

UAV Toolbox

Reference



MATLAB® & SIMULINK®

R2021a



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 2020–2021 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)

1	Classes
2	Methods
3	Functions
4	Blocks
5	Apps
6	Scenes
7	Vehicles

Classes

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.

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

Examples

Create and Translate Cuboid Mesh

This example shows how to create an `extendedObjectMesh` object and translate the object.

Construct a cuboid mesh.

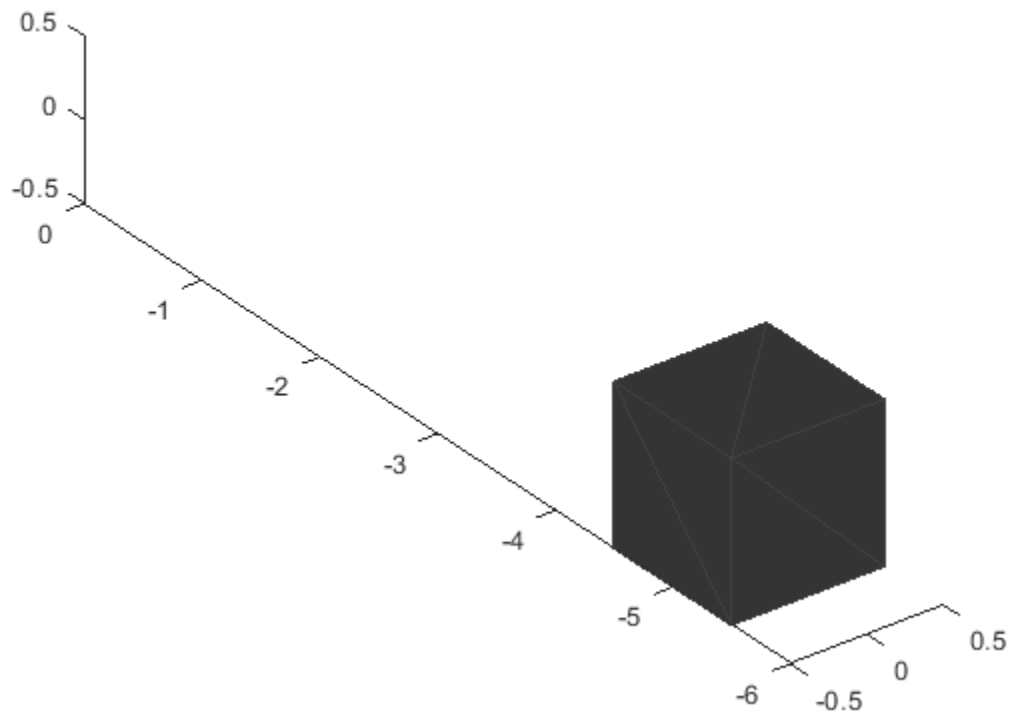
```
mesh = extendedObjectMesh('cuboid');
```

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

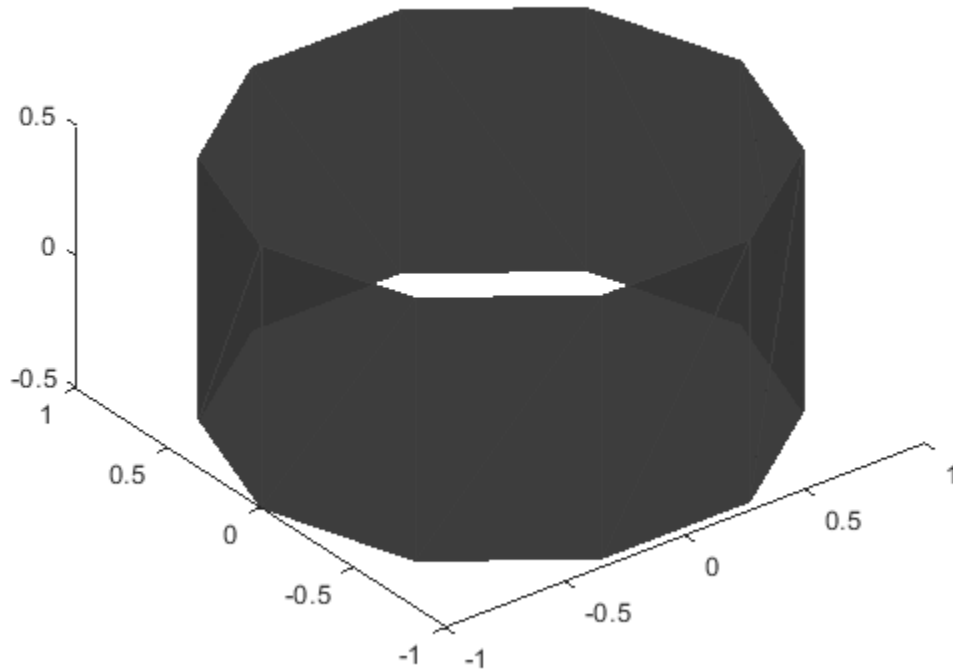
This example shows how to create an `extendedObjectMesh` object and visualize the object.

Construct a cylinder mesh.

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

Visualize the mesh.

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

Create and Auto-scale Sphere Mesh

This example shows how to create an `extendedObjectMesh` object and auto-scale the object to the required dimensions.

Construct a sphere mesh of unit dimensions.

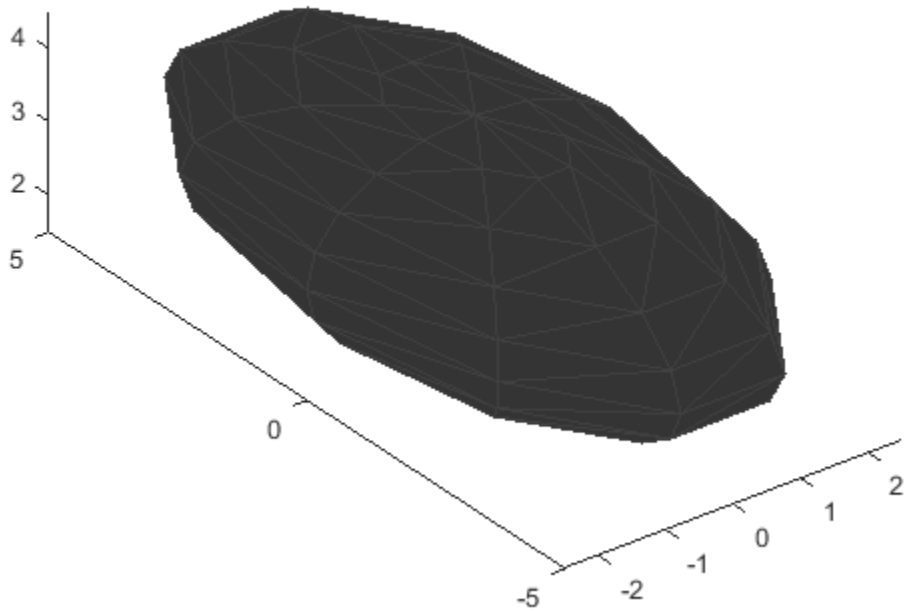
```
sph = extendedObjectMesh('sphere');
```

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);
```



See Also

Introduced in R2020b

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'.

Data Type — 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.

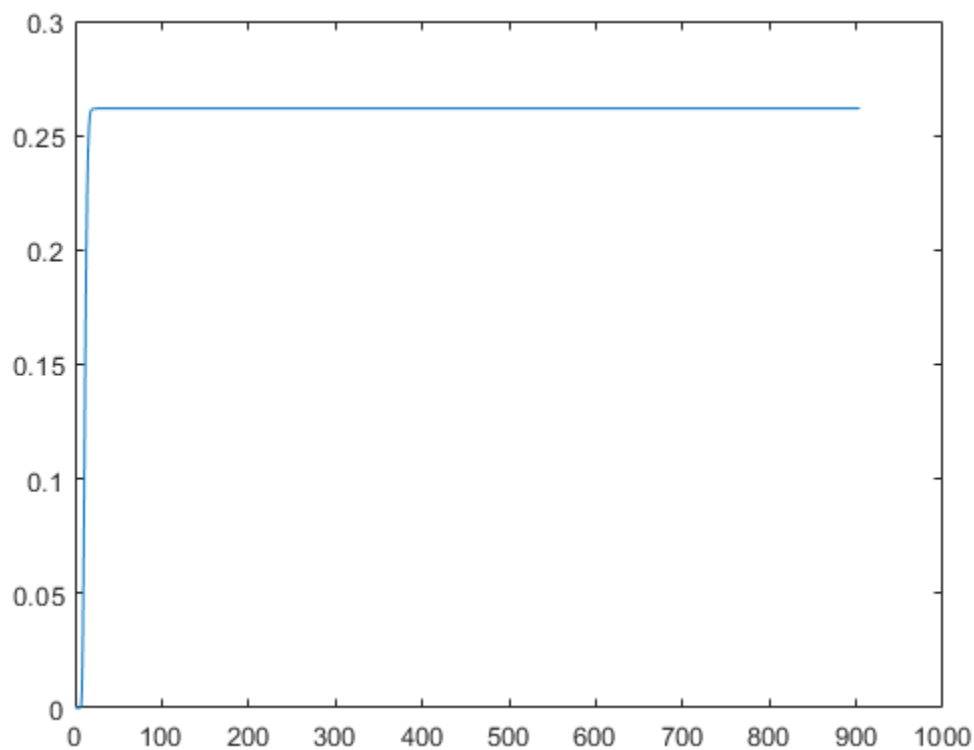
```
simOut = ode45(@(~,x)derivative(model,x,u,e), [0 50], s);
size(simOut.y)
```

```
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.

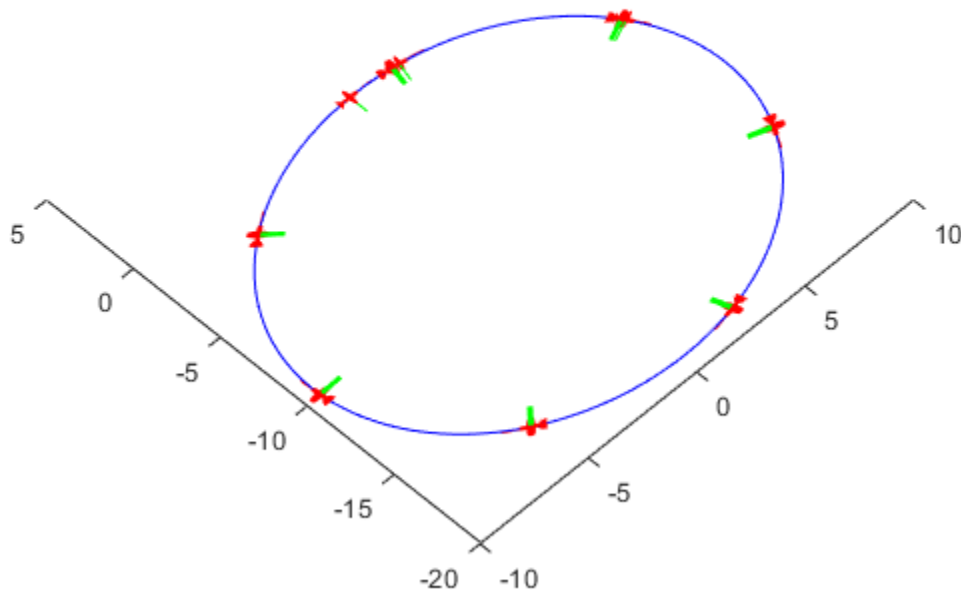
```
plot(simOut.y(7,:))
```



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:

$$\begin{aligned}\dot{x}_e &= V_g \cos \chi \cos \gamma \\ \dot{y}_e &= V_g \sin \chi \cos \gamma \\ \dot{h} &= V_g \sin \gamma \\ \dot{\chi} &= \frac{g \cos(\chi - \psi)}{V_g} \tan \phi \\ V_g \sin(\gamma^c) &= \min(\max(k_h(h^c - h), -V_g), V_g) \\ \dot{\gamma} &= k_\gamma(\gamma^c - \gamma) \\ \dot{V}_a &= k_{V_a}(V_a^c - V_a) \\ \frac{g \cos(\chi - \psi)}{V_g} \tan(\phi^c) &= k_\chi(\chi^c - \chi) \\ \ddot{\phi} &= k_{P\phi}(\phi^c - \phi) + k_{D\phi}(-\dot{\phi})\end{aligned}$$

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:

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

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

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

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

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"

Introduced in R2018b

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 1-14 and Predefined Plots on page 1-15 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 1-15 or call `updatePlot`.

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

Data Types: string

Object Functions

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

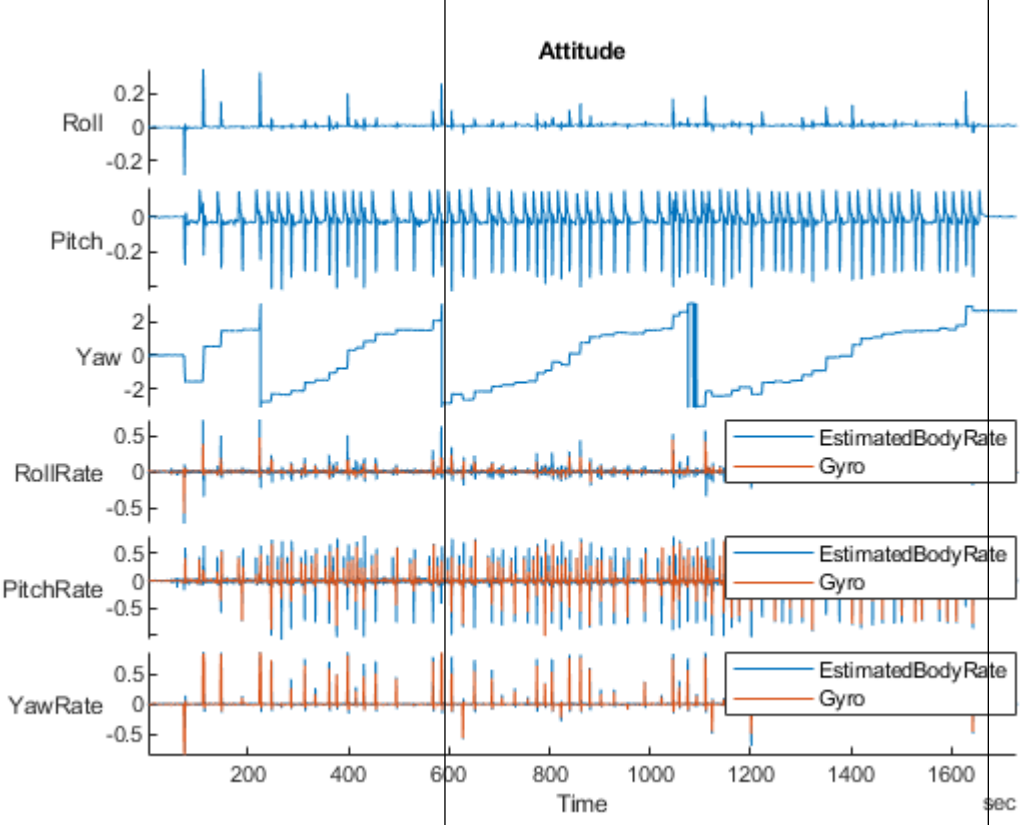
Signal Name	Description	Fields	Unit
Accel#	Raw magnetometer reading from IMU sensor	[ax ay az]	m/s ²
Airspeed#	Airspeed reading of pressure differential, indicated air speed, and temperature	[PressDiff, AirSpeed, Temp]	Pa, m/s, °C
AttitudeEuler	Attitude of UAV in Euler (ZYX) form	[Roll, Pitch, Yaw]	radi
AttitudeRate	Angular velocity along each body axis	[xRotRate, yRotRate, zRotRate]	rad/s
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, m, °C
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, m/s, deg
Gyro#	Raw body angular velocity readings from IMU sensor	[GyroX, GyroY, GyroZ]	rad/s
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

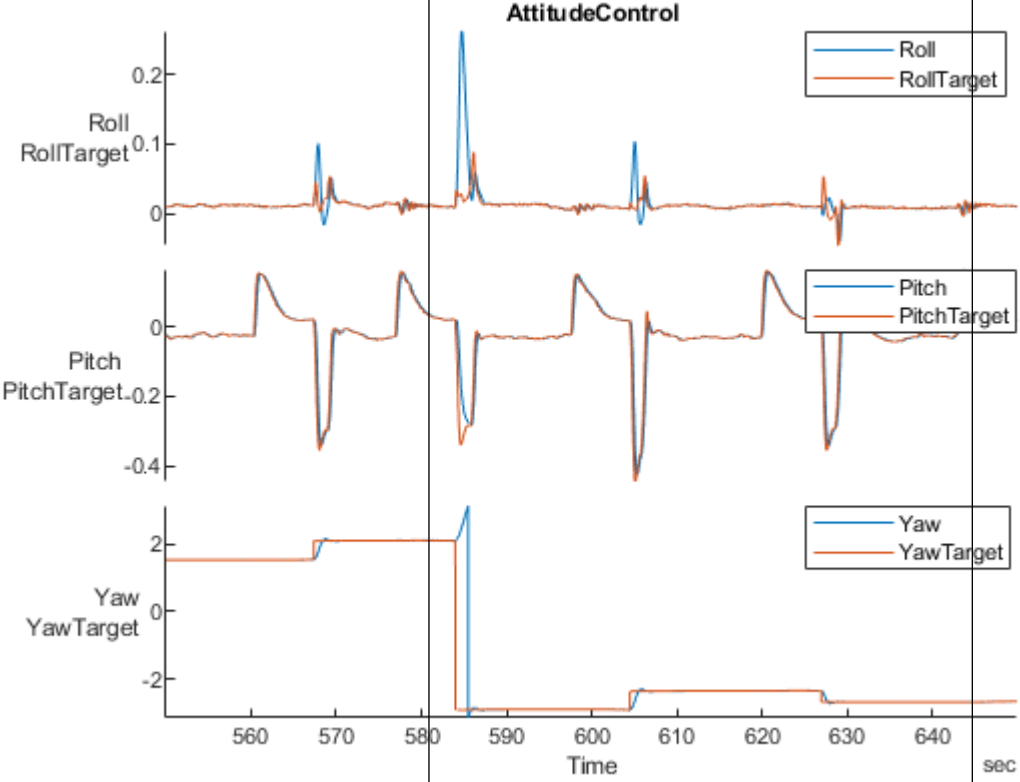
Predefined Plots

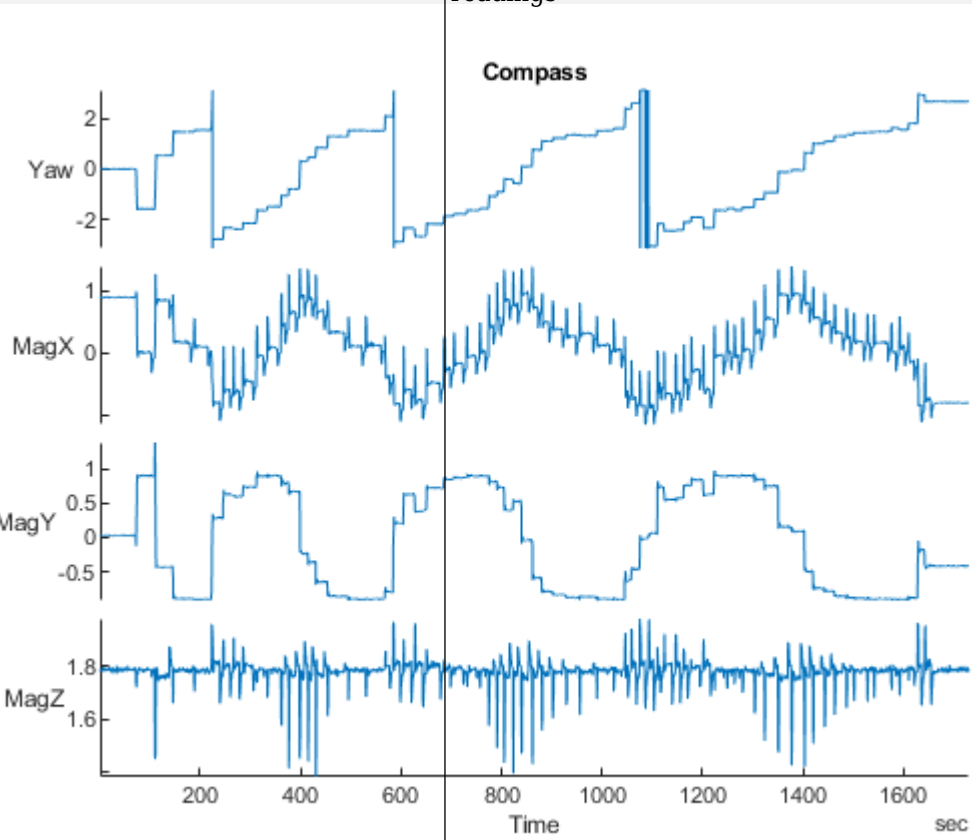
After mapping signals to the list of predefined signals using `mapSignal`, specific plots are made available when calling `show`. To view a list of available plots and their associated signals for your specific object, call `info(mapper, "Plot")`. If you want to define custom plots based on signals,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

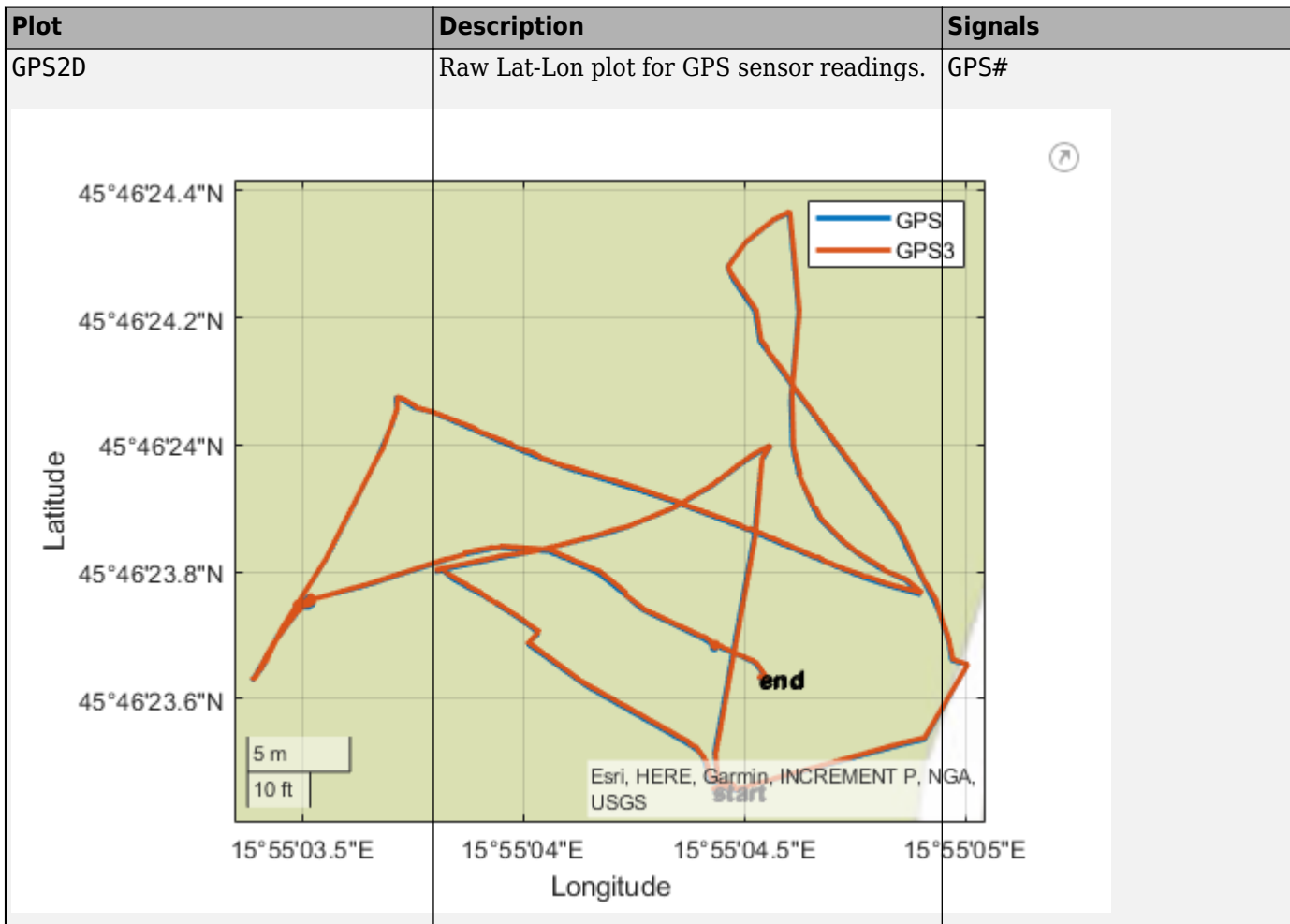
Each predefined plot has a set of required signals that must be mapped.

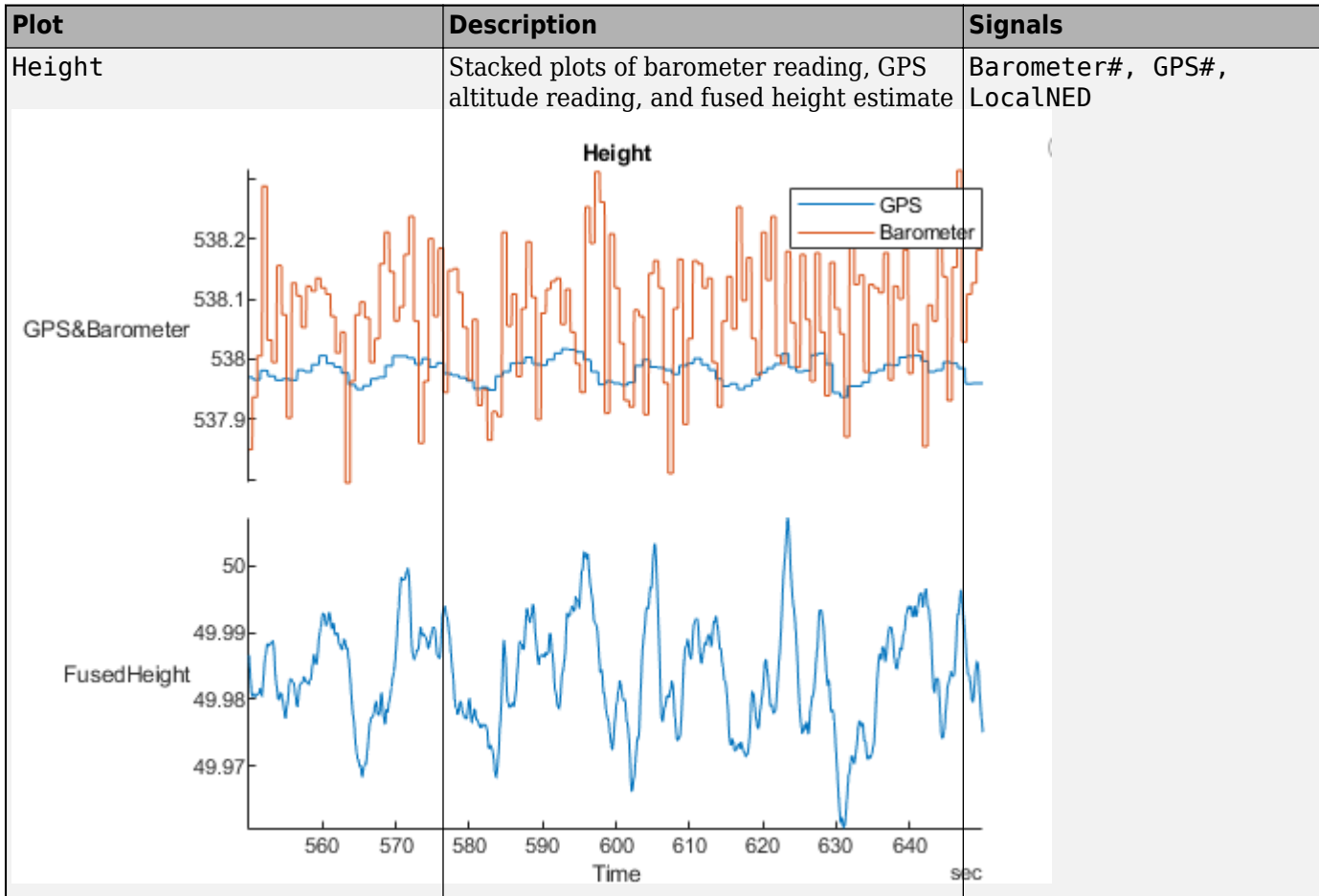
Predefined Plots

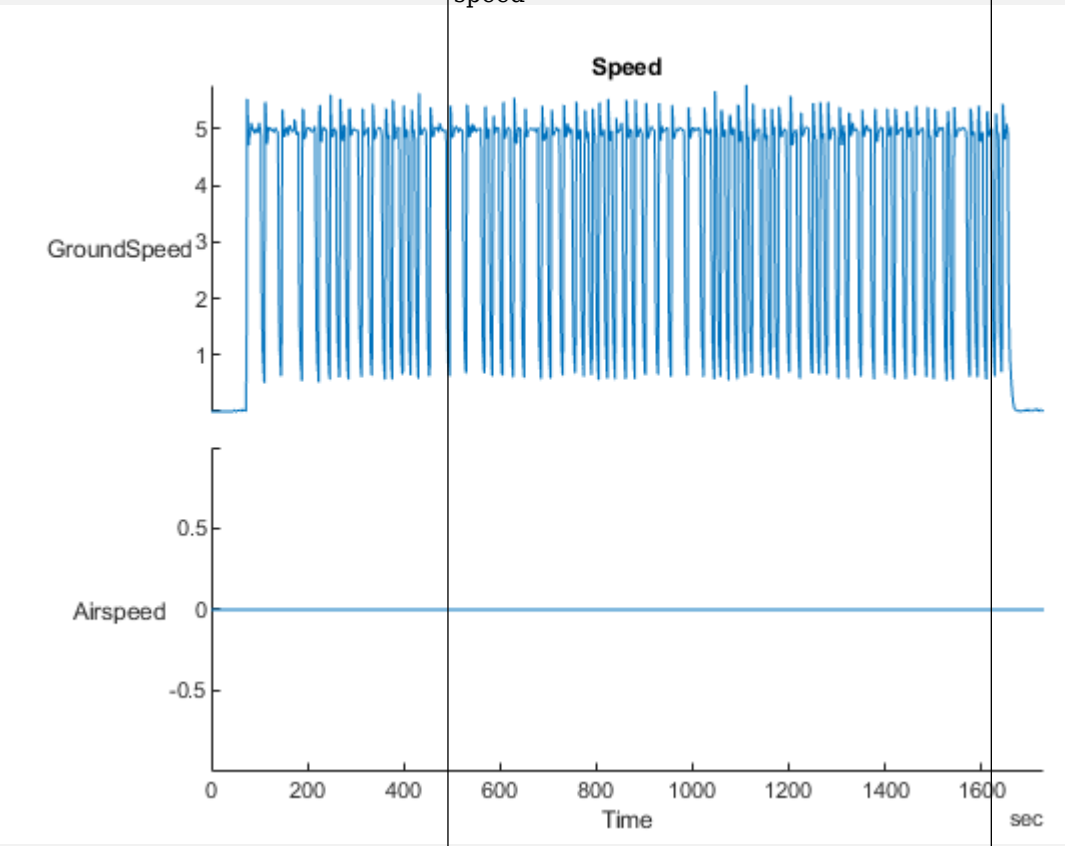
Plot	Description	Signals
<p data-bbox="240 346 375 380">Attitude</p> 	<p data-bbox="691 346 1232 409">Stacked plot of roll, pitch, yaw angles and body rotation rates</p>	<p data-bbox="1232 346 1601 409">AttitudeEuler, AttitudeRate, Gyro#</p>

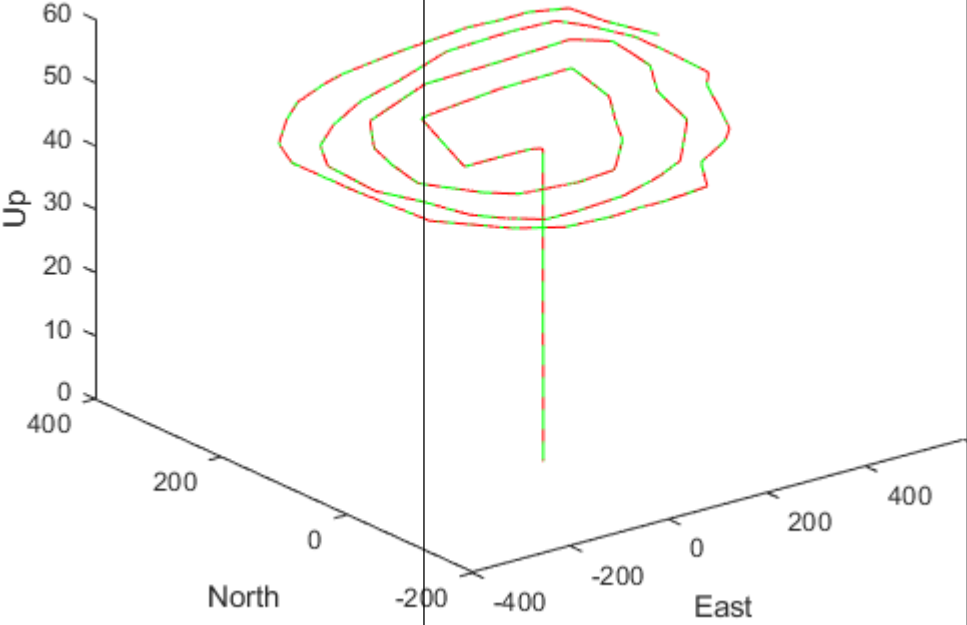
Plot	Description	Signals
<p data-bbox="240 296 483 327">AttitudeControl</p>  <p data-bbox="771 409 950 441">AttitudeControl</p> <p data-bbox="284 514 349 546">Roll</p> <p data-bbox="284 556 381 588">RollTarget</p> <p data-bbox="284 745 349 777">Pitch</p> <p data-bbox="284 787 381 819">PitchTarget</p> <p data-bbox="284 987 349 1018">Yaw</p> <p data-bbox="284 1029 381 1060">YawTarget</p> <p data-bbox="828 1144 885 1176">Time</p> <p data-bbox="1242 1144 1282 1176">sec</p>	<p data-bbox="691 296 1230 367">Estimated attitude of UAV and the attitude target set point</p>	<p data-bbox="1230 296 1604 367">AttitudeEuler, AttitudeTargetEuler</p>
<p data-bbox="240 1247 349 1278">Battery</p>	<p data-bbox="691 1247 1015 1278">Battery consumption plot</p>	<p data-bbox="1230 1247 1356 1278">Battery</p>

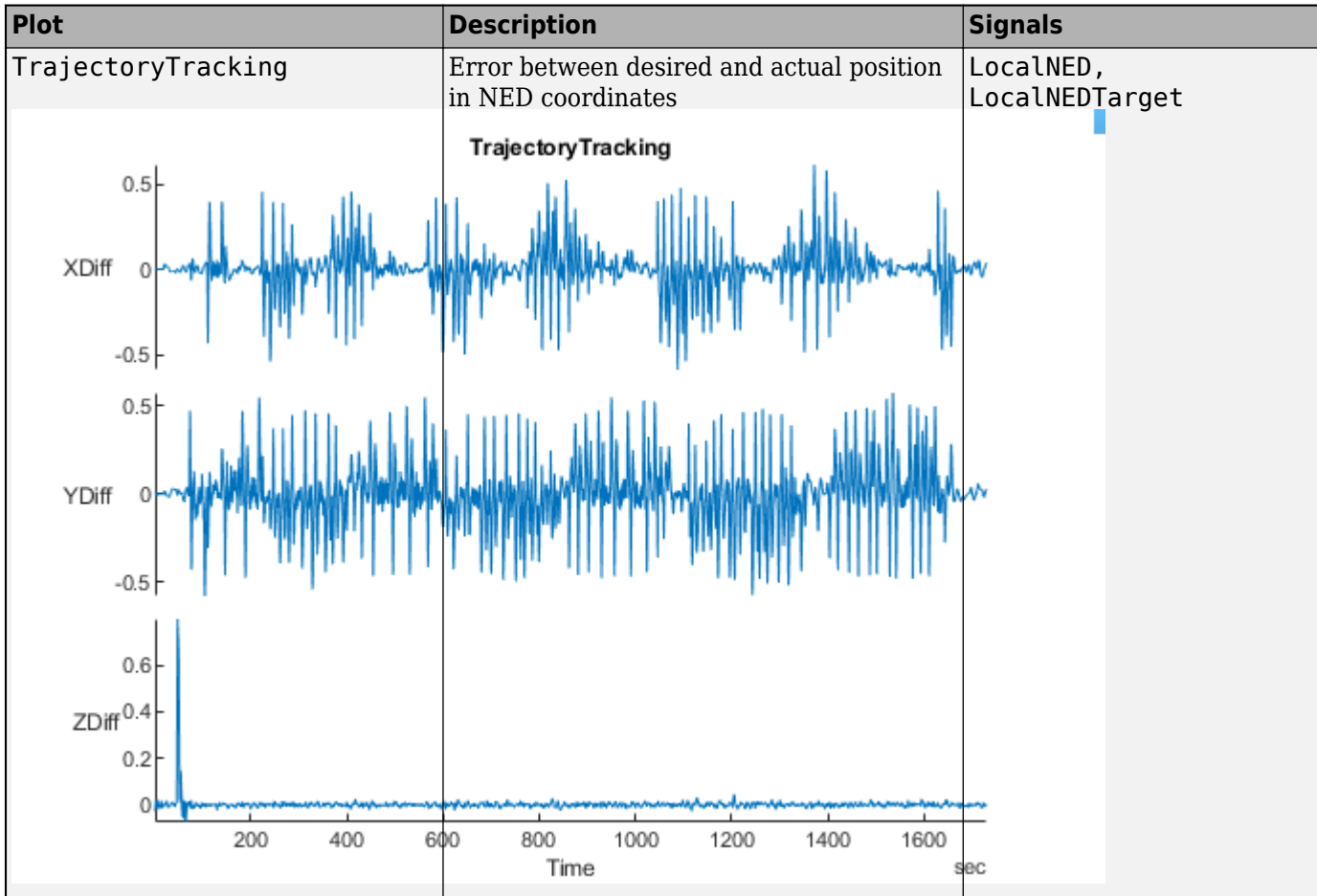
Plot	Description	Signals
<p data-bbox="240 296 357 327">Compass</p> 	<p data-bbox="691 296 1117 359">Estimated yaw and magnetometer readings</p>	<p data-bbox="1232 296 1559 359">AttitudeEuler, Mag#, GPS#</p>

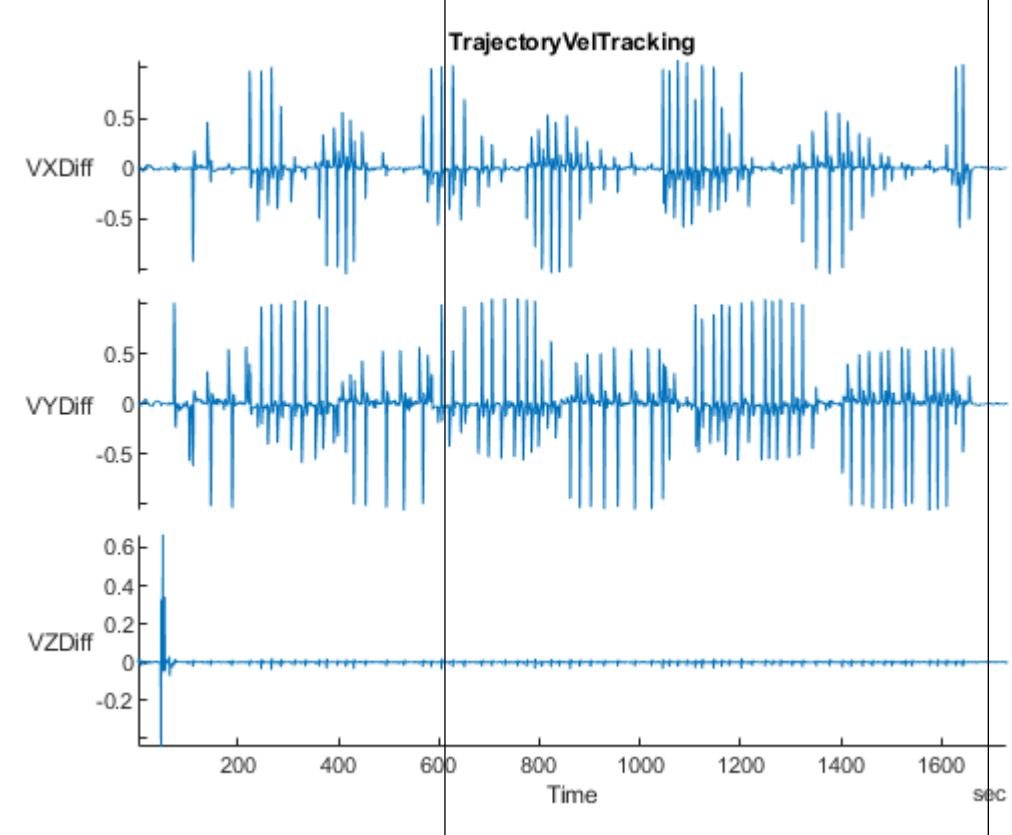




Plot	Description	Signals
<p data-bbox="240 296 324 327">Speed</p>  <p>The figure is a stacked line plot titled "Speed". The x-axis is labeled "Time" and ranges from 0 to 1600 seconds with major ticks every 200 units. The top y-axis is labeled "GroundSpeed" and ranges from 0 to 5 with major ticks every 1 unit. The bottom y-axis is labeled "Airspeed" and ranges from -0.5 to 0.5 with major ticks at -0.5, 0, and 0.5. The "GroundSpeed" signal is a blue line that starts at approximately 0.5, jumps to about 5.0 at 50 seconds, and then exhibits high-frequency oscillations between approximately 0.5 and 5.5 until 1650 seconds, where it drops back to 0.5. The "Airspeed" signal is a blue horizontal line at 0.0 for the entire duration.</p>	<p data-bbox="691 296 1230 359">Stacked plot of ground velocity and air speed</p>	<p data-bbox="1230 296 1602 327">GPS#, Airspeed#</p>

Plot	Description	Signals
<p>Trajectory</p> 	<p>Trajectory in local coordinates versus target set points</p>	<p>LocalNED, LocalNEDTarget</p>



Plot	Description	Signals
<p>TrajectoryVelTracking</p> 	<p>Error between desired and actual velocity in NED coordinates</p>	<p>LocalNEDVel, LocalNEDVelTarget</p>

See Also

mavlinktlog

Introduced in R2020b

gpsSensor

GPS receiver simulation model

Description

The `gpsSensor` System object™ models data output from a Global Positioning System (GPS) receiver.

To model a GPS receiver:

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

To learn more about how System objects work, see [What Are System Objects?](#).

Creation

Syntax

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

Description

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

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

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

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

Properties

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

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

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

SampleRate — Update rate of receiver (Hz)

1 (default) | positive real scalar

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

Data Types: `single` | `double`

ReferenceLocation — Origin of local navigation reference frame

[0 0 0] (default) | [degrees degrees meters]

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

Data Types: `single` | `double`

PositionInputFormat — Position coordinate input format

'Local' (default) | 'Geodetic'

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

- If you set the property as 'Local', then you need to specify the `truePosition` input as Cartesian coordinates with respect to the local navigation frame whose origin is fixed and defined by the `ReferenceLocation` 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: YesData 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: YesData 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

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

Examples

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;
```

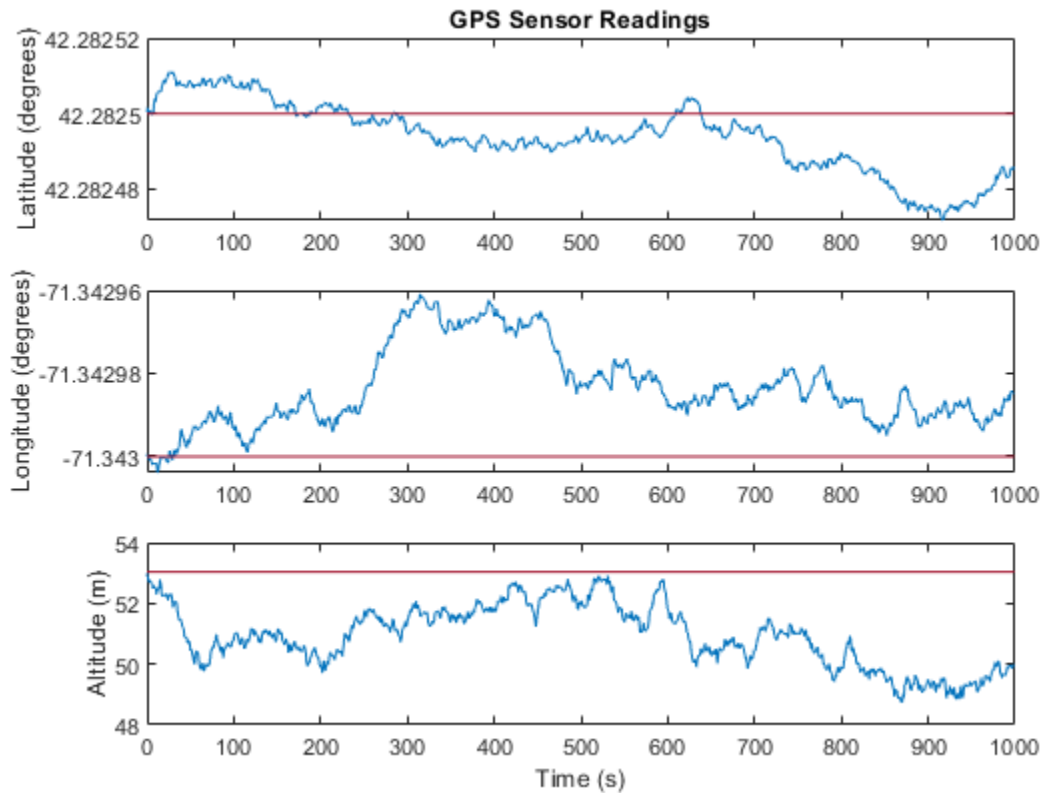
```
refLoc = [42.2825 -71.343 53.0352];  
  
truePosition = zeros(numSamples,3);  
trueVelocity = zeros(numSamples,3);  
  
gps = gpsSensor('SampleRate', fs, 'ReferenceLocation', refLoc);
```

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);

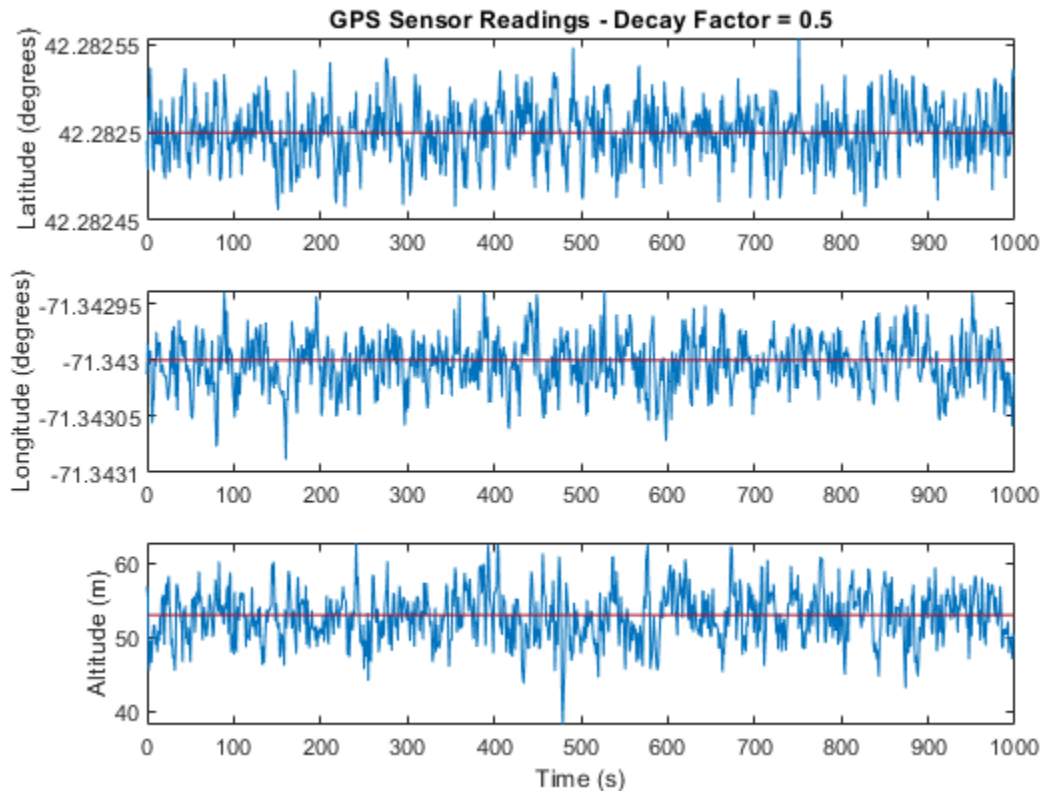
subplot(3, 1, 1)
plot(t, position(:,1), ...
     t, ones(numSamples)*refLoc(1))
title('GPS Sensor Readings - Decay Factor = 0.5')
ylabel('Latitude (degrees)')

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.

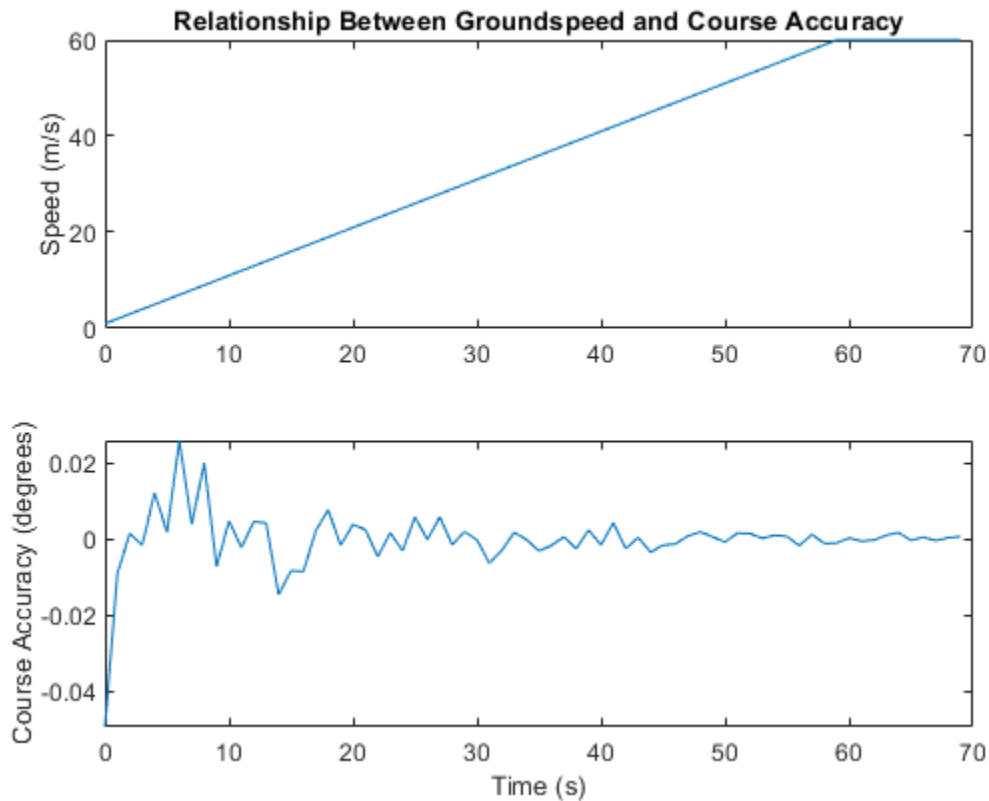
```

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

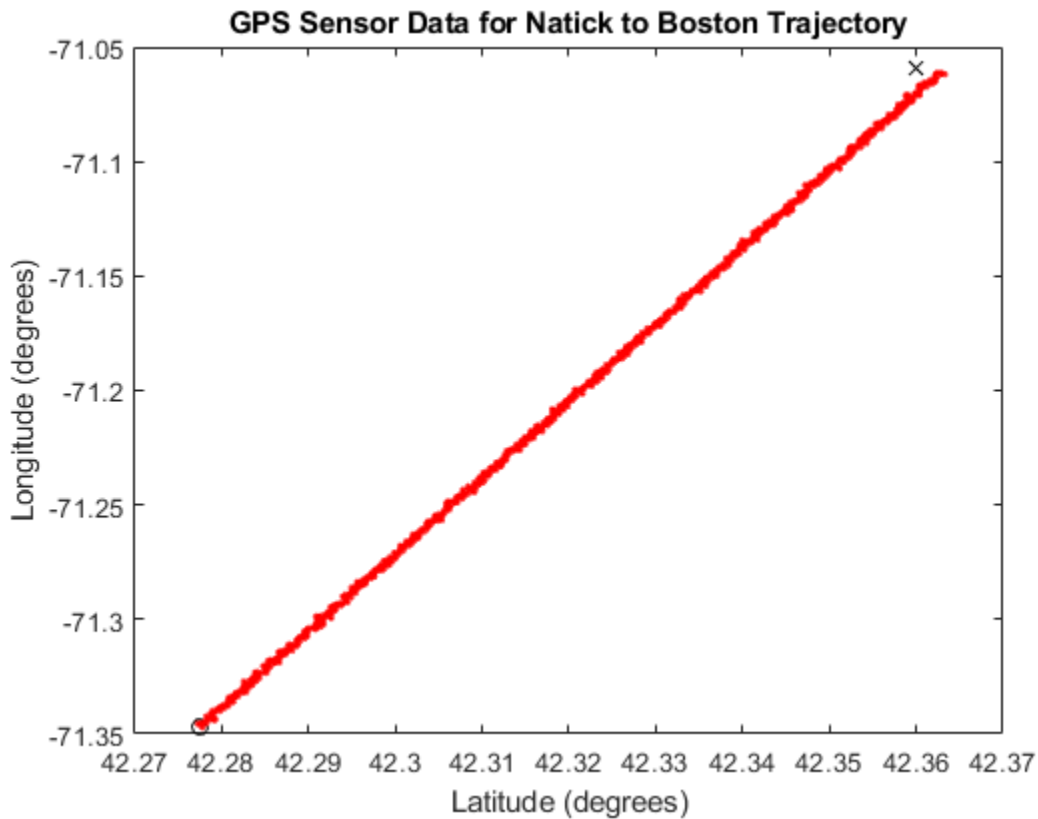
Open a figure and plot the position of Natick and Boston in LLA. Ignore altitude for simplicity.

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(trajjectory)
```

Extended Capabilities

C/C++ Code Generation

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

Usage notes and limitations:

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

See Also

Objects

insSensor

Introduced in R2020b

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 1-37 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/s²)

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: <ul style="list-style-type: none"> 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 meters per second squared, specified as a real, finite N -by-3 matrix of $[\omega_x\ \omega_y\ \omega_z]$ vectors. N is the number of samples in the current frame.

The 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: <ul style="list-style-type: none"> N-element column vector of quaternion objects 3-by-3-by-N array of rotation matrices N-by-3 matrix of $[x_{\text{roll}}\ y_{\text{pitch}}\ z_{\text{yaw}}]$ 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 meters per second squared, specified as a real, finite N -by-3 matrix of $[\omega_x\ \omega_y\ \omega_z]$ vectors. N is the number of samples in the current frame.

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

Object Functions

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

```
release(obj)
```

Specific to `insSensor`

`perturbations` Perturbation defined on object
`perturb` Apply perturbations to object

Common to All System Objects

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

Examples

Generate INS Measurements from Stationary Input

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

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

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

INS = insSensor;

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

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

```
for i = 1:numSamples
    measurements = INS(motion);

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

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

```
end
```

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

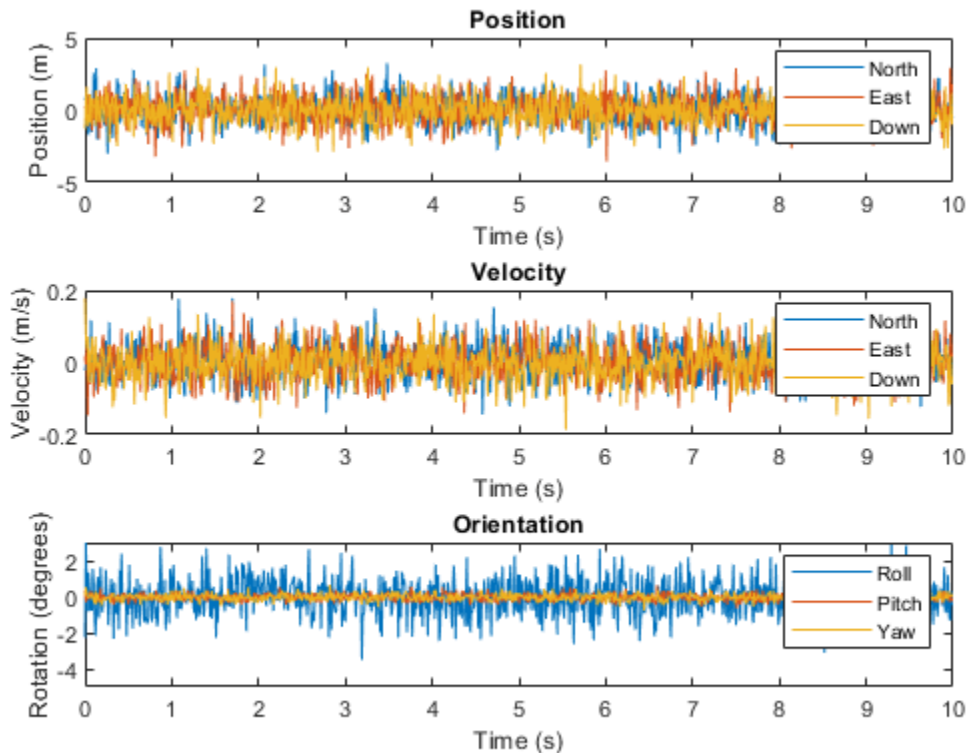
```
orientationMeasurements = eulerd(orientationMeasurements, 'ZYX', 'frame');
```

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

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

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

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

Generate INS Measurements for a Turning Platform

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

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

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

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

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

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

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

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

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

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

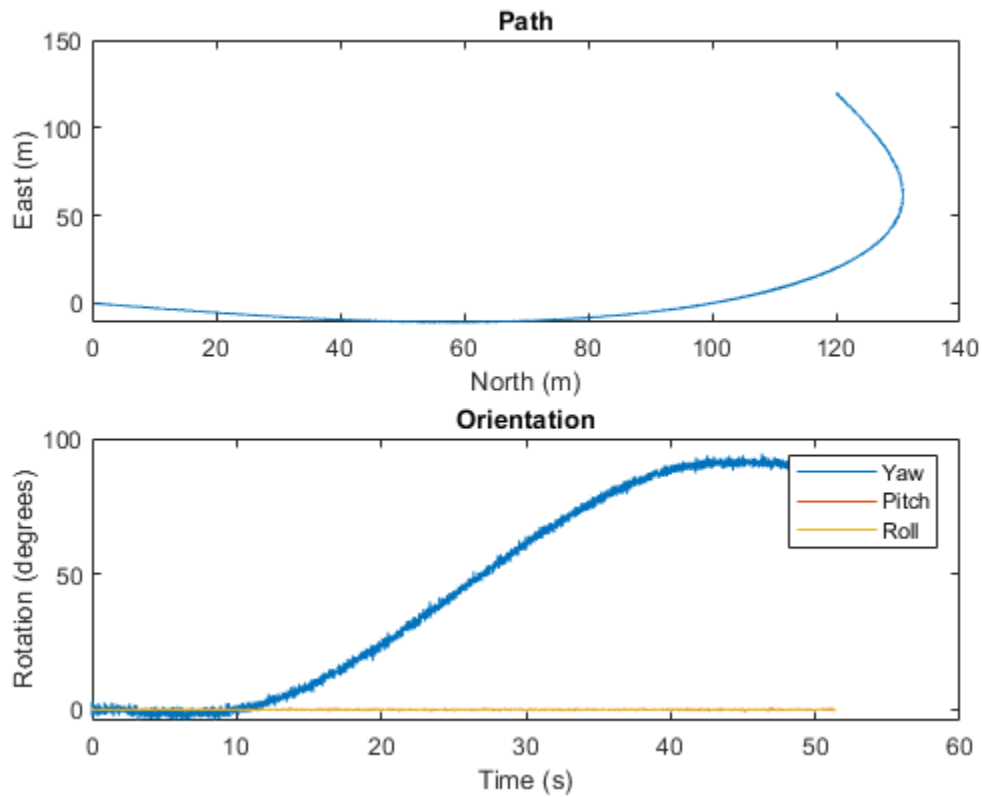
```
[motion.Position,motion.Orientation,motion.Velocity] = path();
insMeas = ins(motion);
```

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

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

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

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



Extended Capabilities

C/C++ Code Generation

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

Usage notes and limitations:

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

See Also

Objects

gpsSensor | uavScenario

Introduced in R2020b

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)
```

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.

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=133x3 table
```

Name	Value
------	-------

"MAV_CMD_NAV_WAYPOINT"	16	"Navigate to waypoint."
"MAV_CMD_NAV_LOITER_UNLIM"	17	"Loiter around this waypoint an unlimited amount of time"
"MAV_CMD_NAV_LOITER_TURNS"	18	"Loiter around this waypoint for X turns"
"MAV_CMD_NAV_LOITER_TIME"	19	"Loiter around this waypoint for X seconds"
"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"
"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 only)"
"MAV_CMD_NAV_FOLLOW"	25	"Vehicle following, i.e. this waypoint repeats the previous one"
"MAV_CMD_NAV_CONTINUE_AND_CHANGE_ALT"	30	"Continue on the current course and climb/descend to the specified altitude"
"MAV_CMD_NAV_LOITER_TO_ALT"	31	"Begin loiter at the specified Latitude and Longitude and then continue to the specified altitude"
"MAV_CMD_DO_FOLLOW"	32	"Begin following a target"
"MAV_CMD_DO_FOLLOW_REPOSITION"	33	"Reposition the MAV after a follow target has been reached"
"MAV_CMD_DO_ORBIT"	34	"Start orbiting on the circumference of a circle with the specified radius around the specified location"
"MAV_CMD_NAV_ROI"	80	"Sets the region of interest (ROI) for a camera or vision sensor"
:		

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

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

```
info=1x4 table
```

MessageID	MessageName	
0	"HEARTBEAT"	"The heartbeat message shows that a system is present and responsive"

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

See Also

[mavlinkclient](#) | [mavlinkio](#) | [mavlinksub](#)

Topics

"Tune UAV Parameters Using MAVLink Parameter Protocol"

External Websites

[MAVLink Developer Guide](#)

Introduced in R2019a

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)
```

See Also

`mavlinkdialect` | `mavlinkio` | `mavlinksub`

Topics

“Tune UAV Parameters Using MAVLink Parameter Protocol”

External Websites

MAVLink Developer Guide

Introduced in R2019a

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

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

<code>connect</code>	Connect to MAVLink clients through UDP port
<code>disconnect</code>	Disconnect from MAVLink clients
<code>sendmsg</code>	Send MAVLink message
<code>sendudpsmsg</code>	Send MAVLink message to UDP port
<code>serializemsg</code>	Serialize MAVLink message to binary buffer
<code>listConnections</code>	List all active MAVLink connections
<code>listClients</code>	List all connected MAVLink clients
<code>listTopics</code>	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.

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.

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=1x4 table
  SystemID  ComponentID  ComponentType  AutopilotType
  _____  _____  _____  _____
         255         1      "MAV_TYPE_GCS"  "MAV_AUTOPILOT_INVALID"
```

```
listConnections(mavlink)
```

```
ans=1x2 table
  ConnectionName  ConnectionInfo
  _____  _____
  "Connection1"  "UDP@0.0.0.0:51064"
```

```
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)
```

See Also

[connect](#) | [mavlinkclient](#) | [mavlinkdialect](#) | [mavlinksub](#)

Topics

[“Tune UAV Parameters Using MAVLink Parameter Protocol”](#)

External Websites

[MAVLink Developer Guide](#)

Introduced in R2019a

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

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)
```

See Also

[latestmsgs](#) | [mavlinkclient](#) | [mavlinkdialect](#) | [mavlinkio](#)

Topics

"Tune UAV Parameters Using MAVLink Parameter Protocol"

External Websites

[MAVLink Developer Guide](#)

Introduced in R2019a

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('flight.tlog');
```

Read the 'REQUEST_DATA_STREAM' messages from the file.

```
msgData = readmsg(result, 'MessageName', 'REQUEST_DATA_STREAM');
```

See Also

[mavlinkclient](#) | [mavlinkdialect](#) | [mavlinkio](#) | [readmsg](#)

Topics

“Visualize and Playback MAVLink Flight Log”

Introduced in R2019a

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.

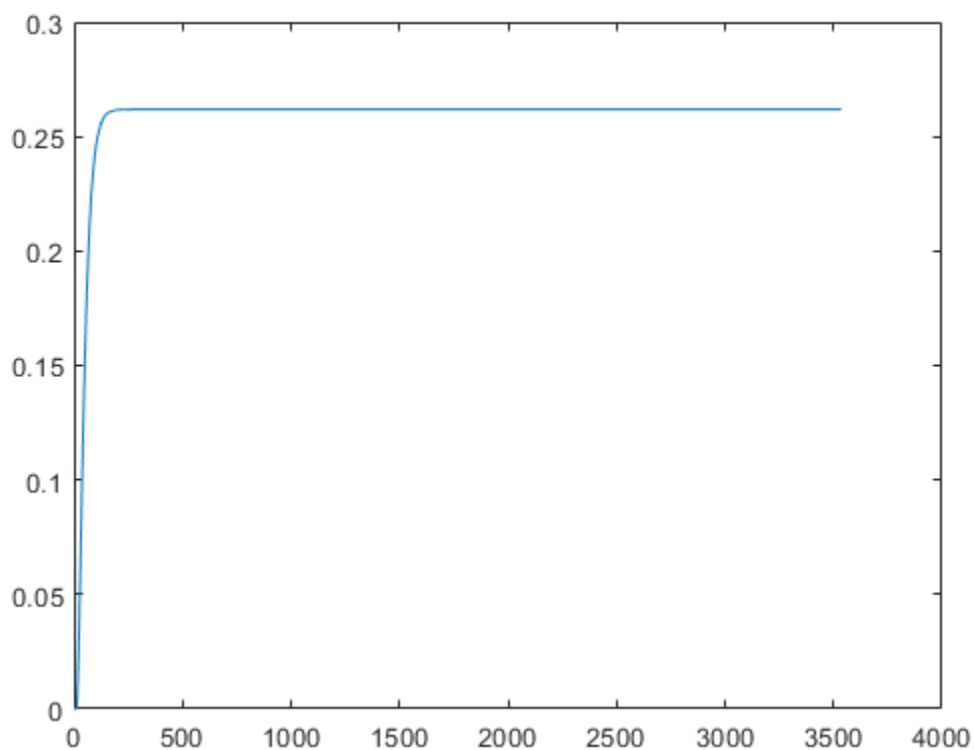
```
simOut = ode45(@(~,x)derivative(model,x,u,e), [0 3], s);  
size(simOut.y)
```

```
ans = 1x2
```

```
13      3536
```

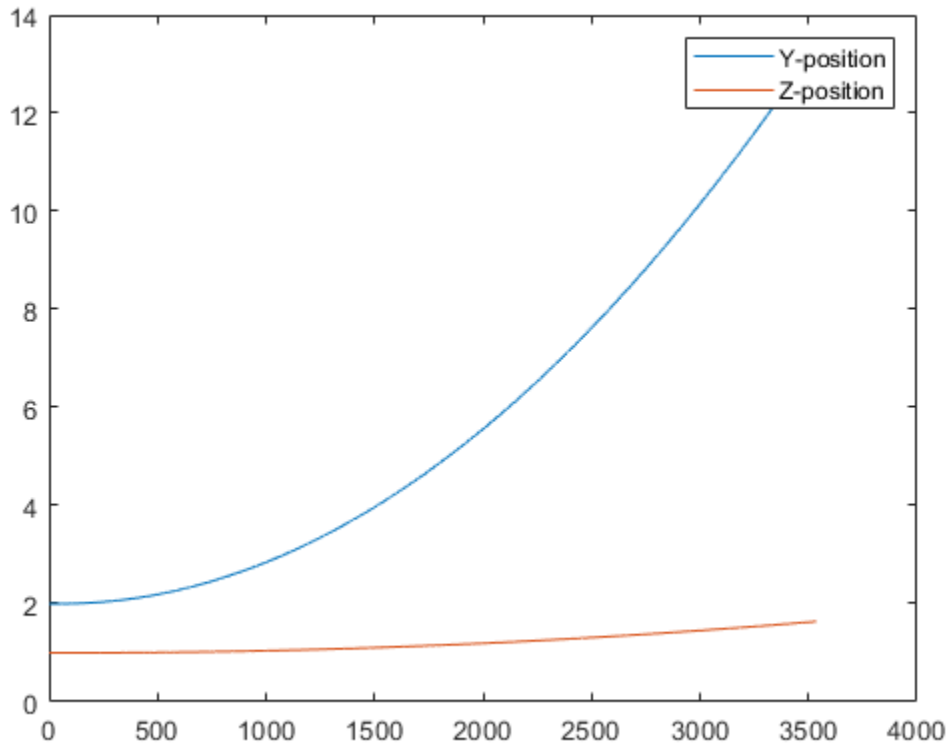
Plot the change in roll angle based on the simulation output. The roll angle (the X Euler angle) is the 9th row of the `simOut.y` output.

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



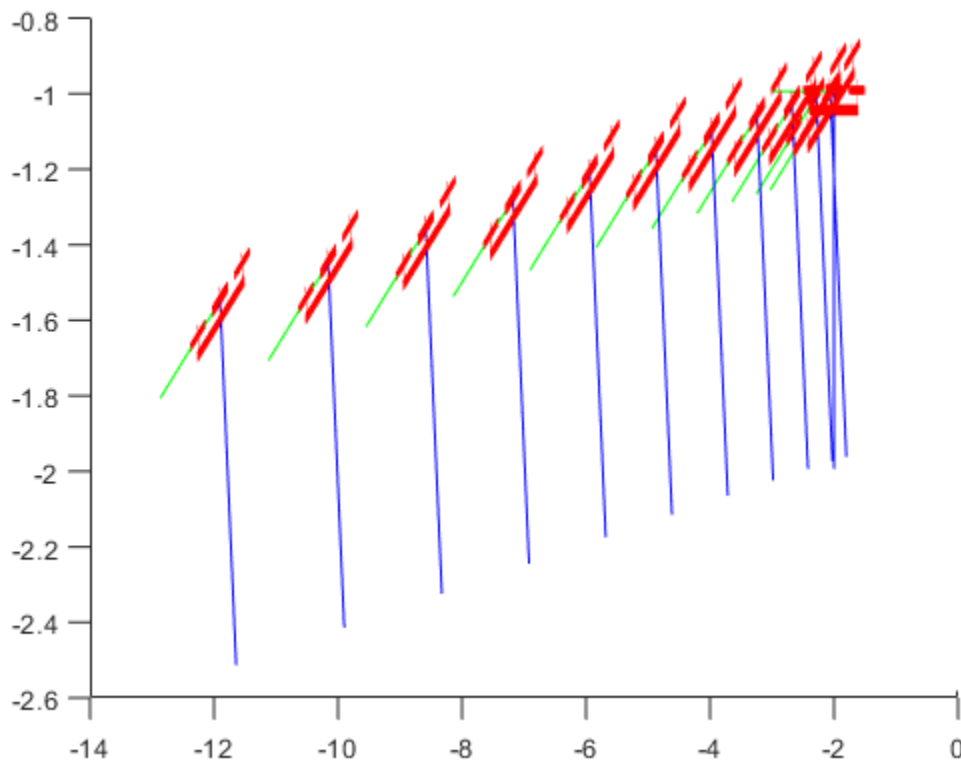
Plot the change in the Y and Z positions. With the specified thrust and roll angle, the multirotor should fly over and lose some altitude. A positive value for Z is expected as positive Z is down.

```
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 from world to body 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:

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

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

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

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

This model assumes the autopilot takes in commanded roll, pitch, yaw rate, $[\psi^c, \theta^c, \phi^c]$ and a commanded total thrust force, F_{thrust}^c . 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:

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

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

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

control | derivative | environment | ode45 | plotTransforms | state

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”

Introduced in R2018b

quaternion

Create a quaternion array

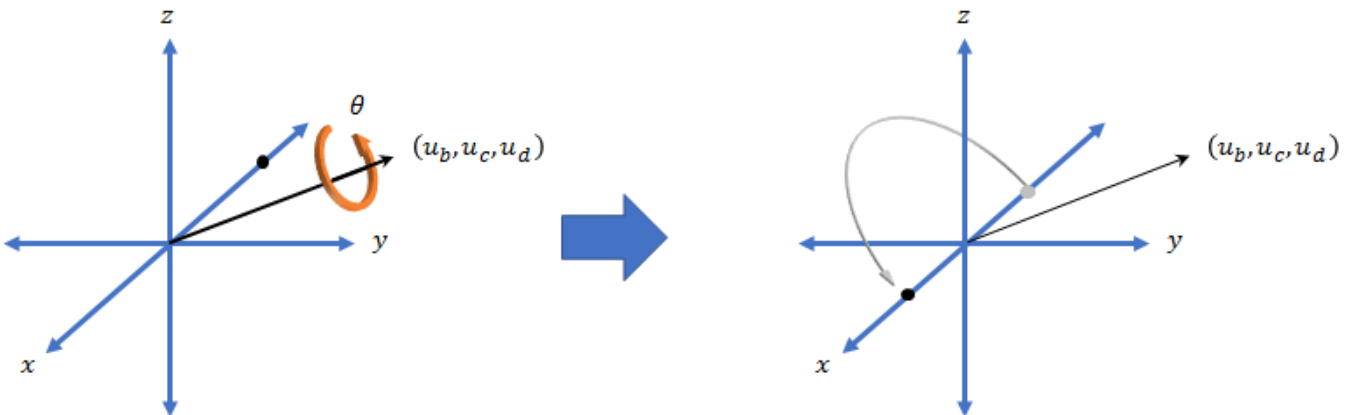
Description

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

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

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

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



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

Creation

Syntax

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

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

Description

`quat = quaternion()` creates an empty quaternion.

`quat = quaternion(A,B,C,D)` creates a quaternion array where the four quaternion parts are taken from the arrays A, B, C, and D. All the inputs must have the same size and be of the same data type.

`quat = quaternion(matrix)` creates an N -by-1 quaternion array from an N -by-4 matrix, where each column becomes one part of the quaternion.

`quat = quaternion(RV, 'rotvec')` creates an N -by-1 quaternion array from an N -by-3 matrix of rotation vectors, RV. Each row of RV represents a rotation vector in radians.

`quat = quaternion(RV, 'rotvecd')` creates an N -by-1 quaternion array from an N -by-3 matrix of rotation vectors, RV. Each row of RV represents a rotation vector in degrees.

`quat = quaternion(RM, 'rotmat', PF)` creates an N -by-1 quaternion array from the 3-by-3-by- N array of rotation matrices, RM. PF can be either 'point' if the Euler angles represent point rotations or 'frame' for frame rotations.

`quat = quaternion(E, 'euler', RS, PF)` creates an N -by-1 quaternion array from the N -by-3 matrix, E. Each row of E represents a set of Euler angles in radians. The angles in E are rotations about the axes in sequence RS.

`quat = quaternion(E, 'eulerd', RS, PF)` creates an N -by-1 quaternion array from the N -by-3 matrix, E. Each row of E represents a set of Euler angles in degrees. The angles in E are rotations about the axes in sequence RS.

Input Arguments

A, B, C, D — Quaternion parts

comma-separated arrays of the same size

Parts of a quaternion, specified as four comma-separated scalars, matrices, or multi-dimensional arrays of the same size.

Example: `quat = quaternion(1,2,3,4)` creates a quaternion of the form $1 + 2i + 3j + 4k$.

Example: `quat = quaternion([1,5],[2,6],[3,7],[4,8])` creates a 1-by-2 quaternion array where `quat(1,1) = 1 + 2i + 3j + 4k` and `quat(1,2) = 5 + 6i + 7j + 8k`

Data Types: `single` | `double`

matrix — Matrix of quaternion parts

N -by-4 matrix

Matrix of quaternion parts, specified as an N -by-4 matrix. Each row represents a separate quaternion. Each column represents a separate quaternion part.

Example: `quat = quaternion(rand(10,4))` creates a 10-by-1 quaternion array.

Data Types: `single` | `double`

RV — Matrix of rotation vectors

N-by-3 matrix

Matrix of rotation vectors, specified as an *N*-by-3 matrix. Each row of **RV** represents the [X Y Z] elements of a rotation vector. A rotation vector is a unit vector representing the axis of rotation scaled by the angle of rotation in radians or degrees.

To use this syntax, specify the first argument as a matrix of rotation vectors and the second argument as the `'rotvec'` or `'rotvecd'`.

Example: `quat = quaternion(rand(10,3), 'rotvec')` creates a 10-by-1 quaternion array.

Data Types: `single` | `double`

RM — Rotation matrices

3-by-3 matrix | 3-by-3-by-*N* array

Array of rotation matrices, specified by a 3-by-3 matrix or 3-by-3-by-*N* array. Each page of the array represents a separate rotation matrix.

Example: `quat = quaternion(rand(3), 'rotmat', 'point')`

Example: `quat = quaternion(rand(3), 'rotmat', 'frame')`

Data Types: `single` | `double`

PF — Type of rotation matrix

`'point'` | `'frame'`

Type of rotation matrix, specified by `'point'` or `'frame'`.

Example: `quat = quaternion(rand(3), 'rotmat', 'point')`

Example: `quat = quaternion(rand(3), 'rotmat', 'frame')`

Data Types: `char` | `string`

E — Matrix of Euler angles

N-by-3 matrix

Matrix of Euler angles, specified by an *N*-by-3 matrix. If using the `'euler'` syntax, specify *E* in radians. If using the `'eulerd'` syntax, specify *E* in degrees.

Example: `quat = quaternion(E, 'euler', 'YZY', 'point')`

Example: `quat = quaternion(E, 'euler', 'XYZ', 'frame')`

Data Types: `single` | `double`

RS — Rotation sequence

character vector | scalar string

Rotation sequence, specified as a three-element character vector:

- `'YZY'`
- `'YXY'`
- `'ZYZ'`

- 'ZXZ'
- 'XYX'
- 'XZX'
- 'XYZ'
- 'YZX'
- 'ZXY'
- 'XZY'
- 'ZYX'
- 'YXZ'

Assume you want to determine the new coordinates of a point when its coordinate system is rotated using frame rotation. The point is defined in the original coordinate system as:

```
point = [sqrt(2)/2,sqrt(2)/2,0];
```

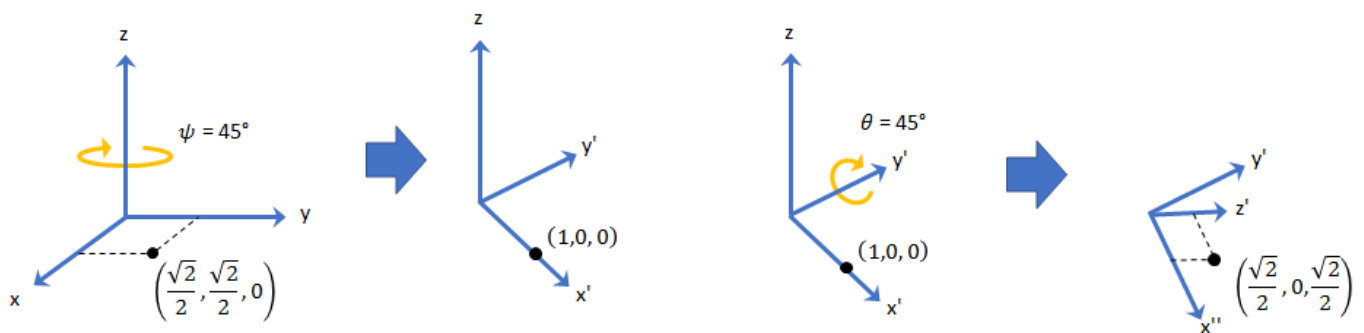
In this representation, the first column represents the x-axis, the second column represents the y-axis, and the third column represents the z-axis.

You want to rotate the point using the Euler angle representation [45,45,0]. Rotate the point using two different rotation sequences:

- If you create a quaternion rotator and specify the 'ZYX' sequence, the frame is first rotated 45° around the z-axis, then 45° around the new y-axis.

```
quatRotator = quaternion([45,45,0], 'eulerd', 'ZYX', 'frame');
newPointCoordinate = rotateframe(quatRotator,point)
```

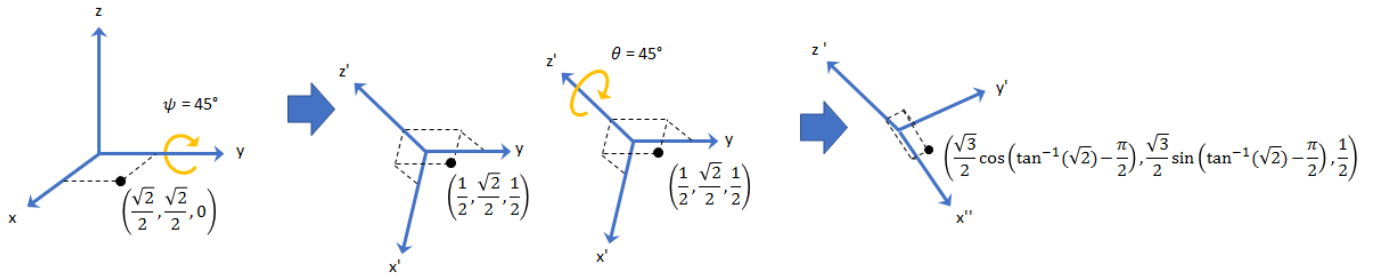
```
newPointCoordinate =
    0.7071    -0.0000    0.7071
```



- If you create a quaternion rotator and specify the 'YZX' sequence, the frame is first rotated 45° around the y-axis, then 45° around the new z-axis.

```
quatRotator = quaternion([45,45,0], 'eulerd', 'YZX', 'frame');
newPointCoordinate = rotateframe(quatRotator,point)
```

```
newPointCoordinate =
    0.8536    0.1464    0.5000
```



Data Types: char | string

Object Functions

angvel	Angular velocity from quaternion array
classUnderlying	Class of parts within quaternion
compact	Convert quaternion array to N-by-4 matrix
conj	Complex conjugate of quaternion
ctranspose, 'r'	Complex conjugate transpose of quaternion array
dist	Angular distance in radians
euler	Convert quaternion to Euler angles (radians)
eulerd	Convert quaternion to Euler angles (degrees)
exp	Exponential of quaternion array
ldivide, ./	Element-wise quaternion left division
log	Natural logarithm of quaternion array
meanrot	Quaternion mean rotation
minus, -	Quaternion subtraction
mtimes, *	Quaternion multiplication
norm	Quaternion norm
normalize	Quaternion normalization
ones	Create quaternion array with real parts set to one and imaginary parts set to zero
parts	Extract quaternion parts
power, .^	Element-wise quaternion power
prod	Product of a quaternion array
randrot	Uniformly distributed random rotations
rdivide, ./	Element-wise quaternion right division
rotateframe	Quaternion frame rotation
rotatepoint	Quaternion point rotation
rotmat	Convert quaternion to rotation matrix
rotvec	Convert quaternion to rotation vector (radians)
rotvecd	Convert quaternion to rotation vector (degrees)
slerp	Spherical linear interpolation
times, .*	Element-wise quaternion multiplication
transpose, 'r'	Transpose a quaternion array
uminus, -	Quaternion unary minus
zeros	Create quaternion array with all parts set to zero

Examples

Create Empty Quaternion

```
quat = quaternion()
```

```
quat =  
    0x0 empty quaternion array
```

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

```
classUnderlying(quat)
```

```
ans =  
'double'
```

Create Quaternion by Specifying Individual Quaternion Parts

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

Define quaternion parts as scalars.

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

Define quaternion parts as column vectors.

