

# Robotics System Toolbox™

Reference



MATLAB® & SIMULINK®

R2019a



# How to Contact MathWorks



Latest news: [www.mathworks.com](http://www.mathworks.com)  
Sales and services: [www.mathworks.com/sales\\_and\\_services](http://www.mathworks.com/sales_and_services)  
User community: [www.mathworks.com/matlabcentral](http://www.mathworks.com/matlabcentral)  
Technical support: [www.mathworks.com/support/contact\\_us](http://www.mathworks.com/support/contact_us)



Phone: 508-647-7000



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

## *Robotics System Toolbox™ Reference*

© COPYRIGHT 2015–2019 by The MathWorks, Inc.

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

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

### **Trademarks**

MATLAB and Simulink are registered trademarks of The MathWorks, Inc. See [www.mathworks.com/trademarks](http://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](http://www.mathworks.com/patents) for more information.

**Revision History**

March 2015	Online only	New for Version 1.0 (R2015a)
September 2015	Online only	Revised for Version 1.1 (R2015b)
October 2015	Online only	Rereleased for Version 1.0.1 (Release 2015aSP1)
March 2016	Online only	Revised for Version 1.2 (R2016a)
September 2016	Online only	Revised for Version 1.3 (R2016b)
March 2017	Online only	Revised for Version 1.4 (R2017a)
September 2017	Online only	Revised for Version 1.5 (R2017b)
March 2018	Online only	Revised for Version 2.0 (R2018a)
September 2018	Online only	Revised for Version 2.1 (R2018b)
March 2019	Online only	Revised for Version 2.2 (R2019a)



<b>1</b>	<b>Classes – Alphabetical List</b>
<b>2</b>	<b>Functions – Alphabetical List</b>
<b>3</b>	<b>Methods – Alphabetical List</b>
<b>4</b>	<b>Blocks – Alphabetical List</b>
<b>5</b>	<b>Apps in Robotics System Toolbox</b>



# Classes — Alphabetical List

---

## BagSelection

Object for storing rosbag selection

### Description

The `BagSelection` object is an index of the messages within a rosbag. You can use it to extract message data from a rosbag, select messages based on specific criteria, or create a `timeseries` of the message properties.

Use `rosbag` to load a rosbag and create the `BagSelection` object.

Use `select` to filter the rosbag by criteria such as time and topic.

### Creation

### Syntax

```
bag = rosbag(filename)
```

```
bagsel = select(bag)
```

### Description

`bag = rosbag(filename)` creates an indexable `BagSelection` object, `bag`, that contains all the message indexes from the rosbag at the input path, `filename`. To access the data, you can call `readMessages` or `timeseries` to extract relevant data.

See `rosbag` for other syntaxes.

`bagsel = select(bag)` returns an object, `bagsel`, that contains all the messages in the `BagSelection` object, `bag`.

This function does not change the contents of the original `BagSelection` object. The return object, `bagsel`, a new object that contains the specified message selection.



See `select` for other syntaxes and to filter by criteria such as time and topic.

## Properties

### **FilePath** — Absolute path to rosbag file

character vector

This property is read-only.

Absolute path to the rosbag file, specified as a character vector.

Data Types: `char`

### **StartTime** — Timestamp of first message in selection

scalar

This property is read-only.

Timestamp of the first message in the selection, specified as a scalar in seconds.

Data Types: `double`

### **EndTime** — Timestamp of last message in selection

scalar

This property is read-only.

Timestamp of the last message in the selection, specified as a scalar in seconds.

Data Types: `double`

### **NumMessages** — Number of messages in selection

scalar

This property is read-only.

Number of messages in the selection, specified as a scalar. When you first load a rosbag, this property contains the number of messages in the rosbag. Once you select a subset of messages with `select`, the property shows the number of messages in this subset.

Data Types: `double`

## AvailableTopics — Table of topics in selection

table

This property is read-only.

Table of topics in the selection, specified as a table. Each row in the table lists one topic, the number of messages for this topic, the message type, and the definition of the type. For example:

	NumMessages	MessageType	MessageDefinition
/odom	99	nav_msgs/Odometry	'# This represents an estimate of a p

Data Types: table

## AvailableFrames — List of available coordinate frames

cell array of character vectors

This property is read-only.

List of available coordinate frames, returned as a cell array of character vectors. Use `canTransform` to check whether specific transformations between frames are available, or `getTransform` to query a transformation.

Data Types: cell array

## MessageList — List of messages in selection

table

This property is read-only.

List of messages in the selection, specified as a table. Each row in the table lists one message.

Data Types: table

## Object Functions

<code>canTransform</code>	Verify if transformation is available
<code>getTransform</code>	Retrieve transformation between two coordinate frames
<code>readMessages</code>	Read messages from rosbag

---

select	Select subset of messages in rosbag
timeseries	Creates a time series object for selected message properties

## Examples

### Create rosbag Selection Using BagSelection Object

Load a rosbag log file and parse out specific messages based on the selected criteria.

Create a BagSelection object of all the messages in the rosbag log file.

```
bagMsgs = rosbag('ex_multiple_topics.bag');
```

Select a subset of the messages based on their timestamp and topic.

```
bagMsgs2 = select(bagMsgs, 'Time', ...  
                [bagMsgs.StartTime bagMsgs.StartTime + 1], 'Topic', '/odom');
```

Retrieve the messages in the selection as a cell array.

```
msgs = readMessages(bagMsgs2);
```

Return certain message properties as a time series.

```
ts = timeseries(bagMsgs2, 'Pose.Pose.Position.X', ...  
               'Twist.Twist.Angular.Y');
```

### Retrieve Information from rosbag

Retrieve information from the rosbag. Specify the full path to the rosbag if it is not already available on the MATLAB® path.

```
bagselect = rosbag('ex_multiple_topics.bag');
```

Select a subset of the messages, filtered by time and topic.

```
bagselect2 = select(bagselect, 'Time', ...  
                  [bagselect.StartTime bagselect.StartTime + 1], 'Topic', '/odom');
```

## Display rosbag Information from File

To view information about a rosbag log file, use `rosbag info filename`, where *filename* is a rosbag (.bag) file.

```
rosbag info 'ex_multiple_topics.bag'
```

```
Path:      C:\TEMP\Bdoc19a_1067994_6688\ib99EA80\22\tpbda3f8b1\robotics-ex61825935\ex_mu
Version:   2.0
Duration:  2:00s (120s)
Start:     Dec 31 1969 19:03:21.34 (201.34)
End:       Dec 31 1969 19:05:21.34 (321.34)
Size:      23.6 MB
Messages:  36963
Types:     gazebo_msgs/LinkStates [48c080191eb15c41858319b4d8a609c2]
           nav_msgs/Odometry      [cd5e73d190d741a2f92e81eda573aca7]
           rosgraph_msgs/Clock    [a9c97c1d230cfc112e270351a944ee47]
           sensor_msgs/LaserScan  [90c7ef2dc6895d81024acba2ac42f369]
Topics:    /clock                  12001 msgs : rosgraph_msgs/Clock
           /gazebo/link_states    11999 msgs : gazebo_msgs/LinkStates
           /odom                  11998 msgs : nav_msgs/Odometry
           /scan                   965 msgs  : sensor_msgs/LaserScan
```

## Get Transformations from rosbag File

Get transformations from rosbag (.bag) files by loading the rosbag and checking the available frames. From these frames, use `getTransform` to query the transformation between two coordinate frames.

Load the rosbag.

```
bag = rosbag('ros_turtlesim.bag');
```

Get a list of available frames.

```
frames = bag.AvailableFrames;
```

Get the latest transformation between two coordinate frames.

```
tf = getTransform(bag, 'world', frames{1});
```

Check for a transformation available at a specific time and retrieve the transformation. Use `canTransform` to check if the transformation is available. Specify the time using `rostime`.

```
tfTime = rostime(bag.StartTime + 1);  
if (canTransform(bag, 'world', frames{1}, tfTime))  
    tf2 = getTransform(bag, 'world', frames{1}, tfTime);  
end
```

## Read Messages from a rosbag as a Structure

Load the rosbag.

```
bag = rosbag('ros_turtlesim.bag');
```

Select a specific topic.

```
bSel = select(bag, 'Topic', '/turtle1/pose');
```

Read messages as a structure. Specify the `DataFormat` name-value pair when reading the messages. Inspect the first structure in the returned cell array of structures.

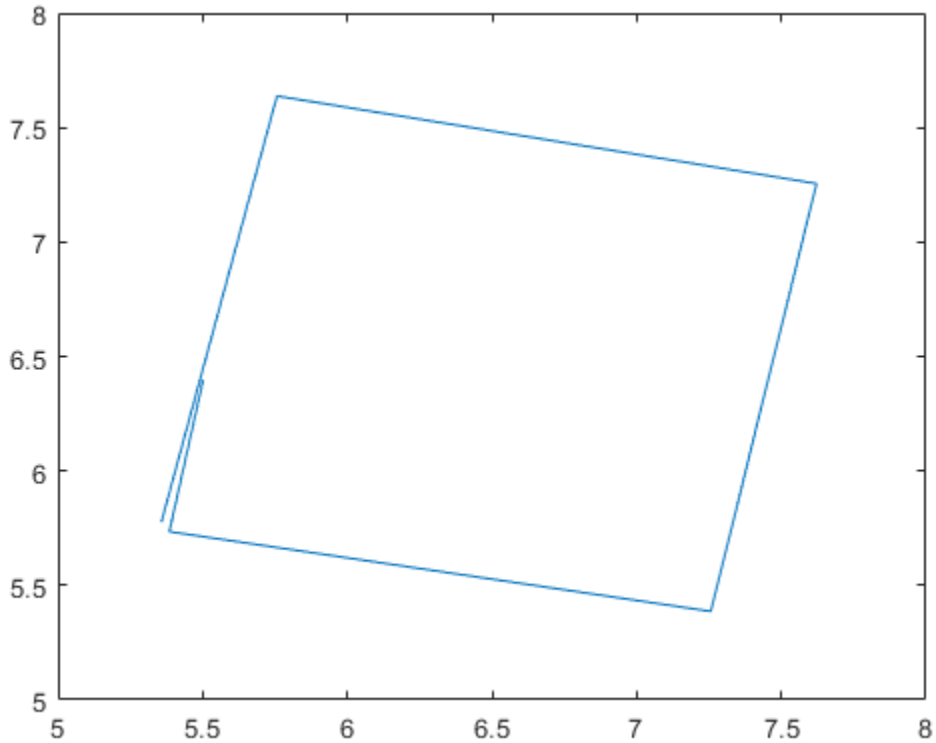
```
msgStructs = readMessages(bSel, 'DataFormat', 'struct');  
msgStructs{1}
```

```
ans = struct with fields:  
    MessageType: 'turtlesim/Pose'  
        X: 5.5016  
        Y: 6.3965  
        Theta: 4.5377  
    LinearVelocity: 1  
    AngularVelocity: 0
```

Extract the xy points from the messages and plot the robot trajectory.

Use `cellfun` to extract all the X and Y fields from the structure. These fields represent the xy positions of the robot during the rosbag recording.

```
xPoints = cellfun(@(m) double(m.X), msgStructs);  
yPoints = cellfun(@(m) double(m.Y), msgStructs);  
plot(xPoints, yPoints)
```



## See Also

`canTransform` | `getTransform` | `readMessages` | `rosbag` | `select` | `timeseries`

## Topics

“Work with rosbag Logfiles”

“ROS Log Files (rosbags)”

**Introduced in R2015a**

## Core

Create ROS Core

### Description

The ROS Core encompasses many key components and nodes that are essential for the ROS network. You must have exactly one ROS core running in the ROS network for nodes to communicate. Using this class allows the creation of a ROS core in MATLAB®. Once the core is created, you can connect to it by calling `rosinit` or `robotics.ros.Node`.

### Creation

### Syntax

```
core = robotics.ros.Core  
core = robotics.ros.Core(port)
```

### Description

`core = robotics.ros.Core` returns a Core object and starts a ROS core in MATLAB. This ROS core has a default port of 11311. MATLAB only allows the creation of one core on any given port and displays an error if another core is detected on the same port.

`core = robotics.ros.Core(port)` starts a ROS core at the specified port, `port`.

### Properties

#### **Port — Network port at which the ROS master is listening**

11311 (default) | scalar

This property is read-only.

Network port at which the ROS master is listening, returned as a scalar.

## **MasterURI — The URI on which the ROS master can be reached**

'http://<HOSTNAME>:11311' (default) | character vector

This property is read-only.

The URI on which the ROS master can be reached, returned as a character vector. The MasterURI is constructed based on the host name of your computer. If your host name is not valid, the IP address of your first network interface is used.

## **Examples**

### **Create ROS Core**

Create ROS core on localhost and default port 11311.

```
core = robotics.ros.Core;
```

Clear the ROS core to shut down the ROS network.

```
clear('core')
```

### **Create ROS Core On Specific Port**

Create ROS core on localhost and port 12000.

```
core = robotics.ros.Core(12000);
```

Clear the ROS core to shut down the ROS network.

```
clear('core')
```

## **See Also**

Node | rosinit

## **Topics**

“Connect to a ROS Network”



“ROS Network Setup”

## **External Websites**

ROS Core

**Introduced in R2015a**

## CompressedImage

Create compressed image message

### Description

The `CompressedImage` object is an implementation of the `sensor_msgs/CompressedImage` message type in ROS. The object contains the compressed image and meta-information about the message. You can create blank `CompressedImage` messages and populate them with data, or subscribe to image messages over the ROS network. To convert the image to a MATLAB image, use the `readImage` function.

Only images that are sent through the ROS Image Transport package are supported for conversion to MATLAB images.

### Creation

### Syntax

```
msg = rosmessage('sensor_msgs/CompressedImage')
```

### Description

`msg = rosmessage('sensor_msgs/CompressedImage')` creates an empty `CompressedImage` object. To specify image data, use the `msg.Data` property. You can also get these image messages off the ROS network using `rossubscriber`.

### Properties

#### MessageType — Message type of ROS message

character vector

This property is read-only.

Message type of ROS message, returned as a character vector.

Data Types: char

### **Header — ROS Header message**

Header object

This property is read-only.

ROS Header message, returned as a Header object. This header message contains the MessageType, sequence (Seq), timestamp (Stamp), and FrameId.

### **Format — Image format**

character vector

Image format, specified as a character vector.

Example: 'bgr8; jpeg compressed bgr8'

### **Data — Image data**

uint8 array

Image data, specified as a uint8 array.

## **Object Functions**

`readImage` Convert ROS image data into MATLAB image

## **Examples**

### **Read and Write CompressedImage Messages**

Read and write a sample ROS CompressedImage message by converting it

Load sample ROS messages and inspect the image message. `imgcomp` is a sample ROS CompressedImage message object.

```
exampleHelperROSLoadMessages
imgcomp
```

```
imgcomp =
  ROS CompressedImage message with properties:
```

```
MessageType: 'sensor_msgs/CompressedImage'  
Header: [1x1 Header]  
Format: 'bgr8; jpeg compressed bgr8'  
Data: [30376x1 uint8]
```

Use `showdetails` to show the contents of the message

Create a MATLAB image from the `CompressedImage` message using `readImage` and display it.

```
I = readImage(imgcomp);  
imshow(I)
```



### Create Blank Compressed Image Message

```
compImg = rosmesssage('sensor_msgs/CompressedImage')
```

```
compImg =  
  ROS CompressedImage message with properties:  
  
  MessageType: 'sensor_msgs/CompressedImage'  
  Header: [1x1 Header]  
  Format: ''  
  Data: [0x1 uint8]
```

Use `showdetails` to show the contents of the message

## **See Also**

`readImage` | `rosmmessage` | `rossubscriber`

## **Topics**

“Work with Specialized ROS Messages”

**Introduced in R2015a**

# Image

Create image message

## Description

The Image object is an implementation of the `sensor_msgs/Image` message type in ROS. The object contains the image and meta-information about the message. You can create blank Image messages and populate them with data, or subscribe to image messages over the ROS network. To convert the image to a MATLAB image, use the `readImage` function.

## Creation

## Syntax

```
msg = rosmessage('sensor_msgs/Image')
```

## Description

`msg = rosmessage('sensor_msgs/Image')` creates an empty Image object. To specify image data, use the `msg.Data` property. You can also get these image messages off the ROS network using `rossubscriber`.

## Properties

### MessageType — Message type of ROS message

character vector

This property is read-only.

Message type of ROS message, returned as a character vector.

Data Types: char

## **Header — ROS Header message**

Header object

This property is read-only.

ROS Header message, returned as a Header object. This header message contains the MessageType, sequence (Seq), timestamp (Stamp), and FrameId.

## **Height — Image height in pixels**

scalar

Image height in pixels, specified as a scalar.

## **Width — Image width in pixels**

scalar

Image width in pixels, specified as a scalar.

## **Encoding — Image encoding**

character vector

Image encoding, specified as a character vector.

Example: 'rgb8'

## **IsBigendian — Image byte sequence**

true | false

Image byte sequence, specified as a true or false.

- `true` —Big endian sequence. Stores the most significant byte in the smallest address.
- `false` —Little endian sequence. Stores the least significant byte in the smallest address.

The Big endian sequence stores the most significant byte in the smallest address. The Little endian sequence stores the least significant byte in the smallest address.

## **Step — Full row length in bytes**

integer

Full row length in bytes, specified as an integer. This length depends on the color depth and the pixel width of the image. For example, an RGB image has 3 bytes per pixel, so an image with width 640 has a step of 1920.



## Data — Image data

uint8 array

Image data, specified as a uint8 array.

## Object Functions

readImage Convert ROS image data into MATLAB image

writeImage Write MATLAB image to ROS image message

## Examples

### Read and Write Image Messages

Read and write a sample ROS Image message by converting it to a MATLAB image. Then, convert a MATLAB® image to ROS message.

Load sample ROS messages and inspect the image message data. `img` is a sample ROS Image message object.

```
exampleHelperROSLoadMessages
```

```
img
```

```
img =
```

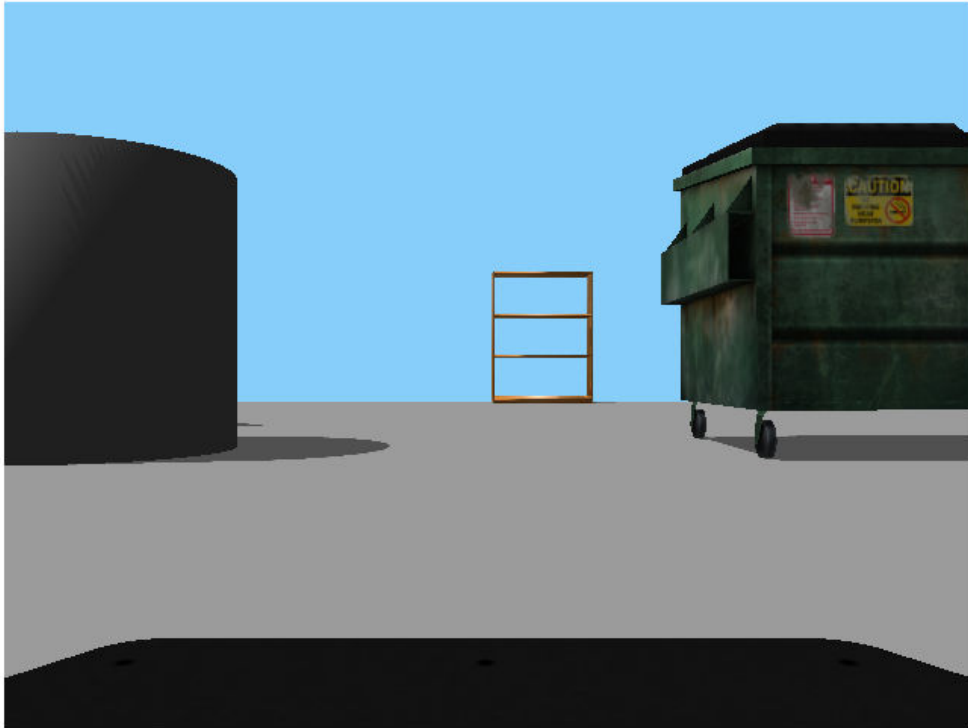
```
ROS Image message with properties:
```

```
  MessageType: 'sensor_msgs/Image'  
    Header: [1x1 Header]  
    Height: 480  
    Width: 640  
    Encoding: 'rgb8'  
  IsBigendian: 0  
    Step: 1920  
    Data: [921600x1 uint8]
```

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

Create a MATLAB image from the Image message using `readImage` and display it.

```
I = readImage(img);  
imshow(I)
```



Create a ROS Image message from a MATLAB image.

```
imgMsg = rosmesssage('sensor_msgs/Image');  
imgMsg.Encoding = 'rgb8'; % Specifies Image Encoding Type  
writeImage(imgMsg,I)  
imgMsg
```

```
imgMsg =  
  ROS Image message with properties:  
    MessageType: 'sensor_msgs/Image'
```

```
Header: [1x1 Header]
Height: 480
Width: 640
Encoding: 'rgb8'
IsBigendian: 0
Step: 1920
Data: [921600x1 uint8]
```

Use `showdetails` to show the contents of the message

### Create Blank Image Message

```
msg = rosmassage('sensor_msgs/Image')
```

```
msg =
  ROS Image message with properties:
```

```
MessageType: 'sensor_msgs/Image'
Header: [1x1 Header]
Height: 0
Width: 0
Encoding: ''
IsBigendian: 0
Step: 0
Data: [0x1 uint8]
```

Use `showdetails` to show the contents of the message

## See Also

[readImage](#) | [rosmassage](#) | [rossubscriber](#) | [writeImage](#)

## Topics

“Work with Specialized ROS Messages”

**Introduced in R2015a**

# LaserScan

Create laser scan message

## Description

The LaserScan object is an implementation of the `sensor_msgs/LaserScan` message type in ROS. The object contains meta-information about the message and the laser scan data. You can extract the ranges and angles using the `Ranges` property and the `readScanAngles` function. To access points in Cartesian coordinates, use `readCartesian`.

You can also convert this to a `lidarScan` object to use with other robotics algorithms such as `matchScans`, `robotics.VectorFieldHistogram`, or `robotics.MonteCarloLocalization`.

## Creation

## Syntax

```
scan = rosmessage('sensor_msgs/LaserScan')
```

## Description

`scan = rosmessage('sensor_msgs/LaserScan')` creates an empty LaserScan object. You can specify scan info and data using the properties, or you can get these messages off a ROS network using `rossubscriber`.

## Properties

### MessageType — Message type of ROS message

character vector

This property is read-only.

Message type of ROS message, returned as a character vector.

Data Types: char

### **Header — ROS Header message**

Header object

This property is read-only.

ROS Header message, returned as a Header object. This header message contains the MessageType, sequence (Seq), timestamp (Stamp), and FrameId. Timestamp relates to the acquisition time of the first ray in the scan.

### **AngleMin — Minimum angle of range data**

scalar

Minimum angle of range data, specified as a scalar in radians. Positive angles are measured from the forward direction of the robot.

### **AngleMax — Maximum angle of range data**

scalar

Maximum angle of range data, specified as a scalar in radians. Positive angles are measured from the forward direction of the robot.

### **AngleIncrement — Angle increment of range data**

scalar

Angle increment of range data, specified as a scalar in radians.

### **TimeIncrement — Time between individual range data points in seconds**

scalar

Time between individual range data points in seconds, specified as a scalar.

### **ScanTime — Time to complete a full scan in seconds**

scalar

Time to complete a full scan in seconds, specified as a scalar.

### **RangeMin — Minimum valid range value**

scalar

Minimum valid range value, specified as a scalar.

## **RangeMax — Maximum valid range value**

scalar

Maximum valid range value, specified as a scalar.

## **Ranges — Range readings from laser scan**

vector

Range readings from laser scan, specified as a vector. To get the corresponding angles, use `readScanAngles`.

## **Intensities — Intensity values from range readings**

vector

Intensity values from range readings, specified as a vector. If no valid intensity readings are found, this property is empty.

## **Object Functions**

<code>lidarScan</code>	Create object for storing 2-D lidar scan
<code>plot</code>	Display laser or lidar scan readings
<code>readCartesian</code>	Read laser scan ranges in Cartesian coordinates
<code>readScanAngles</code>	Return scan angles for laser scan range readings

## **Examples**

### **Inspect Sample Laser Scan Message**

Load, inspect, and display a sample laser scan message.

Create sample messages and inspect the laser scan message data. `scan` is a sample ROS `LaserScan` message object.

```
exampleHelperROSLoadMessages
scan
```

```
scan =
  ROS LaserScan message with properties:

  MessageType: 'sensor_msgs/LaserScan'
```

```
Header: [1x1 Header]
AngleMin: -0.5467
AngleMax: 0.5467
AngleIncrement: 0.0017
TimeIncrement: 0
ScanTime: 0.0330
RangeMin: 0.4500
RangeMax: 10
Ranges: [640x1 single]
Intensities: [0x1 single]
```

Use `showdetails` to show the contents of the message

Get ranges and angles from the object properties. Check that the ranges and angles are the same size.

```
ranges = scan.Ranges;
angles = scan.readScanAngles;
size(ranges)
```

```
ans = 1x2
```

```
640    1
```

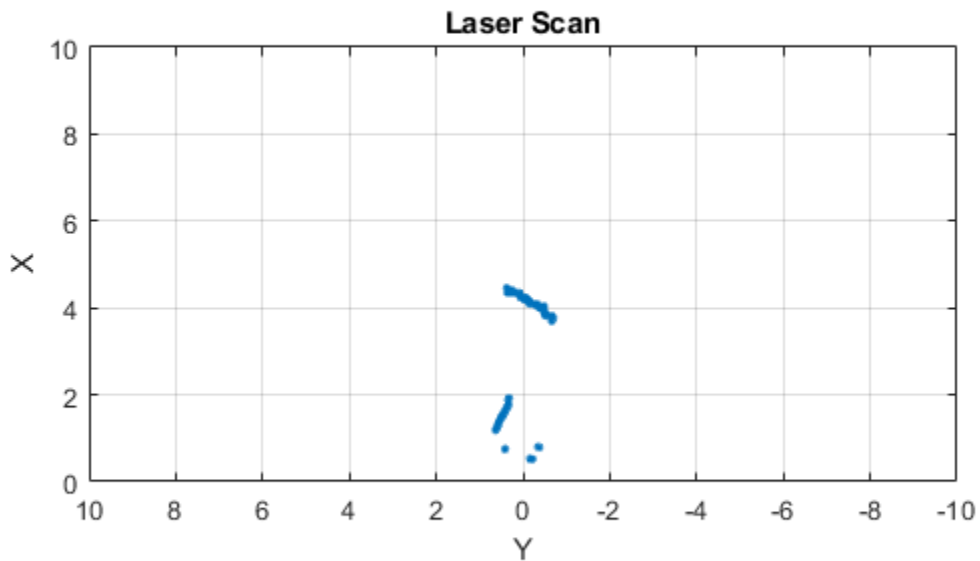
```
size(angles)
```

```
ans = 1x2
```

```
640    1
```

Display laser scan data in a figure using `plot`.

```
plot(scan)
```



## Create Empty LaserScan Message

```
scan = rosmesssage('sensor_msgs/LaserScan')
```

```
scan =
```

```
ROS LaserScan message with properties:
```

```
    MessageType: 'sensor_msgs/LaserScan'  
      Header: [1x1 Header]  
    AngleMin: 0  
    AngleMax: 0  
AngleIncrement: 0
```



```
TimeIncrement: 0
  ScanTime: 0
  RangeMin: 0
  RangeMax: 0
  Ranges: [0x1 single]
  Intensities: [0x1 single]
```

Use `showdetails` to show the contents of the message

## See Also

[lidarScan](#) | [plot](#) | [readCartesian](#) | [readScanAngles](#) | [rosmessage](#) | [rossubscriber](#) | [showdetails](#)

## Topics

“Work with Specialized ROS Messages”

**Introduced in R2016a**

## Node

Start ROS node and connect to ROS master

## Description

The `robotics.ros.Node` object represents a ROS node in the ROS network. The object enables you to communicate with the rest of the ROS network. You must create a node before you can use other ROS functionality, such as publishers, subscribers, and services.

You can create a ROS node using the `rosinit` function, or by calling `robotics.ros.Node`:

- `rosinit` — Creates a single ROS node in MATLAB. You can specify an existing ROS master, or the function creates one for you. The `Node` object is not visible.
- `robotics.ros.Node`— Creates multiple ROS nodes for use on the same ROS network in MATLAB.

## Creation

## Syntax

```
N = robotics.ros.Node(Name)
N = robotics.ros.Node(Name,Host)
N = robotics.ros.Node(Name,Host,Port)
N = robotics.ros.Node(Name,MasterURI,Port)
N = robotics.ros.Node( ____, 'NodeHost',HostName)
```

## Description

`N = robotics.ros.Node(Name)` initializes the ROS node with `Name` and tries to connect to the ROS master at default URI, `http://localhost:11311`.

`N = robotics.ros.Node(Name,Host)` tries to connect to the ROS master at the specified IP address or host name, `Host` using the default port number, `11311`.

`N = robotics.ros.Node(Name,Host,Port)` tries to connect to the ROS master with port number, `Port`.

`N = robotics.ros.Node(Name,MasterURI,Port)` tries to connect to the ROS master at the specified IP address, `MasterURI`.

`N = robotics.ros.Node( ____, 'NodeHost',HostName)` specifies the IP address or host name that the node uses to advertise itself to the ROS network. Examples include "192.168.1.1" or "comp-home". You can use any of the arguments from the previous syntaxes.

## Properties

### **Name — Name of the node**

string scalar | character vector

Name of the node, specified as a string scalar or character vector. The node name must be a valid ROS graph name. See ROS Names.

### **MasterURI — URI of the ROS master**

string scalar | character vector

URI of the ROS master, specified as a string scalar or character vector. The node is connected to the ROS master with the given URI.

### **NodeURI — URI for the node**

string scalar | character vector

URI for the node, specified as a string scalar or character vector. The node uses this URI to advertise itself on the ROS network for others to connect to it.

### **CurrentTime — Current ROS network time**

Time object

Current ROS network time, specified as a Time object. For more information, see `rostime`.

## Examples

## Create Multiple ROS Nodes

Create multiple ROS nodes. Use the `Node` object with publishers, subscribers, and other ROS functionality to specify with which node you are connecting to.

Create a ROS master.

```
master = robotics.ros.Core;
```

Initialize multiple nodes.

```
node1 = robotics.ros.Node('/test_node_1');  
node2 = robotics.ros.Node('/test_node_2');
```

Use these nodes to perform separate operations and send separate messages. A message published by `node1` can be accessed by a subscriber running in `node2`.

```
pub = robotics.ros.Publisher(node1, '/chatter', 'std_msgs/String');  
sub = robotics.ros.Subscriber(node2, '/chatter', 'std_msgs/String');
```

```
msg = rosmessage('std_msgs/String');  
msg.Data = 'Message from Node 1';
```

Send a message from `node1`. The subscriber attached to `node2` will receive the message.

```
send(pub,msg) % Sent from node 1  
pause(1) % Wait for message to update  
sub.LatestMessage
```

```
ans =  
  ROS String message with properties:  
  
  MessageType: 'std_msgs/String'  
  Data: 'Message from Node 1'
```

Use `showdetails` to show the contents of the message

Clear the ROS network of publisher, subscriber, and nodes. Delete the `Core` object to shut down the ROS master.

```
clear('pub', 'sub', 'node1', 'node2')  
clear('master')
```

## Connect to Multiple ROS Masters

Connecting to multiple ROS masters is possible using MATLAB®. These separate ROS masters do not share information and must have different port numbers. Connect ROS nodes to each master based on how you want to separate information across the network.

Create two ROS masters on different ports.

```
m1 = robotics.ros.Core; % Default port of 11311
m2 = robotics.ros.Core(12000);
```

Connect separate ROS nodes to each ROS master.

```
node1 = robotics.ros.Node('/test_node_1','localhost');
node2 = robotics.ros.Node('/test_node_2','localhost',12000);
```

Clear the ROS nodes. Shut down the ROS masters.

```
clear('node1','node2')
clear('m1','m2')
```

## See Also

[rosinit](#) | [roshutdown](#)

## Topics

“ROS Network Setup”

## External Websites

[ROS Nodes](#)

**Introduced in R2015a**

# OccupancyGrid

Create occupancy grid message

## Description

The `OccupancyGrid` object is an implementation of the `nav_msgs/OccupancyGrid` message type in ROS. The object contains meta-information about the message and the occupancy grid data. To create a `robotics.BinaryOccupancyGrid` object from a ROS message, use `readBinaryOccupancyGrid`.

---

**Note** See `robotics.OccupancyGrid` for the MATLAB representation of occupancy grids independent of ROS.

---

## Creation

## Syntax

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

## Description

`msg = rosmesssage('nav_msgs/OccupancyGrid');` creates an empty `OccupancyGrid` object. To specify map information and data, use the `map.Info` and `msg.Data` properties. You can also get the occupancy grid messages off the ROS network using `rossubscriber`.

## Properties

### **MessageType** — Message type of ROS message

character vector

This property is read-only.

Message type of ROS message, returned as a character vector.

Data Types: char

### Header — ROS Header message

Header object

This property is read-only.

ROS Header message, returned as a Header object. This header message contains the MessageType, sequence (Seq), timestamp (Stamp), and FrameId.

### Info — Information about the map

MapMetaData object

Information about the map, specified as a MapMetaData object. It contains the width, height, resolution, and origin of the map.

### Data — Map data

vector

Map data, specified as a vector. The vector is all the occupancy data from each grid location in a single 1-D array.

## Object Functions

readBinaryOccupancyGrid Read binary occupancy grid

writeBinaryOccupancyGrid Write values from grid to ROS message

## Examples

### Create Occupancy Grid from 2-D Map

Load two maps, simpleMap and complexMap, as logical matrices. Use whos to show the map.

```
load exampleMaps.mat
whos *Map*
```

Name	Size	Bytes	Class	Attributes
------	------	-------	-------	------------

complexMap	41x52	2132	logical
emptyMap	26x27	702	logical
simpleMap	26x27	702	logical
ternaryMap	501x501	2008008	double

Create a ROS message from `simpleMap` using a `BinaryOccupancyGrid` object. Write the `OccupancyGrid` message using `writeBinaryOccupancyGrid`.

```
bogMap = robotics.BinaryOccupancyGrid(double(simpleMap));
mapMsg = rosmessage('nav_msgs/OccupancyGrid');
writeBinaryOccupancyGrid(mapMsg,bogMap)
mapMsg
```

```
mapMsg =
  ROS OccupancyGrid message with properties:
```

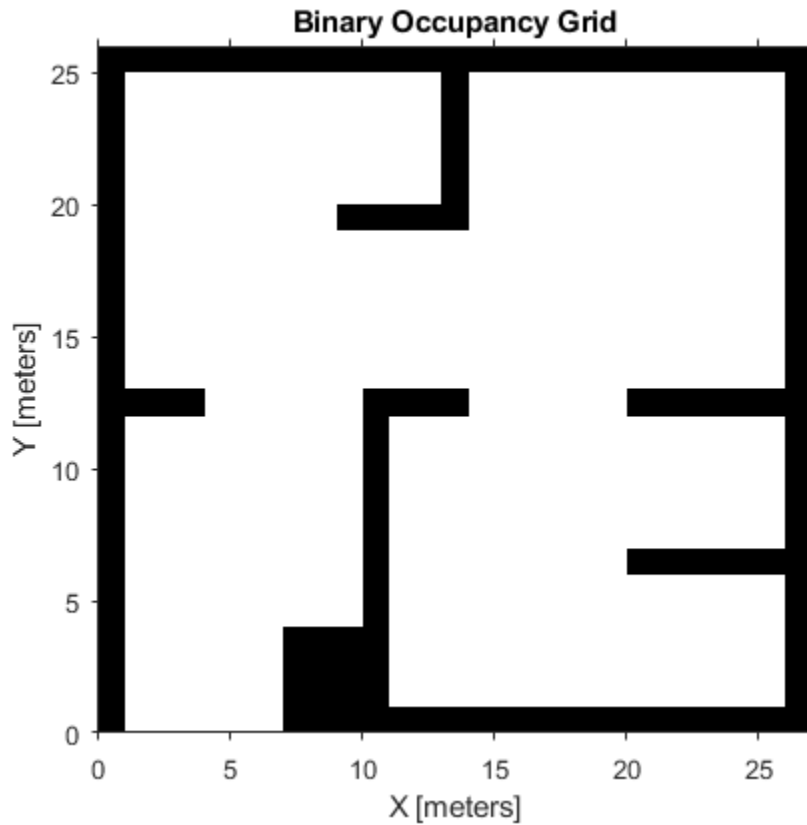
```
  MessageType: 'nav_msgs/OccupancyGrid'
  Header: [1x1 Header]
  Info: [1x1 MapMetaData]
  Data: [702x1 int8]
```

Use `showdetails` to show the contents of the message

Use `readBinaryOccupancyGrid` to convert the ROS message to a `BinaryOccupancyGrid` object. Use the object function `show` to display the map.

```
bogMap2 = readBinaryOccupancyGrid(mapMsg);
show(bogMap2);
```





## See Also

`readBinaryOccupancyGrid` | `robotics.BinaryOccupancyGrid` | `rosmessage` | `rossubscriber` | `writeBinaryOccupancyGrid`

## Topics

“Occupancy Grids”

**Introduced in R2015a**

# PointCloud2

Access point cloud messages

## Description

The `PointCloud2` object is an implementation of the `sensor_msgs/PointCloud2` message type in ROS. The object contains meta-information about the message and the point cloud data. To access the actual data, use `readXYZ` to get the point coordinates and `readRGB` to get the color information, if available.

## Creation

## Syntax

```
ptcloud = rosmessage('sensor_msgs/PointCloud2')
```

## Description

`ptcloud = rosmessage('sensor_msgs/PointCloud2')` creates an empty `PointCloud2` object. To specify point cloud data, use the `ptcloud.Data` property. You can also get point cloud data messages off the ROS network using `rossubscriber`.

## Properties

### **PreserveStructureOnRead — Preserve the shape of point cloud matrix**

`false` (default) | `true`

This property is read-only.

Preserve the shape of point cloud matrix, specified as `false` or `true`. When the property is `true`, the output data from `readXYZ` and `readRGB` are returned as matrices instead of vectors.

**MessageType — Message type of ROS message**

character vector

This property is read-only.

Message type of ROS message, returned as a character vector.

Data Types: char

**Header — ROS Header message**

Header object

This property is read-only.

ROS Header message, returned as a Header object. This header message contains the MessageType, sequence (Seq), timestamp (Stamp), and FrameId.

**Height — Point cloud height in pixels**

integer

Point cloud height in pixels, specified as an integer.

**Width — Point cloud width in pixels**

integer

Point cloud width in pixels, specified as an integer.

**IsBigendian — Image byte sequence**

true | false

Image byte sequence, specified as a true or false.

- `true` —Big endian sequence. Stores the most significant byte in the smallest address.
- `false` —Little endian sequence. Stores the least significant byte in the smallest address.

The Big endian sequence stores the most significant byte in the smallest address. The Little endian sequence stores the least significant byte in the smallest address.

**PointStep — Length of a point in bytes**

integer

Length of a point in bytes, specified as an integer.

## RowStep — Full row length in bytes

integer

Full row length in bytes, specified as an integer. The row length equals the `PointStep` property multiplied by the `Width` property.

## Data — Point cloud data

uint8 array

Point cloud data, specified as a `uint8` array. To access the data, use the “Object Functions” on page 1-38.

## Object Functions

<code>readAllFieldNames</code>	Get all available field names from ROS point cloud
<code>readField</code>	Read point cloud data based on field name
<code>readRGB</code>	Extract RGB values from point cloud data
<code>readXYZ</code>	Extract XYZ coordinates from point cloud data
<code>scatter3</code>	Display point cloud in scatter plot
<code>showdetails</code>	Display all ROS message contents

## Examples

### Inspect Point Cloud Image

Access and visualize the data inside a point cloud message.

Create sample ROS messages and inspect a point cloud image. `ptcloud` is a sample ROS `PointCloud2` message object.

```
exampleHelperROSLoadMessages  
ptcloud
```

```
ptcloud =  
  ROS PointCloud2 message with properties:  
  
  PreserveStructureOnRead: 0  
    MessageType: 'sensor_msgs/PointCloud2'  
      Header: [1x1 Header]  
      Height: 480
```

```
      Width: 640
IsBigendian: 0
PointStep: 32
RowStep: 20480
IsDense: 0
Fields: [4x1 PointField]
Data: [9830400x1 uint8]
```

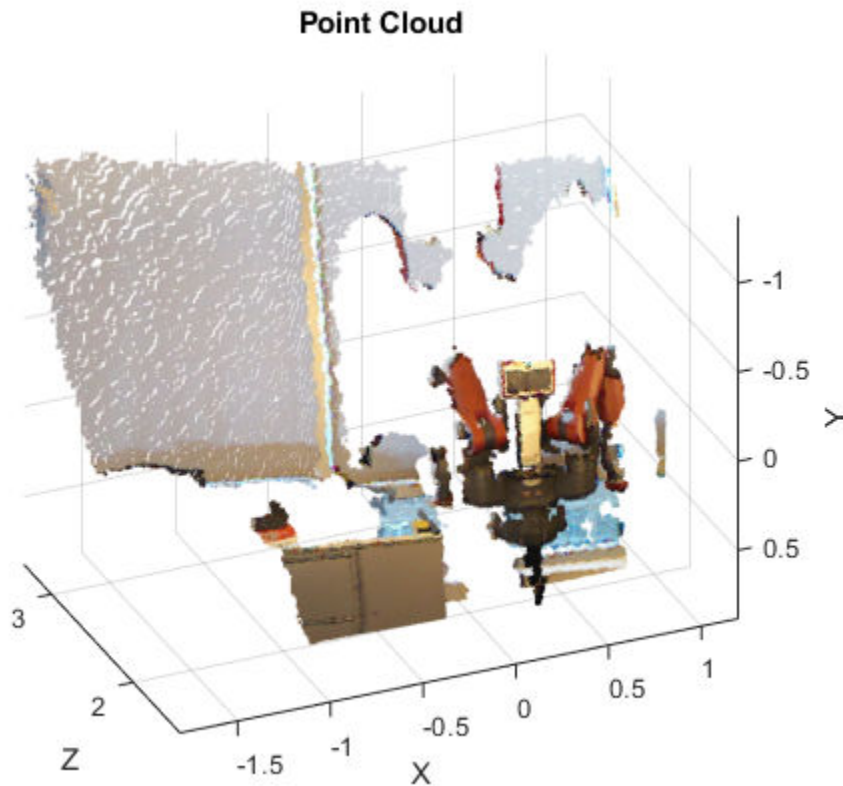
Use `showdetails` to show the contents of the message

Get RGB info and xyz-coordinates from the point cloud using `readXYZ` and `readRGB`.

```
xyz = readXYZ(ptcloud);
rgb = readRGB(ptcloud);
```

Display the point cloud in a figure using `scatter3`.

```
scatter3(ptcloud)
```



## Create pointCloud Object Using Point Cloud Message

Convert a Robotics System Toolbox™ point cloud message into a Computer Vision System Toolbox™ pointCloud object.

Load sample messages.

```
exampleHelperROSLoadMessages
```

Convert a ptcloud message to the pointCloud object.

```
pcobj = pointCloud(readXYZ(ptcloud), 'Color', uint8(255*readRGB(ptcloud)))
```

```
pcobj =  
  pointCloud with properties:  
  
    Location: [307200x3 single]  
      Color: [307200x3 uint8]  
    Normal: []  
  Intensity: []  
    Count: 307200  
  XLimits: [-1.8147 1.1945]  
  YLimits: [-1.3714 0.8812]  
  ZLimits: [1.4190 3.3410]
```

## See Also

[readAllFieldNames](#) | [readField](#) | [readRGB](#) | [readXYZ](#) | [rosmessage](#) | [rossubscriber](#) | [scatter3](#) | [showdetails](#)

## Topics

“Work with Specialized ROS Messages”

**Introduced in R2015a**

## rosdevice

Connect to remote ROS device

### Description

The `rosdevice` object is used to create a connection with a ROS device. The object contains the necessary login information and other parameters of the ROS distribution. Once a connection is made using `rosdevice`, you can run and stop a ROS core or ROS nodes and check the status of the ROS network. Before running ROS nodes, you must connect MATLAB to the ROS network using `rosinit`.

You can deploy ROS nodes to a ROS device using Simulink® models. For an example, see “Generate a Standalone ROS Node from Simulink®”.

---

**Note** To connect to a ROS device, an SSH server must be installed on the device.

---

### Creation

### Syntax

```
device = rosdevice(deviceAddress,username,password)
device = rosdevice
```

### Description

`device = rosdevice(deviceAddress,username,password)` creates a `rosdevice` object connected to the ROS device at the specified address and with the specified user name and password.

`device = rosdevice` creates a `rosdevice` object connected to a ROS device using the saved values for `deviceAddress`, `username`, and `password`.



## Properties

### **DeviceAddress — Hostname or IP address of the ROS device**

character vector

This property is read-only.

Hostname or IP address of the ROS device, specified as a character vector.

Example: '192.168.1.10'

Example: 'samplehost.foo.com'

### **UserName — User name used to connect to the ROS device**

character vector

This property is read-only.

User name used to connect to the ROS device, specified as a character vector.

Example: 'user'

### **ROSFolder — Location of ROS installation**

character vector

Location of ROS installation, specified as a character vector. If a folder is not specified, MATLAB tries to determine the correct folder for you. When you deploy a ROS node, set this value from Simulink in the **Configuration Parameters** dialog box, under **Hardware Implementation**.

Example: '/opt/ros/hydro'

### **CatkinWorkspace — Catkin folder where models are deployed on device**

character vector

Catkin folder where models are deployed on device, specified as a character vector. When you deploy a ROS node, set this value from Simulink in the **Configuration Parameters** dialog box, under **Hardware Implementation**.

Example: '~/catkin\_ws\_test'

### **AvailableNodes — Nodes available to run on ROS device**

cell array of character vectors

This property is read-only.

Nodes available to run on ROS device, returned as a cell array of character vectors. Nodes are only listed if they are part of the `CatkinWorkspace` and have been deployed to the device using `Simulink`.

Example: `{'robotcontroller','publishernode'}`

## Object Functions

<code>runNode</code>	Start ROS node
<code>stopNode</code>	Stop ROS node
<code>isNodeRunning</code>	Determine if ROS node is running
<code>runCore</code>	Start ROS core
<code>stopCore</code>	Stop ROS core
<code>isCoreRunning</code>	Determine if ROS core is running
<code>system</code>	Execute system command on device
<code>putFile</code>	Copy file to device
<code>getFile</code>	Get file from device
<code>deleteFile</code>	Delete file from device
<code>dir</code>	List folder contents on device
<code>openShell</code>	Open interactive command shell to device

## Examples

### Run ROS Core on ROS Device

Connect to a remote ROS device and start a ROS core. The ROS core is needed to run ROS nodes to communicate via a ROS network. You can run and stop a ROS core or node and check their status using a `rosdevice` object.

Create a connection to a ROS device. Specify the address, user name, and password of your specific ROS device. The device contains information about the ROS device, including the available ROS nodes that can be run using `runNode`.

```
ipaddress = '192.168.154.131';  
d = rosdevice(ipaddress,'user','password')
```

```
d =
```

rosdevice with properties:

```
DeviceAddress: '192.168.154.131'  
Username: 'user'  
ROSFolder: '/opt/ros/hydro'  
CatkinWorkspace: '~/catkin_ws_test'  
AvailableNodes: {'robotcontroller' 'robotcontroller2'}
```

Run a ROS core and check if it is running.

```
runCore(d)  
running = isCoreRunning(d)
```

```
running =
```

```
logical
```

```
1
```

Stop the ROS core and confirm that it is no longer running.

```
stopCore(d)  
running = isCoreRunning(d)
```

```
running =
```

```
logical
```

```
0
```

### Run ROS Node on ROS Device

Connect to a remote ROS device and start a ROS node. First, run a ROS core so that ROS nodes can communicate via a ROS network. You can run and stop a ROS core or node and check their status using a `rosdevice` object.

Create a connection to a ROS device. Specify the address, user name and password of your specific ROS device. The device contains information about the ROS device, including the available ROS nodes that can be run using `runNode`.

```
ipaddress = '192.168.203.129';  
d = rosdevice(ipaddress, 'user', 'password');  
d.ROSFolder = '/opt/ros/hydro';  
d.CatkinWorkspace = '~/catkin_ws_test'
```

```
d =
```

```
rosdevice with properties:
```

```
DeviceAddress: '192.168.203.129'  
Username: 'user'  
ROSFolder: '/opt/ros/hydro'  
CatkinWorkspace: '~/catkin_ws_test'  
AvailableNodes: {'robotcontroller' 'robotcontroller2'}
```

Run a ROS core. Connect MATLAB® to the ROS master using `rosinit`. This core enables you to run ROS nodes on your ROS device.

```
runCore(d)  
rosinit(d.DeviceAddress, 11311)
```

```
Initializing global node /matlab_global_node_12272 with NodeURI http://192.168.203.1:63
```

Check the available ROS nodes on the connected ROS device. These nodes were generated from Simulink® models following the process in the “Generate a Standalone ROS Node from Simulink®” example.

```
d.AvailableNodes
```

```
ans =
```

```
1×2 cell array
```

```
{'robotcontroller'} {'robotcontroller2'}
```

Run a ROS node. specifying the node name. Check if the node is running.

```
runNode(d, 'robotcontroller')
running = isNodeRunning(d, 'robotcontroller')
```

```
running =
    logical
    1
```

Stop the ROS node. Disconnect from the ROS network. Stop the ROS core.

```
stopNode(d, 'robotcontroller')
roshutdown
stopCore(d)
```

```
Shutting down global node /matlab_global_node_12272 with NodeURI http://192.168.203.1:
```

## Run Multiple ROS Nodes

Run multiple ROS nodes on a connected ROS device. ROS nodes can be generated using Simulink® models to perform different tasks on the ROS network. These nodes are then deployed on a ROS device and can be run independently of Simulink®.

This example uses two different Simulink models that have been deployed as ROS nodes. See “Generate a Standalone ROS Node from Simulink®” and follow the instructions to generate and deploy a ROS node. The 'robotcontroller' node sends velocity commands to a robot to navigate it to a given point. The 'robotcontroller2' node uses the same model, but doubles the linear velocity to drive the robot faster.

Create a connection to a ROS device. Specify the address, user name, and password of your specific ROS device. The device contains information about the ROS device, including the available ROS nodes that can be run using runNode.

```
ipaddress = '192.168.203.129';
d = rosdevice(ipaddress, 'user', 'password')
```

```
d =
```

```
rosdevice with properties:
```

```
DeviceAddress: '192.168.203.129'  
Username: 'user'  
ROSFolder: '/opt/ros/indigo'  
CatkinWorkspace: '~/catkin_ws_test'  
AvailableNodes: {'robotcontroller' 'robotcontroller2'}
```

Run a ROS core. The ROS Core is the master enables you to run ROS nodes on your ROS device. Connect MATLAB® to the ROS master using `rosinit`. For this example, the port is set to 11311. `rosinit` can automatically select a port for you without specifying this input.

```
runCore(d)  
rosinit(d.DeviceAddress,11311)
```

```
Initializing global node /matlab_global_node_15972 with NodeURI http://192.168.203.1:5
```

Check the available ROS nodes on the connected ROS device. The nodes listed in this example were generated from Simulink® models following the process in the “Generate a Standalone ROS Node from Simulink®” example. Two separate nodes are generated, 'robotcontroller' and 'robotcontroller2', which have the linear velocity set to 1 and 2 in the model respectively.

```
d.AvailableNodes
```

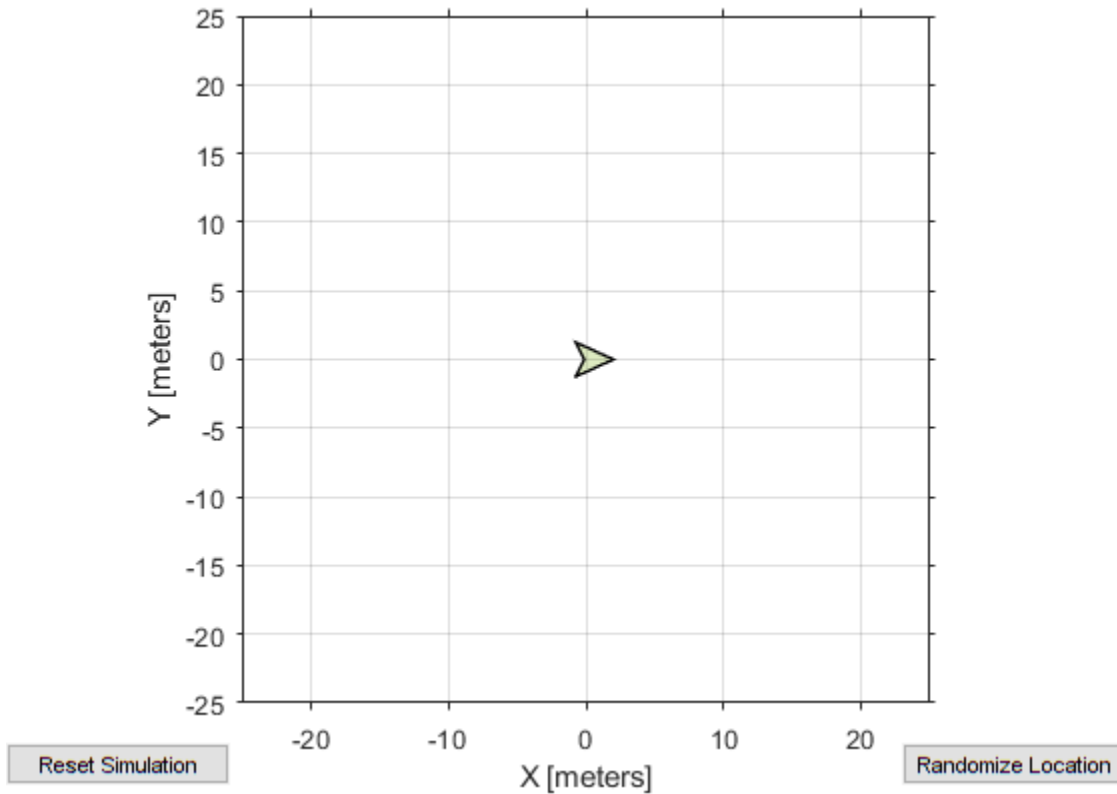
```
ans =
```

```
1x2 cell array
```

```
{'robotcontroller'} {'robotcontroller2'}
```

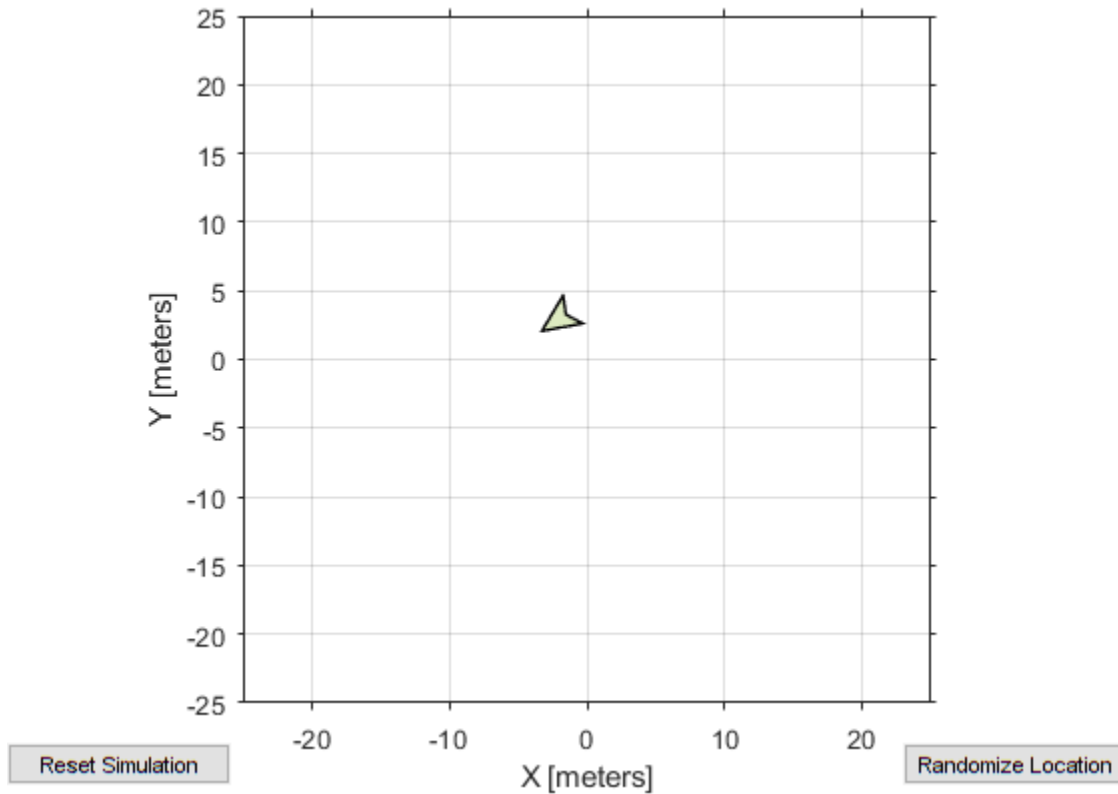
Start up the Robot Simulator using `ExampleHelperSimulinkRobotROS`. This simulator automatically connects to the ROS master on the ROS device. You will use this simulator to run a ROS node and control the robot.

```
sim = ExampleHelperSimulinkRobotROS;
```



Run a ROS node, specifying the node name. The 'robotcontroller' node commands the robot to a specific location ([ -10 10]). Wait to see the robot drive.

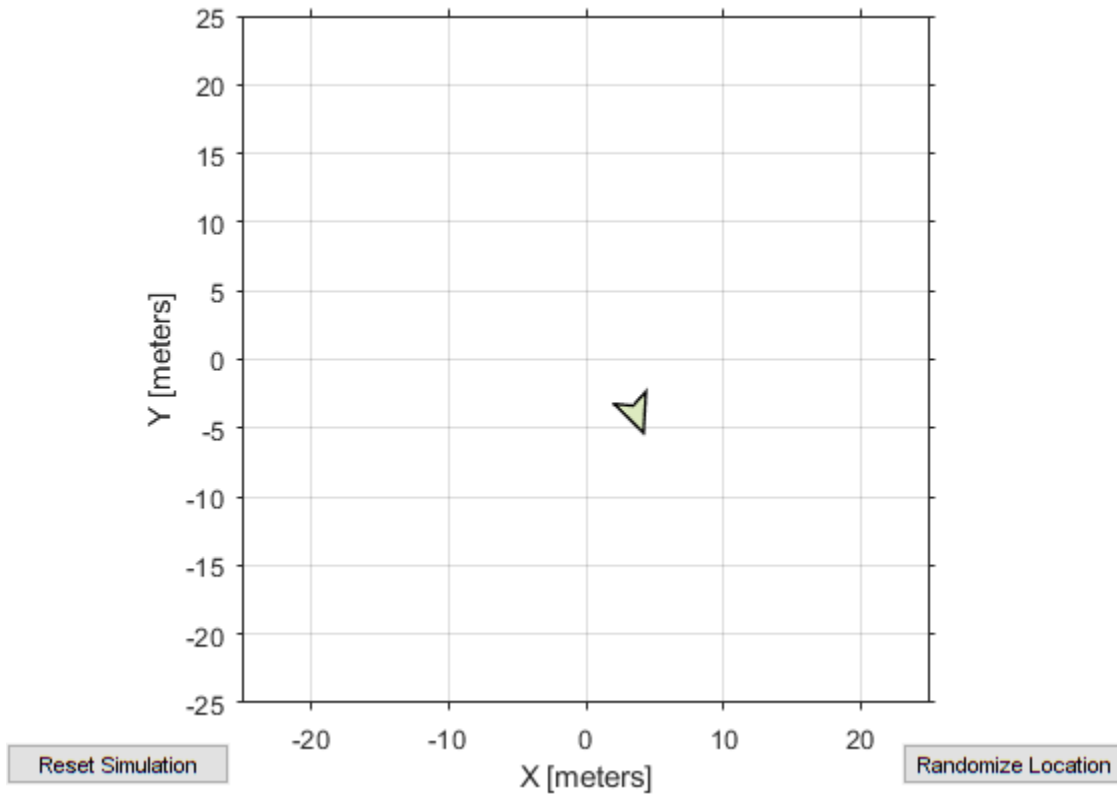
```
runNode(d, 'robotcontroller')  
pause(10)
```



Reset the Robot Simulator to reset the robot position. Alternatively, click **Reset Simulation**. Because the node is still running, the robot continues back to the specific location. To stop sending commands, stop the node.

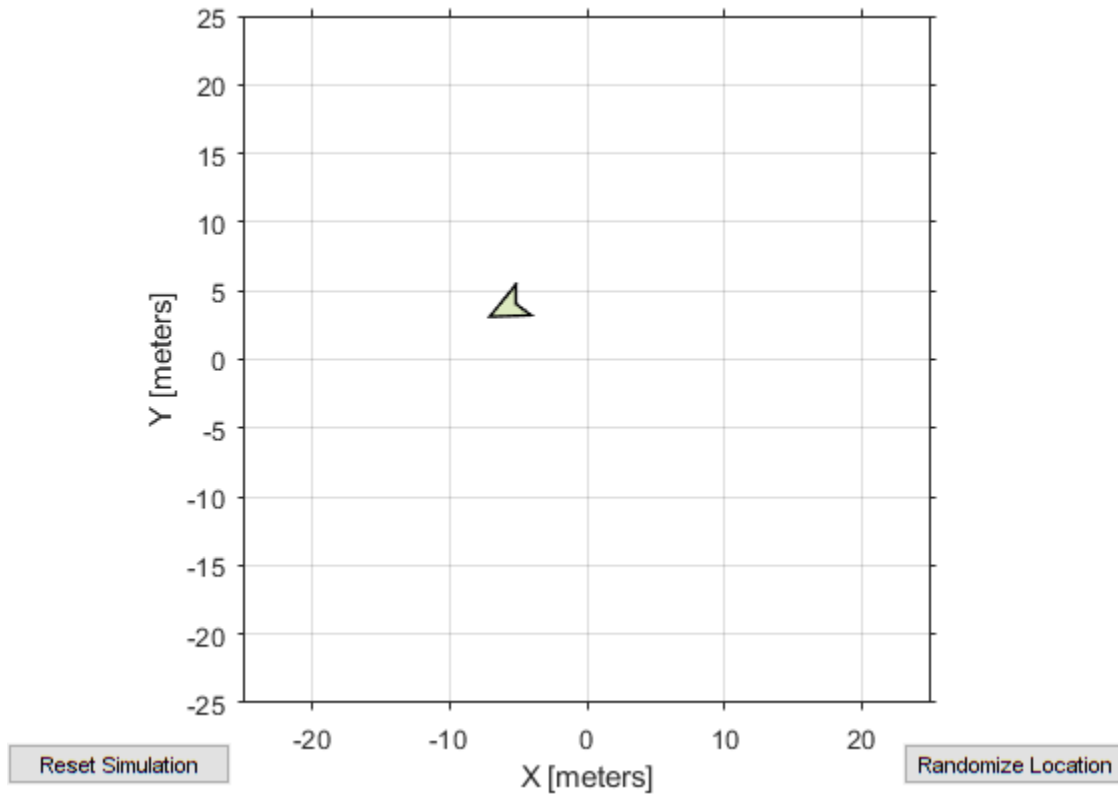
```
resetSimulation(sim.Simulator)  
pause(5)  
stopNode(d, 'robotcontroller')
```





Run the 'robotcontroller2' node. This model drives the robot with twice the linear velocity. Reset the robot position. Wait to see the robot drive. You should see a wider turn due to the increased velocity.

```
runNode(d, 'robotcontroller2')
resetSimulation(sim.Simulator)
pause(10)
```



Close the simulator. Stop the ROS node. Disconnect from the ROS network and stop the ROS core.

```
close
stopNode(d, 'robotcontroller2')
roshutdown
stopCore(d)
```

```
Shutting down global node /matlab_global_node_15972 with NodeURI http://192.168.203.1:5
```

## See Also

[isNodeRunning](#) | [runCore](#) | [runNode](#) | [stopNode](#)

## **Topics**

“Generate a Standalone ROS Node from Simulink®”

**Introduced in R2016b**

# TransformStamped

Create transformation message

## Description

The TransformStamped object is an implementation of the `geometry_msgs/TransformStamped` message type in ROS. The object contains meta-information about the message itself and the transformation. The transformation has a translational and rotational component.

## Creation

## Syntax

```
tform = getTransform(tftree, targetframe, sourceframe)
```

## Description

`tform = getTransform(tftree, targetframe, sourceframe)` returns the latest known transformation between two coordinate frames. Transformations are structured as a 3-D translation (3-element vector) and a 3-D rotation (quaternion).

## Properties

### MessageType — Message type of ROS message

character vector

This property is read-only.

Message type of ROS message, returned as a character vector.

Data Types: char

**Header — ROS Header message**

Header object

This property is read-only.

ROS Header message, returned as a Header object. This header message contains the MessageType, sequence (Seq), timestamp (Stamp), and FrameId.

**ChildFrameID — Second coordinate frame to transform point into**

character vector

Second coordinate frame to transform point into, specified as a character vector.

**Transform — Transformation message**

Transform object

This property is read-only.

Transformation message, specified as a Transform object. The object contains the MessageType with a Translation vector and Rotation quaternion.

## Object Functions

apply Transform message entities into target frame

## Examples

**Inspect Sample TransformStamped Object**

This example looks at the TransformStamped object to show the underlying structure of a TransformStamped ROS message. After setting up a network and transformations, you can create a transformation tree and get transformations between specific coordinate systems. Using showdetails lets you inspect the information in the transformation. It contains the ChildFrameId, Header, and Transform.

Start ROS network and setup transformations.

```
rosinit
```

```
Initializing ROS master on http://ah-sradford:11311/.
Initializing global node /matlab_global_node_28474 with NodeURI http://ah-sradford:6500
```

```
exampleHelperROSStartTfPublisher
```

Create transformation tree and wait for tree to update. Get the transform between the robot base and its camera center.

```
tftree = rostf;
waitForTransform(tftree, 'camera_center', 'robot_base');
tform = getTransform(tftree, 'camera_center', 'robot_base');
```

Inspect the TransformStamped object.

```
showdetails(tform)
```

```
ChildFrameId : robot_base
Header
  Seq      : 20
  FrameId  : camera_center
  Stamp
    Sec    : 1512065171
    Nsec   : 111000064
Transform
  Translation
    X : 0.4999999999999998
    Y : 0
    Z : -1
  Rotation
    X : 0
    Y : -0.7071067811865475
    Z : 0
    W : 0.7071067811865476
```

Access the Translation vector inside the Transform property.

```
trans = tform.Transform.Translation
```

```
trans =
ROS Vector3 message with properties:

  MessageType: 'geometry_msgs/Vector3'
    X: 0.5000
    Y: 0
    Z: -1
```

Use `showdetails` to show the contents of the message

Shutdown ROS network.

```
roshutdown
```

```
Shutting down global node /matlab_global_node_28474 with NodeURI http://ah-sradford:6255
Shutting down ROS master on http://ah-sradford:11311/.
```

## Apply Transformation using TransformStamped Object

Apply a transformation from a `TransformStamped` object to a `PointStamped` message.

Start ROS network and setup transformations.

```
roslaunch
```

```
Initializing ROS master on http://ah-sradford:11311/.
Initializing global node /matlab_global_node_71764 with NodeURI http://ah-sradford:6255
```

```
exampleHelperROSstartTfPublisher
```

Create transformation tree and wait for tree to update. Get the transform between the robot base and its camera center. Inspect the transformation.

```
tftree = rostf;
waitForTransform(tftree, 'camera_center', 'robot_base');
tform = getTransform(tftree, 'camera_center', 'robot_base');
showdetails(tform)
```

```
ChildFrameId : robot_base
Header
  Seq      : 15
  FrameId  : camera_center
  Stamp
    Sec    : 1512064258
    Nsec   : 689999936
Transform
  Translation
    X : 0.4999999999999998
    Y : 0
```

```
Z : -1
Rotation
X : 0
Y : -0.7071067811865475
Z : 0
W : 0.7071067811865476
```

Create point to transform. You could also get this point message off the ROS network.

```
pt = rosmesssage('geometry_msgs/PointStamped');
pt.Header.FrameId = 'camera_center';
pt.Point.X = 3;
pt.Point.Y = 1.5;
pt.Point.Z = 0.2;
```

Apply the transformation to the point.

```
tfpt = apply(tform,pt);
```

Shutdown ROS network.

```
roshutdowndown
```

```
Shutting down global node /matlab_global_node_71764 with NodeURI http://ah-sradford:62
Shutting down ROS master on http://ah-sradford:11311/.
```

## See Also

### Functions

[apply](#) | [getTransform](#) | [rostop](#) | [transform](#) | [waitForTransform](#)

### Topics

“Access the tf Transformation Tree in ROS”

### Introduced in R2015a



# robotics.AimingConstraint class

**Package:** robotics

Create aiming constraint for pointing at a target location

## Description

The `AimingConstraint` object describes a constraint that requires the z-axis of one body (the end effector) to aim at a target point on another body (the reference body). This constraint is satisfied if the z-axis of the end-effector frame is within an angular tolerance in any direction of the line connecting the end-effector origin and the target point. The position of the target point is defined relative to the reference body.

Constraint objects are used in `GeneralizedInverseKinematics` objects to specify multiple kinematic constraints on a robot.

For an example that uses multiple constraint objects, see “Plan a Reaching Trajectory With Multiple Kinematic Constraints”.

## Construction

`aimConst = robotics.AimingConstraint(endeffector)` returns an aiming constraint object that represents a constraint on a body specified by `endeffector`.

`aimConst = robotics.AimingConstraint(endeffector, Name, Value)` returns an aiming constraint object with each specified property name set to the specified value by one or more `Name, Value` pair arguments.

## Input Arguments

**endeffector** — End-effector name

string scalar | character vector

End-effector name, specified as a string scalar or character vector. When using this constraint with `GeneralizedInverseKinematics`, the name must match a body specified in the robot model (`RigidBodyTree`).

Example: "left\_palm"

Data Types: char | string

## Properties

### **EndEffector — Name of the end effector**

string scalar | character vector

Name of the end effector, specified as a string scalar or character vector. When using this constraint with `GeneralizedInverseKinematics`, the name must match a body specified in the robot model (`RigidBodyTree`).

Example: "left\_palm"

Data Types: char | string

### **ReferenceBody — Name of the reference body frame**

' ' (default) | string scalar | character vector

Name of the reference body frame, specified as a string scalar or character vector. The default ' ' indicates that the constraint is relative to the base of the robot model. When using this constraint with `GeneralizedInverseKinematics`, the name must match a body specified in the robot model (`RigidBodyTree`).

Data Types: char | string

### **TargetPoint — Position of the target relative to the reference body**

[0 0 0] (default) | [x y z] vector

Position of the target relative to the reference body, specified as an [x y z] vector. The constraint uses the line between the origin of the `EndEffector` body frame and this target point for maintaining the specified `AngularTolerance`.

### **AngularTolerance — Maximum allowed angle**

0 (default) | numeric scalar

Maximum allowed angle between the z-axis of the end-effector frame and the line connecting the end-effector origin to the target point, specified as a numeric scalar in radians.

### **Weights — Weight of the constraint**

1 (default) | numeric scalar

Weight of the constraint, specified as a numeric scalar. This weight is used with the `Weights` property of all the constraints specified in `GeneralizedInverseKinematics` to properly balance each constraint.

## Extended Capabilities

### C/C++ Code Generation

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

### See Also

#### Classes

`robotics.GeneralizedInverseKinematics` | `robotics.OrientationTarget` | `robotics.PoseTarget` | `robotics.PositionTarget`

#### Topics

“Plan a Reaching Trajectory With Multiple Kinematic Constraints”

**Introduced in R2017a**

## robotics.BinaryOccupancyGrid class

**Package:** robotics

Create occupancy grid with binary values

### Description

BinaryOccupancyGrid creates a 2-D occupancy grid object, which you can use to represent and visualize a robot workspace, including obstacles. The integration of sensor data and position estimates create a spatial representation of the approximate locations of the obstacles.

Occupancy grids are used in robotics algorithms such as path planning. They are also used in mapping applications, such as for finding collision-free paths, performing collision avoidance, and calculating localization. You can modify your occupancy grid to fit your specific application.

Each cell in the occupancy grid has a value representing the occupancy status of that cell. An occupied location is represented as `true` (1) and a free location is represented as `false` (0).

The two coordinate systems supported are world and grid coordinates. The world coordinates origin is defined by `GridLocationInWorld`, which defines the bottom-left corner of the map. The number and size of grid locations are defined by the `Resolution`. Also, the first grid location with index (1, 1) begins in the top-left corner of the grid.

### Construction

`map = robotics.BinaryOccupancyGrid(width,height)` creates a 2-D binary occupancy grid representing a work space of `width` and `height` in meters. The default grid resolution is one cell per meter.

`map = robotics.BinaryOccupancyGrid(width,height,resolution)` creates a grid with `resolution` specified in cells per meter. The map is in world coordinates by default. You can use any of the arguments from previous syntaxes.

`map = robotics.BinaryOccupancyGrid(rows,cols,resolution,"grid")`  
creates a 2-D binary occupancy grid of size (rows,cols).

`map = robotics.BinaryOccupancyGrid(p)` creates a grid from the values in matrix `p`. The size of the grid matches the size of the matrix, with each cell value interpreted from its location in the matrix. `p` contains any numeric or logical type with zeros (0) and ones (1).

`map = robotics.BinaryOccupancyGrid(p,resolution)` creates a `BinaryOccupancyGrid` object with `resolution` specified in cells per meter.

## Input Arguments

### **width — Map width**

double in meters

Map width, specified as a double in meters.

Data Types: double

### **height — Map height**

double in meters

Map height, specified as a double in meters.

Data Types: double

### **resolution — Grid resolution**

1 (default) | double in cells per meter

Grid resolution, specified as a double in cells per meter.

Data Types: double

### **p — Input occupancy grid**

matrix of ones and zeros

Input occupancy grid, specified as a matrix of ones and zeros. The size of the grid matches the size of the matrix. Each matrix element corresponds to an occupied location (1) or free location (0).

## Properties

### **GridSize — Number of rows and columns in grid**

two-element horizontal vector

Number of rows and columns in grid, stored as a two-element horizontal vector of the form `[rows cols]`. This value is read only.

### **Resolution — Grid resolution**

1 (default) | scalar in cells per meter

Grid resolution, stored as a scalar in cells per meter. This value is read only.

Data Types: double

### **XWorldLimits — Minimum and maximum values of x-coordinates**

two-element vector

Minimum and maximum values of  $x$ -coordinates, stored as a two-element horizontal vector of the form `[min max]`. These values indicate the world range of the  $x$ -coordinates in the grid. This value is read only.

### **YWorldLimits — Minimum and maximum values of y-coordinates**

two-element vector

Minimum and maximum values of  $y$ -coordinates, stored as a two-element vector of the form `[min max]`. These values indicate the world range of the  $y$ -coordinates in the grid. This value is read only.

### **GridLocationInWorld — $[x,y]$ world coordinates of grid**

`[0 0]` (default) | two-element vector

$[x, y]$  world coordinates of the bottom-left corner of the grid, specified as a two-element vector.

Data Types: double

## Methods

copy	Copy array of handle objects
getOccupancy	Get occupancy value for one or more positions
grid2world	Convert grid indices to world coordinates
inflate	Inflate each occupied grid location
occupancyMatrix	Convert occupancy grid to matrix
setOccupancy	Set occupancy value for one or more positions
show	Show occupancy grid values
world2grid	Convert world coordinates to grid indices

## Examples

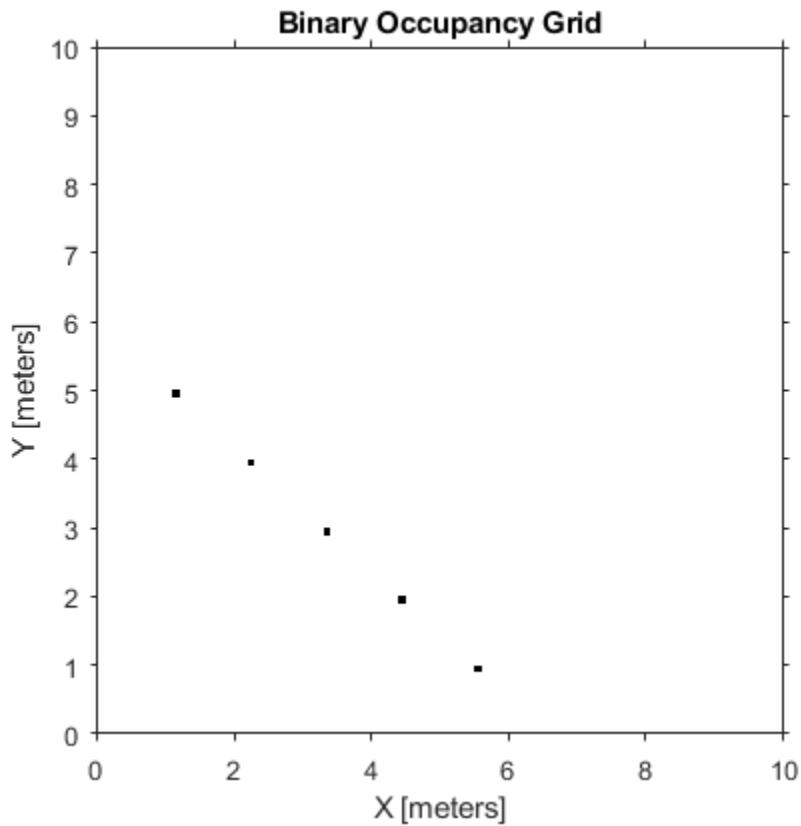
### Create and Modify Binary Occupancy Grid

Create a 10m x 10m empty map.

```
map = robotics.BinaryOccupancyGrid(10,10,10);
```

Set occupancy of world locations and show map.

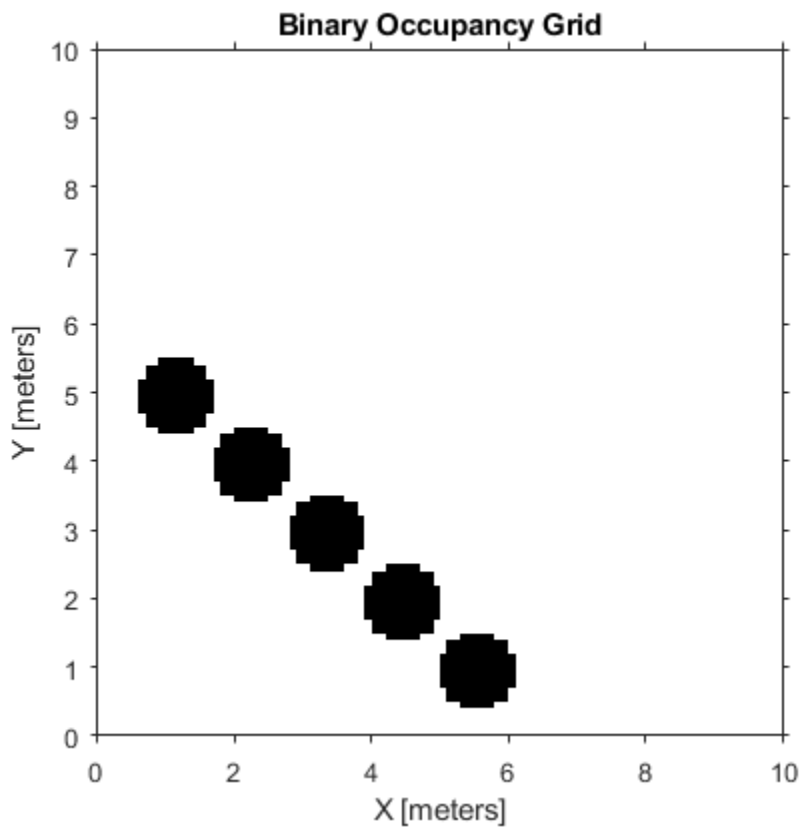
```
map = robotics.BinaryOccupancyGrid(10,10,10);  
x = [1.2; 2.3; 3.4; 4.5; 5.6];  
y = [5.0; 4.0; 3.0; 2.0; 1.0];  
  
setOccupancy(map, [x y], ones(5,1))  
figure  
show(map)
```



Inflate occupied locations by a given radius.

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





Get grid locations from world locations.

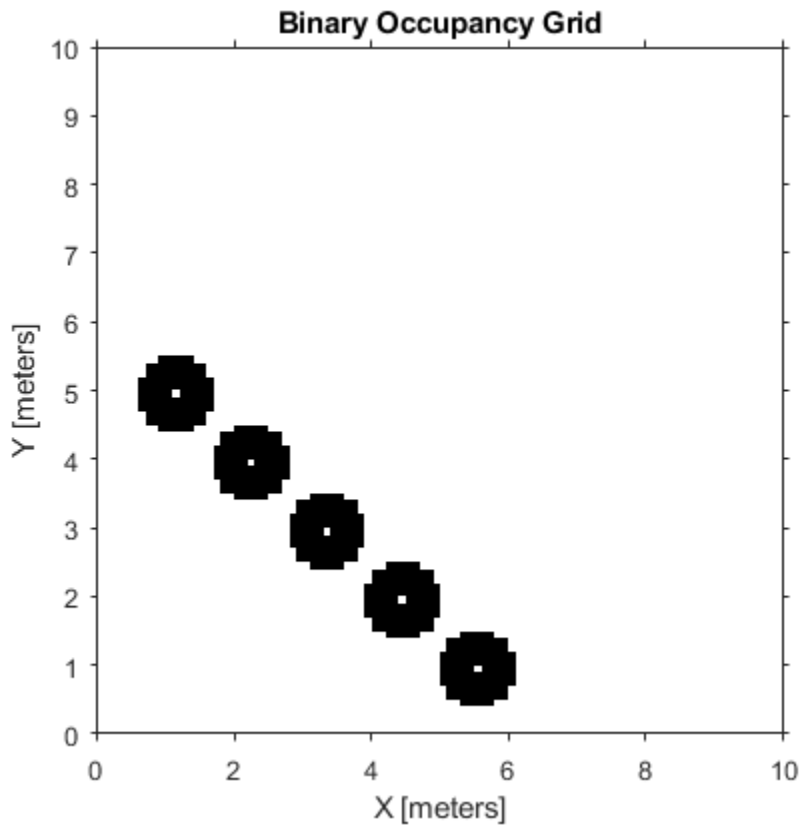
```
ij = world2grid(map, [x y]);
```

Set grid locations to free locations.

```
setOccupancy(map, ij, zeros(5,1), 'grid')
```

```
figure
```

```
show(map)
```



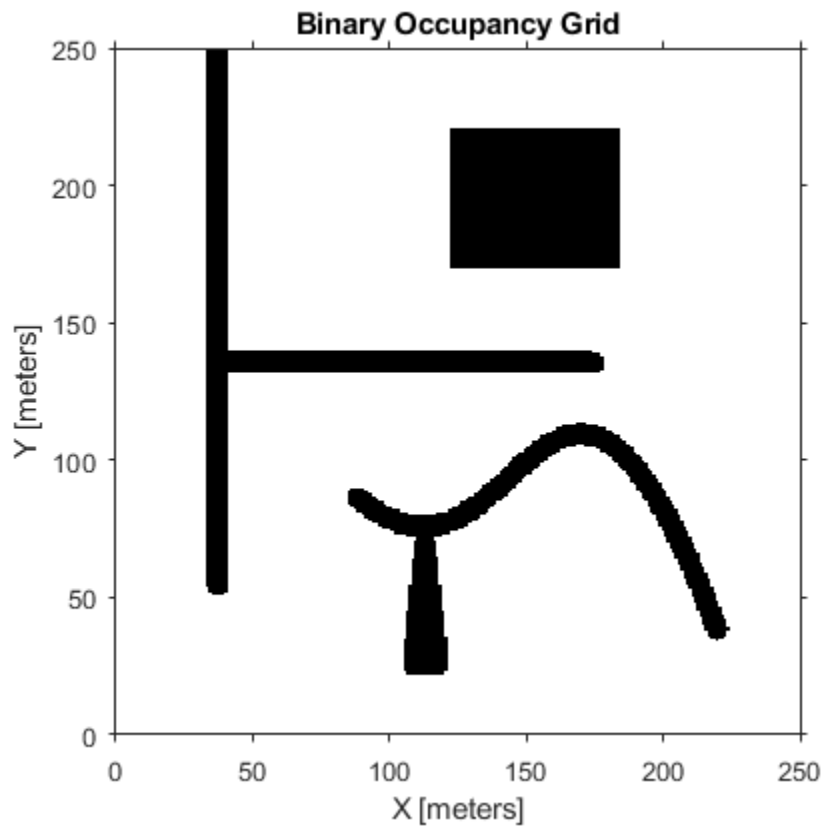
## Image to Binary Occupancy Grid Example

This example shows how to convert an image to a binary occupancy grid for using with the Robotics System Toolbox®

```
% Import Image
filepath = fullfile(matlabroot, 'examples', 'robotics', 'imageMap.png');
image = imread(filepath);

% Convert to grayscale and then black and white image based on arbitrary
% threshold.
```

```
grayimage = rgb2gray(image);  
bwimage = grayimage < 0.5;  
  
% Use black and white image as matrix input for binary occupancy grid  
grid = robotics.BinaryOccupancyGrid(bwimage);  
  
show(grid)
```

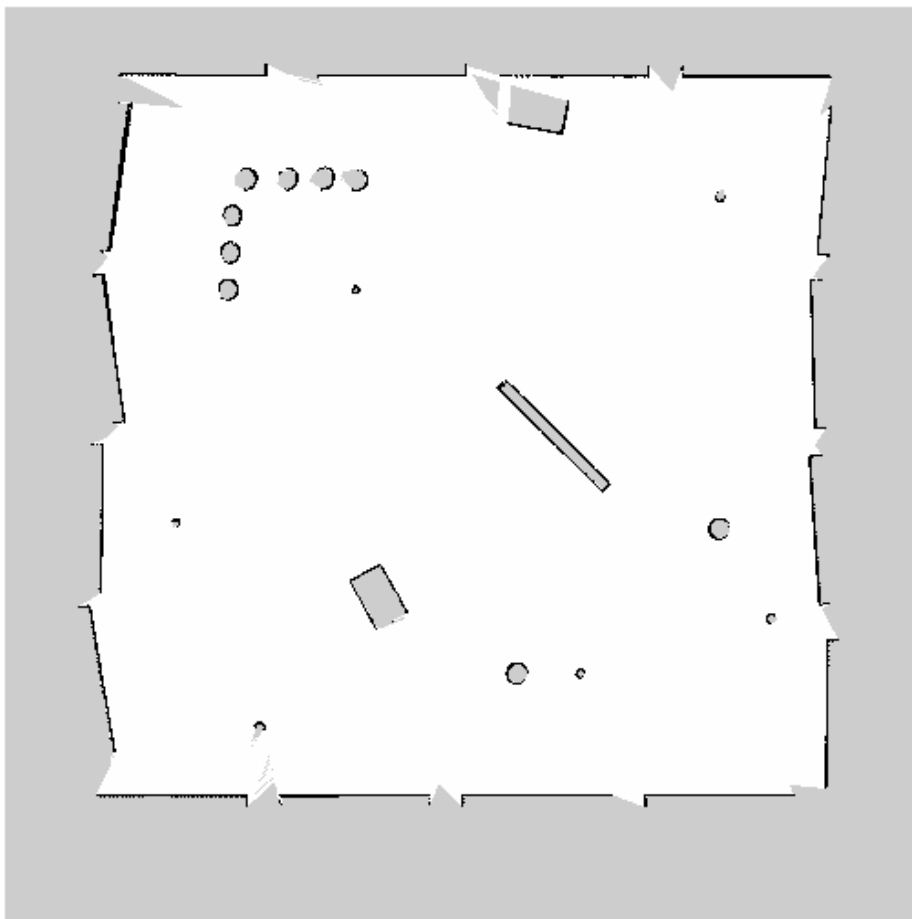


## Convert PGM Image to Map

This example shows how to convert a .pgm file which contains a ROS map into a BinaryOccupancyGrid map for use in MATLAB.

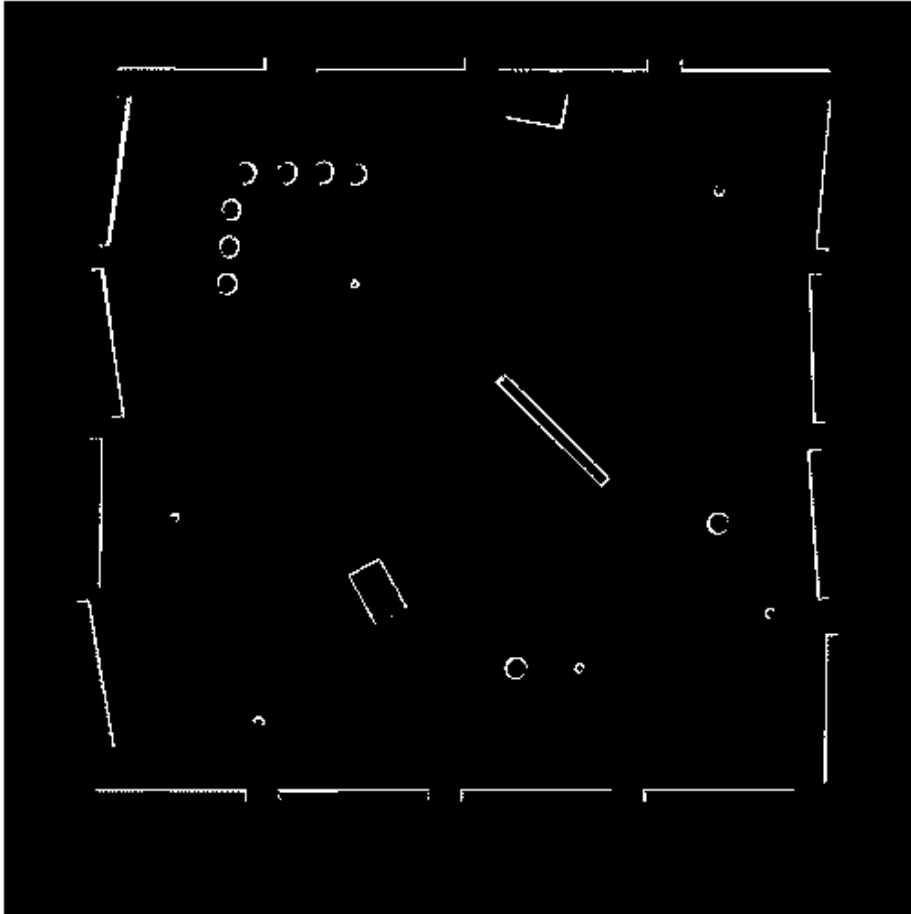
Import image using `imread`. The image is quite large and should be cropped to the relevant area.

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



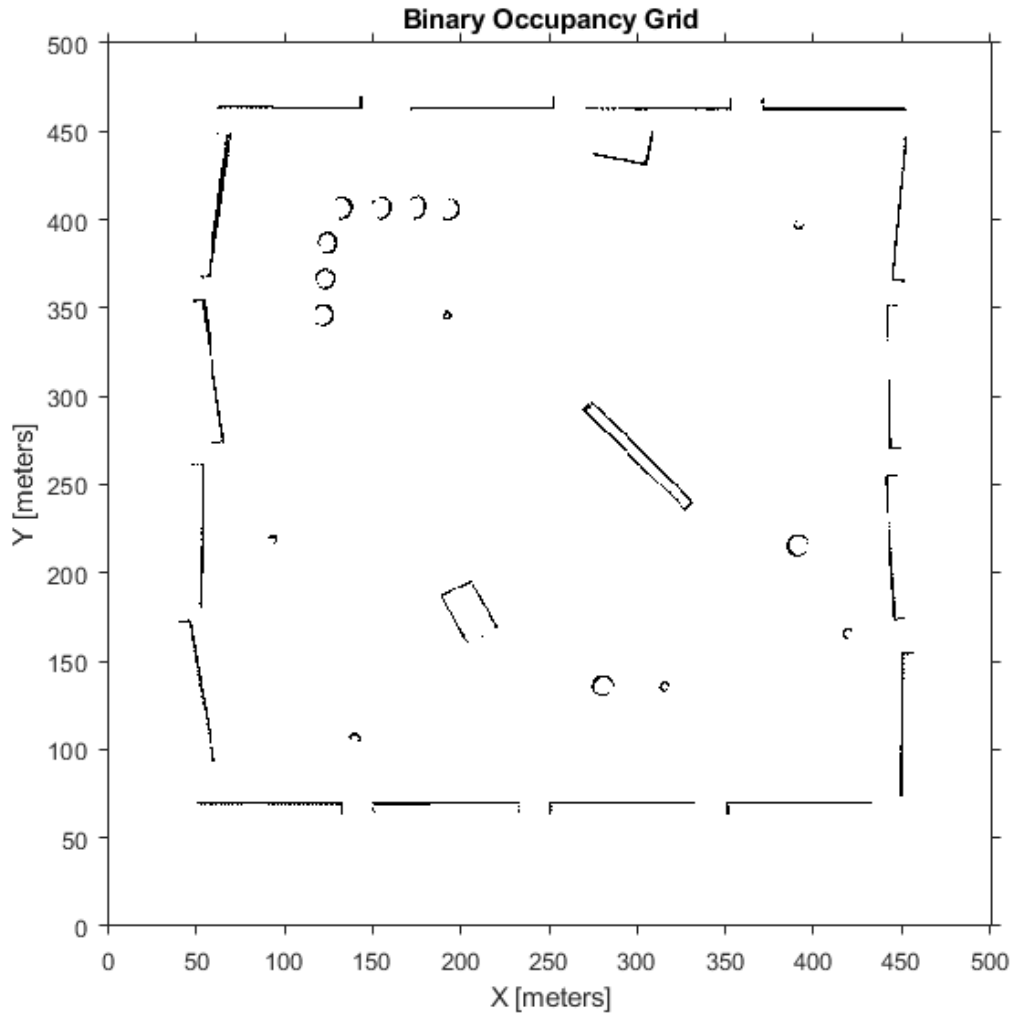
Unknown areas (gray) should be removed and treated as free space. Create a logical matrix based on a threshold. Depending on your image, this value could be different. Occupied space should be set as 1 (white in image).

```
imageBW = imageCropped < 100;  
imshow(imageBW)
```



Create BinaryOccupancyGrid object using adjusted map image.

```
map = robotics.BinaryOccupancyGrid(imageBW);  
show(map)
```



## **Extended Capabilities**

### **C/C++ Code Generation**

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

### **See Also**

`robotics.PRM` | `robotics.PurePursuit`

### **Topics**

“Occupancy Grids”

**Introduced in R2015a**



# robotics.CartesianBounds class

**Package:** robotics

Create constraint to keep body origin inside Cartesian bounds

## Description

The `CartesianBounds` object describes a constraint on the position of one body (the end effector) relative to a target frame fixed on another body (the reference body). This constraint is satisfied if the position of the end-effector origin relative to the target frame remains within the `Bounds` specified. The `TargetTransform` property is the homogeneous transform that converts points in the target frame to points in the `ReferenceBody` frame.

Constraint objects are used in `GeneralizedInverseKinematics` objects to specify multiple kinematic constraints on a robot.

For an example that uses multiple constraint objects, see “Plan a Reaching Trajectory With Multiple Kinematic Constraints”.

## Construction

`cartConst = robotics.CartesianBounds(endeffector)` returns a Cartesian bounds object that represents a constraint on the body of the robot model specified by `endeffector`.

`cartConst = robotics.CartesianBounds(endeffector, Name, Value)` returns a Cartesian bounds object with each specified property name set to the specified value by one or more `Name, Value` pair arguments.

## Input Arguments

**endeffector** — End-effector name

string scalar | character vector

End-effector name, specified as a string scalar or character vector. When using this constraint with `GeneralizedInverseKinematics`, the name must match a body specified in the robot model (`RigidBodyTree`).

Example: `"left_palm"`

Data Types: `char` | `string`

## Properties

### **EndEffector — Name of the end effector**

`string` scalar | character vector

Name of the end effector, specified as a string scalar or character vector. When using this constraint with `GeneralizedInverseKinematics`, the name must match a body specified in the robot model (`RigidBodyTree`).

Example: `"left_palm"`

Data Types: `char` | `string`

### **ReferenceBody — Name of the reference body frame**

`' '` (default) | `string` scalar | character vector

Name of the reference body frame, specified as a string scalar or character vector. The default `' '` indicates that the constraint is relative to the base of the robot model. When using this constraint with `GeneralizedInverseKinematics`, the name must match a body specified in the robot model (`RigidBodyTree`).

### **TargetTransform — Pose of the target frame relative to the reference body**

`eye(4)` (default) | `matrix`

Pose of the target frame relative to the reference body, specified as a matrix. The matrix is a homogeneous transform that specifies the relative transformation to convert a point in the target frame to the reference body frame.

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

### **Bounds — Bounds on end-effector position relative to target frame**

`zeros(3,2)` (default) | `[xMin xMax; yMin yMax; zMin zMax]` vector

Bounds on end-effector position relative to target frame, specified as a 3-by-2 vector, [xMin xMax; yMin yMax; zMin zMax]. Each row defines the minimum and maximum values for the xyz-coordinates respectively.

**Weights — Weights of the constraint**

[1 1 1] (default) | [x y z] vector

Weights of the constraint, specified as an [x y z] vector. Each element of the vector corresponds to the weight for the xyz-coordinates, respectively. These weights are used with the `Weights` property of all the constraints specified in `GeneralizedInverseKinematics` to properly balance each constraint.

## Extended Capabilities

### C/C++ Code Generation

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

## See Also

**Classes**

`robotics.GeneralizedInverseKinematics` | `robotics.OrientationTarget` | `robotics.PoseTarget` | `robotics.PositionTarget`

**Topics**

“Plan a Reaching Trajectory With Multiple Kinematic Constraints”

**Introduced in R2017a**

## robotics.DubinsConnection

Dubins path connection type

### Description

The `DubinsConnection` object holds information for computing a `DubinsPathSegment` path segment to connect between poses. A Dubins path segment connects two poses as a sequence of three motions. The motion options are:

- Straight
- Left turn at maximum steer
- Right turn at maximum steer

A Dubins path segment only allows motion in the forward direction.

Use this connection object to define parameters for a robot motion model, including the minimum turning radius and options for path types. To generate a path segment between poses using this connection type, call the `connect` function.

### Creation

### Syntax

```
dubConnObj = robotics.DubinsConnection  
dubConnObj = robotics.DubinsConnection(Name,Value)
```

### Description

`dubConnObj = robotics.DubinsConnection` creates an object using default property values.

`dubConnObj = robotics.DubinsConnection(Name,Value)` specifies property values using name-value pairs. To set multiple properties, specify multiple name-value pairs.

## Properties

### MinTurningRadius — Minimum turning radius

1 (default) | positive scalar in meters

Minimum turning radius for the robot, specified as a positive scalar in meters. The minimum turning radius is for the smallest circle the robot can make with maximum steer in a single direction.

Data Types: double

### DisabledPathTypes — Path types to disable

{ } (default) | cell array of three-element character vectors | vector of three-element string scalars

Dubins path types to disable, specified as a cell array of three-element character vectors or vector of string scalars. The cell array defines three sequences of motions that are prohibited by the robot motion model.

Valid values are:

Motion Type	Description
"S"	Straight
"L"	Left turn at the maximum steering angle of the vehicle
"R"	Right turn at the maximum steering angle of the vehicle

To see all available path types, see the AllPathTypes property.

Example: ["LSL", "LSR"]

Data Types: string | cell

### AllPathTypes — All possible path types

cell array of character vectors

This property is read-only.

All possible path types, returned as a cell array of character vectors. This property lists all types. To disable certain types, specify types from this list in DisabledPathTypes.

For Dubins connections, the available path types are: {'LSL'} {'LSR'} {'RSL'} {'RSR'} {'RLR'} {'LRL'}.

Data Types: cell

## Object Functions

`connect` Connect poses for given connection type

## Examples

### Connect Poses Using Dubins Connection Path

Create a DubinsConnection object.

```
dubConnObj = robotics.DubinsConnection;
```

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

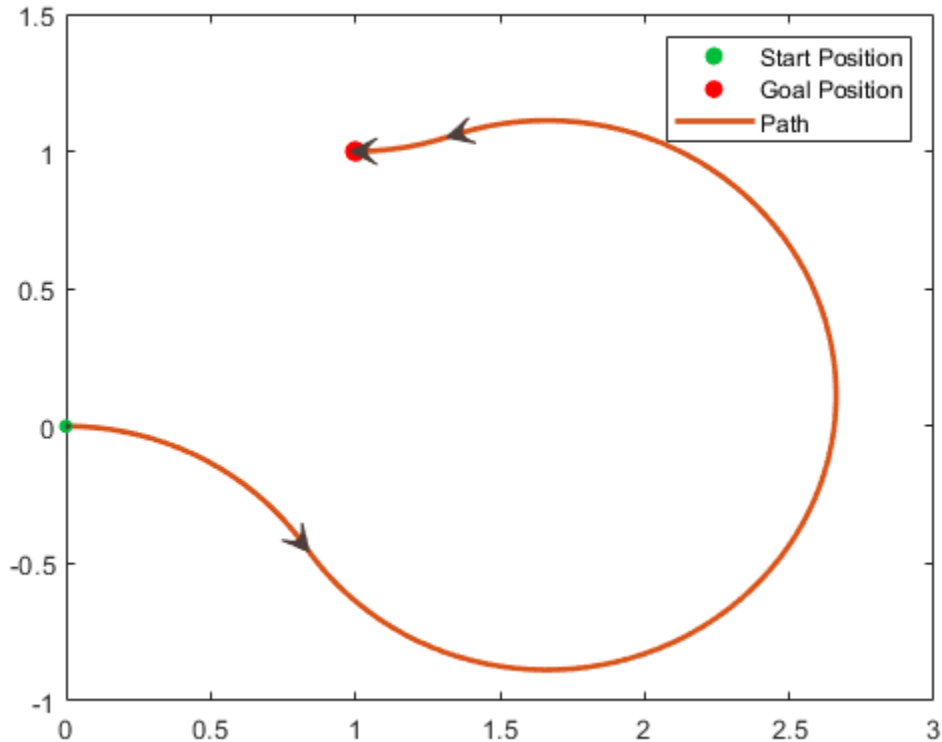
```
startPose = [0 0 0];  
goalPose = [1 1 pi];
```

Calculate a valid path segment to connect the poses.

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

Show the generated path.

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



### Modify Connection Types for Dubins Path

Create a DubinsConnection object.

```
dubConnObj = robotics.DubinsConnection;
```

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

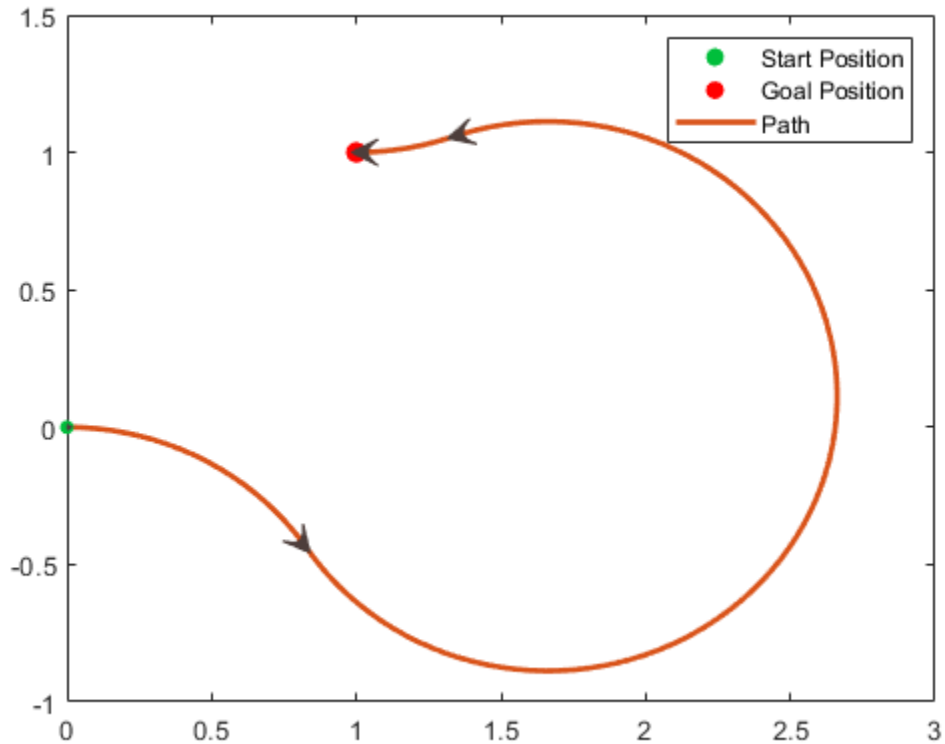
```
startPose = [0 0 0];  
goalPose = [1 1 pi];
```

Calculate a valid path segment to connect the poses.

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

Show the generated path. Notice the direction of the turns.

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



```
pathSegObj{1}.MotionTypes
```

```
ans = 1x3 cell array  
    {'R'}    {'L'}    {'R'}
```

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

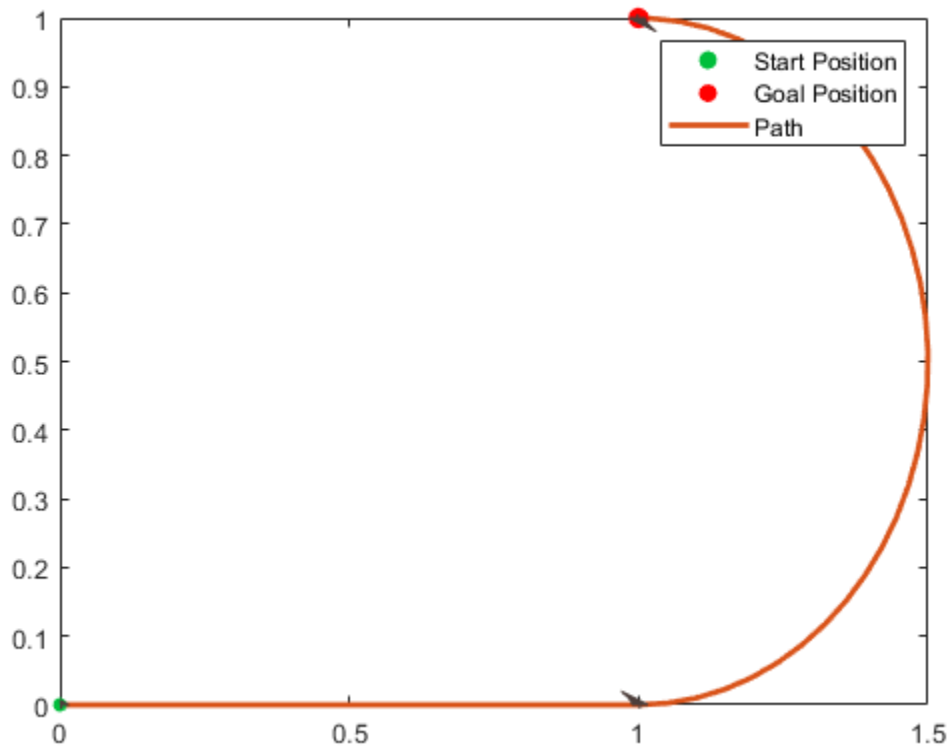


```
dubConnObj = robotics.DubinsConnection('DisabledPathTypes',{ 'RLR' });
dubConnObj.MinTurningRadius = 0.5;

[pathSegObj, pathCosts] = connect(dubConnObj, startPose, goalPose);
pathSegObj{1}.MotionTypes

ans = 1x3 cell array
    {'L'}    {'S'}    {'L'}

show(pathSegObj{1})
```



## References

- [1] Shkel, Andrei M., and Vladimir Lumelsky. "Classification of the Dubins set." *Robotics and Autonomous Systems*. Vol. 34, No. 4, 2001, pp. 179–202.

## Extended Capabilities

### C/C++ Code Generation

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

## See Also

### Objects

`robotics.DubinsPathSegment` | `robotics.ReedsSheppConnection` |  
`robotics.ReedsSheppPathSegment`

### Functions

`connect` | `interpolate` | `show`

**Introduced in R2018b**

# robotics.DubinsPathSegment

Dubins path segment connecting two poses

## Description

The `DubinsPathSegment` object holds information for a Dubins path segment that connects two poses as a sequence of three motions. The motion options are:

- Straight
- Left turn at maximum steer
- Right turn at maximum steer

## Creation

To generate a `DubinsPathSegment` object, use the `connect` function with a `robotics.DubinsConnection` object:

`dubPathSegObj = connect(connectionObj, start, goal)` connects the start and goal pose using the specified connection type object.

To specifically define a path segment:

`dubPathSegObj = robotics.DubinsPathSegment(connectionObj, start, goal, motionLengths, motionTypes)` specifies the Dubins connection type, the start and goal poses, and the corresponding motion lengths and types. These values are set to the corresponding properties in the object.

## Properties

### **MinTurningRadius** — Minimum turning radius of vehicle

positive scalar

This property is read-only.

Minimum turning radius of the robot, specified as a positive scalar in meters. This value corresponds to the radius of the turning circle at the maximum steering angle of the robot.

Data Types: double

### **StartPose — Initial pose of robot**

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

This property is read-only.

Initial pose of the robot at the start of the path segment, specified as an [ $x$ ,  $y$ ,  $\theta$ ] vector.  $x$  and  $y$  are in meters.  $\theta$  is in radians.

Data Types: double

### **GoalPose — Goal pose of robot**

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

This property is read-only.

Goal pose of the robot at the end of the path segment, specified as an [ $x$ ,  $y$ ,  $\theta$ ] vector.  $x$  and  $y$  are in meters.  $\theta$  is in radians.

Data Types: double

### **MotionLengths — Length of each motion**

three-element numeric vector

This property is read-only.

Length of each motion in the path segment, in world units, specified as a three-element numeric vector. Each motion length corresponds to a motion type specified in `MotionTypes`.

Data Types: double

### **MotionTypes — Type of each motion**

three-element string cell array

This property is read-only.

Type of each motion in the path segment, specified as a three-element string cell array. Valid values are:

Motion Type	Description
"S"	Straight
"L"	Left turn at the maximum steering angle of the vehicle
"R"	Right turn at the maximum steering angle of the vehicle

Each motion type corresponds to a motion length specified in `MotionLengths`.

Example: {"R" "S" "R"}

Data Types: cell

### Length — Length of path segment

positive scalar

This property is read-only.

Length of the path segment, specified as a positive scalar in meters. This length is just a sum of the elements in `MotionLengths`.

Data Types: double

## Object Functions

interpolate Interpolate poses along path segment

show Visualize path segment

## Examples

### Connect Poses Using Dubins Connection Path

Create a `DubinsConnection` object.

```
dubConnObj = robotics.DubinsConnection;
```

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

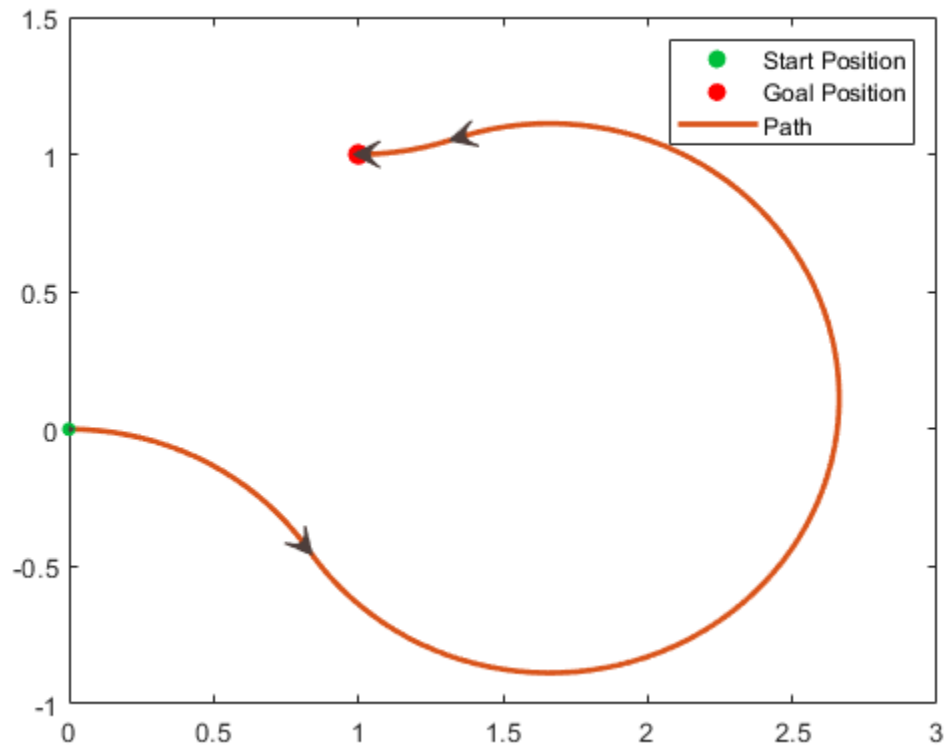
```
startPose = [0 0 0];
goalPose = [1 1 pi];
```

Calculate a valid path segment to connect the poses.

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

Show the generated path.

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



## Extended Capabilities

### C/C++ Code Generation

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

### See Also

#### Objects

`robotics.DubinsConnection` | `robotics.ReedsSheppConnection` |  
`robotics.ReedsSheppPathSegment`

#### Functions

`connect` | `interpolate` | `show`

**Introduced in R2018b**

## 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.

---

**Note** This object requires you to install the UAV Library for Robotics System Toolbox. To install add-ons, use `roboticsAddons` and select the desired add-on.

---

For multicopter UAVs, see `multicopter`.

### 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.

**Note:** To use UAV algorithms, you must install the UAV Library for Robotics System Toolbox®. To install, use `roboticsAddons`.

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; % 10 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.

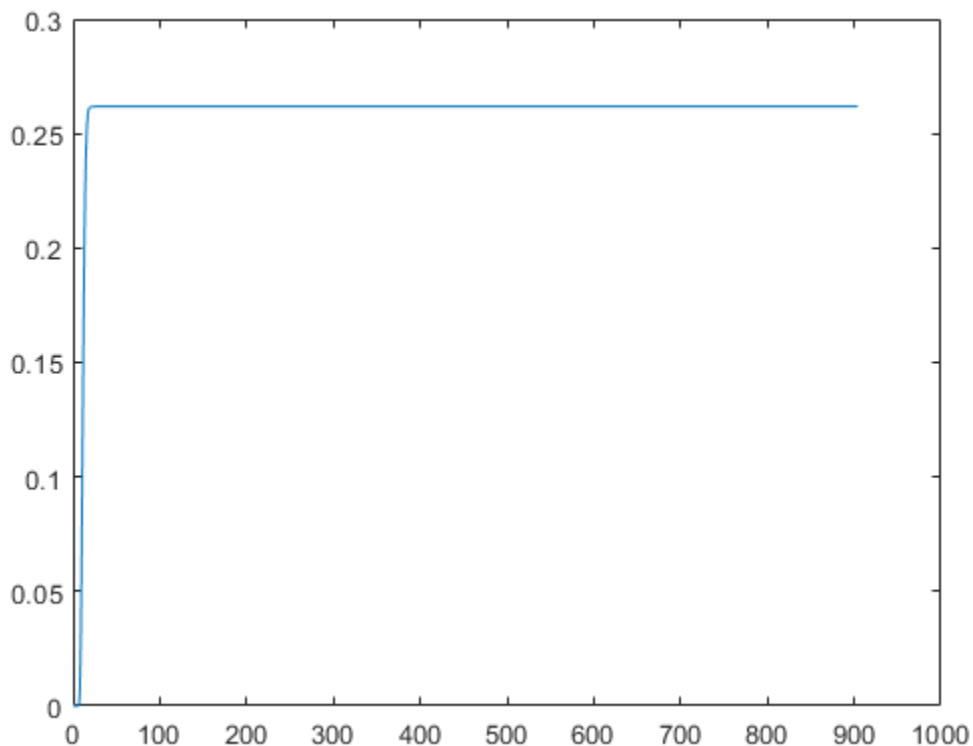
```
simOut = ode45(@(~,x)derivative(model,x,u,e), [0 50], s);  
size(simOut.y)
```

```
ans = 1x2
```

8 904

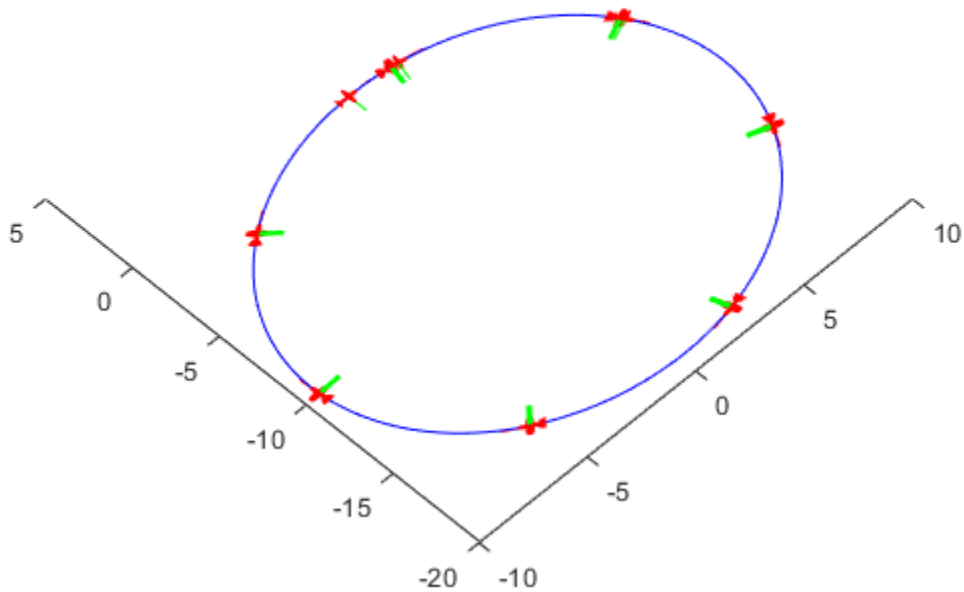
Plot the change in roll angle based on the simulation output. The roll angle is the 7th row of the `simOut.y` output.

```
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)']
    '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
```



## Definitions

### UAV Coordinate Systems

The UAV Library for Robotics System 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,  $\psi$ . Then, rotate about the intermediate  $y$ -axis by the pitch angle,  $\phi$ . Then, rotate about the intermediate  $x$ -axis by the roll angle,  $\theta$ .

The angular velocity of the UAV is represented by  $[r, p, q]$  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 heading 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` | `roboticsAddons` | `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

## **robotics.GeneralizedInverseKinematics**

**Package:** robotics

Create multiconstraint inverse kinematics solver

### **Description**

The `GeneralizedInverseKinematics` System object™ uses a set of kinematic constraints to compute a joint configuration for the rigid body tree model specified by a `RigidBodyTree` object. The `GeneralizedInverseKinematics` object uses a nonlinear solver to satisfy the constraints or reach the best approximation.

Specify the constraint types, `ConstraintInputs`, before calling the object. To change constraint inputs after calling the object, call `release(gik)`.

Specify the constraint inputs as constraint objects and call `GeneralizedInverseKinematics` with these objects passed into it. To create constraint objects, use these classes:

- `AimingConstraint`
- `CartesianBounds`
- `JointPositionBounds`
- `OrientationTarget`
- `PoseTarget`
- `PositionTarget`

If your only constraint is the end-effector position and orientation, consider using `InverseKinematics` as your solver instead.

To solve the generalized inverse kinematics constraints:

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

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



## Creation

### Syntax

```
gik = robotics.GeneralizedInverseKinematics
gik = robotics.GeneralizedInverseKinematics('
RigidBodyTree',rigidbodytree,'ConstraintInputs',inputTypes)
gik = robotics.GeneralizedInverseKinematics(Name,Value)
```

### Description

`gik = robotics.GeneralizedInverseKinematics` returns a generalized inverse kinematics solver with no rigid body tree model specified. Specify a `RigidBodyTree` model and the `ConstraintInputs` property before using this solver.

`gik = robotics.GeneralizedInverseKinematics('RigidBodyTree',rigidbodytree,'ConstraintInputs',inputTypes)` returns a generalized inverse kinematics solver with the rigid body tree model and the expected constraint inputs specified.

`gik = robotics.GeneralizedInverseKinematics(Name,Value)` returns a generalized inverse kinematics solver with each specified property name set to the specified value by one or more `Name,Value` pair arguments. `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 \(MATLAB\)](#).

## **NumConstraints — Number of constraint inputs**

scalar

This property is read-only.

Number of constraint inputs, specified as a scalar. The value of this property is the number of constraint types specified in the `ConstraintInputs` property.

## **ConstraintInputs — Constraint input types**

cell array of character vectors

Constraint input types, specified as a cell array of character vectors. The possible constraint input types with their associated constraint objects are:

- 'orientation' — `OrientationTarget`
- 'position' — `PositionTarget`
- 'pose' — `PoseTarget`
- 'aiming' — `AimingConstraint`
- 'cartesian' — `CartesianBounds`
- 'joint' — `JointPositionBounds`

Use the constraint objects to specify the required parameters and pass those object types into the object when you call it. For example:

Create the generalized inverse kinematics solver object. Specify the `RigidBodyTree` and `ConstraintInputs` properties.

```
gik = robotics.GeneralizedInverseKinematics(...  
    'RigidBodyTree',rigidbodytree,  
    'ConstraintInputs',{'position','aiming'});
```

Create the corresponding constraint objects.

```
positionTgt = robotics.PositionTarget('left_palm');  
aimConst = robotics.AimingConstraint('right_palm');
```

Pass the constraint objects into the solver object with an initial guess.

```
configSol = gik(initialGuess,positionTgt,aimConst);
```

## **RigidBodyTree — Rigid body tree model**

`RigidBodyTree` object

Rigid body tree model, specified as a `RigidBodyTree` object. Define this property before using the solver. If you modify your rigid body tree model, reassign the rigid body tree to this property. For example:

Create IK solver and specify the rigid body tree.

```
gik = robotics.GeneralizedInverseKinematics(...
    'RigidBodyTree',rigidbodytree,
    'ConstraintInputs',{ 'position', 'aiming' });
```

Modify the rigid body tree model.

```
addBody(rigidbodytree,robotics.RigidBody('body1'), 'base')
```

Reassign the rigid body tree to the IK solver. If the solver or the `step` function is called before modifying the rigid body tree model, use `release` to allow the property to be changed.

```
gik.RigidBodyTree = rigidbodytree;
```

### **SolverAlgorithm — Algorithm for solving inverse kinematics**

'BFGSGradientProjection' (default) | 'LevenbergMarquardt'

Algorithm for solving inverse kinematics, specified as either 'BFGSGradientProjection' or 'LevenbergMarquardt'. For details of each algorithm, see “Inverse Kinematics Algorithms”.

### **SolverParameters — Parameters associated with algorithm**

structure

Parameters associated with the specified algorithm, specified as a structure. The fields in the structure are specific to the algorithm. See “Solver Parameters”.

## **Usage**

## **Syntax**

```
[configSol,solInfo] = gik(initialguess,
constraintObj,...,constraintObjN)
```

## Description

`[configSol,solInfo] = gik(initialguess, constraintObj,...,constraintObjN)` finds a joint configuration, `configSol`, based on the initial guess and a comma-separated list of constraint description objects. The number of constraint descriptions depends on the `ConstraintInputs` property.

## Input Arguments

### **initialguess** — Initial guess of robot configuration

structure array | vector

Initial guess of robot configuration, specified as a structure array or vector. The value of `initialguess` depends on the `DataFormat` property of the object specified in the `RigidBodyTree` property specified in `gik`.

Use this initial guess to guide the solver to the target robot configuration. However, the solution is not guaranteed to be close to this initial guess.

### **constraintObj,...,constraintObjN** — Constraint descriptions

constraint objects

Constraint descriptions defined by the `ConstraintInputs` property of `gik`, specified as one or more of these constraint objects:

- `AimingConstraint`
- `CartesianBounds`
- `JointPositionBounds`
- `OrientationTarget`
- `PoseTarget`
- `PositionTarget`

## Output Arguments

### **configSol** — Robot configuration solution

structure array | vector

Robot configuration solution, returned as a structure array or vector. depends on the `DataFormat` property of the object specified in the `RigidBodyTree` property specified in `gik`.

The structure array contains these fields:

- `JointName` — Character vector for the name of the joint specified in the `RigidBodyTree` robot model
- `JointPosition` — Position of the corresponding joint

The vector output is an array of the joint positions that would be given in `JointPosition` for a structure output.

This joint configuration is the computed solution that achieves the target end-effector pose within the solution tolerance.

### **solInfo — Solution information**

structure

Solution information, returned as a structure containing these fields:

- `Iterations` — Number of iterations run by the solver.
- `NumRandomRestarts` — Number of random restarts because the solver got stuck in a local minimum.
- `ConstraintViolation` — Information about the constraint, returned as a structure array. Each structure in the array has these fields:
  - `Type`: Type of the corresponding constraint input, as specified in the `ConstraintInputs` property.
  - `Violation`: Vector of constraint violations for the corresponding constraint type. 0 indicates that the constraint is satisfied.
- `ExitFlag` — Code that gives more details on the solver execution and what caused it to return. For the exit flags of each solver type, see “Exit Flags”.
- `Status` — Character vector describing whether the solution is within the tolerances defined by each constraint ('success'). If the solution is outside the tolerance, the best possible solution that the solver could find is given ('best available').

## 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

### Solve Generalized Inverse Kinematics for a Set of Constraints

Create a generalized inverse kinematics solver that holds a robotic arm at a specific location and points toward the robot base. Create the constraint objects to pass the necessary constraint parameters into the solver.

Load predefined KUKA LBR robot model, which is specified as a `RigidBodyTree` object.

```
load exampleRobots.mat lbr
```

Create the System object™ for solving generalized inverse kinematics.

```
gik = robotics.GeneralizedInverseKinematics;
```

Configure the System object to use the KUKA LBR robot.

```
gik.RigidBodyTree = lbr;
```

Tell the solver to expect a `PositionTarget` object and an `AimingConstraint` object as the constraint inputs.

```
gik.ConstraintInputs = {'position', 'aiming'};
```

Create the two constraint objects.

- 1 The origin of the body named `tool0` is located at `[0.0 0.5 0.5]` relative to the robot's base frame.
- 2 The  $z$ -axis of the body named `tool0` points toward the origin of the robot's base frame.

```
posTgt = robotics.PositionTarget('tool0');  
posTgt.TargetPosition = [0.0 0.5 0.5];
```

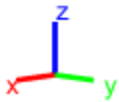
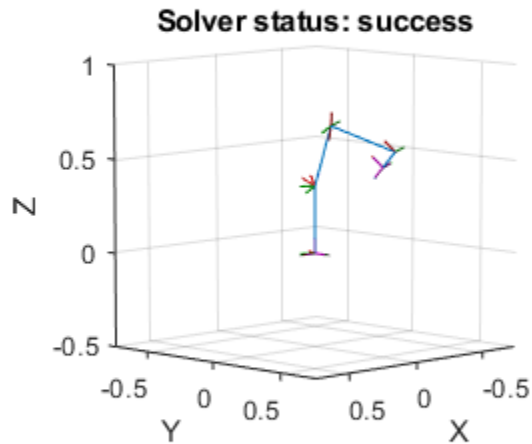
```
aimCon = robotics.AimingConstraint('tool0');  
aimCon.TargetPoint = [0.0 0.0 0.0];
```

Find a configuration that satisfies the constraints. You must pass the constraint objects into the `System` object in the order in which they were specified in the `ConstraintInputs` property. Specify an initial guess at the robot configuration.

```
q0 = homeConfiguration(lbr); % Initial guess for solver  
[q,solutionInfo] = gik(q0,posTgt,aimCon);
```

Visualize the configuration returned by the solver.

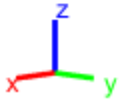
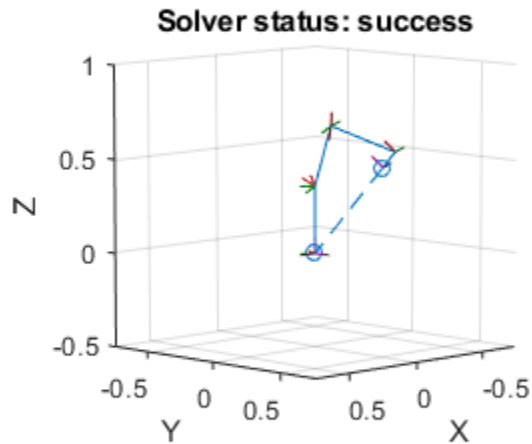
```
show(lbr,q);  
title(['Solver status: ' solutionInfo.Status])  
axis([-0.75 0.75 -0.75 0.75 -0.5 1])
```



Plot a line segment from the target position to the origin of the base. The origin of the `tool0` frame coincides with one end of the segment, and its z-axis is aligned with the segment.

```
hold on
plot3([0.0 0.0],[0.5 0.0],[0.5 0.0], '--o')
hold off
```





## Extended Capabilities

### C/C++ Code Generation

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

When using code generation, you must specify the `ConstraintInputs` and `RigidBodyTree` properties on construction of the object. For example:

```
gik = robotics.GeneralizedInverseKinematics(...  
    'ConstraintInputs', {'pose', 'position'}, ...  
    'RigidBodyTree', rigidbodytree);
```

You also cannot change the SolverAlgorithm property after creation. To specify the solver algorithm on creation, use:

```
gik = robotics.GeneralizedInverseKinematics(...  
    'ConstraintInputs', {'pose', 'position'}, ...  
    'RigidBodyTree', rigidbodytree, ...  
    'SolverAlgorithm', 'LevenbergMarquardt');
```

## See Also

### Classes

robotics.AimingConstraint | robotics.CartesianBounds |  
robotics.InverseKinematics | robotics.JointPositionBounds |  
robotics.OrientationTarget | robotics.PoseTarget |  
robotics.PositionTarget

### Topics

“Control PR2 Arm Movements Using ROS Actions and Inverse Kinematics”  
“Plan a Reaching Trajectory With Multiple Kinematic Constraints”

### Introduced in R2017a

# robotics.InverseKinematics

**Package:** robotics

Create inverse kinematic solver

## Description

The `robotics.InverseKinematics` System object creates an inverse kinematic (IK) solver to calculate joint configurations for a desired end-effector pose based on a specified rigid body tree model. Create a rigid body tree model for your robot using the `robotics.RigidBodyTree` class. This model defines all the joint constraints that the solver enforces. If a solution is possible, the joint limits specified in the robot model are obeyed.

To specify more constraints besides the end-effector pose, including aiming constraints, position bounds, or orientation targets, consider using `robotics.GeneralizedInverseKinematics`. This class allows you to compute multiconstraint IK solutions.

To compute joint configurations for a desired end-effector pose:

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

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

## Creation

## Syntax

```
ik = robotics.InverseKinematics  
ik = robotics.InverseKinematics(Name,Value)
```

## Description

`ik = robotics.InverseKinematics` creates an inverse kinematic solver. To use the solver, specify a rigid body tree model in the `RigidBodyTree` property.

`ik = robotics.InverseKinematics(Name, Value)` creates an inverse kinematic solver 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 \(MATLAB\)](#).

### **RigidBodyTree — Rigid body tree model**

`RigidBodyTree` object

Rigid body tree model, specified as a `RigidBodyTree` object. If you modify your rigid body tree model, reassign the rigid body tree to this property. For example:

Create IK solver and specify the rigid body tree.

```
ik = robotics.InverseKinematics('RigidBodyTree', rigidbodytree)
```

Modify the rigid body tree model.

```
addBody(rigidbodytree, robotics.RigidBody('body1'), 'base')
```

Reassign the rigid body tree to the IK solver. If the solver or the `step` function is called before modifying the rigid body tree model, use `release` to allow the property to be changed.

```
ik.RigidBodyTree = rigidbodytree;
```

**SolverAlgorithm — Algorithm for solving inverse kinematics**`'BFGSGradientProjection'` (default) | `'LevenbergMarquardt'`

Algorithm for solving inverse kinematics, specified as either `'BFGSGradientProjection'` or `'LevenbergMarquardt'`. For details of each algorithm, see “Inverse Kinematics Algorithms”.

**SolverParameters — Parameters associated with algorithm**

structure

Parameters associated with the specified algorithm, specified as a structure. The fields in the structure are specific to the algorithm. See “Solver Parameters”.

## Usage

## Syntax

```
[configSol,solInfo] = ik(endeffector,pose,weights,initialguess)
```

## Description

`[configSol,solInfo] = ik(endeffector,pose,weights,initialguess)` finds a joint configuration that achieves the specified end-effector pose. Specify an initial guess for the configuration and your desired weights on the tolerances for the six components of pose. Solution information related to execution of the algorithm, `solInfo`, is returned with the joint configuration solution, `configSol`.

## Input Arguments

**endeffector — End-effector name**

character vector

End-effector name, specified as a character vector. The end effector must be a body on the `RigidBodyTree` object specified in the `robotics.InverseKinematics` System object.

**pose — End-effector pose**

4-by-4 homogeneous transform

End-effector pose, specified as a 4-by-4 homogeneous transform. This transform defines the desired position and orientation of the rigid body specified in the `endeffector` property.

### **weights — Weight for pose tolerances**

six-element vector

Weight for pose tolerances, specified as a six-element vector. The first three elements correspond to the weights on the error in orientation for the desired pose. The last three elements correspond to the weights on the error in xyz position for the desired pose.

### **initialguess — Initial guess of robot configuration**

structure array | vector

Initial guess of robot configuration, specified as a structure array or vector. Use this initial guess to help guide the solver to a desired robot configuration. The solution is not guaranteed to be close to this initial guess.

To use the vector form, set the `DataFormat` property of the object assigned in the `RigidBodyTree` property to either `'row'` or `'column'`.

## **Output Arguments**

### **configSol — Robot configuration solution**

structure array | vector

Robot configuration, returned as a structure array. The structure array contains these fields:

- `JointName` — Character vector for the name of the joint specified in the `RigidBodyTree` robot model
- `JointPosition` — Position of the corresponding joint

This joint configuration is the computed solution that achieves the desired end-effector pose within the solution tolerance.

To use the vector form, set the `DataFormat` property of the object assigned in the `RigidBodyTree` property to either `'row'` or `'column'`.

### **solInfo — Solution information**

structure

Solution information, returned as a structure. The solution information structure contains these fields:

- **Iterations** — Number of iterations run by the algorithm.
- **NumRandomRestarts** — Number of random restarts because algorithm got stuck in a local minimum.
- **PoseErrorNorm** — The magnitude of the pose error for the solution compared to the desired end-effector pose.
- **ExitFlag** — Code that gives more details on the algorithm execution and what caused it to return. For the exit flags of each algorithm type, see “Exit Flags”.
- **Status** — Character vector describing whether the solution is within the tolerance ('success') or the best possible solution the algorithm could find ('best available').

## 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 Joint Positions to Achieve End-Effector Position

Generate joint positions for a robot model to achieve a desired end-effector position. The `InverseKinematics` system object uses inverse kinematic algorithms to solve for valid joint positions.

Load example robots. The `puma1` robot is a `RigidBodyTree` model of a six-axis robot arm with six revolute joints.

```
load exampleRobots.mat
showdetails(puma1)
```

```
-----
Robot: (6 bodies)
```

Idx	Body Name	Joint Name	Joint Type	Parent Name(Idx)	Children Name(s)
1	L1	jnt1	revolute	base(0)	L2(2)
2	L2	jnt2	revolute	L1(1)	L3(3)
3	L3	jnt3	revolute	L2(2)	L4(4)
4	L4	jnt4	revolute	L3(3)	L5(5)
5	L5	jnt5	revolute	L4(4)	L6(6)
6	L6	jnt6	revolute	L5(5)	

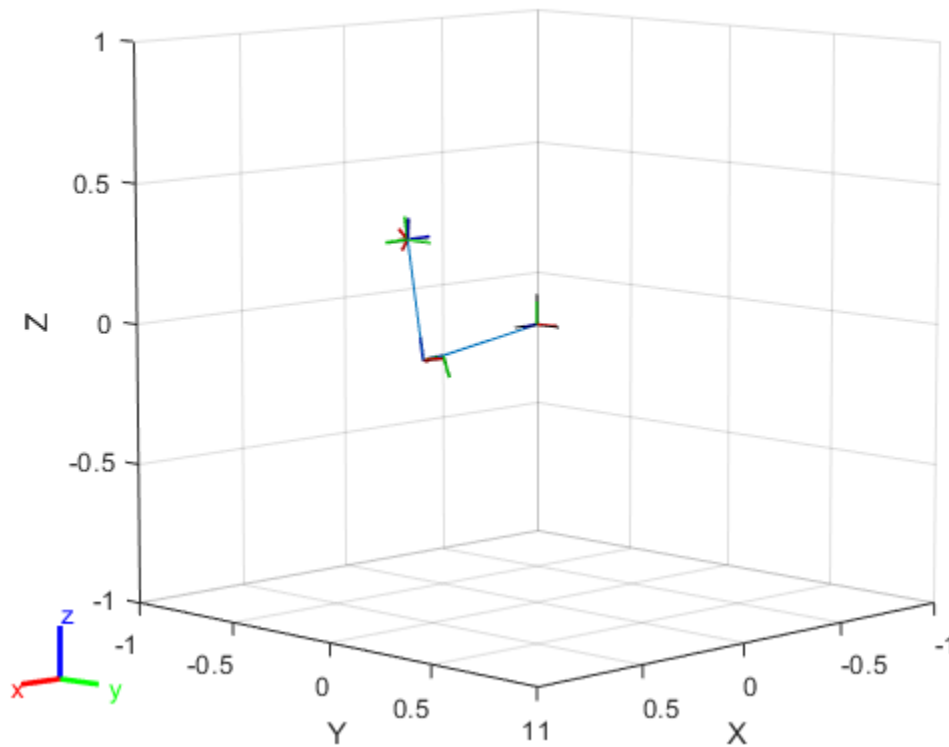
```
-----
```

Generate a random configuration. Get the transformation from the end effector (L6) to the base for that random configuration. Use this transform as a goal pose of the end effector. Show this configuration.

```
randConfig = puma1.randomConfiguration;
tform = getTransform(puma1,randConfig,'L6','base');

show(puma1,randConfig);
```





Create an `InverseKinematics` object for the `puma1` model. Specify weights for the different components of the pose. Use a lower magnitude weight for the orientation angles than the position components. Use the home configuration of the robot as an initial guess.

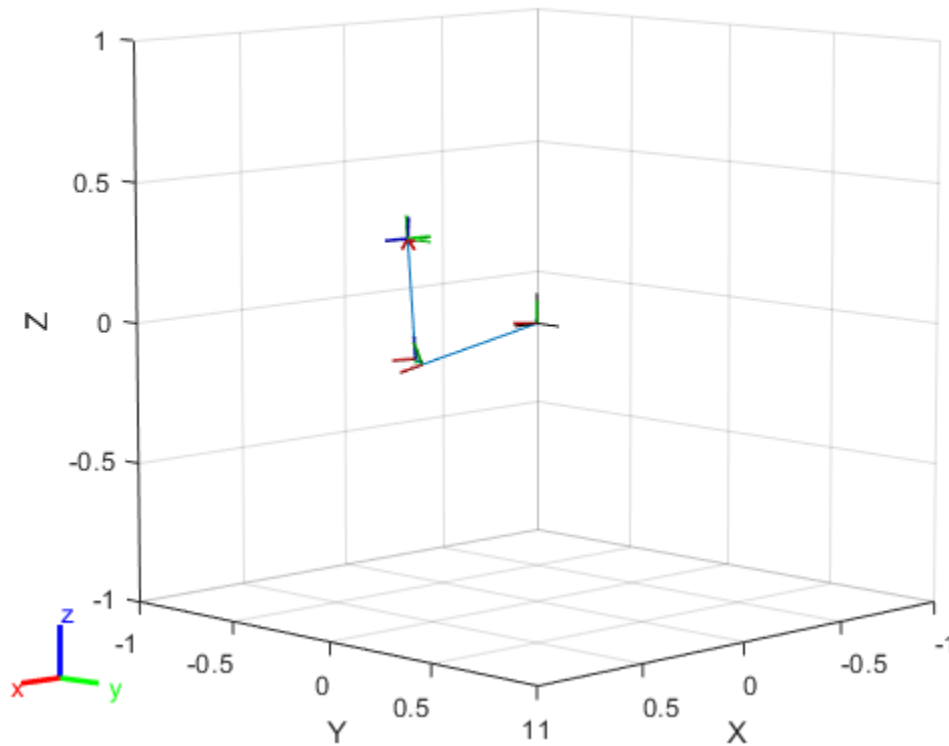
```
ik = robotics.InverseKinematics('RigidBodyTree',puma1);
weights = [0.25 0.25 0.25 1 1 1];
initialguess = puma1.homeConfiguration;
```

Calculate the joint positions using the `ik` object.

```
[configSoln,solnInfo] = ik('L6',tform,weights,initialguess);
```

Show the newly generated solution configuration. The solution is a slightly different joint configuration that achieves the same end-effector position. Multiple calls to the ik object can give similar or very different joint configurations.

```
show(puma1,configSoln);
```



## References

- [1] Badreddine, Hassan, Stefan Vandewalle, and Johan Meyers. "Sequential Quadratic Programming (SQP) for Optimal Control in Direct Numerical Simulation of Turbulent Flow." *Journal of Computational Physics*. 256 (2014): 1-16. doi:10.1016/j.jcp.2013.08.044.

