and fourth waypoint. Create timestamps to sample the trajectory.

```
t0 = traj.TimeOfArrival(2);
tf = traj.TimeOfArrival(4);
sampleTimes = linspace(t0,tf,1000);
```

Obtain the position, orientation, velocity, and acceleration information at the sampled timestamps using the `lookupPose` object function.

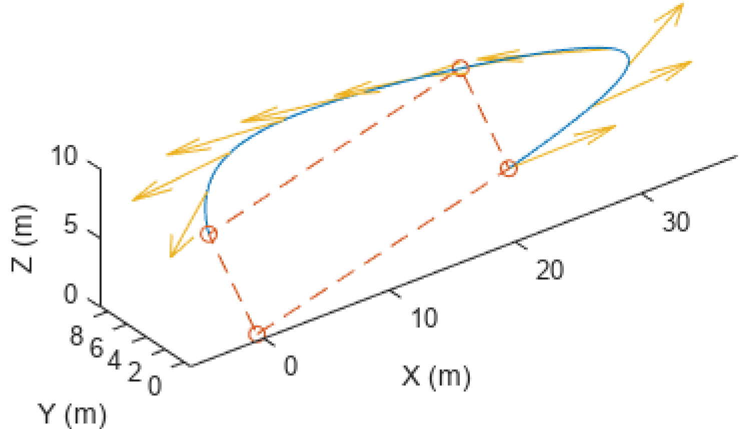
```
[pos,orient,vel,accel,~] = lookupPose(traj,sampleTimes);
```

Get the yaw angle from the orientation.

```
eulOrientation = quat2eul(orient);
yawAngle = eulOrientation(:,1);
```

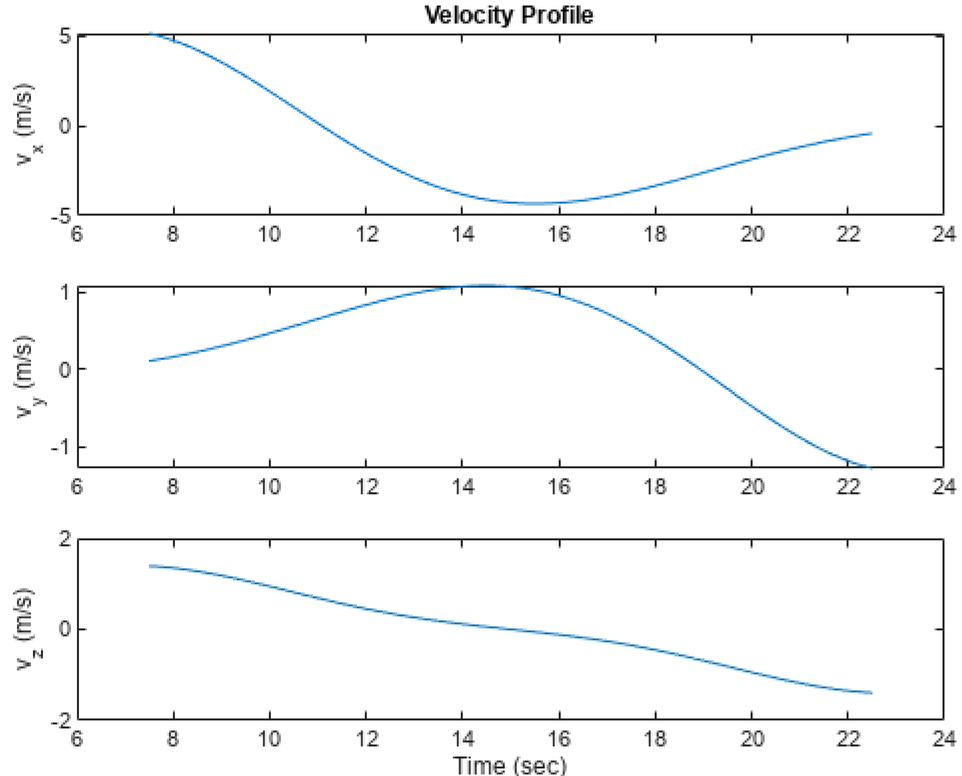
Plot the generated positions and orientations, as well as the specified waypoints.

```
plot3(pos(:,1),pos(:,2),pos(:,3), ...
      waypoints(1,:),waypoints(2,:),waypoints(3,:),"--o")
hold on
% Plot the yaw using quiver.
quiverIdx = 1:100:length(pos);
quiver3(pos(quiverIdx,1),pos(quiverIdx,2),pos(quiverIdx,3), ...
        cos(yawAngle(quiverIdx)),sin(yawAngle(quiverIdx)), ...
        zeros(numel(quiverIdx),1))
title("Position and Orientation")
xlabel("X (m)")
ylabel("Y (m)")
zlabel("Z (m)")
legend({"Position","Waypoints","Orientation"})
axis equal
hold off
```



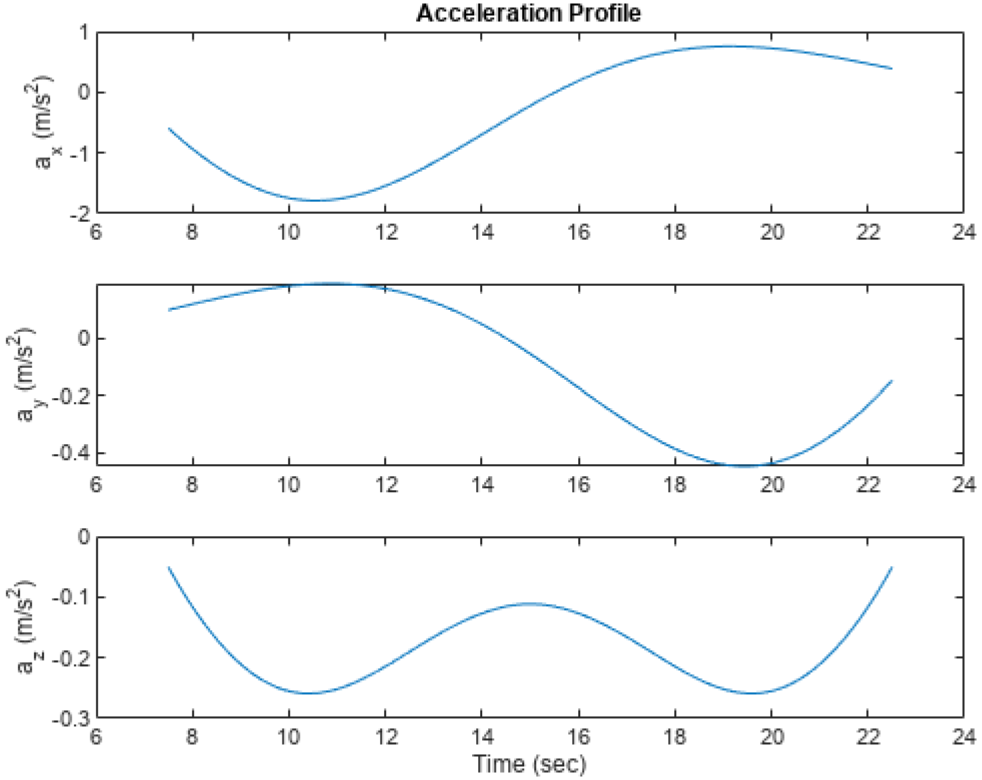
Plot the velocity profiles.

```
figure
subplot(3,1,1)
plot(sampleTimes,vel(:,1))
title("Velocity Profile")
ylabel("v_x (m/s)")
subplot(3,1,2)
plot(sampleTimes,vel(:,2))
ylabel("v_y (m/s)")
subplot(3,1,3)
plot(sampleTimes,vel(:,3))
ylabel("v_z (m/s)")
xlabel("Time (sec)")
```



Plot the acceleration profiles.

```
figure
subplot(3,1,1)
plot(sampleTimes,accel(:,1))
title("Acceleration Profile")
ylabel("a_x (m/s^2)")
subplot(3,1,2)
plot(sampleTimes,accel(:,2))
ylabel("a_y (m/s^2)")
subplot(3,1,3)
plot(sampleTimes,accel(:,3))
ylabel("a_z (m/s^2)")
xlabel("Time (sec)")
```



Version History

Introduced in R2023a

Extended Capabilities

C/C++ Code Generation

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

See Also

Objects

`waypointTrajectory` | `uavPlatform`

Functions

`waypointInfo` | `lookupPose`

quaternion

Create a quaternion array

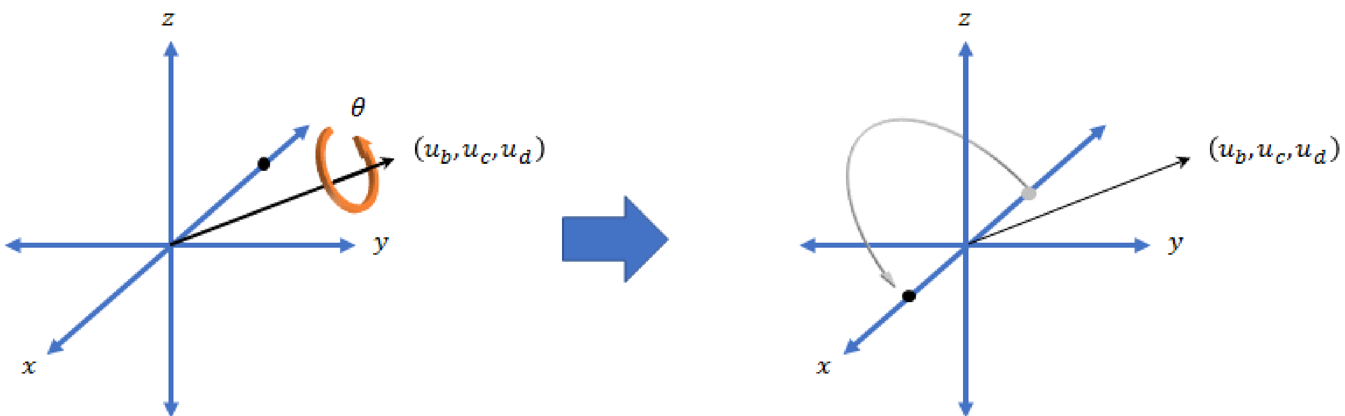
Description

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

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

The set of quaternions, denoted by **H**, is defined within a four-dimensional vector space over the real numbers, $\mathbf{R}^4$. Every element of **H** has a unique representation based on a linear combination of the basis elements, i, j, and k.

All rotations in 3-D can be described by an axis of rotation and angle about that axis. An advantage of quaternions over rotation matrices is that the axis and angle of rotation is easy to interpret. For example, consider a point in $\mathbf{R}^3$. To rotate the point, you define an axis of rotation and an angle of rotation.



The quaternion representation of the rotation may be expressed as
$q = \cos\left(\theta/2\right) + \sin\left(\theta/2\right)(u_b\text{i} + u_c\text{j} + u_d\text{k})$, where θ is the angle of rotation and $[u_b$, u_c, and $u_d]$ is the axis of rotation.

Creation

Syntax

```
quat = quaternion()
quat = quaternion(A,B,C,D)
quat = quaternion(matrix)
quat = quaternion(RV,'rotvec')
```

```
quat = quaternion(RV,'rotvecd')
quat = quaternion(RM,'rotmat',PF)
quat = quaternion(E,'euler',RS,PF)
quat = quaternion(E,'eulerd',RS,PF)
quat = quaternion(transformation)
quat = quaternion(rotation)
```

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`.

`quat = quaternion(transformation)` creates a quaternion array from the SE(3) transformation `transformation`.

`quat = quaternion(rotation)` creates an quaternion array from the SO(3) rotation `rotation`.

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];
```

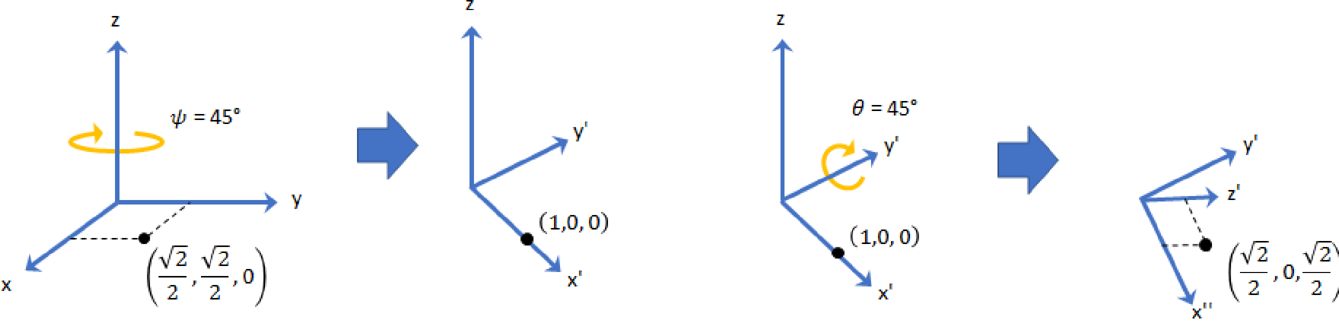
In this representation, the first column represents the *x*-axis, the second column represents the *y*-axis, and the third column represents the *z*-axis.

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)
  ```

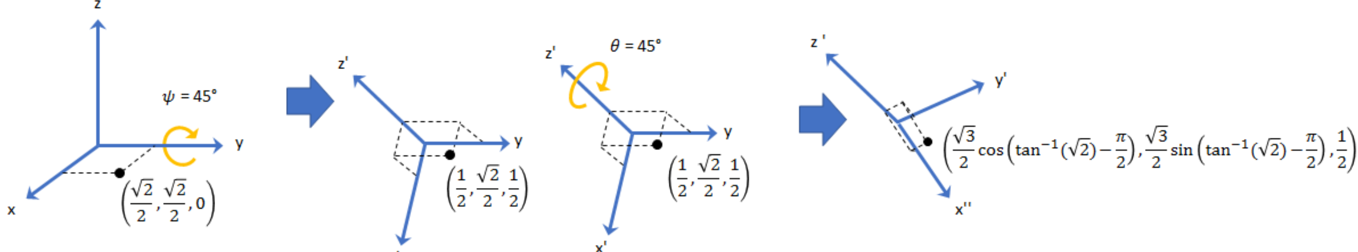
  ```
  newPointCoordinate =

      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`

`transformation` — Homogeneous transformation
`se3` object | *N*-element array of `se3` objects

Transformation, specified as an `se3` object, or as an *N*-element array of `se3` objects. *N* is the total number of transformations.

The `quaternion` object ignores the translational component of the transformation and converts the rotational 3-by-3 submatrix of the transformation to a quaternion.

`rotation` — Orthonormal rotation
`so3` object | *N*-element array of `so3` objects

Orthonormal rotation, specified as an `so3` object, or as an *N*-element array of `so3` objects. *N* is the total number of rotations.

Object Functions

angvel	Angular velocity from quaternion array
classUnderlying	Class of parts within quaternion
compact	Convert quaternion array to N-by-4 matrix
conj	Complex conjugate of quaternion
'	Complex conjugate transpose of quaternion array
dist	Angular distance in radians
euler	Convert quaternion to Euler angles (radians)
eulerd	Convert quaternion to Euler angles (degrees)
exp	Exponential of quaternion array
.\,ldivide	Element-wise quaternion left division
log	Natural logarithm of quaternion array
meanrot	Quaternion mean rotation
-	Quaternion subtraction
*	Quaternion multiplication
norm	Quaternion norm
normalize	Quaternion normalization
ones	Create quaternion array with real parts set to one and imaginary parts set to zero
parts	Extract quaternion parts

.^,power	Element-wise quaternion power
prod	Product of a quaternion array
randrot	Uniformly distributed random rotations
./,rdivide	Element-wise quaternion right division
rotateframe	Quaternion frame rotation
rotatepoint	Quaternion point rotation
rotmat	Convert quaternion to rotation matrix
rotvec	Convert quaternion to rotation vector (radians)
rotvecd	Convert quaternion to rotation vector (degrees)
slerp	Spherical linear interpolation
.*,times	Element-wise quaternion multiplication
'	Transpose a quaternion array
-	Quaternion unary minus
zeros	Create quaternion array with all parts set to zero

Examples

Create Empty Quaternion

```
quat = quaternion()
```

```
quat =

  0x0 empty quaternion array
```

By default, the underlying class of the quaternion is a double.

```
classUnderlying(quat)
```

```
ans =
'double'
```

Create Quaternion by Specifying Individual Quaternion Parts

You can create a quaternion array by specifying the four parts as comma-separated scalars, matrices, or multidimensional arrays of the same size.

Define quaternion parts as scalars.

```
A = 1.1;
B = 2.1;
C = 3.1;
D = 4.1;
quatScalar = quaternion(A,B,C,D)
```

```
quatScalar = quaternion
     1.1 + 2.1i + 3.1j + 4.1k
```

Define quaternion parts as column vectors.

```
A = [1.1;1.2];
B = [2.1;2.2];
```

```
C = [3.1;3.2];
D = [4.1;4.2];
quatVector = quaternion(A,B,C,D)
```

```
quatVector = 2x1 quaternion array
     1.1 + 2.1i + 3.1j + 4.1k
     1.2 + 2.2i + 3.2j + 4.2k
```

Define quaternion parts as matrices.

```
A = [1.1,1.3; ...
     1.2,1.4];
B = [2.1,2.3; ...
     2.2,2.4];
C = [3.1,3.3; ...
     3.2,3.4];
D = [4.1,4.3; ...
     4.2,4.4];
quatMatrix = quaternion(A,B,C,D)
```

```
quatMatrix = 2x2 quaternion array
     1.1 + 2.1i + 3.1j + 4.1k     1.3 + 2.3i + 3.3j + 4.3k
     1.2 + 2.2i + 3.2j + 4.2k     1.4 + 2.4i + 3.4j + 4.4k
```

Define quaternion parts as three dimensional arrays.

```
A = randn(2,2,2);
B = zeros(2,2,2);
C = zeros(2,2,2);
D = zeros(2,2,2);
quatMultiDimArray = quaternion(A,B,C,D)
```

```
quatMultiDimArray = 2x2x2 quaternion array
quatMultiDimArray(:,:,1) =

     0.53767 +       0i +       0j +       0k     -2.2588 +       0i +       0j +       0k
      1.8339 +       0i +       0j +       0k     0.86217 +       0i +       0j +       0k


quatMultiDimArray(:,:,2) =

     0.31877 +       0i +       0j +       0k    -0.43359 +       0i +       0j +       0k
     -1.3077 +       0i +       0j +       0k     0.34262 +       0i +       0j +       0k
```

Create Quaternion by Specifying Quaternion Parts Matrix

You can create a scalar or column vector of quaternions by specify an *N*-by-4 matrix of quaternion parts, where columns correspond to the quaternion parts A, B, C, and D.

Create a column vector of random quaternions.

```
quatParts = rand(3,4)
```

```
quatParts = 3×4
```

```
    0.8147    0.9134    0.2785    0.9649
    0.9058    0.6324    0.5469    0.1576
    0.1270    0.0975    0.9575    0.9706
```

```
quat = quaternion(quatParts)
```

```
quat = 3x1 quaternion array
     0.81472 + 0.91338i +  0.2785j + 0.96489k
     0.90579 + 0.63236i + 0.54688j + 0.15761k
     0.12699 + 0.09754i + 0.95751j + 0.97059k
```

To retrieve the `quatParts` matrix from quaternion representation, use `compact`.

```
retrievedquatParts = compact(quat)
```

```
retrievedquatParts = 3×4

    0.8147    0.9134    0.2785    0.9649
    0.9058    0.6324    0.5469    0.1576
    0.1270    0.0975    0.9575    0.9706
```

Create Quaternion by Specifying Rotation Vectors

You can create an *N*-by-1 quaternion array by specifying an *N*-by-3 matrix of rotation vectors in radians or degrees. Rotation vectors are compact spatial representations that have a one-to-one relationship with normalized quaternions.

Rotation Vectors in Radians

Create a scalar quaternion using a rotation vector and verify the resulting quaternion is normalized.

```
rotationVector = [0.3491,0.6283,0.3491];
quat = quaternion(rotationVector,'rotvec')
```

```
quat = quaternion
     0.92124 + 0.16994i + 0.30586j + 0.16994k
```

```
norm(quat)
```

```
ans = 1.0000
```

You can convert from quaternions to rotation vectors in radians using the `rotvec` function. Recover the `rotationVector` from the quaternion, `quat`.

```
rotvec(quat)
```

```
ans = 1×3

    0.3491    0.6283    0.3491
```

Rotation Vectors in Degrees

Create a scalar quaternion using a rotation vector and verify the resulting quaternion is normalized.

```
rotationVector = [20,36,20];
quat = quaternion(rotationVector,'rotvecd')
```

```
quat = quaternion
     0.92125 + 0.16993i + 0.30587j + 0.16993k
```

```
norm(quat)
```

```
ans = 1
```

You can convert from quaternions to rotation vectors in degrees using the `rotvecd` function. Recover the `rotationVector` from the quaternion, `quat`.

```
rotvecd(quat)
```

```
ans = 1×3

   20.0000   36.0000   20.0000
```

Create Quaternion by Specifying Rotation Matrices

You can create an N-by-1 quaternion array by specifying a 3-by-3-by-N array of rotation matrices. Each page of the rotation matrix array corresponds to one element of the quaternion array.

Create a scalar quaternion using a 3-by-3 rotation matrix. Specify whether the rotation matrix should be interpreted as a frame or point rotation.

```
rotationMatrix = [1 0         0; ...
                  0 sqrt(3)/2 0.5; ...
                  0 -0.5      sqrt(3)/2];
quat = quaternion(rotationMatrix,'rotmat','frame')
```

```
quat = quaternion
     0.96593 + 0.25882i +       0j +       0k
```

You can convert from quaternions to rotation matrices using the `rotmat` function. Recover the `rotationMatrix` from the quaternion, `quat`.

```
rotmat(quat,'frame')
```

```
ans = 3×3

    1.0000         0         0
         0    0.8660    0.5000
         0   -0.5000    0.8660
```

Create Quaternion by Specifying Euler Angles

You can create an *N*-by-1 quaternion array by specifying an *N*-by-3 array of Euler angles in radians or degrees.

Euler Angles in Radians

Use the `euler` syntax to create a scalar quaternion using a 1-by-3 vector of Euler angles in radians. Specify the rotation sequence of the Euler angles and whether the angles represent a frame or point rotation.

```
E = [pi/2,0,pi/4];
quat = quaternion(E,'euler','ZYX','frame')
```

```
quat = quaternion
     0.65328 +  0.2706i +  0.2706j + 0.65328k
```

You can convert from quaternions to Euler angles using the `euler` function. Recover the Euler angles, `E`, from the quaternion, `quat`.

```
euler(quat,'ZYX','frame')
```

```
ans = 1×3

    1.5708         0    0.7854
```

Euler Angles in Degrees

Use the `eulerd` syntax to create a scalar quaternion using a 1-by-3 vector of Euler angles in degrees. Specify the rotation sequence of the Euler angles and whether the angles represent a frame or point rotation.

```
E = [90,0,45];
quat = quaternion(E,'eulerd','ZYX','frame')
```

```
quat = quaternion
     0.65328 +  0.2706i +  0.2706j + 0.65328k
```

You can convert from quaternions to Euler angles in degrees using the `eulerd` function. Recover the Euler angles, `E`, from the quaternion, `quat`.

```
eulerd(quat,'ZYX','frame')
```

```
ans = 1×3

   90.0000         0   45.0000
```

Quaternion Algebra

Quaternions form a noncommutative associative algebra over the real numbers. This example illustrates the rules of quaternion algebra.

Addition and Subtraction

Quaternion addition and subtraction occur part-by-part, and are commutative:

```
Q1 = quaternion(1,2,3,4)
```

```
Q1 = quaternion
     1 + 2i + 3j + 4k
```

```
Q2 = quaternion(9,8,7,6)
```

```
Q2 = quaternion
     9 + 8i + 7j + 6k
```

```
Q1plusQ2 = Q1 + Q2
```

```
Q1plusQ2 = quaternion
     10 + 10i + 10j + 10k
```

```
Q2plusQ1 = Q2 + Q1
```

```
Q2plusQ1 = quaternion
     10 + 10i + 10j + 10k
```

```
Q1minusQ2 = Q1 - Q2
```

```
Q1minusQ2 = quaternion
    -8 - 6i - 4j - 2k
```

```
Q2minusQ1 = Q2 - Q1
```

```
Q2minusQ1 = quaternion
     8 + 6i + 4j + 2k
```

You can also perform addition and subtraction of real numbers and quaternions. The first part of a quaternion is referred to as the *real* part, while the second, third, and fourth parts are referred to as the *vector*. Addition and subtraction with real numbers affect only the real part of the quaternion.

```
Q1plusRealNumber = Q1 + 5
```

```
Q1plusRealNumber = quaternion
     6 + 2i + 3j + 4k
```

```
Q1minusRealNumber = Q1 - 5
```

```
Q1minusRealNumber = quaternion
    -4 + 2i + 3j + 4k
```

Multiplication

Quaternion multiplication is determined by the products of the basis elements and the distributive law. Recall that multiplication of the basis elements, *i*, *j*, and *k*, are not commutative, and therefore quaternion multiplication is not commutative.

```
Q1timesQ2 = Q1 * Q2
```

```
Q1timesQ2 = quaternion
    -52 + 16i + 54j + 32k
```

```
Q2timesQ1 = Q2 * Q1
```

```
Q2timesQ1 = quaternion
    -52 + 36i + 14j + 52k
```

```
isequal(Q1timesQ2,Q2timesQ1)
```

```
ans = logical
   0
```

You can also multiply a quaternion by a real number. If you multiply a quaternion by a real number, each part of the quaternion is multiplied by the real number individually:

```
Q1times5 = Q1*5
```

```
Q1times5 = quaternion
      5 + 10i + 15j + 20k
```

Multiplying a quaternion by a real number is commutative.

```
isequal(Q1*5,5*Q1)
```

```
ans = logical
   1
```

Conjugation

The complex conjugate of a quaternion is defined such that each element of the vector portion of the quaternion is negated.

```
Q1
```

```
Q1 = quaternion
     1 + 2i + 3j + 4k
```

```
conj(Q1)
```

```
ans = quaternion
     1 - 2i - 3j - 4k
```

Multiplication between a quaternion and its conjugate is commutative:

```
isequal(Q1*conj(Q1),conj(Q1)*Q1)
```

```
ans = logical
   1
```

Quaternion Array Manipulation

You can organize quaternions into vectors, matrices, and multidimensional arrays. Built-in MATLAB® functions have been enhanced to work with quaternions.

Concatenate

Quaternions are treated as individual objects during concatenation and follow MATLAB rules for array manipulation.

```
Q1 = quaternion(1,2,3,4);
Q2 = quaternion(9,8,7,6);

qVector = [Q1,Q2]
```

```
qVector = 1x2 quaternion array
     1 + 2i + 3j + 4k     9 + 8i + 7j + 6k
```

```
Q3 = quaternion(-1,-2,-3,-4);
Q4 = quaternion(-9,-8,-7,-6);

qMatrix = [qVector;Q3,Q4]
```

```
qMatrix = 2x2 quaternion array
     1 + 2i + 3j + 4k     9 + 8i + 7j + 6k
    -1 - 2i - 3j - 4k    -9 - 8i - 7j - 6k
```

```
qMultiDimensionalArray(:,:,1) = qMatrix;
qMultiDimensionalArray(:,:,2) = qMatrix
```

```
qMultiDimensionalArray = 2x2x2 quaternion array
qMultiDimensionalArray(:,:,1) =

     1 + 2i + 3j + 4k     9 + 8i + 7j + 6k
    -1 - 2i - 3j - 4k    -9 - 8i - 7j - 6k


qMultiDimensionalArray(:,:,2) =

     1 + 2i + 3j + 4k     9 + 8i + 7j + 6k
    -1 - 2i - 3j - 4k    -9 - 8i - 7j - 6k
```

Indexing

To access or assign elements in a quaternion array, use indexing.

```
qLoc2 = qMultiDimensionalArray(2)
```

```
qLoc2 = quaternion
    -1 - 2i - 3j - 4k
```

Replace the quaternion at index two with a quaternion one.

```
qMultiDimensionalArray(2) = ones('quaternion')
```

```
qMultiDimensionalArray = 2x2x2 quaternion array
qMultiDimensionalArray(:,:,1) =

     1 + 2i + 3j + 4k     9 + 8i + 7j + 6k
     1 + 0i + 0j + 0k    -9 - 8i - 7j - 6k


qMultiDimensionalArray(:,:,2) =

     1 + 2i + 3j + 4k     9 + 8i + 7j + 6k
    -1 - 2i - 3j - 4k    -9 - 8i - 7j - 6k
```

Reshape

To reshape quaternion arrays, use the `reshape` function.

```
qMatReshaped = reshape(qMatrix,4,1)
```

```
qMatReshaped = 4x1 quaternion array
     1 + 2i + 3j + 4k
    -1 - 2i - 3j - 4k
     9 + 8i + 7j + 6k
    -9 - 8i - 7j - 6k
```

Transpose

To transpose quaternion vectors and matrices, use the `transpose` function.

```
qMatTransposed = transpose(qMatrix)
```

```
qMatTransposed = 2x2 quaternion array
     1 + 2i + 3j + 4k    -1 - 2i - 3j - 4k
     9 + 8i + 7j + 6k    -9 - 8i - 7j - 6k
```

Permute

To permute quaternion vectors, matrices, and multidimensional arrays, use the `permute` function.

```
qMultiDimensionalArray
```

```
qMultiDimensionalArray = 2x2x2 quaternion array
qMultiDimensionalArray(:,:,1) =

     1 + 2i + 3j + 4k     9 + 8i + 7j + 6k
     1 + 0i + 0j + 0k    -9 - 8i - 7j - 6k


qMultiDimensionalArray(:,:,2) =
```

```
     1 + 2i + 3j + 4k     9 + 8i + 7j + 6k
    -1 - 2i - 3j - 4k    -9 - 8i - 7j - 6k
```

```
qMatPermute = permute(qMultiDimensionalArray,[3,1,2])
```

```
qMatPermute = 2x2x2 quaternion array
qMatPermute(:,:,1) =

     1 + 2i + 3j + 4k     1 + 0i + 0j + 0k
     1 + 2i + 3j + 4k    -1 - 2i - 3j - 4k


qMatPermute(:,:,2) =

     9 + 8i + 7j + 6k    -9 - 8i - 7j - 6k
     9 + 8i + 7j + 6k    -9 - 8i - 7j - 6k
```

Version History

Introduced in R2020b

Extended Capabilities

C/C++ Code Generation

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

See Also

se3 | so3

se3

SE(3) homogeneous transformation

Description

The `se3` object represents an SE(3) transformation as a 3-D homogeneous transformation matrix consisting of a translation and rotation:

For more information, see the "3-D Homogeneous Transformation Matrix" on page 1-116 section.

This object acts like a numerical matrix enabling you to compose poses using multiplication and division.

Creation

Syntax

```
transformation = se3
transformation = se3(rotation)
transformation = se3(rotation,translation)
transformation = se3(transformation)

transformation = se3(euler,"eul")
transformation = se3(euler,"eul",sequence)
transformation = se3(quat,"quat")
transformation = se3(quaternion)
transformation = se3(axang,"axang")
transformation = se3(angle,axis)
transformation = se3( ___ ,translation,"trvec")

transformation = se3(translation,"trvec")
transformation = se3(pose,"xyzquat")
```

Description

Rotation Matrices, Translation Vectors, and Transformation Matrices

`transformation = se3` creates an SE(3) transformation representing an identity rotation with no translation.

$$transformation = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

`transformation = se3(rotation)` creates an SE(3) transformation representing a pure rotation defined by the orthonormal rotation `rotation` with no translation. The rotation matrix is represented by the elements in the top left of the `transformation` matrix.

$$rotation = \begin{bmatrix} r_{11} & r_{12} & r_{13} \\ r_{11} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{bmatrix}$$

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

`transformation = se3(rotation,translation)` creates an SE(3) transformation representing a rotation defined by the orthonormal rotation `rotation` and the translation `translation`. The function applies the rotation matrix first, then translation vector to create the transformation.

$$rotation = \begin{bmatrix} r_{11} & r_{12} & r_{13} \\ r_{11} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{bmatrix}, translation = \begin{bmatrix} t_1 \\ t_2 \\ t_3 \end{bmatrix}$$

$$transformation = \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_1 \\ r_{21} & r_{22} & r_{23} & t_2 \\ r_{31} & r_{32} & r_{33} & t_3 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & t_1 \\ 0 & 1 & 0 & t_2 \\ 0 & 0 & 1 & t_3 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} r_{11} & r_{12} & r_{13} & 0 \\ r_{21} & r_{22} & r_{23} & 0 \\ r_{31} & r_{32} & r_{33} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

`transformation = se3(transformation)` creates an SE(3) transformation representing a translation and rotation as defined by the homogeneous transformation `transformation`.

Other 3-D Rotation Representations

`transformation = se3(euler,"eul")` creates an SE(3) transformation from the rotations defined by the Euler angles `euler`.

`transformation = se3(euler,"eul",sequence)` specifies the sequence of the Euler angle rotations `sequence`. For example, the sequence `"ZYX"` rotates the *z*-axis, then the *y*-axis and *x*-axis.

`transformation = se3(quat,"quat")` creates an SE(3) transformation from the rotations defined by the numeric quaternions `quat`.

`transformation = se3(quaternion)` creates an SE(3) transformation from the rotations defined by the quaternion `quaternion`.

`transformation = se3(axang,"axang")` creates an SE(3) transformation from the rotations defined by the axis-angle rotation `axang`.

`transformation = se3(angle,axis)` creates an SE(3) transformation from the rotations `angles` about the rotation axis `axis`.

`transformation = se3( ___ ,translation,"trvec")` creates an SE(3) transformation from the translation vector `translation` along with any other type of rotation input arguments.

Other Translations and Transformation Representations

`transformation = se3(translation,"trvec")` creates an SE(3) transformation from the translation vector `translation`.

`transformation = se3(pose,"xyzquat")` creates an SE(3) transformation from the 3-D compact pose `pose`.

Note If any inputs contain more than one rotation, translation, or transformation, then the output `transformation` is an *N*-element array of `se3` objects corresponding to each of the *N* input rotations, translations, or transformations.

Input Arguments

`rotation` — Orthonormal rotation

3-by-3 matrix | 3-by-3-by-*N* matrix | `so3` object | *N*-element array of `so3` objects

Orthonormal rotation, specified as a 3-by-3 matrix, a 3-by-3-by-*N* array, a scalar `so3` object, or an *N*-element array of `so3` objects. *N* is the total number of rotations.

If `rotation` contains more than one rotation and you also specify `translation` at construction, the number of translations in `translation` must be one or equal to the number of rotations in `rotation`. The resulting number of transformation objects is equal to the value of the `translation` or `rotation` argument, whichever is larger.

Example: `eye(3)`

Data Types: `single` | `double`

`translation` — Translation

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

Translation, specified as a three-element row vector or an *N*-by-3 array. *N* is the total number of translations and each translation is of the form [*x y z*].

If `translation` contains more than one translation, the number of rotations in `rotation` must be one or equal to the number of translations in `translation`. The resulting number of created transformation objects is equal to the value of the `translation` or `rotation` argument, whichever is larger.

Example: `[1 4 3]`

Data Types: `single` | `double`

`transformation` — Homogeneous transformation

4-by-4 matrix | 4-by-4-*N* array | `se3` object | *N*-element array of `se3` objects

Homogeneous transformation, specified as a 4-by-4 matrix, a 4-by-4-*N* array, a scalar `se3` object, or an *N*-element array of `se3` objects. *N* is the total number of transformations specified.

If `transformation` is an array, the resulting number of created `se3` objects is equal to *N*.

Example: `eye(4)`

Data Types: `single` | `double`

`quaternion` — Quaternion

`quaternion` object | *N*-element array of `quaternion` objects

Quaternion, specified as a scalar `quaternion` object or as an *N*-element array of `quaternion` objects. *N* is the total number of specified quaternions.

If `quaternion` is an N-element array, the resulting number of created `se3` objects is equal to N.

Example: `quaternion(1,0.2,0.4,0.2)`

`euler` — Euler angles
N-by-3 matrix

Euler angles, specified as an N-by-3 matrix, in radians. Each row represents one set of Euler angles with the axis-rotation sequence defined by the `sequence` argument. The default axis-rotation sequence is ZYX.

If `euler` is an N-by-3 matrix, the resulting number of created `se3` objects is equal to N.

Example: `[pi/2 pi pi/4]`

Data Types: `single` | `double`

`sequence` — Axis-rotation sequence
`"ZYX"` (default) | `"ZYZ"` | `"ZXY"` | `"ZXZ"` | `"YXY"` | `"YZX"` | `"YXZ"` | `"YZY"` | `"XYX"` | `"XYZ"` | `"XZX"` | `"XZY"`

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

- `"ZYX"` (default)
- `"ZYZ"`
- `"ZXY"`
- `"ZXZ"`
- `"YXY"`
- `"YZX"`
- `"YXZ"`
- `"YZY"`
- `"XYX"`
- `"XYZ"`
- `"XZX"`
- `"XZY"`

These are orthonormal rotation matrices for rotations of ϕ, ψ, and θ about the x-, y-, and z-axis, respectively:

$$R_x(\phi) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\phi & \sin\phi \\ 0 & -\sin\phi & \cos\phi \end{bmatrix}, R_y(\psi) = \begin{bmatrix} \cos\psi & 0 & \sin\psi \\ 0 & 1 & 0 \\ -\sin\psi & 0 & \cos\psi \end{bmatrix}, R_z(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

When constructing the rotation matrix from this sequence, each character indicates the corresponding axis. For example, if the sequence is `"XYZ"`, then the `se3` object constructs the rotation matrix R by multiplying the rotation about x-axis with the rotation about the y-axis, and then multiplying that product with the rotation about the z-axis:

$$R = R_x(\phi) \cdot R_y(\psi) \cdot R_z(\theta)$$

Example: `se3([pi/2 pi/3 pi/4],"eul","ZYZ")` rotates a point by `pi/4` radians about the z-axis, then rotates the point by `pi/3` radians about the y-axis, and then rotates the point by `pi/2`

radians about the *z*-axis. This is equivalent to `se3(pi/2,"rotz") * se3(pi/3,"roty") * se3(pi/4,"rotz")`

Data Types: `string` | `char`

`quat` — Quaternion
N-by-4 matrix

Quaternion, specified as an *N*-by-4 matrix. *N* is the number of specified quaternions. Each row represents one quaternion of the form [*qw qx qy qz*], where *qw* is a scalar number.

If `quat` is an *N*-by-4 matrix, the resulting number of created `se3` objects is equal to *N*.

Note The `se3` object normalizes the input quaternions before converting the quaternions to a rotation matrix.

Example: `[0.7071 0.7071 0 0]`

Data Types: `single` | `double`

`axang` — Axis-angle rotation
N-by-4 matrix

Axis-angle rotation, specified as an *N*-by-4 matrix in the form [*x y z theta*]. *N* is the total number of axis-angle rotations. *x*, *y*, and *z* are vector components from the *x*-, *y*-, and *z*-axis, respectively. The vector defines the axis to rotate by the angle *theta*, in radians.

If `axang` is an *N*-by-4 matrix, the resulting number of created `se3` objects is equal to *N*.

Example: `[.2 .15 .25 pi/4]` rotates the axis, defined as `0.2` in the *x*-axis, `0.15` along the *y*-axis, and `0.25` along the *z*-axis, by `pi/4` radians.

Data Types: `single` | `double`

`angle` — Single-axis-angle rotation
N-by-*M* matrix

Single-axis-angle rotation, specified as an *N*-by-*M* matrix. Each element of the matrix is an angle, in radians, about the axis specified using the `axis` argument, and the `se3` object creates an `se3` object for each angle.

If `angle` is an *N*-by-*M* matrix, the resulting number of created `se3` objects is equal to *N*.

The rotation angle is counterclockwise positive when you look along the specified axis toward the origin.

Data Types: `single` | `double`

`axis` — Axis to rotate
`"rotx"` | `"roty"` | `"rotz"`

Axis to rotate, specified as one of these options:

- `"rotx"` — Rotate about the *x*-axis:

$$R_x(\phi) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\phi & \sin\phi \\ 0 & -\sin\phi & \cos\phi \end{bmatrix}$$

- `"roty"` — Rotate about the *y*-axis:

$$R_y(\psi) = \begin{bmatrix} \cos\psi & 0 & \sin\psi \\ 0 & 1 & 0 \\ -\sin\psi & 0 & \cos\psi \end{bmatrix}$$

- `"rotz"` — Rotate about the *z*-axis:

$$R_z(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Use the `angle` argument to specify how much to rotate about the specified axis.

Example: `Rx = se3(phi,"rotx");`

Example: `Ry = se3(psi,"roty");`

Example: `Rz = se3(theta,"rotz");`

Data Types: `string` | `char`

pose — 3-D compact pose
N-by-7 matrix

3-D compact pose, specified as an *N*-by-7 matrix, where *N* is the total number of compact poses. Each row is a pose, comprised of a xyz position and quaternion, in the form [*x y z qw qx qy qz*]. *x*, *y*, and *z* are the positions in the *x*-, *y*-, and *z*-axes, respectively. *qw*, *qx*, *qy*, and *qz* together are the quaternion rotation in *w*, *x*, *y*, and *z*, respectively.

If `pose` is an *N*-by-7 matrix, the resulting number of created `se3` objects is equal to *N*.

Data Types: `single` | `double`

Object Functions

Mathematical Operations

mtimes, *	Transformation or rotation multiplication
mrdivide, /	Transformation or rotation right division
rdivide, ./	Element-wise transformation or rotation right division
times, .*	Element-wise transformation or rotation multiplication

Utilities

interp	Interpolate between transformations
dist	Calculate distance between transformations
normalize	Normalize transformation or rotation matrix
transform	Apply rigid body transformation to points

Numerical Conversions

axang	Convert transformation or rotation into axis-angle rotations
eul	Convert transformation or rotation into Euler angles
rotm	Extract rotation matrix
quat	Convert transformation or rotation to numeric quaternion
quaternion	Create a quaternion array
trvec	Extract translation vector
tform	Extract homogeneous transformation
xyzquat	Convert transformation or rotation to compact 3-D pose representation

Object Conversions

so3	SO(3) rotation

Examples

Create SE(3) Transformation Using Euler Angles and Translation

Define an Euler-angle rotation of `[pi/2 0 pi/7]` with a `"XYZ"` rotation sequence, and a *xyz* translation of `[6 4 1]`.

```
angles = [pi/2 0 pi/7];
trvec = [6 4 1];
```

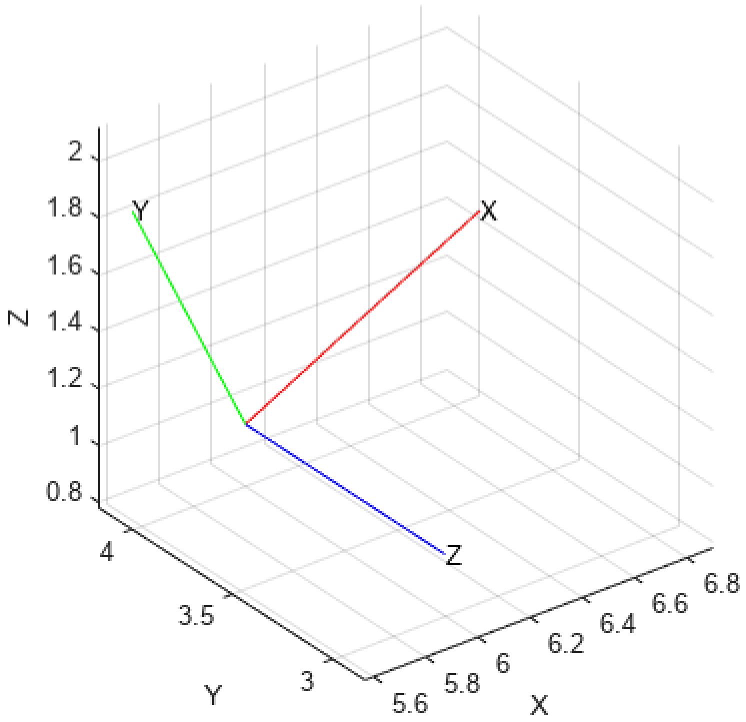
Create an SE(3) transformation using the Euler angles and the translation.

```
TF = se3(angles,"eul","XYZ",trvec)
```

```
TF = se3
    0.9010   -0.4339         0    6.0000
    0.0000    0.0000   -1.0000    4.0000
    0.4339    0.9010    0.0000    1.0000
         0         0         0    1.0000
```

Plot the transformation.

```
plotTransforms(TF,AxisLabels="on",FrameAxisLabels="on");
```



Algorithms

3-D Homogeneous Transformation Matrix

3-D homogeneous transformation matrices consist of both an SO(3) rotation and an *xyz*-translation.

To read more about SO(3) rotations, see the “3-D Orthonormal Rotation Matrix” on page 1-129 section of the `so3` object.

The translation is along the *x*-, *y*-, and *z*-axes as a three-element column vector:

$$t = \begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

The SO(3) rotation matrix R is applied to the translation vector t to create the homogeneous translation matrix T. The rotation matrix is present in the upper-left of the transformation matrix as 3-by-3 submatrix, and the translation vector is present as a three-element vector in the last column.

$$T = \begin{bmatrix} R & t \\ 0_{1x3} & 1 \end{bmatrix} = \begin{bmatrix} I_3 & t \\ 0_{1x3} & 1 \end{bmatrix} \cdot \begin{bmatrix} R & 0 \\ 0_{1x3} & 1 \end{bmatrix}$$

Version History

Introduced in R2022b

R2023a: New methods and syntaxes

`se3` supports new methods and syntaxes for converting to and from other transformations and rotations.

The new methods are:

- `axang`
- `eul`
- `so3`
- `quat`
- `quaternion`
- `xyzquat`

Extended Capabilities

C/C++ Code Generation

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

See Also

Functions

`axang2tform` | `eul2tform` | `quat2tform` | `rotm2tform` | `trvec2tform` | `plotTransforms`

Objects

`se2` | `so2` | `so3` | `quaternion`

se2

SE(2) homogeneous transformation

Description

The `se2` object represents an SE(2) transformation as a 2-D homogeneous transformation matrix consisting of a translation and rotation.

For more information, see the “2-D Homogeneous Transformation Matrix” on page 1-122 section.

This object acts like a numerical matrix, enabling you to compose poses using multiplication and division.

Creation

Syntax

```
transformation = se2
transformation = se2(rotation)
transformation = se2(rotation,translation)
transformation = se2(transformation)
```

```
transformation = se2(angle,"theta")
transformation = se2(angle,"theta",translation)
transformation = se2(translation,"trvec")
transformation = se2(pose,"xytheta")
```

Description

Rotation Matrices, Translation Vectors, and Transformation Matrices

`transformation = se2` creates an SE(2) transformation representing an identity rotation with no translation.

$$transformation = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

`transformation = se2(rotation)` creates an SE(2) transformation representing a pure rotation defined by the orthonormal rotation `rotation` with no translation. The rotation matrix is represented by the elements in the top left of the `transformation` matrix.

$$rotation = \begin{bmatrix} r_{11} & r_{12} \\ r_{21} & r_{22} \end{bmatrix}$$

$$transformation = \begin{bmatrix} r_{11} & r_{12} & 0 \\ r_{21} & r_{22} & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

`transformation = se2(rotation,translation)` creates an SE(2) transformation representing a rotation defined by the orthonormal rotation `rotation` and the translation `translation`. The function applies the rotation matrix first, then the translation vector, to create the transformation.

$$rotation = \begin{bmatrix} r_{11} & r_{12} \\ r_{21} & r_{22} \end{bmatrix}, translation = \begin{bmatrix} t_1 \\ t_2 \end{bmatrix}$$

$$transformation = \begin{bmatrix} r_{11} & r_{12} & t_1 \\ r_{21} & r_{22} & t_2 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & t_1 \\ 0 & 1 & t_2 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} r_{11} & r_{12} & 0 \\ r_{21} & r_{22} & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

`transformation = se2(transformation)` creates an SE(2) transformation representing a translation and rotation as defined by the homogeneous transformation `transformation`.

Other 2-D Rotations and Transformation Representations

`transformation = se2(angle,"theta")` creates SE(2) transformations `transformation` from rotations around the *z*-axis, in radians. The transformation contains zero translation.

`transformation = se2(angle,"theta",translation)` creates SE(2) transformations from rotations around the *z*-axis, in radians, with translations `translation`.

`transformation = se2(translation,"trvec")` creates an SE(2) transformation from the translation vector `translation`.

`transformation = se2(pose,"xytheta")` creates an SE(2) transformation from the 2-D compact pose `pose`.

Input Arguments

`rotation` — Orthonormal rotation
2-by-2 matrix | 2-by-2-by-*N* matrix | `so2` object | *N*-element array of `so2` objects

Orthonormal rotation, specified as a 2-by-2 matrix, a 2-by-2-by-*N* array, a scalar `so2` object, or an *N*-element array of `so2` objects. *N* is the total number of rotations.

If `rotation` contains more than one rotation and you also specify `translation` at construction, the number of translations in `translation` must be one or equal to the number of rotations in `rotation`. The resulting number of transformation objects is equal to the value of the `translation` or `rotation` argument, whichever is larger.

If `rotation` contains one rotation and you also specify `translation` as an *N*-by-2 matrix, then the resulting transformations contain the same rotation specified by `rotation` and the corresponding translation vector in `translation`. The resulting number of transformation objects is equal to the number of translations in `translation`.

Example: `eye(2)`

Data Types: `single` | `double`

`translation` — Translation
two-element row vector | *N*-by-2 matrix

Translation, specified as an *N*-by-2 matrix. *N* is the total number of translations and each translation is of the form [*x y*].

If `translation` contains more than one translation, the number of rotations in `rotation` must be one or equal to the number of translations in `translation`. The resulting number of created transformation objects is equal to the value of the `translation` or `rotation` argument, whichever is larger.

If you specify more than one translation but only one rotation, the resulting transformations contain the same rotation specified in `rotation` and the corresponding translation in `translation`. The resulting number of created `se2` objects is equal to the value of the `translation`.

Example: `[1 4]`

Data Types: `single` | `double`

`transformation` — Homogeneous transformation
3-by-3 matrix | 3-by-3-*N* array | `se2` object | *N*-element array of `se2` objects

Homogeneous transformation, specified as a 3-by-3 matrix, a 3-by-3-*N* array, a scalar `se3` object, or an *N*-element array of `se2` objects. *N* is the total number of transformations specified.

If `transformation` is an array, the resulting number of created `se2` objects is equal to *N*.

Example: `eye(3)`

Data Types: `single` | `double`

`angle` — *z*-axis rotation angle
N-by-*M* matrix

z-axis rotation angle, specified as an *N*-by-*M* matrix. Each element of the matrix is an angle, in radians, about the *z*-axis. The `se2` object creates an `se2` object for each angle.

If `angle` is an *N*-by-*M* matrix, the resulting number of created `se2` objects is equal to *N*.

The rotation angle is counterclockwise positive when you look along the specified axis toward the origin.

Data Types: `single` | `double`

`pose` — 2-D compact pose
N-by-3 matrix

3-D compact pose, specified as an *N*-by-3 matrix, where *N* is the total number of compact poses. Each row is a pose, comprised of an xy position and a rotation about the *z*-axis, in the form [*x y theta*]. *x*, *y* are the xy-positions and *theta* is the rotation about the *z*-axis.

If `pose` is an *N*-by-3 matrix, the resulting number of created `se2` objects is equal to *N*.

Data Types: `single` | `double`

Object Functions

Mathematical Operations

mtimes, *	Transformation or rotation multiplication
mrdivide, /	Transformation or rotation right division
rdivide, ./	Element-wise transformation or rotation right division

times, .*	Element-wise transformation or rotation multiplication

Utilities

interp	Interpolate between transformations
dist	Calculate distance between transformations
normalize	Normalize transformation or rotation matrix
transform	Apply rigid body transformation to points

Numerical Conversions

rotm	Extract rotation matrix
trvec	Extract translation vector
tform	Extract homogeneous transformation
theta	Convert transformation or rotation to 2-D rotation angle
xytheta	Convert transformation or rotation to compact 2-D pose representation

Object Conversions

so2	SO(2) rotation

Examples

Create SE(2) Transformation Using Angle and Translation

Define an angle rotation of `pi/4` and a *xyz* translation of `[6 4]`.

```
angle = pi/6;
trvec = [2 1];
```

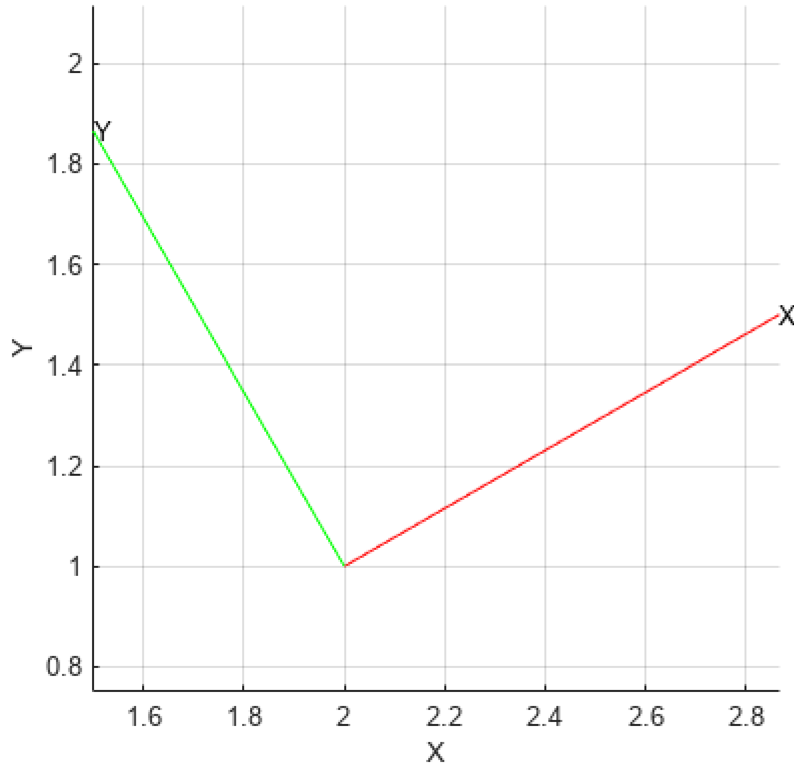
Create an SE(2) transformation using the angle and translation.

```
TF = se2(angle,"theta",trvec)
```

```
TF = se2
    0.8660   -0.5000    2.0000
    0.5000    0.8660    1.0000
         0         0    1.0000
```

Plot the transformation.

```
plotTransforms(TF,AxisLabels="on",FrameAxisLabels="on");
```



Algorithms

2-D Homogeneous Transformation Matrix

2-D homogeneous transformation matrices consist of both an SO(2) rotation and an *xy*-translation.

To read more about SO(2) rotations, see the "2-D Orthonormal Rotation Matrix" on page 1-133 section of the `so2` object.

The translation is along the *x*-, *y*-, and *z*-axes as a three-element column vector:

$$t = \begin{bmatrix} x \\ y \end{bmatrix}$$

The SO(2) rotation matrix R is applied to the translation vector t to create the homogeneous translation matrix T. The rotation matrix is present in the upper-left of the transformation matrix as 2-by-2 submatrix, and the translation vector is present as a two-element vector in the last column.

$$T = \begin{bmatrix} R & t \\ 0_{1 \times 2} & 1 \end{bmatrix} = \begin{bmatrix} I_2 & t \\ 0_{1 \times 2} & 1 \end{bmatrix} \cdot \begin{bmatrix} R & 0 \\ 0_{1 \times 2} & 1 \end{bmatrix}$$

Version History

Introduced in R2022b

R2023a: New methods and syntaxes

se2 supports new methods and syntaxes for converting to and from other transformations and rotations.

The new methods are:

- se3
- so2
- theta
- xytheta

Extended Capabilities

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

See Also

Functions
axang2tform | eul2tform | quat2tform | rotm2tform | trvec2tform | plotTransforms

Objects
se3 | so2 | so3 | quaternion

so3

SO(3) rotation

Description

The `so3` object represents an SO(3) rotation in 3-D in a right-handed Cartesian coordinate system.

The SO(3) rotation is a 3-by-3 orthonormal rotation matrix. For example, these are orthonormal rotation matrices for rotations of ϕ, ψ, and θ about the *x*-, *y*-, and *z*-axis, respectively:

$$R_x(\phi) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\phi & \sin\phi \\ 0 & -\sin\phi & \cos\phi \end{bmatrix}, R_y(\psi) = \begin{bmatrix} \cos\psi & 0 & \sin\psi \\ 0 & 1 & 0 \\ -\sin\psi & 0 & \cos\psi \end{bmatrix}, R_z(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

For more information, see the 3-D Orthonormal Rotation Matrix section.

This object acts like a numerical matrix, enabling you to compose rotations using multiplication and division.

Creation

Syntax

```
rotation = so3
rotation = so3(rotation)
rotation = so3(quaternion)
rotation = so3(transformation)

rotation = so3(euler,"eul")
rotation = so3(euler,"eul",sequence)
rotation = so3(quat,"quat")
rotation = so3(axang,"axang")
rotation = so3(angle,axis)
```

Description

3-D Rotation Representations

`rotation = so3` creates an SO(3) rotation representing an identity rotation with no translation.

$$rotation = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

`rotation = so3(rotation)` creates an SO(3) rotation representing a pure rotation defined by the orthonormal rotation `rotation`.

$$rotation = \begin{bmatrix} r_{11} & r_{12} & r_{13} \\ r_{11} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{bmatrix}$$

`rotation = so3(quaternion)` creates an SO(3) rotation from the rotations defined by the quaternion `quaternion`.

`rotation = so3(transformation)` creates an SO(3) rotation from the SE(3) transformation `transformation`.

Other Numeric 3-D Rotation Representations

`rotation = so3(euler,"eul")` creates an SO(3) rotation from the rotations defined by the Euler angles `euler`.

`rotation = so3(euler,"eul",sequence)` specifies the sequence of the Euler angle rotations `sequence`. For example, the sequence `"ZYX"` rotates the *z*-axis, then the *y*-axis and *x*-axis.

`rotation = so3(quat,"quat")` creates an SO(3) rotation from the rotations defined by the numeric quaternions `quat`.

`rotation = so3(axang,"axang")` creates an SO(3) rotation from the rotations defined by the axis-angle rotation `axang`.

`rotation = so3(angle,axis)` creates an SO(3) rotation from the rotations `angles` about the rotation axis `axis`.

Note If any inputs contain more than one rotation, then the output `rotation` is an *N*-element array of `so3` objects corresponding to each of the *N* input rotations.

Input Arguments

`rotation` — Orthonormal rotation

3-by-3 matrix | 3-by-3-by-*N* matrix | `so3` object | *N*-element array of `so3` objects

Orthonormal rotation, specified as a 3-by-3 matrix, a 3-by-3-by*N* array, a scalar `so3` object, or an *N*-element array of `so3` objects. *N* is the total number of rotations.

If `rotation` is an array, the resulting number of created `so3` objects in the output array is equal to *N*.

Example: `eye(3)`

`transformation` — Homogeneous transformation

`se3` object | *N*-element array of `se3` objects

Homogeneous transformation, specified as an `se3` object or a *N*-element array of `se3` objects. *N* is the total number of transformations specified.

The output `so3` object contains only the rotational submatrix of the `se3` object.

If `transformation` is an array, the resulting number of created `so3` objects in the output array is equal to *N*.

Example: `se3(pi/4,"rotx")`

`quaternion` — Quaternion
`quaternion` object | *N*-element array of `quaternion` objects

Quaternion, specified as a scalar `quaternion` object or as an *N*-element array of `quaternion` objects. *N* is the total number of specified quaternions.

If `quaternion` is an *N*-element array, the resulting number of created `so3` objects is equal to *N*.

Example: `quaternion(1,0.2,0.4,0.2)`

`euler` — Euler angles
N-by-3 matrix

Euler angles, specified as an *N*-by-3 matrix, in radians. Each row represents one set of Euler angles with the axis-rotation sequence defined by the `sequence` argument. The default axis-rotation sequence is *ZYX*.

If `euler` is an *N*-by-3 matrix, the resulting number of created `so3` objects is equal to *N*.

Example: `[pi/2 pi pi/4]`

Data Types: `single` | `double`

`sequence` — Axis-rotation sequence
`"ZYX"` (default) | `"ZYZ"` | `"ZXY"` | `"ZXZ"` | `"YXY"` | `"YZX"` | `"YXZ"` | `"YZY"` | `"XYX"` | `"XYZ"` | `"XZX"` | `"XZY"`

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

- `"ZYX"` (default)
- `"ZYZ"`
- `"ZXY"`
- `"ZXZ"`
- `"YXY"`
- `"YZX"`
- `"YXZ"`
- `"YZY"`
- `"XYX"`
- `"XYZ"`
- `"XZX"`
- `"XZY"`

These are orthonormal rotation matrices for rotations of ϕ, ψ, and θ about the *x*-, *y*-, and *z*-axis, respectively:

$$R_x(\phi) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\phi & \sin\phi \\ 0 & -\sin\phi & \cos\phi \end{bmatrix}, R_y(\psi) = \begin{bmatrix} \cos\psi & 0 & \sin\psi \\ 0 & 1 & 0 \\ -\sin\psi & 0 & \cos\psi \end{bmatrix}, R_z(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

When constructing the rotation matrix from this sequence, each character indicates the corresponding axis. For example, if the sequence is `"XYZ"`, then the `so3` object constructs the

rotation matrix R by multiplying the rotation about x-axis with the rotation about the y-axis, and then multiplying that product with the rotation about the z-axis:

$$R = R_x(\phi) \cdot R_y(\psi) \cdot R_z(\theta)$$

Example: `so3([pi/2 pi/3 pi/4],"eul","ZYZ")` rotates a point by `pi/4` radians about the z-axis, then rotates the point by `pi/3` radians about the y-axis, and then rotates the point by `pi/2` radians about the z-axis. This is equivalent to `so3(pi/2,"rotz") * so3(pi/3,"roty") * so3(pi/4,"rotz")`

Data Types: `string` | `char`

`quat` — Quaternion
N-by-4

Quaternion, specified as an *N*-by-4 matrix. *N* is the number of specified quaternions. Each row represents one quaternion of the form [*qw qx qy qz*], where *qw* is a scalar number.

If `quat` is an *N*-by-4 matrix, the resulting number of created `so3` objects is equal to *N*.

Note The `so3` object normalizes the input quaternions before converting the quaternions to a rotation matrix.

Example: `[0.7071 0.7071 0 0]`

Data Types: `single` | `double`

`axang` — Axis-angle rotation
N-by-4 matrix

Axis-angle rotation, specified as an *N*-by-4 matrix in the form [*x y z theta*]. *N* is the total number of axis-angle rotations. *x*, *y*, and *z* are vector components from the *x*-, *y*-, and *z*-axis, respectively. The vector defines the axis to rotate by the angle *theta*, in radians.

If `axang` is an *N*-by-4 matrix, the resulting number of created `so3` objects is equal to *N*.

Example: `[.2 .15 .25 pi/4]` rotates the axis, defined as `0.2` in the *x*-axis, `0.15` along the *y*-axis, and `0.25` along the *z*-axis, by `pi/4` radians.

Data Types: `single` | `double`

`angle` — Single-axis-angle rotation
N-by-*M* matrix

Single-axis-angle rotation, specified as an *N*-by-*M* matrix. Each element of the matrix is an angle, in radians, about the axis specified using the `axis` argument, and the `so3` object creates an `so3` object for each angle.

If `angle` is an *N*-by-*M* matrix, the resulting number of created `so3` objects is equal to *N*.

The rotation angle is counterclockwise positive when you look along the specified axis toward the origin.

Data Types: `single` | `double`

`axis` — Axis to rotate
`"rotx"` | `"roty"` | `"rotz"`

Axis to rotate, specified as one of these options:

- `"rotx"` — Rotate about the *x*-axis:

$$R_x(\phi) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\phi & \sin\phi \\ 0 & -\sin\phi & \cos\phi \end{bmatrix}$$

- `"roty"` — Rotate about the *y*-axis:

$$R_y(\psi) = \begin{bmatrix} \cos\psi & 0 & \sin\psi \\ 0 & 1 & 0 \\ -\sin\psi & 0 & \cos\psi \end{bmatrix}$$

- `"rotz"` — Rotate about the *z*-axis:

$$R_z(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Use the `angle` argument to specify how much to rotate about the specified axis.

Example: `Rx = so3(phi,"rotx");`

Example: `Ry = so3(psi,"roty");`

Example: `Rz = so3(theta,"rotz");`

Data Types: `string` | `char`

Object Functions

Mathematical Operations

mtimes, *	Transformation or rotation multiplication
mrdivide, /	Transformation or rotation right division
rdivide, ./	Element-wise transformation or rotation right division
times, .*	Element-wise transformation or rotation multiplication

Utilities

interp	Interpolate between transformations
dist	Calculate distance between transformations
normalize	Normalize transformation or rotation matrix
transform	Apply rigid body transformation to points

Numerical Conversions

axang	Convert transformation or rotation into axis-angle rotations
eul	Convert transformation or rotation into Euler angles
rotm	Extract rotation matrix
quat	Convert transformation or rotation to numeric quaternion
trvec	Extract translation vector

tform	Extract homogeneous transformation
xyzquat	Convert transformation or rotation to compact 3-D pose representation

Object Conversions

se3	SE(3) homogeneous transformation
quaternion	Create a quaternion array

Examples

Convert SO(3) Rotation to Euler Angles

Create SO(3) rotation defined by a Euler angles.

```
eul1 = [pi/4 pi/3 pi/8]
```

```
eul1 = 1×3

    0.7854    1.0472    0.3927
```

```
R = so3(eul1,"eul")
```

```
R = so3
    0.3536   -0.4189    0.8364
    0.3536    0.8876    0.2952
   -0.8660    0.1913    0.4619
```

Get the Euler angles from the transformation.

```
eul2 = eul(R)
```

```
eul2 = 1×3

    0.7854    1.0472    0.3927
```

Algorithms

3-D Orthonormal Rotation Matrix

SO(3) rotation matrices are 3-by-3 orthonormal matrices that represent any rotation in 3-D Euclidean space. SO(3) rotations have many special properties. For example, SO(3) rotation matrices are in the 3-D special orthogonal group, so the product of two SO(3) rotation matrices is an SO(3) rotation matrix. This enables you to compose rotations from multiple rotations. For example, these are orthonormal rotation matrices for rotations of ϕ, ψ, and θ about the *x*-, *y*-, and *z*-axis, respectively:

$$R_x(\phi) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\phi & \sin\phi \\ 0 & -\sin\phi & \cos\phi \end{bmatrix}, R_y(\psi) = \begin{bmatrix} \cos\psi & 0 & \sin\psi \\ 0 & 1 & 0 \\ -\sin\psi & 0 & \cos\psi \end{bmatrix}, R_z(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Depending on the order in which you multiply these *x*-, *y*-, and *z*-axis rotations, you can construct compound matrices that represent any rotation in 3-D Euclidean space.

These are other properties of SO(3) rotations:

- Each column is both orthogonal and a unit vector, meaning that none of the columns are multiples of each other.
- The determinant of the matrix is positive 1.
- The inverse of the matrix is the same as the transpose of the matrix: $R^{-1} = R^T$.

Version History

Introduced in R2022b

R2023a: New methods and syntaxes

`so3` supports new methods and syntaxes for converting to and from other transformations and rotations.

The new methods are:

- `axang`
- `eul`
- `quat`
- `quaternion`
- `tform`
- `trvec`
- `xyzquat`

Extended Capabilities

C/C++ Code Generation

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

See Also

Functions

`axang2rotm` | `eul2rotm` | `quat2rotm` | `tform2rotm`

Objects

`se2` | `se3` | `so2` | `quaternion`

so2

SO(2) rotation

Description

The `so2` object represents an SO(2) rotation in 2-D.

For more information, see the "2-D Orthonormal Rotation Matrix" on page 1-133 section.

This object acts like a numerical matrix, enabling you to compose rotations using multiplication and division.

Creation

Syntax

```
rotation = so2
rotation = so2(rotation)
rotation = so2(transformation)
rotation = so2(angle,"theta")
```

Description

`rotation = so2` creates an SO(2) rotation representing an identity rotation with no translation.

$$rotation = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

`rotation = so2(rotation)` creates an SO(2) rotation `rotation` representing a pure rotation defined by the orthonormal rotation `rotation`.

$$rotation = \begin{bmatrix} r_{11} & r_{12} \\ r_{21} & r_{22} \end{bmatrix}$$

`rotation = so2(transformation)` creates an SO(2) rotation from the SE(2) transformation `transformation`.

`rotation = so2(angle,"theta")` creates an SO(2) rotation `rotation` from a rotation angle about the z-axis `angle`.

Note If any inputs contain more than one rotation, the output `rotation` is an N-element array of `so2` objects corresponding to each of the N input rotations.

Input Arguments

`rotation` — Orthonormal rotation
2-by-2 matrix | 2-by-2-by-*N* matrix | `so2` object | *N*-element array of `so2` objects

Orthonormal rotation, specified as a 2-by-2 matrix, a 3-by-3-by-*N* array, a scalar `so2` object, or an *N*-element array of `so2` objects. *N* is the total number of rotations.

If `rotation` is an array, the resulting number of created `so2` objects in the output array is equal to *N*.

Example: `eye(3)`

Data Types: `single` | `double`

`transformation` — Homogeneous transformation
`se2` object | *N*-element array of `se2` objects

Homogeneous transformation, specified as an `se2` object or an *N*-element array of `se2` objects. *N* is the total number of transformations specified.

The output `so2` object contains only the rotational submatrix of the `se2` object.

If `transformation` is an array, the resulting number of created `so2` objects in the output array is equal to *N*.

Example: `se2([1 2],"trvec")`

`angle` — *z*-axis rotation angle
N-by-*M* matrix

z-axis rotation angle, specified as an *N*-by-*M* matrix. Each element of the matrix is an angle, in radians, about the *z*-axis. The `so2` object creates an `so2` object for each angle.

If `angle` is an *N*-by-*M* matrix, the resulting number of created `so2` objects is equal to *N*.

The rotation angle is counterclockwise positive when you look along the specified axis toward the origin.

Data Types: `single` | `double`

Object Functions

Mathematical Operations

mtimes, *	Transformation or rotation multiplication
mrdivide, /	Transformation or rotation right division
rdivide, ./	Element-wise transformation or rotation right division
times, .*	Element-wise transformation or rotation multiplication

Utilities

interp	Interpolate between transformations
dist	Calculate distance between transformations
normalize	Normalize transformation or rotation matrix
transform	Apply rigid body transformation to points

Numerical Conversions

rotm	Extract rotation matrix
trvec	Extract translation vector
tform	Extract homogeneous transformation
theta	Convert transformation or rotation to 2-D rotation angle
xytheta	Convert transformation or rotation to compact 2-D pose representation

Object Conversions

so3	SO(3) rotation

Examples

Create SO(2) Rotation Using Angle

Define an angle rotation of `pi/4` and a *xy* translation of `[6 4]`.

```
angle = pi/4;
```

Create an SO(2) rotation using the angle.

```
R = so2(angle,"theta")
```

```
R = so2
    0.7071   -0.7071
    0.7071    0.7071
```

Algorithms

2-D Orthonormal Rotation Matrix

SO(2) rotation matrices are 2-by-2 orthonormal matrices that represent a rotation about a single axis 2-D Euclidean space. SO(2) rotations have many special properties. For example, SO(2) rotation matrices are in the 2-D special orthogonal group, so the product of two SO(2) rotation matrices is an SO(2) rotation matrix. This enables you to compose rotations from multiple rotations.

This is a 2-D orthonormal rotation matrix that describes describe a rotation θ about the *z*-axis:

$$R_z(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}$$

These are other properties of SO(2) rotations:

- Each column is both orthogonal and a unit vector, meaning that none of the columns are multiples of each other.
- The determinant of the matrix is positive 1.
- The inverse of the matrix is the same as the transpose of the matrix: $R^{-1} = R^T$.

Version History

Introduced in R2022b

R2023a: New methods and syntaxes

`so2` supports new methods and syntaxes for converting to and from other transformations and rotations.

The new methods are:

- `so3`
- `theta`
- `tform`
- `trvec`
- `xytheta`

Extended Capabilities

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

See Also

Functions
`axang2rotm` | `eul2rotm` | `quat2rotm` | `tform2rotm`

Objects
`se2` | `se3` | `so3` | `quaternion`

sim3d.Editor

Interface to the Unreal Engine project

Description

Use the `sim3d.Editor` class to interface with the Unreal® Editor.

To develop scenes with the Unreal Editor and co-simulate with Simulink®, you need the UAV Toolbox Interface for Unreal Engine Projects support package. The support package contains an Unreal Engine project that allows you to customize the UAV Toolbox scenes. For information about the support package, see "Customize Unreal Engine Scenes for UAVs".

Creation

Syntax

`sim3d.Editor(project)`

Description

MATLAB creates an `sim3d.Editor` object for the Unreal Editor project specified in `sim3d.Editor( project)`.

Input Arguments

`project` — Project path and name
string array

Project path and name.

Example: `"C:\Local\AutoVrtlEnv\AutoVrtlEnv.uproject"`

Data Types: `string`

Properties

`Uproject` — Project path and name
string array

This property is read-only.

Project path and name with Unreal Engine project file extension.

Example: `"C:\Local\AutoVrtlEnv\AutoVrtlEnv.uproject"`

Data Types: `string`

Object Functions

open   Open the Unreal Editor

Examples

Open Project in Unreal Editor

Open an Unreal Engine project in the Unreal Editor.

Create an instance of the `sim3d.Editor` class for the Unreal Engine project located in `C:\Local\AutoVrtlEnv\AutoVrtlEnv.uproject`.

```
editor = sim3d.Editor(fullfile("C:\Local\AutoVrtlEnv\AutoVrtlEnv.uproject"))
```

Open the project in the Unreal Editor.

```
editor.open();
```

Version History

Introduced in R2020b

See Also

Topics

"Customize Unreal Engine Scenes for UAVs"

transformTree

Define coordinate frames and relative transformations

Description

The `transformTree` object contains an organized tree structure for coordinate frames and their relative transformations over time. The object stores the relative transformations between children frames and their parents. You can specify a timestamped transform for frames and query the relative transformations between different frames in the tree. The object interpolates intermediate timestamps using a constant velocity assumption for linear motion, and spherical linear interpolation (SLERP) for angular motion. Otherwise, the relative transformations are kept constant past the range of the timestamps specified. Times prior to the first timestamp return `NaN`.

Use the `updateTransform` function to add timestamps to the tree by defining the parent-to-child relationships. Query specific transformations at given timestamps using `getTransform` and display the frame relationships using `show`.

Creation

Syntax

```
frames = transformTree
frames = transformTree(baseName)
frames = transformTree(baseName,numFrames)
frames = transformTree(baseName,numFrames,numTransforms)
frames = transformTree(baseName,numFrames,numTransforms,rootTime)
```

Description

`frames = transformTree` creates a transformation tree data structure with a single frame, `"root"`, with the maximum number of frames and timestamped transforms per frame, set to 10.

`frames = transformTree(baseName)` specifies the name of the root frame as a string or character vector.

`frames = transformTree(baseName,numFrames)` additionally sets the MaxNumFrames property, which defines the max number of named frames in the object.

`frames = transformTree(baseName,numFrames,numTransforms)` additionally sets the MaxNumTransforms property, which defines the max number of timestamped transforms per frame name.

`frames = transformTree(baseName,numFrames,numTransforms,rootTime)` additionally specifies the timestamp of the initial `baseName` frame as a scalar time in seconds.

Properties

`MaxNumFrames` — Maximum number of frames in tree

10 (default) | positive integer

Maximum number of frames in the tree, specified as a positive integer. Each frame has associated timestamped transforms that define the state of the frame at those specific times.

Data Types: `double`

`MaxNumTransforms` — Maximum number of timestamped transforms per frame

10 (default) | positive integer

Maximum number of timestamped transforms per frame, specified as a positive integer. This property sets an upper limit on the number of timestamped transforms the object can store for each frame named in the structure. A `transformTree` object with MaxNumFrames and `MaxNumTransforms` set to 10 can store a maximum of 100 transformations with 10 for each frame.

Data Types: `double`

`NumFrames` — Current number of coordinate frames stored

1 (default) | positive integer

Current number of coordinate frames stored, specified as a positive integer. The object starts with a single root frame, and new frames and specific timestamps are added using `updateTransform` function.

Data Types: `double`

Object Functions

getGraph	Graph object representing tree structure
getTransform	Get relative transform between frames
info	List all frame names and stored timestamps
removeTransform	Remove frame transform relative to its parent
show	Show transform tree
updateTransform	Update frame transform relative to its parent

Version History

Introduced in R2020b

See Also

Objects

`uavScenario` | `fixedwing` | `multirotor` | `uavDubinsPathSegment`

Functions

`getGraph` | `getTransform` | `info` | `removeTransform` | `show` | `updateTransform`

uavDubinsConnection

Dubins path connection for UAV

Description

The `uavDubinsConnection` object holds information for computing a `uavDubinsPathSegment` path segment to connect start and goal poses of a UAV.

A UAV Dubins path segment connects two poses as a sequence of motions in the north-east-down coordinate system.

The motion options are:

- Straight
- Left turn (counterclockwise)
- Right turn (clockwise)
- Helix left turn (counterclockwise)
- Helix right turn (clockwise)
- No motion

The turn direction is defined as viewed from the top of the UAV. Helical motions are used to ascend or descend.

Use this connection object to define parameters for a UAV 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

```
connectionObj = uavDubinsConnection
connectionObj = uavDubinsConnection(Name,Value)
```

Description

`connectionObj = uavDubinsConnection` creates an object using default property values.

`connectionObj = uavDubinsConnection(Name,Value)` specifies property values using name-value pairs. To set multiple properties, specify multiple name-value pairs.

Properties

`AirSpeed` — Airspeed of UAV
`10` (default) | positive numeric scalar

Airspeed of the UAV, specified as a positive numeric scalar in m/s.

Data Types: `double`

`MaxRollAngle` — Maximum roll angle

`0.5` (default) | positive numeric scalar

Maximum roll angle to make the UAV turn left or right, specified as a positive numeric scalar in radians.

Note The minimum and maximum values for `MaxRollAngle` are greater than `0` and less than `pi/2`, respectively.

Data Types: `double`

`FlightPathAngleLimit` — Minimum and maximum flight path angles

`[-0.5 0.5]` (default) | two-element numeric vector

Flight path angle limits, specified as a two-element numeric vector `[min max]` in radians.

min is the minimum flight path angle the UAV takes to lose altitude, and *max* is the maximum flight path angle to gain altitude.

Note The minimum and maximum values for `FlightPathAngleLimit` are greater than `-pi/2` and less than `pi/2`, respectively.

Data Types: `double`

`DisabledPathTypes` — Path types to disable

`{}` (default) | cell array of four-element character vectors | vector of four-element string scalars

UAV Dubins path types to disable, specified as a cell array of four-element character vectors or vector of string scalars. The cell array defines the four prohibited sequences of motions.

Motion Type	Description
`"S"`	Straight
`"L"`	Left turn (counterclockwise)
`"R"`	Right turn (clockwise)
`"Hl"`	Helix left turn (counterclockwise)
`"Hr"`	Helix right turn (clockwise)
`"N"`	No motion

Note The no motion segment `"N"` is used as a filler at the end when only three path segments are needed.

To see all available path types, see the AllPathTypes property.

Example: `{'RLRN'}`

Data Types: `string` | `cell`

`MinTurningRadius` — Minimum turning radius
positive numeric scalar

This property is read-only.

Minimum turning radius of the UAV, specified as a positive numeric scalar in meters. This value corresponds to the radius of the circle at the maximum roll angle and a constant airspeed of the UAV.

Data Types: `double`

`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 the DisabledPathTypes property.

For UAV Dubins connections, the available path types are: `{'LSLN'}` `{'LSRN'}` `{'RSLN'}` `{'RSRN'}` `{'RLRN'}` `{'LRLN'}` `{'HlLSL'}` `{'HlLSR'}` `{'HrRSL'}` `{'HrRSR'}` `{'HrRLR'}` `{'HlLRL'}` `{'LSLHl'}` `{'LSRHr'}` `{'RSLHl'}` `{'RSRHr'}` `{'RLRHr'}` `{'LRLHl'}` `{'LRSL'}` `{'LRSR'}` `{'LRLR'}` `{'RLSR'}` `{'RLRL'}` `{'RLSL'}` `{'LSRL'}` `{'RSRL'}` `{'LSLR'}` `{'RSLR'}`.

Data Types: `cell`

Object Functions

connect	Connect poses with UAV Dubins connection path

Examples

Connect Poses Using UAV Dubins Connection Path

This example shows how to calculate a UAV Dubins path segment and connect poses using the `uavDubinsConnection` object.

Create a `uavDubinsConnection` object.

```
connectionObj = uavDubinsConnection;
```

Define start and goal poses as [`x`, `y`, `z`, `headingAngle`] vectors.

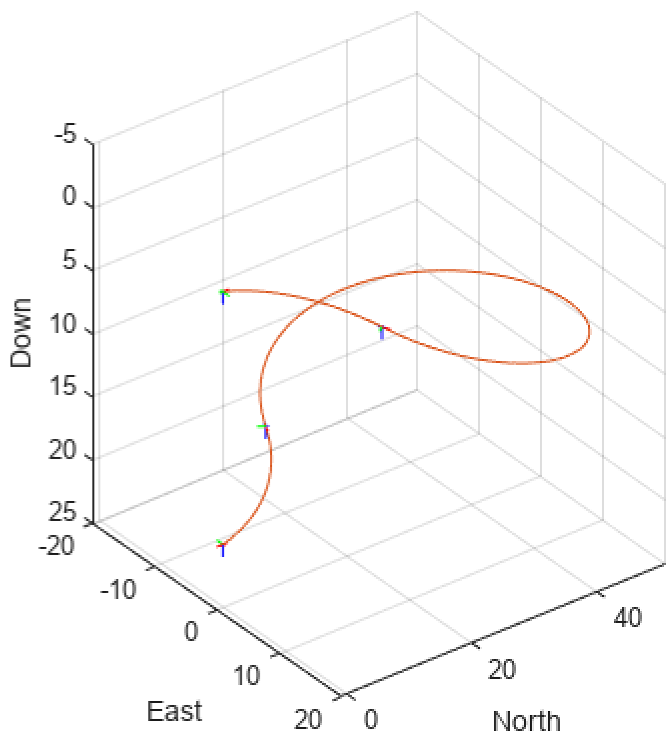
```
startPose = [0 0 0 0]; % [meters, meters, meters, radians]
goalPose = [0 0 20 pi];
```

Calculate a valid path segment and connect the poses. Returns a path segment object with the lowest path cost.

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

Show the generated path.

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



Display the motion type and the path cost of the generated path.

```
fprintf('Motion Type: %s\nPath Cost: %f\n',strjoin(pathSegObj{1}.MotionTypes),pathCosts);
```

```
Motion Type: R L R N
Path Cost: 138.373157
```

Modify Connection Types for UAV Dubins Connection Path

This example shows how to modify an existing `uavDubinsPathSegment`object.

Connect Poses Using UAV Dubins Connection Path

Create a `uavDubinsConnection` object.

```
connectionObj = uavDubinsConnection;
```

Define start and goal poses as [`x`, `y`, `z`, `headingAngle`] vectors.

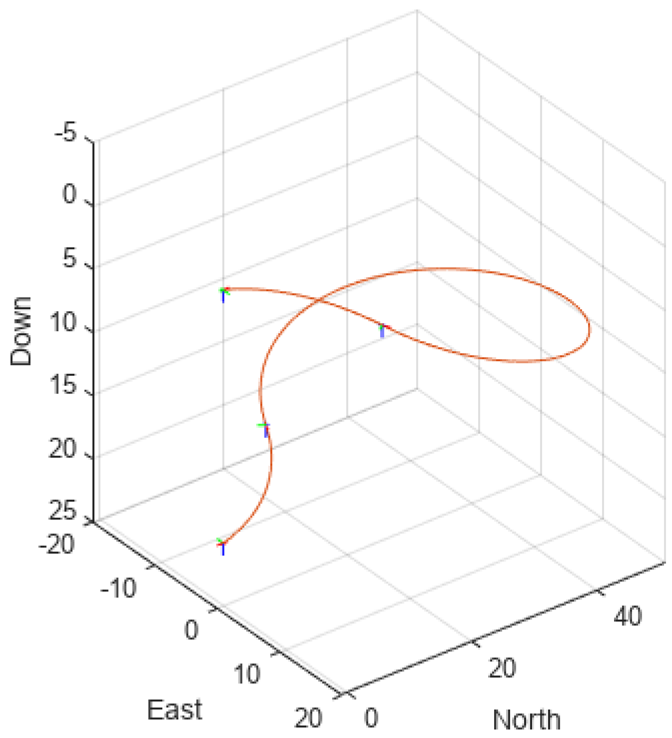
```
startPose = [0 0 0 0]; % [meters, meters, meters, radians]
goalPose = [0 0 20 pi];
```

Calculate a valid path segment and connect the poses. Returns a path segment object with the lowest path cost.

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

Show the generated path.

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



Verify the motion type and the path cost of the returned path segment.

```
fprintf('Motion Type: %s\nPath Cost: %f\n',strjoin(pathSegObj{1}.MotionTypes),pathCosts);
```

```
Motion Type: R L R N
Path Cost: 138.373157
```

Modify Connection Type and Properties

Disable this specific motion sequence in a new connection object. Specify the `AirSpeed`, `MaxRollAngle`, and `FlightPathAngleLimit` properties of the connection object.

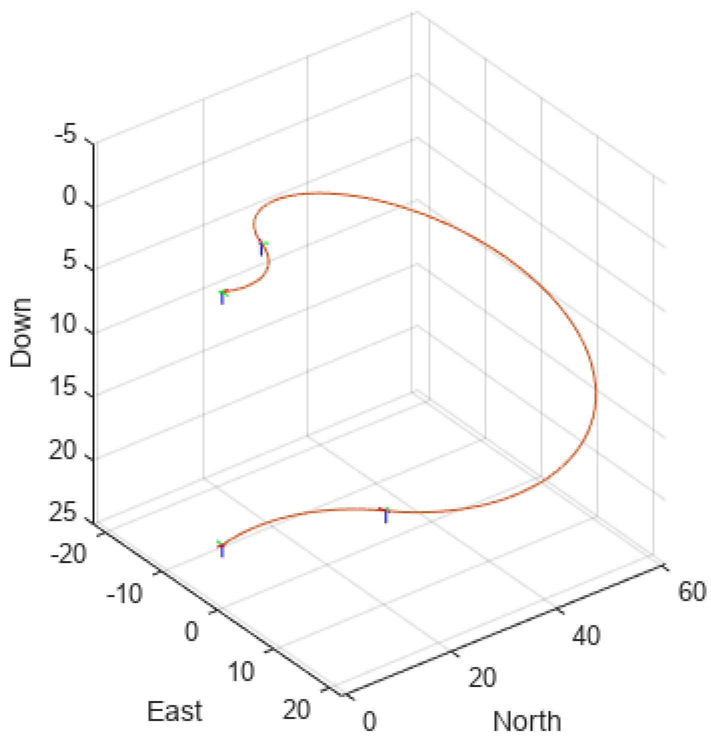
```
connectionObj = uavDubinsConnection('DisabledPathTypes',{'RLRN'});
connectionObj.AirSpeed = 15;
connectionObj.MaxRollAngle = 0.8;
connectionObj.FlightPathAngleLimit = [-1.47 1.47];
```

Connect the poses again to get a different path. Returns a path segment object with the next lowest path cost.

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

Show the modified path.

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



Verify the motion type and the path cost of the modified path segment.

```
fprintf('Motion Type: %s\nPath Cost: %f\n',strjoin(pathSegObj{1}.MotionTypes),pathCosts);
```

```
Motion Type: L R L N
Path Cost: 164.674067
```

Version History

Introduced in R2019b

References

[1] Owen, Mark, Randal W. Beard, and Timothy W. McLain. "Implementing Dubins Airplane Paths on Fixed-Wing UAVs." *Handbook of Unmanned Aerial Vehicles*, 2015, pp. 1677–1701.

Extended Capabilities

C/C++ Code Generation

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

See Also

`uavDubinsPathSegment`

uavDubinsPathSegment

Dubins path segment connecting two poses of UAV

Description

The `uavDubinsPathSegment` object holds information for a Dubins path segment that connects start and goal poses of a UAV as a sequence of motions in the north-east-down coordinate system.

The motion options are:

- Straight
- Left turn (counterclockwise)
- Right turn (clockwise)
- Helix left turn (counterclockwise)
- Helix right turn (clockwise)
- No motion

The turn direction is defined as viewed from the top of the UAV. Helical motions are used to ascend or descend.

Creation

Syntax

```
pathSegObj = connect(connectionObj,start,goal)
```

```
pathSegObj = uavDubinsPathSegment(connectionObj,start,goal)
pathSegObj = uavDubinsPathSegment(connectionObj,start,goal,motionTypes)
pathSegObj = uavDubinsPathSegment(start,goal,flightPathAngle,airSpeed,
minTurningRadius,helixRadius,motionTypes,motionLengths)
```

Description

To generate a `uavDubinsPathSegment` object, use the `connect` function with a `uavDubinsConnection` object:

`pathSegObj = connect(connectionObj,start,goal)` connects the start and goal poses using the specified `uavDubinsConnection` object. The `start` and `goal` inputs set the value of the properties StartPose and GoalPose, respectively.

To specifically define a path segment:

`pathSegObj = uavDubinsPathSegment(connectionObj,start,goal)` creates a Dubins path segment to connect start and goal poses of a UAV. The `uavDubinsConnection` object provides the minimum turning radius and flight path angle. It internally computes the optimal path and assigns it to `pathSegObj`.

`pathSegObj = uavDubinsPathSegment(connectionObj,start,goal,motionTypes)` creates a Dubins path segment to connect start and goal poses of a UAV with the given `motionTypes`. The `motionTypes` input sets the value of the MotionTypes property.

`pathSegObj = uavDubinsPathSegment(start,goal,flightPathAngle,airSpeed, minTurningRadius,helixRadius,motionTypes,motionLengths)` creates a Dubins path segment to connect start and goal poses of a UAV by explicitly specifying all the parameters. The input values are set to their corresponding properties in the object.

Properties

`StartPose` — Initial pose of UAV
four-element numeric vector

This property is read-only.

Initial pose of the UAV at the start of the path segment, specified as a four-element numeric vector [*x*, *y*, *z*, *headingAngle*].

x, *y*, and *z* specify the position in meters. *headingAngle* specifies the heading angle in radians.

Data Types: `double`

`GoalPose` — Goal pose of UAV
four-element numeric vector

This property is read-only.

Goal pose of the UAV at the end of the path segment, specified as a four-element numeric vector [*x*, *y*, *z*, *headingAngle*].

x, *y*, and *z* specify the position in meters. *headingAngle* specifies the heading angle in radians.

Data Types: `double`

`MinTurningRadius` — Minimum turning radius
positive numeric scalar

This property is read-only.

Minimum turning radius of the UAV, specified as a positive numeric scalar in meters. This value corresponds to the radius of the circle at the maximum roll angle and a constant airspeed of the UAV.

Data Types: `double`

`HelixRadius` — Helical path radius
positive numeric scalar

This property is read-only.

Helical path radius of the UAV, specified as a positive numeric scalar in meters.

Data Types: `double`

`FlightPathAngle` — Flight path angle
positive numeric scalar

This property is read-only.

Flight path angle of the UAV to reach the goal altitude, specified as a positive numeric scalar in radians.

Data Types: `double`

`AirSpeed` — Airspeed of UAV
positive numeric scalar

This property is read-only.

Airspeed of the UAV, specified as a positive numeric scalar in m/s.

Data Types: `double`

`MotionLengths` — Length of each motion
four-element numeric vector

This property is read-only.

Length of each motion in the path segment, specified as a four-element numeric vector in meters. Each motion length corresponds to a motion type specified in the MotionTypes property.

Data Types: `double`

`MotionTypes` — Type of each motion
four-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 (counterclockwise)
`"R"`	Right turn (clockwise)
`"Hl"`	Helix left turn (counterclockwise)
`"Hr"`	Helix right turn (clockwise)
`"N"`	No motion

Note The no motion segment `"N"` is used as a filler at the end when only three path segments are needed.

Each motion type corresponds to a motion length specified in the MotionLengths property.

For UAV Dubins connections, the available path types are: `{'LSLN'} {'LSRN'} {'RSLN'} {'RSRN'} {'RLRN'} {'LRLN'} {'HlLSL'} {'HlLSR'} {'HrRSL'} {'HrRSR'} {'HrRLR'} {'HlLRL'} {'LSLHl'} {'LSRHr'} {'RSLHl'} {'RSRHr'} {'RLRHr'} {'LRLHl'} {'LRSL'} {'LRSR'} {'LRLR'} {'RLSR'} {'RLRL'} {'RLSL'} {'LSRL'} {'RSRL'} {'LSLR'} {'RSLR'}`.

Example: `{'L','R','L','N'}`

Data Types: `cell`

`Length` — Length of path segment
positive numeric scalar

This property is read-only.

Length of the path segment or the flight path, specified as a positive numeric scalar in meters. This length is the sum of the elements in the MotionLengths vector.

Data Types: `double`

Object Functions

interpolate	Interpolate poses along UAV Dubins path segment
show	Visualize UAV Dubins path segment

Examples

Specify Motion Type for UAV Dubins Path

This example shows how to calculate a UAV Dubins path segment and connect poses using the `uavDubinsConnection` object for a specified motion type.

Create a `uavDubinsConnection` object.

```
connectionObj = uavDubinsConnection;
```

Define start and goal poses as [`x`, `y`, `z`, `headingAngle`] vectors.

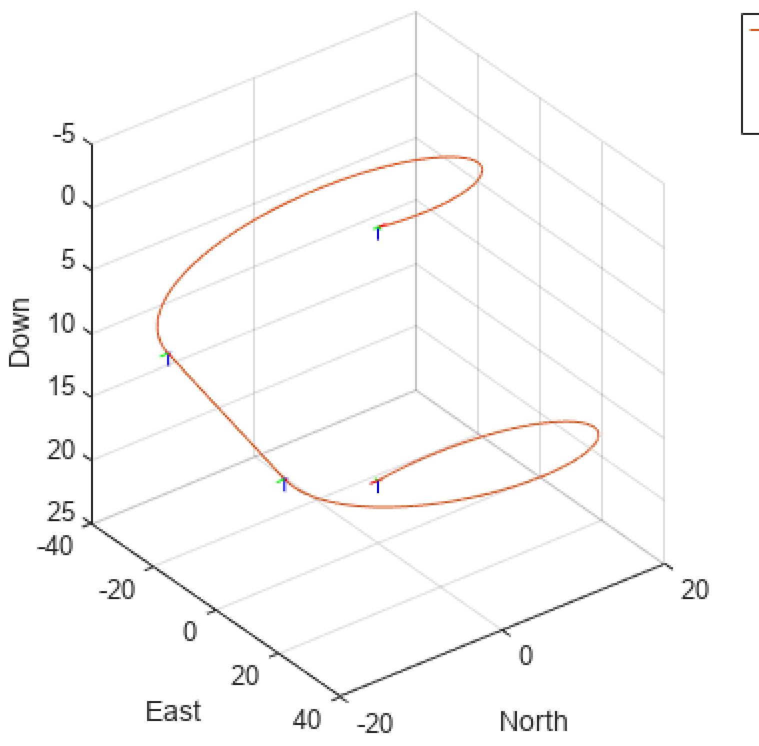
```
startPose = [0 0 0 0]; % [meters, meters, meters, radians]
goalPose = [0 0 20 pi];
```

Calculate a valid path segment and connect the poses for a specified motion type.

```
pathSegObj = uavDubinsPathSegment(connectionObj,startPose,goalPose,{'L','S','L','N'});
```

Show the generated path.

```
show(pathSegObj)
```



Verify the motion type of the returned path segment.

```
fprintf('Motion Type: %s\n',strjoin(pathSegObj.MotionTypes));
```

```
Motion Type: L S L N
```

Version History

Introduced in R2019b

Extended Capabilities

C/C++ Code Generation

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

See Also

`interpolate` | `show`

uavCoveragePlanner

Path planner for UAV space coverage

Description

The `uavCoveragePlanner` object plans an optimal path that a UAV can follow to cover an region of interest with a sensor such as a camera for precision agriculture and image mapping applications. The object finds the optimal path by finding a sweep angle that minimizes the number of turns in each polygon and uses one of two specified solver algorithms to optimize the connecting path between regions.

Creation

Syntax

```
planner = uavCoveragePlanner(coverageSpace)
planner = uavCoveragePlanner(coverageSpace,Name=Value)
```

Description

`planner = uavCoveragePlanner(coverageSpace)` creates a coverage planner `planner` from a coverage space `coverageSpace` and sets the `CoverageSpace` property.

`planner = uavCoveragePlanner(coverageSpace,Name=Value)` sets properties using one or more name-value arguments.

Properties

CoverageSpace — Coverage space for planner
`uavCoverageSpace` object

Coverage space for the planner, specified as a `uavCoverageSpace` object.

`SolverAlgorithm` — Solver algorithm for path planning
`"MinTraversal"` (default) | `"Exhaustive"`

Solver algorithm for path finding, specified as either `"MinTraversal"` or `"Exhaustive"`.

- `"Exhaustive"` — The planner exhaustively iterates through all permutations of sweep options to find the optimal one that minimizes connection distance between regions. See [1] for more details.
- `"MinTraversal"` — The planner uses a recursive minimum traversal approach through a graph of adjacent polygons to minimize the connection distances between the polygons from takeoff to landing. When a coverage path has no more immediate neighbors, the UAV flies to the nearest node in the graph and continues the optimality criteria. See [2] for more details.

Note The number of polygons must be less than 12 when using the exhaustive solver. If you have 12 or more polygons to survey, consider merging polygons or setting `SolverAlgorithm` to `"MinTraversal"`.

Generally, the exhaustive planner algorithm is better suited for smaller regions or separated regions where the distance optimality of the path is a high priority. Whereas the minimum traversal planner algorithm is faster focuses more on providing an intuitive solution for interconnected regions.

Data Types: `char` | `string`

`SolverParameters` — Solver algorithm parameters
structure

State space for the planner, specified as a structure containing these fields depending on the value of `SolverAlgorithm`:

When `SolverAlgorithm` is `"Exhaustive"`, specify a structure with these fields:

- `MinAdjacencyCount` — Minimum number of adjacent polygons that a sequence must have for the planner to consider the sequence as a valid solution, specified as a nonnegative integer.

  For example, consider a coverage space that has three polygons. Polygon 1 is adjacent to polygon 2, and polygon 2 is adjacent to polygon 3.

  - If the visiting sequence is [1 2 3], there are two adjacencies in the sequence: one adjacency between 1 and 2, and another between 2 and 3. The `uavCoveragePlanner` object considers this sequence as valid when `MinAdjacencyCount` is either `1` or `2`.
  - If the visiting sequence is [1 3 2], then the only adjacency in the sequence is between polygon 2 and polygon 3. So the `uavCoveragePlanner` object can only consider this sequence as valid when `MinAdjacencyCount` is `1`.

  Default is `1`.
- `VisitingSequence` — Order of traversal of polygons, specified as an *N*-element row vector, where *N* is the total number of polygons in the coverage space. For example, a visiting sequence of `[1 3 2]`, specifies that the polygons must be traversed in the order, polygon `1`, polygon `3`, and then polygon `2`. An empty row vector specifies no visiting sequence, enabling the solver algorithm to determine the visiting sequence.

  Default is `[]`.

Note The number of polygons must be less than 12 when using the exhaustive solver. If you have 12 or more polygons to survey, consider merging polygons or setting `SolverAlgorithm` to `"MinTraversal"`.

When `SolverAlgorithm` is `"MinTraversal"`, specify a structure with these fields:

- `StartingArea` — Index of polygon, where the UAV starts coverage specified as an integer scalar in the range [1, *N*]. *N* is the total number of polygons in the coverage space.

  Default is `1`.
- `VisitingSequence` — Order of traversal of polygons, specified as an *N*-element row vector, where *N* is the total number of polygons in the coverage space. For example, a visiting sequence of `[1 3 2]`, specifies that the polygons must be traversed in the order, polygon `1`, polygon `3`, and

then polygon 2. An empty row vector specifies no visiting sequence, enabling the solver algorithm to determine the visiting sequence.

When `VisitingSequence` is specified, the planner ignores the `startingArea`.

Default is `[]`.

Object Functions

plan	Plan coverage path between takeoff and landing
exportWaypointsPlan	Export waypoints to file

Examples

Plan Coverage Path Using Geodetic Coordinates

This example shows how to plan a coverage path that surveys the parking lots of the MathWorks Lakeside campus.

Get the geodetic coordinates for the MathWorks Lakeside campus. Then create the limits for our map.

```
mwLS = [42.3013 -71.375 0];
latlim = [mwLS(1)-0.003 mwLS(1)+0.003];
lonlim = [mwLS(2)-0.003 mwLS(2)+0.003];
```

Create a figure containing the map with the longitude and latitude limits.

```
fig = figure;
g = geoaxes(fig,Basemap="satellite");
geolimits(latlim,lonlim)
```

Get the outline of the first parking lot in longitude and latitude coordinates. Then create the polygon by concatenating them.

```
pl1lat = [42.3028 42.30325 42.3027 42.3017 42.3019]';
pl1lon = [-71.37527 -71.37442 -71.3736 -71.37378 -71.375234]';
pl1Poly = [pl1lat pl1lon];
```

Repeat the process for the second parking lot.

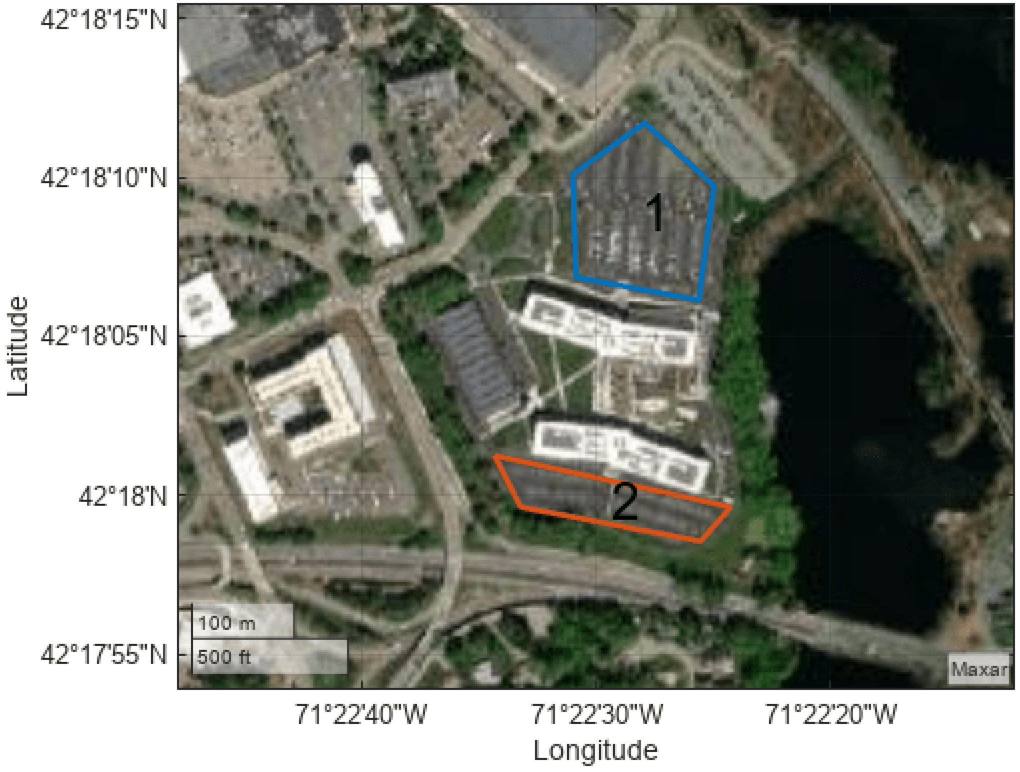
```
pl2lat = [42.30035 42.2999 42.2996 42.2999]';
pl2lon = [-71.3762 -71.3734 -71.37376 -71.37589]';
pl2poly = [pl2lat pl2lon];
```

Create the coverage space with both of those polygons, set the coverage space to use geodetic coordinates, and set the reference location to the MathWorks Lakeside campus location.

```
cs = uavCoverageSpace(Polygons={pl1Poly,pl2poly},UseLocalCoordinates=false,ReferenceLocation=mwLS
```

Set the height at which to fly the UAV to 25 meters, and the sensor footprint width to 20 meters. Then show the coverage space on the map.

```
ReferenceHeight = 25;
cs.UnitWidth = 20;
show(cs,Parent=g);
```

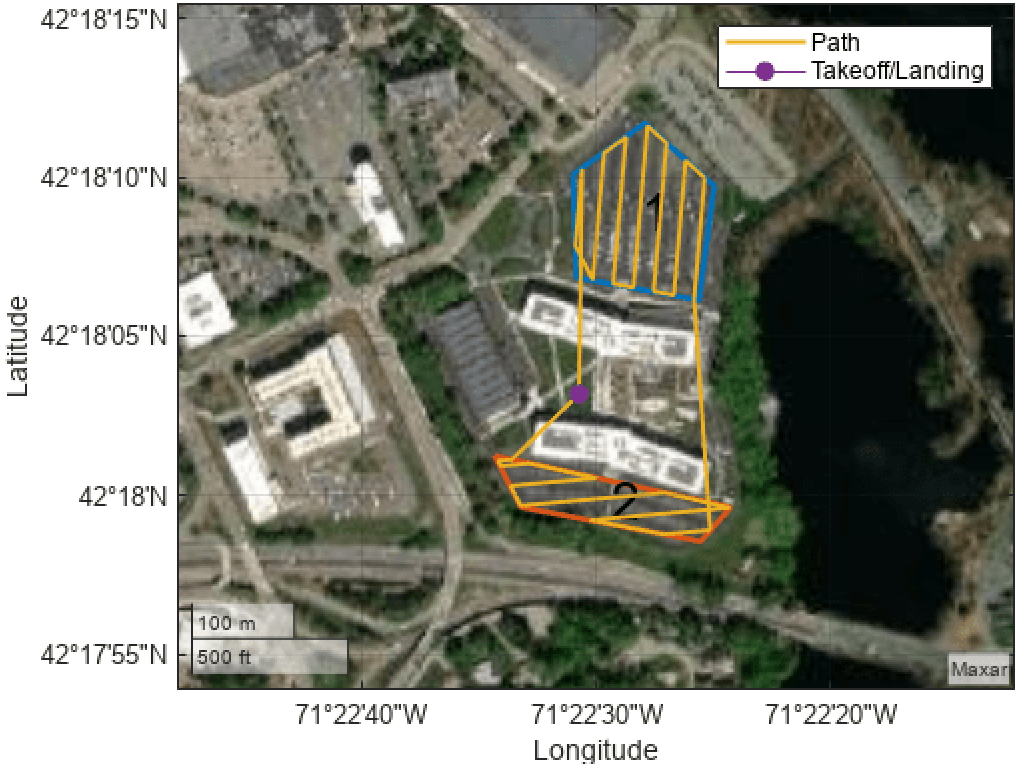


Set the sweep angle for polygons 1 and 2 to 85 and 5 degrees, respectively, to have paths that are parallel to the roads in the parking lots. Then create the coverage planner for that coverage space with the exhaustive solver algorithm.

```
setCoveragePattern(cs,1,SweepAngle=85)
setCoveragePattern(cs,2,SweepAngle=5)
cp = uavCoveragePlanner(cs,Solver="Exhaustive");
```

Set the takeoff position to a location in the courtyard, then plan the coverage path.

```
takeoff = [42.30089 -71.3752, 0];
[wp,soln] = plan(cp,takeoff);
hold on
geoplot(wp(:,1),wp(:,2),LineWidth=1.5);
geoplot(takeoff(1),takeoff(2),MarkerSize=25,Marker=".")
legend("","","Path","Takeoff/Landing")
hold off
```



Plan Coverage Path for Defined Region

This example shows how to plan a coverage path for a region in local coordinates and compares the results of using the exhaustive solver with the results of using the minimum traversal solver.

Define the vertices for a coverage space.

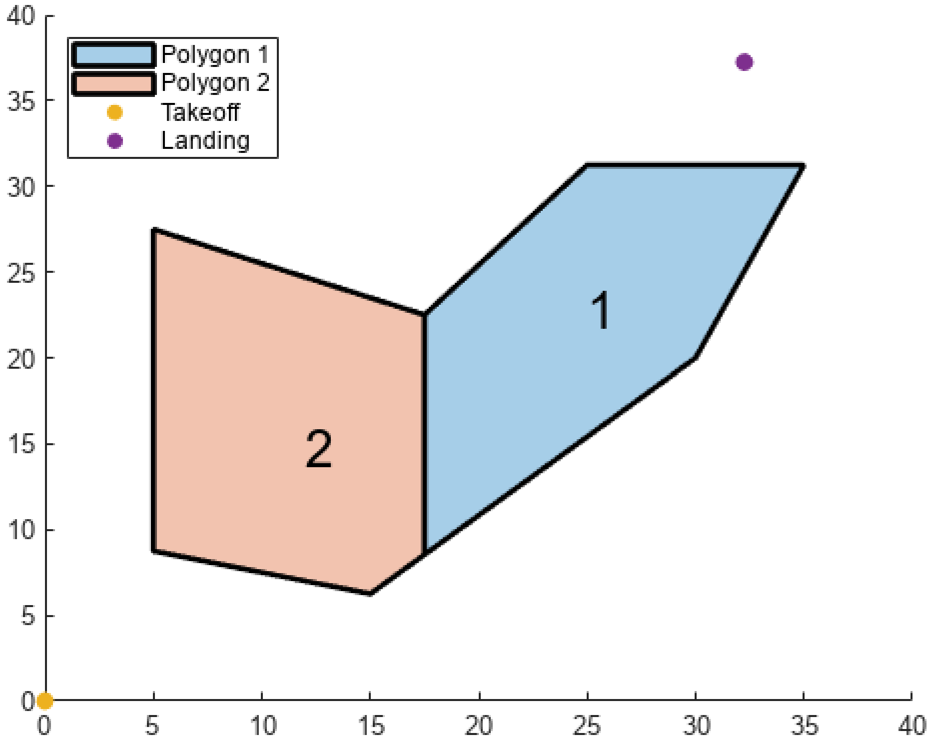
```
area = [5 8.75; 5 27.5; 17.5 22.5; 25 31.25; 35 31.25; 30 20; 15 6.25];
```

Because vertices define a concave polygon and the coverage planner requires convex polygons, decompose the polygon into convex polygons. Then create a coverage space with the polygons from decomposition.

```
polygons = coverageDecomposition(area);
cs = uavCoverageSpace(Polygons=polygons);
```

Define the takeoff and landing positions at `[0 0 0]` and `[32.25 37.25 0]`, respectively. Then show the coverage space and plot the takeoff and landing positions.

```
takeoff = [0 0 0];
landing = [32.25 37.25 0];
show(cs);
exampleHelperPlotTakeoffLandingLegend(takeoff,landing)
```



Create a coverage planner with the exhaustive solver algorithm and another coverage planner with a minimum traversal solver algorithm. Because `Polygon 2` is closer to the takeoff position, set the visiting sequence of the solver parameters such that we traverse `Polygon 2` first.

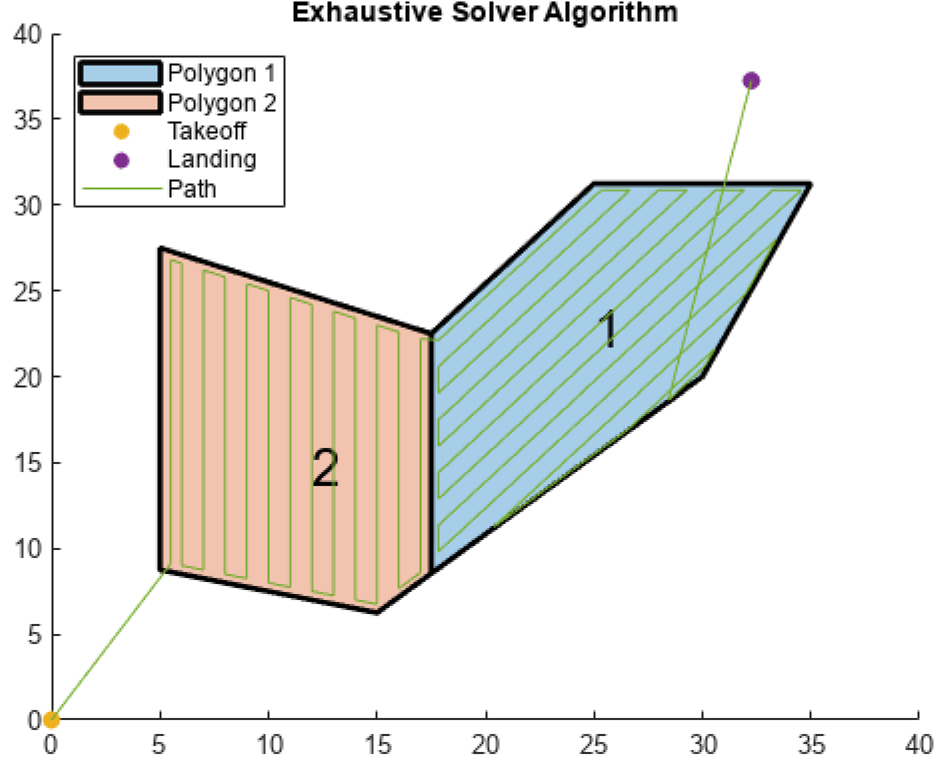
```
cpeExh = uavCoveragePlanner(cs,Solver="Exhaustive");
cpMin = uavCoveragePlanner(cs,Solver="MinTraversal");
cpeExh.SolverParameters.VisitingSequence = [2 1];
cpMin.SolverParameters.VisitingSequence = [2 1];
```

Plan with both solver algorithms using the same takeoff and landing positions.

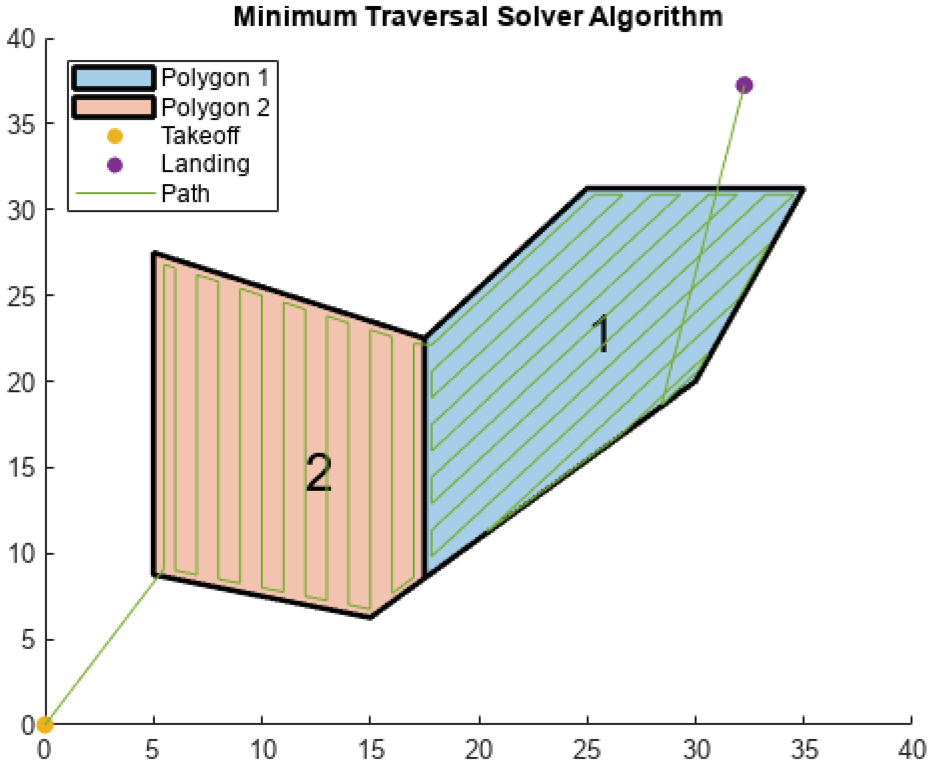
```
[wptsExh,solnExh] = plan(cpeExh,takeoff,landing);
[wptsMin,solnMin] = plan(cpMin,takeoff,landing);
```

Show the planned path for both the exhaustive and the minimum traversal algorithms.

```
figure
show(cs);
title("Exhaustive Solver Algorithm")
exampleHelperPlotTakeoffLandingLegend(takeoff,landing,wptsExh)
```




```
figure
show(cs);
title("Minimum Traversal Solver Algorithm")
exampleHelperPlotTakeoffLandingLegend(takeoff,landing,wptsMin)
```



Export the waypoints from the minimum traversal solver to a `.waypoints` file with the reference frame set to north-east-down.

```
exportWaypointsPlan(cpMin,solnMin,"coveragepath.waypoints",ReferenceFrame="NED")
```

Execute Coverage Plan Using UAV Mission

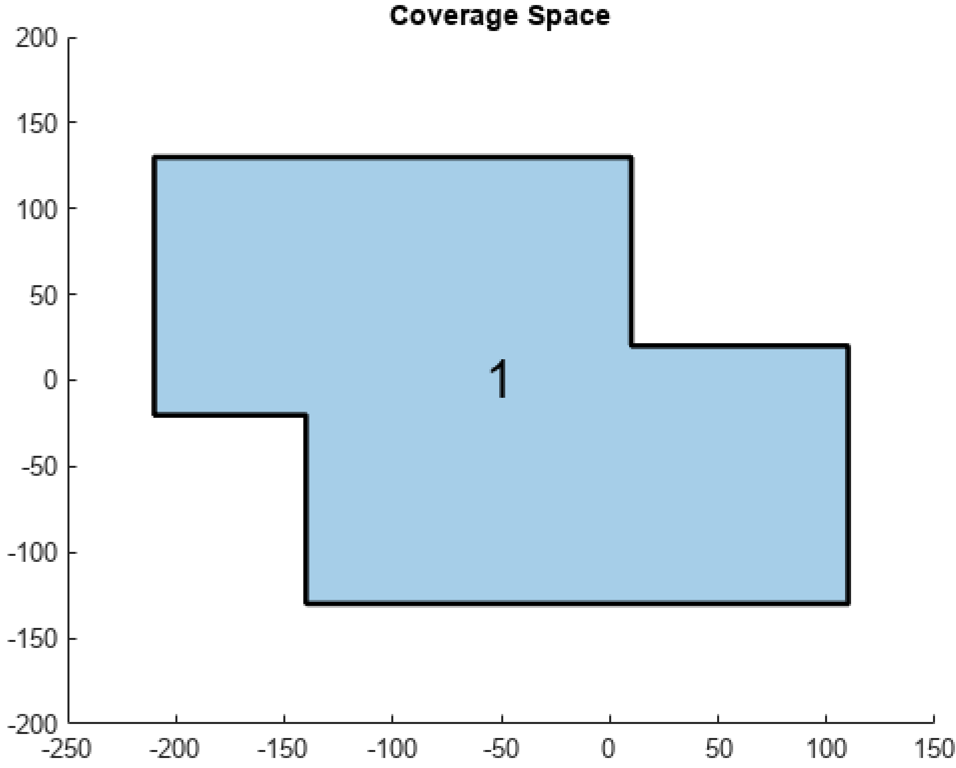
Initialize the settings to use for the coverage planner, coverage space, and mission. Set a coverage width to 65 meters, the region as polygon vertices, takeoff and landing locations, the UAV elevation during flight to 150 meters, and a geocenter.

```
coverageWidth = 65;
region = [-210 130; 10 130; 10 20; 110 20;
          110 -130; -140 -130; -140 -20; -210 -20];
takeoff = [-250 150 0];
landing = [0 -200 0];
uavElevation = 150;
geocenter = [-45 71 0];
```

Create the coverage space with those UAV coverage space settings.

```
cs = uavCoverageSpace(Polygons=region, ...
                      UnitWidth=coverageWidth, ...
                      ReferenceHeight=uavElevation, ...
                      ReferenceLocation=geocenter);
```

```
cs.show;
title("Coverage Space")
```

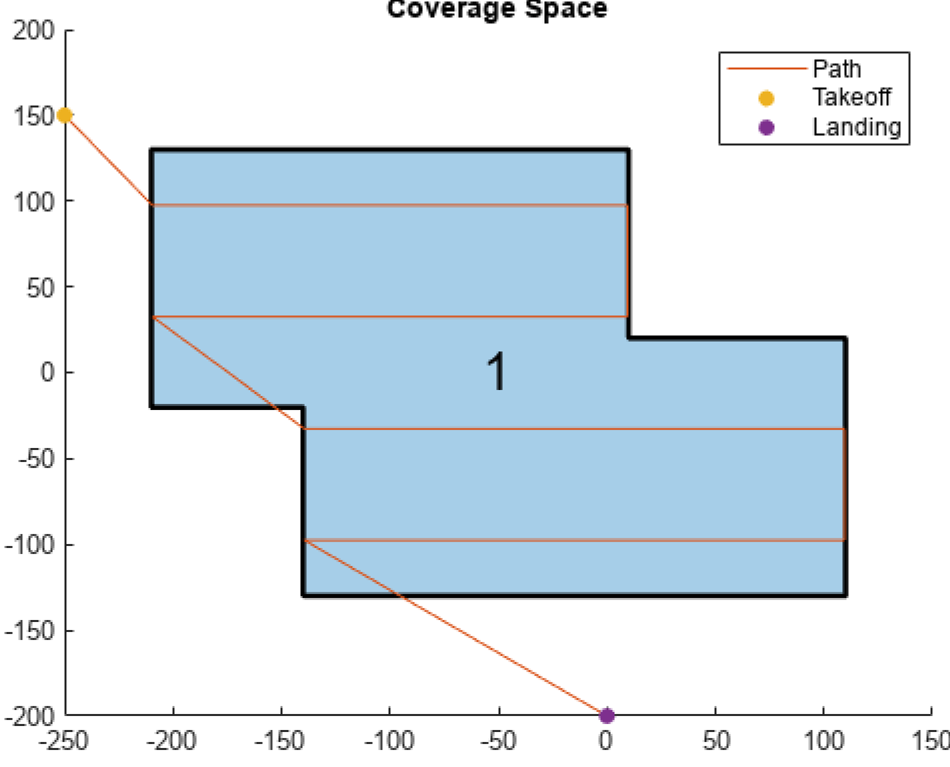


Create a coverage planner for the coverage space and plan the coverage path with the specified takeoff and landing locations.

```
cp = uavCoveragePlanner(cs);
[waypoints,solnInfo] = cp.plan(takeoff,landing);
```

Plot the waypoints, and the takeoff and landing locations on the coverage space.

```
hold on
plot(waypoints(:,1),waypoints(:,2))
scatter(takeoff(1),takeoff(2),"filled")
scatter(landing(1),landing(2),"filled")
legend("","Path","Takeoff","Landing")
hold off
```

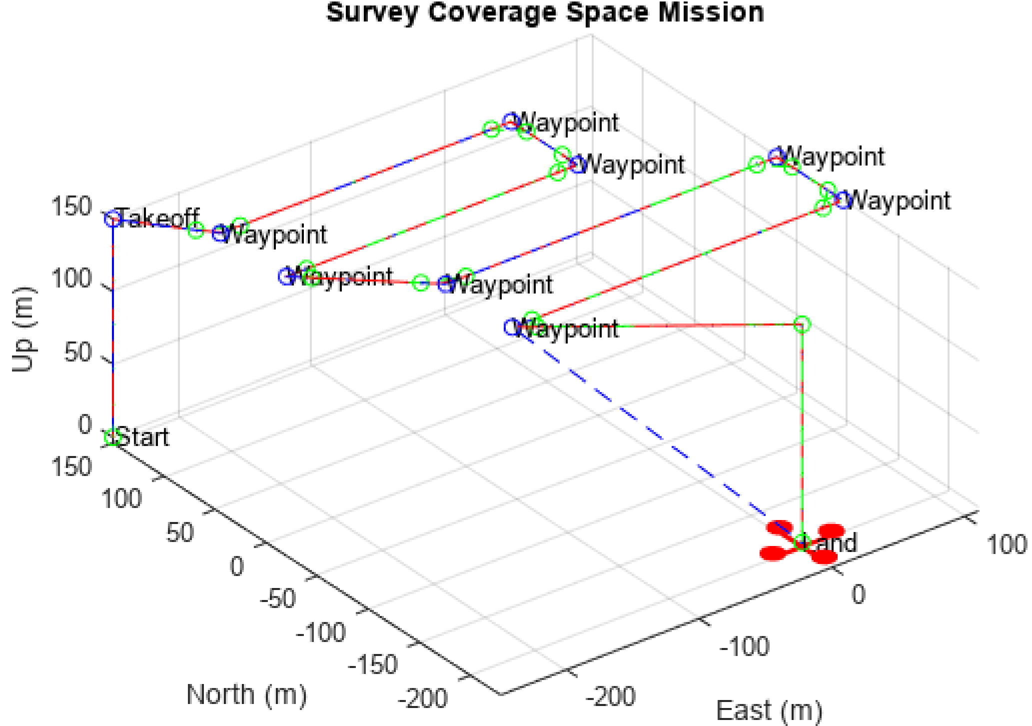


Export the waypoints to a `waypoints` file and create a UAV mission from that file with a speed of `10` meters per second and an initial yaw of `90` degrees.

```
exportWaypointsPlan(cp,solnInfo,"customCoverage.waypoints");
mission = uavMission(PlanFile="customCoverage.waypoints",Speed=10,InitialYaw=90);
```

Use the `exampleHelperSimulateUAVMission` helper function to visualize the UAV mission with a simulation time of 60 seconds.

```
exampleHelperSimulateUAVMission(mission,geocenter)
```



Version History

Introduced in R2023a

References

[1] Torres, Marina, David A. Pelta, José L. Verdegay, and Juan C. Torres. "Coverage Path Planning with Unmanned Aerial Vehicles for 3D Terrain Reconstruction." *Expert Systems with Applications* 55 (August 2016): 441–51. https://doi.org/10.1016/j.eswa.2016.02.007.

[2] Li, Yan, Hai Chen, Meng Joo Er, and Xinmin Wang. "Coverage Path Planning for UAVs Based on Enhanced Exact Cellular Decomposition Method." *Mechatronics* 21, no. 5 (August 2011): 876–85. https://doi.org/10.1016/j.mechatronics.2010.10.009.

See Also

`uavCoverageSpace` | `uavMission`

Topics

"Optimally Survey and Customize Coverage of Region of Interest using Coverage Planner"

uavCoverageSpace

2D coverage area for coverage planner

Description

The `uavCoverageSpace` object represents the coverage space as a set of convex polygons to use with the `uavCoveragePlanner` to perform coverage planning. Given a coverage area, the polygons form its configuration space.

Creation

Syntax

```
space = uavCoverageSpace
space = uavCoverageSpace(polygonVertices)
space = uavCoverageSpace( ___ ,Name=Value)
```

Description

`space = uavCoverageSpace` creates an empty coverage space with default properties.

`space = uavCoverageSpace(polygonVertices)` creates a coverage space defined by convex one or more convex polygons as vertices`polygonVertices`, and sets the `Polygons` property.

`space = uavCoverageSpace( ___ ,Name=Value)` sets properties using one or more name-value arguments. For example, `uavCoverageSpace(Overlap=0.5)` sets the `Overlap` property to `0.5`.

Input Arguments

`polygonVertices` — Vertices of convex polygons to survey
N-element cell array of *M*-by-2 matrices

Convex polygons to survey, specified as a *N*-element cell array of *M*-by-2 matrices. *N* is the total number of polygons, and *M* is the total number of vertices that define each polygon.

Concave polygons must be decomposed into convex polygons using the `coverageDecomposition` function.

The format of the vertices depends on the `UseLocalCoordinates` property:

- `UseLocalCoordinates=true` — Format is local *xy*-coordinates in the form [*x y*], in meters. `UseLocalCoordinates` is true by default.
- `UseLocalCoordinates=false` — Format is geodetic coordinates in the form [*latitude longitude*]. *latitude* and *longitude* are in degrees.

Example: `{[0 0; 0 1; 1 1; 1 0],[1 1; 2 2; 3 1]}`

Data Types: `single` | `double`

Properties

`Polygons` — Polygons to survey
`[1×0 struct]` (default) | *N*-element row vector of structures

Polygons to survey, specified as a *N*-element row vector of structures. *N* is the total number of polygons specified in the `polygons` argument. Each structure contains the `Vertices` and `SweepAngle`:

- `Vertices` — Vertices that define the polygon, specified as an *M*-by-2 matrix. *M* is the total number of vertices.

  The format of the vertices depends on the `UseLocalCoordinates` property:

  - `UseLocalCoordinates=true` — Format is local *xy*-coordinates in the form [*x y*], in meters. `UseLocalCoordinates` is true by default.
  - `UseLocalCoordinates=false` — Format is geodetic coordinates in the form [*latitude longitude*]. *latitude* and *longitude* are in degrees.
- `SweepAngle` — Angle at which the UAV sweeps the polygon, specified as numeric scalar, in degrees. This value is NaN by default until you specify it or until the solver algorithm determines it when you plan the path with the `uavCoveragePlanner` object.

`Overlap` — Overlap between sensor footprints in single row
`0` (default) | nonnegative numeric scalar in the range [0, 1)

Overlap between sensor footprints in a single row, specified as a nonnegative numeric scalar in the range [0, 1). A value of `0` specifies that the sensor footprints have no overlap, and a value of `1` indicates that the sensor footprints fully overlap.

Data Types: `single` | `double`

`Sidelap` — Overlap between sensor footprints of two rows
`0` (default) | nonnegative numeric scalar in the range [0, 1]

Overlap between sensor footprints of two rows, specified as a nonnegative numeric scalar in the range [0, 1]. A value of `0` specifies that the sensor footprints have no sidelap, and a value of `1` specifies that the sensor footprints fully sidelap.

Data Types: `single` | `double`

`UnitWidth` — Width of sensor footprint
`1` (default) | positive numeric scalar

Width of sensor footprint, specified as a positive numeric scalar, in meters. This measurement is along the sweep line direction.

Data Types: `single` | `double`

`UnitLength` — Length of sensor footprint
`1` (default) | positive numeric scalar

Length of sensor footprint, specified as a positive numeric scalar, in meters. This measurement is perpendicular to the sweep line direction.

Data Types: `single` | `double`

`ReferenceHeight` — Flight altitude of UAV
`100` (default) | nonnegative numeric scalar

Flight altitude of UAV, specified as a nonnegative numeric scalar, in meters.

When `UseLocalCoordinates` is `true` or `1`, the reference height is with respect to the local reference frame. When `UseLocalCoordinates` is `false` or `0`, the coverage space assumes a flat-Earth approximation and the reference height is with respect to the ground at absolute zero meters.

Data Types: `single` | `double`

`UseLocalCoordinates` — Compute waypoints in local or geodetic format
`true` or `1` (default) | `false` or `0`

Use local coordinate or geodetic coordinate format, specified as `1` (`true`) for local coordinate format or `0` (`false`) for geodetic coordinate format.

- When `UseLocalCoordinates` is `true` or `1`:
  - Vertices of polygons in the `uavCoverageSpace` are specified as *xy*-coordinates in the form [*x y*], in meters.
  - `takeoff` and `landing` positions are specified as *xyz*-coordinates in the form [*x y z*], in meters.
  - Planned waypoints are returned as *xyz*-coordinates in the form [*x y z*], in meters.
- When `UseLocalCoordinates` is `false` or `0`:
  - Vertices of polygons in the `uavCoverageSpace` are specified as geodetic coordinates in the form [*latitude longitude*]. *latitude* and *longitude* are in degrees.
  - `takeoff` and `landing` positions are specified as geodetic coordinates in the form [*latitude longitude altitude*]. *latitude* and *longitude* are in degrees, and altitude is in meters.
  - Planned waypoints are returned as geodetic coordinates in the form [*latitude longitude altitude*]. *latitude* and *longitude* are in degrees, and *altitude* is in meters.

`ReferenceLocation` — Georeference to compute geodetic waypoint coordinates
`[0 0 0]` | three-element row vector

Georeference to compute and export to geodetic waypoint coordinates, specified as an three-element row vector of the form [*latitude longitude altitude*]. *latitude* and *longitude* are in degrees, and *altitude* is in meters.

Data Types: `double`

Object Functions

setCoveragePattern	Set sweep angle for polygon in coverage area
show	Visualize 2D coverage space

Examples

Plan Coverage Path Using Geodetic Coordinates

This example shows how to plan a coverage path that surveys the parking lots of the MathWorks Lakeside campus.

Get the geodetic coordinates for the MathWorks Lakeside campus. Then create the limits for our map.

```
mwLS = [42.3013 -71.375 0];
latlim = [mwLS(1)-0.003 mwLS(1)+0.003];
lonlim = [mwLS(2)-0.003 mwLS(2)+0.003];
```

Create a figure containing the map with the longitude and latitude limits.

```
fig = figure;
g = geoaxes(fig,Basemap="satellite");
geolimits(latlim,lonlim)
```

Get the outline of the first parking lot in longitude and latitude coordinates. Then create the polygon by concatenating them.

```
pl1lat = [42.3028 42.30325 42.3027 42.3017 42.3019]';
pl1lon = [-71.37527 -71.37442 -71.3736 -71.37378 -71.375234]';
pl1Poly = [pl1lat pl1lon];
```

Repeat the process for the second parking lot.

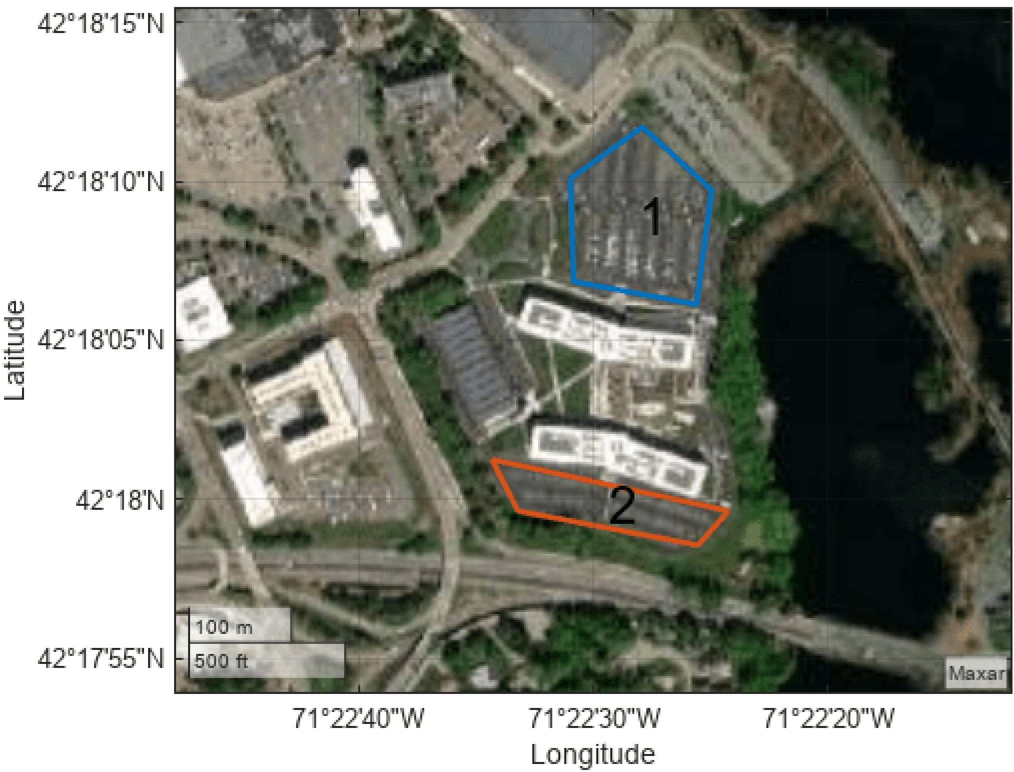
```
pl2lat = [42.30035 42.2999 42.2996 42.2999]';
pl2lon = [-71.3762 -71.3734 -71.37376 -71.37589]';
pl2poly = [pl2lat pl2lon];
```

Create the coverage space with both of those polygons, set the coverage space to use geodetic coordinates, and set the reference location to the MathWorks Lakeside campus location.

```
cs = uavCoverageSpace(Polygons={pl1Poly,pl2poly},UseLocalCoordinates=false,ReferenceLocation=mwLS
```

Set the height at which to fly the UAV to 25 meters, and the sensor footprint width to 20 meters. Then show the coverage space on the map.

```
ReferenceHeight = 25;
cs.UnitWidth = 20;
show(cs,Parent=g);
```

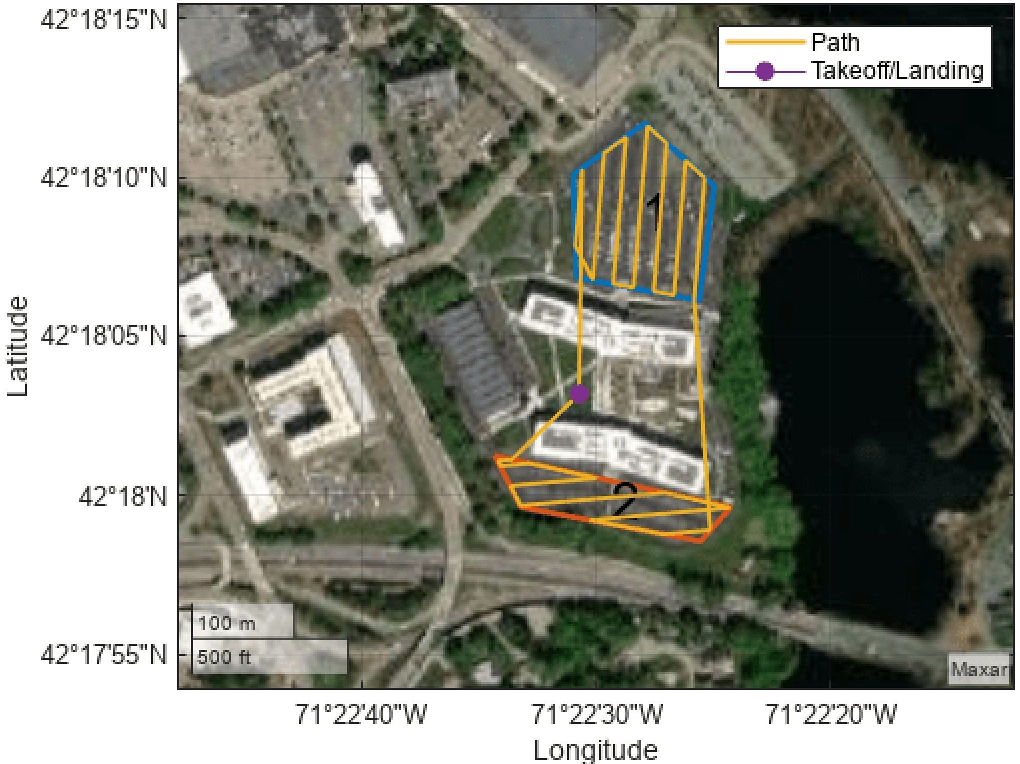


Set the sweep angle for polygons 1 and 2 to 85 and 5 degrees, respectively, to have paths that are parallel to the roads in the parking lots. Then create the coverage planner for that coverage space with the exhaustive solver algorithm.

```
setCoveragePattern(cs,1,SweepAngle=85)
setCoveragePattern(cs,2,SweepAngle=5)
cp = uavCoveragePlanner(cs,Solver="Exhaustive");
```

Set the takeoff position to a location in the courtyard, then plan the coverage path.

```
takeoff = [42.30089 -71.3752, 0];
[wp,soln] = plan(cp,takeoff);
hold on
geoplot(wp(:,1),wp(:,2),LineWidth=1.5);
geoplot(takeoff(1),takeoff(2),MarkerSize=25,Marker=".")
legend("","","Path","Takeoff/Landing")
hold off
```



Plan Coverage Path for Defined Region

This example shows how to plan a coverage path for a region in local coordinates and compares the results of using the exhaustive solver with the results of using the minimum traversal solver.

Define the vertices for a coverage space.

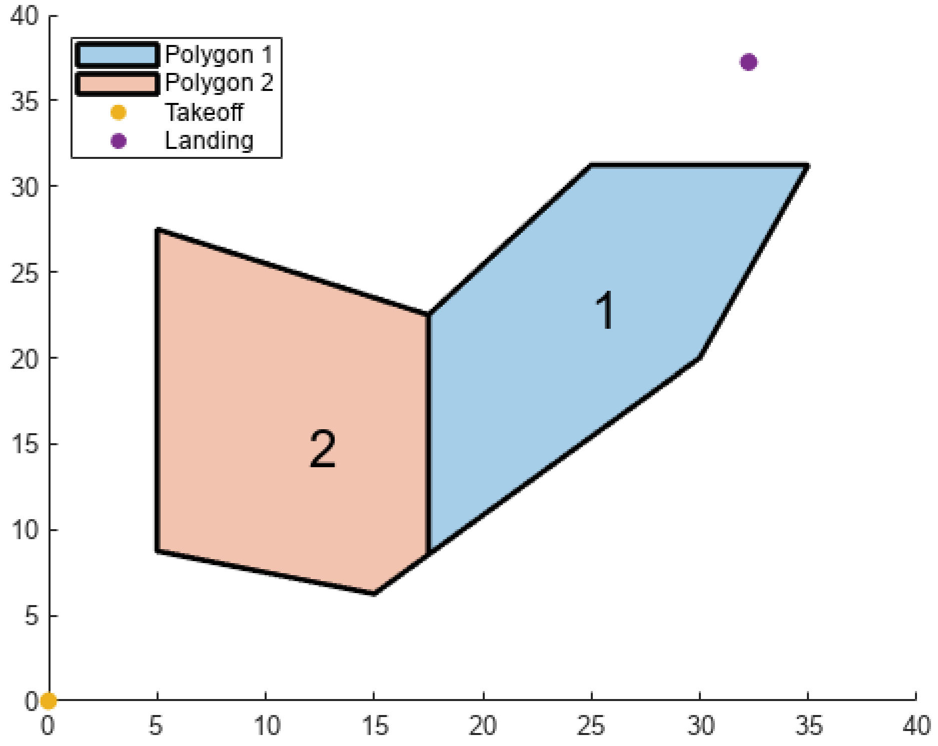
```
area = [5 8.75; 5 27.5; 17.5 22.5; 25 31.25; 35 31.25; 30 20; 15 6.25];
```

Because vertices define a concave polygon and the coverage planner requires convex polygons, decompose the polygon into convex polygons. Then create a coverage space with the polygons from decomposition.

```
polygons = coverageDecomposition(area);
cs = uavCoverageSpace(Polygons=polygons);
```

Define the takeoff and landing positions at `[0 0 0]` and `[32.25 37.25 0]`, respectively. Then show the coverage space and plot the takeoff and landing positions.

```
takeoff = [0 0 0];
landing = [32.25 37.25 0];
show(cs);
exampleHelperPlotTakeoffLandingLegend(takeoff,landing)
```



Create a coverage planner with the exhaustive solver algorithm and another coverage planner with a minimum traversal solver algorithm. Because `Polygon 2` is closer to the takeoff position, set the visiting sequence of the solver parameters such that we traverse `Polygon 2` first.

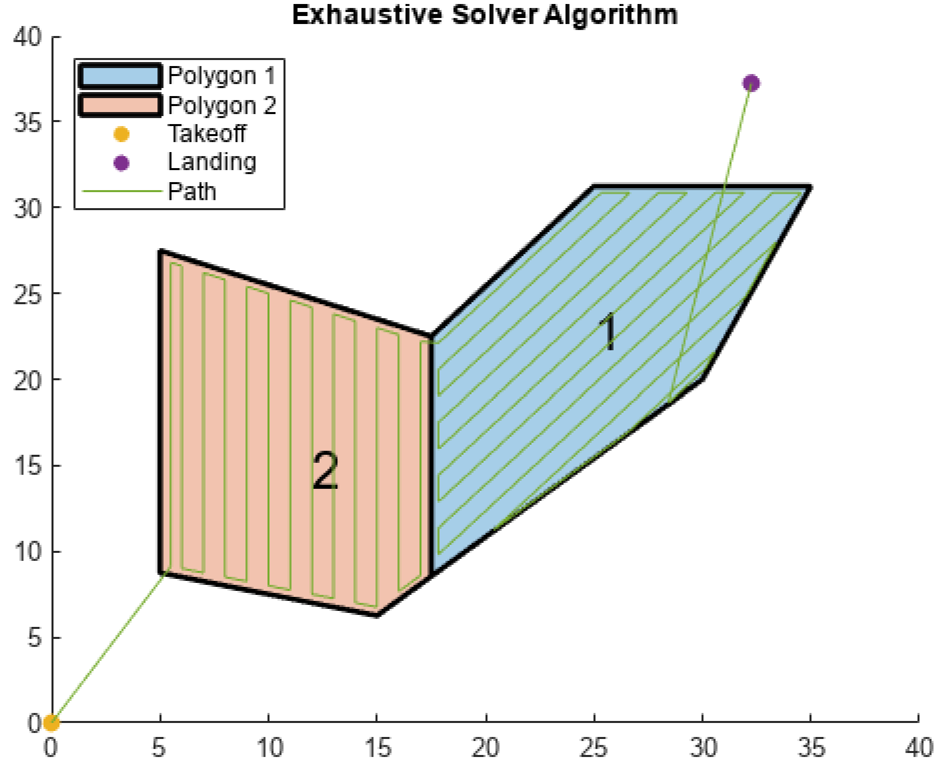
```
cpeExh = uavCoveragePlanner(cs,Solver="Exhaustive");
cpMin = uavCoveragePlanner(cs,Solver="MinTraversal");
cpeExh.SolverParameters.VisitingSequence = [2 1];
cpMin.SolverParameters.VisitingSequence = [2 1];
```

Plan with both solver algorithms using the same takeoff and landing positions.

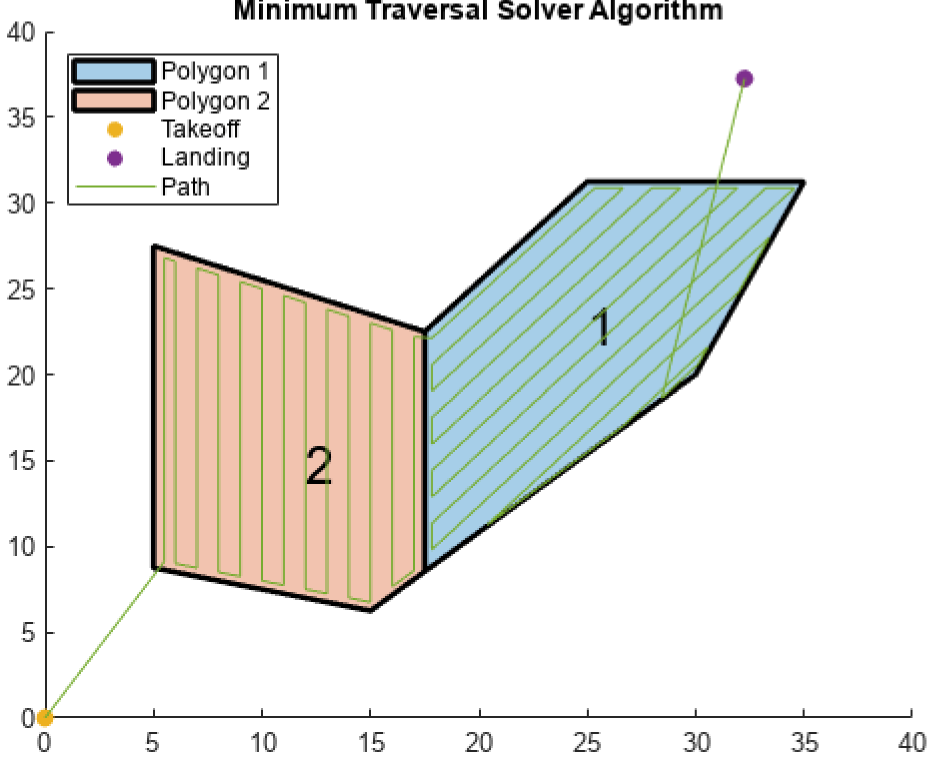
```
[wptsExh,solnExh] = plan(cpeExh,takeoff,landing);
[wptsMin,solnMin] = plan(cpMin,takeoff,landing);
```

Show the planned path for both the exhaustive and the minimum traversal algorithms.

```
figure
show(cs);
title("Exhaustive Solver Algorithm")
exampleHelperPlotTakeoffLandingLegend(takeoff,landing,wptsExh)
```




```
figure
show(cs);
title("Minimum Traversal Solver Algorithm")
exampleHelperPlotTakeoffLandingLegend(takeoff,landing,wptsMin)
```



Export the waypoints from the minimum traversal solver to a `.waypoints` file with the reference frame set to north-east-down.

```
exportWaypointsPlan(cpMin,solnMin,"coveragepath.waypoints",ReferenceFrame="NED")
```

Version History

Introduced in R2023a

See Also

`uavCoveragePlanner` | `coverageDecomposition`

Topics

"Optimally Survey and Customize Coverage of Region of Interest using Coverage Planner"

uavLidarPointCloudGenerator

Generate point clouds from meshes

Description

The `uavLidarPointCloudGenerator` System object generates detections from a statistical simulated lidar sensor. The system object uses a statistical sensor model to simulate lidar detections with added random noise. All detections are with respect to the coordinate frame of the vehicle-mounted sensor. You can use the `uavLidarPointCloudGenerator` object in a scenario, created using a `uavSensor`, containing static meshes, UAV platforms, and sensors.

To generate lidar point clouds:

1. Create the `uavLidarPointCloudGenerator` object and set its properties.
2. Call the object with arguments, as if it were a function.

To learn more about how System objects work, see What Are System Objects?

Creation

Syntax

```
lidar = uavLidarPointCloudGenerator
lidar = uavLidarPointCloudGenerator(Name,Value)
```

Description

`lidar = uavLidarPointCloudGenerator` creates a statistical sensor model to generate point cloud for a lidar. This sensor model will have default properties.

`lidar = uavLidarPointCloudGenerator(Name,Value)` sets properties using one or more name-value pairs. For example, `uavLidarPointCloudGenerator('UpdateRate',100,'HasNoise',0)` creates a lidar point cloud generator that reports detections at an update rate of 100 Hz without noise.

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.

`UpdateRate` — Update rate of the lidar sensor
`10` (default) | positive real scalar

Update rate of the lidar sensor, specified as a positive real scalar in Hz. This property sets the frequency at which new detections happen.

Example: `20`

Data Types: `double`

`MaxRange` — Maximum detection range
`120` (default) | positive real scalar

Maximum detection range of the sensor, specified as a positive real scalar. The sensor does not detect objects beyond this range. The units are in meters.

Example: `120`

Data Types: `double`

`RangeAccuracy` — Accuracy of range measurements
`0.0020` (default) | positive real scalar

Accuracy of the range measurements, specified as a positive real scalar in meters. This property sets the one-standard-deviation accuracy of the sensor range measurements.

Example: 0.001

Data Types: `single` | `double`

`AzimuthResolution` — Azimuthal resolution of lidar sensor
`0.1600` (default) | positive real scalar

Azimuthal resolution of lidar sensor, specified as a positive real scalar in degrees. The azimuthal resolution defines the minimum separation in azimuth angle at which the lidar sensor can distinguish two targets.

Example: `0.6000`

Data Types: `single` | `double`

`ElevationResolution` — Elevation resolution of lidar sensor
`1.2500` (default) | positive real scalar

Elevation resolution of lidar sensor, specified as a positive real scalar with units in degrees. The elevation resolution defines the minimum separation in elevation angle at which the lidar can distinguish two targets.

Example: `0.6000`

Data Types: `single` | `double`

`AzimuthLimits` — Azimuthal limits of lidar sensor
`[-180 180]` (default) | two-element vector

Azimuth limits of the lidar, specified as a two-element vector of the form [*min max*]. Units are in degrees.

Example: `[-60 100]`

Data Types: `single` | `double`

`ElevationLimits` — Elevation limits of lidar sensor
`[-20 20]` (default) | two-element vector

Elevation limits of the lidar, specified as a two-element vector of the form [*min max*]. Units are in degrees.

Example: `[-60 100]`

Data Types: `single` | `double`

`HasNoise` — Add noise to lidar sensor measurements
`true` or `1` (default) | `false` or `0`

Add noise to lidar sensor measurements, specified as `true` or `false`. Set this property to `true` to add noise to the sensor measurements. Otherwise, the measurements have no noise.

Example: `false`

Data Types: `logical`

`HasOrganizedOutput` — Output generated data as organized point cloud
`true` or `1` (default) | `false` or `0`

Output generated data as organized point cloud, specified as `true` or `false`. Set this property to `true` to output an organized point cloud. Otherwise, the output is unorganized.

Example: `false`

Data Types: `logical`

Usage

Syntax

```
ptCloud = lidar(tgts,simTime)
[ptCloud,isValidTime] = lidar(tgts,simTime)
```

Description

`ptCloud = lidar(tgts,simTime)` generates a lidar point cloud object `ptCloud` from the specified target object, `tgts`, at the specified simulation time `simTime`.

`[ptCloud,isValidTime] = lidar(tgts,simTime)` additionally returns `isValidTime` which specifies if the specified `simTime` is a multiple of the sensor's update interval (1/`UpdateRate`).

Input Arguments

`tgts` — Target object data
structure | structure array

Target object data, specified as a structure or structure array. Each structure corresponds to a mesh. The table shows the properties that the object uses to generate detections.

Target Object Data

Field	Description
Mesh	An extendedObjectMesh object representing the geometry of the target object in its own coordinate frame.
Position	A three-element vector defining the coordinate position of the target with respect to the sensor frame.
Orientation	A quaternion object or a 3-by-3 matrix, containing Euler angles, defining the orientation of the target with respect to the sensor frame.

simTime — Current simulation time
positive real scalar

Current simulation time, specified as a positive real scalar. The lidar object calls the lidar point cloud generator at regular intervals to generate new point clouds at a frequency defined by the updateRate property. The value of the UpdateRate property must be an integer multiple of the simulation time interval. Updates requested from the sensor between update intervals do not generate a point cloud. Units are in seconds.

Output Arguments

ptCloud — Point cloud data
pointCloud object

Point cloud data, returned as a pointCloud object.

isValidTime — Valid time to generate point cloud
false or 0 | true or 1

Valid time to generate point cloud, returned as logical 0 (false) or 1 (false). isValidTime is 0 when the requested update time is not a multiple of the updateRate property value.

Data Types: logical

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 Point Clouds from Mesh

This example shows how to use a statistical lidar sensor model to generate point clouds from a mesh.

Create Sensor Model

Create a statistical sensor model, `lidar`, using the `uavLidarPointCloudGenerator` System object.

```
lidar = uavLidarPointCloudGenerator('HasOrganizedOutput',false);
```

Create Floor

Use the `extendedObjectMesh` object to create mesh for the target object.

```
tgts.Mesh = scale(extendedObjectMesh('cuboid'),[100 100 2]);
```

Define the position of the target object with respect to the sensor frame.

```
tgts.Position = [0 0 -10];
```

Define the orientation of the target with respect to the sensor frame.

```
tgts.Orientation = quaternion([1 0 0 0]);
```

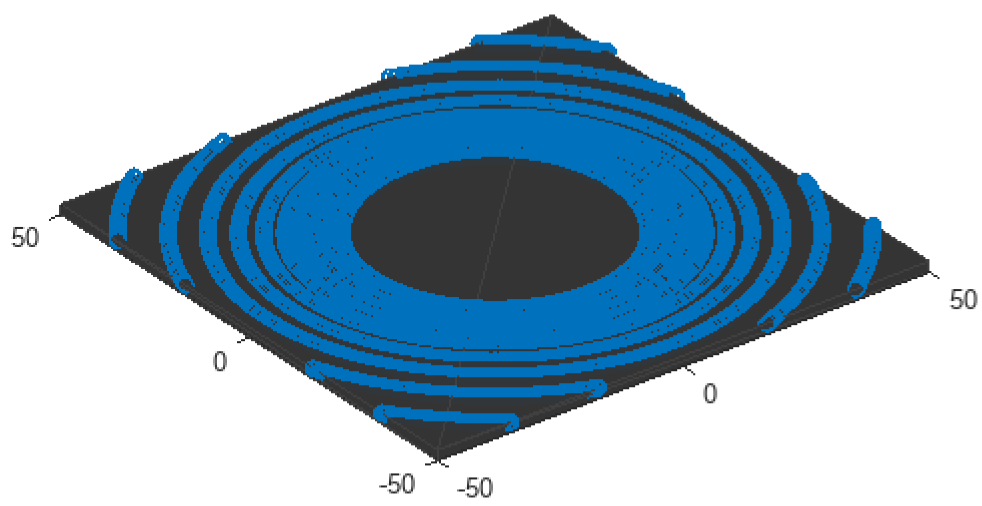
Generate Point Clouds from Floor

```
ptCloud = lidar(tgts,0);
```

Visualize

Use the `translate` function to translate the object mesh to its specified location and use the `show` function to visualize it. Use the `scatter3` function to plot the point clouds stored in `ptCloud`.

```
figure
show(translate(tgts.Mesh,tgts.Position));
hold on
scatter3(ptCloud.Location(:,1),ptCloud.Location(:,2), ...
       ptCloud.Location(:,3));
```



Version History

Introduced in R2020b

See Also

`uavScenario`

Topics

"UAV Scenario Tutorial"

uavOrbitFollower

Orbit location of interest using a UAV

Description

The `uavOrbitFollower` object is a 3-D path follower for unmanned aerial vehicles (UAVs) to follow circular paths that is based on a lookahead distance. Given the circle center, radius, and the pose, the orbit follower computes a desired yaw and course to follow a lookahead point on the path. The object also computes the cross-track error from the UAV pose to the path and tracks how many times the circular orbit has been completed.

Tune the `lookaheadDistance` input to help improve path tracking. Decreasing the distance can improve tracking, but may lead to oscillations in the path.

To orbit a location using a UAV:

1. Create the `uavOrbitFollower` object and set its properties.
2. Call the object with arguments, as if it were a function.

To learn more about how System objects work, see What Are System Objects?

Creation

Syntax

```
orbit = uavOrbitFollower
orbit = uavOrbitFollower(Name,Value)
```

Description

`orbit = uavOrbitFollower` returns an orbit follower object with default property values.

`orbit = uavOrbitFollower(Name,Value)` creates an orbit follower 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

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.

`UAV type` — Type of UAV
`'fixed-wing'` (default) | `'multirotor'`

Type of UAV, specified as either `'fixed-wing'` or `'multirotor'`.

`OrbitCenter` — Center of orbit
`[x y z]` vector

Center of orbit, specified as an `[x y z]` vector. `[x y z]` is the orbit center position in NED-coordinates (north-east-down) specified in meters.

Example: `[5,5,-10]`

Data Types: `single` | `double`

`OrbitRadius` — Radius of orbit
positive scalar

Radius of orbit, specified as a positive scalar in meters.

Example: `5`

Data Types: `single` | `double`

`TurnDirection` — Direction of orbit
scalar

Direction of orbit, specified as a scalar. Positive values indicate a clockwise turn as viewed from above. Negative values indicate a counter-clockwise turn. A value of `0` automatically determines the value based on the input `Pose`.

Example: `-1`

Data Types: `single` | `double`

`MinOrbitRadius` — Minimum orbit radius
`1` (default) | positive numeric scalar

Minimum orbit radius, specified as a positive numeric scalar in meters.

Data Types: `single` | `double`

`MinLookaheadDistance` — Minimum lookahead distance
`0.1` (default) | positive numeric scalar

Minimum lookahead distance, specified as a positive numeric scalar in meters.

Data Types: `single` | `double`

Usage

Syntax

```
[lookaheadPoint,desiredCourse,desiredYaw,orbitRadiusFlag,lookaheadDistFlag,
crossTrackError,numTurns] = orbit(currentPose,lookaheadDistance)
```

Description

`[lookaheadPoint,desiredCourse,desiredYaw,orbitRadiusFlag,lookaheadDistFlag,crossTrackError,numTurns] = orbit(currentPose,lookaheadDistance)` follows the set of waypoints specified in the waypoint follower object. The object takes the current position and lookahead distance to compute the lookahead point on the path. The desired course, yaw, and cross track error are also based on this lookahead point compared to the current position. `status` returns zero until the UAV has navigated all the waypoints.

Input Arguments

`currentPose` — Current UAV pose
`[x y z course]` vector

Current UAV pose, specified as a `[x y z course]` vector. This pose is used to calculate the lookahead point based on the input `LookaheadDistance`. `[x y z]` is the current position in meters. `course` is the current course in radians. The UAV course is the angle of direction of the velocity vector relative to north measured in radians.

Data Types: `single` | `double`

`lookaheadDistance` — Lookahead distance
positive numeric scalar

Lookahead distance along the path, specified as a positive numeric scalar in meters.

Data Types: `single` | `double`

Output Arguments

`lookaheadPoint` — Lookahead point on path
`[x y z]` position vector

Lookahead point on path, returned as an `[x y z]` position vector in meters.

Data Types: `double`

`desiredCourse` — Desired course
numeric scalar

Desired course, returned as numeric scalar in radians in the range of `[-pi, pi]`. The UAV course is the angle of direction of the velocity vector relative to north measured in radians. For fixed-wing type UAV, the values of desired course and desired yaw are equal.

Data Types: `double`

`desiredYaw` — Desired yaw
numeric scalar

Desired yaw, returned as numeric scalar in radians in the range of `[-pi, pi]`. The UAV yaw is the forward direction of the UAV regardless of the velocity vector relative to north measured in radians. For fixed-wing type UAV, the values of desired course and desired yaw are equal.

Data Types: `double`

`orbitRadiusFlag` — Orbit radius flag
`0` (default) | `1`

Orbit radius flag, returned as `0` or `1`. `0` indicates orbit radius is not saturated, `1` indicates orbit radius is saturated to minimum orbit radius value specified.

Data Types: `uint8`

`lookaheadDistFlag` — Lookahead distance flag
`0` (default) | `1`

Lookahead distance flag, returned as `0` or `1`. `0` indicates lookahead distance is not saturated, `1` indicates lookahead distance is saturated to minimum lookahead distance value specified.

Data Types: `uint8`

`crossTrackError` — Cross track error from UAV position to path
positive numeric scalar

Cross track error from UAV position to path, returned as a positive numeric scalar in meters. The error measures the perpendicular distance from the UAV position to the closest point on the path.

Data Types: `double`

`numTurns` — Number of times the UAV has completed the orbit
numeric scalar

Number of times the UAV has completed the orbit, specified as a numeric scalar. As the UAV circles the center point, this value increases or decreases based on the specified `Turn Direction` property. Decimal values indicate partial completion of a circle. If the UAV cross track error exceeds the lookahead distance, the number of turns is not updated.

`NumTurns` is reset whenever `Center`, `Radius`, or `TurnDirection` properties are changed.

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 Control Commands for Orbit Following

This example shows how to use the `uavOrbitFollower` to generate course and yaw commands for orbiting a location of interest with a UAV.

NOTE: This example requires you to install the UAV Library for Robotics System Toolbox®. Call `roboticsAddons` to open the Add-ons Explorer and install the library.

Create the orbit follower. Set the center of the location of interest and the radius of orbit. Set a `TurnDirection` of 1 for counter-clockwise rotation around the location.

```
orbFollower = uavOrbitFollower;

orbFollower.OrbitCenter = [1 1 5]';
orbFollower.OrbitRadius = 2.5;
orbFollower.TurnDirection = 1;
```

Specify the pose of the UAV and the lookahead distance for tracking the path.

```
pose = [0;0;5;0];
lookaheadDistance = 2;
```

Call the `orbFollower` object with the pose and lookahead distance. The object returns a lookahead point on the path, the desired course, and yaw. You can use the desired course and yaw to generate control commands for the UAV.

```
[lookaheadPoint,desiredCourse,desiredYaw,~,~] = orbFollower(pose,lookaheadDistance);
```

Version History

Introduced in R2019a

Extended Capabilities

C/C++ Code Generation

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

See Also

Functions

`control` | `derivative` | `environment` | `state` | `plotTransforms`

Objects

`uavWaypointFollower` | `fixedwing` | `multirotor`

Blocks

Orbit Follower | Waypoint Follower | UAV Guidance Model

uavMission

Mission data for UAV flight

Description

The `uavMission` object stores UAV mission data that can be used to generate flight trajectories for use in UAV scenario simulation.

Creation

Syntax

```
M = uavMission
M = uavMission(PlanFile=file)
M = uavMission( ___ ,Name=Value)
```

Description

`M = uavMission` creates an empty mission, `M`.

`M = uavMission(PlanFile=file)` creates a mission from the specified mission plan file `file`.

`M = uavMission( ___ ,Name=Value)` specifies properties using one or more name-value arguments in addition to any combination of input arguments from previous syntaxes.

Input Arguments

`file` — Mission plan file path
string scalar | character vector

Mission plan file path, specified as a string scalar or character vector. You must specify either a PX4 Mission JSON file (`.plan`) or MAVLink waypoints file (`.waypoints`).

Example: `uavMission(PlanFile="documents/matlab/uavwpts.waypoints")`

Data Types: `char` | `string`

Properties

`HomeLocation` — Home location
`[0 0 0]` (default) | three-element row vector

Home location, specified as a three-element row vector of the form [*latitude longitude altitude*]. The first two elements specify the latitude and longitude, respectively, of the UAV starting location in degrees, and the third specifies the starting altitude of the UAV in meters.

Example: `uavMission(HomeLocation=[4 2 10])`

`InitialYaw` — Initial yaw angle
`0` (default) | numeric scalar

Initial yaw angle, specified as a numeric scalar. This property specifies the angle between the *x*-axis of the UAV and north direction in the local NED frame of the mission's home location, in degrees.

Example: `uavMission(InitialYaw=45)`

Speed — Reference speed of UAV
`8` (default) | numeric scalar

Reference speed of the UAV, specified as a numeric scalar in meters per second.

Example: `uavMission(Speed=5)`

Data Types: `double`

`Frame` — Mission waypoint reference frame
`"Global"` (default) | `"GlobalRelativeAlt"` | `"LocalENU"` | `"LocalNED"`

Mission waypoint reference frame, specified as one of these values:

- `"Global"` — Global reference frame
- `"GlobalRelativeAlt"` — Global reference frame with an altitude relative to the home location
- `"LocalENU"` — Local east-north-up (ENU) reference frame
- `"LocalNED"` — Local north-east-down (NED) reference frame

Example: `uavMission(Frame="Global")`

Data Types: `char` | `string`

`NumMissionItems` — Number of mission items
`0` (default) | nonnegative integer

Number of mission items stored as a nonnegative integer. `NumMissionItems` increments each time a mission item is added to the mission.

This property is read-only.

Object Functions

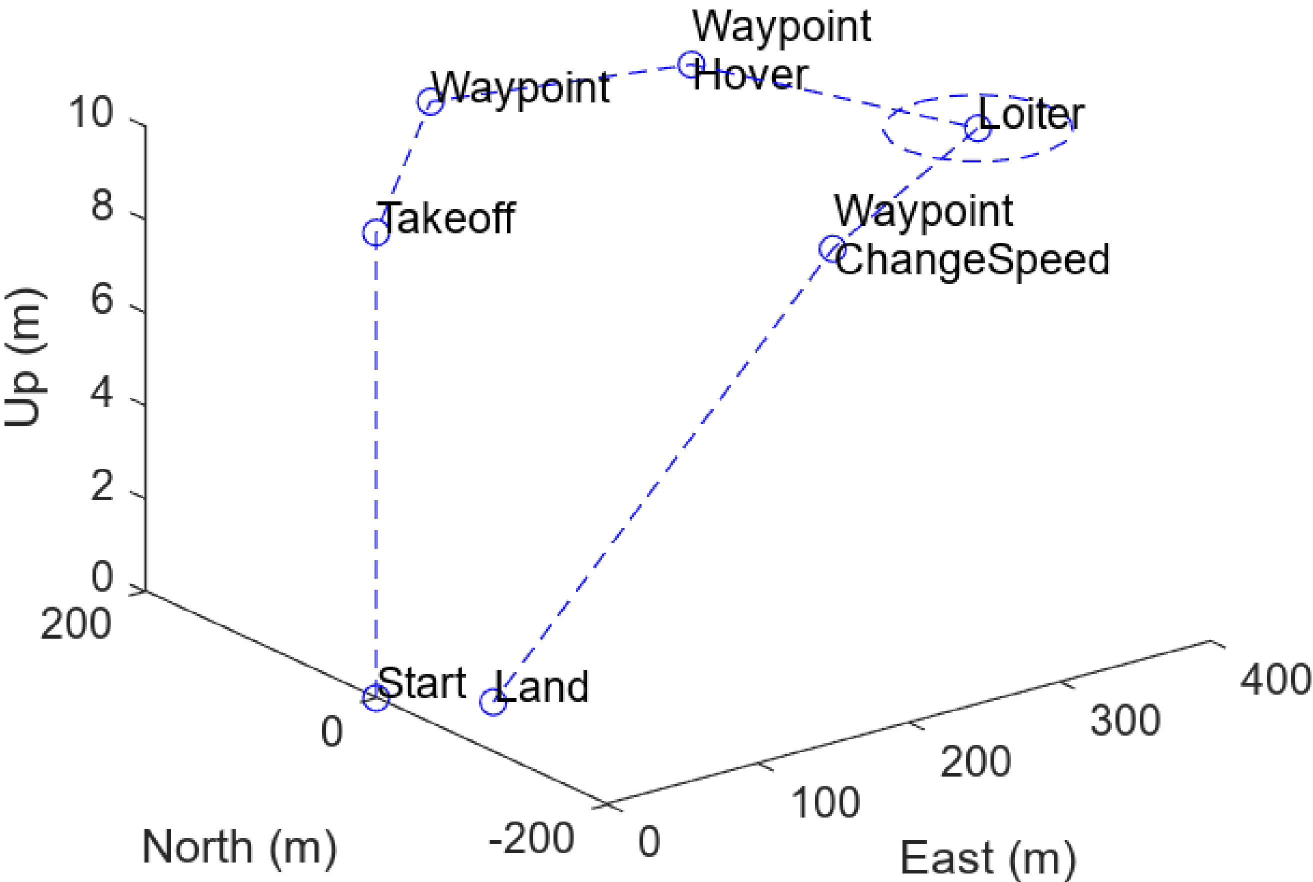
addChangeSpeed	Add change speed mission item
addHover	Add hover mission item
addLand	Add landing mission item
addLoiter	Add loiter mission item
addTakeoff	Add takeoff mission item
addWaypoint	Add waypoint mission item
copy	Copy UAV Mission
removeItem	Remove mission items at specified indices
show	Visualize UAV mission
showdetails	UAV mission data table

Examples

Generate Flight Trajectory for UAV Mission

Create a UAV mission by using the flight plan stored in a `.plan` file and show the mission.

```
mission = uavMission(PlanFile="flight.plan");
show(mission);
```



Create parsers for a multirotor UAV and a fixed-wing UAV.

```
mrmParser = multirotorMissionParser(TransitionRadius=2,TakeoffSpeed=2);
fwmParser = fixedwingMissionParser(TransitionRadius=15,TakeoffPitch=10);
```

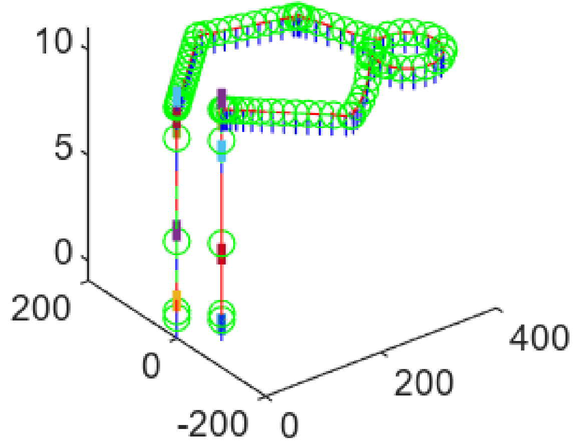
Generate one flight trajectory using each parser.

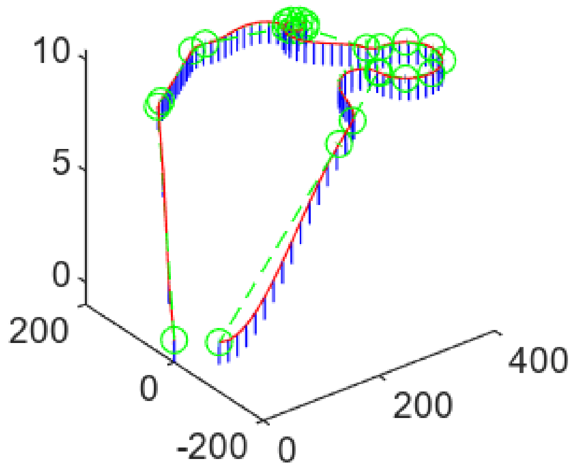
```
mrmTraj = parse(mrmParser,mission);
fwmTraj = parse(fwmParser,mission);
```

Visualize the mission and the flight trajectories separately.

```
figure
subplot(1,2,1)
```

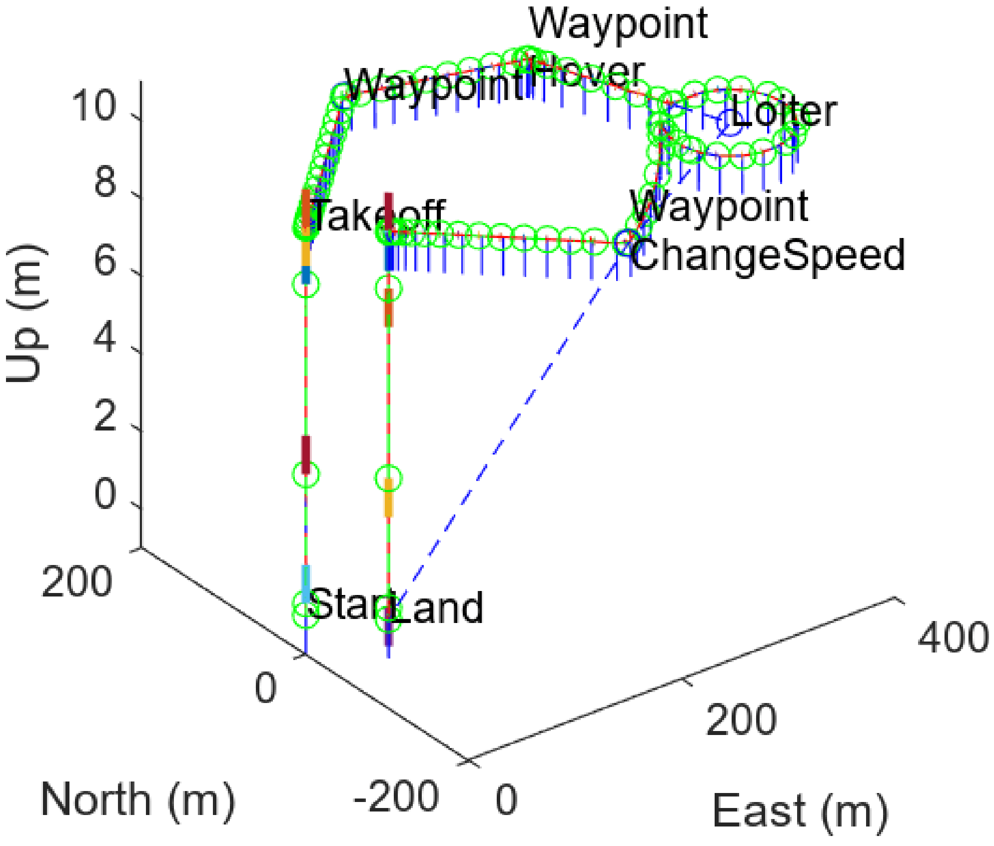
```
show(mrmTraj);
title("Multirotor Trajectory")
axis square
subplot(1,2,2)
show(fwmTraj);
title("Fixed-Wing Trajectory")
axis square
```



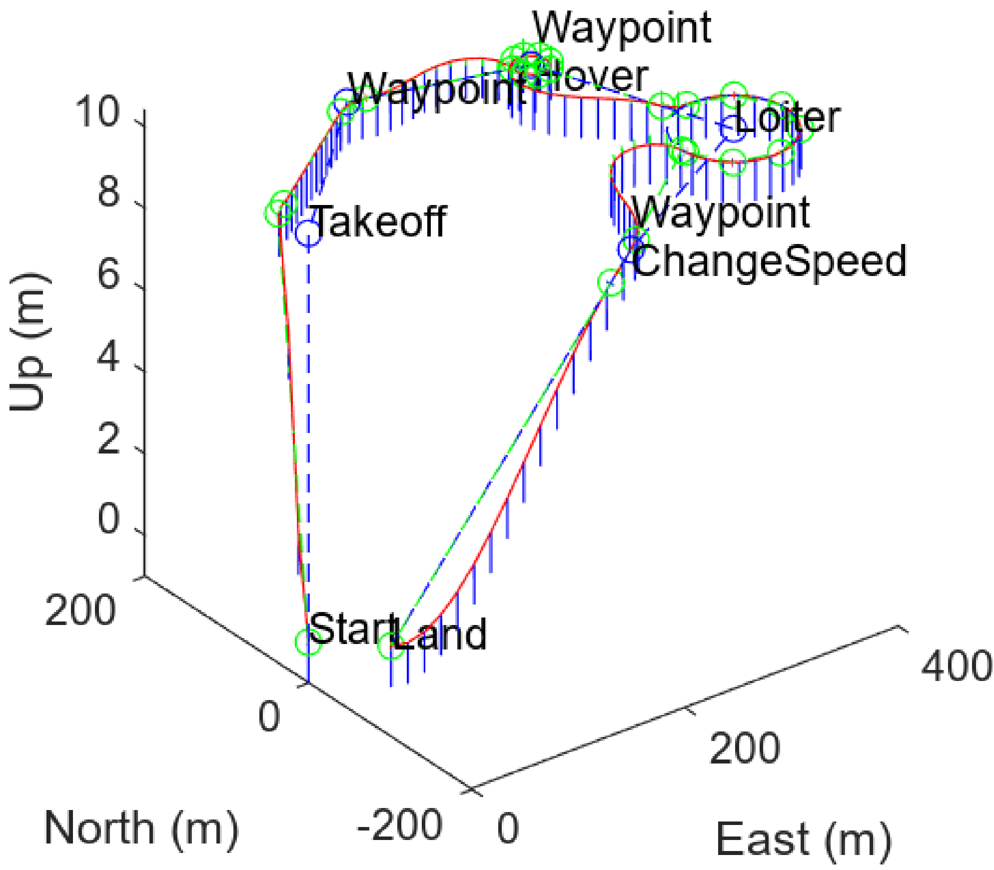


Plot the mission and flight trajectories overlapping.

```
figure
show(mission);
hold on
show(mrmTraj);
hold off
title("Mission Using Multirotor Trajectory")
axis square
```




```
show(mission);
hold on
show(fwmTraj);
hold off
title("Mission Using Fixed-Wing Trajectory")
axis square
```



Create UAV Mission Manually

Create a UAV mission object with a home location at the origin of the local ENU coordinate frame and an initial speed of 5 meters per second.

```
m = uavMission(Frame="LocalENU",HomeLocation=[0 0 0],Speed=5)
```

```
m = 
  uavMission with properties:

        HomeLocation: [0 0 0]
          InitialYaw: 0
               Frame: "LocalENU"
               Speed: 5
    NumMissionItems: 0
```

Add a takeoff mission item to the mission with an altitude of `25` meters, pitch of `15` degrees, and yaw of `0` degrees.

```
addTakeoff(m,20,Pitch=15,Yaw=0);
```

Add two waypoint mission items to the mission. Between the two waypoints, increase the speed of the UAV to 20 meters per second. After the second waypoint, reduce the speed of the UAV back to 5 meters per second.

```
addWaypoint(m,[10 0 30]);
addChangeSpeed(m,20)
addWaypoint(m,[20 0 40]);
addChangeSpeed(m,5)
addWaypoint(m,[30 0 50])
```

Add loiter and hover mission items to the mission, specifying for the UAV to loiter and hover around the second waypoint at a radius of 50 meters for 20 seconds each.

```
addLoiter(m,[40 0 60],10,20);
addHover(m,[50 0 70],10,20);
```

Add a landing mission item to the mission to land the UAV.

```
addLand(m,[70 0 0],Yaw=0);
```

Show the mission item data table.

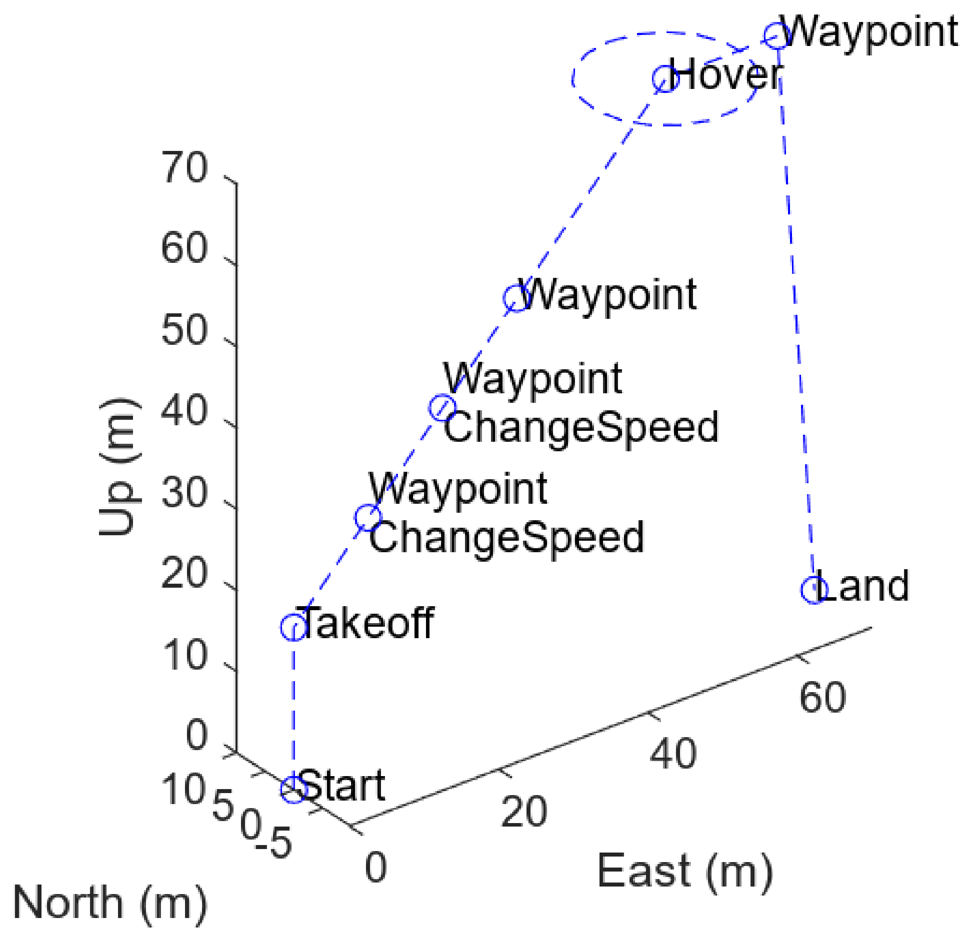
```
showdetails(m)
```

Remove the hover action at index 7, and then add another waypoint at index 8 after the hover item moves to index 7. Show the mission details table again to see the changes.

```
removeItem(m,7);
addWaypoint(m,[65 0 70],InsertAtRow=8);
showdetails(m)
```

Visualize the mission.

```
show(m);
axis equal
```



Execute Coverage Plan Using UAV Mission

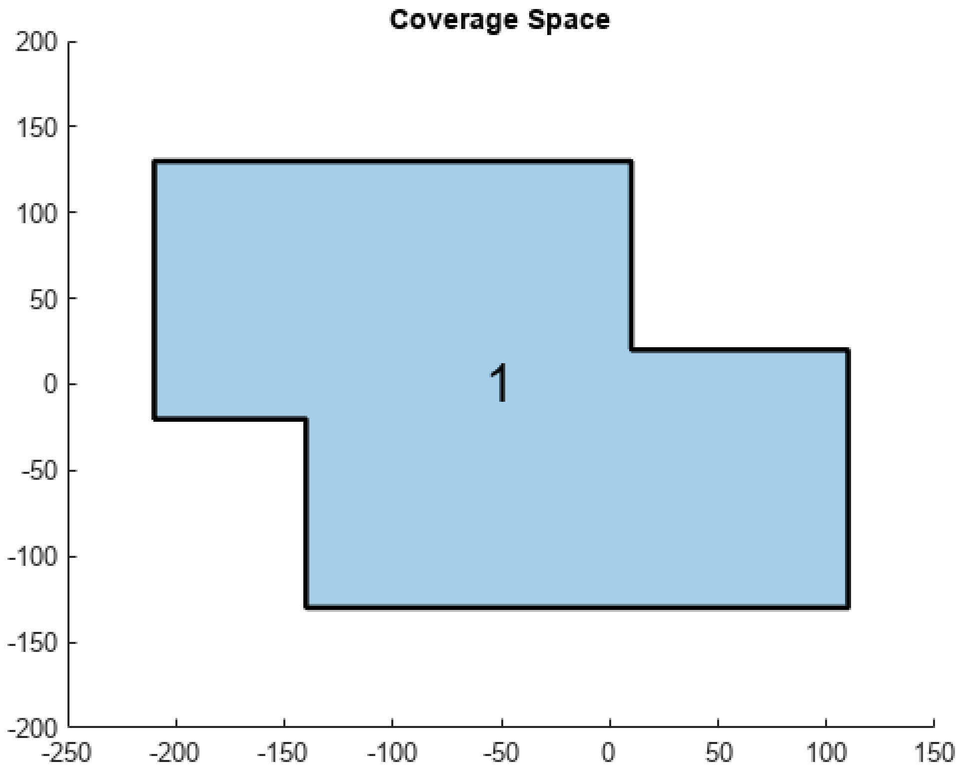
Initialize the settings to use for the coverage planner, coverage space, and mission. Set a coverage width to 65 meters, the region as polygon vertices, takeoff and landing locations, the UAV elevation during flight to 150 meters, and a geocenter.

```
coverageWidth = 65;
region = [-210 130; 10 130; 10 20; 110 20;
          110 -130; -140 -130; -140 -20; -210 -20];
takeoff = [-250 150 0];
landing = [0 -200 0];
uavElevation = 150;
geocenter = [-45 71 0];
```

Create the coverage space with those UAV coverage space settings.

```
cs = uavCoverageSpace(Polygons=region, ...
                      UnitWidth=coverageWidth, ...
```

```
                              ReferenceHeight=uavElevation, ...
                              ReferenceLocation=geocenter);

cs.show;
title("Coverage Space")
```

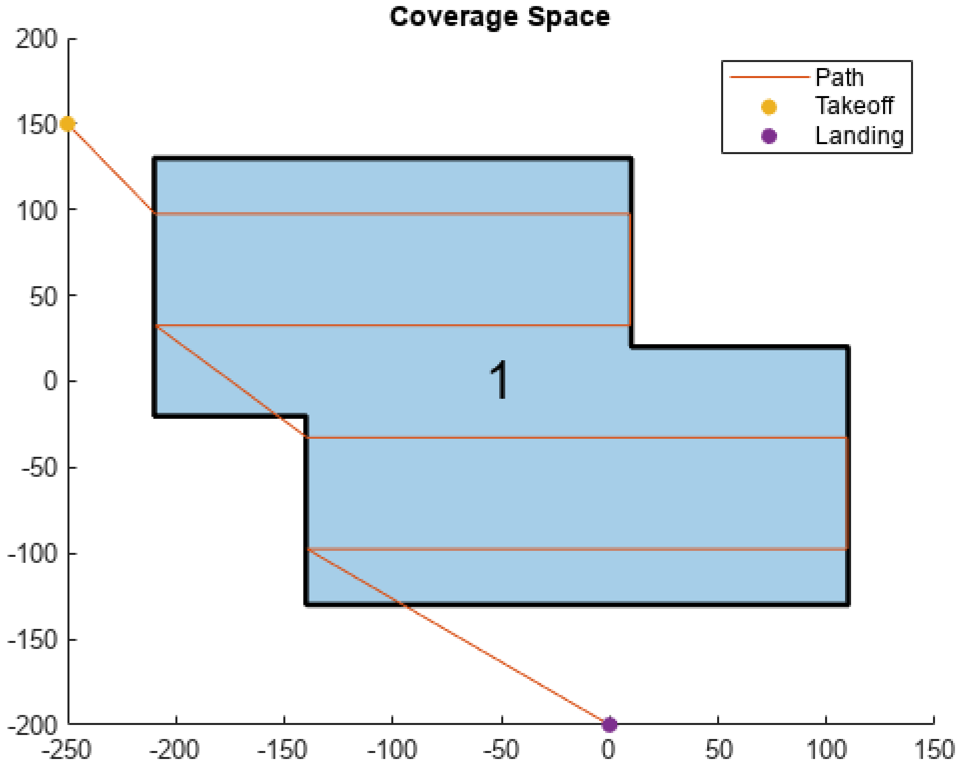


Create a coverage planner for the coverage space and plan the coverage path with the specified takeoff and landing locations.

```
cp = uavCoveragePlanner(cs);
[waypoints,solnInfo] = cp.plan(takeoff,landing);
```

Plot the waypoints, and the takeoff and landing locations on the coverage space.

```
hold on
plot(waypoints(:,1),waypoints(:,2))
scatter(takeoff(1),takeoff(2),"filled")
scatter(landing(1),landing(2),"filled")
legend("","Path","Takeoff","Landing")
hold off
```

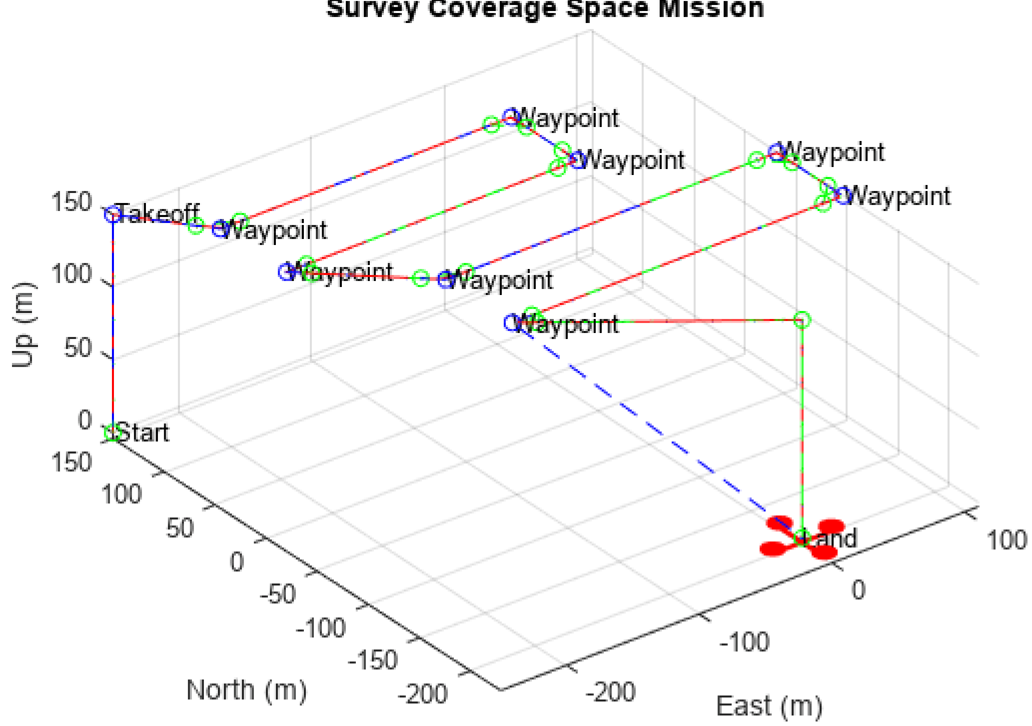


Export the waypoints to a `waypoints` file and create a UAV mission from that file with a speed of `10` meters per second and an initial yaw of `90` degrees.

```
exportWaypointsPlan(cp,solnInfo,"customCoverage.waypoints");
mission = uavMission(PlanFile="customCoverage.waypoints",Speed=10,InitialYaw=90);
```

Use the `exampleHelperSimulateUAVMission` helper function to visualize the UAV mission with a simulation time of 60 seconds.

```
exampleHelperSimulateUAVMission(mission,geocenter)
```



Version History

Introduced in R2022b

See Also

Objects

`fixedwingFlightTrajectory` | `fixedwingMissionParser` | `multirotorFlightTrajectory` | `multirotorMissionParser`

Topics

"Simulate UAV Mission in Urban Environment"

addChangeSpeed

Add change speed mission item

Syntax

```
addChangeSpeed(mission,speed)
addChangeSpeed( ___ ,Name=Value)
```

Description

`addChangeSpeed(mission,speed)` adds a change speed mission item that changes the reference speed of the UAV in subsequent mission items to `speed`.

`addChangeSpeed( ___ ,Name=Value)` sets additional options specified by one or more name-value pair arguments in addition to all input arguments from the previous syntax. For example, `addChangeSpeed(mission,speed,InsertAtRow=3)` adds a change speed mission item at index `3` of the mission items.

Examples

Create UAV Mission Manually

Create a UAV mission object with a home location at the origin of the local ENU coordinate frame and an initial speed of 5 meters per second.

```
m = uavMission(Frame="LocalENU",HomeLocation=[0 0 0],Speed=5)
```

```
m = 
  uavMission with properties:

       HomeLocation: [0 0 0]
         InitialYaw: 0
              Frame: "LocalENU"
              Speed: 5
    NumMissionItems: 0
```

Add a takeoff mission item to the mission with an altitude of `25` meters, pitch of `15` degrees, and yaw of `0` degrees.

```
addTakeoff(m,20,Pitch=15,Yaw=0);
```

Add two waypoint mission items to the mission. Between the two waypoints, increase the speed of the UAV to `20` meters per second. After the second waypoint, reduce the speed of the UAV back to `5` meters per second.

```
addWaypoint(m,[10 0 30]);
addChangeSpeed(m,20)
addWaypoint(m,[20 0 40]);
addChangeSpeed(m,5)
addWaypoint(m,[30 0 50])
```

Add loiter and hover mission items to the mission, specifying for the UAV to loiter and hover around the second waypoint at a radius of `50` meters for `20` seconds each.

```
addLoiter(m,[40 0 60],10,20);
addHover(m,[50 0 70],10,20);
```

Add a landing mission item to the mission to land the UAV.

```
addLand(m,[70 0 0],Yaw=0);
```

Show the mission item data table.

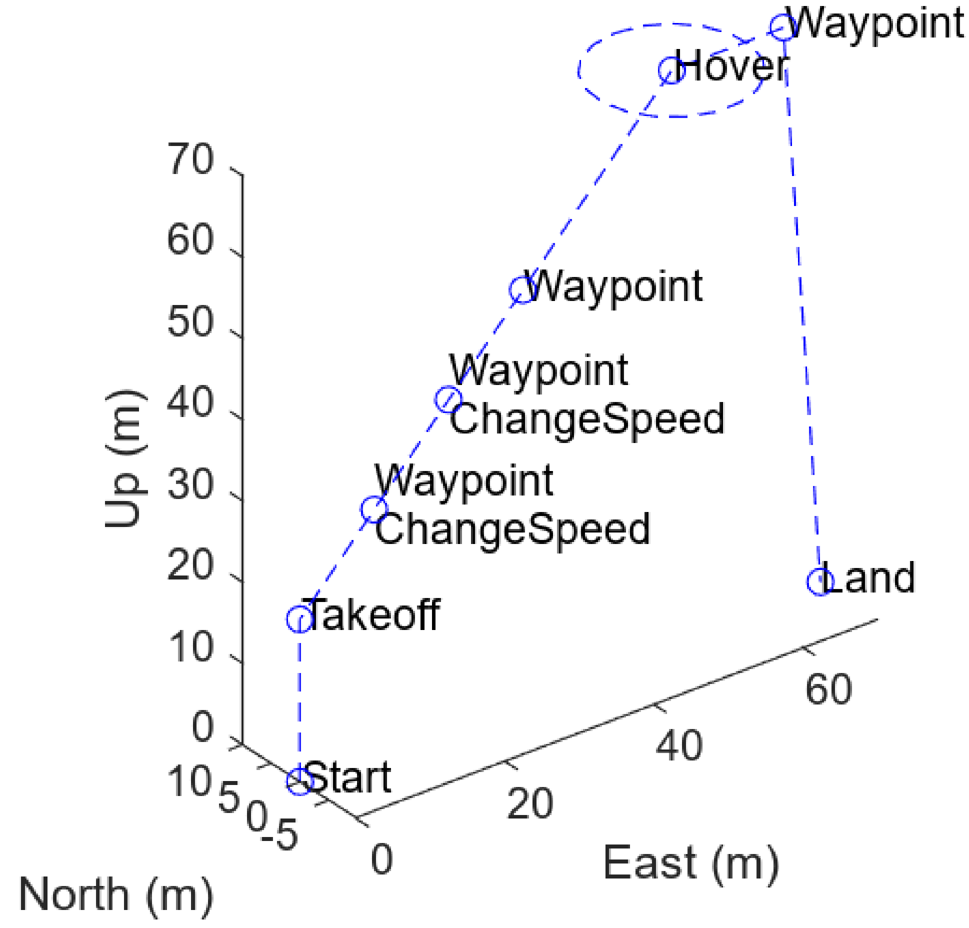
```
showdetails(m)
```

Remove the hover action at index `7`, and then add another waypoint at index `8` after the hover item moves to index `7`. Show the mission details table again to see the changes.

```
removeItem(m,7);
addWaypoint(m,[65 0 70],InsertAtRow=8);
showdetails(m)
```

Visualize the mission.

```
show(m);
axis equal
```



Input Arguments

`mission` — UAV mission
`uavMission` object

UAV mission, specified as a `uavMission` object.

`speed` — Reference speed of UAV
numeric scalar

Reference speed of the UAV for subsequent mission items, specified as a numeric scalar, in meters per second.

Example: 2

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Example: `addChangeSpeed(mission,2,InsertAtRow=3)`

`Timestamp` — Timestamp

numeric scalar

Timestamp of this item, specified as a numeric scalar, in seconds.

If you do not specify the `Timestamp` argument, `addChangeSpeed` determines the timestamp automatically based on the reference speed of the `uavMission` object.

`InsertAtRow` — Item position in mission order

positive integer

Item position in the mission order, specified as an positive integer.

If you do not specify the `Timestamp` argument or you specify a value that is greater than the total number of items in the mission, the item is inserted after the last item as the next index value.

Version History

Introduced in R2022b

See Also

Objects
`uavMission`

Functions
`addHover` | `addLand` | `addLoiter` | `addTakeoff` | `addWaypoint` | `removeItem` | `show` | `showdetails`

addHover

Add hover mission item

Syntax

```
addHover(mission,center,radius,duration)
addHover( ___ ,Name=Value)
```

Description

`addHover(mission,center,radius,duration)` adds a hover mission item that commands the UAV to hover around the specified center coordinate, `center`, at a radius `radius` for the specified duration `duration`.

`addHover( ___ ,Name=Value)` specifies options using one or more name-value arguments in addition to all input arguments from the previous syntax. For example, `addHover(mission,center,radius,duration,Frame="LocalNED")` adds a hover mission item with the center coordinates specified in the local north-east-down (NED) reference frame.

Examples

Create UAV Mission Manually

Create a UAV mission object with a home location at the origin of the local ENU coordinate frame and an initial speed of 5 meters per second.

```
m = uavMission(Frame="LocalENU",HomeLocation=[0 0 0],Speed=5)
```

```
m = 
  uavMission with properties:

       HomeLocation: [0 0 0]
         InitialYaw: 0
              Frame: "LocalENU"
              Speed: 5
    NumMissionItems: 0
```

Add a takeoff mission item to the mission with an altitude of `25` meters, pitch of `15` degrees, and yaw of `0` degrees.

```
addTakeoff(m,20,Pitch=15,Yaw=0);
```

Add two waypoint mission items to the mission. Between the two waypoints, increase the speed of the UAV to `20` meters per second. After the second waypoint, reduce the speed of the UAV back to `5` meters per second.

```
addWaypoint(m,[10 0 30]);
addChangeSpeed(m,20)
addWaypoint(m,[20 0 40]);
```

```
addChangeSpeed(m,5)
addWaypoint(m,[30 0 50])
```

Add loiter and hover mission items to the mission, specifying for the UAV to loiter and hover around the second waypoint at a radius of 50 meters for 20 seconds each.

```
addLoiter(m,[40 0 60],10,20);
addHover(m,[50 0 70],10,20);
```

Add a landing mission item to the mission to land the UAV.

```
addLand(m,[70 0 0],Yaw=0);
```

Show the mission item data table.

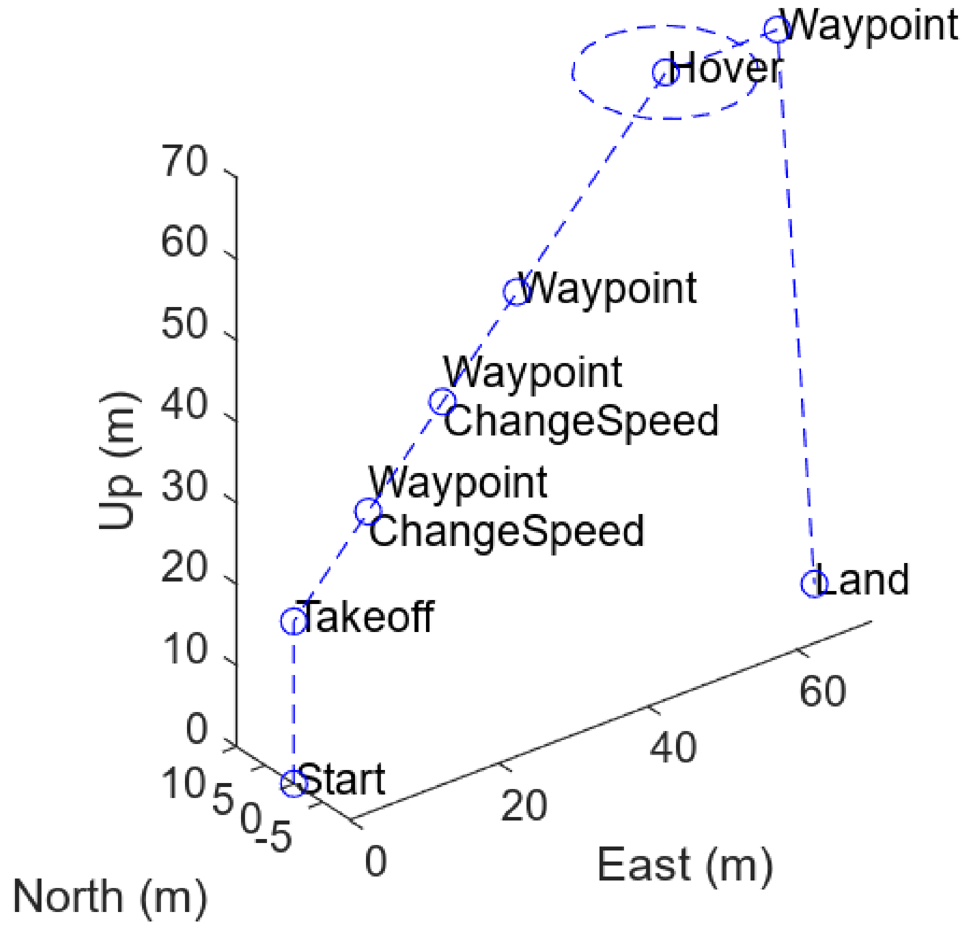
```
showdetails(m)
```

Remove the hover action at index 7, and then add another waypoint at index 8 after the hover item moves to index 7. Show the mission details table again to see the changes.

```
removeItem(m,7);
addWaypoint(m,[65 0 70],InsertAtRow=8);
showdetails(m)
```

Visualize the mission.

```
show(m);
axis equal
```



Input Arguments

`mission` — UAV mission
`uavMission` object

UAV mission, specified as a `uavMission` object.

`center` — Center coordinate
three-element row vector

Center coordinate, specified as a three-element row vector. The form of the vector depends on the reference frame, as specified by either the `uavMission` object `mission` or the `Frame` argument:

- `"Global"` — [*latitude longitude altitude*]
- `"GlobalRelativeAlt"` — [*latitude longitude relativeAltitude*]. *relativeAltitude* is the altitude relative to the home location specified by `mission`.
- `"LocalENU"` — [*X Y Z*]

- `"LocalNED"` — [*X Y Z*]

Specify latitude and longitude in degrees, and altitude or relative altitude in meters. Local coordinates are in meters.

Example: `[40 20 40]`

`radius` — Hover radius
nonnegative scalar

Hover radius, specified as a nonnegative scalar, in meters.

Example: `3`

`duration` — Hover duration
nonnegative scalar

Hover duration, specified as a nonnegative scalar, in seconds.

Example: `5`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Example: `addHover(mission,[30 60 10],10,5,Frame="LocalNED")`

`Frame` — Mission waypoint reference frame
`mission.Frame` (default) | `"Global"` | `"GlobalRelativeAlt"` | `"LocalENU"` | `"LocalNED"`

Mission waypoint reference frame, specified as one of these values:

- `"Global"` — Global reference frame
- `"GlobalRelativeAlt"` — Global reference frame with an altitude relative to the home location specified by `mission`
- `"LocalENU"` — Local east-north-up (ENU) reference frame
- `"LocalNED"` — Local north-east-down (NED) reference frame

If you do not specify the Frame argument, `addHover` uses the reference frame of the `uavMission` object `mission`.

Data Types: `char` | `string`

`Timestamp` — Timestamp
numeric scalar

Timestamp of this item, specified as a numeric scalar, in seconds.

If you do not specify the `Timestamp` argument, `addHover` determines the timestamp automatically based on the reference speed of the `uavMission` object.

`InsertAtRow` — Item position in mission order
positive integer

Item position in the mission order, specified as an positive integer.

If you do not specify the `Timestamp` argument or you specify a value that is greater than the total number of items in the mission, the item is inserted after the last item as the next index value.

Version History

Introduced in R2022b

See Also

Objects
`uavMission`

Functions
`addChangeSpeed` | `addLand` | `addLoiter` | `addTakeoff` | `addWaypoint` | `removeItem` | `show` | `showdetails`

addLand

Add landing mission item

Syntax

```
addLand(mission,waypoint)
addLand( ___ ,Name=Value)
```

Description

`addLand(mission,waypoint)` adds a landing mission item that commands the UAV to land at the specified position `waypoint`.

`addLand( ___ ,Name=Value)` sets additional options specified by one or more name-value arguments in addition to all input arguments from the previous. For example, `addLand(mission,waypoint,Frame="LocalNED")` adds a landing mission item with the position specified in the local north-east-down (NED) reference frame.

Examples

Create UAV Mission Manually

Create a UAV mission object with a home location at the origin of the local ENU coordinate frame and an initial speed of 5 meters per second.

```
m = uavMission(Frame="LocalENU",HomeLocation=[0 0 0],Speed=5)
```

```
m = 
  uavMission with properties:

       HomeLocation: [0 0 0]
         InitialYaw: 0
              Frame: "LocalENU"
              Speed: 5
    NumMissionItems: 0
```

Add a takeoff mission item to the mission with an altitude of `25` meters, pitch of `15` degrees, and yaw of `0` degrees.

```
addTakeoff(m,20,Pitch=15,Yaw=0);
```

Add two waypoint mission items to the mission. Between the two waypoints, increase the speed of the UAV to `20` meters per second. After the second waypoint, reduce the speed of the UAV back to `5` meters per second.

```
addWaypoint(m,[10 0 30]);
addChangeSpeed(m,20)
addWaypoint(m,[20 0 40]);
addChangeSpeed(m,5)
addWaypoint(m,[30 0 50])
```

Add loiter and hover mission items to the mission, specifying for the UAV to loiter and hover around the second waypoint at a radius of 50 meters for 20 seconds each.

```
addLoiter(m,[40 0 60],10,20);
addHover(m,[50 0 70],10,20);
```

Add a landing mission item to the mission to land the UAV.

```
addLand(m,[70 0 0],Yaw=0);
```

Show the mission item data table.

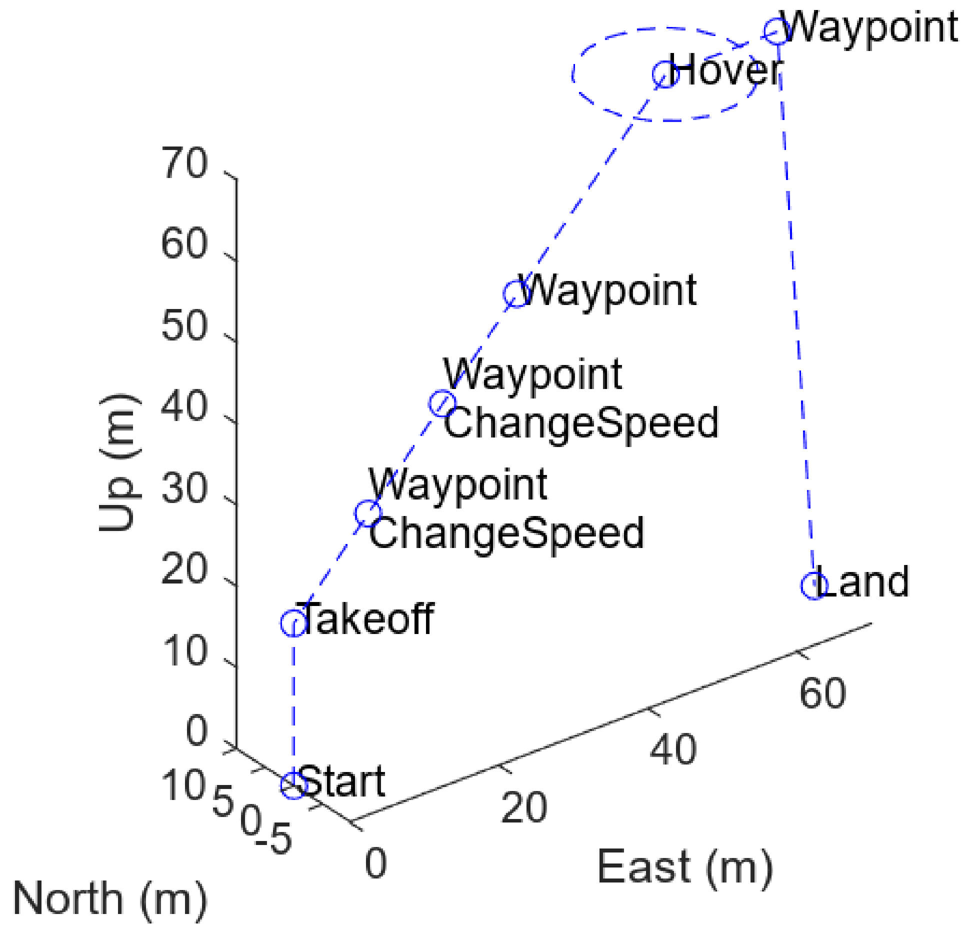
```
showdetails(m)
```

Remove the hover action at index 7, and then add another waypoint at index 8 after the hover item moves to index 7. Show the mission details table again to see the changes.

```
removeItem(m,7);
addWaypoint(m,[65 0 70],InsertAtRow=8);
showdetails(m)
```

Visualize the mission.

```
show(m);
axis equal
```



Input Arguments

`mission` — UAV mission
`uavMission` object

UAV mission, specified as a `uavMission` object.

`waypoint` — Landing location
three-element row vector

Landing location, specified as a three-element row vector. The form of the vector depends on the reference frame, as specified by either the `uavMission` object `mission` or the `Frame` argument:

- `"Global"` — [*latitude longitude altitude*]
- `"GlobalRelativeAlt"` — [*latitude longitude relativeAltitude*]. *relativeAltitude* is the altitude relative to the home location specified by `mission`.
- `"LocalENU"` — [*X Y Z*]

- "LocalNED" — [X Y Z]

Specify latitude and longitude in degrees, and altitude or relative altitude in meters. Local coordinates are in meters.

Example: [0 2 20]

Name-Value Pair Arguments

Specify optional pairs of arguments as Name1=Value1,...,NameN=ValueN, where Name is the argument name and Value is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Example: addLand(mission,[42 70 10],Yaw=15,Frame="LocalNED")

Yaw — Desired yaw
NaN (default) | numeric scalar

Desired yaw, specified as a numeric scalar, in degrees.

Frame — Mission waypoint reference frame
mission.Frame (default) | "Global" | "GlobalRelativeAlt" | "LocalENU" | "LocalNED"

Mission waypoint reference frame, specified as one of these values:

- "Global" — Global reference frame
- "GlobalRelativeAlt" — Global reference frame with an altitude relative to the home location specified by mission
- "LocalENU" — Local east-north-up (ENU) reference frame
- "LocalNED" — Local north-east-down (NED) reference frame

If you do not specify the Frame argument, addLand uses the reference frame of the uavMission object mission.

Data Types: char | string

Timestamp — Timestamp
numeric scalar

Timestamp of this item, specified as a numeric scalar, in seconds.

If you do not specify the Timestamp argument, addLand determines the timestamp automatically based on the reference speed of the uavMission object.

InsertAtRow — Item position in mission order
positive integer

Item position in the mission order, specified as an positive integer.

If you do not specify the Timestamp argument or you specify a value that is greater than the total number of items in the mission, the item is inserted after the last item as the next index value.

Version History

Introduced in R2022b

See Also

Objects
uavMission

Functions
addChangeSpeed | addHover | addLoiter | addTakeoff | addWaypoint | removeItem | show | showdetails

addLoiter

Add loiter mission item

Syntax

```
addLoiter(mission,center,radius,duration)
addLoiter( ___ ,Name=Value)
```

Description

`addLoiter(mission,center,radius,duration)` adds a loiter mission item to command the UAV to loiter around a center coordinate `center`, with a radius `radius` for a duration `duration`.

`addLoiter( ___ ,Name=Value)` sets additional options specified by one or more name-value pair arguments in addition to all input arguments from the previous syntax. For example, `addLoiter(mission,center,radius,duration,Frame="LocalNED")` adds a loiter mission item with the center coordinates specified in the local north-east-down (NED) reference frame.

Examples

Create UAV Mission Manually

Create a UAV mission object with a home location at the origin of the local ENU coordinate frame and an initial speed of 5 meters per second.

```
m = uavMission(Frame="LocalENU",HomeLocation=[0 0 0],Speed=5)
```

```
m = 
  uavMission with properties:

       HomeLocation: [0 0 0]
         InitialYaw: 0
              Frame: "LocalENU"
              Speed: 5
    NumMissionItems: 0
```

Add a takeoff mission item to the mission with an altitude of `25` meters, pitch of `15` degrees, and yaw of `0` degrees.

```
addTakeoff(m,20,Pitch=15,Yaw=0);
```

Add two waypoint mission items to the mission. Between the two waypoints, increase the speed of the UAV to `20` meters per second. After the second waypoint, reduce the speed of the UAV back to `5` meters per second.

```
addWaypoint(m,[10 0 30]);
addChangeSpeed(m,20)
addWaypoint(m,[20 0 40]);
addChangeSpeed(m,5)
addWaypoint(m,[30 0 50])
```

Add loiter and hover mission items to the mission, specifying for the UAV to loiter and hover around the second waypoint at a radius of 50 meters for 20 seconds each.

```
addLoiter(m,[40 0 60],10,20);
addHover(m,[50 0 70],10,20);
```

Add a landing mission item to the mission to land the UAV.

```
addLand(m,[70 0 0],Yaw=0);
```

Show the mission item data table.

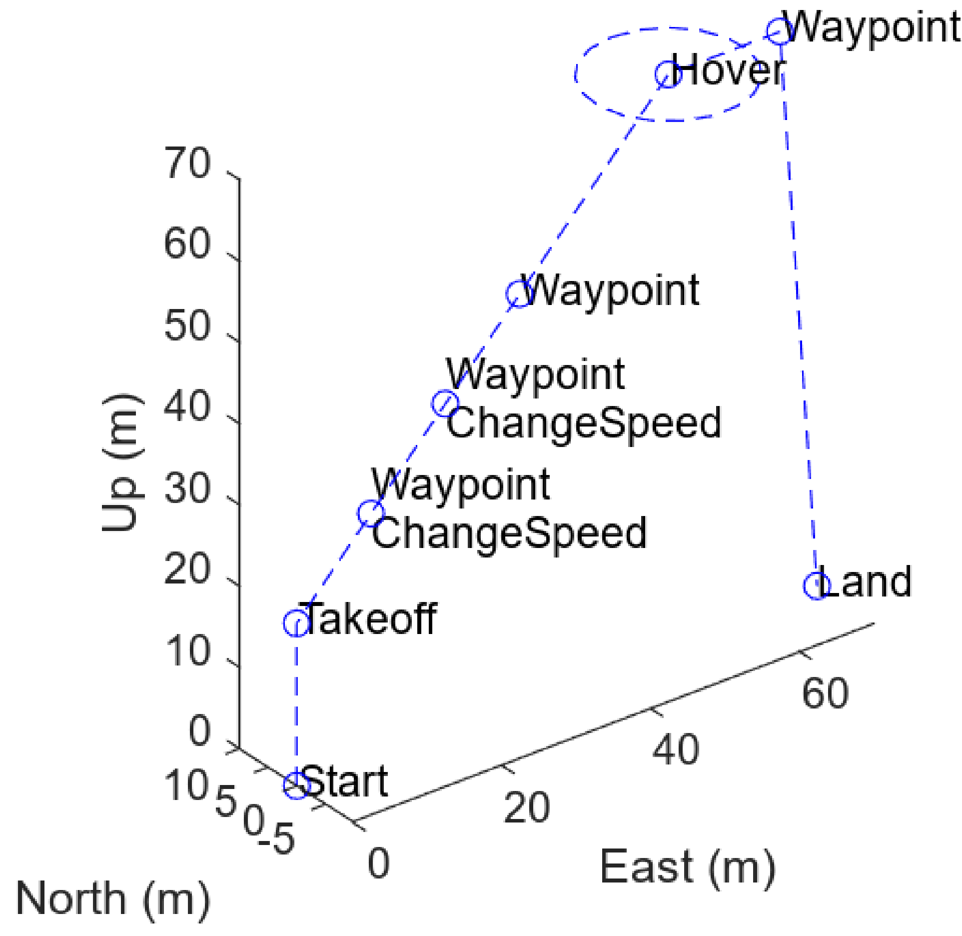
```
showdetails(m)
```

Remove the hover action at index 7, and then add another waypoint at index 8 after the hover item moves to index 7. Show the mission details table again to see the changes.

```
removeItem(m,7);
addWaypoint(m,[65 0 70],InsertAtRow=8);
showdetails(m)
```

Visualize the mission.

```
show(m);
axis equal
```



Input Arguments

`mission` — UAV mission
`uavMission` object

UAV mission, specified as a `uavMission` object.

`center` — Center coordinate
three-element row vector

Center coordinate, specified as a three-element row vector. The form of the vector depends on the reference frame, as specified by either the `uavMission` object `mission` or the `Frame` argument:

- `"Global"` — [*latitude longitude altitude*]
- `"GlobalRelativeAlt"` — [*latitude longitude relativeAltitude*]. *relativeAltitude* is the altitude relative to the home location specified by `mission`.
- `"LocalENU"` — [*X Y Z*]

- "LocalNED" — [*X Y Z*]

Specify latitude and longitude in degrees, and altitude or relative altitude in meters. Local coordinates are in meters.

Example: [40 20 40]

radius — Loiter radius
nonnegative numeric scalar

Loiter radius, specified as a nonnegative numeric scalar, in meters.

Example: 3

duration — Loiter duration
nonnegative numeric scalar

Loiter duration, specified as a nonnegative numeric scalar, in seconds.

Example: 5

Name-Value Pair Arguments

Specify optional pairs of arguments as Name1=Value1,...,NameN=ValueN, where Name is the argument name and Value is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Example: addLoiter(mission,[30 60 10],10,5,Frame="LocalNED")

Frame — Mission waypoint reference frame
mission.Frame (default) | "Global" | "GlobalRelativeAlt" | "LocalENU" | "LocalNED"

Mission waypoint reference frame, specified as one of these values:

- "Global" — Global reference frame
- "GlobalRelativeAlt" — Global reference frame with an altitude relative to the home location specified by mission
- "LocalENU" — Local east-north-up (ENU) reference frame
- "LocalNED" — Local north-east-down (NED) reference frame

If you do not specify the Frame argument, addLoiter uses the reference frame of the uavMission object mission.

Data Types: char | string

Timestamp — Timestamp
numeric scalar

Timestamp of this item, specified as a numeric scalar, in seconds.

If you do not specify the Timestamp argument, addLoiter determines the timestamp automatically based on the reference speed of the uavMission object.

InsertAtRow — Item position in mission order
positive integer

Item position in the mission order, specified as an positive integer.

If you do not specify the `Timestamp` argument or you specify a value that is greater than the total number of items in the mission, the item is inserted after the last item as the next index value.

Version History

Introduced in R2022b

See Also

Objects
`uavMission`

Functions
`addChangeSpeed` | `addHover` | `addLand` | `addTakeoff` | `addWaypoint` | `removeItem` | `show` | `showdetails`

addTakeoff

Add takeoff mission item

Syntax

```
addTakeoff(mission,altitude)
addTakeoff( ___ ,Name=Value)
```

Description

`addTakeoff(mission,altitude)` adds a takeoff mission item that commands a UAV to takeoff and ascend to the specified altitude `altitude`.

`addTakeoff( ___ ,Name=Value)` sets additional options specified by one or more name-value pair arguments in addition to all input arguments from the previous syntax. For example, `addTakeoff(mission,altitude,Frame="LocalNED")` adds a takeoff mission item with the altitude specified in the local north-east-down (NED) reference frame.

Examples

Create UAV Mission Manually

Create a UAV mission object with a home location at the origin of the local ENU coordinate frame and an initial speed of 5 meters per second.

```
m = uavMission(Frame="LocalENU",HomeLocation=[0 0 0],Speed=5)
```

```
m = 
  uavMission with properties:

       HomeLocation: [0 0 0]
         InitialYaw: 0
              Frame: "LocalENU"
              Speed: 5
    NumMissionItems: 0
```

Add a takeoff mission item to the mission with an altitude of `25` meters, pitch of `15` degrees, and yaw of `0` degrees.

```
addTakeoff(m,20,Pitch=15,Yaw=0);
```

Add two waypoint mission items to the mission. Between the two waypoints, increase the speed of the UAV to `20` meters per second. After the second waypoint, reduce the speed of the UAV back to `5` meters per second.

```
addWaypoint(m,[10 0 30]);
addChangeSpeed(m,20)
addWaypoint(m,[20 0 40]);
addChangeSpeed(m,5)
addWaypoint(m,[30 0 50])
```

Add loiter and hover mission items to the mission, specifying for the UAV to loiter and hover around the second waypoint at a radius of 50 meters for 20 seconds each.

```
addLoiter(m,[40 0 60],10,20);
addHover(m,[50 0 70],10,20);
```

Add a landing mission item to the mission to land the UAV.

```
addLand(m,[70 0 0],Yaw=0);
```

Show the mission item data table.

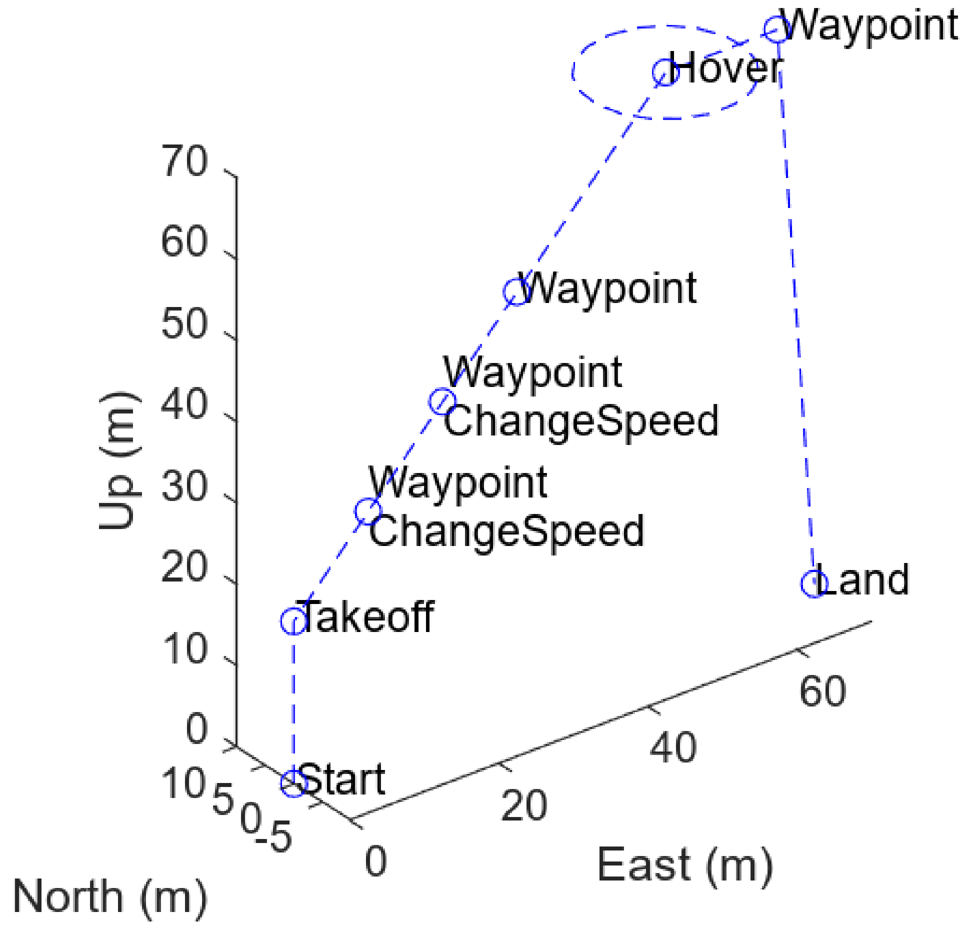
```
showdetails(m)
```

Remove the hover action at index 7, and then add another waypoint at index 8 after the hover item moves to index 7. Show the mission details table again to see the changes.

```
removeItem(m,7);
addWaypoint(m,[65 0 70],InsertAtRow=8);
showdetails(m)
```

Visualize the mission.