```
A = [1.1;1.2];  
B = [2.1;2.2];  
C = [3.1;3.2];  
D = [4.1;4.2];  
quatVector = quaternion(A,B,C,D)  
  
quatVector=2x1 quaternion array  
    1.1 + 2.1i + 3.1j + 4.1k  
    1.2 + 2.2i + 3.2j + 4.2k
```

Define quaternion parts as matrices.

```
A = [1.1,1.3; ...  
    1.2,1.4];  
B = [2.1,2.3; ...  
    2.2,2.4];  
C = [3.1,3.3; ...  
    3.2,3.4];  
D = [4.1,4.3; ...  
    4.2,4.4];  
quatMatrix = quaternion(A,B,C,D)  
  
quatMatrix=2x2 quaternion array  
    1.1 + 2.1i + 3.1j + 4.1k    1.3 + 2.3i + 3.3j + 4.3k  
    1.2 + 2.2i + 3.2j + 4.2k    1.4 + 2.4i + 3.4j + 4.4k
```


Define quaternion parts as three dimensional arrays.

```

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

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

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

quatMultiDimArray(:,:,2) =

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

```

Create Quaternion by Specifying Quaternion Parts Matrix

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

Create a column vector of random quaternions.

```

quatParts = rand(3,4)

quatParts = 3x4

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

quat = quaternion(quatParts)

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

```

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

```

retrievedquatParts = compact(quat)

retrievedquatParts = 3x4

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

```

Create Quaternion by Specifying Rotation Vectors

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

Rotation Vectors in Radians

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

```
rotationVector = [0.3491,0.6283,0.3491];  
quat = quaternion(rotationVector,'rotvec')  
  
quat = quaternion  
      0.92124 + 0.16994i + 0.30586j + 0.16994k  
  
norm(quat)  
  
ans = 1.0000
```

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

```
rotvec(quat)  
  
ans = 1×3  
      0.3491    0.6283    0.3491
```

Rotation Vectors in Degrees

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

```
rotationVector = [20,36,20];  
quat = quaternion(rotationVector,'rotvecd')  
  
quat = quaternion  
      0.92125 + 0.16993i + 0.30587j + 0.16993k  
  
norm(quat)  
  
ans = 1
```

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

```
rotvecd(quat)  
  
ans = 1×3  
      20.0000    36.0000    20.0000
```

Create Quaternion by Specifying Rotation Matrices

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

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

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

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

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

```
rotmat(quat, 'frame')

ans = 3×3

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

Create Quaternion by Specifying Euler Angles

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

Euler Angles in Radians

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

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

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

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

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

ans = 1×3

    1.5708      0    0.7854
```

Euler Angles in Degrees

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

```
E = [90,0,45];  
quat = quaternion(E, 'eulerd', 'ZYX', 'frame')  
  
quat = quaternion  
    0.65328 + 0.2706i + 0.2706j + 0.65328k
```

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

```
eulerd(quat, 'ZYX', 'frame')  
  
ans = 1×3  
    90.0000         0    45.0000
```

Quaternion Algebra

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

Addition and Subtraction

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

```
Q1 = quaternion(1,2,3,4)  
  
Q1 = quaternion  
    1 + 2i + 3j + 4k  
  
Q2 = quaternion(9,8,7,6)  
  
Q2 = quaternion  
    9 + 8i + 7j + 6k  
  
Q1plusQ2 = Q1 + Q2  
  
Q1plusQ2 = quaternion  
    10 + 10i + 10j + 10k  
  
Q2plusQ1 = Q2 + Q1  
  
Q2plusQ1 = quaternion  
    10 + 10i + 10j + 10k  
  
Q1minusQ2 = Q1 - Q2
```

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

```
Q2minusQ1 = Q2 - Q1
```

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

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

```
Q1plusRealNumber = Q1 + 5
```

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

```
Q1minusRealNumber = Q1 - 5
```

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

Multiplication

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

```
Q1timesQ2 = Q1 * Q2
```

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

```
Q2timesQ1 = Q2 * Q1
```

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

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

```
ans = logical
0
```

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

```
Q1times5 = Q1*5
```

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

Multiplying a quaternion by a real number is commutative.

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

```
ans = logical  
     1
```

Conjugation

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

Q1

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

```
conj(Q1)
```

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

Multiplication between a quaternion and its conjugate is commutative:

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

```
ans = logical  
     1
```

Quaternion Array Manipulation

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

Concatenate

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

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

```
qVector = [Q1,Q2]
```

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

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

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

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

$$-1 - 2i - 3j - 4k \quad -9 - 8i - 7j - 6k$$

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

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

$$\begin{array}{cc} 1 + 2i + 3j + 4k & 9 + 8i + 7j + 6k \\ -1 - 2i - 3j - 4k & -9 - 8i - 7j - 6k \end{array}$$

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

$$\begin{array}{cc} 1 + 2i + 3j + 4k & 9 + 8i + 7j + 6k \\ -1 - 2i - 3j - 4k & -9 - 8i - 7j - 6k \end{array}$$

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) =
```

$$\begin{array}{cc} 1 + 2i + 3j + 4k & 9 + 8i + 7j + 6k \\ 1 + 0i + 0j + 0k & -9 - 8i - 7j - 6k \end{array}$$

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

$$\begin{array}{cc} 1 + 2i + 3j + 4k & 9 + 8i + 7j + 6k \\ -1 - 2i - 3j - 4k & -9 - 8i - 7j - 6k \end{array}$$

Reshape

To reshape quaternion arrays, use the reshape function.

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

```
qMatReshaped=4x1 quaternion array
```

$$\begin{array}{l} 1 + 2i + 3j + 4k \\ -1 - 2i - 3j - 4k \\ 9 + 8i + 7j + 6k \\ -9 - 8i - 7j - 6k \end{array}$$

Transpose

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

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

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

Permute

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

```
qMultiDimensionalArray
```

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

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

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

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

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

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

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

```
qMatPermute(:,:,2) =
```

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

Extended Capabilities

C/C++ Code Generation

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

See Also

Introduced in R2020b

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();
```

See Also

Topics

“Customize Unreal Engine Scenes for UAVs”

Introduced in R2020b

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 `updateFrame`.

Data Types: double

Object Functions

<code>getGraph</code>	Graph object representing tree structure
<code>getTransform</code>	Get relative transform between frames
<code>info</code>	List all frame names and stored timestamps
<code>removeTransform</code>	Remove frame transform relative to its parent
<code>show</code>	Show transform tree
<code>updateTransform</code>	Update frame transform relative to its parent

See Also

Objects

`fixedwing` | `multirotor` | `uavDubinsPathSegment`

Functions

`getGraph` | `getTransform` | `info` | `removeTransform` | `show` | `updateTransform`

Introduced in R2020b

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.

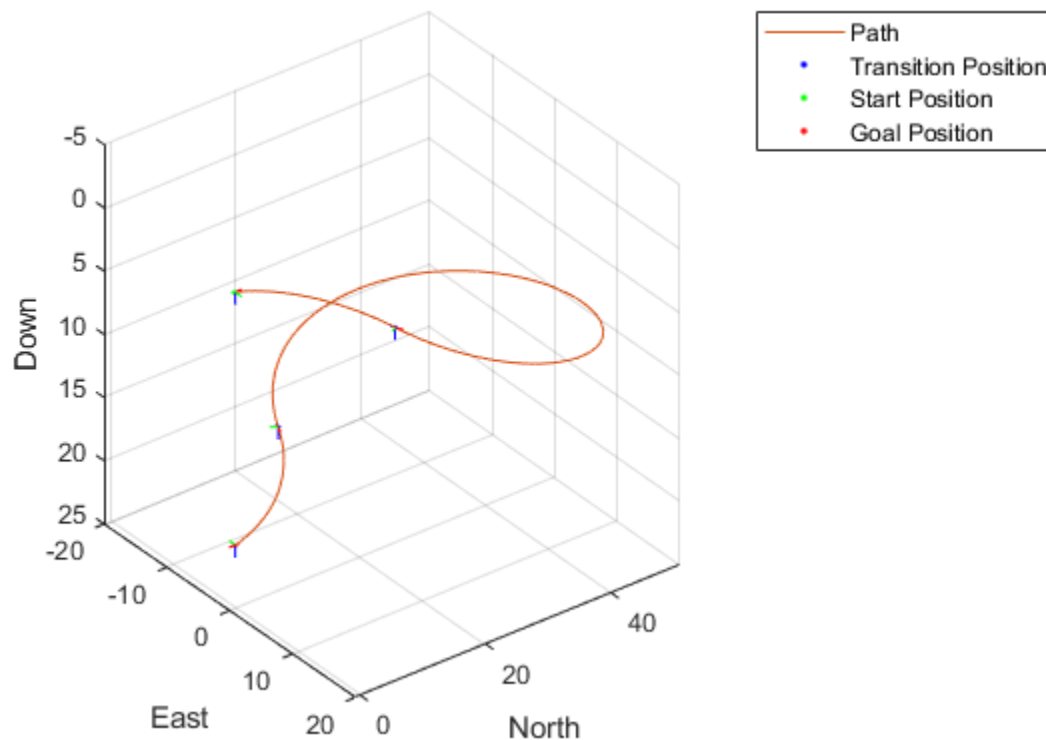
```
startPose = [0 0 0 0]; % [meters, meters, meters, radians]
goalPose = [0 0 20 pi];
```

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.

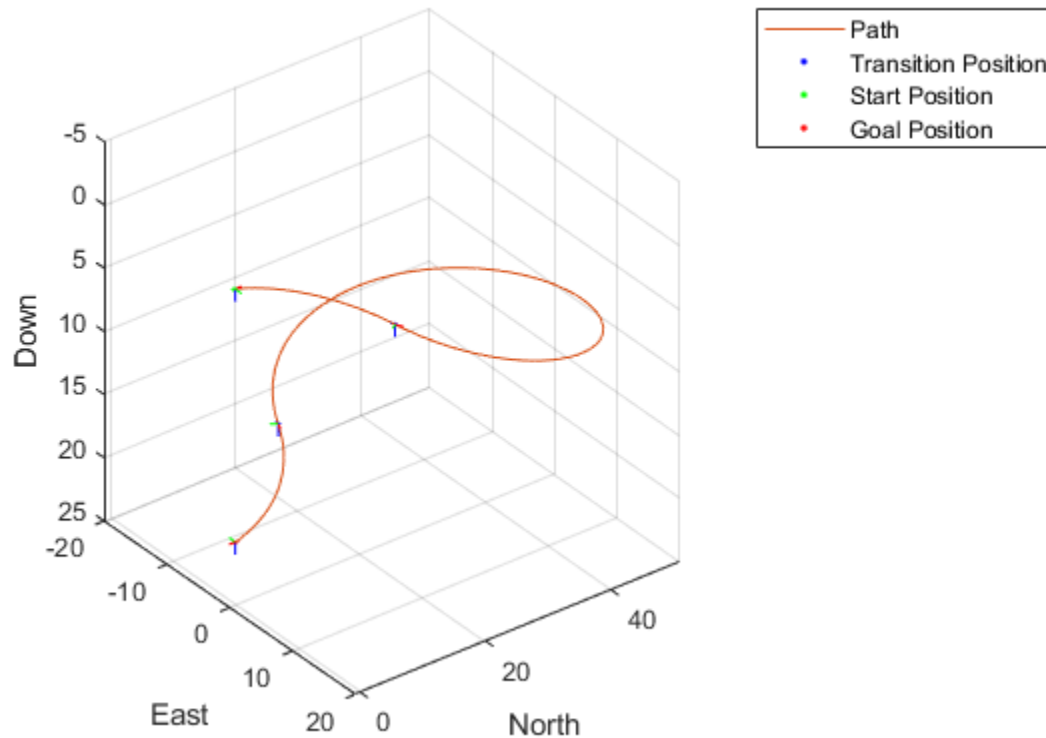
```
startPose = [0 0 0 0]; % [meters, meters, meters, radians]
goalPose = [0 0 20 pi];
```

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.

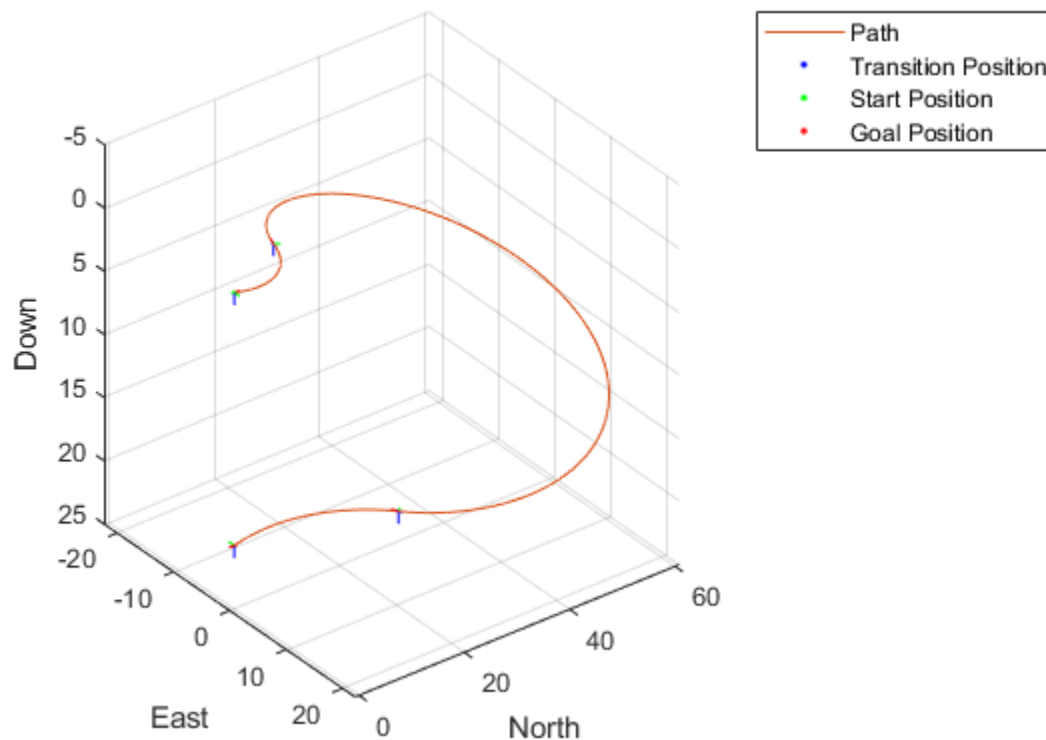
```
connectionObj = uavDubinsConnection('DisabledPathTypes',{'RLRN'});
connectionObj.AirSpeed = 15;
connectionObj.MaxRollAngle = 0.8;
connectionObj.FlightPathAngleLimit = [-1.47 1.47];
```

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
```

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

Introduced in R2019b

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.

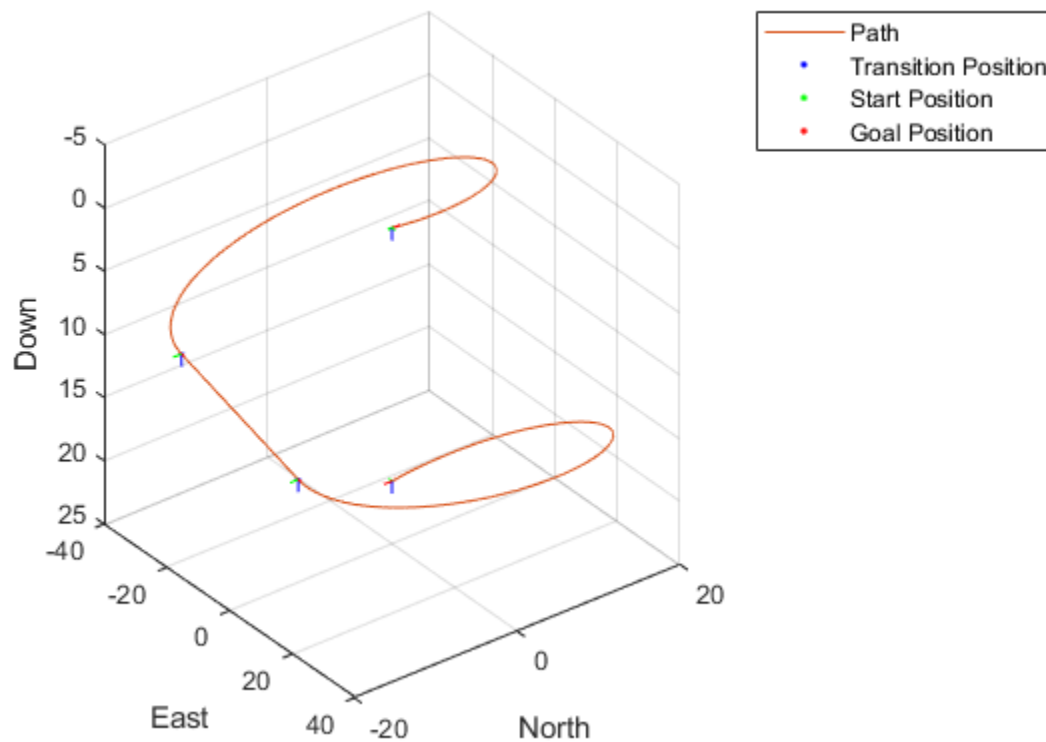
```
startPose = [0 0 0 0]; % [meters, meters, meters, radians]  
goalPose = [0 0 20 pi];
```

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
```

Extended Capabilities

C/C++ Code Generation

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

See Also

[interpolate](#) | [show](#)

Introduced in R2019b

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 <code>extendedObjectMesh</code> 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 (true). `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]);
```

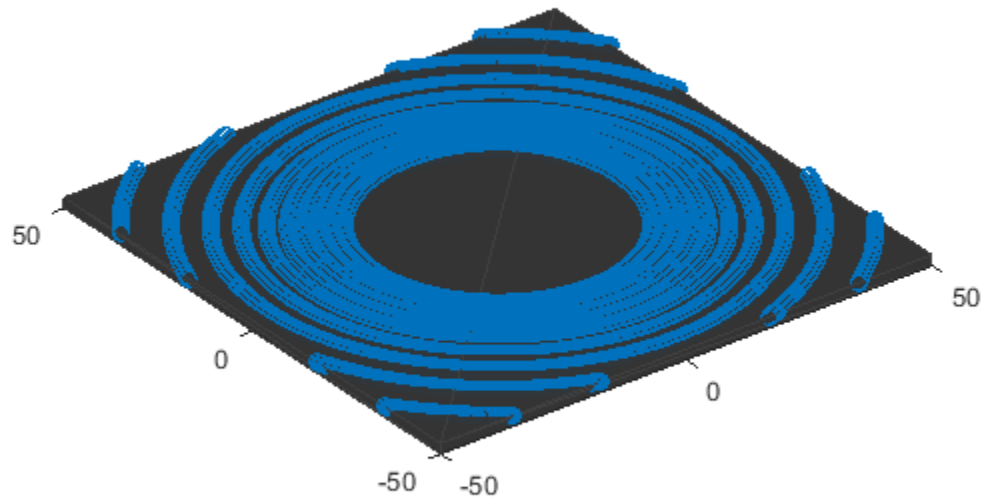
Generate Point Clouds from Floor

```
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));
```

**See Also**

uavScenario

Topics

"UAV Scenario Tutorial"

Introduced in R2020b

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);
```

Extended Capabilities

C/C++ Code Generation

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

See Also

Functions

[control](#) | [derivative](#) | [environment](#) | [plotTransforms](#) | [state](#)

Objects

[fixedwing](#) | [multirotor](#) | [uavWaypointFollower](#)

Blocks

[Orbit Follower](#) | [UAV Guidance Model](#) | [Waypoint Follower](#)

Introduced in R2019a

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 structuresUAV 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)	[<i>x</i> ; <i>y</i> ; <i>z</i>] <i>x</i> , <i>y</i> , and <i>z</i> is the center of the circular orbit in NED coordinates specified in meters	[<i>radius</i> ; <i>turnDir</i> ; <i>numTurns</i> ; <i>p4</i>] <i>radius</i> — Radius of the orbit in meters <i>turnDir</i> — Turn direction, specified as one of these: <ul style="list-style-type: none"> • 1 — Clockwise turn • -1 — Counter-clockwise turn • 0 — Automatic selection of turn direction <i>numTurns</i> — Number of turns	Orbit — Orbit along the circumference of a circle defined by the parameters
uint8(4)	[<i>x</i> ; <i>y</i> ; <i>z</i>]	[<i>p1</i> ; <i>p2</i> ; <i>p3</i> ; <i>p4</i>]	Land — Land at the specified position
uint8(5)	[<i>x</i> ; <i>y</i> ; <i>z</i>] The launch position is specified in the Home property	[<i>p1</i> ; <i>p2</i> ; <i>p3</i> ; <i>p4</i>]	RTL — Return to launch position
uint8(6)	[<i>x</i> ; <i>y</i> ; <i>z</i>]	[<i>p1</i> ; <i>p2</i> ; <i>p3</i> ; <i>p4</i>] <i>p1</i> , <i>p2</i> , <i>p3</i> , and <i>p4</i> 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 $[x; y; z; \text{courseAngle}]$. 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.

- Multicopter 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

Current Mission Mode	Output Mission Parameters			
	Mission Points	mode	position	params
Waypoint	Row 1: 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
	Row 2: Previous	mode of the previous mission point	position of the previous mission point	<ul style="list-style-type: none"> [yaw;radius;p3;p4] if the previous mission point was Takeoff [courseAngle;25;p3;p4] otherwise <i>courseAngle</i> — Angle of the line segment between the previous and the current position, specified in radians in the range [-pi, pi]
Orbit	Row 1: 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: <ul style="list-style-type: none"> 1 — Clockwise turn -1 — Counterclockwise turn 0 — Automatic selection of turn direction <i>numTurns</i> — Number of turns

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
Multicopter	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

See Also

fixedwing | multicopter | uavOrbitFollower | uavWaypointFollower

Introduced in R2020b

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

Example: `'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.

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

Trajectory for UAV platform motion, specified as a waypointTrajectory 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 waypointTrajectory 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

Trajectory for UAV platform motion, specified as a `waypointTrajectory` object. By default, the object assumes the platform is stationary and at the scenario 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 `waypointTrajectory` 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 platformarray 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**

<code>move</code>	Move UAV platform in scenario
<code>read</code>	Read UAV motion vector
<code>updateMesh</code>	Update body mesh for UAV platform
<code>addGeoFence</code>	Add geographical fencing to UAV platform
<code>checkPermission</code>	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 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]);

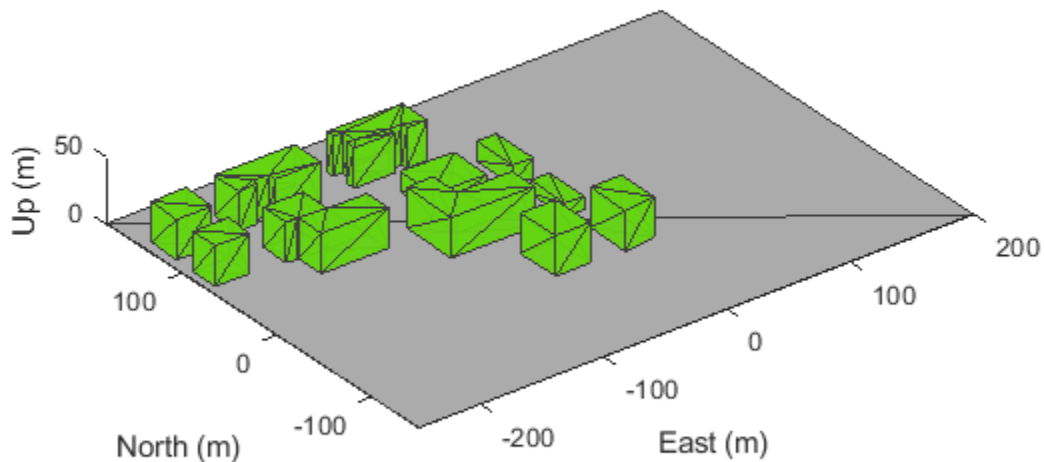
% Add a ground plane.
color.Gray = 0.651*ones(1,3);
color.Green = [0.3922 0.8314 0.0745];
color.Red = [1 0 0];
```

```
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.
traj = plot3(nan,nan,nan,"Color",[1 1 1],"LineWidth",2);
traj.XDataSource = "position(:,1,1:idx+1)";
traj.YDataSource = "position(:,2,1:idx+1)";
```



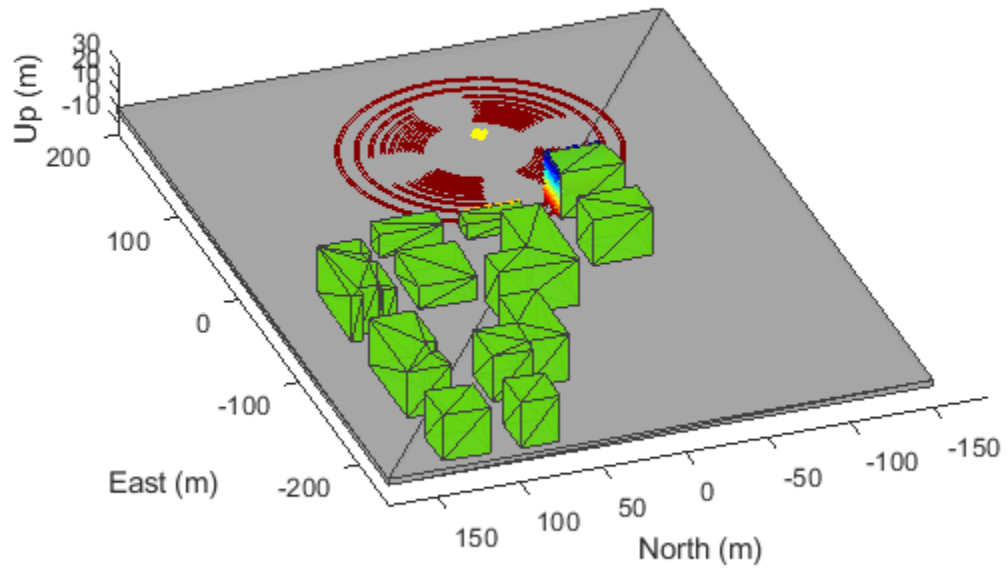
```

traj.ZDataSource = "position(:,3,1:idx+1)";

% Create a scatter plot for the point cloud.
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),[],1),1)";

% Start Simulation
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(:,1:3,idx+1),zeros(1,3)],eul2quat(orientation(:,1:3,idx+1)),zeros(1,3))
    % Update all sensors in the scene.
    updateSensors(scene)
end
hold off

```



See Also

Functions

`addGeoFence` | `checkPermission` | `move` | `read` | `updateMesh`

Objects

`uavScenario` | `uavSensor`

Topics

“UAV Scenario Tutorial”

Introduced in R2020b

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

`transformTree` 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

<code>setup</code>	Prepare UAV scenario for simulation
<code>addCustomTerrain</code>	Add custom terrain data
<code>addMesh</code>	Add new static mesh to UAV scenario
<code>addInertialFrame</code>	Define new inertial frame in UAV scenario
<code>advance</code>	Advance UAV scenario simulation by one time step
<code>updateSensors</code>	Update sensor readings in UAV scenario
<code>removeCustomTerrain</code>	Remove custom terrain data
<code>restart</code>	Reset simulation of UAV scenario
<code>show</code>	Visualize UAV scenario in 2-D
<code>show3D</code>	Visualize UAV scenario in 3-D
<code>terrainHeight</code>	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.

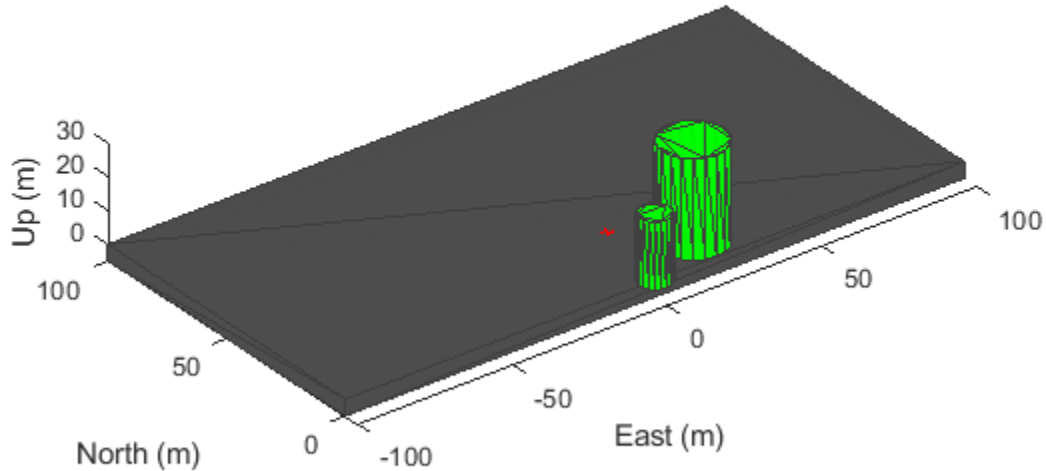
```
insModel = insSensor();  
ins = uavSensor("INS",uavPlat,insModel,"MountingLocation",[4 0 0]);
```

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.

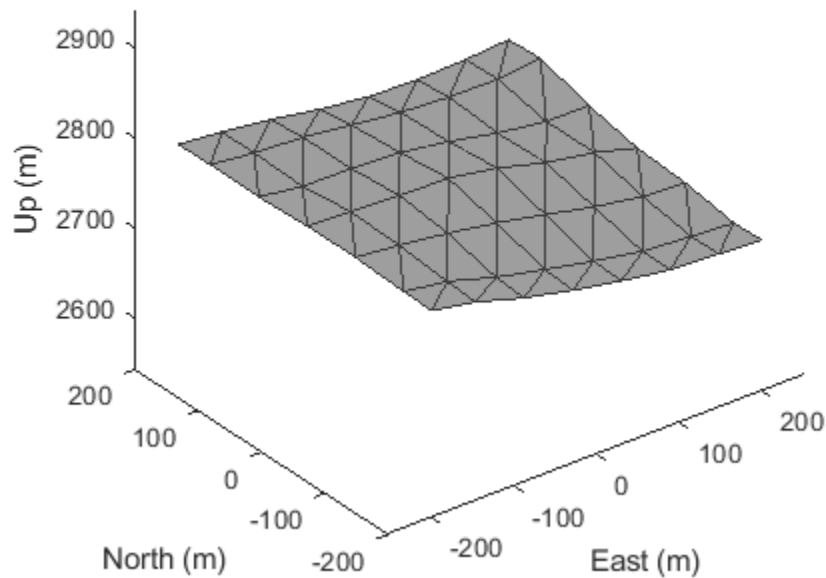
Add Terrain Surface

Add terrain surface based on terrain elevation data from the n39_w106_3arc_v2.dt1 DTED file.

```

addCustomTerrain("CustomTerrain","n39_w106_3arc_v2.dtl");
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.

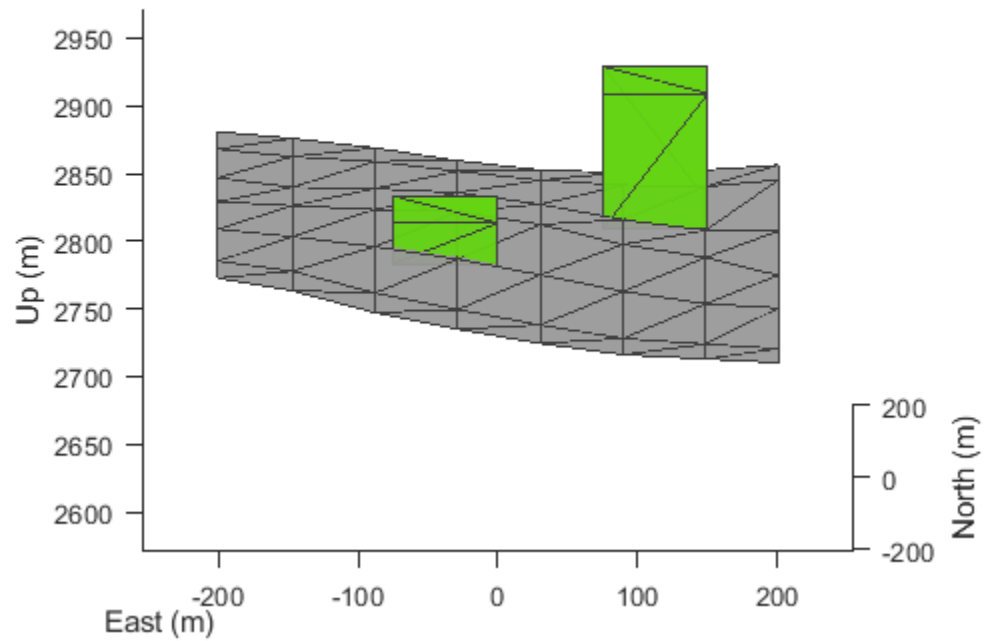
```

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)]}, [0.39
end

show3D(scenario);
view([0 15])

```



Remove Custom Terrain

Remove the custom terrain that was imported.

```
removeCustomTerrain("CustomTerrain")
```

See Also

[uavPlatform](#) | [uavSensor](#)

Topics

"UAV Scenario Tutorial"

Introduced in R2020b

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`, or `uavLidarPointCloudGenerator` System object.

Creation

Syntax

```
sensor = uavSensor(name,platform,sensormodel)
sensor = uavSensor( ___,Name,Value)
```

Description

`sensor = uavSensor(name,platform,sensormodel)` creates a sensor with the specified 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 1-133 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 ($1/\text{UpdateRate}$) 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 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)
```

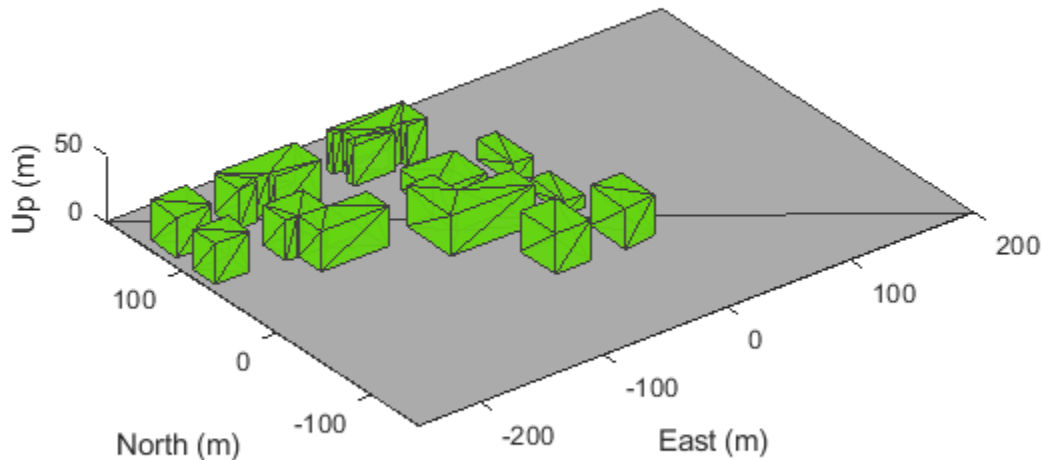
```
% Show the scenario.
```

```
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.
```

```
traj = plot3(nan,nan,nan,"Color",[1 1 1],"LineWidth",2);
```

```
traj.XDataSource = "position(:,1,1:idx+1)";
```

```
traj.YDataSource = "position(:,2,1:idx+1)";
```

```
traj.ZDataSource = "position(:,3,1:idx+1)";
```

```
% Create a scatter plot for the point cloud.
```

```
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),[],1),...";
```

```
% Start Simulation
```

```
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.
```

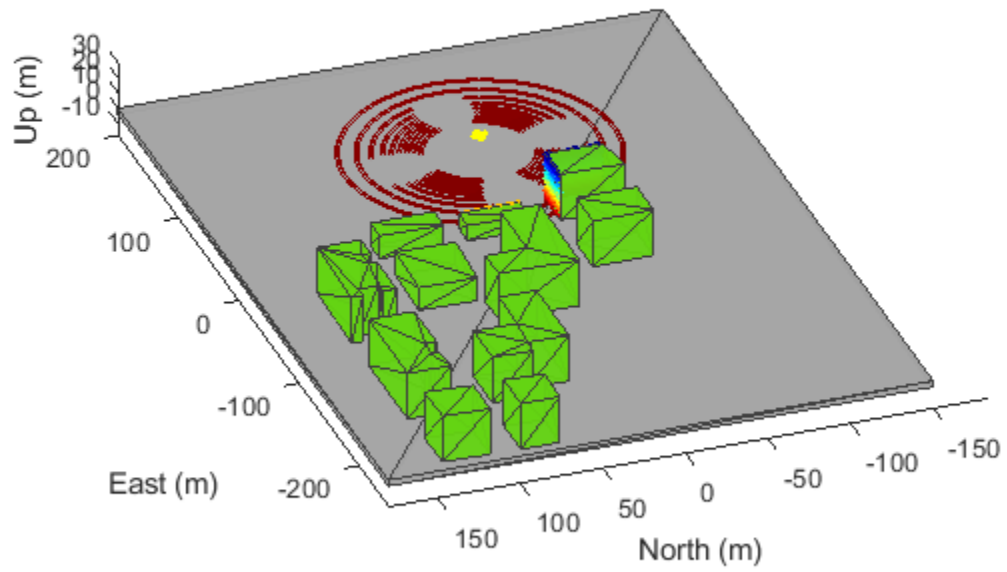
```
    advance(scene);
```

```
    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
```



See Also

Functions

read

Objects

uavPlatform | uavScenario | uavSensor

Topics

“UAV Scenario Tutorial”

Introduced in R2020b

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

Static Methods

uav.SensorAdaptor.getMotion Get sensor motion in platform reference frame

See Also

Functions

getEmptyOutputs | read | reset | setup | uav.SensorAdaptor.getMotion

Objects

uavPlatform | uavScenario | uavSensor

Topics

“Simulate Radar Sensor Mounted On UAV”

Introduced in R2021a

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)
```

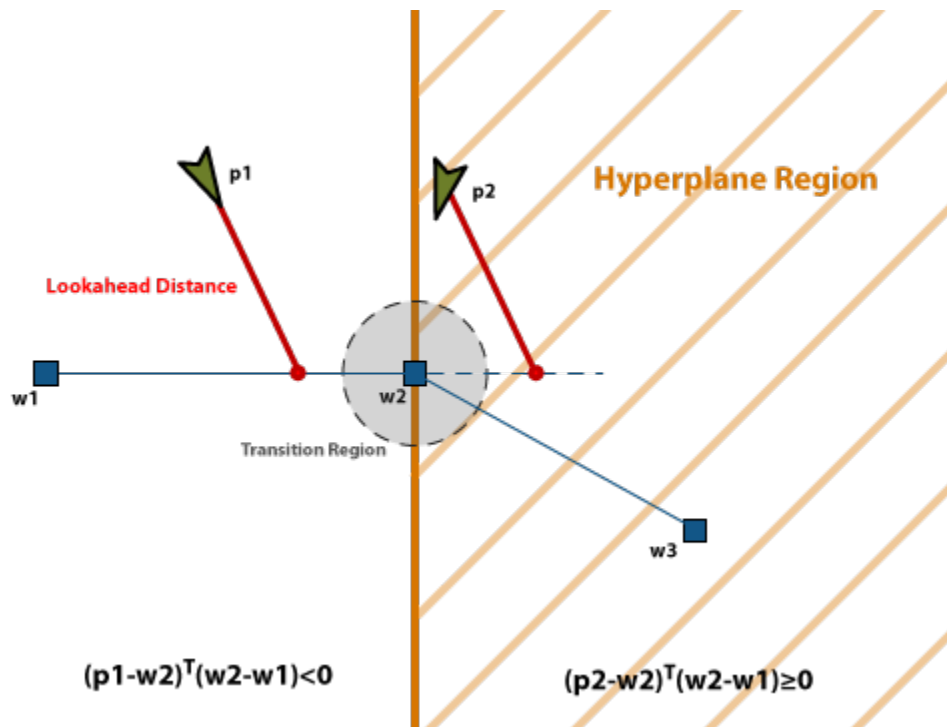
Common to All System Objects

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:

$$(p-w1)^T (w2-w1) \geq 0$$

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.

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 | plotTransforms | state

Objects

fixedwing | multirotor | uavOrbitFollower

Blocks

UAV Guidance Model

Topics

“Approximate High-Fidelity UAV model with UAV Guidance Model block”
“Tuning Waypoint Follower for Fixed-Wing UAV”

Introduced in R2018b

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.SSSSS'` 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.SSSSS'` 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]);
```

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

Introduced in R2020b

waypointTrajectory

Waypoint trajectory generator

Description

The `waypointTrajectory` System object generates trajectories using specified waypoints. When you create the System object, you can optionally specify the time of arrival, velocity, and orientation at each waypoint. See “Algorithms” on page 1-177 for more details.

To generate a trajectory from waypoints:

- 1 Create the `waypointTrajectory` object and set its properties.
- 2 Call the object as if it were a function.

To learn more about how System objects work, see [What Are System Objects?](#)

Creation

Syntax

```
trajectory = waypointTrajectory
trajectory = waypointTrajectory(Waypoints,TimeOfArrival)
trajectory = waypointTrajectory(Waypoints,TimeOfArrival,Name,Value)
```

Description

`trajectory = waypointTrajectory` returns a System object, `trajectory`, that generates a trajectory based on default stationary waypoints.

`trajectory = waypointTrajectory(Waypoints,TimeOfArrival)` specifies the `Waypoints` that the generated trajectory passes through and the `TimeOfArrival` at each waypoint.

`trajectory = waypointTrajectory(Waypoints,TimeOfArrival,Name,Value)` sets each creation argument or property `Name` to the specified `Value`. Unspecified properties and creation arguments have default or inferred values.

Example: `trajectory = waypointTrajectory([10,10,0;20,20,0;20,20,10],[0,0.5,10])` creates a waypoint trajectory System object, `trajectory`, that starts at waypoint `[10,10,0]`, and then passes through `[20,20,0]` after 0.5 seconds and `[20,20,10]` after 10 seconds.

Creation Arguments

Creation arguments are properties which are set during creation of the System object and cannot be modified later. If you do not explicitly set a creation argument value, the property value is inferred.

If you specify any creation argument, then you must specify both the `Waypoints` and `TimeOfArrival` creation arguments. You can specify `Waypoints` and `TimeOfArrival` as value-only arguments or name-value pairs.

Properties

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

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

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

SampleRate — Sample rate of trajectory (Hz)

100 (default) | positive scalar

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

Tunable: Yes

Data Types: double

SamplesPerFrame — Number of samples per output frame

1 (default) | positive scalar integer

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

Tunable: Yes

Data Types: double

Waypoints — Positions in the navigation coordinate system (m)

N -by-3 matrix

Positions in the navigation coordinate system in meters, specified as an N -by-3 matrix. The columns of the matrix correspond to the first, second, and third axes, respectively. The rows of the matrix, N , correspond to individual waypoints.

Dependencies

To set this property, you must also set valid values for the `TimeOfArrival` property.

Data Types: double

TimeOfArrival — Time at each waypoint (s)

N -element column vector of nonnegative increasing numbers

Time corresponding to arrival at each waypoint in seconds, specified as an N -element column vector. The first element of `TimeOfArrival` must be 0. The number of samples, N , must be the same as the number of samples (rows) defined by `Waypoints`.

Dependencies

To set this property, you must also set valid values for the `Waypoints` property.

Data Types: double

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

N -by-3 matrix

Velocity in the navigation coordinate system at each way point in meters per second, specified as an N -by-3 matrix. The columns of the matrix correspond to the first, second, and third axes, respectively.

The number of samples, N , must be the same as the number of samples (rows) defined by `Waypoints`.

If the velocity is specified as a non-zero value, the object automatically calculates the course of the trajectory. If the velocity is specified as zero, the object infers the course of the trajectory from adjacent waypoints.

Dependencies

To set this property, you must also set valid values for the `Waypoints` and `TimeOfArrival` properties.

Data Types: `double`

Course — Horizontal direction of travel (degree)

N -element real vector

Horizontal direction of travel, specified as an N -element real vector in degrees. The number of samples, N , must be the same as the number of samples (rows) defined by `Waypoints`. If neither `Velocities` nor `Course` is specified, course is inferred from the waypoints.

Dependencies

To set this property, the `Velocities` property must not be specified in object creation.

Data Types: `double`

GroundSpeed — Groundspeed at each waypoint (m/s)

N -element real vector

Groundspeed at each waypoint, specified as an N -element real vector in m/s. If the property is not specified, it is inferred from the waypoints. The number of samples, N , must be the same as the number of samples (rows) defined by `Waypoints`.

Dependencies

To set this property, the `Velocities` property must not be specified at object creation.

Data Types: `double`

ClimbRate — Climb rate at each waypoint (m/s)

N -element real vector

Climb Rate at each waypoint, specified as an N -element real vector in degrees. The number of samples, N , must be the same as the number of samples (rows) defined by `Waypoints`. If neither `Velocities` nor `Course` is specified, `climbrate` is inferred from the waypoints.

Dependencies

To set this property, the `Velocities` property must not be specified at object creation.

Data Types: `double`

Orientation — Orientation at each waypoint

N -element quaternion column vector | 3-by-3-by- N array of real numbers

Orientation at each waypoint, specified as an N -element quaternion column vector or 3-by-3-by- N array of real numbers. The number of quaternions or rotation matrices, N , must be the same as the number of samples (rows) defined by `Waypoints`.

If `Orientation` is specified by quaternions, the underlying class must be `double`.

Dependencies

To set this property, you must also set valid values for the `Waypoints` and `TimeOfArrival` properties.

Data Types: `quaternion` | `double`

AutoPitch — Align pitch angle with direction of motion

`false` (default) | `true`

Align pitch angle with the direction of motion, specified as `true` or `false`. When specified as `true`, the pitch angle automatically aligns with the direction of motion. If specified as `false`, the pitch angle is set to zero (level orientation).

Dependencies

To set this property, the `Orientation` property must not be specified at object creation.

AutoBank — Align roll angle to counteract centripetal force

`false` (default) | `true`

Align roll angle to counteract the centripetal force, specified as `true` or `false`. When specified as `true`, the roll angle automatically counteract the centripetal force. If specified as `false`, the roll angle is set to zero (flat orientation).

Dependencies

To set this property, the `Orientation` property must not be specified at object creation.

ReferenceFrame — Reference frame of trajectory

`'NED'` (default) | `'ENU'`

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

Usage

Syntax

```
[position,orientation,velocity,acceleration,angularVelocity] = trajectory()
```

Description

`[position,orientation,velocity,acceleration,angularVelocity] = trajectory()` outputs a frame of trajectory data based on specified creation arguments and properties.

Output Arguments

position — Position in local navigation coordinate system (m)

M-by-3 matrix

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

M is specified by the `SamplesPerFrame` property.

Data Types: `double`

orientation — Orientation in local navigation coordinate system

M-element quaternion column vector | 3-by-3-by-*M* real array

Orientation in the local navigation coordinate system, returned as an *M*-by-1 quaternion column vector or a 3-by-3-by-*M* real array.

Each quaternion or 3-by-3 rotation matrix is a frame rotation from the local navigation coordinate system to the current body coordinate system.

M is specified by the SamplesPerFrame property.

Data Types: double

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

M-by-3 matrix

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

M is specified by the SamplesPerFrame property.

Data Types: double

acceleration — Acceleration in local navigation coordinate system (m/s²)

M-by-3 matrix

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

M is specified by the SamplesPerFrame property.

Data Types: double

angularVelocity — Angular velocity in local navigation coordinate system (rad/s)

M-by-3 matrix

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

M is specified by the SamplesPerFrame property.

Data Types: double

Object Functions

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

```
release(obj)
```

Specific to waypointTrajectory

waypointInfo	Get waypoint information table
lookupPose	Obtain pose information for certain time
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=2x2 table
   TimeOfArrival   Waypoints
   _____   _____
           0           0     0     0
           1           0     0     0
```

Create Square Trajectory

Create a square trajectory and examine the relationship between waypoint constraints, sample rate, and the generated trajectory.

Create a square trajectory by defining the vertices of the square. Define the orientation at each waypoint as pointing in the direction of motion. Specify a 1 Hz sample rate and use the default `SamplesPerFrame` of 1.

```
waypoints = [0,0,0; ... % Initial position
            0,1,0; ...
            1,1,0; ...
            1,0,0; ...
            0,0,0]; % Final position

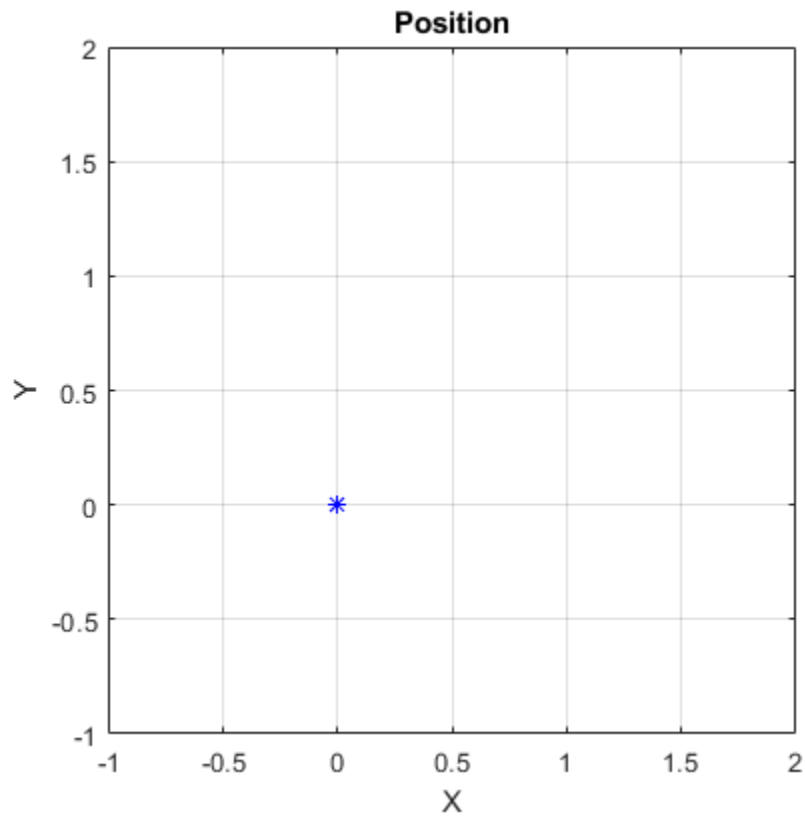
toa = 0:4; % time of arrival

orientation = quaternion([0,0,0; ...
                        45,0,0; ...
                        135,0,0; ...
                        225,0,0; ...
                        0,0,0], ...
                        'eulerd','ZYX','frame');

trajectory = waypointTrajectory(waypoints, ...
                                'TimeOfArrival',toa, ...
                                'Orientation',orientation, ...
                                'SampleRate',1);
```

Create a figure and plot the initial position of the platform.

```
figure(1)
plot(waypoints(1,1),waypoints(1,2),'b*')
title('Position')
axis([-1,2,-1,2])
axis square
xlabel('X')
ylabel('Y')
grid on
hold on
```

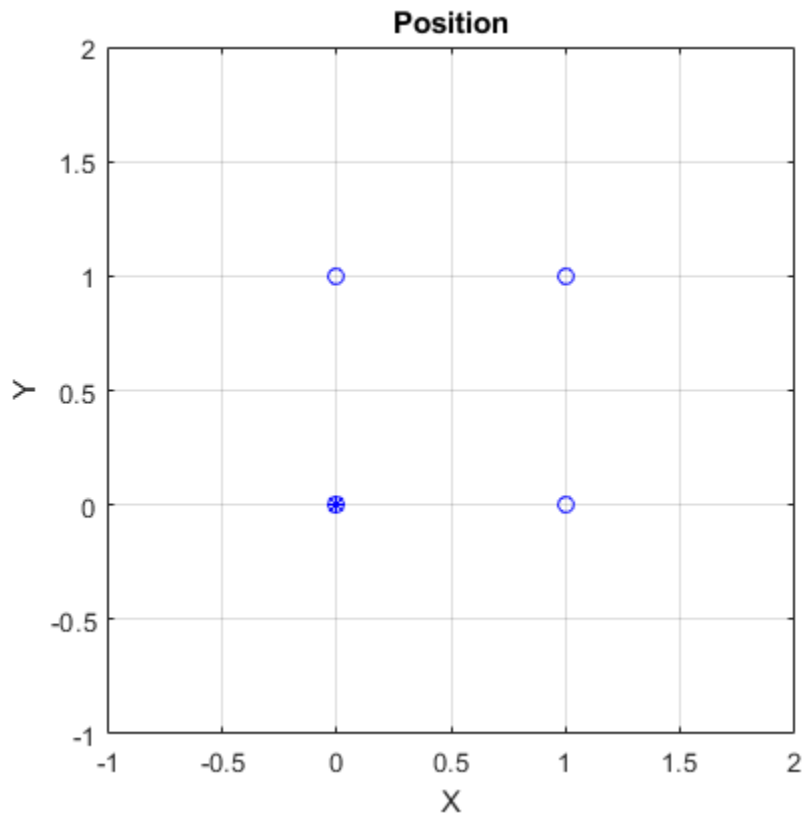



In a loop, step through the trajectory to output the current position and current orientation. Plot the current position and log the orientation. Use `pause` to mimic real-time processing.

```
orientationLog = zeros(toa(end)*trajectory.SampleRate,1,'quaternion');
count = 1;
while ~isDone(trajectory)
    [currentPosition,orientationLog(count)] = trajectory();

    plot(currentPosition(1),currentPosition(2),'bo')

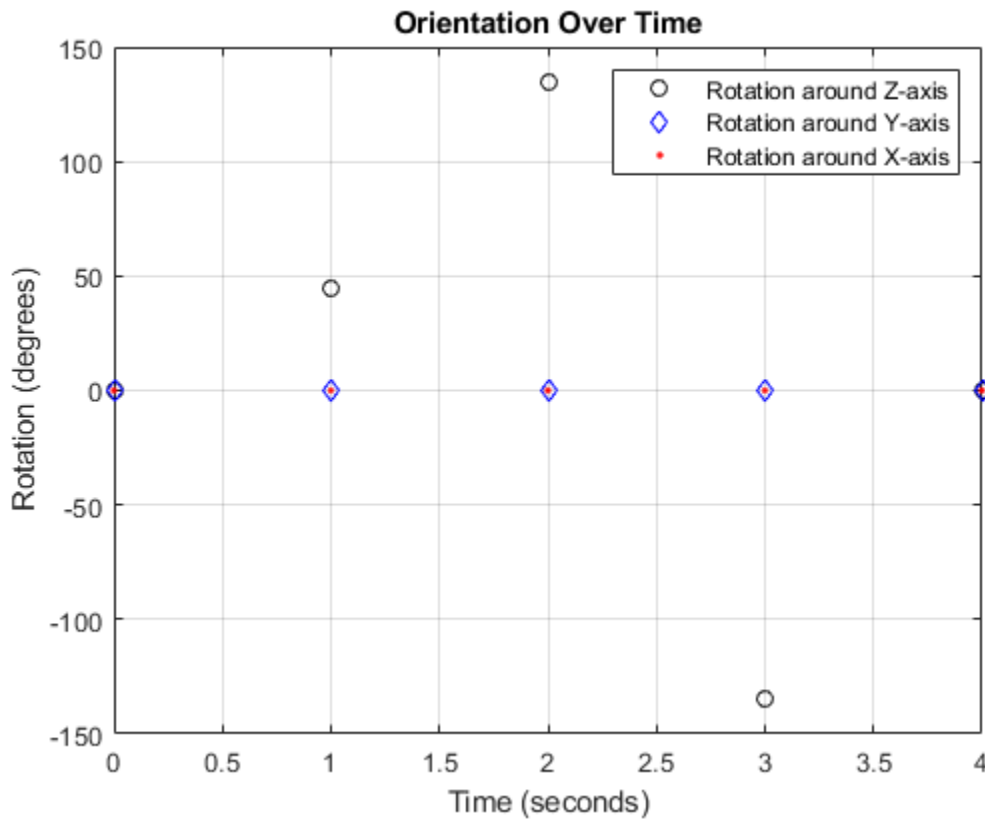
    pause(trajectory.SamplesPerFrame/trajectory.SampleRate)
    count = count + 1;
end
hold off
```



Convert the orientation quaternions to Euler angles for easy interpretation, and then plot orientation over time.

```
figure(2)
eulerAngles = eulerd([orientation(1);orientationLog], 'ZYX', 'frame');
plot(toa,eulerAngles(:,1),'ko', ...
     toa,eulerAngles(:,2),'bd', ...
     toa,eulerAngles(:,3),'r.');
```

title('Orientation Over Time')
 legend('Rotation around Z-axis','Rotation around Y-axis','Rotation around X-axis')
 xlabel('Time (seconds)')
 ylabel('Rotation (degrees)')
 grid on



So far, the trajectory object has only output the waypoints that were specified during construction. To interpolate between waypoints, increase the sample rate to a rate faster than the time of arrivals of the waypoints. Set the trajectory sample rate to 100 Hz and call reset.

```
trajectory.SampleRate = 100;
reset(trajectory)
```

Create a figure and plot the initial position of the platform. In a loop, step through the trajectory to output the current position and current orientation. Plot the current position and log the orientation. Use pause to mimic real-time processing.

```
figure(1)
plot(waypoints(1,1),waypoints(1,2),'b*')
title('Position')
axis([-1,2,-1,2])
axis square
xlabel('X')
ylabel('Y')
grid on
hold on

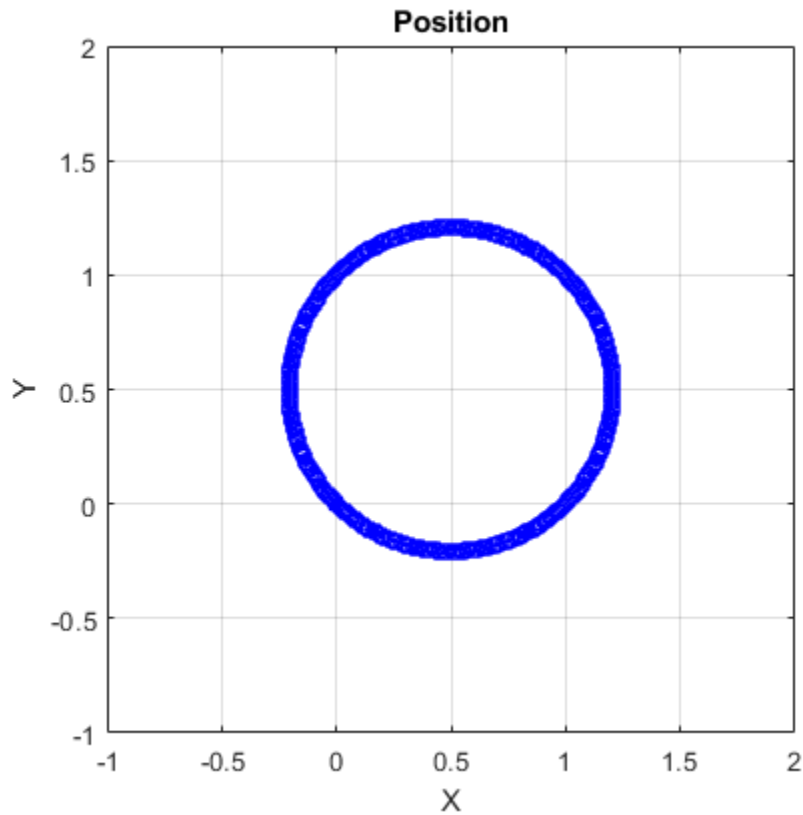
orientationLog = zeros(toa(end)*trajectory.SampleRate,1,'quaternion');
count = 1;
while ~isDone(trajectory)
    [currentPosition,orientationLog(count)] = trajectory();

    plot(currentPosition(1),currentPosition(2),'bo')
```

```

    pause(trajjectory.SamplesPerFrame/trajjectory.SampleRate)
    count = count + 1;
end
hold off

```



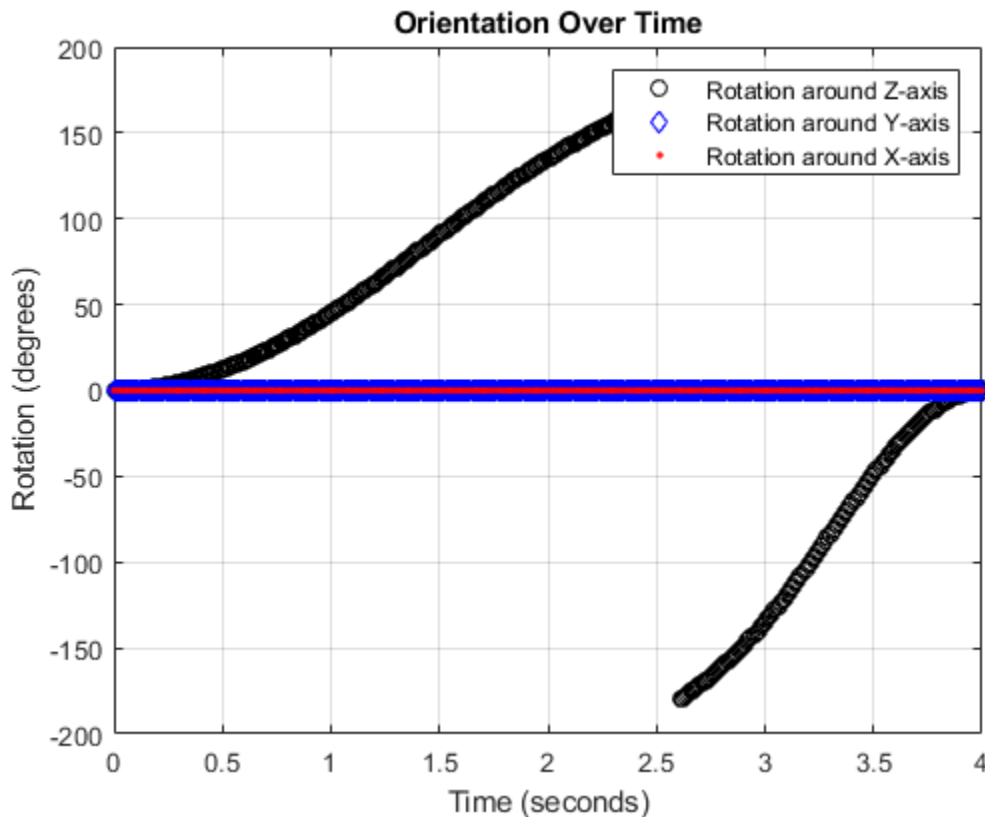
The trajectory output now appears circular. This is because the `waypointTrajectory System object™` minimizes the acceleration and angular velocity when interpolating, which results in smoother, more realistic motions in most scenarios.

Convert the orientation quaternions to Euler angles for easy interpretation, and then plot orientation over time. The orientation is also interpolated.

```

figure(2)
eulerAngles = eulerd([orientation(1);orientationLog], 'ZYX', 'frame');
t = 0:1/trajjectory.SampleRate:4;
plot(t,eulerAngles(:,1),'ko', ...
      t,eulerAngles(:,2),'bd', ...
      t,eulerAngles(:,3),'r. ');
title('Orientation Over Time')
legend('Rotation around Z-axis','Rotation around Y-axis','Rotation around X-axis')
xlabel('Time (seconds)')
ylabel('Rotation (degrees)')
grid on

```



The waypointTrajectory algorithm interpolates the waypoints to create a smooth trajectory. To return to the square trajectory, provide more waypoints, especially around sharp changes. To track corresponding times, waypoints, and orientation, specify all the trajectory info in a single matrix.

```

% Time, Waypoint, Orientation
trajectoryInfo = [0, 0,0,0, 0,0,0; ... % Initial position
                 0.1, 0,0.1,0, 0,0,0; ...
                 0.9, 0,0.9,0, 0,0,0; ...
                 1, 0,1,0, 45,0,0; ...
                 1.1, 0.1,1,0, 90,0,0; ...
                 1.9, 0.9,1,0, 90,0,0; ...
                 2, 1,1,0, 135,0,0; ...
                 2.1, 1,0.9,0, 180,0,0; ...
                 2.9, 1,0.1,0, 180,0,0; ...
                 3, 1,0,0, 225,0,0; ...
                 3.1, 0.9,0,0, 270,0,0; ...
                 3.9, 0.1,0,0, 270,0,0; ...
                 4, 0,0,0, 270,0,0]; % Final position

trajectory = waypointTrajectory(trajectoryInfo(:,2:4), ...
    'TimeOfArrival',trajectoryInfo(:,1), ...
    'Orientation',quaternion(trajectoryInfo(:,5:end),'eulerd','ZYX','frame'), ...
    'SampleRate',100);

```

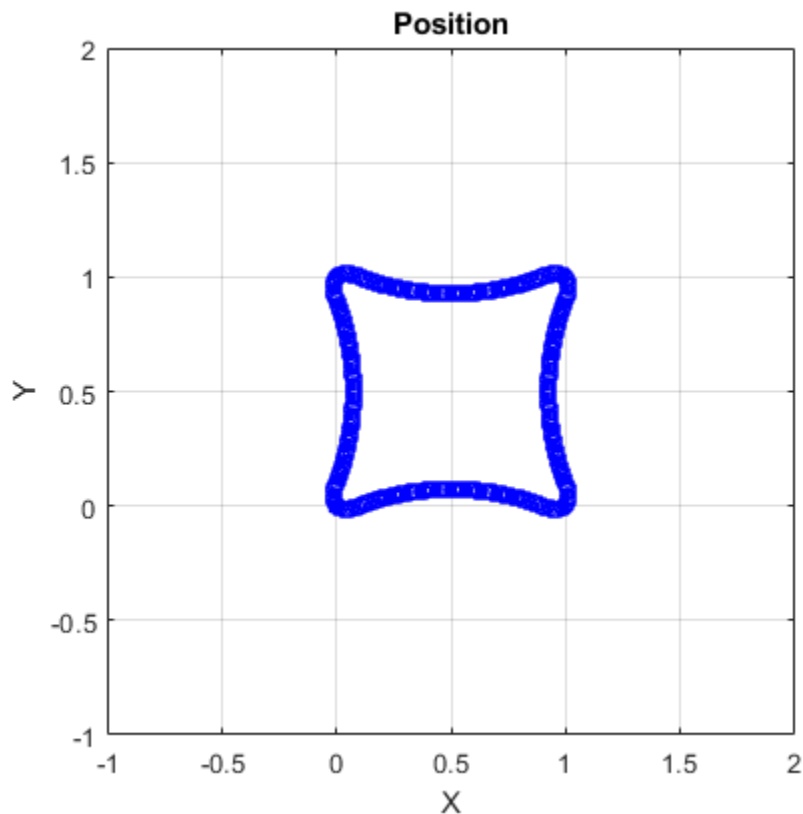
Create a figure and plot the initial position of the platform. In a loop, step through the trajectory to output the current position and current orientation. Plot the current position and log the orientation. Use `pause` to mimic real-time processing.

```
figure(1)
plot(waypoints(1,1),waypoints(1,2), 'b*')
title('Position')
axis([-1,2,-1,2])
axis square
xlabel('X')
ylabel('Y')
grid on
hold on

orientationLog = zeros(toa(end)*trajectory.SampleRate,1, 'quaternion');
count = 1;
while ~isDone(trajectory)
    [currentPosition,orientationLog(count)] = trajectory();

    plot(currentPosition(1),currentPosition(2), 'bo')

    pause(trajectory.SamplesPerFrame/trajectory.SampleRate)
    count = count+1;
end
hold off
```

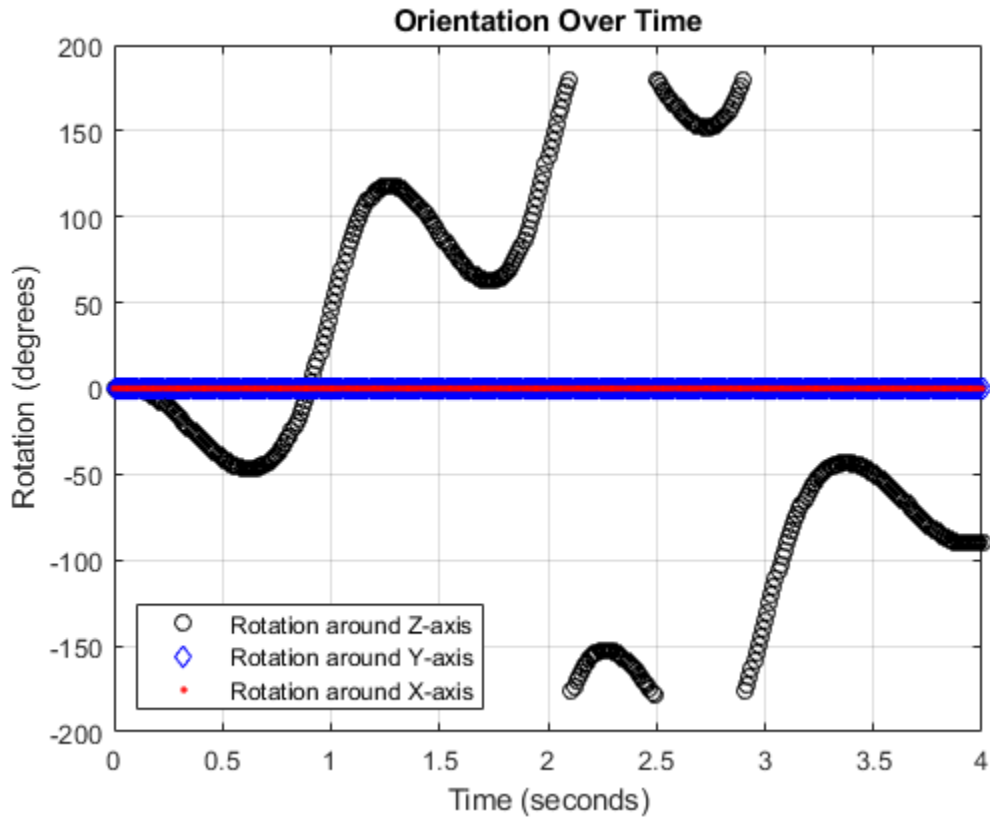


The trajectory output now appears more square-like, especially around the vertices with waypoints.

Convert the orientation quaternions to Euler angles for easy interpretation, and then plot orientation over time.

```
figure(2)
eulerAngles = eulerd([orientation(1);orientationLog],'ZYX','frame');
t = 0:1/trajectory.SampleRate:4;
eulerAngles = plot(t,eulerAngles(:,1),'ko', ...
                  t,eulerAngles(:,2),'bd', ...
                  t,eulerAngles(:,3),'r.');
```

```
title('Orientation Over Time')
legend('Rotation around Z-axis', ...
       'Rotation around Y-axis', ...
       'Rotation around X-axis', ...
       'Location', 'SouthWest')
xlabel('Time (seconds)')
ylabel('Rotation (degrees)')
grid on
```



Create Arc Trajectory

This example shows how to create an arc trajectory using the `waypointTrajectory` System object™. `waypointTrajectory` creates a path through specified waypoints that minimizes acceleration and angular velocity. After creating an arc trajectory, you restrict the trajectory to be within preset bounds.

Create an Arc Trajectory

Define a constraints matrix consisting of waypoints, times of arrival, and orientation for an arc trajectory. The generated trajectory passes through the waypoints at the specified times with the specified orientation. The `waypointTrajectory` System object requires orientation to be specified using quaternions or rotation matrices. Convert the Euler angles saved in the constraints matrix to quaternions when specifying the `Orientation` property.

```
% Arrival, Waypoints, Orientation
constraints = [0,    20,20,0,    90,0,0;
              3,    50,20,0,    90,0,0;
              4,    58,15.5,0, 162,0,0;
              5.5, 59.5,0,0    180,0,0];

trajectory = waypointTrajectory(constraints(:,2:4), ...
    'TimeOfArrival',constraints(:,1), ...
    'Orientation',quaternion(constraints(:,5:7),'eulerd','ZYX','frame'));
```

Call `waypointInfo` on `trajectory` to return a table of your specified constraints. The creation properties `Waypoints`, `TimeOfArrival`, and `Orientation` are variables of the table. The table is convenient for indexing while plotting.

```
tInfo = waypointInfo(trajectory)
```

```
tInfo =
```

```
4x3 table
```

TimeOfArrival	Waypoints			Orientation
0	20	20	0	{1x1 quaternion}
3	50	20	0	{1x1 quaternion}
4	58	15.5	0	{1x1 quaternion}
5.5	59.5	0	0	{1x1 quaternion}

The trajectory object outputs the current position, velocity, acceleration, and angular velocity at each call. Call `trajectory` in a loop and plot the position over time. Cache the other outputs.

```
figure(1)
plot(tInfo.Waypoints(1,1),tInfo.Waypoints(1,2),'b*')
title('Position')
axis([20,65,0,25])
xlabel('North')
ylabel('East')
grid on
daspect([1 1 1])
hold on

orient = zeros(tInfo.TimeOfArrival(end)*trajectory.SampleRate,1,'quaternion');
vel = zeros(tInfo.TimeOfArrival(end)*trajectory.SampleRate,3);
acc = vel;
angVel = vel;

count = 1;
while ~isDone(trajectory)
```

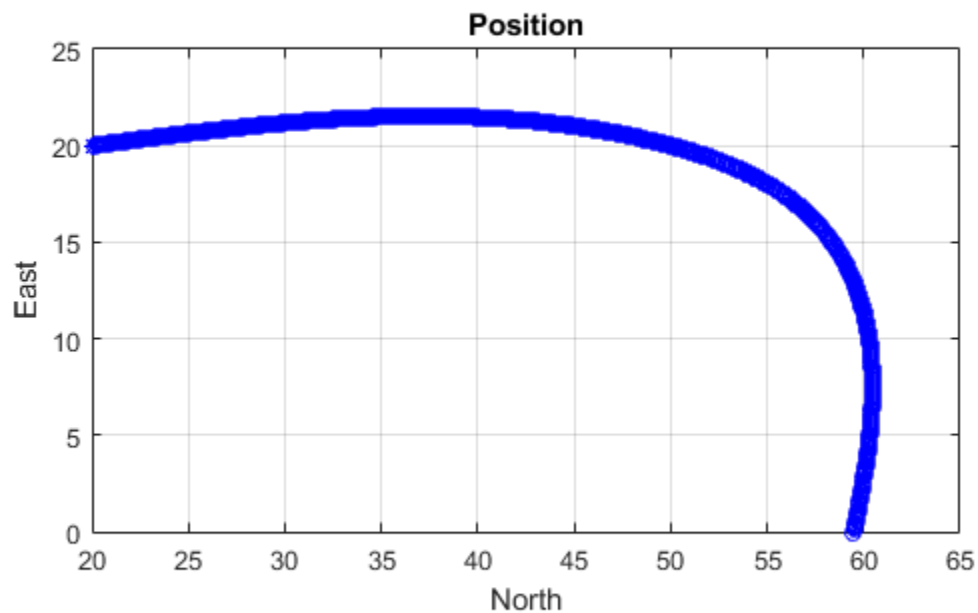


```

[pos,orient(count),vel(count,:),acc(count,:),angVel(count,:)] = trajectory();
plot(pos(1),pos(2),'bo')

pause(trajectory.SamplesPerFrame/trajectory.SampleRate)
count = count + 1;
end

```



Inspect the orientation, velocity, acceleration, and angular velocity over time. The `waypointTrajectory` System object™ creates a path through the specified constraints that minimized acceleration and angular velocity.

```

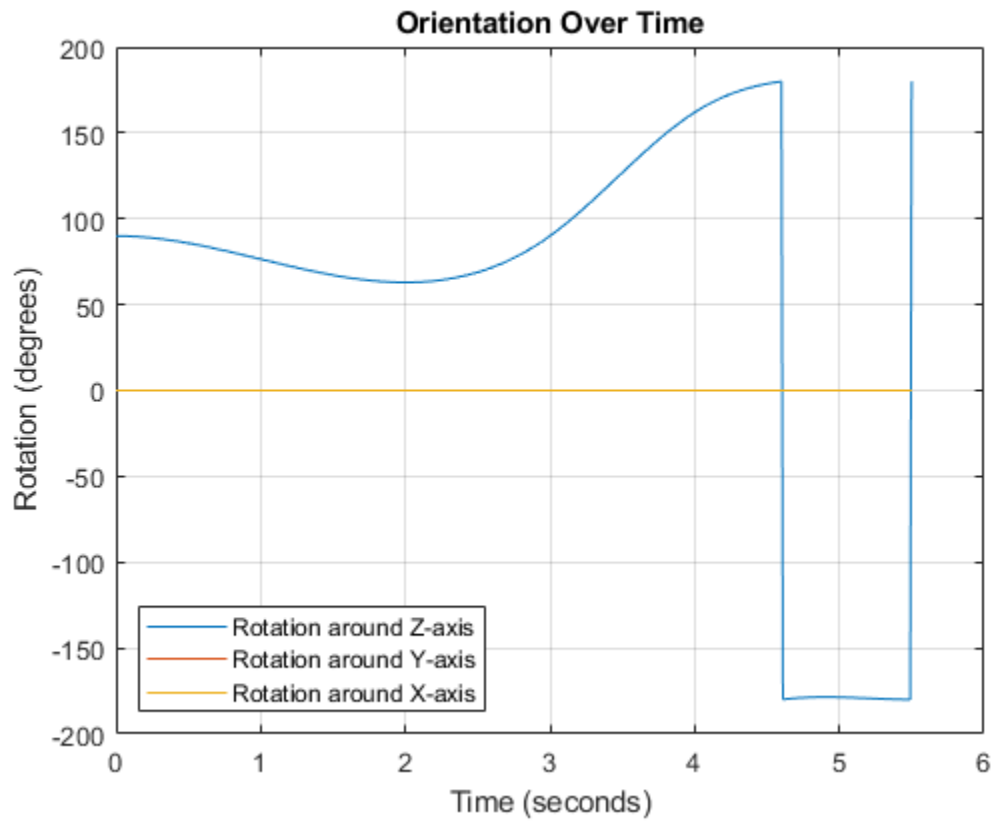
figure(2)
timeVector = 0:(1/trajectory.SampleRate):tInfo.TimeOfArrival(end);
eulerAngles = eulerd([tInfo.Orientation{1};orient],'ZYX','frame');
plot(timeVector,eulerAngles(:,1), ...
      timeVector,eulerAngles(:,2), ...
      timeVector,eulerAngles(:,3));
title('Orientation Over Time')
legend('Rotation around Z-axis', ...
       'Rotation around Y-axis', ...
       'Rotation around X-axis', ...
       'Location','southwest')
xlabel('Time (seconds)')
ylabel('Rotation (degrees)')
grid on

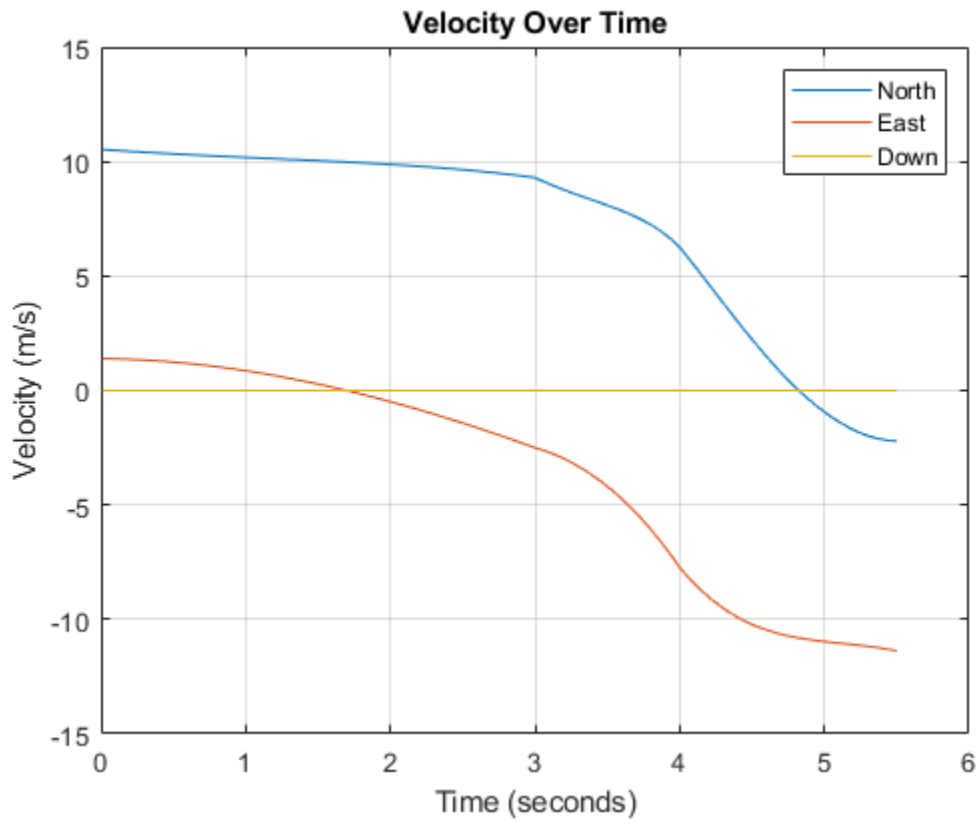
```

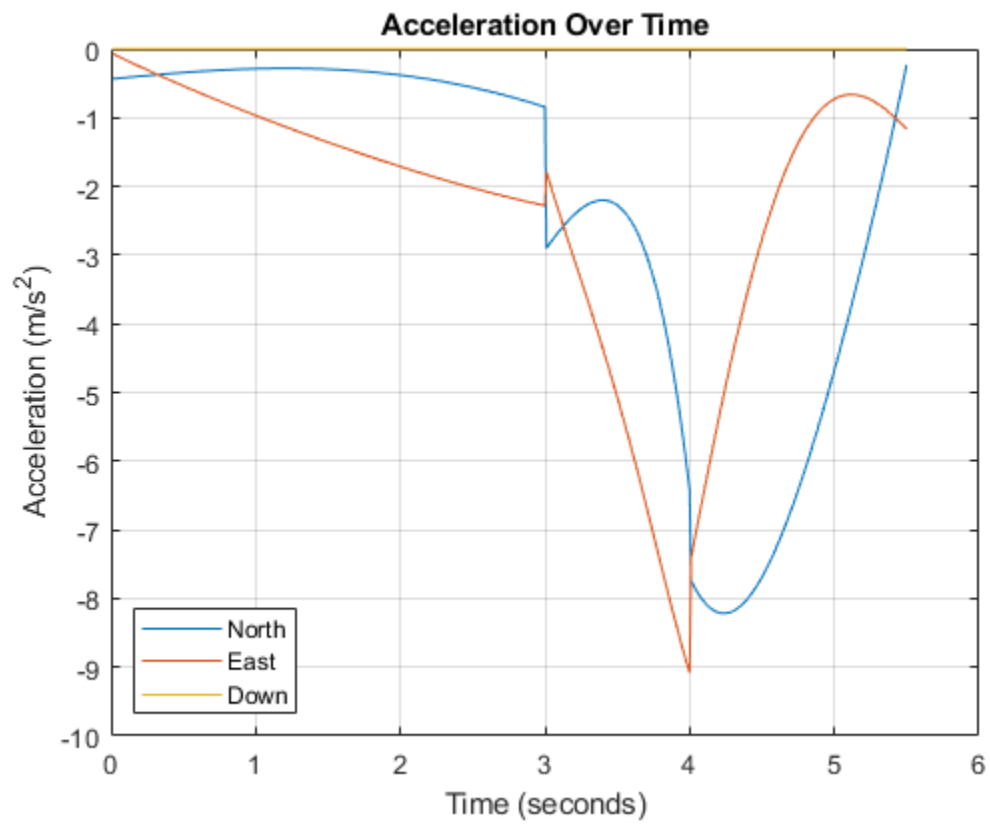
```
figure(3)
plot(timeVector(2:end),vel(:,1), ...
      timeVector(2:end),vel(:,2), ...
      timeVector(2:end),vel(:,3));
title('Velocity Over Time')
legend('North','East','Down')
xlabel('Time (seconds)')
ylabel('Velocity (m/s)')
grid on

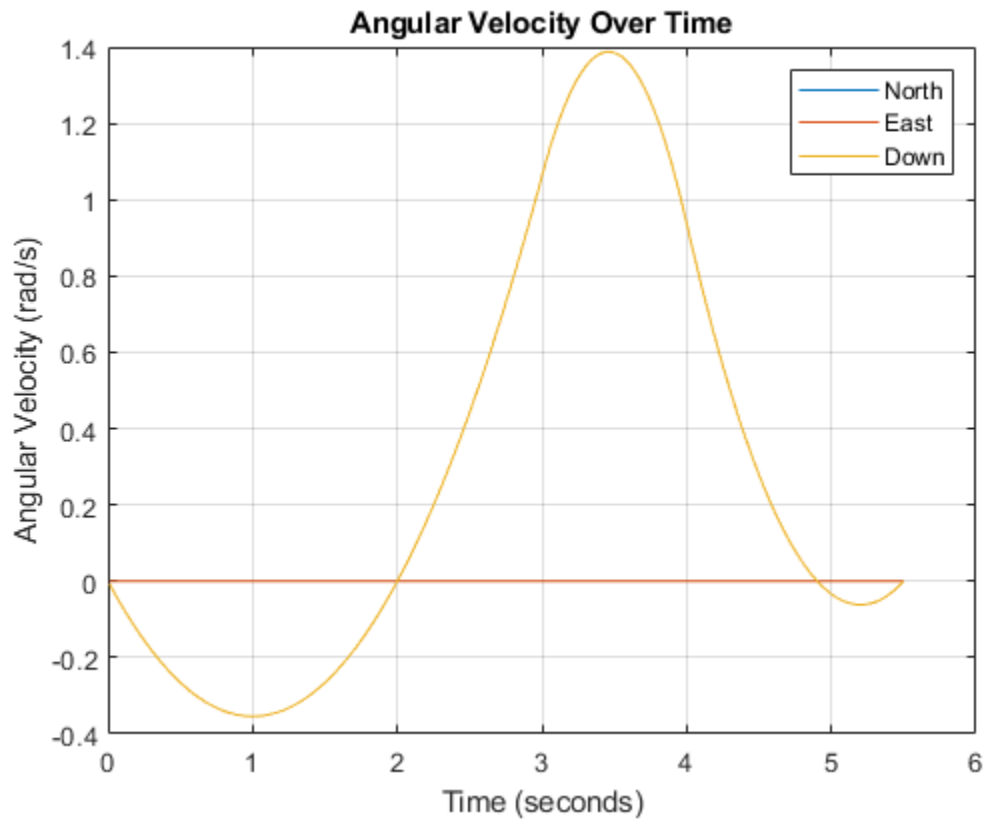
figure(4)
plot(timeVector(2:end),acc(:,1), ...
      timeVector(2:end),acc(:,2), ...
      timeVector(2:end),acc(:,3));
title('Acceleration Over Time')
legend('North','East','Down','Location','southwest')
xlabel('Time (seconds)')
ylabel('Acceleration (m/s^2)')
grid on

figure(5)
plot(timeVector(2:end),angVel(:,1), ...
      timeVector(2:end),angVel(:,2), ...
      timeVector(2:end),angVel(:,3));
title('Angular Velocity Over Time')
legend('North','East','Down')
xlabel('Time (seconds)')
ylabel('Angular Velocity (rad/s)')
grid on
```









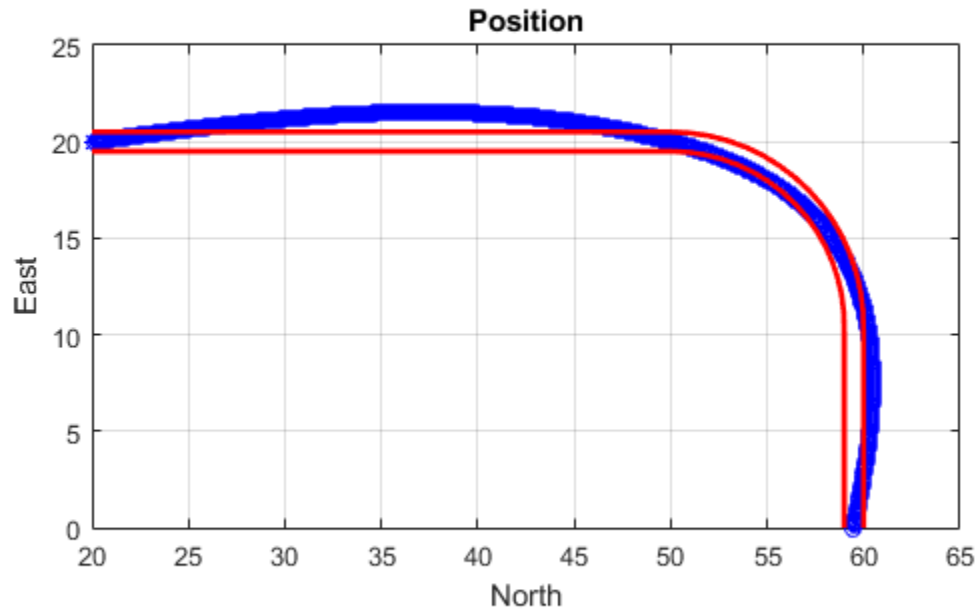
Restrict Arc Trajectory Within Preset Bounds

You can specify additional waypoints to create trajectories within given bounds. Create upper and lower bounds for the arc trajectory.