- [2] Bertsekas, Dimitri P. *Nonlinear Programming*. Belmont, MA: Athena Scientific, 1999.
- [3] Goldfarb, Donald. "Extension of Davidon's Variable Metric Method to Maximization Under Linear Inequality and Equality Constraints." *SIAM Journal on Applied Mathematics*. Vol. 17, No. 4 (1969): 739-64. doi:10.1137/0117067.
- [4] Nocedal, Jorge, and Stephen Wright. *Numerical Optimization*. New York, NY: Springer, 2006.
- [5] Sugihara, Tomomichi. "Solvability-Unconcerned Inverse Kinematics by the Levenberg-Marquardt Method." *IEEE Transactions on Robotics* Vol. 27, No. 5 (2011): 984-91. doi:10.1109/tro.2011.2148230.
- [6] Zhao, Jianmin, and Norman I. Badler. "Inverse Kinematics Positioning Using Nonlinear Programming for Highly Articulated Figures." *ACM Transactions on Graphics* Vol. 13, No. 4 (1994): 313-36. doi:10.1145/195826.195827.

## Extended Capabilities

### C/C++ Code Generation

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

When using code generation, you must specify the `RigidBodyTree` property to define the robot on construction of the object. For example:

```
ik = robotics.InverseKinematics('RigidBodyTree',robotModel);
```

You also cannot change the `SolverAlgorithm` property after creation. To specify the solver algorithm on creation, use:

```
ik = robotics.InverseKinematics('RigidBodyTree',robotModel,...  
    'SolverAlgorithm','LevenbergMarquardt');
```

### See Also

`robotics.GeneralizedInverseKinematics` | `robotics.Joint` |  
`robotics.RigidBody` | `robotics.RigidBodyTree`

## **Topics**

“Control PR2 Arm Movements Using ROS Actions and Inverse Kinematics”

“Inverse Kinematics Algorithms”

Class Attributes (MATLAB)

Property Attributes (MATLAB)

**Introduced in R2016b**

# robotics.Joint class

**Package:** robotics

Create a joint

## Description

The `Joint` class creates a joint object that defines how a rigid body moves relative to an attachment point. In a tree-structured robot, a joint always belongs to a specific rigid body, and each rigid body has one joint.

The `Joint` object can describe joints of various types. When building a rigid body tree structure with `robotics.RigidBodyTree`, you must assign the `Joint` object to a rigid body using the `robotics.RigidBody` class.

The different joint types supported are:

- `fixed` — Fixed joint that prevents relative motion between two bodies.
- `revolute` — Single degree of freedom (DOF) joint that rotates around a given axis. Also called a pin or hinge joint.
- `prismatic` — Single DOF joint that slides along a given axis. Also called a sliding joint.

Each joint type has different properties with different dimensions, depending on its defined geometry.

## Construction

`jointObj = robotics.Joint(jname)` creates a fixed joint with the specified name.

`jointObj = robotics.Joint(jname, jtype)` creates a joint of the specified type with the specified name.

## Input Arguments

### **jname — Joint name**

string scalar | character vector

Joint name, specified as a string scalar or character vector. The joint name must be unique to access it off the rigid body tree.

Example: "elbow\_right"

Data Types: char | string

### **jtype — Joint type**

'fixed' (default) | string scalar | character vector

Joint type, specified as a string scalar or character vector. The joint type predefines certain properties when creating the joint.

The different joint types supported are:

- **fixed** — Fixed joint that prevents relative motion between two bodies.
- **revolute** — Single degree of freedom (DOF) joint that rotates around a given axis. Also called a pin or hinge joint.
- **prismatic** — Single DOF joint that slides along a given axis. Also called a sliding joint.

Data Types: char | string

## Properties

### **Type — Joint type**

'fixed' (default) | string scalar | character vector

This property is read-only.

Joint type, returned as a string scalar or character vector. The joint type predefines certain properties when creating the joint.

The different joint types supported are:

- `fixed` — Fixed joint that prevents relative motion between two bodies.
- `revolute` — Single degree of freedom (DOF) joint that rotates around a given axis. Also called a pin or hinge joint.
- `prismatic` — Single DOF joint that slides along a given axis. Also called a sliding joint.

If the rigid body that contains this joint is added to a robot model, the joint type must be changed by replacing the joint using `robotics.RigidBodyTree.replaceJoint`.

Data Types: `char` | `string`

### **Name — Joint name**

`string` scalar | character vector

Joint name, returned as a string scalar or character vector. The joint name must be unique to access it off the rigid body tree. If the rigid body that contains this joint is added to a robot model, the joint name must be changed by replacing the joint using `robotics.RigidBodyTree.replaceJoint`.

Example: "elbow\_right"

Data Types: `char` | `string`

### **PositionLimits — Position limits of joint**

vector

Position limits of the joint, specified as a vector of `[min max]` values. Depending on the type of joint, these values have different definitions.

- `fixed` — `[NaN NaN]` (default). A fixed joint has no joint limits. Bodies remain fixed between each other.
- `revolute` — `[-pi pi]` (default). The limits define the angle of rotation around the axis in radians.
- `prismatic` — `[0 0.5]` (default). The limits define the linear motion along the axis in meters.

### **HomePosition — Home position of joint**

scalar

Home position of joint, specified as a scalar that depends on your joint type. The home position must fall in the range set by `PositionLimits`. This property is used by

`robotics.RigidBodyTree.homeConfiguration` to generate the predefined home configuration for an entire rigid body tree.

Depending on the joint type, the home position has a different definition.

- `fixed` — 0 (default). A fixed joint has no relevant home position.
- `revolute` — 0 (default). A revolute joint has a home position defined by the angle of rotation around the joint axis in radians.
- `prismatic` — 0 (default). A prismatic joint has a home position defined by the linear motion along the joint axis in meters.

### **JointAxis — Axis of motion for joint**

`[NaN NaN NaN]` (default) | three-element unit vector

Axis of motion for joint, specified as a three-element unit vector. The vector can be any direction in 3-D space in local coordinates.

Depending on the joint type, the joint axis has a different definition.

- `fixed` — A fixed joint has no relevant axis of motion.
- `revolute` — A revolute joint rotates the body in the plane perpendicular to the joint axis.
- `prismatic` — A prismatic joint moves the body in a linear motion along the joint axis direction.

### **JointToParentTransform — Fixed transform from joint to parent frame**

`eye(4)` (default) | 4-by-4 homogeneous transform matrix

This property is read-only.

Fixed transform from joint to parent frame, returned as a 4-by-4 homogeneous transform matrix. The transform converts the coordinates of points in the joint predecessor frame to the parent body frame.

### **ChildToJointTransform — Fixed transform from child body to joint frame**

`eye(4)` (default) | 4-by-4 homogeneous transform matrix

This property is read-only.

Fixed transform from child body to joint frame, returned as a 4-by-4 homogeneous transform matrix. The transform converts the coordinates of points in the child body frame to the joint successor frame.



## Methods

copy	Create copy of joint
setFixedTransform	Set fixed transform properties of joint

## Examples

### Attach Rigid Body and Joint to Rigid Body Tree

Add a rigid body and corresponding joint to a rigid body tree. Each `RigidBody` object contains a `Joint` object and must be added to the `RigidBodyTree` using `addBody`.

Create a rigid body tree.

```
rbtree = robotics.RigidBodyTree;
```

Create a rigid body with a unique name.

```
body1 = robotics.RigidBody('b1');
```

Create a revolute joint. By default, the `RigidBody` object comes with a fixed joint. Replace the joint by assigning a new `Joint` object to the `body1.Joint` property.

```
jnt1 = robotics.Joint('jnt1','revolute');
body1.Joint = jnt1;
```

Add the rigid body to the tree. Specify the body name that you are attaching the rigid body to. Because this is the first body, use the base name of the tree.

```
basename = rbtree.BaseName;
addBody(rbtree,body1,basename)
```

Use `showdetails` on the tree to confirm the rigid body and joint were added properly.

```
showdetails(rbtree)
```

```
-----
Robot: (1 bodies)
```

Idx	Body Name	Joint Name	Joint Type	Parent Name(Idx)	Children Name(s)
-----	-----------	------------	------------	------------------	------------------

```
-----  
1          b1          jnt1          revolute          base(0)  
-----
```

### Build Manipulator Robot Using Denavit-Hartenberg Parameters

Use the Denavit-Hartenberg (DH) parameters of the Puma560® robot to build a robot. Each rigid body is added one at a time, with the child-to-parent transform specified by the joint object.

The DH parameters define the geometry of the robot with relation to how each rigid body is attached to its parent. For convenience, setup the parameters for the Puma560 robot in a matrix. The Puma robot is a serial chain manipulator. The DH parameters are relative to the previous line in the matrix, corresponding to the previous joint attachment.

```
dhparams = [0          pi/2      0          0;  
            0.4318     0          0          0  
            0.0203     -pi/2     0.15005    0;  
            0          pi/2      0.4318     0;  
            0          -pi/2     0          0;  
            0          0          0          0];
```

Create a rigid body tree object to build the robot.

```
robot = robotics.RigidBodyTree;
```

Create the first rigid body and add it to the robot. To add a rigid body:

- 1 Create a `RigidBody` object and give it a unique name.
- 2 Create a `Joint` object and give it a unique name.
- 3 Use `setFixedTransform` to specify the body-to-body transformation using DH parameters. The last element of the DH parameters, `theta`, is ignored because the angle is dependent on the joint position.
- 4 Call `addBody` to attach the first body joint to the base frame of the robot.

```
body1 = robotics.RigidBody('body1');  
jnt1 = robotics.Joint('jnt1', 'revolute');  
  
setFixedTransform(jnt1, dhparams(1,:), 'dh');  
body1.Joint = jnt1;
```

```
addBody(robot, body1, 'base')
```

Create and add other rigid bodies to the robot. Specify the previous body name when calling `addBody` to attach it. Each fixed transform is relative to the previous joint coordinate frame.

```
body2 = robotics.RigidBody('body2');
jnt2 = robotics.Joint('jnt2', 'revolute');
body3 = robotics.RigidBody('body3');
jnt3 = robotics.Joint('jnt3', 'revolute');
body4 = robotics.RigidBody('body4');
jnt4 = robotics.Joint('jnt4', 'revolute');
body5 = robotics.RigidBody('body5');
jnt5 = robotics.Joint('jnt5', 'revolute');
body6 = robotics.RigidBody('body6');
jnt6 = robotics.Joint('jnt6', 'revolute');

setFixedTransform(jnt2, dhparams(2,:), 'dh');
setFixedTransform(jnt3, dhparams(3,:), 'dh');
setFixedTransform(jnt4, dhparams(4,:), 'dh');
setFixedTransform(jnt5, dhparams(5,:), 'dh');
setFixedTransform(jnt6, dhparams(6,:), 'dh');

body2.Joint = jnt2;
body3.Joint = jnt3;
body4.Joint = jnt4;
body5.Joint = jnt5;
body6.Joint = jnt6;

addBody(robot, body2, 'body1')
addBody(robot, body3, 'body2')
addBody(robot, body4, 'body3')
addBody(robot, body5, 'body4')
addBody(robot, body6, 'body5')
```

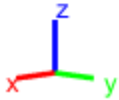
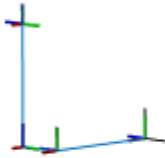
Verify that your robot was built properly by using the `showdetails` or `show` function. `showdetails` lists all the bodies in the MATLAB® command window. `show` displays the robot with a given configuration (home by default). Calls to `axis` modify the axis limits and hide the axis labels.

```
showdetails(robot)
```

```
-----
Robot: (6 bodies)
```

Idx	Body Name	Joint Name	Joint Type	Parent Name(Idx)	Children Name(s)
1	body1	jnt1	revolute	base(0)	body2(2)
2	body2	jnt2	revolute	body1(1)	body3(3)
3	body3	jnt3	revolute	body2(2)	body4(4)
4	body4	jnt4	revolute	body3(3)	body5(5)
5	body5	jnt5	revolute	body4(4)	body6(6)
6	body6	jnt6	revolute	body5(5)	

```
show(robot);  
axis([-0.5,0.5,-0.5,0.5,-0.5,0.5])  
axis off
```



### Modify a Robot Rigid Body Tree Model

Make changes to an existing RigidBodyTree object. You can get replace joints, bodies and subtrees in the rigid body tree.

Load example robots as RigidBodyTree objects.

```
load exampleRobots.mat
```

View the details of the Puma robot using showdetails.

```
showdetails(puma1)
```

```
-----  
Robot: (6 bodies)  
  
  Idx   Body Name   Joint Name   Joint Type   Parent Name(Idx)   Children Name(s)  
-----  
    1      L1       jnt1     revolute      base(0)      L2(2)  
    2      L2       jnt2     revolute      L1(1)       L3(3)  
    3      L3       jnt3     revolute      L2(2)       L4(4)  
    4      L4       jnt4     revolute      L3(3)       L5(5)  
    5      L5       jnt5     revolute      L4(4)       L6(6)  
    6      L6       jnt6     revolute      L5(5)  
-----
```

Get a specific body to inspect the properties. The only child of the L3 body is the L4 body. You can copy a specific body as well.

```
body3 = getBody(puma1, 'L3');  
childBody = body3.Children{1}
```

```
childBody =  
  RigidBody with properties:  
  
      Name: 'L4'  
      Joint: [1x1 robotics.Joint]  
      Mass: 1  
  CenterOfMass: [0 0 0]  
      Inertia: [1 1 1 0 0 0]  
      Parent: [1x1 robotics.RigidBody]  
      Children: {[1x1 robotics.RigidBody]}  
      Visuals: {}
```

```
body3Copy = copy(body3);
```

Replace the joint on the L3 body. You must create a new `Joint` object and use `replaceJoint` to ensure the downstream body geometry is unaffected. Call `setFixedTransform` if necessary to define a transform between the bodies instead of with the default identity matrices.

```
newJoint = robotics.Joint('prismatic');  
replaceJoint(puma1, 'L3', newJoint);
```

```
showdetails(puma1)
```

```
-----  
Robot: (6 bodies)
```

Idx	Body Name	Joint Name	Joint Type	Parent Name(Idx)	Children Name
1	L1	jnt1	revolute	base(0)	L2(2)
2	L2	jnt2	revolute	L1(1)	L3(3)
3	L3	prismatic	fixed	L2(2)	L4(4)
4	L4	jnt4	revolute	L3(3)	L5(5)
5	L5	jnt5	revolute	L4(4)	L6(6)
6	L6	jnt6	revolute	L5(5)	

Remove an entire body and get the resulting subtree using `removeBody`. The removed body is included in the subtree.

```
subtree = removeBody(puma1, 'L4')
```

```
subtree =
```

```
  RigidBodyTree with properties:
```

```
    NumBodies: 3
      Bodies: {1x3 cell}
        Base: [1x1 robotics.RigidBody]
    BodyNames: {'L4' 'L5' 'L6'}
      BaseName: 'L3'
      Gravity: [0 0 0]
    DataFormat: 'struct'
```

Remove the modified L3 body. Add the original copied L3 body to the L2 body, followed by the returned subtree. The robot model remains the same. See a detailed comparison through `showdetails`.

```
removeBody(puma1, 'L3');
addBody(puma1,body3Copy, 'L2')
addSubtree(puma1, 'L3',subtree)
```

```
showdetails(puma1)
```

```
-----
Robot: (6 bodies)
```

Idx	Body Name	Joint Name	Joint Type	Parent Name(Idx)	Children Name(s)
1	L1	jnt1	revolute	base(0)	L2(2)
2	L2	jnt2	revolute	L1(1)	L3(3)

3	L3	jnt3	revolute	L2(2)	L4(4)
4	L4	jnt4	revolute	L3(3)	L5(5)
5	L5	jnt5	revolute	L4(4)	L6(6)
6	L6	jnt6	revolute	L5(5)	

-----

## References

- [1] Craig, John J. *Introduction to Robotics: Mechanics and Control*. Reading, MA: Addison-Wesley, 1989.
- [2] Siciliano, Bruno. *Robotics: Modelling, Planning and Control*. London: Springer, 2009.

## Extended Capabilities

### C/C++ Code Generation

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

### See Also

`robotics.RigidBody` | `robotics.RigidBodyTree`

### Topics

“Build a Robot Step by Step”  
“Rigid Body Tree Robot Model”  
Class Attributes (MATLAB)  
Property Attributes (MATLAB)

**Introduced in R2016b**



# robotics.JointPositionBounds class

**Package:** robotics

Create constraint on joint positions of robot model

## Description

The `JointPositionBounds` object describes a constraint on the joint positions of a rigid body tree. This constraint is satisfied if the robot configuration vector maintains all joint positions within the `Bounds` specified. The configuration vector contains positions for all nonfixed joints in a `RigidBodyTree` object.

Constraint objects are used in `GeneralizedInverseKinematics` objects to specify multiple kinematic constraints on a robot.

For an example that uses multiple constraint objects, see “Plan a Reaching Trajectory With Multiple Kinematic Constraints”.

## Construction

`jointConst = robotics.JointPositionBounds(robot)` returns a joint position bounds object that represents a constraint on the configuration vector of the robot model specified by `robot`.

`jointConst = robotics.JointPositionBounds(robot, Name, Value)` returns a joint position bounds object with each specified property name set to the specified value by one or more `Name, Value` pair arguments.

## Input Arguments

**robot — Rigid body tree model**

`RigidBodyTree` object

Rigid body tree model, specified as a `RigidBodyTree` object.

## Properties

### **Bounds — Bounds on the configuration vector**

*n*-by-2 matrix

Bounds on the configuration vector, specified as an *n*-by-2 matrix. Each row of the array corresponds to a nonfixed joint on the robot model and gives the minimum and maximum position for that joint. By default, the bounds are set based on the `PositionLimits` property of each `robotics.Joint` object within the input rigid body tree model, `robot`.

### **Weights — Weights of the constraint**

`ones(1, n)` (default) | *n*-element vector

Weights of the constraint, specified as an *n*-element vector, where each element corresponds to a row in `Bounds` and gives relative weights for each bound. The default is a vector of ones to give equal weight to all joint positions. These weights are used with the `Weights` property of all the constraints specified in `GeneralizedInverseKinematics` to properly balance each constraint

## Extended Capabilities

### **C/C++ Code Generation**

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

## See Also

### **Classes**

`robotics.GeneralizedInverseKinematics` | `robotics.OrientationTarget` | `robotics.PoseTarget` | `robotics.PositionTarget`

### **Topics**

“Plan a Reaching Trajectory With Multiple Kinematic Constraints”

**Introduced in R2017a**

# lidarScan

Create object for storing 2-D lidar scan

## Description

A `lidarScan` object contains data for a single 2-D lidar (light detection and ranging) scan. The lidar scan is a laser scan for a 2-D plane with distances (`Ranges`) measured from the sensor to obstacles in the environment at specific angles (`Angles`). Use this laser scan object as an input to other robotics algorithms such as `matchScans`, `robotics.VectorFieldHistogram`, or `robotics.MonteCarloLocalization`.

## Creation

## Syntax

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

## Description

`scan = lidarScan(ranges,angles)` creates a `lidarScan` object from the `ranges` and `angles`, that represent the data collected from a lidar sensor. The `ranges` and `angles` inputs are vectors of the same length and are set directly to the `Ranges` and `Angles` properties.

`scan = lidarScan(cart)` creates a `lidarScan` object using the input Cartesian coordinates as an  $n$ -by-2 matrix. The `Cartesian` property is set directly from this input.

`scan = lidarScan(scanMsg)` creates a `lidarScan` object from a `LaserScan` ROS message object.

## Properties

### **Ranges — Range readings from lidar**

vector

Range readings from lidar, specified as a vector. This vector is the same length as `Angles`, and the vector elements are measured in meters.

Data Types: `single` | `double`

### **Angles — Angle of readings from lidar**

vector

Angle of range readings from lidar, specified as a vector. This vector is the same length as `Ranges`, and the vector elements are measured in radians. Angles are measured counter-clockwise around the positive  $z$ -axis.

Data Types: `single` | `double`

### **Cartesian — Cartesian coordinates of lidar readings**

[ $x$   $y$ ] matrix

Cartesian coordinates of lidar readings, returned as an [ $x$   $y$ ] matrix. In the lidar coordinate frame, positive  $x$  is forward and positive  $y$  is to the left.

Data Types: `single` | `double`

### **Count — Number of lidar readings**

scalar

Number of lidar readings, returned as a scalar. This scalar is also equal to the length of the `Ranges` and `Angles` vectors or the number of rows in `Cartesian`.

Data Types: `double`

## Object Functions

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

## Examples

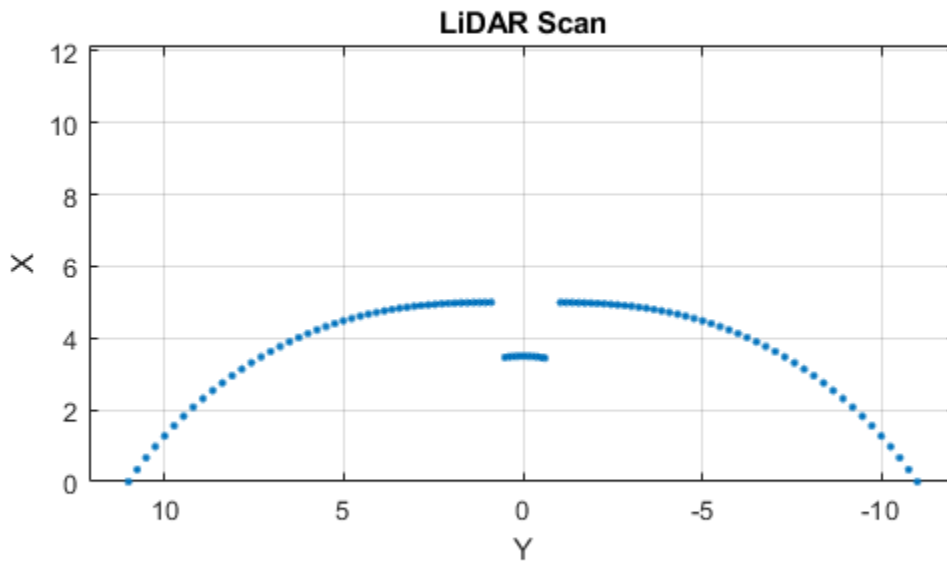
### Plot Lidar Scan and Remove Invalid Points

Specify lidar data as vectors of ranges and angles. These values include readings outside of the sensors range.

```
x = linspace(-2,2);  
ranges = abs((1.5).*x.^2 + 5);  
ranges(45:55) = 3.5;  
angles = linspace(-pi/2,pi/2,numel(ranges));
```

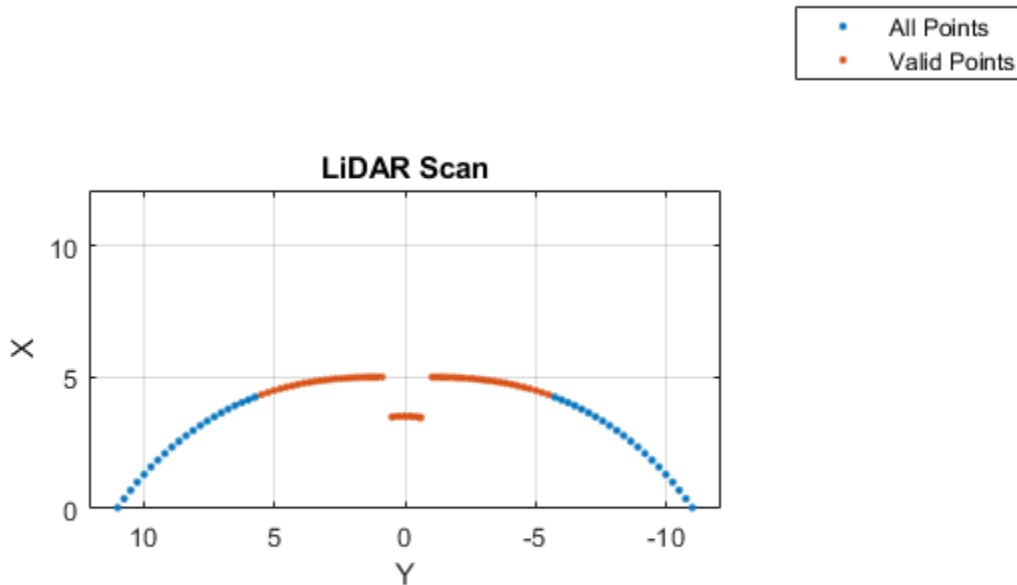
Create a lidar scan by specifying the ranges and angles. Plot all points of the lidar scan.

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



Remove invalid points based on a specified minimum and maximum range.

```
minRange = 0.1;  
maxRange = 7;  
scan2 = removeInvalidData(scan, 'RangeLimits', [minRange maxRange]);  
hold on  
plot(scan2)  
legend('All Points', 'Valid Points')
```



### Match Lidar Scans

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

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

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

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

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

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

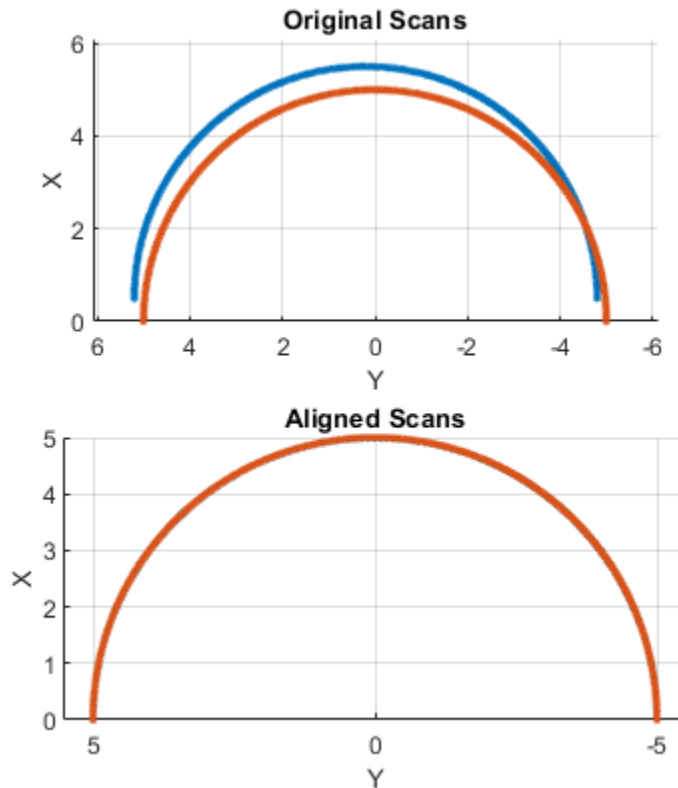
Use the `transformScan` function to align the scans by transforming the second scan into the frame of the first scan using the relative pose difference. Plot both the original scans and the aligned scans.

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

```
subplot(2,1,1);  
hold on  
plot(currScan)  
plot(refScan)  
title('Original Scans')  
hold off
```

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





## Extended Capabilities

### C/C++ Code Generation

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

Usage notes and limitations:

Lidar scans require a limited size in code generation. The lidar scans are limited to 4000 points (range and angles) as a maximum.

## **See Also**

LaserScan | matchScans | robotics.MonteCarloLocalization |  
robotics.VectorFieldHistogram | transformScan

**Introduced in R2017b**

# robotics.LidarSLAM class

**Package:** robotics

Perform localization and mapping using lidar scans

## Description

The LidarSLAM class performs simultaneous localization and mapping (SLAM) for lidar scan sensor inputs. The SLAM algorithm takes in lidar scans and attaches them to a node in an underlying pose graph. The algorithm then correlates the scans using scan matching. It also searches for loop closures, where scans overlap previously mapped regions, and optimizes the node poses in the pose graph.

## Construction

`slamObj = robotics.LidarSLAM` creates a lidar SLAM object. The default occupancy map size is 20 cells per meter. The maximum range for each lidar scan is 8 meters.

`slamObj = robotics.LidarSLAM(mapResolution,maxLidarRange)` creates a lidar SLAM object and sets the `MapResolution` and `MaxLidarRange` properties based on the inputs.

`slamObj = robotics.LidarSLAM(mapResolution,maxLidarRange,maxNumScans)` specifies the upper bound on the number of accepted scans allowed when generating code. `maxNumScans` is a positive integer. This scan limit is only required when generating code.

## Examples

### Perform SLAM Using Lidar Scans

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

## Load Data and Set Up SLAM Algorithm

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

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

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

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

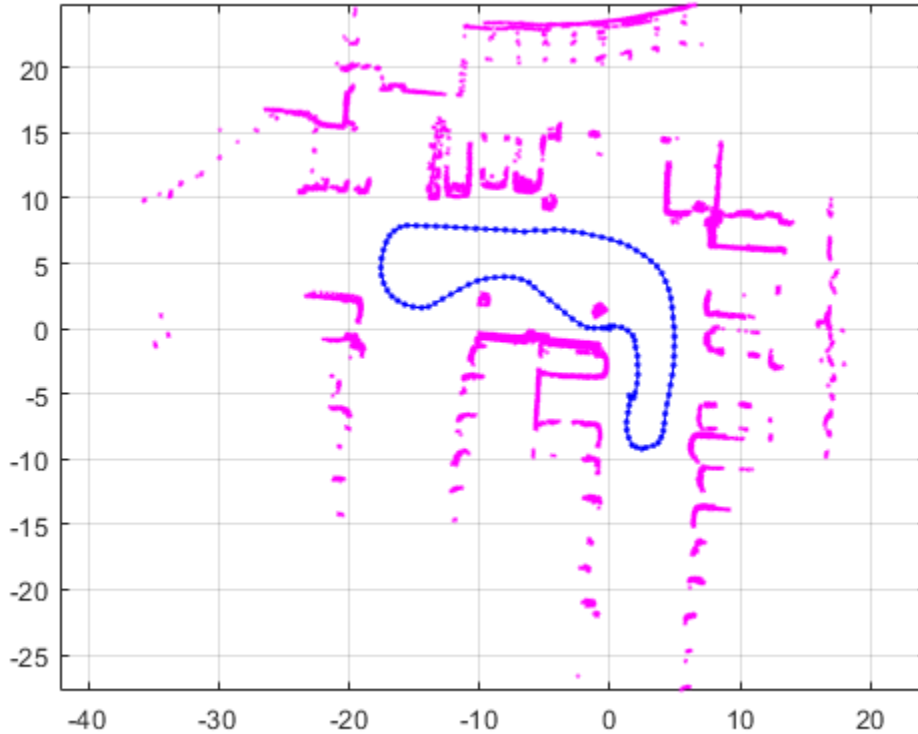
slamObj = robotics.LidarSLAM(resolution,maxRange);
slamObj.LoopClosureThreshold = 360;
slamObj.LoopClosureSearchRadius = 8;
```

## Add Scans Iteratively

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

```
for i = 1:numel(scans)
    addScan(slamObj,scans{i});

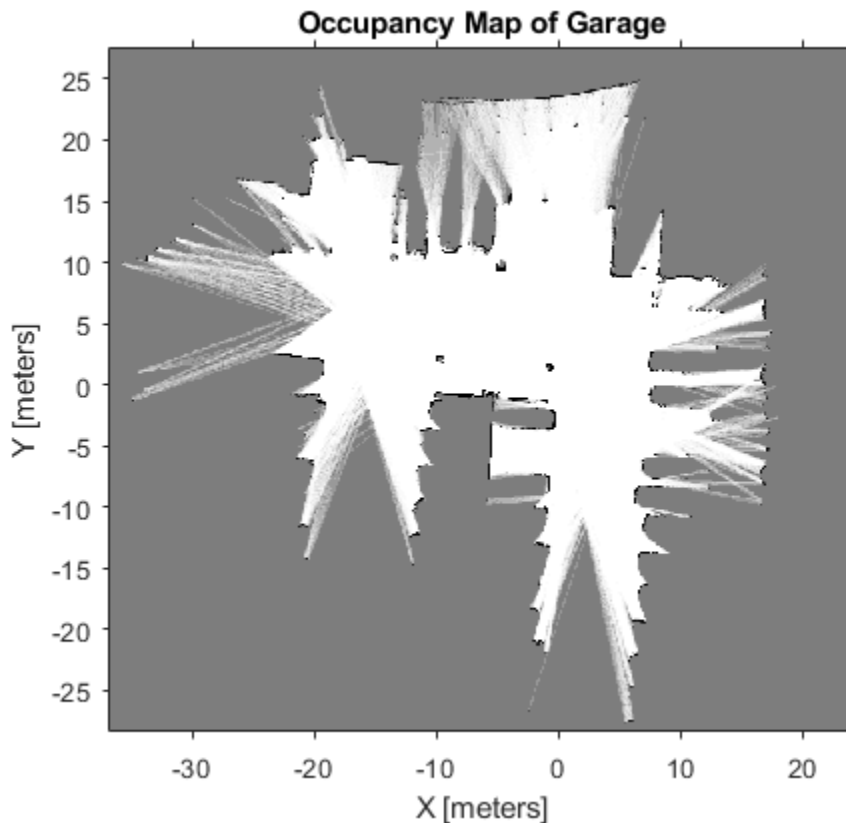
    if rem(i,10) == 0
        show(slamObj);
    end
end
```



### View Occupancy Map

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

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



## Properties

### **PoseGraph** — Underlying pose graph that connects scans

PoseGraph object

Underlying pose graph that connects scans, specified as a PoseGraph object. Adding scans to LidarSLAM updates this pose graph. When loop closures are found, the pose graph is optimized using OptimizationFcn.

### **MapResolution** — Resolution of occupancy grid map

20 cells per meter (default) | positive integer

Resolution of the occupancy grid map, specified as a positive integer in cells per meter. Specify the map resolution on construction.

### **MaxLidarRange — Maximum range of lidar sensor**

8 meters (default) | positive scalar

Maximum range of the lidar sensor, specified as a positive scalar in meters. Specify the maximum range on construction.

### **OptimizationFcn — Pose graph optimization function**

optimizePoseGraph (default) | function handle

Pose graph optimization function, specified as a function handle. By default, the algorithm calls the `optimizePoseGraph` function. To specify your own optimization method, the class requires the function signature to be:

```
[updatedPose,stat] = myOptimizationFcn(poseGraph)
```

`poseGraph` is a `PoseGraph` object. `updatedPose` is an  $n$ -by-3 vector of `[x y theta]` poses listed in sequential node ID order. `stat` is a structure containing a `ResidualError` field as a positive scalar. Use the `stat` structure to include other information relevant to your optimization.

### **LoopClosureThreshold — Threshold for accepting loop closures**

100 (default) | positive scalar

Threshold on the score from the scan matching algorithm for accepting loop closures, specified as a positive scalar. Higher thresholds correspond to a better match, but scores vary based on sensor data.

### **LoopClosureSearchRadius — Search radius for loop closure detection**

8 meters (default) | positive scalar

Search radius for loop closure detection, specified as a positive scalar. Increasing this radius affects performance by increasing search time. Tune this distance based on your environment and the expected robot trajectory.

### **LoopClosureMaxAttempts — Number of attempts at finding loop closures**

1 (default) | positive integer

Number of attempts at finding looping closures, specified as a positive integer. Increasing the number of attempts affects performance by increasing search time.

## **LoopClosureAutoRollback — Allow automatic rollback of added loop closures**

`true` (default) | `false`

Allow automatic rollback of added loop closures, specified as `true` or `false`. The SLAM object tracks the residual error returned by the `OptimizationFcn`. If it detects a sudden change in the residual error and this property is `true`, it rejects (rolls back) the loop closure.

## **OptimizationInterval — Number of loop closures accepted to trigger optimization**

`1` (default) | positive integer

Number of loop closures accepted to trigger optimization, specified as a positive integer. By default, the `PoseGraph` is optimized every time `LidarSLAM` adds a loop closure.

## **MovementThreshold — Minimum change in pose required to process scans**

`[0 0]` (default) | `[translation rotation]`

Minimum change in pose required to process scans, specified as a `[translation rotation]` vector. A relative pose change for a newly added scan is calculated as `[x y theta]`. If the translation in `xy`-position or rotation of `theta` exceeds these thresholds, the `LidarSLAM` object accepts the scan and adds a pose is added to the `PoseGraph`.

## **Methods**

<code>addScan</code>	Add scan to lidar SLAM map
<code>copy</code>	Copy lidar SLAM object
<code>removeLoopClosures</code>	Remove loop closures from pose graph
<code>scansAndPoses</code>	Extract scans and corresponding poses
<code>show</code>	Plot scans and robot poses

## **Definitions**

### **SLAM**

Simultaneous localization and mapping (SLAM) is a general concept for algorithms correlating different sensor readings to build a map of a robot environment and track



pose estimates. Different algorithms use different types of sensors and methods for correlating data.

The LidarSLAM algorithm uses lidar scans and odometry information as sensor inputs. The lidar scans map the environment and are correlated between each other to build an underlying pose graph of the robot trajectory. Odometry information is an optional input that gives an initial pose estimate for the scans to aid in the correlation. Scan matching algorithms correlate scans to previously added scans to estimate the relative pose between them and add them to an underlying pose graph.

The pose graph contains nodes connected by edges that represent the relative poses of the robot. Edges specify constraints on the node as an information matrix. To correct for drifting pose estimates, the algorithm optimizes over the whole pose graph whenever it detects loop closures.

The algorithm assumes that data comes from a robot navigating an environment and incrementally getting laser scans along its path. Therefore, scans are first compared to the most recent scan to identify relative poses and are added to the pose graph incrementally. However, the algorithm also searches for loop closures, which identify when the robot scans an area that was previously visited.

When working with SLAM algorithms, the environment and robot sensors affect the performance and data correlation quality. Tune your parameters properly for your expected environment or dataset.

## References

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

## Extended Capabilities

### C/C++ Code Generation

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

Use this syntax when constructing LidarSLAM objects for code generation:

```
slamObj= robotics.LidarSLAM(mapResolution,maxLidarRange,maxNumScans)
```

specifies the upper bound on the number of accepted scans allowed when generating code. `maxNumScans` is a positive integer. This scan limit is only required when generating code.

### See Also

`optimizePoseGraph` | `robotics.PoseGraph`

### Topics

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

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

**Introduced in R2018a**

# robotics.LikelihoodFieldSensorModel class

**Package:** robotics

Create a likelihood field range sensor model

## Description

`LikelihoodFieldSensorModel` creates a likelihood field sensor model object for range sensors. This object contains specific sensor model parameters. You can use this object to specify the model parameters in a `robotics.MonteCarloLocalization` object.

## Construction

`lf = robotics.LikelihoodFieldSensorModel` creates a likelihood field sensor model object for range sensors.

## Properties

### Map — Occupancy grid representing the map

`BinaryOccupancyGrid` object (default)

Occupancy grid representing the map, specified as a `robotics.BinaryOccupancyGrid` object. This object represents the environment of the robot as a grid with binary values indicating obstacles as `true` (1) and free locations as `false` (0).

### SensorPose — Pose of the range sensor relative to the robot

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

Pose of the range sensor relative to the coordinate frame of the robot, specified as a three-element vector, `[x y theta]`.

### SensorLimits — Minimum and maximum range of sensor

`[0 12]` (default) | two-element vector

Minimum and maximum range of sensor, specified as a two-element vector in meters.

## **NumBeams — Number of beams used for likelihood computation**

60 (default) | scalar

Number of beams used for likelihood computation, specified as a scalar. The computation efficiency can be improved by specifying a smaller number of beams than the actual number available from the sensor.

## **MeasurementNoise — Standard deviation for measurement noise**

0.2 (default) | scalar

Standard deviation for measurement noise, specified as a scalar.

## **RandomMeasurementWeight — Weight for probability of random measurement**

0.05 (default) | scalar

Weight for probability of random measurement, specified as a scalar. This scalar is the probability that the measurement is not accurate due to random interference.

## **ExpectedMeasurementWeight — Weight for probability of expected measurement**

0.95 (default) | scalar

Weight for probability of expected measurement, specified as a scalar. The weight is the probability of getting a correct range measurement within the noise limits specified in MeasurementNoise property.

## **MaxLikelihoodDistance — Maximum distance to find nearest obstacles**

2.0 (default) | scalar

Maximum distance to find nearest obstacles, specified as a scalar in meters.

## **Limitations**

If you change your sensor model after using it with the MonteCarloLocalization object, call `release` on that object beforehand. For example:

```
mcl = robotics.MonteCarloLocalization(...);  
[isUpdated,pose,covariance] = mcl(...);  
release(mcl)  
mcl.SensorModel.PropName = value;
```

## See Also

`robotics.MonteCarloLocalization` | `robotics.OdometryMotionModel`

## Topics

“Localize TurtleBot Using Monte Carlo Localization”

“Monte Carlo Localization Algorithm”

Class Attributes (MATLAB)

Property Attributes (MATLAB)

**Introduced in R2016a**

## robotics.MonteCarloLocalization

**Package:** robotics

Localize robot using range sensor data and map

### Description

The `robotics.MonteCarloLocalization` System object creates a Monte Carlo localization (MCL) object. The MCL algorithm is used to estimate the position and orientation of a robot in its environment using a known map of the environment, lidar scan data, and odometry sensor data.

To localize the robot, the MCL algorithm uses a particle filter to estimate the robot's position. The particles represent the distribution of likely states for the robot, where each particle represents a possible robot state. The particles converge around a single location as the robot moves in the environment and senses different parts of the environment using a range sensor. An odometry sensor measures the robot's motion.

A `robotics.MonteCarloLocalization` object takes the pose and lidar scan data as inputs. The input lidar scan sensor data is given in its own coordinate frame, and the algorithm transforms the data according to the `SensorModel`. `SensorPose` property that you must specify. The input pose is computed by integrating the odometry sensor data over time. If the change in pose is greater than any of the specified update thresholds, `UpdateThresholds`, then the particles are updated and the algorithm computes a new state estimate from the particle filter. The particles are updated using this process:

- 1 The particles are propagated based on the change in the pose and the specified motion model, `MotionModel`.
- 2 The particles are assigned weights based on the likelihood of receiving the range sensor reading for each particle. These likelihood weights are based on the sensor model you specify in `SensorModel`.
- 3 Based on the `ResamplingInterval` property, the particles are resampled from the posterior distribution, and the particles of low weight are eliminated. For example, a resampling interval of 2 means that the particles are resampled after every other update.

The outputs of the object are the estimated pose and covariance, and the value of `isUpdated`. This estimated state is the mean and covariance of the highest weighted cluster of particles. The output pose is given in the map's coordinate frame that is specified in the `SensorModel.Map` property. If the change in pose is greater than any of the update thresholds, then the state estimate has been updated and `isUpdated` is `true`. Otherwise, `isUpdated` is `false` and the estimate remains the same. For continuous tracking the best estimate of a robot's state, repeat this process of propagating particles, evaluating their likelihood, and resampling.

To estimate robot pose and covariance using lidar scan data:

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

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

## Creation

### Syntax

```
mcl = robotics.MonteCarloLocalization  
mcl = robotics.MonteCarloLocalization(Name,Value)
```

### Description

`mcl = robotics.MonteCarloLocalization` returns an MCL object that estimates the pose of a robot using a map, a range sensor, and odometry data. By default, an empty map is assigned, so a valid map assignment is required before using the object.

`mcl = robotics.MonteCarloLocalization(Name,Value)` creates an MCL object with additional options specified by one or more `Name,Value` pair arguments.

`Name` is a property name and `Value` is the corresponding value. `Name` must appear inside single quotes ( ' '). You can specify several name-value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

## Properties

### **InitialPose** — Initial pose of robot

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

Initial pose of the robot used to start localization, specified as a three-element vector, [x y theta], that indicates the position and heading of the robot. Initializing the MCL object with an initial pose estimate enables you to use a smaller value for the maximum number of particles and still converge on a location.

### **InitialCovariance** — Covariance of initial pose

diag([1 1 1]) (default) | diagonal matrix | three-element vector | scalar

Covariance of the Gaussian distribution for the initial pose, specified as a diagonal matrix. Three-element vector and scalar inputs are converted to a diagonal matrix. This matrix gives an estimate of the uncertainty of the `InitialPose`.

### **GlobalLocalization** — Flag to start global localization

false (default) | true

Flag indicating whether to perform global localization, specified as `false` or `true`. The default value, `false`, initializes particles using the `InitialPose` and `InitialCovariance` properties. A `true` value initializes uniformly distributed particles in the entire map and ignores the `InitialPose` and `InitialCovariance` properties. Global localization requires a large number of particles to cover the entire workspace. Use global localization only when the initial estimate of robot location and orientation is not available.

### **ParticleLimits** — Minimum and maximum number of particles

[500 5000] (default) | two-element vector

Minimum and maximum number of particles, specified as a two-element vector, [min max].

### **SensorModel** — Likelihood field sensor model

`LikelihoodFieldSensorModel` object

Likelihood field sensor model, specified as a `LikelihoodFieldSensorModel` object. The default value uses the default `robotics.LikelihoodFieldSensorModel` object. After using the object to get output, call `release` on the object to make changes to `SensorModel`. For example:



```
mcl = robotics.MonteCarloLocalization(_);
[isUpdated,pose,covariance] = mcl(_);
release(mcl)
mcl.SensorModel.PropName = value;
```

### **MotionModel — Odometry motion model for differential drive**

OdometryMotionModel object

Odometry motion model for differential drive, specified as an OdometryMotionModel object. The default value uses the default robotics.OdometryMotionModel object. After using the object to get output, call release on the object to make changes to MotionModel. For example:

```
mcl = robotics.MonteCarloLocalization(_);
[isUpdated,pose,covariance] = mcl(_);
release(mcl)
mcl.MotionModel.PropName = value;
```

### **UpdateThresholds — Minimum change in states required to trigger update**

[0.2 0.2 0.2] (default) | three-element vector

Minimum change in states required to trigger update, specified as a three-element vector. The localization updates the particles if the minimum change in any of the [x y theta] states is met. The pose estimate updates only if the particle filter is updated.

### **ResamplingInterval — Number of filter updates between resampling of particles**

1 (default) | positive integer

Number of filter updates between resampling of particles, specified as a positive integer.

### **UseLidarScan — Use lidarScan object as scan input**

false (default) | true

Use a lidarScan object as scan input, specified as either false or true.

## **Usage**

**Note** For versions earlier than R2016b, use the step function to run the System object™ algorithm. The arguments to step are the object you created, followed by the arguments shown in this section.

For example, `y = step(obj, x)` and `y = obj(x)` perform equivalent operations.

---

## Syntax

```
[isUpdated, pose, covariance] = mcl(odomPose, scan)
```

```
[isUpdated, pose, covariance] = mcl(odomPose, ranges, angles)
```

## Description

`[isUpdated, pose, covariance] = mcl(odomPose, scan)` estimates the pose and covariance of a robot using the MCL algorithm. The estimates are based on the pose calculated from the specified robot odometry, `odomPose`, and the specified lidar scan sensor data, `scan`. `mcl` is the `robotics.MonteCarloLocalization` object. `isUpdated` indicates whether the estimate is updated based on the `UpdateThreshold` property.

To enable this syntax, you must set the `UseLidarScan` property to `true`. For example:

```
mcl = robotics.MonteCarloLocalization('UseLidarScan', true);  
...  
[isUpdated, pose, covariance] = mcl(odomPose, scan);
```

`[isUpdated, pose, covariance] = mcl(odomPose, ranges, angles)` specifies the lidar scan data as `ranges` and `angles`.

## Input Arguments

### **odomPose** — Pose based on odometry

three-element vector

Pose based on odometry, specified as a three-element vector, `[x y theta]`. This pose is calculated by integrating the odometry over time.

### **scan** — Lidar scan readings

lidarScan object

Lidar scan readings, specified as a `lidarScan` object.

## Dependencies

To use this argument, you must set the `UseLidarScan` property to `true`.

```
mcl.UseLidarScan = true;
```

## **ranges** — Range values from scan data

vector

Range values from scan data, specified as a vector with elements measured in meters. These range values are distances from a laser scan sensor at the specified `angles`. The `ranges` vector must have the same number of elements as the corresponding `angles` vector.

## **angles** — Angle values from scan data

vector

Angle values from scan data, specified as a vector with elements measured in radians. These angle values are the angles at which the specified `ranges` were measured. The `angles` vector must be the same length as the corresponding `ranges` vector.

## Output Arguments

### **isUpdated** — Flag for pose update

logical

Flag for pose update, returned as a logical. If the change in pose is more than any of the update thresholds, then the output is `true`. Otherwise, it is `false`. A `true` output means that updated pose and covariance are returned. A `false` output means that pose and covariance are not updated and are the same as at the last update.

### **pose** — Current pose estimate

three-element vector

Current pose estimate, returned as a three-element vector, `[x y theta]`. The pose is computed as the mean of the highest-weighted cluster of particles.

### **covariance** — Covariance estimate for current pose

matrix

Covariance estimate for current pose, returned as a matrix. This matrix gives an estimate of the uncertainty of the current pose. The covariance is computed as the covariance of the highest-weighted cluster of particles.

## Object Functions

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

```
release(obj)
```

### Specific to robotics.MonteCarloLocalization

`getParticles` Get particles from localization algorithm

### Common to All System Objects

`step` Run System object algorithm

`release` Release resources and allow changes to System object property values and input characteristics

`reset` Reset internal states of System object

## Examples

### Estimate Robot Pose from Range Sensor Data

Create a `MonteCarloLocalization` object, assign a sensor model, and calculate a pose estimate using the `step` method.

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Create an MCL object.

```
mcl = robotics.MonteCarloLocalization;
```

Assign a sensor model with an occupancy grid map to the object.

```
sm = robotics.LikelihoodFieldSensorModel;
p = zeros(200,200);
sm.Map = robotics.OccupancyGrid(p,20);
mcl.SensorModel = sm;
```

Create sample laser scan data input.

```
ranges = 10*ones(1,300);
ranges(1,130:170) = 1.0;
angles = linspace(-pi/2,pi/2,300);
odometryPose = [0 0 0];
```

Estimate robot pose and covariance.

```
[isUpdated,estimatedPose,covariance] = mcl(odometryPose,ranges,angles)
```

```
isUpdated = logical
    1
```

```
estimatedPose = 1×3
```

```
    0.0343    0.0193    0.0331
```

```
covariance = 3×3
```

```
    0.9467    0.0048         0
    0.0048    0.9025         0
         0         0    1.0011
```

## References

- [1] Thrun, Sebastian, Wolfram Burgard, and Dieter Fox. *Probabilistic Robotics*. MIT Press, 2005.
- [2] Dellaert, F., D. Fox, W. Burgard, and S. Thrun. "Monte Carlo Localization for Mobile Robots." *Proceedings 1999 IEEE International Conference on Robotics and Automation*.

## See Also

`lidarScan` | `robotics.LikelihoodFieldSensorModel` |  
`robotics.OdometryMotionModel`

## Topics

“Localize TurtleBot Using Monte Carlo Localization”

“Monte Carlo Localization Algorithm”

Class Attributes (MATLAB)

Property Attributes (MATLAB)

**Introduced in R2016a**

# 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.

---

**Note** This object requires you to install the UAV Library for Robotics System Toolbox. To install add-ons, use `roboticsAddons` and select the desired add-on.

---

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

<code>control</code>	Control commands for UAV
<code>derivative</code>	Time derivative of UAV states
<code>environment</code>	Environmental inputs for UAV
<code>state</code>	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.

**Note:** To use UAV algorithms, you must install the UAV Library for Robotics System Toolbox®. To install, use `roboticsAddons`.

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);
```

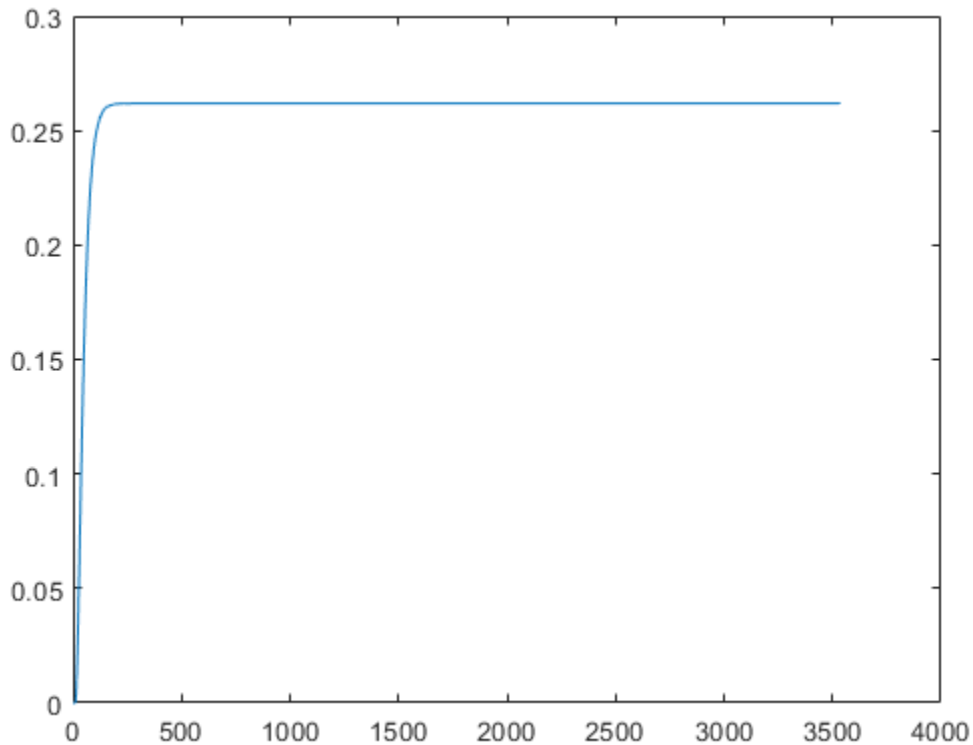
Simulate the UAV state using `ode45` integration. The `y` field outputs the fixed-wing UAV states as a 13-by- $n$  matrix.

```
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,:))
```



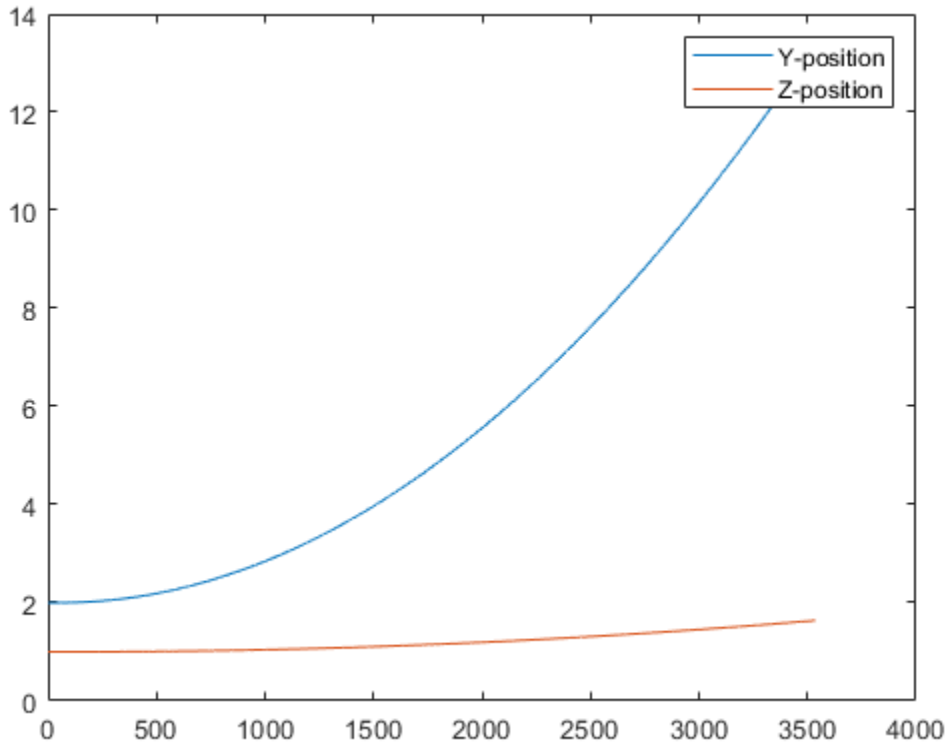
Plot the change in the Y and Z positions. With the specified thrust and roll angle, the multirotor should fly over and lose some altitude. A positive value for Z is expected as positive Z is down.

```
figure  
plot(simOut.y(2,:));  
hold on  
plot(simOut.y(3,:));
```

```

legend('Y-position','Z-position')
hold off

```

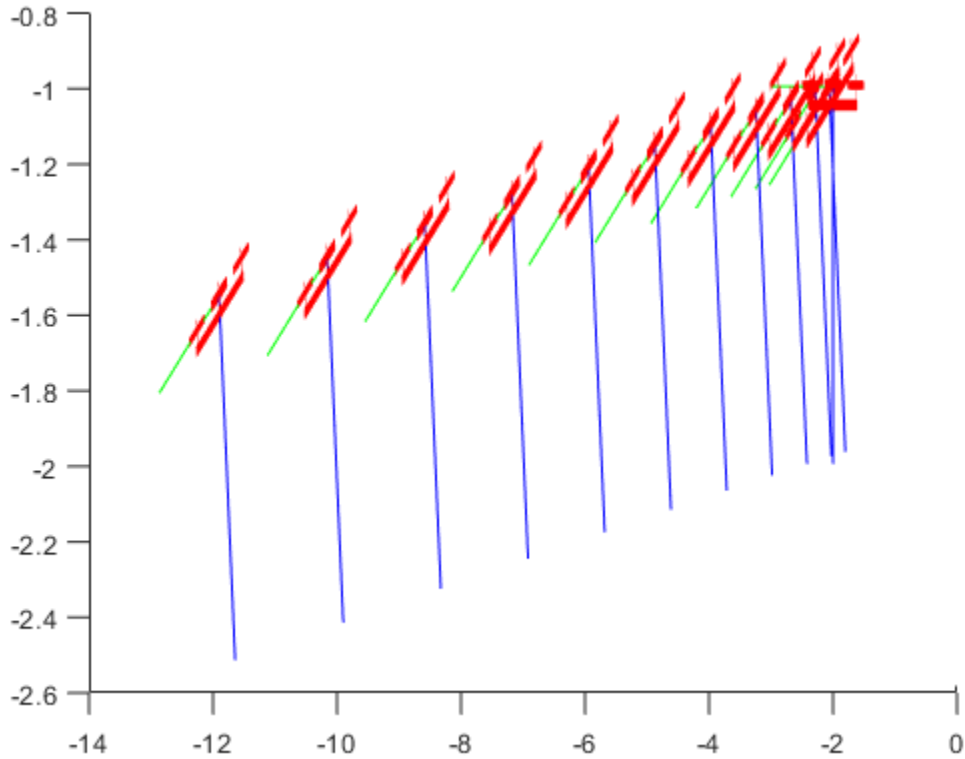


You can also plot the multirotor trajectory using `plotTransforms`. Create the translation and rotation vectors from the simulated state. Downsample (every 300th element) and transpose the `simOut` elements, and convert the Euler angles to quaternions. Specify the mesh as the `multirotor.stl` file and 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])

```



## Definitions

### UAV Coordinate Systems

The UAV Library for Robotics System 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,  $\psi$ . Then, rotate about the intermediate  $y$ -axis by the pitch angle,  $\phi$ . Then, rotate about the intermediate  $x$ -axis by the roll angle,  $\theta$ .

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

When converting coordinates from the world (earth) frame to the body frame of the UAV, the rotation matrix 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:

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

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

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

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

This model assumes the autopilot takes in commanded roll, pitch, yaw angles,  $[\psi^c, \theta^c, \phi^c]$  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  $KP_\alpha$  and  $KD_\alpha$ , where  $\alpha$  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:

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

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 | roboticsAddons | 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**



# robotics.OccupancyGrid class

**Package:** robotics

Create occupancy grid with probabilistic values

## Description

OccupancyGrid creates a 2-D occupancy grid map. Each cell in the occupancy grid has a value representing the probability of the occupancy of that cell. Values close to 1 represent a high certainty that the cell contains an obstacle. Values close to 0 represent certainty that the cell is not occupied and obstacle free.

Occupancy grids are used in robotics algorithms such as path planning (see `robotics.PRM`). They are also used in mapping applications for finding collision-free paths, performing collision avoidance, and calculating localization (see `robotics.MonteCarloLocalization`). You can modify your occupancy grid to fit your specific application.

The `OccupancyGrid` objects support world and grid coordinates. The world coordinates origin is defined by the `GridLocationInWorld` property of the object, which defines the bottom-left corner of the grid. The number and size of grid locations are defined by the `Resolution` property. The first grid location with index  $(1, 1)$  begins in the top-left corner of the grid.

Use the `OccupancyGrid` class to create 2-D maps of an environment with probability values representing different obstacles in your world. You can specify exact probability values of cells or include observations from sensors such as laser scanners.

Probability values are stored using a binary Bayes filter to estimate the occupancy of each grid cell. A log-odds representation is used, with values stored as `int16` to reduce the map storage size and allow for real-time applications.

If memory size is a limitation, consider using `robotics.BinaryOccupancyGrid` instead. The binary occupancy grid uses less memory with binary values, but still works with Robotics System Toolbox algorithms and other applications.

## Construction

`map = robotics.OccupancyGrid(width,height)` creates a 2-D occupancy grid object representing a world space of `width` and `height` in meters. The default grid resolution is 1 cell per meter.

`map = robotics.OccupancyGrid(width,height,resolution)` creates an occupancy grid with a specified grid resolution in cells per meter.

`map = robotics.OccupancyGrid(rows,cols,resolution,"grid")` creates an occupancy grid with the specified number of rows and columns and with the resolution in cells per meter.

`map = robotics.OccupancyGrid(p)` creates an occupancy grid from the values in matrix `p`. The grid size matches the size of the matrix, with each cell probability value interpreted from the matrix location.

`map = robotics.OccupancyGrid(p,resolution)` creates an occupancy grid from the specified matrix and resolution in cells per meter.

## Input Arguments

### **width — Map width**

scalar in meters

Map width, specified as a scalar in meters.

Data Types: `double`

### **height — Map height**

scalar in meters

Map height, specified as a scalar in meters.

Data Types: `double`

### **resolution — Grid resolution**

1 (default) | scalar in cells per meter

Grid resolution, specified as a scalar in cells per meter.

Data Types: `double`

**p — Input occupancy grid**

matrix of probability values from 0 to 1

Input occupancy grid, specified as a matrix of probability values from 0 to 1. The size of the grid matches the size of the matrix. Each matrix element corresponds to the probability of the grid cell location being occupied. Values close to 0 represent a high certainty that the cell contains an obstacle. Values close to 1 represent certainty that the cell is not occupied and obstacle free.

Data Types: double

## Properties

**FreeThreshold — Threshold to consider cells as obstacle-free**

scalar

Threshold to consider cells as obstacle-free, specified as a scalar. Probability values below this threshold are considered obstacle free. This property also defines the free locations for path planning when using `robotics.PRM`.

**OccupiedThreshold — Threshold to consider cells as occupied**

scalar

Threshold to consider cells as occupied, specified as a scalar. Probability values above this threshold are considered occupied.

**ProbabilitySaturation — Saturation limits for probability**

[0.001 0.999] (default) | [min max] vector

Saturation limits for probability, specified as a [min max] vector. Values above or below these saturation values are set to the min and max values. This property reduces oversaturating of cells when incorporating multiple observations.

**GridSize — Number of rows and columns in grid**

[rows cols] vector

This property is read-only.

Number of rows and columns in grid, stored as a [rows cols] vector.

**Resolution — Grid resolution**

1 (default) | scalar in cells per meter

Grid resolution, stored as a scalar in cells per meter. This value is read only.

**XWorldLimits — Minimum and maximum world range values of x-coordinates**

[min max] vector

Minimum and maximum world range values of x-coordinates, stored as a [min max] vector. This value is read only.

**YWorldLimits — Minimum and maximum world range values of y-coordinates**

[min max] vector

Minimum and maximum world range values of y-coordinates, stored as a [min max] vector. This value is read only.

**GridLocationInWorld — [x,y] world coordinates of grid**

[0 0] (default) | two-element vector

[x, y] world coordinates of the bottom-left corner of the grid, specified as a two-element vector.

## Methods

checkOccupancy	Check locations for free, occupied, or unknown values
copy	Create copy of occupancy grid
getOccupancy	Get occupancy of a location
grid2world	Convert grid indices to world coordinates
inflate	Inflate each occupied grid location
insertRay	Insert ray from laser scan observation
occupancyMatrix	Convert occupancy grid to double matrix
raycast	Compute cell indices along a ray
rayIntersection	Compute map intersection points of rays
setOccupancy	Set occupancy of a location
show	Show grid values in a figure
updateOccupancy	Integrate probability observation at a location
world2grid	Convert world coordinates to grid indices

## Examples

### Insert Laser Scans Into Occupancy Grid

Take range and angle readings from a laser scan and insert these readings into an occupancy grid.

Create an empty occupancy grid map.

```
map = robotics.OccupancyGrid(10,10,20);
```

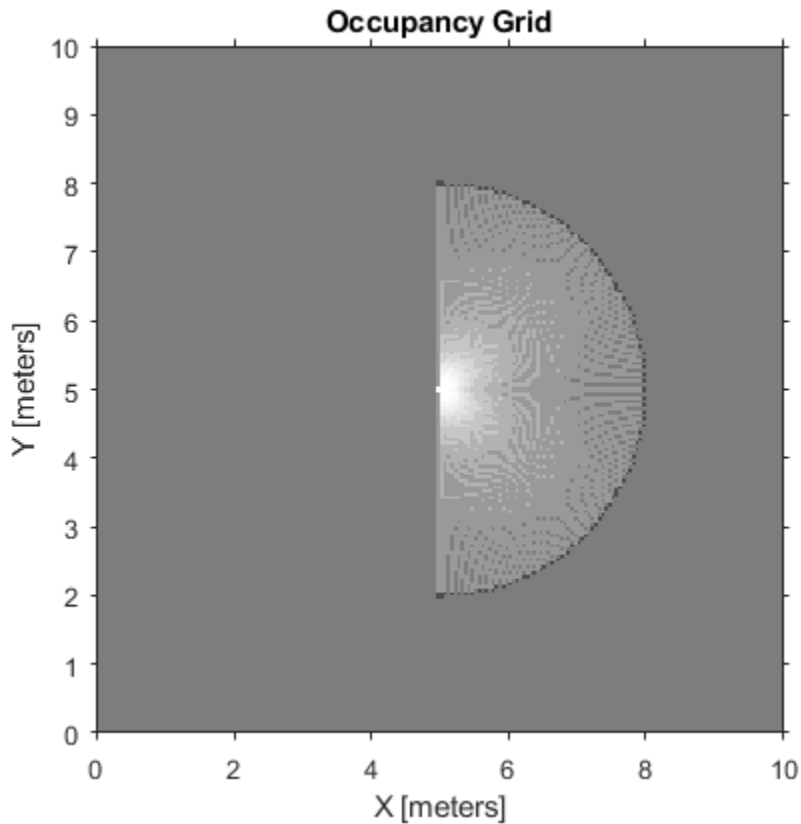
Insert a laser scan into the occupancy grid. Specify the pose of the robot ranges and angles and the max range of the laser scan.

```
pose = [5,5,0];  
ranges = 3*ones(100, 1);  
angles = linspace(-pi/2, pi/2, 100);  
maxrange = 20;
```

```
insertRay(map,pose,ranges,angles,maxrange);
```

Show the map to see the results of inserting the laser scan. Check the occupancy of the spot directly in front of the robot.

```
show(map)
```

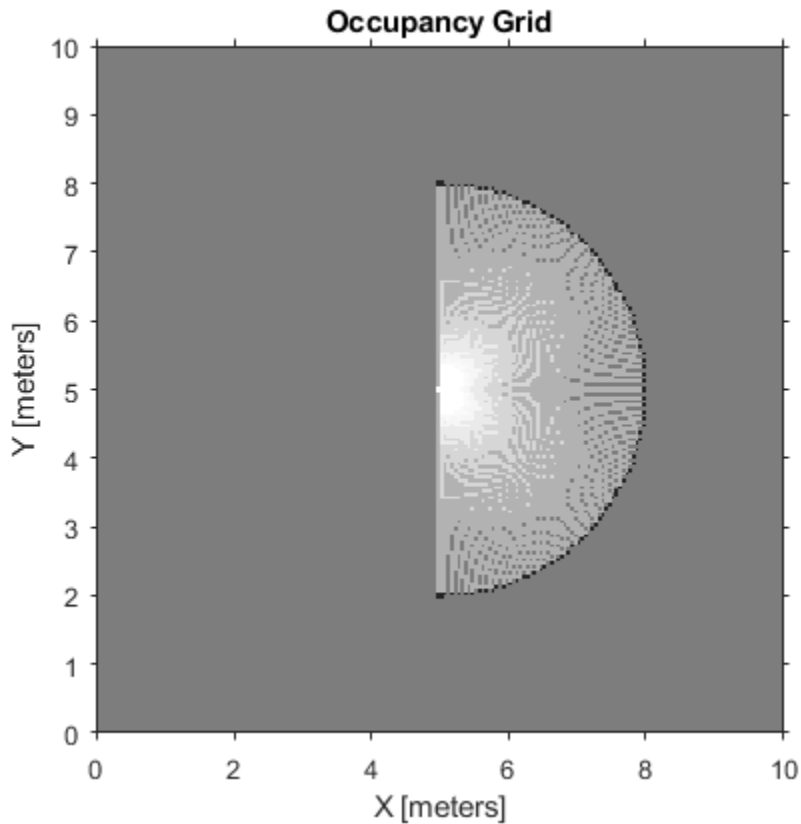


```
getOccupancy(map, [8 5])
```

```
ans = 0.7000
```

Add a second reading and view the update to the occupancy values. The additional reading increases the confidence in the readings. The free and occupied values become more distinct.

```
insertRay(map, pose, ranges, angles, maxrange);  
show(map)
```



```
getOccupancy(map,[8 5])
```

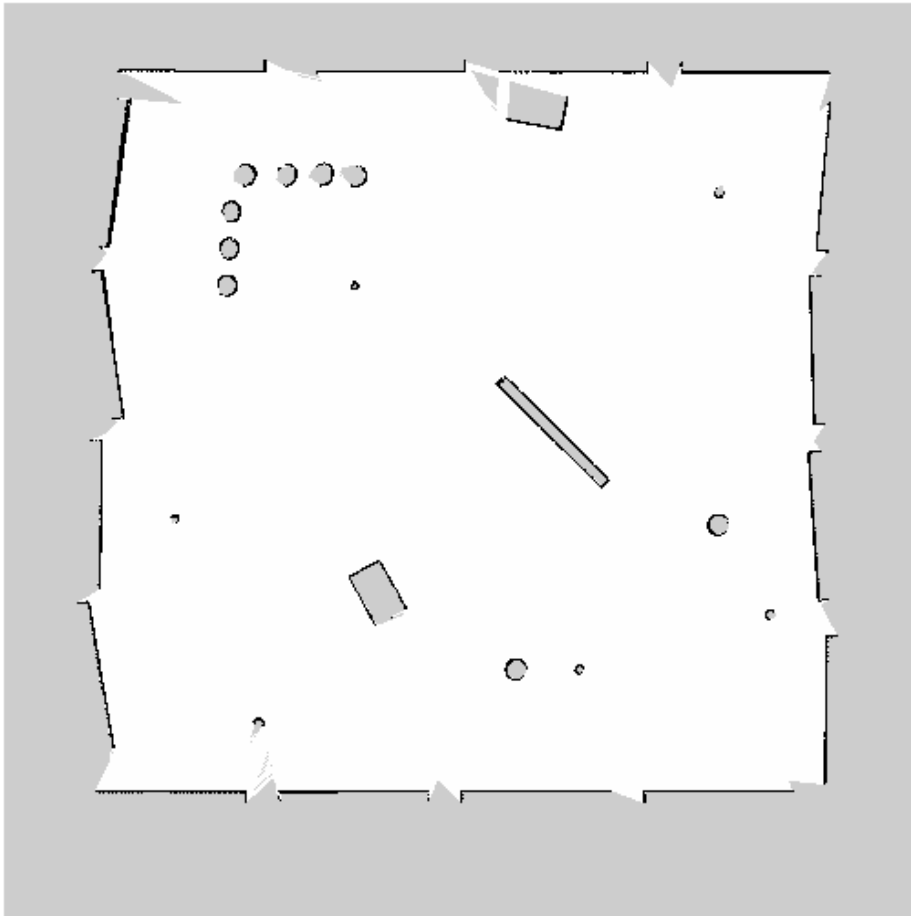
```
ans = 0.8448
```

### Convert PGM Image to Map

Convert a portable graymap (.pgm) file containing a ROS map into an `OccupancyGrid` map for use in MATLAB.

Import the image using `imread`. Crop the image to the relevant area.

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



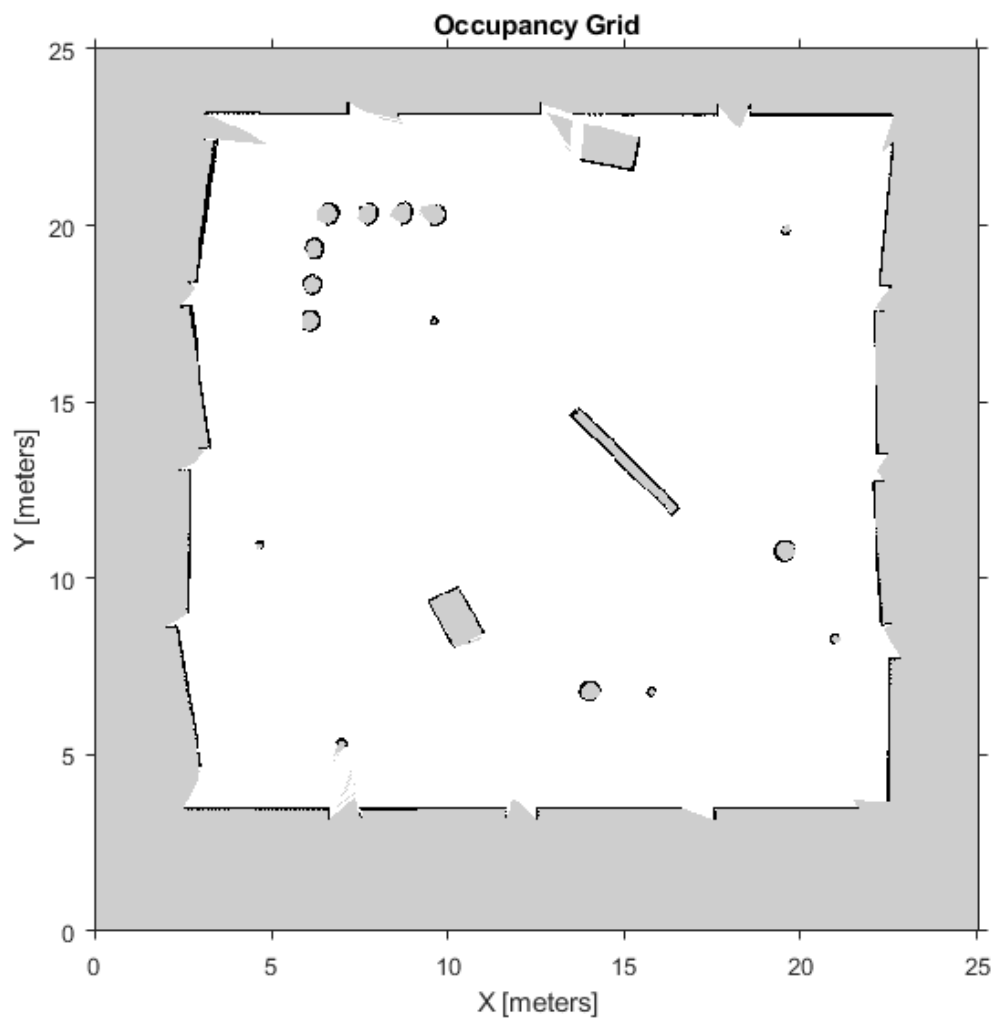


PGM values are expressed from 0 to 255 as `uint8`. Normalize these values by converting the cropped image to `double` and dividing each cell by 255. This image shows obstacles as values close to 0. Subtract the normalized image from 1 to get occupancy values with 1 representing occupied space.

```
imageNorm = double(imageCropped)/255;  
imageOccupancy = 1 - imageNorm;
```

Create the `OccupancyGrid` object using an adjusted map image. The imported map resolution is 20 cells per meter.

```
map = robotics.OccupancyGrid(imageOccupancy,20);  
show(map)
```



## Limitations

Occupancy values have a limited resolution of  $\pm 0.001$ . The values are stored as `int16` using a log-odds representation. This data type limits resolution, but saves you memory when storing large maps in MATLAB. When calling `set` and then `get`, the value returned might not equal the value you set. For more information, see the log-odds representations section in “Occupancy Grids”.

## Extended Capabilities

### C/C++ Code Generation

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

### See Also

`readOccupancyGrid` | `robotics.BinaryOccupancyGrid` | `robotics.PRM` | `robotics.PurePursuit` | `writeOccupancyGrid`

### Topics

“Mapping With Known Poses”

“Occupancy Grids”

Class Attributes (MATLAB)

Property Attributes (MATLAB)

**Introduced in R2016b**

## robotics.OccupancyMap3D class

**Package:** robotics

Create 3-D occupancy map

### Description

The `OccupancyMap3D` class stores a 3-D map and map information. The map is stored as probabilistic values in an octree data structure on page 1-187. The class handles arbitrary environments and expands its size dynamically based on observation inputs. You can add observations as point clouds or as specific `xyz` locations. These observations update the probability values. Probabilistic values represent the occupancy of locations. The octree data structure trims data appropriately to remain efficient both in memory and on disk.

### Construction

`omap = robotics.OccupancyMap3D` creates an empty 3-D occupancy map with no observations and default property values.

`omap = robotics.OccupancyMap3D(res)` specifies a map resolution in cells/meter and sets the `Resolution` property.

`omap = robotics.OccupancyMap3D(res, Name, Value)` creates an object with additional options specified by one or more `Name, Value` pair arguments. For example, `'FreeThreshold', 0.25` sets the threshold to consider cells obstacle-free as a probability value of 0.25. Enclose each property name in quotes.

### Properties

#### Resolution — Grid resolution

1 (default) | positive scalar

Grid resolution in cells per meter, specified as a scalar. Specify resolution on construction. Inserting observations with precisions higher than this value are rounded down and applied at this resolution.

**FreeThreshold — Threshold to consider cells as obstacle-free**`0.2` (default) | positive scalar

Threshold to consider cells as obstacle-free, specified as a positive scalar. Probability values below this threshold are considered obstacle-free.

**OccupiedThreshold — Threshold to consider cells as occupied**`0.65` (default) | positive scalar

Threshold to consider cells as occupied, specified as a positive scalar. Probability values above this threshold are considered occupied.

**ProbabilitySaturation — Saturation limits on probability values**`[0.001 0.999]` (default) | [`min` `max`] vector

Saturation limits on probability values, specified as a [`min` `max`] vector. Values above or below these saturation values are set to the `min` or `max` values. This property reduces oversaturating of cells when incorporating multiple observations.

## Methods

<code>checkOccupancy</code>	Check if locations are free or occupied
<code>getOccupancy</code>	Get occupancy probability of locations
<code>inflate</code>	Inflate map
<code>insertPointCloud</code>	Insert 3-D points or point cloud observation into map
<code>setOccupancy</code>	Set occupancy probability of locations
<code>show</code>	Show occupancy map
<code>updateOccupancy</code>	Update occupancy probability at locations

## Examples

**Create 3-D Occupancy Map and Inflate Points**

The `OccupancyMap3D` object stores obstacles in 3-D space, using sensor observations to map an environment. Create a map and add points from a point cloud to identify

obstacles. Then inflate the obstacles in the map to ensure safe operating space around obstacles.

Create an `OccupancyMap3D` object with a map resolution of 10 cells/meter.

```
map3D = robotics.OccupancyMap3D(10);
```

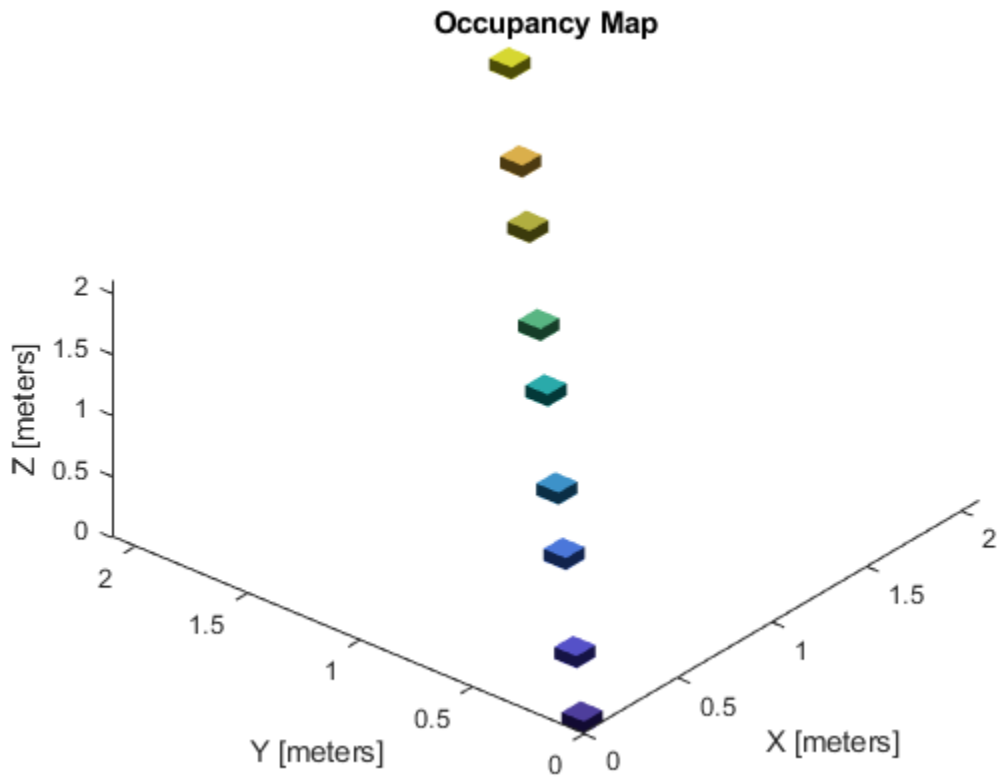
Define a set of 3-D points as an observation from a pose `[x y z qw qx qy qz]`. This pose is for the sensor that observes these points and is centered on the origin. Define two sets of points to insert multiple observations.

```
pose = [ 0 0 0 1 0 0 0];
```

```
points = repmat([0:0.25:2]', 1, 3);  
points2 = [(0:0.25:2)' (2:-0.25:0)' (0:0.25:2)'];  
maxRange = 5;
```

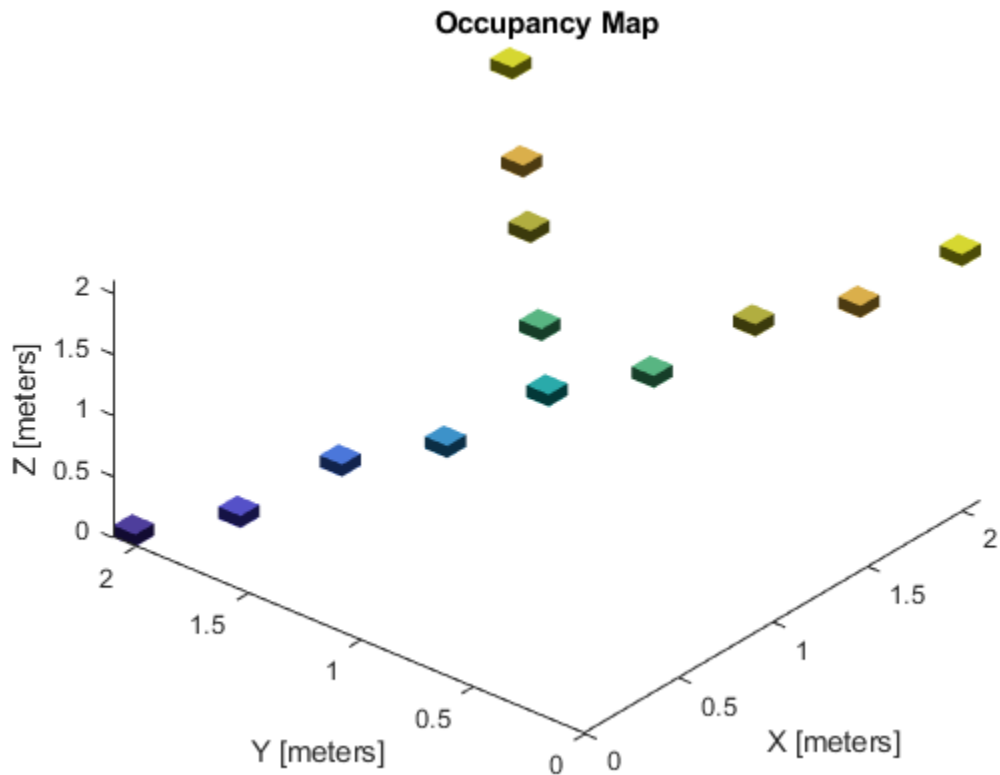
Insert the first set of points using `insertPointCloud`. The function uses the sensor pose and the given points to insert observations into the map. The colors displayed correlate to the height of the point only for illustrative purposes.

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



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

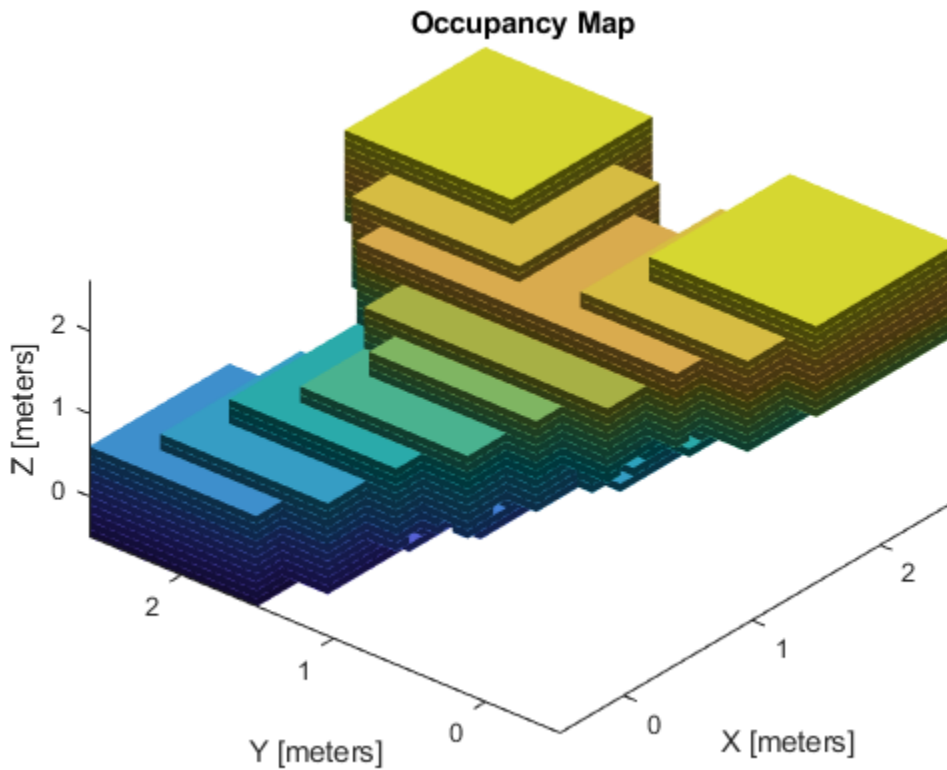
```
insertPointCloud(map3D,pose,points2,maxRange)  
show(map3D)
```



Inflate the map to add a buffer zone for safe operation around obstacles. Define the robot radius and safety distance and use the sum of these values to define the inflation radius for the map.

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

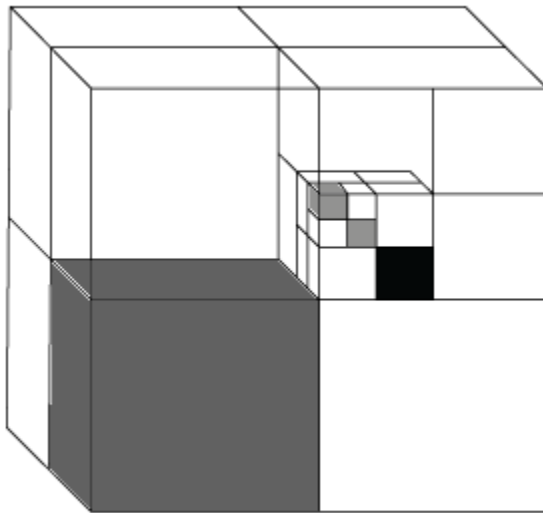




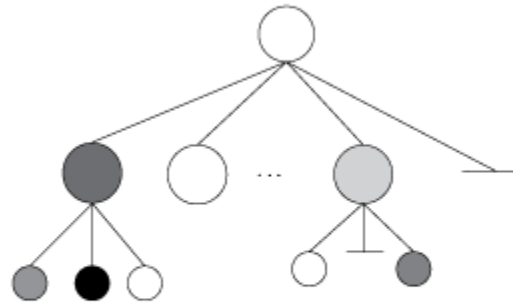
## Algorithms

### Octree Data Structure

The octree data structure is a hierarchical structure used for subdivision of an environment into cubic volumes called voxels. For a given map volume, the space is recursively subdivided into eight voxels until achieving a desired map resolution (voxel size) is achieved. This subdivision can be represented as a tree, which stores probability values for locations in the map.

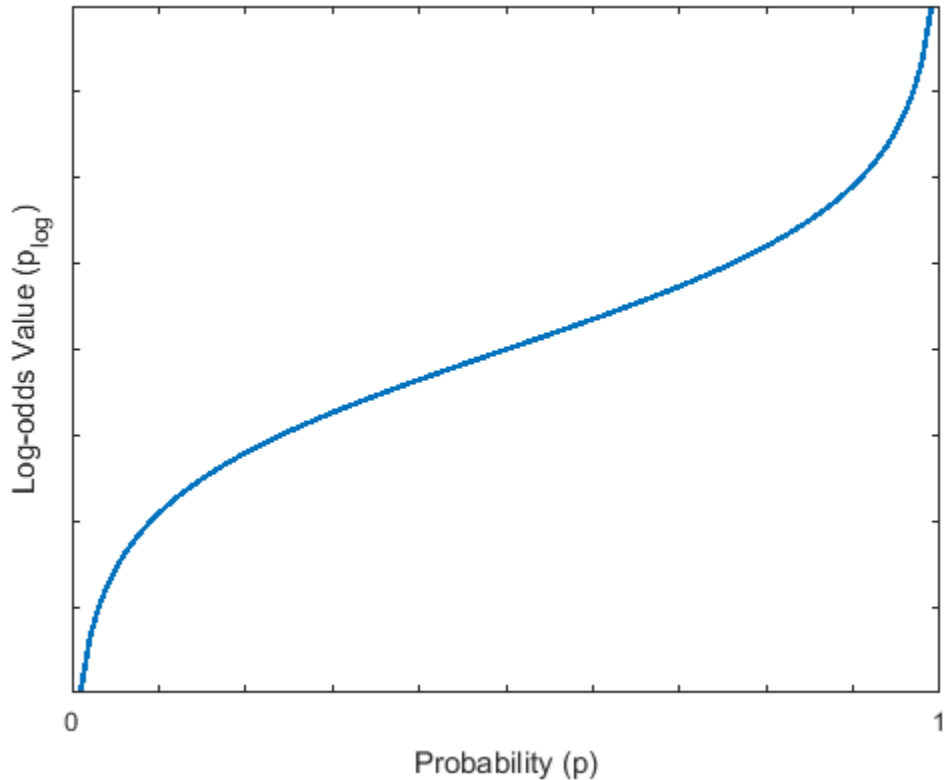


3-D Voxels



Octree Branching Structure

The probability values in the tree have a log-odds representation. Using this representation, locations easily recover from dynamic observations and numerical errors due to small probabilities are reduced. To remain efficient in memory, lower branches of the tree are pruned in the structure if they share the same occupancy values using this log-odds representation.



The class internally handles the organization of this data structure, including the pruning of branches. Specify all observations as spatial coordinates when using functions such as `setOccupancy`, `getOccupancy`, or `insertPointCloud`. Insertions into the tree, and navigation through the tree, is determined based on the spatial coordinates and the resolution of the map.

## References

- [1] Hornung, Armin, Kai M. Wurm, Maren Bennewitz, Cyrill Stachniss, and Wolfram Burgard. "OctoMap: an efficient probabilistic 3D mapping framework based on octrees." *Autonomous Robots*, Vol. 34, No. 3, 2013, pp. 189–206.. doi:10.1007/s10514-012-9321-0.

## See Also

### Classes

[BinaryOccupancyGrid](#) | [OccupancyGrid](#)

### Functions

[inflate](#) | [insertPointCloud](#) | [readOccupancyMap3D](#) | [setOccupancy](#) | [show](#)

**Introduced in R2018a**

# robotics.OdometryMotionModel class

**Package:** robotics

Create an odometry motion model

## Description

`OdometryMotionModel` creates an odometry motion model object for differential drive robots. This object contains specific motion model parameters. You can use this object to specify the motion model parameters in the `robotics.MonteCarloLocalization` object.

This motion model assumes that the robot makes pure rotation and translation motions to travel from one location to the other. The model propagates points for either forward or backwards motion based on these motion patterns. The elements of the `Noise` property refer to the variance in the motion. To see the effect of changing the noise parameters, use `robotics.OdometryMotionModel.showNoiseDistribution`.

## Construction

`omm = robotics.OdometryMotionModel` creates an odometry motion model object for differential drive robots.

## Properties

### Noise — Gaussian noise for robot motion

[0.2 0.2 0.2 0.2] (default) | 4-element vector

Gaussian noise for robot motion, specified as a 4-element vector. This property represents the variance parameters for Gaussian noise applied to robot motion. The elements of the vector correspond to the following errors in order:

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

- Translational error due to translation motion
- Translational error due to rotational motion

## Type — Type of the odometry motion model

'DifferentialDrive' (default)

This property is read-only.

Type of the odometry motion model, returned as 'DifferentialDrive'. This read-only property indicates the type of odometry motion model being used by the object.

## Examples

### Predict Poses Based On An Odometry Motion Model

This example shows how to use the `robotics.OdometryMotionModel` class to predict the pose of a robot. An `OdometryMotionModel` object contains the motion model parameters for a differential drive robot. Use the object to predict the pose of a robot based on its current and previous poses and the motion model parameters.

Create odometry motion model object.

```
motionModel = robotics.OdometryMotionModel;
```

Define previous poses and the current odometry reading. Each pose prediction corresponds to a row in `previousPoses` vector.

```
previousPoses = rand(10,3);  
currentOdom = [0.1 0.1 0.1];
```

The first call to the object initializes values and returns the previous poses as the current poses.

```
currentPoses = motionModel(previousPoses, currentOdom);
```

Subsequent calls to the object with updated odometry poses returns the predicted poses based on the motion model.

```
currentOdom = currentOdom + [0.1 0.1 0.05];  
predPoses = motionModel(previousPoses, currentOdom);
```

### **Show Noise Distribution Effects for Odometry Motion Model**

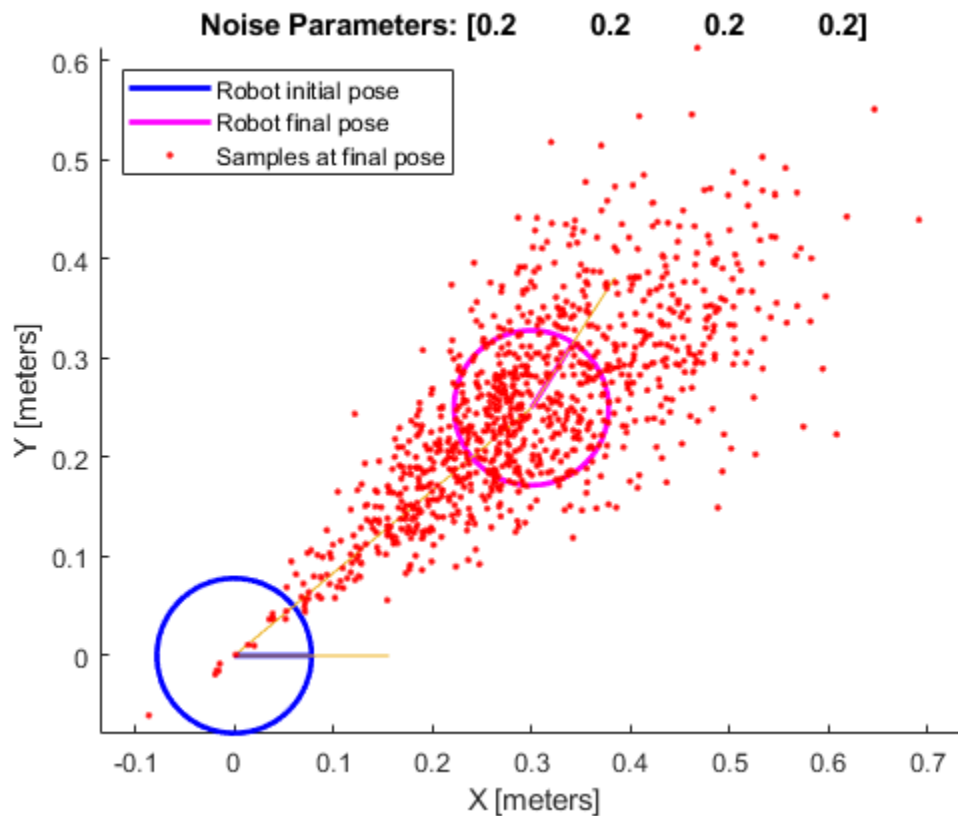
This example shows how to visualize the effect of different noise parameters on the `robotics.OdometryMotionModel` class. An `OdometryMotionModel` object contains the motion model noise parameters for a differential drive robot. Use `showNoiseDistribution` to visualize how changing these values affect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

Create a motion model object.

```
motionModel = robotics.OdometryMotionModel;
```

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

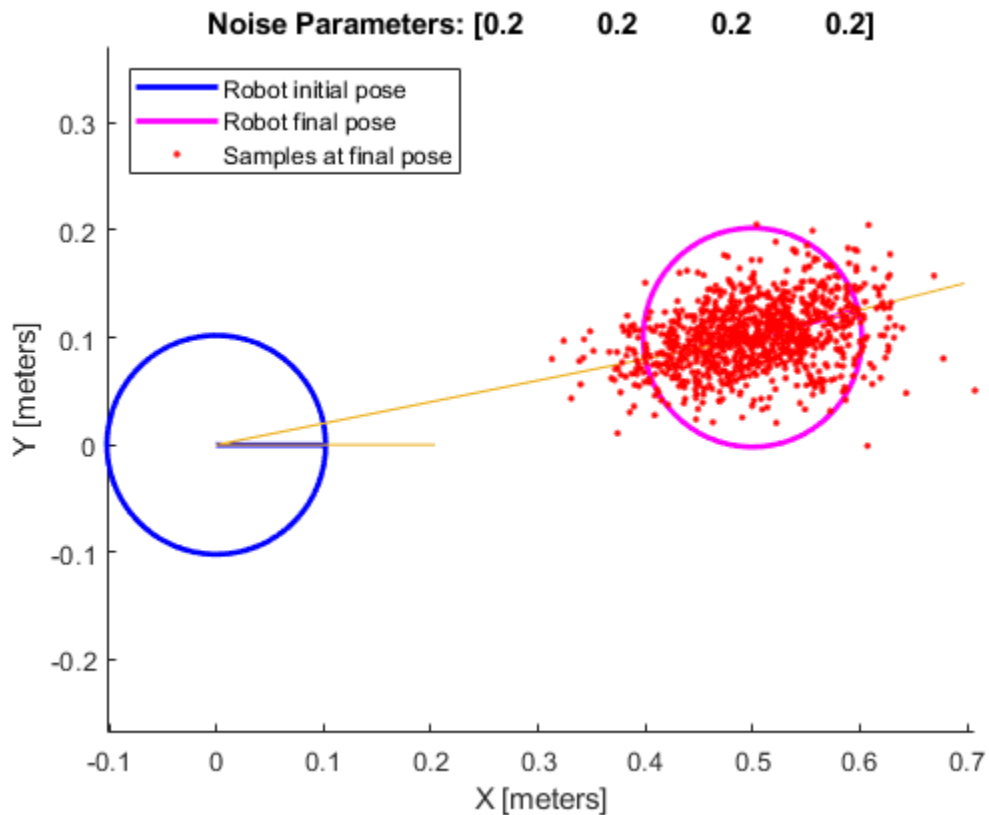
```
showNoiseDistribution(motionModel);
```



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

```
showNoiseDistribution(motionModel, ...  
                    'OdometryPoseChange', [0.5 0.1 0.25], ...  
                    'NumSamples', 1000);
```

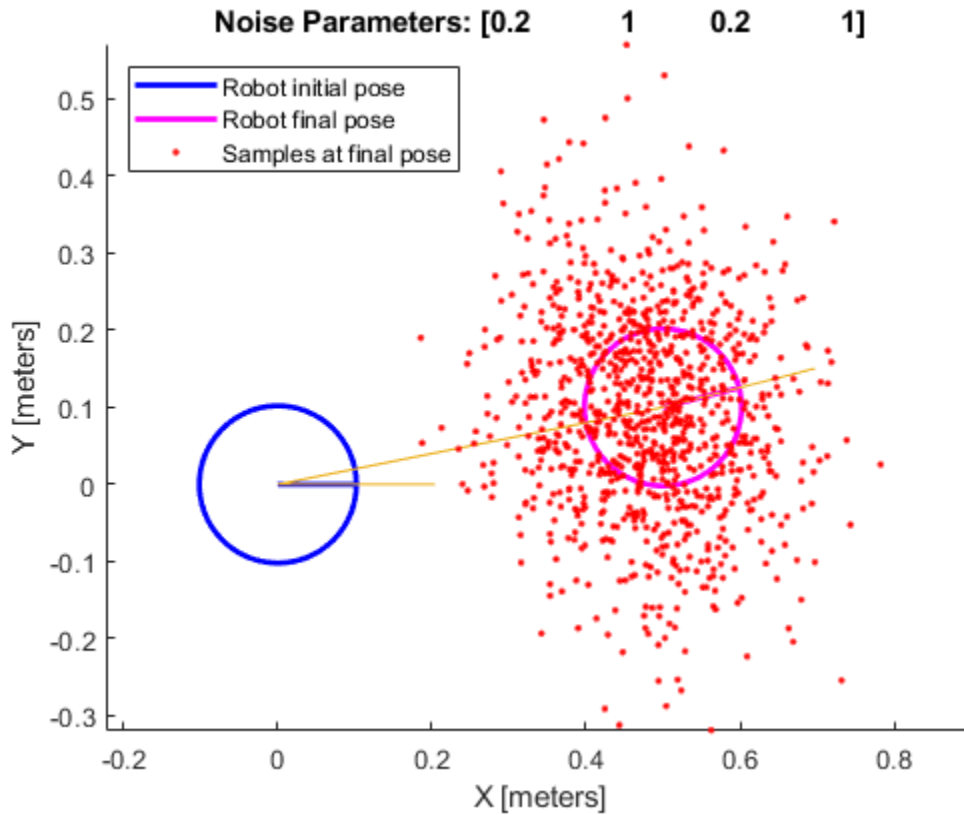




Change the Noise parameters and visualize the effects. Use the same odometry pose change and number of samples.

```
motionModel.Noise = [0.2 1 0.2 1];

showNoiseDistribution(motionModel, ...
    'OdometryPoseChange', [0.5 0.1 0.25], ...
    'NumSamples', 1000);
```



## Methods

showNoiseDistribution  
step

Display noise parameter effects  
Computer next pose from previous pose

## Limitations

If you make changes to your motion model after using it with the `MonteCarloLocalization` object, call `release` on that object beforehand. For example:

```
mcl = robotics.MonteCarloLocalization(...);  
[isUpdated,pose,covariance] = mcl(...);  
release(mcl)  
mcl.MotionModel.PropName = value;
```

## References

[1] Thrun, Sebastian, Wolfram Burgard, and Dieter Fox. *Probabilistic Robotics*. MIT Press, 2005.

## Extended Capabilities

### C/C++ Code Generation

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

### See Also

`robotics.LikelihoodFieldSensorModel` | `robotics.MonteCarloLocalization`

### Topics

“Localize TurtleBot Using Monte Carlo Localization”

**Introduced in R2016a**

## robotics.OrientationTarget class

**Package:** robotics

Create constraint on relative orientation of body

### Description

The `OrientationTarget` object describes a constraint that requires the orientation of one body (the end effector) to match a target orientation within an angular tolerance in any direction. The target orientation is specified relative to the body frame of the reference body.

Constraint objects are used in `GeneralizedInverseKinematics` objects to specify multiple kinematic constraints on a robot.

For an example that uses multiple constraint objects, see “Plan a Reaching Trajectory With Multiple Kinematic Constraints”.

### Construction

`orientationConst = robotics.OrientationTarget(endeffector)` returns an orientation target object that represents a constraint on a body of the robot model specified by `endeffector`.

`orientationConst = robotics.OrientationTarget(endeffector, Name, Value)` returns an orientation target object with each specified property name set to the specified value by one or more `Name, Value` pair arguments.

### Input Arguments

**endeffector** — End-effector name

string scalar | character vector

End-effector name, specified as a string scalar or character vector. When using this constraint with `GeneralizedInverseKinematics`, the name must match a body specified in the robot model (`RigidBodyTree`).

Example: "left\_palm"

Data Types: char | string

## Properties

### **EndEffector — Name of the end effector**

string scalar | character vector

Name of the end effector, specified as a string scalar or character vector. When using this constraint with `GeneralizedInverseKinematics`, the name must match a body specified in the robot model (`RigidBodyTree`).

Example: "left\_palm"

Data Types: char | string

### **ReferenceBody — Name of the reference body frame**

' ' (default) | string scalar | character vector

Name of the reference body frame, specified as a string scalar or character vector. The default ' ' indicates that the constraint is relative to the base of the robot model. When using this constraint with `GeneralizedInverseKinematics`, the name must match a body specified in the robot model (`RigidBodyTree`).

Data Types: char | string

### **TargetOrientation — Target orientation of the end effector relative to the reference body**

[1 0 0 0] (default) | four-element vector

Target orientation of the end effector relative to the reference body, specified as four-element vector that represents a unit quaternion. The orientation of the end effector relative to the reference body frame is the orientation that converts a direction specified in the end-effector frame to the same direction specified in the reference body frame.

### **OrientationTolerance — Maximum allowed rotation angle**

0 (default) | numeric scalar

Maximum allowed rotation angle in radians, specified as a numeric scalar. This value is the upper bound on the magnitude of the rotation required to make the end-effector orientation match the target orientation.

## **Weights — Weight of the constraint**

1 (default) | numeric scalar

Weight of the constraint, specified as a numeric scalar. This weight is used with the `Weights` property of all the constraints specified in `GeneralizedInverseKinematics` to properly balance each constraint.

## **Extended Capabilities**

### **C/C++ Code Generation**

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

### **See Also**

#### **Classes**

`robotics.GeneralizedInverseKinematics` | `robotics.JointPositionBounds` | `robotics.PoseTarget` | `robotics.PositionTarget`

#### **Topics**

“Plan a Reaching Trajectory With Multiple Kinematic Constraints”

**Introduced in R2017a**

# ParameterTree

Access ROS parameter server

## Description

A `ParameterTree` object communicates with the ROS parameter server. The ROS parameter server can store strings, integers, doubles, Booleans, and cell arrays. The parameters are accessible globally over the ROS network. You can use these parameters to store static data such as configuration parameters.

To directly set, get, or access ROS parameters without creating a `ParameterTree` object, see `rosparam`.

The following ROS data types are supported as values of parameters. For each ROS data type, the corresponding MATLAB data type is also listed.

ROS Data Type	MATLAB Data Type
32-bit integer	<code>int32</code>
boolean	<code>logical</code>
double	<code>double</code>
string	character vector ( <code>char</code> )
list	cell array ( <code>cell</code> )
dictionary	structure ( <code>struct</code> )

## Creation

## Syntax

```
ptree = rosparam
```

```
ptree = robotics.ros.ParameterTree(node)
```

## Description

`ptree = rosparam` creates a parameter tree object, `ptree`. After `ptree` is created, the connection to the parameter server remains persistent until the object is deleted or the ROS master becomes unavailable.

`ptree = robotics.ros.ParameterTree(node)` returns a `ParameterTree` object to communicate with the ROS parameter server. The parameter tree attaches to the ROS node, `node`. To connect to the global node, specify `node` as `[]`.

## Properties

### AvailableParameters — List of parameter names on the server

cell array

This property is read-only.

List of parameter names on the server, specified as a cell array.

Example: `{'/myParam'; '/robotSize'; '/hostname'}`

Data Types: `cell`

## Object Functions

<code>get</code>	Get ROS parameter value
<code>has</code>	Check if ROS parameter name exists
<code>search</code>	Search ROS network for parameter names
<code>set</code>	Set value of ROS parameter or add new parameter
<code>del</code>	Delete a ROS parameter

## Examples

### Create ROS ParameterTree Object and Modify Parameters

Start the ROS master and create a ROS node.

```
master = robotics.ros.Core;  
node = robotics.ros.Node('/test1');
```



Create the parameter tree object.

```
ptree = robotics.ros.ParameterTree(node);
```

Set multiple parameters.

```
set(ptree, 'DoubleParam', 1.0)
set(ptree, 'CharParam', 'test')
set(ptree, 'CellParam', {'test'}, {1,2});
```

View the available parameters.

```
parameters = ptree.AvailableParameters
```

```
parameters = 3x1 cell array
    {'/CellParam' }
    {'/CharParam' }
    {'/DoubleParam'}
```

Get a parameter value.

```
data = get(ptree, 'CellParam')
```

```
data = 1x2 cell array
    {1x1 cell}    {1x2 cell}
```

Search for a parameter name.

```
search(ptree, 'char')
```

```
ans = 1x1 cell array
    {'/CharParam'}
```

Delete the parameter tree and ROS node. Shut down the ROS master.

```
clear('ptree', 'node')
clear('master')
```

## Set A Dictionary Of Parameter Values

Use structures to specify a dictionary of ROS parameters under a specific namespace.

Connect to a ROS network.

```
roslint
```

```
Initializing ROS master on http://bat5742win64:64232/.
```

```
Initializing global node /matlab_global_node_67361 with NodeURI http://bat5742win64:64232/.
```

Create a dictionary of parameters values. This dictionary contains the information relevant to an image. Display the structure to verify values.

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

```
pval.ImageWidth = size(image,1);  
pval.ImageHeight = size(image,2);  
pval.ImageTitle = 'peppers.png';  
disp(pval)
```

```
    ImageWidth: 384  
    ImageHeight: 512  
    ImageTitle: 'peppers.png'
```

Set the dictionary of values using the desired namespace.

```
rosparam('set', 'ImageParam', pval)
```

Get the parameters using the namespace. Verify the values.

```
pval2 = rosparam('get', 'ImageParam')
```

```
pval2 = struct with fields:  
    ImageHeight: 512  
    ImageTitle: 'peppers.png'  
    ImageWidth: 384
```

Shutdown ROS network.

```
roshutdn
```

```
Shutting down global node /matlab_global_node_67361 with NodeURI http://bat5742win64:64232/.
```

```
Shutting down ROS master on http://bat5742win64:64232/.
```

## See Also

[del](#) | [get](#) | [has](#) | [rosparam](#) | [search](#) | [set](#)

## **Topics**

“Access the ROS Parameter Server”

**Introduced in R2015a**

## robotics.ParticleFilter class

**Package:** robotics

Create particle filter state estimator

### Description

The particle filter is a recursive, Bayesian state estimator that uses discrete particles to approximate the posterior distribution of the estimated state.

The particle filter algorithm computes the state estimate recursively and involves two steps: prediction and correction. The prediction step uses the previous state to predict the current state based on a given system model. The correction step uses the current sensor measurement to correct the state estimate. The algorithm periodically redistributes, or resamples, the particles in the state space to match the posterior distribution of the estimated state.

The estimated state consists of state variables. Each particle represents a discrete state hypothesis of these state variables. The set of all particles is used to help determine the final state estimate.

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

For more information on the particle filter workflow and setting specific parameters, see:

- “Particle Filter Workflow”
- “Particle Filter Parameters”

### Construction

`pf = robotics.ParticleFilter` creates a `ParticleFilter` object that enables the state estimation for a simple system with three state variables. Use the `initialize` method to initialize the particles with a known mean and covariance or uniformly distributed particles within defined bounds. To customize the particle filter’s system and

measurement models, modify the `StateTransitionFcn` and `MeasurementLikelihoodFcn` properties.

After you create the `ParticleFilter` object, use `robotics.ParticleFilter.initialize` to initialize the `NumStateVariables` and `NumParticles` properties. The `initialize` function sets these two properties based on your inputs.

## Properties

### **NumStateVariables — Number of state variables**

3 (default) | scalar

This property is read-only.

Number of state variables, specified as a scalar. This property is set based on the inputs to the `initialize` method. The number of states is implicit based on the specified matrices for initial state and covariance.

### **NumParticles — Number of particles used in the filter**

1000 (default) | scalar

This property is read-only.

Number of particles using in the filter, specified as a scalar. You can specify this property only by calling the `initialize` method.

### **StateTransitionFcn — Callback function for determining the state transition between particle filter steps**

function handle

Callback function for determining the state transition between particle filter steps, specified as a function handle. The state transition function evolves the system state for each particle. The function signature is:

```
function predictParticles = stateTransitionFcn(pf,prevParticles,varargin)
```

The callback function accepts at least two input arguments: the `ParticleFilter` object, `pf`, and the particles at the previous time step, `prevParticles`. These specified particles are the `predictParticles` returned from the previous `step` call of the

ParticleFilter object. predictParticles and prevParticles are the same size: NumParticles-by-NumStateVariables.

You can also use varargin to pass in a variable number of arguments from the predict function. When you call:

```
predict(pf, arg1, arg2)
```

MATLAB essentially calls stateTransitionFcn as:

```
stateTransitionFcn(pf, prevParticles, arg1, arg2)
```

### **MeasurementLikelihoodFcn — Callback function calculating the likelihood of sensor measurements**

function handle

Callback function calculating the likelihood of sensor measurements, specified as a function handle. Once a sensor measurement is available, this callback function calculates the likelihood that the measurement is consistent with the state hypothesis of each particle. You must implement this function based on your measurement model. The function signature is:

```
function likelihood = measurementLikelihoodFcn(PF, predictParticles, measurement, varargin)
```

The callback function accepts at least three input arguments:

- 1** pf - The associated ParticleFilter object
- 2** predictParticles - The particles that represent the predicted system state at the current time step as an array of size NumParticles-by-NumStateVariables
- 3** measurement - The state measurement at the current time step

You can also use varargin to pass in a variable number of arguments. These arguments are passed by the correct function. When you call:

```
correct(pf, measurement, arg1, arg2)
```

MATLAB essentially calls measurementLikelihoodFcn as:

```
measurementLikelihoodFcn(pf, predictParticles, measurement, arg1, arg2)
```

The callback needs to return exactly one output, likelihood, which is the likelihood of the given measurement for each particle state hypothesis.

**IsStateVariableCircular** — Indicator if state variables have a circular distribution`[0 0 0]` (default) | logical array

Indicator if state variables have a circular distribution, specified as a logical array. Circular (or angular) distributions use a probability density function with a range of  $[-\pi, \pi]$ . If the `ParticleFilter` object has multiple state variables, then `IsStateVariableCircular` is a row vector. Each vector element indicates if the associated state variable is circular. If the object has only one state variable, then `IsStateVariableCircular` is a scalar.

**ResamplingPolicy** — Policy settings that determine when to trigger resampling object

Policy settings that determine when to trigger resampling, specified as an object. You can trigger resampling either at fixed intervals, or you can trigger it dynamically, based on the number of effective particles. See `robotics.ResamplingPolicy` for more information.

**ResamplingMethod** — Method used for particle resampling`'multinomial'` (default) | `'residual'` | `'stratified'` | `'systematic'`

Method used for particle resampling, specified as `'multinomial'`, `'residual'`, `'stratified'`, and `'systematic'`.

**StateEstimationMethod** — Method used for state estimation`'mean'` (default) | `'maxweight'`

Method used for state estimation, specified as `'mean'` and `'maxweight'`.

**Particles** — Array of particle values`NumParticles-by-NumStateVariables` matrix

Array of particle values, specified as a `NumParticles-by-NumStateVariables` matrix. Each row corresponds to the state hypothesis of a single particle.

**Weights** — Particle weights`NumParticles-by-1` vector

Particle weights, specified as a `NumParticles-by-1` vector. Each weight is associated with the particle in the same row in the `Particles` property.

**State** — Best state estimate

vector

This property is read-only.

Best state estimate, returned as a vector with length `NumStateVariables`. The estimate is extracted based on the `StateEstimationMethod` property.

### **State Covariance — Corrected system covariance**

*N*-by-*N* matrix | []

This property is read-only.

Corrected system variance, returned as an *N*-by-*N* matrix, where *N* is equal to the `NumStateVariables` property. The corrected state is calculated based on the `StateEstimationMethod` property and the `MeasurementLikelihoodFcn`. If you specify a state estimate method that does not support covariance, then the property is set to [].

## **Methods**

<code>copy</code>	Create copy of particle filter
<code>correct</code>	Adjust state estimate based on sensor measurement
<code>getStateEstimate</code>	Extract best state estimate and covariance from particles
<code>initialize</code>	Initialize the state of the particle filter
<code>predict</code>	Predict state of robot in next time step

## **Examples**

### **Particle Filter Prediction and Correction**

Create a `ParticleFilter` object, and execute a prediction and correction step for state estimation. The particle filter gives a predicted state estimate based on the return value of `StateTransitionFcn`. It then corrects the state based on a given measurement and the return value of `MeasurementLikelihoodFcn`.

Create a particle filter with the default three states.

```
pf = robotics.ParticleFilter
```



```

pf =
  ParticleFilter with properties:

      NumStateVariables: 3
      NumParticles: 1000
      StateTransitionFcn: @robotics.algs.gaussianMotion
      MeasurementLikelihoodFcn: @robotics.algs.fullStateMeasurement
      IsStateVariableCircular: [0 0 0]
      ResamplingPolicy: [1x1 robotics.ResamplingPolicy]
      ResamplingMethod: 'multinomial'
      StateEstimationMethod: 'mean'
      StateOrientation: 'row'
      Particles: [1000x3 double]
      Weights: [1000x1 double]
      State: 'Use the getStateEstimate function to see the value.'
      StateCovariance: 'Use the getStateEstimate function to see the value.'

```

Specify the mean state estimation method and systematic resampling method.

```

pf.StateEstimationMethod = 'mean';
pf.ResamplingMethod = 'systematic';

```

Initialize the particle filter at state [4 1 9] with unit covariance (`eye(3)`). Use 5000 particles.

```

initialize(pf,5000,[4 1 9],eye(3));

```

Assuming a measurement [4.2 0.9 9], run one predict and one correct step.

```

[statePredicted,stateCov] = predict(pf);
[stateCorrected,stateCov] = correct(pf,[4.2 0.9 9]);

```

Get the best state estimate based on the StateEstimationMethod algorithm.

```

stateEst = getStateEstimate(pf)

```

```

stateEst = 1x3

```

```

    4.1562    0.9185    9.0202

```

## Estimate Robot Position in a Loop Using Particle Filter

Use the `ParticleFilter` object to track a robot as it moves in a 2-D space. The measured position has random noise added. Using `predict` and `correct`, track the robot based on the measurement and on an assumed motion model.

Initialize the particle filter and specify the default state transition function, the measurement likelihood function, and the resampling policy.

```
pf = robotics.ParticleFilter;  
pf.StateEstimationMethod = 'mean';  
pf.ResamplingMethod = 'systematic';
```

Sample 1000 particles with an initial position of [0 0] and unit covariance.

```
initialize(pf,1000,[0 0],eye(2));
```

Prior to estimation, define a sine wave path for the dot to follow. Create an array to store the predicted and estimated position. Define the amplitude of noise.

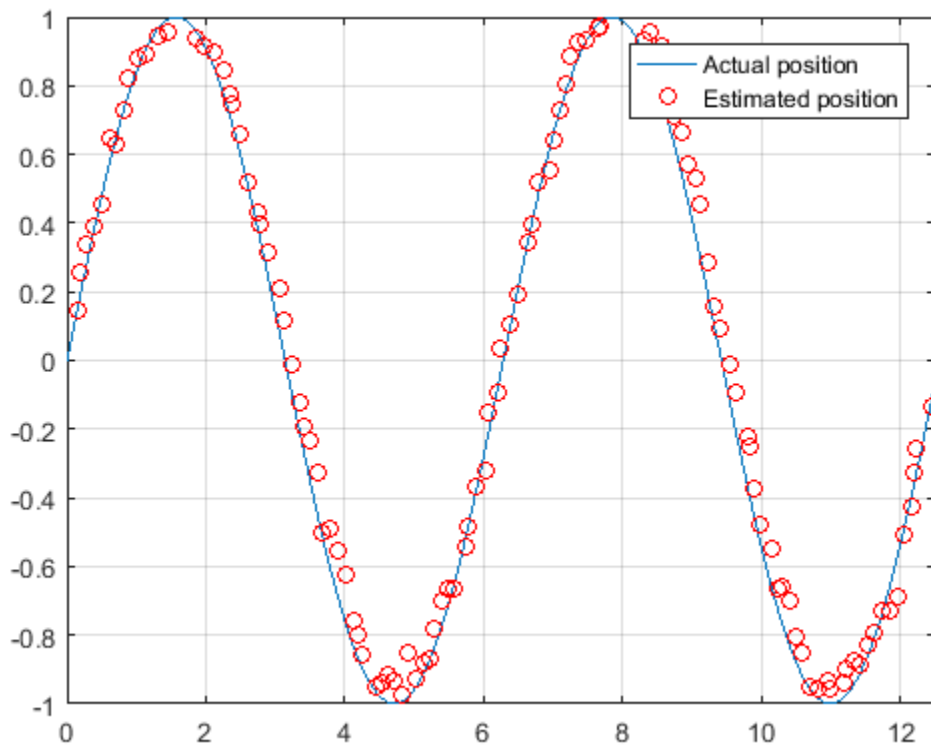
```
t = 0:0.1:4*pi;  
dot = [t; sin(t)]';  
robotPred = zeros(length(t),2);  
robotCorrected = zeros(length(t),2);  
noise = 0.1;
```

Begin the loop for predicting and correcting the estimated position based on measurements. The resampling of particles occurs based on the `ResamplingPolicy` property. The robot moves based on a sine wave function with random noise added to the measurement.

```
for i = 1:length(t)  
    % Predict next position. Resample particles if necessary.  
    [robotPred(i,:),robotCov] = predict(pf);  
    % Generate dot measurement with random noise. This is  
    % equivalent to the observation step.  
    measurement(i,:) = dot(i,:) + noise*(rand([1 2])-noise/2);  
    % Correct position based on the given measurement to get best estimation.  
    % Actual dot position is not used. Store corrected position in data array.  
    [robotCorrected(i,:),robotCov] = correct(pf,measurement(i,:));  
end
```

Plot the actual path versus the estimated position. Actual results may vary due to the randomness of particle distributions.

```
plot(dot(:,1),dot(:,2),robotCorrected(:,1),robotCorrected(:,2),'or')
xlim([0 t(end)])
ylim([-1 1])
legend('Actual position','Estimated position')
grid on
```



The figure shows how close the estimate state matches the actual position of the robot. Try tuning the number of particles or specifying a different initial position and covariance to see how it affects tracking over time.

## References

- [1] Arulampalam, M.S., S. Maskell, N. Gordon, and T. Clapp. "A Tutorial on Particle Filters for Online Nonlinear/Non-Gaussian Bayesian Tracking." *IEEE Transactions on Signal Processing*. Vol. 50, No. 2, Feb 2002, pp. 174-188.
- [2] Chen, Z. "Bayesian Filtering: From Kalman Filters to Particle Filters, and Beyond." *Statistics*. Vol. 182, No. 1, 2003, pp. 1-69.

## Extended Capabilities

### C/C++ Code Generation

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

### See Also

`robotics.ParticleFilter.correct` | `robotics.ParticleFilter.predict` | `robotics.ResamplingPolicy`

### Topics

"Track a Car-Like Robot Using Particle Filter"

"Particle Filter Parameters"

"Particle Filter Workflow"

Class Attributes (MATLAB)

Property Attributes (MATLAB)

**Introduced in R2016a**

# robotics.PoseGraph

Create 2-D pose graph

## Description

A `PoseGraph` object stores information for a 2-D pose graph representation. A pose graph contains `nodes` connected by `edges`, with edge constraints that define the relative pose between nodes and the uncertainty on that measurement. The `optimizePoseGraph` function modifies the nodes to account for the uncertainty and improve the overall graph.

For 3-D pose graphs, see `PoseGraph3D`.

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

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

## Creation

## Syntax

```
poseGraph = robotics.PoseGraph
poseGraph =
robotics.PoseGraph('MaxNumEdges',maxEdges,'MaxNumNodes',maxNodes)
```

## Description

`poseGraph = robotics.PoseGraph` creates a 2-D pose graph object. Add poses using `addRelativePose` to construct a pose graph iteratively.

`poseGraph = robotics.PoseGraph('MaxNumEdges', maxEdges, 'MaxNumNodes', maxNodes)` specifies an upper bound on the number of edges and nodes allowed in the pose graph when generating code. This limit is only required when generating code.

## Properties

### **NumNodes — Number of nodes in pose graph**

1 (default) | positive integer

This property is read-only.

Number of nodes in pose graph, specified as a positive integer. Each node represents a pose in the pose graph as an [x y theta] vector with an xy-position and orientation angle, theta. To specify relative poses between nodes, use `addRelativePose`. To get a list of all nodes, use `nodes`.

### **NumEdges — Number of edges in pose graph**

0 (default) | nonnegative integer

This property is read-only.

Number of edges in pose graph, specified as a nonnegative integer. Each edge connects two nodes in the pose graph. Loop closure edges are included.

### **NumLoopClosureEdges — Number of loop closures**

0 (default) | nonnegative integer

This property is read-only.

Number of loop closures in pose graph, specified as a nonnegative integer. To get the edge IDs of the loop closures, use the `LoopClosureEdgeIDs` property.

### **LoopClosureEdgeIDs — Loop closure edge IDs**

vector

This property is read-only.

Loop closure edges IDs, specified as a vector of edge IDs.

## Object Functions

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

## Examples

### Optimize a 2-D Pose Graph

Optimize a pose graph based on the nodes and edge constraints. The pose graph used in this example is from the Intel Research Lab Dataset and was generated from collecting wheel odometry and a laser range finder sensor information in an indoor lab.

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

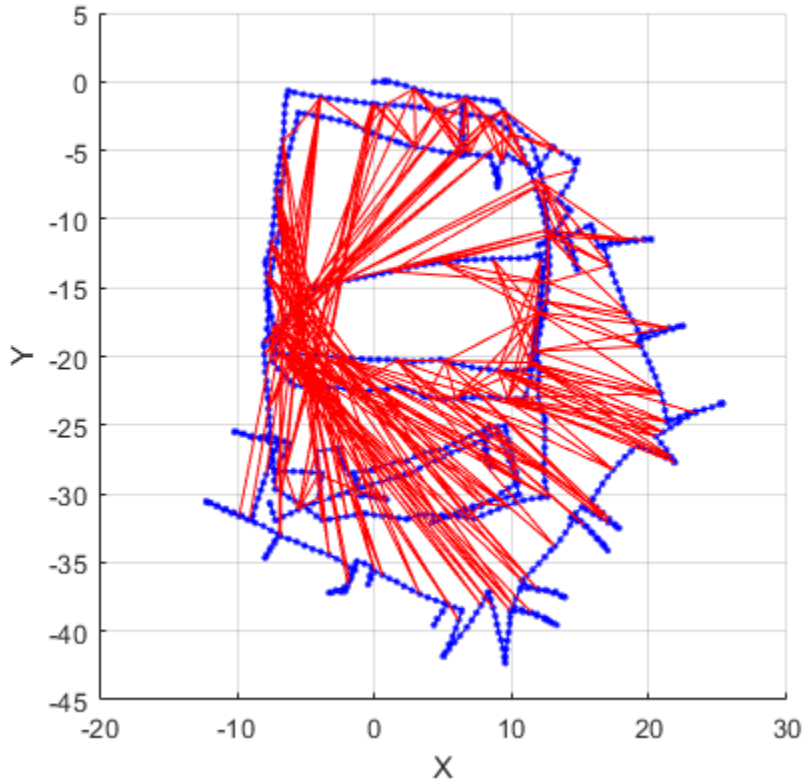
```
load intel-2d-posegraph.mat pg
disp(pg)
```

```
PoseGraph with properties:
```

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

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

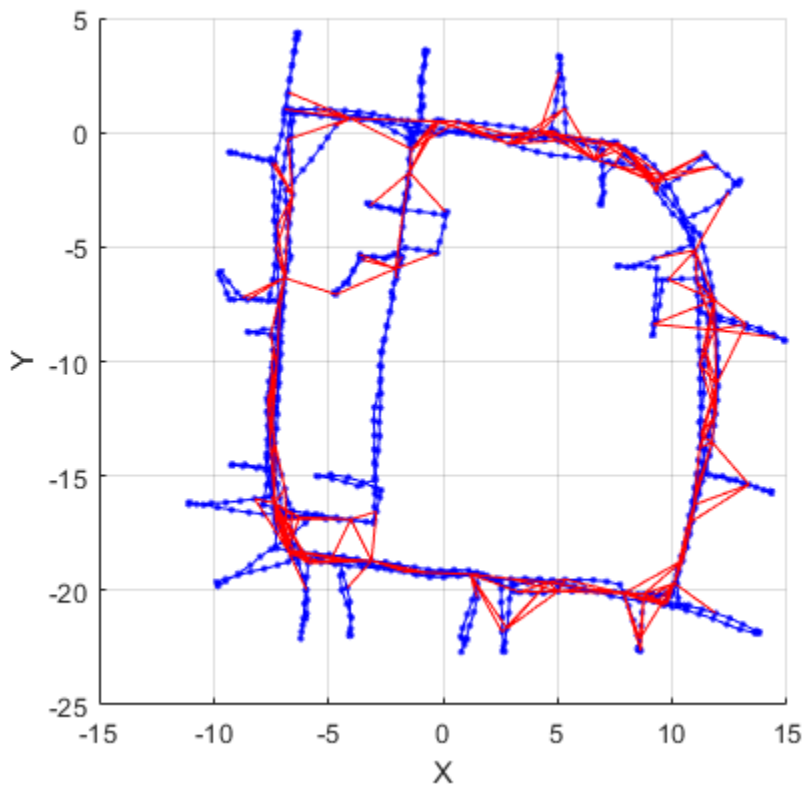
```
title('Original Pose Graph')
show(pg, 'IDs', 'off');
```



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

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





## References

- [1] Grisetti, G., R. Kummerle, C. Stachniss, and W. Burgard. "A Tutorial on Graph-Based SLAM." *IEEE Intelligent Transportation Systems Magazine*. Vol. 2, No. 4, 2010, pp. 31-43. doi:10.1109/mits.2010.939925.

## Extended Capabilities

### C/C++ Code Generation

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

Use this syntax when constructing PoseGraph objects for code generation:

```
poseGraph =  
robotics.PoseGraph( 'MaxNumEdges',maxEdges, 'MaxNumNodes',maxNodes)  
specifies an upper bound on the number of edges and nodes allowed in the pose graph  
when generating code. This limit is only required when generating code.
```

### See Also

#### Functions

`addRelativePose` | `optimizePoseGraph` | `show`

#### Objects

`robotics.LidarSLAM` | `robotics.PoseGraph3D`

### Topics

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

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

**Introduced in R2018a**

# robotics.PoseGraph3D

Create 3-D pose graph

## Description

A PoseGraph3D object stores information for a 3-D pose graph representation. A pose graph contains nodes connected by edges, with edge constraints that define the relative pose between nodes and the uncertainty on that measurement. The optimizePoseGraph function modifies the nodes to account for the uncertainty and improve the overall graph.

For 2-D pose graphs, see PoseGraph.

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

## Creation

## Syntax

```
poseGraph = robotics.PoseGraph3D
poseGraph =
robotics.PoseGraph3D('MaxNumEdges',maxEdges,'MaxNumNodes',maxNodes)
```

## Description

poseGraph = robotics.PoseGraph3D creates a 3-D pose graph object. Add poses using addRelativePose to construct a pose graph iteratively.

```
poseGraph =
robotics.PoseGraph3D('MaxNumEdges',maxEdges,'MaxNumNodes',maxNodes)
```

specifies an upper bound on the number of edges and nodes allowed in the pose graph when generating code. This limit is only required when generating code.

## Properties

### **NumNodes — Number of nodes in pose graph**

1 (default) | positive integer

This property is read-only.

Number of nodes in pose graph, specified as a positive integer. Each node represents a pose in the pose graph as an  $[x \ y \ z \ qw \ qx \ qy \ qz]$  vector with an xyz-position and quaternion orientation,  $[qw \ qx \ qy \ qz]$ . To specify relative poses between nodes, use `addRelativePose`. To get a list of all nodes, use `nodes`.

---

**Note** The order of the quaternion  $[qw \ qx \ qy \ qz]$  uses the standard convention. Some robot coordinate systems instead specify the order as  $[qx \ qy \ qz \ qw]$ . Check the source of your pose graph data before adding nodes to your `PoseGraph3D` object.

---

### **NumEdges — Number of edges in pose graph**

0 (default) | nonnegative integer

This property is read-only.

Number of edges in pose graph, specified as a nonnegative integer. Each edge connects two nodes in the pose graph. Loop closure edges are included.

### **NumLoopClosureEdges — Number of loop closures**

0 (default) | nonnegative integer

This property is read-only.

Number of loop closures in pose graph, specified as a nonnegative integer. To get the edge IDs of the loop closures, use the `LoopClosureEdgeIDs` property.

### **LoopClosureEdgeIDs — Loop closure edge IDs**

vector

This property is read-only.

Loop closure edges IDs, specified as a vector of edge IDs.

## Object Functions

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

## Examples

### Optimize a 3-D Pose Graph

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

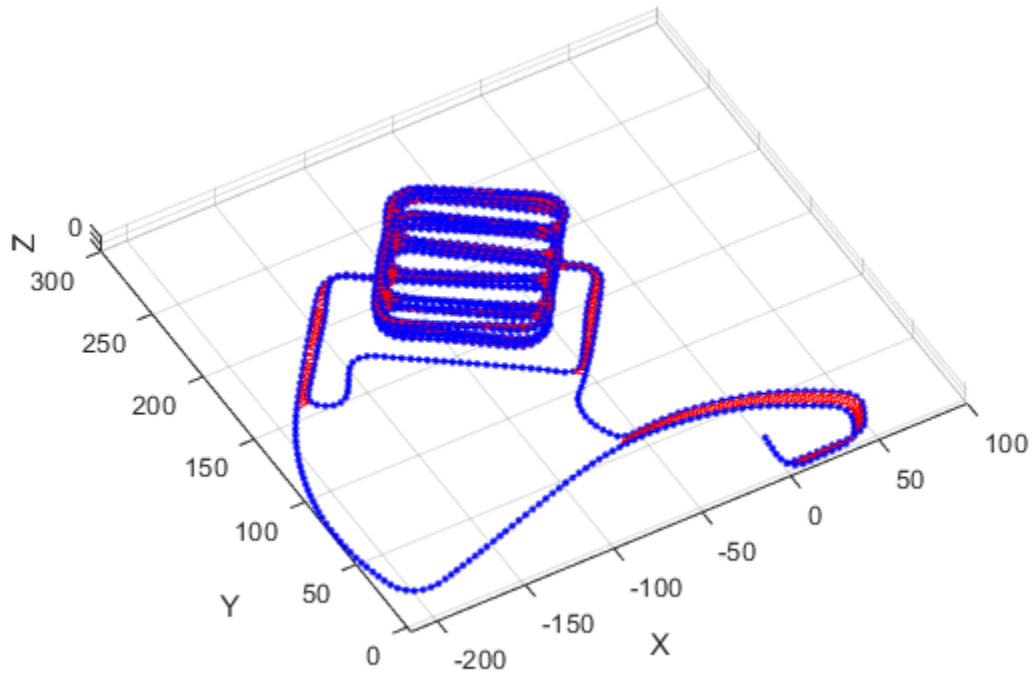
Load the pose graph from the MIT dataset. Inspect the `robotics.PoseGraph3D` object to view the number of nodes and loop closures.

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

PoseGraph3D with properties:
    NumNodes: 1661
    NumEdges: 6275
    NumLoopClosureEdges: 4615
    LoopClosureEdgeIDs: [1x4615 double]
```

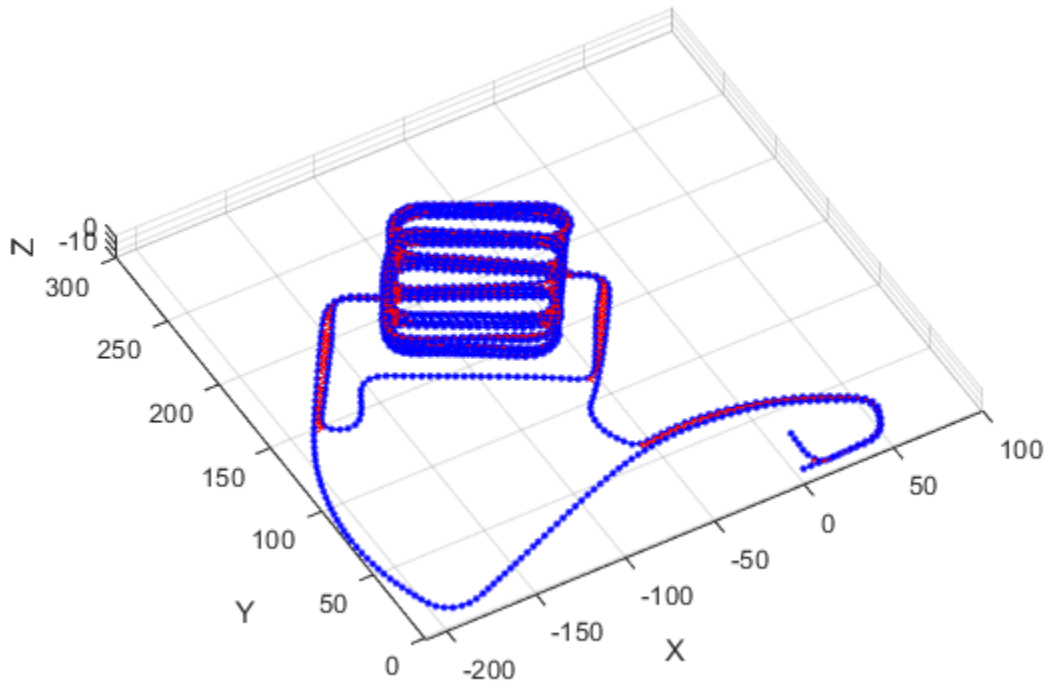
Plot the pose graph with IDs off. Red lines indicate loop closures identified in the dataset.

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



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

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



## References

- [1] Carlone, Luca, Roberto Tron, Kostas Daniilidis, and Frank Dellaert. "Initialization Techniques for 3D SLAM: a Survey on Rotation Estimation and its Use in Pose Graph Optimization." *2015 IEEE International Conference on Robotics and Automation (ICRA)*. 2015, pp. 4597-4604.

## Extended Capabilities

### C/C++ Code Generation

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

Use this syntax when constructing PoseGraph3D objects for code generation:

```
poseGraph =  
robotics.PoseGraph3D('MaxNumEdges',maxEdges,'MaxNumNodes',maxNodes)
```

specifies an upper bound on the number of edges and nodes allowed in the pose graph when generating code. This limit is only required when generating code.