```
show(m);
axis equal
```



Input Arguments

`mission` — UAV mission
`uavMission` object

UAV mission, specified as a `uavMission` object.

`altitude` — Desired altitude
numeric scalar

Desired altitude, specified as a numeric scalar, in meters.

If the mission waypoint reference frame is `Global`, this value specifies an absolute altitude above the WGS84 reference ellipsoid. Otherwise, this value specifies altitude relative to the home location of the mission.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Example: `addTakeoff(mission,5,Pitch=15,Frame="LocalNED")`

`Pitch` — Desired pitch
`NaN` (default) | nonnegative scalar

Desired pitch, specified as a nonnegative numeric scalar, in degrees.

Data Types: `double`

`Yaw` — Desired yaw
`NaN` (default) | numeric scalar

Desired yaw, specified as a numeric scalar, in degrees.

`Frame` — Mission waypoint reference frame
`mission.Frame` (default) | `"Global"` | `"GlobalRelativeAlt"` | `"LocalENU"` | `"LocalNED"`

Mission waypoint reference frame, specified as one of these values:

- `"Global"` — Global reference frame
- `"GlobalRelativeAlt"` — Global reference frame with an altitude relative to the home location specified by `mission`
- `"LocalENU"` — Local east-north-up (ENU) reference frame
- `"LocalNED"` — Local north-east-down (NED) reference frame

If you do not specify the Frame argument, `addTakeoff` uses the reference frame of the `uavMission` object `mission`.

Data Types: `char` | `string`

`Timestamp` — Timestamp
numeric scalar

Timestamp of this item, specified as a numeric scalar, in seconds.

If you do not specify the `Timestamp` argument, `addTakeoff` determines the timestamp automatically based on the reference speed of the `uavMission` object.

`InsertAtRow` — Item position in mission order
positive integer

Item position in the mission order, specified as an positive integer.

If you do not specify the `Timestamp` argument or you specify a value that is greater than the total number of items in the mission, the item is inserted after the last item as the next index value.

Version History

Introduced in R2022b

See Also

Objects
`uavMission`

Functions
`addChangeSpeed` | `addHover` | `addLand` | `addLoiter` | `addWaypoint` | `removeItem` | `show` | `showdetails`

addWaypoint

Add waypoint mission item

Syntax

```
addWaypoint(mission,waypoint)
addWaypoint( ___ ,Name=Value)
```

Description

`addWaypoint(mission,waypoint)` adds a waypoint mission item that commands the UAV to travel to the specified waypoint `waypoint`.

`addWaypoint( ___ ,Name=Value)` sets additional options specified by one or more name-value pair arguments in addition to all input arguments from the previous syntax. For example, `addWaypoint(mission,waypoints,Frame="LocalNED")` adds a waypoint mission item with the coordinates specified in the local north-east-down (NED) reference frame.

Examples

Create UAV Mission Manually

Create a UAV mission object with a home location at the origin of the local ENU coordinate frame and an initial speed of 5 meters per second.

```
m = uavMission(Frame="LocalENU",HomeLocation=[0 0 0],Speed=5)
```

```
m = 
  uavMission with properties:

       HomeLocation: [0 0 0]
         InitialYaw: 0
              Frame: "LocalENU"
              Speed: 5
    NumMissionItems: 0
```

Add a takeoff mission item to the mission with an altitude of `25` meters, pitch of `15` degrees, and yaw of `0` degrees.

```
addTakeoff(m,20,Pitch=15,Yaw=0);
```

Add two waypoint mission items to the mission. Between the two waypoints, increase the speed of the UAV to `20` meters per second. After the second waypoint, reduce the speed of the UAV back to `5` meters per second.

```
addWaypoint(m,[10 0 30]);
addChangeSpeed(m,20)
addWaypoint(m,[20 0 40]);
addChangeSpeed(m,5)
addWaypoint(m,[30 0 50])
```

Add loiter and hover mission items to the mission, specifying for the UAV to loiter and hover around the second waypoint at a radius of `50` meters for `20` seconds each.

```
addLoiter(m,[40 0 60],10,20);
addHover(m,[50 0 70],10,20);
```

Add a landing mission item to the mission to land the UAV.

```
addLand(m,[70 0 0],Yaw=0);
```

Show the mission item data table.

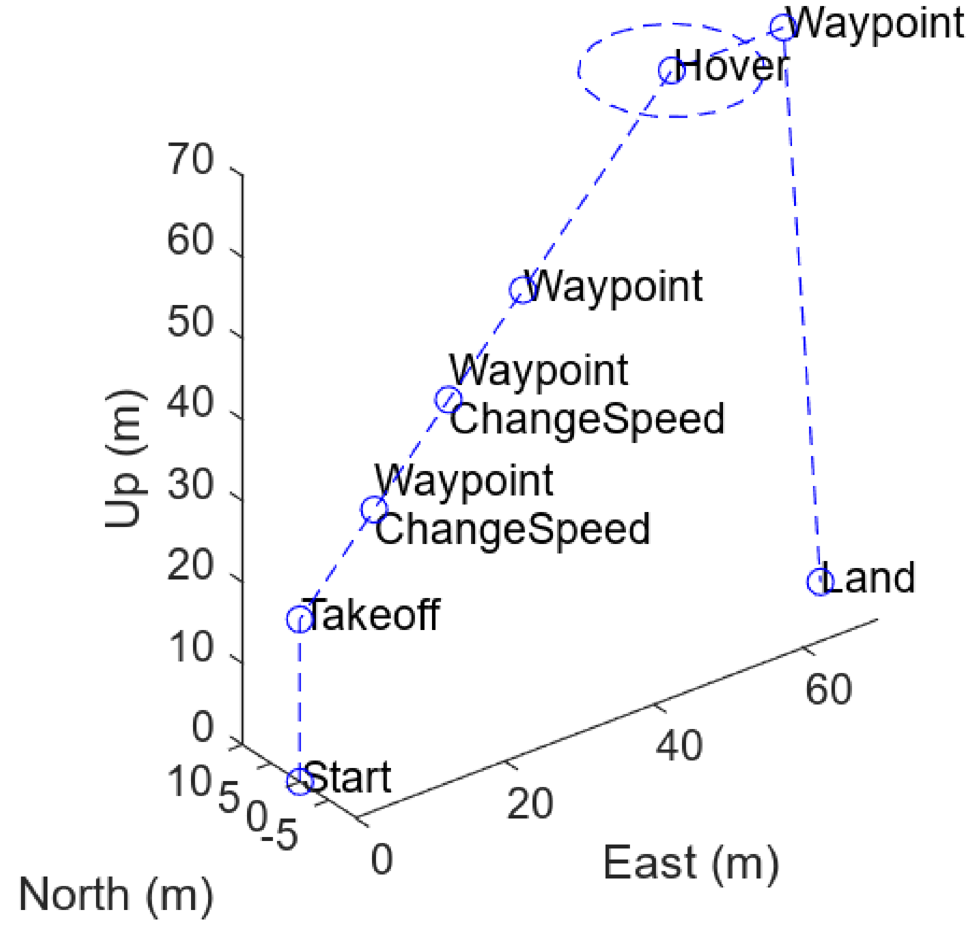
```
showdetails(m)
```

Remove the hover action at index `7`, and then add another waypoint at index `8` after the hover item moves to index `7`. Show the mission details table again to see the changes.

```
removeItem(m,7);
addWaypoint(m,[65 0 70],InsertAtRow=8);
showdetails(m)
```

Visualize the mission.

```
show(m);
axis equal
```



Input Arguments

mission — UAV mission
uavMission object

UAV mission, specified as a uavMission object.

waypoint — Desired waypoint
three-element row vector

Desired waypoint, specified as a three-element row vector. The form of the vector depends on the reference frame, as specified by either the uavMission object mission or the Frame argument:

- "Global" — [*latitude longitude altitude*]
- "GlobalRelativeAlt" — [*latitude longitude relativeAltitude*]. *relativeAltitude* is the altitude relative to the home location specified by mission.
- "LocalENU" — [*X Y Z*]

- `"LocalNED"` — [*X Y Z*]

Specify latitude and longitude in degrees, and altitude or relative altitude in meters. Local coordinates are in meters.

Example: `[0 2 20]`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Example: `addWaypoint(mission,[42 70 10],Yaw=15,Frame="LocalNED")`

`AcceptanceRadius` — Acceptance radius
positive numeric scalar

Acceptance radius, specified as positive numeric scalar, in meters. This value specifies the maximum distance from the waypoint, in any direction, at which the `uavMission` object considers the UAV to have reached the waypoint

Data Types: `double`

`Yaw` — Desired yaw
`NaN` (default) | numeric scalar

Desired yaw, specified as a numeric scalar, in degrees.

`Frame` — Mission waypoint reference frame
`mission.Frame` (default) | `"Global"` | `"GlobalRelativeAlt"` | `"LocalENU"` | `"LocalNED"`

Mission waypoint reference frame, specified as one of these values:

- `"Global"` — Global reference frame
- `"GlobalRelativeAlt"` — Global reference frame with an altitude relative to the home location specified by `mission`
- `"LocalENU"` — Local east-north-up (ENU) reference frame
- `"LocalNED"` — Local north-east-down (NED) reference frame

If you do not specify the Frame argument, `addWaypoint` uses the reference frame of the `uavMission` object `mission`.

Data Types: `char` | `string`

`Timestamp` — Timestamp
numeric scalar

Timestamp of this item, specified as a numeric scalar, in seconds.

If you do not specify the `Timestamp` argument, `addWaypoint` determines the timestamp automatically based on the reference speed of the `uavMission` object.

`InsertAtRow` — Item position in mission order
positive integer

Item position in the mission order, specified as an positive integer.

If you do not specify the `Timestamp` argument or you specify a value that is greater than the total number of items in the mission, the item is inserted after the last item as the next index value.

Version History

Introduced in R2022b

See Also

Objects
`uavMission`

Functions
`addChangeSpeed` | `addHover` | `addLand` | `addLoiter` | `addTakeoff` | `removeItem` | `show` | `showdetails`

copy

Copy UAV Mission

Syntax

`missionCopy = copy(mission)`

Description

`missionCopy = copy(mission)` creates a deep copy of the `uavMission` object `mission` with the same properties.

Input Arguments

`mission` — UAV mission
`uavMission` object

UAV mission, specified as a `uavMission` object.

Output Arguments

`missionCopy` — Deep copy of UAV mission
`uavMission` object

Deep copy of the UAV mission `mission`, returned as a `uavMission` object with the same properties.

Version History

Introduced in R2022b

See Also

`uavMission`

removeItem

Remove mission items at specified indices

Syntax

```
removeItem(mission,idx)
```

Description

`removeItem(mission,idx)` removes items at specified indices `idx`.

Examples

Create UAV Mission Manually

Create a UAV mission object with a home location at the origin of the local ENU coordinate frame and an initial speed of 5 meters per second.

```
m = uavMission(Frame="LocalENU",HomeLocation=[0 0 0],Speed=5)
```

```
m = 
  uavMission with properties:

       HomeLocation: [0 0 0]
         InitialYaw: 0
              Frame: "LocalENU"
              Speed: 5
    NumMissionItems: 0
```

Add a takeoff mission item to the mission with an altitude of `25` meters, pitch of `15` degrees, and yaw of `0` degrees.

```
addTakeoff(m,20,Pitch=15,Yaw=0);
```

Add two waypoint mission items to the mission. Between the two waypoints, increase the speed of the UAV to `20` meters per second. After the second waypoint, reduce the speed of the UAV back to `5` meters per second.

```
addWaypoint(m,[10 0 30]);
addChangeSpeed(m,20)
addWaypoint(m,[20 0 40]);
addChangeSpeed(m,5)
addWaypoint(m,[30 0 50])
```

Add loiter and hover mission items to the mission, specifying for the UAV to loiter and hover around the second waypoint at a radius of `50` meters for `20` seconds each.

```
addLoiter(m,[40 0 60],10,20);
addHover(m,[50 0 70],10,20);
```

Add a landing mission item to the mission to land the UAV.

```
addLand(m,[70 0 0],Yaw=0);
```

Show the mission item data table.

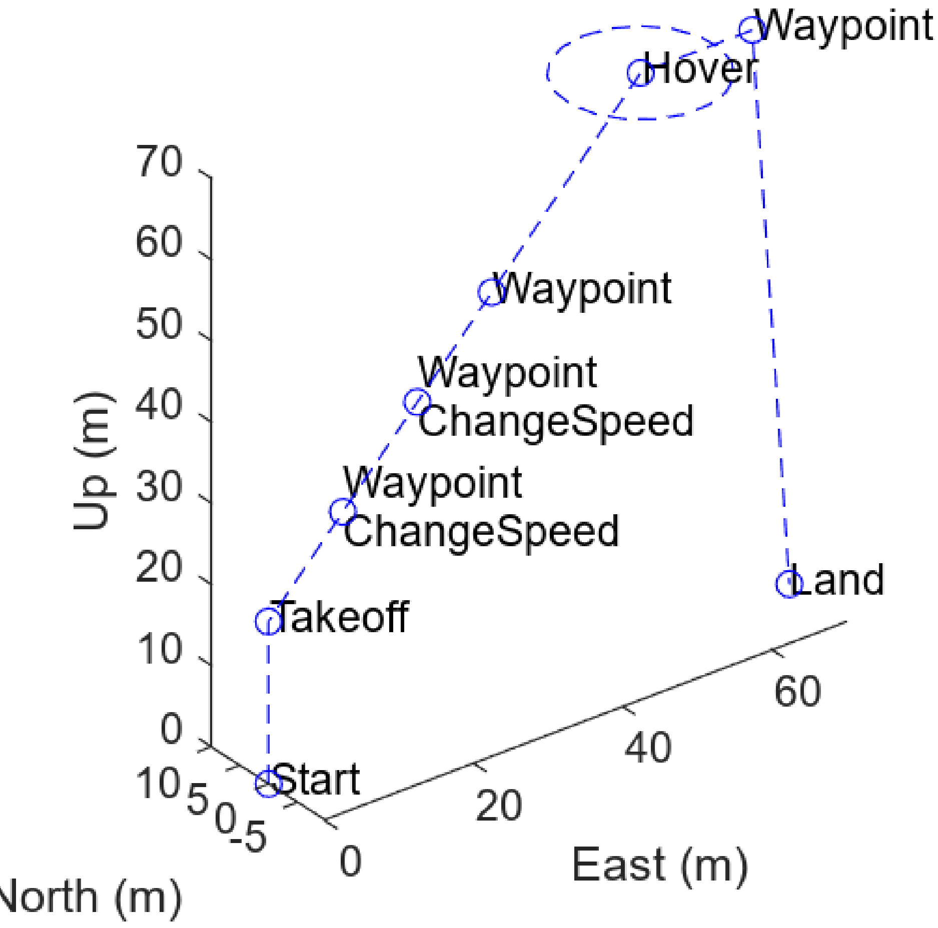
```
showdetails(m)
```

Remove the hover action at index 7, and then add another waypoint at index 8 after the hover item moves to index 7. Show the mission details table again to see the changes.

```
removeItem(m,7);
addWaypoint(m,[65 0 70],InsertAtRow=8);
showdetails(m)
```

Visualize the mission.

```
show(m);
axis equal
```



Input Arguments

`mission` — UAV mission

`uavMission` object

UAV mission, specified as a `uavMission` object.

`idx` — Indices of items to remove

N-element vector of integers in the range [1, `size(mission.NumMissionItems)`]

Indices of items to remove, specified as an *N*-element vector of integers in the range [1, `size(mission.NumMissionItems)`], where *N* is the number of items to remove.

Example: `[1 3]`

Version History

Introduced in R2022b

See Also

Objects

`uavMission`

Functions

`addChangeSpeed` | `addHover` | `addLand` | `addLoiter` | `addTakeoff` | `addWaypoint` | `show` | `showdetails`

show

Visualize UAV mission

Syntax

```
AX = show(mission)
AX = show( ___ ,Name=Value)
```

Description

`AX = show(mission)` visualizes the UAV mission in a figure in the local east-north-up (ENU) frame, and returns the axes handle `AX` that contains the plot.

`AX = show( ___ ,Name=Value)` sets additional options specified by one or more name-value pair arguments in addition to all input arguments from the previous syntax. For example, `show(mission,ReferenceLocation=[0 0 0])` shows the mission with the reference location set to `[0 0 0]`.

Examples

Create UAV Mission Manually

Create a UAV mission object with a home location at the origin of the local ENU coordinate frame and an initial speed of 5 meters per second.

```
m = uavMission(Frame="LocalENU",HomeLocation=[0 0 0],Speed=5)
```

```
m = 
  uavMission with properties:

       HomeLocation: [0 0 0]
         InitialYaw: 0
              Frame: "LocalENU"
              Speed: 5
    NumMissionItems: 0
```

Add a takeoff mission item to the mission with an altitude of `25` meters, pitch of `15` degrees, and yaw of `0` degrees.

```
addTakeoff(m,20,Pitch=15,Yaw=0);
```

Add two waypoint mission items to the mission. Between the two waypoints, increase the speed of the UAV to `20` meters per second. After the second waypoint, reduce the speed of the UAV back to `5` meters per second.

```
addWaypoint(m,[10 0 30]);
addChangeSpeed(m,20)
addWaypoint(m,[20 0 40]);
addChangeSpeed(m,5)
addWaypoint(m,[30 0 50])
```

Add loiter and hover mission items to the mission, specifying for the UAV to loiter and hover around the second waypoint at a radius of 50 meters for 20 seconds each.

```
addLoiter(m,[40 0 60],10,20);
addHover(m,[50 0 70],10,20);
```

Add a landing mission item to the mission to land the UAV.

```
addLand(m,[70 0 0],Yaw=0);
```

Show the mission item data table.

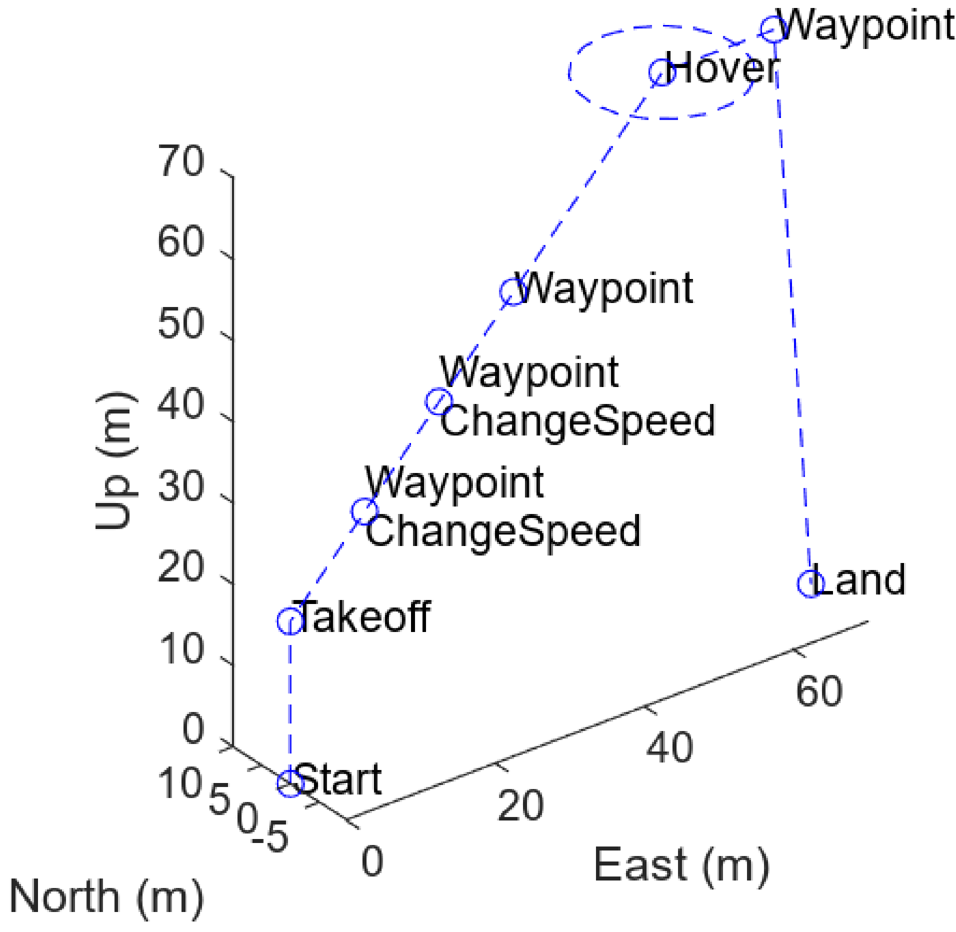
```
showdetails(m)
```

Remove the hover action at index 7, and then add another waypoint at index 8 after the hover item moves to index 7. Show the mission details table again to see the changes.

```
removeItem(m,7);
addWaypoint(m,[65 0 70],InsertAtRow=8);
showdetails(m)
```

Visualize the mission.

```
show(m);
axis equal
```



Input Arguments

mission — UAV mission
uavMission object

UAV mission, specified as a `uavMission` object.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Example: `show(m,ReferenceLocation=[43 32 55])`

Parent — Parent axes for plot
Axes object

Parent axes for the plot, specified as an `Axes` object.

Example: `show(m,Parent=ax2)`

`ReferenceLocation` — Reference location of plot origin
three-element row vector

Reference location of the plot origin, specified as a three-element row vector of the form [*latitude longitude altitude*]. The first two elements specify the latitude and longitude, respectively, of the UAV starting location in degrees, and the third specifies the starting altitude of the UAV in meters.

Example: `show(m,ReferenceLocation=[43 32 55])`

Output Arguments

`AX` — Axes handle
`Axes` object

Axes handle, specified as an `Axes` object.

Version History

Introduced in R2022b

See Also

Objects
`uavMission`

Functions
`addChangeSpeed` | `addHover` | `addLand` | `addLoiter` | `addTakeoff` | `addWaypoint` | `removeItem` | `showdetails`

showdetails

UAV mission data table

Syntax

```
details = showdetails(mission)
```

Description

`details = showdetails(mission)` returns a UAV mission data table `table` that contains details about the mission.

Examples

Create UAV Mission Manually

Create a UAV mission object with a home location at the origin of the local ENU coordinate frame and an initial speed of 5 meters per second.

```
m = uavMission(Frame="LocalENU",HomeLocation=[0 0 0],Speed=5)
```

```
m = 
  uavMission with properties:

       HomeLocation: [0 0 0]
         InitialYaw: 0
              Frame: "LocalENU"
              Speed: 5
    NumMissionItems: 0
```

Add a takeoff mission item to the mission with an altitude of `25` meters, pitch of `15` degrees, and yaw of `0` degrees.

```
addTakeoff(m,20,Pitch=15,Yaw=0);
```

Add two waypoint mission items to the mission. Between the two waypoints, increase the speed of the UAV to `20` meters per second. After the second waypoint, reduce the speed of the UAV back to `5` meters per second.

```
addWaypoint(m,[10 0 30]);
addChangeSpeed(m,20)
addWaypoint(m,[20 0 40]);
addChangeSpeed(m,5)
addWaypoint(m,[30 0 50])
```

Add loiter and hover mission items to the mission, specifying for the UAV to loiter and hover around the second waypoint at a radius of `50` meters for `20` seconds each.

```
addLoiter(m,[40 0 60],10,20);
addHover(m,[50 0 70],10,20);
```

Add a landing mission item to the mission to land the UAV.

```
addLand(m,[70 0 0],Yaw=0);
```

Show the mission item data table.

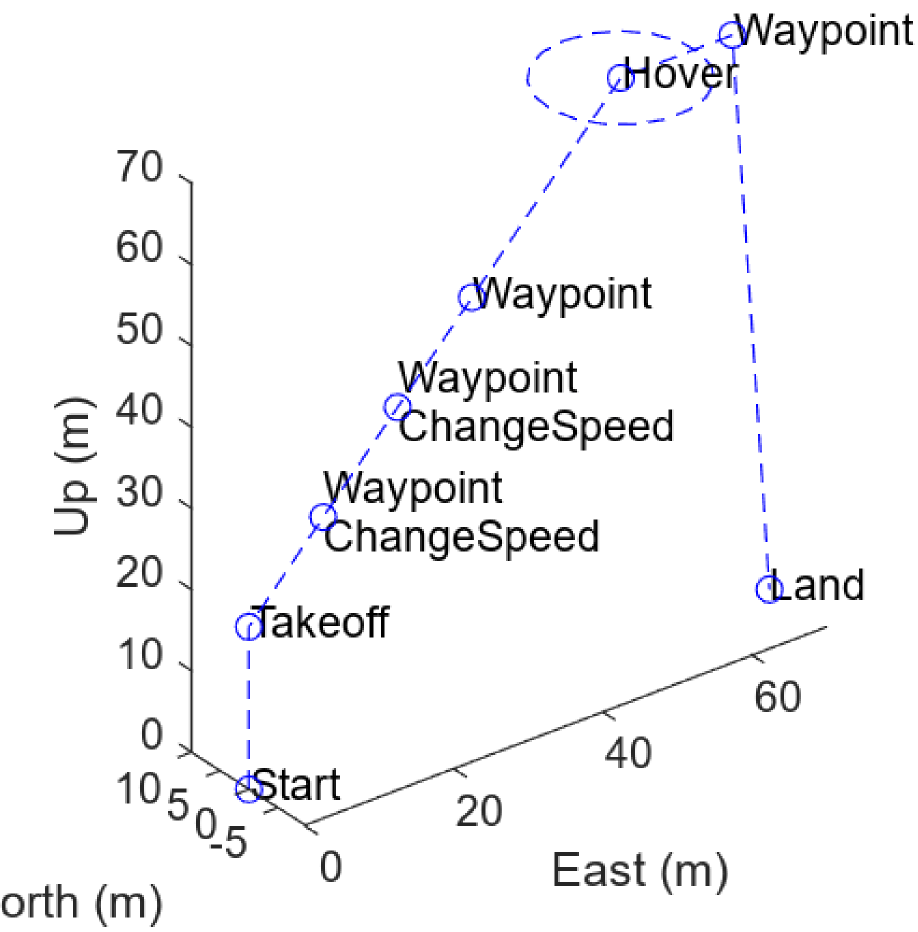
```
showdetails(m)
```

Remove the hover action at index 7, and then add another waypoint at index 8 after the hover item moves to index 7. Show the mission details table again to see the changes.

```
removeItem(m,7);
addWaypoint(m,[65 0 70],InsertAtRow=8);
showdetails(m)
```

Visualize the mission.

```
show(m);
axis equal
```



Input Arguments

`mission` — UAV mission
`uavMission` object

UAV mission, specified as a `uavMission` object.

Output Arguments

`details` — Mission data table
table

Mission data table, returned as a table.

Data Types: `table`

Version History

Introduced in R2022b

See Also

Objects
`uavMission`

Functions
`addChangeSpeed` | `addHover` | `addLand` | `addLoiter` | `addTakeoff` | `addWaypoint` | `removeItem` | `show`

fixedwingMissionParser

Generate trajectory for fixed-wing UAV from mission

Description

The `fixedwingMissionParser` object parses the mission defined in a `uavMission` object and generates a flight trajectory as a `fixedwingFlightTrajectory` object.

Creation

Syntax

```
parser = fixedwingMissionParser
parser = fixedwingMissionParser(Name=Value)
```

Description

`parser = fixedwingMissionParser` creates a fixed-wing UAV mission parser `parser`.

`parser = fixedwingMissionParser(Name=Value)` specifies properties using one or more name-value arguments.

Properties

`TransitionRadius` — Transition radius (m)
`10` (default) | positive numeric scalar

Transition radius for computing trajectory waypoints based on the mission definition, specified as a positive numeric scalar, in meters.

Example: `fixedwingMissionParser(mission,TransitionRadius=3)`

`TakeoffPitch` — Takeoff pitch angle (°)
`5` (default) | positive scalar

Takeoff pitch angle, specified as a positive numeric scalar, in degrees. This value specifies the angle between the flight path of the UAV and the horizontal plane during the takeoff action.

Example: `fixedwingMissionParser(mission,TakeoffPitch=8)`

Object Functions

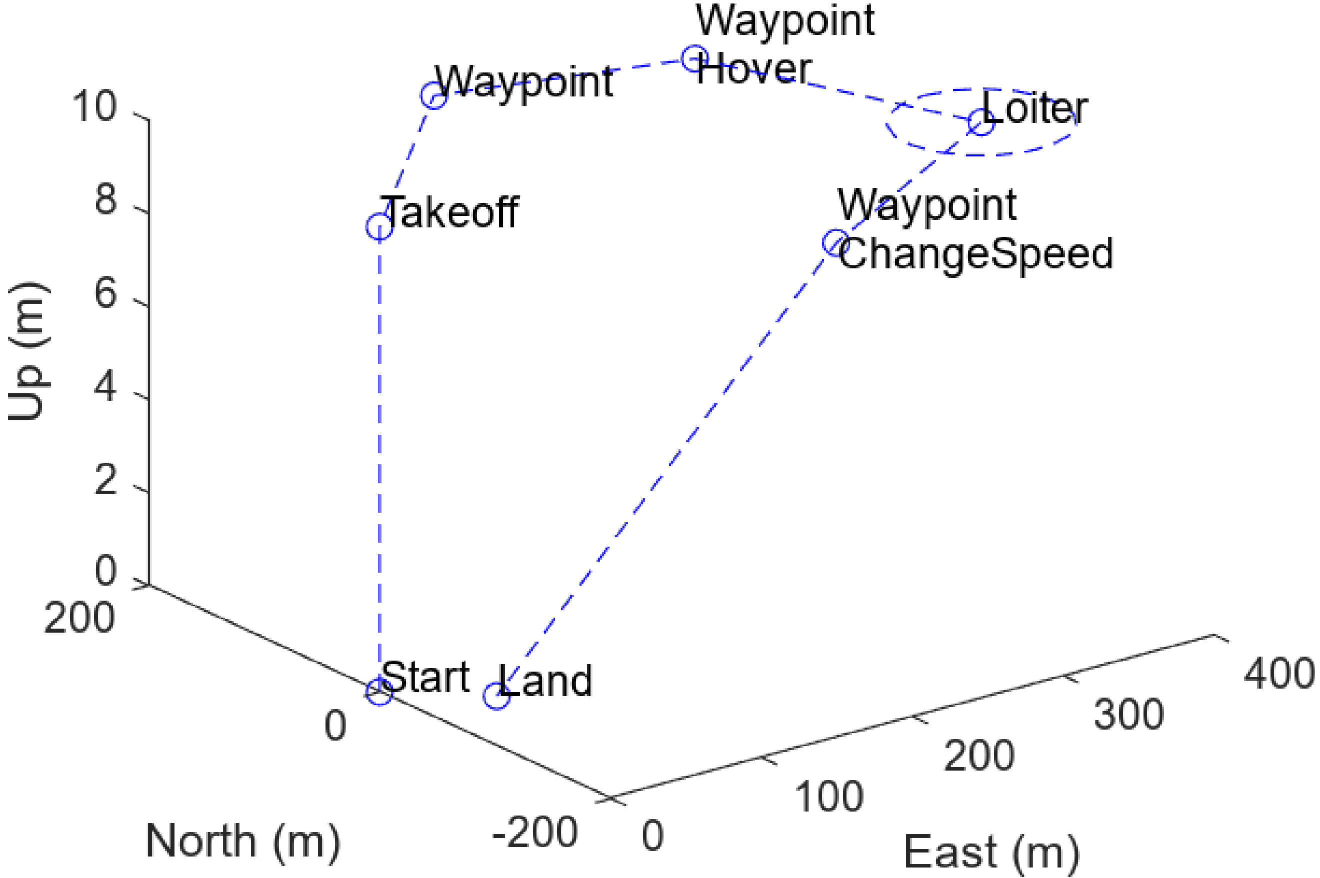
copy	Copy mission parser
parse	Generate trajectory in local NED frame

Examples

Generate Flight Trajectory for UAV Mission

Create a UAV mission by using the flight plan stored in a `.plan` file and show the mission.

```
mission = uavMission(PlanFile="flight.plan");
show(mission);
```



Create parsers for a multirotor UAV and a fixed-wing UAV.

```
mrmParser = multirotorMissionParser(TransitionRadius=2,TakeoffSpeed=2);
fwmParser = fixedwingMissionParser(TransitionRadius=15,TakeoffPitch=10);
```

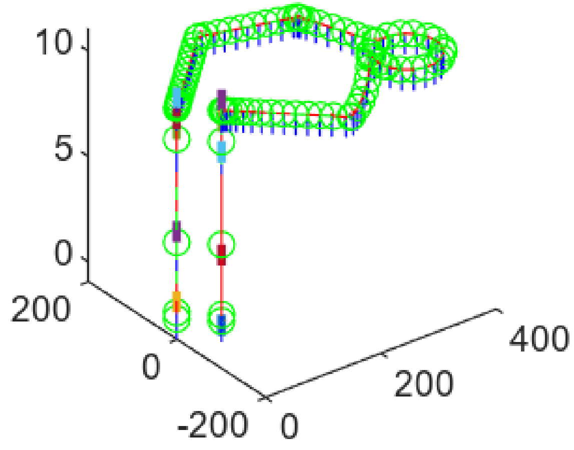
Generate one flight trajectory using each parser.

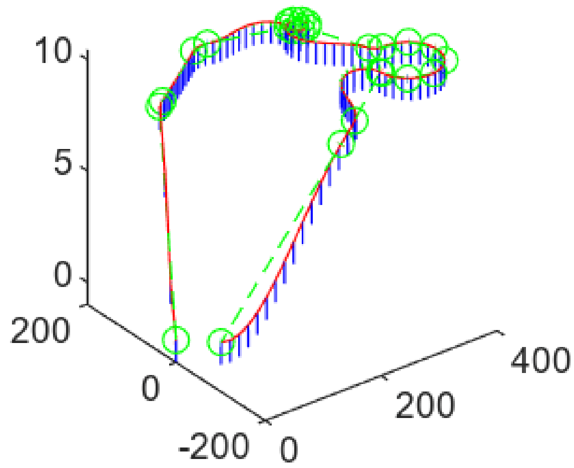
```
mrmTraj = parse(mrmParser,mission);
fwmTraj = parse(fwmParser,mission);
```

Visualize the mission and the flight trajectories separately.

```
figure
subplot(1,2,1)
```

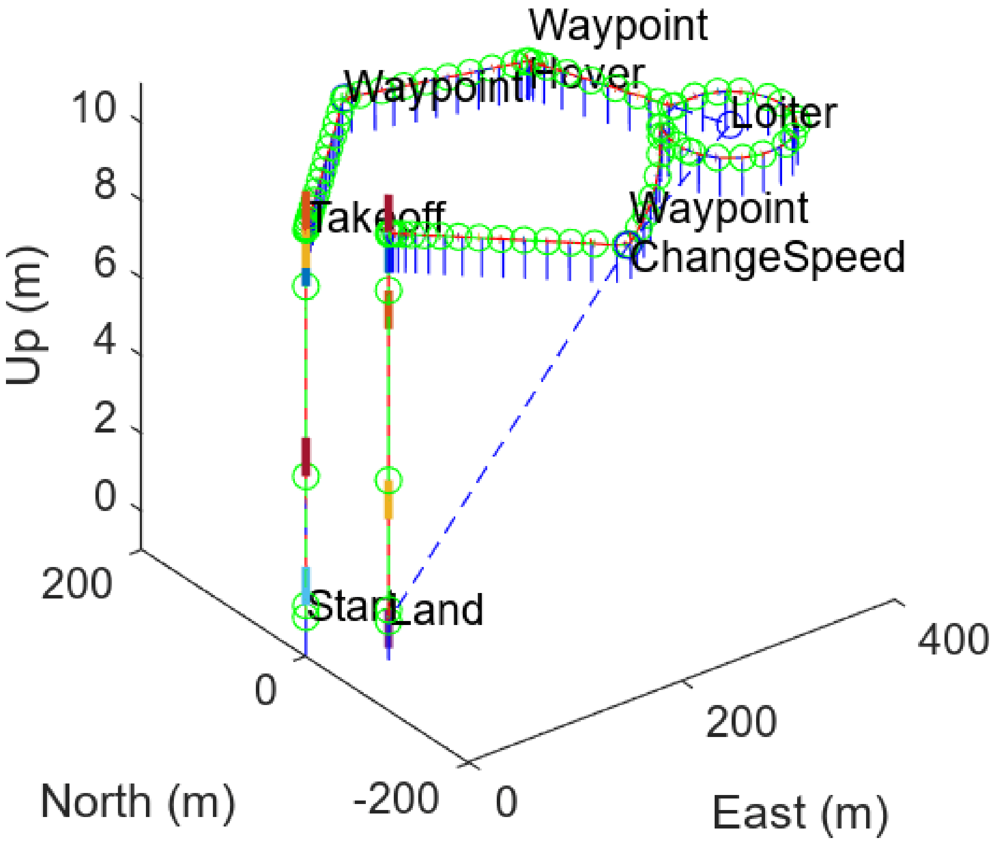
```
show(mrmTraj);
title("Multirotor Trajectory")
axis square
subplot(1,2,2)
show(fwmTraj);
title("Fixed-Wing Trajectory")
axis square
```



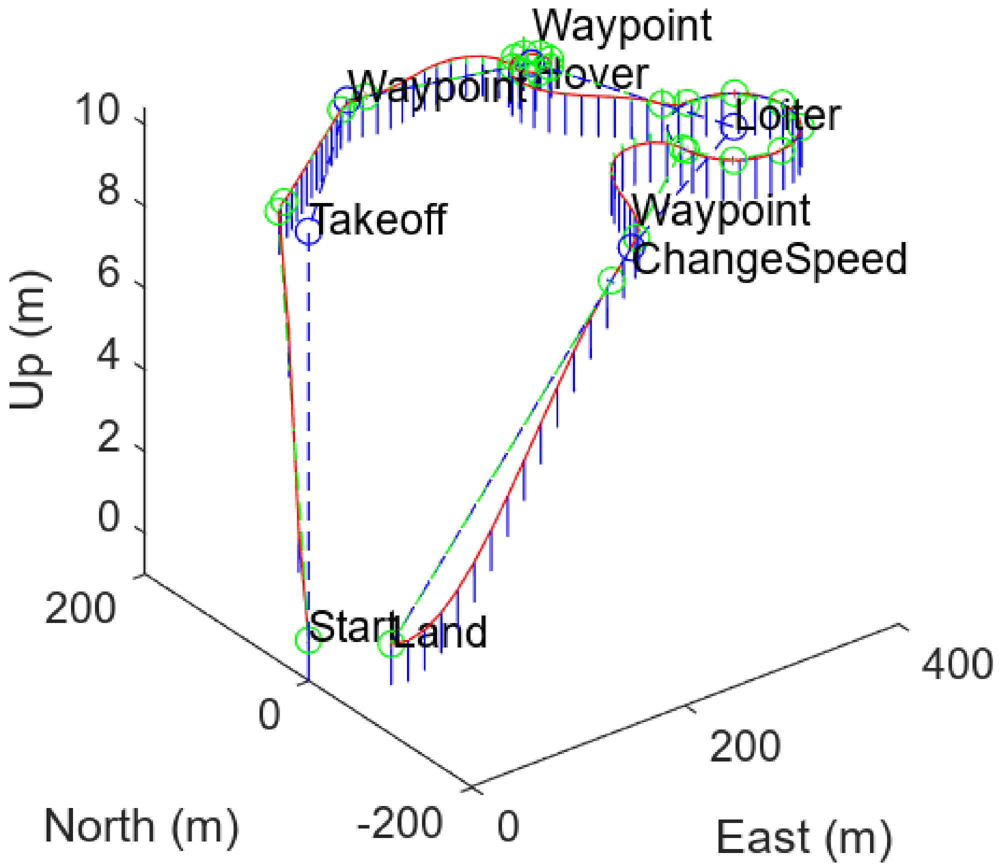


Plot the mission and flight trajectories overlapping.

```
figure
show(mission);
hold on
show(mrmTraj);
hold off
title("Mission Using Multirotor Trajectory")
axis square
```




```
show(mission);
hold on
show(fwmTraj);
hold off
title("Mission Using Fixed-Wing Trajectory")
axis square
```



Version History

Introduced in R2022b

See Also

`fixedwingFlightTrajectory` | `multirotorMissionParser` | `uavMission`

multirotorMissionParser

Generate trajectory for multirotor UAV from mission

Description

The `multirotorMissionParser` object parses the mission defined in a `uavMission` object and generates a flight trajectory as a `multirotorFlightTrajectory` object.

Creation

Syntax

```
parser = multirotorMissionParser
parser = multirotorMissionParser(Name=Value)
```

Description

`parser = multirotorMissionParser` creates a multirotor UAV mission parser `parser`.

`parser = multirotorMissionParser(Name=Value)` specifies properties using one or more name-value arguments.

Properties

`TransitionRadius` — Transition radius (m)
2 (default) | positive numeric scalar

Transition radius for computing trajectory waypoints based on the mission definition, specified as a positive numeric scalar, in meters.

Example: `multirotorMissionParser(mission,TransitionRadius=3)`

`TakeoffSpeed` — Takeoff speed (m/s)
2 (default) | positive numeric scalar

Takeoff speed for the multirotor UAV, specified as a positive numeric scalar, in meters per second. The parser uses this speed to calculate the time required for the multirotor to reach the desired altitude during takeoff and landing.

Example: `multirotorMissionParser(mission,TakeoffSpeed=8)`

Object Functions

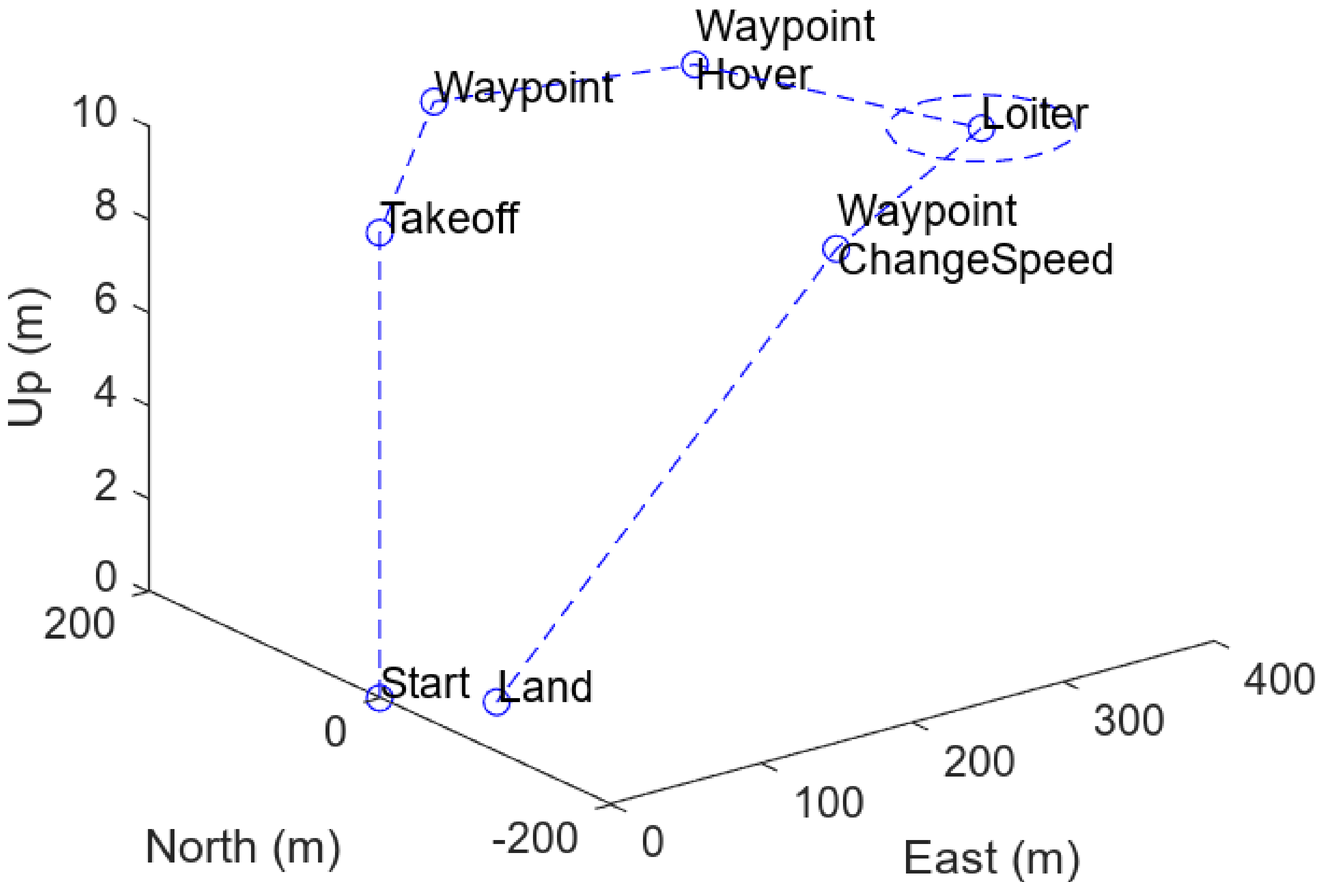
copy	Copy mission parser
parse	Generate trajectory in local NED frame

Examples

Generate Flight Trajectory for UAV Mission

Create a UAV mission by using the flight plan stored in a `.plan` file and show the mission.

```
mission = uavMission(PlanFile="flight.plan");
show(mission);
```



Create parsers for a multirotor UAV and a fixed-wing UAV.

```
mrmParser = multirotorMissionParser(TransitionRadius=2,TakeoffSpeed=2);
fwmParser = fixedwingMissionParser(TransitionRadius=15,TakeoffPitch=10);
```

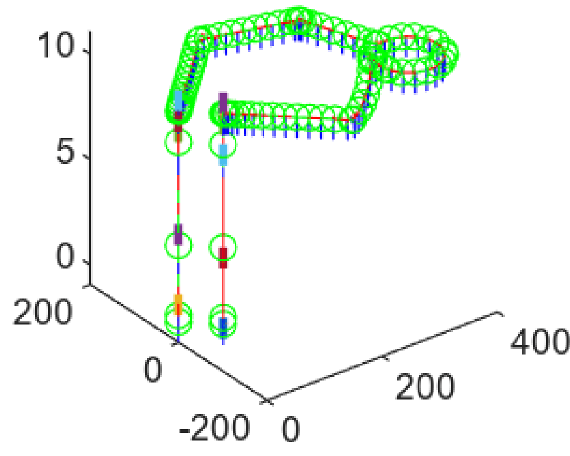
Generate one flight trajectory using each parser.

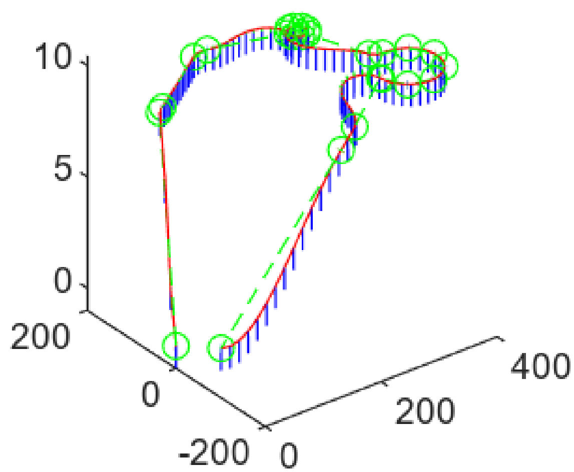
```
mrmTraj = parse(mrmParser,mission);
fwmTraj = parse(fwmParser,mission);
```

Visualize the mission and the flight trajectories separately.

```
figure
subplot(1,2,1)
```

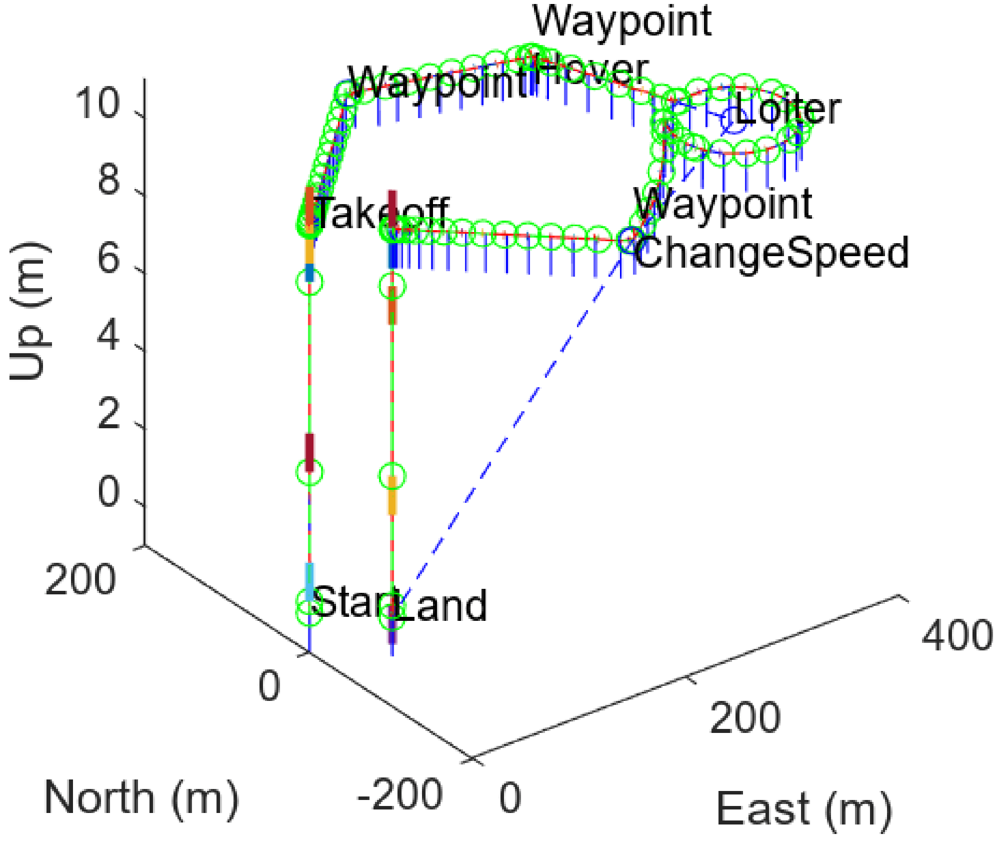
```
show(mrmTraj);
title("Multirotor Trajectory")
axis square
subplot(1,2,2)
show(fwmTraj);
title("Fixed-Wing Trajectory")
axis square
```



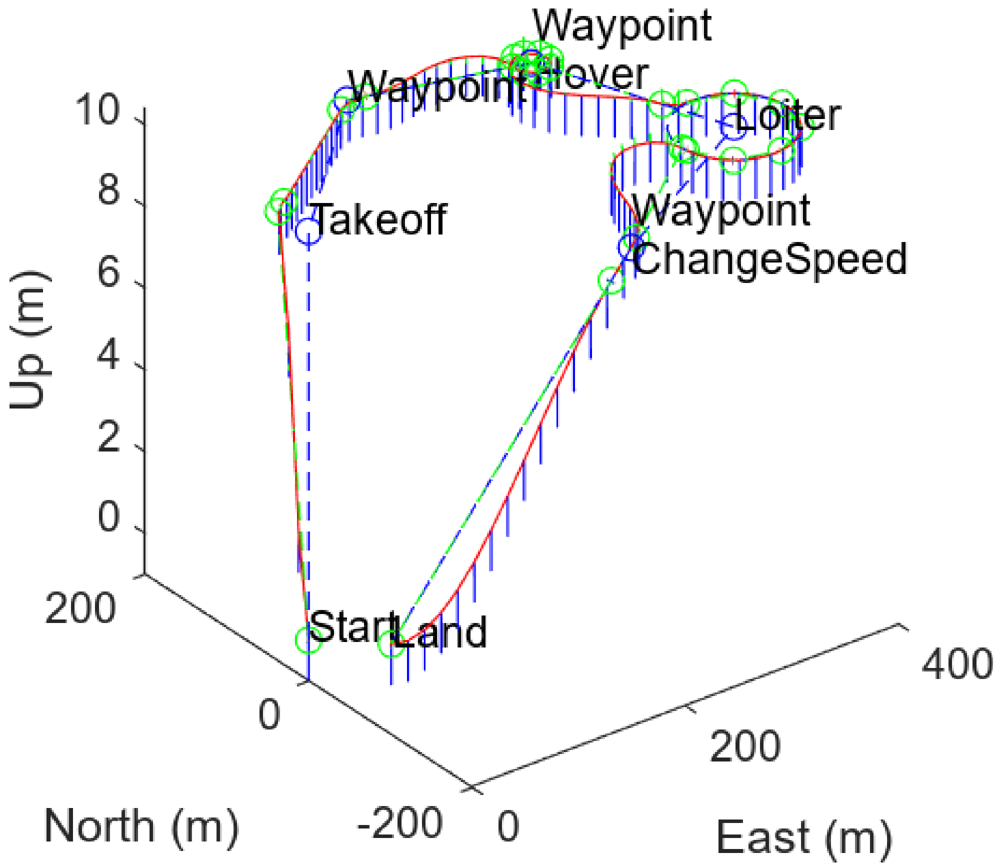


Plot the mission and flight trajectories overlapping.

```
figure
show(mission);
hold on
show(mrmTraj);
hold off
title("Mission Using Multirotor Trajectory")
axis square
```




```
show(mission);
hold on
show(fwmTraj);
hold off
title("Mission Using Fixed-Wing Trajectory")
axis square
```



Version History

Introduced in R2022b

See Also

`fixedwingMissionParser` | `multirotorFlightTrajectory` | `uavMission`

copy

Copy mission parser

Syntax

```
parserCopy = copy(parser)
```

Description

`parserCopy = copy(parser)` creates a deep copy of a `fixedwingMissionParser` or a `multirotorMissionParser` object. The copy has the same properties as the original.

Input Arguments

`parser` — UAV Mission parser
`multirotorMissionParser` object | `fixedwingMissionParser` object

UAV mission parser, specified as a `multirotorMissionParser` object or a `fixedwingMissionParser` object.

Output Arguments

`parserCopy` — Deep copy of mission parser
`multirotorMissionParser` object | `fixedwingMissionParser` object

Deep copy of the mission parser, returned as either a `multirotorMissionParser` object or a `fixedwingMissionParser` object. The copy is of the same object type, and has the same properties, as the object specified to `parser`.

Version History

Introduced in R2022b

See Also

`fixedwingMissionParser` | `multirotorMissionParser`

parse

Generate trajectory in local NED frame

Syntax

```
traj = parse(parser,mission)
traj = parse(parser,mission,refLocation)
```

Description

`traj = parse(parser,mission)` generates a reference trajectory `traj` for a multirotor or fixed-wing UAV following the mission `mission`. The `parse` function generates the reference trajectory in a local north-east-down (NED) frame with its origin at the global coordinates specified by the `HomeLocation` property of `mission`.

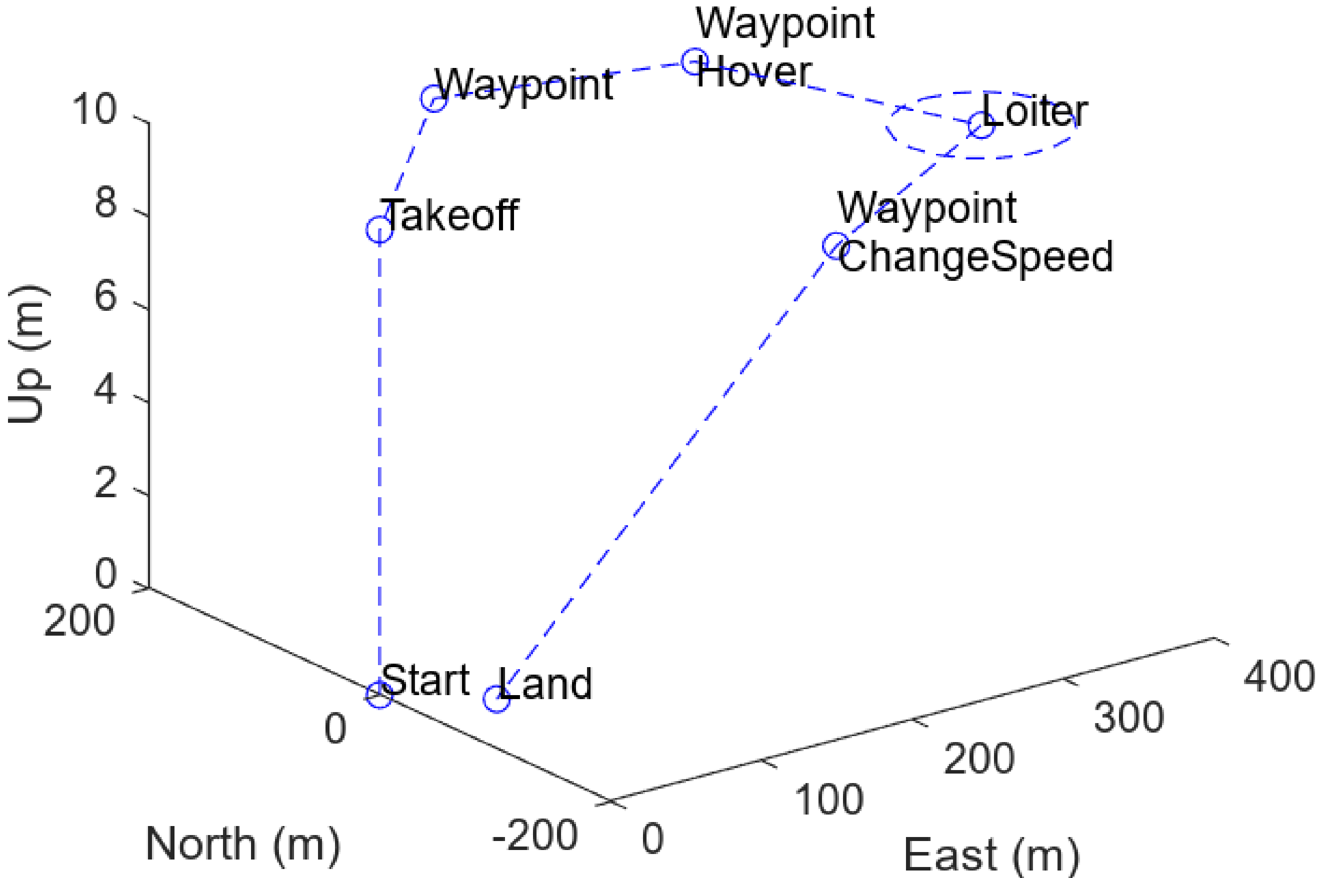
`traj = parse(parser,mission,refLocation)` generates the reference trajectory in a local NED frame with its origin defined by the global coordinates `refLocation`.

Examples

Generate Flight Trajectory for UAV Mission

Create a UAV mission by using the flight plan stored in a `.plan` file and show the mission.

```
mission = uavMission(PlanFile="flight.plan");
show(mission);
```



Create parsers for a multirotor UAV and a fixed-wing UAV.

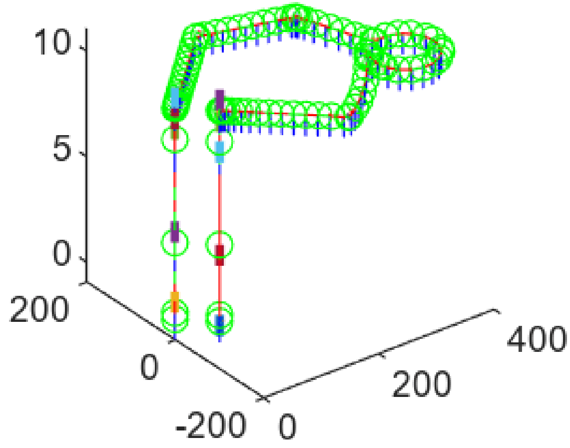
```
mrmParser = multirotorMissionParser(TransitionRadius=2,TakeoffSpeed=2);
fwmParser = fixedwingMissionParser(TransitionRadius=15,TakeoffPitch=10);
```

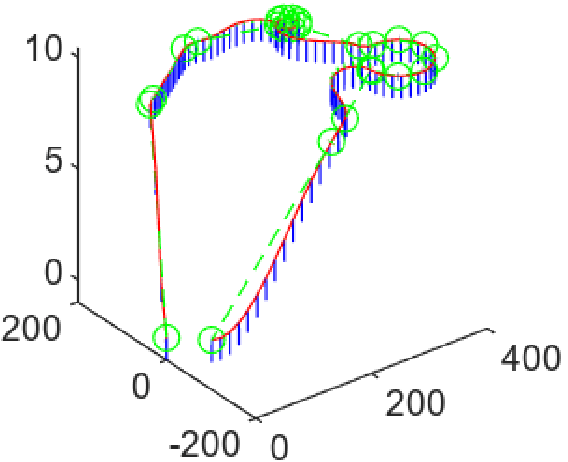
Generate one flight trajectory using each parser.

```
mrmTraj = parse(mrmParser,mission);
fwmTraj = parse(fwmParser,mission);
```

Visualize the mission and the flight trajectories separately.

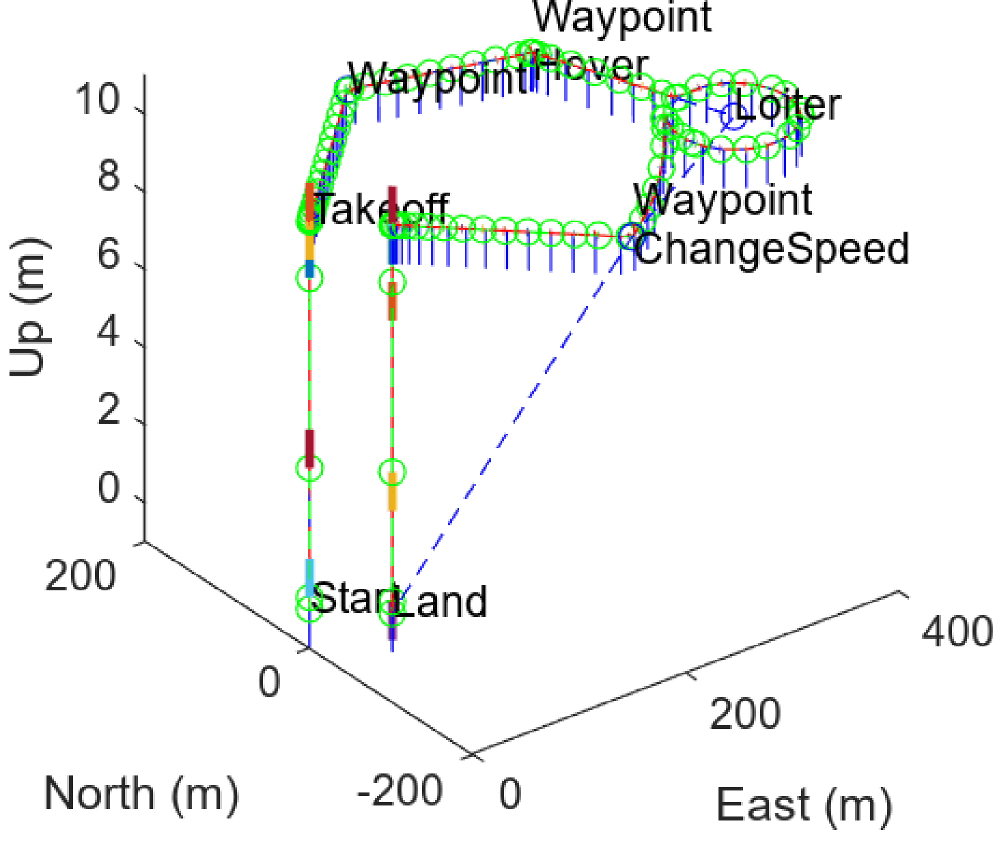
```
figure
subplot(1,2,1)
show(mrmTraj);
title("Multirotor Trajectory")
axis square
subplot(1,2,2)
show(fwmTraj);
title("Fixed-Wing Trajectory")
axis square
```



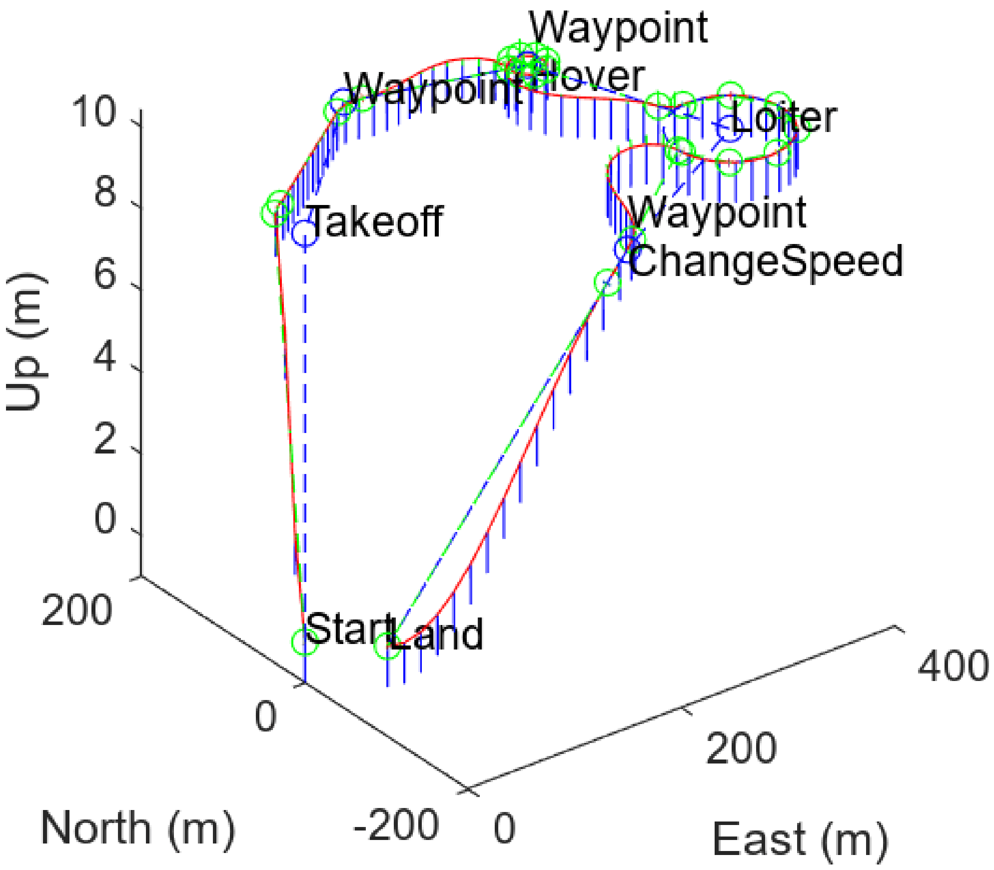


Plot the mission and flight trajectories overlapping.

```
figure
show(mission);
hold on
show(mrmTraj);
hold off
title("Mission Using Multirotor Trajectory")
axis square
```




```
show(mission);
hold on
show(fwmTraj);
hold off
title("Mission Using Fixed-Wing Trajectory")
axis square
```



Input Arguments

parser — UAV Mission parser

`multirotorMissionParser` object | `fixedwingMissionParser` object

UAV mission parser, specified as a `multirotorMissionParser` object or a `fixedwingMissionParser` object.

mission — UAV mission

`uavMission` object

UAV mission, specified as a `uavMission` object.

refLocation — Reference location

`mission.HomeLocation` (default) | three-element row vector

Reference location, specified as a three-element row vector of the form [*latitude longitude altitude*], representing global coordinates. *latitude* and *longitude* are in degrees. *altitude* is the height in meters above the WGS84 ellipsoid.

Example: `parse(parser,mission,[20 40.1 72])`

Output Arguments

`traj` — Trajectory
`multirotorFlightTrajectory` object | `fixedwingFlightTrajectory` object

Trajectory, returned as a `multirotorFlightTrajectory` object for `multirotorMissionParser` objects and `fixedwingFlightTrajectory` object for `fixedwingMissionParser` objects.

Version History

Introduced in R2022b

See Also

Objects
`fixedwingMissionParser` | `multirotorMissionParser` | `fixedwingFlightTrajectory` | `multirotorFlightTrajectory`

fixedwingFlightTrajectory

Fixed-wing UAV trajectory

Description

The `fixedwingFlightTrajectory` object stores a fixed-wing UAV trajectory created by using a `waypointTrajectory` System object to interpolate positions between specified waypoints given velocities and times of arrival.

Creation

Syntax

```
T = fixedwingFlightTrajectory(wpts,vels,toas)
T = parse(parser,mission)
```

Description

`T = fixedwingFlightTrajectory(wpts,vels,toas)` creates a `fixedwingFlightTrajectory`, `T`, using the specified waypoints `wpts`, velocities `vels`, and times of arrival `toas`. The input arguments set the `Waypoints`, `Velocities`, and `TimesOfArrival` properties, respectively.

`T = parse(parser,mission)` parses a UAV mission `mission` using a fixed-wing mission parser `parser` to create a `fixedwingFlightTrajectory` object.

Properties

`Waypoints` — Waypoints
N-by-3 matrix

Waypoints, specified as an *N*-by-3 matrix in the local north-east-down (NED) reference frame, in meters. *N* is the total number of waypoints, and each row contains the *xyz*-coordinates of a waypoint, [*X Y Z*].

Example: `[2 1 3; 3 2 4]`

`Velocities` — Desired velocities
N-by-3 matrix

Desired velocities, specified as an *N*-by-3 matrix in the local north-east-down (NED) reference frame, in meters per second. *N* is the total number of waypoints, and each row is the desired velocity of the UAV at the corresponding waypoint, [*dX dY dZ*].

Example: `[2 1 3; 3 2 4]`

`TimeOfArrival` — Times of arrival
N-element column vector

Times of arrival, specified as an *N*-element column vector, in seconds. *N* is the total number of waypoints, and each row is the time of arrival at the corresponding waypoint.

The first and last elements of `TimeOfArrival` set the `StartTime` and `EndTime` properties, respectively.

Example: `[2; 3; 5; 6]`

`StartTime` — Start time of trajectory
nonnegative numeric scalar

This property is read-only.

Start time of the trajectory, stored as a nonnegative numeric scalar, in seconds. The value of `StartTime` is the same as the value of the first element of the `TimeOfArrival` property.

`EndTime` — End time of trajectory
nonnegative numeric scalar

This property is read-only.

End time of the trajectory, stored as a nonnegative numeric scalar, in seconds. The value of `EndTime` is the same as the value of the last element of the `TimeOfArrival` property.

`parser` — Fixed-wing UAV mission parser
`fixedwingMissionParser` object

Fixed-wing UAV mission parser, specified as a `fixedwingMissionParser` object.

Example: `fixedwingMissionParser(TakeoffPitch=3)`

`mission` — UAV mission
`uavMission` object

UAV mission, specified as a `uavMission` object.

Object Functions

copy	Copy flight trajectory
show	Visualize the UAV trajectory
query	Get UAV motion vectors at timestamps

Examples

Create and Query UAV Flight Trajectory

Create a set of waypoints for both the multirotor and the fixed-wing UAV to follow.

```
wpts = [0  0  0;
        2  2 -2;
       10 10 -3;
       12 12 -6];
numwpts = size(wpts);
```

Specify additional trajectory information, such as desired velocities, accelerations, jerks, snaps, and yaws, as well as start time, an end time, and times of arrival.

```
vels = 2*ones(numwpts);
accs = ones(numwpts);
jerks = zeros(numwpts);
snaps = zeros(numwpts);
yaws = zeros(1,numwpts(1));
starttime = 0;
endtime = 8;
toas = linspace(starttime,endtime,numwpts(1));
```

Use the trajectory information to create the flight trajectories for the multirotor and the fixed-wing UAVs. Query and display the trajectories.

```
mrft = multirotorFlightTrajectory(wpts,vels,accs,jerks,snaps,yaws,toas);
fwft = fixedwingFlightTrajectory(wpts,vels,toas);
query(mrft,1:4)
```

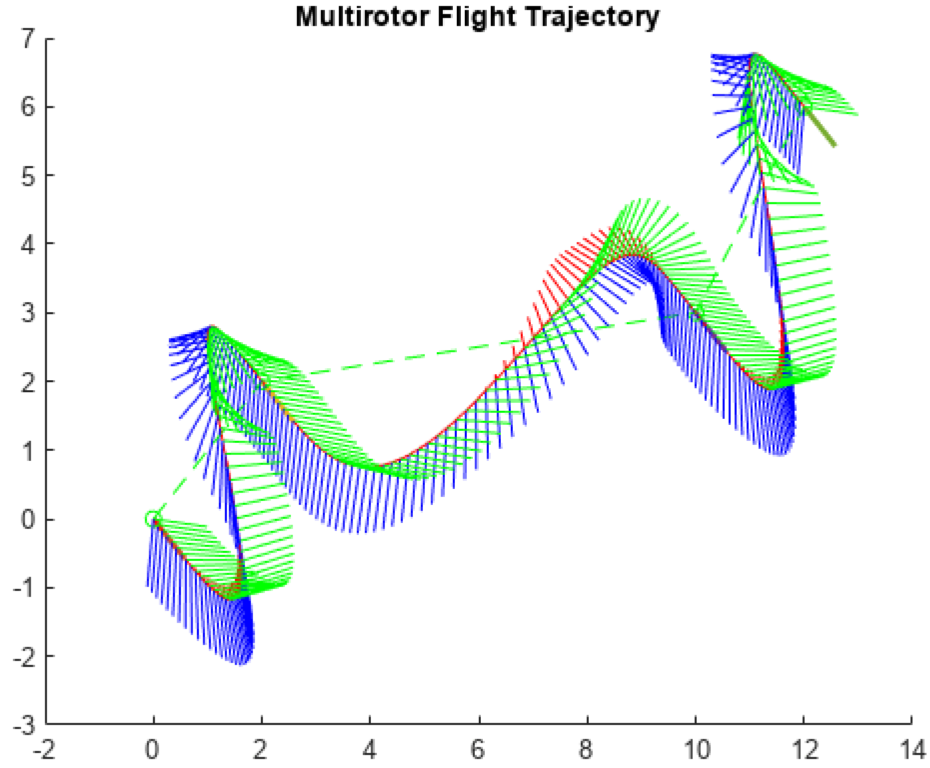
```
ans = 4×16

    1.6184    1.6184    0.7520   -0.0243   -0.0243   -2.8758   -4.6045   -4.6045   -9.1669    0.9
    1.0236    1.0236   -2.7807    0.5482    0.5482   -0.6198    3.9704    3.9704    8.6424    0.7
    2.7277    2.7277   -1.2947    2.4069    2.4069    2.1026    1.7442    1.7442   -1.3857    0.9
    6.4028    6.4028   -2.0972    4.4609    4.4609   -3.8447   -1.1875   -1.1875   -1.1875    0.9
```

```
query(fwft,1:4)
```

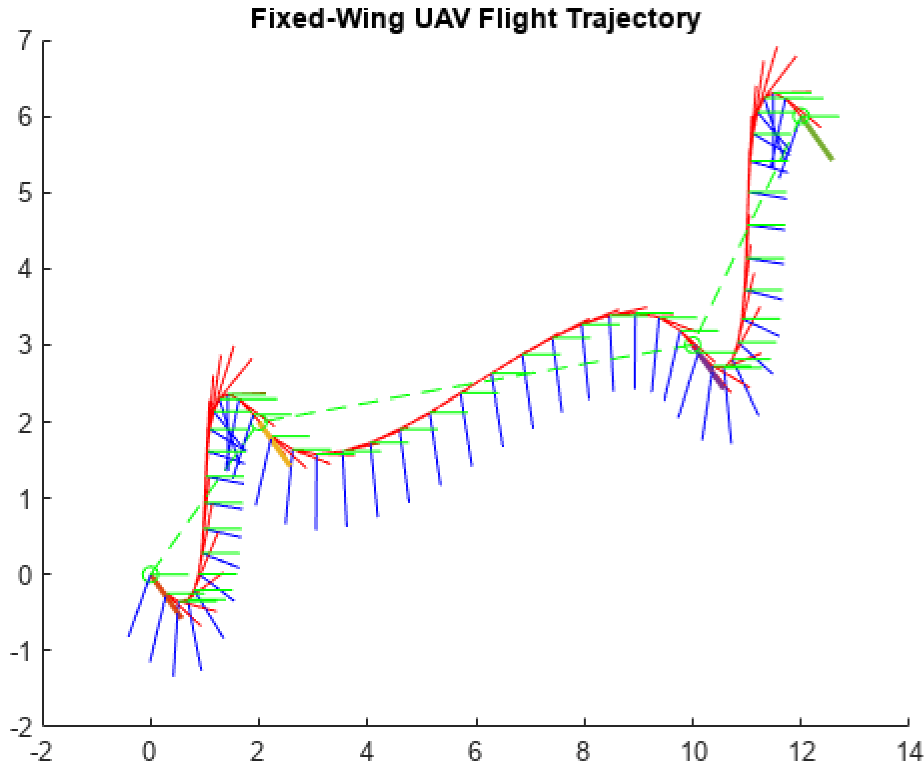
```
ans = 4×16

    0.9453    0.9453   -0.3203    0.2422    0.2422   -1.8672   -0.7031   -0.7031   -1.5469    0.7
    1.1875    1.1875   -2.1875    0.5938    0.5938   -1.0938    1.4063    1.4062    3.0938    0.8
    2.7813    2.7812   -1.6055    2.6563    2.6562    0.4414    1.6875    1.6875   -4.0078    0.9
    6.0000    6.0000   -2.5000    3.5000    3.5000   -1.5625   -0.0000   -0.0000    0.0000    0.9
```

Visualize both the multirotor flight trajectory and the fixed-wing flight trajectory.

```
ax = show(mrft,NumSamples=200);
title("Multirotor Flight Trajectory")
view([0 0])
```



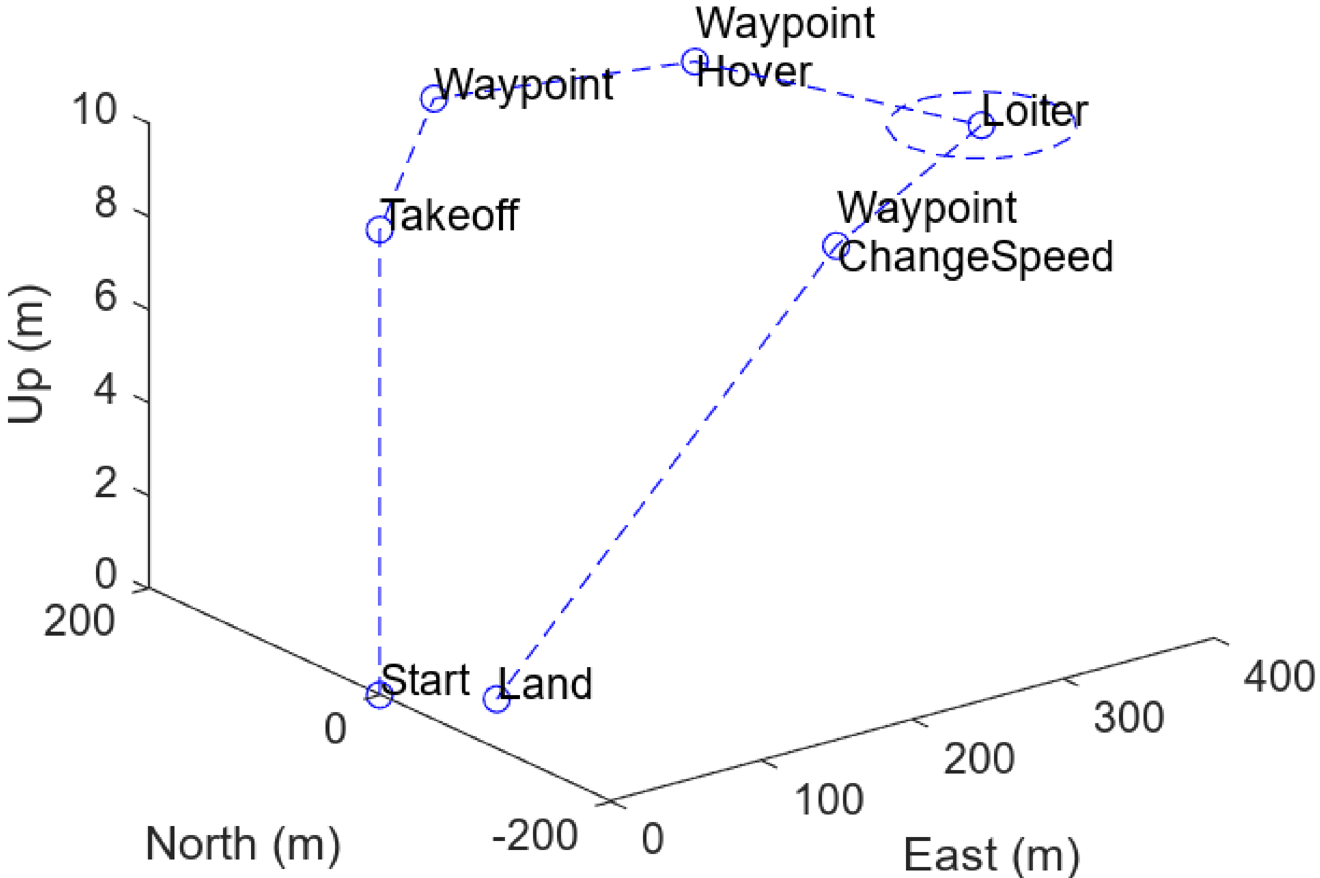

```
show(fwft,NumSamples=50);
title("Fixed-Wing UAV Flight Trajectory")
view([0 0])
```



Generate Flight Trajectory for UAV Mission

Create a UAV mission by using the flight plan stored in a `.plan` file and show the mission.

```
mission = uavMission(PlanFile="flight.plan");
show(mission);
```



Create parsers for a multirotor UAV and a fixed-wing UAV.

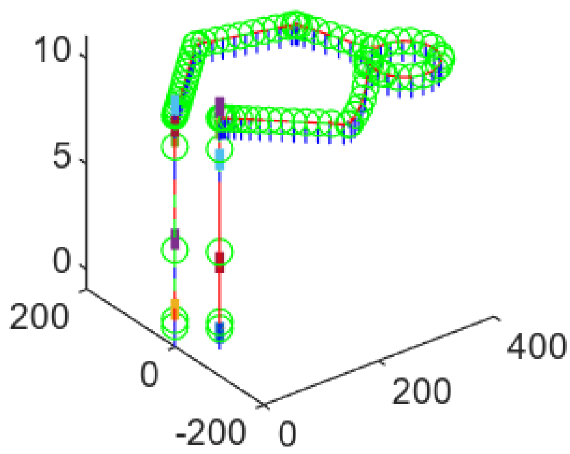
```
mrmParser = multirotorMissionParser(TransitionRadius=2,TakeoffSpeed=2);
fwmParser = fixedwingMissionParser(TransitionRadius=15,TakeoffPitch=10);
```

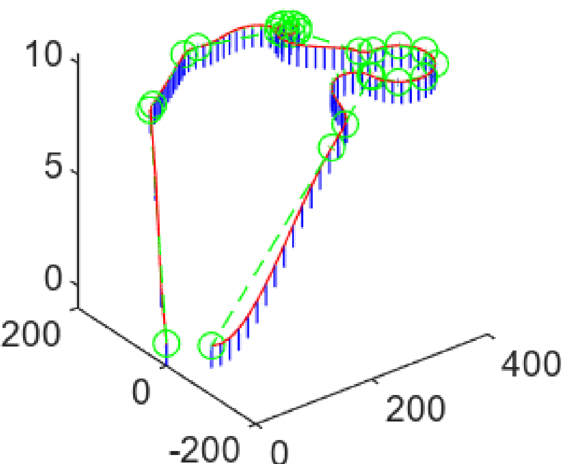
Generate one flight trajectory using each parser.

```
mrmTraj = parse(mrmParser,mission);
fwmTraj = parse(fwmParser,mission);
```

Visualize the mission and the flight trajectories separately.

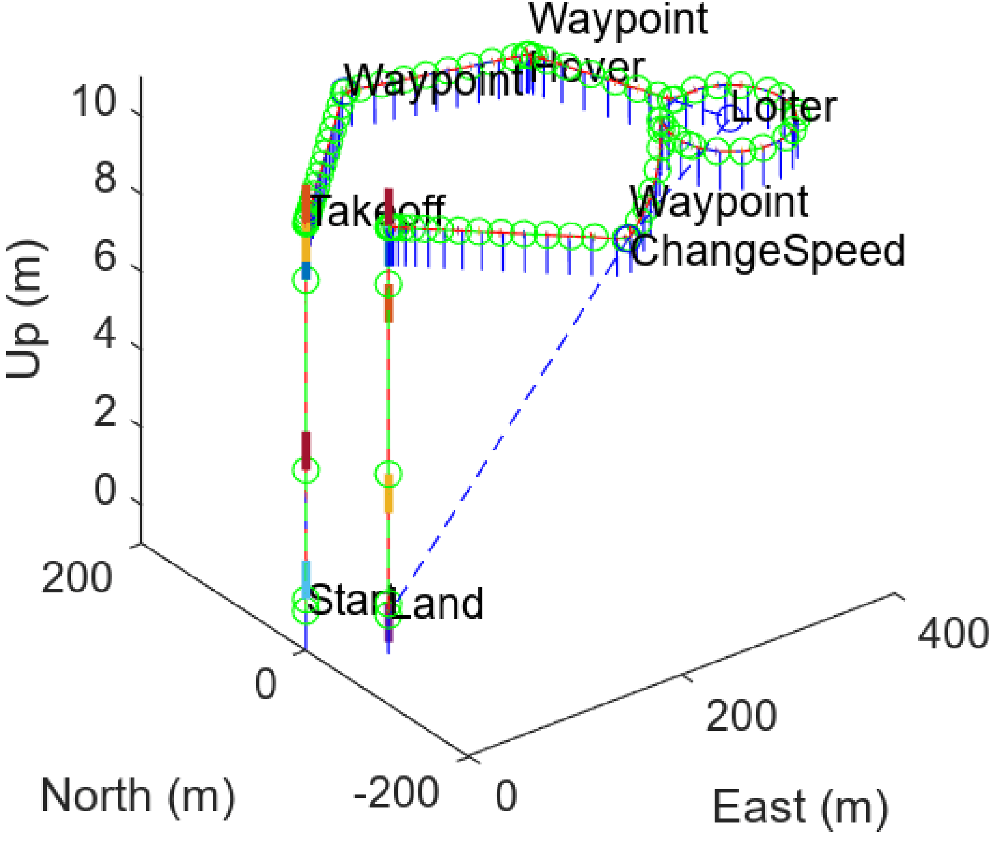
```
figure
subplot(1,2,1)
show(mrmTraj);
title("Multirotor Trajectory")
axis square
subplot(1,2,2)
show(fwmTraj);
title("Fixed-Wing Trajectory")
axis square
```



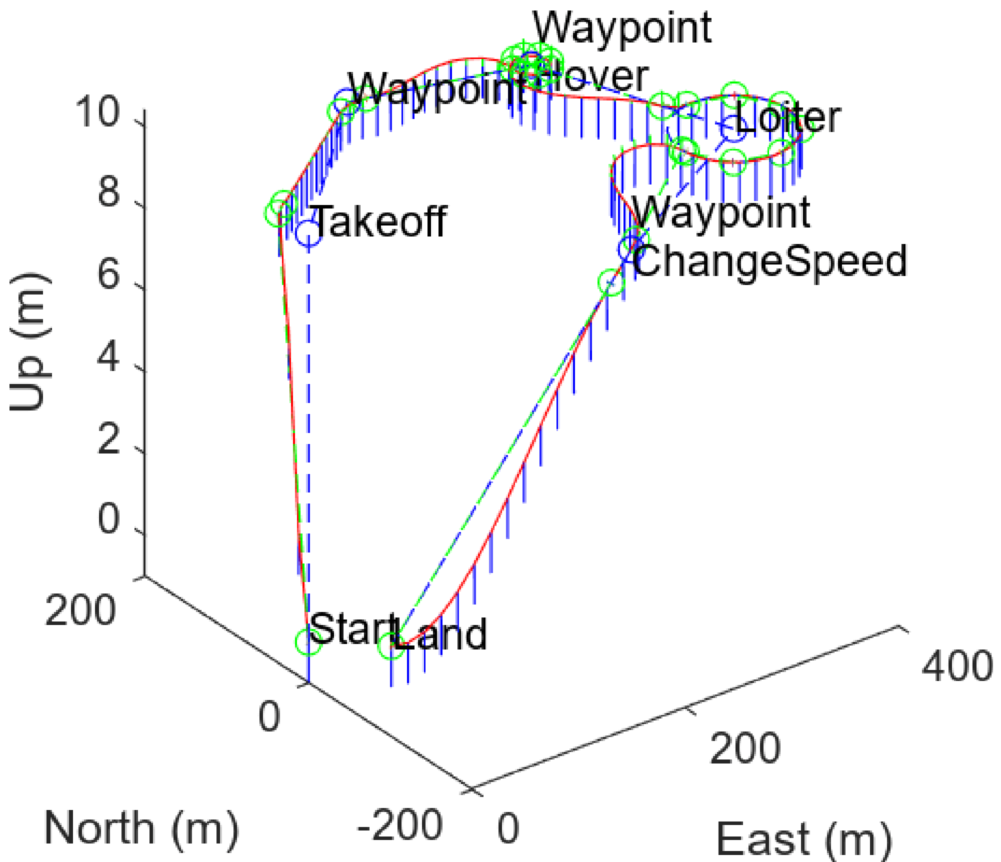


Plot the mission and flight trajectories overlapping.

```
figure
show(mission);
hold on
show(mrmTraj);
hold off
title("Mission Using Multirotor Trajectory")
axis square
```




```
show(mission);
hold on
show(fwmTraj);
hold off
title("Mission Using Fixed-Wing Trajectory")
axis square
```



Version History

Introduced in R2022b

Extended Capabilities

C/C++ Code Generation

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

See Also

`multirotorFlightTrajectory` | `fixedwingMissionParser` | `uavMission`

multirotorFlightTrajectory

Multirotor UAV trajectory

Description

The `multirotorFlightTrajectory` object stores a multirotor UAV trajectory created by using piecewise 9th degree polynomial to interpolate linear positions and the `cubicpolytraj` function to interpolate yaw angles between specified waypoints.

Creation

Syntax

```
T = multirotorFlightTrajectory(wpts,vels,accs,jerks,snaps,yaws,toas)
T = parse(parser,mission)
```

Description

`T = multirotorFlightTrajectory(wpts,vels,accs,jerks,snaps,yaws,toas)` creates a `multirotorFlightTrajectory` object, `T` using the specified waypoints `wpts`, velocities `vels`, accelerations `accs`, jerks `jerks`, snaps `snaps`, yaws `yaws`, and times of arrival `toas`. The input arguments set the `Waypoints`, `Velocities`, `Accelerations`, `Jerks`, `Snaps`, `Yaws`, and `TimeOfArrival` properties, respectively.

`T = parse(parser,mission)` parses a UAV mission `mission` using a multirotor mission parser `parser` to create a `multirotorFlightTrajectory` object.

Properties

`Waypoints` — Waypoints
N-by-3 matrix

Waypoints, specified as an *N*-by-3 matrix in the local north-east-down (NED) reference frame, in meters. *N* is the total number of waypoints, and each row contains the *xyz*-coordinates of a waypoint, [*X Y Z*].

Example: `[2 1 3; 3 2 4]`

`Velocities` — Desired velocities
N-by-3 matrix

Desired velocities, specified as an *N*-by-3 matrix in the local north-east-down (NED) reference frame, in meters per second. *N* is the total number of waypoints, and each row is the desired velocity of the UAV at the corresponding waypoint, [*dX dY dZ*].

Example: `[2 1 3; 3 2 4]`

`Accelerations` — Desired accelerations
N-by-3 matrix

Desired accelerations, specified as an *N*-by-3 matrix in the local north-east-down (NED) reference frame, in m/s^2. *N* is the total number of waypoints, and each row is the desired acceleration at the corresponding waypoint, [d^2X d^2Y d^2Z].

Example: `[2 1 3; 3 2 4]`

`Jerks` — Desired jerks
N-by-3 matrix

Desired jerks, specified as an *N*-by-3 matrix in the local north-east-down (NED) reference frame, in m/s^3. *N* is the total number of waypoints, and each row is the desired jerk at the corresponding waypoint, [d^3X d^3Y d^3Z].

Example: `[2 1 3; 3 2 4]`

`Snaps` — Desired snaps
N-by-3 matrix

Desired snaps, specified as an *N*-by-3 matrix in the local north-east-down (NED) reference frame, in m/s^4. *N* is the total number of waypoints, and each row is the desired snap at the corresponding waypoint, [d^4X d^4Y d^4Z].

Example: `[2 1 3; 3 2 4]`

`Yaws` — Desired yaw angles
N-element vector

Desired yaw angles, specified as an *N*-element vector in the local north-east-down (NED) reference frame, in degrees. *N* is the total number of waypoints, and each row is the desired yaw at the corresponding waypoint.

Example: `[20;32;0;-20]`

`TimeOfArrival` — Times of arrival
N-element column vector

Times of arrival, specified as an *N*-element column vector, in seconds. *N* is the total number of waypoints, and each row is the time of arrival at the corresponding waypoint.

The first and last elements of `TimeOfArrival` set the `StartTime` and `EndTime` properties, respectively.

Example: `[2; 3; 5; 6]`

`StartTime` — Start time of trajectory
nonnegative numeric scalar

This property is read-only.

Start time of the trajectory, stored as a nonnegative numeric scalar, in seconds. The value of `StartTime` is the same as the value of the first element of the `TimeOfArrival` property.

`EndTime` — End time of trajectory
nonnegative numeric scalar

This property is read-only.

End time of the trajectory, stored as a nonnegative numeric scalar, in seconds. The value of `EndTime` is the same as the value of the last element of the `TimeOfArrival` property.

Object Functions

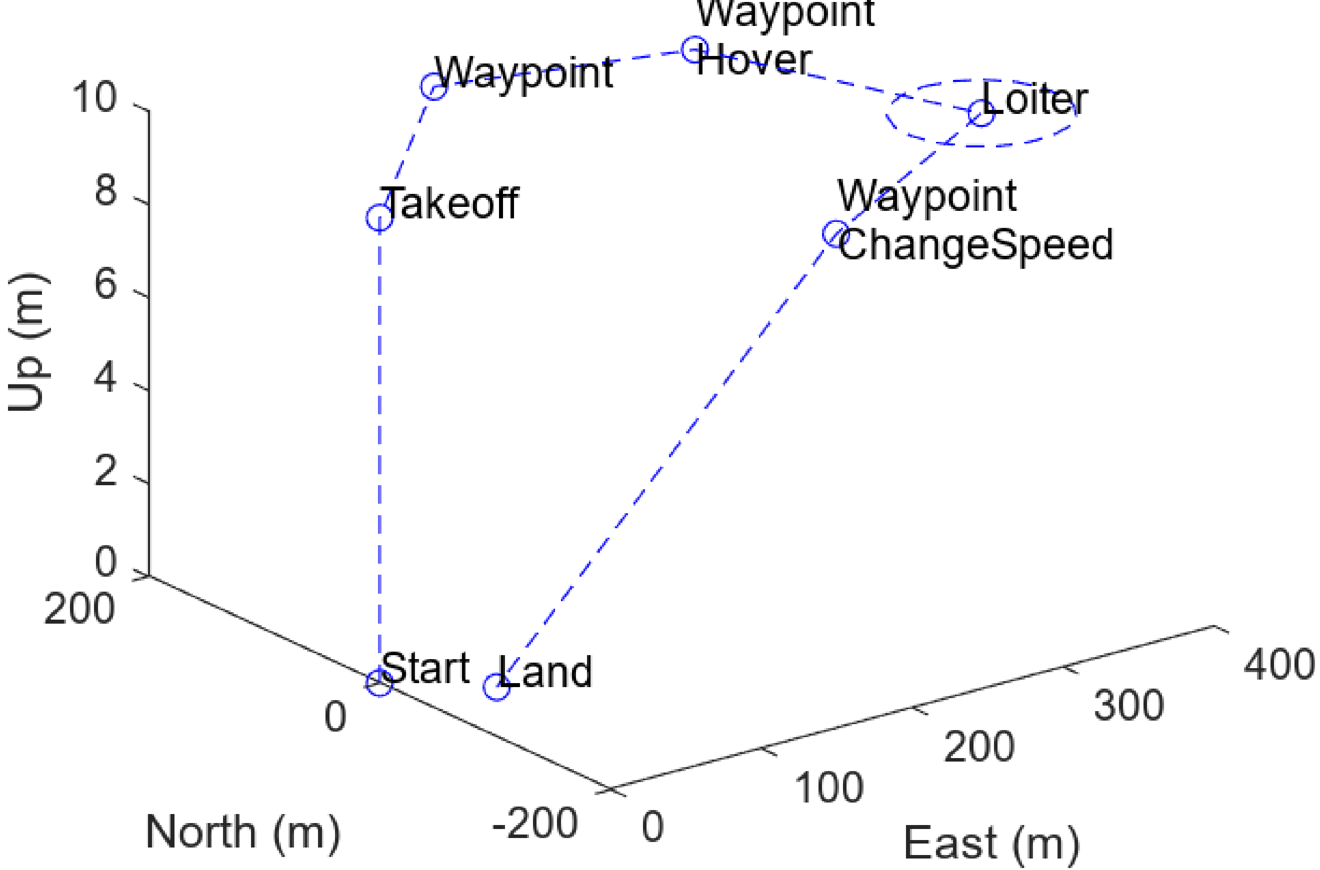
copy	Copy flight trajectory
show	Visualize the UAV trajectory
query	Get UAV motion vectors at timestamps

Examples

Generate Flight Trajectory for UAV Mission

Create a UAV mission by using the flight plan stored in a `.plan` file and show the mission.

```
mission = uavMission(PlanFile="flight.plan");
show(mission);
```



Create parsers for a multirotor UAV and a fixed-wing UAV.

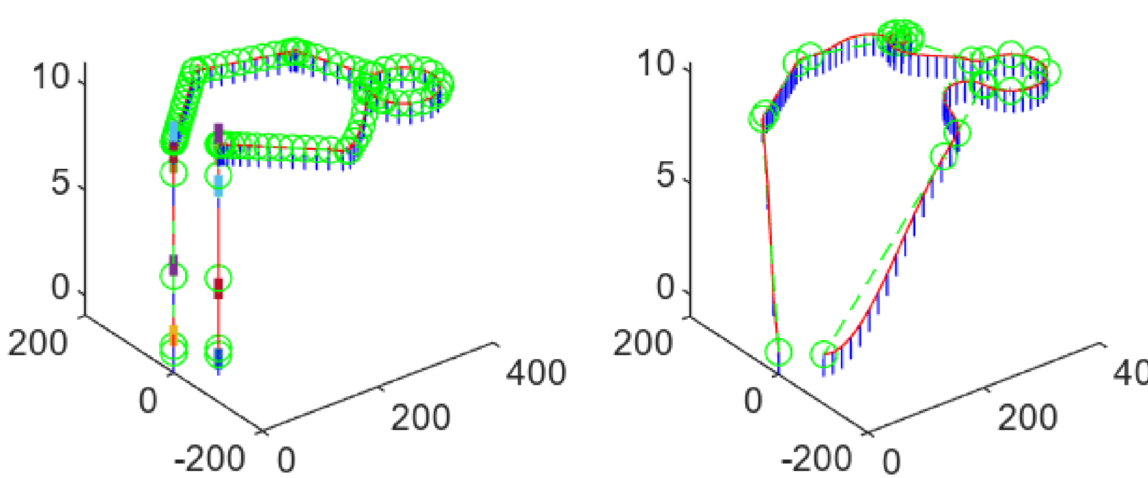
```
mrmParser = multirotorMissionParser(TransitionRadius=2,TakeoffSpeed=2);
fwmParser = fixedwingMissionParser(TransitionRadius=15,TakeoffPitch=10);
```

Generate one flight trajectory using each parser.

```
mrmTraj = parse(mrmParser,mission);
fwmTraj = parse(fwmParser,mission);
```

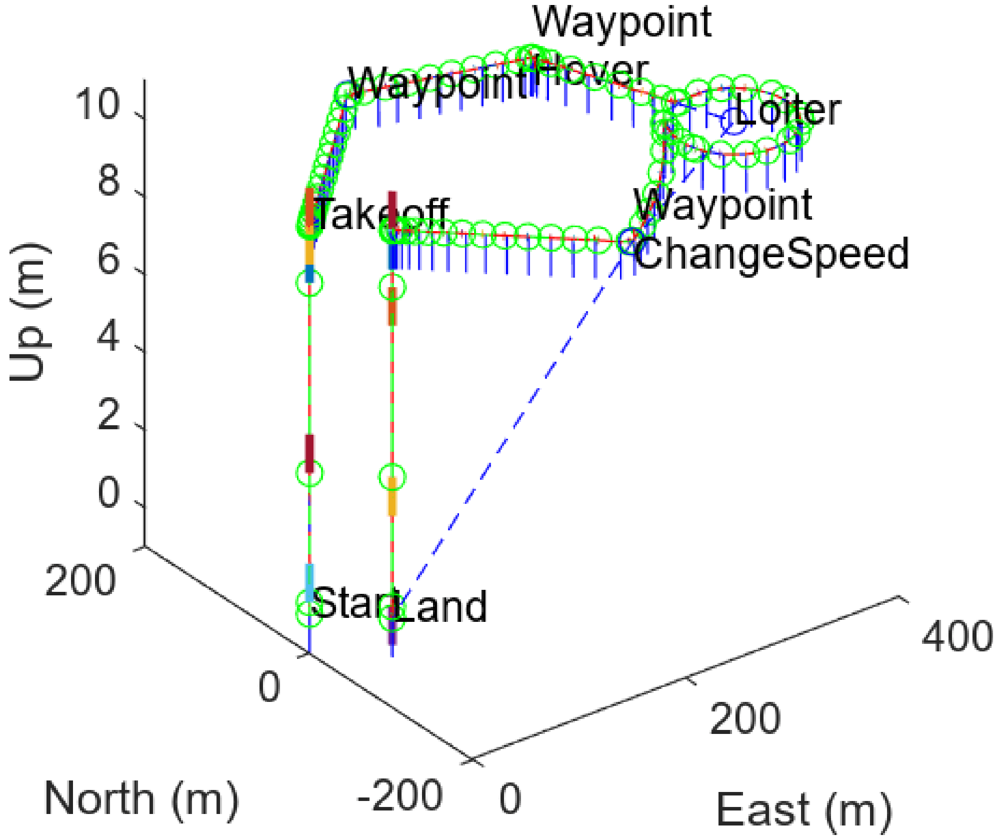
Visualize the mission and the flight trajectories separately.

```
figure
subplot(1,2,1)
show(mrmTraj);
title("Multirotor Trajectory")
axis square
subplot(1,2,2)
show(fwmTraj);
title("Fixed-Wing Trajectory")
axis square
```

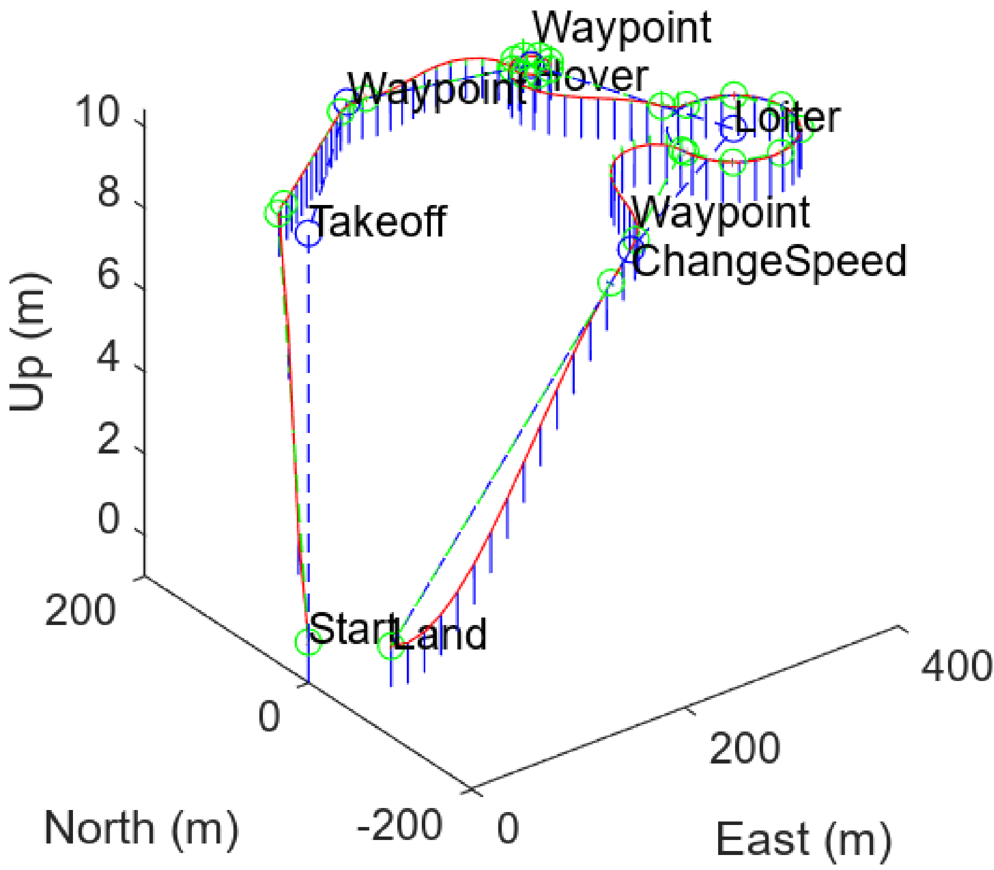


Plot the mission and flight trajectories overlapping.

```
figure
show(mission);
hold on
show(mrmTraj);
hold off
title("Mission Using Multirotor Trajectory")
axis square
```




```
show(mission);
hold on
show(fwmTraj);
hold off
title("Mission Using Fixed-Wing Trajectory")
axis square
```



Create and Query UAV Flight Trajectory

Create a set of waypoints for both the multirotor and the fixed-wing UAV to follow.

```
wpts = [0  0  0;
        2  2 -2;
       10 10 -3;
       12 12 -6];
numwpts = size(wpts);
```

Specify additional trajectory information, such as desired velocities, accelerations, jerks, snaps, and yaws, as well as start time, an end time, and times of arrival.

```
vels = 2*ones(numwpts);
accs = ones(numwpts);
jerks = zeros(numwpts);
snaps = zeros(numwpts);
yaws = zeros(1,numwpts(1));
```

```
starttime = 0;
endtime = 8;
toas = linspace(starttime,endtime,numwpts(1));
```

Use the trajectory information to create the flight trajectories for the multirotor and the fixed-wing UAVs. Query and display the trajectories.

```
mrft = multirotorFlightTrajectory(wpts,vels,accs,jerks,snaps,yaws,toas);
fwft = fixedwingFlightTrajectory(wpts,vels,toas);
query(mrft,1:4)
```

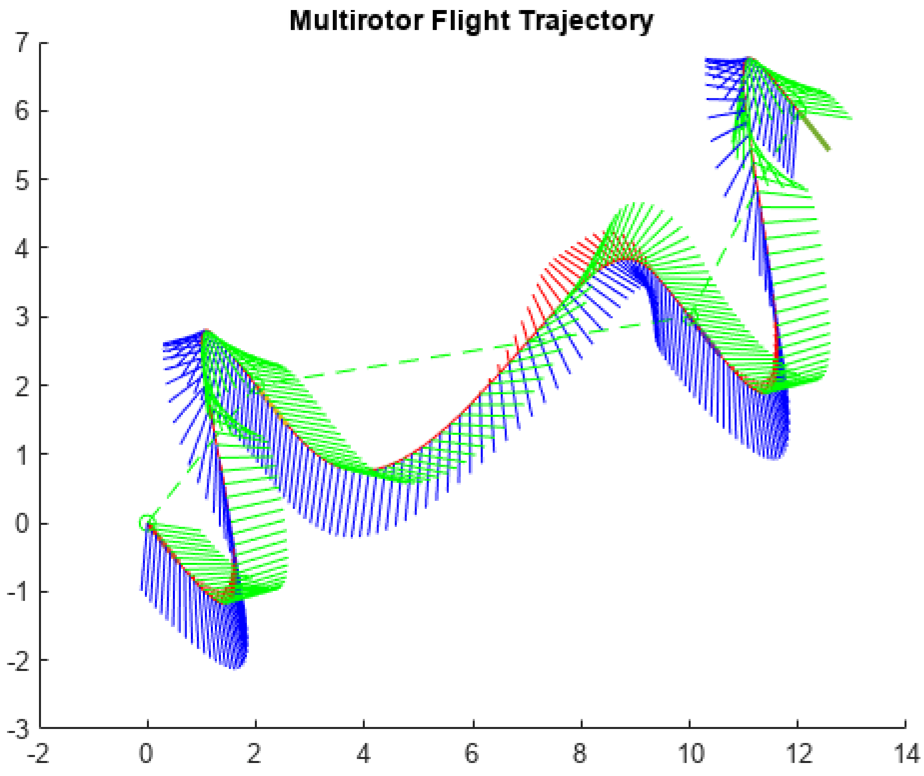
```
ans = 4×16

    1.6184    1.6184    0.7520   -0.0243   -0.0243   -2.8758   -4.6045   -4.6045   -9.1669    0.9
    1.0236    1.0236   -2.7807    0.5482    0.5482   -0.6198    3.9704    3.9704    8.6424    0.7
    2.7277    2.7277   -1.2947    2.4069    2.4069    2.1026    1.7442    1.7442   -1.3857    0.9
    6.4028    6.4028   -2.0972    4.4609    4.4609   -3.8447   -1.1875   -1.1875   -1.1875    0.9
```

```
query(fwft,1:4)
```

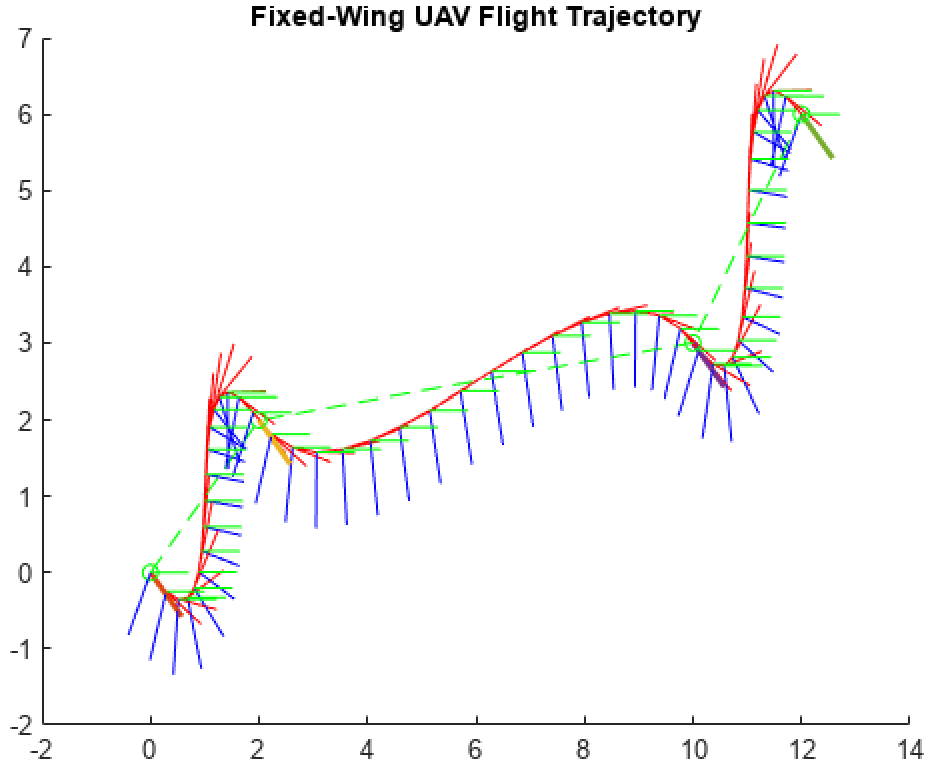
```
ans = 4×16

    0.9453    0.9453   -0.3203    0.2422    0.2422   -1.8672   -0.7031   -0.7031   -1.5469    0.7
    1.1875    1.1875   -2.1875    0.5938    0.5938   -1.0938    1.4063    1.4062    3.0938    0.8
    2.7813    2.7812   -1.6055    2.6563    2.6562    0.4414    1.6875    1.6875   -4.0078    0.9
    6.0000    6.0000   -2.5000    3.5000    3.5000   -1.5625   -0.0000   -0.0000    0.0000    0.9
```

Visualize both the multirotor flight trajectory and the fixed-wing flight trajectory.

```
ax = show(mrft,NumSamples=200);
title("Multirotor Flight Trajectory")
view([0 0])
```




```
show(fwft,NumSamples=50);
title("Fixed-Wing UAV Flight Trajectory")
view([0 0])
```



Version History

Introduced in R2022b

Extended Capabilities

C/C++ Code Generation

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

See Also

`fixedwingFlightTrajectory` | `multirotorMissionParser` | `uavMission`

copy

Copy flight trajectory

Syntax

```
trajCopy = copy(traj)
```

Description

`trajCopy = copy(traj)` creates a deep copy of a `fixedwingTrajectory` or a `multirotorFlightTrajectory` object. The copy has the same properties as the original.

Input Arguments

`traj` — Flight trajectory
`multirotorFlightTrajectory` object | `fixedwingFlightTrajectory` object

Flight trajectory, specified as a `multirotorFlightTrajectory` or `fixedwingFlightTrajectory` object.

Output Arguments

`trajCopy` — Deep copy of flight trajectory
`multirotorFlightTrajectory` object | `fixedwingFlightTrajectory` object

Deep copy of the flight trajectory, returned as a `multirotorFlightTrajectory` object or a `fixedwingFlightTrajectory` object. The copy is of the same object type, and has the same properties, as the object specified to `traj`.

Version History

Introduced in R2022b

See Also

`fixedwingFlightTrajectory` | `multirotorFlightTrajectory`

show

Visualize the UAV trajectory

Syntax

```
AX = show(traj)
AX = show( ___ ,Name=Value)
```

Description

`AX = show(traj)` visualizes the UAV trajectory in 3-D. The visualization includes the waypoints, velocity direction at each waypoint, flight trajectory and UAV body frames along flight trajectory.

`AX = show( ___ ,Name=Value)` specifies options using one or more name-value arguments. For example, `show(traj,NumSamples=300)` draws 300 body frames for the UAV.

Examples

Create and Query UAV Flight Trajectory

Create a set of waypoints for both the multirotor and the fixed-wing UAV to follow.

```
wpts = [0  0  0;
        2  2 -2;
       10 10 -3;
       12 12 -6];
numwpts = size(wpts);
```

Specify additional trajectory information, such as desired velocities, accelerations, jerks, snaps, and yaws, as well as start time, an end time, and times of arrival.

```
vels = 2*ones(numwpts);
accs = ones(numwpts);
jerks = zeros(numwpts);
snaps = zeros(numwpts);
yaws = zeros(1,numwpts(1));
starttime = 0;
endtime = 8;
toas = linspace(starttime,endtime,numwpts(1));
```

Use the trajectory information to create the flight trajectories for the multirotor and the fixed-wing UAVs. Query and display the trajectories.

```
mrft = multirotorFlightTrajectory(wpts,vels,accs,jerks,snaps,yaws,toas);
fwft = fixedwingFlightTrajectory(wpts,vels,toas);
query(mrft,1:4)
```

```
ans = 4×16

    1.6184    1.6184    0.7520   -0.0243   -0.0243   -2.8758   -4.6045   -4.6045   -9.1669    0.9
    1.0236    1.0236   -2.7807    0.5482    0.5482   -0.6198    3.9704    3.9704    8.6424    0.7
```

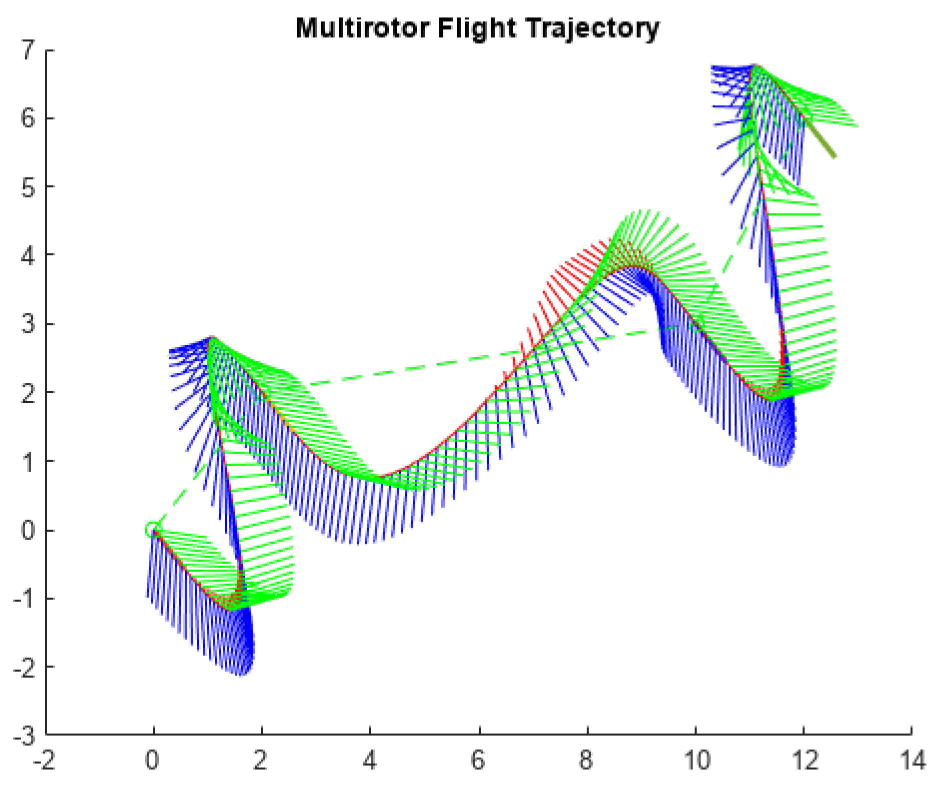
```
    2.7277    2.7277   -1.2947    2.4069    2.4069    2.1026    1.7442    1.7442   -1.3857    0.9
    6.4028    6.4028   -2.0972    4.4609    4.4609   -3.8447   -1.1875   -1.1875   -1.1875    0.9
```

```
query(fwft,1:4)
```

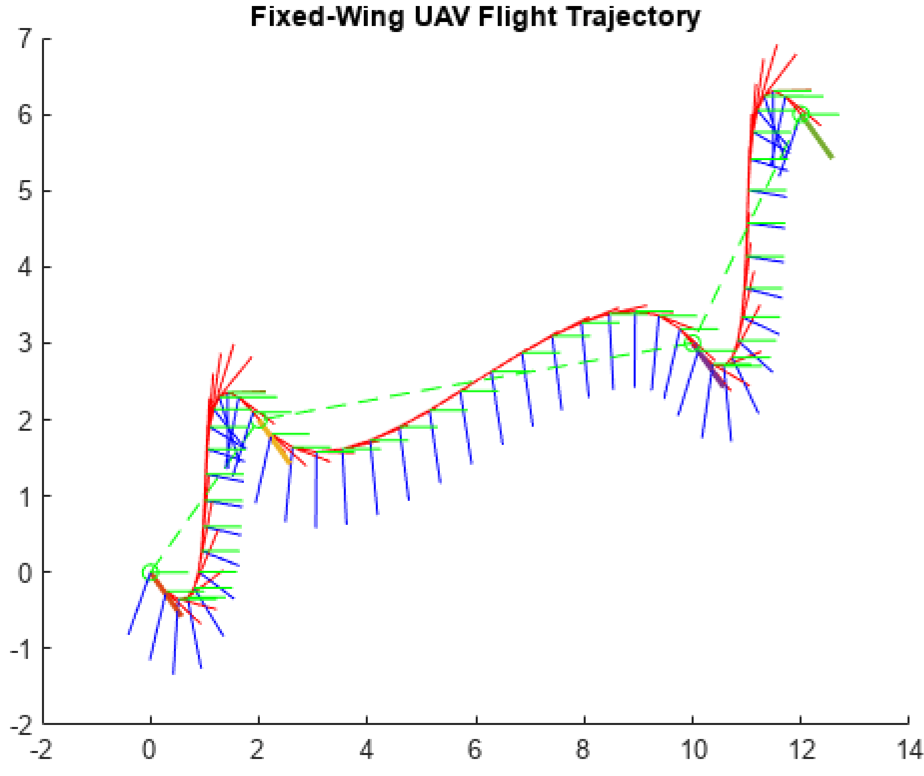
```
ans = 4×16

    0.9453    0.9453   -0.3203    0.2422    0.2422   -1.8672   -0.7031   -0.7031   -1.5469    0.7
    1.1875    1.1875   -2.1875    0.5938    0.5938   -1.0938    1.4063    1.4062    3.0938    0.8
    2.7813    2.7812   -1.6055    2.6563    2.6562    0.4414    1.6875    1.6875   -4.0078    0.9
    6.0000    6.0000   -2.5000    3.5000    3.5000   -1.5625   -0.0000   -0.0000    0.0000    0.9
```

Visualize both the multirotor flight trajectory and the fixed-wing flight trajectory.

```
ax = show(mrft,NumSamples=200);
title("Multirotor Flight Trajectory")
view([0 0])
```



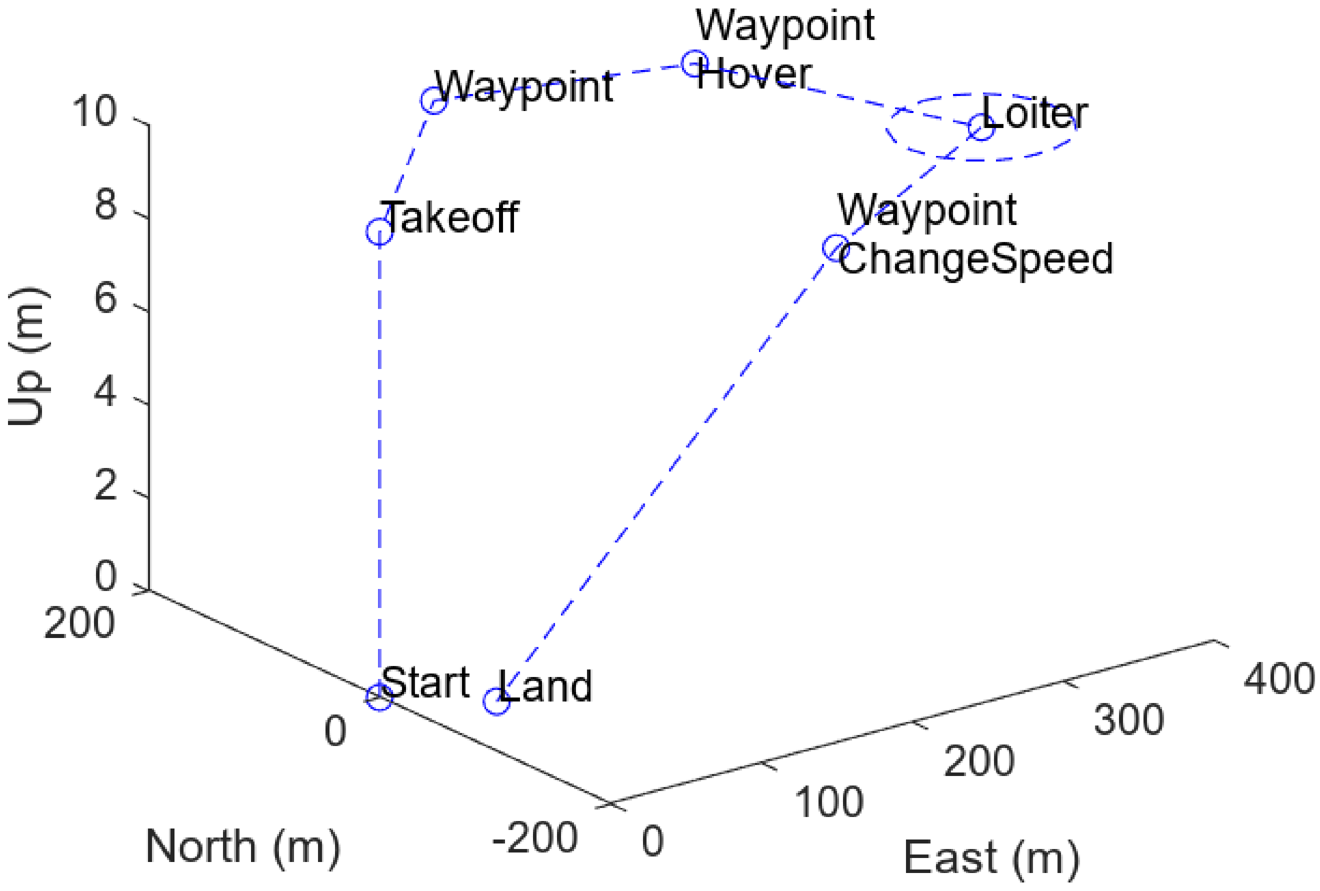

```
show(fwft,NumSamples=50);
title("Fixed-Wing UAV Flight Trajectory")
view([0 0])
```



Generate Flight Trajectory for UAV Mission

Create a UAV mission by using the flight plan stored in a `.plan` file and show the mission.

```
mission = uavMission(PlanFile="flight.plan");
show(mission);
```



Create parsers for a multirotor UAV and a fixed-wing UAV.

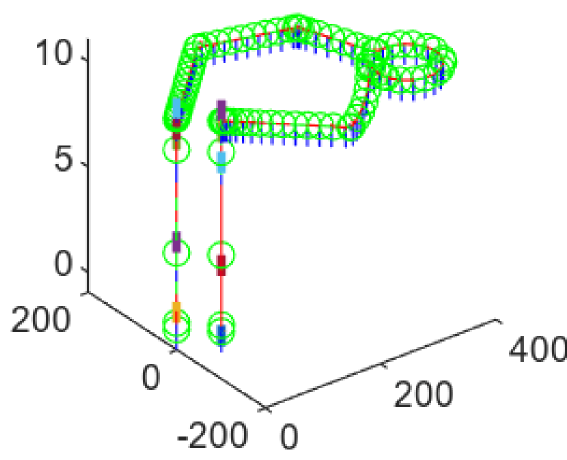
```
mrmParser = multirotorMissionParser(TransitionRadius=2,TakeoffSpeed=2);
fwmParser = fixedwingMissionParser(TransitionRadius=15,TakeoffPitch=10);
```

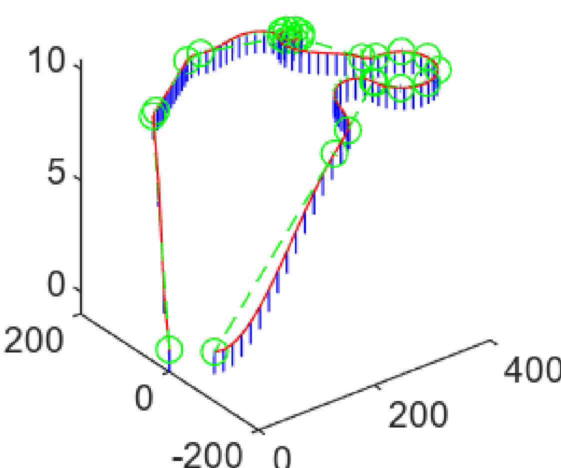
Generate one flight trajectory using each parser.

```
mrmTraj = parse(mrmParser,mission);
fwmTraj = parse(fwmParser,mission);
```

Visualize the mission and the flight trajectories separately.

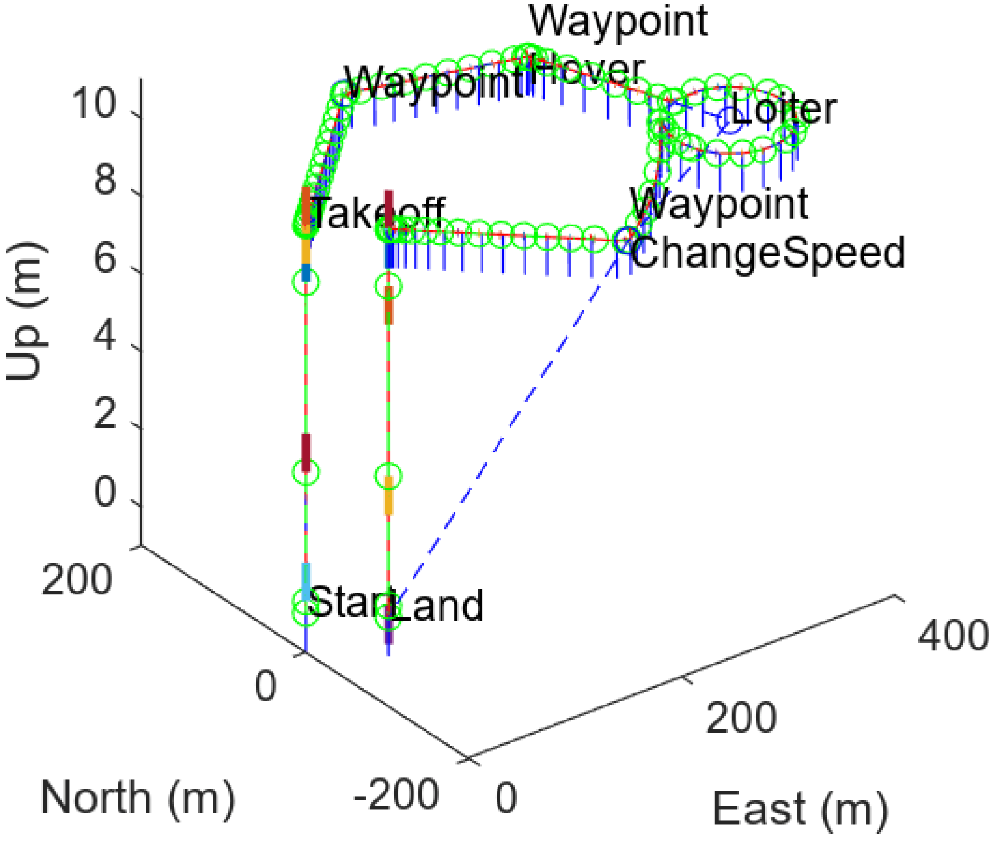
```
figure
subplot(1,2,1)
show(mrmTraj);
title("Multirotor Trajectory")
axis square
subplot(1,2,2)
show(fwmTraj);
title("Fixed-Wing Trajectory")
axis square
```



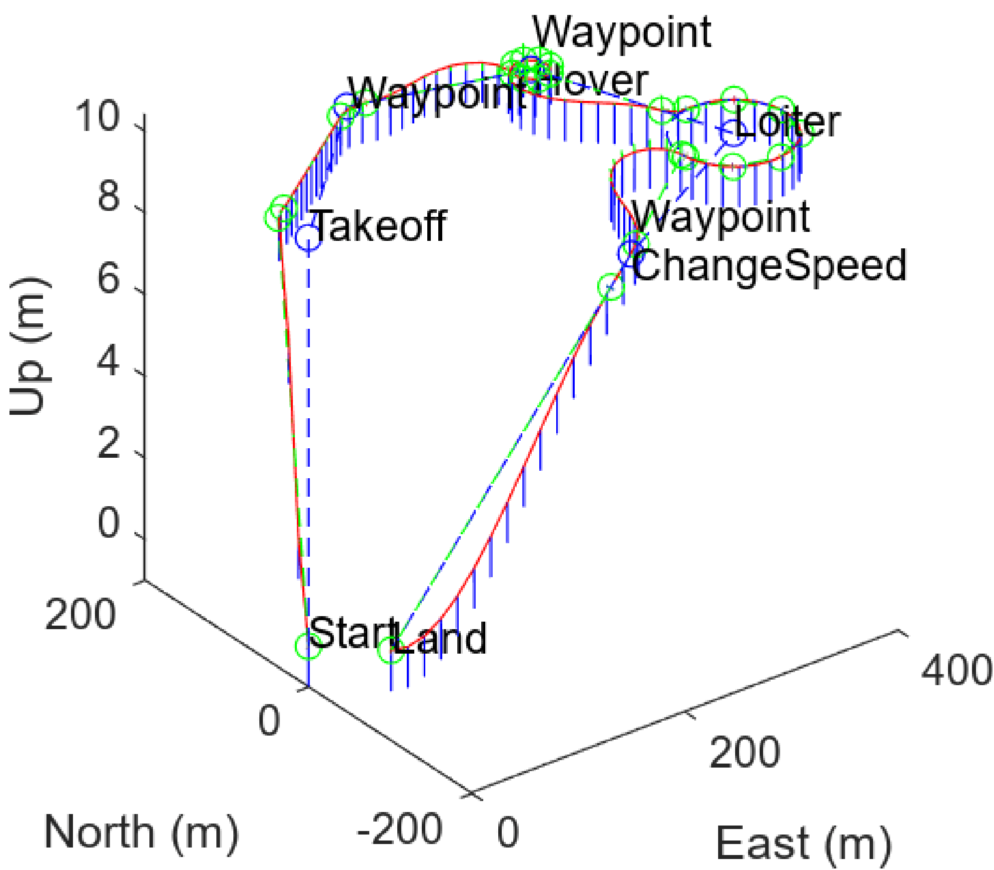


Plot the mission and flight trajectories overlapping.

```
figure
show(mission);
hold on
show(mrmTraj);
hold off
title("Mission Using Multirotor Trajectory")
axis square
```




```
show(mission);
hold on
show(fwmTraj);
hold off
title("Mission Using Fixed-Wing Trajectory")
axis square
```



Input Arguments

`traj` — Flight trajectory
`multirotorFlightTrajectory` object | `fixedwingFlightTrajectory` object

Flight trajectory, specified as a `multirotorFlightTrajectory` or `fixedwingFlightTrajectory` object.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Example: `show(traj,NumSamples=300)` draws 300 body frames for the UAV.

`Parent` — Parent axes
`gca` (default) | `Axes` object

Parent axes, specified as an `Axes` object.

`NumSamples` — Number of UAV body frames to draw
`100` (default) | positive integer

Number of UAV body frames to draw, specified as a positive integer.

Example: `show(traj,NumSamples=300)`

`StartTime` — Start time of trajectory drawing
`traj.StartTime` (default) | nonnegative numeric scalar

Start time of trajectory drawing, specified as a nonnegative numeric scalar, in seconds.

Example: `show(traj,StartTime=2)`

`EndTime` — End time of trajectory drawing
`traj.EndTime` (default) | positive numeric scalar

End time of trajectory drawing, specified as a positive numeric scalar, in seconds.

Example: `show(traj,EndTime=8)`

`FrameSize` — Length of UAV frame axes
`1` (default) | positive numeric scalar

Length of the UAV frame axes, specified as a positive numeric scalar, in meters.

Example: `show(traj,FrameSize=1.8)`

`VelocityLineSize` — Length of velocity direction lines
`1` (default) | nonnegative numeric scalar

Length of the velocity direction lines, specified as a nonnegative numeric scalar, in meters.

Example: `show(traj,VelocityLineSize=2.3)`

`VelocityLineWidth` — Width of velocity direction lines
`2` (default) | nonnegative numeric scalar

Width of the velocity direction lines, specified as a nonnegative numeric scalar, in meters.

Example: `show(traj,VelocityLineWidth=2.3)`

Version History

Introduced in R2022b

See Also

`multirotorFlightTrajectory` | `fixedwingFlightTrajectory`

query

Get UAV motion vectors at timestamps

Syntax

```
motions = query(traj,timestamps)
```

Description

`motions = query(traj,timestamps)` gets the UAV motion vectors `motions` from the trajectory `traj` at the timestamps `timestamps`.

Examples

Create and Query UAV Flight Trajectory

Create a set of waypoints for both the multirotor and the fixed-wing UAV to follow.

```
wpts = [0  0  0;
        2  2 -2;
       10 10 -3;
       12 12 -6];
numwpts = size(wpts);
```

Specify additional trajectory information, such as desired velocities, accelerations, jerks, snaps, and yaws, as well as start time, an end time, and times of arrival.

```
vels = 2*ones(numwpts);
accs = ones(numwpts);
jerks = zeros(numwpts);
snaps = zeros(numwpts);
yaws = zeros(1,numwpts(1));
starttime = 0;
endtime = 8;
toas = linspace(starttime,endtime,numwpts(1));
```

Use the trajectory information to create the flight trajectories for the multirotor and the fixed-wing UAVs. Query and display the trajectories.

```
mrft = multirotorFlightTrajectory(wpts,vels,accs,jerks,snaps,yaws,toas);
fwft = fixedwingFlightTrajectory(wpts,vels,toas);
query(mrft,1:4)
```

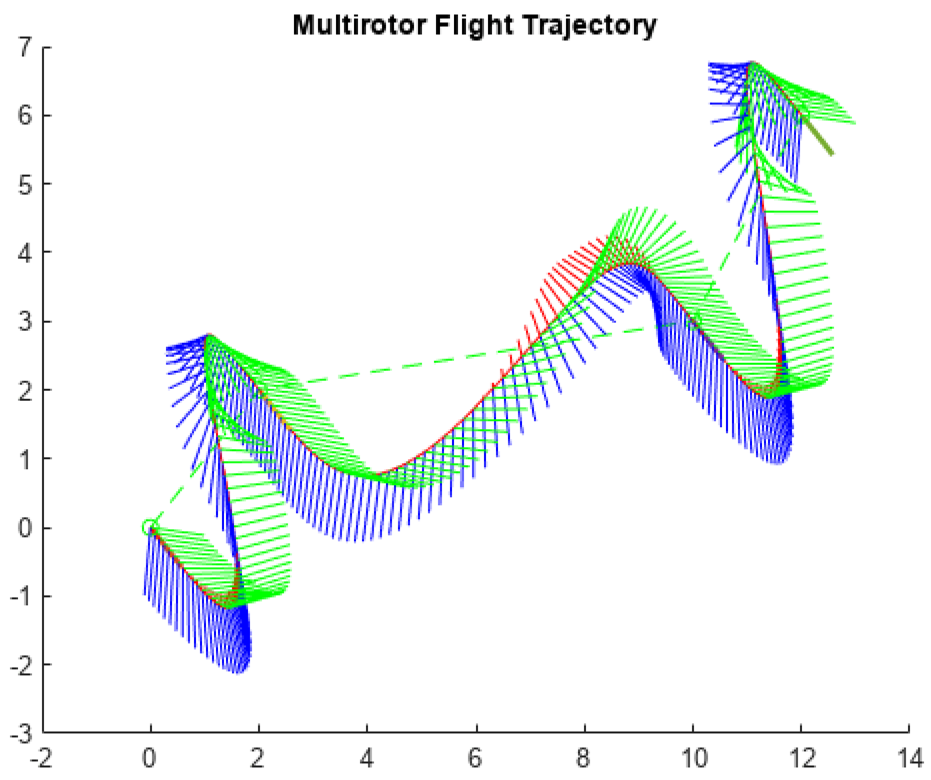
```
ans = 4×16

    1.6184    1.6184    0.7520   -0.0243   -0.0243   -2.8758   -4.6045   -4.6045   -9.1669    0.9
    1.0236    1.0236   -2.7807    0.5482    0.5482   -0.6198    3.9704    3.9704    8.6424    0.7
    2.7277    2.7277   -1.2947    2.4069    2.4069    2.1026    1.7442    1.7442   -1.3857    0.9
    6.4028    6.4028   -2.0972    4.4609    4.4609   -3.8447   -1.1875   -1.1875   -1.1875    0.9
```

```
query(fwft,1:4)
```

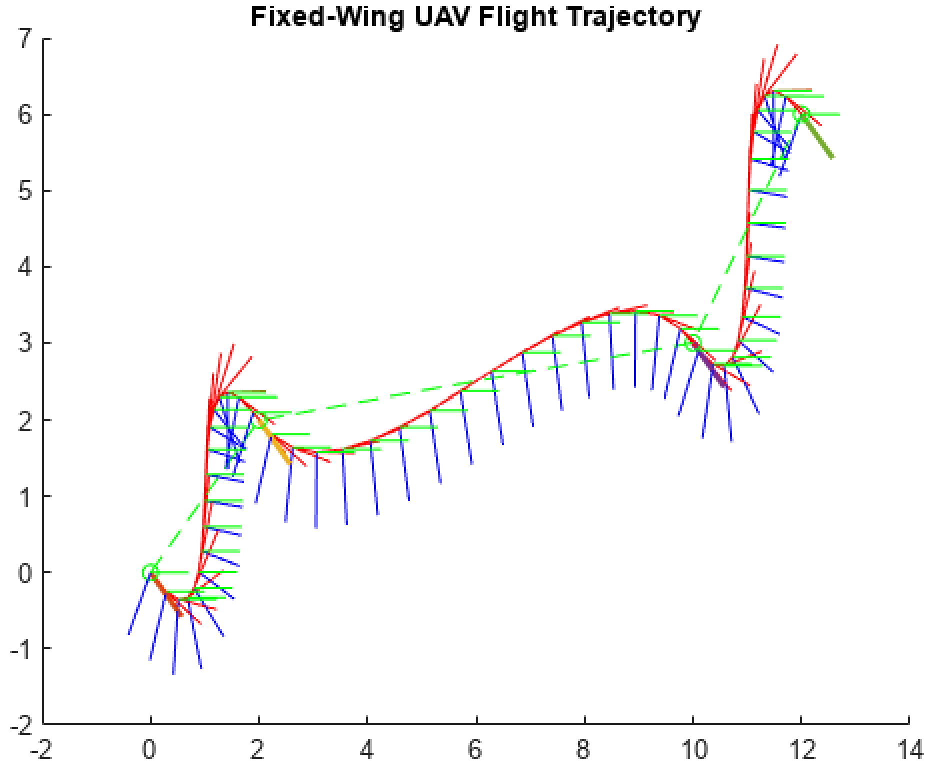
```
ans = 4×16

    0.9453    0.9453   -0.3203    0.2422    0.2422   -1.8672   -0.7031   -0.7031   -1.5469    0.7
    1.1875    1.1875   -2.1875    0.5938    0.5938   -1.0938    1.4063    1.4062    3.0938    0.8
    2.7813    2.7812   -1.6055    2.6563    2.6562    0.4414    1.6875    1.6875   -4.0078    0.9
    6.0000    6.0000   -2.5000    3.5000    3.5000   -1.5625   -0.0000   -0.0000    0.0000    0.9
```

Visualize both the multirotor flight trajectory and the fixed-wing flight trajectory.

```
ax = show(mrft,NumSamples=200);
title("Multirotor Flight Trajectory")
view([0 0])
```




```
show(fwft,NumSamples=50);
title("Fixed-Wing UAV Flight Trajectory")
view([0 0])
```



Input Arguments

traj — Flight trajectory

multirotorFlightTrajectory object | fixedwingFlightTrajectory object

Flight trajectory, specified as a `multirotorFlightTrajectory` or `fixedwingFlightTrajectory` object.

timestamps — Timestamps

N-element vector

Timestamps, specified as an *N*-element vector containing time values, in seconds. *N* is the total number of specified timestamps.

Output Arguments

motions — Motion vectors

N-by-16 matrix

Motion vectors, returned as an *N*-by-16 matrix, where each row contains the motion vector of the UAV, [*X Y Z VX VY VZ AX AY AZ QW QX QY QZ P Q R*], at the time specified by the corresponding element of the `timestamps` vector. *N* is the total number of specified timestamps.

- *X*, *Y*, and *Z* are positions.
- *VX*, *VY*, and *VZ* are velocities.
- *AX*, *AY*, and *AZ* are accelerations.
- *QW*, *QX*, *QY*, and *QZ*, are quaternions for rotations from the local north-east-down (NED) reference frame to the body frame.
- *P*, *Q*, and *R* are angular velocities along local the NED reference frame axes.

All motion values are expressed in the local NED frame.

For timestamps outside the `StartTime` and `EndTime` of the specified trajectory object, the values of the corresponding motion vector row are `NaN`.

Version History

Introduced in R2022b

See Also

`multirotorFlightTrajectory` | `fixedwingFlightTrajectory`

uavPathManager

Compute and execute a UAV autonomous mission

Description

The `uavPathManager` System object computes mission parameters for an unmanned aerial vehicle (UAV) by sequentially switching between the mission points specified in the **MissionData** property. The **MissionCmd** property changes the execution order at runtime. The object supports both multirotor and fixed-wing UAV types.

To compute mission parameters:

1. Create the `uavPathManager` object and set its properties.
2. Call the object with arguments, as if it were a function.

To learn more about how System objects work, see What Are System Objects?

Creation

Syntax

```
pathManagerObj = uavPathManager
pathManagerObj = uavPathManager(Name,Value)
```

Description

`pathManagerObj = uavPathManager` creates a UAV path manager System object with default property values.

`pathManagerObj = uavPathManager(Name,Value)` creates a UAV path manager 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`.

Example: `uavPathManager('UAVType','fixed-wing')`

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.

`UAVType` — Type of UAV
`'multirotor'` (default) | `'fixed-wing'`

Type of UAV, specified as either `'multirotor'` or `'fixed-wing'`.

Data Types: `string`

`LoiterRadius` — Loiter radius for fixed-wing UAV
25 (default) | positive numeric scalar

Loiter radius for the fixed-wing UAV, specified as a positive numeric scalar in meters.

Dependencies: To enable this parameter, set the **UAV type** property to `'fixed-wing'`.

Data Types: `single` | `double`

`MissionData` — UAV mission data
n-by-1 array of structures

UAV mission data, specified as an *n*-by-1 array of structures. *n* is the number of mission points. The fields of each structure are:

- `mode` — Mode of the mission point, specified as an 8-bit unsigned integer between 1 and 6.
- `position` — Position of the mission point, specified as a three-element column vector of `[x;y;z]`. *x*, *y*, and *z* is the position in north-east-down (NED) coordinates specified in meters.
- `params` — Parameters of the mission point, specified as a four-element column vector.

This table describes types of `mode` and the corresponding values for the `position` and `params` fields in a mission point structure.

mode	position	params	Mode description
`uint8(1)`	`[x;y;z]`	`[p1;p2;p3;p4]`	**Takeoff** — Take off from the ground and travel towards the specified position
`uint8(2)`	`[x;y;z]`	`[yaw;radius;p3;p4]` *yaw* — Yaw angle in radians in the range `[-pi, pi]` *radius* — Transition radius in meters	**Waypoint** — Navigate to waypoint

mode	position	params	Mode description
`uint8(3)`	`[x;y;z]` *x*, *y*, and *z* is the center of the circular orbit in NED coordinates specified in meters	`[radius;turnDir;numTurns;p4]` *radius* — Radius of the orbit in meters *turnDir* — Turn direction, specified as one of these: • `1` — Clockwise turn • `–1` — Counter-clockwise turn • `0` — Automatic selection of turn direction *numTurns* — Number of turns	**Orbit** — Orbit along the circumference of a circle defined by the parameters
`uint8(4)`	`[x;y;z]`	`[p1;p2;p3;p4]`	**Land** — Land at the specified position
`uint8(5)`	`[x;y;z]` The launch position is specified in the **Home** property	`[p1;p2;p3;p4]`	**RTL** — Return to launch position
`uint8(6)`	`[x;y;z]`	`[p1;p2;p3;p4]` *p1*, *p2*, *p3*, and *p4* are user-specified parameters corresponding to the custom mission point	**Custom** — Custom mission point

Note *p1*, *p2*, *p3*, and *p4* are user-specified parameters.

Example: `[struct('mode',uint8(1),'position',[0;0;100],'params',[0;0;0;0])]`

Tunable: Yes

`IsModeDone` — Determine if mission point was executed

false (default) | true

Determine if the mission point was executed, specified as `true` (`1`) or `false` (`0`).

Tunable: Yes

Data Types: `logical`

`MissionCmd` — Command to change mission
`uint8(0)` (default) | 8-bit unsigned integer between 0 and 3

Command to change mission at runtime, specified as an 8-bit unsigned integer between 0 and 3.

This table describes the possible mission commands.

Mission Command	Description
`uint8(0)`	**Default** — Execute the mission from first to the last mission point in the sequence
`uint8(1)`	**Hold** — Hold at the current mission point Loiter around the current position for fixed-wing, and hover at the current position for multirotor UAVs
`uint8(2)`	**Repeat** — Repeat the mission after reaching the last mission point
`uint8(3)`	**RTL** — Execute return to launch (RTL) mode After **RTL**, the mission resumes if the **MissionCmd** property is changed to **Default** or **Repeat**

Tunable: Yes

Data Types: `uint8`

`Home` — UAV home location
three-element column vector

UAV home location, specified as a three-element column vector of `[x;y;z]`. *x*, *y*, and *z* is the position in north-east-down (NED) coordinates specified in meters.

Tunable: Yes

Data Types: `single` | `double`

Usage

Syntax

```
missionParams = pathManagerObj(pose)
```

Description

```
missionParams = pathManagerObj(pose)
```

Input Arguments

`pose` — Current UAV pose
four-element column vector

Current UAV pose, specified as a four-element column vector of `[x;y;z;courseAngle]`. *x*, *y*, and *z* is the current position in north-east-down (NED) coordinates specified in meters. *courseAngle* specifies the course angle in radians in the range `[-pi, pi]`.

Data Types: `single` | `double`

Output Arguments

`missionParams` — UAV mission parameters

2-by-1 array of structures

UAV mission parameters, returned as a 2-by-1 array of structures. The first row of the array contains the structure of the current mission point, and the second row of the array contains the structure of the previous mission point. The fields of each structure are:

- `mode` — Mode of the mission point, returned as an 8-bit unsigned integer between 0 and 7.
- `position` — Position of the mission point based on the mode, returned as a three-element column vector of `[x;y;z]`. *x*, *y*, and *z* is the position in north-east-down (NED) coordinates specified in meters.
- `params` — Parameters of the mission point based on the mode, returned as a four-element column vector.

At start of simulation, the previous mission point is set to the **Armed** mode.

```
struct('mode',uint8(0),'position',[x;y;z],'params',[-1;-1;-1;-1])
```

Note The **Armed** mode cannot be configured by the user.

Set the end mission point to **RTL** or **Land** mode, else the end mission point is automatically set to **Hold** mode.

- Multirotor UAVs hover at the current position.

  ```
  struct('mode',uint8(7),'position',[x;y;z],'params',[-1;-1;-1;-1])
  ```
- Fixed-wing UAVs loiter around the current position.

  ```
  struct('mode',uint8(7),'position',[x;y;z],'params',[radius;turnDir;-1;-1])
  ```

Note The **Hold** mode cannot be configured by the user.

This table describes the output mission parameters depending on the mission mode.

Current Mission Mode	Output Mission Parameters			
	Mission Points	mode	position	params
Takeoff	Row 1: Current	`uint8(1)`	`[x;y;z]`	`[p1;p2;p3;p4]`
	Row 2: Previous	`mode` of the previous mission point	`position` of the previous mission point	`params` of the previous mission point

<table>
<tr><th rowspan="2">Current Mission Mode</th><th colspan="4">Output Mission Parameters</th></tr>
<tr><th>Mission Points</th><th>mode</th><th>position</th><th>params</th></tr>
<tr><td rowspan="2">Waypoint</td><td>Row 1: Current</td><td><code>uint8(2)</code></td><td><code>[x;y;z]</code></td><td><code>[yaw;radius;p3;p4]</code>

yaw — Yaw angle in radians in the range <code>[-pi, pi]</code>

radius — Transition radius in meters</td></tr>
<tr><td>Row 2: Previous</td><td><code>mode</code> of the previous mission point</td><td><code>position</code> of the previous mission point</td><td>• <code>[yaw;radius;p3;p4]</code> if the previous mission point was Takeoff
• <code>[courseAngle;25;p3;p4]</code> otherwise

courseAngle — Angle of the line segment between the previous and the current position, specified in radians in the range <code>[-pi, pi]</code></td></tr>
<tr><td>Orbit</td><td>Row 1: Current</td><td><code>uint8(3)</code></td><td><code>[x;y;z]</code>

x, y, and z is the center of the circular orbit in NED coordinates specified in meters</td><td><code>[radius;turnDir;numTurns;p4]</code>

radius — Radius of the orbit in meters

turnDir — Turn direction, specified as one of these:
• <code>1</code> — Clockwise turn
• <code>–1</code> — Counter-clockwise turn
• <code>0</code> — Automatic selection of turn direction

numTurns — Number of turns</td></tr>
</table>

Current Mission Mode	Output Mission Parameters			
	Mission Points	**mode**	**position**	**params**
	Row 2: Previous	`mode` of the previous mission point	`position` of the previous mission point	`params` of the previous mission point
Land	Row 1: Current	`uint8(4)`	[*x*;*y*;*z*]	[*p1*;*p2*;*p3*;*p4*]
	Row 2: Previous	`mode` of the previous mission point	`position` of the previous mission point	`params` of the previous mission point
RTL	Row 1: Current	`uint8(5)`	[*x*;*y*;*z*] The launch position is specified in the **Home** property	[*p1*;*p2*;*p3*;*p4*]
	Row 2: Previous	`mode` of the previous mission point	`position` of the previous mission point	`params` of the previous mission point
Custom	Row 1: Current	`uint8(6)`	[*x*;*y*;*z*]	[*p1*;*p2*;*p3*;*p4*] *p1*, *p2*, *p3*, and *p4* are user-specified parameters corresponding to the custom mission point
	Row 2: Previous	`mode` of the previous mission point	`position` of the previous mission point	`params` of the previous mission point

Note *p1*, *p2*, *p3*, and *p4* are user-specified parameters.

This table describes the output mission parameters when the input to the **MissionCmd** property is set to **Hold** mode.

UAV Type	Output Mission Parameters			
	Mission Points	**mode**	**position**	**params**
Multirotor	Row 1: Current	`uint8(7)`	[*x*;*y*;*z*]	`[-1;-1;-1;-1]`
	Row 2: Previous	`mode` of the previous mission point	`position` of the previous mission point	`params` of the previous mission point

UAV Type	Output Mission Parameters			
	Mission Points	**mode**	**position**	**params**
Fixed-Wing	Row 1: Current	`uint8(7)`	`[x;y;z]` *x*, *y*, and *z* is the center of the circular orbit in NED coordinates specified in meters	`[radius;turnDir;-1;-1]` *radius* — Loiter radius is specified in the **LoiterRadius** property *turnDir* — Turn direction is specified as `0` for automatic selection of turn direction
	Row 2: Previous	`mode` of the previous mission point	`position` of the previous mission point	`params` of the previous mission point

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

Version History

Introduced in R2020b

See Also

`uavWaypointFollower` | `uavOrbitFollower` | `fixedwing` | `multirotor`

uavPlatform

UAV platform for sensors in scenario

Description

The `uavPlatform` object represents an unmanned aerial vehicle (UAV) platform in a given UAV scenario. Use the platform to define and track the trajectory of a UAV in the scenario. To simulate sensor readings for the platform, mount sensors such as the `gpsSensor`, `insSensor`, and `uavLidarPointCloudGenerator` System object to the platform as `uavSensor` objects. Add a body mesh to the platform for visualization using the `updateMesh` object function. Set geofencing limitations using the `addGeoFence` object and check those limits using the `checkPermission` object function.

Creation

Syntax

```
platform = uavPlatform(name,scenario)
platform = uavPlatform(name,scenario,Name,Value)
```

Description

`platform = uavPlatform(name,scenario)` creates a platform with a specified name `name` and adds it to the scenario, specified as a `uavScenario` object. Specify the name argument as a string scalar. The name argument sets the `Name` property.

`platform = uavPlatform(name,scenario,Name,Value)` specifies options using one or more name-value pair arguments. You can specify properties as name-value pair arguments as well. For example, `uavPlatform("UAV1",scene,'StartTime',10)` sets the initial time for the platform trajectory to 10 seconds. Enclose each property name in quotes

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `'StartTime',10` sets the initial time of the platform trajectory to 10 seconds.

`StartTime` — Initial time of platform trajectory
`0` (default) | scalar in seconds

Initial time of the platform trajectory, specified as the comma-separated pair consisting of `'StartTime'` and a scalar in seconds. The UAV platform starts following the trajectory at the time of the first waypoint in the trajectory plus the start time of the platform.

Data Types: `double`

InitialPosition — Initial platform position for UAV
[0 0 0] (default) | vector of the form [x y z]

Initial platform position for UAV, specified as the comma-separated pair consisting of 'InitialPosition' and a vector of the form [x y z]. Only specify this name-value pair if not specifying the Trajectory property.

Data Types: double

InitialOrientation — Initial platform orientation for UAV
[1 0 0 0] (default) | vector of the form [w x y z]

Initial platform orientation for UAV, specified as the comma-separated pair consisting of 'InitialOrientation' and a vector of the form [w x y z], representing a quaternion. Only specify this name-value pair if not specifying the Trajectory property.

Data Types: double

InitialVelocity — Initial platform velocity for UAV
[0 0 0] (default) | vector of the form [vx vy vz]

Initial platform velocity for UAV, specified as the comma-separated pair consisting of 'InitialVelocity' and a vector of the form [vx vy vz]. Only specify this name-value pair if not specifying the Trajectory property.

Data Types: double

InitialAcceleration — Initial platform acceleration for UAV
[0 0 0] (default) | vector of the form [ax ay az]

Initial platform acceleration for UAV, specified as the comma-separated pair consisting of 'InitialAcceleration' and a vector of the form [ax ay az]. Only specify this name-value pair if not specifying the Trajectory property.

Data Types: double

InitialAngularVelocity — Initial platform angular velocity for UAV
[0 0 0] (default) | three-element vector of the form [x y z] | vector

Initial platform angular velocity for UAV, specified as the comma-separated pair consisting of 'InitialAngularVelocity' and a three-element vector of the form [x y z]. The magnitude of the vector defines the angular speed in radians per second. The *xyz*-coordinates define the axis of clockwise rotation. Only specify this name-value pair if not specifying the Trajectory property.

Data Types: double

Trajectory — Trajectory for UAV platform motion
[] (default) | waypointTrajectory object | polynomialTrajectory object

Trajectory for UAV platform motion, specified as a waypointTrajectory or polynomialTrajectory object. By default, the platform is assumed to be stationary and at the origin. To move the platform at each simulation step of the scenario, use the move object function.

Note The uavPlatform object must specify the same ReferenceFrame property as the specified trajectory object.

ReferenceFrame — Reference frame for computing UAV platform motion
string scalar

Reference frame for computing UAV platform motion, specified as string scalar, which matches any reference frame in the uavScenario. All platform motion is computed relative to this inertial frame.

Data Types: string

Properties

Name — Identifier for UAV platform
string scalar | character vector

Identifier for the UAV platform, specified as a string scalar or character vector.

Example: "uav1"

Data Types: string | char

Trajectory — Trajectory for UAV platform motion
[] (default) | waypointTrajectory object | polynomialTrajectory object

Trajectory for UAV platform motion, specified as a waypointTrajectory or polynomialTrajectory object. When specified as a waypointTrajectory or polynomialTrajectory object, the platform is moved along the trajectory during the scenario simulation. To move the platform at each simulation step of the scenario, use the move object function .

Note The uavPlatform object must specify the same ReferenceFrame property as the specified trajectory object.

ReferenceFrame — Reference frame for computing UAV platform motion
string scalar | character vector

Reference frame for computing UAV platform motion, specified as string scalar or character vector, which matches any reference frame in the uavScenario. The object computes all platform motion relative to this inertial frame.

Data Types: string | char

Mesh — UAV platform body mesh
extendedObjectMesh object

UAV platform body mesh, specified as an extendedObjectMesh object. The body mesh describes the 3-D model of the platform for visualization purposes.

MeshColor — UAV platform body mesh color
RGB triplet

UAV platform body mesh color when displayed in the scenario, specified as an RGB triplet.

Data Types: double

MeshTransform — Transform between UAV platform body and mesh frame
4-by-4 homogeneous transformation matrix

Transform between UAV platform body and mesh frame, specified as a 4-by-4 homogeneous transformation matrix that maps points in the platform mesh frame to points in the body frame.

Data Types: `double`

`Sensors` — Sensors mounted on UAV platform
array of `uavSensor` objects

Sensors mount on UAV platform, specified as an array of `uavSensor` objects.

`GeoFences` — Geofence restrictions for UAV platform
structure array

Geofence restrictions for UAV platform, specified as a structure array with these fields:

- `Geometry` — An `extendedObjectMesh` object representing the 3-D space for the geofence in the scenario frame.
- `Permission` — A logical scalar that indicates if the platform is permitted inside the geofence (`true`) or not permitted (`false`).

Data Types: `double`

Object Functions

move	Move UAV platform in scenario
read	Read UAV motion vector
updateMesh	Update body mesh for UAV platform
addGeoFence	Add geographical fencing to UAV platform
checkPermission	Check UAV platform permission based on geofencing

Examples

UAV Scenario Tutorial

Create a scenario to simulate unmanned aerial vehicle (UAV) flights between a set of buildings. The example demonstrates updating the UAV pose in open-loop simulations. Use the UAV scenario to visualize the UAV flight and generate simulated point cloud sensor readings.

Introduction

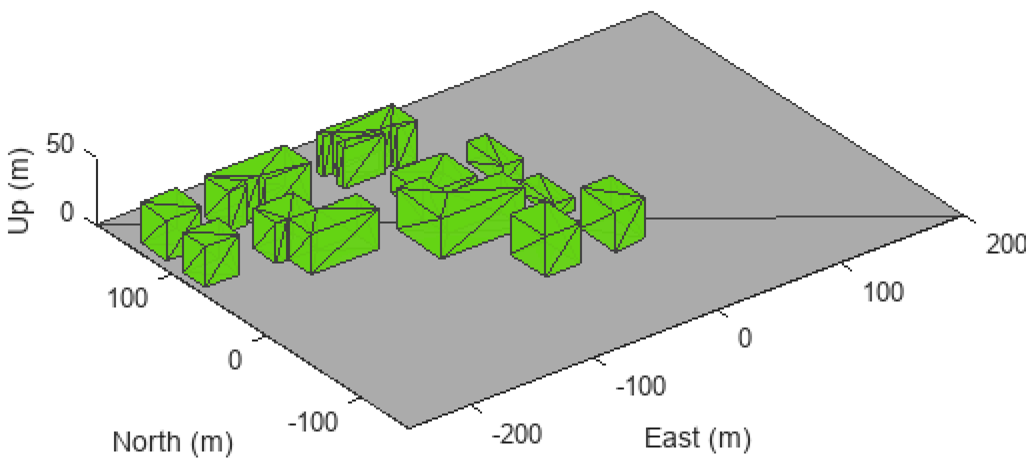
To test autonomous algorithms, a UAV scenario enables you to generate test cases and generate sensor data from the environment. You can specify obstacles in the workspace, provide trajectories of UAVs in global coordinates, and convert data between coordinate frames. The UAV scenario enables you to visualize this information in the reference frame of the environment.

Create Scenario with Polygon Building Meshes

A `uavScenario` object is a model consisting of a set of static obstacles and movable objects called platforms. Use `uavPlatform` objects to model fixed-wing UAVs, multirotors, and other objects within the scenario. This example builds a scenario consisting of a ground plane and 11 buildings as by extruded polygons. The polygon data for the buildings is loaded and used to add polygon meshes.

```
% Create the UAV scenario.
scene = uavScenario("UpdateRate",2,"ReferenceLocation",[75 -46 0]);
```

```
% Add a ground plane.
color.Gray = 0.651*ones(1,3);
color.Green = [0.3922 0.8314 0.0745];
color.Red = [1 0 0];
addMesh(scene,"polygon",{[-250 -150; 200 -150; 200 180; -250 180],[-4 0]},color.Gray)

% Load building polygons.
load("buildingData.mat");

% Add sets of polygons as extruded meshes with varying heights from 10-30.
addMesh(scene,"polygon",{buildingData{1}(1:4,:),[0 30]},color.Green)
addMesh(scene,"polygon",{buildingData{2}(2:5,:),[0 30]},color.Green)
addMesh(scene,"polygon",{buildingData{3}(2:10,:),[0 30]},color.Green)
addMesh(scene,"polygon",{buildingData{4}(2:9,:),[0 30]},color.Green)
addMesh(scene,"polygon",{buildingData{5}(1:end-1,:),[0 30]},color.Green)
addMesh(scene,"polygon",{buildingData{6}(1:end-1,:),[0 15]},color.Green)
addMesh(scene,"polygon",{buildingData{7}(1:end-1,:),[0 30]},color.Green)
addMesh(scene,"polygon",{buildingData{8}(2:end-1,:),[0 10]},color.Green)
addMesh(scene,"polygon",{buildingData{9}(1:end-1,:),[0 15]},color.Green)
addMesh(scene,"polygon",{buildingData{10}(1:end-1,:),[0 30]},color.Green)
addMesh(scene,"polygon",{buildingData{11}(1:end-2,:),[0 30]},color.Green)

% Show the scenario.
show3D(scene);
xlim([-250 200])
ylim([-150 180])
zlim([0 50])
```



Define UAV Platform and Mount Sensor

You can define a `uavPlatform` in the scenario as a carrier of your sensor models and drive them through the scenario to collect simulated sensor data. You can associate the platform with various meshes, such as `fixedwing`, `quadrotor`, and `cuboid` meshes. You can define a custom mesh defined ones represented by vertices and faces. Specify the reference frame for describing the motion of your platform.

Load flight data into the workspace and create a quadrotor platform using an NED reference frame. Specify the initial position and orientation based on loaded flight log data. The configuration of the UAV body frame orients the *x*-axis as forward-positive, the *y*-axis as right-positive, and the *z*-axis downward-positive.

```
load("flightData.mat")

% Set up platform
plat = uavPlatform("UAV",scene,"ReferenceFrame","NED", ...
    "InitialPosition",position(:,:,1),"InitialOrientation",eul2quat(orientation(:,:,1)));

% Set up platform mesh. Add a rotation to orient the mesh to the UAV body frame.
updateMesh(plat,"quadrotor",{10},color.Red,[0 0 0],eul2quat([0 0 pi]))
```

You can choose to mount different sensors, such as the `insSensor`, `gpsSensor`, or `uavLidarPointCloudGenerator` System objects to your UAV. Mount a lidar point cloud generator and a `uavSensor` object that contains the lidar sensor model. Specify a mounting location of the sensor that is relative to the UAV body frame.

```
lidarmodel = uavLidarPointCloudGenerator("AzimuthResolution",0.3324099,...
    "ElevationLimits",[-20 20],"ElevationResolution",1.25,...
    "MaxRange",90,"UpdateRate",2,"HasOrganizedOutput",true);

lidar = uavSensor("Lidar",plat,lidarmodel,"MountingLocation",[0,0,-1]);
```

Fly the UAV Platform Along Pre-Defined Trajectory and Collect Point Cloud Sensor Readings

Move the UAV along a pre-defined trajectory, and collect the lidar sensor readings along the way. This data could be used to test lidar-based mapping and localization algorithms.

Preallocate the `traj` and `scatterPlot` line plots and then specify the plot-specific data sources. During the simulation of the `uavScenario`, use the provided `plotFrames` output from the scene as the parent axes to visualize your sensor data in the correct coordinate frames.

Visualize the scene.

```
[ax,plotFrames] = show3D(scene);
```

Update plot view for better visibility.

```
xlim([-250 200])
ylim([-150 180])
zlim([0 50])
view([-110 30])
axis equal
hold on
```

Create a line plot for the trajectory. First create the plot with `plot3`, then manually modify the data source properties of the plot. This improves performance of the plotting.

```
traj = plot3(nan,nan,nan,"Color",[1 1 1],"LineWidth",2);
traj.XDataSource = "position(:,2,1:idx+1)";
traj.YDataSource = "position(:,1,1:idx+1)";
traj.ZDataSource = "-position(:,3,1:idx+1)";
```

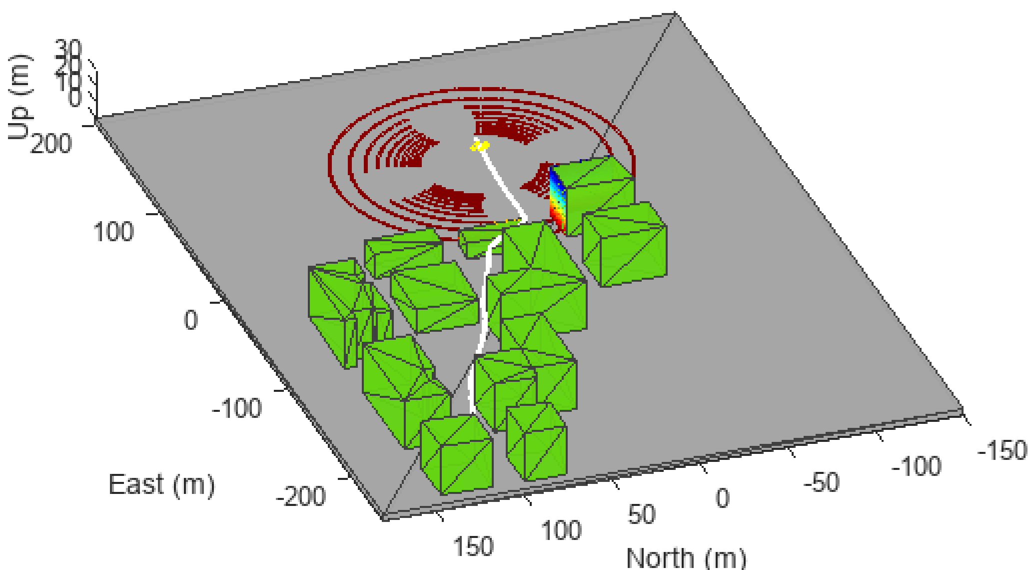
Create a scatter plot for the point cloud. Update the data source properties again.

```
colormap("jet")
pt = pointCloud(nan(1,1,3));
scatterplot = scatter3(nan,nan,nan,1,[0.3020 0.7451 0.9333],...
    "Parent",plotFrames.UAV.Lidar);
scatterplot.XDataSource = "reshape(pt.Location(:,:,1),[],1)";
scatterplot.YDataSource = "reshape(pt.Location(:,:,2),[],1)";
scatterplot.ZDataSource = "reshape(pt.Location(:,:,3),[],1)";
scatterplot.CDataSource = "reshape(pt.Location(:,:,3),[],1) - min(reshape(pt.Location(:,:,3),[],
```

Set up the simulation. Then, iterate through the positions and show the scene each time the lidar sensor updates. Advance the scene, move the UAV platform, and update the sensors.

```
setup(scene)
for idx = 0:size(position, 3)-1
    [isupdated,lidarSampleTime, pt] = read(lidar);
    if isupdated
        % Use fast update to move platform visualization frames.
        show3D(scene,"Time",lidarSampleTime,"FastUpdate",true,"Parent",ax);
        % Refresh all plot data and visualize.
        refreshdata
```

```
        drawnow limitrate
    end
    % Advance scene simulation time and move platform.
    advance(scene);
    move(plat,[position(:,:,idx+1),zeros(1,6),eul2quat(orientation(:,:,idx+1)),zeros(1,3)])
    % Update all sensors in the scene.
    updateSensors(scene)
end
hold off
```



Version History

Introduced in R2020b

See Also

Functions
`move` | `read` | `updateMesh` | `addGeoFence` | `checkPermission`

Objects
`uavScenario` | `uavSensor`

Topics
"UAV Scenario Tutorial"

uavScenario

Generate UAV simulation scenario

Description

The `uavScenario` object generates a simulation scenario consisting of static meshes, UAV platforms, and sensors in a 3-D environment.

Creation

`scene = uavScenario` creates an empty UAV scenario with default property values. The default inertial frames are the north-east-down (NED) and the east-north-up (ENU) frames.

`scene = uavScenario(Name,Value)` configures a `uavScenario` object with properties using one or more `Name,Value` pair arguments. `Name` is a property name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name-value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`. Any unspecified properties take default values.

Using this syntax, you can specify the `UpdateRate`, `StopTime`, `HistoryBufferSize`, `ReferenceLocation`, and `MaxNumFrames` properties. You cannot specify other properties of the `uavScenario` object, which are read-only.

Properties

`UpdateRate` — Simulation update rate
`10` (default) | positive scalar

Simulation update rate, specified as a positive scalar in Hz. The step size of the scenario when using an `advance` object function is equal to the inverse of the update rate.

Example: `2`

Data Types: `double`

`StopTime` — Stop time of simulation
`Inf` (default) | nonnegative scalar

Stop time of the simulation, specified as a nonnegative scalar. A scenario stops advancing when it reaches the stop time.

Example: `60.0`

Data Types: `double`

`HistoryBufferSize` — Maximum number of steps stored in scenario
`100` (default) | positive integer greater than 1

Maximum number of steps stored in scenario, specified as a positive integer greater than 1. This property determines the maximum number of frames of platform poses stored in the scenario. If the

number of simulated steps exceeds the value of this property, then the scenario stores only latest steps.

Example: `60`

Data Types: `double`

`ReferenceLocation` — Scenario origin in geodetic coordinates
`[0 0 0]` (default) | 3-element vector of scalars

Scenario origin in geodetic coordinates, specified as a 3-element vector of scalars in the form [`latitude` `longitude` `altitude`]. `latitude` and `longitude` are geodetic coordinates in degrees. `altitude` is the height above the WGS84 reference ellipsoid in meters.

Data Types: `double`

`MaxNumFrames` — Maximum number of frames in the scenario
`10` (default) | positive integer

Maximum number of frames in the scenario, specified as a positive integer. The combined number of inertial frames, platforms, and sensors added to the scenario must be less than or equal to the value of this property.

Example: `15`

Data Types: `double`

`CurrentTime` — Current simulation time
nonnegative scalar

This property is read-only.

Current simulation time, specified as a nonnegative scalar.

Data Types: `double`

`IsRunning` — Indicate whether scenario is running
`true` | `false`

This property is read-only.

Indicate whether the scenario is running, specified as `true` or `false`. After a scenario simulation starts, it runs until it reaches the stop time.

Data Types: `logical`

`TransformTree` — Transformation information between frames
`tranformTree` object

This property is read-only.

Transformation information between all the frames in the scenario, specified as a `transformTree` object. This property contains the transformation information between the inertial, platform, and sensor frames associated with the scenario.

Data Types: `object`

`InertialFrames` — Names of inertial frames in scenario
vector of string

This property is read-only.

Names of the inertial frames in the scenario, specified as a vector of strings.

Data Types: `string`

`Platforms` — UAV platforms in scenario
array of `uavPlatform` objects

This property is read-only.

UAV platforms in the scenario, specified as an array of `uavPlatform` objects.

Object Functions

setup	Prepare UAV scenario for simulation
addCustomTerrain	Add custom terrain data
addMesh	Add new static mesh to UAV scenario
addInertialFrame	Define new inertial frame in UAV scenario
advance	Advance UAV scenario simulation by one time step
copy	Copy UAV scenario
updateSensors	Update sensor readings in UAV scenario
removeCustomTerrain	Remove custom terrain data
restart	Reset simulation of UAV scenario
show	Visualize UAV scenario in 2-D
show3D	Visualize UAV scenario in 3-D
terrainHeight	Returns terrain height in UAV scenarios

Examples

Create and Simulate UAV Scenario

Create a UAV scenario and set its local origin.

```
scene = uavScenario("UpdateRate",200,"StopTime",2,"ReferenceLocation",[46, 42, 0]);
```

Add an inertial frame called `MAP` to the scenario.

```
scene.addInertialFrame("ENU","MAP",trvec2tform([1 0 0]));
```

Add one ground mesh and two cylindrical obstacle meshes to the scenario.

```
scene.addMesh("Polygon", {[-100 0; 100 0; 100 100; -100 100],[-5 0]},[0.3 0.3 0.3]);
scene.addMesh("Cylinder", {[20 10 10],[0 30]}, [0 1 0]);
scene.addMesh("Cylinder", {[46 42 5],[0 20]}, [0 1 0], "UseLatLon", true);
```

Create a UAV platform with a specified waypoint trajectory in the scenario. Define the mesh for the UAV platform.

```
traj = waypointTrajectory("Waypoints", [0 -20 -5; 20 -20 -5; 20 0 -5],"TimeOfArrival",[0 1 2]);
uavPlat = uavPlatform("UAV",scene,"Trajectory",traj);
updateMesh(uavPlat,"quadrotor", {4}, [1 0 0],eul2tform([0 0 pi]));
addGeoFence(uavPlat,"Polygon", {[-50 0; 50 0; 50 50; -50 50],[0 100]},true,"ReferenceFrame","ENU
```

Attach an INS sensor to the front of the UAV platform.

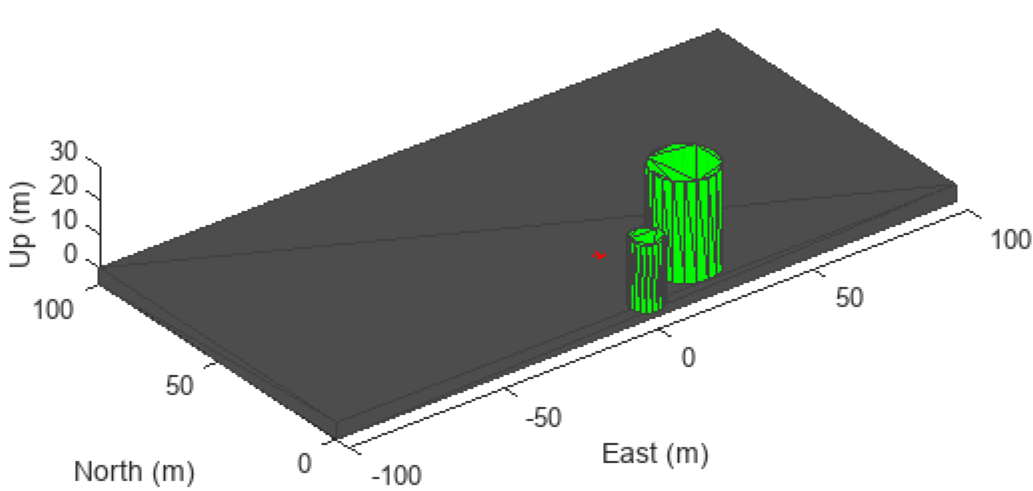
```
insModel = insSensor();
ins = uavSensor("INS",uavPlat,insModel,"MountingLocation",[4 0 0]);
```

Visualize the scenario in 3-D.

```
ax = show3D(scene);
axis(ax,"equal");
```

Simulate the scenario.

```
setup(scene);
while advance(scene)
    % Update sensor readings
    updateSensors(scene);

    % Visualize the scenario
    show3D(scene,"Parent",ax,"FastUpdate",true);
    drawnow limitrate
end
```



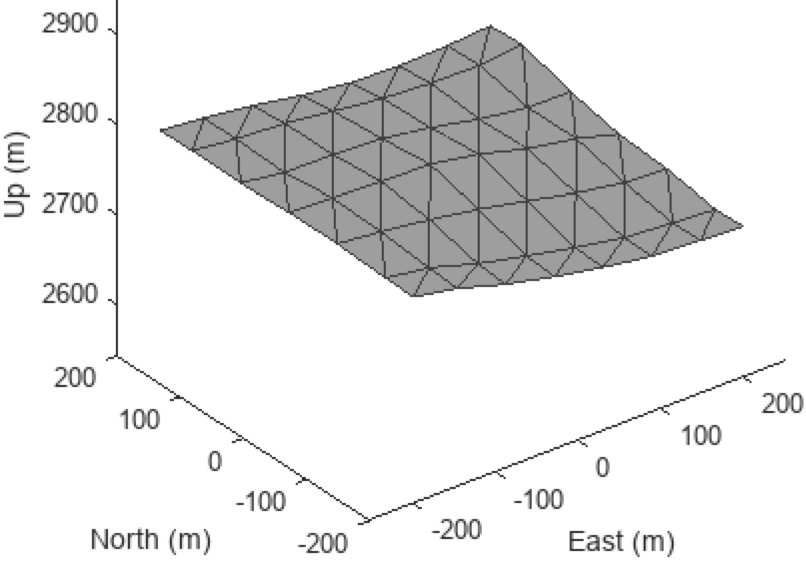
Add Terrain and Buildings to UAV Scenario

This example shows how to add terrain and custom building mesh to a UAV scenario.

Add Terrain Surface

Add terrain surface based on terrain elevation data from the `n39_w106_3arc_v2.dt1` DTED file.

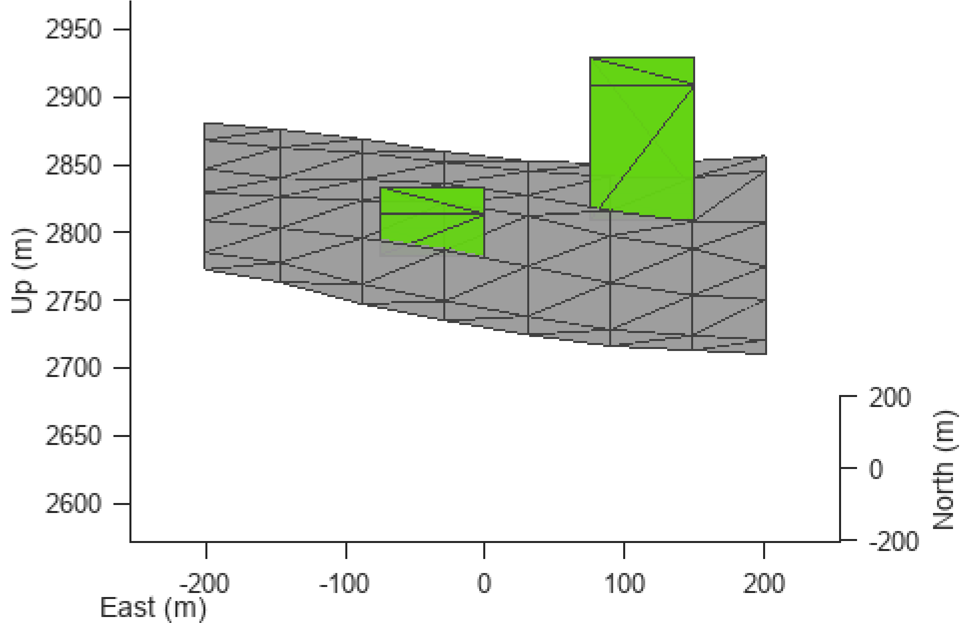
```
addCustomTerrain("CustomTerrain","n39_w106_3arc_v2.dt1");
scenario = uavScenario("ReferenceLocation", [39.5 -105.5 0]);
addMesh(scenario,"terrain", {"CustomTerrain", [-200 200], [-200 200]}, [0.6 0.6 0.6]);
show3D(scenario);
```



Add Buildings

Add a couple custom building meshes using vertices and polygon meshes into the scenario. Use the `terrainHeight` function to get ground height for each build base.

```
buildingCenters = [-50, -50; 100 100];

buildingHeights = [30 100];
buildingBoundary = [-25 -25; -25 50; 50 50; 50 -25];
for idx = 1:size(buildingCenters,1)
    buildingVertices = buildingBoundary+buildingCenters(idx,:);
    buildingBase = min(terrainHeight(scenario,buildingVertices(:,1),buildingVertices(:,2)));
    addMesh(scenario,"polygon", {buildingVertices, buildingBase+[0 buildingHeights(idx)]}, [0.392
end

show3D(scenario);
view([0 15])
```



Remove Custom Terrain

Remove the custom terrain that was imported.

```
removeCustomTerrain("CustomTerrain")
```

Version History

Introduced in R2020b

See Also

`uavMission` | `uavPlatform` | `uavSensor`

Topics

“UAV Scenario Tutorial”

uavSensor

Sensor for UAV scenario

Description

The `uavSensor` object creates a sensor that is rigidly attached to a UAV platform, specified as a `uavPlatform` object. You can specify different mounting positions and orientations. Configure this object to automatically generate readings from a sensor specified as an `insSensor`, `gpsSensor`, `uavLidarPointCloudGenerator` System object, or `uav.SensorAdaptor` class.

Creation

Syntax

```
sensor = uavSensor(name,platform,sensormodel)
sensor = uavSensor( ___ ,Name,Value)
```

Description

`sensor = uavSensor(name,platform,sensormodel)` creates a sensor with the specified name `name` and sensor model `sensormodel`, which set the `Name` and `SensorModel` properties respectively. The sensor is added to the platform `platform` specified as a `uavPlatform` object.

`sensor = uavSensor( ___ ,Name,Value)` sets properties on page 1-302 using one or more name-value pair arguments in addition to the input arguments in the previous syntax. You can specify the `MountingLocation`, `MountingAngles`, or `UpdateRate` properties as name-value pairs. For example, `uavSensor("uavLidar",plat,lidarmodel,'MountingLocation',[1 0 0])"` places the sensor one meter forward in the *x*-direction relative to the platform body frame. Enclose each property name in quotes.

Properties

Name — Sensor name
string scalar

Sensor name, specified as a string scalar. Choose a name to identify this specific sensor.

Example: `"uavLidar"`

Data Types: `string` | `char`

`MountingLocation` — Sensor position on platform
vector of the form `[x y z]`

Sensor position on platform, specified as a vector of the form `[x y z]` in the platform frame. Units are in meters.

Example: `[1 0 0]` is 1 m in the *x*-direction.

Data Types: `double`

`MountingAngles` — Sensor orientation rotation angles
vector of the form `[z y x]`

Sensor orientation rotation angles, specified as a vector of the form `[z y x]` where `z`, `y`, `x` are rotations about the *z*-axis, *y*-axis, and *x*-axis, sequentially, in degrees. The orientation is relative to the platform body frame.

Example: `[30 90 0]`

Data Types: `double`

`UpdateRate` — Update rate of sensor
positive scalar

Update rate of the sensor, specified as a positive scalar in hertz . By default, the object uses the `UpdateRate` property of the specified sensor model object.

The sensor update interval (`1/UpdateRate`) must be a multiple of the update interval for the associated `uavScenario` object.

Data Types: `double`

`SensorModel` — Sensor model for generating readings
`insSensor` System object | `gpsSensor` System object | `uavLidarPointCloudGenerator` System object

Sensor model for generating readings, specified as an `insSensor`, `gpsSensor`, or `uavLidarPointCloudGenerator` System object.

Object Functions

read   Gather latest reading from UAV sensor

Examples

UAV Scenario Tutorial

Create a scenario to simulate unmanned aerial vehicle (UAV) flights between a set of buildings. The example demonstrates updating the UAV pose in open-loop simulations. Use the UAV scenario to visualize the UAV flight and generate simulated point cloud sensor readings.

Introduction

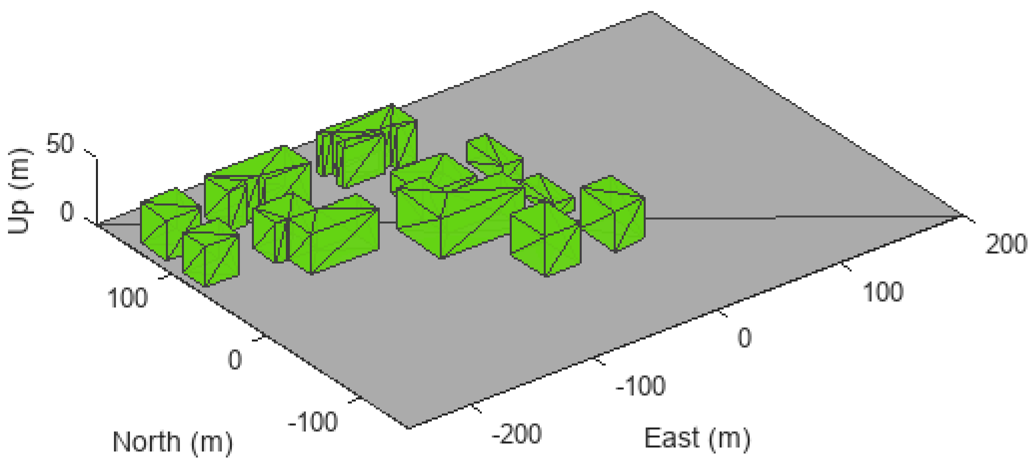
To test autonomous algorithms, a UAV scenario enables you to generate test cases and generate sensor data from the environment. You can specify obstacles in the workspace, provide trajectories of UAVs in global coordinates, and convert data between coordinate frames. The UAV scenario enables you to visualize this information in the reference frame of the environment.

Create Scenario with Polygon Building Meshes

A `uavScenario` object is a model consisting of a set of static obstacles and movable objects called platforms. Use `uavPlatform` objects to model fixed-wing UAVs, multirotors, and other objects within

the scenario. This example builds a scenario consisting of a ground plane and 11 buildings as by extruded polygons. The polygon data for the buildings is loaded and used to add polygon meshes.

```
% Create the UAV scenario.
scene = uavScenario("UpdateRate",2,"ReferenceLocation",[75 -46 0]);

% Add a ground plane.
color.Gray = 0.651*ones(1,3);
color.Green = [0.3922 0.8314 0.0745];
color.Red = [1 0 0];
addMesh(scene,"polygon",{[-250 -150; 200 -150; 200 180; -250 180],[-4 0]},color.Gray)

% Load building polygons.
load("buildingData.mat");

% Add sets of polygons as extruded meshes with varying heights from 10-30.
addMesh(scene,"polygon",{buildingData{1}(1:4,:),[0 30]},color.Green)
addMesh(scene,"polygon",{buildingData{2}(2:5,:),[0 30]},color.Green)
addMesh(scene,"polygon",{buildingData{3}(2:10,:),[0 30]},color.Green)
addMesh(scene,"polygon",{buildingData{4}(2:9,:),[0 30]},color.Green)
addMesh(scene,"polygon",{buildingData{5}(1:end-1,:),[0 30]},color.Green)
addMesh(scene,"polygon",{buildingData{6}(1:end-1,:),[0 15]},color.Green)
addMesh(scene,"polygon",{buildingData{7}(1:end-1,:),[0 30]},color.Green)
addMesh(scene,"polygon",{buildingData{8}(2:end-1,:),[0 10]},color.Green)
addMesh(scene,"polygon",{buildingData{9}(1:end-1,:),[0 15]},color.Green)
addMesh(scene,"polygon",{buildingData{10}(1:end-1,:),[0 30]},color.Green)
addMesh(scene,"polygon",{buildingData{11}(1:end-2,:),[0 30]},color.Green)

% Show the scenario.
show3D(scene);
xlim([-250 200])
ylim([-150 180])
zlim([0 50])
```



Define UAV Platform and Mount Sensor

You can define a `uavPlatform` in the scenario as a carrier of your sensor models and drive them through the scenario to collect simulated sensor data. You can associate the platform with various meshes, such as `fixedwing`, `quadrotor`, and `cuboid` meshes. You can define a custom mesh defined ones represented by vertices and faces. Specify the reference frame for describing the motion of your platform.

Load flight data into the workspace and create a quadrotor platform using an NED reference frame. Specify the initial position and orientation based on loaded flight log data. The configuration of the UAV body frame orients the *x*-axis as forward-positive, the *y*-axis as right-positive, and the *z*-axis downward-positive.

```
load("flightData.mat")

% Set up platform
plat = uavPlatform("UAV",scene,"ReferenceFrame","NED", ...
    "InitialPosition",position(:,:,1),"InitialOrientation",eul2quat(orientation(:,:,1)));

% Set up platform mesh. Add a rotation to orient the mesh to the UAV body frame.
updateMesh(plat,"quadrotor",{10},color.Red,[0 0 0],eul2quat([0 0 pi]))
```

You can choose to mount different sensors, such as the `insSensor`, `gpsSensor`, or `uavLidarPointCloudGenerator` System objects to your UAV. Mount a lidar point cloud generator and a `uavSensor` object that contains the lidar sensor model. Specify a mounting location of the sensor that is relative to the UAV body frame.

```
lidarmodel = uavLidarPointCloudGenerator("AzimuthResolution",0.3324099,...
    "ElevationLimits",[-20 20],"ElevationResolution",1.25,...
    "MaxRange",90,"UpdateRate",2,"HasOrganizedOutput",true);

lidar = uavSensor("Lidar",plat,lidarmodel,"MountingLocation",[0,0,-1]);
```

Fly the UAV Platform Along Pre-Defined Trajectory and Collect Point Cloud Sensor Readings

Move the UAV along a pre-defined trajectory, and collect the lidar sensor readings along the way. This data could be used to test lidar-based mapping and localization algorithms.

Preallocate the `traj` and `scatterPlot` line plots and then specify the plot-specific data sources. During the simulation of the `uavScenario`, use the provided `plotFrames` output from the scene as the parent axes to visualize your sensor data in the correct coordinate frames.

Visualize the scene.

```
[ax,plotFrames] = show3D(scene);
```

Update plot view for better visibility.

```
xlim([-250 200])
ylim([-150 180])
zlim([0 50])
view([-110 30])
axis equal
hold on
```

Create a line plot for the trajectory. First create the plot with `plot3`, then manually modify the data source properties of the plot. This improves performance of the plotting.

```
traj = plot3(nan,nan,nan,"Color",[1 1 1],"LineWidth",2);
traj.XDataSource = "position(:,2,1:idx+1)";
traj.YDataSource = "position(:,1,1:idx+1)";
traj.ZDataSource = "-position(:,3,1:idx+1)";
```

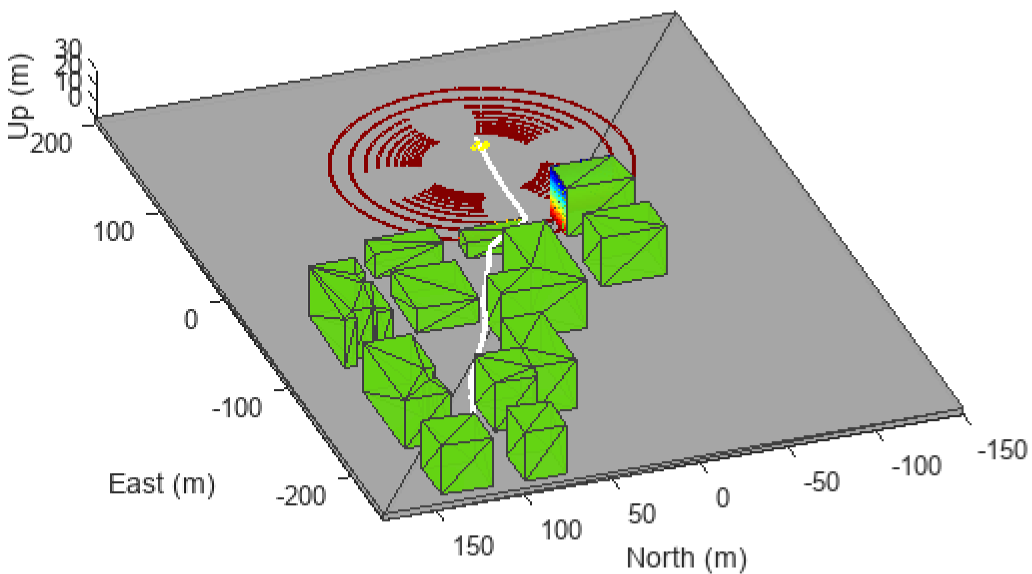
Create a scatter plot for the point cloud. Update the data source properties again.

```
colormap("jet")
pt = pointCloud(nan(1,1,3));
scatterplot = scatter3(nan,nan,nan,1,[0.3020 0.7451 0.9333],...
    "Parent",plotFrames.UAV.Lidar);
scatterplot.XDataSource = "reshape(pt.Location(:,:,1),[],1)";
scatterplot.YDataSource = "reshape(pt.Location(:,:,2),[],1)";
scatterplot.ZDataSource = "reshape(pt.Location(:,:,3),[],1)";
scatterplot.CDataSource = "reshape(pt.Location(:,:,3),[],1) - min(reshape(pt.Location(:,:,3),[],
```

Set up the simulation. Then, iterate through the positions and show the scene each time the lidar sensor updates. Advance the scene, move the UAV platform, and update the sensors.

```
setup(scene)
for idx = 0:size(position, 3)-1
    [isupdated,lidarSampleTime, pt] = read(lidar);
    if isupdated
        % Use fast update to move platform visualization frames.
        show3D(scene,"Time",lidarSampleTime,"FastUpdate",true,"Parent",ax);
        % Refresh all plot data and visualize.
        refreshdata
```

```
        drawnow limitrate
    end
    % Advance scene simulation time and move platform.
    advance(scene);
    move(plat,[position(:,:,idx+1),zeros(1,6),eul2quat(orientation(:,:,idx+1)),zeros(1,3)])
    % Update all sensors in the scene.
    updateSensors(scene)
end
hold off
```



Version History

Introduced in R2020b

See Also

Functions
read

Objects
uavScenario | uavPlatform | insSensor | gpsSensor | uavLidarPointCloudGenerator | uav.SensorAdaptor

Topics
"UAV Scenario Tutorial"

uav.SensorAdaptor class

Package: `uav`

Custom UAV sensor interface

Description

The `uav.SensorAdaptor` class is an interface for adapting custom sensor models to for use with the `uavScenario` object for UAV scenario simulation.

The `uav.SensorAdaptor` class is a `handle` class.

Class Attributes

`Abstract`	`true`

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

Creation

Syntax

`sensorObj = uav.SensorAdaptor(sensorModel)`

Description

`sensorObj = uav.SensorAdaptor(sensorModel)` creates a sensor object compatible with the `uavScenario` object. `sensorModel` is an object handle for a custom implementation of the `SensorAdaptor` class.

To get a template for a custom sensor implementation, use the `createCustomSensorTemplate` function.

Properties

`UpdateRate` — Sensor update rate
positive scalar

Sensor update rate, specified as a positive scalar in Hz.

Example: 10 Hz

Data Types: `double`

`SensorModel` — Custom sensor model implementation
object handle

Custom sensor model implementation, specified as an object handle. To get a template for a custom sensor implementation, use the `createCustomSensorTemplate` function.

Attributes:

`SetAccess`	`private`

Methods

Public Methods

setup	Set up custom sensor model
read	Read from custom sensor model
reset	Reset custom sensor model
getEmptyOutputs	Return empty sensor outputs without sensor inputs
copy	Copy sensor adaptor
copyElement	Copy sensor adaptor object

Static Methods

uav.SensorAdaptor.getMotion	Get sensor motion in platform reference frame

Version History

Introduced in R2021a

See Also

Functions

`copy` | `copyElement` | `uav.SensorAdaptor.getMotion` | `getEmptyOutputs` | `reset` | `setup` | `read`

Objects

`uavSensor` | `uavPlatform` | `uavScenario`

Topics

"Simulate Radar Sensor Mounted On UAV"

uavWaypointFollower

Follow waypoints for UAV

Description

The `uavWaypointFollower` System object follows a set of waypoints for an unmanned aerial vehicle (UAV) using a lookahead point. The object calculates the lookahead point, desired course, and desired yaw given a UAV position, a set of waypoints, and a lookahead distance. Specify a set of waypoints and tune the`lookAheadDistance` input argument and `TransitionRadius` property for navigating the waypoints. The object supports both multirotor and fixed-wing UAV types.

To follow a set of waypoints:

1. Create the `uavWaypointFollower` object and set its properties.
2. Call the object with arguments, as if it were a function.

To learn more about how System objects work, see What Are System Objects?

Creation

Syntax

```
wpFollowerObj = uavWaypointFollower
wpFollowerObj = uavWaypointFollower(Name,Value)
```

Description

`wpFollowerObj = uavWaypointFollower` creates a UAV waypoint follower with default properties.

`wpFollowerObj = uavWaypointFollower(Name,Value)` creates a UAV waypoint follower 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

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.

UAV type — Type of UAV
'fixed-wing' (default) | 'multirotor'

Type of UAV, specified as either 'fixed-wing' or 'multirotor'.

StartFrom — Waypoint start behavior
'first' (default) | 'closest'

Waypoint start behavior, specified as either 'first' or 'closest'.

When set to 'first', the UAV flies to the first path segment between waypoints listed in Waypoints. When set to 'closest', the UAV flies to the closest path segment between waypoints listed in Waypoints. When the waypoints input changes, the UAV recalculates the closest path segment.

Waypoints — Set of waypoints
n-by-3 matrix of [x y z] vectors

Set of waypoints for UAV to follow, specified as a *n*-by-3 matrix of [x y z] vectors in meters.

Data Types: single | double

YawAngles — Yaw angle for each waypoint
scalar | *n*-element column vector | []

Yaw angle for each waypoint, specified as a scalar or *n*-element column vector in radians. A scalar is applied to each waypoint in Waypoints. An input of [] keeps the yaw aligned with the desired course based on the lookahead point.

Data Types: single | double

TransitionRadius — Transition radius for each waypoint
numeric scalar | *n*-element column vector

Transition radius for each waypoint, specified as a scalar or *n*-element vector in meter. When specified as a scalar, this parameter is applied to each waypoint in Waypoints. When the UAV is within the transition radius, the object transitions to following the next path segment between waypoints.

Data Types: single | double

MinLookaheadDistance — Minimum lookahead distance
0.1 (default) | positive numeric scalar

Minimum lookahead distance, specified as a positive numeric scalar in meters.

Data Types: single | double

Usage

Syntax

```
[lookaheadPoint,desiredCourse,desiredYaw,lookaheadDistFlag,crossTrackError,
status] = wpFollowerObj(currentPose,lookaheadDistance)
```

Description

`[lookaheadPoint,desiredCourse,desiredYaw,lookaheadDistFlag,crossTrackError, status] = wpFollowerObj(currentPose,lookaheadDistance)` follows the set of waypoints specified in the waypoint follower object. The object takes the current position and lookahead distance to compute the lookahead point on the path. The desired course, yaw, and cross track error are also based on this lookahead point compared to the current position. `status` returns zero until the UAV has navigated all the waypoints.

Input Arguments

`currentPose` — Current UAV pose
`[x y z chi]` vector

Current UAV pose, specified as a `[x y z chi]` vector. This pose is used to calculate the lookahead point based on the input `lookaheadDistance`. `[x y z]` is the current position in meters. `chi` is the current course in radians.

Data Types: `single` | `double`

`lookaheadDistance` — Lookahead distance along the path
positive numeric scalar

Lookahead distance along the path, specified as a positive numeric scalar in meters.

Data Types: `single` | `double`

Output Arguments

`lookaheadPoint` — Lookahead point on path
`[x y z]` position vector

Lookahead point on path, returned as an `[x y z]` position vector in meters.

Data Types: `single` | `double`

`desiredCourse` — Desired course
numeric scalar

Desired course, returned as a numeric scalar in radians in the range of `[-pi, pi]`. The UAV course is the direction of the velocity vector. For fixed-wing type UAV, the values of desired course and desired yaw are equal.

Data Types: `single` | `double`

`desiredYaw` — Desired yaw
numeric scalar

Desired yaw, returned as a numeric scalar in radians in the range of `[-pi, pi]`. The UAV yaw is the angle of the forward direction of the UAV regardless of the velocity vector. The desired yaw is computed using linear interpolation between the yaw angle for each waypoint. For fixed-wing type UAV, the values of desired course and desired yaw are equal.

Data Types: `single` | `double`

`lookaheadDistFlag` — Lookahead distance flag
`0` (default) | `1`

Lookahead distance flag, returned as `0` or `1`. `0` indicates lookahead distance is not saturated, `1` indicates lookahead distance is saturated to minimum lookahead distance value specified.

Data Types: `uint8`

`crossTrackError` — Cross track error from UAV position to path
positive numeric scalar

Cross track error from UAV position to path, returned as a positive numeric scalar in meters. The error measures the perpendicular distance from the UAV position to the closest point on the path.

Data Types: `single` | `double`

`status` — Status of waypoint navigation
`0` | `1`

Status of waypoint navigation, returned as `0` or `1`. When the follower has navigated all waypoints, the object outputs `1`. Otherwise, the object outputs `0`.

Data Types: `uint8`

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)
```

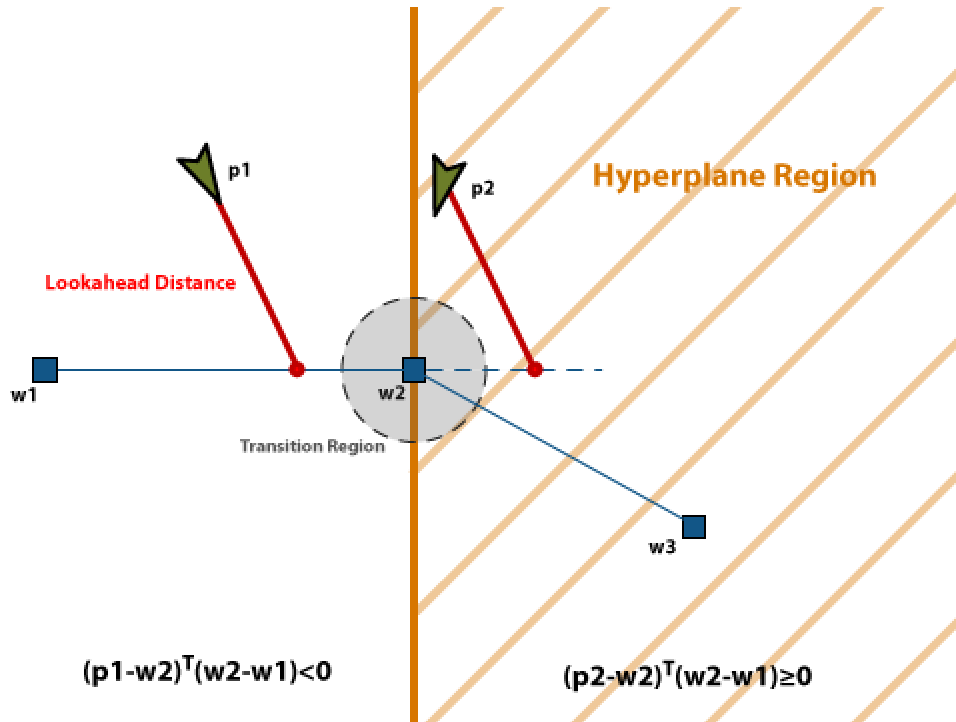
Common to All System Objects

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

More About

Waypoint Hyperplane Condition

When following a set of waypoints, the first waypoint may be ignored based on the pose of the UAV. Due to the nature of the lookahead distance used to track the path, the waypoint follower checks if the UAV is near the next waypoint to transition to the next path segment using a transition region. However, there is also a condition where the UAV transitions when outside of this region. A 3-D hyperplane is drawn at the next waypoint. If the UAV pose is inside this hyperplane, the waypoint follower transitions to the next waypoint. This behavior helps to ensure the UAV follows an achievable path.



The hyperplane condition is satisfied if:

$(p\text{-}w1)^T (w2\text{-}w1) \geq 0$

p is the UAV position, and *w1* and *w2* are sequential waypoint positions.

If you find this behavior limiting, consider adding more waypoints based on your initial pose to force the follower to navigate towards your initial waypoint.

Version History

Introduced in R2018b

References

[1] Park, Sanghyuk, John Deyst, and Jonathan How. "A New Nonlinear Guidance Logic for Trajectory Tracking." *AIAA Guidance, Navigation, and Control Conference and Exhibit*, 2004.

Extended Capabilities

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

See Also

Functions
`control` | `derivative` | `environment` | `state` | `plotTransforms`

Objects
`uavOrbitFollower` | `fixedwing` | `multirotor`

Blocks
UAV Guidance Model

Topics
“Approximate High-Fidelity UAV Model with UAV Guidance Model Block”
“Tuning Waypoint Follower for Fixed-Wing UAV”

ulogreader

Read messages from ULOG file

Description

The `ulogreader` object reads a ULOG file (`.ulg`). The object stores information about the file, including start and end logging times, summary of available topics, and dropout intervals.

Creation

Syntax

`ulogOBJ = ulogreader(filePath)`

Description

`ulogOBJ = ulogreader(filePath)` reads the ULOG file from the specified path and returns an object containing information about the file. The information in `filePath` is used to set the FileName property.

Properties

`FileName` — Name of ULOG file
string scalar | character vector

This property is read-only.

Name of the ULOG file, specified as a string scalar or character vector. The `FileName` is the path specified in the `filePath` input.

Data Types: `char` | `string`

`StartTime` — Start time of logging
`duration` object

This property is read-only.

Start time of logging offset from the system start time in the ULOG file, specified as a `duration` object in the `'hh:mm:ss.SSSSSS'` format.

Data Types: `duration`

`EndTime` — Timestamp of last timestamped message
`duration` object

This property is read-only.

Timestamp of the last timestamped message logged in the ULOG file, specified as a `duration` object in the `'hh:mm:ss.SSSSSS'` format.

Data Types: `duration`

`AvailableTopics` — Table of all logged topics
table

This property is read-only.

Summary of all the logged topics, specified as a table that contains the columns:

- `TopicNames`
- `InstanceID`
- `StartTimestamp`
- `LastTimestamp`
- `NumMessages`

Data Types: `table`

`DropoutIntervals` — Time intervals in which messages were dropped while logging
n-by-2 matrix

This property is read-only.

Time intervals in which messages were dropped while logging, specified as an *n*-by-2 matrix of `duration` arrays in the `'hh:mm:ss.SSSSSS'` format, where *n* is the number of dropouts.

Data Types: `duration`

Object Functions

readTopicMsgs	Read topic messages
readSystemInformation	Read information messages
readParameters	Read parameter values
readLoggedOutput	Read logged output messages

Examples

Read Messages from ULOG File

Load the ULOG file. Specify the relative path of the file.

```
ulog = ulogreader('flight.ulg');
```

Read all topic messages.

```
msg = readTopicMsgs(ulog);
```

Specify the time interval between which to select messages.

```
d1 = ulog.StartTime;
d2 = d1 + duration([0 0 55],'Format','hh:mm:ss.SSSSSS');
```

Read messages from the topic `'vehicle_attitude'` with an instance ID of `0` in the time interval `[d1 d2]`.

```
data = readTopicMsgs(ulog,'TopicNames',{'vehicle_attitude'}, ...
'InstanceID',{0},'Time',[d1 d2]);
```

Extract topic messages for the topic.

```
vehicle_attitude = data.TopicMessages{1,1};
```

Read all system information.

```
systeminfo = readSystemInformation(ulog);
```

Read all initial parameter values.

```
params = readParameters(ulog);
```

Read all logged output messages.

```
loggedoutput = readLoggedOutput(ulog);
```

Read logged output messages in the time interval.

```
log = readLoggedOutput(ulog,'Time',[d1 d2]);
```

Version History

Introduced in R2020b

References

[1] PX4 Developer Guide. "ULog File Format." Accessed December 6, 2019. https://dev.px4.io/v1.9.0/en/log/ulog_file_format.html.

See Also

`mavlinktlog`

waypointTrajectory

Waypoint trajectory generator

Description

The `waypointTrajectory` System object generates trajectories based on specified waypoints. When you create the System object, you can choose to specify the time of arrival, velocity, or ground speed at each waypoint. You can optionally specify other properties such as orientation at each waypoint. See "Algorithms" on page 1-353 for more details.

To generate a trajectory from waypoints:

1. Create the `waypointTrajectory` object and set its properties.
2. Call the object as if it were a function.

To learn more about how System objects work, see What Are System Objects?.

Creation

Syntax

```
trajectory = waypointTrajectory
trajectory = waypointTrajectory(Waypoints,TimeOfArrival)
trajectory = waypointTrajectory(Waypoints,GroundSpeed=groundSpeed)
trajectory = waypointTrajectory(Waypoints,Velocities=velocities)
trajectory = waypointTrajectory( ___ ,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 time of arrival at which the generated trajectory passes through each waypoint. See the `TimeOfArrival` property for more details.

Tip When you specify the `TimeOfArrival` argument, you must not specify these properties:

- `JerkLimit`
- `InitialTime`
- `WaitTime`

`trajectory = waypointTrajectory(Waypoints,GroundSpeed=groundSpeed)` specifies the ground speed at which the generated trajectory passes through at each waypoint. See the `GroundSpeed` property for more details.

`trajectory = waypointTrajectory(Waypoints,Velocities=velocities)` specifies the velocity at which the generated trajectory passes through at each waypoint. See the `Velocities` property for more details.

`trajectory = waypointTrajectory( ___ ,Name=Value)` sets each property by using name-value arguments. Unspecified properties have default or inferred values. You can use this syntax with any of the previous syntaxes.

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.

Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see System Design in MATLAB Using System Objects.

`SampleRate` — Sample rate of trajectory (Hz)
`100` (default) | positive scalar

Sample rate of trajectory in Hz, specified as a positive scalar.

Tunable: Yes

Data Types: `double`

`SamplesPerFrame` — Number of samples per output frame
`1` (default) | positive scalar integer

Number of samples per output frame, specified as a positive scalar integer.

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.

Tip To let the trajectory wait at a specific waypoint, use one of the two options:

- If you specified the `TimeOfArrival` input argument, repeat the waypoint coordinate in two consecutive rows.
- If you did not specify the `TimeOfArrival` input argument, specify the wait time using the `WaitTime` 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 not specify these properties:

- `JerkLimit`
- `InitialTime`
- `WaitTime`

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 waypoint 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 based on the velocity. If the velocity is specified as zero, the object infers the course of the trajectory from adjacent waypoints.

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.

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`.

- To render forward motion, specify positive ground speed values.
- To render backward motion, specify negative ground speed values.
- To render reverse motion, separate positive and negative groundspeed values by a zero groundspeed value.

Dependencies

To set this property, the Velocities property must not be specified.

Data Types: `double`

`ClimbRate` — Climb rate at each waypoint (m/s)

N-element real vector

Climb Rate at each waypoint in meters per second, specified as an *N*-element real vector. 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, climb rate is inferred from the waypoints.

Dependencies

To set this property, the Velocities property must not be specified.

Data Types: `double`

`JerkLimit` — Longitudinal jerk limit (m/s^3)

`Inf` (default) | positive scalar

Longitudinal jerk limit, specified as a positive scalar in m/s^3. Jerk is the time derivative of the acceleration. When you specify this property, the object produces a horizontal trapezoidal acceleration profile based on the jerk limit. If the `waypointTrajectory` object cannot achieve the specified `JerkLimit`, the object issues an error. You can set this property only during object creation.

Dependencies

To set this property, the TimeOfArrival property must not be specified.

Data Types: `double`

`InitialTime` — Time before trajectory starts (s)

`0` (default) | nonnegative scalar

Time before the trajectory starts, specified as a nonnegative scalar in seconds. The object reports quantities, such as position and velocity, as `NaN` before the trajectory starts. You can set this property only during object creation.

Dependencies

To set this property, the TimeOfArrival property must not be specified. Instead, you must specify either the `GroundSpeed` or `Velocities` property when creating the object.

Data Types: `double`

`WaitTime` — Wait time at each waypoint (s)

N-element vector of `0` (default) | *N*-element vector of nonnegative scalars

Wait time at each waypoint, specified as an *N*-element vector of nonnegative scalars. *N* must be the same as the number of samples (rows) defined by `Waypoints`. You can set this property only during object creation.

Dependencies

To set this property, the TimeOfArrival property must not be specified.

If you specified the TimeOfArrival property, then you cannot specify wait time through this property. Instead, specify wait time by repeating the waypoint coordinate in two consecutive rows in the Waypoints property.

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. Each `quaternion` must have a norm of `1`. Each 3-by-3 rotation matrix must be an orthonormal matrix. 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`.

Data Types: `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.

`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 counteracts 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.

`ReferenceFrame` — Reference frame of trajectory
`'NED'` (default) | `'ENU'`

Reference frame of the trajectory, specified as `'NED'` (North-East-Down) or `'ENU'` (East-North-Up).

Usage

Syntax

```
[position,orientation,velocity,acceleration,angularVelocity] = trajectory()
```

Description

`[position,orientation,velocity,acceleration,angularVelocity] = trajectory()` outputs a frame of trajectory data based on specified creation arguments and properties.

Output Arguments

`position` — Position in local navigation coordinate system (m)
M-by-3 matrix

Position in the local navigation coordinate system in meters, returned as an *M*-by-3 matrix.

M is specified by the SamplesPerFrame property.

Data Types: `double`

`orientation` — Orientation in local navigation coordinate system
M-element quaternion column vector | 3-by-3-by-*M* real array

Orientation in the local navigation coordinate system, returned as an *M*-by-1 `quaternion` column vector or a 3-by-3-by-*M* real array.

Each quaternion or 3-by-3 rotation matrix is a frame rotation from the local navigation coordinate system to the current body coordinate system.

M is specified by the SamplesPerFrame property.

Data Types: `double`

`velocity` — Velocity in local navigation coordinate system (m/s)
M-by-3 matrix

Velocity in the local navigation coordinate system in meters per second, returned as an *M*-by-3 matrix.

M is specified by the SamplesPerFrame property.

Data Types: `double`

`acceleration` — Acceleration in local navigation coordinate system (m/s^2)
M-by-3 matrix

Acceleration in the local navigation coordinate system in meters per second squared, returned as an *M*-by-3 matrix.

M is specified by the SamplesPerFrame property.

Data Types: `double`

`angularVelocity` — Angular velocity in local navigation coordinate system (rad/s)
M-by-3 matrix

Angular velocity in the local navigation coordinate system in radians per second, returned as an *M*-by-3 matrix.

M is specified by the SamplesPerFrame property.

Data Types: `double`

Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

Specific to `waypointTrajectory`

waypointInfo	Get waypoint information table
lookupPose	Obtain pose information for certain time
perturbations	Perturbation defined on object
perturb	Apply perturbations to object

Common to All System Objects

clone	Create duplicate System object
step	Run System object algorithm
release	Release resources and allow changes to System object property values and input characteristics
reset	Reset internal states of System object
isDone	End-of-data status

Examples

Create Default `waypointTrajectory`

```
trajectory = waypointTrajectory
```

```
trajectory =
  waypointTrajectory with properties:

         SampleRate: 100
    SamplesPerFrame: 1
          Waypoints: [2x3 double]
      TimeOfArrival: [2x1 double]
         Velocities: [2x3 double]
             Course: [2x1 double]
        GroundSpeed: [2x1 double]
          ClimbRate: [2x1 double]
        Orientation: [2x1 quaternion]
          AutoPitch: 0
           AutoBank: 0
     ReferenceFrame: 'NED'
```

Inspect the default waypoints and times of arrival by calling `waypointInfo`. By default, the waypoints indicate a stationary position for one second.

```
waypointInfo(trajectory)
```

```
ans=2×2 table
    TimeOfArrival     Waypoints
    _____________    ___________
```

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

Create Square Trajectory

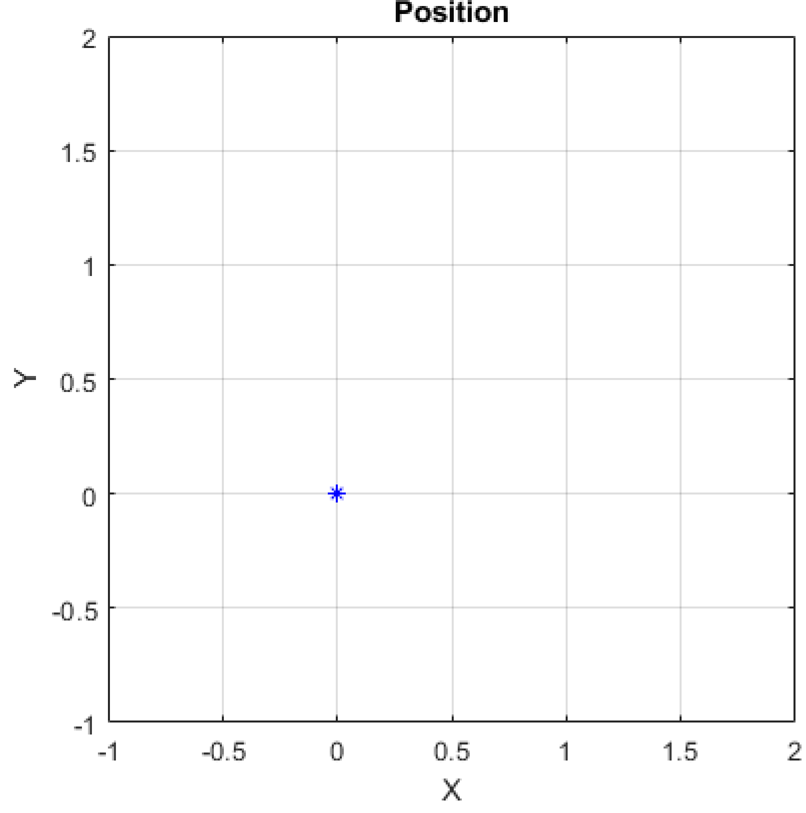
Create a square trajectory and examine the relationship between waypoint constraints, sample rate, and the generated trajectory.

Create a square trajectory by defining the vertices of the square. Define the orientation at each waypoint as pointing in the direction of motion. Specify a 1 Hz sample rate and use the default `SamplesPerFrame` of 1.

```
waypoints = [0,0,0; ... % Initial position
             0,1,0; ...
             1,1,0; ...
             1,0,0; ...
             0,0,0];    % Final position

toa = 0:4; % time of arrival

orientation = quaternion([0,0,0; ...
                          45,0,0; ...
                          135,0,0; ...
                          225,0,0; ...
                          0,0,0], ...
                          "eulerd","ZYX","frame");

trajectory = waypointTrajectory(waypoints, ...
    TimeOfArrival=toa, ...
    Orientation=orientation, ...
    SampleRate=1);
```

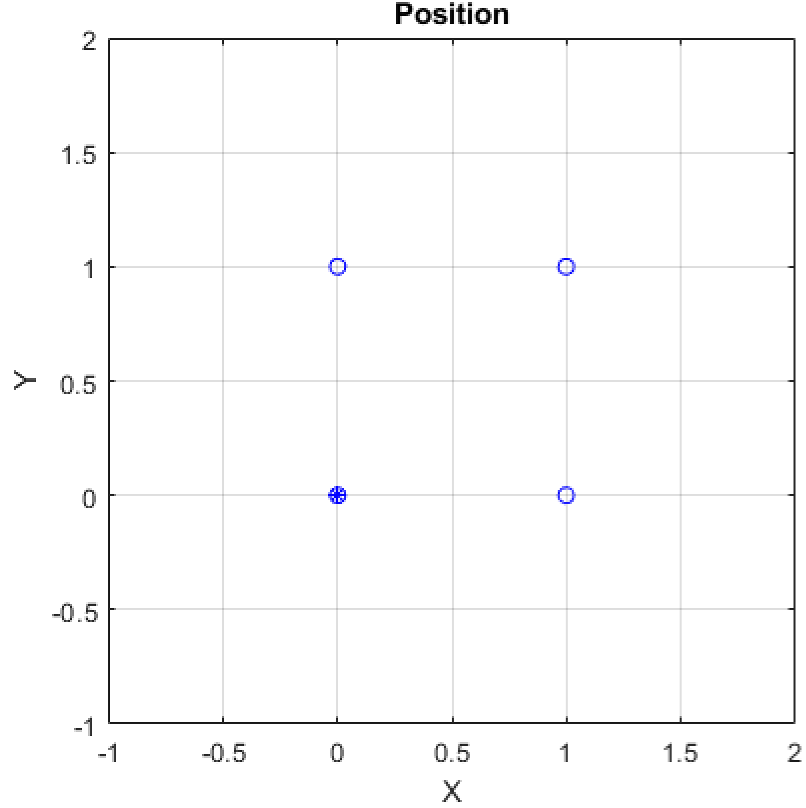
Create a figure and plot the initial position of the platform.

```
figure(1)
plot(waypoints(1,1),waypoints(1,2),"b*")
title("Position")
axis([-1,2,-1,2])
axis square
xlabel("X")
ylabel("Y")
grid on
hold on
```



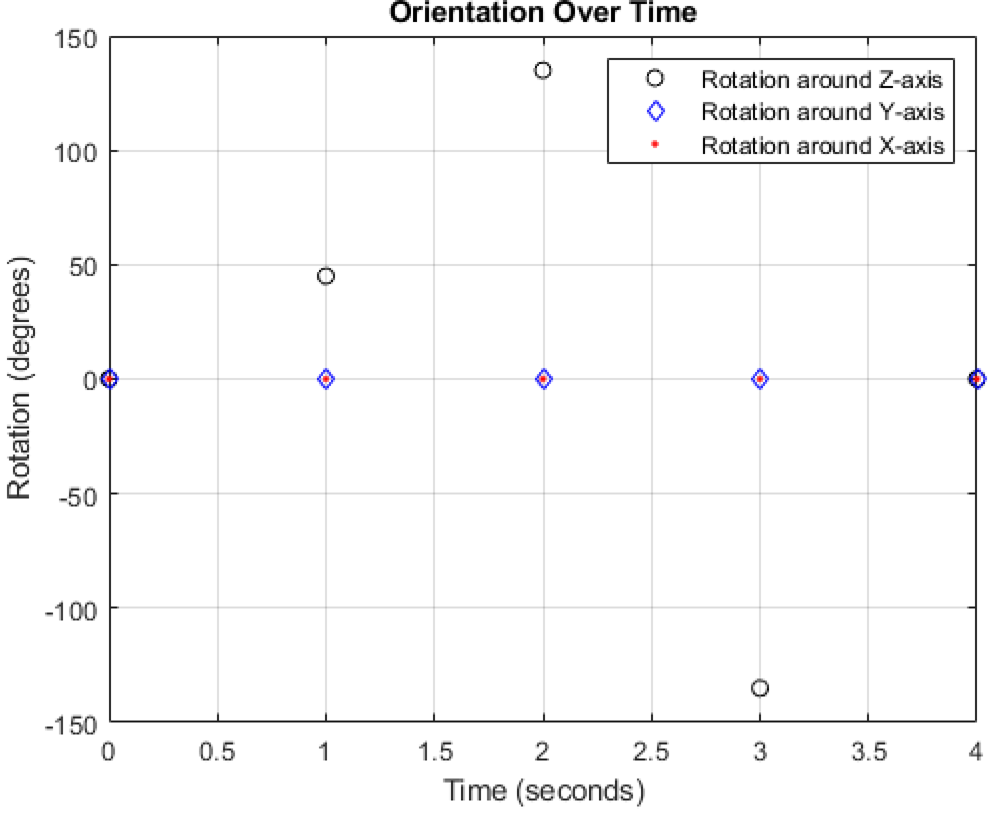
In a loop, step through the trajectory to output the current position and current orientation. Plot the current position and log the orientation. Use `pause` to mimic real-time processing.

```
orientationLog = zeros(toa(end)*trajectory.SampleRate,1,"quaternion");
count = 1;
while ~isDone(trajectory)
   [currentPosition,orientationLog(count)] = trajectory();

   plot(currentPosition(1),currentPosition(2),"bo")

   pause(trajectory.SamplesPerFrame/trajectory.SampleRate)
   count = count + 1;
end
hold off
```



Convert the orientation quaternions to Euler angles for easy interpretation, and then plot orientation over time.

```
figure(2)
eulerAngles = eulerd([orientation(1);orientationLog],"ZYX","frame");
plot(toa,eulerAngles(:,1),"ko", ...
     toa,eulerAngles(:,2),"bd", ...
     toa,eulerAngles(:,3),"r.");
title("Orientation Over Time")
legend("Rotation around Z-axis","Rotation around Y-axis","Rotation around X-axis")
xlabel("Time (seconds)")
ylabel("Rotation (degrees)")
grid on
```



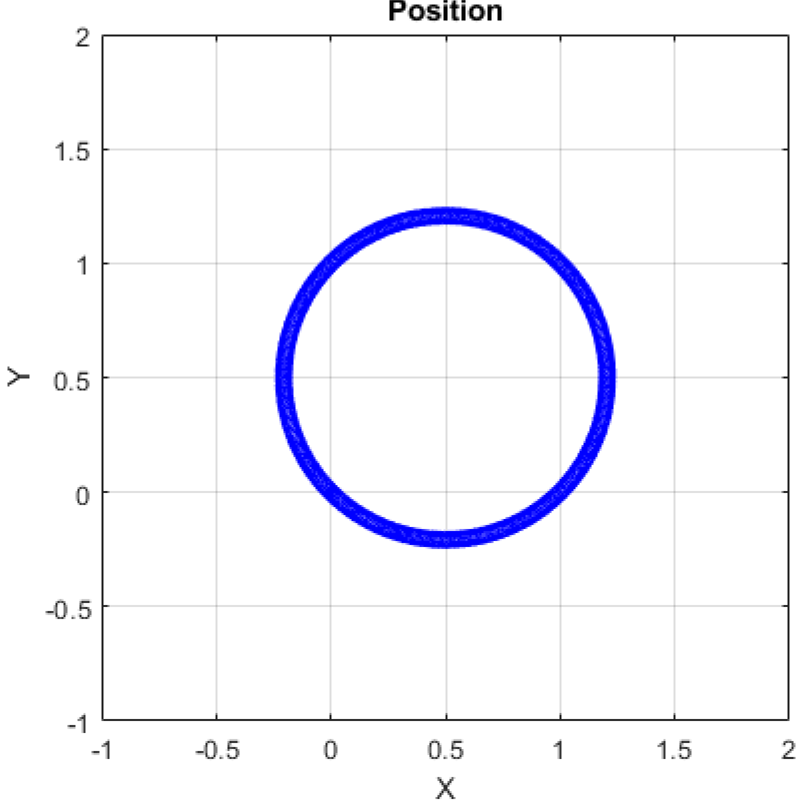
So far, the trajectory object has only output the waypoints that were specified during construction. To interpolate between waypoints, increase the sample rate to a rate faster than the time of arrivals of the waypoints. Set the `trajectory` sample rate to 100 Hz and call `reset`.