```
figure(1)
xUpperBound = [(20:50)';50+10*sin(0:0.1:pi/2)';60*ones(11,1)];
yUpperBound = [20.5.*ones(31,1);10.5+10*cos(0:0.1:pi/2)';(10:-1:0)'];

xLowerBound = [(20:49)';50+9*sin(0:0.1:pi/2)';59*ones(11,1)];
yLowerBound = [19.5.*ones(30,1);10.5+9*cos(0:0.1:pi/2)';(10:-1:0)'];

plot(xUpperBound,yUpperBound,'r','LineWidth',2);
plot(xLowerBound,yLowerBound,'r','LineWidth',2)
```



To create a trajectory within the bounds, add additional waypoints. Create a new `waypointTrajectory` System object™, and then call it in a loop to plot the generated trajectory. Cache the orientation, velocity, acceleration, and angular velocity output from the trajectory object.

```

% Time, Waypoint, Orientation
constraints = [0, 20,20,0, 90,0,0;
              1.5, 35,20,0, 90,0,0;
              2.5, 45,20,0, 90,0,0;
              3, 50,20,0, 90,0,0;
              3.3, 53,19.5,0, 108,0,0;
              3.6, 55.5,18.25,0, 126,0,0;
              3.9, 57.5,16,0, 144,0,0;
              4.2, 59,14,0, 162,0,0;
              4.5, 59.5,10,0, 180,0,0;
              5, 59.5,5,0, 180,0,0;
              5.5, 59.5,0,0, 180,0,0];

trajectory = waypointTrajectory(constraints(:,2:4), ...
    'TimeOfArrival',constraints(:,1), ...
    'Orientation',quaternion(constraints(:,5:7),'eulerd','ZYX','frame'));
tInfo = waypointInfo(trajectory);

figure(1)
plot(tInfo.Waypoints(1,1),tInfo.Waypoints(1,2),'b*')

count = 1;

```

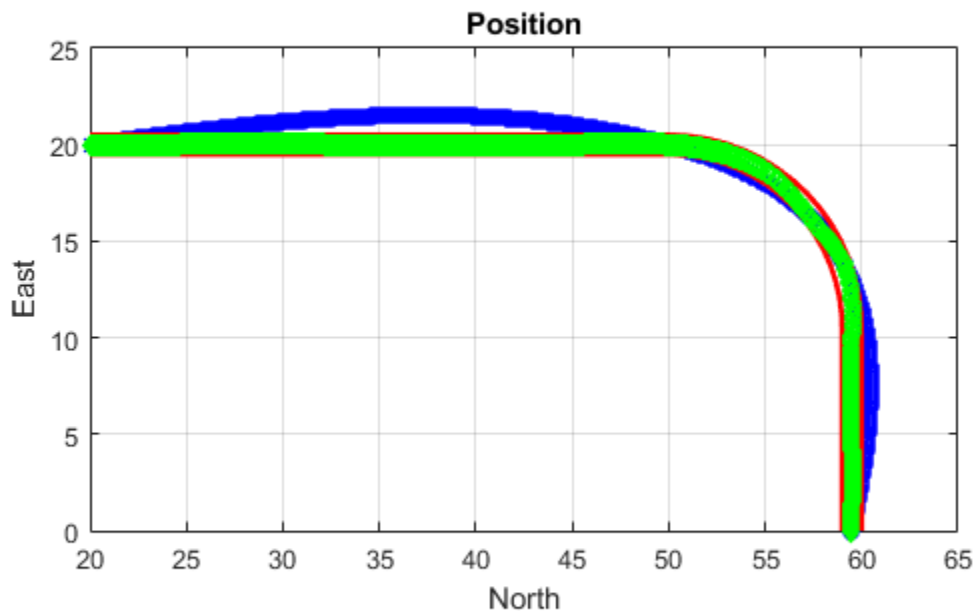
```

while ~isDone(trajjectory)
    [pos,orient(count),vel(count,:),acc(count,:),angVel(count,:)] = trajjectory();

    plot(pos(1),pos(2),'gd')

    pause(trajjectory.SamplesPerFrame/trajjectory.SampleRate)
    count = count + 1;
end

```



The generated trajectory now fits within the specified boundaries. Visualize the orientation, velocity, acceleration, and angular velocity of the generated trajectory.

```

figure(2)
timeVector = 0:(1/trajjectory.SampleRate):tInfo.TimeOfArrival(end);
eulerAngles = eulerd(orient,'ZYX','frame');
plot(timeVector(2:end),eulerAngles(:,1), ...
      timeVector(2:end),eulerAngles(:,2), ...
      timeVector(2:end),eulerAngles(:,3));
title('Orientation Over Time')
legend('Rotation around Z-axis', ...
       'Rotation around Y-axis', ...
       'Rotation around X-axis', ...
       'Location','southwest')
xlabel('Time (seconds)')
ylabel('Rotation (degrees)')
grid on

figure(3)

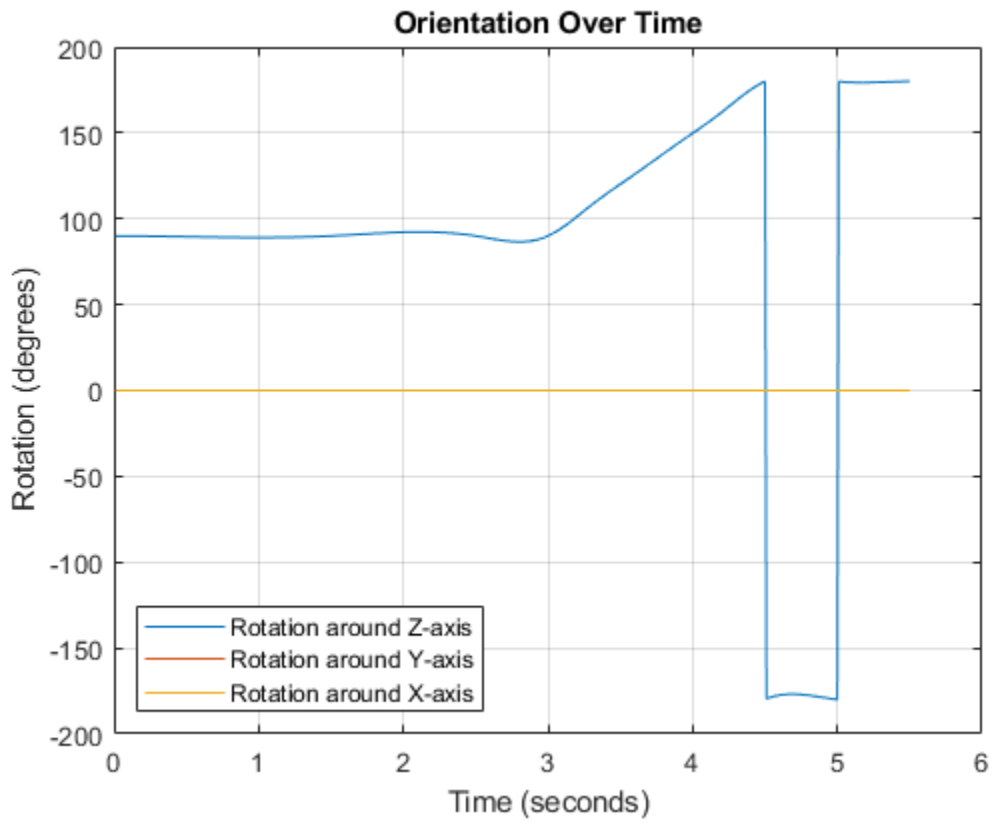
```

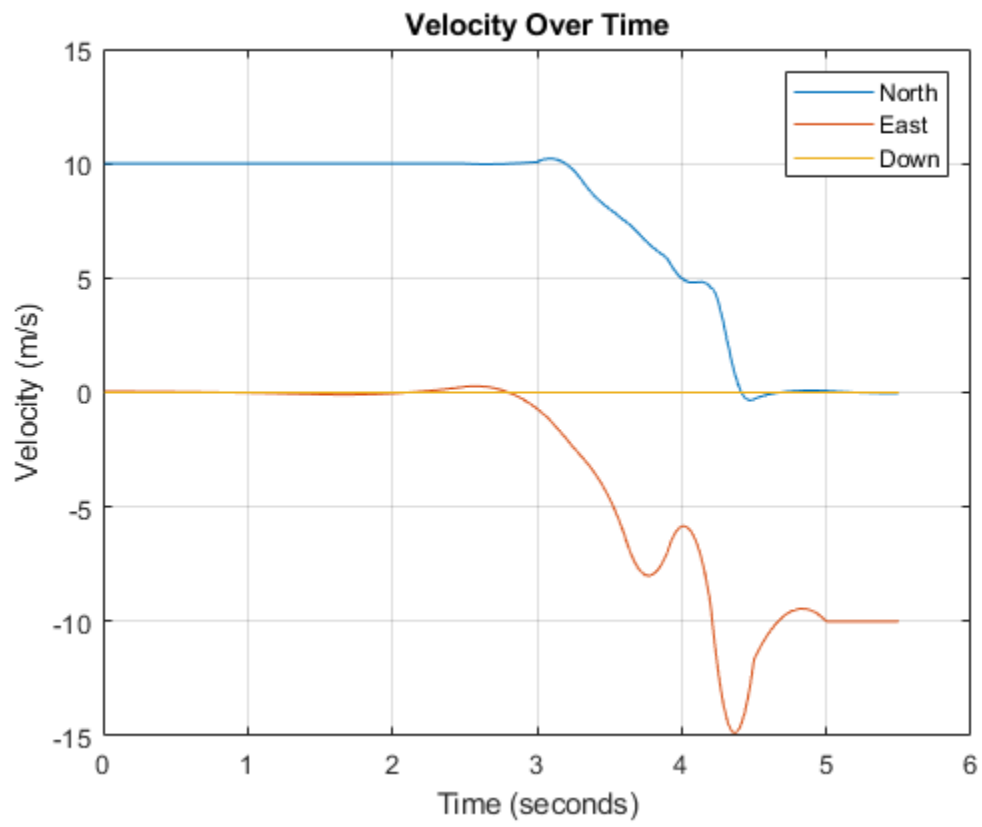


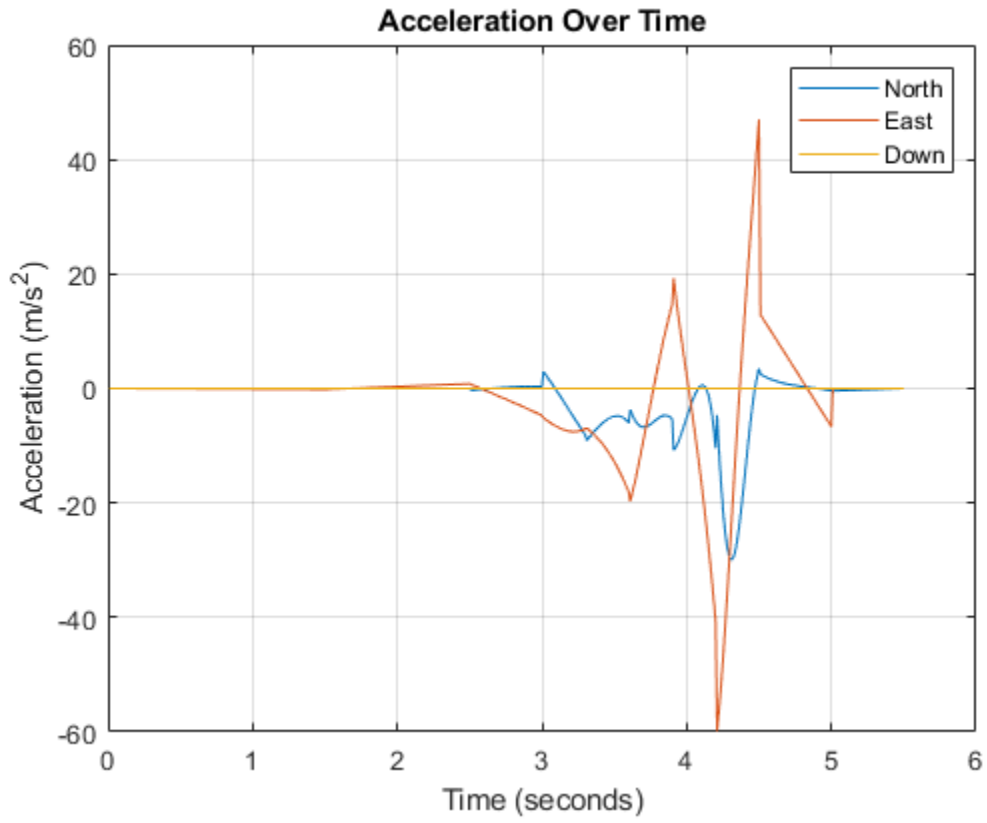
```
plot(timeVector(2:end),vel(:,1), ...
      timeVector(2:end),vel(:,2), ...
      timeVector(2:end),vel(:,3));
title('Velocity Over Time')
legend('North','East','Down')
xlabel('Time (seconds)')
ylabel('Velocity (m/s)')
grid on

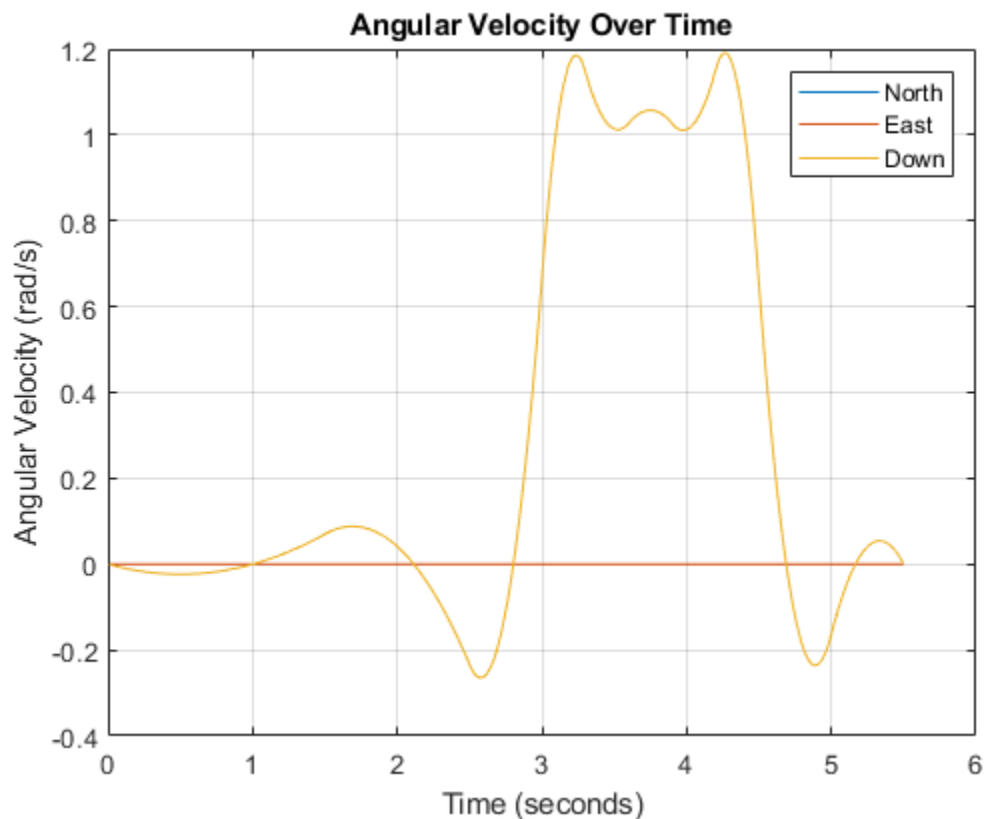
figure(4)
plot(timeVector(2:end),acc(:,1), ...
      timeVector(2:end),acc(:,2), ...
      timeVector(2:end),acc(:,3));
title('Acceleration Over Time')
legend('North','East','Down')
xlabel('Time (seconds)')
ylabel('Acceleration (m/s^2)')
grid on

figure(5)
plot(timeVector(2:end),angVel(:,1), ...
      timeVector(2:end),angVel(:,2), ...
      timeVector(2:end),angVel(:,3));
title('Angular Velocity Over Time')
legend('North','East','Down')
xlabel('Time (seconds)')
ylabel('Angular Velocity (rad/s)')
grid on
```









Note that while the generated trajectory now fits within the spatial boundaries, the acceleration and angular velocity of the trajectory are somewhat erratic. This is due to over-specifying waypoints.

Algorithms

The `waypointTrajectory` System object defines a trajectory that smoothly passes through waypoints. The trajectory connects the waypoints through an interpolation that assumes the gravity direction expressed in the trajectory reference frame is constant. Generally, you can use `waypointTrajectory` to model platform or vehicle trajectories within a hundreds of kilometers distance span.

The planar path of the trajectory (the x - y plane projection) consists of piecewise, clothoid curves. The curvature of the curve between two consecutive waypoints varies linearly with the curve length between them. The tangent direction of the path at each waypoint is chosen to minimize discontinuities in the curvature, unless the course is specified explicitly via the `Course` property or implicitly via the `Velocities` property. Once the path is established, the object uses cubic Hermite interpolation to compute the location of the vehicle throughout the path as a function of time and the planar distance travelled.

The normal component (z -component) of the trajectory is subsequently chosen to satisfy a shape-preserving piecewise spline (PCHIP) unless the climb rate is specified explicitly via the `ClimbRate` property or the third column of the `Velocities` property. Choose the sign of the climb rate based on the selected `ReferenceFrame`:

- When an 'ENU' reference frame is selected, specifying a positive climb rate results in an increasing value of z.
- When an 'NED' reference frame is selected, specifying a positive climb rate results in a decreasing value of z.

You can define the orientation of the vehicle through the path in two primary ways:

- If the **Orientation** property is specified, then the object uses a piecewise-cubic, quaternion spline to compute the orientation along the path as a function of time.
- If the **Orientation** property is not specified, then the yaw of the vehicle is always aligned with the path. The roll and pitch are then governed by the **AutoBank** and **AutoPitch** property values, respectively.

AutoBank	AutoPitch	Description
false	false	The vehicle is always level (zero pitch and roll). This is typically used for large marine vessels.
false	true	The vehicle pitch is aligned with the path, and its roll is always zero. This is typically used for ground vehicles.
true	false	The vehicle pitch and roll are chosen so that its local z-axis is aligned with the net acceleration (including gravity). This is typically used for rotary-wing craft.
true	true	The vehicle roll is chosen so that its local transverse plane aligns with the net acceleration (including gravity). The vehicle pitch is aligned with the path. This is typically used for two-wheeled vehicles and fixed-wing aircraft.

Extended Capabilities

C/C++ Code Generation

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

The object function, `waypointInfo`, does not support code generation.

Usage notes and limitations:

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

See Also

Introduced in R2020b

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

Example: `'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.

Feature	Description						
Datatip	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 border="1"> <thead> <tr> <th>Point Cloud Data</th> <th>Data Value Properties</th> </tr> </thead> <tbody> <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> </tbody> </table>	Point Cloud Data	Data Value Properties	Depth image (RGB-D sensor)	Color, row, column	Lidar	Intensity, range, azimuth angle, elevation angle, row, column
	Point Cloud Data	Data Value Properties					
Depth image (RGB-D sensor)	Color, row, column						
Lidar	Intensity, range, azimuth angle, elevation angle, row, column						

Feature	Description
Background color	Click Rotate and then right-click in the figure for background options.
Colormap value	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.
View	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.

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.

See Also

pointCloud

Introduced in R2020b

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.

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.

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.

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 <code>z</code> -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:

- 1-by-3 RGB vector
- short name of a MATLAB ColorSpec color, such as 'b'
- long name of a MATLAB ColorSpec color, such as 'blue'
- M -by-3 matrix
- M -by- N -by-3 matrix

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 MATLAB ColorSpec 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 <code>colorMap</code> .

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 1-189. Use “Object Functions” on page 1-190 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 1-189.

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
<i>M</i> -by-3 array	<i>M</i> -by-3 array containing RGB values for each point
<i>M</i> -by- <i>N</i> -by-3 array	<i>M</i> -by- <i>N</i> -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
<i>M</i> -by-3 array	<i>M</i> -by-3 array, where each row contains a corresponding normal vector.
<i>M</i> -by- <i>N</i> -by-3 array	<i>M</i> -by- <i>N</i> -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
<i>M</i> -by-3 array	<i>M</i> -by-1 vector, where each row contains a corresponding intensity value.

Coordinates	Intensity
<i>M</i> -by- <i>N</i> -by-3 array	<i>M</i> -by- <i>N</i> 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

<code>findNearestNeighbors</code>	Find nearest neighbors of a point in point cloud
<code>findNeighborsInRadius</code>	Find neighbors within a radius of a point in the point cloud
<code>findPointsInROI</code>	Find points within a region of interest in the point cloud
<code>removeInvalidPoints</code>	Remove invalid points from point cloud
<code>select</code>	Select points in point cloud
<code>copy</code>	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)`

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

Functions

`pcplayer`

Introduced in R2020b

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.

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

Example: `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.

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

pointCloud

Introduced in R2020b

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.

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

Example: `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.

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

pointCloud

Introduced in R2020b

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.

Input Arguments

ptCloud — Point cloud

pointCloud object

Point cloud, specified as a pointCloud object.

roi — Region of interest

six-element vector

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.

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

pointCloud

Introduced in R2020b

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.

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.

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

pointCloud

Introduced in R2020b

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.

`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.

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

pointCloud

Introduced in R2020b

Methods

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

This example shows how to create an `extendedObjectMesh` object and transform the object in a way defined by a given 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.

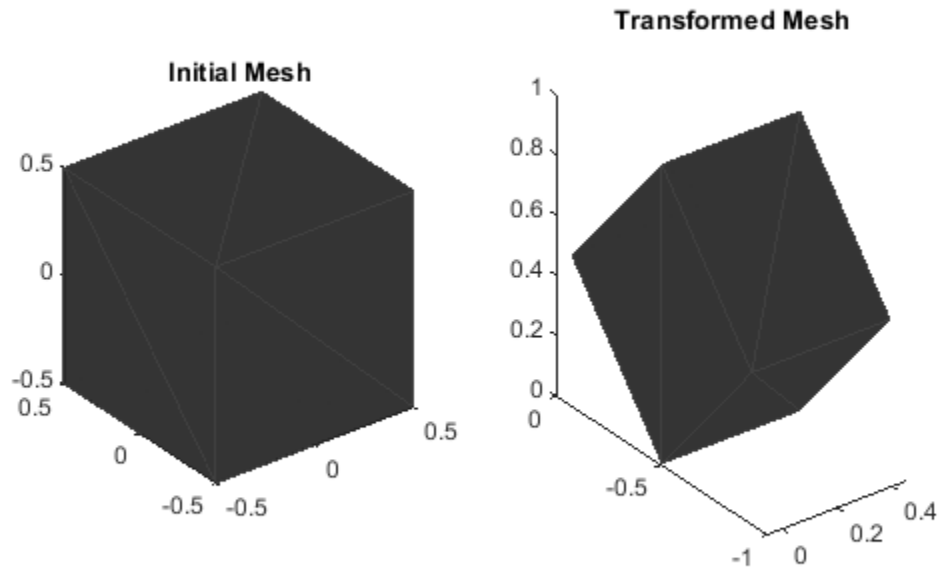
```
T = makehgtform('translate',[0.2 -0.5 0.5], 'scale',[0.5 0.6 0.7], 'xrotate',pi/4);
```

Transform the mesh.

```
transformedCuboid = applyTransform(cuboid,T);
```

Visualize the mesh.

```
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:

$$\begin{bmatrix} x_T \\ y_T \\ z_T \\ 1 \end{bmatrix} = T \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

x_T , y_T , and z_T 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.

See Also

Objects

extendedObjectMesh

Functions

join | rotate | scale | scaleToFit | show | translate

Introduced in R2020b

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

This example shows how to create `extendedObjectMesh` objects and join them together.

Construct two meshes of unit dimensions.

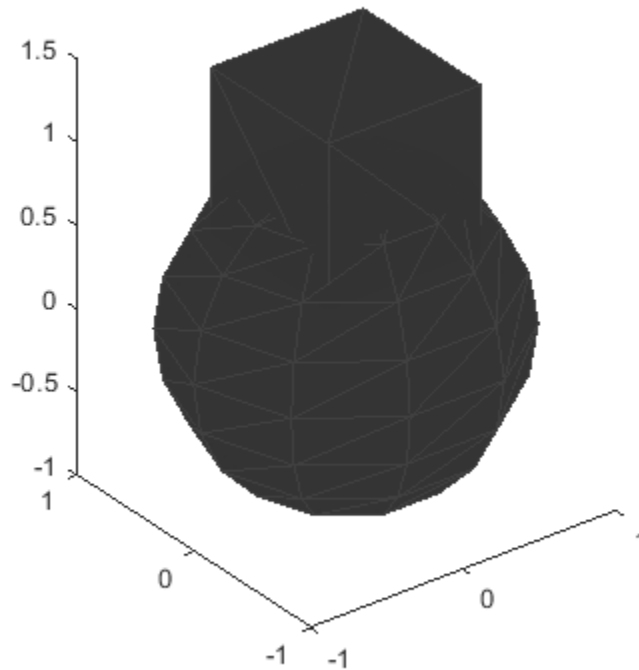
```
sph = extendedObjectMesh('sphere');  
cub = extendedObjectMesh('cuboid');
```

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.

See Also

Objects
`extendedObjectMesh`

Functions

applyTransform | rotate | scale | scaleToFit | show | translate

Introduced in R2020b

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

This example shows how to create an `extendedObjectMesh` object and rotate the object.

Construct a cuboid mesh.

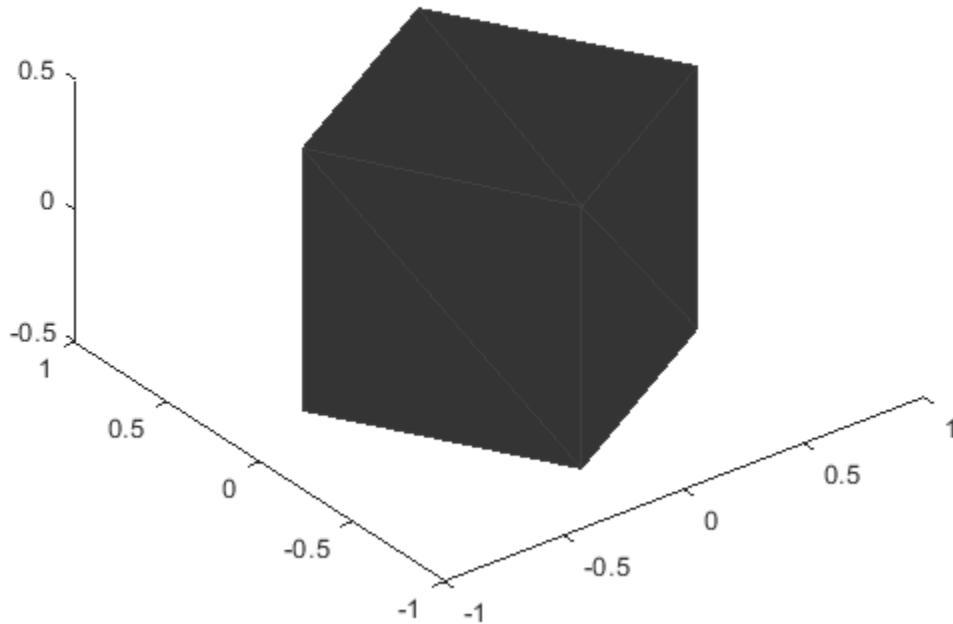
```
mesh = extendedObjectMesh('cuboid');
```

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.

See Also

Objects

extendedObjectMesh

Functions

applyTransform | join | scale | scaleToFit | show | translate

Introduced in R2020a

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

This example shows how to create an `extendedObjectMesh` object and scale the object.

Construct a cuboid mesh of unit dimensions.

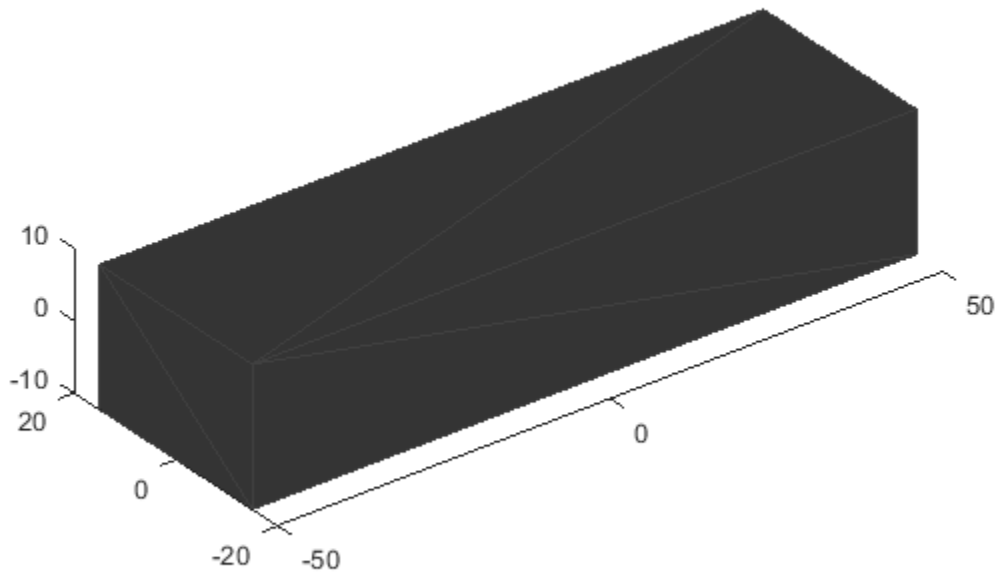
```
cuboid = extendedObjectMesh('cuboid');
```

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 single positive real value 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.

See Also

Objects

`extendedObjectMesh`

Functions

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

Introduced in R2020b

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

This example shows how to create an `extendedObjectMesh` object and auto-scale the object to the required dimensions.

Construct a sphere mesh of unit dimensions.

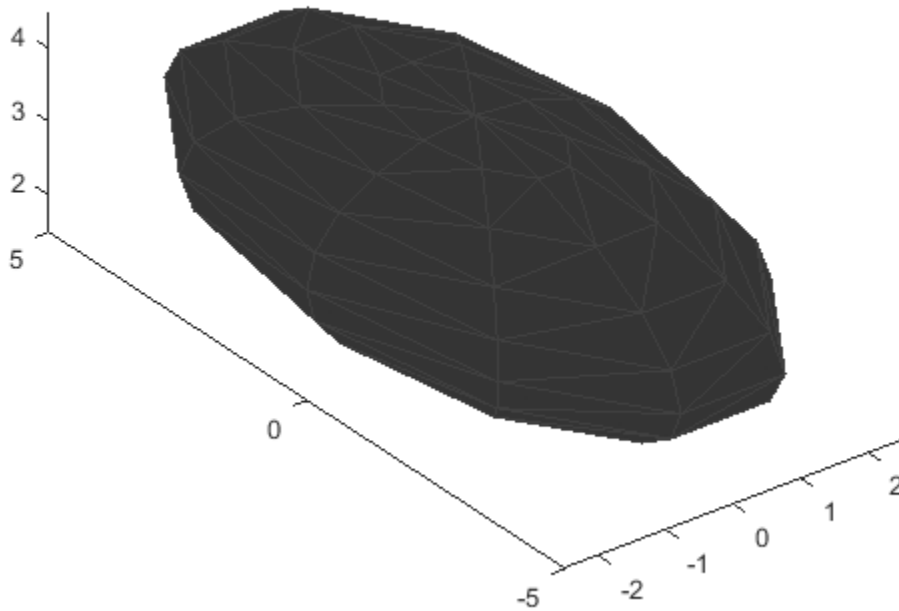
```
sph = extendedObjectMesh('sphere');
```

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`

dims — Cuboid dimensions

`struct`

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.

See Also

Objects

extendedObjectMesh

Functions

applyTransform | join | rotate | scale | show | translate

Introduced in R2020b

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

This example shows how to create an `extendedObjectMesh` object and translate the object.

Construct a cuboid mesh.

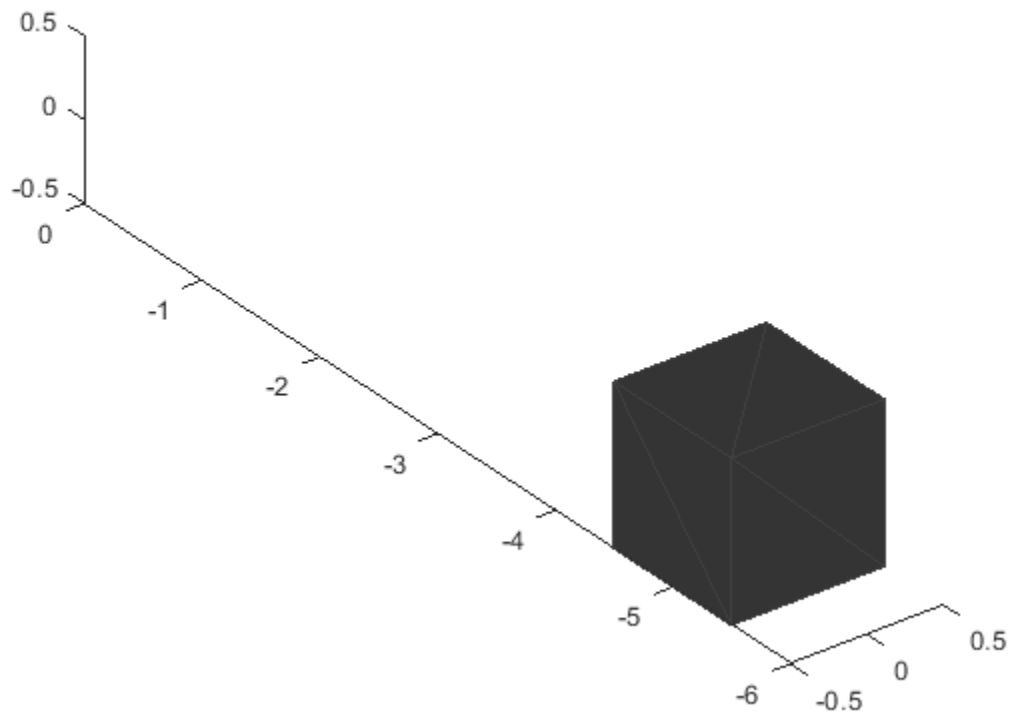
```
mesh = extendedObjectMesh('cuboid');
```

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`

Current axes, specified as an `axes` object.

See Also

Objects

`extendedObjectMesh`

Functions

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

Introduced in R2020b

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

This example shows how to create an `extendedObjectMesh` object and translate the object.

Construct a cuboid mesh.

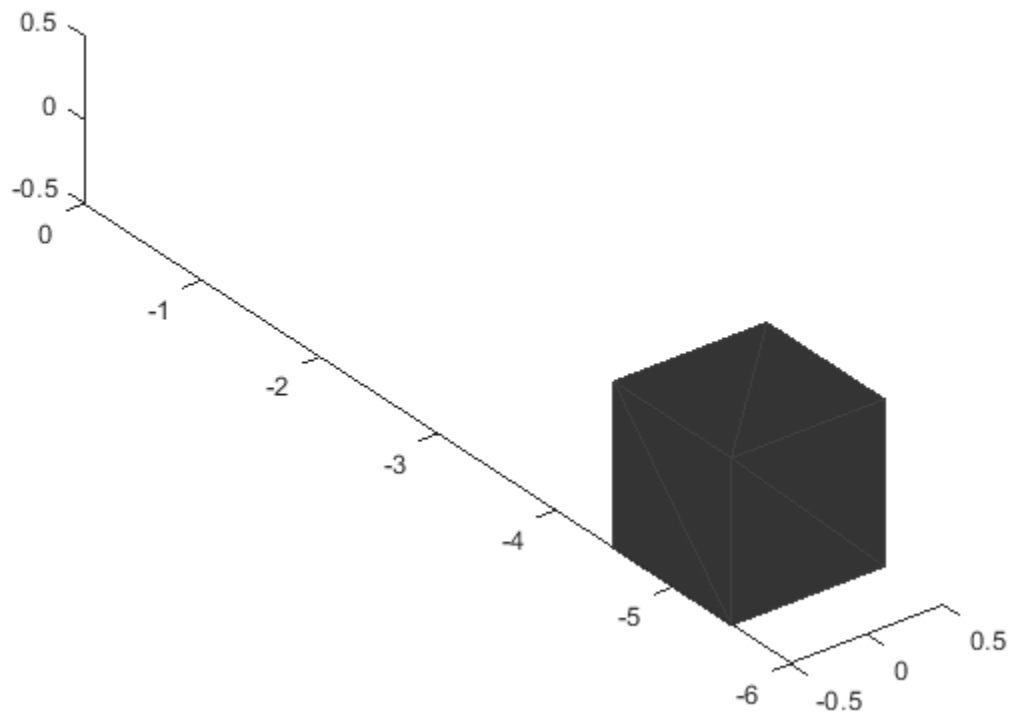
```
mesh = extendedObjectMesh('cuboid');
```

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.

See Also

Objects

extendedObjectMesh

Functions

applyTransform | join | rotate | scale | scaleToFit | show

Introduced in R2020b

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)
```

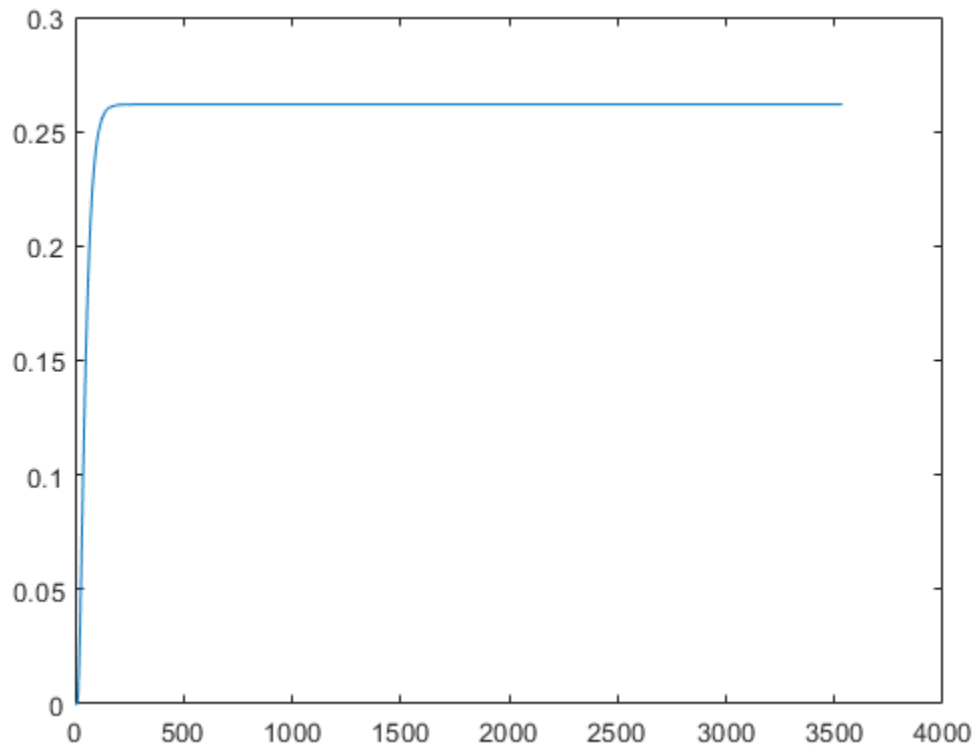
```
ans = 1×2
```

13

3536

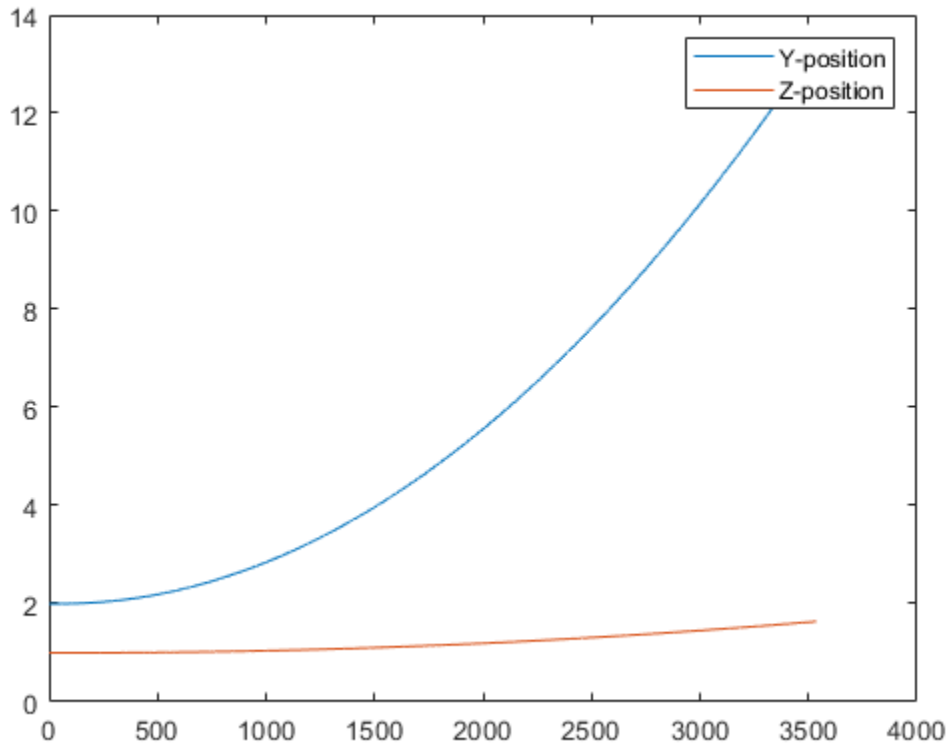
Plot the change in roll angle based on the simulation output. The roll angle (the X Euler angle) is the 9th row of the `simOut.y` output.

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



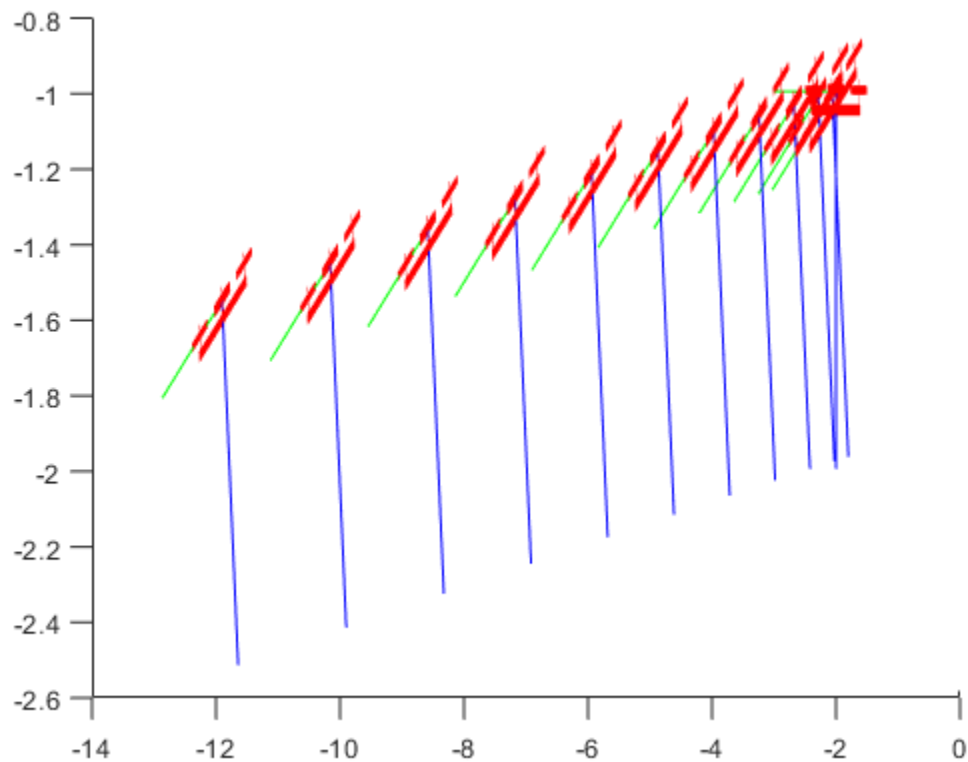
Plot the change in the Y and Z positions. With the specified thrust and roll angle, the multirotor should fly over and lose some altitude. A positive value for Z is expected as positive Z is down.

```
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.

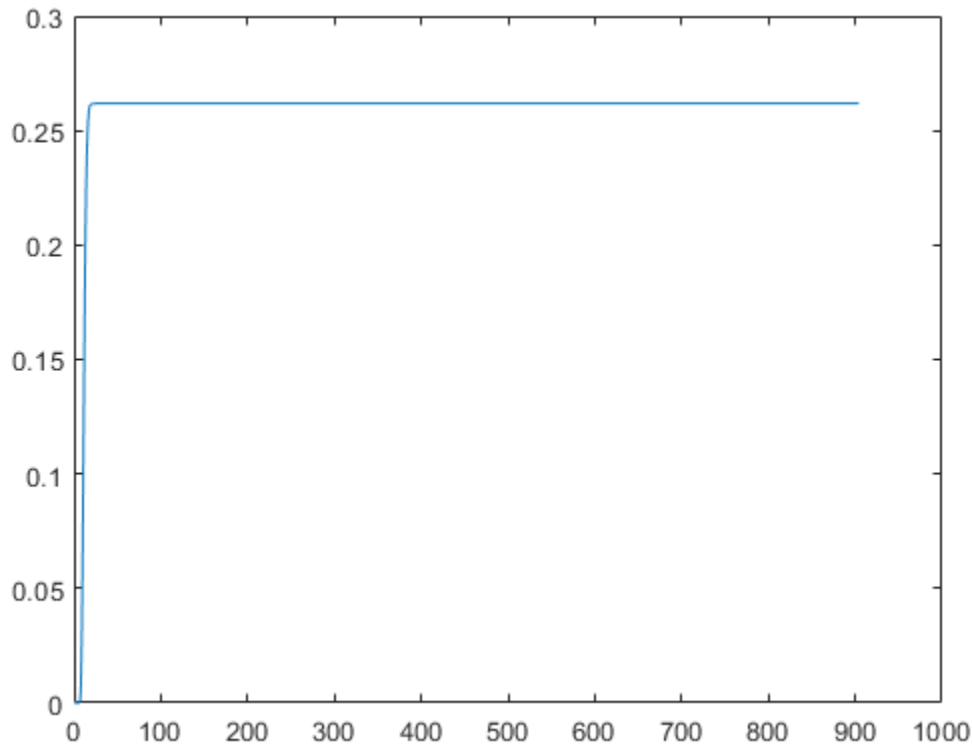
```
simOut = ode45(@(~,x)derivative(model,x,u,e), [0 50], s);
size(simOut.y)
```

```
ans = 1x2
```

```
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.

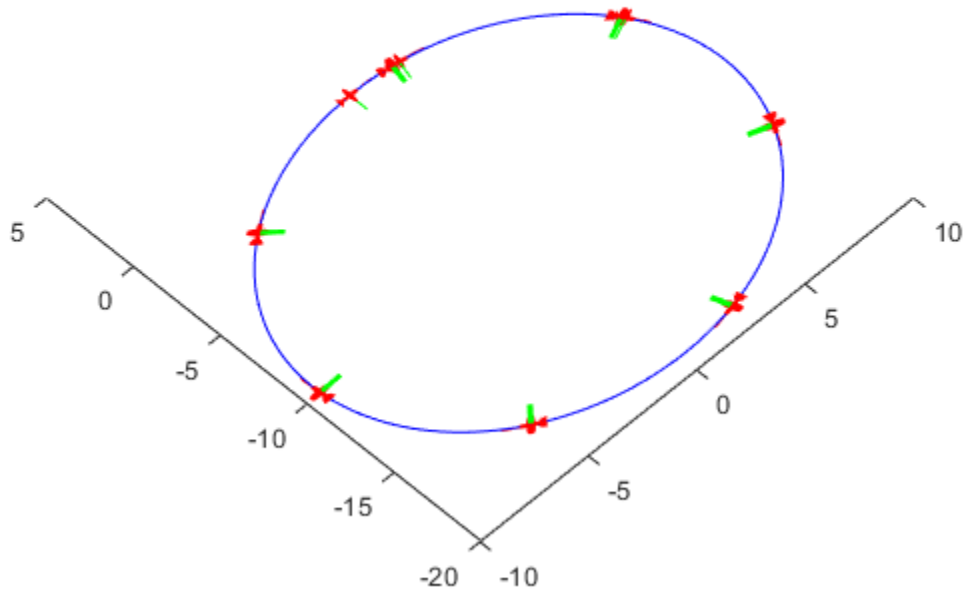
```
downsample = 1:30:size(simOut.y,2);
translations = simOut.y(1:3,downsample)'; % xyz-position
rotations = eul2quat([simOut.y(5,downsample)',simOut.y(6,downsample)',simOut.y(7,downsample)']');
```



```

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.

Extended Capabilities

C/C++ Code Generation

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

See Also

Functions

`derivative` | `environment` | `ode45` | `plotTransforms` | `state`

Objects

`fixedwing` | `multirotor`

Blocks

UAV Guidance Model | Waypoint Follower

Topics

“Approximate High-Fidelity UAV model with UAV Guidance Model block”

“Tuning Waypoint Follower for Fixed-Wing UAV”

Introduced in R2018b

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);
```

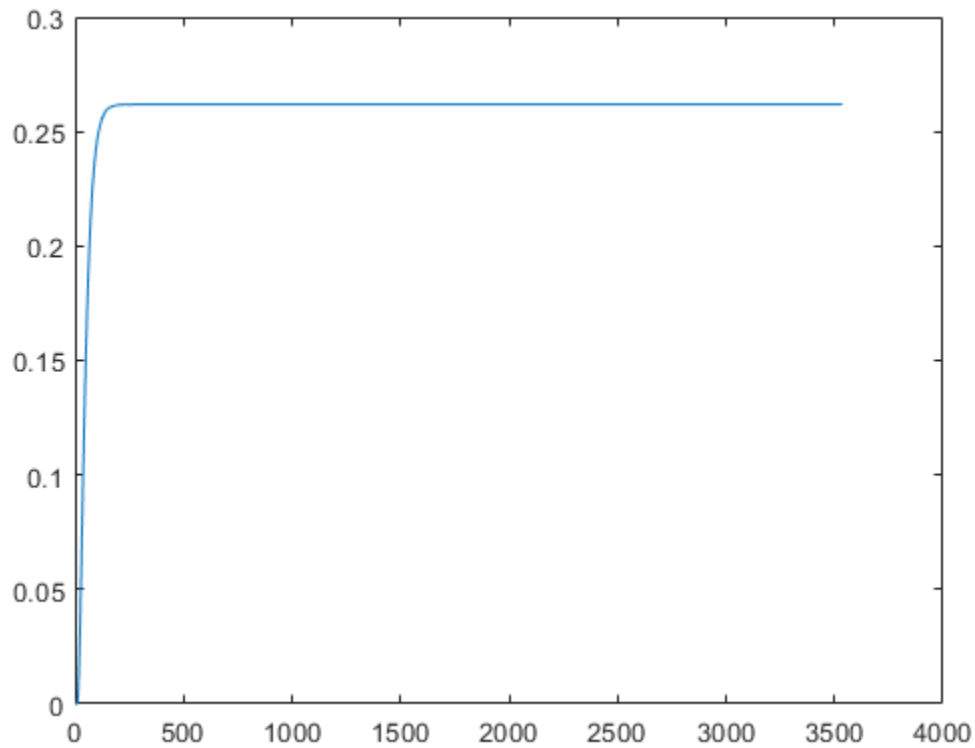
Simulate the UAV state using `ode45` integration. The `y` field outputs the multirotor UAV states as a 13-by- n matrix.

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

```
ans = 1×2
```

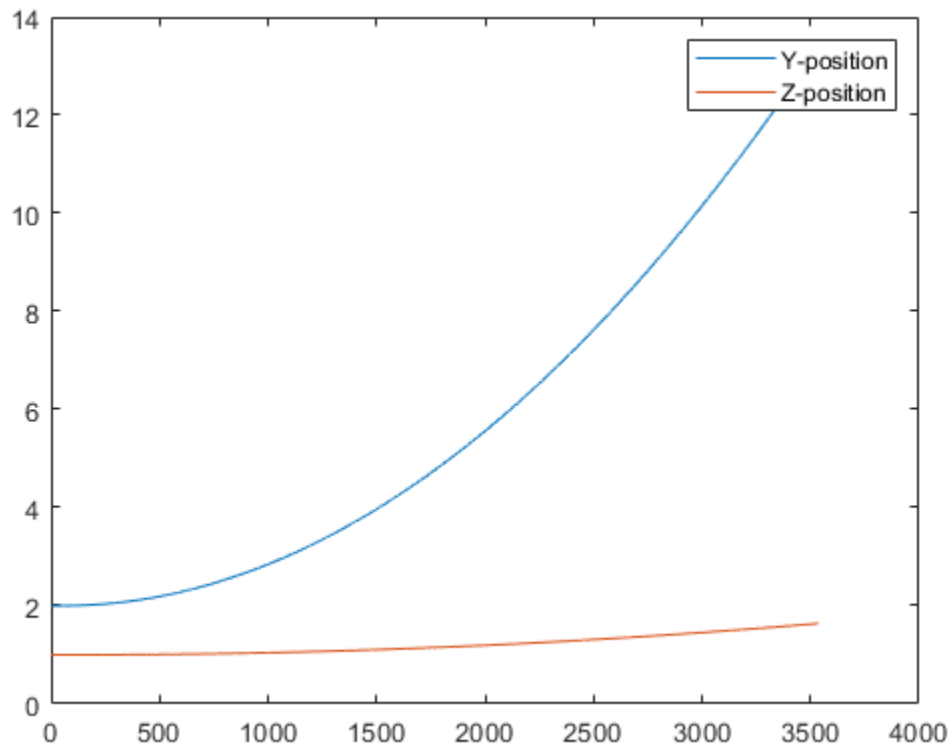
Plot the change in roll angle based on the simulation output. The roll angle (the X Euler angle) is the 9th row of the `simOut.y` output.

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



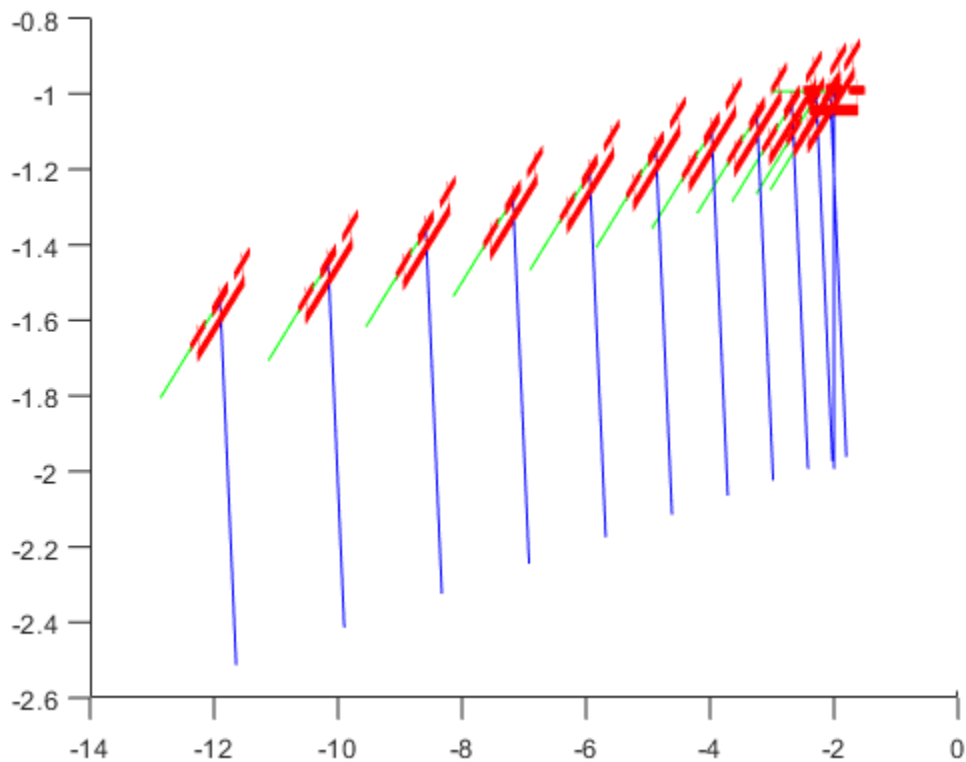
Plot the change in the Y and Z positions. With the specified thrust and roll angle, the multirotor should fly over and lose some altitude. A positive value for Z is expected as positive Z is down.

```
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.

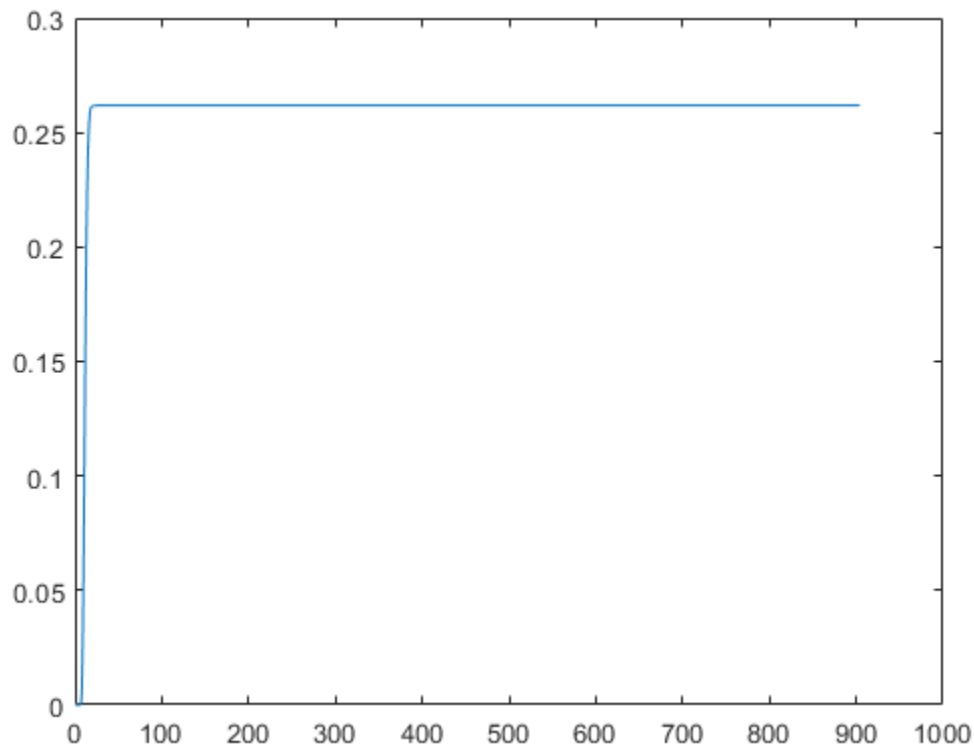
```
simOut = ode45(@(~,x)derivative(model,x,u,e), [0 50], s);
size(simOut.y)
```

```
ans = 1x2
```

```
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,:))
```



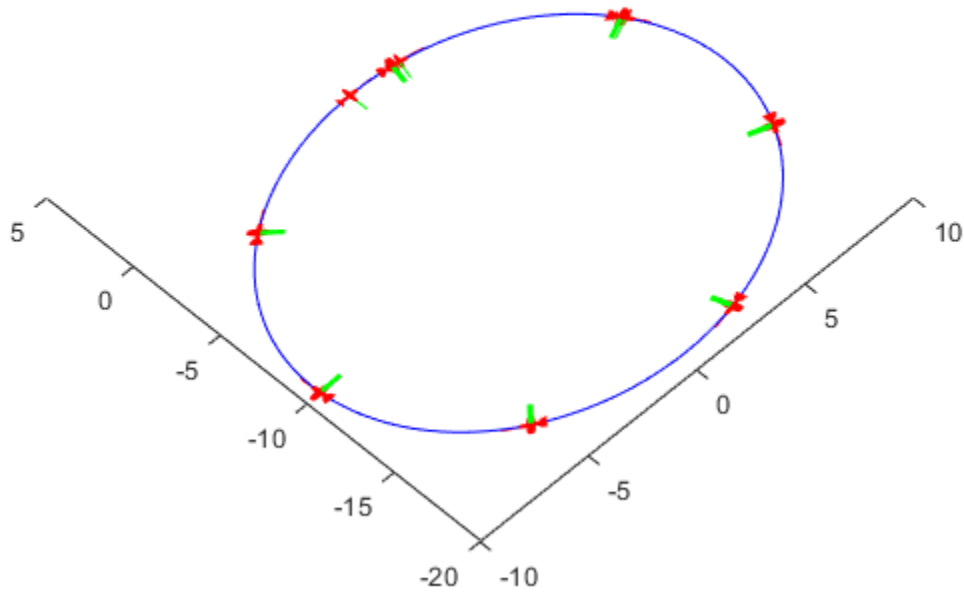
You can also plot the fixed-wing trajectory using `plotTransforms`. Create the translation and rotation vectors from the simulated state. Downsample (every 30th element) and transpose the `simOut` elements, and convert the Euler angles to quaternions. Specify the mesh as the `fixedwing.stl` file and 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.

Extended Capabilities

C/C++ Code Generation

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

See Also

Functions

[control](#) | [derivative](#) | [environment](#) | [ode45](#) | [plotTransforms](#) | [state](#)

Objects

[fixedwing](#) | [multirotor](#)

Blocks

[UAV Guidance Model](#) | [Waypoint Follower](#)

Topics

“Approximate High-Fidelity UAV model with UAV Guidance Model block”

“Tuning Waypoint Follower for Fixed-Wing UAV”

Introduced in R2018b

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);
```

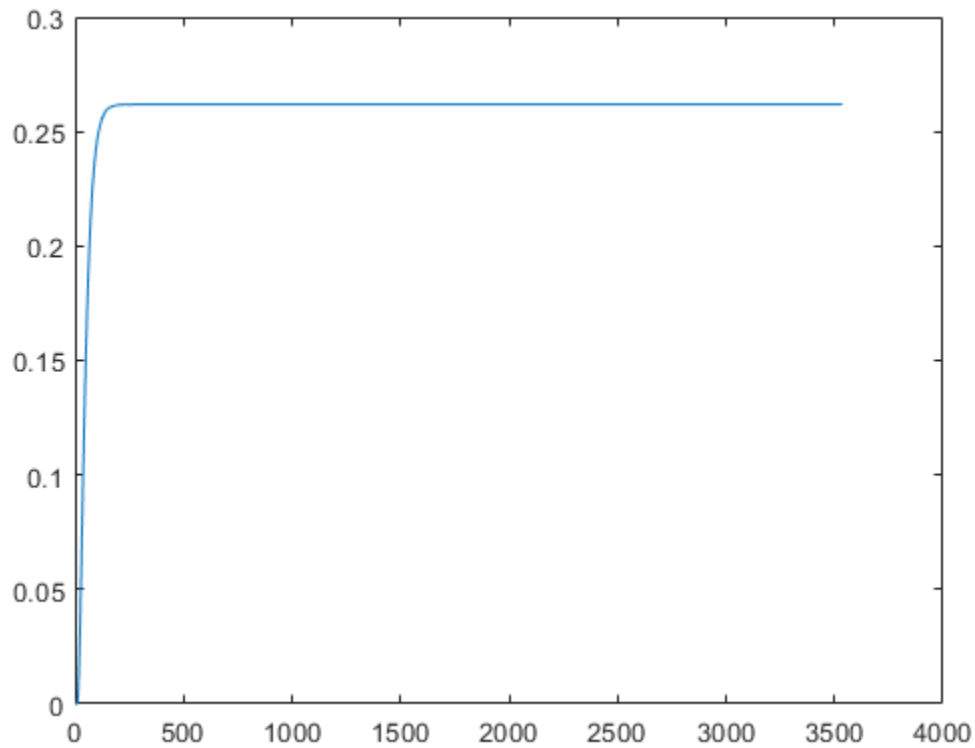
Simulate the UAV state using `ode45` integration. The `y` field outputs the multirotor UAV states as a 13-by- n matrix.

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

```
ans = 1×2
```

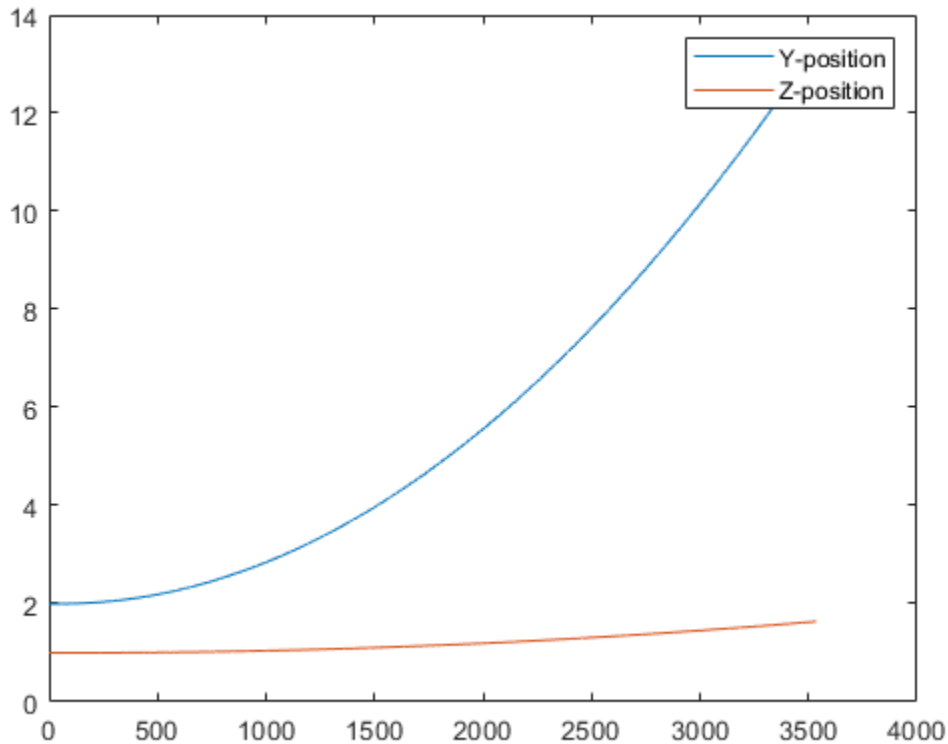
Plot the change in roll angle based on the simulation output. The roll angle (the X Euler angle) is the 9th row of the `simOut.y` output.

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



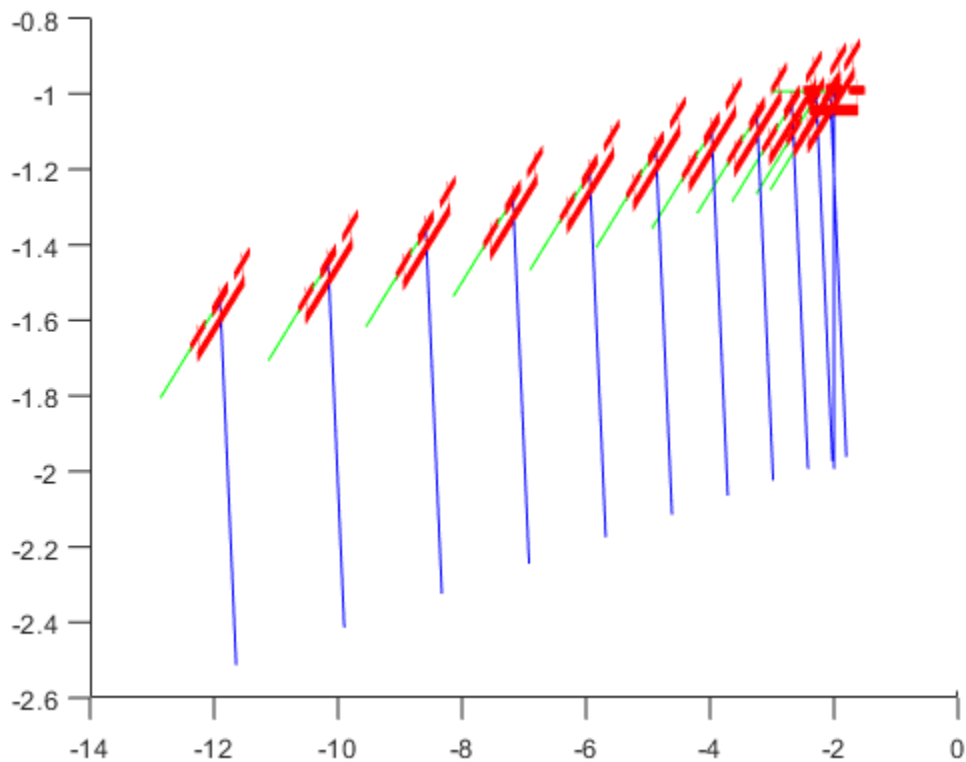
Plot the change in the Y and Z positions. With the specified thrust and roll angle, the multirotor should fly over and lose some altitude. A positive value for Z is expected as positive Z is down.

```
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.

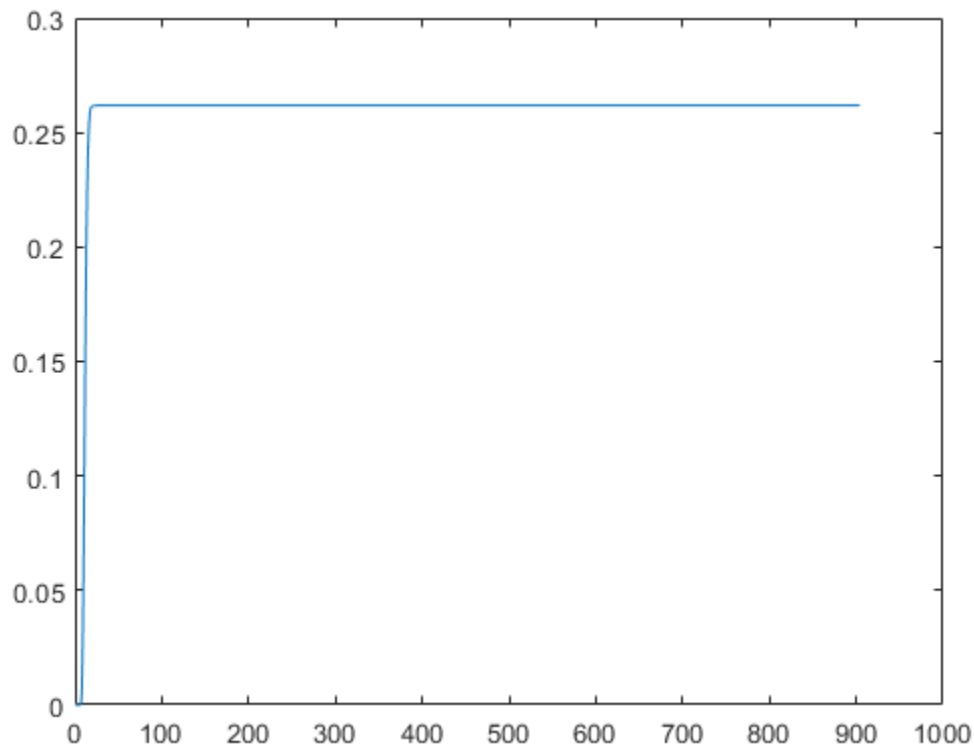
```
simOut = ode45(@(~,x)derivative(model,x,u,e), [0 50], s);
size(simOut.y)
```

```
ans = 1x2
```

```
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.

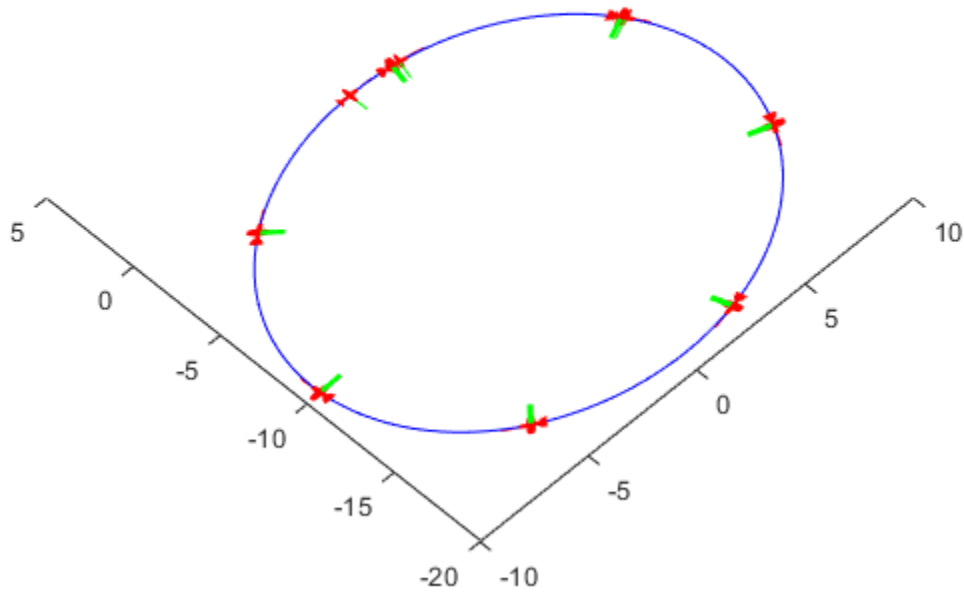
```
plot(simOut.y(7,:))
```



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.

Extended Capabilities

C/C++ Code Generation

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

See Also

Functions

`control` | `derivative` | `ode45` | `plotTransforms` | `state`

Objects

`fixedwing` | `multirotor`

Blocks

UAV Guidance Model | Waypoint Follower

Topics

“Approximate High-Fidelity UAV model with UAV Guidance Model block”
“Tuning Waypoint Follower for Fixed-Wing UAV”

Introduced in R2018b

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)
```

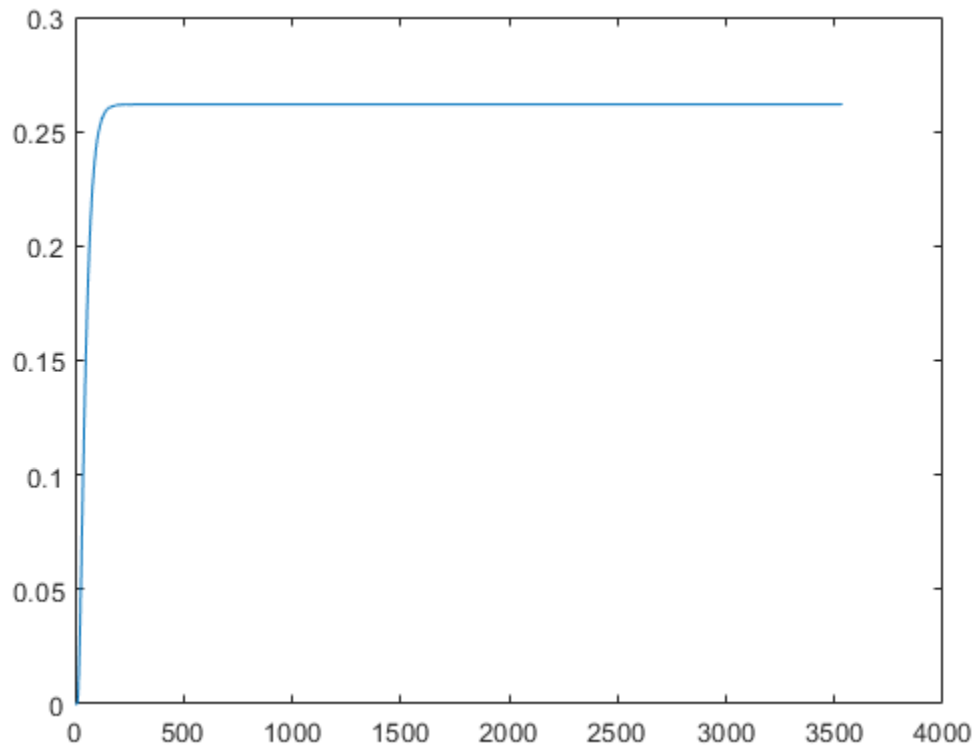
```
ans = 1×2
```

13

3536

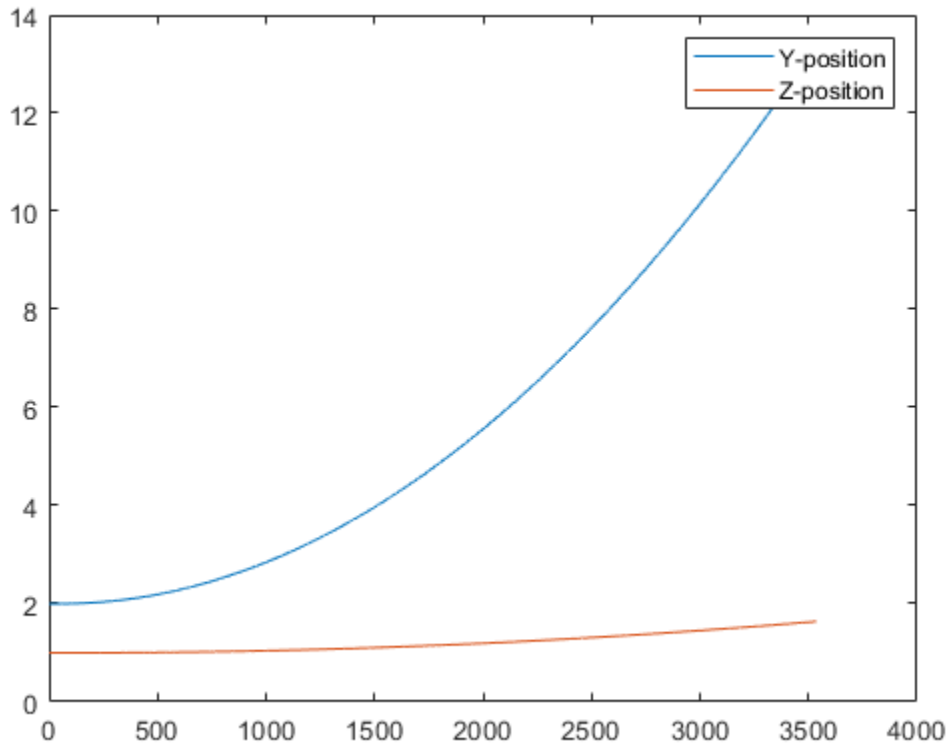
Plot the change in roll angle based on the simulation output. The roll angle (the X Euler angle) is the 9th row of the `simOut.y` output.