### See Also

#### Functions

`addRelativePose` | `optimizePoseGraph`

#### Objects

`robotics.LidarSLAM` | `robotics.PoseGraph`

**Introduced in R2018a**



# robotics.PoseTarget class

**Package:** robotics

Create constraint on relative pose of body

## Description

The `PoseTarget` object describes a constraint that requires the pose of one body (the end effector) to match a target pose within a distance and angular tolerance in any direction. The target pose is specified relative to the body frame of the reference body.

Constraint objects are used in `GeneralizedInverseKinematics` objects to specify multiple kinematic constraints on a robot.

For an example that uses multiple constraint objects, see “Plan a Reaching Trajectory With Multiple Kinematic Constraints”.

## Construction

`poseConst = robotics.PoseTarget(endeffector)` returns a pose target object that represents a constraint on the body of the robot model specified by `endeffector`.

`poseConst = robotics.PoseTarget(endeffector, Name, Value)` returns a pose target object with each specified property name set to the specified value by one or more `Name, Value` pair arguments.

## Input Arguments

### **endeffector** — End-effector name

string scalar | character vector

End-effector name, specified as a string scalar or character vector. When using this constraint with `GeneralizedInverseKinematics`, the name must match a body specified in the robot model (`RigidBodyTree`).

Example: "left\_palm"

Data Types: char | string

## Properties

### **EndEffector — Name of the end effector**

string scalar | character vector

Name of the end effector, specified as a string scalar or character vector. When using this constraint with `GeneralizedInverseKinematics`, the name must match a body specified in the robot model (`RigidBodyTree`).

Example: "left\_palm"

Data Types: char | string

### **ReferenceBody — Name of the reference body frame**

' ' (default) | string scalar | character vector

Name of the reference body frame, specified as a string scalar or character vector. The default ' ' indicates that the constraint is relative to the base of the robot model. When using this constraint with `GeneralizedInverseKinematics`, the name must match a body specified in the robot model (`RigidBodyTree`).

Data Types: char | string

### **TargetTransform — Pose of the target frame relative to the reference body**

eye(4) (default) | matrix

Pose of the target frame relative to the reference body, specified as a matrix. The matrix is a homogeneous transform that specifies the relative transformation to convert a point in the target frame to the reference body frame.

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

### **OrientationTolerance — Maximum allowed rotation angle**

0 (default) | numeric scalar

Maximum allowed rotation angle in radians, specified as a numeric scalar. This value is the upper bound on the magnitude of the rotation required to make the end-effector orientation match the target orientation.

### **PositionTolerance — Maximum allowed distance from target**

0 (default) | numeric scalar in meters

Maximum allowed distance from target, specified as a numeric scalar in meters. This value is the upper bound on the distance between the end-effector origin and the target position.

**Weights — Weights of the constraint**

[1 1] (default) | two-element vector

Weights of the constraint, specified as a two-element vector. Each element of the vector corresponds to the weight for the `PositionTolerance` and `OrientationTolerance` respectively. These weights are used with the `Weights` of all the constraints specified in `GeneralizedInverseKinematics` to properly balance each constraint.

## Extended Capabilities

### C/C++ Code Generation

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

## See Also

**Classes**

`robotics.CartesianBounds` | `robotics.GeneralizedInverseKinematics` | `robotics.OrientationTarget` | `robotics.PositionTarget`

**Topics**

“Plan a Reaching Trajectory With Multiple Kinematic Constraints”

**Introduced in R2017a**

## robotics.PositionTarget class

**Package:** robotics

Create constraint on relative position of body

### Description

The `PositionTarget` object describes a constraint that requires the position of one body (the end effector) to match a target position within a distance tolerance in any direction. The target position is specified relative to the body frame of the reference body.

Constraint objects are used in `GeneralizedInverseKinematics` objects to specify multiple kinematic constraints on a robot.

For an example that uses multiple constraint objects, see “Plan a Reaching Trajectory With Multiple Kinematic Constraints”.

### Construction

`positionConst = robotics.PositionTarget(endeffector)` returns a position target object that represents a constraint on the body of the robot model specified by `endeffector`.

`positionConst = robotics.PositionTarget(endeffector, Name, Value)` returns a position target object with each specified property name set to the specified value by one or more `Name, Value` pair arguments.

### Input Arguments

**endeffector — End-effector name**

string scalar | character vector

End-effector name, specified as a string scalar or character vector. When using this constraint with `GeneralizedInverseKinematics`, the name must match a body specified in the robot model (`RigidBodyTree`).

Example: "left\_palm"

Data Types: char | string

## Properties

### **EndEffector — Name of the end effector**

string scalar | character vector

Name of the end effector, specified as a string scalar or character vector. When using this constraint with `GeneralizedInverseKinematics`, the name must match a body specified in the robot model (`RigidBodyTree`).

Example: "left\_palm"

Data Types: char | string

### **ReferenceBody — Name of the reference body frame**

' ' (default) | character vector

Name of the reference body frame, specified as a character vector. The default ' ' indicates that the constraint is relative to the base of the robot model. When using this constraint with `GeneralizedInverseKinematics`, the name must match a body specified in the robot model (`RigidBodyTree`).

### **TargetPosition — Position of the target relative to the reference body**

[0 0 0] (default) | [x y z] vector

Position of the target relative to the reference body, specified as an [x y z] vector. The target position is a point specified in the reference body frame.

### **PositionTolerance — Maximum allowed distance from target**

0 (default) | numeric scalar

Maximum allowed distance from target in meters, specified as a numeric scalar. This value is the upper bound on the distance between the end-effector origin and the target position.

### **Weights — Weight of the constraint**

1 (default) | numeric scalar

Weight of the constraint, specified as a numeric scalar. This weight is used with the `Weights` property of all the constraints specified in `GeneralizedInverseKinematics` to properly balance each constraint.

## Extended Capabilities

### C/C++ Code Generation

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

### See Also

#### Classes

`robotics.CartesianBounds` | `robotics.GeneralizedInverseKinematics` | `robotics.OrientationTarget` | `robotics.PoseTarget`

#### Topics

“Plan a Reaching Trajectory With Multiple Kinematic Constraints”

**Introduced in R2017a**

# robotics.PRM class

**Package:** robotics

Create probabilistic roadmap path planner

## Description

PRM creates a roadmap path planner object for the environment map specified in the `Map` property. The object uses the map to generate a roadmap, which is a network graph of possible paths in the map based on free and occupied spaces. You can customize the number of nodes, `NumNodes`, and the connection distance, `ConnectionDistance`, to fit the complexity of the map and find an obstacle-free path from a start to an end location.

After the map is defined, the PRM path planner generates the specified number of nodes throughout the free spaces in the map. A connection between nodes is made when a line between two nodes contains no obstacles and is within the specified connection distance.

After defining a start and end location, to find an obstacle-free path using this network of connections, use the `findpath` method. If `findpath` does not find a connected path, it returns an empty array. By increasing the number of nodes or the connection distance, you can improve the likelihood of finding a connected path, but tuning these properties is necessary. To see the roadmap and the generated path, use the visualization options in `show`. If you change any of the PRM properties, call `update`, `show`, or `findpath` to recreate the roadmap.

## Construction

`planner = robotics.PRM` creates an empty roadmap with default properties. Before you can use the roadmap, you must specify a `robotics.BinaryOccupancyGrid` object in the `Map` property.

`planner = robotics.PRM(map)` creates a roadmap with `map` set as the `Map` property, where `map` is an object of the `robotics.BinaryOccupancyGrid` class.

`planner = robotics.PRM(map,numnodes)` sets the maximum number of nodes, `numnodes`, to the `NumNodes` property.

## Input Arguments

### **map — Map representation**

BinaryOccupancyGrid object

Map representation, specified as a `robotics.BinaryOccupancyGrid` object. This object represents the environment of the robot. The object is a matrix grid with binary values indicating obstacles as `true` (1) and free locations as `false` (0).

### **numnodes — Maximum number of nodes in roadmap**

50 (default) | scalar

Maximum number of nodes in roadmap, specified as a scalar. By increasing this value, the complexity and computation time for the path planner increases.

## Properties

### **ConnectionDistance — Maximum distance between two connected nodes**

inf (default) | scalar in meters

Maximum distance between two connected nodes, specified as the comma-separated pair consisting of "ConnectionDistance" and a scalar in meters. This property controls whether nodes are connected based on their distance apart. Nodes are connected only if no obstacles are directly in the path. By decreasing this value, the number of connections is lowered, but the complexity and computation time decreases as well.

### **Map — Map representation**

BinaryOccupancyGrid object | OccupancyGrid object

Map representation, specified as the comma-separated pair consisting of "Map" and a `robotics.BinaryOccupancyGrid` or `robotics.OccupancyGrid` object. This object represents the environment of the robot. The object is a matrix grid with values indicating the occupancy of locations in the map.

### **NumNodes — Number of nodes in the map**

50 (default) | scalar

Number of nodes in the map, specified as the comma-separated pair consisting of "NumNodes" and a scalar. By increasing this value, the complexity and computation time for the path planner increases.



## Methods

<code>findpath</code>	Find path between start and goal points on roadmap
<code>show</code>	Show map, roadmap, and path
<code>update</code>	Create or update roadmap

## Extended Capabilities

### C/C++ Code Generation

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

Usage notes and limitations:

The map input must be specified on creation of the PRM object.

### See Also

`robotics.BinaryOccupancyGrid` | `robotics.OccupancyGrid` |  
`robotics.PurePursuit`

### Topics

“Path Planning in Environments of Different Complexity”  
“Probabilistic Roadmaps (PRM)”

**Introduced in R2015a**

## robotics.PurePursuit

**Package:** robotics

Create controller to follow set of waypoints

### Description

The `robotics.PurePursuit` System object creates a controller object used to make a differential drive robot follow a set of waypoints. The object computes the linear and angular velocities for the robot given the current pose of the robot. Successive calls to the object with updated poses provide updated velocity commands for the robot. Use the `MaxAngularVelocity` and `DesiredLinearVelocity` properties to update the velocities based on the robot's performance.

The `LookaheadDistance` property computes a look-ahead point on the path, which is a local goal for the robot. The angular velocity command is computed based on this point. Changing `LookaheadDistance` has a significant impact on the performance of the algorithm. A higher look-ahead distance results in a smoother trajectory for the robot, but can cause the robot to cut corners along the path. A low look-ahead distance can result in oscillations in tracking the path, causing unstable behavior. For more information on the pure pursuit algorithm, see "Pure Pursuit Controller".

---

**Note** Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

---

To compute linear and angular velocity control commands:

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

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

## Creation

### Syntax

```
controller = robotics.PurePursuit
controller = robotics.PurePursuit(Name,Value)
```

### Description

`controller = robotics.PurePursuit` creates a pure pursuit object that uses the pure pursuit algorithm to compute the linear and angular velocity inputs for a differential drive robot.

`controller = robotics.PurePursuit(Name,Value)` creates a pure pursuit object with additional options specified by one or more `Name,Value` pairs. `Name` is the property name and `Value` is the corresponding value. `Name` must appear inside single quotes ( ' '). You can specify several name-value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`. Properties not specified retain their default values.

Example: `controller = robotics.PurePursuit('DesiredLinearVelocity', 0.5)`

## Properties

### **DesiredLinearVelocity** — Desired constant linear velocity

0.1 (default) | scalar in meters per second

Desired constant linear velocity, specified as a scalar in meters per second. The controller assumes that the robot drives at a constant linear velocity and that the computed angular velocity is independent of the linear velocity.

Data Types: `double`

### **LookaheadDistance** — Look-ahead distance

1.0 (default) | scalar in meters

Look-ahead distance, specified as a scalar in meters. The look-ahead distance changes the response of the controller. A robot with a higher look-ahead distance produces smooth

paths but takes larger turns at corners. A robot with a smaller look-ahead distance follows the path closely and takes sharp turns, but potentially creating oscillations in the path.

Data Types: double

## **MaxAngularVelocity — Maximum angular velocity**

1.0 (default) | scalar in radians per second

Maximum angular velocity, specified a scalar in radians per second. The controller saturates the absolute angular velocity output at the given value.

Data Types: double

## **Waypoints — Waypoints**

[ ] (default) | *n*-by-2 array

Waypoints, specified as an *n*-by-2 array of [x y] pairs, where *n* is the number of waypoints. You can generate the waypoints from the PRM class or from another source.

Data Types: double

## **Usage**

## **Syntax**

```
[vel,angvel] = controller(pose)
[vel,angvel,lookaheadpoint] = controller(pose)
```

## **Description**

[vel,angvel] = controller(pose) processes the robot's position and orientation, pose, and outputs the linear velocity, `vel`, and angular velocity, `angvel`.

[vel,angvel,lookaheadpoint] = controller(pose) returns the look-ahead point, which is a location on the path used to compute the velocity commands. This location on the path is computed using the `LookaheadDistance` property on the `controller` object.

## Input Arguments

### **pose** — Position and orientation of robot

3-by-1 vector in the form [x y theta]

Position and orientation of robot, specified as a 3-by-1 vector in the form [x y theta]. The robot's pose is an x and y position with angular orientation  $\theta$  (in radians) measured from the x-axis.

## Output Arguments

### **vel** — Linear velocity

scalar in meters per second

Linear velocity, specified as a scalar in meters per second.

Data Types: double

### **angvel** — Angular velocity

scalar in radians per second

Angular velocity, specified as a scalar in radians per second.

Data Types: double

### **lookaheadpoint** — Look-ahead point on path

[x y] vector

Look-ahead point on the path, returned as an [x y] vector. This value is calculated based on the LookaheadDistance property.

## Object Functions

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

```
release(obj)
```

### **Specific torobotics.PurePursuit**

info Characteristic information about PurePursuit object

## Common to All System Objects

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

## Examples

### Get Additional PurePursuit Object Information

Use the `info` method to get more information about a `PurePursuit` object. `info` returns two fields, `RobotPose` and `LookaheadPoint`, which correspond to the current position and orientation of the robot and the point on the path used to compute outputs from the last call of the object.

Create a `PurePursuit` object.

```
pp = robotics.PurePursuit;
```

Assign waypoints.

```
pp.Waypoints = [0 0;1 1];
```

Compute control commands using the `pp` object with the initial pose `[x y theta]` given as the input.

```
[v,w] = pp([0 0 0]);
```

Get additional information.

```
s = info(pp)
```

```
s = struct with fields:  
    RobotPose: [0 0 0]  
    LookaheadPoint: [0.7071 0.7071]
```

## Extended Capabilities

### C/C++ Code Generation

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

Usage notes and limitations:

Use in a MATLAB Function block is not supported.

For additional information about code generation for System objects, see “System Objects in MATLAB Code Generation” (MATLAB Coder)

### See Also

`robotics.BinaryOccupancyGrid` | `robotics.OccupancyGrid` | `robotics.PRM`

### Topics

“Path Following for a Differential Drive Robot”

“Pure Pursuit Controller”

**Introduced in R2015a**

# quaternion

Create a quaternion array

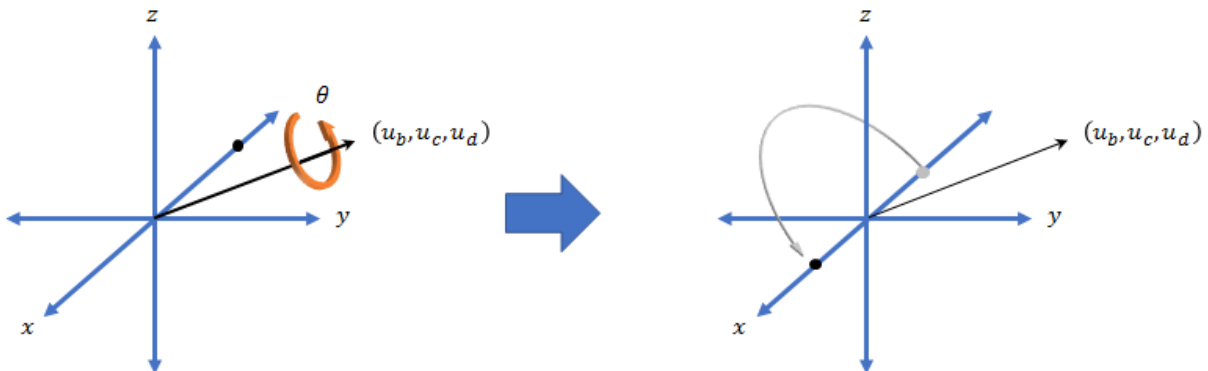
## Description

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  $\theta$  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];
```

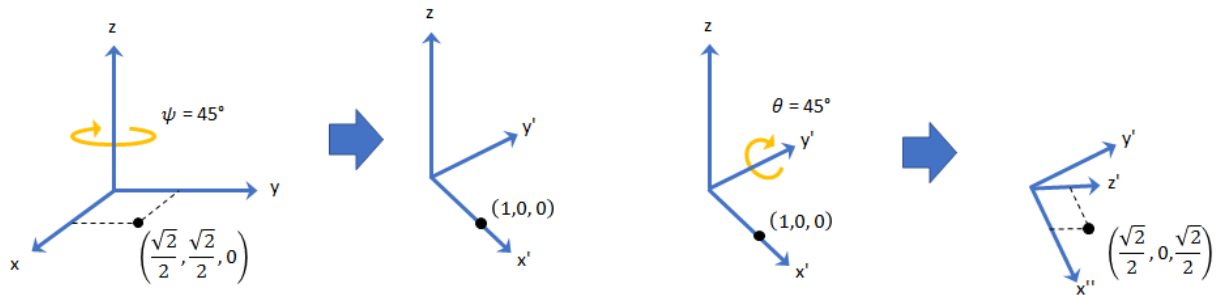
In this representation, the first column represents the x-axis, the second column represents the y-axis, and the third column represents the z-axis.

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)
```

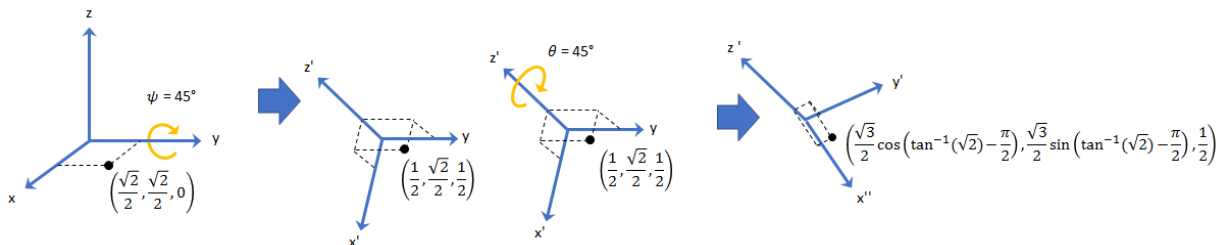
```
newPointCoordinate =  
    0.7071    -0.0000    0.7071
```



- 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

classUnderlying	Class of parts within quaternion
compact	Convert quaternion array to N-by-4 matrix
conj	Complex conjugate of quaternion
ctranspose	Complex conjugate transpose of quaternion array
dist	Angular distance in radians
euler	Convert quaternion to Euler angles (radians)

eulerd	Convert quaternion to Euler angles (degrees)
exp	Exponential of quaternion array
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
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
rdivide, ./	Element-wise quaternion right division
ldivide, .\	Element-wise quaternion left division
transpose	Transpose a quaternion array
uminus, -	Quaternion unary minus
zeros	Create quaternion array with all parts set to zero

## Examples

### Create Empty Quaternion

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

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

```
classUnderlying(quat)
```

```
ans =  
'double'
```

### Create Quaternion by Specifying Individual Quaternion Parts

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

#### Define quaternion parts as scalars.

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

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

#### Define quaternion parts as column vectors.

```
A = [1.1;1.2];  
B = [2.1;2.2];  
C = [3.1;3.2];  
D = [4.1;4.2];  
quatVector = quaternion(A,B,C,D)
```

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

#### Define quaternion parts as matrices.

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

```
4.2,4.4];
quatMatrix = quaternion(A,B,C,D)

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

### Define quaternion parts as three dimensional arrays.

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

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

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

quatMultiDimArray(:,:,2) =

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

### 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 = 3x3

    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
```

$$Q2 = \text{quaternion}(9,8,7,6)$$

$$Q2 = \text{quaternion} \\ 9 + 8i + 7j + 6k$$

$$Q1\text{plus}Q2 = Q1 + Q2$$

$$Q1\text{plus}Q2 = \text{quaternion} \\ 10 + 10i + 10j + 10k$$

$$Q2\text{plus}Q1 = Q2 + Q1$$

$$Q2\text{plus}Q1 = \text{quaternion} \\ 10 + 10i + 10j + 10k$$

$$Q1\text{minus}Q2 = Q1 - Q2$$

$$Q1\text{minus}Q2 = \text{quaternion} \\ -8 - 6i - 4j - 2k$$

$$Q2\text{minus}Q1 = Q2 - Q1$$

$$Q2\text{minus}Q1 = \text{quaternion} \\ 8 + 6i + 4j + 2k$$

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.

$$Q1\text{plusRealNumber} = Q1 + 5$$

$$Q1\text{plusRealNumber} = \text{quaternion} \\ 6 + 2i + 3j + 4k$$

$$Q1\text{minusRealNumber} = Q1 - 5$$

$$Q1\text{minusRealNumber} = \text{quaternion} \\ -4 + 2i + 3j + 4k$$

## Multiplication

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

```
Q1timesQ2 = Q1 * Q2
```

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

```
Q2timesQ1 = Q2 * Q1
```

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

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

```
ans = logical  
0
```

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

```
Q1times5 = Q1*5
```

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

Multiplying a quaternion by a real number is commutative.

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

```
ans = logical  
1
```

## Conjugation

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

```
Q1
```

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

```
conj(Q1)
```

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

Multiplication between a quaternion and its conjugate is commutative:

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

```
ans = logical  
    1
```

## Quaternion Array Manipulation

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

### Concatenate

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

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

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

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

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

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

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

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

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

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

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

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

## Indexing

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

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

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

Replace the quaternion at index two with a quaternion one.

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

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

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

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

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



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

## 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
```

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

## Permute

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

```
qMultiDimensionalArray
```

```
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) =
```

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

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

```
qMatPermute = 2x2x2 quaternion array
```

```
qMatPermute(:,:,1) =
```

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

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

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

## Extended Capabilities

### C/C++ Code Generation

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

### See Also

**Introduced in R2018a**

# robotics.Rate

Execute loop at fixed frequency

## Description

The `Rate` object enables you to run a loop at a fixed frequency. It also collects statistics about the timing of the loop iterations. Use `waitfor` in the loop to pause code execution until the next time step. The loop operates every `DesiredPeriod` seconds, unless the enclosed code takes longer to operate. The object uses the `OverrunAction` property to determine how it handles longer loop operation times. The default setting, `'slip'`, immediately executes the loop if `LastPeriod` is greater than `DesiredPeriod`. Using `'drop'` causes the `waitfor` method to wait until the next multiple of `DesiredPeriod` is reached to execute the next loop.

---

**Tip** The scheduling resolution of your operating system and the level of other system activity can affect rate execution accuracy. As a result, accurate rate timing is limited to 100 Hz for execution of MATLAB code. To improve performance and execution speeds, use code generation.

---

## Creation

## Syntax

```
rateObj = robotics.Rate(desiredRate)
```

## Description

`rateObj = robotics.Rate(desiredRate)` creates a `Rate` object that operates loops at a fixed-rate based on your system time and directly sets the `DesireRate` property.

## Properties

### **DesiredRate — Desired execution rate**

scalar

Desired execution rate of loop, specified as a scalar in Hz. When using `waitfor`, the loop operates every `DesiredRate` seconds, unless the loop takes longer. It then begins the next loop based on the specified `OverrunAction`.

### **DesiredPeriod — Desired time period between executions**

scalar

Desired time period between executions, specified as a scalar in seconds. This property is equal to the inverse of `DesiredRate`.

### **TotalElapsedTime — Elapsed time since construction or reset**

scalar

Elapsed time since construction or reset, specified as a scalar in seconds.

### **LastPeriod — Elapsed time between last two calls to `waitfor`**

NaN (default) | scalar

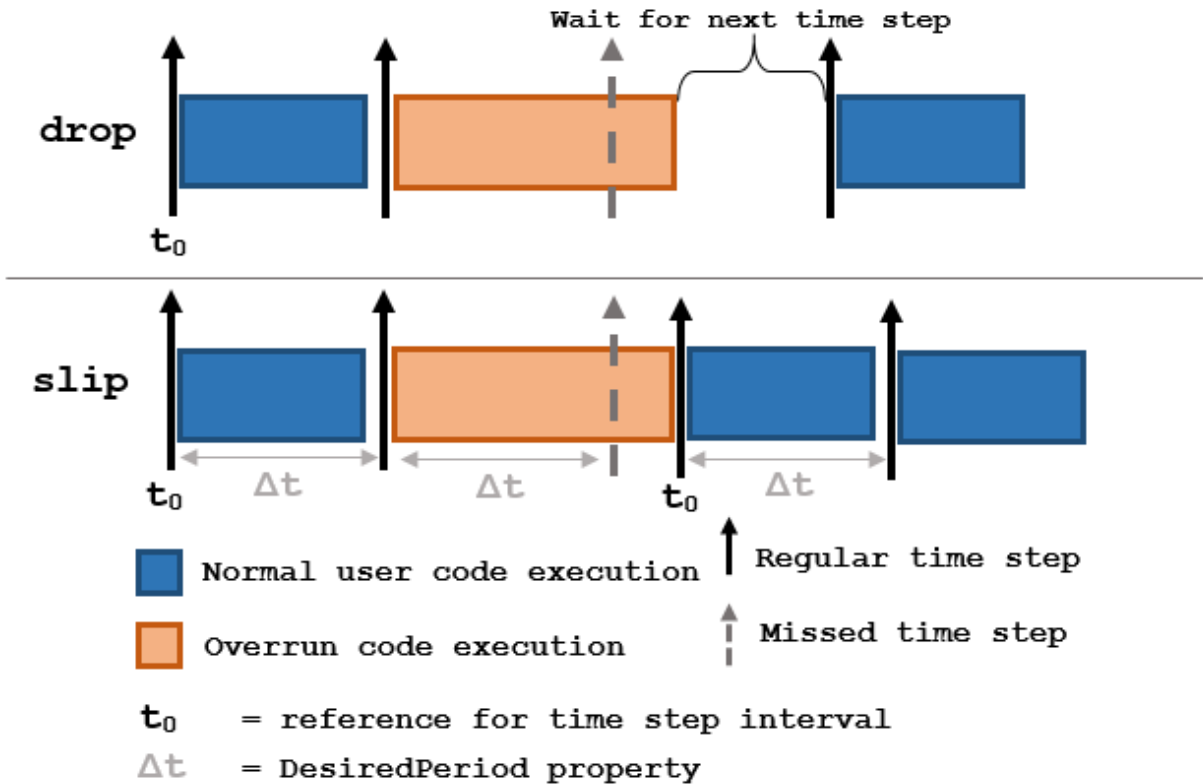
Elapsed time between last two calls to `waitfor`, specified as a scalar. By default, `LastPeriod` is set to NaN until `waitfor` is called for the first time. After the first call, `LastPeriod` equals `TotalElapsedTime`.

### **OverrunAction — Method for handling overruns**

'slip' (default) | 'drop'

Method for handling overruns, specified as one of these character vectors:

- 'drop' — waits until the next time interval equal to a multiple of `DesiredPeriod`
- 'slip' — immediately executes the loop again



Each code section calls wait for at the end of execution.

## Object Functions

waitfor    Pause code execution to achieve desired execution rate  
 statistics    Statistics of past execution periods  
 reset        Reset Rate object

## Examples

## Run Loop at Fixed Rate

Create a rate object that runs at 1 Hz.

```
r = robotics.Rate(1);
```

Start a loop using the Rate object inside to control the loop execution. Reset the object prior to the loop execution to reset timer. Print the iteration and time elapsed.

```
reset(r)
for i = 1:10
    time = r.TotalElapsedTime;
    fprintf('Iteration: %d - Time Elapsed: %f\n',i,time)
    waitfor(r);
end
```

```
Iteration: 1 - Time Elapsed: 0.004475
Iteration: 2 - Time Elapsed: 1.005114
Iteration: 3 - Time Elapsed: 2.004950
Iteration: 4 - Time Elapsed: 3.004543
Iteration: 5 - Time Elapsed: 4.005585
Iteration: 6 - Time Elapsed: 5.005309
Iteration: 7 - Time Elapsed: 6.005151
Iteration: 8 - Time Elapsed: 7.004336
Iteration: 9 - Time Elapsed: 8.004075
Iteration: 10 - Time Elapsed: 9.005004
```

Each iteration executes at a 1-second interval.

## Get Statistics From Rate Object Execution

Create a Rate object for running at 20 Hz.

```
r = robotics.Rate(2);
```

Start a loop and control operation using the Rate object.

```
for i = 1:30
    % Your code goes here
    waitfor(r);
end
```

Get Rate object statistics after loop operation.

```
stats = statistics(r)
```

```
stats = struct with fields:
    Periods: [1x30 double]
    NumPeriods: 30
    AveragePeriod: 0.5000
    StandardDeviation: 4.7101e-04
    NumOverruns: 0
```

### Run Loop At Fixed Rate and Reset Rate Object

Create a Rate object for running at 20 Hz.

```
r = robotics.Rate(2);
```

Start a loop and control operation using the Rate object.

```
for i = 1:30
    % Your code goes here
    waitfor(r);
end
```

Display the Rate object properties after loop operation.

```
disp(r)
```

```
Rate with properties:
    DesiredRate: 2
    DesiredPeriod: 0.5000
    OverrunAction: 'slip'
    TotalElapsedTime: 15.0270
    LastPeriod: 0.5000
```

Reset the object to restart the time statistics.

```
reset(r);
disp(r)
```

```
Rate with properties:
    DesiredRate: 2
    DesiredPeriod: 0.5000
    OverrunAction: 'slip'
    TotalElapsedTime: 0.0273
    LastPeriod: NaN
```

## **See Also**

`rosclock` | `waitfor`

## **Topics**

“Execute Code at a Fixed-Rate”

**Introduced in R2016a**



# robotics.ReedsSheppConnection

Reeds-Shepp path connection type

## Description

The `ReedsSheppConnection` object holds information for computing a `ReedsSheppPathSegment` object to connect between poses. A Reeds-Shepp path segment connects two poses as a sequence of five motions. The motion options are:

- Straight
- Left turn at maximum steer
- Right turn at maximum steer
- No movement

A Reeds-Shepp path segment supports both forward and backward motion.

Use this connection object to define parameters for a robot motion model, including the minimum turning radius and options for path types. To generate a path segment between poses using this connection type, call the `connect` function.

## Creation

## Syntax

```
reedsConnObj = robotics.ReedsSheppConnection  
reedsConnObj = robotics.ReedsSheppConnection(Name, Value)
```

## Description

`reedsConnObj = robotics.ReedsSheppConnection` creates an object using default property values.

`reedsConnObj = robotics.ReedsSheppConnection(Name, Value)` specifies property values using name-value pairs. To set multiple properties, specify multiple name-value pairs.

## Properties

### **MinTurningRadius** — Minimum turning radius

1 (default) | positive scalar in meters

Minimum turning radius for the robot, specified as a positive scalar in meters. The minimum turning radius is for the smallest circle the robot can make with maximum steer in a single direction.

Data Types: `double`

### **DisabledPathTypes** — Path types to disable

{ } (default) | vector of string scalars | cell array of character vectors

Path types to disable, specified as a vector of string scalars or cell array of character vectors. Valid values are:

Motion Type	Description
"Sp", "Sn"	Straight (p = forward, n=reverse)
"Lp", "Ln"	Left turn at the maximum steering angle of the vehicle (p = forward, n=reverse)
"Rp", "Rn"	Right turn at the maximum steering angle of the vehicle (p = forward, n=reverse)
"N"	No motion

If a path segment has fewer than five motion types, the remaining elements are "N" (no motion).

To see all available path types, see the `AllPathTypes` property.

Example: `["LpSnLp", "LnSnRpSn", "LnSnRpSnLp"]`

Data Types: `cell`

### **AllPathTypes** — All possible path types

cell array of character vectors

This property is read-only.

All possible path types, specified as a cell array of character vectors. This property lists all types. To disable certain types, specify types from this list in `DisabledPathTypes`.

For Reeds-Shepp connections, there are 44 possible combinations of motion types.

Data Types: `cell`

### **ForwardCost — Cost multiplier to travel forward**

1 (default) | positive numeric scalar

Cost multiple to travel forward, specified as a positive numeric scalar. Increase this property to penalize forward motion.

Data Types: `double`

### **ReverseCost — Cost multiplier to travel in reverse**

1 (default) | positive numeric scalar

Cost multiple to travel in reverse, specified as a positive numeric scalar. Increase this property to penalize reverse motion.

Data Types: `double`

## **Object Functions**

`connect` Connect poses for given connection type

## **Examples**

### **Connect Poses Using ReedsShepp Connection Path**

Create a `ReedsSheppConnection` object.

```
reedsConnObj = robotics.ReedsSheppConnection;
```

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

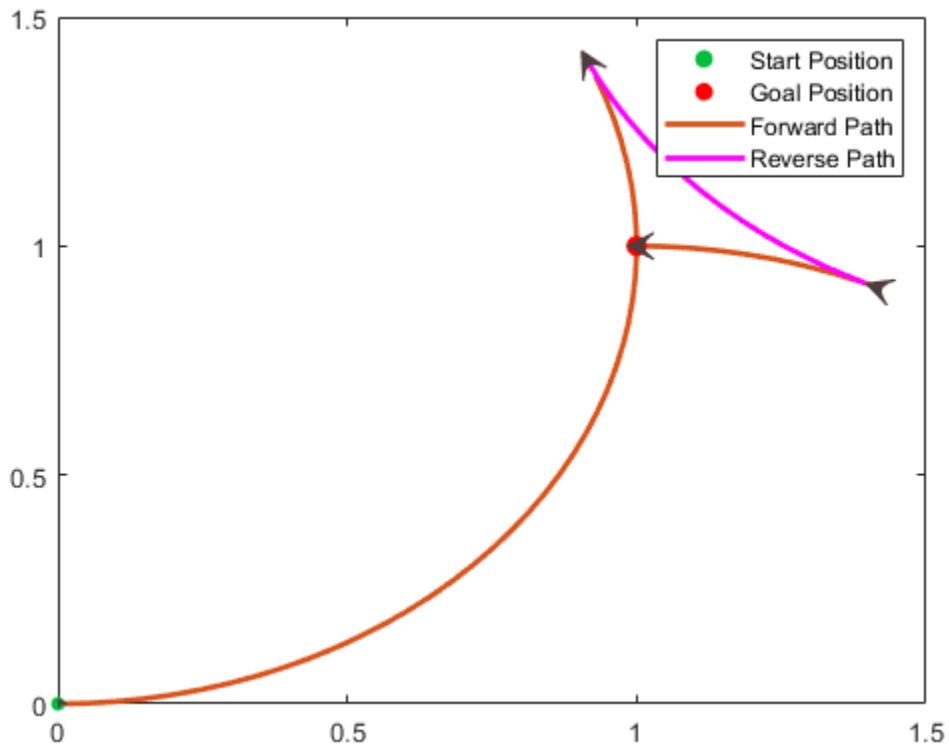
```
startPose = [0 0 0];
goalPose = [1 1 pi];
```

Calculate a valid path segment to connect the poses.

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

Show the generated path.

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



### **Modify Connection Types for Reeds-Shepp Path**

Create a ReedsSheppConnection object.

```
reedsConnObj = robotics.ReedsSheppConnection;
```

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

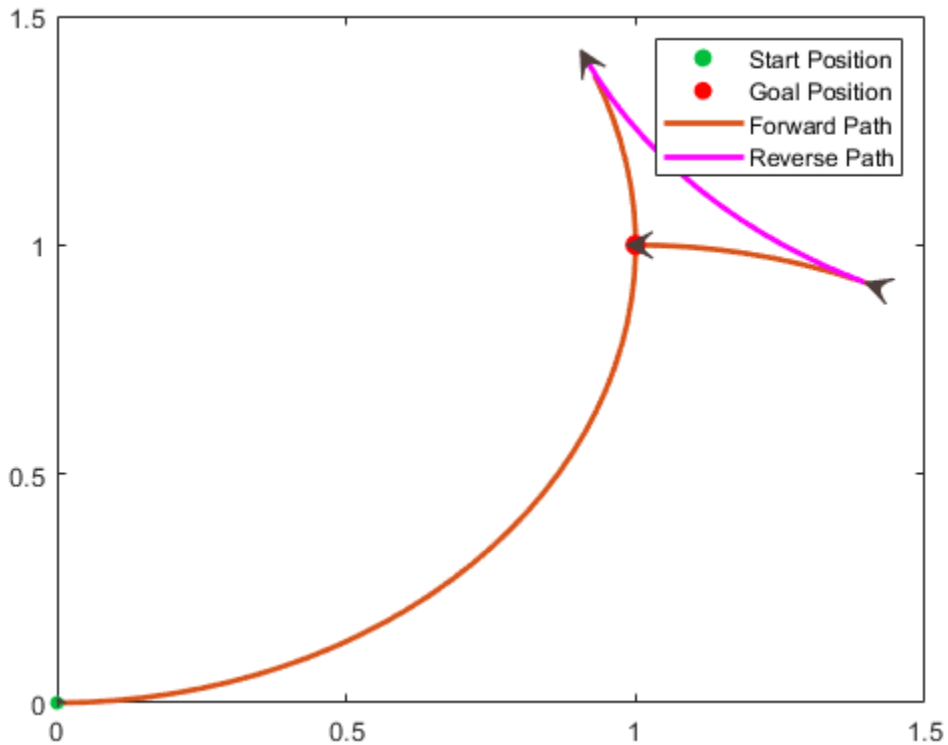
```
startPose = [0 0 0];  
goalPose = [1 1 pi];
```

Calculate a valid path segment to connect the poses.

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

Show the generated path. Notice the direction of the turns.

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



```
pathSegObj{1}.MotionTypes
```

```
ans = 1x5 cell array
      {'L'}      {'R'}      {'L'}      {'N'}      {'N'}
```

```
pathSegObj{1}.MotionDirections
```

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

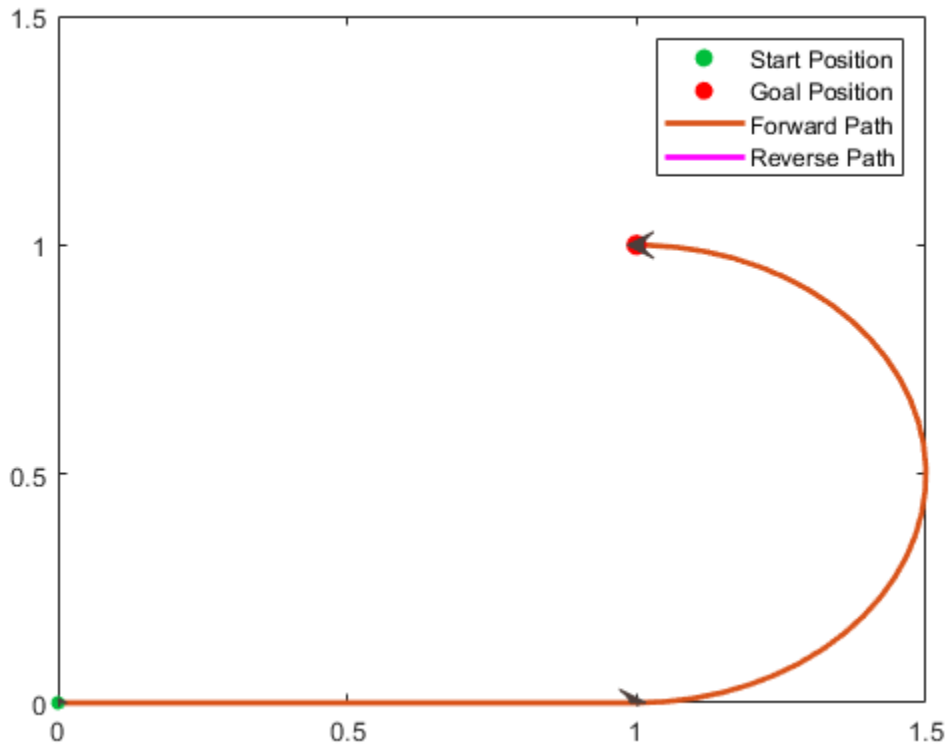
Disable this specific motion sequence in a new connection object. Reduce the `MinTurningRadius` if the robot is more maneuverable. Increase the reverse cost to reduce the likelihood of reverse directions being used. Connect the poses again to get a different path.

```
reedsConnObj = robotics.ReedsSheppConnection('DisabledPathTypes',{'LpRnLp'});
reedsConnObj.MinTurningRadius = 0.5;
reedsConnObj.ReverseCost = 5;
```

```
[pathSegObj,pathCosts] = connect(reedsConnObj,startPose,goalPose);
pathSegObj{1}.MotionTypes
```

```
ans = 1x5 cell array
      {'L'}      {'S'}      {'L'}      {'N'}      {'N'}
```

```
show(pathSegObj{1})
xlim([0 1.5])
ylim([0 1.5])
```



## Extended Capabilities

### C/C++ Code Generation

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

## See Also

### Objects

`robotics.DubinsConnection` | `robotics.DubinsPathSegment` |  
`robotics.ReedsSheppPathSegment`

### Functions

`connect` | `interpolate` | `show`

**Introduced in R2018b**



# robotics.ReedsSheppPathSegment

Reeds-Shepp path segment connecting two poses

## Description

The `ReedsSheppPathSegment` object holds information for a Reeds-Shepp path segment to connect between poses. A Reeds-Shepp path segment connects two poses as a sequence of five motion types. The motion options are:

- Straight
- Left turn at maximum steer
- Right turn at maximum steer
- No movement

## Creation

To generate a `ReedsSheppPathSegment` object, use the `connect` function with a `robotics.ReedsSheppConnection` object:

`reedsPathSegObj = connect(connectionObj, start, goal)` connects the start and goal poses using the specified connection type object.

To specifically define a path segment:

`reedsPathSegObj = robotics.ReedsSheppPathSegment(connectionObj, start, goal, motionLengths, motionTypes)` specifies the Reeds-Shepp connection type, the start and goal poses, and the corresponding motion lengths and types. These values are set to the corresponding properties in the object.

## Properties

**MinTurningRadius** — Minimum turning radius of vehicle

positive scalar

This property is read-only.

Minimum turning radius of the robot, specified as a positive scalar in meters. This value corresponds to the radius of the turning circle at the maximum steering angle of the robot.

Data Types: double

### **StartPose — Initial pose of robot**

$[x, y, \theta]$  vector

This property is read-only.

Initial pose of the robot at the start of the path segment, specified as an  $[x, y, \theta]$  vector.  $x$  and  $y$  are in meters.  $\theta$  is in radians.

Data Types: double

### **GoalPose — Goal pose of robot**

$[x, y, \theta]$  vector

This property is read-only.

Goal pose of the robot at the end of the path segment, specified as an  $[x, y, \theta]$  vector.  $x$  and  $y$  are in meters.  $\theta$  is in radians.

Data Types: double

### **MotionLengths — Length of each motion**

five-element numeric vector

This property is read-only.

Length of each motion in the path segment, specified as a five-element numeric vector in meters. Each motion length corresponds to a motion type specified in `MotionTypes`.

Data Types: double

### **MotionTypes — Type of each motion**

five-element string cell array

This property is read-only.

Type of each motion in the path segment, specified as a five-element string cell array. Valid values are:

<b>Motion Type</b>	<b>Description</b>
"S"	Straight (forward, p or reverse, n)
"L"	Left turn at the maximum steering angle of the vehicle (forward, p or reverse, n)
"R"	Right turn at the maximum steering angle of the vehicle (forward, p or reverse, n)
"N"	No motion

If a path segment has fewer than five motion types, the remaining elements are "N" (no motion).

Example: {"L", "S", "R", "L", "R"}

Data Types: cell

#### **MotionDirections — Direction of each motion**

five-element vector of 1s (forward motion) and -1s (reverse motion)

This property is read-only.

Direction of each motion in the path segment, specified as a five-element vector of 1s (forward motion) and -1s (reverse motion). Each motion direction corresponds to a motion length specified in `MotionLengths` and a motion type specified in `MotionTypes`.

When no motion occurs, that is, when a `MotionTypes` value is "N", then the corresponding `MotionDirections` element is 1.

Example: [-1 1 -1 1 1]

Data Types: double

#### **Length — Length of path segment**

positive scalar

This property is read-only.

Length of the path segment, specified as a positive scalar in meters. This length is just a sum of the elements in `MotionLengths`.

Data Types: double

## Object Functions

interpolate Interpolate poses along path segment  
show Visualize path segment

## Examples

### Connect Poses Using ReedsShepp Connection Path

Create a ReedsSheppConnection object.

```
reedsConnObj = robotics.ReedsSheppConnection;
```

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

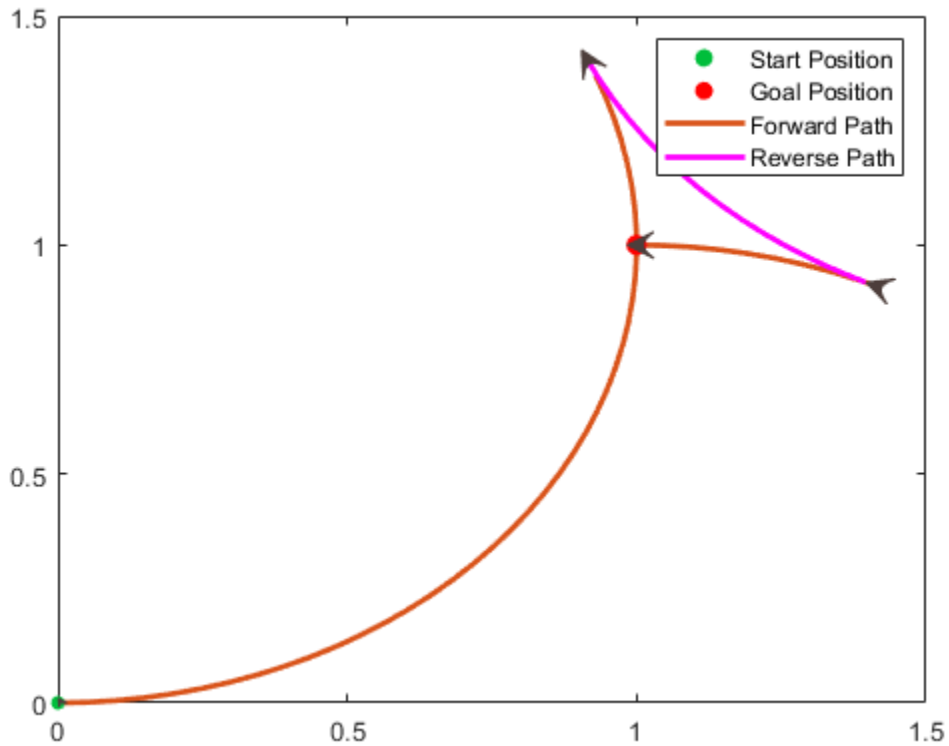
```
startPose = [0 0 0];  
goalPose = [1 1 pi];
```

Calculate a valid path segment to connect the poses.

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

Show the generated path.

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



## References

- [1] Reeds, J. A., and L. A. Shepp. "Optimal Paths for a Car That Goes Both Forwards and Backwards." *Pacific Journal of Mathematics*. Vol. 145, Number 2, 1990, pp. 367–393.

## Extended Capabilities

### C/C++ Code Generation

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

### See Also

#### Objects

`robotics.DubinsConnection` | `robotics.DubinsPathSegment` |  
`robotics.ReedsSheppConnection`

#### Functions

`connect` | `interpolate` | `show`

**Introduced in R2018b**

# robotics.ResamplingPolicy class

**Package:** robotics

Create resampling policy object with resampling settings

## Description

ResamplingPolicy creates an object encapsulating settings for when resampling should occur when using a particle filter for state estimation. The object contains the method that triggers resampling and the relevant threshold for this resampling. Use this object as the ResamplingPolicy property of the ParticleFilter class.

## Construction

`policy = robotics.ResamplingPolicy` creates a ResamplingPolicy object which contains properties to be modified to control when resampling should be triggered. Use this object as the ResamplingPolicy property of the ParticleFilter class.

## Properties

### TriggerMethod — Method for determining if resampling should occur

'ratio' (default) | character vector

Method for determining if resampling should occur, specified as a character vector. Possible choices are 'ratio' and 'interval'. The 'interval' method triggers resampling at regular intervals of operating the particle filter. The 'ratio' method triggers resampling based on the ratio of effective total particles.

### SamplingInterval — Fixed interval between resampling

1 (default) | scalar

Fixed interval between resampling, specified as a scalar. This interval determines during which correction steps the resampling is executed. For example, a value of 2 means the resampling is executed every second correction step. A value of `inf` means that resampling is never executed.

This property only applies with the `TriggerMethod` is set to `'interval'`.

## **MinEffectiveParticleRatio** — Minimum desired ratio of effective to total particles

0.5 (default) | scalar

Minimum desired ratio of effective to total particles, specified as a scalar. The effective number of particles is a measure of how well the current set of particles approximates the posterior distribution. A lower effective particle ratio means less particles are contributing to the estimation and resampling might be required. If the ratio of effective particles to total particles falls below the `MinEffectiveParticleRatio`, a resampling step is triggered.

## **See Also**

`robotics.ParticleFilter` | `robotics.ParticleFilter.correct`

## **Topics**

“Track a Car-Like Robot Using Particle Filter”

Class Attributes (MATLAB)

Property Attributes (MATLAB)

**Introduced in R2016a**



# robotics.RigidBody class

**Package:** robotics

Create a rigid body

## Description

The `RigidBody` class represents a rigid body. A rigid body is the building block for any tree-structured robot manipulator. Each `RigidBody` has a `robotics.Joint` object attached to it that defines how the rigid body can move. Rigid bodies are assembled into a tree-structured robot model using `robotics.RigidBodyTree`.

Set a joint object to the `Joint` property before calling `robotics.RigidBodyTree.addBody` to add the rigid body to the robot model. When a rigid body is in a rigid body tree, you cannot directly modify its properties because it corrupts the relationships between bodies. Use `robotics.RigidBodyTree.replaceJoint` to modify the entire tree structure.

## Construction

`body = robotics.RigidBody(name)` creates a rigid body with the specified name. By default, the body comes with a fixed joint.

## Input Arguments

**name** — Name of rigid body

string scalar | character vector

Name of the rigid body, specified as a string scalar or character vector. This name must be unique to the body so that it can be accessed in a `RigidBodyTree` object.

## Properties

### **Name — Name of rigid body**

string scalar | character vector

Name of the rigid body, specified as a string scalar or character vector. This name must be unique to the body so that it can be found in a `RigidBodyTree` object.

Data Types: `char` | `string`

### **Joint — Joint object**

handle

Joint object, specified as a handle. By default, the joint is 'fixed' type. Create the joint using `robotics.Joint` and specify the joint type on creation.

### **Mass — Mass of rigid body**

1 kg (default) | numeric scalar

Mass of rigid body, specified as a numeric scalar in kilograms.

### **CenterOfMass — Center of mass position of rigid body**

[0 0 0] m (default) | [x y z] vector

Center of mass position of rigid body, specified as an [x y z] vector. The vector describes the location of the center of mass relative to the body frame in meters.

### **Inertia — Inertia of rigid body**

[1 1 1 0 0 0] kg•m<sup>2</sup> (default) | [Ixx Iyy Izz Iyz Ixz Ixy] vector

Inertia of rigid body, specified as a [Ixx Iyy Izz Iyz Ixz Ixy] vector relative to the body frame in kilogram square meters. The first three elements of the vector are the diagonal elements of the inertia tensor. The last three elements are the off-diagonal elements of the inertia tensor. The inertia tensor is a positive definite symmetric matrix:

$$\begin{pmatrix} I_{xx} & I_{xy} & I_{xz} \\ I_{xy} & I_{yy} & I_{yz} \\ I_{xz} & I_{yz} & I_{zz} \end{pmatrix}$$

### **Parent — Rigid body parent**

RigidBody object handle

Rigid body parent, specified as a `RigidBody` object handle. The rigid body joint defines how this body can move relative to the parent. This property is empty until the rigid body is added to a `RigidBodyTree` robot model.

### Children — Rigid body children

cell array of `RigidBody` object handles

Rigid body children, specified as a cell array of `RigidBody` object handles. These rigid body children are all attached to this rigid body object. This property is empty until the rigid body is added to a `RigidBodyTree` robot model, and at least one other body is added to the tree with this body as its parent.

### Visuals — Visual geometries

cell array of string scalars | cell array of character vectors

Visual geometries, specified as a cell array of string scalars or character vectors. Each character vector describes a type and source of a visual geometry. For example, if a mesh file, `link_0.stl`, is attached to the rigid body, the visual would be `Mesh:link_0.stl`. Visual geometries are added to the rigid body using `robotics.RigidBody.addVisual`.

## Methods

<code>addVisual</code>	Add visual geometry data to rigid body
<code>clearVisual</code>	Clear all visual geometries
<code>copy</code>	Create a deep copy of rigid body

## Examples

### Attach Rigid Body and Joint to Rigid Body Tree

Add a rigid body and corresponding joint to a rigid body tree. Each `RigidBody` object contains a `Joint` object and must be added to the `RigidBodyTree` using `addBody`.

Create a rigid body tree.

```
rbtree = robotics.RigidBodyTree;
```

Create a rigid body with a unique name.

```
body1 = robotics.RigidBody('b1');
```

Create a revolute joint. By default, the `RigidBody` object comes with a fixed joint. Replace the joint by assigning a new `Joint` object to the `body1.Joint` property.

```
jnt1 = robotics.Joint('jnt1', 'revolute');  
body1.Joint = jnt1;
```

Add the rigid body to the tree. Specify the body name that you are attaching the rigid body to. Because this is the first body, use the base name of the tree.

```
basename = rbtree.BaseName;  
addBody(rbtree, body1, basename)
```

Use `showdetails` on the tree to confirm the rigid body and joint were added properly.

```
showdetails(rbtree)
```

```
-----  
Robot: (1 bodies)  
  
  Idx   Body Name   Joint Name   Joint Type   Parent Name(Idx)   Children Name(s)  
  ---   -  
    1     b1         jnt1        revolute        base(0)  
-----
```

### **Build Manipulator Robot Using Denavit-Hartenberg Parameters**

Use the Denavit-Hartenberg (DH) parameters of the Puma560® robot to build a robot. Each rigid body is added one at a time, with the child-to-parent transform specified by the joint object.

The DH parameters define the geometry of the robot with relation to how each rigid body is attached to its parent. For convenience, setup the parameters for the Puma560 robot in a matrix. The Puma robot is a serial chain manipulator. The DH parameters are relative to the previous line in the matrix, corresponding to the previous joint attachment.

```
dhparams = [0      pi/2    0      0;  
            0.4318  0      0      0;  
            0.0203 -pi/2   0.15005  0;
```

```

0      pi/2    0.4318    0;
0      -pi/2   0         0;
0      0       0         0];

```

Create a rigid body tree object to build the robot.

```
robot = robotics.RigidBodyTree;
```

Create the first rigid body and add it to the robot. To add a rigid body:

- 1 Create a `RigidBody` object and give it a unique name.
- 2 Create a `Joint` object and give it a unique name.
- 3 Use `setFixedTransform` to specify the body-to-body transformation using DH parameters. The last element of the DH parameters, `theta`, is ignored because the angle is dependent on the joint position.
- 4 Call `addBody` to attach the first body joint to the base frame of the robot.

```

body1 = robotics.RigidBody('body1');
jnt1 = robotics.Joint('jnt1','revolute');

setFixedTransform(jnt1,dhparams(1,:), 'dh');
body1.Joint = jnt1;

addBody(robot,body1,'base')

```

Create and add other rigid bodies to the robot. Specify the previous body name when calling `addBody` to attach it. Each fixed transform is relative to the previous joint coordinate frame.

```

body2 = robotics.RigidBody('body2');
jnt2 = robotics.Joint('jnt2','revolute');
body3 = robotics.RigidBody('body3');
jnt3 = robotics.Joint('jnt3','revolute');
body4 = robotics.RigidBody('body4');
jnt4 = robotics.Joint('jnt4','revolute');
body5 = robotics.RigidBody('body5');
jnt5 = robotics.Joint('jnt5','revolute');
body6 = robotics.RigidBody('body6');
jnt6 = robotics.Joint('jnt6','revolute');

setFixedTransform(jnt2,dhparams(2,:), 'dh');
setFixedTransform(jnt3,dhparams(3,:), 'dh');
setFixedTransform(jnt4,dhparams(4,:), 'dh');

```

```
setFixedTransform(jnt5,dhparams(5,:), 'dh');
setFixedTransform(jnt6,dhparams(6,:), 'dh');

body2.Joint = jnt2;
body3.Joint = jnt3;
body4.Joint = jnt4;
body5.Joint = jnt5;
body6.Joint = jnt6;

addBody(robot,body2, 'body1')
addBody(robot,body3, 'body2')
addBody(robot,body4, 'body3')
addBody(robot,body5, 'body4')
addBody(robot,body6, 'body5')
```

Verify that your robot was built properly by using the `showdetails` or `show` function. `showdetails` lists all the bodies in the MATLAB® command window. `show` displays the robot with a given configuration (home by default). Calls to `axis` modify the axis limits and hide the axis labels.

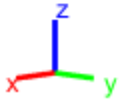
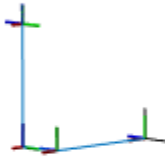
```
showdetails(robot)
```

```
-----
Robot: (6 bodies)
```

Idx	Body Name	Joint Name	Joint Type	Parent Name(Idx)	Children Name(s)
---	-----	-----	-----	-----	-----
1	body1	jnt1	revolute	base(0)	body2(2)
2	body2	jnt2	revolute	body1(1)	body3(3)
3	body3	jnt3	revolute	body2(2)	body4(4)
4	body4	jnt4	revolute	body3(3)	body5(5)
5	body5	jnt5	revolute	body4(4)	body6(6)
6	body6	jnt6	revolute	body5(5)	

```
-----
```

```
show(robot);
axis([-0.5,0.5,-0.5,0.5,-0.5,0.5])
axis off
```



## References

- [1] Craig, John J. *Introduction to Robotics: Mechanics and Control*. Reading, MA: Addison-Wesley, 1989.
- [2] Siciliano, Bruno. *Robotics: Modelling, Planning and Control*. London: Springer, 2009.

## **Extended Capabilities**

### **C/C++ Code Generation**

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

### **See Also**

`robotics.Joint` | `robotics.RigidBodyTree` |  
`robotics.RigidBodyTree.addBody` | `robotics.RigidBodyTree.replaceJoint`

### **Topics**

“Build a Robot Step by Step”  
“Rigid Body Tree Robot Model”  
Class Attributes (MATLAB)  
Property Attributes (MATLAB)

**Introduced in R2016b**



# robotics.RigidBodyTree class

**Package:** robotics

Create tree-structured robot

## Description

The `RigidBodyTree` is a representation of the connectivity of rigid bodies with joints. Use this class to build robot manipulator models in MATLAB. If you have a robot model specified using the Unified Robot Description Format (URDF), use `importrobot` to import your robot model.

A rigid body tree model is made up of rigid bodies as `RigidBody` objects. Each rigid body has a `Joint` object associated with it that defines how it can move relative to its parent body. Use `setFixedTransform` to define the fixed transformation between the frame of a joint and the frame of one of the adjacent bodies. You can add, replace, or remove rigid bodies from the model using the methods of the `RigidBodyTree` class.

Robot dynamics calculations are also possible. Specify the `Mass`, `CenterOfMass`, and `Inertia` properties for each `RigidBody` in the robot model. You can calculate forward and inverse dynamics with or without external forces and compute dynamics quantities given robot joint motions and joint inputs. To use the dynamics-related functions, set the `DataFormat` property to "row" or "column".

For a given rigid body tree model, you can also use the robot model to calculate joint angles for desired end-effector positions using the robotics inverse kinematics algorithms. Specify your rigid body tree model when using `InverseKinematics` or `GeneralizedInverseKinematics`.

The `show` method supports visualization of body meshes. Meshes are specified as `.stl` files and can be added to individual rigid bodies using `addVisual`. Also, by default, the `importrobot` function loads all the accessible `.stl` files specified in your URDF robot model.

## Construction

`robot = robotics.RigidBodyTree` creates a tree-structured robot object. Add rigid bodies to it using `addBody`.

`robot = robotics.RigidBodyTree("MaxNumBodies", N, "DataFormat", dataFormat)` specifies an upper bound on the number of bodies allowed in the robot when generating code. You must also specify the `DataFormat` property as a name-value pair.

## Properties

### **NumBodies — Number of bodies**

integer

This property is read-only.

Number of bodies in the robot model (not including the base), returned as an integer.

### **Bodies — List of rigid bodies**

cell array of handles

This property is read-only.

List of rigid bodies in the robot model, returned as a cell array of handles. Use this list to access specific `RigidBody` objects in the model. You can also call `robotics.RigidBodyTree.getBody` to get a body by its name.

### **BodyNames — Names of rigid bodies**

cell array of string scalars | cell array of character vectors

This property is read-only.

Names of rigid bodies, returned as a cell array of character vectors.

### **BaseName — Name of robot base**

'base' (default) | string scalar | character vector

Name of robot base, returned as a string scalar or character vector.

**Gravity — Gravitational acceleration experienced by robot**

`[0 0 0]` m/s<sup>2</sup> (default) | `[x y z]` vector

Gravitational acceleration experienced by robot, specified as an `[x y z]` vector in meters per second squared. Each element corresponds to the acceleration of the base robot frame in that direction.

**DataFormat — Input/output data format for kinematics and dynamics functions**

`"struct"` (default) | `"row"` | `"column"`

Input/output data format for kinematics and dynamics functions, specified as `"struct"`, `"row"`, or `"column"`. To use dynamics functions, you must use either `"row"` or `"column"`.

## Methods

<code>addBody</code>	Add body to robot
<code>addSubtree</code>	Add subtree to robot
<code>centerOfMass</code>	Center of mass position and Jacobian
<code>copy</code>	Copy robot model
<code>externalForce</code>	Compose external force matrix relative to base
<code>forwardDynamics</code>	Joint accelerations given joint torques and states
<code>geometricJacobian</code>	Geometric Jacobian for robot configuration
<code>gravityTorque</code>	Joint torques that compensate gravity
<code>getBody</code>	Get robot body handle by name
<code>getTransform</code>	Get transform between body frames
<code>homeConfiguration</code>	Get home configuration of robot
<code>inverseDynamics</code>	Required joint torques for given motion
<code>massMatrix</code>	Joint-space mass matrix
<code>randomConfiguration</code>	Generate random configuration of robot
<code>removeBody</code>	Remove body from robot
<code>replaceBody</code>	Replace body on robot
<code>replaceJoint</code>	Replace joint on body
<code>show</code>	Show robot model in a figure
<code>showdetails</code>	Show details of robot model
<code>subtree</code>	Create subtree from robot model
<code>velocityProduct</code>	Joint torques that cancel velocity-induced forces

## Examples

### **Attach Rigid Body and Joint to Rigid Body Tree**

Add a rigid body and corresponding joint to a rigid body tree. Each `RigidBody` object contains a `Joint` object and must be added to the `RigidBodyTree` using `addBody`.

Create a rigid body tree.

```
rbtree = robotics.RigidBodyTree;
```

Create a rigid body with a unique name.

```
body1 = robotics.RigidBody('b1');
```

Create a revolute joint. By default, the `RigidBody` object comes with a fixed joint. Replace the joint by assigning a new `Joint` object to the `body1.Joint` property.

```
jnt1 = robotics.Joint('jnt1', 'revolute');
body1.Joint = jnt1;
```

Add the rigid body to the tree. Specify the body name that you are attaching the rigid body to. Because this is the first body, use the base name of the tree.

```
basename = rbtree.BaseName;
addBody(rbtree, body1, basename)
```

Use `showdetails` on the tree to confirm the rigid body and joint were added properly.

```
showdetails(rbtree)
```

```
-----
Robot: (1 bodies)
```

Idx	Body Name	Joint Name	Joint Type	Parent Name(Idx)	Children Name(s)
1	b1	jnt1	revolute	base(0)	

## Build Manipulator Robot Using Denavit-Hartenberg Parameters

Use the Denavit-Hartenberg (DH) parameters of the Puma560® robot to build a robot. Each rigid body is added one at a time, with the child-to-parent transform specified by the joint object.

The DH parameters define the geometry of the robot with relation to how each rigid body is attached to its parent. For convenience, setup the parameters for the Puma560 robot in a matrix. The Puma robot is a serial chain manipulator. The DH parameters are relative to the previous line in the matrix, corresponding to the previous joint attachment.

```
dhparams = [0      pi/2    0      0;
            0.4318  0      0      0;
            0.0203 -pi/2   0.15005 0;
            0      pi/2    0.4318 0;
            0      -pi/2   0      0;
            0      0      0      0];
```

Create a rigid body tree object to build the robot.

```
robot = robotics.RigidBodyTree;
```

Create the first rigid body and add it to the robot. To add a rigid body:

- 1 Create a `RigidBody` object and give it a unique name.
- 2 Create a `Joint` object and give it a unique name.
- 3 Use `setFixedTransform` to specify the body-to-body transformation using DH parameters. The last element of the DH parameters, `theta`, is ignored because the angle is dependent on the joint position.
- 4 Call `addBody` to attach the first body joint to the base frame of the robot.

```
body1 = robotics.RigidBody('body1');
jnt1 = robotics.Joint('jnt1','revolute');

setFixedTransform(jnt1,dhparams(1,:), 'dh');
body1.Joint = jnt1;

addBody(robot,body1,'base')
```

Create and add other rigid bodies to the robot. Specify the previous body name when calling `addBody` to attach it. Each fixed transform is relative to the previous joint coordinate frame.

```
body2 = robotics.RigidBody('body2');
jnt2 = robotics.Joint('jnt2','revolute');
body3 = robotics.RigidBody('body3');
jnt3 = robotics.Joint('jnt3','revolute');
body4 = robotics.RigidBody('body4');
jnt4 = robotics.Joint('jnt4','revolute');
body5 = robotics.RigidBody('body5');
jnt5 = robotics.Joint('jnt5','revolute');
body6 = robotics.RigidBody('body6');
jnt6 = robotics.Joint('jnt6','revolute');
```

```

setFixedTransform(jnt2,dhparams(2,:), 'dh');
setFixedTransform(jnt3,dhparams(3,:), 'dh');
setFixedTransform(jnt4,dhparams(4,:), 'dh');
setFixedTransform(jnt5,dhparams(5,:), 'dh');
setFixedTransform(jnt6,dhparams(6,:), 'dh');

body2.Joint = jnt2;
body3.Joint = jnt3;
body4.Joint = jnt4;
body5.Joint = jnt5;
body6.Joint = jnt6;

addBody(robot,body2, 'body1')
addBody(robot,body3, 'body2')
addBody(robot,body4, 'body3')
addBody(robot,body5, 'body4')
addBody(robot,body6, 'body5')

```

Verify that your robot was built properly by using the `showdetails` or `show` function. `showdetails` lists all the bodies in the MATLAB® command window. `show` displays the robot with a given configuration (home by default). Calls to `axis` modify the axis limits and hide the axis labels.

```
showdetails(robot)
```

```

-----
Robot: (6 bodies)

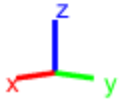
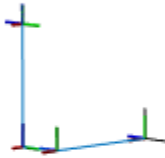
  Idx   Body Name   Joint Name   Joint Type   Parent Name(Idx)   Children Name(s)
  ---   -
  1     body1         jnt1        revolute    base(0)            body2(2)
  2     body2         jnt2        revolute    body1(1)           body3(3)
  3     body3         jnt3        revolute    body2(2)           body4(4)
  4     body4         jnt4        revolute    body3(3)           body5(5)
  5     body5         jnt5        revolute    body4(4)           body6(6)
  6     body6         jnt6        revolute    body5(5)
-----

```

```

show(robot);
axis([-0.5,0.5,-0.5,0.5,-0.5,0.5])
axis off

```



## Modify a Robot Rigid Body Tree Model

Make changes to an existing `RigidBodyTree` object. You can get replace joints, bodies and subtrees in the rigid body tree.

Load example robots as `RigidBodyTree` objects.

```
load exampleRobots.mat
```

View the details of the Puma robot using `showdetails`.

```
showdetails(puma1)
```



```

-----
Robot: (6 bodies)

  Idx   Body Name   Joint Name   Joint Type   Parent Name(Idx)   Children Name(s)
-----
    1      L1       jnt1       revolute      base(0)           L2(2)
    2      L2       jnt2       revolute      L1(1)             L3(3)
    3      L3       jnt3       revolute      L2(2)             L4(4)
    4      L4       jnt4       revolute      L3(3)             L5(5)
    5      L5       jnt5       revolute      L4(4)             L6(6)
    6      L6       jnt6       revolute      L5(5)
-----

```

Get a specific body to inspect the properties. The only child of the L3 body is the L4 body. You can copy a specific body as well.

```

body3 = getBody(puma1, 'L3');
childBody = body3.Children{1}

childBody =
  RigidBody with properties:

      Name: 'L4'
      Joint: [1x1 robotics.Joint]
      Mass: 1
  CenterOfMass: [0 0 0]
      Inertia: [1 1 1 0 0 0]
      Parent: [1x1 robotics.RigidBody]
      Children: {[1x1 robotics.RigidBody]}
      Visuals: {}

```

```
body3Copy = copy(body3);
```

Replace the joint on the L3 body. You must create a new `Joint` object and use `replaceJoint` to ensure the downstream body geometry is unaffected. Call `setFixedTransform` if necessary to define a transform between the bodies instead of with the default identity matrices.

```

newJoint = robotics.Joint('prismatic');
replaceJoint(puma1, 'L3', newJoint);

showdetails(puma1)

```

```

-----
Robot: (6 bodies)

```

Idx	Body Name	Joint Name	Joint Type	Parent Name(Idx)	Children Name
1	L1	jnt1	revolute	base(0)	L2(2)
2	L2	jnt2	revolute	L1(1)	L3(3)
3	L3	prismatic	fixed	L2(2)	L4(4)
4	L4	jnt4	revolute	L3(3)	L5(5)
5	L5	jnt5	revolute	L4(4)	L6(6)
6	L6	jnt6	revolute	L5(5)	

Remove an entire body and get the resulting subtree using `removeBody`. The removed body is included in the subtree.

```
subtree = removeBody(puma1, 'L4')
```

```
subtree =
  RigidBodyTree with properties:
```

```
  NumBodies: 3
  Bodies: {1x3 cell}
  Base: [1x1 robotics.RigidBody]
  BodyNames: {'L4' 'L5' 'L6'}
  BaseName: 'L3'
  Gravity: [0 0 0]
  DataFormat: 'struct'
```

Remove the modified L3 body. Add the original copied L3 body to the L2 body, followed by the returned subtree. The robot model remains the same. See a detailed comparison through `showdetails`.

```
removeBody(puma1, 'L3');
addBody(puma1,body3Copy, 'L2')
addSubtree(puma1, 'L3',subtree)
```

```
showdetails(puma1)
```

```
-----
Robot: (6 bodies)
```

Idx	Body Name	Joint Name	Joint Type	Parent Name(Idx)	Children Name(s)
1	L1	jnt1	revolute	base(0)	L2(2)
2	L2	jnt2	revolute	L1(1)	L3(3)

3	L3	jnt3	revolute	L2(2)	L4(4)
4	L4	jnt4	revolute	L3(3)	L5(5)
5	L5	jnt5	revolute	L4(4)	L6(6)
6	L6	jnt6	revolute	L5(5)	

-----

### Specify Dynamics Properties to Rigid Body Tree

To use dynamics functions to calculate joint torques and accelerations, specify the dynamics properties for the `robotics.RigidBodyTree` object and `robotics.RigidBody`.

Create a rigid body tree model. Create two rigid bodies to attach to it.

```
robot = robotics.RigidBodyTree('DataFormat', 'row');
body1 = robotics.RigidBody('body1');
body2 = robotics.RigidBody('body2');
```

Specify joints to attach to the bodies. Set the fixed transformation of `body2` to `body1`. This transform is 1m in the x-direction.

```
joint1 = robotics.Joint('joint1', 'revolute');
joint2 = robotics.Joint('joint2');
setFixedTransform(joint2, trvec2tform([1 0 0]))
body1.Joint = joint1;
body2.Joint = joint2;
```

Specify dynamics properties for the two bodies. Add the bodies to the robot model. For this example, basic values for a rod (`body1`) with an attached spherical mass (`body2`) are given.

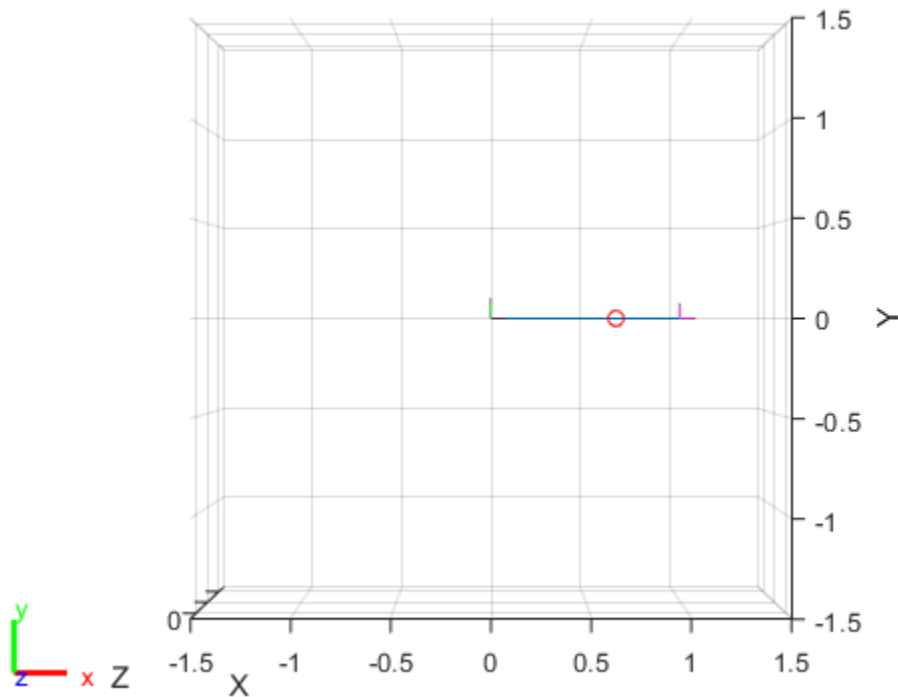
```
body1.Mass = 2;
body1.CenterOfMass = [0.5 0 0];
body1.Inertia = [0.167 0.001 0.167 0 0 0];

body2.Mass = 1;
body2.CenterOfMass = [0 0 0];
body2.Inertia = 0.0001*[4 4 4 0 0 0];

addBody(robot, body1, 'base');
addBody(robot, body2, 'body1');
```

Compute the center of mass position of the whole robot. Plot the position on the robot.  
Move the view to the xy plane.

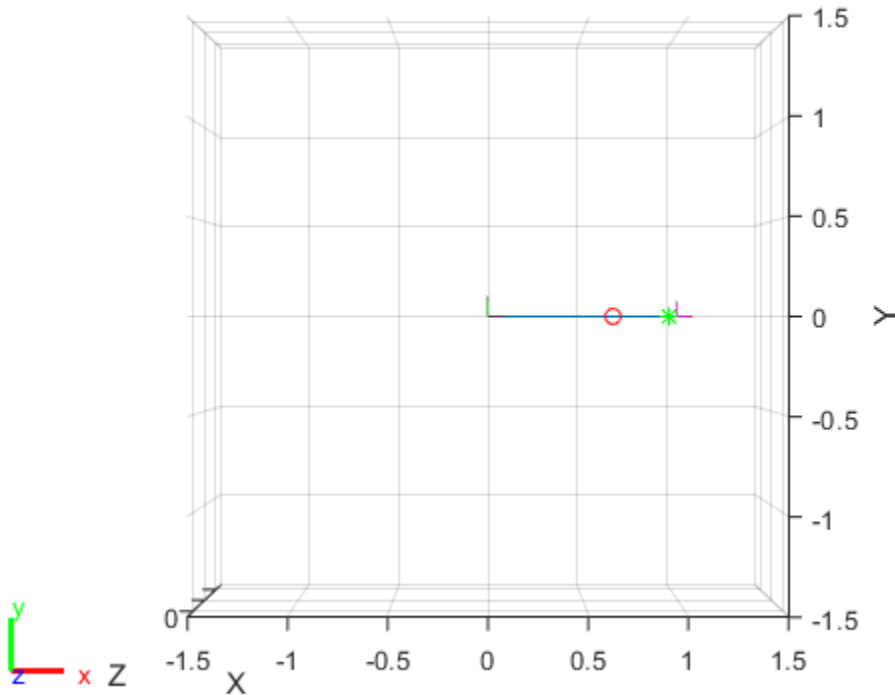
```
comPos = centerOfMass(robot);  
  
show(robot);  
hold on  
plot(comPos(1), comPos(2), 'or')  
view(2)
```



Change the mass of the second body. Notice the change in center of mass.

```
body2.Mass = 20;  
replaceBody(robot, 'body2', body2)
```

```
comPos2 = centerOfMass(robot);  
plot(comPos2(1),comPos2(2), '*g')  
hold off
```



### Compute Forward Dynamics Due to External Forces on Rigid Body Tree Model

Calculate the resultant joint accelerations for a given robot configuration with applied external forces and forces due to gravity. A wrench is applied to a specific body with the gravity being specified for the whole robot.

Load a predefined KUKA LBR robot model, which is specified as a `RigidBodyTree` object.

```
load exampleRobots.mat lbr
```

Set the data format to 'row'. For all dynamics calculations, the data format must be either 'row' or 'column'.

```
lbr.DataFormat = 'row';
```

Set the gravity. By default, gravity is assumed to be zero.

```
lbr.Gravity = [0 0 -9.81];
```

Get the home configuration for the `lbr` robot.

```
q = homeConfiguration(lbr);
```

Specify the wrench vector that represents the external forces experienced by the robot. Use the `externalForce` function to generate the external force matrix. Specify the robot model, the end effector that experiences the wrench, the wrench vector, and the current robot configuration. `wrench` is given relative to the 'tool0' body frame, which requires you to specify the robot configuration, `q`.

```
wrench = [0 0 0.5 0 0 0.3];  
fext = externalForce(lbr, 'tool0', wrench, q);
```

Compute the resultant joint accelerations due to gravity, with the external force applied to the end-effector 'tool0' when `lbr` is at its home configuration. The joint velocities and joint torques are assumed to be zero (input as an empty vector []).

```
qddot = forwardDynamics(lbr, q, [], [], fext);
```

### Compute Inverse Dynamics from Static Joint Configuration

Use the `inverseDynamics` function to calculate the required joint torques to statically hold a specific robot configuration. You can also specify the joint velocities, joint accelerations, and external forces using other syntaxes.

Load a predefined KUKA LBR robot model, which is specified as a `RigidBodyTree` object.

```
load exampleRobots.mat lbr
```

Set the data format to 'row'. For all dynamics calculations, the data format must be either 'row' or 'column'.

```
lbr.DataFormat = 'row';
```

Set the Gravity property to give a specific gravitational acceleration.

```
lbr.Gravity = [0 0 -9.81];
```

Generate a random configuration for lbr.

```
q = randomConfiguration(lbr);
```

Compute the required joint torques for lbr to statically hold that configuration.

```
tau = inverseDynamics(lbr,q);
```

### Compute Joint Torque to Counter External Forces

Use the `externalForce` function to generate force matrices to apply to a rigid body tree model. The force matrix is an  $m$ -by-6 vector that has a row for each joint on the robot to apply a six-element wrench. Use the `externalForce` function and specify the end effector to properly assign the wrench to the correct row of the matrix. You can add multiple force matrices together to apply multiple forces to one robot.

To calculate the joint torques that counter these external forces, use the `inverseDynamics` function.

Load a predefined KUKA LBR robot model, which is specified as a `RigidBodyTree` object.

```
load exampleRobots.mat lbr
```

Set the data format to 'row'. For all dynamics calculations, the data format must be either 'row' or 'column'.

```
lbr.DataFormat = 'row';
```

Set the Gravity property to give a specific gravitational acceleration.

```
lbr.Gravity = [0 0 -9.81];
```

Get the home configuration for `lbr`.

```
q = homeConfiguration(lbr);
```

Set external force on `link1`. The input wrench vector is expressed in the base frame.

```
fext1 = externalForce(lbr, 'link_1', [0 0 0.0 0.1 0 0]);
```

Set external force on the end effector, `tool0`. The input wrench vector is expressed in the `tool0` frame.

```
fext2 = externalForce(lbr, 'tool0', [0 0 0.0 0.1 0 0], q);
```

Compute the joint torques required to balance the external forces. To combine the forces, add the force matrices together. Joint velocities and accelerations are assumed to be zero (input as []).

```
tau = inverseDynamics(lbr, q, [], [], fext1+fext2);
```

### Display Robot Model with Visual Geometries

You can import robots that have `.stl` files associated with the Unified Robot Description format (URDF) file to describe the visual geometries of the robot. Each rigid body has an individual visual geometry specified. The `importrobot` function parses the URDF file to get the robot model and visual geometries. Use the `show` function to visualize the robot model in a figure. You can then interact with the model by clicking components to inspect them and right-clicking to toggle visibility.

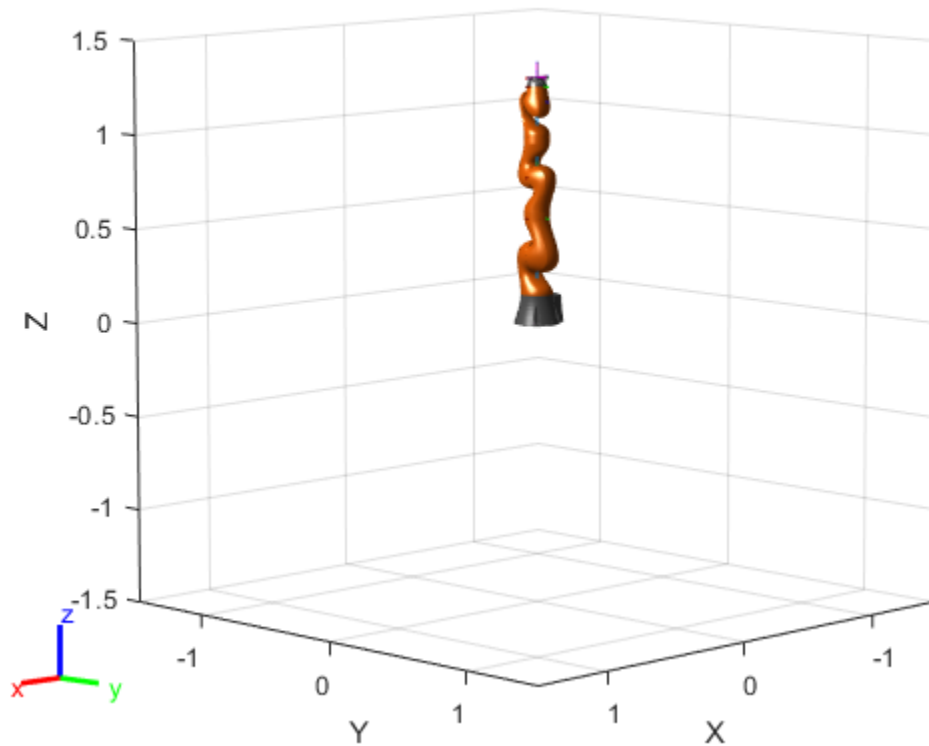
Import a robot model as a URDF file. The `.stl` file locations must be properly specified in this URDF. To add other `.stl` files to individual rigid bodies, see `addVisual`.

```
robot = importrobot('iiwa14.urdf');
```

Visualize the robot with the associated visual model. Click bodies or frames to inspect them. Right-click bodies to toggle visibility for each visual geometry.

```
show(robot);
```





## References

- [1] Craig, John J. *Introduction to Robotics: Mechanics and Control*. Reading, MA: Addison-Wesley, 1989.
- [2] Siciliano, Bruno, Lorenzo Sciavicco, Luigi Villani, and Giuseppe Oriolo. *Robotics: Modelling, Planning and Control*. London: Springer, 2009.

## Extended Capabilities

### C/C++ Code Generation

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

Usage notes and limitations:

Use the syntax that specifies the `MaxNumBodies` as an upper bound for adding bodies to the robot model. You must also specify the `DataFormat` property as a name-value pair. For example:

```
robot = robotics.RigidBodyTree("MaxNumBodies",15,"DataFormat","row")
```

To minimize data usage, limit the upper bound to a number close to the expected number of bodies in the model. All data formats are supported for code generation. To use the dynamics functions, the data format must be set to "row" or "column".

Also, the `show` and `showdetails` functions do not support code generation.

### See Also

```
importrobot | robotics.GeneralizedInverseKinematics |  
robotics.InverseKinematics | robotics.Joint | robotics.RigidBody
```

### Topics

- "Build a Robot Step by Step"
- "Rigid Body Tree Robot Model"
- "Solve Inverse Kinematics for a Four-Bar Linkage"
- "Control PR2 Arm Movements Using ROS Actions and Inverse Kinematics"
- "Plan a Reaching Trajectory With Multiple Kinematic Constraints"
- "Control LBR Manipulator Motion Through Joint Torque Commands"

**Introduced in R2016b**

# robotics.RigidBodyTreeImportInfo

Object for storing RigidBodyTree import information

## Description

The `RigidBodyTreeImportInfo` object is created by the `importrobot` function when converting a Simulink model using Simscape™ Multibody™ components. Get import information for specific bodies, joints, or blocks using the object functions. Changes to the Simulink model are not reflected in this object after initially calling `importrobot`.

## Creation

`[robot,importInfo] = importrobot(model)` imports a Simscape Multibody model and returns an equivalent `robotics.RigidBodyTree` object, `robot`, and info about the import in `importInfo`. Only fixed, prismatic, and revolute joints are supported in the output `RigidBodyTree` object.

If you are importing a model that uses other joint types, constraint blocks, or variable inertias, use the “Simscape Multibody Model Import” on page 2-0 name-value pairs to disable errors.

## Properties

### SourceModelName — Name of source model from Simscape Multibody

character vector

This property is read-only.

Name of the source model from Simscape Multibody, specified as a character vector. This property matches the name of the input `model` when calling `importrobot`.

Example: `'sm_import_humanoid_urdf'`

Data Types: `char`

## **RigidBodyTree — Robot model**

RigidBodyTree object

This property is read-only.

Robot model, returned as a `robotics.RigidBodyTree` object.

## **BlockConversionInfo — List of blocks that were converted**

structure

This property is read-only.

List of blocks that were converted from Simscape Multibody blocks to preserve compatibility, specified as a structure with the nested fields:

- `AddedBlocks`
  - `ImplicitJoints` — Cell array of implicit joints added during the conversion process.
- `ConvertedBlocks`
  - `Joints` — Cell array of joint blocks that were converted to fixed joints.
  - `JointSourceType` — `containers.Map` object that associates converted joint blocks to their original joint type.
- `RemovedBlocks`
  - `ChainClosureJoints` — Cell array of joint blocks removed to open closed chains.
  - `SMConstraints` — Cell array of constraint blocks that were removed.
  - `VariableInertias` — Cell array of variable inertia blocks that were removed.

## **Object Functions**

<code>bodyInfo</code>	Import information for body
<code>bodyInfoFromBlock</code>	Import information for block name
<code>bodyInfoFromJoint</code>	Import information for given joint name
<code>showdetails</code>	Display details of imported robot

## **See Also**

`importrobot` | `robotics.RigidBodyTree`

## **Topics**

“Rigid Body Tree Robot Model”

**Introduced in R2018b**

## rosactionclient

Create ROS action client

### Description

Use the `rosactionclient` to connect to an action server using a `SimpleActionClient` object and request the execution of action goals. You can get feedback on the execution process and cancel the goal at anytime. The `SimpleActionClient` object encapsulates a simple action client and enables you to track a single goal at a time.

### Creation

### Syntax

```
client = rosactionclient(actionname)
client = rosactionclient(actionname,actiontype)
[client,goalMsg] = rosactionclient( ___ )
```

```
client = robotics.ros.SimpleActionClient(node,actionname)
client = robotics.ros.SimpleActionClient(node,actionname,actiontype)
```

### Description

`client = rosactionclient(actionname)` creates a client for the specified ROS ActionName. The client determines the action type automatically. If the action is not available, this function displays an error.

Use `rosactionclient` to connect to an action server and request the execution of action goals. You can get feedback on the execution progress and cancel the goal at any time.

`client = rosactionclient(actionname, actiontype)` creates an action client with the specified name and type (`ActionType`). If the action is not available, or the name and type do not match, the function displays an error.

`[client, goalMsg] = rosactionclient( ___ )` returns a goal message to send the action client created using any of the arguments from the previous syntaxes. The `Goal` message is initialized with default values for that message.

If the `ActionFcn`, `FeedbackFcn`, and `ResultFcn` callbacks are defined, they are called when the goal is processing on the action server. All callbacks associated with a previously sent goal are disabled, but the previous goal is not canceled.

`client = robotics.ros.SimpleActionClient(node, actionname)` creates a client for the specified ROS action name. `node` is the `Node` object that is connected to the ROS network. The client determines the action type automatically. If the action is not available, the function displays an error.

`client = robotics.ros.SimpleActionClient(node, actionname, actiontype)` creates an action client with the specified name and type. You can get the type of an action using `rosaction type actionname`.

## Properties

### **ActionName — ROS action name**

character vector

ROS action name, returned as a character vector. The action name must match one of the topics that `rosaction("list")` outputs.

### **ActionType — Action type for a ROS action**

string scalar | character vector

Action type for a ROS action, returned as a string scalar or character vector. You can get the action type of an action using `rosaction type <action_name>`. For more details, see `rosaction`.

### **IsServerConnected — Indicates if client is connected to ROS action server**

false (default) | true

Indicator of whether the client is connected to a ROS action server, returned as `false` or `true`. Use `waitForServer` to wait until the server is connected when setting up an action client.

## **Goal — Tracked goal**

ROS message

Tracked goal, returned as a ROS message. This message is the last goal message this client sent. The goal message depends on the action type.

## **GoalState — Goal state**

character vector

Goal state, returned as one of the following:

- `'pending'` — Goal was received, but has not yet been accepted or rejected.
- `'active'` — Goal was accepted and is running on the server.
- `'succeeded'` — Goal executed successfully.
- `'preempted'` — An action client canceled the goal before it finished executing.
- `'aborted'` — The goal was aborted before it finished executing. The action server typically aborts a goal.
- `'rejected'` — The goal was not accepted after being in the `'pending'` state. The action server typically triggers this status.
- `'recalled'` — A client canceled the goal while it was in the `'pending'` state.
- `'lost'` — An internal error occurred in the action client.

## **ActivationFcn — Activation function**

@(~) disp('Goal is active.') (default) | function handle

Activation function, returned as a function handle. This function executes when `GoalState` is set to `'active'`. By default, the function displays `'Goal is active.'`. You can set the function to `[]` to have the action client do nothing upon activation.

## **FeedbackFcn — Feedback function**

@(~,msg) disp(['Feedback: ', showdetails(msg)]) (default) | function handle

Feedback function, returned as a function handle. This function executes when a new feedback message is received from the action server. By default, the function displays the details of the message. You can set the function to `[]` to have the action client not give any feedback.



**ResultFcn — Result function**

```
@(~,msg,s,~) disp(['Result with state ' s ': ', showdetails(msg)])
(default) | function handle
```

Result function, returned as a function handle. This function executes when the server finishes executing the goal and returns a result state and message. By default, the function displays the state and details of the message. You can set the function to [] to have the action client do nothing once the goal is completed.

**Object Functions**

cancelGoal	Cancel last goal sent by client
cancelAllGoals	Cancel all goals on action server
rosmessage	Create ROS messages
sendGoal	Send goal message to action server
sendGoalAndWait	Send goal message and wait for result
waitForServer	Wait for action server to start

**Examples****Setup a ROS Action Client and Execute an Action**

This example shows how to create a ROS action client and execute the action. Action types must be setup beforehand with an action server running.

You must have the '/fibonacci' action type setup. To run this action server use the following command on the ROS system:

```
roslaunch actionlib_tutorials fibonacci_server
```

Connect to a ROS network. You must be connected to a ROS network to gather information about what actions are available. Replace `ipaddress` with your network address.

```
ipaddress = '192.168.154.131';
rosinit(ipaddress)
```

```
Initializing global node /matlab_global_node_68978 with NodeURI http://192.168.154.1:5
```

List actions available on the network. The only action setup on this network is the '/fibonacci' action.

```
roaction list
/fibonacci
```

Create an action client. Specify the action name.

```
[actClient,goalMsg] = roactionclient('/fibonacci');
```

Wait for action client to connect to server.

```
waitForServer(actClient);
```

The fibonacci action will calculate the fibonacci sequence for a given order specified in the goal message. The goal message was returned when creating the action client and can be modified to send goals to the ROS action server.

```
goalMsg.Order = 8
```

```
goalMsg =
```

```
ROS FibonacciGoal message with properties:
```

```
  MessageType: 'actionlib_tutorials/FibonacciGoal'
  Order: 8
```

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

Send goal and wait for its completion. Specify a timeout of 10 seconds to complete the action.

```
[resultMsg,resultState] = sendGoalAndWait(actClient,goalMsg,10)
```

```
Goal active
```

```
Feedback:
```

```
  Sequence : [0, 1, 1]
```

```
Feedback:
```

```
  Sequence : [0, 1, 1, 2]
```

```
Feedback:
```

```
  Sequence : [0, 1, 1, 2, 3]
```

```
Feedback:
```

```

Sequence : [0, 1, 1, 2, 3, 5]
Feedback:
Sequence : [0, 1, 1, 2, 3, 5, 8]
Feedback:
Sequence : [0, 1, 1, 2, 3, 5, 8, 13]
Feedback:
Sequence : [0, 1, 1, 2, 3, 5, 8, 13, 21]
Feedback:
Sequence : [0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
Final state succeeded with result:
Sequence : [0, 1, 1, 2, 3, 5, 8, 13, 21, 34]

```

```
resultMsg =
```

```
ROS FibonacciResult message with properties:
```

```

MessageType: 'actionlib_tutorials/FibonacciResult'
Sequence: [10x1 int32]

```

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

```
resultState =
```

```
1x9 char array
```

```
succeeded
```

Disconnect from the ROS network.

```
roshutdown
```

```
Shutting down global node /matlab_global_node_68978 with NodeURI http://192.168.154.1:5
```

## Send and Cancel ROS Action Goals

Send and cancel goals for ROS actions. First, setup a ROS action client. Then send a goal message with modified parameters. Finally, cancel your goal and all goals on the action server.

Connect to a ROS network with a specified IP address. Create a ROS action client connected using `rosactionclient`. Specify the action name. Wait for the client to be connected to the server.

```
rosinit('192.168.154.131')
[actClient,goalMsg] = rosactionclient('/fibonacci');
waitForServer(actClient);
```

Initializing global node /matlab\_global\_node\_40739 with NodeURI http://192.168.154.1:5

Send a goal message with modified parameters. Wait for the goal to finish executing.

```
goalMsg.Order = 4;
sendGoalAndWait(actClient,goalMsg)
```

Goal active

Feedback:

Sequence : [0, 1, 1]

Feedback:

Sequence : [0, 1, 1, 2]

Feedback:

Sequence : [0, 1, 1, 2, 3]

Feedback:

Sequence : [0, 1, 1, 2, 3, 5]

ans =

ROS FibonacciResult message with properties:

MessageType: 'actionlib\_tutorials/FibonacciResult'

Sequence: [6×1 int32]

Use showdetails to show the contents of the message

Send a new goal message without waiting.

```
goalMsg.Order = 5;
sendGoal(actClient,goalMsg)
```

Cancel the goal on the ROS action client, actClient.

```
cancelGoal(actClient)
```

Cancel all the goals on the action server that actClient is connected to.

```
cancelAllGoals(actClient)
```

Delete the action client.

`delete(actClient)`

Disconnect from the ROS network.

`roshutdown`

Shutting down global node /matlab\_global\_node\_40739 with NodeURI http://192.168.154.1:5

## See Also

`cancelGoal` | `roaction` | `rosmessage` | `sendGoal` | `waitForServer`

## Topics

“ROS Actions Overview”

“Move a Turtlebot Robot Using ROS Actions”

## External Websites

ROS Actions

**Introduced in R2016b**

# rospublisher

Publish message on a topic

## Description

Use `rospublisher` to create a ROS publisher for sending messages via a ROS network. To create ROS messages, use `rosmessage`. Send these messages via the ROS publisher with the `send` function.

The `Publisher` object created by the function represents a publisher on the ROS network. The object publishes a specific message type on a given topic. When the `Publisher` object publishes a message to the topic, all subscribers to the topic receive this message. The same topic can have multiple publishers and subscribers.

The publisher gets the topic message type from the topic list on the ROS master. When the MATLAB global node publishes messages on that topic, ROS nodes that subscribe to that topic receive those messages. If the topic is not on the ROS master topic list, this function displays an error message. If the ROS master topic list already contains a matching topic, the ROS master adds the MATLAB global node to the list of publishers for that topic. To see a list of available topic names, at the MATLAB command prompt, type `rostopic list`.

You can create a `Publisher` object using the `rospublisher` function, or by calling `robotics.ros.Publisher`:

- `rospublisher` only works with the global node using `rosinit`. It does not require a node object handle as an argument.
- `robotics.ros.Publisher` works with additional nodes that are created using `robotics.ros.Node`. It requires a node object handle as the first argument.

## Creation

### Syntax

```
pub = rospublisher(topicname)
pub = rospublisher(topicname,msgtype)
pub = rospublisher( ____,Name,Value)
[pub,msg] = rospublisher( ____ )

pub = robotics.ros.Publisher(node,topicname)
pub = robotics.ros.Publisher(node,topicname,type)
pub = robotics.ros.Publisher( ____, "IsLatching",value)
```

### Description

`pub = rospublisher(topicname)` creates a publisher for a specific topic name and sets the `TopicName` property. The topic must already exist on the ROS master topic list with an established `MessageType`.

`pub = rospublisher(topicname,msgtype)` creates a publisher for a topic and adds that topic to the ROS master topic list. The inputs are set to the `TopicName` and `MessageType` properties of the publisher. If the topic already exists and `msgtype` differs from the topic type on the ROS master topic list, the function displays an error message.

`pub = rospublisher( ____,Name,Value)` provides additional options specified by one or more `Name,Value` pair arguments using any of the arguments from previous syntaxes. `Name` is the property name and `Value` is the corresponding value.

`[pub,msg] = rospublisher( ____ )` returns a message, `msg`, that you can send with the publisher, `pub`. The message is initialized with default values. You can also get the ROS message using the `rosmesssage` function.

`pub = robotics.ros.Publisher(node,topicname)` creates a publisher for a topic with name, `topicname`. `node` is the `robotics.ros.Node` object handle that this publisher attaches to. If `node` is specified as `[]`, the publisher tries to attach to the global node.

`pub = robotics.ros.Publisher(node,topicname,type)` creates a publisher with specified message type, `type`. If the topic already exists, MATLAB checks the message

type and displays an error if the input type differs. If the ROS master topic list already contains a matching topic, the ROS master adds the MATLAB global node to the list of publishers for that topic.

`pub = robotics.ros.Publisher( ____, "IsLatching", value)` specifies if the publisher is latching with a Boolean, `value`. If a publisher is latching, it saves the last sent message and sends it to any new subscribers. By default, `IsLatching` is enabled.

## Properties

### **TopicName — Name of the published topic**

string scalar | character vector

This property is read-only.

Name of the published topic, specified as a string scalar or character vector. If the topic does not exist, the object creates the topic using its associated message type.

Example: `"/chatter"`

Data Types: char

### **MessageType — Message type of published messages**

string scalar | character vector

This property is read-only.

Message type of published messages, specified as a string scalar or character vector. This message type remains associated with the topic and must be used for new messages published.

Example: `"std_msgs/String"`

Data Types: char

### **IsLatching — Indicator of whether publisher is latching**

true (default) | false

This property is read-only.

Indicator of whether publisher is latching, specified as `true` or `false`. A publisher that is latching saves the last sent message and resends it to any new subscribers.



Data Types: logical

### **NumSubscribers — Number of subscribers**

integer

This property is read-only.

Number of subscribers to the published topic, specified as an integer.

Data Types: double

## **Object Functions**

send            Publish ROS message to topic  
rosmessages    Create ROS messages

## **Examples**

### **Create a ROS Publisher and Send Data**

Connect to a ROS network.

```
rosinit
```

```
Initializing ROS master on http://bat5742win64:64143/.
```

```
Initializing global node /matlab_global_node_59264 with NodeURI http://bat5742win64:64143/
```

Create publisher for the '/ chatter' topic with the 'std\_msgs/String' message type.

```
chatpub = rospublisher('/ chatter', 'std_msgs/String');
```

Create a message to send. Specify the Data property.

```
msg = rosmessages(chatpub);  
msg.Data = 'test phrase';
```

Send message via the publisher.

```
send(chatpub, msg);
```

Shutdown ROS network.

```
roshutdown
```

```
Shutting down global node /matlab_global_node_59264 with NodeURI http://bat5742win64:64143/
Shutting down ROS master on http://bat5742win64:64143/.
```

## Create ROS Publisher with `rospublisher` and View Properties

Create a ROS publisher and view the associated properties for the `robotics.ros.Publisher` object. Add a subscriber to view the updated properties.

Start ROS master.

```
rosinit
```

```
Initializing ROS master on http://bat5742win64:61651/.
Initializing global node /matlab_global_node_22920 with NodeURI http://bat5742win64:61651/.
```

Create a publisher and view its properties.

```
pub = rospublisher('/chatter', 'std_msgs/String');
```

```
topic = pub.TopicName
```

```
topic =
'/chatter'
```

```
subCount = pub.NumSubscribers
```

```
subCount = 0
```

Subscribe to the publisher topic and view the changes in the `NumSubscribers` property.

```
sub = rossubscriber('/chatter');
pause(1)
```

```
subCount = pub.NumSubscribers
```

```
subCount = 1
```

```
roshutdown
```

```
Shutting down global node /matlab_global_node_22920 with NodeURI http://bat5742win64:61651/
Shutting down ROS master on http://bat5742win64:61651/.
```

## Publish Data Without A ROS Publisher

Connect to a ROS network.

```
rosinit
```

```
Initializing ROS master on http://bat5742win64:50543/.  
Initializing global node /matlab_global_node_70833 with NodeURI http://bat5742win64:50543/.
```

Create a message to send. Specify the Data property.

```
msg = rosmessage('std_msgs/String');  
msg.Data = 'test phrase';
```

Send message via the '/chatter' topic.

```
rospublisher('/chatter',msg)
```

```
ans =
```

```
    []
```

Shutdown ROS network.

```
roshutdown
```

```
Shutting down global node /matlab_global_node_70833 with NodeURI http://bat5742win64:50543/.  
Shutting down ROS master on http://bat5742win64:50543/.
```

## Use ROS Publisher Object

Create a Publisher object using the class constructor.

Start the ROS master.

```
master = robotics.ros.Core;
```

Create a ROS node, which connects to the master.

```
node = robotics.ros.Node('/test1');
```

Create a publisher and send string data. The publisher attaches to the node object in the first argument.

```
pub = robotics.ros.Publisher(node, '/robotname', 'std_msgs/String');  
msg = rosmesssage('std_msgs/String');  
msg.Data = 'robot1';  
send(pub,msg);
```

Clear the publisher and ROS node. Shut down the ROS master.

```
clear('pub','node')  
clear('master')
```

## See Also

### Functions

rosmesssage | send

### Topics

“Exchange Data with ROS Publishers and Subscribers”

**Introduced in R2015a**

## rostrate

Execute loop at fixed frequency

### Description

The `robotics.ros.Rate` object uses the `robotics.Rate` superclass to inherit most of its properties and methods. The main difference is that `robotics.ros.Rate` uses the ROS node as a source for time information. Therefore, it can use the ROS simulation or wall clock time (see the `IsSimulationTime` property).

If `rosinit` creates a ROS master in MATLAB, the global node uses wall clock time.

The performance of the `ros.Rate` object and the ability to maintain the `DesiredRate` value depends on the publishing of the clock information in ROS.

---

**Tip** The scheduling resolution of your operating system and the level of other system activity can affect rate execution accuracy. As a result, accurate rate timing is limited to 100 Hz for execution of MATLAB code. To improve performance and execution speeds, use code generation.

---

### Creation

### Syntax

```
rate = rostrate(desiredRate)
rate = robotics.ros.Rate(node,desiredRate)
```

### Description

`rate = rostrate(desiredRate)` creates a `robotics.ros.Rate` object, which enables you to execute a loop at a fixed frequency, `DesiredRate`. The time source is linked to the time source of the global ROS node, which requires you to connect MATLAB to a ROS network using `rosinit`.

`rate = robotics.ros.Rate(node, desiredRate)` creates a `Rate` object that operates loops at a fixed rate based on the time source linked to the specified ROS node, `node`.

## Properties

### **DesiredRate — Desired execution rate**

scalar

Desired execution rate of loop, specified as a scalar in Hz. When using `waitfor`, the loop operates every `DesiredRate` seconds, unless the loop takes longer. It then begins the next loop based on the specified `OverRunAction`.

### **DesiredPeriod — Desired time period between executions**

scalar

Desired time period between executions, specified as a scalar in seconds. This property is equal to the inverse of `DesiredRate`.

### **TotalElapsedTime — Elapsed time since construction or reset**

scalar

Elapsed time since construction or reset, specified as a scalar in seconds.

### **LastPeriod — Elapsed time between last two calls to waitfor**

NaN (default) | scalar

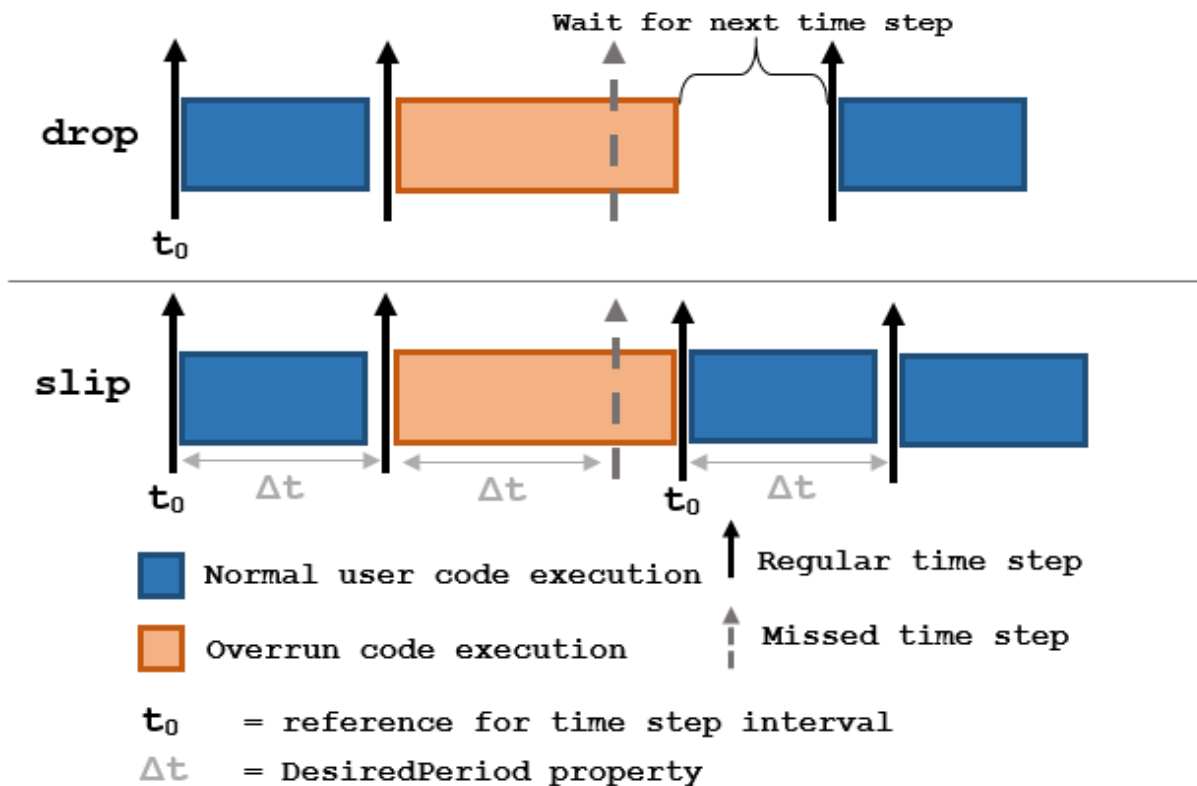
Elapsed time between last two calls to `waitfor`, specified as a scalar. By default, `LastPeriod` is set to NaN until `waitfor` is called for the first time. After the first call, `LastPeriod` equals `TotalElapsedTime`.

### **OverrunAction — Method for handling overruns**

'slip' (default) | 'drop'

Method for handling overruns, specified as one of these character vectors:

- 'drop' — waits until the next time interval equal to a multiple of `DesiredPeriod`
- 'slip' — immediately executes the loop again



Each code section calls waitfor at the end of execution.

**IsSimulationTime** – Indicator if simulation or wall clock time is used

true | false

Indicator if simulation or wall clock time is used, returned as true or false. If true, the Rate object is using the ROS simulation time to regulate the rate of loop execution.

## Object Functions

waitfor    Pause code execution to achieve desired execution rate  
 statistics    Statistics of past execution periods  
 reset        Reset Rate object

## Examples

### Run Loop At Fixed Rate Using `rosclock`

Initialize the ROS master and node.

```
rosinit
```

```
Initializing ROS master on http://bat5742win64:50750/.  
Initializing global node /matlab_global_node_43747 with NodeURI http://bat5742win64:50750/
```

Create a rate object that runs at 1 Hz.

```
r = rosclock(1);
```

Start loop that prints iteration and time elapsed. Use `waitfor` to pause the loop until the next time interval. Reset `r` prior to the loop execution. Notice that each iteration executes at a 1-second interval.

```
reset(r)  
for i = 1:10  
    time = r.TotalElapsedTime;  
    fprintf('Iteration: %d - Time Elapsed: %f\n',i,time)  
    waitfor(r);  
end
```

```
Iteration: 1 - Time Elapsed: 0.006459  
Iteration: 2 - Time Elapsed: 1.013668  
Iteration: 3 - Time Elapsed: 2.006047  
Iteration: 4 - Time Elapsed: 3.004924  
Iteration: 5 - Time Elapsed: 4.004910  
Iteration: 6 - Time Elapsed: 5.005591  
Iteration: 7 - Time Elapsed: 6.005486  
Iteration: 8 - Time Elapsed: 7.004376  
Iteration: 9 - Time Elapsed: 8.004269  
Iteration: 10 - Time Elapsed: 9.004932
```

Shut down the ROS network.

```
roscleanup
```

```
Shutting down global node /matlab_global_node_43747 with NodeURI http://bat5742win64:50750/  
Shutting down ROS master on http://bat5742win64:50750/.
```



## Run Loop At Fixed Rate Using ROS Time

Initialize the ROS master and node.

```
rosinit
```

```
Initializing ROS master on http://bat5742win64:59977/.
```

```
Initializing global node /matlab_global_node_68058 with NodeURI http://bat5742win64:59977/.
```

```
node = robotics.ros.Node('/testTime');
```

```
Using Master URI http://localhost:59977 from the global node to connect to the ROS master.
```

Create a `ros.Rate` object running at 20 Hz.

```
r = robotics.ros.Rate(node,20);
```

Reset the object to restart the timer and run the loop for 30 iterations. Insert code you want to run in the loop before calling `waitfor`.

```
reset(r)
for i = 1:30
    % User code goes here.
    waitfor(r);
end
```

Shutdown ROS node.

```
roshutdown
```

```
Shutting down global node /matlab_global_node_68058 with NodeURI http://bat5742win64:59977/.
```

```
Shutting down ROS master on http://bat5742win64:59977/.
```

## See Also

`robotics.Rate` | `waitfor`

## Topics

“Execute Code at a Fixed-Rate”

**Introduced in R2016a**

# rossubscriber

Subscribe to messages on a topic

## Description

Use `rossubscriber` to create a ROS subscriber for receiving messages on the ROS network. To send messages, use `rospublisher`. To wait for a new ROS message, use the `receive` function with your created subscriber.

The `Subscriber` object created by the `rossubscriber` function represents a subscriber on the ROS network. The `Subscriber` object subscribes to an available topic or to a topic that it creates. This topic has an associated message type. Publishers can send messages over the network that the `Subscriber` object receives.

You can create a `Subscriber` object by using the `rossubscriber` function, or by calling `robotics.ros.Subscriber`:

- `rossubscriber` only works with the global node using `roslint`. It does not require a node object handle as an argument.
- `robotics.ros.Subscriber` works with additional nodes that are created using `robotics.ros.Node`. It requires a node object handle as the first argument.

## Creation

### Syntax

```
sub = rossubscriber(topicname)
sub = rossubscriber(topicname,msgtype)
sub = rossubscriber(topicname,callback)
sub = rossubscriber(topicname, msgtype,callback)
sub = rossubscriber( __ ,Name,Value)

sub = robotics.ros.Subscriber(node,topicname)
sub = robotics.ros.Subscriber(node,topicname,msgtype)
```

```
sub = robotics.ros.Subscriber(node,topicname,callback)
sub = robotics.ros.Subscriber(node,topicname,type,callback)
sub = robotics.ros.Subscriber( __ , "BufferSize",value)
```

## Description

`sub = rossubscriber(topicname)` subscribes to a topic with the given `TopicName`. The topic must already exist on the ROS master topic list with an established message type. When ROS nodes publish messages on that topic, MATLAB receives those messages through this subscriber.

`sub = rossubscriber(topicname,msgtype)` subscribes to a topic that has the specified name, `TopicName`, and type, `MessageType`. If the topic list on the ROS master does not include a topic with that specified name and type, it is added to the topic list. Use this syntax to avoid errors when subscribing to a topic before a publisher has added the topic to the topic list on the ROS master.

`sub = rossubscriber(topicname,callback)` specifies a callback function, `callback` that runs when the subscriber object handle receives a topic message. Use this syntax to avoid the blocking receive function. `callback` can be a single function handle or a cell array. The first element of the cell array must be a function handle or a string containing the name of a function. The remaining elements of the cell array can be arbitrary user data that is passed to the callback function.

`sub = rossubscriber(topicname, msgtype,callback)` specifies a callback function and subscribes to a topic that has the specified name, `TopicName`, and type, `MessageType`.

`sub = rossubscriber( __ ,Name,Value)` provides additional options specified by one or more `Name, Value` pair arguments using any of the arguments from previous syntaxes. `Name` is the property name and `Value` is the corresponding value.

`sub = robotics.ros.Subscriber(node,topicname)` subscribes to a topic with name, `TopicName`. `node` is the `robotics.ros.Node` object handle that this publisher attaches to.

`sub = robotics.ros.Subscriber(node,topicname,msgtype)` specifies the message type, `MessageType`, of the topic. If a topic with the same name exists with a different message type, MATLAB creates a new topic with the given message type.

`sub = robotics.ros.Subscriber(node, topicname, callback)` specifies a callback function, and optional data, to run when the subscriber object receives a topic message. See `NewMessageFcn` for more information about the callback function.

`sub = robotics.ros.Subscriber(node, topicname, type, callback)` specifies the topic name, message type, and callback function for the subscriber.

`sub = robotics.ros.Subscriber( ____, "BufferSize", value)` specifies the queue size in `BufferSize` for incoming messages. You can use any combination of previous inputs with this syntax.

## Properties

### **TopicName — Name of the subscribed topic**

string scalar | character vector

This property is read-only.

Name of the subscribed topic, specified as a string scalar or character vector. If the topic does not exist, the object creates the topic using its associated message type.

Example: `"/chatter"`

Data Types: `char` | `string`

### **MessageType — Message type of subscribed messages**

string scalar | character vector

This property is read-only.

Message type of subscribed messages, specified as a string scalar or character vector. This message type remains associated with the topic.

Example: `"std_msgs/String"`

Data Types: `char` | `string`

### **LatestMessage — Latest message sent to the topic**

Message object

Latest message sent to the topic, specified as a `Message` object. The `Message` object is specific to the given `MessageType`. If the subscriber has not received a message, then the `Message` object is empty.

## BufferSize — Buffer size

1 (default) | scalar

Buffer size of the incoming message queue, specified as the comma-separated pair consisting of "BufferSize" and a scalar. If messages arrive faster and than your callback can process them, they are deleted once the incoming queue is full.

## NewMessageFcn — Callback property

function handle | cell array

Callback property, specified as a function handle or cell array. In the first element of the cell array, specify either a function handle or a string representing a function name. In subsequent elements, specify user data.

The subscriber callback function requires at least two input arguments. The first argument, `src`, is the associated subscriber object. The second argument, `msg`, is the received message object. The function header for the callback is:

```
function subCallback(src,msg)
```

Specify the `NewMessageFcn` property as:

```
sub.NewMessageFcn = @subCallback;
```

When setting the callback, you pass additional parameters to the callback function by including both the callback function and the parameters as elements of a cell array. The function header for the callback is:

```
function subCallback(src,msg,userData)
```

Specify the `NewMessageFcn` property as:

```
sub.NewMessageFcn = {@subCallback,userData};
```

## Object Functions

`receive`      Wait for new ROS message  
`rosmessage`    Create ROS messages

## Examples

## Create A Subscriber and Get Data From ROS

Connect to a ROS network. Set up a sample ROS network. The '/scan' topic is being published on the network.

```
rosinit
```

```
Initializing ROS master on http://bat5742win64:64693/.
```

```
Initializing global node /matlab_global_node_49772 with NodeURI http://bat5742win64:64693/.
```

```
exampleHelperROSCreateSampleNetwork
```

Create a subscriber for the '/scan' topic. Wait for the subscriber to register with the master.

```
sub = rossubscriber('/scan');
pause(1);
```

Receive data from the subscriber as a ROS message. Specify a 10 second timeout.

```
msg2 = receive(sub,10)
```

```
msg2 =
```

```
ROS LaserScan message with properties:
```

```
    MessageType: 'sensor_msgs/LaserScan'
           Header: [1x1 Header]
           AngleMin: -0.5216
           AngleMax: 0.5243
AngleIncrement: 0.0016
TimeIncrement: 0
           ScanTime: 0.0330
           RangeMin: 0.4500
           RangeMax: 10
           Ranges: [640x1 single]
Intensities: [0x1 single]
```

Use showdetails to show the contents of the message

Shutdown ROS network.

```
rosshutdown
```

```
Shutting down global node /matlab_global_node_49772 with NodeURI http://bat5742win64:64693/.
```

```
Shutting down ROS master on http://bat5742win64:64693/.
```

### Create A Subscriber That Uses A Callback Function

You can trigger callback functions when subscribers receive messages. Specify the callback when you create it or use the `NewMessageFcn` property.

Connect to a ROS network.

```
rosinit
```

```
Initializing ROS master on http://bat5742win64:61499/.  
Initializing global node /matlab_global_node_12736 with NodeURI http://bat5742win64:61499/
```

Setup a publisher to publish a message to the  `'/chatter'`  topic. This topic is used to trigger the subscriber callback. Specify the `Data` property of the message. Wait 1 second to allow the publisher to register with the network.

```
pub = rospublisher('/chatter','std_msgs/String');  
msg = rosmessage(pub);  
msg.Data = 'hello world';  
pause(1)
```

Setup a subscriber with a specified callback function. The `exampleHelperROSchatterCallback` function displays the `Data` inside the received message.

```
sub = rossubscriber('/chatter',@exampleHelperROSchatterCallback);  
pause(1)
```

Send message via the publisher. The subscriber should execute the callback to display the new message. Wait for the message to be received.

```
send(pub,msg);  
pause(1)
```

```
ans =  
'hello world'
```

Shutdown ROS network.

```
roshutdown
```

```
Shutting down global node /matlab_global_node_12736 with NodeURI http://bat5742win64:61499/  
Shutting down ROS master on http://bat5742win64:61499/.
```



## Use ROS Subscriber Object

Use a ROS Subscriber object to receive messages over the ROS network.

Start the ROS master and node.

```
master = robotics.ros.Core;
node = robotics.ros.Node('/test');
```

Create a publisher and subscriber to send and receive a message over the ROS network.

```
pub = robotics.ros.Publisher(node, '/chatter', 'std_msgs/String');
pause(1)
sub = robotics.ros.Subscriber(node, '/chatter', 'std_msgs/String');
```

Send a message over the network.

```
msg = rosmesssage('std_msgs/String');
msg.Data = 'hello world';
send(pub, msg)
```

View the message data using the LatestMessage property of the Subscriber object.

```
pause(1)
sub.LatestMessage
```

```
ans =
  ROS String message with properties:
```

```
  MessageType: 'std_msgs/String'
  Data: 'hello world'
```

Use showdetails to show the contents of the message

Clear the publisher, subscriber, and ROS node. Shut down the ROS master.

```
clear('pub', 'sub', 'node')
clear('master')
```

## **See Also**

receive | rosmesssage | rospublisher

## **Topics**

“Exchange Data with ROS Publishers and Subscribers”

**Introduced in R2015a**

# rossvcclient

Connect to ROS service server

## Description

Use `rossvcclient` or `robotics.ros.ServiceClient` to create a ROS service client object. This service client uses a persistent connection to send requests to, and receive responses from, a ROS service server. The connection persists until the service client is deleted or the service server becomes unavailable.

Use the `robotics.ros.ServiceClient` syntax when connecting to a specific ROS node.

## Creation

### Syntax

```
client = rossvcclient(servicename)
client = rossvcclient(servicename,Name,Value)
```

```
[client,reqmsg] = rossvcclient( ___ )
```

```
client = robotics.ros.ServiceClient(node, name)
client = robotics.ros.ServiceClient(node, name, "Timeout", timeout)
```

### Description

`client = rossvcclient(servicename)` creates a service client with the given `ServiceName` that connects to, and gets its `ServiceType` from, a service server. This command syntax blocks the current MATLAB program from running until it can connect to the service server.

`client = rossvcclient(servicename,Name,Value)` provides additional options specified by one or more `Name, Value` pair arguments.

`[client, reqmsg] = rossvcclient( ___ )` returns a new service request message in `reqmsg`, using any of the arguments from previous syntaxes. The message type of `reqmsg` is determined by the service that `client` is connected to. The message is initialized with default values. You can also create the request message using `rosmessage wutg`.

`client = robotics.ros.ServiceClient(node, name)` creates a service client that connects to a service server. The client gets its service type from the server. The service client attaches to the `robotics.ros.Node` object handle, `node`.

`client = robotics.ros.ServiceClient(node, name, "Timeout", timeout)` specifies a timeout period in seconds for the client to connect the service server.

## Properties

### **ServiceName — Name of the service**

string scalar | character vector

This property is read-only.

Name of the service, specified as a string scalar or character vector.

Example: `"/gazebo/get_model_state"`

### **ServiceType — Type of service**

string scalar | character vector

This property is read-only.

Type of service, specified as a string scalar or character vector.

Example: `"gazebo_msgs/GetModelState"`

## Object Functions

`rosmessage` Create ROS messages

`call` Call the ROS service server and receive a response

## Examples

## Call Service Client with Default Message

Connect to a ROS network.

```
rosinit
```

```
Initializing ROS master on http://ah-sradford:11311/.  
Initializing global node /matlab_global_node_03934 with NodeURI http://ah-sradford:6130
```

Set up a service server and client.

```
server = rossvcserver('/test', 'std_srvs/Empty', @exampleHelperROSEmptyCallback);  
client = rossvcclient('/test');
```

Call service server with default message.

```
response = call(client)
```

A service client is calling

```
response =
```

```
ROS EmptyResponse message with properties:
```

```
  MessageType: 'std_srvs/EmptyResponse'
```

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

Shut down ROS network.

```
rosshutdown
```

```
Shutting down global node /matlab_global_node_03934 with NodeURI http://ah-sradford:6130  
Shutting down ROS master on http://ah-sradford:11311/.
```

## Use ROS Service Server with ServiceServer and ServiceClient Objects

Create a ROS service server by creating a `ServiceServer` object and use `ServiceClient` objects to request information over the network. The callback function used by the server takes a string, reverses it, and returns the reversed string.

Start the ROS master and node.

```
master = robotics.ros.Core;  
node = robotics.ros.Node('/test');
```

Create a service server. This server expects a string as a request and responds with a string based on the callback.

```
server = robotics.ros.ServiceServer(node, '/data/string', ...  
                                   'roseus/StringString');
```

Create a callback function. This function takes an input string as the `Str` property of `req` and returns it as the `Str` property of `resp`. You must create and save this function separately. `req` is a ROS message you create using `rosmessage`.

```
% Copyright 2015 The MathWorks, Inc.
```

```
function [resp] = flipString(~,req,resp)  
% FLIPSTRING Reverses the order of a string in REQ and returns it in RESP.  
resp.Str = fliplr(req.Str);  
end
```

Save this code as a file named `flipString.m` to a folder on your MATLAB® path.

Assign the callback function for incoming service calls.

```
server.NewRequestFcn = @flipString;
```

Create a service client and connect to the service server. Create a request message based on the client.

```
client = robotics.ros.ServiceClient(node, '/data/string');  
request = rosmessage(client);  
request.Str = 'hello world';
```

Send a service request and wait for a response. Specify that the service waits 3 seconds for a response.

```
response = call(client,request,'Timeout',3)
```

```
response =
```

ROS StringStringResponse message with properties:

```
MessageType: 'roseus/StringStringResponse'  
  Str: 'dlrow olleh'
```

Use `showdetails` to show the contents of the message

The response is a flipped string from the request message.

Clear the service client, service server, and ROS node. Shut down the ROS master.

```
clear('client', 'server', 'node')  
clear('master')
```

## See Also

[call](#) | [rosmessage](#) | [rosservice](#) | [rossvcserver](#)

## Topics

“Call and Provide ROS Services”

**Introduced in R2015a**

## rossvcserver

Create ROS service server

### Description

Use `rossvcserver` or `robotics.ros.ServiceServer` to create a ROS service server that can receive requests from, and send responses to, a ROS service client. You must create the service server before creating the service client (see `ROSSVCCLIENT`).

When you create the service client, it establishes a connection to the server. The connection persists while both client and server exist and can reach each other. When you create the service server, it registers itself with the ROS master. To get a list of services, or to get information about a particular service that is available on the current ROS network, use the `rosservice` function.

The service has an associated message type and contains a pair of messages: one for the request and one for the response. The service server receives a request, constructs an appropriate response based on a call function, and returns it to the client. The behavior of the service server is inherently asynchronous, because it becomes active only when a service client connects to the ROS network and issues a call.

Use the `robotics.ros.ServiceServer` syntax when connecting to a specific ROS node.

### Creation

### Syntax

```
server = rossvcserver(servicename,svctype)
server = rossvcserver(servicename,svctype,callback)

server = robotics.ros.ServiceServer(node, name,type)
server = robotics.ros.ServiceServer(node, name,type,callback)
```



## Description

`server = rossvcserver(servicename, svctype)` creates a service server object with the specified `ServiceType` available in the ROS network under the name `ServiceName`. The service object cannot respond to service requests until you specify a function handle `callback`, `NewMessageFcn`.

`server = rossvcserver(servicename, svctype, callback)` specifies the callback function that constructs a response when the server receives a request. `callback` specifies the `NewMessageFcn` property.

`server = robotics.ros.ServiceServer(node, name, type)` creates a service server that attaches to the ROS node, `node`. The server becomes available through the specified service name and type once a callback function handle is specified in `NewMessageFcn`.

`server = robotics.ros.ServiceServer(node, name, type, callback)` specifies the callback function which is set to the `NewMessageFcn` property.

## Properties

### ServiceName — Name of the service

string scalar | character vector

This property is read-only.

Name of the service, specified as a string scalar or character vector.

Example: `"/gazebo/get_model_state"`

Data Types: `char` | `string`

### ServiceType — Type of service

string scalar | character vector

This property is read-only.

Type of service, specified as a string scalar or character vector.

Example: `"gazebo_msgs/GetModelState"`

Data Types: `char` | `string`

**NewMessageFcn — Callback property**

function handle | cell array

Callback property, specified as a function handle or cell array. In the first element of the cell array, specify either a function handle, string scalar, or character vector representing a function name. In subsequent elements, specify user data.

The service callback function requires at least three input arguments with one output. The first argument, `src`, is the associated service server object. The second argument, `reqMsg`, is the request message object sent by the service client. The third argument is the default response message object, `defaultRespMsg`. The callback returns a response message, `response`, based on the input request message and sends it back to the service client. Use the default response message as a starting point for constructing the request message. The function header for the callback is:

```
function response = serviceCallback(src, reqMsg, defaultRespMsg)
```

Specify the `NewMessageFcn` property as:

```
server.NewMessageFcn = @serviceCallback;
```

When setting the callback, you pass additional parameters to the callback function by including both the callback function and the parameters as elements of a cell array. The function header for the callback is:

```
function response = serviceCallback(src, reqMsg, defaultRespMsg, userData)
```

Specify the `NewMessageFcn` property as:

```
server.NewMessageFcn = {@serviceCallback, userData};
```

## Object Functions

`rosmessage` Create ROS messages

## Examples

### Call Service Client with Default Message

Connect to a ROS network.

```
rosinit
```

```
Initializing ROS master on http://ah-sradford:11311/.
```

```
Initializing global node /matlab_global_node_03934 with NodeURI http://ah-sradford:6130/.
```

Set up a service server and client.

```
server = rossvcserver('/test', 'std_srvs/Empty', @exampleHelperROSEmptyCallback);  
client = rossvcclient('/test');
```

Call service server with default message.

```
response = call(client)
```

A service client is calling

```
response =
```

```
ROS EmptyResponse message with properties:
```

```
  MessageType: 'std_srvs/EmptyResponse'
```

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

Shut down ROS network.

```
rosshutdown
```

```
Shutting down global node /matlab_global_node_03934 with NodeURI http://ah-sradford:6130/.
```

```
Shutting down ROS master on http://ah-sradford:11311/.
```

## See Also

[call](#) | [rosmessage](#) | [rossvcclient](#)

## Topics

“Call and Provide ROS Services”

**Introduced in R2015a**

## rostopic

Receive, send, and apply ROS transformations

### Description

Calling `rostopic` creates a ROS `TransformationTree` object, which allows you to access the `tf` coordinate transformations that are shared on the ROS network. You can receive transformations and apply them to different entities. You can also send transformations and share them with the rest of the ROS network.

ROS uses the `tf` transform library to keep track of the relationship between multiple coordinate frames. The relative transformations between these coordinate frames is maintained in a tree structure. Querying this tree lets you transform entities like poses and points between any two coordinate frames. To access available frames use the syntax:

```
tfTree.AvailableFrames
```

Use the `robotics.ros.TransformationTree` syntax when connecting to a specific ROS node, otherwise use `rostopic` to create the transformation tree.

### Creation

#### Syntax

```
tfTree = rostopic
```

```
trtree = robotics.ros.TransformationTree(node)
```

#### Description

`tfTree = rostopic` creates a ROS `TransformationTree` object.

`trtree = robotics.ros.TransformationTree(node)` creates a ROS transformation tree object handle that the transformation tree is attached to. `node` is the node connected to the ROS network that publishes transformations.

## Properties

### AvailableFrames — List of all available coordinate frames

cell array

This property is read-only.

List of all available coordinate frames, specified as a cell array. This list of available frames updates if new transformations are received by the transformation tree object.

Example: {'camera\_center'; 'mounting\_point'; 'robot\_base'}

Data Types: cell

### LastUpdateTime — Time when the last transform was received

ROS Time object

This property is read-only.

Time when the last transform was received, specified as a ROS Time object.

### BufferTime — Length of time transformations are buffered

10 (default) | scalar

This property is read-only.

Length of time transformations are buffered, specified as a scalar in seconds. If you change the buffer time from the current value, the transformation tree and all transformations are reinitialized. You must wait the entire buffer time to get a fully buffered transformation tree.

## Object Functions

waitForTransform	Wait until a transformation is available
getTransform	Retrieve transformation between two coordinate frames
transform	Transform message entities into target coordinate frame
sendTransform	Send transformation to ROS network

## Examples

## Create a ROS Transformation Tree

Connect to a ROS network and create a transformation tree.

Connect to a ROS network. Specify the IP address.

```
rosinit('192.168.203.129')
```

```
Initializing global node /matlab_global_node_92595 with NodeURI http://192.168.203.1:64
```

Create a transformation tree. Use the AvailableFrames property to see the transformation frames available. These transformations were specified separately prior to connecting to the network.

```
tree = rostf;  
pause(1);  
tree.AvailableFrames
```

```
ans =
```

```
36×1 cell array
```

```
{'base_footprint'      }  
{'base_link'          }  
{'camera_depth_frame'}  
{'camera_depth_optical_frame'}  
{'camera_link'       }  
{'camera_rgb_frame'  }  
{'camera_rgb_optical_frame'}  
{'caster_back_link'  }  
{'caster_front_link' }  
{'cliff_sensor_front_link'}  
{'cliff_sensor_left_link'}  
{'cliff_sensor_right_link'}  
{'gyro_link'         }  
{'mount_asus_xtion_pro_link'}  
{'odom'              }  
{'plate_bottom_link' }  
{'plate_middle_link' }  
{'plate_top_link'    }  
{'pole_bottom_0_link'}  
{'pole_bottom_1_link'}  
{'pole_bottom_2_link'}  
{'pole_bottom_3_link' }
```

```

{'pole_bottom_4_link'      }
{'pole_bottom_5_link'      }
{'pole_kinect_0_link'      }
{'pole_kinect_1_link'      }
{'pole_middle_0_link'      }
{'pole_middle_1_link'      }
{'pole_middle_2_link'      }
{'pole_middle_3_link'      }
{'pole_top_0_link'         }
{'pole_top_1_link'         }
{'pole_top_2_link'         }
{'pole_top_3_link'         }
{'wheel_left_link'         }
{'wheel_right_link'        }

```

Disconnect from the ROS network.

```
rosshutdown
```

```
Shutting down global node /matlab_global_node_92595 with NodeURI http://192.168.203.1:0
```

## Use TransformationTree Object

Create a ROS transformation tree. You can then view or use transformation information for different coordinate frames setup in the ROS network.

Start ROS network and broadcast sample transformation data.

```
rosinit
```

```
Initializing ROS master on http://ah-sradford:11311/.
```

```
Initializing global node /matlab_global_node_75001 with NodeURI http://ah-sradford:522
```

```
node = robotics.ros.Node('/testTf');
```

```
Using Master URI http://localhost:11311 from the global node to connect to the ROS master
```

```
exampleHelperROSstartTfPublisher
```

Retrieve the TransformationTree object. Pause to wait for tftree to update.

```
tftree = robotics.ros.TransformationTree(node);
pause(1)
```

View available coordinate frames and the time when they were last received.

```
frames = tftree.AvailableFrames
```

```
frames = 3x1 cell array
  {'camera_center' }
  {'mounting_point'}
  {'robot_base'   }
```

```
updateTime = tftree.LastUpdateTime
```

```
updateTime =
  ROS Time with properties:
```

```
    Sec: 1.5121e+09
    Nsec: 262000000
```

Wait for the transform between two frames, 'camera\_center' and 'robot\_base'. This will wait until the transformation is valid and block all other operations. A time out of 5 seconds is also given.

```
waitForTransform(tftree, 'robot_base', 'camera_center', 5)
```

Define a point in the camera's coordinate frame.

```
pt = rosmesssage('geometry_msgs/PointStamped');
pt.Header.FrameId = 'camera_center';
pt.Point.X = 3;
pt.Point.Y = 1.5;
pt.Point.Z = 0.2;
```

Transform the point into the 'base\_link' frame.

```
tfpt = transform(tftree, 'robot_base', pt)
```

```
tfpt =
  ROS PointStamped message with properties:
```

```
    MessageType: 'geometry_msgs/PointStamped'
    Header: [1x1 Header]
    Point: [1x1 Point]
```

Use `showdetails` to show the contents of the message



Display the transformed point coordinates.

```
tfpt.Point
```

```
ans =  
  ROS Point message with properties:  
  
  MessageType: 'geometry_msgs/Point'  
             X: 1.2000  
             Y: 1.5000  
             Z: -2.5000
```

Use `showdetails` to show the contents of the message

Clear ROS node. Shut down ROS master.

```
clear('node')  
roshutdown
```

```
Shutting down global node /matlab_global_node_75001 with NodeURI http://ah-sradford:52  
Shutting down ROS master on http://ah-sradford:11311/.
```

## See Also

`getTransform` | `sendTransform` | `transform` | `waitForTransform`

## Topics

“Access the tf Transformation Tree in ROS”

**Introduced in R2015a**

## rostime

Access ROS time functionality

### Description

A ROS Time object representing an instance of time in seconds and nanoseconds. This time can be based off your system time, the ROS simulation time, or an arbitrary time.

### Creation

### Syntax

```
time = rostime(totalSecs)
time = rostime(secs,nsecs)

time = rostime("now")
[time,issimtime] = rostime("now")
time = rostime("now","system")
```

### Description

`time = rostime(totalSecs)` initializes the time values for seconds and nanoseconds based on `totalSecs`, which represents the time in seconds as a floating-point number.

`time = rostime(secs,nsecs)` initializes the time values for seconds and nanoseconds individually. Both inputs must be integers. Large values for `nsecs` are wrapped automatically with the remainder added to `secs`.

`time = rostime("now")` returns the current ROS time. If the `use_sim_time` ROS parameter is set to `true`, the `rostime` returns the simulation time published on the `clock` topic. Otherwise, the function returns the system time of your machine. `time` is a ROS Time object. If no output argument is given, the current time (in seconds) is printed to the screen.

rostime can be used to timestamp messages or to measure time in the ROS network.

`[time,issimtime] = rostime("now")` also returns a Boolean that indicates if `time` is in simulation time (`true`) or system time (`false`).

`time = rostime("now", "system")` always returns the system time of your machine, even if ROS publishes simulation time on the `clock` topic. If no output argument is given, the system time (in seconds) is printed to the screen.

The system time in ROS follows the Unix or POSIX time standard. POSIX time is defined as the time that has elapsed since 00:00:00 Coordinated Universal Time (UTC), 1 January 1970, not counting leap seconds.

## Properties

### **totalSecs** — Total time

0 (default) | scalar

Total time, specified as a floating-point scalar. The integer portion is set to the `Sec` property with the remainder applied to `Nsec` property of the `Time` object.

### **Sec** — Whole seconds

0 (default) | positive integer

Whole seconds, specified as a positive integer.

---

**Note** The maximum and minimum values for `secs` are `[0, 4294967294]`.

---

### **Nsec** — Nanoseconds

0 (default) | positive integer

Nanoseconds, specified as a positive integer. If this value is greater than or equal to  $10^9$ , then the value is then wrapped and the remainders are added to the value of `Sec`.

## Examples

## Get Current ROS Time

Connect to a ROS network.

```
rosinit
```

```
Initializing ROS master on http://AH-SRADFORD:11311/.
```

```
Initializing global node /matlab_global_node_00466 with NodeURI http://AH-SRADFORD:643
```

Get current ROS Time. You can also check whether is it system time by getting the `issim` output.

```
[t,issim] = rostime('now')
```

```
t =
```

```
ROS Time with properties:
```

```
Sec: 1.4734e+09
```

```
Nsec: 408000000
```

```
issim =
```

```
logical
```

```
0
```

## Timestamp ROS Message Data

Create a stamped ROS message. Specify the `Header.Stamp` property with the current system time.

```
point = rosmesssage('geometry_msgs/PointStamped');
```

```
point.Header.Stamp = rostime('now','system');
```

## ROS Time to MATLAB Time Example

This example shows how to convert current ROS time into a MATLAB® standard time. The ROS Time object is first converted to a double in seconds, then to the specified MATLAB time.

```
% Sets up ROS network and stores ROS time
rosinit

Initializing ROS master on http://bat5742win64:49927/.
Initializing global node /matlab_global_node_41439 with NodeURI http://bat5742win64:49927/

t = rostime('now');

% Converts ROS time to a double in seconds
secondtime = double(t.Sec)+double(t.Nsec)*10^-9;

% Sets time to a specified MATLAB format
time = datetime(secondtime, 'ConvertFrom','posixtime')

time = datetime
    02-Mar-2019 16:18:20

% Shuts down ROS network
rosshutdown

Shutting down global node /matlab_global_node_41439 with NodeURI http://bat5742win64:49927/
Shutting down ROS master on http://bat5742win64:49927/.
```

## Get Seconds From A Time Object

Use the `seconds` function to get the total seconds of a `Time` object from its `Secs` and `Nsecs` properties.

Create a `Time` object.

```
time = rostime(1,860000000)

time =
    ROS Time with properties:

        Sec: 1
       Nsec: 860000000
```

Get the total seconds from the time object.