```
trajectory.SampleRate = 100;
reset(trajectory)
```

Create a figure and plot the initial position of the platform. In a loop, step through the trajectory to output the current position and current orientation. Plot the current position and log the orientation. Use `pause` to mimic real-time processing.

```
figure(1)
plot(waypoints(1,1),waypoints(1,2),"b*")
title("Position")
axis([-1,2,-1,2])
axis square
xlabel("X")
ylabel("Y")
grid on
hold on

orientationLog = zeros(toa(end)*trajectory.SampleRate,1,"quaternion");
count = 1;
while ~isDone(trajectory)
   [currentPosition,orientationLog(count)] = trajectory();

   plot(currentPosition(1),currentPosition(2),"bo")
```

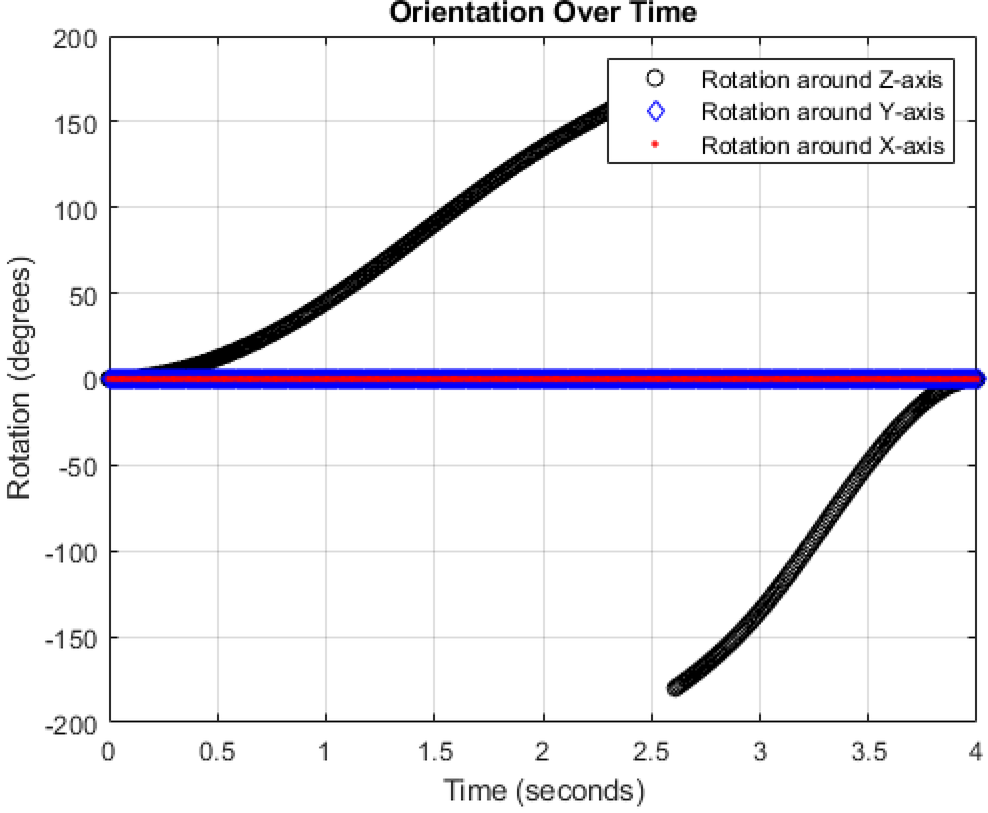
```
    pause(trajectory.SamplesPerFrame/trajectory.SampleRate)
    count = count + 1;
end
hold off
```



The trajectory output now appears circular. This is because the `waypointTrajectory` System object™ minimizes the acceleration and angular velocity when interpolating, which results in smoother, more realistic motions in most scenarios.

Convert the orientation quaternions to Euler angles for easy interpretation, and then plot orientation over time. The orientation is also interpolated.

```
figure(2)
eulerAngles = eulerd([orientation(1);orientationLog],"ZYX","frame");
t = 0:1/trajectory.SampleRate:4;
plot(t,eulerAngles(:,1),"ko", ...
     t,eulerAngles(:,2),"bd", ...
     t,eulerAngles(:,3),"r.");
title("Orientation Over Time")
legend("Rotation around Z-axis","Rotation around Y-axis","Rotation around X-axis")
xlabel("Time (seconds)")
ylabel("Rotation (degrees)")
grid on
```

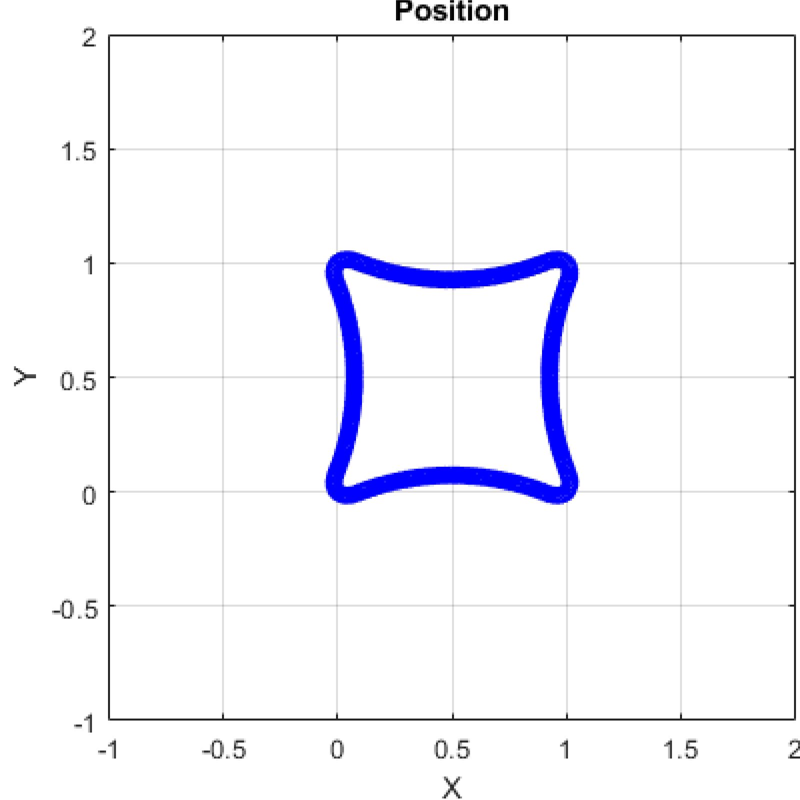


The `waypointTrajectory` algorithm interpolates the waypoints to create a smooth trajectory. To return to the square trajectory, provide more waypoints, especially around sharp changes. To track corresponding times, waypoints, and orientation, specify all the trajectory info in a single matrix.

```
               % Time, Waypoint, Orientation
trajectoryInfo = [0,   0,0,0,    0,0,0; ... % Initial position
                  0.1, 0,0.1,0,  0,0,0; ...

                  0.9, 0,0.9,0,  0,0,0; ...
                  1,   0,1,0,    45,0,0; ...
                  1.1, 0.1,1,0,  90,0,0; ...

                  1.9, 0.9,1,0,  90,0,0; ...
                  2,   1,1,0,    135,0,0; ...
                  2.1, 1,0.9,0,  180,0,0; ...

                  2.9, 1,0.1,0,  180,0,0; ...
                  3,   1,0,0,    225,0,0; ...
                  3.1, 0.9,0,0,  270,0,0; ...

                  3.9, 0.1,0,0,  270,0,0; ...
                  4,   0,0,0,    270,0,0];    % Final position

trajectory = waypointTrajectory(trajectoryInfo(:,2:4), ...
    TimeOfArrival=trajectoryInfo(:,1), ...
    Orientation=quaternion(trajectoryInfo(:,5:end),"eulerd","ZYX","frame"), ...
    SampleRate=100);
```

Create a figure and plot the initial position of the platform. In a loop, step through the trajectory to output the current position and current orientation. Plot the current position and log the orientation. Use `pause` to mimic real-time processing.

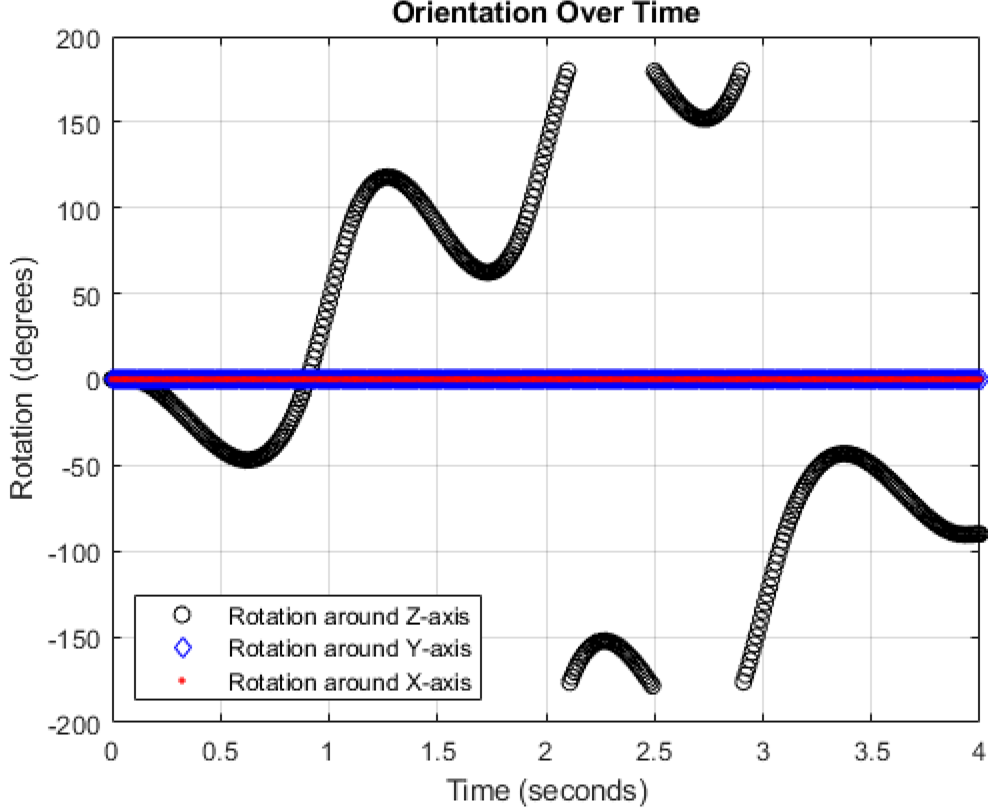
```
figure(1)
plot(waypoints(1,1),waypoints(1,2),"b*")
title("Position")
axis([-1,2,-1,2])
axis square
xlabel("X")
ylabel("Y")
grid on
hold on

orientationLog = zeros(toa(end)*trajectory.SampleRate,1,"quaternion");
count = 1;
while ~isDone(trajectory)
   [currentPosition,orientationLog(count)] = trajectory();

   plot(currentPosition(1),currentPosition(2),"bo")

   pause(trajectory.SamplesPerFrame/trajectory.SampleRate)
   count = count+1;
end
hold off
```



The trajectory output now appears more square-like, especially around the vertices with waypoints.

Convert the orientation quaternions to Euler angles for easy interpretation, and then plot orientation over time.

```
figure(2)
eulerAngles = eulerd([orientation(1);orientationLog],"ZYX","frame");
t = 0:1/trajectory.SampleRate:4;
eulerAngles = plot(t,eulerAngles(:,1),"ko", ...
                   t,eulerAngles(:,2),"bd", ...
                   t,eulerAngles(:,3),"r.");
title("Orientation Over Time")
legend("Rotation around Z-axis", ...
       "Rotation around Y-axis", ...
       "Rotation around X-axis", ...
       "Location", "SouthWest")
xlabel("Time (seconds)")
ylabel("Rotation (degrees)")
grid on
```



Create Arc Trajectory

This example shows how to create an arc trajectory using the `waypointTrajectory` System object™. `waypointTrajectory` creates a path through specified waypoints that minimizes acceleration and angular velocity. After creating an arc trajectory, you restrict the trajectory to be within preset bounds.

Create an Arc Trajectory

Define a constraints matrix consisting of waypoints, times of arrival, and orientation for an arc trajectory. The generated trajectory passes through the waypoints at the specified times with the specified orientation. The `waypointTrajectory` System object requires orientation to be specified using quaternions or rotation matrices. Convert the Euler angles saved in the 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.

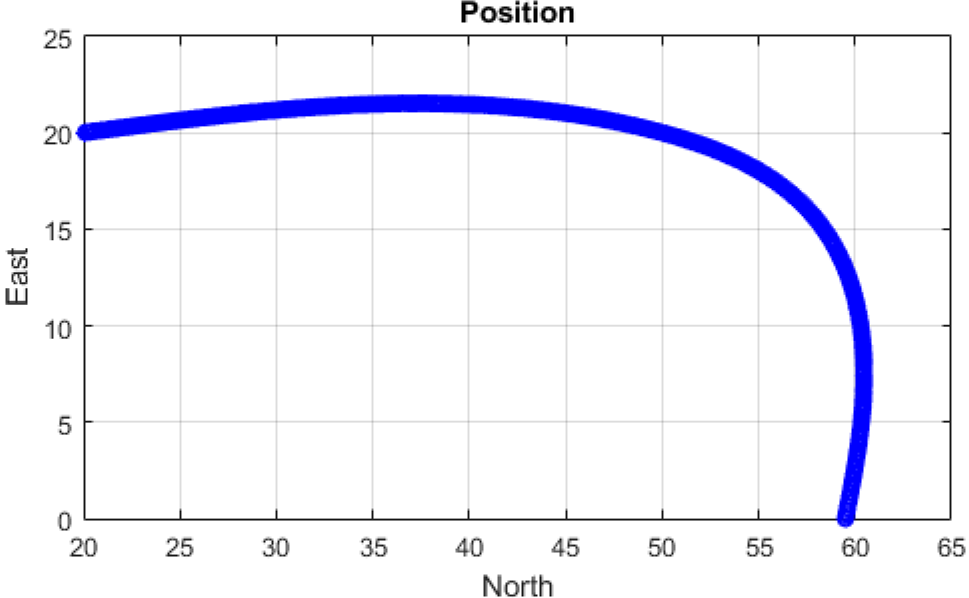
```
tInfo = waypointInfo(trajectory)
```

```
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)
```

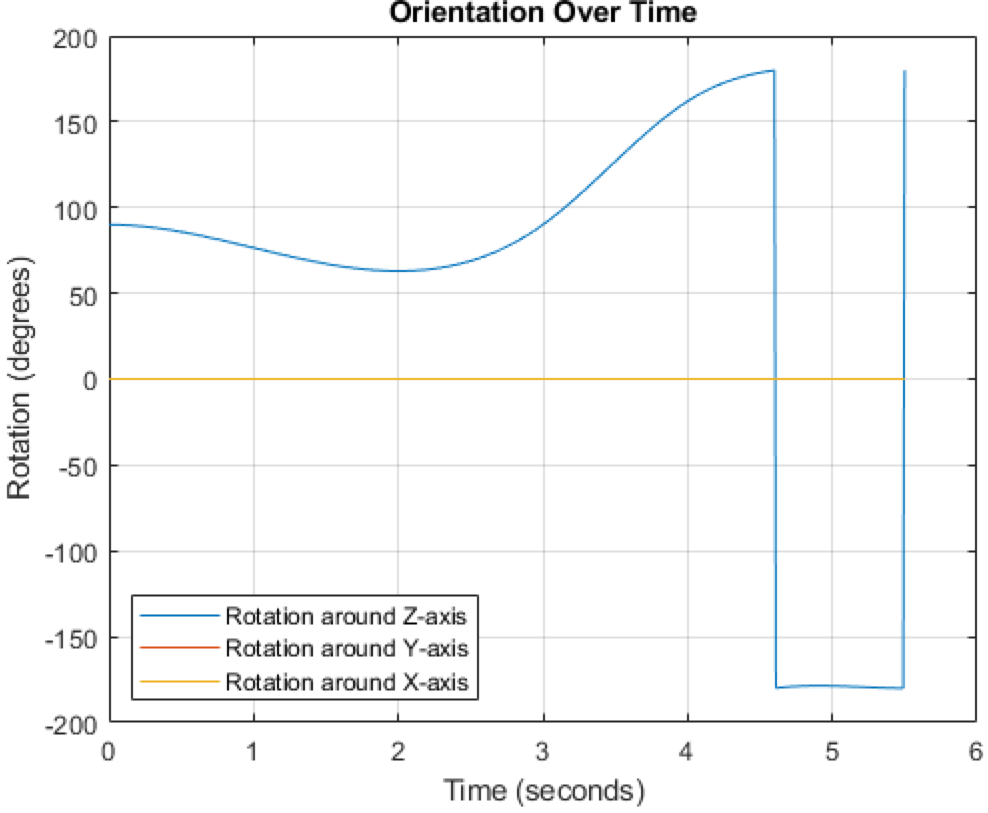
```
    [pos,orient(count),vel(count,:),acc(count,:),angVel(count,:)] = trajectory();

    plot(pos(1),pos(2),"bo")

    pause(trajectory.SamplesPerFrame/trajectory.SampleRate)
    count = count + 1;
end
```

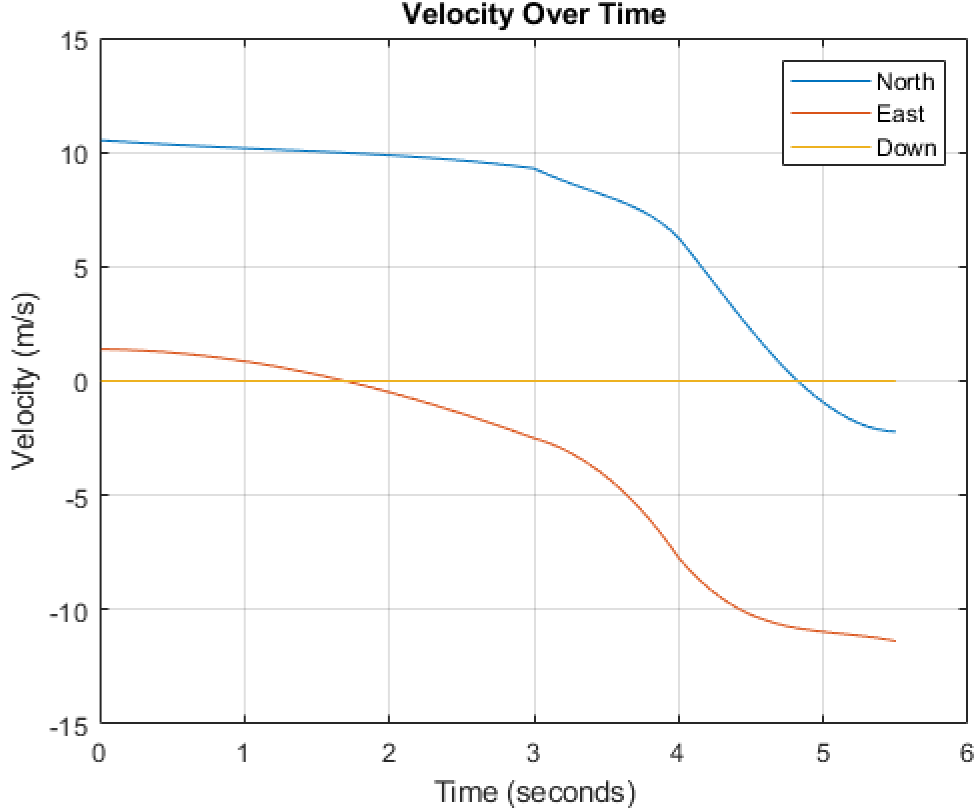


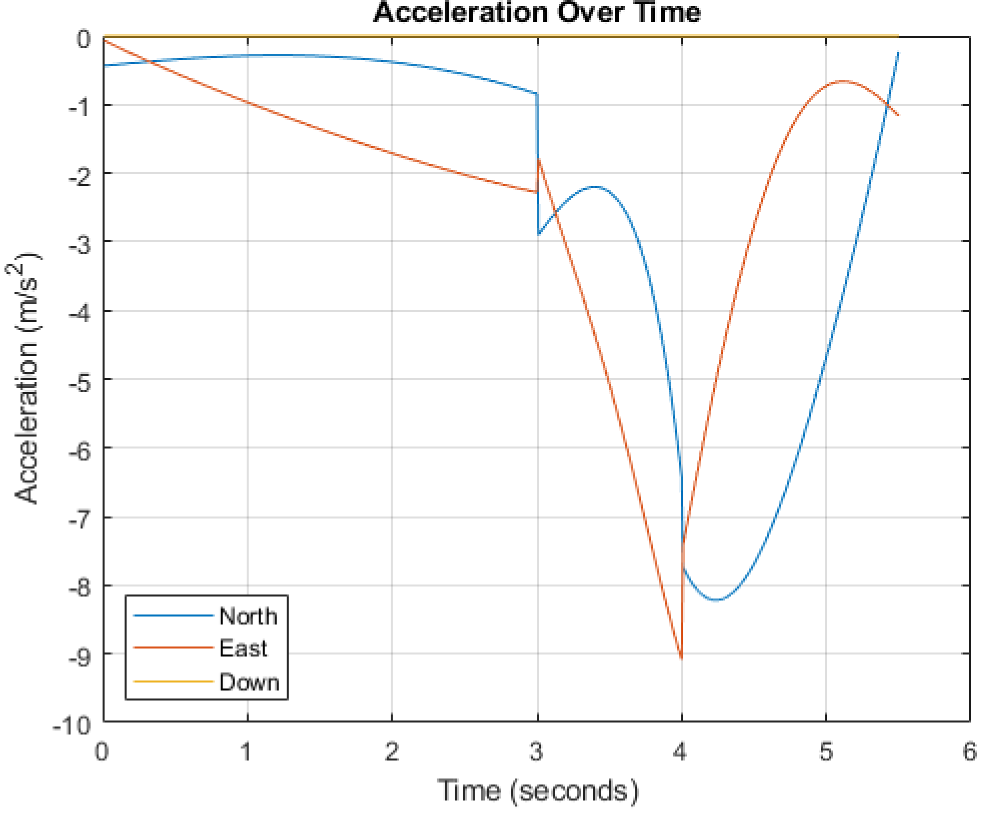
Inspect the orientation, velocity, acceleration, and angular velocity over time. The `waypointTrajectory` System object™ creates a path through the specified constraints that minimized acceleration and angular velocity.

```
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
```

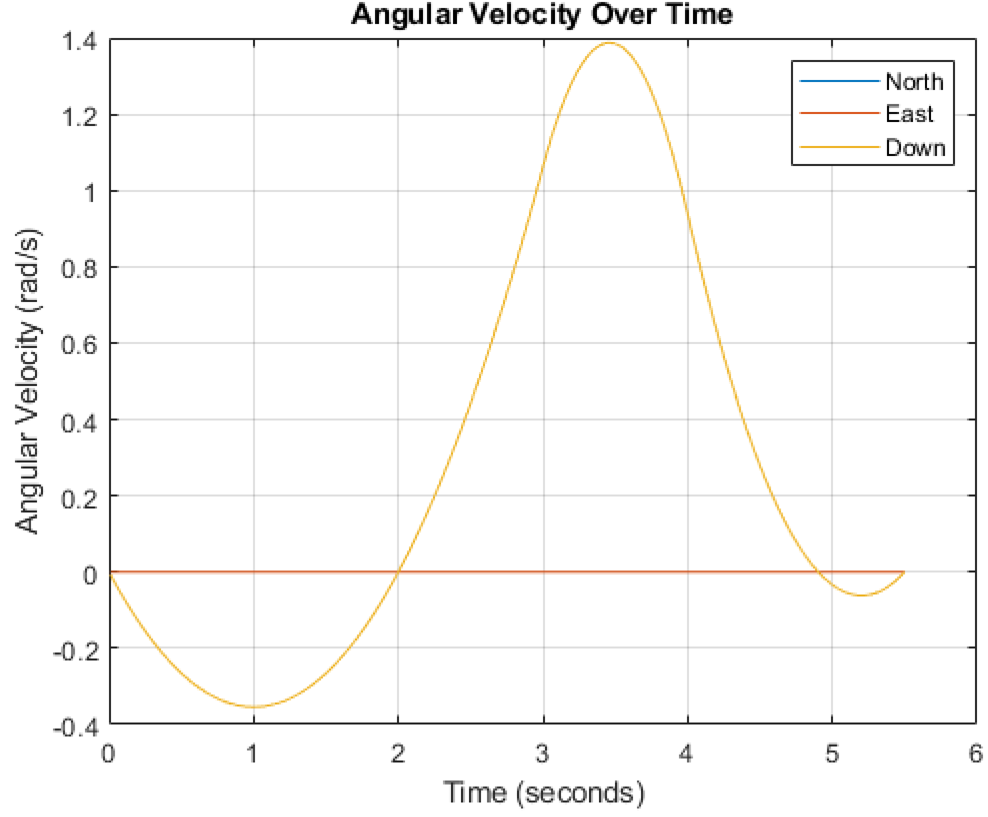
```
figure(3)
plot(timeVector(2:end),vel(:,1), ...
     timeVector(2:end),vel(:,2), ...
     timeVector(2:end),vel(:,3));
title("Velocity Over Time")
legend("North","East","Down")
xlabel("Time (seconds)")
ylabel("Velocity (m/s)")
grid on

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
```


Orientation Over Time
Rotation (degrees)
Time (seconds)
Rotation around Z-axis
Rotation around Y-axis
Rotation around X-axis
200
150
100
50
0
-50
-100
-150
-200
0
1
2
3
4
5
6


Velocity Over Time
North
East
Down
Velocity (m/s)
Time (seconds)
15
10
5
0
-5
-10
-15
0
1
2
3
4
5
6



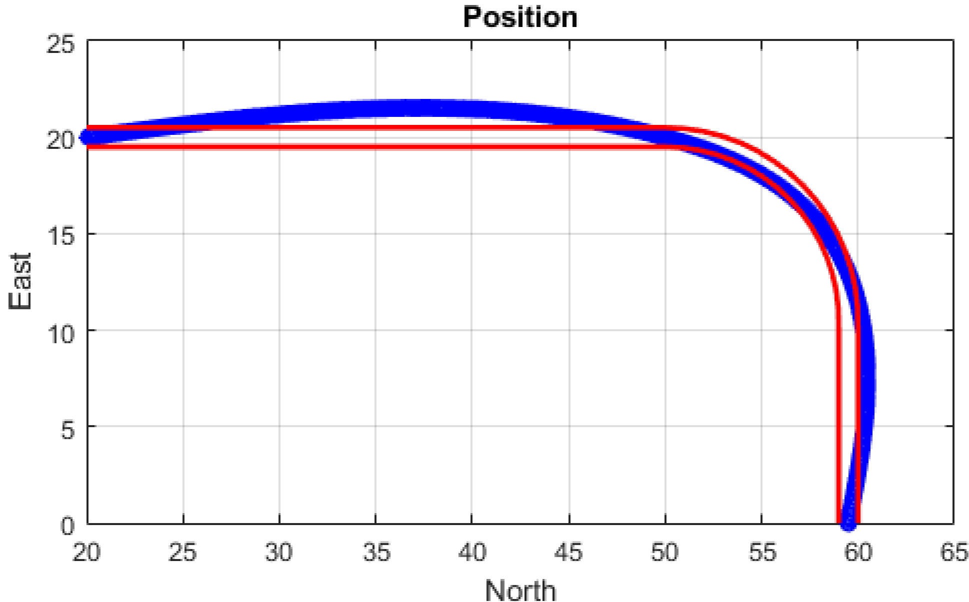
Acceleration Over Time
Acceleration (m/s^2)
Time (seconds)
North
East
Down
0
-1
-2
-3
-4
-5
-6
-7
-8
-9
-10
0
1
2
3
4
5
6



Restrict Arc Trajectory Within Preset Bounds

You can specify additional waypoints to create trajectories within given bounds. Create upper and lower bounds for the arc trajectory.

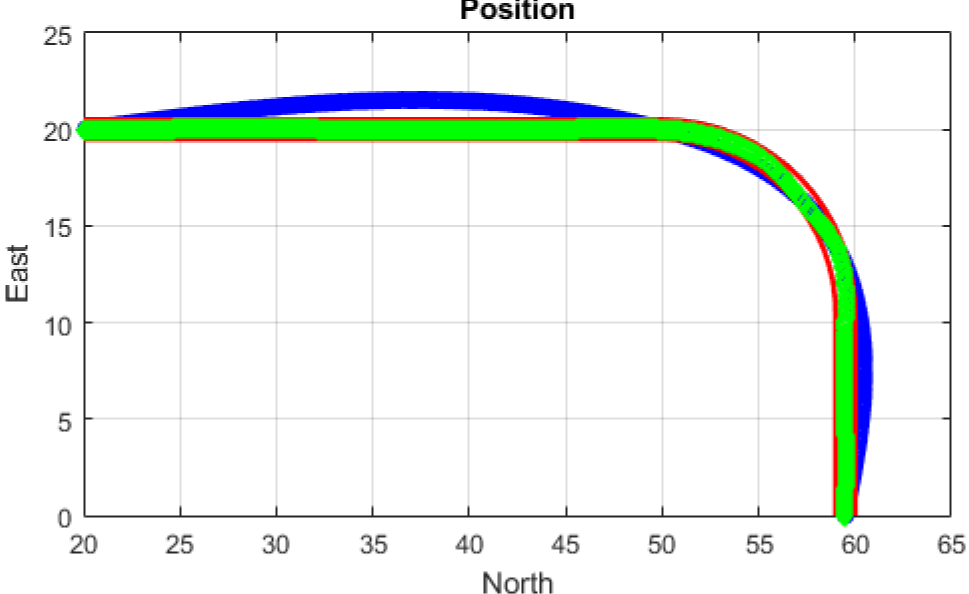
```
figure(1)
xUpperBound = [(20:50)';50+10*sin(0:0.1:pi/2)';60*ones(11,1)];
yUpperBound = [20.5.*ones(31,1);10.5+10*cos(0:0.1:pi/2)';(10:-1:0)'];

xLowerBound = [(20:49)';50+9*sin(0:0.1:pi/2)';59*ones(11,1)];
yLowerBound = [19.5.*ones(30,1);10.5+9*cos(0:0.1:pi/2)';(10:-1:0)'];

plot(xUpperBound,yUpperBound,"r","LineWidth",2);
plot(xLowerBound,yLowerBound,"r","LineWidth",2)
```



To create a trajectory within the bounds, add additional waypoints. Create a new `waypointTrajectory` System object™, and then call it in a loop to plot the generated trajectory. Cache the orientation, velocity, acceleration, and angular velocity output from the `trajectory` object.

```
              % Time,  Waypoint,     Orientation
constraints = [0,     20,20,0,      90,0,0;
               1.5,   35,20,0,      90,0,0;
               2.5    45,20,0,      90,0,0;
               3,     50,20,0,      90,0,0;
               3.3,   53,19.5,0,    108,0,0;
               3.6,   55.5,18.25,0, 126,0,0;
               3.9,   57.5,16,0,    144,0,0;
               4.2,   59,14,0,      162,0,0;
               4.5,   59.5,10,0     180,0,0;
               5,     59.5,5,0      180,0,0;
               5.5,   59.5,0,0      180,0,0];

trajectory = waypointTrajectory(constraints(:,2:4), ...
    TimeOfArrival=constraints(:,1), ...
    Orientation=quaternion(constraints(:,5:7),"eulerd","ZYX","frame"));
tInfo = waypointInfo(trajectory);

figure(1)
plot(tInfo.Waypoints(1,1),tInfo.Waypoints(1,2),"b*")

count = 1;
```

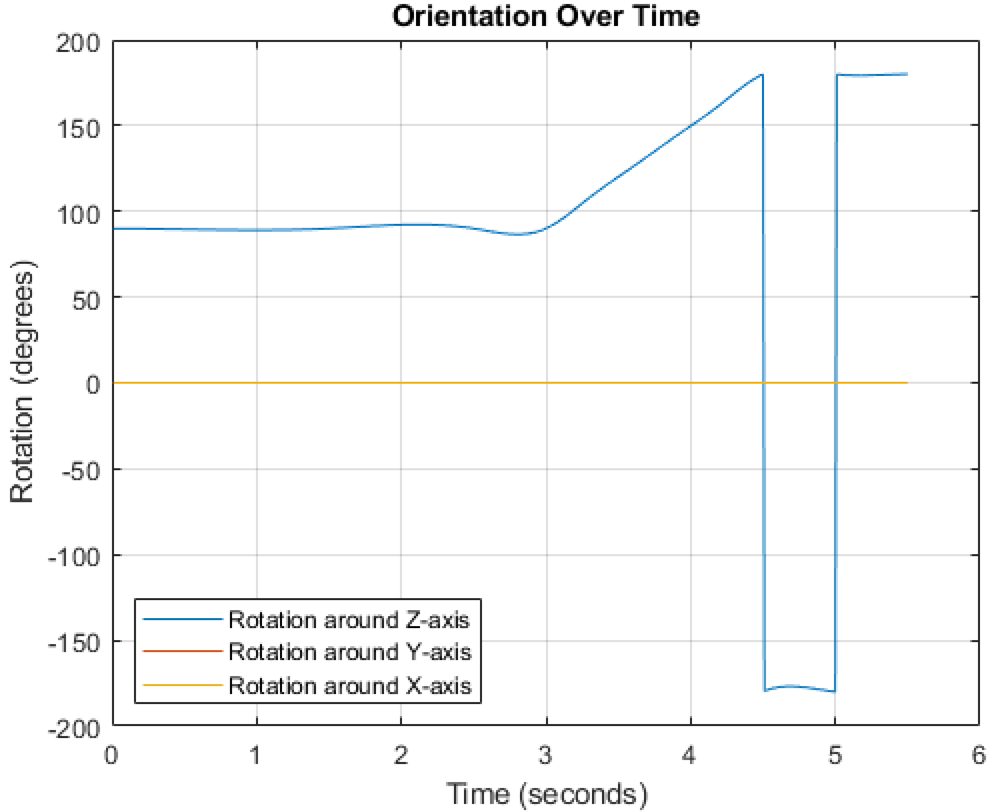
```
while ~isDone(trajectory)
   [pos,orient(count),vel(count,:),acc(count,:),angVel(count,:)] = trajectory();

   plot(pos(1),pos(2),"gd")

   pause(trajectory.SamplesPerFrame/trajectory.SampleRate)
   count = count + 1;
end
```

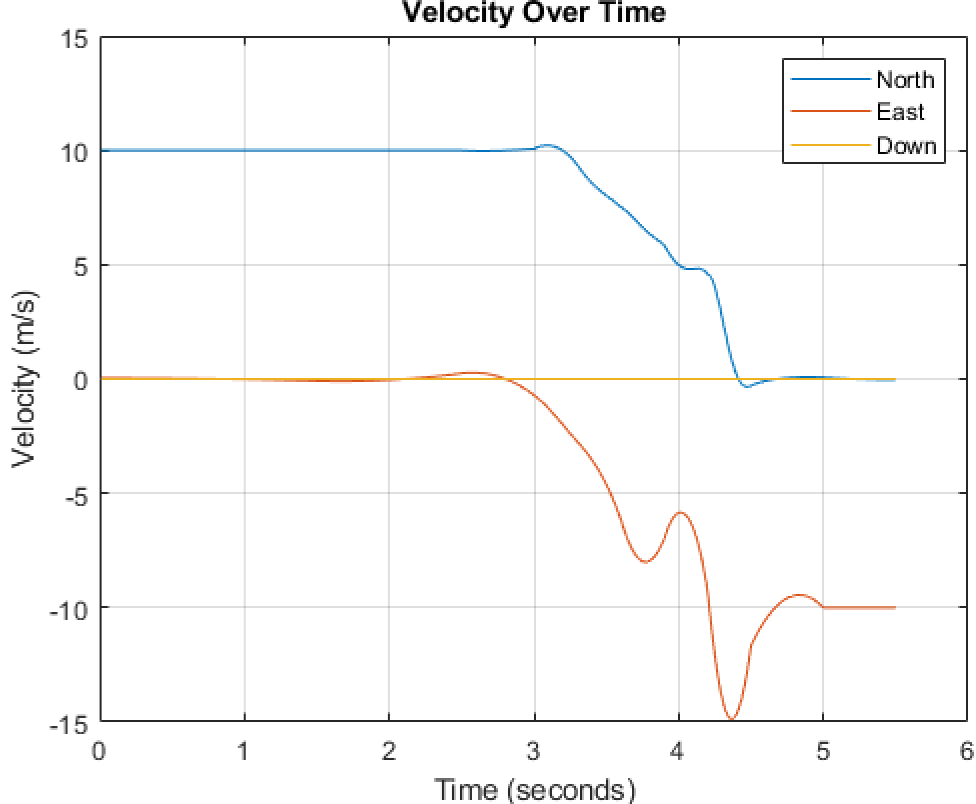


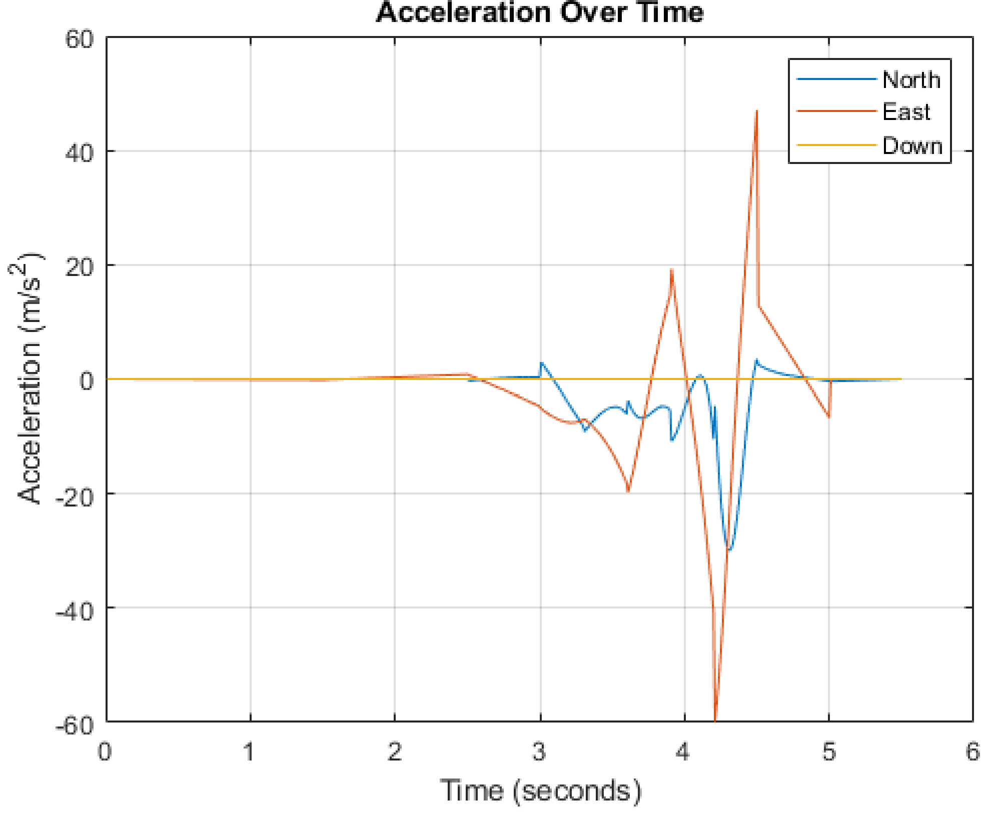
The generated trajectory now fits within the specified boundaries. Visualize the orientation, velocity, acceleration, and angular velocity of the generated trajectory.

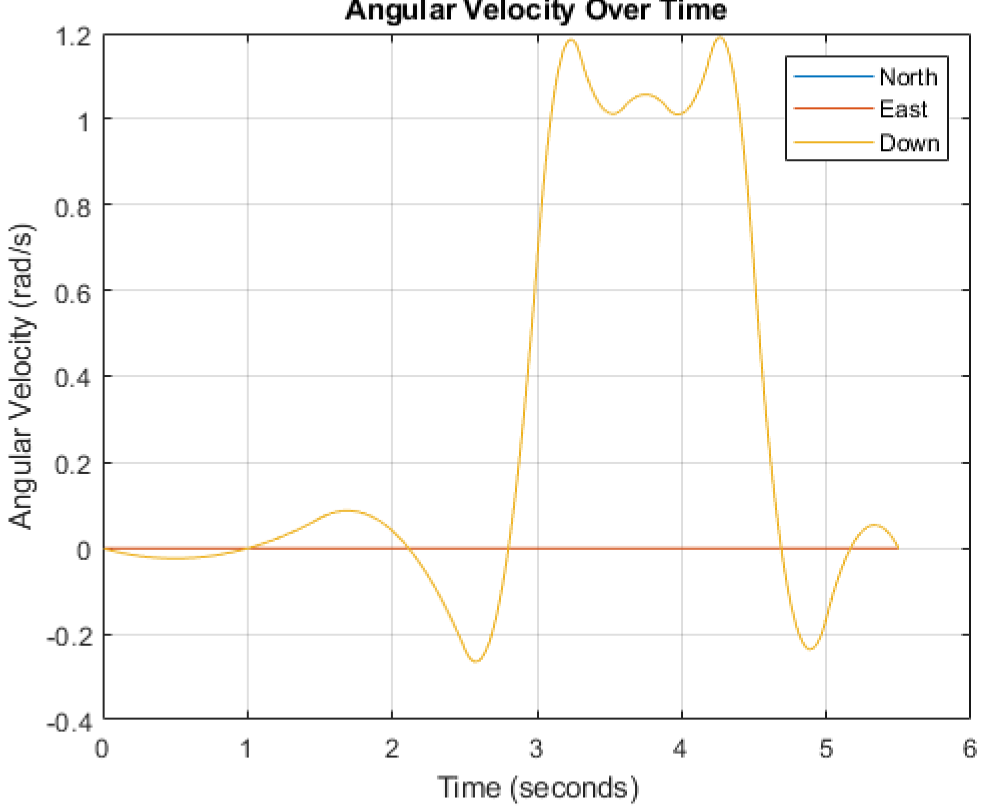
```
figure(2)
timeVector = 0:(1/trajectory.SampleRate):tInfo.TimeOfArrival(end);
eulerAngles = eulerd(orient,"ZYX","frame");
plot(timeVector(2:end),eulerAngles(:,1), ...
     timeVector(2:end),eulerAngles(:,2), ...
     timeVector(2:end),eulerAngles(:,3));
title("Orientation Over Time")
legend("Rotation around Z-axis", ...
       "Rotation around Y-axis", ...
       "Rotation around X-axis", ...
       "Location","southwest")
xlabel("Time (seconds)")
ylabel("Rotation (degrees)")
grid on

figure(3)
```

```
plot(timeVector(2:end),vel(:,1), ...
     timeVector(2:end),vel(:,2), ...
     timeVector(2:end),vel(:,3));
title("Velocity Over Time")
legend("North","East","Down")
xlabel("Time (seconds)")
ylabel("Velocity (m/s)")
grid on

figure(4)
plot(timeVector(2:end),acc(:,1), ...
     timeVector(2:end),acc(:,2), ...
     timeVector(2:end),acc(:,3));
title("Acceleration Over Time")
legend("North","East","Down")
xlabel("Time (seconds)")
ylabel("Acceleration (m/s^2)")
grid on

figure(5)
plot(timeVector(2:end),angVel(:,1), ...
     timeVector(2:end),angVel(:,2), ...
     timeVector(2:end),angVel(:,3));
title("Angular Velocity Over Time")
legend("North","East","Down")
xlabel("Time (seconds)")
ylabel("Angular Velocity (rad/s)")
grid on
```


Orientation Over Time
Rotation (degrees)
Time (seconds)
200
150
100
50
0
-50
-100
-150
-200
0
1
2
3
4
5
6
Rotation around Z-axis
Rotation around Y-axis
Rotation around X-axis


Velocity Over Time
North
East
Down
Velocity (m/s)
Time (seconds)
15
10
5
0
-5
-10
-15
0
1
2
3
4
5
6

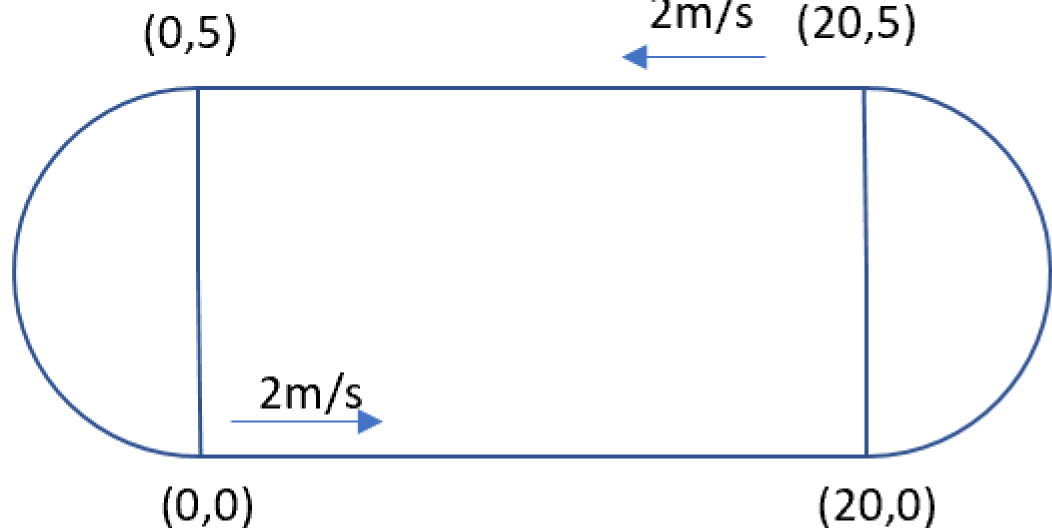

Acceleration Over Time
North
East
Down
Acceleration (m/s^2)
Time (seconds)
60
40
20
0
-20
-40
-60
0
1
2
3
4
5
6



Note that while the generated trajectory now fits within the spatial boundaries, the acceleration and angular velocity of the trajectory are somewhat erratic. This is due to over-specifying waypoints.

Generate Racetrack Trajectory Using `waypointTrajectory`

Consider a racetrack trajectory as the following.



The four corner points of the trajectory are (0,0,0), (20,0,0), (20,5,0) and (0,5,0) in meters, respectively. Therefore, specify the waypoints of a loop as:

```
wps = [0 0 0;
      20 0 0;
      20 5 0;
      0  5 0;
      0  0 0];
```

Assume the trajectory has a constant speed of 2 m/s, and thus the velocities at the five waypoints are:

```
vels = [2 0 0;
        2 0 0;
       -2 0 0;
       -2 0 0;
        2 0 0];
```

The time of arrival for the five waypoints is:

```
t = cumsum([0 20/2 5*pi/2/2 20/2 5*pi/2/2]');
```

The orientation of the trajectory at the five waypoints are:

```
eulerAngs = [0 0 0;
             0 0 0;
           180 0 0;
           180 0 0;
             0 0 0]; % Angles in degrees.
% Convert Euler angles to quaternions.
quats = quaternion(eulerAngs,"eulerd","ZYX","frame");
```

Specify the sample rate as 100 for smoothing trajectory lines.

```
fs = 100;
```

Construct the `waypointTrajectory`.

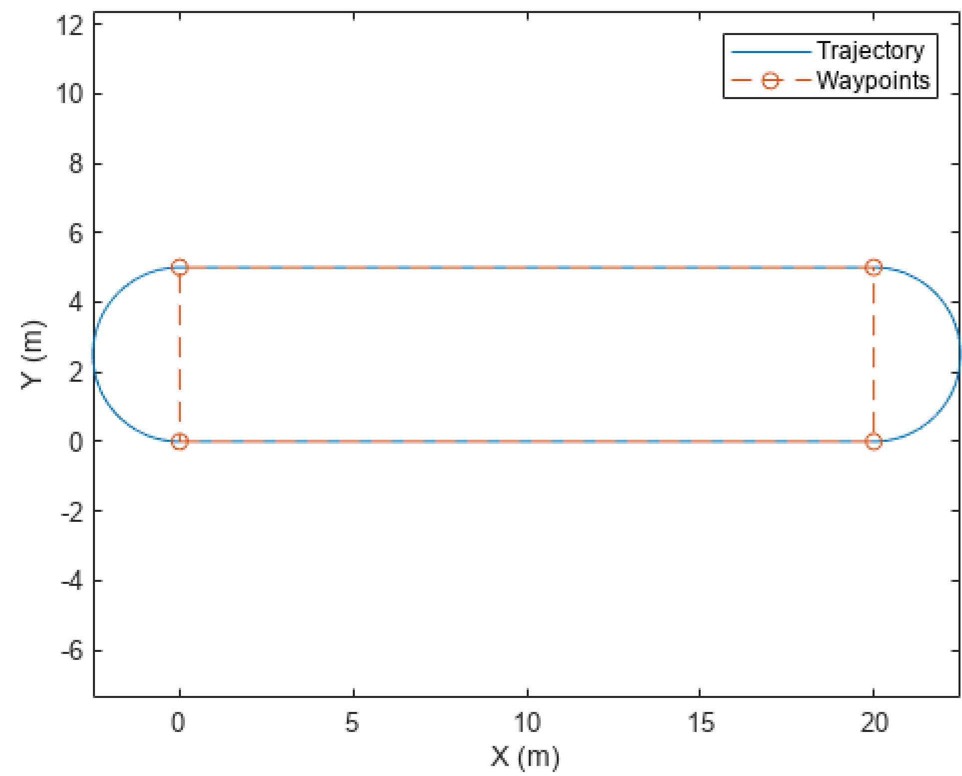
```
traj = waypointTrajectory(wps,SampleRate=fs, ...
        Velocities=vels,...
        TimeOfArrival=t,...
        Orientation=quats);
```

Sample and plot the trajectory.

```
[pos, orient, vel, acc, angvel] = traj();
i = 1;

spf = traj.SamplesPerFrame;
while ~isDone(traj)
    idx = (i+1):(i+spf);
    [pos(idx,:), orient(idx,:), ...
        vel(idx,:), acc(idx,:), angvel(idx,:)] = traj();
    i = i+spf;
end
```

Plot the trajectory and the specified waypoints.

```
plot(pos(:,1),pos(:,2), wps(:,1),wps(:,2), "--o")
xlabel("X (m)")
ylabel("Y (m)")
zlabel("Z (m)")
legend({"Trajectory", "Waypoints"})
axis equal
```



Create Trajectory Using Waypoints and Ground Speed

Create a `waypointTrajectory` object that connects two waypoints. The velocity of the trajectory at the two waypoints is 0 m/s and 10 m/s, respectively. Restrict the jerk limit to 0.5 m/s^3 to enable the trapezoidal acceleration profile.

```
waypoints = [0  0  0;
             10 50 10];
speeds = [0 10];
jerkLimit = 0.5;
trajectory = waypointTrajectory(waypoints,GroundSpeed=speeds,JerkLimit=jerkLimit);
```

Obtain the initial time and final time of the trajectory by querying the `TimeOfArrival` property. Create time stamps to sample the trajectory.

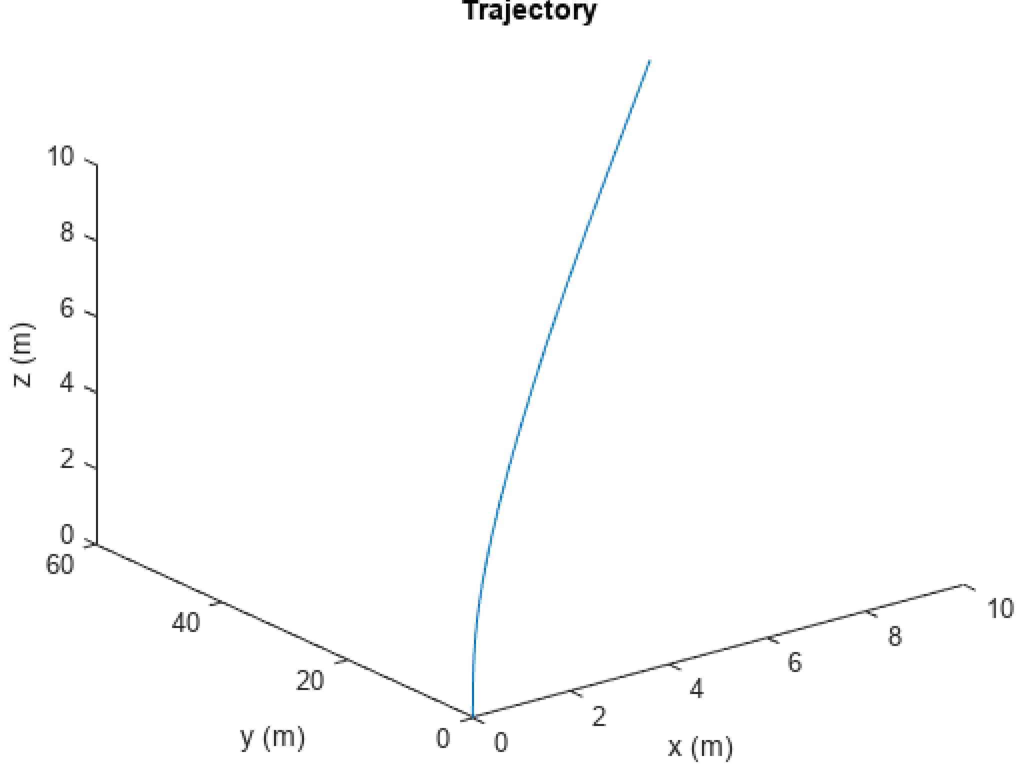
```
t0 = trajectory.TimeOfArrival(1);
tf = trajectory.TimeOfArrival(end);
sampleTimes = linspace(t0,tf,100);
```

Obtain the position, velocity, and acceleration information at these sampled time stamps using the `lookupPose` object function.

```
[position,~,velocity,acceleration,~] = lookupPose(trajectory,sampleTimes);
```

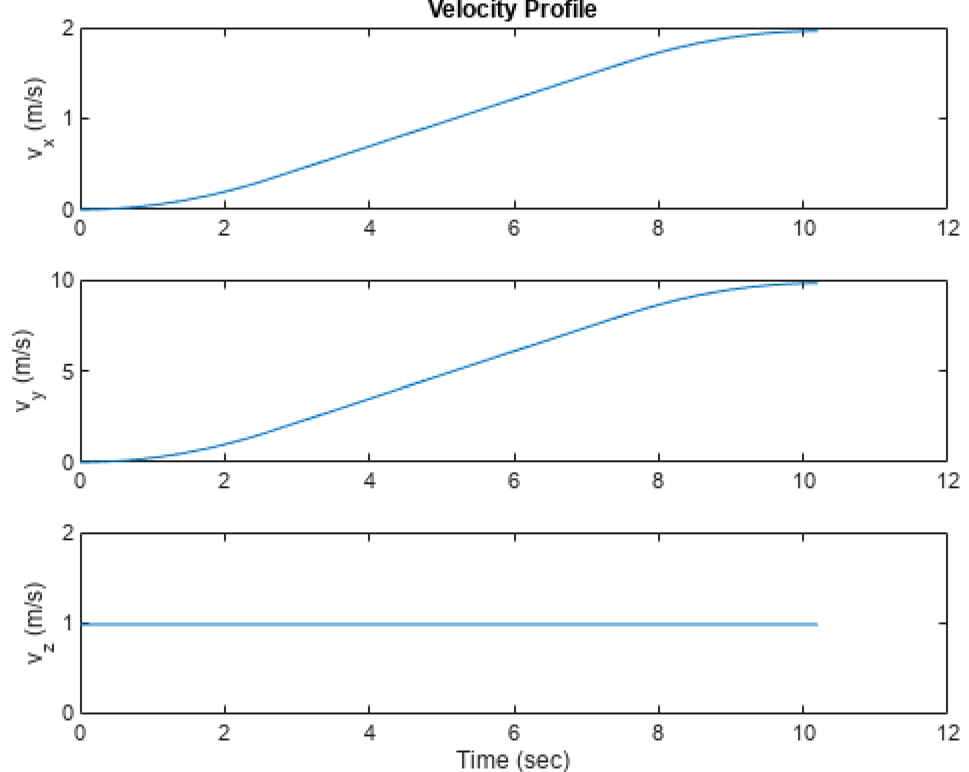
Plot the trajectory.

```
figure()
plot3(position(:,1),position(:,2),position(:,3))
xlabel("x (m)")
ylabel("y (m)")
zlabel("z (m)")
title("Trajectory")
```



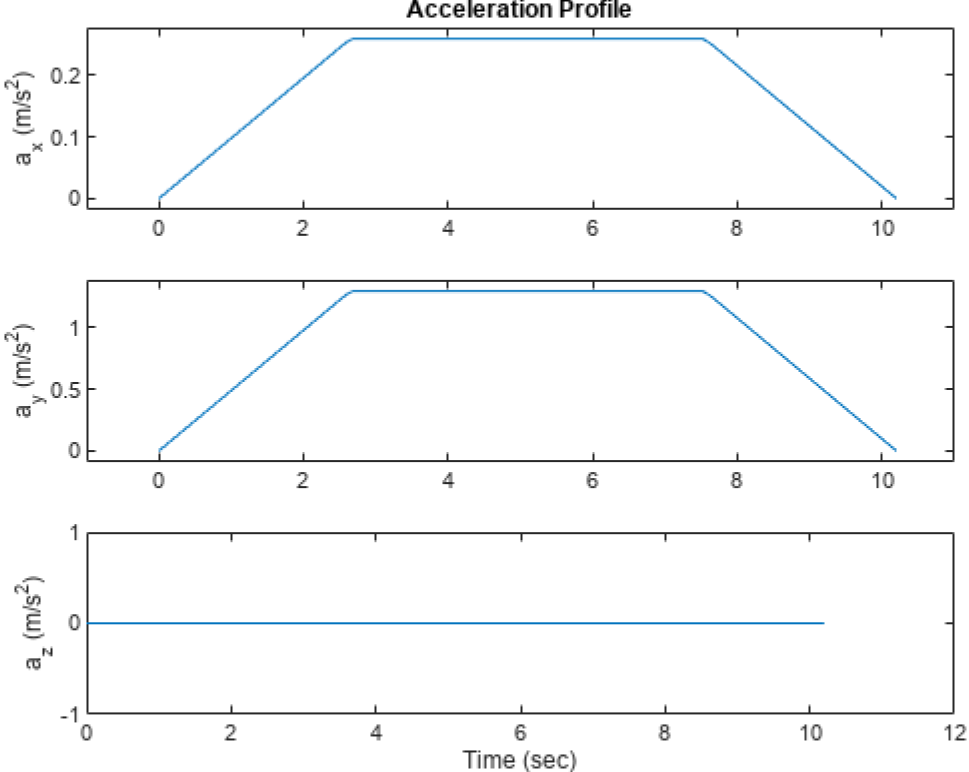
Plot the velocity profile.

```
figure()
subplot(3,1,1)
plot(sampleTimes,velocity(:,1));
ylabel("v_x (m/s)")
title("Velocity Profile")
subplot(3,1,2)
plot(sampleTimes,velocity(:,2));
ylabel("v_y (m/s)")
subplot(3,1,3)
plot(sampleTimes,velocity(:,3));
ylabel("v_z (m/s)")
xlabel("Time (sec)")
```



Plot the acceleration profile. From the results, the acceleration profile of the planar motion is trapezoidal.

```
figure()
subplot(3,1,1)
plot(sampleTimes,acceleration(:,1));
axis padded
ylabel("a_x (m/s^2)")
title("Acceleration Profile")
subplot(3,1,2)
plot(sampleTimes,acceleration(:,2));
ylabel("a_y (m/s^2)")
axis padded
subplot(3,1,3)
plot(sampleTimes,acceleration(:,3));
ylabel("a_z (m/s^2)")
xlabel("Time (sec)")
```



Algorithms

The `waypointTrajectory` System object defines a trajectory that smoothly passes through waypoints. The trajectory connects the waypoints through an interpolation that assumes the gravity direction expressed in the trajectory reference frame is constant. Generally, you can use `waypointTrajectory` to model platform or vehicle trajectories within a hundreds of kilometers distance span.

The planar path of the trajectory (the *x*-*y* plane projection) consists of piecewise, clothoid curves. The curvature of the curve between two consecutive waypoints varies linearly with the curve length between them. The tangent direction of the path at each waypoint is chosen to minimize discontinuities in the curvature, unless the course is specified explicitly via the `Course` property or implicitly via the `Velocities` property. Once the path is established, the object uses cubic Hermite interpolation to compute the location of the vehicle throughout the path as a function of time and the planar distance traveled. If the `JerkLimit` property is specified, the objects produces a horizontal trapezoidal acceleration profile for any segment that is between two waypoints. The trapezoidal acceleration profile consists of three subsegments:

- A constant-magnitude jerk subsegment
- A constant-magnitude acceleration subsegment
- A constant-magnitude jerk subsegment

The normal component (*z*-component) of the trajectory is subsequently chosen to satisfy a shape-preserving piecewise spline (PCHIP) unless the climb rate is specified explicitly via the `ClimbRate`

property or the third column of the `Velocities` property. Choose the sign of the climb rate based on the selected `ReferenceFrame`:

- When an 'ENU' reference frame is selected, specifying a positive climb rate results in an increasing value of *z*.
- When an 'NED' reference frame is selected, specifying a positive climb rate results in a decreasing value of *z*.

You can define the orientation of the vehicle through the path in two primary ways:

- If the `Orientation` property is specified, then the object uses a piecewise-cubic, quaternion spline to compute the orientation along the path as a function of time.
- If the `Orientation` property is not specified, then the yaw of the vehicle is always aligned with the path. The roll and pitch are then governed by the `AutoBank` and `AutoPitch` property values, respectively.

AutoBank	AutoPitch	Description
`false`	`false`	The vehicle is always level (zero pitch and roll). This is typically used for large marine vessels.
`false`	`true`	The vehicle pitch is aligned with the path, and its roll is always zero. This is typically used for ground vehicles.
`true`	`false`	The vehicle pitch and roll are chosen so that its local *z*-axis is aligned with the net acceleration (including gravity). This is typically used for rotary-wing craft.
`true`	`true`	The vehicle roll is chosen so that its local transverse plane aligns with the net acceleration (including gravity). The vehicle pitch is aligned with the path. This is typically used for two-wheeled vehicles and fixed-wing aircraft.

Version History

Introduced in R2020b

R2023a: Specify `waypointTrajectory` using ground speed or velocity input and new properties

When creating a waypointTrajectory object, if you specify the velocity or ground speed input, the time-of-arrival input is no longer required. When you do not specify the time-of-arrival input, you can use these new properties:

- JerkLimit — Longitudinal limit of trajectory jerk. Jerk is the derivative of the translational acceleration. If you specify a finite value for the jerk limit, waypointTrajectory produces a horizontal trapezoidal acceleration profile based on JerkLimit.
- InitialTime — Time before trajectory starts. If specified as nonzero, waypointTrajectory delays the start of the trajectory by the initial time.
- WaitTime— Wait time at each waypoint. If specified as nonzero for a waypoint, waypointTrajectory waits at the waypoint.

R2022b: Specify wait and reverse motion for waypoint trajectory

You can now specify wait and reverse motion using the waypointTrajectory System object.

- To let the trajectory wait at a specific waypoint, simply repeat the waypoint coordinate in two consecutive rows when specifying the Waypoints property.
- To render reverse motion, separate positive (forward) and negative (backward) groundspeed values by a zero value in the GroundSpeed property.

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

Objects

uavPlatform

pcplayer

Visualize streaming 3-D point cloud data

Description

Visualize 3-D point cloud data streams from devices such as Microsoft®Kinect®.

To improve performance, `pcplayer` automatically downsamples the rendered point cloud during interaction with the figure. The downsampling occurs only for rendering the point cloud and does not affect the saved points.

Creation

Syntax

```
player = pcplayer(xlimits,ylimits,zlimits)
player = pcplayer(xlimits,ylimits,zlimits,Name,Value)
```

Description

`player = pcplayer(xlimits,ylimits,zlimits)` returns a player with `xlimits`,`ylimits`, and `zlimits` set for the axes limits.

`player = pcplayer(xlimits,ylimits,zlimits,Name,Value)` returns a player with additional properties specified by one or more `Name,Value` pair arguments.

Input Arguments

`xlimits` — Range of *x*-axis coordinates
1-by-2 vector

Range of *x*-axis coordinates, specified as a 1-by-2 vector in the format [*min max*]. `pcplayer` does not display data outside these limits.

`ylimits` — Range of *y*-axis coordinates
1-by-2 vector

Range of *y*-axis coordinates, specified as a 1-by-2 vector in the format [*min max*]. `pcplayer` does not display data outside these limits.

`zlimits` — Range of *z*-axis coordinates
1-by-2 vector

Range of *z*-axis coordinates, specified as a 1-by-2 vector in the format [*min max*].`pcplayer` does not display data outside these limits.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `'VerticalAxisDir'`, `'Up'`.

`MarkerSize` — Diameter of marker
`6` (default) | positive scalar

Diameter of marker, specified as the comma-separated pair consisting of '`MarkerSize`' and a positive scalar. The value specifies the approximate diameter of the point marker. MATLAB graphics defines the unit as points. A marker size larger than six can reduce the rendering performance.

`VerticalAxis` — Vertical axis
`'Z'` (default) | `'X'` | `'Y'`

Vertical axis, specified as the comma-separated pair consisting of '`VerticalAxis`' and `'X'`, `'Y'`, or `'Z'`. When you reload a saved figure, any action on the figure resets the vertical axis to the *z*-axis.

`VerticalAxisDir` — Vertical axis direction
`'Up'` (default) | `'Down'`

Vertical axis direction, specified as the comma-separated pair consisting of '`VerticalAxisDir`' and `'Up'` or `'Down'`. When you reload a saved figure, any action on the figure resets the direction to the up direction.

Properties

`Axes` — Player axes handle
`axes` graphics object

Player axes handle, specified as an `axes` graphics object.

Usage

Color and Data Point Values in Figure

To view point data or modify color display values, hover over the axes toolbar and select one of the following options.

<table>
<tr><th>Feature</th><th>Description</th></tr>
<tr><td>Datatip</td><td>Click Data Tips to view the data point values for any point in the point cloud figure. For a normal point cloud, the Data Tips displays the x,y,z values. Additional data properties for the depth image and lidar are:
<table><tr><th>Point Cloud Data</th><th>Data Value Properties</th></tr><tr><td>Depth image (RGB-D sensor)</td><td>Color, row, column</td></tr><tr><td>Lidar</td><td>Intensity, range, azimuth angle, elevation angle, row, column</td></tr></table></td></tr>
<tr><td>Background color</td><td>Click Rotate and then right-click in the figure for background options.</td></tr>
<tr><td>Colormap value</td><td>Click Rotate and then right-click in the figure for colormap options. You can modify colormap values for the coordinate and range values available, depending on the type of point cloud displayed.</td></tr>
<tr><td>View</td><td>Click Rotate to change the viewing angle of the point cloud figure to the XZ, ZX,YZ, ZY, XY, or the YX plane. Click Restore View to reset the viewing angle.</td></tr>
</table>

OpenGL Option

`pcplayer` supports the `'opengl'` option for the `Renderer` `figure` property only.

Object Functions

hide　Hide player figure
isOpen　Visible or hidden status for player
show　Show player
view　Display point cloud

Examples

Terminate a Point Cloud Processing Loop

Create the player and add data.

```
player = pcplayer([0 1],[0 1],[0 1]);
```

Display continuous player figure. Use the `isOpen` function to check if player figure window is open.

```
while isOpen(player)
     ptCloud = pointCloud(rand(1000,3,'single'));
     view(player,ptCloud);
end
```

Terminate while-loop by closing pcplayer figure window.

Version History

Introduced in R2020b

See Also

`pointCloud`

hide

Hide player figure

Syntax

`hide(player)`

Description

`hide(player)` hides the figure. To redisplay the player, use `show(player)`.

Input Arguments

`player` — Player
object

Video player, specified as a `pcplayer` object.

Version History

Introduced in R2020b

isOpen

Visible or hidden status for player

Syntax

`isOpen(player)`

Description

`isOpen(player)` returns `true` or `false` to indicate whether the player is visible.

Input Arguments

`player` — Player
object

Video player, specified as a `pcplayer` object.

Version History
Introduced in R2020b

show

Show player

Syntax

```
show(player)
```

Description

`show(player)` makes the player figure visible again after closing or hiding it.

Input Arguments

`player` — Player
object

Player for visualizing data streams, specified as a `pcplayer` object. Use this method to view the figure after you have removed it from display. For example, after you x-out of a figure and you want to view it again. This is particularly useful to use after a while loop that contains display code ends.

Version History

Introduced in R2020b

view

Display point cloud

Syntax

```
view(player,ptCloud)
view(player,xyzPoints)
view(player,xyzPoints,color)
view(player,xyzPoints,colorMap)
```

Description

`view(player,ptCloud)` displays a point cloud in the `pcplayer` figure window, `player`. The points, locations, and colors are stored in the `ptCloud` object.

`view(player,xyzPoints)` displays the points of a point cloud at the locations specified by the `xyzPoints` matrix. The color of each point is determined by the *z* value.

`view(player,xyzPoints,color)` displays a point cloud with colors specified by `color`.

`view(player,xyzPoints,colorMap)` displays a point cloud with colors specified by `colorMap`.

Input Arguments

`ptCloud` — Point cloud
`pointCloud` object

Point cloud, specified as a `pointCloud` object. The object contains the locations, intensities, and RGB colors to render the point cloud.

Point Cloud Property	Color Rendering Result
`Location` only	Maps the z-value to a color value in the current color map.
`Location` and `Intensity`	Maps the intensity to a color value in the current color map.
`Location` and `Color`	Use provided color.
`Location`, `Intensity`, and `Color`	Use provided color.

`player` — Player
`pcplayer` object

Player for visualizing 3-D point cloud data streams, specified as a `pcplayer` object.

`xyzPoints` — Point cloud *x*, *y*, and *z* locations
M-by-3 numeric matrix | *M*-by-*N*-by-3 numeric matrix

Point cloud *x*, *y*, and *z* locations, specified as either an *M*-by-3 or an *M*-by-*N*-by-3 numeric matrix. The *M*-by-*N*-by-3 numeric matrix is commonly referred to as an organized point cloud. The `xyzPoints`

numeric matrix contains *M* or *M*-by-*N* [*x*,*y*,*z*] points. The *z* values in the numeric matrix, which generally correspond to depth or elevation, determine the color of each point.

`color` — Point cloud color

1-by-3 RGB vector | short name of color | long name of color | *M*-by-3 matrix | *M*-by-*N*-by-3 matrix

Point cloud color of points, specified as one of:

- RGB triplet
- A color name or a short name
- *M*-by-3 matrix
- *M*-by-*N*-by-3 matrix

Color Name	Short Name	RGB Triplet	Appearance
`"red"`	`"r"`	`[1 0 0]`	
`"green"`	`"g"`	`[0 1 0]`	
`"blue"`	`"b"`	`[0 0 1]`	
`"cyan"`	`"c"`	`[0 1 1]`	
`"magenta"`	`"m"`	`[1 0 1]`	
`"yellow"`	`"y"`	`[1 1 0]`	
`"black"`	`"k"`	`[0 0 0]`	
`"white"`	`"w"`	`[1 1 1]`	

You can specify the same color for all points or a different color for each point. When you set `color` to `single` or `double`, the RGB values range between [0, 1]. When you set `color` to `uint8`, the values range between [0, 255].

Points Input	Color Selection	Valid Values of `C`
`xyzPoints`	Same color for all points	1-by-3 RGB vector, or the short or long name of a color
	Different color for each point	*M*-by-3 matrix or *M*-by-*N*-by-3 matrix containing RGB values for each point.

`colorMap` — Point cloud color map

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

Point cloud color of points, specified as one of:

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

Points Input	Color Selection	Valid Values of `C`
`xyzPoints`	Different color for each point	Vector or *M*-by-*N* matrix. The matrix must contain values that are linearly mapped to a color in the current `colormap`.

Version History

Introduced in R2020b

pointCloud

Object for storing 3-D point cloud

Description

The `pointCloud` object creates point cloud data from a set of points in 3-D coordinate system. The point cloud data is stored as an object with the properties listed in “Properties” on page 1-367. Use “Object Functions” on page 1-368 to retrieve, select, and remove desired points from the point cloud data.

Creation

Syntax

```
ptCloud = pointCloud(xyzPoints)
ptCloud = pointCloud(xyzPoints,Name,Value)
```

Description

`ptCloud = pointCloud(xyzPoints)` returns a point cloud object with coordinates specified by `xyzPoints`.

`ptCloud = pointCloud(xyzPoints,Name,Value)` creates a `pointCloud` object with properties specified as one or more `Name,Value` pair arguments. For example, `pointCloud(xyzPoints,'Color',[0 0 0])` sets the `Color` property of the point `xyzPoints` as `[0 0 0]`. Enclose each property name in quotes. Any unspecified properties have default values.

Input Arguments

`xyzPoints` — 3-D coordinate points
M-by-3 list of points | *M*-by-*N*-by-3 array for organized point cloud

3-D coordinate points, specified as an *M*-by-3 list of points or an *M*-by-*N*-by-3 array for an organized point cloud. The 3-D coordinate points specify the *x*, *y*, and *z* positions of a point in the 3-D coordinate space. The first two dimensions of an organized point cloud correspond to the scanning order from sensors such as RGBD or lidar. This argument sets the `Location` property.

Data Types: `single` | `double`

Output Arguments

`ptCloud` — Point cloud
`pointCloud` object

Point cloud, returned as a `pointCloud` object with the properties listed in “Properties” on page 1-367.

Properties

`Location` — Position of the points in 3-D coordinate space

M-by-3 array | *M*-by-*N*-by-3 array

This property is read-only.

Position of the points in 3-D coordinate space, specified as an *M*-by-3 or *M*-by-*N*-by-3 array. Each entry specifies the *x*, *y*, and *z* coordinates of a point in the 3-D coordinate space. You cannot set this property as a name-value pair. Use the `xyzPoints` input argument.

Data Types: `single` | `double`

`Color` — Point cloud color

`[ ]` (default) | *M*-by-3 array | *M*-by-*N*-by-3 array

Point cloud color, specified as an *M*-by-3 or *M*-by-*N*-by-3 array. Use this property to set the color of points in point cloud. Each entry specifies the RGB color of a point in the point cloud data. Therefore, you can specify the same color for all points or a different color for each point.

- The specified RGB values must lie within the range [0, 1], when you specify the data type for `Color` as `single` or `double`.
- The specified RGB values must lie within the range [0, 255], when you specify the data type for `Color` as `uint8`.

Coordinates	**Valid assignment of** `Color`
M-by-3 array	*M*-by-3 array containing RGB values for each point
M-by-*N*-by-3 array	*M*-by-*N*-by-3 array containing RGB values for each point

Data Types: `uint8`

`Normal` — Surface normals

`[ ]` (default) | *M*-by-3 array | *M*-by-*N*-by-3 array

Surface normals, specified as a *M*-by-3 or *M*-by-*N*-by-3 array. Use this property to specify the normal vector with respect to each point in the point cloud. Each entry in the surface normals specifies the *x*, *y*, and *z* component of a normal vector.

Coordinates	Surface Normals
M-by-3 array	*M*-by-3 array, where each row contains a corresponding normal vector.
M-by-*N*-by-3 array	*M*-by-*N*-by-3 array containing a 1-by-1-by-3 normal vector for each point.

Data Types: `single` | `double`

`Intensity` — Grayscale intensities

`[ ]` (default) | *M*-by-1 vector | *M*-by-*N* matrix

Grayscale intensities at each point, specified as a *M*-by-1 vector or *M*-by-*N* matrix. The function maps each intensity value to a color value in the current colormap.

Coordinates	Intensity
M-by-3 array	*M*-by-1 vector, where each row contains a corresponding intensity value.

Coordinates	Intensity
M-by-*N*-by-3 array	*M*-by-*N* matrix containing intensity value for each point.

Data Types: `single` | `double` | `uint8`

`Count` — Number of points
positive integer

This property is read-only.

Number of points in the point cloud, stored as a positive integer.

`XLimits` — Range of *x* coordinates
1-by-2 vector

This property is read-only.

Range of coordinates along *x*-axis, stored as a 1-by-2 vector.

`YLimits` — Range of *y* coordinates
1-by-2 vector

This property is read-only.

Range of coordinates along *y*-axis, stored as a 1-by-2 vector.

`ZLimits` — Range of *z* coordinates
1-by-2 vector

This property is read-only.

Range of coordinates along *z*-axis, stored as a 1-by-2 vector.

Object Functions

findNearestNeighbors	Find nearest neighbors of a point in point cloud
findNeighborsInRadius	Find neighbors within a radius of a point in the point cloud
findPointsInROI	Find points within a region of interest in the point cloud
removeInvalidPoints	Remove invalid points from point cloud
select	Select points in point cloud
copy	Copy array of handle objects

Tips

The `pointCloud` object is a `handle` object. If you want to create a separate copy of a point cloud, you can use the MATLAB `copy` method.
`ptCloudB = copy(ptCloudA)`

If you want to preserve a single copy of a point cloud, which can be modified by point cloud functions, use the same point cloud variable name for the input and output.
`ptCloud =` *pcFunction*`(ptCloud)`

Version History

Introduced in R2020b

Extended Capabilities

C/C++ Code Generation

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

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

Usage notes and limitations:

- GPU code generation for variable input sizes is not optimized. Consider using constant size inputs for an optimized code generation.
- GPU code generation supports the `'Color'`, `'Normal'`, and `'Intensity'` name-value pairs.
- GPU code generation supports the `findNearestNeighbors`, `findNeighborsInRadius`, `findPointsInROI`, `removeInvalidPoints`, and `select` methods.
- For very large inputs, the memory requirements of the algorithm may exceed the GPU device limits. In such cases, consider reducing the input size to proceed with code generation.

See Also

Objects

`pcplayer`

Functions

`findNearestNeighbors` | `findNeighborsInRadius` | `findPointsInROI` | `removeInvalidPoints` | `select`

findNearestNeighbors

Find nearest neighbors of a point in point cloud

Syntax

```
[indices,dists] = findNearestNeighbors(ptCloud,point,K)
[indices,dists] = findNearestNeighbors( ___ ,Name,Value)
```

Description

`[indices,dists] = findNearestNeighbors(ptCloud,point,K)` returns the `indices` for the K-nearest neighbors of a query point in the input point cloud. `ptCloud` can be an unorganized or organized point cloud. The K-nearest neighbors of the query point are computed by using the Kd-tree based search algorithm. This function requires a Computer Vision Toolbox™ license.

`[indices,dists] = findNearestNeighbors( ___ ,Name,Value)` specifies options using one or more name-value arguments in addition to the input arguments in the preceding syntaxes.

Input Arguments

`ptCloud` — Point cloud
`pointCloud` object

Point cloud, specified as a `pointCloud` object.

`point` — Query point
three-element vector of form `[x y z]`

Query point, specified as a three-element vector of form `[x y z]`.

`K` — Number of nearest neighbors
positive integer

Number of nearest neighbors, specified as a positive integer.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `findNearestNeighbors(ptCloud,point,k,'Sort',true)`

`Sort` — Sort indices
`false` (default) | `true`

Sort indices, specified as a comma-separated pair of `'Sort'` and a logical scalar. When you set `Sort` to `true`, the returned indices are sorted in the ascending order based on the distance from a query point. To turn off sorting, set `Sort` to `false`.

`MaxLeafChecks` — Number of leaf nodes to check
`Inf` (default) | integer

Number of leaf nodes to check, specified as a comma-separated pair consisting of `'MaxLeafChecks'` and an integer. When you set this value to `Inf`, the entire tree is searched. When the entire tree is searched, it produces exact search results. Increasing the number of leaf nodes to check increases accuracy, but reduces efficiency.

Note The name-value argument `'MaxLeafChecks'` is valid only with Kd-tree based search method.

Output Arguments

`indices` — Indices of stored points
column vector

Indices of stored points, returned as a column vector. The vector contains K linear indices of the nearest neighbors stored in the point cloud.

`dists` — Distances to query point
column vector

Distances to query point, returned as a column vector. The vector contains the Euclidean distances between the query point and its nearest neighbors.

Version History

Introduced in R2020b

References

[1] Muja, M. and David G. Lowe. "Fast Approximate Nearest Neighbors with Automatic Algorithm Configuration". *In VISAPP International Conference on Computer Vision Theory and Applications*. 2009. pp. 331–340.

Extended Capabilities

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

Usage notes and limitations:

- For code generation in non-host platforms, the value for `'MaxLeafChecks'` must be set to the default value `Inf`. If you specify values other than `Inf`, the function generates a warning and automatically assigns the default value for `'MaxLeafChecks'`.

GPU Code Generation
Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

Usage notes and limitations:

- For GPU code generation, the `'MaxLeafChecks'` name-value pair option is ignored.

See Also

Objects
`pointCloud`

Functions
`findNeighborsInRadius` | `findPointsInROI` | `removeInvalidPoints` | `select`

findNeighborsInRadius

Find neighbors within a radius of a point in the point cloud

Syntax

```
[indices,dists] = findNeighborsInRadius(ptCloud,point,radius)
[indices,dists] = findNeighborsInRadius( ___ ,Name,Value)
```

Description

`[indices,dists] = findNeighborsInRadius(ptCloud,point,radius)` returns the `indices` of neighbors within a radius of a query point in the input point cloud. `ptCloud` can be an unorganized or organized point cloud. The neighbors within a radius of the query point are computed by using the Kd-tree based search algorithm. This function requires a Computer Vision Toolbox license.

`[indices,dists] = findNeighborsInRadius( ___ ,Name,Value)` specifies options using one or more name-value pair arguments in addition to the input arguments in the preceding syntaxes.

Input Arguments

`ptCloud` — Point cloud
`pointCloud` object

Point cloud, specified as a `pointCloud` object.

`point` — Query point
three-element vector of form [*x* *y* *z*]

Query point, specified as a three-element vector of form [*x* *y* *z*].

`radius` — Search radius
scalar

Search radius, specified as a scalar. The function finds the neighbors within the specified `radius` around a query point in the input point cloud.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `findNeighborsInRadius(ptCloud,point,radius,'Sort',true)`

`Sort` — Sort indices
`false` (default) | `true`

Sort indices, specified as a comma-separated pair of `'Sort'` and a logical scalar. When you set `Sort` to `true`, the returned indices are sorted in the ascending order based on the distance from a query point. To turn off sorting, set `Sort` to `false`.

`MaxLeafChecks` — Number of leaf nodes
`Inf` (default) | integer

Number of leaf nodes, specified as a comma-separated pair consisting of `'MaxLeafChecks'` and an integer. When you set this value to `Inf`, the entire tree is searched. When the entire tree is searched, it produces exact search results. Increasing the number of leaf nodes to check increases accuracy, but reduces efficiency.

Output Arguments

`indices` — Indices of stored points
column vector

Indices of stored points, returned as a column vector. The vector contains the linear indices of the radial neighbors stored in the point cloud.

`dists` — Distances to query point
column vector

Distances to query point, returned as a column vector. The vector contains the Euclidean distances between the query point and its radial neighbors.

Version History

Introduced in R2020b

References

[1] Muja, M. and David G. Lowe. "Fast Approximate Nearest Neighbors with Automatic Algorithm Configuration". *In VISAPP International Conference on Computer Vision Theory and Applications*. 2009. pp. 331–340.

Extended Capabilities

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

Usage notes and limitations:

- For code generation in non-host platforms, the value for `'MaxLeafChecks'` must be set to the default value `Inf`. If you specify values other than `Inf`, the function generates a warning and automatically assigns the default value for `'MaxLeafChecks'`.

GPU Code Generation
Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

Usage notes and limitations:

- For GPU code generation, the `'MaxLeafChecks'` name-value pair option is ignored.

See Also

Objects
`pointCloud`

Functions
`findNearestNeighbors` | `findPointsInROI` | `removeInvalidPoints` | `select`

findPointsInROI

Find points within a region of interest in the point cloud

Syntax

```
indices = findPointsInROI(ptCloud,roi)
```

Description

`indices = findPointsInROI(ptCloud,roi)` returns the points within a region of interest (ROI) in the input point cloud. The points within the specified ROI are obtained using a Kd-tree based search algorithm. This function requires a Computer Vision Toolbox license.

Input Arguments

`ptCloud` — Point cloud
`pointCloud` object

Point cloud, specified as a `pointCloud` object.

`roi` — Region of interest
six-element vector of form [*xmin xmax ymin ymax zmin zmax*]

Region of interest, specified as a six-element vector of form [*xmin xmax ymin ymax zmin zmax*], where:

- *xmin* and *xmax* are the minimum and the maximum limits along the *x*-axis respectively.
- *ymin* and *ymax* are the minimum and the maximum limits along the *y*-axis respectively.
- *zmin* and *zmax* are the minimum and the maximum limits along the *z*-axis respectively.

Output Arguments

`indices` — Indices of stored points
column vector

Indices of stored points, returned as a column vector. The vector contains the linear indices of the ROI points stored in the point cloud.

Version History

Introduced in R2020b

References

[1] Muja, M. and David G. Lowe. "Fast Approximate Nearest Neighbors with Automatic Algorithm Configuration". *In VISAPP International Conference on Computer Vision Theory and Applications*. 2009. pp. 331–340.

Extended Capabilities

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

GPU Code Generation
Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

-

See Also

Objects
`pointCloud`

Functions
`findNearestNeighbors` | `findNeighborsInRadius` | `removeInvalidPoints` | `select`

removeInvalidPoints

Remove invalid points from point cloud

Syntax

```
[ptCloudOut,indices] = removeInvalidPoints(ptCloud)
```

Description

`[ptCloudOut,indices] = removeInvalidPoints(ptCloud)` removes points with `Inf` or `NaN` coordinate values from point cloud and returns the indices of valid points. This function requires a Computer Vision Toolbox license.

Input Arguments

`ptCloud` — Point cloud
`pointCloud` object

Point cloud, specified as a `pointCloud` object.

Output Arguments

`ptCloudOut` — Point cloud with points removed
`pointCloud` object

Point cloud, returned as a `pointCloud` object with `Inf` or `NaN` coordinates removed.

Note The output is always an unorganized (*X*-by-3) point cloud. If the input `ptCloud` is an organized point cloud (*M*-by-*N*-by-3), the function returns the output as an unorganized point cloud.

`indices` — Indices of valid points
vector

Indices of valid points in the point cloud, specified as a vector.

Version History

Introduced in R2020b

Extended Capabilities

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

GPU Code Generation
Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

See Also

Objects
`pointCloud`

Functions
`findNearestNeighbors` | `findNeighborsInRadius` | `findPointsInROI` | `select`

select

Select points in point cloud

Syntax

```
ptCloudOut = select(ptCloud,indices)
ptCloudOut = select(ptCloud,row,column)
ptCloudOut = select( ___ ,'OutputSize',outputSize)
```

Description

`ptCloudOut = select(ptCloud,indices)` returns a `pointCloud` object containing only the points that are selected using linear indices. This function requires a Computer Vision Toolbox license.

`ptCloudOut = select(ptCloud,row,column)` returns a `pointCloud` object containing only the points that are selected using row and column subscripts. This syntax applies only if the input is an organized point cloud data of size *M*-by-*N*-by-3.

`ptCloudOut = select( ___ ,'OutputSize',outputSize)` returns the selected points as a `pointCloud` object of size specified by `outputSize`.

Input Arguments

`ptCloud` — Point cloud
`pointCloud` object

Point cloud, specified as a `pointCloud` object.

`indices` — Indices of selected points
vector

Indices of selected points, specified as a vector.

`row` — Row indices
vector

Row indices, specified as a vector. This argument applies only if the input is an organized point cloud data of size *M*-by-*N*-by-3.

`column` — Column indices
vector

Column indices, specified as a vector. This argument applies only if the input is an organized point cloud data of size *M*-by-*N*-by-3.

`outputSize` — Size of output point cloud
`'selected'` (default) | `'full'`

Size of the output point cloud, `ptCloudOut`, specified as `'selected'` or `'full'`.

- If the size is 'selected', then the output contains only the selected points from the input point cloud, ptCloud.
- If the size is 'full', then the output is same size as the input point cloud ptCloud. Cleared points are filled with NaN and the color is set to [0 0 0].

Output Arguments

ptCloudOut — Selected point cloud
pointCloud object

Point cloud, returned as a pointCloud object.

Version History

Introduced in R2020b

Extended Capabilities

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

GPU Code Generation
Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

See Also

Objects
pointCloud

Functions
findNearestNeighbors | findNeighborsInRadius | findPointsInROI | removeInvalidPoints

sim3d.maps

Access additional scenes from the server

Description

Use the `sim3d.maps` to download and access additional scenes from the server so that they can be automatically available in the Simulation 3D Scene Configuration block.

Object Functions

sim3d.maps.Map.download	Download maps from the server
sim3d.maps.Map.server	List of maps available for download from the server
sim3d.maps.Map.delete	Delete local maps downloaded from the server
sim3d.maps.Map.local	List of locally available maps

Troubleshooting

- If you cannot reach the server, the download will fail due to a timeout.
- If the download fails while updating an existing map, the existing outdated file will remain functional.
- If you delete the CSV file, you will lose automatic tracking of updates for the existing maps.

Version History

Introduced in R2022b

See Also

Simulation 3D Scene Configuration

sim3d.maps.Map.delete

Delete local maps downloaded from the server

Syntax

```
sim3d.maps.Map.delete(Scene)
```

Description

`sim3d.maps.Map.delete(Scene)` deletes the map `Scene` from your local system.

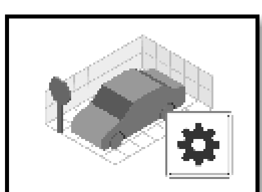
Examples

Download Suburban Scene Map

This example shows how to download and access the Suburban scene map from the Simulation 3D Scene Configuration block.

To begin, check the maps available in the server.

Add the Simulation 3D Scene Configuration block to your model.



Open the block mask and select the suburban scene from **Scene name**.

Run the model.

Delete the model and check if the map is till available locally.

```
sim3d.maps.Map.delete('Suburban scene')
```

```
Suburban scene was successfully deleted
```

Input Arguments

Scene — Name of scene
string | character array

Name of the map being deleted, specified as a string or character array. Once the map is deleted, it automatically disappears from the Simulation 3D Scene Configuration block mask menu.

Version History

Introduced in R2022b

See Also

`sim3d.maps` | `sim3d.maps.Map.download` | `sim3d.maps.Map.server` | `sim3d.maps.Map.local`

sim3d.maps.Map.download

Download maps from the server

Syntax

```
sim3d.maps.Map.download(Scene)
```

Description

`sim3d.maps.Map.download(Scene)` downloads the map `Scene` from the server.

Examples

Download Suburban Scene Map

This example shows how to download and access the Suburban scene map from the Simulation 3D Scene Configuration block.

To begin, check the maps available in the server.

```
sim3d.maps.Map.server
```

```
        MapName                          Description                      Version    MinimumRelease
    _________________    ____________________________________________    _______    ______________

    "Suburban scene"    "a suburban area beyond the city's border"       "1"           "R2022b"
```

Download the Suburban scene from the server.

```
sim3d.maps.Map.download('Suburban scene')
```

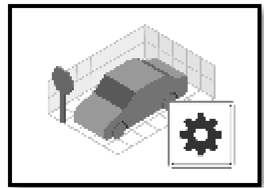
```
Map is susccesfully downloaded and is up-to-date
```

Check if the downloaded maps are available in your local machine.

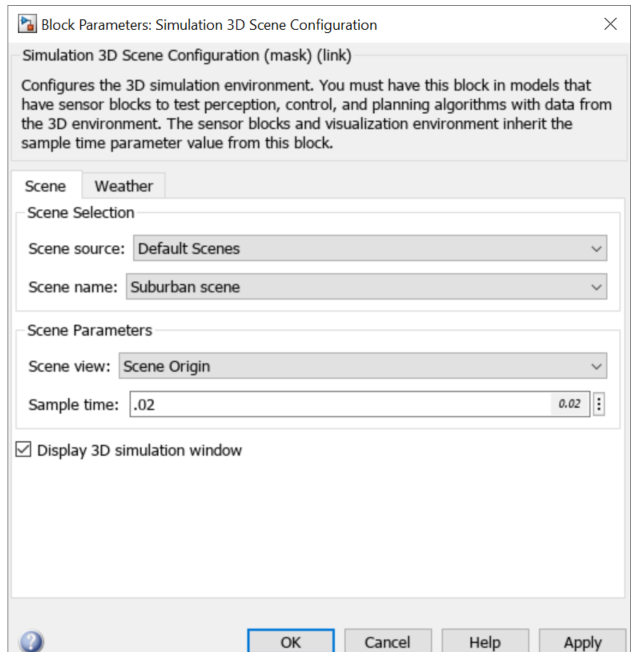
```
sim3d.maps.Map.local
```

```
        MapName                          Description                      Version    MinimumRelease
    _________________    ____________________________________________    _______    ______________

    "Suburban scene"    "a suburban area beyond the city's border"       "1"           "R2022b"
```

Add the Simulation 3D Scene Configuration block to your model.



Open the block mask and select the suburban scene from **Scene name**.



Run the model.

Input Arguments

Scene — Name of scene

string | character array

Name of the map being downloaded from the server, specified as a string or character array. Maps are downloaded in the default folder that is added to MATLAB search path at startup.

Maps are stored by user profile. For multiuser setup with a single MATLAB installation, the maps will be downloaded multiple times.

If a new version of the map is available on the server, you will see a warning message asking you to download the map again to get the recent version.

Version History

Introduced in R2022b

See Also

`sim3d.maps` | `sim3d.maps.Map.server` | `sim3d.maps.Map.delete` | `sim3d.maps.Map.local`

sim3d.maps.Map.local

List of locally available maps

Syntax

```
sim3d.maps.Map.local
```

Description

`sim3d.maps.Map.local` lists the locally available maps.

Examples

Download Suburban Scene Map

This example shows how to download and access the Suburban scene map from the Simulation 3D Scene Configuration block.

To begin, check the maps available in the server.

```
sim3d.maps.Map.server
```

MapName	Description	Version	MinimumRelease
"Suburban scene"	"a suburban area beyond the city's border"	"1"	"R2022b"

Download the Suburban scene from the server.

```
sim3d.maps.Map.download('Suburban scene')
```

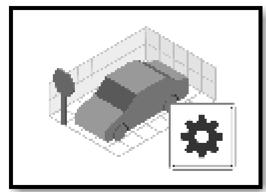
```
Map is susccesfully downloaded and is up-to-date
```

Check if the downloaded maps are available in your local machine.

```
sim3d.maps.Map.local
```

MapName	Description	Version	MinimumRelease
"Suburban scene"	"a suburban area beyond the city's border"	"1"	"R2022b"

Add the Simulation 3D Scene Configuration block to your model.



Open the block mask and select the suburban scene from **Scene name**.

Run the model.

Version History

Introduced in R2022b

See Also

`sim3d.maps` | `sim3d.maps.Map.download` | `sim3d.maps.Map.server` | `sim3d.maps.Map.delete`

sim3d.maps.Map.server

List of maps available for download from the server

Syntax

```
sim3d.maps.Map.server
```

Description

`sim3d.maps.Map.server` lists the available maps in the server.

Examples

Download Suburban Scene Map

This example shows how to download and access the Suburban scene map from the Simulation 3D Scene Configuration block.

To begin, check the maps available in the server.

```
sim3d.maps.Map.server
```

MapName	Description	Version	MinimumRelease
"Suburban scene"	"a suburban area beyond the city's border"	"1"	"R2022b"

Download the Suburban scene from the server.

```
sim3d.maps.Map.download('Suburban scene')
```

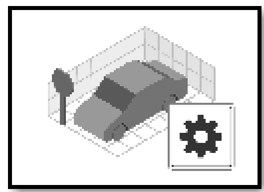
```
Map is susccesfully downloaded and is up-to-date
```

Check if the downloaded maps are available in your local machine.

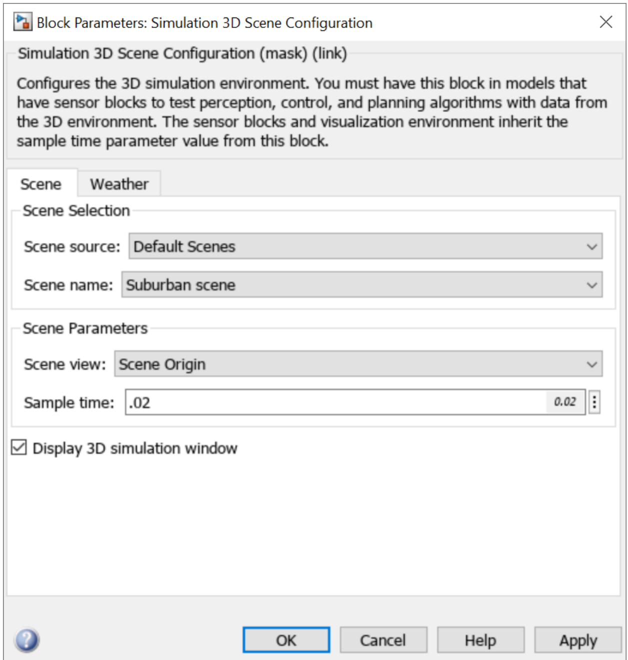
```
sim3d.maps.Map.local
```

MapName	Description	Version	MinimumRelease
"Suburban scene"	"a suburban area beyond the city's border"	"1"	"R2022b"

Add the Simulation 3D Scene Configuration block to your model.



Open the block mask and select the suburban scene from **Scene name**.



Run the model.

Version History

Introduced in R2022b

See Also

`sim3d.maps` | `sim3d.maps.Map.download` | `sim3d.maps.Map.delete` | `sim3d.maps.Map.local`

2

Methods

show

Display 3D vector field histogram

Syntax

```
show(vfh3D)
show(vfh3D,Name=Value)
ax = show( ___ )
```

Description

`show(vfh3D)` displays the 3D histogram of the desired direction calculated by the 3DVFH+ algorithm. The figure also includes the UAV position, UAV orientation, obstacle positions, and target position specified to the `controllerVFH3D` object `vfh3D` at the most recent object call.

`show(vfh3D,Name=Value)` specifies options using one or more name-value arguments. For example, `show(vfh3D,PlotsToShow="Cost Matrix")` displays only the cost matrix plot instead of the default histogram.

`ax = show( ___ )` returns the axes handles of all displayed histograms, in addition to any combination of input arguments from previous syntaxes.

Examples

Create 3D Vector Field Histogram Object and Visualize Data

Create a `controllerVFH3D` object.

```
vfh3D = controllerVFH3D;
```

Create obstacles.

```
az = [-60:-20 20:60]*(pi/180);
el = (-30:30)*(pi/180);
[El,Az] = meshgrid(el,az);
```

Specify the distances of the obstacles from the sensor, and convert to Cartesian coordinates.

```
obstacleDist = linspace(15,20,numel(El(:)));
[xSensor,ySensor,zSensor] = sph2cart(Az(:),El(:),obstacleDist');
```

Align the sensor and histogram frames.

```
vfh3D.SensorOrientation = [-180 0 0];
```

Specify the sensor range limits.

```
vfh3D.DistanceLimits = [0.2 25];
```

Specify the current UAV position and orientation, the locations of obstacles, and the target position for the UAV.

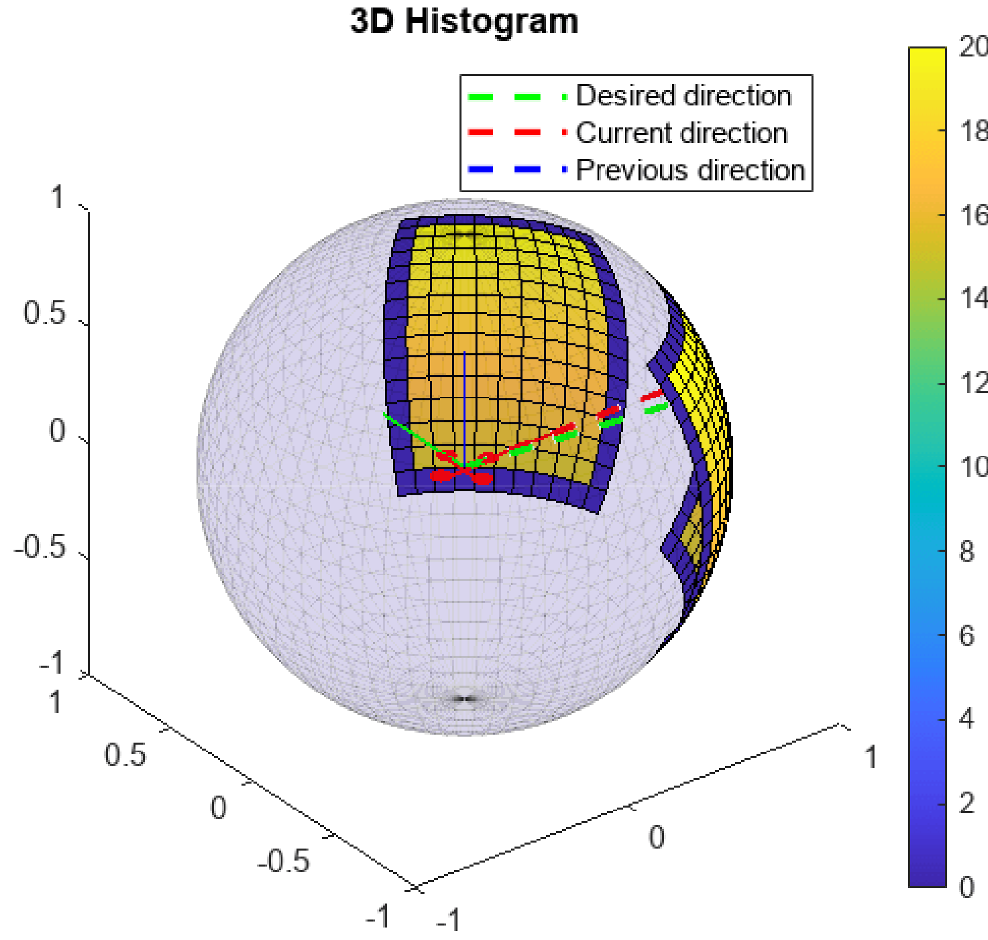
```
uavPosition = [0; 0; 0];
uavOrientation = [1; 0; 0; 0];
sensorPoints = [xSensor ySensor zSensor];
targetPosition = [20; 0; 0];
```

Compute an obstacle-free direction and desired yaw for the UAV, and return the status of the obstacle-free direction.

```
[desiredDirection,desiredYaw,status] = vfh3D(uavPosition, ...
                                            uavOrientation, ...
                                            sensorPoints, ...
                                            targetPosition);
```

Visualize the default histogram of the calculated direction.

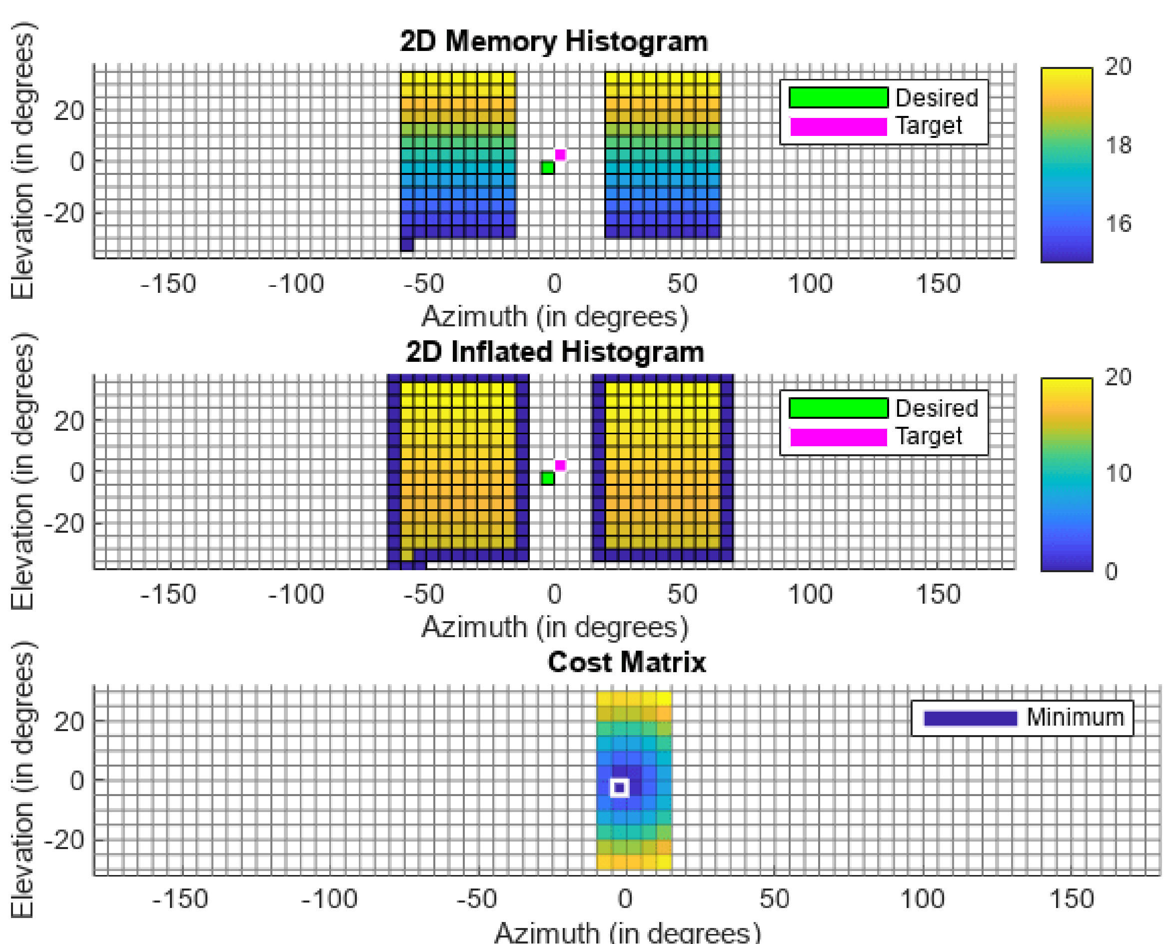
```
show(vfh3D)
axis equal
```



Visualize the 2D memory histogram, 2D inflated histogram, and cost matrix.

```
figure
ax(1) = subplot(3,1,1);
```

```
ax(2) = subplot(3,1,2);
ax(3) = subplot(3,1,3);
show(vfh3D, ...
     Parent=ax, ...
     PlotsToShow=["2D Memory Histogram","2D Inflated Histogram","Cost Matrix"])
axis(ax,"equal")
xlim(ax,"tight")
```



Input Arguments

vfh3D — 3-D vector field histogram algorithm
controllerVFH3D object

3-D vector field histogram algorithm, specified as a `controllerVFH3D` object. This object contains properties for tuning the 3DVFH+ algorithm.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Example: `show(vfh3D,PlotsToShow="Cost Matrix")` shows the cost matrix plot.

`Parent` — Axes on which to plot histograms
`Axes` object | array of `Axes` objects

Axes on which to plot the histograms, specified as an `Axes` object or an array of `Axes` objects.

Example: `Parent=ax`

`PlotsToShow` — Histogram plots to display
`"3D Histogram"` (default) | string scalar | string array

Histogram plots to display, specified as a string scalar or string array. Each element must be specified as one of these options: `"3D Histogram"`, `"2D Memory Histogram"`, `"2D Inflated Histogram"`, and `"Cost Matrix"` or `"All"`.

Example: `PlotsToShow="All"`

Example: `PlotsToShow="2D Memory Histogram"`

Example: `PlotsToShow=["2D Memory Histogram","Cost Matrix"]`

Data Types: `string`

Output Arguments

ax — Axes handles of 3D vector field histograms
`Axes` object | array of `Axes` objects

Axes handles of the 3D vector field histograms, returned as an `Axes` object or an array of `Axes` objects.

Version History

Introduced in R2022b

See Also

`controllerVFH3D`

applyTransform

Apply forward transformation to mesh vertices

Syntax

`transformedMesh = applyTransform(mesh,T)`

Description

`transformedMesh = applyTransform(mesh,T)` applies the forward transformation matrix `T` to the vertices of the object mesh.

Examples

Create and Transform Cuboid Mesh

Create an `extendedObjectMesh` object and transform the object by using a transformation matrix.

Create a cuboid mesh of unit dimensions.

```
cuboid = extendedObjectMesh('cuboid');
```

Create a transformation matrix that is a combination of a translation, a scaling, and a rotation.

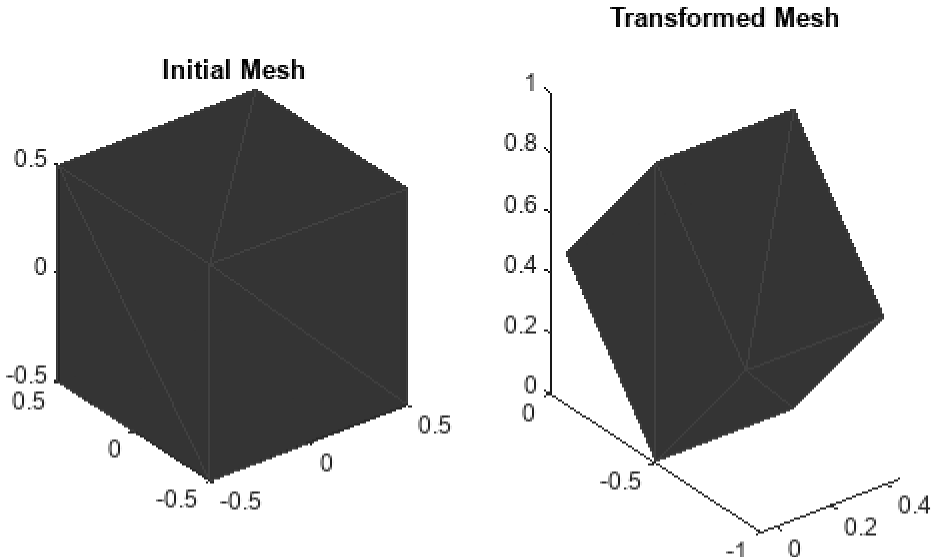
```
tform = makehgtform('translate',[0.2 -0.5 0.5], ...
    'scale',[0.5 0.6 0.7], ...
    'xrotate',pi/4);
```

Transform the mesh.

```
transformedCuboid = applyTransform(cuboid,tform);
```

Visualize the meshes.

```
subplot(1,2,1);
show(cuboid);
title('Initial Mesh')

subplot(1,2,2);
show(transformedCuboid);
title('Transformed Mesh')
```



Input Arguments

`mesh` — Extended object mesh
`extendedObjectMesh` object

Extended object mesh, specified as an `extendedObjectMesh` object.

`T` — Transformation matrix
4-by-4 matrix

Transformation matrix applied on the object mesh, specified as a 4-by-4 matrix. The 3-D coordinates of each point in the object mesh is transformed according to this formula:

```
[xT; yT; zT; 1] = T*[x; y; z; 1]
```

`xT`, `yT`, and `zT` are the transformed 3-D coordinates of the point.

Data Types: `single` | `double`

Output Arguments

`transformedMesh` — Transformed object mesh
`extendedObjectMesh` object

Transformed object mesh, returned as an `extendedObjectMesh` object.

Version History

Introduced in R2020b

See Also

Objects
extendedObjectMesh

Functions
rotate | translate | scale | join | scaleToFit | show

join

Join two object meshes

Syntax

```
joinedMesh = join(mesh1,mesh2)
```

Description

`joinedMesh = join(mesh1,mesh2)` joins the object meshes `mesh1` and `mesh2` and returns `joinedMesh` with the combined objects.

Examples

Create and Join Two Object Meshes

Create `extendedObjectMesh` objects and join them together.

Construct two meshes of unit dimensions.

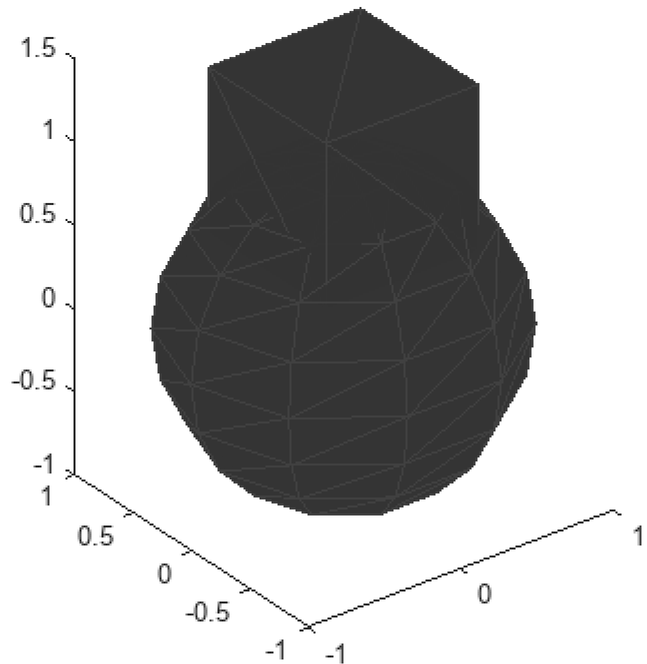
```
sph = extendedObjectMesh('sphere');
cub = extendedObjectMesh('cuboid');
```

Join the two meshes.

```
cub = translate(cub,[0 0 1]);
sphCub = join(sph,cub);
```

Visualize the final mesh.

```
show(sphCub);
```



Input Arguments

`mesh1` — Extended object mesh
`extendedObjectMesh` object

Extended object mesh, specified as an `extendedObjectMesh` object.

`mesh2` — Extended object mesh
`extendedObjectMesh` object

Extended object mesh, specified as an `extendedObjectMesh` object.

Output Arguments

`joinedMesh` — Joined object mesh
`extendedObjectMesh` object

Joined object mesh, specified as an `extendedObjectMesh` object.

Version History

Introduced in R2020b

See Also

Objects
extendedObjectMesh

Functions
rotate | translate | scale | applyTransform | scaleToFit | show

rotate

Rotate mesh about coordinate axes

Syntax

```
rotatedMesh = rotate(mesh,orient)
```

Description

`rotatedMesh = rotate(mesh,orient)` rotate the `mesh` object by an orientation, `orient`.

Examples

Create and Rotate Cuboid Mesh

Create an `extendedObjectMesh` object and rotate the object.

Construct a cuboid mesh.

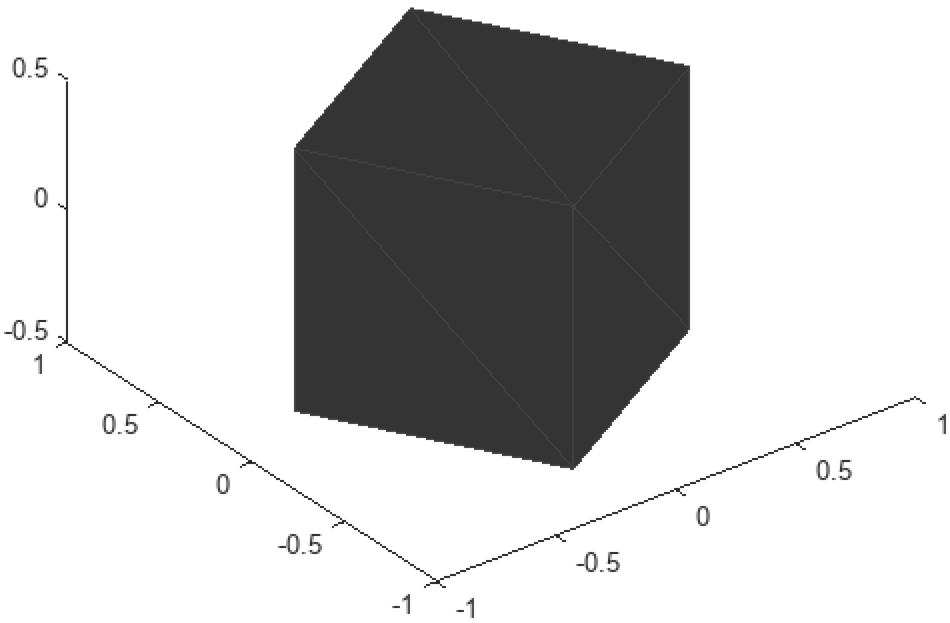
```
mesh = extendedObjectMesh('cuboid');
```

Rotate the mesh by 30 degrees around the *z* axis.

```
mesh = rotate(mesh,[30 0 0]);
```

Visualize the mesh.

```
ax = show(mesh);
```



Input Arguments

`mesh` — Extended object mesh
`extendedObjectMesh` object

Extended object mesh, specified as an `extendedObjectMesh` object.

`orient` — Description of rotation
3-by-3 orthonormal matrix | quaternion | 1-by-3 vector

Description of rotation for an object mesh, specified as:

- 3-by-3 orthonormal rotation matrix
- `quaternion`
- 1-by-3 vector, where the elements are positive rotations in degrees about the *z*, *y*, and *x* axes, in that order.

Output Arguments

`rotatedMesh` — Rotated object mesh
`extendedObjectMesh` object

Rotated object mesh, returned as an `extendedObjectMesh` object.

Version History

Introduced in R2020b

See Also

Objects
`extendedObjectMesh`

Functions
`translate` | `scale` | `applyTransform` | `join` | `scaleToFit` | `show`

scale

Scale mesh in each dimension

Syntax

```
scaledMesh = scale(mesh,scaleFactor)
scaledMesh = scale(mesh,[sx sy sz])
```

Description

`scaledMesh = scale(mesh,scaleFactor)` scales the object mesh by `scaleFactor`. `scaleFactor` can be the same for all dimensions or defined separately as elements of a 1-by-3 vector in the order *x*, *y*, and *z*.

`scaledMesh = scale(mesh,[sx sy sz])` scales the object mesh along the dimensions *x*, *y*, and *z* by the scaling factors `sx`, `sy`, and `sz`.

Examples

Create and Scale Cuboid Mesh

Create an `extendedObjectMesh` object and scale the object.

Construct a cuboid mesh of unit dimensions.

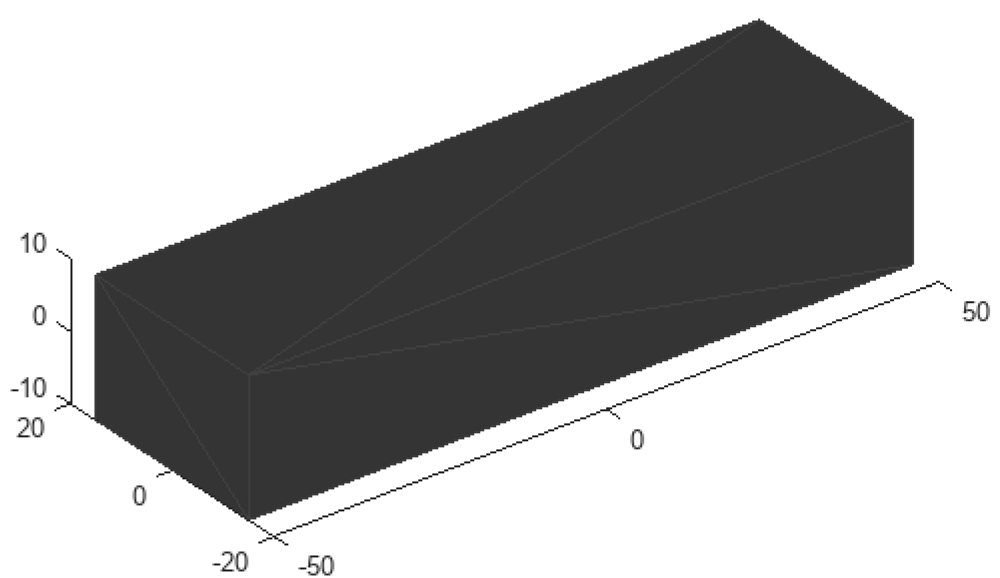
```
cuboid = extendedObjectMesh('cuboid');
```

Scale the mesh by different factors along each of the three axes.

```
scaledCuboid = scale(cuboid,[100 30 20]);
```

Visualize the mesh.

```
show(scaledCuboid);
```



Input Arguments

`mesh` — Extended object mesh
`extendedObjectMesh` object

Extended object mesh, specified as an `extendedObjectMesh` object.

`scaleFactor` — Scaling factor
positive real scalar | 1-by-3 vector

Scaling factor for the object mesh, specified as a positive real scalar or as a 1-by-3 vector in the order *x*, *y*, and *z*.

Data Types: `single` | `double`

`sx` — Scaling factor for *x*-axis
positive real scalar

Scaling factor for *x*-axis, specified as a positive real scalar.

Data Types: `single` | `double`

`sy` — Scaling factor for *y*-axis
positive real scalar

Scaling factor for *y*-axis, specified as a positive real scalar.

Data Types: `single` | `double`

`sz` — Scaling factor for *z*-axis
positive real scalar

Scaling factor for *z*-axis, specified as a positive real scalar.

Data Types: `single` | `double`

Output Arguments

`scaledMesh` — Scaled object mesh
`extendedObjectMesh` object

Scaled object mesh, returned as an `extendedObjectMesh` object.

Version History

Introduced in R2020b

See Also

Objects
`extendedObjectMesh`

Functions
`rotate` | `translate` | `applyTransform` | `join` | `scaleToFit` | `show`

scaleToFit

Auto-scale object mesh to match specified cuboid dimensions

Syntax

```
scaledMesh = scaleToFit(mesh,dims)
```

Description

`scaledMesh = scaleToFit(mesh,dims)` auto-scales the object mesh to match the dimensions of a cuboid specified in the structure `dims`.

Examples

Create and Auto-Scale Sphere Mesh

Create an `extendedObjectMesh` object and auto-scale the object to the required dimensions.

Construct a sphere mesh of unit dimensions.

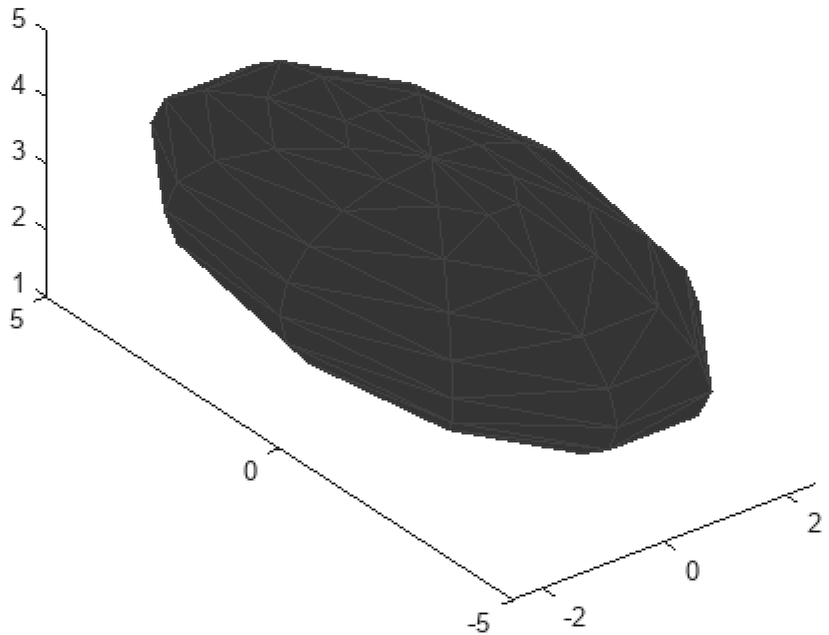
```
sph = extendedObjectMesh('sphere');
```

Auto-scale the mesh to the dimensions in `dims`.

```
dims = struct('Length',5,'Width',10,'Height',3,'OriginOffset',[0 0 -3]);
sph = scaleToFit(sph,dims);
```

Visualize the mesh.

```
show(sph);
```



Input Arguments

mesh — Extended object mesh
extendedObjectMesh object

Extended object mesh, specified as an extendedObjectMesh object.

dims — Cuboid dimensions
structure

Dimensions of the cuboid to scale an object mesh, specified as a struct with these fields:

- Length – Length of the cuboid
- Width – Width of the cuboid
- Height – Height of the cuboid
- OriginOffset – Origin offset in 3-D coordinates

All the dimensions are in meters.

Data Types: struct

Output Arguments

`scaledMesh` — Scaled object mesh
`extendedObjectMesh` object

Scaled object mesh, returned as an `extendedObjectMesh` object.

Version History

Introduced in R2020b

See Also

Objects
`extendedObjectMesh`

Functions
`rotate` | `translate` | `scale` | `applyTransform` | `join` | `show`

show

Display the mesh as a patch on the current axes

Syntax

```
show(mesh)
show(mesh,ax)
ax = show(mesh)
```

Description

`show(mesh)` displays the `extendedObjectMesh` as a patch on the current axes. If there are no active axes, the function creates new axes.

`show(mesh,ax)` displays the object mesh as a patch on the axes `ax`.

`ax = show(mesh)` optionally outputs the handle to the axes where the mesh was plotted.

Examples

Create and Translate Cuboid Mesh

Create an `extendedObjectMesh` object and translate the object.

Construct a cuboid mesh.

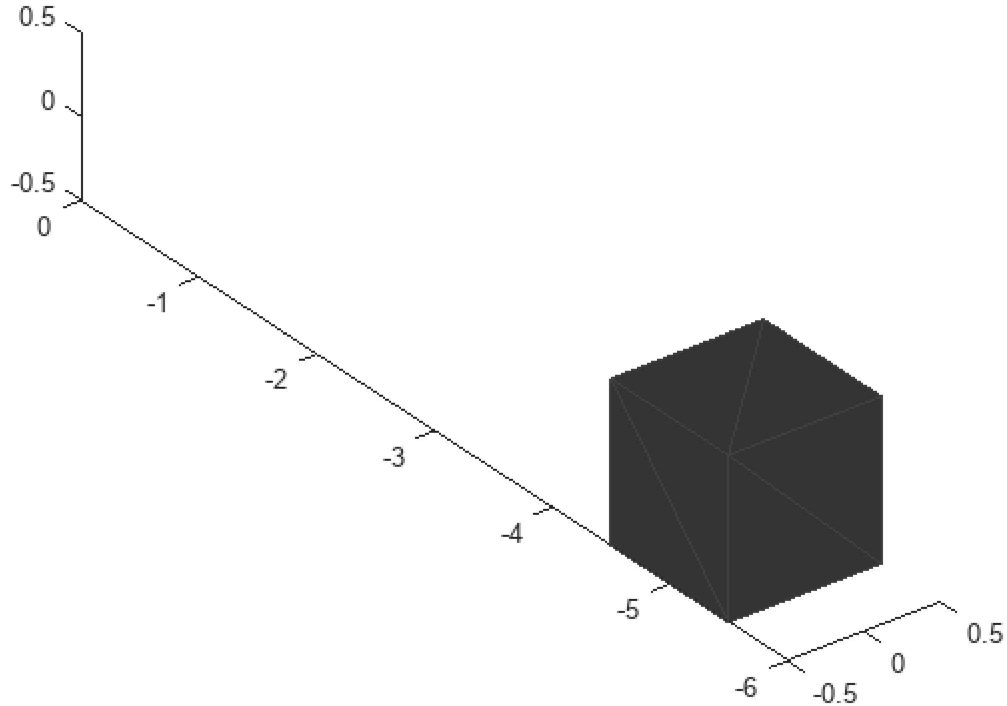
```
mesh = extendedObjectMesh('cuboid');
```

Translate the mesh by 5 units along the negative *y* axis.

```
mesh = translate(mesh,[0 -5 0]);
```

Visualize the mesh.

```
ax = show(mesh);
ax.YLim = [-6 0];
```



Input Arguments

`mesh` — Extended object mesh
`extendedObjectMesh` object

Extended object mesh, specified as an `extendedObjectMesh` object.

ax — Current axes
`axes` object

Current axes, specified as an `axes` object.

Version History

Introduced in R2020b

See Also

Objects
`extendedObjectMesh`

Functions
`rotate` | `translate` | `scale` | `applyTransform` | `join` | `scaleToFit`

translate

Translate mesh along coordinate axes

Syntax

```
translatedMesh = translate(mesh,deltaPos)
```

Description

`translatedMesh = translate(mesh,deltaPos)` translates the object mesh by the distances specified by `deltaPos` along the coordinate axes.

Examples

Create and Translate Cuboid Mesh

Create an `extendedObjectMesh` object and translate the object.

Construct a cuboid mesh.

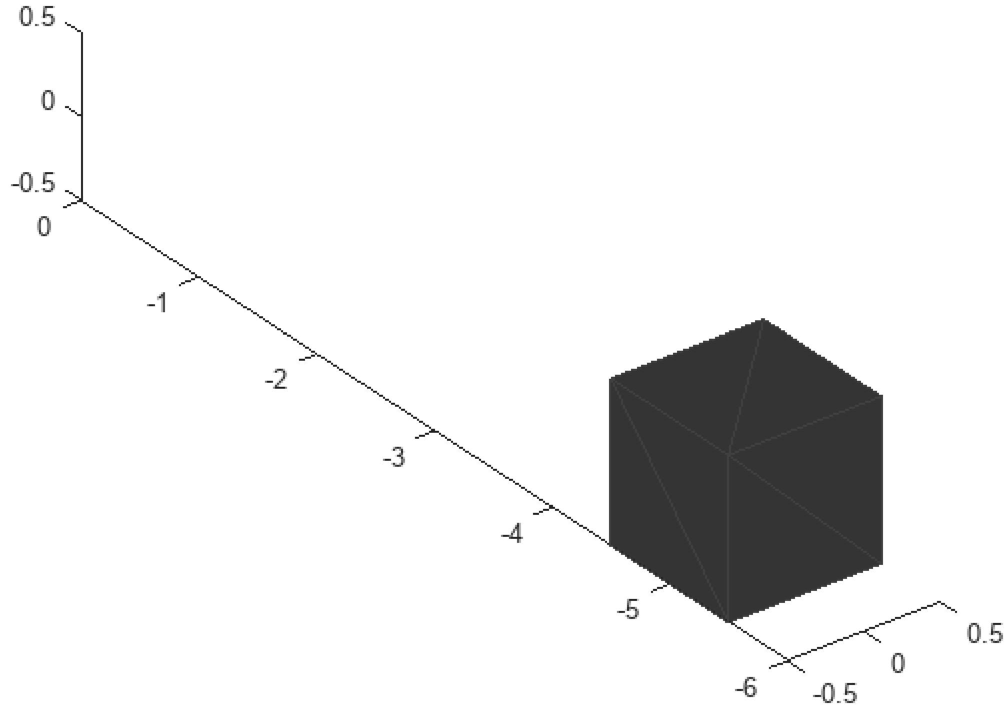
```
mesh = extendedObjectMesh('cuboid');
```

Translate the mesh by 5 units along the negative *y* axis.

```
mesh = translate(mesh,[0 -5 0]);
```

Visualize the mesh.

```
ax = show(mesh);
ax.YLim = [-6 0];
```



Input Arguments

`mesh` — Extended object mesh
`extendedObjectMesh` object

Extended object mesh, specified as an `extendedObjectMesh` object.

`deltaPos` — Translation vector
three-element real-valued vector

Translation vector for an object mesh, specified as a three-element real-valued vector. The three elements in the vector define the translation along the *x*, *y*, and *z* axes.

Data Types: `single` | `double`

Output Arguments

`translatedMesh` — Translated object mesh
`extendedObjectMesh` object

Translated object mesh, returned as an `extendedObjectMesh` object.

Version History

Introduced in R2020b

See Also

Objects

`extendedObjectMesh`

Functions

`rotate` | `scale` | `applyTransform` | `join` | `scaleToFit` | `show`

control

Control commands for UAV

Syntax

```
controlStruct = control(uavGuidanceModel)
```

Description

`controlStruct = control(uavGuidanceModel)` returns a structure that captures all the relevant control commands for the specified UAV guidance model. Use the output of this function to ensure you have the proper fields for your control. Use the control commands as an input to the `derivative` function to get the state time-derivative of the UAV.

Examples

Simulate A Multirotor Control Command

This example shows how to use the `multirotor` guidance model to simulate the change in state of a UAV due to a command input.

Create the multirotor guidance model.

```
model = multirotor;
```

Create a state structure. Specify the location in world coordinates.

```
s = state(model);
s(1:3) = [3;2;1];
```

Specify a control command, `u`, that specified the roll and thrust of the multirotor.

```
u = control(model);
u.Roll = pi/12;
u.Thrust = 1;
```

Create a default environment without wind.

```
e = environment(model);
```

Compute the time derivative of the state given the current state, control command, and environment.

```
sdot = derivative(model,s,u,e);
```

Simulate the UAV state using `ode45` integration. The `y` field outputs the multirotor UAV states as a 13-by-*n* matrix.

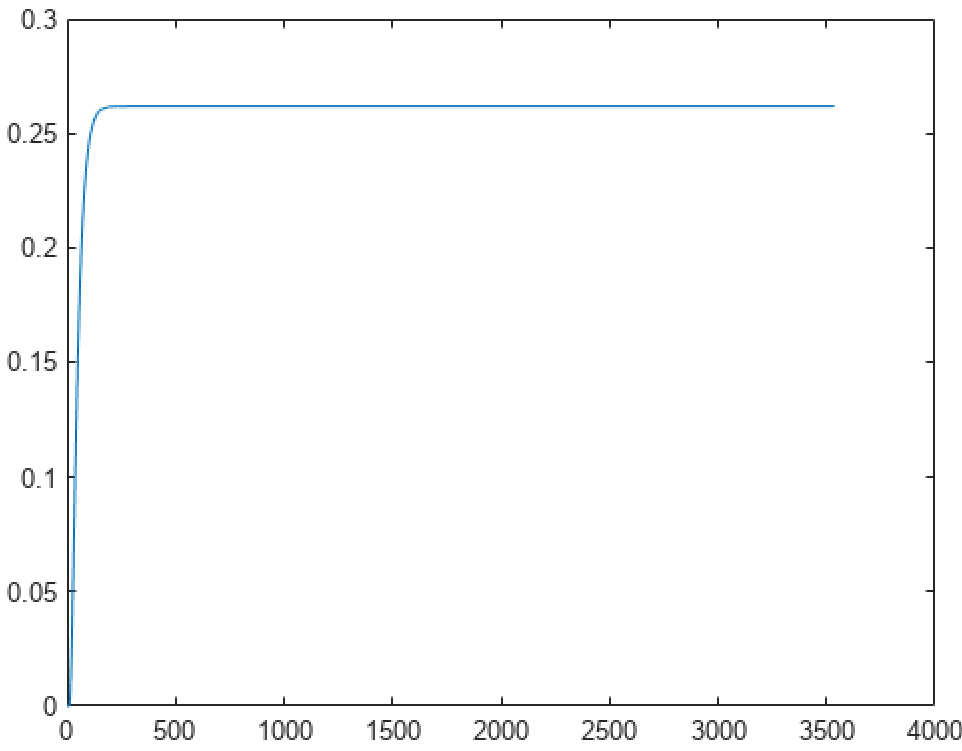
```
simOut = ode45(@(~,x)derivative(model,x,u,e), [0 3], s);
size(simOut.y)
```

```
ans = 1×2
```

```
13          3536
```

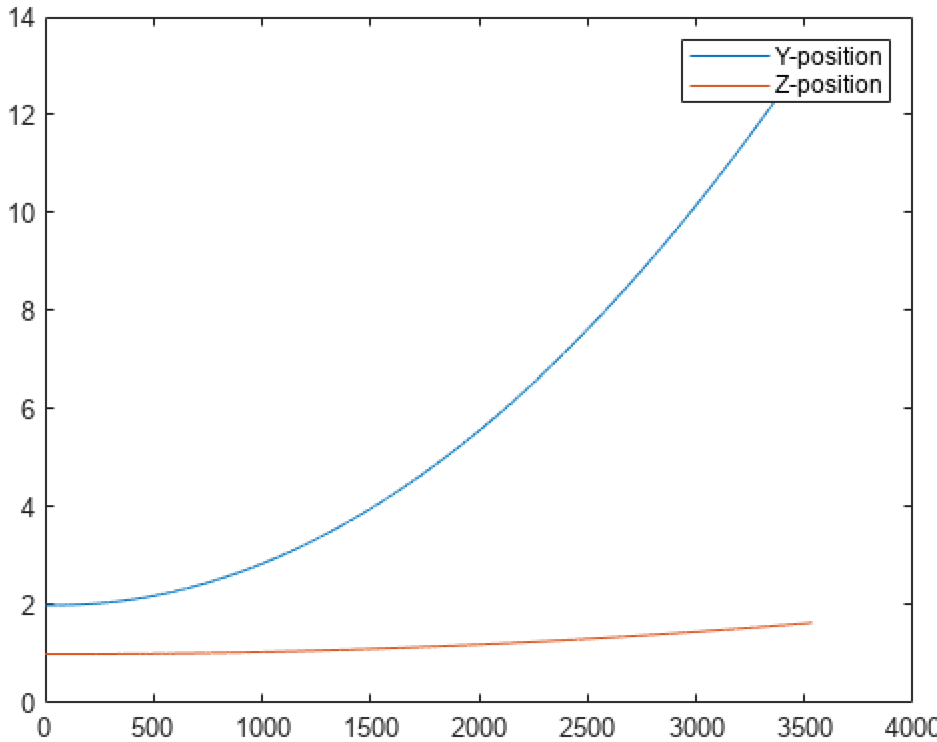
Plot the change in roll angle based on the simulation output. The roll angle (the X Euler angle) is the 9th row of the `simOut.y` output.

```
plot(simOut.y(9,:))
```



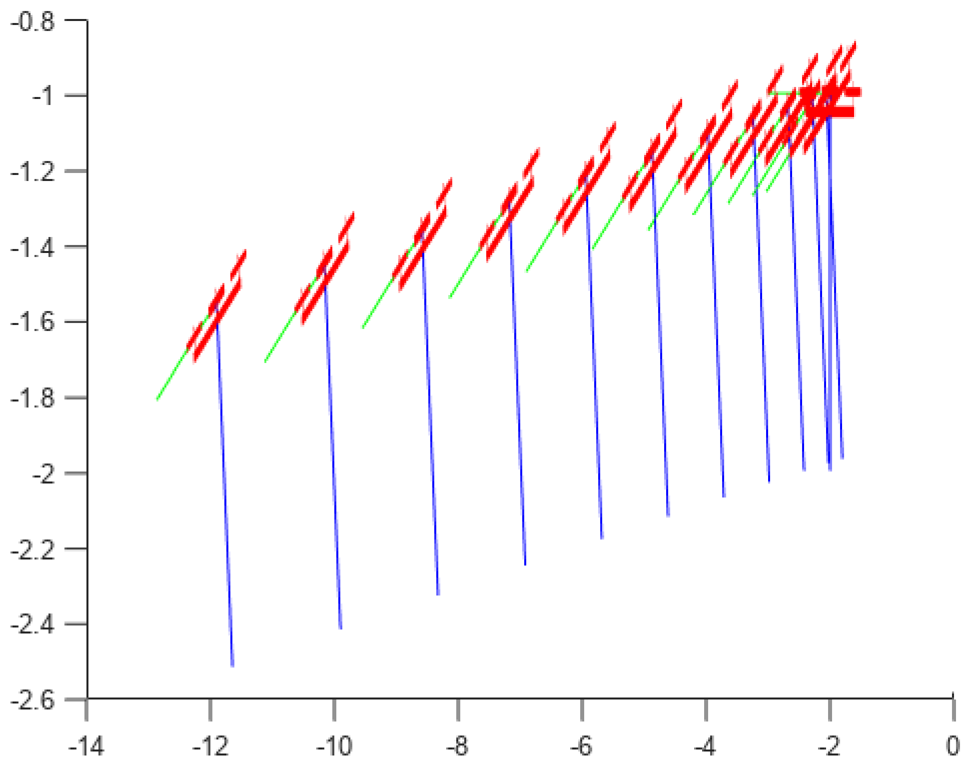
Plot the change in the Y and Z positions. With the specified thrust and roll angle, the multirotor should fly over and lose some altitude. A positive value for Z is expected as positive Z is down.

```
figure
plot(simOut.y(2,:));
hold on
plot(simOut.y(3,:));
legend('Y-position','Z-position')
hold off
```



You can also plot the multirotor trajectory using `plotTransforms`. Create the translation and rotation vectors from the simulated state. Downsample (every 300th element) and transpose the `simOut` elements, and convert the Euler angles to quaternions. Specify the mesh as the `multirotor.stl` file and the positive Z-direction as `"down"`. The displayed view shows the UAV translating in the Y-direction and losing altitude.

```
translations = simOut.y(1:3,1:300:end)'; % xyz position
rotations = eul2quat(simOut.y(7:9,1:300:end)'); % ZYX Euler
plotTransforms(translations,rotations,...
    'MeshFilePath','multirotor.stl','InertialZDirection',"down")
view([90.00 -0.60])
```



Simulate A Fixed-Wing Control Command

This example shows how to use the `fixedwing` guidance model to simulate the change in state of a UAV due to a command input.

Create the fixed-wing guidance model.

```
model = fixedwing;
```

Set the air speed of the vehicle by modifying the structure from the `state` function.

```
s = state(model);
s(4) = 5; % 5 m/s
```

Specify a control command, `u`, that maintains the air speed and gives a roll angle of `pi/12`.

```
u = control(model);
u.RollAngle = pi/12;
u.AirSpeed = 5;
```

Create a default environment without wind.

```
e = environment(model);
```

Compute the time derivative of the state given the current state, control command, and environment.

```
sdot = derivative(model,s,u,e);
```

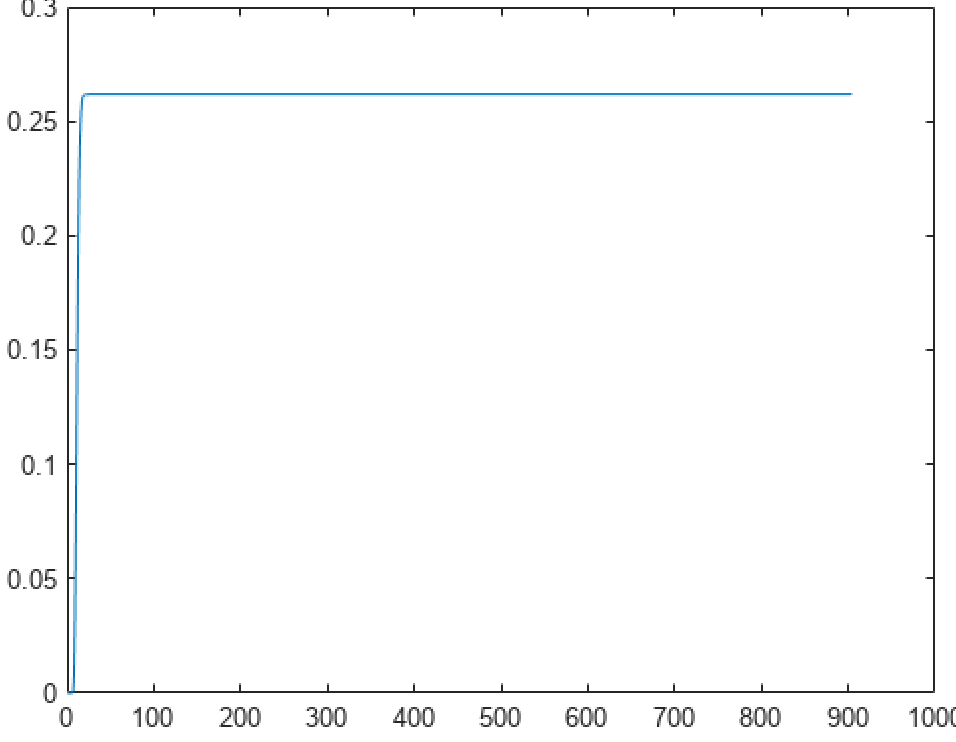
Simulate the UAV state using `ode45` integration. The `y` field outputs the fixed-wing UAV states based on this simulation.

```
simOut = ode45(@(~,x)derivative(model,x,u,e), [0 50], s);
size(simOut.y)
```

```
ans = 1×2

     8   904
```

Plot the change in roll angle based on the simulation output. The roll angle is the 7th row of the `simOut.y` output.

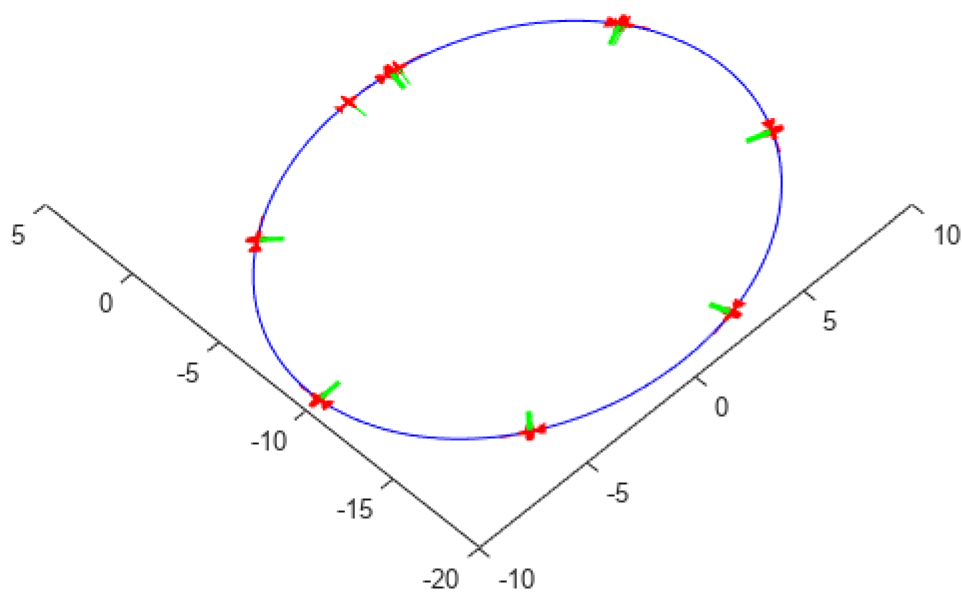
```
plot(simOut.y(7,:))
```



You can also plot the fixed-wing trajectory using `plotTransforms`. Create the translation and rotation vectors from the simulated state. Downsample (every 30th element) and transpose the `simOut` elements, and convert the Euler angles to quaternions. Specify the mesh as the `fixedwing.stl` file and the positive Z-direction as `"down"`. The displayed view shows the UAV making a constant turn based on the constant roll angle.

```
downsample = 1:30:size(simOut.y,2);
translations = simOut.y(1:3,downsample)'; % xyz-position
rotations = eul2quat([simOut.y(5,downsample)',simOut.y(6,downsample)',simOut.y(7,downsample)']);
```

```
plotTransforms(translations,rotations,...
    'MeshFilePath','fixedwing.stl','InertialZDirection',"down")
hold on
plot3(simOut.y(1,:),-simOut.y(2,:),simOut.y(3,:),'--b') % full path
xlim([-10.0 10.0])
ylim([-20.0 5.0])
zlim([-0.5 4.00])
view([-45 90])
hold off
```



Input Arguments

uavGuidanceModel — UAV guidance model
fixedwing object | multirotor object

UAV guidance model, specified as a `fixedwing` or `multirotor` object.

Output Arguments

controlStruct — Control commands for UAV
structure

Control commands for UAV, returned as a structure.

For multirotor UAVs, the guidance model is approximated as separate PD controllers for each command. The elements of the structure are control commands:

- `Roll` - Roll angle in radians.
- `Pitch` - Pitch angle in radians.
- `YawRate` - Yaw rate in radians per second. (D = 0. P only controller)
- `Thrust` - Vertical thrust of the UAV in Newtons. (D = 0. P only controller)

For fixed-wing UAVs, the model assumes the UAV is flying under the coordinated-turn condition. The guidance model equations assume zero side-slip. The elements of the structure are:

- `Height` - Altitude above the ground in meters.
- `Airspeed` - UAV speed relative to wind in meters per second.
- `RollAngle` - Roll angle along body forward axis in radians. Because of the coordinated-turn condition, the heading angular rate is based on the roll angle.

Version History

Introduced in R2018b

Extended Capabilities

C/C++ Code Generation

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

See Also

Functions

`ode45` | `derivative` | `environment` | `state` | `plotTransforms`

Objects

`fixedwing` | `multirotor`

Blocks

Waypoint Follower | UAV Guidance Model

Topics

"Approximate High-Fidelity UAV Model with UAV Guidance Model Block"
"Tuning Waypoint Follower for Fixed-Wing UAV"

derivative

Time derivative of UAV states

Syntax

```
stateDerivative = derivative(uavGuidanceModel,state,control,environment)
```

Description

`stateDerivative = derivative(uavGuidanceModel,state,control,environment)` determines the time derivative of the state of the UAV guidance model using the current state, control commands, and environmental inputs. Use the state and time derivative with `ode45` to simulate the UAV.

Examples

Simulate A Multirotor Control Command

This example shows how to use the `multirotor` guidance model to simulate the change in state of a UAV due to a command input.

Create the multirotor guidance model.

```
model = multirotor;
```

Create a state structure. Specify the location in world coordinates.

```
s = state(model);
s(1:3) = [3;2;1];
```

Specify a control command, `u`, that specified the roll and thrust of the multirotor.

```
u = control(model);
u.Roll = pi/12;
u.Thrust = 1;
```

Create a default environment without wind.

```
e = environment(model);
```

Compute the time derivative of the state given the current state, control command, and environment.

```
sdot = derivative(model,s,u,e);
```

Simulate the UAV state using `ode45` integration. The `y` field outputs the multirotor UAV states as a 13-by-*n* matrix.

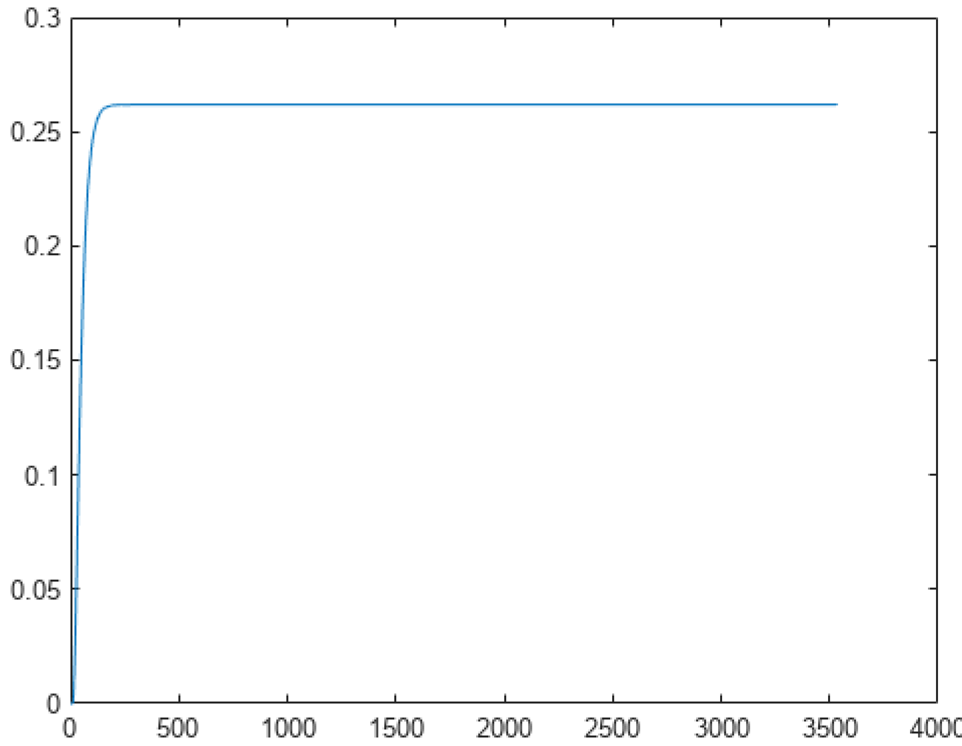
```
simOut = ode45(@(~,x)derivative(model,x,u,e), [0 3], s);
size(simOut.y)
```

```
ans = 1×2
```

```
    13          3536
```

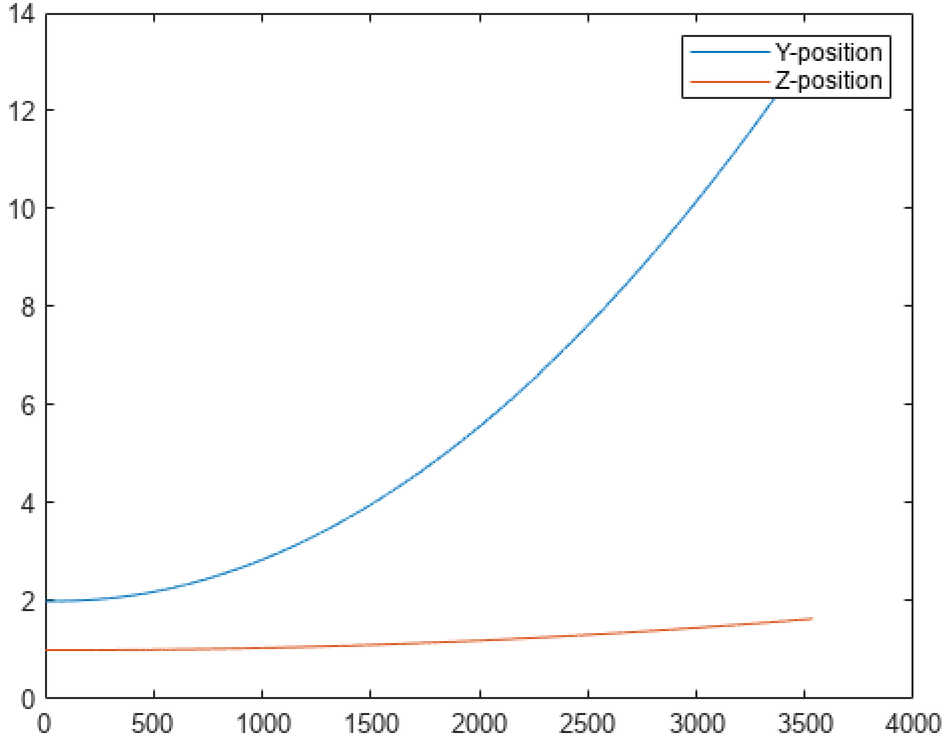
Plot the change in roll angle based on the simulation output. The roll angle (the X Euler angle) is the 9th row of the `simOut.y` output.

```
plot(simOut.y(9,:))
```



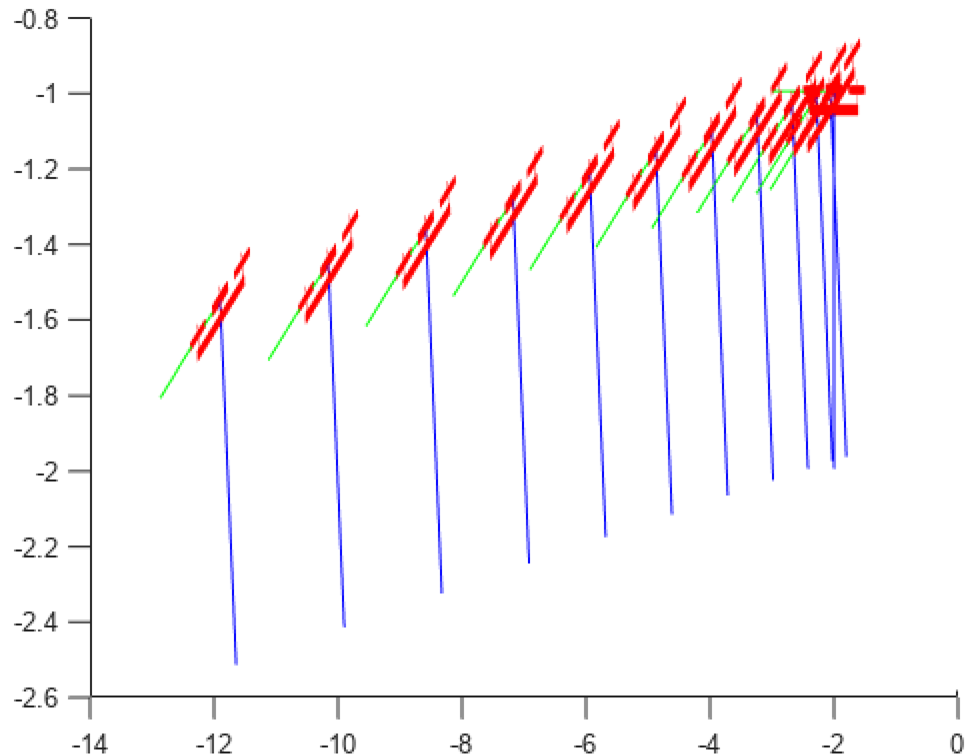
Plot the change in the Y and Z positions. With the specified thrust and roll angle, the multirotor should fly over and lose some altitude. A positive value for Z is expected as positive Z is down.

```
figure
plot(simOut.y(2,:));
hold on
plot(simOut.y(3,:));
legend('Y-position','Z-position')
hold off
```



You can also plot the multirotor trajectory using `plotTransforms`. Create the translation and rotation vectors from the simulated state. Downsample (every 300th element) and transpose the `simOut` elements, and convert the Euler angles to quaternions. Specify the mesh as the `multirotor.stl` file and the positive Z-direction as `"down"`. The displayed view shows the UAV translating in the Y-direction and losing altitude.

```
translations = simOut.y(1:3,1:300:end)'; % xyz position
rotations = eul2quat(simOut.y(7:9,1:300:end)'); % ZYX Euler
plotTransforms(translations,rotations,...
    'MeshFilePath','multirotor.stl','InertialZDirection',"down")
view([90.00 -0.60])
```



Simulate A Fixed-Wing Control Command

This example shows how to use the `fixedwing` guidance model to simulate the change in state of a UAV due to a command input.

Create the fixed-wing guidance model.

```
model = fixedwing;
```

Set the air speed of the vehicle by modifying the structure from the `state` function.

```
s = state(model);
s(4) = 5; % 5 m/s
```

Specify a control command, `u`, that maintains the air speed and gives a roll angle of `pi/12`.

```
u = control(model);
u.RollAngle = pi/12;
u.AirSpeed = 5;
```

Create a default environment without wind.

```
e = environment(model);
```

Compute the time derivative of the state given the current state, control command, and environment.

```
sdot = derivative(model,s,u,e);
```

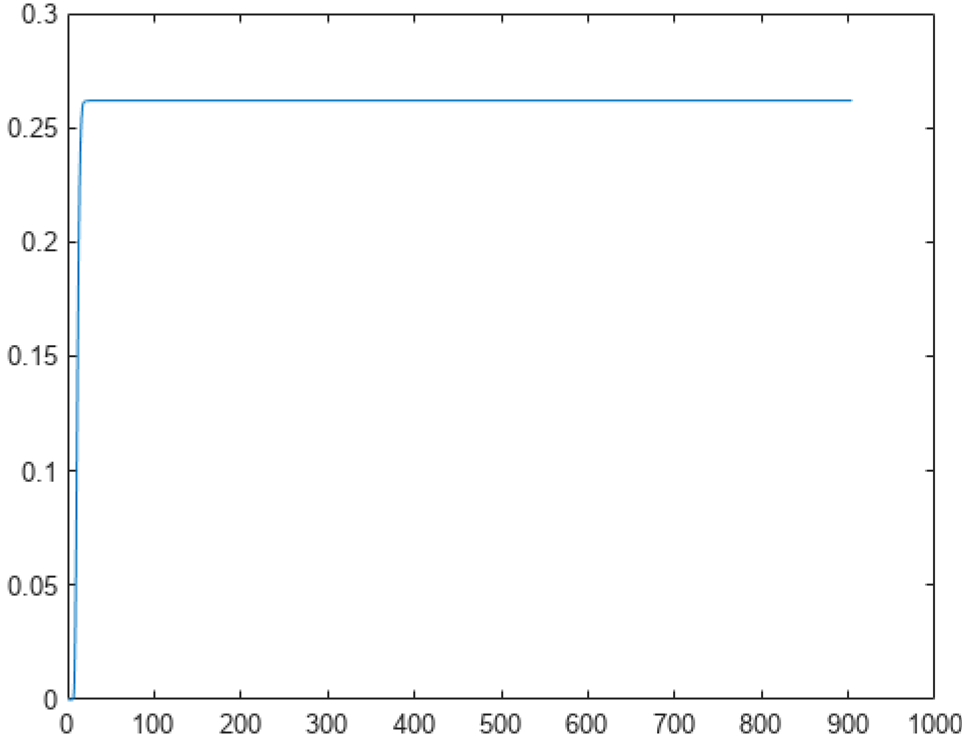
Simulate the UAV state using `ode45` integration. The `y` field outputs the fixed-wing UAV states based on this simulation.

```
simOut = ode45(@(~,x)derivative(model,x,u,e), [0 50], s);
size(simOut.y)
```

```
ans = 1×2

     8   904
```

Plot the change in roll angle based on the simulation output. The roll angle is the 7th row of the `simOut.y` output.

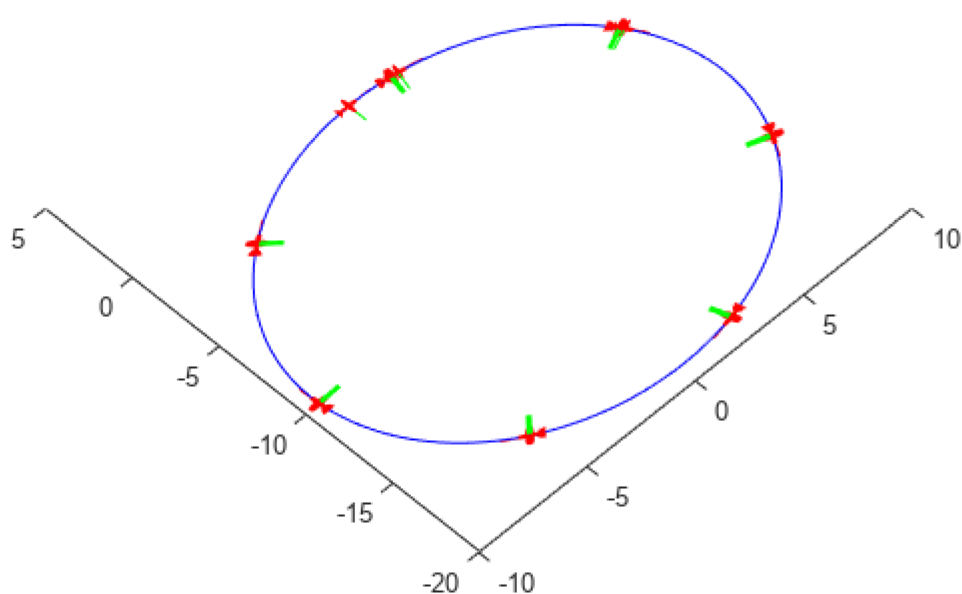
```
plot(simOut.y(7,:))
```



You can also plot the fixed-wing trajectory using `plotTransforms`. Create the translation and rotation vectors from the simulated state. Downsample (every 30th element) and transpose the `simOut` elements, and convert the Euler angles to quaternions. Specify the mesh as the `fixedwing.stl` file and the positive Z-direction as `"down"`. The displayed view shows the UAV making a constant turn based on the constant roll angle.

```
downsample = 1:30:size(simOut.y,2);
translations = simOut.y(1:3,downsample)'; % xyz-position
rotations = eul2quat([simOut.y(5,downsample)',simOut.y(6,downsample)',simOut.y(7,downsample)']);
```

```
plotTransforms(translations,rotations,...
    'MeshFilePath','fixedwing.stl','InertialZDirection',"down")
hold on
plot3(simOut.y(1,:),-simOut.y(2,:),simOut.y(3,:),'--b') % full path
xlim([-10.0 10.0])
ylim([-20.0 5.0])
zlim([-0.5 4.00])
view([-45 90])
hold off
```



Input Arguments

`uavGuidanceModel` — UAV guidance model
`fixedwing` object | `multirotor` object

UAV guidance model, specified as a `fixedwing` or `multirotor` object.

`state` — State vector
eight-element vector | thirteen-element vector

State vector, specified as a eight-element or thirteen-element vector. The vector is always filled with zeros. Use this function to ensure you have the proper size for your state vector.

For fixed-wing UAVs, the state is an eight-element vector:

- **North** - Position in north direction in meters.
- **East** - Position in east direction in meters.
- **Height** - Height above ground in meters.
- **AirSpeed** - Speed relative to wind in meters per second.
- **HeadingAngle** - Angle between ground velocity and north direction in radians.
- **FlightPathAngle** - Angle between ground velocity and north-east plane in radians.
- **RollAngle** - Angle of rotation along body *x*-axis in radians per second.
- **RollAngleRate** - Angular velocity of rotation along body *x*-axis in radians per second.

For multirotor UAVs, the state is a thirteen-element vector in this order:

- **World Position** - `[x y z]` in meters.
- **World Velocity** - `[vx vy vz]` in meters per second.
- **Euler Angles (ZYX)** - `[psi theta phi]` in radians.
- **Body Angular Rates** - `[p q r]` in radians per second.
- **Thrust** - `F` in Newtons.

`environment` — Environmental input parameters

structure

Environmental input parameters, returned as a structure. To generate this structure, use `environment`.

For fixed-wing UAVs, the fields of the structure are `WindNorth`, `WindEast`,`WindDown`, and `Gravity`. Wind speeds are in meters per second, and negative speeds point in the opposite direction. Gravity is in meters per second squared (default `9.81`).

For multirotor UAVs, the only element of the structure is `Gravity` (default `9.81`) in meters per second squared.

`control` — Control commands for UAV

structure

Control commands for UAV, specified as a structure. To generate this structure, use `control`.

For multirotor UAVs, the guidance model is approximated as separate PD controllers for each command. The elements of the structure are control commands:

- `Roll` - Roll angle in radians.
- `Pitch` - Pitch angle in radians.
- `YawRate` - Yaw rate in radians per second. (D = 0. P only controller)
- `Thrust` - Vertical thrust of the UAV in Newtons. (D = 0. P only controller)

For fixed-wing UAVs, the model assumes the UAV is flying under the coordinated-turn condition. The Guidance Model equations assume zero side-slip. The elements of the bus are:

- `Height` - Altitude above the ground in meters.
- `Airspeed` - UAV speed relative to wind in meters per second.
- `RollAngle` - Roll angle along body forward axis in radians. Because of the coordinated-turn condition, the heading angular rate is based on the roll angle.

Output Arguments

`stateDerivative` — Time derivative of state
vector

Time derivative of state, returned as a vector. The time derivative vector has the same length as the input `state`.

Version History

Introduced in R2018b

Extended Capabilities

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

See Also

Functions
`ode45` | `control` | `environment` | `state` | `plotTransforms`

Objects
`fixedwing` | `multirotor`

Blocks
Waypoint Follower | UAV Guidance Model

Topics
"Approximate High-Fidelity UAV Model with UAV Guidance Model Block"
"Tuning Waypoint Follower for Fixed-Wing UAV"

environment

Environmental inputs for UAV

Syntax

`envStruct = environment(uavGuidanceModel)`

Description

`envStruct = environment(uavGuidanceModel)` returns a structure that captures all the relevant environmental variables for the specified UAV guidance model. Use this function to ensure you have the proper fields for your environmental parameters. Use the environmental inputs as an input to the `derivative` function to get the state time-derivative of the UAV.

Examples

Simulate A Multirotor Control Command

This example shows how to use the `multirotor` guidance model to simulate the change in state of a UAV due to a command input.

Create the multirotor guidance model.

```
model = multirotor;
```

Create a state structure. Specify the location in world coordinates.

```
s = state(model);
s(1:3) = [3;2;1];
```

Specify a control command, `u`, that specified the roll and thrust of the multirotor.

```
u = control(model);
u.Roll = pi/12;
u.Thrust = 1;
```

Create a default environment without wind.

```
e = environment(model);
```

Compute the time derivative of the state given the current state, control command, and environment.

```
sdot = derivative(model,s,u,e);
```

Simulate the UAV state using `ode45` integration. The `y` field outputs the multirotor UAV states as a 13-by-*n* matrix.

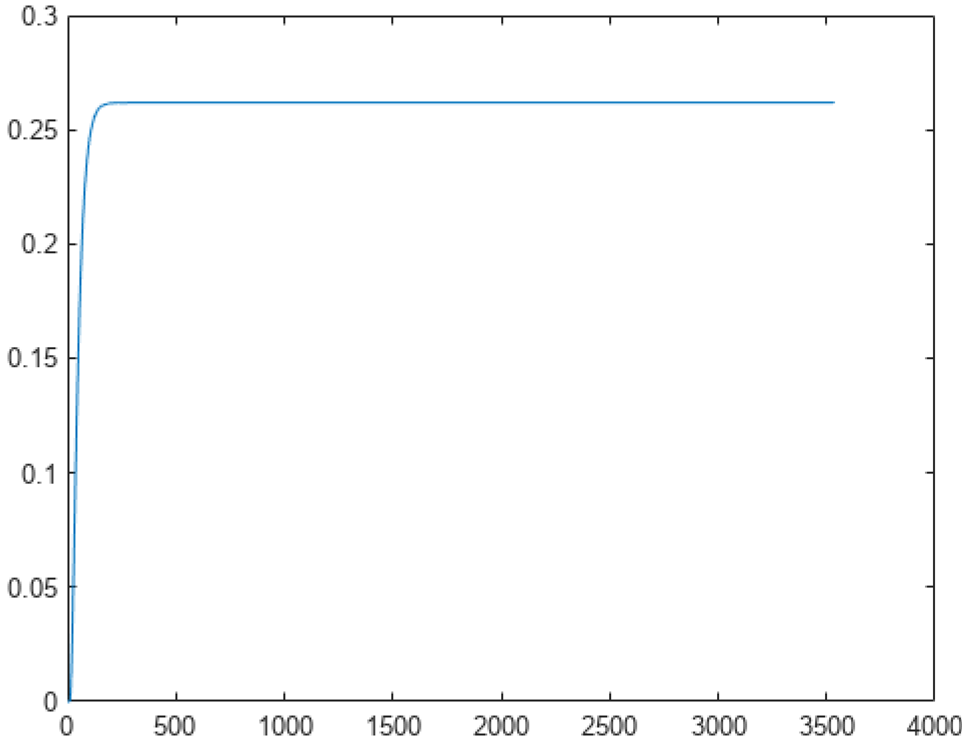
```
simOut = ode45(@(~,x)derivative(model,x,u,e), [0 3], s);
size(simOut.y)
```

```
ans = 1×2
```

```
          13          3536
```

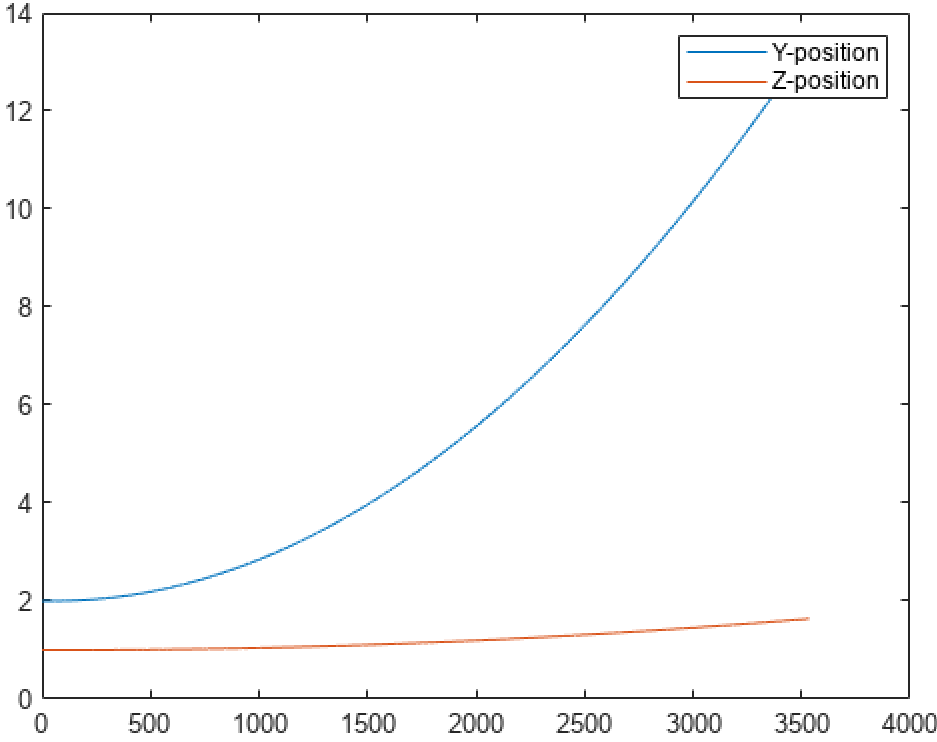
Plot the change in roll angle based on the simulation output. The roll angle (the X Euler angle) is the 9th row of the `simOut.y` output.

```
plot(simOut.y(9,:))
```



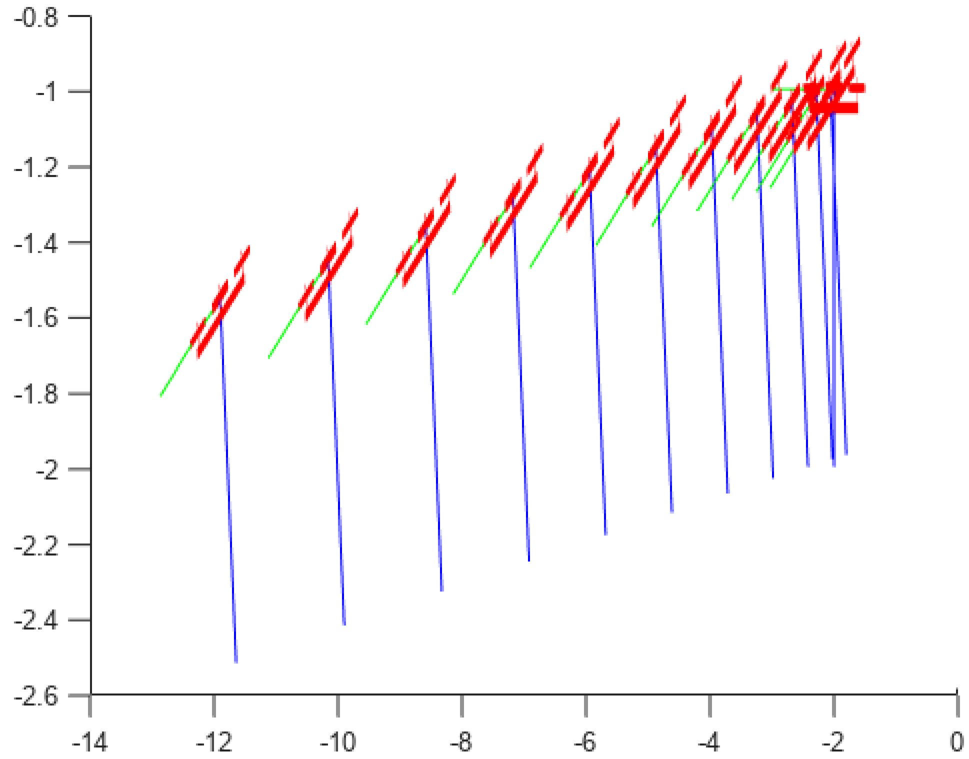
Plot the change in the Y and Z positions. With the specified thrust and roll angle, the multirotor should fly over and lose some altitude. A positive value for Z is expected as positive Z is down.

```
figure
plot(simOut.y(2,:));
hold on
plot(simOut.y(3,:));
legend('Y-position','Z-position')
hold off
```



You can also plot the multirotor trajectory using `plotTransforms`. Create the translation and rotation vectors from the simulated state. Downsample (every 300th element) and transpose the `simOut` elements, and convert the Euler angles to quaternions. Specify the mesh as the `multirotor.stl` file and the positive Z-direction as `"down"`. The displayed view shows the UAV translating in the Y-direction and losing altitude.

```
translations = simOut.y(1:3,1:300:end)'; % xyz position
rotations = eul2quat(simOut.y(7:9,1:300:end)'); % ZYX Euler
plotTransforms(translations,rotations,...
    'MeshFilePath','multirotor.stl','InertialZDirection',"down")
view([90.00 -0.60])
```



Simulate A Fixed-Wing Control Command

This example shows how to use the `fixedwing` guidance model to simulate the change in state of a UAV due to a command input.

Create the fixed-wing guidance model.

```
model = fixedwing;
```

Set the air speed of the vehicle by modifying the structure from the `state` function.

```
s = state(model);
s(4) = 5; % 5 m/s
```

Specify a control command, `u`, that maintains the air speed and gives a roll angle of `pi/12`.

```
u = control(model);
u.RollAngle = pi/12;
u.AirSpeed = 5;
```

Create a default environment without wind.

```
e = environment(model);
```

Compute the time derivative of the state given the current state, control command, and environment.

```
sdot = derivative(model,s,u,e);
```

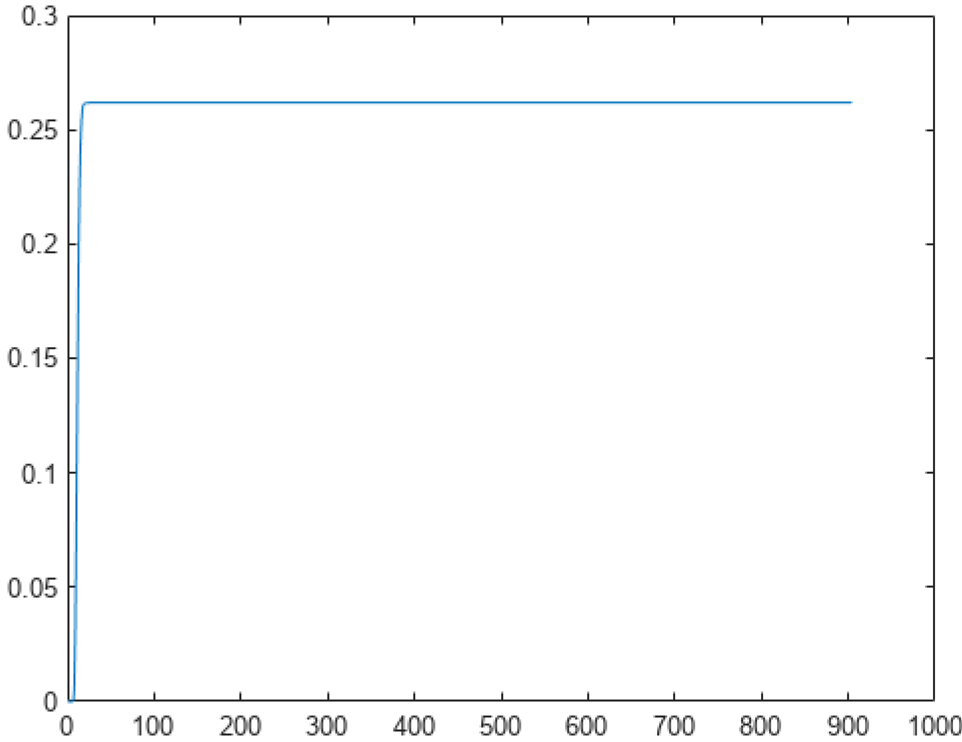
Simulate the UAV state using `ode45` integration. The `y` field outputs the fixed-wing UAV states based on this simulation.

```
simOut = ode45(@(~,x)derivative(model,x,u,e), [0 50], s);
size(simOut.y)
```

```
ans = 1×2

     8   904
```

Plot the change in roll angle based on the simulation output. The roll angle is the 7th row of the `simOut.y` output.

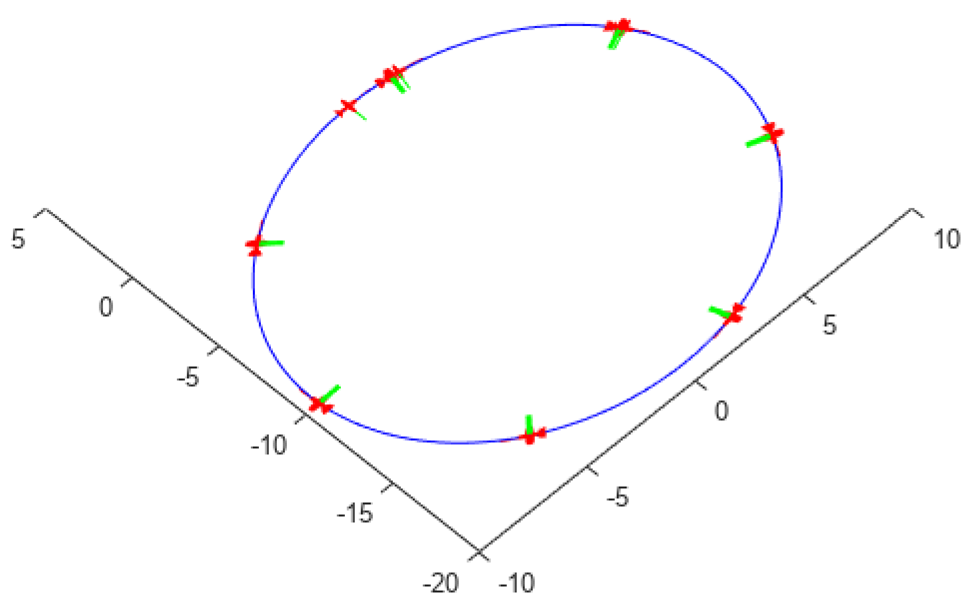
```
plot(simOut.y(7,:))
```



You can also plot the fixed-wing trajectory using `plotTransforms`. Create the translation and rotation vectors from the simulated state. Downsample (every 30th element) and transpose the `simOut` elements, and convert the Euler angles to quaternions. Specify the mesh as the `fixedwing.stl` file and the positive Z-direction as `"down"`. The displayed view shows the UAV making a constant turn based on the constant roll angle.

```
downsample = 1:30:size(simOut.y,2);
translations = simOut.y(1:3,downsample)'; % xyz-position
rotations = eul2quat([simOut.y(5,downsample)',simOut.y(6,downsample)',simOut.y(7,downsample)']);
```

```
plotTransforms(translations,rotations,...
    'MeshFilePath','fixedwing.stl','InertialZDirection',"down")
hold on
plot3(simOut.y(1,:),-simOut.y(2,:),simOut.y(3,:),'--b') % full path
xlim([-10.0 10.0])
ylim([-20.0 5.0])
zlim([-0.5 4.00])
view([-45 90])
hold off
```



Input Arguments

`uavGuidanceModel` — UAV guidance model
`fixedwing` object | `multirotor` object

UAV guidance model, specified as a `fixedwing` or `multirotor` object.

Output Arguments

`envStruct` — Environmental input parameters
structure

Environmental input parameters, returned as a structure.

For fixed-wing UAVs, the fields of the structure are `WindNorth`, `WindEast`,`WindDown`, and `Gravity`. Wind speeds are in meters per second and negative speeds point in the opposite direction. `Gravity` is in meters per second squared (default `9.81`).

For multirotor UAVs, the only element of the structure is `Gravity` (default `9.81`) in meters per second.

Version History

Introduced in R2018b

Extended Capabilities

C/C++ Code Generation

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

See Also

Functions

`ode45` | `control` | `derivative` | `state` | `plotTransforms`

Objects

`fixedwing` | `multirotor`

Blocks

Waypoint Follower | UAV Guidance Model

Topics

"Approximate High-Fidelity UAV Model with UAV Guidance Model Block"
"Tuning Waypoint Follower for Fixed-Wing UAV"

state

UAV state vector

Syntax

`stateVec = state(uavGuidanceModel)`

Description

`stateVec = state(uavGuidanceModel)` returns a state vector for the specified UAV guidance model. The vector is always filled with zeros. Use this function to ensure you have the proper size for your state vector. Use the state vector as an input to the `derivative` function or when simulating the UAV using `ode45`.

Examples

Simulate A Multirotor Control Command

This example shows how to use the `multirotor` guidance model to simulate the change in state of a UAV due to a command input.

Create the multirotor guidance model.

```
model = multirotor;
```

Create a state structure. Specify the location in world coordinates.

```
s = state(model);
s(1:3) = [3;2;1];
```

Specify a control command, `u`, that specified the roll and thrust of the multirotor.

```
u = control(model);
u.Roll = pi/12;
u.Thrust = 1;
```

Create a default environment without wind.

```
e = environment(model);
```

Compute the time derivative of the state given the current state, control command, and environment.

```
sdot = derivative(model,s,u,e);
```

Simulate the UAV state using `ode45` integration. The `y` field outputs the multirotor UAV states as a 13-by-*n* matrix.

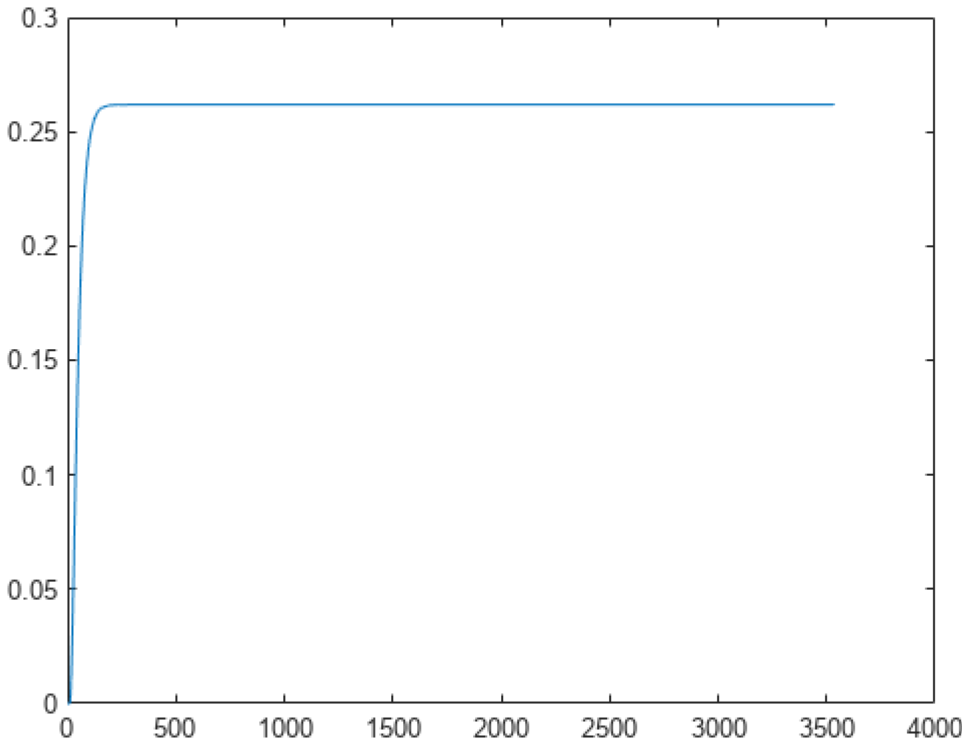
```
simOut = ode45(@(~,x)derivative(model,x,u,e), [0 3], s);
size(simOut.y)
```

```
ans = 1×2
```

```
       13          3536
```

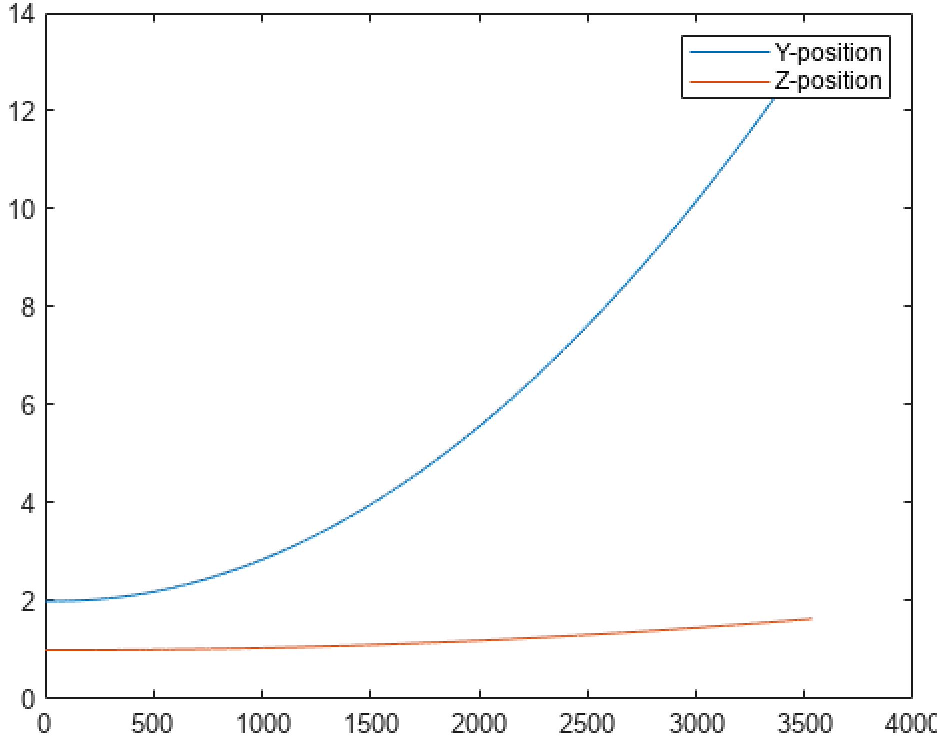
Plot the change in roll angle based on the simulation output. The roll angle (the X Euler angle) is the 9th row of the `simOut.y` output.

```
plot(simOut.y(9,:))
```



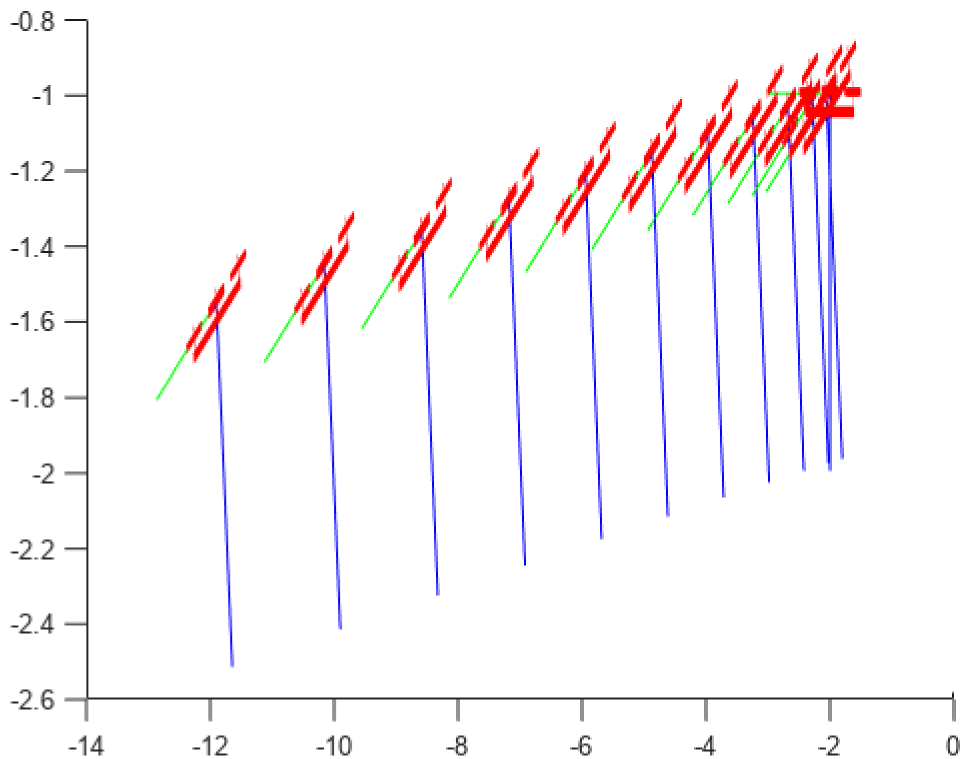
Plot the change in the Y and Z positions. With the specified thrust and roll angle, the multirotor should fly over and lose some altitude. A positive value for Z is expected as positive Z is down.

```
figure
plot(simOut.y(2,:));
hold on
plot(simOut.y(3,:));
legend('Y-position','Z-position')
hold off
```



You can also plot the multirotor trajectory using `plotTransforms`. Create the translation and rotation vectors from the simulated state. Downsample (every 300th element) and transpose the `simOut` elements, and convert the Euler angles to quaternions. Specify the mesh as the `multirotor.stl` file and the positive Z-direction as `"down"`. The displayed view shows the UAV translating in the Y-direction and losing altitude.

```
translations = simOut.y(1:3,1:300:end)'; % xyz position
rotations = eul2quat(simOut.y(7:9,1:300:end)'); % ZYX Euler
plotTransforms(translations,rotations,...
    'MeshFilePath','multirotor.stl','InertialZDirection',"down")
view([90.00 -0.60])
```



Simulate A Fixed-Wing Control Command

This example shows how to use the `fixedwing` guidance model to simulate the change in state of a UAV due to a command input.

Create the fixed-wing guidance model.

```
model = fixedwing;
```

Set the air speed of the vehicle by modifying the structure from the `state` function.

```
s = state(model);
s(4) = 5; % 5 m/s
```

Specify a control command, `u`, that maintains the air speed and gives a roll angle of `pi/12`.

```
u = control(model);
u.RollAngle = pi/12;
u.AirSpeed = 5;
```

Create a default environment without wind.

```
e = environment(model);
```

Compute the time derivative of the state given the current state, control command, and environment.

```
sdot = derivative(model,s,u,e);
```

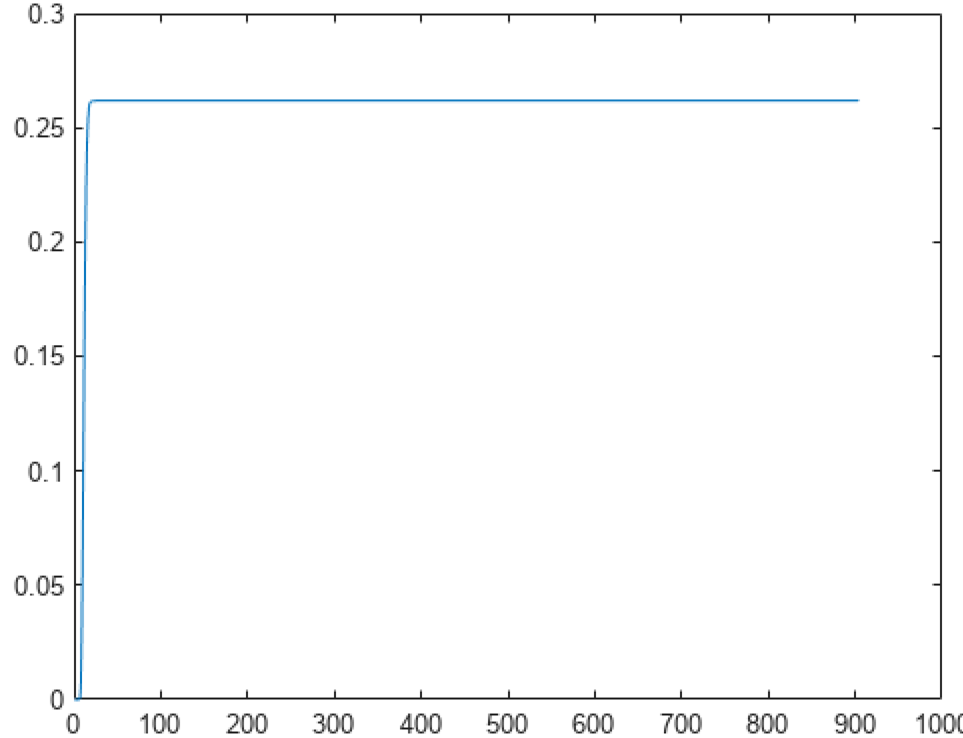
Simulate the UAV state using `ode45` integration. The `y` field outputs the fixed-wing UAV states based on this simulation.

```
simOut = ode45(@(~,x)derivative(model,x,u,e), [0 50], s);
size(simOut.y)
```

```
ans = 1×2

     8   904
```

Plot the change in roll angle based on the simulation output. The roll angle is the 7th row of the `simOut.y` output.

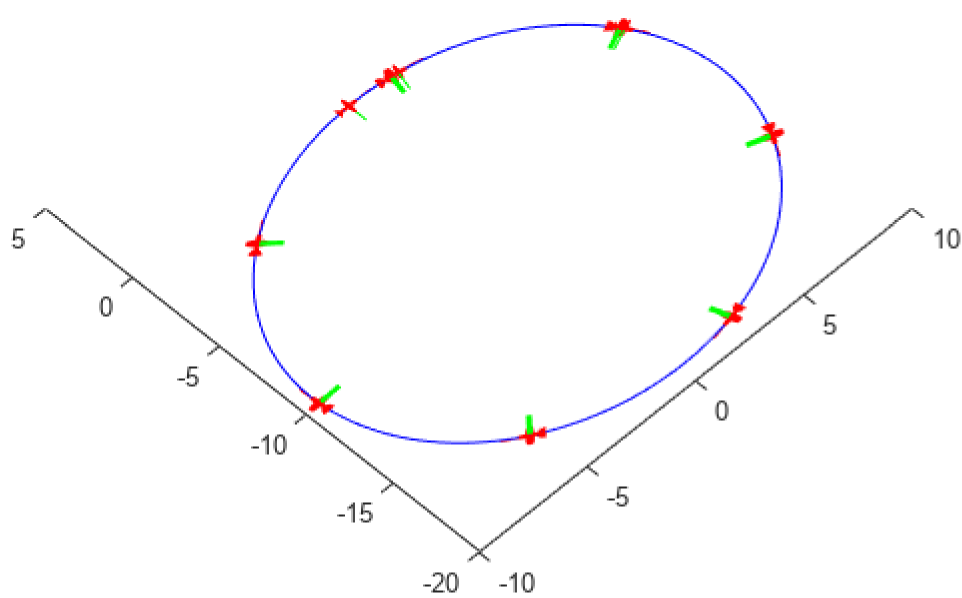
```
plot(simOut.y(7,:))
```



You can also plot the fixed-wing trajectory using `plotTransforms`. Create the translation and rotation vectors from the simulated state. Downsample (every 30th element) and transpose the `simOut` elements, and convert the Euler angles to quaternions. Specify the mesh as the `fixedwing.stl` file and the positive Z-direction as `"down"`. The displayed view shows the UAV making a constant turn based on the constant roll angle.

```
downsample = 1:30:size(simOut.y,2);
translations = simOut.y(1:3,downsample)'; % xyz-position
rotations = eul2quat([simOut.y(5,downsample)',simOut.y(6,downsample)',simOut.y(7,downsample)']);
```

```
plotTransforms(translations,rotations,...
    'MeshFilePath','fixedwing.stl','InertialZDirection',"down")
hold on
plot3(simOut.y(1,:),-simOut.y(2,:),simOut.y(3,:),'--b') % full path
xlim([-10.0 10.0])
ylim([-20.0 5.0])
zlim([-0.5 4.00])
view([-45 90])
hold off
```



Input Arguments

uavGuidanceModel — UAV guidance model
fixedwing object | multirotor object

UAV guidance model, specified as a `fixedwing` or `multirotor` object.

Output Arguments

stateVec — State vector
zeros(7,1) | zeros(13,1)

State vector, returned as a seven-element or thirteen-element vector. The vector is always filled with zeros. Use this function to ensure you have the proper size for your state vector.

For fixed-wing UAVs, the state is an eight-element vector:

- **North** - Position in north direction in meters.
- **East** - Position in east direction in meters.
- **Height** - Height above ground in meters.
- **AirSpeed** - Speed relative to wind in meters per second.
- **HeadingAngle** - Angle between ground velocity and north direction in radians.
- **FlightPathAngle** - Angle between ground velocity and north-east plane in radians.
- **RollAngle** - Angle of rotation along body *x*-axis in radians.
- **RollAngleRate** - Angular velocity of rotation along body *x*-axis in radians per second.

For multirotor UAVs, the state is a thirteen-element vector in this order:

- **World Position** - `[x y z]` in meters.
- **World Velocity** - `[vx vy vz]` in meters per second.
- **Euler Angles (ZYX)** - `[psi theta phi]` in radians.
- **Body Angular Rates** - `[p q r]` in radians per second.
- **Thrust** - `F` in Newtons.

Version History

Introduced in R2018b

Extended Capabilities

C/C++ Code Generation

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

See Also

Functions

`ode45` | `control` | `derivative` | `environment` | `plotTransforms`

Objects

`fixedwing` | `multirotor`

Blocks

Waypoint Follower | UAV Guidance Model

Topics

"Approximate High-Fidelity UAV Model with UAV Guidance Model Block"
"Tuning Waypoint Follower for Fixed-Wing UAV"

checkSignal

Check mapped signal

Syntax

```
[summary,errorIndex] = checkSignal(mapper,logData)
[summary,errorIndex] = checkSignal( ___ ,Name,Value)
```

Description

`[summary,errorIndex] = checkSignal(mapper,logData)` checks mapped signals stored in `mapper` using the imported flight log `logData`. Import your flight log using `mavlinktlog` or `ulogreader`.

`[summary,errorIndex] = checkSignal( ___ ,Name,Value)` specifies options using one or more name-value pair arguments in addition to the input arguments in the previous syntax. For example, `'Preview',"on"` shows a preview of the extracted signal.

Examples

Check Mapped Signals Using Flight Log Data

Create a `flightLogSignalMapping` object for the ULOG file.

```
mapping = flightLogSignalMapping("ulog");
```

Load the ULOG file. Specify the relative path of the file.

```
logData = ulogreader("flight.ulg");
```

Check all the mapped signals stored in the `flightLogSignalMapping` object using the imported flight log.

```
[summary,errorIndex] = checkSignal(mapping,logData)
```

```
--------------------------------------------
SignalName: Accel
Pass
--------------------------------------------
SignalName: Gyro
Pass
--------------------------------------------
SignalName: Mag
Pass
--------------------------------------------
SignalName: Barometer
Pass
--------------------------------------------
SignalName: GPS
Pass
--------------------------------------------
```

```
SignalName: LocalNED
Pass
--------------------------------------------
SignalName: LocalENU
Pass
--------------------------------------------
SignalName: LocalNEDVel
Pass
--------------------------------------------
SignalName: LocalENUVel
Pass
--------------------------------------------
SignalName: LocalNEDTarget
Unable to extract vehicle local position value from log data
--------------------------------------------
SignalName: LocalENUTarget
Unable to extract vehicle local position value from log data
--------------------------------------------
SignalName: LocalNEDVelTarget
Unable to extract vehicle local velocity value from log data
--------------------------------------------
SignalName: LocalENUVelTarget
Unable to extract vehicle local velocity value from log data
--------------------------------------------
SignalName: AttitudeEuler
Pass
--------------------------------------------
SignalName: AttitudeRate
Unable to extract attitude rate value from log data
--------------------------------------------
SignalName: AttitudeTargetEuler
Pass
--------------------------------------------
SignalName: Airspeed
Pass
--------------------------------------------
SignalName: Battery
Pass

summary=1×18 struct array with fields:
    SignalName
    Result


errorIndex = 1×5

    10    11    12    13    15
```

Check specific set of signals.