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



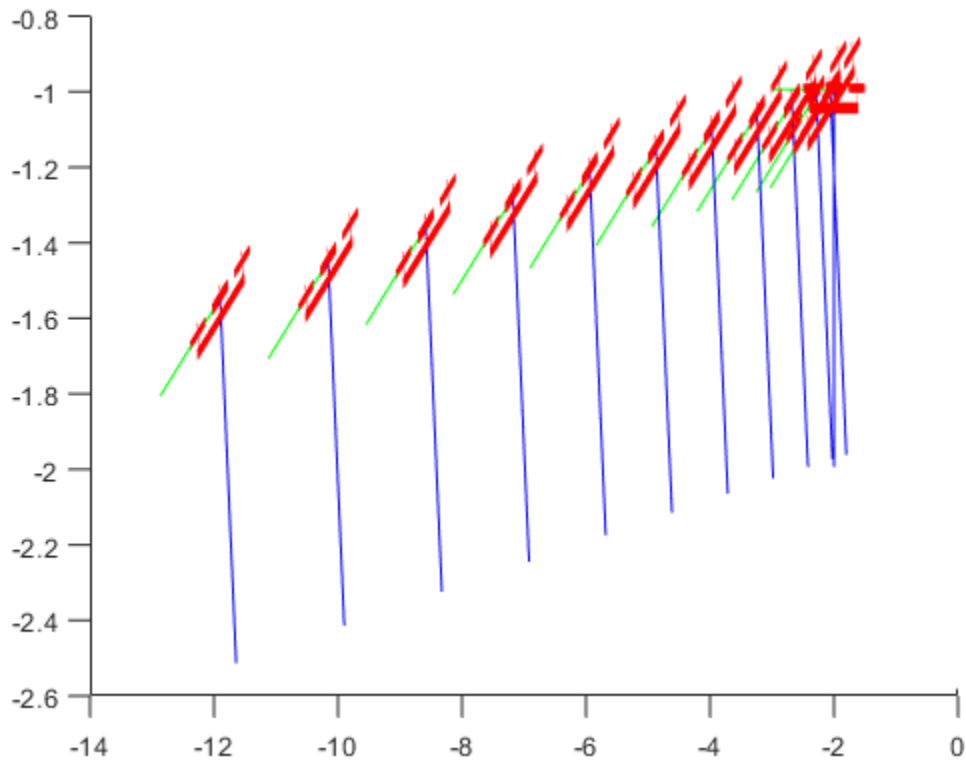
Plot the change in the Y and Z positions. With the specified thrust and roll angle, the multirotor should fly over and lose some altitude. A positive value for Z is expected as positive Z is down.

```
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.

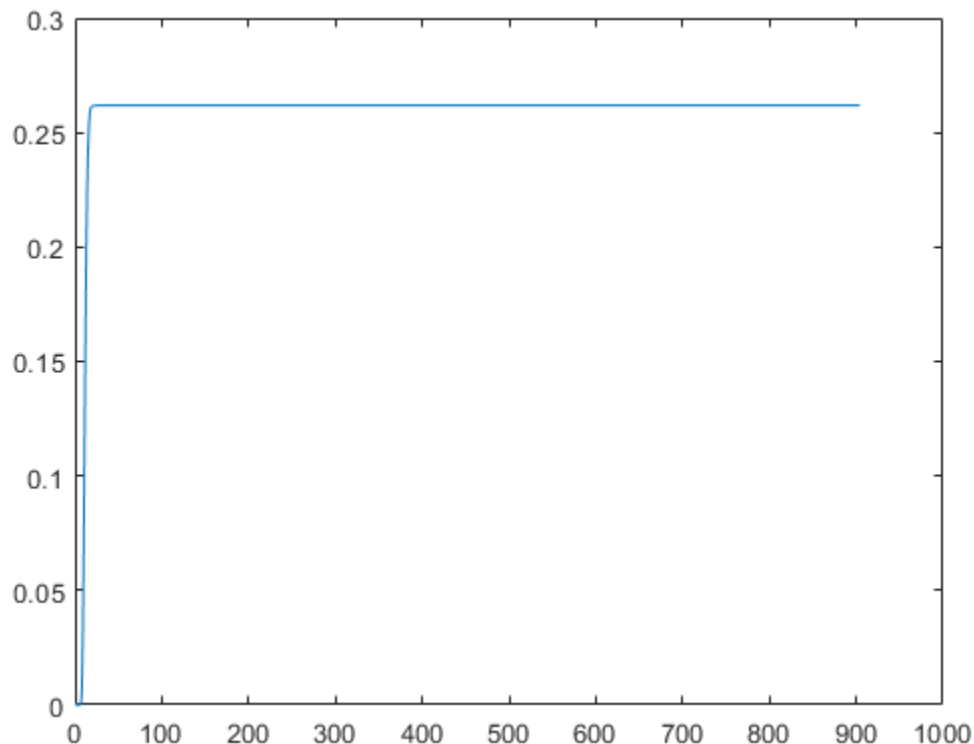
```
simOut = ode45(@(~,x)derivative(model,x,u,e), [0 50], s);
size(simOut.y)
```

```
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,:))
```



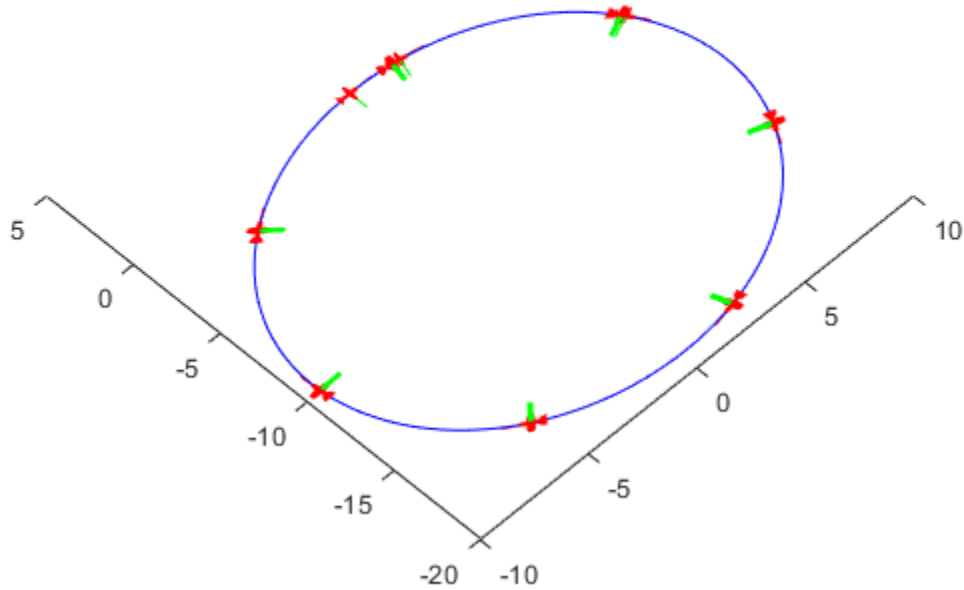
You can also plot the fixed-wing trajectory using `plotTransforms`. Create the translation and rotation vectors from the simulated state. Downsample (every 30th element) and transpose the `simOut` elements, and convert the Euler angles to quaternions. Specify the mesh as the `fixedwing.stl` file and 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** - $[v_x \ v_y \ v_z]$ 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.

Extended Capabilities

C/C++ Code Generation

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

See Also

Functions

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

Objects

`fixedwing` | `multirotor`

Blocks

UAV Guidance Model | Waypoint Follower

Topics

“Approximate High-Fidelity UAV model with UAV Guidance Model block”

“Tuning Waypoint Follower for Fixed-Wing UAV”

Introduced in R2018b

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
Unable to extract barometer value from log data
-----
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=1x18 struct array with fields:
  SignalName
  Result

errorIndex = 1x6

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

Example: '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.

See Also

Objects

`flightLogSignalMapping` | `mavlinktlog` | `ulogreader`

Functions

`copy` | `extract` | `info` | `mapSignal` | `show` | `updatePlot`

Introduced in R2021a

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.

See Also

Objects

`flightLogSignalMapping`

Functions

`checkSignal` | `extract` | `info` | `mapSignal` | `show` | `updatePlot`

Introduced in R2021a

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.

See Also

`flightLogSignalMapping` | `info` | `mapSignal` | `mavlinktlog` | `show` | `updatePlot`

Introduced in R2020b

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 2-59 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 2-60 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

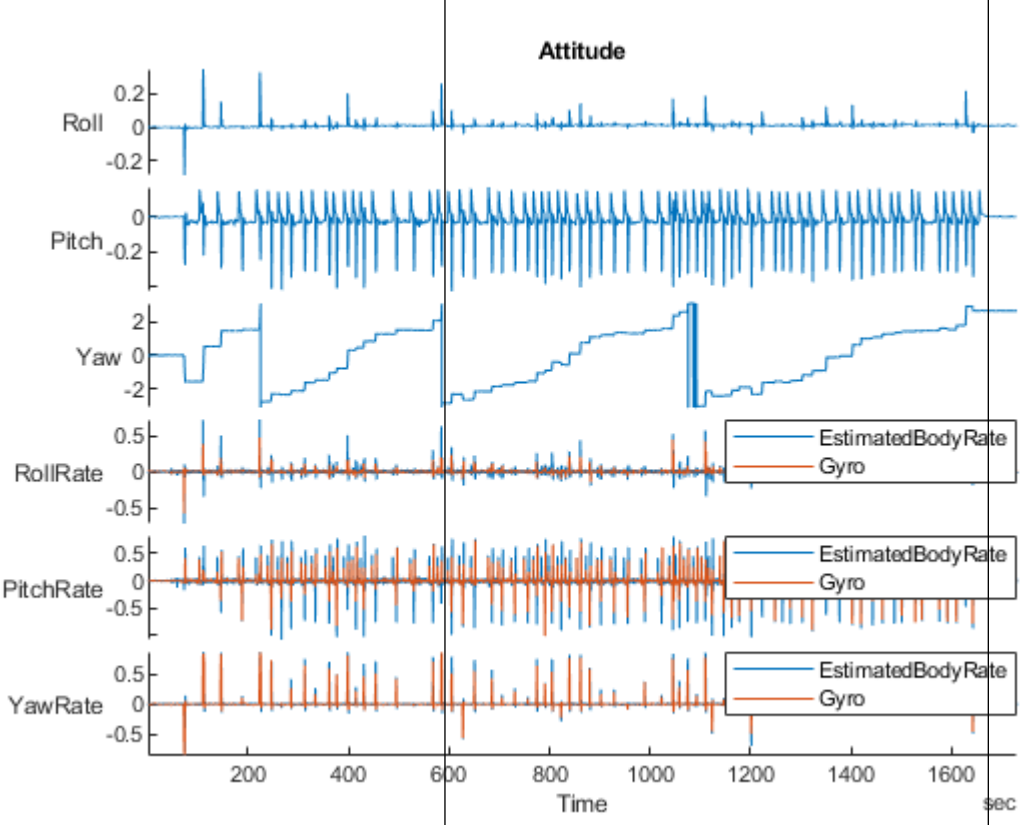
Signal Name	Description	Fields	Unit
Accel#	Raw magnetometer reading from IMU sensor	[ax ay az]	m/s ²
Airspeed#	Airspeed reading of pressure differential, indicated air speed, and temperature	[PressDiff, AirSpeed, Temp]	Pa, m/s, °C
AttitudeEuler	Attitude of UAV in Euler (ZYX) form	[Roll, Pitch, Yaw]	radi
AttitudeRate	Angular velocity along each body axis	[xRotRate, yRotRate, zRotRate]	rad/s
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, m, °C
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, m/s, deg
Gyro#	Raw body angular velocity readings from IMU sensor	[GyroX, GyroY, GyroZ]	rad/s
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

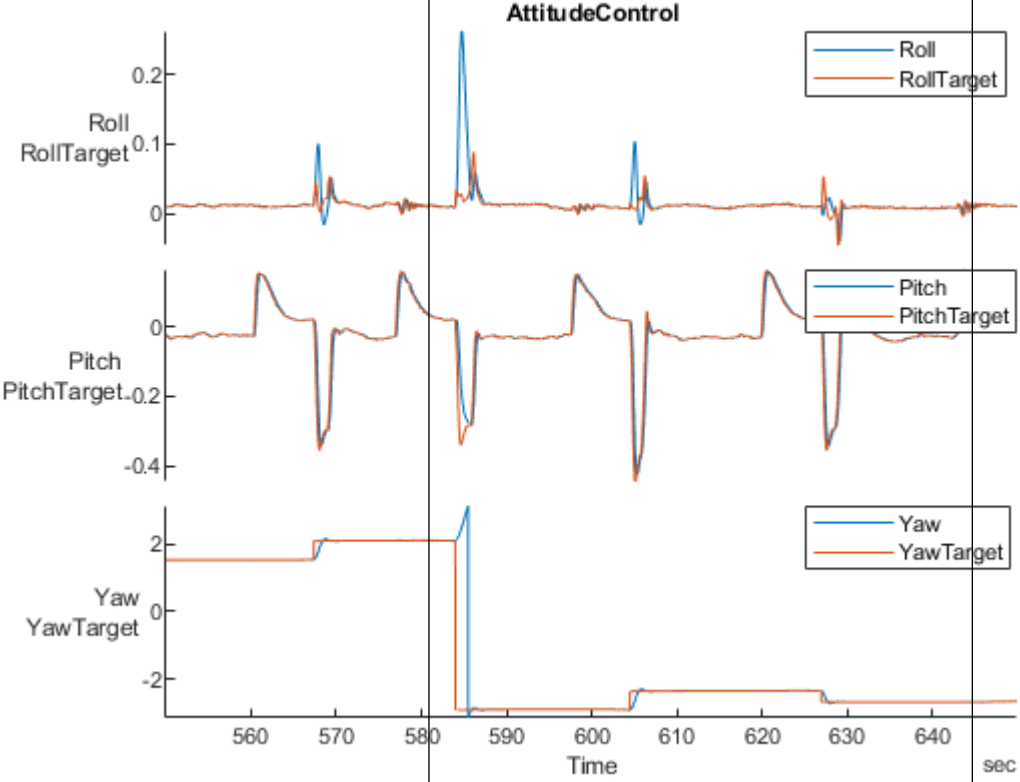
Predefined Plots

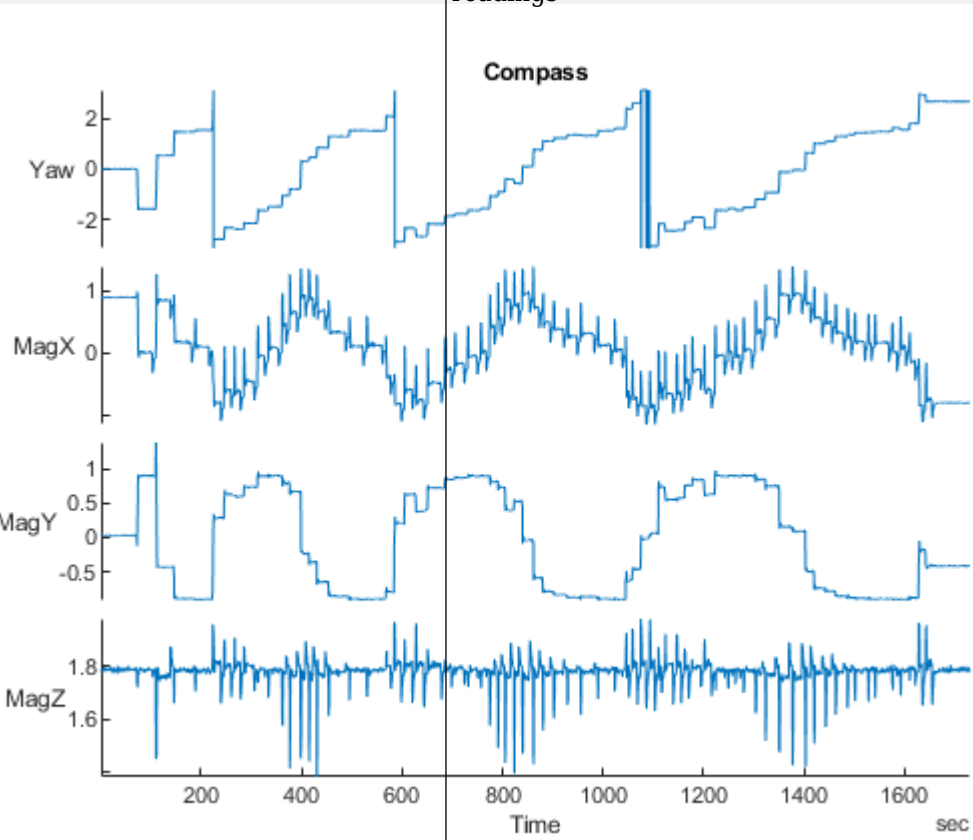
After mapping signals to the list of predefined signals using `mapSignal`, specific plots are made available when calling `show`. To view a list of available plots and their associated signals for your specific object, call `info(mapper, "Plot")`. If you want to define custom plots based on signals,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

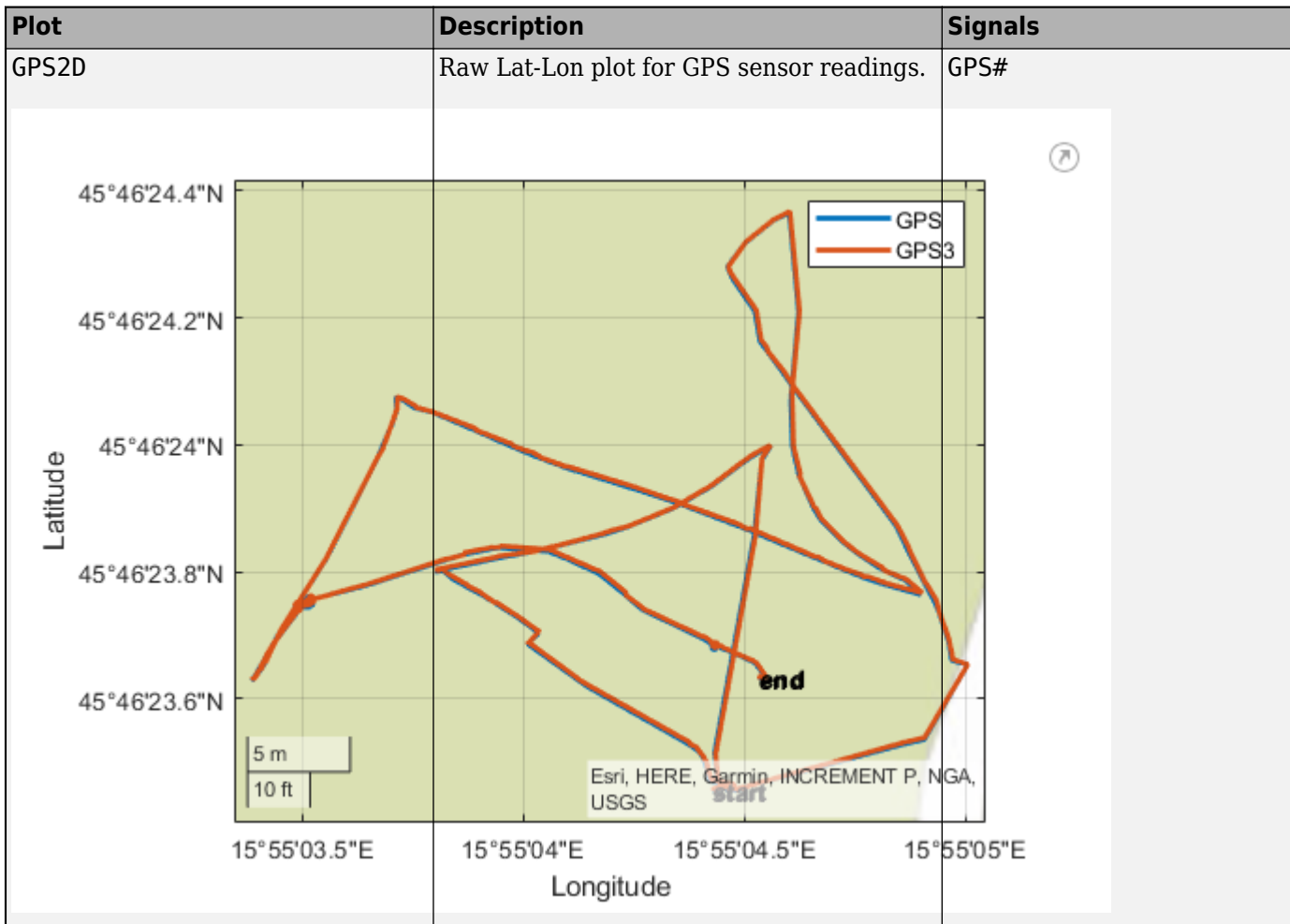
Each predefined plot has a set of required signals that must be mapped.

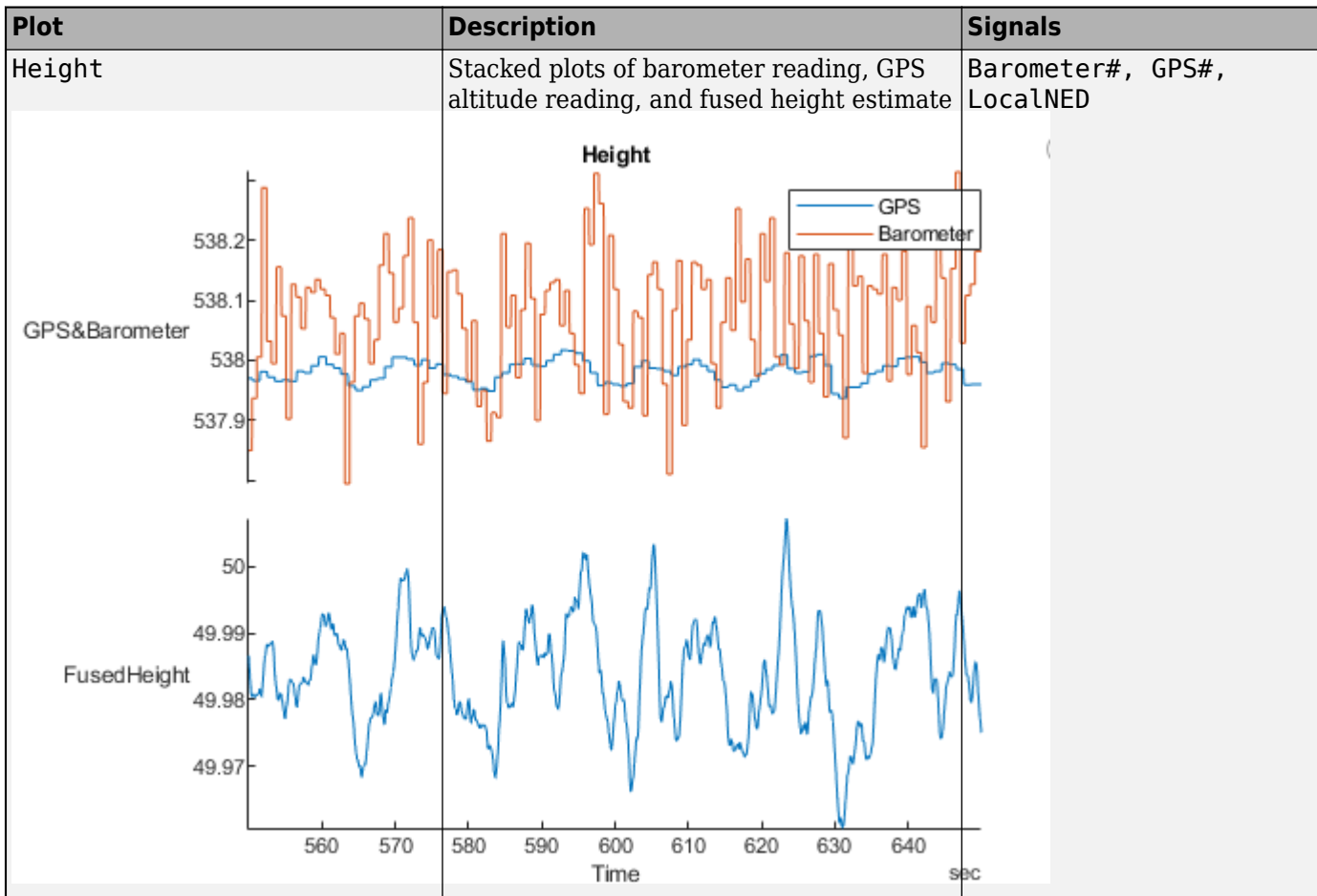
Predefined Plots

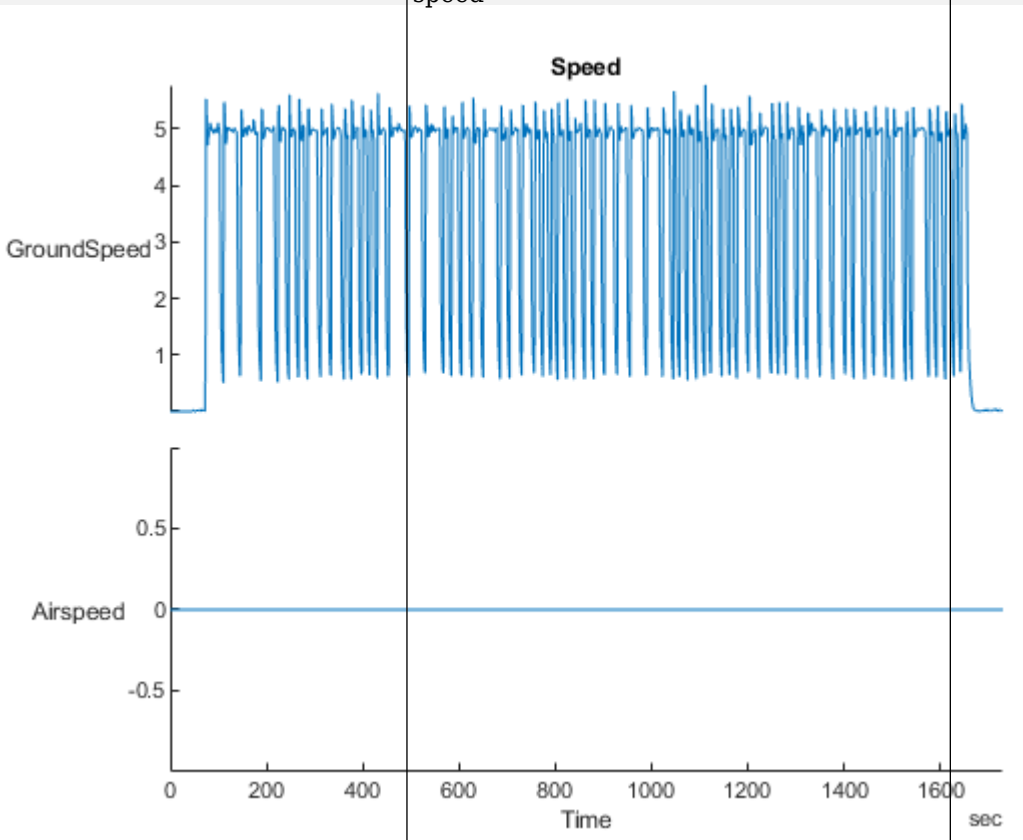
Plot	Description	Signals
<p data-bbox="240 346 375 380">Attitude</p> 	<p data-bbox="691 346 1232 409">Stacked plot of roll, pitch, yaw angles and body rotation rates</p>	<p data-bbox="1232 346 1604 409">AttitudeEuler, AttitudeRate, Gyro#</p>

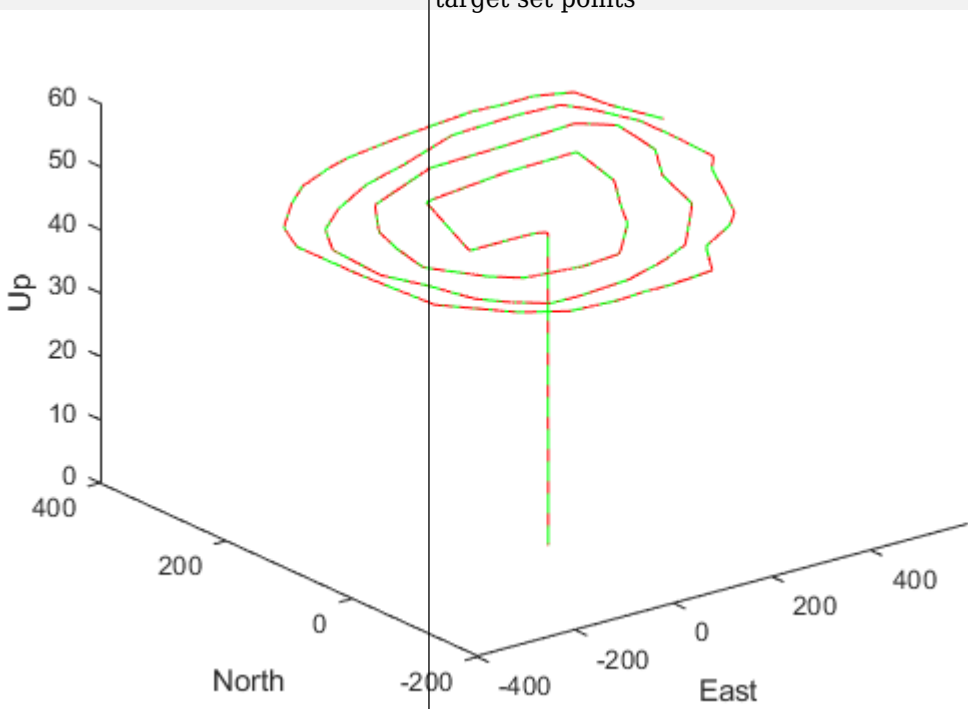
Plot	Description	Signals
<p data-bbox="240 296 483 327">AttitudeControl</p>  <p data-bbox="771 409 950 436">AttitudeControl</p> <p data-bbox="284 514 349 546">Roll</p> <p data-bbox="284 546 381 577">RollTarget</p> <p data-bbox="284 745 349 777">Pitch</p> <p data-bbox="284 777 381 808">PitchTarget</p> <p data-bbox="284 987 349 1018">Yaw</p> <p data-bbox="284 1018 381 1050">YawTarget</p> <p data-bbox="828 1144 885 1176">Time</p> <p data-bbox="1242 1144 1282 1176">sec</p>	<p data-bbox="691 296 1230 367">Estimated attitude of UAV and the attitude target set point</p>	<p data-bbox="1230 296 1602 367">AttitudeEuler, AttitudeTargetEuler</p>
<p data-bbox="240 1247 357 1278">Battery</p>	<p data-bbox="691 1247 1006 1278">Battery consumption plot</p>	<p data-bbox="1230 1247 1347 1278">Battery</p>

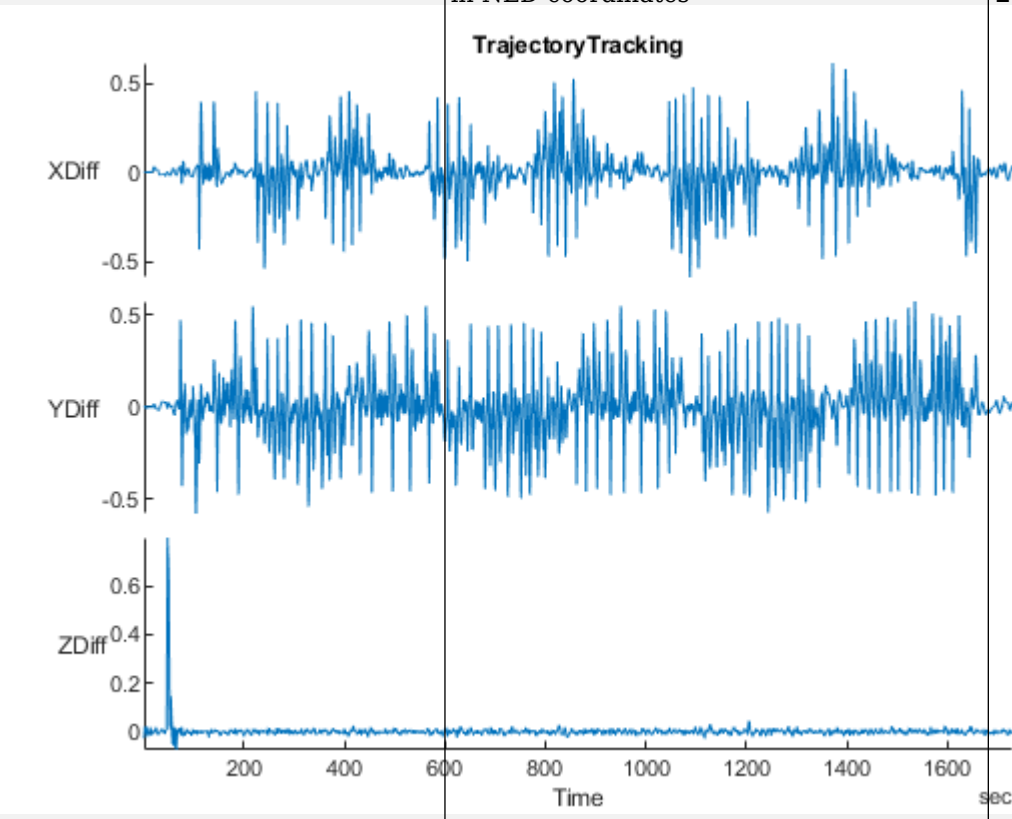
Plot	Description	Signals
<p data-bbox="240 298 354 331">Compass</p> 	<p data-bbox="691 298 1117 361">Estimated yaw and magnetometer readings</p>	<p data-bbox="1230 298 1559 361">AttitudeEuler, Mag#, GPS#</p>

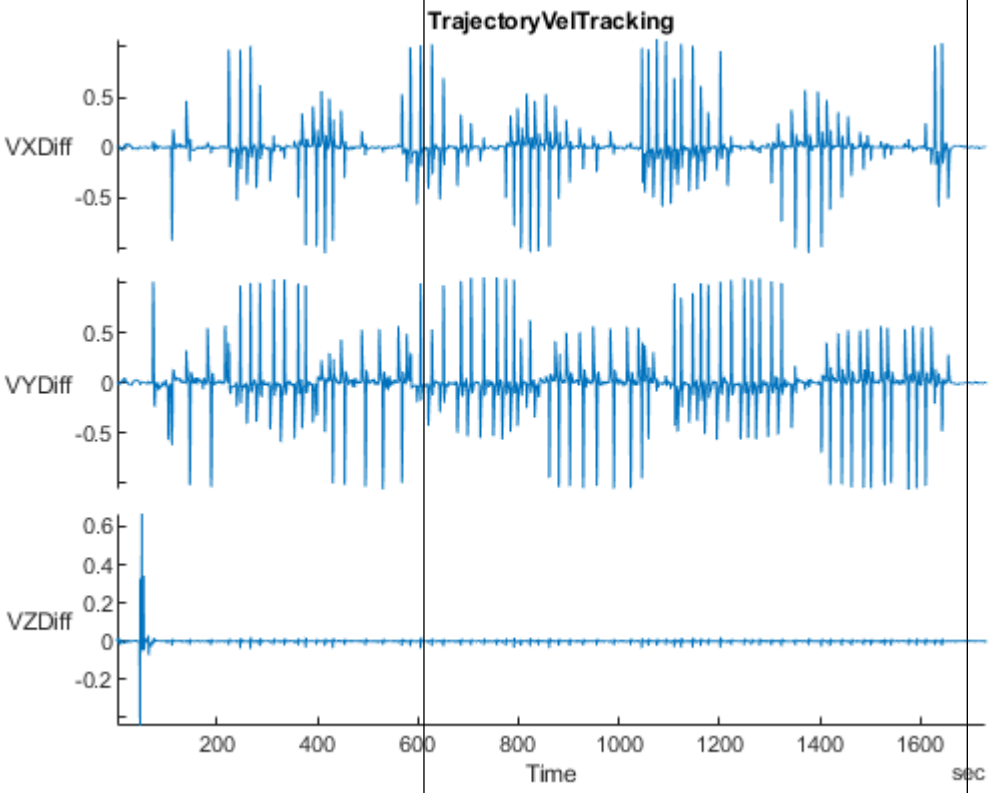




Plot	Description	Signals
<p data-bbox="240 296 324 327">Speed</p> 	<p data-bbox="691 296 1177 359">Stacked plot of ground velocity and air speed</p>	<p data-bbox="1230 296 1485 327">GPS#, Airspeed#</p>

Plot	Description	Signals
<p>Trajectory</p> 	<p>Trajectory in local coordinates versus target set points</p>	<p>LocalNED, LocalNEDTarget</p>

Plot	Description	Signals
<p>TrajectoryTracking</p>  <p>The plot displays three error signals over a 1600-second period. The top two signals, XDiff and YDiff, show high-frequency oscillations between approximately -0.5 and 0.5. The bottom signal, ZDiff, shows a sharp initial spike to about 0.7, followed by a rapid decay to near zero by 100 seconds, remaining stable thereafter.</p>	<p>Error between desired and actual position in NED coordinates</p>	<p>LocalNED, LocalNEDTarget</p>

Plot	Description	Signals
<p>TrajectoryVelTracking</p>  <p>The plot displays three stacked time-series signals for velocity error in NED coordinates. The x-axis is labeled 'Time' and ranges from 0 to 1600 seconds with major ticks every 200 seconds. The top signal, 'VXDiff', has a y-axis from -0.5 to 0.5. The middle signal, 'VYDiff', also has a y-axis from -0.5 to 0.5. The bottom signal, 'VZDiff', has a y-axis from -0.2 to 0.6. All three signals show high-frequency oscillations around zero, with a notable spike at approximately 600 seconds. The 'VZDiff' signal is significantly smaller in magnitude than the 'VXDiff' and 'VYDiff' signals.</p>	<p>Error between desired and actual velocity in NED coordinates</p>	<p>LocalNEDVel, LocalNEDVelTarget</p>

See Also

Objects

flightLogSignalMapping

Functions

checkSignal | copy | extract | mapSignal | show | updatePlot

Introduced in R2020b

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 2-71 and [Predefined Plots](#) on page 2-72.

`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

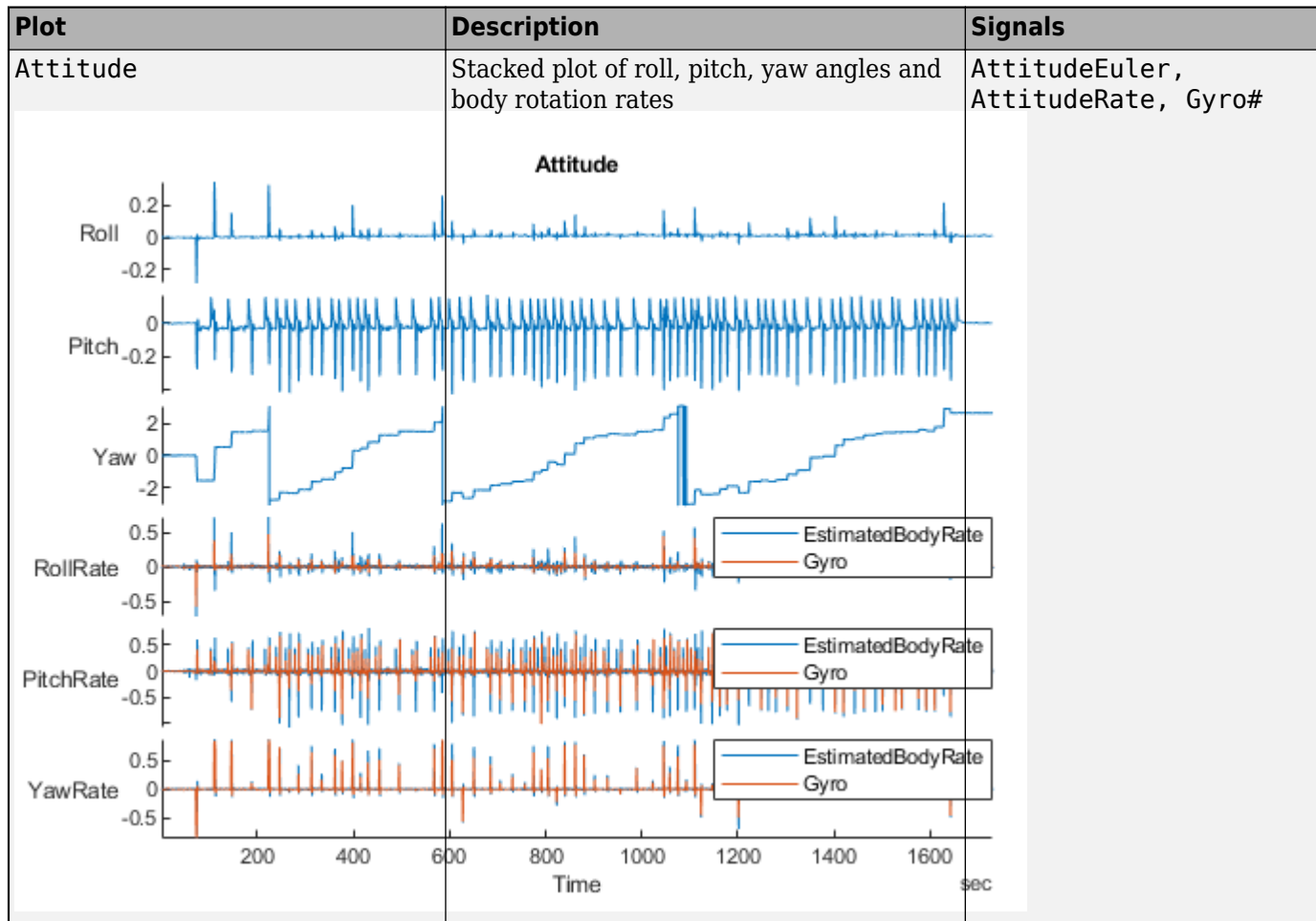
Signal Name	Description	Fields	Unit
Accel#	Raw magnetometer reading from IMU sensor	[ax ay az]	m/s ²
Airspeed#	Airspeed reading of pressure differential, indicated air speed, and temperature	[PressDiff, AirSpeed, Temp]	Pa, m/s, °C
AttitudeEuler	Attitude of UAV in Euler (ZYX) form	[Roll, Pitch, Yaw]	rad
AttitudeRate	Angular velocity along each body axis	[xRotRate, yRotRate, zRotRate]	rad/s
AttitudeTargetEuler	Target attitude of UAV in Euler (ZYX) form	[TargetRoll, TargetPitch, TargetYaw]	rad
Barometer#	Barometer readings for absolute pressure, relative pressure, and temperature	[PressAbs, PressAltitude, Temp]	Pa, m, °C
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, m/s, deg
Gyro#	Raw body angular velocity readings from IMU sensor	[GyroX, GyroY, GyroZ]	rad/s
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

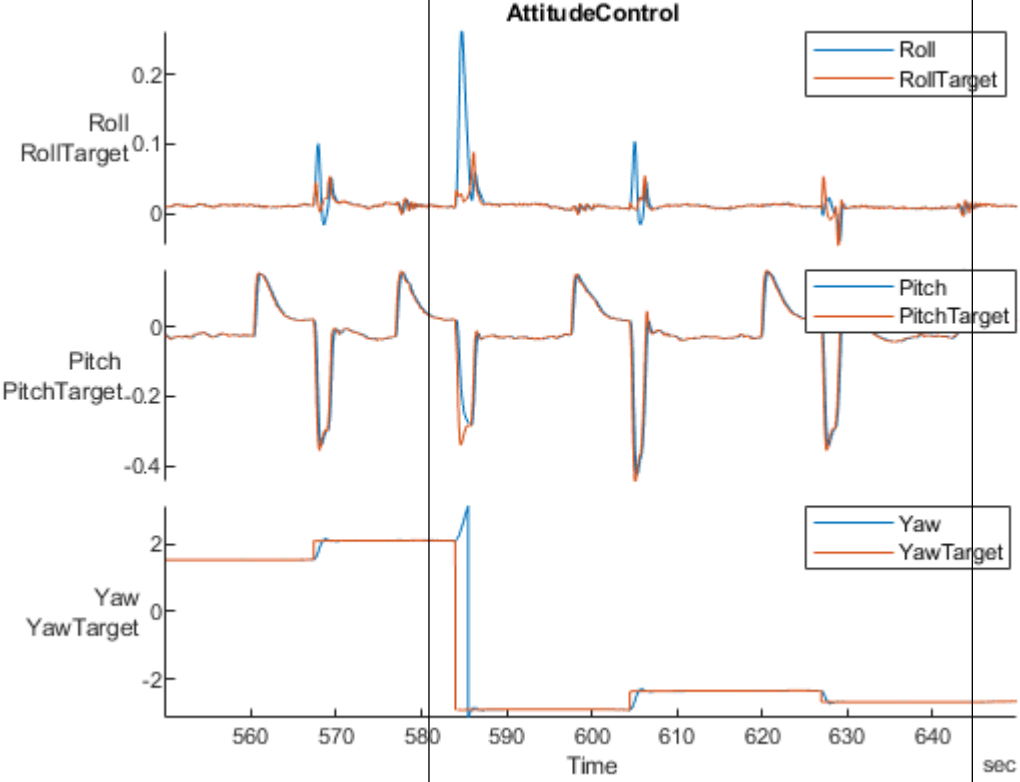
Predefined Plots

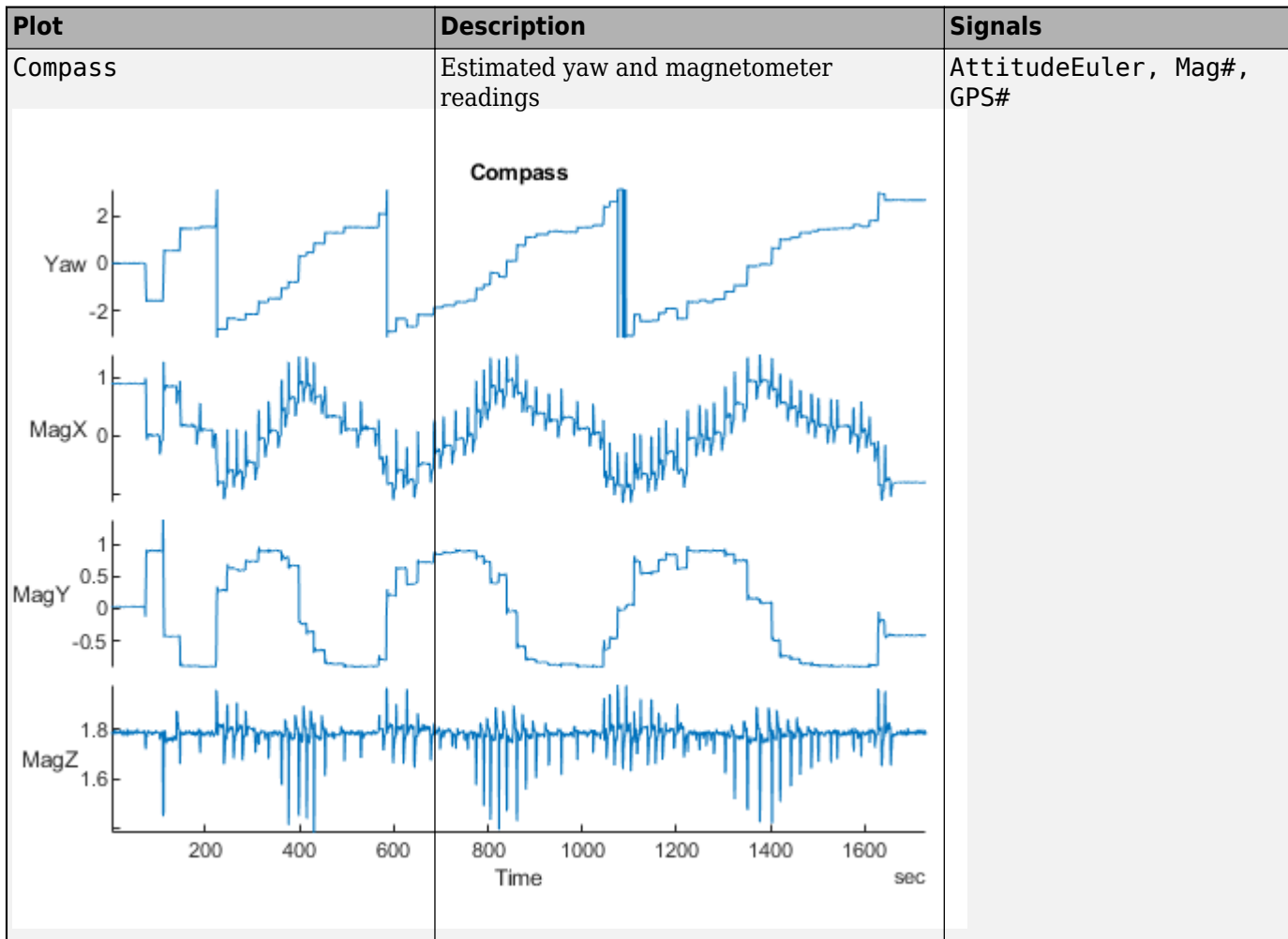
After mapping signals to the list of predefined signals using `mapSignal`, specific plots are made available when calling `show`. To view a list of available plots and their associated signals for your specific object, call `info(mapper, "Plot")`. If you want to define custom plots based on signals,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

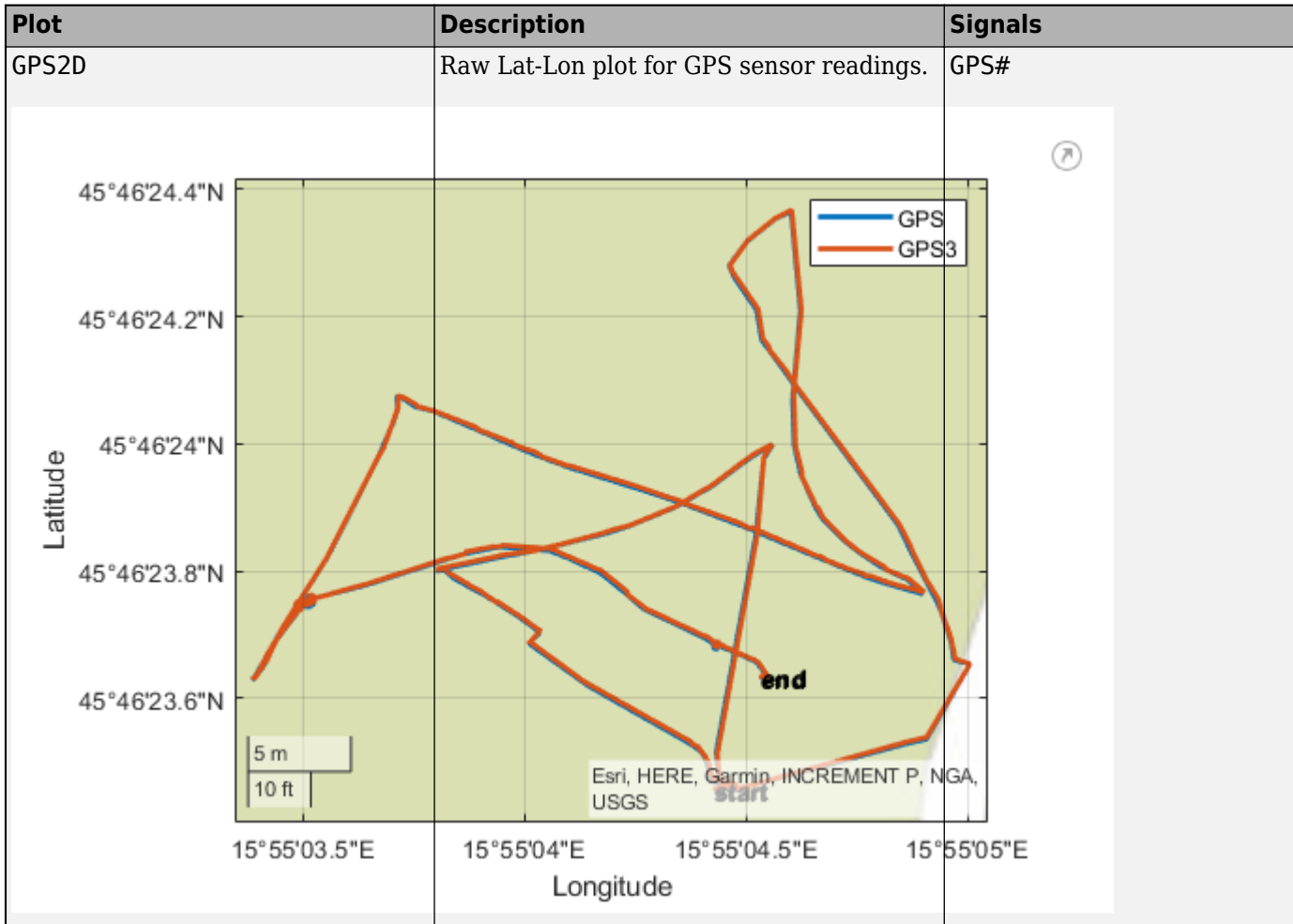
Each predefined plot has a set of required signals that must be mapped.

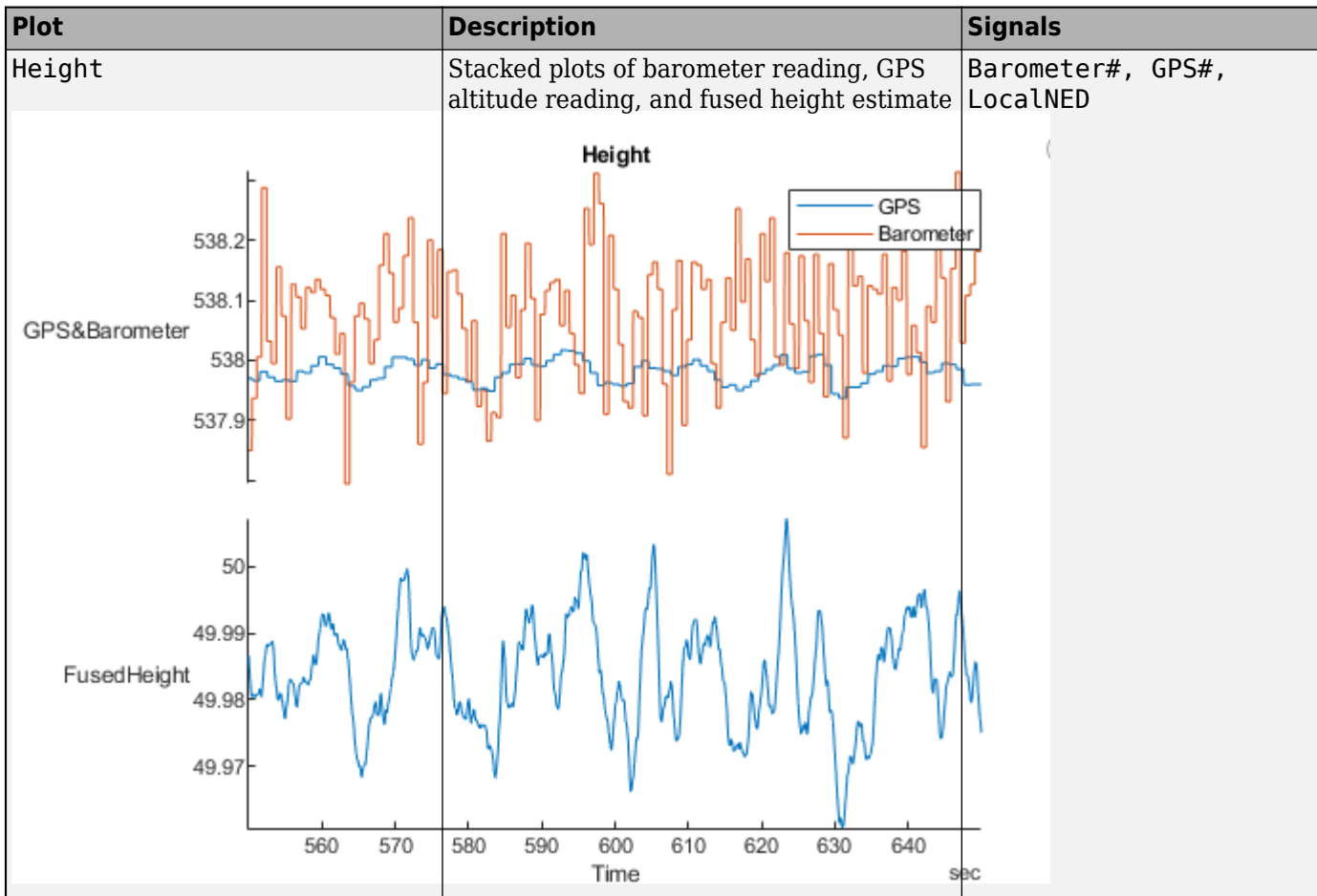
Predefined Plots

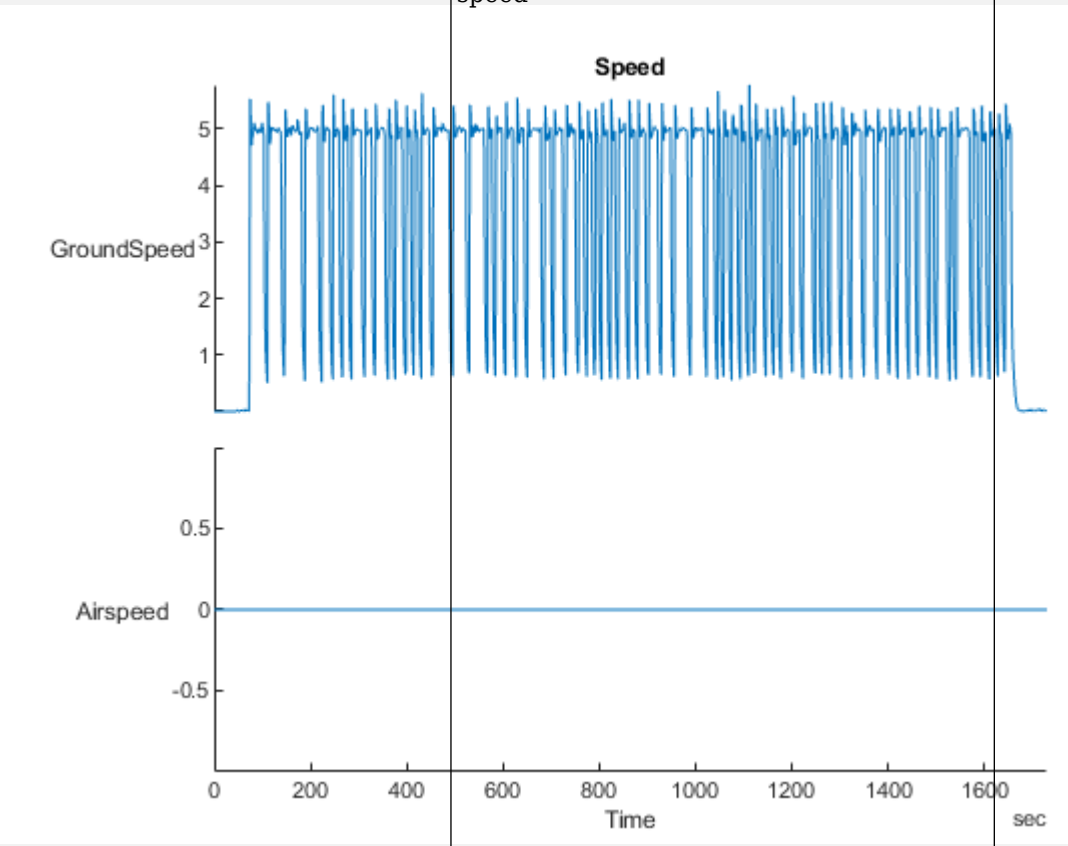


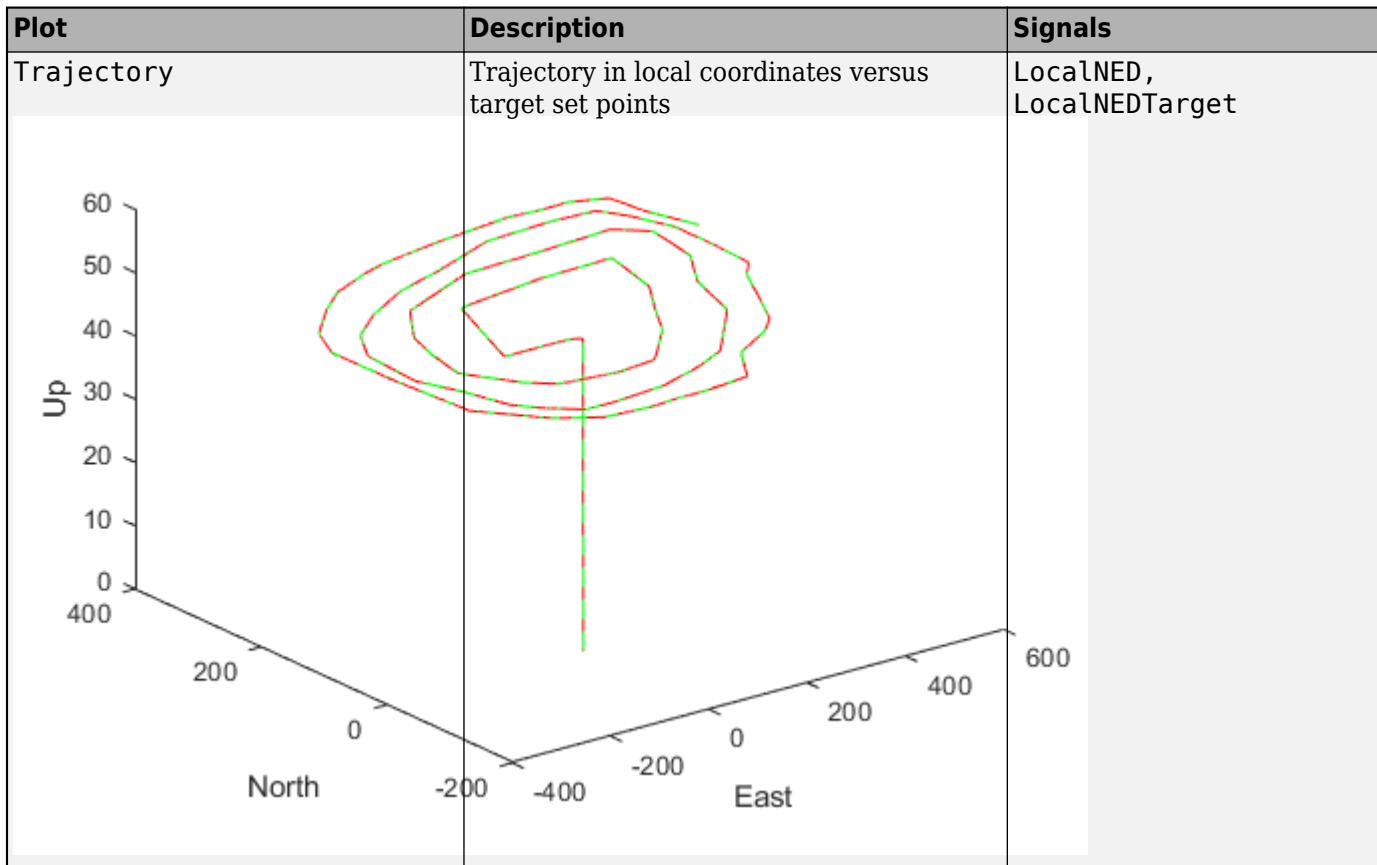
Plot	Description	Signals
<p data-bbox="240 296 483 327">AttitudeControl</p> 	<p data-bbox="691 296 1230 359">Estimated attitude of UAV and the attitude target set point</p>	<p data-bbox="1230 296 1602 359">AttitudeEuler, AttitudeTargetEuler</p>
<p data-bbox="240 1247 358 1278">Battery</p>	<p data-bbox="691 1247 1008 1278">Battery consumption plot</p>	<p data-bbox="1230 1247 1352 1278">Battery</p>

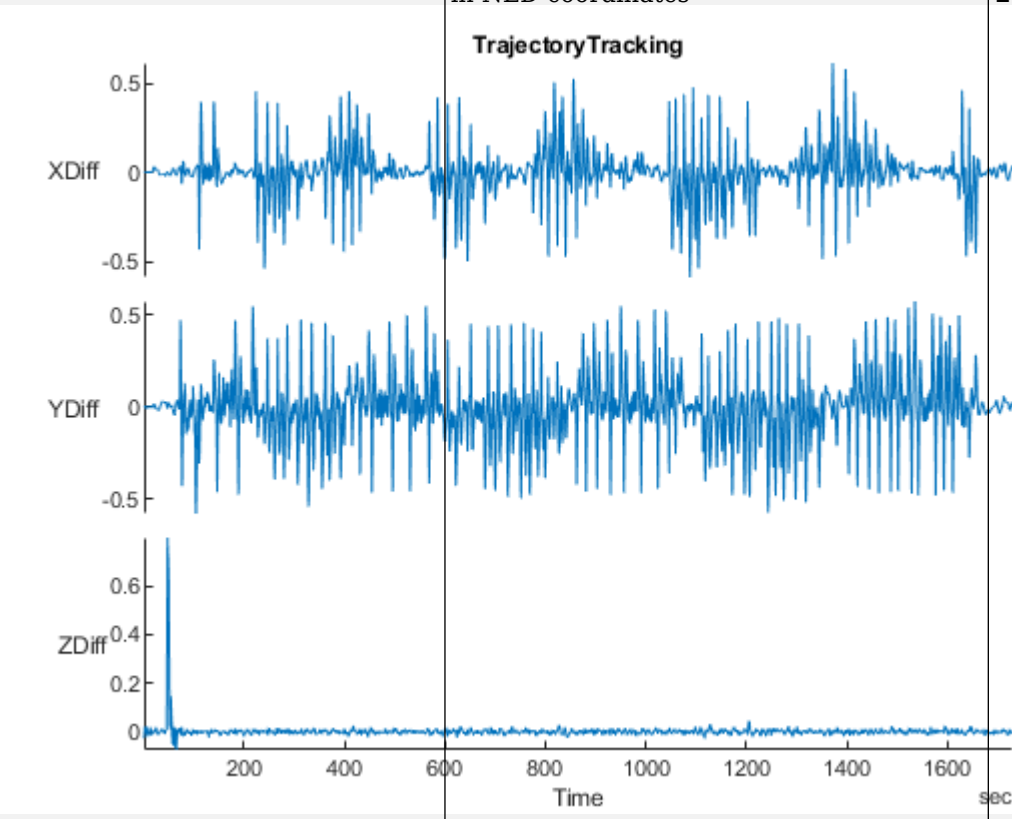


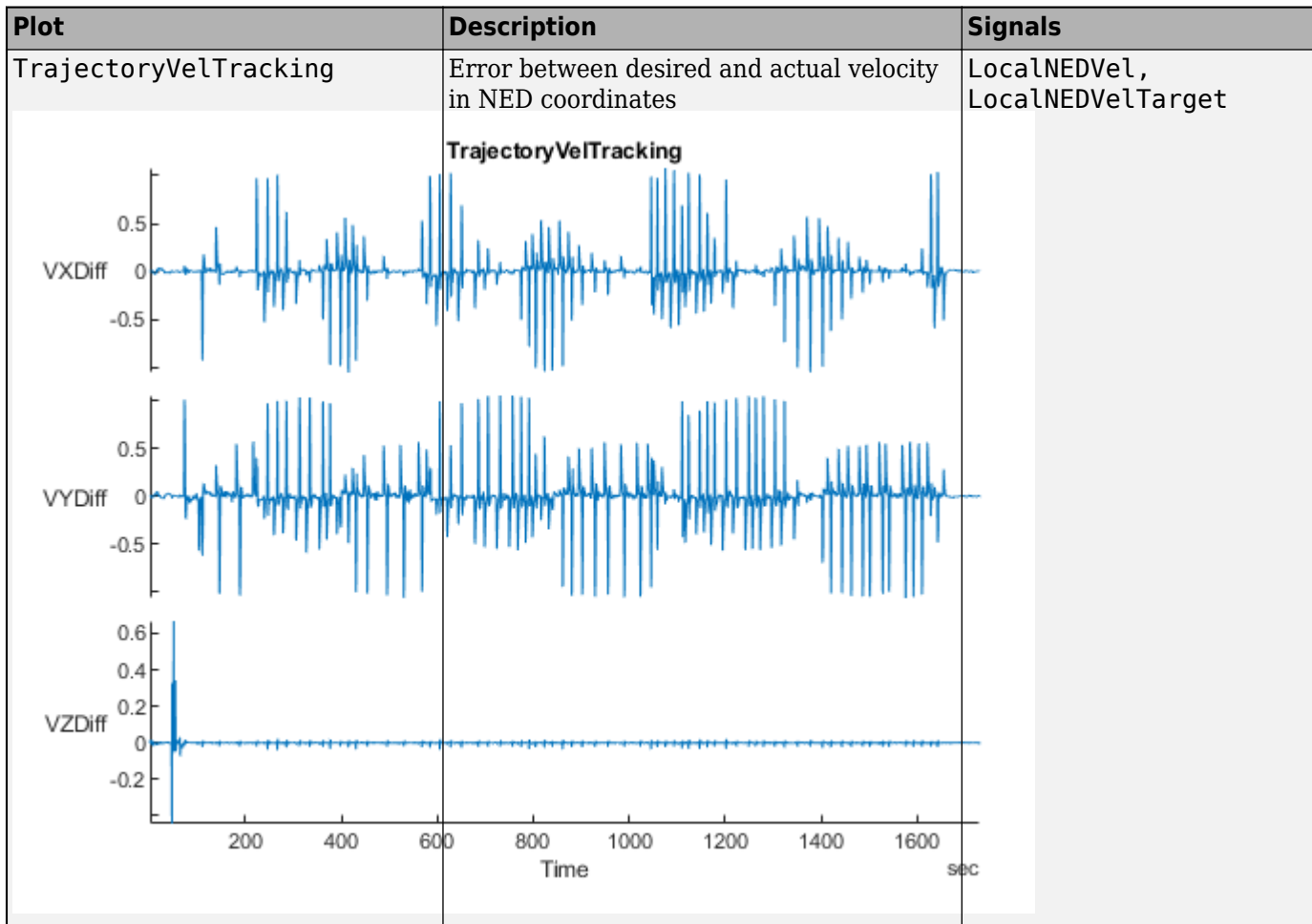




Plot	Description	Signals
<p data-bbox="240 296 324 327">Speed</p> 	<p data-bbox="691 296 1230 359">Stacked plot of ground velocity and air speed</p>	<p data-bbox="1230 296 1604 327">GPS#, Airspeed#</p>



Plot	Description	Signals
<p>TrajectoryTracking</p>  <p>The plot displays three error signals over a 1600-second period. The top plot shows XDiff (horizontal error) fluctuating between approximately -0.5 and 0.5. The middle plot shows YDiff (vertical error) also fluctuating between -0.5 and 0.5. The bottom plot shows ZDiff (depth error), which starts at approximately 0.7 and rapidly decays to near zero by 100 seconds, remaining stable thereafter. The x-axis for all plots is labeled 'Time' in seconds, ranging from 0 to 1600.</p>	<p>Error between desired and actual position in NED coordinates</p>	<p>LocalNED, LocalNEDTarget</p>



See Also

[extract](#) | [flightLogSignalMapping](#) | [info](#) | [mapSignal](#) | [mavlinktlog](#) | [show](#) | [updatePlot](#)

Introduced in R2020b

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 2-83 and [Predefined Plots](#) on page 2-84.

`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

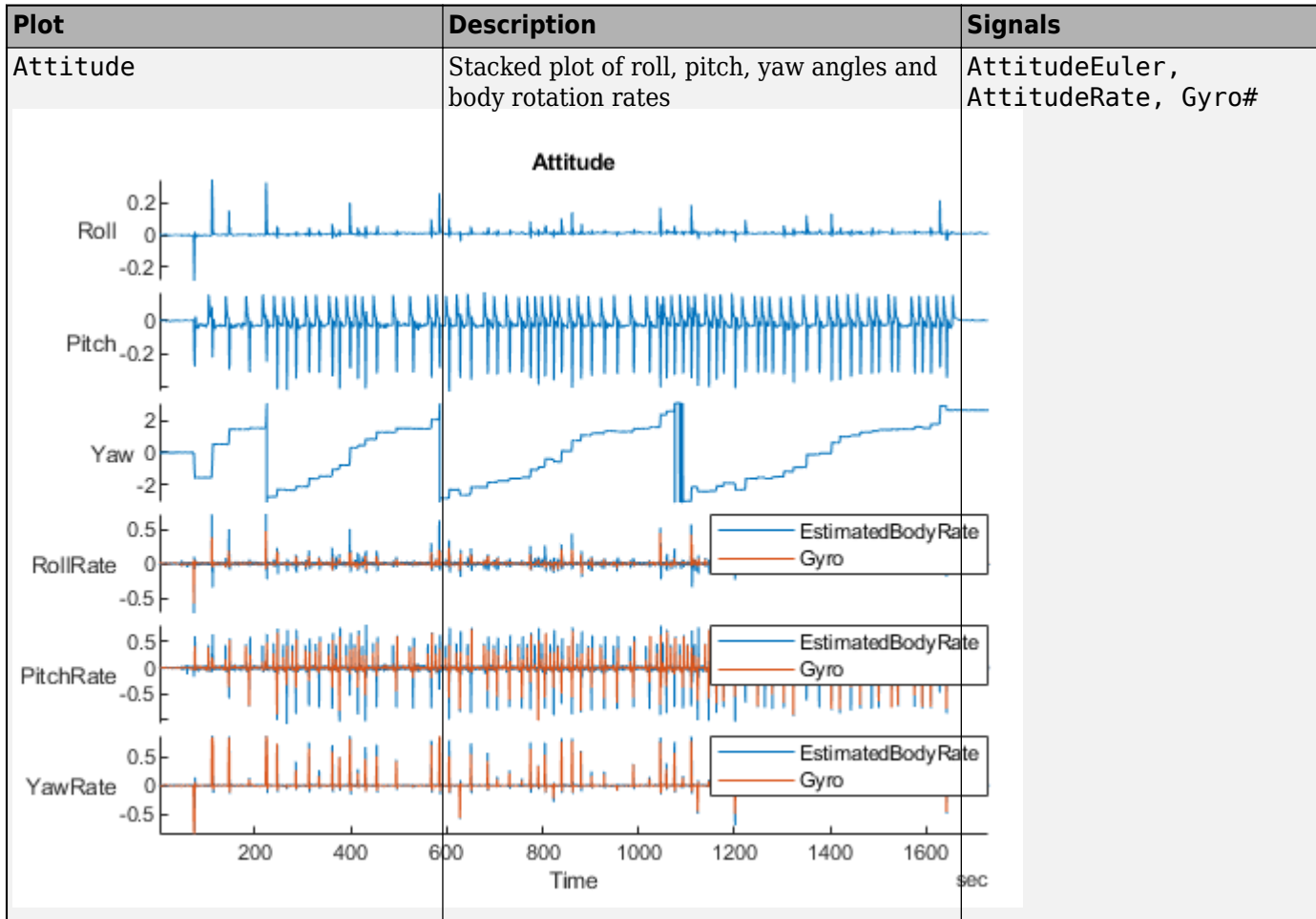
Signal Name	Description	Fields	Unit
Accel#	Raw magnetometer reading from IMU sensor	[ax ay az]	m/s ²
Airspeed#	Airspeed reading of pressure differential, indicated air speed, and temperature	[PressDiff, AirSpeed, Temp]	Pa, m/s, °C
AttitudeEuler	Attitude of UAV in Euler (ZYX) form	[Roll, Pitch, Yaw]	radi
AttitudeRate	Angular velocity along each body axis	[xRotRate, yRotRate, zRotRate]	rad/s
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, m, °C
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, m/s, deg
Gyro#	Raw body angular velocity readings from IMU sensor	[GyroX, GyroY, GyroZ]	rad/s
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

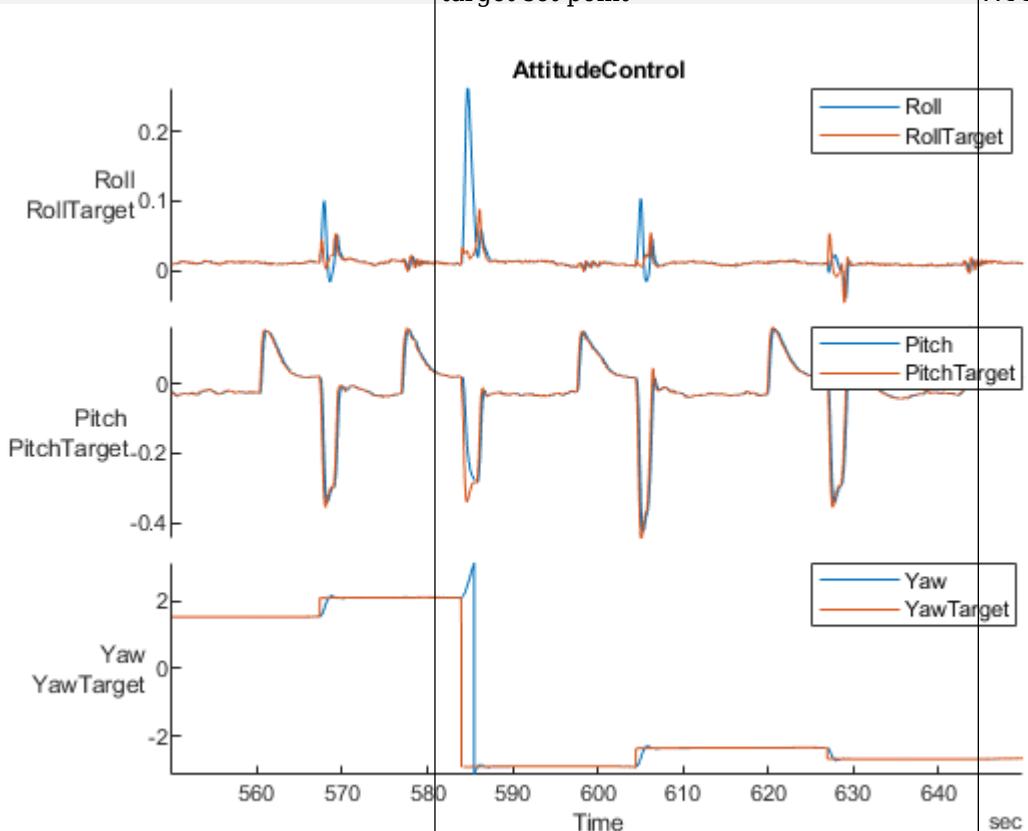
Predefined Plots

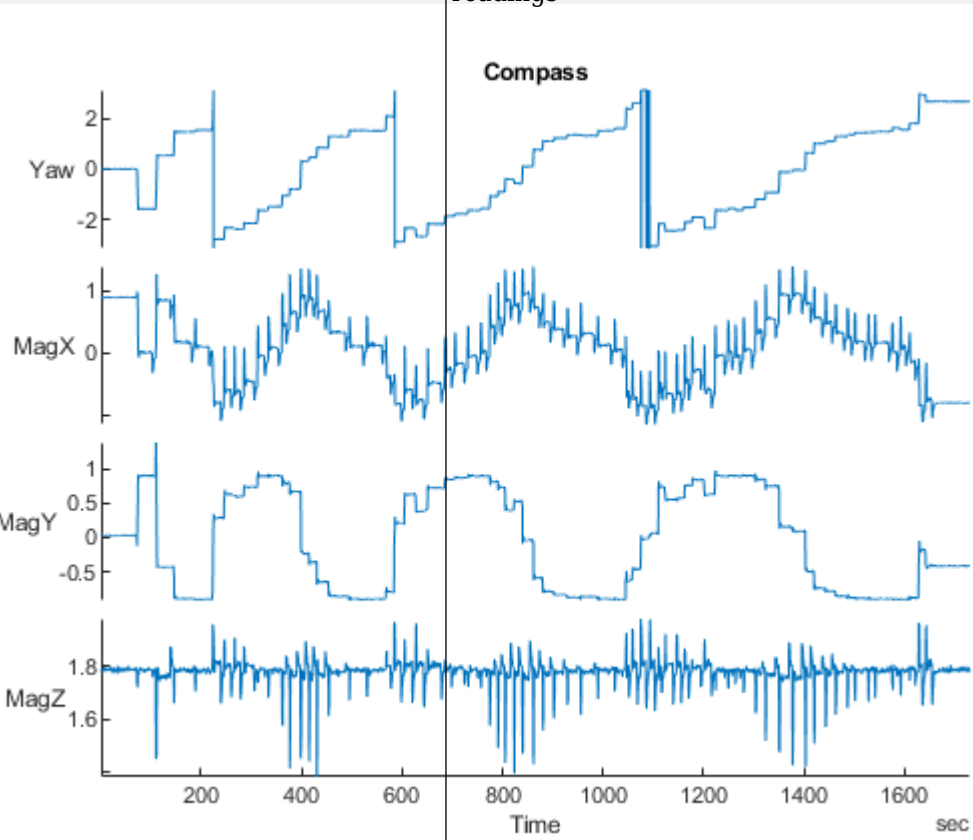
After mapping signals to the list of predefined signals using `mapSignal`, specific plots are made available when calling `show`. To view a list of available plots and their associated signals for your specific object, call `info(mapper, "Plot")`. If you want to define custom plots based on signals,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

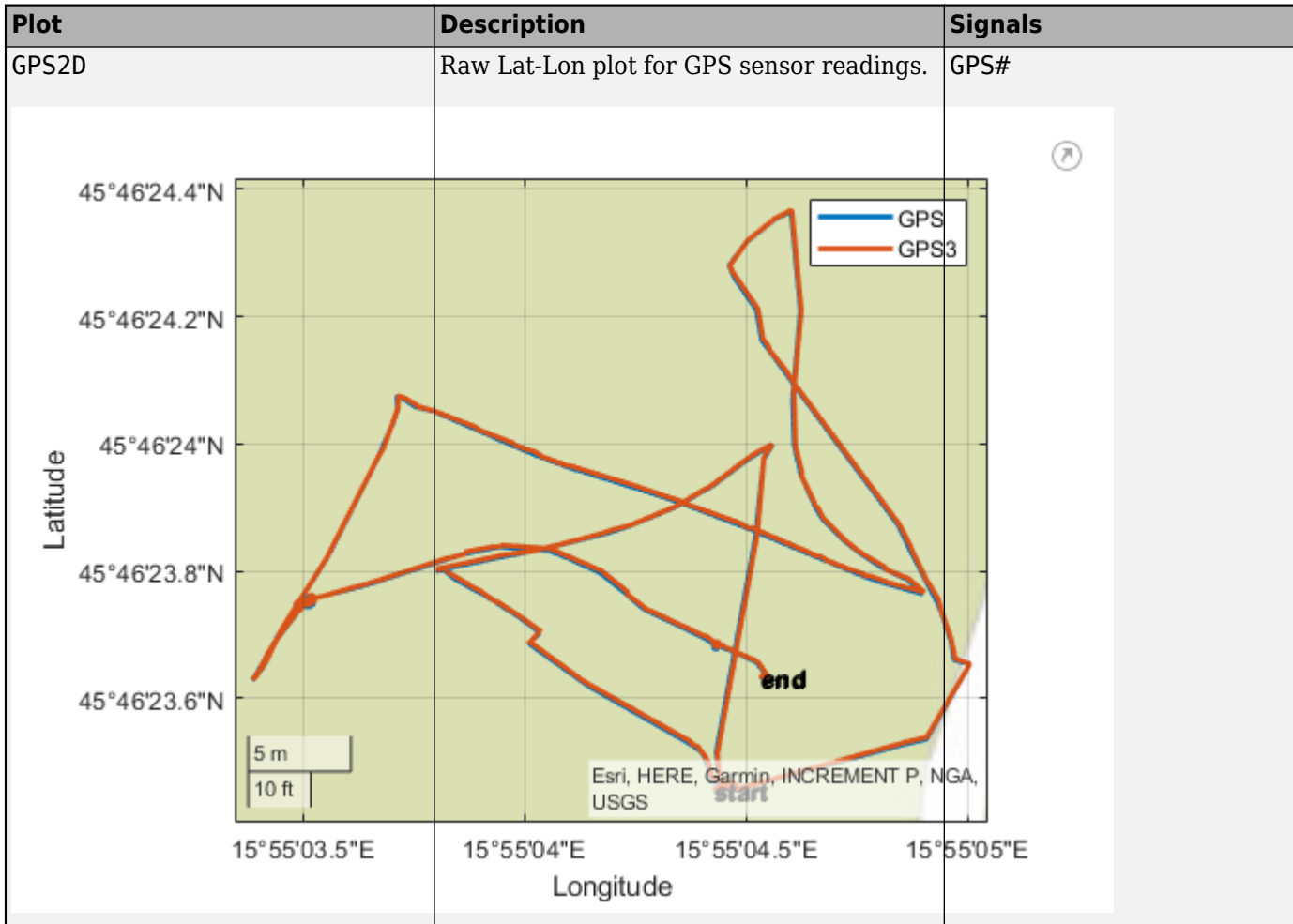
Each predefined plot has a set of required signals that must be mapped.

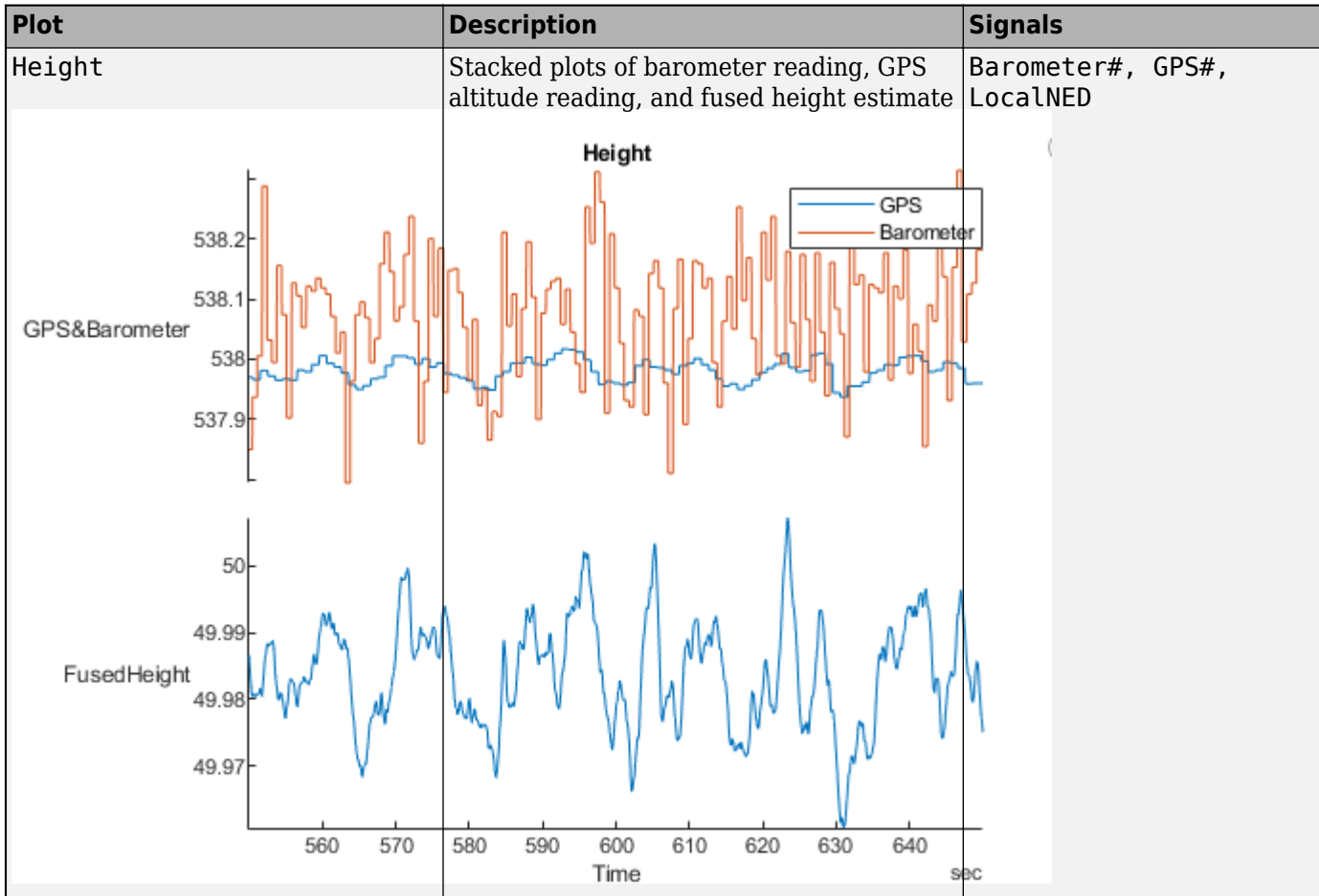
Predefined Plots

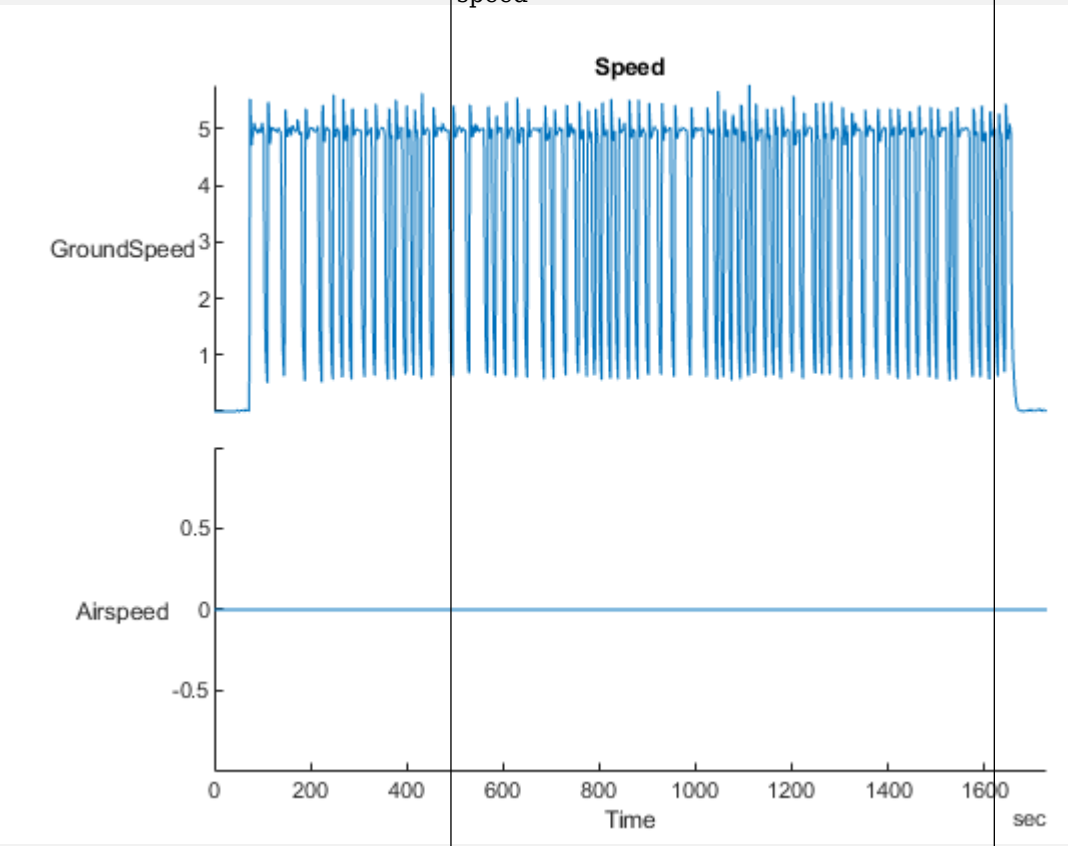


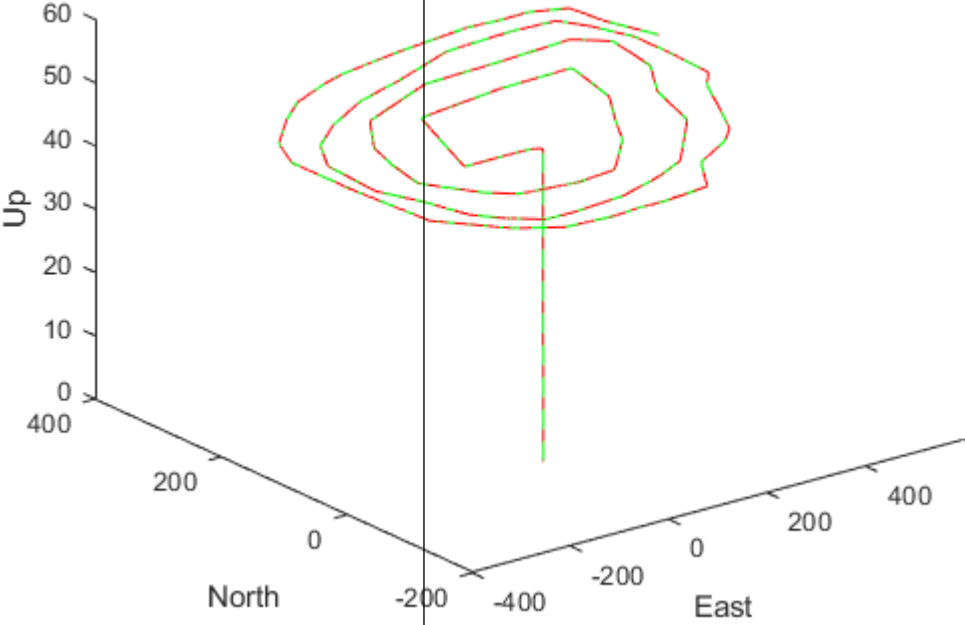
Plot	Description	Signals
<p data-bbox="240 296 483 327">AttitudeControl</p> 	<p data-bbox="691 296 1230 359">Estimated attitude of UAV and the attitude target set point</p>	<p data-bbox="1230 296 1602 359">AttitudeEuler, AttitudeTargetEuler</p>
<p data-bbox="240 1247 358 1278">Battery</p>	<p data-bbox="691 1247 1008 1278">Battery consumption plot</p>	<p data-bbox="1230 1247 1349 1278">Battery</p>

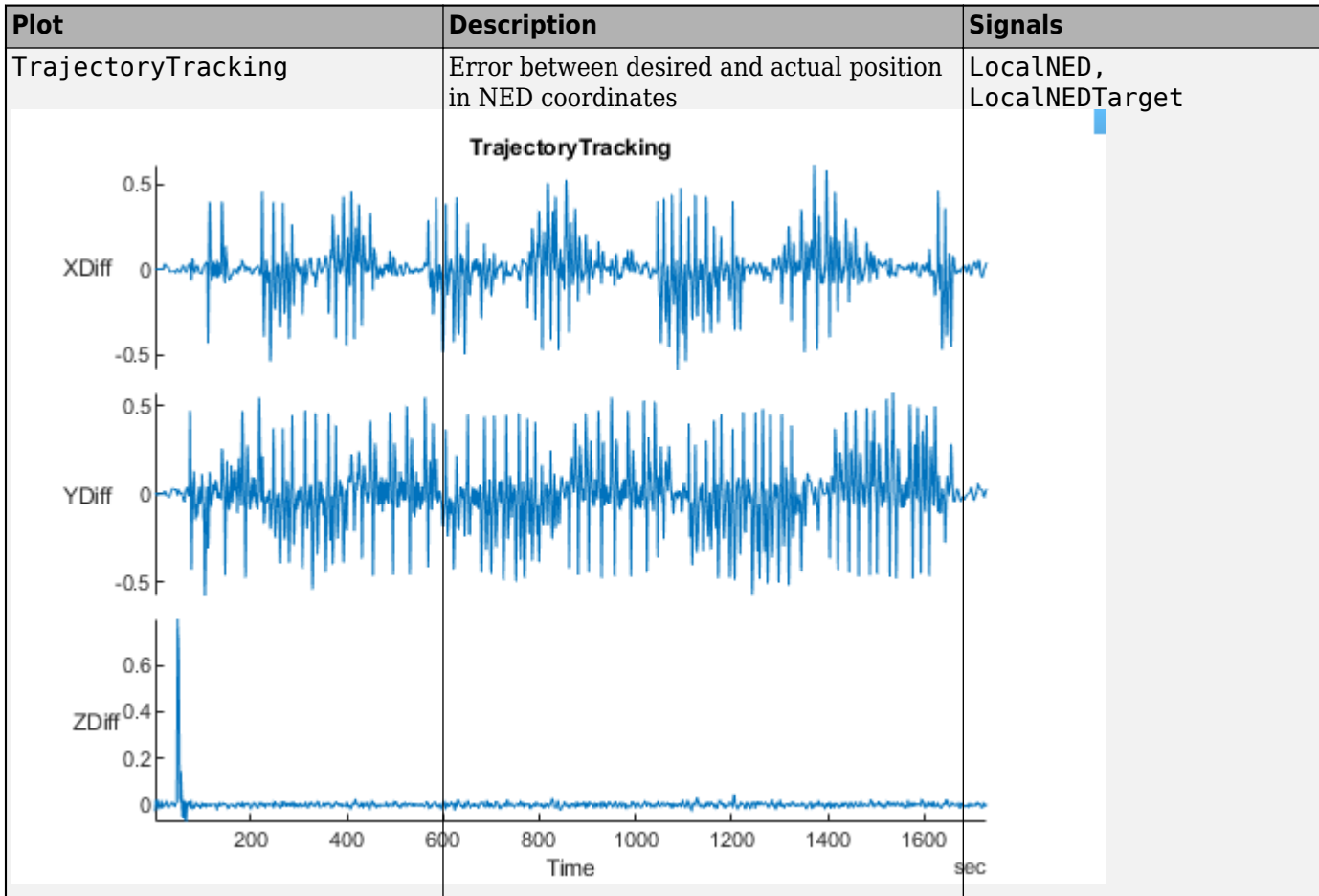
Plot	Description	Signals
<p data-bbox="240 296 358 327">Compass</p> 	<p data-bbox="691 296 1117 359">Estimated yaw and magnetometer readings</p>	<p data-bbox="1232 296 1560 359">AttitudeEuler, Mag#, GPS#</p>

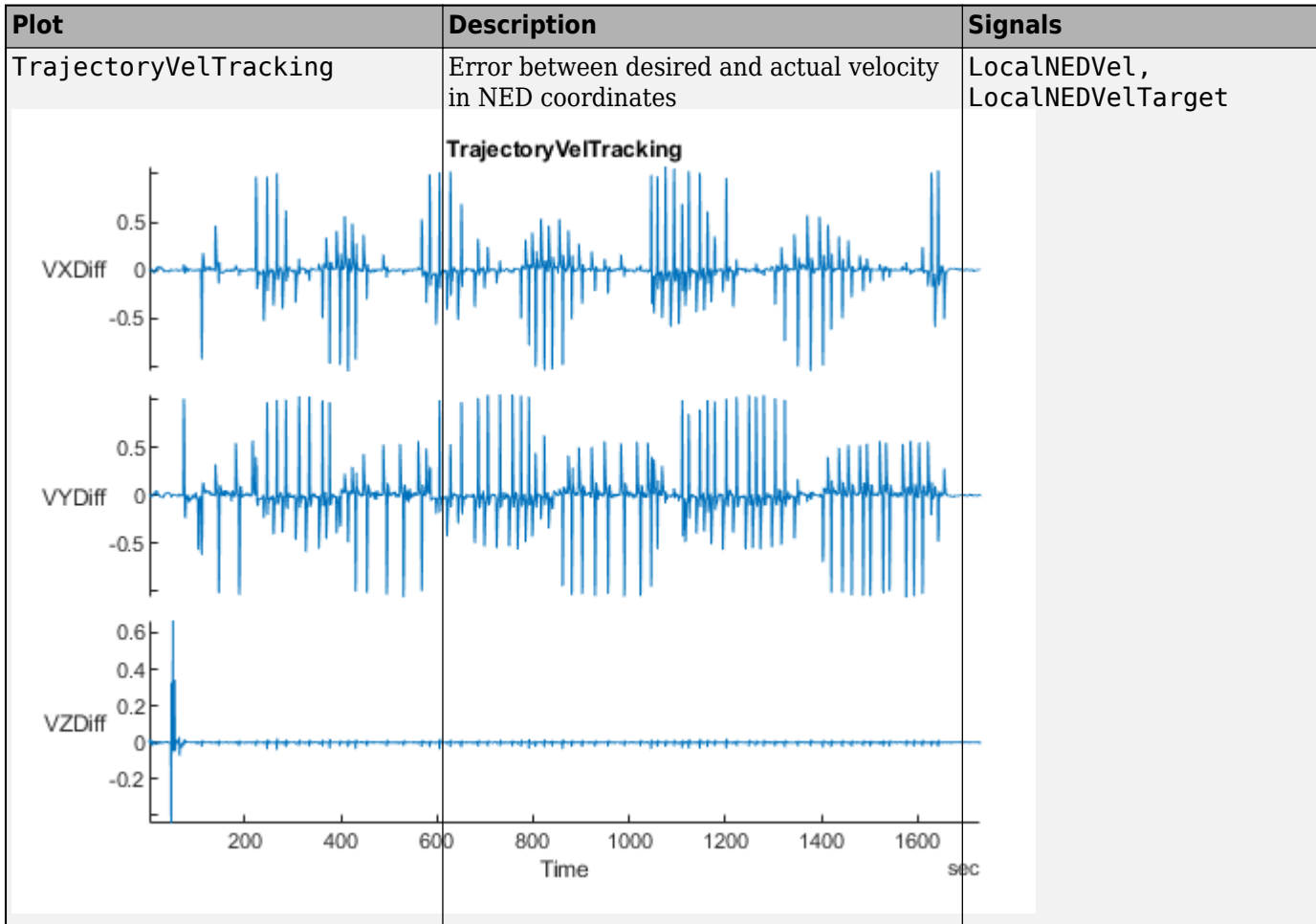




Plot	Description	Signals
<p data-bbox="240 296 324 327">Speed</p> 	<p data-bbox="691 296 1230 359">Stacked plot of ground velocity and air speed</p>	<p data-bbox="1230 296 1604 327">GPS#, Airspeed#</p>

Plot	Description	Signals
<p data-bbox="240 296 406 327">Trajectory</p> 	<p data-bbox="691 296 1159 359">Trajectory in local coordinates versus target set points</p>	<p data-bbox="1232 296 1468 359">LocalNED, LocalNEDTarget</p>





See Also

[extract](#) | [flightLogSignalMapping](#) | [info](#) | [mapSignal](#) | [mavlinktlog](#) | [show](#) | [updatePlot](#)

Introduced in R2020b

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 2-95 and Predefined Plots on page 2-96.

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

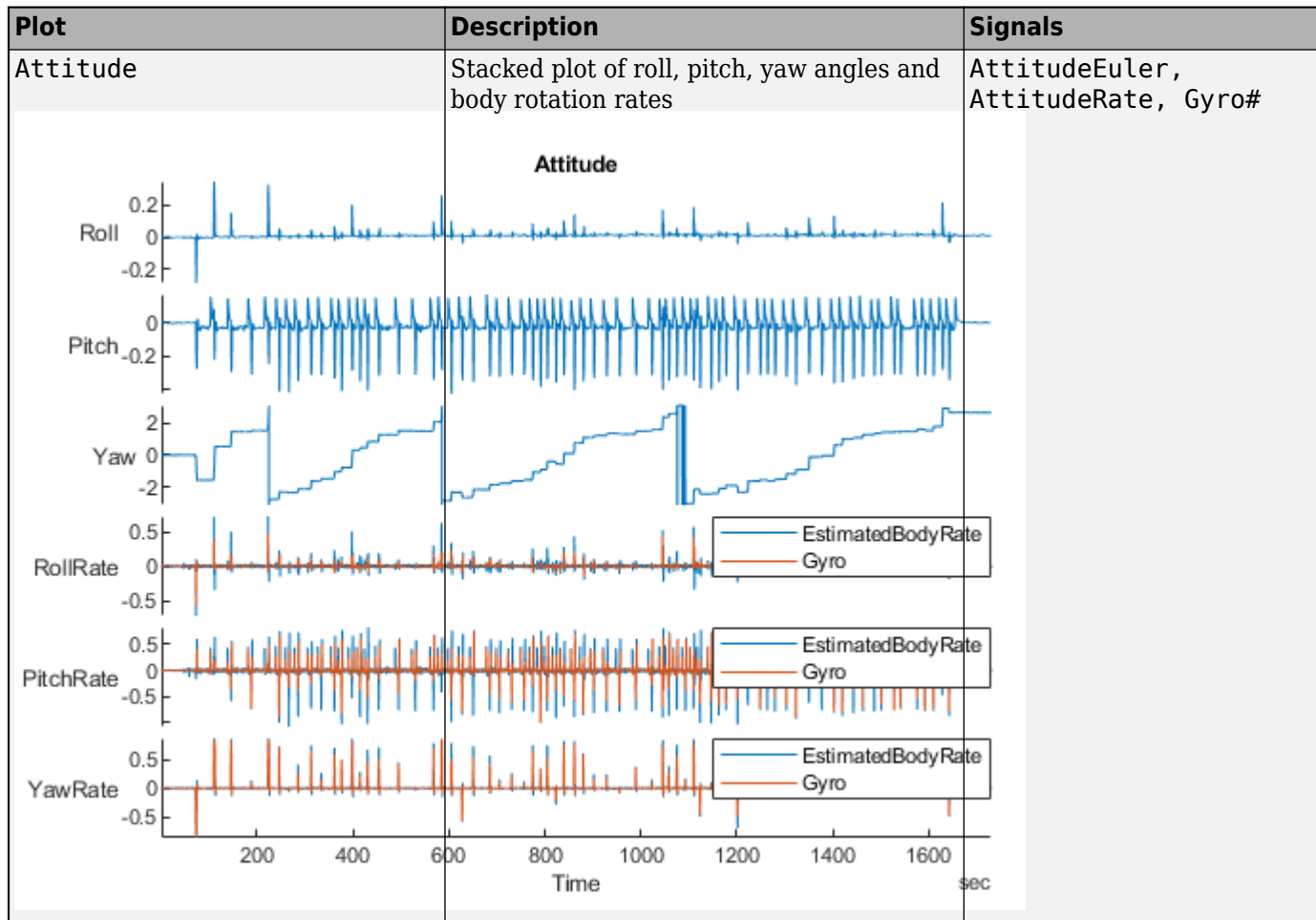
Signal Name	Description	Fields	Unit
Accel#	Raw magnetometer reading from IMU sensor	[ax ay az]	m/s ²
Airspeed#	Airspeed reading of pressure differential, indicated air speed, and temperature	[PressDiff, AirSpeed, Temp]	Pa, m/s, °C
AttitudeEuler	Attitude of UAV in Euler (ZYX) form	[Roll, Pitch, Yaw]	rad
AttitudeRate	Angular velocity along each body axis	[xRotRate, yRotRate, zRotRate]	rad/s
AttitudeTargetEuler	Target attitude of UAV in Euler (ZYX) form	[TargetRoll, TargetPitch, TargetYaw]	rad
Barometer#	Barometer readings for absolute pressure, relative pressure, and temperature	[PressAbs, PressAltitude, Temp]	Pa, m, °C
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, m, m/s, deg, deg
Gyro#	Raw body angular velocity readings from IMU sensor	[GyroX, GyroY, GyroZ]	rad/s
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

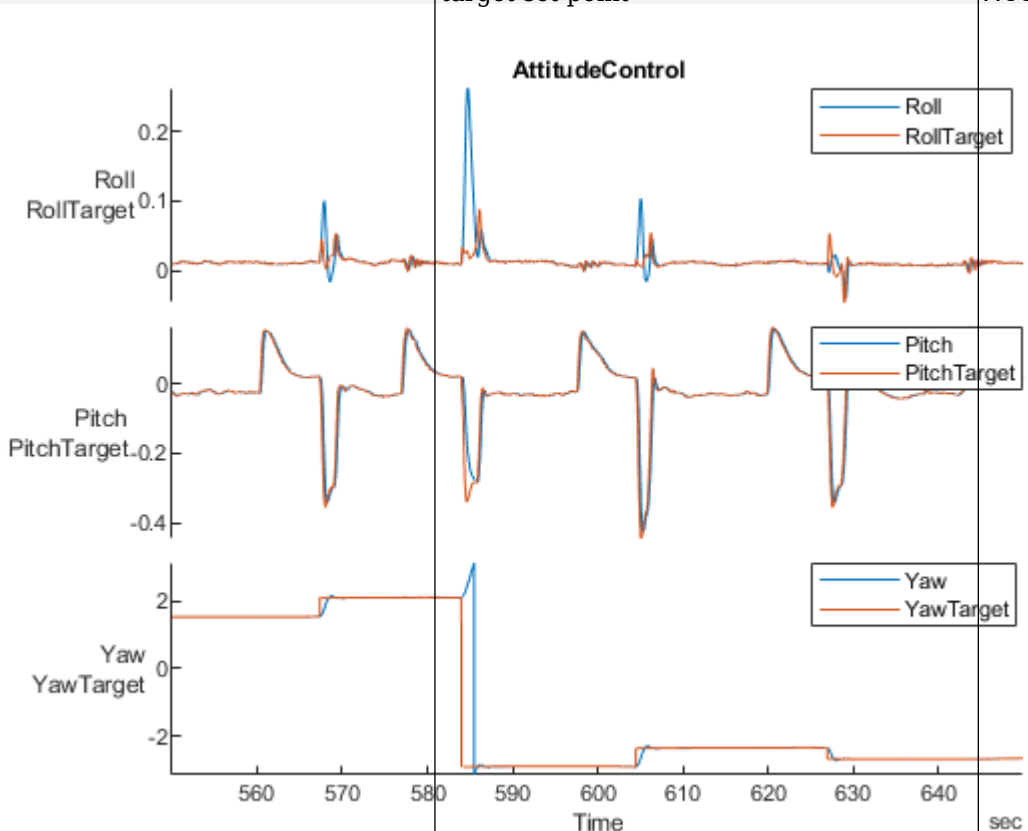
Predefined Plots

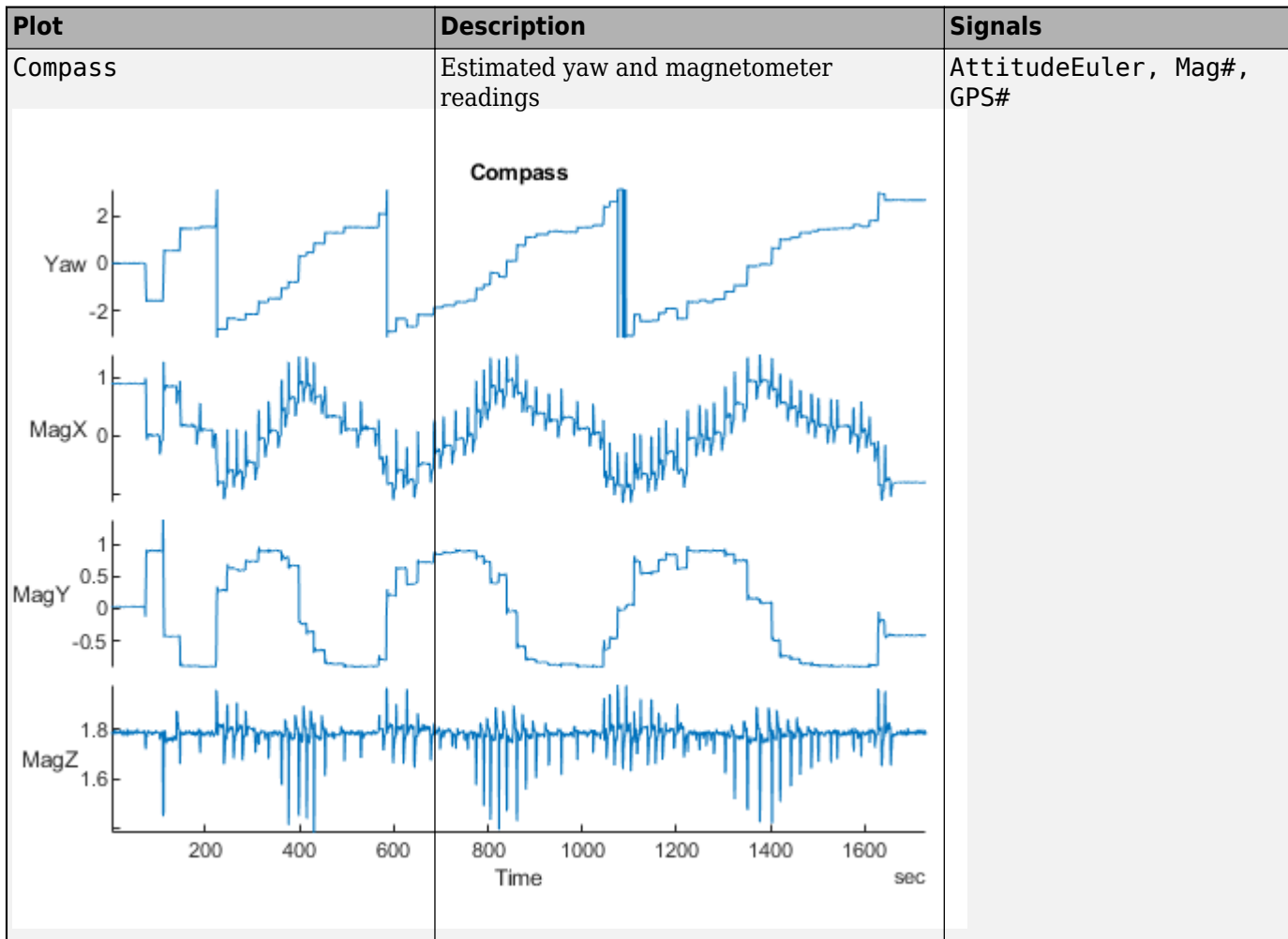
After mapping signals to the list of predefined signals using `mapSignal`, specific plots are made available when calling `show`. To view a list of available plots and their associated signals for your specific object, call `info(mapper, "Plot")`. If you want to define custom plots based on signals,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

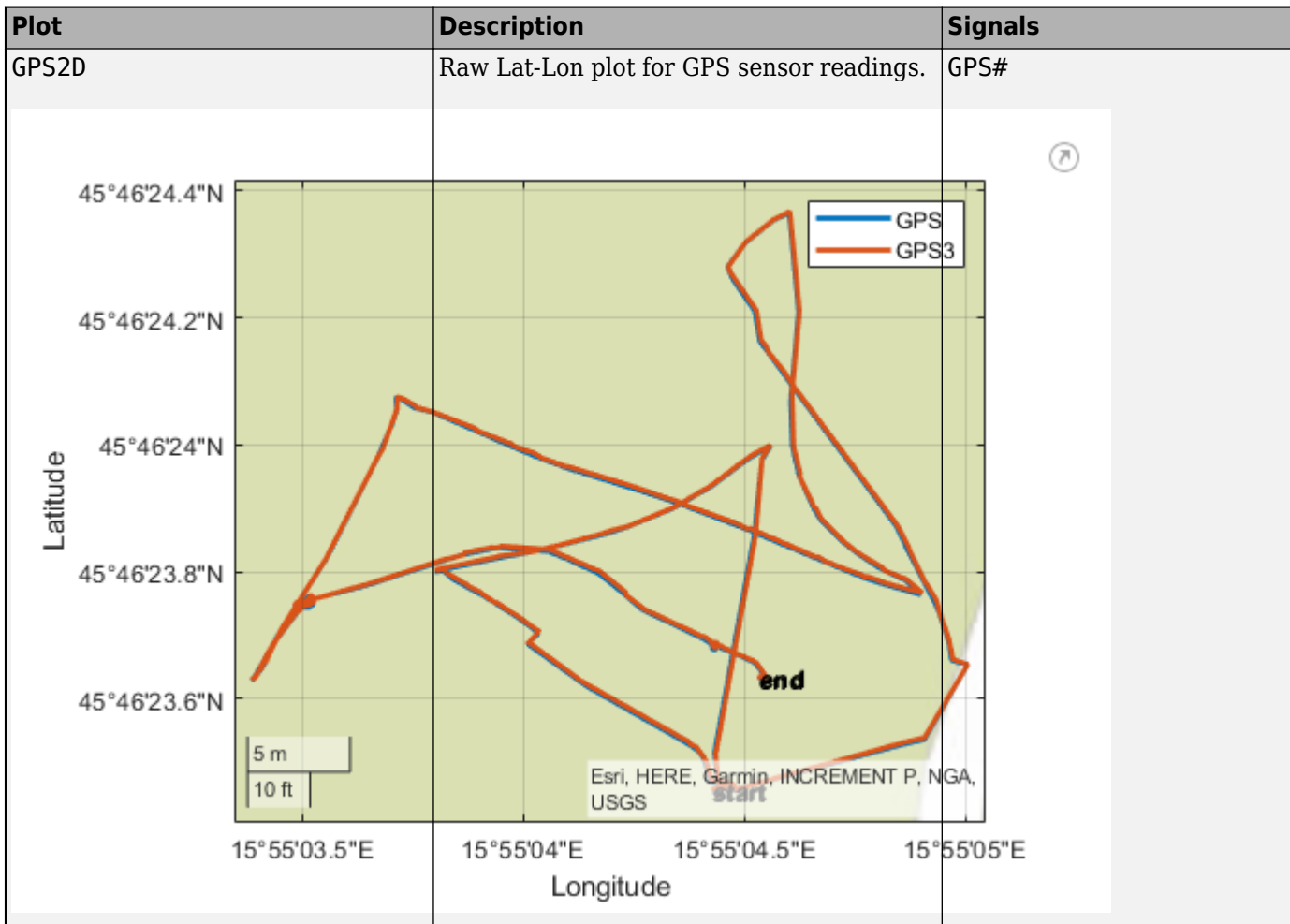
Each predefined plot has a set of required signals that must be mapped.

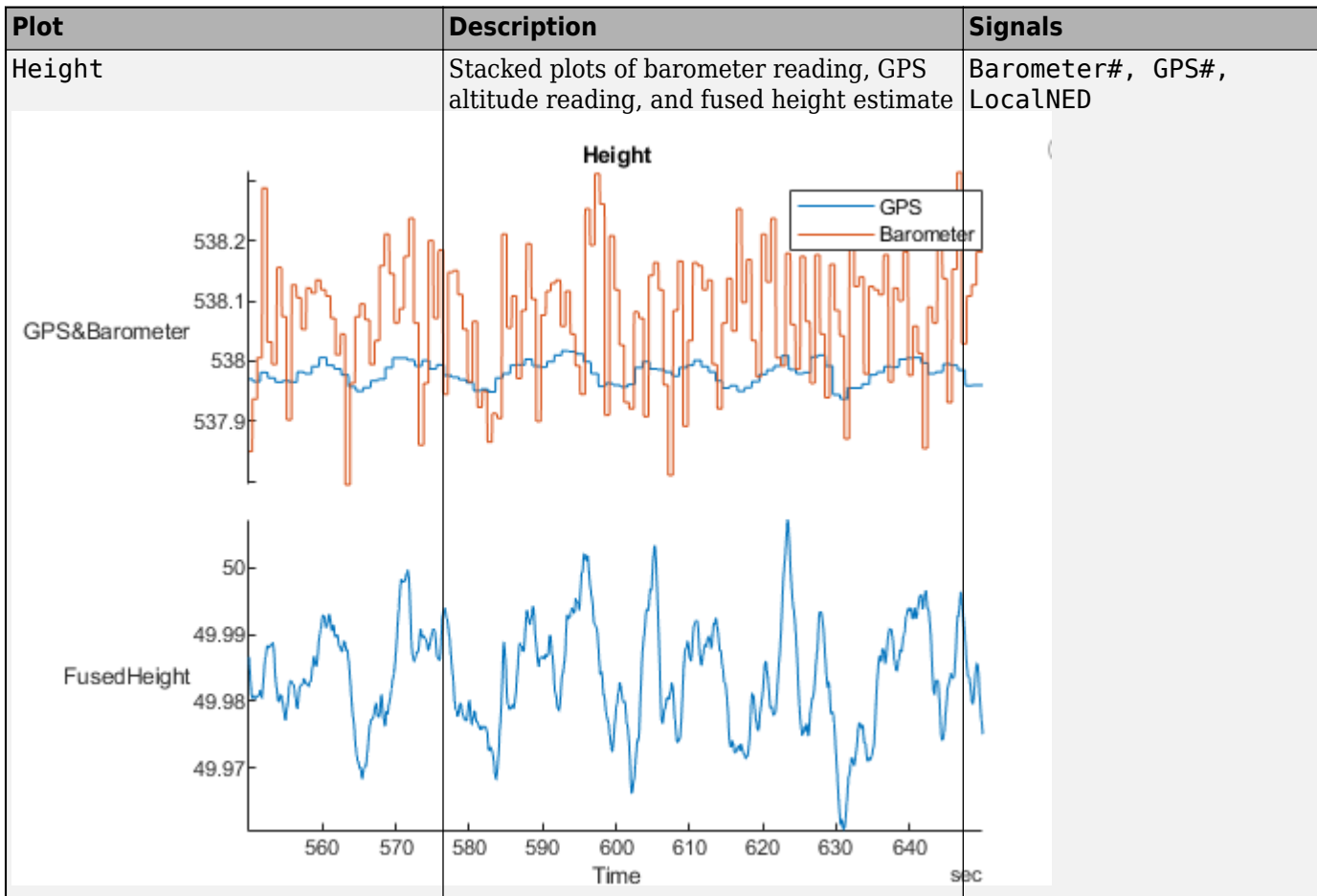
Predefined Plots

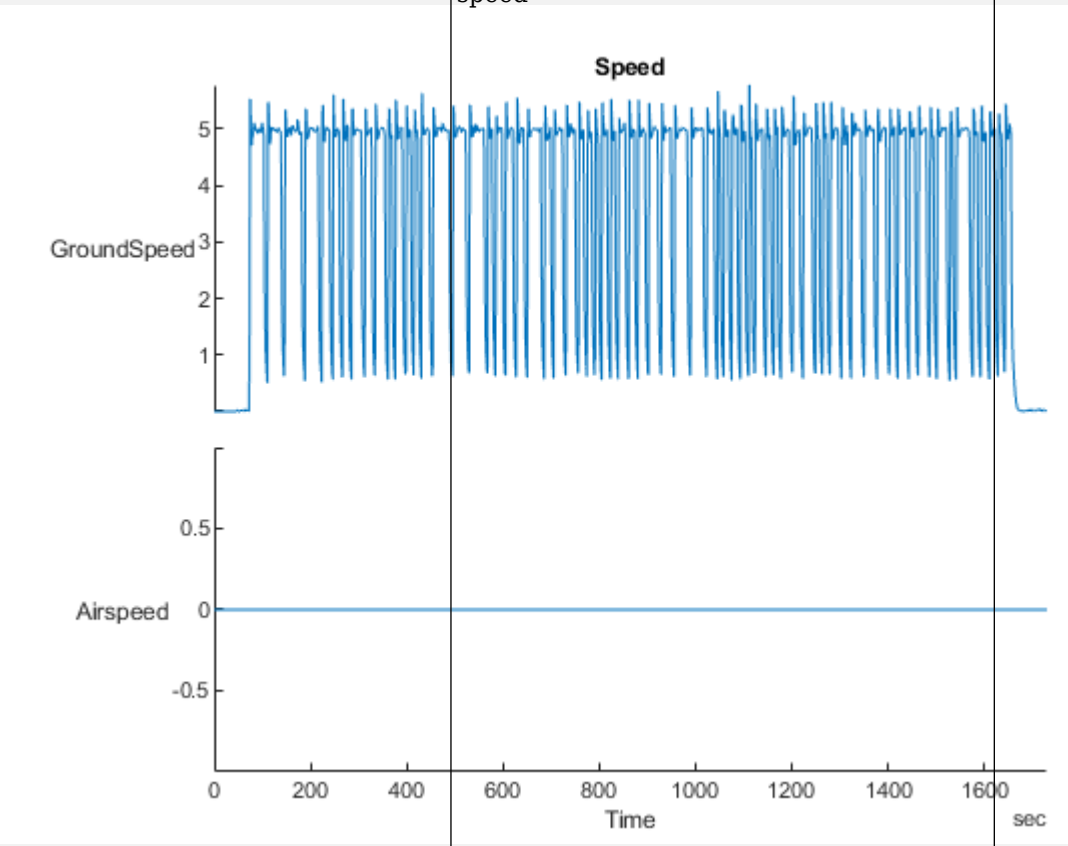


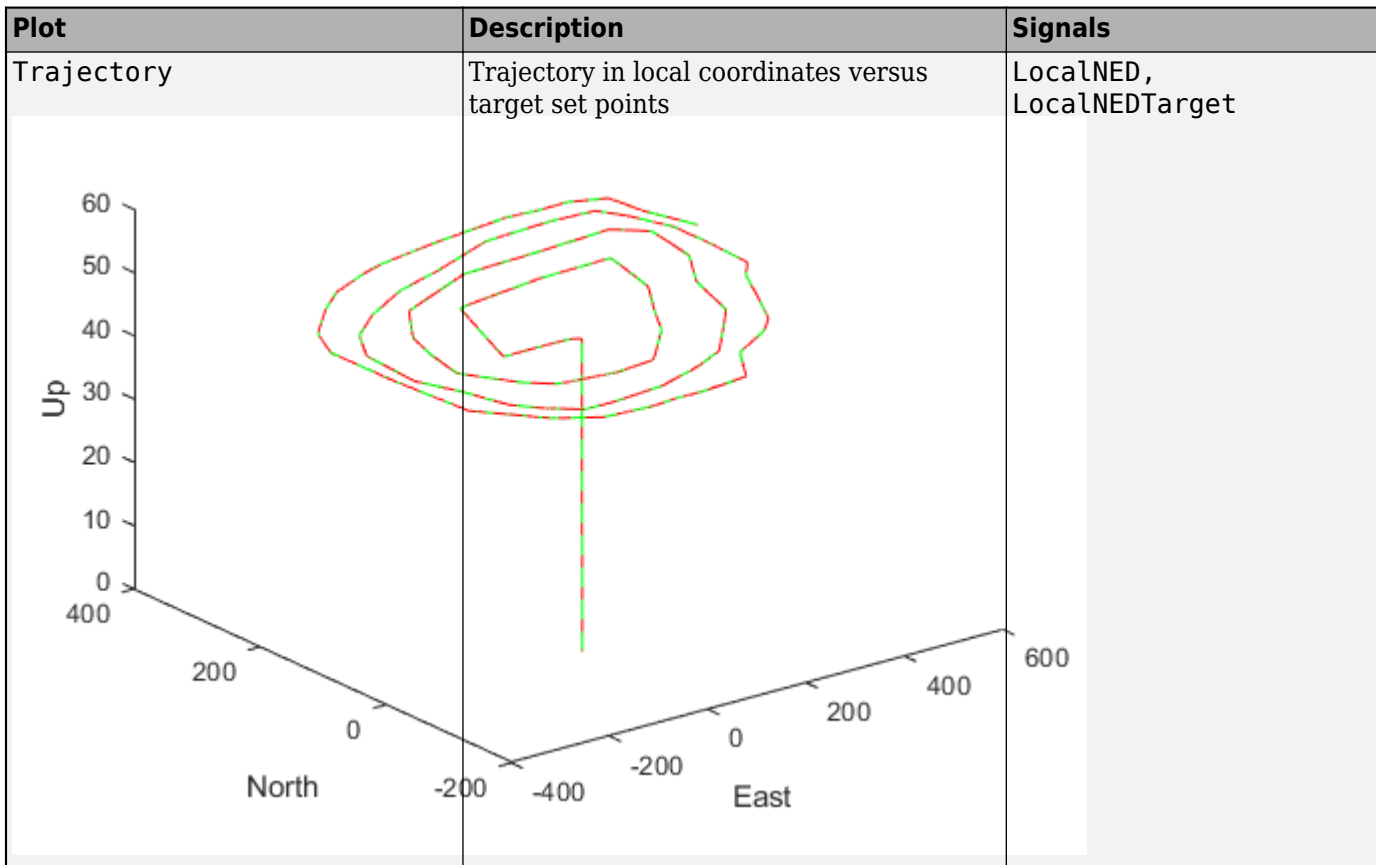
Plot	Description	Signals
<p data-bbox="240 296 483 327">AttitudeControl</p> 	<p data-bbox="691 296 1230 359">Estimated attitude of UAV and the attitude target set point</p>	<p data-bbox="1230 296 1602 359">AttitudeEuler, AttitudeTargetEuler</p>
<p data-bbox="240 1241 358 1272">Battery</p>	<p data-bbox="691 1241 1008 1272">Battery consumption plot</p>	<p data-bbox="1230 1241 1349 1272">Battery</p>

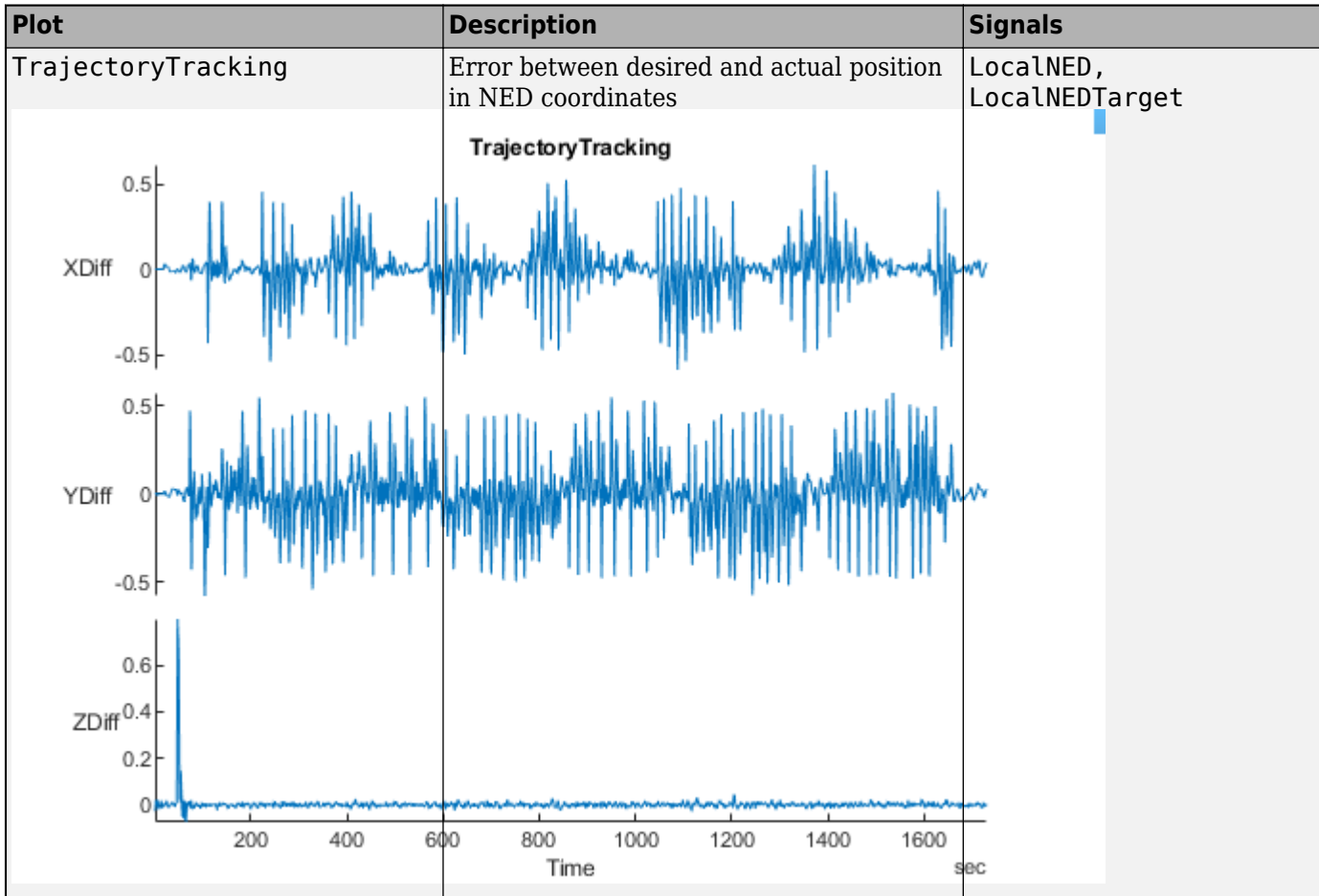






Plot	Description	Signals
<p data-bbox="240 296 324 327">Speed</p> 	<p data-bbox="691 296 1230 359">Stacked plot of ground velocity and air speed</p>	<p data-bbox="1230 296 1604 327">GPS#, Airspeed#</p>





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

TrajectoryVelTracking

The plot displays three stacked time-series signals for velocity errors in NED coordinates. The x-axis represents time in seconds, ranging from 0 to 1600. The top signal, VXDiff, shows high-frequency oscillations between -0.5 and 0.5. The middle signal, VYDiff, shows similar oscillations between -0.5 and 0.5. The bottom signal, VZDiff, shows a sharp initial spike to 0.6, followed by smaller oscillations between -0.2 and 0.2.

See Also

[extract](#) | [flightLogSignalMapping](#) | [info](#) | [mapSignal](#) | [mavlinktlog](#) | [show](#)

Introduced in R2020b

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=133×3 table

Name	Value	
"MAV_CMD_NAV_WAYPOINT"	16	"Navigate to waypoint."
"MAV_CMD_NAV_LOITER_UNLIM"	17	"Loiter around this waypoint an unlimited amount of time"
"MAV_CMD_NAV_LOITER_TURNS"	18	"Loiter around this waypoint for X turns"
"MAV_CMD_NAV_LOITER_TIME"	19	"Loiter around this waypoint for X seconds"
"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"
"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 only)"
"MAV_CMD_NAV_FOLLOW"	25	"Vehicle following, i.e. this waypoint represents a target"
"MAV_CMD_NAV_CONTINUE_AND_CHANGE_ALT"	30	"Continue on the current course and climb/descend to the specified altitude"
"MAV_CMD_NAV_LOITER_TO_ALT"	31	"Begin loiter at the specified Latitude and Longitude and then continue on the current course and climb/descend to the specified altitude"
"MAV_CMD_DO_FOLLOW"	32	"Begin following a target"
"MAV_CMD_DO_FOLLOW_REPOSITION"	33	"Reposition the MAV after a follow target is reached"
"MAV_CMD_DO_ORBIT"	34	"Start orbiting on the circumference of a circle of the specified radius at the specified location and altitude"
"MAV_CMD_NAV_ROI"	80	"Sets the region of interest (ROI) for a camera or vision sensor"
:		

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 is present and responsive"