```
secs = seconds(time)
```

```
secs = 1.8600
```

## See Also

[rosduration](#) | [rosmesssage](#) | [seconds](#)

**Introduced in R2015a**

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

## 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 (MATLAB).

### **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

## **Usage**

## **Syntax**

```
[lookaheadPoint,desiredHeading,desiredYaw,crossTrackError,numTurns]  
= orbit(currentPose,lookaheadDistance)
```

## **Description**

[lookaheadPoint,desiredHeading,desiredYaw,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 heading, 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 heading] vector

Current UAV pose, specified as a [x y z heading] vector. This pose is used to calculate the lookahead point based on the input LookaheadDistance. [x y z] is the current position in meters. heading is the current heading in radians. The UAV heading 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`

### **desiredHeading — Desired heading**

numeric scalar

Desired heading, returned as numeric scalar in radians in the range of  $[-\pi, \pi]$ . The UAV heading is the angle of direction of the velocity vector relative to north measured in radians.

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.

Data Types: `double`

### **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

<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 Control Commands for Orbit Following

This example shows how to use the `uavOrbitFollower` to generate heading 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 heading, and yaw. You can use the desired heading and yaw to generate control commands for the UAV.

```
[lookaheadPoint,desiredHeading,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](#) | [roboticsAddons](#) | [state](#)

### Objects

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

### Blocks

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

### Introduced in R2019a

# 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 heading, 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? \(MATLAB\)](#).

## 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 (MATLAB).

### **UAV type — Type of UAV**

'fixed-wing' (default) | 'multicopter'

Type of UAV, specified as either 'fixed-wing' or 'multicopter'.

### **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 heading 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`

## **Usage**

## **Syntax**

```
[lookaheadPoint,desiredHeading,desiredYaw,crossTrackError,status] =  
wpFollowerObj(currentPose,lookaheadDistance)
```

## **Description**

`[lookaheadPoint,desiredHeading,desiredYaw,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 heading, 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  $[x \ y \ z \ \text{chi}]$  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 heading 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: `single` | `double`

### **desiredHeading — Desired heading**

numeric scalar

Desired heading, returned as a numeric scalar in radians in the range of  $[-\pi, \pi]$ . The UAV heading is the direction of the velocity vector.

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.

Data Types: `single` | `double`

### **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)
```

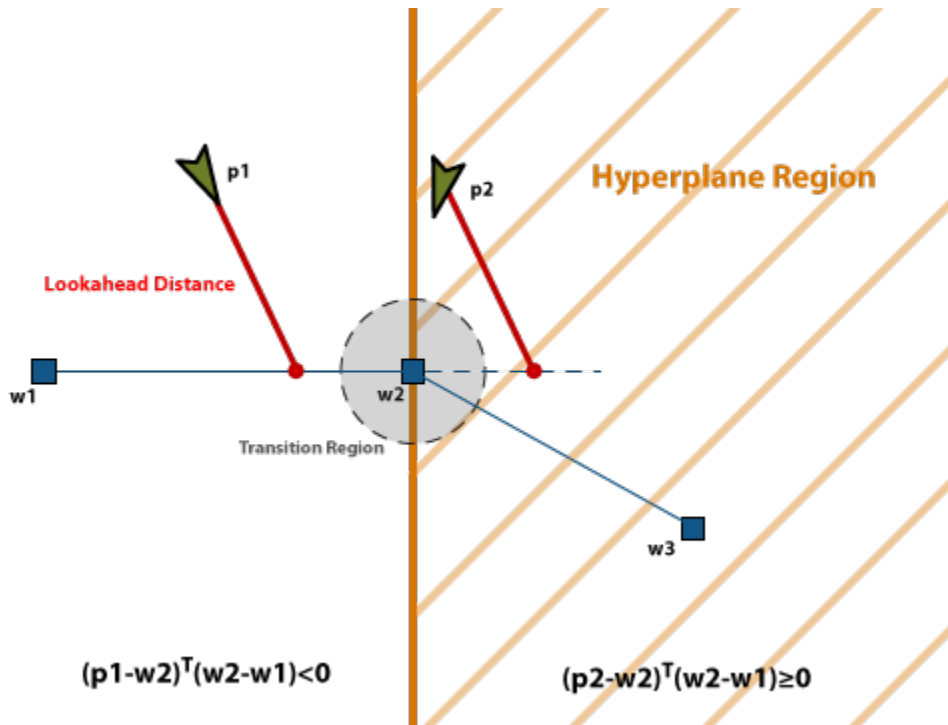
### **Common to All System Objects**

<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

## **Definitions**

### **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:

$$(p - w_1)^T (w_2 - w_1) \geq 0$$

$p$  is the UAV position, and  $w_1$  and  $w_2$  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` | `roboticsAddons` | `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**

## robotics.VectorFieldHistogram

**Package:** robotics

Avoid obstacles using vector field histogram

### Description

The `robotics.VectorFieldHistogram` System object enables your robot to avoid obstacles based on range sensor data using vector field histograms (VFH). Given laser scan readings and a target direction to drive toward, the object computes an obstacle-free steering direction.

`VectorFieldHistogram` specifically uses the VFH+ algorithm to compute an obstacle-free direction. First, the algorithm takes the ranges and angles from laser scan data and builds a polar histogram for obstacle locations. Then, the input histogram thresholds are used to calculate a binary histogram that indicates occupied and free directions. Finally, the algorithm computes a masked histogram, which is computed from the binary histogram based on the minimum turning radius of the robot.

The algorithm selects multiple steering directions based on the open space and possible driving directions. A cost function, with weights corresponding to the previous, current, and target directions, calculates the cost of different possible directions. The object then returns an obstacle-free direction with minimal cost. Using the obstacle-free direction, you can input commands to move your robot in that direction.

To use this object for your own application and environment, you must tune the properties of the algorithm. Property values depend on the type of robot, the range sensor, and the hardware you use.

To find an obstacle-free steering direction:

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

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

## Creation

## Syntax

```
VFH = robotics.VectorFieldHistogram  
VFH = robotics.VectorFieldHistogram(Name,Value)
```

## Description

`VFH = robotics.VectorFieldHistogram` returns a vector field histogram object that computes the obstacle-free steering direction using the VFH+ algorithm.

`VFH = robotics.VectorFieldHistogram(Name,Value)` returns a vector field histogram object with additional options specified by one or more `Name, Value` pairs. `Name` is the property name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name-value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`. Properties not specified retain their default values.

## Properties

### **NumAngularSectors** — Number of angular sectors in histogram

180 (default) | positive integer

Number of angular sectors in the vector field histogram, specified as a scalar. This property defines the number of bins used to create the histograms. This property is non-tunable. You can only set this when the object is initialized.

### **DistanceLimits** — Limits for range readings

[0.05 2] (default) | 2-element vector

Limits for range readings, specified as a 2-element vector with elements measured in meters. The range readings specified when calling the object are considered only if they fall within the distance limits. Use the lower distance limit to ignore false positives from poor sensor performance at lower ranges. Use the upper limit to ignore obstacles that are too far from the robot.

## **RobotRadius — Radius of robot**

0.1 (default) | scalar

Radius of the robot in meters, specified as a scalar. This dimension defines the smallest circle that can circumscribe your robot. The robot radius is used to account for robot size when computing the obstacle-free direction.

## **SafetyDistance — Safety distance around robot**

0.1 (default) | scalar

Safety distance around the robot, specified as a scalar in meters. This is a safety distance to leave around the robot position in addition to the value of the `RobotRadius` parameter. The sum of the robot radius and the safety distance is used to compute the obstacle-free direction.

## **MinTurningRadius — Minimum turning radius at current speed**

0.1 (default) | scalar

Minimum turning radius in meters for the robot moving at its current speed, specified as a scalar.

## **TargetDirectionWeight — Cost function weight for target direction**

5 (default) | scalar

Cost function weight for moving toward the target direction, specified as a scalar. To follow a target direction, set this weight to be higher than the sum of the `CurrentDirectionWeight` and `PreviousDirectionWeight` properties. To ignore the target direction cost, set this weight to zero.

## **CurrentDirectionWeight — Cost function weight for current direction**

2 (default) | scalar

Cost function weight for moving the robot in the current heading direction, specified as a scalar. Higher values of this weight produce efficient paths. To ignore the current direction cost, set this weight to zero.

## **PreviousDirectionWeight — Cost function weight for previous direction**

2 (default) | scalar

Cost function weight for moving in the previously selected steering direction, specified as a scalar. Higher values of this weight produces smoother paths. To ignore the previous direction cost, set this weight to zero.

**HistogramThresholds — Thresholds for binary histogram computation**

[3 10] (default) | 2-element vector

Thresholds for binary histogram computation, specified as a 2-element vector. The algorithm uses these thresholds to compute the binary histogram from the polar obstacle density. Polar obstacle density values higher than the upper threshold are represented as occupied space (1) in the binary histogram. Values smaller than the lower threshold are represented as free space (0). Values that fall between the limits are set to the values in the previous binary histogram, with the default being free space (0).

**UseLidarScan — Use lidarScan object as scan input**

false (default) | true

Use lidarScan object as scan input, specified as either true or false.

## Usage

## Syntax

```
steeringDir = vfh(scan,targetDir)
steeringDir = vfh(ranges,angles,targetDir)
```

## Description

`steeringDir = vfh(scan,targetDir)` finds an obstacle-free steering direction using the VFH+ algorithm for the input lidarScan object, `scan`. A target direction is given based on the target location.

To enable this syntax, you must set the `UseLidarScan` property to `true`. For example:

```
mcl = robotics.MonteCarloLocalization('UseLidarScan','true');
...
[isUpdated,pose,covariance] = mcl(odomPose,scan);
```

`steeringDir = vfh(ranges,angles,targetDir)` defines the lidar scan with two vectors: `ranges` and `angles`.

## Input Arguments

### **scan — Lidar scan readings**

lidarScan object

Lidar scan readings, specified as a lidarScan object.

#### **Dependencies**

To use this argument, you must set the UseLidarScan property to true.

```
mcl.UseLidarScan = true;
```

### **ranges — Range values from scan data**

vector

Range values from scan data, specified as a vector in meters. These range values are distances from a sensor at given angles. The vector must be the same length as the corresponding angles vector.

### **angles — Angle values from scan data**

vector

Angle values from scan data, specified as a vector in radians. These angle values are the specific angles of the given ranges. The vector must be the same length as the corresponding ranges vector.

### **targetDir — Target direction for robot**

scalar

Target direction for the robot, specified as a scalar in radians. The forward direction of the robot is considered zero radians, with positive angles measured counterclockwise.

## Output Arguments

### **steeringDir — Steering direction for robot**

scalar

Steering direction for the robot, specified as a scalar in radians. This obstacle-free direction is calculated based on the VFH+ algorithm. The forward direction of the robot is considered zero radians, with positive angles measured counterclockwise.



## Object Functions

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

```
release(obj)
```

### Specific to robotics.VectorFieldHistogram

`show` Display VectorFieldHistogram information in figure window

### Common to All System Objects

`step` Run System object algorithm

`release` Release resources and allow changes to System object property values and input characteristics

`reset` Reset internal states of System object

## Examples

### Create a Vector Field Histogram Object and Visualize Data

This example shows how to calculate a steering direction based on input laser scan data.

Create a VectorFieldHistogram object.

```
vfh = robotics.VectorFieldHistogram;
```

Input laser scan data and target direction.

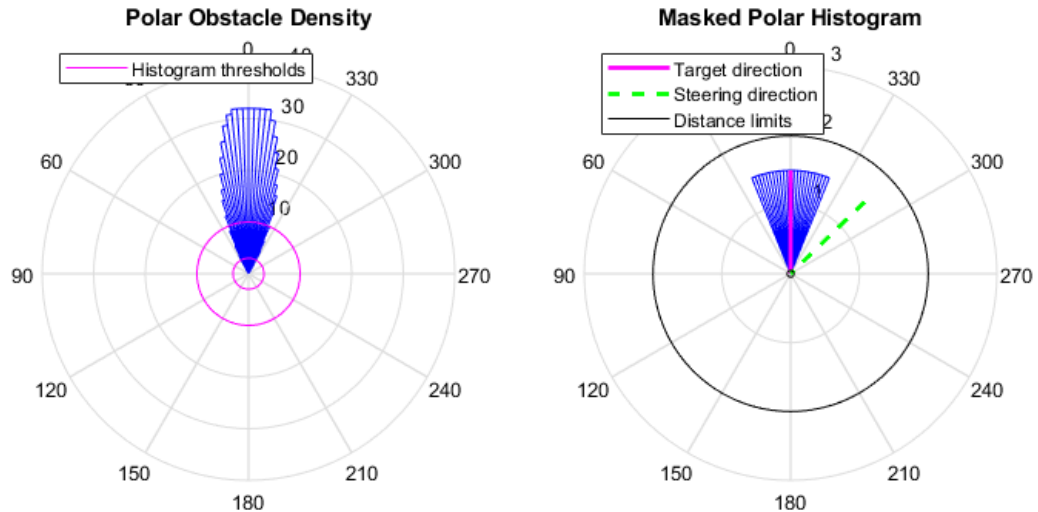
```
ranges = 10*ones(1,500);
ranges(1,225:275) = 1.0;
angles = linspace(-pi,pi,500);
targetDir = 0;
```

Compute an obstacle-free steering direction.

```
steeringDir = vfh(ranges,angles,targetDir)
steeringDir = -0.8014
```

Visualize the VectorFieldHistogram computation.

```
h = figure;  
set(h, 'Position', [50 50 800 400])  
show(vfh)
```



## References

- [1] Borenstein, J., and Y. Koren. "The Vector Field Histogram - Fast Obstacle Avoidance for Mobile Robots." *IEEE Journal of Robotics and Automation*. Vol. 7, Number 3, 1991, pp.278-88.
- [2] Ulrich, I., and J. Borenstein. "VFH : Reliable Obstacle Avoidance for Fast Mobile Robots." *Proceedings. 1998 IEEE International Conference on Robotics and Automation*. (1998): 1572-1577.

## Extended Capabilities

### C/C++ Code Generation

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

Usage notes and limitations:

Use in a MATLAB Function block is not supported.

Lidar scans require a limited size in code generation. The lidar scans, `scan`, are limited to 4000 points (range and angles) as a maximum.

For additional information about code generation for System objects, see “System Objects in MATLAB Code Generation” (MATLAB Coder)

### See Also

`lidarScan` | `show`

### Topics

“Obstacle Avoidance Using TurtleBot”

“Vector Field Histogram”

**Introduced in R2015b**

## 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.

---

**Note** This object requires you to install the UAV Library for Robotics System Toolbox. To install add-ons, use `roboticsAddons` and select the desired add-on.

---

### 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 u
"MAV_CMD_NAV_LOITER_TURNS"	18	"Loiter around this waypoint for
"MAV_CMD_NAV_LOITER_TIME"	19	"Loiter around this waypoint for
"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 fr
"MAV_CMD_NAV_TAKEOFF_LOCAL"	24	"Takeoff from local position (loc
"MAV_CMD_NAV_FOLLOW"	25	"Vehicle following, i.e. this way
"MAV_CMD_NAV_CONTINUE_AND_CHANGE_ALT"	30	"Continue on the current course a
"MAV_CMD_NAV_LOITER_TO_ALT"	31	"Begin loiter at the specified La
"MAV_CMD_DO_FOLLOW"	32	"Being following a target"
"MAV_CMD_DO_FOLLOW_REPOSITION"	33	"Reposition the MAV after a follo
"MAV_CMD_DO_ORBIT"	34	"Start orbiting on the circumfere
"MAV_CMD_NAV_ROI"	80	"Sets the region of interest (ROI
:		

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 a

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

## See Also

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

## Topics

"Use a MAVLink Parameter Protocol for Tuning UAV Parameters in MATLAB"

## 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.

---

**Note** This object requires you to install the UAV Library for Robotics System Toolbox. To install add-ons, use `roboticsAddons` and select the desired add-on.

---

### 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

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

## Examples

### Store MAVLink Client Information

**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.

```
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@0.0.0.0:64627"
```

```
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

"Use a MAVLink Parameter Protocol for Tuning UAV Parameters in MATLAB"

## 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.

---

**Note** This object requires you to install the UAV Library for Robotics System Toolbox. To install add-ons, use `roboticsAddons` and select the desired add-on.

---

## 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

**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.

```
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

“Use a MAVLink Parameter Protocol for Tuning UAV Parameters in MATLAB”

## 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.

---

**Note** This object requires you to install the UAV Library for Robotics System Toolbox. To install add-ons, use `roboticsAddons` and select the desired add-on.

---

### 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

**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.

```
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

“Use a MAVLink Parameter Protocol for Tuning UAV Parameters in MATLAB”

**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

`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**

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');
```

## **See Also**

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

## **Topics**

“Load and Playback MAVLink TLOG”

**Introduced in R2019a**



# Functions — Alphabetical List

---

# addRelativePose

**Package:** robotics

Add relative pose to pose graph

## Syntax

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

## Description

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

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

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

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

`[edge, edgeID] = addRelativePose( ___ )` returns the newly added edge and edge ID using any of the previous syntaxes.

## Input Arguments

### **poseGraph** — Pose graph

PoseGraph object | PoseGraph3D object

Pose graph, specified as a PoseGraph or PoseGraph3D object.

### **relPose** — Relative pose between nodes

[x y theta] vector | [x y z qw qx qy qz] vector

Relative pose between nodes, specified as one of the following:

For PoseGraph (2-D), the pose is a [x y theta] vector, which defines a xy-position and orientation angle, theta.

For PoseGraph3D, the pose is a [x y z qw qx qy qz] vector, which defines by an xyz-position and quaternion orientation, [qw qx qy qz]

---

**Note** Many other sources for 3-D pose graphs, including .g2o formats, specify the quaternion orientation in a different order, for example, [qx qy qz qw]. Check the source of your pose graph data before adding nodes to your PoseGraph3D object.

---

### **infoMatrix** — Information matrix

6-element vector | 21-element vector

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

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

For PoseGraph3D, the information matrix is a 21-element vector. The default is [1 0 0 0 0 0 1 0 0 0 0 1 0 0 0 1 0 0 1 0 1].

### **fromNodeID** — Node to attach from

positive integer

Node to attach from, specified as a positive integer. This integer corresponds to the node ID of a node in `poseGraph`. When specified without `toNodeID`, `addRelativePose` creates a new node and adds an edge between the new node and the `fromNodeID` node.

**toNodeID — Node to attach to**

positive integer

Node to attach to, specified as a positive integer. This integer corresponds to the node ID of a node in `poseGraph`. `addRelativePose` adds an edge between this node and the `fromNodeID` node.

## Output Arguments

**edge — Added edge**

two-element vector

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

**edgeID — ID of added edge**

positive integer

ID of added edge, returned as a positive integer.

## Extended Capabilities

### C/C++ Code Generation

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

Use this syntax when constructing `PoseGraph` or `PoseGraph3D` objects for code generation:

```
poseGraph =  
robotics.PoseGraph( 'MaxNumEdges', maxEdges, 'MaxNumNodes', maxNodes )
```

specifies an upper bound on the number of edges and nodes allowed in the pose graph when generating code. This limit is only required when generating code.

```
poseGraph =  
robotics.PoseGraph3D('MaxNumEdges', maxEdges, 'MaxNumNodes', maxNodes)
```

## See Also

### Functions

edgeConstraints | edges | findEdgeID | nodes | optimizePoseGraph |  
removeEdges

### Objects

robotics.LidarSLAM | robotics.PoseGraph | robotics.PoseGraph3D

## Topics

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

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

**Introduced in R2018a**

# 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**



# apply

Transform message entities into target frame

## Syntax

```
tfentity = apply(tfmsg,entity)
```

## Description

`tfentity = apply(tfmsg,entity)` applies the transformation represented by the 'TransformStamped' ROS message to the input message object `entity`.

This function determines the message type of `entity` and applies the appropriate transformation method to it. If the object cannot handle a particular message type, then MATLAB displays an error message.

If you only want to use the most current transformation, call `transform` instead. If you want to store a transformation message for later use, call `getTransform` and then call `apply`.

## Examples

### Apply A Transformation To A Point

Connect to a ROS network to get a TransformStamped ROS message. Specify the IP address to connect. Create a transformation tree and get the transformation between desired frames.

```
rosinit('192.168.203.129')
tftree = rostf;
pause(1);
tform = getTransform(tftree,'base_link','camera_link',...
                    rostime('now'),'Timeout',5);
```

Initializing global node /matlab\_global\_node\_77541 with NodeURI http://192.168.203.1:5

Create a ROS Point message and apply the transformation. You could also get point messages off the ROS network.

```
pt = rosmesssage('geometry_msgs/PointStamped');
pt.Header.FrameId = 'camera_link';
pt.Point.X = 3;
pt.Point.Y = 1.5;
pt.Point.Z = 0.2;
```

```
tfpt = apply(tform,pt);
```

Shut down ROS network.

```
roshutdown
```

```
Shutting down global node /matlab_global_node_77541 with NodeURI http://192.168.203.1:11311
```

## Input Arguments

### **tfmsg** — Transformation message

TransformStamped ROS message handle

Transformation message, specified as a TransformStamped ROS message handle. The `tfmsg` is a ROS message of type: `geometry_msgs/TransformStamped`.

### **entity** — ROS message

Message object handle

ROS message, specified as a Message object handle.

Supported messages are:

- `geometry_msgs/PointStamped`
- `geometry_msgs/PoseStamped`
- `geometry_msgs/PointCloud2Stamped`
- `geometry_msgs/QuaternionStamped`
- `geometry_msgs/Vector3Stamped`

## Output Arguments

### **tfentity** – Transformed ROS message

Message object handle

Transformed ROS message, returned as a Message object handle.

## See Also

`getTransform` | `transform`

**Introduced in R2015a**

# axang2quat

Convert axis-angle rotation to quaternion

## Syntax

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

## Description

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

## Examples

### Convert Axis-Angle Rotation to Quaternion

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

```
quat = 1×4
```

```
    0.7071    0.7071         0         0
```

## Input Arguments

### **axang** — Rotation given in axis-angle form

*n*-by-4 matrix

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

Example: [1 0 0 pi/2]

## Output Arguments

### **quat** — Unit quaternion

*n*-by-4 matrix

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

Example: [0.7071 0.7071 0 0]

## Extended Capabilities

### **C/C++ Code Generation**

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

### **See Also**

quat2axang

### **Topics**

“Coordinate Transformations in Robotics”

**Introduced in R2015a**

## axang2rotm

Convert axis-angle rotation to rotation matrix

### Syntax

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

### Description

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

### Examples

#### Convert Axis-Angle Rotation to Rotation Matrix

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

```
rotm = 3×3
```

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

### Input Arguments

**axang** — Rotation given in axis-angle form  
*n*-by-4 matrix

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

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

## Output Arguments

### **rotm** — Rotation matrix

3-by-3-by- $n$  matrix

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

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

## Extended Capabilities

### C/C++ Code Generation

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

### See Also

`rotm2axang`

### Topics

“Coordinate Transformations in Robotics”

**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 = 4x4
```

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

## Input Arguments

**axang** — Rotation given in axis-angle form  
*n*-by-4 matrix



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

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

## Output Arguments

### **tform** — Homogeneous transformation

4-by-4-by- $n$  matrix

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

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

## Extended Capabilities

### **C/C++ Code Generation**

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

### **See Also**

`tform2axang`

### **Topics**

“Coordinate Transformations in Robotics”

**Introduced in R2015a**

# bodyInfo

**Package:** robotics

Import information for body

## Syntax

```
info = bodyInfo(importInfo, bodyName)
```

## Description

`info = bodyInfo(importInfo, bodyName)` returns the import information for a body in a `RigidBodyTree` object that is created from calling `importrobot`. Specify the `robotics.RigidBodyTreeImportInfo` object from the import process.

## Input Arguments

### **importInfo — Robot import information**

`RigidBodyTreeImportInfo` object

Robot import information, specified as a `robotics.RigidBodyTreeImportInfo` object. This object is returned when you use `importrobot`.

### **bodyName — Name of body**

character vector | string scalar

Name of a body in the `RigidBodyTree` object that was created using `importrobot`, specified as a character vector or string scalar. Partial string matching is accepted and returns a cell array of structures that match the partial string.

Example: 'Body01'

Data Types: char | string

## Output Arguments

### **info** — Import information for specific component

structure | cell array of structures

Import information for specific component, returned as a structure or cell array of structures. This structure contains the information about the imported blocks from Simscape Multibody and the associated components in the `robotics.RigidBodyTree` object. The fields of each structure are:

- **BodyName** — Name of the body in the `robotics.RigidBodyTree` object.
- **JointName** — Name of the joint associated with **BodyName**.
- **BodyBlocks** — Blocks used from the Simscape Multibody model.
- **JointBlocks** — Joint blocks used from the Simscape Multibody model.

## See Also

`importrobot` | `robotics.RigidBodyTree` | `robotics.RigidBodyTreeImportInfo` | `showdetails`

**Introduced in R2018b**

# bodyInfoFromBlock

**Package:** robotics

Import information for block name

## Syntax

```
info = bodyInfo(importInfo,blockName)
```

## Description

`info = bodyInfo(importInfo,blockName)` returns the import information for a block in a Simscape Multibody model that is imported from calling `importrobot`. Specify the `robotics.RigidBodyTreeImportInfo` object from the import process.

## Input Arguments

### **importInfo** — Robot import information

`RigidBodyTreeImportInfo` object

Robot import information, specified as a `robotics.RigidBodyTreeImportInfo` object. This object is returned when you use `importrobot`.

### **blockName** — Name of block

character vector | string scalar

Name of a block in the Simscape Multibody model that was imported using `importrobot`, specified as a character vector or string scalar. Partial string matching is accepted and returns a cell array of structures that match the partial string.

Example: 'Prismatic Joint 2'

Data Types: char | string

## Output Arguments

### **info** — Import information for specific component

structure | cell array of structures

Import information for specific component, returned as a structure or cell array of structures. This structure contains the information about the imported blocks from Simscape Multibody and the associated components in the `robotics.RigidBodyTree` object. The fields of each structure are:

- **BodyName** — Name of the body in the `robotics.RigidBodyTree` object.
- **JointName** — Name of the joint associated with **BodyName**.
- **BodyBlocks** — Blocks used from the Simscape Multibody model.
- **JointBlocks** — Joint blocks used from the Simscape Multibody model.

## See Also

`importrobot` | `robotics.RigidBodyTree` | `robotics.RigidBodyTreeImportInfo` | `showdetails`

**Introduced in R2018b**

# bodyInfoFromJoint

**Package:** robotics

Import information for given joint name

## Syntax

```
info = bodyInfo(importInfo, jointName)
```

## Description

`info = bodyInfo(importInfo, jointName)` returns the import information for a joint in a `RigidBodyTree` object that is created from calling `importrobot`. Specify the `robotics.RigidBodyTreeImportInfo` object from the import process.

## Input Arguments

### **importInfo** — Robot import information

`RigidBodyTreeImportInfo` object

Robot import information, specified as a `robotics.RigidBodyTreeImportInfo` object. This object is returned when you use `importrobot`.

### **jointName** — Name of joint

character vector | string scalar

Name of a joint in the `RigidBodyTree` object that was created using `importrobot`, specified as a character vector or string scalar. Partial string matching is accepted and returns a cell array of structures that match the partial string.

Example: 'Joint01'

Data Types: char | string

## Output Arguments

### **info** — Import information for specific component

structure | cell array of structures

Import information for specific component, specified as a structure or cell array of structures. This structure contains the information about the imported blocks from Simscape Multibody and the associated components in the `robotics.RigidBodyTree` object. The fields of each structure are:

- `BodyName` — Name of the body in the `robotics.RigidBodyTree` object.
- `JointName` — Name of the joint associated with `BodyName`.
- `BodyBlocks` — Blocks used from the Simscape Multibody model.
- `JointBlocks` — Joint blocks used from the Simscape Multibody model.

## See Also

`importrobot` | `robotics.RigidBodyTree` | `robotics.RigidBodyTreeImportInfo`

**Introduced in R2018b**

## bsplinepolytraj

Generate polynomial trajectories using B-splines

### Syntax

```
[q,qd,qdd,pp] = bsplinepolytraj(controlPoints,tInterval,tSamples)
```

### Description

`[q,qd,qdd,pp] = bsplinepolytraj(controlPoints,tInterval,tSamples)` generates a piecewise cubic B-spline trajectory that falls in the control polygon defined by `controlPoints`. The trajectory is uniformly sampled between the start and end times given in `tInterval`. The function returns the positions, velocities, and accelerations at the input time samples, `tSamples`. The function also returns the piecewise polynomial `pp` form of the polynomial trajectory with respect to time.

### Examples

#### Compute B-Spline Trajectory for 2-D Planar Motion

Use the `bsplinepolytraj` function with a given set of 2-D  $xy$  control points. The B-spline uses these control points to create a trajectory inside the polygon. Time points for the waypoints are also given.

```
cpts = [1 4 4 3 -2 0; 0 1 2 4 3 1];  
tpts = [0 5];
```

Compute the B-spline trajectory. The function outputs the trajectory positions (`q`), velocity (`qd`), acceleration (`qdd`), time vector (`tvec`), and polynomial coefficients (`pp`) of the polynomial that achieves the waypoints using trapezoidal velocities.

```
tvec = 0:0.01:5;  
[q, qd, qdd, pp] = bsplinepolytraj(cpts,tpts,tvec);
```

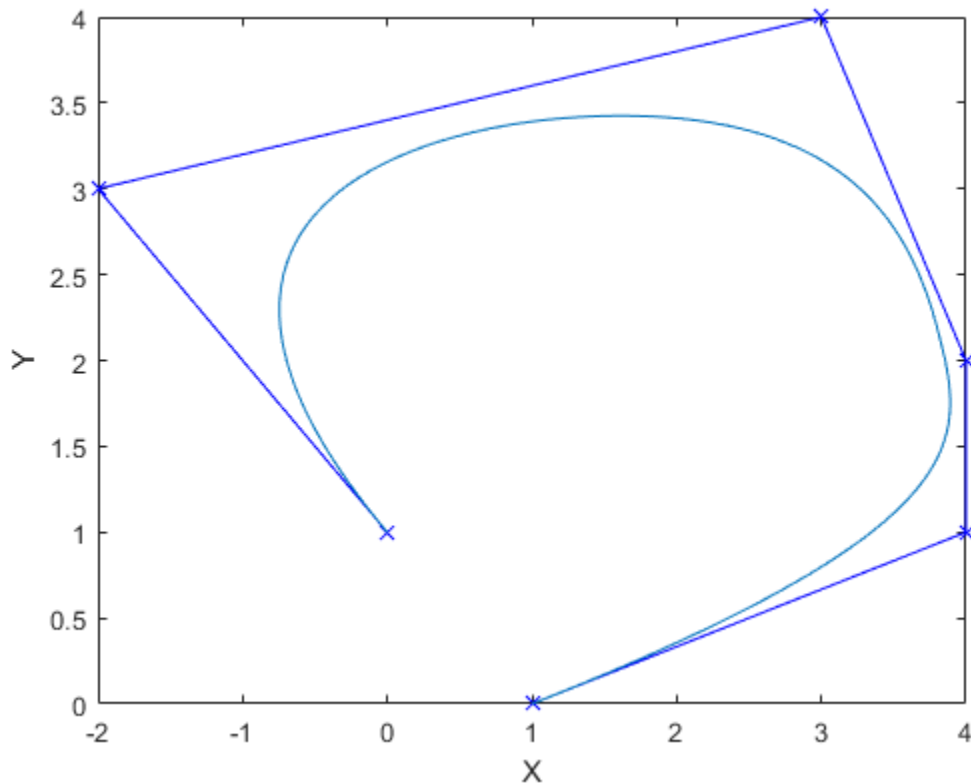
Plot the results. Show the control points and the resulting trajectory inside them.



```

figure
plot(cpts(1,:),cpts(2:,:), 'xb-')
hold all
plot(q(1,:), q(2,:))
xlabel('X')
ylabel('Y')
hold off

```



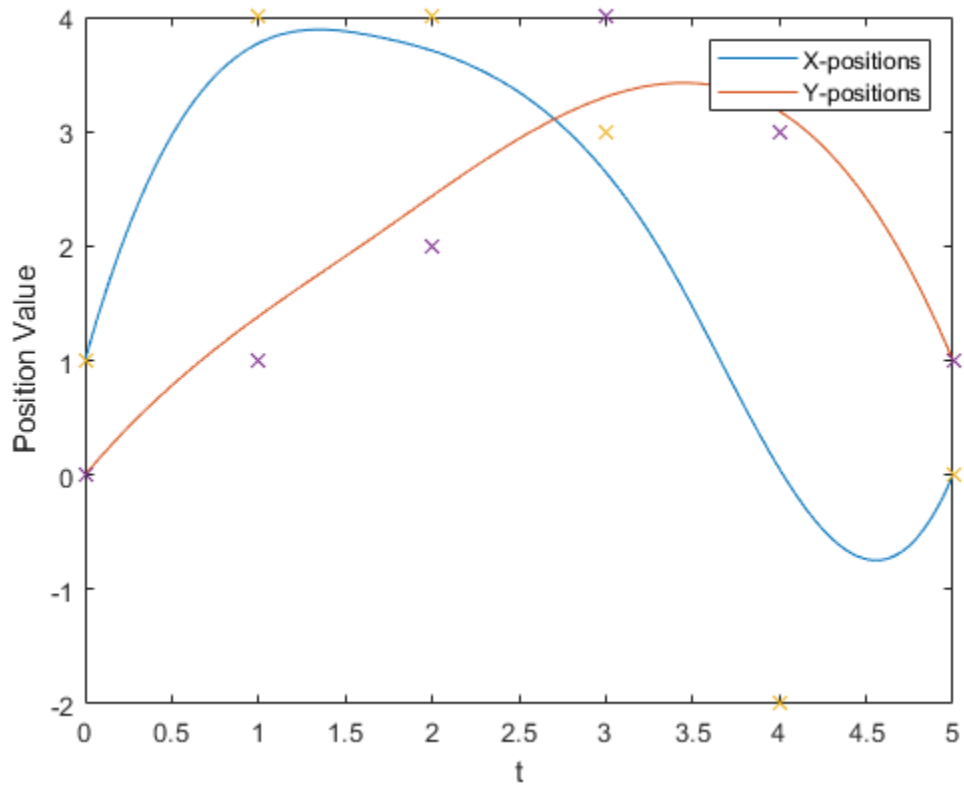
Plot the position of each element of the B-spline trajectory. These trajectories are cubic piecewise polynomials parameterized in time.

```

figure
plot(tvec,q)
hold all

```

```
plot([0:length(cpts)-1],cpts,'x')
xlabel('t')
ylabel('Position Value')
legend('X-positions','Y-positions')
hold off
```



## Input Arguments

**controlPoints** — Points for control polygon

*n*-by-*p* matrix

Points for control polygon of B-spline trajectory, specified as an  $n$ -by- $p$  matrix, where  $n$  is the dimension of the trajectory and  $p$  is the number of control points.

Example: [1 4 4 3 -2 0; 0 1 2 4 3 1]

Data Types: single | double

### **tInterval — Start and end times for trajectory**

two-element vector

Start and end times for the trajectory, specified as a two-element vector.

Example: [0 10]

Data Types: single | double

### **tSamples — Time samples for trajectory**

vector

Time samples for the trajectory, specified as a vector. The output position,  $q$ , velocity,  $q_d$ , and accelerations,  $q_{dd}$ , are sampled at these time intervals.

Example: 0:0.01:10

Data Types: single | double

## **Output Arguments**

### **q — Positions of trajectory**

vector

Positions of the trajectory at the given time samples in `tSamples`, returned as a vector.

Data Types: single | double

### **q<sub>d</sub> — Velocities of trajectory**

vector

Velocities of the trajectory at the given time samples in `tSamples`, returned as a vector.

Data Types: single | double

### **q<sub>dd</sub> — Accelerations of trajectory**

vector

Accelerations of the trajectory at the given time samples in `tSamples`, returned as a vector.

Data Types: `single` | `double`

### **pp** — Piecewise-polynomial structure

Piecewise-polynomial, returned as a structure that defines the polynomial for each section of the piecewise trajectory. You can build your own piecewise polynomials using `mkpp`, or evaluate the polynomial at specified times using `ppval`. The structure contains the fields:

- `form`: 'pp'.
- `breaks`:  $p$ -element vector of times when the piecewise trajectory changes forms.  $p$  is the number of waypoints.
- `coefs`:  $n(p-1)$ -by-order matrix for the coefficients for the polynomials.  $n(p-1)$  is the dimension of the trajectory times the number of pieces. Each set of  $n$  rows defines the coefficients for the polynomial that described each variable trajectory.
- `pieces`:  $p-1$ . The number of breaks minus 1.
- `order`: Degree of the polynomial + 1. For example, cubic polynomials have an order of 4.
- `dim`:  $n$ . The dimension of the control point positions.

## References

- [1] Farin, Gerald E. *Curves and Surfaces for Computer Aided Geometric Design: A Practical Guide*. San Diego, CA: Academic Press, 1993.

## Extended Capabilities

### C/C++ Code Generation

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

### See Also

`cubicpolytraj` | `quinticpolytraj` | `rottraj` | `transformtraj` | `trapveltraj`

**Introduced in R2019a**

# buildMap

Build occupancy grid from lidar scans

## Syntax

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

## Description

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

## Examples

### Build Occupancy Map from Lidar Scans and Poses

The `buildMap` function takes in lidar scan readings and associated poses to build an occupancy grid. as `lidarScan` objects and associated `[x y theta]` poses to build an `robotics.OccupancyGrid`.

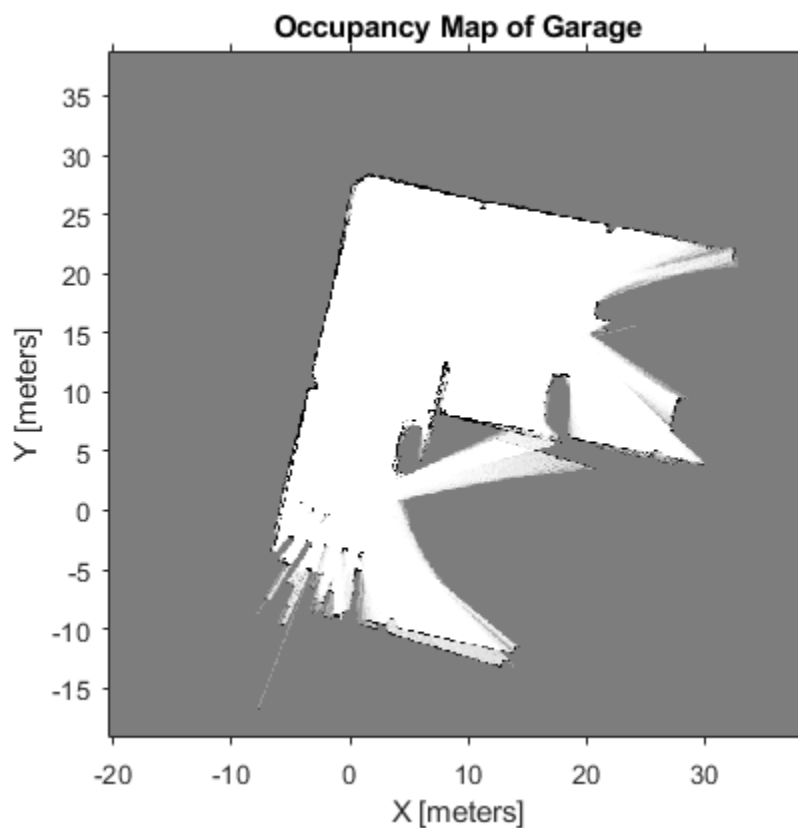
Load scan and pose estimates collected from sensors on a robot in a parking garage. The data collected is correlated using a `robotics.LidarSLAM` algorithm, which performs scan matching to associate scans and adjust poses over the full robot trajectory. Check to make sure scans and poses are the same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

```
load scansAndPoses.mat
length(scans) == length(poses)

ans = logical
     1
```

Build the map. Specify the scans and poses in the `buildMap` function and include the desired map resolution (10 cells per meter) and the max range of the lidar (19.2 meters). Each scan is added at the associated poses and probability values in the occupancy grid are updated.

```
occGrid = buildMap(scans,poses,10,19.2);  
figure  
show(occGrid)  
title('Occupancy Map of Garage')
```



### Perform SLAM Using Lidar Scans

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

### Load Data and Set Up SLAM Algorithm

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

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

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

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

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

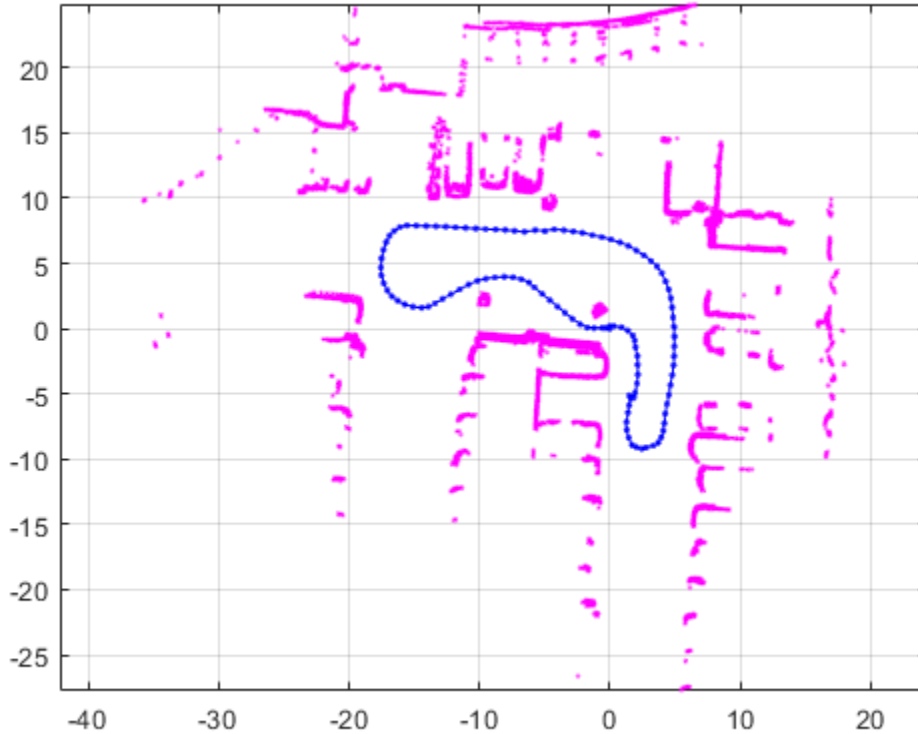
### Add Scans Iteratively

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

```
for i = 1:numel(scans)
    addScan(slamObj,scans{i});

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

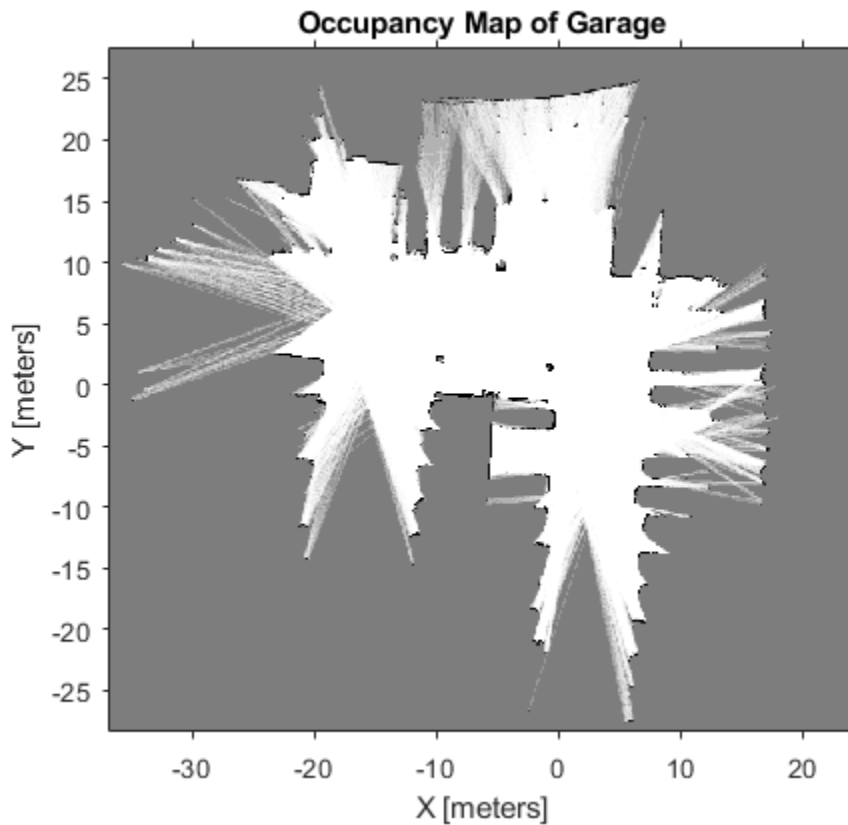




### View Occupancy Map

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

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



## Input Arguments

### **scans — Lidar scans**

cell array of `lidarScan` objects

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

### **poses — Poses of lidar scans**

$n$ -by-3 matrix

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

**mapResolution — Resolution of occupancy grid**

positive integer

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

**maxRange — Maximum range of lidar sensor**

positive scalar

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

**Name-Value Pair Arguments**

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

Example: `['MapWidth', 10]`

**MapWidth — Width of occupancy grid**

positive scalar

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

**MapHeight — Height of occupancy grid**

positive scalar

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

**Output Arguments****map — Occupancy grid**`robotics.OccupancyGrid` object

Occupancy grid, returned as a `robotics.OccupancyGrid` object.

## See Also

### Functions

lidarScan | matchScans | matchScansGrid | transformScan

### Classes

robotics.LidarSLAM | robotics.OccupancyGrid

### Topics

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

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

**Introduced in R2018a**

# call

Call the ROS service server and receive a response

## Syntax

```
response = call(serviceclient)
response = call(serviceclient,requestmsg)
response = call( ____,Name,Value)
```

## Description

`response = call(serviceclient)` sends a default service request message and waits for a service response. The default service request message is an empty message of type `serviceclient.ServiceType`.

`response = call(serviceclient,requestmsg)` specifies a service request message, `requestmsg`, to be sent to the service.

`response = call( ____,Name,Value)` provides additional options specified by one or more `Name,Value` pair arguments, using any of the arguments from the previous syntaxes.

## Examples

### Call Service Client with Default Message

Connect to a ROS network.

```
rosinit
```

```
Initializing ROS master on http://ah-sradford:11311/.
```

```
Initializing global node /matlab_global_node_03934 with NodeURI http://ah-sradford:6130
```

Set up a service server and client.

```
server = rossvcserver('/test', 'std_srvs/Empty', @exampleHelperROSEmptyCallback);
client = rossvcclient('/test');
```

Call service server with default message.

```
response = call(client)
```

A service client is calling

```
response =
```

```
ROS EmptyResponse message with properties:
```

```
  MessageType: 'std_srvs/EmptyResponse'
```

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

Shut down ROS network.

```
rosshutdown
```

```
Shutting down global node /matlab_global_node_03934 with NodeURI http://ah-sradford:6181/
Shutting down ROS master on http://ah-sradford:11311/.
```

### Call for Response Using Specific Request Message

Connect to a ROS network.

```
rosinit
```

```
Initializing ROS master on http://ah-sradford:11311/.
Initializing global node /matlab_global_node_18061 with NodeURI http://ah-sradford:6181/.
```

Set up a service server and client. This server calculates the sum of two integers and is based on a ROS service tutorial.

```
sumserver = rossvcserver('/sum', 'roscpp_tutorials/TwoInts', @exampleHelperROSSumCallback);
sumclient = rossvcclient('/sum');
```

Get the request message for the client and modify the parameters.

```
reqMsg = rosmesssage(sumclient);  
reqMsg.A = 2;  
reqMsg.B = 1;
```

Call service and get response. The response should be the sum of the two integers given in the request message. Wait 5 seconds for the service to timeout.

```
response = call(sumclient, reqMsg, 'Timeout', 5)
```

```
response =
```

```
ROS TwoIntsResponse message with properties:
```

```
  MessageType: 'roscpp_tutorials/TwoIntsResponse'  
    Sum: 3
```

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

Shut down ROS network.

```
roshutdn
```

```
Shutting down global node /matlab_global_node_18061 with NodeURI http://ah-sradford:6111  
Shutting down ROS master on http://ah-sradford:11311/.
```

## Input Arguments

### **serviceclient** — Service client

ServiceClient object handle

Service client, specified as a ServiceClient object handle.

### **requestmsg** — Request message

Message object handle

Request message, specified as a Message object handle. The default message type is `serviceclient.ServiceType`.

### 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: `"TimeOut", 5`

#### **TimeOut** — Timeout for service response in seconds

`inf` (default) | scalar

Timeout for service response in seconds, specified as a comma-separated pair consisting of `"TimeOut"` and a scalar. If the service client does not receive a service response and the timeout period elapses, `call` displays an error message and lets MATLAB continue running the current program. The default value of `inf` blocks MATLAB from running the current program until the service client receives a service response.

### Output Arguments

#### **response** — Response message

Message object handle

llResponse message sent by the service server, returned as a Message object handle.

### See Also

`rossvcclient`

**Introduced in R2015a**



# cancelAllGoals

Cancel all goals on action server

## Syntax

```
cancelAllGoals(client)
```

## Description

`cancelAllGoals(client)` sends a request from the specified client to the ROS action server to cancel all currently pending or active goals, including goals from other clients.

## Examples

### Send and Cancel ROS Action Goals

Send and cancel goals for ROS actions. First, setup a ROS action client. Then send a goal message with modified parameters. Finally, cancel your goal and all goals on the action server.

Connect to a ROS network with a specified IP address. Create a ROS action client connected using `roactionclient`. Specify the action name. Wait for the client to be connected to the server.

```
rosinit('192.168.154.131')  
[actClient,goalMsg] = roactionclient('/fibonacci');  
waitForServer(actClient);
```

```
Initializing global node /matlab_global_node_40739 with NodeURI http://192.168.154.1:5
```

Send a goal message with modified parameters. Wait for the goal to finish executing.

```
goalMsg.Order = 4;  
sendGoalAndWait(actClient,goalMsg)
```

```
Goal active
Feedback:
  Sequence : [0, 1, 1]
Feedback:
  Sequence : [0, 1, 1, 2]
Feedback:
  Sequence : [0, 1, 1, 2, 3]
Feedback:
  Sequence : [0, 1, 1, 2, 3, 5]
```

```
ans =
```

```
ROS FibonacciResult message with properties:
```

```
  MessageType: 'actionlib_tutorials/FibonacciResult'
  Sequence: [6x1 int32]
```

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

Send a new goal message without waiting.

```
goalMsg.Order = 5;
sendGoal(actClient,goalMsg)
```

Cancel the goal on the ROS action client, `actClient`.

```
cancelGoal(actClient)
```

Cancel all the goals on the action server that `actClient` is connected to.

```
cancelAllGoals(actClient)
```

Delete the action client.

```
delete(actClient)
```

Disconnect from the ROS network.

```
roshutdown
```

Shutting down global node /matlab\_global\_node\_40739 with NodeURI http://192.168.154.1:

## Input Arguments

### **client** — ROS action client

SimpleActionClient object handle

ROS action client, specified as a SimpleActionClient object handle. This simple action client enables you to track a single goal at a time.

## See Also

cancelGoal | rosaction | sendGoal | sendGoalAndWait

## Topics

“ROS Actions Overview”

“Move a Turtlebot Robot Using ROS Actions”

**Introduced in R2016b**

# cancelGoal

Cancel last goal sent by client

## Syntax

```
cancelGoal(client)
```

## Description

`cancelGoal(client)` sends a cancel request for the tracked goal, which is the last one sent to the action server. The specified client sends the request.

If the goal is in the 'active' state, the server preempts the execution of the goal. If the goal is 'pending', it is recalled. If this client has not sent a goal, or if the previous goal was achieved, this function returns immediately.

## Examples

### Send and Cancel ROS Action Goals

Send and cancel goals for ROS actions. First, setup a ROS action client. Then send a goal message with modified parameters. Finally, cancel your goal and all goals on the action server.

Connect to a ROS network with a specified IP address. Create a ROS action client connected using `roactionclient`. Specify the action name. Wait for the client to be connected to the server.

```
rosinit('192.168.154.131')  
[actClient,goalMsg] = roactionclient('/fibonacci');  
waitForServer(actClient);
```

Initializing global node /matlab\_global\_node\_40739 with NodeURI http://192.168.154.1:57

Send a goal message with modified parameters. Wait for the goal to finish executing.

```
goalMsg.Order = 4;  
sendGoalAndWait(actClient,goalMsg)
```

Goal active

Feedback:

Sequence : [0, 1, 1]

Feedback:

Sequence : [0, 1, 1, 2]

Feedback:

Sequence : [0, 1, 1, 2, 3]

Feedback:

Sequence : [0, 1, 1, 2, 3, 5]

ans =

ROS FibonacciResult message with properties:

MessageType: 'actionlib\_tutorials/FibonacciResult'

Sequence: [6x1 int32]

Use showdetails to show the contents of the message

Send a new goal message without waiting.

```
goalMsg.Order = 5;  
sendGoal(actClient,goalMsg)
```

Cancel the goal on the ROS action client, actClient.

```
cancelGoal(actClient)
```

Cancel all the goals on the action server that actClient is connected to.

```
cancelAllGoals(actClient)
```

Delete the action client.

```
delete(actClient)
```

Disconnect from the ROS network.

```
roshutdown
```

Shutting down global node /matlab\_global\_node\_40739 with NodeURI http://192.168.154.1:

## Input Arguments

### **client** — ROS action client

SimpleActionClient object handle

ROS action client, specified as a SimpleActionClient object handle. This simple action client enables you to track a single goal at a time.

## See Also

cancelAllGoals | rosaction | sendGoal | sendGoalAndWait

## Topics

“ROS Actions Overview”

“Move a Turtlebot Robot Using ROS Actions”

**Introduced in R2016b**

## canTransform

Verify if transformation is available

### Syntax

```
isAvailable = canTransform(tftree, targetframe, sourceframe)
isAvailable = canTransform(tftree, targetframe, sourceframe,
sourcetime)
```

```
isAvailable = canTransform(bagSel, targetframe, sourceframe)
isAvailable = canTransform(bagSel, targetframe, sourceframe,
sourcetime)
```

### Description

`isAvailable = canTransform(tftree, targetframe, sourceframe)` verifies if a transformation between the source frame and target frame is available at the current time in `tftree`. Create the `tftree` object using `rostdf`, which requires a connection to a ROS network.

`isAvailable = canTransform(tftree, targetframe, sourceframe, sourcetime)` verifies if a transformation is available for the source time. If `sourcetime` is outside the buffer window, the function returns `false`.

`isAvailable = canTransform(bagSel, targetframe, sourceframe)` verifies if a transformation is available in a rosbag in `bagSel`. To get the `bagSel` input, load a rosbag using `rosbag`.

`isAvailable = canTransform(bagSel, targetframe, sourceframe, sourcetime)` verifies if a transformation is available in a rosbag for the source time. If `sourcetime` is outside the buffer window, the function returns `false`.

### Examples

### Send a Transformation to ROS Network

This example shows how to create a transformation and send it over the ROS network.

Create a ROS transformation tree. Use `rosinit` to connect a ROS network. Replace `ipaddress` with your ROS network address.

```
ipaddress = '192.168.203.129';  
rosinit(ipaddress)  
tftree = rostf;  
pause(2)
```

```
Initializing global node /matlab_global_node_69912 with NodeURI http://192.168.203.1:5
```

Verify the transformation you want to send over the network does not already exist. `canTransform` returns false if the transformation is not immediately available.

```
canTransform(tftree, 'new_frame', 'base_link')
```

```
ans =
```

```
logical
```

```
0
```

Create a `TransformStamped` message. Populate the message fields with the transformation information.

```
tform = rosmessage('geometry_msgs/TransformStamped');  
tform.ChildFrameId = 'new_frame';  
tform.Header.FrameId = 'base_link';  
tform.Transform.Translation.X = 0.5;  
tform.Transform.Rotation.Z = 0.75;
```

Send the transformation over the ROS network.

```
sendTransform(tftree, tform)
```

Verify the transformation is now on the ROS network

```
canTransform(tftree, 'new_frame', 'base_link')
```

```
ans =
```



```
logical
```

```
1
```

Shut down the ROS network.

```
roshutdown
```

```
Shutting down global node /matlab_global_node_69912 with NodeURI http://192.168.203.1:5
```

### Get ROS Transformations and Apply to ROS Messages

This example shows how to set up a ROS transformation tree and transform frames based on this information. It uses time-buffered transformations to access transformations at different times.

Create a ROS transformation tree. Use `roslaunch` to connect to a ROS network. Replace `ipaddress` with your ROS network address.

```
ipaddress = '192.168.203.129';
roslaunch(ipaddress)
tftree = rostf;
pause(1)
```

```
Initializing global node /matlab_global_node_60416 with NodeURI http://192.168.203.1:5
```

Look at the available frames on the transformation tree.

```
tftree.AvailableFrames
```

```
ans =
```

```
36×1 cell array
```

```

{'base_footprint'      }
{'base_link'          }
{'camera_depth_frame' }
{'camera_depth_optical_frame'}
{'camera_link'        }
{'camera_rgb_frame'   }
```

```
{'camera_rgb_optical_frame' }
{'caster_back_link' }
{'caster_front_link' }
{'cliff_sensor_front_link' }
{'cliff_sensor_left_link' }
{'cliff_sensor_right_link' }
{'gyro_link' }
{'mount_asus_xtion_pro_link' }
{'odom' }
{'plate_bottom_link' }
{'plate_middle_link' }
{'plate_top_link' }
{'pole_bottom_0_link' }
{'pole_bottom_1_link' }
{'pole_bottom_2_link' }
{'pole_bottom_3_link' }
{'pole_bottom_4_link' }
{'pole_bottom_5_link' }
{'pole_kinect_0_link' }
{'pole_kinect_1_link' }
{'pole_middle_0_link' }
{'pole_middle_1_link' }
{'pole_middle_2_link' }
{'pole_middle_3_link' }
{'pole_top_0_link' }
{'pole_top_1_link' }
{'pole_top_2_link' }
{'pole_top_3_link' }
{'wheel_left_link' }
{'wheel_right_link' }
```

Check if the desired transformation is available now. For this example, check for the transformation from 'camera\_link' to 'base\_link'.

```
canTransform(tftree, 'base_link', 'camera_link')
```

```
ans =
```

```
logical
```

```
1
```

Get the transformation for 3 seconds from now. `getTransform` will wait until the transformation becomes available with the specified timeout.

```
desiredTime = rostime('now') + 3;
tform = getTransform(tftree,'base_link','camera_link',...
                    desiredTime,'Timeout',5);
```

Create a ROS message to transform. Messages could also be retrieved off the ROS network.

```
pt = rosmesssage('geometry_msgs/PointStamped');
pt.Header.FrameId = 'camera_link';
pt.Point.X = 3;
pt.Point.Y = 1.5;
pt.Point.Z = 0.2;
```

Transform the ROS message to the 'base\_link' frame using the desired time saved from before.

```
tfpt = transform(tftree,'base_link',pt,desiredTime);
```

*Optional:* You can also use `apply` with the stored `tform` to apply this transformation to the `pt` message.

```
tfpt2 = apply(tform,pt);
```

Shut down the ROS network.

```
roshutdown
```

```
Shutting down global node /matlab_global_node_60416 with NodeURI http://192.168.203.1:5
```

## Get Transformations from rosbag File

Get transformations from rosbag (.bag) files by loading the rosbag and checking the available frames. From these frames, use `getTransform` to query the transformation between two coordinate frames.

Load the rosbag.

```
bag = rosbag('ros_turtlesim.bag');
```

Get a list of available frames.

```
frames = bag.AvailableFrames;
```

Get the latest transformation between two coordinate frames.

```
tf = getTransform(bag, 'world', frames{1});
```

Check for a transformation available at a specific time and retrieve the transformation. Use `canTransform` to check if the transformation is available. Specify the time using `rostime`.

```
tftime = rostime(bag.StartTime + 1);  
if (canTransform(bag, 'world', frames{1}, tftime))  
    tf2 = getTransform(bag, 'world', frames{1}, tftime);  
end
```

## Input Arguments

### **tftree** — ROS transformation tree

`TransformationTree` object handle

ROS transformation tree, specified as a `TransformationTree` object handle. Create a transformation tree by calling the `rostf` function.

### **bagSel** — Selection of rosbag messages

`BagSelection` object handle

Selection of rosbag messages, specified as a `BagSelection` object handle. To create a selection of rosbag messages, use `rosbag`.

### **targetframe** — Target coordinate frame

string scalar | character vector

Target coordinate frame, specified as a string scalar or character vector. You can view the available frames for transformation calling `tftree.AvailableFrames` or `bagSel.AvailableFrames`.

### **sourceframe** — Initial coordinate frame

string scalar | character vector

Initial coordinate frame, specified as a string scalar or character vector. You can view the available frames for transformation by calling `tftree.AvailableFrames` or `bagSel.AvailableFrames`.

**sourcetime — ROS or system time**

scalar | Time object handle

ROS or system time, specified as a scalar or Time object handle. The scalar input is converted to a Time object using `rostime`.

## Output Arguments

**isAvailable — Indicator if transform exists**

boolean

Indicator if transform exists, returned as a boolean. The function returns false if:

- `sourcetime` is outside the buffer window for a `tftree` object.
- `sourcetime` is outside the time of the `bagSel` object.
- `sourcetime` is in the future.
- The transformation is not published yet.

## See Also

`getTransform` | `rosviz` | `rostopic` | `rostopic` | `transform` | `waitForTransform`

**Introduced in R2016b**

## cart2hom

Convert Cartesian coordinates to homogeneous coordinates

### Syntax

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

### Description

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

### Examples

#### Convert 3-D Cartesian Points to Homogeneous Coordinates

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

```
h = 2×4
```

```
    0.8147    0.1270    0.6324    1.0000  
    0.9058    0.9134    0.0975    1.0000
```

### Input Arguments

#### **cart** — Cartesian coordinates

*n*-by-*(k-1)* matrix

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

Example: [0.8147 0.1270 0.6324; 0.9058 0.9134 0.0975]

## Output Arguments

### **hom** — Homogeneous points

*n*-by-*k* matrix

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

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

## Extended Capabilities

### **C/C++ Code Generation**

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

### **See Also**

hom2cart

### **Topics**

“Coordinate Transformations in Robotics”

**Introduced in R2015a**

# 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™.

**Introduced in R2018a**

# connect

**Package:** robotics

Connect poses for given connection type

## Syntax

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

## Description

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

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

## Examples

### Connect Poses Using Dubins Connection Path

Create a `DubinsConnection` object.

```
dubConnObj = robotics.DubinsConnection;
```

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

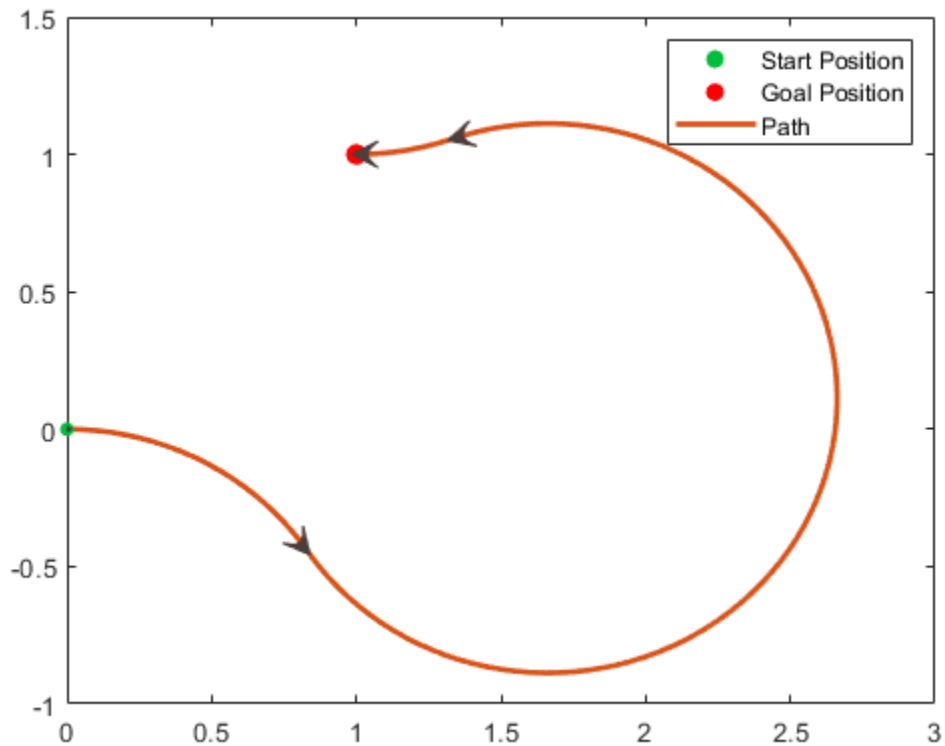
```
startPose = [0 0 0];
goalPose = [1 1 pi];
```

Calculate a valid path segment to connect the poses.

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

Show the generated path.

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



### Connect Poses Using ReedsShepp Connection Path

Create a ReedsSheppConnection object.

```
reedsConnObj = robotics.ReedsSheppConnection;
```

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

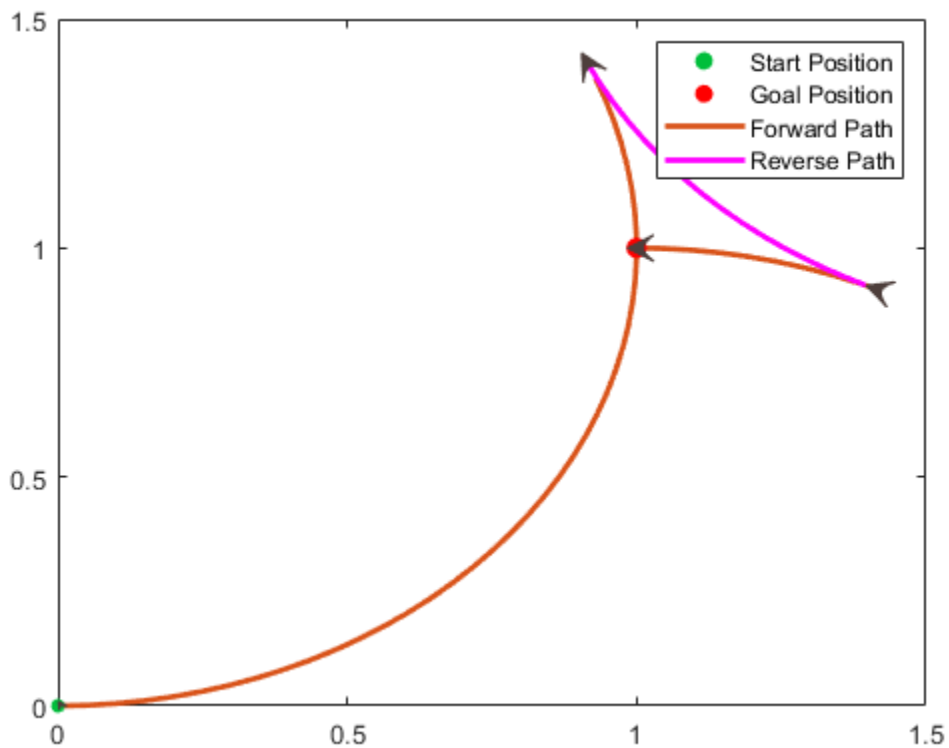
```
startPose = [0 0 0];  
goalPose = [1 1 pi];
```

Calculate a valid path segment to connect the poses.

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

Show the generated path.

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



# Input Arguments

### **connectionObj** — Path connection type

DubinsPathSegment object | ReedsSheppPathSegment object

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

### **start** — Initial pose of robot

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

This property is read-only.

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

$x$  and  $y$  are in meters.  $\theta$  is in radians.

The `connect` function supports:

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

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

### **goal** — Goal pose of robot

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

This property is read-only.

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

$x$  and  $y$  are in meters.  $\theta$  is in radians.

The `connect` function supports:

- Singular start pose with singular goal pose.

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

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

## Output Arguments

### **pathSegments** — Path segments

cell array of objects

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

### **pathCosts** — Cost of path segment

positive numeric scalar | positive numeric vector | positive numeric matrix

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

Example: `[7.6484,7.5122]`

## Extended Capabilities

### **C/C++ Code Generation**

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

## See Also

### Functions

`interpolate` | `show`

### Objects

`robotics.DubinsConnection` | `robotics.DubinsPathSegment` |  
`robotics.ReedsSheppConnection` | `robotics.ReedsSheppPathSegment`

### Introduced in R2018b



## 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  $i^{\text{th}}$  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 = 1x4
    0.5377    1.8339   -2.2588    0.8622
```

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

```
quatArray = [quaternion([1:4;5:8]), quaternion([9:12;13:16])]
```

```
quatArray = 2x2 quaternion array
    1 + 2i + 3j + 4k    9 + 10i + 11j + 12k
    5 + 6i + 7j + 8k    13 + 14i + 15j + 16k
```

```
quatArrayParts = compact(quatArray)
```

```
quatArrayParts = 4x4
    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™.

**Introduced in R2018a**

# 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™.

**Introduced in R2018a**

---

# control

**Package:** robotics

Control commands for UAV

## Syntax

```
controlStruct = control(uavGuidanceModel)
```

## Description

---

**Note** This function requires you to install the UAV Library for Robotics System Toolbox. To install add-ons, use `roboticsAddons` and select the desired add-on.

---

`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.

**Note:** To use UAV algorithms, you must install the UAV Library for Robotics System Toolbox®. To install, use `roboticsAddons`.

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 fixed-wing UAV states as a 13-by-*n* matrix.

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

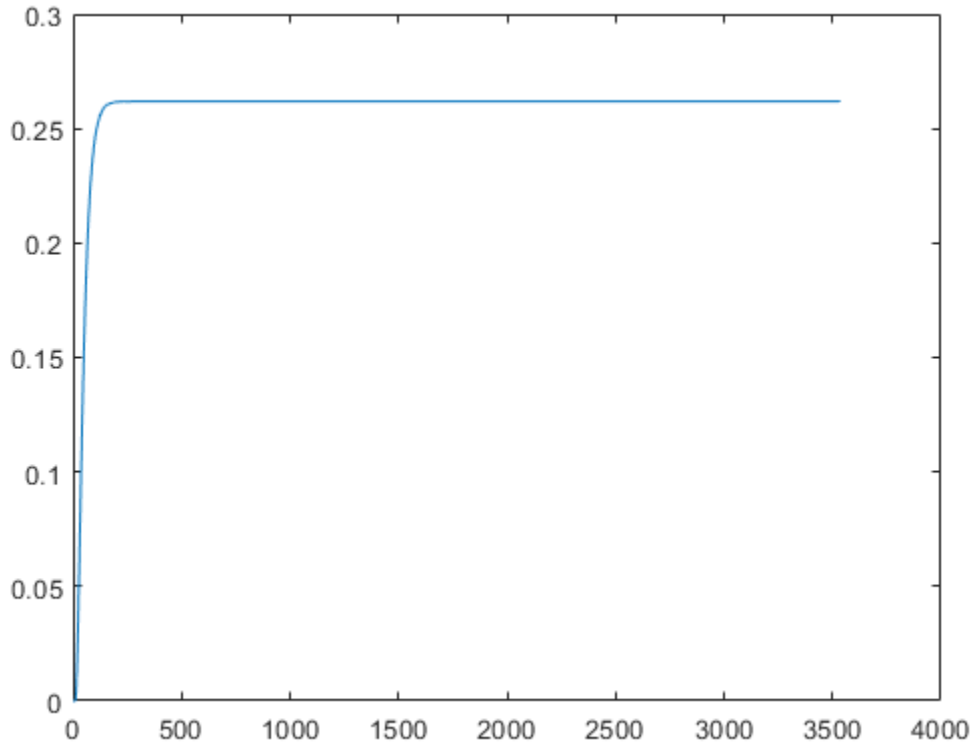
```
ans = 1×2
```

```
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.

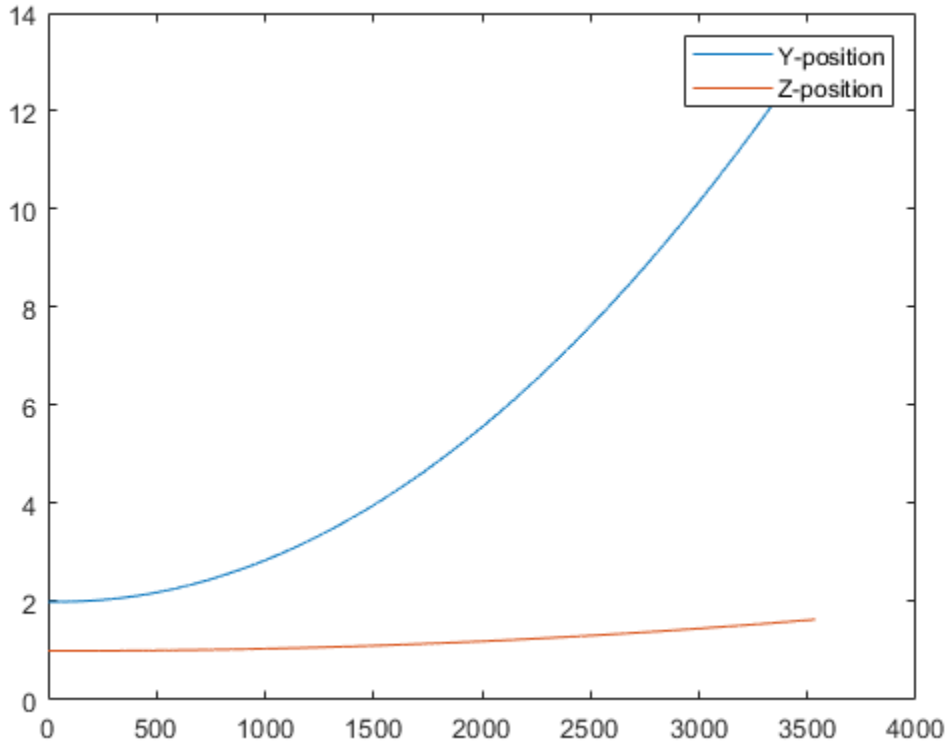
```
plot(simOut.y(9,:))
```





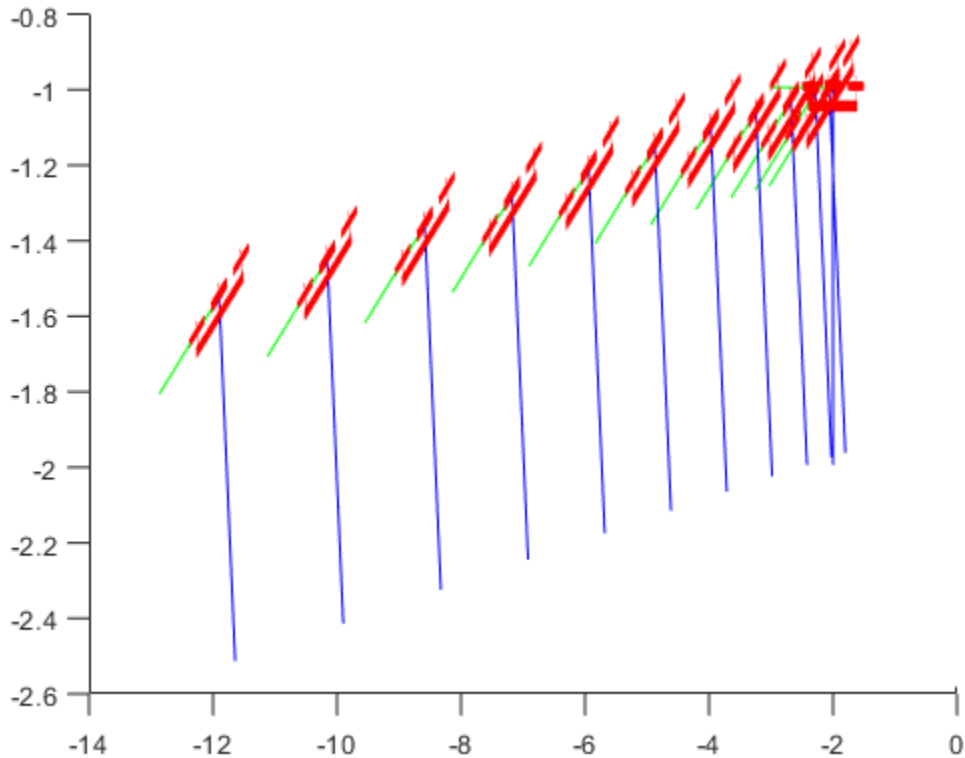
Plot the change in the Y and Z positions. With the specified thrust and roll angle, the multicopter should fly over and lose some altitude. A positive value for Z is expected as positive Z is down.

```
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.

**Note:** To use UAV algorithms, you must install the UAV Library for Robotics System Toolbox®. To install, use `roboticsAddons`.

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; % 10 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.

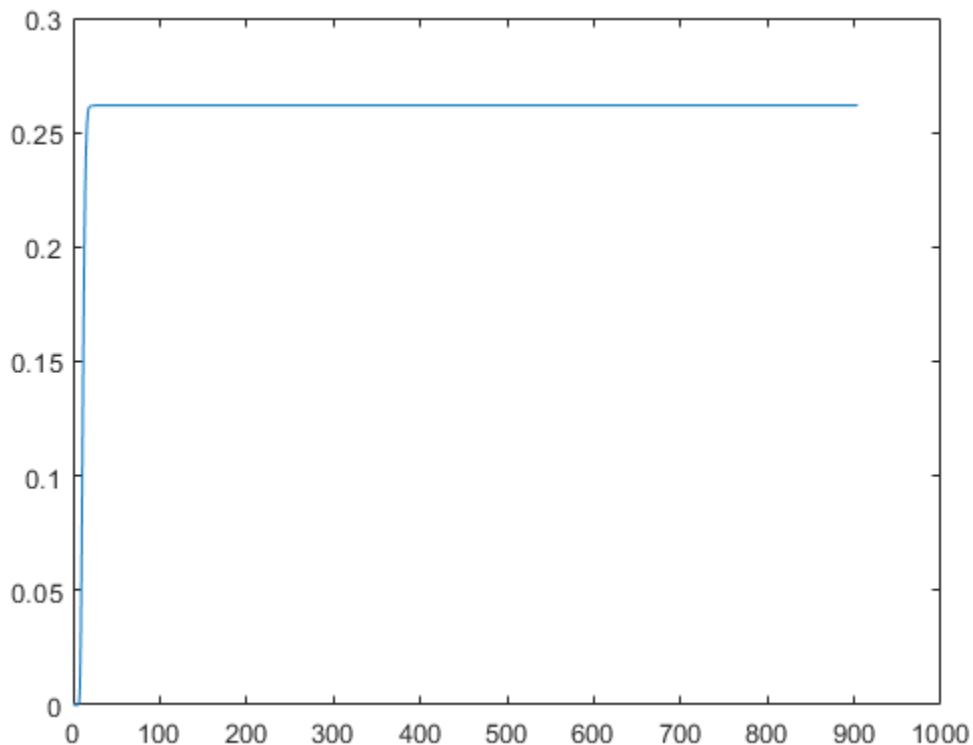
```
simOut = ode45(@(~,x)derivative(model,x,u,e), [0 50], s);  
size(simOut.y)
```

```
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,:))
```



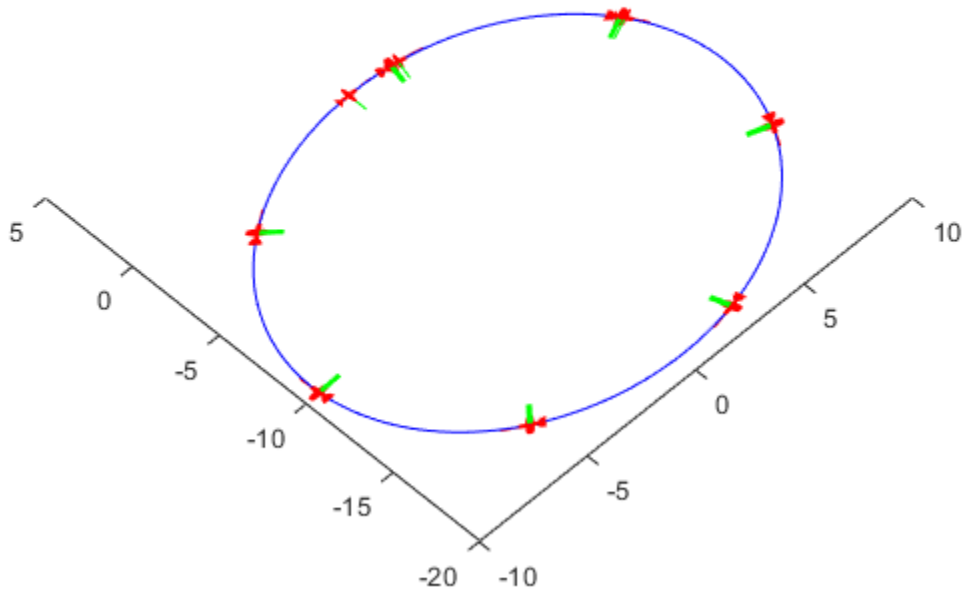
You can also plot the fixed-wing trajectory using `plotTransforms`. Create the translation and rotation vectors from the simulated state. Downsample (every 30th element) and transpose the `simOut` elements, and convert the Euler angles to quaternions. Specify the mesh as the `fixedwing.stl` file and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

### **Objects**

fixedwing | multicopter

### **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**



## 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
```

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

```
quatTransposed = quat'
```

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

## 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™.

**Introduced in R2018a**

## cubicpolytraj

Generate third-order polynomial trajectories

### Syntax

```
[q,qd,qdd,pp] = cubicpolytraj(wayPoints,timePoints,tSamples)  
[q,qd,qdd,pp] = cubicpolytraj( ____,Name,Value)
```

### Description

`[q,qd,qdd,pp] = cubicpolytraj(wayPoints,timePoints,tSamples)` generates a third-order polynomial that achieves a given set of input waypoints with corresponding time points. The function outputs positions, velocities, and accelerations at the given time samples, `tSamples`. The function also returns the piecewise polynomial `pp` form of the polynomial trajectory with respect to time.

`[q,qd,qdd,pp] = cubicpolytraj( ____,Name,Value)` specifies additional parameters as `Name,Value` pair arguments using any combination of the previous syntaxes.

### Examples

#### Compute Cubic Trajectory for 2-D Planar Motion

Use the `cubicpolytraj` function with a given set of 2-D `xy` waypoints. Time points for the waypoints are also given.

```
wpts = [1 4 4 3 -2 0; 0 1 2 4 3 1];  
tpts = 0:5;
```

Specify a time vector for sampling the trajectory. Sample at a smaller interval than the specified time points.

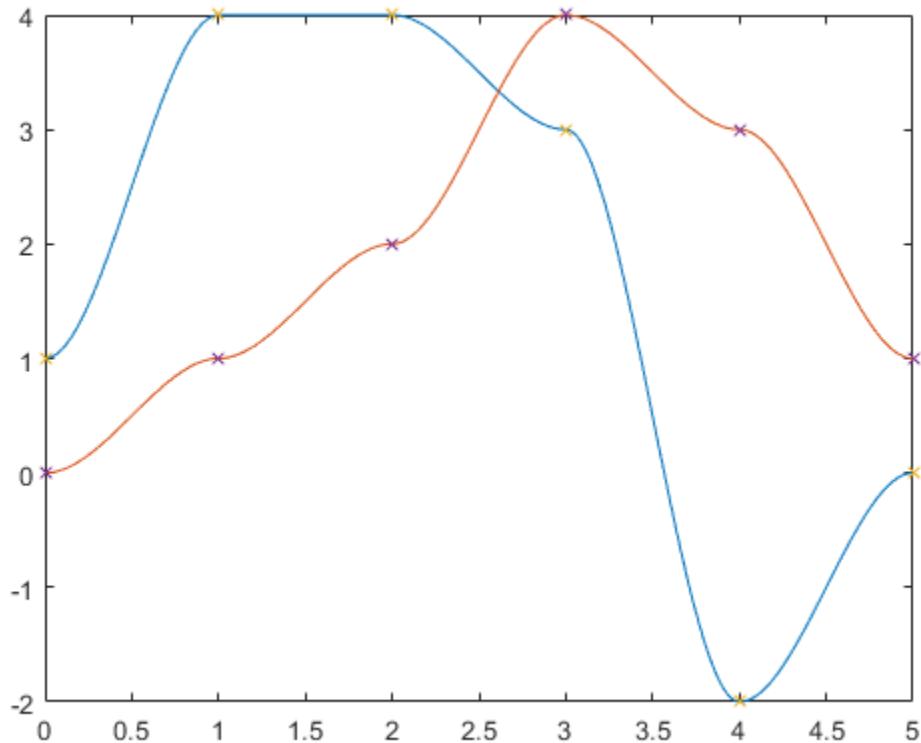
```
tvec = 0:0.01:5;
```

Compute the cubic trajectory. The function outputs the trajectory positions ( $q$ ), velocity ( $q\dot{d}$ ), acceleration ( $q\ddot{d}$ ), and polynomial coefficients ( $pp$ ) of the cubic polynomial.

```
[q, qd, qdd, pp] = cubicpolytraj(wpts, tpts, tvec);
```

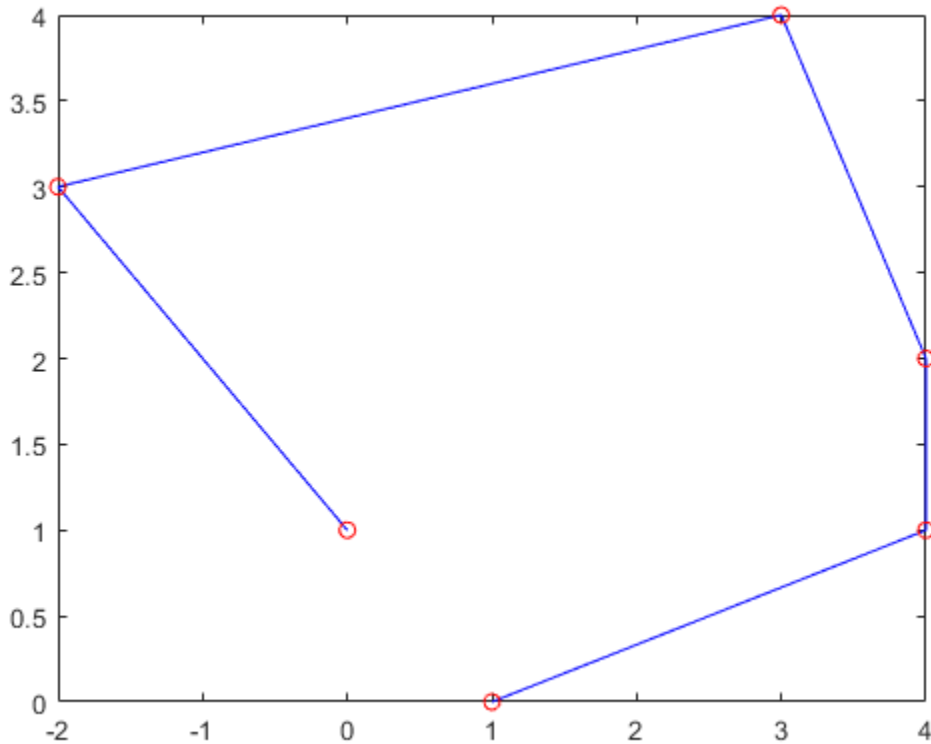
Plot the cubic trajectories for the  $x$  and  $y$ -positions. Compare the trajectory with each waypoint.

```
plot(tvec, q)  
hold all  
plot(tpts, wpts, 'x')  
hold off
```



You can also verify the actual positions in the 2-D plane. Plot the separate rows of the  $q$  vector and the waypoints as  $x$  and  $y$  positions.

```
figure  
plot(q(1,:),q(2,:), '-b', wpts(1,:),wpts(2,:), 'or')
```



## Input Arguments

**wayPoints** — Waypoints for trajectory

*n*-by-*p* matrix

Points for waypoints of trajectory, specified as an  $n$ -by- $p$  matrix, where  $n$  is the dimension of the trajectory and  $p$  is the number of waypoints.

Example: [1 4 4 3 -2 0; 0 1 2 4 3 1]

Data Types: single | double

### **timePoints — Time points for waypoints of trajectory**

$p$ -element vector

Time points for waypoints of trajectory, specified as a  $p$ -element vector.

Example: [0 2 4 5 8 10]

Data Types: single | double

### **tSamples — Time samples for trajectory**

$m$ -element vector

Time samples for the trajectory, specified as an  $m$ -element vector. The output position,  $q$ , velocity,  $qd$ , and accelerations,  $qdd$ , are sampled at these time intervals.

Example: 0:0.01:10

Data Types: single | double

## **Name-Value Pair Arguments**

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

Example: 'VelocityBoundaryCondition',[1 0 -1 -1 0 0; 1 1 1 -1 -1 -1]

### **VelocityBoundaryCondition — Velocity boundary conditions for each waypoint**

`zeroes(n,p)` (default) |  $n$ -by- $p$  matrix

Velocity boundary conditions for each waypoint, specified as the comma-separated pair consisting of 'VelocityBoundaryCondition' and an  $n$ -by- $p$  matrix. Each row corresponds to the velocity at all  $p$  waypoints for the respective variable in the trajectory.

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

Data Types: single | double

## Output Arguments

### **q — Positions of trajectory**

*m*-element vector

Positions of the trajectory at the given time samples in `tSamples`, returned as an *m*-element vector, where *m* is the length of `tSamples`.

Data Types: `single` | `double`

### **qd — Velocities of trajectory**

vector

Velocities of the trajectory at the given time samples in `tSamples`, returned as a vector.

Data Types: `single` | `double`

### **qdd — Accelerations of trajectory**

vector

Accelerations of the trajectory at the given time samples in `tSamples`, returned as a vector.

Data Types: `single` | `double`

### **pp — Piecewise-polynomial**

structure

Piecewise-polynomial, returned as a structure that defines the polynomial for each section of the piecewise trajectory. You can build your own piecewise polynomials using `mkpp`, or evaluate the polynomial at specified times using `ppval`. The structure contains the fields:

- `form`: 'pp'.
- `breaks`: *p*-element vector of times when the piecewise trajectory changes forms. *p* is the number of waypoints.
- `coefs`: *n*(*p*-1)-by-order matrix for the coefficients for the polynomials. *n*(*p*-1) is the dimension of the trajectory times the number of pieces. Each set of *n* rows defines the coefficients for the polynomial that described each variable trajectory.
- `pieces`: *p*-1. The number of breaks minus 1.
- `order`: Degree of the polynomial + 1. For example, cubic polynomials have an order of 4.



- `dim: n`. The dimension of the control point positions.

## Extended Capabilities

### C/C++ Code Generation

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

### See Also

[bsplinepolytraj](#) | [quinticpolytraj](#) | [rottraj](#) | [transformtraj](#) | [trapveltraj](#)

**Introduced in R2019a**

# definition

Retrieve definition of ROS message type

## Syntax

```
def = definition(msg)
```

## Description

`def = definition(msg)` returns the ROS definition of the message type associated with the message object, `msg`. The details of the message definition include the structure, property data types, and comments from the authors of that specific message.

## Examples

### Access ROS Message Definition for Message

Create a Point Message.

```
point = rosmessage('geometry_msgs/Point');
```

Access the definition.

```
def = definition(point)
```

```
def =  
    '% This contains the position of a point in free space  
    double X  
    double Y  
    double Z  
'
```

## Input Arguments

### **msg** — ROS message

Message object handle

ROS message, specified as a Message object handle. This message can be created using the `rosmessage` function.

## Output Arguments

### **def** — Details of message definition

character vector

Details of the information inside the ROS message definition, returned as a character vector.

## See Also

`rosmessage` | `rosmmsg`

**Introduced in R2015a**

# del

Delete a ROS parameter

## Syntax

```
del(ptree, paramname)  
del(ptree, namespace)
```

## Description

`del(ptree, paramname)` deletes a parameter with name `paramname` from the parameter tree, `ptree`. The parameter is also deleted from the ROS parameter server. If the specified `paramname` does not exist, the function displays an error.

`del(ptree, namespace)` deletes from the parameter tree all parameter values under the specified namespace.

## Examples

### Delete Parameter on ROS Master

Connect to the ROS network. Create a parameter tree and a 'MyParam' parameter. Check that the parameter exists.

```
rosinit
```

```
Initializing ROS master on http://bat5742win64:61477/.  
Initializing global node /matlab_global_node_30320 with NodeURI http://bat5742win64:61477/
```

```
ptree = rosparam;  
set(ptree, 'MyParam', 'test')  
has(ptree, 'MyParam')
```

```
ans = logical
      1
```

Delete the parameter. Verify it was deleted. Shut down the ROS network.

```
del(ptree, 'MyParam')
has(ptree, 'MyParam')
```

```
ans = logical
      0
```

```
rosshutdown
```

```
Shutting down global node /matlab_global_node_30320 with NodeURI http://bat5742win64:6
Shutting down ROS master on http://bat5742win64:61477/.
```

## Input Arguments

### **ptree** — Parameter tree

ParameterTree object handle

Parameter tree, specified as a ParameterTree object handle. Create this object using the `rosparam` function.

### **paramname** — ROS parameter name

string scalar | character vector

ROS parameter name, specified as a string scalar or character vector. This string must match the parameter name exactly.

### **namespace** — ROS parameter namespace

string scalar | character vector

ROS parameter namespace, specified as a string scalar or character vector. All parameter names starting with this string are listed when calling `rosparam("list", namespace)`.

## See Also

has | rosparam | set

**Introduced in R2015a**

## deleteFile

Delete file from device

### Syntax

```
deleteFile(device,filename)
```

### Description

`deleteFile(device,filename)` deletes the specified file from the ROS device.

### Examples

#### Put, Get, and Delete Files on ROS Device

Put a file from your host computer onto a ROS device, get it back, and then delete it.

Connect to a ROS device. Specify the device address, user name, and password of your ROS device.

```
d = rosdevice('192.168.154.131','user','password');
```

Put a new text file that is in the MATLAB® current folder onto the ROS device. The destination folder must exist.

```
putFile(d,'test_file.txt','/home/user/test_folder')
```

Get a text file from the ROS device. You can get any file, not just ones added from MATLAB®. By default, the file is added to the MATLAB current folder.

```
getFile(d,'/home/user/test_folder/test_file.txt')
```

Delete the text file on the ROS device.

```
deleteFile(d, '/home/user/test_folder/test_file.txt')
```

### Put, Get, and Delete Files on ROS Device Using Wildcards

Put a file from your host computer onto a ROS device, get it back, and then delete it. Use wildcards to search for all matching files.

**Note:** You must have a valid ROS device to connect to at the IP address specified in the example.

Connect to a ROS device. Specify the device address, user name, and password of your ROS device.

```
d = rosdevice('192.168.203.129', 'user', 'password');
```

Put all text files at the specified path onto the ROS device. The destination folder must exist.

```
putFile(d, 'C:/MATLAB/*.txt', '/home/user/test_folder')
```

Get all text files from the ROS device. You can get any files, not just ones added from MATLAB®. By default, the files are added to the MATLAB current folder.

```
getFile(d, '/home/user/test_folder/*.txt')
```

Delete all text files on the ROS device at the specified folder.

```
deleteFile(d, '/home/user/test_folder/*.txt')
```

## Input Arguments

### **device** — ROS device

rosdevice object

ROS device, specified as a rosdevice object.

### **filename** — File to delete

character vector

File to delete, specified as a character vector. When you specify the file name, you can use path information and wildcards.



Example: '/home/user/image.jpg'

Example: '/home/user/\*.jpg'

Data Types: cell

## **See Also**

dir | getFile | openShell | putFile | rosdevice | system

**Introduced in R2016b**

# derivative

**Package:** robotics

Time derivative of UAV states

## Syntax

```
stateDerivative = derivative(uavGuidanceModel, state, control,  
environment)
```

## Description

---

**Note** This function requires you to install the UAV Library for Robotics System Toolbox. To install add-ons, use `roboticsAddons` and select the desired add-on.

---

`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.

**Note:** To use UAV algorithms, you must install the UAV Library for Robotics System Toolbox®. To install, use `roboticsAddons`.

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 fixed-wing UAV states as a 13-by-*n* matrix.

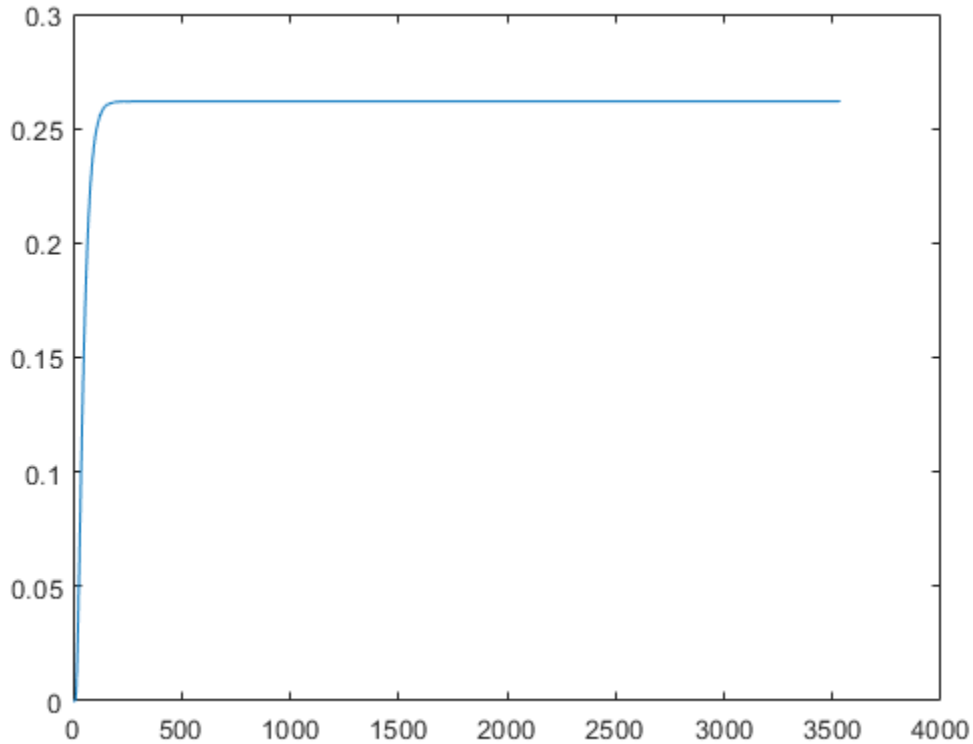
```
simOut = ode45(@(~,x)derivative(model,x,u,e), [0 3], s);
size(simOut.y)
```

```
ans = 1×2
```

```
13      3536
```

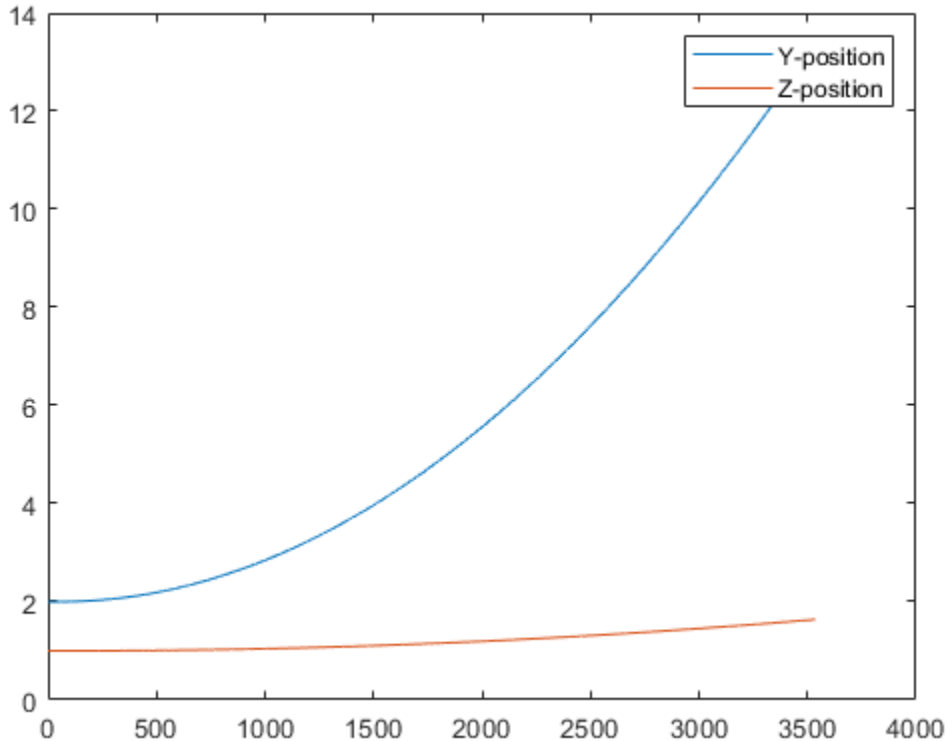
Plot the change in roll angle based on the simulation output. The roll angle (the X Euler angle) is the 9th row of the `simOut.y` output.

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



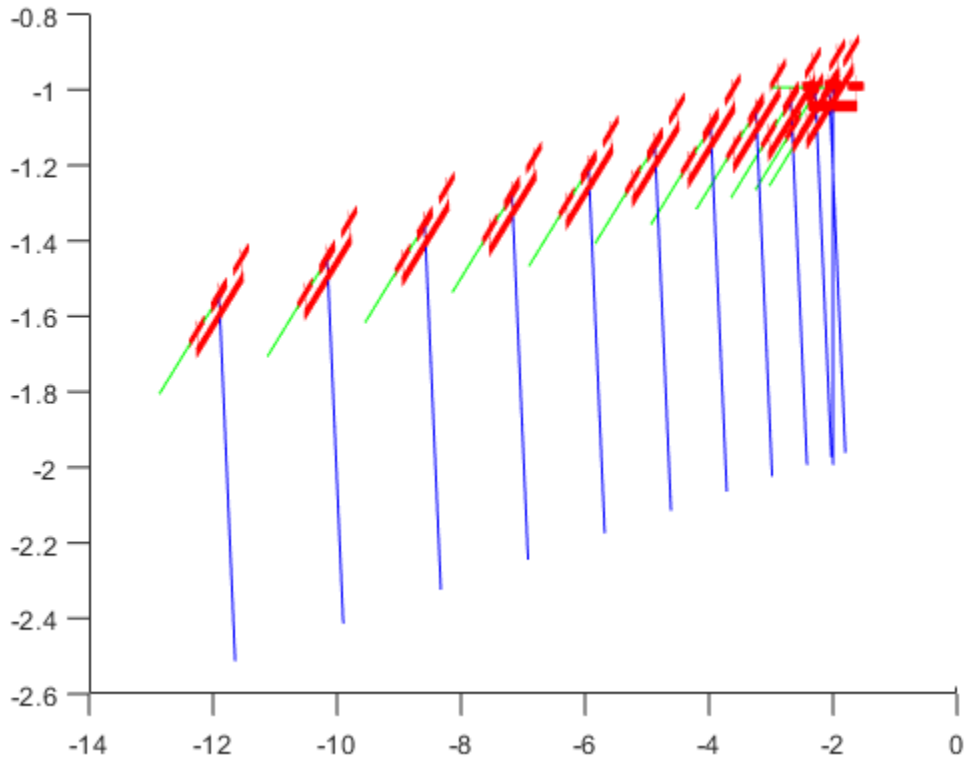
Plot the change in the Y and Z positions. With the specified thrust and roll angle, the multicopter should fly over and lose some altitude. A positive value for Z is expected as positive Z is down.

```
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.

**Note:** To use UAV algorithms, you must install the UAV Library for Robotics System Toolbox®. To install, use `roboticsAddons`.

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; % 10 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.

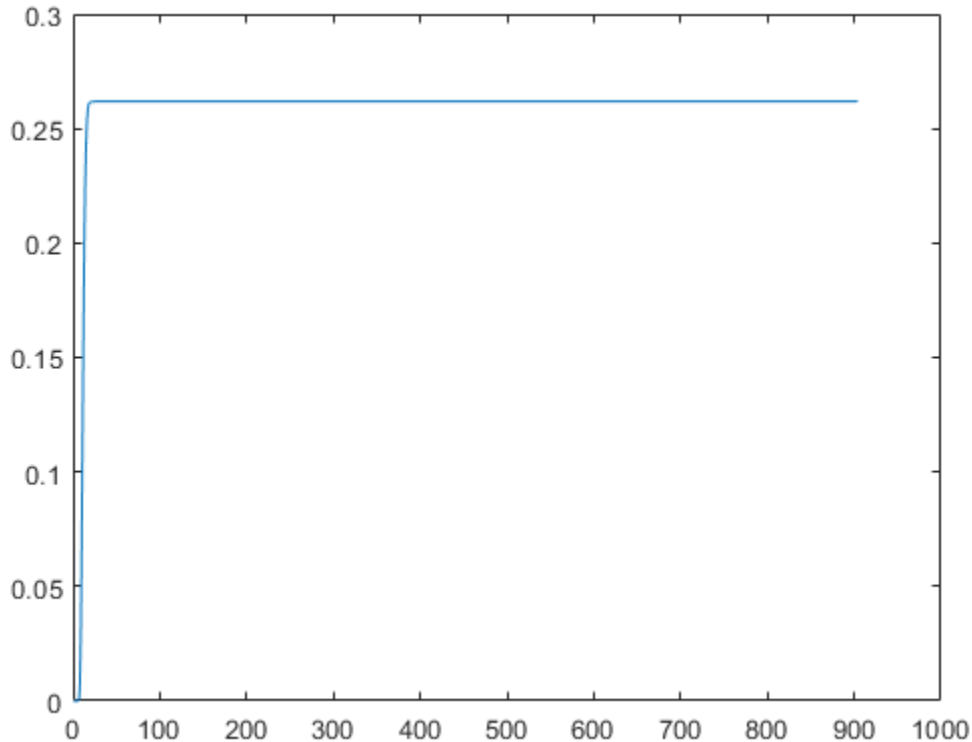
```
simOut = ode45(@(~,x)derivative(model,x,u,e), [0 50], s);  
size(simOut.y)
```

```
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,:))
```

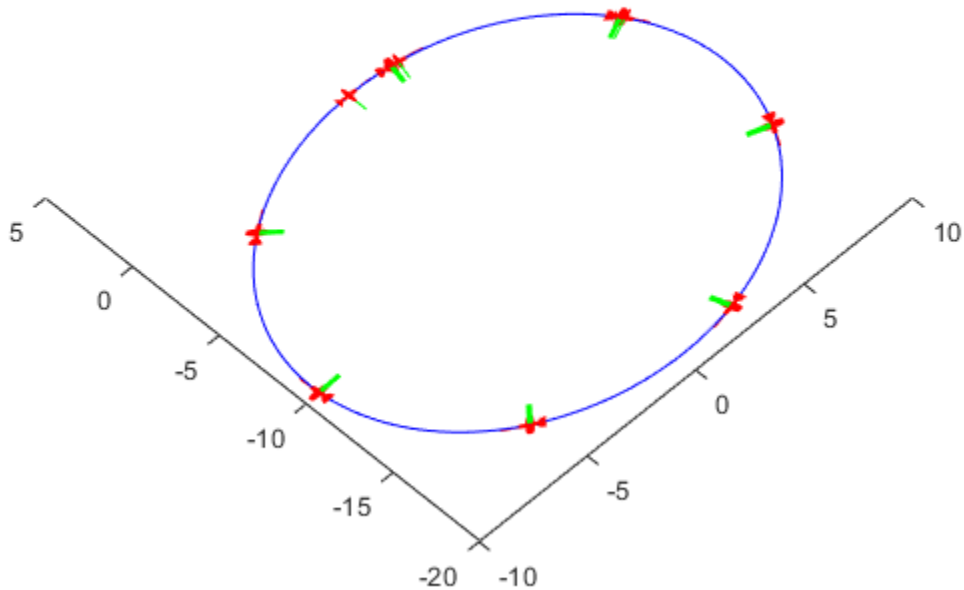


You can also plot the fixed-wing trajectory using `plotTransforms`. Create the translation and rotation vectors from the simulated state. Downsample (every 30th element) and transpose the `simOut` elements, and convert the Euler angles to quaternions. Specify the mesh as the `fixedwing.stl` file and 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 per second.
- **FlightPathAngle** - Angle between ground velocity and north-east plane in meters per second.
- **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** - [r p q] 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` | `roboticsAddons` | `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

# dir

List folder contents on device

## Syntax

```
dir(device, folder)
clist = dir(device, folder)
```

## Description

`dir(device, folder)` lists the files in a folder on the ROS device. Wildcards are supported.

`clist = dir(device, folder)` stores the list of files as a structure

## Examples

### View Folder Contents on ROS Device

Connect to a ROS device and list the contents of a folder.

Connect to a ROS device. Specify the device address, username, and password of your ROS device.

```
d = rosdevice('192.168.154.131', 'user', 'password');
```

Get the folder list of a Catkin workspace on your ROS device. View the folder as a table.

```
flist = dir(d, '/home/user/catkin_ws_test/');
ftable = struct2table(flist)
```

```
ftable =
```

name	folder	isdir	bytes
------	--------	-------	-------

'.'	'/home/user/catkin_ws_test'	true	0
'..'	'/home/user/catkin_ws_test'	true	0
'.catkin_workspace'	'/home/user/catkin_ws_test'	false	98
'build'	'/home/user/catkin_ws_test'	true	0
'devel'	'/home/user/catkin_ws_test'	true	0
'robotcontroller2_node.log'	'/home/user/catkin_ws_test'	false	75
'robotcontroller_node.log'	'/home/user/catkin_ws_test'	false	75
'src'	'/home/user/catkin_ws_test'	true	0

## Input Arguments

### **device** — ROS device

rosdevice object

ROS device, specified as a rosdevice object.

### **folder** — Folder name

character vector

Name of the folder to list the contents of, specified as a character vector.

## Output Arguments

### **cList** — Contents list

structure

Contents list, returned as a structure. The structure contains these fields:

- **name** — File name (char)
- **folder** — Absolute path (char)
- **bytes** — Size of the file in bytes (double)
- **isdir** — Indicator of whether name is a folder (logical)

## See Also

`deleteFile` | `getFile` | `openShell` | `putFile` | `rosdevice` | `system`

**Introduced in R2016b**

# dist

Angular distance in radians

## Syntax

```
distance = dist(quatA,quatB)
```

## Description

`distance = dist(quatA,quatB)` returns the angular distance in radians between the quaternion rotation operators for `quatA` and `quatB`.

## Examples

### Calculate Quaternion Distance

Calculate the quaternion distance between a single quaternion and each element of a vector of quaternions. Define the quaternions using Euler angles.

```
q = quaternion([0,0,0], 'eulerd', 'zyx', 'frame')
```

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

```
qArray = quaternion([0,45,0;0,90,0;0,180,0;0,-90,0;0,-45,0], 'eulerd', 'zyx', 'frame')
```

```
qArray = 5x1 quaternion array  
    0.92388 +          0i +    0.38268j +          0k  
    0.70711 +          0i +    0.70711j +          0k  
    6.1232e-17 +        0i +          1j +          0k  
    0.70711 +          0i -    0.70711j +          0k  
    0.92388 +          0i -    0.38268j +          0k
```



```

quaternionDistance = rad2deg(dist(q,qArray))
quaternionDistance = 5×1

    45.0000
    90.0000
   180.0000
    90.0000
    45.0000

```

If both arguments to `dist` are vectors, the quaternion distance is calculated between corresponding elements. Calculate the quaternion distance between two quaternion vectors.

```

angles1 = [30,0,15; ...
           30,5,15; ...
           30,10,15; ...
           30,15,15];
angles2 = [30,6,15; ...
           31,11,15; ...
           30,16,14; ...
           30.5,21,15.5];

qVector1 = quaternion(angles1,'eulerd','zyx','frame');
qVector2 = quaternion(angles2,'eulerd','zyx','frame');

rad2deg(dist(qVector1,qVector2))

ans = 4×1

    6.0000
    6.0827
    6.0827
    6.0287

```

Note that a quaternion represents the same rotation as its negative. Calculate a quaternion and its negative.

```

qPositive = quaternion([30,45,-60],'eulerd','zyx','frame')

qPositive = quaternion
    0.72332 - 0.53198i + 0.20056j + 0.3919k

```

```
qNegative = -qPositive
```

```
qNegative = quaternion  
    -0.72332 + 0.53198i - 0.20056j - 0.3919k
```

Find the distance between the quaternion and its negative.

```
dist(qPositive,qNegative)
```

```
ans = 0
```

The components of a quaternion may look different from the components of its negative, but both expressions represent the same rotation.

## Input Arguments

### **quatA, quatB — Quaternions to calculate distance between**

scalar | vector | matrix | multidimensional array

Quaternions to calculate distance between, specified as comma-separated quaternions or arrays of quaternions. `quatA` and `quatB` must have compatible sizes:

- `size(quatA) == size(quatB)`, or
- `numel(quatA) == 1`, or
- `numel(quatB) == 1`, or
- if `[Adim1,...,AdimN] = size(quatA)` and `[Bdim1,...,BdimN] = size(quatB)`, then for `i = 1:N`, either `Adimi==Bdimi` or `Adim==1` or `Bdim==1`.

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

Data Types: quaternion

## Output Arguments

### **distance — Angular distance (radians)**

scalar | vector | matrix | multidimensional array

Angular distance in radians, returned as an array. The dimensions are the maximum of the union of `size(quatA)` and `size(quatB)`.

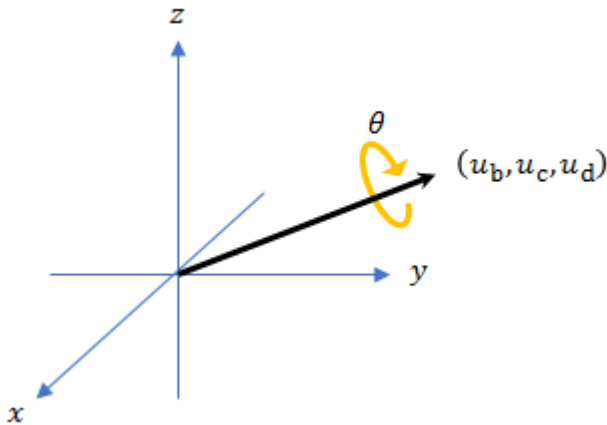
Data Types: `single` | `double`

## Algorithms

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

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

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



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

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

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

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

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

## **Extended Capabilities**

### **C/C++ Code Generation**

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

**Introduced in R2018a**

# edgeConstraints

**Package:** robotics

Edge constraints in pose graph

## Syntax

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

## Description

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

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

`[relPoses,infoMatrices] = edgeConstraints(poseGraph,edgeIDs)` returns edge constraints for the specified edge IDs.

## Input Arguments

### **poseGraph** — Pose graph

PoseGraph object | PoseGraph3D object

Pose graph, specified as a PoseGraph or PoseGraph3D object.

### **edgeIDs** — Edge IDs

vector of positive integers

Edge IDs, specified as a vector of positive integers.

## Output Arguments

### **relPoses** — Relative poses measured between nodes

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

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

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

For PoseGraph3D, each row is an [x y z qw qx qy qz] vector, which defines the relative xyz-position and quaternion orientation, [qw qx qy qz], of a pose in the graph.

---

**Note** Many other sources for 3-D pose graphs, including .g2o formats, specify the quaternion orientation in a different order, for example, [qx qy qz qw]. Check the source of your pose graph data before adding nodes to your PoseGraph3D object.

---

### **infoMatrices** — Information matrices

*n*-by-6 matrix | *n*-by-21 matrix

Information matrices, specified in compact form as a *n*-by-6 or *n*-by-21 matrix, where *n* is the number of poses in the pose graph.

Each row is a vector that contains the elements of the upper triangle of the square information matrix. The information matrix is the inverse of the covariance of the pose and represents the uncertainty of the measurement. If the pose vector is [x y theta], the covariance is a 3-by-3 matrix of pairwise covariance calculations. Typically, the uncertainty is determined by the sensor model.

For PoseGraph (2-D), each information matrix is a six-element vector. The default is [1 0 0 1 0 1].

For PoseGraph3D, each information matrix is a 21-element vector. The default is [1 0 0 0 0 0 1 0 0 0 0 1 0 0 0 1 0 0 1 0 1].

## Extended Capabilities

### C/C++ Code Generation

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

Use this syntax when constructing PoseGraph or PoseGraph3D objects for code generation:

```
poseGraph =  
robotics.PoseGraph( 'MaxNumEdges',maxEdges, 'MaxNumNodes',maxNodes)
```

specifies an upper bound on the number of edges and nodes allowed in the pose graph when generating code. This limit is only required when generating code.

### See Also

#### Functions

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

#### Objects

`robotics.LidarSLAM` | `robotics.PoseGraph` | `robotics.PoseGraph3D`

### Topics

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

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

**Introduced in R2018a**

# edges

**Package:** robotics

Edges in pose graph

## Syntax

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

## Description

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

`edges = edges (poseGraph, edgeIDs)` returns edges corresponding to the specified edge IDs.

## Input Arguments

**poseGraph — Pose graph**

PoseGraph object | PoseGraph3D object

Pose graph, specified as a PoseGraph or PoseGraph3D object.

**edgeIDs — Edge IDs**

vector of positive integers

Edge IDs, specified as a vector of positive integers.

## Output Arguments

**edges — Edges in pose graph**

*n*-by-2 matrix



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

## Extended Capabilities

### C/C++ Code Generation

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

Use this syntax when constructing PoseGraph or PoseGraph3D objects for code generation:

```
poseGraph =  
robotics.PoseGraph( 'MaxNumEdges',maxEdges, 'MaxNumNodes',maxNodes )
```

specifies an upper bound on the number of edges and nodes allowed in the pose graph when generating code. This limit is only required when generating code.

## See Also

### Functions

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

### Objects

`robotics.LidarSLAM` | `robotics.PoseGraph` | `robotics.PoseGraph3D`

### Topics

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

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

**Introduced in R2018a**

## 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

### **Topics**

“Coordinate Transformations in Robotics”

**Introduced in R2015a**

## eul2rotm

Convert Euler angles to rotation matrix

### Syntax

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

### Description

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

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

### Examples

#### Convert Euler Angles to Rotation Matrix

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

```
rotmZYX = 3×3
```

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

### Convert Euler Angles to Rotation Matrix Using ZYZ Axis Order

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

```
rotmZYZ = 3×3
```

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

## Input Arguments

### **eul** — Euler rotation angles

*n*-by-3 matrix

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

Example: [0 0 1.5708]

### **sequence** — Axis rotation sequence

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

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

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

Data Types: string | char

## Output Arguments

### **rotm** — Rotation matrix

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

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

Example:  $[0 \ 0 \ 1; \ 0 \ 1 \ 0; \ -1 \ 0 \ 0]$

## Extended Capabilities

### C/C++ Code Generation

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

### See Also

rotm2eul

### Topics

“Coordinate Transformations in Robotics”

**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 = 4x4
```

```
    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 = 4x4
```

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

## Input Arguments

### **eul** — Euler rotation angles

*n*-by-3 matrix

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

Example: [0 0 1.5708]

### **sequence** — Axis rotation sequence

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

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

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

Data Types: string | char

# Output Arguments

## **tform** — Homogeneous transformation

4-by-4-by- $n$  matrix

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

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

# Extended Capabilities

## **C/C++ Code Generation**

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

## **See Also**

`tform2eul`

## **Topics**

“Coordinate Transformations in Robotics”

**Introduced in R2015a**

# 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' | 'YZZ' | 'ZXY' | 'ZXZ' | 'YXZ' | 'YXY' | 'YZX' | 'XYZ' | 'XYX' | 'XZY' | 'XZX'

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

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

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

Data Types: char | string

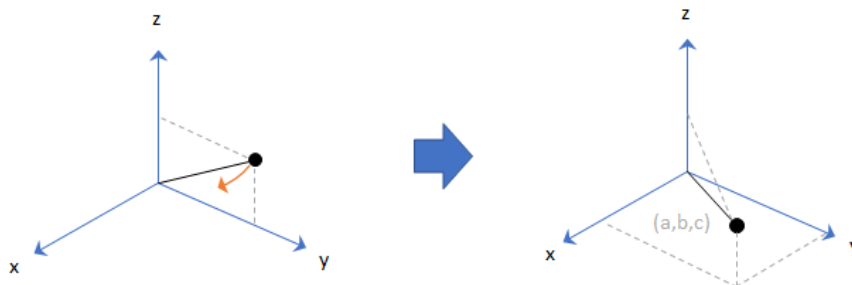
### **rotationType — Type of rotation**

'point' | 'frame'

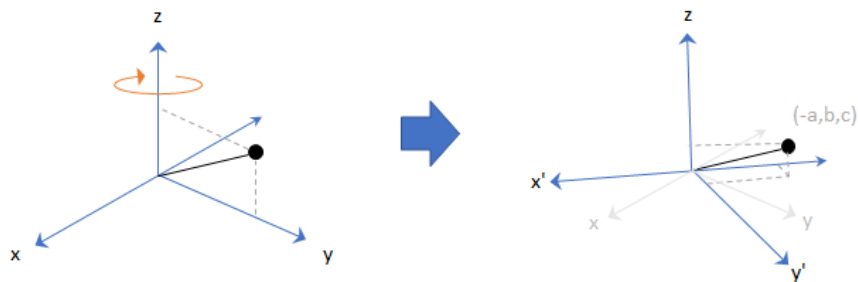
Type of rotation, specified as 'point' or 'frame'.

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

Point Rotation



Frame Rotation



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™.

**Introduced in R2018a**

# 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' | 'YZZ' | 'ZXY' | 'ZXZ' | 'YXZ' | 'YXY' | 'YZX' | 'XYZ' | 'XYX' | 'XZY' | 'XZX'

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

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

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

Data Types: char | string

### **rotationType — Type of rotation**

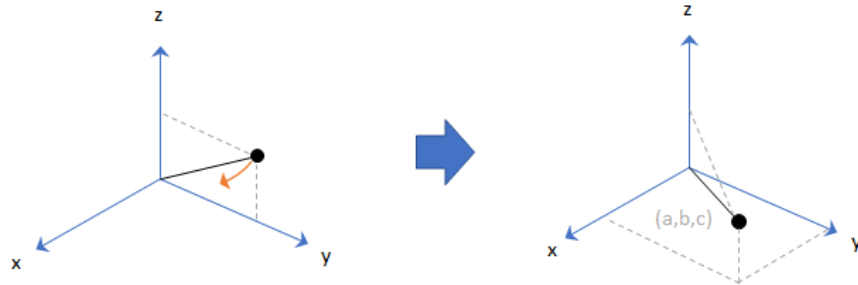
'point' | 'frame'

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

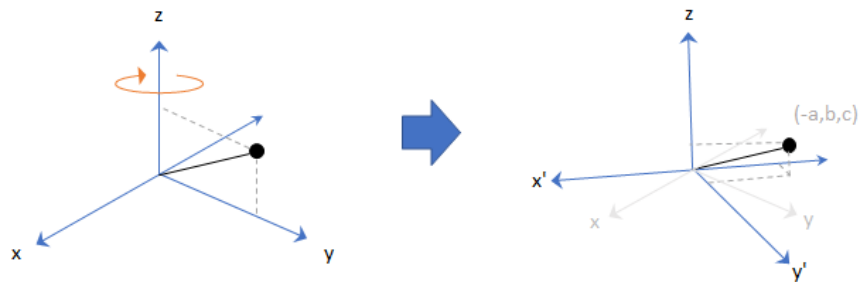
In a point rotation, the frame is static and the point moves. In a frame rotation, the point is static and the frame moves. Point rotation and frame rotation define equivalent angular displacements but in opposite directions.



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™.

**Introduced in R2018a**

# environment

**Package:** robotics

Environmental inputs for UAV

## Syntax

```
envStruct = environment(uavGuidanceModel)
```

## Description

---

**Note** This function requires you to install the UAV Library for Robotics System Toolbox. To install add-ons, use `roboticsAddons` and select the desired add-on.

---

`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.

**Note:** To use UAV algorithms, you must install the UAV Library for Robotics System Toolbox®. To install, use `roboticsAddons`.

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 fixed-wing UAV states as a 13-by-*n* matrix.

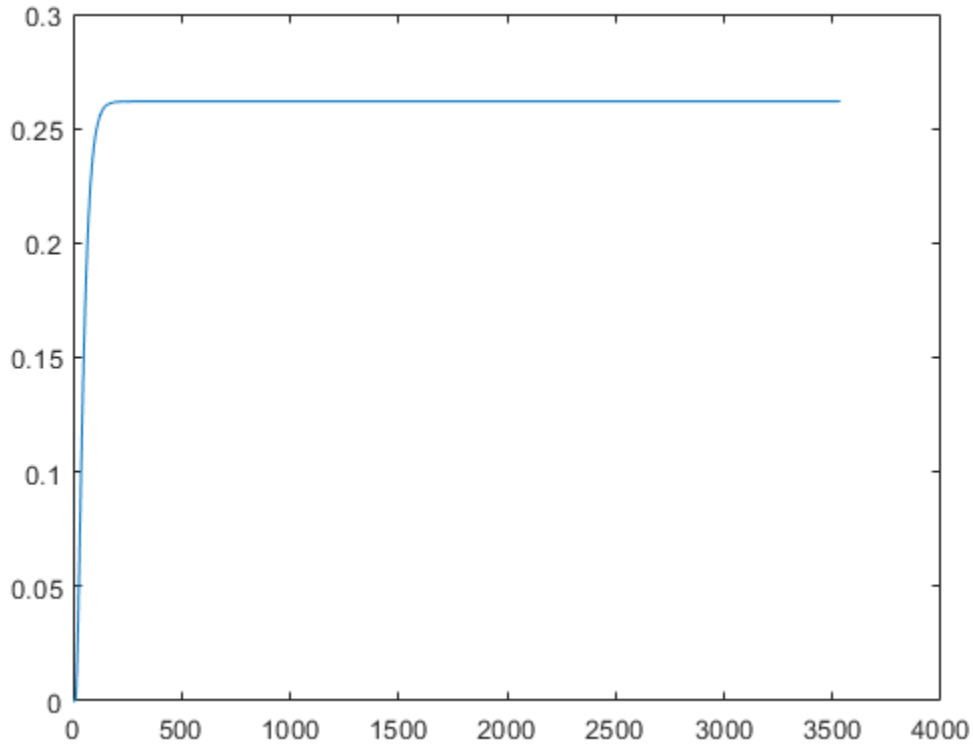
```
simOut = ode45(@(~,x)derivative(model,x,u,e), [0 3], s);  
size(simOut.y)
```

```
ans = 1×2
```

```
13      3536
```

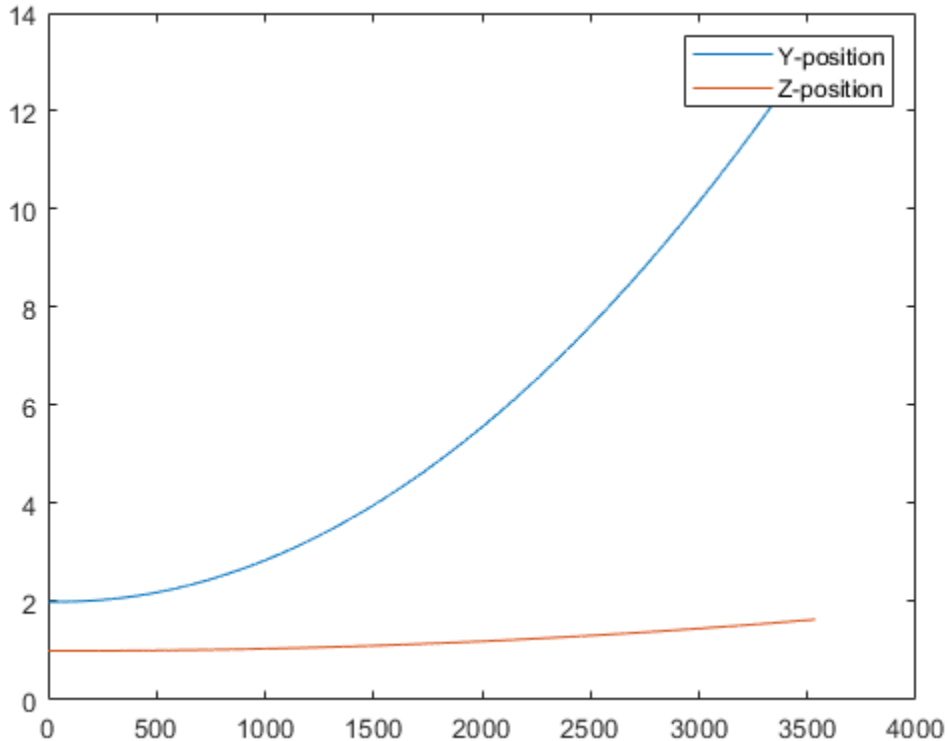
Plot the change in roll angle based on the simulation output. The roll angle (the X Euler angle) is the 9th row of the `simOut.y` output.

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



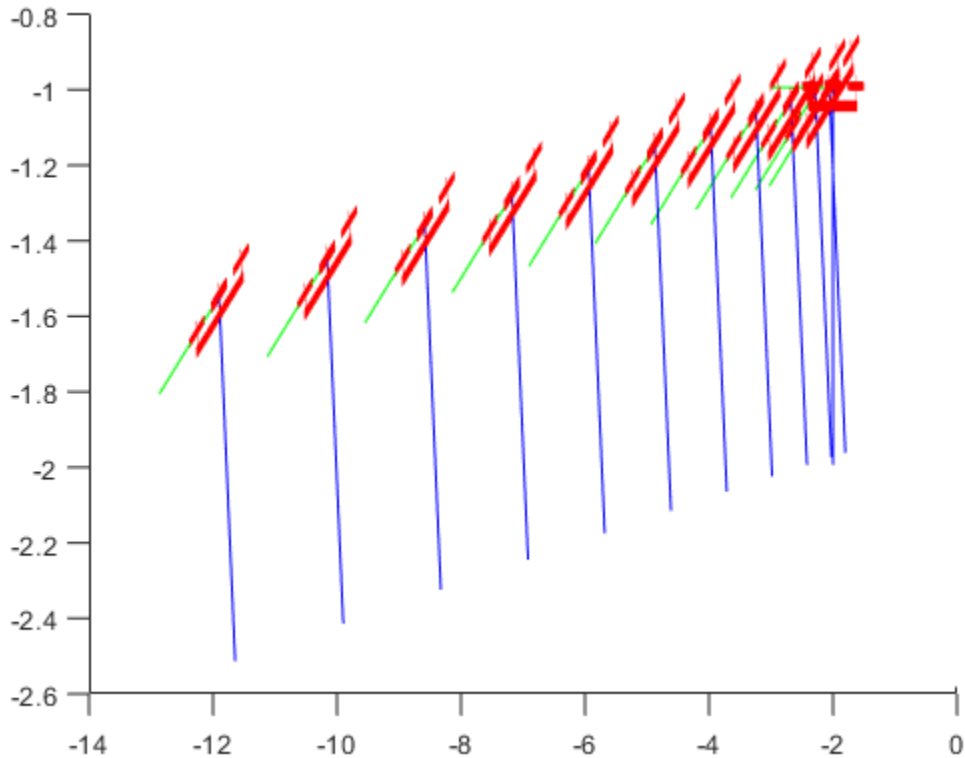
Plot the change in the Y and Z positions. With the specified thrust and roll angle, the multicopter should fly over and lose some altitude. A positive value for Z is expected as positive Z is down.

```
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.

**Note:** To use UAV algorithms, you must install the UAV Library for Robotics System Toolbox®. To install, use `roboticsAddons`.

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; % 10 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)
```

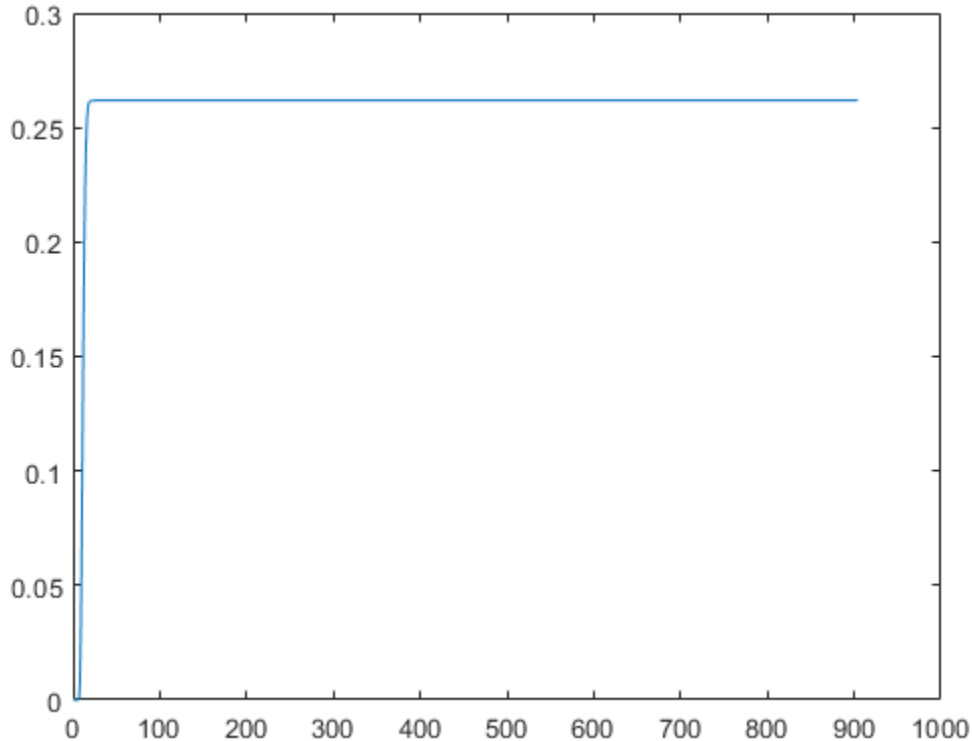
```
ans = 1x2
```

```
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,:))
```





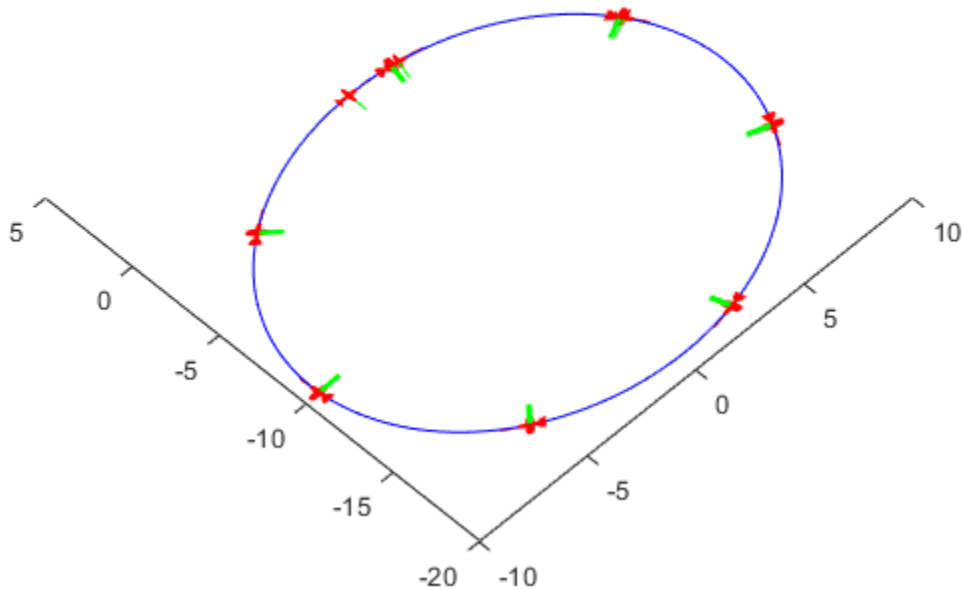
You can also plot the fixed-wing trajectory using `plotTransforms`. Create the translation and rotation vectors from the simulated state. Downsample (every 30th element) and transpose the `simOut` elements, and convert the Euler angles to quaternions. Specify the mesh as the `fixedwing.stl` file and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

## exp

Exponential of quaternion array

## Syntax

`B = exp(A)`

## Description

`B = exp(A)` computes the exponential of the elements of the quaternion array `A`.

## Examples

### Exponential of Quaternion Array

Create a 4-by-1 quaternion array `A`.

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

```
A = 4x1 quaternion array
    16 + 2i + 3j + 13k
     5 + 11i + 10j + 8k
     9 + 7i + 6j + 12k
     4 + 14i + 15j + 1k
```

Compute the exponential of `A`.

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

```
B = 4x1 quaternion array
 5.3525e+06 + 1.0516e+06i + 1.5774e+06j + 6.8352e+06k
 -57.359 - 89.189i - 81.081j - 64.865k
-6799.1 + 2039.1i + 1747.8j + 3495.6k
 -6.66 + 36.931i + 39.569j + 2.6379k
```

## Input Arguments

### **A — Input quaternion**

scalar | vector | matrix | multidimensional array

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

Data Types: quaternion

## Output Arguments

### **B — Result**

scalar | vector | matrix | multidimensional array

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

Data Types: quaternion

## Algorithms

Given a quaternion  $A = a + bi + cj + dk = a + \bar{v}$ , 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 R2018b**

# findEdgeID

**Package:** robotics

Find edge ID of edge

## Syntax

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

## Description

`edgeID = findEdgeID(poseGraph, edge)` finds the edge ID for a specified edge. Edges are defined by the IDs of the two nodes that connect them.

## Input Arguments

**poseGraph — Pose graph**

PoseGraph object | PoseGraph3D object

Pose graph, specified as a PoseGraph or PoseGraph3D object.

**edge — Edge in pose graph**

two-element vector

Edge in pose graph, specified as a two-element vector that lists the IDs of the two nodes that the edge connects.

## Output Arguments

**edgeID — Edge ID**

positive integer

Edge IDs, returned as a positive integer.



## Extended Capabilities

### C/C++ Code Generation

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

Use this syntax when constructing PoseGraph or PoseGraph3D objects for code generation:

```
poseGraph =  
robotics.PoseGraph('MaxNumEdges',maxEdges,'MaxNumNodes',maxNodes)
```

specifies an upper bound on the number of edges and nodes allowed in the pose graph when generating code. This limit is only required when generating code.

### See Also

#### Functions

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

#### Objects

`robotics.LidarSLAM` | `robotics.PoseGraph` | `robotics.PoseGraph3D`

### Topics

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

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

**Introduced in R2018a**

# get

Get ROS parameter value

## Syntax

```
pvalue = get(ptree)
pvalue = get(ptree,paramname)
pvalue = get(ptree,namespace)
```

## Description

`pvalue = get(ptree)` returns a dictionary of parameter values under the root namespace: `/`. The dictionary is stored in a structure.

`pvalue = get(ptree,paramname)` gets the value of the parameter with the name `paramname` from the parameter tree object `ptree`.

`pvalue = get(ptree,namespace)` returns a dictionary of parameter values under the specified namespace.

The following ROS data types are supported as values of parameters. For each ROS data type, the corresponding MATLAB data type is also listed.

- 32-bit integer — `int32`
- boolean — `logical`
- double — `double`
- strings — string scalar, `string`, or character vector, `char`
- list — cell array (`cell`)
- dictionary — structure (`struct`)

## Examples

## Set and Get Parameter Value

Create the parameter tree. A ROS network must be available using `rosinit`.

```
rosinit
```

```
Initializing ROS master on http://bat5742win64:49844/.  
Initializing global node /matlab_global_node_94216 with NodeURI http://bat5742win64:49
```

```
ptree = rosparam;
```

Set a parameter value. You can also use the simplified version without a parameter tree:

```
rosparam set 'DoubleParam' 1.0
```

```
set(ptree, 'DoubleParam', 1.0)
```

Get the parameter value.

```
get(ptree, 'DoubleParam')
```

```
ans = 1
```

Alternatively, use the simplified versions without using the parameter tree.

```
rosparam set 'DoubleParam' 2.0
```

```
rosparam get 'DoubleParam'
```

```
2
```

Disconnect from ROS network.

```
roshutdown
```

```
Shutting down global node /matlab_global_node_94216 with NodeURI http://bat5742win64:49  
Shutting down ROS master on http://bat5742win64:49844/.
```

## Input Arguments

### **ptree** — Parameter tree

ParameterTree object handle

Parameter tree, specified as a ParameterTree object handle. Create this object using the `rosparam` function.

**paramname — ROS parameter name**

string scalar | character vector

ROS parameter name, specified as a character vector. This string must match the parameter name exactly.

**namespace — ROS parameter namespace**

string scalar | character vector

ROS parameter namespace, specified as a string scalar or character vector. All parameter names starting with this string are listed when calling `rosparam("list", namespace)`.