```
[summary,errorIndex] = checkSignal(mapping,logData,"Signal",["Accel" "Gyro"]);
```

```
--------------------------------------------
SignalName: Accel
Pass
--------------------------------------------
SignalName: Gyro
Pass
```

Input Arguments

`mapper` — Flight log signal mapping object
`flightLogSignalMapping` object

Flight log signal mapping object, specified as a `flightLogSignalMapping` object.

`logData` — Data from flight log
table | `ulogreader` object | `mavlinktlog` object

Data from the flight log, specified as a table, `ulogreader` object, `mavlinktlog` object, or other custom formats.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `'Preview',"on"` shows a preview of the extracted signal.

`Signal` — Signal names to check
string array | cell array of character vectors

Signal names to check, specified as the comma-separated pair consisting of `'Signal'` and a string array or cell array of character vectors.

Example: `["Accel","Gyro"]`

Data Types: `char` | `string`

`Preview` — Preview of extracted signals in plot
`"off"` (default) | `"on"`

Preview of extracted signals in a plot, specified as the comma-separated pair consisting of `'Preview'` and `"on"` or `"off"`. Specify `"on"` to display plots of the signals in the order the mapped signals are stored. Press any key to display the next signal. Press *Q* to close the figure.

Example: `'Preview',"on"`

Data Types: `char` | `string`

Output Arguments

`summary` — Summary of signal extraction
structure

Summary of signal extraction, returned as a structure with these fields:

- `SignalName` –– Name of the mapped signals as a string
- `Result` –– Status of signal extraction as a character vector

`errorIndex` — Indices of unsuccessful signal extraction
vector of positive integers

Indices of unsuccessful signal extraction, returned as a vector of positive integers.

Version History

Introduced in R2021a

See Also

Objects
`flightLogSignalMapping` | `ulogreader` | `mavlinktlog`

Functions
`copy` | `extract` | `info` | `mapSignal` | `show` | `updatePlot`

copy

Create deep copy of flight log signal mapping object

Syntax

`mapperCopy = copy(mapper)`

Description

`mapperCopy = copy(mapper)` creates a deep copy of the `flightLogSignalMapping` object with the same properties.

Input Arguments

`mapper` — Flight log signal mapping object
`flightLogSignalMapping` object

Flight log signal mapping object, specified as a `flightLogSignalMapping` object.

Output Arguments

`mapperCopy` — Copy of flight log signal mapping object
`flightLogSignalMapping` object

Copy of flight log signal mapping object, returned as a `flightLogSignalMapping` object with the same properties.

Version History

Introduced in R2021a

See Also

Objects
`flightLogSignalMapping`

Functions
`checkSignal` | `extract` | `info` | `mapSignal` | `show` | `updatePlot`

extract

Extract UAV flight log signals as timetables

Syntax

```
signals = extract(mapper,data,signalNames)
signals = extract(mapper,data,signalNames,timeStart)
signals = extract(mapper,data,signalNames,timeStart,timeEnd)
```

Description

`signals = extract(mapper,data,signalNames)` obtains signals with the given names `signalNames` as timetables from imported flight log, `data`. Import your flight log using `mavlinktlog` or `ulogreader`.

`signals = extract(mapper,data,signalNames,timeStart)` obtains signals with the given names with time stamps greater than or equal to `timeStart`.

`signals = extract(mapper,data,signalNames,timeStart,timeEnd)` obtains signals with the given names with time stamps within the interval `[timeStart timeEnd]` inclusive.

Input Arguments

`mapper` — Flight log signal mapping
`flightLogSignalMapping` object

Flight log signal mapping object, specified as a `flightLogSignalMapping` object.

`data` — Flight log data
table

Flight log data, specified as a table.

`signalNames` — Signal names to extract from log
string array

Signal names to extract from log, specified as a string array.

`timeStart` — Initial time stamp for signal
`duration` object

Initial time stamp for signal to extract, specified as a `duration` object.

`timeEnd` — Final time stamp for signal
`duration` object

Final time stamp for signal to extract, specified as a `duration` object.

Output Arguments

`signals` — Extracted signals
cell array

Extracted signals, returned as a cell array. Each signal name maps to an element of the cell array.

Version History

Introduced in R2020b

See Also

`flightLogSignalMapping` | `mavlinktlog` | `info` | `mapSignal` | `show` | `updatePlot`

info

Signal mapping and plot information for UAV log signal mapping

Syntax

```
signalTable = info(mapper,"Signal")
signalTable = info(mapper,"Signal",signalNames)
plotTable = info(mapper,"Plot")
signalTable = info(mapper,"Plot",plotNames)
```

Description

`signalTable = info(mapper,"Signal")` generates a table of information for the Predefined Signals on page 2-63 available and the signals mapped in the flight log signal mapping object. The table contains a list of signal names, field names, units, and whether the signal has a value function mapped to it (`IsMapped` column).

`signalTable = info(mapper,"Signal",signalNames)` generates the signal table for the specified signal names.

`plotTable = info(mapper,"Plot")` generates a table of information for the Predefined Plots on page 2-64 and custom plots available in the flight log signal mapping object. The table contains plots names, required signals, missing signals, and whether the plot is ready to plot.

`signalTable = info(mapper,"Plot",plotNames)` generates the plot table for the specified plot names.

Input Arguments

`mapper` — Flight log signal mapping
`flightLogSignalMapping` object

Flight log signal mapping object, specified as a `flightLogSignalMapping` object.

`signalNames` — Signal names
string array

Signal names, specified as a string array.

`plotNames` — Plot names
string array

Plot names, specified as a string array.

Output Arguments

`signalTable` — Table of available signals
table

Table of available signals, returned as a table. This table includes preconfigured signals and any signals you added to the flight log signal mapping object using `mapSignal`. The table has these fields:

- `SignalName` –– String scalar of the name of the signal.
- `IsMapped` –– Logical indicating if the signal is properly mapped. To update signal mapping, see `mapSignal`.
- `SignalFields` –– String scalar that lists the fields of the signal.
- `FieldUnits` –– String scalar that lists the units of each field.

`plotTable` — Table of available plots
table

Table of available plots, returned as a table. This table includes preconfigured plots and any plots you added to the flight log signal mapping object using `updatePlot`. The table has these fields:

- `PlotName` –– String scalar of the name of the plot.
- `ReadyToPlot` –– Logical indicating if the plot is configured properly. To update the plot, see `updatePlot`.
- `MissingSignals` –– String scalar that lists the signals that need to be mapped using `mapSignal`.
- `RequiredSignals` –– String scalar that lists all required signals for a specific plot name.

More About

Predefined Signals

A set of predefined signals and plots are configured in the `flightLogSignalMapping` object. Depending on your log file type, you can map specific signals to the provided signal names using `mapSignal`. You can also call `info` to view the table for your log type and see whether you have already mapped a signal to that plot type.

Specify the `SignalName` as the input to `mapSignal`. Signals with the format `SignalName#` support mapping multiple signals of the same type. Replace # with incremental integers for each signal name when calling `mapSignal`.

The predefined signals have specific names and required fields when mapping the signal.

Predefined Signals

Signal Name	Description	Fields	Unit
Accel#	Raw magnetometer reading from IMU sensor	[ax ay az]	m/s^2
Airspeed#	Airspeed reading of pressure differential, indicated air speed, and temperature	[PressDiff, AirSpeed, Temp]	Pa, r
AttitudeEuler	Attitude of UAV in Euler (ZYX) form	[Roll, Pitch, Yaw]	radi
AttitudeRate	Angular velocity along each body axis	[xRotRate, yRotRate, zRotRate]	rad/
AttitudeTargetEuler	Target attitude of UAV in Euler (ZYX) form	[TargetRoll, TargetPitch, TargetYaw]	radi
Barometer#	Barometer readings for absolute pressure, relative pressure, and temperature	[PressAbs, PressAltitude, Temp]	Pa, r
Battery	Voltage readings for battery and remaining battery capacity (%)	[Volt1,Volt2, ... Volt16, RemainingCapacity	V, %
GPS#	GPS readings for latitude, longitude, altitude, ground speed, course angle, and number of satellites visible	[lat, long, alt, groundspeed, courseAngle, satellites]	degr degr
Gyro#	Raw body angular velocity readings from IMU sensor	[GyroX, GyroY, GyroZ]	rad/
LocalNED	Local NED coordinates estimated by the UAV	[xNED, yNED, zNED]	met
LocalNEDTarget	Target location in local NED coordinates	[xTarget, yTarget, zTarget]	met
LocalNEDVel	Local NED velocity estimated by the UAV	[vx vy vz]	m/s
LocalNEDVelTarget	Target velocity in NED in local NED	[vxTarget, vyTarget, vzTarget]	m/s
Mag#	Raw magnetometer reading from IMU sensor	[x y z]	Gs

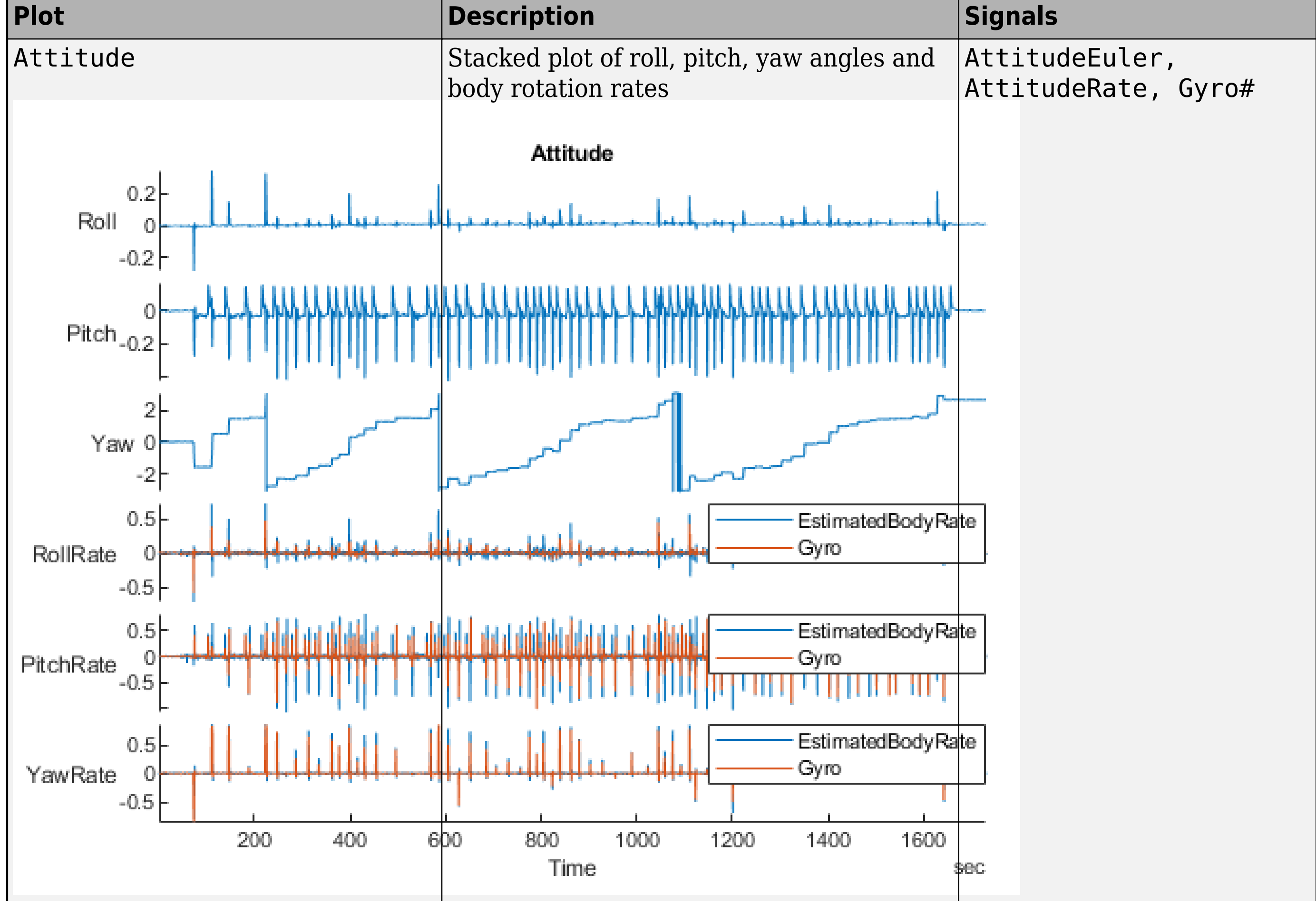
Predefined Plots

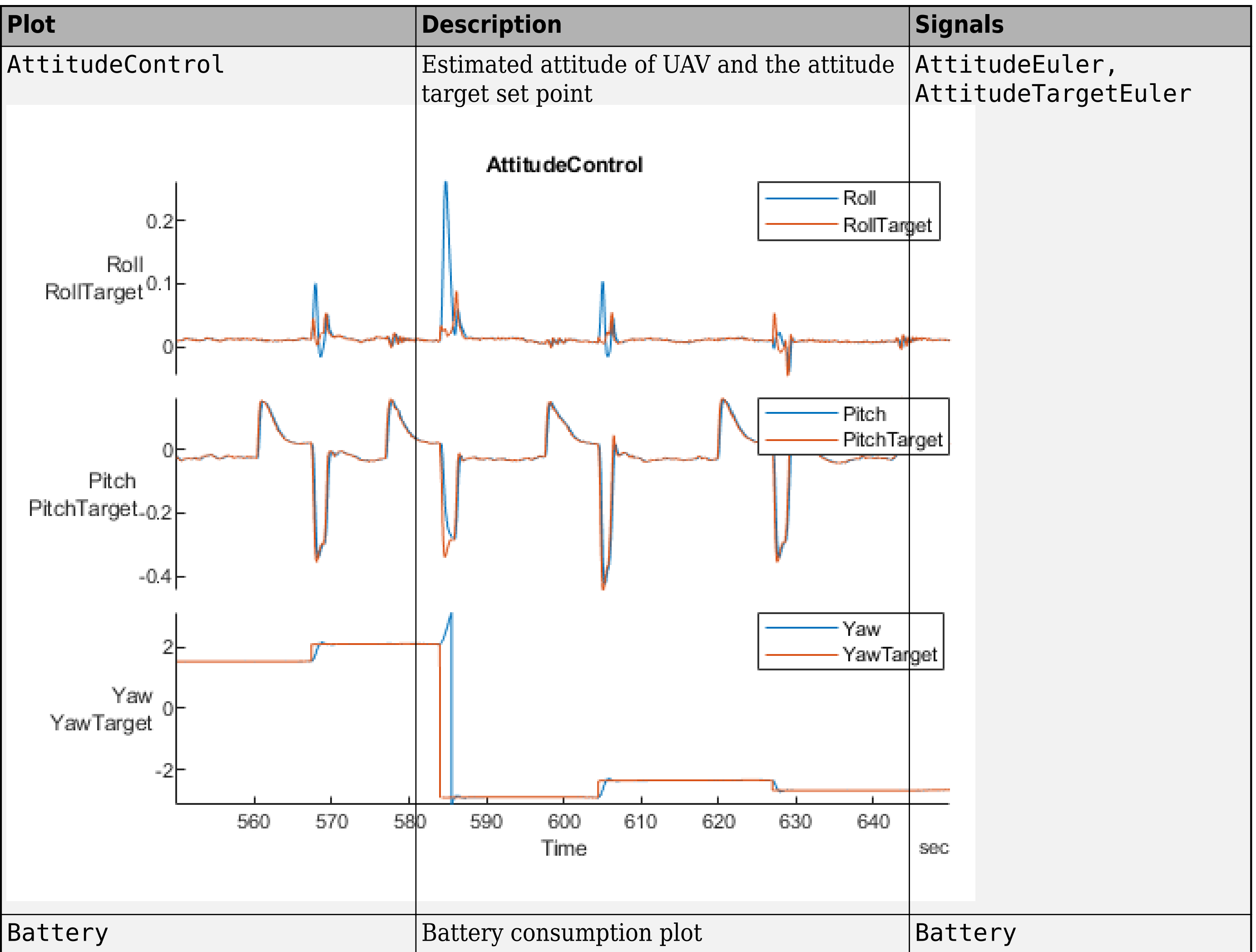
After mapping signals to the list of predefined signals using `mapSignal`, specific plots are made available when calling `show`. To view a list of available plots and their associated signals for your specific object, call `info(mapper,"Plot")`. If you want to define custom plots based on signals, use `updatePlot`.

Each predefined plot has a set of required signals that must be mapped.

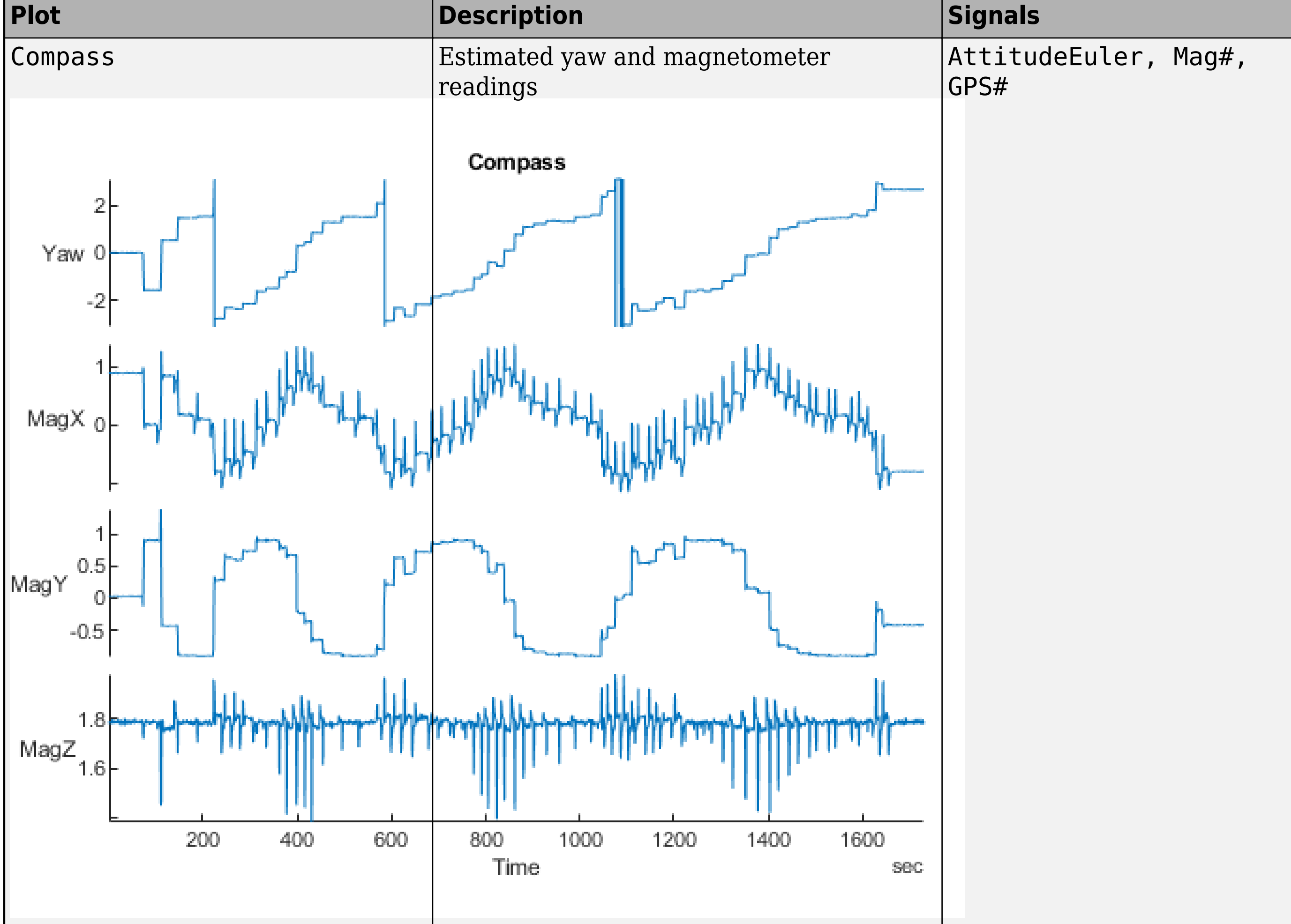
Predefined Plots

Plot	Description	Signals
`Attitude`	Stacked plot of roll, pitch, yaw angles and body rotation rates	`AttitudeEuler`, `AttitudeRate`, `Gyro#`

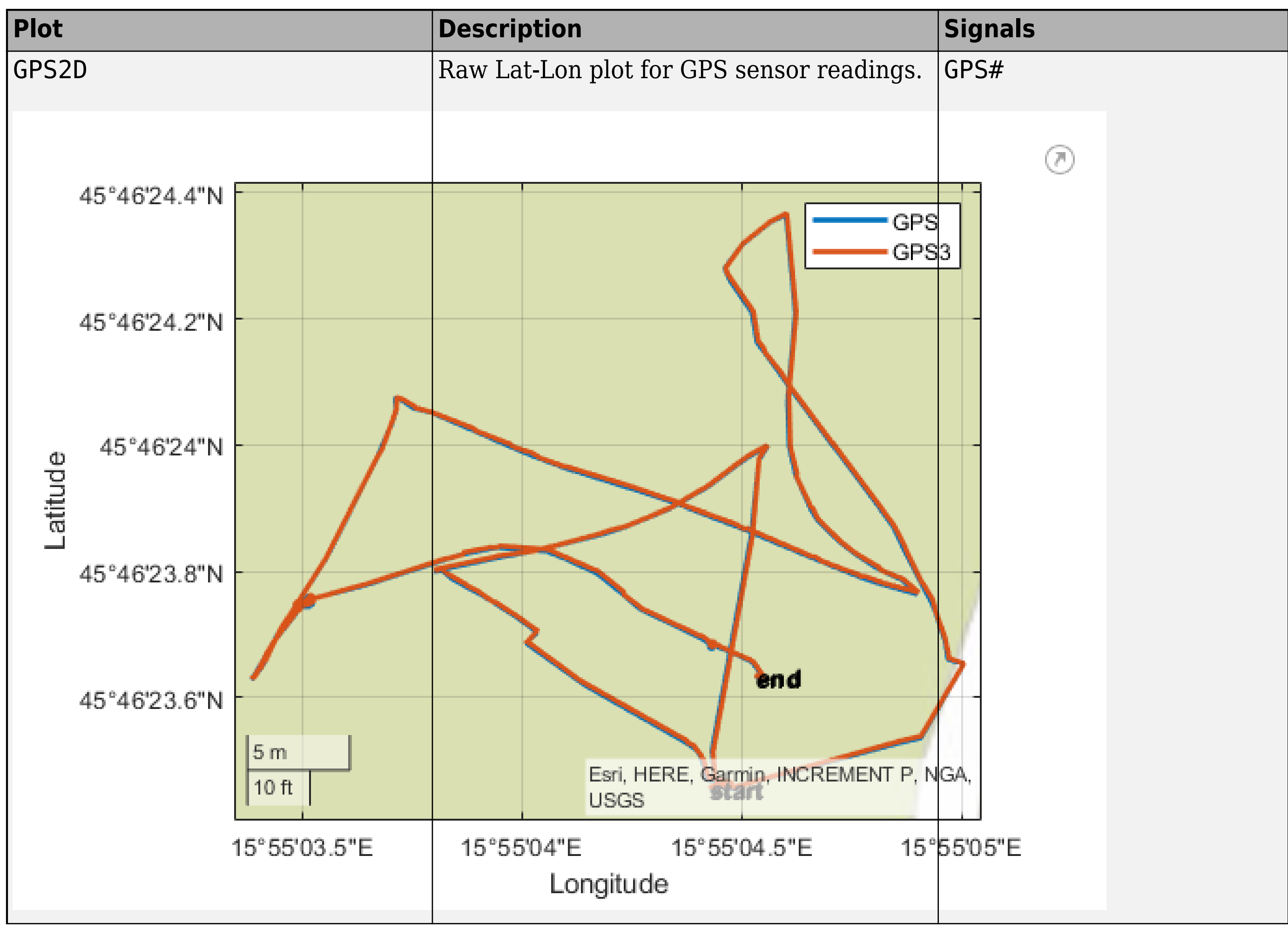


Plot	Description	Signals
AttitudeControl  	Estimated attitude of UAV and the attitude target set point	AttitudeEuler, AttitudeTargetEuler
Battery	Battery consumption plot	Battery

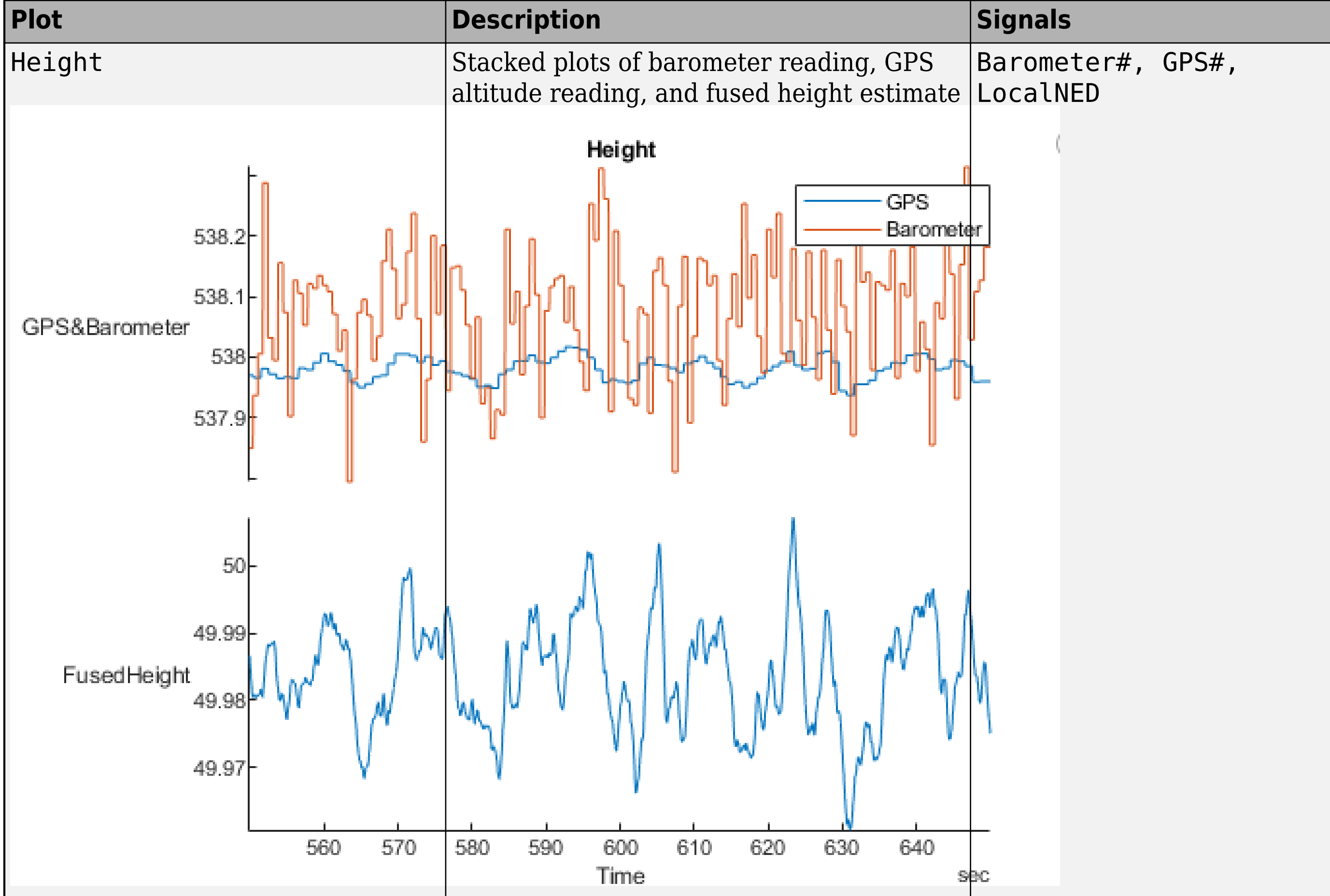
Plot	Description	Signals
Compass	Estimated yaw and magnetometer readings	AttitudeEuler, Mag#, GPS#



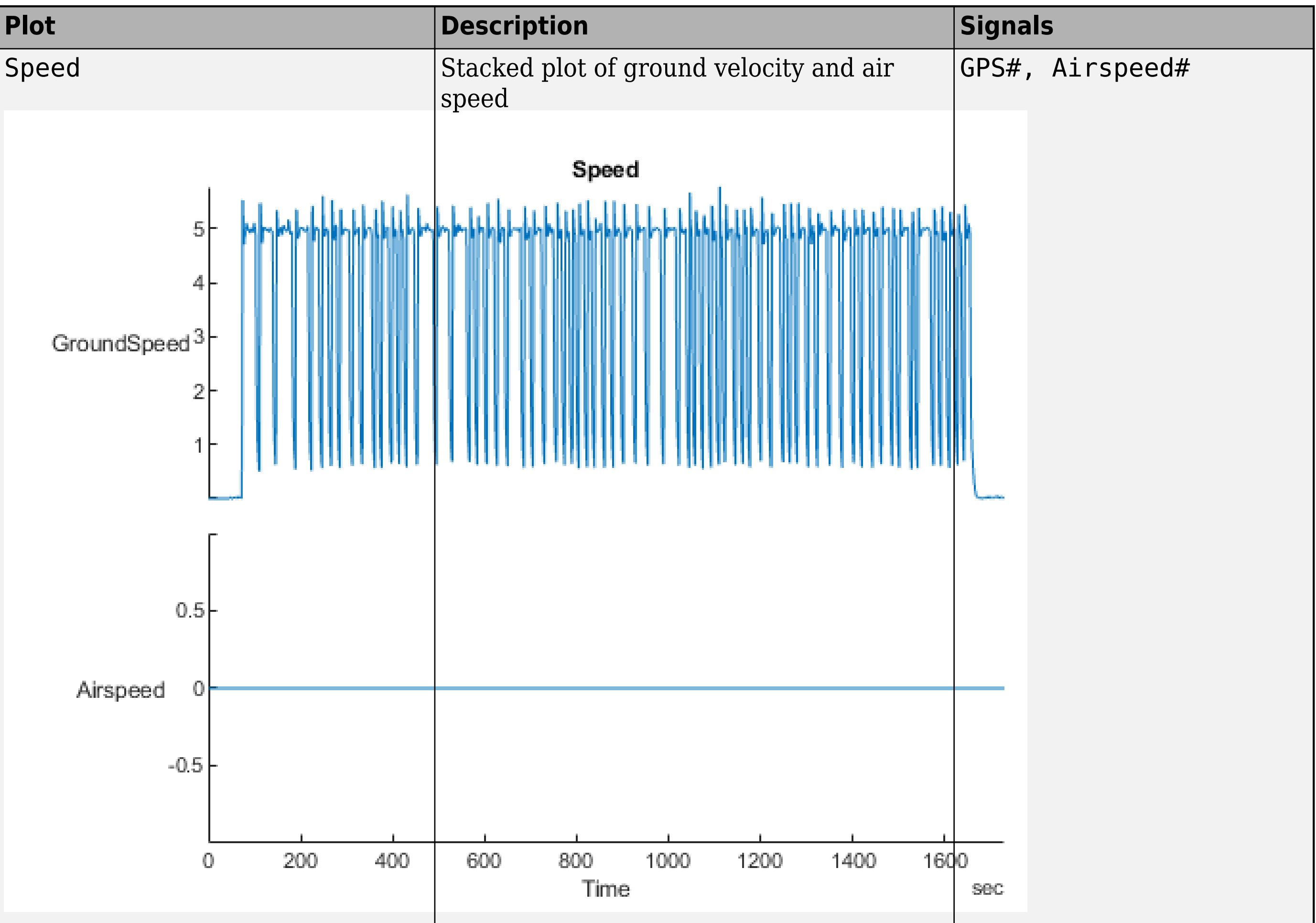
Plot	Description	Signals
GPS2D	Raw Lat-Lon plot for GPS sensor readings.	GPS#



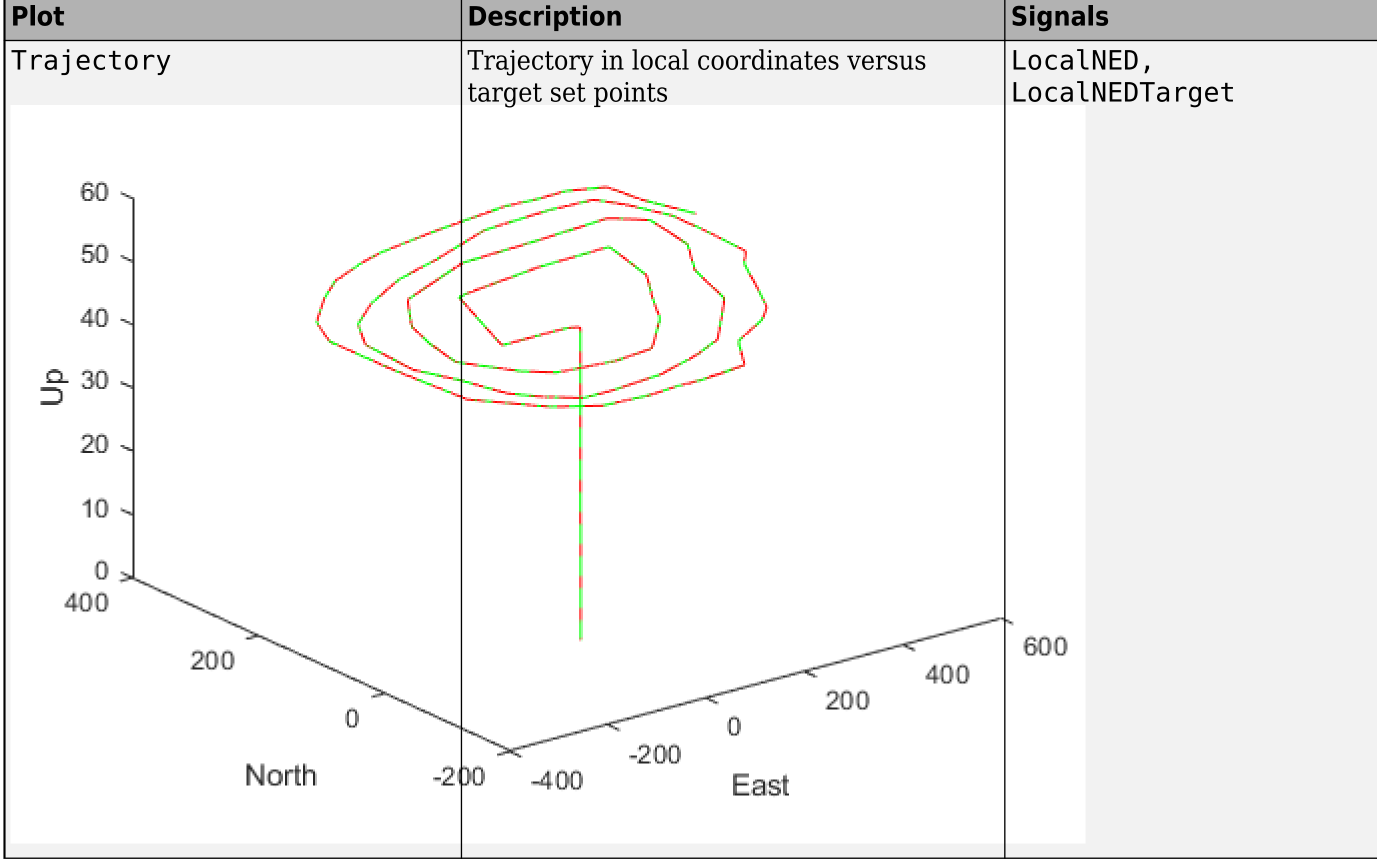
Plot	Description	Signals
`Height`	Stacked plots of barometer reading, GPS altitude reading, and fused height estimate	`Barometer#, GPS#, LocalNED`



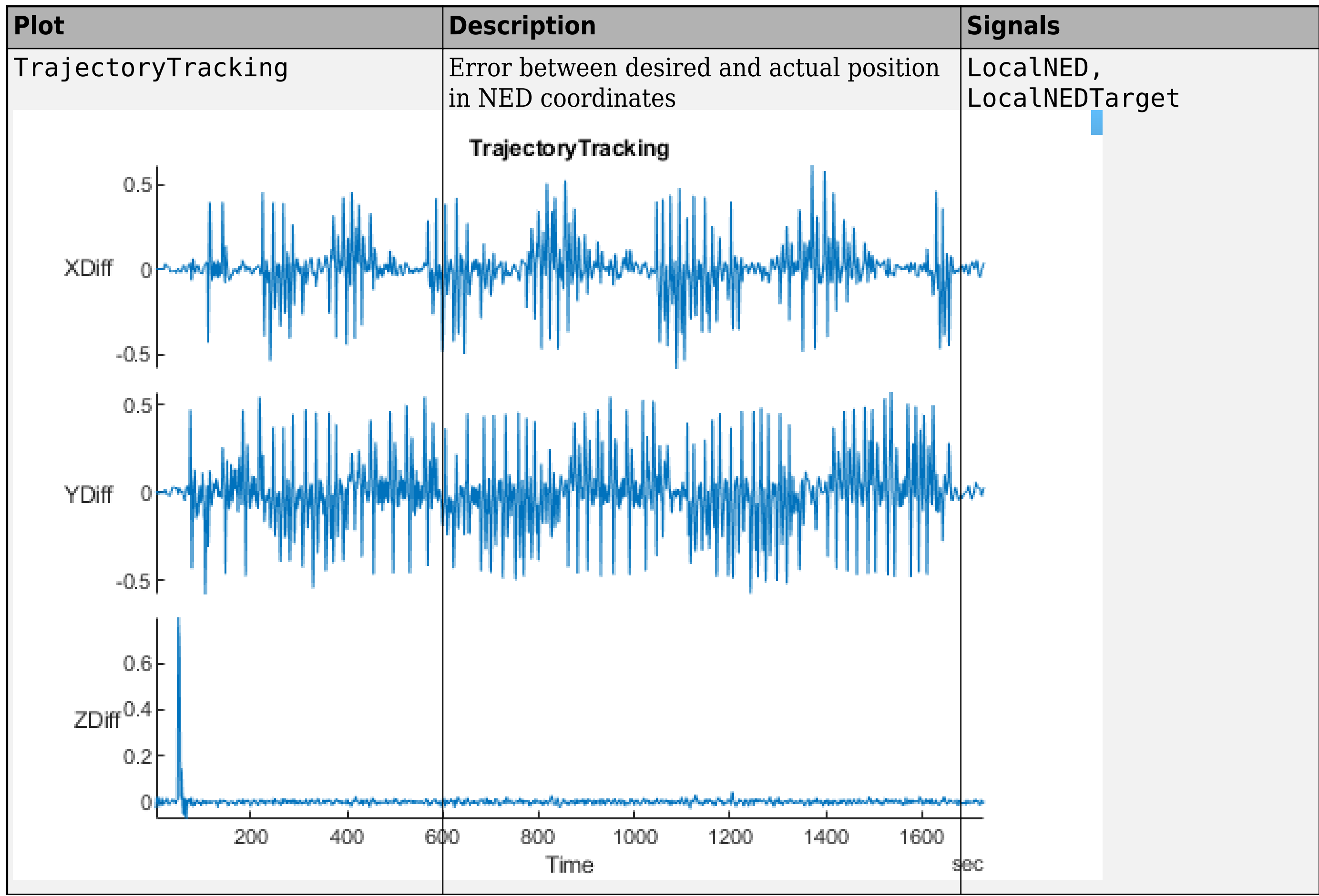
Plot	Description	Signals
Speed	Stacked plot of ground velocity and air speed	GPS#, Airspeed#



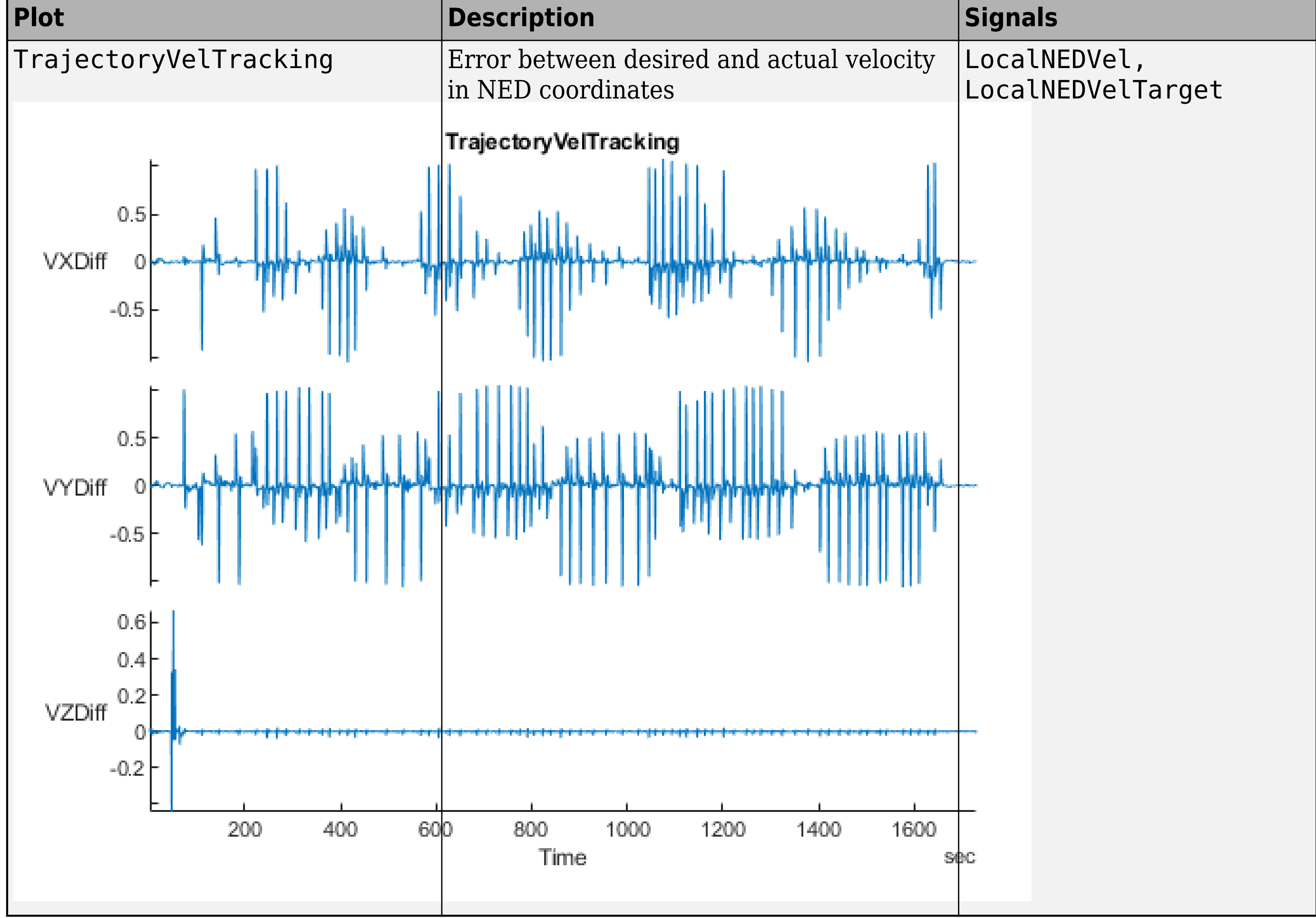
Plot	Description	Signals
Trajectory	Trajectory in local coordinates versus target set points	LocalNED, LocalNEDTarget



Plot	Description	Signals
TrajectoryTracking	Error between desired and actual position in NED coordinates	LocalNED, LocalNEDTarget



Plot	Description	Signals
`TrajectoryVelTracking`	Error between desired and actual velocity in NED coordinates	`LocalNEDVel`, `LocalNEDVelTarget`



Version History

Introduced in R2020b

See Also

Objects
`flightLogSignalMapping`

Functions
`checkSignal` | `copy` | `extract` | `mapSignal` | `show` | `updatePlot`

mapSignal

Map UAV flight log signal

Syntax

```
mapSignal(mapper,signalName,timeFunc,valueFunc)
mapSignal(mapper,signalName,timeFunc,valueFunc,varNames)
mapSignal(mapper,signalName,timeFunc,valueFunc,varNames,varUnits)
```

Description

`mapSignal(mapper,signalName,timeFunc,valueFunc)` maps the signal with name `signalName` to a pair of function handles, `timeFunc` and `valueFunc`. These functions define the time stamps and values of signals from a flight log file, which can be imported using `mavlinktlog` or `ulogreader`. For a list of preconfigured signals and plots, see Predefined Signals on page 2-75 and Predefined Plots on page 2-76.

`mapSignal(mapper,signalName,timeFunc,valueFunc,varNames)` maps the signal with name `signalName` and specifies the variable names, `varName`, for the columns of a matrix generated from `valueFunc`.

`mapSignal(mapper,signalName,timeFunc,valueFunc,varNames,varUnits)` maps the signal with name `signalName` and specifies the units, `varUnits` for `varName`.

Input Arguments

`mapper` — Flight log signal mapping
`flightLogSignalMapping` object

Flight log signal mapping object, specified as a `flightLogSignalMapping` object.

`signalName` — Signal name to map data
string scalar | character vector

Signal name to map data, specified as a string scalar or character vector.

Example: `"Gyro"`

Data Types: `char` | `string`

`timeFunc` — Timestamps for signal
function handle

Timestamps for signal values , specified as a function handle. Typically, this function handle extracts time data from a flight log, which can be imported using `mavlinktlog` or `ulogreader`.

Example: `@(x)x.Gyro.Time`

Data Types: `function_handle`

`valueFunc` — Values for signal
function handle

Values for signal, specified as a function handle. Typically, this function handle extracts signal data from a flight log, which can be imported using `mavlinktlog` or `ulogreader`.

Example: `@(x)x.Gyro.Value`

Data Types: `function_handle`

`varNames` — Variable names for a matrix of values
string array | cell array of character vectors

Variable names for a matrix of values, specified as a string array or cell array of character vectors. Each element corresponds to a column in the matrix of values generated from `valueFunc`.

Example: `["xPos" "yPos" "zPos"]`

Data Types: `char` | `string`

`varUnits` — Variable units for a matrix of values
string array | cell array of character vectors

Variable units for a matrix of values, specified as a string array or cell array of character vectors. Each element corresponds to an element in `varNames`.

Example: `["m" "m" "rad"]`

Data Types: `char` | `string`

More About

Predefined Signals

A set of predefined signals and plots are configured in the `flightLogSignalMapping` object. Depending on your log file type, you can map specific signals to the provided signal names using `mapSignal`. You can also call `info` to view the table for your log type and see whether you have already mapped a signal to that plot type.

Specify the `SignalName` as the input to `mapSignal`. Signals with the format `SignalName#` support mapping multiple signals of the same type. Replace `#` with incremental integers for each signal name when calling `mapSignal`.

The predefined signals have specific names and required fields when mapping the signal.

Predefined Signals

Signal Name	Description	Fields	Unit
Accel#	Raw magnetometer reading from IMU sensor	[ax ay az]	m/s^2
Airspeed#	Airspeed reading of pressure differential, indicated air speed, and temperature	[PressDiff, AirSpeed, Temp]	Pa, r
AttitudeEuler	Attitude of UAV in Euler (ZYX) form	[Roll, Pitch, Yaw]	radi
AttitudeRate	Angular velocity along each body axis	[xRotRate, yRotRate, zRotRate]	rad/
AttitudeTargetEuler	Target attitude of UAV in Euler (ZYX) form	[TargetRoll, TargetPitch, TargetYaw]	radi
Barometer#	Barometer readings for absolute pressure, relative pressure, and temperature	[PressAbs, PressAltitude, Temp]	Pa, r
Battery	Voltage readings for battery and remaining battery capacity (%)	[Volt1,Volt2, ... Volt16, RemainingCapacity	V, %
GPS#	GPS readings for latitude, longitude, altitude, ground speed, course angle, and number of satellites visible	[lat, long, alt, groundspeed, courseAngle, satellites]	deg deg
Gyro#	Raw body angular velocity readings from IMU sensor	[GyroX, GyroY, GyroZ]	rad/
LocalNED	Local NED coordinates estimated by the UAV	[xNED, yNED, zNED]	met
LocalNEDTarget	Target location in local NED coordinates	[xTarget, yTarget, zTarget]	met
LocalNEDVel	Local NED velocity estimated by the UAV	[vx vy vz]	m/s
LocalNEDVelTarget	Target velocity in NED in local NED	[vxTarget, vyTarget, vzTarget]	m/s
Mag#	Raw magnetometer reading from IMU sensor	[x y z]	Gs

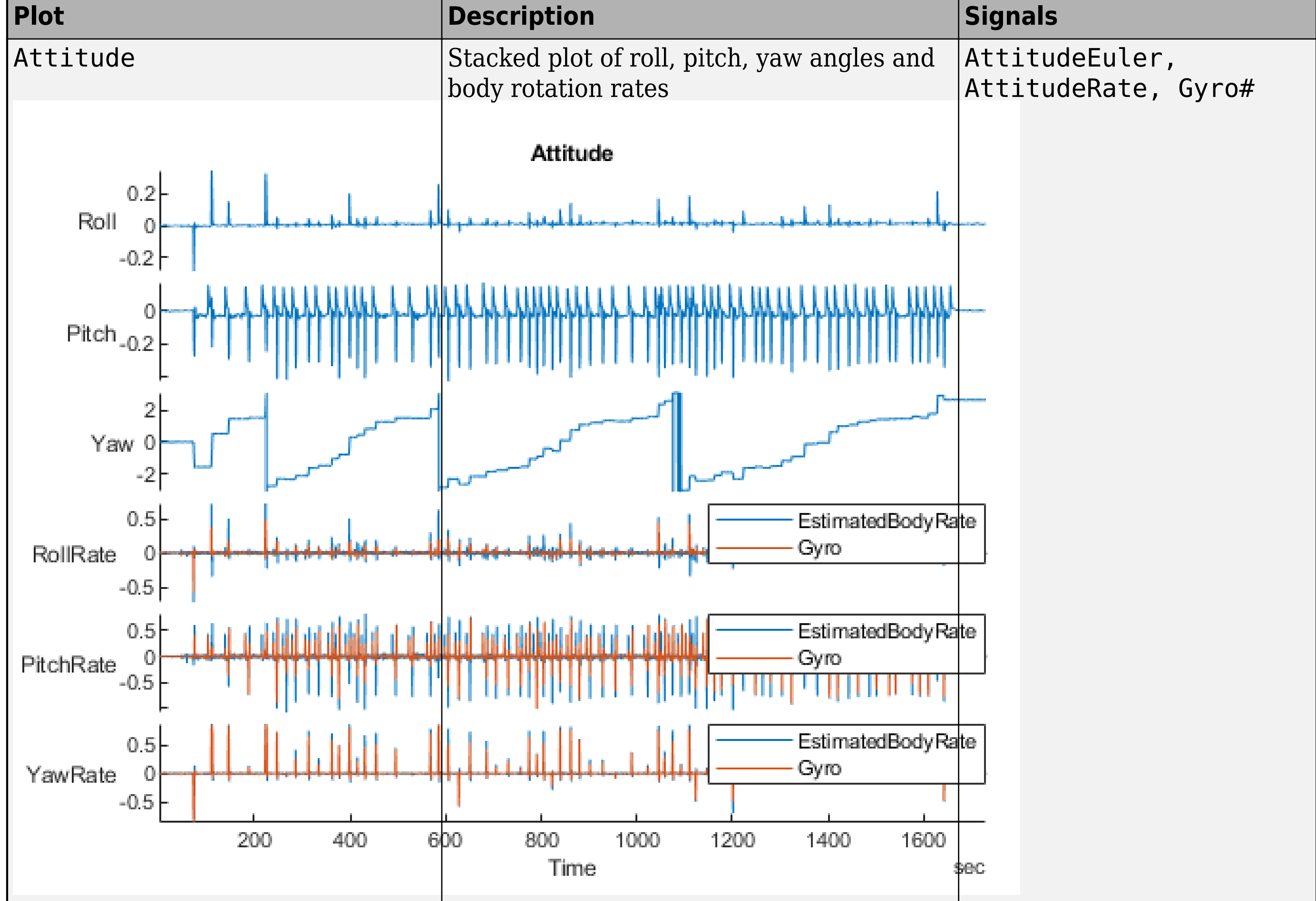
Predefined Plots

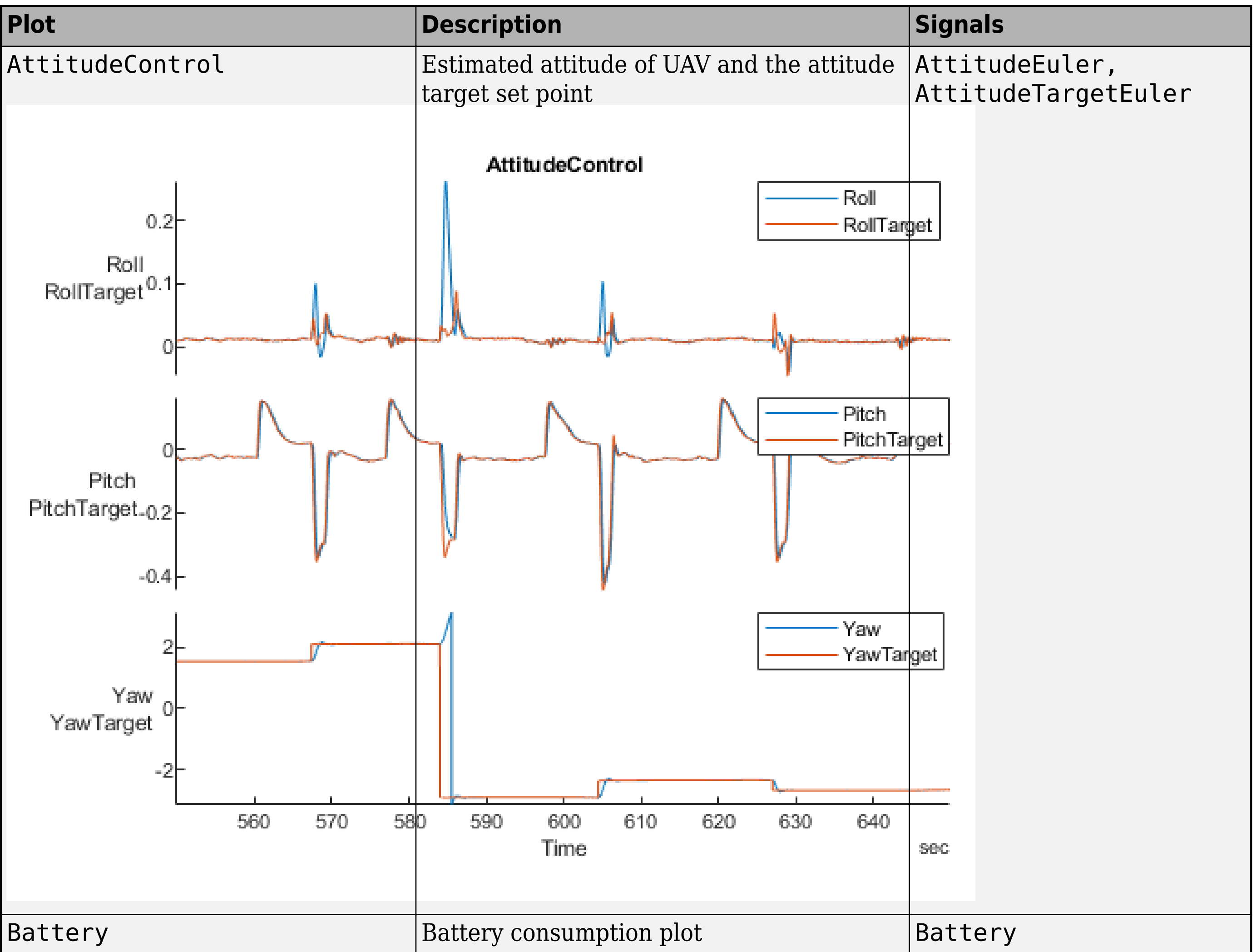
After mapping signals to the list of predefined signals using `mapSignal`, specific plots are made available when calling `show`. To view a list of available plots and their associated signals for your specific object, call `info(mapper,"Plot")`. If you want to define custom plots based on signals, use `updatePlot`.

Each predefined plot has a set of required signals that must be mapped.

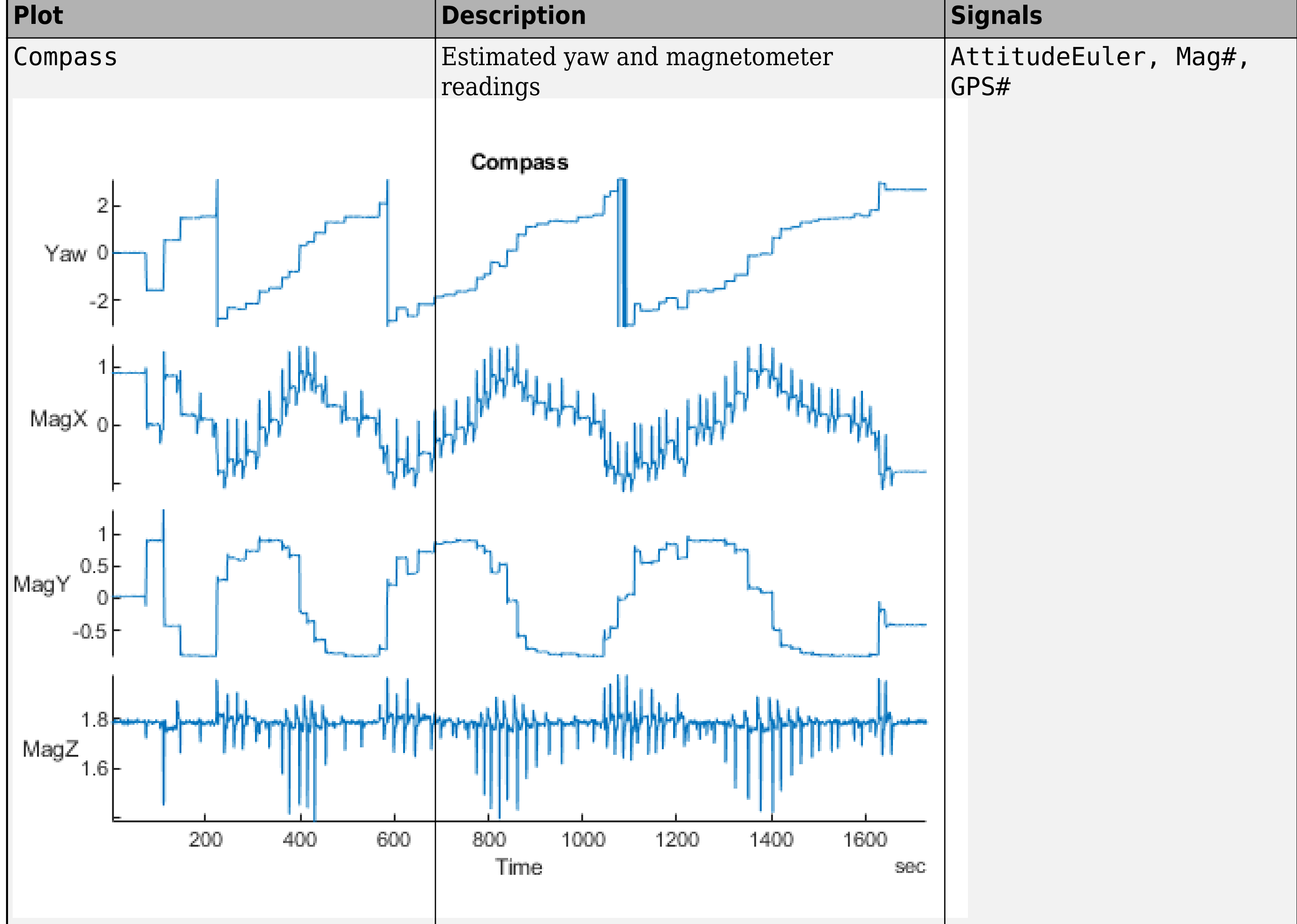
Predefined Plots

Plot	Description	Signals
`Attitude`	Stacked plot of roll, pitch, yaw angles and body rotation rates	`AttitudeEuler`, `AttitudeRate`, `Gyro#`

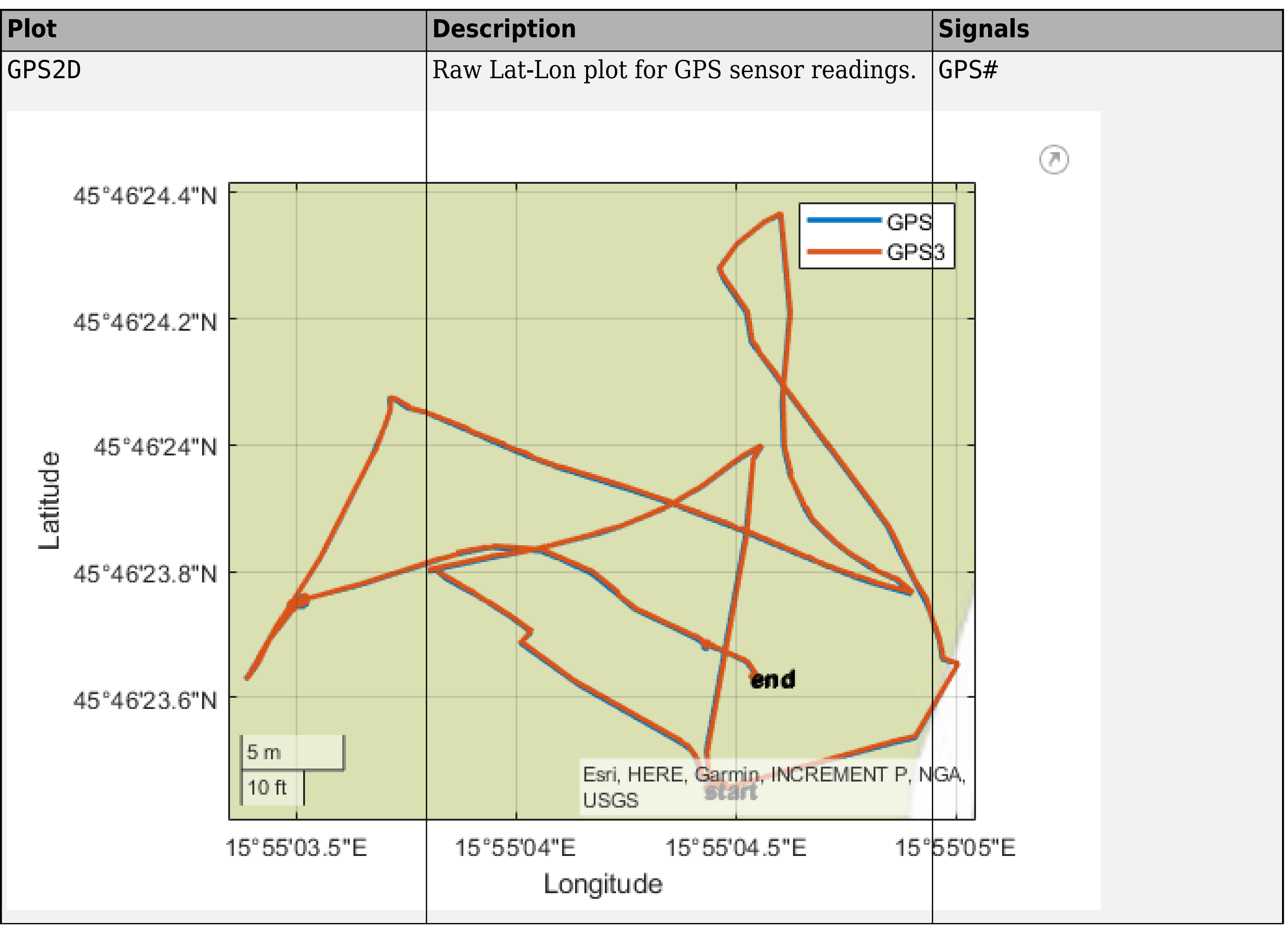


Plot	Description	Signals
AttitudeControl  	Estimated attitude of UAV and the attitude target set point	AttitudeEuler, AttitudeTargetEuler
Battery	Battery consumption plot	Battery

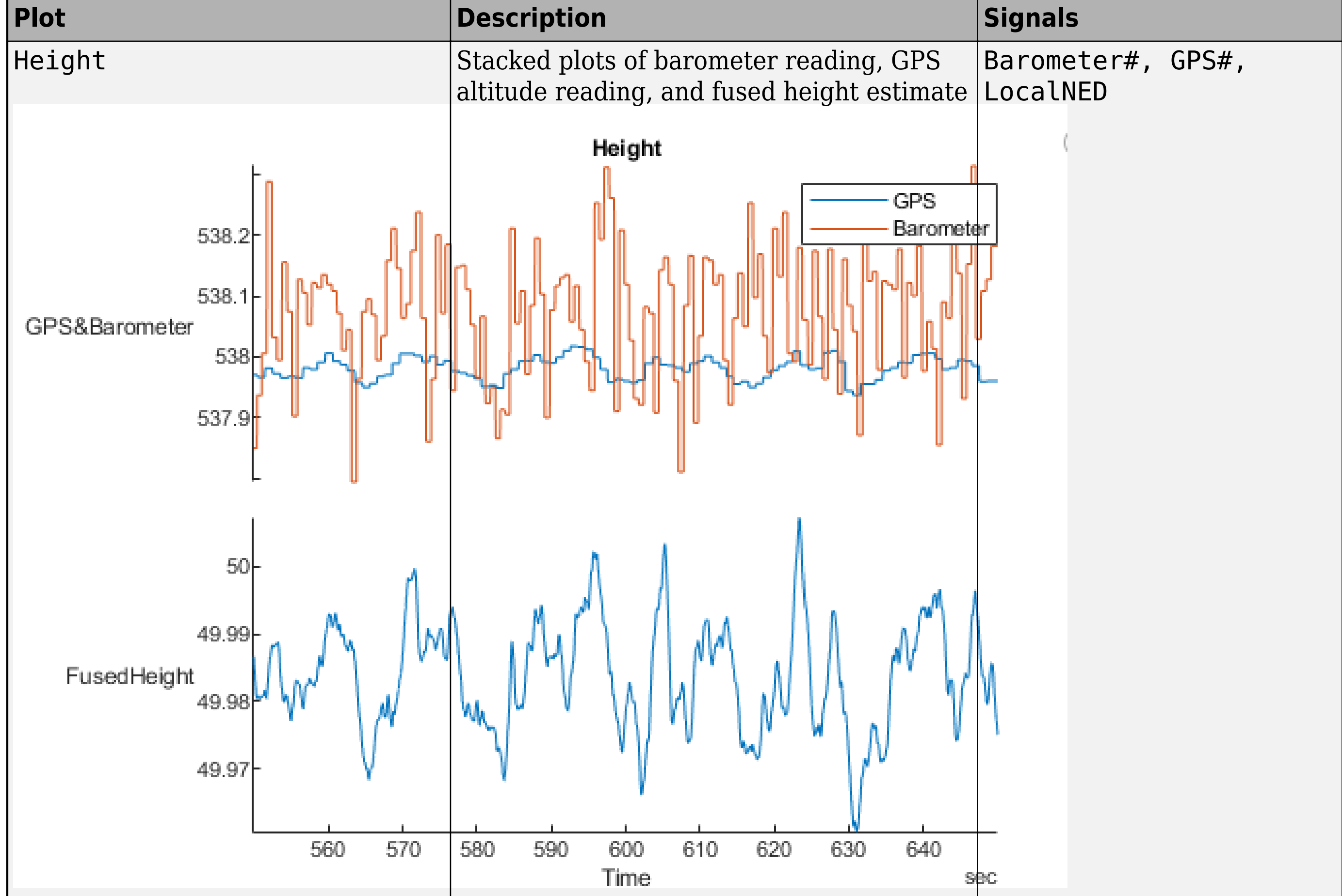
Plot	Description	Signals
`Compass`	Estimated yaw and magnetometer readings	`AttitudeEuler, Mag#, GPS#`

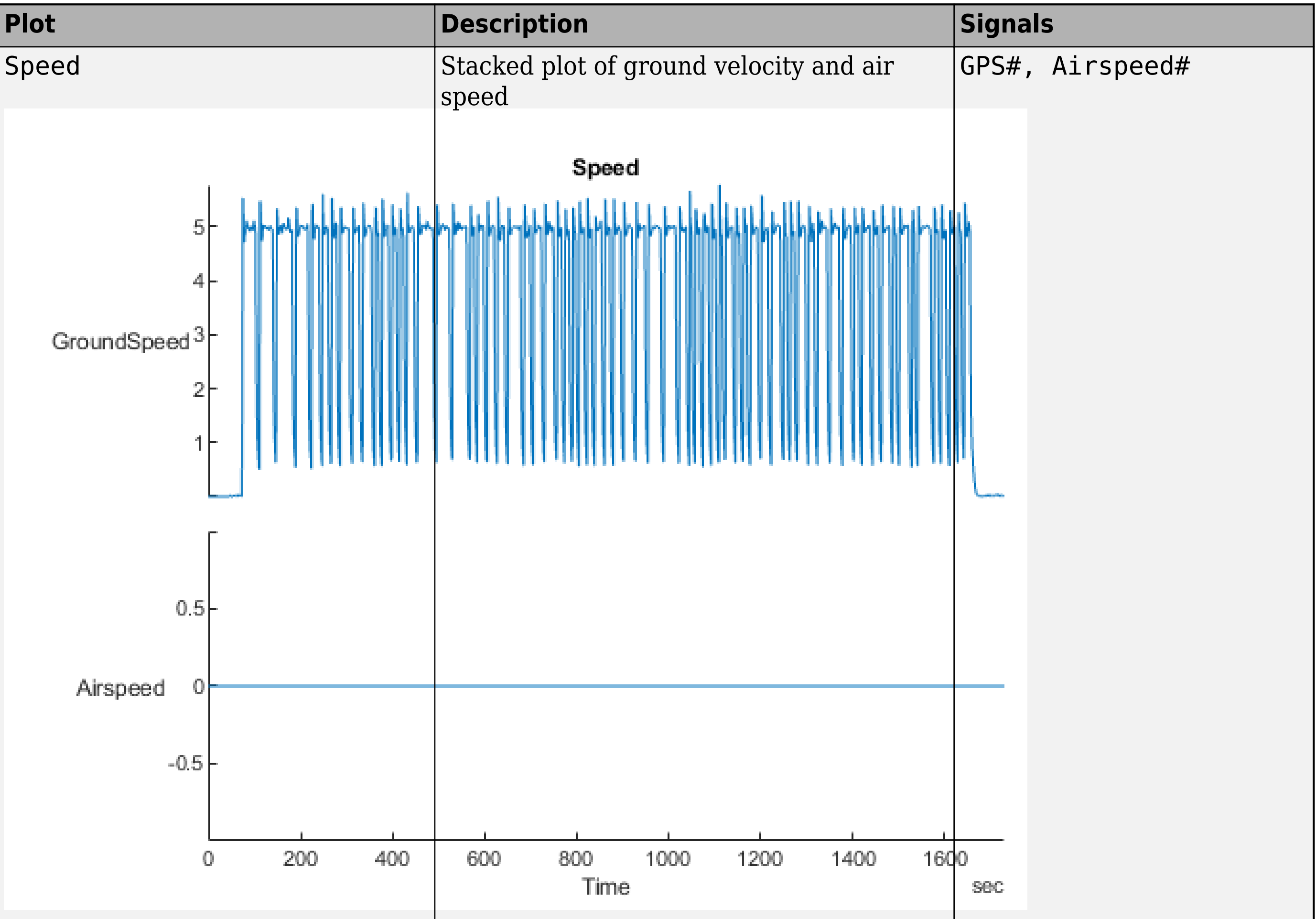


Plot	Description	Signals
GPS2D	Raw Lat-Lon plot for GPS sensor readings.	GPS#

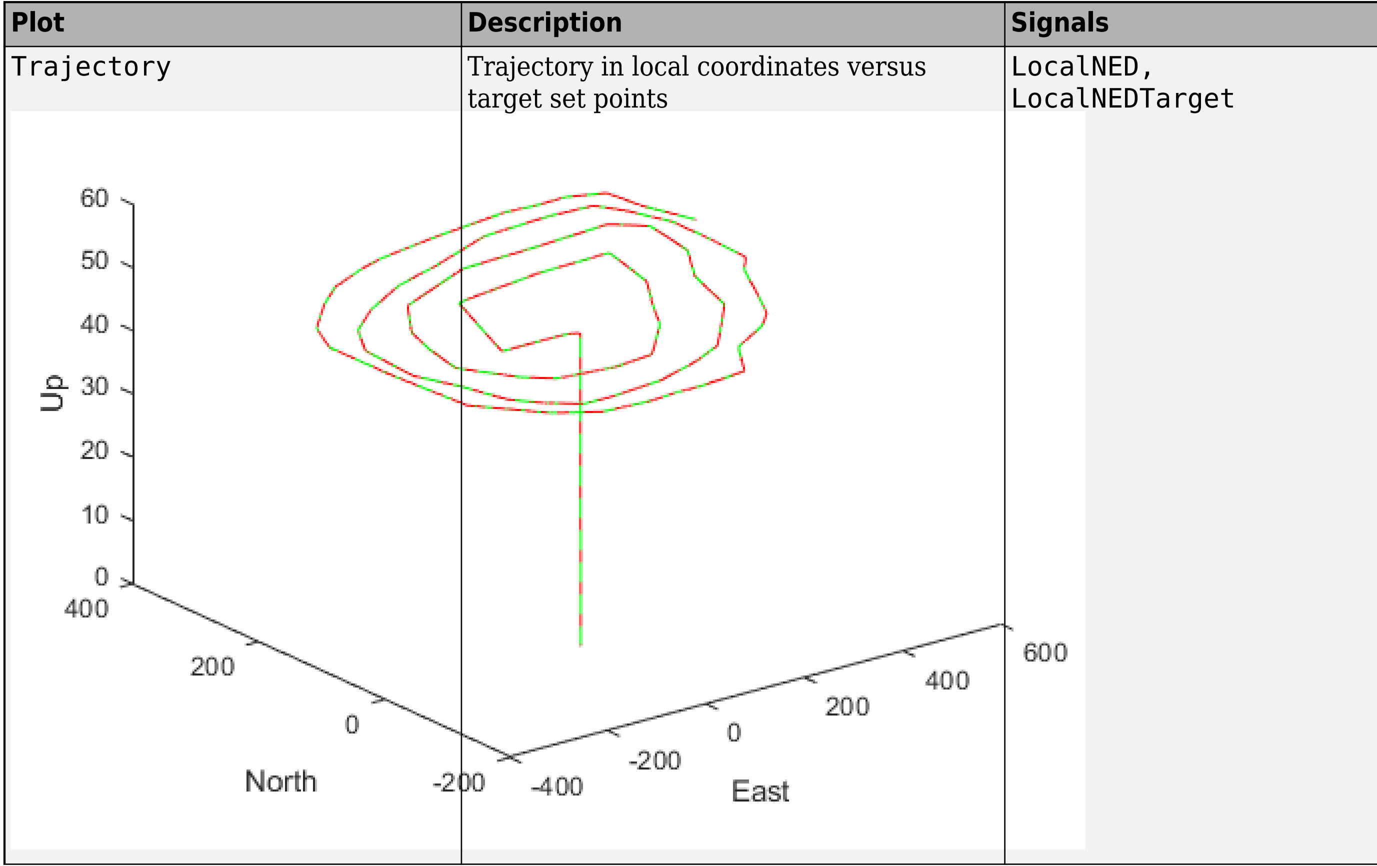


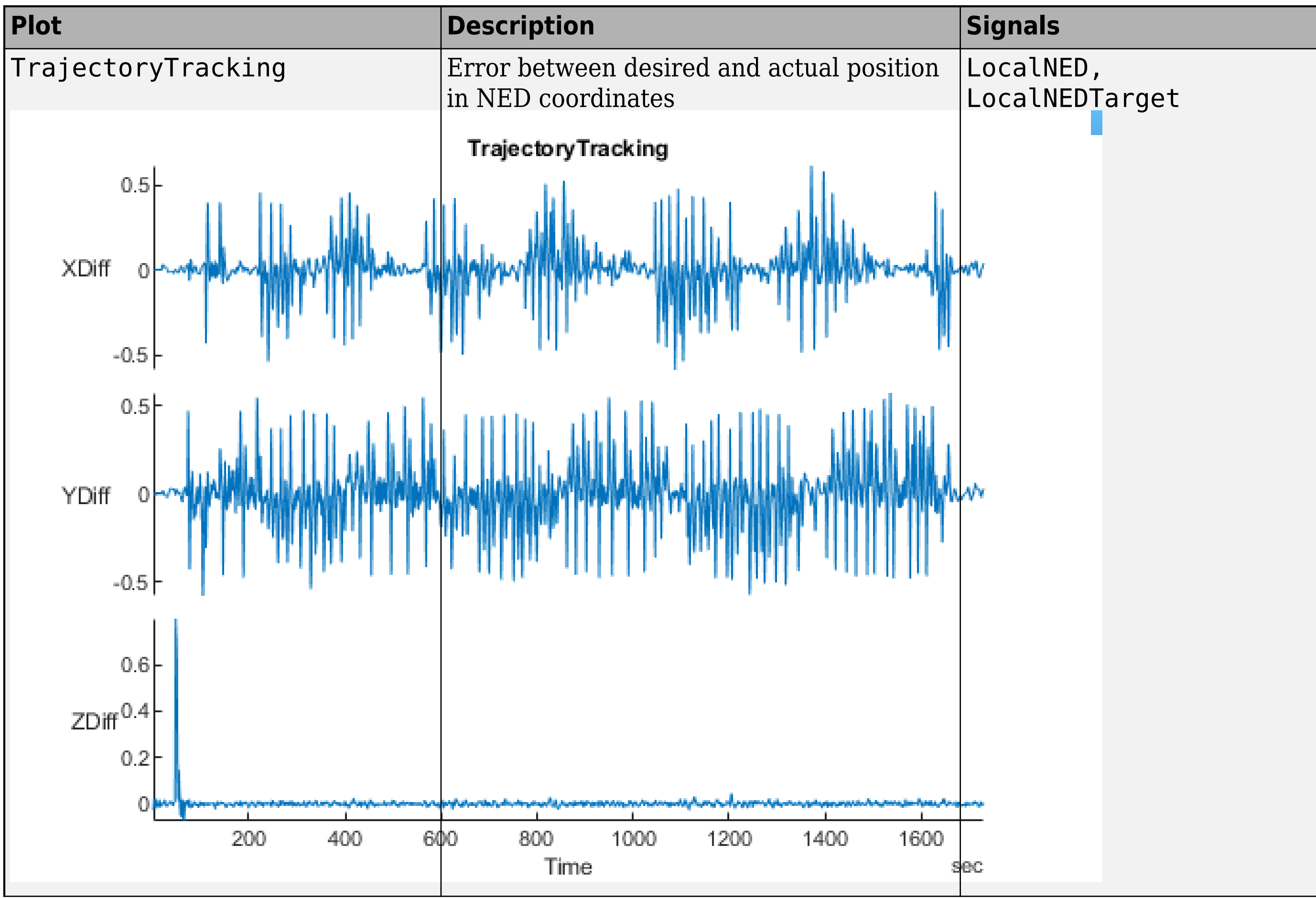
Plot	Description	Signals
`Height`	Stacked plots of barometer reading, GPS altitude reading, and fused height estimate	`Barometer#`, `GPS#`, `LocalNED`

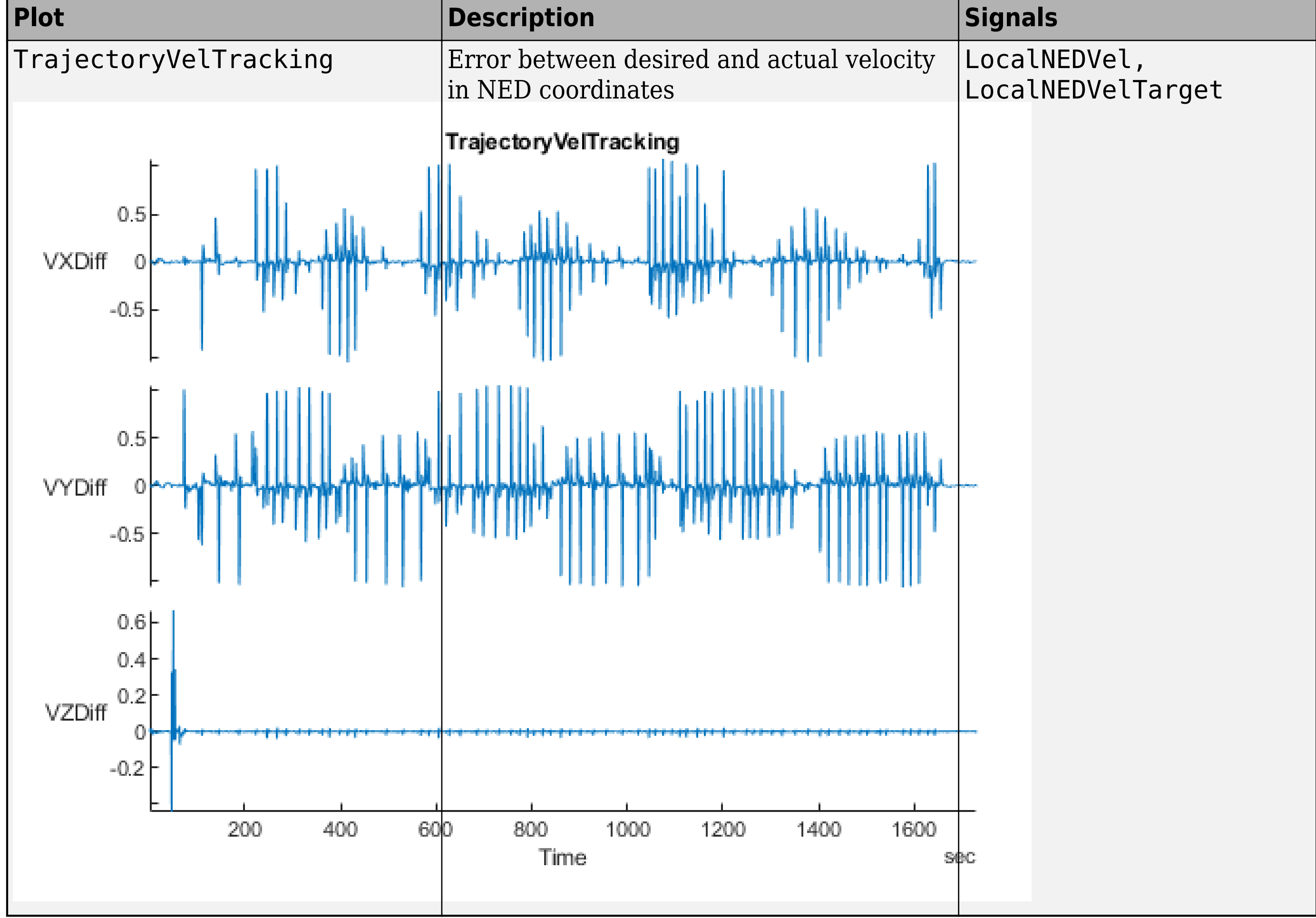


Plot	Description	Signals
Speed 	Stacked plot of ground velocity and air speed	GPS#, Airspeed#

Plot	Description	Signals
`Trajectory`	Trajectory in local coordinates versus target set points	`LocalNED`, `LocalNEDTarget`



Plot	Description	Signals
`TrajectoryTracking`	Error between desired and actual position in NED coordinates	`LocalNED`, `LocalNEDTarget`
 		

Plot	Description	Signals
`TrajectoryVelTracking`	Error between desired and actual velocity in NED coordinates	`LocalNEDVel`, `LocalNEDVelTarget`
		

Version History

Introduced in R2020b

See Also

`flightLogSignalMapping` | `mavlinktlog` | `extract` | `info` | `show` | `updatePlot`

show

Display plots for inspection of UAV logs

Syntax

```
show(mapper,data)
show(mapper,data,timeStart)
show(mapper,data,timeStart,timeEnd)
show( ___ ,"PlotsToShow",plotNames)
plotStruct = show( ___ )
```

Description

`show(mapper,data)` generates all the plots stored in the flight log signal mapping object using the data from an imported flight log. For a list of preconfigured signals and plots, see Predefined Signals on page 2-87 and Predefined Plots on page 2-88.

`show(mapper,data,timeStart)` plots all data starting at the given start time.

`show(mapper,data,timeStart,timeEnd)` plots all data within the interval `[timeStart timeEnd]` inclusive.

`show( ___ ,"PlotsToShow",plotNames)` plots data using any of the previous syntaxes with plot names specified as a string array. These plot names are listed in `mapper.AvailablePlots`

`plotStruct = show( ___ )` returns the plots as a structure of plot names and figure handles.

Input Arguments

`mapper` — Flight log signal mapping
`flightLogSignalMapping` object

Flight log signal mapping object, specified as a `flightLogSignalMapping` object.

`data` — Data from flight log
table | `ulogreader` object

Data from flight log, specified as a table, `ulogreader` object, or other custom option. The data is fed directly into the plot functions specified when you call `updatePlot`.

`timeStart` — Initial time stamp for signal
`duration` object

Initial time stamp for signal to extract, specified as a `duration` object.

`timeEnd` — Final time stamp for signal
`duration` object

Final time stamp for signal to extract, specified as a `duration` object.

Output Arguments

`plotStruct` — Figures of individual plots
structure

Figured of individual plots, returned as a structure of plot names and associated figure handles.

More About

Predefined Signals

A set of predefined signals and plots are configured in the `flightLogSignalMapping` object. Depending on your log file type, you can map specific signals to the provided signal names using `mapSignal`. You can also call `info` to view the table for your log type and see whether you have already mapped a signal to that plot type.

Specify the `SignalName` as the input to `mapSignal`. Signals with the format `SignalName#` support mapping multiple signals of the same type. Replace `#` with incremental integers for each signal name when calling `mapSignal`.

The predefined signals have specific names and required fields when mapping the signal.

Predefined Signals

Signal Name	Description	Fields	Unit
Accel#	Raw magnetometer reading from IMU sensor	[ax ay az]	m/s^2
Airspeed#	Airspeed reading of pressure differential, indicated air speed, and temperature	[PressDiff, AirSpeed, Temp]	Pa, r
AttitudeEuler	Attitude of UAV in Euler (ZYX) form	[Roll, Pitch, Yaw]	radi
AttitudeRate	Angular velocity along each body axis	[xRotRate, yRotRate, zRotRate]	rad/
AttitudeTargetEuler	Target attitude of UAV in Euler (ZYX) form	[TargetRoll, TargetPitch, TargetYaw]	radi
Barometer#	Barometer readings for absolute pressure, relative pressure, and temperature	[PressAbs, PressAltitude, Temp]	Pa, r
Battery	Voltage readings for battery and remaining battery capacity (%)	[Volt1,Volt2, ... Volt16, RemainingCapacity	V, %
GPS#	GPS readings for latitude, longitude, altitude, ground speed, course angle, and number of satellites visible	[lat, long, alt, groundspeed, courseAngle, satellites]	degr degr
Gyro#	Raw body angular velocity readings from IMU sensor	[GyroX, GyroY, GyroZ]	rad/
LocalNED	Local NED coordinates estimated by the UAV	[xNED, yNED, zNED]	met
LocalNEDTarget	Target location in local NED coordinates	[xTarget, yTarget, zTarget]	met
LocalNEDVel	Local NED velocity estimated by the UAV	[vx vy vz]	m/s
LocalNEDVelTarget	Target velocity in NED in local NED	[vxTarget, vyTarget, vzTarget]	m/s
Mag#	Raw magnetometer reading from IMU sensor	[x y z]	Gs

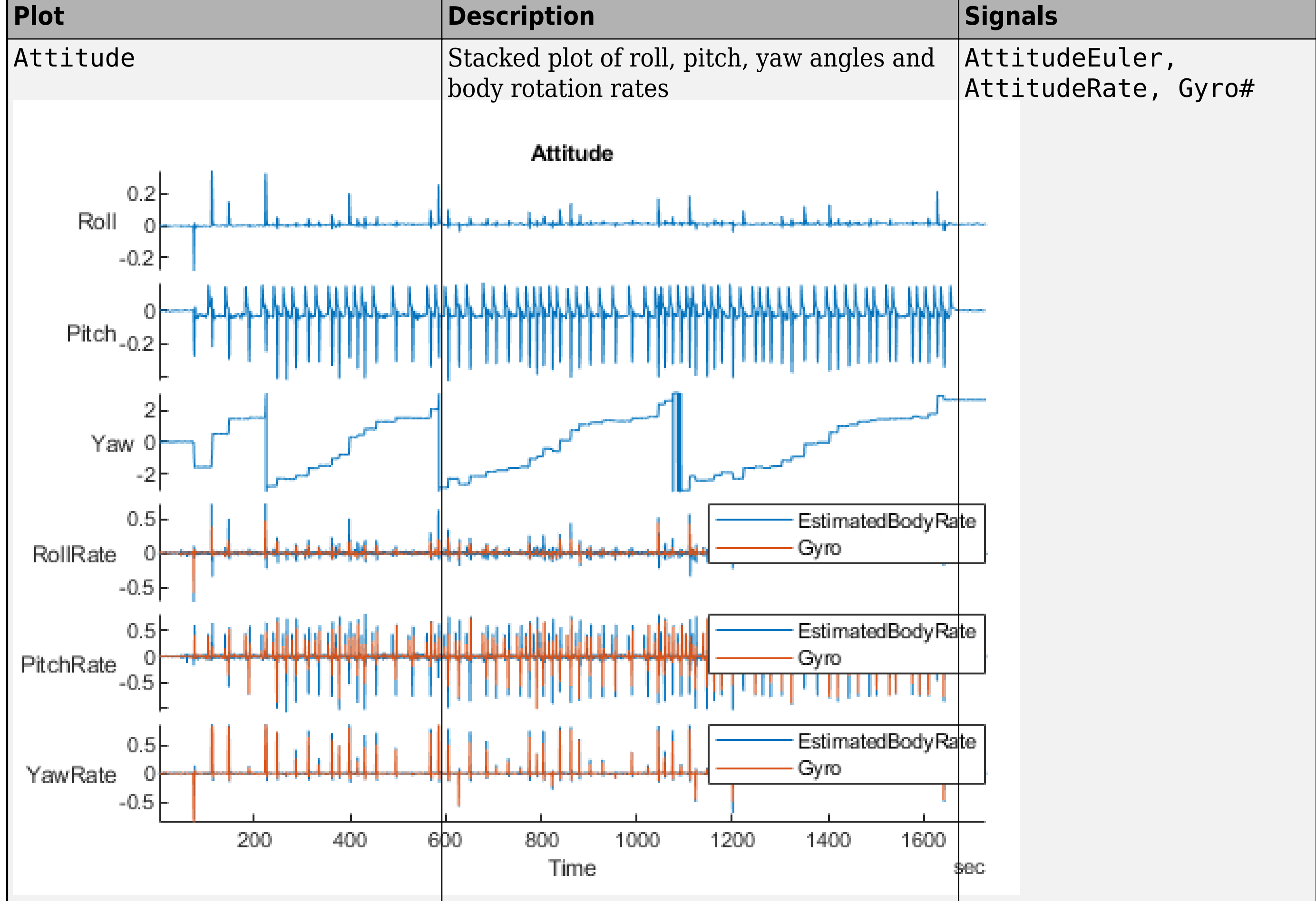
Predefined Plots

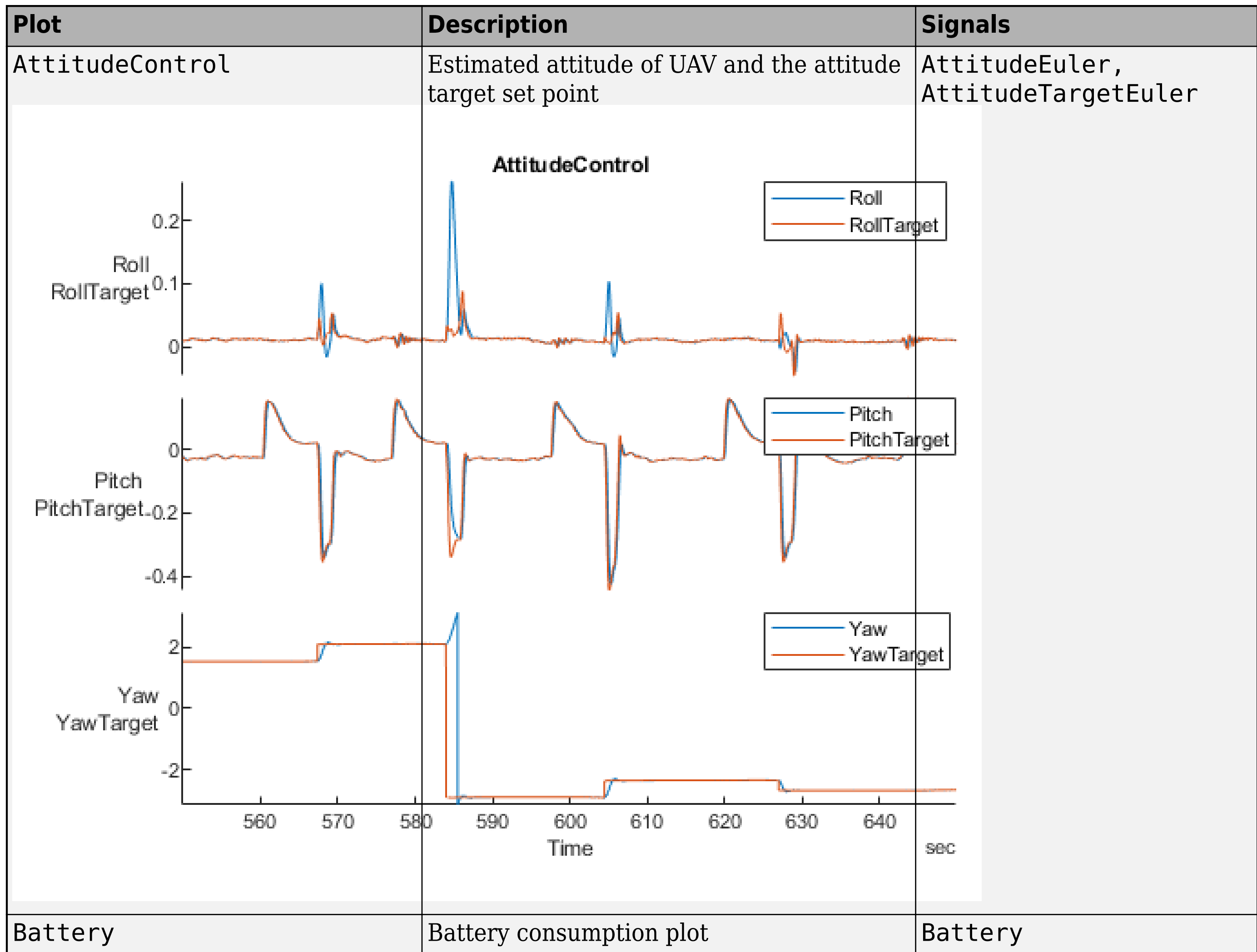
After mapping signals to the list of predefined signals using `mapSignal`, specific plots are made available when calling `show`. To view a list of available plots and their associated signals for your specific object, call `info(mapper,"Plot")`. If you want to define custom plots based on signals, use `updatePlot`.

Each predefined plot has a set of required signals that must be mapped.

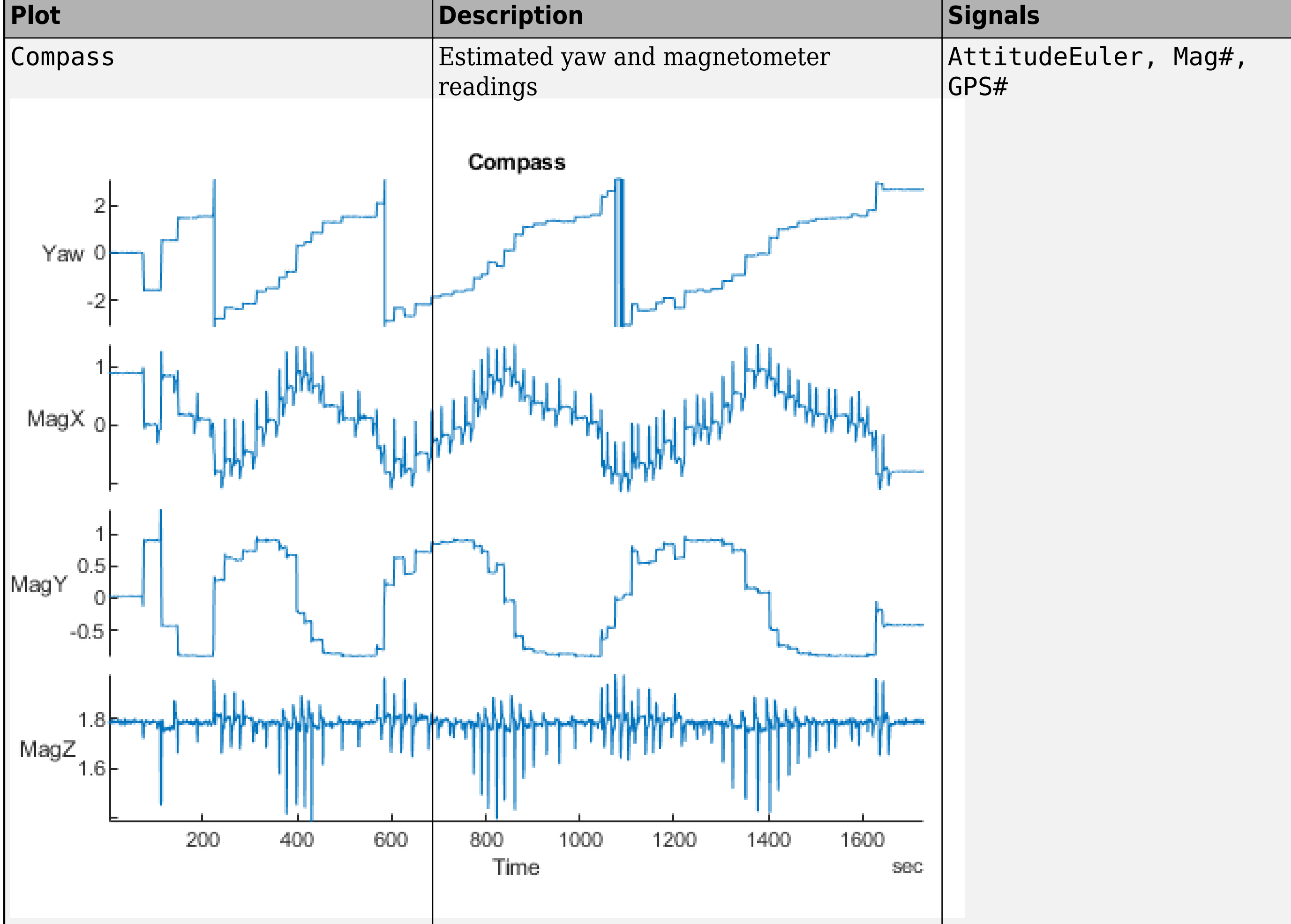
Predefined Plots

Plot	Description	Signals
`Attitude`	Stacked plot of roll, pitch, yaw angles and body rotation rates	`AttitudeEuler, AttitudeRate, Gyro#`

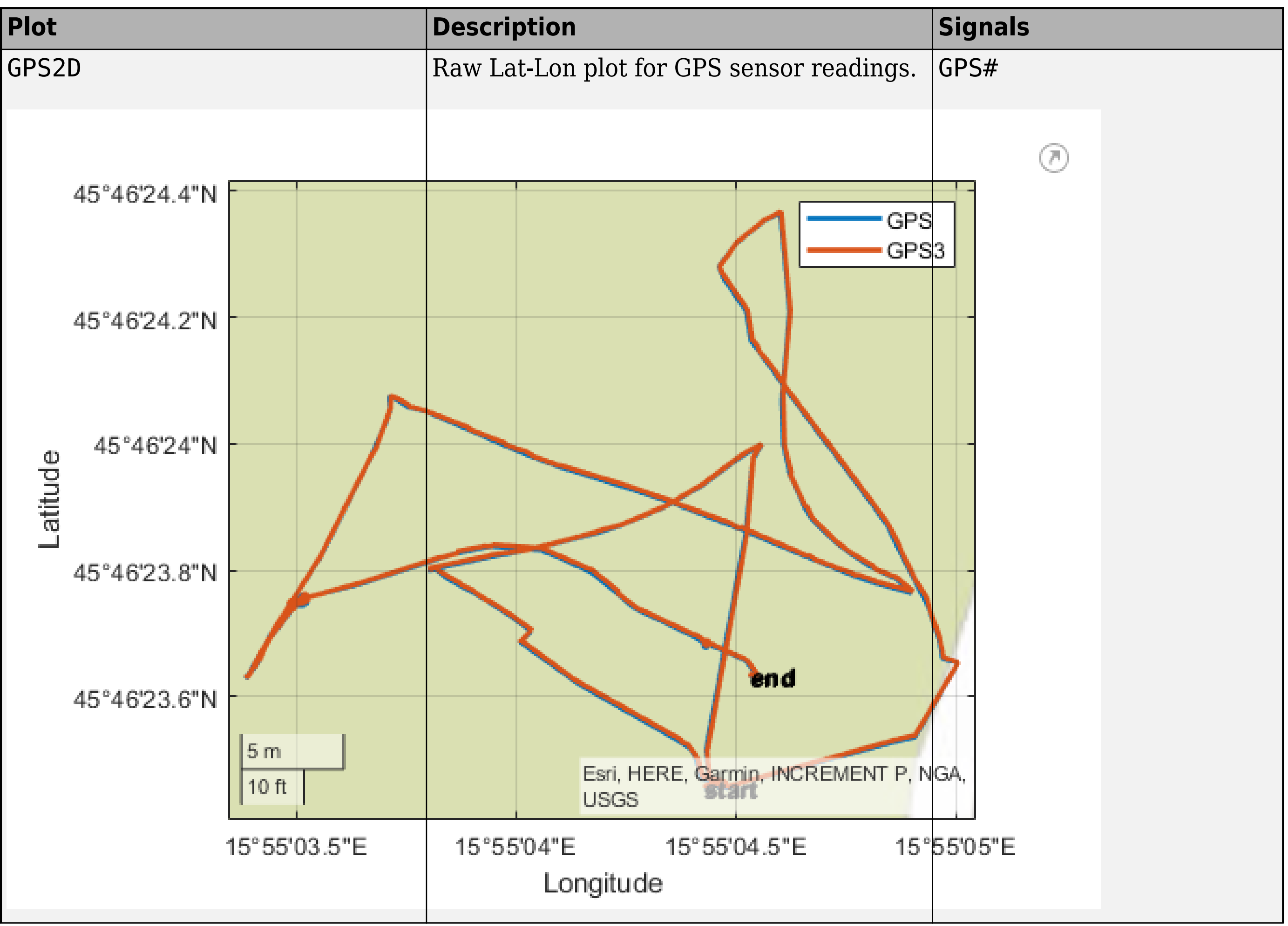


Plot	Description	Signals
AttitudeControl  	Estimated attitude of UAV and the attitude target set point	AttitudeEuler, AttitudeTargetEuler
Battery	Battery consumption plot	Battery

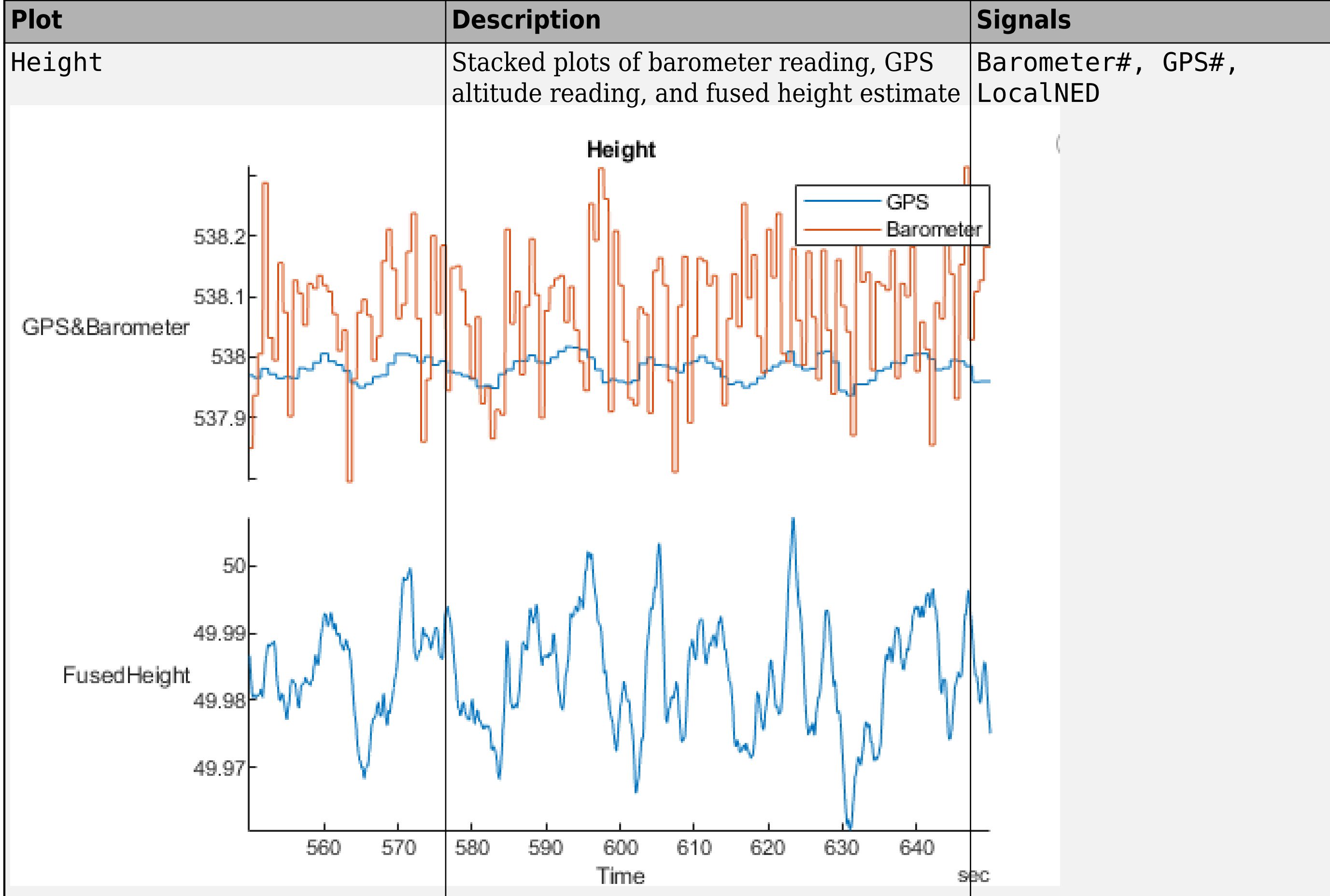
Plot	Description	Signals
`Compass`	Estimated yaw and magnetometer readings	`AttitudeEuler, Mag#, GPS#`



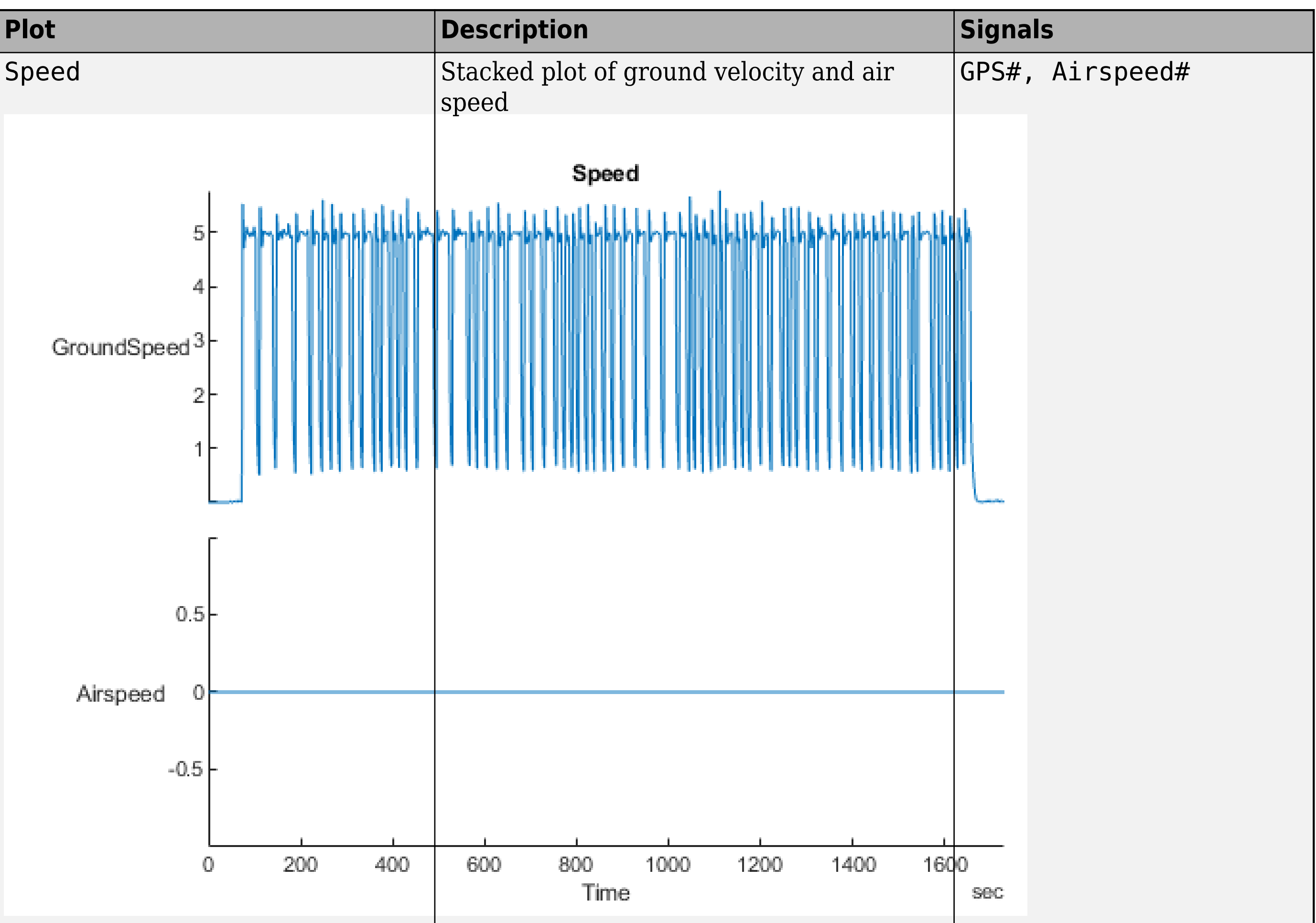
Plot	Description	Signals
GPS2D	Raw Lat-Lon plot for GPS sensor readings.	GPS#



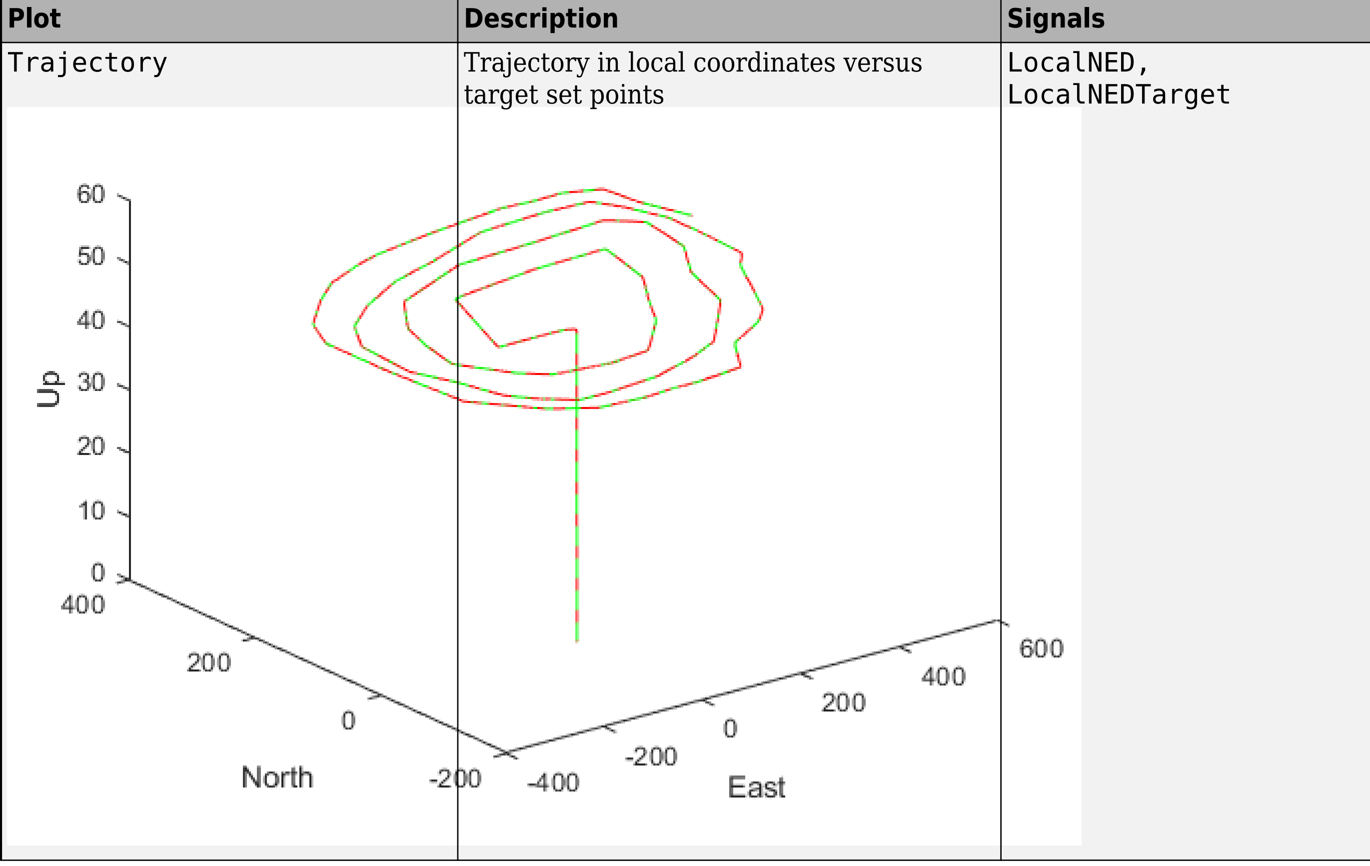
Plot	Description	Signals
Height	Stacked plots of barometer reading, GPS altitude reading, and fused height estimate	Barometer#, GPS#, LocalNED



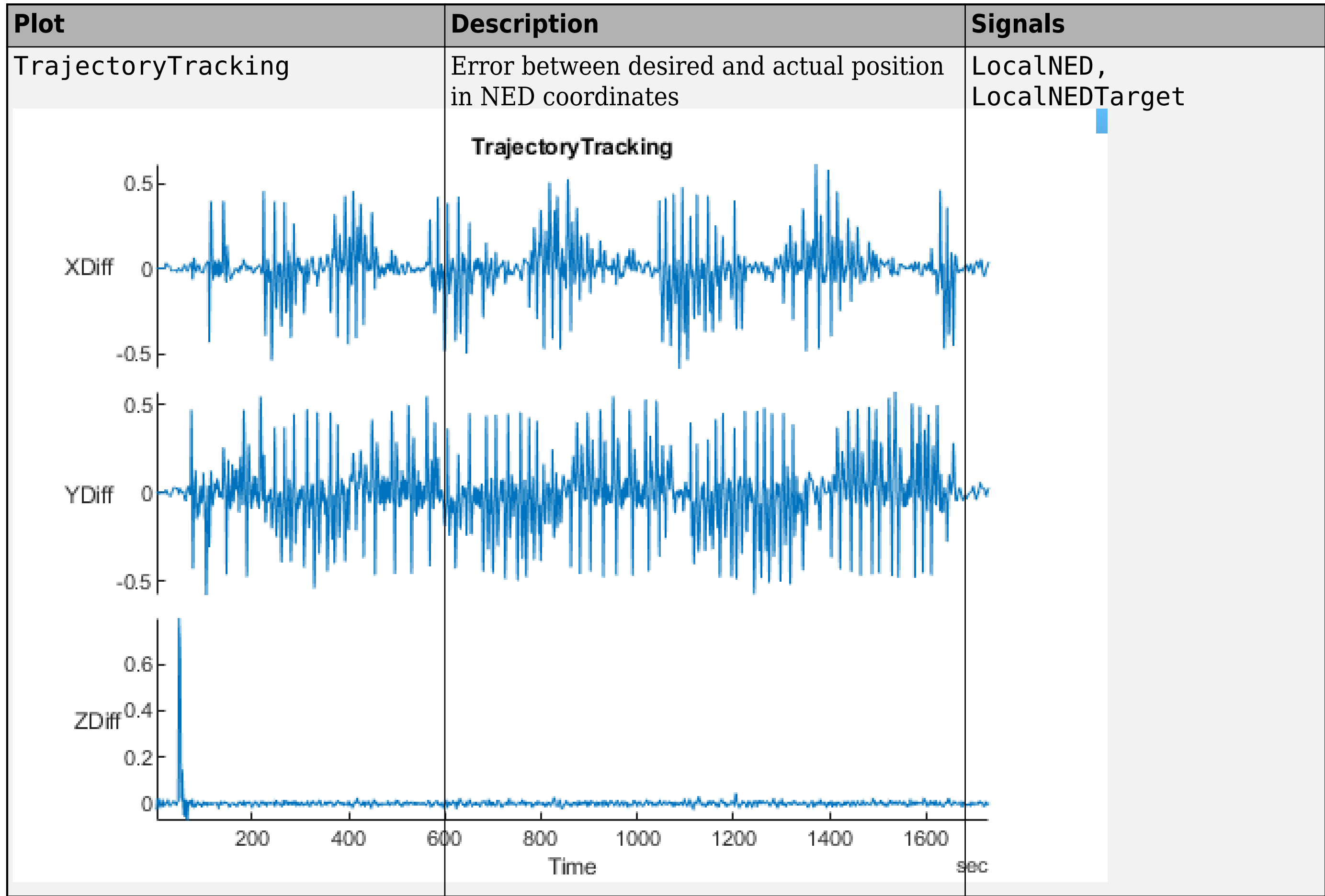
Plot	Description	Signals
`Speed`	Stacked plot of ground velocity and air speed	`GPS#, Airspeed#`



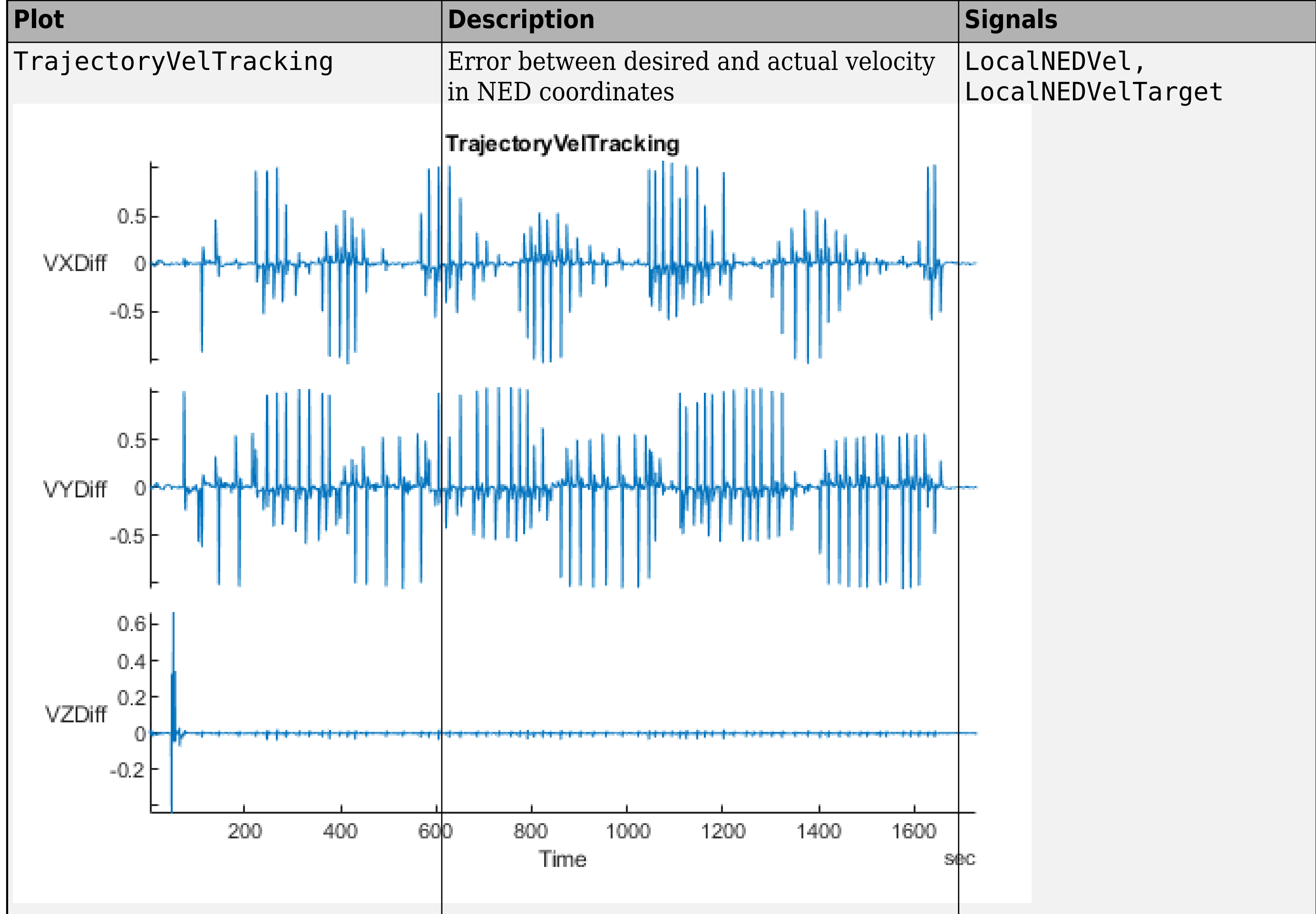
Plot	Description	Signals
Trajectory	Trajectory in local coordinates versus target set points	LocalNED, LocalNEDTarget



Plot	Description	Signals
TrajectoryTracking	Error between desired and actual position in NED coordinates	LocalNED, LocalNEDTarget



Plot	Description	Signals
TrajectoryVelTracking	Error between desired and actual velocity in NED coordinates	LocalNEDVel, LocalNEDVelTarget



Version History

Introduced in R2020b

See Also

flightLogSignalMapping | mavlinktlog | extract | info | mapSignal | updatePlot

updatePlot

Update UAV flight log plot functions

Syntax

`updatePlot(mapper,plotName,plotFunc,requiredSignals)`

Description

`updatePlot(mapper,plotName,plotFunc,requiredSignals)` adds or updates the plot with name `plotName` stored in `mapper`. Specify the plot function as a predefined plot name or function handle and the required signals for the plot. For a list of preconfigured signals and plots, see Predefined Signals on page 2-99 and Predefined Plots on page 2-100.

Input Arguments

`mapper` — Flight log signal mapping

`flightLogSignalMapping` object

Flight log signal mapping object, specified as a `flightLogSignalMapping` object.

`plotName` — Name of plot

string scalar | character vector

Name of plot, specified as a string scalar or character vector. This name is either added or updated in the `AvailablePlots` property of `mapper`.

Example: `"IMU"`

Data Types: `char` | `string`

`plotFunc` — Function for generating plot

function handle

Function for generating plot, specified as a function handle. The function is of the form:

```
f = plotFunc(signal1, signal2, ...)
```

The function takes input signals as structures with two fields, `"Names"` and `"Values"`, and generates a plot output as a figure handle using those signals.

Example: `@(acc, gyro, mag)plotIMU(acc, gyro, mag)`

Data Types: `function_handle`

`requiredSignals` — List of required signal names

string array | cell array of character vectors

List of required signal names, specified as a string array or cell array of character vectors.

Example: `["LocalNED.X" "LocalNED.Y" "LocalNED.Z"]`

Data Types: `char` | `string`

More About

Predefined Signals

A set of predefined signals and plots are configured in the `flightLogSignalMapping` object. Depending on your log file type, you can map specific signals to the provided signal names using `mapSignal`. You can also call `info` to view the table for your log type and see whether you have already mapped a signal to that plot type.

Specify the `SignalName` as the input to `mapSignal`. Signals with the format `SignalName#` support mapping multiple signals of the same type. Replace # with incremental integers for each signal name when calling `mapSignal`.

The predefined signals have specific names and required fields when mapping the signal.

Predefined Signals

Signal Name	Description	Fields	Uni
`Accel#`	Raw magnetometer reading from IMU sensor	`[ax ay az]`	m/s
`Airspeed#`	Airspeed reading of pressure differential, indicated air speed, and temperature	`[PressDiff, AirSpeed, Temp]`	Pa,
`AttitudeEuler`	Attitude of UAV in Euler (ZYX) form	`[Roll, Pitch, Yaw]`	radi
`AttitudeRate`	Angular velocity along each body axis	`[xRotRate, yRotRate, zRotRate]`	rad/
`AttitudeTargetEuler`	Target attitude of UAV in Euler (ZYX) form	`[TargetRoll, TargetPitch, TargetYaw]`	radi
`Barometer#`	Barometer readings for absolute pressure, relative pressure, and temperature	`[PressAbs, PressAltitude, Temp]`	Pa,
`Battery`	Voltage readings for battery and remaining battery capacity (%)	`[Volt1,Volt2, ... Volt16, RemainingCapacity`	V, %
`GPS#`	GPS readings for latitude, longitude, altitude, ground speed, course angle, and number of satellites visible	`[lat, long, alt, groundspeed, courseAngle, satellites]`	deg deg
`Gyro#`	Raw body angular velocity readings from IMU sensor	`[GyroX, GyroY, GyroZ]`	rad/
`LocalNED`	Local NED coordinates estimated by the UAV	`[xNED, yNED, zNED]`	met
`LocalNEDTarget`	Target location in local NED coordinates	`[xTarget, yTarget, zTarget]`	met
`LocalNEDVel`	Local NED velocity estimated by the UAV	`[vx vy vz]`	m/s
`LocalNEDVelTarget`	Target velocity in NED in local NED	`[vxTarget, vyTarget, vzTarget]`	m/s
`Mag#`	Raw magnetometer reading from IMU sensor	`[x y z]`	Gs

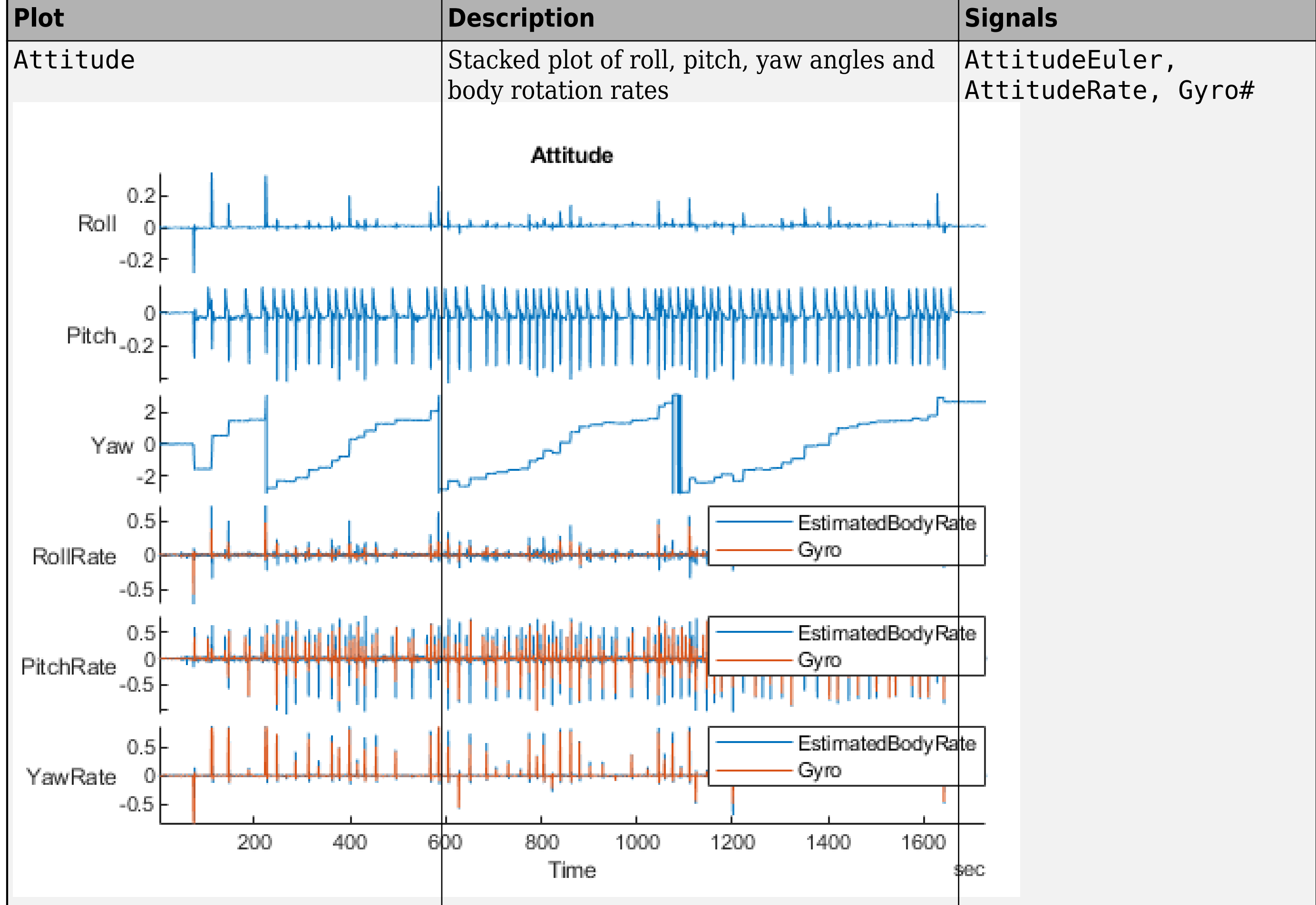
Predefined Plots

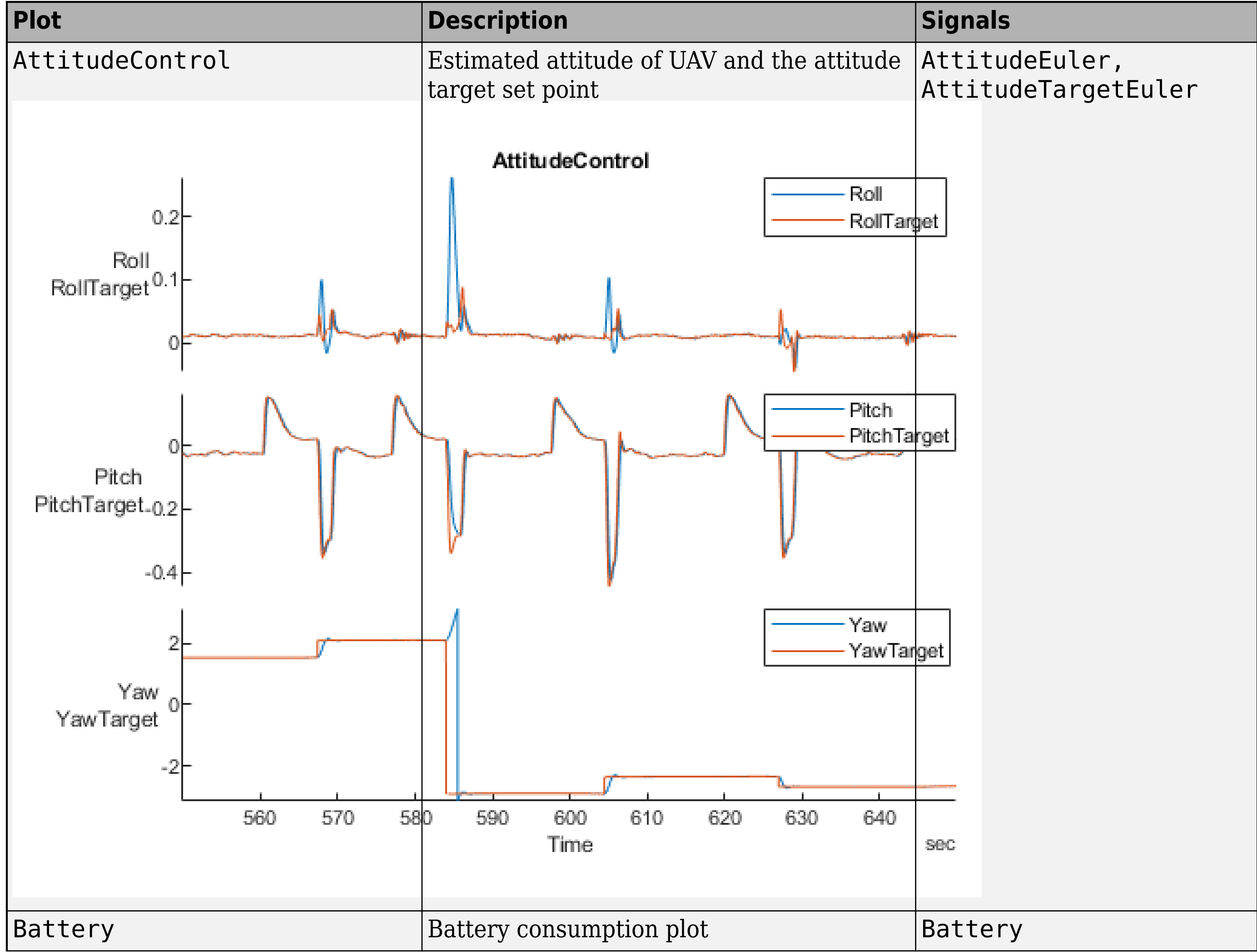
After mapping signals to the list of predefined signals using `mapSignal`, specific plots are made available when calling `show`. To view a list of available plots and their associated signals for your specific object, call `info(mapper,"Plot")`. If you want to define custom plots based on signals, use `updatePlot`.

Each predefined plot has a set of required signals that must be mapped.

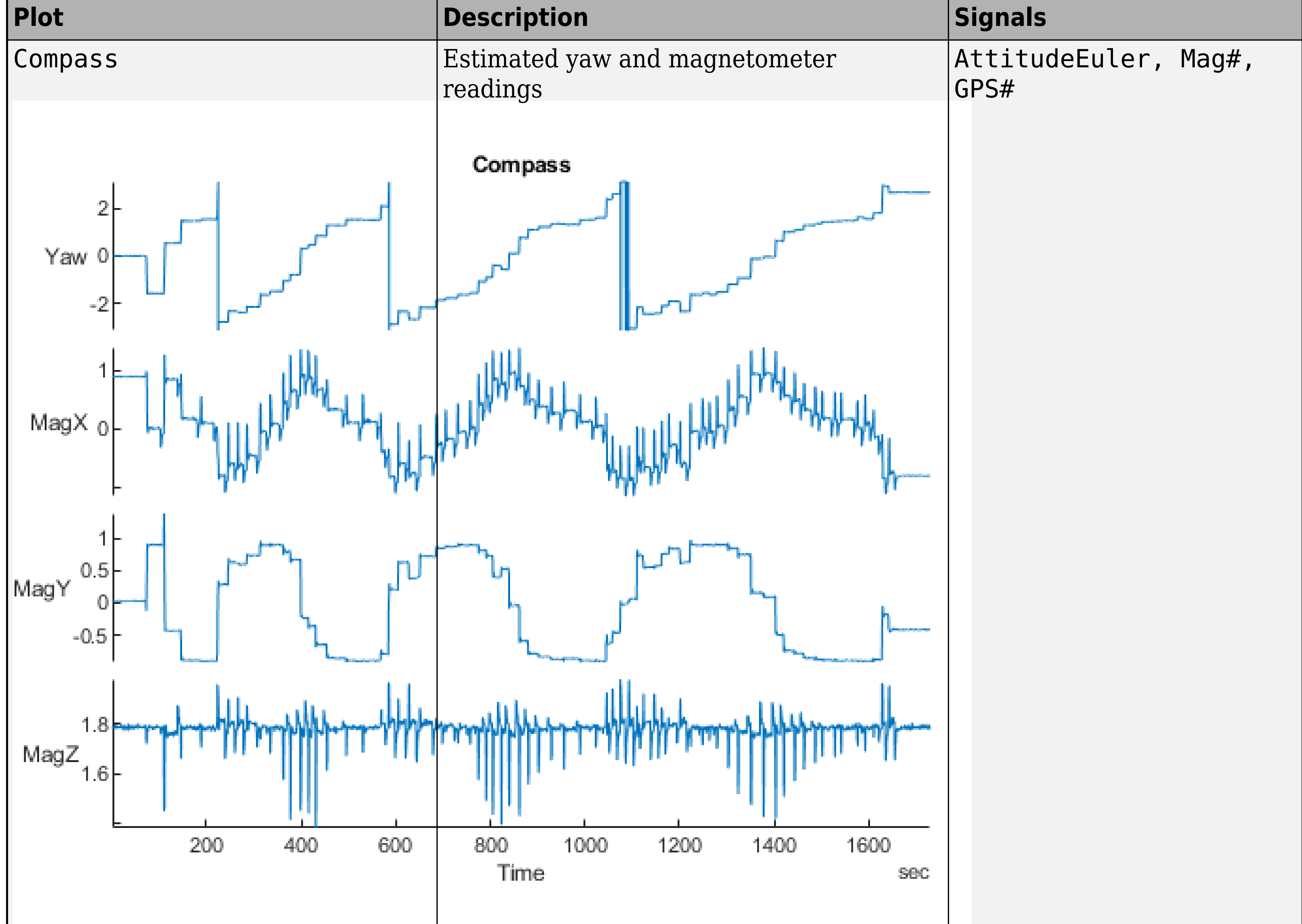
Predefined Plots

Plot	Description	Signals
`Attitude`	Stacked plot of roll, pitch, yaw angles and body rotation rates	`AttitudeEuler`, `AttitudeRate`, `Gyro#`

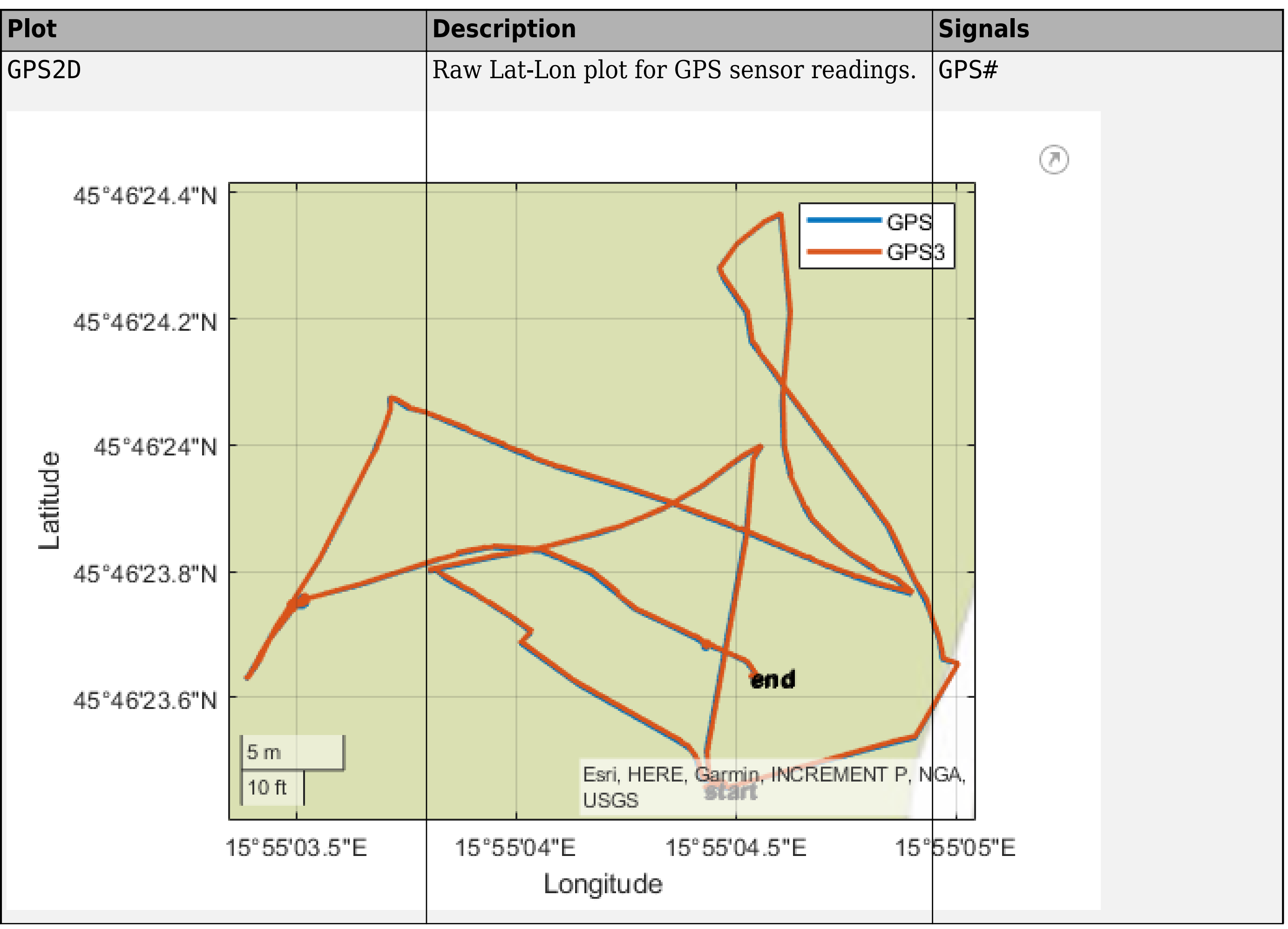


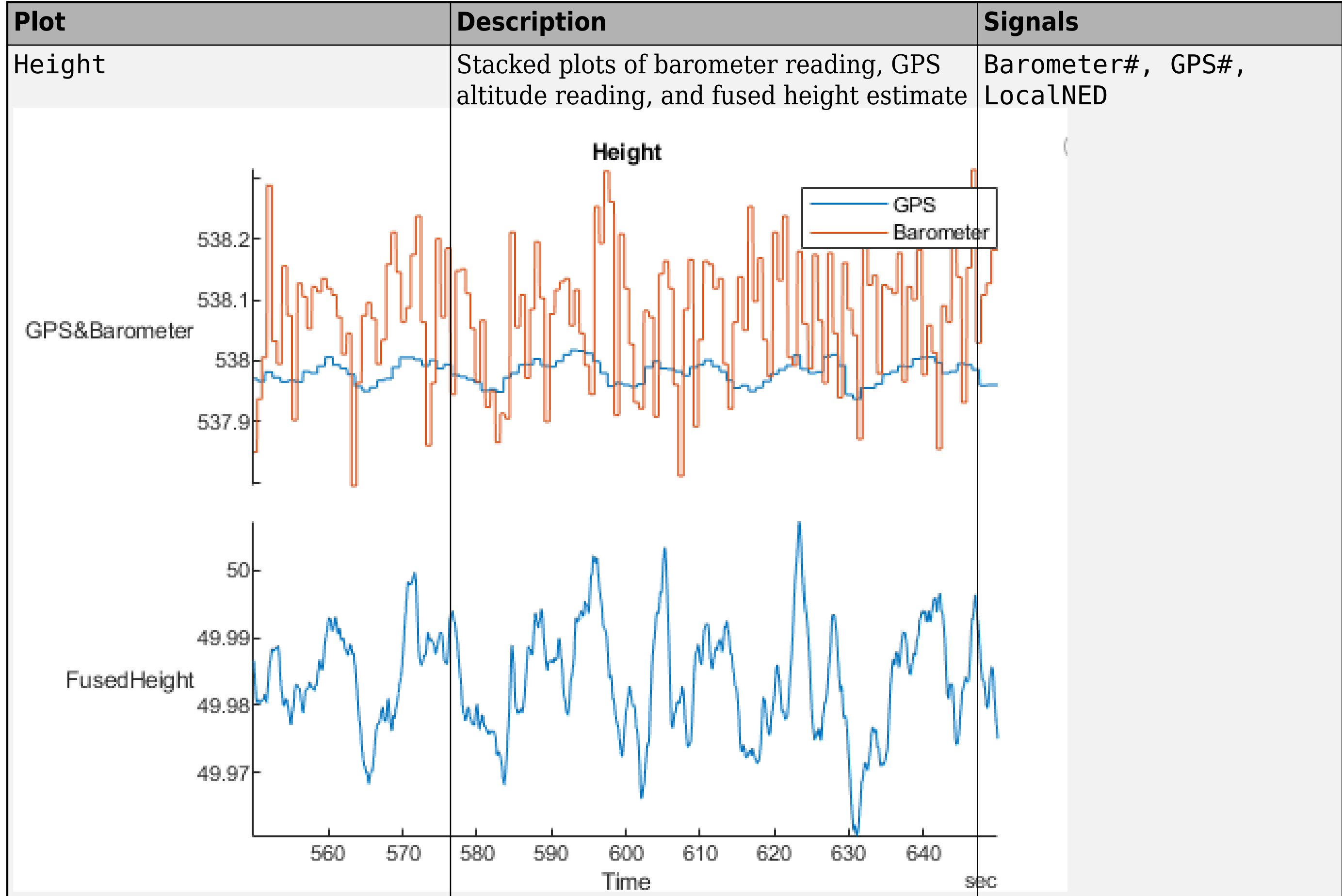
Plot	Description	Signals
AttitudeControl  	Estimated attitude of UAV and the attitude target set point	AttitudeEuler, AttitudeTargetEuler
Battery	Battery consumption plot	Battery

Plot	Description	Signals
`Compass`	Estimated yaw and magnetometer readings	`AttitudeEuler, Mag#, GPS#`

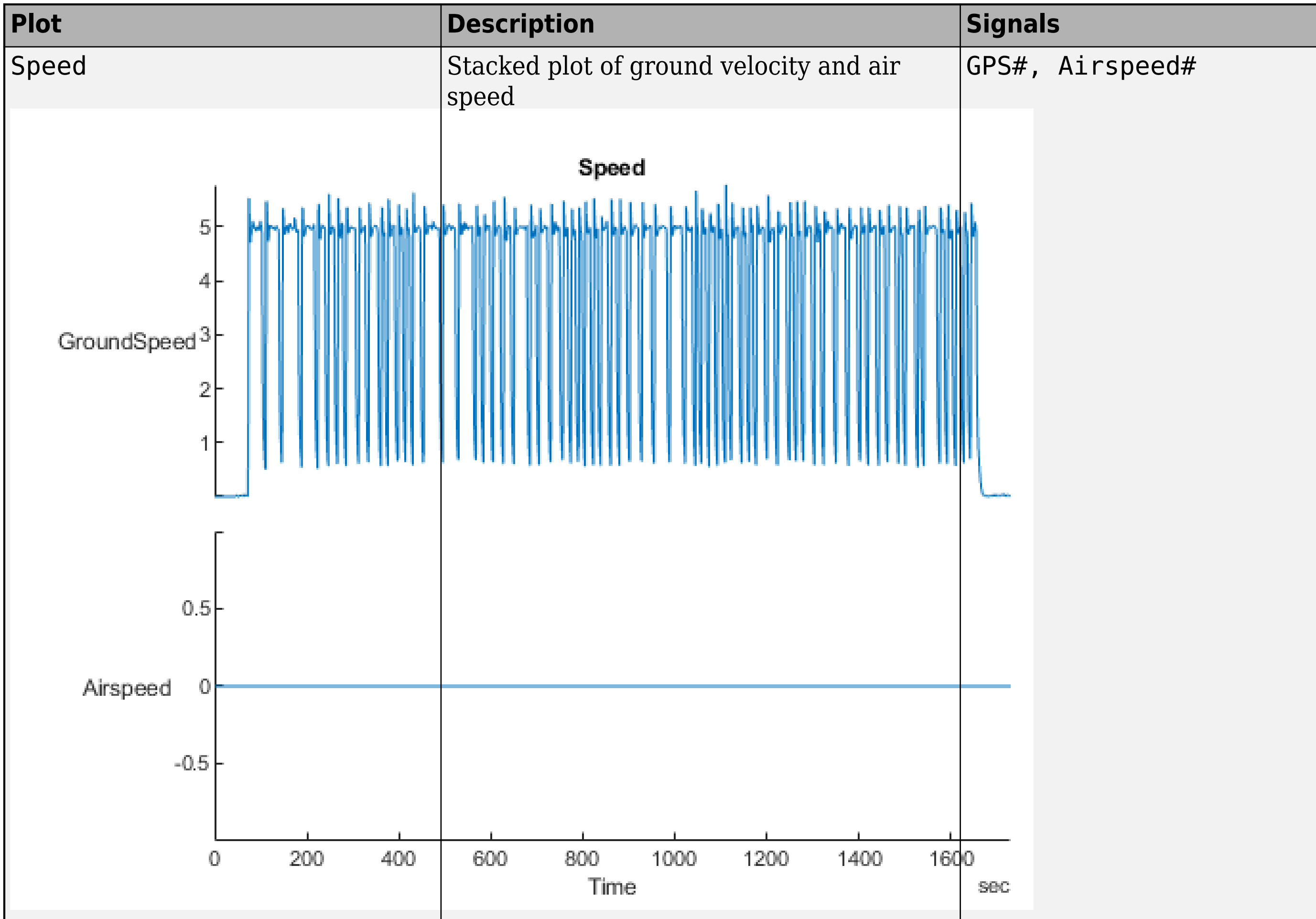


Plot	Description	Signals
GPS2D	Raw Lat-Lon plot for GPS sensor readings.	GPS#

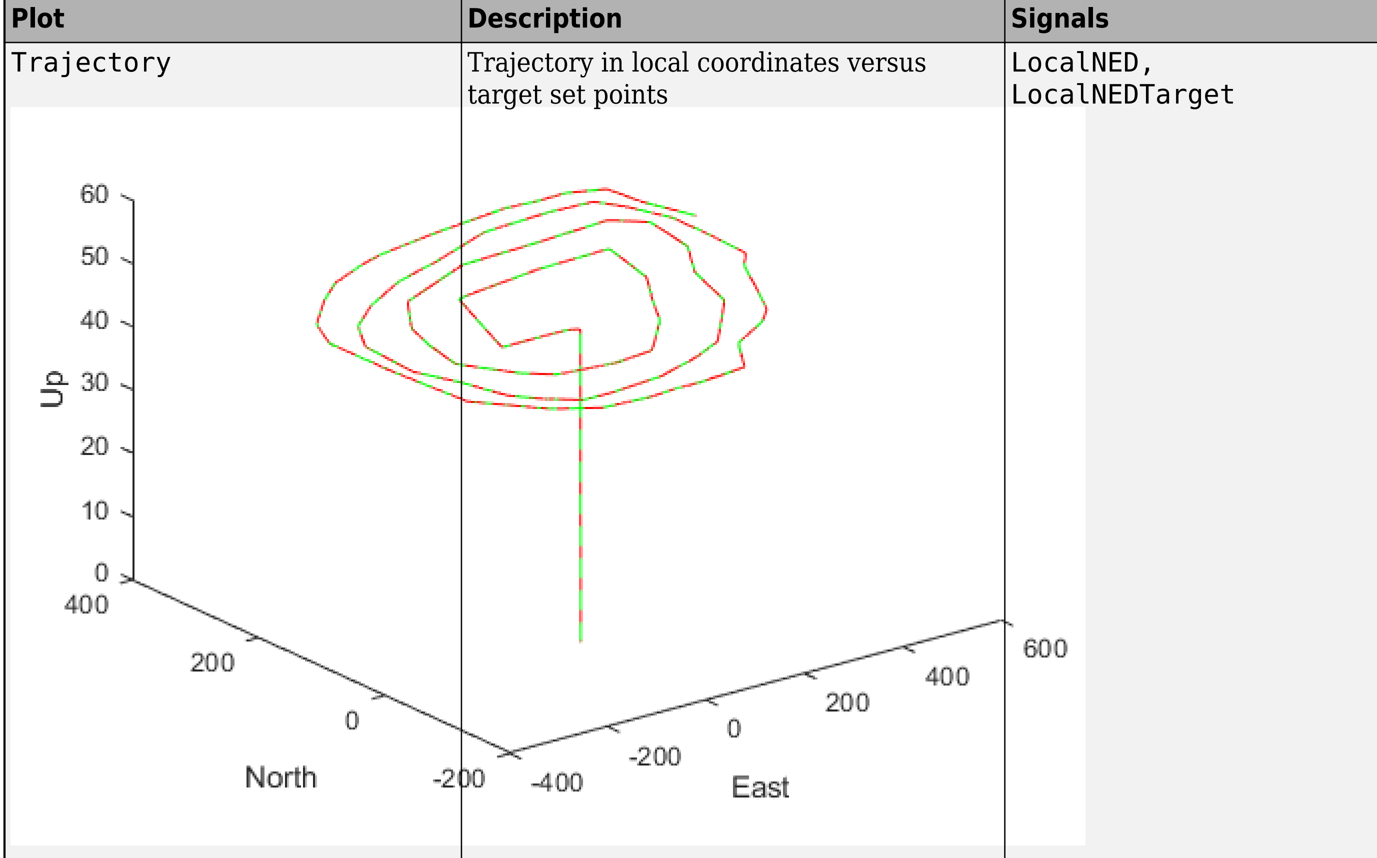


Plot	Description	Signals
`Height` 	Stacked plots of barometer reading, GPS altitude reading, and fused height estimate	`Barometer#`, `GPS#`, `LocalNED`

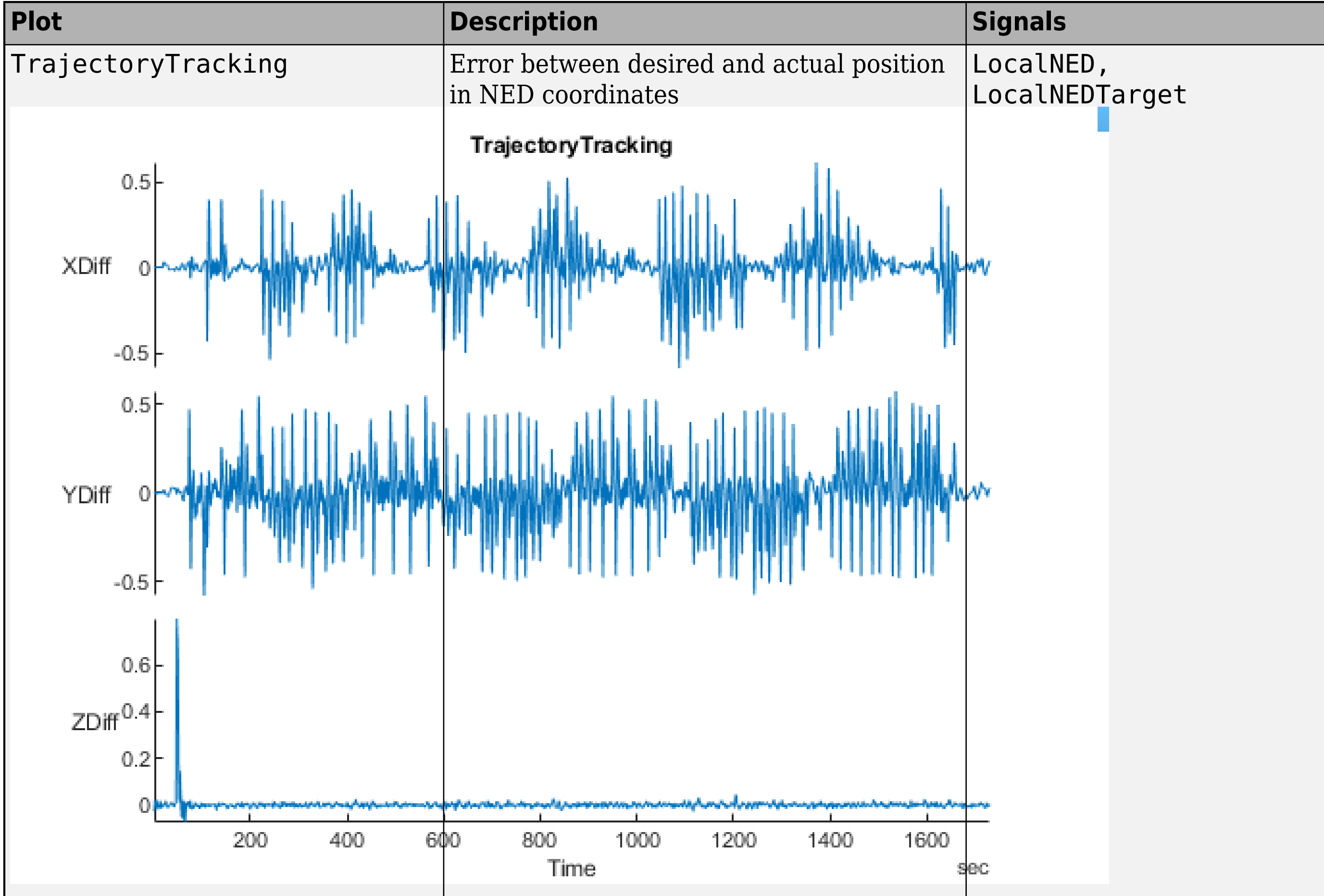
Plot	Description	Signals
Speed	Stacked plot of ground velocity and air speed	GPS#, Airspeed#



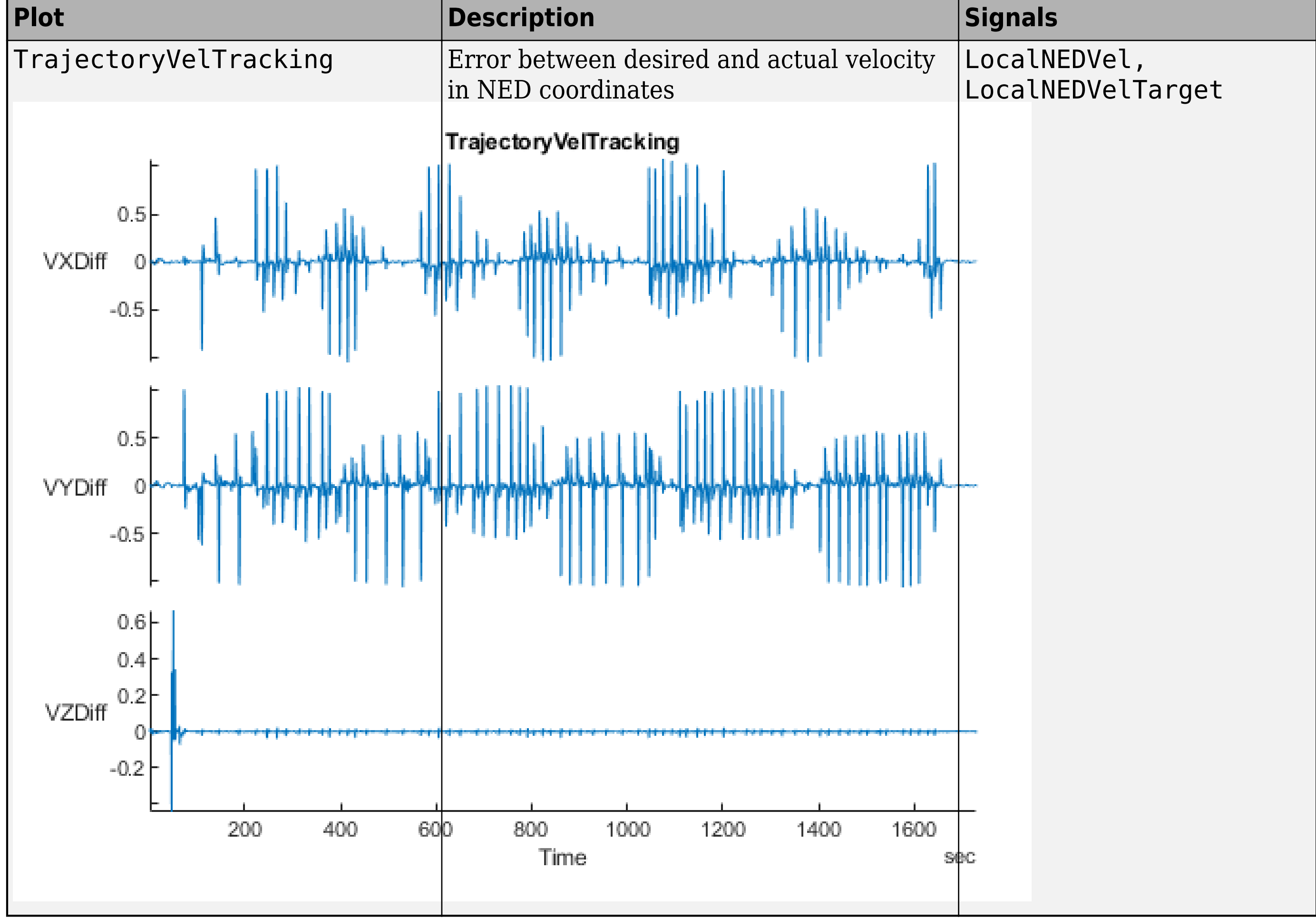
Plot	Description	Signals
`Trajectory`	Trajectory in local coordinates versus target set points	`LocalNED`, `LocalNEDTarget`



Plot	Description	Signals
TrajectoryTracking	Error between desired and actual position in NED coordinates	LocalNED, LocalNEDTarget



Plot	Description	Signals
TrajectoryVelTracking	Error between desired and actual velocity in NED coordinates	LocalNEDVel, LocalNEDVelTarget



Version History

Introduced in R2020b

See Also

flightLogSignalMapping | mavlinktlog | extract | info | mapSignal | show

createcmd

Create MAVLink command message

Syntax

```
cmdMsg = createcmd(dialect,cmdSetting,cmdType)
```

Description

`cmdMsg = createcmd(dialect,cmdSetting,cmdType)` returns a blank `COMMAND_INT` or `COMMAND_LONG` message structure based on the command setting and type. The command definitions are contained in the `mavlinkdialect` object, `dialect`.

Examples

Parse and Use MAVLink Dialect

This example shows how to parse a MAVLink XML file and create messages and commands from the definitions.

NOTE: This example requires you to install the UAV Library for Robotics System Toolbox®. Call `roboticsAddons` to open the Add-ons Explorer and install the library.

Parse and store the MAVLink dialect XML. Specify the XML path. The default `"common.xml"` dialect is provided. This XML file contains all the message and enum definitions.

```
dialect = mavlinkdialect("common.xml");
```

Create a MAVLink command from the `MAV_CMD` enum, which is an enum of MAVLink commands to send to the UAV. Specify the setting as `"int"` or `"long"`, and the type as an integer or string.

```
cmdMsg = createcmd(dialect,"long",22)
```

```
cmdMsg = struct with fields:
      MsgID: 76
    Payload: [1x1 struct]
```

Verify the command name using `num2enum`. Command 22 is a take-off command for the UAV. You can convert back to an ID using `enum2num`. Your dialect can contain many different enums with different names and IDs.

```
cmdName = num2enum(dialect,"MAV_CMD",22)
```

```
cmdName = 
"MAV_CMD_NAV_TAKEOFF"
```

```
cmdID = enum2num(dialect,"MAV_CMD",cmdName)
```

```
cmdID = 22
```

Use `enuminfo` to view the table of the `MAV_CMD` enum entries.

```
info = enuminfo(dialect,"MAV_CMD");
info.Entries{:}
```

```
ans=148×3 table
                    Name                     Value
    _____________________________________    _____    ___________________________________________

    "MAV_CMD_NAV_WAYPOINT"                    16      "Navigate to waypoint."
    "MAV_CMD_NAV_LOITER_UNLIM"                17      "Loiter around this waypoint an unlimited a
    "MAV_CMD_NAV_LOITER_TURNS"                18      "Loiter around this waypoint for X turns"
    "MAV_CMD_NAV_LOITER_TIME"                 19      "Loiter at the specified latitude, longitud
    "MAV_CMD_NAV_RETURN_TO_LAUNCH"            20      "Return to launch location"
    "MAV_CMD_NAV_LAND"                        21      "Land at location."
    "MAV_CMD_NAV_TAKEOFF"                     22      "Takeoff from ground / hand. Vehicles that
    "MAV_CMD_NAV_LAND_LOCAL"                  23      "Land at local position (local frame only)"
    "MAV_CMD_NAV_TAKEOFF_LOCAL"               24      "Takeoff from local position (local frame o
    "MAV_CMD_NAV_FOLLOW"                      25      "Vehicle following, i.e. this waypoint repr
    "MAV_CMD_NAV_CONTINUE_AND_CHANGE_ALT"     30      "Continue on the current course and climb/d
    "MAV_CMD_NAV_LOITER_TO_ALT"               31      "Begin loiter at the specified Latitude and
    "MAV_CMD_DO_FOLLOW"                       32      "Begin following a target"
    "MAV_CMD_DO_FOLLOW_REPOSITION"            33      "Reposition the MAV after a follow target c
    "MAV_CMD_DO_ORBIT"                        34      "Start orbiting on the circumference of a c
    "MAV_CMD_NAV_ROI"                         80      "Sets the region of interest (ROI) for a se
      ⋮
```

Query the dialect for a specific message ID. Create a blank MAVLink message using the message ID.

```
info = msginfo(dialect,"HEARTBEAT")
```

```
info=1×4 table
    MessageID    MessageName
    _________    ___________    ___________________________________________________________________

        0        "HEARTBEAT"    "The heartbeat message shows that a system or component is presen
```

```
msg = createmsg(dialect,info.MessageID);
```

Input Arguments

dialect — MAVLink dialect

`mavlinkdialect` object

MAVLink dialect, specified as a `mavlinkdialect` object. The dialect specifies the message structure for the MAVLink protocol.

cmdSetting — Command setting

`"int"` | `"long"`

Command setting, specified as either `"int"` or `"long"` for either a `COMMAND_INT` or `COMMAND_LONG` command.

cmdType — Command type

positive integer | string

Command type, specified as either a positive integer or string. If specified as an integer, the command definition with the matching ID from the `MAV_CMD` enum in `dialect` is returned. If specified as a string, the command with the matching name is returned.

To get the command types for the `MAV_CMD` enum, use `enuminfo`:

```
enumTable = enuminfo(dialect,"MAV_CMD")
enumTable.Entries{1}
```

Output Arguments

`cmdMsg` — MAVLink command message
structure

MAVLink command message, returned as a structure with the fields:

- `MsgID`: Positive integer for message ID.
- `Payload`: Structure containing fields for the specific message definition.

Version History

Introduced in R2019a

See Also

Functions
`createmsg` | `msginfo` | `enuminfo` | `enum2num` | `num2enum`

Objects
`mavlinkdialect` | `mavlinkio` | `mavlinkclient` | `mavlinksub`

createmsg

Create MAVLink message

Syntax

```
msg = createmsg(dialect,msgID)
```

Description

`msg = createmsg(dialect,msgID)` returns a blank message structure based on the message definitions specified in the `mavlinkdialect` object, `dialect`, and the input message ID, `msgID`.

Examples

Parse and Use MAVLink Dialect

This example shows how to parse a MAVLink XML file and create messages and commands from the definitions.

NOTE: This example requires you to install the UAV Library for Robotics System Toolbox®. Call `roboticsAddons` to open the Add-ons Explorer and install the library.

Parse and store the MAVLink dialect XML. Specify the XML path. The default `"common.xml"` dialect is provided. This XML file contains all the message and enum definitions.

```
dialect = mavlinkdialect("common.xml");
```

Create a MAVLink command from the `MAV_CMD` enum, which is an enum of MAVLink commands to send to the UAV. Specify the setting as `"int"` or `"long"`, and the type as an integer or string.

```
cmdMsg = createcmd(dialect,"long",22)
```

```
cmdMsg = struct with fields:
      MsgID: 76
    Payload: [1x1 struct]
```

Verify the command name using `num2enum`. Command 22 is a take-off command for the UAV. You can convert back to an ID using `enum2num`. Your dialect can contain many different enums with different names and IDs.

```
cmdName = num2enum(dialect,"MAV_CMD",22)
```

```
cmdName = 
"MAV_CMD_NAV_TAKEOFF"
```

```
cmdID = enum2num(dialect,"MAV_CMD",cmdName)
```

```
cmdID = 22
```

Use `enuminfo` to view the table of the `MAV_CMD` enum entries.

```
info = enuminfo(dialect,"MAV_CMD");
info.Entries{:}
```

```
ans=148×3 table
                   Name                      Value
    ____________________________________    _____    __________________________________________

    "MAV_CMD_NAV_WAYPOINT"                   16      "Navigate to waypoint."
    "MAV_CMD_NAV_LOITER_UNLIM"               17      "Loiter around this waypoint an unlimited a
    "MAV_CMD_NAV_LOITER_TURNS"               18      "Loiter around this waypoint for X turns"
    "MAV_CMD_NAV_LOITER_TIME"                19      "Loiter at the specified latitude, longitud
    "MAV_CMD_NAV_RETURN_TO_LAUNCH"           20      "Return to launch location"
    "MAV_CMD_NAV_LAND"                       21      "Land at location."
    "MAV_CMD_NAV_TAKEOFF"                    22      "Takeoff from ground / hand. Vehicles that
    "MAV_CMD_NAV_LAND_LOCAL"                 23      "Land at local position (local frame only)"
    "MAV_CMD_NAV_TAKEOFF_LOCAL"              24      "Takeoff from local position (local frame o
    "MAV_CMD_NAV_FOLLOW"                     25      "Vehicle following, i.e. this waypoint repr
    "MAV_CMD_NAV_CONTINUE_AND_CHANGE_ALT"    30      "Continue on the current course and climb/d
    "MAV_CMD_NAV_LOITER_TO_ALT"              31      "Begin loiter at the specified Latitude and
    "MAV_CMD_DO_FOLLOW"                      32      "Begin following a target"
    "MAV_CMD_DO_FOLLOW_REPOSITION"           33      "Reposition the MAV after a follow target c
    "MAV_CMD_DO_ORBIT"                       34      "Start orbiting on the circumference of a c
    "MAV_CMD_NAV_ROI"                        80      "Sets the region of interest (ROI) for a se
      ⋮
```

Query the dialect for a specific message ID. Create a blank MAVLink message using the message ID.

```
info = msginfo(dialect,"HEARTBEAT")
```

```
info=1×4 table
    MessageID    MessageName
    _________    ___________    ________________________________________________________________

        0        "HEARTBEAT"    "The heartbeat message shows that a system or component is presen
```

```
msg = createmsg(dialect,info.MessageID);
```

Input Arguments

dialect — MAVLink dialect

`mavlinkdialect` object

MAVLink dialect, specified as a `mavlinkdialect` object. The dialect specifies the message structure for the MAVLink protocol.

msgID — Message ID

positive integer | string

Message ID, specified as either a positive integer or string. If specified as an integer, the message definition with the matching ID from the `dialect` is returned. If specified as a string, the message with the matching name is returned.

Output Arguments

`msg` — MAVLink message
structure

MAVLink message, returned as a structure with the fields:

- `MsgID`: Positive integer for message ID.
- `Payload`: Structure containing fields for the specific message definition.

Version History

Introduced in R2019a

See Also

Functions
`createcmd` | `msginfo` | `enuminfo` | `enum2num` | `num2enum`

Objects
`mavlinkdialect` | `mavlinkio` | `mavlinkclient` | `mavlinksub`

Topics
"Tune UAV Parameters Using MAVLink Parameter Protocol"

enum2num

Enum value for given entry

Syntax

```
enumValue = enum2num(dialect,enum,entry)
```

Description

`enumValue = enum2num(dialect,enum,entry)` returns the value for the given entry in the enum.

Examples

Parse and Use MAVLink Dialect

This example shows how to parse a MAVLink XML file and create messages and commands from the definitions.

NOTE: This example requires you to install the UAV Library for Robotics System Toolbox®. Call `roboticsAddons` to open the Add-ons Explorer and install the library.

Parse and store the MAVLink dialect XML. Specify the XML path. The default `"common.xml"` dialect is provided. This XML file contains all the message and enum definitions.

```
dialect = mavlinkdialect("common.xml");
```

Create a MAVLink command from the `MAV_CMD` enum, which is an enum of MAVLink commands to send to the UAV. Specify the setting as `"int"` or `"long"`, and the type as an integer or string.

```
cmdMsg = createcmd(dialect,"long",22)
```

```
cmdMsg = struct with fields:
      MsgID: 76
    Payload: [1x1 struct]
```

Verify the command name using `num2enum`. Command 22 is a take-off command for the UAV. You can convert back to an ID using `enum2num`. Your dialect can contain many different enums with different names and IDs.

```
cmdName = num2enum(dialect,"MAV_CMD",22)
```

```
cmdName = 
"MAV_CMD_NAV_TAKEOFF"
```

```
cmdID = enum2num(dialect,"MAV_CMD",cmdName)
```

```
cmdID = 22
```

Use `enuminfo` to view the table of the `MAV_CMD` enum entries.

```
info = enuminfo(dialect,"MAV_CMD");
info.Entries{:}
```

```
ans=148×3 table
                 Name                     Value
    _____________________________________    _____    __________________________________________

    "MAV_CMD_NAV_WAYPOINT"                     16      "Navigate to waypoint."
    "MAV_CMD_NAV_LOITER_UNLIM"                 17      "Loiter around this waypoint an unlimited a
    "MAV_CMD_NAV_LOITER_TURNS"                 18      "Loiter around this waypoint for X turns"
    "MAV_CMD_NAV_LOITER_TIME"                  19      "Loiter at the specified latitude, longitud
    "MAV_CMD_NAV_RETURN_TO_LAUNCH"             20      "Return to launch location"
    "MAV_CMD_NAV_LAND"                         21      "Land at location."
    "MAV_CMD_NAV_TAKEOFF"                      22      "Takeoff from ground / hand. Vehicles that
    "MAV_CMD_NAV_LAND_LOCAL"                   23      "Land at local position (local frame only)"
    "MAV_CMD_NAV_TAKEOFF_LOCAL"                24      "Takeoff from local position (local frame o
    "MAV_CMD_NAV_FOLLOW"                       25      "Vehicle following, i.e. this waypoint repr
    "MAV_CMD_NAV_CONTINUE_AND_CHANGE_ALT"      30      "Continue on the current course and climb/d
    "MAV_CMD_NAV_LOITER_TO_ALT"                31      "Begin loiter at the specified Latitude and
    "MAV_CMD_DO_FOLLOW"                        32      "Begin following a target"
    "MAV_CMD_DO_FOLLOW_REPOSITION"             33      "Reposition the MAV after a follow target c
    "MAV_CMD_DO_ORBIT"                         34      "Start orbiting on the circumference of a c
    "MAV_CMD_NAV_ROI"                          80      "Sets the region of interest (ROI) for a se
      ⋮
```

Query the dialect for a specific message ID. Create a blank MAVLink message using the message ID.

```
info = msginfo(dialect,"HEARTBEAT")
```

```
info=1×4 table
    MessageID    MessageName
    _________    ___________    ______________________________________________________________

        0        "HEARTBEAT"    "The heartbeat message shows that a system or component is presen
```

```
msg = createmsg(dialect,info.MessageID);
```

Input Arguments

`dialect` — MAVLink dialect

`mavlinkdialect` object

MAVLink dialect, specified as a `mavlinkdialect` object, which contains a parsed dialect XML for MAVLink message definitions.

`enum` — MAVLink enum name

string

MAVLink enum name, specified as a string.

`entry` — MAVLink enum entry name

string

MAVLink enum entry name, specified as a string.

Output Arguments

`enumValue` — Enum value
integer

Enum value, returned as an integer.

Version History

Introduced in R2019a

See Also

`num2enum` | `enuminfo` | `msginfo` | `mavlinkdialect` | `mavlinkio` | `mavlinkclient` | `mavlinksub`

External Websites
MAVLink Developer Guide

enuminfo

Enum definition for enum ID

Syntax

```
enumTable = enuminfo(dialect,enumID)
```

Description

`enumTable = enuminfo(dialect,enumID)` returns a table detailing the enumeration definition based on the given `enumID`.

Examples

Parse and Use MAVLink Dialect

This example shows how to parse a MAVLink XML file and create messages and commands from the definitions.

NOTE: This example requires you to install the UAV Library for Robotics System Toolbox®. Call `roboticsAddons` to open the Add-ons Explorer and install the library.

Parse and store the MAVLink dialect XML. Specify the XML path. The default `"common.xml"` dialect is provided. This XML file contains all the message and enum definitions.

```
dialect = mavlinkdialect("common.xml");
```

Create a MAVLink command from the `MAV_CMD` enum, which is an enum of MAVLink commands to send to the UAV. Specify the setting as `"int"` or `"long"`, and the type as an integer or string.

```
cmdMsg = createcmd(dialect,"long",22)
```

```
cmdMsg = struct with fields:
      MsgID: 76
    Payload: [1x1 struct]
```

Verify the command name using `num2enum`. Command 22 is a take-off command for the UAV. You can convert back to an ID using `enum2num`. Your dialect can contain many different enums with different names and IDs.

```
cmdName = num2enum(dialect,"MAV_CMD",22)
```

```
cmdName = 
"MAV_CMD_NAV_TAKEOFF"
```

```
cmdID = enum2num(dialect,"MAV_CMD",cmdName)
```

```
cmdID = 22
```

Use `enuminfo` to view the table of the `MAV_CMD` enum entries.

```
info = enuminfo(dialect,"MAV_CMD");
info.Entries{:}
```

```
ans=148×3 table
                    Name                        Value
    ____________________________________        _____    ___________________________________________

    "MAV_CMD_NAV_WAYPOINT"                       16      "Navigate to waypoint."
    "MAV_CMD_NAV_LOITER_UNLIM"                   17      "Loiter around this waypoint an unlimited a
    "MAV_CMD_NAV_LOITER_TURNS"                   18      "Loiter around this waypoint for X turns"
    "MAV_CMD_NAV_LOITER_TIME"                    19      "Loiter at the specified latitude, longitud
    "MAV_CMD_NAV_RETURN_TO_LAUNCH"               20      "Return to launch location"
    "MAV_CMD_NAV_LAND"                           21      "Land at location."
    "MAV_CMD_NAV_TAKEOFF"                        22      "Takeoff from ground / hand. Vehicles that
    "MAV_CMD_NAV_LAND_LOCAL"                     23      "Land at local position (local frame only)"
    "MAV_CMD_NAV_TAKEOFF_LOCAL"                  24      "Takeoff from local position (local frame o
    "MAV_CMD_NAV_FOLLOW"                         25      "Vehicle following, i.e. this waypoint repr
    "MAV_CMD_NAV_CONTINUE_AND_CHANGE_ALT"        30      "Continue on the current course and climb/d
    "MAV_CMD_NAV_LOITER_TO_ALT"                  31      "Begin loiter at the specified Latitude and
    "MAV_CMD_DO_FOLLOW"                          32      "Begin following a target"
    "MAV_CMD_DO_FOLLOW_REPOSITION"               33      "Reposition the MAV after a follow target c
    "MAV_CMD_DO_ORBIT"                           34      "Start orbiting on the circumference of a c
    "MAV_CMD_NAV_ROI"                            80      "Sets the region of interest (ROI) for a se
      ⋮
```

Query the dialect for a specific message ID. Create a blank MAVLink message using the message ID.

```
info = msginfo(dialect,"HEARTBEAT")
```

```
info=1×4 table
    MessageID    MessageName
    _________    ___________    ______________________________________________________________

        0        "HEARTBEAT"    "The heartbeat message shows that a system or component is presen
```

```
msg = createmsg(dialect,info.MessageID);
```

Input Arguments

dialect — MAVLink dialect

mavlinkdialect object

MAVLink dialect, specified as a `mavlinkdialect` object, which contains a parsed dialect XML for MAVLink message definitions.

enumID — MAVLink enum ID

string

MAVLink enum ID, specified as a string.

Output Arguments

enumTable — Enum definition

table

Enum definition, returned as a table containing the message ID, name, description, and entries. The entries are given as another table with their own information listed. All this information is defined by dialect XML file.

Version History

Introduced in R2019a

See Also

`msginfo` | `mavlinkdialect` | `mavlinkio` | `mavlinkclient` | `mavlinksub`

External Websites

MAVLink Developer Guide

msginfo

Message definition for message ID

Syntax

```
msgTable = msginfo(dialect,messageID)
```

Description

`msgTable = msginfo(dialect,messageID)` returns a table detailing the message definition based on the given `messageID`.

Examples

Parse and Use MAVLink Dialect

This example shows how to parse a MAVLink XML file and create messages and commands from the definitions.

NOTE: This example requires you to install the UAV Library for Robotics System Toolbox®. Call `roboticsAddons` to open the Add-ons Explorer and install the library.

Parse and store the MAVLink dialect XML. Specify the XML path. The default `"common.xml"` dialect is provided. This XML file contains all the message and enum definitions.

```
dialect = mavlinkdialect("common.xml");
```

Create a MAVLink command from the `MAV_CMD` enum, which is an enum of MAVLink commands to send to the UAV. Specify the setting as `"int"` or `"long"`, and the type as an integer or string.

```
cmdMsg = createcmd(dialect,"long",22)
```

```
cmdMsg = struct with fields:
      MsgID: 76
    Payload: [1x1 struct]
```

Verify the command name using `num2enum`. Command 22 is a take-off command for the UAV. You can convert back to an ID using `enum2num`. Your dialect can contain many different enums with different names and IDs.

```
cmdName = num2enum(dialect,"MAV_CMD",22)
```

```
cmdName = 
"MAV_CMD_NAV_TAKEOFF"
```

```
cmdID = enum2num(dialect,"MAV_CMD",cmdName)
```

```
cmdID = 22
```

Use `enuminfo` to view the table of the `MAV_CMD` enum entries.

```
info = enuminfo(dialect,"MAV_CMD");
info.Entries{:}
```

```
ans=148×3 table
                   Name                         Value
    ____________________________________      _____    ____________________________________________

    "MAV_CMD_NAV_WAYPOINT"                      16      "Navigate to waypoint."
    "MAV_CMD_NAV_LOITER_UNLIM"                  17      "Loiter around this waypoint an unlimited a
    "MAV_CMD_NAV_LOITER_TURNS"                  18      "Loiter around this waypoint for X turns"
    "MAV_CMD_NAV_LOITER_TIME"                   19      "Loiter at the specified latitude, longitud
    "MAV_CMD_NAV_RETURN_TO_LAUNCH"              20      "Return to launch location"
    "MAV_CMD_NAV_LAND"                          21      "Land at location."
    "MAV_CMD_NAV_TAKEOFF"                       22      "Takeoff from ground / hand. Vehicles that
    "MAV_CMD_NAV_LAND_LOCAL"                    23      "Land at local position (local frame only)"
    "MAV_CMD_NAV_TAKEOFF_LOCAL"                 24      "Takeoff from local position (local frame o
    "MAV_CMD_NAV_FOLLOW"                        25      "Vehicle following, i.e. this waypoint repr
    "MAV_CMD_NAV_CONTINUE_AND_CHANGE_ALT"       30      "Continue on the current course and climb/d
    "MAV_CMD_NAV_LOITER_TO_ALT"                 31      "Begin loiter at the specified Latitude and
    "MAV_CMD_DO_FOLLOW"                         32      "Begin following a target"
    "MAV_CMD_DO_FOLLOW_REPOSITION"              33      "Reposition the MAV after a follow target c
    "MAV_CMD_DO_ORBIT"                          34      "Start orbiting on the circumference of a c
    "MAV_CMD_NAV_ROI"                           80      "Sets the region of interest (ROI) for a se
      ⋮
```

Query the dialect for a specific message ID. Create a blank MAVLink message using the message ID.

```
info = msginfo(dialect,"HEARTBEAT")
```

```
info=1×4 table
    MessageID    MessageName
    _________    ___________    ________________________________________________________________

        0        "HEARTBEAT"    "The heartbeat message shows that a system or component is presen
```

```
msg = createmsg(dialect,info.MessageID);
```

Input Arguments

dialect — MAVLink dialect

`mavlinkdialect` object

MAVLink dialect, specified as a `mavlinkdialect` object, which contains a parsed dialect XML for MAVLink message definitions.

messageID — MAVLink message ID or name

integer | string

MAVLink message ID or name, specified as an integer or string.

Output Arguments

msgTable — Message definition

table

Message definition, returned as a table containing the message ID, name, description, and fields. The fields are given as another table with their own information. All this information is defined by dialect XML file.

Version History

Introduced in R2019a

See Also

`createmsg` | `enuminfo` | `mavlinkdialect` | `mavlinkio` | `mavlinkclient` | `mavlinksub`

External Websites

MAVLink Developer Guide

connect

Connect to MAVLink clients through UDP port

Syntax

```
connectionName = connect(mavlink,"UDP")
connectionName = connect( ___ ,Name,Value)
```

Description

`connectionName = connect(mavlink,"UDP")` connects to the `mavlinkio` client through a UDP port.

`connectionName = connect( ___ ,Name,Value)` additionally specifies arguments using name-value pairs.

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Examples

Store MAVLink Client Information

Connect to a MAVLink client.

```
mavlink = mavlinkio("common.xml");
connect(mavlink,"UDP");
```

Create the object for storing the client information. Specify the system and component ID.

```
client = mavlinkclient(mavlink,1,1)
```

```
client = 
  mavlinkclient with properties:

         SystemID: 1
      ComponentID: 1
    ComponentType: "Unknown"
    AutopilotType: "Unknown"
```

Disconnect from client.

```
disconnect(mavlink)
```

Work with MAVLink Connection

This example shows how to connect to MAVLink clients, inspect the list of topics, connections, and clients, and send messages through UDP ports using the MAVLink communication protocol.

Connect to a MAVLink client using the `"common.xml"` dialect. This local client communicates with any other clients through a UDP port.

```
dialect = mavlinkdialect("common.xml");
mavlink = mavlinkio(dialect);
connect(mavlink,"UDP")
```

```
ans = 
"Connection1"
```

You can list all the active clients, connections, and topics for the MAVLink connection. Currently, there is only one client connection and no topics have received messages.

```
listClients(mavlink)
```

```
ans=1×4 table
    SystemID    ComponentID    ComponentType          AutopilotType       
    ________    ___________    _______________    _________________________

      255            1         "MAV_TYPE_GCS"     "MAV_AUTOPILOT_INVALID"
```

```
listConnections(mavlink)
```

```
ans=1×2 table
    ConnectionName      ConnectionInfo    
    ______________    ___________________

    "Connection1"     "UDP@0.0.0.0:53894"
```

```
listTopics(mavlink)
```

```
ans =

  0x5 empty table
```

Create a subscriber for receiving messages on the client. This subscriber listens for the `"HEARTBEAT"` message topic with ID equal to `0`.

```
sub = mavlinksub(mavlink,0);
```

Create a `"HEARTBEAT"` message using the `mavlinkdialect` object. Specify payload information and send the message over the MAVLink client.

```
msg = createmsg(dialect,"HEARTBEAT");
msg.Payload.type(:) = enum2num(dialect,'MAV_TYPE','MAV_TYPE_QUADROTOR');
sendmsg(mavlink,msg)
```

Disconnect from the client.

```
disconnect(mavlink)
```

Input Arguments

`mavlink` — MAVLink client connection
`mavlinkio` object

MAVLink client connection, specified as a `mavlinkio` object.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `'LocalPort',12345`

`ConnectionName` — Identifying connection name
`"Connection#"` (default) | string scalar

Identifying connection name, specified as the comma-separated pair consisting of `'ConnectionName'` and a string scalar. The default connection name is `"Connection#"`.

Data Types: `string`

`LocalPort` — Local port for UDP connection
`0` (default) | numeric scalar

Local port for UDP connection, specified as a numeric scalar. A value of `0` binds to a random open port.

Data Types: `double`

Output Arguments

`connectionName` — Identifying connection name
`"Connection#"` (default) | string scalar

Identifying connection name, specified as a string scalar. The default connection name is `"Connection#"`, where `#` is an integer starting at `1` and increases with each new connection created.

Data Types: `string`

Version History

Introduced in R2019a

See Also

`disconnect` | `mavlinkdialect` | `mavlinkclient` | `mavlinksub`

Topics

"Tune UAV Parameters Using MAVLink Parameter Protocol"

External Websites

MAVLink Developer Guide

disconnect

Disconnect from MAVLink clients

Syntax

```
disconnect(mavlink)
disconnect(mavlink,connection)
```

Description

`disconnect(mavlink)` disconnects from all MAVLink clients connected through the `mavlinkio` client.

`disconnect(mavlink,connection)` disconnects from the specific client connection name.

Examples

Store MAVLink Client Information

Connect to a MAVLink client.

```
mavlink = mavlinkio("common.xml");
connect(mavlink,"UDP");
```

Create the object for storing the client information. Specify the system and component ID.

```
client = mavlinkclient(mavlink,1,1)
```

```
client = 
  mavlinkclient with properties:

         SystemID: 1
      ComponentID: 1
    ComponentType: "Unknown"
    AutopilotType: "Unknown"
```

Disconnect from client.

```
disconnect(mavlink)
```

Work with MAVLink Connection

This example shows how to connect to MAVLink clients, inspect the list of topics, connections, and clients, and send messages through UDP ports using the MAVLink communication protocol.

Connect to a MAVLink client using the `"common.xml"` dialect. This local client communicates with any other clients through a UDP port.

```
dialect = mavlinkdialect("common.xml");
mavlink = mavlinkio(dialect);
connect(mavlink,"UDP")
```

```
ans = 
"Connection1"
```

You can list all the active clients, connections, and topics for the MAVLink connection. Currently, there is only one client connection and no topics have received messages.

```
listClients(mavlink)
```

ans=*1×4 table*

SystemID	ComponentID	ComponentType	AutopilotType
255	1	"MAV_TYPE_GCS"	"MAV_AUTOPILOT_INVALID"

```
listConnections(mavlink)
```

ans=*1×2 table*

ConnectionName	ConnectionInfo
"Connection1"	"UDP@0.0.0.0:53894"

```
listTopics(mavlink)
```

```
ans = 

  0x5 empty table
```

Create a subscriber for receiving messages on the client. This subscriber listens for the `"HEARTBEAT"` message topic with ID equal to `0`.

```
sub = mavlinksub(mavlink,0);
```

Create a `"HEARTBEAT"` message using the `mavlinkdialect` object. Specify payload information and send the message over the MAVLink client.

```
msg = createmsg(dialect,"HEARTBEAT");
msg.Payload.type(:) = enum2num(dialect,'MAV_TYPE','MAV_TYPE_QUADROTOR');
sendmsg(mavlink,msg)
```

Disconnect from the client.

```
disconnect(mavlink)
```

Input Arguments

`mavlink` — MAVLink client connection

`mavlinkio` object

MAVLink client connection, specified as a `mavlinkio` object.

`connection` — Connection name
string scalar

Connection name, specified as a string scalar.

Version History

Introduced in R2019a

See Also

`connect` | `mavlinkio` | `mavlinkdialect` | `mavlinkclient` | `mavlinksub`

Topics

"Tune UAV Parameters Using MAVLink Parameter Protocol"

External Websites

MAVLink Developer Guide

listClients

List all connected MAVLink clients

Syntax

```
clientTable = listClients(mavlink)
```

Description

`clientTable = listClients(mavlink)` lists all active connections for the `mavlinkio` client connection.

Examples

Work with MAVLink Connection

This example shows how to connect to MAVLink clients, inspect the list of topics, connections, and clients, and send messages through UDP ports using the MAVLink communication protocol.

Connect to a MAVLink client using the `"common.xml"` dialect. This local client communicates with any other clients through a UDP port.

```
dialect = mavlinkdialect("common.xml");
mavlink = mavlinkio(dialect);
connect(mavlink,"UDP")
```

```
ans =
"Connection1"
```

You can list all the active clients, connections, and topics for the MAVLink connection. Currently, there is only one client connection and no topics have received messages.

```
listClients(mavlink)
```

```
ans=1×4 table
    SystemID    ComponentID    ComponentType          AutopilotType
    ________    ___________    ______________    _______________________

      255            1         "MAV_TYPE_GCS"    "MAV_AUTOPILOT_INVALID"
```

```
listConnections(mavlink)
```

```
ans=1×2 table
    ConnectionName      ConnectionInfo
    ______________    ___________________

    "Connection1"     "UDP@0.0.0.0:53894"
```

```
listTopics(mavlink)
```

```
ans =

  0x5 empty table
```

Create a subscriber for receiving messages on the client. This subscriber listens for the "HEARTBEAT" message topic with ID equal to 0.

```
sub = mavlinksub(mavlink,0);
```

Create a "HEARTBEAT" message using the mavlinkdialect object. Specify payload information and send the message over the MAVLink client.

```
msg = createmsg(dialect,"HEARTBEAT");
msg.Payload.type(:) = enum2num(dialect,'MAV_TYPE','MAV_TYPE_QUADROTOR');
sendmsg(mavlink,msg)
```

Disconnect from the client.

```
disconnect(mavlink)
```

Input Arguments

mavlink — MAVLink client connection
mavlinkio object

MAVLink client connection, specified as a mavlinkio object.

Output Arguments

clientTable — Active client info
table

Active connection info, returned as a table with SystemID, ComponentID, ConnectionType, and AutopilotType fields for each active client.

Version History

Introduced in R2019a

See Also

connect | listConnections | listTopics | mavlinkio | mavlinkdialect | mavlinkclient | mavlinksub

External Websites
MAVLink Developer Guide

listConnections

List all active MAVLink connections

Syntax

```
connectionTable = listConnections(mavlink)
```

Description

`connectionTable = listConnections(mavlink)` lists all active connections for the `mavlinkio` client connection.

Examples

Work with MAVLink Connection

This example shows how to connect to MAVLink clients, inspect the list of topics, connections, and clients, and send messages through UDP ports using the MAVLink communication protocol.

Connect to a MAVLink client using the `"common.xml"` dialect. This local client communicates with any other clients through a UDP port.

```
dialect = mavlinkdialect("common.xml");
mavlink = mavlinkio(dialect);
connect(mavlink,"UDP")
```

```
ans = 
"Connection1"
```

You can list all the active clients, connections, and topics for the MAVLink connection. Currently, there is only one client connection and no topics have received messages.

```
listClients(mavlink)
```

```
ans=1×4 table
    SystemID    ComponentID    ComponentType          AutopilotType      
    ________    ___________    ______________    _______________________

      255            1         "MAV_TYPE_GCS"    "MAV_AUTOPILOT_INVALID"
```

```
listConnections(mavlink)
```

```
ans=1×2 table
    ConnectionName       ConnectionInfo    
    ______________    ___________________

    "Connection1"     "UDP@0.0.0.0:53894"
```

```
listTopics(mavlink)
```

```
ans =

  0x5 empty table
```

Create a subscriber for receiving messages on the client. This subscriber listens for the "HEARTBEAT" message topic with ID equal to 0.

```
sub = mavlinksub(mavlink,0);
```

Create a "HEARTBEAT" message using the mavlinkdialect object. Specify payload information and send the message over the MAVLink client.

```
msg = createmsg(dialect,"HEARTBEAT");
msg.Payload.type(:) = enum2num(dialect,'MAV_TYPE','MAV_TYPE_QUADROTOR');
sendmsg(mavlink,msg)
```

Disconnect from the client.

```
disconnect(mavlink)
```

Input Arguments

mavlink — MAVLink client connection
mavlinkio object

MAVLink client connection, specified as a mavlinkio object.

Output Arguments

connectionTable — Active connection info
table

Active connection info, returned as a table with ConnectionName and ConnectionInfo fields for each active connection.

Version History

Introduced in R2019a

See Also

connect | listClients | listTopics | mavlinkio | mavlinkdialect | mavlinkclient | mavlinksub

External Websites

MAVLink Developer Guide

listTopics

List all topics received by MAVLink client

Syntax

`topicTable = listTopics(mavlink)`

Description

`topicTable = listTopics(mavlink)` returns a table of topics received on the connected `mavlinkio` client with information on the message frequency.

Examples

Work with MAVLink Connection

This example shows how to connect to MAVLink clients, inspect the list of topics, connections, and clients, and send messages through UDP ports using the MAVLink communication protocol.

Connect to a MAVLink client using the `"common.xml"` dialect. This local client communicates with any other clients through a UDP port.

```
dialect = mavlinkdialect("common.xml");
mavlink = mavlinkio(dialect);
connect(mavlink,"UDP")
```

```
ans = 
"Connection1"
```

You can list all the active clients, connections, and topics for the MAVLink connection. Currently, there is only one client connection and no topics have received messages.

```
listClients(mavlink)
```

```
ans=1×4 table
    SystemID    ComponentID    ComponentType          AutopilotType      
    ________    ___________    ______________    ________________________

      255            1         "MAV_TYPE_GCS"    "MAV_AUTOPILOT_INVALID"
```

```
listConnections(mavlink)
```

```
ans=1×2 table
    ConnectionName       ConnectionInfo    
    ______________    ____________________

    "Connection1"     "UDP@0.0.0.0:53894"
```

```
listTopics(mavlink)
```

```
ans =

  0x5 empty table
```

Create a subscriber for receiving messages on the client. This subscriber listens for the `"HEARTBEAT"` message topic with ID equal to `0`.

```
sub = mavlinksub(mavlink,0);
```

Create a `"HEARTBEAT"` message using the `mavlinkdialect` object. Specify payload information and send the message over the MAVLink client.

```
msg = createmsg(dialect,"HEARTBEAT");
msg.Payload.type(:) = enum2num(dialect,'MAV_TYPE','MAV_TYPE_QUADROTOR');
sendmsg(mavlink,msg)
```

Disconnect from the client.

```
disconnect(mavlink)
```

Input Arguments

`mavlink` — MAVLink client connection
`mavlinkio` object

MAVLink client connection, specified as a `mavlinkio` object.

Output Arguments

`topicTable` — Topic info
table

Topic info, returned as a table with `SystemID`, `ComponentID`, `MessageID`, `MessageName`, and `MessageFrequency` fields for each topic receiving messages on the client.

Version History

Introduced in R2019a

See Also

`connect` | `listConnections` | `listClients` | `mavlinkio` | `mavlinkdialect` | `mavlinkclient` | `mavlinksub`

External Websites

MAVLink Developer Guide

sendmsg

Send MAVLink message

Syntax

```
sendmsg(mavlink,msg)
sendmsg(mavlink,msg,client)
```

Description

`sendmsg(mavlink,msg)` sends a message to all connected MAVLink clients in the `mavlinkio` object.

`sendmsg(mavlink,msg,client)` sends a message to the MAVLink client specified as a `mavlinkclient` object. If the client is not connected, no message is sent.

Examples

Work with MAVLink Connection

This example shows how to connect to MAVLink clients, inspect the list of topics, connections, and clients, and send messages through UDP ports using the MAVLink communication protocol.

Connect to a MAVLink client using the `"common.xml"` dialect. This local client communicates with any other clients through a UDP port.

```
dialect = mavlinkdialect("common.xml");
mavlink = mavlinkio(dialect);
connect(mavlink,"UDP")
```

```
ans = 
"Connection1"
```

You can list all the active clients, connections, and topics for the MAVLink connection. Currently, there is only one client connection and no topics have received messages.

```
listClients(mavlink)
```

```
ans=1×4 table
    SystemID    ComponentID    ComponentType          AutopilotType      
    ________    ___________    ______________    ________________________

      255            1         "MAV_TYPE_GCS"    "MAV_AUTOPILOT_INVALID"
```

```
listConnections(mavlink)
```

```
ans=1×2 table
    ConnectionName      ConnectionInfo     
    ______________    ___________________
```

```
    "Connection1"     "UDP@0.0.0.0:53894"
```

```
listTopics(mavlink)
```

```
ans =

  0x5 empty table
```

Create a subscriber for receiving messages on the client. This subscriber listens for the "HEARTBEAT" message topic with ID equal to 0.

```
sub = mavlinksub(mavlink,0);
```

Create a "HEARTBEAT" message using the mavlinkdialect object. Specify payload information and send the message over the MAVLink client.

```
msg = createmsg(dialect,"HEARTBEAT");
msg.Payload.type(:) = enum2num(dialect,'MAV_TYPE','MAV_TYPE_QUADROTOR');
sendmsg(mavlink,msg)
```

Disconnect from the client.

```
disconnect(mavlink)
```

Input Arguments

mavlink — MAVLink client connection

mavlinkio object

MAVLink client connection, specified as a mavlinkio object.

msg — MAVLink message

structure

MAVLink message, specified as a structure with the fields:

- MsgID: Positive integer for message ID.
- Payload: Structure containing fields for the specific message definition.

To create a blank message, use the createmsg with a mavlinkdialect object.

client — MAVLink client information

mavlinkclient object

MAVLink client information, specified as a mavlinkclient object.

Version History

Introduced in R2019a

See Also

connect | listConnections | listClients | mavlinkio | mavlinkdialect | mavlinkclient | mavlinksub

Topics
"Tune UAV Parameters Using MAVLink Parameter Protocol"

External Websites
MAVLink Developer Guide

serializemsg

Serialize MAVLink message to binary buffer

Syntax

`buffer = serializemsg(mavlink,msg)`

Description

`buffer = serializemsg(mavlink,msg)` serializes a MAVLink message structure to a binary buffer for transmission. This buffer is for manual transmission using your own communication channel. To send over UDP, see `sendmsg`.

Input Arguments

`mavlink` — MAVLink client connection
`mavlinkio` object

MAVLink client connection, specified as a `mavlinkio` object.

`msg` — MAVLink message
structure

MAVLink message, specified as a structure with the fields:

- `MsgID`: Positive integer for message ID.
- `Payload`: Structure containing fields for the specific message definition.

To create a blank message, use the `createmsg` with a `mavlinkdialect` object.

Output Arguments

`buffer` — Serialized message
vector of `uint8` integers

Serialized messaged, returned as vector of `uint8` integers.

Data Types: `uint8`

Version History

Introduced in R2019a

See Also

`sendmsg` | `connect` | `listConnections` | `listClients` | `mavlinkio` | `mavlinkdialect` | `mavlinkclient` | `mavlinksub`

External Websites

MAVLink Developer Guide

sendudpmsg

Send MAVLink message to UDP port

Syntax

`sendudpmsg(mavlink,msg,remoteHost,remotePort)`

Description

`sendudpmsg(mavlink,msg,remoteHost,remotePort)` sends the message, `msg`, to the remote UDP port specified by the host name, `remoteHost`, and port number, `remotePort`.

Input Arguments

`mavlink` — MAVLink client connection
`mavlinkio` object

MAVLink client connection, specified as a `mavlinkio` object.

`msg` — MAVLink message
structure

MAVLink message, specified as a structure with the fields:

- `MsgID`: Positive integer for message ID.
- `Payload`: Structure containing fields for the specific message definition.

To create a blank message, use the `createmsg` with a `mavlinkdialect` object.

`remoteHost` — Remote host IP address
string

Remote host IP address, specified as a string.

Example: `"192.168.1.10"`

`remotePort` — Remote host port
five-digit numeric scalar

Remote host IP address, specified as a five-digit numeric scalar.

Example: `14550`

Version History

Introduced in R2019a

See Also

`sendmsg` | `connect` | `listConnections` | `listClients` | `mavlinkio` | `mavlinkdialect` | `mavlinkclient` | `mavlinksub`

Topics
"Tune UAV Parameters Using MAVLink Parameter Protocol"

External Websites
MAVLink Developer Guide

latestmsgs

Received messages from MAVLink subscriber

Syntax

```
msgs = latestmsgs(sub,count)
```

Description

`msgs = latestmsgs(sub,count)` returns the latest received messages for the `mavlinksub` object. The messages are in a structure array in reverse-chronological order with the most recent being first. If `count` is larger than the number of stored messages, the structure array contains only the number of stored messages.

Examples

Subscribe to MAVLink Topic

Connect to a MAVLink client.

```
mavlink = mavlinkio("common.xml")
```

```
mavlink = 
  mavlinkio with properties:

        Dialect: [1x1 mavlinkdialect]
    LocalClient: [1x1 struct]
```

```
connect(mavlink,"UDP")
```

```
ans = 
"Connection1"
```

Get the client information.

```
client = mavlinkclient(mavlink,1,1);
```

Subscribe to the `"HEARTBEAT"` topic.

```
heartbeat = mavlinksub(mavlink,client,'HEARTBEAT');
```

Get the latest message. You must wait for a message to be received. Currently, no heartbeat message has been received on the `mavlink` object.

```
latestmsgs(heartbeat,1)
```

```
ans = 

  1x0 empty struct array with fields:

    MsgID
```

```
SystemID
ComponentID
Payload
Seq
```

Disconnect from client.

```
disconnect(mavlink)
```

Input Arguments

`sub` — MAVLink subscriber
`mavlinksub` object

MAVLink subscriber, specified as a `mavlinksub` object.

`count` — Number of messages
positive integer

Number of messages, specified as a positive integer. If `count` is larger than the number of stored messages, the structure array is padded with empty structures.

Output Arguments

`msgs` — Recently received messages
structure array

Recently received messages, returned as a structure array. Each structure has the fields:

- `MsgID`
- `SystemID`
- `ComponentID`
- `Payload`

The `Payload` is a structure defined by the message definition for the MAVLink dialect.

If `count` is larger than the number of stored messages, the structure array contains only the number of stored messages..

Version History

Introduced in R2019a

See Also

`mavlinksub` | `mavlinkclient` | `mavlinkio` | `mavlinkdialect`

num2enum

Enum entry for given value

Syntax

```
entry = num2enum(dialect,enum,enumValue)
```

Description

`entry = num2enum(dialect,enum,enumValue)` returns the value for the given entry in the enum.

Examples

Parse and Use MAVLink Dialect

This example shows how to parse a MAVLink XML file and create messages and commands from the definitions.

NOTE: This example requires you to install the UAV Library for Robotics System Toolbox®. Call `roboticsAddons` to open the Add-ons Explorer and install the library.

Parse and store the MAVLink dialect XML. Specify the XML path. The default `"common.xml"` dialect is provided. This XML file contains all the message and enum definitions.

```
dialect = mavlinkdialect("common.xml");
```

Create a MAVLink command from the `MAV_CMD` enum, which is an enum of MAVLink commands to send to the UAV. Specify the setting as `"int"` or `"long"`, and the type as an integer or string.

```
cmdMsg = createcmd(dialect,"long",22)
```

```
cmdMsg = struct with fields:
      MsgID: 76
    Payload: [1x1 struct]
```

Verify the command name using `num2enum`. Command 22 is a take-off command for the UAV. You can convert back to an ID using `enum2num`. Your dialect can contain many different enums with different names and IDs.

```
cmdName = num2enum(dialect,"MAV_CMD",22)
```

```
cmdName = 
"MAV_CMD_NAV_TAKEOFF"
```

```
cmdID = enum2num(dialect,"MAV_CMD",cmdName)
```

```
cmdID = 22
```

Use `enuminfo` to view the table of the `MAV_CMD` enum entries.

```
info = enuminfo(dialect,"MAV_CMD");
info.Entries{:}
```

```
ans=148×3 table
                     Name                        Value
    _____________________________________    _____    ___________________________________________

    "MAV_CMD_NAV_WAYPOINT"                     16     "Navigate to waypoint."
    "MAV_CMD_NAV_LOITER_UNLIM"                 17     "Loiter around this waypoint an unlimited a
    "MAV_CMD_NAV_LOITER_TURNS"                 18     "Loiter around this waypoint for X turns"
    "MAV_CMD_NAV_LOITER_TIME"                  19     "Loiter at the specified latitude, longitud
    "MAV_CMD_NAV_RETURN_TO_LAUNCH"             20     "Return to launch location"
    "MAV_CMD_NAV_LAND"                         21     "Land at location."
    "MAV_CMD_NAV_TAKEOFF"                      22     "Takeoff from ground / hand. Vehicles that
    "MAV_CMD_NAV_LAND_LOCAL"                   23     "Land at local position (local frame only)"
    "MAV_CMD_NAV_TAKEOFF_LOCAL"                24     "Takeoff from local position (local frame o
    "MAV_CMD_NAV_FOLLOW"                       25     "Vehicle following, i.e. this waypoint repr
    "MAV_CMD_NAV_CONTINUE_AND_CHANGE_ALT"      30     "Continue on the current course and climb/d
    "MAV_CMD_NAV_LOITER_TO_ALT"                31     "Begin loiter at the specified Latitude and
    "MAV_CMD_DO_FOLLOW"                        32     "Begin following a target"
    "MAV_CMD_DO_FOLLOW_REPOSITION"             33     "Reposition the MAV after a follow target c
    "MAV_CMD_DO_ORBIT"                         34     "Start orbiting on the circumference of a c
    "MAV_CMD_NAV_ROI"                          80     "Sets the region of interest (ROI) for a se
      ⋮
```

Query the dialect for a specific message ID. Create a blank MAVLink message using the message ID.

```
info = msginfo(dialect,"HEARTBEAT")
```

```
info=1×4 table
    MessageID    MessageName
    _________    ___________    ____________________________________________________________

        0        "HEARTBEAT"    "The heartbeat message shows that a system or component is presen
```

```
msg = createmsg(dialect,info.MessageID);
```

Input Arguments

dialect — MAVLink dialect

`mavlinkdialect` object

MAVLink dialect, specified as a `mavlinkdialect` object, which contains a parsed dialect XML for MAVLink message definitions.

enum — MAVLink enum name

string

MAVLink enum name, specified as a string.

enumValue — Enum value

integer

Enum value, specified as an integer.

Output Arguments

`entry` — MAVLink enum entry name
string

MAVLink enum entry name, returned as a string.

Version History

Introduced in R2019a

See Also

`enum2num` | `enuminfo` | `msginfo` | `mavlinkdialect` | `mavlinkio` | `mavlinkclient` | `mavlinksub`

External Websites
MAVLink Developer Guide

addChannel

Add MAVLink signing channel

Syntax

`channel = addChannel(stream,sysid,compid,linkid,keyname)`

Description

`channel = addChannel(stream,sysid,compid,linkid,keyname)` adds a new channel `channel` defined by the system ID `sysid`, component ID `compid`, and link ID `linkid` to MAVLink signing stream `stream`.

Examples

Add and Remove MAVLink Signing Channels

Create a `mavlinksigning` object to store MAVLink signing channels.

```
stream = mavlinksigning;
```

Load and list the keys from the `keys.env` file.

```
addmavlinkkeys("keys.env");
lsmavlinkkeys
```

```
ans = 1x2 string
    "Key1"    "Key2"
```

Add channel with a system ID of 1, component ID of 2, link ID of 3.

```
addChannel(stream,1,2,3,"Key1")
```

```
ans = struct with fields:
          Stream: [1x1 mavlinksigning]
        SystemID: 1
     ComponentID: 2
          LinkID: 3
             Key: "Key1"
       Timestamp: 25782415012287
    CreationTime: 04-Mar-2023 01:49:10
```

Remove the same channel.

```
removeChannel(stream,1,2,3)
```

Input Arguments

`stream` — MAVLink signing stream
`mavlinksigning` object

MAVLink signing stream, specified as a `mavlinksigning` object.

`sysid` — System ID
integer in range [0, 255]

System ID, specified as an integer in the range [0, 255].

`compid` — Component ID
integer in range [0, 255]

Component ID, specified as an integer in the range [0, 255].

`linkid` — Link ID
integer in range [0, 255]

Link ID, specified as an integer in the range [0, 255].

keyname — MAVLink key
string scalar

MAVLink key, specified as a string scalar.

Output Arguments

`channel` — Added channel
structure

Added channel, returned as a structure containing these fields:

- `Stream` — Signing stream, specified by `stream`.
- `SystemID` — System ID, specified by `sysid`.
- `ComponentID` — Component ID, specified by `compid`.
- `LinkID` — Link ID, specified by `linkid`.
- `Key` — Key, specified by `keyname`.
- `Timestamp` — Time passed since `2015-01-01 UTC`. Unit is 10 microseconds.
- `CreationTime` — Time of creation, specified as a `datetime` array.

Version History

Introduced in R2022a

See Also

`mavlinksigning` | `removeChannel`

removeChannel

Remove MAVLink signing channel

Syntax

`removeChannel(stream,sysid,compid,linkid)`

Description

`removeChannel(stream,sysid,compid,linkid)` adds removes the channel defined by the system ID `sysid`, component ID `compid`, and link ID `linkid` from the MAVLink signing stream `stream`.

Examples

Add and Remove MAVLink Signing Channels

Create a `mavlinksigning` object to store MAVLink signing channels.

```
stream = mavlinksigning;
```

Load and list the keys from the `keys.env` file.

```
addmavlinkkeys("keys.env");
lsmavlinkkeys
```

```
ans = 1x2 string
    "Key1"    "Key2"
```

Add channel with a system ID of 1, component ID of 2, link ID of 3.

```
addChannel(stream,1,2,3,"Key1")
```

```
ans = struct with fields:
          Stream: [1x1 mavlinksigning]
        SystemID: 1
     ComponentID: 2
          LinkID: 3
             Key: "Key1"
       Timestamp: 25782415012287
    CreationTime: 04-Mar-2023 01:49:10
```

Remove the same channel.

```
removeChannel(stream,1,2,3)
```

Input Arguments

`stream` — MAVLink signing stream
`mavlinksigning` object

MAVLink signing stream, specified as a `mavlinksigning` object.

`sysid` — System ID
integer in range [0, 255]

System ID, specified as an integer in the range [0, 255].

`compid` — Component ID
integer in range [0, 255]

Component ID, specified as an integer in the range [0, 255].

`linkid` — Link ID
integer in range [0, 255]

Link ID, specified as an integer in the range [0, 255].

Version History

Introduced in R2022a

See Also

`mavlinksigning` | `addChannel`

readmsg

Read specific messages from TLOG file

Syntax

```
msgTable = readmsg(tlogReader)
msgTable = readmsg(tlogReader,Name,Value)
```

Description

`msgTable = readmsg(tlogReader)` reads all message data from the specified `mavlinkdialect` object and returns a table, `msgTable`, that contains all the messages separated by message type, system ID, and component ID.

`msgTable = readmsg(tlogReader,Name,Value)` reads specific messages based on the specified name-value pairs for filtering specific properties of the messages. You can filter by message name, system ID, component ID, and time.

Examples

Read Messages from MAVLink TLOG File

Load the TLOG file. Specify the relative path of the file name.

```
tlogReader = mavlinktlog("mavlink_flightlog.tlog")
```

```
tlogReader = 
  mavlinktlog with properties:

           FileName: "mavlink_flightlog"
            Dialect: [1x1 mavlinkdialect]
          StartTime: '2018-05-18 17:53:28.893 (0.000 Seconds)'
            EndTime: '2018-05-18 18:04:52.524 (683.631 Seconds)'
        NumMessages: 20370
    AvailableTopics: [5x5 table]
     NumPacketsLost: 251
```

Read the `"HEARTBEAT"` messages from the file.

```
msgData = readmsg(tlogReader,"MessageName","HEARTBEAT")
```

msgData=*1×6 table*

MessageID	MessageName	SystemID	ComponentID	Messages	Version
0	"HEARTBEAT"	5	1	{3897x6 timetable}	2

Input Arguments

`tlogReader` — MAVLink TLOG reader
`mavlinktlog` object

MAVLink TLOG reader, specified as a `mavlinktlog` object.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `'MessageID',22`

`MessageName` — Name of message in tlog
string scalar | character vector

Name of message in TLOG, specified as string scalar or character vector.

Data Types: `char` | `string`

`SystemID` — MAVLink system ID
positive integer from 1 through 255

MAVLink system ID, specified as a positive integer from 1 through 255. MAVLink protocol only supports up to 255 systems. Usually, each UAV has its own system ID, but multiple UAVs could be considered one system.

`ComponentID` — MAVLink component ID
positive integer from 1 through 255

MAVLink system ID, specified as a positive integer from 1 through 255.

`Time` — Time interval
two-element vector

Time interval between which to select messages, specified as a two-element vector in seconds.

Output Arguments

`msgTable` — Table of messages
table

Table of messages with columns:

- `MessageID`
- `MessageName`
- `ComponentID`
- `SystemID`
- `Messages`

Each row of `Messages` is a `timetable` containing the message `Payload` and the associated timestamp.

Version History

Introduced in R2019a

See Also

`mavlinktlog` | `mavlinkdialect` | `mavlinkclient` | `mavlinkio`

Topics

"Visualize and Play Back MAVLink Flight Log"

deserializemsg

Deserialize MAVLink message from binary buffer

Syntax

```
msg = deserializemsg(dialect,buffer)
[msg,status] = deserializemsg(dialect,buffer,OutputAllMessage=true)
```

Description

`msg = deserializemsg(dialect,buffer)` deserializes binary buffer data specified in `buffer` based on the specified MAVLink dialect. If a message is received as multiple buffers, you can combine them by concatenating the vectors in the proper order to get a valid message.

`[msg,status] = deserializemsg(dialect,buffer,OutputAllMessage=true)` returns all messages `msg` that can be deserialized from buffer, and the status `status` of each message. `deserializemsg` returns messages and statuses even if an error occurs during the signature check or checksum check.

Input Arguments

`dialect` — MAVLink dialect
`mavlinkdialect` object

MAVLink dialect, specified as a `mavlinkdialect` object, which contains a parsed dialect XML for MAVLink message definitions.

`buffer` — Serialized message
vector of `uint8` integers

Serialized messaged, specified as vector of `uint8` integers.

Data Types: `uint8`

Output Arguments

`msg` — MAVLink message
structure

MAVLink message, returned as a structure if `OutputAllMessage` is `false`, or a structure array if `OutputAllMessage` is `true`. Each message is a structure with the fields:

- `MsgID`: Positive integer for message ID.
- `Payload`: Structure containing fields for the specific message definition.

`status` — Statuses of messages
N-element row vector of integers in the range [0, 2]

Statuses of messages, returned as an N-element row vector of integers in the range [0, 2]. N is the total number of messages in `msg`. The ith element of `status` corresponds to the parse state of the ith message in `msg`.

These are the possible statuses and the meaning for the corresponding message:

- `0` — Correctly serialized
- `1` — Checksum mismatch
- `2` — Signature mismatch

Version History

Introduced in R2019a

See Also

Functions

`createmsg` | `createcmd` | `msginfo` | `enuminfo` | `enum2num` | `num2enum`

Objects

`mavlinkdialect` | `mavlinkio` | `mavlinkclient` | `mavlinksub`

lookupPose

Obtain pose information for certain time

Syntax

```
[position,orientation,velocity,acceleration,angularVelocity] = lookupPose(
trajectory,sampleTimes)
```

Description

`[position,orientation,velocity,acceleration,angularVelocity] = lookupPose(trajectory,sampleTimes)` returns the pose information of the polynomial trajectory at the specified sample times. If any sample time is beyond the duration of the trajectory, the function returns the corresponding pose information as `NaN`.

Examples

Obtain Pose Information of Polynomial Trajectory at Certain Time

Use the `minsnappolytraj` function to generate the piecewise polynomial and the time samples for the specified waypoints of a trajectory.

```
waypoints = [0 20 20 0 0; 0 0 5 5 0; 0 5 10 5 0];
timePoints = linspace(0,30,5);
numSamples = 100;
[~,~,~,~,~,~,pp,~,~] = minsnappolytraj(waypoints,timePoints,numSamples);
```

Use the `polynomialTrajectory` System object to generate a trajectory from the piecewise polynomial. Specify the sample rate of the trajectory.

```
traj = polynomialTrajectory(pp,SampleRate=100);
```

Inspect the waypoints and times of arrival by using `waypointInfo`.

```
waypointInfo(traj)
```

```
ans=5×2 table
    TimeOfArrival                   Waypoints
    _____________    ________________________________________

          0                  0               0               0
        7.5                 20               0               5
         15                 20               5              10
       22.5                  0               5               5
         30         3.3973e-13     -2.7018e-12     -2.6041e-12
```

Obtain the time of arrival between the second and fourth waypoint. Create timestamps to sample the trajectory.

```
t0 = traj.TimeOfArrival(2);
tf = traj.TimeOfArrival(4);
sampleTimes = linspace(t0,tf,1000);
```

Obtain the position, orientation, velocity, and acceleration information at the sampled timestamps using the `lookupPose` object function.

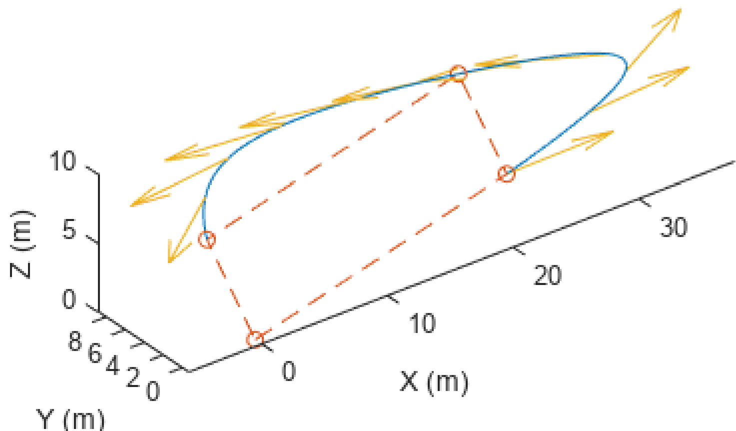
```
[pos,orient,vel,accel,~] = lookupPose(traj,sampleTimes);
```

Get the yaw angle from the orientation.

```
eulOrientation = quat2eul(orient);
yawAngle = eulOrientation(:,1);
```

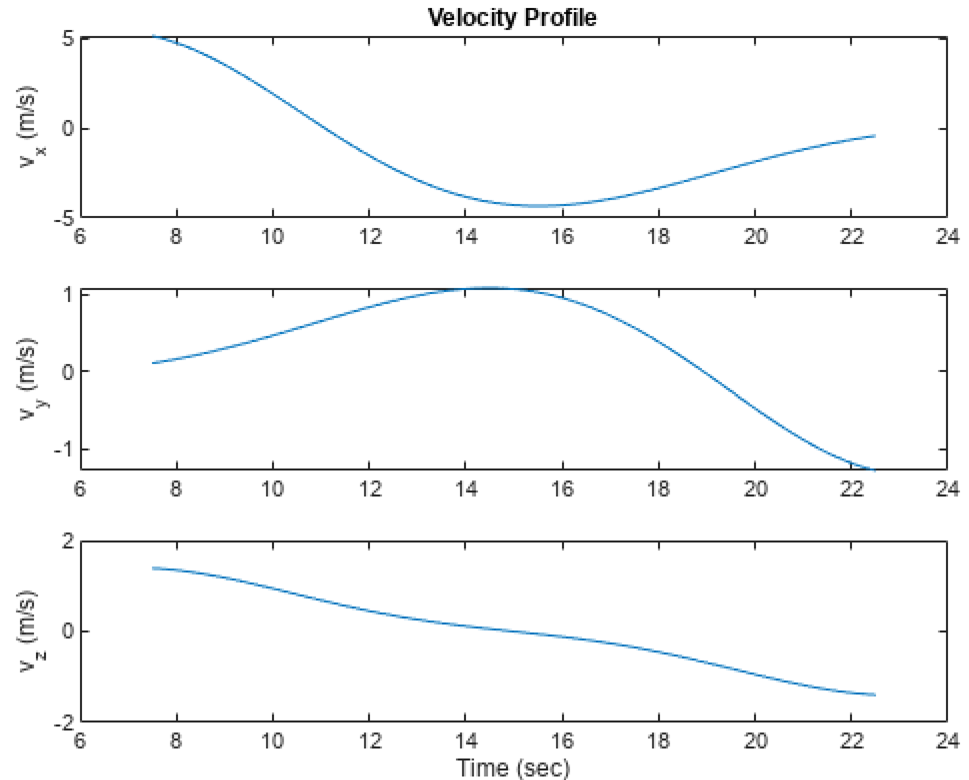
Plot the generated positions and orientations, as well as the specified waypoints.

```
plot3(pos(:,1),pos(:,2),pos(:,3), ...
      waypoints(1,:),waypoints(2,:),waypoints(3,:),"--o")
hold on
% Plot the yaw using quiver.
quiverIdx = 1:100:length(pos);
quiver3(pos(quiverIdx,1),pos(quiverIdx,2),pos(quiverIdx,3), ...
        cos(yawAngle(quiverIdx)),sin(yawAngle(quiverIdx)), ...
        zeros(numel(quiverIdx),1))
title("Position and Orientation")
xlabel("X (m)")
ylabel("Y (m)")
zlabel("Z (m)")
legend({"Position","Waypoints","Orientation"})
axis equal
hold off
```



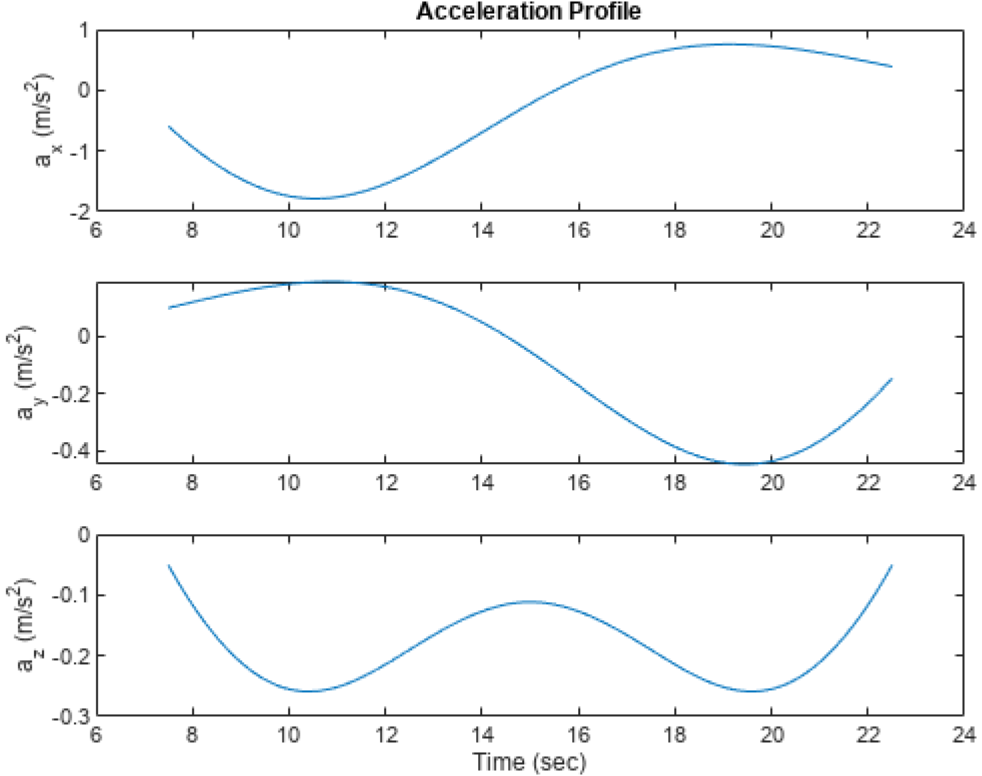
Plot the velocity profiles.

```
figure
subplot(3,1,1)
plot(sampleTimes,vel(:,1))
title("Velocity Profile")
ylabel("v_x (m/s)")
subplot(3,1,2)
plot(sampleTimes,vel(:,2))
ylabel("v_y (m/s)")
subplot(3,1,3)
plot(sampleTimes,vel(:,3))
ylabel("v_z (m/s)")
xlabel("Time (sec)")
```



Plot the acceleration profiles.

```
figure
subplot(3,1,1)
plot(sampleTimes,accel(:,1))
title("Acceleration Profile")
ylabel("a_x (m/s^2)")
subplot(3,1,2)
plot(sampleTimes,accel(:,2))
ylabel("a_y (m/s^2)")
subplot(3,1,3)
plot(sampleTimes,accel(:,3))
ylabel("a_z (m/s^2)")
xlabel("Time (sec)")
```



Input Arguments

`trajectory` — Polynomial trajectory
`polynomialTrajectory` object

Polynomial trajectory, specified as a `polynomialTrajectory` object.

`sampleTimes` — Sample times
M-element vector of nonnegative numbers

Sample times, in seconds, specified as an *M*-element vector of nonnegative numbers.

Output Arguments

`position` — Position in local navigation coordinate system (m)
M-by-3 matrix

Position in the local navigation coordinate system, in meters, returned as an *M*-by-3 matrix.

M is specified by the `sampleTimes` input.

Data Types: `double`

`orientation` — Orientation in local navigation coordinate system
M-element `quaternion` column vector | 3-by-3-by-*M* real array

Orientation in the local navigation coordinate system, returned as an *M*-element `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 at the corresponding sample time.

M is specified by the `sampleTimes` input.

Data Types: `double`

`velocity` — Velocity in local navigation coordinate system (m/s)
M-by-3 matrix

Velocity in the local navigation coordinate system, in meters per second, returned as an *M*-by-3 matrix.

M is specified by the `sampleTimes` input.

Data Types: `double`

`acceleration` — Acceleration in local navigation coordinate system (m/s^2)
M-by-3 matrix

Acceleration in the local navigation coordinate system, in meters per second squared, returned as an *M*-by-3 matrix.

M is specified by the `sampleTimes` input.

Data Types: `double`

`angularVelocity` — Angular velocity in local navigation coordinate system (rad/s)
M-by-3 matrix

Angular velocity in the local navigation coordinate system, in radians per second, returned as an *M*-by-3 matrix.

M is specified by the `sampleTimes` input.

Data Types: `double`

Version History
Introduced in R2023a

See Also

Objects
`polynomialTrajectory`

Functions
`waypointInfo`

waypointInfo

Get waypoint information table

Syntax

```
trajectoryInfo = waypointInfo(trajectory)
```

Description

`trajectoryInfo = waypointInfo(trajectory)` returns a table of waypoints, times of arrival, and orientations for the polynomial trajectory.

Examples

Generate Trajectory from Piecewise Polynomial Using polynomialTrajectory

Use the `minjerkpolytraj` function to generate the piecewise polynomial and the time samples for the specified waypoints of a trajectory.

```
waypoints = [0 20 20 0 0; 0 0 5 5 0; 0 5 10 5 0];
timePoints = cumsum([0 10 1.25*pi 10 1.25*pi]);
numSamples = 100;
[~,~,~,~,pp,~,tsamples] = minjerkpolytraj(waypoints,timePoints,numSamples);
```

Use the `polynomialTrajectory` System object to generate a trajectory from the piecewise polynomial that a multirotor must follow. Specify the sample rate of the trajectory and the orientation at each waypoint.

```
eulerAngles = [0 0 0; 0 0 0; 180 0 0; 180 0 0; 0 0 0];
q = quaternion(eulerAngles,"eulerd","ZYX","frame");
traj = polynomialTrajectory(pp,SampleRate=100,Orientation=q);
```

Inspect the waypoints, times of arrival, and orientation by using `waypointInfo`.

```
waypointInfo(traj)
```

```
ans=5×3 table
    TimeOfArrival                    Waypoints                      Orientation
    _____________    _______________________________________    ________________

           0                0              0              0     {1x1 quaternion}
          10               20              0              5     {1x1 quaternion}
      13.927               20              5             10     {1x1 quaternion}
      23.927                0              5              5     {1x1 quaternion}
      27.854        6.409e-14    -1.1102e-13    -1.1902e-13     {1x1 quaternion}
```

Obtain pose information one buffer frame at a time.

```
[pos,orient,vel,acc,angvel] = traj();
i = 1;
```

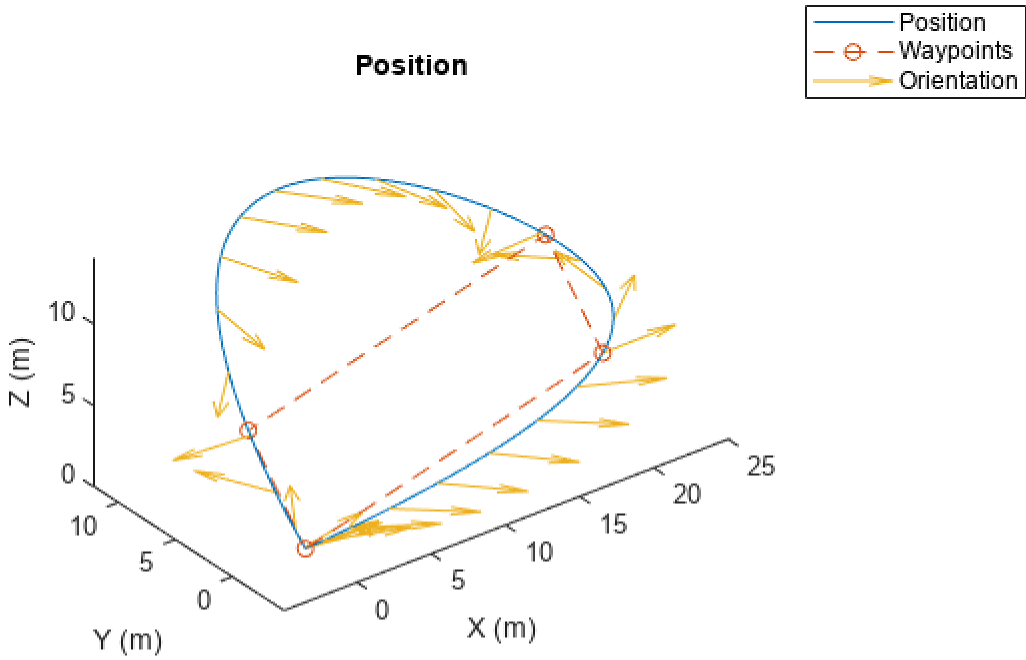
```
spf = traj.SamplesPerFrame;
while ~isDone(traj)
    idx = (i+1):(i+spf);
    [pos(idx,:),orient(idx,:), ...
     vel(idx,:),acc(idx,:),angvel(idx,:)] = traj();
    i = i + spf;
end
```

Get the yaw angle from the orientation.

```
eulOrientation = quat2eul(orient);
yawAngle = eulOrientation(:,1);
```

Plot the generated positions and orientations, as well as the specified waypoints.

```
plot3(pos(:,1),pos(:,2),pos(:,3), ...
      waypoints(1,:),waypoints(2,:),waypoints(3,:),"--o")
hold on
% Plot the yaw using quiver.
quiverIdx = 1:100:length(pos);
quiver3(pos(quiverIdx,1),pos(quiverIdx,2),pos(quiverIdx,3), ...
        cos(yawAngle(quiverIdx)),sin(yawAngle(quiverIdx)), ...
        zeros(numel(quiverIdx),1))
title("Position")
xlabel("X (m)")
ylabel("Y (m)")
zlabel("Z (m)")
legend({"Position","Waypoints","Orientation"})
axis equal
hold off
```



Obtain Pose Information of Polynomial Trajectory at Certain Time

Use the `minsnappolytraj` function to generate the piecewise polynomial and the time samples for the specified waypoints of a trajectory.

```
waypoints = [0 20 20 0 0; 0 0 5 5 0; 0 5 10 5 0];
timePoints = linspace(0,30,5);
numSamples = 100;
[~,~,~,~,~,pp,~,~] = minsnappolytraj(waypoints,timePoints,numSamples);
```

Use the `polynomialTrajectory` System object to generate a trajectory from the piecewise polynomial. Specify the sample rate of the trajectory.

```
traj = polynomialTrajectory(pp,SampleRate=100);
```

Inspect the waypoints and times of arrival by using `waypointInfo`.

```
waypointInfo(traj)
```

```
ans=5×2 table
    TimeOfArrival                    Waypoints
    _____________    ________________________________________

           0                0              0              0
         7.5               20              0              5
```

```
   15              20            5            10
 22.5               0            5             5
   30      3.3973e-13  -2.7018e-12   -2.6041e-12
```

Obtain the time of arrival between the second and fourth waypoint. Create timestamps to sample the trajectory.

```
t0 = traj.TimeOfArrival(2);
tf = traj.TimeOfArrival(4);
sampleTimes = linspace(t0,tf,1000);
```

Obtain the position, orientation, velocity, and acceleration information at the sampled timestamps using the `lookupPose` object function.

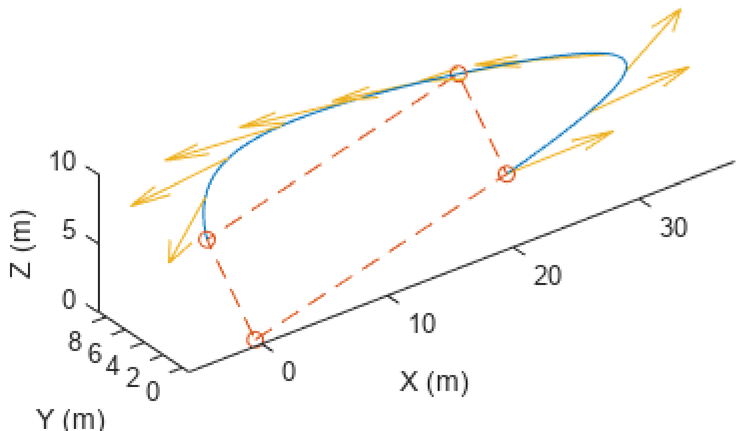
```
[pos,orient,vel,accel,~] = lookupPose(traj,sampleTimes);
```

Get the yaw angle from the orientation.

```
eulOrientation = quat2eul(orient);
yawAngle = eulOrientation(:,1);
```

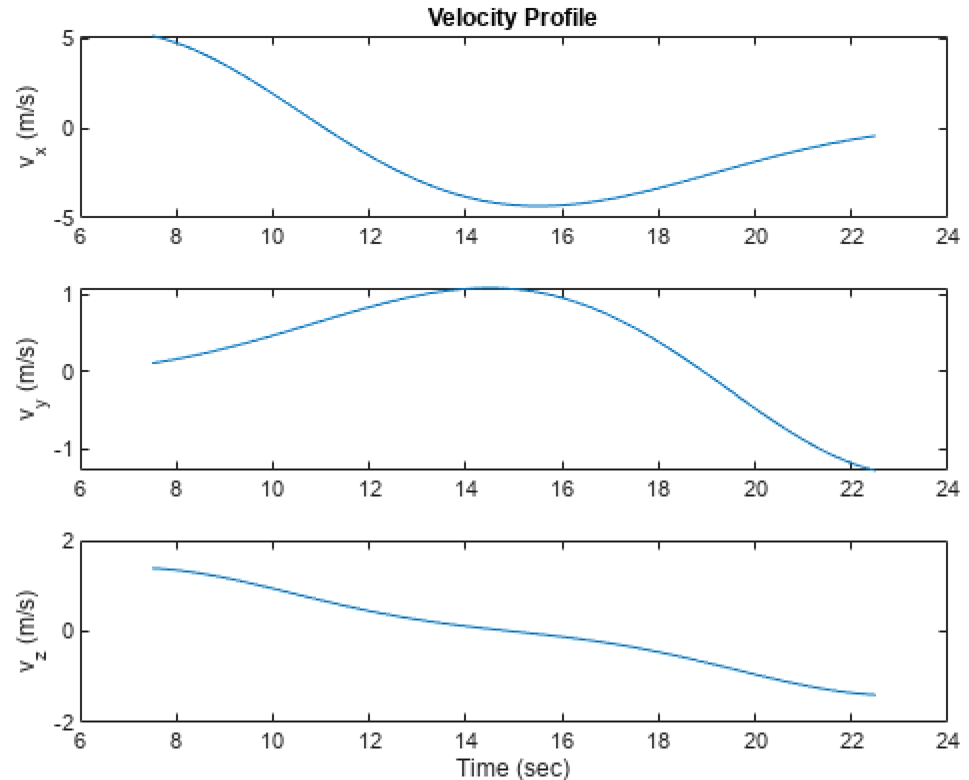
Plot the generated positions and orientations, as well as the specified waypoints.

```
plot3(pos(:,1),pos(:,2),pos(:,3), ...
      waypoints(1,:),waypoints(2,:),waypoints(3,:),"--o")
hold on
% Plot the yaw using quiver.
quiverIdx = 1:100:length(pos);
quiver3(pos(quiverIdx,1),pos(quiverIdx,2),pos(quiverIdx,3), ...
        cos(yawAngle(quiverIdx)),sin(yawAngle(quiverIdx)), ...
        zeros(numel(quiverIdx),1))
title("Position and Orientation")
xlabel("X (m)")
ylabel("Y (m)")
zlabel("Z (m)")
legend({"Position","Waypoints","Orientation"})
axis equal
hold off
```



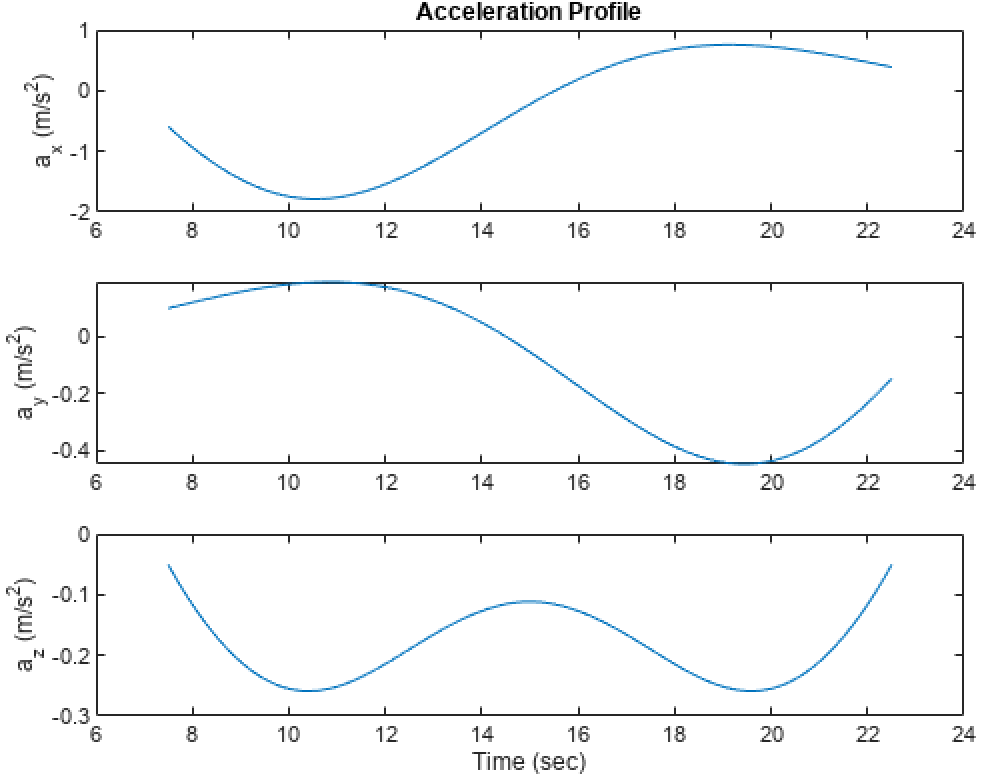
Plot the velocity profiles.

```
figure
subplot(3,1,1)
plot(sampleTimes,vel(:,1))
title("Velocity Profile")
ylabel("v_x (m/s)")
subplot(3,1,2)
plot(sampleTimes,vel(:,2))
ylabel("v_y (m/s)")
subplot(3,1,3)
plot(sampleTimes,vel(:,3))
ylabel("v_z (m/s)")
xlabel("Time (sec)")
```



Plot the acceleration profiles.

```
figure
subplot(3,1,1)
plot(sampleTimes,accel(:,1))
title("Acceleration Profile")
ylabel("a_x (m/s^2)")
subplot(3,1,2)
plot(sampleTimes,accel(:,2))
ylabel("a_y (m/s^2)")
subplot(3,1,3)
plot(sampleTimes,accel(:,3))
ylabel("a_z (m/s^2)")
xlabel("Time (sec)")
```



Input Arguments

trajectory — Polynomial trajectory
polynomialTrajectory object

Polynomial trajectory, specified as a `polynomialTrajectory` object.

Output Arguments

trajectoryInfo — Trajectory information
table

Trajectory information, returned as a table with variables corresponding to these properties of trajectory:

- Waypoints
- TimeOfArrival
- Orientation

The trajectory information table always has columns for `Waypoints` and `TimeOfArrival`. If you set the `Orientation` property when constructing,the trajectory information table additionally returns orientation.

Data Types: `table`

Version History

Introduced in R2023a

See Also

Objects
`polynomialTrajectory`

Functions
`lookupPose`

angvel

Angular velocity from quaternion array

Syntax

```
AV = angvel(Q,dt,'frame')
AV = angvel(Q,dt,'point')
[AV,qf] = angvel(Q,dt,fp,qi)
```

Description

`AV = angvel(Q,dt,'frame')` returns the angular velocity array from an array of quaternions, `Q`. The quaternions in `Q` correspond to frame rotation. The initial quaternion is assumed to represent zero rotation.

`AV = angvel(Q,dt,'point')` returns the angular velocity array from an array of quaternions, `Q`. The quaternions in `Q` correspond to point rotation. The initial quaternion is assumed to represent zero rotation.

`[AV,qf] = angvel(Q,dt,fp,qi)` allows you to specify the initial quaternion, `qi`, and the type of rotation, `fp`. It also returns the final quaternion, `qf`.

Examples

Generate Angular Velocity From Quaternion Array

Create an array of quaternions.

```
eulerAngles = [(0:10:90).',zeros(numel(0:10:90),2)];
q = quaternion(eulerAngles,'eulerd','ZYX','frame');
```

Specify the time step and generate the angular velocity array.

```
dt = 1;
av = angvel(q,dt,'frame') % units in rad/s
```

```
av = 10×3

         0         0         0
         0         0    0.1743
         0         0    0.1743
         0         0    0.1743
         0         0    0.1743
         0         0    0.1743
         0         0    0.1743
         0         0    0.1743
         0         0    0.1743
         0         0    0.1743
```

Input Arguments

`Q` — Quaternions
N-by-1 vector of quaternions

Quaternions, specified as an *N*-by-1 vector of quaternions.

Data Types: `quaternion`

`dt` — Time step
nonnegative scalar

Time step, specified as a nonnegative scalar.

Data Types: `single` | `double`

`fp` — Type of rotation
`'frame'` | `'point'`

Type of rotation, specified as `'frame'` or `'point'`.

`qi` — Initial quaternion
quaternion

Initial quaternion, specified as a quaternion.

Data Types: `quaternion`

Output Arguments

`AV` — Angular velocity
N-by-3 real matrix

Angular velocity, returned as an *N*-by-3 real matrix. *N* is the number of quaternions given in the input *Q*. Each row of the matrix corresponds to an angular velocity vector.

`qf` — Final quaternion
quaternion

Final quaternion, returned as a quaternion. `qf` is the same as the last quaternion in the *Q* input.

Data Types: `quaternion`

Version History
Introduced in R2020b

Extended Capabilities

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

See Also

`quaternion`

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`

Version History

Introduced in R2020b

Extended Capabilities

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

See Also

Functions
`compact` | `parts`

Objects
`quaternion`

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 ith row of the matrix corresponds to `quat(i)`.

Examples

Convert Quaternion Array to Compact Representation of Parts

Create a scalar quaternion with random parts. Convert the parts to a 1-by-4 vector using `compact`.

```
randomParts = randn(1,4)
```

```
randomParts = 1×4

    0.5377    1.8339   -2.2588    0.8622
```

```
quat = quaternion(randomParts)
```

```
quat = quaternion
     0.53767 +  1.8339i -  2.2588j + 0.86217k
```

```
quatParts = compact(quat)
```

```
quatParts = 1×4

    0.5377    1.8339   -2.2588    0.8622
```

Create a 2-by-2 array of quaternions, then convert the representation to a matrix of quaternion parts. The output rows correspond to the linear indices of the quaternion array.

```
quatArray = [quaternion([1:4;5:8]),quaternion([9:12;13:16])]
```

```
quatArray = 2x2 quaternion array
      1 +  2i +  3j +  4k      9 + 10i + 11j + 12k
      5 +  6i +  7j +  8k     13 + 14i + 15j + 16k
```

```
quatArrayParts = compact(quatArray)
```

```
quatArrayParts = 4×4
```

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

Input Arguments

quat — Quaternion to convert
scalar | vector | matrix | multidimensional array

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

Data Types: `quaternion`

Output Arguments

matrix — Quaternion in matrix form
N-by-4 matrix

Quaternion in matrix form, returned as an *N*-by-4 matrix, where *N* = `numel(quat)`.

Data Types: `single` | `double`

Version History

Introduced in R2020b

Extended Capabilities

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

See Also

Functions
`parts` | `classUnderlying`

Objects
`quaternion`

conj

Complex conjugate of quaternion

Syntax

```
quatConjugate = conj(quat)
```

Description

`quatConjugate = conj(quat)` returns the complex conjugate of the quaternion, `quat`.

If $q = a + b\text{i} + c\text{j} + d\text{k}$, the complex conjugate of q is $q^* = a - b\text{i} - c\text{j} - d\text{k}$. Considered as a rotation operator, the conjugate performs the opposite rotation. For example,

```
q = quaternion(deg2rad([16 45 30]),'rotvec');
a = q*conj(q);
rotatepoint(a,[0,1,0])
```

```
ans =

     0     1     0
```

Examples

Complex Conjugate of Quaternion

Create a quaternion scalar and get the complex conjugate.

```
q = normalize(quaternion([0.9 0.3 0.3 0.25]))
```

```
q = quaternion
     0.87727 + 0.29242i + 0.29242j + 0.24369k
```

```
qConj = conj(q)
```

```
qConj = quaternion
     0.87727 - 0.29242i - 0.29242j - 0.24369k
```

Verify that a quaternion multiplied by its conjugate returns a quaternion one.

```
q*qConj
```

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

Input Arguments

quat — Quaternion
scalar | vector | matrix | multidimensional array

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

Data Types: quaternion

Output Arguments

quatConjugate — Quaternion conjugate
scalar | vector | matrix | multidimensional array

Quaternion conjugate, returned as a quaternion or array of quaternions the same size as quat.

Data Types: quaternion

Version History

Introduced in R2020b

Extended Capabilities

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

See Also

Functions
norm | .*,times

Objects
quaternion

ctranspose, '

Complex conjugate transpose of quaternion array

Syntax

```
quatTransposed = quat'
```

Description

`quatTransposed = quat'` returns the complex conjugate transpose of the quaternion, `quat`.

Examples

Vector Complex Conjugate Transpose

Create a vector of quaternions and compute its complex conjugate transpose.

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

```
quat = 4x1 quaternion array
      0.53767 +  0.31877i +   3.5784j +   0.7254k
       1.8339 -   1.3077i +   2.7694j - 0.063055k
      -2.2588 -  0.43359i -   1.3499j +  0.71474k
      0.86217 +  0.34262i +   3.0349j -  0.20497k
```

```
quatTransposed = quat'
```

```
quatTransposed = 1x4 quaternion array
      0.53767 -  0.31877i -   3.5784j -   0.7254k       1.8339 +   1.3077i -   2.7694j + 0.06305
```

Matrix Complex Conjugate Transpose

Create a matrix of quaternions and compute its complex conjugate transpose.

```
quat = [quaternion(randn(2,4)),quaternion(randn(2,4))]
```

```
quat = 2x2 quaternion array
      0.53767 -   2.2588i +  0.31877j -  0.43359k       3.5784 -   1.3499i +   0.7254j +  0.7147
       1.8339 +  0.86217i -   1.3077j +  0.34262k       2.7694 +   3.0349i - 0.063055j -  0.2049
```

```
quatTransposed = quat'
```

```
quatTransposed = 2x2 quaternion array
      0.53767 +   2.2588i -  0.31877j +  0.43359k       1.8339 -  0.86217i +   1.3077j -  0.3426
       3.5784 +   1.3499i -   0.7254j -  0.71474k       2.7694 -   3.0349i + 0.063055j +  0.2049
```

Input Arguments

`quat` — Quaternion to transpose
scalar | vector | matrix

Quaternion to transpose, specified as a vector or matrix or quaternions. The complex conjugate transpose is defined for 1-D and 2-D arrays.

Data Types: `quaternion`

Output Arguments

`quatTransposed` — Conjugate transposed quaternion
scalar | vector | matrix

Conjugate transposed quaternion, returned as an *N*-by-*M* array, where `quat` was specified as an *M*-by-*N* array.

Data Types: `quaternion`

Version History

Introduced in R2020b

Extended Capabilities

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

See Also

Functions
`transpose, '`

Objects
`quaternion`

dist

Angular distance in radians

Syntax

```
distance = dist(quatA,quatB)
```

Description

`distance = dist(quatA,quatB)` returns the angular distance in radians between two quaternions, `quatA` and `quatB`.

Examples

Calculate Quaternion Distance

Calculate the quaternion distance between a single quaternion and each element of a vector of quaternions. Define the quaternions using Euler angles.

```
q = quaternion([0,0,0],'eulerd','zyx','frame')
```

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

```
qArray = quaternion([0,45,0;0,90,0;0,180,0;0,-90,0;0,-45,0],'eulerd','zyx','frame')
```

```
qArray = 5x1 quaternion array
       0.92388 +         0i +   0.38268j +         0k
       0.70711 +         0i +   0.70711j +         0k
    6.1232e-17 +         0i +         1j +         0k
       0.70711 +         0i -   0.70711j +         0k
       0.92388 +         0i -   0.38268j +         0k
```

```
quaternionDistance = rad2deg(dist(q,qArray))
```

```
quaternionDistance = 5×1

   45.0000
   90.0000
  180.0000
   90.0000
   45.0000
```

If both arguments to `dist` are vectors, the quaternion distance is calculated between corresponding elements. Calculate the quaternion distance between two quaternion vectors.