```
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.

See Also

Functions

`createmsg` | `enum2num` | `enuminfo` | `msginfo` | `num2enum`

Objects

`mavlinkclient` | `mavlinkdialect` | `mavlinkio` | `mavlinksub`

Introduced in R2019a

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=133×3 table

Name	Value	
"MAV_CMD_NAV_WAYPOINT"	16	"Navigate to waypoint."
"MAV_CMD_NAV_LOITER_UNLIM"	17	"Loiter around this waypoint an unlimited amount of time"
"MAV_CMD_NAV_LOITER_TURNS"	18	"Loiter around this waypoint for X turns"
"MAV_CMD_NAV_LOITER_TIME"	19	"Loiter around this waypoint for X seconds"
"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"
"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 only)"
"MAV_CMD_NAV_FOLLOW"	25	"Vehicle following, i.e. this waypoint represents a target"
"MAV_CMD_NAV_CONTINUE_AND_CHANGE_ALT"	30	"Continue on the current course and climb/descend to the specified altitude"
"MAV_CMD_NAV_LOITER_TO_ALT"	31	"Begin loiter at the specified Latitude and Longitude and then continue on the current course and climb/descend to the specified altitude"
"MAV_CMD_DO_FOLLOW"	32	"Begin following a target"
"MAV_CMD_DO_FOLLOW_REPOSITION"	33	"Reposition the MAV after a follow target is reached"
"MAV_CMD_DO_ORBIT"	34	"Start orbiting on the circumference of a circle with the specified radius around the specified location"
"MAV_CMD_NAV_ROI"	80	"Sets the region of interest (ROI) for a MAV. The ROI is a circle with the specified radius around the specified location"
⋮		

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 is present and responsive"

```
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.

See Also

Functions

[createcmd](#) | [enum2num](#) | [enuminfo](#) | [msginfo](#) | [num2enum](#)

Objects

[mavlinkclient](#) | [mavlinkdialect](#) | [mavlinkio](#) | [mavlinksub](#)

Topics

[“Tune UAV Parameters Using MAVLink Parameter Protocol”](#)

Introduced in R2019a

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=133x3 table

Name	Value	
"MAV_CMD_NAV_WAYPOINT"	16	"Navigate to waypoint."
"MAV_CMD_NAV_LOITER_UNLIM"	17	"Loiter around this waypoint an unlimited amount of time"
"MAV_CMD_NAV_LOITER_TURNS"	18	"Loiter around this waypoint for X turns"
"MAV_CMD_NAV_LOITER_TIME"	19	"Loiter around this waypoint for X seconds"
"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"
"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 only)"
"MAV_CMD_NAV_FOLLOW"	25	"Vehicle following, i.e. this waypoint represents a target"
"MAV_CMD_NAV_CONTINUE_AND_CHANGE_ALT"	30	"Continue on the current course and climb/descend to the specified altitude"
"MAV_CMD_NAV_LOITER_TO_ALT"	31	"Begin loiter at the specified Latitude and Longitude and then continue on the current course and climb/descend to the specified altitude"
"MAV_CMD_DO_FOLLOW"	32	"Begin following a target"
"MAV_CMD_DO_FOLLOW_REPOSITION"	33	"Reposition the MAV after a follow target is reached"
"MAV_CMD_DO_ORBIT"	34	"Start orbiting on the circumference of a circle with the specified radius around the specified location"
"MAV_CMD_NAV_ROI"	80	"Sets the region of interest (ROI) for a camera or vision system"
:		

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

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

info=1x4 table

MessageID	MessageName	
0	"HEARTBEAT"	"The heartbeat message shows that a system is present and responsive"

```
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.

See Also

[enuminfo](#) | [mavlinkclient](#) | [mavlinkdialect](#) | [mavlinkio](#) | [mavlinksub](#) | [msginfo](#) | [num2enum](#)

External Websites

[MAVLink Developer Guide](#)

Introduced in R2019a

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=133×3 table

Name	Value	
"MAV_CMD_NAV_WAYPOINT"	16	"Navigate to waypoint."
"MAV_CMD_NAV_LOITER_UNLIM"	17	"Loiter around this waypoint an unlimited amount of time"
"MAV_CMD_NAV_LOITER_TURNS"	18	"Loiter around this waypoint for X turns"
"MAV_CMD_NAV_LOITER_TIME"	19	"Loiter around this waypoint for X seconds"
"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"
"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 only)"
"MAV_CMD_NAV_FOLLOW"	25	"Vehicle following, i.e. this waypoint represents a target"
"MAV_CMD_NAV_CONTINUE_AND_CHANGE_ALT"	30	"Continue on the current course and climb/descend to the specified altitude"
"MAV_CMD_NAV_LOITER_TO_ALT"	31	"Begin loiter at the specified Latitude and Longitude and then continue on the current course and climb/descend to the specified altitude"
"MAV_CMD_DO_FOLLOW"	32	"Begin following a target"
"MAV_CMD_DO_FOLLOW_REPOSITION"	33	"Reposition the MAV after a follow target is reached"
"MAV_CMD_DO_ORBIT"	34	"Start orbiting on the circumference of a circle with the specified radius"
"MAV_CMD_NAV_ROI"	80	"Sets the region of interest (ROI) for a camera or vision sensor"
:		

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 is present and responsive"

```
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.

See Also

mavlinkclient | mavlinkdialect | mavlinkio | mavlinksub | msginfo

External Websites

MAVLink Developer Guide

Introduced in R2019a

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=133×3 table

Name	Value	
"MAV_CMD_NAV_WAYPOINT"	16	"Navigate to waypoint."
"MAV_CMD_NAV_LOITER_UNLIM"	17	"Loiter around this waypoint an unlimited amount of time"
"MAV_CMD_NAV_LOITER_TURNS"	18	"Loiter around this waypoint for X turns"
"MAV_CMD_NAV_LOITER_TIME"	19	"Loiter around this waypoint for X seconds"
"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"
"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 only)"
"MAV_CMD_NAV_FOLLOW"	25	"Vehicle following, i.e. this waypoint represents a target"
"MAV_CMD_NAV_CONTINUE_AND_CHANGE_ALT"	30	"Continue on the current course and climb/descend to the specified altitude"
"MAV_CMD_NAV_LOITER_TO_ALT"	31	"Begin loiter at the specified Latitude and Longitude and then climb/descend to the specified altitude"
"MAV_CMD_DO_FOLLOW"	32	"Begin following a target"
"MAV_CMD_DO_FOLLOW_REPOSITION"	33	"Reposition the MAV after a follow target has been reached"
"MAV_CMD_DO_ORBIT"	34	"Start orbiting on the circumference of a circle with the specified radius at the specified altitude"
"MAV_CMD_NAV_ROI"	80	"Sets the region of interest (ROI) for a camera or vision sensor"
:		

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 is present and responsive"

```
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.

See Also

`createmsg` | `enuminfo` | `mavlinkclient` | `mavlinkdialect` | `mavlinkio` | `mavlinksub`

External Websites

MAVLink Developer Guide

Introduced in R2019a

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

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.

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.

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=1x4 table
  SystemID  ComponentID  ComponentType  AutopilotType
  _____  _____  _____  _____
      255         1      "MAV_TYPE_GCS"  "MAV_AUTOPILOT_INVALID"
```

```
listConnections(mavlink)
```

```
ans=1x2 table
  ConnectionName  ConnectionInfo
  _____  _____
  "Connection1"  "UDP@0.0.0.0:51064"
```

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

Example: `'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

See Also

`disconnect` | `mavlinkclient` | `mavlinkdialect` | `mavlinksub`

Topics

"Tune UAV Parameters Using MAVLink Parameter Protocol"

External Websites

MAVLink Developer Guide

Introduced in R2019a

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.

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.

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=1x4 table  
  SystemID  ComponentID  ComponentType  AutopilotType  
-----  
      255         1  "MAV_TYPE_GCS"  "MAV_AUTOPILOT_INVALID"
```

```
listConnections(mavlink)
```

```
ans=1x2 table  
  ConnectionName  ConnectionInfo  
-----  
  "Connection1"  "UDP@0.0.0.0:51064"
```

```
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.

See Also

[connect](#) | [mavlinkclient](#) | [mavlinkdialect](#) | [mavlinkio](#) | [mavlinksub](#)

Topics

[“Tune UAV Parameters Using MAVLink Parameter Protocol”](#)

External Websites

[MAVLink Developer Guide](#)

Introduced in R2019a

listClients

List all connected MAVLink clients

Syntax

```
clientTable = listConnections(mavlink)
```

Description

`clientTable = 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.

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.

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=1x4 table
  SystemID  ComponentID  ComponentType  AutopilotType
  _____  _____  _____  _____
           255           1      "MAV_TYPE_GCS"  "MAV_AUTOPILOT_INVALID"
```

```
listConnections(mavlink)
```

```
ans=1x2 table
  ConnectionName  ConnectionInfo
  _____  _____
  "Connection1"  "UDP@0.0.0.0:51064"
```

```
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.

See Also

`connect` | `listConnections` | `listTopics` | `mavlinkclient` | `mavlinkdialect` | `mavlinkio` | `mavlinksub`

External Websites

MAVLink Developer Guide

Introduced in R2019a

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.

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.

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=1x4 table
  SystemID   ComponentID   ComponentType   AutopilotType
  _____   _____   _____   _____
          255           1           "MAV_TYPE_GCS"   "MAV_AUTOPILOT_INVALID"
```

```
listConnections(mavlink)
```

```
ans=1x2 table
  ConnectionName   ConnectionInfo
  _____   _____
  "Connection1"   "UDP@0.0.0.0:51064"
```



```
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.

See Also

`connect` | `listClients` | `listTopics` | `mavlinkclient` | `mavlinkdialect` | `mavlinkio` | `mavlinksub`

External Websites

MAVLink Developer Guide

Introduced in R2019a

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.

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.

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=1x4 table
  SystemID  ComponentID  ComponentType  AutopilotType
  _____  _____  _____  _____
           255           1      "MAV_TYPE_GCS"  "MAV_AUTOPILOT_INVALID"
```

```
listConnections(mavlink)
```

```
ans=1x2 table
  ConnectionName  ConnectionInfo
  _____  _____
  "Connection1"  "UDP@0.0.0.0:51064"
```

```
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.

See Also

`connect` | `listClients` | `listConnections` | `mavlinkclient` | `mavlinkdialect` | `mavlinkio` | `mavlinksub`

External Websites

MAVLink Developer Guide

Introduced in R2019a

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.

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.

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=1x4 table
```

SystemID	ComponentID	ComponentType	AutopilotType
255	1	"MAV_TYPE_GCS"	"MAV_AUTOPILOT_INVALID"

```
listConnections(mavlink)
```

```
ans=1x2 table
```

ConnectionName	ConnectionInfo
----------------	----------------

```

"Connection1"      "UDP@0.0.0.0:51064"

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.

See Also

`connect` | `listClients` | `listConnections` | `mavlinkclient` | `mavlinkdialect` | `mavlinkio` | `mavlinksub`

Topics

"Tune UAV Parameters Using MAVLink Parameter Protocol"

External Websites

MAVLink Developer Guide

Introduced in R2019a

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 message, returned as vector of `uint8` integers.

Data Types: `uint8`

See Also

`connect` | `listClients` | `listConnections` | `mavlinkclient` | `mavlinkdialect` | `mavlinkio` | `mavlinksub` | `sendmsg`

External Websites

MAVLink Developer Guide

Introduced in R2019a

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

See Also

`connect` | `listClients` | `listConnections` | `mavlinkclient` | `mavlinkdialect` | `mavlinkio` | `mavlinksub` | `sendmsg`

Topics

"Tune UAV Parameters Using MAVLink Parameter Protocol"

External Websites

MAVLink Developer Guide

Introduced in R2019a

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 structs.

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..

See Also

mavlinkclient | mavlinkdialect | mavlinkio | mavlinksub

Introduced in R2019a

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=133×3 table

Name	Value	
"MAV_CMD_NAV_WAYPOINT"	16	"Navigate to waypoint."
"MAV_CMD_NAV_LOITER_UNLIM"	17	"Loiter around this waypoint an unlimited amount of time"
"MAV_CMD_NAV_LOITER_TURNS"	18	"Loiter around this waypoint for X turns"
"MAV_CMD_NAV_LOITER_TIME"	19	"Loiter around this waypoint for X seconds"
"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"
"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 only)"
"MAV_CMD_NAV_FOLLOW"	25	"Vehicle following, i.e. this waypoint represents a target"
"MAV_CMD_NAV_CONTINUE_AND_CHANGE_ALT"	30	"Continue on the current course and climb/descend to the specified altitude"
"MAV_CMD_NAV_LOITER_TO_ALT"	31	"Begin loiter at the specified Latitude and Longitude and then continue on the current course and climb/descend to the specified altitude"
"MAV_CMD_DO_FOLLOW"	32	"Begin following a target"
"MAV_CMD_DO_FOLLOW_REPOSITION"	33	"Reposition the MAV after a follow target"
"MAV_CMD_DO_ORBIT"	34	"Start orbiting on the circumference of a circle with the specified radius"
"MAV_CMD_NAV_ROI"	80	"Sets the region of interest (ROI) for a camera"
:		

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 is present and responsive"

```
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.

See Also

[enum2num](#) | [enuminfo](#) | [mavlinkclient](#) | [mavlinkdialect](#) | [mavlinkio](#) | [mavlinksub](#) | [msginfo](#)

External Websites

[MAVLink Developer Guide](#)

Introduced in R2019a

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

This example shows how to load a MAVLink TLOG file and select a specific message type.

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

```
tlogReader = mavlinktlog('flight.tlog');
```

Read the 'REQUEST_DATA_STREAM' messages from the file.

```
msgData = readmsg(result,'MessageName','REQUEST_DATA_STREAM');
```

Input Arguments

tlogReader — MAVLink TLOG reader

mavlinktlog object

MAVLink TLOG reader, specified as a `mavlinktlog` object.

Name-Value Pair Arguments

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

Example: '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.

See Also

`mavlinkclient` | `mavlinkdialect` | `mavlinkio` | `mavlinktlog`

Topics

“Visualize and Playback MAVLink Flight Log”

Introduced in R2019a

deserializemsg

Deserialize MAVLink message from binary buffer

Syntax

```
msg = deserializemsg(dialect,buffer)
```

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.

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 message, specified as vector of `uint8` integers.

Data Types: `uint8`

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.

See Also

Functions

`createcmd` | `createmsg` | `enum2num` | `enuminfo` | `msginfo` | `num2enum`

Objects

`mavlinkclient` | `mavlinkdialect` | `mavlinkio` | `mavlinksub`

Introduced in R2019a

angvel

Angular velocity from quaternion array

Syntax

```
AV = angvel(Q,dt,'frame')
AV = angvel(Q,dt,'point')
[AV,qf] = angvel(Q,dt,fp,qi)
```

Description

`AV = angvel(Q,dt,'frame')` returns the angular velocity array from an array of quaternions, `Q`. The quaternions in `Q` correspond to frame rotation. The initial quaternion is assumed to represent zero rotation.

`AV = angvel(Q,dt,'point')` returns the angular velocity array from an array of quaternions, `Q`. The quaternions in `Q` correspond to point rotation. The initial quaternion is assumed to represent zero rotation.

`[AV,qf] = angvel(Q,dt,fp,qi)` allows you to specify the initial quaternion, `qi`, and the type of rotation, `fp`. It also returns the final quaternion, `qf`.

Examples

Generate Angular Velocity From Quaternion Array

Create an array of quaternions.

```
eulerAngles = [(0:10:90).',zeros(numel(0:10:90),2)];
q = quaternion(eulerAngles,'eulerd','ZYX','frame');
```

Specify the time step and generate the angular velocity array.

```
dt = 1;
av = angvel(q,dt,'frame') % units in rad/s
```

```
av = 10x3
```

```
    0         0         0
    0         0    0.1743
    0         0    0.1743
    0         0    0.1743
    0         0    0.1743
    0         0    0.1743
    0         0    0.1743
    0         0    0.1743
    0         0    0.1743
    0         0    0.1743
```

Input Arguments

Q — Quaternions

N-by-1 vector of quaternions

Quaternions, specified as an *N*-by-1 vector of quaternions.

Data Types: quaternion

dt — Time step

nonnegative scalar

Time step, specified as a nonnegative scalar.

Data Types: single | double

fp — Type of rotation

'frame' | 'point'

Type of rotation, specified as 'frame' or 'point'.

qi — Initial quaternion

quaternion

Initial quaternion, specified as a quaternion.

Data Types: quaternion

Output Arguments

AV — Angular velocity

N-by-3 real matrix

Angular velocity, returned as an *N*-by-3 real matrix. *N* is the number of quaternions given in the input *Q*. Each row of the matrix corresponds to an angular velocity vector.

qf — Final quaternion

quaternion

Final quaternion, returned as a quaternion. *qf* is the same as the last quaternion in the *Q* input.

Data Types: quaternion

Extended Capabilities

C/C++ Code Generation

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

See Also

quaternion

Introduced in R2020b

classUnderlying

Class of parts within quaternion

Syntax

```
underlyingClass = classUnderlying(quat)
```

Description

`underlyingClass = classUnderlying(quat)` returns the name of the class of the parts of the quaternion `quat`.

Examples

Get Underlying Class of Quaternion

A quaternion is a four-part hyper-complex number used in three-dimensional representations. The four parts of the quaternion are of data type `single` or `double`.

Create two quaternions, one with an underlying data type of `single`, and one with an underlying data type of `double`. Verify the underlying data types by calling `classUnderlying` on the quaternions.

```
qSingle = quaternion(single([1,2,3,4]))
```

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

```
classUnderlying(qSingle)
```

```
ans =  
'single'
```

```
qDouble = quaternion([1,2,3,4])
```

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

```
classUnderlying(qDouble)
```

```
ans =  
'double'
```

You can separate quaternions into their parts using the `parts` function. Verify the parts of each quaternion are the correct data type. Recall that `double` is the default MATLAB® type.

```
[aS,bS,cS,dS] = parts(qSingle)
```

```
aS = single  
1
```

```

bS = single
    2

cS = single
    3

dS = single
    4

[aD,bD,cD,dD] = parts(qDouble)

aD = 1

bD = 2

cD = 3

dD = 4

```

Quaternions follow the same implicit casting rules as other data types in MATLAB. That is, a quaternion with underlying data type `single` that is combined with a quaternion with underlying data type `double` results in a quaternion with underlying data type `single`. Multiply `qDouble` and `qSingle` and verify the resulting underlying data type is `single`.

```

q = qDouble*qSingle;
classUnderlying(q)

ans =
'single'

```

Input Arguments

quat — Quaternion to investigate

scalar | vector | matrix | multi-dimensional array

Quaternion to investigate, specified as a quaternion or array of quaternions.

Data Types: quaternion

Output Arguments

underlyingClass — Underlying class of quaternion object

'single' | 'double'

Underlying class of quaternion, returned as the character vector 'single' or 'double'.

Data Types: char

Extended Capabilities

C/C++ Code Generation

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

See Also

Functions

compact | parts

Objects

quaternion

Introduced in R2020b

compact

Convert quaternion array to N -by-4 matrix

Syntax

```
matrix = compact(quat)
```

Description

`matrix = compact(quat)` converts the quaternion array, `quat`, to an N -by-4 matrix. The columns are made from the four quaternion parts. The i^{th} row of the matrix corresponds to `quat(i)`.

Examples

Convert Quaternion Array to Compact Representation of Parts

Create a scalar quaternion with random parts. Convert the parts to a 1-by-4 vector using `compact`.

```
randomParts = randn(1,4)
randomParts = 1×4
    0.5377    1.8339   -2.2588    0.8622

quat = quaternion(randomParts)
quat = quaternion
    0.53767 + 1.8339i - 2.2588j + 0.86217k

quatParts = compact(quat)
quatParts = 1×4
    0.5377    1.8339   -2.2588    0.8622
```

Create a 2-by-2 array of quaternions, then convert the representation to a matrix of quaternion parts. The output rows correspond to the linear indices of the quaternion array.

```
quatArray = [quaternion([1:4;5:8]),quaternion([9:12;13:16])]
quatArray=2×2 quaternion array
    1 + 2i + 3j + 4k    9 + 10i + 11j + 12k
    5 + 6i + 7j + 8k    13 + 14i + 15j + 16k

quatArrayParts = compact(quatArray)
quatArrayParts = 4×4
```

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

Input Arguments

quat — Quaternion to convert

scalar | vector | matrix | multidimensional array

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

Data Types: quaternion

Output Arguments

matrix — Quaternion in matrix form

N-by-4 matrix

Quaternion in matrix form, returned as an *N*-by-4 matrix, where $N = \text{numel}(\text{quat})$.

Data Types: single | double

Extended Capabilities

C/C++ Code Generation

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

See Also

Functions

classUnderlying | parts

Objects

quaternion

Introduced in R2020b

conj

Complex conjugate of quaternion

Syntax

```
quatConjugate = conj(quat)
```

Description

`quatConjugate = conj(quat)` returns the complex conjugate of the quaternion, `quat`.

If $q = a + bi + cj + dk$, the complex conjugate of q is $q^* = a - bi - cj - dk$. Considered as a rotation operator, the conjugate performs the opposite rotation. For example,

```
q = quaternion(deg2rad([16 45 30]), 'rotvec');
a = q*conj(q);
rotatepoint(a,[0,1,0])
```

```
ans =
```

```
    0    1    0
```

Examples

Complex Conjugate of Quaternion

Create a quaternion scalar and get the complex conjugate.

```
q = normalize(quaternion([0.9 0.3 0.3 0.25]))
q = quaternion
    0.87727 + 0.29242i + 0.29242j + 0.24369k
```

```
qConj = conj(q)
```

```
qConj = quaternion
    0.87727 - 0.29242i - 0.29242j - 0.24369k
```

Verify that a quaternion multiplied by its conjugate returns a quaternion one.

```
q*qConj
```

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

Input Arguments

quat — Quaternion

scalar | vector | matrix | multidimensional array

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

Data Types: quaternion

Output Arguments

quatConjugate — Quaternion conjugate

scalar | vector | matrix | multidimensional array

Quaternion conjugate, returned as a quaternion or array of quaternions the same size as `quat`.

Data Types: quaternion

Extended Capabilities

C/C++ Code Generation

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

See Also

Functions

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

Objects

quaternion

Introduced in R2020b

ctranspose, '

Complex conjugate transpose of quaternion array

Syntax

```
quatTransposed = quat'
```

Description

`quatTransposed = quat'` returns the complex conjugate transpose of the quaternion, `quat`.

Examples

Vector Complex Conjugate Transpose

Create a vector of quaternions and compute its complex conjugate transpose.

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

```
quat=4x1 quaternion array
    0.53767 + 0.31877i + 3.5784j + 0.7254k
    1.8339 - 1.3077i + 2.7694j - 0.063055k
   -2.2588 - 0.43359i - 1.3499j + 0.71474k
    0.86217 + 0.34262i + 3.0349j - 0.20497k
```

```
quatTransposed = quat'
```

```
quatTransposed=1x4 quaternion array
    0.53767 - 0.31877i - 3.5784j - 0.7254k    1.8339 + 1.3077i - 2.7694j + 0.063055k
   -2.2588 + 0.43359i + 1.3499j - 0.71474k
    0.86217 - 0.34262i - 3.0349j + 0.20497k
```

Matrix Complex Conjugate Transpose

Create a matrix of quaternions and compute its complex conjugate transpose.

```
quat = [quaternion(randn(2,4)), quaternion(randn(2,4))]
```

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

```
quatTransposed = quat'
```

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

Input Arguments

quat — Quaternion to transpose

scalar | vector | matrix

Quaternion to transpose, specified as a vector or matrix or quaternions. The complex conjugate transpose is defined for 1-D and 2-D arrays.

Data Types: quaternion

Output Arguments

quatTransposed — Conjugate transposed quaternion

scalar | vector | matrix

Conjugate transposed quaternion, returned as an N -by- M array, where `quat` was specified as an M -by- N array.

Data Types: quaternion

Extended Capabilities

C/C++ Code Generation

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

See Also

Functions

`transpose`, `'`

Objects

quaternion

Introduced in R2020b

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 = 5x1
    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 $[A_{dim1}, \dots, A_{dimN}] = \text{size}(\text{quatA})$ and $[B_{dim1}, \dots, B_{dimN}] = \text{size}(\text{quatB})$, then for $i = 1:N$, either $A_{dim_i} == B_{dim_i}$ or $A_{dim_i} == 1$ or $B_{dim_i} == 1$.

If one of the quaternion arguments contains only one quaternion, then this function returns the distances between that quaternion and every quaternion in the other argument.

Data Types: quaternion

Output Arguments

distance — Angular distance (radians)

scalar | vector | matrix | multidimensional array

Angular distance in radians, returned as an array. The dimensions are the maximum of the union of $\text{size}(\text{quatA})$ and $\text{size}(\text{quatB})$.

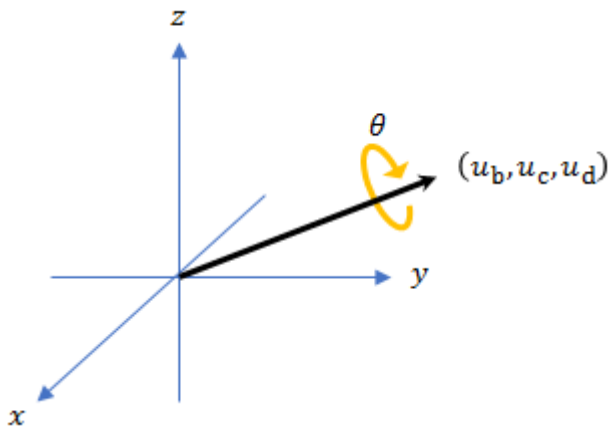
Data Types: single | double

Algorithms

The `dist` function returns the angular distance between two quaternions.

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

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



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

Consider two quaternions, p and q , and the product $z = p * \text{conjugate}(q)$. As p approaches q , the angle of z goes to 0, and z approaches the unit quaternion.

The angular distance between two quaternions can be expressed as $\theta_z = 2\cos^{-1}(\text{real}(z))$.

Using the quaternion data type syntax, the angular distance is calculated as:

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

Extended Capabilities

C/C++ Code Generation

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

See Also

Functions

conj | parts

Objects

quaternion

Introduced in R2020b

euler

Convert quaternion to Euler angles (radians)

Syntax

```
eulerAngles = euler(quat, rotationSequence, rotationType)
```

Description

`eulerAngles = euler(quat, rotationSequence, rotationType)` converts the quaternion, `quat`, to an N -by-3 matrix of Euler angles.

Examples

Convert Quaternion to Euler Angles in Radians

Convert a quaternion frame rotation to Euler angles in radians using the 'ZYX' rotation sequence.

```
quat = quaternion([0.7071 0.7071 0 0]);
eulerAnglesRadians = euler(quat, 'ZYX', 'frame')
```

```
eulerAnglesRadians = 1×3
    0         0    1.5708
```

Input Arguments

quat — Quaternion to convert to Euler angles

scalar | vector | matrix | multidimensional array

Quaternion to convert to Euler angles, specified as a scalar, vector, matrix, or multidimensional array of quaternions.

Data Types: quaternion

rotationSequence — Rotation sequence

'ZYX' | 'YZX' | 'ZYX' | 'ZYX' | 'ZYX' | 'ZYX' | 'ZYX' | 'ZYX' | 'ZYX' | 'ZYX'

Rotation sequence of Euler representation, specified as a character vector or string.

The rotation sequence defines the order of rotations about the axes. For example, if you specify a rotation sequence of 'YZX':

- 1 The first rotation is about the y-axis.
- 2 The second rotation is about the new z-axis.
- 3 The third rotation is about the new x-axis.

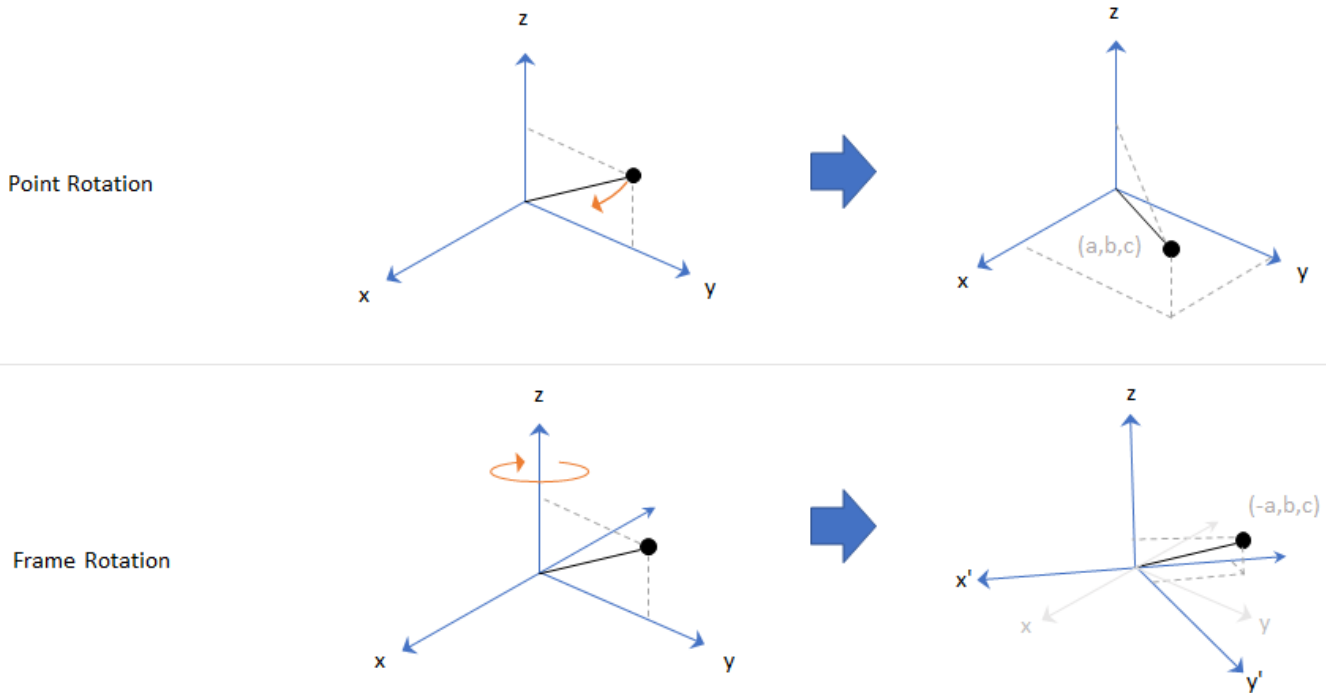
Data Types: char | string

rotationType — Type of rotation

'point' | 'frame'

Type of rotation, specified as 'point' or 'frame'.

In a point rotation, the frame is static and the point moves. In a frame rotation, the point is static and the frame moves. Point rotation and frame rotation define equivalent angular displacements but in opposite directions.



Data Types: char | string

Output Arguments**eulerAngles — Euler angle representation (radians)***N*-by-3 matrix

Euler angle representation in radians, returned as a *N*-by-3 matrix. *N* is the number of quaternions in the `quat` argument.

For each row of `eulerAngles`, the first element corresponds to the first axis in the rotation sequence, the second element corresponds to the second axis in the rotation sequence, and the third element corresponds to the third axis in the rotation sequence.

The data type of the Euler angles representation is the same as the underlying data type of `quat`.

Data Types: single | double

Extended Capabilities

C/C++ Code Generation

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

See Also

Functions

eulerd | rotateframe | rotatepoint

Objects

quaternion

Introduced in R2020b

eulerd

Convert quaternion to Euler angles (degrees)

Syntax

```
eulerAngles = eulerd(quat, rotationSequence, rotationType)
```

Description

`eulerAngles = eulerd(quat, rotationSequence, rotationType)` converts the quaternion, `quat`, to an N -by-3 matrix of Euler angles in degrees.

Examples

Convert Quaternion to Euler Angles in Degrees

Convert a quaternion frame rotation to Euler angles in degrees using the 'ZYX' rotation sequence.

```
quat = quaternion([0.7071 0.7071 0 0]);
eulerAnglesDegrees = eulerd(quat, 'ZYX', 'frame')
```

```
eulerAnglesDegrees = 1×3
```

```
    0         0    90.0000
```

Input Arguments

quat — Quaternion to convert to Euler angles

scalar | vector | matrix | multidimensional array

Quaternion to convert to Euler angles, specified as a scalar, vector, matrix, or multidimensional array of quaternions.