## Output Arguments

**pvalue — ROS parameter value or dictionary of values**

int32 | logical | double | character vector | cell array | structure

ROS parameter value, returned as a supported MATLAB data type. When specifying the namespace input argument, `pvalue` is returned as a dictionary of parameter values under the specified namespace. The dictionary is represented in MATLAB as a structure.

The following ROS data types are supported as values of parameters. For each ROS data type, the corresponding MATLAB data type is also listed.

- 32-bit integer — `int32`
- boolean — `logical`
- double — `double`
- string — character vector (`char`)
- list — cell array (`cell`)
- dictionary — structure (`struct`)

## Limitations

Base64-encoded binary data and iso8601 data from ROS are not supported.

## **See Also**

rosparam | set

**Introduced in R2015a**

## getParticles

**Package:** robotics

Get particles from localization algorithm

### Syntax

```
[particles,weights] = getParticles(mcl)
```

### Description

`[particles,weights] = getParticles(mcl)` returns the current particles used by the `MonteCarloLocalization` object. `particles` is an  $n$ -by-3 matrix that contains the location and orientation of each particle. Each row has a corresponding weight value specified in `weights`. The number of rows can change with each iteration of the MCL algorithm. Use this method to extract the particles and analyze them separately from the algorithm.

### Examples

#### Get Particles from Monte Carlo Localization Algorithm

Get particles from the particle filter used in the Monte Carlo Localization object.

Create a map and a Monte Carlo localization object.

```
map = robotics.BinaryOccupancyGrid(10,10,20);  
mcl = robotics.MonteCarloLocalization(map);
```

Create robot data for the range sensor and pose.

```
ranges = 10*ones(1,300);  
ranges(1,130:170) = 1.0;  
angles = linspace(-pi/2,pi/2,300);  
odometryPose = [0 0 0];
```

Initialize particles using `step`.

```
[isUpdated,estimatedPose,covariance] = step(mcl,odometryPose,ranges,angles);
```

Get particles from the updated object.

```
[particles,weights] = getParticles(mcl);
```

## Input Arguments

**mcl** — MonteCarloLocalization object

handle

robotics.MonteCarloLocalization object, specified as an object handle.

## Output Arguments

**particles** — Estimation particles

*n*-by-3 vector

Estimation particles, returned as an *n*-by-3 vector, [*x y theta*]. Each row corresponds to the position and orientation of a single particle. The length can change with each iteration of the algorithm.

**weights** — Weights of particles

*n*-by-1 vector

Weights of particles, returned as a *n*-by-1 vector. Each row corresponds to the weight of the particle in the matching row of `particles`. These weights are used in the final estimate of the pose of the robot. The length can change with each iteration of the algorithm.

## See Also

robotics.MonteCarloLocalization

## Topics

“Monte Carlo Localization Algorithm”

**Introduced in R2016a**



## getFile

Get file from device

### Syntax

```
getFile(device,remoteSource)
getFile(device,remoteSource,localDestination)
```

### Description

`getFile(device,remoteSource)` copies the specified file from the ROS device to the MATLAB current folder. Wildcards are supported.

`getFile(device,remoteSource,localDestination)` copies the remote file to a destination path. Specify a file name at the end of the destination path to copy with a custom file name.

### Examples

#### Put, Get, and Delete Files on ROS Device

Put a file from your host computer onto a ROS device, get it back, and then delete it.

Connect to a ROS device. Specify the device address, user name, and password of your ROS device.

```
d = rosdevice('192.168.154.131','user','password');
```

Put a new text file that is in the MATLAB® current folder onto the ROS device. The destination folder must exist.

```
putFile(d,'test_file.txt','/home/user/test_folder')
```

Get a text file from the ROS device. You can get any file, not just ones added from MATLAB®. By default, the file is added to the MATLAB current folder.

```
getFile(d, '/home/user/test_folder/test_file.txt')
```

Delete the text file on the ROS device.

```
deleteFile(d, '/home/user/test_folder/test_file.txt')
```

### Put, Get, and Delete Files on ROS Device Using Wildcards

Put a file from your host computer onto a ROS device, get it back, and then delete it. Use wildcards to search for all matching files.

**Note:** You must have a valid ROS device to connect to at the IP address specified in the example.

Connect to a ROS device. Specify the device address, user name, and password of your ROS device.

```
d = rosdevice('192.168.203.129', 'user', 'password');
```

Put all text files at the specified path onto the ROS device. The destination folder must exist.

```
putFile(d, 'C:/MATLAB/*.txt', '/home/user/test_folder')
```

Get all text files from the ROS device. You can get any files, not just ones added from MATLAB®. By default, the files are added to the MATLAB current folder.

```
getFile(d, '/home/user/test_folder/*.txt')
```

Delete all text files on the ROS device at the specified folder.

```
deleteFile(d, '/home/user/test_folder/*.txt')
```

## Input Arguments

### **device** — ROS device

rosdevice object

ROS device, specified as a rosdevice object.

**remoteSource — Path and name of file on ROS device**

source path

Path and name of the file on the ROS device. Specify the path as a character vector. You can use an absolute path or a relative path from the MATLAB Current Folder. Use the path and file naming conventions of the operating system on your host computer.

Example:  `'/home/user/test_folder/test_file.txt'`

Data Types: char

**localDestination — Destination folder path and optional file name**

character vector

Destination folder path and optional file name, specified as a character vector. Specify a file name at the end of the destination path to copy with a custom file name. Use the host computer path and file naming conventions.

Example:  `'C:/User/username/test_folder'`

Data Types: char

**See Also**

`deleteFile` | `dir` | `openShell` | `putFile` | `rosdevice` | `system`

**Introduced in R2016b**

# getTransform

Retrieve transformation between two coordinate frames

---

**Note** The behavior of `getTransform` changed in R2018a. When using the `tftree` input argument, the function no longer returns an empty transform when the transform is unavailable and no `sourcetime` is specified. If `getTransform` waits for the specified timeout period and the transform is still not available, the function returns an error. The timeout period is 0 by default.

---

## Syntax

```
tf = getTransform(tftree, targetframe, sourceframe)
tf = getTransform(tftree, targetframe, sourceframe, sourcetime)
tf = getTransform( ___, "Timeout", timeout)

tf = getTransform(bagSel, targetframe, sourceframe)
tf = getTransform(bagSel, targetframe, sourceframe, sourcetime)
```

## Description

`tf = getTransform(tftree, targetframe, sourceframe)` returns the latest known transformation between two coordinate frames in `tftree`. Create the `tftree` object using `rostopf`, which requires a connection to a ROS network.

Transformations are structured as a 3-D translation (three-element vector) and a 3-D rotation (quaternion).

`tf = getTransform(tftree, targetframe, sourceframe, sourcetime)` returns the transformation from the `tftree` at the given source time. If the transformation is not available at that time, an error is displayed.

`tf = getTransform( ___, "Timeout", timeout)` also specifies a timeout period, in seconds, to wait for the transformation to be available. Otherwise, if the transformation doesn't become available in the timeout period, the function returns an error. This option can be combined with the previous syntaxes.

`tf = getTransform(bagSel, targetframe, sourceframe)` returns the latest transformation between two frames in the rosbag in `bagSel`. To get the `bagSel` input, load a rosbag using `rosbag`.

`tf = getTransform(bagSel, targetframe, sourceframe, sourcetime)` returns the transformation at the specified `sourcetime` in the rosbag in `bagSel`.

## Examples

### Get ROS Transformations and Apply to ROS Messages

This example shows how to set up a ROS transformation tree and transform frames based on this information. It uses time-buffered transformations to access transformations at different times.

Create a ROS transformation tree. Use `roslaunch` to connect to a ROS network. Replace `ipaddress` with your ROS network address.

```
ipaddress = '192.168.203.129';
roslaunch(ipaddress)
tftree = rostf;
pause(1)
```

```
Initializing global node /matlab_global_node_60416 with NodeURI http://192.168.203.1:5
```

Look at the available frames on the transformation tree.

```
tftree.AvailableFrames
```

```
ans =
```

```
36x1 cell array
```

```

{'base_footprint'      }
{'base_link'          }
{'camera_depth_frame' }
{'camera_depth_optical_frame'}
{'camera_link'        }
{'camera_rgb_frame'   }
{'camera_rgb_optical_frame' }
```

```
{'caster_back_link'      }
{'caster_front_link'     }
{'cliff_sensor_front_link'}
{'cliff_sensor_left_link'}
{'cliff_sensor_right_link'}
{'gyro_link'             }
{'mount_asus_xtion_pro_link'}
{'odom'                  }
{'plate_bottom_link'     }
{'plate_middle_link'     }
{'plate_top_link'        }
{'pole_bottom_0_link'    }
{'pole_bottom_1_link'    }
{'pole_bottom_2_link'    }
{'pole_bottom_3_link'    }
{'pole_bottom_4_link'    }
{'pole_bottom_5_link'    }
{'pole_kinect_0_link'    }
{'pole_kinect_1_link'    }
{'pole_middle_0_link'    }
{'pole_middle_1_link'    }
{'pole_middle_2_link'    }
{'pole_middle_3_link'    }
{'pole_top_0_link'       }
{'pole_top_1_link'       }
{'pole_top_2_link'       }
{'pole_top_3_link'       }
{'wheel_left_link'       }
{'wheel_right_link'      }
```

Check if the desired transformation is available now. For this example, check for the transformation from 'camera\_link' to 'base\_link'.

```
canTransform(tftree, 'base_link', 'camera_link')
```

```
ans =
```

```
logical
```

```
1
```

Get the transformation for 3 seconds from now. `getTransform` will wait until the transformation becomes available with the specified timeout.

```
desiredTime = rostime('now') + 3;
tform = getTransform(tftree,'base_link','camera_link',...
                    desiredTime,'Timeout',5);
```

Create a ROS message to transform. Messages could also be retrieved off the ROS network.

```
pt = rosmesssage('geometry_msgs/PointStamped');
pt.Header.FrameId = 'camera_link';
pt.Point.X = 3;
pt.Point.Y = 1.5;
pt.Point.Z = 0.2;
```

Transform the ROS message to the 'base\_link' frame using the desired time saved from before.

```
tfpt = transform(tftree,'base_link',pt,desiredTime);
```

*Optional:* You can also use `apply` with the stored `tform` to apply this transformation to the `pt` message.

```
tfpt2 = apply(tform,pt);
```

Shut down the ROS network.

```
roshutdn
```

```
Shutting down global node /matlab_global_node_60416 with NodeURI http://192.168.203.1:5
```

## Get Buffered Transformations from ROS Network

This example shows how to access time-buffered transformations on the ROS network. Access transformations for specific times and modify the `BufferTime` property based on your desired times.

Create a ROS transformation tree. Use `rosinit` to connect to a ROS network. Replace `ipaddress` with your ROS network address.

```
ipaddress = '192.168.203.129';
rosinit(ipaddress)
```

```
tftree = rostf;  
pause(2);
```

Initializing global node /matlab\_global\_node\_29163 with NodeURI http://192.168.203.1:5

Get the transformation from 1 second ago.

```
desiredTime = rostime('now') - 1;  
tform = getTransform(tftree, 'base_link', 'camera_link', desiredTime);
```

The transformation buffer time is 10 seconds by default. Modify the `BufferTime` property of the transformation tree to increase the buffer time and wait for that buffer to fill.

```
tftree.BufferTime = 15;  
pause(15);
```

Get the transformation from 12 seconds ago.

```
desiredTime = rostime('now') - 12;  
tform = getTransform(tftree, 'base_link', 'camera_link', desiredTime);
```

You can also get transformations at a time in the future. `getTransform` will wait until the transformation is available. You can also specify a timeout to error out if no transformation is found. This example waits 5 seconds for the transformation at 3 seconds from now to be available.

```
desiredTime = rostime('now') + 3;  
tform = getTransform(tftree, 'base_link', 'camera_link', desiredTime, 'Timeout', 5);
```

Shut down the ROS network.

```
roshutdown
```

Shutting down global node /matlab\_global\_node\_29163 with NodeURI http://192.168.203.1:5

### Get Transformations from rosbag File

Get transformations from rosbag (`.bag`) files by loading the rosbag and checking the available frames. From these frames, use `getTransform` to query the transformation between two coordinate frames.

Load the rosbag.



```
bag = rosbag('ros_turtlesim.bag');
```

Get a list of available frames.

```
frames = bag.AvailableFrames;
```

Get the latest transformation between two coordinate frames.

```
tf = getTransform(bag, 'world', frames{1});
```

Check for a transformation available at a specific time and retrieve the transformation. Use `canTransform` to check if the transformation is available. Specify the time using `rostime`.

```
tfTime = rostime(bag.StartTime + 1);
if (canTransform(bag, 'world', frames{1}, tfTime))
    tf2 = getTransform(bag, 'world', frames{1}, tfTime);
end
```

## Input Arguments

### **tftree** — ROS transformation tree

TransformationTree object handle

ROS transformation tree, specified as a TransformationTree object handle. You can create a transformation tree by calling the `rostopic` function.

### **bagSel** — Selection of rosbag messages

BagSelection object handle

Selection of rosbag messages, specified as a BagSelection object handle. To create a selection of rosbag messages, use `rostopic`.

### **targetframe** — Target coordinate frame

string scalar | character vector

Target coordinate frame, specified as a string scalar or character vector. You can view the available frames for transformation by calling `tftree.AvailableFrames`.

### **sourceframe** — Initial coordinate frame

string scalar | character vector

Initial coordinate frame, specified as a string scalar or character vector. You can view the available frames for transformation by calling `tfTree.AvailableFrames`.

### **sourcetime** — ROS or system time

Time object handle

ROS or system time, specified as a Time object handle. By default, `sourcetime` is the ROS simulation time published on the `clock` topic. If the `use_sim_time` ROS parameter is set to `true`, `sourcetime` returns the system time. You can create a Time object using `rostime`.

### **timeout** — Timeout for receiving transform

0 (default) | scalar in seconds

Timeout for receiving transform, specified as a scalar in seconds. The function returns an error if the timeout is reached and no transform becomes available.

## Output Arguments

### **tf** — Transformation between coordinate frames

TransformStamped object handle

Transformation between coordinate frames, returned as a TransformStamped object handle. Transformations are structured as a 3-D translation (three-element vector) and a 3-D rotation (quaternion).

## See Also

`canTransform` | `rosviz` | `rostopic` | `tf` | `tf2` | `tf2_ros` | `waitForTransform`

**Introduced in R2015a**

# has

Check if ROS parameter name exists

## Syntax

```
exists = has(ptree,paramname)
```

## Description

`exists = has(ptree,paramname)` checks if the parameter with name `paramname` exists in the parameter tree, `ptree`.

## Examples

### Check If ROS Parameter Exists

Connect to a ROS network. Create a parameter tree and check for the 'MyParam' parameter.

```
rosinit
```

```
Initializing ROS master on http://bat5742win64:61561/.
```

```
Initializing global node /matlab_global_node_08560 with NodeURI http://bat5742win64:61561/.
```

```
ptree = rosparam;  
has(ptree,'MyParam')
```

```
ans = logical  
     0
```

Set the 'MyParam' parameter and verify it exists. Disconnect from ROS network.

```
set(ptree,'MyParam','test')  
has(ptree,'MyParam')
```

```
ans = logical  
     1
```

```
rosshutdown
```

```
Shutting down global node /matlab_global_node_08560 with NodeURI http://bat5742win64:6  
Shutting down ROS master on http://bat5742win64:61561/.
```

## Input Arguments

### **ptree** — Parameter tree

ParameterTree object handle

Parameter tree, specified as a ParameterTree object handle. Create this object using the `rosparam` function.

### **paramname** — ROS parameter name

string scalar | character vector

ROS parameter name, specified as a string scalar or character vector. This string must match the parameter name exactly.

## Output Arguments

### **exists** — Flag indicating whether the parameter exists

true | false

Flag indicating whether the parameter exists, returned as `true` or `false`.

## See Also

`get` | `rosparam` | `search` | `set`

**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`

## **Topics**

“Coordinate Transformations in Robotics”

**Introduced in R2015a**

## info

**Package:** robotics

Characteristic information about PurePursuit object

## Syntax

```
controllerInfo = info(controller)
```

## Description

`controllerInfo = info(controller)` returns a structure, `controllerInfo`, with additional information about the status of the PurePursuit object, `controller`. The structure contains the fields, `RobotPose` and `LookaheadPoint`.

## Examples

### Get Additional PurePursuit Object Information

Use the `info` method to get more information about a PurePursuit object. `info` returns two fields, `RobotPose` and `LookaheadPoint`, which correspond to the current position and orientation of the robot and the point on the path used to compute outputs from the last call of the object.

Create a PurePursuit object.

```
pp = robotics.PurePursuit;
```

Assign waypoints.

```
pp.Waypoints = [0 0;1 1];
```

Compute control commands using the `pp` object with the initial pose `[x y theta]` given as the input.

```
[v,w] = pp([0 0 0]);
```

Get additional information.

```
s = info(pp)
```

```
s = struct with fields:
    RobotPose: [0 0 0]
    LookaheadPoint: [0.7071 0.7071]
```

## Input Arguments

### **controller** — Pure pursuit controller

PurePursuit object

Pure pursuit controller, specified as a PurePursuit object.

## Output Arguments

### **controllerInfo** — Information on the PurePursuit object

structure

Information on the PurePursuit object, returned as a structure. The structure contains two fields:

- **RobotPose** - A three-element vector in the form `[x y theta]` that corresponds to the x-y position and orientation of the robot. The angle, `theta`, is measured in radians with positive angles measured counterclockwise from the x-axis.
- **LookaheadPoint**- A two-element vector in the form `[x y]`. The location is a point on the path that was used to compute outputs of the last call to the object.

## See Also

`robotics.PurePursuit`

## Topics

“Pure Pursuit Controller”



**Introduced in R2015a**

# importrobot

Import rigid body tree model from URDF file, text, or Simscape Multibody model

## Syntax

```
robot = importrobot(filename)
robot = importrobot(URDFtext)

[robot,importInfo] = importrobot(model)
___ = importrobot( ___,Name,Value)
```

## Description

`robot = importrobot(filename)` returns a `robotics.RigidBodyTree` object by parsing the Unified Robot Description Format (URDF) file specified by `filename`.

`robot = importrobot(URDFtext)` parses the URDF text. Specify `URDFtext` as a string scalar or character vector.

`[robot,importInfo] = importrobot(model)` imports a Simscape Multibody model and returns an equivalent `robotics.RigidBodyTree` object and info about the import in `importInfo`. Only fixed, prismatic, and revolute joints are supported in the output `RigidBodyTree` object. Use the “Simscape Multibody Model Import” on page 2-0 name-value pairs to import a model that uses other joint types, constraint blocks, or variable inertias.

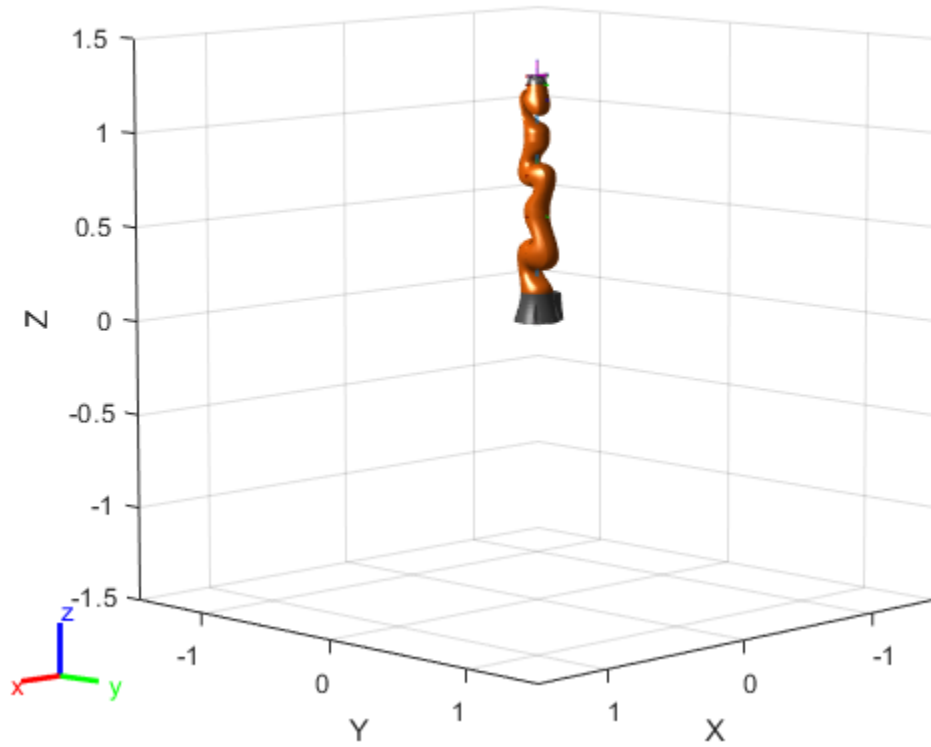
`___ = importrobot( ___,Name,Value)` provides additional options specified by `Name,Value` pair arguments. Use any of the previous syntaxes. Only certain name-value pairs apply depending on whether you convert from a URDF file or a Simscape Multibody model.

## Examples

### Import Robot from URDF File

Import a URDF file as a `robotics.RigidBodyTree` object.

```
robot = importrobot('iiwa14.urdf')  
  
robot =  
  RigidBodyTree with properties:  
  
    NumBodies: 10  
    Bodies: {1x10 cell}  
    Base: [1x1 robotics.RigidBody]  
    BodyNames: {1x10 cell}  
    BaseName: 'world'  
    Gravity: [0 0 0]  
    DataFormat: 'struct'  
  
show(robot)
```



```
ans =  
  Axes (Primary) with properties:  
    XLim: [-1.5000 1.5000]  
    YLim: [-1.5000 1.5000]  
    XScale: 'linear'  
    YScale: 'linear'  
    GridLineStyle: '-'  
    Position: [0.1300 0.1100 0.7750 0.8150]  
    Units: 'normalized'
```

Show all properties

## Import Robot from URDF Character Vector

Specify the URDF character vector. This character vector is a minimalist description for creating a valid robot model.

```
URDFtext = '<?xml version="1.0" ?><robot name="min"><link name="L0"/></robot>';
```

Import the robot model. The description creates a `RigidBodyTree` object that has only a robot base link named 'L0'.

```
robot = importrobot(URDFtext)

robot =
  RigidBodyTree with properties:

    NumBodies: 0
    Bodies: {1x0 cell}
    Base: [1x1 robotics.RigidBody]
    BodyNames: {1x0 cell}
    BaseName: 'L0'
    Gravity: [0 0 0]
    DataFormat: 'struct'
```

## Display Robot Model with Visual Geometries

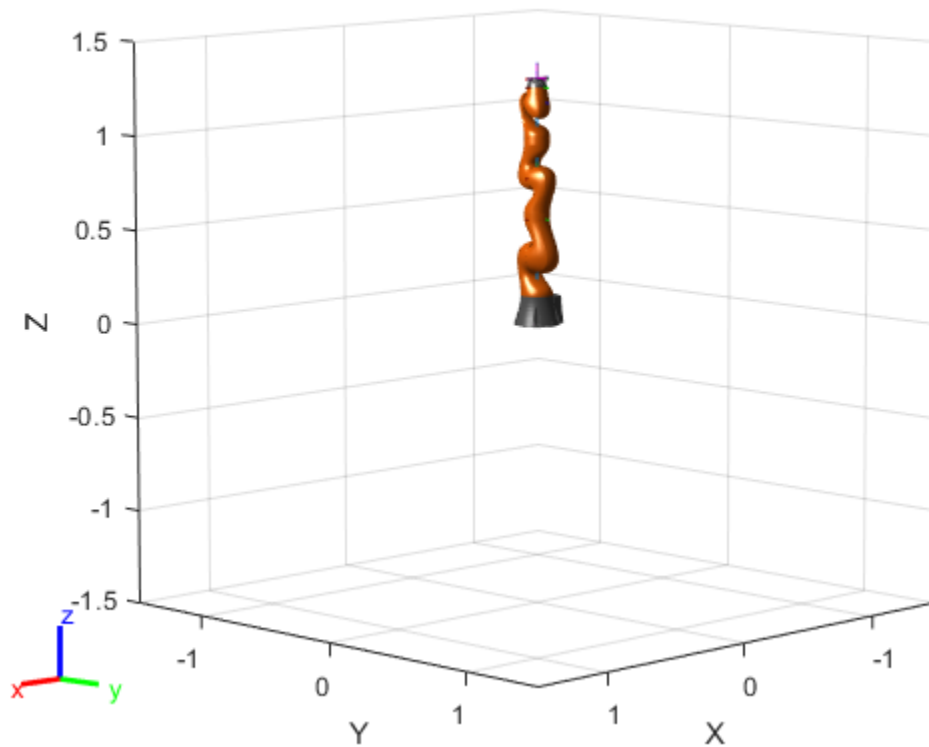
You can import robots that have `.stl` files associated with the Unified Robot Description format (URDF) file to describe the visual geometries of the robot. Each rigid body has an individual visual geometry specified. The `importrobot` function parses the URDF file to get the robot model and visual geometries. Use the `show` function to visualize the robot model in a figure. You can then interact with the model by clicking components to inspect them and right-clicking to toggle visibility.

Import a robot model as a URDF file. The `.stl` file locations must be properly specified in this URDF. To add other `.stl` files to individual rigid bodies, see `addVisual`.

```
robot = importrobot('iiwa14.urdf');
```

Visualize the robot with the associated visual model. Click bodies or frames to inspect them. Right-click bodies to toggle visibility for each visual geometry.

```
show(robot);
```



### Import Simscape™ Multibody™ model to RigidBodyTree Object

Import an existing Simscape™ Multibody™ robot model into the Robotics System Toolbox™ as a `robotics.RigidBodyTree` object.

Open the Simscape™ Multibody™ model. This is a model for a humanoid robot.

```
open_system('sm_import_humanoid_urdf.slx')
```

Import the model.

```
[robot,importInfo] = importrobot(gcs)
```

```

robot =
  RigidBodyTree with properties:

    NumBodies: 21
    Bodies: {1x21 cell}
    Base: [1x1 robotics.RigidBody]
    BodyNames: {'Body01' 'Body02' 'Body03' 'Body04' 'Body05' 'Body06' 'Body07'
    BaseName: 'Base'
    Gravity: [0 0 -9.8066]
    DataFormat: 'struct'

importInfo =
  RigidBodyTreeImportInfo with properties:

    SourceModelName: 'sm_import_humanoid_urdf'
    RigidBodyTree: [1x1 robotics.RigidBodyTree]
    BlockConversionInfo: [1x1 struct]

```

Display details about the created `robotics.RigidBodyTree` object.

```
showdetails(importInfo)
```

```
-----
Robot: (21 bodies)
```

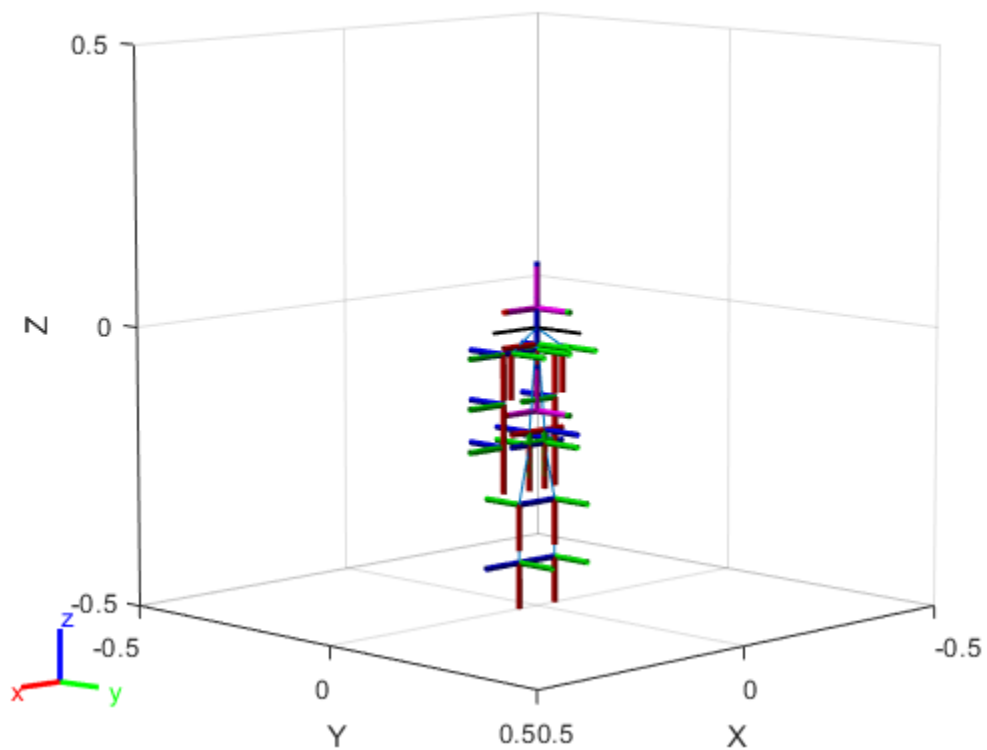
Idx	Body Name	Simulink Source Blocks	Joint Name	Simulink Source Blocks
1	Body01	Info   List   Highlight	Joint01	Info   List   Highlight
2	Body02	Info   List   Highlight	Joint02	Info   List   Highlight
3	Body03	Info   List   Highlight	Joint03	Info   List   Highlight
4	Body04	Info   List   Highlight	Joint04	Info   List   Highlight
5	Body05	Info   List   Highlight	Joint05	Info   List   Highlight
6	Body06	Info   List   Highlight	Joint06	Info   List   Highlight
7	Body07	Info   List   Highlight	Joint07	Info   List   Highlight
8	Body08	Info   List   Highlight	Joint08	Info   List   Highlight
9	Body09	Info   List   Highlight	Joint09	Info   List   Highlight
10	Body10	Info   List   Highlight	Joint10	Info   List   Highlight
11	Body11	Info   List   Highlight	Joint11	Info   List   Highlight
12	Body12	Info   List   Highlight	Joint12	Info   List   Highlight
13	Body13	Info   List   Highlight	Joint13	Info   List   Highlight
14	Body14	Info   List   Highlight	Joint14	Info   List   Highlight
15	Body15	Info   List   Highlight	Joint15	Info   List   Highlight
16	Body16	Info   List   Highlight	Joint16	Info   List   Highlight

```
17      Body17  Info | List | Highlight      Joint17  Info | List | Highlight
18      Body18  Info | List | Highlight      Joint18  Info | List | Highlight
19      Body19  Info | List | Highlight      Joint19  Info | List | Highlight
20      Body20  Info | List | Highlight      Joint20  Info | List | Highlight
21      Body21  Info | List | Highlight      Joint21  Info | List | Highlight
```

-----

Show the robot in a figure window. You can see the full humanoid shape based on the frames displayed.

```
show(robot)
```



```
ans =  
    Axes (Primary) with properties:
```



```
        XLim: [-0.5000 0.5000]
        YLim: [-0.5000 0.5000]
        XScale: 'linear'
        YScale: 'linear'
GridLineStyle: '-'
        Position: [0.1300 0.1100 0.7750 0.8150]
        Units: 'normalized'
```

Show all properties

## Input Arguments

### **filename** — Name of URDF file

string scalar | character vector

Name of URDF file, specified as a string scalar or character vector. This file must be a valid URDF robot description.

Example: "robot\_file.urdf"

Data Types: char | string

### **URDFtext** — URDF text

string scalar | character vector

URDF robot text, specified as a string scalar or character vector.

Example: "<?xml version="1.0" ?><robot name="min"><link name="L0"/></robot>"

Data Types: char | string

### **model** — Simscape Multibody model

model handle | string scalar | character vector

Simscape Multibody model, specified as a model handle, string scalar, or character vector.

Data Types: double | 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: `"MeshPath", {"../arm_meshes", "../body_meshes"}`

#### URDF File Import

##### **MeshPath** — Relative search paths for mesh files

string scalar | character vector | cell array of string scalars or character vectors

Relative search paths for mesh files, specified as a string scalar, character vector, or cell array of string scalars or character vectors. Mesh files must still be specified inside the URDF file, but `MeshPath` defines the relative paths for these specified files. When using this function, the URDF importer searches for the mesh files specified in the URDF using all the specified relative paths.

Data Types: char | string | cell

#### Simscape Multibody Model Import

##### **BreakChains** — Indicates whether to break closed chains

"error" (default) | "remove-joints"

Indicates whether to break closed chains in the given `model` input, specified as "error" or "remove-joints". If you specify "remove-joints", the resulting `robot` output has chain closure joints removed. Otherwise, the function throws an error.

Data Types: char | string

##### **ConvertJoints** — Indicates whether to convert unsupported joints to fixed

"error" (default) | "convert-to-fixed"

Indicates whether to convert unsupported joints to fixed joints in the given `model` input, specified as "error" or "convert-to-fixed". If you specify "convert-to-fixed", the resulting `robot` output has any unsupported joints converted to fixed joints. Only fixed, prismatic, and revolute joints are supported in the output `RigidBodyTree` object. Otherwise, if the `model` contains unsupported joints, the function throws an error.

Data Types: char | string

**SMConstraints — Indicates whether to remove constraint blocks**

"error" (default) | "remove"

Indicates whether to remove constraint blocks in the given model input, specified as "error" or "remove". If you specify "remove", the resulting robot output has the constraints removed. Otherwise, if the model contains constraint blocks, the function throws an error.

Data Types: char | string

**VariableInertias — Indicates whether to remove variable inertia blocks**

"error" (default) | "remove"

Indicates whether to remove variable inertia blocks in the given model input, specified as "error" or "remove". If you specify "remove", the resulting robot output has the variable inertias removed. Otherwise, if the model contains variable inertia blocks, the function throws an error.

Data Types: char | string

## Output Arguments

**robot — Robot model**

RigidBodyTree object

Robot model, returned as a `robotics.RigidBodyTree` object.

---

**Note** If the gravity is not specified in the URDF file, the default `Gravity` property is set to `[0 0 0]`. Simscape Multibody uses a default of `[0 0 -9.80665]` m/s<sup>2</sup> when using `smimport` to import a URDF.

---

**importInfo — Object for storing import information**

RigidBodyTreeImportInfo object

Object for storing import information, returned as a `robotics.RigidBodyTreeImportInfo` object. This object contains the relationship between the input model and the resulting robot output.

Use `showdetails` to list all the import info for each body in the robot. Links to display the rigid body info, their corresponding blocks in the model, and highlighting specific blocks in the model are output to the command window.

Use `bodyInfo`, `bodyInfoFromBlock`, or `bodyInfoFromJoint` to get information about specific components in either the `robot` output or the `model` input.

## Tips

When importing a robot model with visual meshes, the `importrobot` function searches for the `.stl` files to assign to each rigid body using these rules:

- The function searches the raw mesh path for a specified rigid body from the URDF file. References to ROS packages have the `package:\\<pkg_name>` removed.
- Absolute paths are checked directly with no modification.
- Relative paths are checked using the following directories in order:
  - User-specified `MeshPath`
  - Current folder
  - MATLAB path
  - The folder containing the URDF file
  - One level above the folder containing the URDF file
- The file name from the mesh path in the URDF file is appended to the `MeshPath` input argument.

If the mesh file is still not found, the parser ignores the mesh file and returns a `robotics.RigidBodyTree` object without visual.

## See Also

`robotics.RigidBodyTree` | `robotics.RigidBodyTreeImportInfo`

## Topics

“Rigid Body Tree Robot Model”

**Introduced in R2017a**

# isCoreRunning

Determine if ROS core is running

## Syntax

```
running = isCoreRunning(device)
```

## Description

`running = isCoreRunning(device)` determines if the ROS core is running on the connected device.

## Examples

### Run ROS Core on ROS Device

Connect to a remote ROS device and start a ROS core. The ROS core is needed to run ROS nodes to communicate via a ROS network. You can run and stop a ROS core or node and check their status using a `rosdevice` object.

Create a connection to a ROS device. Specify the address, user name, and password of your specific ROS device. The device contains information about the ROS device, including the available ROS nodes that can be run using `runNode`.

```
ipaddress = '192.168.154.131';  
d = rosdevice(ipaddress, 'user', 'password')
```

```
d =
```

```
rosdevice with properties:
```

```
DeviceAddress: '192.168.154.131'  
Username: 'user'
```

```
    ROSFolder: '/opt/ros/hydro'  
    CatkinWorkspace: '~/catkin_ws_test'  
    AvailableNodes: {'robotcontroller' 'robotcontroller2'}
```

Run a ROS core and check if it is running.

```
runCore(d)  
running = isCoreRunning(d)
```

```
running =
```

```
    logical
```

```
    1
```

Stop the ROS core and confirm that it is no longer running.

```
stopCore(d)  
running = isCoreRunning(d)
```

```
running =
```

```
    logical
```

```
    0
```

## Input Arguments

**device** — ROS device

rosdevice object

ROS device, specified as a rosdevice object.

## Output Arguments

**running** — Status of whether ROS core is running

true | false

Status of whether ROS core is running, returned as `true` or `false`.

## See Also

`rosdevice` | `runCore` | `stopCore`

## Topics

“Generate a Standalone ROS Node from Simulink®”

**Introduced in R2016b**

# interpolate

**Package:** robotics

Interpolate poses along path segment

## Syntax

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

## Description

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

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

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

## Examples

### Interpolate Poses For Dubins Path

Create a `DubinsConnection` object.

```
dubConnObj = robotics.DubinsConnection;
```

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



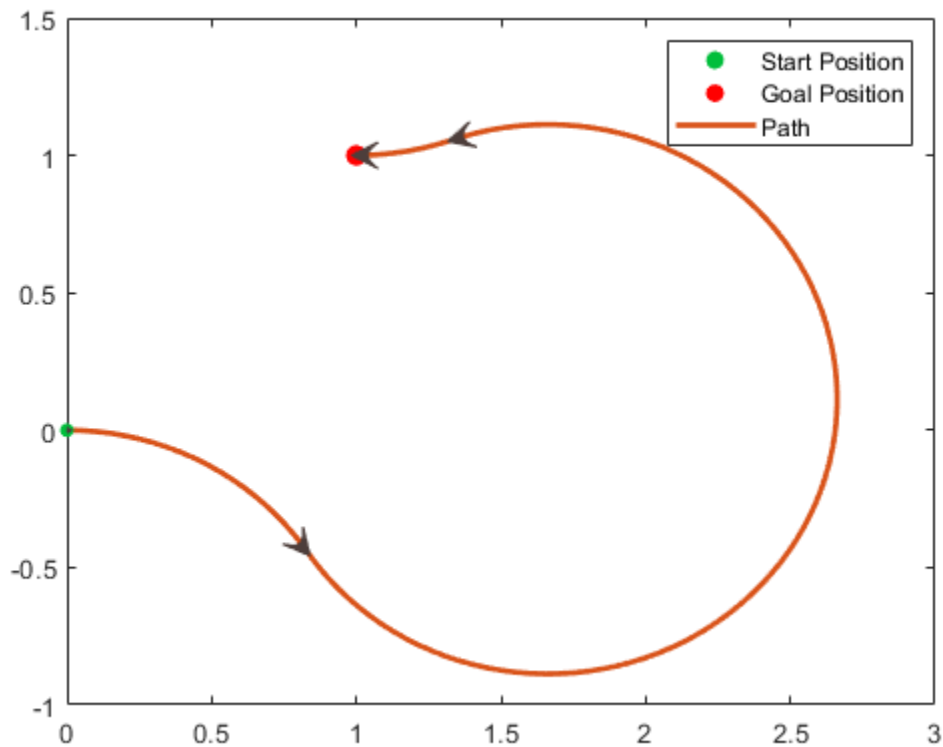
```
startPose = [0 0 0];  
goalPose = [1 1 pi];
```

Calculate a valid path segment to connect the poses.

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

Show the generated path.

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



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

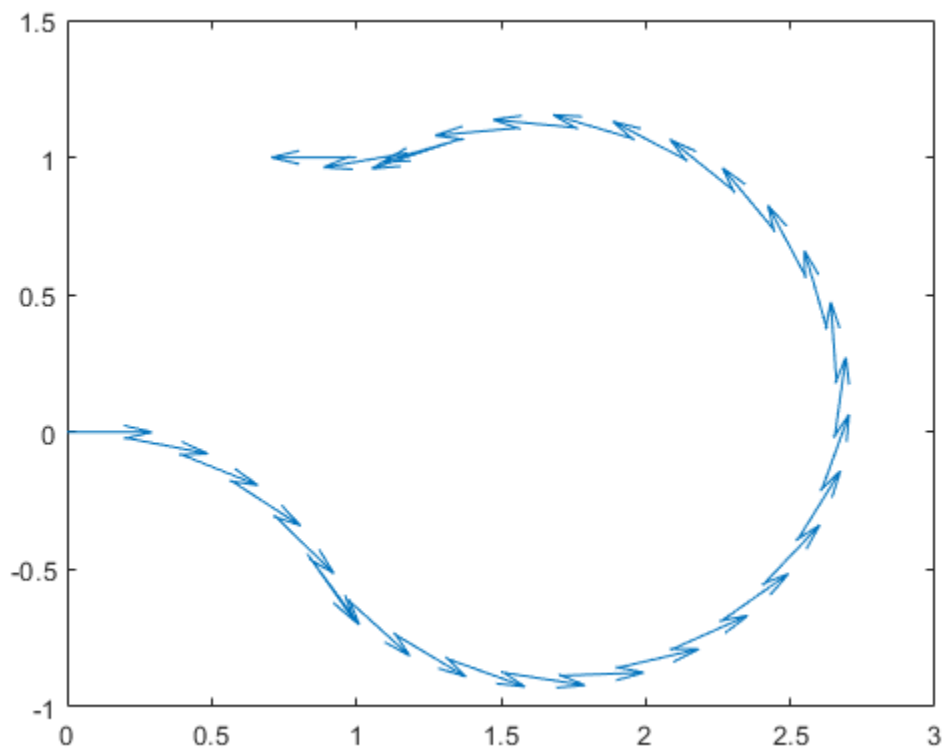
```
length = pathSegObj{1}.Length;  
poses = interpolate(pathSegObj{1}, [0:0.2:length])
```

```
poses = 32×3
```

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

Use the `quiver` function to plot these poses.

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



## Input Arguments

### **pathSeg** — Path segment

DubinsPathSegment object | ReedsSheppPathSegment object

Path segment, specified as a `robotics.DubinsPathSegment` or `robotics.ReedsSheppPathSegment` object.

### **lengths** — Lengths along path to interpolate at

positive numeric vector

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

## Output Arguments

### **poses** — Interpolated poses

`[x, y,  $\theta$ ]` matrix

This property is read-only.

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

`x` and `y` are in meters.  $\theta$  is in radians.

### **directions** — Directions of motion

vector of 1s (forward) and -1s (reverse)

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

## Extended Capabilities

### **C/C++ Code Generation**

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

## See Also

### **Functions**

`connect` | `show`

### **Objects**

`robotics.DubinsConnection` | `robotics.DubinsPathSegment` |  
`robotics.ReedsSheppConnection` | `robotics.ReedsSheppPathSegment`

**Introduced in R2018b**

# isNodeRunning

Determine if ROS node is running

## Syntax

```
running = isNodeRunning(device,modelName)
```

## Description

`running = isNodeRunning(device,modelName)` determines if the ROS node associated with the specified Simulink model is running on the specified `rosdevice`, `device`.

## Examples

### Run ROS Node on ROS Device

Connect to a remote ROS device and start a ROS node. First, run a ROS core so that ROS nodes can communicate via a ROS network. You can run and stop a ROS core or node and check their status using a `rosdevice` object.

Create a connection to a ROS device. Specify the address, user name and password of your specific ROS device. The device contains information about the ROS device, including the available ROS nodes that can be run using `runNode`.

```
ipaddress = '192.168.203.129';  
d = rosdevice(ipaddress,'user','password');  
d.ROSFolder = '/opt/ros/hydro';  
d.CatkinWorkspace = '~/catkin_ws_test'
```

```
d =
```

```
rosdevice with properties:
```

```

DeviceAddress: '192.168.203.129'
Username: 'user'
ROSFolder: '/opt/ros/hydro'
CatkinWorkspace: '~/catkin_ws_test'
AvailableNodes: {'robotcontroller' 'robotcontroller2'}

```

Run a ROS core. Connect MATLAB® to the ROS master using `rosinit`. This core enables you to run ROS nodes on your ROS device.

```

runCore(d)
rosinit(d.DeviceAddress,11311)

```

```

Initializing global node /matlab_global_node_12272 with NodeURI http://192.168.203.1:6

```

Check the available ROS nodes on the connected ROS device. These nodes were generated from Simulink® models following the process in the “Generate a Standalone ROS Node from Simulink®” example.

```

d.AvailableNodes

```

```

ans =

```

```

1x2 cell array

```

```

{'robotcontroller'} {'robotcontroller2'}

```

Run a ROS node. specifying the node name. Check if the node is running.

```

runNode(d, 'robotcontroller')
running = isNodeRunning(d, 'robotcontroller')

```

```

running =

```

```

logical

```

```

1

```

Stop the ROS node. Disconnect from the ROS network. Stop the ROS core.

```
stopNode(d, 'robotcontroller')  
roshutdown  
stopCore(d)
```

Shutting down global node /matlab\_global\_node\_12272 with NodeURI http://192.168.203.1:

## Input Arguments

### **device** — ROS device

rosdevice object

ROS device, specified as a rosdevice object.

### **modelName** — Name of the deployed Simulink model

character vector

Name of the deployed Simulink model, specified as a character vector. If the model name is not valid, the function returns `false`.

## Output Arguments

### **running** — Status of whether ROS node is running

`true` | `false`

Status of whether ROS node is running, returned as `true` or `false`.

## See Also

`rosdevice` | `runNode` | `stopNode`

## Topics

“Generate a Standalone ROS Node from Simulink®”

**Introduced in R2016b**



## ldivide, \

Element-wise quaternion left division

### Syntax

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

### Description

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

### Examples

#### Divide a Quaternion Array by a Real Scalar

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

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

```
A = 2x1 quaternion array
    1 + 2i + 3j + 4k
    5 + 6i + 7j + 8k
```

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

```
C = 2x1 quaternion array
    0.066667 - 0.13333i - 0.2j - 0.26667k
    0.057471 - 0.068966i - 0.08046j - 0.091954k
```

## Divide a Quaternion Array by Another Quaternion Array

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

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

```
A = 2x2 quaternion array  
    1 + 2i + 3j + 4k    4 + 5i + 6j + 7k  
    2 + 3i + 4j + 5k    5 + 6i + 7j + 8k
```

```
q2 = quaternion(magic(4));  
B = reshape(q2,2,2)
```

```
B = 2x2 quaternion array  
    16 + 2i + 3j + 13k    9 + 7i + 6j + 12k  
    5 + 11i + 10j + 8k    4 + 14i + 15j + 1k
```

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

```
C = 2x2 quaternion array  
    2.7 - 1.9i - 0.9j - 1.7k    1.5159 - 0.37302i - 0.15079j  
    2.2778 + 0.46296i - 0.57407j + 0.092593k    1.2471 + 0.91379i - 0.33908j
```

## 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

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

## 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 R2018b**

# log

Natural logarithm of quaternion array

## Syntax

$B = \log(A)$

## Description

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

## Examples

### Logarithmic Values of Quaternion Array

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

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

```
A = 3x1 quaternion array
    0.53767 + 0.86217i - 0.43359j + 2.7694k
    1.8339 + 0.31877i + 0.34262j - 1.3499k
   -2.2588 - 1.3077i + 3.5784j + 3.0349k
```

Compute the logarithmic values of  $A$ .

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

```
B = 3x1 quaternion array
    1.0925 + 0.40848i - 0.20543j + 1.3121k
    0.8436 + 0.14767i + 0.15872j - 0.62533k
    1.6807 - 0.53829i + 1.473j + 1.2493k
```

## Input Arguments

### **A — Input array**

scalar | vector | matrix | multidimensional array

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

Data Types: quaternion

## Output Arguments

### **B — Logarithm values**

scalar | vector | matrix | multidimensional array

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

Data Types: quaternion

## Algorithms

Given a quaternion  $A = a + \bar{v} = a + bi + cj + dk$ , 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 R2018b**

## matchScans

Estimate pose between two laser scans

### Syntax

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

### Description

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

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

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

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

### Examples

#### Match Lidar Scans

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

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



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

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

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

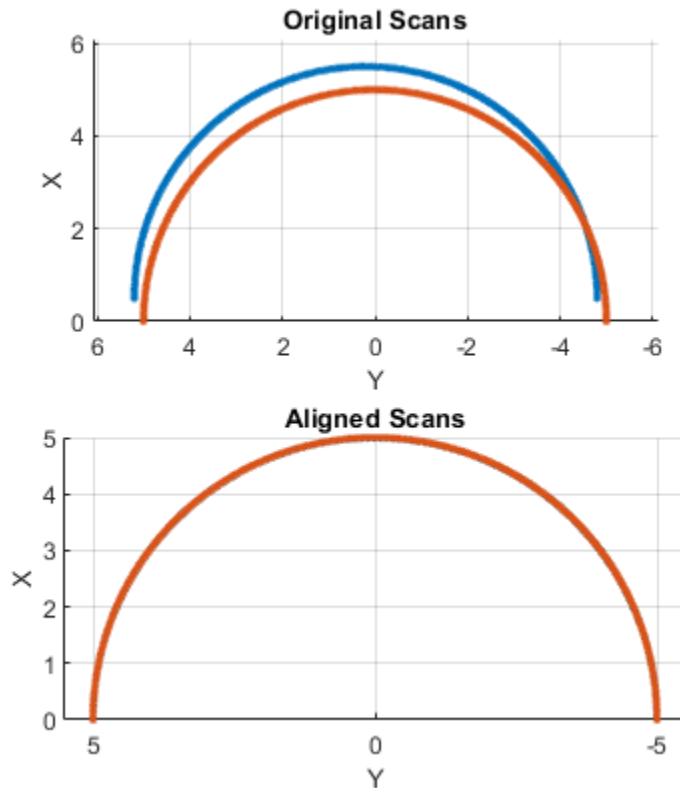
```
pose = matchScans(currScan, refScan);
```

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

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

```
subplot(2,1,1);  
hold on  
plot(currScan)  
plot(refScan)  
title('Original Scans')  
hold off
```

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



### Match Laser Scans

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

Specify a reference laser scan as ranges and angles.

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

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

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

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

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

Improve the estimate by giving an initial pose estimate.

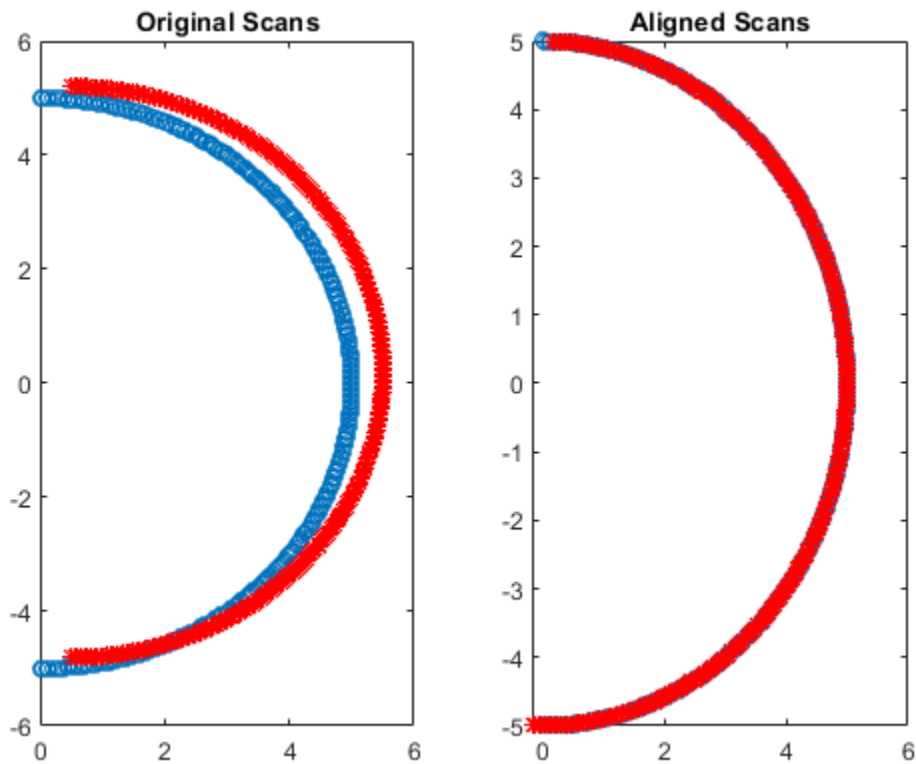
```
pose = matchScans(currRanges,currAngles,refRanges,refAngles,...
    'SolverAlgorithm','fminunc','InitialPose',[-0.4 -0.1 0]);
```

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

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

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

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



## Input Arguments

### **currScan** — Current lidar scan readings

lidarScan object

Current lidar scan readings, specified as a lidarScan object.

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

### **refScan** — Reference lidar scan readings

lidarScan object

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

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

### **currRanges — Current laser scan ranges**

vector in meters

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

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

### **currAngles — Current laser scan angles**

vector in radians

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

### **refRanges — Reference laser scan ranges**

vector in meters

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

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

### **refAngles — Reference laser scan angles**

vector in radians

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

## **Name-Value Pair Arguments**

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

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

### **SolverAlgorithm — Optimization algoerithm**

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

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

### **InitialPose — Initial guess of current pose**

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

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

### **CellSize — Length of cell side**

1 (default) | numeric scalar

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

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

### **MaxIterations — Maximum number of iterations**

400 (default) | scalar integer

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

### **ScoreTolerance — Lower bounds on the change in NDT score**

1e-6 (default) | numeric scalar

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

## Output Arguments

### **pose** — Pose of current scan

[x y theta]

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

### **stats** — Scan matching statistics

structure

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

- **Score** — Numeric scalar representing the NDT score while performing scan matching. This score is an estimate of the likelihood that the transformed current scan matches the reference scan. Score is always nonnegative. Larger scores indicate a better match.
- **Hessian** — 3-by-3 matrix representing the Hessian of the NDT cost function at the given pose solution. The Hessian is used as an indicator of the uncertainty associated with the pose estimate.

## References

- [1] Biber, P., and W. Strasser. "The Normal Distributions Transform: A New Approach to Laser Scan Matching." *Intelligent Robots and Systems Proceedings*. 2003.
- [2] Magnusson, Martin. "The Three-Dimensional Normal-Distributions Transform -- an Efficient Representation for Registration, Surface Analysis, and Loop Detection." PhD Dissertation. Örebro University, School of Science and Technology, 2009.

## Extended Capabilities

### **C/C++ Code Generation**

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

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

## See Also

### Functions

lidarScan | readCartesian | readScanAngles | transformScan

### Classes

MonteCarloLocalization | OccupancyGrid

### Topics

“Estimate Robot Pose with Scan Matching”

“Compose a Series of Laser Scans with Pose Changes”

### Introduced in R2017a



# matchScansGrid

Estimate pose between two lidar scans using grid-based search

## Syntax

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

## Description

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

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

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

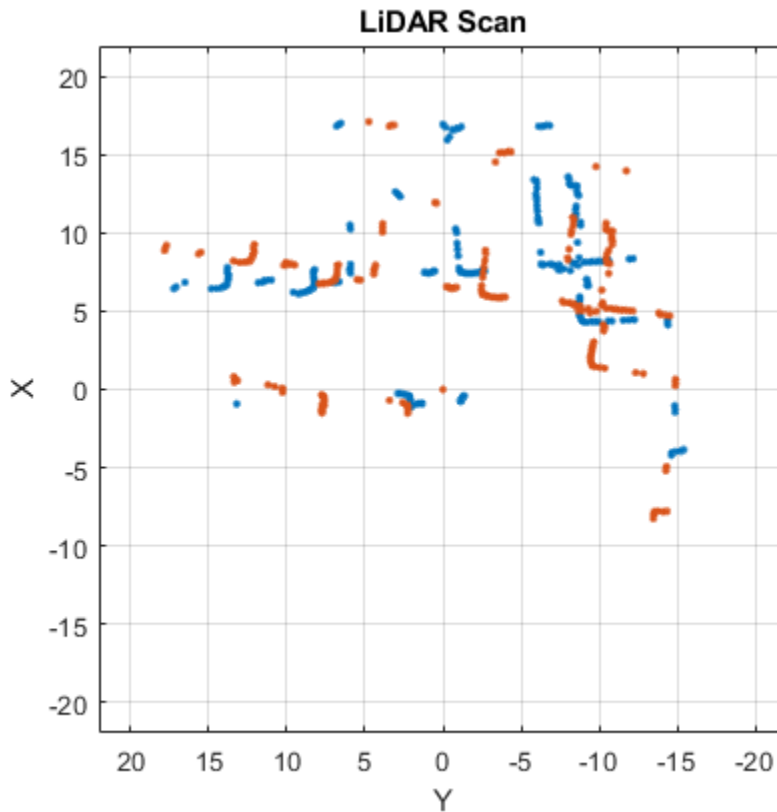
## Examples

### Match Scans Using Grid-Based Search

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

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

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



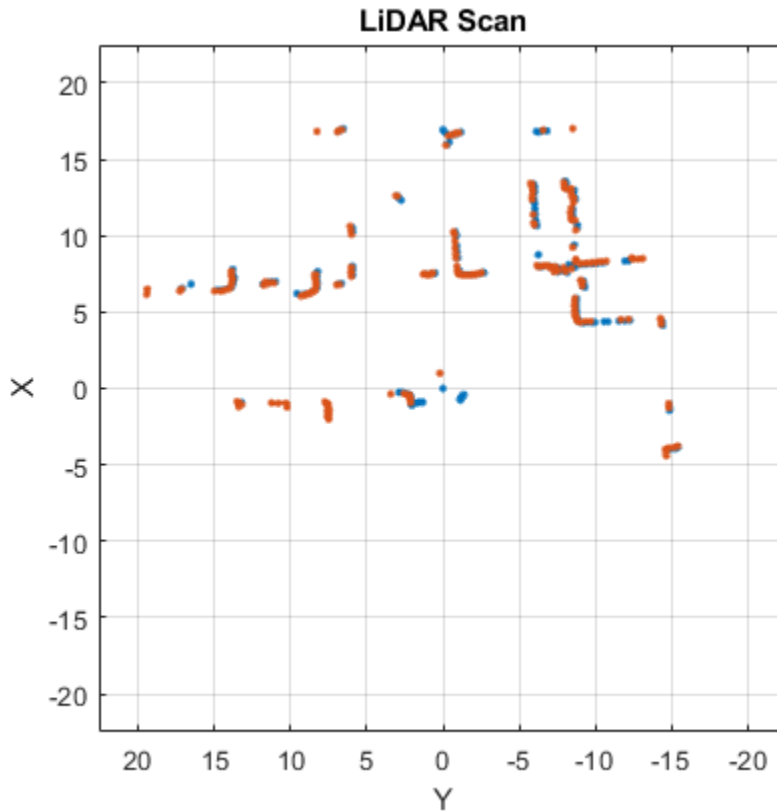
Use `matchScansGrid` to estimate the pose between the two scans.

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

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

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

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



## Input Arguments

**currScan** — Current lidar scan readings

lidarScan object

Current lidar scan readings, specified as a lidarScan object.

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

### **refScan — Reference lidar scan readings**

lidarScan object

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

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

## **Name-Value Pair Arguments**

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

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

### **InitialPose — Initial guess of current pose**

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

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

### **Resolution — Grid cells per meter**

20 (default) | positive integer

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

### **MaxRange — Maximum range of lidar sensor**

8 (default) | positive scalar

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

### **TranslationSearchRange — Search range for translation**

`[4 4]` (default) | `[x y]` vector

Search range for translation, specified as the comma-separated pair consisting of `'TranslationSearchRange'` and an `[x y]` vector. These values define the search

window in meters around the initial translation estimate given in `InitialPose`. If the `InitialPose` is given as  $[x_0 \ y_0]$ , then the search window coordinates are  $[x_0 - x \ x_0 + x]$  and  $[y_0 - y \ y_0 + y]$ . This parameter is used only when `InitialPose` is specified.

### **RotationSearchRange — Search range for rotation**

$\pi/4$  (default) | positive scalar

Search range for rotation, specified as the comma-separated pair consisting of 'RotationSearchRange' and a positive scalar. This value defines the search window in radians around the initial rotation estimate given in `InitialPose`. If the `InitialPose` rotation is given as  $\theta_0$ , then the search window is  $[\theta_0 - a \ \theta_0 + a]$ , where  $a$  is the rotation search range. This parameter is used only when `InitialPose` is specified.

### **MaxLevel — Maximum resolution level used for scan matching**

5 (default) | positive integer

Maximum resolution level used for scan matching, specified as the comma-separated pair consisting of 'MaxLevel' and a positive integer. A level below 6 is recommended. Decreasing the level speeds up performance, but can result in a coarser resolution of the matching solution.

## **Output Arguments**

### **pose — Pose of current scan**

$[x \ y \ \theta]$  vector

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

### **stats — Scan matching statistics**

structure

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

- **Score** — Numeric scalar representing the score while performing scan matching. This score is an estimate of the likelihood that the transformed current scan matches the reference scan. `Score` is always nonnegative. Larger scores indicate a better match, but values vary depending on the lidar data used.

### References

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

### Extended Capabilities

#### C/C++ Code Generation

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

### See Also

#### Functions

`lidarScan` | `matchScans` | `readCartesian` | `readScanAngles` | `transformScan`

#### Classes

`robotics.LidarSLAM`

#### Topics

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

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

#### Introduced in R2018a

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

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

```
eulerAverage =
    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');

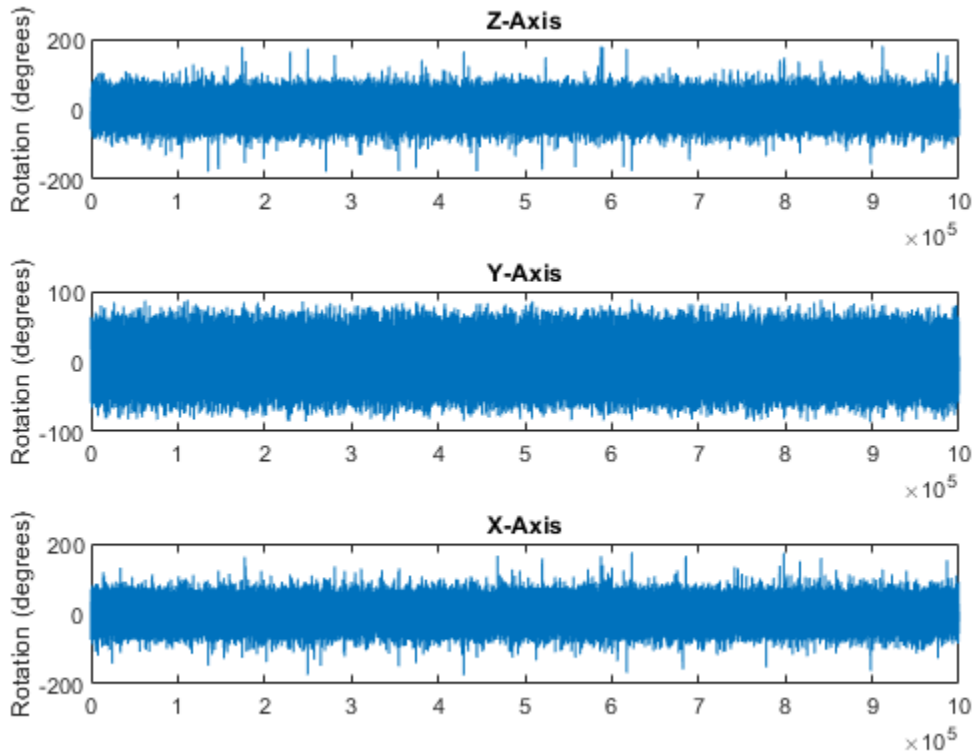
noisyEulerAngles = eulerd(q, 'ZYX', 'frame');

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.

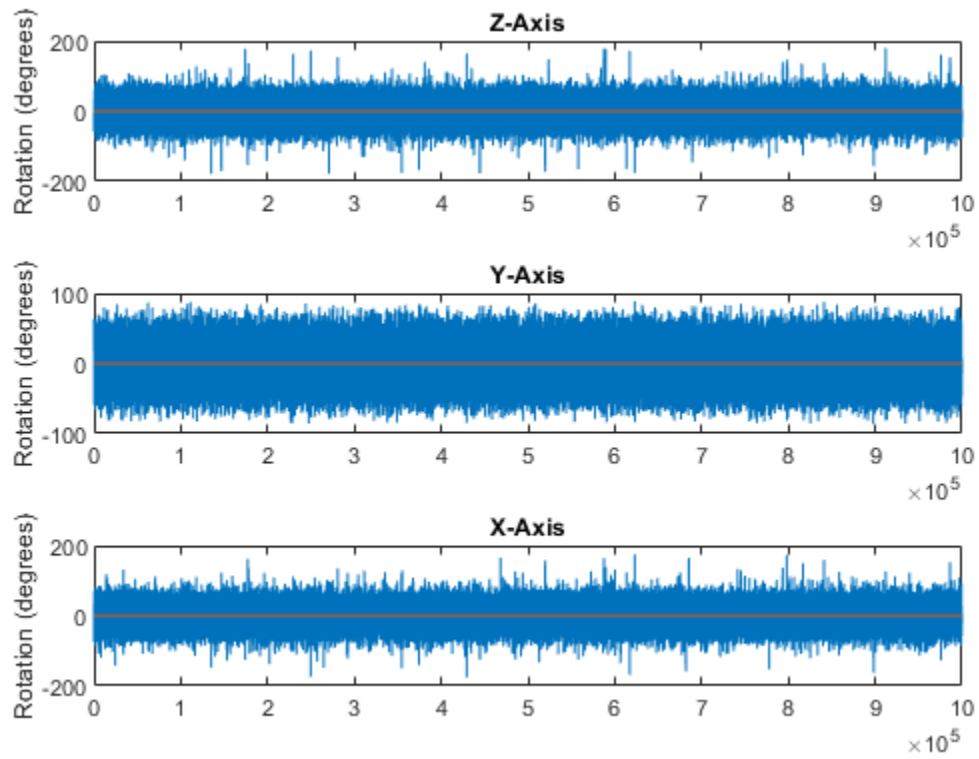
```

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');
```

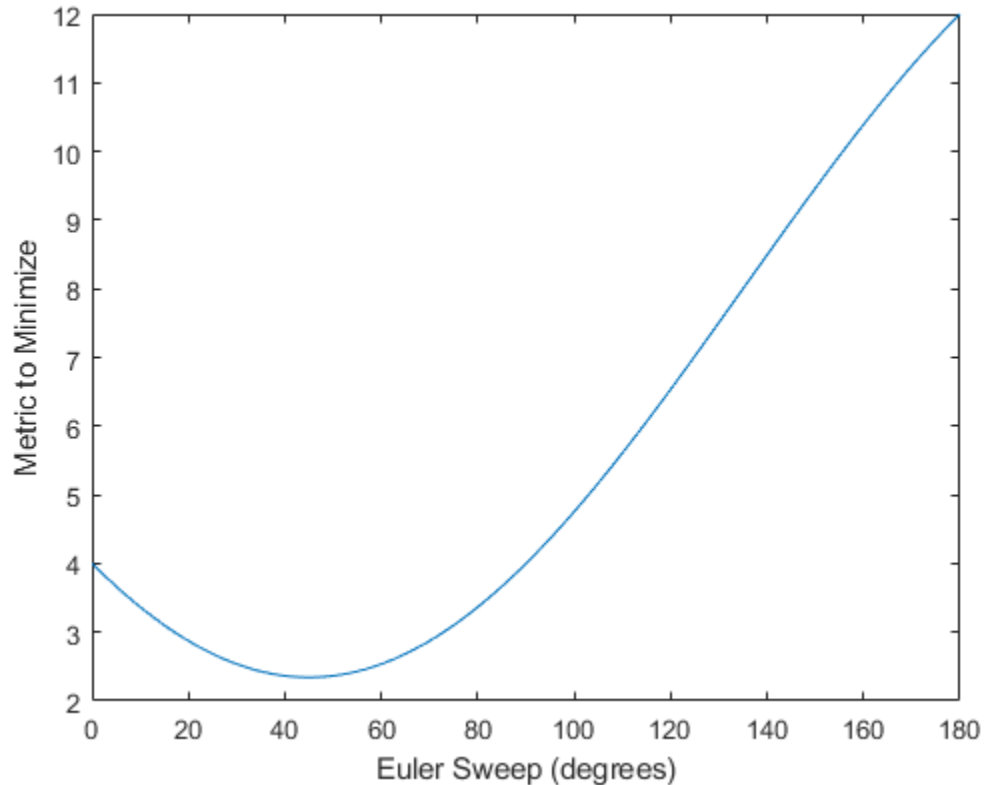
```
metricToMinimize = zeros(size(rSweep,3),1);  
for i = 1:numel(qSweep)  
    metricToMinimize(i) = norm((rSweep(:, :, i) - r0), 'fro').^2 + ...  
        norm((rSweep(:, :, i) - r90), 'fro').^2;  
end
```

```
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 =
```

```
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');

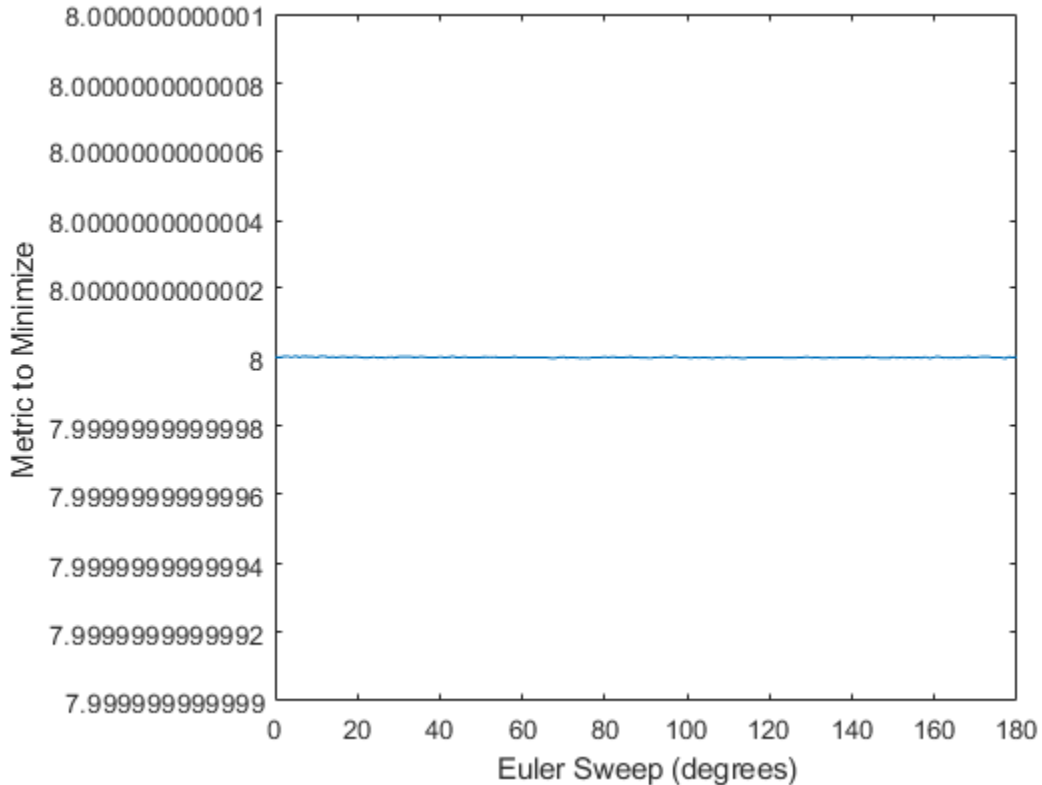
for i = 1:numel(qSweep)
    metricToMinimize(i) = norm((rSweep(:, :, i) - r0), 'fro').^2 + ...
        norm((rSweep(:, :, i) - r180), 'fro').^2;
end

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 =
```

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] (Sensor Fusion and Tracking Toolbox).  $\bar{q}$  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™.

Introduced in R2018b

# 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™.

**Introduced in R2018a**

## 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  $\beta$  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:

	<b>1</b>	<b>i</b>	<b>j</b>	<b>k</b>
<b>1</b>	1	i	j	k
<b>i</b>	i	-1	k	-j
<b>j</b>	j	-k	-1	i
<b>k</b>	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™.

**Introduced in R2018a**



# nodes

**Package:** robotics

Poses of nodes in pose graph

## Syntax

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

## Description

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

`nodes = nodes(poseGraph, nodeIDs)` lists the poses with the specified node IDs.

## Input Arguments

**poseGraph** — Pose graph

PoseGraph object | PoseGraph3D object

Pose graph, specified as a PoseGraph or PoseGraph3D object.

**nodeIDs** — Node IDs

positive integer | vector of positive integers

Node IDs, specified as a positive integer or vector of positive integers. Each node added gets an ID sequentially in the graph.

## Output Arguments

**nodes** — Pose of each node

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

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

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

For PoseGraph3D, each row is an `[x y z qw qx qy qz]` vector, which defines the relative  $xyz$ -position and quaternion orientation, `[qw qx qy qz]`, of a pose in the graph.

---

**Note** Many other sources for 3-D pose graphs, including `.g2o` formats, specify the quaternion orientation in a different order, for example, `[qx qy qz qw]`. Check the source of your pose graph data before adding nodes to your PoseGraph3D object.

---

## Extended Capabilities

### C/C++ Code Generation

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

Use this syntax when constructing PoseGraph or PoseGraph3D objects for code generation:

```
poseGraph =  
robotics.PoseGraph('MaxNumEdges',maxEdges,'MaxNumNodes',maxNodes)
```

specifies an upper bound on the number of edges and nodes allowed in the pose graph when generating code. This limit is only required when generating code.

## See Also

### Functions

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

### Objects

`robotics.LidarSLAM` | `robotics.PoseGraph` | `robotics.PoseGraph3D`

## **Topics**

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

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

**Introduced in R2018a**

## 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™.

**Introduced in R2018a**

## 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™.

**Introduced in R2018a**

# openShell

Open interactive command shell to device

## Syntax

```
openShell(device)
```

## Description

`openShell(device)` opens an SSH terminal on your host computer that provides encrypted access to the Linux<sup>®</sup> command shell on the ROS device. When prompted, enter a user name and password.

## Examples

### Open Command Shell on ROS Device

Connect to a ROS device and open the command shell on your host computer.

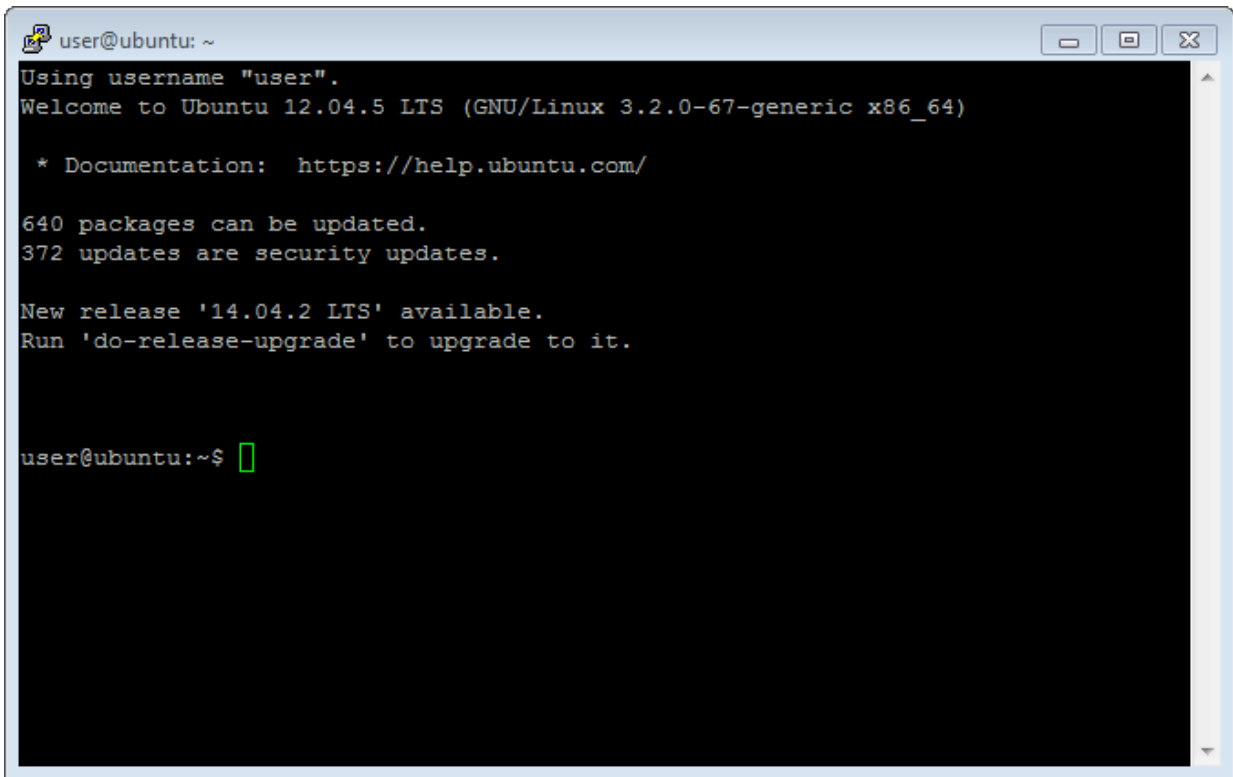
Connect to a ROS device. Specify the device address, user name, and password of your ROS device.

```
d = rosdevice('192.168.154.131', 'user', 'password');
```

Open the command shell.

```
openShell(d);
```



A terminal window titled 'user@ubuntu: ~' with standard window controls. The terminal output shows a login message for 'user', a welcome message for Ubuntu 12.04.5 LTS, and system update information. It states that 640 packages can be updated, with 372 security updates. A new release '14.04.2 LTS' is available, and the user is prompted to run 'do-release-upgrade'. The prompt 'user@ubuntu:~\$' is followed by a green cursor.

```
user@ubuntu: ~
Using username "user".
Welcome to Ubuntu 12.04.5 LTS (GNU/Linux 3.2.0-67-generic x86_64)

 * Documentation:  https://help.ubuntu.com/

640 packages can be updated.
372 updates are security updates.

New release '14.04.2 LTS' available.
Run 'do-release-upgrade' to upgrade to it.

user@ubuntu:~$ █
```

## Input Arguments

**device** — ROS device

rosdevice object

ROS device, specified as a rosdevice object.

## See Also

deleteFile | dir | getFile | putFile | rosdevice | system

**Introduced in R2016b**

## 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`.

```
quat0nes = ones(2, 'like', single(1), 'quaternion')
```

```
quat0nes = 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(quat0nes)
```

```
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™.

**Introduced in R2018a**

# optimizePoseGraph

Optimize nodes in pose graph

## Syntax

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

## Description

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

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

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

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

## Examples

### Optimize a 3-D Pose Graph

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

Load the pose graph from the MIT dataset. Inspect the `robotics.PoseGraph3D` object to view the number of nodes and loop closures.

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

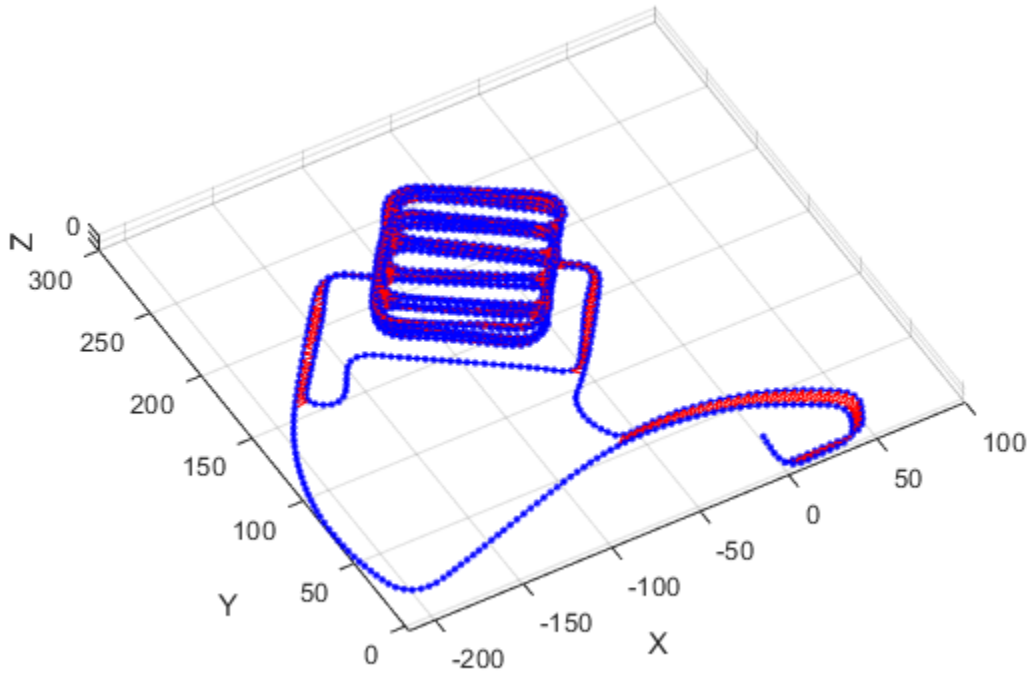
```
PoseGraph3D with properties:
```

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

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

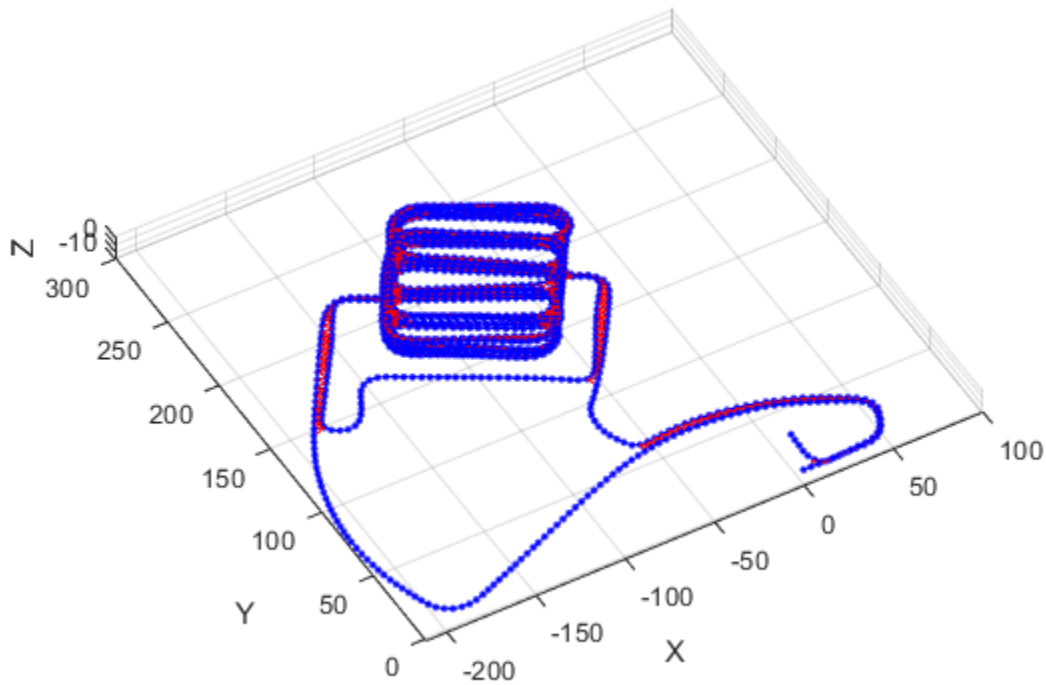
```
title('Original Pose Graph')
show(pg, 'IDs', 'off');
view(-30,45)
```





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

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



## Input Arguments

**poseGraph** — 2-D or 3-D pose graph

PoseGraph object | PoseGraph3D object

2-D or 3-D pose graph, specified as a PoseGraph or PoseGraph3D object.

**solver** — Pose graph solver

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

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

## Name-Value Pair Arguments

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

Example: `'MaxTime', 300`

---

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

---

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

### **MaxTime** — Maximum time allowed

500 (default) | positive numeric scalar

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

### **GradientTolerance** — Lower bound on norm of gradient

0.5e-8 (default) | scalar

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

### **FunctionTolerance** — Lower bound on change in cost function

1e-8 (default) | scalar

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

### **StepTolerance** — Lower bound on step size

1e-12 (default) | scalar

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

### **InitialTrustRegionRadius — Initial trust region radius**

100 (default) | scalar

Initial trust region radius, specified as a scalar.

### **VerboseOutput — Display intermediate iteration information**

'off' (default) | 'on'

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

### **LoopClosuresToIgnore — IDs of loop closure edges in pose graph**

vector

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

### **FirstNodePose — Pose of first node**

[0 0 0] or [0 0 0 1 0 0 0] (default) | [x y theta] | [x y z qw qx qy qz]

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

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

For `PoseGraph3D`, the pose is an [x y z qw qx qy qz] vector, which defines the relative xyz-position and quaternion orientation, [qw qx qy qz].

---

**Note** Many other sources for 3-D pose graphs, including .g2o formats, specify the quaternion orientation in a different order, for example, [qx qy qz qw]. Check the source of your pose graph data before adding nodes to your `PoseGraph3D` object.

---

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

### **MaxIterations — Maximum number of iterations**

300 (default) | positive integer

Maximum number of iterations, specified as the comma-separated pair consisting of 'MaxIterations' and a positive integer. The optimizer exits after it exceeds this number of iterations.

**MaxTime — Maximum time allowed**

500 (default) | positive numeric scalar

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

**FunctionTolerance — Lower bound on change in cost function**

1e-8 (default) | scalar

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

**VerboseOutput — Display intermediate iteration information**

'off' (default) | 'on'

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

**LoopClosuresToIgnore — IDs of loop closure edges in pose graph**

vector

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

**FirstNodePose — Pose of first node**

[0 0 0] or [0 0 0 1 0 0 0] (default) | [x y theta] | [x y z qw qx qy qz]

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

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

For PoseGraph3D, the pose is an [x y z qw qx qy qz] vector, which defines the relative xyz-position and quaternion orientation, [qw qx qy qz].

---

**Note** Many other sources for 3-D pose graphs, including .g2o formats, specify the quaternion orientation in a different order, for example, [qx qy qz qw]. Check the source of your pose graph data before adding nodes to your PoseGraph3D object.

---

## Output Arguments

### **updatedGraph — Optimized 2-D or 3-D pose graph**

PoseGraph object | PoseGraph3D object

Optimized 2-D or 3-D pose graph, returned as a PoseGraph or PoseGraph3D object.

### **solutionInfo — Statistics of optimization process**

structure

Statistics of optimization process, returned as a structure with these fields:

- **Iterations** — Number of iterations used in optimization.
- **ResidualError** — Value of cost function when optimizer exits.
- **Exit Flag** — Exit condition for optimizer:
  - 1 — Local minimum found.
  - 2 — Maximum number of iterations reached. See `MaxIterations` name-value pair argument.
  - 3 — Algorithm timed out during operation.
  - 4 — Minimum step size. The step size is below the `StepTolerance` name-value pair argument.
  - 5 — The change in error is below the minimum.
  - 8 — Trust region radius is below the minimum set in `InitialTrustRegionRadius`.

## References

- [1] Grisetti, G., R. Kummerle, C. Stachniss, and W. Burgard. "A Tutorial on Graph-Based SLAM." *IEEE Intelligent Transportation Systems Magazine*. Vol. 2, No. 4, 2010, pp. 31-43. doi:10.1109/mits.2010.939925.
- [2] Carlone, Luca, Roberto Tron, Kostas Daniilidis, and Frank Dellaert. "Initialization Techniques for 3D SLAM: a Survey on Rotation Estimation and its Use in Pose

Graph Optimization." *2015 IEEE International Conference on Robotics and Automation (ICRA)*. 2015, pp. 4597–4604.

## Extended Capabilities

### C/C++ Code Generation

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

- Use this syntax when constructing PoseGraph or PoseGraph3D objects for code generation:

```
poseGraph =  
robotics.PoseGraph( 'MaxNumEdges', maxEdges, 'MaxNumNodes', maxNodes )
```

specifies an upper bound on the number of edges and nodes allowed in the pose graph when generating code. This limit is only required when generating code.

- The "g2o-levenberg-marquardt" solver input argument is not supported for code generation.

## See Also

### Functions

`addRelativePose` | `edgeConstraints` | `edges` | `findEdgeID` | `nodes` | `removeEdges`

### Objects

`robotics.LidarSLAM` | `robotics.PoseGraph` | `robotics.PoseGraph3D`

### Topics

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

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

**Introduced in R2018a**

## parts

Extract quaternion parts

### Syntax

```
[a,b,c,d] = parts(quat)
```

### Description

`[a,b,c,d] = parts(quat)` returns the parts of the quaternion array as arrays, each the same size as `quat`.

### Examples

#### Convert Quaternion to Matrix of Quaternion Parts

Convert a quaternion representation to parts using the `parts` function.

Create a two-element column vector of quaternions by specifying the parts.

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

```
quat = 2x1 quaternion array
    1 + 2i + 3j + 4k
    5 + 6i + 7j + 8k
```

Recover the parts from the quaternion matrix using the `parts` function. The parts are returned as separate output arguments, each the same size as the input 2-by-1 column vector of quaternions.

```
[qA,qB,qC,qD] = parts(quat)
```

```
qA = 2x1
```



---

1  
5

qB = 2×1

2  
6

qC = 2×1

3  
7

qD = 2×1

4  
8

## Input Arguments

### **quat** — Quaternion

scalar | vector | matrix | multidimensional array

Quaternion, specified as a quaternion or array of quaternions.

Data Types: quaternion

## Output Arguments

### **[a, b, c, d]** — Quaternion parts

scalar | vector | matrix | multidimensional array

Quaternion parts, returned as four arrays: a, b, c, and d. Each part is the same size as quat.

Data Types: single | double

## **Extended Capabilities**

### **C/C++ Code Generation**

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

**Introduced in R2018a**

## plot

Display laser or lidar scan readings

### Syntax

```
plot(scanMsg)
plot(scanObj)
plot( ____,Name,Value)
linehandle = plot( ____ )
```

### Description

`plot(scanMsg)` plots the laser scan readings specified in the input `LaserScan` object message. Axes are automatically scaled to the maximum range that the laser scanner supports.

`plot(scanObj)` plots the lidar scan readings specified in `scanObj`.

`plot( ____,Name,Value)` provides additional options specified by one or more `Name,Value` pair arguments.

`linehandle = plot( ____ )` returns a column vector of line series handles, using any of the arguments from previous syntaxes. Use `linehandle` to modify properties of the line series after it is created.

When plotting ROS laser scan messages, MATLAB follows the standard ROS convention for axis orientation. This convention states that **positive x is forward, positive y is left, and positive z is up**. For more information, see [Axis Orientation on the ROS Wiki](#).

## Examples

### Plot Laser Scan Message

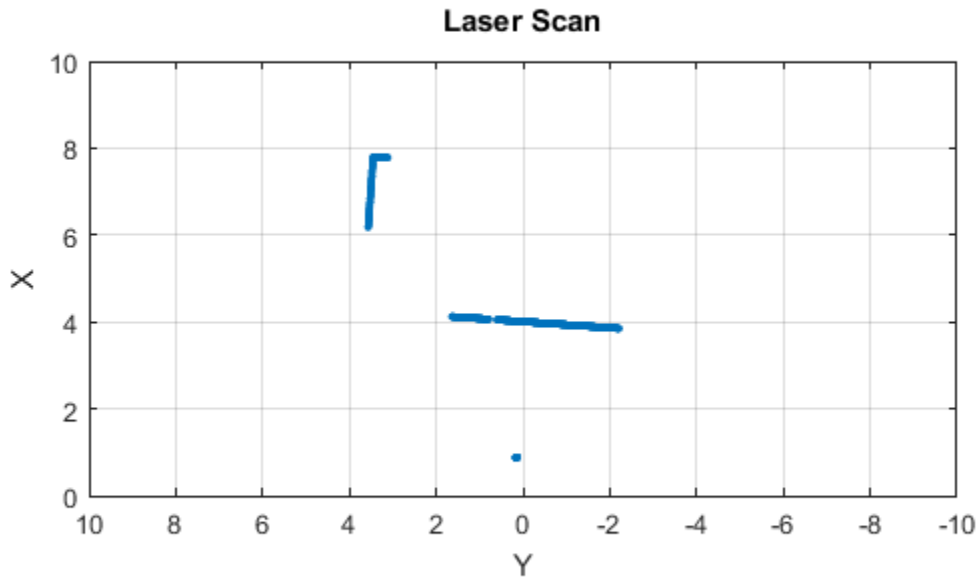
Connect to ROS network. Subscribe to a laser scan topic, and receive a message.

```
rosinit('192.168.154.131')
sub = rossubscriber('/scan');
scan = receive(sub);
```

Initializing global node /matlab\_global\_node\_06485 with NodeURI http://192.168.154.1:6

Plot the laser scan.

```
plot(scan)
```



Shutdown ROS network.

```
roshutdown
```

Shutting down global node /matlab\_global\_node\_06485 with NodeURI http://192.168.154.1:6

### **Plot Laser Scan Message With Maximum Range**

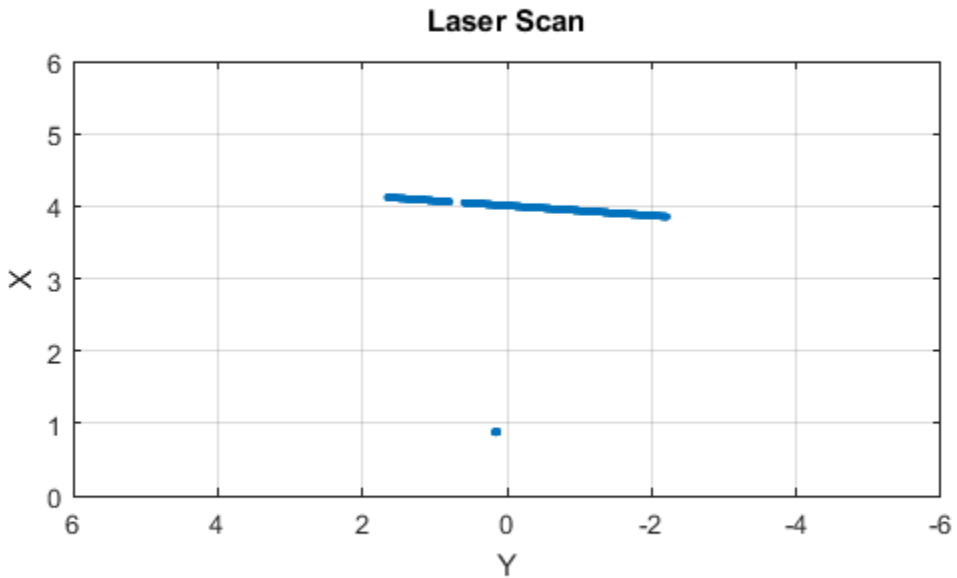
Connect to ROS network. Subscribe to a laser scan topic, and receive a message.

```
rosinit('192.168.154.131')  
sub = rossubscriber('/scan');  
scan = receive(sub);
```

Initializing global node /matlab\_global\_node\_29862 with NodeURI http://192.168.154.1:6

Plot the laser scan specifying the maximum range.

```
plot(scan, 'MaximumRange', 6)
```



Shutdown ROS network.

```
roshutdown
```

```
Shutting down global node /matlab_global_node_29862 with NodeURI http://192.168.154.1:
```

### Plot Lidar Scan and Remove Invalid Points

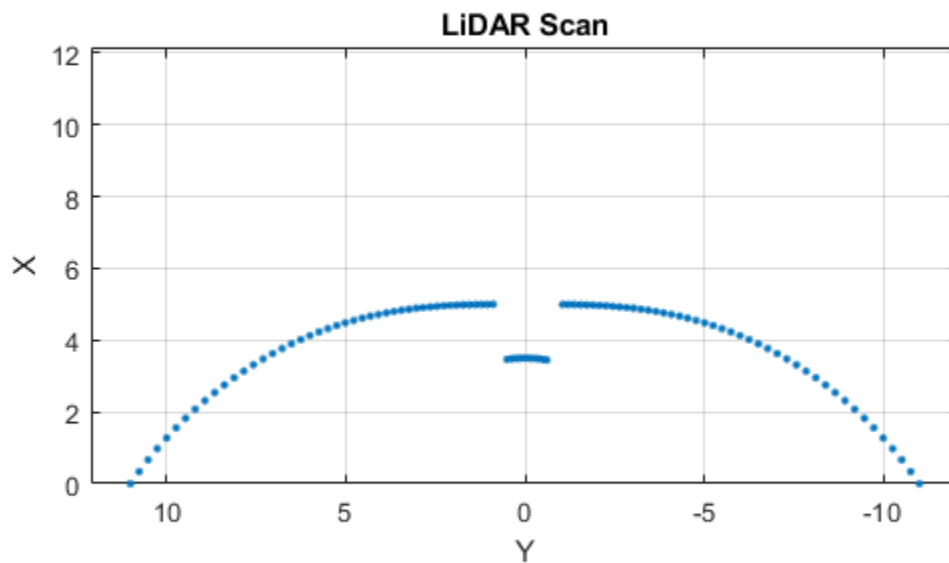
Specify lidar data as vectors of ranges and angles. These values include readings outside of the sensors range.

```
x = linspace(-2,2);  
ranges = abs((1.5).*x.^2 + 5);
```

```
ranges(45:55) = 3.5;  
angles = linspace(-pi/2,pi/2,numel(ranges));
```

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

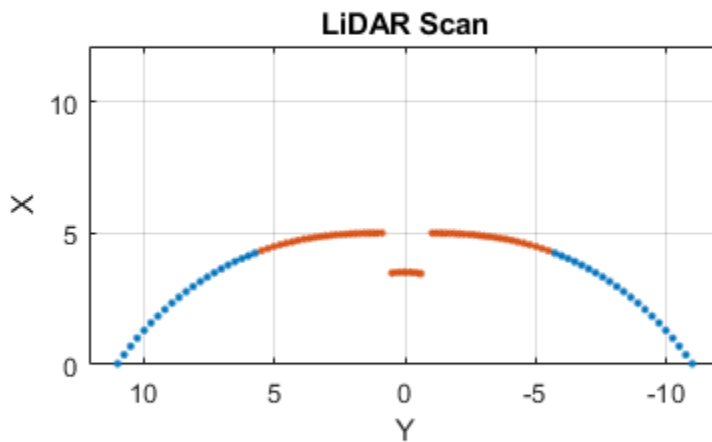
```
scan = lidarScan(ranges,angles);  
plot(scan)
```



Remove invalid points based on a specified minimum and maximum range.

```
minRange = 0.1;  
maxRange = 7;  
scan2 = removeInvalidData(scan, 'RangeLimits', [minRange maxRange]);  
hold on
```

```
plot(scan2)
legend('All Points', 'Valid Points')
```



## Input Arguments

**scanMsg** — Laser scan message

LaserScan object handle

sensor\_msgs/LaserScan ROS message, specified as a LaserScan object handle.

**scanObj** — Lidar scan readings

lidarScan object



Lidar scan readings, specified as a `LidarScan` object.

## Name-Value Pair Arguments

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

Example: `"MaximumRange", 5`

### Parent — Parent of axes

axes object

Parent of axes, specified as the comma-separated pair consisting of `"Parent"` and an axes object in which the laser scan is drawn. By default, the laser scan is plotted in the currently active axes.

### MaximumRange — Range of laser scan

`scan.RangeMax` (default) | scalar

Range of laser scan, specified as the comma-separated pair consisting of `"MaximumRange"` and a scalar. When you specify this name-value pair argument, the minimum and maximum x-axis and the maximum y-axis limits are set based on specified value. The minimum y-axis limit is automatically determined by the opening angle of the laser scanner.

This name-value pair only works when you input `scanMsg` as the laser scan.

## Outputs

### linehandle — One or more chart line objects

scalar | vector

One or more chart line objects, returned as a scalar or a vector. These are unique identifiers, which you can use to query and modify properties of a specific chart line.

## See Also

`readCartesian`

**Introduced in R2015a**

# 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.

Example: `'fixedwing.stl'` or `'multirotor.stl'`

#### **MeshColor** — Color of attached mesh

"red" (default) | RGB triplet | string scalar

Color of attached mesh, specified as an RGB triple or string scalar.

Example: `[0 0 1]` or `"green"`

#### **Parent** — Axes used to plot transforms

Axes object | UIAxes object

Axes used to plot the pose graph, specified as the comma-separated pair consisting of 'Parent' and either an Axes or UIAxes object. See `axes` or `uiaxes`.

## Output Arguments

### **ax — Axes used to plot transforms**

Axes object | UIAxes object

Axes used to plot the pose graph, specified as the comma-separated pair consisting of 'Parent' and either an Axes or UIAxes object. See `axes` or `uiaxes`.

## See Also

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

## Topics

"Axis-Angle"

**Introduced in R2018b**

## **power, .^**

Element-wise quaternion power

### **Syntax**

```
C = A.^b
```

### **Description**

`C = A.^b` raises each element of `A` to the corresponding power in `b`.

### **Examples**

#### **Raise a Quaternion to a Real Scalar Power**

Create a quaternion and raise it to a real scalar power.

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

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

```
b = 3;  
C = A.^b
```

```
C = quaternion  
   -86 - 52i - 78j - 104k
```

#### **Raise a Quaternion Array to Powers from a Multidimensional Array**

Create a 2-by-1 quaternion array and raise it to powers from a 2-D array.

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

```
A = 2x1 quaternion array
    1 + 2i + 3j + 4k
    5 + 6i + 7j + 8k
```

```
b = [1 0 2; 3 2 1]
```

```
b = 2x3
```

```
    1    0    2
    3    2    1
```

```
C = A.^b
```

```
C = 2x3 quaternion array
    1 + 2i + 3j + 4k    1 + 0i + 0j + 0k    -28 + 4i +
-2110 - 444i - 518j - 592k    -124 + 60i + 70j + 80k    5 + 6i +
```

## 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  $\theta$  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 R2018b**



## prod

Product of a quaternion array

## Syntax

```
quatProd = prod(quat)
quatProd = prod(quat,dim)
```

## Description

`quatProd = prod(quat)` returns the quaternion product of the elements of the array.

`quatProd = prod(quat,dim)` calculates the quaternion product along dimension `dim`.

## Examples

### Product of Quaternions in Each Column

Create a 3-by-3 array whose elements correspond to their linear indices.

```
A = reshape(quaternion(randn(9,4)),3,3)
```

*A = 3x3 quaternion array*

```
0.53767 + 2.7694i + 1.409j - 0.30344k    0.86217 + 0.7254i - 1.2075j
1.8339 - 1.3499i + 1.4172j + 0.29387k    0.31877 - 0.063055i + 0.71724j
-2.2588 + 3.0349i + 0.6715j - 0.78728k    -1.3077 + 0.71474i + 1.6302j
```

Find the product of the quaternions in each column. The length of the first dimension is 1, and the length of the second dimension matches `size(A,2)`.

```
B = prod(A)
```

*B = 1x3 quaternion array*

```
-19.837 - 9.1521i + 15.813j - 19.918k    -5.4708 - 0.28535i + 3.077j - 1.2
```

### 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 - 4.2  
    0.38986 - 3.6606i - 2.0474j - 6.047k    -1.741 - 0.26782i + 5.4346j + 4.1
```

## 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™.

**Introduced in R2018a**

# putFile

Copy file to device

## Syntax

```
putFile(device,localSource)
putFile(device,localSource,remoteDestination)
```

## Description

`putFile(device,localSource)` copies the specified source file from the MATLAB current folder to the print working directory (`pwd`) on the ROS device. Wildcards are supported.

`putFile(device,localSource,remoteDestination)` copies the file to a destination path. Specify a file name at the end of the destination path to copy with a custom file name.

## Examples

### Put, Get, and Delete Files on ROS Device

Put a file from your host computer onto a ROS device, get it back, and then delete it.

Connect to a ROS device. Specify the device address, user name, and password of your ROS device.

```
d = rosdevice('192.168.154.131','user','password');
```

Put a new text file that is in the MATLAB® current folder onto the ROS device. The destination folder must exist.

```
putFile(d,'test_file.txt','/home/user/test_folder')
```

Get a text file from the ROS device. You can get any file, not just ones added from MATLAB®. By default, the file is added to the MATLAB current folder.

```
getFile(d, '/home/user/test_folder/test_file.txt')
```

Delete the text file on the ROS device.

```
deleteFile(d, '/home/user/test_folder/test_file.txt')
```

### Put, Get, and Delete Files on ROS Device Using Wildcards

Put a file from your host computer onto a ROS device, get it back, and then delete it. Use wildcards to search for all matching files.

**Note:** You must have a valid ROS device to connect to at the IP address specified in the example.

Connect to a ROS device. Specify the device address, user name, and password of your ROS device.

```
d = rosdevice('192.168.203.129', 'user', 'password');
```

Put all text files at the specified path onto the ROS device. The destination folder must exist.

```
putFile(d, 'C:/MATLAB/*.txt', '/home/user/test_folder')
```

Get all text files from the ROS device. You can get any files, not just ones added from MATLAB®. By default, the files are added to the MATLAB current folder.

```
getFile(d, '/home/user/test_folder/*.txt')
```

Delete all text files on the ROS device at the specified folder.

```
deleteFile(d, '/home/user/test_folder/*.txt')
```

## Input Arguments

**device** — ROS device

rosdevice object

ROS device, specified as a `rosdevice` object.

### **localSource** — Path and name of file on host computer

character vector

Path and name of the file on the host computer, specified as a character vector. You can use an absolute path or a path relative from the MATLAB current folder. Use the path and file naming conventions of the operating system on your host computer.

Example: `'C:\Work\.profile'`

Data Types: `char`

### **remoteDestination** — Destination folder path and optional file name

character vector

Destination folder path and optional file name, specified as a character vector. Specify a file name at the end of the destination path to copy with a custom file name. Use the Linux path and file naming conventions.

Example: `'/home/user/.profile'`

Data Types: `char`

## **See Also**

`deleteFile` | `dir` | `getFile` | `openShell` | `rosdevice` | `system`

**Introduced in R2016b**

# quat2axang

Convert quaternion to axis-angle rotation

## Syntax

```
axang = quat2axang(quat)
```

## Description

`axang = quat2axang(quat)` converts a quaternion, `quat`, to the equivalent axis-angle rotation, `axang`.

## Examples

### Convert Quaternion to Axis-Angle Rotation

```
quat = [0.7071 0.7071 0 0];  
axang = quat2axang(quat)  
  
axang = 1×4  
    1.0000    0    0    1.5708
```

## Input Arguments

### **quat** — Unit quaternion

*n*-by-4 matrix | *n*-element vector of quaternion objects

Unit quaternion, specified as an *n*-by-4 matrix or *n*-element vector of quaternion objects containing *n* quaternions. If the input is a matrix, each row is a quaternion vector of the form  $q = [w \ x \ y \ z]$ , with *w* as the scalar number.

Example: `[0.7071 0.7071 0 0]`

## Output Arguments

### **axang** — Rotation given in axis-angle form

*n*-by-4 matrix

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

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

## Extended Capabilities

### **C/C++ Code Generation**

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

### **See Also**

`axang2quat` | `quaternion`

### **Topics**

“Coordinate Transformations in Robotics”

**Introduced in R2015a**



# quat2eul

Convert quaternion to Euler angles

## Syntax

```
eul = quat2eul(quat)
eul = quat2eul(quat,sequence)
```

## Description

`eul = quat2eul(quat)` converts a quaternion rotation, `quat`, to the corresponding Euler angles, `eul`. The default order for Euler angle rotations is "ZYX".

`eul = quat2eul(quat,sequence)` converts a quaternion into Euler angles. The Euler angles are specified in the axis rotation sequence, `sequence`. The default order for Euler angle rotations is "ZYX".

## Examples

### Convert Quaternion to Euler Angles

```
quat = [0.7071 0.7071 0 0];
eulZYX = quat2eul(quat)
```

```
eulZYX = 1×3
         0         0    1.5708
```

### Convert Quaternion to Euler Angles Using ZYZ Axis Order

```
quat = [0.7071 0.7071 0 0];  
eulZYZ = quat2eul(quat, 'ZYZ')  
  
eulZYZ = 1×3  
  
    1.5708    -1.5708    -1.5708
```

## Input Arguments

### quat — Unit quaternion

*n*-by-4 matrix | *n*-element vector of quaternion objects

Unit quaternion, specified as an *n*-by-4 matrix or *n*-element vector of quaternion objects containing *n* quaternions. If the input is a matrix, each row is a quaternion vector of the form  $q = [w \ x \ y \ z]$ , with *w* as the scalar number.

Example: [0.7071 0.7071 0 0]

### sequence — Axis rotation sequence

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

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

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

Data Types: string | char

## Output Arguments

### eul — Euler rotation angles

*n*-by-3 matrix

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

Example: [0 0 1.5708]

## Extended Capabilities

### C/C++ Code Generation

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

### See Also

eul2quat | quaternion

### Topics

“Coordinate Transformations in Robotics”

**Introduced in R2015a**

# quat2rotm

Convert quaternion to rotation matrix

## Syntax

```
rotm = quat2rotm(quat)
```

## Description

`rotm = quat2rotm(quat)` converts a quaternion `quat` to an orthonormal rotation matrix, `rotm`. When using the rotation matrix, premultiply it with the coordinates to be rotated (as opposed to postmultiplying).

## Examples

### Convert Quaternion to Rotation Matrix

```
quat = [0.7071 0.7071 0 0];  
rotm = quat2rotm(quat)
```

```
rotm = 3×3
```

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

## Input Arguments

### **quat** — Unit quaternion

*n*-by-4 matrix | *n*-element vector of quaternion objects

Unit quaternion, specified as an  $n$ -by-4 matrix or  $n$ -element vector of quaternion objects containing  $n$  quaternions. If the input is a matrix, each row is a quaternion vector of the form  $q = [w \ x \ y \ z]$ , with  $w$  as the scalar number.

Example: `[0.7071 0.7071 0 0]`

## Output Arguments

### **rotm** — Rotation matrix

3-by-3-by- $n$  matrix

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

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

## Extended Capabilities

### C/C++ Code Generation

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

### See Also

`quaternion` | `rotm2quat`

### Topics

“Coordinate Transformations in Robotics”

**Introduced in R2015a**

# quat2tform

Convert quaternion to homogeneous transformation

## Syntax

```
tform = quat2tform(quat)
```

## Description

`tform = quat2tform(quat)` converts a quaternion, `quat`, to a homogeneous transformation matrix, `tform`. When using the transformation matrix, premultiply it with the coordinates to be transformed (as opposed to postmultiplying).

## Examples

### Convert Quaternion to Homogeneous Transformation

```
quat = [0.7071 0.7071 0 0];  
tform = quat2tform(quat)
```

```
tform = 4×4
```

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

## Input Arguments

### **quat** — Unit quaternion

*n*-by-4 matrix | *n*-element vector of quaternion objects

Unit quaternion, specified as an  $n$ -by-4 matrix or  $n$ -element vector of quaternion objects containing  $n$  quaternions. If the input is a matrix, each row is a quaternion vector of the form  $q = [w \ x \ y \ z]$ , with  $w$  as the scalar number.

Example: `[0.7071 0.7071 0 0]`

## Output Arguments

### **tform** — Homogeneous transformation

4-by-4-by- $n$  matrix

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

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

## Extended Capabilities

### C/C++ Code Generation

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

### See Also

`quaternion` | `tform2quat`

### Topics

“Coordinate Transformations in Robotics”

**Introduced in R2015a**

## **rdivide, ./**

Element-wise quaternion right division

### **Syntax**

`C = A./B`

### **Description**

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

### **Examples**

#### **Divide a Quaternion Array by a Real Scalar**

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

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

```
A = 2x1 quaternion array
    1 + 2i + 3j + 4k
    5 + 6i + 7j + 8k
```

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

```
C = 2x1 quaternion array
    0.5 +    1i + 1.5j +    2k
    2.5 +    3i + 3.5j +    4k
```



## Divide a Quaternion Array by Another Quaternion Array

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

```
q1 = quaternion(magic(4));
A = reshape(q1,2,2)
```

```
A = 2x2 quaternion array
    16 + 2i + 3j + 13k    9 + 7i + 6j + 12k
    5 + 11i + 10j + 8k   4 + 14i + 15j + 1k
```

```
q2 = quaternion([1:4;3:6;2:5;4:7]);
B = reshape(q2,2,2)
```

```
B = 2x2 quaternion array
    1 + 2i + 3j + 4k    2 + 3i + 4j + 5k
    3 + 4i + 5j + 6k    4 + 5i + 6j + 7k
```

```
C = A./B
```

```
C = 2x2 quaternion array
    2.7 - 0.1i - 2.1j - 1.7k    2.2778 + 0.092593i - 0.46296j
    1.8256 - 0.081395i + 0.45349j - 0.24419k    1.4524 - 0.5i + 1.0238j
```

## 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  $p$ ,  $A ./ p = A ./ p$ .

---

### **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 R2018b**

## quinticpolytraj

Generate fifth-order trajectories

### Syntax

```
[q,qd,qdd,pp] = quinticpolytraj(wayPoints,timePoints,tSamples)
[q,qd,qdd,pp] = cubicpolytraj( ____,Name,Value)
```

### Description

`[q,qd,qdd,pp] = quinticpolytraj(wayPoints,timePoints,tSamples)` generates a fifth-order polynomial that achieves a given set of input waypoints with corresponding time points. The function outputs positions, velocities, and accelerations at the given time samples, `tSamples`. The function also returns the piecewise polynomial `pp` form of the polynomial trajectory with respect to time.

`[q,qd,qdd,pp] = cubicpolytraj( ____,Name,Value)` specifies additional parameters as `Name,Value` pair arguments using any combination of the previous syntaxes.

### Examples

#### Compute Quintic Trajectory for 2-D Planar Motion

Use the `cubicpolytraj` function with a given set of 2-D  $xy$  waypoints. Time points for the waypoints are also given.

```
wpts = [1 4 4 3 -2 0; 0 1 2 4 3 1];
tpts = 0:5;
```

Specify a time vector for sampling the trajectory. Sample at a smaller interval than the specified time points.

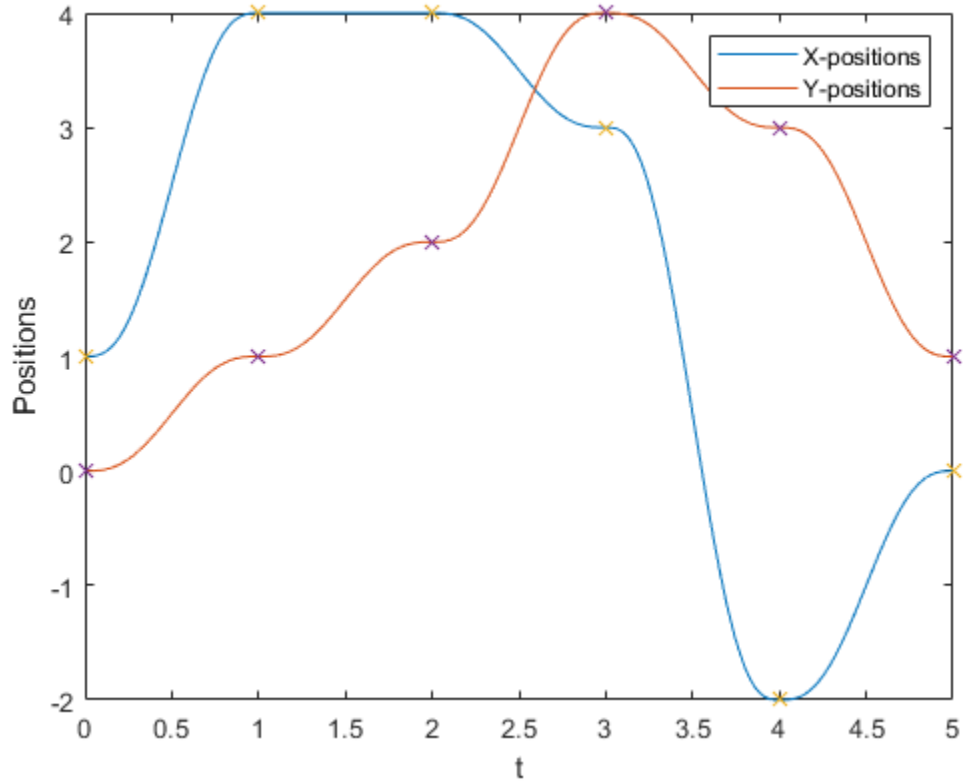
```
tvec = 0:0.01:5;
```

Compute the cubic trajectory. The function outputs the trajectory positions ( $q$ ), velocity ( $q\dot{d}$ ), acceleration ( $q\ddot{d}$ ), and polynomial coefficients ( $pp$ ) of the cubic polynomial.

```
[q, qd, qdd, pp] = quinticpolytraj(wpts, tpts, tvec);
```

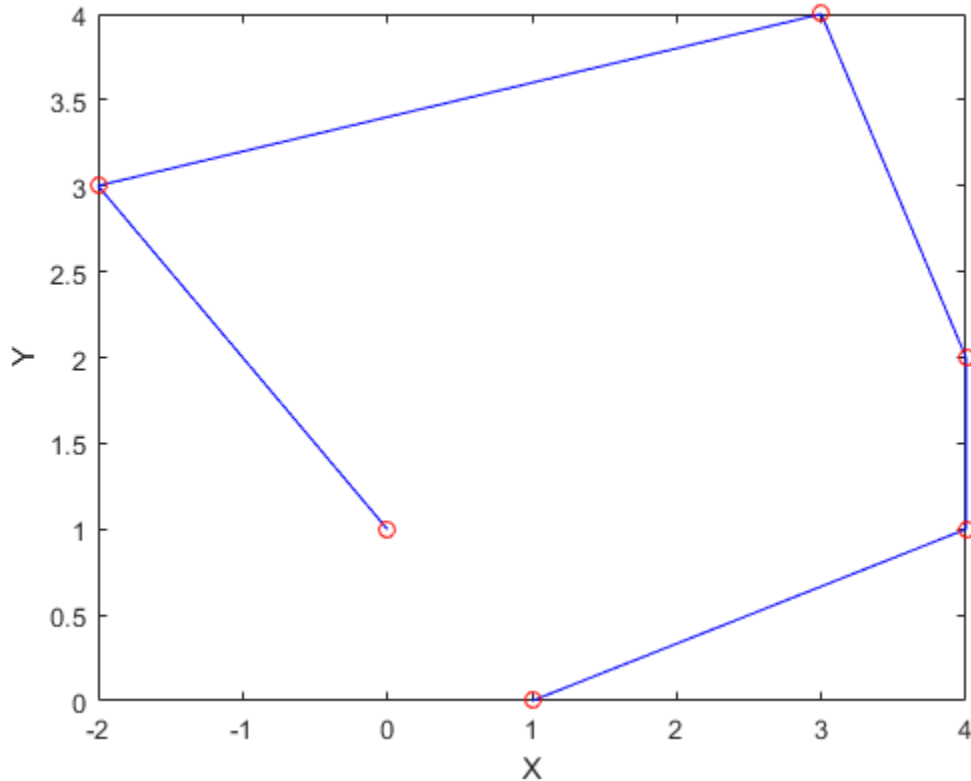
Plot the cubic trajectories for the  $x$ - and  $y$ -positions. Compare the trajectory with each waypoint.

```
plot(tvec, q)
hold all
plot(tpts, wpts, 'x')
xlabel('t')
ylabel('Positions')
legend('X-positions', 'Y-positions')
hold off
```



You can also verify the actual positions in the 2-D plane. Plot the separate rows of the  $q$  vector and the waypoints as x- and y- positions.

```
figure
plot(q(1,:),q(2,:), '-b', wpts(1,:),wpts(2,:), 'or')
xlabel('X')
ylabel('Y')
```



## Input Arguments

### **wayPoints** — Waypoints for trajectory

*n*-by-*p* matrix

Points for waypoints of trajectory, specified as an *n*-by-*p* matrix, where *n* is the dimension of the trajectory and *p* is the number of waypoints.

Example: [1 4 4 3 -2 0; 0 1 2 4 3 1]

Data Types: single | double

### **timePoints** — Time points for waypoints of trajectory

*p*-element vector

Time points for waypoints of trajectory, specified as a *p*-element vector.

Example: [0 2 4 5 8 10]

Data Types: single | double

### **tSamples** — Time samples for trajectory

*m*-element vector

Time samples for the trajectory, specified as an *m*-element vector. The output position, *q*, velocity, *qd*, and accelerations, *qdd*, are sampled at these time intervals.

Example: 0:0.01:10

Data Types: single | double

## **Name-Value Pair Arguments**

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

Example: 'VelocityBoundaryCondition',[1 0 -1 -1 0 0; 1 1 1 -1 -1 -1]

### **VelocityBoundaryCondition** — Velocity boundary conditions for each waypoint

*zeroes*(*n*,*p*) (default) | *n*-by-*p* matrix

Velocity boundary conditions for each waypoint, specified as the comma-separated pair consisting of 'VelocityBoundaryCondition' and an *n*-by-*p* matrix. Each row corresponds to the velocity at all of *p* waypoints for the respective variable in the trajectory.

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

Data Types: single | double

### **AccelerationBoundaryCondition** — Acceleration boundary conditions for each waypoint

*zeroes*(*n*,*p*) (default) | *n*-by-*p* matrix



Acceleration boundary conditions for each waypoint, specified as the comma-separated pair consisting of 'VelocityBoundaryCondition' and an  $n$ -by- $p$  matrix. Each row corresponds to the acceleration at all of  $p$  waypoints for the respective variable in the trajectory.

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

Data Types: single | double

## Output Arguments

### **q** — Positions of trajectory

$m$ -element vector

Positions of the trajectory at the given time samples in `tSamples`, returned as an  $m$ -element vector, where  $m$  is the length of `tSamples`.

Data Types: single | double

### **qd** — Velocities of trajectory

vector

Velocities of the trajectory at the given time samples in `tSamples`, returned as a vector.

Data Types: single | double

### **qdd** — Accelerations of trajectory

vector

Accelerations of the trajectory at the given time samples in `tSamples`, returned as a vector.

Data Types: single | double

### **pp** — Piecewise-polynomial

structure

Piecewise-polynomial, returned as a structure that defines the polynomial for each section of the piecewise trajectory. You can build your own piecewise polynomials using `mkpp`, or evaluate the polynomial at specified times using `ppval`. The structure contains the fields:

- `form`: 'pp'.

- **breaks**:  $p$ -element vector of times when the piecewise trajectory changes forms.  $p$  is the number of waypoints.
- **coefs**:  $n(p-1)$ -by-order matrix for the coefficients for the polynomials.  $n(p-1)$  is the dimension of the trajectory times the number of **pieces**. Each set of  $n$  rows defines the coefficients for the polynomial that described each variable trajectory.
- **pieces**:  $p-1$ . The number of breaks minus 1.
- **order**: Degree of the polynomial + 1. For example, cubic polynomials have an order of 4.
- **dim**:  $n$ . The dimension of the control point positions.

## Extended Capabilities

### C/C++ Code Generation

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

### See Also

[bsplinepolytraj](#) | [cubicpolytraj](#) | [rottraj](#) | [transformtraj](#) | [trapveltraj](#)

**Introduced in R2019a**

# 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  $m1, \dots, mN$  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  $m1, \dots, mN$  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 =
```

```
3×3 quaternion array
```

```
0.71601 - 0.048195i + 0.69548j + 0.036254k    -0.33542 - 0.39466i - 0.84503j  
0.31625 + 0.20986i + 0.29758j - 0.87601k     0.42409 - 0.047461i + 0.28419j  
0.16941 + 0.32961i - 0.74097j + 0.56002k     0.42141 + 0.88708i + 0.09635j
```

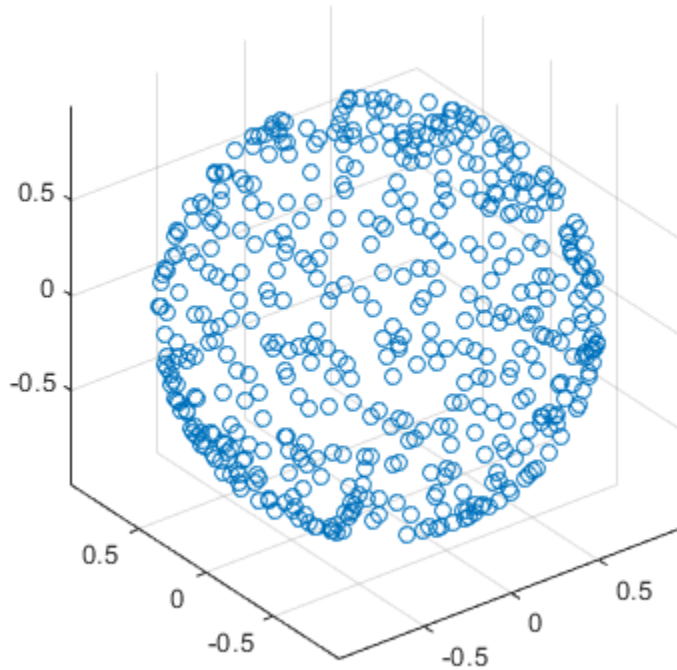
### Create Uniform Distribution of Random Rotations

Create a vector of 500 random quaternions. Use `rotatepoint` to visualize the distribution of the random rotations applied to point (1, 0, 0).

```
q = randrot(500,1);
```

```
pt = rotatepoint(q, [1 0 0]);
```

```
figure  
scatter3(pt(:,1), pt(:,2), pt(:,3))  
axis equal
```



## Input Arguments

**m** — Size of square matrix

integer

Size of square quaternion matrix, specified as an integer value. If  $m$  is 0 or negative, then  $R$  is returned as an empty matrix.

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

### **m1, . . . , mN — Size of each dimension**

two or more integer values

Size of each dimension, specified as two or more integer values. If the size of any dimension is 0 or negative, then R is returned as an empty array.

Example: `randrot(2,3)` returns a 2-by-3 matrix of random quaternions.

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

### **[m1, . . . , mN] — Vector of size of each dimension**

row vector of integer values

Vector of size of each dimension, specified as a row vector of two or more integer values. If the size of any dimension is 0 or negative, then R is returned as an empty array.

Example: `randrot([2,3])` returns a 2-by-3 matrix of random quaternions.

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

## Output Arguments

### **R — Random quaternions**

scalar | vector | matrix | multidimensional array

Random quaternions, returned as a quaternion or array of quaternions.

Data Types: `quaternion`

## 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™.

**Introduced in R2019a**

# readAllFieldNames

Get all available field names from ROS point cloud

## Syntax

```
fieldnames = readAllFieldNames(pcloud)
```

## Description

`fieldnames = readAllFieldNames(pcloud)` gets the names of all point fields that are stored in the `PointCloud2` object message, `pcloud`, and returns them in `fieldnames`.

## Examples

### Read All Fields From Point Cloud Message

Load sample ROS messages including a ROS point cloud message, `ptcloud`.

```
exampleHelperROSLoadMessages
```

Read all the field names available on the point cloud message.

```
fieldnames = readAllFieldNames(ptcloud)
```

```
fieldnames =
```

```
    1×4 cell array
```



'x' 'y' 'z' 'rgb'

## Input Arguments

### **pcloud** — Point cloud

PointCloud2 object handle

Point cloud, specified as a PointCloud2 object handle for a 'sensor\_msgs/PointCloud2' ROS message.

## Output Arguments

### **fieldnames** — List of field names in PointCloud2 object

cell array of character vectors

List of field names in PointCloud2 object, returned as a cell array of character vectors. If no fields exist in the object, fieldname returns an empty cell array.

## See Also

PointCloud2 | readField

**Introduced in R2015a**

# readBinaryOccupancyGrid

Read binary occupancy grid

## Syntax

```
map = readBinaryOccupancyGrid(msg)
map = readBinaryOccupancyGrid(msg, thresh)
map = readBinaryOccupancyGrid(msg, thresh, val)
```

## Description

`map = readBinaryOccupancyGrid(msg)` returns a `robotics.BinaryOccupancyGrid` object by reading the data inside a ROS message, `msg`, which must be a `'nav_msgs/OccupancyGrid'` message. All message data values greater than or equal to the occupancy threshold are set to occupied, `1`, in the map. All other values, including unknown values (`-1`) are set to unoccupied, `0`, in the map.

`map = readBinaryOccupancyGrid(msg, thresh)` specifies a threshold, `thresh`, for occupied values. All values greater than or equal to the threshold are set to occupied, `1`. All other values are set to unoccupied, `0`.

`map = readBinaryOccupancyGrid(msg, thresh, val)` specifies a value to set for unknown values (`-1`). By default, all unknown values are set to unoccupied, `0`.

## Examples

### Read Binary Occupancy Data from ROS Message

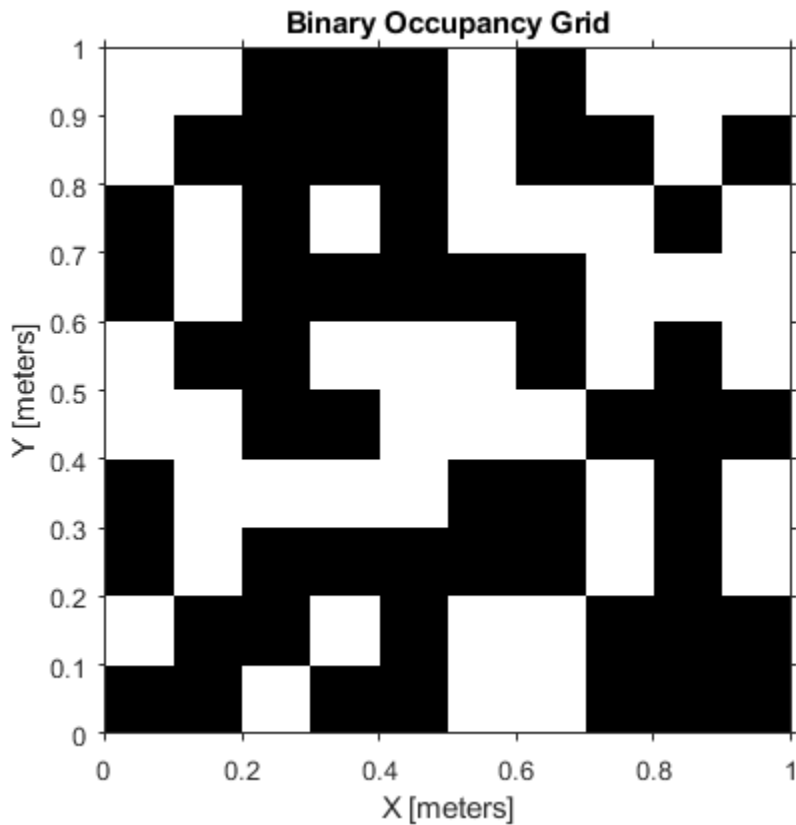
Create a occupancy grid message and populate it with data.

```
msg = rosmassage('nav_msgs/OccupancyGrid');
msg.Info.Height = 10;
msg.Info.Width = 10;
```

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

Read data from message. Show the map.

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



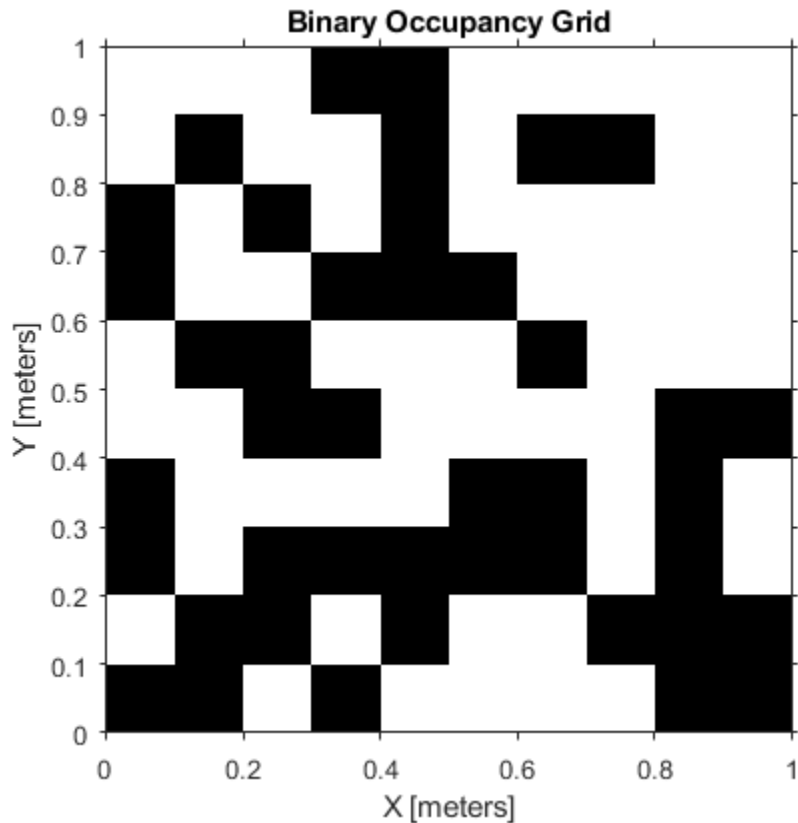
### Read Binary Occupancy Data from ROS Message Using Threshold and Unknown Value Replacement

Create an occupancy grid message and populate it with data.

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

Read data from message. Specify the threshold value and what unknown values should be set as. Show the map.

```
map = readBinaryOccupancyGrid(msg,65,1);  
show(map)
```



## Input Arguments

### **msg** — 'nav\_msgs/OccupancyGrid' ROS message

OccupancyGrid object handle

'nav\_msgs/OccupancyGrid' ROS message, specified as a OccupancyGrid object handle.

### **thresh** — Threshold for occupied values

50 (default) | scalar

Threshold for occupied values, specified as a scalar. Any value greater than or equal to the threshold is set to occupied, 1. All other values are set to unoccupied, 0.

Data Types: double

### **val** — Value to replace unknown values

0 (default) | 1

Value to replace unknown values, specified as either 0 or 1. Unknown message values (-1) are set to the given value.

Data Types: double | logical

## Output Arguments

### **map** — Binary occupancy grid

BinaryOccupancyGrid object handle

Binary occupancy grid, returned as a BinaryOccupancyGrid object handle. map is converted from a 'nav\_msgs/OccupancyGrid' message on the ROS network. It is an object with a grid of binary values, where 1 indicates an occupied location and 0 indications an unoccupied location.

## See Also

readOccupancyGrid | robotics.BinaryOccupancyGrid |  
robotics.OccupancyGrid | writeBinaryOccupancyGrid | writeOccupancyGrid

**Introduced in R2015a**

# readCartesian

Read laser scan ranges in Cartesian coordinates

## Syntax

```
cart = readCartesian(scan)
cart = readCartesian( ____, Name, Value)
[angles, cart] = readCartesian( ____ )
```

## Description

`cart = readCartesian(scan)` converts the polar measurements of the laser scan object, `scan`, into Cartesian coordinates, `cart`. This function uses the metadata in the message, such as angular resolution and opening angle of the laser scanner, to perform the conversion. Invalid range readings, usually represented as `NaN`, are ignored in this conversion.

`cart = readCartesian( ____, Name, Value)` provides additional options specified by one or more `Name, Value` pair arguments. `Name` must appear inside single quotes ( `' '` ). You can specify several name-value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

`[angles, cart] = readCartesian( ____ )` returns the scan angles, `angles` that are associated with each Cartesian coordinate. Angles are measured counter-clockwise around the positive  $z$ -axis, with the zero angle along the  $x$ -axis. `angles` is returned in radians and wrapped to the  $[-\pi, \pi]$  interval.

## Examples

### Get Cartesian Coordinates from Laser Scan

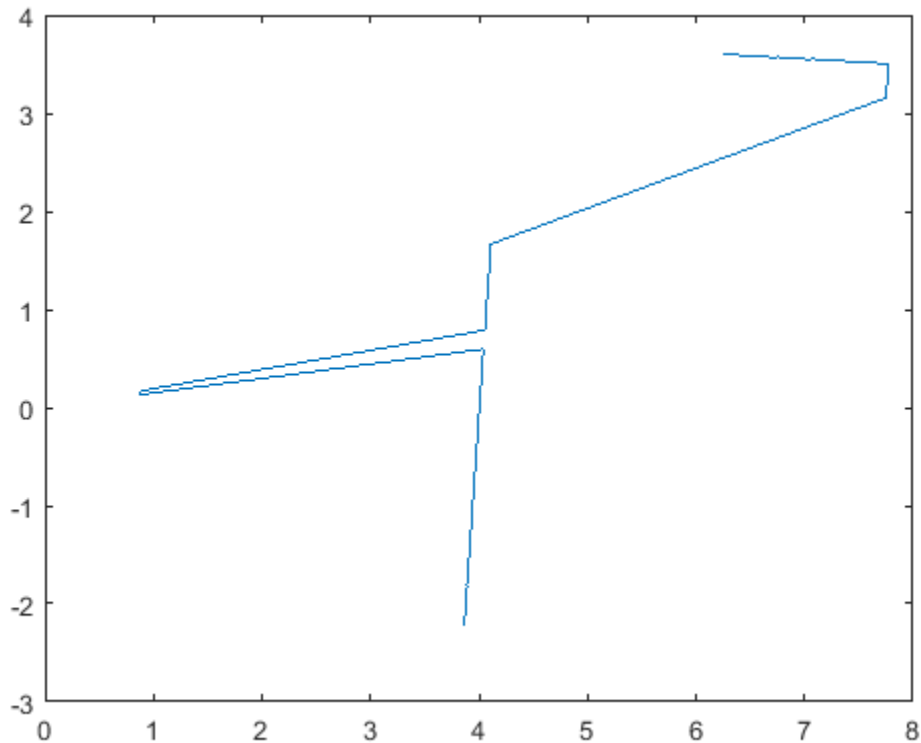
Connect to ROS network. Subscribe to a laser scan topic, and receive a message.

```
rosinit('192.168.154.131')
sub = rossubscriber('/scan');
scan = receive(sub);
```

Initializing global node /matlab\_global\_node\_60179 with NodeURI http://192.168.154.1:60179

Read the Cartesian points from the laser scan. Plot the laser scan.

```
cart = readCartesian(scan);
plot(cart(:,1),cart(:,2))
```



Shutdown ROS network.

```
rosshutdown
```



Shutting down global node /matlab\_global\_node\_60179 with NodeURI http://192.168.154.1:60

### **Get Cartesian Coordinates from Laser Scan With Scan Range**

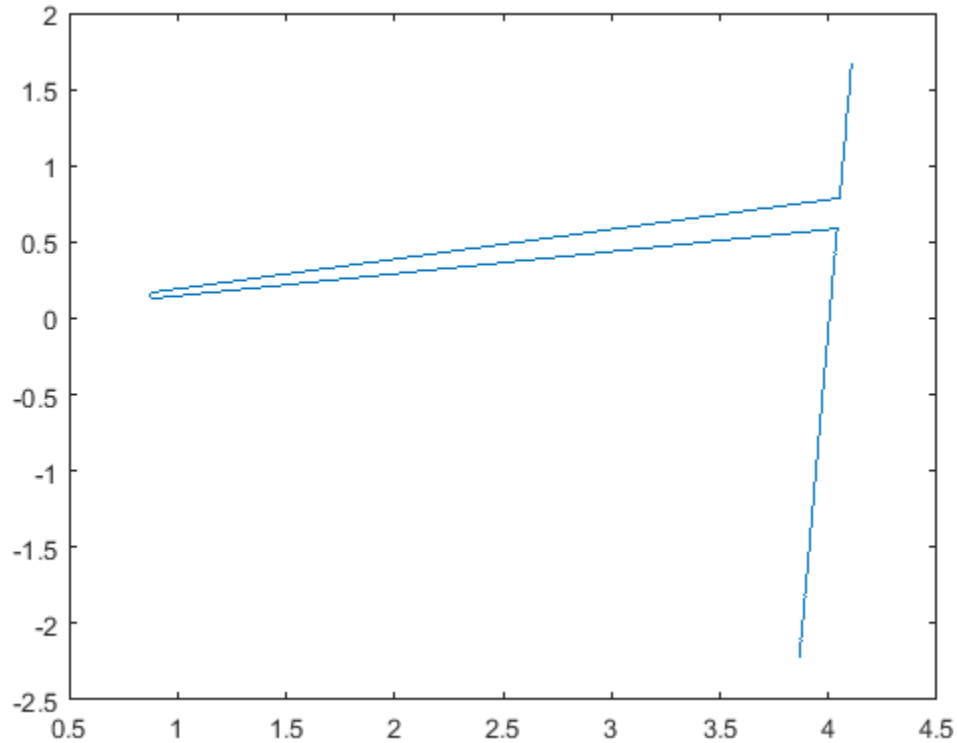
Connect to ROS network. Subscribe to a laser scan topic, and receive a message.

```
rosinit('192.168.154.131')  
sub = rossubscriber('/scan');  
scan = receive(sub);
```

Initializing global node /matlab\_global\_node\_98143 with NodeURI http://192.168.154.1:60

Read the Cartesian points from the laser scan with specified range limits. Plot the laser scan.

```
cart = readCartesian(scan, 'RangeLimit', [0.5 6]);  
plot(cart(:,1), cart(:,2))
```



Shutdown ROS network.

```
roshUTDOWN
```

```
Shutting down global node /matlab_global_node_98143 with NodeURI http://192.168.154.1:
```

## Input Arguments

**scan** — Laser scan message

LaserScan object handle

'sensor\_msgs/LaserScan' ROS message, specified as a LaserScan object handle.