```
angles1 = [30,0,15; ...
           30,5,15; ...
```

```
            30,10,15; ...
            30,15,15];
angles2 = [30,6,15; ...
            31,11,15; ...
            30,16,14; ...
            30.5,21,15.5];

qVector1 = quaternion(angles1,'eulerd','zyx','frame');
qVector2 = quaternion(angles2,'eulerd','zyx','frame');

rad2deg(dist(qVector1,qVector2))
```

```
ans = 4×1

    6.0000
    6.0827
    6.0827
    6.0287
```

Note that a quaternion represents the same rotation as its negative. Calculate a quaternion and its negative.

```
qPositive = quaternion([30,45,-60],'eulerd','zyx','frame')
```

```
qPositive = quaternion
     0.72332 - 0.53198i + 0.20056j +  0.3919k
```

```
qNegative = -qPositive
```

```
qNegative = quaternion
    -0.72332 + 0.53198i - 0.20056j -  0.3919k
```

Find the distance between the quaternion and its negative.

```
dist(qPositive,qNegative)
```

```
ans = 0
```

The components of a quaternion may look different from the components of its negative, but both expressions represent the same rotation.

Input Arguments

quatA,quatB — Quaternions to calculate distance between

scalar | vector | matrix | multidimensional array

Quaternions to calculate distance between, specified as comma-separated quaternions or arrays of quaternions. `quatA` and `quatB` must have compatible sizes:

- `size(quatA) == size(quatB)`, or
- `numel(quatA) == 1`, or
- `numel(quatB) == 1`, or

- if `[Adim1,…,AdimN] = size(quatA)` and `[Bdim1,…,BdimN] = size(quatB)`, then for `i = 1:N`, either `Adimi==Bdimi` or `Adim==1` or `Bdim==1`.

  If one of the quaternion arguments contains only one quaternion, then this function returns the distances between that quaternion and every quaternion in the other argument.

Data Types: `quaternion`

Output Arguments

`distance` — Angular distance (radians)
scalar | vector | matrix | multidimensional array

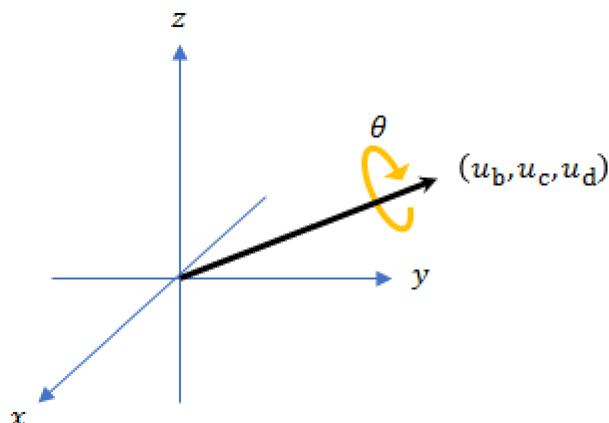
Angular distance in radians, returned as an array. The dimensions are the maximum of the union of `size(quatA)` and `size(quatB)`.

Data Types: `single` | `double`

Algorithms

The `dist` function returns the angular distance between two quaternions.

A quaternion may be defined by an axis (u_b,u_c,u_d) and angle of rotation θ_q:
$q = \cos\left(\theta_q/2\right) + \sin\left(\theta_q/2\right)(u_b\text{i} + u_c\text{j} + u_d\text{k})$.



Given a quaternion in the form, $q = a + b\text{i} + c\text{j} + d\text{k}$, where a is the real part, you can solve for the angle of q as $\theta_q = 2\cos^{-1}(a)$.

Consider two quaternions, p and q, and the product $z = p*\text{conjugate}(q)$. As p approaches q, the angle of z goes to 0, and z approaches the unit quaternion.

The angular distance between two quaternions can be expressed as $\theta_z = 2\cos^{-1}(\text{real}(z))$.

Using the `quaternion` data type syntax, the angular distance is calculated as:

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

Version History

Introduced in R2020b

Extended Capabilities

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

See Also

Functions
`parts` | `conj`

Objects
`quaternion`

euler

Convert quaternion to Euler angles (radians)

Syntax

```
eulerAngles = euler(quat,rotationSequence,rotationType)
```

Description

`eulerAngles = euler(quat,rotationSequence,rotationType)` converts the quaternion, `quat`, to an *N*-by-3 matrix of Euler angles.

Examples

Convert Quaternion to Euler Angles in Radians

Convert a quaternion frame rotation to Euler angles in radians using the 'ZYX' rotation sequence.

```
quat = quaternion([0.7071 0.7071 0 0]);
eulerAnglesRandians = euler(quat,'ZYX','frame')
```

```
eulerAnglesRandians = 1×3

         0         0    1.5708
```

Input Arguments

`quat` — Quaternion to convert to Euler angles
scalar | vector | matrix | multidimensional array

Quaternion to convert to Euler angles, specified as a scalar, vector, matrix, or multidimensional array of quaternions.

Data Types: `quaternion`

`rotationSequence` — Rotation sequence
`'ZYX'` | `'ZYZ'` | `'ZXY'` | `'ZXZ'` | `'YXZ'` | `'YXY'` | `'YZX'` | `'YZY'` | `'XYZ'` | `'XYX'` | `'XZY'` | `'XZX'`

Rotation sequence of Euler representation, specified as a character vector or string.

The rotation sequence defines the order of rotations about the axes. For example, if you specify a rotation sequence of `'YZX'`:

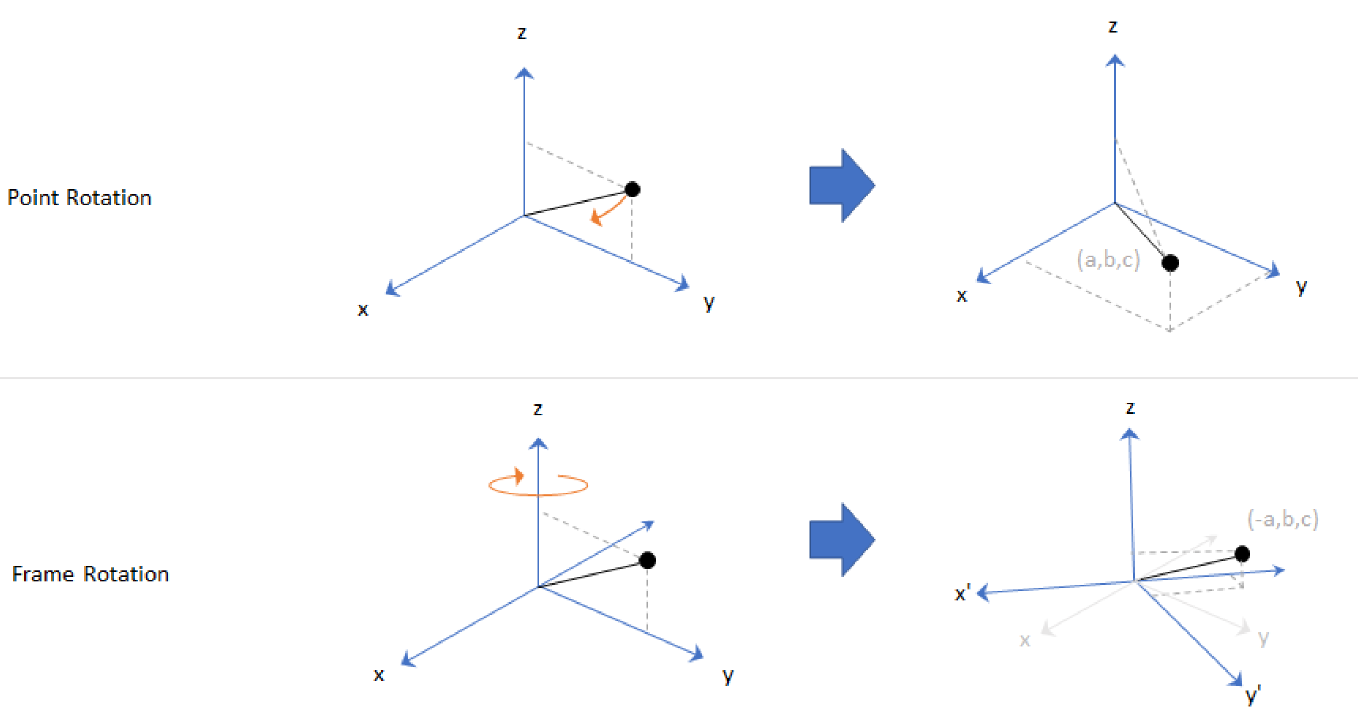
1. The first rotation is about the y-axis.
2. The second rotation is about the new z-axis.
3. The third rotation is about the new x-axis.

Data Types: `char` | `string`

`rotationType` — Type of rotation
`'point'` | `'frame'`

Type of rotation, specified as `'point'` or `'frame'`.

In a point rotation, the frame is static and the point moves. In a frame rotation, the point is static and the frame moves. Point rotation and frame rotation define equivalent angular displacements but in opposite directions.



Data Types: `char` | `string`

Output Arguments

`eulerAngles` — Euler angle representation (radians)
N-by-3 matrix

Euler angle representation in radians, returned as a *N*-by-3 matrix. *N* is the number of quaternions in the `quat` argument.

For each row of `eulerAngles`, the first element corresponds to the first axis in the rotation sequence, the second element corresponds to the second axis in the rotation sequence, and the third element corresponds to the third axis in the rotation sequence.

The data type of the Euler angles representation is the same as the underlying data type of `quat`.

Data Types: `single` | `double`

Version History

Introduced in R2020b

Extended Capabilities

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

See Also

Functions
`eulerd` | `rotateframe` | `rotatepoint`

Objects
`quaternion`

eulerd

Convert quaternion to Euler angles (degrees)

Syntax

```
eulerAngles = eulerd(quat,rotationSequence,rotationType)
```

Description

`eulerAngles = eulerd(quat,rotationSequence,rotationType)` converts the quaternion, `quat`, to an *N*-by-3 matrix of Euler angles in degrees.

Examples

Convert Quaternion to Euler Angles in Degrees

Convert a quaternion frame rotation to Euler angles in degrees using the 'ZYX' rotation sequence.

```
quat = quaternion([0.7071 0.7071 0 0]);
eulerAnglesDegrees = eulerd(quat,'ZYX','frame')
```

```
eulerAnglesDegrees = 1×3

         0         0   90.0000
```

Input Arguments

`quat` — Quaternion to convert to Euler angles
scalar | vector | matrix | multidimensional array

Quaternion to convert to Euler angles, specified as a scalar, vector, matrix, or multidimensional array of quaternions.

Data Types: `quaternion`

`rotationSequence` — Rotation sequence
`'ZYX'` | `'ZYZ'` | `'ZXY'` | `'ZXZ'` | `'YXZ'` | `'YXY'` | `'YZX'` | `'YZY'` | `'XYZ'` | `'XYX'` | `'XZY'` | `'XZX'`

Rotation sequence of Euler angle representation, specified as a character vector or string.

The rotation sequence defines the order of rotations about the axes. For example, if you specify a rotation sequence of `'YZX'`:

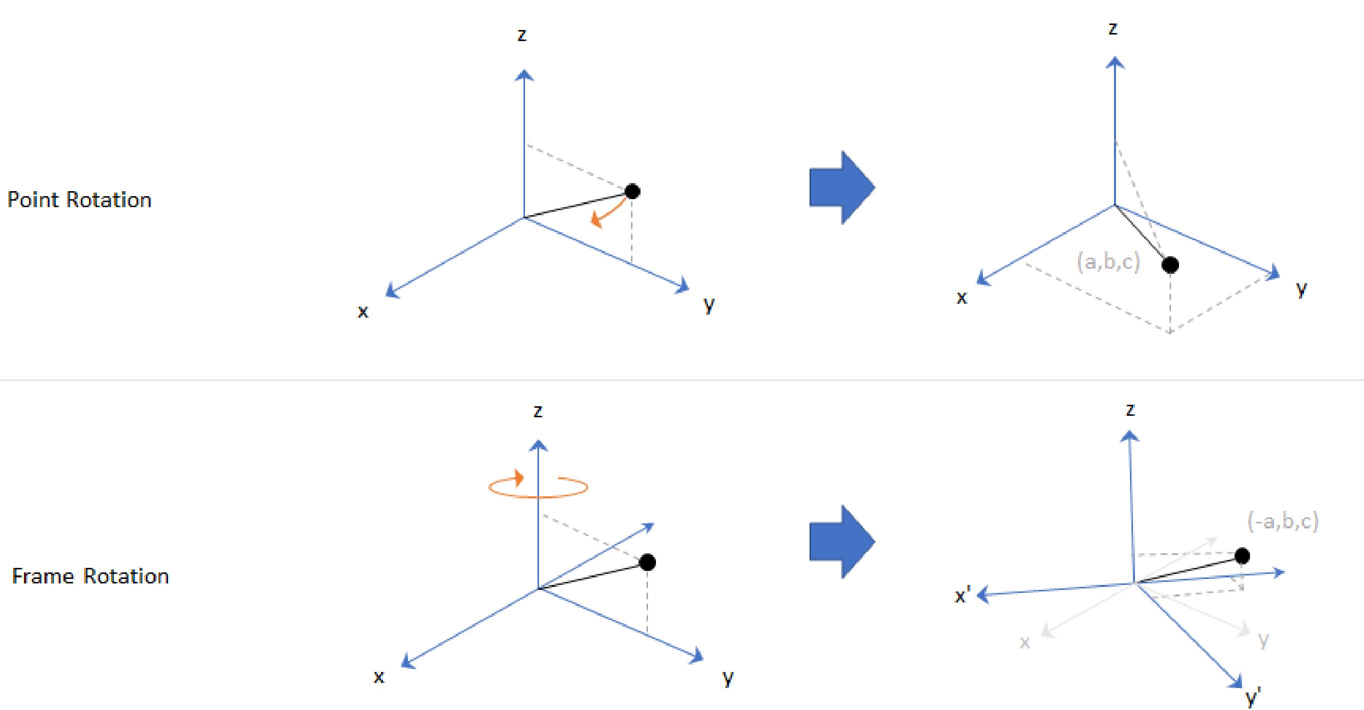
1. The first rotation is about the *y*-axis.
2. The second rotation is about the new *z*-axis.
3. The third rotation is about the new *x*-axis.

Data Types: `char` | `string`

rotationType — Type of rotation
'point' | 'frame'

Type of rotation, specified as 'point' or 'frame'.

In a point rotation, the frame is static and the point moves. In a frame rotation, the point is static and the frame moves. Point rotation and frame rotation define equivalent angular displacements but in opposite directions.



Data Types: char | string

Output Arguments

eulerAngles — Euler angle representation (degrees)
N-by-3 matrix

Euler angle representation in degrees, returned as a *N*-by-3 matrix. *N* is the number of quaternions in the quat argument.

For each row of eulerAngles, the first column corresponds to the first axis in the rotation sequence, the second column corresponds to the second axis in the rotation sequence, and the third column corresponds to the third axis in the rotation sequence.

The data type of the Euler angles representation is the same as the underlying data type of quat.

Data Types: single | double

Version History

Introduced in R2020b

Extended Capabilities

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

See Also

Functions
`euler` | `rotateframe` | `rotatepoint`

Objects
`quaternion`

exp

Exponential of quaternion array

Syntax

```
B = exp(A)
```

Description

`B = exp(A)` computes the exponential of the elements of the quaternion array `A`.

Examples

Exponential of Quaternion Array

Create a 4-by-1 quaternion array A.

```
A = quaternion(magic(4))
```

```
A = 4x1 quaternion array
     16 +  2i +  3j + 13k
      5 + 11i + 10j +  8k
      9 +  7i +  6j + 12k
      4 + 14i + 15j +  1k
```

Compute the exponential of A.

```
B = exp(A)
```

```
B = 4x1 quaternion array
     5.3525e+06 + 1.0516e+06i + 1.5774e+06j + 6.8352e+06k
        -57.359 -     89.189i -     81.081j -     64.865k
        -6799.1 +     2039.1i +     1747.8j +     3495.6k
          -6.66 +     36.931i +     39.569j +     2.6379k
```

Input Arguments

A — Input quaternion
scalar | vector | matrix | multidimensional array

Input quaternion, specified as a scalar, vector, matrix, or multidimensional array.

Data Types: `quaternion`

Output Arguments

B — Result
scalar | vector | matrix | multidimensional array

Result of quaternion exponential, returned as a scalar, vector, matrix, or multidimensional array.

Data Types: `quaternion`

Algorithms

Given a quaternion $A = a + b\mathrm{i} + c\mathrm{j} + d\mathrm{k} = a + \bar{v}$, the exponential is computed by

$$\exp(A) = e^{a}\left(\cos\|\bar{v}\| + \frac{\bar{v}}{\|\bar{v}\|}\sin\|\bar{v}\|\right)$$

Version History

Introduced in R2020b

Extended Capabilities

C/C++ Code Generation

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

See Also

Functions

`.^,power` | `log`

Objects

`quaternion`

ldivide, .\

Element-wise quaternion left division

Syntax

```
C = A.\B
```

Description

`C = A.\B` performs quaternion element-wise division by dividing each element of quaternion `B` by the corresponding element of quaternion `A`.

Examples

Divide a Quaternion Array by a Real Scalar

Create a 2-by-1 quaternion array, and divide it element-by-element by a real scalar.

```
A = quaternion([1:4;5:8])
```

```
A = 2x1 quaternion array
     1 + 2i + 3j + 4k
     5 + 6i + 7j + 8k
```

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

```
C = 2x1 quaternion array
     0.066667 -  0.13333i -      0.2j -  0.26667k
     0.057471 - 0.068966i -  0.08046j - 0.091954k
```

Divide a Quaternion Array by Another Quaternion Array

Create a 2-by-2 quaternion array, and divide it element-by-element by another 2-by-2 quaternion array.

```
q1 = quaternion([1:4;2:5;4:7;5:8]);
A = reshape(q1,2,2)
```

```
A = 2x2 quaternion array
     1 + 2i + 3j + 4k     4 + 5i + 6j + 7k
     2 + 3i + 4j + 5k     5 + 6i + 7j + 8k
```

```
q2 = quaternion(magic(4));
B = reshape(q2,2,2)
```

```
B = 2x2 quaternion array
     16 +  2i +  3j + 13k      9 +  7i +  6j + 12k
      5 + 11i + 10j +  8k      4 + 14i + 15j +  1k
```

```
C = A.\B
```

```
C = 2x2 quaternion array
         2.7 -      1.9i -      0.9j -      1.7k       1.5159 -  0.37302i -  0.15079j -  0.0238
      2.2778 +  0.46296i -  0.57407j + 0.092593k       1.2471 +  0.91379i -  0.33908j -   0.109
```

Input Arguments

A — Divisor
scalar | vector | matrix | multidimensional array

Divisor, specified as a quaternion, an array of quaternions, a real scalar, or an array of real numbers.

A and B must have compatible sizes. In the simplest cases, they can be the same size or one can be a scalar. Two inputs have compatible sizes if, for every dimension, the dimension sizes of the inputs are the same or one of the dimensions is 1.

Data Types: quaternion | single | double

B — Dividend
scalar | vector | matrix | multidimensional array

Dividend, specified as a quaternion, an array of quaternions, a real scalar, or an array of real numbers.

A and B must have compatible sizes. In the simplest cases, they can be the same size or one can be a scalar. Two inputs have compatible sizes if, for every dimension, the dimension sizes of the inputs are the same or one of the dimensions is 1.

Data Types: quaternion | single | double

Output Arguments

C — Result
scalar | vector | matrix | multidimensional array

Result of quaternion division, returned as a scalar, vector, matrix, or multidimensional array.

Data Types: quaternion

Algorithms

Quaternion Division

Given a quaternion $A = a_1 + a_2\mathrm{i} + a_3\mathrm{j} + a_4\mathrm{k}$ and a real scalar p,

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

Note For a real scalar p, $A./p = A.\backslash p$.

Quaternion Division by a Quaternion Scalar

Given two quaternions A and B of compatible sizes, then

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

Version History

Introduced in R2020b

Extended Capabilities

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

See Also

Functions
`.*,times` | `conj` | `norm` | `./,ldivide`

Objects
`quaternion`

log

Natural logarithm of quaternion array

Syntax

```
B = log(A)
```

Description

`B = log(A)` computes the natural logarithm of the elements of the quaternion array `A`.

Examples

Logarithmic Values of Quaternion Array

Create a 3-by-1 quaternion array A.

```
A = quaternion(randn(3,4))
```

```
A = 3x1 quaternion array
     0.53767 + 0.86217i - 0.43359j +  2.7694k
      1.8339 + 0.31877i + 0.34262j -  1.3499k
     -2.2588 -  1.3077i +  3.5784j +  3.0349k
```

Compute the logarithmic values of A.

```
B = log(A)
```

```
B = 3x1 quaternion array
      1.0925 + 0.40848i - 0.20543j +  1.3121k
      0.8436 + 0.14767i + 0.15872j - 0.62533k
      1.6807 - 0.53829i +   1.473j +  1.2493k
```

Input Arguments

A — Input array
scalar | vector | matrix | multidimensional array

Input array, specified as a scalar, vector, matrix, or multidimensional array.

Data Types: `quaternion`

Output Arguments

B — Logarithm values
scalar | vector | matrix | multidimensional array

Quaternion natural logarithm values, returned as a scalar, vector, matrix, or multidimensional array.

Data Types: `quaternion`

Algorithms

Given a quaternion $A = a + \bar{v} = a + b\mathrm{i} + c\mathrm{j} + d\mathrm{k}$, the logarithm is computed by

$$\log(A) = \log\|A\| + \frac{\bar{v}}{\|\bar{v}\|}\arccos\frac{a}{\|A\|}$$

Version History

Introduced in R2020b

Extended Capabilities

C/C++ Code Generation

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

See Also

Functions

`exp` | `.^,power`

Objects

`quaternion`

meanrot

Quaternion mean rotation

Syntax

```
quatAverage = meanrot(quat)
quatAverage = meanrot(quat,dim)
quatAverage = meanrot( ___ ,nanflag)
```

Description

`quatAverage = meanrot(quat)` returns the average rotation of the elements of `quat` along the first array dimension whose size not does equal 1.

- If `quat` is a vector, `meanrot(quat)` returns the average rotation of the elements.
- If `quat` is a matrix, `meanrot(quat)` returns a row vector containing the average rotation of each column.
- If `quat` is a multidimensional array, then `mearot(quat)` operates along the first array dimension whose size does not equal 1, treating the elements as vectors. This dimension becomes 1 while the sizes of all other dimensions remain the same.

The `meanrot` function normalizes the input quaternions, `quat`, before calculating the mean.

`quatAverage = meanrot(quat,dim)` return the average rotation along dimension `dim`. For example, if `quat` is a matrix, then `meanrot(quat,2)` is a column vector containing the mean of each row.

`quatAverage = meanrot( ___ ,nanflag)` specifies whether to include or omit `NaN` values from the calculation for any of the previous syntaxes. `meanrot(quat,'includenan')` includes all `NaN` values in the calculation while `mean(quat,'omitnan')` ignores them.

Examples

Quaternion Mean Rotation

Create a matrix of quaternions corresponding to three sets of Euler angles.

```
eulerAngles = [40 20 10; ...
               50 10 5; ...
               45 70 1];

quat = quaternion(eulerAngles,'eulerd','ZYX','frame');
```

Determine the average rotation represented by the quaternions. Convert the average rotation to Euler angles in degrees for readability.

```
quatAverage = meanrot(quat)
```

```
quatAverage = quaternion
      0.88863 - 0.062598i +  0.27822j +  0.35918k
```

```
eulerAverage = eulerd(quatAverage,'ZYX','frame')
```

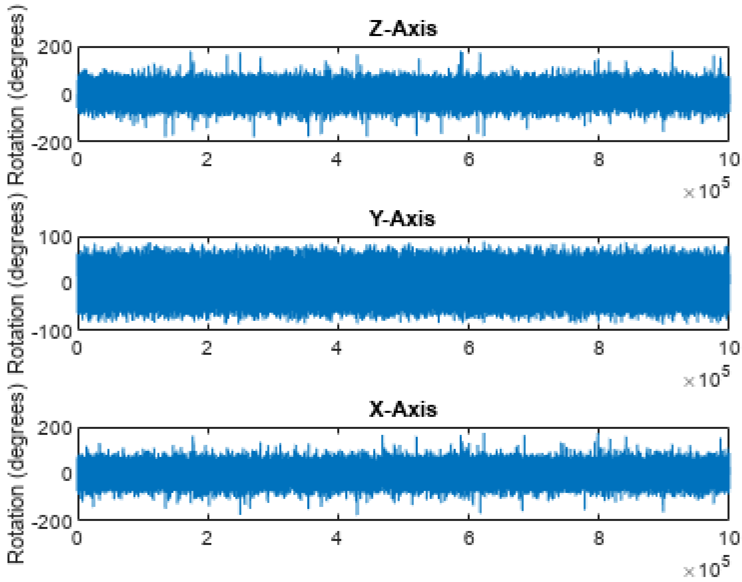
```
eulerAverage = 1×3

   45.7876   32.6452    6.0407
```

Average Out Rotational Noise

Use `meanrot` over a sequence of quaternions to average out additive noise.

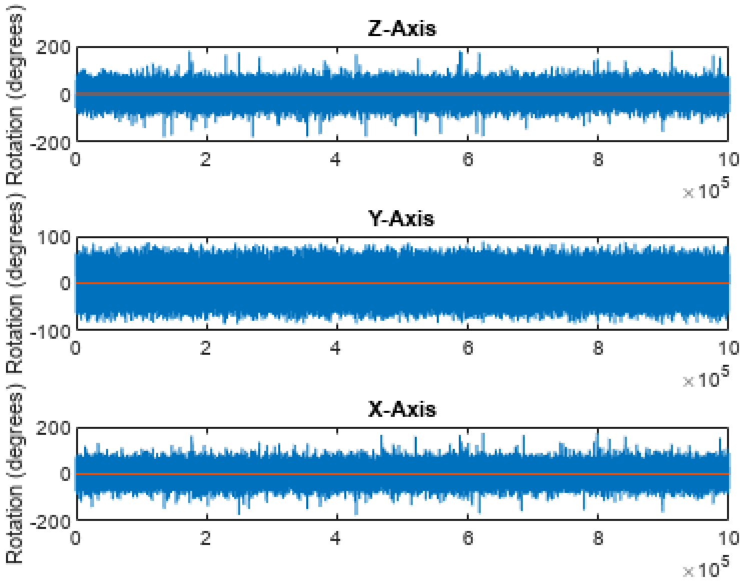
Create a vector of 1e6 quaternions whose distance, as defined by the `dist` function, from quaternion(1,0,0,0) is normally distributed. Plot the Euler angles corresponding to the noisy quaternion vector.

```
nrows = 1e6;
ax = 2*rand(nrows,3) - 1;
ax = ax./sqrt(sum(ax.^2,2));
ang = 0.5*randn(size(ax,1),1);
q = quaternion(ax.*ang ,'rotvec');

noisyEulerAngles = eulerd(q,'ZYX','frame');

figure(1)

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

subplot(3,1,2)
plot(noisyEulerAngles(:,2))
title('Y-Axis')
ylabel('Rotation (degrees)')
hold on

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



Use `meanrot` to determine the average quaternion given the vector of quaternions. Convert to Euler angles and plot the results.

```
qAverage = meanrot(q);

qAverageInEulerAngles = eulerd(qAverage,'ZYX','frame');

figure(1)

subplot(3,1,1)
plot(ones(nrows,1)*qAverageInEulerAngles(:,1))
title('Z-Axis')

subplot(3,1,2)
plot(ones(nrows,1)*qAverageInEulerAngles(:,2))
title('Y-Axis')

subplot(3,1,3)
plot(ones(nrows,1)*qAverageInEulerAngles(:,3))
title('X-Axis')
```



The meanrot Algorithm and Limitations

The meanrot Algorithm

The meanrot function outputs a quaternion that minimizes the squared Frobenius norm of the difference between rotation matrices. Consider two quaternions:

- q0 represents no rotation.
- q90 represents a 90 degree rotation about the *x*-axis.

```
q0 = quaternion([0 0 0],'eulerd','ZYX','frame');
q90 = quaternion([0 0 90],'eulerd','ZYX','frame');
```

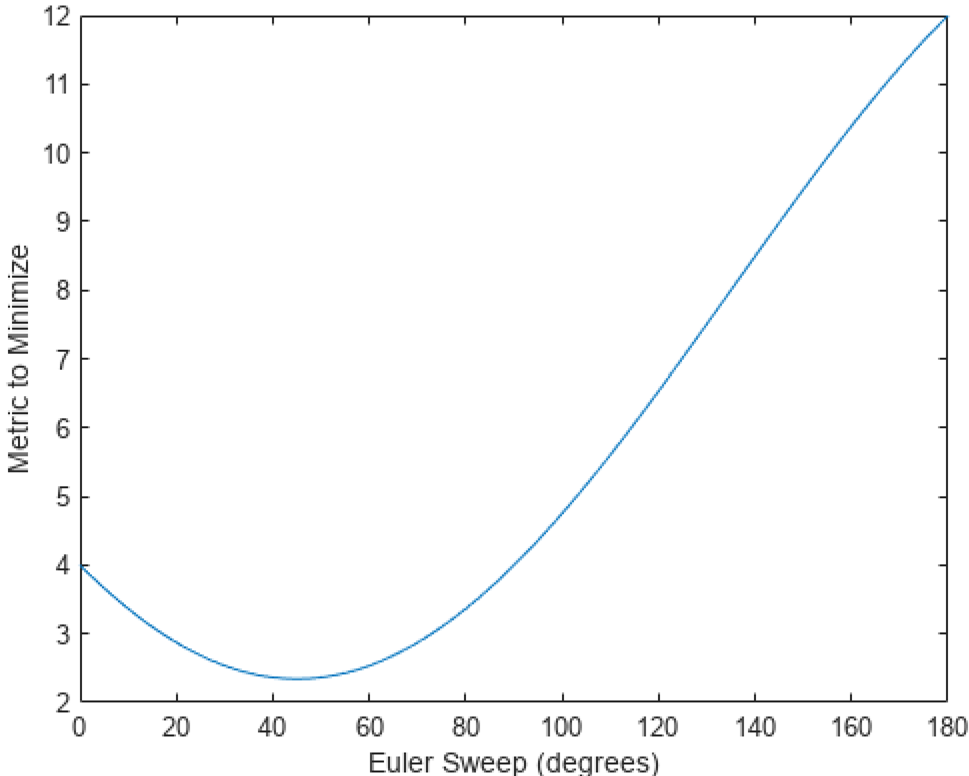
Create a quaternion sweep, qSweep, that represents rotations from 0 to 180 degrees about the *x*-axis.

```
eulerSweep = (0:1:180)';
qSweep = quaternion([zeros(numel(eulerSweep),2),eulerSweep], ...
    'eulerd','ZYX','frame');
```

Convert q0, q90, and qSweep to rotation matrices. In a loop, calculate the metric to minimize for each member of the quaternion sweep. Plot the results and return the value of the Euler sweep that corresponds to the minimum of the metric.

```
r0     = rotmat(q0,'frame');
r90    = rotmat(q90,'frame');
rSweep = rotmat(qSweep,'frame');

metricToMinimize = zeros(size(rSweep,3),1);
for i = 1:numel(qSweep)
    metricToMinimize(i) = norm((rSweep(:,:,i) - r0),'fro').^2 + ...
                          norm((rSweep(:,:,i) - r90),'fro').^2;
end
```

```
plot(eulerSweep,metricToMinimize)
xlabel('Euler Sweep (degrees)')
ylabel('Metric to Minimize')
```




```
[~,eulerIndex] = min(metricToMinimize);
eulerSweep(eulerIndex)
```

```
ans = 45
```

The minimum of the metric corresponds to the Euler angle sweep at 45 degrees. That is, `meanrot` defines the average between `quaterion([0 0 0],'ZYX','frame')` and `quaternion([0 0 90],'ZYX','frame')` as `quaternion([0 0 45],'ZYX','frame')`. Call `meanrot` with `q0` and `q90` to verify the same result.

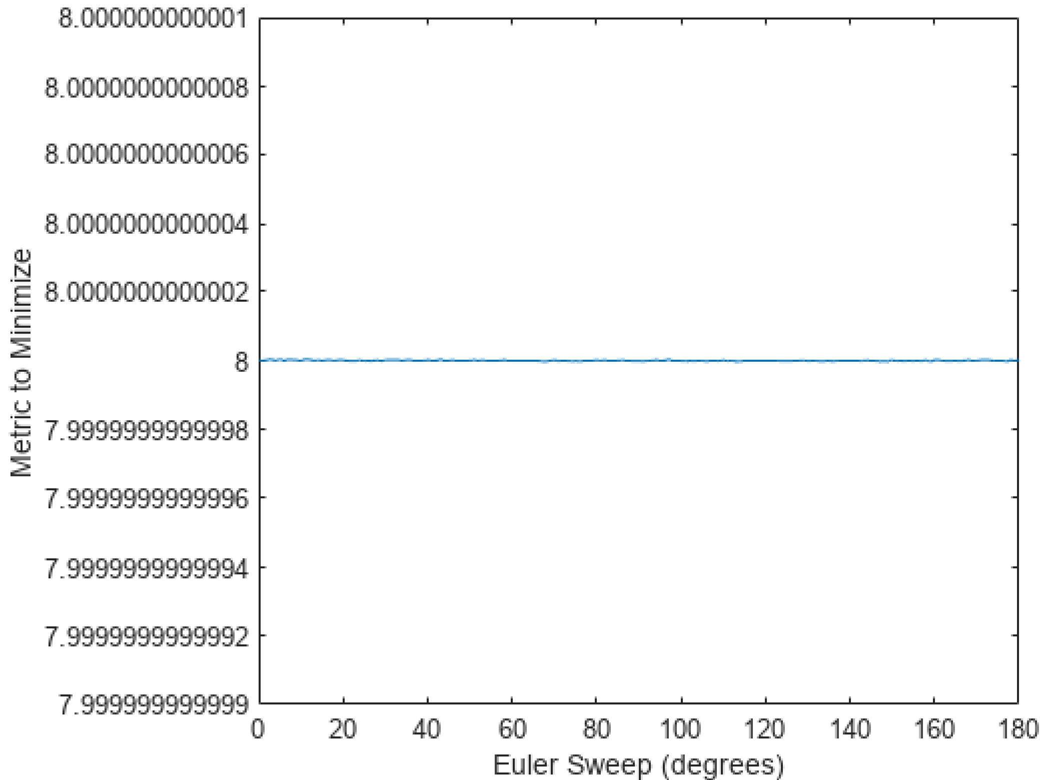
```
eulerd(meanrot([q0,q90]),'ZYX','frame')
```

```
ans = 1×3

         0         0   45.0000
```

Limitations

The metric that `meanrot` uses to determine the mean rotation is not unique for quaternions significantly far apart. Repeat the experiment above for quaternions that are separated by 180 degrees.

```
q180 = quaternion([0 0 180],'eulerd','ZYX','frame');
r180 = rotmat(q180,'frame');

for i = 1:numel(qSweep)
    metricToMinimize(i) = norm((rSweep(:,:,i) - r0),'fro').^2 + ...
                          norm((rSweep(:,:,i) - r180),'fro').^2;
end

plot(eulerSweep,metricToMinimize)
xlabel('Euler Sweep (degrees)')
ylabel('Metric to Minimize')
```




```
[~,eulerIndex] = min(metricToMinimize);
eulerSweep(eulerIndex)
```

```
ans = 159
```

Quaternion means are usually calculated for rotations that are close to each other, which makes the edge case shown in this example unlikely in real-world applications. To average two quaternions that are significantly far apart, use the `slerp` function. Repeat the experiment using `slerp` and verify that the quaternion mean returned is more intuitive for large distances.

```
qMean = slerp(q0,q180,0.5);
q0_q180 = eulerd(qMean,'ZYX','frame')
```

```
q0_q180 = 1×3
```

```
         0         0   90.0000
```

Input Arguments

`quat` — Quaternion
scalar | vector | matrix | multidimensional array

Quaternion for which to calculate the mean, specified as a scalar, vector, matrix, or multidimensional array of quaternions.

Data Types: `quaternion`

`dim` — Dimension to operate along
positive integer scalar

Dimension to operate along, specified as a positive integer scalar. If no value is specified, then the default is the first array dimension whose size does not equal 1.

Dimension `dim` indicates the dimension whose length reduces to 1. The `size(quatAverage,dim)` is `1`, while the sizes of all other dimensions remain the same.

Data Types: `double` | `single`

`nanflag` — NaN condition
`'includenan'` (default) | `'omitnan'`

`NaN` condition, specified as one of these values:

- `'includenan'` –– Include `NaN` values when computing the mean rotation, resulting in `NaN`.
- `'omitnan'` –– Ignore all `NaN` values in the input.

Data Types: `char` | `string`

Output Arguments

`quatAverage` — Quaternion average rotation
scalar | vector | matrix | multidimensional array

Quaternion average rotation, returned as a scalar, vector, matrix, or multidimensional array.

Data Types: `single` | `double`

Algorithms

`meanrot` determines a quaternion mean, $\bar{q}$, according to [1]. $\bar{q}$ is the quaternion that minimizes the squared Frobenius norm of the difference between rotation matrices:

$$\bar{q} = \arg \min_{q \in S^3} \sum_{i=1}^{n} \|A(q) - A(q_i)\|_F^2$$

Version History

Introduced in R2020b

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`

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`

Version History

Introduced in R2020b

Extended Capabilities

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

See Also

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

Objects
`quaternion`

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

$$q = a_{\mathrm{q}} + b_{\mathrm{q}}\mathrm{i} + c_{\mathrm{q}}\mathrm{j} + d_{\mathrm{q}}\mathrm{k},$$

the product of q and a real scalar β is

$$\beta q = \beta a_{\mathrm{q}} + \beta b_{\mathrm{q}}\mathrm{i} + \beta c_{\mathrm{q}}\mathrm{j} + \beta d_{\mathrm{q}}\mathrm{k}$$

Quaternion Multiplication by a Quaternion Scalar

The definition of the basis elements for quaternions,

$$\mathrm{i}^2 = \mathrm{j}^2 = \mathrm{k}^2 = \mathrm{ijk} = -1\,,$$

can be expanded to populate a table summarizing quaternion basis element multiplication:

	1	**i**	**j**	**k**
1	1	i	j	k
i	i	−1	k	−j

j	j	−k	−1	i
k	k	j	−i	−1

When reading the table, the rows are read first, for example: ij = k and ji = −k.

Given two quaternions, $q = a_q + b_q\mathrm{i} + c_q\mathrm{j} + d_q\mathrm{k}$, and $p = a_p + b_p\mathrm{i} + c_p\mathrm{j} + d_p\mathrm{k}$, the multiplication can be expanded as:

$$\begin{aligned} z = pq &= (a_p + b_p\mathrm{i} + c_p\mathrm{j} + d_p\mathrm{k})(a_q + b_q\mathrm{i} + c_q\mathrm{j} + d_q\mathrm{k}) \\ &= a_pa_q + a_pb_q\mathrm{i} + a_pc_q\mathrm{j} + a_pd_q\mathrm{k} \\ &+ b_pa_q\mathrm{i} + b_pb_q\mathrm{i}^2 + b_pc_q\mathrm{ij} + b_pd_q\mathrm{ik} \\ &+ c_pa_q\mathrm{j} + c_pb_q\mathrm{ji} + c_pc_q\mathrm{j}^2 + c_pd_q\mathrm{jk} \\ &+ d_pa_qk + d_pb_q\mathrm{ki} + d_pc_q\mathrm{kj} + d_pd_q\mathrm{k}^2 \end{aligned}$$

You can simplify the equation using the quaternion multiplication table:

$$\begin{aligned} z = pq \; &= a_pa_q + a_pb_q\mathrm{i} + a_pc_q\mathrm{j} + a_pd_q\mathrm{k} \\ &+ b_pa_q\mathrm{i} - b_pb_q + b_pc_q\mathrm{k} - b_pd_q\mathrm{j} \\ &+ c_pa_q\mathrm{j} - c_pb_q\mathrm{k} - c_pc_q + c_pd_q\mathrm{i} \\ &+ d_pa_qk + d_pb_q\mathrm{j} - d_pc_q\mathrm{i} - d_pd_q \end{aligned}$$

Version History

Introduced in R2020b

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`

norm

Quaternion norm

Syntax

```
N = norm(quat)
```

Description

`N = norm(quat)` returns the norm of the quaternion, `quat`.

Given a quaternion of the form $Q = a + b\text{i} + c\text{j} + d\text{k}$, 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`

Version History
Introduced in R2020b

Extended Capabilities

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

See Also

Functions
`normalize` | `parts` | `conj`

Objects
`quaternion`

normalize

Quaternion normalization

Syntax

`quatNormalized = normalize(quat)`

Description

`quatNormalized = normalize(quat)` normalizes the quaternion.

Given a quaternion of the form $Q = a + b\text{i} + c\text{j} + d\text{k}$, the normalized quaternion is defined as $Q/\sqrt{a^2 + b^2 + c^2 + d^2}$.

Examples

Normalize Elements of Quaternion Vector

Quaternions can represent rotations when normalized. You can use `normalize` to normalize a scalar, elements of a matrix, or elements of a multi-dimensional array of quaternions. Create a column vector of quaternions, then normalize them.

```
quatArray = quaternion([1,2,3,4; ...
                        2,3,4,1; ...
                        3,4,1,2]);
quatArrayNormalized = normalize(quatArray)
```

```
quatArrayNormalized = 3x1 quaternion array
     0.18257 + 0.36515i + 0.54772j +  0.7303k
     0.36515 + 0.54772i +  0.7303j + 0.18257k
     0.54772 +  0.7303i + 0.18257j + 0.36515k
```

Input Arguments

`quat` — Quaternion to normalize
scalar | vector | matrix | multidimensional array

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

Data Types: `quaternion`

Output Arguments

`quatNormalized` — Normalized quaternion
scalar | vector | matrix | multidimensional array

Normalized quaternion, returned as a quaternion or array of quaternions the same size as `quat`.

Data Types: `quaternion`

Version History

Introduced in R2020b

Extended Capabilities

C/C++ Code Generation

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

See Also

Functions

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

Objects

`quaternion`

ones

Create quaternion array with real parts set to one and imaginary parts set to zero

Syntax

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

quatOnes = ones( ___ ,'like',prototype,'quaternion')
```

Description

`quatOnes = ones('quaternion')` returns a scalar quaternion with the real part set to 1 and the imaginary parts set to 0.

Given a quaternion of the form $Q = a + b\text{i} + c\text{j} + d\text{k}$, a quaternion one is defined as $Q = 1 + 0\text{i} + 0\text{j} + 0\text{k}$.

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

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

Example: `ones([1,4,2],'quaternion')` returns a 1-by-4-by-2 array of quaternions with the real parts set to `1` and the imaginary parts set to `0`.

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

`quatOnes = ones( ___ ,'like',prototype,'quaternion')` specifies the underlying class of the returned quaternion array to be the same as the underlying class of the quaternion prototype.

Examples

Quaternion Scalar One

Create a quaternion scalar one.

```
quatOnes = ones('quaternion')
```

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

Square Matrix of Quaternion Ones

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

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

```
quatOnes = 3x3 quaternion array
     1 + 0i + 0j + 0k     1 + 0i + 0j + 0k     1 + 0i + 0j + 0k
     1 + 0i + 0j + 0k     1 + 0i + 0j + 0k     1 + 0i + 0j + 0k
     1 + 0i + 0j + 0k     1 + 0i + 0j + 0k     1 + 0i + 0j + 0k
```

Multidimensional Array of Quaternion Ones

Create a multidimensional array of quaternion ones by defining array dimensions in order. In this example, you create a 3-by-1-by-2 array. You can specify dimensions using a row vector or comma-separated integers. Specify the dimensions using a row vector and display the results:

```
dims = [3,1,2];
quatOnesSyntax1 = ones(dims,'quaternion')
```

```
quatOnesSyntax1 = 3x1x2 quaternion array
quatOnesSyntax1(:,:,1) =

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


quatOnesSyntax1(:,:,2) =

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

Specify the dimensions using comma-separated integers, and then verify the equivalency of the two syntaxes:

```
quatOnesSyntax2 = ones(3,1,2,'quaternion');
isequal(quatOnesSyntax1,quatOnesSyntax2)
```

```
ans = logical
   1
```

Underlying Class of Quaternion Ones

A quaternion is a four-part hyper-complex number used in three-dimensional rotations and orientations. You can specify the underlying data type of the parts as `single` or `double`. The default is `double`.

Create a quaternion array of ones with the underlying data type set to `single`.

```
quatOnes = ones(2,'like',single(1),'quaternion')
```

```
quatOnes = 2x2 quaternion array
     1 + 0i + 0j + 0k     1 + 0i + 0j + 0k
     1 + 0i + 0j + 0k     1 + 0i + 0j + 0k
```

Verify the underlying class using the `classUnderlying` function.

```
classUnderlying(quatOnes)
```

```
ans =
'single'
```

Input Arguments

n — Size of square quaternion matrix
integer value

Size of square quaternion matrix, specified as an integer value.

If `n` is zero or negative, then `quatOnes` is returned as an empty matrix.

Example: `ones(4,'quaternion')` returns a 4-by-4 matrix of quaternions with the real parts set to `1` and the imaginary parts set to `0`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

sz — Output size
row vector of integer values

Output size, specified as a row vector of integer values. Each element of `sz` indicates the size of the corresponding dimension in `quatOnes`. If the size of any dimension is `0` or negative, then `quatOnes` is returned as an empty array.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

prototype — Quaternion prototype
variable

Quaternion prototype, specified as a variable.

Example: `ones(2,'like',quat,'quaternion')` returns a 2-by-2 matrix of quaternions with the same underlying class as the prototype quaternion, `quat`.

Data Types: `quaternion`

sz1,...,szN — Size of each dimension
two or more integer values

Size of each dimension, specified as two or more integers. If the size of any dimension is `0` or negative, then `quatOnes` is returned as an empty array.

Example: `ones(2,3,'quaternion')` returns a 2-by-3 matrix of quaternions with the real parts set to `1` and the imaginary parts set to `0`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Output Arguments

`quatOnes` — Quaternion ones
scalar | vector | matrix | multidimensional array

Quaternion ones, returned as a scalar, vector, matrix, or multidimensional array of quaternions.

Given a quaternion of the form $Q = a + b\text{i} + c\text{j} + d\text{k}$, a quaternion one is defined as $Q = 1 + 0\text{i} + 0\text{j} + 0\text{k}$.

Data Types: `quaternion`

Version History

Introduced in R2020b

Extended Capabilities

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

See Also

Functions
`zeros`

Objects
`quaternion`

parts

Extract quaternion parts

Syntax

`[a,b,c,d] = parts(quat)`

Description

`[a,b,c,d] = parts(quat)` returns the parts of the quaternion array as arrays, each the same size as `quat`.

Examples

Convert Quaternion to Matrix of Quaternion Parts

Convert a quaternion representation to parts using the `parts` function.

Create a two-element column vector of quaternions by specifying the parts.

```
quat = quaternion([1:4;5:8])
```

```
quat = 2x1 quaternion array
     1 + 2i + 3j + 4k
     5 + 6i + 7j + 8k
```

Recover the parts from the quaternion matrix using the `parts` function. The parts are returned as separate output arguments, each the same size as the input 2-by-1 column vector of quaternions.

```
[qA,qB,qC,qD] = parts(quat)
```

```
qA = 2×1

     1
     5


qB = 2×1

     2
     6


qC = 2×1

     3
     7


qD = 2×1
```

```
     4
     8
```

Input Arguments

`quat` — Quaternion
scalar | vector | matrix | multidimensional array

Quaternion, specified as a quaternion or array of quaternions.

Data Types: `quaternion`

Output Arguments

`[a,b,c,d]` — Quaternion parts
scalar | vector | matrix | multidimensional array

Quaternion parts, returned as four arrays: a, b, c, and d. Each part is the same size as `quat`.

Data Types: `single` | `double`

Version History

Introduced in R2020b

Extended Capabilities

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

See Also

Functions
`classUnderlying` | `compact`

Objects
`quaternion`

power, .^

Element-wise quaternion power

Syntax

```
C = A.^b
```

Description

`C = A.^b` raises each element of `A` to the corresponding power in `b`.

Examples

Raise a Quaternion to a Real Scalar Power

Create a quaternion and raise it to a real scalar power.

```
A = quaternion(1,2,3,4)
```

```
A = quaternion
     1 + 2i + 3j + 4k
```

```
b = 3;
C = A.^b
```

```
C = quaternion
     -86 -  52i -  78j - 104k
```

Raise a Quaternion Array to Powers from a Multidimensional Array

Create a 2-by-1 quaternion array and raise it to powers from a 2-D array.

```
A = quaternion([1:4;5:8])
```

```
A = 2x1 quaternion array
     1 + 2i + 3j + 4k
     5 + 6i + 7j + 8k
```

```
b = [1 0 2; 3 2 1]
```

```
b = 2×3

     1     0     2
     3     2     1
```

```
C = A.^b
```

```
C = 2x3 quaternion array
       1 +    2i +    3j +    4k        1 +    0i +    0j +    0k      -28 +    4i +    6j +
    -2110 -  444i -  518j -  592k     -124 +   60i +   70j +   80k        5 +    6i +    7j +
```

Input Arguments

A — Base
scalar | vector | matrix | multidimensional array

Base, specified as a scalar, vector, matrix, or multidimensional array.

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

b — Exponent
scalar | vector | matrix | multidimensional array

Exponent, specified as a real scalar, vector, matrix, or multidimensional array.

Data Types: `single` | `double`

Output Arguments

C — Result
scalar | vector | matrix | multidimensional array

Each element of quaternion A raised to the corresponding power in b, returned as a scalar, vector, matrix, or multidimensional array.

Data Types: `quaternion`

Algorithms

The polar representation of a quaternion $A = a + b\mathrm{i} + c\mathrm{j} + d\mathrm{k}$ is given by

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

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

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

Version History

Introduced in R2020b

Extended Capabilities

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

See Also

Functions
`log` | `exp`

Objects
`quaternion`

prod

Product of a quaternion array

Syntax

```
quatProd = prod(quat)
quatProd = prod(quat,dim)
```

Description

`quatProd = prod(quat)` returns the quaternion product of the elements of the array.

`quatProd = prod(quat,dim)` calculates the quaternion product along dimension `dim`.

Examples

Product of Quaternions in Each Column

Create a 3-by-3 array whose elements correspond to their linear indices.

```
A = reshape(quaternion(randn(9,4)),3,3)
```

```
A = 3x3 quaternion array
      0.53767 +   2.7694i +    1.409j -  0.30344k      0.86217 +   0.7254i -   1.2075j +   0.888
       1.8339 -   1.3499i +   1.4172j +  0.29387k      0.31877 - 0.063055i +  0.71724j -   1.147
      -2.2588 +   3.0349i +   0.6715j -  0.78728k      -1.3077 +  0.71474i +   1.6302j -   1.068
```

Find the product of the quaternions in each column. The length of the first dimension is `1`, and the length of the second dimension matches `size(A,2)`.

```
B = prod(A)
```

```
B = 1x3 quaternion array
    -19.837 -  9.1521i +  15.813j -  19.918k     -5.4708 - 0.28535i +   3.077j -  1.2295k
```

Product of Specified Dimension of Quaternion Array

You can specify which dimension of a quaternion array to take the product of.

Create a 2-by-2-by-2 quaternion array.

```
A = reshape(quaternion(randn(8,4)),2,2,2);
```

Find the product of the elements in each page of the array. The length of the first dimension matches `size(A,1)`, the length of the second dimension matches `size(A,2)`, and the length of the third dimension is `1`.

```
dim = 3;
B = prod(A,dim)

B = 2x2 quaternion array
    -2.4847 +  1.1659i - 0.37547j +  2.8068k     0.28786 - 0.29876i - 0.51231j -  4.2972k
    0.38986 -  3.6606i -  2.0474j -   6.047k      -1.741 - 0.26782i +  5.4346j +  4.1452k
```

Input Arguments

quat — Quaternion
scalar | vector | matrix | multidimensional array

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

Example: `qProd = prod(quat)` calculates the quaternion product along the first non-singleton dimension of `quat`.

Data Types: `quaternion`

dim — Dimension
first non-singleton dimension (default) | positive integer

Dimension along which to calculate the quaternion product, specified as a positive integer. If `dim` is not specified, `prod` operates along the first non-singleton dimension of `quat`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Output Arguments

quatProd — Quaternion product
positive integer

Quaternion product, returned as quaternion array with one less non-singleton dimension than `quat`.

For example, if `quat` is a 2-by-2-by-5 array,

- `prod(quat,1)` returns a 1-by-2-by-5 array.
- `prod(quat,2)` returns a 2-by-1-by-5 array.
- `prod(quat,3)` returns a 2-by-2 array.

Data Types: `quaternion`

Version History

Introduced in R2020b

Extended Capabilities

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

See Also

Functions
`mtimes` | `.*,times`

Objects
`quaternion`

rdivide, ./

Element-wise quaternion right division

Syntax

`C = A./B`

Description

`C = A./B` performs quaternion element-wise division by dividing each element of quaternion `A` by the corresponding element of quaternion `B`.

Examples

Divide a Quaternion Array by a Real Scalar

Create a 2-by-1 quaternion array, and divide it element-by-element by a real scalar.

```
A = quaternion([1:4;5:8])
```

```
A = 2x1 quaternion array
     1 + 2i + 3j + 4k
     5 + 6i + 7j + 8k
```

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

```
C = 2x1 quaternion array
     0.5 +   1i + 1.5j +   2k
     2.5 +   3i + 3.5j +   4k
```

Divide a Quaternion Array by Another Quaternion Array

Create a 2-by-2 quaternion array, and divide it element-by-element by another 2-by-2 quaternion array.

```
q1 = quaternion(magic(4));
A = reshape(q1,2,2)
```

```
A = 2x2 quaternion array
     16 +  2i +  3j + 13k      9 +  7i +  6j + 12k
      5 + 11i + 10j +  8k      4 + 14i + 15j +  1k
```

```
q2 = quaternion([1:4;3:6;2:5;4:7]);
B = reshape(q2,2,2)
```

```
B = 2x2 quaternion array
     1 + 2i + 3j + 4k     2 + 3i + 4j + 5k
     3 + 4i + 5j + 6k     4 + 5i + 6j + 7k
```

```
C = A./B
```

```
C = 2x2 quaternion array
          2.7 -      0.1i -      2.1j -      1.7k       2.2778 + 0.092593i -  0.46296j -  0.5740
       1.8256 - 0.081395i +  0.45349j -  0.24419k       1.4524 -      0.5i +   1.0238j -   0.261
```

Input Arguments

A — Dividend
scalar | vector | matrix | multidimensional array

Dividend, specified as a quaternion, an array of quaternions, a real scalar, or an array of real numbers.

A and B must have compatible sizes. In the simplest cases, they can be the same size or one can be a scalar. Two inputs have compatible sizes if, for every dimension, the dimension sizes of the inputs are the same or one of the dimensions is 1.

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

B — Divisor
scalar | vector | matrix | multidimensional array

Divisor, specified as a quaternion, an array of quaternions, a real scalar, or an array of real numbers.

A and B must have compatible sizes. In the simplest cases, they can be the same size or one can be a scalar. Two inputs have compatible sizes if, for every dimension, the dimension sizes of the inputs are the same or one of the dimensions is 1.

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

Output Arguments

C — Result
scalar | vector | matrix | multidimensional array

Result of quaternion division, returned as a scalar, vector, matrix, or multidimensional array.

Data Types: `quaternion`

Algorithms

Quaternion Division

Given a quaternion $A = a_1 + a_2\mathrm{i} + a_3\mathrm{j} + a_4\mathrm{k}$ and a real scalar p,

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

Note For a real scalar *p*, *A./p* = *A.\p*.

Quaternion Division by a Quaternion Scalar

Given two quaternions *A* and *B* of compatible sizes,

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

Version History

Introduced in R2020b

Extended Capabilities

C/C++ Code Generation

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

See Also

Functions

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

Objects

`quaternion`

randrot

Uniformly distributed random rotations

Syntax

```
R = randrot
R = randrot(m)
R = randrot(m1,...,mN)
R = randrot([m1,...,mN])
```

Description

`R = randrot` returns a unit quaternion drawn from a uniform distribution of random rotations.

`R = randrot(m)` returns an `m`-by-`m` matrix of unit quaternions drawn from a uniform distribution of random rotations.

`R = randrot(m1,...,mN)` returns an `m1`-by-...-by-`mN` array of random unit quaternions, where `m1`, ..., `mN` indicate the size of each dimension. For example, `randrot(3,4)` returns a 3-by-4 matrix of random unit quaternions.

`R = randrot([m1,...,mN])` returns an `m1`-by-...-by-`mN` array of random unit quaternions, where `m1`,..., `mN` indicate the size of each dimension. For example, `randrot([3,4])` returns a 3-by-4 matrix of random unit quaternions.

Examples

Matrix of Random Rotations

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

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

```
r = 3x3 quaternion array
      0.17446 +  0.59506i -  0.73295j +  0.27976k      0.69704 - 0.060589i +  0.68679j -  0.1969
      0.21908 -  0.89875i -    0.298j +  0.23548k    -0.049744 +  0.59691i +  0.56459j +  0.5678
       0.6375 +  0.49338i -  0.24049j +  0.54068k       0.2979 -  0.53568i +  0.31819j +  0.7232
```

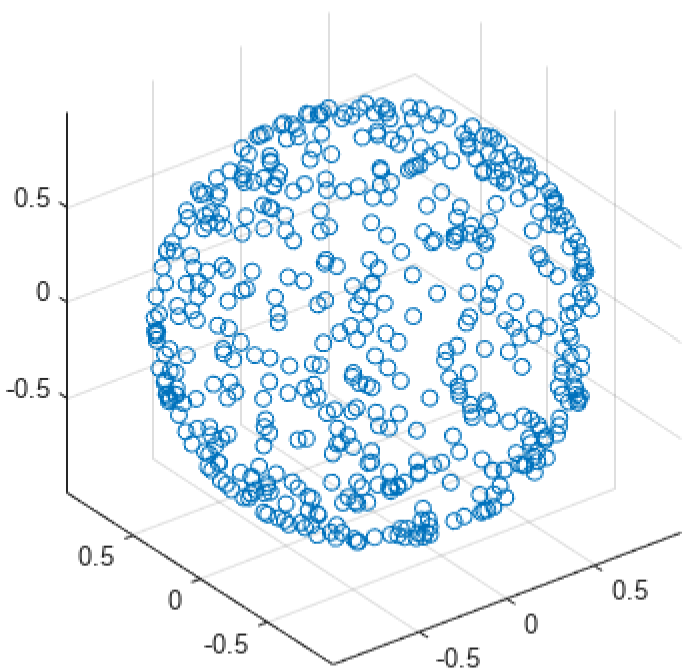
Create Uniform Distribution of Random Rotations

Create a vector of 500 random quaternions. Use `rotatepoint` 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`

Version History

Introduced in R2020b

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`

rotateframe

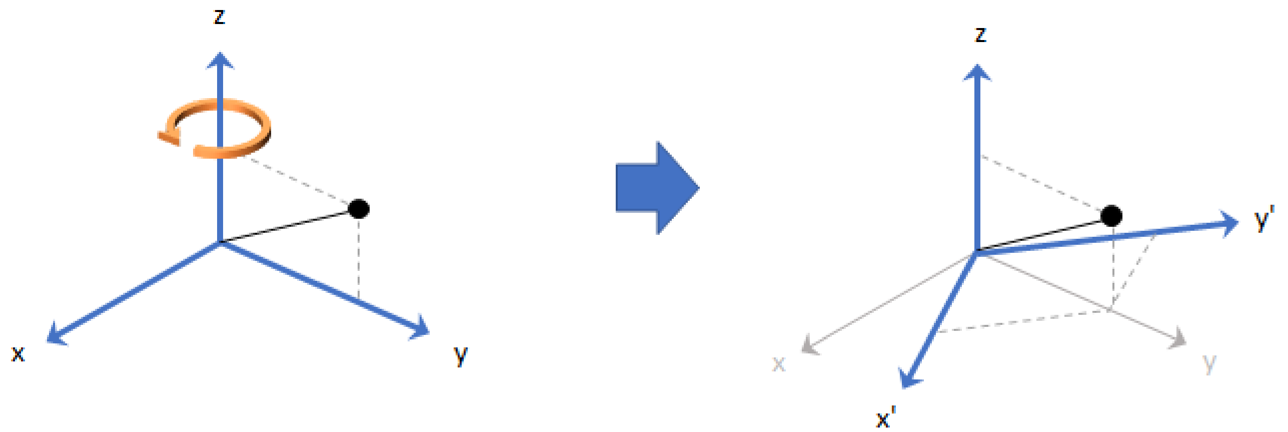
Quaternion frame rotation

Syntax

```
rotationResult = rotateframe(quat,cartesianPoints)
```

Description

`rotationResult = rotateframe(quat,cartesianPoints)` rotates the frame of reference for the Cartesian points using the quaternion, `quat`. The elements of the quaternion are normalized before use in the rotation.

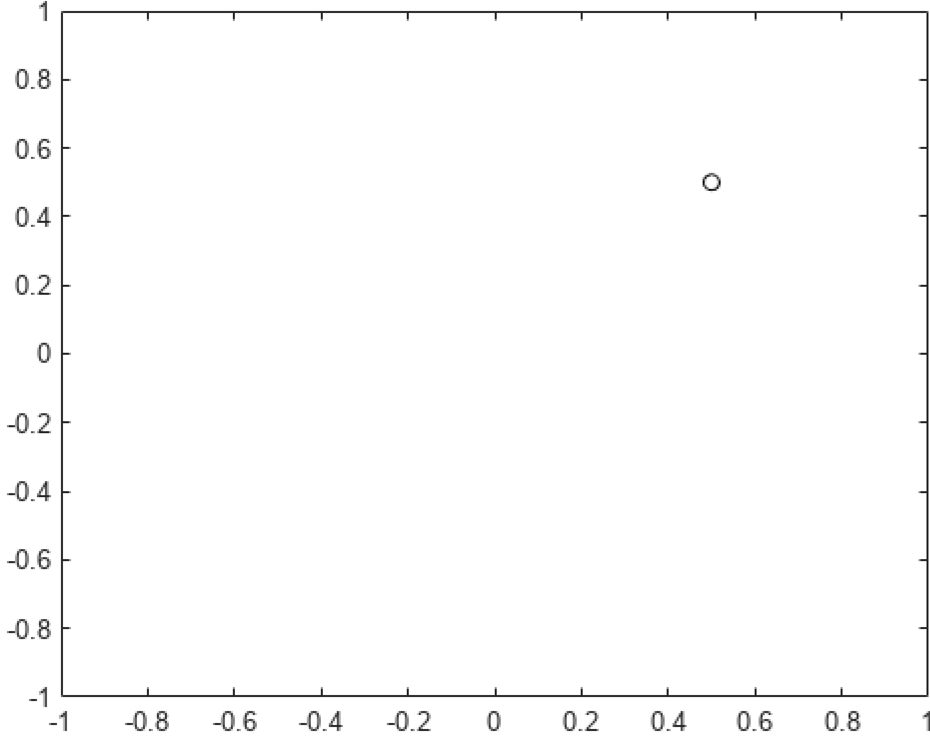


Examples

Rotate Frame Using Quaternion Vector

Define a point in three dimensions. The coordinates of a point are always specified in the order *x*, *y*, and *z*. For convenient visualization, define the point on the *x*-*y* plane.

```
x = 0.5;
y = 0.5;
z = 0;
plot(x,y,'ko')
hold on
axis([-1 1 -1 1])
```



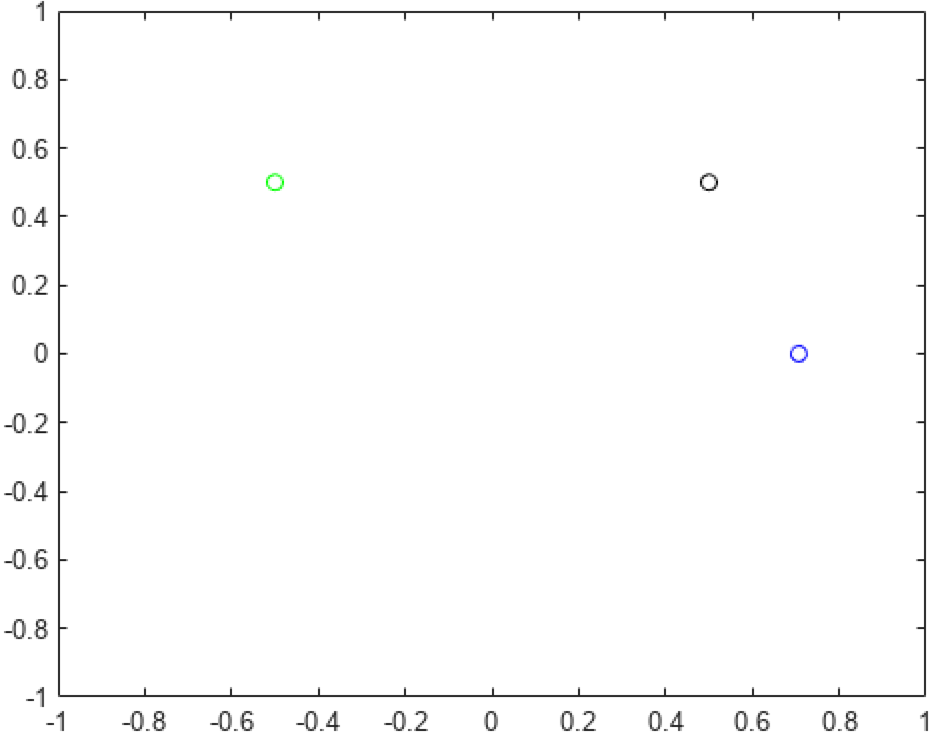
Create a quaternion vector specifying two separate rotations, one to rotate the frame 45 degrees and another to rotate the point -90 degrees about the *z*-axis. Use `rotateframe` to perform the rotations.

```
quat = quaternion([0,0,pi/4; ...
                   0,0,-pi/2],'euler','XYZ','frame');

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

```
rereferencedPoint = 2×3

    0.7071   -0.0000         0
   -0.5000    0.5000         0
```

Plot the rereferenced points.

```
plot(rereferencedPoint(1,1),rereferencedPoint(1,2),'bo')
plot(rereferencedPoint(2,1),rereferencedPoint(2,2),'go')
```



Rereference Group of Points using Quaternion

Define two points in three-dimensional space. Define a quaternion to rereference the points by first rotating the reference frame about the *z*-axis 30 degrees and then about the new *y*-axis 45 degrees.

```
a = [1,0,0];
b = [0,1,0];
quat = quaternion([30,45,0],'eulerd','ZYX','point');
```

Use `rotateframe` to reference both points using the quaternion rotation operator. Display the result.

```
rP = rotateframe(quat,[a;b])
```

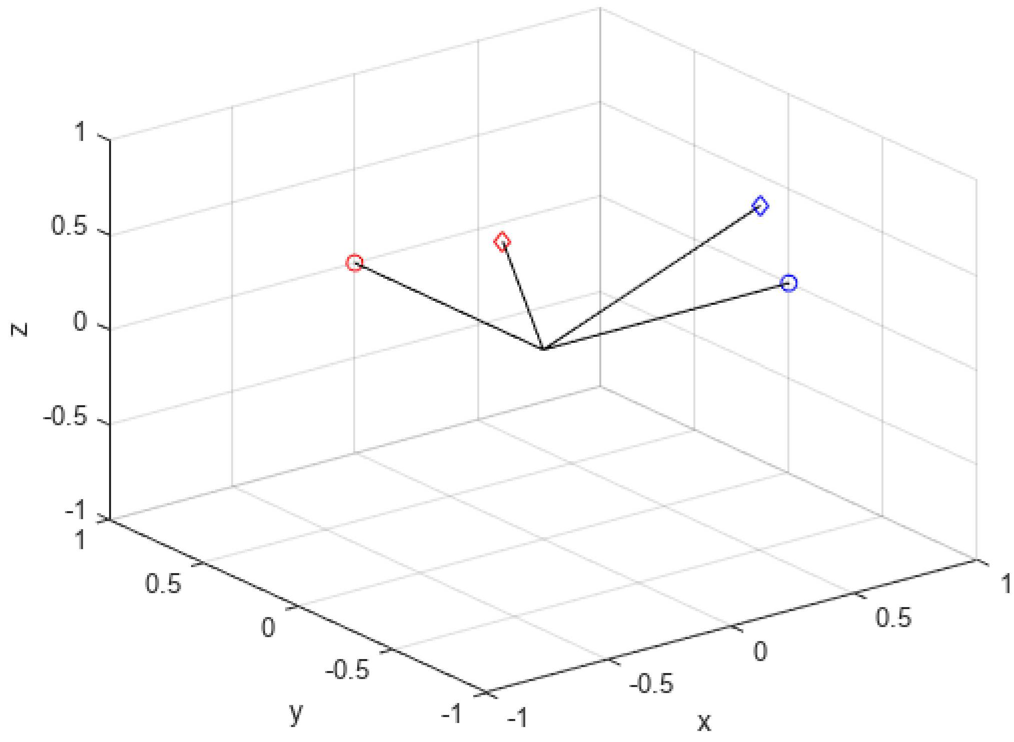
```
rP = 2×3

    0.6124   -0.3536    0.7071
    0.5000    0.8660   -0.0000
```

Visualize the original orientation and the rotated orientation of the points. Draw lines from the origin to each of the points for visualization purposes.

```
plot3(a(1),a(2),a(3),'bo');
```

```
hold on
```

```
grid on
axis([-1 1 -1 1 -1 1])
xlabel('x')
ylabel('y')
zlabel('z')

plot3(b(1),b(2),b(3),'ro');
plot3(rP(1,1),rP(1,2),rP(1,3),'bd')
plot3(rP(2,1),rP(2,2),rP(2,3),'rd')

plot3([0;rP(1,1)],[0;rP(1,2)],[0;rP(1,3)],'k')
plot3([0;rP(2,1)],[0;rP(2,2)],[0;rP(2,3)],'k')
plot3([0;a(1)],[0;a(2)],[0;a(3)],'k')
plot3([0;b(1)],[0;b(2)],[0;b(3)],'k')
```



Input Arguments

quat — Quaternion that defines rotation
scalar | vector

Quaternion that defines rotation, specified as a scalar quaternion or vector of quaternions.

Data Types: `quaternion`

`cartesianPoints` — Three-dimensional Cartesian points
1-by-3 vector | *N*-by-3 matrix

Three-dimensional Cartesian points, specified as a 1-by-3 vector or *N*-by-3 matrix.

Data Types: `single` | `double`

Output Arguments

`rotationResult` — Re-referenced Cartesian points

vector | matrix

Cartesian points defined in reference to rotated reference frame, returned as a vector or matrix the same size as `cartesianPoints`.

The data type of the re-referenced Cartesian points is the same as the underlying data type of `quat`.

Data Types: `single` | `double`

Algorithms

Quaternion frame rotation re-references a point specified in $\mathbf{R}^3$ by rotating the original frame of reference according to a specified quaternion:

$$L_q(u) = q^* u q$$

where q is the quaternion, * represents conjugation, and u is the point to rotate, specified as a quaternion.

For convenience, the `rotateframe` function takes a point in $\mathbf{R}^3$ and returns a point in $\mathbf{R}^3$. Given a function call with some arbitrary quaternion, $q = a + b\text{i} + c\text{j} + d\text{k}$, and arbitrary coordinate, $[x,y,z]$,

```
point = [x,y,z];
rereferencedPoint = rotateframe(q,point)
```

the `rotateframe` function performs the following operations:

1. Converts point $[x,y,z]$ to a quaternion:

   $$u_q = 0 + xi + yj + zk$$

2. Normalizes the quaternion, q:

   $$q_n = \frac{q}{\sqrt{a^2 + b^2 + c^2 + d^2}}$$

3. Applies the rotation:

   $$v_q = q^* u_q q$$

4. Converts the quaternion output, v_q, back to $\mathbf{R}^3$

Version History

Introduced in R2020b

Extended Capabilities

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

See Also

Functions
`rotatepoint`

Objects
`quaternion`

rotatepoint

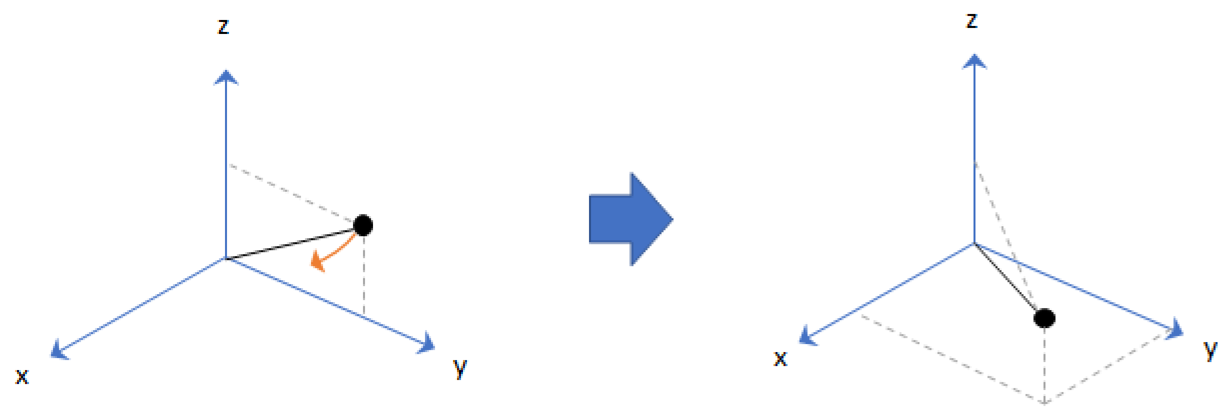
Quaternion point rotation

Syntax

```
rotationResult = rotatepoint(quat,cartesianPoints)
```

Description

`rotationResult = rotatepoint(quat,cartesianPoints)` rotates the Cartesian points using the quaternion, `quat`. The elements of the quaternion are normalized before use in the rotation.

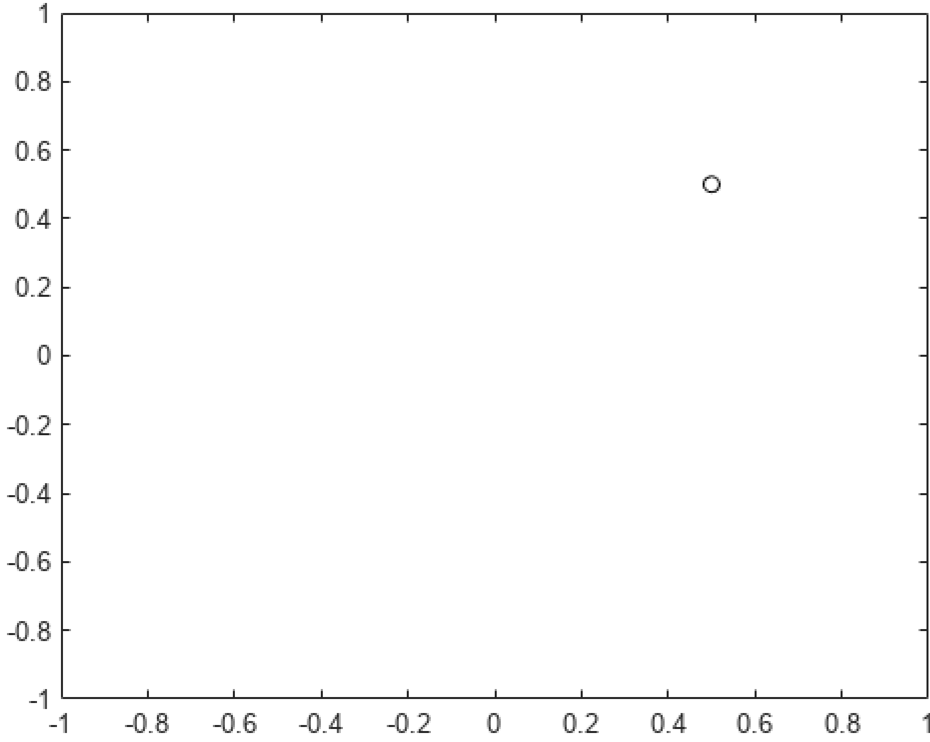


Examples

Rotate Point Using Quaternion Vector

Define a point in three dimensions. The coordinates of a point are always specified in order *x*, *y*, *z*. For convenient visualization, define the point on the *x*-*y* plane.

```
x = 0.5;
y = 0.5;
z = 0;

plot(x,y,'ko')
hold on
axis([-1 1 -1 1])
```



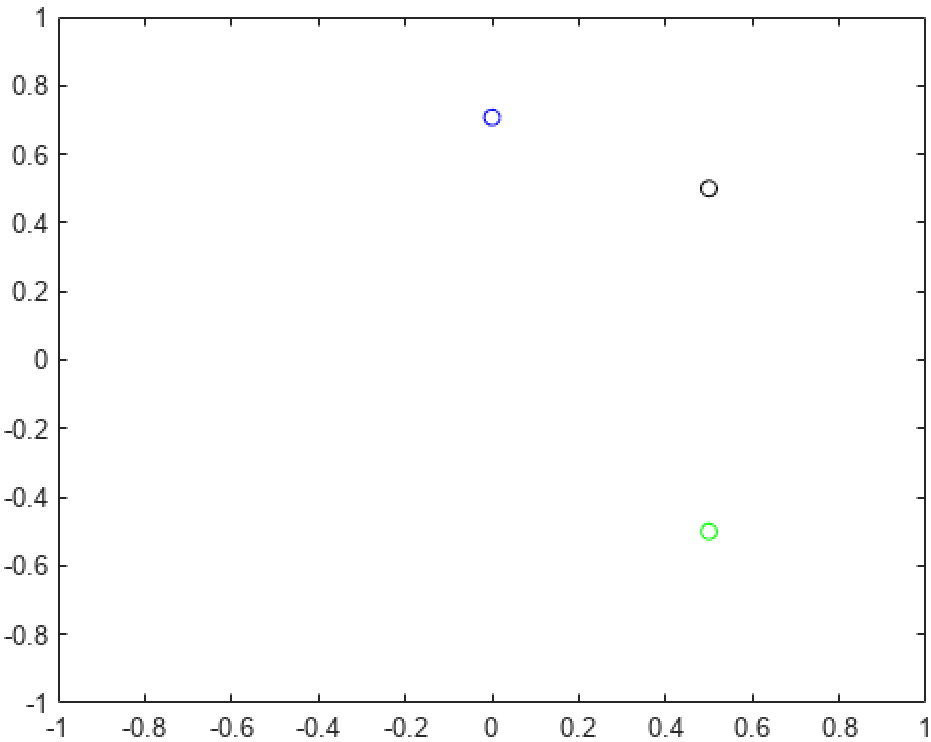
Create a quaternion vector specifying two separate rotations, one to rotate the point 45 and another to rotate the point -90 degrees about the *z*-axis. Use `rotatepoint` to perform the rotation.

```
quat = quaternion([0,0,pi/4; ...
                   0,0,-pi/2],'euler','XYZ','point');

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

```
rotatedPoint = 2×3

   -0.0000    0.7071         0
    0.5000   -0.5000         0
```

Plot the rotated points.

```
plot(rotatedPoint(1,1),rotatedPoint(1,2),'bo')
plot(rotatedPoint(2,1),rotatedPoint(2,2),'go')
```



Rotate Group of Points Using Quaternion

Define two points in three-dimensional space. Define a quaternion to rotate the point by first rotating about the *z*-axis 30 degrees and then about the new *y*-axis 45 degrees.

```
a = [1,0,0];
b = [0,1,0];
quat = quaternion([30,45,0],'eulerd','ZYX','point');
```

Use `rotatepoint` to rotate both points using the quaternion rotation operator. Display the result.

```
rP = rotatepoint(quat,[a;b])
```

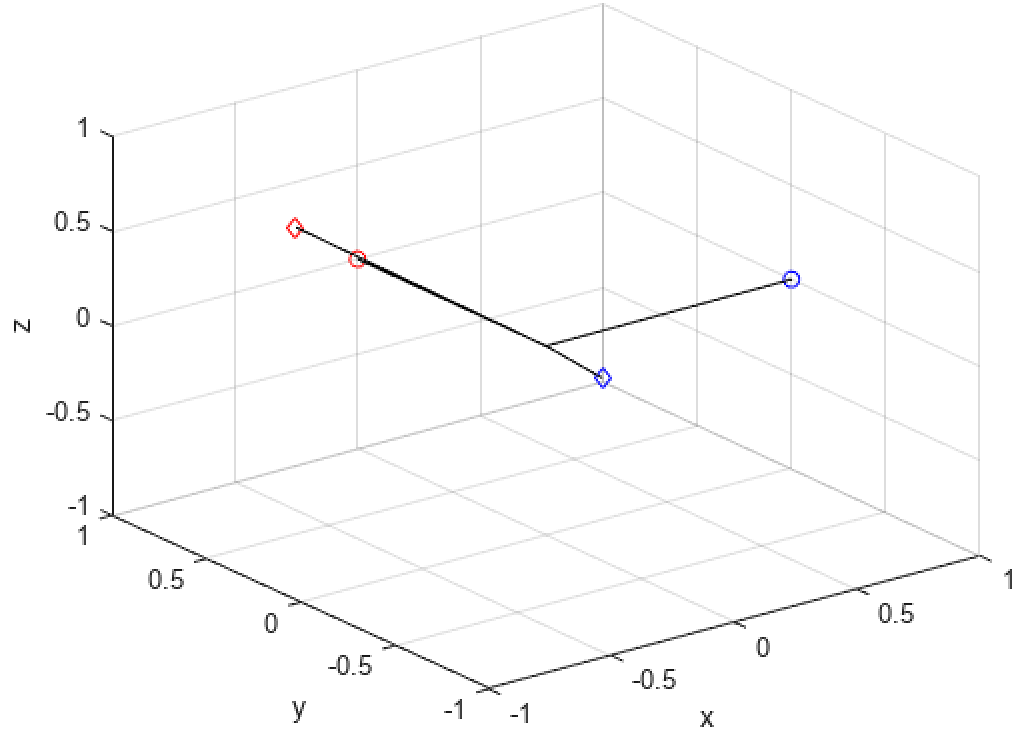
```
rP = 2×3

    0.6124    0.5000   -0.6124
   -0.3536    0.8660    0.3536
```

Visualize the original orientation and the rotated orientation of the points. Draw lines from the origin to each of the points for visualization purposes.

```
plot3(a(1),a(2),a(3),'bo');
```

```
hold on
```

```
grid on
axis([-1 1 -1 1 -1 1])
xlabel('x')
ylabel('y')
zlabel('z')

plot3(b(1),b(2),b(3),'ro');
plot3(rP(1,1),rP(1,2),rP(1,3),'bd')
plot3(rP(2,1),rP(2,2),rP(2,3),'rd')

plot3([0;rP(1,1)],[0;rP(1,2)],[0;rP(1,3)],'k')
plot3([0;rP(2,1)],[0;rP(2,2)],[0;rP(2,3)],'k')
plot3([0;a(1)],[0;a(2)],[0;a(3)],'k')
plot3([0;b(1)],[0;b(2)],[0;b(3)],'k')
```



Input Arguments

quat — Quaternion that defines rotation

scalar | vector

Quaternion that defines rotation, specified as a scalar quaternion, row vector of quaternions, or column vector of quaternions.

Data Types: `quaternion`

`cartesianPoints` — Three-dimensional Cartesian points
1-by-3 vector | *N*-by-3 matrix

Three-dimensional Cartesian points, specified as a 1-by-3 vector or *N*-by-3 matrix.

Data Types: `single` | `double`

Output Arguments

`rotationResult` — Repositioned Cartesian points
vector | matrix

Rotated Cartesian points defined using the quaternion rotation, returned as a vector or matrix the same size as `cartesianPoints`.

Data Types: `single` | `double`

Algorithms

Quaternion point rotation rotates a point specified in $\mathbf{R}^3$ according to a specified quaternion:

$$L_q(u) = quq^*$$

where q is the quaternion, * represents conjugation, and u is the point to rotate, specified as a quaternion.

For convenience, the `rotatepoint` function takes in a point in $\mathbf{R}^3$ and returns a point in $\mathbf{R}^3$. Given a function call with some arbitrary quaternion, $q = a + b\text{i} + c\text{j} + d\text{k}$, and arbitrary coordinate, $[x,y,z]$, for example,

```
rereferencedPoint = rotatepoint(q,[x,y,z])
```

the `rotatepoint` function performs the following operations:

1. Converts point $[x,y,z]$ to a quaternion:

   $$u_q = 0 + xi + yj + zk$$

2. Normalizes the quaternion, q:

   $$q_n = \frac{q}{\sqrt{a^2 + b^2 + c^2 + d^2}}$$

3. Applies the rotation:

   $$v_q = qu_qq^*$$

4. Converts the quaternion output, v_q, back to $\mathbf{R}^3$

Version History

Introduced in R2020b

Extended Capabilities

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

See Also

Functions
`rotateframe`

Objects
`quaternion`

rotmat

Convert quaternion to rotation matrix

Syntax

```
rotationMatrix = rotmat(quat,rotationType)
```

Description

`rotationMatrix = rotmat(quat,rotationType)` converts the quaternion, `quat`, to an equivalent rotation matrix representation.

Examples

Convert Quaternion to Rotation Matrix for Point Rotation

Define a quaternion for use in point rotation.

```
theta = 45;
gamma = 30;
quat = quaternion([0,theta,gamma],'eulerd','ZYX','point')
```

```
quat = quaternion
       0.8924 +  0.23912i +  0.36964j + 0.099046k
```

Convert the quaternion to a rotation matrix.

```
rotationMatrix = rotmat(quat,'point')
```

```
rotationMatrix = 3×3

    0.7071   -0.0000    0.7071
    0.3536    0.8660   -0.3536
   -0.6124    0.5000    0.6124
```

To verify the rotation matrix, directly create two rotation matrices corresponding to the rotations about the *y*- and *x*-axes. Multiply the rotation matrices and compare to the output of `rotmat`.

```
theta = 45;
gamma = 30;

ry = [cosd(theta)   0           sind(theta) ; ...
      0             1           0           ; ...
     -sind(theta)   0           cosd(theta)];

rx = [1             0           0           ;      ...
      0             cosd(gamma) -sind(gamma) ;     ...
      0             sind(gamma) cosd(gamma)];

rotationMatrixVerification = rx*ry
```

```
rotationMatrixVerification = 3×3

    0.7071         0    0.7071
    0.3536    0.8660   -0.3536
   -0.6124    0.5000    0.6124
```

Convert Quaternion to Rotation Matrix for Frame Rotation

Define a quaternion for use in frame rotation.

```
theta = 45;
gamma = 30;
quat = quaternion([0,theta,gamma],'eulerd','ZYX','frame')
```

```
quat = quaternion
       0.8924 +  0.23912i +  0.36964j - 0.099046k
```

Convert the quaternion to a rotation matrix.

```
rotationMatrix = rotmat(quat,'frame')
```

```
rotationMatrix = 3×3

    0.7071   -0.0000   -0.7071
    0.3536    0.8660    0.3536
    0.6124   -0.5000    0.6124
```

To verify the rotation matrix, directly create two rotation matrices corresponding to the rotations about the *y*- and *x*-axes. Multiply the rotation matrices and compare to the output of `rotmat`.

```
theta = 45;
gamma = 30;

ry = [cosd(theta)   0           -sind(theta) ; ...
      0             1           0            ; ...
     sind(theta)   0           cosd(theta)];

rx = [1             0           0            ;      ...
      0             cosd(gamma) sind(gamma) ;     ...
      0             -sind(gamma) cosd(gamma)];

rotationMatrixVerification = rx*ry
```

```
rotationMatrixVerification = 3×3

    0.7071         0   -0.7071
    0.3536    0.8660    0.3536
    0.6124   -0.5000    0.6124
```

Convert Quaternion Vector to Rotation Matrices

Create a 3-by-1 normalized quaternion vector.

```
qVec = normalize(quaternion(randn(3,4)));
```

Convert the quaternion array to rotation matrices. The pages of `rotmatArray` correspond to the linear index of `qVec`.

```
rotmatArray = rotmat(qVec,'frame');
```

Assume `qVec` and `rotmatArray` correspond to a sequence of rotations. Combine the quaternion rotations into a single representation, then apply the quaternion rotation to arbitrarily initialized Cartesian points.

```
loc = normalize(randn(1,3));
quat = prod(qVec);
rotateframe(quat,loc)
```

```
ans = 1×3

    0.9524    0.5297    0.9013
```

Combine the rotation matrices into a single representation, then apply the rotation matrix to the same initial Cartesian points. Verify the quaternion rotation and rotation matrix result in the same orientation.

```
totalRotMat = eye(3);
for i = 1:size(rotmatArray,3)
    totalRotMat = rotmatArray(:,:,i)*totalRotMat;
end
totalRotMat*loc'
```

```
ans = 3×1

    0.9524
    0.5297
    0.9013
```

Input Arguments

quat — Quaternion to convert
scalar | vector | matrix | multidimensional array

Quaternion to convert, specified as a scalar, vector, matrix, or multidimensional array.

Data Types: `quaternion`

`rotationType` — Type or rotation
`'frame'` | `'point'`

Type of rotation represented by the `rotationMatrix` output, specified as `'frame'` or `'point'`.

Data Types: `char` | `string`

Output Arguments

`rotationMatrix` — Rotation matrix representation
3-by-3 matrix | 3-by-3-by-*N* multidimensional array

Rotation matrix representation, returned as a 3-by-3 matrix or 3-by-3-by-*N* multidimensional array.

- If `quat` is a scalar, `rotationMatrix` is returned as a 3-by-3 matrix.
- If `quat` is non-scalar, `rotationMatrix` is returned as a 3-by-3-by-*N* multidimensional array, where `rotationMatrix(:,:,i)` is the rotation matrix corresponding to `quat(i)`.

The data type of the rotation matrix is the same as the underlying data type of `quat`.

Data Types: `single` | `double`

Algorithms

Given a quaternion of the form

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

the equivalent rotation matrix for frame rotation is defined as

$$\begin{bmatrix} 2a^2 - 1 + 2b^2 & 2bc + 2ad & 2bd - 2ac \\ 2bc - 2ad & 2a^2 - 1 + 2c^2 & 2cd + 2ab \\ 2bd + 2ac & 2cd - 2ab & 2a^2 - 1 + 2d^2 \end{bmatrix}.$$

The equivalent rotation matrix for point rotation is the transpose of the frame rotation matrix:

$$\begin{bmatrix} 2a^2 - 1 + 2b^2 & 2bc - 2ad & 2bd + 2ac \\ 2bc + 2ad & 2a^2 - 1 + 2c^2 & 2cd - 2ab \\ 2bd - 2ac & 2cd + 2ab & 2a^2 - 1 + 2d^2 \end{bmatrix}.$$

Version History

Introduced in R2020b

References

[1] Kuipers, Jack B. *Quaternions and Rotation Sequences: A Primer with Applications to Orbits, Aerospace, and Virtual Reality*. Princeton, NJ: Princeton University Press, 2007.

Extended Capabilities

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

See Also

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

Objects
`quaternion`

rotvec

Convert quaternion to rotation vector (radians)

Syntax

```
rotationVector = rotvec(quat)
```

Description

`rotationVector = rotvec(quat)` converts the quaternion array, `quat`, to an *N*-by-3 matrix of equivalent rotation vectors in radians. The elements of `quat` are normalized before conversion.

Examples

Convert Quaternion to Rotation Vector in Radians

Convert a random quaternion scalar to a rotation vector in radians

```
quat = quaternion(randn(1,4));
rotvec(quat)
```

```
ans = 1×3

    1.6866   -2.0774    0.7929
```

Input Arguments

`quat` — Quaternion to convert
scalar | vector | matrix | multidimensional array

Quaternion to convert, specified as scalar quaternion, vector, matrix, or multidimensional array of quaternions.

Data Types: `quaternion`

Output Arguments

`rotationVector` — Rotation vector (radians)
N-by-3 matrix

Rotation vector representation, returned as an *N*-by-3 matrix of rotations vectors, where each row represents the [X Y Z] angles of the rotation vectors in radians. The *i*th row of `rotationVector` corresponds to the element `quat(i)`.

The data type of the rotation vector is the same as the underlying data type of `quat`.

Data Types: `single` | `double`

Algorithms

All rotations in 3-D can be represented by a three-element axis of rotation and a rotation angle, for a total of four elements. If the rotation axis is constrained to be unit length, the rotation angle can be distributed over the vector elements to reduce the representation to three elements.

Recall that a quaternion can be represented in axis-angle form

$$q = \cos\left(\theta/2\right) + \sin\left(\theta/2\right)(\mathrm{xi} + y\mathrm{j} + z\mathrm{k}),$$

where θ is the angle of rotation and $[x,y,z]$ represent the axis of rotation.

Given a quaternion of the form

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

you can solve for the rotation angle using the axis-angle form of quaternions:

$$\theta = 2\cos^{-1}(a)\ .$$

Assuming a normalized axis, you can rewrite the quaternion as a rotation vector without loss of information by distributing θ over the parts b, c, and d. The rotation vector representation of q is

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

Version History

Introduced in R2020b

Extended Capabilities

C/C++ Code Generation

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

See Also

Functions

`rotvecd` | `euler` | `eulerd`

Objects

`quaternion`

rotvecd

Convert quaternion to rotation vector (degrees)

Syntax

```
rotationVector = rotvecd(quat)
```

Description

`rotationVector = rotvecd(quat)` converts the quaternion array, `quat`, to an *N*-by-3 matrix of equivalent rotation vectors in degrees. The elements of `quat` are normalized before conversion.

Examples

Convert Quaternion to Rotation Vector in Degrees

Convert a random quaternion scalar to a rotation vector in degrees.

```
quat = quaternion(randn(1,4));
rotvecd(quat)
```

```
ans = 1×3

   96.6345 -119.0274   45.4312
```

Input Arguments

`quat` — Quaternion to convert
scalar | vector | matrix | multidimensional array

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

Data Types: `quaternion`

Output Arguments

`rotationVector` — Rotation vector (degrees)
N-by-3 matrix

Rotation vector representation, returned as an *N*-by-3 matrix of rotation vectors, where each row represents the [*x y z*] angles of the rotation vectors in degrees. The *i*th row of `rotationVector` corresponds to the element `quat(i)`.

The data type of the rotation vector is the same as the underlying data type of `quat`.

Data Types: `single` | `double`

Algorithms

All rotations in 3-D can be represented by four elements: a three-element axis of rotation and a rotation angle. If the rotation axis is constrained to be unit length, the rotation angle can be distributed over the vector elements to reduce the representation to three elements.

Recall that a quaternion can be represented in axis-angle form

$$q = \cos\left(\theta/2\right) + \sin\left(\theta/2\right)(\mathrm{xi} + y\mathrm{j} + z\mathrm{k}),$$

where θ is the angle of rotation in degrees, and $[x,y,z]$ represent the axis of rotation.

Given a quaternion of the form

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

you can solve for the rotation angle using the axis-angle form of quaternions:

$$\theta = 2\cos^{-1}(a)\,.$$

Assuming a normalized axis, you can rewrite the quaternion as a rotation vector without loss of information by distributing θ over the parts b, c, and d. The rotation vector representation of q is

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

Version History

Introduced in R2020b

Extended Capabilities

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

See Also

Functions
`rotvec` | `euler` | `eulerd`

Objects
`quaternion`

slerp

Spherical linear interpolation

Syntax

```
q0 = slerp(q1,q2,T)
```

Description

`q0 = slerp(q1,q2,T)` spherically interpolates between `q1` and `q2` by the interpolation coefficient `T`. The function always chooses the shorter interpolation path between `q1` and `q2`.

Examples

Interpolate Between Two Quaternions

Create two quaternions with the following interpretation:

1. `a` = 45 degree rotation around the *z*-axis
2. `c` = -45 degree rotation around the *z*-axis

```
a = quaternion([45,0,0],'eulerd','ZYX','frame');
c = quaternion([-45,0,0],'eulerd','ZYX','frame');
```

Call `slerp` with the quaternions `a` and `c` and specify an interpolation coefficient of 0.5.

```
interpolationCoefficient = 0.5;

b = slerp(a,c,interpolationCoefficient);
```

The output of `slerp`, `b`, represents an average rotation of `a` and `c`. To verify, convert `b` to Euler angles in degrees.

```
averageRotation = eulerd(b,'ZYX','frame')
```

```
averageRotation = 1×3

     0     0     0
```

The interpolation coefficient is specified as a normalized value between `0` and `1`, inclusive. An interpolation coefficient of `0` corresponds to the `a` quaternion, and an interpolation coefficient of `1` corresponds to the `c` quaternion. Call `slerp` with coefficients `0` and `1` to confirm.

```
b = slerp(a,c,[0,1]);
eulerd(b,'ZYX','frame')
```

```
ans = 2×3

   45.0000         0         0
```

```
  -45.0000         0         0
```

You can create smooth paths between quaternions by specifying arrays of equally spaced interpolation coefficients.

```
path = 0:0.1:1;

interpolatedQuaternions = slerp(a,c,path);
```

For quaternions that represent rotation only about a single axis, specifying interpolation coefficients as equally spaced results in quaternions equally spaced in Euler angles. Convert `interpolatedQuaternions` to Euler angles and verify that the difference between the angles in the path is constant.

```
k = eulerd(interpolatedQuaternions,'ZYX','frame');
abc = abs(diff(k))
```

```
abc = 10×3

    9.0000         0         0
    9.0000         0         0
    9.0000         0         0
    9.0000         0         0
    9.0000         0         0
    9.0000         0         0
    9.0000         0         0
    9.0000         0         0
    9.0000         0         0
    9.0000         0         0
```

Alternatively, you can use the `dist` function to verify that the distance between the interpolated quaternions is consistent. The `dist` function returns angular distance in radians; convert to degrees for easy comparison.

```
def = rad2deg(dist(interpolatedQuaternions(2:end),interpolatedQuaternions(1:end-1)))
```

```
def = 1×10

    9.0000    9.0000    9.0000    9.0000    9.0000    9.0000    9.0000    9.0000    9.0000    9.0
```

SLERP Minimizes Great Circle Path

The SLERP algorithm interpolates along a great circle path connecting two quaternions. This example shows how the SLERP algorithm minimizes the great circle path.

Define four quaternions:

1. `q0` - quaternion indicating no rotation from the global frame
2. `q179` - quaternion indicating a 179 degree rotation about the *z*-axis
3. `q180` - quaternion indicating a 180 degree rotation about the *z*-axis

4 `q181` - quaternion indicating a 181 degree rotation about the *z*-axis

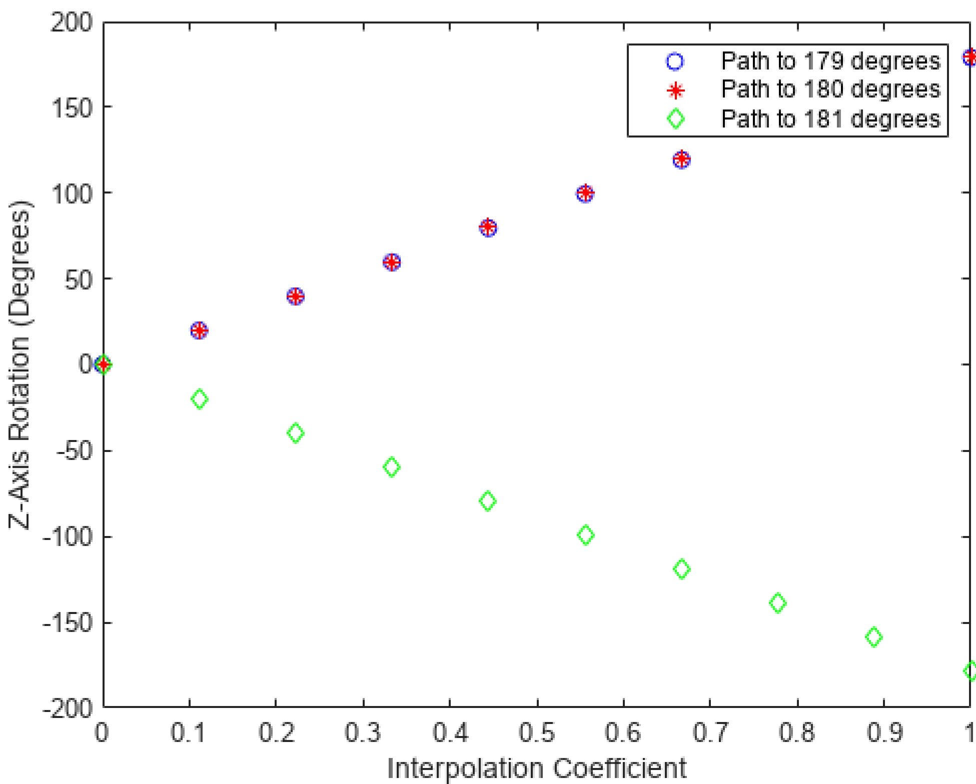
```
q0 = ones(1,'quaternion');

q179 = quaternion([179,0,0],'eulerd','ZYX','frame');

q180 = quaternion([180,0,0],'eulerd','ZYX','frame');

q181 = quaternion([181,0,0],'eulerd','ZYX','frame');
```

Use `slerp` to interpolate between `q0` and the three quaternion rotations. Specify that the paths are traveled in 10 steps.

```
T = linspace(0,1,10);

q179path = slerp(q0,q179,T);
q180path = slerp(q0,q180,T);
q181path = slerp(q0,q181,T);
```

Plot each path in terms of Euler angles in degrees.

```
q179pathEuler = eulerd(q179path,'ZYX','frame');
q180pathEuler = eulerd(q180path,'ZYX','frame');
q181pathEuler = eulerd(q181path,'ZYX','frame');

plot(T,q179pathEuler(:,1),'bo', ...
     T,q180pathEuler(:,1),'r*', ...
     T,q181pathEuler(:,1),'gd');
legend('Path to 179 degrees', ...
       'Path to 180 degrees', ...
       'Path to 181 degrees')
xlabel('Interpolation Coefficient')
ylabel('Z-Axis Rotation (Degrees)')
```



The path between `q0` and `q179` is clockwise to minimize the great circle distance. The path between `q0` and `q181` is counterclockwise to minimize the great circle distance. The path between `q0` and `q180` can be either clockwise or counterclockwise, depending on numerical rounding.

Show Interpolated Quaternions on Sphere

Create two quaternions.

```
q1 = quaternion([75,-20,-10],'eulerd','ZYX','frame');
q2 = quaternion([-45,20,30],'eulerd','ZYX','frame');
```

Define the interpolation coefficient.

```
T = 0:0.01:1;
```

Obtain the interpolated quaternions.

```
quats = slerp(q1,q2,T);
```

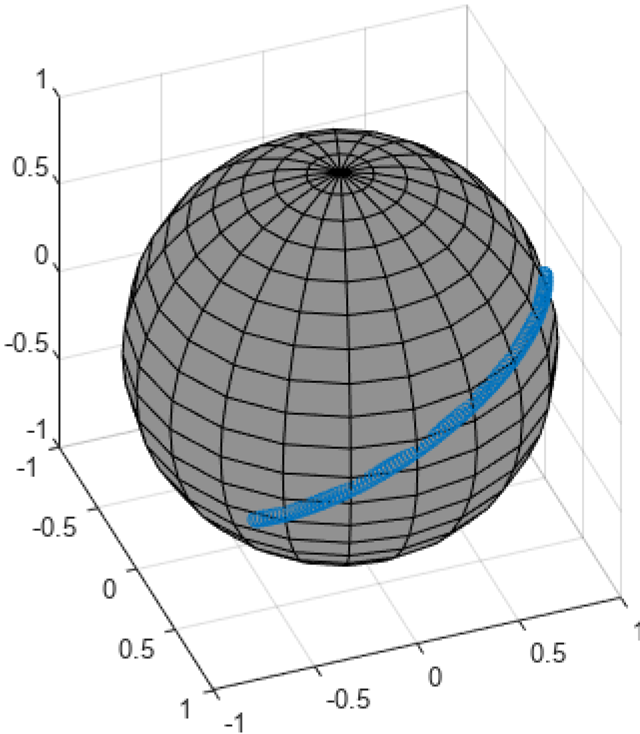
Obtain the corresponding rotate points.

```
pts = rotatepoint(quats,[1 0 0]);
```

Show the interpolated quaternions on a unit sphere.

```
figure
[X,Y,Z] = sphere;
```

```
surf(X,Y,Z,'FaceColor',[0.57 0.57 0.57])
hold on;

scatter3(pts(:,1),pts(:,2),pts(:,3))
view([69.23 36.60])
axis equal
```



Note that the interpolated quaternions follow the shorter path from `q1` to `q2`.

Input Arguments

q1 — Quaternion
scalar | vector | matrix | multidimensional array

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

`q1`, `q2`, and `T` must have compatible sizes. In the simplest cases, they can be the same size or any one can be a scalar. Two inputs have compatible sizes if, for every dimension, the dimension sizes of the inputs are either the same or one of them is 1.

Data Types: `quaternion`

q2 — Quaternion
scalar | vector | matrix | multidimensional array

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

`q1`, `q2`, and `T` must have compatible sizes. In the simplest cases, they can be the same size or any one can be a scalar. Two inputs have compatible sizes if, for every dimension, the dimension sizes of the inputs are either the same or one of the dimension sizes is 1.

Data Types: `quaternion`

`T` — Interpolation coefficient
scalar | vector | matrix | multidimensional array

Interpolation coefficient, specified as a scalar, vector, matrix, or multidimensional array of numbers with each element in the range [0,1].

`q1`, `q2`, and `T` must have compatible sizes. In the simplest cases, they can be the same size or any one can be a scalar. Two inputs have compatible sizes if, for every dimension, the dimension sizes of the inputs are either the same or one of the dimension sizes is 1.

Data Types: `single` | `double`

Output Arguments

`q0` — Interpolated quaternion
scalar | vector | matrix | multidimensional array

Interpolated quaternion, returned as a scalar, vector, matrix, or multidimensional array.

Data Types: `quaternion`

Algorithms

Quaternion **s**pherical **l**inear int**erp**olation (SLERP) is an extension of linear interpolation along a plane to spherical interpolation in three dimensions. The algorithm was first proposed in [1]. Given two quaternions, q_1 and q_2, SLERP interpolates a new quaternion, q_0, along the great circle that connects q_1 and q_2. The interpolation coefficient, T, determines how close the output quaternion is to either q_1 and q_2.

The SLERP algorithm can be described in terms of sinusoids:

$$q_0 = \frac{\sin((1 - T)\theta)}{\sin(\theta)} q_1 + \frac{\sin(T\theta)}{\sin(\theta)} q_2$$

where q_1 and q_2 are normalized quaternions, and θ is half the angular distance between q_1 and q_2.

Version History

Introduced in R2020b

References

[1] Shoemake, Ken. "Animating Rotation with Quaternion Curves." *ACM SIGGRAPH Computer Graphics* Vol. 19, Issue 3, 1985, pp. 245–254.

Extended Capabilities

C/C++ Code Generation

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

See Also

Functions

`dist` | `meanrot`

Objects

`quaternion`

times, .*

Element-wise quaternion multiplication

Syntax

```
quatC = A.*B
```

Description

`quatC = A.*B` returns the element-by-element quaternion multiplication of quaternion arrays.

You can use quaternion multiplication to compose rotation operators:

- To compose a sequence of frame rotations, multiply the quaternions in the same order as the desired sequence of rotations. For example, to apply a *p* quaternion followed by a *q* quaternion, multiply in the order *pq*. The rotation operator becomes $(pq)^{*}v(pq)$, where *v* represents the object to rotate in quaternion form. * represents conjugation.
- To compose a sequence of point rotations, multiply the quaternions in the reverse order of the desired sequence of rotations. For example, to apply a *p* quaternion followed by a *q* quaternion, multiply in the reverse order, *qp*. The rotation operator becomes $(qp)v(qp)^{*}$.

Examples

Multiply Two Quaternion Vectors

Create two vectors, `A` and `B`, and multiply them element by element.

```
A = quaternion([1:4;5:8]);
B = A;
C = A.*B
```

```
C = 2x1 quaternion array
     -28 +   4i +   6j +   8k
    -124 +  60i +  70j +  80k
```

Multiply Two Quaternion Arrays

Create two 3-by-3 arrays, `A` and `B`, and multiply them element by element.

```
A = reshape(quaternion(randn(9,4)),3,3);
B = reshape(quaternion(randn(9,4)),3,3);
C = A.*B
```

```
C = 3x3 quaternion array
     0.60169 +  2.4332i -  2.5844j + 0.51646k    -0.49513 +  1.1722i +  4.4401j -   1.217k      2
     -4.2329 +  2.4547i +  3.7768j + 0.77484k    -0.65232 - 0.43112i -  1.4645j - 0.90073k     -1
```

```
    -4.4159 +  2.1926i +  1.9037j -  4.0303k     -2.0232 +  0.4205i - 0.17288j +  3.8529k     -2
```

Note that quaternion multiplication is not commutative:

```
isequal(C,B.*A)
```

```
ans = logical
   0
```

Multiply Quaternion Row and Column Vectors

Create a row vector a and a column vector b, then multiply them. The 1-by-3 row vector and 4-by-1 column vector combine to produce a 4-by-3 matrix with all combinations of elements multiplied.

```
a = [zeros('quaternion'),ones('quaternion'),quaternion(randn(1,4))]
```

```
a = 1x3 quaternion array
           0 +       0i +       0j +       0k            1 +       0i +       0j +       0k     0
```

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

```
b = 4x1 quaternion array
      0.31877 +   3.5784i +   0.7254j -  0.12414k
      -1.3077 +   2.7694i - 0.063055j +   1.4897k
     -0.43359 -   1.3499i +  0.71474j +    1.409k
      0.34262 +   3.0349i -  0.20497j +   1.4172k
```

```
a.*b
```

```
ans = 4x3 quaternion array
            0 +       0i +        0j +        0k      0.31877 +   3.5784i +   0.7254j -  0.12414
            0 +       0i +        0j +        0k      -1.3077 +   2.7694i - 0.063055j +   1.4897
            0 +       0i +        0j +        0k     -0.43359 -   1.3499i +  0.71474j +    1.409
            0 +       0i +        0j +        0k      0.34262 +   3.0349i -  0.20497j +   1.4172
```

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,

$$q = a_q + b_q\text{i} + c_q\text{j} + d_q\text{k},$$

the product of q and a real scalar β is

$$\beta q = \beta a_q + \beta b_q\text{i} + \beta c_q\text{j} + \beta d_q\text{k}$$

Quaternion Multiplication by a Quaternion Scalar

The definition of the basis elements for quaternions,

$$\text{i}^2 = \text{j}^2 = \text{k}^2 = \text{ijk} = -1\ ,$$

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, $q = a_q + b_q\text{i} + c_q\text{j} + d_q\text{k}$, and $p = a_p + b_p\text{i} + c_p\text{j} + d_p\text{k}$, the multiplication can be expanded as:

$$
\begin{aligned}
z = pq &= \left(a_\mathrm{p} + b_\mathrm{p}\mathrm{i} + c_\mathrm{p}\mathrm{j} + d_\mathrm{p}\mathrm{k}\right)\left(a_\mathrm{q} + b_\mathrm{q}\mathrm{i} + c_\mathrm{q}\mathrm{j} + d_\mathrm{q}\mathrm{k}\right) \\
&= a_\mathrm{p}a_\mathrm{q} + a_\mathrm{p}b_\mathrm{q}\mathrm{i} + a_\mathrm{p}c_\mathrm{q}\mathrm{j} + a_\mathrm{p}d_\mathrm{q}\mathrm{k} \\
&\quad + b_\mathrm{p}a_\mathrm{q}\mathrm{i} + b_\mathrm{p}b_\mathrm{q}\mathrm{i}^2 + b_\mathrm{p}c_\mathrm{q}\mathrm{ij} + b_\mathrm{p}d_\mathrm{q}\mathrm{ik} \\
&\quad + c_\mathrm{p}a_\mathrm{q}\mathrm{j} + c_\mathrm{p}b_\mathrm{q}\mathrm{ji} + c_\mathrm{p}c_\mathrm{q}\mathrm{j}^2 + c_\mathrm{p}d_\mathrm{q}\mathrm{jk} \\
&\quad + d_\mathrm{p}a_\mathrm{q}k + d_\mathrm{p}b_\mathrm{q}\mathrm{ki} + d_\mathrm{p}c_\mathrm{q}\mathrm{kj} + d_\mathrm{p}d_\mathrm{q}\mathrm{k}^2
\end{aligned}
$$

You can simplify the equation using the quaternion multiplication table.

$$
\begin{aligned}
z = pq &= a_\mathrm{p}a_\mathrm{q} + a_\mathrm{p}b_\mathrm{q}\mathrm{i} + a_\mathrm{p}c_\mathrm{q}\mathrm{j} + a_\mathrm{p}d_\mathrm{q}\mathrm{k} \\
&\quad + b_\mathrm{p}a_\mathrm{q}\mathrm{i} - b_\mathrm{p}b_\mathrm{q} + b_\mathrm{p}c_\mathrm{q}\mathrm{k} - b_\mathrm{p}d_\mathrm{q}\mathrm{j} \\
&\quad + c_\mathrm{p}a_\mathrm{q}\mathrm{j} - c_\mathrm{p}b_\mathrm{q}\mathrm{k} - c_\mathrm{p}c_\mathrm{q} + c_\mathrm{p}d_\mathrm{q}\mathrm{i} \\
&\quad + d_\mathrm{p}a_\mathrm{q}k + d_\mathrm{p}b_\mathrm{q}\mathrm{j} - d_\mathrm{p}c_\mathrm{q}\mathrm{i} - d_\mathrm{p}d_\mathrm{q}
\end{aligned}
$$

Version History

Introduced in R2020b

References

[1] Kuipers, Jack B. *Quaternions and Rotation Sequences: A Primer with Applications to Orbits, Aerospace, and Virtual Reality*. Princeton, NJ: Princeton University Press, 2007.

Extended Capabilities

C/C++ Code Generation

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

See Also

Functions

`prod` | `mtimes, *`

Objects

`quaternion`

transpose, .'

Transpose a quaternion array

Syntax

`Y = quat.'`

Description

`Y = quat.'` returns the non-conjugate transpose of the quaternion array, `quat`.

Examples

Vector Transpose

Create a vector of quaternions and compute its nonconjugate transpose.

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

```
quat = 4x1 quaternion array
      0.53767 +  0.31877i +   3.5784j +   0.7254k
       1.8339 -   1.3077i +   2.7694j - 0.063055k
      -2.2588 -  0.43359i -   1.3499j +  0.71474k
      0.86217 +  0.34262i +   3.0349j -  0.20497k
```

```
quatTransposed = quat.'
```

```
quatTransposed = 1x4 quaternion array
      0.53767 +  0.31877i +   3.5784j +   0.7254k       1.8339 -   1.3077i +   2.7694j - 0.06305
```

Matrix Transpose

Create a matrix of quaternions and compute its nonconjugate transpose.

```
quat = [quaternion(randn(2,4)),quaternion(randn(2,4))]
```

```
quat = 2x2 quaternion array
      0.53767 -   2.2588i +  0.31877j -  0.43359k       3.5784 -   1.3499i +   0.7254j +  0.7147
       1.8339 +  0.86217i -   1.3077j +  0.34262k       2.7694 +   3.0349i - 0.063055j -  0.2049
```

```
quatTransposed = quat.'
```

```
quatTransposed = 2x2 quaternion array
      0.53767 -   2.2588i +  0.31877j -  0.43359k       1.8339 +  0.86217i -   1.3077j +  0.3426
       3.5784 -   1.3499i +   0.7254j +  0.71474k       2.7694 +   3.0349i - 0.063055j -  0.2049
```

Input Arguments

`quat` — Quaternion array to transpose
vector | matrix

Quaternion array to transpose, specified as a vector or matrix of quaternions. `transpose` is defined for 1-D and 2-D arrays. For higher-order arrays, use `permute`.

Data Types: `quaternion`

Output Arguments

`Y` — Transposed quaternion array
vector | matrix

Transposed quaternion array, returned as an *N*-by-*M* array, where `quat` was specified as an *M*-by-*N* array.

Version History

Introduced in R2020b

Extended Capabilities

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

See Also

Functions
`ctranspose, '`

Objects
`quaternion`

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 +  0.7147
       1.8339 -   1.3077i +   2.7694j - 0.063055k      0.86217 +  0.34262i +   3.0349j -  0.2049
```

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 -  0.7147
      -1.8339 +   1.3077i -   2.7694j + 0.063055k     -0.86217 -  0.34262i -   3.0349j +  0.2049
```

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`

Version History

Introduced in R2020b

Extended Capabilities

C/C++ Code Generation

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

See Also

Functions

`minus, -`

Objects

`quaternion`

zeros

Create quaternion array with all parts set to zero

Syntax

```
quatZeros = zeros('quaternion')
quatZeros = zeros(n,'quaternion')
quatZeros = zeros(sz,'quaternion')
quatZeros = zeros(sz1,...,szN,'quaternion')

quatZeros = zeros( ___ ,'like',prototype,'quaternion')
```

Description

`quatZeros = zeros('quaternion')` returns a scalar quaternion with all parts set to zero.

`quatZeros = zeros(n,'quaternion')` returns an `n`-by-`n` matrix of quaternions.

`quatZeros = zeros(sz,'quaternion')` returns an array of quaternions where the size vector, `sz`, defines `size(quatZeros)`.

`quatZeros = zeros(sz1,...,szN,'quaternion')` returns a `sz1`-by-...-by-`szN` array of quaternions where `sz1,…,szN` indicates the size of each dimension.

`quatZeros = zeros( ___ ,'like',prototype,'quaternion')` specifies the underlying class of the returned quaternion array to be the same as the underlying class of the quaternion prototype.

Examples

Quaternion Scalar Zero

Create a quaternion scalar zero.

```
quatZeros = zeros('quaternion')
```

```
quatZeros = quaternion
     0 + 0i + 0j + 0k
```

Square Matrix of Quaternions

Create an n-by-n array of quaternion zeros.

```
n = 3;
quatZeros = zeros(n,'quaternion')
```

```
quatZeros = 3x3 quaternion array
     0 + 0i + 0j + 0k     0 + 0i + 0j + 0k     0 + 0i + 0j + 0k
```

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

Multidimensional Array of Quaternion Zeros

Create a multidimensional array of quaternion zeros by defining array dimensions in order. In this example, you create a 3-by-1-by-2 array. You can specify dimensions using a row vector or comma-separated integers.

Specify the dimensions using a row vector and display the results:

```
dims = [3,1,2];
quatZerosSyntax1 = zeros(dims,'quaternion')
```

```
quatZerosSyntax1 = 3x1x2 quaternion array
quatZerosSyntax1(:,:,1) =

     0 + 0i + 0j + 0k
     0 + 0i + 0j + 0k
     0 + 0i + 0j + 0k


quatZerosSyntax1(:,:,2) =

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

Specify the dimensions using comma-separated integers, and then verify the equivalence of the two syntaxes:

```
quatZerosSyntax2 = zeros(3,1,2,'quaternion');
isequal(quatZerosSyntax1,quatZerosSyntax2)
```

```
ans = logical
   1
```

Underlying Class of Quaternion Zeros

A quaternion is a four-part hyper-complex number used in three-dimensional representations. You can specify the underlying data type of the parts as `single` or `double`. The default is `double`.

Create a quaternion array of zeros with the underlying data type set to `single`.

```
quatZeros = zeros(2,'like',single(1),'quaternion')
```

```
quatZeros = 2x2 quaternion array
     0 + 0i + 0j + 0k     0 + 0i + 0j + 0k
     0 + 0i + 0j + 0k     0 + 0i + 0j + 0k
```

Verify the underlying class using the `classUnderlying` function.

```
classUnderlying(quatZeros)
```

```
ans =
'single'
```

Input Arguments

`n` — Size of square quaternion matrix
integer value

Size of square quaternion matrix, specified as an integer value. If `n` is `0` or negative, then `quatZeros` is returned as an empty matrix.

Example: `zeros(4,'quaternion')` returns a 4-by-4 matrix of quaternion zeros.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`sz` — Output size
row vector of integer values

Output size, specified as a row vector of integer values. Each element of `sz` indicates the size of the corresponding dimension in `quatZeros`. If the size of any dimension is `0` or negative, then `quatZeros` is returned as an empty array.

Example: `zeros([1,4,2],'quaternion')` returns a 1-by-4-by-2 array of quaternion zeros.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`prototype` — Quaternion prototype
variable

Quaternion prototype, specified as a variable.

Example: `zeros(2,'like',quat,'quaternion')` returns a 2-by-2 matrix of quaternions with the same underlying class as the prototype quaternion, `quat`.

Data Types: `quaternion`

`sz1,...,szN` — Size of each dimension
two or more integer values

Size of each dimension, specified as two or more integers.

- If the size of any dimension is `0`, then `quatZeros` is returned as an empty array.
- If the size of any dimension is negative, then it is treated as `0`.

Example: `zeros(2,3,'quaternion')` returns a 2-by-3 matrix of quaternion zeros.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Output Arguments

`quatZeros` — Quaternion zeros
scalar | vector | matrix | multidimensional array

Quaternion zeros, returned as a quaternion or array of quaternions.

Given a quaternion of the form $Q = a + b\text{i} + c\text{j} + d\text{k}$, a quaternion zero is defined as $Q = 0 + 0\text{i} + 0\text{j} + 0\text{k}$.

Data Types: `quaternion`

Version History

Introduced in R2020b

Extended Capabilities

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

See Also

Functions
`ones`

Objects
`quaternion`

axang

Convert transformation or rotation into axis-angle rotations

Syntax

```
angles = axang(transformation)
angles = axang(rotation)
```

Description

`angles = axang(transformation)` converts the rotation of the transformation `transformation` to the axis-angle rotations `angles`.

`angles = axang(rotation)` converts the rotation `rotation` to the axis-angle rotations `angles`.

Examples

Convert SE(3) Transformation to Axis-Angle Rotation

Create SE(3) transformation with no translation but with a rotation defined by an axis-angle rotation. Define the axis-rotation with vector of `[0.5 0.25 0.5]` to be the axis and a `pi/2` rotation about that axis.

```
axa1 = [0.5 0.25 0.5 pi/2];
T = se3(axa1,"axang");
```

Plot the axis-angle and the transformation on the same axes.

```
plot3([0 axa1(1)],[0 axa1(2)],[0 axa1(3)],LineWidth=1)
hold on
plotTransforms(T,FrameAxisLabels="on")
```

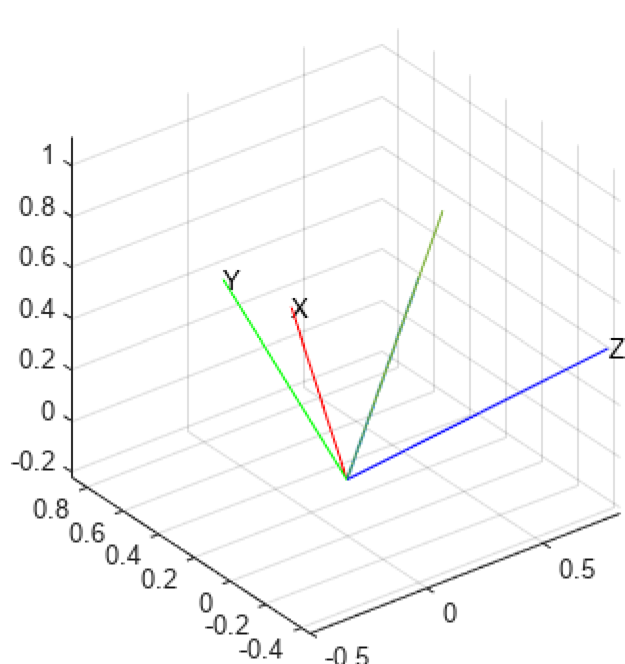
Get the axis-angle rotation from the transformation. Note that the vector of the axis-angle rotation has a different magnitude from the axis-angle rotation specified to the transformation but the defined axis and rotation are the same.

```
axa2 = axang(T)
```

```
axa2 = 1×4

    0.6667    0.3333    0.6667    1.5708
```

Plot the new axis-angle rotation on the same axis.

```
plot3([0 axa2(1)],[0 axa2(2)],[0 axa2(3)])
legend(["Original Rotation Axis","New Rotation Axis"])
hold off
```



Convert SO(3) Rotation to Axis-Angle Rotation

Create SO(3) transformation with a rotation defined by an axis-angle rotation. Define the axis-rotation with vector of `[0.5 0.25 0.5]` to be the axis and a `pi/2` rotation about that axis.

```
axa1 = [0.5 0.25 0.5 pi/2];
R = so3(axa1,"axang");
```

Plot the axis-angle and the transformation on the same axes.

```
plot3([0 axa1(1)],[0 axa1(2)],[0 axa1(3)],LineWidth=1)
hold on
plotTransforms([0 0 0],R,FrameAxisLabels="on")
```

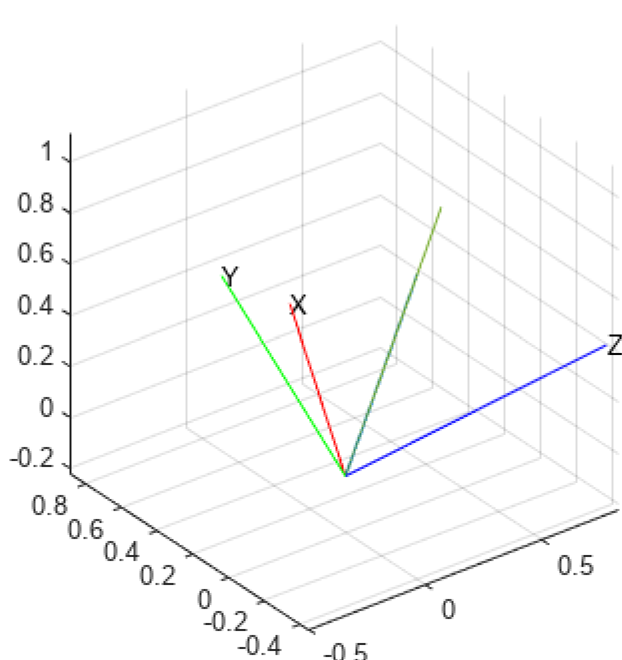
Get the axis-angle rotation from the transformation. Note that the vector of the axis-angle rotation has a different magnitude from the axis-angle rotation specified to the transformation but the defined axis and rotation are the same.

```
axa2 = axang(R)
```

```
axa2 = 1×4

    0.6667    0.3333    0.6667    1.5708
```

Plot the new axis-angle rotation on the same axis.

```
plot3([0 axa2(1)],[0 axa2(2)],[0 axa2(3)])
legend(["Original Rotation Axis","New Rotation Axis"])
hold off
```



Input Arguments

`transformation` — Transformation
`se3` object | *N*-element array of `se3` objects

Transformation, specified as an `se3` object or as an *N*-element array of `se3` objects. *N* is the total number of transformations.

`rotation` — Rotation
`so3` object | *N*-element array of `so3` objects

Rotation, specified as an `so3` object or as an *N*-element array of `so3` objects. *N* is the total number of rotations.

Output Arguments

`angles` — Axis-angle rotation angles
N-by-4 matrix

Axis-angle rotation angles, 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.

Version History

Introduced in R2023a

See Also

`se3` | `so3`

eul

Convert transformation or rotation into Euler angles

Syntax

```
angles = eul(transformation)
angles = eul(rotation)
angles = eul( ___ ,sequence)
```

Description

`angles = eul(transformation)` converts the rotation of the transformation `transformation` to the Euler angles `angles`.

`angles = eul(rotation)` converts the rotation `rotation` to the Euler angles `angles`.

`angles = eul( ___ ,sequence)` specifies the sequence of the Euler-angle rotations `sequence` using any of the input arguments in previous syntaxes. For example, a sequence of `"ZYX"` first rotates the *z*-axis, followed by the *y*-axis and *x*-axis.

Examples

Convert SE(3) Transformation to Euler Angles

Create SE(3) transformation with no translation but with a rotation defined by a Euler angles.

```
eul1 = [pi/4 pi/3 pi/8]
```

```
eul1 = 1×3

    0.7854    1.0472    0.3927
```

```
T = se3(eul1,"eul")
```

```
T = se3
    0.3536   -0.4189    0.8364         0
    0.3536    0.8876    0.2952         0
   -0.8660    0.1913    0.4619         0
         0         0         0    1.0000
```

Get the Euler angles from the transformation.

```
eul2 = eul(T)
```

```
eul2 = 1×3

    0.7854    1.0472    0.3927
```

Convert SO(3) Rotation to Euler Angles

Create SO(3) rotation defined by a Euler angles.

```
eul1 = [pi/4 pi/3 pi/8]
```

```
eul1 = 1×3

    0.7854    1.0472    0.3927
```

```
R = so3(eul1,"eul")
```

```
R = so3
    0.3536   -0.4189    0.8364
    0.3536    0.8876    0.2952
   -0.8660    0.1913    0.4619
```

Get the Euler angles from the transformation.

```
eul2 = eul(R)
```

```
eul2 = 1×3

    0.7854    1.0472    0.3927
```

Input Arguments

`transformation` — Transformation
`se3` object | *N*-element array of `se3` objects

Transformation, specified as an `se3` object or as an *N*-element array of `se3` objects. *N* is the total number of transformations.

`rotation` — Rotation
`so3` object | *N*-element array of `so3` objects

Rotation, specified as an `so3` object or as an *N*-element array of `so3` objects. *N* is the total number of rotations.

`sequence` — Axis-rotation sequence
`"ZYX"` (default) | `"ZYZ"` | `"ZXY"` | `"ZXZ"` | `"YXY"` | `"YZX"` | `"YXZ"` | `"YZY"` | `"XYX"` | `"XYZ"` | `"XZX"` | `"XZY"`

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

- `"ZYX"` (default)
- `"ZYZ"`
- `"ZXY"`
- `"ZXZ"`

- `"YXY"`
- `"YZX"`
- `"YXZ"`
- `"YZY"`
- `"XYX"`
- `"XYZ"`
- `"XZX"`
- `"XZY"`

Each character indicates the corresponding axis. For example, if the sequence is `"ZYX"`, then the three specified Euler angles are interpreted in order as a rotation around the *z*-axis, a rotation around the *y*-axis, and a rotation around the *x*-axis. When applying this rotation to a point, it will apply the axis rotations in the order *x*, then *y*, then *z*.

Data Types: `string` | `char`

Output Arguments

`angles` — Euler angles
M-by-3 matrix

Euler angles, returned as an *M*-by-3 matrix of Euler rotation angles. Each row represents one Euler angle set.

Version History

Introduced in R2023a

See Also

`se3` | `so3`

dist

Calculate distance between transformations

Syntax

```
distance = dist(transformation1,transformation2)
distance = dist(transformation1,transformation2,weights)
distance = dist(rotation1,rotation2)
```

Description

`distance = dist(transformation1,transformation2)` returns the distance `distance` between the poses represented by transformation `transformation1` and transformation `transformation2`.

For the homogeneous transformation objects `se2`, and `se3`, the `dist` function calculates translational and rotational distance independently and combines them in a weighted sum. Translational distance is the Euclidean distance, and rotational distance is the angular difference between the rotation quaternions of `transformation1` and `transformation2`.

`distance = dist(transformation1,transformation2,weights)` specifies the weights `weights` for the translational and rotational distances for calculating the weighted sum of two homogeneous transformations.

`distance = dist(rotation1,rotation2)` returns the distance `distance` between the poses represented by transformation `rotation1` and transformation `rotation2`.

For the homogeneous transformation objects `se2`, and `se3`, the `dist` function calculates translational and rotational distance independently and combines them in a weighted sum. Translational distance is the Euclidean distance, and rotational distance is the angular difference between the rotation quaternions of `rotation1` and `rotation2`.

For rotation objects `so2`, and `so3`, the `dist` function calculates the rotational distance as the angular difference between the rotation quaternions of `rotation1` and `rotation2`.

Input Arguments

`transformation1` — First transformation
`se2` object | `se3` object | *N*-element array of transformation objects

First transformation, specified as a scalar `se2` object, a scalar `se3` object, or as an *N*-element array of transformation objects, where *N* is the total number of transformations. If you specify `transformation1` as an array, each element must be of the same type.

Either `transformation1` or `transformation2` must be a scalar transformation object of the same type. For example, if `transformation1` is an array of `se2` objects, `transformation2` must be a scalar `se2` object.

`transformation2` — Last transformation
`se2` object | `se3` object | *N*-element array of transformation objects

Last transformation, specified as a scalar `se2` object, a scalar `se3` object, or as an *N*-element array of transformation objects, where *N* is the total number of transformations. If you specify `transformation2` as an array, each element must be of the same type.

Either `transformation1` or `transformation2` must be a scalar transformation object of the same type. For example, if `transformation1` is an array of `se2` objects, `transformation2` must be a scalar `se2` object.

`rotation1` — First rotation
`so2` object | `so3` object | *N*-element array of rotation objects

First rotation, specified as a scalar `so2` object, a scalar `so3` object, or as an *N*-element array of rotation objects, where *N* is the total number of rotations. If you specify `rotation1` as an array, each element must be of the same type.

Either `rotation1` or `rotation2` must be a scalar rotation object of the same type. For example, if `rotation1` is an array of `so2` objects, `rotation2` must be a scalar `so2` object.

`rotation2` — Last rotation
`so2` object | `so3` object | *N*-element array of rotation objects

Last rotation, specified as a scalar `so2` object, a scalar `so3` object, or as an *N*-element array of rotation objects, where *N* is the total number of rotations. If you specify `rotation2` as an array, each element must be of the same type.

Either `rotation1` or `rotation2` must be a scalar rotation object of the same type. For example, if `rotation1` is an array of `se2` objects, `rotation2` must be a scalar `se2` object.

`weights` — Weights of translation and rotation in distance sum
`[1.0 0.1]` (default) | two-element row vector

Weights of the translation and rotation in the distance sum, specified as a two-element row vector in the form [*WeightXYZ* *WeightQ*]. *WeightXYZ* is the translational weight and *WeightQ* is the rotational weight. Both weights must be nonnegative numeric values.

Data Types: `single` | `double`

Output Arguments

`distance` — Distance between transformations or rotations
nonnegative numeric scalar

Distance between transformations, returned as a nonnegative numeric scalar. The distance calculate changes depending on the transformation object type of `transformation1` and `transformation2` or `rotation1` and `rotation2`:

- `se2` and `se3` — The `dist` function calculates translational and rotational distance independently and combines them in a weighted sum specified by the `weights` argument. The translational distance is the Euclidean distance between `transformation1` and `transformation2`. The rotational distance is the angular difference between the rotations of `transformation1` and `transformation2`.
- `so2` and `so3` — The `dist` function calculates the rotational distance as the angular difference between the rotations of `rotation1` and `rotation2`.

To calculate the rotational distance, the `dist` function converts the rotation matrix of `transformation1` and `transformation2` or `rotation1` and `rotation2` into `quaternion` objects and uses the `quaternion` `dist` function to calculate the angular distance.

Version History

Introduced in R2022b

See Also

Functions
`normalize` | `interp` | `transform` | `plotTransforms`

Objects
`se2` | `se3` | `so2` | `so3`

interp

Interpolate between transformations

Syntax

```
transformation0 = interp(transformation1,transformation2,points)
rotation0 = interp(rotation1,rotation2,points)
___ = interp( ___ ,transformation2,N)
```

Description

`transformation0 = interp(transformation1,transformation2,points)` interpolates at normalized positions `points` between transformations `transformation1` and `transformation2`.

The function interpolates rotations using a quaternion spherical linear interpolation, and linearly interpolates translations.

`rotation0 = interp(rotation1,rotation2,points)` interpolates at normalized rotations `points` between rotations `rotation1` and `rotation2`.

The function interpolates rotations using a quaternion spherical linear interpolation

`___ = interp( ___ ,transformation2,N)` interpolates `N` steps between the specified transformations or rotations.

Input Arguments

`transformation1` — First transformation
`se2` object | `se3` object | *N*-element array of transformation objects

First transformation, specified as a scalar `se2` object, a scalar `se3` object, or as an *N*-element array of transformation objects, where *N* is the total number of transformations. If you specify `transformation1` as an array, each element must be of the same type.

Either `transformation1` or `transformation2` must be a scalar transformation object of the same type. For example, if `transformation1` is an array of `se2` objects, `transformation2` must be a scalar `se2` object.

`transformation2` — Last transformation
`se2` object | `se3` object | *N*-element array of transformation objects

Last transformation, specified as a scalar `se2` object, a scalar `se3` object, or as an *N*-element array of transformation objects, where *N* is the total number of transformations. If you specify `transformation2` as an array, each element must be of the same type.

Either `transformation1` or `transformation2` must be a scalar transformation object of the same type. For example, if `transformation1` is an array of `se2` objects, `transformation2` must be a scalar `se2` object.

rotation1 — First rotation
so2 object | so3 object | *N*-element array of rotation objects

First rotation, specified as a scalar so2 object, a scalar so3 object, or as an *N*-element array of rotation objects, where *N* is the total number of rotations. If you specify rotation1 as an array, each element must be of the same type.

Either rotation1 or rotation2 must be a scalar rotation object of the same type. For example, if rotation1 is an array of so2 objects, rotation2 must be a scalar so2 object.

rotation2 — Last rotation
so2 object | so3 object | *N*-element array of rotation objects

Last rotation, specified as a scalar so2 object, a scalar so3 object, or as an *N*-element array of rotation objects, where *N* is the total number of rotations. If you specify rotation2 as an array, each element must be of the same type.

Either rotation1 or rotation2 must be a scalar rotation object of the same type. For example, if rotation1 is an array of se2 objects, rotation2 must be a scalar se2 object.

points — Normalized positions
N-element row vector of values in range [0, 1]

Normalized positions, specified as an *N*-element row vector of values in the range [0, 1], where *N* is the total number of interpolated positions. Normalized positions 0 and 1 correspond to the first and last transformations or rotations, respectively.

Example: interp(tf1,tf2,0.5) interpolates a transformation halfway between tf1 and tf2.

Example: interp(r1,r2,0.5) interpolates a rotation halfway between r1 and r2.

N — Number of interpolated positions
positive integer

Number of interpolated positions, specified as a positive integer.

Example: interp(tf1,tf2,5) interpolates five transformations between transformations tf1 and tf2.

Example: interp(r1,r2,7) interpolates seven rotations between rotations r1 and r2.

Output Arguments

transformation0 — Interpolated transformations
N-by-*M* matrix

Interpolated transformations, returned as an *N*-by-*M* matrix of the same transformation type as transformation1 and transformation2, where *N* is the length of the longer argument between transformation1 and transformation2, and *M* is the number of interpolated positions. Each row represents an interpolated transformation between transformation1 and transformation2.

rotation0 — Interpolated rotations
N-by-*M* matrix

Interpolated rotations, returned as an *N*-by-*M* matrix of the same rotation type as rotation1 and rotation2, where *N* is the length of the longer argument between rotation1 and rotation2, and

M is the number of interpolated positions. Each row represents an interpolated transformation between `rotation1` and `rotation2`.

Version History

Introduced in R2022b

Extended Capabilities

C/C++ Code Generation

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

See Also

Functions

`dist` | `normalize` | `transform`

Objects

`se2` | `se3` | `so2` | `so3`

mrdivide, /

Transformation or rotation right division

Syntax

```
transformationC = transformationA/transformationB
rotationC = rotationA/rotationB
```

Description

`transformationC = transformationA/transformationB` right divides transformation `transformationA` by transformation `transformationB` and returns the quotient, transformation `transformationC`. `transformationC` is the same value as `transformationA*inv(transformationB)`.

You can use division to compose a sequence of transformations, so that `transformationC` represents a transformation where the inverse of `transformationB` is applied first, followed by `transformationA`.

`rotationC = rotationA/rotationB` right divides transformation `rotationA` by transformation `rotationB` and returns the quotient, transformation `rotationC`. `rotationC` is the same value as `rotationA*inv(rotationB)`.

Input Arguments

`transformationA` — First transformation
`se2` object | `se3` object | *N*-element array of transformation objects

First transformation, specified as a scalar `se2` object, a scalar `se3` object, or as an *N*-element array of transformation objects, where *N* is the total number of transformations. If you specify `transformationA` as an array, each element must be of the same type.

Either `transformationA` or `transformationB` must be a scalar transformation object of the same type. For example, if `transformationA` is an array of `se2` objects, `transformationB` must be a scalar `se2` object.

`transformationB` — Last transformation
`se2` object | `se3` object | *N*-element array of transformation objects

Last transformation, specified as a scalar `se2` object, a scalar `se3` object, or as an *N*-element array of transformation objects, where *N* is the total number of transformations. If you specify `transformationB` as an array, each element must be of the same type.

Either `transformationA` or `transformationB` must be a scalar transformation object of the same type. For example, if `transformationA` is an array of `se2` objects, `transformationB` must be a scalar `se2` object.

`rotationA` — First rotation
`so2` object | `so3` object | *N*-element array of rotation objects

First rotation, specified as a scalar `so2` object, a scalar `so3` object, or as an *N*-element array of rotation objects, where *N* is the total number of rotations. If you specify `rotationA` as an array, each element must be of the same type.

Either `rotationA` or `rotationB` must be a scalar rotation object of the same type. For example, if `rotationA` is an array of `so2` objects, `rotationB` must be a scalar `so2` object.

`rotationB` — Last rotation
`so2` object | `so3` object | *N*-element array of rotation objects

Last rotation, specified as a scalar `so2` object, a scalar `so3` object, or as an *N*-element array of rotation objects, where *N* is the total number of rotations. If you specify `rotationB` as an array, each element must be of the same type.

Either `rotationA` or `rotationB` must be a scalar rotation object of the same type. For example, if `rotationA` is an array of `se2` objects, `rotationB` must be a scalar `se2` object.

Output Arguments

`transformationC` — Transformation quotient
`se2` object | `se3` object | *N*-element array of transformation objects

Transformation quotient, returned as a scalar `se2` object, a scalar `se3` object, or as an *N*-element array of the same transformation type as `transformationA` and `transformationB`. *N* is the length of the longer argument between `transformationA` and `transformationB` and each row represents the quotient between `transformationA` and `transformationB`.

`rotationC` — Rotation quotient
`so2` object | `so3` object | *N*-element array of rotation objects

Rotation quotient, returned as a scalar `so2` object, a scalar `so3` object, or as an *N*-element array of the same rotation type as `rotationA` and `rotationB`. *N* is the length of the longer argument between `rotationA` and `rotationB` and each row represents the quotient between `rotationA` and `rotationB`.

Version History

Introduced in R2022b

See Also

Functions
`rdivide, ./` | `mtimes, *` | `times, .*`

Objects
`se2` | `se3` | `so2` | `so3`

mtimes, *

Transformation or rotation multiplication

Syntax

```
transformationC = transformationA*transformationB
rotationC = rotationA*rotationB
```

Description

`transformationC = transformationA*transformationB` performs transformation multiplication between transformation `transformationA` and transformation `transformationB` and returns the product, transformation `transformationC`.

You can use transformation multiplication to compose a sequence of transformations, so that `transformationC` represents a transformation where `transformationB` is applied first, followed by `transformationA`.

`rotationC = rotationA*rotationB` performs rotation multiplication between rotation `rotationA` and rotation `rotationB` and returns the product, rotation `rotationC`.

You can use rotation multiplication to compose a sequence of rotations, so that `rotationC` represents a rotation where `rotationB` is applied first, followed by `rotationA`.

Input Arguments

`transformationA` — First transformation

`se2` object | `se3` object | *N*-element array of transformation objects

First transformation, specified as a scalar `se2` object, a scalar `se3` object, or as an *N*-element array of transformation objects, where *N* is the total number of transformations. If you specify `transformationA` as an array, each element must be of the same type.

Either `transformationA` or `transformationB` must be a scalar transformation object of the same type. For example, if `transformationA` is an array of `se2` objects, `transformationB` must be a scalar `se2` object.

`transformationB` — Last transformation

`se2` object | `se3` object | *N*-element array of transformation objects

Last transformation, specified as a scalar `se2` object, a scalar `se3` object, or as an *N*-element array of transformation objects, where *N* is the total number of transformations. If you specify `transformationB` as an array, each element must be of the same type.

Either `transformationA` or `transformationB` must be a scalar transformation object of the same type. For example, if `transformationA` is an array of `se2` objects, `transformationB` must be a scalar `se2` object.

`rotationA` — First rotation

`so2` object | `so3` object | *N*-element array of rotation objects

First rotation, specified as a scalar so2 object, a scalar so3 object, or as an *N*-element array of rotation objects, where *N* is the total number of rotations. If you specify rotationA as an array, each element must be of the same type.

Either rotationA or rotationB must be a scalar rotation object of the same type. For example, if rotationA is an array of so2 objects, rotationB must be a scalar so2 object.

rotationB — Last rotation
so2 object | so3 object | *N*-element array of rotation objects

Last rotation, specified as a scalar so2 object, a scalar so3 object, or as an *N*-element array of rotation objects, where *N* is the total number of rotations. If you specify rotationB as an array, each element must be of the same type.

Either rotationA or rotationB must be a scalar rotation object of the same type. For example, if rotationA is an array of se2 objects, rotationB must be a scalar se2 object.

Output Arguments

transformationC — Transformation product
se2 object | se3 object | *N*-element array of transformation objects

Transformation product, returned as a scalar se2 object, a scalar se3 object, or as an *N*-element array of the same transformation type as transformationA and transformationB. *N* is the length of the longer argument between transformationA and transformationB and each row represents the product between transformationA and transformationB.

rotationC — Rotation product
so2 object | so3 object | *N*-element array of rotation objects

Rotation product, returned as a scalar so2 object, a scalar so3 object, or as an *N*-element array of the same rotation type as rotationA and rotationB. *N* is the length of the longer argument between rotationA and rotationB and each row represents the product between rotationA and rotationB.

Version History

Introduced in R2022b

See Also

Functions
mrdivide, / | rdivide, ./ | times, .*

Objects
se2 | se3 | so2 | so3

normalize

Normalize transformation or rotation matrix

Syntax

```
transformationN = normalize(transformation)
rotationN = normalize(rotation)
___ = normalize(___,Method=normMethod)
```

Description

`transformationN = normalize(transformation)` normalizes the rotation of the transformation `transformation` and returns a transformation, `transformationN`, that is equivalent to `transformation`, but with normalized rotation.

`rotationN = normalize(rotation)` normalizes the rotation of the rotation `rotation` and returns a rotation, `rotationN`, that is equivalent to `rotation`, but with normalized rotation.

Note The transformation and rotation objects do not automatically normalize their rotations. You must use `normalize` each time you need to normalize a transformation or rotation. You may need to do this if:

- You specified an unnormalized input transformation or rotation at the creation of the transformation or rotation object.
- You performed many operations on the transformation or rotation objects such as `mtimes, *`, which may cause the transformation or rotation to become unnormalized due to data type precision.

`___ = normalize(___,Method=normMethod)` specifies the normalization method `normMethod` that the `normalize` function uses to normalize the specified transformation or rotation.

Input Arguments

`transformation` — Transformation
`se2` object | `se3` object | *N*-element array of transformation objects

Transformation, specified as a scalar `se2` object, a scalar `se3` object, or an *N*-element array of transformation objects. *N* is the total number of transformations.

If you specify `transformation` as an array, each element must be of the same type.

`rotation` — Rotation
`so2` object | `so3` object | *N*-element array of rotation objects

Rotation, specified as a scalar `so2` object, a scalar `so3` object, or as an *N*-element array of rotation objects. *N* is the total number of rotations.

If you specify `rotation` as an array, each element must be of the same type.

`normMethod` — Normalization method
`"quat"` (default) | `"cross"` | `"svd"`

Normalization method, specified as one of these options:

- `"quat"` — Convert the rotation submatrix into a normalized quaternion and then convert the normalized quaternion back to a transformation or rotation object. For more information, see the `normalize` of the `quaternion` object.
- `"cross"` — Normalize the third column of the rotation submatrix and then determine the other two columns through cross products.
- `"svd"` — Use singular value decomposition to find the closest orthonormal matrix by setting singular values to `1`. This solves the orthogonal Procrustes problem.

Data Types: `char` | `string`

Output Arguments

`transformationN` — Normalized transformation
`se2` object | `se3` object

Normalized transformation, returned as an `se2` or `se3` object.

`rotationN` — Normalized rotation
`so2` object | `so3` object

Normalized rotation, returned as an `so2` or `so3` object.

Version History

Introduced in R2022b

Extended Capabilities

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

See Also

Functions
`dist` | `interp` | `transform` | `plotTransforms`

Objects
`se2` | `se3` | `so2` | `so3` | `quaternion`

plot

Draw transformation coordinate frame

Syntax

```
plot(T)
plot( ___ ,Name=Value)
AX = plot( ___ ,Name=Value)
```

Description

`plot(T)` draws a 3-D coordinate frame of transformation `T` with labeled axes. The *x*-axis is colored in red, the *y*-axis in green, and the *z*-axis in blue.

`plot( ___ ,Name=Value)` specifies optional arguments using one or more name-value arguments. For example, `plot(T,AxisLabels="off")` hides the *xyz* labels.

`AX = plot( ___ ,Name=Value)` returns the axis object, `AX`, containing the transformation plots.

Input Arguments

T — Transformation

`SE3` object | `SO3` object | *M*-element array of `SE3` or `SO3` objects

Transformation, specified as either an individual `SE3` or `SO3` object, or as an *M*-element array of transformation objects. *M* is the total number of transformations. Ever transformation in `T` is plotted if `T` is an *M*-element array.

Data Types: `single` | `double`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `plot(T,AxisLabels="off")`

`AxisLabels` — Show axis labels

`"on"` (default) | `"off"`

Show axis labels, specified as `"off"` or `"on"`.

Example: `plot(T,AxisLabels="off")`

Data Types: `char` | `string`

`FrameLabel` — Name of coordinate frame

`""` (default) | string scalar | character vector

Name of the coordinate frame, specified as a string scalar or character vector.

Example: `plot(T,FrameLabel="TF1")`

Data Types: `char` | `string`

`Color` — Use uniform color for coordinate frame
`"off"` (default) | `"on"`

Use uniform color for coordinate frame, specified as `"off"` or `"on"`.

Example: `plot(T,Color="on")`

Data Types: `char` | `string`

Output Arguments

`AX` — Axes handle
`Axes` object

Axes handle, specified as an `Axes` object.

Version History

Introduced in R2022b

Extended Capabilities

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

See Also

Functions
`interpolate` | `normalize` | `rotm` | `showdetails` | `tform` | `transformPoints` | `trvec`

Objects
SE3 | SO3

quat

Convert transformation or rotation to numeric quaternion

Syntax

```
q = quat(transformation)
q = quat(rotation)
```

Description

`q = quat(transformation)` creates a quaternion `q` from the rotation of the transformation `transformation`.

`q = quat(rotation)` creates a quaternion `q` from the rotation `rotation`.

Examples

Convert SE(3) Transformation to Numeric Quaternion

Create SE(3) transformation with zero translation and a rotation defined by a numeric quaternion. Use the `eul2quat` function to create the numeric quaternion.

```
quat1 = eul2quat([0 0 deg2rad(30)])
```

```
quat1 = 1×4

    0.9659    0.2588         0         0
```

```
T = se3(quat1,"quat")
```

```
T = se3
    1.0000         0         0         0
         0    0.8660   -0.5000         0
         0    0.5000    0.8660         0
         0         0         0    1.0000
```

Convert the transformation back into a numeric quaternion.

```
quat2 = quat(T)
```

```
quat2 = 1×4

    0.9659    0.2588         0         0
```

Convert SO(3) Rotation to Numeric Quaternion

Create SO(3) rotation defined by a numeric quaternion. Use the `eul2quat` function to create the numeric quaternion.

```
quat1 = eul2quat([0 0 deg2rad(30)])
```

```
quat1 = 1×4

    0.9659    0.2588         0         0
```

```
R = so3(quat1,"quat")
```

```
R = so3
    1.0000         0         0
         0    0.8660   -0.5000
         0    0.5000    0.8660
```

Convert the rotation into a numeric quaternion.

```
quat2 = quat(R)
```

```
quat2 = 1×4

    0.9659    0.2588         0         0
```

Input Arguments

`transformation` — Transformation
`se3` object | *N*-element array of `se3` objects

Transformation, specified as an `se3` object or as an *N*-element array of `se3` objects. *N* is the total number of transformations.

`rotation` — Rotation
`so3` object | *N*-element array of `so3` objects

Rotation, specified as an `so3` object or as an *N*-element array of `so3` objects. *N* is the total number of rotations.

Output Arguments

q — Quaternion rotation angles
M-by-4 matrix

Quaternion rotation angles, returned as an *M*-by-4 matrix, where each row is of the form [*qw qx qy qz*]. *M* is the total number of transformations or rotations specified.

Version History

Introduced in R2023a

See Also
`se3` | `so3`

rdivide, ./

Element-wise transformation or rotation right division

Syntax

```
transformationC = transformationA./transformationB
rotationC = rotationA./rotationB
```

Description

`transformationC = transformationA./transformationB` divides transformations element-by-element by dividing each element of transformation `transformationA` with the corresponding element of transformation `transformationB` and returns the quotient, transformation `transformationC`.

`rotationC = rotationA./rotationB` divides rotations element-by-element by dividing each element of rotation `rotationA` with the corresponding element of rotation `rotationB` and returns the quotient, rotation `rotationC`.

Input Arguments

`transformationA` — First transformation
`se2` object | `se3` object | *N*-element array of transformation objects

First transformation, specified as a scalar `se2` object, a scalar `se3` object, or as an *N*-element array of transformation objects, where *N* is the total number of transformations. If you specify `transformationA` as an array, each element must be of the same type.

Either `transformationA` or `transformationB` must be a scalar transformation object of the same type. For example, if `transformationA` is an array of `se2` objects, `transformationB` must be a scalar `se2` object.

`transformationB` — Last transformation
`se2` object | `se3` object | *N*-element array of transformation objects

Last transformation, specified as a scalar `se2` object, a scalar `se3` object, or as an *N*-element array of transformation objects, where *N* is the total number of transformations. If you specify `transformationB` as an array, each element must be of the same type.

Either `transformationA` or `transformationB` must be a scalar transformation object of the same type. For example, if `transformationA` is an array of `se2` objects, `transformationB` must be a scalar `se2` object.

`rotationA` — First rotation
`so2` object | `so3` object | *N*-element array of rotation objects

First rotation, specified as a scalar `so2` object, a scalar `so3` object, or as an *N*-element array of rotation objects, where *N* is the total number of rotations. If you specify `rotationA` as an array, each element must be of the same type.

Either `rotationA` or `rotationB` must be a scalar rotation object of the same type. For example, if `rotationA` is an array of `so2` objects, `rotationB` must be a scalar `so2` object.

`rotationB` — Last rotation
`so2` object | `so3` object | *N*-element array of rotation objects

Last rotation, specified as a scalar `so2` object, a scalar `so3` object, or as an *N*-element array of rotation objects, where *N* is the total number of rotations. If you specify `rotationB` as an array, each element must be of the same type.

Either `rotationA` or `rotationB` must be a scalar rotation object of the same type. For example, if `rotationA` is an array of `se2` objects, `rotationB` must be a scalar `se2` object.

Output Arguments

`transformationC` — Transformation quotient
`se2` object | `se3` object | *N*-element array of transformation objects

Transformation quotient, returned as a scalar `se2` object, a scalar `se3` object, or as an *N*-element array of the same transformation type as `transformationA` and `transformationB`. *N* is the length of the longer argument between `transformationA` and `transformationB` and each row represents the quotient between `transformationA` and `transformationB`.

`rotationC` — Rotation quotient
`so2` object | `so3` object | *N*-element array of rotation objects

Rotation quotient, returned as a scalar `so2` object, a scalar `so3` object, or as an *N*-element array of the same rotation type as `rotationA` and `rotationB`. *N* is the length of the longer argument between `rotationA` and `rotationB` and each row represents the quotient between `rotationA` and `rotationB`.

Version History

Introduced in R2022b

See Also

Functions
`mrdivide, /` | `mtimes, *` | `times, .*`

Objects
`se2` | `se3` | `so2` | `so3`

rotm

Extract rotation matrix

Syntax

```
rotationMatrix = rotm(transformation)
rotationMatrix = rotm(rotation)
```

Description

`rotationMatrix = rotm(transformation)` returns the rotation matrix `rotationMatrix` from the SE(2) or SE(3) transformation `transformation`.

`rotationMatrix = rotm(rotation)` returns the rotation matrix `rotationMatrix` from the SO(2) or SO(3) rotation `rotation`.

Input Arguments

`transformation` — Transformation
`se2` object | `se3` object | *N*-element array of transformation objects

Transformation, specified as a scalar `se2` object, a scalar `se3` object, or an *N*-element array of transformation objects. *N* is the total number of transformations.

If you specify `transformation` as an array, each element must be of the same type.

`rotation` — Rotation
`so2` object | `so3` object | *N*-element array of rotation objects

Rotation, specified as a scalar `so2` object, a scalar `so3` object, or an *N*-element array of rotation objects. *N* is the total number of rotations.

If you specify `rotation` as an array, each element must be of the same type.

Output Arguments

`rotationMatrix` — Rotation matrix
2-by-2-by-*N* array | 3-by-3-by-*N* array

Rotation matrix, returned as a 2-by-2-by-*N* array for 2-D transformations or a 3-by-3-by-*N* array for 3-D transformations. *N* is the total number of transformations.

Version History

Introduced in R2022b

Extended Capabilities

C/C++ Code Generation

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

See Also

Functions

`dist` | `interp` | `normalize` | `tform` | `transform` | `trvec` | `plotTransforms`

Objects

`se2` | `se3` | `so2` | `so3`

showdetails

Display transformation in compact form

Syntax

```
showdetails(transformation)
showdetails(rotation)
showdetails( ___ Name=Value)
```

Description

`showdetails(transformation)` displays the translational and rotational components of the transformation `transformation` on a single line. The rotation units are in degrees.

`showdetails(rotation)` displays the rotational components of the rotation `rotation` on a single line.

`showdetails( ___ Name=Value)` specifies additional options using one or more name-value arguments.

Input Arguments

`transformation` — Transformation
`se2` object | `se3` object | *N*-element array of transformation objects

Transformation, specified as a scalar `se2` object, a scalar `se3`, or an *N*-element array of transformation objects. *N* is the total number of transforms.

If you specify `transformation` as an array, each element must be of the same type. Additionally, `showdetails` prints the details on a new line for each of the *N* transformations.

`rotation` — Rotation
`so2` object | `so3` object | *N*-element array of rotation objects

Rotation, specified as a scalar `so2` object, a scalar `so3` object, or as an *N*-element array of rotation objects. *N* is the total number of rotations.

If you specify `rotation` as an array, each element must be of the same type. Additionally, `showdetails` prints the details on a new line for each of the *N* rotations.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `showdetails(T,Sequence="ZYX")`

Sequence — Euler angle sequence order
"ZYX" (default) | "ZYZ" | "ZXY" | "ZXZ" | "YXY" | "YZX" | "YXZ" | "YZY" | "XYX" | "XYZ" | "XZX" | "XZY"

Euler angle sequence order, specified as one of these string scalars:

- "ZYX" (default)
- "ZYZ"
- "ZXY"
- "ZXZ"
- "YXY"
- "YZX"
- "YXZ"
- "YZY"
- "XYX"
- "XYZ"
- "XZX"
- "XZY"

Each character indicates the corresponding axis. For example if the sequence is "ZYX", then the printed order of rotation angles is *z*-axis, *y*-axis, and then the *x*-axis.

This parameter does not affect the output when `transformation` contains an `se2` object or if `rotation` contains an `so2` object.

Example: `showdetails(T,Sequence="ZYX")`

Data Types: `char` | `string`

`AngleUnit` — Angle unit
"deg" (default) | "rad"

Angle unit, specified as "deg" for degrees, or "rad" for radians.

Example: `showdetails(T,AngleUnit="rad")`

Data Types: `char` | `string`

Version History

Introduced in R2022b

Extended Capabilities

C/C++ Code Generation

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

See Also

Functions

`dist` | `interp` | `normalize` | `transform`

Objects

se2 | se3 | so2 | so3

theta

Convert transformation or rotation to 2-D rotation angle

Syntax

```
angle = theta(transformation)
angle = theta(rotation)
```

Description

`angle = theta(transformation)` extracts the 2-D rotation angle `angle` from the transformation `transformation`.

`angle = theta(rotation)` extracts the 2-D rotation angle `angle` from the rotation `rotation`.

Examples

Convert SE(2) Transformation to Angle

Create SE(2) transformation with a rotation defined by an angle `pi/2`.

```
angle1 = pi/2
```

```
angle1 = 1.5708
```

```
T = se2(angle1,"theta")
```

```
T = se2
    0.0000   -1.0000         0
    1.0000    0.0000         0
         0         0    1.0000
```

Get the rotation angle from the transformation.

```
angle2 = theta(T)
```

```
angle2 = 1.5708
```

Convert SO(2) Transformation to Angle

Create SO(2) rotation defined by an angle `pi/2`.

```
angle1 = pi/2
```

```
angle1 = 1.5708
```

```
R = so2(angle1,"theta")
```

```
R = so2
    0.0000   -1.0000
    1.0000    0.0000
```

Get the rotation angle from the rotation.

```
angle2 = theta(R)
```

```
angle2 = 1.5708
```

Input Arguments

`transformation` — Transformation
`se2` object | *N*-by-*M* array of `se2` objects

Transformation, specified as an `se2` object or as an *N*-by-*M* array of `se2` objects. *N* is the total number of transformations.

If `transformation` is a *N*-by-*M* array, the `angle` argument is the same size and contains an angle for each of the `se2` objects specified in the array.

Data Types: `single` | `double`

`rotation` — Rotation
`so2` object | *N*-by-*M* array of `so2` objects

Rotation, specified as an `so2` object or as an *N*-by-*M* array of `so2` objects. *N* is the total number of rotations.

If `rotation` is a *N*-by-*M* array, the `angle` argument is the same size and contains an angle for each of the `so2` objects specified in the array.

Output Arguments

`angle` — Rotation angle
numeric scalar | *N*-by-*M* matrix

Rotation angle, returned as a numeric scalar for a scalar input and as an *N*-by-*M* matrix for an array input. *N* and *M* are the dimensions of the input `rotation` or `transformation` argument. Each element of the matrix is an angle, in radians, and each angle corresponds to a rotation or transformation in the input at the same index location.

The rotation angle is counterclockwise positive when you look along the specified axis toward the origin.

Data Types: `single` | `double`

Version History

Introduced in R2023a

See Also

se2 | so2

tform

Extract homogeneous transformation

Syntax

```
transformationMatrix = tform(transformation)
transformationMatrix = tform(rotation)
```

Description

`transformationMatrix = tform(transformation)` extracts the homogeneous transformation matrix `transformationMatrix` that corresponds to the SE(2) or SE(3) transformation `transformation`.

`transformationMatrix = tform(rotation)` creates a homogeneous transformation matrix `transformationMatrix`, with zero translation, that corresponds to the SO(2) or SO(3) rotation `rotation`.

Input Arguments

`transformation` — Transformation

`se2` object | `se3` object | *N*-element array of transformation objects

Transformation, specified as a scalar `se2` object, a scalar `se3` object, or an *N*-element array of transformation objects. *N* is the total number of transformations.

If you specify `transformation` as an array, each element must be of the same type.

`rotation` — Rotation

`so2` object | `so3` object | *N*-element array of rotation objects

Rotation, specified as a scalar `so2` object, a scalar `so3` object, or an *N*-element array of rotation objects. *N* is the total number of rotations.

If you specify `rotation` as an array, each element must be of the same type.

Output Arguments

`transformationMatrix` — Homogeneous transformation matrix

3-by-3-by-*N* array | 4-by-4-by-*N* array

Homogeneous transformation matrix, returned as a 3-by-3-by-*N* array for `se2` and `so2` objects, or a 4-by-4-by-*N* array for `se3` and `so3` objects. *N* is the total number of transformations.

Version History

Introduced in R2022b

Extended Capabilities

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

See Also

Functions
rotm | trvec

Objects
se2 | se3 | so2 | so3

times, .*

Element-wise transformation or rotation multiplication

Syntax

```
transformationC = transformationA.*transformationB
rotationC = rotationA.*rotationB
```

Description

`transformationC = transformationA.*transformationB` multiplies transformations element-by-element by multiplying each element of transformation `transformationA` with the corresponding element of transformation `transformationB` and returns the product, transformation `transformationC`.

`rotationC = rotationA.*rotationB` multiplies rotations element-by-element by multiplying each element of rotation `rotationA` with the corresponding element of rotation `rotationB` and returns the product, rotation `rotationC`.

Input Arguments

`transformationA` — First transformation
`se2` object | `se3` object | *N*-element array of transformation objects

First transformation, specified as a scalar `se2` object, a scalar `se3` object, or as an *N*-element array of transformation objects, where *N* is the total number of transformations. If you specify `transformationA` as an array, each element must be of the same type.

Either `transformationA` or `transformationB` must be a scalar transformation object of the same type. For example, if `transformationA` is an array of `se2` objects, `transformationB` must be a scalar `se2` object.

`transformationB` — Last transformation
`se2` object | `se3` object | *N*-element array of transformation objects

Last transformation, specified as a scalar `se2` object, a scalar `se3` object, or as an *N*-element array of transformation objects, where *N* is the total number of transformations. If you specify `transformationB` as an array, each element must be of the same type.

Either `transformationA` or `transformationB` must be a scalar transformation object of the same type. For example, if `transformationA` is an array of `se2` objects, `transformationB` must be a scalar `se2` object.

`rotationA` — First rotation
`so2` object | `so3` object | *N*-element array of rotation objects

First rotation, specified as a scalar `so2` object, a scalar `so3` object, or as an *N*-element array of rotation objects, where *N* is the total number of rotations. If you specify `rotationA` as an array, each element must be of the same type.

Either `rotationA` or `rotationB` must be a scalar rotation object of the same type. For example, if `rotationA` is an array of `so2` objects, `rotationB` must be a scalar `so2` object.

`rotationB` — Last rotation
`so2` object | `so3` object | *N*-element array of rotation objects

Last rotation, specified as a scalar `so2` object, a scalar `so3` object, or as an *N*-element array of rotation objects, where *N* is the total number of rotations. If you specify `rotationB` as an array, each element must be of the same type.

Either `rotationA` or `rotationB` must be a scalar rotation object of the same type. For example, if `rotationA` is an array of `se2` objects, `rotationB` must be a scalar `se2` object.

Output Arguments

`transformationC` — Transformation product
`se2` object | `se3` object | *N*-element array of transformation objects

Transformation product, returned as a scalar `se2` object, a scalar `se3` object, or as an *N*-element array of the same transformation type as `transformationA` and `transformationB`. *N* is the length of the longer argument between `transformationA` and `transformationB` and each row represents the product between `transformationA` and `transformationB`.

`rotationC` — Rotation product
`so2` object | `so3` object | *N*-element array of rotation objects

Rotation product, returned as a scalar `so2` object, a scalar `so3` object, or as an *N*-element array of the same rotation type as `rotationA` and `rotationB`. *N* is the length of the longer argument between `rotationA` and `rotationB` and each row represents the product between `rotationA` and `rotationB`.

Version History

Introduced in R2022b

See Also

Functions
`mrdivide, /` | `rdivide, ./` | `mtimes, *`

Objects
`se2` | `se3` | `so2` | `so3`

transform

Apply rigid body transformation to points

Syntax

```
tpoints = transform(transformation,points)
tpoints = transform(rotation,points)
tpoints = transform( ___ ,isCol=format)
```

Description

`tpoints = transform(transformation,points)` applies the rigid body transformation `transformation` to the input points `points`, and returns the transformed points `tpoints`.

`tpoints = transform(rotation,points)` applies the rotation `rotation` to the input points `points`, and returns the transformed points `tpoints`.

`tpoints = transform( ___ ,isCol=format)` sets the expected format of the input points `points` to be either column-wise or row-wise by using the logical flag `format` in addition to the input arguments from the previous syntax.

Input Arguments

`transformation` — Transformation
`se2` object | `se3` object | *N*-element array of transformation objects

Transformation, specified as a scalar `se2` object, a scalar `se3` object, or an *N*-element array of transformation objects. *N* is the total number of transformations.

If you specify `transformation` as an array, each element must be of the same type.

`rotation` — Rotation
`so2` object | `so3` object | *N*-element array of rotation objects

Rotation, specified as a scalar `so2` object, a scalar `so3` object, or as an *N*-element array of rotation objects. *N* is the total number of rotations.

If you specify `rotation` as an array, each element must be of the same type.

`points` — Points to transform
N-by-*D*-by-*M* array | *D*-by-*N*-by-*M* array

Points to transform, specified as an *N*-by-*D*-by-*M* array, where:

- *D* is the dimension of the transformation, defined as 2 for 2-D transformations and 3 for 3-D transformations.
- *N* is the total number of input points to transform.
- *M* is the total number of transforms to perform on each point.

For 2-D transformations and rotations, each row specifies a point in the form [*X Y*]. For 3-D transformations and rotations, each row specifies a point in the form [*X Y Z*].

If you specify `format` as `true`, then you must specify `points` as a *D*-by-*N*-by-*M* array, where each column specifies a point.

Data Types: `single` | `double`

`format` — Point format
`false` or `0` (default) | `true` or `1`

Point format, specified as a logical `0` (`false`) or `1` (`true`). If you specify this argument as true, you must specify the points in points as columns. Otherwise, specify points as rows.

Example: `isCol=true`

Data Types: `logical`

Output Arguments

`tpoints` — Transformed points
N-by-*D*-by-*M* array | *D*-by-*N*-by-*M* array

Transformed points, returned as an *N*-by-*D*-by-*M* array, where:

- *D* is the dimension of the transformation, defined as 2 for 2-D transformations and rotations and 3 for 3-D transformations or rotations.
- *N* is the total number of input points to transform.
- *M* is the total number of transforms to perform on each point.

For 2-D transformations and rotations, each row specifies a point in the form [*X Y*]. For 3-D transformations and rotations, each row specifies a point in the form [*X Y Z*].

If you specify `format` as `true`, `tpoints` is returned as a *D*-by-*N*-by-*M* array, where each column specifies a point.

Version History

Introduced in R2022b

Extended Capabilities

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

See Also

Functions
`dist` | `interp` | `normalize` | `rotm` | `tform` | `trvec` | `plotTransforms`

Objects
`se2` | `se3` | `so2` | `so3`

trvec

Extract translation vector

Syntax

```
translationVector = trvec(transformation)
```

Description

`translationVector = trvec(transformation)` extracts the translation vector `translationVector` of the SE(2) or SE(3) transformation `transformation`.

Input Arguments

`transformation` — Transformation
`se2` object | `se3` object | *N*-element array of transformation objects

Transformation, specified as a scalar `se2` object, a scalar `se3` object, or an *N*-element array of transformation objects. *N* is the total number of transformations.

If you specify `transformation` as an array, each element must be of the same type.

Output Arguments

`translationVector` — Translation vector
N-by-2 matrix | *N*-by-3 matrix

Translation vector, returned as an *N*-by-2 matrix for `se2` objects or an *N*-by-3 matrix for `se3` objects. *N* is the total number of transformations or rotations, and each row is a translation vector in the form [*X Y*] for 2-D transformations or [*X Y Z*] for 3-D transformations.

Version History

Introduced in R2022b

Extended Capabilities

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

See Also

Functions
`dist` | `interp` | `normalize` | `rotm` | `tform` | `transform` | `plotTransforms`

Objects
`se2` | `se3`

xytheta

Convert transformation or rotation to compact 2-D pose representation

Syntax

```
pose = xytheta(transformation)
pose = xytheta(rotation)
```

Description

`pose = xytheta(transformation)` converts a transformation `transformation` to a compact 2-D pose representation `pose`.

`pose = xytheta(rotation)` converts a rotation `rotation` to a compact 2-D pose representation `pose` with no translation.

Examples

Convert SE(2) Transformation to 2-D Compact Pose

Create SE(2) transformation with an *xy*-position of `[2 3]` and a rotation defined by an angle `pi/2`.

```
pose1 = [2 3 pi/2];
T = se2(pose1,"xytheta")
```

```
T = se2
    0.0000   -1.0000    2.0000
    1.0000    0.0000    3.0000
         0         0    1.0000
```

Convert the transformation back into a compact pose.

```
pose2 = xytheta(T)
```

```
pose2 = 1×3

    2.0000    3.0000    1.5708
```

Convert SO(2) Rotation to 2-D Compact Pose

Create SO(2) rotation defined by an angle `pi/2`.

```
angle = pi/2
```

```
angle = 1.5708
```

```
R = so2(angle,"theta")
```

```
R = so2
    0.0000   -1.0000
    1.0000    0.0000
```

Convert the transformation back into a compact pose.

```
pose = xytheta(R)
```

```
pose = 1×3

         0         0    1.5708
```

Input Arguments

`transformation` — Transformation
`se2` object | *N*-element array of `se2` objects

Transformation, specified as an `se2` object or as an *N*-element array of `se2` objects. *N* is the total number of transformations.

Data Types: `single` | `double`

`rotation` — Rotation
`so2` object | *N*-element array of `so2` objects

Rotation, specified as an `so2` object or as an *N*-element array of `so2` objects. *N* is the total number of rotations.

Output Arguments

`pose` — 2-D compact pose
N-by-3 matrix

2-D compact pose, returned as an *N*-by-3 matrix, where each row is of the form [*x y theta*]. *N* is the total number of transformations specified. *x* and *y* are the *xy*-position and *theta* is the rotation about the *z*-axis.

Version History
Introduced in R2023a

See Also
`se2` | `so2`

xyzquat

Convert transformation or rotation to compact 3-D pose representation

Syntax

```
pose = xyzquat(transformation)
pose = xyzquat(rotation)
```

Description

`pose = xyzquat(transformation)` converts a transformation `transformation` to a compact 3-D pose representation `pose`.

`pose = xyzquat(rotation)` converts a rotation `rotation` to a compact 3-D pose representation `pose` with no translation.

Examples

Convert SE(3) Transformation to 3-D Compact Pose

Create SE(3) transformation with an *xyz*-position of `[2 3 1]` and a rotation defined by a numeric quaternion. Use the `eul2quat` function to create the numeric quaternion.

```
trvec = [2 3 1];
quat1 = eul2quat([0 0 deg2rad(30)]);
pose1 = [trvec quat1]
```

```
pose1 = 1×7

    2.0000    3.0000    1.0000    0.9659    0.2588         0         0
```

```
T = se3(pose1,"xyzquat")
```

```
T = se3
    1.0000         0         0    2.0000
         0    0.8660   -0.5000    3.0000
         0    0.5000    0.8660    1.0000
         0         0         0    1.0000
```

Convert the transformation back into a compact pose.

```
pose2 = xyzquat(T)
```

```
pose2 = 1×7

    2.0000    3.0000    1.0000    0.9659    0.2588         0         0
```

Convert SO(3) Rotation to 3-D Compact Pose

Create SO(3) rotation defined by a numeric quaternion. Use the `eul2quat` function to create the numeric quaternion.

```
quat1 = eul2quat([0 0 deg2rad(30)])
```

```
quat1 = 1×4

    0.9659    0.2588         0         0
```

```
R = so3(quat1,"quat")
```

```
R = so3
    1.0000         0         0
         0    0.8660   -0.5000
         0    0.5000    0.8660
```

Convert the rotation into a 3-D compact pose.

```
pose1 = xyzquat(R)
```

```
pose1 = 1×7

         0         0         0    0.9659    0.2588         0         0
```

Input Arguments

`transformation` — Transformation
`se3` object | *N*-element array of `se3` objects

Transformation, specified as an `se3` object or as an *N*-element array of `se3` objects. *N* is the total number of transformations.

`rotation` — Rotation
`so3` object | *N*-element array of `so3` objects

Rotation, specified as an `so3` object or as an *N*-element array of `so3` objects. *N* is the total number of rotations.

Output Arguments

`pose` — 3-D compact pose
M-by-3 matrix

3-D compact pose, returned as an *M*-by-3 matrix, where each row is of the form [x y z qx qy qz qw]. *M* is the total number of transformations specified. x, y, z comprise the xyz-position and qw, qx, qy, and qz are the quaternion rotations in w, x, y, and z, respectively.

Version History

Introduced in R2023a

See Also

se3 | so3

open

Open the Unreal Editor

Syntax

```
[status,result] = open(sim3dEditorObj)
```

Description

`[status,result] = open(sim3dEditorObj)` opens the Unreal Engine project in the Unreal Editor.

To develop scenes with the Unreal Editor and co-simulate with Simulink, you need the UAV Toolbox Interface for Unreal Engine Projects support package. The support package contains an Unreal Engine project that allows you to customize the UAV Toolbox scenes. For information about the support package, see “Customize Unreal Engine Scenes for UAVs”.

Input Arguments

`sim3dEditorObj` — sim3d.Editor object
`sim3d.Editor` object

`sim3d.Editor` object for the Unreal Engine project.

Output Arguments

`status` — Command exit status
0 | nonzero integer

Command exit status, returned as either `0` or a nonzero integer. When the command is successful, `status` is `0`. Otherwise, `status` is a nonzero integer.

- If `command` includes the ampersand character (&), then `status` is the exit status when `command` starts
- If `command` does not include the ampersand character (&), then `status` is the exit status upon `command` completion.

`result` — Output of operating system command
character vector

Output of the operating system command, returned as a character vector. The system shell might not properly represent non-Unicode® characters.

See Also

`sim3d.Editor`

getGraph

Graph object representing tree structure

Syntax

```
g = getGraph(frames)
g = getGraph(frames,timestamp)
```

Description

`g = getGraph(frames)` returns a MATLAB `graph` object showing the child-parent relationships between frames at the last timestamp in the `frames` `transformTree` object.

`g = getGraph(frames,timestamp)` returns a MATLAB `graph` object showing the child-parent relationships between frames at the specified timestamp.

Input Arguments

`frames` — Transform tree defining the child-parent frame relationship at given timestamps

`transformTree` object

Transform tree defining the child-parent frame relationship at given timestamps, specified as a `transformTree` object.

`timestamp` — Time for querying the frames

scalar in seconds

Time for querying the frames, specified as a scalar in seconds.

Output Arguments

g — MATLAB graph

`graph` object

MATLAB graph, specified as a `graph` object. This graph reflects the parent-child relationship of the transforms defined in the transform tree object, `frames`.

Version History

Introduced in R2020b

See Also

Objects

`transformTree` | `fixedwing` | `multirotor` | `uavDubinsPathSegment`

Functions
`getTransform` | `info` | `removeTransform` | `show` | `updateTransform`

getTransform

Get relative transform between frames

Syntax

```
tform = getTransform(frames,targetframe,sourceframe)
tform = getTransform(frames,targetframe,sourceframe,timestamp)
```

Description

`tform = getTransform(frames,targetframe,sourceframe)` returns the relative transforms that convert points in the `sourceFrame` coordinate frame to the `targetFrame`. By default, this function uses the last timestamp for both frames specified in `frames`.

`tform = getTransform(frames,targetframe,sourceframe,timestamp)` returns the relative transforms at the given timestamp. If the given time is not specified in the transform tree, `frames`, the function performs interpolation using a constant velocity assumption for linear motion, and spherical linear interpolation (SLERP) for angular motion.

Input Arguments

`frames` — Transform tree defining the child-parent frame relationship at given timestamps
`transformTree` object

Transform tree defining the child-parent frame relationship at given timestamps, specified as a `transformTree` object.

`sourceframe` — Source frame names
string scalar | character vector | string array | cell array character vector

Source frame names specified as a string scalar, character vector, string array, or cell array of character vectors. The source frame is the frame you have coordinates in, and the target frame is the frame you want to convert those coordinates to. Each element of the array corresponds to the same element in `targetframe` and the length matches the *n*-dimension of `tform`.

Data Types: `char` | `string` | `cell`

`targetframe` — Target frame names
string scalar | character vector | string array | cell array character vector

Target frame names specified as a string scalar, character vector, string array, or cell array of character vectors. The source frame is the frame you have coordinates in, and the target frame is the frame you want to convert those coordinates to. Each element of the array corresponds to the same element in `sourceframe` and the length matches the *n*-dimension of `tform`.

Data Types: `char` | `string` | `cell`

`timestamp` — Time for querying the frames
scalar in seconds | vector

Time for querying the frames, specified as a scalar or vector of scalars in seconds. For timestamps specified before the first timestamp in `frames`, the function returns `NaN` values. For timestamps specified after the last timestamp, the most recent (largest timestamp) transformation is returned.

Output Arguments

`tform` — Transformations that converts points from source frames to target frames
4-by-4 homogenous transformation matrix | 4-by-4-by-*n* matrix array

Transformations that converts points from the source frames to the target frames specified as a 4-by-4 transformation matrix or a 4-by-4-by-*n* matrix array. Each matrix in the array corresponds to the same element of `targetframe`, `sourceframe`, and `timestamp`.

Version History

Introduced in R2020b

See Also

Objects
`transformTree` | `fixedwing` | `multirotor` | `uavDubinsPathSegment`

Functions
`getGraph` | `info` | `removeTransform` | `show` | `updateTransform`

info

List all frame names and stored timestamps

Syntax

`list = info(frames)`

Description

`list = info(frames)` returns a structure array with an element for each frame containing the frame name, parent frame, and all stored timestamps.

Input Arguments

`frames` — Transform tree defining the child-parent frame relationship at given timestamps
`transformTree` object

Transform tree defining the child-parent frame relationship at given timestamps, specified as a `transformTree` object.

Output Arguments

`list` — List of frame names, parents, and timestamps
structure array

List of frame names, parents, and timestamps, specified as a structure array. The elements of the structure array are:

- `FrameNames` –– String scalars listing each frame name.
- `ParentNames` –– String scalars listing the parent of each frame. The base frame returns an empty string.
- `Timestamps` –– Vectors of timestamps for each frame. Each vector is padded with `NaN`s based on the MaxNumTransforms property of `frames`.

Version History

Introduced in R2020b

See Also

Objects
`transformTree` | `fixedwing` | `multirotor` | `uavDubinsPathSegment`

Functions
`getGraph` | `getTransform` | `removeTransform` | `show` | `updateTransform`

removeTransform

Remove frame transform relative to its parent

Syntax

```
removeTransform(frames,framename,timestamp)
removeTransform(frames,framename,timeStart,timeEnd)
```

Description

`removeTransform(frames,framename,timestamp)` removes the frame transforms between the given frame name and their parent frame at the specified timestamps.

`removeTransform(frames,framename,timeStart,timeEnd)` removes all the frame transforms for the given frame name in the time interval, `[timeStart timeEnd]`.

Input Arguments

`frames` — Transform tree defining the child-parent frame relationship at given timestamps
`transformTree` object

Transform tree defining the child-parent frame relationship at given timestamps, specified as a `transformTree` object.

`framename` — Frame name
string scalar | character vector

Frame name with transforms you want to remove, specified as a string scalar or character vector.

Data Types: `char` | `string` | `cell`

`timestamp` — Times for removing transforms
scalar in seconds | vector

Times for removing transforms, specified as a scalar or vector of scalars in seconds. These timestamps must be specified for each of the frame transforms that you want to remove.

`timeStart` — Initial time for removing transforms
scalar in seconds

Initial time for removing transforms, specified as a scalar in seconds. All transforms for the given `framename` are removed from `timeStart` to `timeEnd`.

`timeEnd` — Final time for removing transforms
scalar in seconds

Final time for removing transforms, specified as a scalar in seconds. All transforms for the given `framename` are removed from `timeStart` to `timeEnd`.

Version History

Introduced in R2020b

See Also

Objects

`transformTree` | `fixedwing` | `multirotor` | `uavDubinsPathSegment`

Functions

`getGraph` | `getTransform` | `info` | `show` | `updateTransform`

show

Show transform tree

Syntax

```
hAx = show(frames)
hAx = show(frames,timestamp)
hAx = show( ___ ,Name,Value)
```

Description

`hAx = show(frames)` displays the transform tree at the last timestamp in the sequence.

`hAx = show(frames,timestamp)` displays the transform tree at the specified timestamp. If the specified time is not specified in the transform tree, `frames`, the function performs interpolation using a constant velocity assumption for linear motion, and spherical linear interpolation (SLERP) for angular motion.

`hAx = show( ___ ,Name,Value)` specifies additional options specified by one or more name-value pair arguments.

Input Arguments

`frames` — Transform tree defining the child-parent frame relationship at given timestamps
`transformTree` object

Transform tree defining the child-parent frame relationship at given timestamps, specified as a `transformTree` object.

`timestamp` — Time for querying the frames
scalar in seconds | vector

Time for querying the frames, specified as a scalar or vector of scalars in seconds. If the given time is not specified in the transform tree, `frames`, the function performs interpolation using a constant velocity assumption for linear motion, and spherical linear interpolation (SLERP) for angular motion. For timestamps specified after the last timestamp, the most recent (largest timestamp) transformation is returned.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `'ShowArrow',true` draws arrows between parent to child frames

`ShowArrow` — Draw arrows from parent to child frames
`false` (default) | `true`

Draw arrows from parent to child frames, specified as `true` or `false`.

Data Types: `logical`

`FrameSizes` — Axis sizes for frames
`struct("root",1)` (default) | structure

Axis sizes for frames, specified as a structure. Specify each frame name as a the field with a scalar for that frame's relative size.

Example: `struct("root",2,"frameA",5)`

Data Types: `struct`

`FrameNames` — Frames to plot
all frames (default) | string scalar | character vector | string array | cell array of character vectors

Frames to plot, specified as a string, character vector, string array, or cell array of character vectors. Use this argument to specify a subset of frame names to display in the figure.

Example: `["Frame1","Frame3","Frame9"]`

Data Types: `char` | `string` | `cell`

`Parent` — Axes on which to plot
`Axes` object

Axes on which to plot, specified as an `Axes` object.

Output Arguments

`hAx` — Axes
`Axes` object

Axes under which the transform tree is shown, returned as an `Axes` object. For more information, see `Axes Properties`.

Version History

Introduced in R2020b

See Also

Objects
`transformTree` | `fixedwing` | `multirotor` | `uavDubinsPathSegment`

Functions
`getGraph` | `getTransform` | `info` | `removeTransform` | `updateTransform`

updateTransform

Update frame transform relative to its parent

Syntax

```
updateTransform(frames,parentframe,childframe,position,orientation,timestamp)
updateTransform(frames,parentframe,childframe,tform,timestamp)
```

Description

`updateTransform(frames,parentframe,childframe,position,orientation,timestamp)` updates the relative transforms between child frames and their parents with a given position and orientation at the specified time stamps. The position and orientation are given in the parent reference frame.

`updateTransform(frames,parentframe,childframe,tform,timestamp)` updates the relative transforms between child frames and their parents with a given 4-by-4 homogenous transform, `tform`.

Input Arguments

`frames` — Transform tree defining the child-parent frame relationship at given timestamps
`transformTree` object

Transform tree defining the child-parent frame relationship at given timestamps, specified as a `transformTree` object.

`parentframe` — Parent frame names
string scalar | character vector | string array | cell array character vector

Parent frame names specified as a string scalar, character vector, string array, or cell array of character vectors. Transformations specified in `tform` or `position` and `orientation` are relative to the parent frame. Each element of `parentframe` corresponds to the same element in `childframe`.

Data Types: `char` | `string` | `cell`

`childframe` — Child frame names
string scalar | character vector | string array | cell array character vector

Child frame names specified as a string scalar, character vector, string array, or cell array of character vectors. The function attaches the child frame to the parent frame. Transformations specified in `tform` or `position` and `orientation` are relative to the parent frame. Each element of `parentframe` corresponds to the same element in `childframe`.

Data Types: `char` | `string` | `cell`

`position` — Relative position of child frame to parent
three-element [*x y z*] vector

Relative position of child frame to parent, specified as a three-element `[x y z]` vector. Specify the relative orientation in `orientation`.

`orientation` — Relative orientation of child frame to parent
three-element `[x y z]` vector

Relative orientation of child frame to parent, specified as a three-element `[x y z]` vector. Specify the relative position in `position`.

`tform` — Relative transform of child frame to parent
4-by-4 homogenous transformation matrix

Relative transform of child frame to parent, specified as a 4-by-4 homogenous transformation matrix.

`timestamp` — Time for querying the frames
scalar in seconds | vector

Time for querying the frames, specified as a scalar or vector of scalars in seconds. If the specified time is not specified in the transform tree, `frames`, the function performs interpolation using a constant velocity assumption for linear motion, and spherical linear interpolation (SLERP) for angular motion. For timestamps specified after the last timestamp, the most recent (largest timestamp) transformation is returned.

Version History

Introduced in R2020b

See Also

Objects
`transformTree` | `fixedwing` | `multirotor` | `uavDubinsPathSegment`

Functions
`getGraph` | `getTransform` | `info` | `removeTransform` | `show`

connect

Connect poses with UAV Dubins connection path

Syntax

```
[pathSegObj,pathCost] = connect(connectionObj,start,goal)
[pathSegObj,pathCost] = connect(connectionObj,start,
goal,'PathSegments','all')
```

Description

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

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

Examples

Connect Poses of All Valid UAV Dubins Paths

This example shows how to calculate all valid UAV Dubins path segments and connect poses using the `uavDubinsConnection` object.

Calculate All Possible Path Segments

Create a `uavDubinsConnection` object.

```
connectionObj = uavDubinsConnection;
```

Define start and goal poses as [`x`, `y`, `z`, `headingAngle`] vectors.

```
startPose = [0 0 0 0]; % [meters, meters, meters, radians]
goalPose = [0 0 20 pi];
```

Calculate all possible path segments and connect the poses.

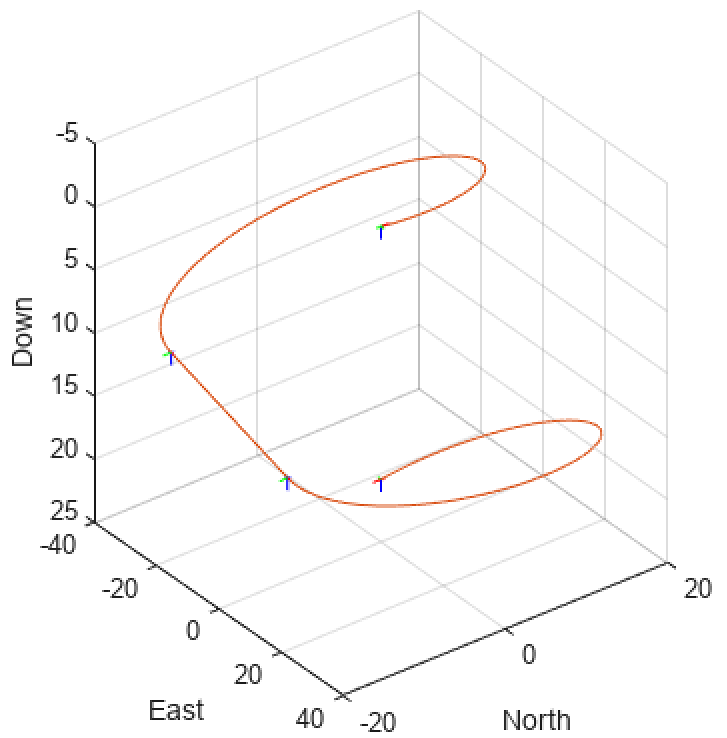
```
[pathSegObj,pathCosts] = connect(connectionObj,startPose,goalPose,'PathSegments','all');
```

Path Validation and Visualization

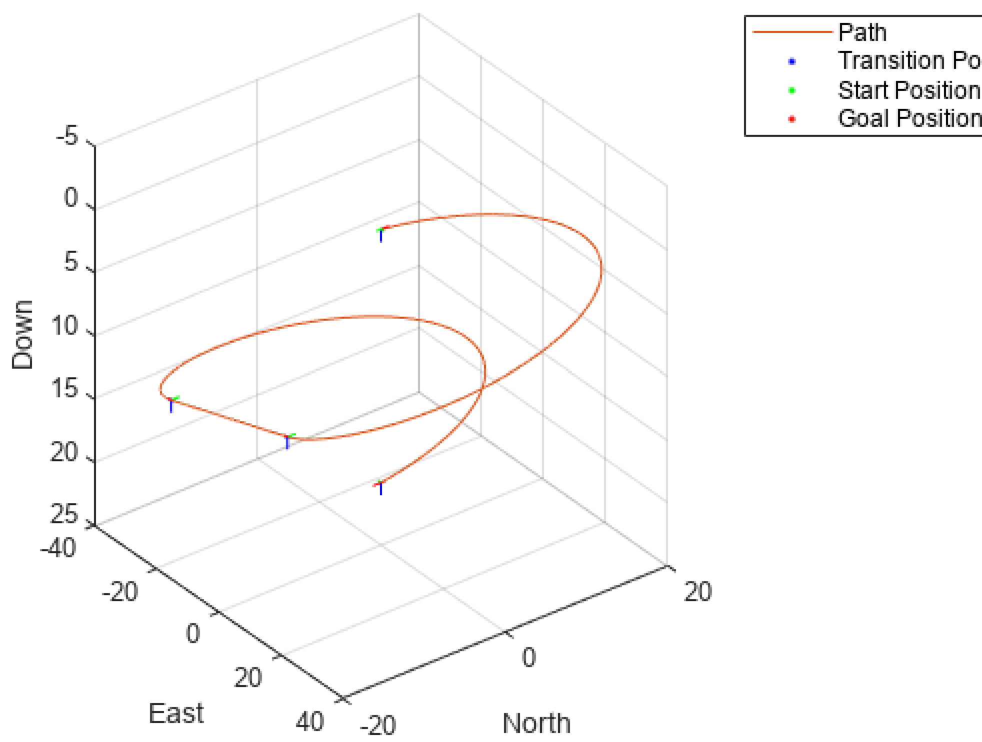
Check the validity of all the possible path segments and display the valid paths along with their motion type and path cost.

```
for i = 1:length(pathSegObj)
    if ~isnan(pathSegObj{i}.Length)
        figure
        show(pathSegObj{i})
        fprintf('Motion Type: %s\nPath Cost: %f\n',strjoin(pathSegObj{i}.MotionTypes),pathCosts(
```

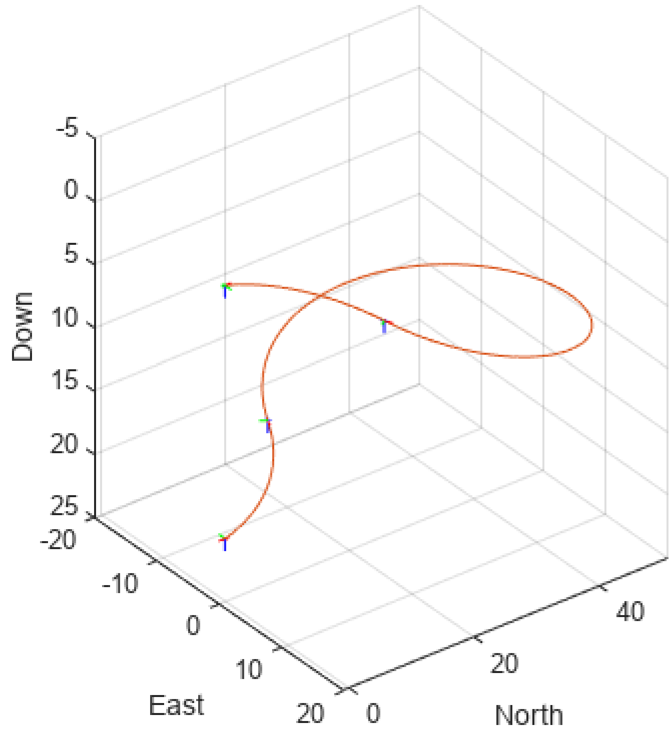
```
    end
end
```



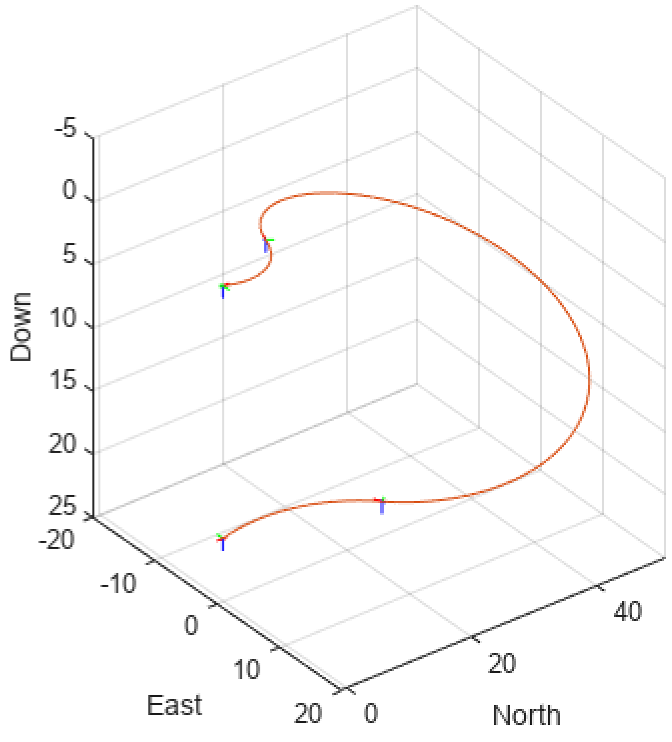

```
Motion Type: L S L N
Path Cost: 214.332271
```




```
Motion Type: R S R N
Path Cost: 214.332271
```




```
Motion Type: R L R N
Path Cost: 138.373157
```




```
Motion Type: L R L N
Path Cost: 138.373157
```

Input Arguments

connectionObj — Path connection type

uavDubinsConnection object

Path connection type, specified as a `uavDubinsConnection` object. This object defines the parameters of the connection.

start — Initial pose of UAV

four-element numeric vector or matrix

Initial pose of the UAV at the start of the path segment, specified as a four-element numeric vector or matrix [*x*, *y*, *z*, *headingAngle*].

x, *y*, and *z* specify the position in meters. *headingAngle* specifies the heading angle in radians. The heading angle is measured clockwise from north to east. Each row of the matrix corresponds to a different start pose.

The pose follows the north-east-down coordinate system.

The `start` and `goal` pose inputs can be any of these combinations:

- Single start pose with single goal pose.
- Multiple start poses with single goal pose.
- Single start pose with multiple goal poses.
- Multiple start poses with multiple goal poses.

goal — Goal pose of UAV
four-element numeric vector or matrix

Goal pose of the UAV at the end of the path segment, specified as a four-element numeric vector or matrix [*x*, *y*, *z*, *headingAngle*].

x, *y*, and *z* specify the position in meters. *headingAngle* specifies the heading angle in radians. The heading angle is measured clockwise from north to east. Each row of the matrix corresponds to a different goal pose.

The pose follows the north-east-down coordinate system.

The `start` and `goal` pose inputs can be any of these combinations:

- Single start pose with single goal pose.
- Multiple start poses with single goal pose.
- Single start pose with multiple goal poses.
- Multiple start poses with multiple goal poses.

Output Arguments

pathSegObj — Path segments
cell array of `uavDubinsPathSegment` objects

Path segments, returned as a cell array of `uavDubinsPathSegment` objects. The type of object depends on the input `connectionObj`. The size of the cell array depends on whether you use single or multiple `start` and `goal` poses.

By default, the function returns the path with the lowest cost for each `start` and `goal` pose.

When calling the `connect` function using the `'PathSegments','all'` name-value pair, the cell array contains all valid path segments between the specified `start` and `goal` poses.

pathCost — Cost of path segment
positive numeric scalar | positive numeric vector | positive numeric matrix

Cost of path segments, returned either as a positive numeric scalar, vector, or matrix. Each element of the cost vector corresponds to a path segment in `pathSegObj`.

By default, the function returns the path with the lowest cost for each `start` and `goal` pose.

Version History

Introduced in R2019b

Extended Capabilities

C/C++ Code Generation

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

See Also

`uavDubinsPathSegment` | `uavDubinsConnection`

exportWaypointsPlan

Export waypoints to file

Syntax

```
exportWaypointsPlan(planner,solninfo,filename)
exportWaypointsPlan( ___ ,ReferenceFrame=frame)
```

Description

`exportWaypointsPlan(planner,solninfo,filename)` uses solution information `solninfo` to export the waypoints and ground control system information to the file `filename`. You can use `filename` to deploy waypoints to a UAV using QGroundControl, a mission planner. or any other ground control software.

`exportWaypointsPlan( ___ ,ReferenceFrame=frame)` exports the waypoints in the reference frame `frame`.

Examples

Plan Coverage Path for Defined Region

This example shows how to plan a coverage path for a region in local coordinates and compares the results of using the exhaustive solver with the results of using the minimum traversal solver.

Define the vertices for a coverage space.

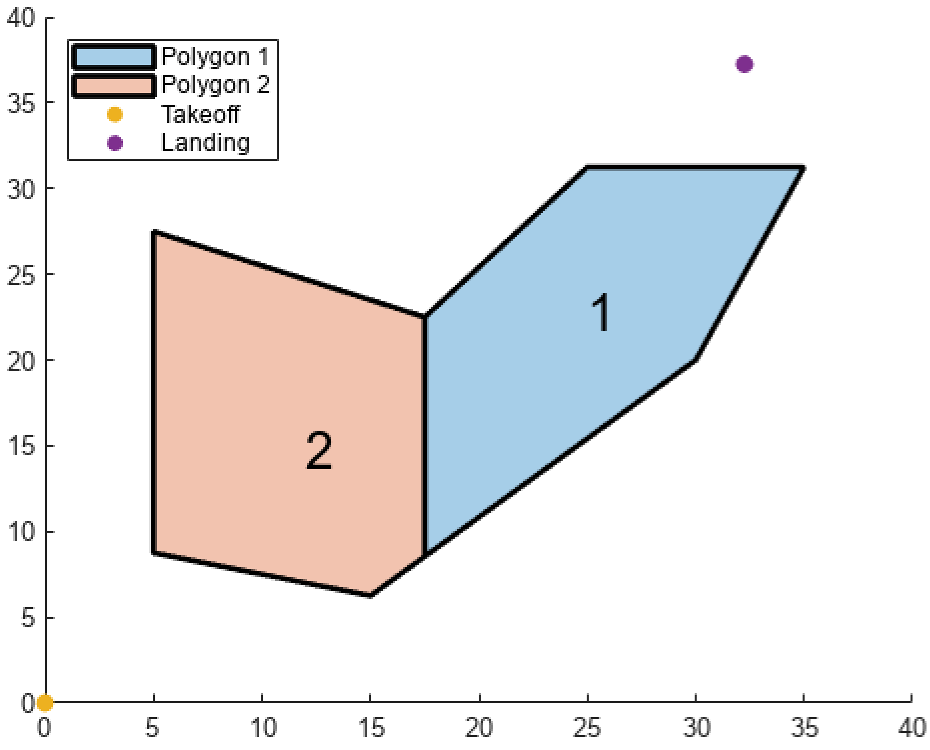
```
area = [5 8.75; 5 27.5; 17.5 22.5; 25 31.25; 35 31.25; 30 20; 15 6.25];
```

Because vertices define a concave polygon and the coverage planner requires convex polygons, decompose the polygon into convex polygons. Then create a coverage space with the polygons from decomposition.

```
polygons = coverageDecomposition(area);
cs = uavCoverageSpace(Polygons=polygons);
```

Define the takeoff and landing positions at `[0 0 0]` and `[32.25 37.25 0]`, respectively. Then show the coverage space and plot the takeoff and landing positions.

```
takeoff = [0 0 0];
landing = [32.25 37.25 0];
show(cs);
exampleHelperPlotTakeoffLandingLegend(takeoff,landing)
```



Create a coverage planner with the exhaustive solver algorithm and another coverage planner with a minimum traversal solver algorithm. Because `Polygon 2` is closer to the takeoff position, set the visiting sequence of the solver parameters such that we traverse `Polygon 2` first.

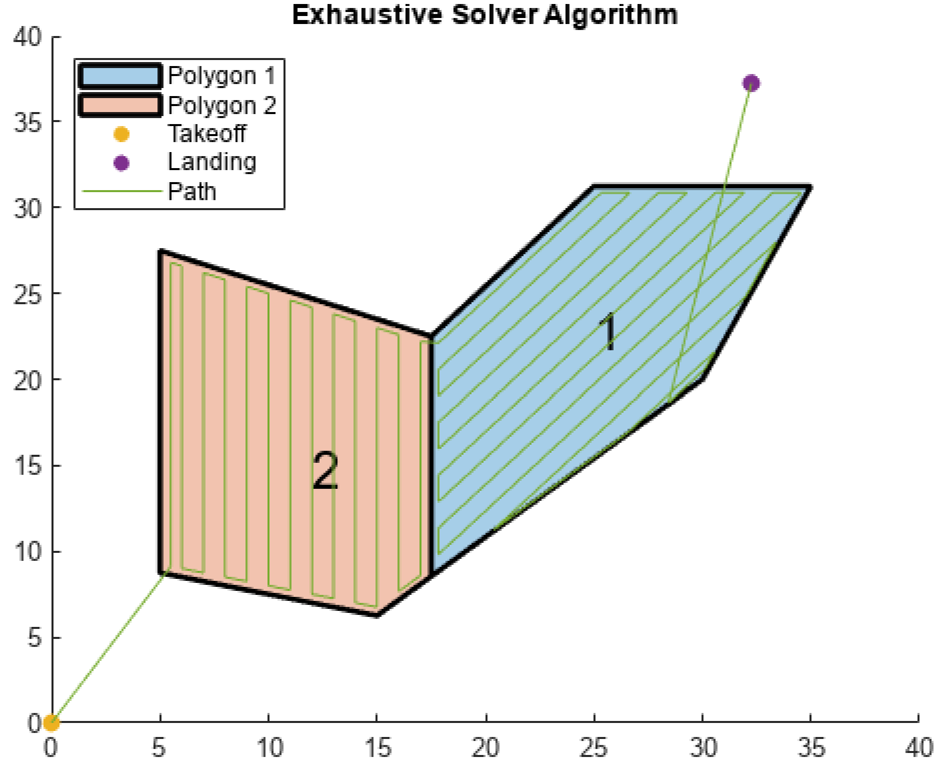
```
cpeExh = uavCoveragePlanner(cs,Solver="Exhaustive");
cpMin = uavCoveragePlanner(cs,Solver="MinTraversal");
cpeExh.SolverParameters.VisitingSequence = [2 1];
cpMin.SolverParameters.VisitingSequence = [2 1];
```

Plan with both solver algorithms using the same takeoff and landing positions.

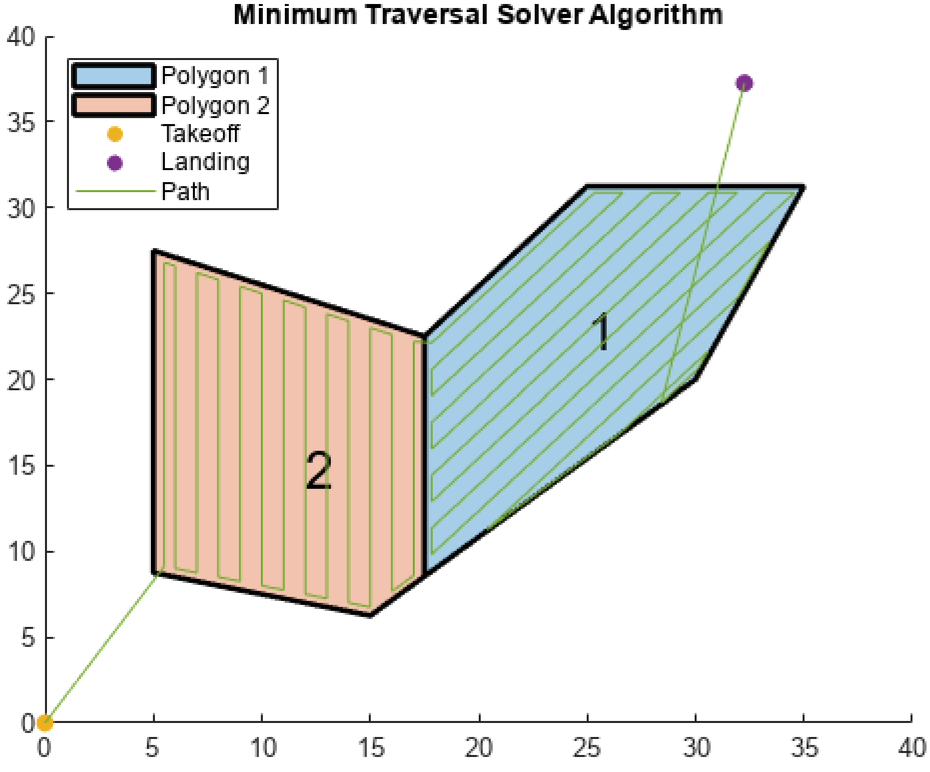
```
[wptsExh,solnExh] = plan(cpeExh,takeoff,landing);
[wptsMin,solnMin] = plan(cpMin,takeoff,landing);
```

Show the planned path for both the exhaustive and the minimum traversal algorithms.

```
figure
show(cs);
title("Exhaustive Solver Algorithm")
exampleHelperPlotTakeoffLandingLegend(takeoff,landing,wptsExh)
```




```
figure
show(cs);
title("Minimum Traversal Solver Algorithm")
exampleHelperPlotTakeoffLandingLegend(takeoff,landing,wptsMin)
```



Export the waypoints from the minimum traversal solver to a `.waypoints` file with the reference frame set to north-east-down.

```
exportWaypointsPlan(cpMin,solnMin,"coveragepath.waypoints",ReferenceFrame="NED")
```

Input Arguments

`planner` — Coverage path planner
`uavCoveragePlanner` object

Coverage path planner, specified as a `uavCoveragePlanner` object.

`solninfo` — Solution information
structure

Solution plan, specified as a structure containing these fields:

- `VisitingSequence` — *N*-element row vector denoting the order of visitation of polygons, where *N* is the total number of polygons in the coverage space. For example, `[2 1 3]` specifies that the UAV should visit polygon 2 first, polygon 1 second, and polygon 3 last.
- `SweepPattern` — *N*-element row vector of integers denoting the sweep pattern for each polygon, where *N* is the total number of polygons in the coverage space. Each element is an integer in the range [1, 4] that indicates a sweep pattern:

  - `1` — Forward sweep pattern
  - `2` — Counter-clockwise sweep pattern
  - `3` — Reverse sweep pattern
  - `4` — Reverse counter-clockwise sweep pattern

  For example, `[3 1 2]` specifies that the UAV should use the reverse sweep pattern for polygon 1, the forward sweep pattern for polygon 2, and the counter-clockwise sweep pattern for polygon 3.
- `TransitionCost` — Euclidean distance cost for transitioning between polygons including takeoff and landing distance.
- `Takeoff` — Takeoff location, specified as a three-element row vector in LLA format.
- `Landing` — Takeoff location, specified as a three-element row vector in LLA format.

Use the `plan` function to get this structure.

`filename` — File name to export information to
character vector | string scalar

File name to export information to, specified as a character vector or string scalar. Specify the format of the file by ending the character vector or string scalar with either `".txt"` or `".waypoints"`. For more information, see MAVLink File Formats.

Example: `"waypointfile.txt"`

Data Types: `char` | `string`

`frame` — Reference frame to export waypoints to
`"ENU"` (default) | `"NED"`

Reference frame to export waypoints to, specified as either `"ENU"` for east-north-up or `"NED"` for north-east-down.

Data Types: `char` | `string`

Version History

Introduced in R2023a

See Also

`uavCoveragePlanner`

Topics

"Optimally Survey and Customize Coverage of Region of Interest using Coverage Planner"

plan

Plan coverage path between takeoff and landing

Syntax

```
[path,solninfo] = plan(planner,takeoff)
[path,solninfo] = plan(planner,takeoff,landing)
[ ___ ] = plan( ___ ,VerboseMode=mode)
```

Description

The `plan` function returns a coverage path that enables you to optimally surveys a geographical area with a UAV for precision agriculture and image mapping applications.

`[path,solninfo] = plan(planner,takeoff)` returns a path consisting of 2D waypoints `path`, and information about the order of visitation of polygons and sweep permutations `solnInfo`. The landing position is the same as the takeoff position.

`[path,solninfo] = plan(planner,takeoff,landing)` specifies a landing position in addition to the takeoff position.

`[ ___ ] = plan( ___ ,VerboseMode=mode)` specifies the diagnostic printing mode.

Examples

Plan Coverage Path Using Geodetic Coordinates

This example shows how to plan a coverage path that surveys the parking lots of the MathWorks Lakeside campus.

Get the geodetic coordinates for the MathWorks Lakeside campus. Then create the limits for our map.

```
mwLS = [42.3013 -71.375 0];
latlim = [mwLS(1)-0.003 mwLS(1)+0.003];
lonlim = [mwLS(2)-0.003 mwLS(2)+0.003];
```

Create a figure containing the map with the longitude and latitude limits.

```
fig = figure;
g = geoaxes(fig,Basemap="satellite");
geolimits(latlim,lonlim)
```

Get the outline of the first parking lot in longitude and latitude coordinates. Then create the polygon by concatenating them.

```
pl1lat = [42.3028 42.30325 42.3027 42.3017 42.3019]';
pl1lon = [-71.37527 -71.37442 -71.3736 -71.37378 -71.375234]';
pl1Poly = [pl1lat pl1lon];
```

Repeat the process for the second parking lot.

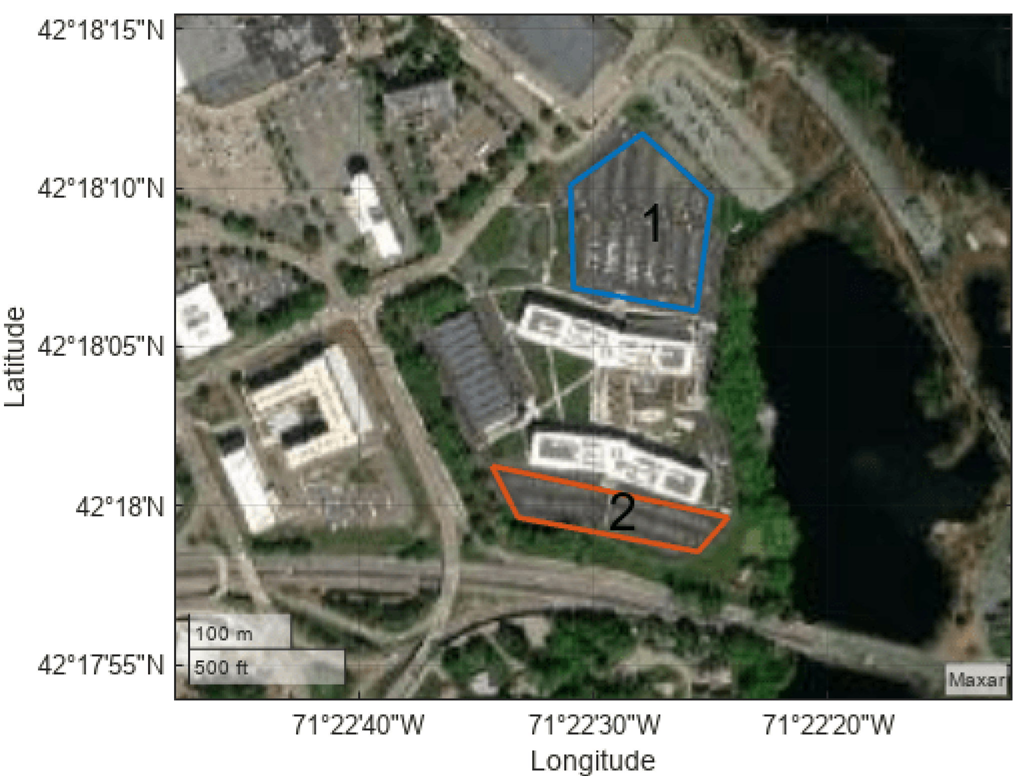
```
pl2lat = [42.30035 42.2999 42.2996 42.2999]';
pl2lon = [-71.3762 -71.3734 -71.37376 -71.37589]';
pl2poly = [pl2lat pl2lon];
```

Create the coverage space with both of those polygons, set the coverage space to use geodetic coordinates, and set the reference location to the MathWorks Lakeside campus location.

```
cs = uavCoverageSpace(Polygons={pl1Poly,pl2poly},UseLocalCoordinates=false,ReferenceLocation=mwLS
```

Set the height at which to fly the UAV to 25 meters, and the sensor footprint width to 20 meters. Then show the coverage space on the map.

```
ReferenceHeight = 25;
cs.UnitWidth = 20;
show(cs,Parent=g);
```



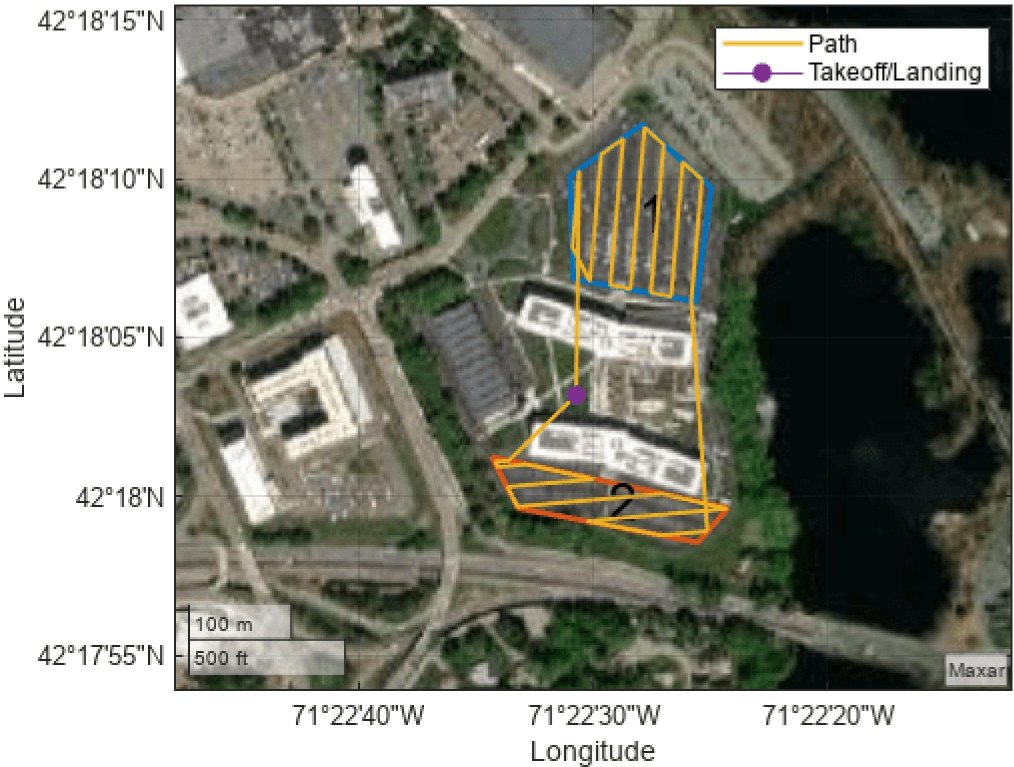
Set the sweep angle for polygons 1 and 2 to 85 and 5 degrees, respectively, to have paths that are parallel to the roads in the parking lots. Then create the coverage planner for that coverage space with the exhaustive solver algorithm.

```
setCoveragePattern(cs,1,SweepAngle=85)
setCoveragePattern(cs,2,SweepAngle=5)
cp = uavCoveragePlanner(cs,Solver="Exhaustive");
```

Set the takeoff position to a location in the courtyard, then plan the coverage path.

```
takeoff = [42.30089 -71.3752, 0];
[wp,soln] = plan(cp,takeoff);
```

```
hold on
geoplot(wp(:,1),wp(:,2),LineWidth=1.5);
geoplot(takeoff(1),takeoff(2),MarkerSize=25,Marker=".")
legend("","","Path","Takeoff/Landing")
hold off
```



Plan Coverage Path for Defined Region

This example shows how to plan a coverage path for a region in local coordinates and compares the results of using the exhaustive solver with the results of using the minimum traversal solver.

Define the vertices for a coverage space.

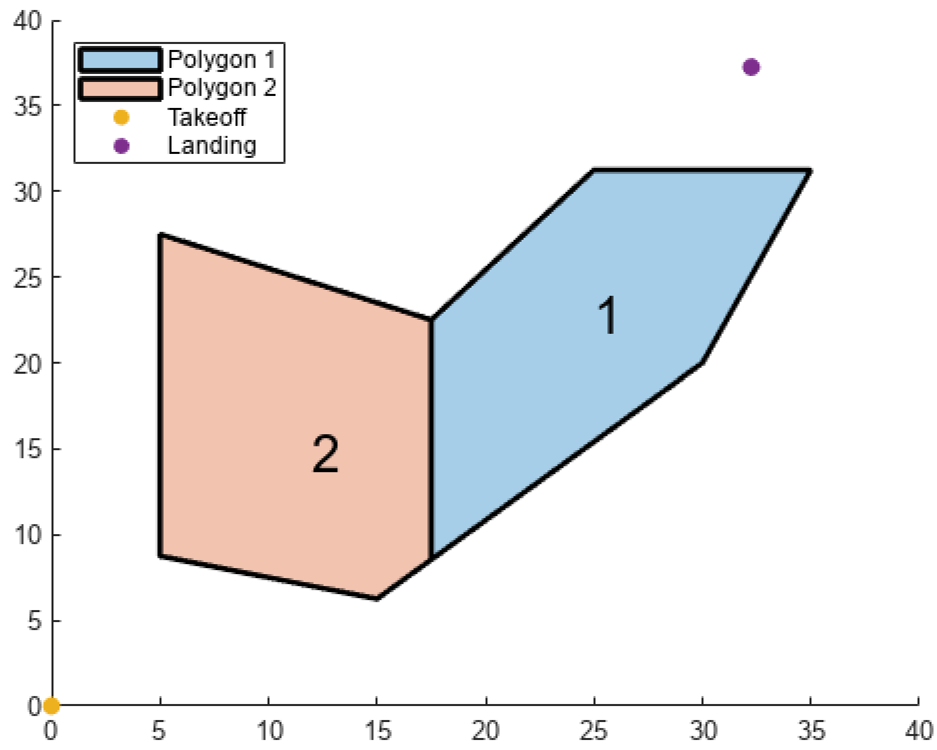
```
area = [5 8.75; 5 27.5; 17.5 22.5; 25 31.25; 35 31.25; 30 20; 15 6.25];
```

Because vertices define a concave polygon and the coverage planner requires convex polygons, decompose the polygon into convex polygons. Then create a coverage space with the polygons from decomposition.

```
polygons = coverageDecomposition(area);
cs = uavCoverageSpace(Polygons=polygons);
```

Define the takeoff and landing positions at `[0 0 0]` and `[32.25 37.25 0]`, respectively. Then show the coverage space and plot the takeoff and landing positions.

```
takeoff = [0 0 0];
landing = [32.25 37.25 0];
show(cs);
exampleHelperPlotTakeoffLandingLegend(takeoff,landing)
```



Create a coverage planner with the exhaustive solver algorithm and another coverage planner with a minimum traversal solver algorithm. Because `Polygon 2` is closer to the takeoff position, set the visiting sequence of the solver parameters such that we traverse `Polygon 2` first.

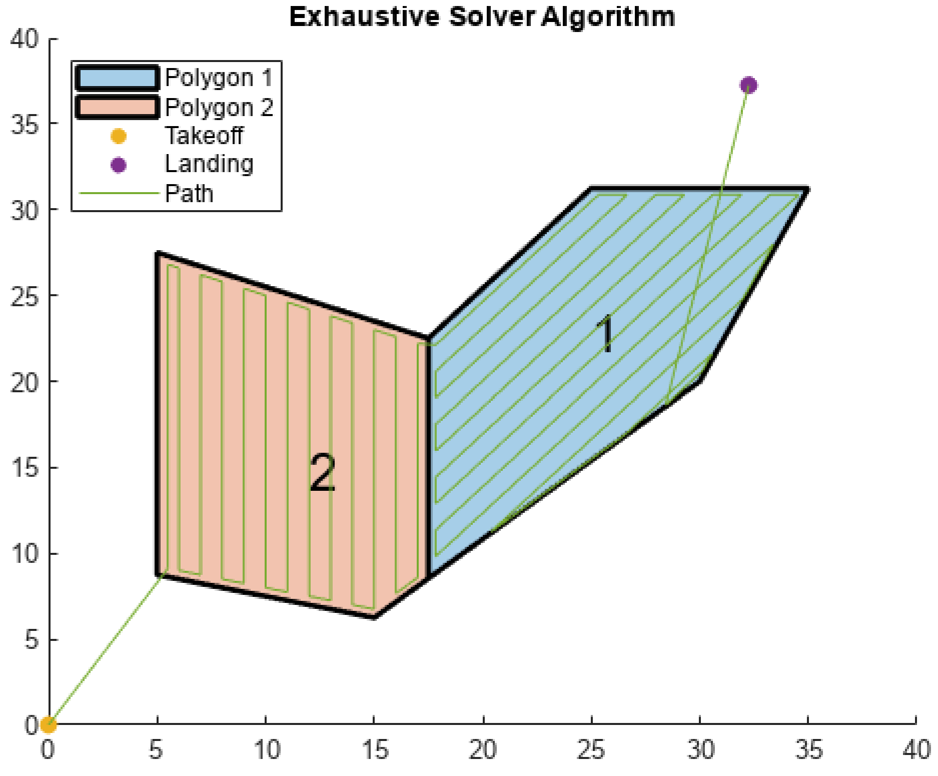
```
cpeExh = uavCoveragePlanner(cs,Solver="Exhaustive");
cpMin = uavCoveragePlanner(cs,Solver="MinTraversal");
cpeExh.SolverParameters.VisitingSequence = [2 1];
cpMin.SolverParameters.VisitingSequence = [2 1];
```

Plan with both solver algorithms using the same takeoff and landing positions.

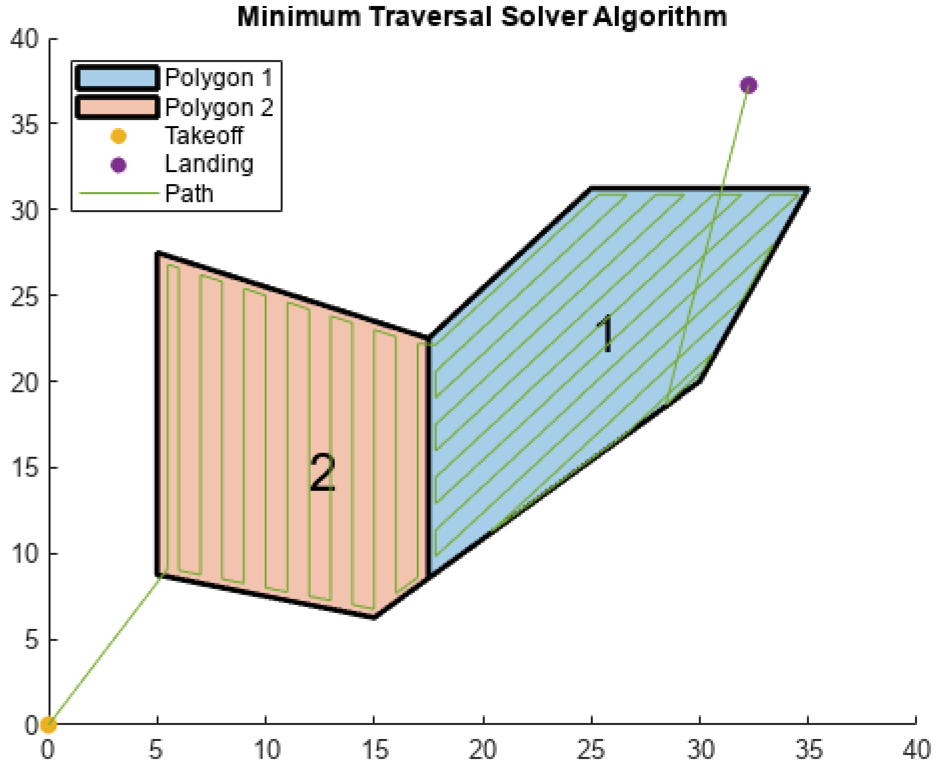
```
[wptsExh,solnExh] = plan(cpeExh,takeoff,landing);
[wptsMin,solnMin] = plan(cpMin,takeoff,landing);
```

Show the planned path for both the exhaustive and the minimum traversal algorithms.