Data Types: quaternion

rotationSequence — Rotation sequence

'ZYX' | 'YZX' | 'ZYX' | 'ZYX' | 'ZYX' | 'ZYX' | 'ZYX' | 'ZYX' | 'ZYX' | 'ZYX'

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

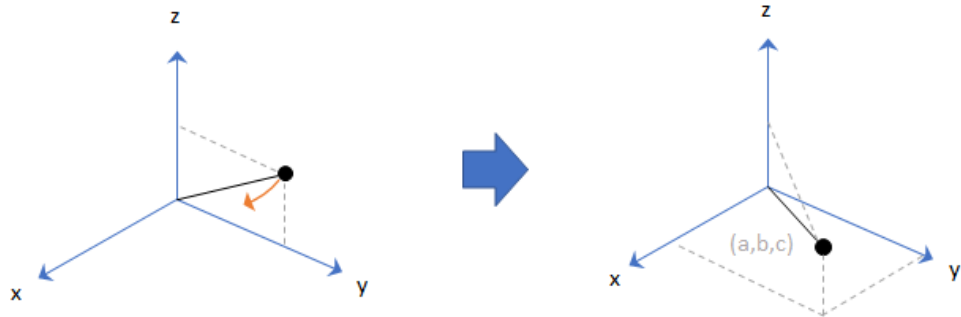
rotationType — Type of rotation

'point' | 'frame'

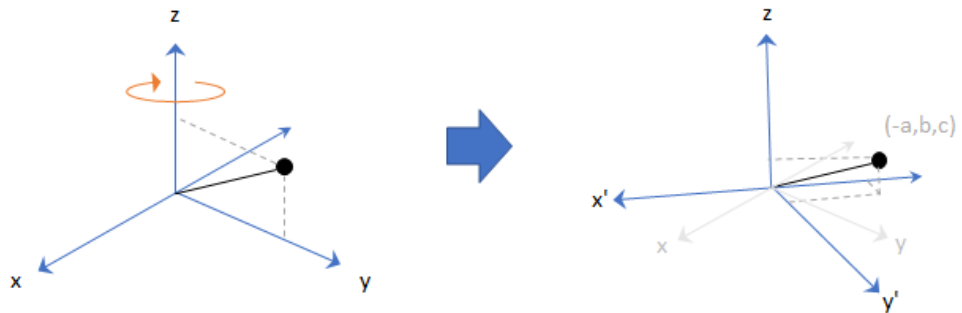
Type of rotation, specified as 'point' or 'frame'.

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.

Point Rotation



Frame Rotation



Data Types: char | string

Output Arguments**eulerAngles — Euler angle representation (degrees)***N*-by-3 matrix

Euler angle representation in degrees, returned as a *N*-by-3 matrix. *N* is the number of quaternions in the `quat` argument.

For each row of `eulerAngles`, the first column corresponds to the first axis in the rotation sequence, the second column corresponds to the second axis in the rotation sequence, and the third column corresponds to the third axis in the rotation sequence.

The data type of the Euler angles representation is the same as the underlying data type of `quat`.

Data Types: single | double

Extended Capabilities

C/C++ Code Generation

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

See Also

Functions

`euler` | `rotateframe` | `rotatepoint`

Objects

`quaternion`

Introduced in R2020b

exp

Exponential of quaternion array

Syntax

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

Description

$B = \text{exp}(A)$ computes the exponential of the elements of the quaternion array A .

Examples

Exponential of Quaternion Array

Create a 4-by-1 quaternion array A .

```
A = quaternion(magic(4))
```

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

Compute the exponential of A .

```
B = exp(A)
```

```
B=4×1 quaternion array
 5.3525e+06 + 1.0516e+06i + 1.5774e+06j + 6.8352e+06k
 -57.359 - 89.189i - 81.081j - 64.865k
 -6799.1 + 2039.1i + 1747.8j + 3495.6k
 -6.66 + 36.931i + 39.569j + 2.6379k
```

Input Arguments

A — Input quaternion

scalar | vector | matrix | multidimensional array

Input quaternion, specified as a scalar, vector, matrix, or multidimensional array.

Data Types: quaternion

Output Arguments

B — Result

scalar | vector | matrix | multidimensional array

Result of quaternion exponential, returned as a scalar, vector, matrix, or multidimensional array.

Data Types: quaternion

Algorithms

Given a quaternion $A = a + bi + cj + dk = a + \bar{v}$, the exponential is computed by

$$\exp(A) = e^a \left(\cos\|\bar{v}\| + \frac{\bar{v}}{\|\bar{v}\|} \sin\|\bar{v}\| \right)$$

Extended Capabilities

C/C++ Code Generation

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

See Also

Functions

log | power, .^

Objects

quaternion

Introduced in R2020b

ldivide, .\

Element-wise quaternion left division

Syntax

`C = A.\B`

Description

`C = A.\B` performs quaternion element-wise division by dividing each element of quaternion B by the corresponding element of quaternion A.

Examples

Divide a Quaternion Array by a Real Scalar

Create a 2-by-1 quaternion array, and divide it element-by-element by a real scalar.

```
A = quaternion([1:4;5:8])
```

A=2×1 quaternion array

```
1 + 2i + 3j + 4k
5 + 6i + 7j + 8k
```

```
B = 2;
```

```
C = A.\B
```

C=2×1 quaternion array

```
0.066667 - 0.133333i - 0.2j - 0.26667k
0.057471 - 0.068966i - 0.08046j - 0.091954k
```

Divide a Quaternion Array by Another Quaternion Array

Create a 2-by-2 quaternion array, and divide it element-by-element by another 2-by-2 quaternion array.

```
q1 = quaternion([1:4;2:5;4:7;5:8]);
```

```
A = reshape(q1,2,2)
```

A=2×2 quaternion array

```
1 + 2i + 3j + 4k    4 + 5i + 6j + 7k
2 + 3i + 4j + 5k    5 + 6i + 7j + 8k
```

```
q2 = quaternion(magic(4));
```

```
B = reshape(q2,2,2)
```

B=2×2 quaternion array

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

C = A.\B

C=2×2 quaternion array

```
2.7 - 1.9i - 0.9j - 1.7k      1.5159 - 0.37302i - 0.15079j - 0.0238
2.2778 + 0.46296i - 0.57407j + 0.092593k      1.2471 + 0.91379i - 0.33908j - 0.109
```

Input Arguments

A — Divisor

scalar | vector | matrix | multidimensional array

Divisor, specified as a quaternion, an array of quaternions, a real scalar, or an array of real numbers.

A and B must have compatible sizes. In the simplest cases, they can be the same size or one can be a scalar. Two inputs have compatible sizes if, for every dimension, the dimension sizes of the inputs are the same or one of the dimensions is 1.

Data Types: quaternion | single | double

B — Dividend

scalar | vector | matrix | multidimensional array

Dividend, specified as a quaternion, an array of quaternions, a real scalar, or an array of real numbers.

A and B must have compatible sizes. In the simplest cases, they can be the same size or one can be a scalar. Two inputs have compatible sizes if, for every dimension, the dimension sizes of the inputs are the same or one of the dimensions is 1.

Data Types: quaternion | single | double

Output Arguments

C — Result

scalar | vector | matrix | multidimensional array

Result of quaternion division, returned as a scalar, vector, matrix, or multidimensional array.

Data Types: quaternion

Algorithms

Quaternion Division

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

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

Note For a real scalar p , $A./p = A.\backslash p$.

Quaternion Division by a Quaternion Scalar

Given two quaternions A and B of compatible sizes, then

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

Extended Capabilities

C/C++ Code Generation

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

See Also

Functions

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

Objects

quaternion

Introduced in R2020b

log

Natural logarithm of quaternion array

Syntax

```
B = log(A)
```

Description

`B = log(A)` computes the natural logarithm of the elements of the quaternion array `A`.

Examples

Logarithmic Values of Quaternion Array

Create a 3-by-1 quaternion array `A`.

```
A = quaternion(randn(3,4))
```

```
A=3×1 quaternion array
    0.53767 + 0.86217i - 0.43359j + 2.7694k
    1.8339 + 0.31877i + 0.34262j - 1.3499k
   -2.2588 - 1.3077i + 3.5784j + 3.0349k
```

Compute the logarithmic values of `A`.

```
B = log(A)
```

```
B=3×1 quaternion array
    1.0925 + 0.40848i - 0.20543j + 1.3121k
    0.8436 + 0.14767i + 0.15872j - 0.62533k
    1.6807 - 0.53829i + 1.473j + 1.2493k
```

Input Arguments

A — Input array

scalar | vector | matrix | multidimensional array

Input array, specified as a scalar, vector, matrix, or multidimensional array.

Data Types: quaternion

Output Arguments

B — Logarithm values

scalar | vector | matrix | multidimensional array

Quaternion natural logarithm values, returned as a scalar, vector, matrix, or multidimensional array.

Data Types: quaternion

Algorithms

Given a quaternion $A = a + \bar{v} = a + bi + cj + dk$, the logarithm is computed by

$$\log(A) = \log\|A\| + \frac{\bar{v}}{\|\bar{v}\|} \arccos \frac{a}{\|A\|}$$

Extended Capabilities

C/C++ Code Generation

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

See Also

Functions

exp | power, .^

Objects

quaternion

Introduced in R2020b

meanrot

Quaternion mean rotation

Syntax

```
quatAverage = meanrot(quat)
quatAverage = meanrot(quat,dim)
quatAverage = meanrot( ____,nanflag)
```

Description

`quatAverage = meanrot(quat)` returns the average rotation of the elements of `quat` along the first array dimension whose size not does equal 1.

- If `quat` is a vector, `meanrot(quat)` returns the average rotation of the elements.
- If `quat` is a matrix, `meanrot(quat)` returns a row vector containing the average rotation of each column.
- If `quat` is a multidimensional array, then `meanrot(quat)` operates along the first array dimension whose size does not equal 1, treating the elements as vectors. This dimension becomes 1 while the sizes of all other dimensions remain the same.

The `meanrot` function normalizes the input quaternions, `quat`, before calculating the mean.

`quatAverage = meanrot(quat,dim)` return the average rotation along dimension `dim`. For example, if `quat` is a matrix, then `meanrot(quat,2)` is a column vector containing the mean of each row.

`quatAverage = meanrot(____,nanflag)` specifies whether to include or omit NaN values from the calculation for any of the previous syntaxes. `meanrot(quat,'includenan')` includes all NaN values in the calculation while `mean(quat,'omitnan')` ignores them.

Examples

Quaternion Mean Rotation

Create a matrix of quaternions corresponding to three sets of Euler angles.

```
eulerAngles = [40 20 10; ...
               50 10 5; ...
               45 70 1];
```

```
quat = quaternion(eulerAngles,'eulerd','ZYX','frame');
```

Determine the average rotation represented by the quaternions. Convert the average rotation to Euler angles in degrees for readability.

```
quatAverage = meanrot(quat)
```

```

quatAverage = quaternion
    0.88863 - 0.062598i + 0.27822j + 0.35918k

eulerAverage = eulerd(quatAverage, 'ZYX', 'frame')

eulerAverage = 1×3

    45.7876    32.6452    6.0407

```

Average Out Rotational Noise

Use `meanrot` over a sequence of quaternions to average out additive noise.

Create a vector of $1e6$ quaternions whose distance, as defined by the `dist` function, from `quaternion(1,0,0,0)` is normally distributed. Plot the Euler angles corresponding to the noisy quaternion vector.

```

nrows = 1e6;
ax = 2*rand(nrows,3) - 1;
ax = ax./sqrt(sum(ax.^2,2));
ang = 0.5*randn(size(ax,1),1);
q = quaternion(ax.*ang, 'rotvec');

noisyEulerAngles = eulerd(q, 'ZYX', 'frame');

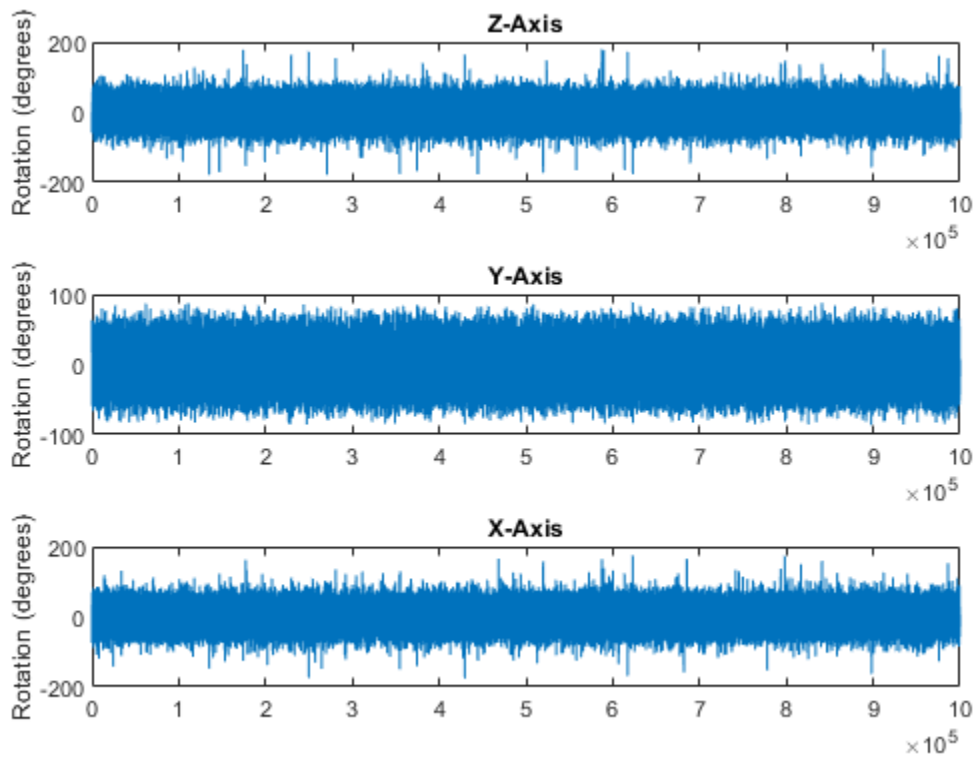
figure(1)

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

subplot(3,1,2)
plot(noisyEulerAngles(:,2))
title('Y-Axis')
ylabel('Rotation (degrees)')
hold on

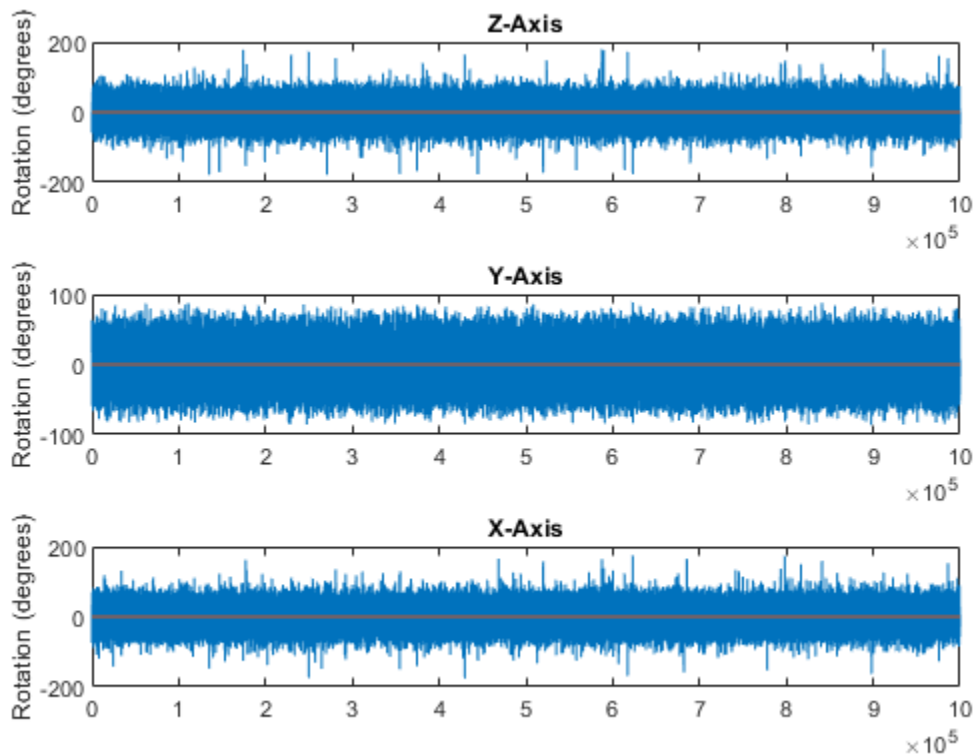
subplot(3,1,3)
plot(noisyEulerAngles(:,3))
title('X-Axis')
ylabel('Rotation (degrees)')
hold on

```



Use `meanrot` to determine the average quaternion given the vector of quaternions. Convert to Euler angles and plot the results.

```
qAverage = meanrot(q);  
qAverageInEulerAngles = eulerd(qAverage, 'ZYX', 'frame');  
figure(1)  
subplot(3,1,1)  
plot(ones(nrows,1)*qAverageInEulerAngles(:,1))  
title('Z-Axis')  
subplot(3,1,2)  
plot(ones(nrows,1)*qAverageInEulerAngles(:,2))  
title('Y-Axis')  
subplot(3,1,3)  
plot(ones(nrows,1)*qAverageInEulerAngles(:,3))  
title('X-Axis')
```

The meanrot Algorithm and Limitations

The meanrot Algorithm

The `meanrot` function outputs a quaternion that minimizes the squared Frobenius norm of the difference between rotation matrices. Consider two quaternions:

- `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');
```

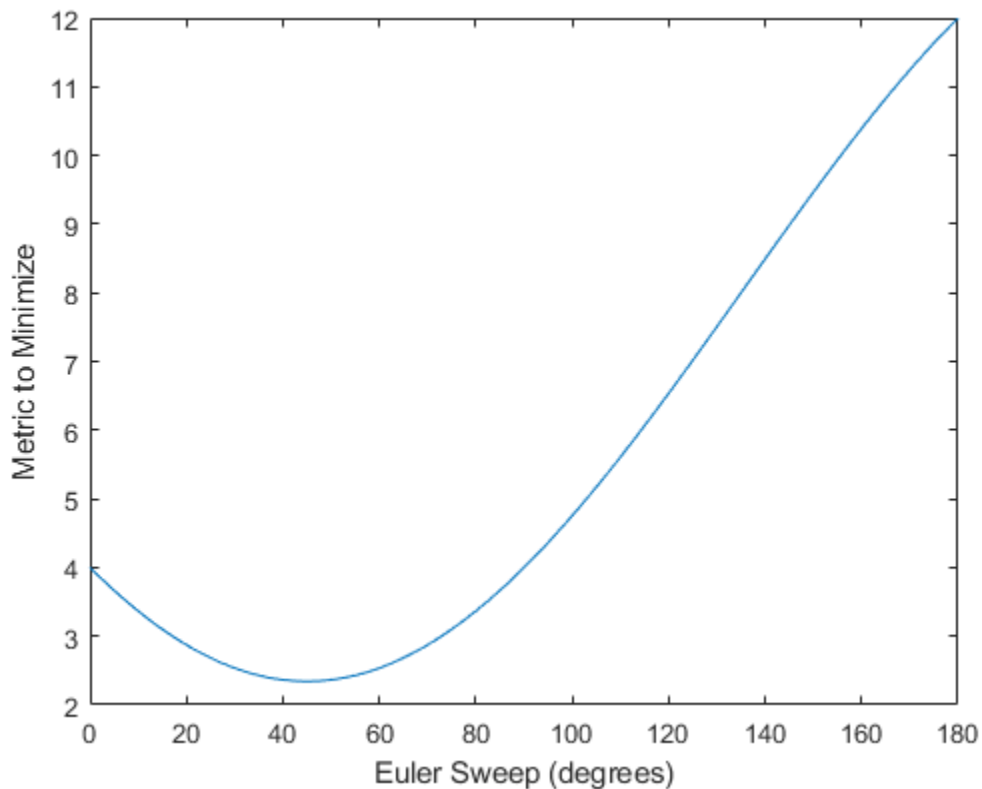
```

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 quaternion([0 0 0], 'ZYX', 'frame') and quaternion([0 0 90], 'ZYX', 'frame') as quaternion([0 0 45], 'ZYX', 'frame'). Call `meanrot` with `q0` and `q90` to verify the same result.

```
eulerd(meanrot([q0,q90]), 'ZYX', 'frame')
```

```
ans = 1x3
```

```
0      0  45.0000
```

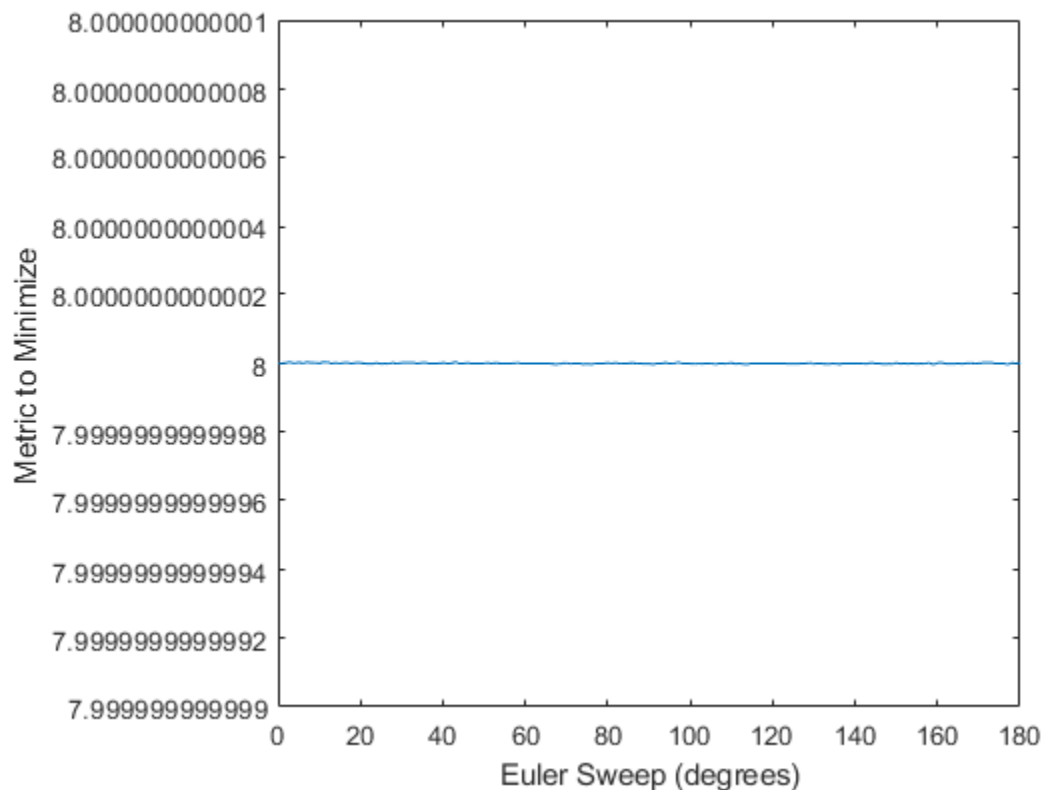
Limitations

The metric that meanrot uses to determine the mean rotation is not unique for quaternions significantly far apart. Repeat the experiment above for quaternions that are separated by 180 degrees.

```
q180 = quaternion([0 0 180], 'eulerd', 'ZYX', 'frame');
r180 = rotmat(q180, 'frame');

for i = 1:numel(qSweep)
    metricToMinimize(i) = norm((rSweep(:, :, i) - r0), 'fro').^2 + ...
        norm((rSweep(:, :, i) - r180), 'fro').^2;
end

plot(eulerSweep, metricToMinimize)
xlabel('Euler Sweep (degrees)')
ylabel('Metric to Minimize')
```



```
[~,eulerIndex] = min(metricToMinimize);
eulerSweep(eulerIndex)
```

```
ans = 159
```

Quaternion means are usually calculated for rotations that are close to each other, which makes the edge case shown in this example unlikely in real-world applications. To average two quaternions that are significantly far apart, use the `slerp` function. Repeat the experiment using `slerp` and verify that the quaternion mean returned is more intuitive for large distances.

```
qMean = slerp(q0,q180,0.5);
q0_q180 = eulerd(qMean,'ZYX','frame')

q0_q180 = 1×3
         0         0    90.0000
```

Input Arguments

quat — Quaternion

scalar | vector | matrix | multidimensional array

Quaternion for which to calculate the mean, specified as a scalar, vector, matrix, or multidimensional array of quaternions.

Data Types: quaternion

dim — Dimension to operate along

positive integer scalar

Dimension to operate along, specified as a positive integer scalar. If no value is specified, then the default is the first array dimension whose size does not equal 1.

Dimension `dim` indicates the dimension whose length reduces to 1. The `size(quatAverage,dim)` is 1, while the sizes of all other dimensions remain the same.

Data Types: double | single

nanflag — NaN condition

'includenan' (default) | 'omitnan'

NaN condition, specified as one of these values:

- 'includenan' -- Include NaN values when computing the mean rotation, resulting in NaN.
- 'omitnan' -- Ignore all NaN values in the input.

Data Types: char | string

Output Arguments

quatAverage — Quaternion average rotation

scalar | vector | matrix | multidimensional array

Quaternion average rotation, returned as a scalar, vector, matrix, or multidimensional array.

Data Types: single | double

Algorithms

meanrot determines a quaternion mean, \bar{q} , according to [1]. \bar{q} is the quaternion that minimizes the squared Frobenius norm of the difference between rotation matrices:

$$\bar{q} = \arg \min_{q \in S^3} \sum_{i=1}^n \|A(q) - A(q_i)\|_F^2$$

References

- [1] Markley, F. Landis, Yang Chen, John Lucas Crassidis, and Yaakov Oshman. "Average Quaternions." *Journal of Guidance, Control, and Dynamics*. Vol. 30, Issue 4, 2007, pp. 1193-1197.

Extended Capabilities

C/C++ Code Generation

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

See Also

Functions

dist | slerp

Objects

quaternion

Introduced in R2020b

minus, -

Quaternion subtraction

Syntax

$C = A - B$

Description

$C = A - B$ subtracts quaternion B from quaternion A using quaternion subtraction. Either A or B may be a real number, in which case subtraction is performed with the real part of the quaternion argument.

Examples

Subtract a Quaternion from a Quaternion

Quaternion subtraction is defined as the subtraction of the corresponding parts of each quaternion. Create two quaternions and perform subtraction.

```
Q1 = quaternion([1,0,-2,7]);  
Q2 = quaternion([1,2,3,4]);
```

```
Q1minusQ2 = Q1 - Q2
```

```
Q1minusQ2 = quaternion  
    0 - 2i - 5j + 3k
```

Subtract a Real Number from a Quaternion

Addition and subtraction of real numbers is defined for quaternions as acting on the real part of the quaternion. Create a quaternion and then subtract 1 from the real part.

```
Q = quaternion([1,1,1,1])
```

```
Q = quaternion  
    1 + 1i + 1j + 1k
```

```
Qminus1 = Q - 1
```

```
Qminus1 = quaternion  
    0 + 1i + 1j + 1k
```

Input Arguments

A — Input

scalar | vector | matrix | multidimensional array

Input, specified as a quaternion, array of quaternions, real number, or array of real numbers.

Data Types: quaternion | single | double

B — Input

scalar | vector | matrix | multidimensional array

Input, specified as a quaternion, array of quaternions, real number, or array of real numbers.

Data Types: quaternion | single | double

Output Arguments

C — Result

scalar | vector | matrix | multidimensional array

Result of quaternion subtraction, returned as a scalar, vector, matrix, or multidimensional array of quaternions.

Data Types: quaternion

Extended Capabilities

C/C++ Code Generation

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

See Also

Functions

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

Objects

quaternion

Introduced in R2020b

mtimes, *

Quaternion multiplication

Syntax

```
quatC = A*B
```

Description

`quatC = A*B` implements quaternion multiplication if either A or B is a quaternion. Either A or B must be a scalar.

You can use quaternion multiplication to compose rotation operators:

- To compose a sequence of frame rotations, multiply the quaternions in the order of the desired sequence of rotations. For example, to apply a p quaternion followed by a q quaternion, multiply in the order pq . The rotation operator becomes $(pq)^*v(pq)$, where v represents the object to rotate specified in quaternion form. $*$ represents conjugation.
- To compose a sequence of point rotations, multiply the quaternions in the reverse order of the desired sequence of rotations. For example, to apply a p quaternion followed by a q quaternion, multiply in the reverse order, qp . The rotation operator becomes $(qp)v(qp)^*$.

Examples

Multiply Quaternion Scalar and Quaternion Vector

Create a 4-by-1 column vector, A, and a scalar, b. Multiply A times b.

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

```
A=4x1 quaternion array
```

```

0.53767 + 0.31877i + 3.5784j + 0.7254k
1.8339 - 1.3077i + 2.7694j - 0.063055k
-2.2588 - 0.43359i - 1.3499j + 0.71474k
0.86217 + 0.34262i + 3.0349j - 0.20497k

```

```
b = quaternion(randn(1,4))
```

```
b = quaternion
```

```
-0.12414 + 1.4897i + 1.409j + 1.4172k
```

```
C = A*b
```

```
C=4x1 quaternion array
```

```

-6.6117 + 4.8105i + 0.94224j - 4.2097k
-2.0925 + 6.9079i + 3.9995j - 3.3614k
1.8155 - 6.2313i - 1.336j - 1.89k
-4.6033 + 5.8317i + 0.047161j - 2.791k

```


Input Arguments

A — Input

scalar | vector | matrix | multidimensional array

Input to multiply, specified as a quaternion, array of quaternions, real scalar, or array of real scalars.

If B is nonscalar, then A must be scalar.

Data Types: quaternion | single | double

B — Input

scalar | vector | matrix | multidimensional array

Input to multiply, specified as a quaternion, array of quaternions, real scalar, or array of real scalars.

If A is nonscalar, then B must be scalar.

Data Types: quaternion | single | double

Output Arguments

quatC — Quaternion product

scalar | vector | matrix | multidimensional array

Quaternion product, returned as a quaternion or array of quaternions.

Data Types: quaternion

Algorithms

Quaternion Multiplication by a Real Scalar

Given a quaternion

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

the product of q and a real scalar β is

$$\beta q = \beta a_q + \beta b_q i + \beta c_q j + \beta d_q k$$

Quaternion Multiplication by a Quaternion Scalar

The definition of the basis elements for quaternions,

$$i^2 = j^2 = k^2 = ijk = -1,$$

can be expanded to populate a table summarizing quaternion basis element multiplication:

	1	i	j	k
1	1	i	j	k
i	i	-1	k	-j

j	j	-k	-1	i
k	k	j	-i	-1

When reading the table, the rows are read first, for example: $ij = k$ and $ji = -k$.

Given two quaternions, $q = a_q + b_q i + c_q j + d_q k$, and $p = a_p + b_p i + c_p j + d_p k$, the multiplication can be expanded as:

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

You can simplify the equation using the quaternion multiplication table:

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

References

- [1] Kuipers, Jack B. *Quaternions and Rotation Sequences: A Primer with Applications to Orbits, Aerospace, and Virtual Reality*. Princeton, NJ: Princeton University Press, 2007.

Extended Capabilities

C/C++ Code Generation

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

See Also

Functions

times, .*

Objects

quaternion

Introduced in R2020b

norm

Quaternion norm

Syntax

`N = norm(quat)`

Description

`N = norm(quat)` returns the norm of the quaternion, `quat`.

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

Examples

Calculate Quaternion Norm

Create a scalar quaternion and calculate its norm.

```
quat = quaternion(1,2,3,4);
norm(quat)
```

```
ans = 5.4772
```

The quaternion norm is defined as the square root of the sum of the quaternion parts squared. Calculate the quaternion norm explicitly to verify the result of the `norm` function.

```
[a,b,c,d] = parts(quat);
sqrt(a^2+b^2+c^2+d^2)
```

```
ans = 5.4772
```

Input Arguments

quat — Quaternion

scalar | vector | matrix | multidimensional array

Quaternion for which to calculate the norm, specified as a scalar, vector, matrix, or multidimensional array of quaternions.

Data Types: quaternion

Output Arguments

N — Quaternion norm

scalar | vector | matrix | multidimensional array

Quaternion norm. If the input `quat` is an array, the output is returned as an array the same size as `quat`. Elements of the array are real numbers with the same data type as the underlying data type of the quaternion, `quat`.

Data Types: `single` | `double`

Extended Capabilities

C/C++ Code Generation

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

See Also

Functions

`conj` | `normalize` | `parts`

Objects

`quaternion`

Introduced in R2020b

normalize

Quaternion normalization

Syntax

```
quatNormalized = normalize(quat)
```

Description

`quatNormalized = normalize(quat)` normalizes the quaternion.

Given a quaternion of the form $Q = a + bi + cj + dk$, the normalized quaternion is defined as $Q/\sqrt{a^2 + b^2 + c^2 + d^2}$.

Examples

Normalize Elements of Quaternion Vector

Quaternions can represent rotations when normalized. You can use `normalize` to normalize a scalar, elements of a matrix, or elements of a multi-dimensional array of quaternions. Create a column vector of quaternions, then normalize them.

```
quatArray = quaternion([1,2,3,4; ...
                       2,3,4,1; ...
                       3,4,1,2]);
quatArrayNormalized = normalize(quatArray)

quatArrayNormalized=3x1 quaternion array
    0.18257 + 0.36515i + 0.54772j + 0.7303k
    0.36515 + 0.54772i + 0.7303j + 0.18257k
    0.54772 + 0.7303i + 0.18257j + 0.36515k
```

Input Arguments

quat — Quaternion to normalize

scalar | vector | matrix | multidimensional array

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

Data Types: quaternion

Output Arguments

quatNormalized — Normalized quaternion

scalar | vector | matrix | multidimensional array

Normalized quaternion, returned as a quaternion or array of quaternions the same size as `quat`.

Data Types: `quaternion`

Extended Capabilities

C/C++ Code Generation

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

See Also

Functions

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

Objects

`quaternion`

Introduced in R2020b

ones

Create quaternion array with real parts set to one and imaginary parts set to zero

Syntax

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

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

Description

`quat0nes = ones('quaternion')` returns a scalar quaternion with the real part set to 1 and the imaginary parts set to 0.

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

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

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

Example: `ones([1,4,2],'quaternion')` returns a 1-by-4-by-2 array of quaternions with the real parts set to 1 and the imaginary parts set to 0.

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

`quat0nes = ones(____, 'like', prototype, 'quaternion')` specifies the underlying class of the returned quaternion array to be the same as the underlying class of the quaternion prototype.

Examples

Quaternion Scalar One

Create a quaternion scalar one.

```
quat0nes = ones('quaternion')

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

Square Matrix of Quaternion Ones

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

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

quat0nes=3x3 quaternion array
    1 + 0i + 0j + 0k    1 + 0i + 0j + 0k    1 + 0i + 0j + 0k
    1 + 0i + 0j + 0k    1 + 0i + 0j + 0k    1 + 0i + 0j + 0k
    1 + 0i + 0j + 0k    1 + 0i + 0j + 0k    1 + 0i + 0j + 0k
```

Multidimensional Array of Quaternion Ones

Create a multidimensional array of quaternion ones by defining array dimensions in order. In this example, you create a 3-by-1-by-2 array. You can specify dimensions using a row vector or comma-separated integers. Specify the dimensions using a row vector and display the results:

```
dims = [3,1,2];
quat0nesSyntax1 = ones(dims, 'quaternion')
```

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

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

```
quat0nesSyntax1(:,:,2) =
```

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

Specify the dimensions using comma-separated integers, and then verify the equivalency of the two syntaxes:

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

```
ans = logical
     1
```

Underlying Class of Quaternion Ones

A quaternion is a four-part hyper-complex number used in three-dimensional rotations and orientations. You can specify the underlying data type of the parts as `single` or `double`. The default is `double`.

Create a quaternion array of ones with the underlying data type set to `single`.

```
quatOnes = ones(2,'like',single(1),'quaternion')
```

```
quatOnes=2x2 quaternion array
    1 + 0i + 0j + 0k    1 + 0i + 0j + 0k
    1 + 0i + 0j + 0k    1 + 0i + 0j + 0k
```

Verify the underlying class using the `classUnderlying` function.

```
classUnderlying(quatOnes)
```

```
ans =
'single'
```

Input Arguments

n — Size of square quaternion matrix

integer value

Size of square quaternion matrix, specified as an integer value.

If `n` is zero or negative, then `quatOnes` is returned as an empty matrix.

Example: `ones(4,'quaternion')` returns a 4-by-4 matrix of quaternions with the real parts set to 1 and the imaginary parts set to 0.

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

sz — Output size

row vector of integer values

Output size, specified as a row vector of integer values. Each element of `sz` indicates the size of the corresponding dimension in `quatOnes`. If the size of any dimension is 0 or negative, then `quatOnes` is returned as an empty array.

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

prototype — Quaternion prototype

variable

Quaternion prototype, specified as a variable.

Example: `ones(2,'like',quat,'quaternion')` returns a 2-by-2 matrix of quaternions with the same underlying class as the prototype quaternion, `quat`.

Data Types: `quaternion`

sz1, ..., szN — Size of each dimension

two or more integer values

Size of each dimension, specified as two or more integers. If the size of any dimension is 0 or negative, then `quatOnes` is returned as an empty array.

Example: `ones(2,3,'quaternion')` returns a 2-by-3 matrix of quaternions with the real parts set to 1 and the imaginary parts set to 0.

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

Output Arguments

quat0nes — Quaternion ones

`scalar` | `vector` | `matrix` | `multidimensional array`

Quaternion ones, returned as a scalar, vector, matrix, or multidimensional array of quaternions.

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

Data Types: `quaternion`

Extended Capabilities

C/C++ Code Generation

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

See Also

Functions

`zeros`

Objects

`quaternion`

Introduced in R2020b

parts

Extract quaternion parts

Syntax

```
[a,b,c,d] = parts(quat)
```

Description

`[a,b,c,d] = parts(quat)` returns the parts of the quaternion array as arrays, each the same size as `quat`.

Examples

Convert Quaternion to Matrix of Quaternion Parts

Convert a quaternion representation to parts using the `parts` function.

Create a two-element column vector of quaternions by specifying the parts.

```
quat = quaternion([1:4;5:8])
```

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

Recover the parts from the quaternion matrix using the `parts` function. The parts are returned as separate output arguments, each the same size as the input 2-by-1 column vector of quaternions.

```
[qA,qB,qC,qD] = parts(quat)
```

```
qA = 2×1
```

```
    1
    5
```

```
qB = 2×1
```

```
    2
    6
```

```
qC = 2×1
```

```
    3
    7
```

```
qD = 2×1
```

4
8

Input Arguments

quat — Quaternion

scalar | vector | matrix | multidimensional array

Quaternion, specified as a quaternion or array of quaternions.

Data Types: quaternion

Output Arguments

[a, b, c, d] — Quaternion parts

scalar | vector | matrix | multidimensional array

Quaternion parts, returned as four arrays: a, b, d, and d. Each part is the same size as `quat`.

Data Types: single | double

Extended Capabilities

C/C++ Code Generation

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

See Also

Functions

classUnderlying | compact

Objects

quaternion

Introduced in R2020b

power, .^

Element-wise quaternion power

Syntax

`C = A.^b`

Description

`C = A.^b` raises each element of `A` to the corresponding power in `b`.

Examples

Raise a Quaternion to a Real Scalar Power

Create a quaternion and raise it to a real scalar power.

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

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

```
b = 3;
C = A.^b
```

```
C = quaternion
   -86 - 52i - 78j - 104k
```

Raise a Quaternion Array to Powers from a Multidimensional Array

Create a 2-by-1 quaternion array and raise it to powers from a 2-D array.

```
A = quaternion([1:4;5:8])
```

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

```
b = [1 0 2; 3 2 1]
```

```
b = 2×3
```

```
    1    0    2
    3    2    1
```

```
C = A.^b
```

$C=2 \times 3$ quaternion array

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

Input Arguments

A — Base

scalar | vector | matrix | multidimensional array

Base, specified as a scalar, vector, matrix, or multidimensional array.

Data Types: quaternion | single | double

b — Exponent

scalar | vector | matrix | multidimensional array

Exponent, specified as a real scalar, vector, matrix, or multidimensional array.

Data Types: single | double

Output Arguments

C — Result

scalar | vector | matrix | multidimensional array

Each element of quaternion A raised to the corresponding power in b , returned as a scalar, vector, matrix, or multidimensional array.

Data Types: quaternion

Algorithms

The polar representation of a quaternion $A = a + bi + cj + dk$ is given by

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

where θ is the angle of rotation, and \hat{u} is the unit quaternion.

Quaternion A raised by a real exponent b is given by

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

Extended Capabilities

C/C++ Code Generation

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

See Also

Functions

exp | log

Objects

quaternion

Introduced in R2020b

prod

Product of a quaternion array

Syntax

```
quatProd = prod(quat)
quatProd = prod(quat,dim)
```

Description

`quatProd = prod(quat)` returns the quaternion product of the elements of the array.

`quatProd = prod(quat,dim)` calculates the quaternion product along dimension `dim`.

Examples

Product of Quaternions in Each Column

Create a 3-by-3 array whose elements correspond to their linear indices.

```
A = reshape(quaternion(randn(9,4)),3,3)
```

A=3×3 quaternion array

```
0.53767 + 2.7694i + 1.409j - 0.30344k    0.86217 + 0.7254i - 1.2075j + 0.888
1.8339 - 1.3499i + 1.4172j + 0.29387k    0.31877 - 0.063055i + 0.71724j - 1.147
-2.2588 + 3.0349i + 0.6715j - 0.78728k    -1.3077 + 0.71474i + 1.6302j - 1.068
```

Find the product of the quaternions in each column. The length of the first dimension is 1, and the length of the second dimension matches `size(A,2)`.

```
B = prod(A)
```

B=1×3 quaternion array

```
-19.837 - 9.1521i + 15.813j - 19.918k    -5.4708 - 0.28535i + 3.077j - 1.2295k
```

Product of Specified Dimension of Quaternion Array

You can specify which dimension of a quaternion array to take the product of.

Create a 2-by-2-by-2 quaternion array.

```
A = reshape(quaternion(randn(8,4)),2,2,2);
```

Find the product of the elements in each page of the array. The length of the first dimension matches `size(A,1)`, the length of the second dimension matches `size(A,2)`, and the length of the third dimension is 1.


```
dim = 3;
B = prod(A,dim)
```

```
B=2x2 quaternion array
   -2.4847 + 1.1659i - 0.37547j + 2.8068k    0.28786 - 0.29876i - 0.51231j - 4.2972k
   0.38986 - 3.6606i - 2.0474j - 6.047k    -1.741 - 0.26782i + 5.4346j + 4.1452k
```

Input Arguments

quat — Quaternion

scalar | vector | matrix | multidimensional array

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

Example: `qProd = prod(quat)` calculates the quaternion product along the first non-singleton dimension of `quat`.

Data Types: quaternion

dim — Dimension

first non-singleton dimension (default) | positive integer

Dimension along which to calculate the quaternion product, specified as a positive integer. If `dim` is not specified, `prod` operates along the first non-singleton dimension of `quat`.

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

Output Arguments

quatProd — Quaternion product

positive integer

Quaternion product, returned as quaternion array with one less non-singleton dimension than `quat`.

For example, if `quat` is a 2-by-2-by-5 array,

- `prod(quat,1)` returns a 1-by-2-by-5 array.
- `prod(quat,2)` returns a 2-by-1-by-5 array.
- `prod(quat,3)` returns a 2-by-2 array.

Data Types: quaternion

Extended Capabilities

C/C++ Code Generation

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

See Also

Functions

`mtimes, *` | `times, .*`

Objects

quaternion

Introduced in R2020b

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=2×2 quaternion array

```

1 + 2i + 3j + 4k      2 + 3i + 4j + 5k
3 + 4i + 5j + 6k      4 + 5i + 6j + 7k

```

C = A./B

C=2×2 quaternion array

```

2.7 - 0.1i - 2.1j - 1.7k      2.2778 + 0.092593i - 0.46296j - 0.5740k
1.8256 - 0.081395i + 0.45349j - 0.24419k      1.4524 - 0.5i + 1.0238j - 0.261k

```

Input Arguments

A — Dividend

scalar | vector | matrix | multidimensional array

Dividend, specified as a quaternion, an array of quaternions, a real scalar, or an array of real numbers.

A and B must have compatible sizes. In the simplest cases, they can be the same size or one can be a scalar. Two inputs have compatible sizes if, for every dimension, the dimension sizes of the inputs are the same or one of the dimensions is 1.

Data Types: quaternion | single | double

B — Divisor

scalar | vector | matrix | multidimensional array

Divisor, specified as a quaternion, an array of quaternions, a real scalar, or an array of real numbers.

A and B must have compatible sizes. In the simplest cases, they can be the same size or one can be a scalar. Two inputs have compatible sizes if, for every dimension, the dimension sizes of the inputs are the same or one of the dimensions is 1.

Data Types: quaternion | single | double

Output Arguments

C — Result

scalar | vector | matrix | multidimensional array

Result of quaternion division, returned as a scalar, vector, matrix, or multidimensional array.

Data Types: quaternion

Algorithms

Quaternion Division

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

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

Note For a real scalar p , $A./p = A.\backslash p$.

Quaternion Division by a Quaternion Scalar

Given two quaternions A and B of compatible sizes,

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

Extended Capabilities

C/C++ Code Generation

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

See Also

Functions

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

Objects

`quaternion`

Introduced in R2020b

randrot

Uniformly distributed random rotations

Syntax

```
R = randrot
R = randrot(m)
R = randrot(m1,...,mN)
R = randrot([m1,...,mN])
```

Description

`R = randrot` returns a unit quaternion drawn from a uniform distribution of random rotations.

`R = randrot(m)` returns an m -by- m matrix of unit quaternions drawn from a uniform distribution of random rotations.

`R = randrot(m1,...,mN)` returns an $m1$ -by-...-by- mN array of random unit quaternions, where $m1$, ..., mN indicate the size of each dimension. For example, `randrot(3,4)` returns a 3-by-4 matrix of random unit quaternions.

`R = randrot([m1,...,mN])` returns an $m1$ -by-...-by- mN array of random unit quaternions, where $m1$, ..., mN indicate the size of each dimension. For example, `randrot([3,4])` returns a 3-by-4 matrix of random unit quaternions.

Examples

Matrix of Random Rotations

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

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

```
r=3x3 quaternion array
```

```
0.17446 + 0.59506i - 0.73295j + 0.27976k    0.69704 - 0.060589i + 0.68679j - 0.1969
0.21908 - 0.89875i - 0.298j + 0.23548k    -0.049744 + 0.59691i + 0.56459j + 0.5678
0.6375 + 0.49338i - 0.24049j + 0.54068k    0.2979 - 0.53568i + 0.31819j + 0.7232
```

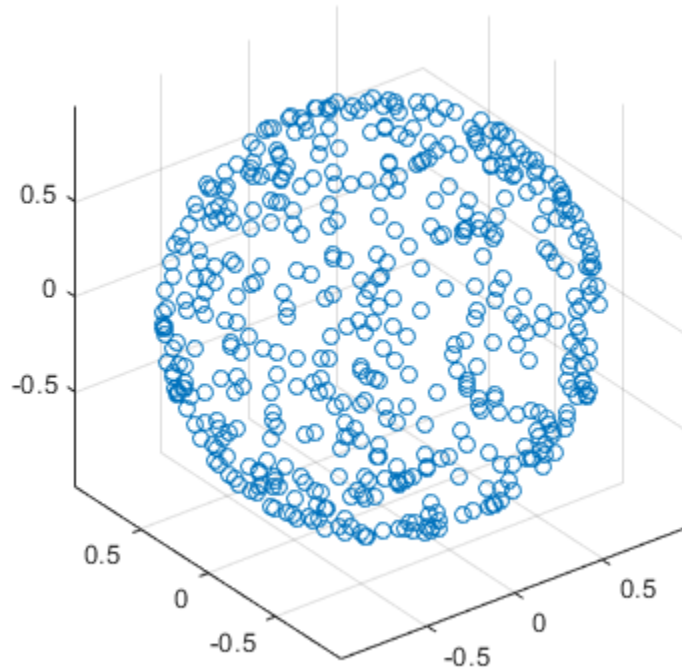
Create Uniform Distribution of Random Rotations

Create a vector of 500 random quaternions. Use `rotatepoint` on page 2-217 to visualize the distribution of the random rotations applied to point (1, 0, 0).

```
q = randrot(500,1);
```

```
pt = rotatepoint(q, [1 0 0]);
```

```
figure
scatter3(pt(:,1), pt(:,2), pt(:,3))
axis equal
```



Input Arguments

m — Size of square matrix

integer

Size of square quaternion matrix, specified as an integer value. If *m* is 0 or negative, then *R* is returned as an empty matrix.

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

m1, ..., mN — Size of each dimension

two or more integer values

Size of each dimension, specified as two or more integer values. If the size of any dimension is 0 or negative, then *R* is returned as an empty array.

Example: `randrot(2,3)` returns a 2-by-3 matrix of random quaternions.

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

[m1, ..., mN] — Vector of size of each dimension

row vector of integer values

Vector of size of each dimension, specified as a row vector of two or more integer values. If the size of any dimension is 0 or negative, then R is returned as an empty array.

Example: `randrot([2,3])` returns a 2-by-3 matrix of random quaternions.

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

Output Arguments

R — Random quaternions

`scalar` | `vector` | `matrix` | `multidimensional array`

Random quaternions, returned as a quaternion or array of quaternions.

Data Types: `quaternion`

References

[1] Shoemake, K. "Uniform Random Rotations." *Graphics Gems III* (K. David, ed.). New York: Academic Press, 1992.

Extended Capabilities

C/C++ Code Generation

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

See Also

`quaternion`

Introduced in R2020b

rotateframe

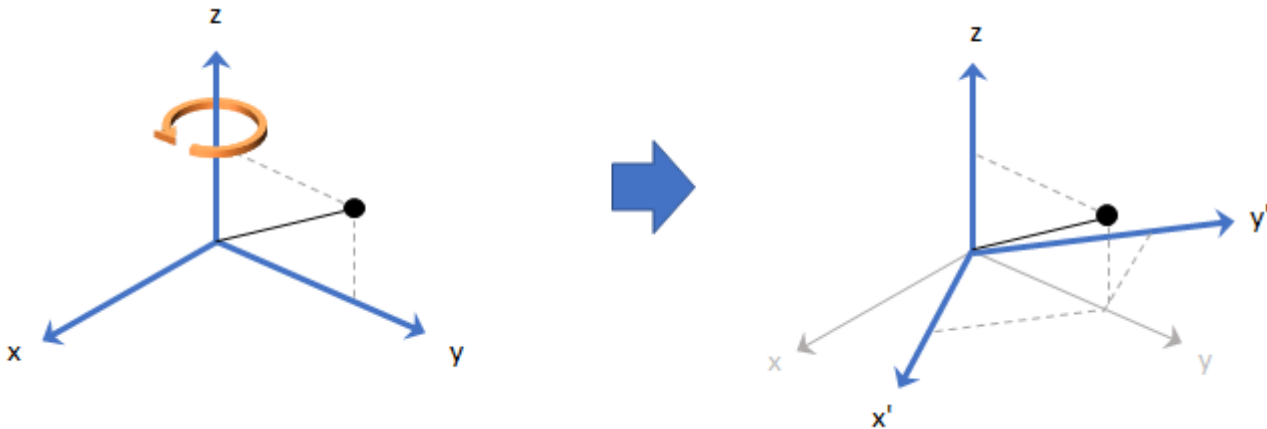
Quaternion frame rotation

Syntax

```
rotationResult = rotateframe(quat, cartesianPoints)
```

Description

`rotationResult = rotateframe(quat, cartesianPoints)` rotates the frame of reference for the Cartesian points using the quaternion, `quat`. The elements of the quaternion are normalized before use in the rotation.

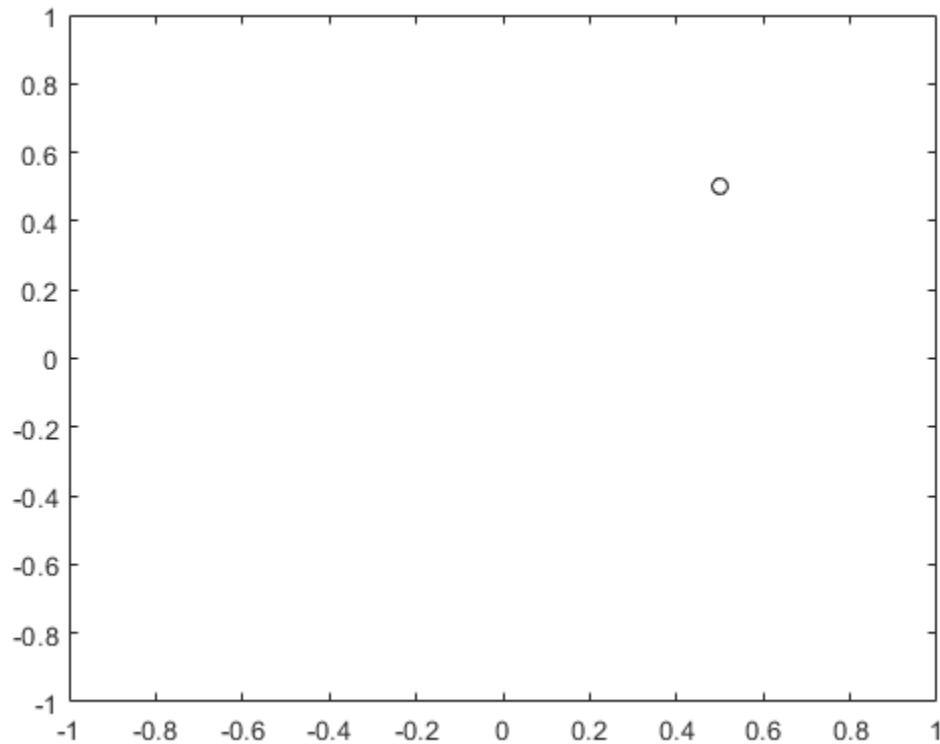


Examples

Rotate Frame Using Quaternion Vector

Define a point in three dimensions. The coordinates of a point are always specified in the order x , y , and z . For convenient visualization, define the point on the x - y plane.

```
x = 0.5;
y = 0.5;
z = 0;
plot(x,y, 'ko')
hold on
axis([-1 1 -1 1])
```



Create a quaternion vector specifying two separate rotations, one to rotate the frame 45 degrees and another to rotate the point -90 degrees about the z-axis. Use `rotateframe` to perform the rotations.

```
quat = quaternion([0,0,pi/4; ...
                  0,0,-pi/2], 'euler', 'XYZ', 'frame');
```

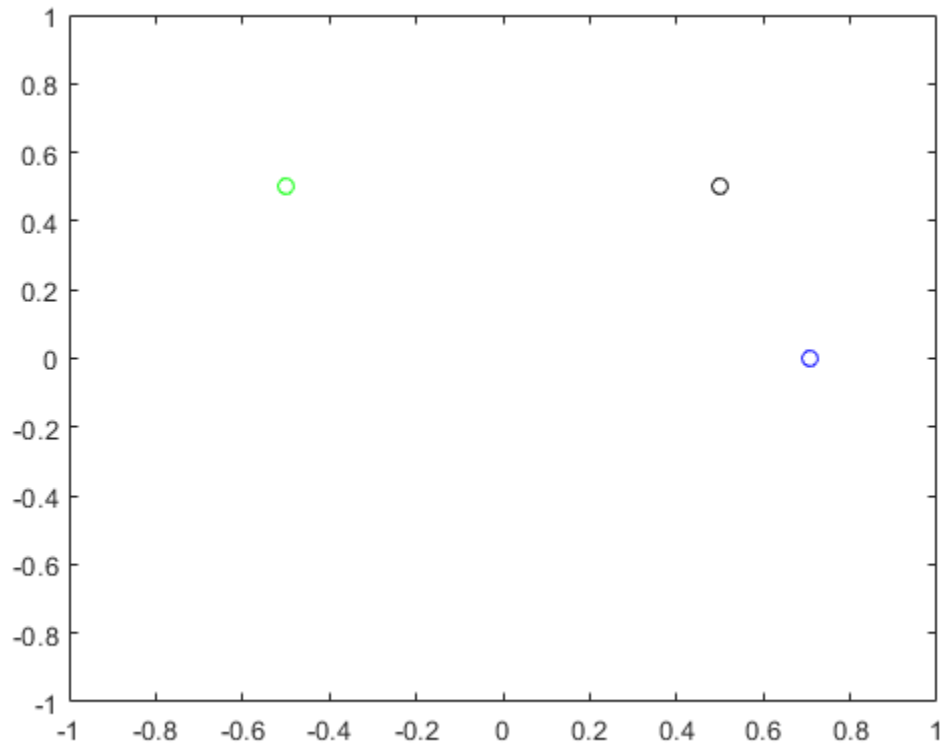
```
rereferencedPoint = rotateframe(quat, [x,y,z])
```

```
rereferencedPoint = 2x3
```

```
    0.7071    -0.0000     0
   -0.5000     0.5000     0
```

Plot the rereferenced points.

```
plot(rereferencedPoint(1,1),rereferencedPoint(1,2),'bo')
plot(rereferencedPoint(2,1),rereferencedPoint(2,2),'go')
```



Rereference Group of Points using Quaternion

Define two points in three-dimensional space. Define a quaternion to rereference the points by first rotating the reference frame about the z-axis 30 degrees and then about the new y-axis 45 degrees.

```
a = [1,0,0];
b = [0,1,0];
quat = quaternion([30,45,0], 'eulerd', 'ZYX', 'point');
```

Use rotateframe to reference both points using the quaternion rotation operator. Display the result.

```
rP = rotateframe(quat, [a;b])
rP = 2x3
    0.6124    -0.3536    0.7071
    0.5000    0.8660    -0.0000
```

Visualize the original orientation and the rotated orientation of the points. Draw lines from the origin to each of the points for visualization purposes.

```
plot3(a(1),a(2),a(3), 'bo');
hold on
```

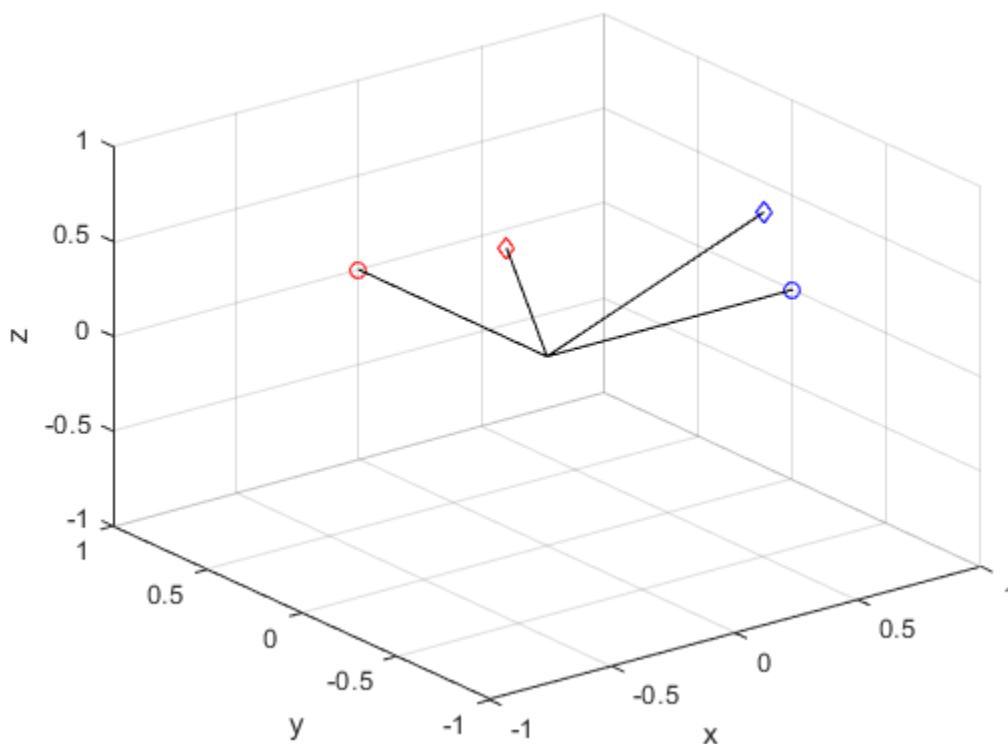
```

grid on
axis([-1 1 -1 1 -1 1])
xlabel('x')
ylabel('y')
zlabel('z')

plot3(b(1),b(2),b(3), 'ro');
plot3(rP(1,1),rP(1,2),rP(1,3), 'bd')
plot3(rP(2,1),rP(2,2),rP(2,3), 'rd')

plot3([0;rP(1,1)],[0;rP(1,2)],[0;rP(1,3)], 'k')
plot3([0;rP(2,1)],[0;rP(2,2)],[0;rP(2,3)], 'k')
plot3([0;a(1)],[0;a(2)],[0;a(3)], 'k')
plot3([0;b(1)],[0;b(2)],[0;b(3)], 'k')

```



Input Arguments

quat — Quaternion that defines rotation

scalar | vector

Quaternion that defines rotation, specified as a scalar quaternion or vector of quaternions.

Data Types: quaternion

cartesianPoints — Three-dimensional Cartesian points

1-by-3 vector | N -by-3 matrix

Three-dimensional Cartesian points, specified as a 1-by-3 vector or N -by-3 matrix.

Data Types: `single` | `double`

Output Arguments

rotationResult — Re-referenced Cartesian points

vector | matrix

Cartesian points defined in reference to rotated reference frame, returned as a vector or matrix the same size as `cartesianPoints`.

The data type of the re-referenced Cartesian points is the same as the underlying data type of `quat`.

Data Types: `single` | `double`

Algorithms

Quaternion frame rotation re-references a point specified in \mathbf{R}^3 by rotating the original frame of reference according to a specified quaternion:

$$L_q(u) = q*uq$$

where q is the quaternion, $*$ represents conjugation, and u is the point to rotate, specified as a quaternion.

For convenience, the `rotateframe` function takes a point in \mathbf{R}^3 and returns a point in \mathbf{R}^3 . Given a function call with some arbitrary quaternion, $q = a + bi + cj + dk$, and arbitrary coordinate, $[x,y,z]$,

```
point = [x,y,z];
rereferencedPoint = rotateframe(q,point)
```

the `rotateframe` function performs the following operations:

- 1 Converts point $[x,y,z]$ to a quaternion:

$$u_q = 0 + xi + yj + zk$$

- 2 Normalizes the quaternion, q :

$$q_n = \frac{q}{\sqrt{a^2 + b^2 + c^2 + d^2}}$$

- 3 Applies the rotation:

$$v_q = q*u_qq$$

- 4 Converts the quaternion output, v_q , back to \mathbf{R}^3

Extended Capabilities

C/C++ Code Generation

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

See Also

Functions

rotatepoint

Objects

quaternion

Introduced in R2020b

rotatepoint

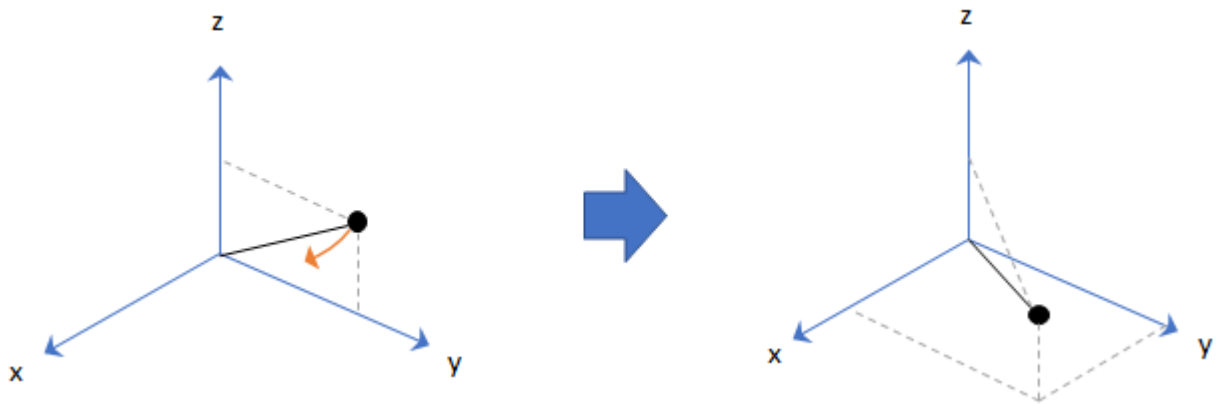
Quaternion point rotation

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.

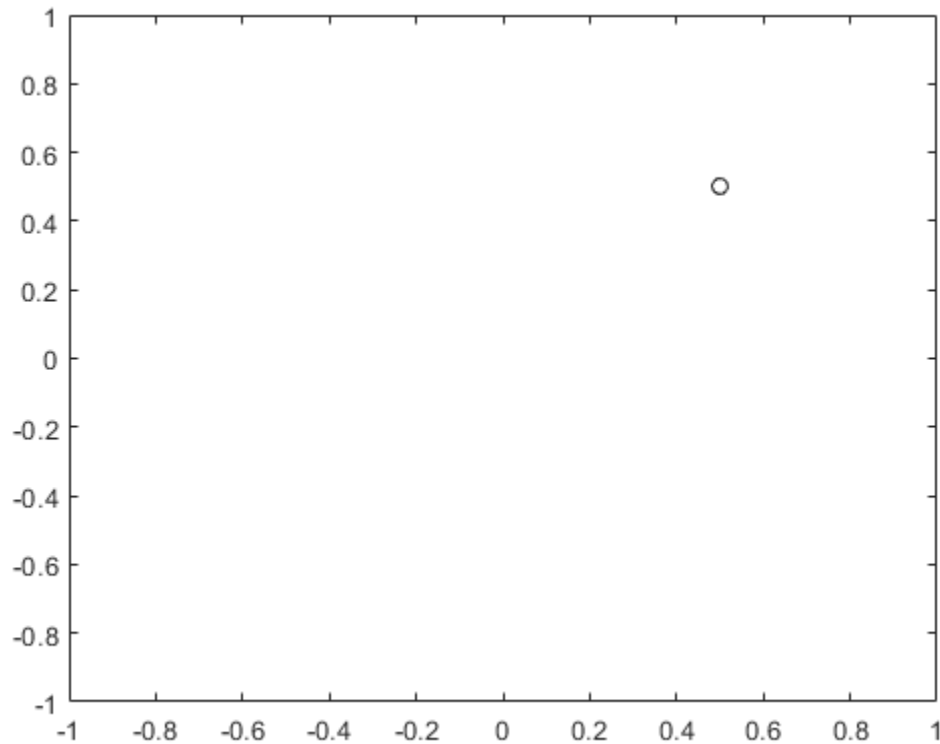


Examples

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');
```

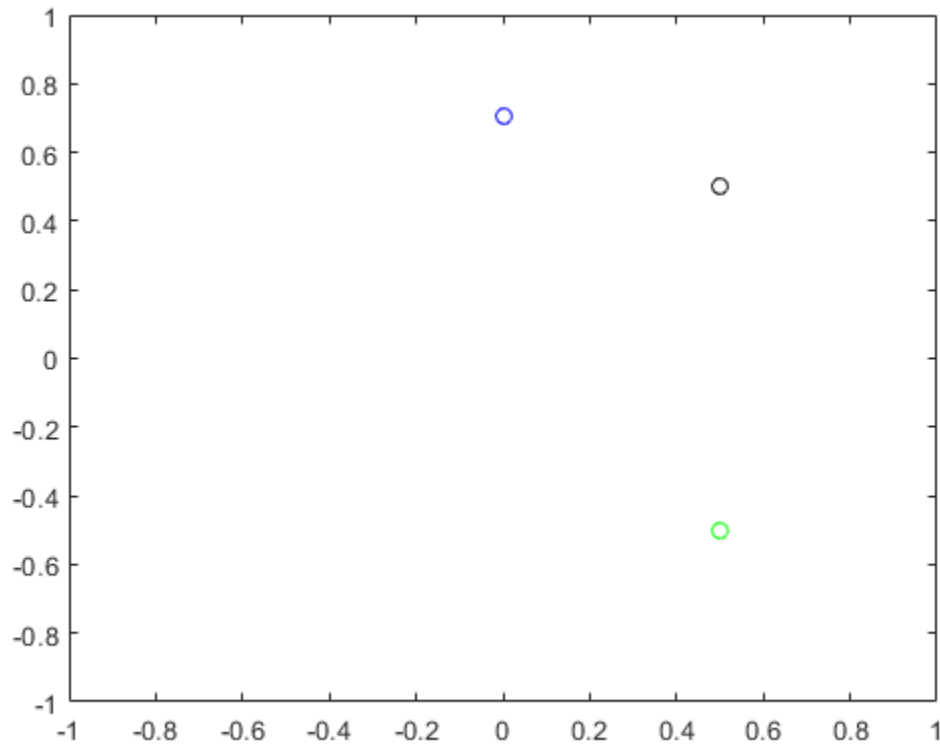
```
rotatedPoint = rotatepoint(quat,[x,y,z])
```

```
rotatedPoint = 2x3
```

```
-0.0000    0.7071    0
 0.5000   -0.5000    0
```

Plot the rotated points.

```
plot(rotatedPoint(1,1),rotatedPoint(1,2), 'bo')
plot(rotatedPoint(2,1),rotatedPoint(2,2), 'go')
```

Rotate Group of Points Using Quaternion

Define two points in three-dimensional space. Define a quaternion to rotate the point by first rotating about the z-axis 30 degrees and then about the new y-axis 45 degrees.

```
a = [1,0,0];
b = [0,1,0];
quat = quaternion([30,45,0], 'eulerd', 'ZYX', 'point');
```

Use rotatepoint to rotate both points using the quaternion rotation operator. Display the result.

```
rP = rotatepoint(quat,[a;b])
rP = 2×3
    0.6124    0.5000   -0.6124
   -0.3536    0.8660    0.3536
```

Visualize the original orientation and the rotated orientation of the points. Draw lines from the origin to each of the points for visualization purposes.

```
plot3(a(1),a(2),a(3), 'bo');
hold on
```

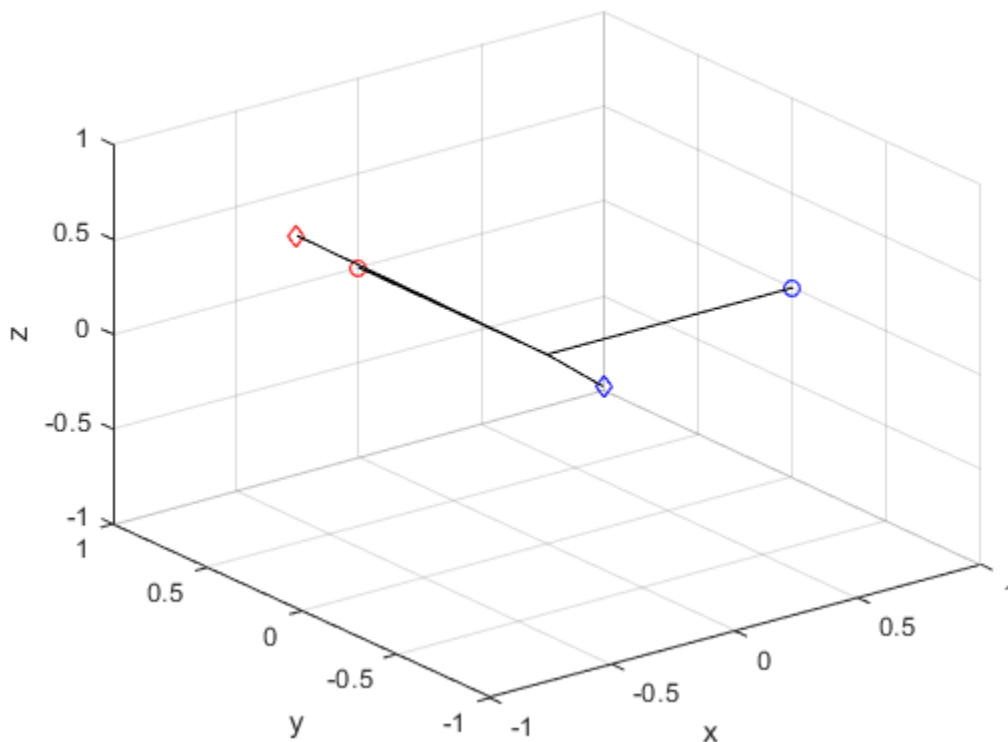
```

grid on
axis([-1 1 -1 1 -1 1])
xlabel('x')
ylabel('y')
zlabel('z')

plot3(b(1),b(2),b(3), 'ro');
plot3(rP(1,1),rP(1,2),rP(1,3), 'bd')
plot3(rP(2,1),rP(2,2),rP(2,3), 'rd')

plot3([0;rP(1,1)],[0;rP(1,2)],[0;rP(1,3)], 'k')
plot3([0;rP(2,1)],[0;rP(2,2)],[0;rP(2,3)], 'k')
plot3([0;a(1)],[0;a(2)],[0;a(3)], 'k')
plot3([0;b(1)],[0;b(2)],[0;b(3)], 'k')

```



Input Arguments

quat — Quaternion that defines rotation

scalar | vector

Quaternion that defines rotation, specified as a scalar quaternion, row vector of quaternions, or column vector of quaternions.

Data Types: quaternion

cartesianPoints — Three-dimensional Cartesian points1-by-3 vector | N -by-3 matrixThree-dimensional Cartesian points, specified as a 1-by-3 vector or N -by-3 matrix.Data Types: `single` | `double`**Output Arguments****rotationResult — Repositioned Cartesian points**

vector | matrix

Rotated Cartesian points defined using the quaternion rotation, returned as a vector or matrix the same size as `cartesianPoints`.Data Types: `single` | `double`**Algorithms**Quaternion point rotation rotates a point specified in \mathbf{R}^3 according to a specified quaternion:

$$L_q(u) = quq^*$$

where q is the quaternion, $*$ represents conjugation, and u is the point to rotate, specified as a quaternion.For convenience, the `rotatepoint` function takes in a point in \mathbf{R}^3 and returns a point in \mathbf{R}^3 . Given a function call with some arbitrary quaternion, $q = a + bi + cj + dk$, and arbitrary coordinate, $[x,y,z]$, for example,

```
rereferencedPoint = rotatepoint(q,[x,y,z])
```

the `rotatepoint` function performs the following operations:

- 1 Converts point $[x,y,z]$ to a quaternion:

$$u_q = 0 + xi + yj + zk$$

- 2 Normalizes the quaternion, q :

$$q_n = \frac{q}{\sqrt{a^2 + b^2 + c^2 + d^2}}$$

- 3 Applies the rotation:

$$v_q = qu_qq^*$$

- 4 Converts the quaternion output, v_q , back to \mathbf{R}^3

Extended Capabilities**C/C++ Code Generation**

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

See Also

Functions

rotateframe

Objects

quaternion

Introduced in R2020b

rotmat

Convert quaternion to rotation matrix

Syntax

```
rotationMatrix = rotmat(quat,rotationType)
```

Description

`rotationMatrix = rotmat(quat,rotationType)` converts the quaternion, `quat`, to an equivalent rotation matrix representation.

Examples

Convert Quaternion to Rotation Matrix for Point Rotation

Define a quaternion for use in point rotation.

```
theta = 45;
gamma = 30;
quat = quaternion([0,theta,gamma], 'eulerd', 'ZYX', 'point')

quat = quaternion
      0.8924 + 0.23912i + 0.36964j + 0.099046k
```

Convert the quaternion to a rotation matrix.

```
rotationMatrix = rotmat(quat, 'point')

rotationMatrix = 3×3

    0.7071    -0.0000    0.7071
    0.3536    0.8660   -0.3536
   -0.6124    0.5000    0.6124
```

To verify the rotation matrix, directly create two rotation matrices corresponding to the rotations about the *y*- and *x*-axes. Multiply the rotation matrices and compare to the output of `rotmat`.

```
theta = 45;
gamma = 30;

ry = [cosd(theta)  0          sind(theta) ; ...
      0            1          0           ; ...
      -sind(theta) 0          cosd(theta)];

rx = [1           0          0          ; ...
      0           cosd(gamma) -sind(gamma) ; ...
      0           sind(gamma) cosd(gamma)];

rotationMatrixVerification = rx*ry
```

```
rotationMatrixVerification = 3×3

    0.7071         0    0.7071
    0.3536    0.8660   -0.3536
   -0.6124    0.5000    0.6124
```

Convert Quaternion to Rotation Matrix for Frame Rotation

Define a quaternion for use in frame rotation.

```
theta = 45;
gamma = 30;
quat = quaternion([0,theta,gamma], 'eulerd', 'ZYX', 'frame')

quat = quaternion
    0.8924 + 0.23912i + 0.36964j - 0.099046k
```

Convert the quaternion to a rotation matrix.

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

rotationMatrix = 3×3

    0.7071   -0.0000   -0.7071
    0.3536    0.8660    0.3536
    0.6124   -0.5000    0.6124
```

To verify the rotation matrix, directly create two rotation matrices corresponding to the rotations about the y- and x-axes. Multiply the rotation matrices and compare to the output of rotmat.

```
theta = 45;
gamma = 30;

ry = [cosd(theta)  0          -sind(theta) ; ...
      0            1           0          ; ...
      sind(theta)  0          cosd(theta)];

rx = [1           0           0           ; ...
      0           cosd(gamma) sind(gamma) ; ...
      0           -sind(gamma) cosd(gamma)];

rotationMatrixVerification = rx*ry

rotationMatrixVerification = 3×3

    0.7071         0   -0.7071
    0.3536    0.8660    0.3536
    0.6124   -0.5000    0.6124
```

Convert Quaternion Vector to Rotation Matrices

Create a 3-by-1 normalized quaternion vector.

```
qVec = normalize( quaternion( randn(3,4) ) );
```

Convert the quaternion array to rotation matrices. The pages of `rotmatArray` correspond to the linear index of `qVec`.

```
rotmatArray = rotmat( qVec, 'frame' );
```

Assume `qVec` and `rotmatArray` correspond to a sequence of rotations. Combine the quaternion rotations into a single representation, then apply the quaternion rotation to arbitrarily initialized Cartesian points.

```
loc = normalize( randn(1,3) );
quat = prod( qVec );
rotateframe( quat, loc)
```

```
ans = 1×3
```

```
    0.9524    0.5297    0.9013
```

Combine the rotation matrices into a single representation, then apply the rotation matrix to the same initial Cartesian points. Verify the quaternion rotation and rotation matrix result in the same orientation.

```
totalRotMat = eye(3);
for i = 1:size( rotmatArray, 3 )
    totalRotMat = rotmatArray( :, :, i ) * totalRotMat;
end
totalRotMat * loc'
```

```
ans = 3×1
```

```
    0.9524
    0.5297
    0.9013
```

Input Arguments

quat — Quaternion to convert

scalar | vector | matrix | multidimensional array

Quaternion to convert, specified as a scalar, vector, matrix, or multidimensional array.

Data Types: quaternion

rotationType — Type or rotation

'frame' | 'point'

Type of rotation represented by the `rotationMatrix` output, specified as 'frame' or 'point'.

Data Types: char | string

Output Arguments

rotationMatrix — Rotation matrix representation

3-by-3 matrix | 3-by-3-by-*N* multidimensional array

Rotation matrix representation, returned as a 3-by-3 matrix or 3-by-3-by-*N* multidimensional array.

- If `quat` is a scalar, `rotationMatrix` is returned as a 3-by-3 matrix.
- If `quat` is non-scalar, `rotationMatrix` is returned as a 3-by-3-by-*N* multidimensional array, where `rotationMatrix(:, :, i)` is the rotation matrix corresponding to `quat(i)`.

The data type of the rotation matrix is the same as the underlying data type of `quat`.

Data Types: `single` | `double`

Algorithms

Given a quaternion of the form

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

the equivalent rotation matrix for frame rotation is defined as

$$\begin{bmatrix} 2a^2 - 1 + 2b^2 & 2bc + 2ad & 2bd - 2ac \\ 2bc - 2ad & 2a^2 - 1 + 2c^2 & 2cd + 2ab \\ 2bd + 2ac & 2cd - 2ab & 2a^2 - 1 + 2d^2 \end{bmatrix}.$$

The equivalent rotation matrix for point rotation is the transpose of the frame rotation matrix:

$$\begin{bmatrix} 2a^2 - 1 + 2b^2 & 2bc - 2ad & 2bd + 2ac \\ 2bc + 2ad & 2a^2 - 1 + 2c^2 & 2cd - 2ab \\ 2bd - 2ac & 2cd + 2ab & 2a^2 - 1 + 2d^2 \end{bmatrix}.$$

References

- [1] Kuipers, Jack B. *Quaternions and Rotation Sequences: A Primer with Applications to Orbits, Aerospace, and Virtual Reality*. Princeton, NJ: Princeton University Press, 2007.

Extended Capabilities

C/C++ Code Generation

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

See Also

Functions

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

Objects

`quaternion`

Introduced in R2020b

rotvec

Convert quaternion to rotation vector (radians)

Syntax

```
rotationVector = rotvec(quat)
```

Description

`rotationVector = rotvec(quat)` converts the quaternion array, `quat`, to an N -by-3 matrix of equivalent rotation vectors in radians. The elements of `quat` are normalized before conversion.

Examples

Convert Quaternion to Rotation Vector in Radians

Convert a random quaternion scalar to a rotation vector in radians

```
quat = quaternion(randn(1,4));  
rotvec(quat)
```

```
ans = 1×3
```

```
    1.6866   -2.0774    0.7929
```

Input Arguments

quat — Quaternion to convert

scalar | vector | matrix | multidimensional array

Quaternion to convert, specified as scalar quaternion, vector, matrix, or multidimensional array of quaternions.

Data Types: quaternion

Output Arguments

rotationVector — Rotation vector (radians)

N -by-3 matrix

Rotation vector representation, returned as an N -by-3 matrix of rotations vectors, where each row represents the [X Y Z] angles of the rotation vectors in radians. The i th row of `rotationVector` corresponds to the element `quat(i)`.

The data type of the rotation vector is the same as the underlying data type of `quat`.

Data Types: single | double

Algorithms

All rotations in 3-D can be represented by a three-element axis of rotation and a rotation angle, for a total of four elements. If the rotation axis is constrained to be unit length, the rotation angle can be distributed over the vector elements to reduce the representation to three elements.

Recall that a quaternion can be represented in axis-angle form

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

where θ is the angle of rotation and $[x,y,z]$ represent the axis of rotation.

Given a quaternion of the form

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

you can solve for the rotation angle using the axis-angle form of quaternions:

$$\theta = 2\cos^{-1}(a).$$

Assuming a normalized axis, you can rewrite the quaternion as a rotation vector without loss of information by distributing θ over the parts b , c , and d . The rotation vector representation of q is

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

Extended Capabilities

C/C++ Code Generation

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

See Also

Functions

euler | eulerd | rotvecd

Objects

quaternion

Introduced in R2020b

rotvecd

Convert quaternion to rotation vector (degrees)

Syntax

```
rotationVector = rotvecd(quat)
```

Description

`rotationVector = rotvecd(quat)` converts the quaternion array, `quat`, to an N -by-3 matrix of equivalent rotation vectors in degrees. The elements of `quat` are normalized before conversion.

Examples

Convert Quaternion to Rotation Vector in Degrees

Convert a random quaternion scalar to a rotation vector in degrees.

```
quat = quaternion(randn(1,4));  
rotvecd(quat)
```

```
ans = 1×3
```

```
    96.6345  -119.0274   45.4312
```

Input Arguments

quat — Quaternion to convert

scalar | vector | matrix | multidimensional array

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

Data Types: quaternion

Output Arguments

rotationVector — Rotation vector (degrees)

N -by-3 matrix

Rotation vector representation, returned as an N -by-3 matrix of rotation vectors, where each row represents the $[x\ y\ z]$ angles of the rotation vectors in degrees. The i th row of `rotationVector` corresponds to the element `quat(i)`.

The data type of the rotation vector is the same as the underlying data type of `quat`.