## Name-Value Pair Arguments

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

Example: `'RangeLimits', [-2 2]`

### **RangeLimits** — Minimum and maximum range for scan in meters

`[scan.RangeMin scan.RangeMax]` (default) | 2-element `[min max]` vector

Minimum and maximum range for scan in meters, specified as a 2-element `[min max]` vector. All ranges smaller than `min` or larger than `max` are ignored during the conversion to Cartesian coordinates.

## Output Arguments

### **cart** — Cartesian coordinates of laser scan

*n*-by-2 matrix in meters

Cartesian coordinates of laser scan, returned as an *n*-by-2 matrix in meters.

### **angles** — Scan angles for laser scan data

*n*-by-1 matrix in radians

Scan angles for laser scan data, returned as an *n*-by-1 matrix in radians. Angles are measured counter-clockwise around the positive *z*-axis, with the zero angle along the *x*-axis. `angles` is returned in radians and wrapped to the `[-pi, pi]` interval.

## See Also

`plot` | `readScanAngles`

**Introduced in R2015a**

# readField

Read point cloud data based on field name

## Syntax

```
fielddata = readField(pcloud,fieldname)
```

## Description

`fielddata = readField(pcloud,fieldname)` reads the point field from the `PointCloud2` object, `pcloud`, specified by `fieldname` and returns it in `fielddata`. If `fieldname` does not exist, the function displays an error. To preserve the structure of the point cloud data, see “Preserving Point Cloud Structure” on page 2-325.

## Examples

### Read Specific Field From Point Cloud Message

Load sample ROS messages including a ROS point cloud message, `ptcloud`.

```
exampleHelperROSLoadMessages
```

Read the 'x' field name available on the point cloud message.

```
x = readField(ptcloud, 'x');
```

## Input Arguments

### **pcloud** — Point cloud

`PointCloud2` object handle

Point cloud, specified as a `PointCloud2` object handle for a `sensor_msgs/PointCloud2` ROS message.

**fieldname** — Field name of point cloud data

string scalar | character vector

Field name of point cloud data, specified as a string scalar or character vector. This string must match the field name exactly. If `fieldname` does not exist, the function displays an error.

## Output Arguments

**fielddata** — List of field values from point cloud

matrix

List of field values from point cloud, returned as a matrix. Each row of is a point cloud reading, where  $n$  is the number of points and  $c$  is the number of values for each point. If the point cloud object being read has the `PreserveStructureOnRead` property set to `true`, the points are returned as an  $h$ -by- $w$ -by- $c$  matrix. For more information, see “Preserving Point Cloud Structure” on page 2-325.

## Preserving Point Cloud Structure

Point cloud data can be organized in either 1-D lists or in 2-D image styles. 2-D image styles usually come from depth sensors or stereo cameras. The input `PointCloud2` object contains a `PreserveStructureOnRead` property that is either `true` or `false` (default). Suppose you set the property to `true`.

```
pcloud.PreserveStructureOnRead = true;
```

Now calling any read functions (`readXYZ`, `readRGB`, or `readField`) preserves the organizational structure of the point cloud. When you preserve the structure, the output matrices are of size  $m$ -by- $n$ -by- $d$ , where  $m$  is the height,  $n$  is the width, and  $d$  is the number of return values for each point. Otherwise, all points are returned as a  $x$ -by- $d$  list. This structure can only be preserved if the point cloud is organized.

## See Also

`PointCloud2` | `readAllFieldNames`

**Introduced in R2015a**

# readImage

Convert ROS image data into MATLAB image

## Syntax

```
img = readImage(msg)  
[img,alpha] = readImage(msg)
```

## Description

`img = readImage(msg)` converts the raw image data in the message object, `msg`, into an image matrix, `img`. You can call `readImage` using either 'sensor\_msgs/Image' or 'sensor\_msgs/CompressedImage' messages.

ROS image message data is stored in a format that is not compatible with further image processing in MATLAB. Based on the specified encoding, this function converts the data into an appropriate MATLAB image and returns it in `img`.

`[img,alpha] = readImage(msg)` returns the alpha channel of the image in `alpha`. If the image does not have an alpha channel, then `alpha` is empty.

## Examples

### Read ROS Image Data

Load sample ROS messages including a ROS image message, `img`.

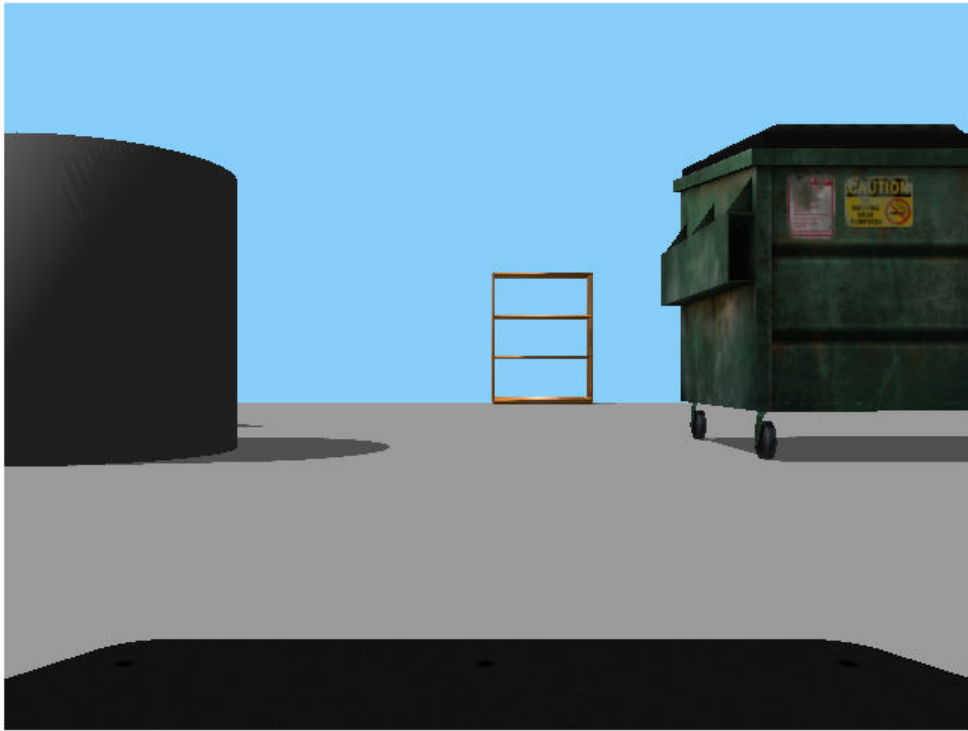
```
exampleHelperROSLoadMessages
```

Read the ROS image message as a MATLAB® image.

```
image = readImage(img);
```

Display the image.

`imshow(image)`



## Input Arguments

**msg** — ROS image message

Image object handle | CompressedImage object handle

'sensor\_msgs/Image' or 'sensor\_msgs/CompressedImage' ROS image message, specified as an Image or Compressed Image object handle.



## Output Arguments

### **img** — Image

grayscale image matrix | RGB image matrix |  $m$ -by- $n$ -by-3 array

Image, returned as a matrix representing a grayscale or RGB image or as  $m$ -by- $n$ -by-3 array, depending on the sensor image.

### **alpha** — Alpha channel

uint8 grayscale image

Alpha channel, returned as a uint8 grayscale image. If no alpha channel exists, **alpha** is empty.

---

**Note** For CompressedImage messages, you cannot output an Alpha channel.

---

## Supported Image Encodings

ROS image messages can have different encodings. The encodings supported for images are different for 'sensor\_msgs/Image' and 'sensor\_msgs/CompressedImage' message types. Less compressed images are supported. The following encodings for raw images of size  $M \times N$  are supported using the 'sensor\_msgs/Image' message type ('sensor\_msgs/CompressedImage' support is in bold):

- **rgb8, rgba8, bgr8, bgra8**: **img** is an rgb image of size  $M \times N \times 3$ . The alpha channel is returned in **alpha**. Each value in the outputs is represented as a uint8.
- **rgb16, rgba16, bgr16, bgra16**: **img** is an RGB image of size  $M \times N \times 3$ . The alpha channel is returned in **alpha**. Each value in the outputs is represented as a uint16.
- **mono8** images are returned as grayscale images of size  $M \times N \times 1$ . Each pixel value is represented as a uint8.
- **mono16** images are returned as grayscale images of size  $M \times N \times 1$ . Each pixel value is represented as a uint16.
- **32fcX** images are returned as floating-point images of size  $M \times N \times D$ , where  $D$  is 1, 2, 3, or 4. Each pixel value is represented as a single.
- **64fcX** images are returned as floating-point images of size  $M \times N \times D$ , where  $D$  is 1, 2, 3, or 4. Each pixel value is represented as a double.

- `8ucX` images are returned as matrices of size  $M \times N \times D$ , where  $D$  is 1, 2, 3, or 4. Each pixel value is represented as a `uint8`.
- `8scX` images are returned as matrices of size  $M \times N \times D$ , where  $D$  is 1, 2, 3, or 4. Each pixel value is represented as a `int8`.
- `16ucX` images are returned as matrices of size  $M \times N \times D$ , where  $D$  is 1, 2, 3, or 4. Each pixel value is represented as a `int16`.
- `16scX` images are returned as matrices of size  $M \times N \times D$ , where  $D$  is 1, 2, 3, or 4. Each pixel value is represented as a `int16`.
- `32scX` images are returned as matrices of size  $M \times N \times D$ , where  $D$  is 1, 2, 3, or 4. Each pixel value is represented as a `int32`.
- `bayer_X` images are returned as either Bayer matrices of size  $M \times N \times 1$ , or as a converted image of size  $M \times N \times 3$  (Image Processing Toolbox™ is required).

The following encoding for raw images of size  $M \times N$  is supported using the '`sensor_msgs/CompressedImage`' message type:

- `rgb8`, `rgba8`, `bgr8`, `bgra8`: `img` is an `rgb` image of size  $M \times N \times 3$ . The alpha channel is returned in `alpha`. Each output value is represented as a `uint8`.

## See Also

`writeImage`

**Introduced in R2015a**

# readMessages

Read messages from rosbag

## Syntax

```
msgs = readMessages(bag)
msgs = readMessages(bag, rows)
msgs = readMessages( ____, "DataFormat", "struct")
```

## Description

`msgs = readMessages(bag)` returns data from all the messages in the `BagSelection` object, `bag`. The messages are returned in a cell array of messages.

To get a `BagSelection` object, use `rosbag`.

`msgs = readMessages(bag, rows)` returns data from messages in the rows specified by `rows`. The range of the rows is `[1, bag.NumMessages]`.

`msgs = readMessages( ____, "DataFormat", "struct")` returns data as a cell array of structures using either set of the previous input arguments. Using structures can be significantly faster than using message objects, and custom message data can be read directly without loading message definitions using `rosgenmsg`.

## Examples

### Return ROS Messages as a Cell Array

Read rosbag and filter by topic and time.

```
bagselect = rosbag('ex_multiple_topics.bag');
bagselect2 = select(bagselect, 'Time', ...
[bagselect.StartTime bagselect.StartTime + 1], 'Topic', '/odom');
```

Return all messages as a cell array.

```
allMsgs = readMessages(bagselect2);
```

Return the first ten messages as a cell array.

```
firstMsgs = readMessages(bagselect2,1:10);
```

### Read Messages from a rosbag as a Structure

Load the rosbag.

```
bag = rosbag('ros_turtlesim.bag');
```

Select a specific topic.

```
bSel = select(bag, 'Topic', '/turtle1/pose');
```

Read messages as a structure. Specify the `DataFormat` name-value pair when reading the messages. Inspect the first structure in the returned cell array of structures.

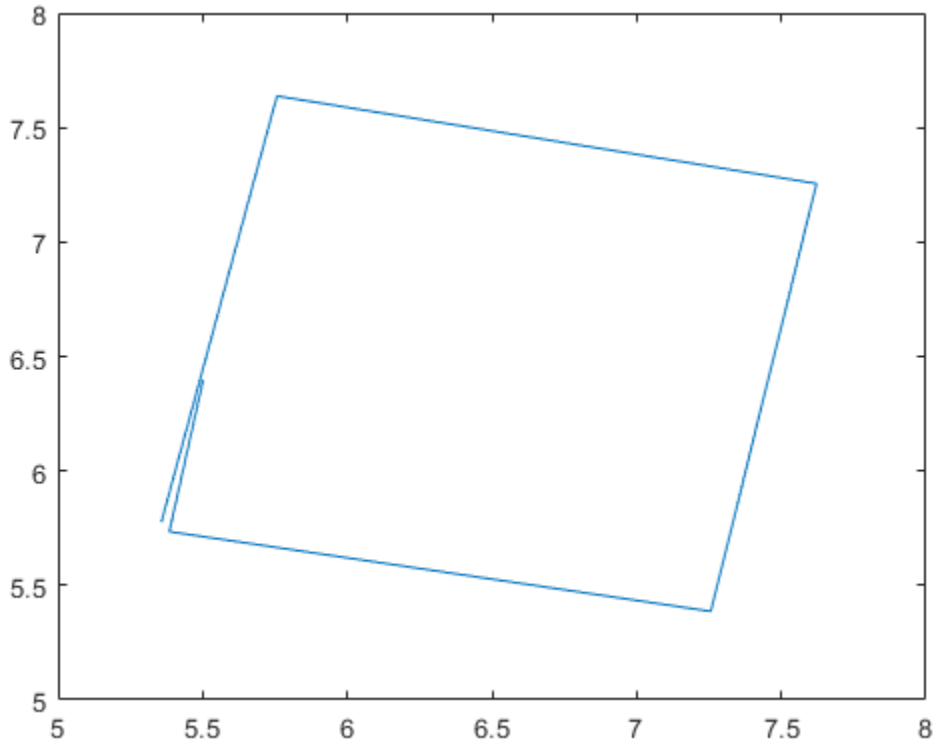
```
msgStructs = readMessages(bSel, 'DataFormat', 'struct');  
msgStructs{1}
```

```
ans = struct with fields:  
    MessageType: 'turtlesim/Pose'  
             X: 5.5016  
             Y: 6.3965  
             Theta: 4.5377  
    LinearVelocity: 1  
    AngularVelocity: 0
```

Extract the xy points from the messages and plot the robot trajectory.

Use `cellfun` to extract all the X and Y fields from the structure. These fields represent the xy positions of the robot during the rosbag recording.

```
xPoints = cellfun(@(m) double(m.X),msgStructs);  
yPoints = cellfun(@(m) double(m.Y),msgStructs);  
plot(xPoints,yPoints)
```



## Input Arguments

**bag** — Message of rosbag

BagSelection object

All the messages contained within a rosbag, specified as a BagSelection object.

**rows** — Rows of BagSelection object

*n*-element vector

Rows of `BagSelection` object, specified as  $n$ -element vector, where  $n$  is the number of rows to retrieve messages from. Each entry in the vector corresponds to a numbered message in the bag. The range of the rows is `[1, bag.NumMessage]`.

## Output Arguments

### **msgs** — ROS message data

object | cell array of message objects | cell array of structures

ROS message data, returned as an object, cell array of message objects, or cell array of structures. Data comes from the `BagSelection` object created using `rosbag`. You must specify `'DataFormat', 'struct'` in the function to get messages as a cell array of structures. Using structures can be significantly faster than using message objects, and custom message data can be read directly without loading message definitions using `rosgenmsg`.

## See Also

`rosbag` | `select` | `timeseries`

**Introduced in R2015a**

# readOccupancyGrid

Read occupancy grid message

## Syntax

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

## Description

`map = readOccupancyGrid(msg)` returns an `OccupancyGrid` object by reading the data inside a ROS message, `msg`, which must be a `'nav_msgs/OccupancyGrid'` message. All message data values are converted to probabilities from 0 to 1. The unknown values (-1) in the message are set as 0.5 in the map.

## Examples

### Read An OccupancyGrid Message from ROS

Create a `nav_msgs/OccupancyGrid` ROS message.

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

Populate the ROS occupancy grid message with data.

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

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

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

## Input Arguments

**msg** — 'nav\_msgs/OccupancyGrid' ROS message

OccupancyGrid object handle

'nav\_msgs/OccupancyGrid' ROS message, specified as an OccupancyGrid ROS message object handle.

## Output Arguments

**map** — **Occupancy grid**

robotics.OccupancyGrid object handle

Occupancy grid, returned as an robotics.OccupancyGrid object handle.

## See Also

OccupancyGrid | readBinaryOccupancyGrid | robotics.BinaryOccupancyGrid | robotics.OccupancyGrid | writeBinaryOccupancyGrid

**Introduced in R2016b**



# readOccupancyMap3D

Read 3-D map from Octomap ROS message

## Syntax

```
map = readOccupancyMap3D(msg)
```

## Description

`map = readOccupancyMap3D(msg)` reads the data inside a ROS 'octomap\_msgs/Octomap' message to return an `OccupancyMap3D` object. All message data values are converted to probabilities from 0 to 1.

## Examples

### Read Octomap ROS Messages

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

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

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

ROS Octomap message with properties:

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

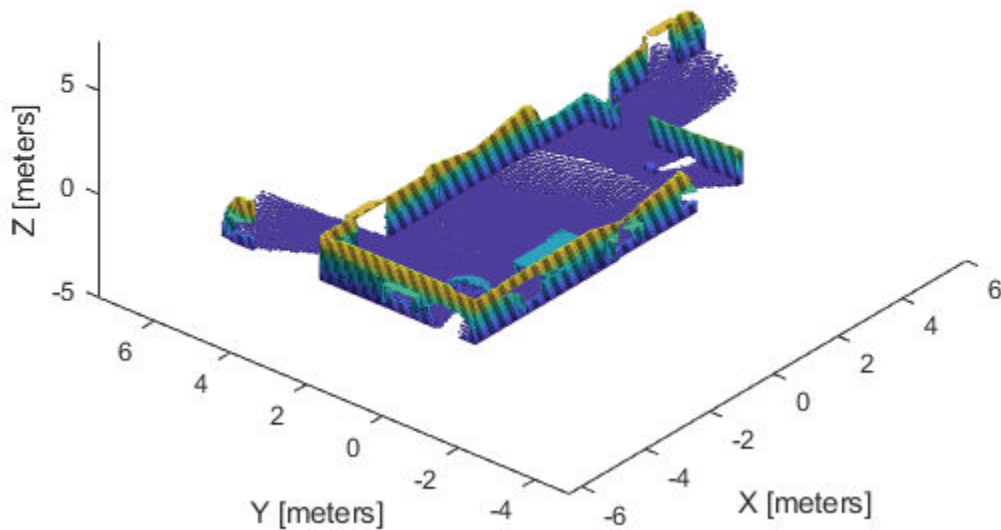
```
Data: [1175340x1 int8]
```

Use `showdetails` to show the contents of the message

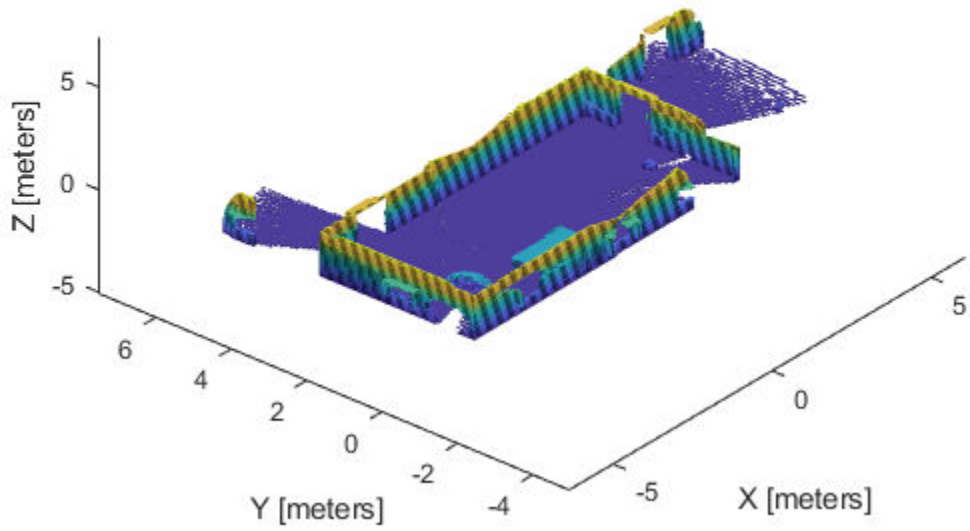
Read the data from the ROS messages into an `occupancyMap3D` object. Display each map.

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

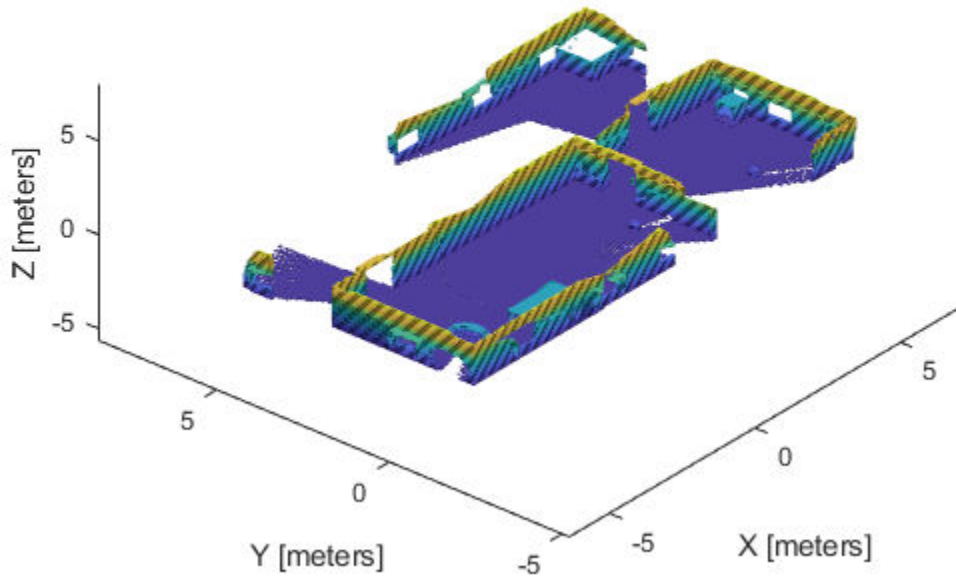
**Occupancy Map**



### Occupancy Map



### Occupancy Map



## Input Arguments

**msg** — 'octomap\_msgs/Octomap' ROS message

Octomap object handle

'octomap\_msgs/Octomap' ROS message, specified as an Octomap object handle. Get this message by subscribing to an 'octomap\_msgs/Octomap' topic using `rossubscriber` on a live ROS network or by creating your own message using `rosmessage`.

## Output Arguments

**map** — 3-D occupancy map

OccupancyMap3D object handle

3-D occupancy map, returned as an OccupancyMap3D object handle.

## See Also

OccupancyMap3D | rosmesssage | rossubscriber

**Introduced in R2018a**

# readRGB

Extract RGB values from point cloud data

## Syntax

```
rgb = readRGB(pcloud)
```

## Description

`rgb = readRGB(pcloud)` extracts the `[r g b]` values from all points in the `PointCloud2` object, `pcloud`, and returns them as an  $n$ -by-3 of  $n$  3-D point coordinates. If the point cloud does not contain the RGB field, this function displays an error. To preserve the structure of the point cloud data, see “Preserving Point Cloud Structure” on page 2-343.

## Examples

### Read RGB Values from ROS Point Cloud Message

Load sample ROS messages including a ROS point cloud message, `ptcloud`.

```
exampleHelperROSLoadMessages
```

Read the RGB values from the point cloud.

```
rgb = readRGB(ptcloud);
```

## Input Arguments

### **pcloud** — Point cloud

`PointCloud2` object handle

Point cloud, specified as a `PointCloud2` object handle for a 'sensor\_msgs/PointCloud2' ROS message.

## Output Arguments

**rgb** — List of RGB values from point cloud

matrix

List of RGB values from point cloud, returned as a matrix. By default, this is an  $n$ -by-3 matrix. If the point cloud object being read has the `PreserveStructureOnRead` property set to true, the points are returned as an  $h$ -by- $w$ -by-3 matrix. For more information, see “Preserving Point Cloud Structure” on page 2-343.

## Preserving Point Cloud Structure

Point cloud data can be organized in either 1-D lists or in 2-D image styles. 2-D image styles usually come from depth sensors or stereo cameras. The input `PointCloud2` object contains a `PreserveStructureOnRead` property that is either true or false (default). Suppose that you set the property to true.

```
pcloud.PreserveStructureOnRead = true;
```

Now calling any read functions (`readXYZ`, `readRGB`, or `readField`) preserves the organizational structure of the point cloud. When you preserve the structure, the output matrices are of size  $m$ -by- $n$ -by- $d$ , where  $m$  is the height,  $n$  is the width, and  $d$  is the number of return values for each point. Otherwise, all points are returned as an  $x$ -by- $d$  list. This structure can only be preserved if the point cloud is organized.

## See Also

`PointCloud2` | `readField` | `readXYZ`

**Introduced in R2015a**

# readScanAngles

Return scan angles for laser scan range readings

## Syntax

```
angles = readScanAngles(scan)
```

## Description

`angles = readScanAngles(scan)` calculates the scan angles, `angles`, corresponding to the range readings in the laser scan message, `scan`. Angles are measured counter-clockwise around the positive  $z$ -axis, with the zero angle along the  $x$ -axis. `angles` is returned in radians and wrapped to the  $[-\pi, \pi]$  interval.

## Examples

### Read Scan Angles from ROS Laser Scan Message

Load sample ROS messages including a ROS laser scan message, `scan`.

```
exampleHelperROSLoadMessages
```

Read the scan angles from the laser scan.

```
angles = readScanAngles(scan);
```

## Input Arguments

### **scan** — Laser scan message

LaserScan object handle

'sensor\_msgs/LaserScan' ROS message, specified as a LaserScan object handle.



## Output Arguments

### **angles** — Scan angles for laser scan data

*n*-by-1 matrix in radians

Scan angles for laser scan data, returned as an *n*-by-1 matrix in radians. Angles are measured counter-clockwise around the positive *z*-axis, with the zero angle along the *x*-axis. `angles` is returned in radians and wrapped to the  $[-\pi, \pi]$  interval.

## See Also

`plot` | `readCartesian`

**Introduced in R2015a**

# readXYZ

Extract XYZ coordinates from point cloud data

## Syntax

```
xyz = readXYZ(pcloud)
```

## Description

`xyz = readXYZ(pcloud)` extracts the [x y z] coordinates from all points in the `PointCloud2` object, `pcloud`, and returns them as an  $n$ -by-3 matrix of  $n$  3-D point coordinates. If the point cloud does not contain the `x`, `y`, and `z` fields, this function returns an error. Points that contain NaN are preserved in the output. To preserve the structure of the point cloud data, see “Preserving Point Cloud Structure” on page 2-347.

## Examples

### Read XYZ Values from ROS Point Cloud Message

Load sample ROS messages including a ROS point cloud message, `ptcloud`.

```
exampleHelperROSLoadMessages
```

Read the XYZ values from the point cloud.

```
xyz = readXYZ(ptcloud);
```

## Input Arguments

### **pcloud** — Point cloud

`PointCloud2` object handle

Point cloud, specified as a `PointCloud2` object handle for a 'sensor\_msgs/PointCloud2' ROS message.

## Output Arguments

**xyz** — List of XYZ values from point cloud

matrix

List of XYZ values from point cloud, returned as a matrix. By default, this is a  $n$ -by-3 matrix. If the point cloud object being read has the `PreserveStructureOnRead` property set to `true`, the points are returned as an  $h$ -by- $w$ -by-3 matrix. For more information, see “Preserving Point Cloud Structure” on page 2-347.

## Preserving Point Cloud Structure

Point cloud data can be organized in either 1-D lists or in 2-D image styles. 2-D image styles usually come from depth sensors or stereo cameras. The input `PointCloud2` object contains a `PreserveStructureOnRead` property that is either `true` or `false` (default). Suppose you set the property to `true`.

```
pcloud.PreserveStructureOnRead = true;
```

Now calling any read functions (`readXYZ`, `readRGB`, or `readField`) preserves the organizational structure of the point cloud. When you preserve the structure, the output matrices are of size  $m$ -by- $n$ -by- $d$ , where  $m$  is the height,  $n$  is the width, and  $d$  is the number of return values for each point. Otherwise, all points are returned as a  $x$ -by- $d$  list. This structure can only be preserved if the point cloud is organized.

## See Also

`PointCloud2` | `readField` | `readRGB`

**Introduced in R2015a**

# receive

Wait for new ROS message

## Syntax

```
msg = receive(sub)
msg = receive(sub,timeout)
```

## Description

`msg = receive(sub)` waits for MATLAB to receive a topic message from the specified subscriber, `sub`, and returns it as `msg`.

`msg = receive(sub,timeout)` specifies in `timeout` the number of seconds to wait for a message. If a message is not received within the timeout limit, the software throws an error.

## Examples

### Create A Subscriber and Get Data From ROS

Connect to a ROS network. Set up a sample ROS network. The  `'/scan '` topic is being published on the network.

```
rosinit
```

```
Initializing ROS master on http://bat5742win64:64693/.
Initializing global node /matlab_global_node_49772 with NodeURI http://bat5742win64:64693/
```

```
exampleHelperROSCreateSampleNetwork
```

Create a subscriber for the  `'/scan '` topic. Wait for the subscriber to register with the master.

```
sub = rossubscriber('/scan');
pause(1);
```

Receive data from the subscriber as a ROS message. Specify a 10 second timeout.

```
msg2 = receive(sub,10)
```

```
msg2 =
  ROS LaserScan message with properties:

      MessageType: 'sensor_msgs/LaserScan'
      Header: [1x1 Header]
      AngleMin: -0.5216
      AngleMax: 0.5243
  AngleIncrement: 0.0016
  TimeIncrement: 0
      ScanTime: 0.0330
      RangeMin: 0.4500
      RangeMax: 10
      Ranges: [640x1 single]
  Intensities: [0x1 single]
```

Use `showdetails` to show the contents of the message

Shutdown ROS network.

```
roshutdown
```

```
Shutting down global node /matlab_global_node_49772 with NodeURI http://bat5742win64:64693/
Shutting down ROS master on http://bat5742win64:64693/.
```

## Create, Send, And Receive ROS Messages

Set up a publisher and subscriber to send and receive a message on a ROS network.

Connect to a ROS network.

```
roslaunch
```

```
Initializing ROS master on http://AH-SRADFORD:11311/.
Initializing global node /matlab_global_node_10876 with NodeURI http://AH-SRADFORD:6511/.
```

Create a publisher with a specific topic and message type. You can also return a default message to send using this publisher.

```
[pub,msg] = rospublisher('position','geometry_msgs/Point');
```

Modify the message before sending over the network.

```
msg.X = 1;
msg.Y = 2;
send(pub,msg);
```

Create a subscriber and wait for the latest message. Verify the message is the one you sent.

```
sub = rossubscriber('position')
pause(1);
sub.LatestMessage
```

```
sub =
```

```
Subscriber with properties:
```

```
    TopicName: '/position'
    MessageType: 'geometry_msgs/Point'
    LatestMessage: [0×1 Point]
    BufferSize: 1
    NewMessageFcn: []
```

```
ans =
```

```
ROS Point message with properties:
```

```
    MessageType: 'geometry_msgs/Point'
         X: 1
         Y: 2
         Z: 0
```

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

Shut down ROS network.

```
roshutdown
```

---

```
Shutting down global node /matlab_global_node_10876 with NodeURI http://AH-SRADFORD:65535  
Shutting down ROS master on http://AH-SRADFORD:11311/.
```

## Read Specific Field From Point Cloud Message

Load sample ROS messages including a ROS point cloud message, `ptcloud`.

```
exampleHelperROSLoadMessages
```

Read the 'x' field name available on the point cloud message.

```
x = readField(ptcloud, 'x');
```

## Input Arguments

### **sub** — ROS subscriber

Subscriber object handle

ROS subscriber, specified as a `Subscriber` object handle. You can create the subscriber using `rossubscriber`.

### **timeout** — Timeout for receiving a message

scalar in seconds

Timeout for receiving a message, specified as a scalar in seconds.

## Output Arguments

### **msg** — ROS message

Message object handle

ROS message, returned as a `Message` object handle.

## See Also

`rosmessage` | `rospublisher` | `rossubscriber` | `rostopic` | `send`

## **Topics**

“Exchange Data with ROS Publishers and Subscribers”

**Introduced in R2015a**



# removeEdges

**Package:** robotics

Remove loop closure edges from graph

## Syntax

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

## Description

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

## Input Arguments

**poseGraph** — Pose graph

PoseGraph object | PoseGraph3D object

Pose graph, specified as a PoseGraph or PoseGraph3D object.

**edgeIDs** — Edge IDs

vector of positive integers

Edge IDs, specified as a vector of positive integers.

## Extended Capabilities

### C/C++ Code Generation

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

Use this syntax when constructing PoseGraph or PoseGraph3D objects for code generation:

```
poseGraph =  
robotics.PoseGraph( 'MaxNumEdges', maxEdges, 'MaxNumNodes', maxNodes )
```

specifies an upper bound on the number of edges and nodes allowed in the pose graph when generating code. This limit is only required when generating code.

## See Also

### Functions

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

### Objects

`robotics.LidarSLAM` | `robotics.PoseGraph` | `robotics.PoseGraph3D`

## Topics

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

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

### Introduced in R2018a

# removeInvalidData

Remove invalid range and angle data

## Syntax

```
validScan = removeInvalidData(scan)
validScan = removeInvalidData(scan,Name,Value)
```

## Description

`validScan = removeInvalidData(scan)` returns a new `lidarScan` object with all `Inf` and `NaN` values from the input `scan` removed. The corresponding angle readings are also removed.

`validScan = removeInvalidData(scan,Name,Value)` provides additional options specified by one or more `Name, Value` pairs.

## Examples

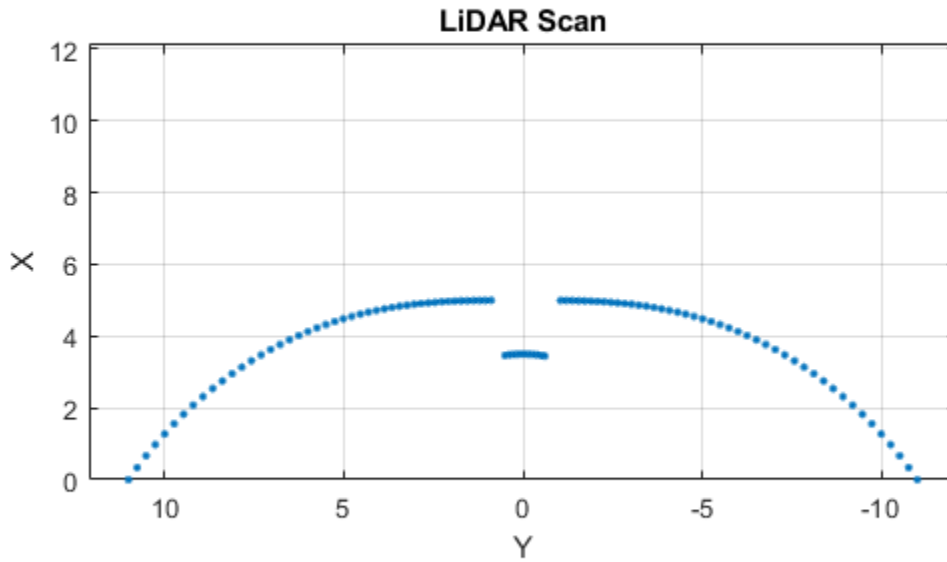
### Plot Lidar Scan and Remove Invalid Points

Specify lidar data as vectors of ranges and angles. These values include readings outside of the sensors range.

```
x = linspace(-2,2);
ranges = abs((1.5).*x.^2 + 5);
ranges(45:55) = 3.5;
angles = linspace(-pi/2,pi/2,numel(ranges));
```

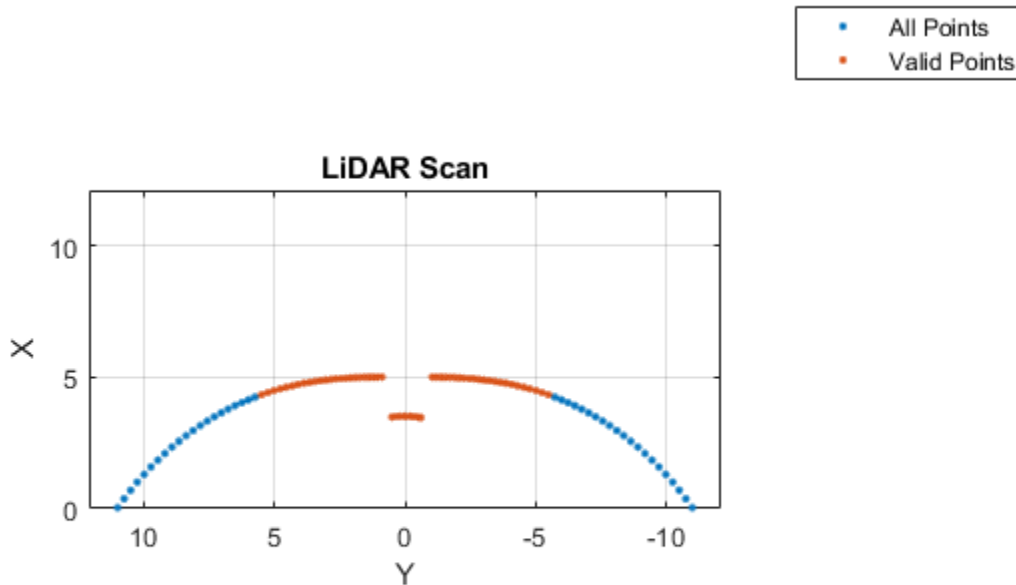
Create a lidar scan by specifying the ranges and angles. Plot all points of the lidar scan.

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



Remove invalid points based on a specified minimum and maximum range.

```
minRange = 0.1;  
maxRange = 7;  
scan2 = removeInvalidData(scan, 'RangeLimits', [minRange maxRange]);  
hold on  
plot(scan2)  
legend('All Points', 'Valid Points')
```



## Input Arguments

**scan** — Lidar scan readings

lidarScan object

Lidar scan readings, specified as a lidarScan object.

## Name-Value Pair Arguments

Specify optional comma-separated pairs of Name, Value arguments. Name is the argument name and Value is the corresponding value. Name must appear inside quotes.

You can specify several name and value pair arguments in any order as `Name1, Value1, . . . , NameN, ValueN`.

Example: `["RangeLimits", [0.05 2]]`

### **RangeLimits — Range reading limits**

two-element vector

Range reading limits, specified as a two-element vector, `[minRange maxRange]`, in meters. All range readings and corresponding angles outside these range limits are removed

Data Types: `single | double`

### **AngleLimits — Angle limits**

two-element vector

Angle limits, specified as a two-element vector, `[minAngle maxAngle]` in radians. All angles and corresponding range readings outside these angle limits are removed.

Angles are measured counter-clockwise around the positive z-axis.

Data Types: `single | double`

## **Output Arguments**

### **validScan — Lidar scan readings**

lidarScan object

Lidar scan readings, specified as a `lidarScan` object. All invalid lidar scan readings are removed.

## **See Also**

`lidarScan | matchScans | transformScan`

**Introduced in R2017b**

# reset

Reset Rate object

## Syntax

```
reset(rate)
```

## Description

`reset(rate)` resets the state of the `Rate` object, including the elapsed time and all statistics about previous periods. `reset` is useful if you want to run multiple successive loops at the same rate, or if the object is created before the loop is executed.

## Input Arguments

**rate** — Rate object

handle

Rate object, specified as a handle. This object contains the information for the desired rate and other information about the execution. See `robotics.Rate` for more information.

## Examples

### Run Loop At Fixed Rate and Reset Rate Object

Create a `Rate` object for running at 20 Hz.

```
r = robotics.Rate(2);
```

Start a loop and control operation using the `Rate` object.

```
for i = 1:30
    % Your code goes here
    waitfor(r);
end
```

Display the Rate object properties after loop operation.

```
disp(r)
```

```
Rate with properties:
```

```
    DesiredRate: 2
    DesiredPeriod: 0.5000
    OverrunAction: 'slip'
    TotalElapsedTime: 15.0270
    LastPeriod: 0.5000
```

Reset the object to restart the time statistics.

```
reset(r);
disp(r)
```

```
Rate with properties:
```

```
    DesiredRate: 2
    DesiredPeriod: 0.5000
    OverrunAction: 'slip'
    TotalElapsedTime: 0.0273
    LastPeriod: NaN
```

## See Also

`robotics.Rate` | `rosclock` | `waitfor`

## Topics

“Execute Code at a Fixed-Rate”

**Introduced in R2016a**



# roboticsAddons

Install add-ons for robotics

## Syntax

```
roboticsAddons
```

## Description

`roboticsAddons` allows you to download and install add-ons for Robotics System Toolbox. Use this function to open the Add-ons Explorer to browse the available add-ons.

## Examples

### Install Add-ons for Robotics System Toolbox™

```
roboticsAddons
```

## See Also

### Topics

“Install Robotics System Toolbox Add-ons”

“ROS Custom Message Support”

“Get Add-Ons” (MATLAB)

“Manage Your Add-Ons” (MATLAB)

**Introduced in R2016a**

## **roboticsSupportPackages**

Download and install support packages for Robotics System Toolbox

---

**Note** `roboticsSupportPackages` has been removed. Use `roboticsAddons` instead.

---

### **Syntax**

`roboticsSupportPackages`

### **Description**

`roboticsSupportPackages` opens the Support Package Installer to download and install support packages for Robotics System Toolbox. For more details, see “Install Robotics System Toolbox Add-ons”

### **Examples**

#### **Open Robotics System Toolbox Support Package Installer**

`roboticsSupportPackages`

**Introduced in R2015a**

## rosaction

Retrieve information about ROS actions

### Syntax

```
rosaction list  
rosaction info actionname  
rosaction type actionname
```

```
actionlist = rosaction("list")  
actioninfo = rosaction("info",actionname)  
actiontype = rosaction("type",actionname)
```

### Description

`rosaction list` returns a list of available ROS actions from the ROS network.

`rosaction info actionname` returns the action type, message types, action server, and action clients for the specified action name.

`rosaction type actionname` returns the action type for the specified action name.

`actionlist = rosaction("list")` returns a list of available ROS actions from the ROS network.

`actioninfo = rosaction("info",actionname)` returns a structure containing the action type, message types, action server, and action clients for the specified action name.

`actiontype = rosaction("type",actionname)` returns the action type for the specified action name.

### Examples

### Get Information About ROS Actions

Get information about ROS actions that are available from the ROS network. You must be connected to a ROS network using `rosinit`.

Connect to a ROS network. You must be connected to a ROS network to gather information about what actions are available. Action types must be set up beforehand with a ROS action server running on the network. Replace `ipaddress` with your network address.

```
ipaddress = '192.168.154.131';  
rosinit(ipaddress)
```

```
Initializing global node /matlab_global_node_87036 with NodeURI http://192.168.154.1:62
```

List the actions available on the network. The only action setup on this network is the `'/fibonacci'` action.

```
rosaction list
```

```
/fibonacci
```

Get information about a specific ROS action type. The action type, message types, action server, and clients are displayed.

```
rosaction info /fibonacci
```

```
Action Type: actionlib_tutorials/Fibonacci
```

```
Goal Message Type: actionlib_tutorials/FibonacciGoal
```

```
Feedback Message Type: actionlib_tutorials/FibonacciFeedback
```

```
Result Message Type: actionlib_tutorials/FibonacciResult
```

```
Action Server:
```

```
* /fibonacci (http://192.168.154.131:38213/)
```

```
Action Clients: None
```

Disconnect from the ROS network.

```
roshutdown
```

---

Shutting down global node /matlab\_global\_node\_87036 with NodeURI http://192.168.154.1:

## Input Arguments

### **actionname** — ROS action name

string scalar | character vector

ROS action name, specified as a string scalar or character vector. The action name must match one of the topics that `rosaction("list")` outputs.

## Output Arguments

### **actionlist** — List of actions available

cell array of character vectors

List of actions available on the ROS network, returned as a cell array of character vectors.

### **actioninfo** — Information about a ROS action

structure

Information about a ROS action, returned as a structure. `actioninfo` contains the following fields:

- `ActionType`
- `GoalMessageType`
- `FeedbackMessageType`
- `ResultMessageType`
- `ActionServer`
- `ActionClients`

For more information about ROS actions, see “ROS Actions Overview”.

### **actiontype** — Type of ROS action

character vector

Type of ROS action, returned as a character vector.

## **See Also**

`cancelGoal` | `rosmessage` | `rostopic` | `sendGoal` | `waitForServer`

## **Topics**

“ROS Actions Overview”

“Move a Turtlebot Robot Using ROS Actions”

**Introduced in R2016b**

# rosvbag

Open and parse rosvbag log file

## Syntax

```
bag = rosvbag(filename)
bagInfo = rosvbag('info',filename)
rosvbag info filename
```

## Description

`bag = rosvbag(filename)` creates an indexable `BagSelection` object, `bag`, that contains all the message indexes from the rosvbag at path `filename`. To get a `BagSelection` object, use `rosvbag`. To access the data, call `readMessages` or `timeseries` to extract relevant data.

A rosvbag, or bag, is a file format for storing ROS message data. They are used primarily to log messages within the ROS network. You can use these bags for offline analysis, visualization, and storage. See the ROS Wiki page for more information about rosvbags.

`bagInfo = rosvbag('info',filename)` returns information as a structure, `bagInfo`, about the contents of the rosvbag at `filename`.

`rosvbag info filename` displays information in the MATLAB command window about the contents of the rosvbag at `filename`. The information includes the number of messages, start and end times, topics, and message types.

## Examples

### Retrieve Information from rosvbag

Retrieve information from the rosvbag. Specify the full path to the rosvbag if it is not already available on the MATLAB® path.

```
bagsselect = rosbag('ex_multiple_topics.bag');
```

Select a subset of the messages, filtered by time and topic.

```
bagsselect2 = select(bagsselect, 'Time', ...  
    [bagsselect.StartTime bagsselect.StartTime + 1], 'Topic', '/odom');
```

### Display rosbag Information from File

To view information about a rosbag log file, use `rosbag info filename`, where *filename* is a rosbag (.bag) file.

```
rosbag info 'ex_multiple_topics.bag'
```

```
Path:      C:\TEMP\Bdoc19a_1067994_6688\ib99EA80\22\tpbda3f8b1\robotics-ex61825935\ex_m  
Version:   2.0  
Duration:  2:00s (120s)  
Start:     Dec 31 1969 19:03:21.34 (201.34)  
End:       Dec 31 1969 19:05:21.34 (321.34)  
Size:      23.6 MB  
Messages:  36963  
Types:     gazebo_msgs/LinkStates [48c080191eb15c41858319b4d8a609c2]  
           nav_msgs/Odometry     [cd5e73d190d741a2f92e81eda573aca7]  
           rosgraph_msgs/Clock   [a9c97c1d230cfc112e270351a944ee47]  
           sensor_msgs/LaserScan [90c7ef2dc6895d81024acba2ac42f369]  
Topics:    /clock                12001 msgs : rosgraph_msgs/Clock  
           /gazebo/link_states 11999 msgs : gazebo_msgs/LinkStates  
           /odom                11998 msgs : nav_msgs/Odometry  
           /scan                 965 msgs  : sensor_msgs/LaserScan
```

### Get Transformations from rosbag File

Get transformations from rosbag (.bag) files by loading the rosbag and checking the available frames. From these frames, use `getTransform` to query the transformation between two coordinate frames.

Load the rosbag.

```
bag = rosbag('ros_turtlesim.bag');
```

Get a list of available frames.



```
frames = bag.AvailableFrames;
```

Get the latest transformation between two coordinate frames.

```
tf = getTransform(bag, 'world', frames{1});
```

Check for a transformation available at a specific time and retrieve the transformation. Use `canTransform` to check if the transformation is available. Specify the time using `rostime`.

```
tfTime = rostime(bag.StartTime + 1);
if (canTransform(bag, 'world', frames{1}, tfTime))
    tf2 = getTransform(bag, 'world', frames{1}, tfTime);
end
```

### Read Messages from a rosvbag as a Structure

Load the rosvbag.

```
bag = rosvbag('ros_turtlesim.bag');
```

Select a specific topic.

```
bSel = select(bag, 'Topic', '/turtle1/pose');
```

Read messages as a structure. Specify the `DataFormat` name-value pair when reading the messages. Inspect the first structure in the returned cell array of structures.

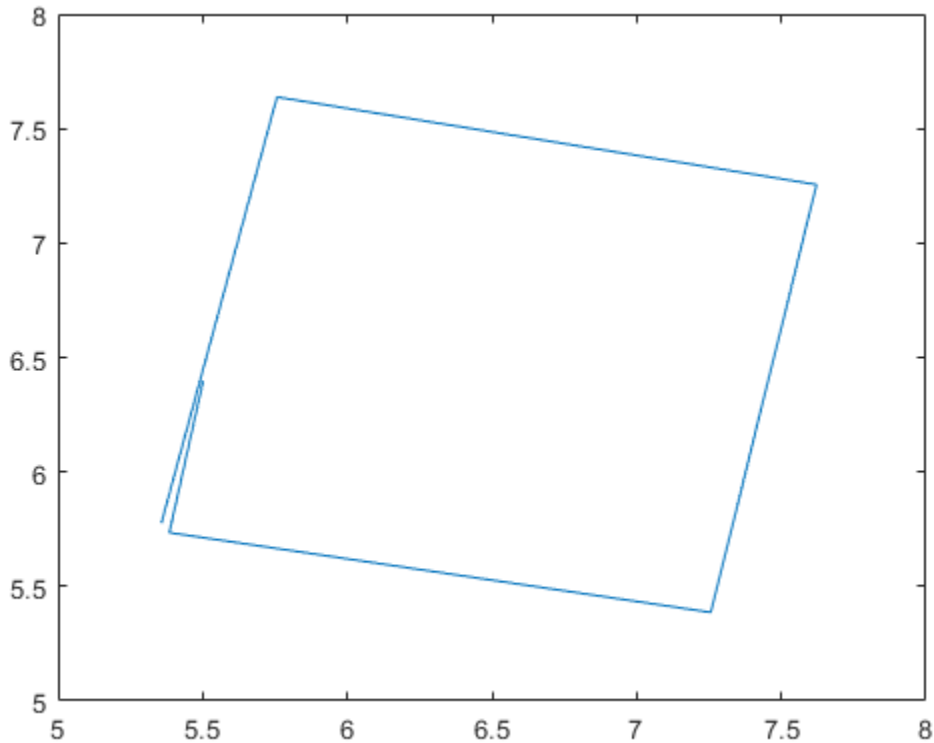
```
msgStructs = readMessages(bSel, 'DataFormat', 'struct');
msgStructs{1}
```

```
ans = struct with fields:
    MessageType: 'turtlesim/Pose'
        X: 5.5016
        Y: 6.3965
        Theta: 4.5377
    LinearVelocity: 1
    AngularVelocity: 0
```

Extract the `xy` points from the messages and plot the robot trajectory.

Use `cellfun` to extract all the `X` and `Y` fields from the structure. These fields represent the `xy` positions of the robot during the rosvbag recording.

```
xPoints = cellfun(@(m) double(m.X),msgStructs);  
yPoints = cellfun(@(m) double(m.Y),msgStructs);  
plot(xPoints,yPoints)
```



## Input Arguments

**filename** — Name of rosbag file and path

string scalar | character vector

Name of file and path for the rosbag you want to access, specified as a string scalar or character vector. This path can be relative or absolute.

## Output Arguments

### **bag** — Selection of rosvbag messages

BagSelection object handle

Selection of rosvbag messages, returned as a BagSelection object handle.

### **bagInfo** — Information about contents of rosvbag

structure

Information about the contents of the rosvbag, returned as a structure. This structure contains fields related to the rosvbag file and its contents. A sample output for a rosvbag as a structure is:

```

Path:      \ros\data\ex_multiple_topics.bag
Version:   2.0
Duration:  2:00s (120s)
Start:     Dec 31 1969 19:03:21.34 (201.34)
End:       Dec 31 1969 19:05:21.34 (321.34)
Size:      23.6 MB
Messages:  36963
Types:     gazebo_msgs/LinkStates [48c080191eb15c41858319b4d8a609c2]
           nav_msgs/Odometry      [cd5e73d190d741a2f92e81eda573aca7]
           rosvgraph_msgs/Clock   [a9c97c1d230cfc112e270351a944ee47]
           sensor_msgs/LaserScan  [90c7ef2dc6895d81024acba2ac42f369]
Topics:    /clock                  12001 msgs : rosvgraph_msgs/Clock
           /gazebo/link_states    11999 msgs : gazebo_msgs/LinkStates
           /odom                  11998 msgs : nav_msgs/Odometry
           /scan                   965  msgs : sensor_msgs/LaserScan

```

## See Also

BagSelection | canTransform | getTransform | readMessages | select | timeseries

**Introduced in R2015a**

# rostduration

Create a ROS duration object

## Syntax

```
dur = rostduration
dur = rostduration(totalSecs)
dur = rostduration(secs, nsecs)
```

## Description

`dur = rostduration` returns a default ROS duration object. The properties for seconds and nanoseconds are set to 0.

`dur = rostduration(totalSecs)` initializes the time values for seconds and nanoseconds based on `totalSecs`, which represents the time in seconds as a floating-point number.

`dur = rostduration(secs, nsecs)` initializes the time values for seconds and nanoseconds individually. Both inputs must be integers. Large values for `nsecs` are wrapped automatically with the remainder added to `secs`.

## Examples

### Work with ROS Duration Objects

Create ROS Duration objects, perform addition and subtraction, and compare duration objects. You can also add duration objects to ROS Time objects to get another Time object.

Create a duration using seconds and nanoseconds.

```
dur1 = rostduration(100, 2000000)
```

```
dur1 =  
  ROS Duration with properties:  
  
    Sec: 100  
    Nsec: 2000000
```

Create a duration using a floating-point value. This sets the seconds using the integer portion and nanoseconds with the remainder.

```
dur2 = rosduration(20.5)  
  
dur2 =  
  ROS Duration with properties:  
  
    Sec: 20  
    Nsec: 500000000
```

Add the two durations together to get a single duration.

```
dur3 = dur1 + dur2  
  
dur3 =  
  ROS Duration with properties:  
  
    Sec: 120  
    Nsec: 502000000
```

Subtract durations and get a negative duration. You can initialize durations with negative values as well.

```
dur4 = dur2 - dur1  
  
dur4 =  
  ROS Duration with properties:  
  
    Sec: -80  
    Nsec: 498000000
```

```
dur5 = rosduration(-1,2000000)  
  
dur5 =  
  ROS Duration with properties:
```

```
    Sec: -1
    Nsec: 2000000
```

Compare durations.

```
dur1 > dur2
ans = logical
      1
```

Add a duration to a ROS Time object.

```
time = rostime('now', 'system')
time =
  ROS Time with properties:

    Sec: 1.5515e+09
    Nsec: 22000000
```

```
timeFuture = time + dur3
```

```
timeFuture =
  ROS Time with properties:

    Sec: 1.5515e+09
    Nsec: 524000000
```

## Input Arguments

### **totalSecs** — Total time

0 (default) | scalar

Total time, specified as a floating-point scalar. The integer portion is set to the `Sec` property with the remainder applied to `Nsec` property of the `Duration` object.

### **secs** — Whole seconds

0 (default) | integer

Whole seconds, specified as an integer. This value is directly set to the `Sec` property of the `Duration` object.

---

**Note** The maximum and minimum values for `secs` are `[-2147483648, 2147483647]`.

---

### **nsecs — Nanoseconds**

0 (default) | positive integer

Nanoseconds, specified as a positive integer. This value is directly set to the `NSec` property of the `Duration` object unless it is greater than or equal to  $10^9$ . The value is then wrapped and the remainders are added to the value of `secs`.

## **Output Arguments**

### **dur — Duration**

ROS `Duration` object

Duration, returned as a ROS `Duration` object with `Sec` and `Nsec` properties.

## **See Also**

`rosmmessage` | `rostime` | `seconds`

**Introduced in R2016b**

# rosgenmsg

Generate custom messages from ROS definitions

## Syntax

```
rosgenmsg(folderpath)
```

## Description

`rosgenmsg(folderpath)` generates ROS custom messages in MATLAB by reading ROS custom message and service definitions in the specified folder path. The function expects ROS package folders inside the folder path. These packages contain the message definitions in `.msg` files and the service definitions in `.srv` files. Also, the packages require a `package.xml` file to define its contents.

After calling this function, you can send and receive your custom messages in MATLAB like all other supported messages. You can create these messages using `rosmessage` or view the list of messages by calling `rosmmsg list`.

---

**Note** You must install the Robotics System Toolbox Interface for ROS Custom Messages add-on using `roboticsAddons` to use this function.

---

## Examples

### Generate MATLAB Code for ROS Custom Messages

After you install the support package and prepare your custom message package folder, specify the folder path and call `rosgenmsg`.



```
folderpath = "C:/Users/user1/Documents/robot_custom_msg/";  
rosgenmsg(folderpath)
```

## Input Arguments

### **folderpath** — Path to ROS package folders

string scalar | character vector

Path to package folders, specified as a string scalar or character vector. These folders contain message definitions in `.msg` files and the service definitions in `.srv` files. Also, the packages require a `package.xml` file to define its contents.

## Limitations

- You must install the Robotics System Toolbox Interface for ROS Custom Messages add-on using `roboticsAddons` to use this function.

## See Also

`roboticsAddons`

## Topics

“Create Custom Messages from ROS Package”

“ROS Custom Message Support”

## External Websites

ROS Tutorials: Defining Custom Messages

ROS Tutorials: Creating a ROS `msg` and `srv`

**Introduced in R2015a**

# rosinit

Connect to ROS network

## Syntax

```
rosinit
rosinit(hostname)
rosinit(hostname,port)
rosinit(URI)
rosinit( ____,Name,Value)
```

## Description

`rosinit` starts the global ROS node with a default MATLAB name and tries to connect to a ROS master running on `localhost` and port 11311. If the global ROS node cannot connect to the ROS master, `rosinit` also starts a ROS core in MATLAB, which consists of a ROS master, a ROS parameter server, and a `rosout` logging node.

`rosinit(hostname)` tries to connect to the ROS master at the host name or IP address specified by `hostname`. This syntax uses 11311 as the default port number.

`rosinit(hostname,port)` tries to connect to the host name or IP address specified by `hostname` and the port number specified by `port`.

`rosinit(URI)` tries to connect to the ROS master at the given resource identifier, `URI`, for example, "`http://192.168.1.1:11311`".

`rosinit( ____,Name,Value)` provides additional options specified by one or more `Name,Value` pair arguments.

Using `rosinit` is a prerequisite for most ROS-related tasks in MATLAB because:

- Communicating with a ROS network requires a ROS node connected to a ROS master.
- By default, ROS functions in MATLAB operate on the global ROS node, or they operate on objects that depend on the global ROS node.

For example, after creating a global ROS node with `rosinit`, you can subscribe to a topic on the global ROS node. When another node on the ROS network publishes messages on that topic, the global ROS node receives the messages.

If a global ROS node already exists, then `rosinit` restarts the global ROS node based on the new set of arguments.

For more advanced ROS networks, connecting to multiple ROS nodes or masters is possible using the `Node` object.

## Examples

### Start ROS Core and Global Node

```
rosinit
```

```
Initializing ROS master on http://bat5742win64:50702/.  
Initializing global node /matlab_global_node_94210 with NodeURI http://bat5742win64:50702/.
```

When you are finished, shut down the ROS network.

```
roshutdown
```

```
Shutting down global node /matlab_global_node_94210 with NodeURI http://bat5742win64:50702/.  
Shutting down ROS master on http://bat5742win64:50702/.
```

### Start Node and Connect to ROS Master at Specified IP Address

```
rosinit('192.168.154.131')
```

```
Initializing global node /matlab_global_node_76850 with NodeURI http://192.168.154.1:50702/.
```

Shut down the ROS network when you are finished.

```
roshutdown
```

```
Shutting down global node /matlab_global_node_76850 with NodeURI http://192.168.154.1:50702/.
```

### Start Global Node at Given IP and NodeName

```
rosinit('192.168.154.131', 'NodeHost', '192.168.1.1', 'NodeName', '/test_node')
```

```
Initializing global node /test_node with NodeURI http://192.168.1.1:59577/
```

Shut down the ROS network when you are finished.

```
roshutdown
```

```
Shutting down global node /test_node with NodeURI http://192.168.1.1:59577/
```

## Input Arguments

### hostname — Host name or IP address

string scalar | character vector

Host name or IP address, specified as a string scalar or character vector.

### port — Port number

numeric scalar

Port number used to connect to the ROS master, specified as a numeric scalar.

### URI — URI for ROS master

string scalar | character vector

URI for ROS master, specified as a string scalar or character vector. Standard format for URIs is either `http://ipaddress:port` or `http://hostname:port`

## 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: `"NodeHost", "192.168.1.1"`

### NodeHost — Host name or IP address

character vector

Host name or IP address under which the node advertises itself to the ROS network, specified as the comma-separated pair consisting of "NodeHost" and a character vector.

Example: "comp-home"

### **NodeName — Global node name**

character vector

Global node name, specified as the comma-separated pair consisting of "NodeName" and a character vector. The node that is created through `rosinit` is registered on the ROS network with this name.

Example: "NodeName", "/test\_node"

## **See Also**

Node | `roshutdown`

## **Topics**

"Connect to a ROS Network"

**Introduced in R2015a**

# rosmessage

Create ROS messages

## Syntax

```
msg = rosmessage(msgtype)
```

```
msg = rosmessage(pub)
```

```
msg = rosmessage(sub)
```

```
msg = rosmessage(client)
```

```
msg = rosmessage(server)
```

## Description

`msg = rosmessage(msgtype)` creates an empty ROS message object with message type. The `msgtype` string scalar is case-sensitive and no partial matches are allowed. It must match a message on the list given by calling `rosmmsg("list")`.

`msg = rosmessage(pub)` creates an empty message determined by the topic published by `pub`.

`msg = rosmessage(sub)` creates an empty message determined by the subscribed topic of `sub`.

`msg = rosmessage(client)` creates an empty message determined by the service associated with `client`.

`msg = rosmessage(server)` creates an empty message determined by the service type of `server`.

## Examples

## Create Empty String Message

```
strMsg = rosmesssage('std_msgs/String')
```

```
strMsg =  
  ROS String message with properties:  
  
  MessageType: 'std_msgs/String'  
  Data: ''
```

Use `showdetails` to show the contents of the message

## Create a ROS Publisher and Send Data

Connect to a ROS network.

```
rosinit
```

```
Initializing ROS master on http://bat5742win64:64143/.  
Initializing global node /matlab_global_node_59264 with NodeURI http://bat5742win64:64143/
```

Create publisher for the `"/ chatter"` topic with the `'std_msgs/String'` message type.

```
chatpub = rospublisher('/ chatter', 'std_msgs/String');
```

Create a message to send. Specify the `Data` property.

```
msg = rosmesssage(chatpub);  
msg.Data = 'test phrase';
```

Send message via the publisher.

```
send(chatpub, msg);
```

Shutdown ROS network.

```
roshutdowndown
```

```
Shutting down global node /matlab_global_node_59264 with NodeURI http://bat5742win64:64143/.  
Shutting down ROS master on http://bat5742win64:64143/.
```

### Create and Access An Array Of ROS Messages

You can create an object array to store multiple messages. The array is indexable similar to any other array. You can modify properties of each object or access specific properties from each element using dot notation.

Create a two message object array.

```
msgArray = [rosmesssage('std_msgs/String') rosmesssage('std_msgs/String')]
msgArray =
    1x2 ROS String message array with properties:
    MessageType
    Data
```

Assign data to individual object elements of the array.

```
msgArray(1).Data = 'Some string';
msgArray(2).Data = 'Other string';
```

Read all the Data properties from the message objects into a cell array.

```
allData = {msgArray.Data}
allData = 1x2 cell array
    {'Some string'}    {'Other string'}
```

### Preallocate A ROS Message Array

To preallocate an array using ROS messages, use the `arrayfun` or `cellfun` functions instead of `repmat`. These functions create object or cell arrays for handle classes properly.

Preallocate an object array of ROS messages.

```
msgArray = arrayfun(@(~) rosmesssage('std_msgs/String'), zeros(1,50));
```

Preallocate a cell array of ROS messages.

```
msgCell = cellfun(@(~) rosmesssage('std_msgs/String'), cell(1,50), 'UniformOutput', false);
```



## Input Arguments

### **messagetype — Message type**

string scalar | character vector

Message type, specified as a string scalar or character vector. The string is case-sensitive and no partial matches are allowed. It must match a message on the list given by calling `rosmg("list")`.

### **pub — ROS publisher**

Publisher object handle

ROS publisher, specified as a Publisher object handle. You can create the object using `rospublisher`.

### **sub — ROS subscriber**

Subscriber object handle

ROS subscriber, specified as a Subscriber object handle. You can create the object using `rossubscriber`.

### **client — ROS service client**

ServiceClient object handle

ROS service client, specified as a ServiceClient object handle. You can create the object using `rossvcclient`.

### **server — ROS service server**

ServiceServer object handle

ROS service server, specified as a ServiceServer object handle. You can create the object using `rossvcserver`.

## Output Arguments

### **msg — ROS message**

Message object handle

ROS message, returned as a Message object handle.

## **See Also**

roboticsAddons | rosmmsg | rostopic

## **Topics**

“Work with Basic ROS Messages”

“Built-In Message Support”

**Introduced in R2015a**

## rosmg

Retrieve information about ROS messages and message types

### Syntax

```
rosmg show msgtype  
rosmg md5 msgtype  
rosmg list
```

```
msginfo = rosmg("show", msgtype)  
msgmd5 = rosmg("md5", msgtype)  
msglist = rosmg("list")
```

### Description

`rosmg show msgtype` returns the definition of the `msgtype` message.

`rosmg md5 msgtype` returns the MD5 checksum of the `msgtype` message.

`rosmg list` returns all available message types that you can use in MATLAB.

`msginfo = rosmg("show", msgtype)` returns the definition of the `msgtype` message as a character vector.

`msgmd5 = rosmg("md5", msgtype)` returns the 'MD5' checksum of the `msgtype` message as a character vector.

`msglist = rosmg("list")` returns a cell array containing all available message types that you can use in MATLAB.

### Examples

### Retrieve Message Type Definition

```
msgInfo = rosmg('show', 'geometry_msgs/Point')

msgInfo =
    '% This contains the position of a point in free space
    double X
    double Y
    double Z
    '
```

### Get the MD5 Checksum of Message Type

```
msgMd5 = rosmg('md5', 'geometry_msgs/Point')

msgMd5 =
    '4a842b65f413084dc2b10fb484ea7f17'
```

## Input Arguments

### **msgtype** — ROS message type

character vector

ROS message type, specified as a character vector. `msgType` must be a valid ROS message type from ROS that MATLAB supports.

Example: "std\_msgs/Int8"

## Output Arguments

### **msginfo** — Details of message definition

character vector

Details of the information inside the ROS message definition, returned as a character vector.

**msgmd5 — MD5 checksum hash value**

character vector

MD5 checksum hash value, returned as a character vector. The MD5 output is a character vector representation of the 16-byte hash value that follows the MD5 standard.

**msglist — List of all message types available in MATLAB**

cell array of character vectors

List of all message types available in MATLAB, returned as a cell array of character vectors.

**Introduced in R2015a**

# roscnode

Retrieve information about ROS network nodes

## Syntax

```
roscnode list
roscnode info nodename
roscnode ping nodename

nodelist = roscnode("list")
nodeinfo = roscnode("info",nodename)
roscnode("ping",nodename)
```

## Description

`roscnode list` returns a list of all nodes registered on the ROS network. Use these nodes to exchange data between MATLAB and the ROS network.

`roscnode info nodename` returns a structure containing the name, URI, publications, subscriptions, and services of a specific ROS node, `nodename`.

`roscnode ping nodename` pings a specific node, `nodename`, and displays the response time.

`nodelist = roscnode("list")` returns a cell array of character vectors containing the nodes registered on the ROS network.

`nodeinfo = roscnode("info",nodename)` returns a structure containing the name, URI, publications, subscriptions, and services of a specific ROS node, `nodename`.

`roscnode("ping",nodename)` pings a specific node, `nodename` and displays the response time.

## Examples

## Retrieve List of ROS Nodes

**Note:** This example requires a valid ROS network to be active with ROS nodes previously set up.

Connect to the ROS network. Specify the IP address for your specific network.

```
rosinit('192.168.203.129')
```

```
Initializing global node /matlab_global_node_90274 with NodeURI http://192.168.203.1:63
```

List the nodes available from the ROS master.

```
rosnode list
```

```
/bumper2pointcloud  
/cmd_vel_mux  
/depthimage_to_laserscan  
/gazebo  
/laserscan_nodelet_manager  
/matlab_global_node_90274  
/mobile_base_nodelet_manager  
/robot_state_publisher  
/rosout
```

Shut down the ROS network.

```
roshutdown
```

```
Shutting down global node /matlab_global_node_90274 with NodeURI http://192.168.203.1:63
```

## Retrieve ROS Node Information

Connect to the ROS network. Specify the IP address for your specific network.

```
rosinit('192.168.154.131')
```

```
Initializing global node /matlab_global_node_40513 with NodeURI http://192.168.154.1:63
```

Get information on the '/robot\_state\_publisher' node. This node is available on the ROS master.

```
nodeinfo = rosnode('info', '/robot_state_publisher')
```

```
nodeinfo =  
  
    struct with fields:  
  
        NodeName: '/robot_state_publisher'  
        URI: 'http://192.168.154.131:40244/'  
        Publications: [2×1 struct]  
        Subscriptions: [2×1 struct]  
        Services: [2×1 struct]
```

Shut down the ROS network.

```
roshutdown
```

```
Shutting down global node /matlab_global_node_40513 with NodeURI http://192.168.154.1:0
```

### **Ping ROS Node**

Connect to the ROS network. Specify the IP address for your specific network.

```
rosinit('192.168.154.131')
```

```
Initializing global node /matlab_global_node_88195 with NodeURI http://192.168.154.1:50
```

Ping the '/robot\_state\_publisher' node. This node is available on the ROS master.

```
nodeinfo = rosnode('info', '/robot_state_publisher')
```

```
nodeinfo =  
  
    struct with fields:  
  
        NodeName: '/robot_state_publisher'  
        URI: 'http://192.168.154.131:40244/'  
        Publications: [2×1 struct]  
        Subscriptions: [2×1 struct]  
        Services: [2×1 struct]
```

Shut down the ROS network.



roshutdown

```
Shutting down global node /matlab_global_node_88195 with NodeURI http://192.168.154.1:5
```

## Input Arguments

**nodename** — Name of node

string scalar | character vector

Name of node, specified as a string scalar or character vector. The name of the node must match the name given in ROS.

## Output Arguments

**nodeinfo** — Information about ROS node

structure

Information about ROS node, returned as a structure containing these fields: `NodeName`, `URI`, `Publications`, `Subscriptions`, and `Services`. Access these properties using dot syntax, for example, `nodeinfo.NodeName`.

**odelist** — List of node names available

cell array of character vectors

List of node names available, returned as a cell array of character vectors.

## See Also

`rosinit` | `rostopic`

**Introduced in R2015a**

# rosparam

Access ROS parameter server values

## Syntax

```
list = rosparam("list")
list = rosparam("list", namespace)
pvalOut = rosparam("get", pname)
pvalOut = rosparam("get", namespace)
rosparam("set", pname, pval)
rosparam("delete", pname)
rosparam("delete", namespace)
```

```
ptree = rosparam
```

## Description

`list = rosparam("list")` returns the list of all ROS parameter names from the ROS master.

**Simplified form:** `rosparam list`

`list = rosparam("list", namespace)` returns the list of all parameter names under the specified ROS namespace.

**Simplified form:** `rosparam list namespace`

`pvalOut = rosparam("get", pname)` retrieves the value of the specified parameter.

**Simplified form:** `rosparam get pname`

`pvalOut = rosparam("get", namespace)` retrieves the values of all parameters under the specified namespace as a structure.

**Simplified form:** `rosparam get namespace`

`rosparam("set", pname, pval)` sets a value for a specified parameter name. If the parameter name does not exist, the function adds a new parameter in the parameter tree.

**Simplified form:** `rosparam set pname pval`

See “Limitations” on page 2-401 for limitations on `pval`.

`rosparam("delete", pname)` deletes a parameter from the parameter tree. If the parameter does not exist, the function displays an error.

**Simplified form:** `rosparam delete pname`

`rosparam("delete", namespace)` deletes all parameters under the given namespace from the parameter tree.

**Simplified form:** `rosparam delete namespace`

`ptree = rosparam` creates a parameter tree object, `ptree`. After `ptree` is created, the connection to the parameter server remains persistent until the object is deleted or the ROS master becomes unavailable.

A ROS parameter tree communicates with the ROS parameter server. The ROS parameter server can store strings, integers, doubles, booleans and cell arrays. The parameters are accessible by every node in the ROS network. Use the parameters to store static data such as configuration parameters. Use the `get`, `set`, `has`, `search`, and `del` functions to manipulate and view parameter values.

The following ROS data types are supported as values of parameters. For each ROS data type, the corresponding MATLAB data type is also listed.

- 32-bit integer — `int32`
- boolean — `logical`
- double — `double`
- string — character vector (`char`)
- list — cell array (`cell`)
- dictionary — structure (`struct`)

## Examples

### Get and Set Parameter Values

Connect to a ROS network to set and get ROS parameter values on the ROS parameter tree. You can get lists of parameters in their given namespaces as well. This example uses the simplified form that mimics the ROS command-line interface.

Connect to a ROS network.

```
rosinit
```

```
Initializing ROS master on http://bat5742win64:64564/.  
Initializing global node /matlab_global_node_82637 with NodeURI http://bat5742win64:64564/
```

Set parameter values.

```
rosparam set /string_param 'param_value'  
rosparam set /double_param 1.2
```

To set a list parameter, use the functional form.

```
rosparam('set', '/list_param', {int32(5), 124.1, -20, 'some_string'});
```

Get the list of parameters using the command-line form.

```
rosparam list  
  
/double_param  
/list_param  
/string_param
```

List parameters in a specific namespace.

```
rosparam list /double  
  
/double_param
```

Get the value of a parameter.

```
rosparam get /list_param  
  
{5, 124.1, -20, some_string}
```

Delete a parameter. List the parameters to verify it was deleted.

```
rosparam delete /double_param  
rosparam list
```

```
/list_param
/string_param
```

Shut down the ROS network.

```
roshutdown
```

```
Shutting down global node /matlab_global_node_82637 with NodeURI http://bat5742win64:64564/
Shutting down ROS master on http://bat5742win64:64564/.
```

### Create Parameter Tree Object and View Parameters

Connect to the ROS network. ROS parameters should already be available on the ROS master.

```
roscpp('192.168.154.131')
```

```
Initializing global node /matlab_global_node_82870 with NodeURI http://192.168.154.1:50053
```

Create a ParameterTree object using rosparam.

```
ptree = rosparam;
```

List the available parameters on the ROS master.

```
ptree.AvailableParameters
```

```
ans =
```

```
33x1 cell array
```

```
'/bumper2pointcloud/pointcloud_radius'
'/camera/imager_rate'
'/camera/rgb/image_raw/compressed/format'
'/camera/rgb/image_raw/compressed/jpeg_quality'
'/camera/rgb/image_raw/compressed/png_level'
'/cmd_vel_mux/yaml_cfg_file'
'/depthimage_to_laserscan/output_frame_id'
'/depthimage_to_laserscan/range_max'
'/depthimage_to_laserscan/range_min'
'/depthimage_to_laserscan/scan_height'
'/depthimage_to_laserscan/scan_time'
```

```
'/gazebo/auto_disable_bodies'  
'/gazebo/cfm'  
'/gazebo/contact_max_correcting_vel'  
'/gazebo/contact_surface_layer'  
'/gazebo/erp'  
'/gazebo/gravity_x'  
'/gazebo/gravity_y'  
'/gazebo/gravity_z'  
'/gazebo/max_contacts'  
'/gazebo/max_update_rate'  
'/gazebo/sor_pgs_iters'  
'/gazebo/sor_pgs_precon_iters'  
'/gazebo/sor_pgs_rms_error_tol'  
'/gazebo/sor_pgs_w'  
'/gazebo/time_step'  
'/robot_description'  
'/robot_state_publisher/publish_frequency'  
'/roscdistro'  
'/roslaunch/uris/host_192_168_154_131__41131'  
'/rosversion'  
'/run_id'  
'/use_sim_time'
```

Shut down the ROS network.

```
roshutdown
```

```
Shutting down global node /matlab_global_node_82870 with NodeURI http://192.168.154.1:5
```

### Set A Dictionary Of Parameter Values

Use structures to specify a dictionary of ROS parameters under a specific namespace.

Connect to a ROS network.

```
roscinit
```

```
Initializing ROS master on http://bat5742win64:64232/.
```

```
Initializing global node /matlab_global_node_67361 with NodeURI http://bat5742win64:64
```

Create a dictionary of parameters values. This dictionary contains the information relevant to an image. Display the structure to verify values.

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

pval.ImageWidth = size(image,1);
pval.ImageHeight = size(image,2);
pval.ImageTitle = 'peppers.png';
disp(pval)

    ImageWidth: 384
    ImageHeight: 512
    ImageTitle: 'peppers.png'
```

Set the dictionary of values using the desired namespace.

```
rosparam('set', 'ImageParam', pval)
```

Get the parameters using the namespace. Verify the values.

```
pval2 = rosparam('get', 'ImageParam')

pval2 = struct with fields:
    ImageHeight: 512
    ImageTitle: 'peppers.png'
    ImageWidth: 384
```

Shutdown ROS network.

```
rosshutdown
```

```
Shutting down global node /matlab_global_node_67361 with NodeURI http://bat5742win64:644232/
Shutting down ROS master on http://bat5742win64:64232/.
```

## Input Arguments

### **namespace** — ROS parameter namespace

string scalar | character vector

ROS parameter namespace, specified as a string scalar or character vector. All parameter names starting with this string are listed when calling `rosparam("list", namespace)`.

### **pname** — ROS parameter name

string scalar | character vector

ROS parameter name, specified as a string scalar or character vector.

### **pval** — ROS parameter value or dictionary of values

`int32` | `logical` | `double` | string scalar | character vector | cell array | structure

ROS parameter value or dictionary of values, specified as a supported data type.

The following ROS data types are supported as values of parameters. For each ROS data type, the corresponding MATLAB data type is also listed:

- 32-bit integers — `int32`
- booleans — `logical`
- doubles — `double`
- strings — string scalar, `string`, or character vector, `char`
- lists — cell array
- dictionaries — structure

## Output Arguments

### **list** — Parameter list

cell array of character vectors

Parameter list, returned as a cell array of character vectors. This is a list of all parameters available on the ROS master.

### **ptree** — Parameter tree

`ParameterTree` object handle

Parameter tree, returned as a `ParameterTree` object handle. Use this object to reference parameter information, for example, `ptree.AvailableFrames`.

### **pvalOut** — ROS parameter value or dictionary of values

`int32` | `logical` | `double` | character vector | cell array | structure

ROS parameter value, specified as a supported MATLAB data type. When specifying the namespace input argument, `pvalOut` is returned as a structure of parameter value under the given namespace.



The following ROS data types are supported as values of parameters. For each ROS data type, the corresponding MATLAB data type is also listed.

ROS Data Type	MATLAB Data Type
32-bit integer	<code>int32</code>
boolean	<code>logical</code>
double	<code>double</code>
string	character vector ( <code>char</code> )
list	cell array ( <code>cell</code> )
dictionary	structure ( <code>struct</code> )

## Limitations

- **Unsupported Data Types:** Base64-encoded binary data and iso8601 data from ROS are not supported.
- **Simplified Commands:** When using the simplified command `rosparam set pname pval`, the parameter value is interpreted as:
  - `logical` — If `pval` is "true" or "false"
  - `int32` — If `pval` is an integer, for example, 5
  - `double` — If `pval` is a fractional number, for example, 1.256
  - `character vector` — If `pval` is any other value

## See Also

### Functions

`del` | `get` | `has` | `search` | `set`

### Objects

`ParameterTree`

**Introduced in R2015a**

# rosservice

Retrieve information about services in ROS network

## Syntax

```
rosservice list
rosservice info svcname
rosservice type svcname
rosservice uri svcname
```

```
svclist = rosservice("list")
svcinfo = rosservice("info",svcname)
svctype = rosservice("type",svcname)
svcuri = rosservice("uri",svcname)
```

## Description

`rosservice list` returns a list of service names for all of the active service servers on the ROS network.

`rosservice info svcname` returns information about the specified service, `svcname`.

`rosservice type svcname` returns the service type.

`rosservice uri svcname` returns the URI of the service.

`svclist = rosservice("list")` returns a list of service names for all of the active service servers on the ROS network. `svclist` contains a cell array of service names.

`svcinfo = rosservice("info",svcname)` returns a structure of information, `svcinfo`, about the service, `svcname`.

`svctype = rosservice("type",svcname)` returns the service type of the service as a character vector.

`svcuri = rosservice("uri",svcname)` returns the URI of the service as a character vector.

## Examples

### View List of ROS Services

Connect to the ROS network. Specify the IP address of your specific network.

```
rosinit('192.168.154.131')
```

```
Initializing global node /matlab_global_node_00003 with NodeURI http://192.168.154.1:5
```

List the services available on the ROS master.

```
rosservice list
```

```
/camera/rgb/image_raw/compressed/set_parameters  
/camera/set_camera_info  
/camera/set_parameters  
/gazebo/apply_body_wrench  
/gazebo/apply_joint_effort  
/gazebo/clear_body_wrenches  
/gazebo/clear_joint_forces  
/gazebo/delete_model  
/gazebo/get_joint_properties  
/gazebo/get_link_properties  
/gazebo/get_link_state  
/gazebo/get_loggers  
/gazebo/get_model_properties  
/gazebo/get_model_state  
/gazebo/get_physics_properties  
/gazebo/get_world_properties  
/gazebo/pause_physics  
/gazebo/reset_simulation  
/gazebo/reset_world  
/gazebo/set_joint_properties  
/gazebo/set_link_properties  
/gazebo/set_link_state  
/gazebo/set_logger_level  
/gazebo/set_model_configuration  
/gazebo/set_model_state  
/gazebo/set_parameters  
/gazebo/set_physics_properties  
/gazebo/spawn_gazebo_model  
/gazebo/spawn_sdf_model
```

```
/gazebo/spawn_urdf_model
/gazebo/unpause_physics
/laserscan_nodelet_manager/get_loggers
/laserscan_nodelet_manager/list
/laserscan_nodelet_manager/load_nodelet
/laserscan_nodelet_manager/set_logger_level
/laserscan_nodelet_manager/unload_nodelet
/mobile_base_nodelet_manager/get_loggers
/mobile_base_nodelet_manager/list
/mobile_base_nodelet_manager/load_nodelet
/mobile_base_nodelet_manager/set_logger_level
/mobile_base_nodelet_manager/unload_nodelet
/robot_state_publisher/get_loggers
/robot_state_publisher/set_logger_level
/rosout/get_loggers
/rosout/set_logger_level
```

Shut down the ROS network.

```
roshutdown
```

```
Shutting down global node /matlab_global_node_00003 with NodeURI http://192.168.154.1:5
```

### Get Information, Service Type, and URI for ROS Service

Connect to the ROS network. Specify the IP address of your specific network.

```
rosinit('192.168.154.131')
```

```
Initializing global node /matlab_global_node_76389 with NodeURI http://192.168.154.1:5
```

Get information on the `/gazebo/pause_physics` service.

```
svcinfo = rosservice('info', 'gazebo/pause_physics')
```

```
svcinfo =
```

```
struct with fields:
```

```
Node: '/gazebo'
URI: 'rosrpc://192.168.154.131:33260'
Type: 'std_srvs/Empty'
```

```
Args: {}
```

Get the service type.

```
svctype = rosservice('type', 'gazebo/pause_physics')
```

```
svctype =
```

```
    'std_srvs/Empty'
```

Get the service URI.

```
svcuri = rosservice('uri', 'gazebo/pause_physics')
```

```
svcuri =
```

```
    'rosrpc://192.168.154.131:33260'
```

Shut down the ROS network.

```
roshutdown
```

```
Shutting down global node /matlab_global_node_76389 with NodeURI http://192.168.154.1:5
```

## Input Arguments

**svcname** — Name of service

string scalar | character vector

Name of service, specified as a string scalar or character vector. The service name must match its name in the ROS network.

## Output Arguments

**svcinfo** — Information about a ROS service

character vector

Information about a ROS service, returned as a character vector.

**svclist — List of available ROS services**

cell array of character vectors

List of available ROS services, returned as a cell array of character vectors.

**svctype — Type of ROS service**

character vector

Type of ROS service, returned as a character vector.

**svcuri — URI for accessing service**

character vector

URI for accessing service, returned as a character vector.

## See Also

rosinit | rosparam

**Introduced in R2015a**

# rosshutdown

Shut down ROS system

## Syntax

```
rosshutdown
```

## Description

`rosshutdown` shuts down the global node and, if it is running, the ROS master. When you finish working with the ROS network, use `rosshutdown` to shut down the global ROS entities created by `rosinit`. If the global node and ROS master are not running, this function has no effect.

---

**Note** After calling `rosshutdown`, any ROS entities (objects) that depend on the global node like subscribers created with `rossubscriber`, are deleted and become unstable.

Prior to calling `rosshutdown`, call `clear` on these objects for a clean removal of ROS entities.

---

## Examples

### Start ROS Core and Global Node

```
rosinit
```

```
Initializing ROS master on http://bat5742win64:50702/.
```

```
Initializing global node /matlab_global_node_94210 with NodeURI http://bat5742win64:50702/
```

When you are finished, shut down the ROS network.

```
rosshutdown
```

```
Shutting down global node /matlab_global_node_94210 with NodeURI http://bat5742win64:50702/
Shutting down ROS master on http://bat5742win64:50702/.
```

### **See Also**

rosinit

**Introduced in R2015a**



# rostopic

Retrieve information about ROS topics

## Syntax

```
rostopic list
rostopic echo topicname
rostopic info topicname
rostopic type topicname
```

```
topiclist = rostopic("list")
msg = rostopic("echo", topicname)
topicinfo = rostopic("info", topicname)
msgtype = rostopic("type", topicname)
```

## Description

`rostopic list` returns a list of ROS topics from the ROS master.

`rostopic echo topicname` returns the messages being sent from the ROS master about a specific topic, `topicname`. To stop returning messages, press **Ctrl+C**.

`rostopic info topicname` returns the message type, publishers, and subscribers for a specific topic, `topicname`.

`rostopic type topicname` returns the message type for a specific topic.

`topiclist = rostopic("list")` returns a cell array containing the ROS topics from the ROS master. If you do not define the output argument, the list is returned in the MATLAB Command Window.

`msg = rostopic("echo", topicname)` returns the messages being sent from the ROS master about a specific topic, `topicname`. To stop returning messages, press **Ctrl+C**. If the output argument is defined, then `rostopic` returns the first message that arrives on that topic.

`topicinfo = rostopic("info", topicname)` returns a structure containing the message type, publishers, and subscribers for a specific topic, `topicname`.

`msgtype = rostopic("type", topicname)` returns a character vector containing the message type for the specified topic, `topicname`.

## Examples

### Get List of ROS Topics

Connect to ROS network. Specify the IP address of the ROS device.

```
rosinit('192.168.154.131')
```

```
Initializing global node /matlab_global_node_23844 with NodeURI http://192.168.154.1:5005
```

List the ROS topic available on the ROS master.

```
rostopic list
```

```
/camera/depth/camera_info
/camera/depth/image_raw
/camera/depth/points
/camera/parameter_descriptions
/camera/parameter_updates
/camera/rgb/camera_info
/camera/rgb/image_raw
/camera/rgb/image_raw/compressed
/camera/rgb/image_raw/compressed/parameter_descriptions
/camera/rgb/image_raw/compressed/parameter_updates
/clock
/cmd_vel_mux/active
/cmd_vel_mux/input/navi
/cmd_vel_mux/input/safety_controller
/cmd_vel_mux/input/teleop
/cmd_vel_mux/parameter_descriptions
/cmd_vel_mux/parameter_updates
/depthimage_to_laserscan/parameter_descriptions
/depthimage_to_laserscan/parameter_updates
/gazebo/link_states
/gazebo/model_states
/gazebo/parameter_descriptions
```

```
/gazebo/parameter_updates
/gazebo/set_link_state
/gazebo/set_model_state
/joint_states
/laserscan_nodelet_manager/bond
/mobile_base/commands/motor_power
/mobile_base/commands/reset_odometry
/mobile_base/commands/velocity
/mobile_base/events/bumper
/mobile_base/events/cliff
/mobile_base/sensors/bumper_pointcloud
/mobile_base/sensors/core
/mobile_base/sensors/imu_data
/mobile_base_nodelet_manager/bond
/odom
/rosout
/rosout_agg
/scan
/tf
```

### Get ROS Topic Info

Connect to ROS network. Specify the IP address of the ROS device.

```
roscppinit('192.168.154.131')
```

```
Initializing global node /matlab_global_node_28473 with NodeURI http://192.168.154.1:5
```

Show info on a specific ROS topic.

```
rostopic info camera/depth/points
```

```
Type: sensor_msgs/PointCloud2
```

```
Publishers:
```

```
* /gazebo http://192.168.154.131:46957/
```

```
Subscribers:
```

### Get ROS Topic Message Type

Connect to ROS network. Specify the IP address of the ROS device.

```
rosinit('192.168.154.131')
```

Initializing global node /matlab\_global\_node\_70141 with NodeURI http://192.168.154.1:5

Get message type for a specific topic. Create a message from the message type to publish to the topic.

```
msgtype = rostopic('type', 'camera/depth/points');  
msg = rosmessage(msgtype);
```

## Input Arguments

**topicname** — ROS topic name

string scalar | character vector

ROS topic name, specified as a string scalar or character vector. The topic name must match one of the topics that `rostopic("list")` outputs.

## Output Arguments

**topiclist** — List of topics from the ROS master

cell array of character vectors

List of topics from ROS master, returned as a cell array of character vectors.

**msg** — ROS message for a given topic

object handle

ROS message for a given topic, returned as an object handle.

**topicinfo** — Information about a given ROS topic

structure

Information about a ROS topic, returned as a structure. `topicinfo` included the message type, publishers, and subscribers associated with that topic.

**msgtype** — Message type for a ROS topic

character vector

Message type for a ROS topic, returned as a character vector.

**Introduced in R2015a**

# rostype

Access available ROS message types

## Syntax

```
rostype
```

## Description

`rostype` creates a blank message of a certain type by browsing the list of available message types. You can use tab completion and do not have to rely on typing error-free message type character vectors. By typing `rostype.partialname`, and pressing **Tab**, a list of matching message types appears in a list. By setting the message type equal to a variable, you can create a character vector of that message type. Alternatively, you can create the message by supplying the message type directly into `rosmmessage` as an input argument.

## Examples

### Create ROS Message Type and ROS Message

Create Message Type String

```
t = rostype.std_msgs_String
```

```
t =  
'std_msgs/String'
```

Create ROS Message from ROS Type

```
msg = rosmmessage(rostype.std_msgs_String)
```

```
msg =  
    ROS String message with properties:
```

```
MessageType: 'std_msgs/String'  
Data: ''
```

Use `showdetails` to show the contents of the message

## See Also

`rosmmessage` | `rostopic`

## Topics

“Built-In Message Support”

“Work with Basic ROS Messages”

**Introduced in R2015a**

## rotateframe

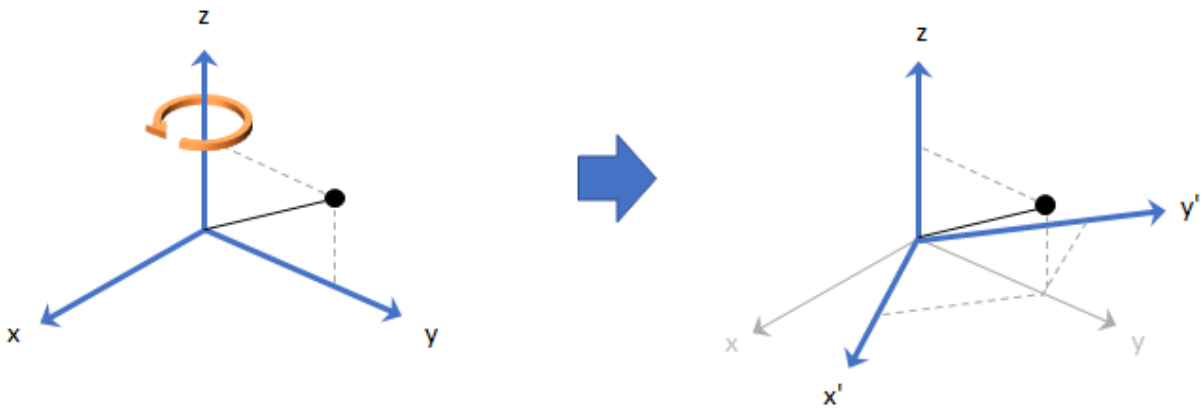
Quaternion frame rotation

### Syntax

```
rotationResult = rotateframe(quat, cartesianPoints)
```

### Description

`rotationResult = rotateframe(quat, cartesianPoints)` rotates the frame of reference for the Cartesian points using the quaternion, `quat`. The elements of the quaternion are normalized before use in the rotation.



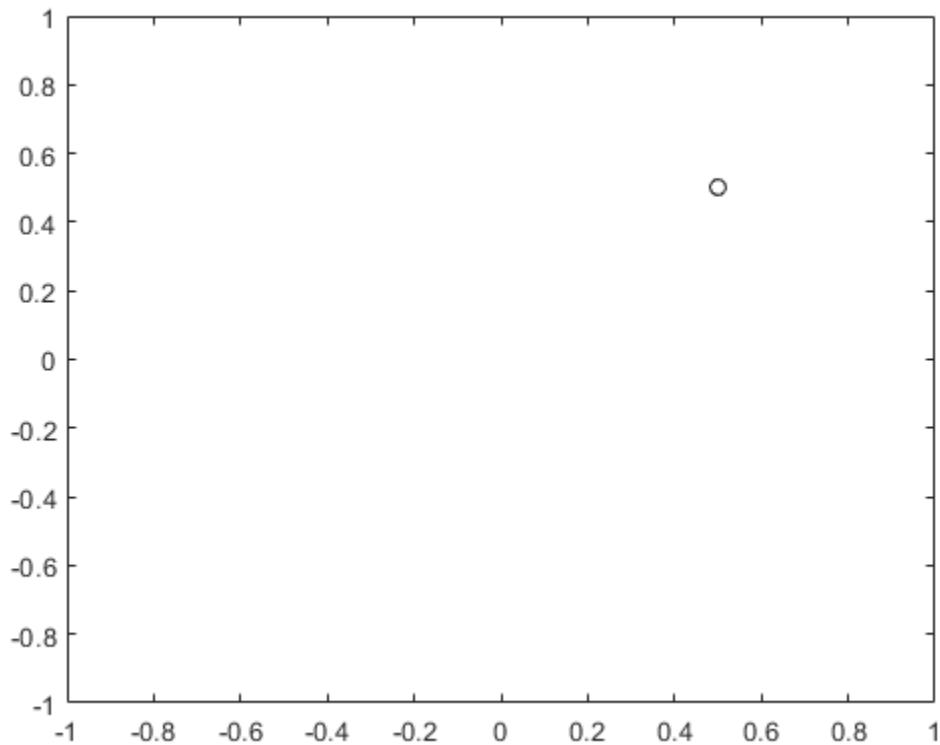
### Examples



## Rotate Frame Using Quaternion Vector

Define a point in three dimensions. The coordinates of a point are always specified in the order  $x$ ,  $y$ , and  $z$ . For convenient visualization, define the point on the  $x$ - $y$  plane.

```
x = 0.5;  
y = 0.5;  
z = 0;  
plot(x,y, 'ko')  
hold on  
axis([-1 1 -1 1])
```



Create a quaternion vector specifying two separate rotations, one to rotate the frame 45 degrees and another to rotate the point -90 degrees about the z-axis. Use `rotateframe` to perform the rotations.

```
quat = quaternion([0,0,pi/4; ...  
                 0,0,-pi/2], 'euler', 'XYZ', 'frame');
```

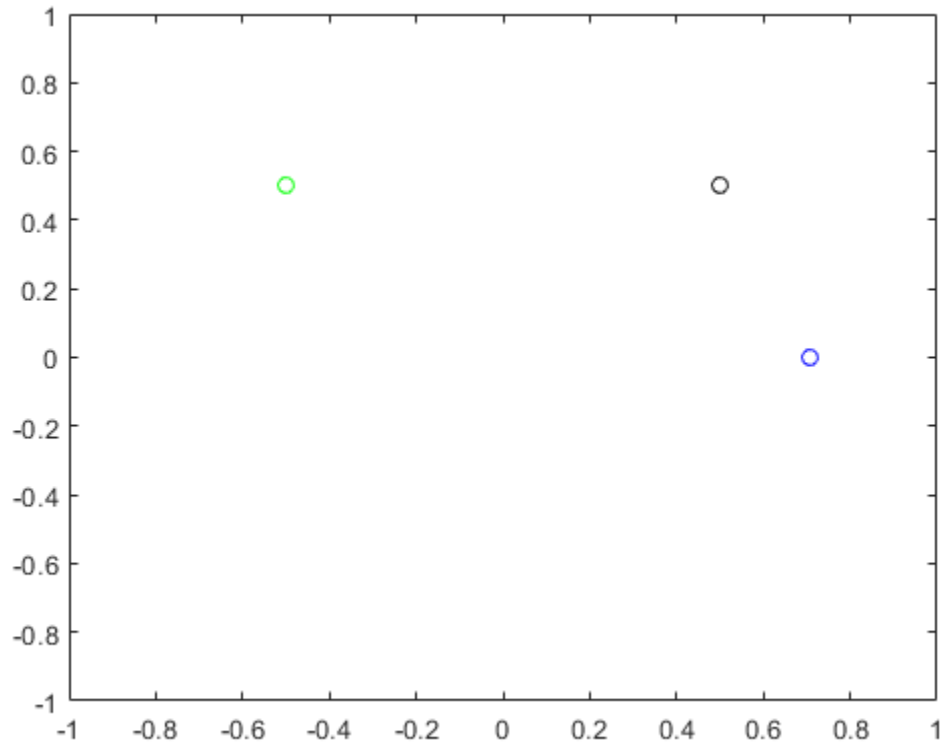
```
rereferencedPoint = rotateframe(quat,[x,y,z])
```

```
rereferencedPoint = 2×3
```

```
    0.7071    -0.0000         0  
   -0.5000     0.5000         0
```

Plot the rereferenced points.

```
plot(rereferencedPoint(1,1),rereferencedPoint(1,2),'bo')  
plot(rereferencedPoint(2,1),rereferencedPoint(2,2),'go')
```



### Rereference Group of Points using Quaternion

Define two points in three-dimensional space. Define a quaternion to rereference the points by first rotating the reference frame about the z-axis 30 degrees and then about the new y-axis 45 degrees.

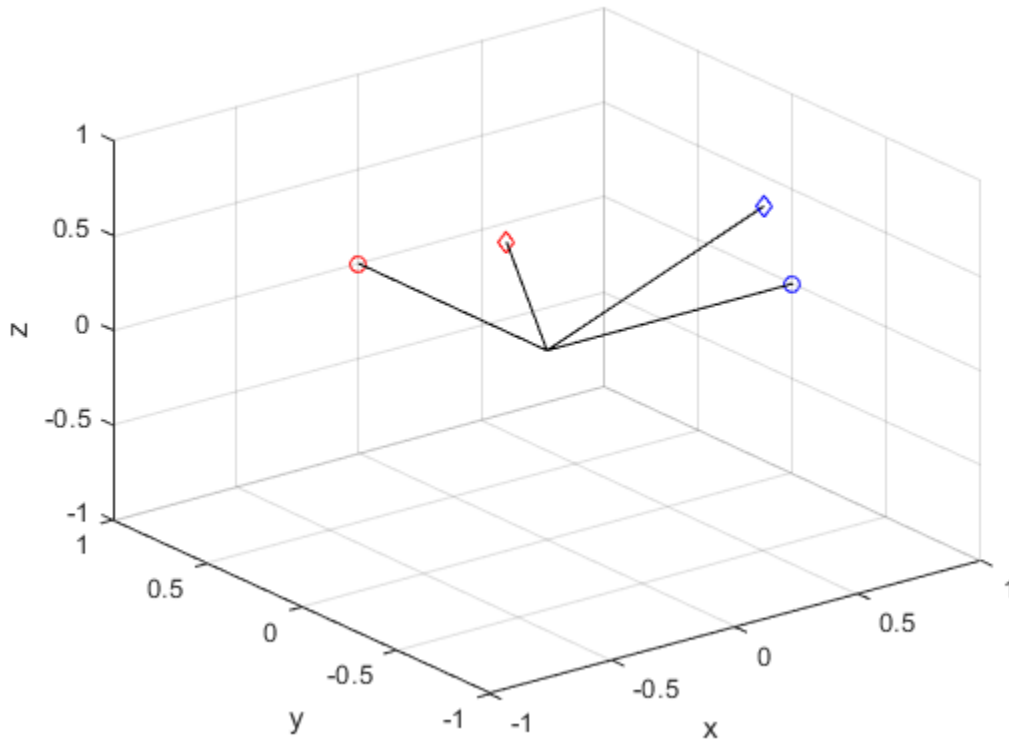
```
a = [1,0,0];  
b = [0,1,0];  
quat = quaternion([30,45,0], 'eulerd', 'ZYX', 'point');
```

Use `rotateframe` to reference both points using the quaternion rotation operator. Display the result.

```
rP = rotateframe(quat,[a;b])  
  
rP = 2×3  
  
    0.6124    -0.3536    0.7071  
    0.5000    0.8660   -0.0000
```

Visualize the original orientation and the rotated orientation of the points. Draw lines from the origin to each of the points for visualization purposes.

```
plot3(a(1),a(2),a(3), 'bo');  
  
hold on  
grid on  
axis([-1 1 -1 1 -1 1])  
xlabel('x')  
ylabel('y')  
zlabel('z')  
  
plot3(b(1),b(2),b(3), 'ro');  
plot3(rP(1,1),rP(1,2),rP(1,3), 'bd')  
plot3(rP(2,1),rP(2,2),rP(2,3), 'rd')  
  
plot3([0;rP(1,1)],[0;rP(1,2)],[0;rP(1,3)], 'k')  
plot3([0;rP(2,1)],[0;rP(2,2)],[0;rP(2,3)], 'k')  
plot3([0;a(1)],[0;a(2)],[0;a(3)], 'k')  
plot3([0;b(1)],[0;b(2)],[0;b(3)], 'k')
```



## Input Arguments

### **quat** — Quaternion that defines rotation

scalar | vector

Quaternion that defines rotation, specified as a scalar quaternion or vector of quaternions.

Data Types: quaternion

### **cartesianPoints** — Three-dimensional Cartesian points

1-by-3 vector |  $N$ -by-3 matrix

Three-dimensional Cartesian points, specified as a 1-by-3 vector or  $N$ -by-3 matrix.

Data Types: `single` | `double`

## Output Arguments

### **rotationResult** — Re-referenced Cartesian points

vector | matrix

Cartesian points defined in reference to rotated reference frame, returned as a vector or matrix the same size as `cartesianPoints`.

The data type of the re-referenced Cartesian points is the same as the underlying data type of `quat`.

Data Types: `single` | `double`

## Algorithms

Quaternion frame rotation re-references a point specified in  $\mathbf{R}^3$  by rotating the original frame of reference according to a specified quaternion:

$$L_q(u) = q*uq$$

where  $q$  is the quaternion,  $*$  represents conjugation, and  $u$  is the point to rotate, specified as a quaternion.

For convenience, the `rotateframe` function takes a point in  $\mathbf{R}^3$  and returns a point in  $\mathbf{R}^3$ . Given a function call with some arbitrary quaternion,  $q = a + bi + cj + dk$ , 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:

$$v_q = q^* u_q q$$

- 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™.

**Introduced in R2018a**

## rotatepoint

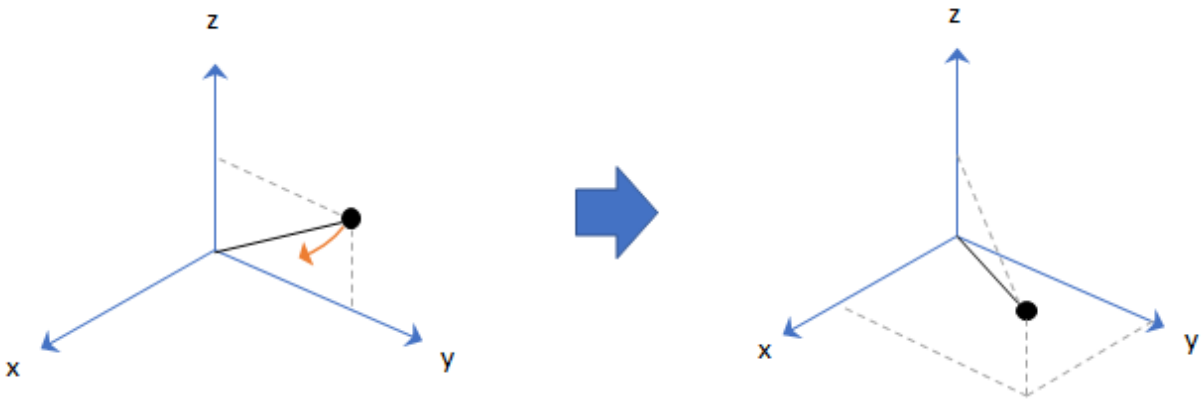
Quaternion point rotation

### 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

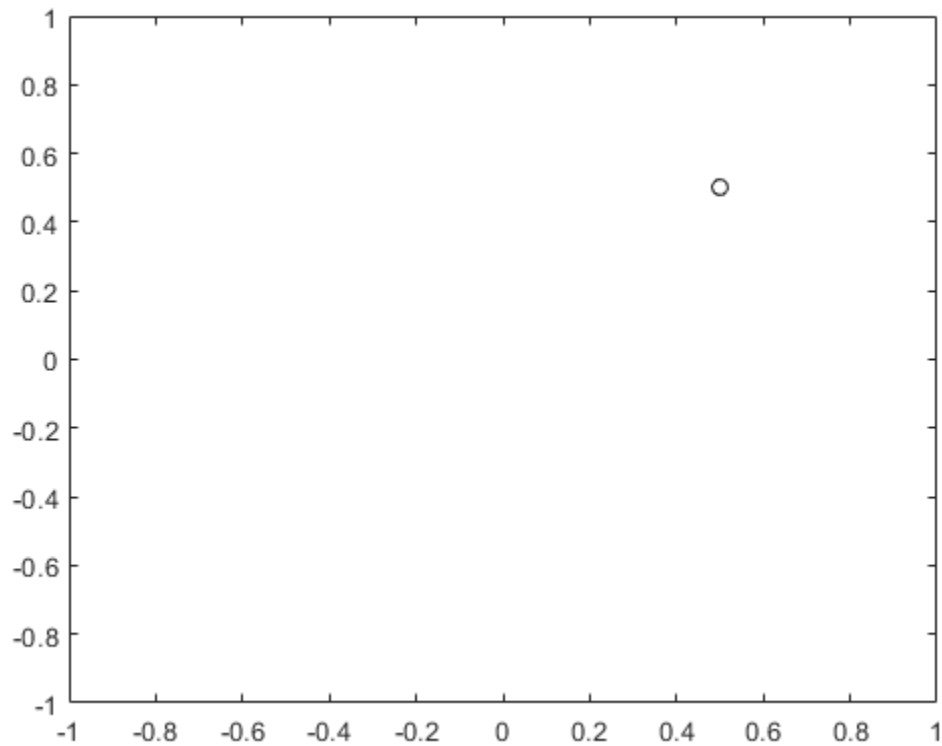
#### Rotate Point Using Quaternion Vector

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 degrees and another to rotate the point -90 degrees about the z-axis. Use `rotatepoint` to perform the rotation.

```
quat = quaternion([0,0,pi/4; ...
                  0,0,-pi/2], 'euler', 'XYZ', 'point');
```

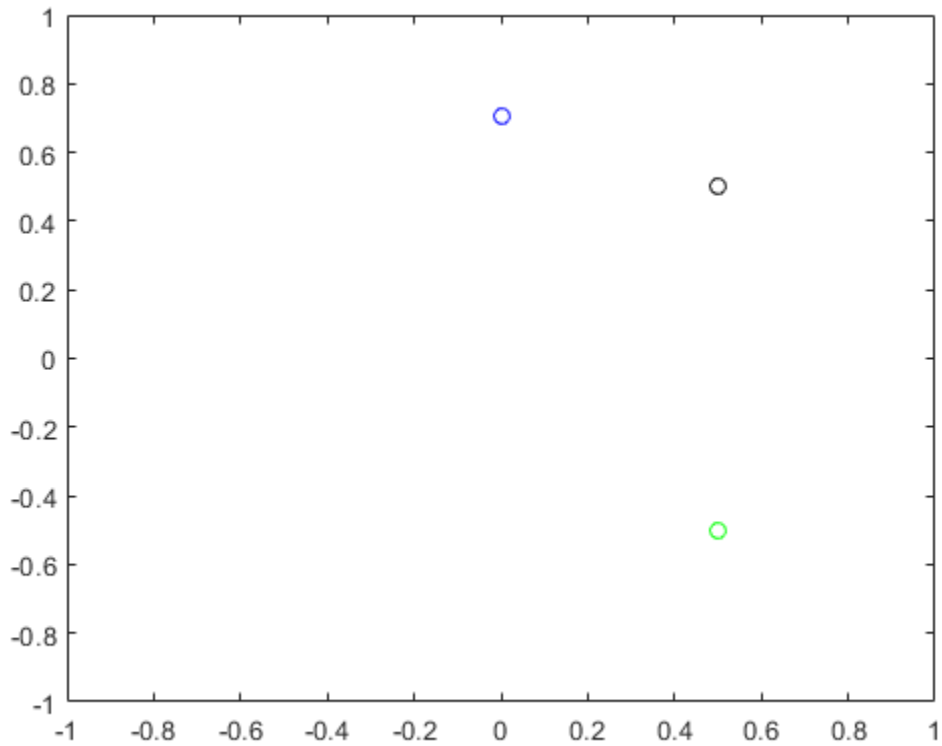
```
rotatedPoint = rotatepoint(quat,[x,y,z])
```

```
rotatedPoint = 2×3
```

```
-0.0000    0.7071    0  
 0.5000   -0.5000    0
```

Plot the rotated points.

```
plot(rotatedPoint(1,1),rotatedPoint(1,2),'bo')  
plot(rotatedPoint(2,1),rotatedPoint(2,2),'go')
```



## Rotate Group of Points Using Quaternion

Define two points in three-dimensional space. Define a quaternion to rotate the point by first rotating about the z-axis 30 degrees and then about the new y-axis 45 degrees.

```
a = [1,0,0];
b = [0,1,0];
quat = quaternion([30,45,0], 'eulerd', 'ZYX', 'point');
```

Use `rotatepoint` to rotate both points using the quaternion rotation operator. Display the result.

```
rP = rotatepoint(quat, [a;b])
```

```
rP = 2×3
```

```
    0.6124    0.5000   -0.6124
   -0.3536    0.8660    0.3536
```

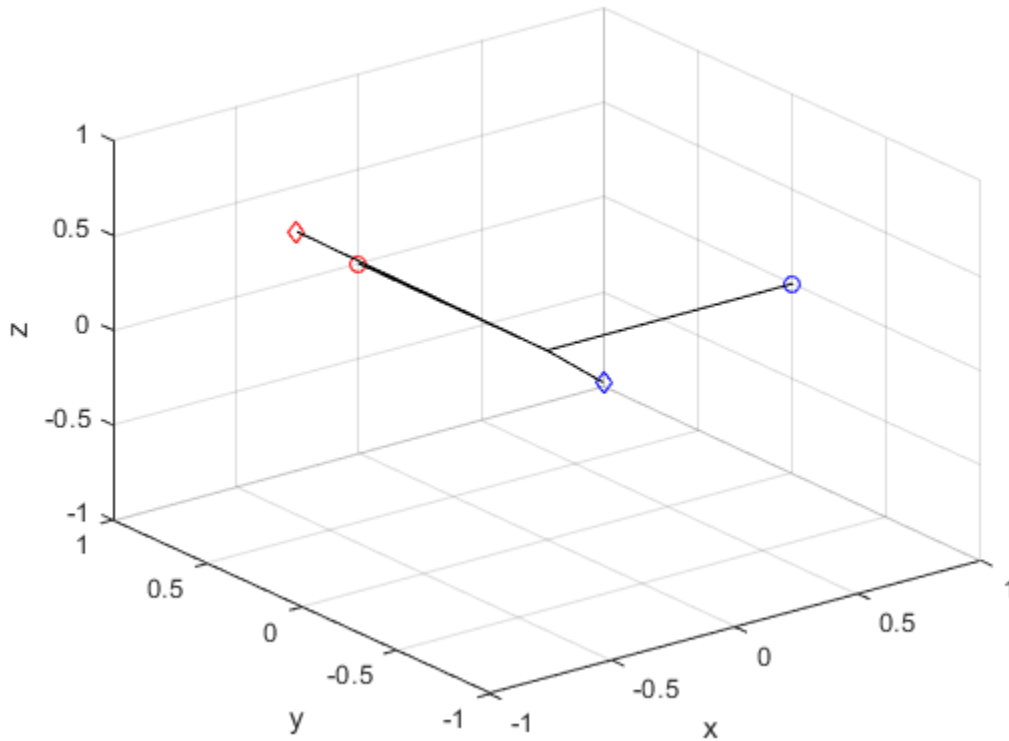
Visualize the original orientation and the rotated orientation of the points. Draw lines from the origin to each of the points for visualization purposes.

```
plot3(a(1),a(2),a(3), 'bo');
```

```
hold on
grid on
axis([-1 1 -1 1 -1 1])
xlabel('x')
ylabel('y')
zlabel('z')
```

```
plot3(b(1),b(2),b(3), 'ro');
plot3(rP(1,1),rP(1,2),rP(1,3), 'bd')
plot3(rP(2,1),rP(2,2),rP(2,3), 'rd')
```

```
plot3([0;rP(1,1)], [0;rP(1,2)], [0;rP(1,3)], 'k')
plot3([0;rP(2,1)], [0;rP(2,2)], [0;rP(2,3)], 'k')
plot3([0;a(1)], [0;a(2)], [0;a(3)], 'k')
plot3([0;b(1)], [0;b(2)], [0;b(3)], 'k')
```



## Input Arguments

### **quat** — Quaternion that defines rotation

scalar | vector

Quaternion that defines rotation, specified as a scalar quaternion, row vector of quaternions, or column vector of quaternions.

Data Types: quaternion

### **cartesianPoints** — Three-dimensional Cartesian points

1-by-3 vector |  $N$ -by-3 matrix

Three-dimensional Cartesian points, specified as a 1-by-3 vector or  $N$ -by-3 matrix.

Data Types: `single` | `double`

## Output Arguments

### **rotationResult** — Repositioned Cartesian points

vector | matrix

Rotated Cartesian points defined using the quaternion rotation, returned as a vector or matrix the same size as `cartesianPoints`.

Data Types: `single` | `double`

## Algorithms

Quaternion point rotation rotates a point specified in  $\mathbf{R}^3$  according to a specified quaternion:

$$L_q(u) = quq^*$$

where  $q$  is the quaternion,  $*$  represents conjugation, and  $u$  is the point to rotate, specified as a quaternion.

For convenience, the `rotatepoint` function takes in a point in  $\mathbf{R}^3$  and returns a point in  $\mathbf{R}^3$ . Given a function call with some arbitrary quaternion,  $q = a + bi + cj + dk$ , 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™.

**Introduced in R2018a**

# rotm2axang

Convert rotation matrix to axis-angle rotation

## Syntax

```
axang = rotm2axang(rotm)
```

## Description

`axang = rotm2axang(rotm)` converts a rotation given as an orthonormal rotation matrix, `rotm`, to the corresponding axis-angle representation, `axang`. The input rotation matrix must be in the premultiply form for rotations.

## Examples

### Convert Rotation Matrix to Axis-Angle Rotation

```
rotm = [1 0 0 ; 0 -1 0; 0 0 -1];  
axang = rotm2axang(rotm)
```

```
axang = 1×4
```

```
1.0000      0      0      3.1416
```

## Input Arguments

**rotm** — Rotation matrix

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

Rotation matrix, specified as a 3-by-3-by- $n$  matrix containing  $n$  rotation matrices. Each rotation matrix has a size of 3-by-3 and must be orthonormal. The input rotation matrix must be in the premultiply form for rotations.

---

**Note** Rotation matrices that are slightly non-orthonormal can give complex outputs. Consider validating your matrix before inputting to the function.

---

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

## Output Arguments

### **axang** — Rotation given in axis-angle form

$n$ -by-4 matrix

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

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

## Extended Capabilities

### **C/C++ Code Generation**

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

### **See Also**

`axang2rotm`

### **Topics**

“Coordinate Transformations in Robotics”

**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) | "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**

eul2rotm

### **Topics**

“Coordinate Transformations in Robotics”

**Introduced in R2015a**

# rotm2quat

Convert rotation matrix to quaternion

## Syntax

```
quat = rotm2quat(rotm)
```

## Description

`quat = rotm2quat(rotm)` converts a rotation matrix, `rotm`, to the corresponding unit quaternion representation, `quat`. The input rotation matrix must be in the premultiply form for rotations.

## Examples

### Convert Rotation Matrix to Quaternion

```
rotm = [0 0 1; 0 1 0; -1 0 0];  
quat = rotm2quat(rotm)
```

```
quat = 1×4
```

```
    0.7071         0    0.7071         0
```

## Input Arguments

**rotm** — Rotation matrix

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

Rotation matrix, specified as a 3-by-3-by- $n$  matrix containing  $n$  rotation matrices. Each rotation matrix has a size of 3-by-3 and is orthonormal. The input rotation matrix must be in the premultiply form for rotations.

---

**Note** Rotation matrices that are slightly non-orthonormal can give complex outputs. Consider validating your matrix before inputting to the function.

---

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

## Output Arguments

### **quat** — Unit quaternion

$n$ -by-4 matrix

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

Example: `[0.7071 0.7071 0 0]`

## Extended Capabilities

### C/C++ Code Generation

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

### See Also

`quat2rotm`

### Topics

“Coordinate Transformations in Robotics”

**Introduced in R2015a**

# rotm2tform

Convert rotation matrix to homogeneous transformation

## Syntax

```
tform = rotm2tform(rotm)
```

## Description

`tform = rotm2tform(rotm)` converts the rotation matrix, `rotm`, into a homogeneous transformation matrix, `tform`. The input rotation matrix must be in the premultiply form for rotations. When using the transformation matrix, premultiply it with the coordinates to be transformed (as opposed to postmultiplying).

## Examples

### Convert Rotation Matrix to Homogeneous Transformation

```
rotm = [1 0 0 ; 0 -1 0; 0 0 -1];  
tform = rotm2tform(rotm)
```

```
tform = 4×4
```

```
    1    0    0    0  
    0   -1    0    0  
    0    0   -1    0  
    0    0    0    1
```

## Input Arguments

**rotm** — Rotation matrix

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

Rotation matrix, specified as a 3-by-3-by- $n$  matrix containing  $n$  rotation matrices. Each rotation matrix has a size of 3-by-3 and is orthonormal. The input rotation matrix must be in the premultiply form for rotations.

---

**Note** Rotation matrices that are slightly non-orthonormal can give complex outputs. Consider validating your matrix before inputting to the function.

---

Example:  $[0 \ 0 \ 1; \ 0 \ 1 \ 0; \ -1 \ 0 \ 0]$

## Output Arguments

### **tform** — Homogeneous transformation

4-by-4-by- $n$  matrix

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

Example:  $[0 \ 0 \ 1 \ 0; \ 0 \ 1 \ 0 \ 0; \ -1 \ 0 \ 0 \ 0; \ 0 \ 0 \ 0 \ 1]$

## Extended Capabilities

### **C/C++ Code Generation**

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

### **See Also**

tform2rotm

### **Topics**

“Coordinate Transformations in Robotics”

**Introduced in R2015a**

# rotmat

Convert quaternion to rotation matrix

## Syntax

```
rotationMatrix = rotmat(quat,rotationType)
```

## Description

`rotationMatrix = rotmat(quat,rotationType)` converts the quaternion, `quat`, to an equivalent rotation matrix representation.

## Examples

### Convert Quaternion to Rotation Matrix for Point Rotation

Define a quaternion for use in point rotation.

```
theta = 45;
gamma = 30;
quat = quaternion([0,theta,gamma], 'eulerd', 'ZYX', 'point')

quat = quaternion
      0.8924 + 0.23912i + 0.36964j + 0.099046k
```

Convert the quaternion to a rotation matrix.

```
rotationMatrix = rotmat(quat, 'point')

rotationMatrix = 3×3

    0.7071    -0.0000    0.7071
    0.3536    0.8660   -0.3536
   -0.6124    0.5000    0.6124
```



To verify the rotation matrix, directly create two rotation matrices corresponding to the rotations about the  $y$ - and  $x$ -axes. Multiply the rotation matrices and compare to the output of `rotmat`.

```
theta = 45;
gamma = 30;

ry = [cosd(theta)  0      sind(theta) ; ...
      0            1      0            ; ...
      -sind(theta) 0      cosd(theta)];

rx = [1      0      0      ; ...
      0      cosd(gamma) -sind(gamma) ; ...
      0      sind(gamma)  cosd(gamma)];

rotationMatrixVerification = rx*ry

rotationMatrixVerification = 3x3

    0.7071         0    0.7071
    0.3536    0.8660   -0.3536
   -0.6124    0.5000    0.6124
```

## Convert Quaternion to Rotation Matrix for Frame Rotation

Define a quaternion for use in frame rotation.

```
theta = 45;
gamma = 30;
quat = quaternion([0,theta,gamma], 'eulerd', 'ZYX', 'frame')

quat = quaternion
    0.8924 + 0.23912i + 0.36964j - 0.099046k
```

Convert the quaternion to a rotation matrix.

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

rotationMatrix = 3x3
```

```
0.7071    -0.0000    -0.7071
0.3536     0.8660     0.3536
0.6124    -0.5000     0.6124
```

To verify the rotation matrix, directly create two rotation matrices corresponding to the rotations about the  $y$ - and  $x$ -axes. Multiply the rotation matrices and compare to the output of `rotmat`.

```
theta = 45;
gamma = 30;

ry = [cosd(theta)  0          -sind(theta) ; ...
      0            1           0           ; ...
      sind(theta)  0          cosd(theta)];

rx = [1           0           0           ; ...
      0           cosd(gamma) sind(gamma) ; ...
      0           -sind(gamma) cosd(gamma)];

rotationMatrixVerification = rx*ry

rotationMatrixVerification = 3x3

0.7071         0    -0.7071
0.3536     0.8660     0.3536
0.6124    -0.5000     0.6124
```

### Convert Quaternion Vector to Rotation Matrices

Create a 3-by-1 normalized quaternion vector.

```
qVec = normalize(quaternion(randn(3,4)));
```

Convert the quaternion array to rotation matrices. The pages of `rotmatArray` correspond to the linear index of `qVec`.

```
rotmatArray = rotmat(qVec, 'frame');
```

Assume `qVec` and `rotmatArray` correspond to a sequence of rotations. Combine the quaternion rotations into a single representation, then apply the quaternion rotation to arbitrarily initialized Cartesian points.

```
loc = normalize(randn(1,3));
quat = prod(qVec);
rotateframe(quat,loc)
```

```
ans = 1×3
    0.9524    0.5297    0.9013
```

Combine the rotation matrices into a single representation, then apply the rotation matrix to the same initial Cartesian points. Verify the quaternion rotation and rotation matrix result in the same orientation.

```
totalRotMat = eye(3);
for i = 1:size(rotmatArray,3)
    totalRotMat = rotmatArray(:,:,i)*totalRotMat;
end
totalRotMat*loc'
```

```
ans = 3×1
    0.9524
    0.5297
    0.9013
```

## Input Arguments

### **quat** — Quaternion to convert

scalar | vector | matrix | multidimensional array

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

Data Types: quaternion

### **rotationType** — Type or rotation

'frame' | 'point'

Type of rotation represented by the `rotationMatrix` output, specified as 'frame' or 'point'.

Data Types: `char` | `string`

## Output Arguments

### **rotationMatrix** — Rotation matrix representation

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

Rotation matrix representation, returned as a 3-by-3 matrix or 3-by-3-by-*N* multidimensional array.

- If `quat` is a scalar, `rotationMatrix` is returned as a 3-by-3 matrix.
- If `quat` is non-scalar, `rotationMatrix` is returned as a 3-by-3-by-*N* multidimensional array, where `rotationMatrix(:, :, i)` is the rotation matrix corresponding to `quat(i)`.

The data type of the rotation matrix is the same as the underlying data type of `quat`.

Data Types: `single` | `double`

## Algorithms

Given a quaternion of the form

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

the equivalent rotation matrix for frame rotation is defined as

$$\begin{bmatrix} 2a^2 - 1 + 2b^2 & 2bc + 2ad & 2bd - 2ac \\ 2bc - 2ad & 2a^2 - 1 + 2c^2 & 2cd + 2ab \\ 2bd + 2ac & 2cd - 2ab & 2a^2 - 1 + 2d^2 \end{bmatrix}.$$

The equivalent rotation matrix for point rotation is the transpose of the frame rotation matrix:

$$\begin{bmatrix} 2a^2 - 1 + 2b^2 & 2bc - 2ad & 2bd + 2ac \\ 2bc + 2ad & 2a^2 - 1 + 2c^2 & 2cd - 2ab \\ 2bd - 2ac & 2cd + 2ab & 2a^2 - 1 + 2d^2 \end{bmatrix}.$$

## References

- [1] Kuipers, Jack B. *Quaternions and Rotation Sequences: A Primer with Applications to Orbits, Aerospace, and Virtual Reality*. Princeton, NJ: Princeton University Press, 2007.

## Extended Capabilities

### C/C++ Code Generation

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

**Introduced in R2018a**

## rottraj

Generate trajectories between orientation rotation matrices

### Syntax

```
[R,omega,alpha] = rottraj(r0,rF,tInterval,tSamples)
[R,omega,alpha] = rottraj(r0,rF,tInterval,tSamples,Name,Value)
```

### Description

`[R,omega,alpha] = rottraj(r0,rF,tInterval,tSamples)` generates a trajectory that interpolates between two orientations, `r0` and `rF`, with points based on the time interval and given time samples.

`[R,omega,alpha] = rottraj(r0,rF,tInterval,tSamples,Name,Value)` specifies additional parameters using `Name,Value` pair arguments.

### Examples

#### Interpolate Trajectory Between Quaternions

Define two quaternion waypoints to interpolate between.

```
q0 = quaternion([0 pi/4 -pi/8], 'euler', 'ZYX', 'point');
qF = quaternion([3*pi/2 0 -3*pi/4], 'euler', 'ZYX', 'point');
```

Specify a vector of times to sample the quaternion trajectory.

```
tvec = 0:0.01:5;
```

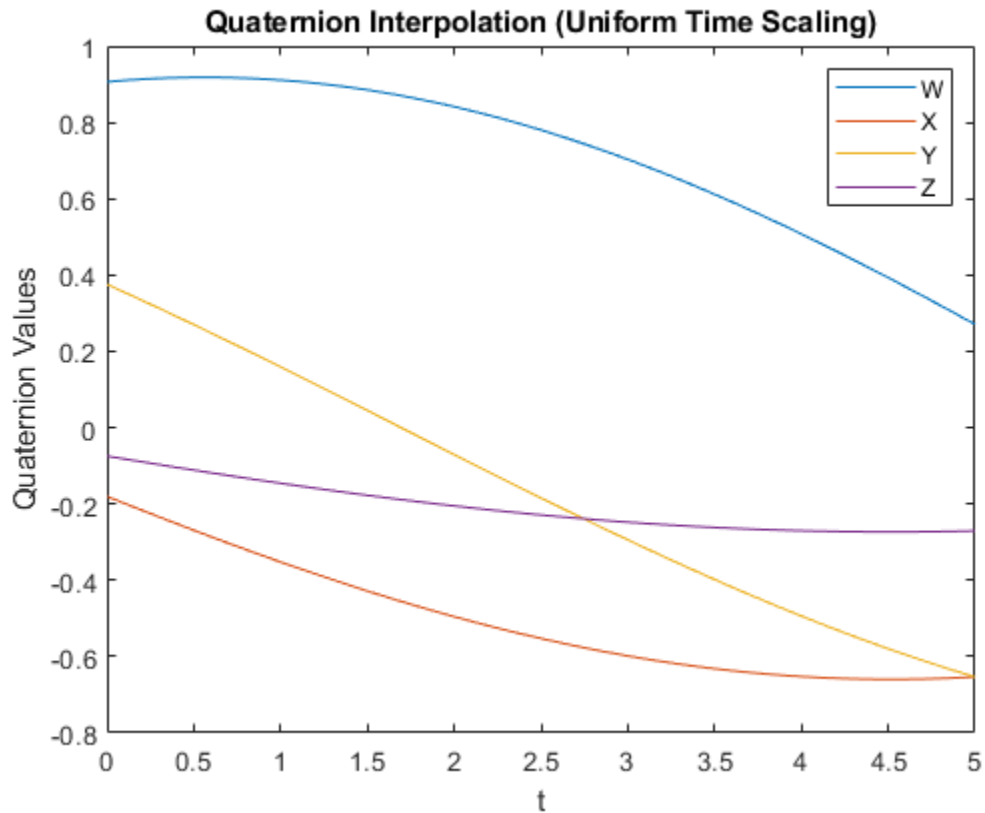
Generate the trajectory. Plot the results.

```
[qInterp1,w1,a1] = rottraj(q0,qF,[0 5],tvec);
plot(tvec,compact(qInterp1))
```

```

title('Quaternion Interpolation (Uniform Time Scaling)')
xlabel('t')
ylabel('Quaternion Values')
legend('W', 'X', 'Y', 'Z')

```



### Interpolate Trajectory Between Rotation Matrices

Define two rotation matrix waypoints to interpolate between.

```

r0 = [1 0 0; 0 1 0; 0 0 1];
rF = [0 0 1; 1 0 0; 0 0 0];

```

Specify a vector of times to sample the quaternion trajectory.

```
tvec = 0:0.1:1;
```

Generate the trajectory. Plot the results using `plotTransforms`. Convert the rotation matrices to quaternions and specify zero translation. The figure shows all the intermediate rotations of the coordinate frame.

```
[rInterpl,wl,a1] = rottraj(r0,rF,[0 1],tvec);
```

```
rotations = rotm2quat(rInterpl);
```

```
zeroVect = zeros(length(rotations),1);
```

```
translations = [zeroVect,zeroVect,zeroVect];
```

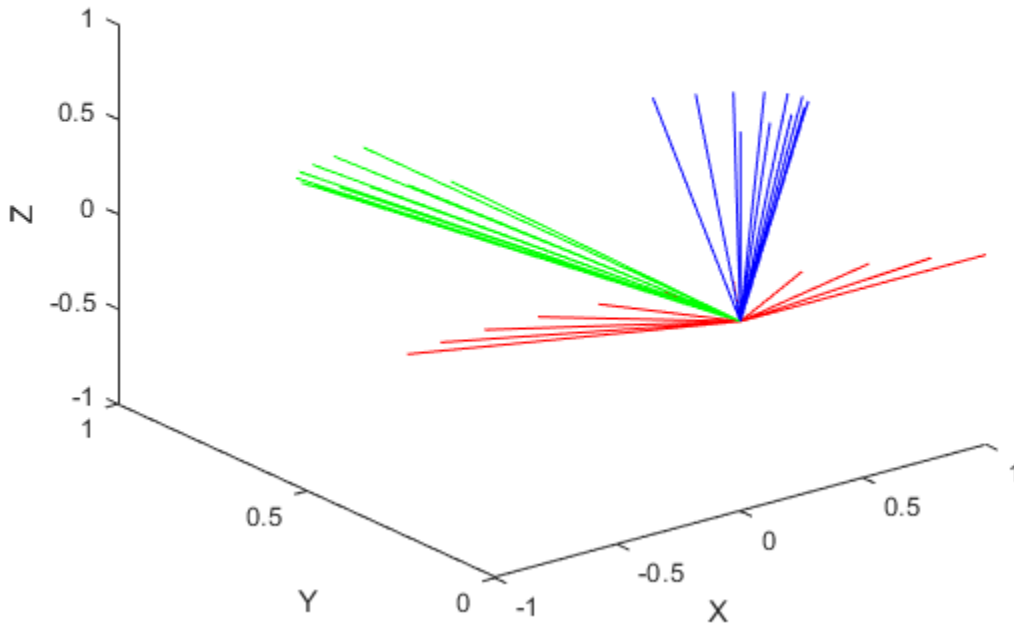
```
plotTransforms(translations,rotations)
```

```
xlabel('X')
```

```
ylabel('Y')
```

```
zlabel('Z')
```





## Input Arguments

### **r0** — Initial orientation

3-by-3 rotation matrix | quaternion object

Initial orientation, specified as a 3-by-3 rotation matrix or quaternion object. The function generates a trajectory that starts at the initial orientation, `r0`, and goes to the final orientation, `rF`.

Example: `quaternion([0 pi/4 -pi/8], 'euler', 'ZYX', 'point');`

Data Types: `single` | `double`

### **rF — Final orientation**

3-by-3 rotation matrix | quaternion object

Final orientation, specified as a 3-by-3 rotation matrix or quaternion object. The function generates a trajectory that starts at the initial orientation, `r0`, and goes to the final orientation, `rF`.

Example: `quaternion([3*pi/2 0 -3*pi/4], 'euler', 'ZYX', 'point')`

Data Types: `single` | `double`

### **tInterval — Start and end times for trajectory**

two-element vector

Start and end times for the trajectory, specified as a two-element vector.

Example: `[0 10]`

Data Types: `single` | `double`

### **tSamples — Time samples for trajectory**

*m*-element vector

Time samples for the trajectory, specified as an *m*-element vector. The output trajectory, `rotVector`, is a vector of orientations.

Example: `0:0.01:10`

Data Types: `single` | `double`

## **Name-Value Pair Arguments**

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

Example: `'TimeScaling', [0 1 2; 0 1 0; 0 0 0]`

### **TimeScaling — Time scaling vector and first two derivatives**

3-by-*m* vector

Time scaling vector and the first two derivatives, specified as the comma-separated pair of `'TimeScaling'` and a 3-by-*m* vector, where *m* is the length of `tSamples`. By default, the time scaling is a linear time scaling between the time points in `tInterval`.

For a nonlinear time scaling, specify the values of the time points in the first row. The second and third rows are the velocity and acceleration of the time points, respectively. For example, to follow the path with a linear velocity to the halfway point, and then jump to the end, the time-scaling would be:

```
s(1,:) = [0 0.25 0.5 1 1 1] % Position
s(2,:) = [1 1 1 0 0 0] % Velocity
s(3,:) = [0 0 0 0 0 0] % Acceleration
```

Data Types: `single` | `double`

## Output Arguments

### **R** — Orientation trajectory

3-by-3-by- $m$  rotation matrix array | quaternion object array

Orientation trajectory, returned as a 3-by-3-by- $m$  rotation matrix array or quaternion object array, where  $m$  is the number of points in `tSamples`. The output type depends on the inputs from `r0` and `rF`.

### **omega** — Orientation angular velocity

3-by- $m$  matrix

Orientation angular velocity, returned as a 3-by- $m$  matrix, where  $m$  is the number of points in `tSamples`.

### **alpha** — Orientation angular acceleration

3-by- $m$  matrix

Orientation angular acceleration, returned as a 3-by- $m$  matrix, where  $m$  is the number of points in `tSamples`

## Limitations

- When specifying your `r0` and `rF` input arguments as a 3-by-3 rotation matrix, they are converted to a quaternion object before interpolating the trajectory. If your rotation matrix does not follow a right-handed coordinate system or does not have a direct conversion to quaternions, this conversion may result in different initial and final rotations in the output trajectory.

## **Extended Capabilities**

### **C/C++ Code Generation**

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

### **See Also**

`bsplinepolytraj` | `cubicpolytraj` | `quaternion` | `quinticpolytraj` | `transformtraj` | `trapveltraj`

**Introduced in R2019a**

## 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  $\theta$  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  $\theta$  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™.

**Introduced in R2018a**

# 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  $\theta$  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  $\theta$  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™.

**Introduced in R2018a**

# runCore

Start ROS core

## Syntax

```
runCore(device)
```

## Description

`runCore(device)` starts the ROS core on the connected device. The ROS master uses a default port number of 11311.

## Examples

### Run ROS Core on ROS Device

Connect to a remote ROS device and start a ROS core. The ROS core is needed to run ROS nodes to communicate via a ROS network. You can run and stop a ROS core or node and check their status using a `rosdevice` object.

Create a connection to a ROS device. Specify the address, user name, and password of your specific ROS device. The device contains information about the ROS device, including the available ROS nodes that can be run using `runNode`.

```
ipaddress = '192.168.154.131';  
d = rosdevice(ipaddress, 'user', 'password')
```

```
d =
```

```
rosdevice with properties:
```

```
DeviceAddress: '192.168.154.131'  
Username: 'user'
```

```
ROSFolder: '/opt/ros/hydro'  
CatkinWorkspace: '~/catkin_ws_test'  
AvailableNodes: {'robotcontroller' 'robotcontroller2'}
```

Run a ROS core and check if it is running.

```
runCore(d)  
running = isCoreRunning(d)
```

```
running =  
  logical  
  1
```

Stop the ROS core and confirm that it is no longer running.

```
stopCore(d)  
running = isCoreRunning(d)
```

```
running =  
  logical  
  0
```

## Input Arguments

### **device** — ROS device

rosdevice object

ROS device, specified as a rosdevice object.

## See Also

isCoreRunning | rosdevice | stopCore

## **Topics**

“Generate a Standalone ROS Node from Simulink®”

**Introduced in R2016b**

# runNode

Start ROS node

## Syntax

```
runNode(device,modelName)
runNode(device,modelName,masterURI)
runNode(device,modelName,masterURI,nodeHost)
```

## Description

`runNode(device,modelName)` starts the ROS node associated with the deployed Simulink model named `modelName`. The ROS node must be deployed in the Catkin workspace specified by the `CatkinWorkspace` property of the input `rosdevice` object, `device`. By default, the node connects to the ROS master that MATLAB is connected to with the `device.DeviceAddress` property.

`runNode(device,modelName,masterURI)` connects to the specified master URI.

`runNode(device,modelName,masterURI,nodeHost)` connects to the specified master URI and node host. The node advertises its address as the hostname or IP address given in `nodeHost`.

## Examples

### Run ROS Node on ROS Device

Connect to a remote ROS device and start a ROS node. First, run a ROS core so that ROS nodes can communicate via a ROS network. You can run and stop a ROS core or node and check their status using a `rosdevice` object.

Create a connection to a ROS device. Specify the address, user name and password of your specific ROS device. The device contains information about the ROS device, including the available ROS nodes that can be run using `runNode`.

```

ipaddress = '192.168.203.129';
d = rosdevice(ipaddress, 'user', 'password');
d.ROSFolder = '/opt/ros/hydro';
d.CatkinWorkspace = '~/catkin_ws_test'

```

```
d =
```

```
rosdevice with properties:
```

```

    DeviceAddress: '192.168.203.129'
      Username: 'user'
      ROSFolder: '/opt/ros/hydro'
CatkinWorkspace: '~/catkin_ws_test'
AvailableNodes: {'robotcontroller' 'robotcontroller2'}

```

Run a ROS core. Connect MATLAB® to the ROS master using `rosinit`. This core enables you to run ROS nodes on your ROS device.

```

runCore(d)
rosinit(d.DeviceAddress, 11311)

```

```
Initializing global node /matlab_global_node_12272 with NodeURI http://192.168.203.1:6
```

Check the available ROS nodes on the connected ROS device. These nodes were generated from Simulink® models following the process in the “Generate a Standalone ROS Node from Simulink®” example.

```
d.AvailableNodes
```

```
ans =
```

```
1×2 cell array
```

```
 {'robotcontroller'} {'robotcontroller2'}
```

Run a ROS node. specifying the node name. Check if the node is running.

```

runNode(d, 'robotcontroller')
running = isNodeRunning(d, 'robotcontroller')

```

```
running =
```

```
logical
```

```
1
```

Stop the ROS node. Disconnect from the ROS network. Stop the ROS core.

```
stopNode(d, 'robotcontroller')  
roshutdown  
stopCore(d)
```

```
Shutting down global node /matlab_global_node_12272 with NodeURI http://192.168.203.1:
```

### Run Multiple ROS Nodes

Run multiple ROS nodes on a connected ROS device. ROS nodes can be generated using Simulink® models to perform different tasks on the ROS network. These nodes are then deployed on a ROS device and can be run independently of Simulink®.