```
figure
show(cs);
title("Exhaustive Solver Algorithm")
exampleHelperPlotTakeoffLandingLegend(takeoff,landing,wptsExh)
```




```
figure
show(cs);
title("Minimum Traversal Solver Algorithm")
exampleHelperPlotTakeoffLandingLegend(takeoff,landing,wptsMin)
```



Export the waypoints from the minimum traversal solver to a `.waypoints` file with the reference frame set to north-east-down.

```
exportWaypointsPlan(cpMin,solnMin,"coveragepath.waypoints",ReferenceFrame="NED")
```

Input Arguments

`planner` — Coverage path planner
`uavCoveragePlanner` object

Coverage path planner, specified as a `uavCoveragePlanner` object.

`takeoff` — UAV takeoff position
three-element row vector

UAV takeoff position, specified as a three-element row vector.

The format depends on the `UseLocalCoordinates` property of the `CoverageSpace` property of `planner`:

- When `UseLocalCoordinates` is `true` the format is local *xyz*-coordinates in the form [*x y z*], in meters. `UseLocalCoordinates` is true by default.
- When `UseLocalCoordinates` is `false` the format is geodetic coordinates in the form [*latitude longitude altitude*]. *latitude* and *longitude* are in degrees, and *altitude* is in meters.

Data Types: `double`

`landing` — UAV landing position
three-element row vector

UAV landing position, specified as a three-element row vector.

The format depends on the `UseLocalCoordinates` property of the `CoverageSpace` property of `planner`:

- `UseLocalCoordinates=true` — Format is local *xyz*-coordinates in the form [*x y z*], in meters. `UseLocalCoordinates` is true by default.
- `UseLocalCoordinates=false` — Format is geodetic coordinates in the form [*latitude longitude altitude*]. *latitude* and *longitude* are in degrees, and *altitude* is in meters.

Data Types: `double`

`mode` — Verbose mode
`false` or `0` (default) | `true` or `1`

Verbose mode, specified as `true` or (`1`) to output diagnostic information about the planning algorithm to the Command Window, or `false` or (`0`) to output no information.

Output Arguments

`path` — Waypoint path
N-by-2 matrix

Waypoint path, specified as a *N*-by-2 matrix.

The format depends on the `UseLocalCoordinates` property of the `CoverageSpace` property of `planner`:

- `UseLocalCoordinates=true` — Format is local *xy*-coordinates in the form [*x y*], in meters. `UseLocalCoordinates` is true by default.
- `UseLocalCoordinates=false` — Format is geodetic coordinates in the form [*latitude longitude*]. *latitude* and *longitude* are in degrees.

`solninfo` — Solution information
structure

Solution plan, returned as a structure containing these fields:

- `VisitingSequence` — *N*-element row vector denoting the order of visitation of polygons, where *N* is the total number of polygons in the coverage space. For example, `[2 1 3]` specifies that the UAV should visit polygon 2 first, polygon 1 second, and polygon 3 last.
- `SweepPattern` — *N*-element row vector of integers denoting the sweep pattern for each polygon, where *N* is the total number of polygons in the coverage space. Each element is an integer in the range [1, 4] that indicates a sweep pattern:
  - `1` — Forward sweep pattern
  - `2` — Counter-clockwise sweep pattern

  - `3` — Reverse sweep pattern
  - `4` — Reverse counter-clockwise sweep pattern

  For example, `[3 1 2]` specifies that the UAV should use the reverse sweep pattern for polygon 1, the forward sweep pattern for polygon 2, and the counter-clockwise sweep pattern for polygon 3.
- `TransitionCost` — Euclidean distance cost for transitioning between polygons including takeoff and landing distance.
- `Takeoff` — Takeoff location, specified as a three-element row vector in LLA format.
- `Landing` — Takeoff location, specified as a three-element row vector in LLA format.

Version History

Introduced in R2023a

See Also

`uavCoveragePlanner` | `exportWaypointsPlan`

Topics

"Optimally Survey and Customize Coverage of Region of Interest using Coverage Planner"

setCoveragePattern

Set sweep angle for polygon in coverage area

Syntax

```
setCoveragePattern(space,index,SweepAngle=angle)
```

Description

`setCoveragePattern(space,index,SweepAngle=angle)` sets the angle at which the UAV sweeps the polygon with the index `index` in coverage space `space` to angle `angle`.

Use the `show` function to visualize the polygons in the coverage space to identify which polygon index to use.

Note `space` must contain at least one polygon.

Examples

Plan Coverage Path Using Geodetic Coordinates

This example shows how to plan a coverage path that surveys the parking lots of the MathWorks Lakeside campus.

Get the geodetic coordinates for the MathWorks Lakeside campus. Then create the limits for our map.

```
mwLS = [42.3013 -71.375 0];
latlim = [mwLS(1)-0.003 mwLS(1)+0.003];
lonlim = [mwLS(2)-0.003 mwLS(2)+0.003];
```

Create a figure containing the map with the longitude and latitude limits.

```
fig = figure;
g = geoaxes(fig,Basemap="satellite");
geolimits(latlim,lonlim)
```

Get the outline of the first parking lot in longitude and latitude coordinates. Then create the polygon by concatenating them.

```
pl1lat = [42.3028 42.30325 42.3027 42.3017 42.3019]';
pl1lon = [-71.37527 -71.37442 -71.3736 -71.37378 -71.375234]';
pl1Poly = [pl1lat pl1lon];
```

Repeat the process for the second parking lot.

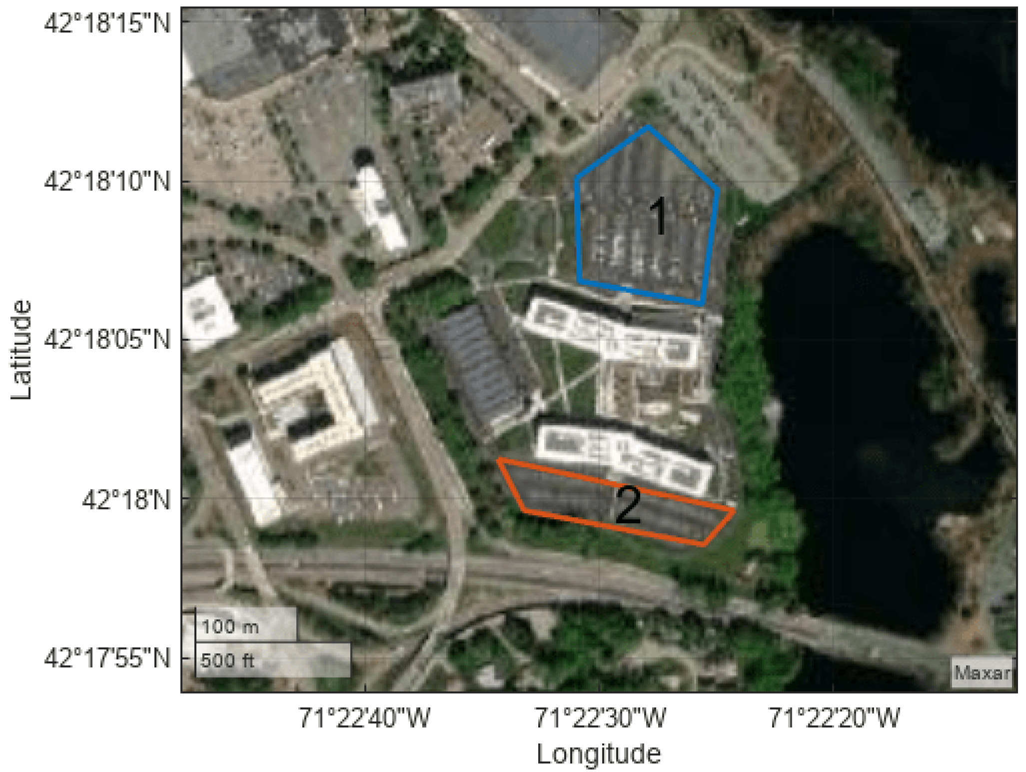
```
pl2lat = [42.30035 42.2999 42.2996 42.2999]';
pl2lon = [-71.3762 -71.3734 -71.37376 -71.37589]';
pl2poly = [pl2lat pl2lon];
```

Create the coverage space with both of those polygons, set the coverage space to use geodetic coordinates, and set the reference location to the MathWorks Lakeside campus location.

```
cs = uavCoverageSpace(Polygons={pl1Poly,pl2poly},UseLocalCoordinates=false,ReferenceLocation=mwLS
```

Set the height at which to fly the UAV to 25 meters, and the sensor footprint width to 20 meters. Then show the coverage space on the map.

```
ReferenceHeight = 25;
cs.UnitWidth = 20;
show(cs,Parent=g);
```



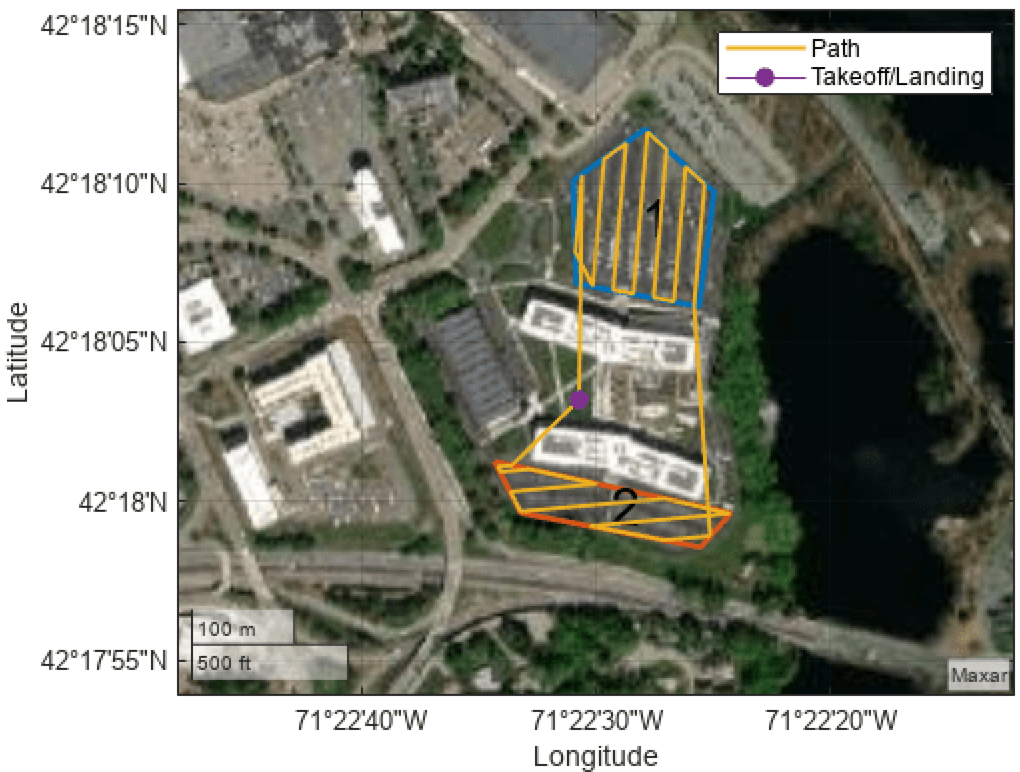
Set the sweep angle for polygons 1 and 2 to 85 and 5 degrees, respectively, to have paths that are parallel to the roads in the parking lots. Then create the coverage planner for that coverage space with the exhaustive solver algorithm.

```
setCoveragePattern(cs,1,SweepAngle=85)
setCoveragePattern(cs,2,SweepAngle=5)
cp = uavCoveragePlanner(cs,Solver="Exhaustive");
```

Set the takeoff position to a location in the courtyard, then plan the coverage path.

```
takeoff = [42.30089 -71.3752, 0];
[wp,soln] = plan(cp,takeoff);
hold on
geoplot(wp(:,1),wp(:,2),LineWidth=1.5);
geoplot(takeoff(1),takeoff(2),MarkerSize=25,Marker=".")
```

```
legend("","","Path","Takeoff/Landing")
hold off
```



Input Arguments

space — Coverage space
`uavCoverageSpace` object

Coverage space, specified as a `uavCoverageSpace` object. `space` must contain at least one polygon.

index — Polygon index
integer in range [1, *N*]

Polygon index, specified as an integer in the range [1, *N*]. *N* is the total number of polygons in `space`.

Data Types: `single` | `double`

angle — Desired sweep angle for polygon
numeric scalar in range [-180, 180]

Desired sweep angle for the polygon, specified as a numeric scalar in range [-180, 180], in degrees.

Data Types: `single` | `double`

Version History

Introduced in R2023a

See Also

`uavCoverageSpace`

Topics

"Optimally Survey and Customize Coverage of Region of Interest using Coverage Planner"

show

Visualize 2D coverage space

Syntax

```
ax = show(space)
ax = show(space,Name=Value)
```

Description

`ax = show(space)` visualizes the polygons in coverage space `space` with numbering corresponding to the order that the polygons were specified at the time of creation of `space`. You can also use the `show` function for showing coverage space polygons that were decomposed with the `coverageDecomposition` function.

`ax = show(space,Name=Value)` specifies additional options using one or more name-value arguments.

Examples

Plan Coverage Path Using Geodetic Coordinates

This example shows how to plan a coverage path that surveys the parking lots of the MathWorks Lakeside campus.

Get the geodetic coordinates for the MathWorks Lakeside campus. Then create the limits for our map.

```
mwLS = [42.3013 -71.375 0];
latlim = [mwLS(1)-0.003 mwLS(1)+0.003];
lonlim = [mwLS(2)-0.003 mwLS(2)+0.003];
```

Create a figure containing the map with the longitude and latitude limits.

```
fig = figure;
g = geoaxes(fig,Basemap="satellite");
geolimits(latlim,lonlim)
```

Get the outline of the first parking lot in longitude and latitude coordinates. Then create the polygon by concatenating them.

```
pl1lat = [42.3028 42.30325 42.3027 42.3017 42.3019]';
pl1lon = [-71.37527 -71.37442 -71.3736 -71.37378 -71.375234]';
pl1Poly = [pl1lat pl1lon];
```

Repeat the process for the second parking lot.

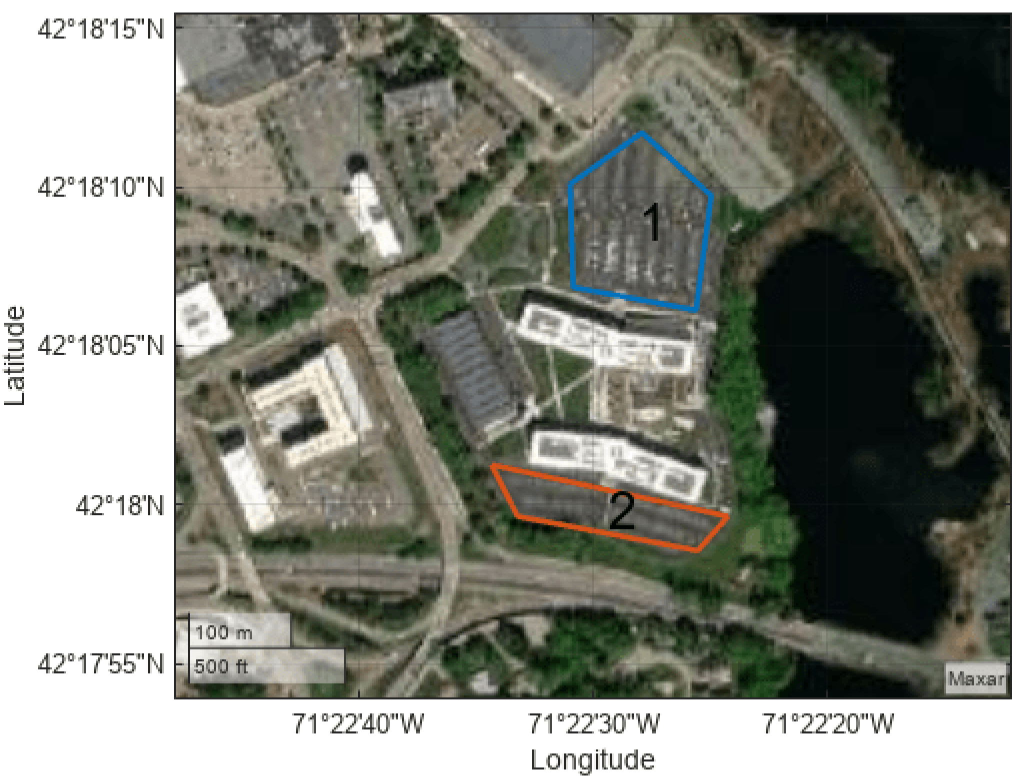
```
pl2lat = [42.30035 42.2999 42.2996 42.2999]';
pl2lon = [-71.3762 -71.3734 -71.37376 -71.37589]';
pl2poly = [pl2lat pl2lon];
```

Create the coverage space with both of those polygons, set the coverage space to use geodetic coordinates, and set the reference location to the MathWorks Lakeside campus location.

```
cs = uavCoverageSpace(Polygons={pl1Poly,pl2poly},UseLocalCoordinates=false,ReferenceLocation=mwLS
```

Set the height at which to fly the UAV to 25 meters, and the sensor footprint width to 20 meters. Then show the coverage space on the map.

```
ReferenceHeight = 25;
cs.UnitWidth = 20;
show(cs,Parent=g);
```



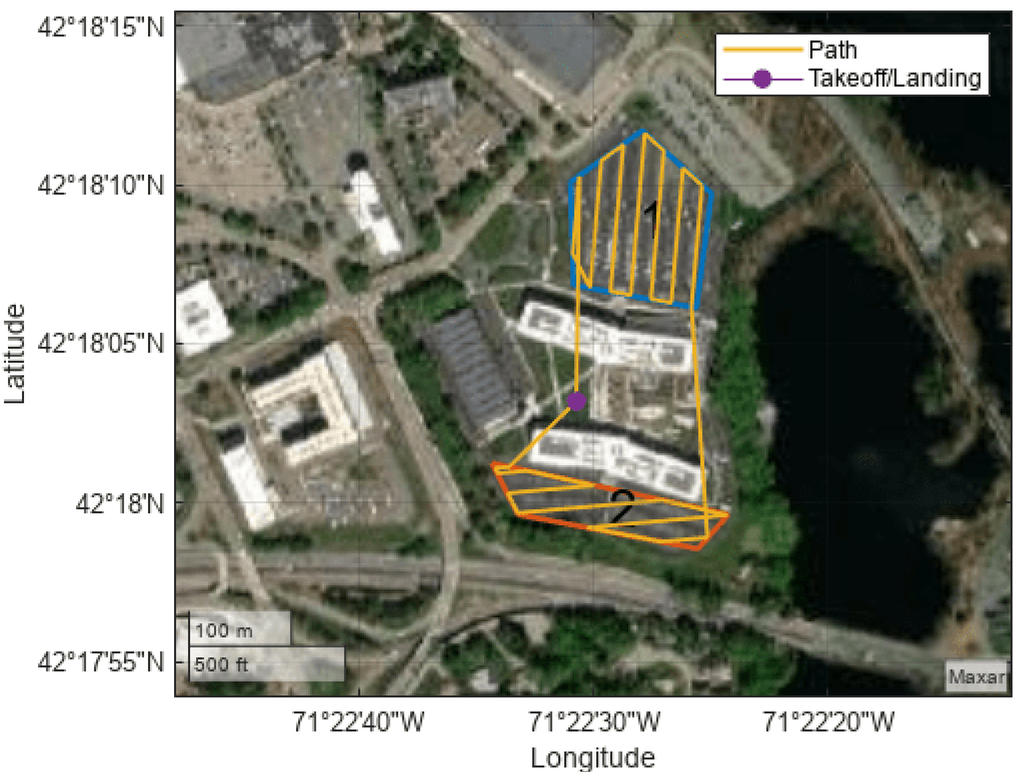
Set the sweep angle for polygons 1 and 2 to 85 and 5 degrees, respectively, to have paths that are parallel to the roads in the parking lots. Then create the coverage planner for that coverage space with the exhaustive solver algorithm.

```
setCoveragePattern(cs,1,SweepAngle=85)
setCoveragePattern(cs,2,SweepAngle=5)
cp = uavCoveragePlanner(cs,Solver="Exhaustive");
```

Set the takeoff position to a location in the courtyard, then plan the coverage path.

```
takeoff = [42.30089 -71.3752, 0];
[wp,soln] = plan(cp,takeoff);
hold on
geoplot(wp(:,1),wp(:,2),LineWidth=1.5);
geoplot(takeoff(1),takeoff(2),MarkerSize=25,Marker=".")
```

```
legend("","","Path","Takeoff/Landing")
hold off
```



Input Arguments

space — Coverage space
uavCoverageSpace object

Coverage space, specified as an `uavCoverageSpace` object.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Example: `show(cs,FontSize=5)`

Parent — Parent axes for plotting
Axes object (default) | GeographicAxes object

Parent axes for plotting, specified as either an `Axes` object or a `GeographicAxes` object.

If the value of the `UseLocalCoordinates` property of `space` is `true`, then the parent axes must be a `Axes` object. If the value is `false`, then the parent axes must be a `GeographicAxes` object.

`FontSize` — Font size of polygon numbering
`20` (default) | positive numeric scalar

Font size of polygon numbering, specified as a positive numeric scalar. Units are in points, where 1 point = 1/72 of an inch.

`LineWidth` — Width of polygon outlines
2 (default) | positive numeric scalar

Width of polygon outlines, specified as a positive numeric scalar. Units are in points, where 1 point = 1/72 of an inch.

Output Arguments

ax — Axes handle
`Axes` object | `GeographicAxes` object

Axes handle, returned as an `Axes` object or a `GeographicAxes` object.

Version History

Introduced in R2023a

See Also

`uavCoverageSpace` | `geoaxes` | GeographicAxes Properties

Topics

"Optimally Survey and Customize Coverage of Region of Interest using Coverage Planner"

interpolate

Interpolate poses along UAV Dubins path segment

Syntax

```
poses = interpolate(pathSegObj,lengths)
```

Description

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

Examples

Interpolate Poses for UAV Dubins Path

This example shows how to connect poses using the `uavDubinsConnection` object and interpolate the poses along the path segment at the specified path lengths.

Connect Poses Using UAV Dubins Connection Path

Create a `uavDubinsConnection` object.

```
connectionObj = uavDubinsConnection;
```

Define start and goal poses as [`x`, `y`, `z`, `headingAngle`] vectors.

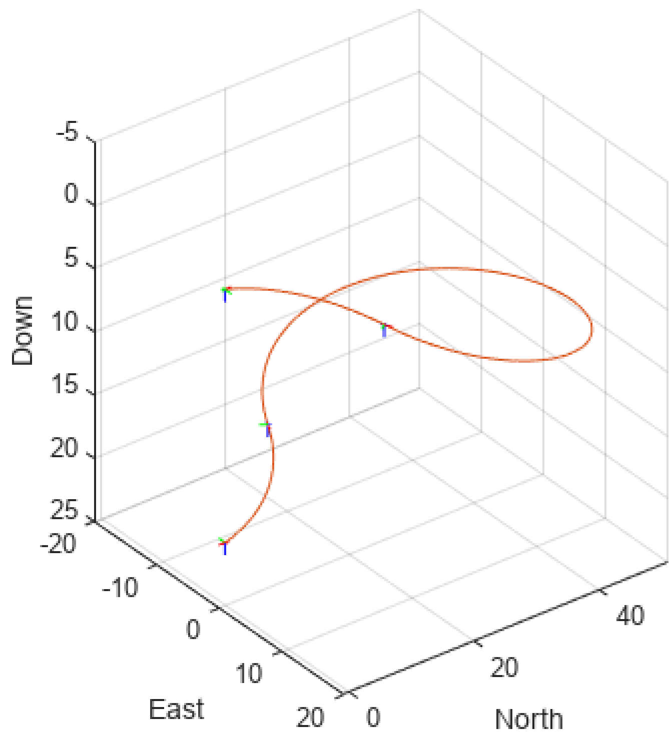
```
startPose = [0 0 0 0]; % [meters, meters, meters, radians]
goalPose = [0 0 20 pi];
```

Calculate a valid path segment and connect the poses.

```
[pathSegObj,pathCost] = connect(connectionObj,startPose,goalPose);
```

Show the generated path.

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



Interpolate the Poses

Specify the interval to interpolate along the path.

```
stepSize = pathSegObj{1}.Length/10;
lengths = 0:stepSize:pathSegObj{1}.Length;
```

Interpolate the poses along the path segment at the specified path lengths.

```
poses = interpolate(pathSegObj{1},lengths); % [x, y, z, headingAngle, flightPathAngle, rollAngle]
```

Visualize the Transition Poses

Compute the translation and rotation matrix of the transition poses, excluding the start and goal poses. The `posesTranslation` matrix consists of the first three columns of the `poses` matrix specifying the position `x`, `y`, and `z`.

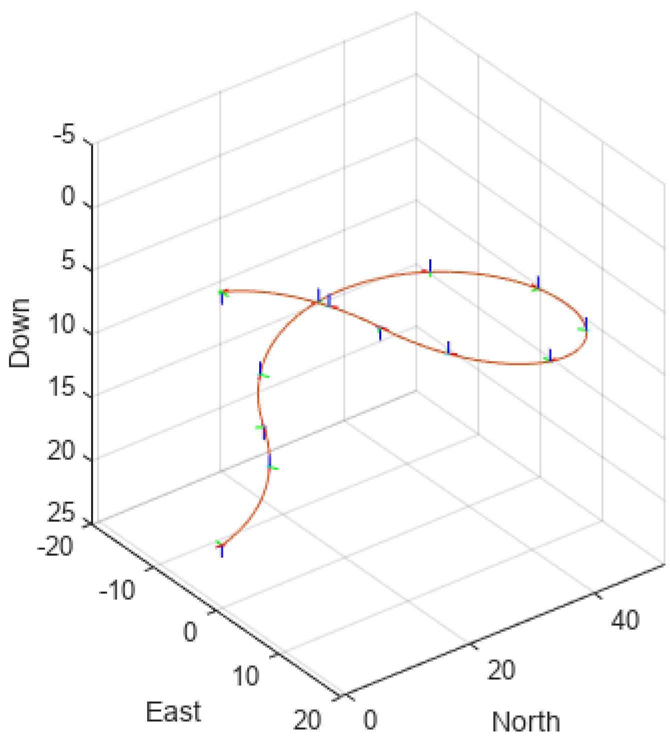
```
posesTranslation = poses(2:end-1,1:3); % [x, y, z]
```

Increment the elements of the fourth column of the `poses` matrix representing the `headingAngle` by `pi` and assign it as the first column of the rotation matrix `posesEulRot` in ZYX Euler angle representation. A column of `pi` and a column of zeros forms the second and the third columns of the `posesEulRot` matrix, respectively. Convert the `posesEulRot` matrix from Euler angles to quaternion and assign to `posesRotation`.

```
N = size(poses,1)-2;
posesEulRot = [poses(2:end-1,4)+pi,ones(N,1)*pi,zeros(N,1)]; % [headingAngle + pi, pi, 0]
posesRotation = quaternion(eul2quat(posesEulRot,'ZYX'));
```

Plot transform frame of the transition poses by specifying their translations and rotations using `plotTransforms`.

```
hold on
plotTransforms(posesTranslation,posesRotation,'MeshFilePath','fixedwing.stl','MeshColor','cyan')
```



Input Arguments

pathSegObj — Path segment
uavDubinsPathSegment object

Path segment, specified as a `uavDubinsPathSegment` object.

lengths — Lengths along path to interpolate poses
positive numeric vector

Lengths along path to interpolate poses, specified as a positive numeric vector in meters.

For example, specify `0:`*`stepSize`*`:pathSegObj{1}.Length` to interpolate at the interval specified by *stepSize* along the path. Transitions between motion types are always included.

Data Types: `double`

Output Arguments

poses — Interpolated poses
six-element numeric matrix

Interpolated poses along the path segment, returned as a six-element numeric matrix [*x*, *y*, *z*, *headingAngle*, *flightPathAngle*, *rollAngle*]. Each row of the matrix corresponds to a different interpolated pose along the path.

x, *y*, and *z* specify the position in meters. *headingAngle*, *flightPathAngle*, and *rollAngle* specify the orientation in radians.

Version History

Introduced in R2019b

Extended Capabilities

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

See Also

`uavDubinsPathSegment` | `show`

show

Visualize UAV Dubins path segment

Syntax

```
axHandle = show(pathSegObj)
axHandle = show(pathSegObj,Name,Value)
```

Description

`axHandle = show(pathSegObj)` plots the path segment with start and goal positions and the transitions between the motion types.

Note Plotting uses only the position and the yaw angle.

`axHandle = show(pathSegObj,Name,Value)` specifies additional name-value pair arguments to control display settings.

Examples

Connect Poses Using UAV Dubins Connection Path

This example shows how to calculate a UAV Dubins path segment and connect poses using the `uavDubinsConnection` object.

Create a `uavDubinsConnection` object.

```
connectionObj = uavDubinsConnection;
```

Define start and goal poses as [`x`, `y`, `z`, `headingAngle`] vectors.

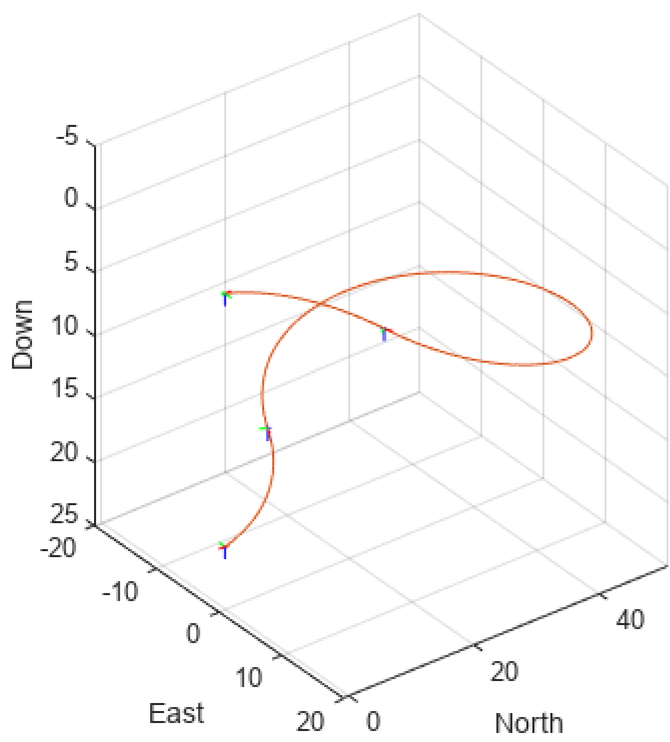
```
startPose = [0 0 0 0]; % [meters, meters, meters, radians]
goalPose = [0 0 20 pi];
```

Calculate a valid path segment and connect the poses. Returns a path segment object with the lowest path cost.

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

Show the generated path.

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



Display the motion type and the path cost of the generated path.

```
fprintf('Motion Type: %s\nPath Cost: %f\n',strjoin(pathSegObj{1}.MotionTypes),pathCosts);
```

```
Motion Type: R L R N
Path Cost: 138.373157
```

Input Arguments

pathSegObj — Path segment
uavDubinsPathSegment object

Path segment, specified as a `uavDubinsPathSegment` object.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `'Positions',{'start','goal'}`

Parent — Axes used to plot path
Axes object

Axes used to plot path, specified as the comma-separated pair consisting of `'Parent'` and an `axes` object.

Example: `'Parent',axHandle`

`Positions` — Positions to display
`{'start','goal','transitions'}` (default) | cell array of string or character vectors or vector of string scalars

Positions to display, specified as the comma-separated pair consisting of `'Positions'` and a cell array of string or character vectors or a vector of string scalars.

Options are any combination of `'start'`, `'goal'`, and `'transitions'`.

To disable all position displays, specify either as an empty cell array `{}` or empty vector `[]`.

Output Arguments

`axHandle` — Axes used to plot path
`Axes` object

Axes used to plot path, returned as an `axes` object.

Version History

Introduced in R2019b

See Also

`uavDubinsPathSegment` | `plotTransforms`

addGeoFence

Add geographical fencing to UAV platform

Syntax

```
addGeoFence(platform,type,geometries,permission)
addGeoFence( ___ ,Name,Value)
```

Description

`addGeoFence(platform,type,geometries,permission)` adds a geofence specified in ENU coordinates to the scenario.

`addGeoFence( ___ ,Name,Value)` specifies options using one or more name-value pair arguments in addition to the input arguments in the previous syntax. For example, `'UseLatLon',true` uses latitude and longitude coordinates for the *xy*-coordinates of the `geometries` input.

Input Arguments

`platform` — UAV platform
`uavPlatform` object

UAV platform in a scenario, specified as a `uavPlatform` object.

`type` — Type of mesh
`"cylinder"` | `"polygon"`

Type of mesh, specified as `"cylinder"` or `"polygon"`.

Data Types: `char` | `string`

`geometries` — Geometric parameters of mesh
cell array

Geometric parameters of the mesh, specified as a cell array with options that depend on the `type` input:

Geometry Parameters

type Input	Geometry Parameters	Description
`"cylinder"`	`{[x y height]}`	Three-element vector of the *xy*-position and height of the cylinder.
`"polygon"`	`{[endptsX endptsY] [zmin zmax]}`	End points of the polygon, specified in either clockwise or counterclockwise order. *z*-coordinates specify the minimum and maximum elevation of the polygon.

`permission` — Geofence permission
`false` or `0` | `true` or `1`

Geofence permission, specified as a `0` (`false`) or `1` (`true`), which indicates whether the UAV platform is permitted inside the geofence (`true`) or not permitted (`false`).

Data Types: `logical`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `'UseLatLon',true` uses latitude and longitude coordinates for the *xy*-coordinates of the `geometries` input.

`UseLatLon` — Use latitude-longitude coordinates for geofence geometry
`false` or `0` | `true` or `1`

Use latitude-longitude coordinates for the geofence geometry, specified as the comma-separated pair consisting of `'UseLatLon'` and a logical `0` (`false`) or `1`(`true`).

Data Types: `logical`

`ReferenceFrame` — Reference frame for computing UAV platform motion
string scalar

Reference frame for computing UAV platform motion, specified as the comma-separated pair consisting of `'ReferenceFrame'` and a string scalar, which matches any reference frame in the `uavScenario`.

Data Types: `char` | `string`

Version History

Introduced in R2020b

See Also

Functions
`move` | `read` | `updateMesh` | `checkPermission`

Objects
`uavScenario` | `uavPlatform` | `uavSensor`

Topics
"UAV Scenario Tutorial"

checkPermission

Check UAV platform permission based on geofencing

Syntax

```
permission = checkPermission(platform)
permission = checkPermission(platform,position)
permission = checkPermission(platform,position,Name,Value)
```

Description

`permission = checkPermission(platform)` checks whether the current UAV platform position is permitted according to the geofences.

`permission = checkPermission(platform,position)` checks whether a specific position in the scenario inertial frame is permitted.

`permission = checkPermission(platform,position,Name,Value)` specifies options using one or more name-value pair arguments. For example, `'UseLatLon',true` uses latitude, longitude, and altitude coordinates for the `positions` input.

Input Arguments

`platform` — UAV platform
`uavPlatform` object

UAV platform in a scenario, specified as a `uavPlatform` object.

`position` — UAV platform position in scenario inertial frame
`[0 0 0]` (default) | vector of the form `[x y z]`

UAV platform position in the scenario inertial frame, specified as a vector of the form `[x y z]`.

Data Types: `double`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `'UseLatLon',true`uses latitude, longitude, and altitude coordinates for the `positions` input.

`UseLatLon` — Use latitude, longitude, and altitude coordinates for platform position
`0` or `false` (default) | `1` or `true`

Use latitude, longitude, and altitude coordinates for platform position, specified as the comma-separated pair `'UseLatLon'` and a logical `0` (`false`) or `1`(`true`).

Data Types: `logical`

`ReferenceFrame` — Reference frame for computing UAV platform motion
string scalar

Reference frame for computing UAV platform motion, specified as the comma-separated pair consisting of `'ReferenceFrame'` and a string scalar, which matches any reference frame in the `uavScenario`.

Data Types: `char` | `string`

Output Arguments

`permission` — Geofence permission for platform
`false` or `0` | `true` or `1`

Geofence permission for platform, returned as a `0` (`false`) or `1` (`true`), which indicates whether the UAV platform is permitted inside the geofence (`true`) or not permitted (`false`).

Data Types: `logical`

Version History

Introduced in R2020b

See Also

Functions
`move` | `read` | `updateMesh` | `addGeoFence`

Objects
`uavScenario` | `uavPlatform` | `uavSensor`

Topics
"UAV Scenario Tutorial"

move

Move UAV platform in scenario

Syntax

`move(platform,motion)`

Description

`move(platform,motion)` moves the UAV platform in the scenario according to the specified motion `motion`.

Input Arguments

`platform` — UAV platform
`uavPlatform` object

UAV platform in a scenario, specified as a `uavPlatform` object.

`motion` — UAV platform motion at current instance in scenario
16-element vector

UAV platform motion at the current instance in a UAV scenario, specified as a 16-element vector with these elements in order:

- `[x y z]` — Positions in *xyz*-axes in meters
- `[vx vy vz]` — Velocities in *xyz*-directions in meters per second
- `[ax ay az]` — Accelerations in *xyz*-directions in meters per second
- `[qw qx qy qz]` — Quaternion vector for orientation
- `[wx wy wz]` — Angular velocities in radians per second

Data Types: `double`

Version History

Introduced in R2020b

See Also

Functions
`read` | `updateMesh` | `addGeoFence` | `checkPermission`

Objects
`uavScenario` | `uavPlatform` | `uavSensor`

Topics
“UAV Scenario Tutorial”

read

Read UAV motion vector

Syntax

```
[motion,LLA] = read(platform)
```

Description

`[motion,LLA] = read(platform)` reads the latest motion of the UAV platform in the scenario.

Input Arguments

`platform` — UAV platform
`uavPlatform` object

UAV platform in a scenario, specified as a `uavPlatform` object.

Output Arguments

`motion` — UAV platform motion at current instance in scenario
16-element vector

UAV platform motion at the current instance in a UAV scenario, returned as a 16-element vector with these elements in order:

- `[x y z]` — Positions in *xyz*-axes in meters
- `[vx vy vz]` — Velocities in *xyz*-directions in meters per second
- `[ax ay az]` — Accelerations in *xyz*-directions in meters per second
- `[qw qx qy qz]` — Quaternion vector for orientation
- `[wx wy wz]` — Angular velocities in radians per second

Data Types: `double`

`LLA` — Latitude, longitude, and altitude coordinates of UAV platform
three-element vector of the form `[lat long alt]`

Latitude, longitude, and altitude coordinates of the UAV platform at the current instance in a UAV scenario, returned as a three-element vector of the form `[lat long alt]`.

Data Types: `double`

Version History

Introduced in R2020b

See Also

Functions
`move` | `updateMesh` | `addGeoFence` | `checkPermission`

Objects
`uavScenario` | `uavPlatform` | `uavSensor`

Topics
"UAV Scenario Tutorial"

updateMesh

Update body mesh for UAV platform

Syntax

```
updateMesh(platform,type,geometries,color,position,orientation)
updateMesh(platform,type,geometries,color,offset)
```

Description

`updateMesh(platform,type,geometries,color,position,orientation)` updates the body mesh of the UAV platform with the specified mesh type, geometry, color, position, and orientation.

`updateMesh(platform,type,geometries,color,offset)` specifies the relative mesh frame position and orientation as a homogeneous transformation matrix `offset`.

Input Arguments

`platform` — UAV platform
`uavPlatform` object

UAV platform in a scenario, specified as a `uavPlatform` object.

`type` — Type of mesh
`"fixedwing"` | `"quadrotor"` | `"cuboid"` | `"custom"`

Type of mesh, specified as `"fixedwing"`, `"quadrotor"`, `"cuboid"`, or `"custom"`.

Data Types: `string` | `char`

`geometries` — Geometric parameters of mesh
cell array

Geometric parameters of the mesh, specified as a cell array with options that depend on the `type` input:

Geometry Parameters

input Type	Geometry Parameters	Description
"fixedwing"	{scale}	Positive scalar specifying the relative size of the fixed-wing mesh. Scale is unitless.
"quadrotor"	{scale}	Positive scalar specifying the relative size of the multirotor mesh. Scale is unitless.
"cuboid"	{[x y height]}	Three-element vector of the *xy*-position and height of the cuboid, specified in meters.
"custom"	{vertices faces}	Vertices and faces that define the mesh as two three-element vectors. Each vertex is a row of [x y z] points in meters. Each face is a row of [a b c] indices of vertex IDs, where a vertex ID is the row number of a vertex in vertices.

color — UAV platform body mesh color
RGB triplet

UAV platform body mesh color, specified as an RGB triplet.

Data Types: double

position — Relative mesh position
[0 0 0] (default) | vector of the form [x y z]

Relative mesh position in the body frame, specified as a vector of the form [x y z].

Data Types: double

orientation — Relative mesh orientation
[1 0 0 0] (default) | quaternion vector of the form [w x y z] | quaternion object

Relative mesh orientation, specified as a quaternion vector of the form [w x y z] or a quaternion object.

Data Types: double

offset — Transformation of mesh relative to body frame
4-by-4 homogeneous transformation matrix

Transform of mesh relative to the body frame, specified as a 4-by-4 homogeneous transformation matrix. The matrix maps points in the platform mesh frame to points in the body frame.

Data Types: double

Version History

Introduced in R2020b

See Also

Functions
`move` | `read` | `addGeoFence` | `checkPermission`

Objects
`uavScenario` | `uavPlatform` | `uavSensor`

Topics
"UAV Scenario Tutorial"

addInertialFrame

Define new inertial frame in UAV scenario

Syntax

```
addInertialFrame(scene,base,name,position,orientation)
addInertialFrame(scene,base,name,transformMatrix)
```

Description

`addInertialFrame(scene,base,name,position,orientation)` adds a new inertial frame to the UAV scenario `scene` by specifying the base, name, position, and orientation of the new inertial frame.

`addInertialFrame(scene,base,name,transformMatrix)` adds a new inertial frame to the UAV scenario `scene` by specifying the base, name, and transformation matrix of the new inertial frame.

Examples

Add an Inertial Frame to UAV Scenario

Create a UAV scenario. By default, the inertial frames are the ENU and the NED frames.

```
scene = uavScenario()
```

```
scene = 
  uavScenario with properties:

           UpdateRate: 10
             StopTime: Inf
    HistoryBufferSize: 100
    ReferenceLocation: [0 0 0]
         MaxNumFrames: 10
          CurrentTime: 0
            IsRunning: 0
        TransformTree: [1x1 transformTree]
       InertialFrames: ["ENU"    "NED"]
               Meshes: {}
            Platforms: [0x0 uavPlatform]
```

Add a new inertial frame named `Map` to the scenario.

```
addInertialFrame(scene,"NED","Map",[100 100 100],[1 0 0 0]);
```

You can now use the `Map` frame as a reference frame to define other objects in the scenario.

```
scene.InertialFrames(3)
```

```
ans = 
"Map"
```

Input Arguments

`scene` — UAV scenario
`uavScenario` object

UAV scenario, specified as a `uavScenario` object.

`base` — Base of new inertial frame
string scalar

Base of the new inertial frame, specified as a string scalar. The base frame must be defined in the scenario in advance.

Example: `"ENU"`

`name` — Name of new inertial frame
string scalar

Name of the new inertial frame, specified as a string scalar.

Example: `"newFrame"`

`position` — Position of new inertial frame
1-by-3 vector of scalar

Position of the new inertial frame with respect to the base frame (specified in the `base` argument), specified as a 1-by-3 vector of scalars in meters.

`orientation` — Orientation of new inertial frame
quaternion | 1-by-4 quaternion vector of scalar

Orientation of the new inertial frame with respect to the base frame (specified in the `base` argument), specified as a `quaternion` or a 1-by-4 quaternion vector of scalars. The specified orientation is from the base frame to the new inertial frame.

`transformMatrix` — Transformation matrix of new inertial frame
4-by-4 homogeneous transform matrix

Transformation matrix that maps points in the new frame (specified in the `base` argument) to the base frame, specified as a 4-by-4 homogeneous transform matrix that maps points in the base frame to the new inertial frame.

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

Version History

Introduced in R2020b

addMesh

Add new static mesh to UAV scenario

Syntax

```
addMesh(scene,type,geometry,color)
addMesh( ___ ,Name=Value)
info = addMesh( ___ ,Verbose=true)
```

Description

`addMesh(scene,type,geometry,color)` adds a static mesh, or imports meshes from an OSM file, to the UAV scenario `scene` by specifying the mesh type, geometry, and color.

`addMesh( ___ ,Name=Value)` specifies additional options using name-value arguments.

`info = addMesh( ___ ,Verbose=true)` returns information, when importing OSM or terrain files, about the file import.

Examples

Add Meshes to UAV Scenario

Create a UAV Scenario.

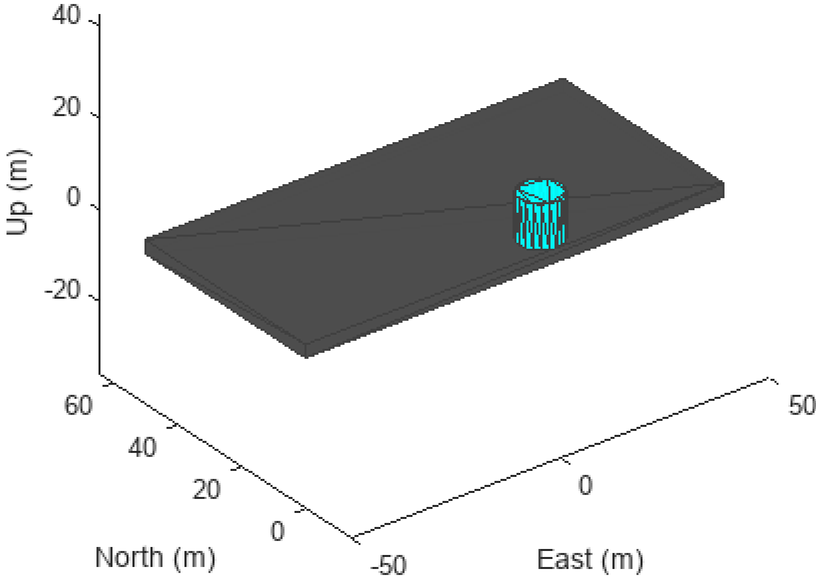
```
scene = uavScenario("UpdateRate",100,"StopTime",1);
```

Add the ground and a building as meshes.

```
addMesh(scene,"Polygon", {[-50 0; 50 0; 50 50; -50 50], [-3 0]}, [0.3 0.3 0.3]);
addMesh(scene,"Cylinder", {[10 5 5], [0 10]}, [0 1 1]);
```

Visualize the scenario.

```
show3D(scene);
```



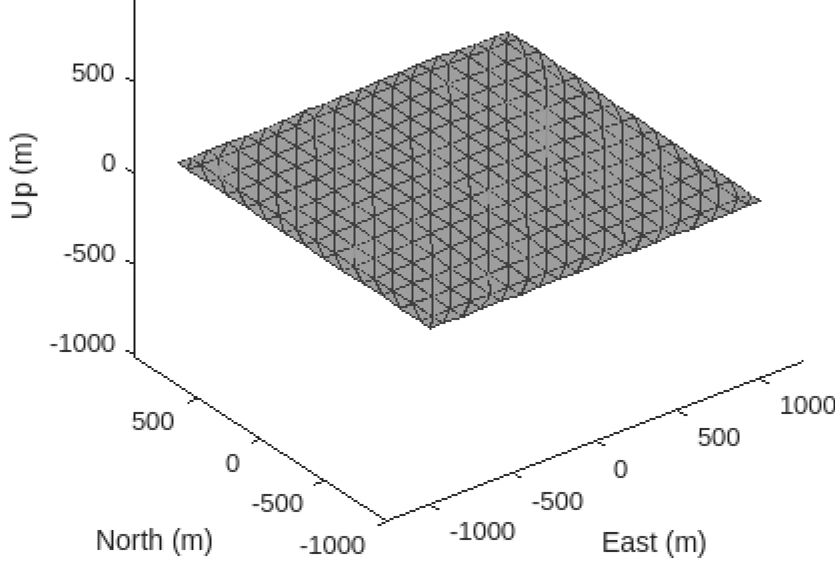
Import OSM Building Meshes and Terrain Mesh into UAV Scenario

Create a UAV scenario centered on New York City, and add a terrain mesh based on the Global Multi-Resolution Terrain Elevation Data (GMTED2010) data set.

```
scene = uavScenario(ReferenceLocation=[40.707088 -74.012146 0]);
xlimits = [-1000 1000];
ylimits = [-1000 1000];
color = [0.6 0.6 0.6];
terrainInfo = addMesh(scene,"terrain",{"gmted2010",xlimits,ylimits},color,Verbose=true)
```

```
terrainInfo = struct with fields:
       TerrainName: "gmted2010"
     LatitudeRange: [-90 90]
    LongitudeRange: [-180 180]
```

```
show3D(scene);
```

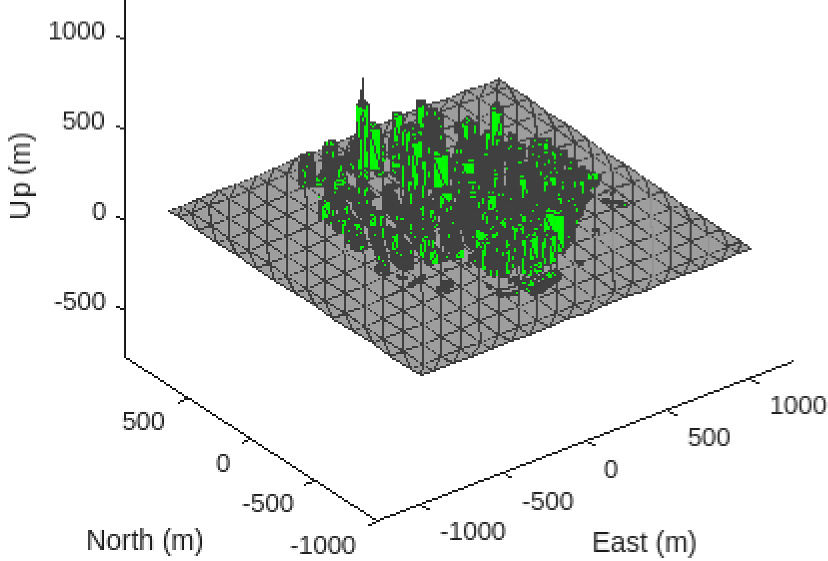


Add buildings by importing them from an OSM file that contains the buildings of Manhattan, `manhattan.osm`.[1] on page 2-385

```
xlimits = [-800 800];
ylimits = [-800 800];
color = [0 1 0];
osmInfo = addMesh(scene,"buildings",{"manhattan.osm",xlimits,ylimits,"auto"},color,Verbose=true)
```

```
osmInfo = struct with fields:
             OSMFileName: "/mathworks/devel/sbs/25/rprasad.Bdoc23a.j2033780.Aug12/matlab/toolbox/
           LatitudeRange: [40.7010 40.7182]
          LongitudeRange: [-74.0188 -74.0003]
       TotalNumBuildings: 836
    NumImportedBuildings: 657
```

```
show3D(scene);
```



[1] The file was downloaded from `https://www.openstreetmap.org`, which provides access to crowd-sourced map data all over the world. The data is licensed under the Open Data Commons Open Database License (ODbL), `https://opendatacommons.org/licenses/odbl/`.

Input Arguments

`scene` — UAV scenario
`uavScenario` object

UAV scenario, specified as a `uavScenario` object.

`type` — Mesh type
`"cylinder"` | `"surface"` | `"terrain"` | `"polygon"` | `"buildings"` | `"custom"`

Mesh type, specified as `"cylinder"`, `"surface"`, `"terrain"`, `"polygon"`, `"buildings"`, or`"custom"`. Specify the geometric parameters of the mesh using the `geometry` input.

Data Types: `string` | `char`

`geometry` — Mesh geometry
cell array

Mesh geometry, specified as a cell array of geometry parameters. Depending on the `type` input, the geometry parameters have different forms:

type Input Argument	Geometry Parameters	Description
"cylinder"	{[centerx, centery, radius],[zmin, zmax]}	centerx and centery are the x- and y-coordinates of the center of the cylinder, respectively. radius is the radius of the cylinder in meters. zmin and zmax are the minimum and maximum z-axis coordinates of the cylinder in meters, respectively.
"surface"	{meshGridX,meshGridY,z}	meshGridX, meshGridY and z are all 2-D matrices of the same size that define the *xyz*-points of the surface mesh.
"terrain"	{terrainName,XLimits,YLimits}	You must first call the addCustomTerrain function to load the terrain data and specify a terrain name. Specify the minimum and maximum *xy*-limits as two separate two-element vectors in local coordinates, or latitude-longitude coordinates if the 'UseLatLon' name-value pair is true. The *xy*-coordinates must be specified in the ENU reference frame.
"polygon"	{cornerPoints,[zmin, zmax]}	zmin and zmax are the minimum and maximum z-axis coordinates of the polygon in meters, respectively. cornerPoints contains the corner points of the polygon, specified as a *N*-by-2 matrix, where *N* is the number of corner points. The first column contains the x-coordinates and the second column contains the y-coordinates in meters.

type Input Argument	Geometry Parameters	Description
"buildings"	{osmFile,xBound,yBound,altitude}	• osmFile — File name of the OSM file in the current folder or on the MATLAB path, or the full or relative path to the OSM file, specified as a character vector or string scalar. • xBound — *x*-axis boundaries of the imported OSM buildings, specified as a two-element row vector in meters. • yBound — *y*-axis boundaries of the imported OSM buildings, specified as a two-element row vector in meters. • altitude — Height of the bases for all imported buildings, specified as a scalar or "auto". If specified as "auto", the base heights of the buildings are defined by the height of the terrain in the scene. If the scene has no terrain, this value is 0.
"custom"	{vertices,faces}	vertices is an *n*-by-3 matrix of mesh points in local coordinates. faces is an *n*-by-3 integer matrix of indexes indicating the triangular faces of the mesh.

color — Mesh color
RGB triplet

Mesh color, specified as a RGB triplet.

Example: [1 0 0]

Name=Value Pair Arguments

Specify optional pairs of arguments as Name1=Value1,...,NameN=ValueN, where Name is the argument name and Value is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose Name *in quotes.*

Example: addMesh(scene,"Cylinder",{[46 42 5],[0 20]},[0 1 0],UseLatLon=true)

`UseLatLon` — Enable latitude and longitude coordinates
`false` (default) | `true`

Enable latitude and longitude coordinates, specified as `true` or `false`.

- When specified as `true`, the x and y coordinates in the `geometry` input are interpreted as longitude and latitude, respectively. Vertices for custom mesh use east-north-up (ENU) as the reference frame.
- When specified as `false`, the x and y coordinates in the `geometry` input are interpreted as Cartesian coordinates.

`ReferenceFrame` — Reference frame of geometry input
`"ENU"` (default) | name of defined inertial frame

Reference frame of the geometry input, specified as an inertial frame name defined in the `InertialFrames` property of the `uavScenario` object `scene`. You can add new inertial frames to the scenario using the `addInertialFrame` object function.

The scenario only accepts frames that have *z*-axis rotation with respect to the `"ENU"` frame.

For terrain and building import, the reference frame must be `"ENU"`.

`Verbose` — Verbose mode for OSM building or terrain file imports
`false` or `0` (default) | `true` or `1`

Verbose mode for OSM building or terrain file imports, specified as a logical `true` (`1`) or `false` (`0`). When you specify this argument as `true`, `addMesh` returns information about the file and import process. The information returned depends on the type of file.

If you specify an output argument, verbose mode returns the import information as a structure, `info`. Otherwise, verbose mode prints the import information in the Command Window.

Output Arguments

`info` — File import information
structure

File import information, returned as a structure. The fields contained depend on the type of file:

File Type	Fields
OSM Building	• `OSMFileName` — OSM file name as a string scalar • `LatitudeRange` — Latitude range as a two-element row vector • `LongitudeRange` — Longitude range as a two-element row vector • `TotalNumBuildings` — Total number of buildings in the OSM file as an integer • `NumImportedBuildings` — Number of buildings imported from the OSM file as an integer

File Type	Fields
Terrain	• `TerrainName` — Terrain file name as a string scalar • `LatitudeRange` — Latitude range as a two-element row vector • `LongitudeRange` — Longitude range as a two-element row vector

To return this output argument, you must specify the `Verbose` name-value argument as `true`.

Tips

- OSM files may load slowly when a high number of buildings specified in the file. If you experience long load times, consider reducing the number of buildings to load more quickly.

Version History

Introduced in R2020b

See Also

`uavScenario` | `addCustomTerrain` | `removeCustomTerrain` | `terrainHeight`

advance

Advance UAV scenario simulation by one time step

Syntax

```
isrunning = advance(scene)
```

Description

`isrunning = advance(scene)` advances the UAV scenario simulation `scene` by one time step. The `UpdateRate` property of the `uavScenario` object determines the time step during simulation. The function returns the running status of the simulation. The function only updates a platform location if the platform has an assigned trajectory.

Examples

Simulate Simple UAV Scenario

Create a UAV scenario.

```
scene = uavScenario("UpdateRate",100,"StopTime",1);
```

Add the ground and a building as meshes.

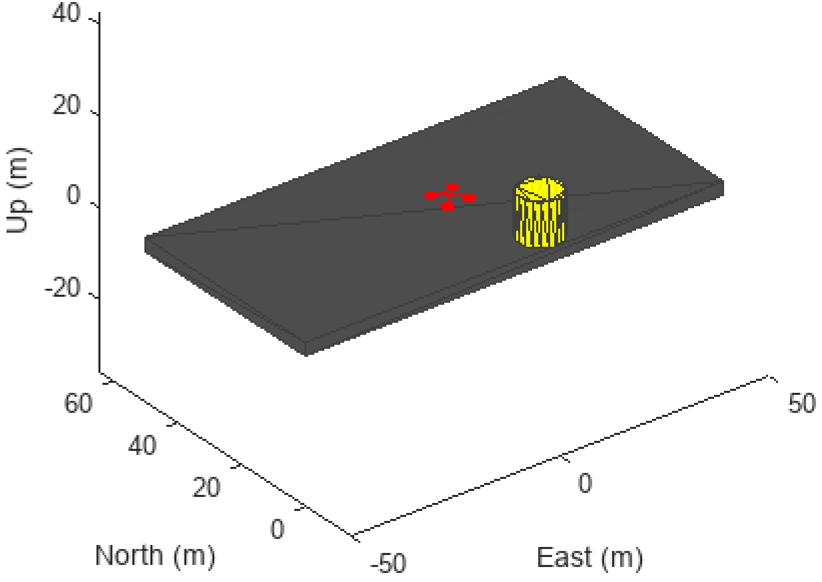
```
addMesh(scene,"Polygon", {[-50 0; 50 0; 50 50; -50 50], [-3 0]}, [0.3 0.3 0.3]);
addMesh(scene,"Cylinder", {[10 5 5], [0 10]}, [1 1 0]);
```

Create a UAV platform with a specified waypoint trajectory in the scenario. Define the mesh for the UAV platform.

```
traj = waypointTrajectory("Waypoints", [0 -20 -5; 20 0 -5], "TimeOfArrival", [0  1]);
uavPlat = uavPlatform("UAV",scene,"Trajectory", traj);
updateMesh(uavPlat,"quadrotor",{10},[1 0 0],eul2tform([0 0 0]));
```

Simulate and visualize the scenario.

```
setup(scene);
while advance(scene)
    show3D(scene);
    drawnow update
end
```




```
restart(scene);
```

Input Arguments

scene — UAV scenario
uavScenario object

UAV scenario, specified as a uavScenario object.

Output Arguments

isrunning — Running state of simulation
true | false

Running state of the simulation, returned as true or false. If isrunning is returned as true, then the simulation is running. If isrunning is returned as false, the simulation has stopped. A simulation stops when the stop time is reached.

Version History

Introduced in R2020b

copy

Copy UAV scenario

Syntax

```
sceneCopy = copy(scene)
```

Description

`sceneCopy = copy(scene)` creates a deep copy of a `uavScenario` object. The copy has the same properties as the original with the exception of properties that store simulation states such as `CurrentTime`.

Input Arguments

scene — UAV scenario
`uavScenario` object

UAV scenario, specified as a `uavScenario` object.

Output Arguments

sceneCopy — Deep copy of UAV scenario
`uavScenario` object

Deep copy of the UAV scenario, returned as a `uavScenario` object. The object is of the same object type, has the same properties, and contains the same platforms, sensors, and meshes as the object specified to `scene`. The `copy` function does not copy simulation states such as `CurrentTime` from `scene`.

Version History

Introduced in R2022a

See Also

`uavScenario`

restart

Reset simulation of UAV scenario

Syntax

```
restart(scene)
```

Description

`restart(scene)` resets the simulation of the UAV scenario `scene`. The function resets platforms' poses and sensor readings to `NaN`, resets the `CurrentTime` property of the scenario to zero, and resets the `IsRunning` property of the scenario to `false`.

Examples

Simulate Simple UAV Scenario

Create a UAV scenario.

```
scene = uavScenario("UpdateRate",100,"StopTime",1);
```

Add the ground and a building as meshes.

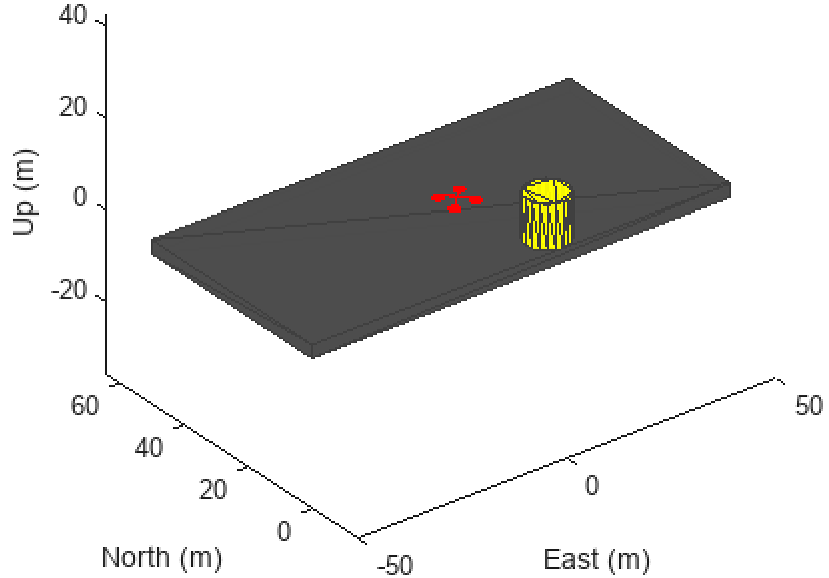
```
addMesh(scene,"Polygon", {[-50 0; 50 0; 50 50; -50 50], [-3 0]}, [0.3 0.3 0.3]);
addMesh(scene,"Cylinder", {[10 5 5], [0 10]}, [1 1 0]);
```

Create a UAV platform with a specified waypoint trajectory in the scenario. Define the mesh for the UAV platform.

```
traj = waypointTrajectory("Waypoints", [0 -20 -5; 20 0 -5], "TimeOfArrival", [0  1]);
uavPlat = uavPlatform("UAV",scene,"Trajectory", traj);
updateMesh(uavPlat,"quadrotor",{10},[1 0 0],eul2tform([0 0 0]));
```

Simulate and visualize the scenario.

```
setup(scene);
while advance(scene)
    show3D(scene);
    drawnow update
end
```




```
restart(scene);
```

Input Arguments

scene — UAV scenario
uavScenario object

UAV scenario, specified as a uavScenario object.

Version History

Introduced in R2020b

setup

Prepare UAV scenario for simulation

Syntax

```
setup(scene)
```

Description

`setup(scene)` prepares the UAV scenario `scene` for simulation and generates initial sensor readings.

Examples

Simulate Simple UAV Scenario

Create a UAV scenario.

```
scene = uavScenario("UpdateRate",100,"StopTime",1);
```

Add the ground and a building as meshes.

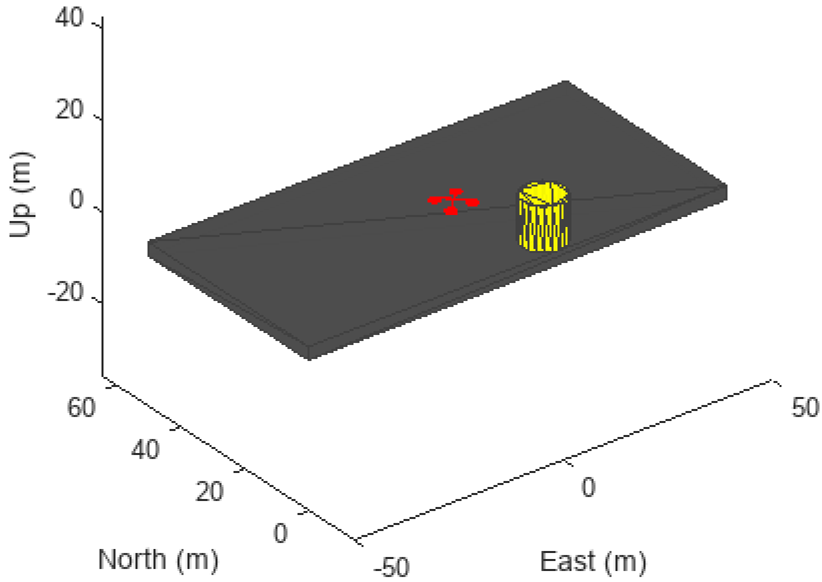
```
addMesh(scene,"Polygon", {[-50 0; 50 0; 50 50; -50 50], [-3 0]}, [0.3 0.3 0.3]);
addMesh(scene,"Cylinder", {[10 5 5], [0 10]}, [1 1 0]);
```

Create a UAV platform with a specified waypoint trajectory in the scenario. Define the mesh for the UAV platform.

```
traj = waypointTrajectory("Waypoints", [0 -20 -5; 20 0 -5], "TimeOfArrival", [0  1]);
uavPlat = uavPlatform("UAV",scene,"Trajectory", traj);
updateMesh(uavPlat,"quadrotor",{10},[1 0 0],eul2tform([0 0 0]));
```

Simulate and visualize the scenario.

```
setup(scene);
while advance(scene)
    show3D(scene);
    drawnow update
end
```




```
restart(scene);
```

Input Arguments

scene — UAV scenario
`uavScenario` object

UAV scenario, specified as a `uavScenario` object.

Version History

Introduced in R2020b

show

Visualize UAV scenario in 2-D

Syntax

```
ax = show(scene)
ax = show(scene,times)
ax = show( ___ ,Name,Value)
```

Description

`ax = show(scene)` visualizes the UAV scenario `scene` in 2-D with latest states of the platforms and returns the axes on which the scenario is plotted.

`ax = show(scene,times)` visualizes the UAV scenario `scene` at timestamps specified by the `times` input.

`ax = show( ___ ,Name,Value)` specifies additional options using Name-Value pairs. Enclose each `Name` in quotes.

Examples

Visualize UAV Scenario in 2D

Create a UAV scenario.

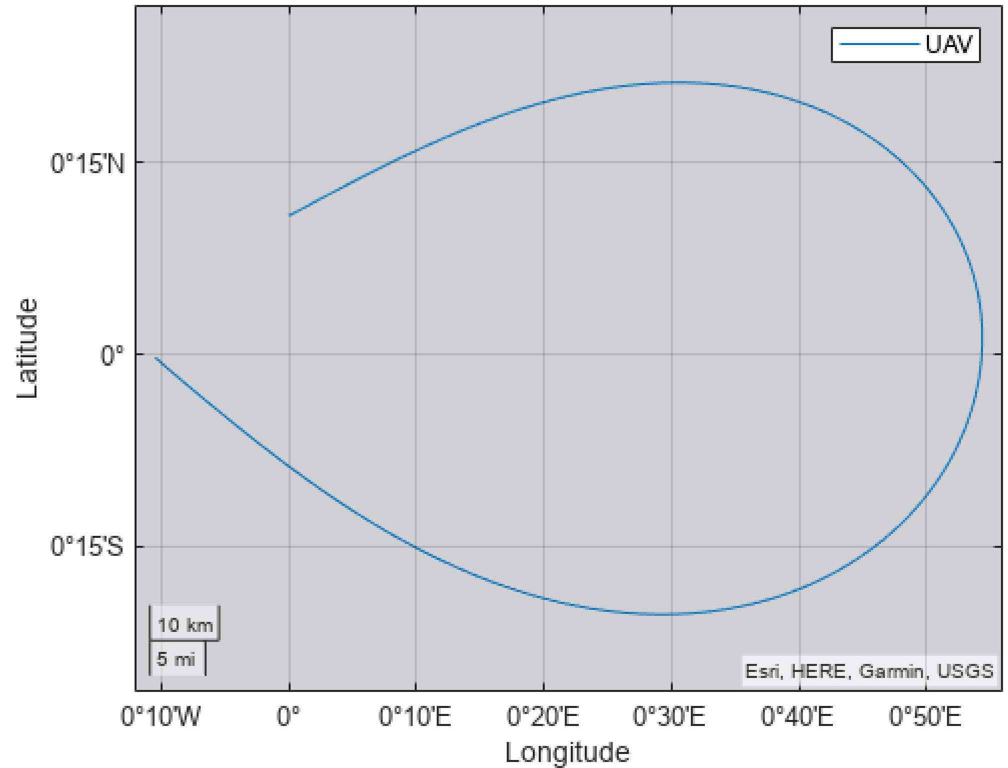
```
scene = uavScenario("UpdateRate",1,"StopTime",1000,"HistoryBufferSize",1000);
```

Create a UAV platform with a specified waypoint trajectory in the scenario.

```
traj = waypointTrajectory("Waypoints", [0 -20000 -50; 10000 100000 -50; 20000 0 -50], "TimeOfArri
uavPlat = uavPlatform("UAV",scene,"Trajectory", traj);
```

Visualize the trajectory in 2D.

```
setup(scene);
while advance(scene)
end
show(scene,0:1:1000)
```




```
ans =
  GeographicAxes with properties:

      Basemap: 'streets-light'
     Position: [0.1300 0.1100 0.7750 0.8150]
        Units: 'normalized'

  Show all properties
```

Input Arguments

scene — UAV scenario
uavScenario object

UAV scenario, specified as a `uavScenario` object.

times — Time stamps
vector of nonnegative scalars

Time stamps at which to show the scenario, specified as a vector of nonnegative scalars. The specified time stamps must be saved in the scenario. To change the number of saved time stamps, use the `HistoryBufferSize` property of the `uavScenario` object.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `ax = show(scene,"MarkerSize",38)`

`Parent` — Parent axes for plotting
`geoaxes`

Parent axes for plotting the scenario, specified as a `geoaxes` object.

`MarkerSize` — Marker size
`36` (default) | positive scalar

Marker size, specified as a positive scalar in points, where 1 point = 1/72 of an inch.

`ShowPlatformName` — Enable showing platform name
`true` (default) | `false`

Enable showing platform name, specified as `true` or `false`.

Output Arguments

ax — Axes on which the scenario is plotted
`geoaxes` object

Axes on which the scenario is plotted, returned as a `geoaxes` object.

Version History

Introduced in R2020b

show3D

Visualize UAV scenario in 3-D

Syntax

```
[ax,plottedFrames] = show3D(scene)
[ax,plottedFrames] = show3D(scene,time)
[ax,plottedFrames] = show3D( ___ ,Name,Value)
```

Description

`[ax,plottedFrames] = show3D(scene)` visualizes latest states of the platforms and sensors in the UAV scenario `scene` along with all static meshes. The function also returns the axes on which the scene is plotted and the frames on which each object is plotted.

`[ax,plottedFrames] = show3D(scene,time)` visualizes the UAV scenario at the specified `time`.

`[ax,plottedFrames] = show3D( ___ ,Name,Value)` specifies additional options using Name-Value pairs. Enclose each `Name` in quotes.

Examples

Create and Simulate UAV Scenario

Create a UAV scenario and set its local origin.

```
scene = uavScenario("UpdateRate",200,"StopTime",2,"ReferenceLocation",[46, 42, 0]);
```

Add an inertial frame called `MAP` to the scenario.

```
scene.addInertialFrame("ENU","MAP",trvec2tform([1 0 0]));
```

Add one ground mesh and two cylindrical obstacle meshes to the scenario.

```
scene.addMesh("Polygon", {[-100 0; 100 0; 100 100; -100 100],[-5 0]},[0.3 0.3 0.3]);
scene.addMesh("Cylinder", {[20 10 10],[0 30]}, [0 1 0]);
scene.addMesh("Cylinder", {[46 42 5],[0 20]}, [0 1 0], "UseLatLon", true);
```

Create a UAV platform with a specified waypoint trajectory in the scenario. Define the mesh for the UAV platform.

```
traj = waypointTrajectory("Waypoints", [0 -20 -5; 20 -20 -5; 20 0 -5],"TimeOfArrival",[0 1 2]);
uavPlat = uavPlatform("UAV",scene,"Trajectory",traj);
updateMesh(uavPlat,"quadrotor", {4}, [1 0 0],eul2tform([0 0 pi]));
addGeoFence(uavPlat,"Polygon", {[-50 0; 50 0; 50 50; -50 50],[0 100]},true,"ReferenceFrame","ENU
```

Attach an INS sensor to the front of the UAV platform.

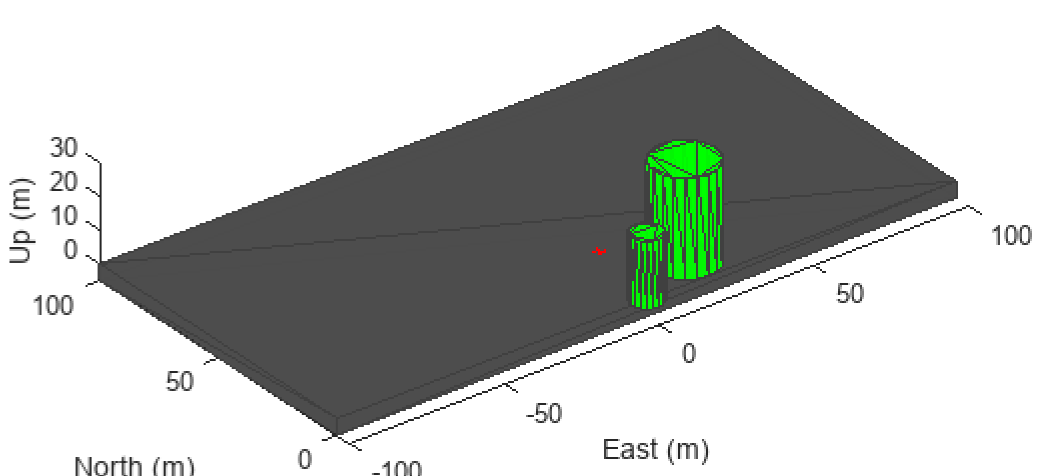
```
insModel = insSensor();
ins = uavSensor("INS",uavPlat,insModel,"MountingLocation",[4 0 0]);
```

Visualize the scenario in 3-D.

```
ax = show3D(scene);
axis(ax,"equal");
```

Simulate the scenario.

```
setup(scene);
while advance(scene)
    % Update sensor readings
    updateSensors(scene);

    % Visualize the scenario
    show3D(scene,"Parent",ax,"FastUpdate",true);
    drawnow limitrate
end
```



Input Arguments

scene — UAV scenario
uavScenario object

UAV scenario, specified as a `uavScenario` object.

time — Time stamp
nonnegative scalar

Time stamp at which to show the scenario, specified as a nonnegative scalar. The time stamp must already be saved in the scenario. To change the number of saved time stamps, use the `HistoryBufferSize` property of the `uavScenario` object, `scene`.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `ax = show3D(scene,"FastUpdate",true)`

`Parent` — Parent axes for plotting
`axes` | `uiaxes`

Parent axes for plotting, specified as an `axes` object or a `uiaxes` object.

`FastUpdate` — Enable updating from previous map
`false` (default) | `true`

Enable updating from previous map, specified as `true` or `false`. When specified as `true`, the function plots the map via a lightweight update to the previous map in the figure. When specified as `false`, the function plots the whole scene on the figure every time.

Output Arguments

`ax` — Axes on which the scenario is plotted
`axes` object | `uiaxes` object

Axes on which the scenario is plotted, returned as an `axes` object or a `uiaxes` object.

`plottedFrames` — Plotted frame information
structure

Plotted frame information, returned as a structure of `hgtransform` objects. The structure has two types of field names:

- Inertial frame names — The corresponding field value is a `hgtransform` object which contains the transform information from the ego frame to the ENU frame.
- UAV platform names — The corresponding field value is a structure which contains the `hgtransform` information for all frames defined on the platform.

Version History
Introduced in R2020b

terrainHeight

Returns terrain height in UAV scenarios

Syntax

```
heights = terrainHeight(scene,x,y)
heights = terrainHeight( ___ ,Name,Value)
```

Description

`heights = terrainHeight(scene,x,y)` returns the terrain heights of the specified *xy*-positions for the terrain data for a `uavScenario` object.

`heights = terrainHeight( ___ ,Name,Value)` specifies additional options using name-value arguments. Enclose each `Name` in quotes.

Examples

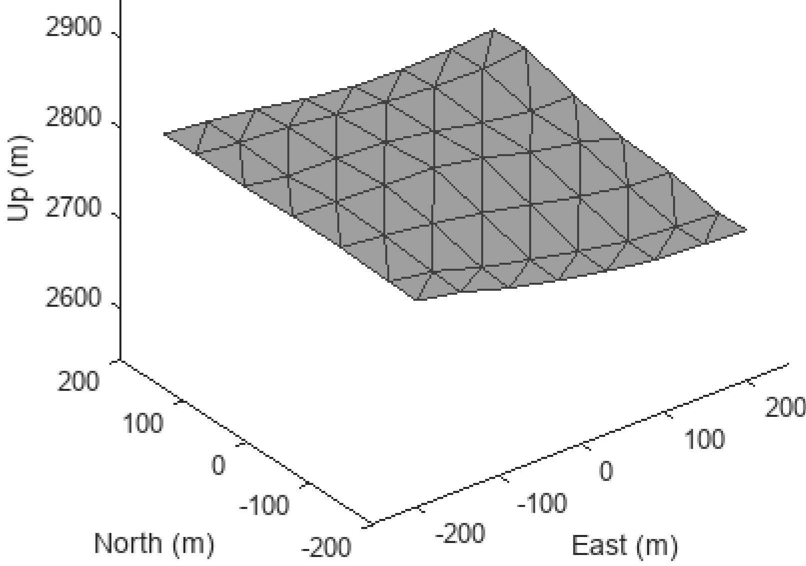
Add Terrain and Buildings to UAV Scenario

This example shows how to add terrain and custom building mesh to a UAV scenario.

Add Terrain Surface

Add terrain surface based on terrain elevation data from the `n39_w106_3arc_v2.dt1` DTED file.

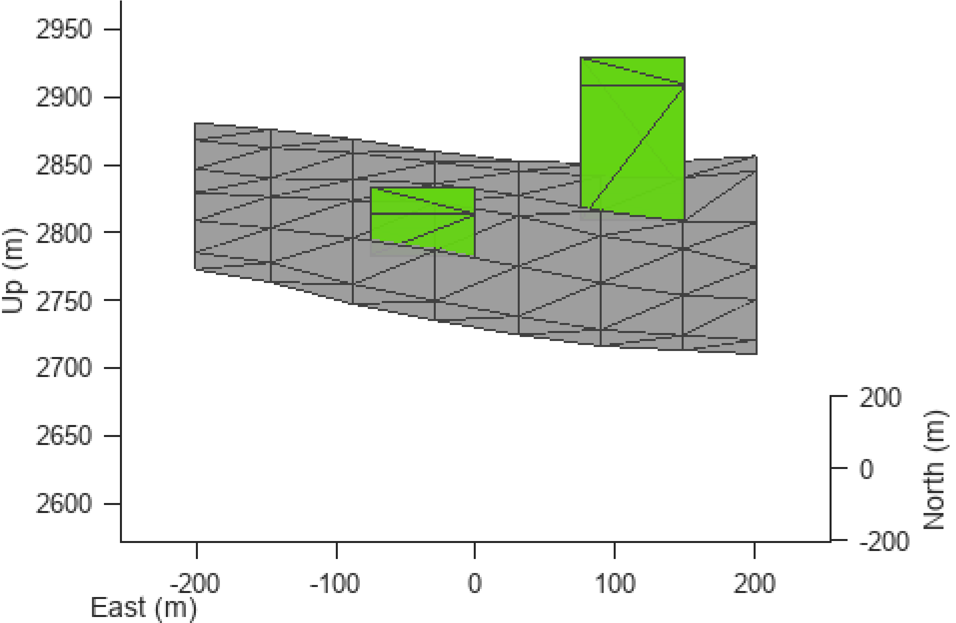
```
addCustomTerrain("CustomTerrain","n39_w106_3arc_v2.dt1");
scenario = uavScenario("ReferenceLocation", [39.5 -105.5 0]);
addMesh(scenario,"terrain", {"CustomTerrain", [-200 200], [-200 200]}, [0.6 0.6 0.6]);
show3D(scenario);
```



Add Buildings

Add a couple custom building meshes using vertices and polygon meshes into the scenario. Use the `terrainHeight` function to get ground height for each build base.

```
buildingCenters = [-50, -50; 100 100];

buildingHeights = [30 100];
buildingBoundary = [-25 -25; -25 50; 50 50; 50 -25];
for idx = 1:size(buildingCenters,1)
    buildingVertices = buildingBoundary+buildingCenters(idx,:);
    buildingBase = min(terrainHeight(scenario,buildingVertices(:,1),buildingVertices(:,2)));
    addMesh(scenario,"polygon", {buildingVertices, buildingBase+[0 buildingHeights(idx)]}, [0.392
end

show3D(scenario);
view([0 15])
```



Remove Custom Terrain

Remove the custom terrain that was imported.

```
removeCustomTerrain("CustomTerrain")
```

Input Arguments

scene — UAV scenario
uavScenario object

UAV scenario, specified as a `uavScenario` object.

x — *x*-positions in scenario
vector | matrix

x-positions in scenario specified as a vector or matrix of scalar values in meters. If specified as a matrix, the `y` input and `heights` output are also a matrix of the same size.

Example: `[1 2 0.5 -0.97]`

Data Types: `double`

y — *y*-positions in scenario
vector | matrix

y-positions in scenario specified as a vector or matrix of scalar values in meters. If specified as a matrix, the x input and heights output are also a matrix of the same size.

Example: [1 2 0.5 -0.97]

Data Types: double

Name-Value Pair Arguments

Specify optional pairs of arguments as Name1=Value1,...,NameN=ValueN, where Name is the argument name and Value is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose Name *in quotes.*

Example: h = terrainHeight(scene,x,y,"UseLatLon",true) uses latitude and longitude for the x and y inputs.

UseLatLon — Enable latitude and longitude coordinates
false (default) | true

Enable latitude and longitude coordinates, specified as true or false.

- When specified as true, the x and y coordinates are interpreted as longitude and latitude, respectively.
- When specified as false, the x and y coordinates are interpreted as Cartesian coordinates.

ReferenceFrame — Reference frame of coordinates
"ENU" (default) | name of defined inertial frame

Reference frame of coordinates, specified as an inertial frame name defined in the InertialFrames property of the uavScenario object scene. You can add new inertial frames to the scenario using the addInertialFrame object function.

Output Arguments

heights — Terrain heights at each position
vector | matrix

Terrain heights at each position, returned as a vector or matrix of scalar values in meters. If returned as a matrix, the x and y inputs are also a matrix of the same size.

Example: [1 2 0.5 -0.97]

Data Types: double

Version History
Introduced in R2021a

See Also
uavScenario | addMesh | addCustomTerrain | removeCustomTerrain

updateSensors

Update sensor readings in UAV scenario

Syntax

`updateSensors(scene)`

Description

`updateSensors(scene)` updates all sensor readings based on latest states of all platforms in the UAV scenario, `scene`.

Examples

Create and Simulate UAV Scenario

Create a UAV scenario and set its local origin.

```
scene = uavScenario("UpdateRate",200,"StopTime",2,"ReferenceLocation",[46, 42, 0]);
```

Add an inertial frame called `MAP` to the scenario.

```
scene.addInertialFrame("ENU","MAP",trvec2tform([1 0 0]));
```

Add one ground mesh and two cylindrical obstacle meshes to the scenario.

```
scene.addMesh("Polygon", {[-100 0; 100 0; 100 100; -100 100],[-5 0]},[0.3 0.3 0.3]);
scene.addMesh("Cylinder", {[20 10 10],[0 30]}, [0 1 0]);
scene.addMesh("Cylinder", {[46 42 5],[0 20]}, [0 1 0], "UseLatLon", true);
```

Create a UAV platform with a specified waypoint trajectory in the scenario. Define the mesh for the UAV platform.

```
traj = waypointTrajectory("Waypoints", [0 -20 -5; 20 -20 -5; 20 0 -5],"TimeOfArrival",[0 1 2]);
uavPlat = uavPlatform("UAV",scene,"Trajectory",traj);
updateMesh(uavPlat,"quadrotor", {4}, [1 0 0],eul2tform([0 0 pi]));
addGeoFence(uavPlat,"Polygon", {[-50 0; 50 0; 50 50; -50 50],[0 100]},true,"ReferenceFrame","ENU
```

Attach an INS sensor to the front of the UAV platform.

```
insModel = insSensor();
ins = uavSensor("INS",uavPlat,insModel,"MountingLocation",[4 0 0]);
```

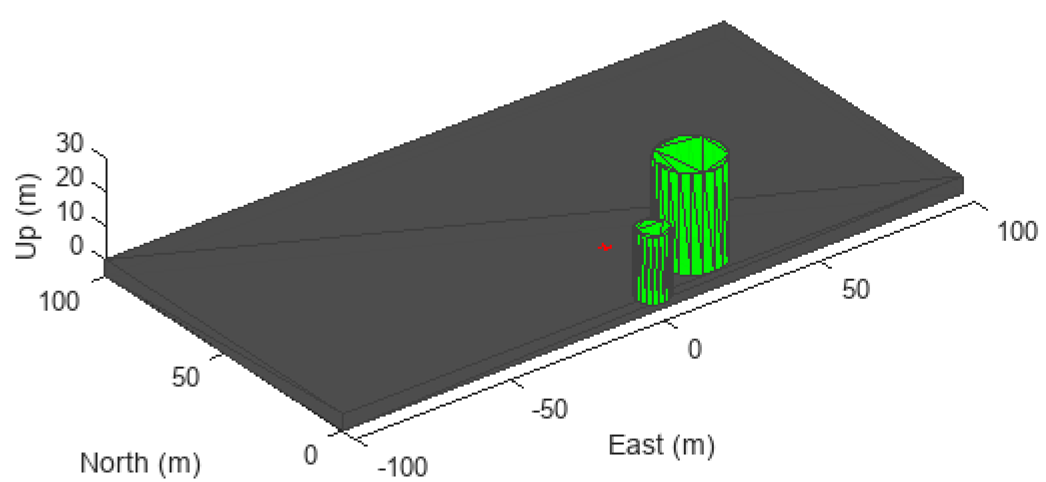
Visualize the scenario in 3-D.

```
ax = show3D(scene);
axis(ax,"equal");
```

Simulate the scenario.

```
setup(scene);
while advance(scene)
```

```
    % Update sensor readings
    updateSensors(scene);

    % Visualize the scenario
    show3D(scene,"Parent",ax,"FastUpdate",true);
    drawnow limitrate
end
```



Input Arguments

scene — UAV scenario
uavScenario object

UAV scenario, specified as a `uavScenario` object.

Version History
Introduced in R2020b

See Also
`insSensor` | `gpsSensor` | `uavSensor`

read

Gather latest reading from UAV sensor

Syntax

`[isUpdated,t,sensorReadings] = read(sensor)`

Description

`[isUpdated,t,sensorReadings] = read(sensor)` gathers the simulated sensor output sensor readings from the latest update of the UAV platform associated with the specified sensor `sensor`. The function returns an indicator `isUpdated` of whether the reading was updated at the simulation step in the scenario with timestamp `t`.

Input Arguments

`sensor` — UAV sensor added to platform in scenario

`uavSensor` object

UAV sensor added to a platform in a scenario, specified as a `uavSensor` object.

Output Arguments

`isUpdated` — Sensor reading update indicator

`0` or `false` | `1` or `true`

Sensor reading update indicator, returned as a logical `0` (`false`) or `1`(`true`). If the sensor reading updated at the current simulation step, the function returns this argument as `true`.

Data Types: `logical`

`t` — Timestamp of the generated sensor reading

scalar in seconds

Timestamp of the generated sensor reading, returned as a scalar in seconds.

Data Types: `double`

`sensorReadings` — Simulated sensor readings

`insSensor` output | `gpsSensor` output | `uavLidarPointCloudGenerator` output

Simulated sensor readings, which depends on the type of sensor specified in the `sensor` input argument. See the **Usage** syntax for the appropriate `insSensor`, `gpsSensor`, or `uavLidarPointCloudGenerator` System object.

Version History

Introduced in R2020b

See Also

Objects
`uavScenario` | `uavPlatform` | `uavSensor`

Topics
"UAV Scenario Tutorial"

copy

Class: `uav.SensorAdaptor`
Package: `uav`

Copy sensor adaptor

Syntax

```
sensorObjCopy = copy(sensorObj)
```

Description

`sensorObjCopy = copy(sensorObj)` creates a deep copy of the `uav.SensorAdaptor` object with the behavior set in the `copyElement` method.

Input Arguments

`sensorObj` — UAV sensor model
object of subclass of `uav.SensorAdaptor`

UAV sensor object, specified as an object of a subclass of `uav.SensorAdaptor`.

Output Arguments

`sensorObjCopy` — Deep copy of UAV sensor model
object of subclass of `uav.SensorAdaptor`

Deep copy of the UAV sensor object, returned as an object of a subclass of `uav.SensorAdaptor`. The copy is of the same object type, and has the same properties, as the object specified to `sensorObj`.

Version History

Introduced in R2022a

See Also

Functions
`copyElement` | `uav.SensorAdaptor.getMotion` | `getEmptyOutputs` | `setup` | `read` | `reset`

Objects
`uav.SensorAdaptor` | `uavSensor` | `uavPlatform` | `uavScenario`

copyElement

Class: `uav.SensorAdaptor`
Package: `uav`

Copy sensor adaptor object

Syntax

```
b = copyElement(h)
```

Description

`b = copyElement(h)` makes a copy of the sensor adapter handle `h`, and returns a sensor adapter handle of the same class.

Override `copyElement` in your subclass to control copy behavior.

Input Arguments

h — Sensor adapter to copy
`uav.SensorAdaptor` object

Sensor adapter to copy, specified as a `uav.SensorAdaptor`.

Output Arguments

b — Copy of sensor adapter
`uav.SensorAdaptor` object

Copy of sensor adapter, returned as a `uav.SensorAdaptor`.

Version History

Introduced in R2022a

See Also

Functions
`copy` | `getEmptyOutputs` | `uav.SensorAdaptor.getMotion` | `reset` | `setup` | `read`

Objects
`uav.SensorAdaptor` | `uavSensor` | `uavPlatform` | `uavScenario`

getEmptyOutputs

Class: `uav.SensorAdaptor`
Package: `uav`

Return empty sensor outputs without sensor inputs

Syntax

`out = getEmptyOutputs(sensorObj)`

Description

`out = getEmptyOutputs(sensorObj)` gets empty outputs when the sensor is not initialized using `setup`.

Input Arguments

`sensorObj` — UAV sensor model
object of subclass of `uav.SensorAdaptor`

UAV sensor object, specified as an object of a subclass of `uav.SensorAdaptor`.

Output Arguments

`out` — Empty sensor outputs
cell array

Empty sensor outputs, returned as a cell array of variables that matches the `varargout` output of the `read` function.

Version History

Introduced in R2021a

See Also

Functions
`copy` | `copyElement` | `uav.SensorAdaptor.getMotion` | `reset` | `setup` | `read`

Objects
`uav.SensorAdaptor` | `uavSensor` | `uavPlatform` | `uavScenario`

Topics
"Simulate Radar Sensor Mounted On UAV"

uav.SensorAdaptor.getMotion

Class: `uav.SensorAdaptor`
Package: `uav`

Get sensor motion in platform reference frame

Syntax

```
motion = getMotion(scenario,platform,sensor,t)
```

Description

`motion = getMotion(scenario,platform,sensor,t)` return the sensor motion in the platform reference frame for the given time `t`.

Input Arguments

`scenario` — UAV scenario
`uavScenario` object

UAV scenario, specified as a `uavScenario` object. This scenario contains the `uavPlatform` object `platform`, which also contains the sensor object `sensorObj`, which is a subclass of `uav.SensorAdaptor`.

`platform` — UAV platform
`uavPlatform` object

UAV scenario, specified as a `uavPlatform` object. This platform contains the sensor object `sensorObj`, which is a subclass of `uav.SensorAdaptor`.

`sensor` — UAV sensor to add to platform in scenario
`uavSensor` object

UAV sensor to add to a platform in a scenario, specified as a `uavSensor` object.

`t` — Simulation time
positive scalar

Simulation time, specified as a positive scalar.

Output Arguments

`motion` — UAV platform motion at current instance in scenario
16-element vector

UAV platform motion at the current instance in a UAV scenario, returned as a 16-element vector with these elements in this order:

- `[x y z]` — Positions in the *xyz*-axes in meters

- `[vx vy vz]` — Velocities in the *xyz*-directions in meters per second
- `[ax ay az]` — Accelerations in the *xyz*-directions in meters per second squared
- `[qw qx qy qz]` — Quaternion vector for orientation
- `[wx wy wz]` — Angular velocities in radians per second

Data Types: `double`

Version History

Introduced in R2021a

See Also

Functions
`copy` | `copyElement` | `getEmptyOutputs` | `reset` | `setup` | `read`

Objects
`uav.SensorAdaptor` | `uavSensor` | `uavPlatform` | `uavScenario`

Topics
"Simulate Radar Sensor Mounted On UAV"

read

Class: `uav.SensorAdaptor`
Package: `uav`

Read from custom sensor model

Syntax

```
varargout = read(sensorObj,scenario,platform,sensor,t)
```

Description

`varargout = read(sensorObj,scenario,platform,sensor,t)` reads sensor data from the sensor model `sensorObj`. Specify the UAV scenario, platform, sensor, and simulation time `t`. The function returns the sensor readings from the implemented sensor model.

Input Arguments

`sensorObj` — UAV sensor model
object of subclass of `uav.SensorAdaptor`

UAV sensor object, specified as an object of a subclass of `uav.SensorAdaptor`.

`scenario` — UAV scenario
`uavScenario` object

UAV scenario, specified as a `uavScenario` object. This scenario contains the `uavPlatform` object `platform`, which also contains the sensor object `sensorObj`, which is a subclass of `uav.SensorAdaptor`.

`platform` — UAV platform
`uavPlatform` object

UAV scenario, specified as a `uavPlatform` object. This platform contains the sensor object `sensorObj`, which is a subclass of `uav.SensorAdaptor`.

`sensor` — UAV sensor to add to platform in scenario
`uavSensor` object

UAV sensor to add to a platform in a scenario, specified as a `uavSensor` object.

`t` — Simulation time
positive scalar

Simulation time, specified as a positive scalar.

Output Arguments

`varargout` — Variable-length output argument list
`varargout`

Variable-length output argument list, returned as `varargout`.

Version History

Introduced in R2021a

See Also

Functions
`copy` | `copyElement` | `uav.SensorAdaptor.getMotion` | `getEmptyOutputs` | `reset` | `setup`

Objects
`uav.SensorAdaptor` | `uavSensor` | `uavPlatform` | `uavScenario`

Topics
"Simulate Radar Sensor Mounted On UAV"

reset

Class: `uav.SensorAdaptor`
Package: `uav`

Reset custom sensor model

Syntax

```
reset(sensorObj)
```

Description

`reset(sensorObj)` resets the sensor model state and releases internal resources if needed.

Input Arguments

`sensorObj` — UAV sensor model
object of subclass of `uav.SensorAdaptor`

UAV sensor object, specified as an object of a subclass of `uav.SensorAdaptor`.

Version History

Introduced in R2021a

See Also

Functions
`copy` | `copyElement` | `uav.SensorAdaptor.getMotion` | `getEmptyOutputs` | `setup` | `read`

Objects
`uav.SensorAdaptor` | `uavSensor` | `uavPlatform` | `uavScenario`

Topics
"Simulate Radar Sensor Mounted On UAV"

setup

Class: uav.SensorAdaptor
Package: uav

Set up custom sensor model

Syntax

```
setup(sensorObj,scenario,platform)
```

Description

setup(sensorObj,scenario,platform) initializes the sensor model with information from the UAV scenario and platform that the sensor is attached to.

Input Arguments

sensorObj — UAV sensor model
object of subclass of uav.SensorAdaptor

UAV sensor object, specified as an object of a subclass of uav.SensorAdaptor.

scenario — UAV scenario
uavScenario object

UAV scenario, specified as a uavScenario object. This scenario contains the uavPlatform object platform, which also contains the sensor object sensorObj, which is a subclass of uav.SensorAdaptor.

platform — UAV platform
uavPlatform object

UAV scenario, specified as a uavPlatform object. This platform contains the sensor object sensorObj, which is a subclass of uav.SensorAdaptor.

Version History

Introduced in R2021a

See Also

Functions
copy | copyElement | uav.SensorAdaptor.getMotion | getEmptyOutputs | reset | read

Objects
uav.SensorAdaptor | uavSensor | uavPlatform | uavScenario

Topics

"Simulate Radar Sensor Mounted On UAV"

readLoggedOutput

Read logged output messages

Syntax

```
logTable = readLoggedOutput(ulogOBJ)
logTable = readLoggedOutput(ulogOBJ,Name,Value)
```

Description

`logTable = readLoggedOutput(ulogOBJ)` reads the data of all logged output messages from the specified `ulogreader` object and returns a timetable that contains log levels and messages.

`logTable = readLoggedOutput(ulogOBJ,Name,Value)` reads specific logged output messages based on the specified name-value pairs.

Example: `readLoggedOutput(ulog,'Time',[d1 d2])`

Examples

Read Messages from ULOG File

Load the ULOG file. Specify the relative path of the file.

```
ulog = ulogreader('flight.ulg');
```

Read all topic messages.

```
msg = readTopicMsgs(ulog);
```

Specify the time interval between which to select messages.

```
d1 = ulog.StartTime;
d2 = d1 + duration([0 0 55],'Format','hh:mm:ss.SSSSSS');
```

Read messages from the topic `'vehicle_attitude'` with an instance ID of `0` in the time interval `[d1 d2]`.

```
data = readTopicMsgs(ulog,'TopicNames',{'vehicle_attitude'}, ...
'InstanceID',{0},'Time',[d1 d2]);
```

Extract topic messages for the topic.

```
vehicle_attitude = data.TopicMessages{1,1};
```

Read all system information.

```
systeminfo = readSystemInformation(ulog);
```

Read all initial parameter values.

```
params = readParameters(ulog);
```

Read all logged output messages.

```
loggedoutput = readLoggedOutput(ulog);
```

Read logged output messages in the time interval.

```
log = readLoggedOutput(ulog,'Time',[d1 d2]);
```

Input Arguments

ulogOBJ — ULOG file reader
ulogreader object

ULOG file reader, specified as a `ulogreader` object.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `'Time',[d1 d2]`

Time — Time interval
two-element vector

Time interval between which to select messages, specified as a two-element vector of `duration`, or a double array. The duration array is specified in the `'hh:mm:ss.SSSSSS'` format. The double array is specified in microseconds.

Example: `'Time',[d1 d2]`

Output Arguments

logTable — Logged output messages
timetable

Logged output messages, returned as a timetable with the columns:

- `LogLevel`
- `Messages`

Version History

Introduced in R2020b

See Also

Objects
`ulogreader`

Functions
`readSystemInformation` | `readParameters` | `readTopicMsgs`

readParameters

Read parameter values

Syntax

```
paramsTable = readParameters(ulogOBJ)
```

Description

`paramsTable = readParameters(ulogOBJ)` reads the data of all initial parameters from the specified `ulogreader` object and returns a table that contains all the parameter names with their respective values.

Examples

Read Messages from ULOG File

Load the ULOG file. Specify the relative path of the file.

```
ulog = ulogreader('flight.ulg');
```

Read all topic messages.

```
msg = readTopicMsgs(ulog);
```

Specify the time interval between which to select messages.

```
d1 = ulog.StartTime;
d2 = d1 + duration([0 0 55],'Format','hh:mm:ss.SSSSSS');
```

Read messages from the topic `'vehicle_attitude'` with an instance ID of `0` in the time interval `[d1 d2]`.

```
data = readTopicMsgs(ulog,'TopicNames',{'vehicle_attitude'}, ...
'InstanceID',{0},'Time',[d1 d2]);
```

Extract topic messages for the topic.

```
vehicle_attitude = data.TopicMessages{1,1};
```

Read all system information.

```
systeminfo = readSystemInformation(ulog);
```

Read all initial parameter values.

```
params = readParameters(ulog);
```

Read all logged output messages.

```
loggedoutput = readLoggedOutput(ulog);
```

Read logged output messages in the time interval.

```
log = readLoggedOutput(ulog,'Time',[d1 d2]);
```

Input Arguments

`ulogOBJ` — ULOG file reader
`ulogreader` object

ULOG file reader, specified as a `ulogreader` object.

Output Arguments

`paramsTable` — Initial parameters
table

Initial parameters, returned as a table with the columns:

- `Parameters`
- `Value`

Version History

Introduced in R2020b

See Also

Objects
`ulogreader`

Functions
`readSystemInformation` | `readLoggedOutput` | `readTopicMsgs`

readSystemInformation

Read information messages

Syntax

```
infoTable = readSystemInformation(ulogOBJ)
```

Description

`infoTable = readSystemInformation(ulogOBJ)` reads the data of information messages from the specified `ulogreader` object and returns a table that contains all the information fields with their respective values.

Examples

Read Messages from ULOG File

Load the ULOG file. Specify the relative path of the file.

```
ulog = ulogreader('flight.ulg');
```

Read all topic messages.

```
msg = readTopicMsgs(ulog);
```

Specify the time interval between which to select messages.

```
d1 = ulog.StartTime;
d2 = d1 + duration([0 0 55],'Format','hh:mm:ss.SSSSSS');
```

Read messages from the topic `'vehicle_attitude'` with an instance ID of `0` in the time interval `[d1 d2]`.

```
data = readTopicMsgs(ulog,'TopicNames',{'vehicle_attitude'}, ...
'InstanceID',{0},'Time',[d1 d2]);
```

Extract topic messages for the topic.

```
vehicle_attitude = data.TopicMessages{1,1};
```

Read all system information.

```
systeminfo = readSystemInformation(ulog);
```

Read all initial parameter values.

```
params = readParameters(ulog);
```

Read all logged output messages.

```
loggedoutput = readLoggedOutput(ulog);
```

Read logged output messages in the time interval.

```
log = readLoggedOutput(ulog,'Time',[d1 d2]);
```

Input Arguments

`ulogOBJ` — ULOG file reader
`ulogreader` object

ULOG file reader, specified as a `ulogreader` object.

Output Arguments

`infoTable` — System information
table

System information, returned as a table with the columns:

- `InformationField`
- `Value`

Version History

Introduced in R2020b

See Also

Objects
`ulogreader`

Functions
`readParameters` | `readLoggedOutput` | `readTopicMsgs`

readTopicMsgs

Read topic messages

Syntax

```
msgTable = readTopicMsgs(ulogOBJ)
msgTable = readTopicMsgs(ulogOBJ,Name,Value)
```

Description

`msgTable = readTopicMsgs(ulogOBJ)` reads the data of all topic messages from the specified `ulogreader` object and returns a table that contains topic names, instance ID, start timestamp, last timestamp, topic messages, and message format for all available topics.

`msgTable = readTopicMsgs(ulogOBJ,Name,Value)` reads the data pertaining to the specified name-value pairs.

Example: `readTopicMsgs(ulog,'TopicNames',{'vehicle_attitude'},'InstanceID',{0},'Time',[d1 d2])`

Examples

Read Messages from ULOG File

Load the ULOG file. Specify the relative path of the file.

```
ulog = ulogreader('flight.ulg');
```

Read all topic messages.

```
msg = readTopicMsgs(ulog);
```

Specify the time interval between which to select messages.

```
d1 = ulog.StartTime;
d2 = d1 + duration([0 0 55],'Format','hh:mm:ss.SSSSSS');
```

Read messages from the topic `'vehicle_attitude'` with an instance ID of `0` in the time interval `[d1 d2]`.

```
data = readTopicMsgs(ulog,'TopicNames',{'vehicle_attitude'}, ...
'InstanceID',{0},'Time',[d1 d2]);
```

Extract topic messages for the topic.

```
vehicle_attitude = data.TopicMessages{1,1};
```

Read all system information.

```
systeminfo = readSystemInformation(ulog);
```

Read all initial parameter values.

```
params = readParameters(ulog);
```

Read all logged output messages.

```
loggedoutput = readLoggedOutput(ulog);
```

Read logged output messages in the time interval.

```
log = readLoggedOutput(ulog,'Time',[d1 d2]);
```

Input Arguments

`ulogOBJ` — ULOG file reader
`ulogreader` object

ULOG file reader, specified as a `ulogreader` object.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `'Time',[d1 d2]`

`TopicNames` — Topic names of desired messages
cell array of character vectors | string array

Topic names of the desired messages, specified as a cell array of character vectors or a string array.

Example: `'TopicNames',{'sensor_combined','actuator_outputs'}` or `'TopicNames',["actuator_outputs","ekf2_timestamps"]`

`InstanceID` — Instance ID of topic of desired messages
cell array of positive integer scalars or vectors

Instance ID of the topic of the desired messages, specified as a cell array of positive integer scalars or vectors. Specify this name-value pair along with its corresponding `'TopicNames'` name-value pair.

Example: `'TopicNames',{'vehicle_attitude','actuator_outputs'},'InstanceID',{0,[0 1]}`

`Time` — Time interval
two-element vector

Time interval between which to select messages, specified as a two-element vector of `duration`, or a double array. The duration array is specified in the `'hh:mm:ss.SSSSSS'` format. The double array is specified in microseconds.

Example: `'Time',[d1 d2]`

Output Arguments

`msgTable` — Topic messages information
table

Topic messages information, returned as a table with the columns:

- `TopicNames`
- `InstanceID`
- `StartTimestamp`
- `LastTimestamp`
- `TopicMessages`
- `MsgFormat`

Version History

Introduced in R2020b

See Also

Objects
`ulogreader`

Functions
`readSystemInformation` | `readParameters` | `readLoggedOutput`

lookupPose

Obtain pose information for certain time

Syntax

```
[position,orientation,velocity,acceleration,angularVelocity] = lookupPose(
traj,sampleTimes)
```

Description

`[position,orientation,velocity,acceleration,angularVelocity] = lookupPose(traj,sampleTimes)` returns the pose information of the waypoint trajectory at the specified sample times. If any sample time is beyond the duration of the trajectory, the corresponding pose information is returned as `NaN`.

Input Arguments

`traj` — Waypoint trajectory
`waypointTrajectory` object

Waypoint trajectory, specified as a `waypointTrajectory` object.

`sampleTimes` — Sample times
M-element vector of nonnegative scalar

Sample times in seconds, specified as an *M*-element vector of nonnegative scalars.

Output Arguments

`position` — Position in local navigation coordinate system (m)
M-by-3 matrix

Position in the local navigation coordinate system in meters, returned as an *M*-by-3 matrix.

M is specified by the `sampleTimes` input.

Data Types: `double`

`orientation` — Orientation in local navigation coordinate system
M-element quaternion column vector | 3-by-3-by-*M* real array

Orientation in the local navigation coordinate system, returned as an *M*-by-1 `quaternion` column vector or a 3-by-3-by-*M* real array.

Each quaternion or 3-by-3 rotation matrix is a frame rotation from the local navigation coordinate system to the current body coordinate system.

M is specified by the `sampleTimes` input.

Data Types: `double`

`velocity` — Velocity in local navigation coordinate system (m/s)
M-by-3 matrix

Velocity in the local navigation coordinate system in meters per second, returned as an *M*-by-3 matrix.

M is specified by the `sampleTimes` input.

Data Types: `double`

`acceleration` — Acceleration in local navigation coordinate system (m/s^2)
M-by-3 matrix

Acceleration in the local navigation coordinate system in meters per second squared, returned as an *M*-by-3 matrix.

M is specified by the `sampleTimes` input.

Data Types: `double`

`angularVelocity` — Angular velocity in local navigation coordinate system (rad/s)
M-by-3 matrix

Angular velocity in the local navigation coordinate system in radians per second, returned as an *M*-by-3 matrix.

M is specified by the `sampleTimes` input.

Data Types: `double`

Version History

Introduced in R2020b

See Also

Objects
`waypointTrajectory`

Functions
`waypointInfo` | `perturbations` | `perturb`

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.

Version History

Introduced in R2020b

See Also

Objects
`waypointTrajectory`

Functions
`lookupPose` | `perturbations` | `perturb`

perturb

Apply perturbations to object

Syntax

```
offsets = perturb(obj)
```

Description

`offsets = perturb(obj)` applies the perturbations defined on the object, `obj` and returns the offset values. You can define perturbations on the object by using the `perturbations` function.

Examples

Perturb Waypoint Trajectory

Define a waypoint trajectory. By default, this trajectory contains two waypoints.

```
traj = waypointTrajectory
```

```
traj = 
  waypointTrajectory with properties:

         SampleRate: 100
    SamplesPerFrame: 1
          Waypoints: [2x3 double]
      TimeOfArrival: [2x1 double]
         Velocities: [2x3 double]
             Course: [2x1 double]
        GroundSpeed: [2x1 double]
          ClimbRate: [2x1 double]
        Orientation: [2x1 quaternion]
          AutoPitch: 0
           AutoBank: 0
     ReferenceFrame: 'NED'
```

Define perturbations on the `Waypoints` property and the `TimeOfArrival` property.

```
rng(2020);
perturbs1 = perturbations(traj,'Waypoints','Normal',1,1)
```

```
perturbs1=2×3 table
       Property         Type            Value      
    _______________    ________    ________________

    "Waypoints"        "Normal"    {[  1]}    {[  1]}
    "TimeOfArrival"    "None"      {[NaN]}    {[NaN]}
```

```
perturbs2 = perturbations(traj,'TimeOfArrival','Selection',{[0;1],[0;2]})
```

```
perturbs2=2×3 table
       Property          Type                    Value
    _______________    ___________    _______________________________

    "Waypoints"        "Normal"       {[     1]}    {[            1]}
    "TimeOfArrival"    "Selection"    {1x2 cell}    {[0.5000 0.5000]}
```

Perturb the trajectory.

```
offsets = perturb(traj)
```

```
offsets=2×1 struct array with fields:
    Property
    Offset
    PerturbedValue
```

The `Waypoints` property and the `TimeOfArrival` property have changed.

```
traj.Waypoints
```

```
ans = 2×3

    1.8674    1.0203    0.7032
    2.3154   -0.3207    0.0999
```

```
traj.TimeOfArrival
```

```
ans = 2×1

     0
     2
```

Perturb Accuracy of insSensor

Create an `insSensor` object.

```
sensor = insSensor
```

```
sensor = 
  insSensor with properties:

           MountingLocation: [0 0 0]            m
               RollAccuracy: 0.2                deg
              PitchAccuracy: 0.2                deg
                YawAccuracy: 1                  deg
           PositionAccuracy: [1 1 1]            m
           VelocityAccuracy: 0.05               m/s
       AccelerationAccuracy: 0                  m/s²
    AngularVelocityAccuracy: 0                  deg/s
                  TimeInput: 0
               RandomStream: 'Global stream'
```

Define the perturbation on the `RollAccuracy` property as three values with an equal possibility each.

```
values = {0.1 0.2 0.3}

values=1×3 cell array
    {[0.1000]}    {[0.2000]}    {[0.3000]}


probabilities = [1/3 1/3 1/3]

probabilities = 1×3

    0.3333    0.3333    0.3333


perturbations(sensor,'RollAccuracy','Selection',values,probabilities)

ans=7×3 table
            Property                  Type                            Value
    _________________________    ___________    ______________________________________

    "RollAccuracy"               "Selection"    {1x3 cell}    {[0.3333 0.3333 0.3333]}
    "PitchAccuracy"              "None"         {[   NaN]}    {[                 NaN]}
    "YawAccuracy"                "None"         {[   NaN]}    {[                 NaN]}
    "PositionAccuracy"           "None"         {[   NaN]}    {[                 NaN]}
    "VelocityAccuracy"           "None"         {[   NaN]}    {[                 NaN]}
    "AccelerationAccuracy"       "None"         {[   NaN]}    {[                 NaN]}
    "AngularVelocityAccuracy"    "None"         {[   NaN]}    {[                 NaN]}
```

Perturb the `sensor` object using the perturb function.

```
rng(2020)
perturb(sensor);
sensor

sensor = 
  insSensor with properties:

           MountingLocation: [0 0 0]            m
               RollAccuracy: 0.5                deg
              PitchAccuracy: 0.2                deg
                YawAccuracy: 1                  deg
           PositionAccuracy: [1 1 1]            m
           VelocityAccuracy: 0.05               m/s
       AccelerationAccuracy: 0                  m/s²
    AngularVelocityAccuracy: 0                  deg/s
                  TimeInput: 0
               RandomStream: 'Global stream'
```

The `RollAccuracy` is perturbed to `0.5` deg.

Input Arguments

`obj` — Object for perturbation
objects

Object for perturbation, specified as an object. The objects that you can perturb include:

- `waypointTrajectory`
- `insSensor`

Output Arguments

`offsets` — Property offsets
array of structure

Property offsets, returned as an array of structures. Each structure contains these fields:

Field Name	Description
`Property`	Name of perturbed property
`Offset`	Offset values applied in the perturbation
`PerturbedValue`	Property values after the perturbation

Version History

Introduced in R2020b

See Also

`perturbations`

perturbations

Perturbation defined on object

Syntax

```
perturbs = perturbations(obj)
perturbs = perturbations(obj,property)
perturbs = perturbations(obj,property,'None')
perturbs = perturbations(obj,property,'Selection',values,probabilities)
perturbs = perturbations(obj,property,'Normal',mean,deviation)
perturbs = perturbations(obj,property,'TruncatedNormal',mean,deviation,
lowerLimit,upperLimit)
perturbs = perturbations(obj,property,'Uniform',minVal,maxVal)
perturbs = perturbations(obj,property,'Custom',perturbFcn)
```

Description

`perturbs = perturbations(obj)` returns the list of property perturbations, `perturbs`, defined on the object, `obj`. The returned `perturbs` lists all the perturbable properties. If any property is not perturbed, then its corresponding `Type` is returned as `"Null"` and its corresponding `Value` is returned as `{Null,Null}`.

`perturbs = perturbations(obj,property)` returns the current perturbation applied to the specified `property`.

`perturbs = perturbations(obj,property,'None')` defines a `property` that must not be perturbed.

`perturbs = perturbations(obj,property,'Selection',values,probabilities)` defines the `property` perturbation offset drawn from a set of `values` that have corresponding `probabilities`.

`perturbs = perturbations(obj,property,'Normal',mean,deviation)` defines the `property` perturbation offset drawn from a normal distribution with specified `mean` and standard `deviation`.

`perturbs = perturbations(obj,property,'TruncatedNormal',mean,deviation,
lowerLimit,upperLimit)` defines the `property` perturbation offset drawn from a normal distribution with specified `mean`, standard `deviation`, lower limit, and upper limit.

`perturbs = perturbations(obj,property,'Uniform',minVal,maxVal)` defines the `property` perturbation offset drawn from a uniform distribution on an interval [`minVal`, `maxValue`].

`perturbs = perturbations(obj,property,'Custom',perturbFcn)` enables you to define a custom function, `perturbFcn`, that draws the perturbation offset value.

Examples

Default Perturbation Properties of `waypointTrajectory`

Create a `waypointTrajectory` object.

```
traj = waypointTrajectory;
```

Show the default perturbation properties using the `perturbations` method.

```
perturbs = perturbations(traj)
```

```
perturbs=2×3 table
       Property          Type           Value
    _______________    ______    __________________

    "Waypoints"        "None"    {[NaN]}    {[NaN]}
    "TimeOfArrival"    "None"    {[NaN]}    {[NaN]}
```

Perturb Accuracy of insSensor

Create an `insSensor` object.

```
sensor = insSensor
```

```
sensor = 
  insSensor with properties:

           MountingLocation: [0 0 0]            m
               RollAccuracy: 0.2                deg
              PitchAccuracy: 0.2                deg
                YawAccuracy: 1                  deg
           PositionAccuracy: [1 1 1]            m
           VelocityAccuracy: 0.05               m/s
       AccelerationAccuracy: 0                  m/s²
    AngularVelocityAccuracy: 0                  deg/s
                  TimeInput: 0
               RandomStream: 'Global stream'
```

Define the perturbation on the `RollAccuracy` property as three values with an equal possibility each.

```
values = {0.1 0.2 0.3}
```

```
values=1×3 cell array
    {[0.1000]}    {[0.2000]}    {[0.3000]}
```

```
probabilities = [1/3 1/3 1/3]
```

```
probabilities = 1×3

    0.3333    0.3333    0.3333
```

```
perturbations(sensor,'RollAccuracy','Selection',values,probabilities)
```

```
ans=7×3 table
              Property                   Type                             Value
    ___________________________    ____________    ________________________________________

    "RollAccuracy"                 "Selection"     {1x3 cell}    {[0.3333 0.3333 0.3333]}
    "PitchAccuracy"                "None"          {[   NaN]}    {[                 NaN]}
    "YawAccuracy"                  "None"          {[   NaN]}    {[                 NaN]}
    "PositionAccuracy"             "None"          {[   NaN]}    {[                 NaN]}
    "VelocityAccuracy"             "None"          {[   NaN]}    {[                 NaN]}
    "AccelerationAccuracy"         "None"          {[   NaN]}    {[                 NaN]}
    "AngularVelocityAccuracy"      "None"          {[   NaN]}    {[                 NaN]}
```

Perturb the `sensor` object using the perturb function.

```
rng(2020)
perturb(sensor);
sensor
```

```
sensor = 
  insSensor with properties:

           MountingLocation: [0 0 0]            m    
               RollAccuracy: 0.5                deg  
              PitchAccuracy: 0.2                deg  
                YawAccuracy: 1                  deg  
           PositionAccuracy: [1 1 1]            m    
           VelocityAccuracy: 0.05               m/s  
       AccelerationAccuracy: 0                  m/s² 
    AngularVelocityAccuracy: 0                  deg/s
                  TimeInput: 0                       
               RandomStream: 'Global stream'         
```

The `RollAccuracy` is perturbed to `0.5` deg.

Perturb Waypoint Trajectory

Define a waypoint trajectory. By default, this trajectory contains two waypoints.

```
traj = waypointTrajectory
```

```
traj = 
  waypointTrajectory with properties:

         SampleRate: 100
    SamplesPerFrame: 1
          Waypoints: [2x3 double]
      TimeOfArrival: [2x1 double]
         Velocities: [2x3 double]
             Course: [2x1 double]
        GroundSpeed: [2x1 double]
          ClimbRate: [2x1 double]
        Orientation: [2x1 quaternion]
          AutoPitch: 0
           AutoBank: 0
```

```
     ReferenceFrame: 'NED'
```

Define perturbations on the `Waypoints` property and the `TimeOfArrival` property.

```
rng(2020);
perturbs1 = perturbations(traj,'Waypoints','Normal',1,1)
```

```
perturbs1=2×3 table
       Property          Type             Value
    _______________    ________    __________________

    "Waypoints"        "Normal"    {[  1]}    {[  1]}
    "TimeOfArrival"    "None"      {[NaN]}    {[NaN]}
```

```
perturbs2 = perturbations(traj,'TimeOfArrival','Selection',{[0;1],[0;2]})
```

```
perturbs2=2×3 table
       Property          Type                     Value
    _______________    ___________    ________________________________

    "Waypoints"        "Normal"       {[     1]}    {[             1]}
    "TimeOfArrival"    "Selection"    {1x2 cell}    {[0.5000 0.5000]}
```

Perturb the trajectory.

```
offsets = perturb(traj)
```

```
offsets=2×1 struct array with fields:
    Property
    Offset
    PerturbedValue
```

The `Waypoints` property and the `TimeOfArrival` property have changed.

```
traj.Waypoints
```

```
ans = 2×3

    1.8674    1.0203    0.7032
    2.3154   -0.3207    0.0999
```

```
traj.TimeOfArrival
```

```
ans = 2×1

     0
     2
```

Input Arguments

obj — Object to be perturbed
objects

Object to be perturbed, specified as an object. The objects that you can perturb include:

- `waypointTrajectory`
- `insSensor`

`property` — Perturbable property
property name

Perturbable property, specified as a property name. Use `perturbations` to obtain a full list of perturbable properties for the specified `obj`.

`values` — Perturbation offset values
n-element cell array of property values

Perturbation offset values, specified as an *n*-element cell array of property values. The function randomly draws the perturbation value for the property from the cell array based on the values' corresponding probabilities specified in the `probabilities` input.

`probabilities` — Drawing probabilities for each perturbation value
n-element array of nonnegative scalar

Drawing probabilities for each perturbation value, specified as an *n*-element array of nonnegative scalars, where *n* is the number of perturbation values provided in the `values` input. The sum of all elements must be equal to one.

For example, you can specify a series of perturbation value-probability pair as `{x1`,`x2`,…,`xn}` and `{p1`,`p2`,…,`pn}`, where the probability of drawing `xi` is `pi` (`i` = 1, 2, …,n).

`mean` — Mean of normal or truncated normal distribution
scalar | vector | matrix

Mean of normal or truncated normal distribution, specified as a scalar, vector, or matrix. The dimension of `mean` must be compatible with the corresponding property that you perturb.

`deviation` — Standard deviation of normal or truncated normal distribution
nonnegative scalar | vector of nonnegative scalar | matrix of nonnegative scalar

Standard deviation of normal or truncated normal distribution, specified as a nonnegative scalar, vector of nonnegative scalars, or matrix of nonnegative scalars. The dimension of `deviation` must be compatible with the corresponding property that you perturb.

`lowerLimit` — Lower limit of truncated normal distribution
scalar | vector | matrix

Lower limit of the truncated normal distribution, specified as a scalar, vector, or matrix. The dimension of `lowerLimit` must be compatible with the corresponding property that you perturb.

`upperLimit` — Upper limit of truncated normal distribution
scalar | vector | matrix

Upper limit of the truncated normal distribution, specified as a scalar, vector, or matrix. The dimension of `upperLimit` must be compatible with the corresponding property that you perturb.

`minVal` — Minimum value of uniform distribution interval
scalar | vector | matrix

Minimum value of the uniform distribution interval, specified as a scalar, vector, or matrix. The dimension of `minVal` must be compatible with the corresponding property that you perturb.

`maxVal` — Maximum value of uniform distribution interval
scalar | vector | matrix

Maximum value of the uniform distribution interval, specified as a scalar, vector, or matrix. The dimension of `maxVal` must be compatible with the corresponding property that you perturb.

`perturbFcn` — Perturbation function
function handle

Perturbation function, specified as a function handle. The function must have this syntax:

```
offset = myfun(propVal)
```

where `propVal` is the value of the `property` and `offset` is the perturbation offset for the property.

Output Arguments

`perturbs` — Perturbations defined on object
table of perturbation property

Perturbations defined on the object, returned as a table of perturbation properties. The table has three columns:

- `Property` — Property names.
- `Type` — Type of perturbations, returned as `"None"`, `"Selection"`, `"Normal"`, `"TruncatedNormal"`, `"Uniform"`, or `"Custom"`.
- `Value` — Perturbation values, returned as a cell array.

More About

Specify Perturbation Distributions

You can specify the distribution for the perturbation applied to a specific property.

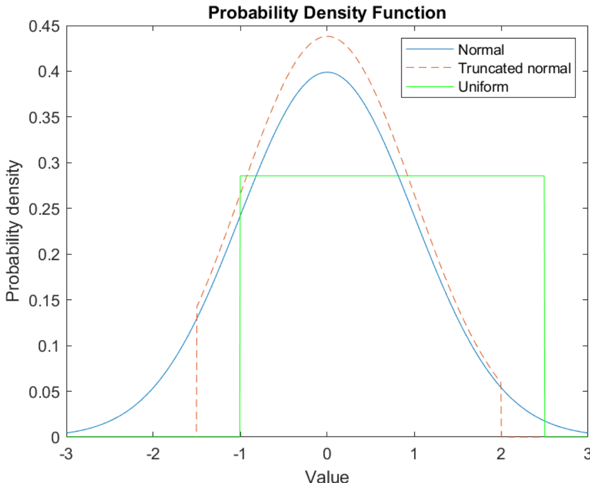
- Selection distribution — The function defines the perturbation offset as one of the specified values with the associated probability. For example, if you specify the values as `[1 2]` and specify the probabilities as `[0.7 0.3]`, then the `perturb` function adds an offset value of `1` to the property with a probability of `0.7` and add an offset value of `2` to the property with a probability of `0.3`. Use selection distribution when you only want to perturb the property with a number of discrete values.
- Normal distribution — The function defines the perturbation offset as a value drawn from a normal distribution with the specified mean and standard deviation (or covariance). Normal distribution is the most commonly used distribution since it mimics the natural perturbation of parameters in most cases.
- Truncated normal distribution — The function defines the perturbation offset as a value drawn from a truncated normal distribution with the specified mean, standard deviation (or covariance), lower limit, and upper limit. Different from the normal distribution, the values drawn from a truncated normal distribution are truncated by the lower and upper limit. Use truncated normal distribution when you want to apply a normal distribution, but the valid values of the property are confined in an interval.

- Uniform distribution — The function defines the perturbation offset as a value drawn from a uniform distribution with the specified minimum and maximum values. All the values in the interval (specified by the minimum and maximum values) have the same probability of realization.
- Custom distribution — Customize your own perturbation function. The function must have this syntax:

  ```
  offset = myfun(propVal)
  ```

  where `propVal` is the value of the `property` and `offset` is the perturbation offset for the property.

This figure shows probability density functions for a normal distribution, a truncated normal distribution, and a uniform distribution, respectively.



Version History

Introduced in R2020b

See Also

`perturb`

3

Functions

addmavlinkkeys

Add MAVLink keys from `.env` file

Syntax

```
addmavlinkkeys(filename)
```

Description

`addmavlinkkeys(filename)` adds the MAVLink keys stored in the `.env` file `filename` to the MATLAB session. Use MAVLink keys in signing channels for message signing. See the `mavlinksigning` object for more details.

Examples

Add and Remove MAVLink Keys

Check the contents of the `keys.env` file. The file contains two keys. `Key1` is a 32-element `uint8` vector and `Key2` is a 32-element `uint8` vector encoded using base 64 encoding using the `matlab.net.base64encode` function.

```
type keys.env
```

```
Key1 = [1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32]
Key2_Base64 = "1xOIT/+tTcjdXaCH794ihyStgzxDll+ZLw3SAzolIu0="
```

Add the MAVLink keys from the file `keys.env` file to the MATLAB session.

```
addmavlinkkeys("keys.env")
```

List all of the keys in the current MATLAB session.

```
keys = lsmavlinkkeys
```

```
keys = 1x2 string
    "Key1"    "Key2"
```

Remove both the `Key` and `Key_Base64` MAVLink keys.

```
rmmavlinkkeys(["Key1","Key2"])
```

Input Arguments

`filename` — MAVLink keys file
string scalar | string array | character vector

MAVLink keys file, specified as a string scalar, string array, or character vector. Use a string array to add keys from multiple `.env` files simultaneously. You must specify the absolute or relative path to each file, and each file path must end with `".env"`.

The files store MAVLink keys in one of these formats:

- `KeyName = KeyValue`, where *KeyName* is the desired key name, and `KeyValue` is a 32-element `uint8` vector.
- `KeyName_Base64 = KeyValueStr`, where *KeyName* is the desired key name, and `KeyValueStr` is a string scalar converted from a 32-element `uint8` vector using base 64 encoding. For more information about base 64 encoding, see the `matlab.net.base64encode` function.

Example: `addmavlinkkeys("keys.env")` adds keys from the `"keys.env"` file on the MATLAB path.

Data Types: `string` | `char`

Version History
Introduced in R2023a

See Also
`mavlinksigning` | `lsmavlinkkeys` | `rmmavlinkkeys`

addCustomTerrain

Add custom terrain data

Syntax

```
addCustomTerrain(terrainName,files)
addCustomTerrain( ___ ,Name,Value)
```

Description

`addCustomTerrain(terrainName,files)` adds terrain data specified by `files` for use with UAV scenarios. Add the terrain to `uavScenario` objects using the `addMesh` object function. Custom terrain data is available for current and future sessions of MATLAB until you call `removeCustomTerrain`.

`addCustomTerrain( ___ ,Name,Value)` adds custom terrain data with additional options specified by one or more name-value pairs.

Input Arguments

`terrainName` — User-defined identifier for terrain data

string scalar | character vector

User-defined identifier for terrain data, specified as a string scalar or a character vector.

Data Types: `char` | `string`

`files` — Names of DTED files to read

string scalar | character vector | string vector | cell array of character vectors

Names of DTED files to read, specified as a string scalar, a character vector, a string vector, or a cell array of character vectors.

- To add custom terrain from one DTED file, specify `files` as a string scalar or a character vector.
- To add custom terrain from multiple DTED files, specify `files` as a string vector or a cell array of character vectors. If you specify multiple files that do not cover a complete rectangular geographic region, you must set the `FillMissing` name-value argument to `true`.

The form of each element of `files` depends on the location of the file.

- If the file is in your current folder or in a folder on the MATLAB path, then specify the name of the file, such as `"myFile.dt1"`.
- If the file is not in the current folder or in a folder on the MATLAB path, then specify the full or relative path name, such as `"C:\myfolder\myFile.dt1"` or `"dataDir\myFile.dt1"`.

Data Types: `char` | `string` | `cell`

Name-Value Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `'FillMissing',true`

`Attribution` — Attribution of custom terrain data
character vector | string scalar

Attribution of custom terrain data, specified as a character vector or a string scalar. Attributions display on geographic plots that use the custom terrain. By default, the attribution is empty.

Attribution of custom terrain data, specified as a character vector or a string scalar. The attribution data is not displayed in UAV scenarios, but can be displayed on geographic plots or the Site Viewer map. By default, the value is empty.

Data Types: `char` | `string`

`FillMissing` — Fill data of missing files with value 0
`false` (default) | `true`

Fill data of missing files with value 0, specified as `true` or `false`. Missing file values are required to complete a rectangular geographic region with the input `files`.

Data Types: `logical`

`WriteLocation` — Name of folder to write extracted terrain files to
character vector | string scalar

Name of folder to write extracted terrain files to, specified as a character vector or a string scalar. The folder must exist and have write permissions. By default, `addCustomTerrain` writes extracted terrain files to a temporary folder that it generates using the `tempname` function.

Data Types: `char` | `string`

Tips

- You can find and download DTED files by using EarthExplorer, a data portal provided by the US Geological Survey (USGS). From the list of data sets, search for DTED files by selecting **Digital Elevation**, **SRTM**, and then **SRTM 1 Arc-Second Global** and **SRTM Void Filled**.

Version History

Introduced in R2021a

See Also

`removeCustomTerrain` | `uavScenario` | `addMesh`

angdiff

Difference between two angles

Syntax

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

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

Description

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

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

Examples

Calculate Difference Between Two Angles

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

```
d = 3.1416
```

Calculate Difference Between Two Angle Arrays

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

```
d = 1×3

    1.5708   -0.7854   -3.1416
```

Calculate Angle Differences of Adjacent Elements

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

```
d = 1×3
```

```
   -1.5708   -0.7854    0.7854
```

Input Arguments

`alpha` — Angle in radians
scalar | vector | matrix | multidimensional array

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

Example: `pi/2`

`beta` — Angle in radians
scalar | vector | matrix | multidimensional array

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

Example: `pi/2`

Output Arguments

`delta` — Difference between two angles
scalar | vector | matrix | multidimensional array

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

Version History

Introduced in R2015a

Extended Capabilities

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

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]`

Version History

Introduced in R2015a

Extended Capabilities

C/C++ Code Generation

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

See Also

`quat2axang` | `quaternion`

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]`

Version History

Introduced in R2015a

Extended Capabilities

C/C++ Code Generation

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

See Also

`rotm2axang` | `so2` | `so3`

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]`

Version History

Introduced in R2015a

Extended Capabilities

C/C++ Code Generation

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

See Also

`tform2axang` | `se2` | `se3`

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 matrix

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

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

Output Arguments

`hom` — Homogeneous points
n-by-(k+1) matrix

Homogeneous points, returned as an n-by-(k+1) matrix, containing n points. k must be greater than or equal to 1.

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

Version History

Introduced in R2015a

R2023a: `cart2hom` Supports 2-D Cartesian Coordinates

The `cart` argument now accepts 2-D Cartesian coordinates and `cart2hom` outputs 2-D homogeneous points.

Extended Capabilities

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

See Also
`hom2cart`

coverageDecomposition

Decompose concave polygon into convex polygons

Syntax

```
polygons = coverageDecomposition(concavePolygon)
polygons = coverageDecomposition(concavePolygon,IntersectionTol=tolerance)
```

Description

The `coverageDecomposition` function decomposes coverage regions into subregions using vertex-edge decomposition[1]. This enables you to plan more efficient coverage paths using optimized planning options for each subregion to have minimum turns and then connect the plans together.

`polygons = coverageDecomposition(concavePolygon)` decomposes the concave polygon `concavePolygon` into convex polygon areas `polygons` based on minimum-width criteria[1].

`polygons = coverageDecomposition(concavePolygon,IntersectionTol=tolerance)` tests the vertex edge decompositions with slopes that are between the tolerance `tolerance`, which respect to the polygon edges.

Examples

Plan Coverage Path for Defined Region

This example shows how to plan a coverage path for a region in local coordinates and compares the results of using the exhaustive solver with the results of using the minimum traversal solver.

Define the vertices for a coverage space.

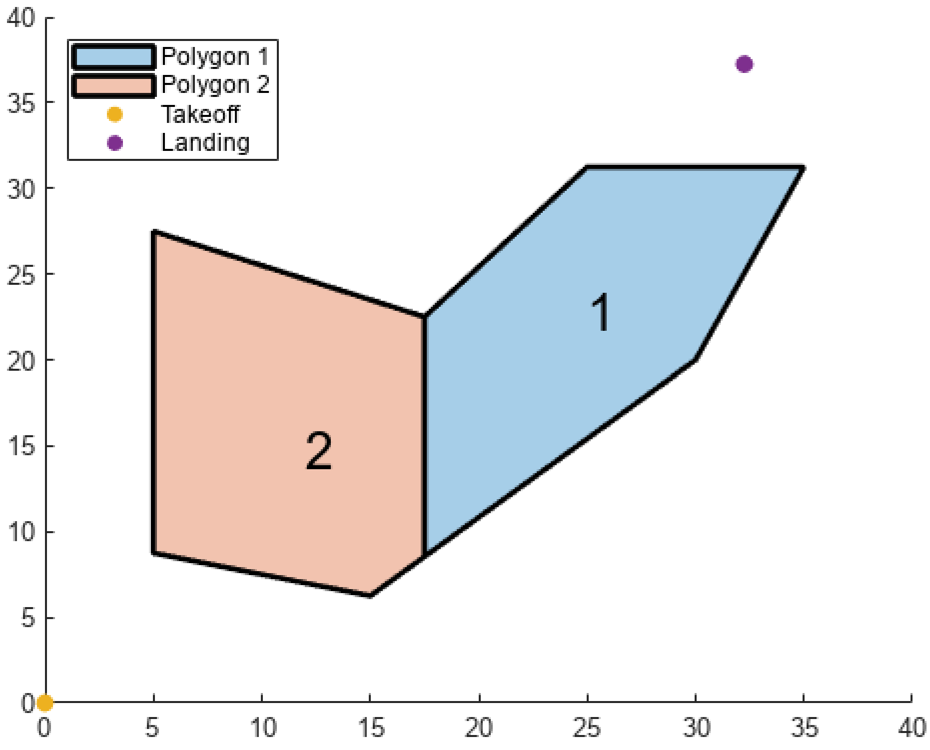
```
area = [5 8.75; 5 27.5; 17.5 22.5; 25 31.25; 35 31.25; 30 20; 15 6.25];
```

Because vertices define a concave polygon and the coverage planner requires convex polygons, decompose the polygon into convex polygons. Then create a coverage space with the polygons from decomposition.

```
polygons = coverageDecomposition(area);
cs = uavCoverageSpace(Polygons=polygons);
```

Define the takeoff and landing positions at `[0 0 0]` and `[32.25 37.25 0]`, respectively. Then show the coverage space and plot the takeoff and landing positions.

```
takeoff = [0 0 0];
landing = [32.25 37.25 0];
show(cs);
exampleHelperPlotTakeoffLandingLegend(takeoff,landing)
```



Create a coverage planner with the exhaustive solver algorithm and another coverage planner with a minimum traversal solver algorithm. Because `Polygon 2` is closer to the takeoff position, set the visiting sequence of the solver parameters such that we traverse `Polygon 2` first.

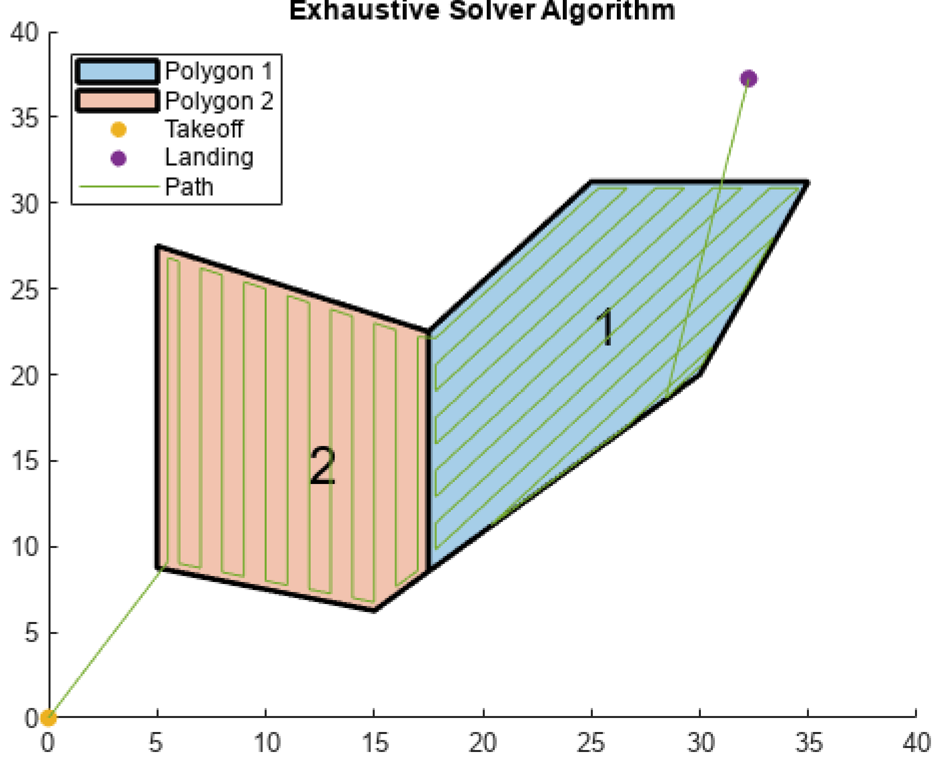
```
cpeExh = uavCoveragePlanner(cs,Solver="Exhaustive");
cpMin = uavCoveragePlanner(cs,Solver="MinTraversal");
cpeExh.SolverParameters.VisitingSequence = [2 1];
cpMin.SolverParameters.VisitingSequence = [2 1];
```

Plan with both solver algorithms using the same takeoff and landing positions.

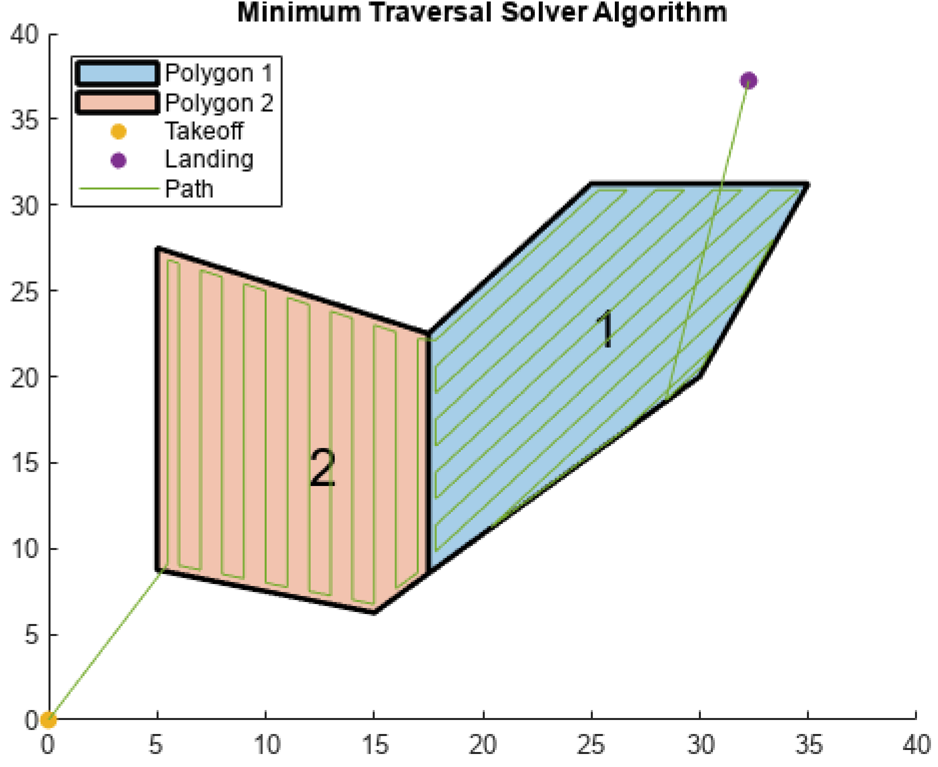
```
[wptsExh,solnExh] = plan(cpeExh,takeoff,landing);
[wptsMin,solnMin] = plan(cpMin,takeoff,landing);
```

Show the planned path for both the exhaustive and the minimum traversal algorithms.

```
figure
show(cs);
title("Exhaustive Solver Algorithm")
exampleHelperPlotTakeoffLandingLegend(takeoff,landing,wptsExh)
```




```
figure
show(cs);
title("Minimum Traversal Solver Algorithm")
exampleHelperPlotTakeoffLandingLegend(takeoff,landing,wptsMin)
```



Export the waypoints from the minimum traversal solver to a `.waypoints` file with the reference frame set to north-east-down.

```
exportWaypointsPlan(cpMin,solnMin,"coveragepath.waypoints",ReferenceFrame="NED")
```

Input Arguments

`concavePolygon` — Concave polygon to decompose

N-by-2 matrix

Concave polygon to decompose, specified as an *N*-by-2 matrix.

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

Data Types: `double`

`tolerance` — Intersection tolerance

`0` (default) | nonnegative numeric scalar

Intersection tolerance, specified nonnegative numeric scalar. The `coverageDecomposition` function uses the intersection tolerance to test if the vertex-edge decompositions are valid. For example, when the `coverageDecomposition` function decomposes a polygon into two polygons, those two polygons now have edges that intersect each other. The slopes of these intersecting edges must be within this tolerance. This is useful when polygons are slightly off alignment.

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

Data Types: `single` | `double`

Output Arguments

`polygons` — Convex polygons
N-element cell array of *M*-by-2 matrices

Convex polygons, returned as a *N*-element cell array of *M*-by-2 matrices. *N* is the total number of polygons, and *M* is the total number of vertices that define each polygon.

The format of the vertices is local *xy*-coordinates in the form [*x y*], in meters.

Example: `{[0 0; 0 1; 1 1; 1 0],[1 1; 2 2; 3 1]}`

Version History

Introduced in R2023a

References

[1] Li, Yan, Hai Chen, Meng Joo Er, and Xinmin Wang. "Coverage Path Planning for UAVs Based on Enhanced Exact Cellular Decomposition Method." *Mechatronics* 21, no. 5 (August 2011): 876–85. https://doi.org/10.1016/j.mechatronics.2010.10.009.

See Also

`uavCoveragePlanner` | `uavCoverageSpace`

Topics

"Optimally Survey and Customize Coverage of Region of Interest using Coverage Planner"

createCustomSensorTemplate

Create sample implementation for UAV custom sensor interface

Syntax

`createCustomSensorTemplate`

Description

`createCustomSensorTemplate` creates a sample implementation for UAV custom sensor that inherits from the `uav.SensorAdaptor` class. This function opens a new file in the MATLAB Editor.

Examples

Simulate IMU Sensor Mounted on UAV

Create a sensor adaptor for an `imuSensor` from Navigation Toolbox™ and gather readings for a simulated UAV flight scenario.

Create Sensor Adaptor

Use the `createCustomSensorTemplate` function to generate a template sensor and update it to adapt an `imuSensor` object for usage in UAV scenario.

```
createCustomSensorTemplate
```

This example provides the adaptor class `uavIMU`, which can be viewed using the following command.

```
edit uavIMU.m
```

Use Sensor Adaptor in UAV Scenario Simulation

Use the IMU sensor adaptor in a UAV Scenario simulation. First, create the scenario.

```
scenario = uavScenario("StopTime", 8, "UpdateRate", 100);
```

Create a UAV platform and specify the trajectory. Add a fixed-wing mesh for visualization.

```
plat = uavPlatform("UAV", scenario, "Trajectory", ...
    waypointTrajectory([0 0 0; 100 0 0; 100 100 0], "TimeOfArrival", [0 5 8], "AutoBank", true))
updateMesh(plat,"fixedwing", {10}, [1 0 0], eul2tform([0 0 pi]));
```

Attach the IMU sensor using the `uavSensor` object and specify the `uavIMU` as an input. Load parameters for the sensor model.

```
imu = uavSensor("IMU", plat, uavIMU(imuSensor));

fn = fullfile(matlabroot,'toolbox','shared',...
    'positioning','positioningdata','generic.json');
loadparams(imu.SensorModel,fn,"GenericLowCost9Axis");
```

Visualize the scenario.

```
figure
ax = show3D(scenario);
xlim([-20 200]);
ylim([-20 200]);
```

Preallocate the `simData` structure and fields to store simulation data. The IMU sensor will output acceleration and angular rates.

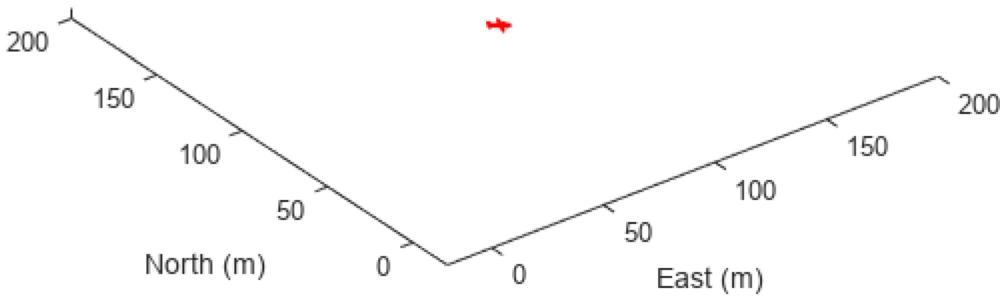
```
simData = struct;
simData.Time = duration.empty;
simData.AccelerationX = zeros(0,1);
simData.AccelerationY = zeros(0,1);
simData.AccelerationZ = zeros(0,1);
simData.AngularRatesX = zeros(0,1);
simData.AngularRatesY = zeros(0,1);
simData.AngularRatesZ = zeros(0,1);
```

Setup the scenario.

```
setup(scenario);
```

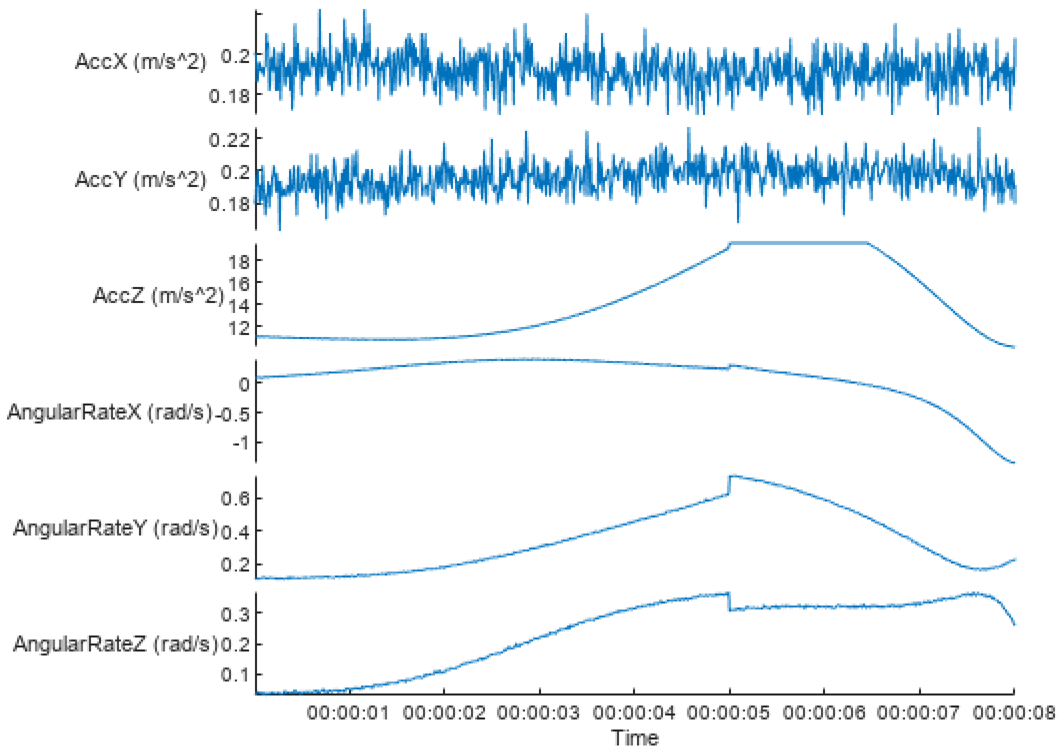
Run the simulation using the `advance` function. Update the sensors and record the data.

```
updateCounter = 0;
while true
    % Advance scenario.
    isRunning = advance(scenario);
    updateCounter = updateCounter + 1;
    % Update sensors and read IMU data.
    updateSensors(scenario);
    [isUpdated, t, acc, gyro] = read(imu);
    % Store data in structure.
    simData.Time = [simData.Time; seconds(t)];
    simData.AccelerationX = [simData.AccelerationX; acc(1)];
    simData.AccelerationY = [simData.AccelerationY; acc(2)];
    simData.AccelerationZ = [simData.AccelerationZ; acc(3)];
    simData.AngularRatesX = [simData.AngularRatesX; gyro(1)];
    simData.AngularRatesY = [simData.AngularRatesY; gyro(2)];
    simData.AngularRatesZ = [simData.AngularRatesZ; gyro(3)];

    % Update visualization every 10 updates.
    if updateCounter > 10
        show3D(scenario, "FastUpdate", true, "Parent", ax);
        updateCounter = 0;
        drawnow limitrate
    end
    % Exit loop when scenario is finished.
    if ~isRunning
        break;
    end
end
```



Visualize the simulated IMU readings.

```
simTable = table2timetable(struct2table(simData));
figure
stackedplot(simTable, ["AccelerationX", "AccelerationY", "AccelerationZ", ...
    "AngularRatesX", "AngularRatesY", "AngularRatesZ"], ...
    "DisplayLabels", ["AccX (m/s^2)", "AccY (m/s^2)", "AccZ (m/s^2)", ...
    "AngularRateX (rad/s)", "AngularRateY (rad/s)", "AngularRateZ (rad/s)"]);
```



Version History

Introduced in R2021a

See Also

`uav.SensorAdaptor`

enu2lla

Transform local east-north-up coordinates to geodetic coordinates

Syntax

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

Description

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

Note

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

Examples

Transform ENU Coordinates to Geodetic Coordinates

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

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

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

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

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

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

```
lla = 1×3
10³ ×

    0.0460    0.0077    4.5254
```

Input Arguments

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

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

Example: `[-7134.8 -4556.3 2852.4]`

Data Types: `double`

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

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

Example: `[46.017 7.750 1673]`

Data Types: `double`

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

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

The flat Earth transformation method has these limitations:

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

Data Types: `char` | `string`

Output Arguments

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

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

Data Types: `double`

Version History

Introduced in R2020b

Extended Capabilities

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

See Also

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

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 intrinsic Euler rotation angles. Each row represents one Euler angle set in the sequence defined by the `sequence` argument. For example, with the default sequence `"ZYX"`, each row of `eul` is of the form `[zAngle yAngle xAngle]`.

Example: `[0 0 1.5708]`

sequence — Axis-rotation sequence
`"ZYX"` (default) | `"ZYZ"` | `"ZXY"` | `"ZXZ"` | `"YXY"` | `"YZX"` | `"YXZ"` | `"YZY"` | `"XYX"` | `"XYZ"` | `"XZX"` | `"XZY"`

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

- `"ZYX"` (default)
- `"ZYZ"`
- `"ZXY"`
- `"ZXZ"`
- `"YXY"`
- `"YZX"`
- `"YXZ"`
- `"YZY"`
- `"XYX"`
- `"XYZ"`
- `"XZX"`
- `"XZY"`

Each character indicates the corresponding axis. For example, if the sequence is `"ZYX"`, then the three specified Euler angles are interpreted in order as a rotation around the *z*-axis, a rotation around the *y*-axis, and a rotation around the *x*-axis. When applying this rotation to a point, it will apply the axis rotations in the order *x*, then *y*, then *z*.

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]`

Version History

Introduced in R2015a

R2023a: Additional Euler sequence support

`eul2quat` supports additional Euler sequences for the `sequences` argument. These are all the supported Euler sequences:

- `"ZYX"`

- "ZYZ"
- "ZXY"
- "ZXZ"
- "YXY"
- "YZX"
- "YXZ"
- "YZY"
- "XYX"
- "XYZ"
- "XZX"
- "XZY"

Extended Capabilities

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

See Also

quat2eul | quaternion

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 intrinsic Euler rotation angles. Each row represents one Euler angle set in the sequence defined by the `sequence` argument. For example, with the default sequence `"ZYX"`, each row of `eul` is of the form `[zAngle yAngle xAngle]`.

Example: `[0 0 1.5708]`

sequence — Axis-rotation sequence
`"ZYX"` (default) | `"ZYZ"` | `"ZXY"` | `"ZXZ"` | `"YXY"` | `"YZX"` | `"YXZ"` | `"YZY"` | `"XYX"` | `"XYZ"` | `"XZX"` | `"XZY"`

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

- `"ZYX"` (default)
- `"ZYZ"`
- `"ZXY"`
- `"ZXZ"`
- `"YXY"`
- `"YZX"`
- `"YXZ"`
- `"YZY"`
- `"XYX"`
- `"XYZ"`
- `"XZX"`
- `"XZY"`

Each character indicates the corresponding axis. For example, if the sequence is `"ZYX"`, then the three specified Euler angles are interpreted in order as a rotation around the *z*-axis, a rotation around the *y*-axis, and a rotation around the *x*-axis. When applying this rotation to a point, it will apply the axis rotations in the order *x*, then *y*, then *z*.

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]`

Version History

Introduced in R2015a

R2023a: Additional Euler sequence support

`eul2rotm` supports additional Euler sequences for the `sequences` argument. These are all the supported Euler sequences:

- `"ZYX"`
- `"ZYZ"`
- `"ZXY"`
- `"ZXZ"`
- `"YXY"`
- `"YZX"`
- `"YXZ"`
- `"YZY"`
- `"XYX"`
- `"XYZ"`
- `"XZX"`
- `"XZY"`

Extended Capabilities

C/C++ Code Generation

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

See Also

`rotm2eul` | `so2` | `so3`

eul2tform

Convert Euler angles to homogeneous transformation

Syntax

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

Description

`tform = 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 intrinsic Euler rotation angles. Each row represents one Euler angle set in the sequence defined by the `sequence` argument. For example, with the default sequence `"ZYX"`, each row of `eul` is of the form `[zAngle yAngle xAngle]`.

Example: `[0 0 1.5708]`

`sequence` — Axis-rotation sequence

`"ZYX"` (default) | `"ZYZ"` | `"ZXY"` | `"ZXZ"` | `"YXY"` | `"YZX"` | `"YXZ"` | `"YZY"` | `"XYX"` | `"XYZ"` | `"XZX"` | `"XZY"`

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

- `"ZYX"` (default)
- `"ZYZ"`
- `"ZXY"`
- `"ZXZ"`
- `"YXY"`
- `"YZX"`
- `"YXZ"`
- `"YZY"`
- `"XYX"`
- `"XYZ"`
- `"XZX"`
- `"XZY"`

Each character indicates the corresponding axis. For example, if the sequence is `"ZYX"`, then the three specified Euler angles are interpreted in order as a rotation around the *z*-axis, a rotation around the *y*-axis, and a rotation around the *x*-axis. When applying this rotation to a point, it will apply the axis rotations in the order *x*, then *y*, then *z*.

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* array of *n* homogeneous transformation matrices. When using the transformation 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]`

Version History

Introduced in R2015a

R2023a: Additional Euler sequence support

`eul2tform` supports additional Euler sequences for the `sequences` argument. These are all the supported Euler sequences:

- `"ZYX"`
- `"ZYZ"`
- `"ZXY"`
- `"ZXZ"`
- `"YXY"`
- `"YZX"`
- `"YXZ"`
- `"YZY"`
- `"XYX"`
- `"XYZ"`
- `"XZX"`
- `"XZY"`

Extended Capabilities

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

See Also
`tform2eul` | `se2` | `se3`

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, 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]`

Output 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*-dimensional space. *k* must be greater than or equal to 1.

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

Version History

Introduced in R2015a

R2023a: `hom2cart` Supports 2-D Homogeneous Points

The `hom` argument now accepts 2-D homogeneous points and `hom2cart` outputs 2-D Cartesian coordinates.

Extended Capabilities

C/C++ Code Generation

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

See Also

`cart2hom`

lla2enu

Transform geodetic coordinates to local east-north-up coordinates

Syntax

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

Description

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

Note

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

Examples

Transform Geodetic Coordinates to ENU Coordinates

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

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

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

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

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

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

```
xyzENU = 1×3
10³ ×

   -7.1244   -4.5572    2.8580
```

Input Arguments

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

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

Example: `[45.976 7.658 4531]`

Data Types: `double`

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

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

Example: `[46.017 7.750 1673]`

Data Types: `double`

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

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

The flat Earth transformation method has these limitations:

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

Data Types: `char` | `string`

Output Arguments

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

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

Data Types: `double`

Version History

Introduced in R2020b

Extended Capabilities

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

See Also

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

lla2ned

Transform geodetic coordinates to local north-east-down coordinates

Syntax

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

Description

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

Note

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

Examples

Transform Geodetic Coordinates to NED Coordinates

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

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

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

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

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

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

```
xyzNED = 1×3
10³ ×

   -4.5572   -7.1244   -2.8580
```

Input Arguments

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

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

Example: `[45.976 7.658 4531]`

Data Types: `double`

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

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

Example: `[46.017 7.750 1673]`

Data Types: `double`

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

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

The flat Earth transformation method has these limitations:

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

Data Types: `char` | `string`

Output Arguments

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

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

Data Types: `double`

Version History

Introduced in R2020b

Extended Capabilities

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

See Also

Functions
`enu2lla` | `lla2enu` | `ned2lla`

lsmavlinkkeys

List MAVLink keys in MATLAB session

Syntax

```
keynames = lsmavlinkkeys
```

Description

`keynames = lsmavlinkkeys` lists all MAVLink keys in the current MATLAB session. Use MAVLink keys in signing channels for message signing. See the `mavlinksigning` object for more details.

Examples

Add and Remove MAVLink Keys

Check the contents of the `keys.env` file. The file contains two keys. `Key1` is a 32-element `uint8` vector and `Key2` is a 32-element `uint8` vector encoded using base 64 encoding using the `matlab.net.base64encode` function.

```
type keys.env
```

```
Key1 = [1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32]
Key2_Base64 = "1xOIT/+tTcjdXaCH794ihyStgzxDll+ZLw3SAzolIu0="
```

Add the MAVLink keys from the file `keys.env` file to the MATLAB session.

```
addmavlinkkeys("keys.env")
```

List all of the keys in the current MATLAB session.

```
keys = lsmavlinkkeys
```

```
keys = 1x2 string
    "Key1"    "Key2"
```

Remove both the `Key` and `Key_Base64` MAVLink keys.

```
rmmavlinkkeys(["Key1","Key2"])
```

Output Arguments

keynames — MAVLink keys in MATLAB session
string scalar | string array

MAVLink keys in the MATLAB session, returned as a string scalar or string array.

Version History

Introduced in R2023a

See Also

`addmavlinkkeys` | `rmmavlinkkeys` | `mavlinksigning`

minjerkpolytraj

Generate minimum jerk trajectory through waypoints

Syntax

```
[q,qd,qdd,qddd,pp,tPoints,tSamples] = minjerkpolytraj(waypoints,timePoints,
numSamples)
[q,qd,qdd,qddd,pp,tPoints,tSamples] = minjerkpolytraj( ___ ,Name=Value)
[q,qd,qdd,qddd,pp,tPoints,tSamples] = minjerkpolytraj( ___
,TimeAllocation=true)
```

Description

`[q,qd,qdd,qddd,pp,tPoints,tSamples] = minjerkpolytraj(waypoints,timePoints, numSamples)` generates a minimum jerk polynomial trajectory that achieves a given set of input waypoints with their corresponding time points. The function returns positions, velocities, accelerations, and jerks at the given number of samples `numSamples`. The function also returns the piecewise polynomial `pp` form of the polynomial trajectory with respect to time, as well as the time points `tPoints` and the sample times `tSamples`.

`[q,qd,qdd,qddd,pp,tPoints,tSamples] = minjerkpolytraj( ___ ,Name=Value)` specifies options using one or more name-value pair arguments in addition to the input arguments from the previous syntax. For example, `minjerkpolytraj(waypoints,timePoints,numSamples,VelocityBoundaryCondition=[1 0 -1 -1; 1 1 1 -1])` generates a two-dimensional minimum jerk trajectory and specifies the velocity boundary conditions in each dimension for each waypoint.

`[q,qd,qdd,qddd,pp,tPoints,tSamples] = minjerkpolytraj( ___ ,TimeAllocation=true)` optimizes a combination of jerk and total segment time cost. In this case, the function treats `timePoints` as an initial guess for the time of arrival at the waypoints.

Examples

Compute Minimum Jerk Trajectory for 2-D Planar Motion

Use the `minjerkpolytraj` function with a given set of 2-D *xy* waypoints. Time points for the waypoints are also given.

```
wpts = [1 4 4 3 -2 0; 0 1 2 4 3 1];
tpts = 0:5;
```

Specify the number of samples in the output trajectory.

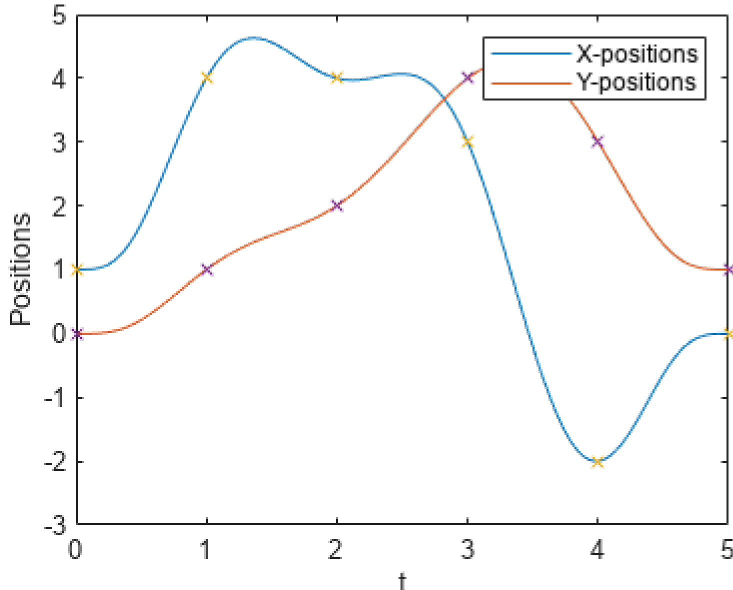
```
numsamples = 100;
```

Compute minimum jerk trajectories. The function outputs the trajectory positions (`q`), velocity (`qd`), acceleration (`qdd`), and jerks (`qddd`) at the given number of samples.

```
[q,qd,qdd,qddd,pp,timepoints,tsamples] = minjerkpolytraj(wpts,tpts,numsamples);
```

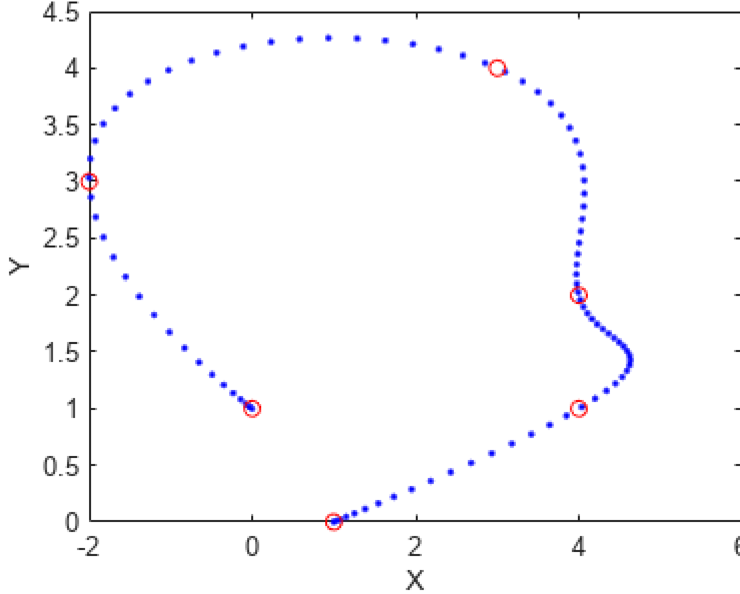
Plot the trajectories for the *x*- and *y*-positions. Compare the trajectory with each waypoint.

```
plot(tsamples,q)
hold on
plot(timepoints,wpts,'x')
xlabel('t')
ylabel('Positions')
legend('X-positions','Y-positions')
hold off
```



You can also verify the actual positions in the 2-D plane. Plot the separate rows of the `q` vector and the waypoints as *x*- and *y*- positions.

```
figure
plot(q(1,:),q(2,:),'.b',wpts(1,:),wpts(2,:),'or')
xlabel('X')
ylabel('Y')
```



Input Arguments

`waypoints` — Waypoints for trajectory
n-by-*p* matrix

Waypoints for the trajectory, specified as an *n*-by-*p* matrix. *n* is the dimension of the trajectory, and *p* is the number of waypoints.

Example: `[2 5 8 4; 3 4 10 12]`

Data Types: `single` | `double`

`timePoints` — Time points for waypoints of trajectory
p-element row vector

Time points for the waypoints of the trajectory, specified as a *p*-element row vector. *p* is the number of waypoints.

Example: `[1 2 3 5]`

Data Types: `single` | `double`

`numSamples` — Number of samples in output trajectory
positive integer

Number of samples in the output trajectory, specified as a positive integer.

Example: `50`

Data Types: `single` | `double`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Example: `minjerkpolytraj(waypoints,timePoints,numSamples,VelocityBoundaryCondition=[1 0 -1 -1; 1 1 1 -1])` generates a two-dimensional minimum jerk trajectory and specifies the velocity boundary conditions in each dimension for each waypoint.

`VelocityBoundaryCondition` — Velocity boundary conditions for each waypoint

n-by-*p* matrix

Velocity boundary conditions for each waypoint, specified as an *n*-by-*p* matrix. Each row sets the velocity boundary for the corresponding dimension of the trajectory at each of *p* waypoints. By default, the function uses a value of `0` at the boundary waypoints and `NaN` at the intermediate waypoints.

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

Data Types: `single` | `double`

`AccelerationBoundaryCondition` — Acceleration boundary conditions for each waypoint

n-by-*p* matrix

Acceleration boundary conditions for each waypoint, specified as an *n*-by-*p* matrix. Each row sets the acceleration boundary for the corresponding dimension of the trajectory at each of *p* waypoints. By default, the function uses a value of `0` at the boundary waypoints and `NaN` at the intermediate waypoints.

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

Data Types: `single` | `double`

`JerkBoundaryCondition` — Jerk boundary conditions for each waypoint

n-by-*p* matrix

Jerk boundary conditions for each waypoint, specified as an *n*-by-*p* matrix. Each row sets the jerk boundary for the corresponding dimension of the trajectory at each of *p* waypoints. By default, the function uses a value of `0` at the boundary waypoints and `NaN` at the intermediate waypoints.

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

Data Types: `single` | `double`

`TimeAllocation` — Time allocation flag

`false` or `0` (default) | `true` or `1`

Time allocation flag, specified as a logical `0` (`false`) or `1` (`true`). Enable this flag to optimize a combination of jerk and total segment time cost.

Note If singularity occurs when the time allocation flag is enabled, reduce the `MaxSegmentTime` to `MinSegmentTime` ratio.

Example: `TimeAllocation=true`

Data Types: `logical`

`TimeWeight` — Weight for time allocation
`100` (default) | positive scalar

Weight for time allocation, specified as a positive scalar.

Example: `TimeWeight=120`

Data Types: `single` | `double`

`MinSegmentTime` — Minimum time segment length
`0.1` (default) | positive scalar | (*p*–1)-element row vector

Minimum time segment length, specified as a positive scalar or (*p*–1)-element row vector.

Example: `MinSegmentTime=0.2`

Data Types: `single` | `double`

`MaxSegmentTime` — Maximum time segment length
`5` (default) | positive scalar | (*p*–1)-element row vector

Maximum time segment length, specified as a positive scalar or (*p*–1)-element row vector

Example: `MaxSegmentTime=10`

Data Types: `single` | `double`

Output Arguments

q — Positions of trajectory
n-by-*m* matrix

Positions of the trajectory at the given time samples in `tSamples`, returned as an *n*-by-*m* matrix. *n* is the dimension of the trajectory, and *m* is equal to `numSamples`.

qd — Velocities of trajectory
n-by-*m* matrix

Velocities of the trajectory at the given time samples in `tSamples`, returned as an *n*-by-*m* matrix. *n* is the dimension of the trajectory, and *m* is equal to `numSamples`.

qdd — Accelerations of trajectory
n-by-*m* matrix

Accelerations of the trajectory at the given time samples in `tSamples`, returned as an *n*-by-*m* matrix. *n* is the dimension of the trajectory, and *m* is equal to `numSamples`.

qddd — Jerks of trajectory
n-by-*m* matrix

Jerks of the trajectory at the given time samples in `tSamples`, returned as an *n*-by-*m* matrix. *n* is the dimension of the trajectory, and *m* is equal to `numSamples`.

pp — Piecewise polynomial
structure

Piecewise-polynomial, returned as a structure that defines the polynomial for each section of the piecewise trajectory. You can build your own piecewise polynomials using `mkpp`, or evaluate the polynomial at specified times using `ppval`. The structure contains the fields:

- `form`: `'pp'`.
- `breaks`: *p*-element vector of times when the piecewise trajectory changes forms. *p* is the number of waypoints.
- `coefs`: *n*(*p*–1)-by-`order` matrix for the coefficients for the polynomials. *n*(*p*–1) is the dimension of the trajectory times the number of `pieces`. Each set of *n* rows defines the coefficients for the polynomial that described each variable trajectory.
- `pieces`: *p*–1. The number of breaks minus 1.
- `order`: Degree of the polynomial + 1. The order of polynomial is 8.
- `dim`: *n*. The dimension of the control point positions.

`tPoints` — Time points for waypoints of trajectory
p-element row vector

Time points for the waypoints of the trajectory, returned as a *p*-element row vector. *p* is the number of waypoints.

`tSamples` — Time samples for trajectory
m-element row vector

Time samples for the trajectory, returned as an *m*-element row vector. Each element of the output position `q`, velocity `qd`, acceleration `qdd`, and jerk `qddd` has been sampled at the corresponding time in this vector.

Version History

Introduced in R2021b

References

[1] Bry, Adam, Charles Richter, Abraham Bachrach, and Nicholas Roy. “Aggressive Flight of Fixed-Wing and Quadrotor Aircraft in Dense Indoor Environments.” *The International Journal of Robotics Research*, 34, no. 7 (June 2015): 969–1002.

[2] Richter, Charles, Adam Bry, and Nicholas Roy. “Polynomial Trajectory Planning for Aggressive Quadrotor Flight in Dense Indoor Environments." *Paper presented at the International Symposium of Robotics Research (ISRR 2013)*, 2013.

Extended Capabilities

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

See Also

Functions
`minsnappolytraj`

minsnappolytraj

Generate minimum snap trajectory through waypoints

Syntax

```
[q,qd,qdd,qddd,qdddd,pp,tPoints,tSamples] = minsnappolytraj(waypoints,
timePoints,numSamples)
[q,qd,qdd,qddd,qdddd,pp,tPoints,tSamples] = minsnappolytraj( ___ ,Name=Value)
[q,qd,qdd,qddd,qdddd,pp,tPoints,tSamples] = minsnappolytraj( ___
,TimeAllocation=true)
```

Description

`[q,qd,qdd,qddd,qdddd,pp,tPoints,tSamples] = minsnappolytraj(waypoints, timePoints,numSamples)` generates a minimum snap polynomial trajectory that achieves a given set of input waypoints with their corresponding time points. The function returns positions, velocities, accelerations, jerks, and snaps at the given number of samples `numSamples`. The function also returns the piecewise polynomial `pp` form of the polynomial trajectory with respect to time, as well as the time points `tPoints`, and the sample times `tSamples`.

`[q,qd,qdd,qddd,qdddd,pp,tPoints,tSamples] = minsnappolytraj( ___ ,Name=Value)` specifies options using one or more name-value pair arguments in addition to the input arguments from the previous syntax. For example, `minsnappolytraj(waypoints,timePoints,numSamples,VelocityBoundaryCondition=[1 0 -1 -1; 1 1 1 -1])` generates a two-dimensional minimum snap trajectory and specifies the velocity boundary conditions in each dimension for each waypoint.

`[q,qd,qdd,qddd,qdddd,pp,tPoints,tSamples] = minsnappolytraj( ___ ,TimeAllocation=true)` optimizes a combination of snap and the total segment time cost. In this case, the function treats `timePoints` as an initial guess for the time of arrival at the waypoints.

Examples

Compute Minimum Snap Trajectory for 2-D Planar Motion

Use the `minsnappolytraj` function with a given set of 2-D *xy* waypoints. Time points for the waypoints are also given.

```
wpts = [1 4 4 3 -2 0; 0 1 2 4 3 1];
tpts = 0:5;
```

Specify the number of samples in the output trajectory.

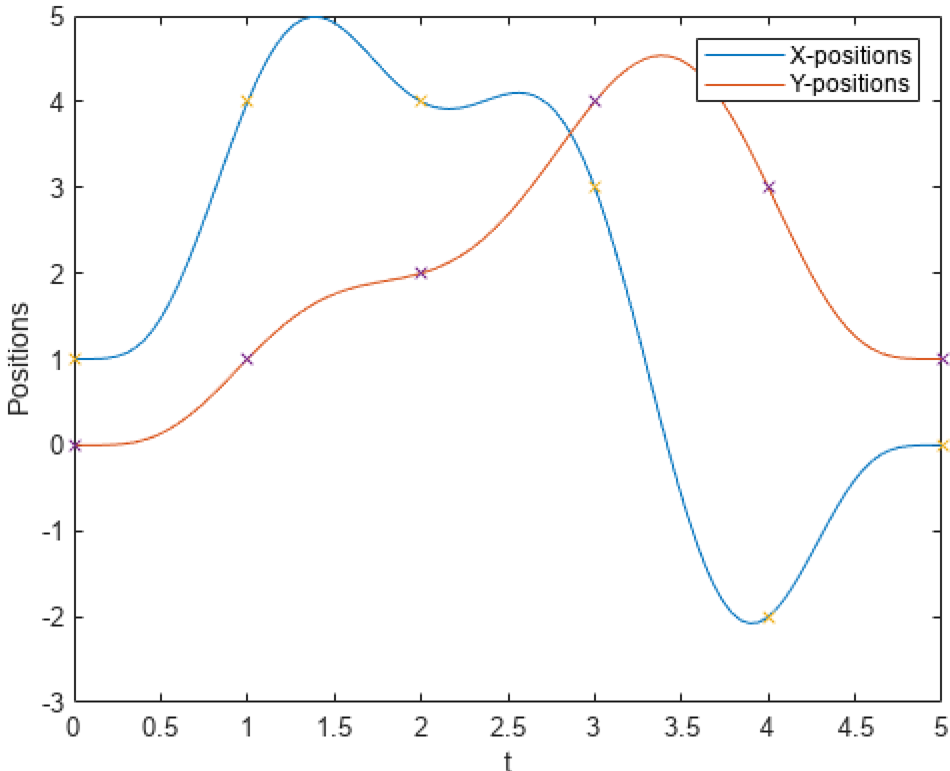
```
numsamples = 100;
```

Compute minimum snap trajectories. The function outputs the trajectory positions (`q`), velocity (`qd`), acceleration (`qdd`), jerks (`qddd`), and snaps (`qdddd`) at the given number of samples.

```
[q,qd,qdd,qddd,qdddd,pp,timepoints,tsamples] = minsnappolytraj(wpts,tpts,numsamples);
```

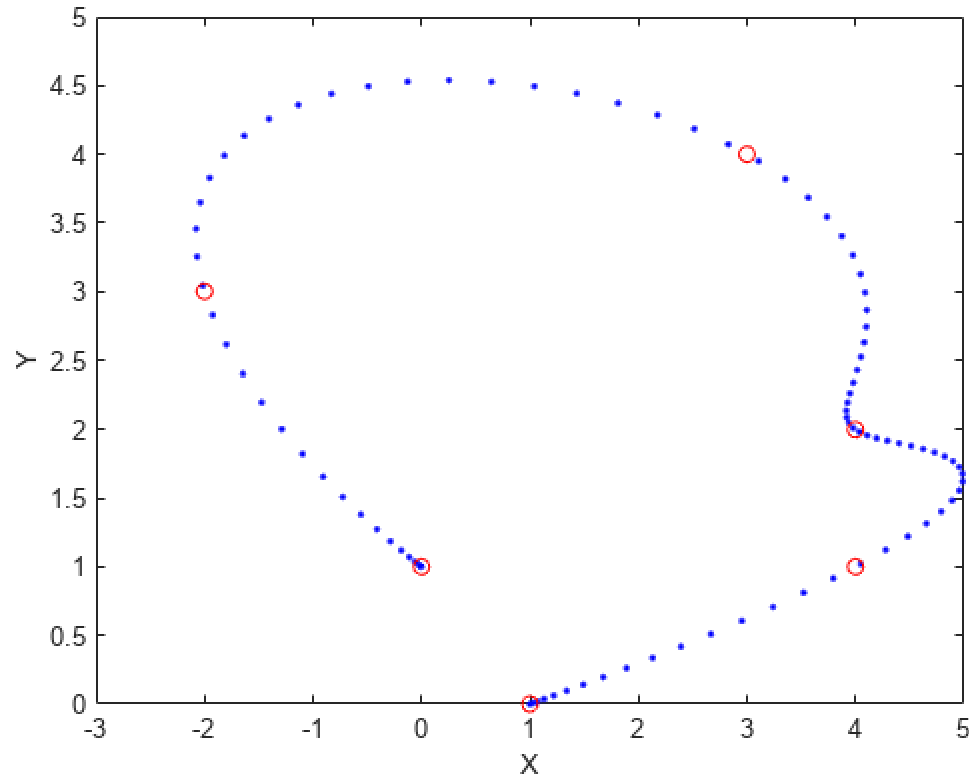
Plot the trajectories for the *x*- and *y*-positions. Compare the trajectory with each waypoint.

```
plot(tsamples,q)
hold on
plot(timepoints,wpts,'x')
xlabel('t')
ylabel('Positions')
legend('X-positions','Y-positions')
hold off
```



You can also verify the actual positions in the 2-D plane. Plot the separate rows of the `q` vector and the waypoints as *x*- and *y*- positions.

```
figure
plot(q(1,:),q(2,:),'.b',wpts(1,:),wpts(2,:),'or')
xlabel('X')
ylabel('Y')
```



Input Arguments

`waypoints` — Waypoints for trajectory
n-by-*p* matrix

Waypoints for the trajectory, specified as an *n*-by-*p* matrix. *n* is the dimension of the trajectory, and *p* is the number of waypoints.

Example: `[2 5 8 4; 3 4 10 12]`

Data Types: `single` | `double`

`timePoints` — Time points for waypoints of trajectory
p-element row vector

Time points for the waypoints of the trajectory, specified as a *p*-element row vector. *p* is the number of waypoints.

Example: `[1 2 3 5]`

Data Types: `single` | `double`

`numSamples` — Number of samples in output trajectory
positive integer

Number of samples in the output trajectory, specified as a positive integer.

Example: `50`

Data Types: `single` | `double`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Example: `minsnappolytraj(waypoints,timePoints,numSamples,VelocityBoundaryCondition=[1 0 -1 -1; 1 1 1 -1])` generates a two-dimensional minimum snap trajectory and specifies the velocity boundary conditions in each dimension for each waypoint.

`VelocityBoundaryCondition` — Velocity boundary conditions for each waypoint

n-by-*p* matrix

Velocity boundary conditions for each waypoint, specified as an *n*-by-*p* matrix. Each row sets the velocity boundary for the corresponding dimension of the trajectory at each of *p* waypoints. By default, the function uses a value of `0` at the boundary waypoints and `NaN` at the intermediate waypoints.

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

Data Types: `single` | `double`

`AccelerationBoundaryCondition` — Acceleration boundary conditions for each waypoint

n-by-*p* matrix

Acceleration boundary conditions for each waypoint, specified as an *n*-by-*p* matrix. Each row sets the acceleration boundary for the corresponding dimension of the trajectory at each of *p* waypoints. By default, the function uses a value of `0` at the boundary waypoints and `NaN` at the intermediate waypoints.

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

Data Types: `single` | `double`

`JerkBoundaryCondition` — Jerk boundary conditions for each waypoint

n-by-*p* matrix

Jerk boundary conditions for each waypoint, specified as an *n*-by-*p* matrix. Each row sets the jerk boundary for the corresponding dimension of the trajectory at each of *p* waypoints. By default, the function uses a value of `0` at the boundary waypoints and `NaN` at the intermediate waypoints.

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

Data Types: `single` | `double`

`SnapBoundaryCondition` — Snap boundary conditions for each waypoint

n-by-*p* matrix

Snap boundary conditions for each waypoint, specified as an *n*-by-*p* matrix. Each row sets the snap boundary for the corresponding dimension of the trajectory at each of *p* waypoints. By default, the function uses a value of `0` at the boundary waypoints and `NaN` at the intermediate waypoints.

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

Data Types: `single` | `double`

`TimeAllocation` — Time allocation flag
`false` or `0` (default) | `true` or `1`

Time allocation flag, specified as a logical `0` (`false`) or `1` (`true`). Enable this flag to optimize a combination of snap and total segment time cost.

Note If singularity occurs when the time allocation flag is enabled, reduce the `MaxSegmentTime` to `MinSegmentTime` ratio.

Example: `TimeAllocation=true`

Data Types: `logical`

`TimeWeight` — Weight for time allocation
`100` (default) | positive scalar

Weight for time allocation, specified as a positive scalar.

Example: `TimeWeight=120`

Data Types: `single` | `double`

`MinSegmentTime` — Minimum time segment length
`0.1` (default) | positive scalar | (*p*–1)-element row vector

Minimum time segment length, specified as a positive scalar or (*p*–1)-element row vector.

Example: `MinSegmentTime=0.2`

Data Types: `single` | `double`

`MaxSegmentTime` — Maximum time segment length
`1` (default) | positive scalar | (*p*–1)-element row vector

Maximum time segment length, specified as a positive scalar or (*p*–1)-element row vector

Example: `MaxSegmentTime=5`

Data Types: `single` | `double`

Output Arguments

q — Positions of trajectory
n-by-*m* matrix

Positions of the trajectory at the given time samples in `tSamples`, returned as an *n*-by-*m* matrix. *n* is the dimension of the trajectory, and *m* is equal to `numSamples`.

qd — Velocities of trajectory
n-by-*m* matrix

Velocities of the trajectory at the given time samples in `tSamples`, returned as an *n*-by-*m* matrix. *n* is the dimension of the trajectory, and *m* is equal to `numSamples`.

qdd — Accelerations of trajectory
n-by-*m* matrix

Accelerations of the trajectory at the given time samples in `tSamples`, returned as an *n*-by-*m* matrix. *n* is the dimension of the trajectory, and *m* is equal to `numSamples`.

qddd — Jerks of trajectory
n-by-*m* matrix

Jerks of the trajectory at the given time samples in `tSamples`, returned as an *n*-by-*m* matrix. *n* is the dimension of the trajectory, and *m* is equal to `numSamples`.

qdddd — Snaps of trajectory
n-by-*m* matrix

Snaps of the trajectory at the given time samples in `tSamples`, returned as an *n*-by-*m* matrix. *n* is the dimension of the trajectory, and *m* is equal to `numSamples`.

pp — Piecewise polynomial
structure

Piecewise-polynomial, returned as a structure that defines the polynomial for each section of the piecewise trajectory. You can build your own piecewise polynomials using `mkpp`, or evaluate the polynomial at specified times using `ppval`. The structure contains the fields:

- `form`: `'pp'`.
- `breaks`: *p*-element vector of times when the piecewise trajectory changes forms. *p* is the number of waypoints.
- `coefs`: *n*(*p*–`1`)-by-`order` matrix for the coefficients for the polynomials. *n*(*p*–`1`) is the dimension of the trajectory times the number of `pieces`. Each set of *n* rows defines the coefficients for the polynomial that described each variable trajectory.
- `pieces`: *p*–`1`. The number of breaks minus 1.
- `order`: Degree of the polynomial `+ 1`. The order of polynomial is 10.
- `dim`: *n*. The dimension of the control point positions.

`tPoints` — Time points for waypoints of trajectory
p-element row vector

Time points for the waypoints of the trajectory, returned as a *p*-element row vector. *p* is the number of waypoints.

`tSamples` — Time samples for trajectory
m-element row vector

Time samples for the trajectory, returned as an *m*-element row vector. Each element of the output position `q`, velocity `qd`, acceleration `qdd`, jerk `qddd`, and snap `qdddd` has been sampled at the corresponding time in this vector.

Version History

Introduced in R2021b

References

[1] Bry, Adam, Charles Richter, Abraham Bachrach, and Nicholas Roy. "Aggressive Flight of Fixed-Wing and Quadrotor Aircraft in Dense Indoor Environments." *The International Journal of Robotics Research*, 34, no. 7 (June 2015): 969–1002.

[2] Richter, Charles, Adam Bry, and Nicholas Roy. "Polynomial Trajectory Planning for Aggressive Quadrotor Flight in Dense Indoor Environments." *Paper presented at the International Symposium of Robotics Research (ISRR 2013)*, 2013.

Extended Capabilities

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

See Also

Functions
`minjerkpolytraj`

ned2lla

Transform local north-east-down coordinates to geodetic coordinates

Syntax

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

Description

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

Note

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

Examples

Transform NED Coordinates to Geodetic Coordinates

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

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

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

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

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

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

```
lla = 1×3
10^3 ×

    0.0460    0.0077    4.5254
```

Input Arguments

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

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

Example: `[-4556.3 -7134.8 -2852.4]`

Data Types: `double`

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

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

Example: `[46.017 7.750 1673]`

Data Types: `double`

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

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

The flat Earth transformation method has these limitations:

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

Data Types: `char` | `string`

Output Arguments

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

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

Data Types: `double`

Version History

Introduced in R2020b

Extended Capabilities

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

See Also

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

plotTransforms

Plot 3-D transforms from translations and rotations

Syntax

```
ax = plotTransforms(translations,rotations)
ax = plotTransforms(transformations)
ax = plotTransforms( ___ ,Name,Value)
```

Description

`ax = plotTransforms(translations,rotations)` draws transform frames in a 3-D figure window using the specified translations `translations`, and rotations, `rotations`. The *z*-axis always points upward.

`ax = plotTransforms(transformations)` draws transform frames for the specified SE(2) or SE(3) transformations, `transformations`.

`ax = plotTransforms( ___ ,Name,Value)` specifies additional options using name-value arguments. Specify multiple name-value arguments 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 | *N*-element array of `so2` or `so3` objects

Rotations of *xyz*-positions specified as a `quaternion` array, *N*-by-4 matrix of `[w x y z]` quaternion vectors, or an *N*-element array of `so2` or `so3` objects. *N* is the total number of rotations, and each element of the array, each row of the matrix or rotation transformation objects represent the rotation of the *xyz*-positions specified in `translations`.

If `rotations` is an *N*-element array of `so2` or `so3` objects, each element must be of the same type.

Example: `[1 1 1 0; 1 3 5 0]`

`transformations` — Transformation

`se2` object | `se3` object | *M*-element array of `se2` or `se3` objects

Transformations, specified as an `se2` object, an `se3` object, or an *M*-element array of `se2` or `se3` objects. *M* is the total number of transformations.

If you specify `transformations` as an array, each element must be of the same type.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `'FrameSize',5`

`FrameSize` — Size of frames and attached meshes

positive numeric scalar

Size of frame and attached meshes, specified as positive numeric scalar.

`FrameColor` — Color of frames

`"rgb"` (default) | RGB triplet | string scalar

Color of frames, specified as an RGB triplet or string scalar.

Example: `[0 0 1]` or `"green"`

`FrameAxisLabels` — *xyz* labels of coordinate frame

`"off"` (default) | `"on"`

xyz labels of the coordinate frame, specified as `"off"` to hide the labels or `"on"` to show the labels.

`FrameAxisLabels` — Frame axis labels

`""` (default) | `string` | *N*-element array of strings

Frame axis labels, specified as a string or *N*-element array of strings, where *N* is the total number of frames and each string corresponds to one frame at the same index of `transformations`, `translations`, or `rotations`.

`AxisLabels` — *xyz* labels of plotting axes

`"off"` (default) | `"on"`

xyz labels of the plotting axes, specified as `"off"` to hide the labels or `"on"`to show the labels.

`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 triplet or string scalar.

Example: `[0 0 1]` or `"green"`

`View` — Plot view
`"3D"` (default) | `"2D"` | three-element vector

Plot view, specified as `"3D"`, `"2D"`, or a three-element vector of the form [*x*,*y*,*z*] that sets the view angle in Cartesian coordinates. The magnitude of *x*,*y*, and *z* are ignored.

`Parent` — Axes used to plot transforms
`Axes` object | `UIAxes` object

Axes used to plot the pose graph, specified as the comma-separated pair consisting of `'Parent'` and either an `Axes` or `UIAxes` object. See `axes` or `uiaxes`.

Output Arguments

ax — Axes used to plot transforms
`Axes` object | `UIAxes` object

Axes used to plot the pose graph, specified as the comma-separated pair consisting of `'Parent'` and either an `Axes` or `UIAxes`object. See `axes` or `uiaxes`.

Version History

Introduced in R2018b

See Also

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

Objects
`se2` | `se3` | `so2` | `so3`

rmmavlinkkeys

Remove MAVLink key from MATLAB session

Syntax

```
rmmavlinkkeys(keyname)
```

Description

`rmmavlinkkeys(keyname)` removes the MAVLink key `keyname` from the current MATLAB session. Use MAVLink keys in signing channels for message signing. See the `mavlinksigning` object for more details.

Examples

Add and Remove MAVLink Keys

Check the contents of the `keys.env` file. The file contains two keys. `Key1` is a 32-element `uint8` vector and `Key2` is a 32-element `uint8` vector encoded using base 64 encoding using the `matlab.net.base64encode` function.

```
type keys.env
```

```
Key1 = [1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32]
Key2_Base64 = "1xOIT/+tTcjdXaCH794ihyStgzxDll+ZLw3SAzolIu0="
```

Add the MAVLink keys from the file `keys.env` file to the MATLAB session.

```
addmavlinkkeys("keys.env")
```

List all of the keys in the current MATLAB session.

```
keys = lsmavlinkkeys
```

```
keys = 1x2 string
    "Key1"    "Key2"
```

Remove both the `Key` and `Key_Base64` MAVLink keys.

```
rmmavlinkkeys(["Key1","Key2"])
```

Input Arguments

keyname — MAVLink key to remove from MATLAB session
string scalar | string array

MAVLink key to remove from the MATLAB session, specified as a string scalar, string array, or character vector. Specify a string array to remove multiple keys from the MATLAB session simultaneously.

Example: `rmmavlinkkeys(["Key1","Key2"])` removes the MAVLink keys `Key1` and `Key2` from the MATLAB session.

Data Types: `string` | `char`

Version History

Introduced in R2023a

See Also

`addmavlinkkeys` | `lsmavlinkkeys` | `mavlinksigning`

copyExampleSim3dProject

Copy support package files and plugins to specified folders

Syntax

```
sim3d.utils.copyExampleSim3dProject(DestFldr)
sim3d.utils.copyExampleSim3dProject(DestFldr,Name=Value)
```

Description

`sim3d.utils.copyExampleSim3dProject(DestFldr)` copies the UAV Toolbox Interface for Unreal Engine Projects support package project files to the destination folder, `DestFldr`. By default, `copyExampleSim3dProject` copies the plugins to your Epic Games® installation folder.

`sim3d.utils.copyExampleSim3dProject(DestFldr,Name=Value)` copies support package files to the destination with additional options specified by name-value arguments.

Running the `sim3d.utils.copyExampleSim3dProject` function configures your environment so that you can customize scenes. The support package contains these UAV Toolbox Interface for Unreal Engine Projects components.

- An Unreals project, defined in `AutoVrtlEnv.uproject`, and its associated files. The project includes editable versions of the prebuilt 3D scenes that you can select from the **Scene name** parameter of the Simulation 3D Scene Configuration block.
- Two plugins: `MathWorkSimulation`, `MathworksUAVContent`, and `RoadRunnerMaterials`. These plugins establish the connection between MATLAB and the Unreal Editor and are required for co-simulation.

Input Arguments

`DestFldr` — Destination folder for Unreal project files

character vector

Destination folder name, specified as a character vector.

Running `copyExampleSim3dProject` copies the Unreal project, defined in `AutoVrtlEnv.uproject`, and its associated files to the destination folder.

Note You must have write permission for the destination folder.

Example: `C:\project`

Data Types: `char` | `string`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Source — Support package source folder
character vector

Support package source folder, specified as a character vector. The folder contains the downloaded support packages files.

By default, if you do not specify the source folder, `copyExampleSim3dProject` copies the file from the support package installation folder, `matlabshared.supportpkg.getSupportPackageRoot()`.

Example: `Source="shared\sim3dprojects\spkg\"`

Data Types: `char` | `string`

PluginDestination — Option to change the plugin destination folder
character vector

Option to change the plugin destination folder, specified as a character vector.

By default, if you do not change the plugin installation folder location, `copyExampleSim3dProject` tries to copy the plugins to `C:\Program Files\Epic Games\UE_4.27\Engine\Plugins\MathWorks`.

Example: `PluginDestination="C:\Program Files\Epic Games\UE_4.27\Engine\Plugins\MathWorks"`

Data Types: `char` | `string`

VerboseOutput — Option to enable verbose logging
`0` or `false` (default) | `1` or `true`

Option to enable verbose logging, specified as a logical `0` (false) or `1` (true). Verbose logging displays intermediate iteration information on the MATLAB command line.

Example: `VerboseOutput=true`

Data Types: `logical`

Examples

Copy Support Package Files to Destination Folder

Copy the support package files to `C:\project`.

```
sim3d.utils.copyExampleSim3dProject("C:\project");
```

Copy the support package files to `C:\project` with `VerboseOutput` set to `true`.