Data Types: single | double

Algorithms

All rotations in 3-D can be represented by four elements: a three-element axis of rotation and a rotation angle. If the rotation axis is constrained to be unit length, the rotation angle can be distributed over the vector elements to reduce the representation to three elements.

Recall that a quaternion can be represented in axis-angle form

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

where θ is the angle of rotation in degrees, and $[x,y,z]$ represent the axis of rotation.

Given a quaternion of the form

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

you can solve for the rotation angle using the axis-angle form of quaternions:

$$\theta = 2\cos^{-1}(a).$$

Assuming a normalized axis, you can rewrite the quaternion as a rotation vector without loss of information by distributing θ over the parts b , c , and d . The rotation vector representation of q is

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

Extended Capabilities

C/C++ Code Generation

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

See Also

Functions

euler | eulerd | rotvec

Objects

quaternion

Introduced in R2020b

slerp

Spherical linear interpolation

Syntax

```
q0 = slerp(q1,q2,T)
```

Description

`q0 = slerp(q1,q2,T)` spherically interpolates between `q1` and `q2` by the interpolation coefficient `T`.

Examples

Interpolate Between Two Quaternions

Create two quaternions with the following interpretation:

- 1 `a` = 45 degree rotation around the z-axis
- 2 `c` = -45 degree rotation around the z-axis

```
a = quaternion([45,0,0], 'eulerd', 'ZYX', 'frame');
c = quaternion([-45,0,0], 'eulerd', 'ZYX', 'frame');
```

Call `slerp` with the quaternions `a` and `c` and specify an interpolation coefficient of 0.5.

```
interpolationCoefficient = 0.5;
b = slerp(a,c,interpolationCoefficient);
```

The output of `slerp`, `b`, represents an average rotation of `a` and `c`. To verify, convert `b` to Euler angles in degrees.

```
averageRotation = eulerd(b, 'ZYX', 'frame')
averageRotation = 1×3
```

```
    0    0    0
```

The interpolation coefficient is specified as a normalized value between 0 and 1, inclusive. An interpolation coefficient of 0 corresponds to the `a` quaternion, and an interpolation coefficient of 1 corresponds to the `c` quaternion. Call `slerp` with coefficients 0 and 1 to confirm.

```
b = slerp(a,c,[0,1]);
eulerd(b, 'ZYX', 'frame')
```

```
ans = 2×3
```

```
45.0000    0    0
```

```
-45.0000    0    0
```

You can create smooth paths between quaternions by specifying arrays of equally spaced interpolation coefficients.

```
path = 0:0.1:1;
interpolatedQuaternions = slerp(a,c,path);
```

For quaternions that represent rotation only about a single axis, specifying interpolation coefficients as equally spaced results in quaternions equally spaced in Euler angles. Convert `interpolatedQuaternions` to Euler angles and verify that the difference between the angles in the path is constant.

```
k = eulerd(interpolatedQuaternions, 'ZYX', 'frame');
abc = abs(diff(k))
```

```
abc = 10×3
```

```
 9.0000    0    0
 9.0000    0    0
 9.0000    0    0
 9.0000    0    0
 9.0000    0    0
 9.0000    0    0
 9.0000    0    0
 9.0000    0    0
 9.0000    0    0
 9.0000    0    0
```

Alternatively, you can use the `dist` function to verify that the distance between the interpolated quaternions is consistent. The `dist` function returns angular distance in radians; convert to degrees for easy comparison.

```
def = rad2deg(dist(interpolatedQuaternions(2:end),interpolatedQuaternions(1:end-1)))
```

```
def = 1×10
```

```
 9.0000    9.0000    9.0000    9.0000    9.0000    9.0000    9.0000    9.0000    9.0000    9.
```

SLERP Minimizes Great Circle Path

The SLERP algorithm interpolates along a great circle path connecting two quaternions. This example shows how the SLERP algorithm minimizes the great circle path.

Define three quaternions:

- 1 q0 - quaternion indicating no rotation from the global frame
- 2 q179 - quaternion indicating a 179 degree rotation about the z-axis
- 3 q180 - quaternion indicating a 180 degree rotation about the z-axis

4 q181 - quaternion indicating a 181 degree rotation about the z-axis

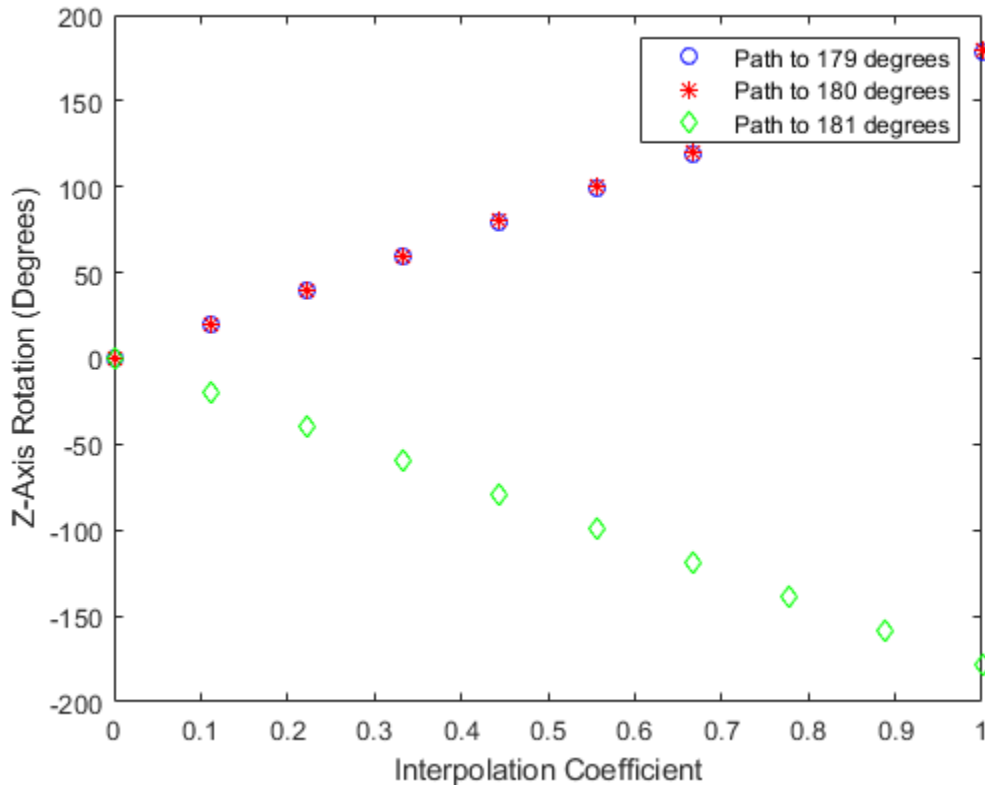
```
q0 = ones(1, 'quaternion');  
q179 = quaternion([179,0,0], 'eulerd', 'ZYX', 'frame');  
q180 = quaternion([180,0,0], 'eulerd', 'ZYX', 'frame');  
q181 = quaternion([181,0,0], 'eulerd', 'ZYX', 'frame');
```

Use `slerp` to interpolate between `q0` and the three quaternion rotations. Specify that the paths are traveled in 10 steps.

```
T = linspace(0,1,10);  
q179path = slerp(q0,q179,T);  
q180path = slerp(q0,q180,T);  
q181path = slerp(q0,q181,T);
```

Plot each path in terms of Euler angles in degrees.

```
q179pathEuler = eulerd(q179path, 'ZYX', 'frame');  
q180pathEuler = eulerd(q180path, 'ZYX', 'frame');  
q181pathEuler = eulerd(q181path, 'ZYX', 'frame');  
  
plot(T,q179pathEuler(:,1), 'bo', ...  
      T,q180pathEuler(:,1), 'r*', ...  
      T,q181pathEuler(:,1), 'gd');  
legend('Path to 179 degrees', ...  
       'Path to 180 degrees', ...  
       'Path to 181 degrees')  
xlabel('Interpolation Coefficient')  
ylabel('Z-Axis Rotation (Degrees)')
```

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

Input Arguments

q1 – Quaternion

scalar | vector | matrix | multidimensional array

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

q_1 , q_2 , and T must have compatible sizes. In the simplest cases, they can be the same size or any one can be a scalar. Two inputs have compatible sizes if, for every dimension, the dimension sizes of the inputs are either the same or one of them is 1.

Data Types: quaternion

q2 – Quaternion

scalar | vector | matrix | multidimensional array

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

q_1 , q_2 , and T must have compatible sizes. In the simplest cases, they can be the same size or any one can be a scalar. Two inputs have compatible sizes if, for every dimension, the dimension sizes of the inputs are either the same or one of the dimension sizes is 1.

Data Types: quaternion

T — Interpolation coefficient

scalar | vector | matrix | multidimensional array

Interpolation coefficient, specified as a scalar, vector, matrix, or multidimensional array of numbers with each element in the range [0,1].

q_1 , q_2 , and T must have compatible sizes. In the simplest cases, they can be the same size or any one can be a scalar. Two inputs have compatible sizes if, for every dimension, the dimension sizes of the inputs are either the same or one of the dimension sizes is 1.

Data Types: single | double

Output Arguments

q0 — Interpolated quaternion

scalar | vector | matrix | multidimensional array

Interpolated quaternion, returned as a scalar, vector, matrix, or multidimensional array.

Data Types: quaternion

Algorithms

Quaternion **spherical linear interpolation** (SLERP) is an extension of linear interpolation along a plane to spherical interpolation in three dimensions. The algorithm was first proposed in [1]. Given two quaternions, q_1 and q_2 , SLERP interpolates a new quaternion, q_0 , along the great circle that connects q_1 and q_2 . The interpolation coefficient, T , determines how close the output quaternion is to either q_1 and q_2 .

The SLERP algorithm can be described in terms of sinusoids:

$$q_0 = \frac{\sin((1 - T)\theta)}{\sin(\theta)}q_1 + \frac{\sin(T\theta)}{\sin(\theta)}q_2$$

where q_1 and q_2 are normalized quaternions, and θ is half the angular distance between q_1 and q_2 .

References

- [1] Shoemake, Ken. "Animating Rotation with Quaternion Curves." *ACM SIGGRAPH Computer Graphics* Vol. 19, Issue 3, 1985, pp. 345-354.

Extended Capabilities

C/C++ Code Generation

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

See Also

Functions

dist | meanrot

Objects

quaternion

Introduced in R2020b

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=2×1 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=3×3 quaternion array
    0.60169 + 2.4332i - 2.5844j + 0.51646k   -0.49513 + 1.1722i + 4.4401j - 1.217k
   -4.2329 + 2.4547i + 3.7768j + 0.77484k   -0.65232 - 0.43112i - 1.4645j - 0.90073k
```

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

Note that quaternion multiplication is not commutative:

```
isequal(C,B.*A)
```

```
ans = logical
      0
```

Multiply Quaternion Row and Column Vectors

Create a row vector **a** and a column vector **b**, then multiply them. The 1-by-3 row vector and 4-by-1 column vector combine to produce a 4-by-3 matrix with all combinations of elements multiplied.

```
a = [zeros('quaternion'),ones('quaternion'),quaternion(randn(1,4))]
```

```
a=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 -      0.12414k
      0 +      0i +      0j +      0k      -1.3077 +      2.7694i -      0.063055j +      1.4897k
      0 +      0i +      0j +      0k      -0.43359 -      1.3499i +      0.71474j +      1.409k
      0 +      0i +      0j +      0k      0.34262 +      3.0349i -      0.20497j +      1.4172k
```

Input Arguments

A — Array to multiply

scalar | vector | matrix | multidimensional array

Array to multiply, specified as a quaternion, an array of quaternions, a real scalar, or an array of real numbers.

A and B must have compatible sizes. In the simplest cases, they can be the same size or one can be a scalar. Two inputs have compatible sizes if, for every dimension, the dimension sizes of the inputs are the same or one of them is 1.

Data Types: quaternion | single | double

B – Array to multiply

scalar | vector | matrix | multidimensional array

Array to multiply, specified as a quaternion, an array of quaternions, a real scalar, or an array of real numbers.

A and B must have compatible sizes. In the simplest cases, they can be the same size or one can be a scalar. Two inputs have compatible sizes if, for every dimension, the dimension sizes of the inputs are the same or one of them is 1.

Data Types: quaternion | single | double

Output Arguments**quatC – Quaternion product**

scalar | vector | matrix | multidimensional array

Quaternion product, returned as a scalar, vector, matrix, or multidimensional array.

Data Types: quaternion

Algorithms**Quaternion Multiplication by a Real Scalar**

Given a quaternion,

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

the product of q and a real scalar β is

$$\beta q = \beta a_q + \beta b_q i + \beta c_q j + \beta d_q k$$

Quaternion Multiplication by a Quaternion Scalar

The definition of the basis elements for quaternions,

$$i^2 = j^2 = k^2 = ijk = -1,$$

can be expanded to populate a table summarizing quaternion basis element multiplication:

	1	i	j	k
1	1	i	j	k
i	i	-1	k	-j
j	j	-k	-1	i
k	k	j	-i	-1

When reading the table, the rows are read first, for example: $ij = k$ and $ji = -k$.

Given two quaternions, $q = a_q + b_q i + c_q j + d_q k$, and $p = a_p + b_p i + c_p j + d_p k$, the multiplication can be expanded as:

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

You can simplify the equation using the quaternion multiplication table.

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

References

- [1] Kuipers, Jack B. *Quaternions and Rotation Sequences: A Primer with Applications to Orbits, Aerospace, and Virtual Reality*. Princeton, NJ: Princeton University Press, 2007.

Extended Capabilities

C/C++ Code Generation

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

See Also

Functions

mtimes, * | prod

Objects

quaternion

Introduced in R2020b

transpose, .'

Transpose a quaternion array

Syntax

`Y = quat.'`

Description

`Y = quat.'` returns the non-conjugate transpose of the quaternion array, `quat`.

Examples

Vector Transpose

Create a vector of quaternions and compute its nonconjugate transpose.

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

```
quat=4×1 quaternion array
    0.53767 + 0.31877i + 3.5784j + 0.7254k
    1.8339 - 1.3077i + 2.7694j - 0.063055k
   -2.2588 - 0.43359i - 1.3499j + 0.71474k
    0.86217 + 0.34262i + 3.0349j - 0.20497k
```

```
quatTransposed = quat.'
```

```
quatTransposed=1×4 quaternion array
    0.53767 + 0.31877i + 3.5784j + 0.7254k    1.8339 - 1.3077i + 2.7694j - 0.063055k
```

Matrix Transpose

Create a matrix of quaternions and compute its nonconjugate transpose.

```
quat = [quaternion(randn(2,4)), quaternion(randn(2,4))]
```

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

```
quatTransposed = quat.'
```

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


Input Arguments

quat — Quaternion array to transpose

vector | matrix

Quaternion array to transpose, specified as a vector or matrix of quaternions. `transpose` is defined for 1-D and 2-D arrays. For higher-order arrays, use `permute`.

Data Types: quaternion

Output Arguments

Y — Transposed quaternion array

vector | matrix

Transposed quaternion array, returned as an N -by- M array, where `quat` was specified as an M -by- N array.

Extended Capabilities

C/C++ Code Generation

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

See Also

Functions

`ctranspose`, '

Objects

quaternion

Introduced in R2020b

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 + 0.71474k  
    1.8339 - 1.3077i + 2.7694j - 0.063055k    0.86217 + 0.34262i + 3.0349j - 0.20494k
```

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 - 0.71474k  
   -1.8339 + 1.3077i - 2.7694j + 0.063055k   -0.86217 - 0.34262i - 3.0349j + 0.20494k
```

Input Arguments

quat — Quaternion array

scalar | vector | matrix | multidimensional array

Quaternion array, specified as a scalar, vector, matrix, or multidimensional array.

Data Types: quaternion

Output Arguments

mQuat — Negated quaternion array

scalar | vector | matrix | multidimensional array

Negated quaternion array, returned as the same size as quat.

Data Types: quaternion

Extended Capabilities

C/C++ Code Generation

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

See Also

Functions

minus, -

Objects

quaternion

Introduced in R2020b

zeros

Create quaternion array with all parts set to zero

Syntax

```
quatZeros = zeros('quaternion')
quatZeros = zeros(n,'quaternion')
quatZeros = zeros(sz,'quaternion')
quatZeros = zeros(sz1,...,szN,'quaternion')

quatZeros = zeros( ____, 'like', prototype, 'quaternion')
```

Description

`quatZeros = zeros('quaternion')` returns a scalar quaternion with all parts set to zero.

`quatZeros = zeros(n,'quaternion')` returns an n-by-n matrix of quaternions.

`quatZeros = zeros(sz,'quaternion')` returns an array of quaternions where the size vector, `sz`, defines `size(quatZeros)`.

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

`quatZeros = zeros(____, 'like', prototype, 'quaternion')` specifies the underlying class of the returned quaternion array to be the same as the underlying class of the quaternion prototype.

Examples

Quaternion Scalar Zero

Create a quaternion scalar zero.

```
quatZeros = zeros('quaternion')

quatZeros = quaternion
           0 + 0i + 0j + 0k
```

Square Matrix of Quaternions

Create an n-by-n array of quaternion zeros.

```
n = 3;
quatZeros = zeros(n,'quaternion')

quatZeros=3x3 quaternion array
           0 + 0i + 0j + 0k           0 + 0i + 0j + 0k           0 + 0i + 0j + 0k
```

```

0 + 0i + 0j + 0k    0 + 0i + 0j + 0k    0 + 0i + 0j + 0k
0 + 0i + 0j + 0k    0 + 0i + 0j + 0k    0 + 0i + 0j + 0k

```

Multidimensional Array of Quaternion Zeros

Create a multidimensional array of quaternion zeros by defining array dimensions in order. In this example, you create a 3-by-1-by-2 array. You can specify dimensions using a row vector or comma-separated integers.

Specify the dimensions using a row vector and display the results:

```

dims = [3,1,2];
quatZerosSyntax1 = zeros(dims, 'quaternion')

```

```

quatZerosSyntax1 = 3x1x2 quaternion array
quatZerosSyntax1(:,:,1) =

```

```

0 + 0i + 0j + 0k
0 + 0i + 0j + 0k
0 + 0i + 0j + 0k

```

```

quatZerosSyntax1(:,:,2) =

```

```

0 + 0i + 0j + 0k
0 + 0i + 0j + 0k
0 + 0i + 0j + 0k

```

Specify the dimensions using comma-separated integers, and then verify the equivalence of the two syntaxes:

```

quatZerosSyntax2 = zeros(3,1,2, 'quaternion');
isequal(quatZerosSyntax1, quatZerosSyntax2)

```

```

ans = logical
     1

```

Underlying Class of Quaternion Zeros

A quaternion is a four-part hyper-complex number used in three-dimensional representations. You can specify the underlying data type of the parts as `single` or `double`. The default is `double`.

Create a quaternion array of zeros with the underlying data type set to `single`.

```

quatZeros = zeros(2, 'like', single(1), 'quaternion')

```

```

quatZeros=2x2 quaternion array
0 + 0i + 0j + 0k    0 + 0i + 0j + 0k
0 + 0i + 0j + 0k    0 + 0i + 0j + 0k

```

Verify the underlying class using the `classUnderlying` function.

```
classUnderlying(quatZeros)

ans =
'single'
```

Input Arguments

n — Size of square quaternion matrix

integer value

Size of square quaternion matrix, specified as an integer value. If `n` is 0 or negative, then `quatZeros` is returned as an empty matrix.

Example: `zeros(4, 'quaternion')` returns a 4-by-4 matrix of quaternion zeros.

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

sz — Output size

row vector of integer values

Output size, specified as a row vector of integer values. Each element of `sz` indicates the size of the corresponding dimension in `quatZeros`. If the size of any dimension is 0 or negative, then `quatZeros` is returned as an empty array.

Example: `zeros([1,4,2], 'quaternion')` returns a 1-by-4-by-2 array of quaternion zeros.

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

prototype — Quaternion prototype

variable

Quaternion prototype, specified as a variable.

Example: `zeros(2, 'like', quat, 'quaternion')` returns a 2-by-2 matrix of quaternions with the same underlying class as the prototype quaternion, `quat`.

Data Types: `quaternion`

sz1, ..., szN — Size of each dimension

two or more integer values

Size of each dimension, specified as two or more integers.

- If the size of any dimension is 0, then `quatZeros` is returned as an empty array.
- If the size of any dimension is negative, then it is treated as 0.

Example: `zeros(2,3, 'quaternion')` returns a 2-by-3 matrix of quaternion zeros.

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

Output Arguments

quatZeros — Quaternion zeros

scalar | vector | matrix | multidimensional array

Quaternion zeros, returned as a quaternion or array of quaternions.

Given a quaternion of the form $Q = a + bi + cj + dk$, a quaternion zero is defined as $Q = 0 + 0i + 0j + 0k$.

Data Types: quaternion

Extended Capabilities

C/C++ Code Generation

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

See Also

Functions

ones

Objects

quaternion

Introduced in R2020b

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`.

See Also

Objects

`fixedwing` | `multirotor` | `transformTree` | `uavDubinsPathSegment`

Functions

`getTransform` | `info` | `removeTransform` | `show` | `updateTransform`

Introduced in R2020b

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 `f 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`.

See Also

Objects

`fixedwing` | `multicopter` | `transformTree` | `uavDubinsPathSegment`

Functions

`getGraph` | `info` | `removeTransform` | `show` | `updateTransform`

Introduced in R2020b

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 NaNs based on the MaxNumTransforms property of frames.

See Also

Objects

fixedwing | multirotor | transformTree | uavDubinsPathSegment

Functions

getGraph | getTransform | removeTransform | show | updateTransform

Introduced in R2020b

removeTransform

Remove frame transform relative to its parent

Syntax

```
removeTransforms(frames, framename, timestamp)
removeTransforms(frames, framename, timeStart, timeEnd)
```

Description

`removeTransforms(frames, framename, timestamp)` removes the frame transforms between the given frame name and their parent frame at the specified timestamps.

`removeTransforms(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.

See Also

Objects

fixedwing | multicopter | transformTree | uavDubinsPathSegment

Functions

getGraph | getTransform | info | show | updateTransform

Introduced in R2020b

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

Example: `'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](#).

See Also

Objects

`fixedwing` | `multirotor` | `transformTree` | `uavDubinsPathSegment`

Functions

`getGraph` | `getTransform` | `info` | `removeTransform` | `updateTransform`

Introduced in R2020b

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.

See Also

Objects

`fixedwing` | `multirotor` | `transformTree` | `uavDubinsPathSegment`

Functions

`getGraph` | `getTransform` | `info` | `removeTransform` | `show`

Introduced in R2020b

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.

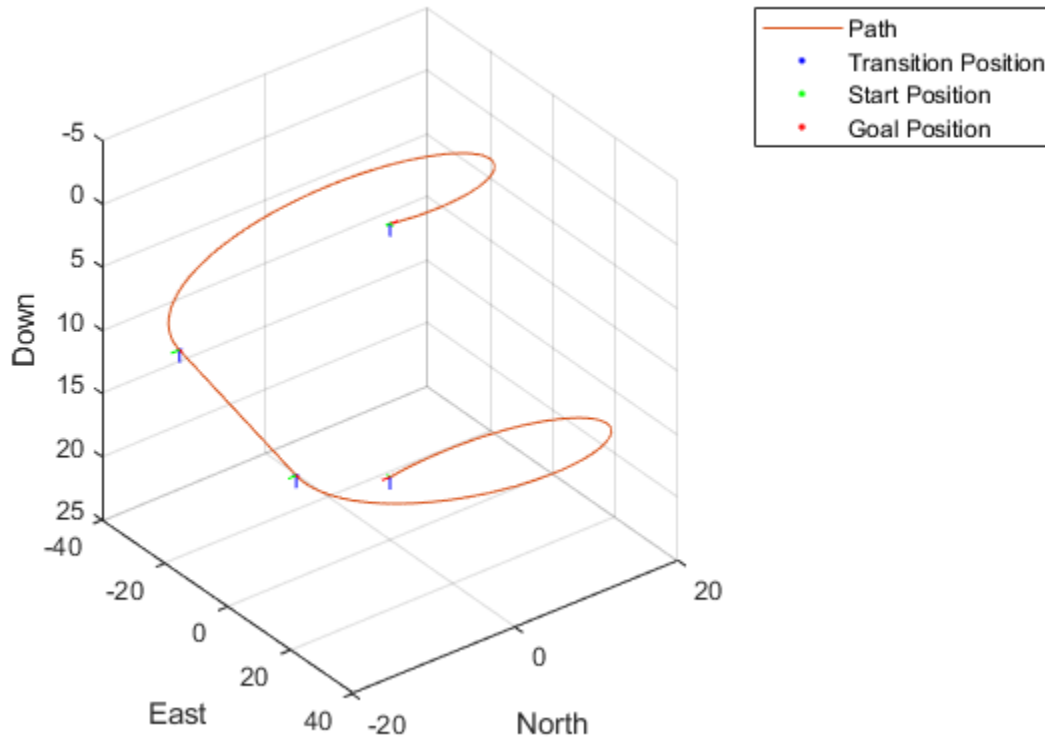
```
[pathSegObj,pathCosts] = connect(connectionObj,startPose,goalPose,'PathSegments','all');
```

Path Validation and Visualization

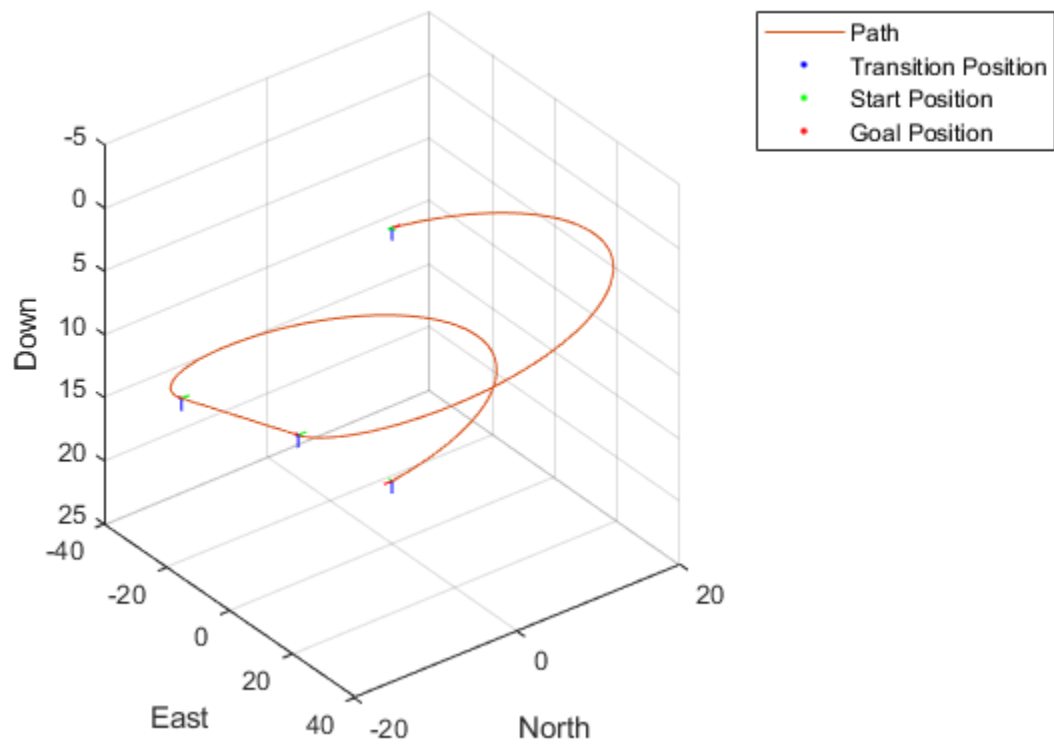
Check the validity of all the possible path segments and display the valid paths along with their motion type and path cost.

```
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(i));
    end
end
```

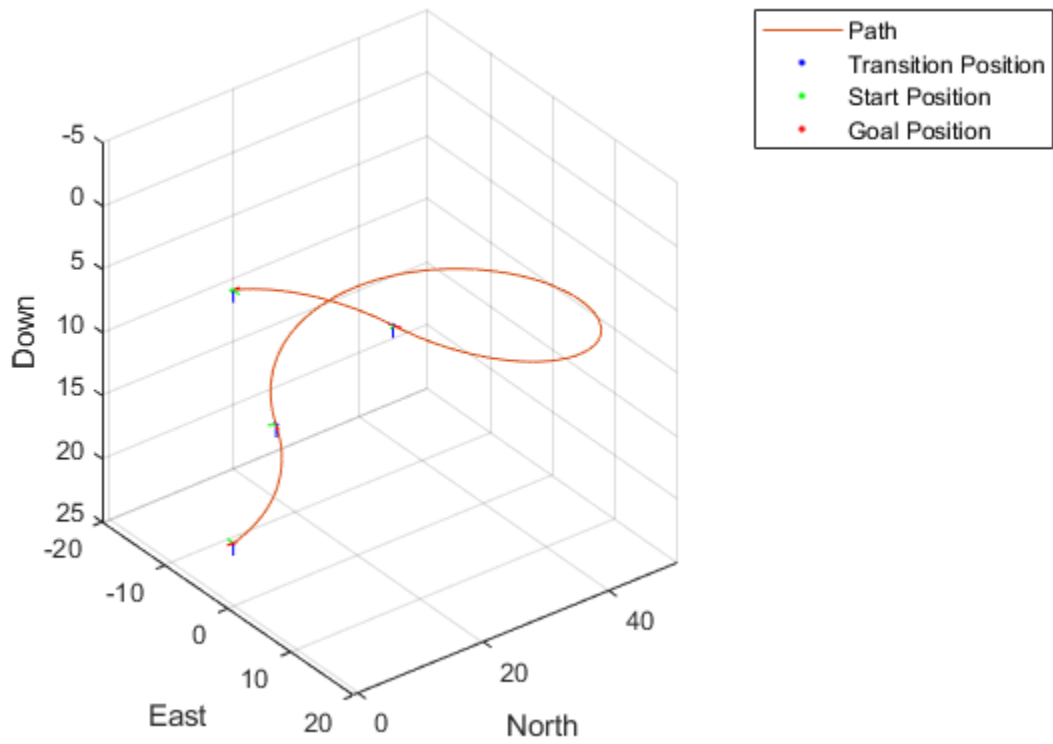
end
end



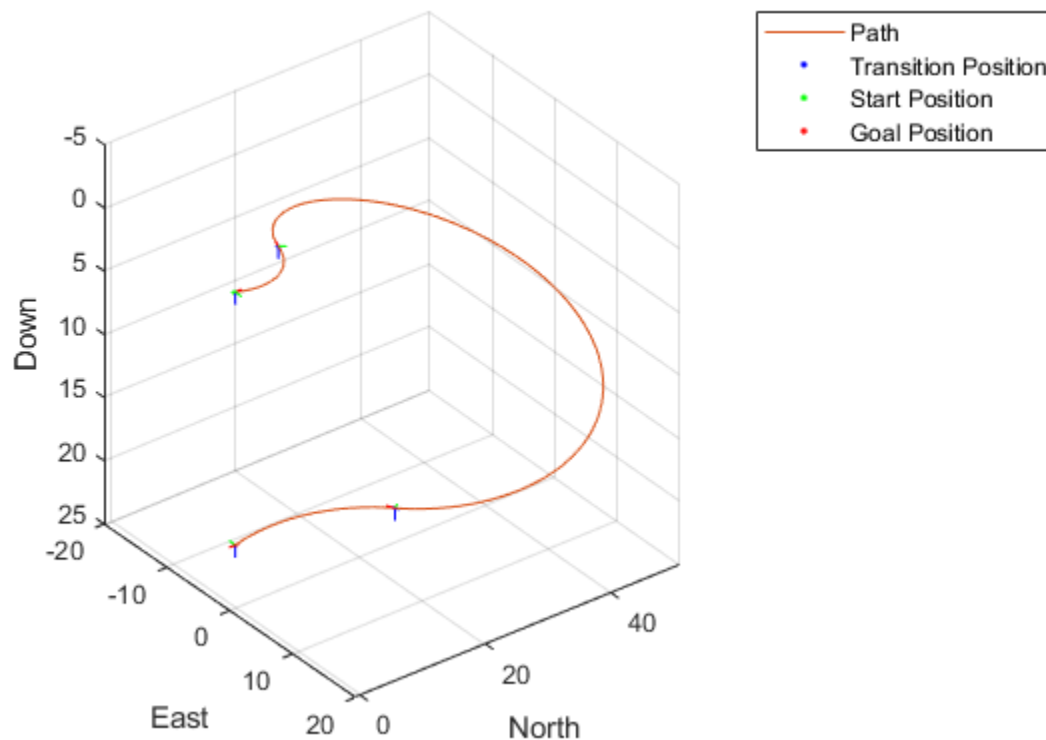
Motion Type: L S L N
Path Cost: 214.332271



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.

Extended Capabilities

C/C++ Code Generation

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

See Also

[uavDubinsConnection](#) | [uavDubinsPathSegment](#)

Introduced in R2019b

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.

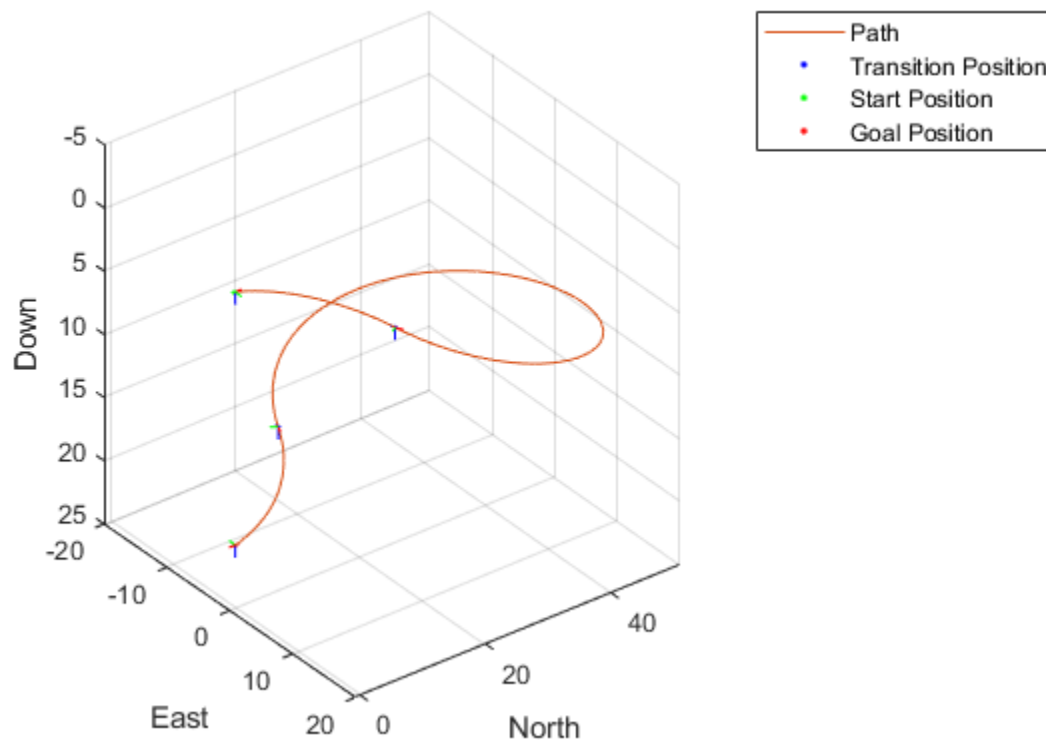
```
startPose = [0 0 0 0]; % [meters, meters, meters, radians]  
goalPose = [0 0 20 pi];
```

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`.

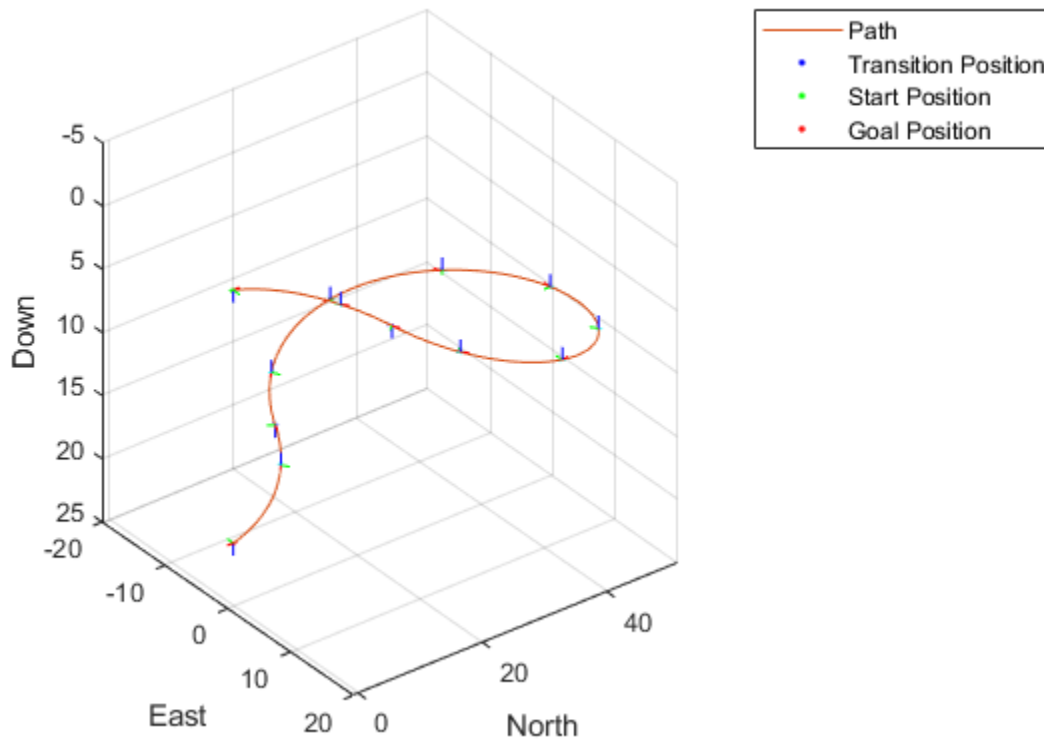
```
posesTranslation = poses(2:end-1,1:3); % [x, y, z]
```

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.

Extended Capabilities

C/C++ Code Generation

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

See Also

[show](#) | [uavDubinsPathSegment](#)

Introduced in R2019b

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.

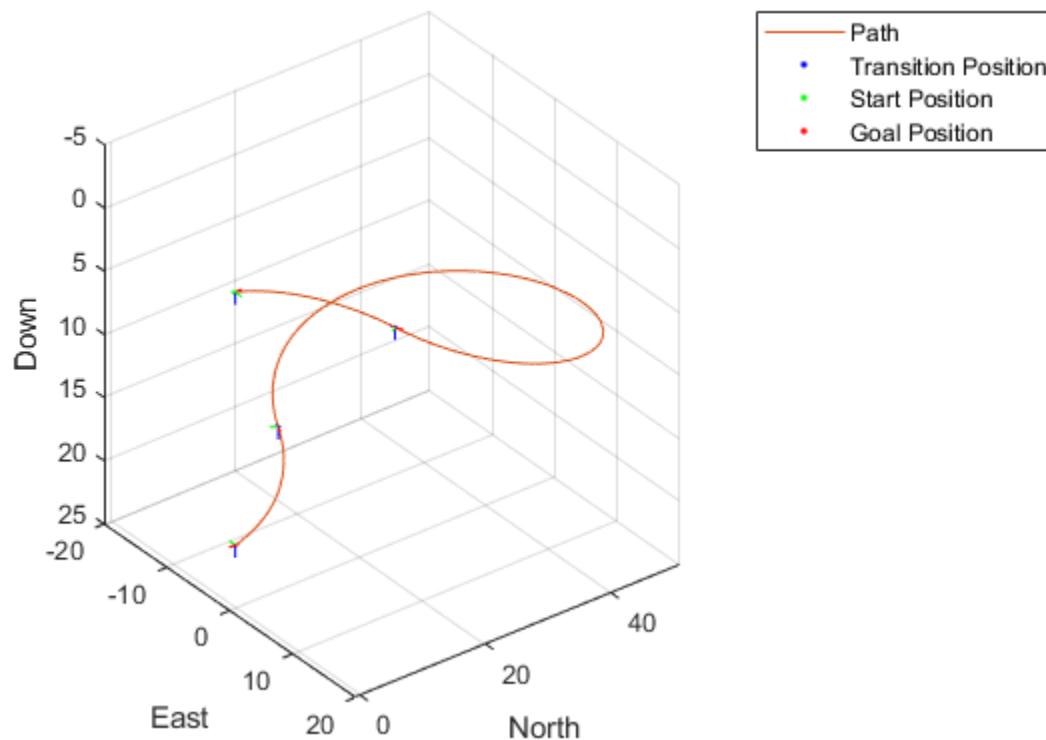
```
startPose = [0 0 0 0]; % [meters, meters, meters, radians]
goalPose = [0 0 20 pi];
```

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

Example: `'Positions',{'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.

See Also

plotTransforms | uavDubinsPathSegment

Introduced in R2019b

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

Example: `'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`

See Also

Functions

`checkPermission` | `move` | `read` | `updateMesh`

Objects

`uavPlatform` | `uavScenario` | `uavSensor`

Topics

“UAV Scenario Tutorial”

Introduced in R2020b

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

Example: '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

See Also**Functions**

addGeoFence | move | read | updateMesh

Objects

uavPlatform | uavScenario | uavSensor

Topics

"UAV Scenario Tutorial"

Introduced in R2020b

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`

See Also

Functions

`addGeoFence` | `checkPermission` | `read` | `updateMesh`

Objects

`uavPlatform` | `uavScenario` | `uavSensor`

Topics

“UAV Scenario Tutorial”

Introduced in R2020b

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

See Also

Functions

addGeoFence | checkPermission | move | updateMesh

Objects

uavPlatform | uavScenario | uavSensor

Topics

“UAV Scenario Tutorial”

Introduced in R2020b

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

See Also

Functions

addGeoFence | checkPermission | move | read

Objects

uavPlatform | uavScenario | uavSensor

Topics

“UAV Scenario Tutorial”

Introduced in R2020b

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]

See Also

Introduced in R2020b

addMesh

Add new static mesh to UAV scenario

Syntax

```
addMesh(scene,type,geometry,color)
addMesh( ____,Name,Value)
```

Description

`addMesh(scene,type,geometry,color)` adds a static mesh to the UAV scenario scene by specifying the mesh type, geometry, and color.

`addMesh(____,Name,Value)` specifies additional options using name-value arguments. Enclose each Name in quotes.

Examples

Add Meshes to UAV scenario

Create a UAV Scenario.

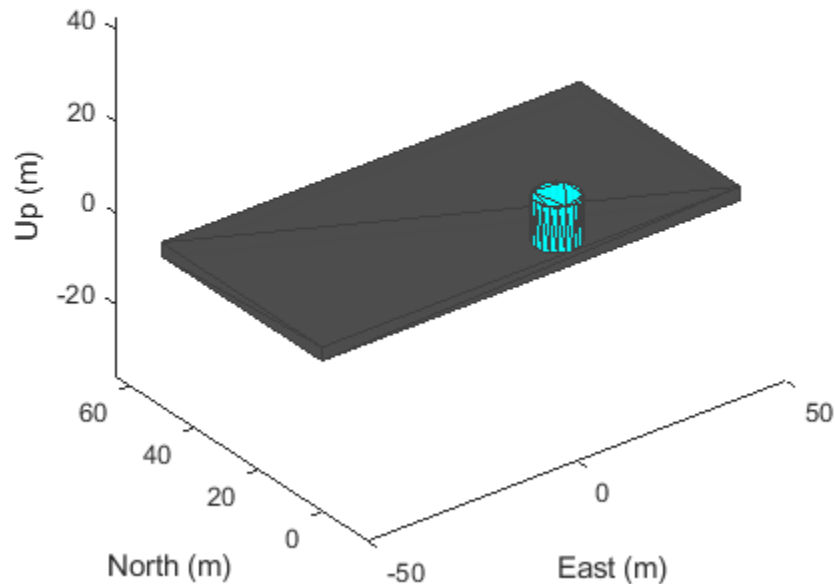
```
scene = uavScenario("UpdateRate",100,"StopTime",1);
```

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);
```



Input Arguments

scene — UAV scenario

uavScenario object

UAV scenario, specified as a uavScenario object.

type — Mesh type

"cylinder" | "surface" | "terrain" | "polygon" | "custom"

Mesh type, specified as "cylinder", "surface", "terrain", "polygon", or "custom". Specify the geometric parameters of the mesh using the geometry input.

Data Types: string

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 <i>N</i> -by-2 matrix, where <i>N</i> is the number of corner points. The first column contains the x-coordinates and the second column contains the y-coordinates in meters.
"custom"	{vertices,faces}	vertices is an <i>n</i> -by-3 matrix of mesh points in local coordinates. faces is an <i>n</i> -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 comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1, Value1, . . . , NameN, ValueN`.

Example: `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.
- 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.

See Also

`addCustomTerrain` | `removeCustomTerrain` | `terrainHeight` | `uavScenario`

Introduced in R2020b

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.

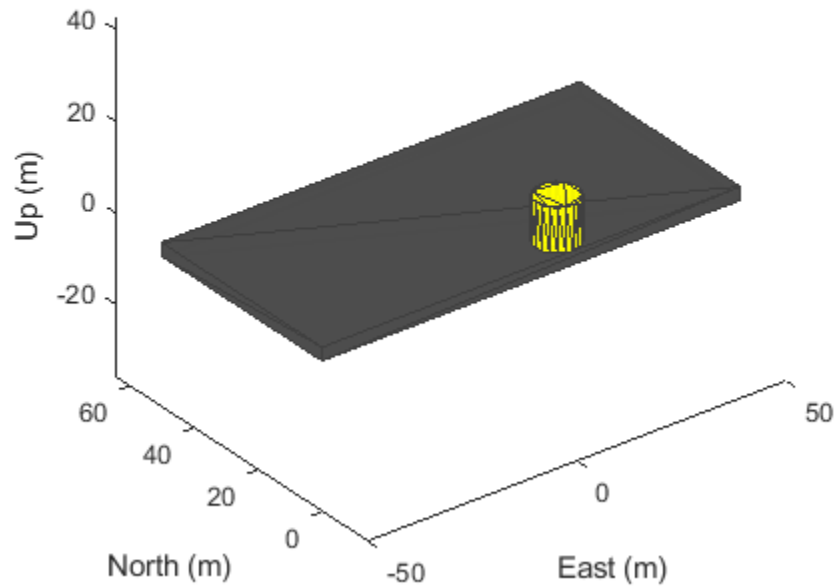
```
addMesh(scene,"Polygon", {[ -50 0; 50 0; 50 50; -50 50], [-3 0]}, [0.3 0.3 0.3]);
addMesh(scene,"Cylinder", {[10 5 5], [0 10]}, [1 1 0]);
```

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.

See Also

Introduced in R2020b

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.

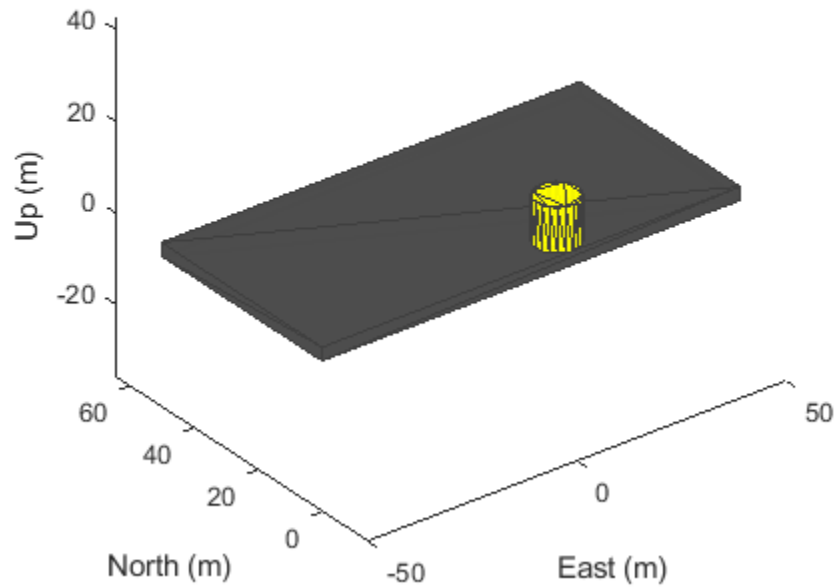
```
addMesh(scene,"Polygon", {[ -50 0; 50 0; 50 50; -50 50], [-3 0]}, [0.3 0.3 0.3]);
addMesh(scene,"Cylinder", {[10 5 5], [0 10]}, [1 1 0]);
```

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.

See Also

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.

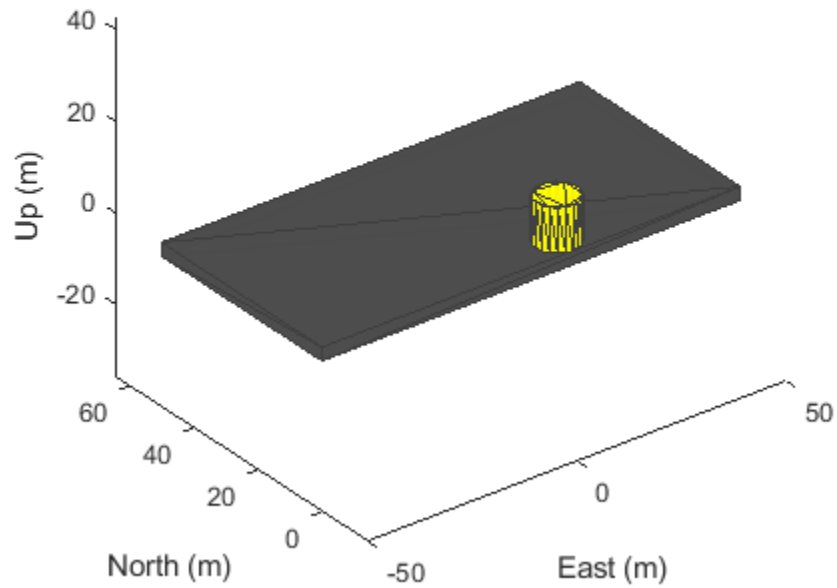
```
addMesh(scene,"Polygon", {[ -50 0; 50 0; 50 50; -50 50], [-3 0]}, [0.3 0.3 0.3]);
addMesh(scene,"Cylinder", {[10 5 5], [0 10]}, [1 1 0]);
```

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.

See Also

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.

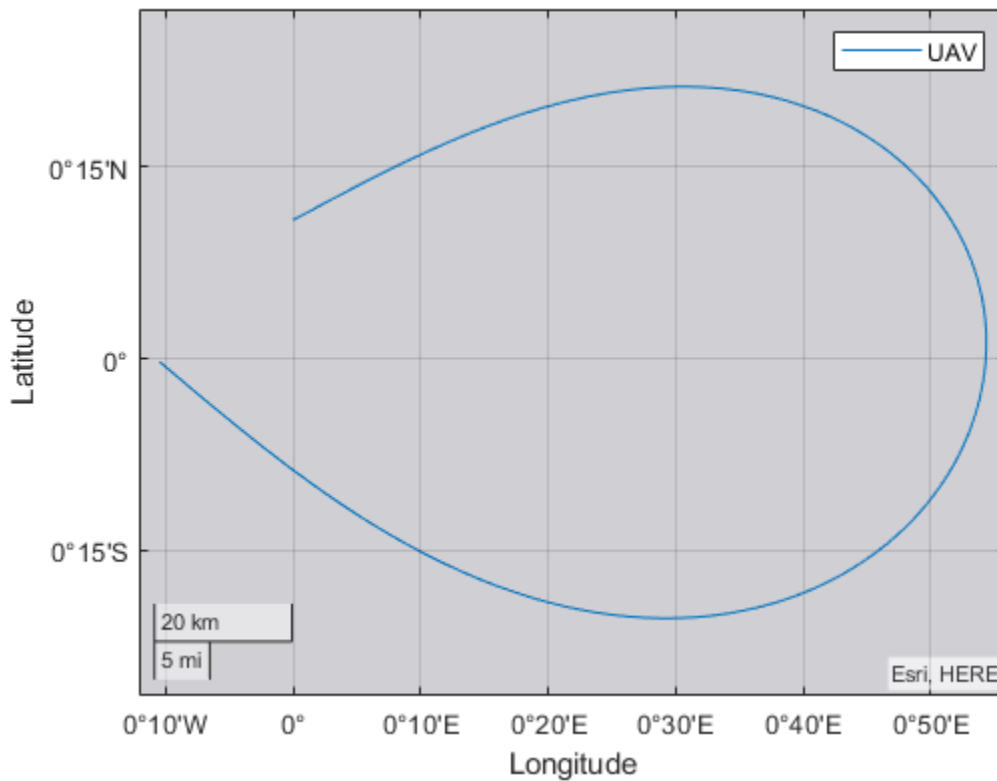
```
scene = uavScenario("UpdateRate",1,"StopTime",1000,"HistoryBufferSize",1000);
```

Create a UAV platform with a specified waypoint trajectory in the scenario.

```
traj = waypointTrajectory("Waypoints",[0 -20000 -50; 10000 100000 -50; 20000 0 -50],"TimeOfArrival",1000);
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 comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1, Value1, . . . , NameN, ValueN`.

Example:

Parent — 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.

See Also

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]);
```

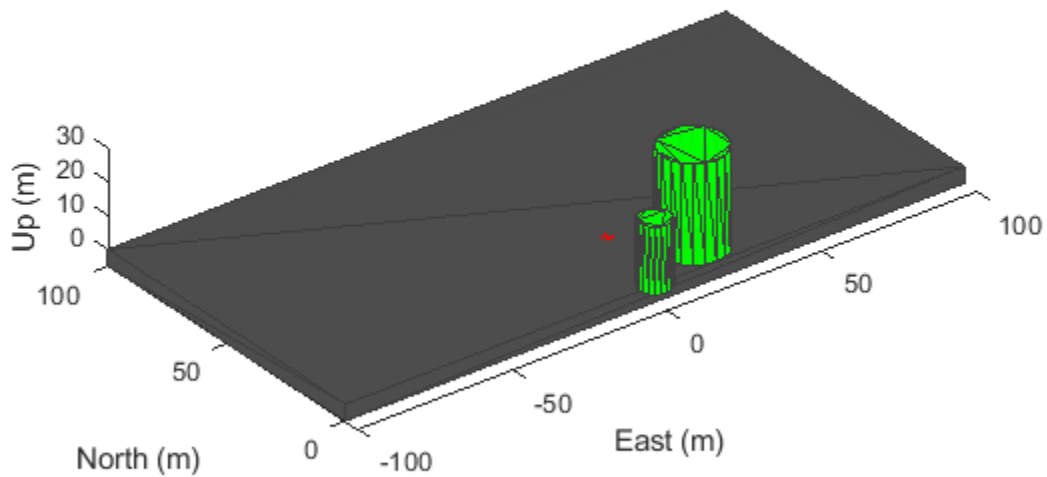
Visualize the scenario in 3-D.

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

Example:

Parent — 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 struct 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.

See Also

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

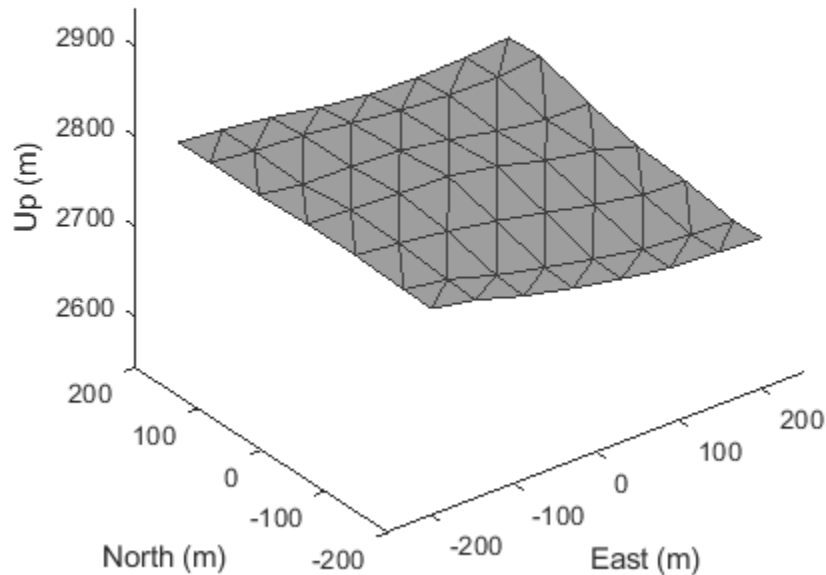
Add Terrain and Buildings to UAV Scenario

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);
```



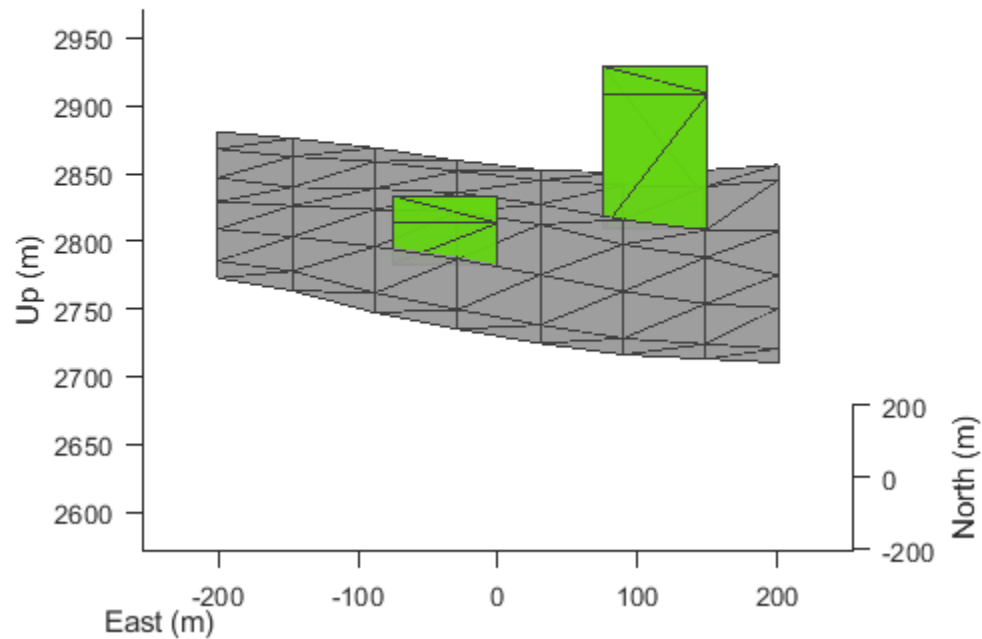
Add Buildings

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)]}, [0.39]);
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 comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example: `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`

See Also

`addCustomTerrain` | `addMesh` | `removeCustomTerrain` | `uavScenario`

Introduced in R2021a

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],[0 1 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.

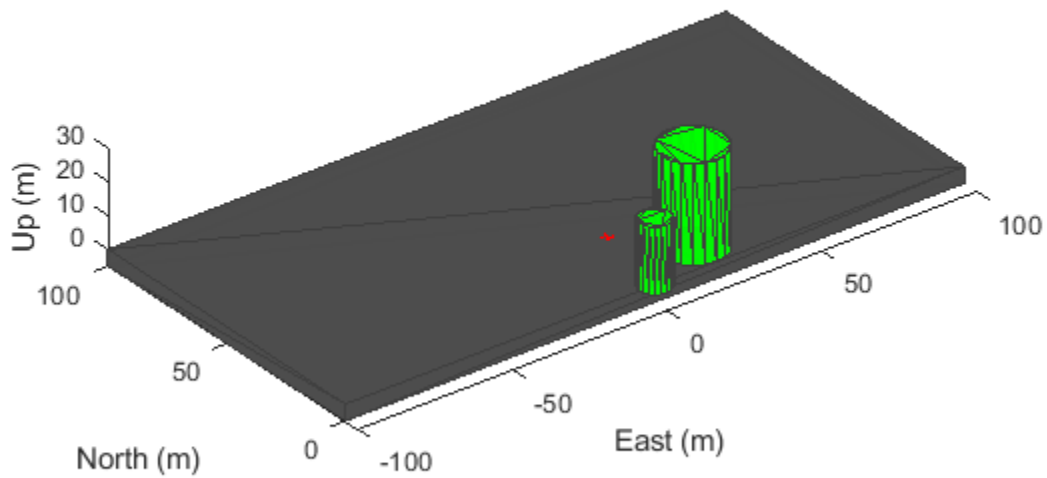
```
ax = show3D(scene);
axis(ax,"equal");
```

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.

See Also

gpsSensor | insSensor | uavSensor

Introduced in R2020b

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.

See Also

Objects

`uavPlatform` | `uavScenario` | `uavSensor`

Topics

“UAV Scenario Tutorial”

Introduced in R2020b

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.

See Also

Functions

`read` | `reset` | `setup` | `uav.SensorAdaptor.getMotion`

Objects

`uav.SensorAdaptor` | `uavPlatform` | `uavScenario` | `uavSensor`

Topics

“Simulate Radar Sensor Mounted On UAV”

Introduced in R2021a

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
- [qw qx qy qz] — Quaternion vector for orientation
- [wx wy wz] — Angular velocities in radians per second

Data Types: double

See Also

Functions

getEmptyOutputs | read | reset | setup

Objects

uav.SensorAdaptor | uavPlatform | uavScenario | uavSensor

Topics

“Simulate Radar Sensor Mounted On UAV”

Introduced in R2021a

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`.

See Also

Functions

`getEmptyOutputs` | `reset` | `setup` | `uav.SensorAdaptor.getMotion`

Objects

`uav.SensorAdaptor` | `uavPlatform` | `uavScenario` | `uavSensor`

Topics

“Simulate Radar Sensor Mounted On UAV”

Introduced in R2021a

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`.

See Also

Functions

`getEmptyOutputs` | `read` | `setup` | `uav.SensorAdaptor.getMotion`

Objects

`uav.SensorAdaptor` | `uavPlatform` | `uavScenario` | `uavSensor`

Topics

“Simulate Radar Sensor Mounted On UAV”

Introduced in R2021a

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`.

See Also

Functions

`getEmptyOutputs` | `read` | `reset` | `setup` | `uav.SensorAdaptor.getMotion`

Objects

`uav.SensorAdaptor` | `uavPlatform` | `uavScenario` | `uavSensor`

Topics

“Simulate Radar Sensor Mounted On UAV”

Introduced in R2021a

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.SSSSS');
```

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

Example: '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

See Also

Objects

uLogreader

Functions

readParameters | readSystemInformation | readTopicMsgs

Introduced in R2020b

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.SSSSS');
```

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

See Also

Objects

uLogreader

Functions

readLoggedOutput | readSystemInformation | readTopicMsgs

Introduced in R2020b

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.SSSSS');
```

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

See Also

Objects

uLogreader

Functions

readLoggedOutput | readParameters | readTopicMsgs

Introduced in R2020b

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.SSSSS');
```

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

Example: '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.SSSSS' 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

See Also

Objects

ulogreader

Functions

readLoggedOutput | readParameters | readSystemInformation

Introduced in R2020b

lookupPose

Obtain pose information for certain time

Syntax

```
[position,orientation,velocity,acceleration,angularVelocity] = lookupPose(
traj,sampleTimes)
```

Description

[position,orientation,velocity,acceleration,angularVelocity] = lookupPose(traj,sampleTimes) returns the pose information of the waypoint trajectory at the specified sample times. If any sample time is beyond the duration of the trajectory, the corresponding pose information is returned as NaN.

Input Arguments

traj — Waypoint trajectory

waypointTrajectory object

Waypoint trajectory, specified as a waypointTrajectory object.

sampleTimes — Sample times

M -element vector of nonnegative scalar

Sample times in seconds, specified as an M -element vector of nonnegative scalars.

Output Arguments

position — Position in local navigation coordinate system (m)

M -by-3 matrix

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

M is specified by the sampleTimes input.

Data Types: double

orientation — Orientation in local navigation coordinate system

M -element quaternion column vector | 3-by-3-by- M real array

Orientation in the local navigation coordinate system, returned as an M -by-1 quaternion column vector or a 3-by-3-by- M real array.

Each quaternion or 3-by-3 rotation matrix is a frame rotation from the local navigation coordinate system to the current body coordinate system.

M is specified by the sampleTimes input.

Data Types: double

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

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

M is specified by the `sampleTimes` input.

Data Types: `double`

acceleration — Acceleration in local navigation coordinate system (m/s²)*M*-by-3 matrix

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

M is specified by the `sampleTimes` input.

Data Types: `double`

angularVelocity — Angular velocity in local navigation coordinate system (rad/s)*M*-by-3 matrix

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

M is specified by the `sampleTimes` input.

Data Types: `double`

See Also`waypointTrajectory`**Introduced in R2020b**

waypointInfo

Get waypoint information table

Syntax

```
trajectoryInfo = waypointInfo(trajectory)
```

Description

`trajectoryInfo = waypointInfo(trajectory)` returns a table of waypoints, times of arrival, velocities, and orientation for the trajectory System object

Input Arguments

trajectory – Object of waypointTrajectory

object

Object of the waypointTrajectory System object.

Output Arguments

trajectoryInfo – Trajectory information

table

Trajectory information, returned as a table with variables corresponding to set creation properties: Waypoints, TimeOfArrival, Velocities, and Orientation.

The trajectory information table always has variables `Waypoints` and `TimeOfArrival`. If the `Velocities` property is set during construction, the trajectory information table additionally returns velocities. If the `Orientation` property is set during construction, the trajectory information table additionally returns orientation.

See Also

`waypointTrajectory`

Introduced in R2020b

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=2x3 table
  Property      Type      Value
  _____  _____  _____
  "Waypoints"   "Normal"  {[ 1]}    {[ 1]}
  "TimeOfArrival" "None"    {[NaN]}   {[NaN]}
```

```
perturbs2 = perturbations(traj, 'TimeOfArrival', 'Selection', {[0;1],[0;2]})
```



```

perturbs2=2x3 table
Property          Type          Value
-----
"Waypoints"      "Normal"      {[ 1]}
"TimeOfArrival" "Selection"   {[0.5000 0.5000]}

```

Perturb the trajectory.

```
offsets = perturb(traj)
```

```

offsets=2x1 struct array with fields:
  Property
  Offset
  PerturbedValue

```

The Waypoints property and the TimeOfArrival property have changed.

```
traj.Waypoints
```

```
ans = 2x3
```

```

    1.8674    1.0203    0.7032
    2.3154   -0.3207    0.0999

```

```
traj.TimeOfArrival
```

```
ans = 2x1
```

```

    0
    2

```

Perturb Accuracy of insSensor

Create an insSensor object.

```
sensor = insSensor
```

```

sensor =
  insSensor with properties:

```

```

    MountingLocation: [0 0 0]
    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=1x3 cell array
    {[0.1000]}    {[0.2000]}    {[0.3000]}
```

```
probabilities = [1/3 1/3 1/3]
```

```
probabilities = 1x3
    0.3333    0.3333    0.3333
```

```
perturbations(sensor, 'RollAccuracy', 'Selection', values, probabilities)
```

```
ans=7x3 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]
    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

See Also

perturbations

Introduced in R2020b

perturbations

Perturbation defined on object

Syntax

```
perturbs = perturbations(obj)
perturbs = perturbations(obj,property)
perturbs = perturbations(obj,property,'None')
perturbs = perturbations(obj,property,'Selection',values,probabilities)
perturbs = perturbations(obj,property,'Normal',mean,deviation)
perturbs = perturbations(obj,property,'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,'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=2x3 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]  
        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=1x3 cell array  
    {[0.1000]}    {[0.2000]}    {[0.3000]}
```

```
probabilities = [1/3 1/3 1/3]
```

```
probabilities = 1x3  
    0.3333    0.3333    0.3333
```

```
perturbations(sensor, 'RollAccuracy', 'Selection', values, probabilities)
```

```
ans=7x3 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]
    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=2x3 table
Property
```

Property	Type	Value	
"Waypoints"	"Normal"	{[1]}	{[1]}
"TimeOfArrival"	"None"	{[NaN]}	{[NaN]}

```
perturbs2 = perturbations(traj, 'TimeOfArrival', 'Selection', {[0;1],[0;2]})
```

```
perturbs2=2x3 table
Property
```

Property	Type	Value	
"Waypoints"	"Normal"	{[1]}	{[1]}
"TimeOfArrival"	"Selection"	{1x2 cell}	{[0.5000 0.5000]}

Perturb the trajectory.

```
offsets = perturb(traj)
```

```
offsets=2x1 struct array with fields:
```

```
Property
Offset
PerturbedValue
```

The Waypoints property and the TimeOfArrival property have changed.

```
traj.Waypoints
```

```
ans = 2x3
```

```
1.8674 1.0203 0.7032
2.3154 -0.3207 0.0999
```

```
traj.TimeOfArrival
```

```
ans = 2x1
```

```
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 $\{x_1, x_2, \dots, x_n\}$ and $\{p_1, p_2, \dots, p_n\}$, where the probability of drawing x_i is p_i ($i = 1, 2, \dots, n$).

mean — Mean of normal distribution

scalar | vector | matrix

Mean of 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 distribution

nonnegative scalar | vector of nonnegative scalar | matrix of nonnegative scalar

Standard deviation of 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.

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", "Uniform", or "Custom".
- **Value** — Perturbation values, returned as a cell array.

See Also

perturb

Introduced in R2020b

Functions

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 — List of DTED files

string scalar | character vector | cell array of character vectors

List of DTED files, specified as a string scalar, a character vector or a cell array of character vectors.

Note If you specify multiple files, they must combine to define a complete rectangular geographic region. If not, you must set the name-value pair `'FillMissing'` to `'true'`.

Data Types: `char` | `string`

Name-Value Pair Arguments

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

Example: `'FillMissing',true`

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`

See Also

`addMesh` | `removeCustomTerrain` | `uavScenario`

Introduced in R2021a

angdiff

Difference between two angles

Syntax

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

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

Description

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

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

Examples

Calculate Difference Between Two Angles

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

```
d = 3.1416
```

Calculate Difference Between Two Angle Arrays

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

```
d = 1×3
```

```
1.5708 -0.7854 -3.1416
```

Calculate Angle Differences of Adjacent Elements

```
angles = [pi pi/2 pi/4 pi/2];
```

```
d = angdiff(angles)
```

```
d = 1×3
```

-1.5708 -0.7854 0.7854

Input Arguments

alpha — Angle in radians

scalar | vector | matrix | multidimensional array

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

Example: `pi/2`

beta — Angle in radians

scalar | vector | matrix | multidimensional array

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

Example: `pi/2`

Output Arguments

delta — Difference between two angles

scalar | vector | matrix | multidimensional array

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

Extended Capabilities

C/C++ Code Generation

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

Introduced in R2015a

axang2quat

Convert axis-angle rotation to quaternion

Syntax

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

Description

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

Examples

Convert Axis-Angle Rotation to Quaternion

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

```
quat = 1×4
```

```
    0.7071    0.7071         0         0
```

Input Arguments

axang — Rotation given in axis-angle form

n-by-4 matrix

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

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

Output Arguments

quat — Unit quaternion

n-by-4 matrix

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

Example: `[0.7071 0.7071 0 0]`

Extended Capabilities

C/C++ Code Generation

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

See Also

quat2axang

Introduced in R2015a

axang2rotm

Convert axis-angle rotation to rotation matrix

Syntax

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

Description

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

Examples

Convert Axis-Angle Rotation to Rotation Matrix

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

```
rotm = 3×3
```

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

Input Arguments

axang — Rotation given in axis-angle form

n-by-4 matrix

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

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

Output Arguments

rotm — Rotation matrix

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

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

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

Extended Capabilities

C/C++ Code Generation

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

See Also

rotm2axang

Introduced in R2015a

axang2tform

Convert axis-angle rotation to homogeneous transformation

Syntax

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

Description

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

Examples

Convert Axis-Angle Rotation to Homogeneous Transformation

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

```
tform = 4×4
```

```

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

Input Arguments

axang — Rotation given in axis-angle form

n-by-4 matrix

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

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

Output Arguments

tform — Homogeneous transformation

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

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

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

Extended Capabilities

C/C++ Code Generation

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

See Also

tform2axang

Introduced in R2015a

cart2hom

Convert Cartesian coordinates to homogeneous coordinates

Syntax

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

Description

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

Examples

Convert 3-D Cartesian Points to Homogeneous Coordinates

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

```
h = 2×4
```

```
    0.8147    0.1270    0.6324    1.0000  
    0.9058    0.9134    0.0975    1.0000
```

Input Arguments

cart — Cartesian coordinates

n-by- $(k-1)$ matrix

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

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

Output Arguments

hom — Homogeneous points

n-by-*k* matrix

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

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

Extended Capabilities

C/C++ Code Generation

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

See Also

hom2cart

Introduced in R2015a

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 `createSensorAdaptorTemplate` 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);
```

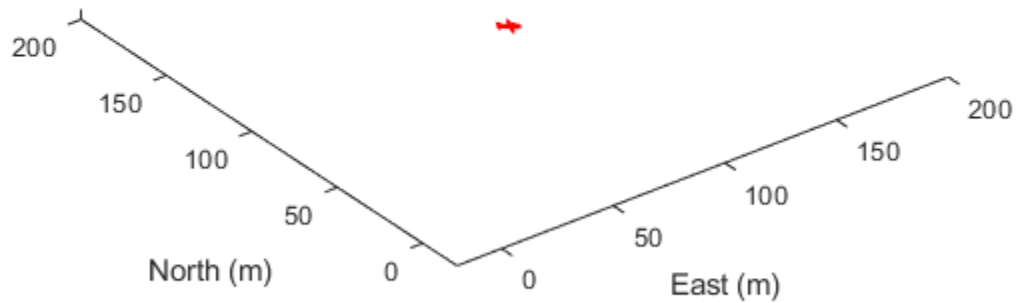
Setup the scenario.

```
setup(scenario);
```

Run the simulation using the `advance` function. Update the sensors and record the data.

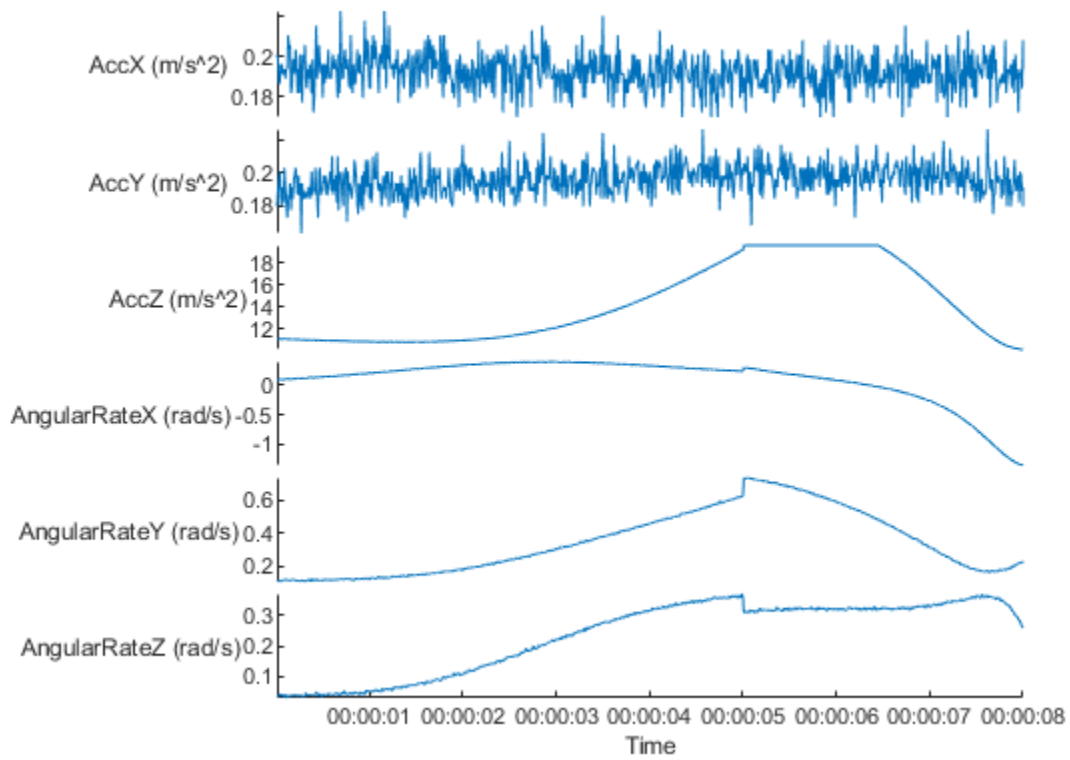
```
updateCounter = 0;
while true
    % Advance scenario.
    isRunning = advance(scenario);
    updateCounter = updateCounter + 1;
    % Update sensors and read IMU data.
    updateSensors(scenario);
    [isUpdated, t, acc, gyro] = read(imu);
    % Store data in structure.
    simData.Time = [simData.Time; 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)"]);
```



See Also

`uav.SensorAdaptor`

Introduced in R2021a

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.

Data Types: double

lla0 — Origin of local ENU system in geodetic coordinates

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

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

Data Types: double

method — Transformation method

'flat' | 'ellipsoid'

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

The flat Earth transformation method has these limitations:

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

Data Types: char | string

Output Arguments

lla — Geodetic coordinates

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

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

Data Types: double

Extended Capabilities

C/C++ Code Generation

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

See Also

Functions

lla2enu | lla2ned | ned2lla

Introduced in R2020b

eul2quat

Convert Euler angles to quaternion

Syntax

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

Description

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

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

Examples

Convert Euler Angles to Quaternion

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

qZYX = 1×4

    0.7071         0    0.7071         0
```

Convert Euler Angles to Quaternion Using Default ZYZ Axis Order

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

qZYZ = 1×4

    0.7071         0         0    0.7071
```

Input Arguments

eul — Euler rotation angles

n-by-3 matrix

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

Example: [0 0 1.5708]

sequence — Axis rotation sequence

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

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

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

Data Types: `string` | `char`

Output Arguments**quat — Unit quaternion**

n-by-4 matrix

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

Example: [0.7071 0.7071 0 0]

Extended Capabilities**C/C++ Code Generation**

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

See Also

`quat2eul` | `quaternion`

Introduced in R2015a

eul2rotm

Convert Euler angles to rotation matrix

Syntax

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

Description

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

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

Examples

Convert Euler Angles to Rotation Matrix

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

```
rotmZYX = 3×3
```

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

Convert Euler Angles to Rotation Matrix Using ZYZ Axis Order

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

```
rotmZYZ = 3×3
```

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

Input Arguments

eul — Euler rotation angles

n-by-3 matrix

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

Example: [0 0 1.5708]

sequence — Axis rotation sequence

"ZYX" (default) | "YZZ" | "XYZ"

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

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

Data Types: `string` | `char`

Output Arguments

rotm — Rotation matrix

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

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

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

Extended Capabilities

C/C++ Code Generation

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

See Also

`rotm2eul`

Introduced in R2015a

eul2tform

Convert Euler angles to homogeneous transformation

Syntax

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

Description

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

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

Examples

Convert Euler Angles to Homogeneous Transformation Matrix

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

tformZYX = 4×4

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

Convert Euler Angles to Homogeneous Transformation Matrix Using ZYZ Axis Order

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

tformZYZ = 4×4

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

Input Arguments

eul — Euler rotation angles

n-by-3 matrix

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

Example: `[0 0 1.5708]`

sequence — Axis rotation sequence

"ZYX" (default) | "YZZ" | "XYZ"

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

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

Data Types: `string` | `char`

Output Arguments

tform — Homogeneous transformation

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

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

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

Extended Capabilities

C/C++ Code Generation

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

See Also

`tform2eul`

Introduced in R2015a

hom2cart

Convert homogeneous coordinates to Cartesian coordinates

Syntax

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

Description

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

Examples

Convert Homogeneous Points to 3-D Cartesian Points

```
h = [0.2785 0.9575 0.1576 0.5; 0.5469 0.9649 0.9706 0.5];
```

```
c = hom2cart(h)
```

```
c = 2×3
```

```
    0.5570    1.9150    0.3152  
    1.0938    1.9298    1.9412
```

Input Arguments

hom — Homogeneous points

n-by-*k* matrix

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

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

Output Arguments

cart — Cartesian coordinates

n-by-(*k*-1) matrix

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

Example: [0.8147 0.1270 0.6324; 0.9058 0.9134 0.0975]

Extended Capabilities

C/C++ Code Generation

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

See Also

cart2hom

Introduced in R2015a

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.

Data Types: double

lla0 — Origin of local ENU system in geodetic coordinates

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

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

Data Types: double

method — Transformation method

'flat' | 'ellipsoid'

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

The flat Earth transformation method has these limitations:

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

Data Types: char | string

Output Arguments

xyzENU — Local ENU Cartesian coordinates

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

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

Data Types: double

Extended Capabilities

C/C++ Code Generation

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

See Also

Functions

enu2lla | lla2ned | ned2lla

Introduced in R2020b

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.