This example uses two different Simulink models that have been deployed as ROS nodes. See “Generate a Standalone ROS Node from Simulink®” and follow the instructions to generate and deploy a ROS node. The 'robotcontroller' node sends velocity commands to a robot to navigate it to a given point. The 'robotcontroller2' node uses the same model, but doubles the linear velocity to drive the robot faster.

Create a connection to a ROS device. Specify the address, user name, and password of your specific ROS device. The device contains information about the ROS device, including the available ROS nodes that can be run using runNode.

```
ipaddress = '192.168.203.129';  
d = rosdevice(ipaddress, 'user', 'password')
```

```
d =
```

```
rosdevice with properties:
```

```
DeviceAddress: '192.168.203.129'  
Username: 'user'  
ROSFolder: '/opt/ros/indigo'  
CatkinWorkspace: '~/catkin_ws_test'
```



```
AvailableNodes: {'robotcontroller' 'robotcontroller2'}
```

Run a ROS core. The ROS Core is the master enables you to run ROS nodes on your ROS device. Connect MATLAB® to the ROS master using `rosinit`. For this example, the port is set to 11311. `rosinit` can automatically select a port for you without specifying this input.

```
runCore(d)
rosinit(d.DeviceAddress,11311)
```

```
Initializing global node /matlab_global_node_15972 with NodeURI http://192.168.203.1:5
```

Check the available ROS nodes on the connected ROS device. The nodes listed in this example were generated from Simulink® models following the process in the “Generate a Standalone ROS Node from Simulink®” example. Two separate nodes are generated, 'robotcontroller' and 'robotcontroller2', which have the linear velocity set to 1 and 2 in the model respectively.

```
d.AvailableNodes
```

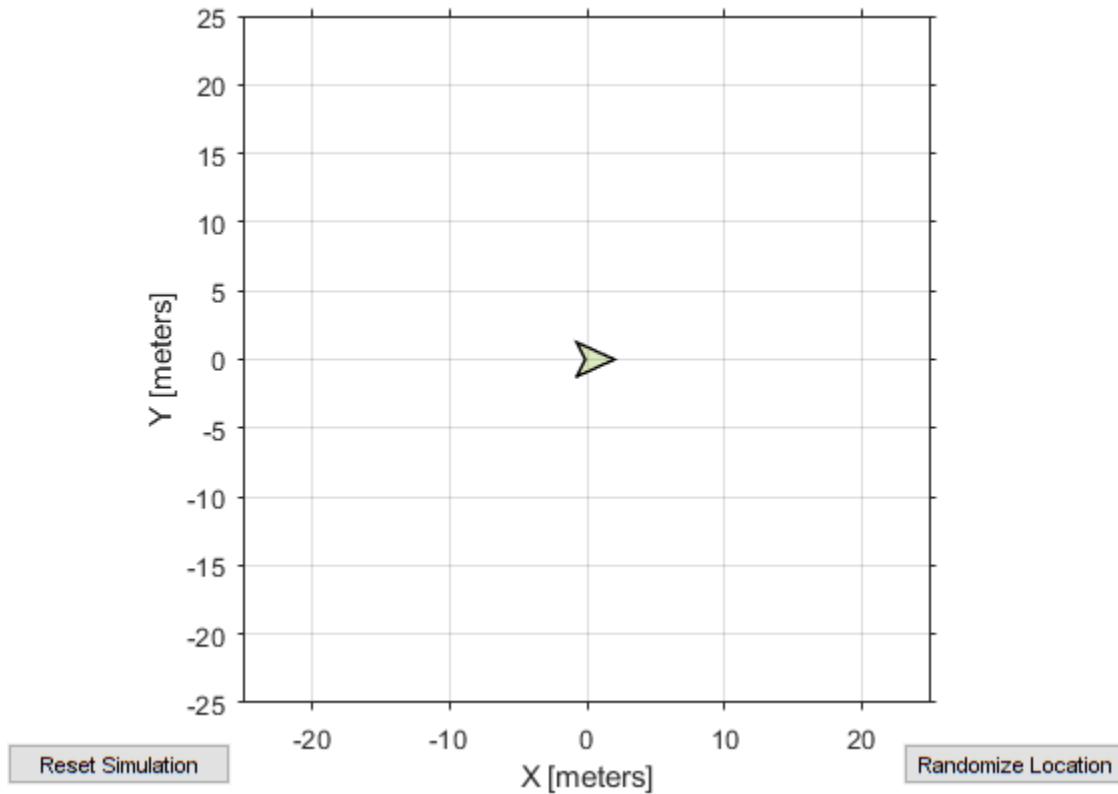
```
ans =
```

```
1×2 cell array
```

```
 {'robotcontroller'}    {'robotcontroller2'}
```

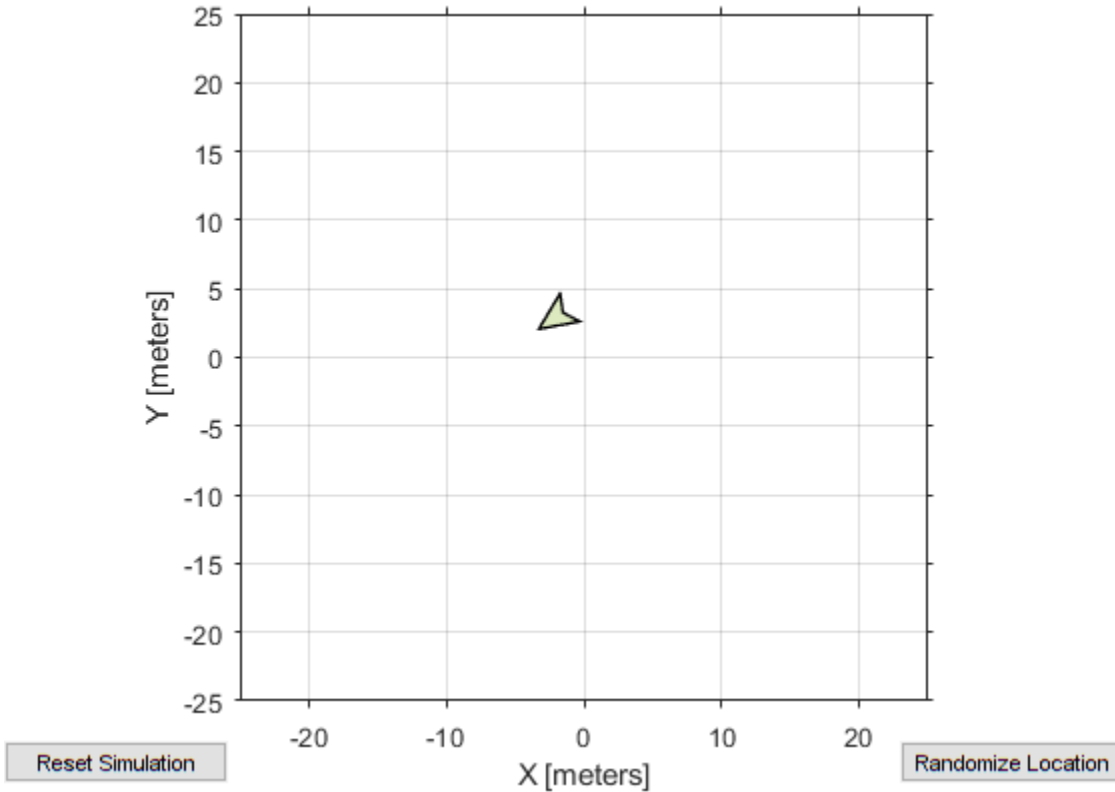
Start up the Robot Simulator using `ExampleHelperSimulinkRobotROS`. This simulator automatically connects to the ROS master on the ROS device. You will use this simulator to run a ROS node and control the robot.

```
sim = ExampleHelperSimulinkRobotROS;
```



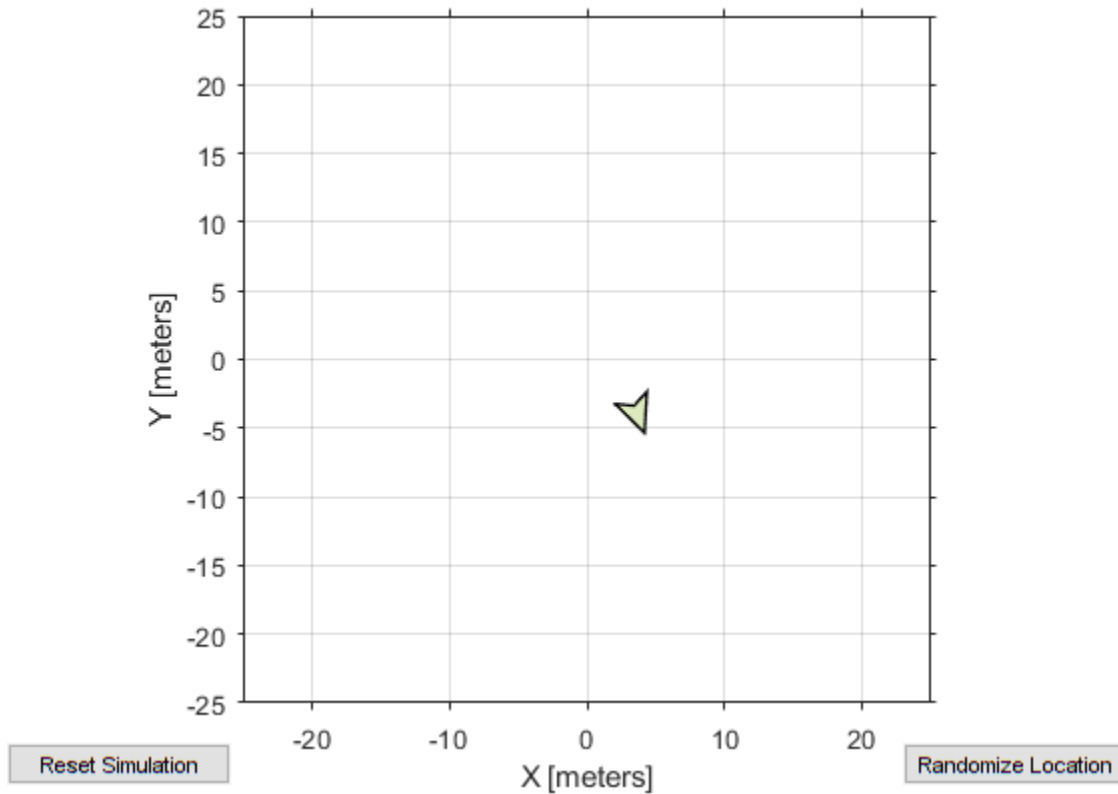
Run a ROS node, specifying the node name. The 'robotcontroller' node commands the robot to a specific location ([ -10 10]). Wait to see the robot drive.

```
runNode(d, 'robotcontroller')  
pause(10)
```



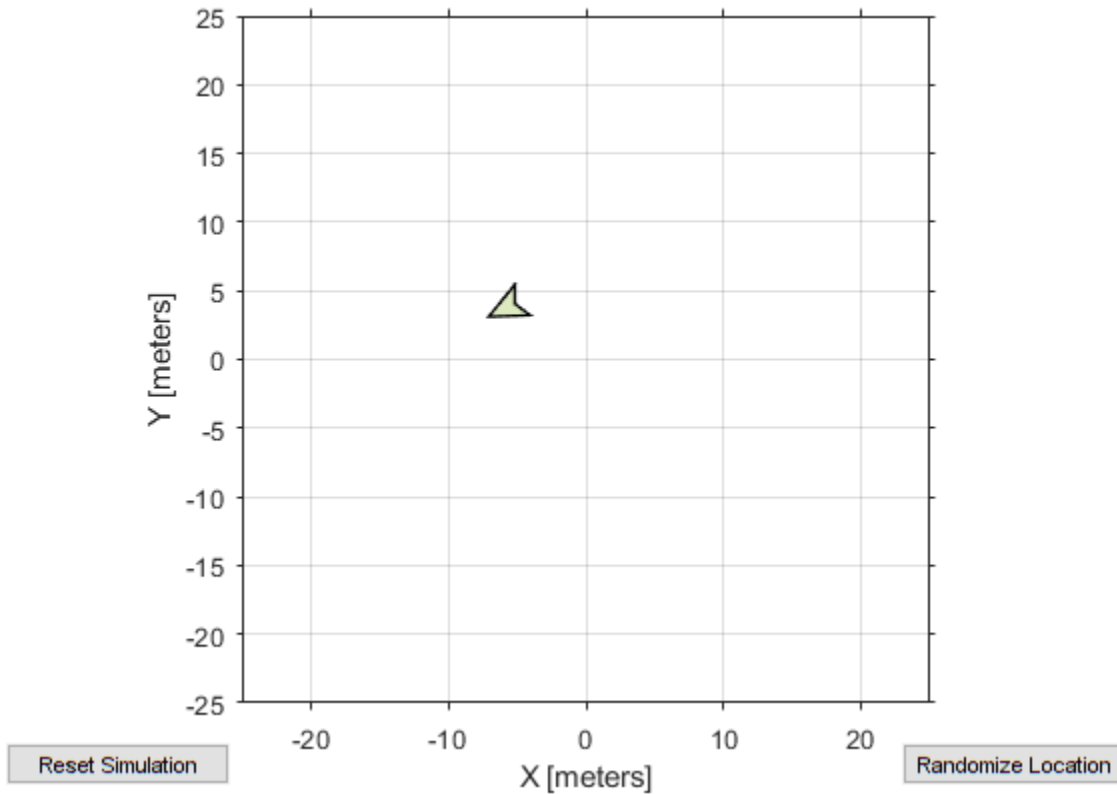
Reset the Robot Simulator to reset the robot position. Alternatively, click **Reset Simulation**. Because the node is still running, the robot continues back to the specific location. To stop sending commands, stop the node.

```
resetSimulation(sim.Simulator)
pause(5)
stopNode(d, 'robotcontroller')
```



Run the 'robotcontroller2' node. This model drives the robot with twice the linear velocity. Reset the robot position. Wait to see the robot drive. You should see a wider turn due to the increased velocity.

```
runNode(d, 'robotcontroller2')
resetSimulation(sim.Simulator)
pause(10)
```



Close the simulator. Stop the ROS node. Disconnect from the ROS network and stop the ROS core.

```
close
stopNode(d, 'robotcontroller2')
roshutdown
stopCore(d)
```

Shutting down global node /matlab\_global\_node\_15972 with NodeURI http://192.168.203.1:

## Input Arguments

### **device — ROS device**

rosdevice object

ROS device, specified as a rosdevice object.

### **modelName — Name of the deployed Simulink model**

character vector

Name of the deployed Simulink model, specified as a character vector. If the model name is not valid, the function returns an error.

### **masterURI — URI of the ROS master**

character vector

URI of the ROS master, specified as a character vector. On start up, the node connects to the ROS master with the given URI.

### **nodeHost — Host name for the node**

character vector

Host name for the node, specified as a character vector. The node uses this host name to advertise itself on the ROS network for others to connect to it.

## See Also

`isNodeRunning` | `rosdevice` | `stopNode`

## Topics

“Connect to a ROS Network”

“Generate a Standalone ROS Node from Simulink®”

## Introduced in R2016b

## scatter3

Display point cloud in scatter plot

### Syntax

```
scatter3(pcloud)
scatter3(pcloud,Name,Value)
h = scatter3( ___ )
```

### Description

`scatter3(pcloud)` plots the input `pcloud` point cloud as a 3-D scatter plot in the current axes handle. If the data contains RGB information for each point, the scatter plot is colored accordingly.

`scatter3(pcloud,Name,Value)` provides additional options specified by one or more `Name,Value` pair arguments. `Name` must appear inside single quotes ( `' '` ). You can specify several name-value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`

`h = scatter3( ___ )` returns the scatter series object, using any of the arguments from previous syntaxes. Use `h` to modify properties of the scatter series after it is created.

When plotting ROS point cloud messages, MATLAB follows the standard ROS convention for axis orientation. This convention states that **positive x is forward, positive y is left, and positive z is up**. However, if cameras are used, a second frame is defined with an “\_optical” suffix which changes the orientation of the axis. In this case, positive z is forward, positive x is right, and positive y is down. MATLAB looks for the “\_optical” suffix and will adjust the axis orientation of the scatter plot accordingly. For more information, see [Axis Orientation](#) on the ROS Wiki.

### Examples

### Get and Plot a 3-D Point Cloud

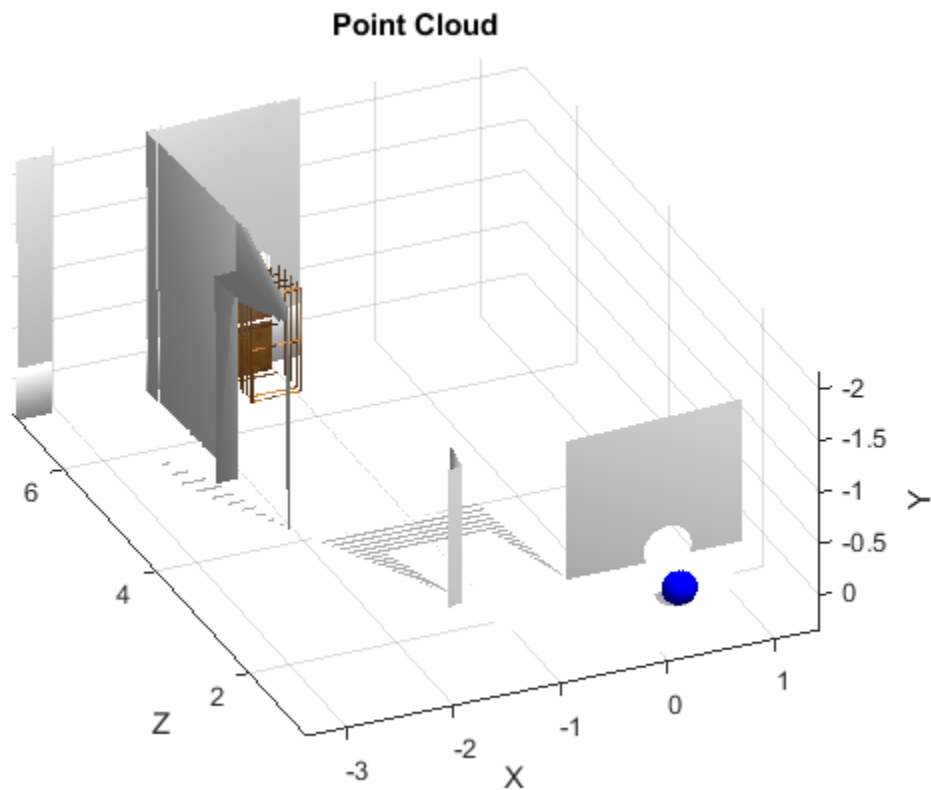
Connect to a ROS network. Subscribe to a point cloud message topic.

```
rosinit('192.168.154.131')
sub = rossubscriber('/camera/depth/points');
pause(1)
```

Initializing global node /matlab\_global\_node\_47682 with NodeURI <http://192.168.154.1:6>

Get the latest point cloud message. Plot the point cloud.

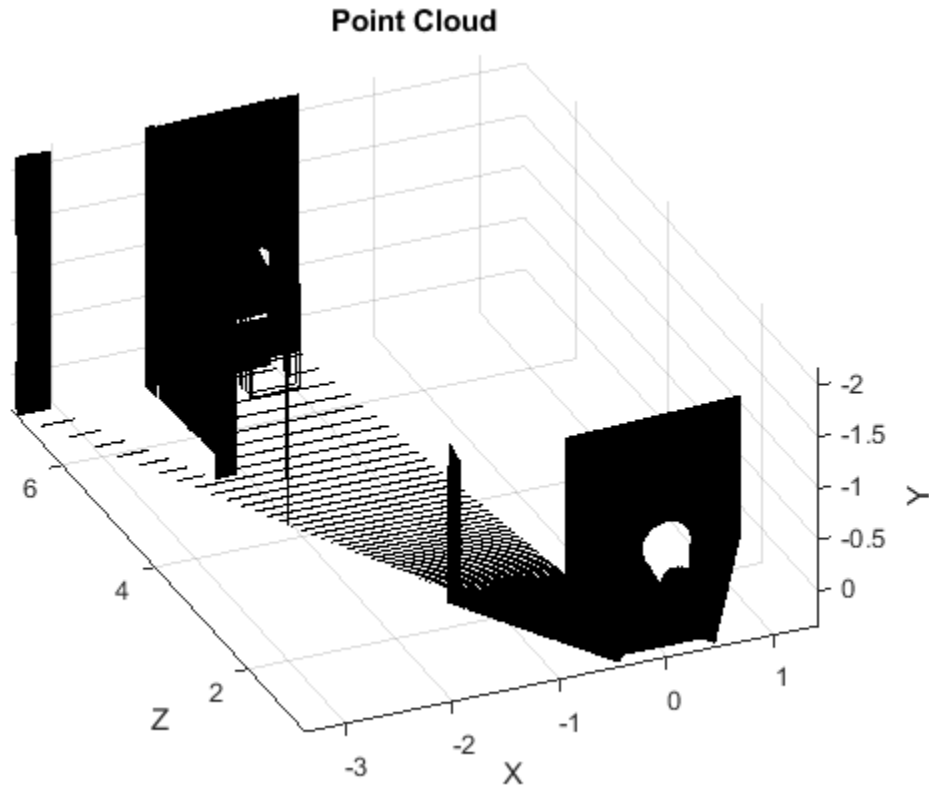
```
pcloud = sub.LatestMessage;
scatter3(pcloud)
```





Plot all points as black dots.

```
scatter3(sub.LatestMessage, 'MarkerEdgeColor', [0 0 0]);
```



## Input Arguments

### **pcloud** — Point cloud

PointCloud2 object handle

Point cloud, specified as a PointCloud2 object handle for a 'sensor\_msgs/PointCloud2' ROS message.

## 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: `'MarkerEdgeColor',[1 0 0]`

### MarkerEdgeColor — Marker outline color





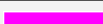


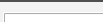
`'flat'` (default) | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Marker outline color, specified `'flat'`, an RGB triplet, a hexadecimal color code, a color name, or a short name. The default value of `'flat'` uses colors from the `CData` property.

For a custom color, specify an RGB triplet or a hexadecimal color code.





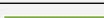
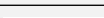
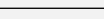
- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (`#`) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
<code>'red'</code>	<code>'r'</code>	<code>[1 0 0]</code>	<code>'#FF0000'</code>	
<code>'green'</code>	<code>'g'</code>	<code>[0 1 0]</code>	<code>'#00FF00'</code>	
<code>'blue'</code>	<code>'b'</code>	<code>[0 0 1]</code>	<code>'#0000FF'</code>	
<code>'cyan'</code>	<code>'c'</code>	<code>[0 1 1]</code>	<code>'#00FFFF'</code>	
<code>'magenta'</code>	<code>'m'</code>	<code>[1 0 1]</code>	<code>'#FF00FF'</code>	
<code>'yellow'</code>	<code>'y'</code>	<code>[1 1 0]</code>	<code>'#FFFF00'</code>	
<code>'black'</code>	<code>'k'</code>	<code>[0 0 0]</code>	<code>'#000000'</code>	
<code>'white'</code>	<code>'w'</code>	<code>[1 1 1]</code>	<code>'#FFFFFF'</code>	

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
'none'	Not applicable	Not applicable	Not applicable	No color

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
[0 0.4470 0.7410]	'#0072BD'	
[0.8500 0.3250 0.0980]	'#D95319'	
[0.9290 0.6940 0.1250]	'#EDB120'	
[0.4940 0.1840 0.5560]	'#7E2F8E'	
[0.4660 0.6740 0.1880]	'#77AC30'	
[0.3010 0.7450 0.9330]	'#4DBEEE'	
[0.6350 0.0780 0.1840]	'#A2142F'	

Example: [0.5 0.5 0.5]

Example: 'blue'

Example: '#D2F9A7'

### Parent — Parent of axes

axes object

Parent of axes, specified as the comma-separated pair consisting of 'Parent' and an axes object in which to draw the point cloud. By default, the point cloud is plotted in the active axes.

## Outputs

### h — Scatter series object

scalar

Scatter series object, returned as a scalar. This value is a unique identifier, which you can use to query and modify the properties of the scatter object after it is created.

## **See Also**

readRGB | readXYZ

**Introduced in R2015a**

# search

Search ROS network for parameter names

## Syntax

```
pnames = search(ptree,searchstr)
[pnames,pvalues] = search(ptree,searchstr)
```

## Description

`pnames = search(ptree,searchstr)` searches within the parameter tree `ptree` and returns the parameter names that contain the specified search string, `searchstr`.

`[pnames,pvalues] = search(ptree,searchstr)` also returns the parameter values.

The following ROS data types are supported as values of parameters. For each ROS data type, the corresponding MATLAB data type is also listed:

- 32-bit integers — `int32`
- booleans — `logical`
- doubles — `double`
- strings — string scalar, `string`, or character vector, `char`
- lists — cell array
- dictionaries — structure

## Examples

### Search for ROS Parameter Names

Connect to ROS network. Specify the IP address of the ROS master.

```
rosinit('192.168.154.131')
```

```
Initializing global node /matlab_global_node_11803 with NodeURI http://192.168.154.1:5
```

Create a parameter tree.

```
ptree = rosparam;
```

Search for parameter names that contain 'gravity'.

```
[pnames,pvalues] = search(ptree,'gravity')
```

```
pnames =
```

```
1x3 cell array
```

```
    '/gazebo/gravity_x'    '/gazebo/gravity_y'    '/gazebo/gravity_z'
```

```
pvalues =
```

```
3x1 cell array
```

```
    [    0]
    [    0]
    [-9.8000]
```

## Input Arguments

### **ptree** — Parameter tree

ParameterTree object handle

Parameter tree, specified as a ParameterTree object handle. Create this object using the `rosparam` function.

### **searchstr** — ROS parameter search string

string scalar | character vector

ROS parameter search string specified as a string scalar or character vector. `search` returns all parameters that contain this character vector.

## Output Arguments

### **pnames** — Parameter values

cell array of character vectors

Parameter names, returned as a cell array of character vectors. These character vectors match the parameter names in the ROS master that contain the search character vector.

### **pvalues** — Parameter values

cell array

The following ROS data types are supported as values of parameters. For each ROS data type, the corresponding MATLAB data type is also listed:

- 32-bit integers — `int32`
- booleans — `logical`
- doubles — `double`
- strings — string scalar, `string`, or character vector, `char`
- lists — cell array
- dictionaries — structure

Base64-encoded binary data and iso8601 data from ROS are not supported.

## Limitations

Base64-encoded binary data and iso8601 data from ROS are not supported.

## See Also

`get` | `rosparam`

**Introduced in R2015a**

# seconds

Returns seconds of a time or duration

## Syntax

```
secs = seconds(time)
secs = seconds(duration)
```

## Description

`secs = seconds(time)` returns the scalar number, `secs`, in seconds that represents the same value as the time object, `time`.

`secs = seconds(duration)` returns the scalar number, `secs`, in seconds that represents the same value as the duration object, `duration`.

## Examples

### Get Seconds From A Time Object

Use the `seconds` function to get the total seconds of a `Time` object from its `Secs` and `Nsecs` properties.

Create a `Time` object.

```
time = rostime(1,860000000)
```

```
time =
  ROS Time with properties:
```

```
    Sec: 1
    Nsec: 860000000
```

Get the total seconds from the time object.



```
secs = seconds(time)
```

```
secs = 1.8600
```

## Input Arguments

### **time** — Current ROS or system time

Time object handle

ROS or system time, specified as a Time object handle. Create a Time object using `rostime`.

### **duration** — Duration

ROS Duration object

Duration, specified as a ROS Duration object with `Sec` and `Nsec` properties. Create a Duration object using `rosduration`

## Output Arguments

### **secs** — Total time

scalar in seconds

Total time of the Time or Duration object, returned as a scalar in seconds.

## See Also

`rosduration` | `rostime`

**Introduced in R2016a**

# select

Select subset of messages in rosbag

## Syntax

```
bagsel = select(bag)
bagsel = select(bag, Name, Value)
```

## Description

`bagsel = select(bag)` returns an object, `bagsel`, that contains all of the messages in the `BagSelection` object, `bag`

This function does not change the contents of the original `BagSelection` object. It returns a new object that contains the specified message selection.

`bagsel = select(bag, Name, Value)` provides additional options specified by one or more `Name, Value` pair arguments.

## Examples

### Create Copy of rosbag

Retrieve the rosbag. Specify the file path.

```
bag = rosbag('ex_multiple_topics.bag');
```

Use `select` with no selection criteria to create a copy of the rosbag.

```
bagCopy = select(bag);
```

## Select Subset of Messages In rosbag

Retrieve the rosbag. Specify the file path.

```
bag = rosbag('ex_multiple_topics.bag');
```

Select all messages within the first second of the rosbag.

```
bag = select(bag, 'Time', [bag.StartTime, bag.StartTime + 1]);
```

## Input Arguments

### **bag** — Messages of a rosbag

BagSelection object

Messages contained within a rosbag, specified as a BagSelection 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: "MessageType", "/geometry\_msgs/Point"

### **MessageType** — ROS message type

string scalar | character vector | cell array

ROS message type, specified as a string scalar, character vector, or cell array. Multiple message types can be specified with a cell array.

### **Time** — Start and end times

*n*-by-2 matrix

Start and end times of the rosbag selection, specified as an *n*-by-2 vector.

### **Topic** — ROS topic name

string scalar | character vector | cell array

ROS topic name, specified as a string scalar, character vector, or cell array. Multiple topic names can be specified with a cell array.

## Output Arguments

### **bagSel** — Copy or subset of rosbag messages

BagSelection object

Copy or subset of rosbag messages, returned as a BagSelection object

## See Also

`readMessages` | `rosbag` | `timeseries`

**Introduced in R2015a**

# send

Publish ROS message to topic

## Syntax

```
send(pub,msg)
```

## Description

`send(pub,msg)` publishes a message to the topic specified by the publisher, `pub`. This message can be received by all subscribers in the ROS network that are subscribed to the topic specified by `pub`

## Examples

### Create, Send, And Receive ROS Messages

Set up a publisher and subscriber to send and receive a message on a ROS network.

Connect to a ROS network.

```
rosinit
```

```
Initializing ROS master on http://AH-SRADFORD:11311/.
```

```
Initializing global node /matlab_global_node_10876 with NodeURI http://AH-SRADFORD:6513
```

Create a publisher with a specific topic and message type. You can also return a default message to send using this publisher.

```
[pub,msg] = rospublisher('position','geometry_msgs/Point');
```

Modify the message before sending over the network.

```
msg.X = 1;
msg.Y = 2;
send(pub,msg);
```

Create a subscriber and wait for the latest message. Verify the message is the one you sent.

```
sub = rossubscriber('position')
pause(1);
sub.LatestMessage
```

```
sub =
```

```
Subscriber with properties:
```

```
    TopicName: '/position'
    MessageType: 'geometry_msgs/Point'
    LatestMessage: [0x1 Point]
    BufferSize: 1
    NewMessageFcn: []
```

```
ans =
```

```
ROS Point message with properties:
```

```
    MessageType: 'geometry_msgs/Point'
         X: 1
         Y: 2
         Z: 0
```

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

Shut down ROS network.

```
roshutdown
```

```
Shutting down global node /matlab_global_node_10876 with NodeURI http://AH-SRADFORD:65  
Shutting down ROS master on http://AH-SRADFORD:11311/.
```

## Input Arguments

### **pub** — ROS publisher

Publisher object handle

ROS publisher, specified as a `Publisher` object handle. You can create the object using `rospublisher`.

### **msg** — ROS message

Message object handle

ROS message, specified as a `Message` object handle.

## See Also

`receive` | `rosmesssage` | `rospublisher` | `rossubscriber` | `rostopic`

## Topics

“Exchange Data with ROS Publishers and Subscribers”

## Introduced in R2015a

# sendGoal

Send goal message to action server

## Syntax

```
sendGoal(client,goalMsg)
```

## Description

`sendGoal(client,goalMsg)` sends a goal message to the action server. The specified action client tracks this goal. The function does not wait for the goal to be executed and returns immediately.

If the `ActionFcn`, `FeedbackFcn`, and `ResultFcn` callbacks of the client are defined, they are called when the goal is processing on the action server. All callbacks associated with a previously sent goal are disabled, but the previous goal is not canceled.

## Examples

### Create And Send A ROS Action Goal Message

This example shows how to create goal messages and send to an already active ROS action server on a ROS network. You must create a ROS action client to connect to this server. To run the action server, this command is used on the ROS distribution:

```
roslaunch turtlebot_actions server_turtlebot_move.launch
```

Afterwards, connect to the ROS node using `rosinit` with the correct IP address.

Create a ROS action client and get a goal message. The `actClient` object connects to the already running ROS action server. `goalMsg` is a valid goal message. Update the message parameters with your specific goal.



```
[actClient, goalMsg] = roactionclient('/turtlebot_move');  
disp(goalMsg)
```

ROS TurtlebotMoveGoal message with properties:

```
    MessageType: 'turtlebot_actions/TurtlebotMoveGoal'  
    TurnDistance: 0  
    ForwardDistance: 0
```

Use `showdetails` to show the contents of the message

You can also create a message using `rosmesssage` and the action client object. This message sends linear and angular velocity commands to a Turtlebot® robot.

```
goalMsg = rosmesssage(actClient);  
disp(goalMsg)
```

ROS TurtlebotMoveGoal message with properties:

```
    MessageType: 'turtlebot_actions/TurtlebotMoveGoal'  
    TurnDistance: 0  
    ForwardDistance: 0
```

Use `showdetails` to show the contents of the message

Modify the goal message parameters and send the goal to the action server.

```
goalMsg.ForwardDistance = 2;  
sendGoal(actClient,goalMsg)
```

## Send and Cancel ROS Action Goals

Send and cancel goals for ROS actions. First, setup a ROS action client. Then send a goal message with modified parameters. Finally, cancel your goal and all goals on the action server.

Connect to a ROS network with a specified IP address. Create a ROS action client connected using `roactionclient`. Specify the action name. Wait for the client to be connected to the server.

```
rosinit('192.168.154.131')
[actClient,goalMsg] = rosactionclient('/fibonacci');
waitForServer(actClient);
```

Initializing global node /matlab\_global\_node\_40739 with NodeURI http://192.168.154.1:5

Send a goal message with modified parameters. Wait for the goal to finish executing.

```
goalMsg.Order = 4;
sendGoalAndWait(actClient,goalMsg)
```

Goal active

Feedback:

Sequence : [0, 1, 1]

Feedback:

Sequence : [0, 1, 1, 2]

Feedback:

Sequence : [0, 1, 1, 2, 3]

Feedback:

Sequence : [0, 1, 1, 2, 3, 5]

ans =

ROS FibonacciResult message with properties:

MessageType: 'actionlib\_tutorials/FibonacciResult'

Sequence: [6×1 int32]

Use showdetails to show the contents of the message

Send a new goal message without waiting.

```
goalMsg.Order = 5;
sendGoal(actClient,goalMsg)
```

Cancel the goal on the ROS action client, actClient.

```
cancelGoal(actClient)
```

Cancel all the goals on the action server that actClient is connected to.

```
cancelAllGoals(actClient)
```

Delete the action client.

```
delete(actClient)
```

Disconnect from the ROS network.

```
roshutdown
```

```
Shutting down global node /matlab_global_node_40739 with NodeURI http://192.168.154.1:5
```

## Input Arguments

### **client** — ROS action client

SimpleActionClient object handle

ROS action client, specified as a SimpleActionClient object handle. This simple action client enables you to track a single goal at a time.

### **goalMsg** — ROS action goal message

Message object handle

ROS action goal message, specified as a Message object handle. Update this message with your goal details and send it to the ROS action client using sendGoal or sendGoalAndWait.

## See Also

[cancelGoal](#) | [rosaction](#) | [rosactionclient](#) | [sendGoalAndWait](#)

## Topics

“ROS Actions Overview”

“Move a Turtlebot Robot Using ROS Actions”

**Introduced in R2016b**

# sendGoalAndWait

Send goal message and wait for result

## Syntax

```
resultMsg = sendGoalAndWait(client,goalMsg)
resultMsg = sendGoalAndWait(client,goalMsg,timeout)
[resultMsg,state,status] = sendGoalAndWait( ___ )
```

## Description

`resultMsg = sendGoalAndWait(client,goalMsg)` sends a goal message using the specified action client to the action server and waits until the action server returns a result message. Press **Ctrl+C** to abort the wait.

`resultMsg = sendGoalAndWait(client,goalMsg,timeout)` specifies a timeout period in seconds. If the server does not return the result in the timeout period, the function displays an error.

`[resultMsg,state,status] = sendGoalAndWait( ___ )` returns the final goal state and associated status text using any of the previous syntaxes. `state` contains information about where the goal execution succeeded or not.

## Examples

### Send and Cancel ROS Action Goals

Send and cancel goals for ROS actions. First, setup a ROS action client. Then send a goal message with modified parameters. Finally, cancel your goal and all goals on the action server.

Connect to a ROS network with a specified IP address. Create a ROS action client connected using `rosactionclient`. Specify the action name. Wait for the client to be connected to the server.

```
rosinit('192.168.154.131')
[actClient,goalMsg] = rosactionclient('/fibonacci');
waitForServer(actClient);
```

Initializing global node /matlab\_global\_node\_40739 with NodeURI http://192.168.154.1:5

Send a goal message with modified parameters. Wait for the goal to finish executing.

```
goalMsg.Order = 4;
sendGoalAndWait(actClient,goalMsg)
```

Goal active

Feedback:

Sequence : [0, 1, 1]

Feedback:

Sequence : [0, 1, 1, 2]

Feedback:

Sequence : [0, 1, 1, 2, 3]

Feedback:

Sequence : [0, 1, 1, 2, 3, 5]

ans =

ROS FibonacciResult message with properties:

MessageType: 'actionlib\_tutorials/FibonacciResult'

Sequence: [6×1 int32]

Use showdetails to show the contents of the message

Send a new goal message without waiting.

```
goalMsg.Order = 5;
sendGoal(actClient,goalMsg)
```

Cancel the goal on the ROS action client, actClient.

```
cancelGoal(actClient)
```

Cancel all the goals on the action server that actClient is connected to.

```
cancelAllGoals(actClient)
```

Delete the action client.

`delete(actClient)`

Disconnect from the ROS network.

`roshutdown`

Shutting down global node /matlab\_global\_node\_40739 with NodeURI http://192.168.154.1:5

## Input Arguments

### **client** — ROS action client

SimpleActionClient object handle

ROS action client, specified as a SimpleActionClient object handle. This simple action client enables you to track a single goal at a time.

### **goalMsg** — ROS action goal message

Message object handle

ROS action goal message, specified as a Message object handle. Update this message with your goal details and send it to the ROS action client using `sendGoal` or `sendGoalAndWait`.

### **timeout** — Timeout period

scalar in seconds

Timeout period for receiving a result message, specified as a scalar in seconds. If the client does not receive a new result message in that time period, an error is displayed.

## Output Arguments

### **resultMsg** — Result message

ROS Message object

Result message, returned as a ROS Message object. The result message contains the result data sent by the action server. This data depends on the action type.

### **state** — Final goal state

character vector

Final goal state, returned as one of the following:

- 'pending' — Goal was received, but has not yet been accepted or rejected.
- 'active' — Goal was accepted and is running on the server.
- 'succeeded' — Goal executed successfully.
- 'preempted' — An action client canceled the goal before it finished executing.
- 'aborted' — The goal was aborted before it finished executing. The action server typically aborts a goal.
- 'rejected' — The goal was not accepted after being in the 'pending' state. The action server typically triggers this status.
- 'recalled' — A client canceled the goal while it was in the 'pending' state.
- 'lost' — An internal error occurred in the action client.

**status — Status text**

character vector

Status text that the server associated with the final goal state, returned as a character vector.

## See Also

`cancelGoal` | `rosaction` | `rosactionclient` | `sendGoal`

## Topics

“ROS Actions Overview”

“Move a Turtlebot Robot Using ROS Actions”

**Introduced in R2016b**

# sendTransform

Send transformation to ROS network

## Syntax

```
sendTransform(tftree,tf)
```

## Description

`sendTransform(tftree,tf)` broadcasts a transform or array of transforms, `tf`, to the ROS network as a `TransformationStamped` ROS message.

## Examples

### Send a Transformation to ROS Network

This example shows how to create a transformation and send it over the ROS network.

Create a ROS transformation tree. Use `roslint` to connect a ROS network. Replace `ipaddress` with your ROS network address.

```
ipaddress = '192.168.203.129';  
roslint(ipaddress)  
tftree = rostf;  
pause(2)
```

```
Initializing global node /matlab_global_node_69912 with NodeURI http://192.168.203.1:5
```

Verify the transformation you want to send over the network does not already exist. `canTransform` returns false if the transformation is not immediately available.

```
canTransform(tftree,'new_frame','base_link')
```

```
ans =
```



```
logical
```

```
0
```

Create a TransformStamped message. Populate the message fields with the transformation information.

```
tform = rosmesssage('geometry_msgs/TransformStamped');
tform.ChildFrameId = 'new_frame';
tform.Header.FrameId = 'base_link';
tform.Transform.Translation.X = 0.5;
tform.Transform.Rotation.Z = 0.75;
```

Send the transformation over the ROS network.

```
sendTransform(tftree,tform)
```

Verify the transformation is now on the ROS network

```
canTransform(tftree,'new_frame','base_link')
```

```
ans =
```

```
logical
```

```
1
```

Shut down the ROS network.

```
roshutdown
```

```
Shutting down global node /matlab_global_node_69912 with NodeURI http://192.168.203.1:
```

## Input Arguments

### **tftree** — ROS transformation tree

TransformationTree object handle

ROS transformation tree, specified as a TransformationTree object handle. You can create a transformation tree by calling the `rostopic` function.

### **tf — Transformations between coordinate frames**

TransformStamped object handle | array of object handles

Transformations between coordinate frames, returned as a TransformStamped object handle or as an array of object handles. Transformations are structured as a 3-D translation (3-element vector) and a 3-D rotation (quaternion).

### **See Also**

getTransform | transform

**Introduced in R2015a**

## set

Set value of ROS parameter or add new parameter

### Syntax

```
set(ptree,paramname,pvalue)  
set(ptree,namespace,pvalue)
```

### Description

`set(ptree,paramname,pvalue)` assigns the value `pvalue` to the parameter with the name `paramname`. This parameter is sent to the parameter tree `ptree`.

`set(ptree,namespace,pvalue)` assigns multiple values as a dictionary in `pvalue` under the specified namespace.

The following ROS data types are supported as values of parameters. For each ROS data type, the corresponding MATLAB data type is also listed.

- 32-bit integer — `int32`
- boolean — `logical`
- double — `double`
- strings — string scalar, `string`, or character vector, `char`
- list — cell array (`cell`)
- dictionary — structure (`struct`)

### Examples

#### Set and Get Parameter Value

Connect to ROS network.

```
rosinit
```

```
Initializing ROS master on http://AH-SRADFORD:11311/.  
Initializing global node /matlab_global_node_68286 with NodeURI http://AH-SRADFORD:6033/
```

Create ROS parameter tree. Set a double parameter. Get the parameter to verify it was set.

```
ptree = rosparam;  
set(ptree, 'DoubleParam', 1.0)  
get(ptree, 'DoubleParam')
```

```
ans =
```

```
1
```

Shut down ROS network.

```
rosshutdown
```

```
Shutting down global node /matlab_global_node_68286 with NodeURI http://AH-SRADFORD:6033/  
Shutting down ROS master on http://AH-SRADFORD:11311/.
```

### Set A Dictionary Of Parameter Values

Use structures to specify a dictionary of ROS parameters under a specific namespace.

Connect to a ROS network.

```
rosinit
```

```
Initializing ROS master on http://bat5742win64:64232/.  
Initializing global node /matlab_global_node_67361 with NodeURI http://bat5742win64:64232/
```

Create a dictionary of parameters values. This dictionary contains the information relevant to an image. Display the structure to verify values.

```
image = imread('peppers.png');  
  
pval.ImageWidth = size(image,1);  
pval.ImageHeight = size(image,2);
```

```
pval.ImageTitle = 'peppers.png';  
disp(pval)
```

```
    ImageWidth: 384  
    ImageHeight: 512  
    ImageTitle: 'peppers.png'
```

Set the dictionary of values using the desired namespace.

```
rosparam('set', 'ImageParam', pval)
```

Get the parameters using the namespace. Verify the values.

```
pval2 = rosparam('get', 'ImageParam')
```

```
pval2 = struct with fields:  
    ImageHeight: 512  
    ImageTitle: 'peppers.png'  
    ImageWidth: 384
```

Shutdown ROS network.

```
roshutdown
```

```
Shutting down global node /matlab_global_node_67361 with NodeURI http://bat5742win64:6  
Shutting down ROS master on http://bat5742win64:64232/.
```

## Input Arguments

### **ptree** — Parameter tree

ParameterTree object handle

Parameter tree, specified as a ParameterTree object handle. Create this object using the `rosparam` function.

### **paramname** — ROS parameter name

string scalar | character vector

ROS parameter name, specified as a string scalar or character vector. This string must match the parameter name exactly.

**pvalue — ROS parameter value or dictionary of values**`int32` | `logical` | `double` | string scalar | character vector | cell array | structure

ROS parameter value or dictionary of values, specified as a supported MATLAB data type.

The following ROS data types are supported as values of parameters. For each ROS data type, the corresponding MATLAB data type is also listed.

<b>ROS Data Type</b>	<b>MATLAB Data Type</b>
32-bit integer	<code>int32</code>
boolean	<code>logical</code>
double	<code>double</code>
string	string scalar, <code>string</code> , or character vector, <code>char</code>
list	cell array ( <code>cell</code> )
dictionary	structure ( <code>struct</code> )

**namespace — ROS parameter namespace**

string scalar | character vector

ROS parameter namespace, specified as a string scalar or character vector. All parameter names starting with this string are listed when calling `rosparam("list", namespace)`.

## Limitations

Base64-encoded binary data and iso8601 data from ROS are not supported.

## See Also

`get` | `rosparam`**Introduced in R2015a**

# show

**Package:** robotics

Visualize path segment

## Syntax

```
show(pathSeg)
show(pathSeg, Name, Value)
```

## Description

`show(pathSeg)` plots the path segment with start and goal positions and their headings.

`show(pathSeg, Name, Value)` also specifies `Name, Value` pairs to control display settings.

## Examples

### Connect Poses Using Dubins Connection Path

Create a `DubinsConnection` object.

```
dubConnObj = robotics.DubinsConnection;
```

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

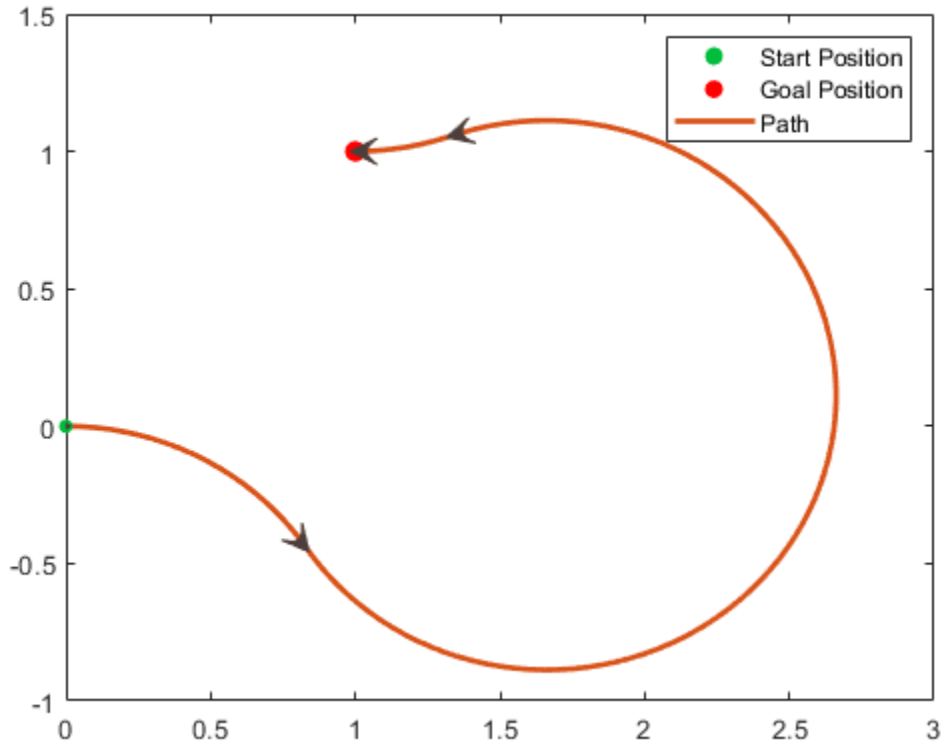
```
startPose = [0 0 0];
goalPose = [1 1 pi];
```

Calculate a valid path segment to connect the poses.

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

Show the generated path.

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



### Modify Connection Types for Reeds-Shepp Path

Create a ReedsSheppConnection object.

```
reedsConnObj = robotics.ReedsSheppConnection;
```

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

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

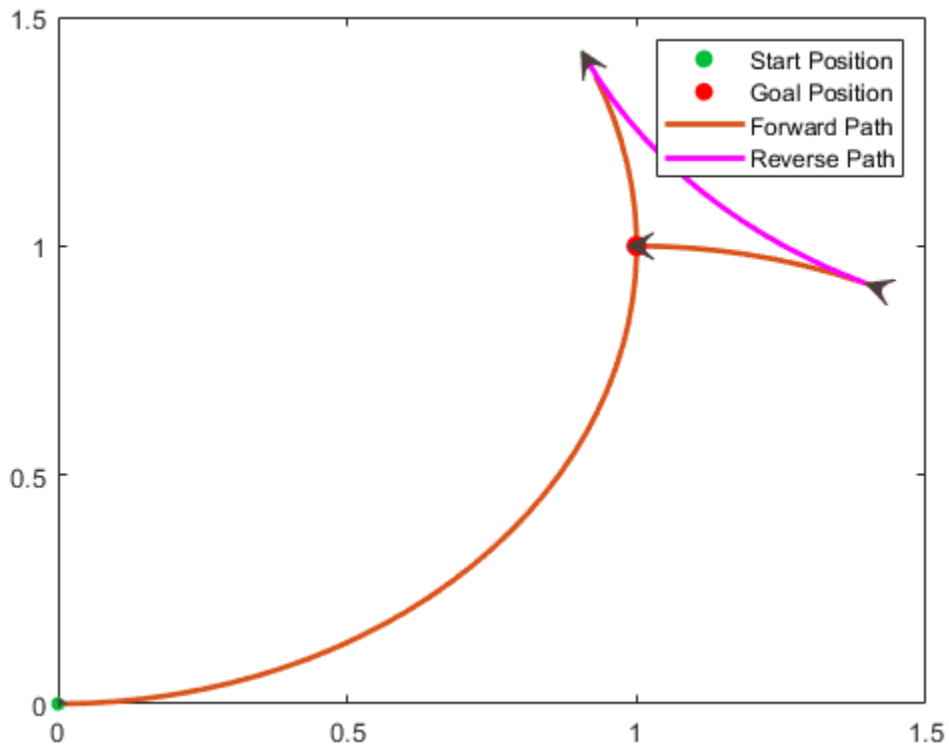


Calculate a valid path segment to connect the poses.

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

Show the generated path. Notice the direction of the turns.

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



```
pathSegObj{1}.MotionTypes
```

```
ans = 1x5 cell array
```

```
    {'L'}    {'R'}    {'L'}    {'N'}    {'N'}
```

```
pathSegObj{1}.MotionDirections
```

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

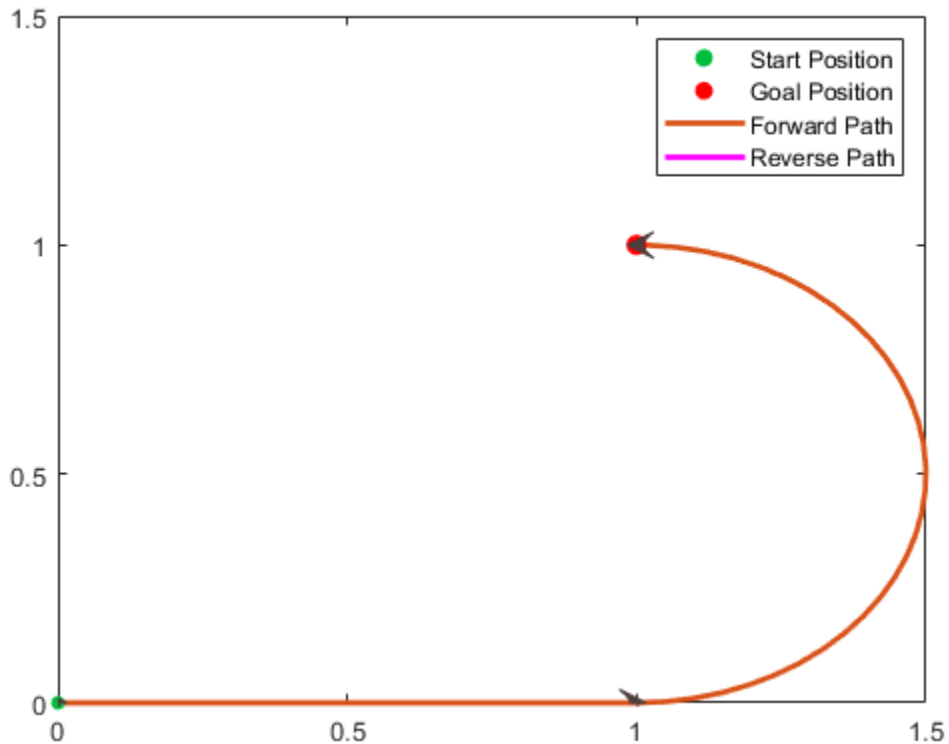
Disable this specific motion sequence in a new connection object. Reduce the `MinTurningRadius` if the robot is more maneuverable. Increase the reverse cost to reduce the likelihood of reverse directions being used. Connect the poses again to get a different path.

```
reedsConnObj = robotics.ReedsSheppConnection('DisabledPathTypes',{'LpRnLp'});
reedsConnObj.MinTurningRadius = 0.5;
reedsConnObj.ReverseCost = 5;
```

```
[pathSegObj,pathCosts] = connect(reedsConnObj,startPose,goalPose);
pathSegObj{1}.MotionTypes
```

```
ans = 1x5 cell array
      {'L'}      {'S'}      {'L'}      {'N'}      {'N'}
```

```
show(pathSegObj{1})
xlim([0 1.5])
ylim([0 1.5])
```



### Interpolate Poses For Dubins Path

Create a DubinsConnection object.

```
dubConnObj = robotics.DubinsConnection;
```

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

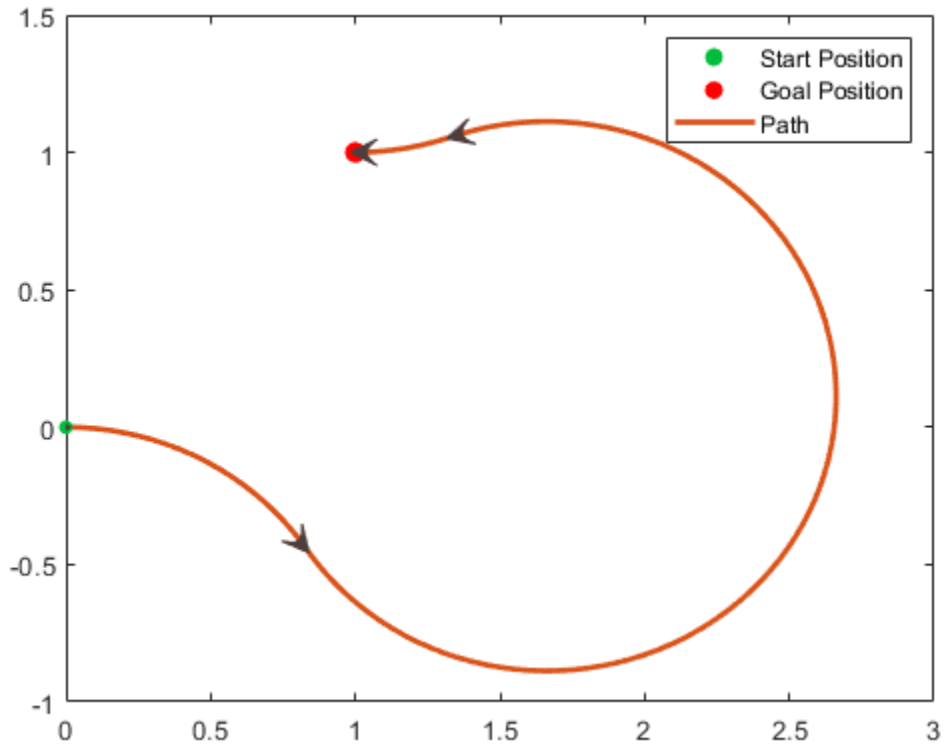
```
startPose = [0 0 0];  
goalPose = [1 1 pi];
```

Calculate a valid path segment to connect the poses.

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

Show the generated path.

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



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

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

```
poses = 32x3
```

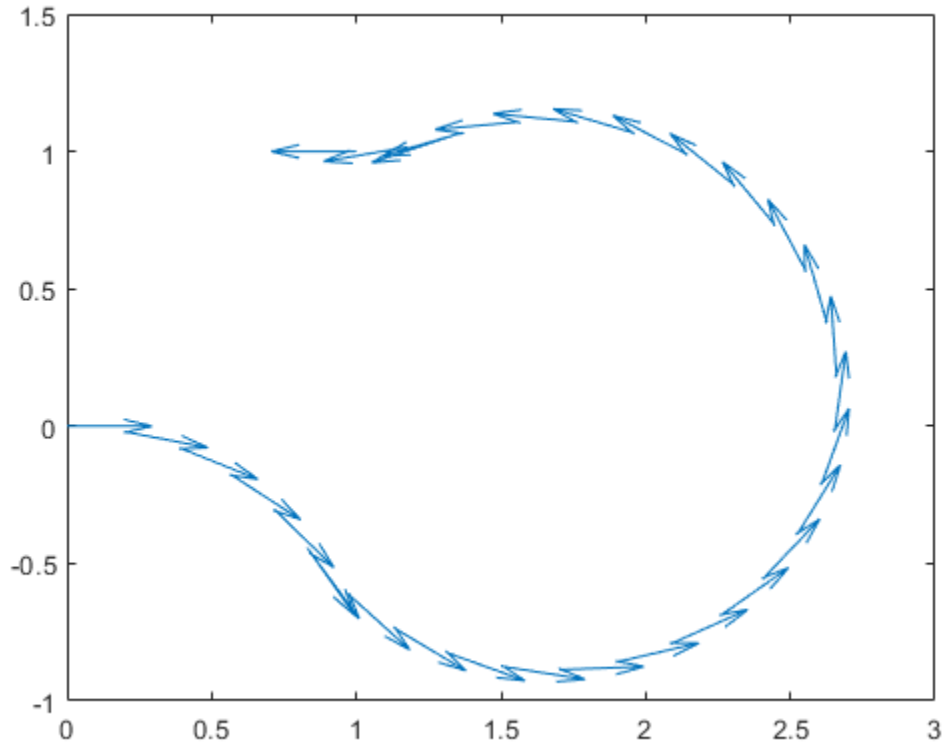
```
0 0 0
```

---

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

Use the `quiver` function to plot these poses.

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



## Input Arguments

### **pathSeg — Path segment**

`DubinsPathSegment` object | `ReedsSheppPathSegment` object

Path segment, specified as a `robotics.DubinsPathSegment` or `robotics.ReedsSheppPathSegment` object.

## Name-Value Pair Arguments

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

Example: `'Headings',{ 'transitions' }`

### Parent — Axes to plot path onto

Axes handle

Axes to plot path onto, specified as an Axes handle.

### Headings — Heading angles to display

cell array of character vector or string scalars

Heading angles to display, specified as a cell array of character vector or string scalars. Options are any combination of `'start'`, `'goal'`, and `'transitions'`. To disable all heading displays, specify `{ '' }`.

### Positions — Positions to display

`'both'` (default) | `'start'` | `'goal'` | `'none'`

Positions to display, specified as `'both'`, `'start'`, `'goal'`, or `'none'`. The start position is marked with green, and the goal position is marked with red.

## See Also

### Functions

`connect` | `interpolate`

### Objects

`robotics.DubinsConnection` | `robotics.DubinsPathSegment` |  
`robotics.ReedsSheppConnection` | `robotics.ReedsSheppPathSegment`

### Introduced in R2018b

# show

**Package:** robotics

Plot pose graph

## Syntax

```
show(poseGraph)
show(poseGraph,Name,Value)
axes = show( ___ )
```

## Description

`show(poseGraph)` plots the specified pose graph in a figure.

`show(poseGraph,Name,Value)` specifies options using `Name,Value` pair arguments. For example, `'IDs','on'` plots all node and edge IDs of the pose graph.

`axes = show( ___ )` returns the axes handle that the pose graph is plotted to using any of previous syntaxes.

## Examples

### Optimize a 2-D Pose Graph

Optimize a pose graph based on the nodes and edge constraints. The pose graph used in this example is from the Intel Research Lab Dataset and was generated from collecting wheel odometry and a laser range finder sensor information in an indoor lab.

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

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

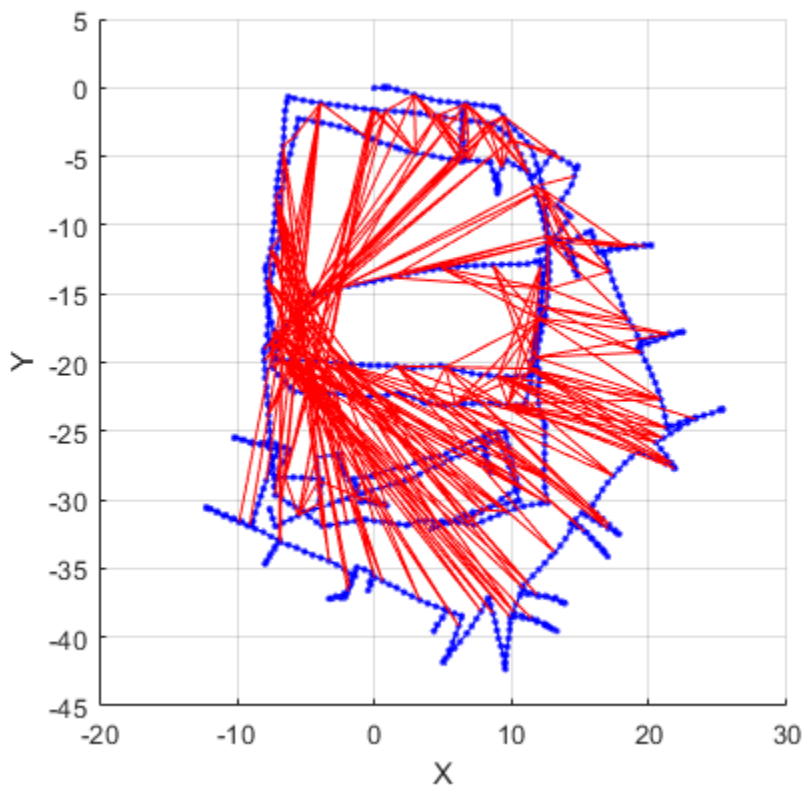


PoseGraph with properties:

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

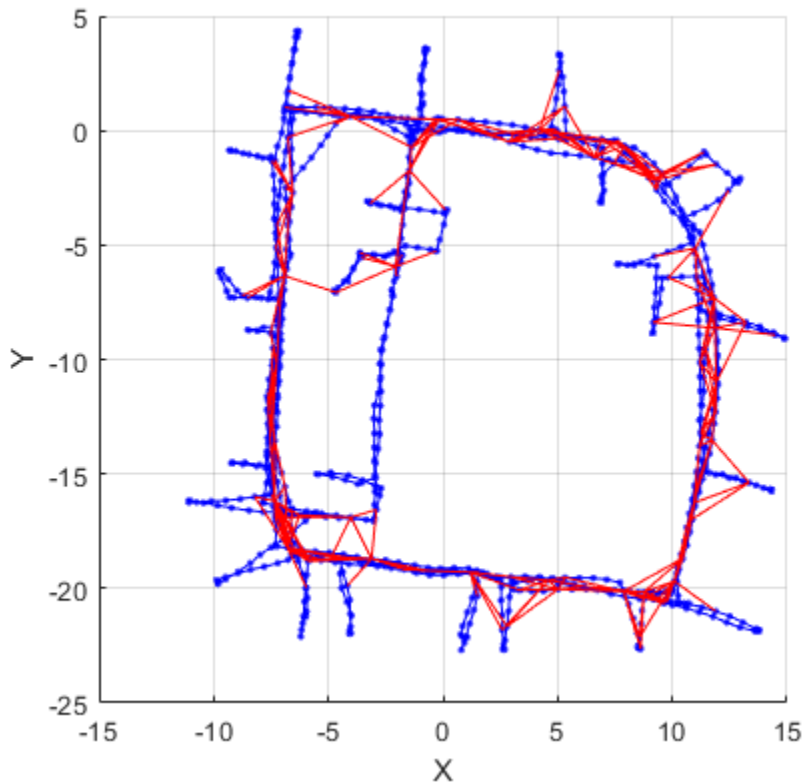
Plot the pose graph with IDs off. Red lines indicate loop closures identified in the dataset.

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



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

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



## Optimize a 3-D Pose Graph

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

Load the pose graph from the MIT dataset. Inspect the `robotics.PoseGraph3D` object to view the number of nodes and loop closures.

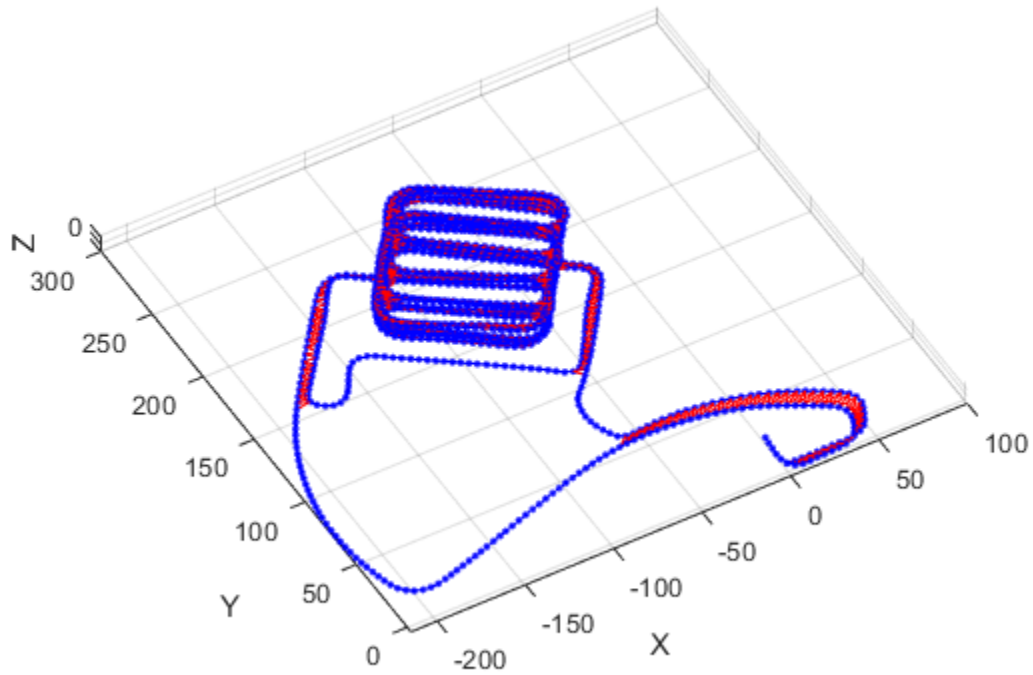
```
load parking-garage-posegraph.mat pg
disp(pg);
```

```
PoseGraph3D with properties:
```

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

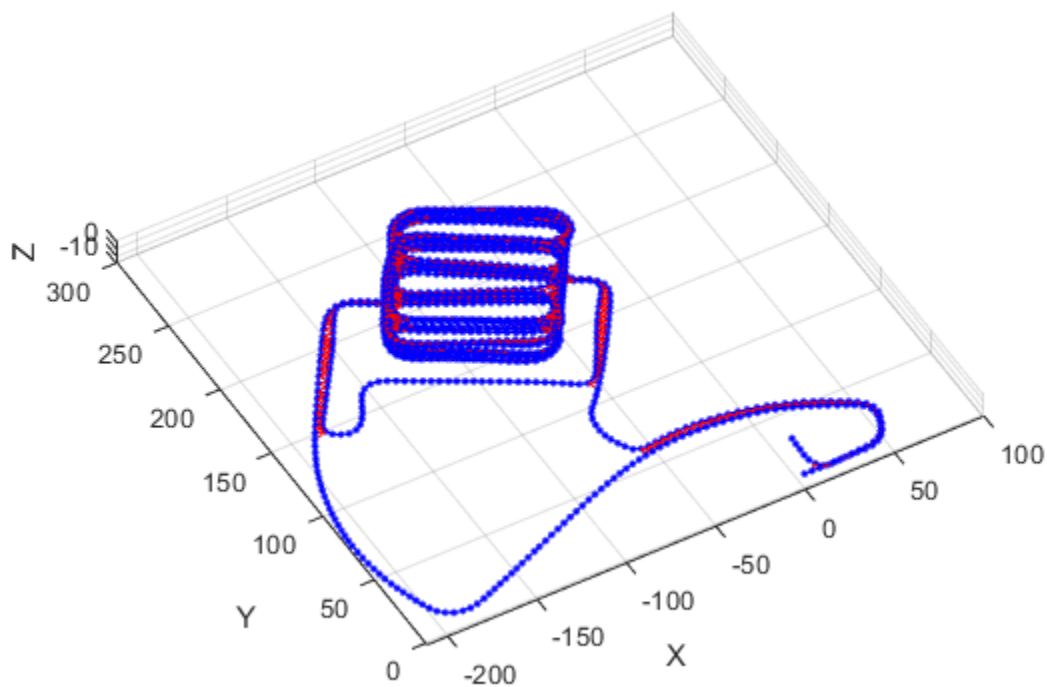
Plot the pose graph with IDs off. Red lines indicate loop closures identified in the dataset.

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



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

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



## Input Arguments

### **poseGraph** — Pose graph

PoseGraph object | PoseGraph3D object

Pose graph, specified as a PoseGraph or PoseGraph3D object.

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes.

You can specify several name and value pair arguments in any order as `Name1, Value1, . . . , NameN, ValueN`.

Example: `'IDs', 'off'`

### **Parent — Axes used to plot pose graph**

`Axes` object | `UIAxes` object

Axes used to plot the pose graph, specified as the comma-separated pair consisting of `'Parent'` and either an `Axes` or `UIAxes` object. See `axes` or `uiaxes`.

### **IDs — Display of IDs on pose graph**

`'loopclosures'` (default) | `'all'` | `'nodes'` | `'off'`

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

- `'all'` — Plot node and edge IDs.
- `'nodes'` — Plot node IDs.
- `'loopclosures'` — Plot loop closure edge IDs.
- `'off'` — Do not plot any IDs.

## **Output Arguments**

### **axes — Axes used to plot the map**

`Axes` object | `UIAxes` object

Axes used to plot the map, returned as either an `Axes` or `UIAxes` object. See `axes` or `uiaxes`.

## **See Also**

### **Functions**

`addRelativePose` | `optimizePoseGraph`

### **Objects**

`robotics.LidarSLAM` | `robotics.PoseGraph` | `robotics.PoseGraph3D`

## **Topics**

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

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

**Introduced in R2018a**

# show

**Package:** robotics

Display VectorFieldHistogram information in figure window

## Syntax

```
show(vfh)
```

```
show(vfh, 'Parent', parent)
```

```
h = show( ___ )
```

## Description

`show(vfh)` shows histograms calculated by the VFH+ algorithm in a figure window. The figure also includes the parameters of the `VectorFieldHistogram` object and range values from the last object call.

`show(vfh, 'Parent', parent)` sets the specified axes handle, `parent`, to the axes.

`h = show( ___ )` returns the figure object handle created by `show` using any of the arguments from the previous syntaxes.

## Examples

### Create a Vector Field Histogram Object and Visualize Data

This example shows how to calculate a steering direction based on input laser scan data.

Create a `VectorFieldHistogram` object.

```
vfh = robotics.VectorFieldHistogram;
```

Input laser scan data and target direction.



```
ranges = 10*ones(1,500);  
ranges(1,225:275) = 1.0;  
angles = linspace(-pi,pi,500);  
targetDir = 0;
```

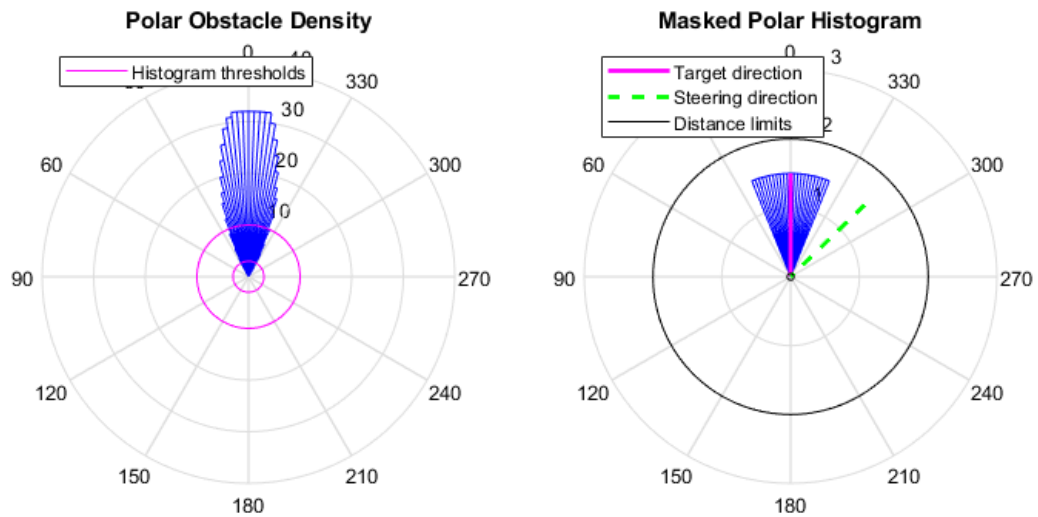
Compute an obstacle-free steering direction.

```
steeringDir = vfh(ranges,angles,targetDir)
```

```
steeringDir = -0.8014
```

Visualize the VectorFieldHistogram computation.

```
h = figure;  
set(h,'Position',[50 50 800 400])  
show(vfh)
```



# Input Arguments

### **vfh** — Vector field histogram algorithm

VectorFieldHistogram object

Vector field histogram algorithm, specified as a VectorFieldHistogram object. This object contains all the parameters for tuning the VFH+ algorithm.

### **parent** — Axes properties

handle

Axes properties, specified as a handle.

# Output Arguments

### **h** — Axes handles for VFH algorithm display

Axes array

Axes handles for VFH algorithm display, specified as an Axes array. The VFH histogram and HistogramThresholds are shown in the first axes. The binary histogram, range sensor readings, target direction, and steering directions are shown in the second axes.

# See Also

robotics.VectorFieldHistogram

**Introduced in R2015b**

# showdetails

**Package:** robotics

Display details of imported robot

## Syntax

```
showdetails(importInfo)
```

## Description

`showdetails(importInfo)` displays the details of each body in the `RigidBodyTree` object that is created from calling `importrobot`. Specify the `robotics.RigidBodyTreeImportInfo` object from the import process.

The list shows the bodies with their associated joint name, joint type, source blocks, parent body name, and children body names. The list also provides highlight links to the associated blocks used in the Simscape Multibody model.

---

**Note** The Highlight links assume the block names are unchanged.

---

## Input Arguments

### **importInfo** — Robot import information

`RigidBodyTreeImportInfo` object

Robot import information, specified as a `robotics.RigidBodyTreeImportInfo` object. This object is returned when you use `importrobot`.

## See Also

`importrobot` | `robotics.RigidBodyTree` | `robotics.RigidBodyTreeImportInfo`

**Introduced in R2018b**

# showdetails

Display all ROS message contents

## Syntax

```
details = showdetails(msg)
```

## Description

`details = showdetails(msg)` gets all data contents of message object `msg`. The details are stored in `details` or displayed on the command line.

## Examples

### Create Message and View Details

Create a message. Populate the message with data using the relevant properties.

```
msg = rosmessage('geometry_msgs/Point');  
msg.X = 1;  
msg.Y = 2;  
msg.Z = 3;
```

View the message details.

```
showdetails(msg)
```

```
X : 1  
Y : 2  
Z : 3
```

## Input Arguments

**msg** — ROS message

Message object handle

ROS message, specified as a Message object handle.

## Output Arguments

**details** — Details of ROS message

character vector

Details of ROS message, returned as a character vector.

## See Also

rosmessage

**Introduced in R2015a**

# 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 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 =  
    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 =  
    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 =  
    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 =
```

```
Columns 1 through 7
```

```
9.0000  9.0000  9.0000  9.0000  9.0000  9.0000  9.0000
```

```
Columns 8 through 10
```

```
9.0000  9.0000  9.0000
```

## 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');
```

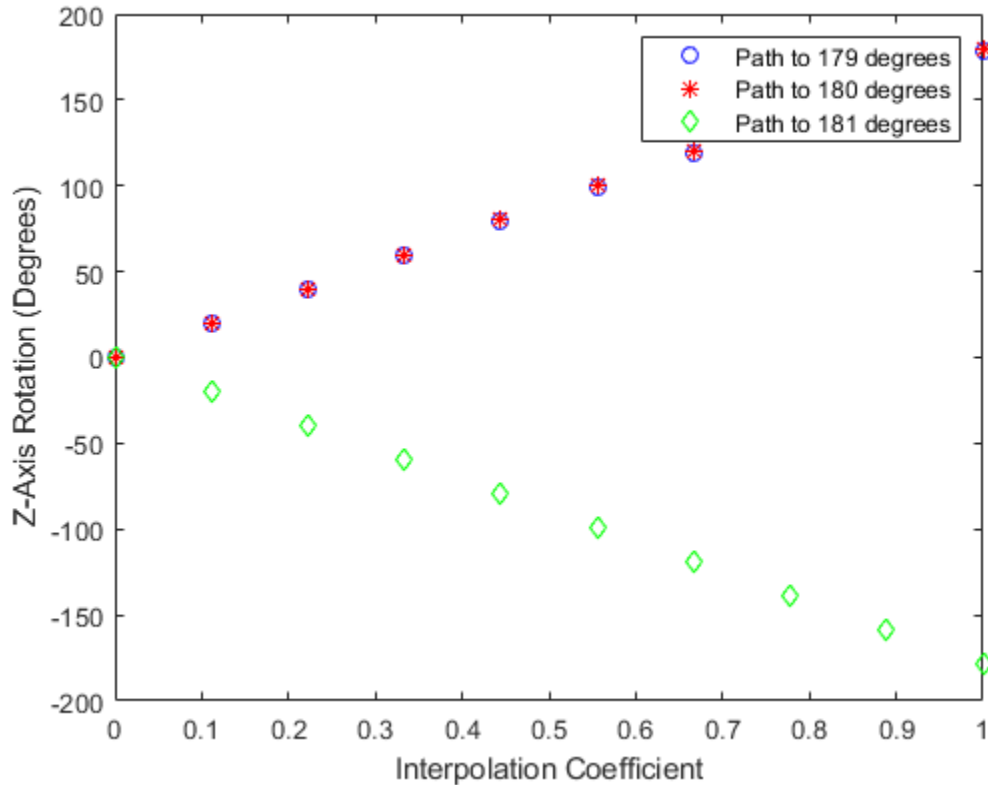
```
legend('Path to 179 degrees', ...
```

```
       '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.

q1, q2, and T must have compatible sizes. In the simplest cases, they can be the same size or any one can be a scalar. Two inputs have compatible sizes if, for every dimension, the dimension sizes of the inputs are either the same or one of them is 1.

Data Types: quaternion

### **q2 — Quaternion**

scalar | vector | matrix | multidimensional array

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

q1, q2, and T must have compatible sizes. In the simplest cases, they can be the same size or any one can be a scalar. Two inputs have compatible sizes if, for every dimension, the dimension sizes of the inputs are either the same or one of the dimension sizes is 1.

Data Types: quaternion

### **T — Interpolation coefficient**

scalar | vector | matrix | multidimensional array

Interpolation coefficient, specified as a scalar, vector, matrix, or multidimensional array of numbers with each element in the range [0,1].

q1, q2, and T must have compatible sizes. In the simplest cases, they can be the same size or any one can be a scalar. Two inputs have compatible sizes if, for every dimension, the dimension sizes of the inputs are either the same or one of the dimension sizes is 1.

Data Types: single | double

## **Output Arguments**

### **q0 — Interpolated quaternion**

scalar | vector | matrix | multidimensional array

Interpolated quaternion, returned as a scalar, vector, matrix, or multidimensional array.

Data Types: quaternion

## Algorithms

Quaternion **spherical linear interpolation** (SLERP) is an extension of linear interpolation along a plane to spherical interpolation in three dimensions. The algorithm was first proposed in [1] (Sensor Fusion and Tracking Toolbox). Given two quaternions,  $q_1$  and  $q_2$ , SLERP interpolates a new quaternion,  $q_0$ , along the great circle that connects  $q_1$  and  $q_2$ . The interpolation coefficient,  $T$ , determines how close the output quaternion is to either  $q_1$  and  $q_2$ .

The SLERP algorithm can be described in terms of sinusoids:

$$q_0 = \frac{\sin((1 - T)\theta)}{\sin(\theta)} q_1 + \frac{\sin(T\theta)}{\sin(\theta)} q_2$$

where  $q_1$  and  $q_2$  are normalized quaternions, and  $\theta$  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™.

**Introduced in R2018b**

# state

**Package:** robotics

UAV state vector

## Syntax

```
stateVec = state(uavGuidanceModel)
```

## Description

---

**Note** This function requires you to install the UAV Library for Robotics System Toolbox. To install add-ons, use `roboticsAddons` and select the desired add-on.

---

`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.

**Note:** To use UAV algorithms, you must install the UAV Library for Robotics System Toolbox®. To install, use `roboticsAddons`.

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 multicopter.

```
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 fixed-wing UAV states as a 13-by- $n$  matrix.

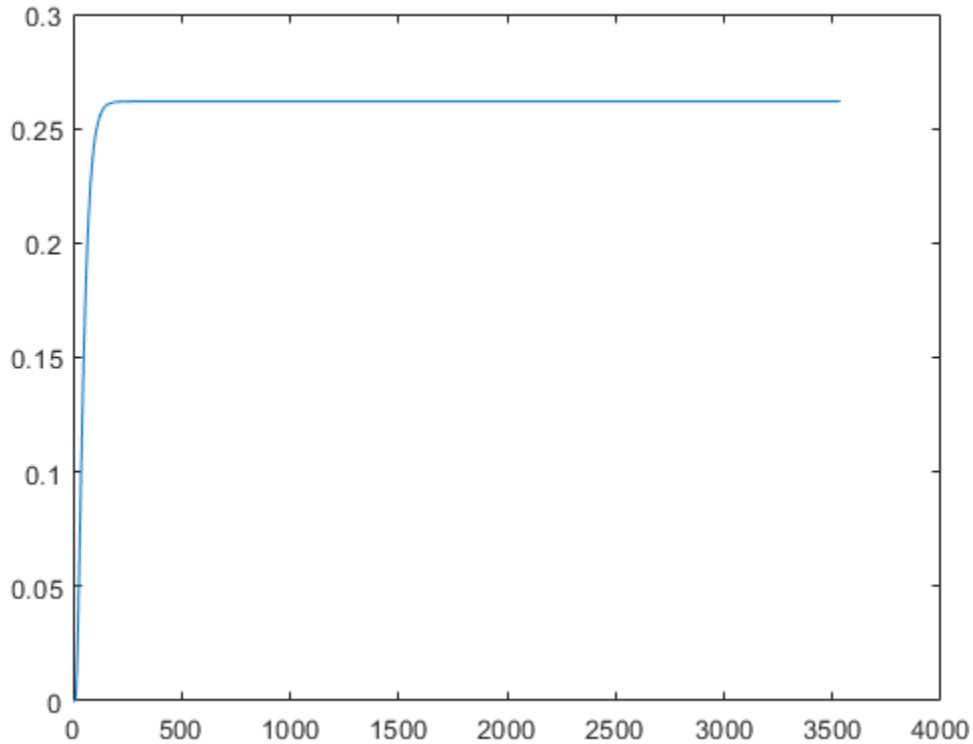
```
simOut = ode45(@(~,x)derivative(model,x,u,e), [0 3], s);  
size(simOut.y)
```

```
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.

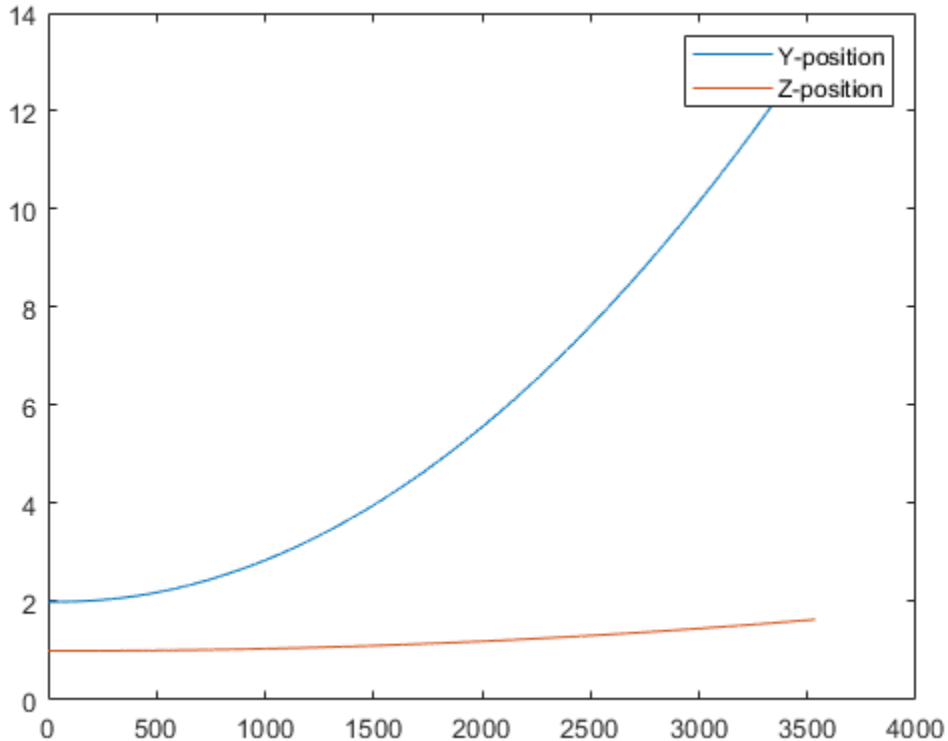
```
plot(simOut.y(9,:))
```



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.

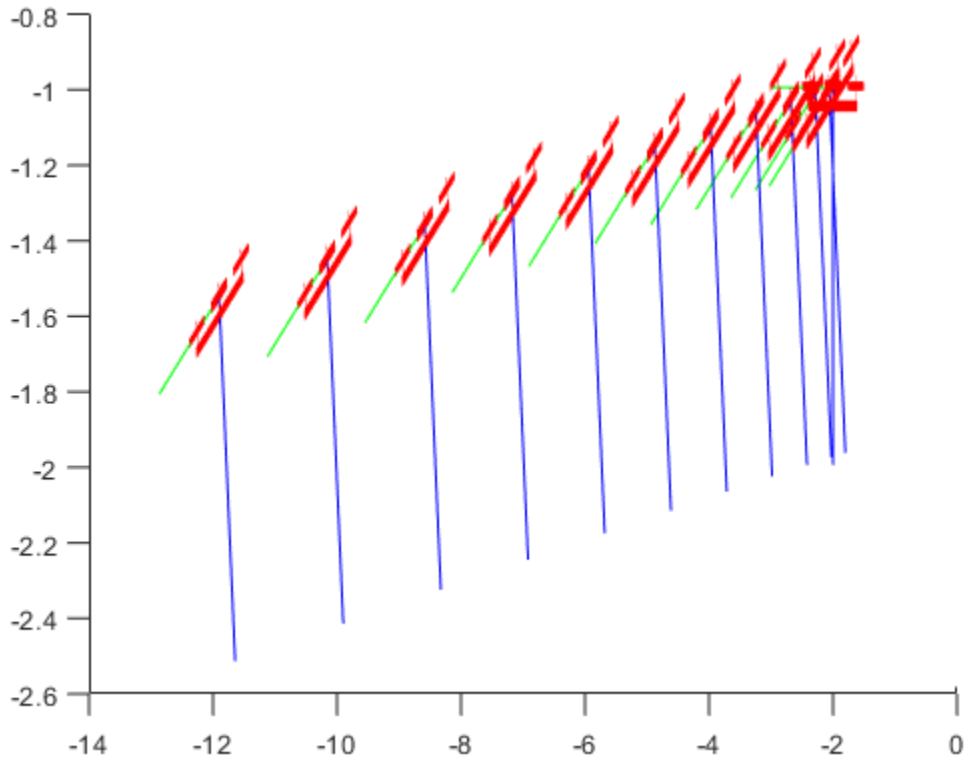
```
figure
plot(simOut.y(2,:));
hold on
plot(simOut.y(3,:));
legend('Y-position', 'Z-position')
hold off
```





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])
```



### 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.

**Note:** To use UAV algorithms, you must install the UAV Library for Robotics System Toolbox®. To install, use `roboticsAddons`.

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; % 10 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.

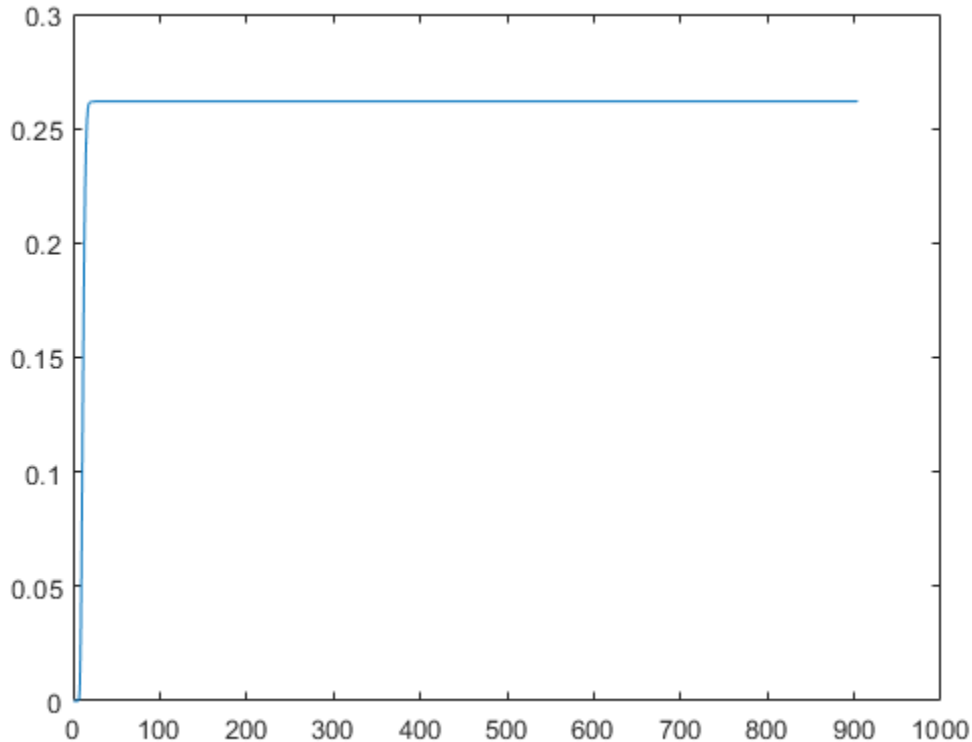
```
simOut = ode45(@(~,x)derivative(model,x,u,e), [0 50], s);  
size(simOut.y)
```

```
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,:))
```



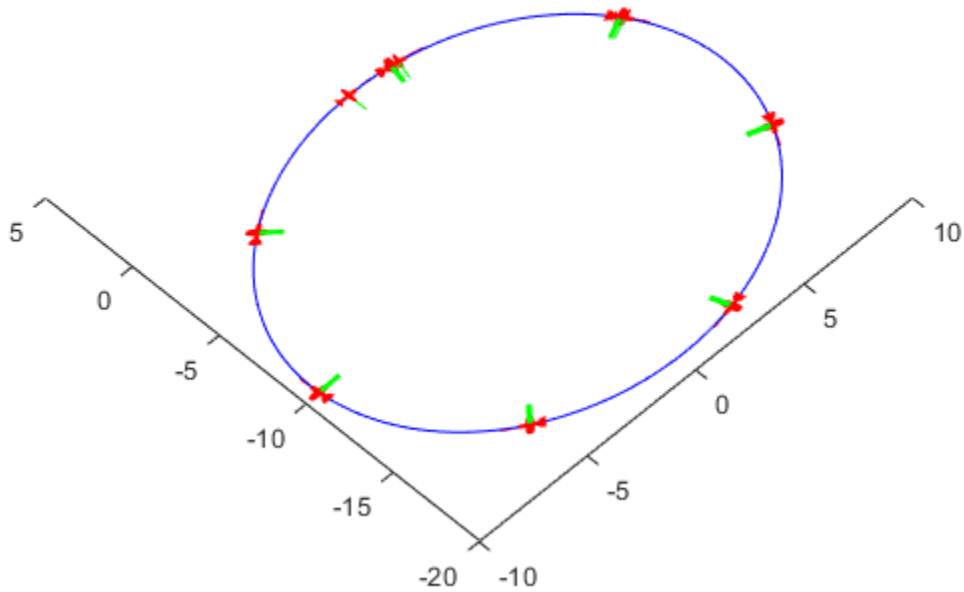
You can also plot the fixed-wing trajectory using `plotTransforms`. Create the translation and rotation vectors from the simulated state. Downsample (every 30th element) and transpose the `simOut` elements, and convert the Euler angles to quaternions. Specify the mesh as the `fixedwing.stl` file and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

### **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 per second.
- **FlightPathAngle** - Angle between ground velocity and north-east plane in meters per second.
- **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 multicopter 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** - [r p q] 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 | roboticsAddons  
| state

### Objects

fixedwing | multicopter

### 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

# stopCore

Stop ROS core

## Syntax

```
stopCore(device)
```

## Description

`stopCore(device)` stops the ROS core on the specified `rosdevice`, `device`. If multiple ROS cores are running on the ROS device, the function stops all of them. If no core is running, the function returns immediately.

## Examples

### Run ROS Core on ROS Device

Connect to a remote ROS device and start a ROS core. The ROS core is needed to run ROS nodes to communicate via a ROS network. You can run and stop a ROS core or node and check their status using a `rosdevice` object.

Create a connection to a ROS device. Specify the address, user name, and password of your specific ROS device. The device contains information about the ROS device, including the available ROS nodes that can be run using `runNode`.

```
ipaddress = '192.168.154.131';  
d = rosdevice(ipaddress, 'user', 'password')
```

```
d =
```

```
rosdevice with properties:
```

```
DeviceAddress: '192.168.154.131'
```



```
Username: 'user'  
ROSFolder: '/opt/ros/hydro'  
CatkinWorkspace: '~/catkin_ws_test'  
AvailableNodes: {'robotcontroller' 'robotcontroller2'}
```

Run a ROS core and check if it is running.

```
runCore(d)  
running = isCoreRunning(d)
```

```
running =  
  logical  
  1
```

Stop the ROS core and confirm that it is no longer running.

```
stopCore(d)  
running = isCoreRunning(d)
```

```
running =  
  logical  
  0
```

## Input Arguments

### **device** — ROS device

rosdevice object

ROS device, specified as a rosdevice object.

## See Also

isCoreRunning | rosdevice | runCore

## **Topics**

“Generate a Standalone ROS Node from Simulink®”

**Introduced in R2016b**

# statistics

Statistics of past execution periods

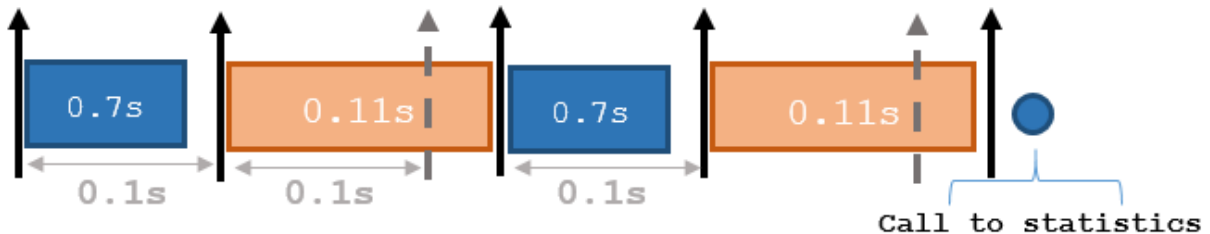
## Syntax

```
stats = statistics(rate)
```

## Description

`stats = statistics(rate)` returns statistics of previous periods of code execution. `stats` is a struct with these fields: `Periods`, `NumPeriods`, `AveragePeriod`, `StandardDeviation`, and `NumOverruns`.

Here is a sample execution graphic using the default setting, 'slip', for the `OverrunAction` property in the `Rate` object. See `OverrunAction` for more information on overrun code execution.



The output of `statistics` is:

```
stats =  
    Periods: [0.7 0.11 0.7 0.11]  
    NumPeriods: 4  
    AveragePeriod: 0.09  
    StandardDeviation: 0.0231  
    NumOverruns: 2
```

# Input Arguments

### rate — Rate object

handle

Rate object, specified as an object handle. This object contains the information for the `DesiredRate` and other info about the execution. See `robotics.Rate` for more information.

# Output Arguments

### stats — Time execution statistics

structure

Time execution statistics, returned as a structure. This structure contains the following fields:

- `Period` — All time periods (returned in seconds) used to calculate statistics as an indexed array. `stats.Period(end)` is the most recent period.
- `NumPeriods` — Number of elements in `Periods`
- `AveragePeriod` — Average time in seconds
- `StandardDeviation` — Standard deviation of all periods in seconds, centered around the mean stored in `AveragePeriod`
- `NumOverruns` — Number of periods with overrun

# Examples

### Get Statistics From Rate Object Execution

Create a Rate object for running at 20 Hz.

```
r = robotics.Rate(2);
```

Start a loop and control operation using the Rate object.

```
for i = 1:30  
    % Your code goes here
```

```
    waitfor(r);  
end
```

Get Rate object statistics after loop operation.

```
stats = statistics(r)  
  
stats = struct with fields:  
    Periods: [1x30 double]  
    NumPeriods: 30  
    AveragePeriod: 0.5000  
    StandardDeviation: 4.7101e-04  
    NumOverruns: 0
```

## See Also

[robotics.Rate](#) | [rosclock](#) | [waitfor](#)

## Topics

“Execute Code at a Fixed-Rate”

**Introduced in R2016a**

# stopNode

Stop ROS node

## Syntax

```
stopNode(device,modelName)
```

## Description

`stopNode(device,modelName)` stops a running ROS node running that was deployed from a Simulink model named `modelName`. The node is running on the specified `rosdevice` object, `device`. If the node is not running, the function immediately.

## Examples

### Run ROS Node on ROS Device

Connect to a remote ROS device and start a ROS node. First, run a ROS core so that ROS nodes can communicate via a ROS network. You can run and stop a ROS core or node and check their status using a `rosdevice` object.

Create a connection to a ROS device. Specify the address, user name and password of your specific ROS device. The device contains information about the ROS device, including the available ROS nodes that can be run using `runNode`.

```
ipaddress = '192.168.203.129';  
d = rosdevice(ipaddress,'user','password');  
d.ROSFolder = '/opt/ros/hydro';  
d.CatkinWorkspace = '~/catkin_ws_test'
```

```
d =
```

```
rosdevice with properties:
```

```
DeviceAddress: '192.168.203.129'  
Username: 'user'  
ROSFolder: '/opt/ros/hydro'  
CatkinWorkspace: '~/catkin_ws_test'  
AvailableNodes: {'robotcontroller' 'robotcontroller2'}
```

Run a ROS core. Connect MATLAB® to the ROS master using `rosinit`. This core enables you to run ROS nodes on your ROS device.

```
runCore(d)  
rosinit(d.DeviceAddress,11311)
```

```
Initializing global node /matlab_global_node_12272 with NodeURI http://192.168.203.1:6
```

Check the available ROS nodes on the connected ROS device. These nodes were generated from Simulink® models following the process in the “Generate a Standalone ROS Node from Simulink®” example.

```
d.AvailableNodes
```

```
ans =
```

```
1×2 cell array
```

```
 {'robotcontroller'}    {'robotcontroller2'}
```

Run a ROS node. specifying the node name. Check if the node is running.

```
runNode(d, 'robotcontroller')  
running = isNodeRunning(d, 'robotcontroller')
```

```
running =
```

```
logical
```

```
1
```

Stop the ROS node. Disconnect from the ROS network. Stop the ROS core.

```
stopNode(d, 'robotcontroller')
rosshutdown
stopCore(d)
```

```
Shutting down global node /matlab_global_node_12272 with NodeURI http://192.168.203.1:
```

### Run Multiple ROS Nodes

Run multiple ROS nodes on a connected ROS device. ROS nodes can be generated using Simulink® models to perform different tasks on the ROS network. These nodes are then deployed on a ROS device and can be run independently of Simulink®.

This example uses two different Simulink models that have been deployed as ROS nodes. See “Generate a Standalone ROS Node from Simulink®” and follow the instructions to generate and deploy a ROS node. The 'robotcontroller' node sends velocity commands to a robot to navigate it to a given point. The 'robotcontroller2' node uses the same model, but doubles the linear velocity to drive the robot faster.

Create a connection to a ROS device. Specify the address, user name, and password of your specific ROS device. The device contains information about the ROS device, including the available ROS nodes that can be run using runNode.

```
ipaddress = '192.168.203.129';
d = rosdevice(ipaddress, 'user', 'password')
```

```
d =
```

```
rosdevice with properties:
```

```
DeviceAddress: '192.168.203.129'
Username: 'user'
ROSFolder: '/opt/ros/indigo'
CatkinWorkspace: '~/catkin_ws_test'
AvailableNodes: {'robotcontroller' 'robotcontroller2'}
```

Run a ROS core. The ROS Core is the master enables you to run ROS nodes on your ROS device. Connect MATLAB® to the ROS master using rosinit. For this example, the port is set to 11311. rosinit can automatically select a port for you without specifying this input.



```
runCore(d)
rosinit(d.DeviceAddress,11311)
```

```
Initializing global node /matlab_global_node_15972 with NodeURI http://192.168.203.1:5
```

Check the available ROS nodes on the connected ROS device. The nodes listed in this example were generated from Simulink® models following the process in the “Generate a Standalone ROS Node from Simulink®” example. Two separate nodes are generated, 'robotcontroller' and 'robotcontroller2', which have the linear velocity set to 1 and 2 in the model respectively.

```
d.AvailableNodes
```

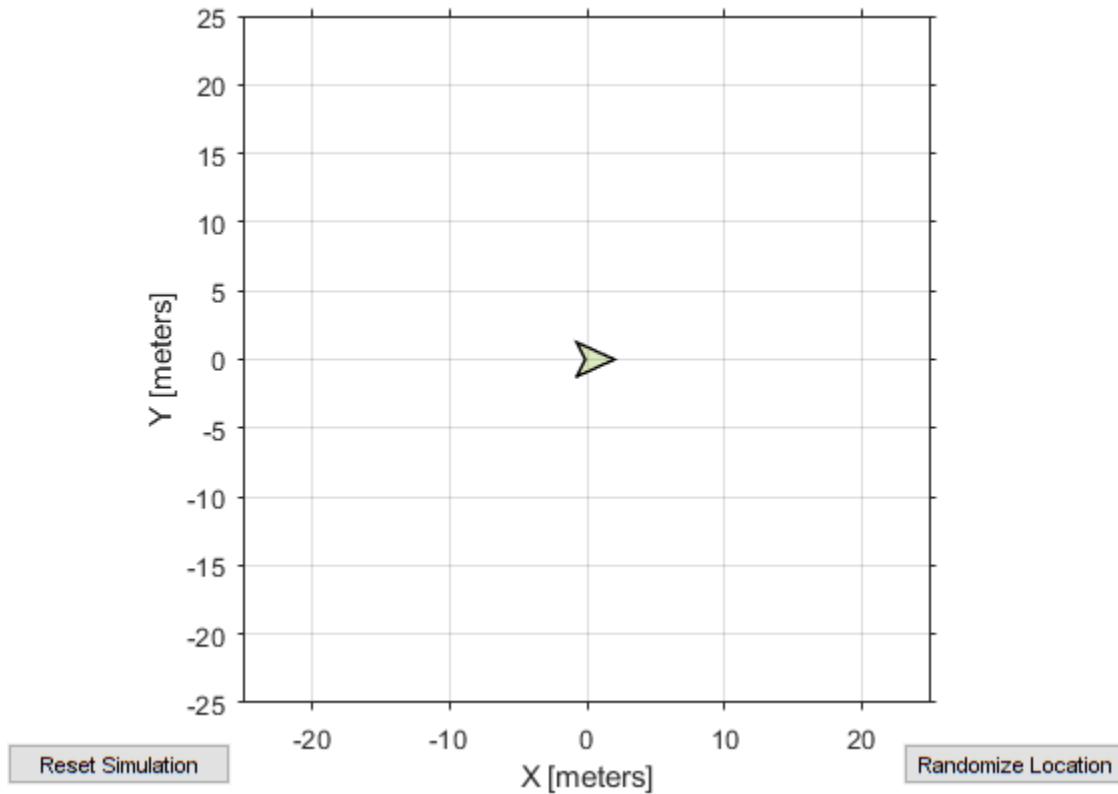
```
ans =
```

```
1×2 cell array
```

```
 {'robotcontroller'}    {'robotcontroller2'}
```

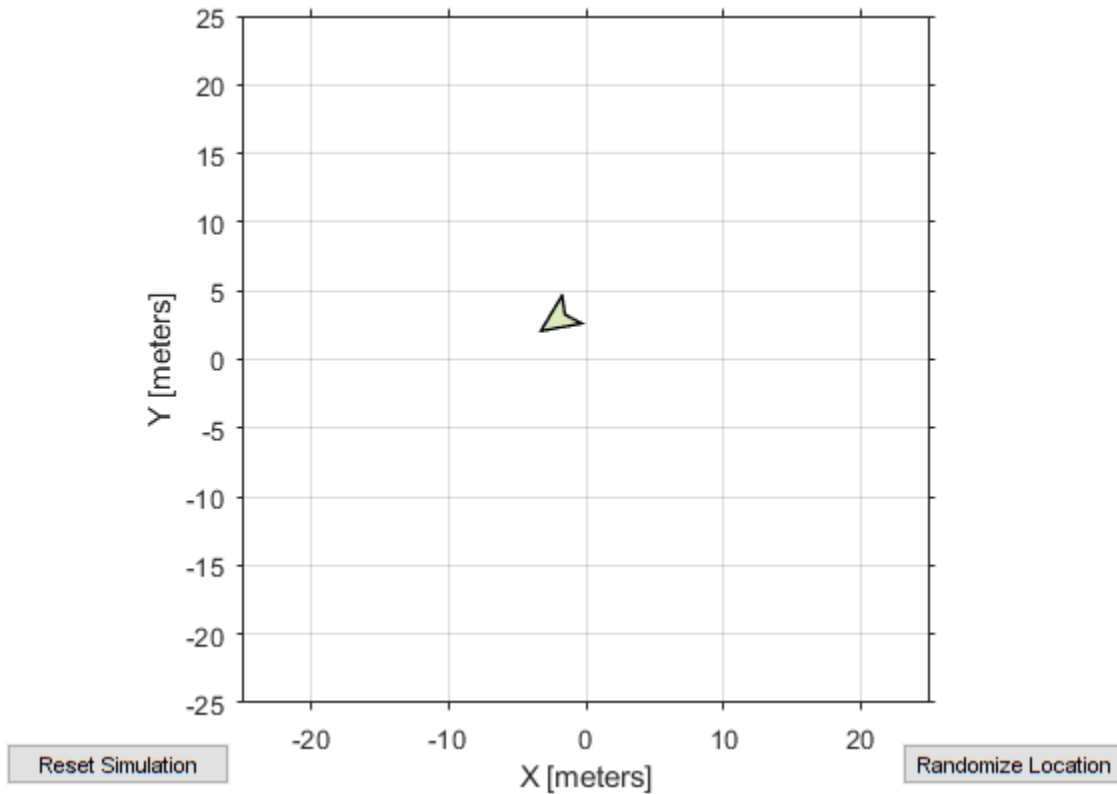
Start up the Robot Simulator using `ExampleHelperSimulinkRobotROS`. This simulator automatically connects to the ROS master on the ROS device. You will use this simulator to run a ROS node and control the robot.

```
sim = ExampleHelperSimulinkRobotROS;
```



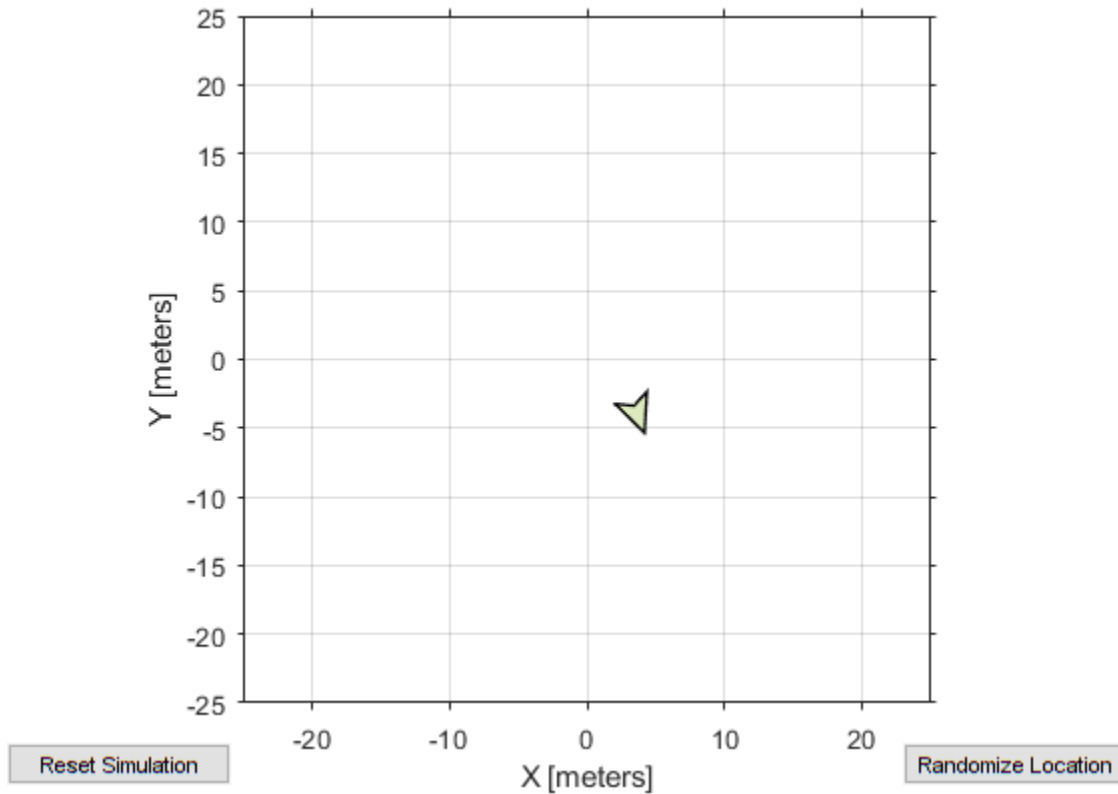
Run a ROS node, specifying the node name. The 'robotcontroller' node commands the robot to a specific location ([ -10 10]). Wait to see the robot drive.

```
runNode(d, 'robotcontroller')  
pause(10)
```



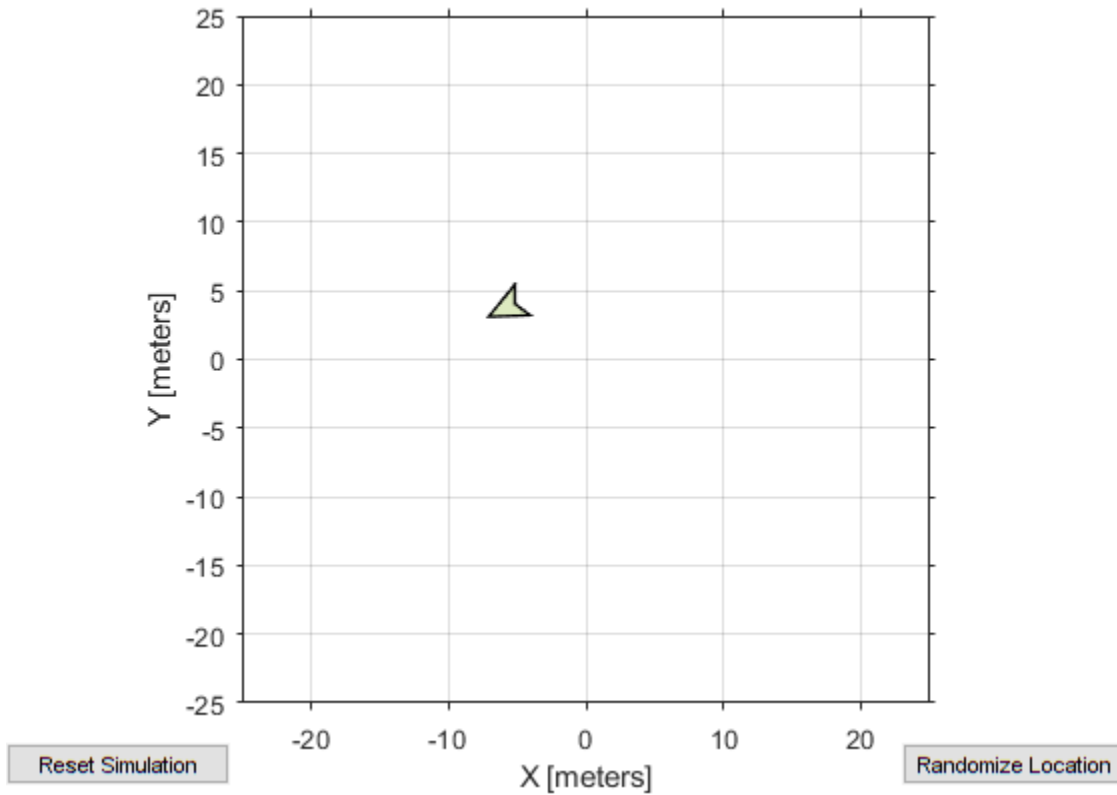
Reset the Robot Simulator to reset the robot position. Alternatively, click **Reset Simulation**. Because the node is still running, the robot continues back to the specific location. To stop sending commands, stop the node.

```
resetSimulation(sim.Simulator)
pause(5)
stopNode(d, 'robotcontroller')
```



Run the 'robotcontroller2' node. This model drives the robot with twice the linear velocity. Reset the robot position. Wait to see the robot drive. You should see a wider turn due to the increased velocity.

```
runNode(d, 'robotcontroller2')
resetSimulation(sim.Simulator)
pause(10)
```



Close the simulator. Stop the ROS node. Disconnect from the ROS network and stop the ROS core.

```
close
stopNode(d, 'robotcontroller2')
roshutdown
stopCore(d)
```

Shutting down global node /matlab\_global\_node\_15972 with NodeURI http://192.168.203.1:

## Input Arguments

### **device** — ROS device

rosdevice object

ROS device, specified as a rosdevice object.

### **modelName** — Name of the deployed Simulink model

character vector

Name of the deployed Simulink model, specified as a character vector. If the model name is not valid, the function returns immediately.

## See Also

`isNodeRunning` | `rosdevice` | `runNode`

## Topics

“Generate a Standalone ROS Node from Simulink®”

**Introduced in R2016b**

# system

Execute system command on device

## Syntax

```
system(device,command)
system(device,command,'sudo')
response = system(____)
```

## Description

`system(device,command)` runs a command in the Linux command shell on the ROS device. This function does not allow you to run interactive commands.

`system(device,command,'sudo')` runs a command with superuser privileges.

`response = system(____)` runs a command using any of the previous syntaxes with the command shell output returned in `response`.

## Examples

### Run Linux Commands on ROS Device

Connect to a ROS device and run commands on the Linux® command shell.

Connect to a ROS device. Specify the device address, user name, and password of your ROS device.

```
d = rosdevice('192.168.154.131','user','password');
```

Run a command that lists the contents of the Catkin workspace folder.

```
system(d,'ls /home/user/catkin_ws_test')
```

```
ans =  
  
build  
devel  
robotcontroller2_node.log  
robotcontroller_node.log  
src
```

## Input Arguments

### **device — ROS device**

rosdevice object

ROS device, specified as a rosdevice object.

### **command — Linux command**

character vector

Linux command, specified as a character vector.

Example: 'ls -al'

## Output Arguments

### **response — Output from Linux shell**

character vector

Output from Linux shell, returned as a character vector.

## See Also

deleteFile | dir | getFile | openShell | putFile | rosdevice

**Introduced in R2016b**



# tform2axang

Convert homogeneous transformation to axis-angle rotation

## Syntax

```
axang = tform2axang(tform)
```

## Description

`axang = tform2axang(tform)` converts the rotational component of a homogeneous transformation, `tform`, to an axis-angle rotation, `axang`. The translational components of `tform` are ignored. The input homogeneous transformation must be in the premultiply form for transformations.

## Examples

### Convert Homogeneous Transformation to Axis-Angle Rotation

```
tform = [1 0 0 0; 0 0 -1 0; 0 1 0 0; 0 0 0 1];  
axang = tform2axang(tform)
```

```
axang = 1×4
```

```
    1.0000         0         0    1.5708
```

## Input Arguments

**tform** — Homogeneous transformation

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

Homogeneous transformation, specified by a 4-by-4-by- $n$  matrix of  $n$  homogeneous transformations. The input homogeneous transformation must be in the premultiply form for transformations.

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

## Output Arguments

### **axang** — Rotation given in axis-angle form

$n$ -by-4 matrix

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

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

## Extended Capabilities

### **C/C++ Code Generation**

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

### **See Also**

`axang2tform`

### **Topics**

“Coordinate Transformations in Robotics”

**Introduced in R2015a**

## tform2eul

Extract Euler angles from homogeneous transformation

### Syntax

```
eul = tform2eul(tform)
eul = tform2eul(tform, sequence)
```

### Description

`eul = tform2eul(tform)` extracts the rotational component from a homogeneous transformation, `tform`, and returns it as Euler angles, `eul`. The translational components of `tform` are ignored. The input homogeneous transformation must be in the premultiply form for transformations. The default order for Euler angle rotations is "ZYX".

`eul = tform2eul(tform, sequence)` extracts the Euler angles, `eul`, from a homogeneous transformation, `tform`, using the specified rotation sequence, `sequence`. The default order for Euler angle rotations is "ZYX".

### Examples

#### Extract Euler Angles from Homogeneous Transformation Matrix

```
tform = [1 0 0 0.5; 0 -1 0 5; 0 0 -1 -1.2; 0 0 0 1];
eulZYX = tform2eul(tform)
```

```
eulZYX = 1×3
          0          0    3.1416
```

### Extract Euler Angles from Homogeneous Transformation Matrix Using ZYZ Rotation

```
tform = [1 0 0 0.5; 0 -1 0 5; 0 0 -1 -1.2; 0 0 0 1];  
eulZYZ = tform2eul(tform, 'ZYZ')
```

```
eulZYZ = 1×3  
         0   -3.1416   3.1416
```

## Input Arguments

### **tform** — Homogeneous transformation

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

Homogeneous transformation, specified by a 4-by-4-by-*n* matrix of *n* homogeneous transformations. The input homogeneous transformation must be in the premultiply form for transformations.

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

### **sequence** — Axis rotation sequence

"ZYX" (default) | "YZZ" | "XYZ"

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

- "ZYX" (default) - The order of rotation angles is z-axis, y-axis, x-axis.
- "YZZ" - The order of rotation angles is z-axis, y-axis, z-axis.
- "XYZ" - The order of rotation angles is x-axis, y-axis, z-axis.

Data Types: string | char

## Output Arguments

### **eul** — Euler rotation angles

*n*-by-3 matrix

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

Example: `[0 0 1.5708]`

## Extended Capabilities

### C/C++ Code Generation

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

### See Also

`eul2tform`

### Topics

“Coordinate Transformations in Robotics”

**Introduced in R2015a**

# tform2quat

Extract quaternion from homogeneous transformation

## Syntax

```
quat = tform2quat(tform)
```

## Description

`quat = tform2quat(tform)` extracts the rotational component from a homogeneous transformation, `tform`, and returns it as a quaternion, `quat`. The translational components of `tform` are ignored. The input homogeneous transformation must be in the premultiply form for transformations.

## Examples

### Extract Quaternion from Homogeneous Transformation

```
tform = [1 0 0 0; 0 -1 0 0; 0 0 -1 0; 0 0 0 1];  
quat = tform2quat(tform)
```

```
quat = 1×4
```

```
    0    1    0    0
```

## Input Arguments

**tform** — Homogeneous transformation

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

Homogeneous transformation, specified by a 4-by-4-by- $n$  matrix of  $n$  homogeneous transformations. The input homogeneous transformation must be in the premultiply form for transformations.

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

## Output Arguments

### **quat** — Unit quaternion

$n$ -by-4 matrix

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

Example: `[0.7071 0.7071 0 0]`

## Extended Capabilities

### C/C++ Code Generation

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

### See Also

`quat2tform`

### Topics

“Coordinate Transformations in Robotics”

**Introduced in R2015a**

## tform2rotm

Extract rotation matrix from homogeneous transformation

### Syntax

```
rotm = tform2rotm(tform)
```

### Description

`rotm = tform2rotm(tform)` extracts the rotational component from a homogeneous transformation, `tform`, and returns it as an orthonormal rotation matrix, `rotm`. The translational components of `tform` are ignored. The input homogeneous transformation must be in the pre-multiply form for transformations. When using the rotation matrix, premultiply it with the coordinates to be rotated (as opposed to postmultiplying).

### Examples

#### Convert Homogeneous Transformation to Rotation Matrix

```
tform = [1 0 0 0; 0 -1 0 0; 0 0 -1 0; 0 0 0 1];  
rotm = tform2rotm(tform)
```

```
rotm = 3×3
```

```
    1     0     0  
    0    -1     0  
    0     0    -1
```



## Input Arguments

### **tform** — Homogeneous transformation

4-by-4-by- $n$  matrix

Homogeneous transformation matrix, specified by a 4-by-4-by- $n$  matrix of  $n$  homogeneous transformations. The input homogeneous transformation must be in the pre-multiply form for transformations.

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

## Output Arguments

### **rotm** — Rotation matrix

3-by-3-by- $n$  matrix

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

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

## Extended Capabilities

### **C/C++ Code Generation**

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

### **See Also**

`rotm2tform`

### **Topics**

“Coordinate Transformations in Robotics”

**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

**tvec** — 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

`tvec2tform`

### Topics

“Coordinate Transformations in Robotics”

**Introduced in R2015a**

## times, .\*

Element-wise quaternion multiplication

## Syntax

```
quatC = A.*B
```

## Description

`quatC = A.*B` returns the element-by-element quaternion multiplication of quaternion arrays.

You can use quaternion multiplication to compose rotation operators:

- To compose a sequence of frame rotations, multiply the quaternions in the same order as the desired sequence of rotations. For example, to apply a  $p$  quaternion followed by a  $q$  quaternion, multiply in the order  $pq$ . The rotation operator becomes  $(pq)^*v(pq)$ , where  $v$  represents the object to rotate in quaternion form.  $*$  represents conjugation.
- To compose a sequence of point rotations, multiply the quaternions in the reverse order of the desired sequence of rotations. For example, to apply a  $p$  quaternion followed by a  $q$  quaternion, multiply in the reverse order,  $qp$ . The rotation operator becomes  $(qp)v(qp)^*$ .

## Examples

### Multiply Two Quaternion Vectors

Create two vectors, A and B, and multiply them element by element.

```
A = quaternion([1:4;5:8]);  
B = A;  
C = A.*B
```

```
C = 2x1 quaternion array
    -28 + 4i + 6j + 8k
   -124 + 60i + 70j + 80k
```

### Multiply Two Quaternion Arrays

Create two 3-by-3 arrays, A and B, and multiply them element by element.

```
A = reshape(quaternion(randn(9,4)),3,3);
B = reshape(quaternion(randn(9,4)),3,3);
C = A.*B
```

```
C = 3x3 quaternion array
    0.60169 + 2.4332i - 2.5844j + 0.51646k   -0.49513 + 1.1722i + 4.4401j - 1.7222k
   -4.2329 + 2.4547i + 3.7768j + 0.77484k   -0.65232 - 0.43112i - 1.4645j - 0.9000k
   -4.4159 + 2.1926i + 1.9037j - 4.0303k    -2.0232 + 0.4205i - 0.17288j + 3.8000k
```

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 +
```

```
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 + 0i + 0j + 0k   -1.3077 + 2.7694i - 0.063055j
    0 + 0i + 0j + 0k   -0.43359 - 1.3499i + 0.71474j
    0 + 0i + 0j + 0k    0.34262 + 3.0349i - 0.20497j

```

## 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  $\beta$  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:

	<b>1</b>	<b>i</b>	<b>j</b>	<b>k</b>
<b>1</b>	1	i	j	k
<b>i</b>	i	-1	k	-j
<b>j</b>	j	-k	-1	i
<b>k</b>	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™.

**Introduced in R2018a**

# timeseries

Creates a time series object for selected message properties

## Syntax

```
[ts,cols] = timeseries(bag)
[ts,cols] = timeseries(bag,property)
[ts,cols] = timeseries(bag,property,...,propertyN)
```

## Description

`[ts,cols] = timeseries(bag)` creates a time series for all numeric and scalar message properties. The function evaluates each message in the current `BagSelection` object, `bag`, as `ts`. The `cols` output argument stores property names as a cell array of character vectors.

The returned time series object is memory-efficient because it stores only particular message properties instead of whole messages.

`[ts,cols] = timeseries(bag,property)` creates a time series for a specific message property, `property`. Property names can also be nested, for example, `Pose.Pose.Position.X` for the x-axis position of a robot.

`[ts,cols] = timeseries(bag,property,...,propertyN)` creates a time series for a range specific message properties. Each property is a different column in the time series object.

## Examples

### Create Time Series from Entire Bag Selection

Load rosbag. Specify the file path.

```
bag = rosbag('ex_multiple_topics.bag');
```

Select a specific topic. Time series only support single topics.

```
bagSelection = select(bag, 'Topic', '/odom');
```

Create time series for the '/odom' topic.

```
ts = timeseries(bagSelection);
```

### **Create Time Series from Single Property**

Load rosbag. Specify the file path.

```
bag = rosbag('ex_multiple_topics.bag');
```

Select a specific topic. Time series only support single topics.

```
bagSelection = select(bag, 'Topic', '/odom');
```

Create time series for the 'Pose.Pose.Position.X' property on the '/odom' topic.

```
ts = timeseries(bagSelection, 'Pose.Pose.Position.X');
```

### **Create Time Series from Multiple Properties**

Load rosbag. Specify the file path.

```
bag = rosbag('ex_multiple_topics.bag');
```

Select a specific topic. Time series only support single topics.

```
bagSelection = select(bag, 'Topic', '/odom');
```

Create time series for all the angular 'Twist' properties on the '/odom' topic.

```
ts = timeseries(bagSelection, 'Twist.Twist.Angular.X', ...  
                'Twist.Twist.Angular.Y', 'Twist.Twist.Angular.Z');
```

## Input Arguments

### **bag** — Bag selection

BagSelection object handle

Bag selection, specified as a BagSelection object handle. You can get a bag selection by calling `rosbag`.

### **property** — Property names

string scalar | character vector

Property names, specified as a string scalar or character vector. Multiple properties can be specified. Each property name is a separate input and represents a different column in the time series object.

## Output Arguments

### **ts** — Time series

Time object handle

Time series, returned as a Time object handle.

### **cols** — List of property names

cell array of character vectors

List of property names, returned as a cell array of character vectors.

## See Also

`readMessages` | `rosbag` | `select`

## Topics

“Time Series” (MATLAB)

**Introduced in R2015a**

# transform

Transform message entities into target coordinate frame

## Syntax

```
tfentity = transform(tftree,targetframe,entity)
tfentity = transform(tftree,targetframe,entity,"msgtime")
tfentity = transform(tftree,targetframe,entity,sourcetime)
```

## Description

`tfentity = transform(tftree,targetframe,entity)` retrieves the latest transformation between `targetframe` and the coordinate frame of `entity` and applies it to `entity`, a ROS message of a specific type. `tftree` is the full transformation tree containing known transformations between entities. If the transformation from `entity` to `targetframe` does not exist, MATLAB throws an error.

`tfentity = transform(tftree,targetframe,entity,"msgtime")` uses the timestamp in the header of the message, `entity`, as the source time to retrieve and apply the transformation.

`tfentity = transform(tftree,targetframe,entity,sourcetime)` uses the given source time to retrieve and apply the transformation to the message, `entity`.

## Examples

### Get ROS Transformations and Apply to ROS Messages

This example shows how to set up a ROS transformation tree and transform frames based on this information. It uses time-buffered transformations to access transformations at different times.

Create a ROS transformation tree. Use `roscpp` to connect to a ROS network. Replace `ipaddress` with your ROS network address.

```
ipaddress = '192.168.203.129';  
rosinit(ipaddress)  
tftree = rostf;  
pause(1)
```

Initializing global node /matlab\_global\_node\_60416 with NodeURI http://192.168.203.1:5

Look at the available frames on the transformation tree.

```
tftree.AvailableFrames
```

```
ans =
```

```
36x1 cell array
```

```
{'base_footprint'      }  
{'base_link'          }  
{'camera_depth_frame' }  
{'camera_depth_optical_frame'}  
{'camera_link'       }  
{'camera_rgb_frame'  }  
{'camera_rgb_optical_frame'}  
{'caster_back_link'  }  
{'caster_front_link' }  
{'cliff_sensor_front_link' }  
{'cliff_sensor_left_link' }  
{'cliff_sensor_right_link' }  
{'gyro_link'         }  
{'mount_asus_xtion_pro_link' }  
{'odom'              }  
{'plate_bottom_link' }  
{'plate_middle_link' }  
{'plate_top_link'    }  
{'pole_bottom_0_link' }  
{'pole_bottom_1_link' }  
{'pole_bottom_2_link' }  
{'pole_bottom_3_link' }  
{'pole_bottom_4_link' }  
{'pole_bottom_5_link' }  
{'pole_kinect_0_link' }  
{'pole_kinect_1_link' }  
{'pole_middle_0_link' }  
{'pole_middle_1_link' }  
{'pole_middle_2_link' }
```

```

{'pole_middle_3_link'      }
{'pole_top_0_link'        }
{'pole_top_1_link'        }
{'pole_top_2_link'        }
{'pole_top_3_link'        }
{'wheel_left_link'        }
{'wheel_right_link'       }

```

Check if the desired transformation is available now. For this example, check for the transformation from 'camera\_link' to 'base\_link'.

```
canTransform(tftree, 'base_link', 'camera_link')
```

```
ans =
```

```
logical
```

```
1
```

Get the transformation for 3 seconds from now. `getTransform` will wait until the transformation becomes available with the specified timeout.

```
desiredTime = rostime('now') + 3;
tform = getTransform(tftree, 'base_link', 'camera_link', ...
                    desiredTime, 'Timeout', 5);
```

Create a ROS message to transform. Messages could also be retrieved off the ROS network.

```
pt = rosmesssage('geometry_msgs/PointStamped');
pt.Header.FrameId = 'camera_link';
pt.Point.X = 3;
pt.Point.Y = 1.5;
pt.Point.Z = 0.2;
```

Transform the ROS message to the 'base\_link' frame using the desired time saved from before.

```
tftpt = transform(tftree, 'base_link', pt, desiredTime);
```

*Optional:* You can also use `apply` with the stored `tform` to apply this transformation to the `pt` message.

```
tfpt2 = apply(tform,pt);
```

Shut down the ROS network.

```
roshutdown
```

```
Shutting down global node /matlab_global_node_60416 with NodeURI http://192.168.203.1:5
```

### Get Buffered Transformations from ROS Network

This example shows how to access time-buffered transformations on the ROS network. Access transformations for specific times and modify the `BufferTime` property based on your desired times.

Create a ROS transformation tree. Use `roslaunch` to connect to a ROS network. Replace `ipaddress` with your ROS network address.

```
ipaddress = '192.168.203.129';  
roslaunch(ipaddress)  
tftree = rostf;  
pause(2);
```

```
Initializing global node /matlab_global_node_29163 with NodeURI http://192.168.203.1:5
```

Get the transformation from 1 second ago.

```
desiredTime = rostime('now') - 1;  
tform = getTransform(tftree,'base_link','camera_link',desiredTime);
```

The transformation buffer time is 10 seconds by default. Modify the `BufferTime` property of the transformation tree to increase the buffer time and wait for that buffer to fill.

```
tftree.BufferTime = 15;  
pause(15);
```

Get the transformation from 12 seconds ago.

```
desiredTime = rostime('now') - 12;  
tform = getTransform(tftree,'base_link','camera_link',desiredTime);
```

You can also get transformations at a time in the future. `getTransform` will wait until the transformation is available. You can also specify a timeout to error out if no



transformation is found. This example waits 5 seconds for the transformation at 3 seconds from now to be available.

```
desiredTime = rostime('now') + 3;  
tform = getTransform(tftree, 'base_link', 'camera_link', desiredTime, 'Timeout', 5);
```

Shut down the ROS network.

```
roshutdown
```

```
Shutting down global node /matlab_global_node_29163 with NodeURI http://192.168.203.1:5
```

## Input Arguments

### **tftree** — ROS transformation tree

TransformationTree object handle

ROS transformation tree, specified as a TransformationTree object handle. You can create a transformation tree by calling the `rostopic` function.

### **targetframe** — Target coordinate frame

string scalar | character vector

Target coordinate frame that entity transforms into, specified as a string scalar or character vector. You can view the available frames for transformation calling `tftree.AvailableFrames`.

### **entity** — Initial message entity

Message object handle

Initial message entity, specified as a Message object handle.

Supported messages are:

- `geometry_msgs/PointStamped`
- `geometry_msgs/PoseStamped`
- `geometry_msgs/QuaternionStamped`
- `geometry_msgs/Vector3Stamped`
- `sensor_msgs/PointCloud2`

### **sourcetime — ROS or system time**

scalar | Time object handle

ROS or system time, specified as a scalar or Time object handle. The scalar is converted to a Time object using `rostime`.

## **Output Arguments**

### **tfentity — Transformed entity**

Message object handle

Transformed entity, returned as a Message object handle.

## **See Also**

`canTransform` | `getTransform`

**Introduced in R2015a**

# transformScan

Transform laser scan based on relative pose

## Syntax

```
transScan = transformScan(scan, relPose)
```

```
[transRanges, transAngles] = transformScan(ranges, angles, relPose)
```

## Description

`transScan = transformScan(scan, relPose)` transforms the laser scan specified in `scan` by using the specified relative pose, `relPose`.

`[transRanges, transAngles] = transformScan(ranges, angles, relPose)` transforms the laser scan specified in `ranges` and `angles` by using the specified relative pose, `relPose`.

## Examples

### Transform Laser Scans

Create a `lidarScan` object. Specify the ranges and angles as vectors.

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

Translate the laser scan by an `[x y]` offset of `(0.5, 0.2)`.

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

Rotate the laser scan by 20 degrees.

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

## Use Scan Matching to Transform Scans

Use the `matchScans` function to find the relative transformation between two laser scans. Then transform the second laser scan into the coordinate frame of the first laser scan.

This example requires an Optimization Toolbox™ license.

Specify a laser scan as ranges and angles. Create a second laser scan that is offset from the first using `transformScan`. This transformation simulates a second laser scan being collected from a new coordinate frame.

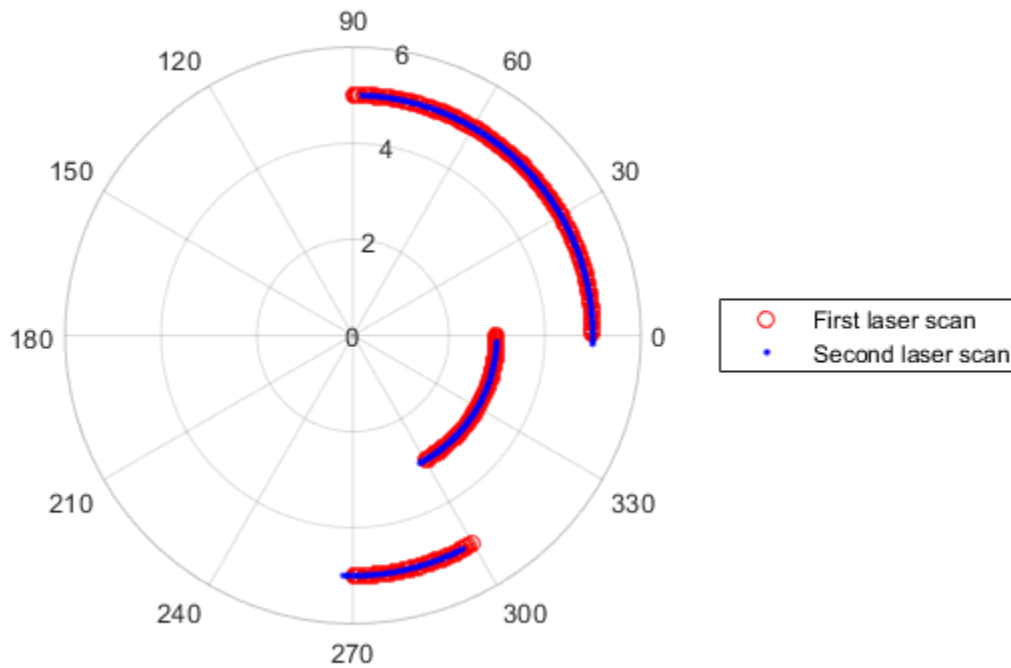
```
refRanges = 5*ones(1,300);  
refRanges(51:150) = 3*ones(1,100);  
refAngles = linspace(-pi/2,pi/2,300);  
offset = [0.5 0.2 0];  
[currRanges,currAngles] = transformScan(refRanges,refAngles,offset);
```

Use scan matching to find the relative pose between the two laser scans. This pose is close to the specified `offset`. You must have an Optimization Toolbox™ license to use the `matchScans` function.

```
pose = matchScans(currRanges,currAngles,refRanges,refAngles,'SolverAlgorithm','fminunc')  
  
pose = 1×3  
  
-0.5102 -0.1806 -0.0394
```

Transform the second scan to the coordinate frame of the first scan. Plot the two scans to see how they overlap.