```
sim3d.utils.copyExampleSim3dProject("C:\project", VerboseOutput=true)
```

```
Copying ...\spkg\project\AutoVrtlEnv to C:\project\AutoVrtlEnv
Creating C:\project\AutoVrtlEnv\Plugins
Copying ...\spkg\plugins\mw_aerospace\MathWorksAerospace to C:\project\AutoVrtlEnv\Plugins\MathWo
Copying ...\spkg\plugins\mw_automotive\MathWorksAutomotiveContent to C:\project\AutoVrtlEnv\Plugi
Copying ...\spkg\plugins\mw_simulation\MathWorksSimulation to C:\project\AutoVrtlEnv\Plugins\Math
Copying ...\spkg\plugins\mw_uav\MathWorksUAVContent to C:\project\AutoVrtlEnv\Plugins\MathWorksU
Copying ...\spkg\plugins\rr_materials\RoadRunnerMaterials to C:\project\AutoVrtlEnv\Plugins\Road
Ensuring C:\project\AutoVrtlEnv\AutoVrtlEnv.uproject is writable
```

```
Enabling plugin MathWorksSimulation in C:\project\AutoVrtlEnv\AutoVrtlEnv.uproject
Enabling plugin MathWorksUAVContent in C:\project\AutoVrtlEnv\AutoVrtlEnv.uproject
Enabling plugin MathWorksAutomotiveContent in C:\project\AutoVrtlEnv\AutoVrtlEnv.uproject
Enabling plugin RoadRunnerMaterials in C:\project\AutoVrtlEnv\AutoVrtlEnv.uproject
```

Version History

Introduced in R2022b

See Also

Topics

"Install Support Package for Customizing Scenes"
"How Unreal Engine Simulation for UAVs Works"
"Unreal Engine Simulation Environment Requirements and Limitations"

External Websites

Unreal Engine
Using Unreal Engine with Simulink

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]`

Version History

Introduced in R2015a

Extended Capabilities

C/C++ Code Generation

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

See Also

`axang2quat` | `quaternion`

quat2eul

Convert quaternion to Euler angles

Syntax

```
eul = quat2eul(quat)
eul = quat2eul(quat,sequence)
[eul,eulAlt] = quat2eul( ___ )
```

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"`.

`[eul,eulAlt] = quat2eul( ___ )` also returns an alternate set of Euler angles that represents the same rotation `eulAlt`.

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]`

`eulAlt` — Alternate Euler rotation angle solution
n-by-3 matrix

Alternate Euler rotation angle solution 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]`

Version History

Introduced in R2015a

R2020a: Alternate Euler angle output

`quat2eul` now optionally outputs an alternate set of Euler angles that also represent the same rotation as the original output Euler angles.

Extended Capabilities

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

See Also

`eul2quat` | `quaternion`

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]`

Version History

Introduced in R2015a

Extended Capabilities

C/C++ Code Generation

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

See Also

`rotm2quat` | `quaternion` | `so2` | `so3`

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]`

Version History

Introduced in R2015a

Extended Capabilities

C/C++ Code Generation

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

See Also

`tform2quat` | `quaternion` | `se2` | `se3`

removeCustomTerrain

Remove custom terrain data

Syntax

```
removeCustomTerrain(terrainName)
```

Description

`removeCustomTerrain(terrainName)` removes the custom terrain data specified by the user-defined `terrainName`. You can use this function to remove terrain data that is no longer needed. The terrain data to be removed must have been previously added using `addCustomTerrain`.

Input Arguments

`terrainName` — User-defined identifier for terrain data
string scalar | character vector

User-defined identifier for terrain data previously added using `addCustomTerrain`, specified as a string scalar or a character vector.

Data Types: `char` | `string`

Version History

Introduced in R2021a

See Also

`addCustomTerrain`

rotm2axang

Convert rotation matrix to axis-angle rotation

Syntax

```
axang = rotm2axang(rotm)
```

Description

`axang = rotm2axang(rotm)` converts a rotation given as an orthonormal rotation matrix, `rotm`, to the corresponding axis-angle representation, `axang`. The input rotation matrix must be in the premultiply form for rotations.

Examples

Convert Rotation Matrix to Axis-Angle Rotation

```
rotm = [1 0 0 ; 0 -1 0; 0 0 -1];
axang = rotm2axang(rotm)
```

```
axang = 1×4

    1.0000         0         0    3.1416
```

Input Arguments

`rotm` — Rotation matrix
3-by-3-by-*n* matrix

Rotation matrix, specified as a 3-by-3-by-*n* matrix containing *n* rotation matrices. Each rotation matrix has a size of 3-by-3 and must be orthonormal. The input rotation matrix must be in the premultiply form for rotations.

Note Rotation matrices that are slightly non-orthonormal can give complex outputs. Consider validating your matrix before inputting to the function.

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

Output Arguments

axang — Rotation given in axis-angle form
n-by-4 matrix

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

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

Version History

Introduced in R2015a

Extended Capabilities

C/C++ Code Generation

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

See Also

`axang2rotm` | `so3`

rotm2eul

Convert rotation matrix to Euler angles

Syntax

```
eul = rotm2eul(rotm)
eul = rotm2eul(rotm,sequence)
[eul,eulAlt] = rotm2eul( ___ )
```

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"`.

`[eul,eulAlt] = rotm2eul( ___ )` also returns an alternate set of Euler angles that represents the same rotation `eulAlt`.

Examples

Convert Rotation Matrix to Euler Angles

```
rotm = [0 0 1; 0 1 0; -1 0 0];
eulZYX = rotm2eul(rotm)
```

```
eulZYX = 1×3

         0    1.5708         0
```

Convert Rotation Matrix to Euler Angles Using ZYZ Axis Order

```
rotm = [0 0 1; 0 1 0; -1 0 0];
eulZYZ = rotm2eul(rotm,'ZYZ')
```

```
eulZYZ = 1×3

   -3.1416   -1.5708   -3.1416
```

Input Arguments

`rotm` — Rotation matrix

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

Rotation matrix, specified as a 3-by-3-by-*n* matrix containing *n* rotation matrices. Each rotation matrix has a size of 3-by-3 and is orthonormal. The input rotation matrix must be in the premultiply form for rotations.

Note Rotation matrices that are slightly non-orthonormal can give complex outputs. Consider validating your matrix before inputting to the function.

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

`sequence` — Axis-rotation sequence

`"ZYX"` (default) | `"ZYZ"` | `"ZXY"` | `"ZXZ"` | `"YXY"` | `"YZX"` | `"YXZ"` | `"YZY"` | `"XYX"` | `"XYZ"` | `"XZX"` | `"XZY"`

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

- `"ZYX"` (default)
- `"ZYZ"`
- `"ZXY"`
- `"ZXZ"`
- `"YXY"`
- `"YZX"`
- `"YXZ"`
- `"YZY"`
- `"XYX"`
- `"XYZ"`
- `"XZX"`
- `"XZY"`

Each character indicates the corresponding axis. For example, if the sequence is `"ZYX"`, then the three specified Euler angles are interpreted in order as a rotation around the *z*-axis, a rotation around the *y*-axis, and a rotation around the *x*-axis. When applying this rotation to a point, it will apply the axis rotations in the order *x*, then *y*, then *z*.

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]`

`eulAlt` — Alternate Euler rotation angle solution
n-by-3 matrix

Alternate Euler rotation angle solution 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]`

Version History

Introduced in R2015a

R2020a: Alternate Euler angle output

`rotm2eul` now optionally outputs an alternate set of Euler angles `eulAlt` that also represent the same rotation as the original output Euler angles `eul`. So if you use `eul` or `eulAlt` to rotate a point, the resulting point is the same.

R2023a: Additional Euler sequence support

`rotm2eul` supports additional Euler sequences for the `sequences` argument. These are all the supported Euler sequences:

- `"ZYX"`
- `"ZYZ"`
- `"ZXY"`
- `"ZXZ"`
- `"YXY"`
- `"YZX"`
- `"YXZ"`
- `"YZY"`
- `"XYX"`
- `"XYZ"`
- `"XZX"`
- `"XZY"`

Extended Capabilities

C/C++ Code Generation

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

See Also

`eul2rotm` | `so2` | `so3`

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]`

Version History

Introduced in R2015a

Extended Capabilities

C/C++ Code Generation

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

See Also

`quat2rotm` | `so3` | `quaternion`

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 by the coordinates to be transformed (as opposed to postmultiplying).

Examples

Convert Rotation Matrix to Homogeneous Transformation

```
rotm = [1 0 0 ; 0 -1 0; 0 0 -1];
tform = rotm2tform(rotm)
```

```
tform = 4×4

     1     0     0     0
     0    -1     0     0
     0     0    -1     0
     0     0     0     1
```

Input Arguments

`rotm` — Rotation matrix
2-by-2-by-*n* array | 3-by-3-by-*n* array

Rotation matrix, specified as a 2-by-2-by-*n* or a 3-by-3-by-*n* array containing *n* rotation matrices. Each rotation matrix is either 2-by-2 or 3-by-3 and is orthonormal. The input rotation matrix must be in the premultiplied form for rotations.

Note Rotation matrices that are not orthonormal can be normalized with the `normalize` function.

2-D rotation matrices are of this form:

3-D rotation matrices are of this form:

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

Output Arguments

`tform` — Homogeneous transformation
3-by-3-by-*n* array | 4-by-4-by-*n* array

Homogeneous transformation, returned as a 3-by-3-by-*n* array or 4-by-4-by-*n* array. *n* is the number of homogeneous transformations. When using the transformation matrix, premultiply it by the coordinates to be transformed (as opposed to postmultiplying).

2-D homogeneous transformation matrices are of this form:

$$T = \begin{bmatrix} r_{11} & r_{12} & t_1 \\ r_{21} & r_{22} & t_2 \\ 0 & 0 & 1 \end{bmatrix}$$

3-D homogeneous transformation matrices are of this form:

$$T = \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_1 \\ r_{21} & r_{22} & r_{23} & t_2 \\ r_{31} & r_{32} & r_{33} & t_3 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

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

More About

2-D Homogeneous Transformation Matrix

2-D homogeneous transformation matrices consist of both an SO(2) rotation and an *xy*-translation.

To read more about SO(2) rotations, see the “2-D Orthonormal Rotation Matrix” on page 1-133 section of the `so2` object.

The translation is along the *x*-, *y*-, and *z*-axes as a three-element column vector:

$$t = \begin{bmatrix} x \\ y \end{bmatrix}$$

The SO(2) rotation matrix *R* is applied to the translation vector *t* to create the homogeneous translation matrix *T*. The rotation matrix is present in the upper-left of the transformation matrix as 2-by-2 submatrix, and the translation vector is present as a two-element vector in the last column.

$$T = \begin{bmatrix} R & t \\ 0_{1\times 2} & 1 \end{bmatrix} = \begin{bmatrix} I_2 & t \\ 0_{1\times 2} & 1 \end{bmatrix} \cdot \begin{bmatrix} R & 0 \\ 0_{1\times 2} & 1 \end{bmatrix}$$

3-D Homogeneous Transformation Matrix

3-D homogeneous transformation matrices consist of both an SO(3) rotation and an *xyz*-translation.

To read more about SO(3) rotations, see the “3-D Orthonormal Rotation Matrix” on page 1-129 section of the `so3` object.

The translation is along the *x*-, *y*-, and *z*-axes as a three-element column vector:

$$t = \begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

The SO(3) rotation matrix R is applied to the translation vector t to create the homogeneous translation matrix T. The rotation matrix is present in the upper-left of the transformation matrix as 3-by-3 submatrix, and the translation vector is present as a three-element vector in the last column.

$$T = \begin{bmatrix} R & t \\ 0_{1x3} & 1 \end{bmatrix} = \begin{bmatrix} I_3 & t \\ 0_{1x3} & 1 \end{bmatrix} \cdot \begin{bmatrix} R & 0 \\ 0_{1x3} & 1 \end{bmatrix}$$

Version History

Introduced in R2015a

R2023a: `rotm2tform` Supports 2-D Rotation Matrices

The `rotm` argument now accepts 2-D rotation matrices as a 2-by-2-by-*n* array and `rotm2tform` outputs 2-D transformation matrices as a 3-by-3-by-*n* array.

Extended Capabilities

C/C++ Code Generation

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

See Also

`tform2rotm` | `se2` | `se3` | `so2` | `so3`

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]`

Version History

Introduced in R2015a

Extended Capabilities

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

See Also

`axang2tform` | `se3`

tform2eul

Extract Euler angles from homogeneous transformation

Syntax

```
eul = tform2eul(tform)
eul = tform2eul(tform, sequence)
[eul,eulAlt] = tform2eul( ___ )
```

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"`.

`[eul,eulAlt] = tform2eul( ___ )` also returns an alternate set of Euler angles that represents the same rotation `eulAlt`.

Examples

Extract Euler Angles from Homogeneous Transformation Matrix

```
tform = [1 0 0 0.5; 0 -1 0 5; 0 0 -1 -1.2; 0 0 0 1];
eulZYX = tform2eul(tform)
```

```
eulZYX = 1×3

         0         0    3.1416
```

Extract Euler Angles from Homogeneous Transformation Matrix Using ZYZ Rotation

```
tform = [1 0 0 0.5; 0 -1 0 5; 0 0 -1 -1.2; 0 0 0 1];
eulZYZ = tform2eul(tform,'ZYZ')
```

```
eulZYZ = 1×3

         0   -3.1416    3.1416
```

Input Arguments

`tform` — Homogeneous transformation

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

Homogeneous transformation, specified by a 4-by-4-by-*n* matrix of *n* homogeneous transformations. The input homogeneous transformation must be in the premultiply form for transformations.

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

`sequence` — Axis-rotation sequence

`"ZYX"` (default) | `"ZYZ"` | `"ZXY"` | `"ZXZ"` | `"YXY"` | `"YZX"` | `"YXZ"` | `"YZY"` | `"XYX"` | `"XYZ"` | `"XZX"` | `"XZY"`

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

- `"ZYX"` (default)
- `"ZYZ"`
- `"ZXY"`
- `"ZXZ"`
- `"YXY"`
- `"YZX"`
- `"YXZ"`
- `"YZY"`
- `"XYX"`
- `"XYZ"`
- `"XZX"`
- `"XZY"`

Each character indicates the corresponding axis. For example, if the sequence is `"ZYX"`, then the three specified Euler angles are interpreted in order as a rotation around the *z*-axis, a rotation around the *y*-axis, and a rotation around the *x*-axis. When applying this rotation to a point, it will apply the axis rotations in the order *x*, then *y*, then *z*.

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]`

`eulAlt` — Alternate Euler rotation angle solution

n-by-3 matrix

Alternate Euler rotation angle solution 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]

Version History

Introduced in R2015a

R2020a: Alternate Euler angle output

`tform2eul` now optionally outputs an alternate set of Euler angles `eulAlt` that also represent the same rotation as the original output Euler angles `eul`. So if you use `eul` or `eulAlt` to rotate a point, the resulting point is the same.

R2023a: Additional Euler sequence support

`tform2eul` supports additional Euler sequences for the `sequences` argument. These are all the supported Euler sequences:

- `"ZYX"`
- `"ZYZ"`
- `"ZXY"`
- `"ZXZ"`
- `"YXY"`
- `"YZX"`
- `"YXZ"`
- `"YZY"`
- `"XYX"`
- `"XYZ"`
- `"XZX"`
- `"XZY"`

Extended Capabilities

C/C++ Code Generation

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

See Also

`eul2tform` | `se2` | `se3`

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]`

Version History

Introduced in R2015a

Extended Capabilities

C/C++ Code Generation

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

See Also

`quat2tform` | `se3` | `quaternion`

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
3-by-3-by-*n* array | 4-by-4-by-*n* array

Homogeneous transformation, specified as a 3-by-3-by-*n* array or 4-by-4-by-*n* array. *n* is the number of homogeneous transformations. The input homogeneous transformation must be in the premultiplied form for transformations.

2-D homogeneous transformation matrices are of the form:

$$T = \begin{bmatrix} r_{11} & r_{12} & t_1 \\ r_{21} & r_{22} & t_2 \\ 0 & 0 & 1 \end{bmatrix}$$

3-D homogeneous transformation matrices are of the form:

$$T = \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_1 \\ r_{21} & r_{22} & r_{23} & t_2 \\ r_{31} & r_{32} & r_{33} & t_3 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

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

Output Arguments

`rotm` — Rotation matrix
2-by-2-by-*n* array | 3-by-3-by-*n* array

Rotation matrix, returned as a 2-by-2-*n* array or 3-by-3-by-*n* array containing *n* rotation matrices. Each rotation matrix in the array has either a size of 2-by-2 or 3-by-3 and is orthonormal. When using the rotation matrix, premultiply it with the coordinates to be rotated (as opposed to postmultiplying).

2-D rotation matrices are of the form:

$$rotation = \begin{bmatrix} r_{11} & r_{12} \\ r_{21} & r_{22} \end{bmatrix}$$

3-D rotation matrices are of the form:

$$rotation = \begin{bmatrix} r_{11} & r_{12} & r_{13} \\ r_{11} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{bmatrix}$$

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

More About

Homogeneous Transformation Matrices

Homogeneous transformation matrices consist of both an orthogonal rotation and a translation.

2-D Transformations

2-D transformations have a rotation θ about the z-axis:

$$R_z(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}$$

, and a translation along the x and y axis:

$$t = \begin{bmatrix} x \\ y \end{bmatrix}$$

, resulting in the 2-D transformation matrix of the form:

$$T = \begin{bmatrix} R & t \\ 0_{1\times 2} & 1 \end{bmatrix} = \begin{bmatrix} I_2 & t \\ 0_{1\times 2} & 1 \end{bmatrix} \cdot \begin{bmatrix} R & 0 \\ 0_{1\times 2} & 1 \end{bmatrix}$$

3-D Transformations

3-D transformations contain information about three rotations about the x-, y-, and z-axes:

$$R_x(\phi) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\phi & \sin\phi \\ 0 & -\sin\phi & \cos\phi \end{bmatrix}, R_y(\psi) = \begin{bmatrix} \cos\psi & 0 & \sin\psi \\ 0 & 1 & 0 \\ -\sin\psi & 0 & \cos\psi \end{bmatrix}, R_z(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

and after multiplying become the rotation about the xyz-axes:

$$R_{xyz} = R_x(\phi)R_y(\psi)R_z(\theta) =$$
$$\begin{bmatrix} \cos\phi\cos\psi\cos\theta - \sin\phi\sin\theta & -\cos\phi\cos\psi\sin\theta - \sin\phi\cos\theta & \cos\phi\sin\psi \\ \sin\phi\cos\psi\cos\theta + \cos\phi\sin\theta & -\sin\phi\cos\psi\sin\theta + \cos\phi\cos\theta & \sin\phi\sin\psi \\ -\sin\psi\cos\theta & \sin\psi\sin\theta & \cos\psi \end{bmatrix}$$

and a translation along the x-, y-, and z-axis:

$$t = \begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

, resulting in the 3-D transformation matrix of the form:

$$T = \begin{bmatrix} R & t \\ 0_{1x3} & 1 \end{bmatrix} = \begin{bmatrix} I_3 & t \\ 0_{1x3} & 1 \end{bmatrix} \cdot \begin{bmatrix} R & 0 \\ 0_{1x3} & 1 \end{bmatrix}$$

Version History

Introduced in R2015a

R2023a: `tform2rotm` Supports 2-D Homogeneous Transformation Matrices

The `tform` argument now accepts 2-D homogeneous transformation matrices as a 3-by-3-by-*n* array and `tform2rotm` outputs 2-D rotation matrices 2-by-2-by-*n* array.

Extended Capabilities

C/C++ Code Generation

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

See Also

`rotm2tform` | `se2` | `se3` | `so2` | `so3`

tform2trvec

Extract translation vector from homogeneous transformation

Syntax

```
trvec = tform2trvec(tform)
```

Description

`trvec = tform2trvec(tform)` extracts the Cartesian representation of the translation vector `trvec` from the homogeneous transformation `tform`. The rotational components of `tform` are ignored. The input homogeneous transformation must be in the premultiplied 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
3-by-3-by-*n* array | 4-by-4-by-*n* array

Homogeneous transformation, specified as a 3-by-3-by-*n* array or 4-by-4-by-*n* array. *n* is the number of homogeneous transformations. The input homogeneous transformation must be in the premultiplied form for transformations.

2-D homogeneous transformation matrices are of the form:

$$T = \begin{bmatrix} r_{11} & r_{12} & t_1 \\ r_{21} & r_{22} & t_2 \\ 0 & 0 & 1 \end{bmatrix}$$

3-D homogeneous transformation matrices are of the form:

$$T = \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_1 \\ r_{21} & r_{22} & r_{23} & t_2 \\ r_{31} & r_{32} & r_{33} & t_3 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

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

Output Arguments

`trvec` — Cartesian representation of translation vector
n-by-2 matrix | *n*-by-3 matrix

Cartesian representation of a translation vector, returned as an *n*-by-2 matrix if `tform` is a 3-by-3-by-*n* array and an *n*-by-3 matrix if `tform` is a 4-by-4-by-*n* array. *n* is the number of translation vectors. Each vector is of the form [*x y*] or [*x y z*].

Example: `[0.5 6 100]`

More About

Homogeneous Transformation Matrices

Homogeneous transformation matrices consist of both an orthogonal rotation and a translation.

2-D Transformations

2-D transformations have a rotation θ about the z-axis:

$$R_z(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}$$

, and a translation along the *x* and *y* axis:

$$t = \begin{bmatrix} x \\ y \end{bmatrix}$$

, resulting in the 2-D transformation matrix of the form:

$$T = \begin{bmatrix} R & t \\ 0_{1\times 2} & 1 \end{bmatrix} = \begin{bmatrix} I_2 & t \\ 0_{1\times 2} & 1 \end{bmatrix} \cdot \begin{bmatrix} R & 0 \\ 0_{1\times 2} & 1 \end{bmatrix}$$

3-D Transformations

3-D transformations contain information about three rotations about the x-, y-, and z-axes:

$$R_x(\phi) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\phi & \sin\phi \\ 0 & -\sin\phi & \cos\phi \end{bmatrix}, R_y(\psi) = \begin{bmatrix} \cos\psi & 0 & \sin\psi \\ 0 & 1 & 0 \\ -\sin\psi & 0 & \cos\psi \end{bmatrix}, R_z(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

and after multiplying become the rotation about the xyz-axes:

$$R_{xyz} = R_x(\phi)R_y(\psi)R_z(\theta) =$$
$$\begin{bmatrix} \cos\phi\cos\psi\cos\theta - \sin\phi\sin\theta & -\cos\phi\cos\psi\sin\theta - \sin\phi\cos\theta & \cos\phi\sin\psi \\ \sin\phi\cos\psi\cos\theta + \cos\phi\sin\theta & -\sin\phi\cos\psi\sin\theta + \cos\phi\cos\theta & \sin\phi\sin\psi \\ -\sin\psi\cos\theta & \sin\psi\sin\theta & \cos\psi \end{bmatrix}$$

and a translation along the *x*-, *y*-, and *z*-axis:

$$t = \begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

, resulting in the 3-D transformation matrix of the form:

$$T = \begin{bmatrix} R & t \\ 0_{1x3} & 1 \end{bmatrix} = \begin{bmatrix} I_3 & t \\ 0_{1x3} & 1 \end{bmatrix} \cdot \begin{bmatrix} R & 0 \\ 0_{1x3} & 1 \end{bmatrix}$$

Version History

Introduced in R2015a

R2023a: `tform2trvec` Supports 2-D Homogeneous Transformation Matrices

The `tform` argument now accepts 2-D homogeneous transformation matrices as a 3-by-3-by-*n* array and `tform2trvec` outputs a *n*-by-2 matrix of 2-D translation vectors.

Extended Capabilities

C/C++ Code Generation

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

See Also

`trvec2tform` | `se2` | `se3`

trvec2tform

Convert translation vector to homogeneous transformation

Syntax

```
tform = trvec2tform(trvec)
```

Description

`tform = trvec2tform(trvec)` converts the Cartesian representation of the translation vector `trvec` to the corresponding homogeneous transformation `tform`. When using the transformation matrix, premultiply it by 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 translation vector

n-by-2 matrix | *n*-by-3 matrix

Cartesian representation of a translation vector, specified as an *n*-by-2 matrix if `tform` is a 3-by-3-by-*n* array and an *n*-by-3 matrix if `tform` is a 4-by-4-by-*n* array. *n* is the number of translation vectors. Each vector is of the form [*x y*] or [*x y z*].

Example: `[0.5 6 100]`

Output Arguments

`tform` — Homogeneous transformation

3-by-3-by-*n* array | 4-by-4-by-*n* array

Homogeneous transformation, returned as a 3-by-3-by-*n* array or 4-by-4-by-*n* array. *n* is the number of 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]`

2-D homogeneous transformation matrices are of the form:

$$T = \begin{bmatrix} r_{11} & r_{12} & t_1 \\ r_{21} & r_{22} & t_2 \\ 0 & 0 & 1 \end{bmatrix}$$

3-D homogeneous transformation matrices are of the form:

$$T = \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_1 \\ r_{21} & r_{22} & r_{23} & t_2 \\ r_{31} & r_{32} & r_{33} & t_3 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

More About

Homogeneous Transformation Matrices

Homogeneous transformation matrices consist of both an orthogonal rotation and a translation.

2-D Transformations

2-D transformations have a rotation θ about the z-axis:

$$R_z(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}$$

, and a translation along the x and y axis:

$$t = \begin{bmatrix} x \\ y \end{bmatrix}$$

, resulting in the 2-D transformation matrix of the form:

$$T = \begin{bmatrix} R & t \\ 0_{1 \times 2} & 1 \end{bmatrix} = \begin{bmatrix} I_2 & t \\ 0_{1 \times 2} & 1 \end{bmatrix} \cdot \begin{bmatrix} R & 0 \\ 0_{1 \times 2} & 1 \end{bmatrix}$$

3-D Transformations

3-D transformations contain information about three rotations about the x-, y-, and z-axes:

$$R_x(\phi) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\phi & \sin\phi \\ 0 & -\sin\phi & \cos\phi \end{bmatrix}, R_y(\psi) = \begin{bmatrix} \cos\psi & 0 & \sin\psi \\ 0 & 1 & 0 \\ -\sin\psi & 0 & \cos\psi \end{bmatrix}, R_z(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

and after multiplying become the rotation about the xyz-axes:

$$R_{xyz} = R_x(\phi)R_y(\psi)R_z(\theta) =$$
$$\begin{bmatrix} \cos\phi\cos\psi\cos\theta - \sin\phi\sin\theta & -\cos\phi\cos\psi\sin\theta - \sin\phi\cos\theta & \cos\phi\sin\psi \\ \sin\phi\cos\psi\cos\theta + \cos\phi\sin\theta & -\sin\phi\cos\psi\sin\theta + \cos\phi\cos\theta & \sin\phi\sin\psi \\ -\sin\psi\cos\theta & \sin\psi\sin\theta & \cos\psi \end{bmatrix}$$

and a translation along the x-, y-, and z-axis:

$$t = \begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

, resulting in the 3-D transformation matrix of the form:

$$T = \begin{bmatrix} R & t \\ 0_{1x3} & 1 \end{bmatrix} = \begin{bmatrix} I_3 & t \\ 0_{1x3} & 1 \end{bmatrix} \cdot \begin{bmatrix} R & 0 \\ 0_{1x3} & 1 \end{bmatrix}$$

Version History

Introduced in R2015a

R2023a: `trvec2tform` Supports 2-D Translation Vectors

The `trvec` argument now accepts 2-D translation vectors as a *n*-by-2 matrix and `trvec2tform` outputs 2-D homogeneous transformation matrices as a 3-by-3-by-*n* array.

Extended Capabilities

C/C++ Code Generation

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

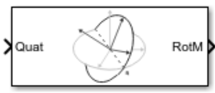
See Also

`tform2trvec` | `se2` | `se3`

Blocks

Coordinate Transformation Conversion

Convert to a specified coordinate transformation representation



Libraries:
Robotics System Toolbox / Utilities
Navigation Toolbox / Utilities
ROS Toolbox / Utilities
UAV Toolbox / Utilities

Description

The Coordinate Transformation Conversion block converts a coordinate transformation from the input representation to a specified output representation. The input and output representations use the following forms:

- Axis-Angle (`AxAng`) – `[x y z theta]`
- Euler Angles (`Eul`) – `[z y x]`, `[z y z]`, or `[x y z]`
- Homogeneous Transformation (`TForm`) – 4-by-4 matrix
- Quaternion (`Quat`) – `[w x y z]`
- Rotation Matrix (`RotM`) – 3-by-3 matrix
- Translation Vector (`TrVec`) – `[x y z]`

All vectors must be **column vectors**.

To accommodate representations that only contain position or orientation information (`TrVec` or `Eul`, for example), you can specify two inputs or outputs to handle all transformation information. When you select the Homogeneous Transformation as an input or output, an optional `Show TrVec input/output port` parameter can be selected on the block mask to toggle the multiple ports.

Ports

Input

Input transformation — Coordinate transformation
column vector | 3-by-3 matrix | 4-by-4 matrix

Input transformation, specified as a coordinate transformation. The following representations are supported:

- Axis-Angle (`AxAng`) – `[x y z theta]`
- Euler Angles (`Eul`) – `[z y x]`, `[z y z]`, or `[x y z]`
- Homogeneous Transformation (`TForm`) – 4-by-4 matrix
- Quaternion (`Quat`) – `[w x y z]`
- Rotation Matrix (`RotM`) – 3-by-3 matrix
- Translation Vector (`TrVec`) – `[x y z]`

All vectors must be **column vectors**.

To accommodate representations that only contain position or orientation information (`TrVec` or `Eul`, for example), you can specify two inputs or outputs to handle all transformation information. When you select the Homogeneous Transformation as an input or output, an optional `Show TrVec input/output port` parameter can be selected on the block mask to toggle the multiple ports.

TrVec — Translation vector
3-element column vector

Translation vector, specified as a 3-element column vector, `[x y z]`, which corresponds to a translation in the *x*, *y*, and *z* axes respectively. This port can be used to input or output the translation information separately from the rotation vector.

Dependencies

You must select Homogeneous Transformation (`TForm`) for the opposite transformation port to get the option to show the additional `TrVec` port. Enable the port by clicking `Show TrVec input/output port`.

Output Arguments

Output transformation — Coordinate transformation
column vector | 3-by-3 matrix | 4-by-4 matrix

Output transformation, returned as a coordinate transformation with the specified representation. The following representations are supported:

- Axis-Angle (`AxAng`) – `[x y z theta]`
- Euler Angles (`Eul`) – `[z y x]`, `[z y z]`, or `[x y z]`
- Homogeneous Transformation (`TForm`) – 4-by-4 matrix
- Quaternion (`Quat`) – `[w x y z]`
- Rotation Matrix (`RotM`) – 3-by-3 matrix
- Translation Vector (`TrVec`) – `[x y z]`

To accommodate representations that only contain position or orientation information (`TrVec` or `Eul`, for example), you can specify two inputs or outputs to handle all transformation information. When you select the Homogeneous Transformation as an input or output, an optional `Show TrVec input/output port` parameter can be selected on the block mask to toggle the multiple ports.

TrVec — Translation vector
three-element column vector

Translation vector, returned as a three-element column vector, `[x y z]`, which corresponds to a translation in the *x*, *y*, and *z* axes respectively. This port can be used to input or output the translation information separately from the rotation vector.

Dependencies

You must select Homogeneous Transformation (`TForm`) for the opposite transformation port to get the option to show the additional `TrVec` port. Enable the port by clicking `Show TrVec input/output port`.

Parameters

Representation — Input or output representation
Axis-Angle | Euler Angles | Homogeneous Transformation | Rotation Matrix | Translation Vector | Quaternion

Select the representation for both the input and output port for the block. If you are using a transformation with only orientation information, you can also select the Show TrVec input/output port when converting to or from a homogeneous transformation.

Axis rotation sequence — Order of Euler angle axis rotations
ZYX (default) | ZYZ | XYZ

Order of the Euler angle axis rotations, specified as ZYX, ZYZ, or XYZ. The order of the angles in the input or output port Eul must match this rotation sequence. The default order ZYX specifies an orientation by:

- Rotating about the initial *z*-axis
- Rotating about the intermediate *y*-axis
- Rotating about the second intermediate *x*-axis

Dependencies

You must select Euler Angles for the Representation input or output parameter. The axis rotation sequence only applies to Euler angle rotations.

Show TrVec input/output port — Toggle TrVec port
off (default) | on

Toggle the TrVec input or output port when you want to specify or receive a separate translation vector for position information along with an orientation representation.

Dependencies

You must select Homogeneous Transformation (TForm) for the opposite transformation port to get the option to show the additional TrVec port.

Simulate using — Type of simulation to run
Interpreted execution (default) | Code generation

- 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.
- 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.

Tunable: No

Version History

Introduced in R2017b

Extended Capabilities

C/C++ Code Generation

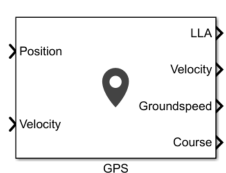
Generate C and C++ code using Simulink® Coder™.

See Also

`axang2quat` | `eul2tform` | `trvec2tform`

GPS

Simulate GPS sensor readings with noise



Libraries:
UAV Toolbox / UAV Scenario and Sensor Modeling
Navigation Toolbox / Multisensor Positioning / Sensor Models
Sensor Fusion and Tracking Toolbox / Multisensor Positioning / Sensor Models

Description

The block outputs noise-corrupted GPS measurements based on the input position and velocity in the local coordinate frame or geodetic frame. It uses the WGS84 earth model to convert local coordinates to latitude-longitude-altitude LLA coordinates.

Ports

Input

Position — Position of GPS receiver in navigation coordinate system
matrix

Specify the input position of the GPS receiver in the navigation coordinate system as a real, finite *N*-by-3 matrix. *N* is the number of samples in the current frame. The format of the matrix rows differs depending on the value of the **Position input format** parameter.

- If the value of the **Position input format** parameter is `Local`, specify each row of the **Position** as Cartesian coordinates in meters with respect to the local navigation reference frame, specified by the **Reference frame** parameter, with the origin specified by the **Reference location** parameter.
- If the value of the **Position input format** parameter is `Geodetic`, specify each row of the **Position** input as geodetic coordinates of the form `[latitude longitude altitude]`. The values of `latitude` and `longitude` are in degrees. `Altitude` is the height above the WGS84 ellipsoid model in meters.

Data Types: `single` | `double`

Velocity — Velocity in local navigation coordinate system (m/s)
matrix

Specify the input velocity of the GPS receiver in the navigation coordinate system in meters per second as a real, finite *N*-by-3 matrix. *N* is the number of samples in the current frame. The format of the matrix rows differs depending on the value of the **Position input format** parameter.

- If the value of the **Position input format** parameter is `Local`, specify each row of the **Velocity** with respect to the local navigation reference frame (NED or ENU), specified by the **Reference frame** parameter, with the origin specified by the **Reference location** parameter.

- If the value of the **Position input format** parameter is `Geodetic`, specify each row of the **Velocity** with respect to the navigation reference frame (NED or ENU), specified by the **Reference frame** parameter, with the origin specified by **Position**.

Data Types: `single` | `double`

Output

LLA — Position in LLA coordinate system
matrix

Position of the GPS receiver in the geodetic latitude, longitude, and altitude (LLA) coordinate system, returned as a real, finite *N*-by-3 array. Latitude and longitude are in degrees with North and East being positive. Altitude is in meters.

N is the number of samples in the current frame.

Data Types: `single` | `double`

Velocity — Velocity in local navigation coordinate system (m/s)
matrix

Velocity of the GPS receiver in the local navigation coordinate system in meters per second, returned as a real, finite *N*-by-3 matrix. *N* is the number of samples in the current frame. The format of the matrix rows differs depending on the value of the **Position input format** parameter.

- If the value of the **Position input format** parameter is `Local`, the **Velocity** output is with respect to the local navigation reference frame (NED or ENU), specified by the **Reference frame** parameter, with the origin specified by the **Reference location** parameter.
- If the value of the **Position input format** parameter is `Geodetic`, the **Velocity** output is with respect to the navigation reference frame (NED or ENU), specified by the **Reference frame** parameter, with the origin specified by **LLA**.

Data Types: `single` | `double`

Groundspeed — Magnitude of horizontal velocity in local navigation coordinate system (m/s)
vector

Magnitude of the horizontal velocity of the GPS receiver in the local navigation coordinate system in meters per second, returned as a real, finite *N*-element column vector.

N is the number of samples in the current frame.

Data Types: `single` | `double`

Course — Direction of horizontal velocity in local navigation coordinate system (°)
vector

Direction of the horizontal velocity of the GPS receiver in the local navigation coordinate system, in degrees, returned as a real, finite *N*-element column vector of values from 0 to 360. North corresponds to 0 degrees and East corresponds to 90 degrees.

N is the number of samples in the current frame.

Data Types: `single` | `double`

Parameters

Reference frame — Reference frame
NED (default) | ENU

Specify the reference frame as NED (North-East-Down) or ENU(East-North-Up).

Position input format — Position coordinate input format
Local (default) | Geodetic

Specify the position coordinate input format as Local or Geodetic.

- If you set this parameter to Local, then the input to the **Position** port must be in the form of Cartesian coordinates with respect to the local navigation frame, specified by the **Reference Frame** parameter, with the origin fixed and defined by the **Reference location** parameter. The input to the **Velocity** input port must also be with respect to this local navigation frame.
- If you set this parameter to Geodetic, then the input to the **Position** port must be geodetic coordinates in [latitude longitude altitude]. The input to the **Velocity** input port must also be with respect to the navigation frame specified by the **Reference frame** parameter, with the origin corresponding to the **Position** port.

Reference location — Origin of local navigation reference frame
[0,0,0] (default) | three-element vector

Specify the origin of the local reference frame as a three-element row vector in geodetic coordinates [latitude longitude altitude], where altitude is the height above the reference ellipsoid model WGS84. The reference location values are in degrees, degrees, and meters, respectively. The degree format is decimal degrees (DD).

Dependencies

To enable this parameter, set the **Position input format** parameter to Local.

Horizontal position accuracy — Horizontal position accuracy (m)
1.6 (default) | nonnegative real scaler

Specify horizontal position accuracy as a nonnegative real scalar in meters. The horizontal position accuracy specifies the standard deviation of the noise in the horizontal position measurement. Increasing this value adds noise to the measurement, decreasing its accuracy.

Tunable: Yes

Vertical position accuracy — Vertical position accuracy (m)
3 (default) | nonnegative real scaler

Specify vertical position accuracy as a nonnegative real scalar in meters. The vertical position accuracy specifies the standard deviation of the noise in the vertical position measurement. Increasing this value adds noise to the measurement, decreasing its accuracy.

Tunable: Yes

Velocity accuracy — Velocity accuracy (m/s)
0.1 (default) | nonnegative real scalar

Specify velocity accuracy per second as a nonnegative real scalar in meters. The velocity accuracy specifies the standard deviation of the noise in the velocity measurement. Increasing this value adds noise to the measurement, decreasing its accuracy.

Tunable: Yes

Decay factor — Global position noise decay factor
`0.999` (default) | scalar in range [0, 1]

Specify the global position noise decay factor as a numeric scalar in the range [0, 1]. A decay factor of `0` models the global position noise as a white noise process. A decay factor of 1 models the global position noise as a random walk process.

Tunable: Yes

Seed — Initial seed
`67` (default) | nonnegative integer

Specify the initial seed of an mt19937ar random number generator algorithm as a nonnegative integer.

Simulate using — Type of simulation to run
`Interpreted execution` (default) | `Code generation`

Select the type of simulation to run from these options:

- `Interpreted execution` — Simulate the model using the MATLAB interpreter. For more information, see "Simulation Modes" (Simulink).
- `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.

Version History

Introduced in R2021b

Extended Capabilities

C/C++ Code Generation

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

See Also

Objects

`gpsSensor`

Fixed-Wing UAV Point Mass

Integrate fourth- or sixth-order point mass equations of motion in coordinated flight

Libraries:
UAV Toolbox / Algorithms
Aerospace Blockset / Equations of Motion / Point Mass

Description

The Fixed-Wing Point Mass block integrates fourth- or sixth-order point mass equations of motion in coordinated flight.

Limitations

- The flat Earth reference frame is considered inertial, an approximation that allows the forces due to the Earth's motion relative to the "fixed stars" to be neglected.
- The block assumes that there is fully coordinated flight, that is, there is no side force (wind axes) and sideslip is always zero.

Ports

Input

Lift — Lift
scalar

Lift, specified as a scalar in units of force.

Data Types: `double`

Drag — Drag
scalar

Drag, specified as a scalar in units of force.

Data Types: `double`

Weight — Weight
scalar

Weight, specified as a scalar in units of force.

Data Types: `double`

Thrust — Thrust
scalar

Thrust, specified as a scalar in units of force.

Data Types: `double`

γ_a — Flight path angle relative to the air mass
scalar

Flight path angle relative to the air mass, specified as a scalar in radians.

Data Types: `double`

μ — Bank angle
scalar

Bank angle, specified as a scalar in radians.

Data Types: `double`

α — Angle of attack
scalar

Angle of attack, specified as a scalar in radians.

Data Types: `double`

V_{wind} — Wind vector
three-element vector

Wind vector in the direction in which the air mass is moving, specified as a three-element vector.

Data Types: `double`

Output

V — Airspeed
scalar

Airspeed, returned as a scalar.

Data Types: `double`

G — Ground speed projection
scalar

Ground speed over the Earth (speed of motion over the ground), returned as a scalar.

Data Types: `double`

V_a — Velocity vector relative to air mass
three-element vector

Velocity vector relative to the air mass, returned as a three-element vector.

Data Types: `double`

V_e — Velocity vector relative to Earth with [North East Down] orientation
three-element vector

Velocity vector relative to Earth with `[North East Down]` orientation, returned as a three-element vector.

Dependencies

To enable this port, set **Reference frame orientation** to [North East Down].

Data Types: double

V_{ENU} — Velocity vector relative to Earth
three-element vector

Velocity vector relative to Earth with [East North Up] orientation, returned as a three-element vector.

Dependencies

To enable this port, set **Reference frame orientation** to [East North Up].

Data Types: double

X_e — Position vector relative to Earth
three-element vector

Position vector relative to Earth with [North East Down] orientation, returned as a three-element vector.

Dependencies

To enable this port, set **Reference frame orientation** to [North East Down].

Data Types: double

X_{ENU} — Position vector relative to Earth
three-element vector

Position vector relative to Earth with [East North Up] orientation, returned as a three-element vector.

Dependencies

To enable this port, set **Reference frame orientation** to [East North Up].

Data Types: double

γ_a — Flight path angle relative to air mass
scalar

Flight path angle relative to the air mass, returned as a scalar.

Data Types: double

γ — Flight path angle relative to Earth
scalar

Flight path angle relative to Earth, returned as a scalar.

Data Types: double

χ_a — Heading angle relative to air mass
scalar

Heading angle relative to air mass, returned as a scalar.

Dependencies

To enable this port, set **Degrees of Freedom** to `6th Order (Coordinated Flight)`.

Data Types: `double`

χ — Heading angle relative to Earth
scalar

Heading angle relative to Earth, returned as a scalar.

Dependencies

To enable this port, set **Degrees of Freedom** to `6th Order (Coordinated Flight)`.

Data Types: `double`

Parameters

Units — Units

`Metric (MKS)` (default) | `English (velocity in ft/s)` | `English (velocity in kts)`

Input and output units, specified as follows:

Units	Forces	Velocity	Position	Mass
`Metric (MKS)`	newtons	meters per second	meters	kilograms
`English (velocity in ft/s)`	pounds	feet per second	feet	slugs
`English (velocity in kts)`	pounds	knots	feet	slugs

Programmatic Use
Block Parameter: `units`
Type: character vector
Values: `'Metric (MKS)'` | `'English (velocity in ft/s)'` | `'English (velocity in kts)'`
Default: `'Metric (MKS)'`

Reference frame orientation — Reference frames

`[North East Down]` (default) | `[East North Up]`

Reference frames used for input ports and output ports, specified as `[East North Up]` or `[North East Down]`.

Programmatic Use
Block Parameter: `frame`
Type: character vector
Values: `'[East North Up]'` | `'[North East Down]'`
Default: `'[North East Down]'`

Degrees of freedom — Degrees of freedom

`6th Order (Coordinated Flight)` (default) | `4th Order (Longitudinal)`

Degrees of freedom, specified as `4th Order (Longitudinal)` or `6th Order (Coordinated Flight)`.

Programmatic Use
Block Parameter: `order`
Type: character vector
Values: `'4th Order (Longitudinal)'` | `'6th Order (Coordinated Flight)'`
Default: `'6th Order (Coordinated Flight)'`

Initial crossrange — Initial East (Earth) crossrange location

`0` (default) | scalar

Initial East (Earth) location in the `[North East Down]` orientation, specified as a scalar.

Dependencies

The direction specification of this parameter depends on the **Reference frame orientation** and **Degrees of Freedom** setting:

Initial crossrange	Reference frame orientation	Degrees of freedom
East	[North East Down]	6th Order (Coordinated Flight)
North	[East North Up]	6th Order (Coordinated Flight)

Programmatic Use
Block Parameter: `east`
Type: character vector
Values: scalar
Default: `'0'`

Initial downrange — Initial North (Earth) downrange

`0` (default) | scalar

Initial North (Earth) downrange of the point mass, specified as a scalar.

Dependencies

The direction specification of this parameter depends on the **Reference frame orientation** and **Degrees of Freedom** setting:

Initial downrange	Reference frame orientation	Degrees of freedom
North	[North East Down]	6th Order (Coordinated Flight)
North	[North East Down]	4th Order (Longitudinal)
East	[East North Up]	6th Order (Coordinated Flight)
East	[East North Up]	4th Order (Longitudinal)

Programmatic Use
Block Parameter: `north`
Type: character vector
Values: scalar

Default: `'0'`

Initial altitude — Initial altitude

`0` (default) | scalar

Initial altitude of the point mass, specified as a scalar.

Programmatic Use
Block Parameter: `altitude`
Type: character vector
Values: scalar
Default: `'0'`

Initial airspeed — Initial airspeed

`50` (default) | scalar

Initial airspeed of the point mass, specified as a scalar.

Programmatic Use
Block Parameter: `'airspeed'`
Type: character vector
Values: scalar
Default: `'50'`

Initial flight path angle — Initial flight path angle

`0` (default) | scalar

Initial flight path angle of the point mass, specified as a scalar.

Programmatic Use
Block Parameter: `gamma`
Type: character vector
Values: scalar
Default: `'0'`

Initial heading angle — Initial heading angle

`0` (default) | scalar

Initial heading angle of the point mass, specified as a scalar.

Dependencies

To enable this parameter, set **Degrees of Freedom** to `6th Order (Coordinated Flight)`.

Programmatic Use
Block Parameter: `chi`
Type: character vector
Values: scalar
Default: `'0'`

Mass — Point mass

`10` (default) | scalar

Mass of the point mass, specified as a scalar.

Programmatic Use
Block Parameter: `mass`
Type: character vector
Values: scalar
Default: `'10'`

Algorithms

The integrated equations of motion for the point mass are:

$$\dot{V} = (T\cos\alpha - D - W\sin\gamma_{ai})/m$$
$$\dot{\gamma}_a = ((L + T\sin\alpha)\cos\mu - W\cos\gamma_{ai})/(mV)$$
$$\dot{X}_e = V_a + V_w$$

6th order equations:

$$\dot{X}_a = ((L + T\sin\alpha)\sin\mu)/(mV\cos\gamma_a)$$
$$\dot{X}_a\Big|_{East} = V\cos\chi_a\cos\gamma_a$$
$$\dot{X}_a\Big|_{North} = V\sin\chi_a\cos\gamma_a$$
$$\dot{X}_a\Big|_{Up} = V\sin\gamma_a$$

4th order equations:

$$\dot{\chi}_a = 0$$
$$\dot{X}_a\Big|_{East} = V\cos\gamma_a$$
$$\dot{X}_a\Big|_{North} = 0$$
$$\dot{X}_a\Big|_{Up} = V\sin\gamma_a$$

where:

- m — Mass.
- g — Gravitational acceleration.
- W — Weight ($m*g$).
- L — Lift force.
- D — Drag force.
- T — Thrust force.
- α — Angle of attack.
- µ — Angle of bank.
- γ_{ai} — Input port value for the flight path angle.
- V — Airspeed, as measured on the aircraft, with respect to the air mass. It is also the magnitude of vector V_a.

- V_w — Steady wind vector.
- Subscript a — For the variables, denotes that they are with respect to the steadily moving air mass:
  - γ_a — Flight path angle.
  - χ_a — Heading angle.
  - X_a — Position [East, North, Up].
- Subscript e — Flat Earth inertial frame such that so X_e is the position on the Earth after correcting X_a for the air mass movement.

Additional outputs are:

$$G = \sqrt{(V_e|_{East}{}^2 + V_e|_{North}{}^2)}$$

$$\gamma = \sin^{-1}\left(\frac{V_e|_{Up}}{\|\underline{V_e}\|}\right)$$

$$\chi = \tan^{-1}\left(\frac{V_e|_{North}}{V_e|_{East}}\right)$$

where:

- The four-quadrant inverse tangent (`atan2`) calculates the heading angle.
- The groundspeed, G, is the speed over the flat Earth (a 2-D projection).

Version History

Introduced in R2021a

Extended Capabilities

C/C++ Code Generation

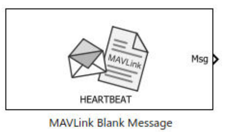
Generate C and C++ code using Simulink® Coder™.

See Also

Guidance Model | `fixedwing`

MAVLink Blank Message

Create blank MAVLink message bus by specifying payload information and MAVLink message type



Libraries:
UAV Toolbox / MAVLink

Description

The MAVLink Blank Message block creates a Simulink nonvirtual bus representing a MAVLink packet based on the specified Message ID, System ID, Component ID, Sequence, Payload information, and MAVLink message type.

Payload information is another nonvirtual bus within the MAVLink packet bus. The block creates Simulink buses for the MAVLink packet and the corresponding message that work with MAVLink Serializer and MAVLink Deserializer blocks. On each sample hit, the block outputs a blank or zero signal for the payload for the designated message type.

All elements of the bus other than the Message ID, System ID, and Component ID are initialized to `0`. The only exception is the `mavlink_version` field in the HEARTBEAT message of the `common.xml` dialect which is initialized to `3`.

Ports

Output

Msg — MAVLink packet
nonvirtual bus

MAVLink packet, returned as a Simulink nonvirtual bus. The bus contains the fields Message ID, System ID, Component ID, Sequence, and Payload. The Payload is another nonvirtual bus corresponding to the MAVLink message type that you selected in the **MAVLink message type** parameter. The Message ID is initialized to the numeric value of the selected MAVLink message ID. The System ID and Component ID are initialized to the corresponding **System ID** and **Component ID** parameters.

Data Types: `bus`

Parameters

MAVLink dialect source — Source for specifying the MAVLink message definition
`Select from standard MAVLink dialects` (default) | `Specify your own`

Source for specifying the MAVLink message definition XML name, specified as one of the following:

- `Select from standard MAVLink dialects` - Use this option to select a definition XML among the 12 commonly used message definition XML names listed in the **MAVLink dialect** parameter.

- `Specify your own` - Enter an XML name in the text box that appears for the **MAVLink dialect** parameter.

MAVLink dialect — Message definition to parse for MAVLink messages
`common.xml` (default) | string

MAVLink message definition file (`.xml`) to parse for MAVLink messages, specified as a string.

If the **MAVLink dialect source** parameter is set to `Select from standard MAVLink dialects`, you need to select a message definition among the available message definition names from the drop down list.

If the **MAVLink dialect source** parameter is set to `Specify your own`, you need to specify the message definition file (`.xml`) that is on current MATLAB path or you can provide the full path of the XML file.

MAVLink version — MAVLink protocol version
`2` (default) | `1`

MAVLink protocol version that is used to serialize and deserialize the MAVLink messages.

MAVLink Message type — MAVLink message
`HEARTBEAT` (default)

MAVLink message, specified as a string. Click **Select** to select from a full list of available MAVLink messages that depends on the values that you selected for **MAVLink dialect** and **MAVLink version** parameters.

Data Types: `string`

System ID — System ID of the sender
`1` (default)

MAVLink system ID, specified as a positive integer between 1 and 255. MAVLink protocol only supports up to 255 systems. Each UAV has its own system ID, but multiple UAVs can be considered as one system.

Data Types: `uint8`

Component ID — Component ID of the sender
`1` (default)

MAVLink component ID, specified as a positive integer between 1 and 255.

Data Types: `uint8`

Sample time — Interval between outputs
`inf` (default) | scalar

The default value (`inf`) indicates that the block output never changes. If you use this value, the simulation and code generation are faster by eliminating the need to recompute the block output. For other values, the block outputs a new blank message at each interval of Sample time.

For more information, see “Specify Sample Time” (Simulink) (Simulink).

Data Types: `uint8`

Tip

You can change the values for the desired fields in Payload bus by using a Bus Assignment block and then pass the MAVLink packet bus to the MAVLink Serializer block as an input.

Version History

Introduced in R2020b

Extended Capabilities

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

Usage and Limitations:

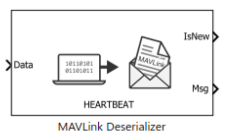
- The C/C++ code generated for the block can be deployed only on a Linux target.

See Also

MAVLink Serializer | MAVLink Deserializer

MAVLink Deserializer

Convert serialized `uint8` MAVLink data stream to Simulink nonvirtual bus



MAVLink Deserializer

Libraries:
UAV Toolbox / MAVLink

Description

The MAVLink Deserializer block receives a `uint8` buffer and decodes the buffer for MAVLink messages. Once the block receives the MAVLink message for the selected MAVLink message type, the block outputs a Simulink nonvirtual bus representing a MAVLink packet containing the Message ID, System ID, Component ID, Sequence, and Payload information corresponding to the selected MAVLink message type.

At each simulation step, the block decodes the input `uint8` buffer and retrieves the MAVLink messages that are received after decoding. If a new message for the selected MAVLink message type has been received, the block retrieves that message from the list of received messages and converts it to a Simulink nonvirtual bus signal.

The MAVLink decoding logic in the block takes care of scenarios where a MAVLink packet has been received partially from a communication channel. The MAVLink Deserializer block internally stores the current state of parsing and resumes decoding from the previous step when the new buffer has been received over the communication channel. If the complete MAVLink packet has been received and the received checksum matches the computed checksum for the received bytes, then this indicates that a MAVLink message has been received. Storing the state of parsing ensures that the block can decode the MAVLink packets received in multiple parts.

By default, the block outputs the latest received MAVLink message for the selected MAVLink message type (if received). This behavior can be changed by selecting **Queue Messages in output** parameter. In this case, all the received MAVLink messages for the desired type are queued and at each Simulation step, the block outputs the oldest message.

Ports

Input

Data — MAVLink data stream
vector

The `uint8` byte stream that contains serialized MAVLink packets. The byte stream is usually received over a communication channel such as UDP, TCP, or Serial. At each sample time, the communication channel receives data and returns a byte stream that contains one or more MAVLink packets. The byte stream can also return a MAVLink packet partially in over multiple sample times. This input port accepts variable-length signals.

Data Types: `uint8`

Length — Length of valid MAVLink data at Data input port
scalar

Optional input port to include the length of valid MAVLink data. To enable this port, select the **Input data stream length is available** parameter. Use this option when you know the exact length of the valid MAVLink data in the data stream.

This option is useful when you have a communication channel receive peripheral that outputs partially received data that contains trailing zeros. Such peripherals also output the length of the actual number of valid data bytes received. You can connect the length output of the peripheral directly with the **Length** input port of MAVLink Deserializer block, so that trailing zeros in the input byte stream do not affect the decoding logic.

Data Types: `uint16`

Output

IsNew — New message indicator
`0` | `1`

New MAVLink message indicator returned as a logical. A value of `1` indicates that a new message is available since the last sample was received by the block. This output can be used to trigger subsystems to process new messages received from the MAVLink Deserializer block.

Data Types: `Boolean`

Msg — MAVLink packet
nonvirtual bus

MAVLink packet, returned as a nonvirtual bus. The type of Payload in the MAVLink packet is a Simulink bus corresponding to the MAVLink message specified in the MAVLink message type parameter. The block outputs blank messages until it receives a message on the message name that you specify. The **Msg** port outputs this new message. If a new message is not available, it outputs the last received MAVLink message. If a message has not been received since the start of the simulation, **Msg** port outputs a blank MAVLink message.

Data Types: `bus`

Parameters

Main

MAVLink dialect source — Source for specifying the MAVLink message definition
`Select from standard MAVLink dialects` (default) | `Specify your own`

Source for specifying the MAVLink message definition XML name, specified as one of the following:

- `Select from standard MAVLink dialects` - Use this option to select a definition XML among the 12 commonly used message definition XML names listed in the **MAVLink dialect** parameter.
- `Specify your own` - Enter an XML name in the text box that appears for the **MAVLink dialect** parameter.

MAVLink dialect — Message definition to parse for MAVLink messages
`common.xml` (default) | string

MAVLink message definition file (`.xml`) to parse for MAVLink messages, specified as a string.

If the **MAVLink dialect source** parameter is set to `Select from standard MAVLink dialects`, you need to select a message definition among the available message definition names from the dropdown list.

If the parameter **MAVLink dialect source** parameter is set to `Specify your own`, you need to specify the message definition file (`.xml`) that is on the current MATLAB path, or you can provide the full path of the XML file.

MAVLink version — MAVLink protocol version
`2` (default) | `1`

MAVLink protocol version that the block uses to serialize and deserialize the MAVLink messages.

MAVLink Message type — MAVLink message
`HEARTBEAT` (default)

MAVLink message, specified as a string. Click **Select** to select from a full list of available MAVLink messages. The list varies based on the values that you selected for **MAVLink dialect** and **MAVLink version** parameters.

Data Types: `string`

Advanced

Input data stream length is available — Length of valid MAVLink data in the input byte stream is known
`off` (default) | `on`

When you select this option, the MAVLink Deserializer block provides an additional input port called **Length**. This input port can be used to pass the actual length of MAVLink data (if known) in the input byte stream. The input byte stream is cropped for this length.

This option is useful when you have a communication channel receive peripheral that outputs partially received data that contains trailing zeros. Such peripherals also output length of the actual number of valid data bytes received. You can connect the length output of the peripheral directly to the **Length** input port of MAVLink Deserializer block so that trailing zeros in the input byte stream do not affect the decoding logic.

Filter output MAVLink messages by System ID — Filter received messages by System ID
`off` (default) | `on`

Select this option to filter the received MAVLink messages for the System ID value mentioned in the **System ID** parameter. This option helps you to filter the received messages by both System ID and Component ID.

System ID — System ID value
`1` (default) | scalar in the range `[0,255]`

Specify the System ID value to use while filtering the decoded MAVLink messages. The block outputs the received MAVLink messages whose System ID matches the specified value and whose Message ID matches the MAVLink message (selected in the **MAVLink Message type** parameter).

Dependencies

To enable this parameter, select **Filter Output MAVLink messages by System ID**.

Filter output MAVLink messages by Component ID — Filter received messages by System ID and Component ID
`off` (default) | `on`

Select this option to filter the received MAVLink messages for the both the System ID and the Component ID mentioned in the **System ID** and **Component ID** parameters, respectively.

Dependencies

This parameter appears only if you select the **Filter output MAVLink messages by System ID** parameter.

Component ID — Component ID value
`1` (default) | `[0,255]`

Specify the Component ID value to use while filtering the decoded MAVLink messages. The block outputs those received MAVLink messages whose System ID and Component ID values match the specified values in the **System ID** and **Component ID** parameters, respectively, and whose Message ID matches the MAVLink message (selected in the **MAVLink Message type** parameter).

Dependencies

To enable this parameter, select **Filter Output MAVLink messages by Component ID**.

Queue MAVLink messages in output — Enable queuing of the received MAVLink messages
`off` (default) | `on`

Select this option to output messages using the first-in-first-out pattern. If you do not select this option, the MAVLink Deserializer block outputs the latest received MAVLink message for the selected **MAVLink message type** (and with matching System ID and Component ID if those parameters are selected) at each simulation step. If more than one message matches the given parameters that are received in a simulation step, the latest message is passed as output, and the rest are discarded. You can reverse this behavior by selecting this option.

When you select this parameter, the behavior of the MAVLink Deserializer block at each simulation step is:

- The block stores the decoded MAVLink messages matching the selected MAVLink message type (and matching System ID and Component ID if the those parameters are selected) in a queue. If there are no messages among the received messages that match the required parameters, no messages are queued.
- If the queue is not empty, the first message in the queue is sent as an output first, and the signal at **IsNew** port is set to `1`.

Selecting the **Queue MAVLink messages in output** parameter makes the **Number of messages to be queued** parameter visible. You can fix the size of the queue by setting the value of this parameter.

Number of messages to be queued — Size of MAVLink message queue
`50` (default) | scalar in the range (`0,65535`]

Specify the size of the queue to be used to store the received MAVLink messages matching the desired parameters.

Dependencies

To enable this parameter, select **Queue MAVLink messages in output**.

Tips

To speed up the conversion of the received serialized data, it is recommended that you apply the following settings in the communication channel receive block:

- Read the data at the highest rate possible to ensure that no packets are dropped. Use the **IsNew** output of MAVLink Deserializer along with the logic to use MAVLink messages to know if the output of the block is a new message or not.
- If the receive block outputs any number of bytes that are received irrespective of the data size requested (partial receive), mention the data read size as a large number and use the length of actual number of bytes received as an input to MAVLink Deserializer block (use the **Length** input port).

Version History

Introduced in R2020b

Extended Capabilities

C/C++ Code Generation

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

Usage and Limitations:

- The C/C++ code generated for the block can be deployed only on a Linux target.

See Also

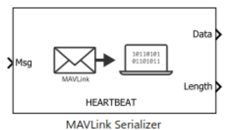
MAVLink Blank Message | MAVLink Serializer

Topics

"Exchange Data for MAVLink Microservices like Mission Protocol and Parameter Protocol Using Simulink"

MAVLink Serializer

Serialize messages of MAVLink packet by converting Simulink nonvirtual bus to `uint8` data stream



Libraries:
UAV Toolbox / MAVLink

Description

The MAVLink Serializer block accepts a Simulink nonvirtual bus and converts it into a `uint8` MAVLink data stream. The nonvirtual bus represents a MAVLink packet containing the Message ID, System ID, Component ID, Sequence, and Payload information corresponding to the selected MAVLink message. Payload information is another nonvirtual bus within the MAVLink packet bus.

MAVLink v2 removes trailing zeros in the payload. Therefore, the length of the payload in the serialized MAVLink data can be less than the maximum payload length of a selected MAVLink message type.

The **Data** port outputs the MAVLink data stream, and the length of the output data is the maximum possible length for the selected MAVLink message. If the length of the serialized data is less than the maximum possible length, trailing zeros are added to the data stream. The **Length** port outputs the true length of the serialized MAVLink data.

Ports

Input

Msg — MAVLink packet
nonvirtual bus

MAVLink packet as a nonvirtual bus. This is the output of the MAVLink Blank Message block in which the values for Message ID, System ID, and Component ID are already initialized. The fields in the Payload bus can be modified using a Bus Assignment block before passing it as an input to MAVLink Serializer block.

Data Types: `bus`

Output

Data — MAVLink data stream
vector

The serialized MAVLink data for the input MAVLink message bus. MAVLink protocol version 2 removes trailing zeros in the payload. Therefore, the length of the payload in the serialized data can be less than the maximum payload length of the MAVLink message in the dialect. In this case, the block outputs the serialized data stream with the trailing zeros included.

Data Types: `uint8`

Length — Length of the serialized data
scalar

The true length of the serialized data including headers and payload. This might be less than the maximum possible length for a MAVLink message depending on how many trailing zeros are removed in the MAVLink payload during serialization.

Data Types: `uint16`

Parameters

MAVLink dialect source — Source for specifying the MAVLink message definition
`Select from standard MAVLink dialects` (default) | `Specify your own`

Source for specifying the MAVLink message definition XML name, specified as one of the following:

- `Select from standard MAVLink dialects` - Use this option to select a definition XML among the 12 commonly used message definition XML names listed in the **MAVLink dialect** parameter.
- `Specify your own` - Enter an XML name in the text box that appears for the **MAVLink dialect** parameter.

MAVLink dialect — Message definition to parse for MAVLink messages
`common.xml` (default) | string

MAVLink message definition file (`.xml`) to parse for MAVLink messages, specified as a string.

If the **MAVLink dialect source** parameter is set to `Select from standard MAVLink dialects`, you need to select a message definition among the available message definition names from the dropdown list.

If the **MAVLink dialect source** parameter is set to `Specify your own`, you need to specify the message definition file (`.xml`) that is on current MATLAB path or you can provide the full path of the XML file.

MAVLink version — MAVLink protocol version
2 (default) | 1

MAVLink protocol version that is used to serialize and deserialize the MAVLink messages.

MAVLink Message type — MAVLink message
HEARTBEAT (default)

MAVLink message, specified as a string. Click **Select** to select from a full list of available MAVLink messages that are specific to the values that you selected for **MAVLink dialect** and **MAVLink version** parameters.

Data Types: `string`

Tip

You can change the values for the desired fields in the Payload in the output of the MAVLink Blank message by using a Bus Assignment block and then pass the MAVLink packet bus to the MAVLink Serializer block as an input.

Version History

Introduced in R2020b

Extended Capabilities

C/C++ Code Generation

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

Usage and Limitations:

- The C/C++ code generated for the block can be deployed only on a Linux target.

See Also

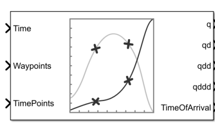
MAVLink Blank Message | MAVLink Deserializer

Topics

"Exchange Data for MAVLink Microservices like Mission Protocol and Parameter Protocol Using Simulink"

Minimum Jerk Polynomial Trajectory

Generate minimum jerk polynomial trajectories through multiple waypoints



Libraries:
UAV Toolbox / Algorithms
Robotics System Toolbox / Utilities

Description

The Minimum Jerk Polynomial Trajectory block generates minimum jerk polynomial trajectories that pass through the waypoints at the times specified in time points. The block outputs positions, velocities, accelerations, jerks, and time of arrival for achieving this trajectory based on the `Time` input.

The block also accepts boundary conditions for waypoints. The block also outputs the coefficients for the polynomials and status of the trajectory generation.

The initial and final values of positions, velocities, accelerations, and jerks of the trajectory are held constant outside the time period defined in `TimePoints` input.

Ports

Input

Time — Time point along trajectory
scalar | vector

Time point along the trajectory, specified as a scalar or vector.

- When the time is specified as a scalar, this value is synced with simulation time and is used to specify the time point for sampling the trajectory. The block outputs a vector of the trajectory variables at that instance in time.
- If the time is specified as a vector, the block outputs a matrix with each column corresponding to each element of the vector.

Data Types: `single` | `double`

Waypoints — Waypoints positions along trajectory
n-by-*p* matrix

Positions of waypoints of the trajectory at given time points, specified as an *n*-by-*p* matrix. *n* is the dimension of the trajectory and *p* is the number of waypoints.

Data Types: `single` | `double`

TimePoints — Time points for waypoints of trajectory
p-element row vector

Time points for the waypoints of the trajectory, specified as a *p*-element row vector. *p* is the number of waypoints.

Data Types: `single` | `double`

VelBC — Velocity boundary conditions for waypoints
n-by-*p* matrix

Velocity boundary conditions for waypoints, specified as an *n*-by-*p* matrix. Each row sets the velocity boundary for the corresponding dimension of the trajectory *n* at each of *p* waypoints.

By default, the block uses a value of `0` at the boundary waypoints and `NaN` at the intermediate waypoints.

Dependencies

To enable this input port, select `Show boundary conditions input ports`.

Data Types: `single` | `double`

AccelBC — Acceleration boundary conditions for waypoints
n-by-*p* matrix

Acceleration boundary conditions for waypoints, specified as an *n*-by-*p* matrix. Each row sets the acceleration boundary for the corresponding dimension of the trajectory *n* at each of *p* waypoints.

By default, the block uses a value of `0` at the boundary waypoints and `NaN` at the intermediate waypoints.

Dependencies

To enable this input port, select `Show boundary conditions input ports`.

Data Types: `single` | `double`

JerkBC — Jerk boundary conditions for waypoints
n-by-*p* matrix

Jerk boundary conditions for waypoints, specified as an *n*-by-*p* matrix. Each row sets the jerk boundary for the corresponding dimension of the trajectory *n* at each of *p* waypoints.

By default, the block uses a value of `0` at the boundary waypoints and `NaN` at the intermediate waypoints.

Dependencies

To enable this input port, select `Show boundary conditions input ports`.

Data Types: `single` | `double`

Output

q — Positions of trajectory
n-element vector | *n*-by-*m* matrix

Positions of the trajectory, returned as an *n*-element vector or *n*-by-*m* matrix.

- If you specify a scalar for the `Time` input with an *n*-dimensional trajectory, the output is a vector with *n*-elements.
- If you specify a vector of *m*-elements for the `Time` input, the output is an *n*-by-*m* matrix.

Data Types: `single` | `double`

qd — Velocities of trajectory
n-element vector | *n*-by-*m* matrix

Velocities of the trajectory, returned as an *n*-element vector or *n*-by-*m* matrix.

- If you specify a scalar for the `Time` input with an *n*-dimensional trajectory, the output is a vector with *n*-elements.
- If you specify a vector of *m*-elements for the `Time` input, the output is an *n*-by-*m* matrix.

Data Types: `single` | `double`

qdd — Accelerations of trajectory
n-element vector | *n*-by-*m* matrix

Accelerations of the trajectory, returned as an *n*-element vector or *n*-by-*m* matrix.

- If you specify a scalar for the `Time` input with an *n*-dimensional trajectory, the output is a vector with *n*-elements.
- If you specify a vector of *m*-elements for the `Time` input, the output is an *n*-by-*m* matrix.

Data Types: `single` | `double`

qddd — Jerks of trajectory
n-element vector | *n*-by-*m* matrix

Jerks of the trajectory, returned as an *n*-element vector or *n*-by-*m* matrix.

- If you specify a scalar for the `Time` input with an *n*-dimensional trajectory, the output is a vector with *n*-elements.
- If you specify a vector of *m*-elements for the `Time` input, the output is an *n*-by-*m* matrix.

Data Types: `single` | `double`

TimeOfArrival — Time of arrival at each waypoint
p-element vector

Time of arrival at each waypoint, returned as a *p*-element vector. *p* is the number of waypoints.

Data Types: `single` | `double`

PolynomialCoefs — Polynomial coefficients
$n(p-1)$-by-8 matrix

Polynomial coefficients, returned as an $n(p-1)$-by-8 matrix. *n* is the dimension of the trajectory and *p* is the number of waypoints. Each set of *n* rows defines the coefficients for the polynomial that described each variable trajectory.

Dependencies

To enable this output port, select `Show polynomial coefficients output port`.

Data Types: `single` | `double`

Status — Status of trajectory generation
three-element vector of the form [*SingularityStatus MaxIterStatus MaxTimeStatus*]

Status of trajectory generation, returned as a three-element vector of the form [*SingularityStatus MaxIterStatus MaxTimeStatus*].

SingularityStatus returned as `0` or `1` indicates the occurrence of singularity. If singularity occurs reduce the `Maximum segment time` to `Minimum segment time` ratio.

MaxIterStatus returned as `0` or `1` indicates if the number of iterations for the solver has exceeded `Maximum iterations`.

MaxTimeStatus returned as `0` or `1` indicates if the time for the solver has exceeded `Maximum time`.

Dependencies

To enable this output port, select `Show status output port`.

Data Types: `uint8`

Parameters

Show boundary conditions input ports — Accept boundary condition inputs
`off` (default) | `on`

Select this parameter to input the velocity, acceleration, and jerk boundary conditions, at the `VelBC`, `AccelBC`, and `JerkBC` ports, respectively.

Tunable: No

Show polynomial coefficients output port — Output polynomial coefficients
`off` (default) | `on`

Select this parameter to output polynomial coefficients at the `PolynomialCoefs` port.

Tunable: No

Show status output port — Output status
`off` (default) | `on`

Select this parameter to output status at the `Status` port.

Tunable: No

Time allocation — Enable time allocation
`off` (default) | `on`

Enable to specify time allocation for the trajectory using the `Time weight`, `Minimum segment time`, `Maximum segment time`, `Maximum iterations`, and `Maximum time` parameters.

Tunable: No

Time weight — Weight for time allocation
`100` (default) | positive scalar

Weight for time allocation, specified as a positive scalar.

Tunable: No

Dependencies

To enable this parameter, select `Time allocation`.

Minimum segment time — Minimum time segment length
`0.1` (default) | positive scalar | (*p*–1)-element positive row vector

Minimum time segment length, specified as a positive scalar or (*p*–1)-element positive row vector. *p* is the number of waypoints.

Tunable: No

Dependencies

To enable this parameter, select `Time allocation`.

Maximum segment time — Maximum time segment length
`1` (default) | positive scalar | (*p*–1)-element positive row vector

Maximum time segment length, specified as a positive scalar or (*p*–1)-element positive row vector. *p* is the number of waypoints.

Tunable: No

Dependencies

To enable this parameter, select `Time allocation`.

Maximum iterations — Maximum iterations for solver
`1500` (default) | positive integer scalar

Maximum iterations for solver, specified as a positive integer scalar.

Tunable: No

Dependencies

To enable this parameter, select `Time allocation`.

Maximum time — Maximum time for solver
`10` (default) | positive scalar

Maximum time for solver, specified as a positive scalar.

Tunable: No

Dependencies

To enable this parameter, select `Time allocation`.

Simulate using — Type of simulation to run
`Interpreted execution` (default) | `Code generation`

Select the type of simulation to run from these options:

- `Interpreted execution` — Simulate 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.
- `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`.

Tunable: No

Tips

For better performance, consider these options:

- Minimize the number of waypoint or parameter changes.
- Set the `Simulate using` parameter to `Code generation`. For more information, see "Simulation Modes" (Simulink).

Version History

Introduced in R2022a

References

[1] Bry, Adam, Charles Richter, Abraham Bachrach, and Nicholas Roy. "Aggressive Flight of Fixed-Wing and Quadrotor Aircraft in Dense Indoor Environments." *The International Journal of Robotics Research*, 34, no. 7 (June 2015): 969–1002.

[2] Richter, Charles, Adam Bry, and Nicholas Roy. "Polynomial Trajectory Planning for Aggressive Quadrotor Flight in Dense Indoor Environments." *Paper presented at the International Symposium of Robotics Research (ISRR 2013)*, 2013.

Extended Capabilities

C/C++ Code Generation

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

See Also

Functions

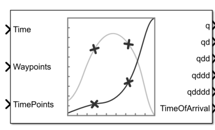
`minjerkpolytraj` | `minsnappolytraj`

Blocks

Minimum Snap Polynomial Trajectory

Minimum Snap Polynomial Trajectory

Generate minimum snap polynomial trajectories through multiple waypoints



Libraries:
UAV Toolbox / Algorithms
Robotics System Toolbox / Utilities

Description

The Minimum Snap Polynomial Trajectory block generates minimum snap polynomial trajectories that pass through the waypoints at the times specified in time points. The block outputs positions, velocities, accelerations, jerks, snap, and time of arrival for achieving this trajectory based on the `Time` input.

The block also accepts boundary conditions for waypoints. The block also outputs the coefficients for the polynomials and status of the trajectory generation.

The initial and final values of positions, velocities, accelerations, jerks, and snap of the trajectory are held constant outside the time period defined in `TimePoints` input.

Ports

Input

Time — Time point along trajectory
scalar | vector

Time point along the trajectory, specified as a scalar or vector.

- When the time is specified as a scalar, this value is synced with simulation time and is used to specify the time point for sampling the trajectory. The block outputs a vector of the trajectory variables at that instance in time.
- If the time is specified as a vector, the block outputs a matrix with each column corresponding to each element of the vector.

Data Types: `single` | `double`

Waypoints — Waypoints positions along trajectory
n-by-*p* matrix

Positions of waypoints of the trajectory at given time points, specified as an *n*-by-*p* matrix. *n* is the dimension of the trajectory and *p* is the number of waypoints.

Data Types: `single` | `double`

TimePoints — Time points for waypoints of trajectory
p-element row vector

Time points for the waypoints of the trajectory, specified as a *p*-element row vector. *p* is the number of waypoints.

Data Types: `single` | `double`

VelBC — Velocity boundary conditions for waypoints
n-by-*p* matrix

Velocity boundary conditions for waypoints, specified as an *n*-by-*p* matrix. Each row sets the velocity boundary for the corresponding dimension of the trajectory *n* at each of *p* waypoints.

By default, the block uses a value of `0` at the boundary waypoints and `NaN` at the intermediate waypoints.

Dependencies

To enable this input port, select `Show boundary conditions input ports`.

Data Types: `single` | `double`

AccelBC — Acceleration boundary conditions for waypoints
n-by-*p* matrix

Acceleration boundary conditions for waypoints, specified as an *n*-by-*p* matrix. Each row sets the acceleration boundary for the corresponding dimension of the trajectory *n* at each of *p* waypoints.

By default, the block uses a value of `0` at the boundary waypoints and `NaN` at the intermediate waypoints.

Dependencies

To enable this input port, select `Show boundary conditions input ports`.

Data Types: `single` | `double`

JerkBC — Jerk boundary conditions for waypoints
n-by-*p* matrix

Jerk boundary conditions for waypoints, specified as an *n*-by-*p* matrix. Each row sets the jerk boundary for the corresponding dimension of the trajectory *n* at each of *p* waypoints.

By default, the block uses a value of `0` at the boundary waypoints and `NaN` at the intermediate waypoints.

Dependencies

To enable this input port, select `Show boundary conditions input ports`.

Data Types: `single` | `double`

SnapBC — Snap boundary conditions for waypoints
n-by-*p* matrix

Snap boundary conditions for waypoints, specified as an *n*-by-*p* matrix. Each row sets the snap boundary for the corresponding dimension of the trajectory *n* at each of *p* waypoints.

By default, the block uses a value of `0` at the boundary waypoints and `NaN` at the intermediate waypoints.

Dependencies

To enable this input port, select `Show boundary conditions input ports`.

Data Types: `single` | `double`

Output

q — Positions of trajectory
n-element vector | *n*-by-*m* matrix

Positions of the trajectory, returned as an *n*-element vector or *n*-by-*m* matrix.

- If you specify a scalar for the `Time` input with an *n*-dimensional trajectory, the output is a vector with *n*-elements.
- If you specify a vector of *m*-elements for the `Time` input, the output is an *n*-by-*m* matrix.

Data Types: `single` | `double`

qd — Velocities of trajectory
n-element vector | *n*-by-*m* matrix

Velocities of the trajectory, returned as an *n*-element vector or *n*-by-*m* matrix.

- If you specify a scalar for the `Time` input with an *n*-dimensional trajectory, the output is a vector with *n*-elements.
- If you specify a vector of *m*-elements for the `Time` input, the output is an *n*-by-*m* matrix.

Data Types: `single` | `double`

qdd — Accelerations of trajectory
n-element vector | *n*-by-*m* matrix

Accelerations of the trajectory, returned as an *n*-element vector or *n*-by-*m* matrix.

- If you specify a scalar for the `Time` input with an *n*-dimensional trajectory, the output is a vector with *n*-elements.
- If you specify a vector of *m*-elements for the `Time` input, the output is an *n*-by-*m* matrix.

Data Types: `single` | `double`

qddd — Jerks of trajectory
n-element vector | *n*-by-*m* matrix

Jerks of the trajectory, returned as an *n*-element vector or *n*-by-*m* matrix.

- If you specify a scalar for the `Time` input with an *n*-dimensional trajectory, the output is a vector with *n*-elements.
- If you specify a vector of *m*-elements for the `Time` input, the output is an *n*-by-*m* matrix.

Data Types: `single` | `double`

qdddd — Snaps of trajectory
n-element vector | *n*-by-*m* matrix

Snaps of the trajectory, returned as an *n*-element vector or *n*-by-*m* matrix.

- If you specify a scalar for the `Time` input with an *n*-dimensional trajectory, the output is a vector with *n*-elements.

- If you specify a vector of *m*-elements for the `Time` input, the output is an *n*-by-*m* matrix.

Data Types: `single` | `double`

TimeOfArrival — Time of arrival at each waypoint
p-element vector

Time of arrival at each waypoint, returned as a *p*-element vector. *p* is the number of waypoints.

Data Types: `single` | `double`

PolynomialCoefs — Polynomial coefficients
n(*p*–1)-by-10 matrix

Polynomial coefficients, returned as an *n*(*p*–1)-by-10 matrix. *n* is the dimension of the trajectory and *p* is the number of waypoints. Each set of *n* rows defines the coefficients for the polynomial that described each variable trajectory.

Dependencies

To enable this output port, select `Show polynomial coefficients output port`.

Data Types: `single` | `double`

Status — Status of trajectory generation
three-element vector of the form [*SingularityStatus MaxIterStatus MaxTimeStatus*]

Status of trajectory generation, returned as a three-element vector of the form [*SingularityStatus MaxIterStatus MaxTimeStatus*].

SingularityStatus returned as `0` or `1` indicates the occurrence of singularity. If singularity occurs reduce the `Maximum segment time` to `Minimum segment time` ratio.

MaxIterStatus returned as `0` or `1` indicates if the number of iterations for the solver has exceeded `Maximum iterations`.

MaxTimeStatus returned as `0` or `1` indicates if the time limit for the solver has exceeded `Maximum time`.

Dependencies

To enable this output port, select `Show status output port`.

Data Types: `uint8`

Parameters

Show boundary conditions input ports — Accept boundary condition inputs
`off` (default) | `on`

Select this parameter to input the velocity, acceleration, jerk, and snap boundary conditions, at the `VelBC`, `AccelBC`, `JerkBC`, and `SnapBC` ports, respectively.

Tunable: No

Show polynomial coefficients output port — Output polynomial coefficients
`off` (default) | `on`

Select this parameter to output polynomial coefficients at the `PolynomialCoefs` port.

Tunable: No

Show status output port — Output status
`off` (default) | `on`

Select this parameter to output status at the `Status` port.

Tunable: No

Time allocation — Enable time allocation
`off` (default) | `on`

Enable to specify time allocation for the trajectory using the `Time weight`, `Minimum segment time`, `Maximum segment time`, `Maximum iterations`, and `Maximum time` parameters.

Tunable: No

Time weight — Weight for time allocation
`100` (default) | positive scalar

Weight for time allocation, specified as a positive scalar.

Tunable: No

Dependencies

To enable this parameter, select `Time allocation`.

Minimum segment time — Minimum time segment length
`0.1` (default) | positive scalar | (p–1)-element positive row vector

Minimum time segment length, specified as a positive scalar or (p–1)-element positive row vector. p is the number of waypoints.

Tunable: No

Dependencies

To enable this parameter, select `Time allocation`.

Maximum segment time — Maximum time segment length
`1` (default) | positive scalar | (p–1)-element positive row vector

Maximum time segment length, specified as a positive scalar or (p–1)-element positive row vector. p is the number of waypoints.

Tunable: No

Dependencies

To enable this parameter, select `Time allocation`.

Maximum iterations — Maximum iterations for solver
`1500` (default) | positive integer scalar

Maximum iterations for solver, specified as a positive integer scalar.

Tunable: No

Dependencies

To enable this parameter, select `Time allocation`.

Maximum time — Maximum time for solver
`10` (default) | positive scalar

Maximum time for solver, specified as a positive scalar.

Tunable: No

Dependencies

To enable this parameter, select `Time allocation`.

Simulate using — Type of simulation to run
`Interpreted execution` (default) | `Code generation`

Select the type of simulation to run from these options:

- `Interpreted execution` — Simulate 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.
- `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`.

Tunable: No

Tips

For better performance, consider these options:

- Minimize the number of waypoint or parameter changes.
- Set the `Simulate using` parameter to `Code generation`. For more information, see “Simulation Modes” (Simulink).

Version History

Introduced in R2022a

References

[1] Bry, Adam, Charles Richter, Abraham Bachrach, and Nicholas Roy. “Aggressive Flight of Fixed-Wing and Quadrotor Aircraft in Dense Indoor Environments.” *The International Journal of Robotics Research*, 34, no. 7 (June 2015): 969–1002.

[2] Richter, Charles, Adam Bry, and Nicholas Roy. “Polynomial Trajectory Planning for Aggressive Quadrotor Flight in Dense Indoor Environments." *Paper presented at the International Symposium of Robotics Research (ISRR 2013)*, 2013.

Extended Capabilities

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

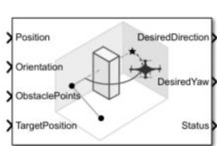
See Also

Functions
`minjerkpolytraj` | `minsnappolytraj`

Blocks
Minimum Jerk Polynomial Trajectory

Obstacle Avoidance

Compute obstacle-free direction using range sensor data and target position



Libraries:
UAV Toolbox / Algorithms

Description

The Obstacle Avoidance block computes an obstacle-free direction using range sensor data and target position.

Ports

Input

Position — Position of UAV
vector

Position of the UAV, specified as a vector of the form `[x; y; z]`, in meters.

Example: `[1; 1; 1]`

Data Types: `double`

Orientation — Orientation of UAV
vector

Orientation of the UAV, specified as a quaternion vector of the form `[w; x; y; z]`.

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

Data Types: `double`

ObstaclePoints — Positions of obstacles
matrix

Positions of the obstacles, specified as an *N*-by-3 matrix with rows of the form `[x y z]`, in meters. *N* is the number of obstacle points.

Example: `[1 1 1; 2 2 2]`

Data Types: `double`

TargetPostion — Position of target
vector

Position of the target, specified as a vector of the form `[x; y; z]`, in meters.

Example: `[2; 3; 4]`

Data Types: `double`

Output

DesiredDirection — Desired direction
vector

Desired direction, returned as a vector of the form [*x*; *y*; *z*], in meters.

Data Types: double

DesiredYaw — Desired yaw
scalar

Desired yaw, returned as numeric scalar in radians in the range of [-pi, pi].

Data Types: double

Status — Status of obstacle-free direction
0 | 1 | 2 | 3

Status of the obstacle-free direction, returned as 0, 1, 2, or 3.

- 0 — An obstacle-free direction is found.
- 1 — No obstacle-free direction is found.
- 2 — An obstacle-free direction is found but is close to the obstacle.
- 3 — No obstacle-free direction is found and is close to obstacle.

Data Types: uint8

Parameters

Main

Sensor range limits (m) — Limits of range sensor
[0.2 10] (default) | vector of form [*min max*]

Specify the minimum and maximum limits of the range sensor as a vector of the form [*min max*], with values in meters.

Data Types: double

Sensor horizontal field of view (deg) — Horizontal field of view limits of range sensor
[-60 60] (default) | vector of form [*min max*]

Specify the minimum and maximum horizontal field of view limits of the range sensor as a vector of the form [*min max*], with values in degrees.

Data Types: double

Sensor vertical field of view (deg) — Vertical field of view limits of range sensor
[-30 30] (default) | vector of form [*min max*]

Specify the minimum and maximum vertical field of view limits of the range sensor as a vector of the form [*min max*], with values in degrees.

Data Types: double

Sensor location [X, Y, Z] (m) — Sensor mounting location on UAV
`[0 0 0]` (default) | vector of form `[x y z]`

Specify the mounting location of the sensor on the UAV as a vector of the form `[x y z]`, with values in meters.

Data Types: `double`

Sensor orientation [Roll, Pitch, Yaw] (deg) — Orientation of sensor mounted on UAV
`[0 0 0]` (default) | vector of form `[roll pitch yaw]`

Specify the orientation of the sensor mounted on the UAV as a vector of the form `[roll pitch yaw]`, with values in degrees.

Data Types: `double`

Vehicle radius (m) — Radius of UAV
`1` (default) | numeric scalar

This dimension defines the smallest circle that can circumscribe your vehicle, in meters. The vehicle radius is used to account for vehicle size when computing the obstacle-free direction.

Data Types: `double`

Minimum distance to obstacle (m) — Safety distance around UAV to obstacle
`1` (default) | numeric scalar

The safety distance specifies, in meters, the space accounted for between the UAV and obstacles in addition to the vehicle radius. The vehicle radius and safety distance are used to compute the obstacle-free direction.

Data Types: `double`

Simulate using — Type of simulation to run
`Interpreted execution` (default) | `Code generation`

Specify whether to simulate the model using `Interpreted execution` or `Code generation`.

- `Interpreted execution` — Simulate the model using the MATLAB interpreter. This option reduces startup time, but has a slower simulation speed than `Code generation`. In this 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 this 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 subsequent simulations is comparable to `Interpreted execution`.

Tunable: No

Histogram

Histogram resolution (deg) — Histogram grid resolution
`5` (default) | `1` | `3` | `6` | `10` | `15` | `18` | `30` | `45` | `60`

To change the histogram grid resolution, select a value from the list. All values are in degrees.

Histogram window size — Histogram window size
`1` (default) | odd integer

The histogram window size determines the angular width of an obstacle-free opening in the azimuth and elevation directions. This value is unitless.

Data Types: `uint8`

Histogram threshold — Threshold for computing histogram
`1` (default) | positive integer

The threshold for computing the histogram specifies the minimum number of obstacle points that should be in an histogram cell to be considered as obstacle. If a cell contains fewer than this number of obstacle points, the cell is considered as obstacle-free.

Data Types: `uint8`

Maximum age of obstacle point — Maximum age of remembered obstacle point
`0` (default) | numeric scalar

Specifies the maximum age of a remembered obstacle point as a numeric scalar. This value specify the number of previous time steps for which the obstacle points from those time steps is remembered.

Data Types: `double`

Cost

Target direction weight — Cost function weight for target direction
`5` (default) | numeric scalar

Specifies the function weight for moving toward the target direction. To follow a target direction, set this weight to be greater than the sum of `Current direction weight` and `Previous direction weight`. To ignore the target direction cost, set this weight to `0`.

Data Types: `double`

Current direction weight — Cost function weight for current direction
`2` (default) | numeric scalar

Specifies the function weight for moving the vehicle in the current heading directions. Higher values of this weight produce efficient paths. To ignore the current direction cost, set this weight to `0`.

Data Types: `double`

Previous direction weight — Cost function weight for previous direction
`2` (default) | numeric scalar

Specifies the function weight for moving in the previously selected steering direction. Higher values of this weight produce smoother paths. To ignore the previous direction cost, set this weight to `0`.

Data Types: `double`

Version History

Introduced in R2021b

Extended Capabilities

C/C++ Code Generation

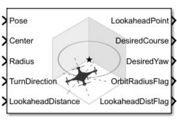
Generate C and C++ code using Simulink® Coder™.

See Also

Waypoint Follower | Guidance Model

Orbit Follower

Orbit location of interest using UAV



Libraries:
UAV Toolbox / Algorithms

Description

The Orbit Follower block generates course and yaw controls for following a circular orbit around a location of interest based on the current pose of the unmanned aerial vehicle (UAV). Select a **UAV type** of fixed-wing or multirotor UAVs. You can specify any orbit center location, orbit radius, and turn direction. A lookahead distance, **LookaheadDistance**, is used for tuning the path tracking and generating the **LookaheadPoint** output.

Ports

Input

Pose — Current UAV pose
`[x y z course]` vector

Current UAV pose, specified as an `[x y z course]` vector. `[x y z]` is the position of the UAV in NED coordinates (north-east-down) specified in meters. `course` is the angle between ground velocity and north direction in radians per second.

Example: `[1,1,-10,pi/4]`

Data Types: `single` | `double`

Center — Center of orbit
`[x y z]` vector

Center of orbit, specified as an `[x y z]` vector. `[x y z]` is the orbit center position in NED coordinates (north-east-down) specified in meters.

Example: `[5,5,-10]`

Data Types: `single` | `double`

Radius — Radius of orbit
positive scalar

Radius of orbit, specified as a positive scalar in meters.

Example: `5`

Data Types: `single` | `double`

TurnDirection — Direction of orbit
scalar

Direction of orbit, specified as a scalar. Positive values indicate a clockwise turn as viewed from above. Negative values indicate a counter-clockwise turn. A value of `0` automatically determines the value based on the input to **Pose**.

Example: `-1`

Data Types: `single` | `double`

LookaheadDistance — Lookahead distance for tracking orbit
positive scalar

Lookahead distance for tracking the orbit, specified as a positive scalar. Tuning this value helps adjust how tightly the UAV follows the orbit circle. Smaller values improve tracking, but can lead to oscillations in the path.

Example: 2

Data Types: `single` | `double`

ResetNumTurns — Reset for counting turns
numeric signal

Reset for counting turns, specified as a numeric signal. Any rising signal triggers a reset of the **NumTurns** output.

Example: 2

Dependencies

To enable this input, select `rising` for **External reset**.

Data Types: `single` | `double`

Output

LookaheadPoint — Lookahead point on path
`[x y z]` position vector

Lookahead point on path, returned as an `[x y z]` position vector in meters.

Data Types: `double`

DesiredCourse — Desired course
numeric scalar

Desired course, returned as numeric scalar in radians in the range of `[-pi, pi]`. The UAV course is the angle of direction of the velocity vector relative to north measured in radians. For fixed-wing type UAV, the values of desired course and desired yaw are equal.

Data Types: `double`

DesiredYaw — Desired yaw
numeric scalar

Desired yaw, returned as numeric scalar in radians in the range of `[-pi, pi]`. The UAV yaw is the forward direction of the UAV (regardless of the velocity vector) relative to north measured in radians. For fixed-wing type UAV, the values of desired course and desired yaw are equal.

Data Types: `double`

OrbitRadiusFlag — Orbit radius flag
`0` (default) | `1`

Orbit radius flag, returned as `0` or `1`. `0` indicates orbit radius is not saturated, `1` indicates orbit radius is saturated to minimum orbit radius value specified.

Data Types: `uint8`

LookaheadDistFlag — Lookahead distance flag
`0` (default) | `1`

Lookahead distance flag, returned as `0` or `1`. `0` indicates lookahead distance is not saturated, `1` indicates lookahead distance is saturated to minimum lookahead distance value specified.

Data Types: `uint8`

CrossTrackError — Cross track error from UAV position to path
positive numeric scalar

Cross track error from UAV position to path, returned as a positive numeric scalar in meters. The error measures the perpendicular distance from the UAV position to the closest point on the path.

Dependencies

This port is only visible if **Show CrossTrackError output port** is checked.

Data Types: `double`

NumTurns — Number of times the UAV has completed the orbit
numeric scalar

Number of times the UAV has completed the orbit, returned as a numeric scalar. As the UAV circles the center point, this value increases or decreases based on the specified **Turn Direction**. Decimal values indicate partial completion of a circle. If the UAV cross track error exceeds the lookahead distance, the number of turns is not updated.

NumTurns is reset whenever **Center**, **Radius**, or **TurnDirection** are changed. You can also use the **ResetNumTurns** input.

Dependencies

This port is only visible if **Show NumTurns output port** is checked.

Parameters

UAV type — Type of UAV
`fixed-wing` (default) | `multirotor`

Type of UAV, specified as either `fixed-wing` or `multirotor`.

This parameter is non-tunable.

Minimum orbit radius (m) — Minimum orbit radius
`1` (default) | positive numeric scalar

Minimum orbit radius, specified as a positive numeric scalar in meters.

When input to the orbit **Radius** port is less than the minimum orbit radius, the **OrbitRadiusFlag** is returned as `1` and the orbit radius value is specified as the value of minimum orbit radius.

This parameter is non-tunable.

Minimum lookahead distance (m) — Minimum lookahead distance
`0.1` (default) | positive numeric scalar

Minimum lookahead distance, specified as a positive numeric scalar in meters.

When input to the **LookaheadDistance** port is less than the minimum lookahead distance, the **LookaheadDistFlag** is returned as `1` and the lookahead distance value is specified as the value of minimum lookahead distance.

This parameter is non-tunable.

External reset — Reset trigger source
`none` (default) | `rising`

Select `rising` to enable the **ResetNumTurns** block input.

This parameter is non-tunable.

Show CrossTrackError output port — Output cross track error
`off` (default) | `on`

Output cross track error from the **CrossTrackError** port.

This parameter is non-tunable.

Show NumTurns output port — Output UAV waypoint status
`off` (default) | `on`

Output UAV waypoint status from the **Status** port.

This parameter is non-tunable.

Version History

Introduced in R2019a

Extended Capabilities

C/C++ Code Generation

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

See Also

Blocks

Waypoint Follower | UAV Guidance Model

Functions

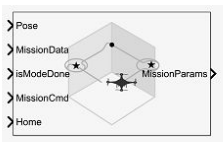
`ode45` | `control` | `derivative` | `environment` | `state` | `plotTransforms`

Objects
`uavOrbitFollower` | `uavWaypointFollower` | `fixedwing` | `multirotor`

Topics
"Approximate High-Fidelity UAV Model with UAV Guidance Model Block"
"Tuning Waypoint Follower for Fixed-Wing UAV"

Path Manager

Compute and execute a UAV autonomous mission



Libraries:
UAV Toolbox / Algorithms

Description

The Path Manager block computes mission parameters for an unmanned aerial vehicle (UAV) by sequentially switching between mission points specified in the **MissionData** input port. The **MissionCmd** input port changes the execution order at runtime. The block supports both multirotor and fixed-wing UAV types.

Ports

Input

Pose — Current UAV pose
four-element column vector

Current UAV pose, specified as a four-element column vector of `[x;y;z;courseAngle]`. *x*, *y*, and *z* is the current position of the UAV in north-east-down (NED) coordinates specified in meters. *courseAngle* specifies the course angle in radians in the range `[-pi, pi]`.

Data Types: `single` | `double`

MissionData — UAV mission data
`UAVPathManagerBus` bus

UAV mission data, specified as a `UAVPathManagerBus` bus. The `UAVPathManagerBus` bus has the three bus elements `mode`, `position`, and `params`.

You can use the Constant block to specify the mission data as an *n*-by-1 array of structures and set the output data type to `Bus:UAVPathManagerBus`. *n* is the number of mission points. The fields of each structure are:

- `mode` — Mode of the mission point, specified as an 8-bit unsigned integer between 1 and 6.
- `position` — Position of the mission point, specified as a three-element column vector of `[x;y;z]`. *x*, *y*, and *z* is the position in north-east-down (NED) coordinates specified in meters.
- `params` — Parameters of the mission point, specified as a four-element column vector.

The values assigned to the fields, in turn, are assigned to their corresponding bus elements in the `UAVPathManagerBus` bus.

This table describes the types of `mode` and the corresponding values for the `position` and `params` fields in a mission point structure.

mode	position	params	Mode description
uint8(1)	[*x*;*y*;*z*]	[*p1*;*p2*;*p3*;*p4*]	**Takeoff** — Take off from the ground and travel toward the specified position
uint8(2)	[*x*;*y*;*z*]	[*yaw*;*radius*;*p3*;*p4*] *yaw* — Yaw angle in radians in the range [-pi, pi] *radius* — Transition radius in meters	**Waypoint** — Navigate to waypoint
uint8(3)	[*x*;*y*;*z*] *x*, *y*, and *z* is the center of the circular orbit in NED coordinates specified in meters	[*radius*;*turnDir*;*numTurns*;*p4*] *radius* — Radius of the orbit in meters *turnDir* — Turn direction, specified as one of these: • 1 — Clockwise turn • –1 — Counter-clockwise turn • 0 — Automatic selection of turn direction *numTurns* — Number of turns	**Orbit** — Orbit along the circumference of a circle defined by the parameters
uint8(4)	[*x*;*y*;*z*]	[*p1*;*p2*;*p3*;*p4*]	**Land** — Land at the specified position
uint8(5)	[*x*;*y*;*z*] The launch position is specified in the Home input port	[*p1*;*p2*;*p3*;*p4*]	**RTL** — Return to launch position
uint8(6)	[*x*;*y*;*z*]	[*p1*;*p2*;*p3*;*p4*] *p1*, *p2*, *p3*, and *p4* are user-specified parameters corresponding to a custom mission point	**Custom** — Custom mission point

Note *p1*, *p2*, *p3*, and *p4* are user-specified parameters.

Example: `[struct('mode',uint8(1),'position',[0;0;100],'params',[0;0;0;0])]`

Data Types: `bus`

IsModeDone — Determine if mission point was executed
`0` (default) | `1`

Determine if the mission point was executed, specified as `0` (`false`) or `1` (`true`).

Data Types: `Boolean`

MissionCmd — Command to change mission
`uint8(0)` (default) | 8-bit unsigned integer between 0 and 3

Command to change mission at runtime, specified as an 8-bit unsigned integer between 0 and 3.

This table describes the possible mission commands.

Mission Command	Description
`uint8(0)`	**Default** — Execute the mission from first to the last mission point in the sequence
`uint8(1)`	**Hold** — Hold at the current mission point Loiter around the current position for fixed-wing and hover at the current position for multirotor UAVs
`uint8(2)`	**Repeat** — Repeat the mission after reaching the last mission point
`uint8(3)`	**RTL** — Execute return to launch (RTL) mode After **RTL**, the mission resumes if the `MissionCmd` input is changed to **Default** or **Repeat**

Data Types: `uint8`

Home — UAV home location
three-element column vector

UAV home location, specified as a three-element column vector of `[x;y;z]`. *x*, *y*, and *z* is the position in north-east-down (NED) coordinates specified in meters.

Data Types: `single` | `double`

Output

MissionParams — UAV mission parameters
`UAVPathManagerBus` bus

UAV mission parameters, returned as a 2-by-1 array of buses of the type `UAVPathManagerBus`. The first element of the bus array is the current mission point, and the second element of the bus array is the previous mission point.

This table describes the output mission parameters depending on the mission mode.

Current Mission Mode	Output Mission Parameters			
	Mission Points	**`mode`**	**`position`**	**`params`**
Takeoff	First bus element: Current	`uint8(1)`	`[`*`x`*`;`*`y`*`;`*`z`*`]`	`[`*`p1`*`;`*`p2`*`;`*`p3`*`;`*`p4`*`]`
	Second bus element: Previous	`mode` of the previous mission point	`position` of the previous mission point	`params` of the previous mission point
Waypoint	First bus element: Current	`uint8(2)`	`[`*`x`*`;`*`y`*`;`*`z`*`]`	`[`*`yaw`*`;`*`radius`*`;`*`p3`*`;`*`p4`*`]` *yaw* — Yaw angle in radians in the range `[-pi, pi]` *radius* — Transition radius in meters
	Second bus element: Previous	`mode` of the previous mission point	`position` of the previous mission point	• `[`*`yaw`*`;`*`radius`*`;`*`p3`*`;`*`p4`*`]` if the previous mission point was **Takeoff** • `[`*`courseAngle`*`;25;`*`p3`*`;`*`p4`*`]` otherwise *courseAngle* — Angle of the line segment between the previous and the current position, specified in radians in the range `[-pi, pi]`

Current Mission Mode	Output Mission Parameters			
	Mission Points	mode	position	params
Orbit	First bus element: Current	`uint8(3)`	`[x;y;z]` *x*, *y*, and *z* is the center of the circular orbit in NED coordinates specified in meters	`[radius;turnDir;numTurns;p4]` *radius* — Radius of the orbit in meters *turnDir* — Turn direction, specified as one of these: • `1` — Clockwise turn • `–1` — Counter-clockwise turn • `0` — Automatic selection of turn direction *numTurns* — Number of turns
	Second bus element: Previous	`mode` of the previous mission point	`position` of the previous mission point	`params` of the previous mission point
Land	First bus element: Current	`uint8(4)`	`[x;y;z]`	`[p1;p2;p3;p4]`
	Second bus element: Previous	`mode` of the previous mission point	`position` of the previous mission point	`params` of the previous mission point
RTL	First bus element: Current	`uint8(5)`	`[x;y;z]` The launch position is specified in the `Home` input port	`[p1;p2;p3;p4]`
	Second bus element: Previous	`mode` of the previous mission point	`position` of the previous mission point	`params` of the previous mission point
Custom	First bus element: Current	`uint8(6)`	`[x;y;z]`	`[p1;p2;p3;p4]` *p1*, *p2*, *p3*, and *p4* are user-specified parameters corresponding to a custom mission point

Current Mission Mode	Output Mission Parameters			
	Mission Points	**mode**	**position**	**params**
	Second bus element: Previous	`mode` of the previous mission point	`position` of the previous mission point	`params` of the previous mission point

Note *p1*, *p2*, *p3*, and *p4* are user-specified parameters.

At start of simulation, the previous mission point is set to the **Armed** mode.

mode	position	params
`uint8(0)`	`[x;y;z]` `position` of the UAV at simulation start.	`[-1;-1;-1;-1]`

Set the end mission point to **RTL** or **Land** mode, else the end mission point is automatically set to **Hold** mode.

This table describes the output mission parameters when the input to the **MissionCmd** input port is set to **Hold** mode.

UAV Type	Output Mission Parameters			
	Mission Points	**mode**	**position**	**params**
Multirotor	First bus element: Current	`uint8(7)`	`[x;y;z]`	`[-1;-1;-1;-1]`
	Second bus element: Previous	`mode` of the previous mission point	`position` of the previous mission point	`params` of the previous mission point
Fixed-Wing	First bus element: Current	`uint8(7)`	`[x;y;z]` *x*, *y*, and *z* is the center of the circular orbit in NED coordinates specified in meters	`[radius;turnDir;-1;-1]` *radius* — Loiter radius is specified in the `Loiter radius` parameter *turnDir* — Turn direction is specified as `0` for automatic selection of turn direction
	Second bus element: Previous	`mode` of the previous mission point	`position` of the previous mission point	`params` of the previous mission point

Data Types: `bus`

Parameters

UAV type — Type of UAV
`multirotor` (default) | `fixed-wing`

Type of UAV, specified as either `multirotor` or `fixed-wing`.

Tunable: No

Loiter radius — Loiter radius for fixed-wing UAV
25 (default) | positive numeric scalar

Loiter radius for the fixed-wing UAV, specified as a positive numeric scalar in meters.

Dependencies: To enable this parameter, set the **UAV type** parameter to `fixed-wing`.

Tunable: No

Data type — Data type of input mission bus
`double` (default) | `single`

Data type of the input mission bus, specified as either `double` or `single`.

Tunable: No

Mission bus name — Name of input mission bus
`'UAVPathManagerBus'` (default)

Name of the input mission bus, specified as `'UAVPathManagerBus'`.

Tunable: No

Version History

Introduced in R2020b

Extended Capabilities

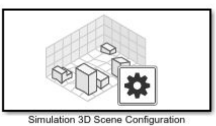
C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

See Also

Guidance Model | Orbit Follower | Waypoint Follower

Simulation 3D Scene Configuration

Scene configuration for 3D simulation environment



Libraries:
UAV Toolbox / Simulation 3D
Aerospace Blockset / Animation / Simulation 3D
Automated Driving Toolbox / Simulation 3D
Vehicle Dynamics Blockset / Vehicle Scenarios / Sim3D / Sim3D Core
Simulink 3D Animation / Simulation 3D

Description

The Simulation 3D Scene Configuration block implements a 3D simulation environment that is rendered by using the Unreal Engine from Epic Games. UAV Toolbox integrates the 3D simulation environment with Simulink so that you can query the world around the vehicle and virtually test perception, control, and planning algorithms.

You can simulate from a set of prebuilt scene or from your own custom scenes. Scene customization requires the UAV Toolbox Interface for Unreal Engine Projects support package. For more details, see “Customize Unreal Engine Scenes for UAVs”.

Note The Simulation 3D Scene Configuration block must execute after blocks that send data to the 3D environment and before blocks that receive data from the 3D environment. To verify the execution order of such blocks, right-click the blocks and select **Properties**. Then, on the **General** tab, confirm these **Priority** settings:

- For blocks that send data to the 3D environment, such as Simulation 3D Vehicle with Ground Following blocks, **Priority** must be set to `-1`. That way, these blocks prepare their data before the 3D environment receives it.
- For the Simulation 3D Scene Configuration block in your model, **Priority** must be set to `0`.
- For blocks that receive data from the 3D environment, such as blocks, **Priority** must be set to `1`. That way, the 3D environment can prepare the data before these blocks receive it.

For more information about execution order, see “Block Execution Order”.

Parameters

Scene Selection

Scene source — Source of scene

`Default Scene` (default) | `Unreal Executable` | `Unreal Editor`

Source of the scene in which to simulate, specified as one of the options in the table.

Option	Description
`Default Scene`	Simulate in the default, prebuilt scene specified in the **Scene name** parameter.
`Unreal Executable`	Simulate in a scene that is part of an Unreal Engine executable file. Specify the executable file in the **File name** parameter. Specify the scene in the **Scene** parameter. Select this option to simulate in custom scenes that have been packaged into an executable for faster simulation.
`Unreal Editor`	Simulate in a scene that is part of an Unreal Engine project (`.uproject`) file and is open in the Unreal Editor. Specify the project file in the **Project** parameter. Select this option when developing custom scenes. By clicking **Open Unreal Editor**, you can co-simulate within Simulink and the Unreal Editor and modify your scenes based on the simulation results.

Scene name — Name of prebuilt 3D scene

`US city block` (default)

Name of the prebuilt 3D scene in which to simulate, specified as one of these options. For details about a scene, see its listed corresponding reference page.

- `US city block` — **US City Block**

The UAV Toolbox Interface for Unreal Engine Projects contains customizable versions of these scenes. For details about customizing scenes, see “Customize Unreal Engine Scenes for UAVs”.

Dependencies

To enable this parameter, set **Scene source** to `Default Scene`.

File name — Name of Unreal Engine executable file

`VehicleSimulation.exe` (default) | valid executable file name

Name of the Unreal Engine executable file, specified as a valid executable file name. You can either browse for the file or specify the full path to the file, using backslashes. To specify a scene from this file to simulate in, use the **Scene** parameter.

By default, **File name** is set to `VehicleSimulation.exe`, which is on the MATLAB search path.

Example: `C:\Local\WindowsNoEditor\AutoVrtlEnv.exe`

Dependencies

To enable this parameter, set **Scene source** to `Unreal Executable`.

Scene — Name of scene from executable file

`/Game/Maps/USCityBlock` (default) | path to valid scene name

Name of a scene from the executable file specified by the **File name** parameter, specified as a path to a valid scene name.

When you package scenes from an Unreal Engine project into an executable file, the Unreal Editor saves the scenes to an internal folder within the executable file. This folder is located at the path `/Game/Maps`. Therefore, you must prepend `/Game/Maps` to the scene name. You must specify this path using forward slashes. For the file name, do not specify the `.umap` extension. For example, if the scene from the executable in which you want to simulate is named `myScene.umap`, specify **Scene** as `/Game/Maps/myScene`.

Alternatively, you can browse for the scene in the corresponding Unreal Engine project. These scenes are typically saved to the `Content/Maps` subfolder of the project. This subfolder contains all the scenes in your project. The scenes have the extension `.umap`. Select one of the scenes that you packaged into the executable file specified by the **File name** parameter. Use backward slashes and specify the `.umap` extension for the scene.

By default, **Scene** is set to `/Game/Maps/USCityBlock`, which is a scene from the default `VehicleSimulation.exe` executable file specified by the **File name** parameter. This scene corresponds to the prebuilt **Straight Road** scene.

Example: `/Game/Maps/scene1`

Example: `C:\Local\myProject\Content\Maps\scene1.umap`

Dependencies

To enable this parameter, set **Scene source** to `Unreal Executable`.

Project — Name of Unreal Engine project file

valid project file name

Name of the Unreal Engine project file, specified as a valid project file name. You can either browse for the file or specify the full path to the file, using backslashes. The file must contain no spaces. To simulate scenes from this project in the Unreal Editor, click **Open Unreal Editor**. If you have an Unreal Editor session open already, then this button is disabled.

To run the simulation, in Simulink, click **Run**. Before you click **Play** in the Unreal Editor, wait until the Diagnostic Viewer window displays this confirmation message:

```
In the Simulation 3D Scene Configuration block, you set the scene source to 'Unreal Editor'.
In Unreal Editor, select 'Play' to view the scene.
```

This message confirms that Simulink has instantiated the scene actors, including the vehicles and cameras, in the Unreal Engine 3D environment. If you click **Play** before the Diagnostic Viewer window displays this confirmation message, Simulink might not instantiate the actors in the Unreal Editor.

Dependencies

To enable this parameter, set **Scene source** to `Unreal Editor`.

Scene Parameters

Scene view — Configure placement of virtual camera that displays scene

`Scene Origin` (default) | vehicle name

Configure the placement of the virtual camera that displays the scene during simulation.

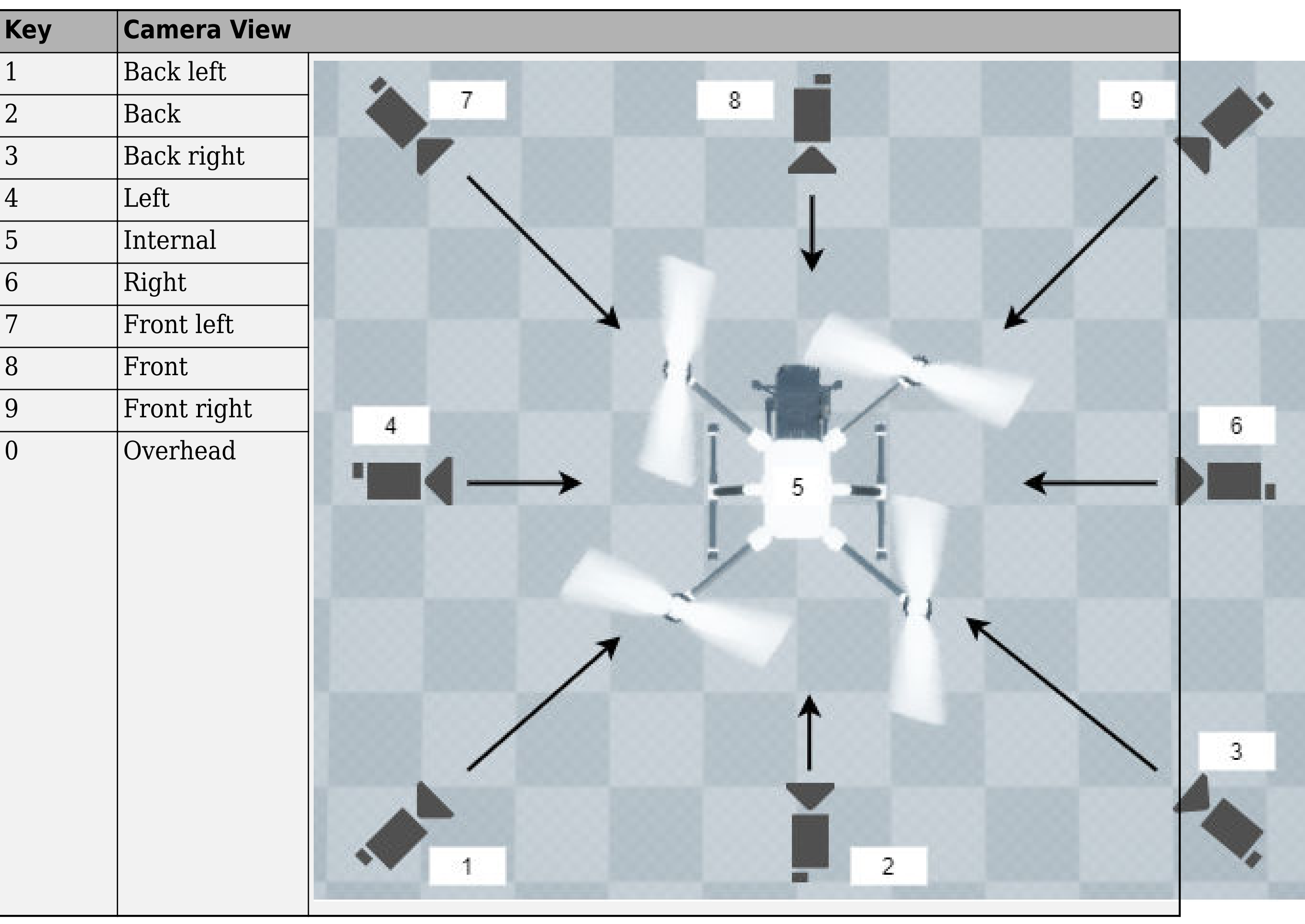
- If your model contains no Simulation 3D UAV Vehicle blocks, then during simulation, you view the scene from a camera positioned at the scene origin.
- If your model contains at least one vehicle block, then by default, you view the scene from behind the first vehicle that was placed in your model. To change the view to a different vehicle, set **Scene view** to the name of that vehicle. The **Scene view** parameter list is populated with all the **Name** parameter values of the vehicle blocks contained in your model.

If you add a Simulation 3D Scene Configuration block to your model before adding any vehicle blocks, the virtual camera remains positioned at the scene. To reposition the camera to follow a vehicle, update this parameter.

When **Scene view** is set to a vehicle name, during simulation, you can change the location of the camera around the vehicle.

To smoothly change the camera views, use these key commands.

Key	Camera View
1	Back left
2	Back
3	Back right
4	Left
5	Internal
6	Right
7	Front left
8	Front
9	Front right
0	Overhead



For additional camera controls, use these key commands.

Key	Camera Control
Tab	Cycle the view between all vehicles in the scene.
Mouse scroll wheel	Control the camera distance from the vehicle.
L	Toggle a camera lag effect on or off. When you enable the lag effect, the camera view includes: • Position lag, based on the vehicle translational acceleration • Rotation lag, based on the vehicle rotational velocity This lag enables improved visualization of overall vehicle acceleration and rotation.
F	Toggle the free camera mode on or off. When you enable the free camera mode, you can use the mouse to change the pitch and yaw of the camera. This mode enables you to orbit the camera around the vehicle.

Sample time — Sample time of visualization engine

(default) | scalar greater than or equal to 0.01

Sample time, T_s, of the visualization engine, specified as a scalar greater than or equal to 0.01. Units are in seconds.

The graphics frame rate of the visualization engine is the inverse of the sample time. For example, if **Sample time** is `1/60`, then the visualization engine solver tries to achieve a frame rate of 60 frames per second. However, the real-time graphics frame rate is often lower due to factors such as graphics card performance and model complexity.

By default, blocks that receive data from the visualization engine, such as Simulation 3D Camera blocks, inherit this sample rate.

Display 3D simulation window — Unreal Engine visualization

`on` (default) | `off`

Select whether to run simulations in the 3D visualization environment without visualizing the results, that is, in headless mode.

Consider running in headless mode in these cases:

- You want to run multiple 3D simulations in parallel to test models in different Unreal Engine scenarios.

Dependencies

To enable this parameter, set **Scene source** to `Default Scene` or `Unreal Executable`.

Display 3D simulation window in a web browser — Web browser visualization

`off` (default) | `on`

Select whether to run simulations in web browsers on local or remote devices, including:

- Local desktops
- Remote desktops
- Mobile phones

To display the simulation in a web browser:

1. If you do not have it installed, install Node.js® on the system that runs the simulation.
2. Select **Display 3D simulation window in a web browser**. Apply the change.
3. In Simulink, select **Run**.
4. Follow the steps provided in the Diagnostic Viewer.

Web Browser Display	Steps
Current device	a. Open a web browser on your current device. b. Navigate to the first *`IP address`* link provided in the Diagnostic Viewer.

Web Browser Display	Steps
Remote device	**a** Open a web browser on a remote device that is on the same network. **b** Navigate to the second *`IP address`* link provided in the Diagnostic Viewer.

Note To establish the connections, the system server uses two ports:

- `7070` – *`http:`* connection to web browser
- `8888`– Streamer connection from Unreal Engine application

Dependencies

To enable this parameter, set **Scene source** to `Default Scenes` or `Unreal Executable`.

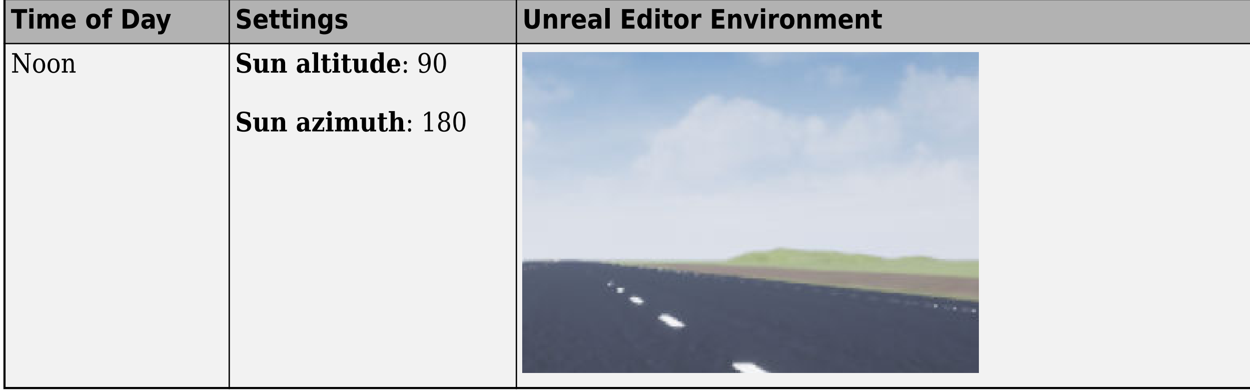
Weather

Override scene weather — Control the scene weather and sun position
`off` (default) | `on`

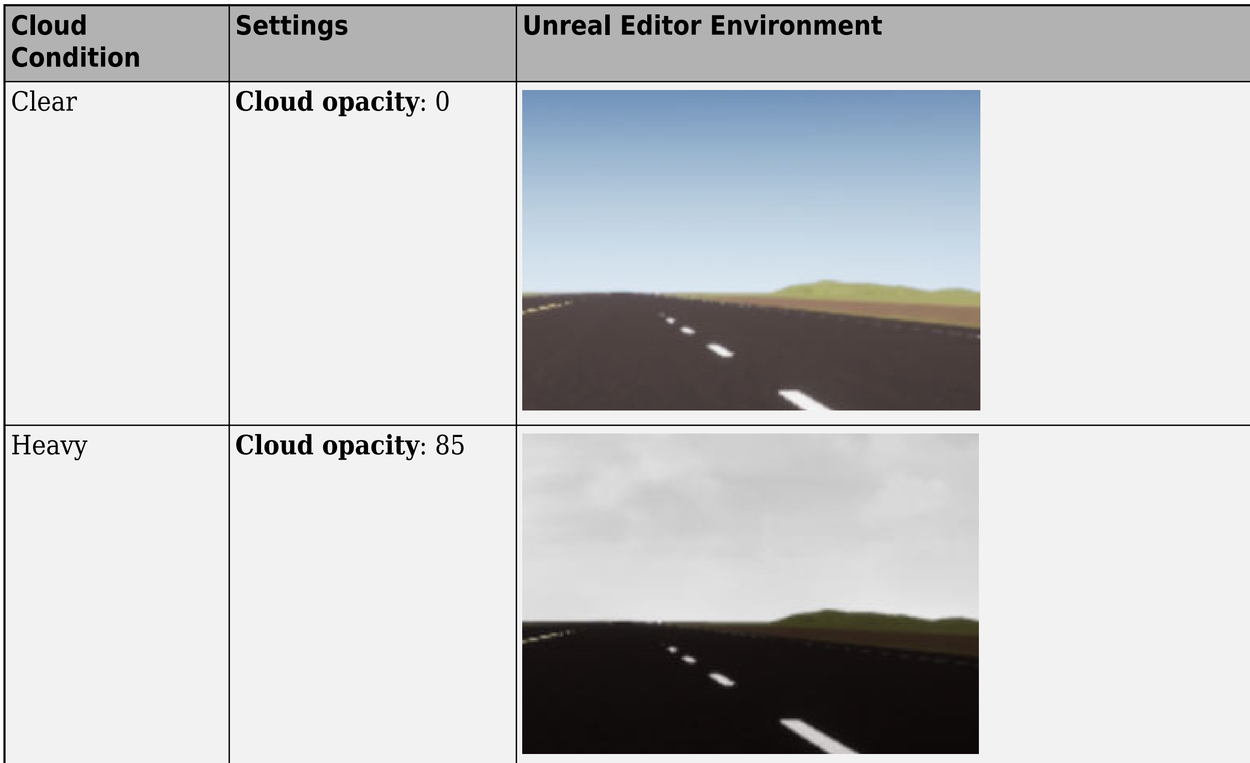
Select whether to control the scene weather and sun position during simulation. Use the enabled parameters to change the sun position, clouds, fog, and rain.

This table summarizes sun position settings for specific times of day.

Time of Day	Settings	Unreal Editor Environment
Midnight	**Sun altitude**: -90 **Sun azimuth**: 180	
Sunrise in the north	**Sun altitude**: 0 **Sun azimuth**: 180	

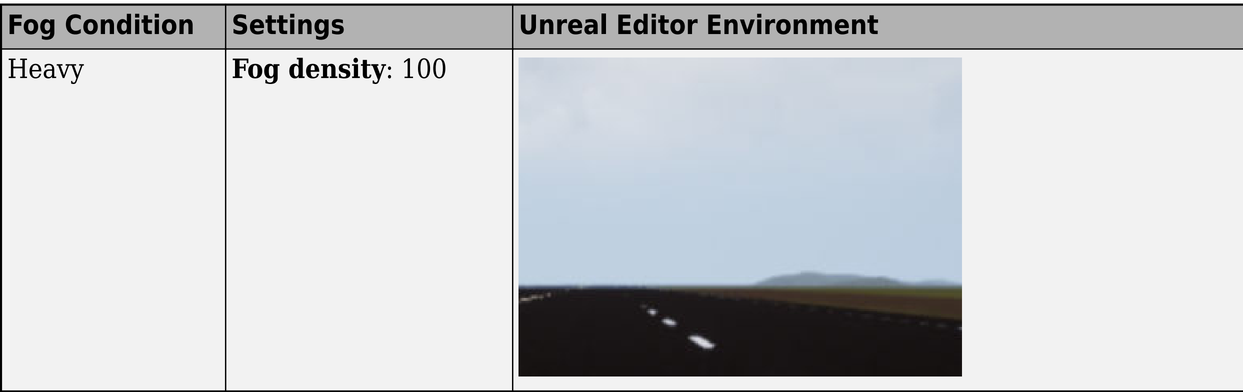
Time of Day	Settings	Unreal Editor Environment
Noon	**Sun altitude**: 90 **Sun azimuth**: 180	

This table summarizes settings for specific cloud conditions.

Cloud Condition	Settings	Unreal Editor Environment
Clear	**Cloud opacity**: 0	
Heavy	**Cloud opacity**: 85	

This table summarizes settings for specific fog conditions.

Fog Condition	Settings	Unreal Editor Environment
None	**Fog density**: 0	

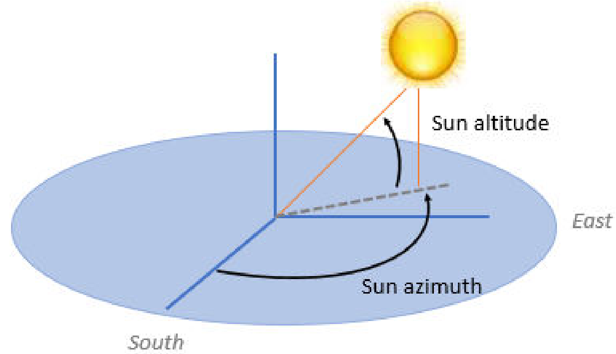
Fog Condition	Settings	Unreal Editor Environment
Heavy	**Fog density**: 100	

This table summarizes settings for specific rain conditions.

Rain Condition	Settings	Unreal Editor Environment
Light	**Cloud opacity**: 10 **Rain density**: 25	
Heavy	**Cloud opacity**: 10 **Rain density**: 80	

Sun altitude — Altitude angle between sun and horizon
`40` (default) | any value between `-90` and `90`

Altitude angle in a vertical plane between the sun's rays and the horizontal projection of the rays, in deg.



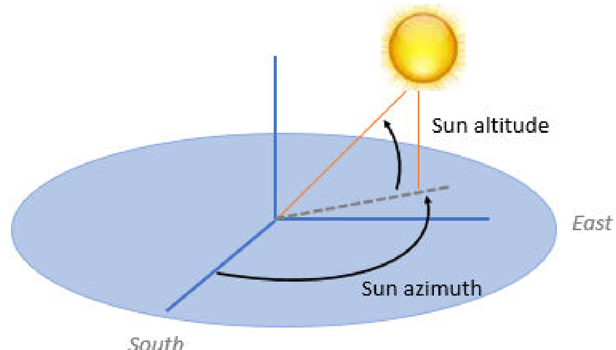
Use the **Sun altitude** and **Sun azimuth** parameters to control the time of day in the scene. For example, to specify sunrise in the north, set **Sun altitude** to 0 deg and **Sun azimuth** to 180 deg.

Dependencies

To enable this parameter, select **Override scene weather**.

Sun azimuth — Azimuth angle from south to horizontal projection of the sun ray
90 (default) | any value between 0 and 360

Azimuth angle in the horizontal plane measured from the south to the horizontal projection of the sun rays, in deg.



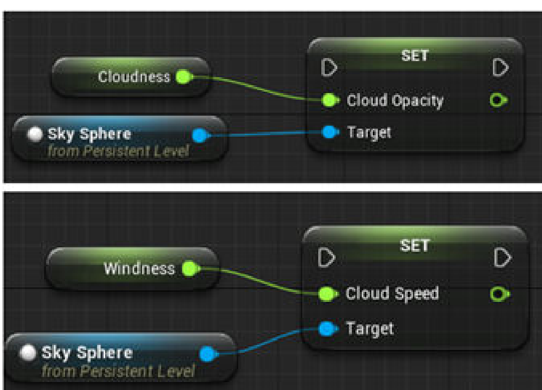
Use the **Sun altitude** and **Sun azimuth** parameters to control the time of day in the scene. For example, to specify sunrise in the north, set **Sun altitude** to 0 deg and **Sun azimuth** to 180 deg.

Dependencies

To enable this parameter, select **Override scene weather**.

Cloud opacity — Unreal Editor Cloud Opacity global actor target value
10 (default) | any value between 0 and 100

Parameter that corresponds to the Unreal Editor **Cloud Opacity** global actor target value, in percent. Zero is a cloudless scene.



Use the **Cloud opacity** and **Cloud speed** parameters to control clouds in the scene.

Dependencies

To enable this parameter, select **Override scene weather**.

Cloud speed — Unreal Editor Cloud Speed global actor target value
`1` (default) | any value between `-100` and `100`

Parameter that corresponds to the Unreal Editor **Cloud Speed** global actor target value. The clouds move from west to east for positive values and east to west for negative values.

Use the **Cloud opacity** and **Cloud speed** parameters to control clouds in the scene.

Dependencies

To enable this parameter, select **Override scene weather**.

Fog density — Unreal Editor Set Fog Density and Set Start Distance target values
`0` (default) | any value between `0` and `100`

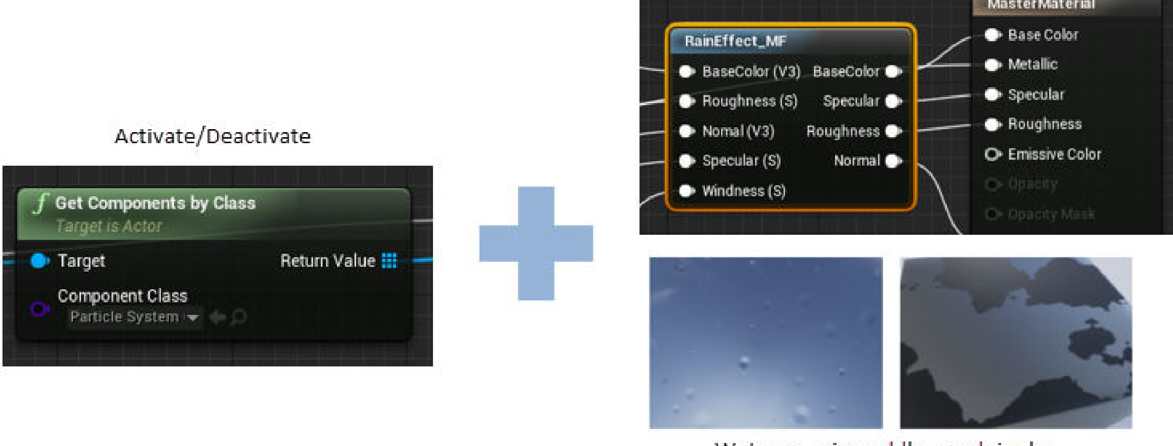
Parameter that corresponds to the Unreal Editor **Set Fog Density** and **Set Start Distance** target values, in percent.

Dependencies

To enable this parameter, select **Override scene weather**.

Rain density — Unreal Editor local actor controlling rain density, wetness, rain puddles, and ripples
`0` (default) | any value between `0` and `100`

Parameter corresponding to the Unreal Editor local actor that controls rain density, wetness, rain puddles, and ripples, in percent.



Use the **Cloud opacity** and **Rain density** parameters to control rain in the scene.

Dependencies

To enable this parameter, select **Override scene weather**.

Geospatial

Enable geospatial configuration — Option to enable geospatial parameters and variant subsystem
`off` (default) | `on`

Select this parameter to enable geospatial parameters and a variant subsystem.

Access token ID — ID of stored token
string representing ID of stored token

Specify your Cesium access token ID. To create this token, create a Cesium® ion account, then generate the token through this account. For more information, see https://cesium.com/ion.

Dependencies

To enable this parameter, select the **Enable geospatial configuration** parameter.

Origin height (m) — Height of origin at georeference point on globe
`200` (default) | real scalar

Specify the height of the origin, in meters, above the 1984 World Geodetic System (WGS84) ellipsoid model of the Earth at the latitude and longitude specified in **Origin latitude** and **Origin longitude**.

Dependencies

To enable this parameter, select the **Enable geospatial configuration** parameter.

Origin latitude — Latitude of origin
`40.744652` (default) | real scalar

Specify the latitude of the origin in decimal degrees.

Dependencies

To enable this parameter, select the **Enable geospatial configuration** parameter.

Origin longitude — Longitude of origin
`-73.988864` (default) | real scalar

Specify the longitude of the origin in decimal degrees.

Dependencies

To enable this parameter, select the **Enable geospatial configuration** parameter.

Map style — Raster overlay type
`Aerial` (default) | `Aerial with labels` | `Road`

Specify the raster overlay type of the map as `Aerial`, `Aerial with labels`, or `Road`.

Dependencies

To enable this parameter, select the **Enable geospatial configuration** parameter.

Additional asset IDs — Local data set IDs
`[]` (default) | array | vector

Specify the local data set IDs.

Dependencies

To enable this parameter, select the **Enable geospatial configuration** parameter.

Use advanced Sun sky — Georeferenced, location-accurate Sun Sky actor
`off` (default) | on

Select this parameter to add a georeferenced, location-accurate Sun Sky actor to the simulation.

Dependencies

To enable this parameter, select the **Enable geospatial configuration** parameter.

Solar time — Current solar time
`11` (default) | real scalar

Specify the current solar time in hours from midnight.

Dependencies

To enable this parameter, select the **Enable geospatial configuration** and **Use advanced Sun sky** parameters.

Time zone — Time zone
`-5` (default) | real scalar

Specify the current time zone in hours offset from Greenwich Mean Time (GMT). To specify time zones earlier than GMT, use a negative value.

Dependencies

To enable this parameter, select the **Enable geospatial configuration** and **Use advanced Sun sky** parameters.

Day — Day
21 (default) | integer in range [1, 31]

Specify the day of the month, from 1 to 31.

Dependencies

To enable this parameter, select the **Enable geospatial configuration** and **Use advanced Sun sky** parameters.

Month — Month
9 (default) | integer in range [1, 12]

Specify the month, from 1 to 12.

Dependencies

To enable this parameter, select the **Enable geospatial configuration** and **Use advanced Sun sky** parameters.

Year — Year
2022 (default) | real scalar

Specify the year.

Dependencies

To enable this parameter, select the **Enable geospatial configuration** and **Use advanced Sun sky** parameters.

Use daylight saving time (DST) — Daylight saving time
on (default) | off

Select this parameter to enable daylight saving time.

Dependencies

To enable this parameter, select the **Enable geospatial configuration**, **Use advanced Sun sky**, and **Use daylight saving time (DST)** parameters.

DST start day — Start day of daylight saving time
10 (default) | integer scalar in the range [1, 31]

Specify the start day of daylight saving time, specified as a scalar from 1 to 31.

Dependencies

To enable this parameter, select the **Enable geospatial configuration**, **Use advanced Sun sky**, and **Use daylight saving time (DST)** parameters.

DST start month — Start month of daylight saving time
3 (default) | integer in the range [1, 12]

Specify the start month of daylight saving time, from 1 to 12.

Dependencies

To enable this parameter, select the **Enable geospatial configuration**, **Use advanced Sun sky**, and **Use daylight saving time (DST)** parameters.

DST end day — End day of daylight saving time
3 (default) | integer in range [1, 31]

Specify the end day of daylight saving time, from 1 to 31.

Dependencies

To enable this parameter, select the **Enable geospatial configuration**, **Use advanced Sun sky**, and **Use daylight saving time (DST)** parameters.

DST end month — End month of daylight saving time
11 (default) | integer in range [1, 12]

Specify the end month of daylight saving time, from 1 to 12.

Dependencies

To enable this parameter, select the **Enable geospatial configuration**, **Use advanced Sun sky**, and **Use daylight saving time (DST)** parameters.

DST switch hour — Hour when daylight saving time switches
2 (default) | integer in range [1, 24]

Hour when daylight saving time switches, from 1 to 24.

Dependencies

To enable this parameter, select the **Enable geospatial configuration**, **Use advanced Sun sky**, and **Use daylight saving time (DST)** parameters.

Authentication manager — Management of access tokens
button

Click to manage access tokens, such to create, update, and delete tokens.

Dependencies

To enable this parameter, select the **Enable geospatial configuration** parameter.

More About

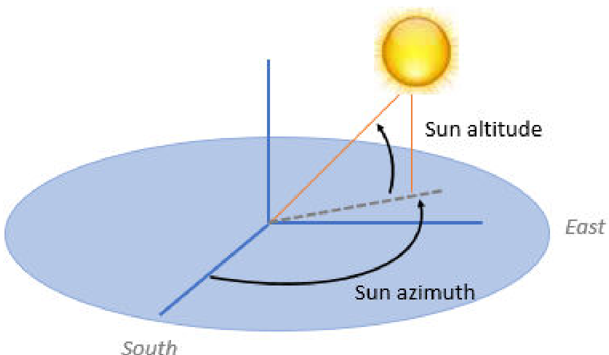
Sun Position and Weather

To control the scene weather and sun position, on the **Weather** tab, select **Override scene weather**. Use the enabled parameters to change the sun position, clouds, fog, and rain during the simulation.

Sun Position

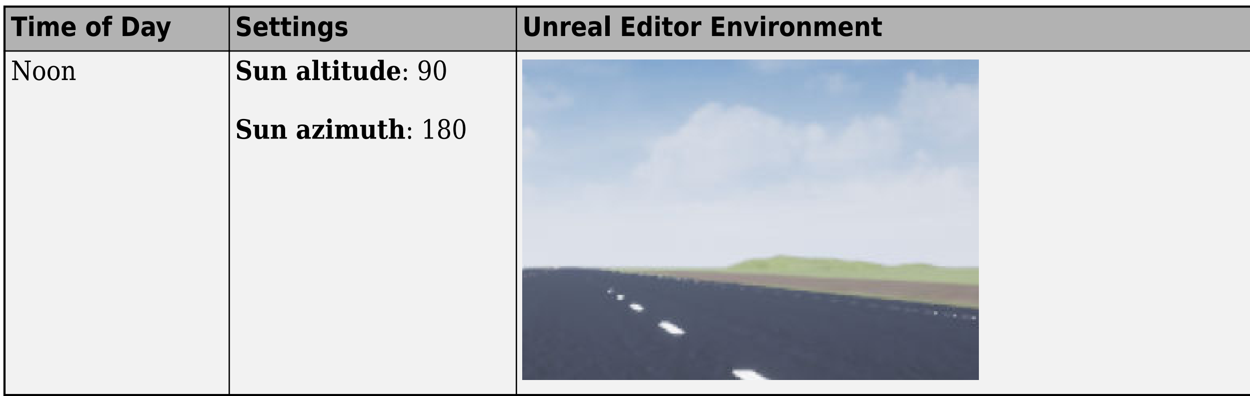
Use **Sun altitude** and **Sun azimuth** to control the sun position.

- **Sun altitude** — Altitude angle in a vertical plane between the sun rays and the horizontal projection of the rays.
- **Sun azimuth** — Azimuth angle in the horizontal plane measured from the south to the horizontal projection of the sun rays.



This table summarizes sun position settings for specific times of day.

Time of Day	Settings	Unreal Editor Environment
Midnight	**Sun altitude**: -90 **Sun azimuth**: 180	
Sunrise in the north	**Sun altitude**: 0 **Sun azimuth**: 180	

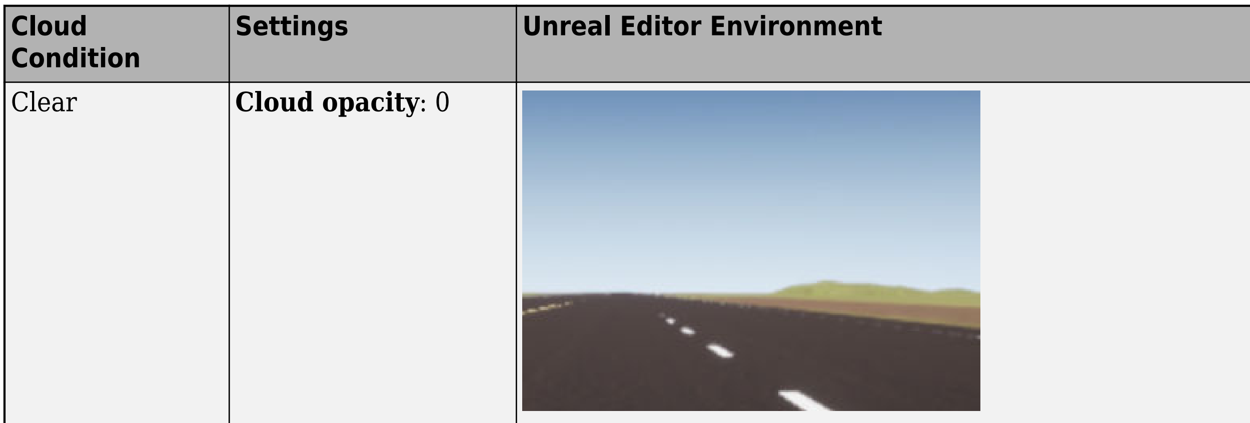
Time of Day	Settings	Unreal Editor Environment
Noon	**Sun altitude**: 90 **Sun azimuth**: 180	

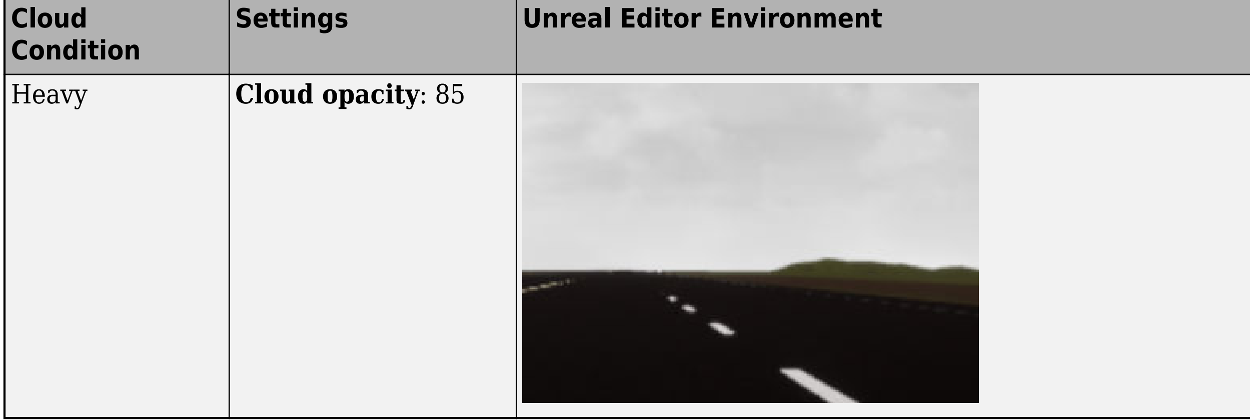
Clouds

Use **Cloud opacity** and **Cloud speed** to control clouds in the scene.

- **Cloud opacity** — Unreal Editor **Cloud Opacity** global actor target value. Zero is a cloudless scene.
- **Cloud speed** — Unreal Editor **Cloud Speed** global actor target value. The clouds move from west to east for positive values and east to west for negative values.

This table summarizes settings for specific cloud conditions.

Cloud Condition	Settings	Unreal Editor Environment
Clear	**Cloud opacity**: 0	



Cloud Condition	Settings	Unreal Editor Environment
Heavy	**Cloud opacity**: 85	

Fog

Use **Fog density** to control fog in the scene. **Fog density** corresponds to the Unreal Editor **Set Fog Density**.

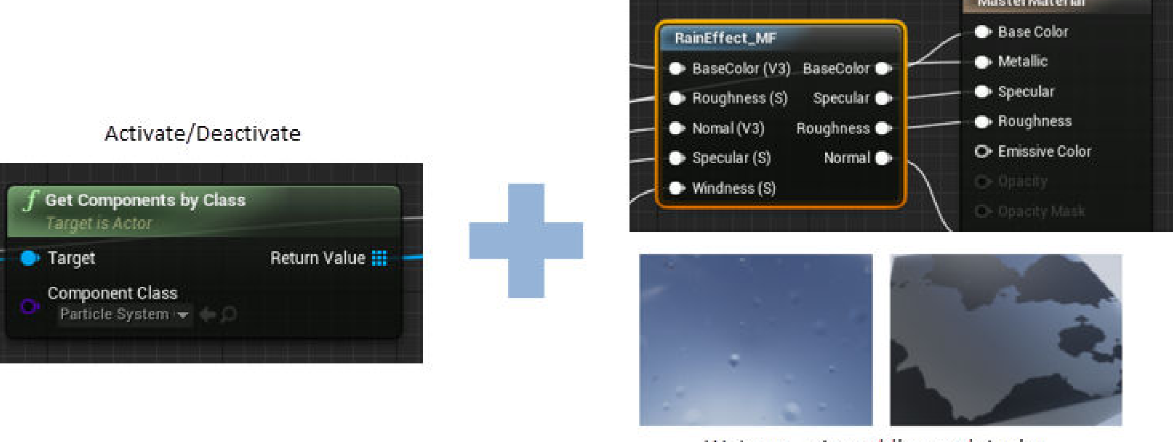
This table summarizes settings for specific fog conditions.

Fog Condition	Settings	Unreal Editor Environment
None	**Fog density**: 0	
Heavy	**Fog density**: 100	

Rain

Use **Cloud opacity** and **Rain density** to control rain in the scene.

- **Cloud opacity** — Unreal Editor **Cloud Opacity** global actor target value.
- **Rain density** — Unreal Editor local actor that controls rain density, wetness, rain puddles, and ripples.



This table summarizes settings for specific rain conditions.

Rain Condition	Settings	Unreal Editor Environment
Light	**Cloud opacity**: 10 **Rain density**: 25	
Heavy	**Cloud opacity**: 10 **Rain density**: 80	

Version History

Introduced in R2020b

R2023a: Cesium support for specifying geospatial coordinates

You can select the **Enable geospatial configuration** parameter to enable the parameters of the **Geospatial** tab, such as **Access token ID** and **Georeference parameters**. You can use the

parameters on the **Geospatial** tab to specify geospatial origin coordinates using a Cesium access token ID.

See Also

Simulation 3D Camera | Simulation 3D Lidar | Simulation 3D Fisheye Camera | Simulation 3D UAV Vehicle

Topics

"Unreal Engine Simulation for Unmanned Aerial Vehicles"
"How Unreal Engine Simulation for UAVs Works"
"Coordinate Systems for Unreal Engine Simulation in UAV Toolbox"

Simulation 3D Camera

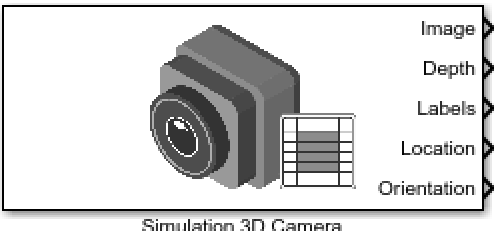
Camera sensor model with lens in 3D simulation environment

Description

The Simulation 3D Camera block provides an interface to a camera with a lens in a 3D simulation environment. This environment is rendered using the Unreal Engine from Epic Games. The sensor is based on the ideal pinhole camera model, with a lens added to represent a full camera model, including lens distortion. This camera model supports a field of view of up to 150 degrees. For more details, see “Algorithms” on page 4-92.

If you set **Sample time** to `-1`, the block uses the sample time specified in the Simulation 3D Scene Configuration block. To use this sensor, you must include a Simulation 3D Scene Configuration block in your model.

The block outputs images captured by the camera during simulation. You can use these images to visualize and verify your driving algorithms. In addition, on the **Ground Truth** tab, you can select options to output the ground truth data for developing depth estimation and semantic segmentation algorithms. You can also output the location and orientation of the camera in the world coordinate system of the scene. The image shows the block with all ports enabled.



The table summarizes the ports and how to enable them.

Port	Description	Parameter for Enabling Port	Sample Visualization
Image	Outputs an RGB image captured by the camera	n/a	

Port	Description	Parameter for Enabling Port	Sample Visualization
Depth	Outputs a depth map with values from 0 m to 1000 meters	**Output depth**	
Labels	Outputs a semantic segmentation map of label IDs that correspond to objects in the scene	**Output semantic segmentation**	
Location	Outputs the location of the camera in the world coordinate system	**Output location (m) and orientation (rad)**	n/a
Orientation	Outputs the orientation of the camera in the world coordinate system	**Output location (m) and orientation (rad)**	n/a

Note The Simulation 3D Scene Configuration block must execute before the Simulation 3D Camera block. That way, the Unreal Engine 3D visualization environment prepares the data before the Simulation 3D Camera block receives it. To check the block execution order, right-click the blocks and select **Properties**. On the **General** tab, confirm these **Priority** settings:

- Simulation 3D Scene Configuration — `0`
- Simulation 3D Camera — `1`

For more information about execution order, see “Block Execution Order”.

Ports

Input

Translation — Relative translation of sensor from mounting point (m)
`[0 0 0]` (default) | real-valued 1-by-3 vector of form [*X Y Z*]

Relative translation of the sensor from its mounting point on the vehicle, in meters, specified as a real-valued 1-by-3 vector of the form [*X Y Z*].

Dependencies

To enable this port, select the **Input** parameter next to the **Relative translation [X, Y, Z] (m)** parameter. When you select **Input**, the **Relative translation [X, Y, Z] (m)** parameter specifies the initial relative translation and the **Translation** port specifies the relative translation during simulation. For more details, see "Sensor Position Transformation" on page 4-107.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Rotation — Relative rotation of sensor from mounting point (deg)
`[0 0 0]` (default) | real-valued 1-by-3 vector of form [*Roll Pitch Yaw*]

Relative rotation of the sensor from its mounting point on the vehicle, in degrees, specified as a real-valued 1-by-3 vector of the form [*Roll Pitch Yaw*].

Dependencies

To enable this port, select the **Input** parameter next to the **Relative rotation [Roll, Pitch, Yaw] (deg)** parameter. When you select **Input**, the **Relative translation [Roll, Pitch, Yaw] (deg)** parameter specifies the initial relative rotation and the **Rotation** port specifies the relative rotation during simulation. For more details, see "Sensor Position Transformation" on page 4-107.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Output

Image — 3D output camera image
m-by-*n*-by-3 array of RGB triplet values

3D output camera image, returned as an *m*-by-*n*-by-3 array of RGB triplet values. *m* is the vertical resolution of the image, and *n* is the horizontal resolution of the image.

Data Types: `int8` | `uint8`

Depth — Object depth from 0 m to 1000 m
m-by-*n* array of object depths

Object depth for each pixel in the image, output as an *m*-by-*n* array. *m* is the vertical resolution of the image, and *n* is the horizontal resolution of the image. Depth is in the range from 0 to 1000 meters.

Dependencies

To enable this port, on the **Ground Truth** tab, select **Output depth**.

Data Types: `double`

Labels — Label identifiers
m-by-*n* array of label identifiers

Label identifier for each pixel in the image, output as an *m*-by-*n* array. *m* is the vertical resolution of the image, and *n* is the horizontal resolution of the image.

The table shows the object IDs used in the default scenes that are selectable from the Simulation 3D Scene Configuration block. If you are using a custom scene, in the Unreal Editor, you can assign new object types to unused IDs. If a scene contains an object that does not have an assigned ID, that object is assigned an ID of `0`. The detection of lane markings is not supported.

ID	Type
`0`	None/default
`1`	Building
`2`	*Not used*
`3`	Other
`4`	Pedestrians
`5`	Pole
`6`	Lane Markings
`7`	Road
`8`	Sidewalk
`9`	Vegetation
`10`	Vehicle
`11`	*Not used*
`12`	Generic traffic sign
`13`	Stop sign
`14`	Yield sign
`15`	Speed limit sign
`16`	Weight limit sign
`17-18`	*Not used*
`19`	Left and right arrow warning sign
`20`	Left chevron warning sign
`21`	Right chevron warning sign
`22`	*Not used*
`23`	Right one-way sign
`24`	*Not used*
`25`	School bus only sign
`26-38`	*Not used*
`39`	Crosswalk sign
`40`	*Not used*
`41`	Traffic signal
`42`	Curve right warning sign
`43`	Curve left warning sign

ID	Type
44	Up right arrow warning sign
45-47	*Not used*
48	Railroad crossing sign
49	Street sign
50	Roundabout warning sign
51	Fire hydrant
52	Exit sign
53	Bike lane sign
54-56	*Not used*
57	Sky
58	Curb
59	Flyover ramp
60	Road guard rail
61	Bicyclist
62-66	*Not used*
67	Deer
68-70	*Not used*
71	Barricade
72	Motorcycle
73-255	*Not used*

Dependencies

To enable this port, on the **Ground Truth** tab, select **Output semantic segmentation**.

Data Types: `uint8`

Location — Sensor location
real-valued 1-by-3 vector

Sensor location along the *X*-axis, *Y*-axis, and *Z*-axis of the scene. The **Location** values are in the world coordinates of the scene. In this coordinate system, the *Z*-axis points up from the ground. Units are in meters.

Dependencies

To enable this port, on the **Ground Truth** tab, select **Output location (m) and orientation (rad)**.

Data Types: `double`

Orientation — Sensor orientation
real-valued 1-by-3 vector

Roll, pitch, and yaw sensor orientation about the *X*-axis, *Y*-axis, and *Z*-axis of the scene. The **Orientation** values are in the world coordinates of the scene. These values are positive in the clockwise direction when looking in the positive directions of these axes. Units are in radians.

Dependencies

To enable this port, on the **Ground Truth** tab, select **Output location (m) and orientation (rad)**.

Data Types: `double`

Parameters

Mounting

Sensor identifier — Unique sensor identifier

`1` (default) | positive integer

Specify the unique identifier of the sensor. In a multisensor system, the sensor identifier enables you to distinguish between sensors. When you add a new sensor block to your model, the **Sensor identifier** of that block is $N + 1$, where N is the highest **Sensor identifier** value among the existing sensor blocks in the model.

Example: `2`

Parent name — Name of parent vehicle

`Scene Origin` (default) | vehicle name

Name of the parent to which the sensor is mounted, specified as `Scene Origin` or as the name of a vehicle in your model. The vehicle names that you can select correspond to the **Name** parameters of the simulation 3D vehicle blocks in your model. If you select `Scene Origin`, the block places a sensor at the scene origin.

Example: `SimulinkVehicle1`

Mounting location — Sensor mounting location

`Origin` (default)

Sensor mounting location.

- When **Parent name** is `Scene Origin`, the block mounts the sensor to the origin of the scene, and **Mounting location** can be set to `Origin` only. During simulation, the sensor remains stationary.
- When **Parent name** is the name of a vehicle (for example, `SimulinkVehicle1`) the block mounts the sensor to one of the predefined mounting locations described in the table. During simulation, the sensor travels with the vehicle.

Vehicle Mounting Location	Description	Orientation Relative to Vehicle Origin [Roll, Pitch, Yaw] (deg)
Origin	Forward-facing sensor mounted to the vehicle origin, which is on the ground, at the geometric center of the vehicle	[0, 0, 0]

Roll, pitch, and yaw are clockwise-positive when looking in the positive direction of the *X*-axis, *Y*-axis, and *Z*-axis, respectively. When looking at a vehicle from the top down, then the yaw angle (that is, the orientation angle) is counterclockwise-positive, because you are looking in the negative direction of the axis.

The (*X*, *Y*, *Z*) mounting location of the sensor relative to the vehicle depends on the vehicle type. To specify the vehicle type, use the **Type** parameter of the Simulation 3D UAV Vehicle block to which you are mounting. To obtain the (*X*, *Y*, *Z*) mounting locations for a vehicle type, see the reference page for that vehicle.

To determine the location of the sensor in world coordinates, open the sensor block. Then, on the **Ground Truth** tab, select **Output location (m) and orientation (rad)** and inspect the data from the **Location** output port.

Specify offset — Specify offset from mounting location

`off` (default) | `on`

Select this parameter to specify an offset from the mounting location by using the **Relative translation [X, Y, Z] (m)** and **Relative rotation [Roll, Pitch, Yaw] (deg)** parameters.

Relative translation [X, Y, Z] (m) — Translation offset relative to mounting location

`[0, 0, 0]` (default) | real-valued 1-by-3 vector

Translation offset relative to the mounting location of the sensor, specified as a real-valued 1-by-3 vector of the form [*X*, *Y*, *Z*]. Units are in meters.

If you mount the sensor to a vehicle by setting **Parent name** to the name of that vehicle, then *X*, *Y*, and *Z* are in the vehicle coordinate system, where:

- The *X*-axis points forward from the vehicle.
- The *Y*-axis points to the left of the vehicle, as viewed when looking in the forward direction of the vehicle.

- The *Z*-axis points up.

The origin is the mounting location specified in the **Mounting location** parameter. This origin is different from the vehicle origin, which is the geometric center of the vehicle.

If you mount the sensor to the scene origin by setting **Parent name** to `Scene Origin`, then *X*, *Y*, and *Z* are in the world coordinates of the scene.

For more details about the vehicle and world coordinate systems, see “Coordinate Systems for Unreal Engine Simulation in UAV Toolbox”.

Example: `[0,0,0.01]`

Adjust Relative Translation During Simulation

To adjust the relative translation of the sensor during simulation, enable the **Translation** input port by selecting the **Input** parameter next to the **Relative translation [X, Y, Z] (m)** parameter. When you enable the **Translation** port, the **Relative translation [X, Y, Z] (m)** parameter specifies the initial relative translation of the sensor and the **Translation** port specifies the relative translation of the sensor during simulation. For more details about the relative translation and rotation of this sensor, see “Sensor Position Transformation” on page 4-107.

Dependencies

To enable this parameter, select **Specify offset**.

Relative rotation [Roll, Pitch, Yaw] (deg) — Rotational offset relative to mounting location

`[0, 0, 0]` (default) | real-valued 1-by-3 vector

Rotational offset relative to the mounting location of the sensor, specified as a real-valued 1-by-3 vector of the form [*Roll*, *Pitch*, *Yaw*] . Roll, pitch, and yaw are the angles of rotation about the *X*-, *Y*-, and *Z*-axes, respectively. Units are in degrees.

If you mount the sensor to a vehicle by setting **Parent name** to the name of that vehicle, then *X*, *Y*, and *Z* are in the vehicle coordinate system, where:

- The *X*-axis points forward from the vehicle.
- The *Y*-axis points to the left of the vehicle, as viewed when looking in the forward direction of the vehicle.
- The *Z*-axis points up.
- Roll, pitch, and yaw are clockwise-positive when looking in the forward direction of the *X*-axis, *Y*-axis, and *Z*-axis, respectively. If you view a scene from a 2D top-down perspective, then the yaw angle (also called the orientation angle) is counterclockwise-positive because you are viewing the scene in the negative direction of the *Z*-axis.

The origin is the mounting location specified in the **Mounting location** parameter. This origin is different from the vehicle origin, which is the geometric center of the vehicle.

If you mount the sensor to the scene origin by setting **Parent name** to `Scene Origin`, then *X*, *Y*, and *Z* are in the world coordinates of the scene.

For more details about the vehicle and world coordinate systems, see “Coordinate Systems for Unreal Engine Simulation in UAV Toolbox”.

Example: `[0,0,10]`

Adjust Relative Rotation During Simulation

To adjust the relative rotation of the sensor during simulation, enable the **Rotation** input port by selecting the **Input** parameter next to the **Relative rotation [Roll, Pitch, Yaw] (deg)** parameter. When you enable the **Rotation** port, the **Relative rotation [Roll, Pitch, Yaw] (deg)** parameter specifies the initial relative rotation of the sensor and the **Rotation** port specifies the relative rotation of the sensor during simulation. For more details about the relative translation and rotation of this sensor, see "Sensor Position Transformation" on page 4-107.

Dependencies

To enable this parameter, select **Specify offset**.

Sample time — Sample time

`-1` (default) | positive scalar

Sample time of the block, in seconds, specified as a positive scalar. The 3D simulation environment frame rate is the inverse of the sample time.

If you set the sample time to `-1`, the block inherits its sample time from the Simulation 3D Scene Configuration block.

Parameters

These intrinsic camera parameters are equivalent to the properties of a `cameraIntrinsics` object. To obtain the intrinsic parameters for your camera, use the **Camera Calibrator** app.

For details about the camera calibration process, see "Using the Single Camera Calibrator App" (Computer Vision Toolbox) and "What Is Camera Calibration?" (Computer Vision Toolbox).

Focal length (pixels) — Focal length of camera

`[1109, 1109]` (default) | 1-by-2 positive integer vector

Focal length of the camera, specified as a 1-by-2 positive integer vector of the form [fx, fy]. Units are in pixels.

$$fx = F \times sx$$
$$fy = F \times sy$$

where:

- F is the focal length in world units, typically millimeters.
- [sx, sy] are the number of pixels per world unit in the x and y direction, respectively.

This parameter is equivalent to the `FocalLength` property of a `cameraIntrinsics` object.

Optical center (pixels) — Optical center of camera

`[640, 360]` (default) | 1-by-2 positive integer vector

Optical center of the camera, specified as a 1-by-2 positive integer vector of the form [cx,cy]. Units are in pixels.

This parameter is equivalent to the `PrincipalPoint` property of a `cameraIntrinsics` object.

Image size (pixels) — Image size produced by camera

`[720, 1280]` (default) | 1-by-2 positive integer vector

Image size produced by the camera, specified as a 1-by-2 positive integer vector of the form [*mrows*,*ncols*]. Units are in pixels.

This parameter is equivalent to the `ImageSize` property of a `cameraIntrinsics` object.

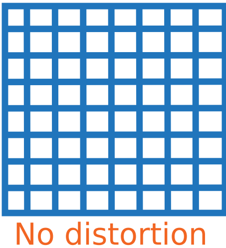
Radial distortion coefficients — Radial distortion coefficients

`[0, 0]` (default) | real-valued 2-element vector | real-valued 3-element vector | real-valued 6-element vector

Specify the radial distortion coefficients as a real-valued 2-element, 3-element, or 6-element vector. Radial distortion is the displacement of image points along radial lines extending from the principal point.

- As image points move away from the principal point (positive radial displacement), image magnification decreases and a pincushion-shaped distortion occurs on the image.
- As image points move toward the principal point (negative radial displacement), image magnification increases and a barrel-shaped distortion occurs on the image.

Pincushion distortion
Positive radial displacement


No distortion

Barrel distortion
Negative radial displacement

The camera sensor calculates the (x_d,y_d) radial-distorted location of a point using a two-coefficient, three-coefficient, or six-coefficient formula. This table shows the various formulas, where:

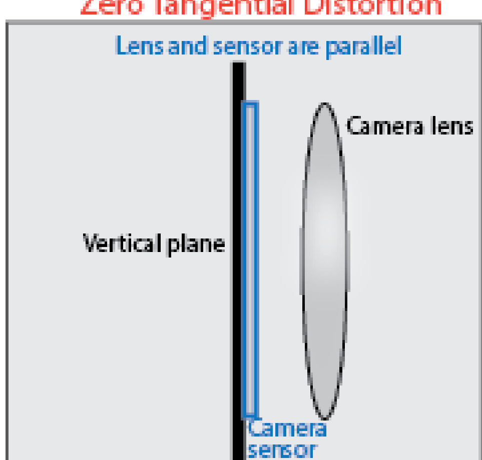
- (x,y) = undistorted pixel locations
- k_1,k_2,k_3,k_4,k_5,k_6 = radial distortion coefficients of the lens
- $r^2 = x^2 + y^2$

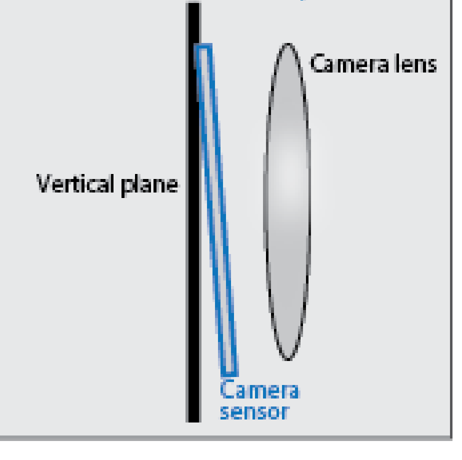
Coefficients	Formula	Description
`[k1, k2]`	$x_d = x(1 + k_1 r^2 + k_2 r^4)$ $y_d = y(1 + k_1 r^2 + k_2 r^4)$	This model is equivalent to the two-coefficient model used by the `RadialDistortion` property of a `cameraIntrinsics` object.
`[k1, k2, k3]`	$x_d = x(1 + k_1 r^2 + k_2 r^4 + k_3 r^6)$ $y_d = y(1 + k_1 r^2 + k_2 r^4 + k_3 r^6)$	This model is equivalent to the three-coefficient model used by the `RadialDistortion` property of a `cameraIntrinsics` object.
`[k1, k2, k3, k4, k5, k6]`	$x_d = x \times \frac{1 + k_1 r^2 + k_2 r^4 + k_3 r^6}{1 + k_4 r^2 + k_5 r^4 + k_6 r^6}$ $y_d = y \times \frac{1 + k_1 r^2 + k_2 r^4 + k_3 r^6}{1 + k_4 r^2 + k_5 r^4 + k_6 r^6}$	The six-coefficient model is based on the OpenCV radial distortion model. **Note** The **Camera Calibrator** app does not support this model. To calibrate a camera using this model, see Camera Calibration and 3D Reconstruction in the OpenCV documentation.

Tangential distortion coefficients — Tangential distortion coefficients

`[0, 0]` (default) | real-valued 2-element vector

Specify the tangential distortion coefficients as a real-valued 2-element vector. Tangential distortion occurs when the lens and the image plane are not parallel.





The camera sensor calculates the tangential distorted location of a point, (x_d, y_d), using this formula:

$$x_d = x + [2p_1xy + p_2 \times (r^2 + 2x^2)]$$
$$y_d = y + [p_1 \times (r^2 + 2y^2) + 2p_2xy]$$

where:

- x, y = undistorted pixel locations
- p_1, p_2 = tangential distortion coefficients of the lens
- $r^2 = x^2 + y^2$

The undistorted pixel locations appear in normalized image coordinates, with the origin at the optical center. The coordinates are expressed in world units.

This parameter is equivalent to the `TangentialDistortion` property of a `cameraIntrinsics` object.

Axis skew — Skew angle of camera axes

`0` (default) | nonnegative scalar

Skew angle of the camera axes, specified as a nonnegative scalar. If the *X*-axis and *Y*-axis are exactly perpendicular, then the skew must be `0`. Units are dimensionless.

This parameter is equivalent to the `Skew` property of a `cameraIntrinsics` object.

Ground Truth

Output depth — Output depth map

`off` (default) | `on`

Select this parameter to output a depth map at the **Depth** port.

Output semantic segmentation — Output semantic segmentation map of label IDs

`off` (default) | `on`

Select this parameter to output a semantic segmentation map of label IDs at the **Labels** port.

Output location (m) and orientation (rad) — Output location and orientation of sensor

`off` (default) | `on`

Select this parameter to output the location and orientation of the sensor at the **Location** and **Orientation** ports, respectively.

Tips

- To visualize the camera images that are output by the **Image** port, use a Video Viewer or To Video Display block.

  To learn how to visualize the depth and semantic segmentation maps that are output by the **Depth** and **Labels** ports, see the "Depth and Semantic Segmentation Visualization Using Unreal Engine Simulation" example.
- Because the Unreal Engine can take a long time to start between simulations, consider logging the signals that the sensors output. You can then use this data to develop perception algorithms in MATLAB. See "Mark Signals for Logging" (Simulink).

Algorithms

The block uses the camera model proposed by Jean-Yves Bouguet [1]. The model includes:

- The pinhole camera model [2]
- Lens distortion [3]

The pinhole camera model does not account for lens distortion because an ideal pinhole camera does not have a lens. To accurately represent a real camera, the full camera model used by the block includes radial and tangential lens distortion.

For more details, see "What Is Camera Calibration?" (Computer Vision Toolbox)

Version History

Introduced in R2020b

References

[1] Bouguet, J. Y. *Camera Calibration Toolbox for Matlab*. http://www.vision.caltech.edu/bouguetj/calib_doc

[2] Zhang, Z. "A Flexible New Technique for Camera Calibration." *IEEE Transactions on Pattern Analysis and Machine Intelligence*. Vol. 22, No. 11, 2000, pp. 1330–1334.

[3] Heikkila, J., and O. Silven. "A Four-step Camera Calibration Procedure with Implicit Image Correction." *IEEE International Conference on Computer Vision and Pattern Recognition*. 1997.

See Also

Blocks
Simulation 3D Lidar | Simulation 3D Fisheye Camera | Simulation 3D UAV Vehicle | Simulation 3D Scene Configuration

Apps
Camera Calibrator

Objects
`cameraIntrinsics`

Topics
"Coordinate Systems for Unreal Engine Simulation in UAV Toolbox"
"Choose a Sensor for Unreal Engine Simulation"
"Apply Semantic Segmentation Labels to Custom Scenes"
"What Is Camera Calibration?" (Computer Vision Toolbox)
"Depth Estimation From Stereo Video" (Computer Vision Toolbox)
"Semantic Segmentation Using Deep Learning" (Computer Vision Toolbox)

Simulation 3D Lidar

Lidar sensor model in 3D simulation environment

Description

The Simulation 3D Lidar block provides an interface to the lidar sensor in a 3D simulation environment. This environment is rendered using the Unreal Engine from Epic Games. The block returns a point cloud with the specified field of view and angular resolution. You can also output the distances from the sensor to object points and the reflectivity of surface materials. In addition, you can output the location and orientation of the sensor in the world coordinate system of the scene.

If you set **Sample time** to `-1`, the block uses the sample time specified in the Simulation 3D Scene Configuration block. To use this sensor, ensure that the Simulation 3D Scene Configuration block is in your model.

Note The Simulation 3D Scene Configuration block must execute before the Simulation 3D Lidar block. That way, the Unreal Engine 3D visualization environment prepares the data before the Simulation 3D Lidar block receives it. To check the block execution order, right-click the blocks and select **Properties**. On the **General** tab, confirm these **Priority** settings:

- Simulation 3D Scene Configuration — `0`
- Simulation 3D Lidar — `1`

For more information about execution order, see “Block Execution Order”.

Ports

Output

Point cloud — Point cloud data
m-by-*n*-by-3 array of positive real-valued [*x*, *y*, *z*] points

Point cloud data, returned as an *m*-by-*n*-by 3 array of positive, real-valued [*x*, *y*, *z*] points. *m* and *n* define the number of points in the point cloud, as shown in this equation:

$$m \times n = \frac{V_{FOV}}{V_{RES}} \times \frac{H_{FOV}}{H_{RES}}$$

where:

- V_{FOV} is the vertical field of view of the lidar, in degrees, as specified by the **Vertical field of view (deg)** parameter.
- V_{RES} is the vertical angular resolution of the lidar, in degrees, as specified by the **Vertical resolution (deg)** parameter.
- H_{FOV} is the horizontal field of view of the lidar, in degrees, as specified by the **Horizontal field of view (deg)** parameter.
- H_{RES} is the horizontal angular resolution of the lidar, in degrees, as specified by the **Horizontal resolution (deg)** parameter.

Each *m*-by-*n* entry in the array specifies the *x*, *y*, and *z* coordinates of a detected point in the sensor coordinate system. If the lidar does not detect a point at a given coordinate, then *x*, *y*, and *z* are returned as `NaN`.

Data Types: `single`

Distance — Distance to object points
m-by-*n* positive real-valued matrix

Distance to object points measured by the lidar sensor, returned as an *m*-by-*n* positive real-valued matrix. Each *m*-by-*n* value in the matrix corresponds to an [*x*, *y*, *z*] coordinate point returned by the **Point cloud** output port.

Dependencies

To enable this port, on the **Parameters** tab, select **Distance outport**.

Data Types: `single`

Reflectivity — Reflectivity of surface materials
m-by-*n* matrix of intensity values in range [0, 1]

Reflectivity of surface materials, returned as an *m*-by-*n* matrix of intensity values in the range [0, 1], where *m* is the number of rows in the point cloud and *n* is the number of columns. Each point in the **Reflectivity** output corresponds to a point in the **Point cloud** output. The block returns points that are not part of a surface material as `NaN`.

To calculate reflectivity, the lidar sensor uses the Phong reflection model. This model describes surface reflectivity as a combination of diffuse reflections (scattered reflections, such as from rough surfaces) and specular reflections (mirror-like reflections, such as from smooth surfaces). For more details on this model, see the Phong reflection model page on Wikipedia.

Dependencies

To enable this port, select the **Reflectivity outport** parameter.

Data Types: `single`

Location — Sensor location
real-valued 1-by-3 vector

Sensor location along the *X*-axis, *Y*-axis, and *Z*-axis of the scene. The **Location** values are in the world coordinates of the scene. In this coordinate system, the *Z*-axis points up from the ground. Units are in meters.

Dependencies

To enable this port, on the **Ground Truth** tab, select **Output location (m) and orientation (rad)**.

Data Types: `double`

Orientation — Sensor orientation
real-valued 1-by-3 vector

Roll, pitch, and yaw sensor orientation about the *X*-axis, *Y*-axis, and *Z*-axis of the scene. The **Orientation** values are in the world coordinates of the scene. These values are positive in the clockwise direction when looking in the positive directions of these axes. Units are in radians.

Dependencies

To enable this port, on the **Ground Truth** tab, select **Output location (m) and orientation (rad)**.

Data Types: `double`

Parameters

Mounting

Sensor identifier — Unique sensor identifier

`1` (default) | positive integer

Specify the unique identifier of the sensor. In a multisensor system, the sensor identifier enables you to distinguish between sensors. When you add a new sensor block to your model, the **Sensor identifier** of that block is $N + 1$, where N is the highest **Sensor identifier** value among the existing sensor blocks in the model.

Example: `2`

Parent name — Name of parent vehicle

`Scene Origin` (default) | vehicle name

Name of the parent to which the sensor is mounted, specified as `Scene Origin` or as the name of a vehicle in your model. The vehicle names that you can select correspond to the **Name** parameters of the simulation 3D vehicle blocks in your model. If you select `Scene Origin`, the block places a sensor at the scene origin.

Example: `SimulinkVehicle1`

Mounting location — Sensor mounting location

`Origin` (default)

Sensor mounting location.

- When **Parent name** is `Scene Origin`, the block mounts the sensor to the origin of the scene, and **Mounting location** can be set to `Origin` only. During simulation, the sensor remains stationary.
- When **Parent name** is the name of a vehicle (for example, `SimulinkVehicle1`) the block mounts the sensor to one of the predefined mounting locations described in the table. During simulation, the sensor travels with the vehicle.

Vehicle Mounting Location	Description	Orientation Relative to Vehicle Origin [Roll, Pitch, Yaw] (deg)
`Origin`	Forward-facing sensor mounted to the vehicle origin, which is on the ground, at the geometric center of the vehicle	[0, 0, 0]

Roll, pitch, and yaw are clockwise-positive when looking in the positive direction of the *X*-axis, *Y*-axis, and *Z*-axis, respectively. When looking at a vehicle from the top down, then the yaw angle (that is, the orientation angle) is counterclockwise-positive, because you are looking in the negative direction of the axis.

The (*X*, *Y*, *Z*) mounting location of the sensor relative to the vehicle depends on the vehicle type. To specify the vehicle type, use the **Type** parameter of the Simulation 3D UAV Vehicle block to which you are mounting. To obtain the (*X*, *Y*, *Z*) mounting locations for a vehicle type, see the reference page for that vehicle.

To determine the location of the sensor in world coordinates, open the sensor block. Then, on the **Ground Truth** tab, select **Output location (m) and orientation (rad)** and inspect the data from the **Location** output port.

Specify offset — Specify offset from mounting location

`off` (default) | `on`

Select this parameter to specify an offset from the mounting location by using the **Relative translation [X, Y, Z] (m)** and **Relative rotation [Roll, Pitch, Yaw] (deg)** parameters.

Relative translation [X, Y, Z] (m) — Translation offset relative to mounting location

`[0, 0, 0]` (default) | real-valued 1-by-3 vector

Translation offset relative to the mounting location of the sensor, specified as a real-valued 1-by-3 vector of the form [*X*, *Y*, *Z*]. Units are in meters.

If you mount the sensor to a vehicle by setting **Parent name** to the name of that vehicle, then *X*, *Y*, and *Z* are in the vehicle coordinate system, where:

- The *X*-axis points forward from the vehicle.
- The *Y*-axis points to the left of the vehicle, as viewed when looking in the forward direction of the vehicle.

- The *Z*-axis points up.

The origin is the mounting location specified in the **Mounting location** parameter. This origin is different from the vehicle origin, which is the geometric center of the vehicle.

If you mount the sensor to the scene origin by setting **Parent name** to `Scene Origin`, then *X*, *Y*, and *Z* are in the world coordinates of the scene.

For more details about the vehicle and world coordinate systems, see “Coordinate Systems for Unreal Engine Simulation in UAV Toolbox”.

Example: `[0,0,0.01]`

Dependencies

To enable this parameter, select **Specify offset**.

Relative rotation [Roll, Pitch, Yaw] (deg) — Rotational offset relative to mounting location

`[0, 0, 0]` (default) | real-valued 1-by-3 vector

Rotational offset relative to the mounting location of the sensor, specified as a real-valued 1-by-3 vector of the form [*Roll*, *Pitch*, *Yaw*] . Roll, pitch, and yaw are the angles of rotation about the *X*-, *Y*-, and *Z*-axes, respectively. Units are in degrees.

If you mount the sensor to a vehicle by setting **Parent name** to the name of that vehicle, then *X*, *Y*, and *Z* are in the vehicle coordinate system, where:

- The *X*-axis points forward from the vehicle.
- The *Y*-axis points to the left of the vehicle, as viewed when looking in the forward direction of the vehicle.
- The *Z*-axis points up.
- Roll, pitch, and yaw are clockwise-positive when looking in the forward direction of the *X*-axis, *Y*-axis, and *Z*-axis, respectively. If you view a scene from a 2D top-down perspective, then the yaw angle (also called the orientation angle) is counterclockwise-positive because you are viewing the scene in the negative direction of the *Z*-axis.

The origin is the mounting location specified in the **Mounting location** parameter. This origin is different from the vehicle origin, which is the geometric center of the vehicle.

If you mount the sensor to the scene origin by setting **Parent name** to `Scene Origin`, then *X*, *Y*, and *Z* are in the world coordinates of the scene.

For more details about the vehicle and world coordinate systems, see “Coordinate Systems for Unreal Engine Simulation in UAV Toolbox”.

Example: `[0,0,10]`

Dependencies

To enable this parameter, select **Specify offset**.

Sample time — Sample time

`-1` (default) | positive scalar

Sample time of the block, in seconds, specified as a positive scalar. The 3D simulation environment frame rate is the inverse of the sample time.

If you set the sample time to `-1`, the block inherits its sample time from the Simulation 3D Scene Configuration block.

Parameters

Detection range (m) — Maximum distance measured by lidar sensor

`120` (default) | positive scalar

Maximum distance measured by the lidar sensor, specified as a positive scalar. Points outside this range are ignored. Units are in meters.

Range resolution (m) — Resolution of lidar sensor range

`0.002` (default) | positive real scalar

Resolution of the lidar sensor range, in meters, specified as a positive real scalar. The range resolution is also known as the quantization factor. The minimal value of this factor is $D_{\text{range}} / 2^{24}$, where D_{range} is the maximum distance measured by the lidar sensor, as specified in the **Detection range (m)** parameter.

Vertical field of view (deg) — Vertical field of view

`40` (default) | positive scalar

Vertical field of view of the lidar sensor, specified as a positive scalar. Units are in degrees.

Vertical resolution (deg) — Vertical angular resolution

`1.25` (default) | positive scalar

Vertical angular resolution of the lidar sensor, specified as a positive scalar. Units are in degrees.

Horizontal field of view (deg) — Horizontal field of view

`360` (default) | positive scalar

Horizontal field of view of the lidar sensor, specified as a positive scalar. Units are in degrees.

Horizontal resolution (deg) — Horizontal angular (azimuth) resolution

`0.16` (default) | positive scalar

Horizontal angular (azimuth) resolution of the lidar sensor, specified as a positive scalar. Units are in degrees.

Distance outport — Output distance to measured object points

`off` (default) | `on`

Select this parameter to output the distance to measured object points at the **Distance** port.

Reflectivity outport — Output reflectivity of surface materials

`off` (default) | `on`

Select this parameter to output the reflectivity of surface materials at the **Reflectivity** port.

Ground Truth

Output location (m) and orientation (rad) — Output location and orientation of sensor

`off` (default) | `on`

Select this parameter to output the location and orientation of the sensor at the **Location** and **Orientation** ports, respectively.

Tips

- To visualize point clouds that are output by the **Point cloud** port, you can use a `pcplayer` object in a MATLAB Function block.
- The Unreal Engine can take a long time to start up between simulations, consider logging the signals that the sensors output. You can then use this data to develop perception algorithms in MATLAB. See “Mark Signals for Logging” (Simulink).

Version History

Introduced in R2020b

See Also

Blocks
Simulation 3D Camera | Simulation 3D Fisheye Camera | Simulation 3D Scene Configuration | Simulation 3D UAV Vehicle

Objects
`pointCloud` | `pcplayer`

Topics
“Coordinate Systems for Unreal Engine Simulation in UAV Toolbox”
“Choose a Sensor for Unreal Engine Simulation”

Simulation 3D Fisheye Camera

Fisheye camera sensor model in 3D simulation environment

Description

The Simulation 3D Fisheye Camera block provides an interface to a camera with a fisheye lens in a 3D simulation environment. This environment is rendered using the Unreal Engine from Epic Games. The sensor is based on the fisheye camera model proposed by Scaramuzza [1] on page 4-107. This model supports a field of view of up to 195 degrees. The block outputs an image with the specified camera distortion and size. You can also output the location and orientation of the camera in the world coordinate system of the scene.

If you set **Sample time** to `-1`, the block uses the sample time specified in the Simulation 3D Scene Configuration block. To use this sensor, you must include a Simulation 3D Scene Configuration block in your model.

Note The Simulation 3D Scene Configuration block must execute before the Simulation 3D Fisheye Camera block. That way, the Unreal Engine 3D visualization environment prepares the data before the Simulation 3D Fisheye Camera block receives it. To check the block execution order, right-click the blocks and select **Properties**. On the **General** tab, confirm these **Priority** settings:

- Simulation 3D Scene Configuration — `0`
- Simulation 3D Fisheye Camera — `1`

For more information about execution order, see “How Unreal Engine Simulation for UAVs Works”.

The Simulation 3D Fisheye Camera block requires Computer Vision Toolbox and Image Processing Toolbox™.

Ports

Input

Translation — Relative translation of sensor from mounting point (m)
`[0 0 0]` (default) | real-valued 1-by-3 vector of form [*X Y Z*]

Relative translation of the sensor from its mounting point on the vehicle, in meters, specified as a real-valued 1-by-3 vector of the form [*X Y Z*].

Dependencies

To enable this port, select the **Input** parameter next to the **Relative translation [X, Y, Z] (m)** parameter. When you select **Input**, the **Relative translation [X, Y, Z] (m)** parameter specifies the initial relative translation and the **Translation** port specifies the relative translation during simulation. For more details, see “Sensor Position Transformation” on page 4-107.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Rotation — Relative rotation of sensor from mounting point (deg)
`[0 0 0]` (default) | real-valued 1-by-3 vector of form [*Roll Pitch Yaw*]

Relative rotation of the sensor from its mounting point on the vehicle, in degrees, specified as a real-valued 1-by-3 vector of the form [*Roll Pitch Yaw*].

Dependencies

To enable this port, select the **Input** parameter next to the **Relative rotation [Roll, Pitch, Yaw] (deg)** parameter. When you select **Input**, the **Relative translation [Roll, Pitch, Yaw] (deg)** parameter specifies the initial relative rotation and the **Rotation** port specifies the relative rotation during simulation. For more details, see “Sensor Position Transformation” on page 4-107.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Output

Image — 3D output camera image
m-by-*n*-by-3 array of RGB triplet values

3D output camera image, returned as an *m*-by-*n*-by-3 array of RGB triplet values. *m* is the vertical resolution of the image, and *n* is the horizontal resolution of the image.

Data Types: `int8` | `uint8`

Location — Sensor location
real-valued 1-by-3 vector

Sensor location along the *X*-axis, *Y*-axis, and *Z*-axis of the scene. The **Location** values are in the world coordinates of the scene. In this coordinate system, the *Z*-axis points up from the ground. Units are in meters.

Dependencies

To enable this port, on the **Ground Truth** tab, select **Output location (m) and orientation (rad)**.

Data Types: `double`

Orientation — Sensor orientation
real-valued 1-by-3 vector

Roll, pitch, and yaw sensor orientation about the *X*-axis, *Y*-axis, and *Z*-axis of the scene. The **Orientation** values are in the world coordinates of the scene. These values are positive in the clockwise direction when looking in the positive directions of these axes. Units are in radians.

Dependencies

To enable this port, on the **Ground Truth** tab, select **Output location (m) and orientation (rad)**.

Data Types: `double`

Parameters

Mounting

Sensor identifier — Unique sensor identifier

`1` (default) | positive integer

Specify the unique identifier of the sensor. In a multisensor system, the sensor identifier enables you to distinguish between sensors. When you add a new sensor block to your model, the **Sensor**

identifier of that block is $N + 1$, where N is the highest **Sensor identifier** value among the existing sensor blocks in the model.

Example: 2

Parent name — Name of parent vehicle

`Scene Origin` (default) | vehicle name

Name of the parent to which the sensor is mounted, specified as `Scene Origin` or as the name of a vehicle in your model. The vehicle names that you can select correspond to the **Name** parameters of the simulation 3D vehicle blocks in your model. If you select `Scene Origin`, the block places a sensor at the scene origin.

Example: `SimulinkVehicle1`

Mounting location — Sensor mounting location

`Origin` (default)

Sensor mounting location.

- When **Parent name** is `Scene Origin`, the block mounts the sensor to the origin of the scene, and **Mounting location** can be set to `Origin` only. During simulation, the sensor remains stationary.
- When **Parent name** is the name of a vehicle (for example, `SimulinkVehicle1`) the block mounts the sensor to one of the predefined mounting locations described in the table. During simulation, the sensor travels with the vehicle.

Vehicle Mounting Location	Description	Orientation Relative to Vehicle Origin [Roll, Pitch, Yaw] (deg)
`Origin`	Forward-facing sensor mounted to the vehicle origin, which is on the ground, at the geometric center of the vehicle	[0, 0, 0]

Roll, pitch, and yaw are clockwise-positive when looking in the positive direction of the X-axis, Y-axis, and Z-axis, respectively. When looking at a vehicle from the top down, then the yaw angle (that is, the orientation angle) is counterclockwise-positive, because you are looking in the negative direction of the axis.

The (*X*, *Y*, *Z*) mounting location of the sensor relative to the vehicle depends on the vehicle type. To specify the vehicle type, use the **Type** parameter of the Simulation 3D UAV Vehicle block to which you are mounting. To obtain the (*X*, *Y*, *Z*) mounting locations for a vehicle type, see the reference page for that vehicle.

To determine the location of the sensor in world coordinates, open the sensor block. Then, on the **Ground Truth** tab, select **Output location (m) and orientation (rad)** and inspect the data from the **Location** output port.

Specify offset — Specify offset from mounting location

`off` (default) | `on`

Select this parameter to specify an offset from the mounting location by using the **Relative translation [X, Y, Z] (m)** and **Relative rotation [Roll, Pitch, Yaw] (deg)** parameters.

Relative translation [X, Y, Z] (m) — Translation offset relative to mounting location

`[0, 0, 0]` (default) | real-valued 1-by-3 vector

Translation offset relative to the mounting location of the sensor, specified as a real-valued 1-by-3 vector of the form [*X*, *Y*, *Z*]. Units are in meters.

If you mount the sensor to a vehicle by setting **Parent name** to the name of that vehicle, then *X*, *Y*, and *Z* are in the vehicle coordinate system, where:

- The *X*-axis points forward from the vehicle.
- The *Y*-axis points to the left of the vehicle, as viewed when looking in the forward direction of the vehicle.
- The *Z*-axis points up.

The origin is the mounting location specified in the **Mounting location** parameter. This origin is different from the vehicle origin, which is the geometric center of the vehicle.

If you mount the sensor to the scene origin by setting **Parent name** to `Scene Origin`, then *X*, *Y*, and *Z* are in the world coordinates of the scene.

For more details about the vehicle and world coordinate systems, see “Coordinate Systems for Unreal Engine Simulation in UAV Toolbox”.

Example: `[0,0,0.01]`

Adjust Relative Translation During Simulation

To adjust the relative translation of the sensor during simulation, enable the **Translation** input port by selecting the **Input** parameter next to the **Relative translation [X, Y, Z] (m)** parameter. When you enable the **Translation** port, the **Relative translation [X, Y, Z] (m)** parameter specifies the initial relative translation of the sensor and the **Translation** port specifies the relative translation of the sensor during simulation. For more details about the relative translation and rotation of this sensor, see “Sensor Position Transformation” on page 4-107.

Dependencies

To enable this parameter, select **Specify offset**.

Relative rotation [Roll, Pitch, Yaw] (deg) — Rotational offset relative to mounting location

`[0, 0, 0]` (default) | real-valued 1-by-3 vector

Rotational offset relative to the mounting location of the sensor, specified as a real-valued 1-by-3 vector of the form [*Roll*, *Pitch*, *Yaw*] . Roll, pitch, and yaw are the angles of rotation about the *X*-, *Y*-, and *Z*-axes, respectively. Units are in degrees.

If you mount the sensor to a vehicle by setting **Parent name** to the name of that vehicle, then *X*, *Y*, and *Z* are in the vehicle coordinate system, where:

- The *X*-axis points forward from the vehicle.
- The *Y*-axis points to the left of the vehicle, as viewed when looking in the forward direction of the vehicle.
- The *Z*-axis points up.
- Roll, pitch, and yaw are clockwise-positive when looking in the forward direction of the *X*-axis, *Y*-axis, and *Z*-axis, respectively. If you view a scene from a 2D top-down perspective, then the yaw angle (also called the orientation angle) is counterclockwise-positive because you are viewing the scene in the negative direction of the *Z*-axis.

The origin is the mounting location specified in the **Mounting location** parameter. This origin is different from the vehicle origin, which is the geometric center of the vehicle.

If you mount the sensor to the scene origin by setting **Parent name** to `Scene Origin`, then *X*, *Y*, and *Z* are in the world coordinates of the scene.

For more details about the vehicle and world coordinate systems, see “Coordinate Systems for Unreal Engine Simulation in UAV Toolbox”.

Example: `[0,0,10]`

Adjust Relative Rotation During Simulation

To adjust the relative rotation of the sensor during simulation, enable the **Rotation** input port by selecting the **Input** parameter next to the **Relative rotation [Roll, Pitch, Yaw] (deg)** parameter. When you enable the **Rotation** port, the **Relative rotation [Roll, Pitch, Yaw] (deg)** parameter specifies the initial relative rotation of the sensor and the **Rotation** port specifies the relative rotation of the sensor during simulation. For more details about the relative translation and rotation of this sensor, see “Sensor Position Transformation” on page 4-107.

Dependencies

To enable this parameter, select **Specify offset**.

Sample time — Sample time

`-1` (default) | positive scalar

Sample time of the block, in seconds, specified as a positive scalar. The 3D simulation environment frame rate is the inverse of the sample time.

If you set the sample time to `-1`, the block inherits its sample time from the Simulation 3D Scene Configuration block.

Parameters

These intrinsic camera parameters are equivalent to the properties of a `fisheyeIntrinsics` object. To obtain the intrinsic parameters for your camera, use the **Camera Calibrator** app.

For details about the fisheye camera calibration process, see “Using the Single Camera Calibrator App” (Computer Vision Toolbox) and “Fisheye Calibration Basics” (Computer Vision Toolbox).

Distortion center (pixels) — Center of distortion

`[640, 360]` (default) | real-valued 1-by-2 vector

Center of distortion, specified as real-valued 2-element vector. Units are in pixels.

Image size (pixels) — Image size produced by camera

`[720, 1280]` (default) | real-valued 1-by-2 vector of positive integers

Image size produced by the camera, specified as a real-valued 1-by-2 vector of positive integers of the form [*mrows*,*ncols*]. Units are in pixels.

Mapping coefficients — Polynomial coefficients for projection function

`[320, 0, 0, 0]` (default) | real-valued 1-by-4 vector

Polynomial coefficients for the projection function described by Scaramuzza's Taylor model [1], specified as a real-valued 1-by-4 vector of the form `[a0 a2 a3 a4]`.

Example: `[320, -0.001, 0, 0]`

Stretch matrix — Transforms point from sensor plane to camera plane

`[1, 0; 0, 1]` (default) | real-valued 2-by-2 matrix

Transforms a point from the sensor plane to a pixel in the camera image plane. The misalignment occurs during the digitization process when the lens is not parallel to sensor.

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

Ground Truth

Output location (m) and orientation (rad) — Output location and orientation of sensor

`off` (default) | `on`

Select this parameter to output the location and orientation of the sensor at the **Location** and **Orientation** ports, respectively.

Tips

- To visualize the camera images that are output by the **Image** port, use a Video Viewer or To Video Display block.
- Because the Unreal Engine can take a long time to start up between simulations, consider logging the signals that the sensors output. You can then use this data to develop perception algorithms in MATLAB. See “Mark Signals for Logging” (Simulink).

Algorithms

Sensor Position Transformation

At each simulation time step, the sensor block transforms the position (translation and rotation) of the sensor by using this equation:

$T_{\text{Vehicle}} + T_{\text{Mount}} + T_{\text{Offset}} + T_{\text{Port}}$

This equation contains these steps:

1. Take the world coordinate position of the vehicle to which the sensor is mounted. (T_{Vehicle})
2. Transform the sensor to the mounting position specified by the **Mounting location** parameter. (T_{Mount})
3. Transform the sensor to the position specified by the **Relative translation [X, Y, Z] (m)** and **Relative rotation [Roll, Pitch, Yaw] (deg)** parameters, if enabled. (T_{Offset})

   To enable these parameters, select the **Specify offset** parameter
4. Transform the sensor from the offset position to the position specified by the **Translation** and **Rotation** ports. (T_{Port})

   To enable these ports, select the **Input** parameters corresponding to the relative translation and rotation parameters.

Version History

Introduced in R2019b

R2022b: Simulation 3D Fisheye Camera block requires Computer Vision Toolbox

Behavior change in future release

Starting in R2022b, Simulation 3D Fisheye Camera requires Computer Vision Toolbox.

References

[1] Scaramuzza, D., A. Martinelli, and R. Siegwart. "A Toolbox for Easy Calibrating Omnidirectional Cameras." *Proceedings to IEEE International Conference on Intelligent Robots and Systems (IROS 2006)*. Beijing, China, October 7–15, 2006.

See Also

Blocks

Simulation 3D Camera | Simulation 3D Lidar | Simulation 3D Scene Configuration | Simulation 3D UAV Vehicle

Apps

Camera Calibrator

Objects

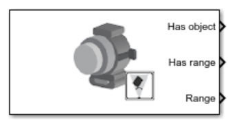
`fisheyeIntrinsics`

Topics

“Coordinate Systems for Unreal Engine Simulation in UAV Toolbox”
“Choose a Sensor for Unreal Engine Simulation”
“Apply Semantic Segmentation Labels to Custom Scenes”
“Fisheye Calibration Basics” (Computer Vision Toolbox)

Simulation 3D Ultrasonic Sensor

Ultrasonic sensor model in 3D simulation environment



Libraries:
Automated Driving Toolbox / Simulation 3D
UAV Toolbox / Simulation 3D

Description

The Simulation 3D Ultrasonic Sensor block generates detections from range measurements taken by an ultrasonic sensor mounted on an ego vehicle in a 3D simulation environment rendered using the Unreal Engine from Epic Games. The block calculates range measurements based on the distance between the sensor and the closest point on the detected object.

If you set **Sample time** to `-1`, the block uses the sample time specified in the Simulation 3D Scene Configuration block. To use this sensor, you must include a Simulation 3D Scene Configuration block in your model.

Note The Simulation 3D Scene Configuration block must execute before the Simulation 3D Ultrasonic Sensor block. That way, the Unreal Engine 3D visualization environment prepares the data before the Simulation 3D Vision Detection Generator block receives it. To check the block execution order, right-click the blocks and select **Properties**. On the **General** tab, confirm these **Priority** settings:

- Simulation 3D Scene Configuration — `0`
- Simulation 3D Ultrasonic Sensor — `1`

For more information about execution order, see “How Unreal Engine Simulation for UAVs Works”.

Ports

Output

Has object — Detectable object present in sensor field-of-view
scalar

Detectable object present in the sensor field-of-view, returned as a Boolean scalar. An object is considered detectable if its closest distance to the sensor is greater than the minimum detection-only range specified in the **Detection Ranges (m)** parameter.

Has range — Range measurement possible for object present in sensor field-of-view
scalar

Range measurement is possible for an object present in the sensor field-of-view, returned as a Boolean scalar. For any object in the field-of-view, range measurement is possible if its closest distance to the sensor is greater than the minimum-distance range specified in the **Detection Ranges (m)** parameter.

Range — Distance measurement to closest object
scalar

Distance measurement to the closest object, returned as a nonnegative scalar, in meters.

Parameters

Mounting

Sensor identifier — Unique sensor identifier

`1` (default) | positive integer

Specify the unique identifier of the sensor. In a multisensor system, the sensor identifier enables you to distinguish between sensors. When you add a new sensor block to your model, the **Sensor identifier** of that block is $N + 1$, where N is the highest **Sensor identifier** value among the existing sensor blocks in the model.

Example: `2`

Parent name — Name of parent vehicle

`Scene Origin` (default) | vehicle name

Name of the parent to which the sensor is mounted, specified as `Scene Origin` or as the name of a vehicle in your model. The vehicle names that you can select correspond to the **Name** parameters of the simulation 3D vehicle blocks in your model. If you select `Scene Origin`, the block places a sensor at the scene origin.

Example: `SimulinkVehicle1`

Mounting location — Sensor mounting location

`Origin` (default)

Sensor mounting location.

- When **Parent name** is `Scene Origin`, the block mounts the sensor to the origin of the scene. You can set the **Mounting location** to `Origin` only. During simulation, the sensor remains stationary.
- When **Parent name** is the name of a vehicle, the block mounts the sensor to one of the predefined mounting locations described in the table. During simulation, the sensor travels with the vehicle.

Vehicle Mounting Location	Description	Orientation Relative to Vehicle Origin [Roll, Pitch, Yaw] (deg)
`Origin`	Forward-facing sensor mounted to the vehicle origin, which is on the ground, at the geometric center of the vehicle	[0, 0, 0]

Roll, pitch, and yaw are clockwise-positive when looking in the positive direction of the *X*-axis, *Y*-axis, and *Z*-axis, respectively. When looking at a vehicle from above, the yaw angle (the orientation angle) is counterclockwise-positive because you are looking in the negative direction of the axis.

The (*X*, *Y*, *Z*) mounting location of the sensor relative to the vehicle depends on the vehicle type. To specify the vehicle type, use the **Type** parameter of the Simulation 3D UAV Vehicle block to which you are mounting. To obtain the (*X*, *Y*, *Z*) mounting locations for a vehicle type, see the reference page for that vehicle.

Specify offset — Specify offset from mounting location

`on` (default) | `off`

Select this parameter to specify an offset from the mounting location by using the **Relative translation [X, Y, Z] (m)** and **Relative rotation [Roll, Pitch, Yaw] (deg)** parameters.

Relative translation [X, Y, Z] (m) — Translation offset relative to mounting location

`[0, 0, 0]` (default) | real-valued 1-by-3 vector

Translation offset relative to the mounting location of the sensor, specified as a real-valued 1-by-3 vector of the form [*X*, *Y*, *Z*]. Units are in meters.

If you mount the sensor to a vehicle by setting **Parent name** to the name of that vehicle, then *X*, *Y*, and *Z* are in the vehicle coordinate system, where:

- The *X*-axis points forward from the vehicle.
- The *Y*-axis points to the left of the vehicle, as viewed when looking in the forward direction of the vehicle.
- The *Z*-axis points up.

The origin is the mounting location specified in the **Mounting location** parameter. This origin is different from the vehicle origin, which is the geometric center of the vehicle.

If you mount the sensor to the scene origin by setting **Parent name** to `Scene Origin`, then *X*, *Y*, and *Z* are in the world coordinates of the scene.

For more details about the vehicle and world coordinate systems, see “Coordinate Systems for Unreal Engine Simulation in UAV Toolbox”.

Example: `[0,0,0.01]`

Dependencies

To enable this parameter, select **Specify offset**.

Relative rotation [Roll, Pitch, Yaw] (deg) — Rotational offset relative to mounting location

`[0, 90, 0]` (default) | real-valued 1-by-3 vector

Rotational offset relative to the mounting location of the sensor, specified as a real-valued 1-by-3 vector of the form [*Roll*, *Pitch*, *Yaw*] . Roll, pitch, and yaw are the angles of rotation about the *X*-, *Y*-, and *Z*-axes, respectively. Units are in degrees.

If you mount the sensor to a vehicle by setting **Parent name** to the name of that vehicle, then *X*, *Y*, and *Z* are in the vehicle coordinate system, where:

- The *X*-axis points forward from the vehicle.
- The *Y*-axis points to the left of the vehicle, as viewed when looking in the forward direction of the vehicle.
- The *Z*-axis points up.
- Roll, pitch, and yaw are clockwise-positive when looking in the forward direction of the *X*-axis, *Y*-axis, and *Z*-axis, respectively. If you view a scene from a 2D top-down perspective, then the yaw angle (also called the orientation angle) is counterclockwise-positive because you are viewing the scene in the negative direction of the *Z*-axis.

The origin is the mounting location specified in the **Mounting location** parameter. This origin is different from the vehicle origin, which is the geometric center of the vehicle.

If you mount the sensor to the scene origin by setting **Parent name** to `Scene Origin`, then *X*, *Y*, and *Z* are in the world coordinates of the scene.

For more details about the vehicle and world coordinate systems, see “Coordinate Systems for Unreal Engine Simulation in UAV Toolbox”.

Example: `[0,0,10]`

Dependencies

To enable this parameter, select **Specify offset**.

Sample time — Sample time

`-1` (default) | positive scalar

Sample time of the block, in seconds, specified as a positive scalar. The 3D simulation environment frame rate is the inverse of the sample time.

If you set the sample time to `-1`, the block inherits its sample time from the Simulation 3D Scene Configuration block.

Sensor Parameters

Detection ranges (m) — Detection range vector of ultrasonic sensor (m)
`[0.03 0.15 5.5]` (default) | 1-by-3 nonnegative real-valued vector of form `[minDetOnlyRange minDistRange maxDistRange]`

Detection range vector of the ultrasonic sensor, specified as a 1-by-3 nonnegative real-valued vector of the form `[minDetOnlyRange minDistRange maxDistRange]`, where `minDetOnlyRange < minDistRange < maxDistRange`. Units are in meters. These values determine the detections and distance values returned by the ultrasonic sensor.

- When the detected object is at a distance between `minDistRange` and `maxDistRange`, the sensor returns a positive distance value.
- When the detected object is at a distance between `minDetOnlyRange` and `minDistRange`, the sensor detects the object, but cannot determine the distance and returns a value of `0`.
- When the object is at a distance below `minDetOnlyRange` or above `maxDistRange`, the sensor returns an empty cell array.

Horizontal Field of view (deg) — Horizontal field of view of ultrasonic sensor
`70` (default) | positive real scalar

Horizontal field of view of ultrasonic sensor, specified as a positive real scalar. This field of view defines the total angular extent spanned by the sensor in the horizontal direction. You must specify the horizontal field of view `horizontalFOV` in the range (0, 360]. Units are in degrees.

Vertical Field of view (deg) — Vertical field of view of ultrasonic sensor
`70` (default) | positive real scalar

Vertical field of view of ultrasonic sensor, specified as a positive real scalar. This field of view defines the total angular extent spanned by the sensor in the vertical direction. You must specify the vertical field of view in the range (0, 180]. Units are in degrees.

Version History

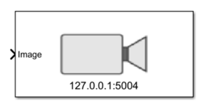
Introduced in R2023a

See Also

Simulation 3D Scene Configuration | Simulation 3D UAV Vehicle | Simulation 3D Lidar | Simulation 3D Camera

Video Send

Send video stream to remote hardware



Libraries:
UAV Toolbox / Simulation 3D

Description

The Video Send block sends video streams from Simulink to a specified remote device. For hardware-in-the-loop simulation applications, you can send grayscale and RGB images, as well as depth and lidar point cloud data collected from Unreal Engine scenes to the remote device. You can also specify the compression and quality of the video stream. To stream the video, this block picks the first available local UDP port, independent of the remote port you specify. This block uses the Gstreamer framework to handle data streaming.

Limitations

- The Video Send block is supported only for use in Windows and Mac.

Ports

Input

Image — RGB or grayscale image signal to stream
M-by-*N*-by-*3* matrix | *M*-by-*N* matrix

RGB or grayscale image signal to stream, specified as an *M*-by-*N*-by-*3* matrix or *M*-by-*N* matrix, respectively.

Dependencies

The `Image Signal` parameter must be set to `One multidimensional signal`.

Data Types: `uint8` | `uint16`

R — Red channel signal of the RGB image to stream
M-by-*N* matrix.

Red channel signal of the RGB image to stream, specified as an `M-by-N` matrix.

Dependencies

The `Image Signal` parameter must be set to `Separate color signals`.

Data Types: `uint8`

G — Green channel signal of the RGB image to stream
M-by-*N* matrix

Green channel signal of the RGB image to stream, specified as an `M-by-N` matrix.

Dependencies

The `Image Signal` parameter must be set to `Separate color signals`.

Data Types: `uint8`

B — Blue channel signal of the RGB image to stream
M-by-*N* matrix

Blue channel signal of the RGB image to stream, specified as an `M-by-N` matrix.

Dependencies

The `Image Signal` parameter must be set to `Separate color signals`.

Data Types: `uint8`

Parameters

Video parameters

Format — Input video stream signal format
`RGB` (default) | `Grayscale` | `Grayscale (16-bit)`

Specify the input video stream signal format as one of the following:

- `RGB` - RGB image (8-bit per color channel).
- `Grayscale` - Grayscale image (8-bit).
- `Grayscale (16-bit)` - Grayscale image (16-bit).

Image signal — Nature of the RGB image input signal
`One multidimensional signal` (default) | `Separate color signals`

Specify the nature of the RGB image input signal as one of the following:

- `One multidimensional signal` - One input port for an *M*-by-*N*-by-*3* color video signal with R, G, and B color channels.
- `Separate color signals` - Three separate input ports for R,G and B channels. Each port accepts one *M*-by-*N* matrix.

Dependencies

The `Format` parameter must be set to `RGB`.

Compression — Compression format for the video stream
`JPEG` (default) | `VP8` | `VP9`

Specify the compression format for the video stream as one of the following:

- `JPEG` - Uses jpeg GStreamer plugin.
  - This uses the following GStreamer pipeline:

    ```
    video/x-raw,format=I420 ! jpegenc quality=<Quality> idct-method=1 ! rtpjpegpay ! udpsink sy
    ```
- `VP8` - Uses vpx GStreamer plugin.

- This uses the following GStreamer pipeline:

  ```
  vp8enc deadline=<Max frame time*1000> bits-per-pixel=0.0434 target-bitrate=0 threads=8 lag-
  ```
- VP9 - Uses vpx GStreamer plugin.
  - This uses the following GStreamer pipeline:

  ```
  vp9enc deadline=<Max frame time*1000> bits-per-pixel=0.0434 target-bitrate=0 threads=8 lag-
  ```

Quality — Quality of the video stream
85 (default) | 0-100

Specify the quality of the JPEG video stream as a positive scalar. This parameter controls the encoding speed and compression ratio of the video stream. A higher quality value increases the image quality at the expense of a higher network bandwidth. Specify a lower value for low-bandwidth network connections.

Dependencies

The Compression parameter must be set to JPEG.

Max frame time (ms) — Maximum time per frame
30 (default) | non-negative integer

Specify the maximum frame time of the VP8 or VP9 video stream in milliseconds, as a positive integer. This parameter controls the maximum processing time that the codec uses to encode an image. Set this parameter based on the frame rate of the input Image. A good starting point value is 1000/fps, where fps is the sample rate of the input video signal. Set 0 if you want the encoder to take as long as it needs, which will increase the quality at the expense of time required for compression.

Dependencies

The Compression parameter must be set to VP8 or VP9.

Connection parameters

Remote address — Remote IP address
127.0.0.1 (default) | character vector

Specify the IP address or host name of the remote device, to which the block sends the message, as a character vector.

Remote port — Remote IP port
5004 (default) | 0-65535

Specify the IP port of the remote device, to which the block sends the message. When streaming the video, this block picks the first available local UDP port, independent of the remote port you specify.

Tips

- VP8 and VP9 compression formats are bandwidth efficient but also computationally expensive. Hence, choose JPEG compression format if you are working on machines with limited CPU resources.

- To receive the video stream on the remote device, in a separate model, use the **Network Video Receive** block from the MATLAB Coder™ Support Package for NVIDIA® Jetson™ and NVIDIA DRIVE® Platforms. Alternatively, you can use your custom GStreamer-based receiver.

Version History

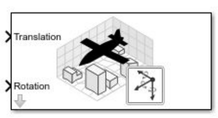
Introduced in R2021b

See Also

Simulation 3D Scene Configuration | Simulation 3D Camera | Simulation 3D UAV Vehicle

Simulation 3D UAV Vehicle

Place UAV vehicle in 3D visualization



Libraries:
UAV Toolbox / Simulation 3D

Description

The Simulation 3D UAV Vehicle block implements an unmanned aerial vehicle (UAV) in a 3D simulation environment. This environment is rendered using the Unreal Engine from Epic Games. The block uses the input (*X*, *Y*, *Z*) position and input (*roll*, *pitch*, *yaw*) attitude of the UAV in the simulation.

To use this block, ensure that the Simulation 3D Scene Configuration block is in your model. If you set the **Sample time** parameter of the Simulation 3D UAV Vehicle block to `-1`, the block inherits the sample time specified in the Simulation 3D Scene Configuration block.

Note The Simulation 3D UAV Vehicle block must execute before the Simulation 3D Scene Configuration block. That way, the Simulation 3D UAV Vehicle block prepares the signal data before the Unreal Engine 3D visualization environment receives it. To check the block execution order, right-click the blocks and select **Properties**. On the **General** tab, confirm these **Priority** settings:

- Simulation 3D Scene Configuration — `0`
- Simulation 3D Vehicle — `-1`

For more information about execution order, see “Block Execution Order”.

Ports

Input

Translation — Translated position of vehicle relative to Unreal Engine scene origin
vector

Translated position of the vehicle relative to the Unreal Engine scene origin.

If you clear the **Use geospatial coordinates for inputs and initial values** parameter, the translation vector defines the *X*-, *Y*-, and *Z*-positions, in meters, of the vehicle using the Unreal Engine world coordinate frame.

If you select the **Use geospatial coordinates for inputs and initial values**, the translation vector defines the latitude, longitude, and altitude of the vehicle in the Unreal Engine world coordinate frame.

For more information on the coordinate systems, see “Coordinate Systems for Unreal Engine Simulation in UAV Toolbox”.

Data Types: `double`

Rotation — Rotation of vehicle relative to Unreal Engine inertial reference frame
vector

Rotation of the vehicle relative to the Unreal Engine inertial reference frame.

If you clear the **Use geospatial coordinates for inputs and initial values** parameter, the rotation vector defines the *Yaw*, *Pitch*, and *Roll* values, in radians, of the vehicle rotation relative to the Unreal Engine world coordinate frame.

If you enable **Use geospatial coordinates for inputs and initial values**, the rotation vector defines the *Yaw*, *Pitch*, and *Roll* values, in radians, of the vehicles rotation relative to the local ENU frame at the current position of the UAV relative to the Unreal Engine world coordinate frame.

For more information on the coordinate systems, see “Coordinate Systems for Unreal Engine Simulation in UAV Toolbox”.

Data Types: `double`

GeoOrigin — Geospatial coordinates of Unreal Engine world origin
vector

Geospatial coordinates of the Unreal Engine world origin, specified in the form [*latitude longitude altitude*]. *latitude* and *longitude* are in degrees, and *altitude* is in meters.

Dependencies

To enable this port, select the **Use geospatial coordinates for inputs and initial values** parameter.

Data Types: `double`

Parameters

Vehicle Parameters

Type — Type of vehicle

`Quadcopter` (default) | `Fixed wing` | `Custom`

Select the type of vehicle. To obtain the dimensions of each vehicle type, see these reference pages:

- `Quadcopter` — **Quadrotor**
- `Fixed wing` — **Fixed Wing Aircraft**
- `Custom` — **Custom UAV**

Path to custom mesh — Path to custom mesh
`/MathWorksSimulation/UAVs/Custom/Meshes/UAV_Custom.UAV_Custom` (default) | valid file path

Specify the path to a custom mesh file for the UAV.

To create a custom UAV mesh, see “Prepare Custom UAV Vehicle Mesh for the Unreal Editor”.

Example: `/MathWorksSimulation/UAVs/Custom/Meshes/UAV_Custom.UAV_Custom`

Dependencies

To enable this parameter, set the **Type** parameter to `Custom`.

Color — Color of vehicle

`Black` (default) | `Orange` | `Yellow` | `Green` | `Blue` | `Red` | `White` | `Silver`

Select the color of the vehicle.

Name — Name of vehicle

`SimulinkVehicle1` (default) | vehicle name

Name of vehicle. By default, when you use the block in your model, the block sets the **Name** parameter to `SimulinkVehicle`*X*. The value of *X* depends on the number of Simulation 3D UAV Vehicle blocks that you have in your model.

The vehicle name appears as a selection in the **Parent name** parameter of any UAV Toolbox Simulation 3D sensor blocks within the same model as the vehicle. With the **Parent name** parameter, you can select the vehicle on which to mount the sensor.

Use geospatial coordinates for inputs and initial values — Use geospatial coordinates for port inputs and initial values

`off` (default) | `on`

Select this parameter to use geospatial coordinates for port inputs and initial values in the form.

Initial Values

Initial Translation (m) — Initial vehicle position
`[0 0 0]` (default) | real-valued 1-by-3 vector

Specify the initial position of your vehicle in one of these forms:

- [*X Y Z*] when you clear the **Use geospatial coordinates for inputs and initial values** parameter. *X*, *Y*, and *Z* are the translation along the *X*-axis, *Y*-axis, and *Z*-axis, respectively, in the inertial *Z*-down coordinate system, in meters.
- [*latitude longitude altitude*] when you enable the **Use geospatial coordinates for inputs and initial values** parameter. You must specify *latitude* and *longitude* in degrees, and *altitude* in meters.

Initial Georeference Origin (LLA) — Initial georeference origin
`[0 0 0]` (default) | real-valued 1-by-3 vector

Specify the initial georeference origin in the form [*latitude longitude altitude*], You must specify *latitude* and *longitude* in degrees, and *altitude* in meters.

Dependencies

To enable this parameter, select the **Use geospatial coordinates for inputs and initial values** parameter.

Initial Rotation (rad) — Initial angle of vehicle rotation
`[0 0 0]` (default) | real-valued 1-by-3 vector

Specify the initial angle of rotation for your vehicle, in radians, in the form [roll pitch yaw].

Sample Time

Sample time — Sample time

-1 (default) | positive scalar

Sample time, T_s, in seconds. The graphics frame rate is the inverse of the sample time.

If you set the sample time to -1, the block uses the sample time specified in the Simulation 3D Scene Configuration block.

Version History

Introduced in R2020b

R2023a: Support for specifying geospatial coordinates

You can select the **Use geospatial coordinates for inputs and initial values** parameter to specify inputs and initial values using geospatial coordinates.

See Also

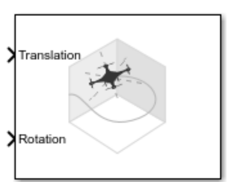
Blocks
Simulation 3D Lidar | Simulation 3D Fisheye Camera | Simulation 3D Scene Configuration

Tools
Quadrotor | Fixed Wing Aircraft

Topics
"Coordinate Systems for Unreal Engine Simulation in UAV Toolbox"
"Choose a Sensor for Unreal Engine Simulation"

UAV Animation

Animate UAV flight path using translations and rotations



Libraries:
UAV Toolbox / Utilities

Description

The UAV Animation block animates a unmanned aerial vehicle (UAV) flight path based on an input array of translations and rotations. A visual mesh is displayed for either a fixed-wing or multirotor at the given position and orientation. Click the **Show animation** button in the block mask to bring up the figure after simulating.

Ports

Input

Translation — *xyz*-positions
`[x y z]` vector

xyz-positions specified as an `[x y z]` vector.

Example: `[1 1 1]`

Rotation — Rotations of UAV body frames
`[w x y z]` quaternion vector

Rotations of UAV body frames relative to the inertial frame, specified as a `[w x y z]` quaternion vector.

Example: `[1 0 0 0]`

Parameters

UAV type — Type of UAV mesh to display
`Multirotor` (default) | `FixedWing`

Type of UAV mesh to display, specified as either `FixedWing` or `Multirotor`.

UAV size — Size of frame and attached mesh
1 (default) | positive numeric scalar

Size of frame and attached mesh, specified as positive numeric scalar.

Inertial frame z-axis direction — Direction of positive *z*-axis of inertial frame
Down (default) | Up

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. The parameter defines the rotation between the inertia frame and plot frame. Set this parameter to Down if the inertial frame is following 'North-East-Down' configuration.

Sample time — Interval between outputs
–1 (default) | scalar

Interval between outputs, specified as a scalar. In simulation, the sample time follows simulation time and not actual wall-clock time.

This default value indicates that the block sample time is *inherited*.

For more information about the inherited sample time type, see “Specify Sample Time” (Simulink).

Version History

Introduced in R2018b

Extended Capabilities

C/C++ Code Generation

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

See Also

Functions

`plotTransforms` | `state`

Objects

`fixedwing` | `multirotor` | `uavWaypointFollower`

Blocks

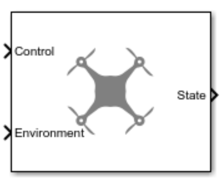
Waypoint Follower

Topics

“Approximate High-Fidelity UAV Model with UAV Guidance Model Block”
“Tuning Waypoint Follower for Fixed-Wing UAV”

Guidance Model

Reduced-order model for UAV



Libraries:
UAV Toolbox / Algorithms

Description

The Guidance Model block represents a small unmanned aerial vehicle (UAV) guidance model that estimates the UAV state based on control and environmental inputs. The model approximates the behavior of a closed-loop system consisting of an autopilot controller and a fixed-wing or multirotor kinematic model for 3-D motion. Use this block as a reduced-order guidance model to simulate your fixed-wing or multirotor UAV. Specify the **ModelType** to select your UAV type. Use the **Initial State** tab to specify the initial state of the UAV depending on the model type. The **Configuration** tab defines the control parameters and physical parameters of the UAV.

Ports

Input

Control — Control commands
bus

Control commands sent to the UAV model, specified as a bus. The name of the input bus is specified in **Input/Output Bus Names**.

For multirotor UAVs, the model is approximated as separate PD controllers for each command. The elements of the bus are control command:

- `Roll` - Roll angle in radians.
- `Pitch` - Pitch angle in radians.
- `YawRate` - Yaw rate in radians per second. (D = 0. P only controller)
- `Thrust` - Vertical thrust of the UAV in Newtons. (D = 0. P only controller)

For fixed-wing UAVs, the model assumes the UAV is flying under the coordinated-turn condition. The guidance model equations assume zero side-slip. The elements of the bus are:

- `Height` - Altitude above the ground in meters.
- `Airspeed` - UAV speed relative to wind in meters per second.
- `RollAngle` - Roll angle along body forward axis in radians. Because of the coordinated-turn condition, the heading angular rate is based on the roll angle.

Environment — Environmental inputs
bus

Environmental inputs, specified as a bus. The model compensates for these environmental inputs when trying to achieve the commanded controls.

For fixed-wing UAVs, the elements of the bus are `WindNorth`, `WindEast`,`WindDown`, and `Gravity`. Wind speeds are in meters per second and negative speeds point in the opposite direction. `Gravity` is in meters per second squared.

For multirotor UAVs, the only element of the bus is `Gravity` in meters per second squared.

Data Types: `bus`

Output

State — Simulated UAV state
bus

Simulated UAV state, returned as a bus. The block uses the `Control` and `Environment` inputs with the guidance model equations to simulate the UAV state.

For multirotor UAVs, the state is a five-element bus:

- **WorldPosition** - `[x y z]` in meters.
- **WorldVelocity** - `[vx vy vz]` in meters per second.
- **EulerZYX** - `[psi phi theta]` Euler angles in radians.
- **BodyAngularRateRPY** - `[r p q]` in radians per second along the `xyz`-axes of the UAV.
- **Thrust** - `F` in Newtons.

For fixed-wing UAVs, the state is an eight-element bus:

- **North** - Position in north direction in meters.
- **East** - Position in east direction in meters.
- **Height** - Height above ground in meters.
- **AirSpeed** - Speed relative to wind in meters per second.
- **HeadingAngle** - Angle between ground velocity and north direction in radians.
- **FlightPathAngle** - Angle between ground velocity and north-east plane in radians.
- **RollAngle** - Angle of rotation along body *x*-axis in radians per second.
- **RollAngleRate** - Angular velocity of rotation along body *x*-axis in radians per second.

Data Types: `bus`

Parameters

ModelType — UAV guidance model type
`MultirotorGuidance` (default) | `FixedWingGuidance`

UAV guidance model type, specified as `MultirotorGuidance` or `FixedWingGuidance`. The model type determines the elements of the UAV `State` and the required `Control` and `Environment` inputs.

Tunable: No

DataType — Input and output numeric data types
`double` (default) | `single`

Input and output numeric data types, specified as either `double` or `single`. Choose the data type based on possible software or hardware limitations.

Tunable: No

Simulate using — Type of simulation to run
`Interpreted execution` (default) | `Code generation`

- `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

Initial State — Initial UAV state tab
multiple table entries

Initial UAV state tab, specified as multiple table entries. All entries on this tab are nontunable.

For multirotor UAVs, the initial state is:

- **World Position** - `[x y z]` in meters.
- **World Velocity** - `[vx vy vz]` in meters per second.
- **Euler Angles (ZYX)** - `[psi phi theta]` in radians.
- **Body Angular Rates** - `[p q r]` in radians per second.
- **Thrust** - `F` in Newtons.

For fixed-wing UAVs, the initial state is:

- **North** - Position in north direction in meters.
- **East** - Position in east direction in meters.
- **Height** - Height above ground in meters.
- **Air Speed** - Speed relative to wind in meters per second.
- **Heading Angle** - Angle between ground velocity and north direction in radians.
- **Flight Path Angle** - Angle between ground velocity and north-east plane in radians.
- **Roll Angle** - Angle of rotation along body *x*-axis in radians per second.
- **Roll Angle Rate** - Angular velocity of rotation along body *x*-axis in radians per second.

Tunable: No

Configuration — UAV controller configuration tab
multiple table entries

UAV controller configuration tab, specified as multiple table entries. This tab allows you to configure the parameters of the internal control behaviour of the UAV. Specify the proportional (P) and derivative (D) gains for the dynamic model and the UAV mass in kilograms (for multirotor).

For multirotor UAVs, the parameters are:

- **PD Roll**
- **PD Pitch**
- **P YawRate**
- **P Thrust**
- **Mass**(kg)

For fixed-wing UAVs, the parameters are:

- **P Height**
- **P Flight Path Angle**
- **PD Roll**
- **P Air Speed**
- **Min/Max Flight Path Angle** (`[min max]` angle in radians)

Tunable: No

Input/Output Bus Names — Simulink bus signal names tab
multiple entries of character vectors

Simulink bus signal names tab, specified as multiple entries of character vectors. These buses have a default name based on the UAV model and input type. To use multiple guidance models in the same Simulink model, specify different bus names that do not intersect. All entries on this tab are nontunable.

More About

UAV Coordinate Systems

The UAV Toolbox uses the North-East-Down (NED) coordinate system convention, which is also sometimes called the local tangent plane (LTP). The UAV position vector consists of three numbers for position along the northern-axis, eastern-axis, and vertical position. The down element complies with the right-hand rule and results in negative values for altitude gain.

The ground plane, or earth frame (NE plane, D = 0), is assumed to be an inertial plane that is flat based on the operation region for small UAV control. The earth frame coordinates are $[x_e,y_e,z_e]$. The body frame of the UAV is attached to the center of mass with coordinates $[x_b,y_b,z_b]$. x_b is the preferred forward direction of the UAV, and z_b is perpendicular to the plane that points downwards when the UAV travels during perfect horizontal flight.

The orientation of the UAV (body frame) is specified in ZYX Euler angles. To convert from the earth frame to the body frame, we first rotate about the z_e-axis by the yaw angle, ψ. Then, rotate about the intermediate y-axis by the pitch angle, ϕ. Then, rotate about the intermediate x-axis by the roll angle, Θ.

The angular velocity of the UAV is represented by $[p,q,r]$ with respect to the body axes, $[x_b,y_b,z_b]$.

UAV Fixed-Wing Guidance Model Equations

For fixed-wing UAVs, the following equations are used to define the guidance model of the UAV. Use the `derivative` function to calculate the time-derivative of the UAV state using these governing equations. Specify the inputs using the `state`, `control`, and `environment` functions.

The UAV position in the earth frame is $[x_e, y_e, h]$ with orientation as heading angle, flight path angle, and roll angle, $[\chi, \gamma, \phi]$ in radians.

The model assumes that the UAV is flying under a coordinated-turn condition, with zero side-slip. The autopilot controls airspeed, altitude, and roll angle. The corresponding equations of motion are:

$$\dot{x}_e = V_g \cos\chi \cos\gamma$$

$$\dot{y}_e = V_g \sin\chi \cos\gamma$$

$$\dot{h} = V_g \sin\gamma$$

$$\dot{\chi} = \frac{g\cos(\chi - \psi)}{V_g} \tan\phi$$

$$V_g \sin(\gamma^c) = min\left(max\left(k_h(h^c - h), -V_g\right), V_g\right)$$

$$\dot{\gamma} = k_\gamma(\gamma^c - \gamma)$$

$$\dot{V}_a = k_{V_a}(V_a^c - V_a)$$

$$\frac{g\cos(\chi - \psi)}{V_g} \tan(\phi^c) = k_\chi(\chi^c - \chi)$$

$$\ddot{\phi} = k_{P\phi}(\phi^c - \phi) + k_{D\phi}(-\dot{\phi})$$

V_a and V_g denote the UAV air and ground speeds.

The wind speed is specified as $[V_{w_n}, V_{w_e}, V_{w_d}]$ for the north, east, and down directions. To generate the structure for these inputs, use the `environment` function.

k_* are controller gains. To specify these gains, use the `Configuration` property of the `fixedwing` object.

From these governing equations, the model gives the following variables:

$$[x_e \quad y_e \quad h \quad V_a \quad \chi \quad \gamma \quad \phi \quad \dot{\phi}]$$

These variables match the output of the `state` function.

UAV Multirotor Guidance Model Equations

For multirotors, the following equations are used to define the guidance model of the UAV. To calculate the time-derivative of the UAV state using these governing equations, use the `derivative` function. Specify the inputs using `state`, `control`, and `environment`.

The UAV position in the earth frame is $[x_e, y_e, z_e]$ with orientation as ZYX Euler angles, $[\psi, \Theta, \phi]$ in radians. Angular velocities are $[p, q, r]$ in radians per second.

The UAV body frame uses coordinates as $[x_b, y_b, z_b]$.

The rotation matrix that rotates vector from body frame to world frame is:

$$R_b^e = \begin{bmatrix} c_\theta c_\psi & c_\psi s_\phi s_\theta - c_\phi s_\psi & c_\phi c_\psi s_\theta + s_\phi s_\psi \\ c_\theta s_\psi & c_\phi c_\psi + s_\phi s_\theta s_\psi & -c_\psi s_\phi + c_\phi s_\theta s_\psi \\ -s_\theta & c_\theta s_\phi & c_\phi c_\theta \end{bmatrix}$$

The cos(x) and sin(x) are abbreviated as c_x and s_x.

The acceleration of the UAV center of mass in earth coordinates is governed by:

$$m \begin{bmatrix} \ddot{x}_e \\ \ddot{y}_e \\ \ddot{z}_e \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ mg \end{bmatrix} + R_b^e \begin{bmatrix} 0 \\ 0 \\ -F_{thrust} \end{bmatrix}$$

m is the UAV mass, g is gravity, and F_{thrust} is the total force created by the propellers applied to the multirotor along the $-z_b$ axis (points upwards in a horizontal pose).

The closed-loop roll-pitch attitude controller is approximated by the behavior of 2 independent PD controllers for the two rotation angles, and 2 independent P controllers for the yaw rate and thrust. The angular velocity, angular acceleration, and thrust are governed by:

$$J = \begin{bmatrix} 1 & \sin\phi\tan\theta & \cos\phi\tan\theta \\ 0 & \cos\phi & -\sin\phi \\ 0 & \dfrac{\sin\phi}{\cos\theta} & \dfrac{\cos\phi}{\cos\theta} \end{bmatrix}$$

$$\begin{bmatrix} \dot{\phi} \\ \dot{\theta} \\ \dot{\psi} \end{bmatrix} = J \begin{bmatrix} p \\ q \\ r \end{bmatrix}$$

$$\begin{bmatrix} \dot{p} \\ \dot{q} \\ \dot{r} \end{bmatrix} = \begin{bmatrix} 1 & 0 & -\sin\theta \\ 0 & \cos\phi & \sin\phi\cos\theta \\ 0 & -\sin\phi & \cos\phi\cos\theta \end{bmatrix} \begin{bmatrix} KP_\phi(\phi^c - \phi) + KD_\phi(-\dot{\phi}) \\ KP_\theta(\theta^c - \theta) + KD_\theta(-\dot{\theta}) \\ KP_\psi(\dot{\psi}^c - \dot{\psi}) \end{bmatrix}$$

$$\dot{F}_{thrust} = KP_F(F^c_{thrust} - F_{thrust})$$

This model assumes the autopilot takes in commanded roll, pitch, yaw rate, $[\phi^c \ \theta^c \ \dot{\psi}^c]$ and a commanded total thrust force, F^c_{thrust}. The structure to specify these inputs is generated from `control`.

The P and D gains for the control inputs are specified as KP_α and KD_α, where α is either the rotation angle or thrust. These gains along with the UAV mass, m, are specified in the `Configuration` property of the `multirotor` object.

From these governing equations, the model gives the following variables:
$[x_e \quad y_e \quad z_e \quad \dot{x}_e \quad \dot{y}_e \quad \dot{z}_e \quad \psi \quad \theta \quad \phi \quad p \quad q \quad r \quad F_{thrust}]$

These variables match the output of the `state` function.

Version History

Introduced in R2018b

References

[1] Randal W. Beard and Timothy W. McLain. "Chapter 9." *Small Unmanned Aircraft Theory and Practice*, NJ: Princeton University Press, 2012.

[2] Mellinger, Daniel, and Nathan Michael. "Trajectory Generation and Control for Precise Aggressive Maneuvers with Quadrotors." *The International Journal of Robotics Research*. 2012, pp. 664-74.

Extended Capabilities

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

See Also

Functions
`ode45` | `control` | `derivative` | `environment` | `state` | `plotTransforms`

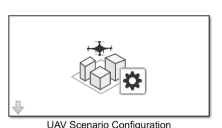
Objects
`fixedwing` | `multirotor` | `uavWaypointFollower`

Blocks
Waypoint Follower

Topics
"Approximate High-Fidelity UAV Model with UAV Guidance Model Block"
"Tuning Waypoint Follower for Fixed-Wing UAV"

UAV Scenario Configuration

Configure and simulate UAV scenarios



Libraries:
UAV Toolbox / UAV Scenario and Sensor Modeling

Description

This block allows you to import a `uavScenario` object and simulate the scenario. This block must be in a model that contains a UAV Scenario Lidar and Motion blocks in order to test perception, control, and planning algorithms with data from a `uavScenario` environment. Models with a UAV Scenario can only use one UAV Scenario Configuration block at a time and is not intended to be used in a cross-model workflow. In a UAV scenario, this block must be executed before other UAV Scenario blocks. Update the UAV Scenario Configuration block with the **Refresh** button and click **Apply** to reflect any changes made to the imported `uavScenario` in MATLAB.

This block internally stores motion states from platforms and sensors in a global data store memory block as buses within a bus with a name specified in the **Scenario motion bus name** parameter. The bus contains the following fields:

- *NumPlatforms* — Number of UAV platforms in the scenario stored as a scalar.
- *Time* — Simulation time associated with the motion state stored as a scalar.
- *Platforms* — Bus array with name specified in the **Platform motion bus name** parameter containing all platforms in the UAV Scenario with each of the following fields:

  - *PlatformID* — ID of platform based on order of platforms in the `uavScenario.Platforms` Property.
  - *Position* — Position in NED frame specified as a 1-by-3 vector.
  - *Orientation* — Orientation in NED frame specified as a 1-by-4 vector, quaternion, frame rotation from NED frame to UAV body frame.
  - *Velocity* — Velocity in NED frame specified as a 1-by-3 vector.
  - *Acceleration* — Acceleration in NED frame specified as a 1-by-3 vector.
  - *AngularVelocity* — Angular velocity in NED frame specified as a 1-by-3 vector.

Limitations

The UAV Scenario blocks do not support:

- Code generation
- Model reference
- Multiple instances of the UAV Scenario Configuration block
- Rapid acceleration mode

In addition, the execution order is important when using these blocks in a closed loop simulation. The UAV Scenario Configuration block must execute first. The UAV Scenario Motion Write block must execute before the UAV Scenario Motion Read, UAV Scenario Lidar, and UAV Scenario Scope blocks.

Parameters

Main

MATLAB or model workspace variable name — MATLAB or model workspace variable name
`'singleUAVScenario'` (default) | string

Specify `uavScenario` object to import from the MATLAB or model workspace, specified by variable name as a string. Click **Refresh** to reload the scenario from the `uavScenario` object.

Sample time — Sample time
`0.1` (default) | positive double

Specify sample time of the UAV scenario in seconds. This block only supports discrete sample time.

Scenario Bus and Signal Names

Sensor motion bus name — Sensor motion bus name
`'SensorMotionBus'` (default) | string

Specify the name of the sensor motion bus type as a string. Sensor motion read by the UAV Scenario Motion Read uses a bus object with the specified name.

Platform motion bus name — Platform motion bus name
`'PlatformMotionBus'` (default) | string

Specify the name of the platform motion bus type as a string. Platform motion read and written by the UAV Scenario Motion Read and UAV Scenario Motion Write respectively will use a bus object with the specified name.

Scenario motion bus name — Scenario motion bus name
`'SensorMotionBus'` (default) | string

Specify the name of the scenario motion bus type as a string. This bus type is used by the UAV Scenario Configuration block to store the motion data of all the platforms and sensors within the UAV Scenario.

Scenario motion signal name — Scenario motion signal name
`'ScenarioMotions'` (default) | string

Specify the name of the scenario motion bus signal as a string. This is the name used for the Data Store Read block that stores the data for the UAV Scenario.

Version History

Introduced in R2021b

See Also

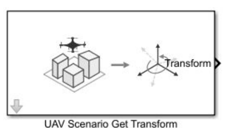
Objects
`uavScenario`

Blocks
UAV Scenario Get Transform | UAV Scenario Lidar | UAV Scenario Motion Read | UAV Scenario Motion Write | UAV Scenario Scope

Topics
"UAV Scenario Tutorial"

UAV Scenario Get Transform

Get transforms from UAV scenario platforms



Libraries:
UAV Toolbox / UAV Scenario and Sensor Modeling

Description

This block outputs a 4-by-4 transformation matrix that maps points in source frame to target frame in a UAV Scenario.

To use this block, ensure that UAV Scenario Configuration block is in your model.

This block uses the sample time specified in the UAV Scenario Configuration block.

Limitations

The UAV Scenario blocks do not support:

- Code generation
- Model reference
- Multiple instances of the UAV Scenario Configuration block
- Rapid acceleration mode

In addition, the execution order is important when using these blocks in a closed loop simulation. The UAV Scenario Configuration block must execute first. The UAV Scenario Motion Write block must execute before the UAV Scenario Motion Read, UAV Scenario Lidar, and UAV Scenario Scope blocks.

Ports

Output

transform — Transformation matrix
4-by-4 matrix

Output transformation specified as a 4-by-4 transformation matrix.

Parameters

Source Frame — Source frame
`'ENU'` (default) | `'NED'` | string from UAV Scenario

Use **Select** to choose a global coordinate source frame from the UAV Scenario loaded in your model. Select either North-East-Down (`'NED'`), or East-North-Up (`'ENU'`).

Target Frame — Target frame
`'NED'` (default) | `'ENU'` | string from UAV Scenario

Use **Select** to choose a global coordinate source frame from the UAV Scenario loaded in your model. Select either North-East-Down (`'NED'`), or East-North-Up (`'ENU'`).

Version History

Introduced in R2021b

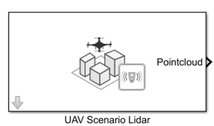
See Also

Functions
`getTransform`

Blocks
UAV Scenario Configuration | UAV Scenario Lidar | UAV Scenario Motion Read | UAV Scenario Motion Write | UAV Scenario Scope

UAV Scenario Lidar

Simulate lidar measurements based on meshes in scenario



Libraries:
UAV Toolbox / UAV Scenario and Sensor Modeling

Description

Use this block to simulate lidar measurements by outputting point cloud data based on meshes in a UAV Scenario. To add meshes to your UAV Scenario, use the `addMesh` function to add it to the `uavScenario` object included by your UAV Scenario Configuration block. See "UAV Scenario Tutorial" for more information on adding meshes.

To use this block, ensure that UAV Scenario Configuration block is in your model.

Limitations

The UAV Scenario blocks do not support:

- Code generation
- Model reference
- Multiple instances of the UAV Scenario Configuration block
- Rapid acceleration mode

In addition, the execution order is important when using these blocks in a closed loop simulation. The UAV Scenario Configuration block must execute first. The UAV Scenario Motion Write block must execute before the UAV Scenario Motion Read, UAV Scenario Lidar, and UAV Scenario Scope blocks.

Ports

Output

Pointcloud — Point cloud data
N-by-*M*-by-3 double matrix | *N*-by-3 double matrix

Point cloud data reported in the Sensor Frame. When `Output organized point cloud locations` is checked, the point cloud data is specified as a *N*-by-*M*-by-3 double matrix, where *N* is the number of vertical scans, and *M* is the number of horizontal scans. When `Output organized point cloud locations` is unchecked, the point cloud data is specified as an *N*-by-3 double matrix, where *N* is the number of points in the point cloud.

Parameters

Sensor name — Lidar name
`'UAV/Lidar'` (default) | string

Use **Select** to choose a lidar sensor from the UAV Scenario.

To add sensors to the UAV Scenario, create `uavSensor` objects with a `uavLidarPointCloudGenerator` as the specified sensor model and attach the `uavSensor` to a `uavPlatform` in UAV Scenario. This scenario must be imported into Simulink using the UAV Scenario Configuration block. The available sensors will be listed as '*platform name/sensor name*'.

Max range — Max range of lidar
`120` (default) | double

Specify max range of lidar as a double scalar in meters.

Range Accuracy — Range accuracy of lidar
`0.002` (default) | double

Specify range accuracy of lidar as a double scalar in meters.

Azimuthal limits — Azimuthal limits of lidar
`[-180, 180]` (default) | two-element vector

Specify azimuthal limits of lidar as a two-element vector in degrees.

Azimuthal resolution — Azimuthal resolution
`0.16` (default) | double

Specify azimuthal resolution of lidar as a double scalar in degrees.

Elevation limits — Elevation limits of lidar
`[-20, 20]` (default) | two-element vector

Specify elevation limits of lidar as a two element vector in degrees.

Elevation resolution — Elevation resolution
`1.25` (default) | double

Specify elevation resolution of lidar as a double scalar in degrees.

Add noise to measurement — Add noise to measurement
`'on'` (default) | `'off'`

Check this box to add noise to measurement of lidar. The noise generation of this block currently does not allow for a user-specified seed.

Output organized point cloud locations — Output point cloud locations
`'on'` (default) | `'off'`

Check this box to output organized point cloud locations specified as *N*-by-*M*-by-3, where *N* is the number of vertical scans, and *M* is the number of horizontal scans. If set to `'off'`, the output of the block is an *N*-by-3 double matrix, where *N* is the number of points in the point cloud.

Sample Time — Sample time
`0.1` (default) | double

Specify sample time of the lidar as a double scalar in seconds. Sample time must be a multiple of the sample time specified in the UAV Scenario Configuration block.

Version History

Introduced in R2021b

See Also

Functions
`addMesh` | `addCustomTerrain`

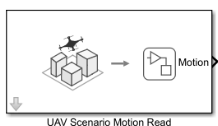
Objects
`uavLidarPointCloudGenerator` | `uavScenario` | `uavSensor`

Blocks
UAV Scenario Configuration | UAV Scenario Get Transform | UAV Scenario Motion Read | UAV Scenario Motion Write | UAV Scenario Scope

Topics
"UAV Scenario Tutorial"

UAV Scenario Motion Read

Read platform and sensor motions from UAV scenario simulation



Libraries:
UAV Toolbox / UAV Scenario and Sensor Modeling

Description

Use this block to read motion as a bus from a sensor or platform in a UAV scenario simulation. The motion bus contains the position, orientation, velocity, angular velocity, acceleration, and ID of the platform or sensor.

To use this block, ensure that a UAV Scenario Configuration block is in your model.

This block uses the sample time specified in the UAV Scenario Configuration block.

Limitations

The UAV Scenario blocks do not support:

- Code generation
- Model reference
- Multiple instances of the UAV Scenario Configuration block
- Rapid acceleration mode

In addition, the execution order is important when using these blocks in a closed loop simulation. The UAV Scenario Configuration block must execute first. The UAV Scenario Motion Write block must execute before the UAV Scenario Motion Read, UAV Scenario Lidar, and UAV Scenario Scope blocks.

Ports

Output

Motion — UAV motion data
bus

Output motion from a platform or sensor specified as a bus with the following properties with the name specified in Platform or sensor name:

- *PlatformID* – ID of platform based on order of platforms in the `uavScenario.Platforms` property.
- *Position* – Position in NED frame specified as a 1-by-3 vector.
- *Orientation* – Orientation in NED frame specified as a 1-by-4 vector, quaternion, frame rotation from NED frame to UAV body frame.

- *Velocity* – Velocity in NED frame specified as a 1-by-3 vector.
- *Acceleration* – Acceleration in NED frame specified as a 1-by-3 vector.
- *AngularVelocity* – Angular velocity in NED frame specified as a 1-by-3 vector.

If a sensor name is specified in `Platform or sensor name`, the *SensorID* field is added to the bus fields of the motion output bus.

- *SensorID* – ID of sensor specified as a scalar based on the order of sensors in the `uavScenario.Platforms` property.

The platform and sensors motion buses are of the type named in the `Sensor motion bus name` and `Sensor motion bus name` properties of the UAV Scenario Configuration block in the model.

Parameters

Platform or sensor name — Platform or sensor name
`'UAV'` (default) | string

Use **Select** to choose one platform or sensor from the UAV Scenario to read motion from.

This parameter effects the fields of the motion bus output. See `Motion` output for more information.

Coordinate frame of output motion — Coordinate frame of output motion
`'ENU'` (default) | `'NED'` | string

Specify the coordinate frame of the output motion as East-North-Up (`'ENU'`) or North-East-Down (`'NED'`).

Version History

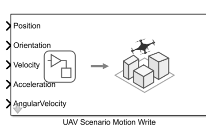
Introduced in R2021b

See Also

Blocks
UAV Scenario Configuration | UAV Scenario Get Transform | UAV Scenario Lidar | UAV Scenario Motion Write | UAV Scenario Scope

UAV Scenario Motion Write

Update platform motion in UAV scenario simulation



Libraries:
UAV Toolbox / UAV Scenario and Sensor Modeling

Description

Use this block to update the motion state of a platform in a UAV scenario simulation with input signals at each time step. The block takes the position, orientation, velocity, acceleration, and angular velocity as inputs to update the motion state bus of the specified platform. However, this block does not move the platform based on velocity and acceleration inputs as there is no kinematic model being simulated in the UAV Scenario.

To use this block, ensure that UAV Scenario Configuration block is in your model.

This block uses the sample time specified in the UAV Scenario Configuration block.

Limitations

The UAV Scenario blocks do not support:

- Code generation
- Model reference
- Multiple instances of the UAV Scenario Configuration block
- Rapid acceleration mode

In addition, the execution order is important when using these blocks in a closed loop simulation. The UAV Scenario Configuration block must execute first. The UAV Scenario Motion Write block must execute before the UAV Scenario Motion Read, UAV Scenario Lidar, and UAV Scenario Scope blocks.

Ports

Input

Position — Position
1-by-3 vector

Specify input position of platform in input coordinate frame as a 1-by-3 vector.

Orientation — Orientation quaternion
1-by-4 vector

Specify input orientation of platform in input coordinate frame as a 1-by-4 vector quaternion.

Velocity — Velocity
1-by-3 vector

Specify input velocity of platform in input coordinate frame as a 1-by-3 vector.

Acceleration — Acceleration
1-by-3 vector

Specify input acceleration of platform in input coordinate frame as a 1-by-3 vector.

AngularVelocity — Angular Velocity
1-by-3 vector

Specify input angular velocity of platform in input coordinate frame, specified as a 1-by-3 vector.

Parameters

Platform name — Platform name
`'UAV'` (default) | string

Use **Select** to choose a platform name string from the UAV Scenario.

Coordinate frame of input motion — Coordinate frame of input motion
`'NED'` (default) | `ENU`

Specify coordinate frame of input motion as `'NED'`(North-East-Down) or `'ENU'`(East-North-Up).

Version History

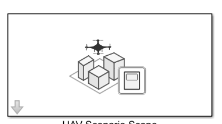
Introduced in R2021b

See Also

Blocks
UAV Scenario Configuration | UAV Scenario Get Transform | UAV Scenario Lidar | UAV Scenario Motion Read | UAV Scenario Scope

UAV Scenario Scope

Visualize UAV scenario and lidar point clouds



Libraries:
UAV Toolbox / UAV Scenario and Sensor Modeling

Description

Use this block to visualize a UAV scenario and lidar point clouds in an animation figure window. Click **Show animation** in the block parameters to visualize the UAV scenario. The visualization in the figure window updates continuously as the model is running.

An input port is created for every sensor checked for visualization in the **Visualize** column of the `Lidar Sensors` table.

To use this block, ensure that UAV Scenario Configuration block is in your model.

Limitations

The UAV Scenario blocks do not support:

- Code generation
- Model reference
- Multiple instances of the UAV Scenario Configuration block
- Rapid acceleration mode

In addition, the execution order is important when using these blocks in a closed loop simulation. The UAV Scenario Configuration block must execute first. The UAV Scenario Motion Write block must execute before the UAV Scenario Motion Read, UAV Scenario Lidar, and UAV Scenario Scope blocks.

Ports

Input

In_1, In_2, ..., In_N — Point cloud data inputs for lidar sensors
N-by-*M*-by-3 double matrix | *N*-by-3 double matrix

Input point cloud data for sensors in the UAV Scenario. An input port is created for every lidar sensor checked for visualization in the `Lidar Sensors` table in the **Visualize** column. The name of the port is set to the name of the sensor, which includes the name of the platform the sensor belongs too. For example, an input port for a lidar sensor on a platform named `EgoVehicle` will have the name, `EgoVehicle/Lidar`.

If the `Organized point cloud locations` parameter of the lidar sensor is set to `'on'`, then the input port expects a *N*-by-*M*-by-3 double matrix, where *N* is the number of vertical scans, and *M* is the number of horizontal scans. If the `Organized point cloud locations` of the lidar sensor is

set to `'off'`, the input port expects an *N*-by-3 double matrix, where *N* is the number of points in the point cloud.

Parameters

Lidar Sensors — List of lidar sensors and their visualization
table

The `Lidar Sensors` table lists all of the lidar sensors in the UAV scenario. In the first column, **Sensor name**, all of the sensor names are listed with their associated platform as '*platform name/sensor name*'. The second column, **Visualize**, contains a check-box for every lidar sensor in the table. Setting a visualization to `'on'` for a lidar sensor creates an input port with the name of that lidar sensor.

Enter a string in the **Filter table contents** to filter through the list of sensors by their name in the **Sensor Name** column of the table.

If the list of lidar sensors in the UAV Scenario has changed, click **Refresh sensor table** to update the `Lidar Sensors` table.

Click **Show animation** to show or hide the animation figure for the UAV scenario.

Sample time — Sample time
`0.1` (default)

Specify sample time of the visualization. If `Sample time` is set to `-1`, this block uses the sample time specified in the UAV Scenario Configuration block.

More About

Animation Window Buttons

Change the focus between platforms by using the previous view and next view buttons. Previous view will show the previous platform in the list of platforms. If the view is already focused on the first platform in the list, it will change to home view. Next performs similarly, changing to home view if the view is currently focused on the last platform.

Rotate, Pan, Zoom, and Home all interact with the animation the same as they do for regular plots.

Version History

Introduced in R2021b

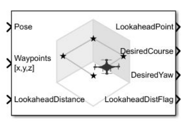
See Also

Blocks

UAV Scenario Configuration | UAV Scenario Get Transform | UAV Scenario Lidar | UAV Scenario Motion Read | UAV Scenario Motion Write

Waypoint Follower

Follow waypoints for UAV



Libraries:
UAV Toolbox / Algorithms

Description

The Waypoint Follower block follows a set of waypoints for an unmanned aerial vehicle (UAV) using a lookahead point. The block calculates the lookahead point, desired course, and desired yaw given a UAV position, a set of waypoints and a lookahead distance. Specify a set of waypoints and tune the lookahead distance and transition radius parameters for navigating the waypoints. The block supports both multirotor and fixed-wing UAV types.

Ports

Input

Pose — Current UAV pose
`[x y z chi]` vector

Current UAV pose, specified as a `[x y z chi]` vector. This pose is used to calculate the lookahead point based on the input to the **LookaheadDistance** port. `[x y z]` is the current position in meters. `chi` is the current course in radians. The course input is used only when the waypoints are empty. The UAV course is the angle of direction of the velocity vector relative to north measured in radians.

Example: `[0.5;1.75;-2.5;pi]`

Data Types: `single` | `double`

Waypoints — Set of waypoints
n-by-3 matrix | *n*-by-4 matrix | *n*-by-5 matrix

Set of waypoints for the UAV to follow, specified as a matrix with number of rows, *n*, equal to the number of waypoints. The number of columns depend on the **Show Yaw input variable** and the **Transition radius source** parameter.

Each row in the matrix has the first three elements as an `[x y z]` position in the sequence of waypoints.

If **Show Yaw input variable** is checked, specify the desired yaw angle, `yaw`, as the fourth element in radians.

If **Show Yaw input variable** is unchecked, and **Transition radius source** is `external`, the transition radius is the fourth element of the vector in meters.

If **Show Yaw input variable** is checked, and **Transition radius source** is `external`, the transition radius is the fifth element of the vector in meters.

The block display updates as the size of the waypoint matrix changes.

Data Types: `single` | `double`

LookaheadDistance — Lookahead distance
positive numeric scalar

Lookahead distance along the path, specified as a positive numeric scalar in meters.

Data Types: `single` | `double`

Output

LookaheadPoint — Lookahead point on path
`[x y z]` position vector

Lookahead point on path, returned as an `[x y z]` position vector in meters.

Data Types: `single` | `double`

DesiredCourse — Desired course
numeric scalar

Desired course, returned as numeric scalar in radians in the range of `[-pi, pi]`. The UAV course is the angle of direction of the velocity vector relative to north measured in radians. For fixed-wing type UAV, the values of desired course and desired yaw are equal.

Data Types: `single` | `double`

DesiredYaw — Desired yaw
numeric scalar

Desired yaw, returned as numeric scalar in radians in the range of `[-pi, pi]`. The UAV yaw is the forward direction of the UAV regardless of the velocity vector relative to north measured in radians. The desired yaw is computed using linear interpolation between the yaw angle for each waypoint. For fixed-wing type UAV, the values of desired course and desired yaw are equal.

Data Types: `single` | `double`

LookaheadDistFlag — Lookahead distance flag
`0` (default) | `1`

Lookahead distance flag, returned as `0` or `1`. `0` indicates lookahead distance is not saturated, `1` indicates lookahead distance is saturated to minimum lookahead distance value specified.

Data Types: `uint8`

CrossTrackError — Cross track error from UAV position to path
positive numeric scalar

Cross track error from UAV position to path, returned as a positive numeric scalar in meters. The error measures the perpendicular distance from the UAV position to the closest point on the path.

Dependencies

This port is only visible if **Show CrossTrackError output port** is checked.

Data Types: `single` | `double`

Status — Status of waypoint navigation
0 | 1

Status of waypoint navigation, returned as 0 or 1. When the follower has navigated all waypoints, the block outputs 1. Otherwise, the block outputs 0.

Dependencies

This port is only visible if **Show UAV Status output port** is checked.

Data Types: uint8

Parameters

UAV type — Type of UAV
fixed-wing (default) | multirotor

Type of UAV, specified as either fixed-wing or multirotor.

This parameter is non-tunable.

StartFrom — Waypoint start behavior
first (default) | closest

Waypoint start behavior, specified as either first or closest.

When set to first, the UAV flies to the first path segment between waypoints. If the set of waypoints input in Waypoints changes, the UAV restarts at the first path segment.

When set to closest, the UAV flies to the closest path segment between waypoints. When the waypoints input changes, the UAV recalculates the closest path segment.

This parameter is non-tunable.

Transition radius source — Source of transition radius
internal (default) | external

Source of transition radius, specified as either internal or external. If specified as internal, the transition radius for each waypoint is set using the **Transition radius (r)** parameter in the block mask. If specified as external, specify each waypoints transition radius independently using the input from the **Waypoints** port.

When the UAV is within the transition radius, the block transitions to following the next path segment between waypoints.

This parameter is non-tunable.

Transition radius (r) — Transition radius for waypoints
10 (default) | positive numeric scalar

Transition radius for waypoints, specified as a positive numeric scalar in meters.

When the UAV is within the transition radius, the block transitions to following the next path segment between waypoints.

This parameter is non-tunable.

Minimum lookahead distance (m) — Minimum lookahead distance
`0.1` (default) | positive numeric scalar

Minimum lookahead distance, specified as a positive numeric scalar in meters.

When input to the **LookaheadDistance** port is less than the minimum lookahead distance, the **LookaheadDistFlag** is returned as `1` and the lookahead distance value is specified as the value of minimum lookahead distance.

This parameter is non-tunable.

Show Yaw input variable — Accept yaw input for waypoints
`off` (default) | `on`

Accept yaw inputs for waypoints when selected. If selected, the **Waypoints** input accepts yaw inputs for each waypoint.

Show CrossTrackError output port — Output cross track error
`off` (default) | `on`

Output cross track error from the **CrossTrackError** port.

This parameter is non-tunable.

Show UAV Status output port — Output UAV waypoint status
`off` (default) | `on`

Output UAV waypoint status from the **Status** port.

This parameter is non-tunable.

Simulate using — Type of simulation to run
`Interpreted execution` (default) | `Code generation`

- `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.
- `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`.

This parameter is non-tunable.

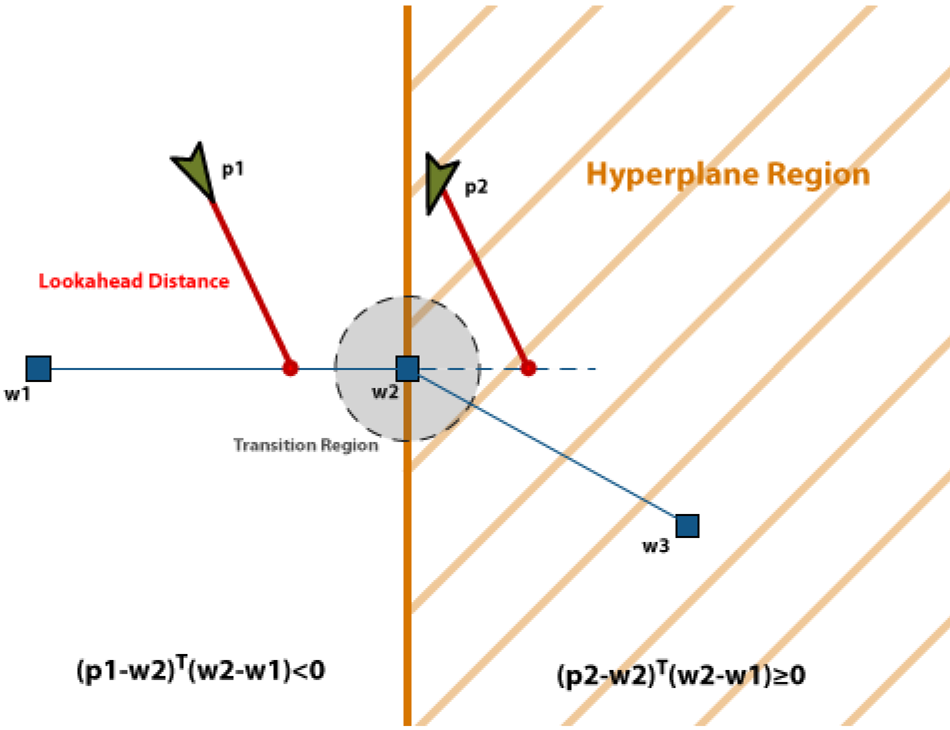
Tunable: No

More About

Waypoint Hyperplane Condition

When following a set of waypoints, the first waypoint may be ignored based on the pose of the UAV. Due to the nature of the lookahead distance used to track the path, the waypoint follower checks if the UAV is near the next waypoint to transition to the next path segment using a transition region. However, there is also a condition where the UAV transitions when outside of this region. A 3-D

hyperplane is drawn at the next waypoint. If the UAV pose is inside this hyperplane, the waypoint follower transitions to the next waypoint. This behavior helps to ensure the UAV follows an achievable path.



The hyperplane condition is satisfied if:

$(p\text{-}w1)^{T}\ (w2\text{-}w1) \geq 0$

p is the UAV position, and *w1* and *w2* are sequential waypoint positions.

If you find this behavior limiting, consider adding more waypoints based on your initial pose to force the follower to navigate towards your initial waypoint.

Version History

Introduced in R2018b

References

[1] Park, Sanghyuk, John Deyst, and Jonathan How. "A New Nonlinear Guidance Logic for Trajectory Tracking." *AIAA Guidance, Navigation, and Control Conference and Exhibit*, 2004.

Extended Capabilities

C/C++ Code Generation

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

See Also

Blocks
Orbit Follower | UAV Guidance Model

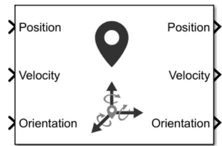
Functions
`ode45` | `control` | `derivative` | `environment` | `state` | `plotTransforms`

Objects
`fixedwing` | `multirotor` | `uavWaypointFollower`

Topics
"Approximate High-Fidelity UAV Model with UAV Guidance Model Block"
"Tuning Waypoint Follower for Fixed-Wing UAV"

INS

Simulate INS sensor



Libraries:
Navigation Toolbox / Multisensor Positioning / Sensor Models
Automated Driving Toolbox / Driving Scenario and Sensor Modeling
Sensor Fusion and Tracking Toolbox / Multisensor Positioning / Sensor Models
UAV Toolbox / UAV Scenario and Sensor Modeling

Description

The block simulates an INS sensor, which outputs noise-corrupted position, velocity, and orientation based on the corresponding inputs. The block can also optionally output acceleration and angular velocity based on the corresponding inputs. To change the level of noise present in the output, you can vary the roll, pitch, yaw, position, velocity, acceleration, and angular velocity accuracies. The accuracy is defined as the standard deviation of the noise.

Ports

Input

Position — Position of INS sensor
N-by-3 real-valued matrix

Position of the INS sensor relative to the navigation frame, in meters, specified as an *N*-by-3 real-valued matrix. *N* is the number of samples.

Data Types: `single` | `double`

Velocity — Velocity of INS sensor
N-by-3 real-valued matrix of scalar

Velocity of the INS sensor relative to the navigation frame, in meters per second, specified as an *N*-by-3 real-valued matrix. *N* is the number of samples.

Data Types: `single` | `double`

Orientation — Orientation of INS sensor
3-by-3-by-*N* real-valued array | *N*-by-4 real-valued matrix | *N*-by-3 matrix of Euler angles

Orientation of the INS sensor relative to the navigation frame, specified as one of these formats:

- A 3-by-3-by-*N* real-valued array, where each page of the array (3-by-3 matrix) is a rotation matrix.
- An *N*-by-4 real-valued matrix, where each row of the matrix is the four elements of a quaternion.
- An *N*-by-3 matrix of Euler angles, where each row of the matrix is the three Euler angles corresponding to the z-y-x rotation convention.

N is the number of samples.

Data Types: `single` | `double`

Acceleration — Acceleration of INS sensor
N-by-3 real-valued matrix

Acceleration of the INS sensor relative to the navigation frame, in meters per second squared, specified as an *N*-by-3 real-valued matrix. *N* is the number of samples.

Dependencies

To enable this input port, select **Use acceleration and angular velocity**.

Data Types: `single` | `double`

AngularVelocity — Angular velocity of INS sensor
N-by-3 real-valued matrix

Angular velocity of the INS sensor relative to the navigation frame, in degrees per second, specified as an *N*-by-3 real-valued matrix. *N* is the number of samples.

Dependencies

To enable this input port, select **Use acceleration and angular velocity**.

Data Types: `single` | `double`

HasGNSSFix — Enable GNSS fix
N-by-1 logical vector

Enable GNNS fix, specified as an *N*-by-1 logical vector. *N* is the number of samples. Specify this input as `false` to simulate the loss of a GNSS receiver fix. When a GNSS receiver fix is lost, position measurements drift at a rate specified by the **Position error factor** parameter.

Dependencies

To enable this input port, select **Enable HasGNSSFix port**.

Data Types: `single` | `double`

Output

Position — Position of INS sensor
N-by-3 real-valued matrix

Position of the INS sensor relative to the navigation frame, in meters, returned as an *N*-by-3 real-valued matrix. *N* is the number of samples in the input.

Data Types: `single` | `double`

Velocity — Velocity of INS sensor
N-by-3 real-valued matrix

Velocity of the INS sensor relative to the navigation frame, in meters per second, returned as an *N*-by-3 real-valued matrix. *N* is the number of samples in the input.

Data Types: `single` | `double`

Orientation — Orientation of INS sensor
3-by-3-by-*N* real-valued array | *N*-by-4 real-valued matrix

Orientation of the INS sensor relative to the navigation frame, returned as one of the formats:

- A 3-by-3-by-*N* real-valued array, where each page of the array (3-by-3 matrix) is a rotation matrix.
- An *N*-by-4 real-valued matrix, where each row of the matrix is the four elements of a quaternion.
- An *N*-by-3 matrix of Euler angles, where each row of the matrix is the three Euler angles corresponding to the z-y-x rotation convention.

N is the number of samples in the input.

Data Types: `single` | `double`

Acceleration — Acceleration of INS sensor
N-by-3 real-valued matrix

Acceleration of the INS sensor relative to the navigation frame, in meters per second squared, returned as an *N*-by-3 real-valued matrix. *N* is the number of samples.

Dependencies

To enable this output port, select **Use acceleration and angular velocity**.

Data Types: `single` | `double`

AngularVelocity — Angular velocity of INS sensor
N-by-3 real-valued matrix

Angular velocity of the INS sensor relative to the navigation frame, in degrees per second, returned as an *N*-by-3 real-valued matrix. *N* is the number of samples.

Dependencies

To enable this output port, select **Use acceleration and angular velocity**.

Data Types: `single` | `double`

Parameters

Mounting location (m) — Location of sensor on platform (m)

`[0 0 0]` (default) | three-element real-valued vector of form [*x y z*]

Location of the sensor on the platform, in meters, specified as a three-element real-valued vector of the form [*x y z*]. The vector defines the offset of the sensor origin from the origin of the platform.

Data Types: `single` | `double`

Roll (X-axis) accuracy (deg) — Accuracy of roll measurement (deg)

`0.2` (default) | nonnegative real scalar

Accuracy of the roll measurement of the sensor body in degrees, specified as a nonnegative real scalar.

Roll is defined as rotation around the *x*-axis of the sensor body. Roll noise is modeled as white process noise with standard deviation equal to the specified **Roll accuracy** in degrees.

Data Types: `single` | `double`

Pitch (Y-axis) accuracy (deg) — Accuracy of pitch measurement (deg)

`0.2` (default) | nonnegative real scalar

Accuracy of the pitch measurement of the sensor body in degrees, specified as a nonnegative real scalar.

Pitch is defined as rotation around the *y*-axis of the sensor body. Pitch noise is modeled as white process noise with standard deviation equal to the specified **Pitch accuracy** in degrees.

Data Types: `single` | `double`

Yaw (Z-axis) accuracy (deg) — Accuracy of yaw measurement (deg)

`1` (default) | nonnegative real scalar

Accuracy of the yaw measurement of the sensor body in degrees, specified as a nonnegative real scalar.

Yaw is defined as rotation around the *z*-axis of the sensor body. Yaw noise is modeled as white process noise with standard deviation equal to the specified **Yaw accuracy** in degrees.

Data Types: `single` | `double`

Position accuracy (m) — Accuracy of position measurement (m)

`1` (default) | nonnegative real scalar | 1-by-3 vector of nonnegative values

Accuracy of the position measurement of the sensor body in meters, specified as a nonnegative real scalar or a 1-by-3 vector of nonnegative values. If you specify the parameter as a scalar value, then the block sets the accuracy of all three position components to this value.

Position noise is modeled as white process noise with a standard deviation equal to the specified **Position accuracy** in meters.

Data Types: `single` | `double`

Velocity accuracy (m/s) — Accuracy of velocity measurement (m/s)

`0.05` (default) | nonnegative real scalar

Accuracy of the velocity measurement of the sensor body in meters per second, specified as a nonnegative real scalar.

Velocity noise is modeled as white process noise with a standard deviation equal to the specified **Velocity accuracy** in meters per second.

Data Types: `single` | `double`

Use acceleration and angular velocity — Use acceleration and angular velocity

off (default) | on

Select this check box to enable the block inputs of acceleration and angular velocity through the **Acceleration** and **AngularVelocity** input ports, respectively. Meanwhile, the block outputs the acceleration and angular velocity measurements through the **Acceleration** and **AngularVelocity**

output ports, respectively. Additionally, selecting this parameter enables you to specify the **Acceleration accuracy** and **Angular velocity accuracy** parameters.

Acceleration accuracy (m/s²) — Accuracy of acceleration measurement (m/s²)

0 (default) | nonnegative real scalar

Accuracy of the acceleration measurement of the sensor body in meters, specified as a nonnegative real scalar.

Acceleration noise is modeled as white process noise with a standard deviation equal to the specified **Acceleration accuracy** in meters per second squared.

Dependencies

To enable this parameter, select **Use acceleration and angular velocity**.

Data Types: `single` | `double`

Angular velocity accuracy (deg/s) — Accuracy of angular velocity measurement (deg/s)

0 (default) | nonnegative real scalar

Accuracy of the angular velocity measurement of the sensor body in meters, specified as a nonnegative real scalar.

Angular velocity noise is modeled as white process noise with a standard deviation equal to the specified **Angular velocity accuracy** in degrees per second.

Dependencies

To enable this parameter, select **Use acceleration and angular velocity**.

Data Types: `single` | `double`

Enable HasGNSSFix port — Enable `HasGNSSFix` input port

off (default) | on

Select this check box to enable the **HasGNSSFix** input port. When the **HasGNSSFix** input is specified as `false`, position measurements drift at a rate specified by the **Position error factor** parameter.

Position error factor — Position error factor (m)

`[0 0 0]` (default) | nonnegative scalar | 1-by-3 real-valued vector

Position error factor without GNSS fix, specified as a scalar or a 1-by-3 real-valued vector. If you specify the parameter as a scalar value, then the block sets the position error factors of all three position components to this value.

When the **HasGNSSFix** input is specified as `false`, the position error grows at a quadratic rate due to constant bias in the accelerometer. The position error for a position component $E(t)$ can be expressed as $E(t) = 1/2\alpha t^2$, where α is the position error factor for the corresponding component and t is the time since the GNSS fix is lost. The computed $E(t)$ values for the x, y, and z components are added to the corresponding position components of the **Position** output.

Dependencies

To enable this parameter, select **Enable HasGNSSFix port**.

Data Types: `double`

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

`Code Generation` (default) | `Interpreted Execution`

- `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.

Version History

Introduced in R2021b

Extended Capabilities

C/C++ Code Generation

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

See Also

`insSensor`

Apps

Flight Log Analyzer

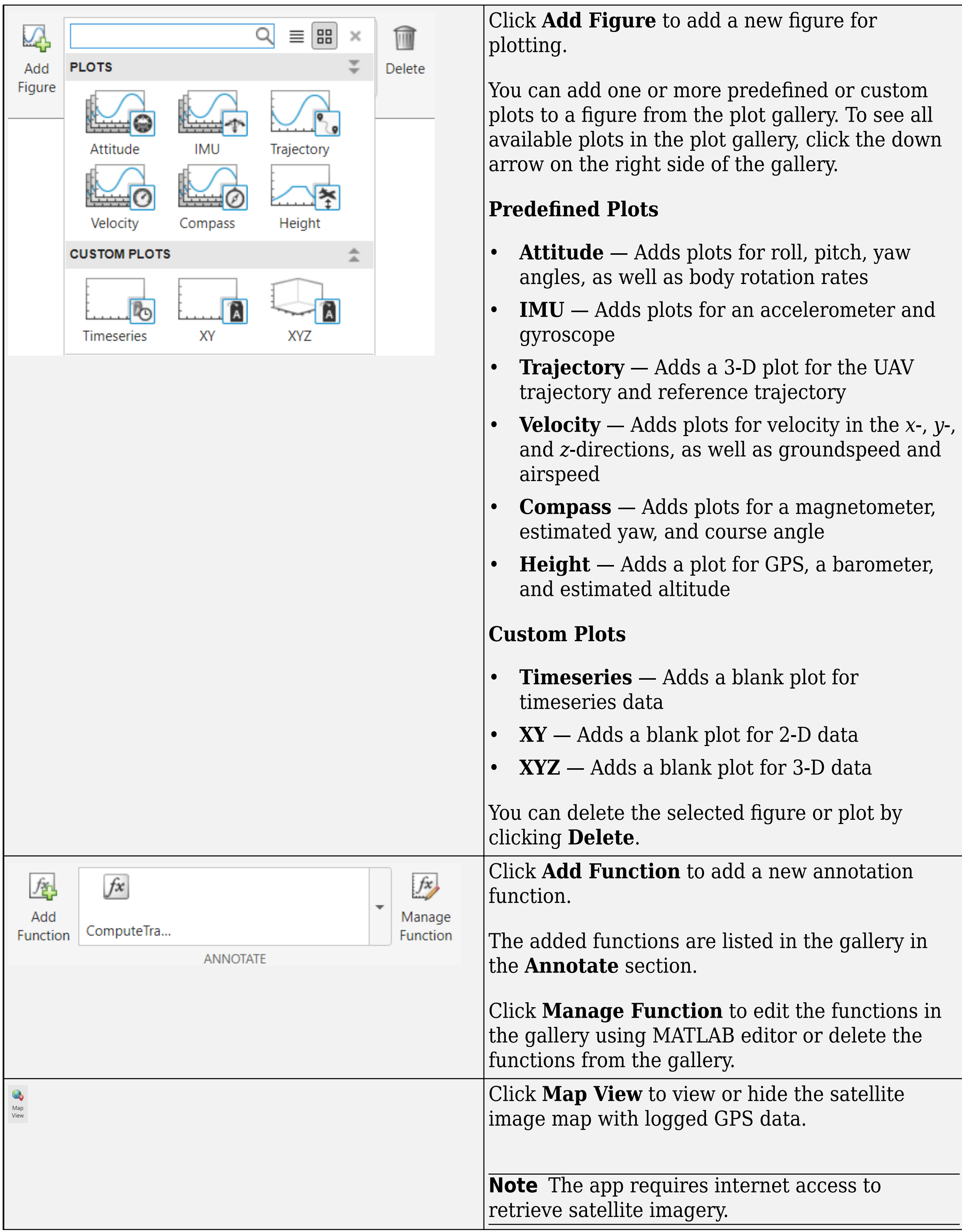
Analyze UAV autopilot flight logs

Description

The **Flight Log Analyzer** app enables you to load and analyze UAV autopilot flight log data, as well as create a customized series of plots.

To use the app:

New Session Open Session Save Session	Click **New Session** to create a new session. You can open saved app sessions by clicking **Open Session**. You can save your progress to a MAT-file (`.mat`) by clicking **Save Session**.
Import From ULOG Load .ulg file or .mat file with ulogreader object From TLOG Load .tlog file or .mat file with mavlinktlog object From Workspace Load a ulogreader or mavlinktlog object, or custom log data and flightLogSignalMapping object from Workspace	To load a ULOG file (`.ulg`) or MAT-file containing a `ulogreader` object, select **Import** > **From ULOG**. To load a TLOG file (`.tlog`) or MAT-file containing a `mavlinktlog` object, select **Import** > **From TLOG**. Select **Import** > **From Workspace** to load a `ulogreader` object, `mavlinktlog` object, or custom log data and a `flightLogSignalMapping` object from the workspace.
Add Custom Signal	Click **Add Custom Signal** to add a custom signal for custom signal mapping.



Click **Add Figure** to add a new figure for plotting.

You can add one or more predefined or custom plots to a figure from the plot gallery. To see all available plots in the plot gallery, click the down arrow on the right side of the gallery.

Predefined Plots

- **Attitude** — Adds plots for roll, pitch, yaw angles, as well as body rotation rates
- **IMU** — Adds plots for an accelerometer and gyroscope
- **Trajectory** — Adds a 3-D plot for the UAV trajectory and reference trajectory
- **Velocity** — Adds plots for velocity in the *x*-, *y*-, and *z*-directions, as well as groundspeed and airspeed
- **Compass** — Adds plots for a magnetometer, estimated yaw, and course angle
- **Height** — Adds a plot for GPS, a barometer, and estimated altitude

Custom Plots

- **Timeseries** — Adds a blank plot for timeseries data
- **XY** — Adds a blank plot for 2-D data
- **XYZ** — Adds a blank plot for 3-D data

You can delete the selected figure or plot by clicking **Delete**.

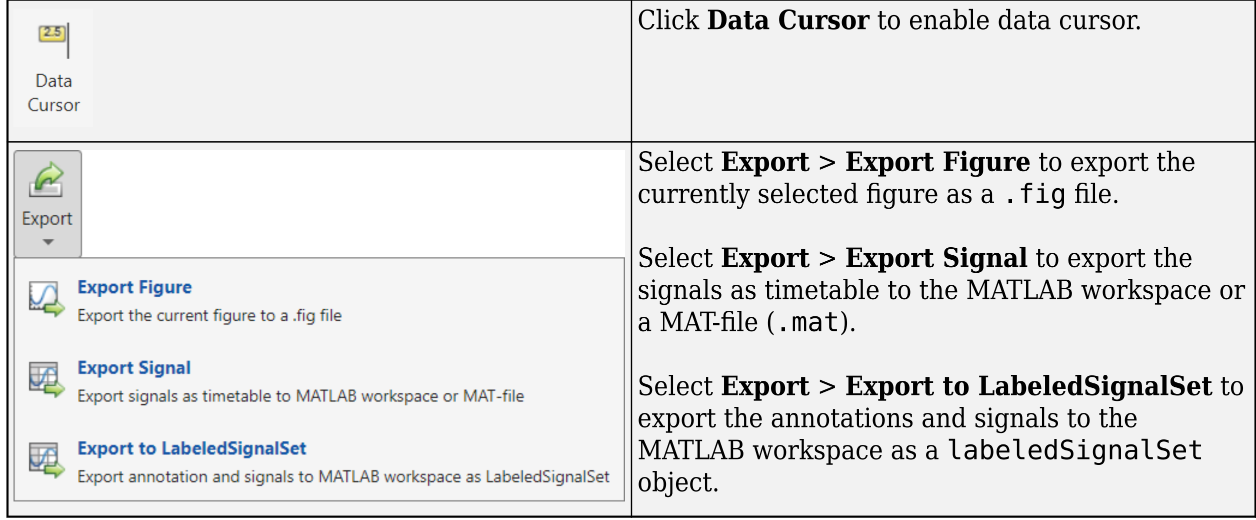
Click **Add Function** to add a new annotation function.

The added functions are listed in the gallery in the **Annotate** section.

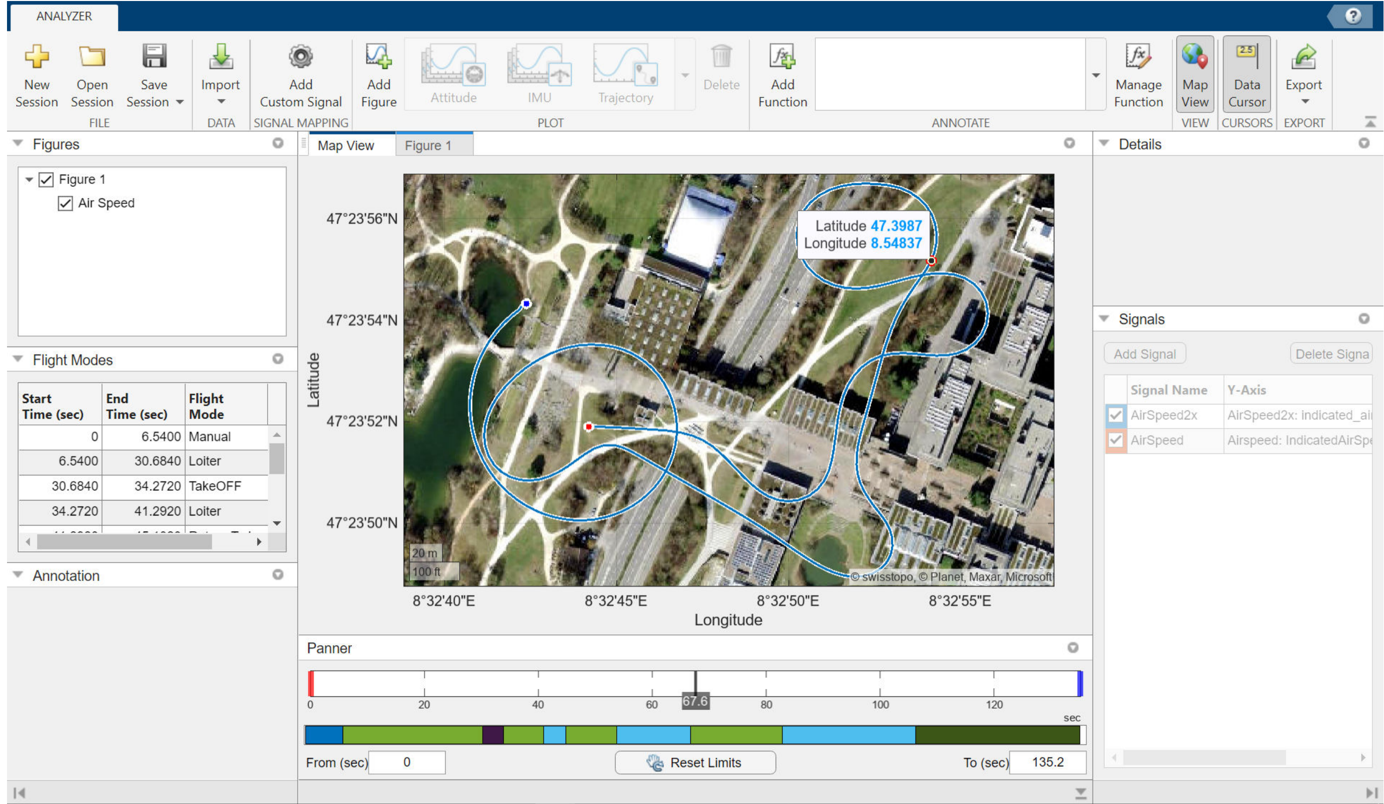
Click **Manage Function** to edit the functions in the gallery using MATLAB editor or delete the functions from the gallery.

Click **Map View** to view or hide the satellite image map with logged GPS data.

Note The app requires internet access to retrieve satellite imagery.



Data Cursor	Click **Data Cursor** to enable data cursor.
Export **Export Figure** Export the current figure to a .fig file **Export Signal** Export signals as timetable to MATLAB workspace or MAT-file **Export to LabeledSignalSet** Export annotation and signals to MATLAB workspace as LabeledSignalSet	Select **Export** > **Export Figure** to export the currently selected figure as a `.fig` file. Select **Export** > **Export Signal** to export the signals as timetable to the MATLAB workspace or a MAT-file (`.mat`). Select **Export** > **Export to LabeledSignalSet** to export the annotations and signals to the MATLAB workspace as a `labeledSignalSet` object.



Open the Flight Log Analyzer App

- MATLAB Toolstrip: On the **Apps** tab, under **Robotics and Autonomous Systems**, click **Flight Log Analyzer**.
- MATLAB command prompt: Enter `flightLogAnalyzer`.

Examples

Analyze Flight Log from ULOG File

Use the **Flight Log Analyzer** app to load and analyze UAV autopilot flight log data from a ULOG file.

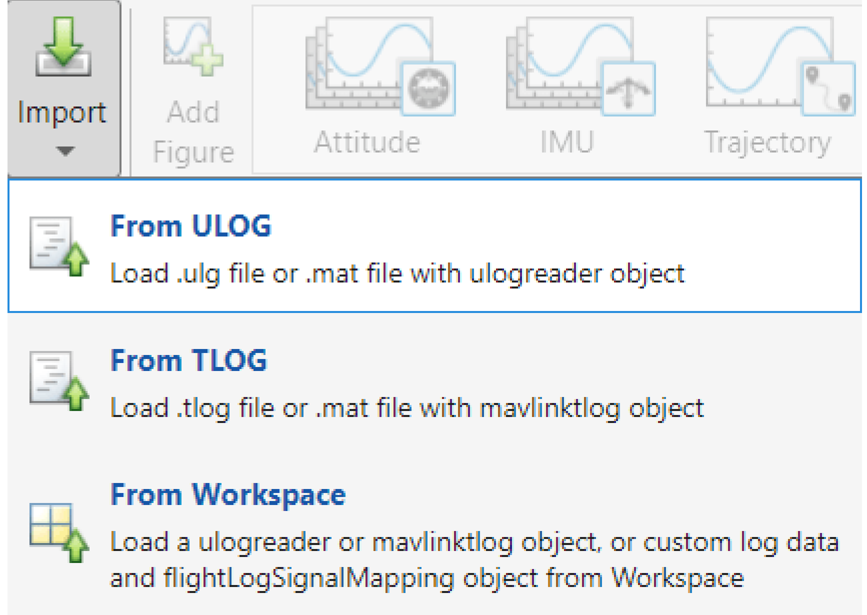
Open Flight Log Analyzer App

In the **Apps** tab, under **Robotics and Autonomous Systems**, click **Flight Log Analyzer**.

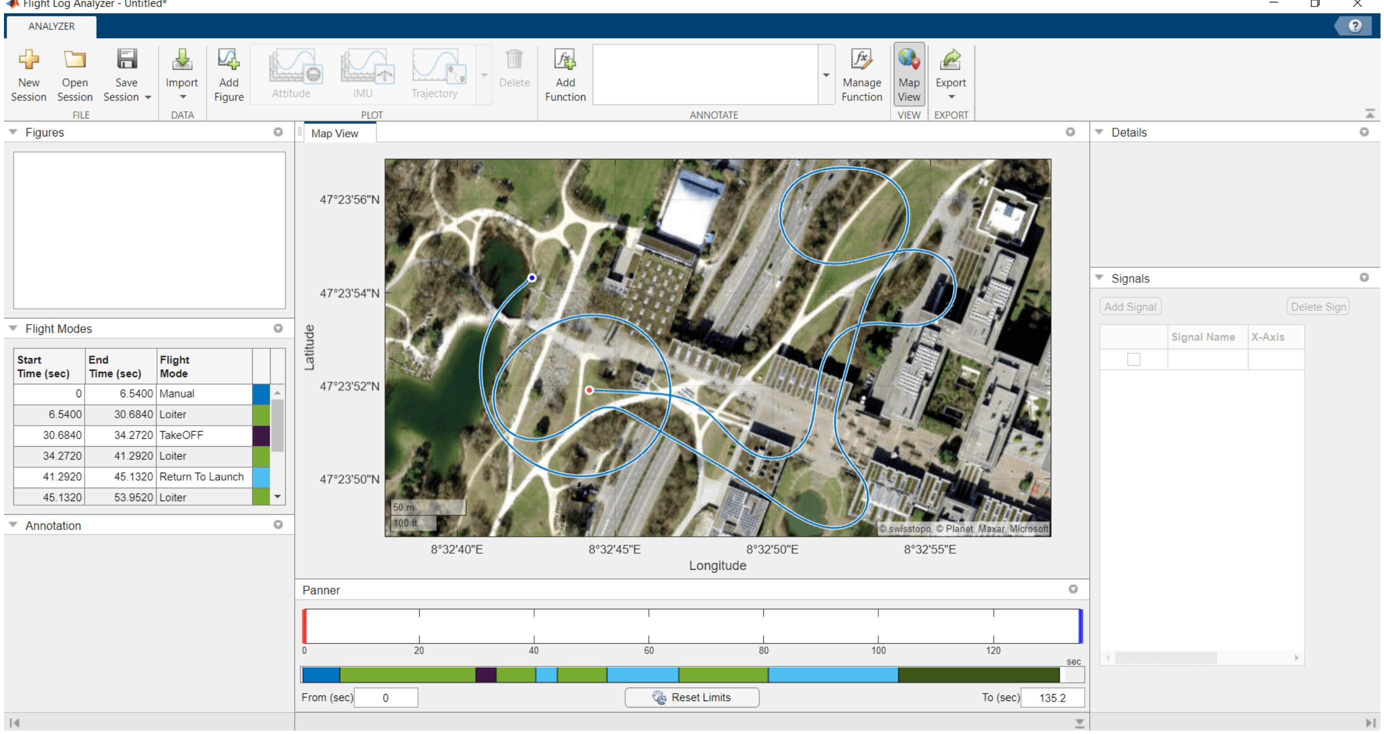
Alternatively, you can use the `flightLogAnalyzer` function from the MATLAB® command prompt.

Import ULOG File

Select **Import** > **From ULOG** to load the UAV flight log data from a ULOG (`flight.ulg`) file.



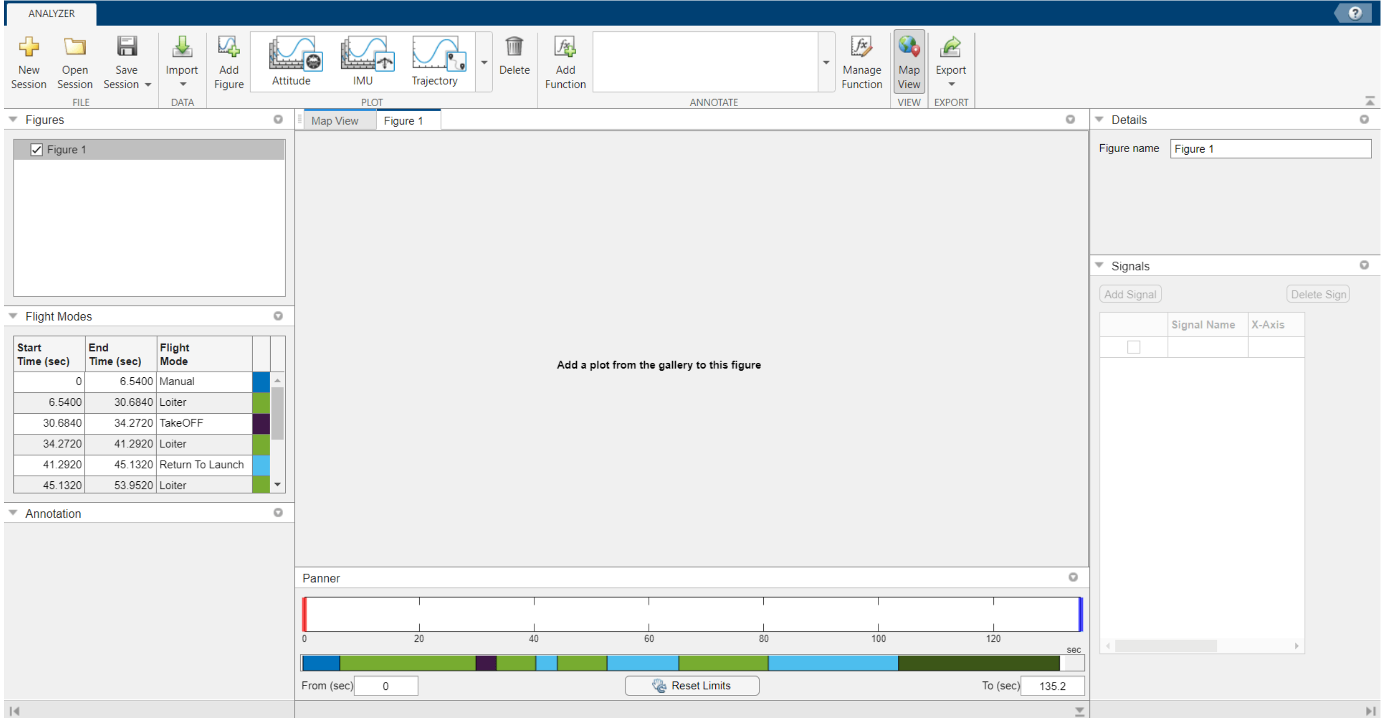
By default, the app displays a satellite map with logged GPS data and the flight modes as a table. The flight modes, along with their corresponding start and end times, are tabulated in the **Flight Modes** pane



Add Figures

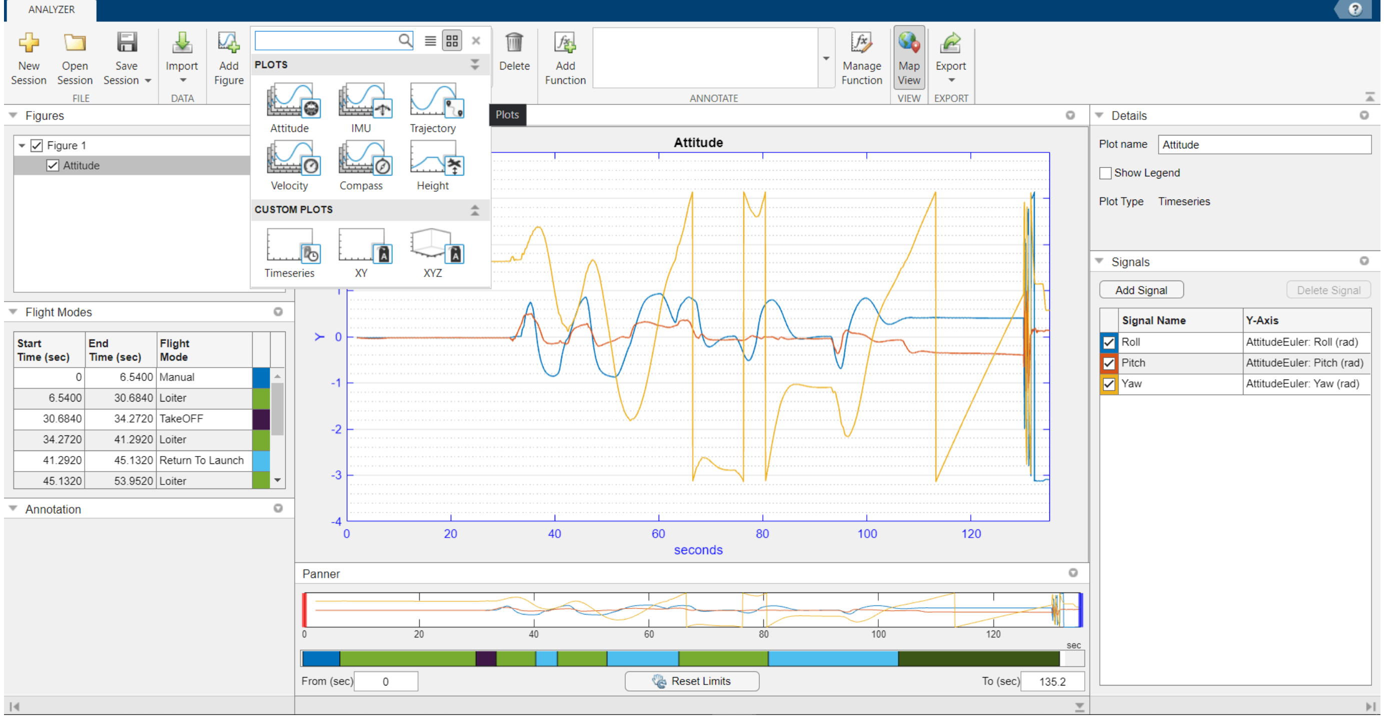
1. To create a new figure for plotting, click **Add Figure**. The app adds an empty figure to the plotting pane. You can continue adding additional figures using this process.
2. The app adds a figure item corresponding to the new figure to the list in the **Figures** pane. Select the check box to the left of the listed figure item to show all plots in the figure. Clear the check box to hide them.
3. To rename a figure, select the associated figure item in the **Figures** pane, click the **Figure name** box in the **Details** pane, and type a new name.
4. To delete a figure, select the figure item in the **Figures** pane and click **Delete** on the app toolstrip. Deleting a figure deletes all plots in the figure.

Creating a figure enables the plot gallery. You can add one or more predefined plots or custom plots to a figure from the plot gallery.



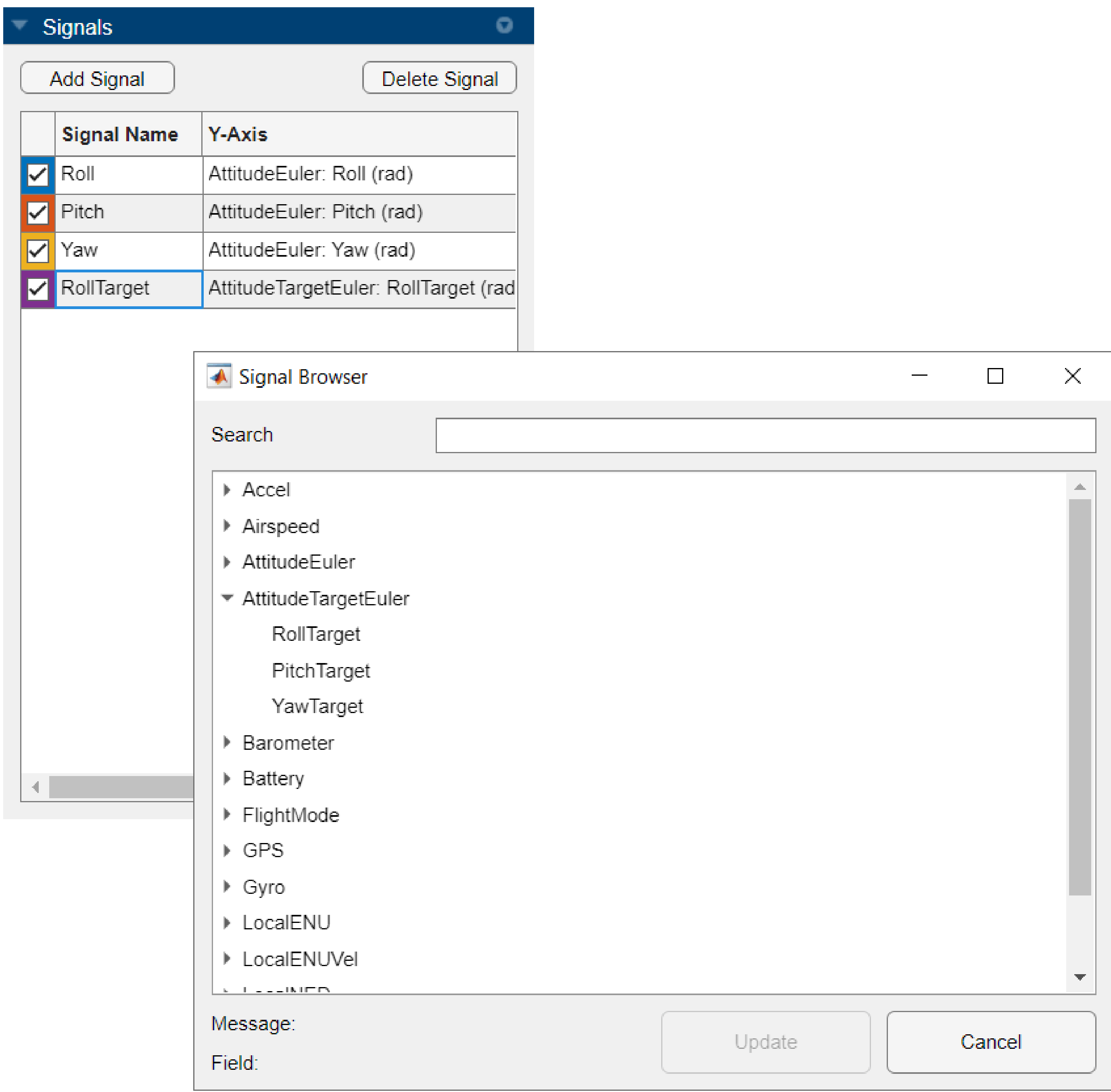
Add Predefined Plot

1. To add a predefined plot to a figure, select one of the six predefined plots from the plot gallery.
2. For example, click **Attitude** to add plots for rotation angles and rotation rates to the figure. You can continue adding additional plots to a figure using this process.
3. The app adds a plot item corresponding to the new plot under the associated figure item in the **Figures** pane. Select the check box to the left of the listed plot item to show the plot in the figure. Clear the check box to hide the plot.
4. To rename a plot, select the associated plot item in the **Figures** pane, click the **Plot name** box in the **Details** pane, and type a new name.
5. Select the **Show Legend** check box in the **Details** pane to show the legend on the plot. Clear the check box to hide the legend.
6. To rename the axis labels, double-click on the predefined labels and type a new name.
7. To delete a plot, select the plot item in the **Figures** pane and click **Delete** on the app toolstrip.



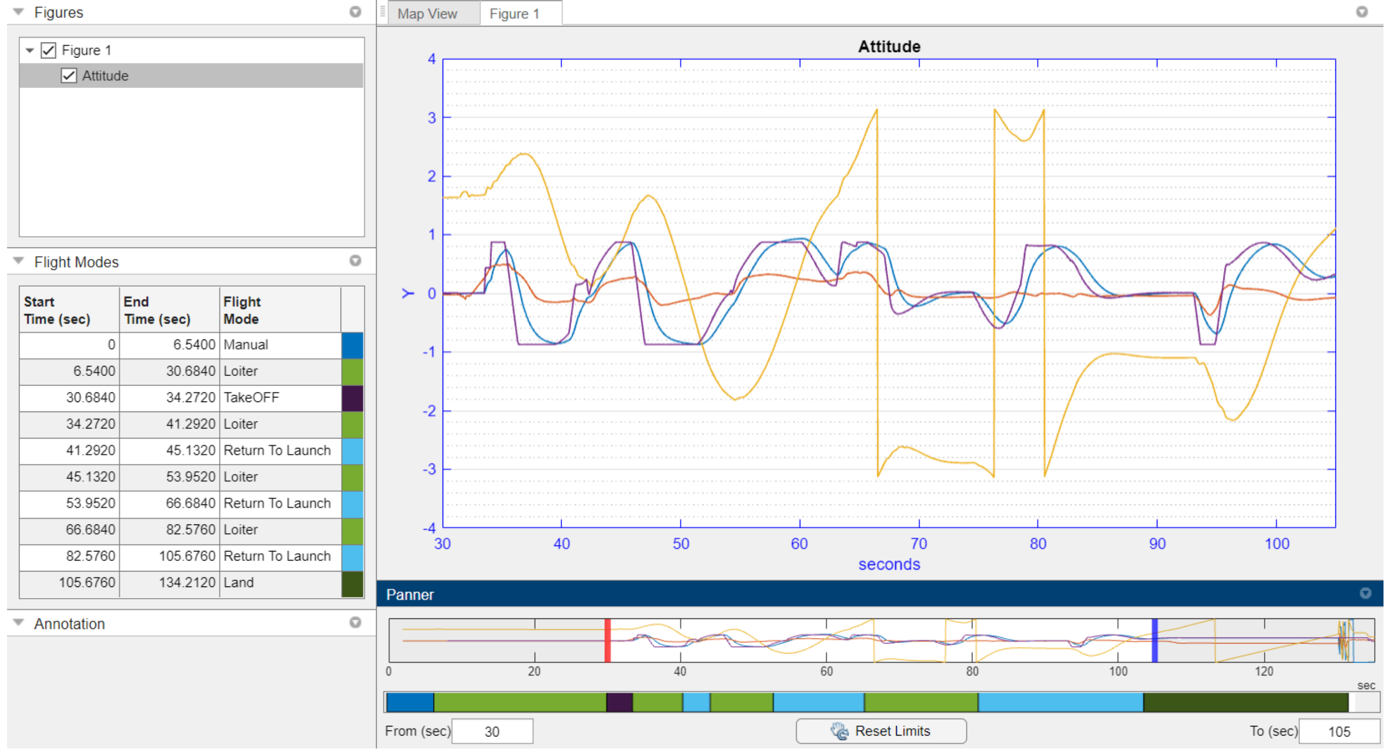
Edit Plot Signals

1. The **Signals** pane displays the signals in the selected plot as a table. The **Signal Name** column contains the names of the signals. The subsequent columns each contain the data associated with that signal for a specific axis.
2. Select the check box in front of a signal item to show that signal in the plot, and clear the check box to hide the signal. The color around the check box is the color of the signal in the plot.
3. To add a new signal to the selected plot, click **Add Signal**.
4. To rename the signal, double-click signal in the **Signal Name** column and type a new name.
5. To add or update the signal data, double-click the **Y-Axis** column of the signal to open the **Signal Browser** window. Choose from available signals.
6. Select one of the signals from the **Signal Browser** window and click **Update**.
7. To delete a signal, select a signal from the **Signals** pane and click **Delete Signal**.



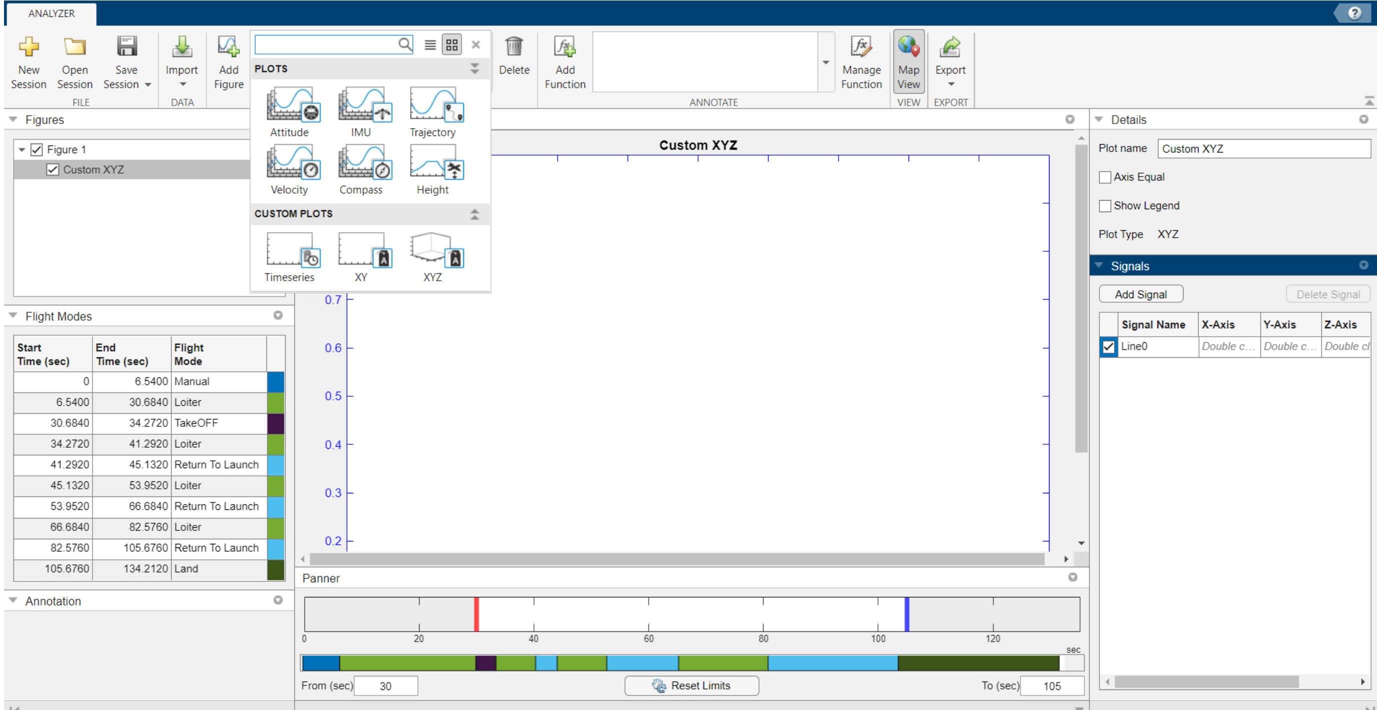
Change Plot Focus Using Panner

1. For timeseries plots, use the **Panner** to focus on data segments in the *x*-axis range. The **Panner** is a strip plot beneath the main plot. To focus on a section of the main plot, drag the red and blue handles to the start and end positions, respectively, of the desired data segment.
2. You can also move the handles by typing new values in the **From (sec)** and **To (sec)** boxes, beneath the strip plot. To reset the handles to their default values, click **Reset Limits**.
3. The color next to each flight mode in the **Flight Modes** pane represents that flight mode in the color bar under the strip plot in the **Panner** pane.



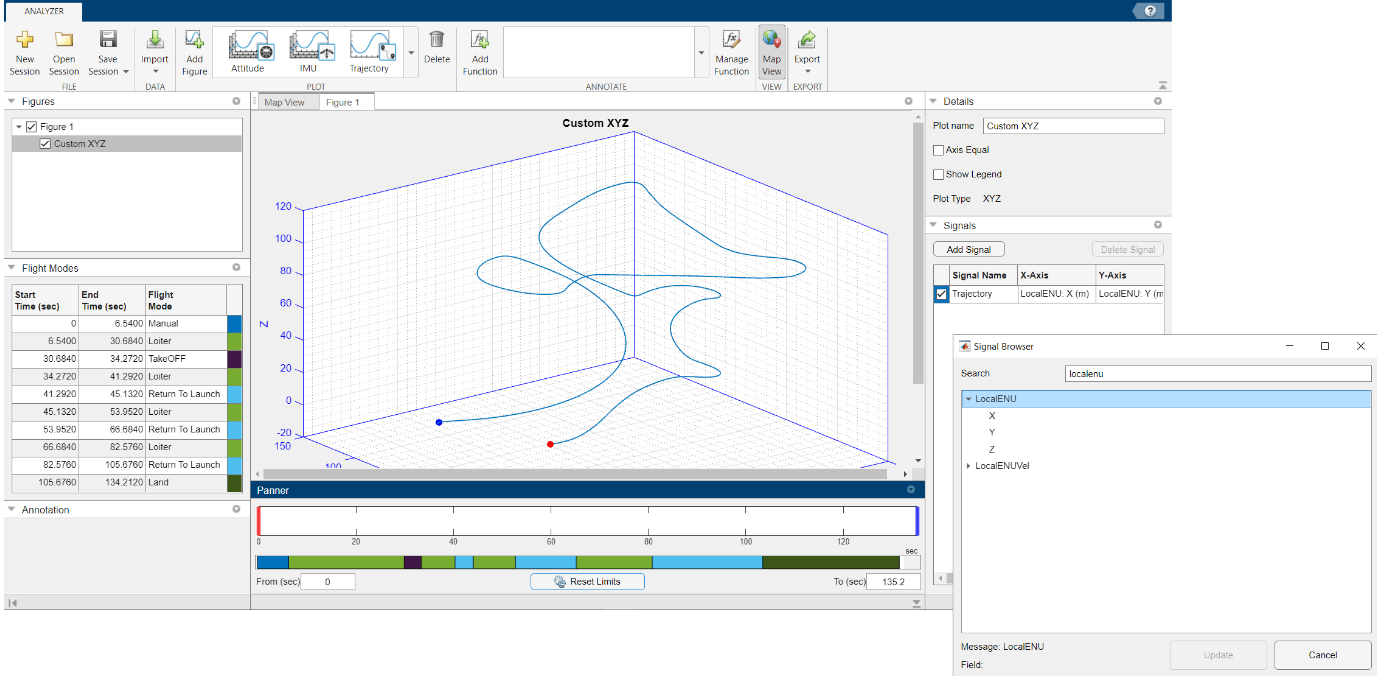
Add Custom Plot

1. To add a custom plot to a figure, select one of the three custom plots from the plot gallery. You can add the new plot to the previously created figure or to a new figure.
2. For example, click **XYZ** to add a blank plot for 3-D data.
3. To add a signal to the plot, click **Add Signal** in the **Signals** pane.
4. To rename the signal, double-click signal in the **Signal Name** column and type a new name.
5. To add signal data to the **X-Axis**, **Y-Axis**, and **Z-Axis** columns, double-click the data field columns of the signal to open the **Signal Browser** window. Choose from the available signals.



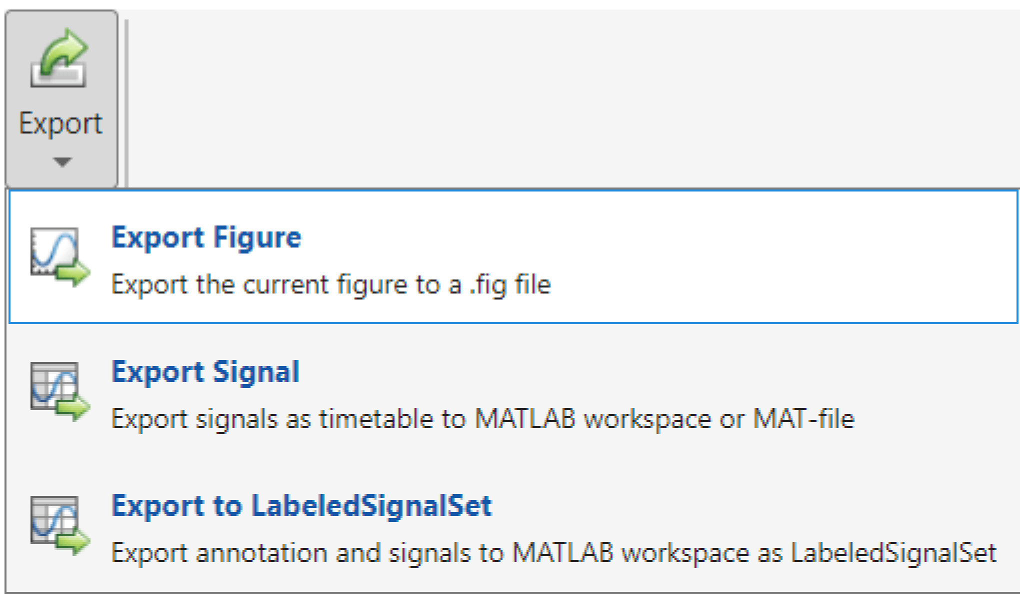
For example, to create a trajectory plot in local east-north-up (ENU) Cartesian coordinates:

1. Double-click the **X-Axis** data field for the desired signal and find the **LocalENU** signal group in the **Signal Browser** pane.
2. Expand the group and select the signal **X**.
3. Click **Update** to update the signal with **X-Axis** data.
4. Repeat these steps to update the **Y-Axis** and **Z-Axis** fields with **Y** and **Z** data, respectively, to create a 3-D trajectory plot.



Export Figure

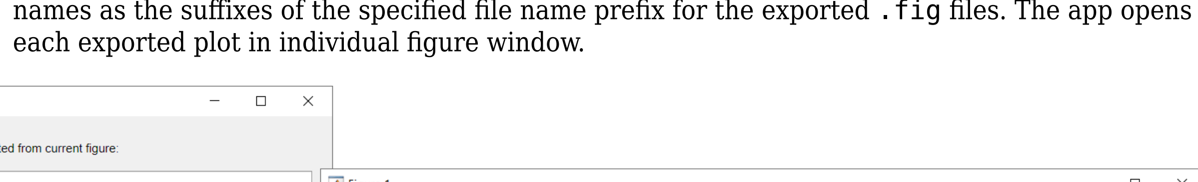
Select **Export** > **Export Figure** to export the current figure to a `.fig` file.

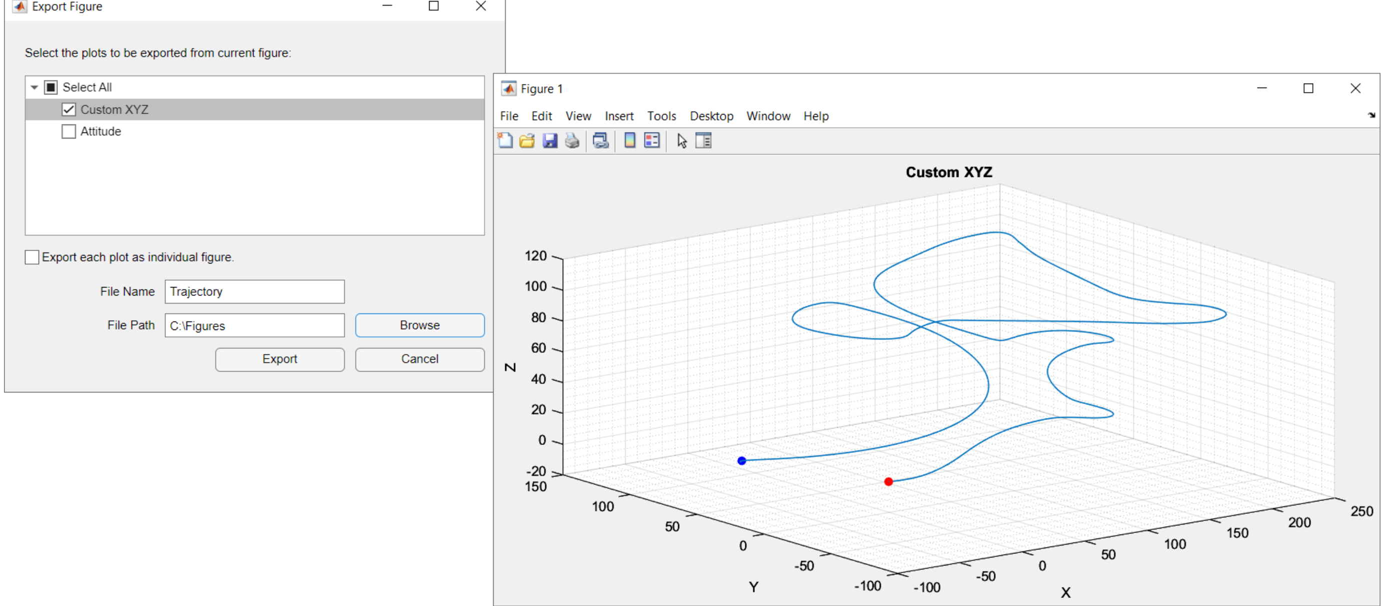


1 Select one or more plots in the current figure to export, specify a file name for the `.fig` file, and click **Browse** to the select the destination folder. Click **Export** to export the selected plots to the `.fig` file. The app opens a figure containing the selected plots in a new figure window.

2 To export each plot as an individual figure, select **Export each plot as individual figure**. Specify the file name prefix for the `.fig` files, and click **Browse** to the select the destination folder. Click **Export** to export the selected plots as individual `.fig` files. The app adds the plot

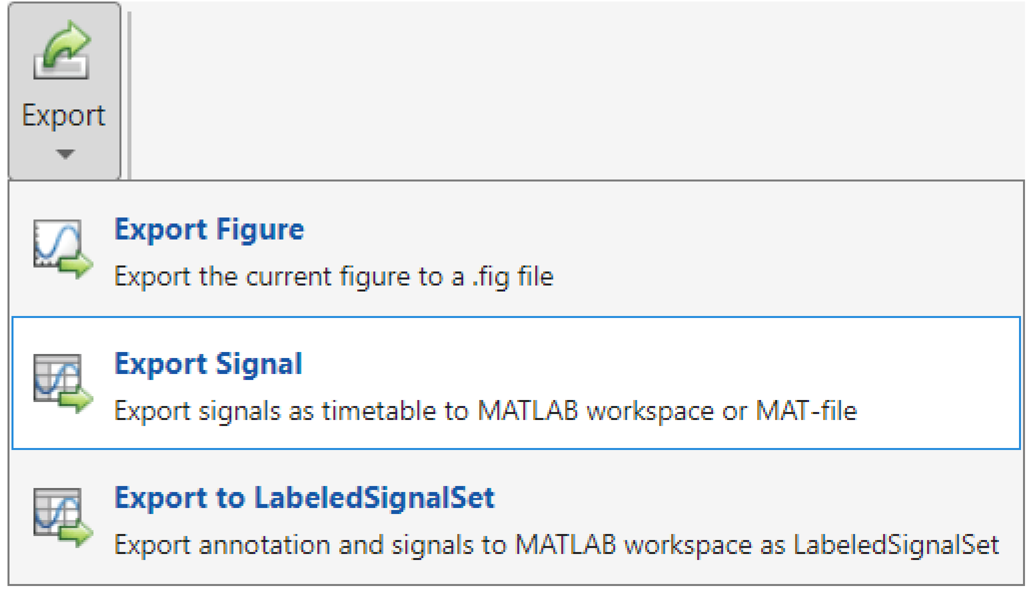
names as the suffixes of the specified file name prefix for the exported `.fig` files. The app opens each exported plot in individual figure window.



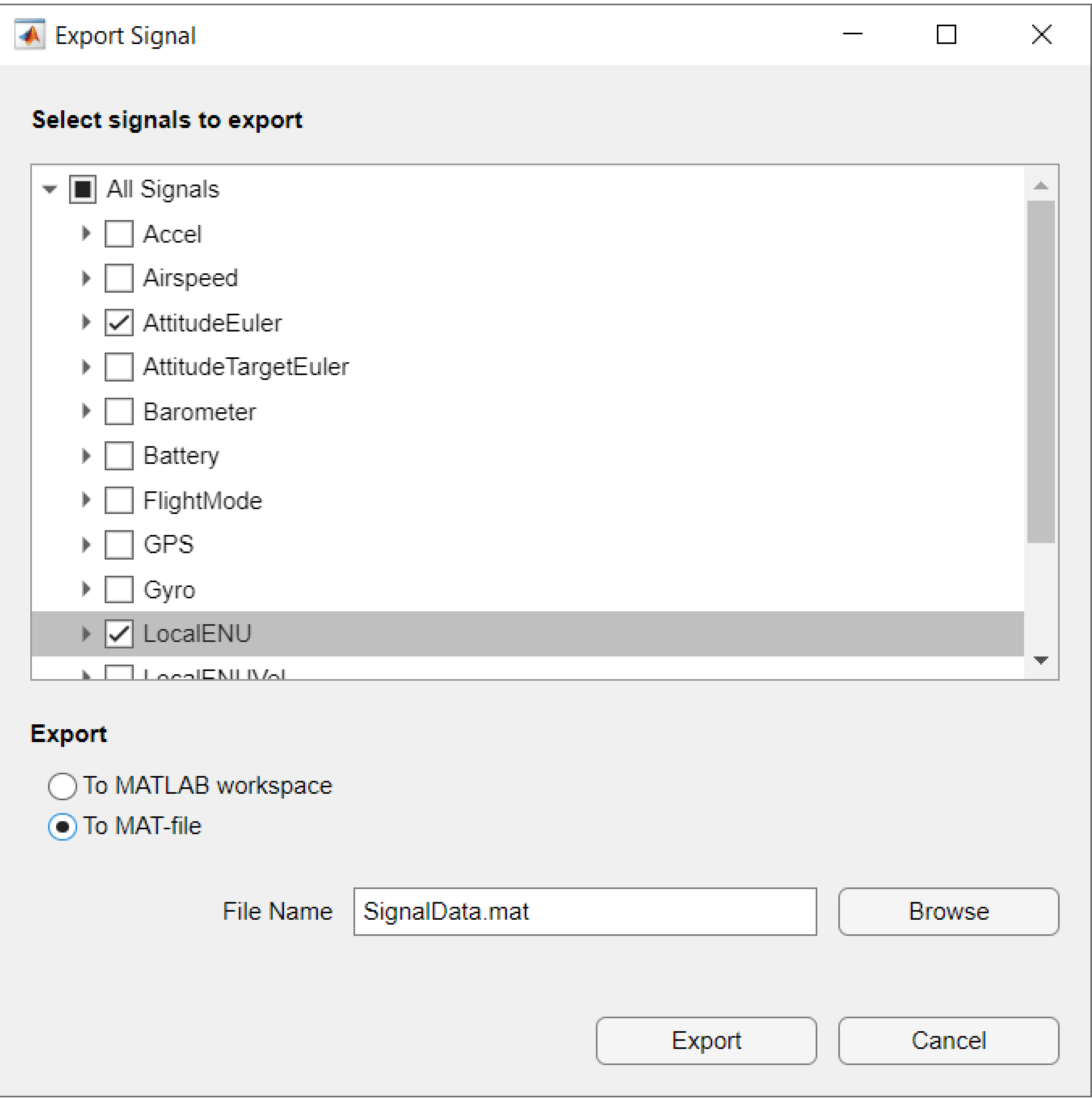


Export Signal

Select **Export** > **Export Signal** to export the signals as a timetable to the MATLAB workspace or a MAT file (`.mat`).



1. Select the signals to export. To export them to a MAT file, select **To MAT-file** and specify a file name for the MAT file. To select a destination folder for the MAT file, click **Browse** and navigate to the folder to which you want to export.
2. To export the signals to the MATLAB workspace, select **To MATLAB workspace** and specify a name for the output workspace variable.



Save and Open Sessions

You can save the **Flight Log Analyzer** app session by clicking **Save Session**. The app writes the current state of the app to a `.mat` file that you can load by clicking **Open Session**.

Annotate Flight Log Signals with Custom Function

Open Flight Log Analyzer App

In the **Apps** tab, under **Robotics and Autonomous Systems,** click **Flight Log Analyzer**.

Alternatively, you can use the `flightLogAnalyzer` function from the MATLAB® command prompt.

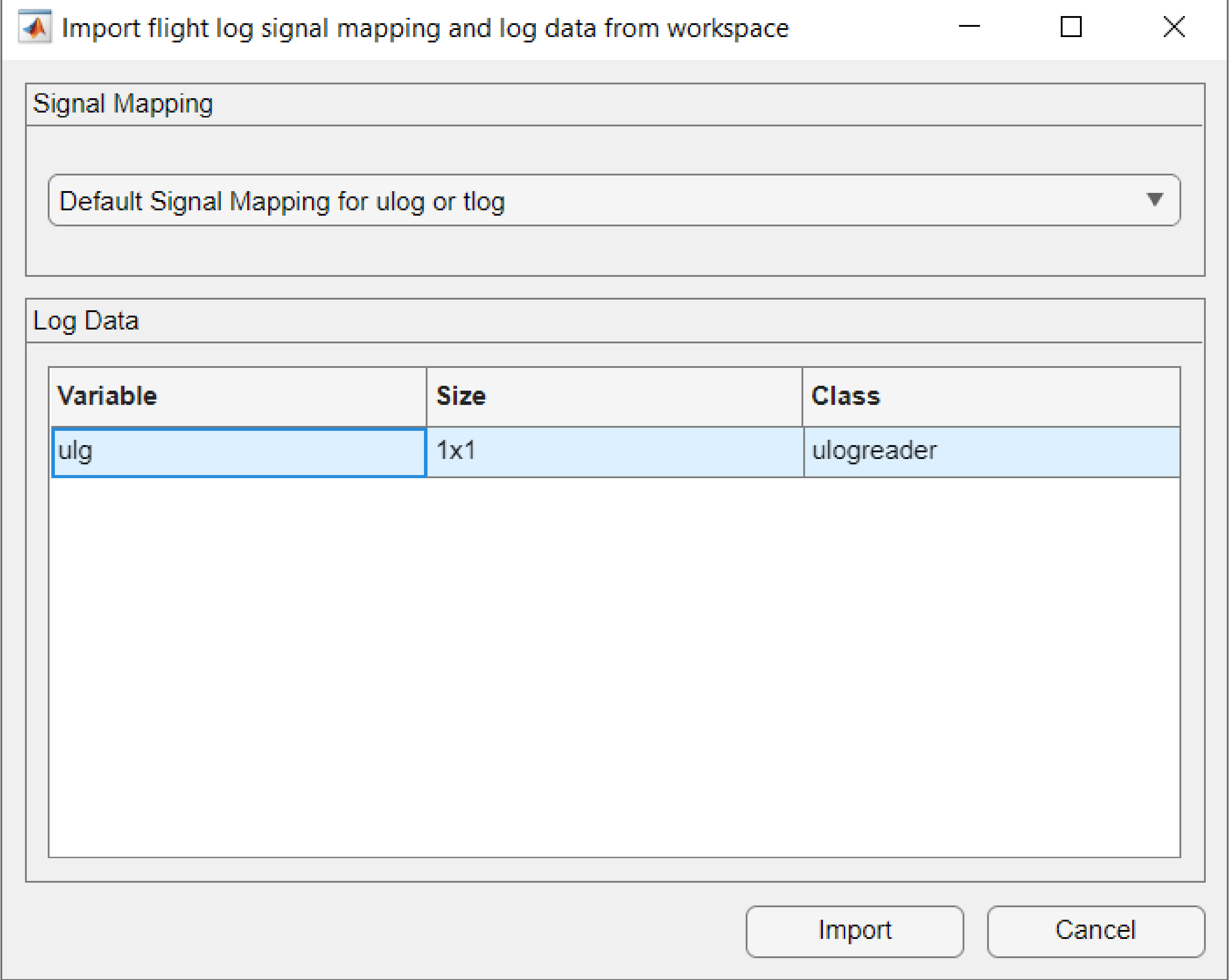
Import ULOG File

Load the ULOG file.

```
ulg = ulogreader("flight.ulg")
```

```
ulg = 
  ulogreader with properties:

            FileName: "flight.ulg"
           StartTime: 00:00:00.176000
             EndTime: 00:02:15.224000
     AvailableTopics: [51x5 table]
    DropoutIntervals: [0x2 duration]
```

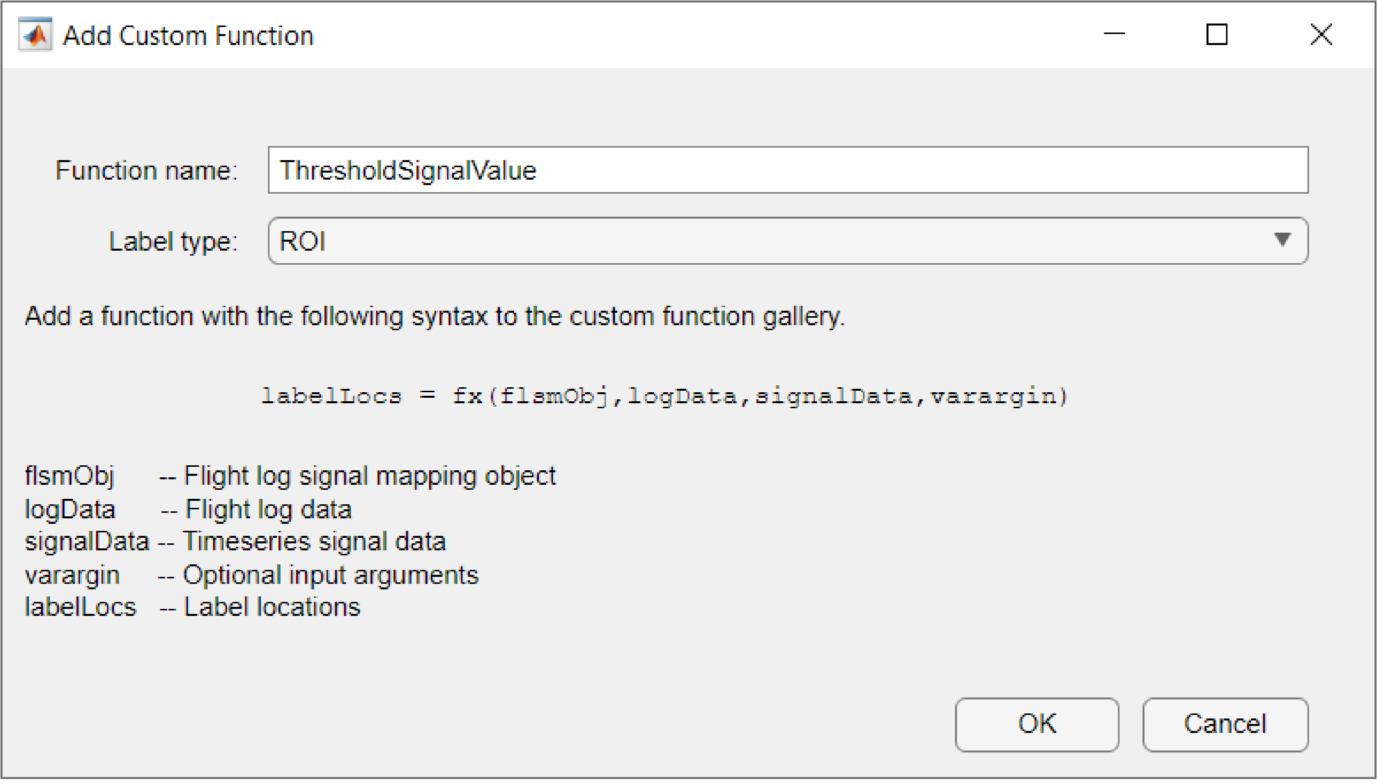
On the **Flight Log Analyzer** app toolstrip, select **Import > From Workspace**. In the **Log Data** section of the **Import flight log signal mapping and log data from workspace** dialog box, select the `ulogreader` object `ulg` and click **Import**.



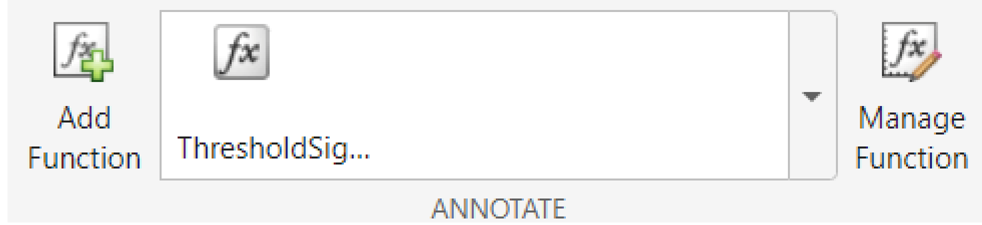
Add Annotation Function

To add a custom function to the **Flight Log Analyzer** app:

1. In the **Annotate** section of the app toolstrip, click **Add Function**.
2. In the dialog box, specify the **Function name** as `ThresholdSignalValue`.
3. Select the **Label type** as **ROI**.
4. Click on **OK**.



The `ThresholdSignalValue` function is added to the gallery in the **Annotate** section, since the function is already provided with this example and it is in MATLAB path.



If the function does not exist in the MATLAB path, the app suggests to create a new function and opens a function template in the MATLAB editor. If the function is at a different location, the location must be added to the MATLAB path.

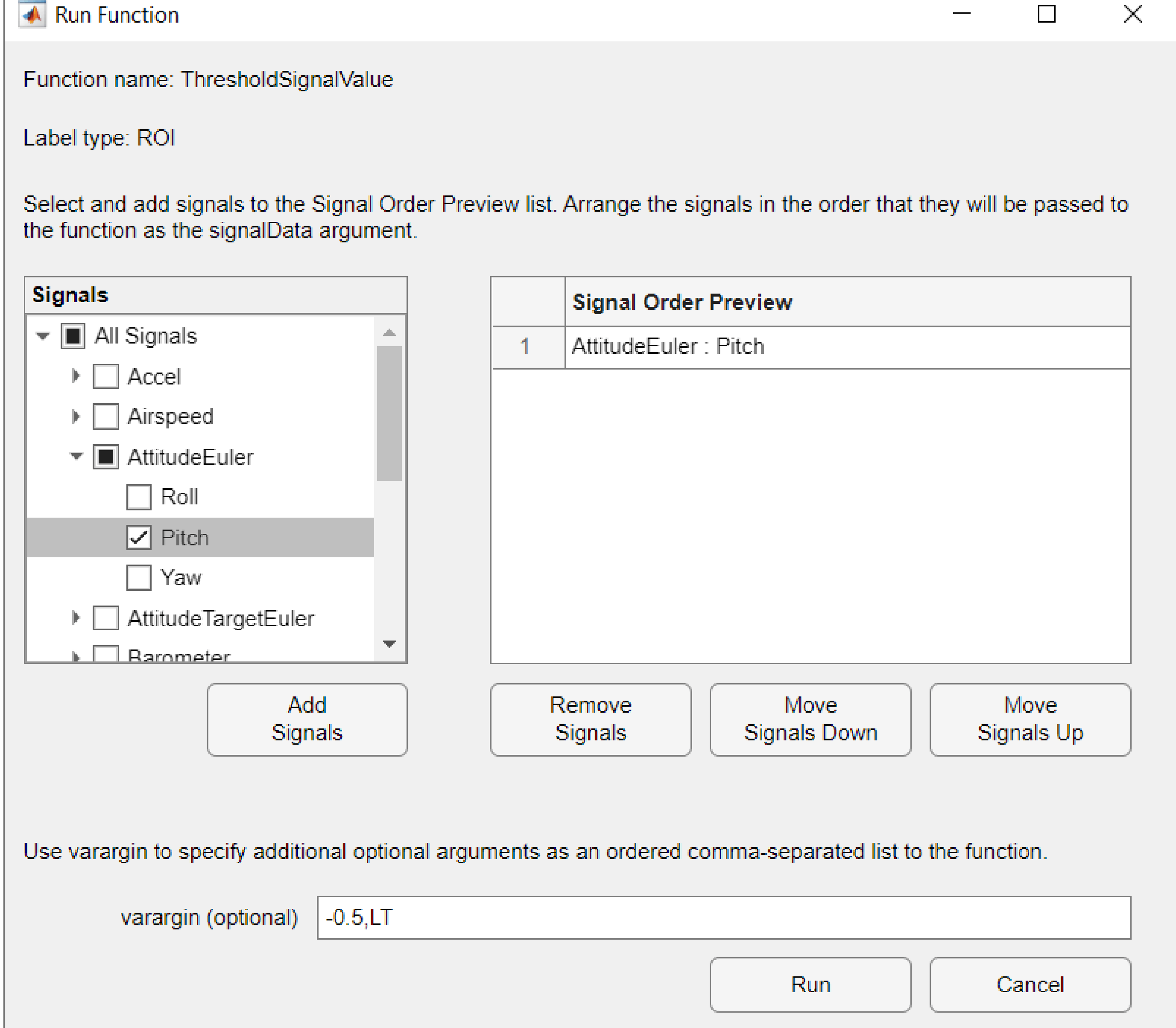
The function can access the `flightLogSignalMapping` object, complete flight log data, selected signals, and additional optional arguments from the app.

Run Annotation Function

Run the `ThresholdSignalValue` annotation function to find the regions with signal value less than the threshold value.

1. In the **Annotate** gallery, click **ThresholdSignalValue.**
2. In the dialog box, select the required signals from the **Signals** panel.
3. Click the arrow next to `AttitudeEuler` and select `Pitch`.

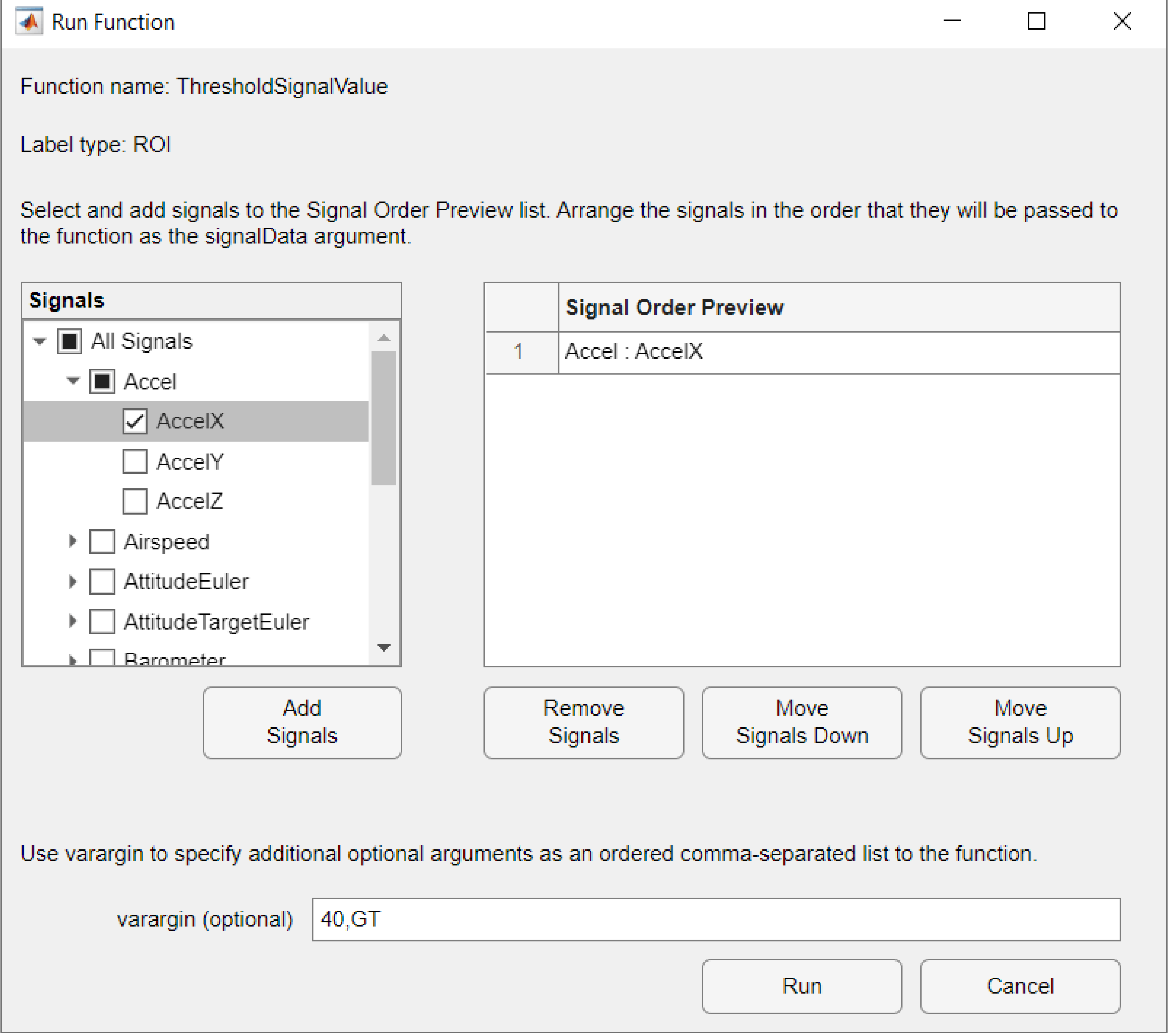
4. Click **Add Signals**.
5. The added signals are listed on the **Signal Order Preview** panel.
6. Specify the threshold value and a string to denote signal value is less than threshold as `-0.5,LT` in **varargin**.
7. Click **Run**.



Run the `ThresholdSignalValue` annotation function again to find the regions with signal value greater than the threshold value.

1. In the **Annotate** gallery, click **ThresholdSignalValue.**
2. In the dialog box, select the required signals from the **Signals** panel.

3 Click the arrow next to `Accel` and select `AccelX`.
4 Click **Add Signals**.
5 The added signals are listed on the **Signal Order Preview** panel.
6 Specify the threshold value and a string to denote signal value is greater than threshold as `40,GT` in **varargin**.
7 Click **Run**.



Create Figures and Plots

In the **Plot** section of the app toolstrip, click **Add Figure** to add an empty figure to the plotting pane.

Create predefined **Attitude** plot.

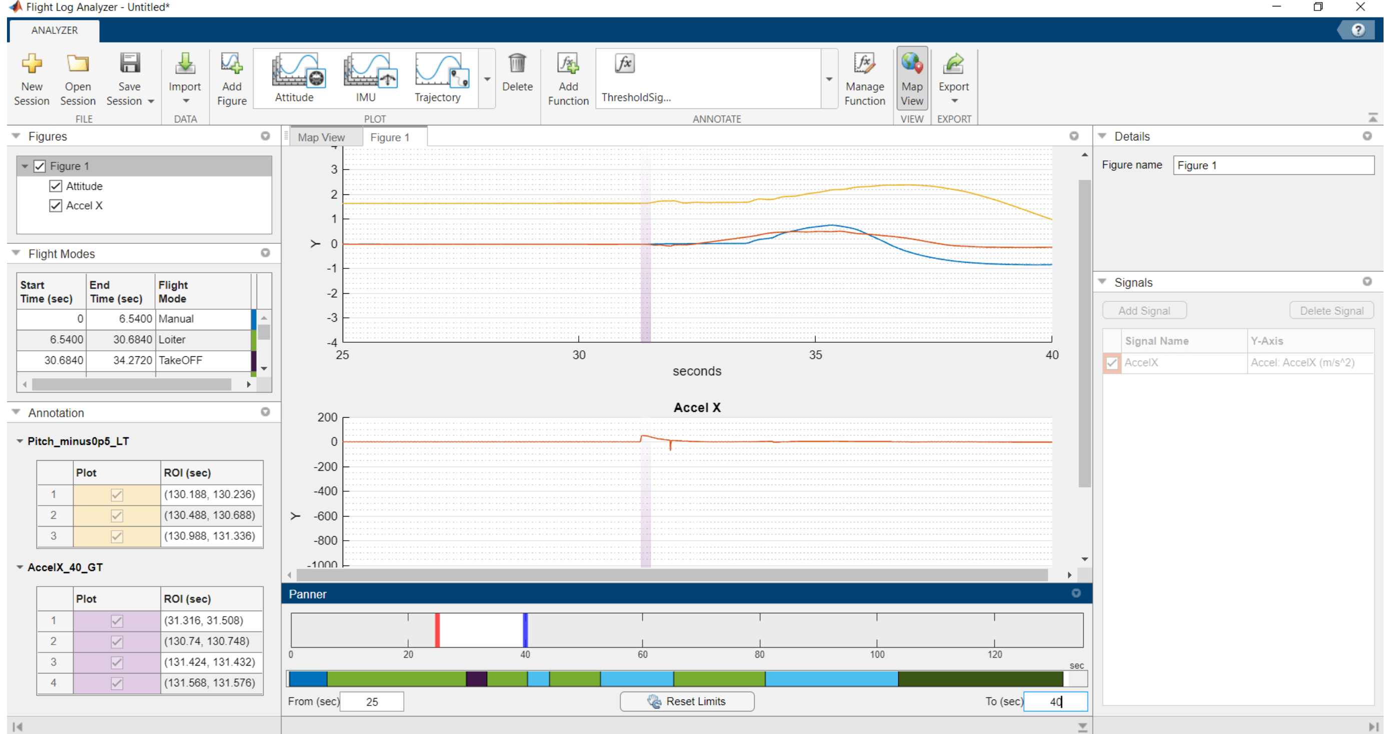
- In the **Plots** section of the plot gallery, click **Attitude**.

Create custom **Timeseries** plot.

1. In the **Custom Plots** section of the plot gallery, click **Timeseries**.
2. In the **Details** pane, rename the plot as `Accel X`.
3. In the **Signals** pane, click **Add Signal**.
4. Double-click the **Y-Axis** column of the signal and, in the **Signal Browser** window, type `AccelX` in the **Search** box, and then click the arrow next to `Accel` and select `AccelX`. Then, click **Update**.
5. Rename the signal to `AccelX`. To rename a signal, double-click its entry in the **Signal Name** column and type the new name.

Annotate Plots

1. In the **Figures** pane, select the plot on which you want to see the detected region of interest.
2. In the **Annotation** pane, select the check boxes to highlight the regions of interest on the plot.
3. In the **Panner** pane, drag the red and blue handles to the **Start Time** and **End Time**, respectively, to focus on a region of interest. Alternatively, you can type the **Start Time** and **End Time** values in the **From (sec)** and **To (sec)** boxes, beneath the strip plot.
4. To rename the default annotation label, right-click on the label and select **Rename Label** and type the new label name.



Custom Signal Mapping in Flight Log Analyzer App

Open Flight Log Analyzer App

Use the `flightLogAnalyzer` function to launch the app.

```
flightLogAnalyzer
```

Alternatively, to launch the app from the **Apps** tab in the MATLAB® Toolstrip, under **Robotics and Autonomous Systems**, click **Flight Log Analyzer**.

Import ULOG File

Select **Import** > **From ULOG** to load the UAV flight log data from a ULOG file, `flight.ulg`.

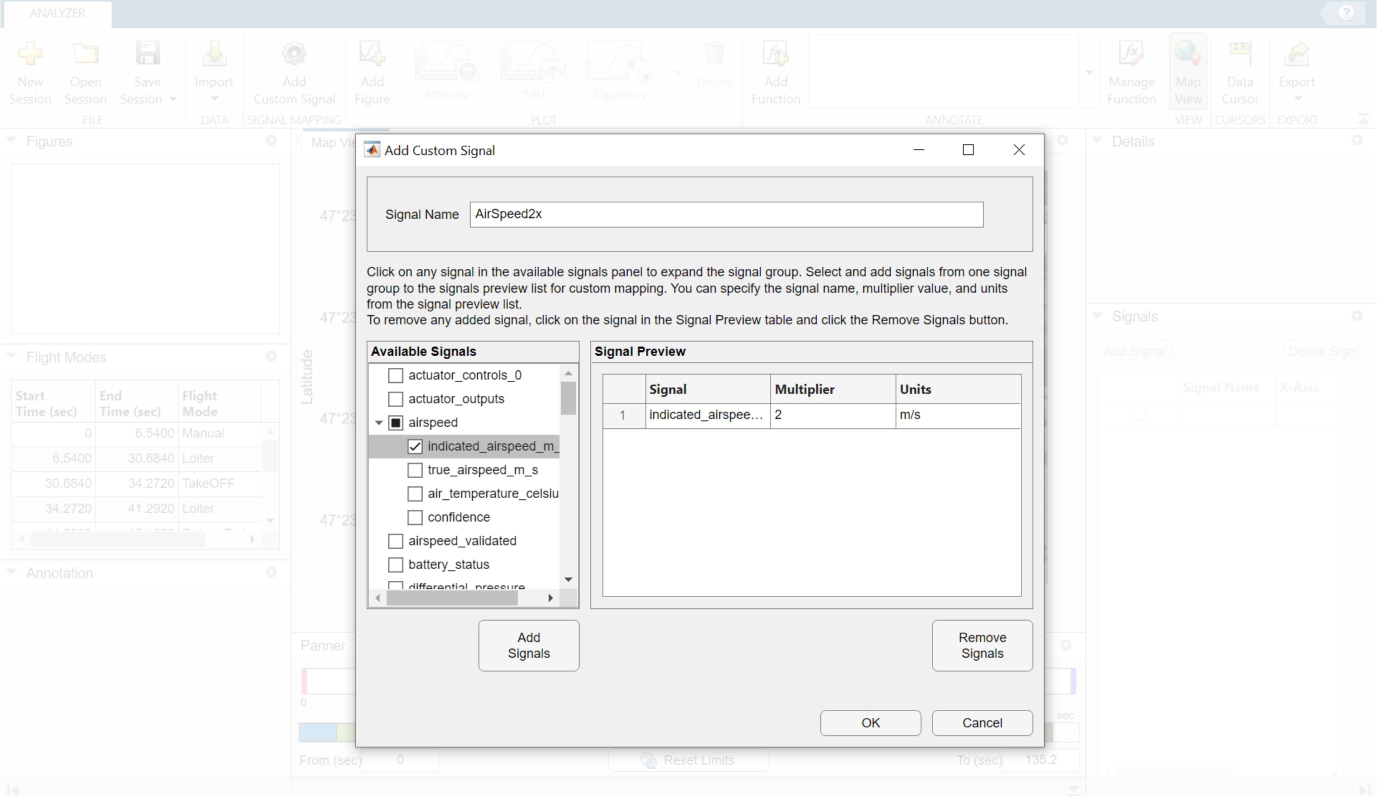
Add Custom Signal

To add a custom signal for custom signal mapping:

1. In the **Signal Mapping** section of the app toolstrip, click **Add Custom Signal** to open the Add Custom Signal window.
2. Click any signal in the **Available Signals** pane to expand the signal group.
3. Select signals from one signal group, and click **Add Signal**.
4. The added signals are listed in the **Signal Preview** pane.
5. Specify a signal name in **Signal Name**.
6. You can also specify the **Multiplier** value and the **Units** for the signals in the **Signal Preview** pane.

7 To remove added signals, click the signals you want to remove in the **Signal Preview** pane and select **Remove Signals**.

8 Click **OK**.



Create Figures and Plots

In the **Plot** section of the app toolstrip, click **Add Figure** to add an empty figure to the plotting pane.

Create a custom **Timeseries** plot by following these steps.

1 In the **Custom Plots** section of the plot gallery, click **Timeseries**.

2 In the **Details** pane, rename the plot to `Air Speed`.

3 In the **Signals** pane, click **Add Signal**.

4 Double-click the **Y-Axis** column of the signal and, in the Signal Browser window, type `airspeed` in the **Search** box, and then click the arrow next to `AirSpeed2x` and select `indicated_airspeed_m_s`. Then, click **Update**.

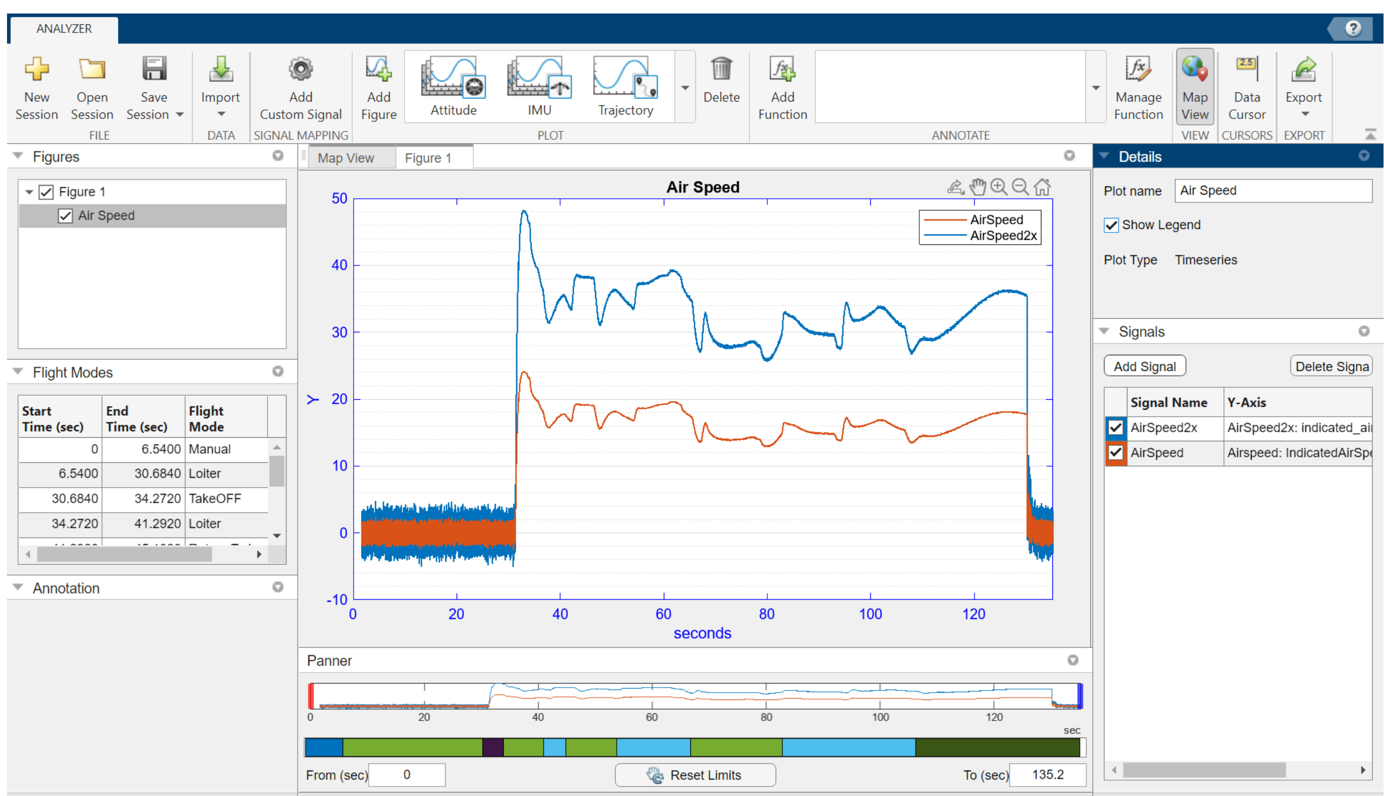
5 Rename the signal to `AirSpeed2x`. To rename a signal, double-click its entry in the **Signal Name** column and type the new name.

6 Add another signal from the **Signals** pane, then click **Add Signal**.

7 Double-click the **Y-Axis** column of the signal and, in the Signal Browser window, type `airspeed` in the **Search** box, and then click the arrow next to `Airspeed` and select `IndicatedAirSpeed`. Then, click **Update**.

8 Rename the signal to `AirSpeed`.

9 Select **Show Legend** in the **Details** pane to show the legend on the plot.



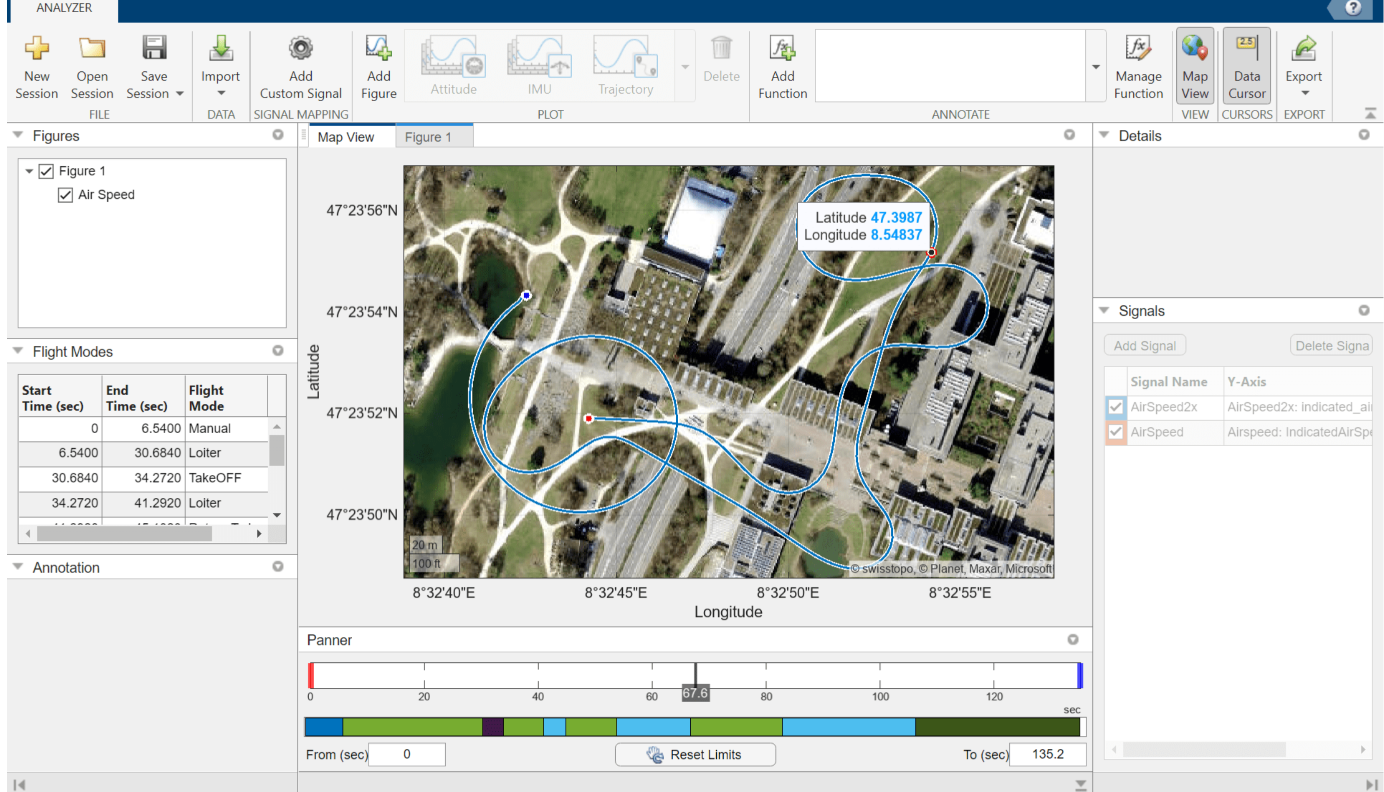
Enable Data Cursor

You can explore plot data interactively using the **Data Cursor**.

1 In the **Cursors** section of the app toolstrip, click **Data Cursor** to add a handle to the **Panner**.
2 You can drag the handle or the data tips to the plot. Data tips are small text boxes that display information about individual data points.

You can use the data cursor with **Timeseries** plots, and on **Map View**.

ANALYZER
New Session
Open Session
Save Session
FILE
Import
DATA
Add Custom Signal
SIGNAL MAPPING
Add Figure
Attitude
IMU
Trajectory
PLOT
Delete
Add Function
ANNOTATE
Manage Function
Map View
VIEW
Data Cursor
CURSORS
Export
EXPORT
Figures
Figure 1
Air Speed
Flight Modes
Start Time (sec)
End Time (sec)
Flight Mode
0
6.5400
Manual
6.5400
30.6840
Loiter
30.6840
34.2720
TakeOFF
34.2720
41.2920
Loiter
Annotation
Map View
Figure 1
Air Speed
AirSpeed
AirSpeed2x
30.4697
15.2349
67.60
seconds
Panner
67.6
sec
From (sec)
Reset Limits
To (sec)
135.2
Details
Plot name
Air Speed
Show Legend
Plot Type
Timeseries
Signals
Add Signal
Delete Signa
Signal Name
Y-Axis
AirSpeed2x
AirSpeed2x: indicated_ai
AirSpeed
Airspeed: IndicatedAirSpe



- “Analyze UAV Autopilot Flight Log Using Flight Log Analyzer”
- “Add Custom Functions in Flight Log Analyzer App to Detect Region of Interest for Analysis”

Programmatic Use

`flightLogAnalyzer` opens the **Flight Log Analyzer** app, which enables you to analyze UAV autopilot flight logs.

More About

Flight Modes

This table describes the types of flight modes:

Flight mode	Description
Manual	Manual remote control mode
TakeOFF	Take off from the ground and travel towards the specified position
Orbit	Orbit in the specified turn direction for the specified number of turns along the circumference of a circle with a specified radius and a center at the specified position

Flight mode	Description
Loiter	A fixed-wing UAV circle around the specified position at the specified radius
Hold	Hold at the current position A fixed-wing UAV loiters around the current position, and a multirotor UAV hovers at the current position
Return To Launch	Return to the launch position
Land	Land at the specified position

Version History

Introduced in R2020b

R2023a: Custom signal mapping and data cursor

The **Flight Log Analyzer** app has been updated with these enhancements:

- The **Flight Log Analyzer** app now enables you to add additional log signals from files of the supported formats (`.ulog`, and `.tlog`) from within the app. Custom signal mapping enables you to map any signal available in the log data within the app. You can specify signal names, scale the signal values, and provide signal units. Once you have successfully mapped a custom signal, you can visualize the signal using custom plots. Click **Add Custom Signal** to open the Add Custom Signal dialog box.
- You can now explore plot data interactively using the **Data Cursor**. In the app toolstrip, select **Data Cursor** to add a handle to the **Panner**, that you can drag, and data tips to the plot, which are small text boxes that display information about individual data points. You can use the data cursor with **Timeseries** plots, and on **Map View**.

R2022b: Custom annotation support

The **Flight Log Analyzer** app has been updated with these enhancements:

- The **Flight Log Analyzer** app now supports custom annotations. Custom annotation enables you to write custom functions to label plots with region of interest (ROI) and point labels. The **Function Gallery** shows all of the custom functions you add. You can use the **Annotation** pane to interact with the labels and display them on the plots. You can export the annotations and signals to workspace as `labeledSignalSet` objects.
- Improved filtering of the `ulogreader` object, the `mavlinktlog` object, and a `flightLogSignalMapping` object with custom log data from other variables when importing them from the workspace.
- The **Signal Browser** pane now opens as a window when you double-click the data field of the desired signal in the **Signals** pane.

R2022b: Panner behavior and Details pane changes

Behavior changed in R2022b

The **Flight Log Analyzer** app has been updated with these changes:

- The **Left** and **Right** boxes in the **Panner** pane have been renamed to **From (sec)** and **To (sec)**, respectively. The boxes are now enabled only when a log file has been loaded into the app.
- The **Name** box in the **Details** pane is now either **Plot name** or **Figure name** depending on the item selected in the **Figures** pane.
  - **Plot name** — Select a plot item in the **Figures** pane.
  - **Figure name** — Select a figure item in the **Figures** pane.

R2021b: Interactive flight modes and export figure

The **Flight Log Analyzer** app has been updated with these enhancements:

- The flight modes in the **Flight Modes** pane are now interactive. You can select one or more consecutive flight modes. The **Panner** limits values and adjusts handles accordingly.
- The **Signals** pane now displays the units of the signals.
- The axes labels of the plots are now customizable.
- You can now export multiple plots to a single figure or separate figures.

R2021a: Export signals as timetables to the MATLAB workspace or a MAT-file

The **Flight Log Analyzer** app now enables you to export signals as timetables to the MATLAB workspace or a MAT-file.

See Also

Objects
`mavlinktlog` | `flightLogSignalMapping` | `ulogreader`

Topics
"Analyze UAV Autopilot Flight Log Using Flight Log Analyzer"
"Add Custom Functions in Flight Log Analyzer App to Detect Region of Interest for Analysis"

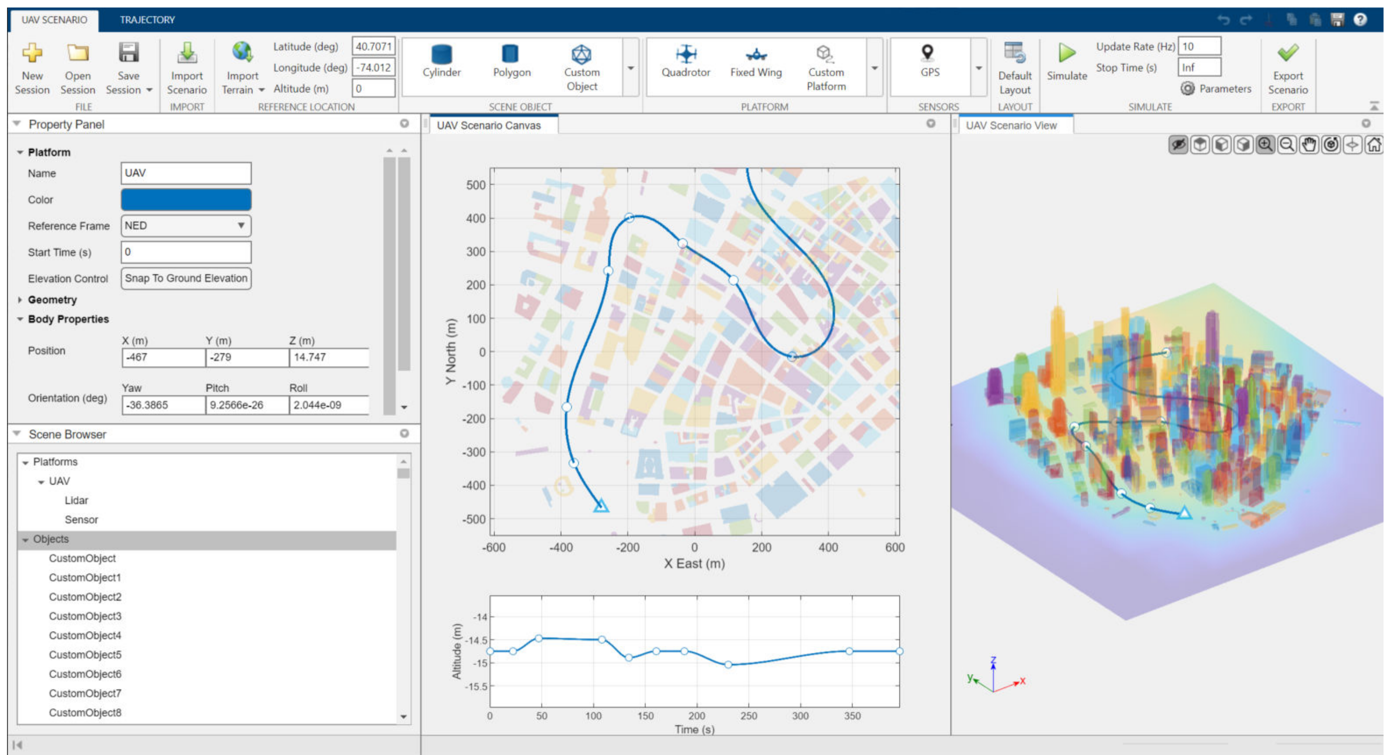
UAV Scenario Designer

Design UAV scenarios with terrain, platforms, and sensors

Description

The **UAV Scenario Designer** app enables you to interactively create a UAV scenario with terrain, platforms, and sensors and simulate trajectories for a UAV platform. Using the app, you can:

- Import, export, and create a UAV scenario
- Import terrain from Digital Terrain Elevation Data (DTED) files
- Add and edit scene objects, platforms, and sensors
- Add custom platforms and scene objects from STL files.
- Create and edit platform trajectories
- Simulate a UAV scenario



Open the UAV Scenario Designer App

- MATLAB Toolstrip: On the **Apps** tab, under **Robotics And Autonomous Systems**, click **UAV Scenario Designer** .
- MATLAB command prompt: Enter `uavScenarioDesigner`.

Examples

Create Session and Add to Scenario

Open the App

Open the **UAV Scenario Designer** app.

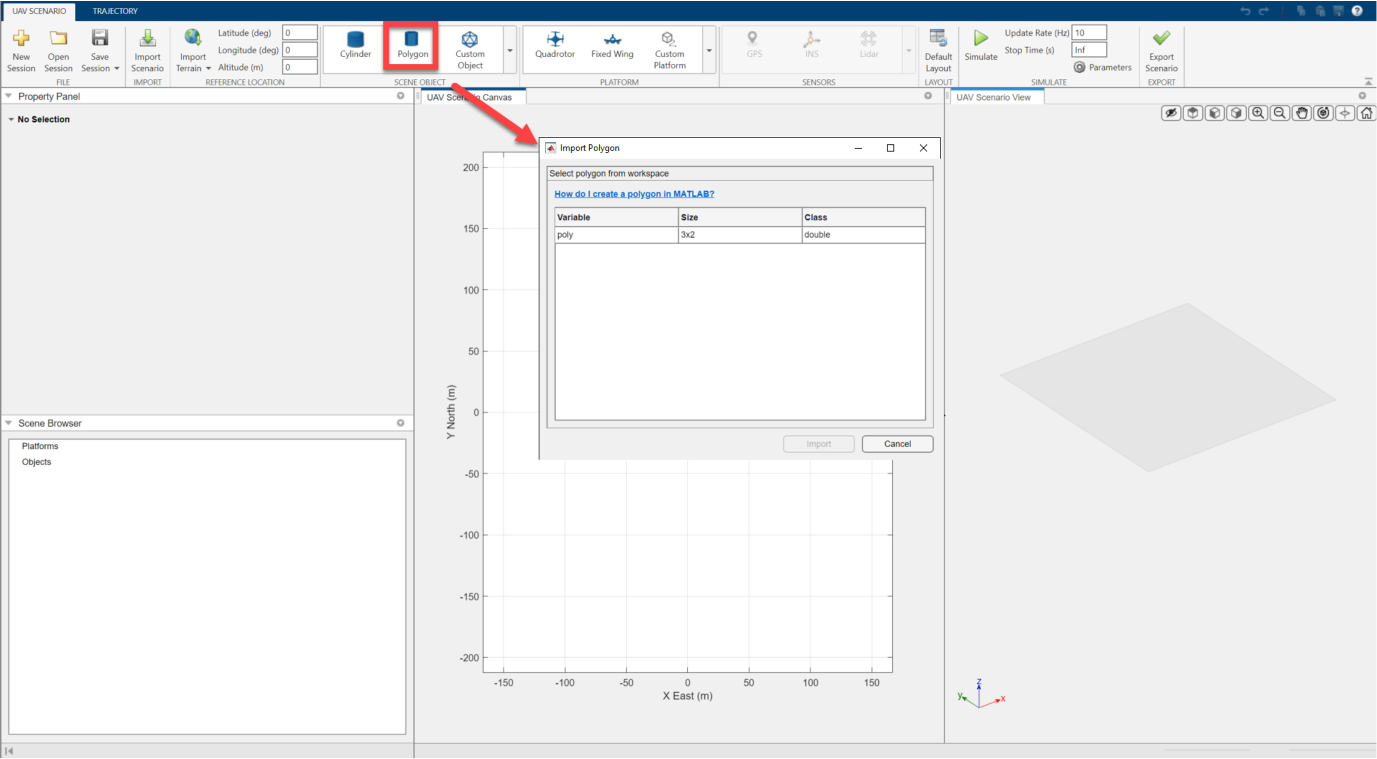
```
uavScenarioDesigner
```

Add Scene Objects

Define a polygon with three vertices. For more information about defining polygon scene objects, see “Create Polygon Scene Objects” on page 5-53.

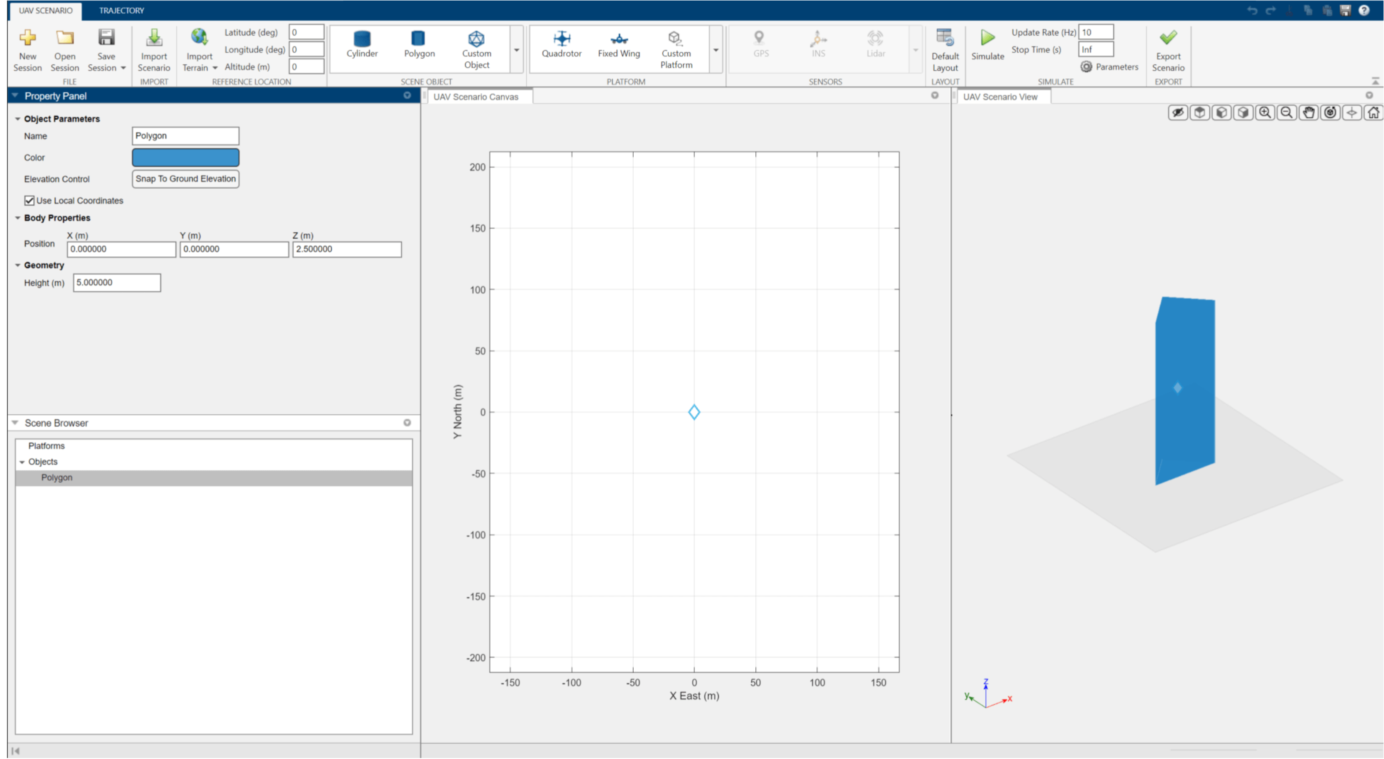
```
poly = [0 0; 1 1; 2 0];
```

In the **UAV Scenario Designer** app toolstrip, in the **Scene Object** section, select **Polygon**.



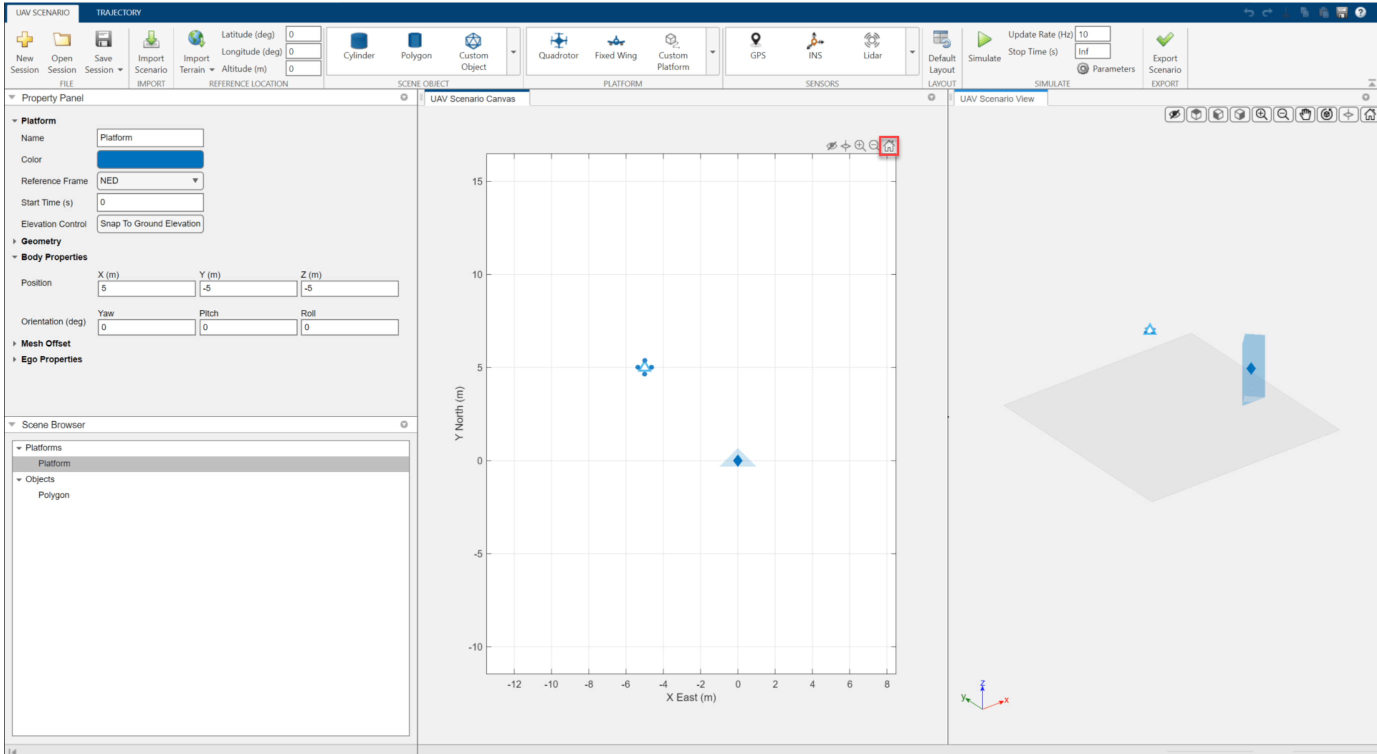
In the Import Polygon dialog box, select `poly` and click **Import**. Click anywhere on the **UAV Scenario Canvas** to place the polygon.

In the **Property Panel** pane, adjust the **Position** values of the polygon to for **X**, **Y**, and **Z** to `0`, and change **Height** to 5. To change the elevation of the polygon so that the bottom face makes contact with the ground, click **Snap To Ground Elevation**.



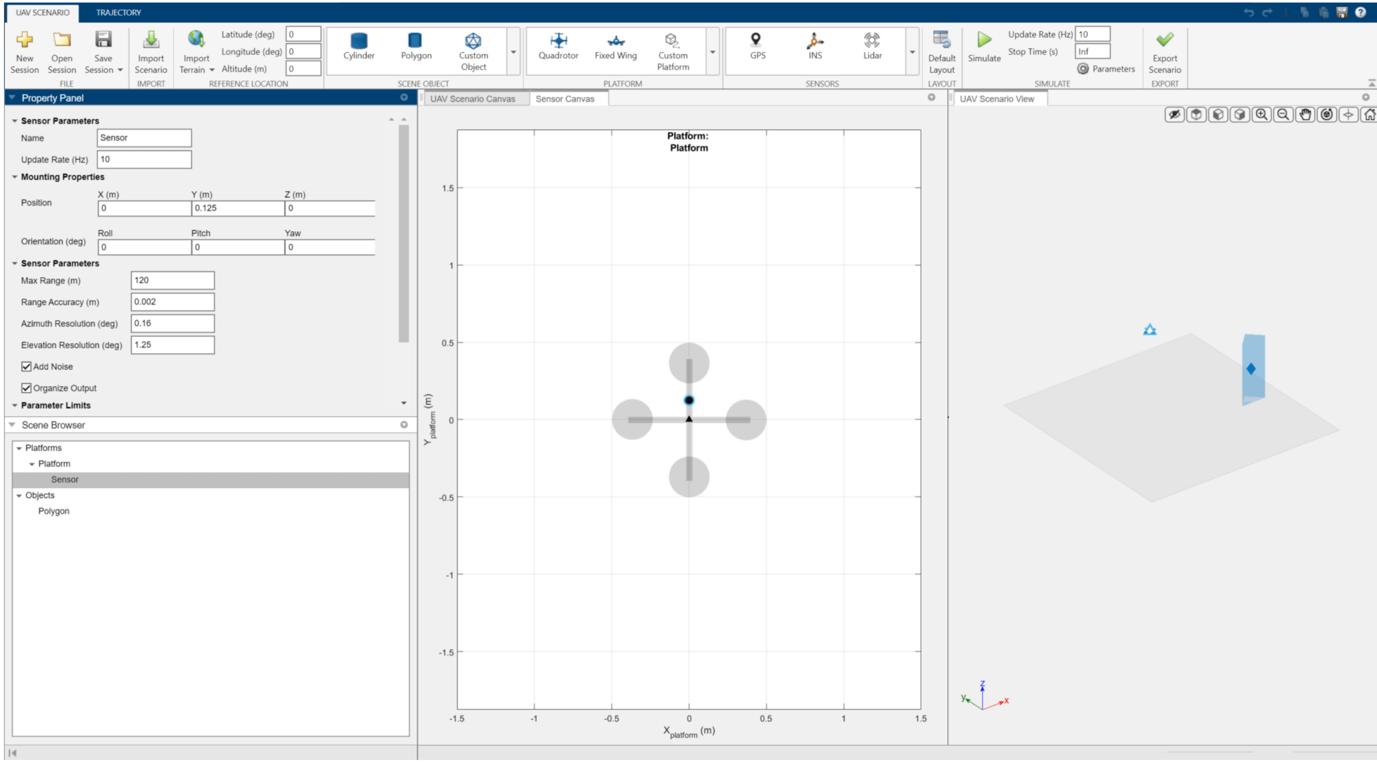
Add Platform

On the app toolstrip, in the **Platform** section, select **Quadrotor** and click anywhere in the **UAV Scenario Canvas** to place the quadrotor. In the **Property Panel** pane, set the **X** position of the quadrotor to 5 and the **Y** and **Z** positions of the quadrotor to -5. Note that the reference frame of the platform is north-east-down (NED) by default. Click **Zoom to Scenario** to zoom in to the polygon and platform.



Add Sensor

With the quadrotor platform selected, on the app toolstrip, in the **Sensors** section, select **Lidar**, and click anywhere on the **Sensor Canvas** to add a lidar sensor. In the **Property Panel** pane, change the **X** position of the mounting point of the lidar sensor to `0` and the **Y** and **Z** positions to `0.125` and `0`, respectively, in the local reference frame of the platform.



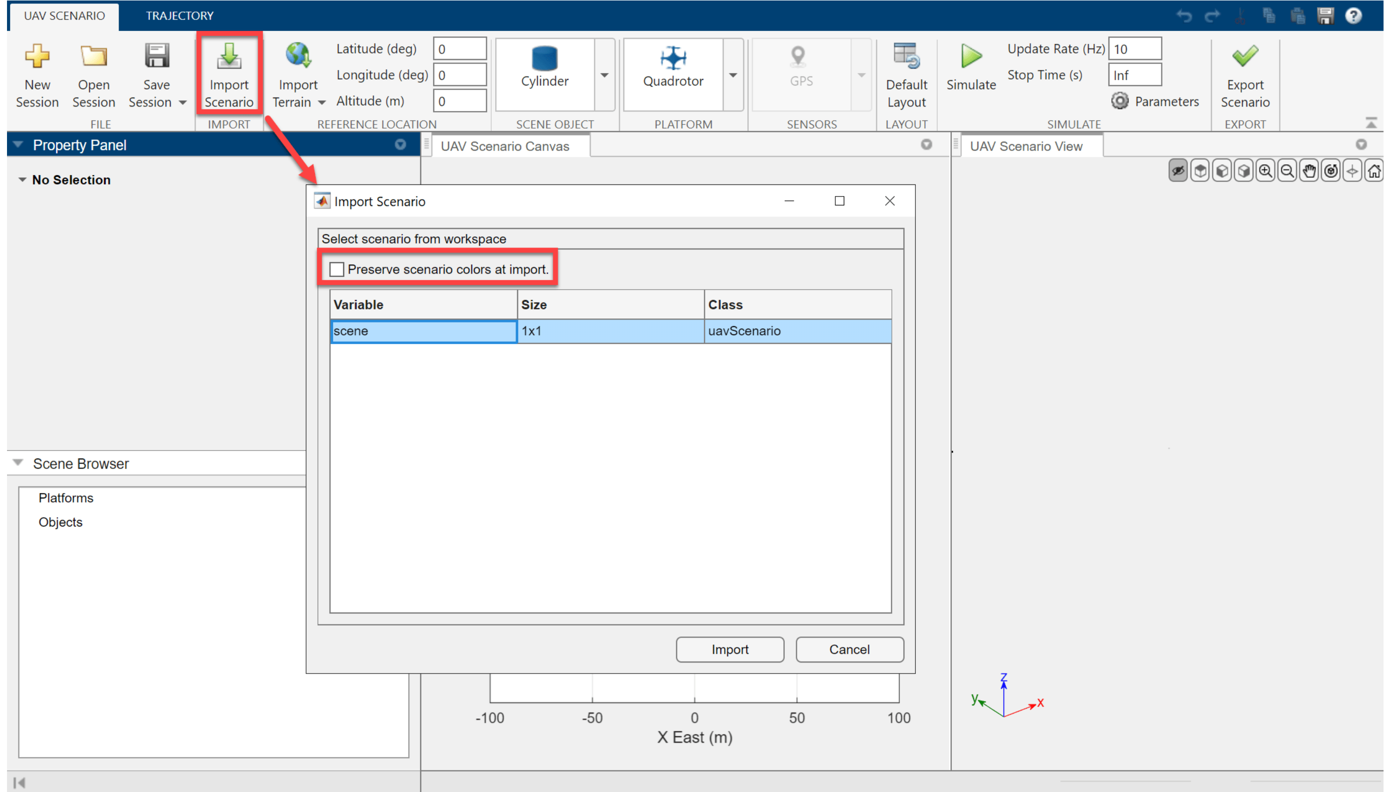
Create Trajectory and Simulate Scenario

Import UAV Scenario and Terrain File

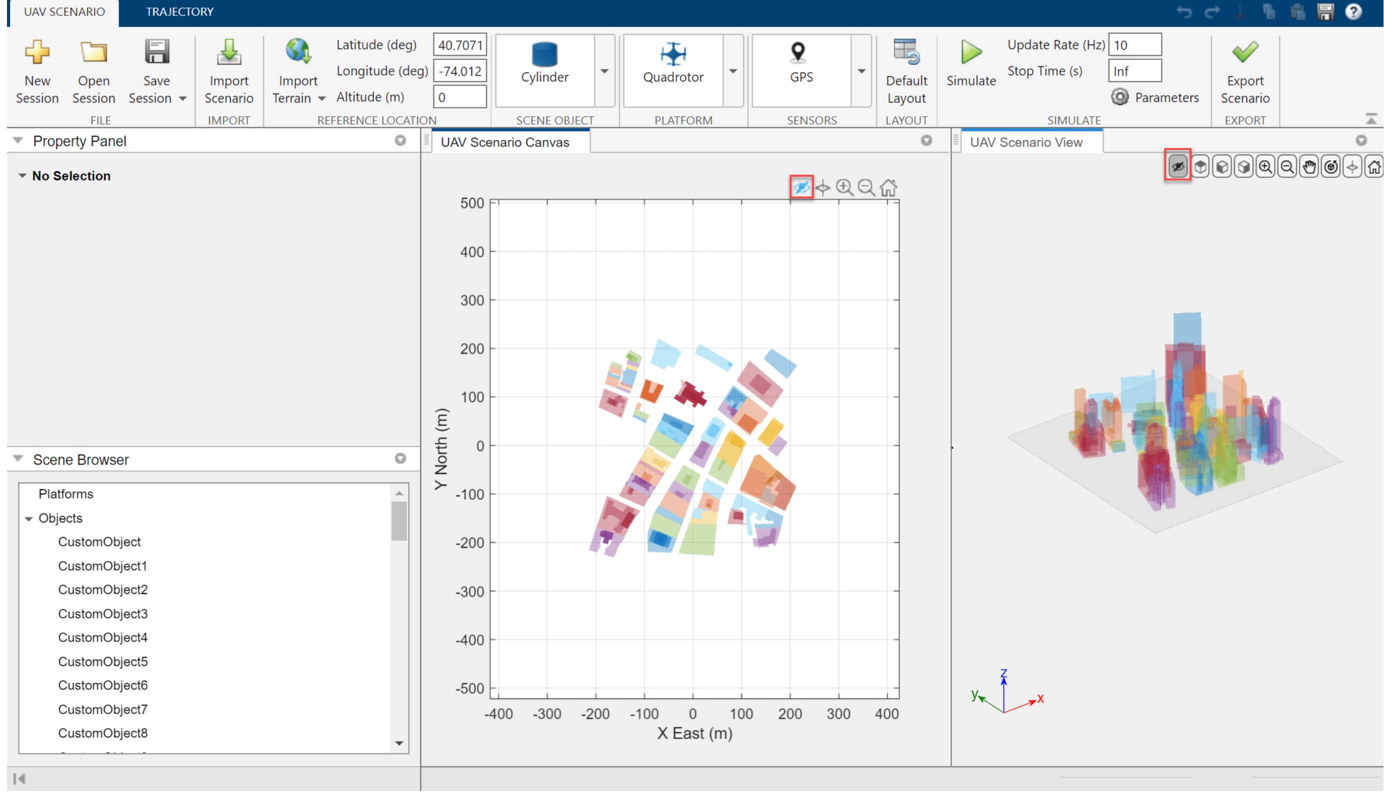
Create a UAV scenario and add building meshes from an OSM file containing building meshes for Manhattan [1] on page 5-43.

```
scene = uavScenario(ReferenceLocation=[40.707088 -74.012146 0]);
addMesh(scene,"terrain",{"gmted2010",[-200 200],[-200 200]},[0.6 0.6 0.6]);
addMesh(scene,"buildings",{"manhattan.osm",[-200 200],[-200 200],"auto"},[0 1 0]);
```

Open the **UAV Scenario Designer** app and click **Import Scenario** to import a scenario from the MATLAB workspace. To make the each building more distinguishable, generate new colors for each of the building meshes by clearing **Preserve scenario colors at import**. Select `scene` and click **Import**.



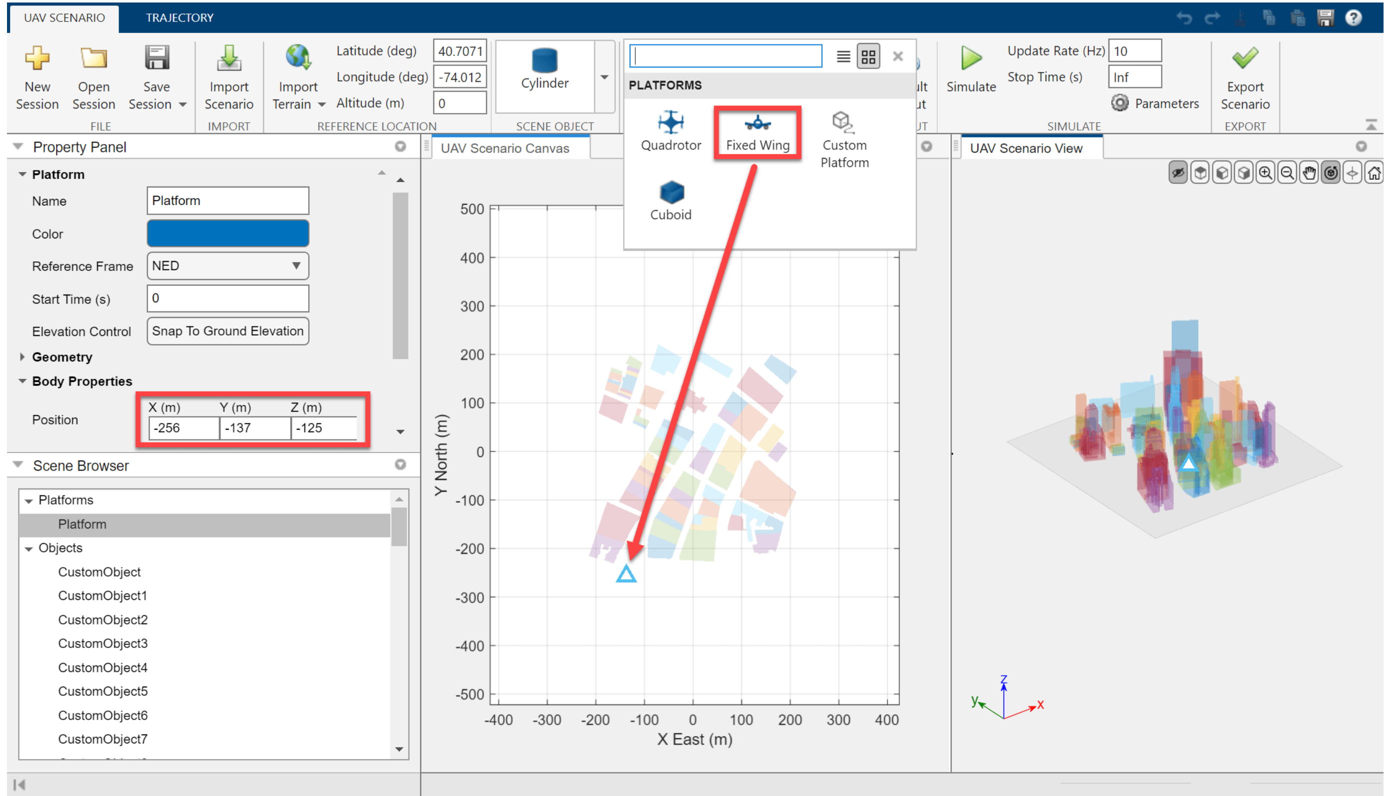
The imported building meshes appear. Turn off the building markers in the **UAV Scenario Canvas** and **UAV Scenario View** pane by selecting Hide Scene Object Markers in each pane.



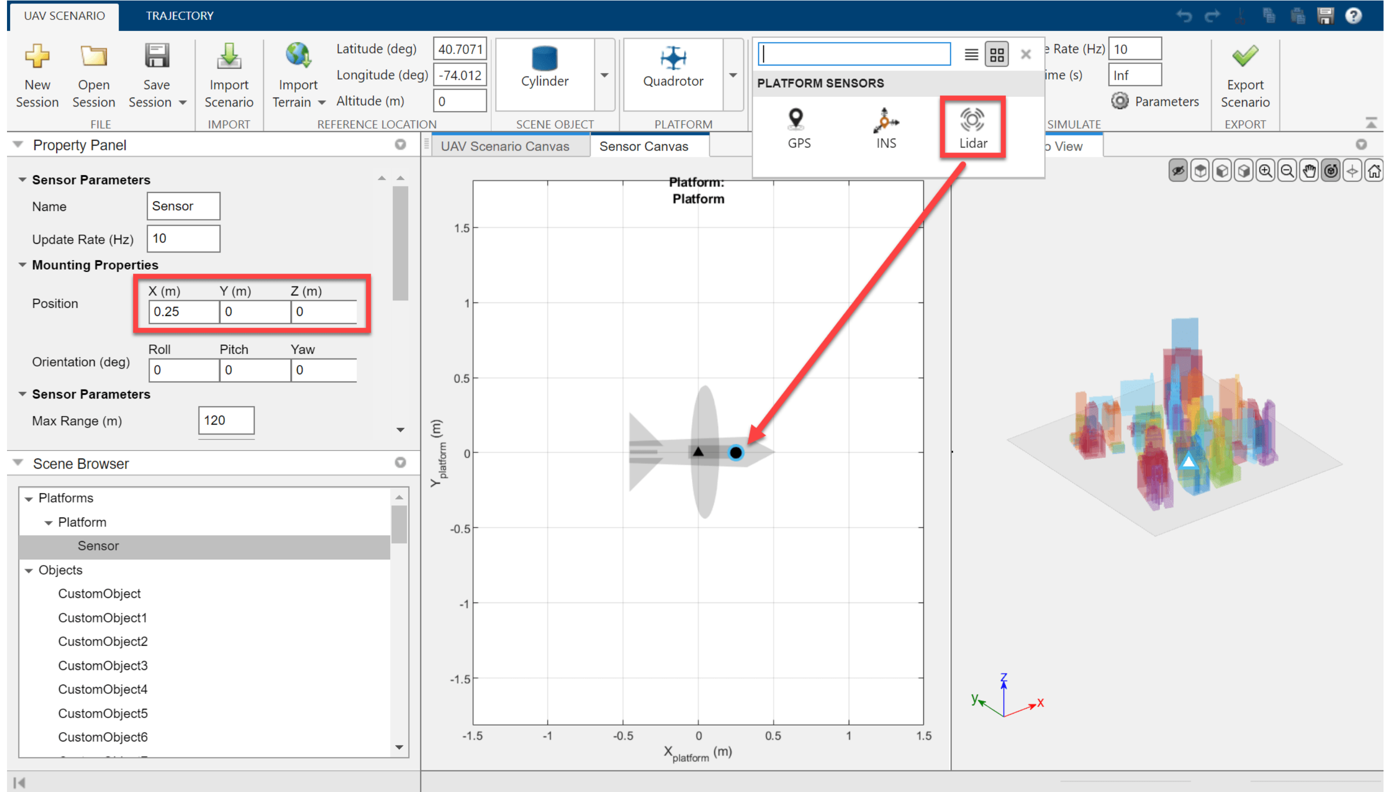
Note that to move buildings, you can drag the scene object marker of the building in the **UAV Scenario Canvas** pane. You can also zoom in to and delete a scene object from the **Scene Browser** pane by clicking and right-clicking on the object respectively.

Add Platform and Sensor

In the **Platform** section, select **Fixed Wing**, and then click anywhere in the **UAV Scenario Canvas** to add a fixed-wing UAV platform to the scenario. With the platform selected, in the **Property Panel** pane, set the **X** value of the **Position** parameter to `-256` and the **Y** and **Z** values to `-137` and `-125`, respectively. Note that the **Reference Frame** parameter of the platform is set to `NED`.

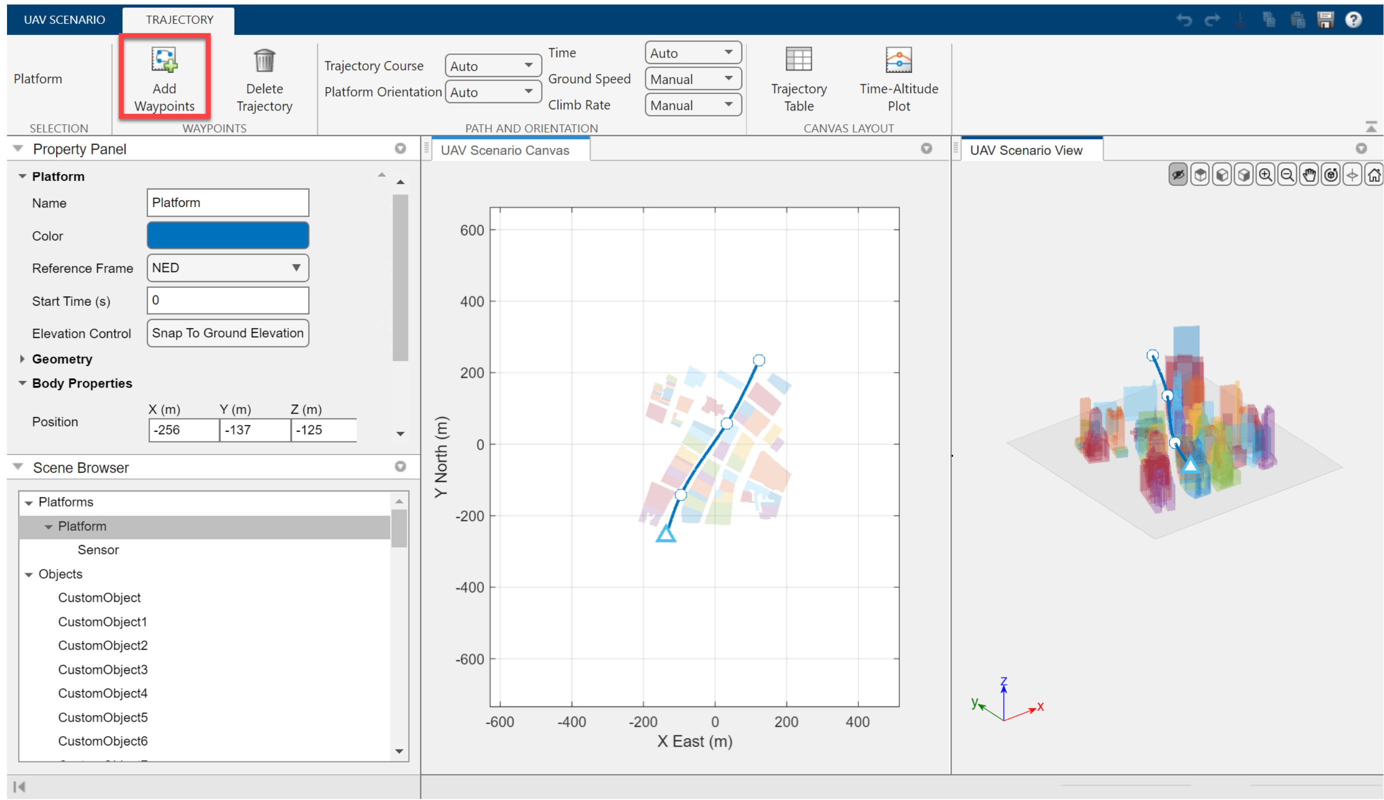


To add a lidar sensor to the platform, in the **Sensors** section, select **Lidar** and click anywhere in the **Sensor Canvas**. Specify the mounting position of the sensor on the platform by, in the **Property Panel** pane, setting the **X** value of the **Position** parameter to `0.25` and the **Y** and **Z** values to `0`. All values are in the local coordinate frame of the platform mesh.

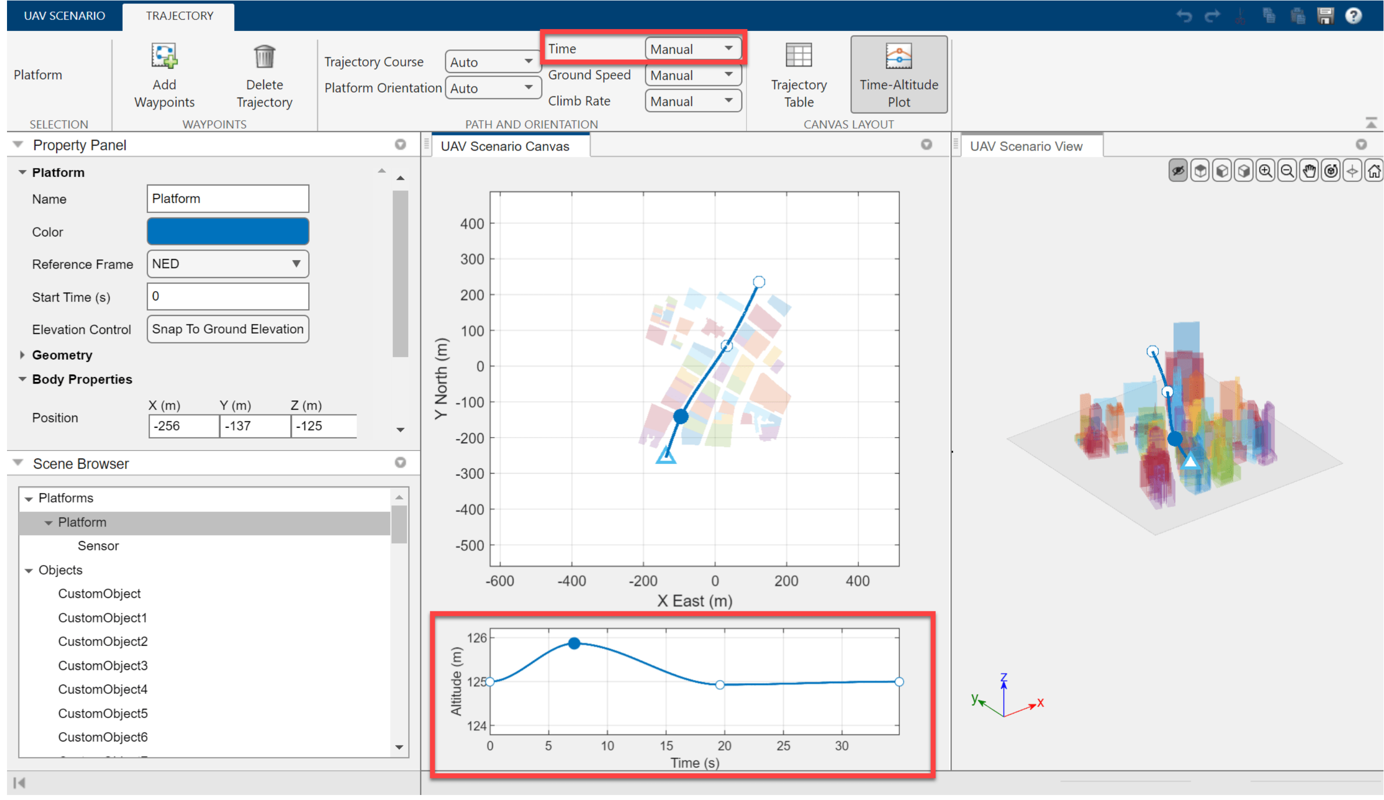


Create Trajectory

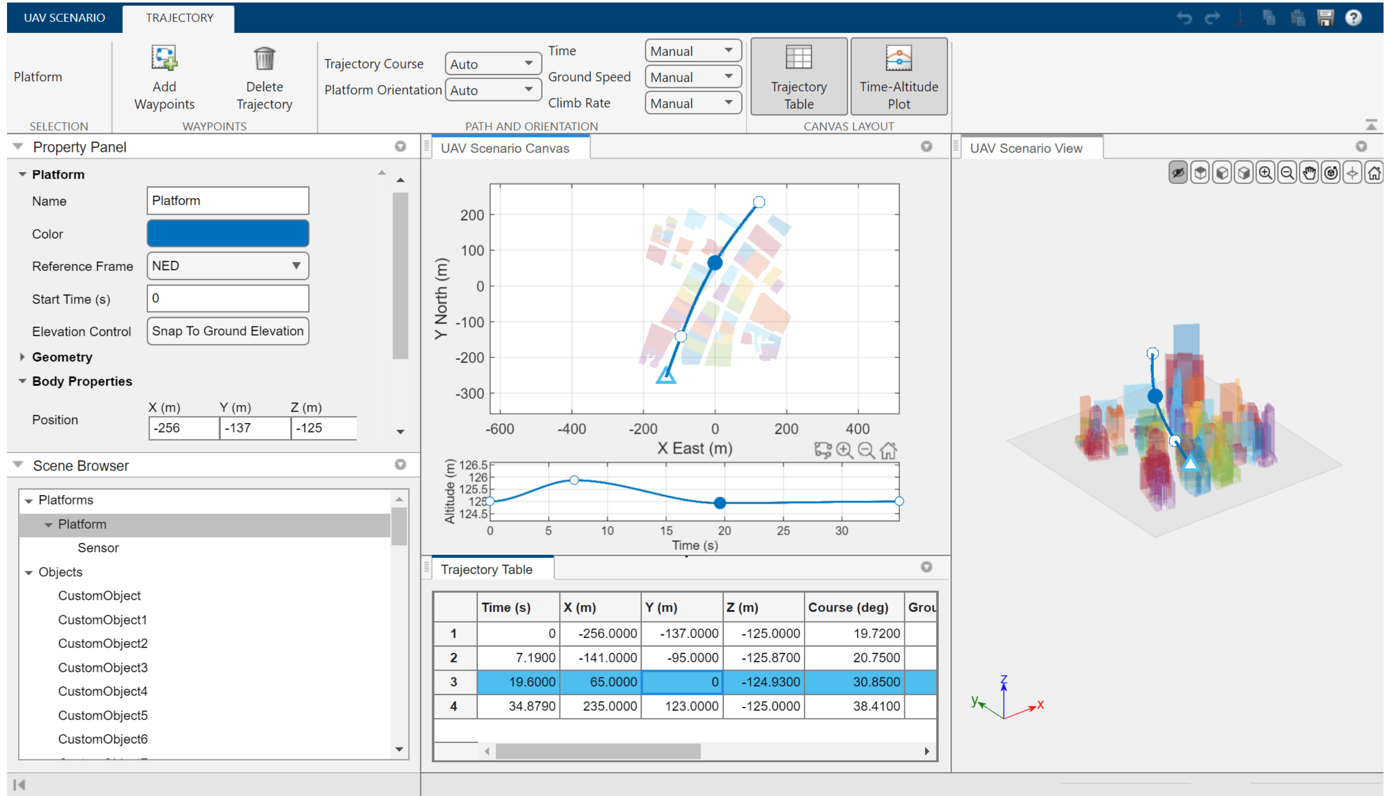
Select the **Trajectory** tab and, with the platform selected, select **Add Waypoints**. To create a trajectory for the platform, click in the **UAV Scenario Canvas**. To zoom in or out while adding points to the trajectory, scroll with the scroll wheel as needed. Add three points to the trajectory, and end trajectory creation by pressing **Esc**, **Enter**, or double-clicking in the **UAV Scenario Canvas**.



Click **Time-Altitude Plot** to open the time-altitude plot in the **UAV Scenario Canvas**. In the time-altitude plot, drag the first waypoint vertically to an altitude of `126` meters in the time-altitude plot. Note that you cannot move the waypoint left or right along the Time axis. To change the time value for waypoints, on the app toolstrip, in the **Path and Orientation** section, set **Time** to `Manual`. Then, drag the second waypoint to approximately 125 meters in altitude at `20` seconds. For more information about the other trajectory settings in the **Path and Orientation** section, see “Trajectory” on page 5-0 .



Note, in the **UAV Scenario View** pane, that the platform may be too close to one of the buildings at the second waypoint. Adjust this waypoint by dragging it on the **UAV Scenario Canvas**, or by editing the position of the waypoint in the **Trajectory Table**. Select **Trajectory Table** to open the **Trajectory Table** pane, and click on the second waypoint. The table highlights the data of the selected waypoint in blue. Set the **X** and **Y** elements to `65` and `0`, respectively. You can edit the data of any waypoint in this table as long as the corresponding **Path and Orientation** parameters are set to `Manual`.

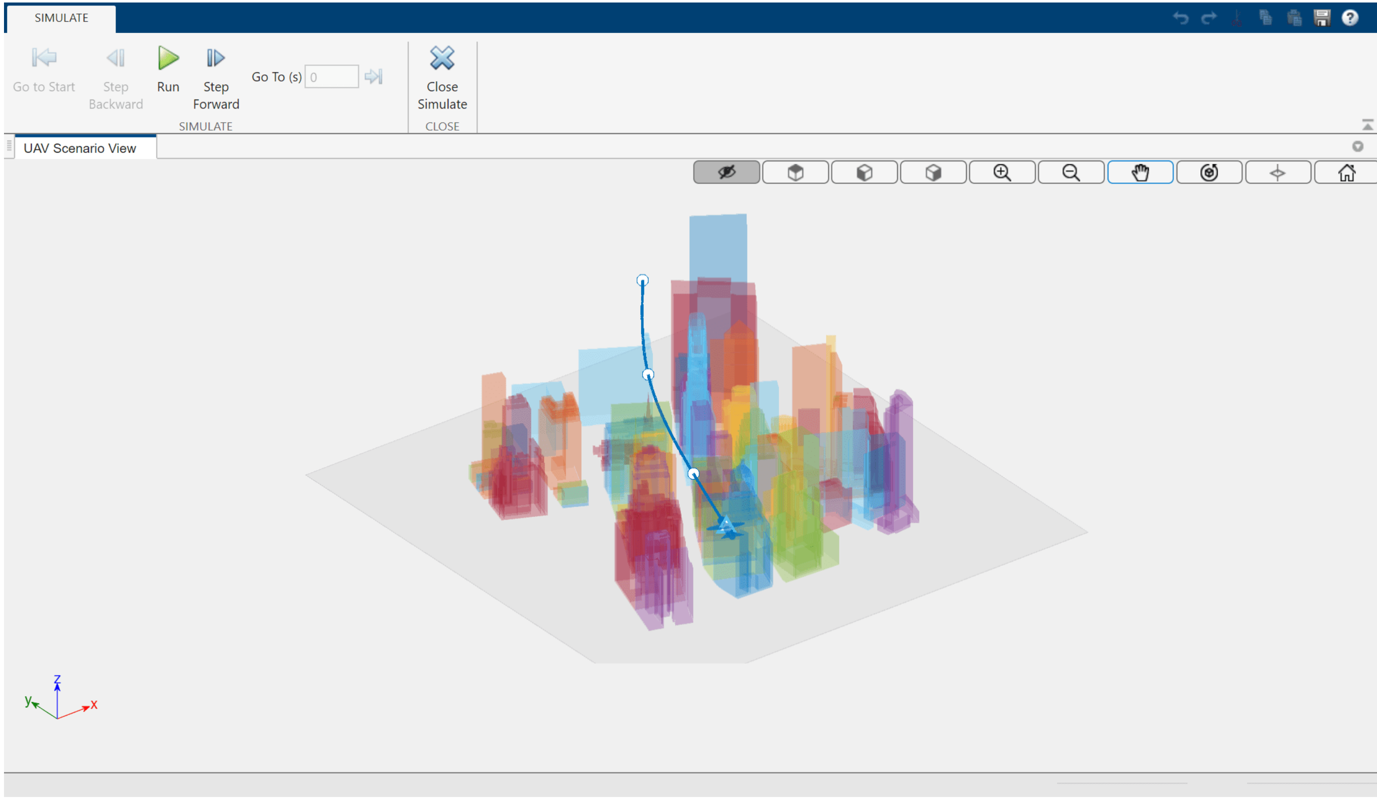


Note that you can also delete or insert waypoints by right-clicking a waypoint or a trajectory between waypoints in the **UAV Scenario Canvas** and clicking **Delete Waypoint** or **Insert Waypoint** respectively in the right-click dialog boxes.

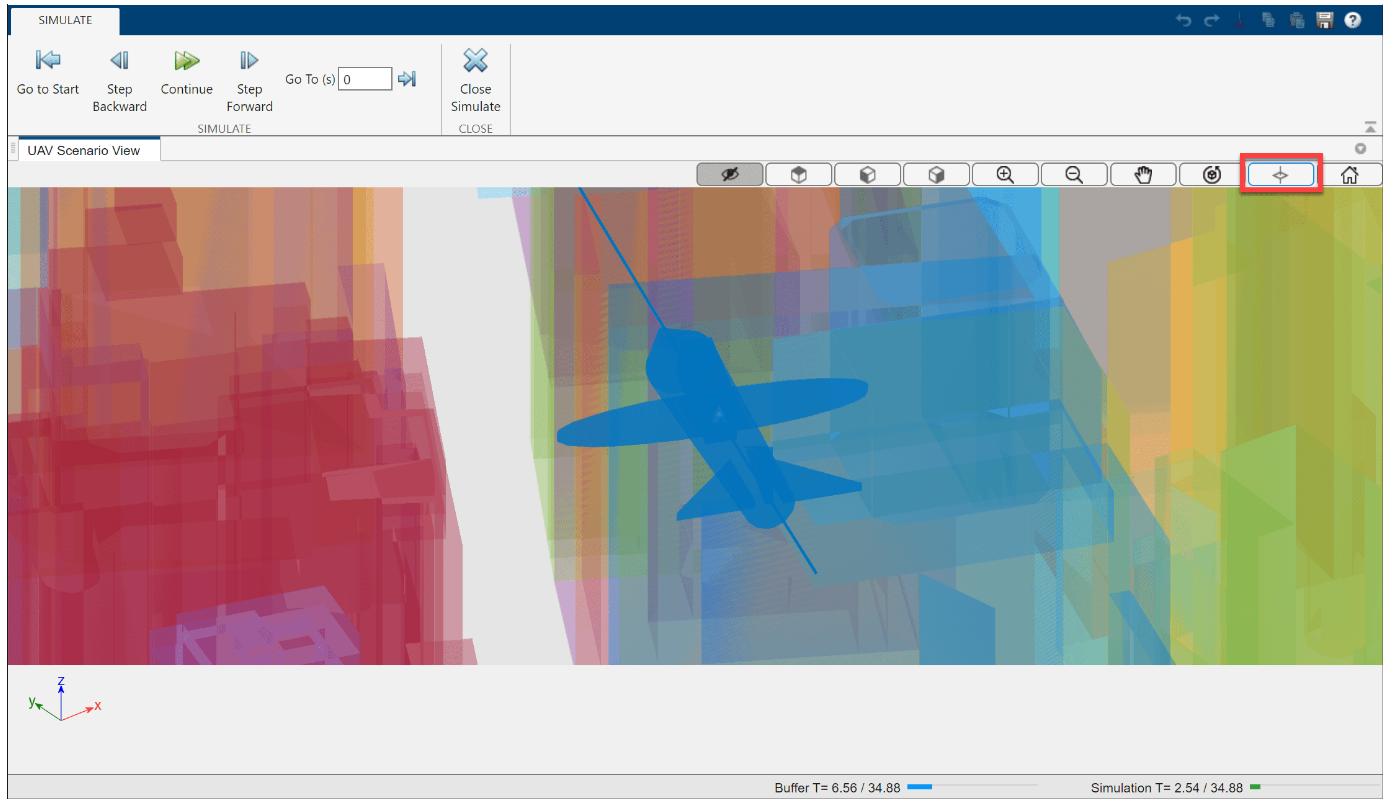
Simulate Scenario

In the **Scene Browser** pane, select the platform. Then select **UAV Scenario** tab and set **Update Rate** to `50` Hz. In the **Simulate** section, select **Parameters** and set **Number of Frames** to `30`.

Click **Simulate** to open the **Simulate** tab.



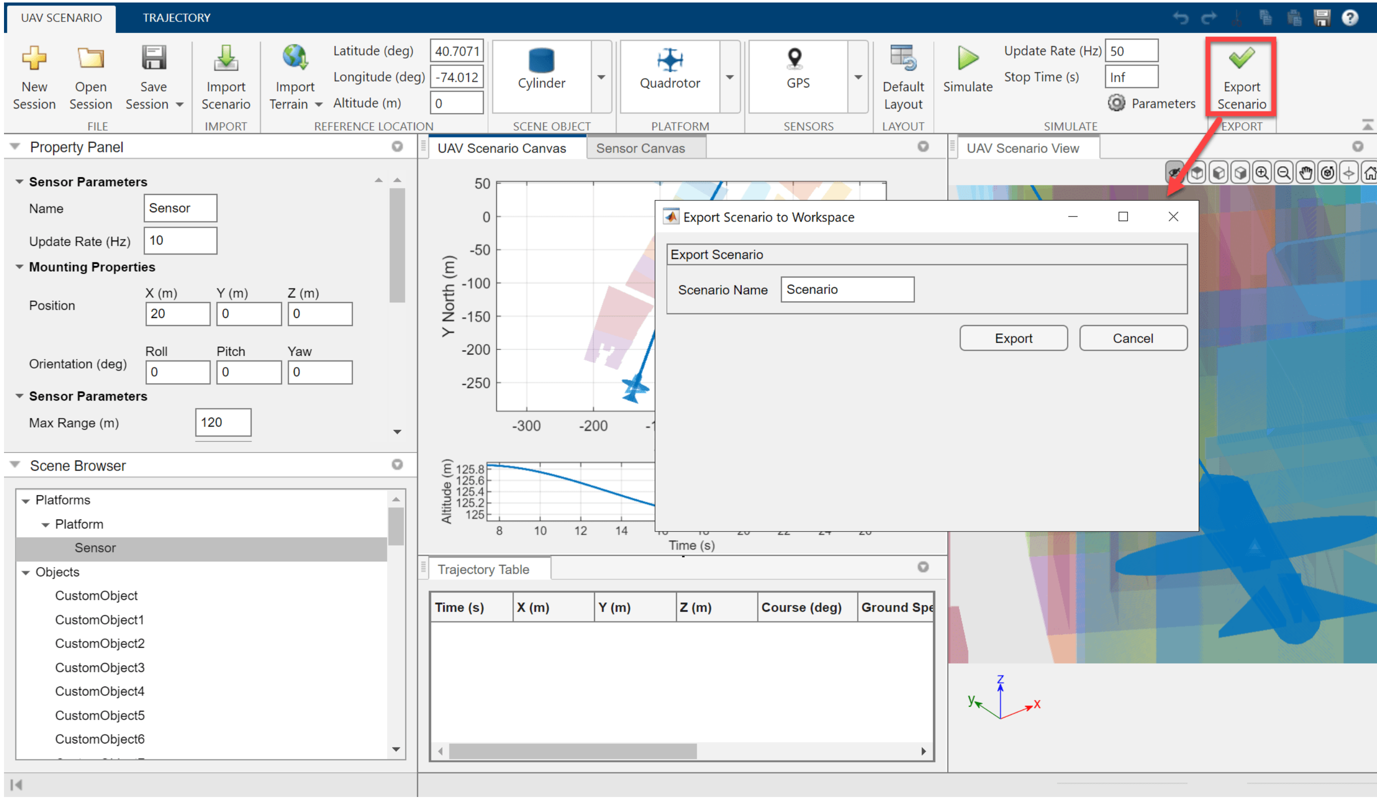
Run the simulation and click Zoom to Selection to center the camera over the platform as it follows the trajectory.



After the platform completes the trajectory, click **Close Simulate** to return to the **UAV Scenario** tab.

Export UAV Scenario

Export the scenario or session to share the scenario or to later modify the scenario in UAV Scenario Designer. Click **Export Scenario** to open the Export Scenario to Workspace dialog box. Name the scenario and click **Export** to export the scenario to the MATLAB workspace as a `uavScenario` object.



To save the session as a MAT file, in the **File** section, select **Save Session**.

References

[1] The file was downloaded from https://www.openstreetmap.org, which provides access to crowd-sourced map data all over the world. The data is licensed under the Open Data Commons Open Database License (ODbL), https://opendatacommons.org/licenses/odbl/

Import UAV Scenario with Polynomial Trajectory

Create UAV Scenario

Create a UAV scenario and set its local origin.

```
scene = uavScenario(UpdateRate=100,StopTime=15,ReferenceLocation=[46 42 0]);
```

Add an inertial frame, `MAP`, to the scenario.

```
scene.addInertialFrame("ENU","MAP",trvec2tform([1 0 0]));
```

Use the `minjerkpolytraj` function to generate the piecewise polynomial for the specified waypoints of the trajectory.

```
waypoints = [0 50 100; 0 -50 0; 0 0 0];
timePoints = [0 3.8596 11.4451];
```

```
numSamples = 100;
[~,~,~,~,pp,~,~] = minjerkpolytraj(waypoints,timePoints,numSamples);
```

Generate Trajectory from Piecewise Polynomial Using polynomialTrajectory

Use the `polynomialTrajectory` System object™ to generate a trajectory from the piecewise polynomial. Specify the sample rate of the trajectory. Set the `AutoBank` and `AutoPitch` properties to `true`.

```
traj = polynomialTrajectory(pp,SampleRate=100,AutoBank=true,AutoPitch=true);
```

Create a UAV platform with a specified polynomial trajectory in the scenario. Define the mesh for the UAV platform.

```
platform = uavPlatform("custom",scene,"Trajectory",traj);
updateMesh(platform,"fixedwing",{20},[1 0 0],eul2tform([0 0 pi]))
```

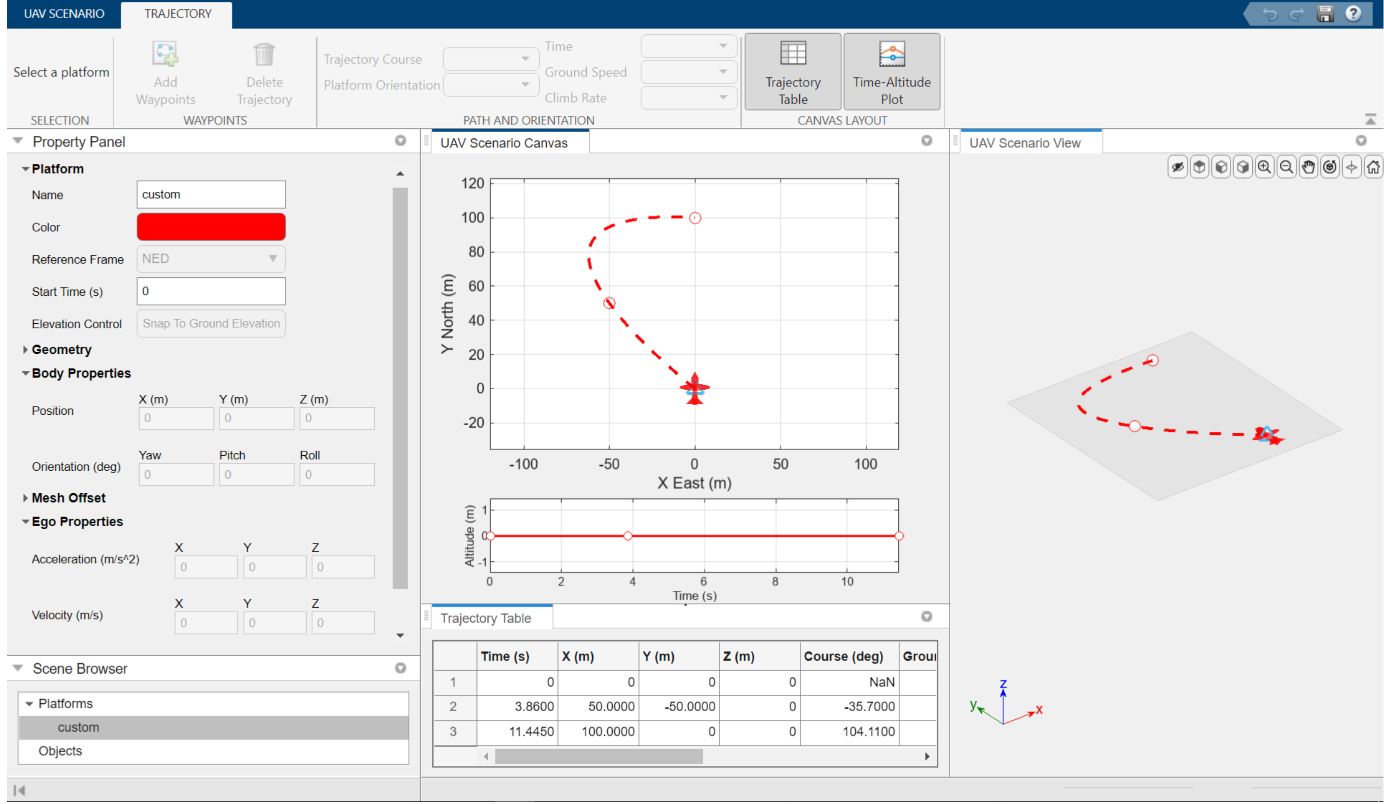
Import UAV Scenario

Open the UAV Scenario Designer app.

```
uavScenarioDesigner
```

To import a UAV scenario into the UAV Scenario Designer app:

1. Click **Import Scenario** to import a scenario from the MATLAB® workspace.
2. Select `scene` and click **Import**. The app displays a pop-up window warning you about the limitations of using a platform with polynomial trajectories. For more information see Limitations on page 5-45. Click **OK** to close the pop-up window.
3. Click the **Trajectory** tab, and select the `custom` platform from the **Scene Browser** pane.
4. Click **Trajectory Table** to open the **Trajectory Table** pane. You cannot edit the data in this table. For more information, see Limitations on page 5-45.
5. Click **Time-Altitude Plot** to open the time-altitude plot. You cannot adjust the waypoints in the time-altitude plot. For more information, see Limitations on page 5-45.
6. You can only rename the platform and change the start time in the **Property Panel** pane. All other properties are read-only.



Simulate Scenario

To simulate the scenario:

1. Select the **UAV Scenario** tab.
2. In the **Simulate** section of the app toolstrip, click **Simulate** to open the **Simulate** tab.
3. Run the simulation.

To save the session as a MAT file, in the **File** section, select **Save Session**.

Limitations

UAV Scenario Designer app support for the `polynomialTrajectory` System object is limited to importing the trajectory into a scene and simulating the trajectory. Editing and creation options are not available when using `polynomialTrajectory`.

- "Design Obstacle Avoidance Package Delivery Scenario Using UAV Scenario Designer"

Parameters

`Trajectory` — Trajectory settings
tab

To add or edit a trajectory and control the trajectory generation, use the trajectory settings.

- Click **Waypoints** to add waypoints to a trajectory of a selected platform.
- Click **Delete Trajectory** to delete an existing trajectory.
- Click **Trajectory Table** to display the trajectory table. See **Trajectory Table** for more information.
- Click **Time-Altitude plot** to display the time vs altitude plot.

You can also choose to automatically generate the waypoint trajectory or manually input waypoints by changing the selections of the **Path and Orientation** parameters.

Parameter	Selection
Trajectory Course	• **Auto**: When selected, the app generates the course by fitting all the waypoints with a smooth curve. • **Table**: When selected, you can manually edit the trajectory course at each waypoint using the **Trajectory Table**.
Platform Orientation	• **Auto**: When selected, the app calculates the yaw and pitch angles of the platform to align the platform with the trajectory and calculates the roll angle to cancel the centripetal acceleration. • **Table**: When selected, you can manually edit the yaw, pitch, and roll angles at each waypoint using the **Trajectory Table**.
Time	• **Auto**: When selected, the app calculates the visiting time at all the waypoints. • **Table**: When selected, you can manually edit the visiting time at each waypoint using the **Trajectory Table**.
Ground speed	• **Auto**: When selected, the app uses the default ground speed for each platform class at each waypoint. • **Table**: When selected, you can manually edit the ground speed at each waypoint using the **Trajectory Table**. **Note** **UAV Scenario Designer** does not support negative ground speeds.
Climb Rate	• **Auto**: When selected, the app calculates the climb rate at each waypoint to smoothly fit all the waypoints. • **Table**: When selected, you can manually edit the climb rate at each waypoint using the **Trajectory Table**.

`Trajectory Table` — Trajectory information
table

Trajectory information for each waypoint, specified as a table of scalars. When you insert waypoints on the platform canvas, the table is automatically generated. Click **Trajectory Table** under the **Trajectory** tab to display the table.

Edit the parameters in the table to adjust or fine-tune the trajectory. After you change the parameter values in the table, the platform trajectory changes accordingly on the canvas. The table includes these trajectory parameters.

Parameter	Description
Times	Time at which the platform visits the waypoint, specified as a scalar in seconds.
X	x coordinate of the waypoint in the scenario navigation frame.
Y	y coordinate of the waypoint in the scenario navigation frame.
Altitude	Altitude of the platform waypoint in the scenario navigation frame.
Course	The direction of motion on the x-y plane, specified as an angle measurement from the x direction.
Ground speed	Magnitude of the projected velocity on the x-y plane, specified as a nonnegative scalar in meters.
Climb Rate	Climb rate of the waypoint, which is the projection of the platform velocity in the z direction.
Roll	Orientation angle of the platform about the x-axis of the scenario frame, in degrees, specified as a scalar.
Pitch	Orientation angle of the platform about the y-axis of the scenario frame, in degrees, specified as a scalar.
Yaw	Orientation angle of the platform about the z-axis of the scenario frame, in degrees, specified as a scalar.

`Simulate` — Simulate UAV scenario tab
tab

Click **Simulate** in the **UAV Scenario** tab to open the **Simulate** tab.

Use the toolstrip buttons to control the simulation.

Click Hide Scene Object Markers in the **UAV Scenario View** pane to hide the object markers.

Click Zoom to Selection in the **UAV Scenario View** pane to zoom in on the selected object. Enable Zoom to Selection in simulation mode to follow the selected object.

`Property Panel (Platforms)` — Platform properties
pane

Use the **Property Panel** to edit the properties of a selected platform, such as geometry, body properties, mesh offset, and ego properties.

Platform

Parameter	Description
Name	Name of the platform.
Color	Color of the platform and platform trajectory.
Reference Frame	Reference frame of the platform, specified as `NED` (north-east-down) or `ENU` (east-north-up).
Start Time	Start time of the platform, in seconds.
Elevation Control	Select **Snap To Ground Elevation** to set the elevation of the platform to ground level.

To edit the geometry of the platform, use the **Geometry** parameters.

Geometry

Parameter	Description
Scale	Scale of the platform mesh. Default is `1`. If the platform is a cuboid platform, this parameter is read-only.
Length	Length of the mesh, in meters. For quadrotor, fixed-wing, and custom platforms, this property is read-only and affected by the **Scale** parameter.
Width	Width of the mesh, in meters. For quadrotor, fixed-wing, and custom platforms, this property is read-only and affected by the **Scale** parameter.
Height	Height of the mesh, in meters. For quadrotor, fixed-wing, and custom platforms, this property is read-only and affected by the **Scale** parameter.

To edit the body properties of the scene object, use the **Body Properties** parameters.

Body Properties

Parameter	Description
Position	Specify the **X**, **Y**, and **Z** positions of the platform in the global coordinate frame, in meters. The properties change depending on the selected reference frame. • `ENU` — **X** (East), **Y** (North), **Z** (Up) • `NED` — **X** (North), **Y** (East), **Z** (Down)
Orientation	Specify the **Yaw**, **Pitch**, **Roll** orientation from the world frame to the body frame in *Z*-*Y*-*X* order in degrees. • `ENU` — **Yaw** (East), **Pitch** (North), **Roll** (Up) • `NED` — **Yaw** (North), **Pitch** (East), **Roll** (Down)

To edit the mesh offset of the scene object, use the **Mesh Offset** parameters.

Mesh Offset

Parameter	Description
Position Offset	Mesh position offset, in meters, from the platform frame in the **X**, **Y**, and **Z** directions.
Orientation Offset	Mesh orientation offset, in degrees, from the platform frame in **Roll**, **Pitch**, and **Yaw** directions.

To edit the ego properties of the scene object, use the **Ego Properties** parameters. The ego properties are initial conditions that move the mesh during simulation. If the platform has a trajectory, then these properties are read-only.

Ego Properties

Parameter	Description
Acceleration	Acceleration of the platform, in meters, per second squared.
Velocity	Velocity of the platform, in meters, per second.
Angular Velocity	Angular velocity of the platform, in degrees, per second.

`Property Panel (Scene Objects)` — Scene object properties
pane

Use the **Property Panel** to edit the scene object parameters and body properties of a selected object. Cylinders, polygons, and custom objects all share these object parameters:

Object Parameters

Parameter	Description
Name	Name of the scene object.
Color	Color of the scene object.
Elevation Control	Select **Snap To Ground Elevation** to set the elevation of the scene object to ground level.
Use Local Coordinates	Select to use a local coordinate frame, specifying **Position** as **X**, **Y**, and **Z**. Clear this parameter to specify **Position** as **Longitude**, **Latitude**, and **Altitude**.

Note Scene objects coordinates are always defined in east-north-up (ENU).

The **Body Properties** contains the position of the scene object, **Position**. If you select **Use Local Coordinates**, these values are in Cartesian coordinates, in meters. Otherwise these values are the latitude and longitude of the object, in degrees, and the altitude in meters.

To edit the geometry of the scene object, use the **Geometry** parameters.

Geometry

Parameter	Description
Radius (Cylinder only)	Radius of the cylinder, in meters.
Height	Height of the cylinder or polygon, in meters

`Property Panel (Sensors)` — Sensor properties
pane

To view and edit sensor properties in the **Property Panel**, select a sensor. **UAV Scenario Designer** supports these three sensors: GPS, INS, and lidar. These MATLAB object equivalents for these sensors are equivalent to the `gpsSensor`, `insSensor`, and `uavLidarPointCloudGenerator` MATLAB objects respectively.

Use the **Sensor Parameters** and **Mounting Properties** to edit sensor parameters and their mounting position on the platform mesh respectively. These properties are the same for all sensor objects.

Sensor Parameters

Parameter	Description
Name	Name of the sensor.
Update Rate	Update rate of the sensor, in hertz

Mounting Properties

Parameter	Description
Position	Mounting position of the sensor with respect to the platform body origin.
Orientation	Mounting orientation of the sensor with respect to the platform body orientation.

To edit the **GPS Parameters**, select a GPS sensor.

GPS Parameters

Parameter	Description
Reference Location	Reference location of the sensor, specified in geodetic coordinates with latitude and longitude in degrees and altitude in meters.
Position Input Format	Position input format of the sensor, specified in local Cartesian or geodetic coordinates.
Reference Frame	Reference frame of the sensor, specified as `NED` (north-east-down) or `ENU` (east-north-up).
Horizontal Position Accuracy	Horizontal position accuracy of the sensor, specified in meters.
Vertical Position Accuracy	Vertical position accuracy of the sensor, specified in meters.
Velocity Accuracy	Velocity accuracy of the sensor, specified in meters per second.
Decay Factor	Decay factor of the sensor, specified as a number in the range `[0, 1]`.
Random Stream	Random stream, specified as `Global stream` or `mt19937ar with seed`.
Seed	Specify seed random stream.

To edit the **INS Parameters**, select an INS sensor.

INS Parameters

Parameter	Description
Position Accuracy	Position accuracy of the INS sensor, specified as **X**, **Y**, and **Z**, in meters.
Orientation Accuracy	Orientation accuracy of the INS sensor, specified as **Roll**, **Pitch**, and **Yaw**, in degrees.
Position Error Factor	Position error factor of the INS sensor, specified in meters.
Velocity Accuracy	Velocity accuracy of the INS sensor, specified in meters per second.
Acceleration Accuracy	Acceleration accuracy of the INS sensor, specified in meters per second squared.
Angular Velocity Accuracy	Angular velocity accuracy of the INS sensor, specified in degrees.
Fix GNSS	Select to lock the GNSS readings of the INS sensor.

To edit the **Lidar Parameters**, select a lidar sensor.

Lidar Parameters

Parameter	Description
Max Range	Maximum range of the lidar sensor, in meters.
Range Accuracy	Range accuracy of the lidar sensor, in meters.
Azimuth Resolution	Azimuth resolution of the lidar sensor, in degrees.
Elevation Resolution	Elevation resolution of the lidar sensor, in degrees.
Add Noise	Select to add noise to the lidar sensor output.
Organize Output	Select to output an organized point cloud.
Elevation Limits	Elevation scanning limits of the lidar sensor, in degrees.
Azimuth Limits	Azimuth scanning limits of the lidar sensor, in degrees.

`Property Panel (Import Terrain)` — Terrain import properties
pane

Use the **Property Panel** to set the terrain import properties for the selected terrain after clicking **Import Terrain**.

Terrain Properties

Parameter	Description
Name	Name of the terrain file. This parameter is read-only.
Import Terrain	Select to import the terrain, once you have set all the parameters.
Use Local Coordinates	Select to specify the terrain limits and bounds using local coordinates. To use geodetic coordinates for the terrain limits and bounds, clear this parameter.

To edit the terrain limits, use the **Terrain Limits** parameters.

Terrain Limits

Parameter	Description
X (m) and **Y (m)**	Minimum and maximum *X*- and *Y*-axis limits of the terrain. To enable this parameter, select **Use Local Coordinates**.
Latitude Limits (deg) and **Longitude Limits (deg)**		Minimum and maximum latitude and longitude limits of the terrain. To enable this parameter, clear **Use Local Coordinates**.

To edit the terrain bounds, use the **Terrain Bounds** parameters.

Terrain Bounds

Parameter	Description
X (m) and **Y (m)**	Minimum and maximum *X*- and *Y*-axis limits of the terrain bounds. To enable this parameter, select **Use Local Coordinates**.
Latitude Limits (deg) and **Longitude Limits (deg)** |	Minimum and maximum latitude and longitude limits of the terrain bounds. To enable this parameter, clear **Use Local Coordinates**.

Programmatic Use

`uavScenarioDesigner` opens the **UAV Scenario Designer** app.

Limitations

- **UAV Scenario Designer** may run slowly if MATLAB is using a software implementation of OpenGL®. To solve the problem, upgrade your graphics hardware driver or use `opengl` to switch to a hardware-accelerated implementation of OpenGL. See “Resolving Low-Level Graphics Issues” for more information.
- **UAV Scenario Designer** app support for `polynomialTrajectory` System object is limited to importing the trajectory into a scene and simulating the trajectory. Edit and create options are not available when using `polynomialTrajectory`.

More About

Create Polygon Scene Objects

To import 3-D polygons into a scene in **UAV Scenario Designer**, define a polygon in MATLAB as an *N*-by-2 matrix of vertices, where each row represents the *x*- and *y*-position of each vertex. The rows should be sequential either clockwise or counter-clockwise. When you import a polygon into the app, the default height of the polygon is set to `10` meters. Select the polygon to edit the height and center of position of the polygon by using the “Property Panel (Scene Objects)” on page 5-0 .

Example: `polygon = [0 0; 1 1; 2 0]`

Version History

Introduced in R2022a

See Also

Objects
`uavScenario` | `gpsSensor` | `insSensor` | `uavLidarPointCloudGenerator`

Functions
`addMesh`

Topics

"Design Obstacle Avoidance Package Delivery Scenario Using UAV Scenario Designer"

6

Scenes

Suburban scene

Suburban Unreal Engine environment

Description

The **Suburban scene** is an Unreal Engine environment of a suburban area that contains houses with backyards, trees, power poles, and vehicles on the road. The scene is rendered using the Unreal Engine from Epic Games.

Setup

To simulate a UAV flight in this scene:

1. Download the **Suburban scene** from the server.
2. Add a Simulation 3D Scene Configuration block to your Simulink model.
3. In this block, set the **Scene source** parameter to `Default Scenes`.
4. Set the enabled **Scene name** parameter to `Suburban scene`.

Examples

Simulate Simple Flight Scenario Using Suburban Scene Map in Unreal Engine Environment

This example shows how to download and access the Suburban scene map from the Simulation 3D Scene Configuration block. Then add a UAV vehicle and simulate a simple flight scenario in the Unreal Engine® simulation environment.

Download Suburban Scene Map

To begin, check the maps available in the server.

```
sim3d.maps.Map.server
```

```
        MapName                       Description                      Version    MinimumRelease
    ________________    __________________________________________    _______    ______________

    "Suburban scene"    "a suburban area beyond the city's border"      "1"         "R2022b"
```

Download the `Suburban scene` from the server.

```
sim3d.maps.Map.download("Suburban scene")
```

```
Map is successfully downloaded and is up-to-date
```

Check if the downloaded maps are available in your local machine.

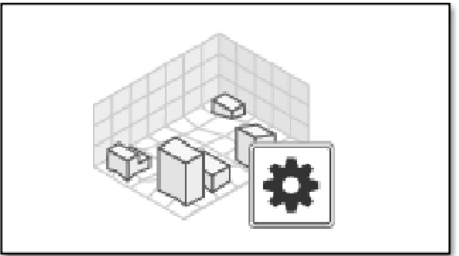
```
sim3d.maps.Map.local
```

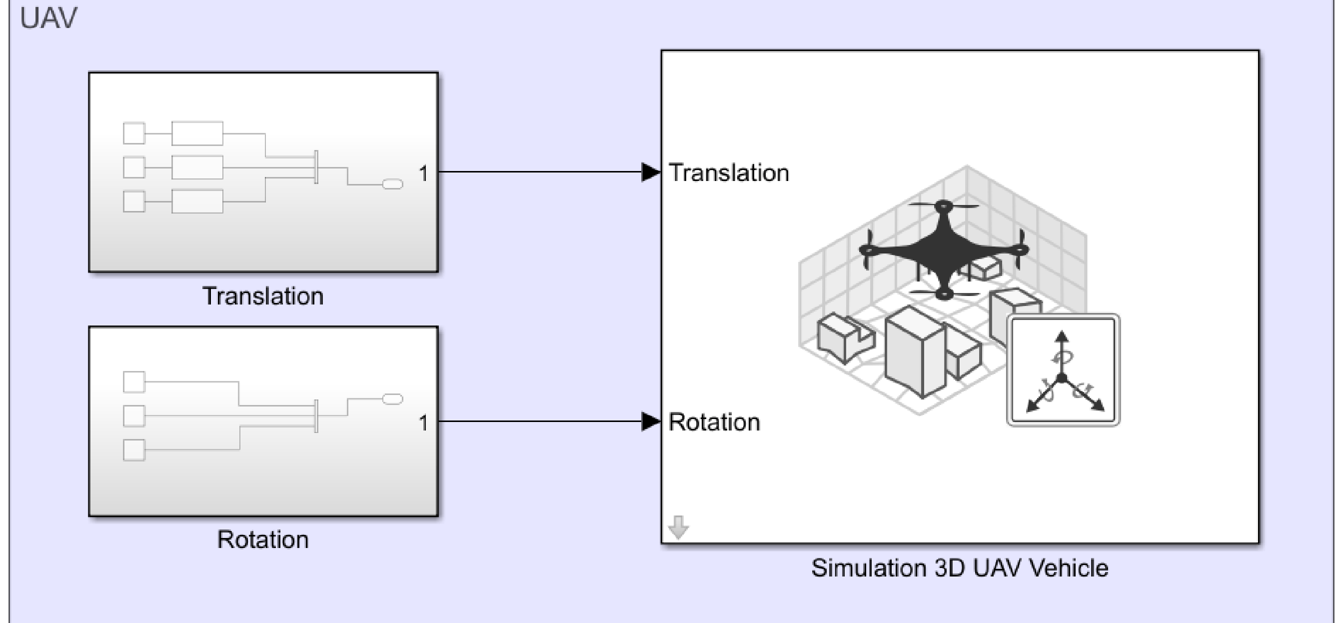
```
        MapName                       Description                      Version    MinimumRelease
    ________________    __________________________________________    _______    ______________

    "Suburban scene"    "a suburban area beyond the city's border"      "1"         "R2022b"
```

Add Suburban Scene Map to Simulink model

Open the Simulink model.

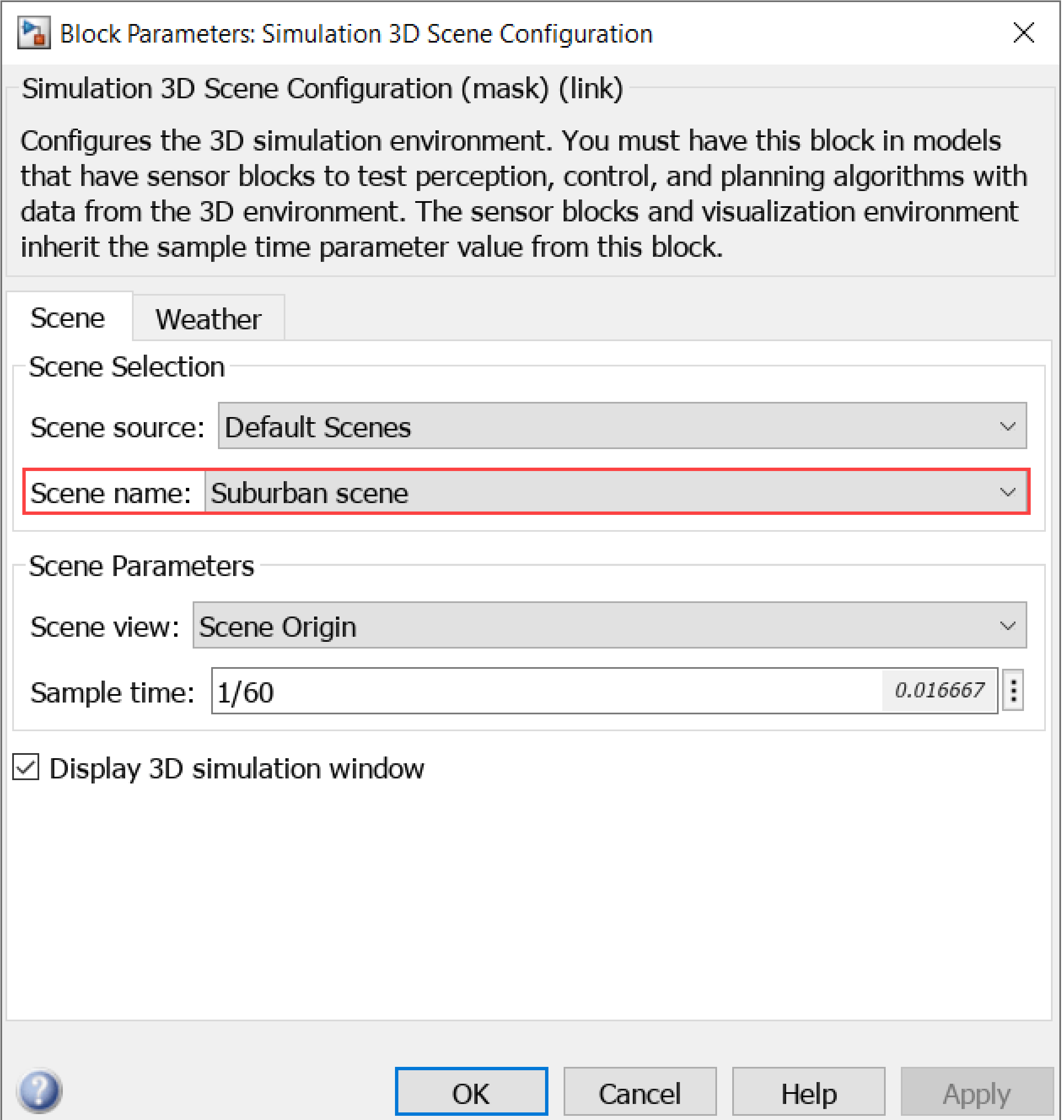
```
open_system("uavSuburbanScene")
```





Open the Simulation 3D Scene Configuration block mask and set **Scene name** to `Suburban scene`.

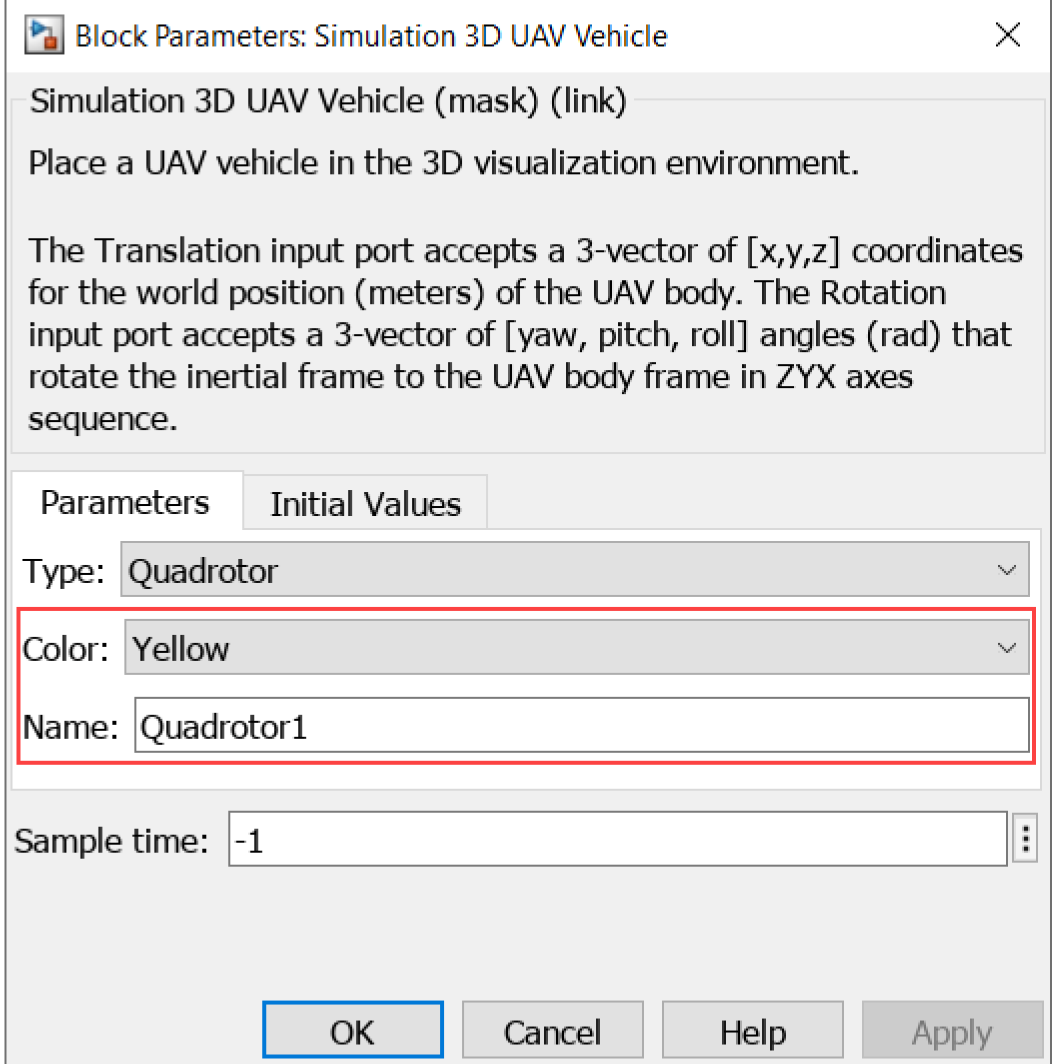
```
set_param("uavSuburbanScene/Simulation 3D Scene Configuration", "SceneDesc", "Suburban scene");
open_system("uavSuburbanScene/Simulation 3D Scene Configuration")
```



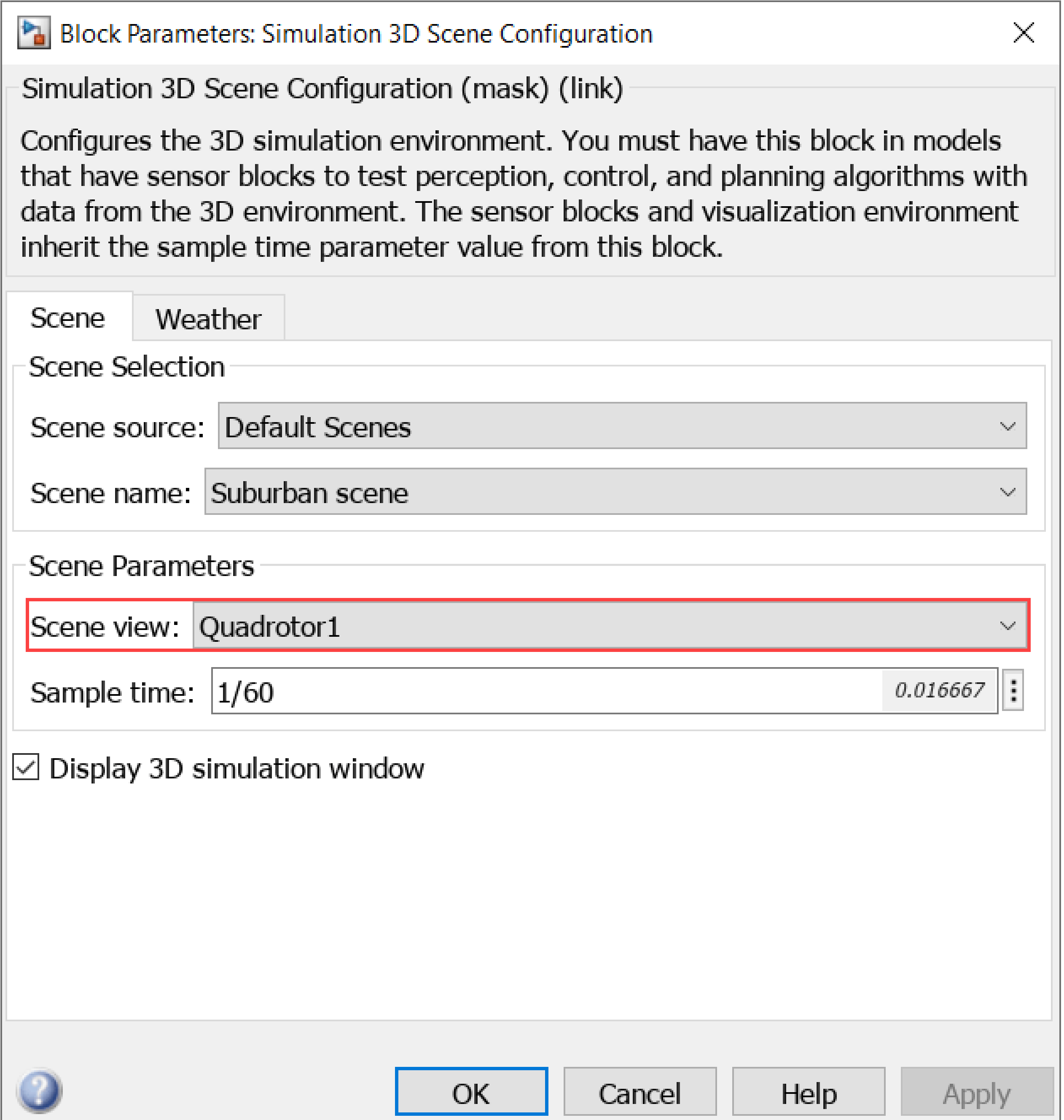
Add UAV Vehicle

Open the Simulation 3D UAV Vehicle block mask and set the color of the vehicle in **Color** to `Yellow` or any color of your choice. Specify the name of the vehicle in **Name** as `Quadrotor1`.

```
open_system("uavSuburbanScene/Simulation 3D UAV Vehicle")
```



To change the prespective of the scene to follow behind the UAV, set **Scene view** to `Quadrotor1` in Simulation 3D Scene Configuration block.



Simulate Model

When the simulation begins, it can take a few seconds for the visualization engine to initialize, especially when you are running it for the first time.

```
sim("uavSuburbanScene","StopTime","10");
```

Tips

- If you have the UAV Toolbox Interface for Unreal Engine Projects support package, then you can modify this scene. In the Unreal Engine project file that comes with the support package, this scene is named `Suburban scene`.

  For more details on customizing scenes, see "Customize Unreal Engine Scenes for UAVs".

Version History

Introduced in R2022b

See Also

Classes
`sim3d.Editor` | `sim3d.maps`

Functions
`sim3d.maps.Map.delete` | `sim3d.maps.Map.download` | `sim3d.maps.Map.local` | `sim3d.maps.Map.server`

Blocks
Simulation 3D Scene Configuration

Topics

“Coordinate Systems for Unreal Engine Simulation in UAV Toolbox”

US City Block

US city block Unreal Engine environment

Description

The **US City Block** scene is an Unreal Engine environment of a US city block that contains 15 intersections and 30 traffic lights. The scene is rendered using the Unreal Engine from Epic Games.

To simulate a UAV flight in this scene:

1. Add a Simulation 3D Scene Configuration block to your Simulink model.
2. In this block, set the **Scene source** parameter to `Default Scenes`.
3. Set the enabled **Scene name** parameter to `US city block`.

Intersections

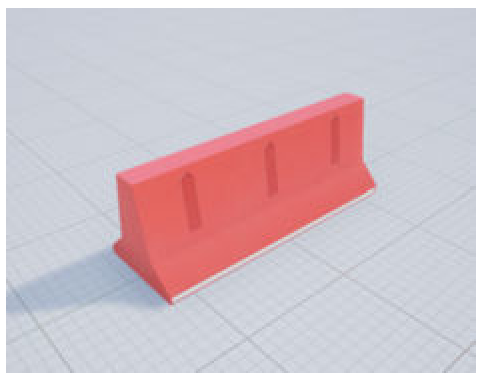
The US city block scene has 15 intersections, as indicated in this diagram.

This table provides the intersection locations in the world coordinate system. Dimensions are in m.

Intersection	Center Location		
	X (m)	Y (m)	Z (m)
1	-202.60	-108	.01
2	-112.60	-108	.01
3	-20.38	-108	.01
4	74.58	-108	.01
5	166.40	-108	.01
6	-184.60	0	.01
7	-112.60	0	.01
8	-20.34	0	.01
9	76.40	0	.01
10	166.46	0	.01
11	-184.60	110.50	.01
12	-112.60	110.50	.01

Intersection	Center Location		
	X (m)	Y (m)	Z (m)
13	-22.60	110.50	.01
14	76.40	110.50	.01
15	166.40	112.50	.01

Barrier



This table provides the object names and locations in the world coordinate system. Dimensions are in m.

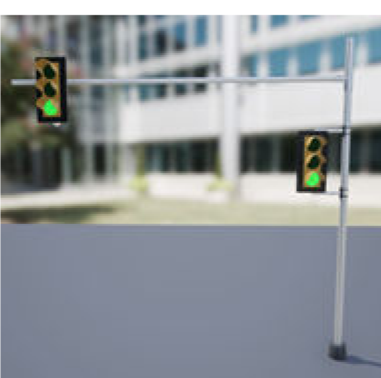
Object	Unreal Engine Editor Name	Location					
		X	Y	Z	Roll	Pitch	Yaw
Barrier	SM_Barrier	163.5	-146.95	0	0	0	90°
	SM_Barrier 2	166.35	-146.95	0	0	0	90°
	SM_Barrier 3	169.2	-146.95	0	0	0	90°
	SM_Barrier 7	163.5	150.15	0	0	0	90°
	SM_Barrier 8	166.35	150.15	0	0	0	90°
	SM_Barrier 9	169.2	150.15	0	0	0	90°
	SM_Barrier 11	197.05	109.65	0	0	0	-180°
	SM_Barrier 13	197.05	112.5	0	0	0	-180°
	SM_Barrier 14	197.05	115.34	0	0	0	-180°

Object	Unreal Engine Editor Name	Location					
		X	Y	Z	Roll	Pitch	Yaw
	SM_Barrier 18	197.05	-2.9	0	0	0	-180°
	SM_Barrier 19	197.05	-0.05	0	0	0	-180°
	SM_Barrier 20	197.05	2.8	0	0	0	-180°
	SM_Barrier 21	-240.5	107.65	0	0	0	-180°
	SM_Barrier 22	197.05	-110.9	0	0	0	-180°
	SM_Barrier 24	197.05	5.6	0	0	0	-180°
	SM_Barrier 27	197.05	-108.05	0	0	0	-180°
	SM_Barrier 28	197.05	-105.25	0	0	0	-180°
	SM_Barrier 31	-240.5	110.5	0	0	0	-180°
	SM_Barrier 32	-240.5	113.35	0	0	0	-180°
	SM_Barrier 36	-240.1	-2.9	0	0	0	-180°
	SM_Barrier 37	-240.1	-0.05	0	0	0	-180°
	SM_Barrier 38	-240.1	2.8	0	0	0	-180°
	SM_Barrier 43	-242.15	110.9	0	0	0	-180°
	SM_Barrier 44	-242.15	-108.05	0	0	0	-180°
	SM_Barrier 45	-242.15	-105.25	0	0	0	-180°
	SM_Barrier 48	73.4	150.15	0	0	0	90°
	SM_Barrier 49	76.25	150.15	0	0	0	90°
	SM_Barrier 50	79.1	150.15	0	0	0	90°
	SM_Barrier 54	-25.55	150.15	0	0	0	90°

Object	Unreal Engine Editor Name	Location					
		X	Y	Z	Roll	Pitch	Yaw
	SM_Barrier 55	-22.7	150.15	0	0	0	90°
	SM_Barrier 56	-19.85	150.15	0	0	0	90°
	SM_Barrier 59	-115.3	150.15	0	0	0	90°
	SM_Barrier 60	-112.45	150.15	0	0	0	90°
	SM_Barrier 61	-109.6	150.15	0	0	0	90°
	SM_Barrier 66	69.25	-147.35	0	0	0	90°
	SM_Barrier 68	75.45	-147.5	0.15	0	0	90°
	SM_Barrier 69	72.45	-147.5	0.15	0	0	90°
	SM_Barrier 70	-25.55	-146.45	0	0	0	90°
	SM_Barrier 71	-22.15	-146.45	0	0	0	90°
	SM_Barrier 72	-18.65	-146.45	0	0	0	90°
	SM_Barrier 75	-115.3	-147.6	0	0	0	90°
	SM_Barrier 76	-112.45	-147.6	0	0	0	90°
	SM_Barrier 77	-109.6	-147.6	0	0	0	90°
	SM_Barrier 84	-15.45	-146.45	0	0	0	90°
	SM_Barrier 88	-187.5	150.15	0	0	0	90°
	SM_Barrier 89	-184.65	150.15	0	0	0	90°
	SM_Barrier 90	-181.8	150.15	0	0	0	90°
	SM_Barrier 94	-205.6	-147.4	0	0	0	90°
	SM_Barrier 95	-202.75	-147.4	0	0	0	90°

Object	Unreal Engine Editor Name	Location					
		X	Y	Z	Roll	Pitch	Yaw
	SM_Barrier 96	-199.9	-147.4	0	0	0	90°
	SM_Barrier 101	44.15	3.05	0	0	0	-50°
	SM_Barrier 102	39.15	0.55	0	0	0	-90°
	SM_Barrier 103	41.95	1.3	0	0	0	-50°
	SM_Barrier 104	36.5	.55	0	0	0	-90°
	SM_Barrier 105	33.85	.55	0	0	0	-90°
	SM_Barrier 106	31.2	.55	0	0	0	-90°
	SM_Barrier 107	28.45	.55	0	0	0	-90°
	SM_Barrier 108	25.8	.55	0	0	0	-90°
	SM_Barrier 109	23.15	.55	0	0	0	-90°
	SM_Barrier 110	20.5	.55	0	0	0	-90°
	SM_Barrier 111	17.95	.55	0	0	0	-90°
	SM_Barrier 112	15.3	.55	0	0	0	-90°
	SM_Barrier 113	12.65	.55	0	0	0	-90°
	SM_Barrier 114	10.0	.55	0	0	0	-90°
	SM_Barrier 115	7.01	1.38	0	0	0	-125°
	SM_Barrier 116	4.75	3.05	0	0	0	-125°

Traffic Lights



The US City Scene contains 30 traffic lights, two at each of the 15 intersections. Each intersection has a traffic light group. If you have the “Customize Unreal Engine Scenes for UAVs” for customizing scenes, you can control the timing of the traffic lights.

This table provides the traffic light names and locations in the world coordinate system. Dimensions are in m. Only one of the traffic lights in the group can be green at a time. The traffic lights are green for 10 s and yellow for 3 s. At the start of the simulation, the first traffic lights in the group are green (for example, `SM_TrafficLights1_3` and `SM_TrafficLights2_4`). The second lights in the group are red (for example, `SM_TrafficLights1_4` and `SM_TrafficLights2_3`).

Intersection	Unreal Engine Editor Name		Location					
	Traffic Light Group	Traffic Light	X	Y	Z	Roll	Pitch	Yaw
1	TrafficLightGroup	SM_TrafficLights1_3	-196.55	-100.65	0	0	0	-90°
		SM_TrafficLights1_4	-210.20	-113.40	0	0	0	0
2	TrafficLightGroup2	SM_TrafficLights2_4	-120.40	-113.50	0	0	0	0
		SM_TrafficLights2_3	-106.35	98.35	0	0	0	-90°
3	TrafficLightGroup3	SM_TrafficLights3_1	-13.10	-116.20	0.2	0	0	90°

Intersection	Unreal Engine Editor Name		Location					
	Traffic Light Group	Traffic Light	X	Y	Z	Roll	Pitch	Yaw
		SM_TrafficLights3_4	-30.60	-113.80	0	0	0	0
4	TrafficLightGroup4	SM_TrafficLights4_4	64.80	-113.0	0	0	0	0
		SM_TrafficLights4_3	71.40	-100.30	0	0	0	-100°
5	TrafficLightGroup5	SM_TrafficLights5_1	171.50	-115.70	0	0	0	90°
		SM_TrafficLights5_4	157.40	-113.50	0	0	0	0
6	TrafficLightGroup6	SM_TrafficLights6_3	-189.60	7.40	0	0	0	-90°
		SM_TrafficLights6_2	-177.30	5.70	0	0	0	180°
7	TrafficLightGroup7	SM_TrafficLights7_3	-117.80	7.70	0.2	0	0	-90°
		SM_TrafficLights7_2	-105.20	5.50	0	0	0	180°
8	TrafficLightGroup8	SM_TrafficLights8_2	-10.90	5.60	0	0	0	180°

Intersection	Unreal Engine Editor Name		Location					
	Traffic Light Group	Traffic Light	X	Y	Z	Roll	Pitch	Yaw
		SM_TrafficLights8_1	-13.10	-7.60	0.1	0	0	90°
9	TrafficLightGroup9	SM_TrafficLights9_3	70.90	9.20	0	0	0	-90°
		SM_TrafficLights9_2	85.90	7.60	0.2	0	0	180°
10	TrafficLightGroup10	SM_TrafficLights10_2	173.70	7.50	0	0	0	180°
		SM_TrafficLights10_1	172.10	-7.70	0	0	0	90°
11	TrafficLightGroup11	SM_TrafficLights11_3	-189.80	118.45	0	0	0	-90°
		SM_TrafficLights11_4	-191.05	104.55	0	0	0	0
12	TrafficLightGroup12	SM_TrafficLights12_4	-120.50	105.40	0	0	0	0
		SM_TrafficLights12_3	-117.60	117.60	0	0	0	-90°
13	TrafficLightGroup13	SM_TrafficLights13_1	-12.80	102.50	0	0	0	90°

Intersection	Unreal Engine Editor Name		Location					
	Traffic Light Group	Traffic Light	X	Y	Z	Roll	Pitch	Yaw
		SM_TrafficLights13_4	-30.50	105.30	0	0	0	0
14	TrafficLightGroup14	SM_TrafficLights14_4	69.30	105.30	0	0	0	0
		SM_TrafficLights14_3	70.90	118.70	0	0	0	-90°
15	TrafficLightGroup15	SM_TrafficLights15_1	171.40	105.20	0	0	0	90°
		SM_TrafficLights15_4	158.40	107.20	0	0	0	0

Tips

- If you have the UAV Toolbox Interface for Unreal Engine Projects support package, then you can modify this scene. In the Unreal Engine project file that comes with the support package, this scene is named `USCityBlock`.

  For more details on customizing scenes, see “Customize Unreal Engine Scenes for UAVs”.

See Also

Simulation 3D Scene Configuration | Suburban scene

Topics

“Stereo Visual SLAM for UAV Navigation in 3D Simulation”
“Coordinate Systems for Unreal Engine Simulation in UAV Toolbox”

Vehicles

Quadrotor

Quadrotor vehicle dimensions

Description

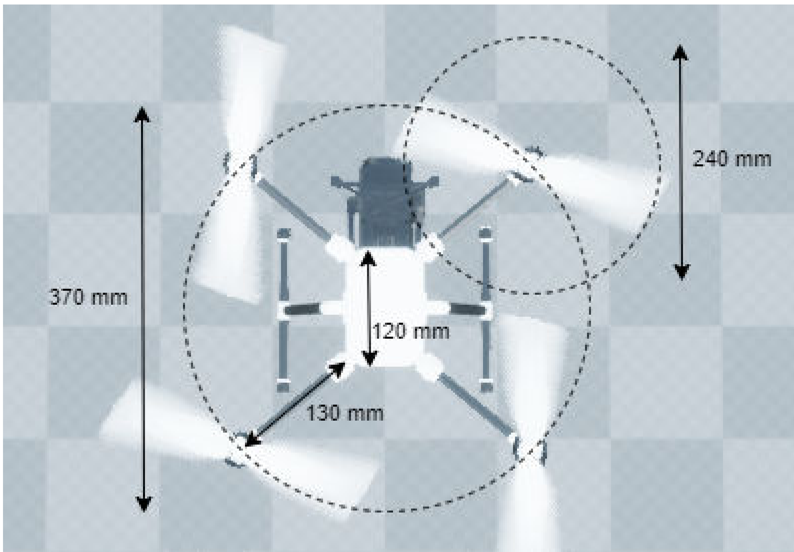
Quadrotor is one of the UAVs that you can use within the Unreal Engine simulation environment. This environment is rendered using the Unreal Engine from Epic Games. The origin is located at the center of the camera gimbal located on the underside of the aircraft. For detailed specifications of the vehicle dimensions , see the **Dimensions** section.

To add this type of vehicle to the Unreal Engine simulation environment:

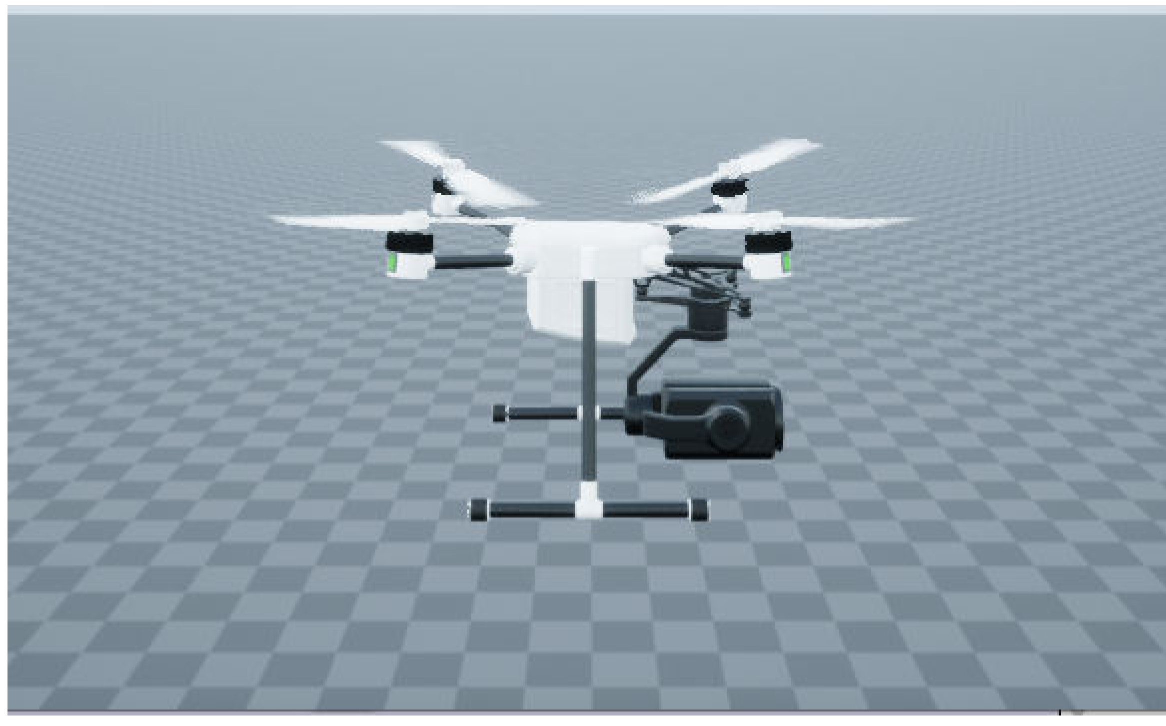
1. Add a Simulation 3D UAV Vehicle block to your Simulink model.
2. In the block, set the **Type** parameter to `Quadrotor`.

Dimensions

`Top-down view` — Vehicle width dimensions
diagram



`Side view` — Vehicle length, front overhang, and rear overhang dimensions
diagram



`Front view` — Tire width and front axle dimensions
diagram

`Rear view` — Vehicle height and rear axle dimensions
diagram

See Also

Simulation 3D UAV Vehicle | Simulation 3D Scene Configuration

Topics

"Unreal Engine Simulation for Unmanned Aerial Vehicles"
"Coordinate Systems for Unreal Engine Simulation in UAV Toolbox"

Fixed Wing Aircraft

Fixed wing aircraft dimensions

Description

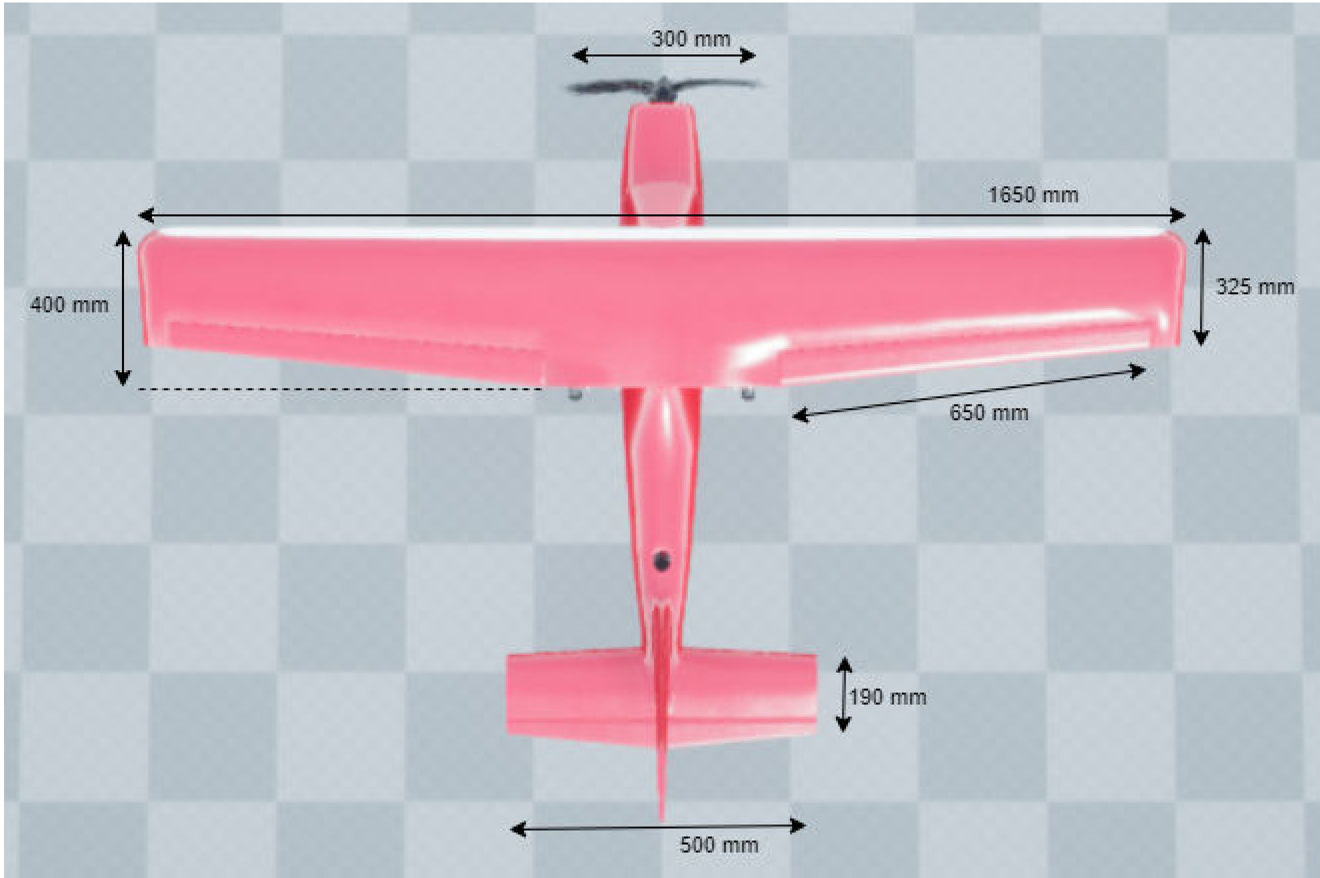
Fixed Wing Aircraft is one of the vehicles that you can use within the Unreal Engine simulation environment. This environment is rendered using the Unreal Engine from Epic Games. The origin is located at the center of the camera gimbal located on the underside of the aircraft. For detailed specifications of the vehicle dimensions , see the **Dimensions** section.

To add this type of vehicle to the 3D simulation environment:

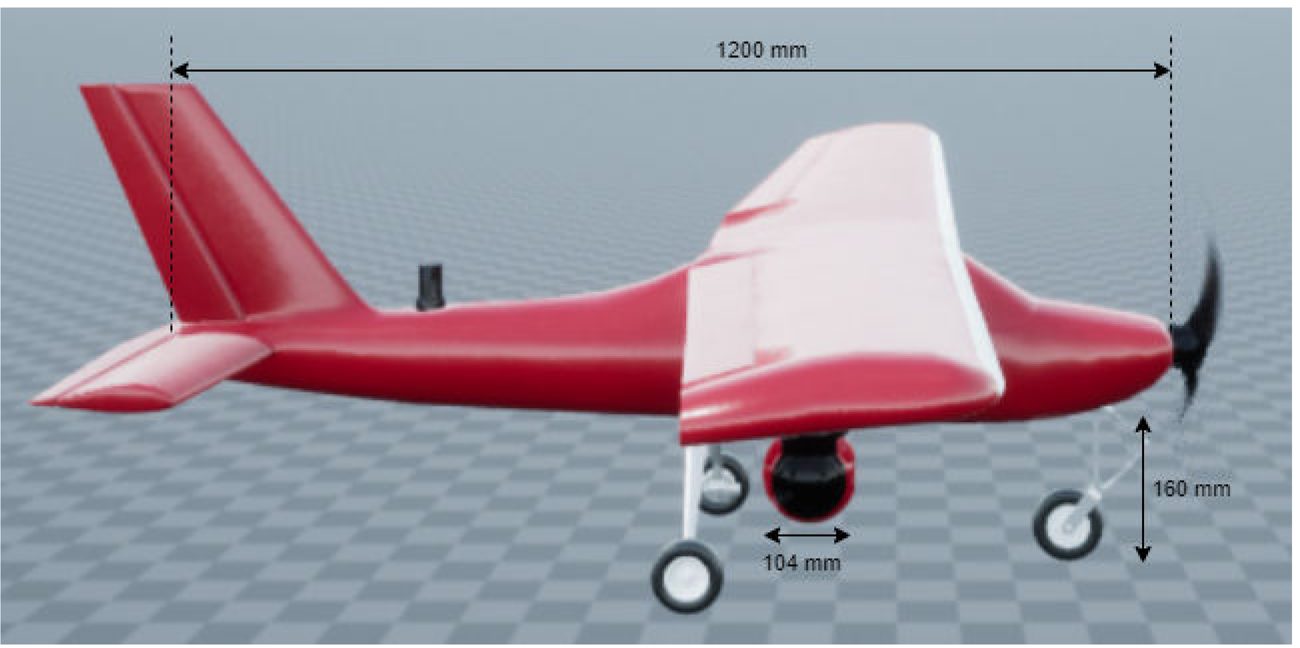
1. Add a Simulation 3D UAV Vehicle block to your Simulink model.
2. In the block, set the **Type** parameter to `Fixed wing`.

Dimensions

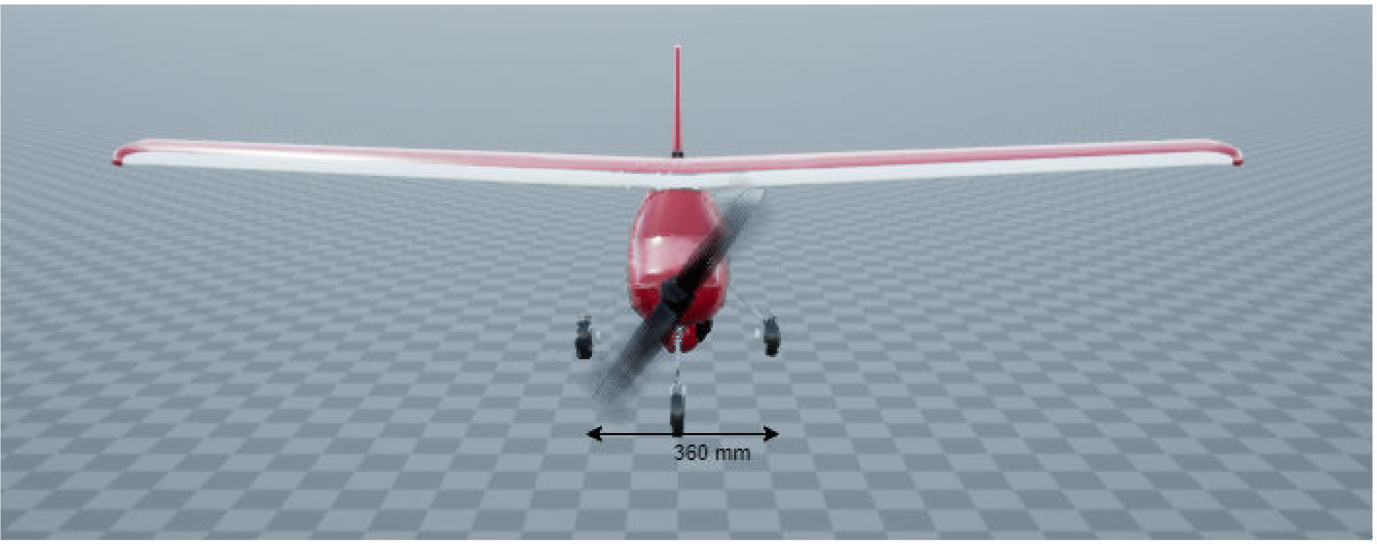
`Top-down view` — Vehicle width dimensions
diagram



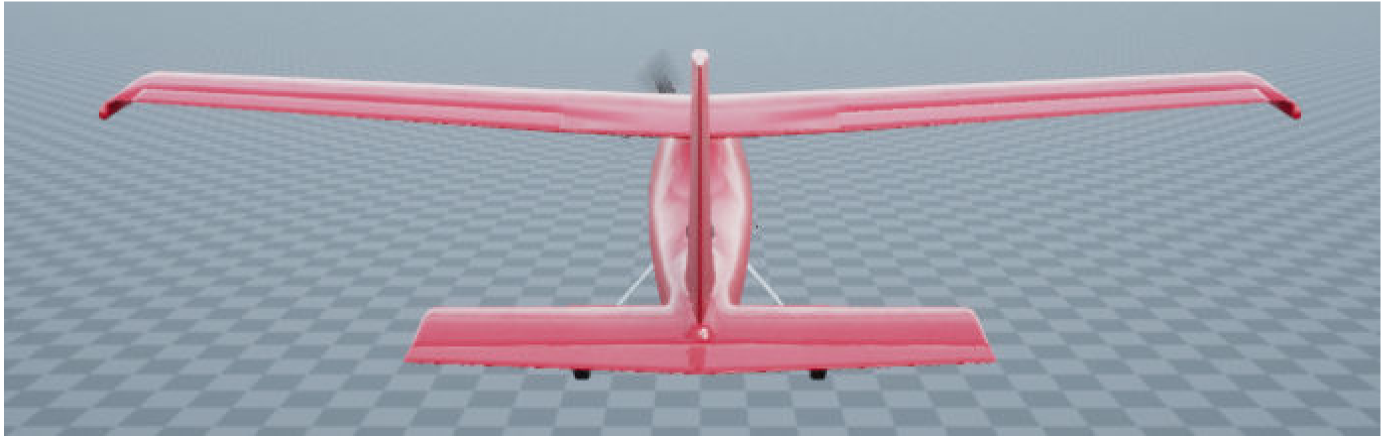
`Side view` — Vehicle length, landing gear height, and camera dimensions diagram



`Front view` — Tire width dimensions
diagram



`Rear view` — Vehicle height and rear axle dimensions
diagram



See Also

Simulation 3D UAV Vehicle | Simulation 3D Scene Configuration

Topics

"How Unreal Engine Simulation for UAVs Works"
"Coordinate Systems for Unreal Engine Simulation in UAV Toolbox"