Data Types: `double`

lla0 — Origin of local NED system in geodetic coordinates

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

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

Data Types: `double`

method — Transformation method

`'flat'` | `'ellipsoid'`

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

The flat Earth transformation method has these limitations:

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

Data Types: `char` | `string`

Output Arguments

xyzNED — Local NED Cartesian coordinates

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

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

Data Types: `double`

Extended Capabilities

C/C++ Code Generation

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

See Also

Functions

`enu2lla` | `lla2enu` | `ned2lla`

Introduced in R2020b

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  
103 ×
```

```
    0.0460    0.0077    4.5254
```

Input Arguments

xyzNED — Local NED Cartesian coordinates

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

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

Data Types: `double`

lla0 — Origin of local NED system in geodetic coordinates

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

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

Data Types: `double`

method — Transformation method

'flat' | 'ellipsoid'

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

The flat Earth transformation method has these limitations:

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

Data Types: `char` | `string`

Output Arguments

lla — Geodetic coordinates

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

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

Data Types: `double`

Extended Capabilities

C/C++ Code Generation

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

See Also

Functions

`enu2lla` | `lla2enu` | `lla2ned`

Introduced in R2020b

plotTransforms

Plot 3-D transforms from translations and rotations

Syntax

```
ax = plotTransforms(translations,rotations)
ax = plotTransforms(translations,rotations,Name,Value)
```

Description

`ax = plotTransforms(translations,rotations)` draws transform frames in a 3-D figure window using the specified translations and rotations. The z-axis always points upward.

`ax = plotTransforms(translations,rotations,Name,Value)` specifies additional options using name-value pair arguments. Specify multiple name-value pairs to set multiple options.

Input Arguments

translations — xyz-positions

[x y z] vector | matrix of [x y z] vectors

xyz-positions specified as a vector or matrix of [x y z] vectors. Each row represents a new frame to plot with a corresponding orientation in `rotations`.

Example: [1 1 1; 2 2 2]

rotations — Rotations of xyz-positions

quaternion array | matrix of [w x y z] quaternion vectors

Rotations of xyz-positions specified as a quaternion array or n -by-4 matrix of [w x y z] quaternion vectors. Each element of the array or each row of the matrix represents the rotation of the xyz-positions specified in `translations`.

Example: [1 1 1 0; 1 3 5 0]

Name-Value Pair Arguments

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

Example: 'FrameSize',5

FrameSize — Size of frames and attached meshes

positive numeric scalar

Size of frame and attached meshes, specified as positive numeric scalar.

InertialZDirection — Direction of positive z-axis of inertial frame

"up" (default) | "down"

Direction of the positive z-axis of inertial frame, specified as either "up" or "down". In the plot, the positive z-axis always points up.

MeshFilePath — File path of mesh file attached to frames

character vector | string scalar

File path of mesh file attached to frames, specified as either a character vector or string scalar. The mesh is attached to each plotted frame at the specified position and orientation. Provided `.stl` are

- "fixedwing.stl"
- "multirotor.stl"
- "groundvehicle.stl"

Example: 'fixedwing.stl'

MeshColor — Color of attached mesh

"red" (default) | RGB triplet | string scalar

Color of attached mesh, specified as an RGB triple or string scalar.

Example: [0 0 1] or "green"

Parent — Axes used to plot transforms

Axes object | UIAxes object

Axes used to plot the pose graph, specified as the comma-separated pair consisting of 'Parent' and either an Axes or UIAxes object. See `axes` or `uiaxes`.

Output Arguments

ax — Axes used to plot transforms

Axes object | UIAxes object

Axes used to plot the pose graph, specified as the comma-separated pair consisting of 'Parent' and either an Axes or UIAxesobject. See `axes` or `uiaxes`.

See Also

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

Introduced in R2018b

quat2axang

Convert quaternion to axis-angle rotation

Syntax

```
axang = quat2axang(quat)
```

Description

`axang = quat2axang(quat)` converts a quaternion, `quat`, to the equivalent axis-angle rotation, `axang`.

Examples

Convert Quaternion to Axis-Angle Rotation

```
quat = [0.7071 0.7071 0 0];  
axang = quat2axang(quat)
```

```
axang = 1×4
```

```
    1.0000         0         0    1.5708
```

Input Arguments

quat — Unit quaternion

n-by-4 matrix | *n*-element vector of quaternion objects

Unit quaternion, specified as an *n*-by-4 matrix or *n*-element vector of quaternion objects containing *n* quaternions. If the input is a matrix, each row is a quaternion vector of the form $q = [w \ x \ y \ z]$, with *w* as the scalar number.

Example: `[0.7071 0.7071 0 0]`

Output Arguments

axang — Rotation given in axis-angle form

n-by-4 matrix

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

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

Extended Capabilities

C/C++ Code Generation

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

See Also

[axang2quat](#) | [quaternion](#)

Introduced in R2015a

quat2eul

Convert quaternion to Euler angles

Syntax

```
eul = quat2eul(quat)
eul = quat2eul(quat, sequence)
```

Description

`eul = quat2eul(quat)` converts a quaternion rotation, `quat`, to the corresponding Euler angles, `eul`. The default order for Euler angle rotations is "ZYX".

`eul = quat2eul(quat, sequence)` converts a quaternion into Euler angles. The Euler angles are specified in the axis rotation sequence, `sequence`. The default order for Euler angle rotations is "ZYX".

Examples

Convert Quaternion to Euler Angles

```
quat = [0.7071 0.7071 0 0];
eulZYX = quat2eul(quat)

eulZYX = 1×3
         0         0    1.5708
```

Convert Quaternion to Euler Angles Using ZYZ Axis Order

```
quat = [0.7071 0.7071 0 0];
eulZYZ = quat2eul(quat, 'ZYZ')

eulZYZ = 1×3
    1.5708   -1.5708   -1.5708
```

Input Arguments

quat — Unit quaternion

n-by-4 matrix | *n*-element vector of quaternion objects

Unit quaternion, specified as an *n*-by-4 matrix or *n*-element vector of quaternion objects containing *n* quaternions. If the input is a matrix, each row is a quaternion vector of the form $q = [w \ x \ y \ z]$, with *w* as the scalar number.

Example: [0.7071 0.7071 0 0]

sequence — Axis rotation sequence

"ZYX" (default) | "YZZ" | "XYZ"

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

- "ZYX" (default) - The order of rotation angles is z-axis, y-axis, x-axis.
- "YZZ" - The order of rotation angles is z-axis, y-axis, z-axis.
- "XYZ" - The order of rotation angles is x-axis, y-axis, z-axis.

Data Types: string | char

Output Arguments

eul — Euler rotation angles

n-by-3 matrix

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

Example: [0 0 1.5708]

Extended Capabilities

C/C++ Code Generation

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

See Also

eul2quat | quaternion

Introduced in R2015a

quat2rotm

Convert quaternion to rotation matrix

Syntax

```
rotm = quat2rotm(quat)
```

Description

`rotm = quat2rotm(quat)` converts a quaternion `quat` to an orthonormal rotation matrix, `rotm`. When using the rotation matrix, premultiply it with the coordinates to be rotated (as opposed to postmultiplying).

Examples

Convert Quaternion to Rotation Matrix

```
quat = [0.7071 0.7071 0 0];
rotm = quat2rotm(quat)
```

```
rotm = 3×3
```

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

Input Arguments

quat — Unit quaternion

n-by-4 matrix | *n*-element vector of quaternion objects

Unit quaternion, specified as an *n*-by-4 matrix or *n*-element vector of quaternion objects containing *n* quaternions. If the input is a matrix, each row is a quaternion vector of the form $q = [w \ x \ y \ z]$, with *w* as the scalar number.

Example: `[0.7071 0.7071 0 0]`

Output Arguments

rotm — Rotation matrix

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

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

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

Extended Capabilities

C/C++ Code Generation

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

See Also

quaternion | rotm2quat

Introduced in R2015a

quat2tform

Convert quaternion to homogeneous transformation

Syntax

```
tform = quat2tform(quat)
```

Description

`tform = quat2tform(quat)` converts a quaternion, `quat`, to a homogeneous transformation matrix, `tform`. When using the transformation matrix, premultiply it with the coordinates to be transformed (as opposed to postmultiplying).

Examples

Convert Quaternion to Homogeneous Transformation

```
quat = [0.7071 0.7071 0 0];
tform = quat2tform(quat)
```

```
tform = 4×4
```

```

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

Input Arguments

quat — Unit quaternion

n -by-4 matrix | n -element vector of quaternion objects

Unit quaternion, specified as an n -by-4 matrix or n -element vector of quaternion objects containing n quaternions. If the input is a matrix, each row is a quaternion vector of the form $q = [w \ x \ y \ z]$, with w as the scalar number.

Example: `[0.7071 0.7071 0 0]`

Output Arguments

tform — Homogeneous transformation

4-by-4-by- n matrix

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

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

Extended Capabilities

C/C++ Code Generation

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

See Also

quaternion | tform2quat

Introduced in R2015a

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`.

Examples

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

See Also

`addCustomTerrain`

Introduced in R2021a

rotm2axang

Convert rotation matrix to axis-angle rotation

Syntax

```
axang = rotm2axang(rotm)
```

Description

`axang = rotm2axang(rotm)` converts a rotation given as an orthonormal rotation matrix, `rotm`, to the corresponding axis-angle representation, `axang`. The input rotation matrix must be in the premultiply form for rotations.

Examples

Convert Rotation Matrix to Axis-Angle Rotation

```
rotm = [1 0 0 ; 0 -1 0; 0 0 -1];
axang = rotm2axang(rotm)

axang = 1x4

    1.0000         0         0    3.1416
```

Input Arguments

rotm — Rotation matrix

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

Rotation matrix, specified as a 3-by-3-by-*n* matrix containing *n* rotation matrices. Each rotation matrix has a size of 3-by-3 and must be orthonormal. The input rotation matrix must be in the premultiply form for rotations.

Note Rotation matrices that are slightly non-orthonormal can give complex outputs. Consider validating your matrix before inputting to the function.

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

Output Arguments

axang — Rotation given in axis-angle form

n-by-4 matrix

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

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

Extended Capabilities

C/C++ Code Generation

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

See Also

`axang2rotm`

Introduced in R2015a

rotm2eul

Convert rotation matrix to Euler angles

Syntax

```
eul = rotm2eul(rotm)
eul = rotm2eul(rotm,sequence)
```

Description

`eul = rotm2eul(rotm)` converts a rotation matrix, `rotm`, to the corresponding Euler angles, `eul`. The input rotation matrix must be in the premultiply form for rotations. The default order for Euler angle rotations is "ZYX".

`eul = rotm2eul(rotm,sequence)` converts a rotation matrix to Euler angles. The Euler angles are specified in the axis rotation sequence, `sequence`. The default order for Euler angle rotations is "ZYX".

Examples

Convert Rotation Matrix to Euler Angles

```
rotm = [0 0 1; 0 1 0; -1 0 0];
eulZYX = rotm2eul(rotm)
```

```
eulZYX = 1×3
         0    1.5708    0
```

Convert Rotation Matrix to Euler Angles Using ZYZ Axis Order

```
rotm = [0 0 1; 0 -1 0; -1 0 0];
eulZYZ = rotm2eul(rotm,'ZYZ')
```

```
eulZYZ = 1×3
    -3.1416    -1.5708    -3.1416
```

Input Arguments

rotm — Rotation matrix
3-by-3-by-*n* matrix

Rotation matrix, specified as a 3-by-3-by- n matrix containing n rotation matrices. Each rotation matrix has a size of 3-by-3 and is orthonormal. The input rotation matrix must be in the premultiply form for rotations.

Note Rotation matrices that are slightly non-orthonormal can give complex outputs. Consider validating your matrix before inputting to the function.

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

sequence — Axis rotation sequence

"ZYX" (default) | "YZ" | "XYZ"

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

- "ZYX" (default) - The order of rotation angles is z-axis, y-axis, x-axis.
- "YZ" - The order of rotation angles is z-axis, y-axis, z-axis.
- "XYZ" - The order of rotation angles is x-axis, y-axis, z-axis.

Data Types: `string` | `char`

Output Arguments

eul — Euler rotation angles

n -by-3 matrix

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

Example: `[0 0 1.5708]`

Extended Capabilities

C/C++ Code Generation

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

See Also

`eul2rotm`

Introduced in R2015a

rotm2quat

Convert rotation matrix to quaternion

Syntax

```
quat = rotm2quat(rotm)
```

Description

`quat = rotm2quat(rotm)` converts a rotation matrix, `rotm`, to the corresponding unit quaternion representation, `quat`. The input rotation matrix must be in the premultiply form for rotations.

Examples

Convert Rotation Matrix to Quaternion

```
rotm = [0 0 1; 0 1 0; -1 0 0];
quat = rotm2quat(rotm)
```

```
quat = 1×4
```

```
    0.7071         0    0.7071         0
```

Input Arguments

rotm — Rotation matrix

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

Rotation matrix, specified as a 3-by-3-by-*n* matrix containing *n* rotation matrices. Each rotation matrix has a size of 3-by-3 and is orthonormal. The input rotation matrix must be in the premultiply form for rotations.

Note Rotation matrices that are slightly non-orthonormal can give complex outputs. Consider validating your matrix before inputting to the function.

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

Output Arguments

quat — Unit quaternion

n-by-4 matrix

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

Example: [0.7071 0.7071 0 0]

Extended Capabilities

C/C++ Code Generation

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

See Also

quat2rotm

Introduced in R2015a

rotm2tform

Convert rotation matrix to homogeneous transformation

Syntax

```
tform = rotm2tform(rotm)
```

Description

`tform = rotm2tform(rotm)` converts the rotation matrix, `rotm`, into a homogeneous transformation matrix, `tform`. The input rotation matrix must be in the premultiply form for rotations. When using the transformation matrix, premultiply it with the coordinates to be transformed (as opposed to postmultiplying).

Examples

Convert Rotation Matrix to Homogeneous Transformation

```
rotm = [1 0 0 ; 0 -1 0; 0 0 -1];
tform = rotm2tform(rotm)
```

```
tform = 4x4
```

```

1     0     0     0
0    -1     0     0
0     0    -1     0
0     0     0     1
```

Input Arguments

rotm — Rotation matrix

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

Rotation matrix, specified as a 3-by-3-by-*n* matrix containing *n* rotation matrices. Each rotation matrix has a size of 3-by-3 and is orthonormal. The input rotation matrix must be in the premultiply form for rotations.

Note Rotation matrices that are slightly non-orthonormal can give complex outputs. Consider validating your matrix before inputting to the function.

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

Output Arguments

tform — Homogeneous transformation

4-by-4-by- n matrix

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

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

Extended Capabilities

C/C++ Code Generation

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

See Also

tform2rotm

Introduced in R2015a

tform2axang

Convert homogeneous transformation to axis-angle rotation

Syntax

```
axang = tform2axang(tform)
```

Description

`axang = tform2axang(tform)` converts the rotational component of a homogeneous transformation, `tform`, to an axis-angle rotation, `axang`. The translational components of `tform` are ignored. The input homogeneous transformation must be in the premultiply form for transformations.

Examples

Convert Homogeneous Transformation to Axis-Angle Rotation

```
tform = [1 0 0 0; 0 0 -1 0; 0 1 0 0; 0 0 0 1];
axang = tform2axang(tform)
```

```
axang = 1×4
```

```
    1.0000         0         0    1.5708
```

Input Arguments

tform — Homogeneous transformation

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

Homogeneous transformation, specified by a 4-by-4-by-*n* matrix of *n* homogeneous transformations. The input homogeneous transformation must be in the premultiply form for transformations.

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

Output Arguments

axang — Rotation given in axis-angle form

n-by-4 matrix

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

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

Extended Capabilities

C/C++ Code Generation

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

See Also

`axang2tform`

Introduced in R2015a

tform2eul

Extract Euler angles from homogeneous transformation

Syntax

```
eul = tform2eul(tform)
eul = tform2eul(tform, sequence)
```

Description

`eul = tform2eul(tform)` extracts the rotational component from a homogeneous transformation, `tform`, and returns it as Euler angles, `eul`. The translational components of `tform` are ignored. The input homogeneous transformation must be in the premultiply form for transformations. The default order for Euler angle rotations is "ZYX".

`eul = tform2eul(tform, sequence)` extracts the Euler angles, `eul`, from a homogeneous transformation, `tform`, using the specified rotation sequence, `sequence`. The default order for Euler angle rotations is "ZYX".

Examples

Extract Euler Angles from Homogeneous Transformation Matrix

```
tform = [1 0 0 0.5; 0 -1 0 5; 0 0 -1 -1.2; 0 0 0 1];
eulZYX = tform2eul(tform)
```

```
eulZYX = 1×3
         0         0    3.1416
```

Extract Euler Angles from Homogeneous Transformation Matrix Using ZYZ Rotation

```
tform = [1 0 0 0.5; 0 -1 0 5; 0 0 -1 -1.2; 0 0 0 1];
eulZYZ = tform2eul(tform, 'ZYZ')
```

```
eulZYZ = 1×3
         0   -3.1416    3.1416
```

Input Arguments

tform — Homogeneous transformation
4-by-4-by-*n* matrix

Homogeneous transformation, specified by a 4-by-4-by- n matrix of n homogeneous transformations. The input homogeneous transformation must be in the premultiply form for transformations.

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

sequence — Axis rotation sequence

"ZYX" (default) | "YZZ" | "XYZ"

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

- "ZYX" (default) - The order of rotation angles is z-axis, y-axis, x-axis.
- "YZZ" - The order of rotation angles is z-axis, y-axis, z-axis.
- "XYZ" - The order of rotation angles is x-axis, y-axis, z-axis.

Data Types: `string` | `char`

Output Arguments

eul — Euler rotation angles

n -by-3 matrix

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

Example: `[0 0 1.5708]`

Extended Capabilities

C/C++ Code Generation

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

See Also

`eul2tform`

Introduced in R2015a

tform2quat

Extract quaternion from homogeneous transformation

Syntax

```
quat = tform2quat(tform)
```

Description

`quat = tform2quat(tform)` extracts the rotational component from a homogeneous transformation, `tform`, and returns it as a quaternion, `quat`. The translational components of `tform` are ignored. The input homogeneous transformation must be in the premultiply form for transformations.

Examples

Extract Quaternion from Homogeneous Transformation

```
tform = [1 0 0 0; 0 -1 0 0; 0 0 -1 0; 0 0 0 1];
quat = tform2quat(tform)
```

```
quat = 1×4
```

```
    0    1    0    0
```

Input Arguments

tform — Homogeneous transformation

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

Homogeneous transformation, specified by a 4-by-4-by-*n* matrix of *n* homogeneous transformations. The input homogeneous transformation must be in the premultiply form for transformations.

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

Output Arguments

quat — Unit quaternion

n-by-4 matrix

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

Example: `[0.7071 0.7071 0 0]`

Extended Capabilities

C/C++ Code Generation

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

See Also

`quat2tform`

Introduced in R2015a

tform2rotm

Extract rotation matrix from homogeneous transformation

Syntax

```
rotm = tform2rotm(tform)
```

Description

`rotm = tform2rotm(tform)` extracts the rotational component from a homogeneous transformation, `tform`, and returns it as an orthonormal rotation matrix, `rotm`. The translational components of `tform` are ignored. The input homogeneous transformation must be in the pre-multiply form for transformations. When using the rotation matrix, premultiply it with the coordinates to be rotated (as opposed to postmultiplying).

Examples

Convert Homogeneous Transformation to Rotation Matrix

```
tform = [1 0 0 0; 0 -1 0 0; 0 0 -1 0; 0 0 0 1];
rotm = tform2rotm(tform)
```

```
rotm = 3×3
```

```
    1    0    0
    0   -1    0
    0    0   -1
```

Input Arguments

tform — Homogeneous transformation

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

Homogeneous transformation matrix, specified by a 4-by-4-by-*n* matrix of *n* homogeneous transformations. The input homogeneous transformation must be in the pre-multiply form for transformations.

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

Output Arguments

rotm — Rotation matrix

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

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

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

Extended Capabilities

C/C++ Code Generation

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

See Also

`rotm2tform`

Introduced in R2015a

tform2trvec

Extract translation vector from homogeneous transformation

Syntax

```
trvec = tform2trvec(tform)
```

Description

`trvec = tform2trvec(tform)` extracts the Cartesian representation of translation vector, `trvec`, from a homogeneous transformation, `tform`. The rotational components of `tform` are ignored. The input homogeneous transformation must be in the premultiply form for transformations.

Examples

Extract Translation Vector from Homogeneous Transformation

```
tform = [1 0 0 0.5; 0 -1 0 5; 0 0 -1 -1.2; 0 0 0 1];
trvec = tform2trvec(tform)
```

```
trvec = 1×3
```

```
    0.5000    5.0000   -1.2000
```

Input Arguments

tform — Homogeneous transformation

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

Homogeneous transformation, specified by a 4-by-4-by-*n* matrix of *n* homogeneous transformations. The input homogeneous transformation must be in the premultiply form for transformations.

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

Output Arguments

trvec — Cartesian representation of a translation vector

n-by-3 matrix

Cartesian representation of a translation vector, returned as an *n*-by-3 matrix containing *n* translation vectors. Each vector is of the form $t = [x \ y \ z]$.

Example: `[0.5 6 100]`

Extended Capabilities

C/C++ Code Generation

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

See Also

`trvec2tform`

Introduced in R2015a

trvec2tform

Convert translation vector to homogeneous transformation

Syntax

```
tform = trvec2tform(trvec)
```

Description

`tform = trvec2tform(trvec)` converts the Cartesian representation of a translation vector, `trvec`, to the corresponding homogeneous transformation, `tform`. When using the transformation matrix, premultiply it with the coordinates to be transformed (as opposed to postmultiplying).

Examples

Convert Translation Vector to Homogeneous Transformation

```
trvec = [0.5 6 100];
tform = trvec2tform(trvec)
```

```
tform = 4×4
```

```

1.0000    0    0    0.5000
    0    1.0000    0    6.0000
    0    0    1.0000  100.0000
    0    0    0    1.0000
```

Input Arguments

trvec — Cartesian representation of a translation vector

n-by-3 matrix

Cartesian representation of a translation vector, specified as an *n*-by-3 matrix containing *n* translation vectors. Each vector is of the form $t = [x \ y \ z]$.

Example: `[0.5 6 100]`

Output Arguments

tform — Homogeneous transformation

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

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

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

Extended Capabilities

C/C++ Code Generation

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

See Also

tform2rvec

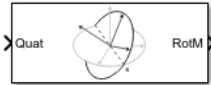
Introduced in R2015a

Blocks

Coordinate Transformation Conversion

Convert to a specified coordinate transformation representation

Library: Robotics System Toolbox / Utilities
 Navigation Toolbox / Utilities
 ROS Toolbox / Utilities
 UAV Toolbox / Utilities



Description

The Coordinate Transformation Conversion block converts a coordinate transformation from the input representation to a specified output representation. The input and output representations use the following forms:

- Axis-Angle (AxAng) - [x y z theta]
- Euler Angles (Eul) - [z y x], [z y z], or [x y z]
- Homogeneous Transformation (TForm) - 4-by-4 matrix
- Quaternion (Quat) - [w x y z]
- Rotation Matrix (RotM) - 3-by-3 matrix
- Translation Vector (TrVec) - [x y z]

All vectors must be **column vectors**.

To accommodate representations that only contain position or orientation information (TrVec or Eul, for example), you can specify two inputs or outputs to handle all transformation information. When you select the Homogeneous Transformation as an input or output, an optional Show TrVec input/output port parameter can be selected on the block mask to toggle the multiple ports.

Ports

Input

Input transformation — Coordinate transformation

column vector | 3-by-3 matrix | 4-by-4 matrix

Input transformation, specified as a coordinate transformation. The following representations are supported:

- Axis-Angle (AxAng) - [x y z theta]
- Euler Angles (Eul) - [z y x], [z y z], or [x y z]
- Homogeneous Transformation (TForm) - 4-by-4 matrix
- Quaternion (Quat) - [w x y z]
- Rotation Matrix (RotM) - 3-by-3 matrix

- Translation Vector (TrVec) - [x y z]

All vectors must be **column vectors**.

To accommodate representations that only contain position or orientation information (TrVec or Eul, for example), you can specify two inputs or outputs to handle all transformation information. When you select the Homogeneous Transformation as an input or output, an optional `Show TrVec input/output port` parameter can be selected on the block mask to toggle the multiple ports.

TrVec – Translation vector

3-element column vector

Translation vector, specified as a 3-element column vector, [x y z], which corresponds to a translation in the x, y, and z axes respectively. This port can be used to input or output the translation information separately from the rotation vector.

Dependencies

You must select Homogeneous Transformation (TForm) for the opposite transformation port to get the option to show the additional TrVec port. Enable the port by clicking `Show TrVec input/output port`.

Output Arguments

Output transformation – Coordinate transformation

column vector | 3-by-3 matrix | 4-by-4 matrix

Output transformation, specified as a coordinate transformation with the specified representation. The following representations are supported:

- Axis-Angle (AxAng) - [x y z theta]
- Euler Angles (Eul) - [z y x], [z y z], or [x y z]
- Homogeneous Transformation (TForm) - 4-by-4 matrix
- Quaternion (Quat) - [w x y z]
- Rotation Matrix (RotM) - 3-by-3 matrix
- Translation Vector (TrVec) - [x y z]

To accommodate representations that only contain position or orientation information (TrVec or Eul, for example), you can specify two inputs or outputs to handle all transformation information. When you select the Homogeneous Transformation as an input or output, an optional `Show TrVec input/output port` parameter can be selected on the block mask to toggle the multiple ports.

TrVec – Translation vector

three-element column vector

Translation vector, specified as a three-element column vector, [x y z], which corresponds to a translation in the x, y, and z axes respectively. This port can be used to input or output the translation information separately from the rotation vector.

Dependencies

You must select Homogeneous Transformation (TForm) for the opposite transformation port to get the option to show the additional TrVec port. Enable the port by clicking `Show TrVec input/output port`.

Parameters

Representation — Input or output representation

Axis-Angle | Euler Angles | Homogeneous Transformation | Rotation Matrix | Translation Vector | Quaternion

Select the representation for both the input and output port for the block. If you are using a transformation with only orientation information, you can also select the Show TrVec input/output port when converting to or from a homogeneous transformation.

Axis rotation sequence — Order of Euler angle axis rotations

ZYX (default) | ZYZ | XYZ

Order of the Euler angle axis rotations, specified as ZYX, ZYZ, or XYZ. The order of the angles in the input or output port `Eul` must match this rotation sequence. The default order ZYX specifies an orientation by:

- Rotating about the initial z-axis
- Rotating about the intermediate y-axis
- Rotating about the second intermediate x-axis

Dependencies

You must select Euler Angles for the Representation input or output parameter. The axis rotation sequence only applies to Euler angle rotations.

Show TrVec input/output port — Toggle TrVec port

off (default) | on

Toggle the TrVec input or output port when you want to specify or receive a separate translation vector for position information along with an orientation representation.

Dependencies

You must select Homogeneous Transformation (TForm) for the opposite transformation port to get the option to show the additional TrVec port.

Extended Capabilities

C/C++ Code Generation

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

See Also

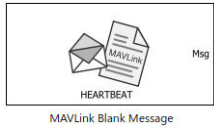
axang2quat | eul2tform | trvec2tform

Introduced in R2017b

MAVLink Blank Message

Create blank MAVLink message bus by specifying payload information and MAVLink message type

Library: 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.

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

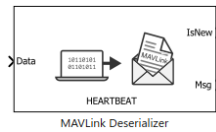
MAVLink Serializer | MAVLink Deserializer

Introduced in R2020b

MAVLink Deserializer

Convert serialized `uint8` MAVLink data stream to Simulink nonvirtual bus

Library: 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

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

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).

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

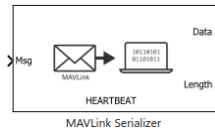
“Exchange Data for MAVLink Microservices like Mission Protocol and Parameter Protocol Using Simulink”

Introduced in R2020b

MAVLink Serializer

Serialize messages of MAVLink packet by converting Simulink nonvirtual bus to `uint8` data stream

Library: 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

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

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.

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

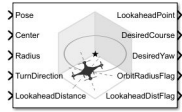
“Exchange Data for MAVLink Microservices like Mission Protocol and Parameter Protocol Using Simulink”

Introduced in R2020b

Orbit Follower

Orbit location of interest using UAV

Library: 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 unmanned aerial vehicle's (UAV's) current pose. 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 UAV's position 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) | multicopter

Type of UAV, specified as either fixed-wing or multicopter.

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.

Extended Capabilities

C/C++ Code Generation

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

See Also

Blocks

UAV Guidance Model | Waypoint Follower

Functions

control | derivative | environment | ode45 | plotTransforms | state

Objects

fixedwing | multirotor | uavOrbitFollower | uavWaypointFollower

Topics

“Approximate High-Fidelity UAV model with UAV Guidance Model block”

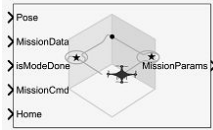
“Tuning Waypoint Follower for Fixed-Wing UAV”

Introduced in R2019a

Path Manager

Compute and execute a UAV autonomous mission

Library: 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 $[x; y; z; \textit{courseAngle}]$. x , y , and z is the current position of the UAV in north-east-down (NED) coordinates specified in meters. $\textit{courseAngle}$ specifies the heading 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: <ul style="list-style-type: none">• 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 (true) or 1 (false).

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	<ul style="list-style-type: none"> [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: <ul style="list-style-type: none"> • 1 — Clockwise turn • -1 — Counterclockwise 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
Multicopter	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 θ 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

Extended Capabilities

C/C++ Code Generation

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

See Also

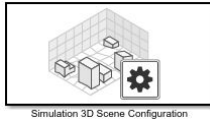
Guidance Model | Orbit Follower | Waypoint Follower

Introduced in R2020b

Simulation 3D Scene Configuration

Scene configuration for 3D simulation environment

Library: UAV Toolbox / 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.

Option	Description
Unreal Executable	<p>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.</p> <p>Select this option to simulate in custom scenes that have been packaged into an executable for faster simulation.</p>
Unreal Editor	<p>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.</p> <p>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.</p>

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 —

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.

Key	Camera Control
L	<p>Toggle a camera lag effect on or off. When you enable the lag effect, the camera view includes:</p> <ul style="list-style-type: none"> • Position lag, based on the vehicle translational acceleration • Rotation lag, based on the vehicle rotational velocity <p>This lag enables improved visualization of overall vehicle acceleration and rotation.</p>
F	<p>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.</p>

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`.

Weather

Override scene weather — Control the scene weather and sun position

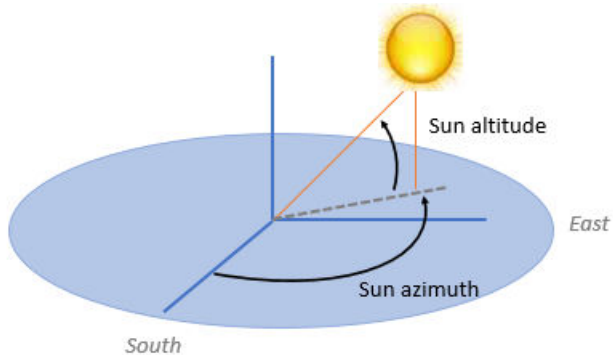
off (default) | on

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.

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.

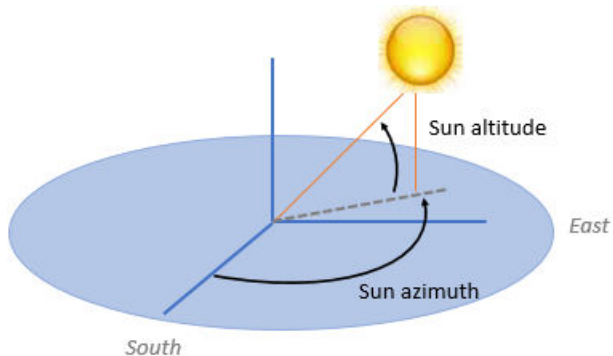


Use the **Sun altitude** and **Sun azimuth** parameters to control the time of day in the scene. For example, to specify sunrise in the north, set **Sun altitude** to 0 deg and **Sun azimuth** to 180 deg.

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.

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.

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.

Fog density – Unreal Editor Set Fog Density and Set Start Distance target values

0 (default) | any value between 0 and 100

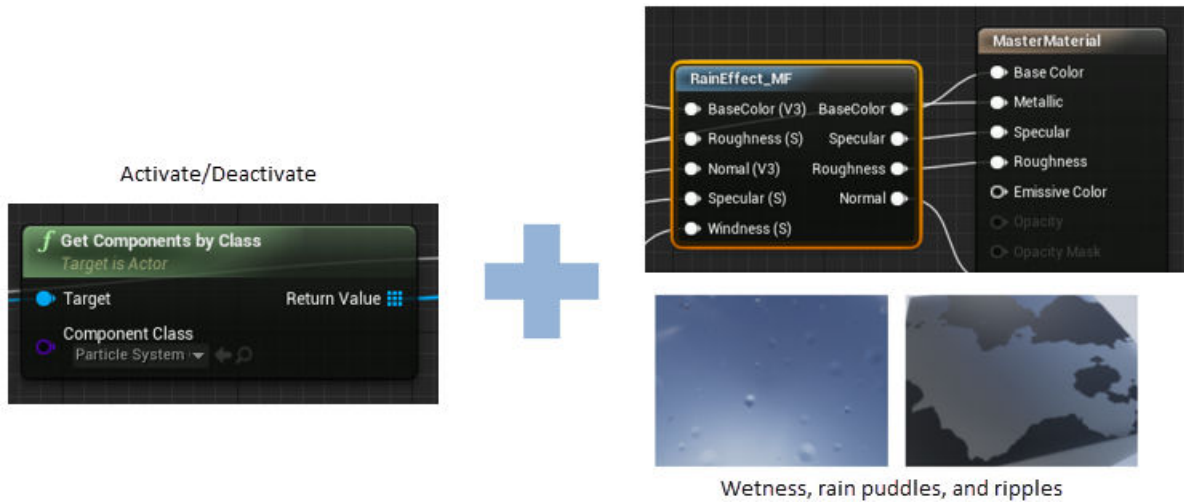
Parameter that corresponds to the Unreal Editor **Set Fog Density** and **Set Start Distance** target values, in percent.



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.

More About

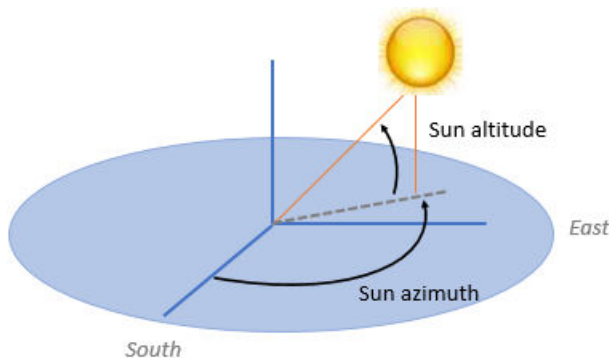
Sun Position and Weather

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.



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.



Fog

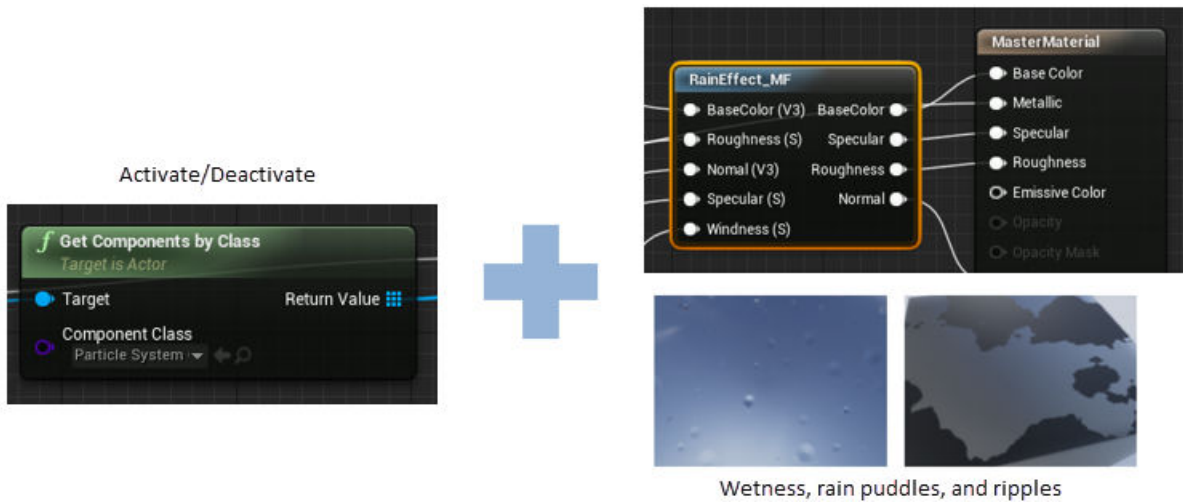
Use **Fog density** to control fog in the scene. **Fog density** corresponds to the Unreal Editor **Set Fog Density**.



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.



Extended Capabilities

C/C++ Code Generation

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

See Also

Simulation 3D Fisheye Camera | Simulation 3D UAV Vehicle | Simulation 3D Camera | Simulation 3D Lidar

Topics

“Unreal Engine Simulation for Unmanned Aerial Vehicles”

“How Unreal Engine Simulation for UAVs Works”

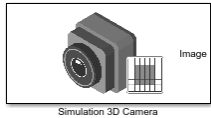
“Coordinate Systems for Unreal Engine Simulation in UAV Toolbox”

Introduced in R2020b

Simulation 3D Camera

Camera sensor model with lens in 3D simulation environment

Library: UAV Toolbox / Simulation 3D



Description



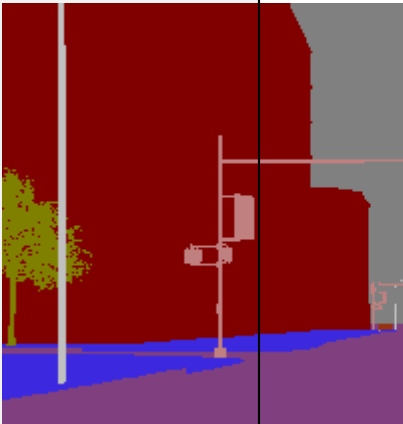
The Simulation 3D Camera block provides an interface to a camera with a lens in a 3D simulation environment. This environment is rendered using the Unreal Engine from Epic Games. The sensor is based on the ideal pinhole camera model, with a lens added to represent a full camera model, including lens distortion. This camera model supports a field of view of up to 150 degrees. For more details, see “Algorithms” on page 4-47.

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	
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

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	<i>Not used</i>
3	Other
4	<i>Not used</i>

ID	Type
5	Pole
6	<i>Not used</i>
7	Road
8	Sidewalk
9	Vegetation
10	Vehicle
11	<i>Not used</i>
12	Generic traffic sign
13	Stop sign
14	Yield sign
15	Speed limit sign
16	Weight limit sign
17 - 18	<i>Not used</i>
19	Left and right arrow warning sign
20	Left chevron warning sign
21	Right chevron warning sign
22	<i>Not used</i>
23	Right one-way sign
24	<i>Not used</i>
25	School bus only sign
26 - 38	<i>Not used</i>
39	Crosswalk sign
40	<i>Not used</i>
41	Traffic signal
42	Curve right warning sign
43	Curve left warning sign
44	Up right arrow warning sign
45 - 47	<i>Not used</i>
48	Railroad crossing sign
49	Street sign
50	Roundabout warning sign
51	Fire hydrant
52	Exit sign
53	Bike lane sign
54 - 56	<i>Not used</i>
57	Sky

ID	Type
58	Curb
59	Flyover ramp
60	Road guard rail
61-66	<i>Not used</i>
67	Deer
68-70	<i>Not used</i>
71	Barricade
72	Motorcycle
73-255	<i>Not used</i>

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

Unique sensor identifier, specified as a positive integer. In a multisensor system, the sensor identifier distinguishes between sensors. When you add a new sensor block to your model, the **Sensor**

identifier of that block is $N + 1$. N is the highest **Sensor identifier** value among existing sensor blocks in the model.

Example: 2

Parent name — Name of parent to which sensor is mounted

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.

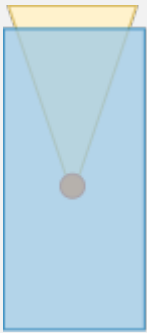
Example: `SimulinkVehicle1`

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 facing forward .
- 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 facing forward .

- 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

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 “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 $[f_x, f_y]$. Units are in pixels.

$$f_x = F \times s_x$$

$$f_y = F \times s_y$$

where:

- F is the focal length in world units, typically millimeters.
- $[s_x, s_y]$ 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 1-by-2 nonnegative vector | real-valued 1-by-3 nonnegative vector

Radial distortion coefficients, specified as a real-valued 1-by-2 or 1-by-3 nonnegative vector. Radial distortion occurs when light rays bend more than the edges of a lens than they do at its optical center. The distortion is greater when the lens is smaller. The block calculates the radial-distorted location of a point. Units are dimensionless.

This parameter is equivalent to the `RadialDistortion` property of a `cameraIntrinsics` object.

Tangential distortion coefficients — Tangential distortion coefficients

$[0, 0]$ (default) | real-valued 1-by-2 nonnegative vector

Tangential distortion coefficients, specified as a real-valued 1-by-2 nonnegative vector. Tangential distortion occurs when the lens and the image plane are not parallel. The coordinates are expressed in world units. Units are dimensionless.

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 “Configure a Signal for Logging” (Simulink).

Algorithms

The block uses the camera model proposed by Jean-Yves Bouquet [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)

References

- [1] Bouquet, J. Y. *Camera Calibration Toolbox for Matlab*. http://www.vision.caltech.edu/bouquetj/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.

Extended Capabilities

C/C++ Code Generation

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

See Also

Blocks

Simulation 3D Fisheye Camera | Simulation 3D UAV Vehicle | Simulation 3D Scene Configuration | Simulation 3D Lidar

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)

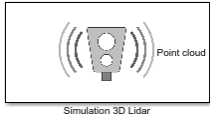
“Semantic Segmentation Using Deep Learning” (Computer Vision Toolbox)

Introduced in R2020b

Simulation 3D Lidar

Lidar sensor model in 3D simulation environment

Library: UAV Toolbox / Simulation 3D



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. 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 output**.

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

Unique sensor identifier, specified as a positive integer. In a multisensor system, the sensor identifier distinguishes between sensors. When you add a new sensor block to your model, the **Sensor identifier** of that block is $N + 1$. N is the highest **Sensor identifier** value among existing sensor blocks in the model.

Example: 2

Parent name — Name of parent to which sensor is mounted

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	<p>Forward-facing sensor mounted to the vehicle origin, which is on the ground, at the geometric center of the vehicle</p> 	[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 facing forward .
- 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 facing forward .
- 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 output – Output distance to measured object points

off (default) | on

Select this parameter to output the distance to measured object points at the **Distance** 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 “Configure a Signal for Logging” (Simulink).

Extended Capabilities**C/C++ Code Generation**

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

See Also**Blocks**

Simulation 3D Camera | Simulation 3D Fisheye Camera | Simulation 3D Scene Configuration | Simulation 3D UAV Vehicle

Objects

`pcplayer` | `pointCloud`

Topics

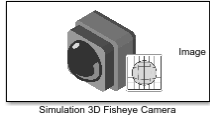
“Coordinate Systems for Unreal Engine Simulation in UAV Toolbox”
“Choose a Sensor for Unreal Engine Simulation”

Introduced in R2020b

Simulation 3D Fisheye Camera

Fisheye camera sensor model in 3D simulation environment

Library: UAV Toolbox / Simulation 3D



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 4-61. 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”.

Ports

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

Unique sensor identifier, specified as a positive integer. In a multisensor system, the sensor identifier distinguishes between sensors. When you add a new sensor block to your model, the **Sensor identifier** of that block is $N + 1$. N is the highest **Sensor identifier** value among existing sensor blocks in the model.

Example: 2

Parent name — Name of parent to which sensor is mounted

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.

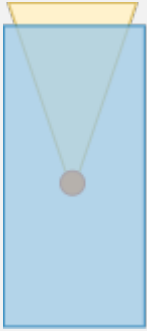
Example: `SimulinkVehicle1`

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 facing forward .
- 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 facing forward.
- 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

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 “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 $[a_0 \ a_2 \ a_3 \ a_4]$.

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 “Configure a Signal for Logging” (Simulink).

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.

Extended Capabilities

C/C++ Code Generation

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

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"

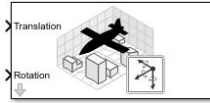
"Fisheye Calibration Basics" (Computer Vision Toolbox)

Introduced in R2019b

Simulation 3D UAV Vehicle

Place UAV vehicle in 3D visualization

Library: 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 — Translation of vehicle relative to scene

vector

Translated position of the vehicle relative to the Unreal Engine scene origin. Translation vector defines the X, Y , and Z positions, in meters, of the vehicle using 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 scene

vector

Rotation of the vehicle relative to the Unreal Engine inertial reference frame. The rotation vector defines the $Yaw, Pitch$, and $Roll$ values, in degrees, of the vehicle rotation 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

Parameters

Vehicle Parameters

Type — Type of vehicle

Quadcopter (default) | Fixed wing

Select the type of vehicle. To obtain the dimensions of each vehicle type, see these reference pages:

- Quadcopter — **Quadcopter**
- Fixed wing — **Fixed wing**

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 `SimulinkVehicleX`. 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.

Initial Values

Initial Translation (m) — Initial vehicle position

[0, 0, 0] (default) | real-valued 1-by-3 vector

Initial vehicle position along the X-axis, Y-axis, and Z-axis in the inertial Z-down coordinate system, in m.

Initial Rotation (rad) — Initial angle of vehicle rotation

[0, 0, 0] (default) | real-valued 1-by-3 vector

Initial angle of vehicle rotation, in rad. The angle of rotation is defined by the roll, pitch, and yaw of the vehicle.

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.

Extended Capabilities

C/C++ Code Generation

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

See Also

Blocks

Simulation 3D Fisheye Camera | Simulation 3D Lidar | Simulation 3D Scene Configuration | Simulation 3D UAV Vehicle

Tools

Fixed Wing Aircraft | **Quadrotor**

Topics

“Coordinate Systems for Unreal Engine Simulation in UAV Toolbox”

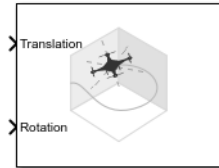
“Choose a Sensor for Unreal Engine Simulation”

Introduced in R2020b

UAV Animation

Animate UAV flight path using translations and rotations

Library: 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).

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”

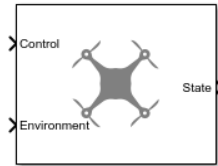
“Tuning Waypoint Follower for Fixed-Wing UAV”

Introduced in R2018b

Guidance Model

Reduced-order model for UAV

Library: 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

Data Type — 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 multicopter).

For multicopter 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:

$$\begin{aligned}\dot{x}_e &= V_g \cos \chi \cos \gamma \\ \dot{y}_e &= V_g \sin \chi \cos \gamma \\ \dot{h} &= V_g \sin \gamma \\ \dot{\chi} &= \frac{g \cos(\chi - \psi)}{V_g} \tan \phi \\ V_g \sin(\gamma^c) &= \min(\max(k_h(h^c - h), -V_g), V_g) \\ \dot{\gamma} &= k_\gamma(\gamma^c - \gamma) \\ \dot{V}_a &= k_{V_a}(V_a^c - V_a) \\ \frac{g \cos(\chi - \psi)}{V_g} \tan(\phi^c) &= k_\chi(\chi^c - \chi) \\ \ddot{\phi} &= k_{P\phi}(\phi^c - \phi) + k_{D\phi}(-\dot{\phi})\end{aligned}$$

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:

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

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 from world to body 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:

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

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

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

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

This model assumes the autopilot takes in commanded roll, pitch, yaw rate, $[\psi^c, \theta^c, \phi^c]$ and a commanded total thrust force, F_{thrust}^c . 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:

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

These variables match the output of the state function.

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

control | derivative | environment | ode45 | plotTransforms | state

Objects

fixedwing | multirotor | uavWaypointFollower

Blocks

Waypoint Follower

Topics

“Approximate High-Fidelity UAV model with UAV Guidance Model block”

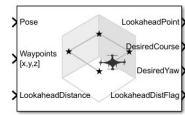
“Tuning Waypoint Follower for Fixed-Wing UAV”

Introduced in R2018b

Waypoint Follower

Follow waypoints for UAV

Library: 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

$[x \ y \ z \ \text{chi}]$ vector

Current UAV pose, specified as a $[x \ y \ z \ \text{chi}]$ 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.

Example: $[0.5; 1.75; -2.5; \text{pi}]$

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.

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**.

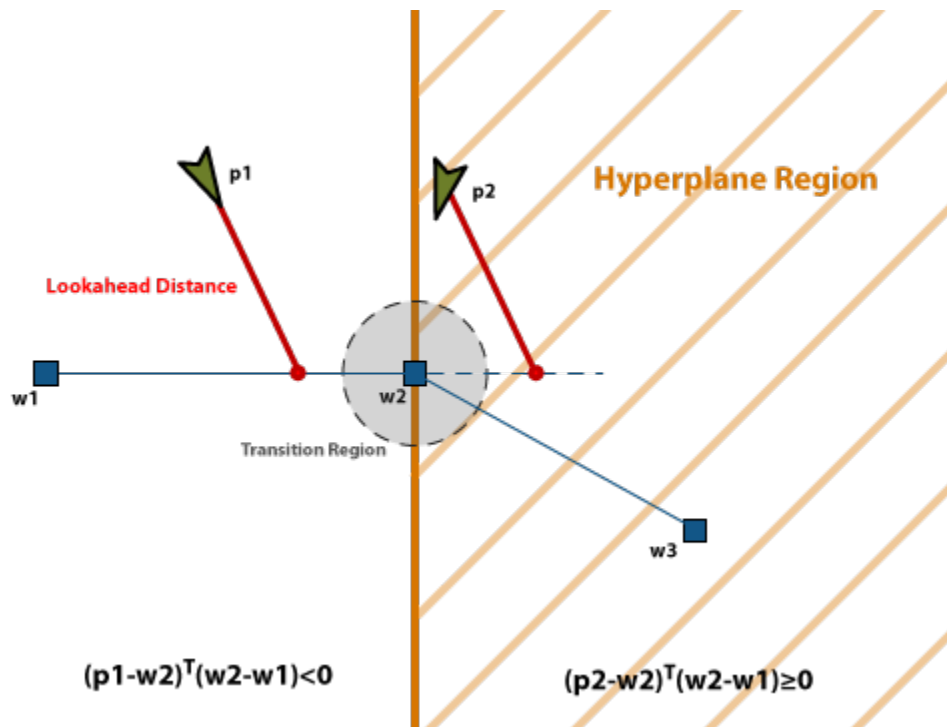
This parameter is non-tunable.

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:

$$(p-w1)^T (w2-w1) \geq 0$$

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.

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

control | derivative | environment | ode45 | plotTransforms | state

Objects

fixedwing | multicopter | uavWaypointFollower

Topics

“Approximate High-Fidelity UAV model with UAV Guidance Model block”

“Tuning Waypoint Follower for Fixed-Wing UAV”

Introduced in R2018b

Apps

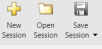

Flight Log Analyzer

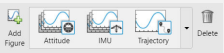


Analyze UAV autopilot flight logs

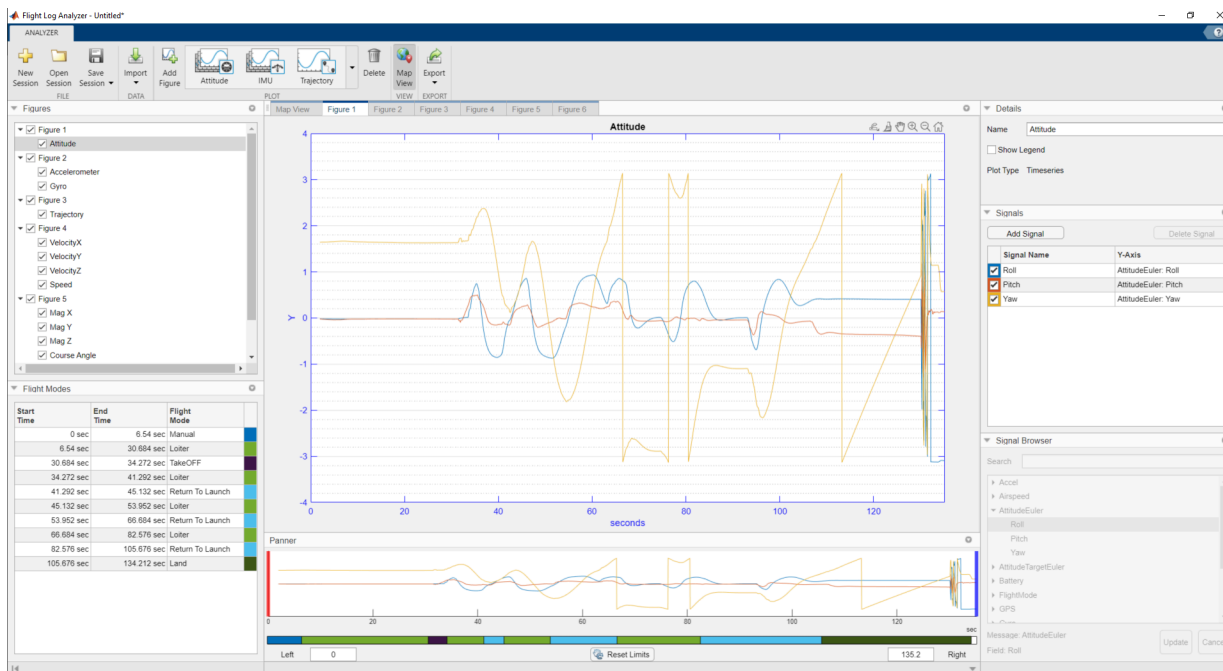
Description

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:

	<p>Click New Session to create a new session.</p> <p>You can open saved app sessions by clicking Open Session.</p> <p>You can save your progress to a MAT-file (.mat) by clicking Save Session.</p>
	<p>To load a ULOG file (.ulg) or MAT-file containing a ulogreader object, select Import > From ULOG.</p> <p>To load a TLOG file (.tlog) or MAT-file containing a mavlinktlog object, select Import > From TLOG.</p> <p>Select Import > From Workspace to load a ulogreader object, mavlinktlog object, or custom log data and a flightLogSignalMapping object from the workspace.</p>

	<p>Click Add Figure to add a new figure for plotting.</p> <p>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.</p> <p>Predefined Plots</p> <ul style="list-style-type: none"> • 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 <p>Custom Plots</p> <ul style="list-style-type: none"> • 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 <p>You can delete the selected figure or plot by clicking Delete.</p>
	<p>Click Map View to view or hide the satellite image map with logged GPS data.</p> <hr/> <p>Note The app requires internet access to retrieve satellite imagery.</p>
	<p>Select Export > Export Figure to export the currently selected figure as a .fig file.</p> <p>Select Export > Export Signal to export the signals as timetable to the MATLAB workspace or a MAT-file (.mat).</p>



Open the Flight Log Analyzer App

- MATLAB Toolstrip: On the **Apps** tab, under **Control System Design and Analysis**, 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 the Flight Log Analyzer App

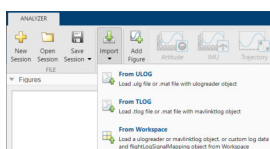
In the **Apps** tab, under **Control System Design and Analysis**, click **Flight Log Analyzer**.

Alternatively, you can use the `flightLogAnalyzer` function from the MATLAB command prompt:

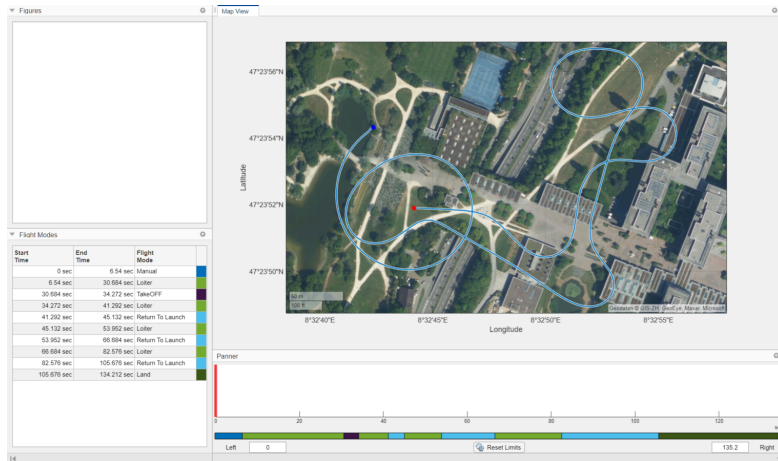
```
flightLogAnalyzer
```

Import a ULOG File

Select **Import** > **From ULOG** to load the UAV flight log data from a ULOG (.ulg) file.

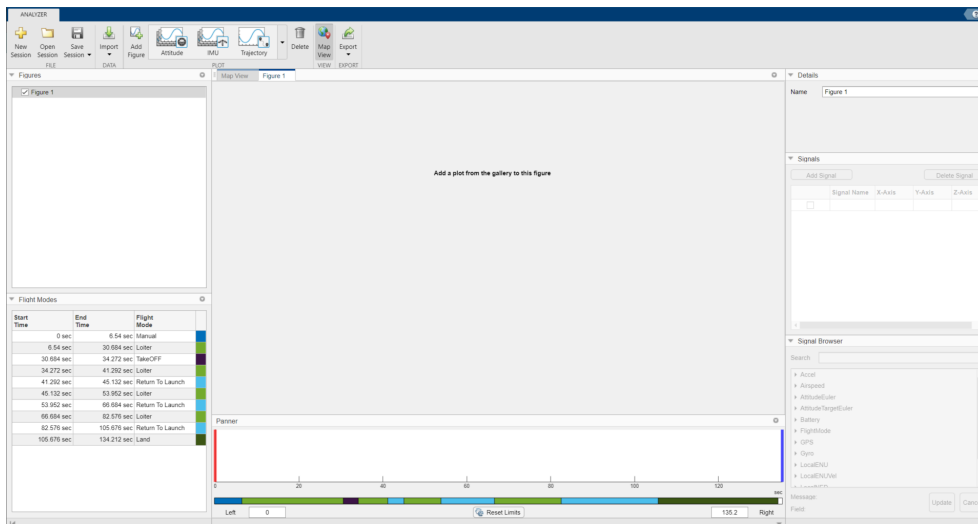


By default, the app displays a satellite map with logged GPS data and the flight modes as a table. The flight modes, along with their corresponding start and end times, are tabulated in the **Flight Modes** pane.



Create Figures and Plots

- 1 To create a new figure for plotting, click **Add Figure**. The app adds an empty figure to the plotting pane.



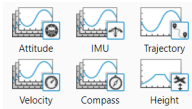
You can continue adding additional figures using this process.

- 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 **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 toolbar. Deleting a figure deletes all plots in the figure.

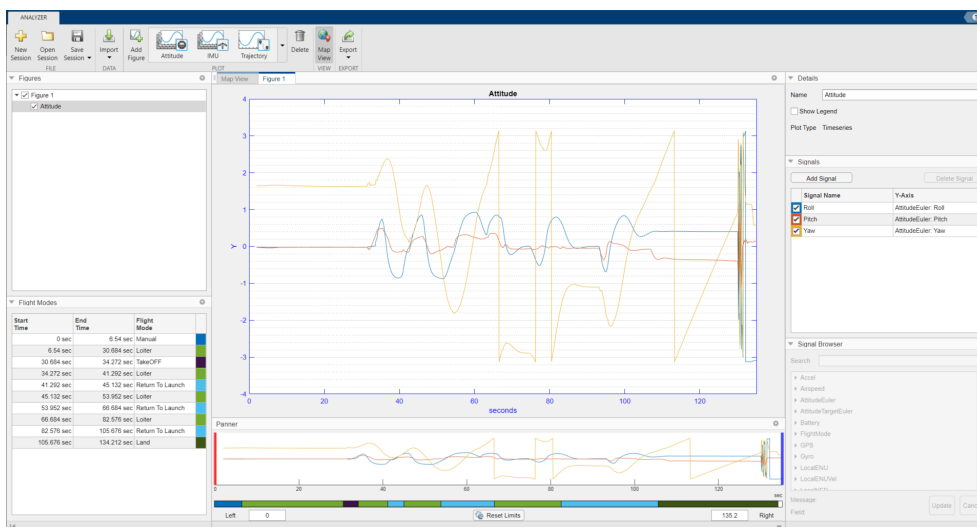
Creating a figure enables the plot gallery. You can add one or more predefined plots or custom plots to a figure from the plot gallery.

Add a Predefined Plot

- To add a predefined plot to a figure, select one of the six predefined plots from the plot gallery.



- For example, click **Attitude** to add plots for rotation angles and rotation rates to the figure.

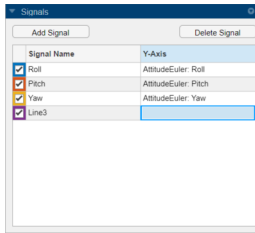


You can continue adding additional plots to a figure using this process.

- 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.
- To rename a plot, select the associated plot item in the **Figures** pane, click the **Name** box in the **Details** pane, and type a new name.
- 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.
- To delete a plot, select the plot item in the **Figures** pane and click **Delete** on the app toolstrip.

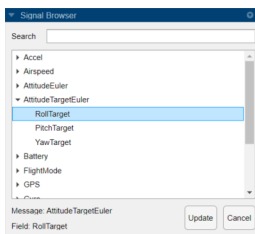
Edit Plot Signals

- 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.
- 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.
- To add a new signal to the selected plot, click **Add Signal**.



To rename the signal, double-click signal in the **Signal Name** column and type a new name.

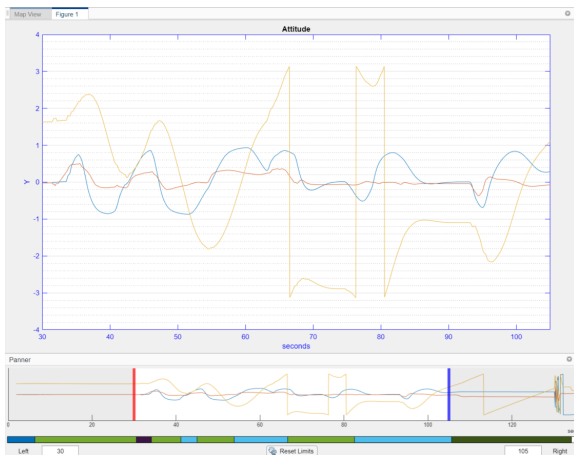
- To add or update the signal data, double-click the data field for the desired signal in the corresponding column to enable the **Signal Browser** pane. Choose from available signals.
- Select one of the signals from the **Signal Browser** pane and click **Update**.



- To delete a signal, select a signal from the **Signals** pane and click **Delete Signal**.

Change the Plot Focus Using the Panner

- 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.



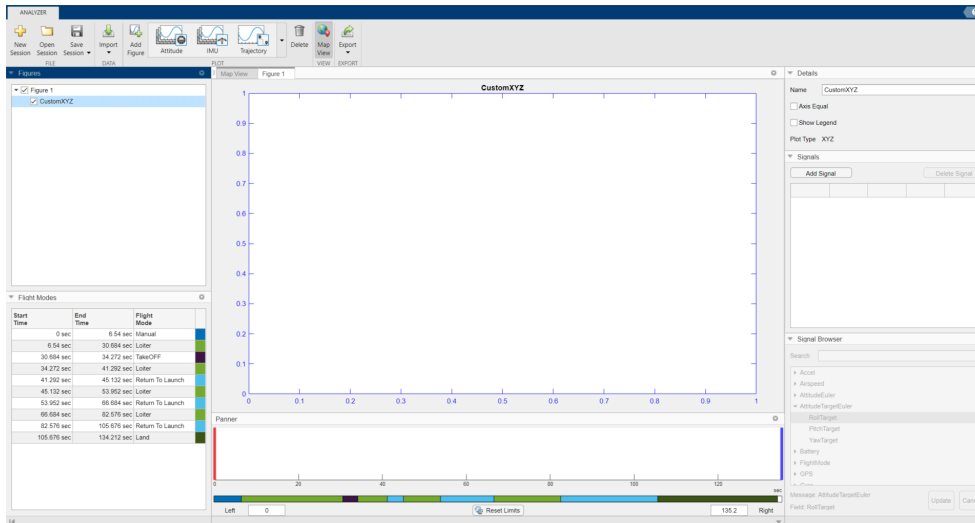
- You can also move the handles by typing new values in the **Left** and **Right** boxes, beneath the strip plot. To reset the handles to their default values, click **Reset Limits**.
- The color next to each flight mode in the **Flight Modes** pane represents that flight mode in the color bar under the strip plot in the **Panner** pane.

Add a Custom Plot

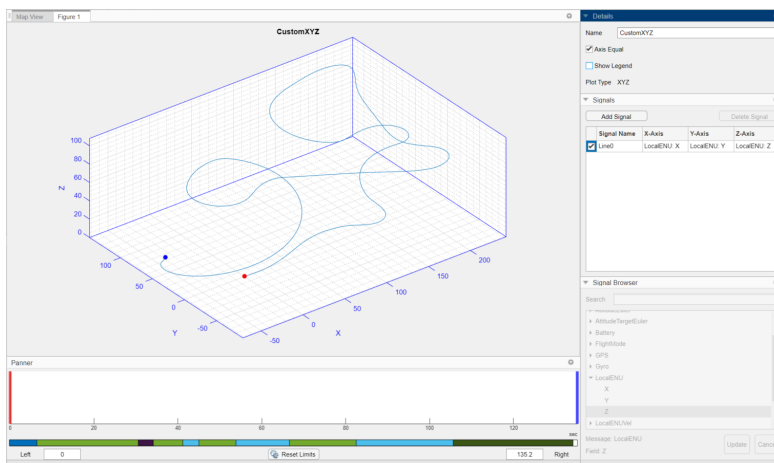
- To add a custom plot to a figure, select one of the three custom plots from the plot gallery. You can add the new plot to the previously created figure or to a new figure.



- 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 for the desired signal in the corresponding column to enable the **Signal Browser** pane. Choose from the available signals.
- 6 For example, to create a trajectory plot in local east-north-up (ENU) Cartesian coordinates:
 - a Double-click the **X-Axis** data field for the desired signal and find the **LocalENU** signal group in the **Signal Browser** pane.
 - b Expand the group and select the signal **X**.
 - c Click **Update** to update the signal with **X-Axis** data.
 - d 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.



Select **Export > Export Signal** to export the signals as timetable to the MATLAB workspace or a MAT-file (.mat).

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**.

- “Analyze UAV Autopilot Flight Log Using Flight Log Analyzer”

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
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 multicopter UAV hovers at the current position
Return To Launch	Return to the launch position
Land	Land at the specified position

See Also

Objects

`flightLogSignalMapping` | `mavlinktlog` | `uLogreader`

Topics

“Analyze UAV Autopilot Flight Log Using Flight Log Analyzer”

Introduced in R2020b

Scenes

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.

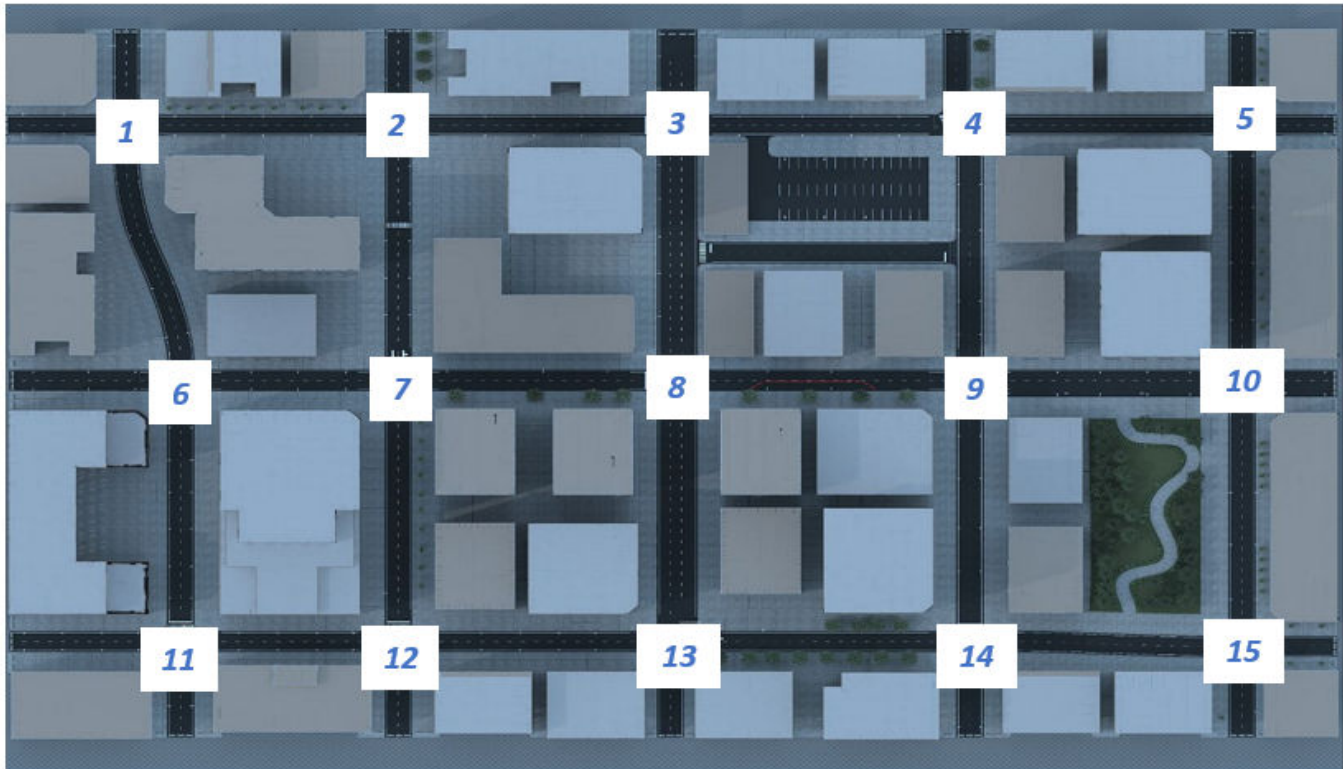


To simulate a UAV flight in this scene:

- 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.

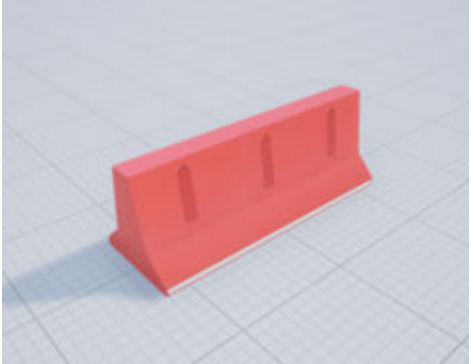


This table provides the intersection locations in the world coordinate system. Dimensions are in m.

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
13	-22.60	110.50	.01
14	76.40	110.50	.01

Intersection	Center Location		
	X (m)	Y (m)	Z (m)
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°
	SM_Barrier 18	197.05	-2.9	0	0	0	-180°

Object	Unreal Engine Editor Name	Location					
		X	Y	Z	Roll	Pitch	Yaw
	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°
	SM_Barrier 55	-22.7	150.15	0	0	0	90°

Object	Unreal Engine Editor Name	Location					
		X	Y	Z	Roll	Pitch	Yaw
	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°
	SM_Barrier 96	-199.9	-147.4	0	0	0	90°

Object	Unreal Engine Editor Name	Location					
		X	Y	Z	Roll	Pitch	Yaw
	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).

Intersect ion	Unreal Engine Editor Name		Location					
	Traffic Light Group	Traffic Light	X	Y	Z	Roll	Pitch	Yaw
1	TrafficLig htGroup	SM_Tr affic Light s1_3	-196.55	-100.65	0	0	0	-90°
		SM_Tr affic Light s1_4	-210.20	-113.40	0	0	0	0
2	TrafficLig htGroup2	SM_Tr affic Light s2_4	-120.40	-113.50	0	0	0	0
		SM_Tr affic Light s2_3	-106.35	98.35	0	0	0	-90°
3	TrafficLig htGroup3	SM_Tr affic Light s3_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_TrafficLight_s3_4	-30.60	-113.80	0	0	0	0
4	TrafficLightGroup4	SM_TrafficLight_s4_4	64.80	-113.0	0	0	0	0
		SM_TrafficLight_s4_3	71.40	-100.30	0	0	0	-100°
5	TrafficLightGroup5	SM_TrafficLight_s5_1	171.50	-115.70	0	0	0	90°
		SM_TrafficLight_s5_4	157.40	-113.50	0	0	0	0
6	TrafficLightGroup6	SM_TrafficLight_s6_3	-189.60	7.40	0	0	0	-90°
		SM_TrafficLight_s6_2	-177.30	5.70	0	0	0	180°
7	TrafficLightGroup7	SM_TrafficLight_s7_3	-117.80	7.70	0.2	0	0	-90°
		SM_TrafficLight_s7_2	-105.20	5.50	0	0	0	180°
8	TrafficLightGroup8	SM_TrafficLight_s8_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_TrafficLight_s13_4	-30.50	105.30	0	0	0	0
14	TrafficLightGroup14	SM_TrafficLight_s14_4	69.30	105.30	0	0	0	0
		SM_TrafficLight_s14_3	70.90	118.70	0	0	0	-90°
15	TrafficLightGroup15	SM_TrafficLight_s15_1	171.40	105.20	0	0	0	90°
		SM_TrafficLight_s15_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

“Coordinate Systems for Unreal Engine Simulation in UAV Toolbox” | Simulation 3D Scene Configuration

Vehicles

Quadrotor

Quadrotor vehicle dimensions

Description



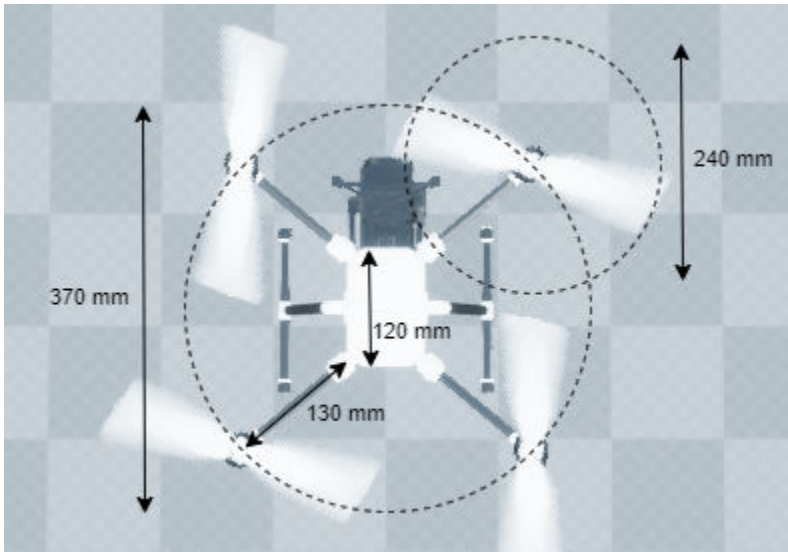
Quadrotot is one of the UAVs that you can use within the Unreal Engine simulation environment. This environment is rendered using the Unreal Engine from Epic Games. The origin is located at the center of the camera gimbal located on the underside of the aircraft. For detailed specifications of the vehicle 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



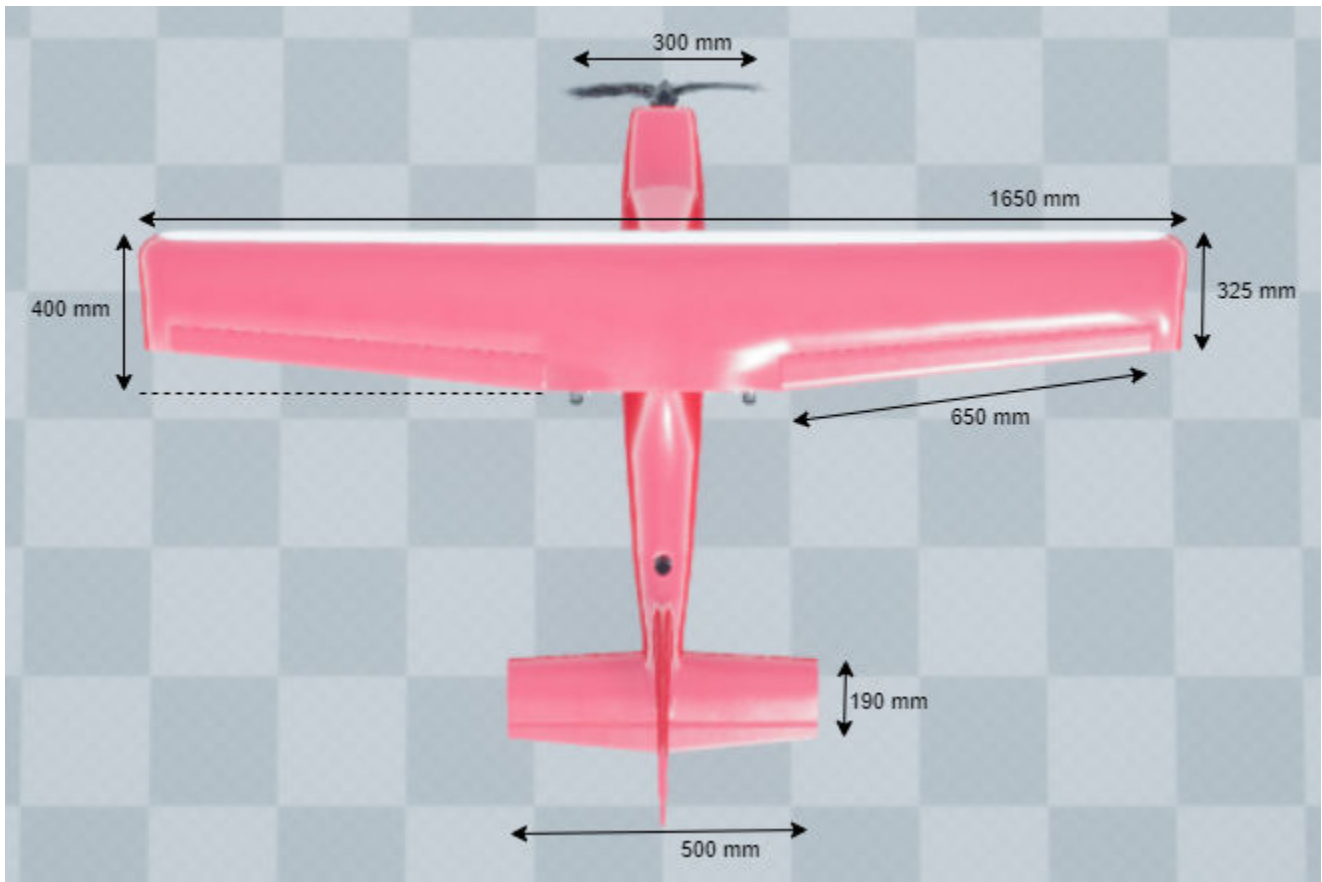
Fixed Wing Aircraft is one of the vehicles that you can use within the Unreal Engine simulation environment. This environment is rendered using the Unreal Engine from Epic Games. The origin is located at the center of the camera gimbal located on the underside of the aircraft. For detailed specifications of the vehicle dimensions, see the **Dimensions** section.

To add this type of vehicle to the 3D simulation environment:

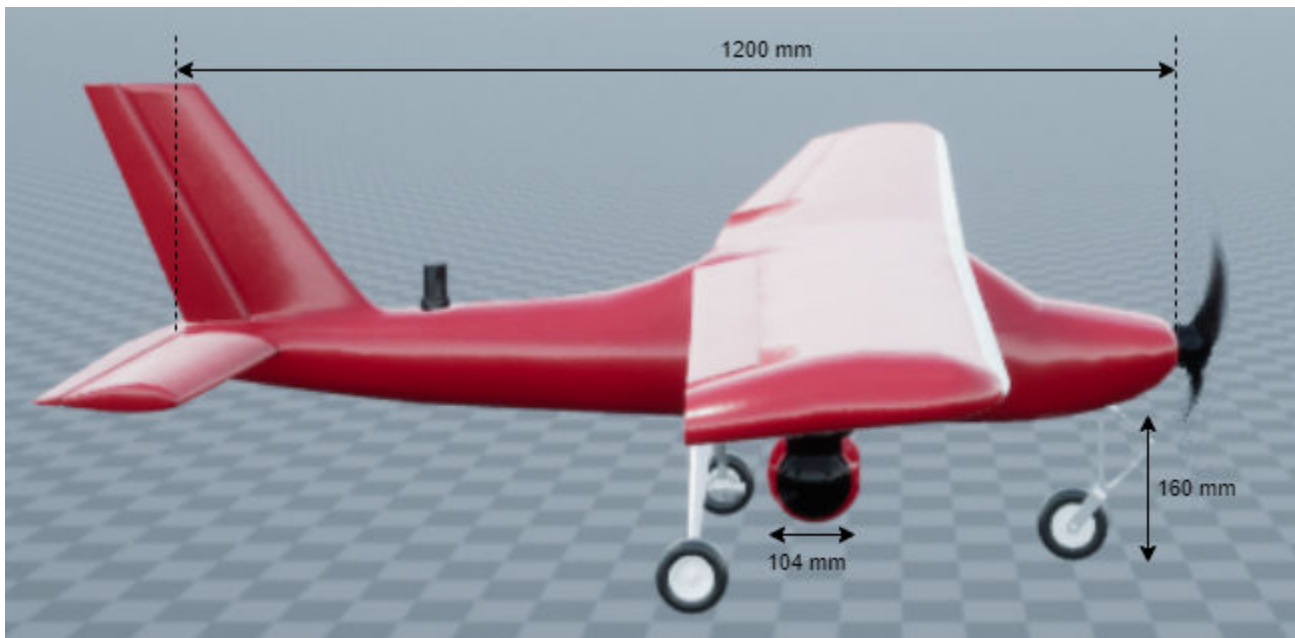
- 1 Add a Simulation 3D UAV Vehicle block to your Simulink model.
- 2 In the block, set the **Type** parameter to Fixed wing.

Dimensions

Top-down view – Vehicle width dimensions
diagram

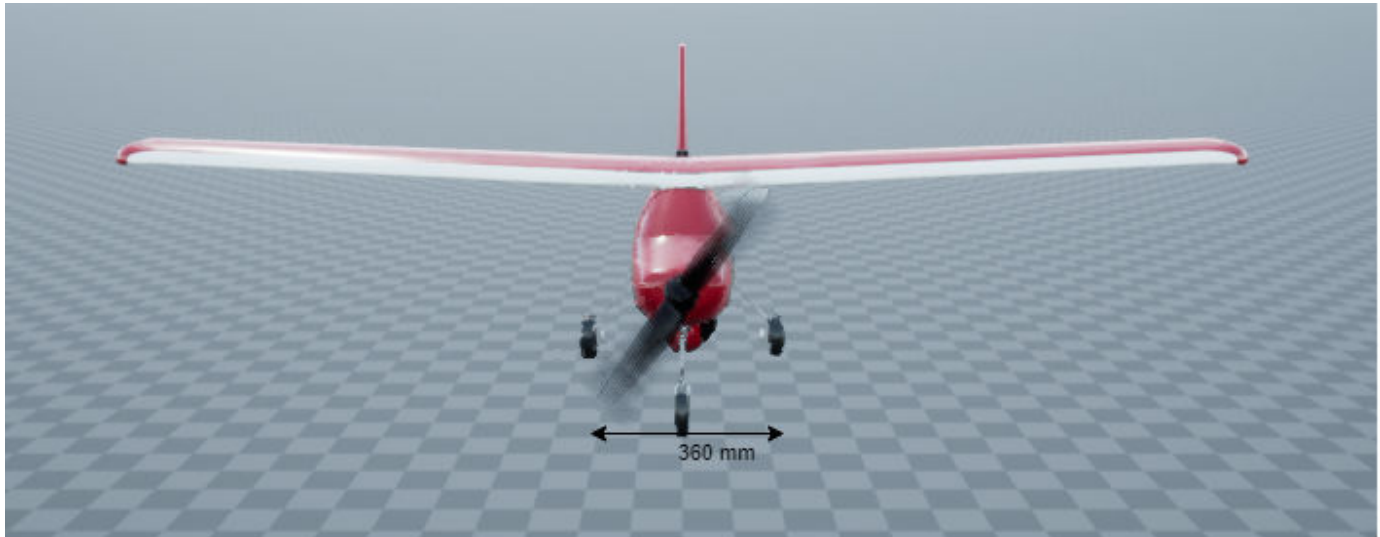


Side view – Vehicle length, landing gear height, and camera dimensions diagram

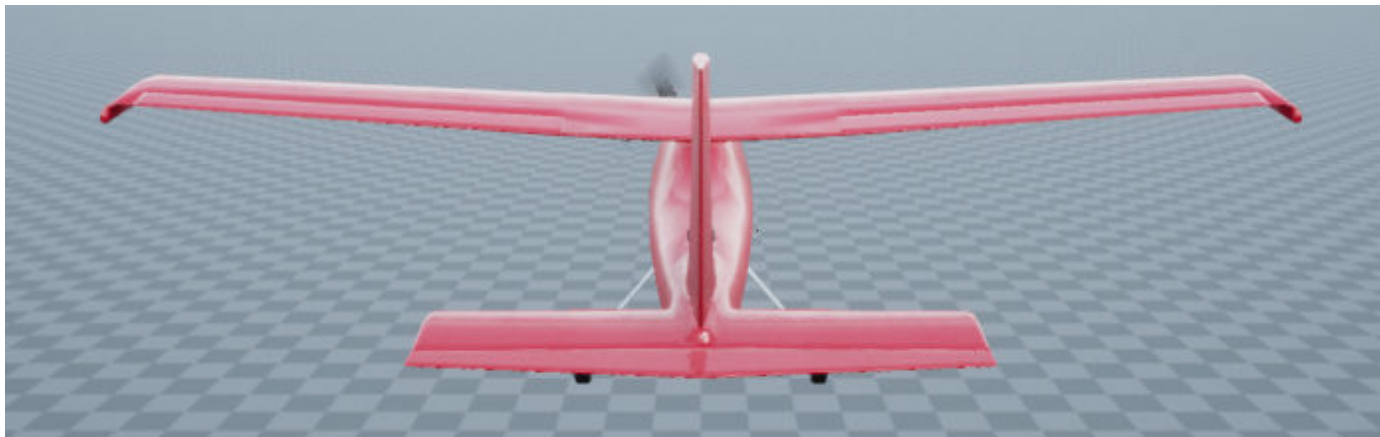


Front view – Tire width dimensions

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”