```
[currRanges2,currAngles2] = transformScan(currRanges,currAngles,pose);  
clf  
polarplot(refAngles,refRanges,'or')  
hold on  
polarplot(currAngles2,currRanges2,'.b')  
legend('First laser scan','Second laser scan')  
hold off
```



## Input Arguments

### **scan** — Lidar scan readings

lidarScan object

Lidar scan readings, specified as a lidarScan object.

### **ranges** — Range values from scan data

vector

Range values from scan data, specified as a vector in meters. These range values are distances from a sensor at specified `angles`. The vector must be the same length as the corresponding `angles` vector.

### **angles — Angle values from scan data**

vector

Angle values from scan data, specified as a vector in radians. These angle values are the specific angles of the specified `ranges`. The vector must be the same length as the corresponding `ranges` vector.

### **relPose — Relative pose of current scan**

[x y theta]

Relative pose of current scan, specified as [x y theta], where [x y] is the translation in meters and theta is the rotation in radians.

## Output Arguments

### **transScan — Transformed lidar scan readings**

lidarScan object

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

### **transRanges — Range values of transformed scan**

vector

Range values of transformed scan, returned as a vector in meters. These range values are distances from a sensor at specified `transAngles`. The vector is the same length as the corresponding `transAngles` vector.

### **transAngles — Angle values from scan data**

vector

Angle values of transformed scan, returned as a vector in radians. These angle values are the specific angles of the specified `transRanges`. The vector is the same length as the corresponding `ranges` vector.

## Extended Capabilities

### C/C++ Code Generation

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

### See Also

#### Functions

LaserScan | lidarScan | matchScans | readCartesian | readScanAngles

#### Classes

MonteCarloLocalization | OccupancyGrid

### Topics

“Estimate Robot Pose with Scan Matching”

“Compose a Series of Laser Scans with Pose Changes”

**Introduced in R2017a**

# transformtraj

Generate trajectories between two transformations

## Syntax

```
[tforms,vel,acc] = transformtraj(T0,TF,tInterval,tSamples)
[tforms,vel,acc] = transformtraj(T0,TF,tInterval,
tSamples,Name,Value)
```

## Description

`[tforms,vel,acc] = transformtraj(T0,TF,tInterval,tSamples)` generates a trajectory that interpolates between two 4-by-4 homogeneous transformations, `T0` and `TF`, with points based on the time interval and given time samples.

`[tforms,vel,acc] = transformtraj(T0,TF,tInterval,tSamples,Name,Value)` specifies additional parameters using `Name, Value` pair arguments.

## Examples

### Interpolate Between Homogenous Transformations

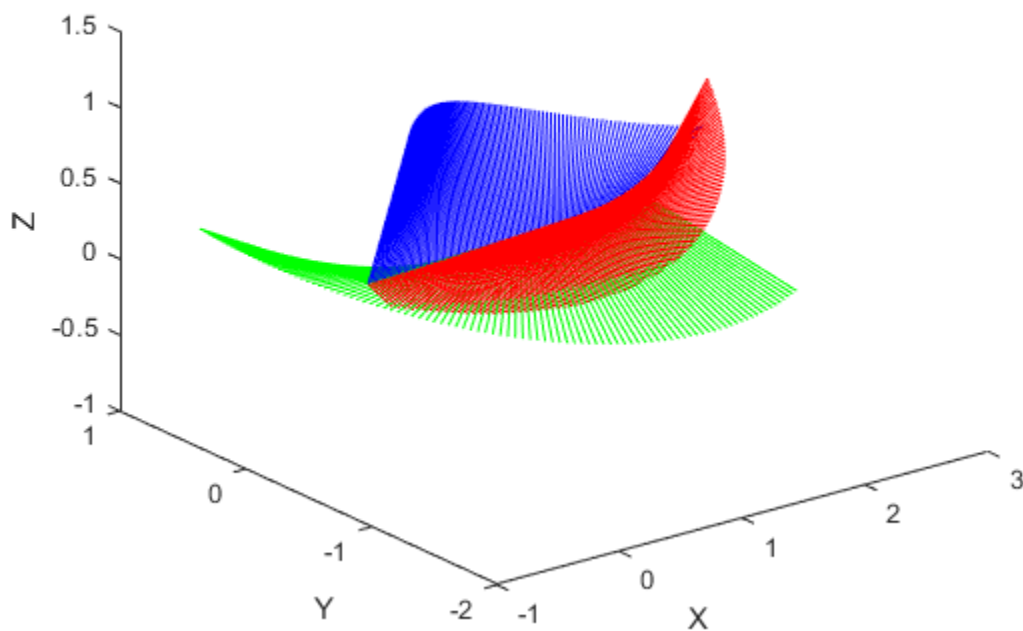
Build transformations from two orientations and positions. Specify the time interval and vector of times for interpolating.

```
t0 = axang2tform([0 1 1 pi/4])*trvec2tform([0 0 0]);
tF = axang2tform([1 0 1 6*pi/5])*trvec2tform([1 1 1]);
tInterval = [0 1];
tvec = 0:0.01:1;
```

Interpolate between the points. Plot the trajectory using `plotTransforms`. Convert the transformations to quaternion rotations and linear transtions. The figure shows all the intermediate transformations of the coordinate frame.



```
[tfInterp, v1, a1] = transformtraj(t0,tF,tInterval,tvec);  
  
rotations = tform2quat(tfInterp);  
translations = tform2trvec(tfInterp);  
  
plotTransforms(translations,rotations)  
xlabel('X')  
ylabel('Y')  
zlabel('Z')
```



## Input Arguments

### **T0 — Initial transformation**

4-by-4 homogeneous transformation

Initial transformation, specified as a 4-by-4 homogeneous transformation. The function generates a trajectory that starts at the initial transformation, `T0`, and goes to the final transformation, `TF`.

Data Types: `single` | `double`

### **TF — Final transformation**

4-by-4 homogeneous transformation

Final transformation, specified as a 4-by-4 homogeneous transformation. The function generates a trajectory that starts at the initial transformation, `T0`, and goes to the final transformation, `TF`.

Data Types: `single` | `double`

### **tInterval — Start and end times for trajectory**

two-element vector

Start and end times for the trajectory, specified as a two-element vector.

Example: `[0 10]`

Data Types: `single` | `double`

### **tSamples — Time samples for trajectory**

$m$ -element vector

Time samples for the trajectory, specified as an  $m$ -element vector. The output trajectory, `rotVector`, is a vector of orientations

Example: `0:0.01:10`

Data Types: `single` | `double`

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes.

You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example: `'TimeScaling', [0 1 2; 0 1 0; 0 0 0]`

### **TimeScaling** — Time scaling vector and first two derivatives

3-by- $m$  vector

Time scaling vector and the first two derivatives, specified as a 3-by- $m$  vector, where  $m$  is the length of `tSamples`. By default, the time scaling is a linear time scaling between the time points in `tInterval`.

For a nonlinear time scaling, specify the values of the time points in the first row. The second and third rows are the velocity and acceleration of the time points, respectively. For example, to follow the path with a linear velocity to the halfway point, and then jump to the end, the time-scaling would be:

```
s(1,:) = [0 0.25 0.5 1 1 1] % Position
s(2,:) = [1 1 1 0 0 0] % Velocity
s(3,:) = [0 0 0 0 0 0] % Acceleration
```

Data Types: `single` | `double`

## **Output Arguments**

### **tforms** — Transformation trajectory

4-by-4-by- $m$  homogeneous transformation matrix array

Transformation trajectory, returned as a 4-by-4-by- $m$  homogeneous transformation matrix array, where  $m$  is the number of points in `tSamples`.

### **vel** — Transformation velocities

6-by- $m$  matrix

Transformation velocities, returned as a 6-by- $m$  matrix, where  $m$  is the number of points in `tSamples`. The first three elements are the angular velocities, and the second three elements are the velocities in time.

### **acc** — Transformation accelerations

6-by- $m$  matrix

Transformation accelerations, returned as a 6-by- $m$  matrix, where  $m$  is the number of points in `tSamples`. The first three elements are the angular accelerations, and the second three elements are the accelerations in time.

## Extended Capabilities

### C/C++ Code Generation

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

### See Also

`bsplinepolytraj` | `cubicpolytraj` | `quinticpolytraj` | `rottraj` | `transformtraj` | `trapveltraj`

**Introduced in R2019a**

## transpose, .'

Transpose a quaternion array

### Syntax

```
Y = quat.'
```

### Description

`Y = quat.'` returns the non-conjugate transpose of the quaternion array, `quat`.

### Examples

#### Vector Transpose

Create a vector of quaternions and compute its nonconjugate transpose.

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

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

```
quatTransposed = quat.'
```

```
quatTransposed = 1x4 quaternion array  
    0.53767 + 0.31877i + 3.5784j + 0.7254k    1.8339 - 1.3077i + 2.7694j
```

## Matrix Transpose

Create a matrix of quaternions and compute its nonconjugate transpose.

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

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

```
quatTransposed = quat.'
```

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

## 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™.

**Introduced in R2018a**

## trapveltraj

Generate trajectories with trapezoidal velocity profiles

### Syntax

```
[q,qd,qdd,tSamples,pp] = trapveltraj(wayPoints,numSamples)
[q,qd,qdd,tSamples,pp] = trapveltraj(wayPoints,numSamples,
Name,Value)
```

### Description

`[q,qd,qdd,tSamples,pp] = trapveltraj(wayPoints,numSamples)` generates a trajectory through a given set of input waypoints that follow a trapezoidal velocity profile. The function outputs positions, velocities, and accelerations at the given time samples, `tSamples`, based on the specified number of samples, `numSamples`. The function also returns the piecewise polynomial `pp` form of the polynomial trajectory with respect to time.

`[q,qd,qdd,tSamples,pp] = trapveltraj(wayPoints,numSamples, Name,Value)` specifies additional parameters using `Name,Value` pair arguments.

### Examples

#### Compute Trapezoidal Velocity Trajectory for 2-D Planar Motion

Use the `trapveltraj` function with a given set of 2-D `xy` waypoints. Time points for the waypoints are also given.

```
wpts = [0 45 15 90 45; 90 45 -45 15 90];
```

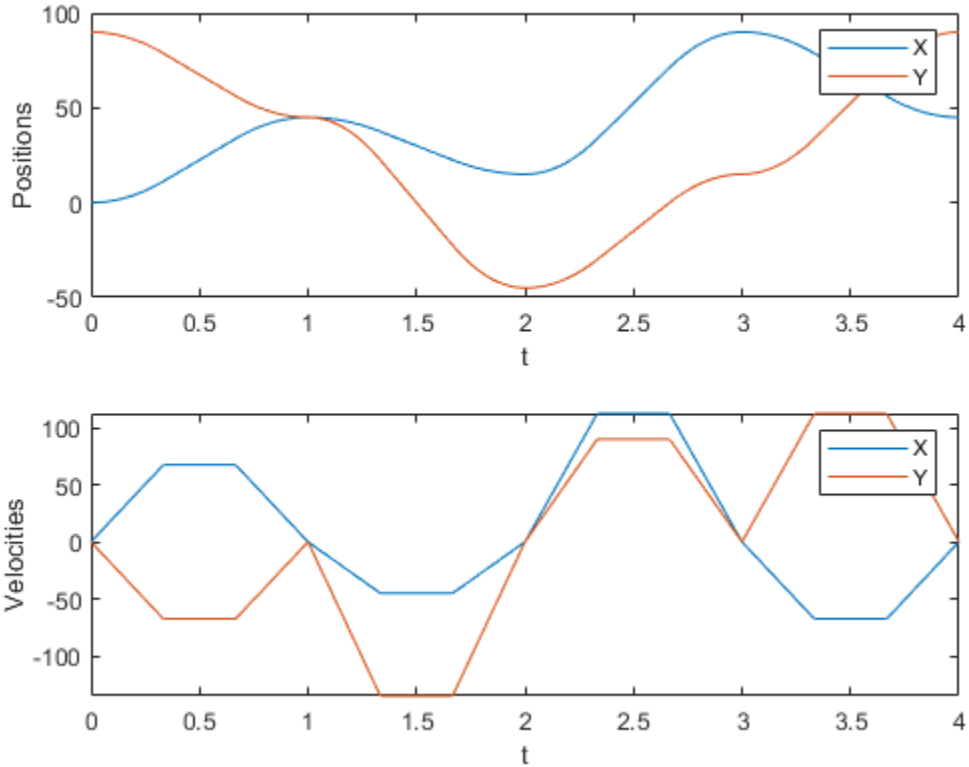
Compute the trajectory for a given number of samples (501). The function outputs the trajectory positions (`q`), velocity (`qd`), acceleration (`qdd`), time vector (`tvec`), and polynomial coefficients (`pp`) of the polynomial that achieves the waypoints using trapezoidal velocities.



```
[q, qd, qdd, tvec, pp] = trapveltraj(wpts, 501);
```

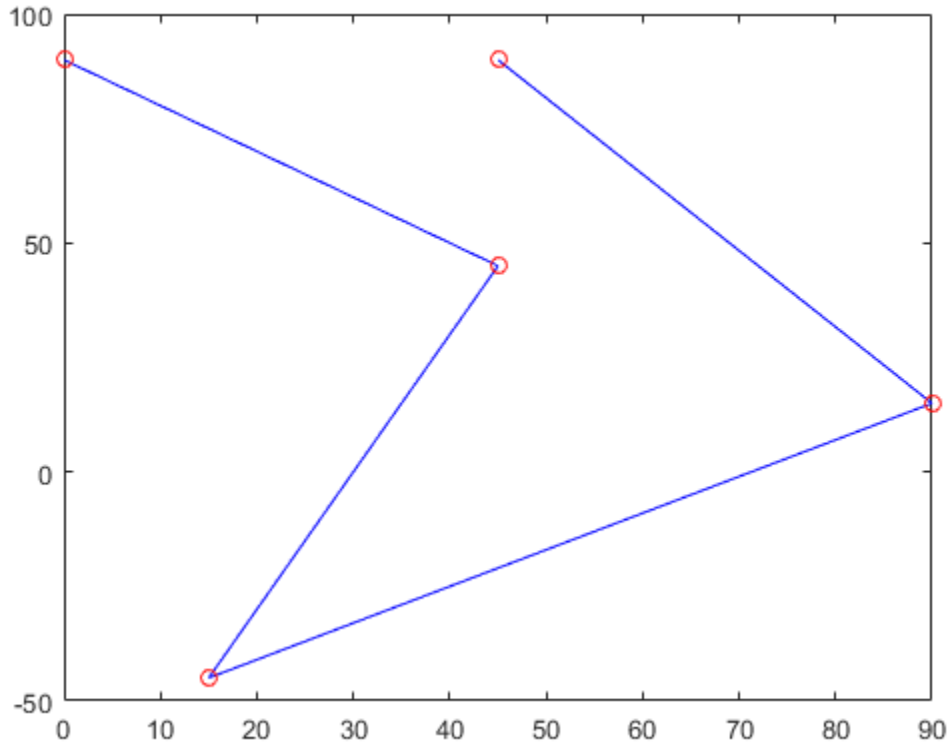
Plot the trajectories for the x- and y-positions and the trapezoidal velocity profile between each waypoint.

```
subplot(2,1,1)
plot(tvec, q)
xlabel('t')
ylabel('Positions')
legend('X', 'Y')
subplot(2,1,2)
plot(tvec, qd)
xlabel('t')
ylabel('Velocities')
legend('X', 'Y')
```



You can also verify the actual positions in the 2-D plane. Plot the separate rows of the  $q$  vector and the waypoints as x- and y-positions.

```
figure
plot(q(1,:),q(2:,:), '-b', wpts(1,:),wpts(2,:), 'or')
```



## Input Arguments

### **wayPoints** — Waypoints for trajectory

*n*-by-*p* matrix

Points for waypoints of trajectory, specified as an *n*-by-*p* matrix, where *n* is the dimension of the trajectory and *p* is the number of waypoints.

Example: [1 4 4 3 -2 0; 0 1 2 4 3 1]

Data Types: single | double

### **numSamples — Number of samples in output trajectory**

positive integer

Number of samples in output trajectory, specified as a positive integer.

Data Types: `single` | `double`

## **Name-Value Pair Arguments**

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

---

**Note** Due to the nature of the trapezoidal velocity profile, you can only set at most two of the following parameters.

---

Example: `'PeakVelocity',5`

### **PeakVelocity — Peak velocity of the velocity profile**

scalar |  $n$ -element vector |  $n$ -by- $(p-1)$  matrix

Peak velocity of the profile segment, specified as the comma-separated pair consisting of `'PeakVelocity'` and a scalar, vector, or matrix. This peak velocity is the highest velocity achieved during the trapezoidal velocity profile.

A scalar value is applied to all elements of the trajectory and between all waypoints. An  $n$ -element vector is applied to each element of the trajectory between all waypoints. An  $n$ -by- $(p-1)$  matrix is applied to each element of the trajectory for each waypoint.

Data Types: `single` | `double`

### **Acceleration — Acceleration of velocity profile**

scalar |  $n$ -element vector |  $n$ -by- $(p-1)$  matrix

Acceleration of the velocity profile, specified as the comma-separated pair consisting of `'Acceleration'` and a scalar, vector, or matrix. This acceleration defines the constant acceleration from zero velocity to the `PeakVelocity` value.

A scalar value is applied to all elements of the trajectory and between all waypoints. An  $n$ -element vector is applied to each element of the trajectory between all waypoints. An  $n$ -by- $(p-1)$  matrix is applied to each element of the trajectory for each waypoint.

Data Types: `single` | `double`

### **EndTime — Duration of each trajectory segment**

scalar |  $n$ -element vector |  $n$ -by- $(p-1)$  matrix

Duration of each of the  $p-1$  trajectory segments, specified as the comma-separated pair consisting of 'EndTime' and a scalar, vector, or matrix.

A scalar value is applied to all elements of the trajectory and between all waypoints. An  $n$ -element vector is applied to each element of the trajectory between all waypoints. An  $n$ -by- $(p-1)$  matrix is applied to each element of the trajectory for each waypoint.

Data Types: `single` | `double`

### **AccelTime — Duration of acceleration phase of velocity profile**

scalar |  $n$ -element vector |  $n$ -by- $(p-1)$  matrix

Duration of acceleration phase of velocity profile, specified as the comma-separated pair consisting of 'AccelTime' and a scalar, vector, or matrix.

A scalar value is applied to all elements of the trajectory and between all waypoints. An  $n$ -element vector is applied to each element of the trajectory between all waypoints. An  $n$ -by- $(p-1)$  matrix is applied to each element of the trajectory for each waypoint.

Data Types: `single` | `double`

## **Output Arguments**

### **q — Positions of trajectory**

$n$ -by- $m$  matrix

Positions of the trajectory at the given time samples in `tSamples`, returned as  $n$ -by- $m$  matrix, where  $n$  is the dimension of the trajectory, and  $m$  is equal to `numSamples`.

Data Types: `single` | `double`

### **qd — Velocities of trajectory**

$n$ -by- $m$  matrix

Velocities of the trajectory at the given time samples in `tSamples`, returned as  $n$ -by- $m$  matrix, where  $n$  is the dimension of the trajectory, and  $m$  is equal to `numSamples`.

Data Types: `single` | `double`

### **qdd — Accelerations of trajectory**

$n$ -by- $m$  matrix

Accelerations of the trajectory at the given time samples in `tSamples`, returned as  $n$ -by- $m$  matrix, where  $n$  is the dimension of the trajectory, and  $m$  is equal to `numSamples`.

Data Types: `single` | `double`

### **tSamples — Time samples for trajectory**

$m$ -element vector

Time samples for the trajectory, returned as an  $m$ -element vector. The output position, `q`, velocity, `qd`, and accelerations, `qdd` are sampled at these time intervals.

Example: `0:0.01:10`

Data Types: `single` | `double`

### **pp — Piecewise polynomials**

cell array or structures

Piecewise polynomials, returned as a cell array of structures that defines the polynomial for each section of the piecewise trajectory. If all the elements of the trajectory share the same breaks, the cell array is a single piecewise polynomial structure. Otherwise, the cell array has  $n$  elements, which correspond to each of the different trajectory elements (dimensions). Each structure contains the fields:

- `form`: 'pp'.
- `breaks`:  $p$ -element vector of times when the piecewise trajectory changes forms.  $p$  is the number of waypoints.
- `coefs`:  $n(p-1)$ -by-order matrix for the coefficients for the polynomials.  $n(p-1)$  is the dimension of the trajectory times the number of pieces. Each set of  $n$  rows defines the coefficients for the polynomial that described each variable trajectory.
- `pieces`:  $p-1$ . The number of breaks minus 1.
- `order`: Degree of the polynomial + 1. For example, cubic polynomials have an order of 4.

- `dim`:  $n$ . The dimension of the control point positions.

You can build your own piecewise polynomials using `mkpp`, or evaluate the polynomial at specified times using `ppval`.

### **pp — Piecewise-polynomial**

structure

Piecewise-polynomial, returned as a structure that defines the polynomial for each section of the piecewise trajectory. You can build your own piecewise polynomials using `mkpp`, or evaluate the polynomial at specified times using `ppval`. The structure contains the fields:

- `form`: 'pp'.
- `breaks`:  $p$ -element vector of times when the piecewise trajectory changes forms.  $p$  is the number of waypoints.
- `coefs`:  $n(p-1)$ -by-order matrix for the coefficients for the polynomials.  $n(p-1)$  is the dimension of the trajectory times the number of pieces. Each set of  $n$  rows defines the coefficients for the polynomial that described each variable trajectory.
- `pieces`:  $p-1$ . The number of breaks minus 1.
- `order`: Degree of the polynomial + 1. For example, cubic polynomials have an order of 4.
- `dim`:  $n$ . The dimension of the control point positions.

## **References**

- [1] Lynch, Kevin M., and Frank C. Park. *Modern Robotics: Mechanics, Planning and Control*. Cambridge: Cambridge University Press, 2017.
- [2] Spong, Mark W., Seth Hutchinson, and M. Vidyasagar. *Robot Modeling and Control*. John Wiley & Sons, 2006.

## **Extended Capabilities**

### **C/C++ Code Generation**

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

## **See Also**

`bsplinepolytraj` | `cubicpolytraj` | `quinticpolytraj` | `rottraj` |  
`transformtraj` | `trapveltraj`

**Introduced in R2019a**



# trvec2tform

Convert translation vector to homogeneous transformation

## Syntax

```
tform = trvec2tform(trvec)
```

## Description

`tform = trvec2tform(trvec)` converts the Cartesian representation of a translation vector, `trvec`, to the corresponding homogeneous transformation, `tform`. When using the transformation matrix, premultiply it with the coordinates to be transformed (as opposed to postmultiplying).

## Examples

### Convert Translation Vector to Homogeneous Transformation

```
trvec = [0.5 6 100];  
tform = trvec2tform(trvec)
```

```
tform = 4×4
```

```
    1.0000         0         0     0.5000  
         0     1.0000         0     6.0000  
         0         0     1.0000    100.0000  
         0         0         0         1.0000
```

## Input Arguments

**trvec** — Cartesian representation of a translation vector  
*n*-by-3 matrix

Cartesian representation of a translation vector, specified as an  $n$ -by-3 matrix containing  $n$  translation vectors. Each vector is of the form  $t = [x \ y \ z]$ .

Example: `[0.5 6 100]`

## Output Arguments

### **tform** — Homogeneous transformation

4-by-4-by- $n$  matrix

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

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

## Extended Capabilities

### **C/C++ Code Generation**

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

### **See Also**

`tform2tvec`

### **Topics**

“Coordinate Transformations in Robotics”

**Introduced in R2015a**

# waitfor

**Package:** robotics

Pause code execution to achieve desired execution rate

## Syntax

```
waitfor(rate)
numMisses = waitfor(rate)
```

## Description

`waitfor(rate)` pauses execution until the code reaches the desired execution rate. The function accounts for the time that is spent executing code between `waitfor` calls.

`numMisses = waitfor(rate)` returns the number of iterations missed while executing code between calls.

## Examples

### Run Loop at Fixed Rate

Create a rate object that runs at 1 Hz.

```
r = robotics.Rate(1);
```

Start a loop using the `Rate` object inside to control the loop execution. Reset the object prior to the loop execution to reset timer. Print the iteration and time elapsed.

```
reset(r)
for i = 1:10
    time = r.TotalElapsedTime;
    fprintf('Iteration: %d - Time Elapsed: %f\n',i,time)
    waitfor(r);
end
```

```
Iteration: 1 - Time Elapsed: 0.004475
Iteration: 2 - Time Elapsed: 1.005114
Iteration: 3 - Time Elapsed: 2.004950
Iteration: 4 - Time Elapsed: 3.004543
Iteration: 5 - Time Elapsed: 4.005585
Iteration: 6 - Time Elapsed: 5.005309
Iteration: 7 - Time Elapsed: 6.005151
Iteration: 8 - Time Elapsed: 7.004336
Iteration: 9 - Time Elapsed: 8.004075
Iteration: 10 - Time Elapsed: 9.005004
```

Each iteration executes at a 1-second interval.

## Input Arguments

### **rate** — Rate object

handle

Rate object, specified as a handle. This object contains the information for the desired rate and other information about the execution. See `robotics.Rate` for more information.

## Output Arguments

### **numMisses** — Number of missed task executions

scalar

Number of missed task executions, returned as a scalar. `waitfor` returns the number of times the task was missed in the Rate object based on the `LastPeriod` time. For example, if the desired rate is 1 Hz and the last period was 3.2 seconds, `numMisses` returns 3.

## See Also

`robotics.Rate` | `rosclock` | `waitfor`

## Topics

“Execute Code at a Fixed-Rate”

**Introduced in R2016a**

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

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

## 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™.

**Introduced in R2018a**

# waitForServer

Wait for action server to start

## Syntax

```
waitForServer(client)
waitForServer(client,timeout)
```

## Description

`waitForServer(client)` waits until the action server is started up and available to send goals. The `IsServerConnected` property of the `SimpleActionClient` shows the status of the server connection. Press **Ctrl+C** to abort the wait.

`waitForServer(client,timeout)` specifies a timeout period in seconds. If the server does not start up in the timeout period, this function displays an error.

## Examples

### Setup a ROS Action Client and Execute an Action

This example shows how to create a ROS action client and execute the action. Action types must be setup beforehand with an action server running.

You must have the `'/fibonacci'` action type setup. To run this action server use the following command on the ROS system:

```
roslaunch actionlib_tutorials fibonacci_server
```

Connect to a ROS network. You must be connected to a ROS network to gather information about what actions are available. Replace `ipaddress` with your network address.



```
ipaddress = '192.168.154.131';
rosinit(ipaddress)
```

Initializing global node /matlab\_global\_node\_68978 with NodeURI http://192.168.154.1:5

List actions available on the network. The only action setup on this network is the '/fibonacci' action.

```
rosaction list
```

```
/fibonacci
```

Create an action client. Specify the action name.

```
[actClient,goalMsg] = rosactionclient('/fibonacci');
```

Wait for action client to connect to server.

```
waitForServer(actClient);
```

The fibonacci action will calculate the fibonacci sequence for a given order specified in the goal message. The goal message was returned when creating the action client and can be modified to send goals to the ROS action server.

```
goalMsg.Order = 8
```

```
goalMsg =
```

```
ROS FibonacciGoal message with properties:
```

```
  MessageType: 'actionlib_tutorials/FibonacciGoal'
  Order: 8
```

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

Send goal and wait for its completion. Specify a timeout of 10 seconds to complete the action.

```
[resultMsg,resultState] = sendGoalAndWait(actClient,goalMsg,10)
```

```
Goal active
```

```
Feedback:
```

```
  Sequence : [0, 1, 1]
```

```
Feedback:
```

```
Sequence : [0, 1, 1, 2]
Feedback:
Sequence : [0, 1, 1, 2, 3]
Feedback:
Sequence : [0, 1, 1, 2, 3, 5]
Feedback:
Sequence : [0, 1, 1, 2, 3, 5, 8]
Feedback:
Sequence : [0, 1, 1, 2, 3, 5, 8, 13]
Feedback:
Sequence : [0, 1, 1, 2, 3, 5, 8, 13, 21]
Feedback:
Sequence : [0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
Final state succeeded with result:
Sequence : [0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
```

```
resultMsg =
```

```
ROS FibonacciResult message with properties:
```

```
MessageType: 'actionlib_tutorials/FibonacciResult'
Sequence: [10x1 int32]
```

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

```
resultState =
```

```
1x9 char array
```

```
succeeded
```

```
Disconnect from the ROS network.
```

```
roshutdown
```

```
Shutting down global node /matlab_global_node_68978 with NodeURI http://192.168.154.1:5
```

## Send and Cancel ROS Action Goals

Send and cancel goals for ROS actions. First, setup a ROS action client. Then send a goal message with modified parameters. Finally, cancel your goal and all goals on the action server.

Connect to a ROS network with a specified IP address. Create a ROS action client connected using `rosactionclient`. Specify the action name. Wait for the client to be connected to the server.

```
rosinit('192.168.154.131')
[actClient,goalMsg] = rosactionclient('/fibonacci');
waitForServer(actClient);
```

```
Initializing global node /matlab_global_node_40739 with NodeURI http://192.168.154.1:5
```

Send a goal message with modified parameters. Wait for the goal to finish executing.

```
goalMsg.Order = 4;
sendGoalAndWait(actClient,goalMsg)
```

```
Goal active
Feedback:
  Sequence : [0, 1, 1]
Feedback:
  Sequence : [0, 1, 1, 2]
Feedback:
  Sequence : [0, 1, 1, 2, 3]
Feedback:
  Sequence : [0, 1, 1, 2, 3, 5]
```

```
ans =
```

```
ROS FibonacciResult message with properties:
```

```
  MessageType: 'actionlib_tutorials/FibonacciResult'
  Sequence: [6×1 int32]
```

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

Send a new goal message without waiting.

```
goalMsg.Order = 5;
sendGoal(actClient,goalMsg)
```

Cancel the goal on the ROS action client, `actClient`.

```
cancelGoal(actClient)
```

Cancel all the goals on the action server that `actClient` is connected to.

```
cancelAllGoals(actClient)
```

Delete the action client.

```
delete(actClient)
```

Disconnect from the ROS network.

```
roshutdown
```

```
Shutting down global node /matlab_global_node_40739 with NodeURI http://192.168.154.1:5
```

## Input Arguments

### **client** — ROS action client

`SimpleActionClient` object handle

ROS action client, specified as a `SimpleActionClient` object handle. This simple action client enables you to track a single goal at a time.

### **timeout** — Timeout period

scalar in seconds

Timeout period for setting up ROS action server, specified as a scalar in seconds. If the client does not connect to the server in the specified time period, an error is displayed.

## See Also

`cancelGoal` | `roaction` | `roactionclient` | `sendGoalAndWait`

## Topics

“ROS Actions Overview”

“Move a Turtlebot Robot Using ROS Actions”

**Introduced in R2016b**

# waitForTransform

Wait until a transformation is available

---

**Note** `waitForTransform` will be removed in a future release. Use `getTransform` with a specified `timeout` instead. Use `inf` to wait indefinitely.

---

## Syntax

```
waitForTransform(tftree, targetframe, sourceframe)
waitForTransform(tftree, targetframe, sourceframe, timeout)
```

## Description

`waitForTransform(tftree, targetframe, sourceframe)` waits until the transformation between `targetframe` and `sourceframe` is available in the transformation tree, `tftree`. This function disables the command prompt until a transformation becomes available on the ROS network.

`waitForTransform(tftree, targetframe, sourceframe, timeout)` specifies a timeout period in seconds. If the transformation does not become available, MATLAB displays an error, but continues running the current program.

## Examples

### Wait for Transformation Between Robot Frames

Connect to the ROS network. Specify the IP address of your network.

```
rosinit('192.168.154.131')
```

```
Initializing global node /matlab_global_node_73613 with NodeURI http://192.168.154.1:5
```

Create a ROS transformation tree.

```
tftree = rostf;
```

Wait for the transformation between the target frame, `/camera_depth_frame`, and the source frame, `/base_link`, to be available. Specify a timeout of 5 seconds.

```
waitForTransform(tftree, '/camera_depth_frame', '/base_link', 5);
```

Get the transformation.

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

When you are finished, disconnect from the ROS network.

```
roshutdown
```

```
Shutting down global node /matlab_global_node_73613 with NodeURI http://192.168.154.1:5
```

## Input Arguments

### **tftree** — ROS transformation tree

TransformationTree object handle

ROS transformation tree, specified as a TransformationTree object handle. You can create a transformation tree by calling the `rostf` function.

### **targetframe** — Target coordinate frame

string scalar | character vector

Target coordinate frame, specified as a string scalar or character vector. You can view the available frames for transformation by calling `tftree.AvailableFrames`.

### **sourceframe** — Initial coordinate frame

string scalar | character vector

Initial coordinate frame, specified as a string scalar or character vector. You can view the available frames for transformation using `tftree.AvailableFrames`.

### **timeout** — Timeout period

numeric scalar in seconds

Timeout period, specified as a numeric scalar in seconds. If the transformation does not become available, MATLAB displays an error, but continues running the current program.

## **See Also**

getTransform | receive | transform

**Introduced in R2015a**



# writeBinaryOccupancyGrid

Write values from grid to ROS message

## Syntax

```
writeBinaryOccupancyGrid(msg, map)
```

## Description

`writeBinaryOccupancyGrid(msg, map)` writes occupancy values and other information to the ROS message, `msg`, from the binary occupancy grid, `map`.

## Examples

### Write Binary Occupancy Grid Information to ROS Message

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

```
map = robotics.BinaryOccupancyGrid(randi([0,1], 10));  
msg = rosmessage('nav_msgs/OccupancyGrid');  
writeBinaryOccupancyGrid(msg, map);
```

## Input Arguments

### **map** — Binary occupancy grid

BinaryOccupancyGrid object handle

Binary occupancy grid, specified as a BinaryOccupancyGrid object handle. `map` is converted to a 'nav\_msgs/OccupancyGrid' message on the ROS network. `map` is an object with a grid of binary values, where 1 indicates an occupied location and 0 indicates an unoccupied location.

**msg — 'nav\_msgs/OccupancyGrid' ROS message**

OccupancyGrid object handle

'nav\_msgs/OccupancyGrid' ROS message, specified as a OccupancyGrid object handle.

**See Also**

readBinaryOccupancyGrid | readOccupancyGrid |  
robotics.BinaryOccupancyGrid | robotics.OccupancyGrid |  
writeOccupancyGrid

**Introduced in R2015a**

# writeOccupancyGrid

Write values from grid to ROS message

## Syntax

```
writeOccupancyGrid(msg, map)
```

## Description

`writeOccupancyGrid(msg, map)` writes occupancy values and other information to the ROS message, `msg`, from the occupancy grid, `map`.

## Examples

### Create OccupancyGrid ROS Message From Grid

The 'nav\_msgs/OccupancyGrid' ROS message contains data for a 2-D occupancy grid with probabilistic values for occupancy. Convert a MATLAB® `OccupancyGrid` object into a ROS message using `writeOccupancyGrid`.

Create an occupancy grid with random data and an empty ROS message to put data into.

```
map = robotics.OccupancyGrid(rand(10));  
msg = rosmesssage('nav_msgs/OccupancyGrid');
```

Write the data from the occupancy grid into the message.

```
writeOccupancyGrid(msg, map);
```

## Input Arguments

**msg** — 'nav\_msgs/OccupancyGrid' ROS message

OccupancyGrid object handle

'nav\_msgs/OccupancyGrid' ROS message, specified as an OccupancyGrid ROS message object handle.

**map** — **Occupancy grid**

robotics.OccupancyGrid object handle

Occupancy grid, returned as an robotics.OccupancyGrid object handle.

## See Also

OccupancyGrid | readBinaryOccupancyGrid | robotics.BinaryOccupancyGrid | robotics.OccupancyGrid | writeBinaryOccupancyGrid

**Introduced in R2016b**

# writeImage

Write MATLAB image to ROS image message

## Syntax

```
writeImage(msg, img)  
writeImage(msg, img, alpha)
```

## Description

`writeImage(msg, img)` converts the MATLAB image, `img`, to a message object and stores the ROS compatible image data in the message object, `msg`. The message must be a 'sensor\_msgs/Image' message. 'sensor\_msgs/CompressedImage' messages are not supported. The function does not perform any color space conversion, so the `img` input needs to have the encoding that you specify in the Encoding property of the message.

`writeImage(msg, img, alpha)` converts the MATLAB image, `img` to a message object. If the image encoding supports an alpha channel (`rgba` or `bgra` family), specify this alpha channel in `alpha`. Alternatively, the input image can store the alpha channel as its fourth channel.

## Examples

### Write Image to Message

Read an image.

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

Create a ROS image message. Specify the default encoding for the image. Write the image to the message.

```
msg = rosmesssage('sensor_msgs/Image');  
msg.Encoding = 'rgb8';  
writeImage(msg, image);
```

## Input Arguments

### **msg** — ROS image message

Image object handle

'sensor\_msgs/Image' ROS image message, specified as an Image object handle. 'sensor\_msgs/Image' image messages are not supported.

### **img** — Image

grayscale image matrix | RGB image matrix | *m*-by-*n*-by-3 array

Image, specified as a matrix representing a grayscale or RGB image or as *m*-by-*n*-by-3 array, depending on the sensor image.

### **alpha** — Alpha channel

uint8 grayscale image

Alpha channel, specified as a uint8 grayscale image. Alpha must be the same size and data type as `img`.

## ROS Image Encoding

You must specify the correct encoding of the input image in the Encoding property of the image message. If you do not specify the image encoding before calling the function, the default encoding, `rgb8`, is used (3-channel RGB image with uint8 values). The function does not perform any color space conversion, so the `img` input needs to have the encoding that you specify in the Encoding property of the message.

All encoding types supported for the `readImage` are also supported in this function. For more information on supported encoding types and their representations in MATLAB, see `readImage`.

Bayer-encoded images (`bayer_rggb8`, `bayer_bggr8`, `bayer_gbrg8`, `bayer_grbg8` and their 16-bit equivalents) must be given as 8-bit or 16-bit single-channel images or they do not encode.

## **See Also**

readImage

**Introduced in R2015a**

## 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™.

**Introduced in R2018a**

# 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`.

---

**Note** This function requires you to install the UAV Library for Robotics System Toolbox. To install add-ons, use `roboticsAddons` and select the desired add-on.

---

## 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 entires.

```
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 u
"MAV_CMD_NAV_LOITER_TURNS"	18	"Loiter around this waypoint for
"MAV_CMD_NAV_LOITER_TIME"	19	"Loiter around this waypoint for
"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 fr
"MAV_CMD_NAV_TAKEOFF_LOCAL"	24	"Takeoff from local position (loc
"MAV_CMD_NAV_FOLLOW"	25	"Vehicle following, i.e. this way
"MAV_CMD_NAV_CONTINUE_AND_CHANGE_ALT"	30	"Continue on the current course a
"MAV_CMD_NAV_LOITER_TO_ALT"	31	"Begin loiter at the specified La

```

"MAV_CMD_DO_FOLLOW"           32      "Being following a target"
"MAV_CMD_DO_FOLLOW_REPOSITION" 33      "Reposition the MAV after a follow"
"MAV_CMD_DO_ORBIT"           34      "Start orbiting on the circumference"
"MAV_CMD_NAV_ROI"            80      "Sets the region of interest (ROI)"
:

```

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 a
```

```
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`.

---

**Note** This function requires you to install the UAV Library for Robotics System Toolbox. To install add-ons, use `roboticsAddons` and select the desired add-on.

---

## 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 entires.

```
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 u
"MAV_CMD_NAV_LOITER_TURNS"	18	"Loiter around this waypoint for
"MAV_CMD_NAV_LOITER_TIME"	19	"Loiter around this waypoint for
"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 fr
"MAV_CMD_NAV_TAKEOFF_LOCAL"	24	"Takeoff from local position (loc
"MAV_CMD_NAV_FOLLOW"	25	"Vehicle following, i.e. this way
"MAV_CMD_NAV_CONTINUE_AND_CHANGE_ALT"	30	"Continue on the current course a
"MAV_CMD_NAV_LOITER_TO_ALT"	31	"Begin loiter at the specified La

```

"MAV_CMD_DO_FOLLOW"           32      "Being following a target"
"MAV_CMD_DO_FOLLOW_REPOSITION" 33      "Reposition the MAV after a follow"
"MAV_CMD_DO_ORBIT"           34      "Start orbiting on the circumference"
"MAV_CMD_NAV_ROI"            80      "Sets the region of interest (ROI)"
:

```

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 a

```
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

“Use a MAVLink Parameter Protocol for Tuning UAV Parameters in MATLAB”

**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.

---

**Note** This function requires you to install the UAV Library for Robotics System Toolbox. To install add-ons, use `roboticsAddons` and select the desired add-on.

---

## 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 entires.

```
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 u
"MAV_CMD_NAV_LOITER_TURNS"	18	"Loiter around this waypoint for
"MAV_CMD_NAV_LOITER_TIME"	19	"Loiter around this waypoint for
"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 fr
"MAV_CMD_NAV_TAKEOFF_LOCAL"	24	"Takeoff from local position (loc
"MAV_CMD_NAV_FOLLOW"	25	"Vehicle following, i.e. this way
"MAV_CMD_NAV_CONTINUE_AND_CHANGE_ALT"	30	"Continue on the current course a
"MAV_CMD_NAV_LOITER_TO_ALT"	31	"Begin loiter at the specified La

---

```

"MAV_CMD_DO_FOLLOW"          32      "Being following a target"
"MAV_CMD_DO_FOLLOW_REPOSITION" 33      "Reposition the MAV after a follow"
"MAV_CMD_DO_ORBIT"          34      "Start orbiting on the circumference"
"MAV_CMD_NAV_ROI"           80      "Sets the region of interest (ROI)"
:

```

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 a
```

```
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`.

---

**Note** This function requires you to install the UAV Library for Robotics System Toolbox. To install add-ons, use `roboticsAddons` and select the desired add-on.

---

## 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 entires.

```
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 u
"MAV_CMD_NAV_LOITER_TURNS"	18	"Loiter around this waypoint for
"MAV_CMD_NAV_LOITER_TIME"	19	"Loiter around this waypoint for
"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 fr
"MAV_CMD_NAV_TAKEOFF_LOCAL"	24	"Takeoff from local position (loc
"MAV_CMD_NAV_FOLLOW"	25	"Vehicle following, i.e. this way
"MAV_CMD_NAV_CONTINUE_AND_CHANGE_ALT"	30	"Continue on the current course a
"MAV_CMD_NAV_LOITER_TO_ALT"	31	"Begin loiter at the specified La

```

"MAV_CMD_DO_FOLLOW"          32      "Being following a target"
"MAV_CMD_DO_FOLLOW_REPOSITION" 33      "Reposition the MAV after a follow"
"MAV_CMD_DO_ORBIT"          34      "Start orbiting on the circumference"
"MAV_CMD_NAV_ROI"           80      "Sets the region of interest (ROI)"
:

```

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 a
```

```
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`.

---

**Note** This function requires you to install the UAV Library for Robotics System Toolbox. To install add-ons, use `roboticsAddons` and select the desired add-on.

---

## 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 entires.

```
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 u
"MAV_CMD_NAV_LOITER_TURNS"	18	"Loiter around this waypoint for
"MAV_CMD_NAV_LOITER_TIME"	19	"Loiter around this waypoint for
"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 fr
"MAV_CMD_NAV_TAKEOFF_LOCAL"	24	"Takeoff from local position (loc
"MAV_CMD_NAV_FOLLOW"	25	"Vehicle following, i.e. this way
"MAV_CMD_NAV_CONTINUE_AND_CHANGE_ALT"	30	"Continue on the current course a
"MAV_CMD_NAV_LOITER_TO_ALT"	31	"Begin loiter at the specified La

```

"MAV_CMD_DO_FOLLOW"           32      "Being following a target"
"MAV_CMD_DO_FOLLOW_REPOSITION" 33      "Reposition the MAV after a follow"
"MAV_CMD_DO_ORBIT"           34      "Start orbiting on the circumference"
"MAV_CMD_NAV_ROI"            80      "Sets the region of interest (ROI)"
:

```

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
```

```
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

**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.

```
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@0.0.0.0:64627"

```
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**

“Use a MAVLink Parameter Protocol for Tuning UAV Parameters in MATLAB”

## **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

**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.

```
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@0.0.0.0:64627"
```

```
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**

“Use a MAVLink Parameter Protocol for Tuning UAV Parameters in MATLAB”

## **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@0.0.0.0:64627"

```
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=1×4 table
```

<u>SystemID</u>	<u>ComponentID</u>	<u>ComponentType</u>	<u>AutopilotType</u>
255	1	"MAV_TYPE_GCS"	"MAV_AUTOPILOT_INVALID"

```
listConnections(mavlink)
```

```
ans=1×2 table
```

<u>ConnectionName</u>	<u>ConnectionInfo</u>
"Connection1"	"UDP@0.0.0.0:64627"

```
listTopics(mavlink)
```

```
ans =
```

```
0×5 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@0.0.0.0:64627"

```
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@0.0.0.0:64627"
```

```
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

“Use a MAVLink Parameter Protocol for Tuning UAV Parameters in MATLAB”

## 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**

# 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.

---

**Note** This function requires you to install the UAV Library for Robotics System Toolbox. To install add-ons, use `roboticsAddons` and select the desired add-on.

---

## Examples

### Subscribe to MAVLink Topic

**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.

```
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.

---

**Note** This function requires you to install the UAV Library for Robotics System Toolbox. To install add-ons, use `roboticsAddons` and select the desired add-on.

---

## 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 entires.

```
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 u
"MAV_CMD_NAV_LOITER_TURNS"	18	"Loiter around this waypoint for
"MAV_CMD_NAV_LOITER_TIME"	19	"Loiter around this waypoint for
"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 fr
"MAV_CMD_NAV_TAKEOFF_LOCAL"	24	"Takeoff from local position (loc
"MAV_CMD_NAV_FOLLOW"	25	"Vehicle following, i.e. this way
"MAV_CMD_NAV_CONTINUE_AND_CHANGE_ALT"	30	"Continue on the current course a
"MAV_CMD_NAV_LOITER_TO_ALT"	31	"Begin loiter at the specified La

---

```

"MAV_CMD_DO_FOLLOW"           32      "Being following a target"
"MAV_CMD_DO_FOLLOW_REPOSITION" 33      "Reposition the MAV after a follow"
"MAV_CMD_DO_ORBIT"           34      "Start orbiting on the circumference"
"MAV_CMD_NAV_ROI"            80      "Sets the region of interest (ROI)"
:

```

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 a
```

```
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

“Load and Playback MAVLink TLOG”

**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**

# 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

"Use a MAVLink Parameter Protocol for Tuning UAV Parameters in MATLAB"

## External Websites

[MAVLink Developer Guide](#)

**Introduced in R2019a**



# Methods — Alphabetical List

---

# copy

**Class:** robotics.BinaryOccupancyGrid

**Package:** robotics

Copy array of handle objects

## Syntax

```
b = copy(a)
```

## Description

`b = copy(a)` copies each element in the array of handles, `a`, to the new array of handles, `b`.

The `copy` method does not copy dependent properties. MATLAB does not call `copy` recursively on any handles contained in property values. MATLAB does not call the class constructor or property set methods during the copy operation.

`b` has the same number of elements and is the same size and class of `a`. `b` is the same class as `a`. If `a` is empty, `b` is also empty. If `a` is heterogeneous, `b` is also heterogeneous. If `a` contains deleted handles, then `copy` creates deleted handles of the same class in `b`. Dynamic properties and listeners associated with objects in `a` are not copied to objects in `b`.

`copy` is a sealed and public method in class `matlab.mixin.Copyable`.

## Input Arguments

**a — Object array**

handle

Object array, specified as a handle.



## Output Arguments

**b** — Object array containing copies of the objects in **a**  
`handle`

Object array containing copies of the object in **a**, specified as a `handle`.

## See Also

`robotics.BinaryOccupancyGrid`

**Introduced in R2015a**

## getOccupancy

**Class:** robotics.BinaryOccupancyGrid

**Package:** robotics

Get occupancy value for one or more positions

## Syntax

```
occval = getOccupancy(map,xy)
occval = getOccupancy(map,ij,"grid")
```

## Description

`occval = getOccupancy(map,xy)` returns an array of occupancy values for an input array of world coordinates, `xy`. Each row of `xy` is a point in the world, represented as an `[x y]` coordinate pair. `occval` is the same length as `xy` and a single column array. An occupied location is represented as `true` (1), and a free location is represented as `false` (0).

`occval = getOccupancy(map,ij,"grid")` returns an array of occupancy values based on a `[rows cols]` input array of grid positions, `ij`.

## Input Arguments

### **map — Map representation**

BinaryOccupancyGrid object

Map representation, specified as a `robotics.BinaryOccupancyGrid` object. This object represents the environment of the robot. The object contains a matrix grid with binary values indicating obstacles as `true` (1) and free locations as `false` (0).

### **xy — World coordinates**

*n*-by-2 vertical array

World coordinates, specified as an  $n$ -by-2 vertical array of  $[x \ y]$  pairs, where  $n$  is the number of world coordinates.

Data Types: double

### **ij — Grid positions**

$n$ -by-2 vertical array

Grid positions, specified as an  $n$ -by-2 vertical array of  $[i \ j]$  pairs in  $[\text{rows} \ \text{cols}]$  format, where  $n$  is the number of grid positions.

Data Types: double

## **Output Arguments**

### **occval — Occupancy values**

$n$ -by-1 vertical array

Occupancy values of the same length as either  $xy$  or  $ij$ , returned as an  $n$ -by-1 vertical array, where  $n$  is the same  $n$  in either  $xy$  or  $ij$ .

## **See Also**

`robotics.BinaryOccupancyGrid` |  
`robotics.BinaryOccupancyGrid.setOccupancy`

**Introduced in R2015a**

## grid2world

**Class:** robotics.BinaryOccupancyGrid

**Package:** robotics

Convert grid indices to world coordinates

### Syntax

```
xy = grid2world(map,ij)
```

### Description

`xy = grid2world(map,ij)` converts a `[row col]` array of grid indices, `ij`, to an array of world coordinates, `xy`.

### Input Arguments

**map — Map representation**

BinaryOccupancyGrid object

Map representation, specified as a `robotics.BinaryOccupancyGrid` object. This object represents the environment of the robot. The object contains a matrix grid with binary values indicating obstacles as `true` (1) and free locations as `false` (0).

**ij — Grid positions**

*n*-by-2 vertical array

Grid positions, specified as an *n*-by-2 vertical array of `[i j]` pairs in `[rows cols]` format, where *n* is the number of grid positions.

### Output Arguments

**xy — World coordinates**

*n*-by-2 vertical array

World coordinates, specified as an  $n$ -by-2 vertical array of  $[x \ y]$  pairs, where  $n$  is the number of world coordinates.

## **See Also**

`robotics.BinaryOccupancyGrid` | `robotics.BinaryOccupancyGrid.world2grid`

**Introduced in R2015a**

# inflate

**Class:** `robotics.BinaryOccupancyGrid`

**Package:** `robotics`

Inflate each occupied grid location

## Syntax

```
inflate(map, radius)
inflate(map, gridradius, 'grid')
```

## Description

`inflate(map, radius)` inflates each occupied position of the map by the radius given in meters. `radius` is rounded up to the nearest cell equivalent based on the resolution of the map. Every cell within the radius is set to `true` (1).

`inflate(map, gridradius, 'grid')` inflates each occupied position by the radius given in number of cells.

## Input Arguments

### **map — Map representation**

`BinaryOccupancyGrid` object

Map representation, specified as a `robotics.BinaryOccupancyGrid` object. This object represents the environment of the robot. The object contains a matrix grid with binary values indicating obstacles as `true` (1) and free locations as `false` (0).

### **radius — Dimension the defines how much to inflate occupied locations**

scalar

Dimension that defines how much to inflate occupied locations, specified as a scalar. `radius` is rounded up to the nearest cell value.

Data Types: double

**gridradius — Dimension the defines how much to inflate occupied locations**

positive scalar

Dimension that defines how much to inflate occupied locations, specified as a positive scalar. `gridradius` is the number of cells to inflate the occupied locations.

Data Types: double

## See Also

`robotics.BinaryOccupancyGrid` |

`robotics.BinaryOccupancyGrid.setOccupancy`

**Introduced in R2015a**

# occupancyMatrix

**Class:** robotics.BinaryOccupancyGrid

**Package:** robotics

Convert occupancy grid to matrix

## Syntax

```
mat = occupancyMatrix(map)
```

## Description

`mat = occupancyMatrix(map)` returns occupancy values stored in the occupancy grid object as a matrix.

## Input Arguments

**map — Map representation**

BinaryOccupancyGrid object

Map representation, specified as a `robotics.BinaryOccupancyGrid` object. This object represents the environment of the robot. The object contains a matrix grid with binary values indicating obstacles as `true` (1) and free locations as `false` (0).

## Output Arguments

**mat — Occupancy values**

matrix

Occupancy values, returned as an  $h$ -by- $w$  matrix, where  $h$  and  $w$  are defined by the two elements of the `GridSize` property of the occupancy grid object.



## See Also

`getOccupancy` | `show` | `robotics.BinaryOccupancyGrid` |  
`robotics.OccupancyGrid`

## Topics

“Occupancy Grids”

**Introduced in R2016b**

# setOccupancy

**Class:** robotics.BinaryOccupancyGrid

**Package:** robotics

Set occupancy value for one or more positions

## Syntax

```
setOccupancy(map,xy,occval)  
setOccupancy(map,ij,occval,"grid")
```

## Description

`setOccupancy(map,xy,occval)` assigns occupancy values, `occval`, to the input array of world coordinates, `xy` in the occupancy grid, `map`. Each row of the array, `xy`, is a point in the world and is represented as an `[x y]` coordinate pair. `occval` is either a scalar or a single column array of the same length as `xy`. An occupied location is represented as `true` (1), and a free location is represented as `false` (0).

`setOccupancy(map,ij,occval,"grid")` assigns occupancy values, `occval`, to the input array of grid indices, `ij`, as `[rows cols]`.

## Input Arguments

### **map — Map representation**

BinaryOccupancyGrid object

Map representation, specified as a `robotics.BinaryOccupancyGrid` object. This object represents the environment of the robot. The object contains a matrix grid with binary values indicating obstacles as `true` (1) and free locations as `false` (0).

### **xy — World coordinates**

*n*-by-2 vertical array

World coordinates, specified as an  $n$ -by-2 vertical array of  $[x \ y]$  pairs, where  $n$  is the number of world coordinates.

Data Types: double

### **ij** — Grid positions

$n$ -by-2 vertical array

Grid positions, specified as an  $n$ -by-2 vertical array of  $[i \ j]$  pairs in `[rows cols]` format, where  $n$  is the number of grid positions.

Data Types: double

### **occval** — Occupancy values

$n$ -by-1 vertical array

Occupancy values of the same length as either `xy` or `ij`, returned as an  $n$ -by-1 vertical array, where  $n$  is the same  $n$  in either `xy` or `ij`.

## Examples

### **Set Occupancy Values**

Set the occupancy of grid locations using `setOccupancy`.

Initialize an occupancy grid object using `BinaryOccupancyGrid`.

```
map = robotics.BinaryOccupancyGrid(10,10);
```

Set the occupancy of a specific location using `setOccupancy`.

```
setOccupancy(map,[8 8],1);
```

Set the occupancy of an array of locations.

```
[x,y] = meshgrid(2:5);  
setOccupancy(map,[x(:) y(:)],1);
```

## **See Also**

`robotics.BinaryOccupancyGrid` |  
`robotics.BinaryOccupancyGrid.getOccupancy`

**Introduced in R2015a**

# show

**Class:** robotics.BinaryOccupancyGrid

**Package:** robotics

Show occupancy grid values

## Syntax

```
show(map)
```

```
show(map, "grid")
```

```
show( ____, "Parent", parent)
```

```
h = show(map, ____)
```

## Description

`show(map)` displays the binary occupancy grid `map` in the current axes, with the axes labels representing the world coordinates.

`show(map, "grid")` displays the binary occupancy grid `map` in the current axes, with the axes labels representing the grid coordinates.

`show( ____, "Parent", parent)` sets the specified axes handle `parent` to the axes, using any of the arguments from previous syntaxes.

`h = show(map, ____)` returns the figure object handle created by `show`.

## Input Arguments

**map** — Map representation

BinaryOccupancyGrid object

Map representation, specified as a `robotics.BinaryOccupancyGrid` object. This object represents the environment of the robot. The object contains a matrix grid with binary values indicating obstacles as `true` (1) and free locations as `false` (0).

**parent — Axes to plot the map**

Axes object | UIAxes object

Axes to plot the map specified as either an Axes or UIAxes object. See `axes` or `uiaxes`.

## See Also

`robotics.BinaryOccupancyGrid`

**Introduced in R2015a**

# world2grid

**Class:** robotics.BinaryOccupancyGrid

**Package:** robotics

Convert world coordinates to grid indices

## Syntax

```
ij = world2grid(map,xy)
```

## Description

`ij = world2grid(map,xy)` converts an array of world coordinates, `xy`, to a [rows cols] array of grid indices, `ij`.

## Input Arguments

### **map** — Map representation

BinaryOccupancyGrid object

Map representation, specified as a `robotics.BinaryOccupancyGrid` object. This object represents the environment of the robot. The object contains a matrix grid with binary values indicating obstacles as `true` (1) and free locations as `false` (0).

### **xy** — World coordinates

*n*-by-2 vertical array

World coordinates, specified as an *n*-by-2 vertical array of [x y] pairs, where *n* is the number of world coordinates.

## Output Arguments

### **ij** — Grid positions

*n*-by-2 vertical array

Grid positions, specified as an  $n$ -by-2 vertical array of `[i j]` pairs in `[rows cols]` format, where  $n$  is the number of grid positions.

### **See Also**

`robotics.BinaryOccupancyGrid` | `robotics.BinaryOccupancyGrid.grid2world`

**Introduced in R2015a**



## copy

**Class:** `robotics.Joint`

**Package:** `robotics`

Create copy of joint

## Syntax

```
jCopy = copy(jointObj)
```

## Description

`jCopy = copy(jointObj)` creates a copy of the `Joint` object with the same properties.

## Input Arguments

**jointObj** — Joint object

handle

Joint object, specified as a handle. Create a joint object using `robotics.Joint`.

## Output Arguments

**jCopy** — Joint object

handle

Joint object, returned as a handle. Create a joint object using `robotics.Joint`. This copy has the same properties.

## See Also

`robotics.Joint` | `robotics.RigidBody` | `robotics.RigidBodyTree`

**Introduced in R2016b**

# setFixedTransform

**Class:** robotics.Joint

**Package:** robotics

Set fixed transform properties of joint

## Syntax

```
setFixedTransform(jointObj, tform)
```

```
setFixedTransform(jointObj, dhparams, "dh")
```

```
setFixedTransform(jointObj, mdhparams, "mdh")
```

## Description

`setFixedTransform(jointObj, tform)` sets the `JointToParentTransform` property of the `Joint` object directly with the supplied homogenous transformation.

`setFixedTransform(jointObj, dhparams, "dh")` sets the `ChildToJointTransform` property using Denavit-Hartenberg (DH) parameters. The `JointToParentTransform` property is set to an identity matrix. DH parameters are given in the order [a alpha d theta].

The `theta` input is ignored when specifying the fixed transformation between joints because that angle is dependent on the joint configuration. For more information, see "Rigid Body Tree Robot Model".

`setFixedTransform(jointObj, mdhparams, "mdh")` sets the `JointToParentTransform` property using modified DH parameters. The `ChildToJointTransform` property is set to an identity matrix. Modified DH parameters are given in the order [a alpha d theta].

## Input Arguments

### **jointObj** — Joint object

handle

Joint object, specified as a handle. Create a joint object using `robotics.Joint`.

### **tform** — Homogeneous transform

4-by-4 matrix

Homogeneous transform, specified as a 4-by-4 matrix. The transform is set to the `ChildToJointTransform` property. The `JointToParentTransform` property is set to an identity matrix.

### **dhparams** — Denavit-Hartenberg (DH) parameters

four-element vector

Denavit-Hartenberg (DH) parameters, specified as a four-element vector, [`a` `alpha` `d` `theta`]. These parameters are used to set the `ChildToJointTransform` property. The `JointToParentTransform` property is set to an identity matrix.

The `theta` input is ignored when specifying the fixed transformation between joints because that angle is dependent on the joint configuration. For more information, see “Rigid Body Tree Robot Model”.

### **mdhparams** — Modified Denavit-Hartenberg (DH) parameters

four-element vector

Modified Denavit-Hartenberg (DH) parameters, specified as a four-element vector, [`a` `alpha` `d` `theta`]. These parameters are used to set the `JointToParentTransform` property. The `ChildToJointTransform` is set to an identity matrix.

The `theta` input is ignored when specifying the fixed transformation between joints because that angle is dependent on the joint configuration. For more information, see “Rigid Body Tree Robot Model”.

## Examples

## Build Manipulator Robot Using Denavit-Hartenberg Parameters

Use the Denavit-Hartenberg (DH) parameters of the Puma560® robot to build a robot. Each rigid body is added one at a time, with the child-to-parent transform specified by the joint object.

The DH parameters define the geometry of the robot with relation to how each rigid body is attached to its parent. For convenience, setup the parameters for the Puma560 robot in a matrix. The Puma robot is a serial chain manipulator. The DH parameters are relative to the previous line in the matrix, corresponding to the previous joint attachment.

```
dhparams = [0      pi/2    0      0;
            0.4318  0      0      0
            0.0203  -pi/2   0.15005  0;
            0      pi/2    0.4318  0;
            0      -pi/2   0      0;
            0      0      0      0];
```

Create a rigid body tree object to build the robot.

```
robot = robotics.RigidBodyTree;
```

Create the first rigid body and add it to the robot. To add a rigid body:

- 1 Create a `RigidBody` object and give it a unique name.
- 2 Create a `Joint` object and give it a unique name.
- 3 Use `setFixedTransform` to specify the body-to-body transformation using DH parameters. The last element of the DH parameters, `theta`, is ignored because the angle is dependent on the joint position.
- 4 Call `addBody` to attach the first body joint to the base frame of the robot.

```
body1 = robotics.RigidBody('body1');
jnt1 = robotics.Joint('jnt1','revolute');

setFixedTransform(jnt1,dhparams(1,:), 'dh');
body1.Joint = jnt1;

addBody(robot,body1, 'base')
```

Create and add other rigid bodies to the robot. Specify the previous body name when calling `addBody` to attach it. Each fixed transform is relative to the previous joint coordinate frame.

```

body2 = robotics.RigidBody('body2');
jnt2 = robotics.Joint('jnt2','revolute');
body3 = robotics.RigidBody('body3');
jnt3 = robotics.Joint('jnt3','revolute');
body4 = robotics.RigidBody('body4');
jnt4 = robotics.Joint('jnt4','revolute');
body5 = robotics.RigidBody('body5');
jnt5 = robotics.Joint('jnt5','revolute');
body6 = robotics.RigidBody('body6');
jnt6 = robotics.Joint('jnt6','revolute');

setFixedTransform(jnt2,dhparams(2,:), 'dh');
setFixedTransform(jnt3,dhparams(3,:), 'dh');
setFixedTransform(jnt4,dhparams(4,:), 'dh');
setFixedTransform(jnt5,dhparams(5,:), 'dh');
setFixedTransform(jnt6,dhparams(6,:), 'dh');

body2.Joint = jnt2;
body3.Joint = jnt3;
body4.Joint = jnt4;
body5.Joint = jnt5;
body6.Joint = jnt6;

addBody(robot,body2,'body1')
addBody(robot,body3,'body2')
addBody(robot,body4,'body3')
addBody(robot,body5,'body4')
addBody(robot,body6,'body5')

```

Verify that your robot was built properly by using the `showdetails` or `show` function. `showdetails` lists all the bodies in the MATLAB® command window. `show` displays the robot with a given configuration (home by default). Calls to `axis` modify the axis limits and hide the axis labels.

```
showdetails(robot)
```

```

-----
Robot: (6 bodies)

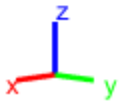
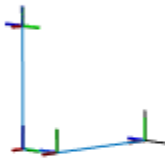
```

Idx	Body Name	Joint Name	Joint Type	Parent Name(Idx)	Children Name(s)
1	body1	jnt1	revolute	base(0)	body2(2)
2	body2	jnt2	revolute	body1(1)	body3(3)
3	body3	jnt3	revolute	body2(2)	body4(4)
4	body4	jnt4	revolute	body3(3)	body5(5)

```
5      body5      jnt5      revolute      body4(4)  body6(6)
6      body6      jnt6      revolute      body5(5)
```

-----

```
show(robot);
axis([-0.5,0.5,-0.5,0.5,-0.5,0.5])
axis off
```



## References

[1] Craig, John J. *Introduction to Robotics: Mechanics and Control*. Reading, MA: Addison-Wesley, 1989.

[2] Siciliano, Bruno. *Robotics: Modelling, Planning and Control*. London: Springer, 2009.

## See Also

`robotics.Joint` | `robotics.RigidBody` | `robotics.RigidBodyTree`

**Introduced in R2016b**



# addScan

**Class:** robotics.LidarSLAM

**Package:** robotics

Add scan to lidar SLAM map

## Syntax

```
addScan(slamObj, currScan)
addScan(slamObj, currScan, relPoseEst)
[isAccepted, loopClosureInfo, optimInfo] = addScan( ___ )
```

## Description

`addScan(slamObj, currScan)` adds a lidar scan, `currScan`, to the lidar SLAM object, `slamObj`. The function uses scan matching to correlate this scan to the most recent one, then adds it to the pose graph defined in `slamObj`. If the scan is accepted, `addScan` detects loop closures and optimizes based on settings in `slamObj`.

`addScan(slamObj, currScan, relPoseEst)` also specifies a relative pose to the latest lidar scan pose in `slamObj`. This relative pose improves the scan matching.

`[isAccepted, loopClosureInfo, optimInfo] = addScan( ___ )` outputs detailed information about adding the scan to the SLAM object. `isAccepted` indicates if the scan is added or rejected. `loopClosureInfo` and `optimInfo` indicate if a loop closure is detected or the pose graph is optimized.

## Examples

### Perform SLAM Using Lidar Scans

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

#### Load Data and Set Up SLAM Algorithm

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

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

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

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

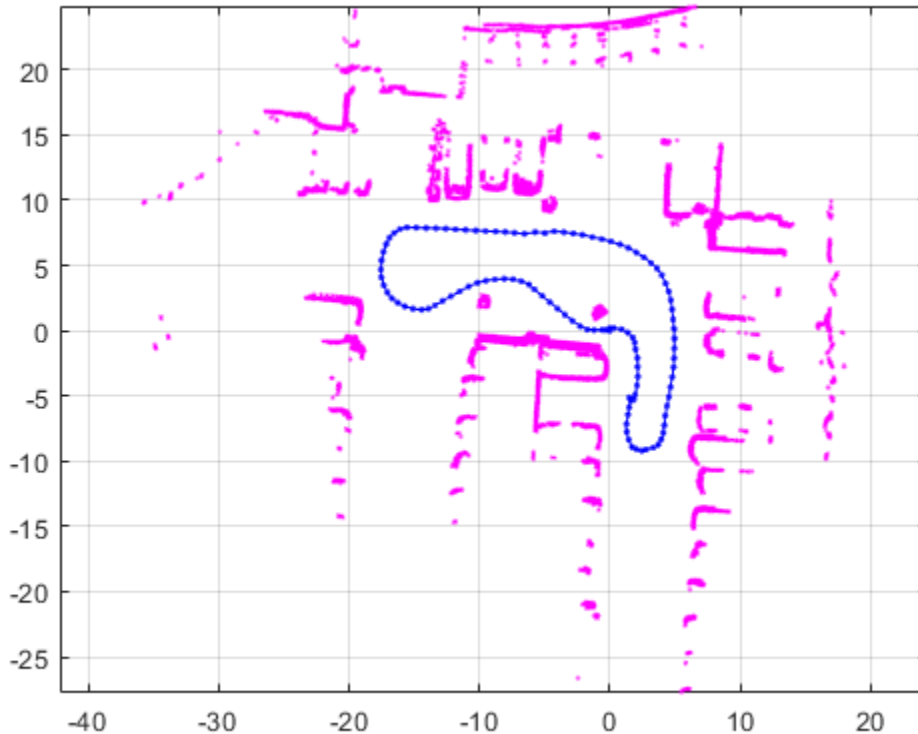
slamObj = robotics.LidarSLAM(resolution,maxRange);
slamObj.LoopClosureThreshold = 360;
slamObj.LoopClosureSearchRadius = 8;
```

#### Add Scans Iteratively

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

```
for i = 1:numel(scans)
    addScan(slamObj,scans{i});

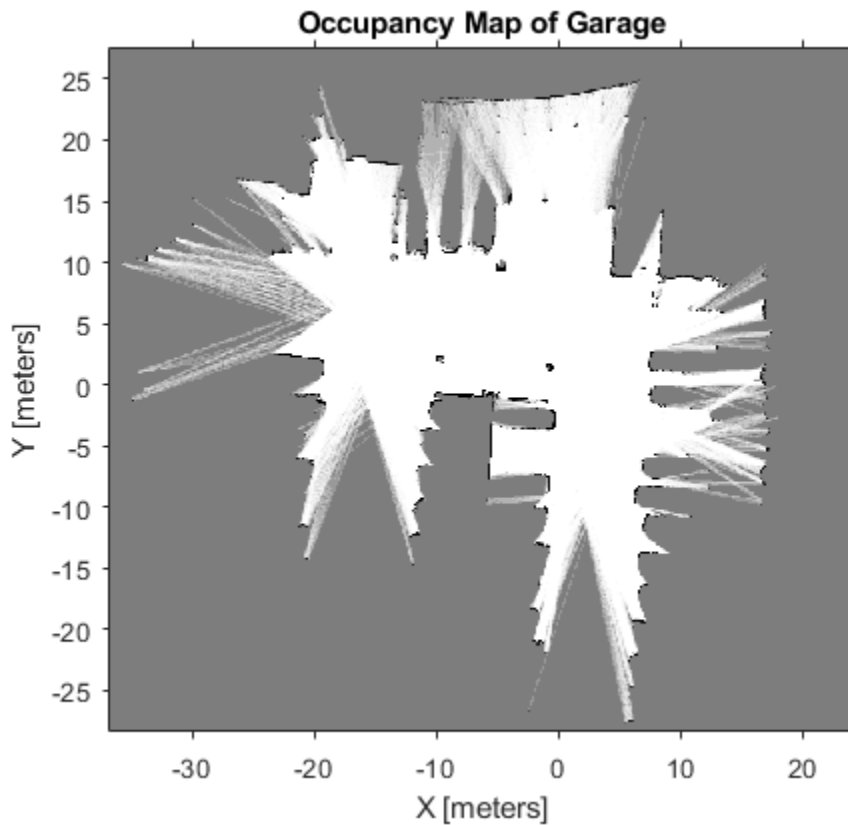
    if rem(i,10) == 0
        show(slamObj);
    end
end
```



### View Occupancy Map

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

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



## Input Arguments

**sLAMobj** — Lidar SLAM object

LidarSLAM object

Lidar SLAM object, specified as a LidarSLAM object. The object contains the SLAM algorithm parameters, sensor data, and underlying pose graph used to build the map.

**currScan** — Lidar scan reading

lidarScan object

Lidar scan reading, specified as a `lidarScan` object. This scan is correlated to the most recent scan in `slamObj` using scan matching.

#### **relPoseEst** — Relative pose estimate of scan

`[x y theta]` vector

Relative pose estimate of scan, specified as an `[x y theta]` vector. This relative pose improves scan matching.

## Output Arguments

#### **isAccepted** — Indicates if scan is accepted

`true` | `false`

Indicates if scan is accepted, returned as `true` or `false`. If the relative pose between scans is below the `MovementThreshold` property of `slamObj`, the scan is rejected. By default, all scans are accepted.

#### **loopClosureInfo** — Loop closure details

structure

Loop closure details, returned as a structure with these fields:

- **EdgeIDs** -- IDs of newly connected edges in the pose graph, returned as a vector.
- **Edges** -- Newly added loop closure edges, returned as an  $n$ -by-2 matrix of node IDs that each edge connects.
- **Scores** -- Scores of newly connected edges in the pose graph returned from scan matching, returned as a vector.

---

**Note** If the `LoopClosureAutoRollback` property is set to `true` in `slamObj`, loop closure edges can be removed from the pose graph. This property rejects loops closures if the residual error changes drastically after optimization. Therefore, some of the edge IDs listed in this structure may not exist in the actual pose graph.

---

#### **optimInfo** — Pose graph optimization details

structure

Pose graph optimization details, returned as a structure with these fields:

- `IsPerformed` -- Boolean indicating if optimization is performed when adding this scan. Optimization performance depends on the `OptimizationInterval` property in `slamObj`.
- `IsAccepted` -- Boolean indicating if optimization was accepted based on `ResidualError`.
- `ResidualError` -- Error associated with optimization, returned as a scalar.
- `LoopClosureRemoved` -- List of IDs of loop closure edges removed during optimization, returned as a vector.

## Extended Capabilities

### C/C++ Code Generation

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

Use this syntax when constructing `LidarSLAM` objects for code generation:

```
slamObj = robotics.LidarSLAM(mapResolution, maxLidarRange, maxNumScans)
```

specifies the upper bound on the number of accepted scans allowed when generating code. `maxNumScans` is a positive integer. This scan limit is only required when generating code.

### See Also

`optimizePoseGraph` | `robotics.PoseGraph`

### Topics

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

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

**Introduced in R2018a**

---

## copy

**Class:** robotics.LidarSLAM

**Package:** robotics

Copy lidar SLAM object

## Syntax

```
newSlamObj = copy(slamObj)
```

## Description

`newSlamObj = copy(slamObj)` creates a deep copy of `slamObj` with the same properties. Any changes made to `newSlamObj` are not reflected in `slamObj`.

## Input Arguments

**slamObj** — Lidar SLAM object

LidarSLAM object

Lidar SLAM object, specified as a LidarSLAM object. The object contains the SLAM algorithm parameters, sensor data, and underlying pose graph used to build the map.

## Output Arguments

**newSlamObj** — Lidar SLAM object

LidarSLAM object

Lidar SLAM object, returned as a LidarSLAM object.

## Extended Capabilities

### C/C++ Code Generation

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

Use this syntax when constructing LidarSLAM objects for code generation:

```
slamObj= robotics.LidarSLAM(mapResolution,maxLidarRange,maxNumScans)
```

specifies the upper bound on the number of accepted scans allowed when generating code. `maxNumScans` is a positive integer. This scan limit is only required when generating code.

### See Also

`optimizePoseGraph` | `robotics.PoseGraph`

### Topics

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

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

**Introduced in R2018a**



# removeLoopClosures

**Class:** robotics.LidarSLAM

**Package:** robotics

Remove loop closures from pose graph

## Syntax

```
removeLoopClosures(slamObj)  
removeLoopClosures(slamObj, lcEdgeIDs)
```

## Description

`removeLoopClosures(slamObj)` removes all loop closures from the underlying pose graph in `slamObj`.

`removeLoopClosures(slamObj, lcEdgeIDs)` removes the loop closure edges with the specified IDs from the underlying pose graph in `slamObj`.

## Input Arguments

**slamObj** — Lidar SLAM object

LidarSLAM object

Lidar SLAM object, specified as a LidarSLAM object. The object contains the SLAM algorithm parameters, sensor data, and underlying pose graph used to build the map

**lcEdgeIDs** — Loop closure edge IDs

vector of positive integers

Loop closure edge IDs, specified as a vector of positive integers. To find specific edge IDs, use `findEdgeID` on the underlying PoseGraph object defined in `slamObj`.

## Extended Capabilities

### C/C++ Code Generation

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

Use this syntax when constructing LidarSLAM objects for code generation:

`slamObj= robotics.LidarSLAM(mapResolution,maxLidarRange,maxNumScans)`  
specifies the upper bound on the number of accepted scans allowed when generating code. `maxNumScans` is a positive integer. This scan limit is only required when generating code.

### See Also

`optimizePoseGraph` | `robotics.PoseGraph`

### Topics

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

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

**Introduced in R2018a**

# scansAndPoses

**Class:** robotics.LidarSLAM

**Package:** robotics

Extract scans and corresponding poses

## Syntax

```
[scans, poses] = scansAndPoses(slamObj)
[scans, poses] = scansAndPoses(slamObj, nodeIDs)
```

## Description

`[scans, poses] = scansAndPoses(slamObj)` returns the scans used by the LidarSLAM object as `lidarScan` objects, along with their associated `[x y theta]` poses from the underlying pose graph of `slamObj`.

`[scans, poses] = scansAndPoses(slamObj, nodeIDs)` returns the scans and poses for the specific node IDs. To get the node IDs, see the underlying `PoseGraph` object in `slamObj` for the node IDs.

## Examples

### Perform SLAM Using Lidar Scans

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

### Load Data and Set Up SLAM Algorithm

Load a cell array of `lidarScan` objects. The lidar scans were collected in a parking garage on a Husky® robot from ClearPath Robotics®. Typically, lidar scans are taken at a

high frequency and each scan is not needed for SLAM. Therefore, down sample the scans by selecting only every 40th scan.

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

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

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

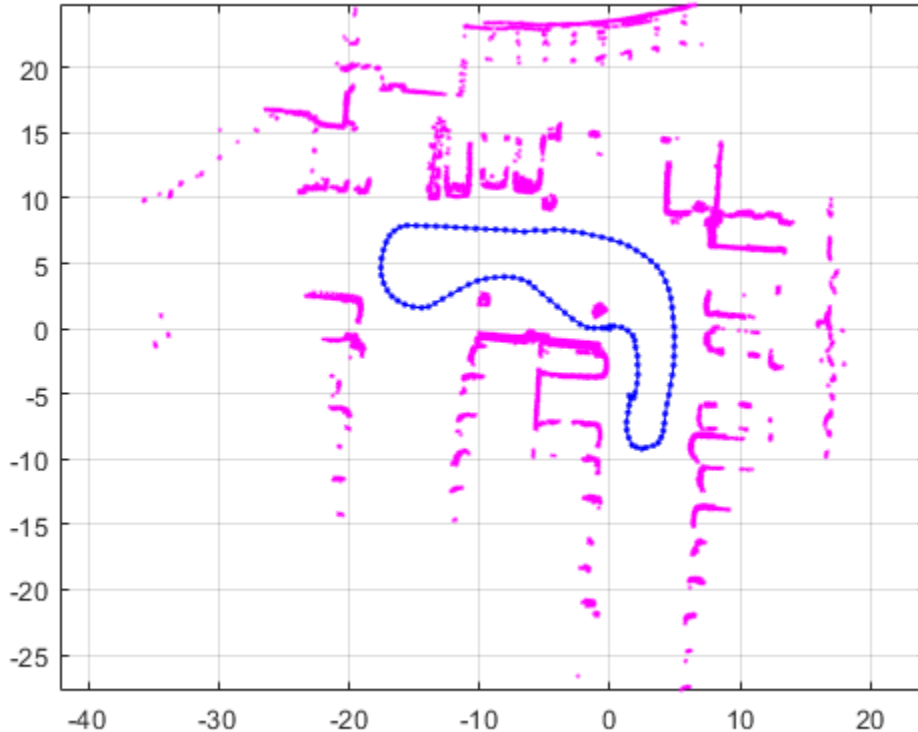
slamObj = robotics.LidarSLAM(resolution,maxRange);
slamObj.LoopClosureThreshold = 360;
slamObj.LoopClosureSearchRadius = 8;
```

#### **Add Scans Iteratively**

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

```
for i = 1:numel(scans)
    addScan(slamObj,scans{i});

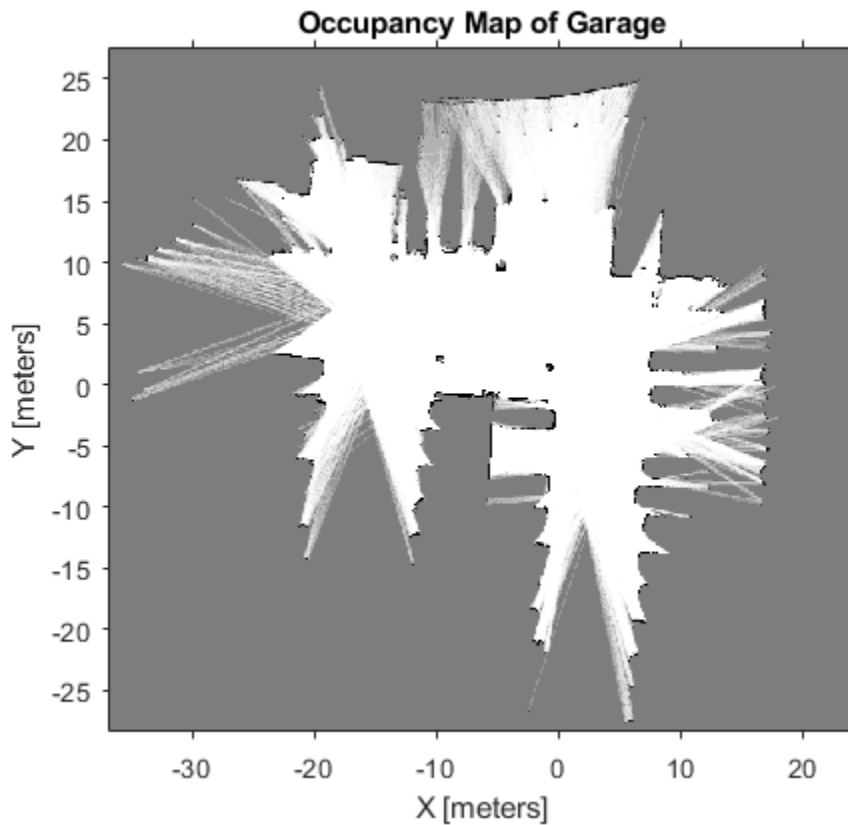
    if rem(i,10) == 0
        show(slamObj);
    end
end
```



### View Occupancy Map

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

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



## Input Arguments

**slamObj** — Lidar SLAM object

LidarSLAM object

Lidar SLAM object, specified as a LidarSLAM object. The object contains the SLAM algorithm parameters, sensor data, and underlying pose graph used to build the map.

**nodeIDs** — Node IDs from pose graph

positive integer

Node IDs from pose graph, specified as a positive integer. Nodes are added to the pose graph with sequential ID numbers. To get the node IDs, see the underlying PoseGraph object in `slamObj` for the node IDs.

## Output Arguments

### **scans** — Lidar scan readings

`lidarScan` object

Lidar scan readings, returned as a `lidarScan` object.

### **poses** — Pose for each scan

$n$ -by-3 matrix | [`x` `y` `theta`] vectors

Pose for each scan, returned as an  $n$ -by-3 matrix of [`x` `y` `theta`] vectors. Each row is a pose that corresponds to a scan in `scans`.

## Extended Capabilities

### C/C++ Code Generation

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

Use this syntax when constructing `LidarSLAM` objects for code generation:

```
slamObj= robotics.LidarSLAM(mapResolution,maxLidarRange,maxNumScans)
```

specifies the upper bound on the number of accepted scans allowed when generating code. `maxNumScans` is a positive integer. This scan limit is only required when generating code.

### See Also

`optimizePoseGraph` | `robotics.PoseGraph`

### Topics

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

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

**Introduced in R2018a**



# show

**Class:** robotics.LidarSLAM

**Package:** robotics

Plot scans and robot poses

## Syntax

```
show(slamObj)
show(slamObj, Name, Value)
axes = show( ___ )
```

## Description

`show(slamObj)` plots all the scans added to the input LidarSLAM object overlaid with the lidar poses in its underlying pose graph.

`show(slamObj, Name, Value)` specifies options using Name, Value pair arguments. For example, "Poses", "off" turns off display of the underlying pose graph in slamObj.

`axes = show( ___ )` returns the axes handle that the lidar SLAM data is plotted to using any of the previous syntaxes.

## Examples

### Perform SLAM Using Lidar Scans

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

### Load Data and Set Up SLAM Algorithm

Load a cell array of `lidarScan` objects. The lidar scans were collected in a parking garage on a Husky® robot from ClearPath Robotics®. Typically, lidar scans are taken at a

high frequency and each scan is not needed for SLAM. Therefore, down sample the scans by selecting only every 40th scan.

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

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

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

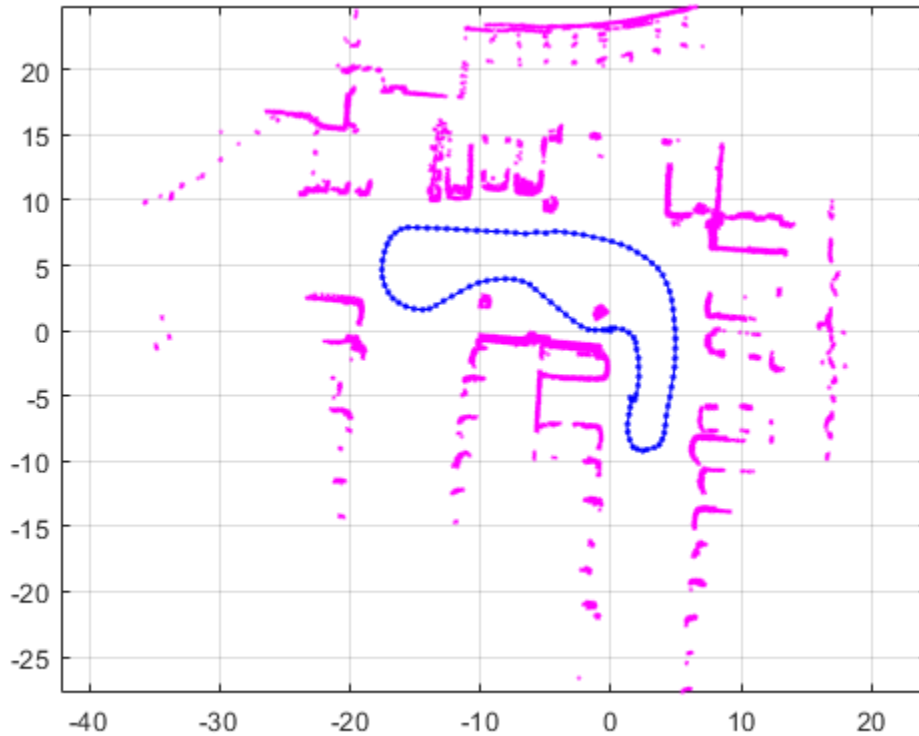
slamObj = robotics.LidarSLAM(resolution,maxRange);
slamObj.LoopClosureThreshold = 360;
slamObj.LoopClosureSearchRadius = 8;
```

#### Add Scans Iteratively

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

```
for i = 1:numel(scans)
    addScan(slamObj,scans{i});

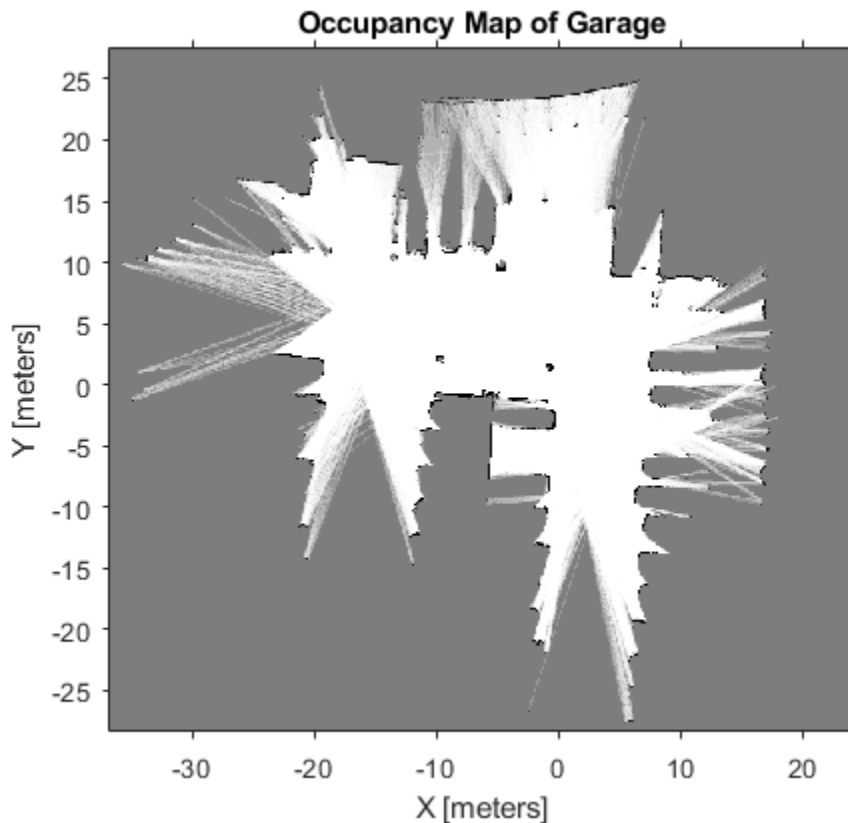
    if rem(i,10) == 0
        show(slamObj);
    end
end
```



### View Occupancy Map

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

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



## Input Arguments

### `slamObj` — Lidar SLAM object

LidarSLAM object

Lidar SLAM object, specified as a LidarSLAM object. The object contains the SLAM algorithm parameters, sensor data, and underlying pose graph used to build the map.

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes.

You can specify several name and value pair arguments in any order as `Name1, Value1, . . . , NameN, ValueN`.

Example: `"Poses", "off"`

### **Parent — Axes used to plot pose graph**

`Axes` object | `UIAxes` object

Axes used to plot the pose graph, specified as the comma-separated pair consisting of `"Parent"` and either an `Axes` or `UIAxes` object. See `axes` or `uiaxes`.

### **Poses — Display lidar poses**

`"on"` (default) | `"off"`

Display lidar poses, specified as the comma-separated pair consisting of `"Poses"` and `"on"` or `"off"`.

## **Output Arguments**

### **axes — Axes used to plot the map**

`Axes` object | `UIAxes` object

Axes used to plot the map, returned as either an `Axes` or `UIAxes` object. See `axes` or `uiaxes`.

## **See Also**

`optimizePoseGraph` | `robotics.PoseGraph`

## **Topics**

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

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

**Introduced in R2018a**

## checkOccupancy

**Class:** robotics.OccupancyGrid

**Package:** robotics

Check locations for free, occupied, or unknown values

### Syntax

```
i0ccval = checkOccupancy(map, xy)
i0ccval = checkOccupancy(map, ij, "grid")
```

### Description

`i0ccval = checkOccupancy(map, xy)` returns an array of occupancy values at the `xy` locations using the `OccupiedThreshold` and `FreeThreshold` properties of the `map` object. Each row is a separate `xy` location in the grid to check the occupancy of. Occupancy values can be obstacle free (0), occupied (1), or unknown (-1).

`i0ccval = checkOccupancy(map, ij, "grid")` specifies `ij` grid cell indices instead of `xy` locations.

### Input Arguments

#### **map — Map representation**

OccupancyGrid object

Map representation, specified as a `robotics.OccupancyGrid` object. This object represents the environment of the robot. The object contains a matrix grid with values representing the probability of the occupancy of that cell. Values close to 1 represent a high certainty that the cell contains an obstacle. Values close to 0 represent certainty that the cell is not occupied and obstacle free.

#### **xy — World coordinates**

*n*-by-2 matrix

World coordinates, specified as an  $n$ -by-2 matrix of  $[x \ y]$  pairs, where  $n$  is the number of world coordinates.

Data Types: double

### **ij** — Grid positions

$n$ -by-2 vertical array

Grid positions, specified as an  $n$ -by-2 matrix of  $[i \ j]$  pairs in  $[rows \ cols]$  format, where  $n$  is the number of grid positions.

Data Types: double

## Output Arguments

### **iOccval** — Interpreted occupancy values

$n$ -by-1 column vector

Interpreted occupancy values, returned as an  $n$ -by-1 column vector equal in length to  $xy$  or  $ij$ .

Occupancy values can be obstacle free (0), occupied (1), or unknown (-1). These values are determined from the actual probability values and the `OccupiedThreshold` and `FreeThreshold` properties of the map object.

## Examples

### Get Occupancy Values and Check Occupancy Status

Access occupancy values and check their occupancy status based on the occupied and free thresholds of the `OccupancyGrid` object.

Create a matrix and populate it with values. Use this matrix to create an occupancy grid.

```
p = 0.5*ones(20,20);  
p(11:20,11:20) = 0.75*ones(10,10);  
map = robotics.OccupancyGrid(p,10);
```

Get the occupancy of different locations and check their occupancy statuses. The occupancy status returns 0 for free space and 1 for occupied space. Unknown values return -1.

```
pocc = getOccupancy(map, [1.5 1])
```

```
pocc = 0.7500
```

```
occupied = checkOccupancy(map, [1.5 1])
```

```
occupied = 1
```

```
pocc2 = getOccupancy(map, [5 5], 'grid')
```

```
pocc2 = 0.5000
```

```
occupied2 = checkOccupancy(map, [5 5], 'grid')
```

```
occupied2 = -1
```

### See Also

[robotics.BinaryOccupancyGrid](#) | [robotics.OccupancyGrid](#) | [robotics.OccupancyGrid.getOccupancy](#)

**Introduced in R2016b**



## copy

**Class:** robotics.OccupancyGrid

**Package:** robotics

Create copy of occupancy grid

## Syntax

```
copyMap = copy(map)
```

## Description

`copyMap = copy(map)` creates a deep copy of the `OccupancyGrid` object with the same properties.

## Input Arguments

**map** — Map representation

`OccupancyGrid` object

Map representation, specified as a `robotics.OccupancyGrid` object. This object represents the environment of the robot. The object contains a matrix grid with values representing the probability of the occupancy of that cell. Values close to 1 represent a high certainty that the cell contains an obstacle. Values close to 0 represent certainty that the cell is not occupied and obstacle free.

## Output Arguments

**copyMap** — Copied map representation

`OccupancyGrid` object

Map representation, specified as a `robotics.OccupancyGrid` object. The properties are the same as the input object, `map`, but they have a different object handle.

## Examples

### Copy Occupancy Grid Map

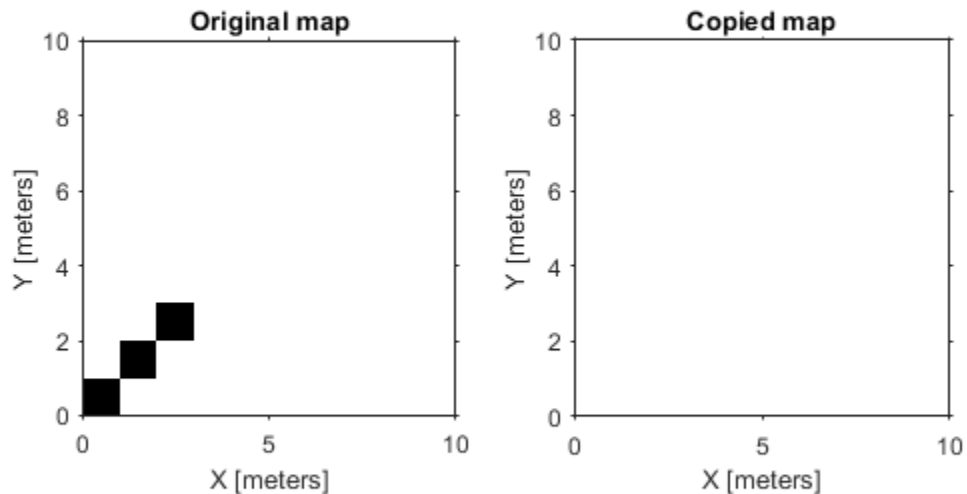
Copy an occupancy grid object. Once copied, the original object can be modified without affecting the copied map.

Create an occupancy grid with zeros for an empty map.

```
p = zeros(10);  
map = robotics.OccupancyGrid(p);
```

Copy the occupancy grid map. Modify the original map. The copied map is not modified. Plot the two maps side by side.

```
mapCopy = copy(map);  
setOccupancy(map, [1:3;1:3]', ones(3,1));  
subplot(1,2,1)  
show(map)  
title('Original map')  
subplot(1,2,2)  
show(mapCopy)  
title('Copied map')
```



## See Also

`occupancyMatrix` | `robotics.BinaryOccupancyGrid` | `robotics.OccupancyGrid`  
| `robotics.OccupancyGrid.getOccupancy`

## Topics

"Occupancy Grids"

Introduced in R2016b

# getOccupancy

**Class:** robotics.OccupancyGrid

**Package:** robotics

Get occupancy of a location

## Syntax

```
occval = getOccupancy(map, xy)
occval = getOccupancy(map, ij, "grid")
```

## Description

`occval = getOccupancy(map, xy)` returns an array of probability occupancy values at the `xy` locations. Values close to 1 represent a high certainty that the cell contains an obstacle. Values close to 0 represent certainty that the cell is not occupied and obstacle free.

`occval = getOccupancy(map, ij, "grid")` specifies `ij` grid cell indices instead of `xy` locations.

## Input Arguments

### **map — Map representation**

OccupancyGrid object

Map representation, specified as a `robotics.OccupancyGrid` object. This object represents the environment of the robot. The object contains a matrix grid with values representing the probability of the occupancy of that cell. Values close to 1 represent a high certainty that the cell contains an obstacle. Values close to 0 represent certainty that the cell is not occupied and obstacle free.

### **xy — World coordinates**

*n*-by-2 matrix

World coordinates, specified as an  $n$ -by-2 matrix of [x y] pairs, where  $n$  is the number of world coordinates.

Data Types: double

### **ij** — Grid positions

$n$ -by-2 matrix

Grid positions, specified as an  $n$ -by-2 matrix of [i j] pairs in [rows cols] format, where  $n$  is the number of grid positions.

Data Types: double

## Output Arguments

### **occval** — Probability occupancy values

column vector

Probability occupancy values, returned as a column vector the same length as either xy or ij.

Values close to 0 represent certainty that the cell is not occupied and obstacle free.

## Examples

### **Get Occupancy Values and Check Occupancy Status**

Access occupancy values and check their occupancy status based on the occupied and free thresholds of the OccupancyGrid object.

Create a matrix and populate it with values. Use this matrix to create an occupancy grid.

```
p = 0.5*ones(20,20);  
p(11:20,11:20) = 0.75*ones(10,10);  
map = robotics.OccupancyGrid(p,10);
```

Get the occupancy of different locations and check their occupancy statuses. The occupancy status returns 0 for free space and 1 for occupied space. Unknown values return -1.

```
pocc = getOccupancy(map,[1.5 1])  
pocc = 0.7500  
occupied = checkOccupancy(map,[1.5 1])  
occupied = 1  
  
pocc2 = getOccupancy(map,[5 5], 'grid')  
pocc2 = 0.5000  
occupied2 = checkOccupancy(map,[5 5], 'grid')  
occupied2 = -1
```

#### **Insert Laser Scans Into Occupancy Grid**

Take range and angle readings from a laser scan and insert these readings into an occupancy grid.

Create an empty occupancy grid map.

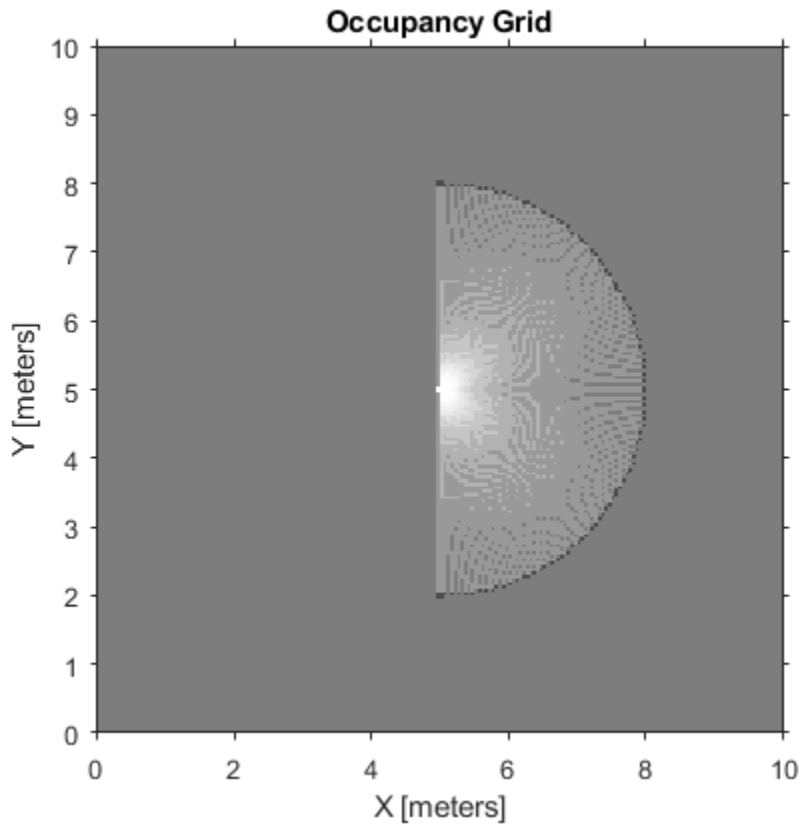
```
map = robotics.OccupancyGrid(10,10,20);
```

Insert a laser scan into the occupancy grid. Specify the pose of the robot ranges and angles and the max range of the laser scan.

```
pose = [5,5,0];  
ranges = 3*ones(100, 1);  
angles = linspace(-pi/2, pi/2, 100);  
maxrange = 20;  
  
insertRay(map,pose,ranges,angles,maxrange);
```

Show the map to see the results of inserting the laser scan. Check the occupancy of the spot directly in front of the robot.

```
show(map)
```

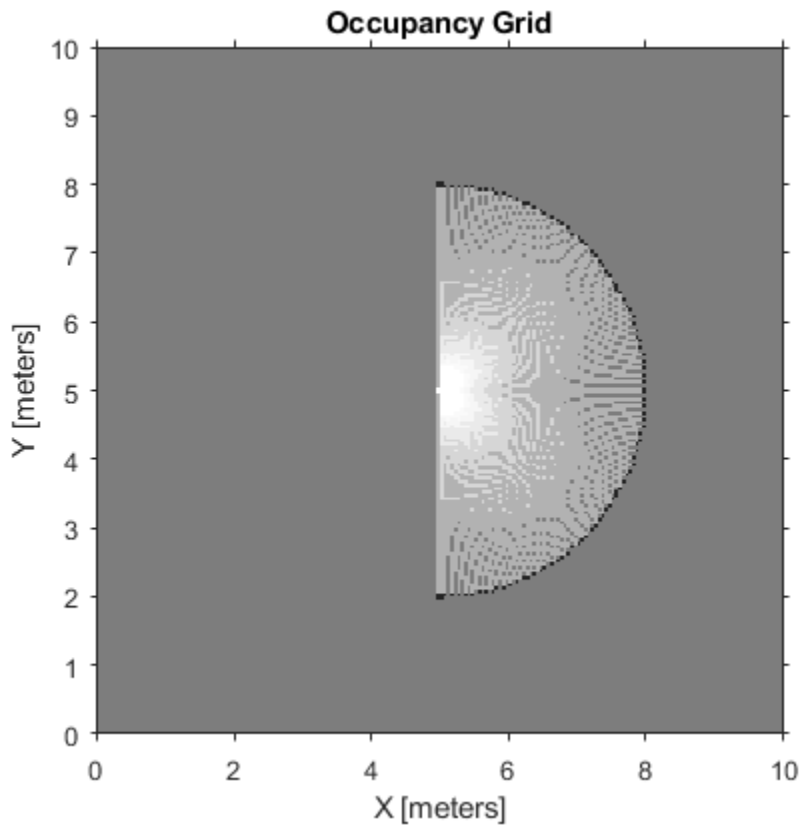


```
getOccupancy(map, [8 5])
```

```
ans = 0.7000
```

Add a second reading and view the update to the occupancy values. The additional reading increases the confidence in the readings. The free and occupied values become more distinct.

```
insertRay(map, pose, ranges, angles, maxrange);  
show(map)
```



```
getOccupancy(map, [8 5])
```

```
ans = 0.8448
```

## Limitations

Occupancy values have a limited resolution of  $\pm 0.001$ . The values are stored as `int16` using a log-odds representation. This data type limits resolution, but saves you memory when storing large maps in MATLAB. When calling `set` and then `get`, the value returned might not equal the value you set. For more information, see the log-odds representations section in “Occupancy Grids”.



## See Also

`robotics.BinaryOccupancyGrid` | `robotics.OccupancyGrid` |  
`robotics.OccupancyGrid.checkOccupancy`

## Topics

“Occupancy Grids”

**Introduced in R2016b**

## grid2world

**Class:** robotics.OccupancyGrid

**Package:** robotics

Convert grid indices to world coordinates

### Syntax

```
xy = grid2world(map,ij)
```

### Description

`xy = grid2world(map,ij)` converts a `[row col]` array of grid indices, `ij`, to an array of world coordinates, `xy`.

### Input Arguments

#### **map — Map representation**

*OccupancyGrid* object

Map representation, specified as a `robotics.OccupancyGrid` object. This object represents the environment of the robot. The object contains a matrix grid with values representing the probability of the occupancy of that cell. Values close to 1 represent a high certainty that the cell contains an obstacle. Values close to 0 represent certainty that the cell is not occupied and obstacle free.

#### **ij — Grid positions**

*n-by-2* matrix

Grid positions, specified as an *n*-by-2 matrix of `[i j]` pairs in `[rows cols]` format, where *n* is the number of grid positions. The grid cell locations are counted from the top left corner of the grid.

Data Types: double

## Output Arguments

### **xy — World coordinates**

*n*-by-2 matrix

World coordinates, returned as an *n*-by-2 matrix of [x y] pairs, where *n* is the number of world coordinates.

Data Types: double

## See Also

`robotics.BinaryOccupancyGrid` | `robotics.OccupancyGrid` |  
`robotics.OccupancyGrid.world2grid`

## Topics

“Occupancy Grids”

**Introduced in R2016b**

# inflate

**Class:** `robotics.OccupancyGrid`

**Package:** `robotics`

Inflate each occupied grid location

## Syntax

```
inflate(map, radius)
inflate(map, gridradius, 'grid')
```

## Description

`inflate(map, radius)` inflates each occupied position of the specified `map` by the `radius` specified in meters. `radius` is rounded up to the nearest equivalent cell based on the resolution of the map. Values are modified using *grayscale inflation* to inflate higher probability values across the grid. This inflation increases the size of the occupied locations in the map.

`inflate(map, gridradius, 'grid')` inflates each occupied position by the `gridradius` in number of cells.

## Input Arguments

### **map — Map representation**

`OccupancyGrid` object

Map representation, specified as a `robotics.OccupancyGrid` object. This object represents the environment of the robot. The object contains a matrix grid with values representing the probability of the occupancy of that cell. Values close to 1 represent a high certainty that the cell contains an obstacle. Values close to 0 represent certainty that the cell is not occupied and obstacle free.

### **radius — Dimension that defines by how much to inflate occupied locations**

scalar

Dimension that defines by how much to inflate occupied locations, specified as a scalar. radius is rounded up to the nearest cell value.

Data Types: double

### **gridradius — Number of cells by which to inflate the occupied locations**

positive scalar

Number of cells by which to inflate the occupied locations, specified as a positive scalar.

Data Types: double

## **Examples**

### **Create and Modify Occupancy Grid**

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

```
map = robotics.OccupancyGrid(10,10,10);
```

Update the occupancy of world locations with specific probability values.

```
map = robotics.OccupancyGrid(10,10,10);
```

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

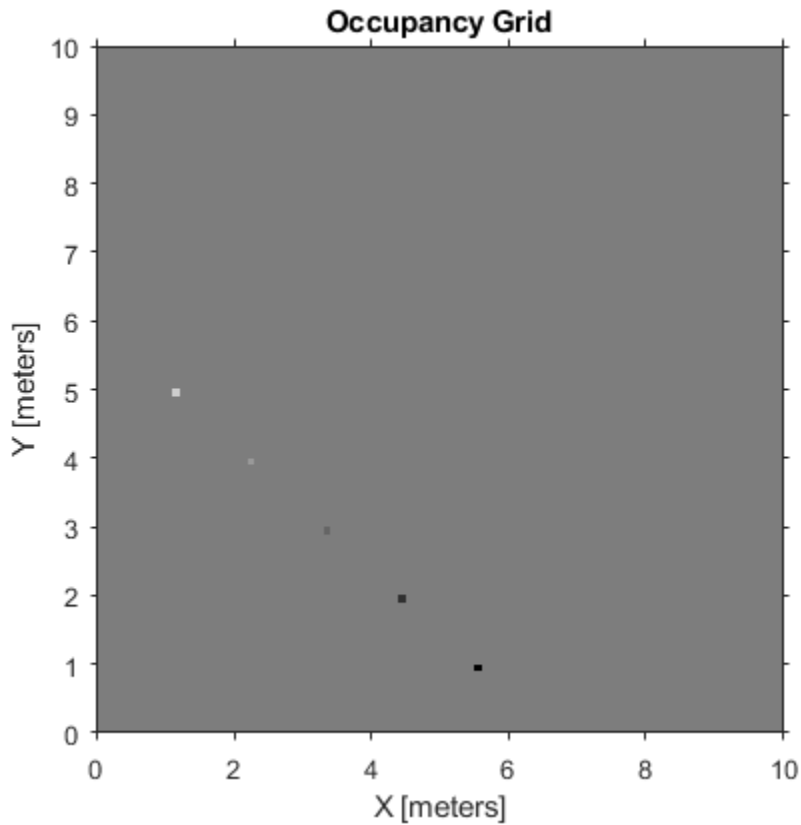
```
y = [5.0; 4.0; 3.0; 2.0; 1.0];
```

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

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

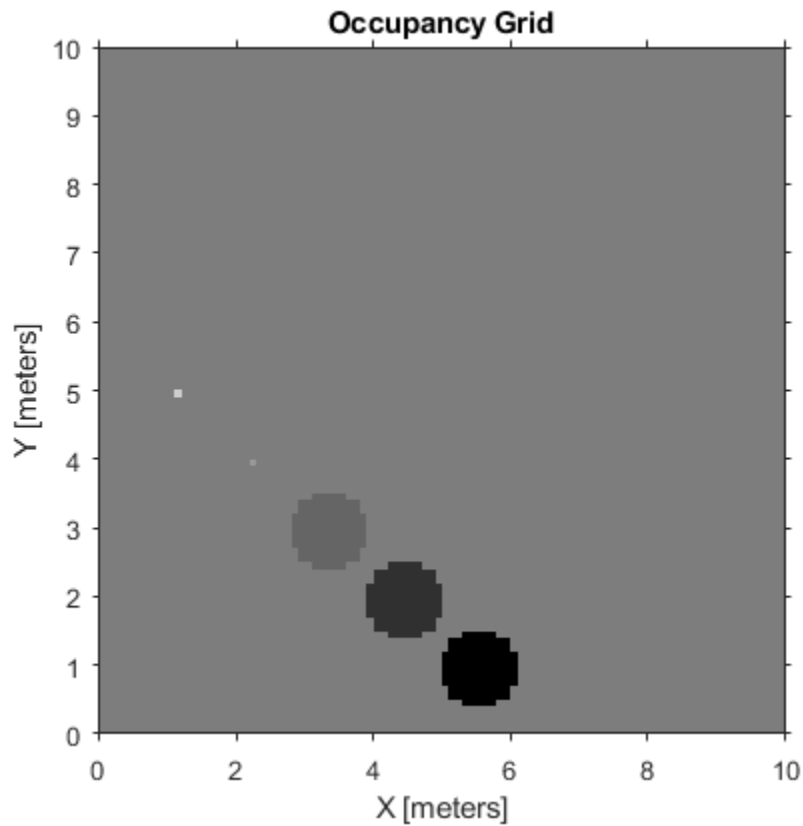
```
figure
```

```
show(map)
```



Inflate occupied areas by a given radius. Larger occupancy values overwrite the smaller values.

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



Get grid locations from world locations.

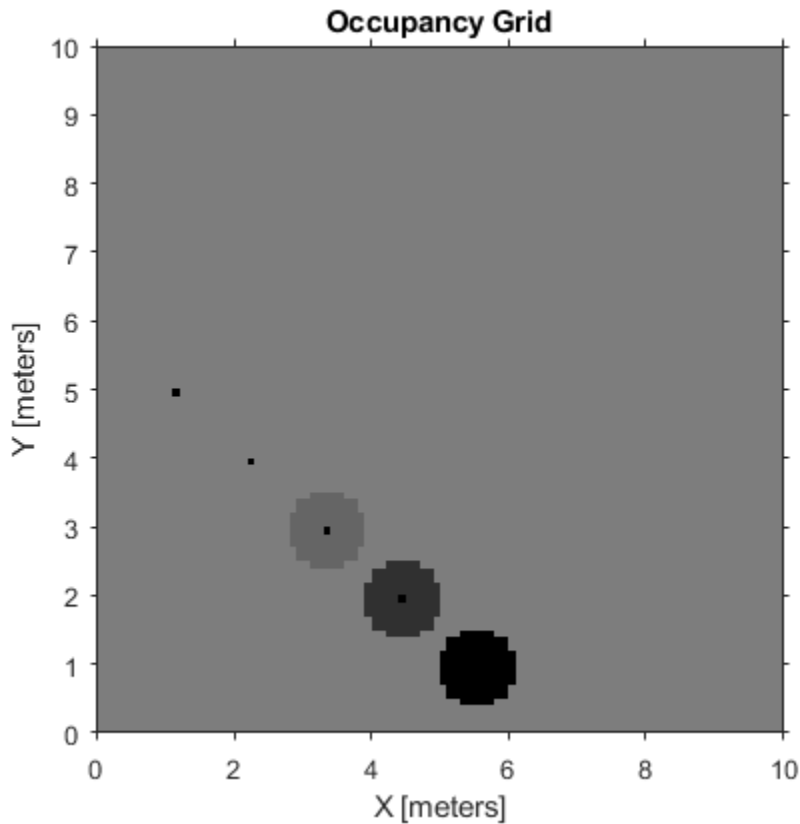
```
ij = world2grid(map,[x y]);
```

Set grid locations to occupied locations.

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

```
figure
```

```
show(map)
```



## Definitions

### Grayscale Inflation

In *grayscale inflation*, the `strel` function creates a circular structuring element using the inflation radius. The grayscale inflation of  $A(x, y)$  by  $B(x, y)$  is defined as:

$$(A \oplus B)(x, y) = \max \{A(x - x', y - y') + B(x', y') \mid (x', y') \in D_B\}.$$

$D_B$  is the domain of the probability values in the structuring element  $B$ .  $A(x, y)$  is assumed to be  $+\infty$  outside the domain of the grid.



Grayscale inflation acts as a local maximum operator and finds the highest probability values for nearby cells. The `inflate` method uses this definition to inflate the higher probability values throughout the grid. This inflation increases the size of any occupied locations and creates a buffer zone for robots to use as they navigate.

## See Also

`OccupancyGrid` | `robotics.BinaryOccupancyGrid` | `robotics.OccupancyGrid` | `robotics.OccupancyGrid.getOccupancy`

## Topics

“Occupancy Grids”

**Introduced in R2016b**

## insertRay

**Class:** robotics.OccupancyGrid

**Package:** robotics

Insert ray from laser scan observation

### Syntax

```
insertRay(map, pose, scan, maxrange)
insertRay(map, pose, ranges, angles, maxrange)
insertRay(map, startpt, endpoints)
insertRay( ____, invModel)
```

### Description

`insertRay(map, pose, scan, maxrange)` inserts one or more lidar scan sensor observations in the occupancy grid, `map`, using the input `lidarScan` object, `scan`, to get ray endpoints. The ray endpoints are considered free space if the input scan ranges are below `maxrange`. Cells observed as occupied are updated with an observation of 0.7. All other points along the ray are treated as obstacle free and updated with an observation of 0.4. Endpoints above `maxrange` are not updated. NaN values are ignored. This behavior correlates to the inverse sensor model.

`insertRay(map, pose, ranges, angles, maxrange)` specifies the range readings as vectors, `ranges` and `angles`.

`insertRay(map, startpt, endpoints)` inserts observations between the line segments from the start point to the end points. The endpoints are updated with a probability observation of 0.7. Cells along the line segments are updated with an observation of 0.4.

`insertRay( ____, invModel)` inserts rays with updated probabilities given in the two-element vector, `invModel`, that corresponds to obstacle-free and occupied observations. Use any of the previous syntaxes to input the rays.

## Input Arguments

### **map** — Map representation

OccupancyGrid object

Map representation, specified as a `robotics.OccupancyGrid` object. This object represents the environment of the robot. The object contains a matrix grid with values representing the probability of the occupancy of that cell. Values close to 1 represent a high certainty that the cell contains an obstacle. Values close to 0 represent certainty that the cell is not occupied and obstacle free.

### **pose** — Position and orientation of robot

[x y theta] vector

Position and orientation of robot, specified as an [x y theta] vector. The robot pose is an x and y position with angular orientation (in radians) measured from the x-axis.

### **scan** — Lidar scan readings

lidarScan object

Lidar scan readings, specified as a `lidarScan` object.

### **ranges** — Range values from scan data

vector of scalars

Range values from scan data, specified as a vector of scalars in meters. These range values are distances from a sensor at given angles. The vector must be the same length as the corresponding angles vector.

### **angles** — Angle values from scan data

vector of scalars

Angle values from scan data, specified as a vector of scalars in radians. These angle values are the specific angles of the given ranges. The vector must be the same length as the corresponding ranges vector.

### **maxrange** — Maximum range of sensor

scalar

Maximum range of laser range sensor, specified as a scalar. Range values greater than or equal to `maxrange` are considered free along the whole length of the ray, up to `maxrange`.

### **startpt — Start point for rays**

two-element vector

Start point for rays, specified as a two-element vector,  $[x \ y]$ , in the world coordinate frame. All rays are line segments that originate at this point.

### **endpoints — Endpoints for rays**

$n$ -by-2 matrix

Endpoints for rays, specified as an  $n$ -by-2 matrix,  $[x \ y]$ , in the world coordinate frame, where  $n$  is the length of ranges or angles. All rays are line segments that originate at `startpt`.

### **invModel — Inverse sensor model values**

two-element vector

Inverse sensor model values, specified as a two-element vector corresponding to obstacle-free and occupied probabilities. Points along the ray are updated according to the inverse sensor model and the specified range readings. NaN range values are ignored. Range values greater than `maxrange` are not updated. See “Inverse Sensor Model” on page 3-73.

## Examples

### **Insert Laser Scans Into Occupancy Grid**

Take range and angle readings from a laser scan and insert these readings into an occupancy grid.

Create an empty occupancy grid map.

```
map = robotics.OccupancyGrid(10,10,20);
```

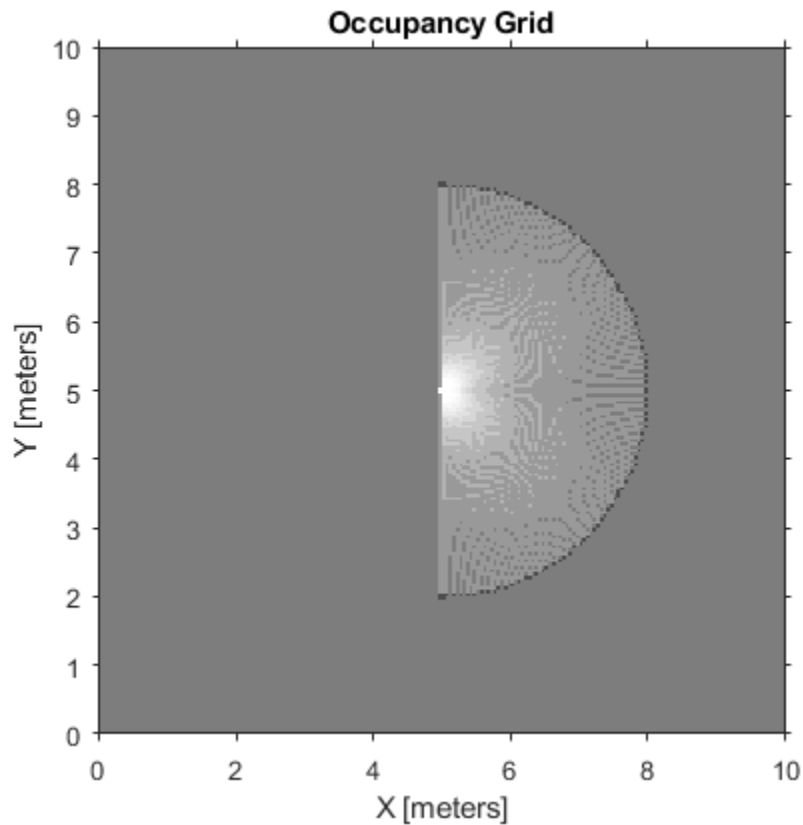
Insert a laser scan into the occupancy grid. Specify the pose of the robot ranges and angles and the max range of the laser scan.

```
pose = [5,5,0];  
ranges = 3*ones(100, 1);  
angles = linspace(-pi/2, pi/2, 100);  
maxrange = 20;
```

```
insertRay(map, pose, ranges, angles, maxrange);
```

Show the map to see the results of inserting the laser scan. Check the occupancy of the spot directly in front of the robot.

```
show(map)
```

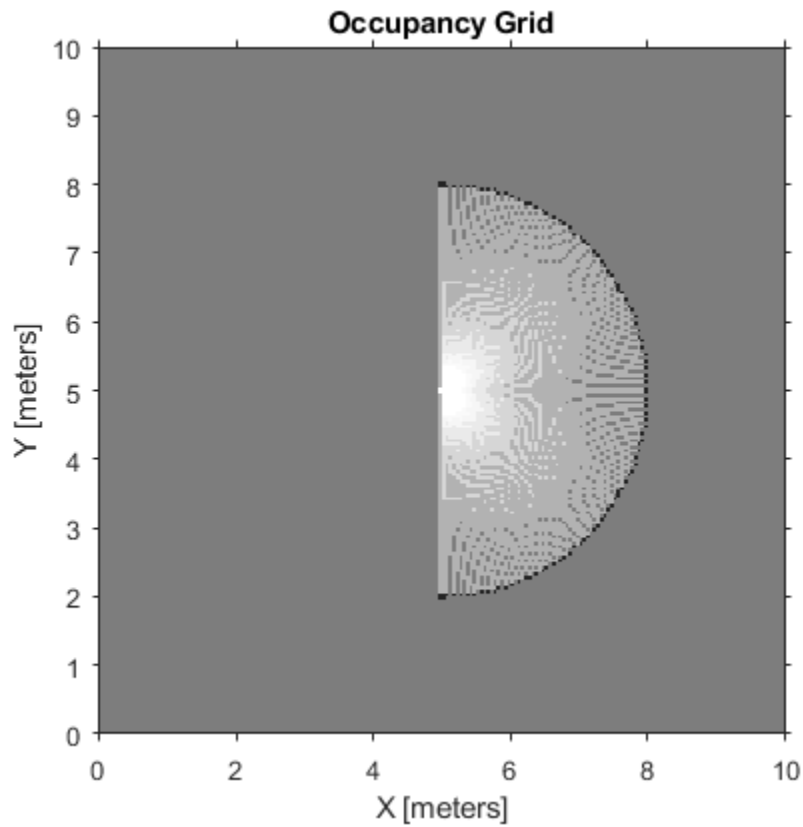


```
getOccupancy(map, [8 5])
```

```
ans = 0.7000
```

Add a second reading and view the update to the occupancy values. The additional reading increases the confidence in the readings. The free and occupied values become more distinct.

```
insertRay(map,pose,ranges,angles,maxrange);  
show(map)
```



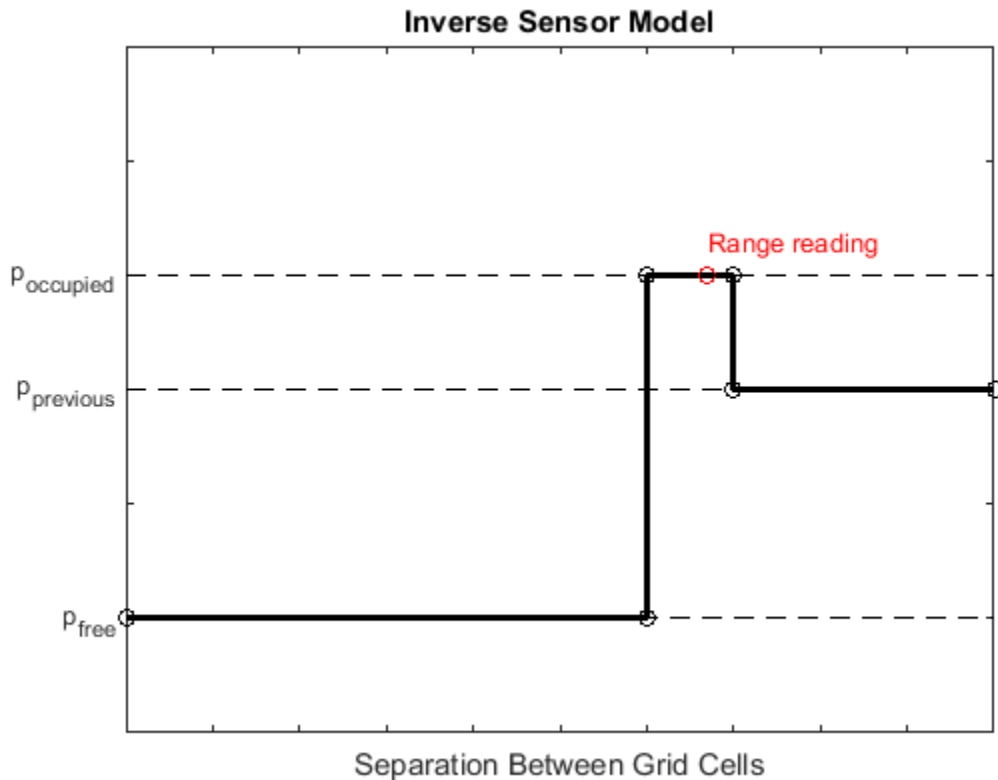
```
getOccupancy(map,[8 5])
```

```
ans = 0.8448
```

## Definitions

### Inverse Sensor Model

The inverse sensor model determines how values are set along a ray from a range sensor reading to the obstacles in the map. You can customize this model by specifying different probabilities for free and occupied locations in the `invModel` argument. NaN range values are ignored. Range values greater than `maxrange` are not updated.



Grid locations that contain range readings are updated with the occupied probability. Locations before the reading are updated with the free probability. All locations after the reading are not updated.

## See Also

`lidarScan` | `robotics.BinaryOccupancyGrid` | `robotics.OccupancyGrid` |  
`robotics.OccupancyGrid.raycast`

## Topics

“Occupancy Grids”

**Introduced in R2016b**



# occupancyMatrix

**Class:** robotics.OccupancyGrid

**Package:** robotics

Convert occupancy grid to double matrix

## Syntax

```
mat = occupancyMatrix(map)
mat = occupancyMatrix(map, "ternary")
```

## Description

`mat = occupancyMatrix(map)` returns probability values stored in the occupancy grid object as a matrix.

`mat = occupancyMatrix(map, "ternary")` returns the occupancy status of each grid cell as a matrix. The `OccupiedThreshold` and `FreeThreshold` properties on the occupancy grid determine the obstacle free cells (0) and occupied cells (1). Unknown values are returned as -1.

## Input Arguments

### **map** — Map representation

OccupancyGrid object

Map representation, specified as a `robotics.OccupancyGrid` object. This object represents the environment of the robot. The object contains a matrix grid with values representing the probability of the occupancy of that cell. Values close to 1 represent a high certainty that the cell contains an obstacle. Values close to 0 represent certainty that the cell is not occupied and obstacle free.

## Output Arguments

### **mat** — Occupancy grid values

matrix

Occupancy grid values, returned as an  $h$ -by- $w$  matrix, where  $h$  and  $w$  are defined by the two elements of the `GridSize` property of the occupancy grid object.

## See Also

`robotics.BinaryOccupancyGrid` | `robotics.OccupancyGrid` |  
`robotics.OccupancyGrid.getOccupancy` | `robotics.OccupancyGrid.show`

## Topics

“Occupancy Grids”

**Introduced in R2016b**

# raycast

**Class:** robotics.OccupancyGrid

**Package:** robotics

Compute cell indices along a ray

## Syntax

```
[endpoints,midpoints] = raycast(map,pose,range,angle)
[endpoints,midpoints] = raycast(map,p1,p2)
```

## Description

`[endpoints,midpoints] = raycast(map,pose,range,angle)` returns cell indices of the specified map for all cells traversed by a ray originating from the specified `pose` at the specified `angle` and `range` values. `endpoints` contains all indices touched by the end of the ray, with all other points included in `midpoints`.

`[endpoints,midpoints] = raycast(map,p1,p2)` returns the cell indices of the line segment between the two specified points.

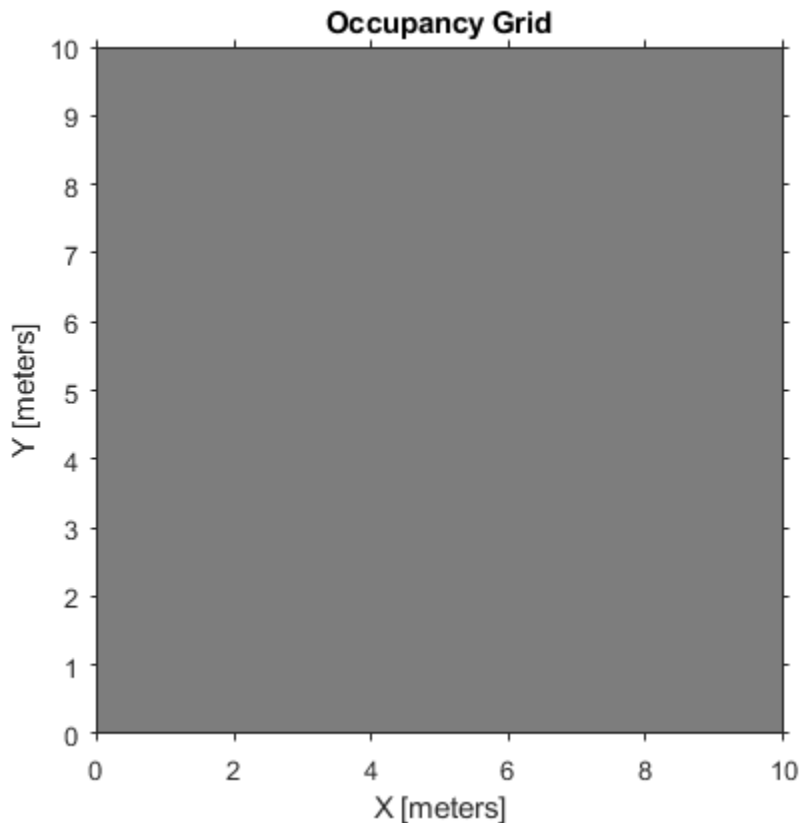
## Examples

### Get Grid Cells Along A Ray

Use the `raycast` method to generate cell indices for all cells traversed by a ray.

Create an empty map. A low resolution map is used to illustrate the affect of grid locations.

```
map = robotics.OccupancyGrid(10,10,1);
show(map)
```

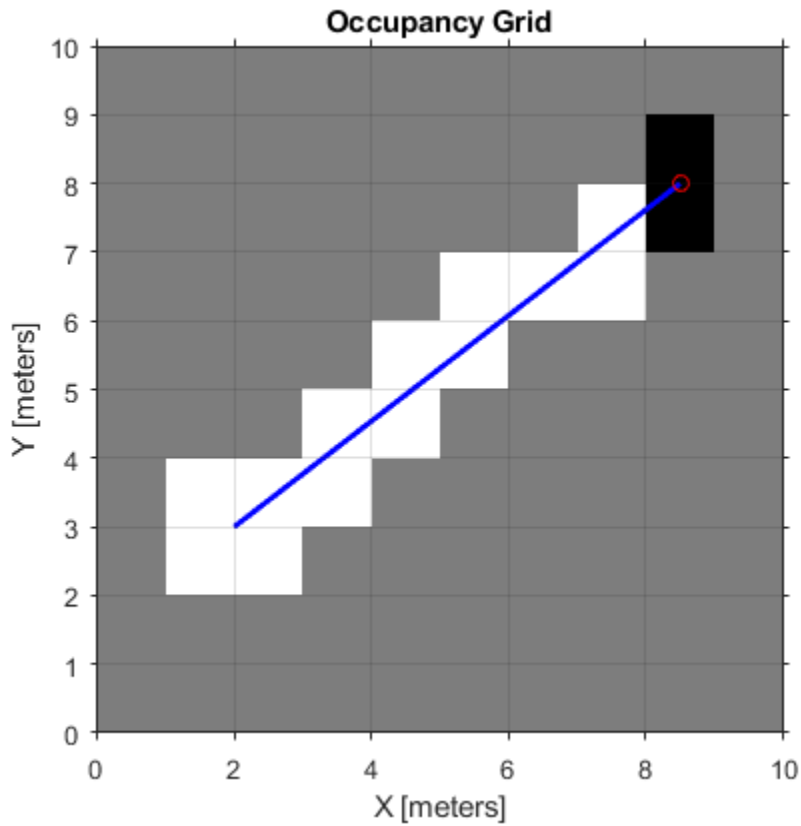


Get the grid indices of the midpoints and end points of a ray from  $p_1$  to  $p_2$ . Set occupancy values for these grid indices. Midpoints are treated as open space. Endpoints are updated with an occupied observation.

```
p1 = [2 3];  
p2 = [8.5 8];  
[endPts,midPts] = raycast(map,p1,p2);  
setOccupancy(map,midPts,zeros(length(midPts),1),'grid');  
setOccupancy(map,endPts,ones(length(endPts),1),'grid');
```

Plot the original ray over the map. Notice that each grid cell touched by the line is updated. The starting point overlaps multiple cells and the line touches the edge of certain cells, but all the cells are still updated.

```
show(map)
hold on
plot([p1(1) p2(1)], [p1(2) p2(2)], '-b', 'LineWidth', 2)
plot(p2(1), p2(2), 'or')
grid on
```



## Input Arguments

**map** — Map representation

OccupancyGrid object

Map representation, specified as a `robotics.OccupancyGrid` object. This object represents the environment of the robot. The object contains a matrix grid with values representing the probability of the occupancy of that cell. Values close to 1 represent a high certainty that the cell contains an obstacle. Values close to 0 represent certainty that the cell is not occupied and obstacle free.

**pose — Position and orientation of robot**

[x y theta] vector

Position and orientation of robot, specified as an [x y theta] vector. The robot pose is an x and y position with angular orientation (in radians) measured from the x-axis.

**range — Range value**

scalar

Range value, specified as a scalar in meters.

**angle — Angle value**

scalar

Angle value, specified as a scalar in radians. The angle value is the specific angle orientation of the given range.

**p1 — Starting point of ray**

[x y] two-element vector

Starting point of ray, specified as an [x y] two-element vector. The point is defined in the robot coordinate frame.

**p2 — Endpoint of ray**

[x y] two-element vector

Endpoint of ray, specified as an [x y] two-element vector. The point is defined in the robot coordinate frame.

## Output Arguments

**endpoints — Endpoint grid indices**

[i j] matrix

Endpoint indices, returned as an [i j] matrix. The endpoints are where the range value hits at the specified angle. Multiple indices are only given if the point intersect grid locations.

**midpoints — Midpoint grid indices**

[i j] matrix

Midpoint indices, returned as an [i j] matrix. This argument includes all grid indices the ray intersects, excluding the endpoint.

**See Also**

`robotics.BinaryOccupancyGrid` | `robotics.OccupancyGrid` | `robotics.OccupancyGrid.insertRay`

**Topics**

“Occupancy Grids”

**Introduced in R2016b**

## rayIntersection

**Class:** robotics.OccupancyGrid

**Package:** robotics

Compute map intersection points of rays

### Syntax

```
intersectionPts = rayIntersection(map,pose,angles,maxrange)
intersectionPts = rayIntersection(map,pose,angles,maxrange,
threshold)
```

### Description

`intersectionPts = rayIntersection(map,pose,angles,maxrange)` returns intersection points in the world coordinate frame of the specified `map` for rays emanating from the specified `pose` with the specified `angles`. If there is no intersection up to the specified `maxrange`, `[NaN NaN]` is returned. By default, the `OccupiedThreshold` property is used to determine occupied cells.

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

### Examples

#### Get Ray Intersection Points on Occupancy Grid

Get the ray intersection points on an occupancy grid that has obstacles in the map. The rays are defined ranges and angles from a starting robot pose.

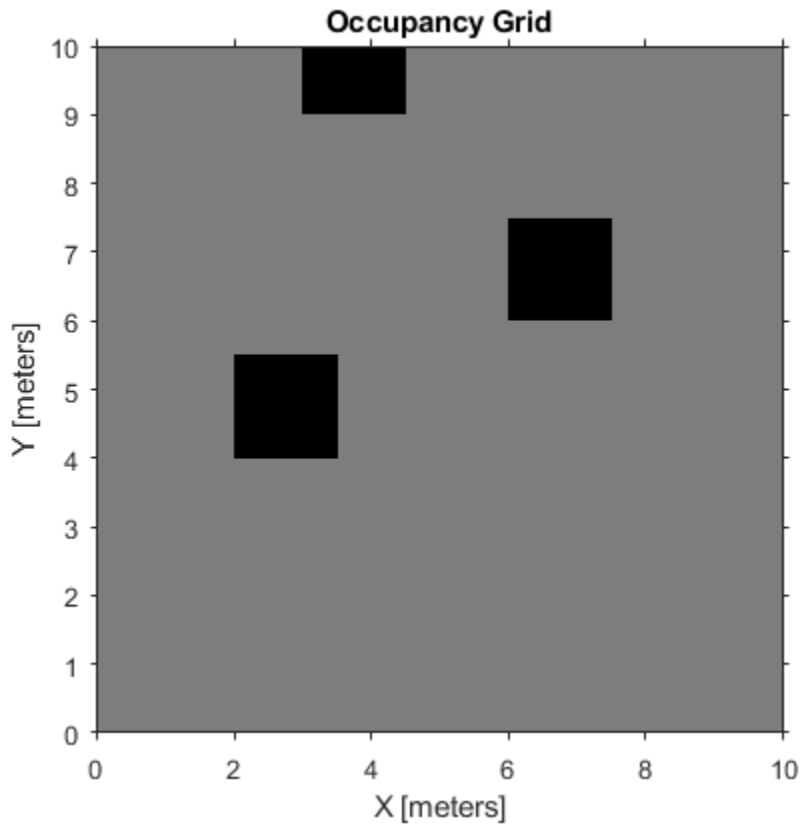
Create an occupancy grid map. Add obstacles and inflate them. A lower resolution map is used to illustrate the importance of using grid cells. Show the map.



```

map = robotics.OccupancyGrid(10,10,2);
obstacles = [4 10; 3 5; 7 7];
setOccupancy(map,obstacles,ones(length(obstacles),1))
inflate(map,0.25)
show(map)

```



Find the intersection point for rays that emit from the given robot pose. The max range and angles for these rays are specified. The last ray does not intersect with an obstacle within the max range, thus it has no collision point.

```

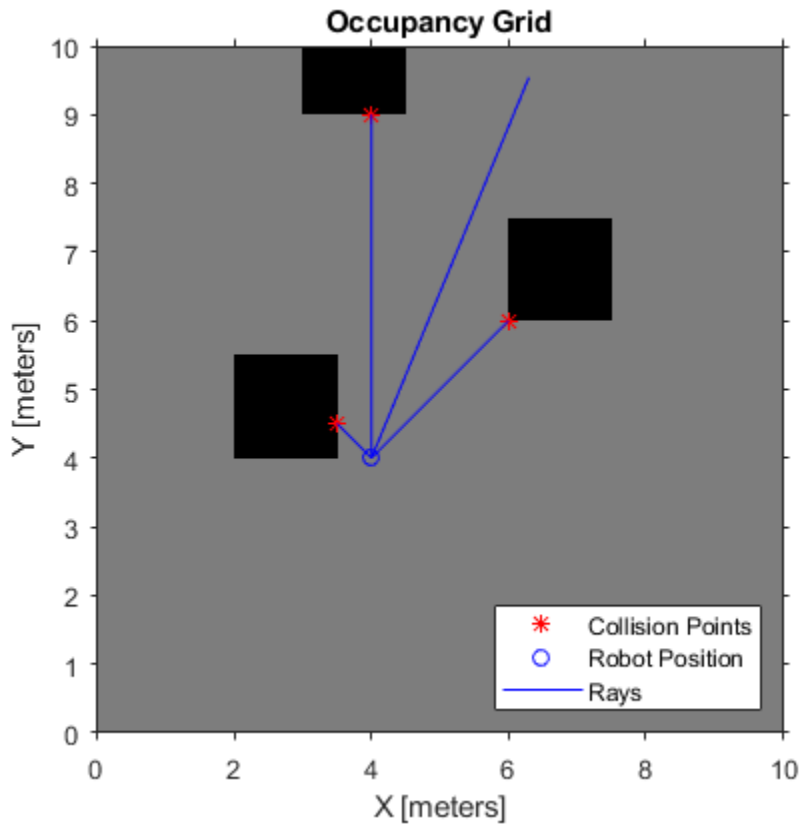
maxrange = 6;
angles = [pi/4,-pi/4,0,-pi/8];
robotPose = [4,4,pi/2];
intsectionPts = rayIntersection(map,robotPose,angles,maxrange,0.7)

```

```
intsectionPts = 4x2
    3.5000    4.5000
    6.0000    6.0000
    4.0000    9.0000
    NaN      NaN
```

Plot the intersection points and rays from the pose.

```
hold on
plot(intsectionPts(:,1),intsectionPts(:,2) , '*r') % Intersection points
plot(robotPose(1),robotPose(2), 'ob') % Robot pose
for i = 1:3
    plot([robotPose(1),intsectionPts(i,1)],...
         [robotPose(2),intsectionPts(i,2)], '-b') % Plot intersecting rays
end
plot([robotPose(1),robotPose(1)-6*sin(angles(4))],...
     [robotPose(2),robotPose(2)+6*cos(angles(4))], '-b') % No intersection ray
legend('Collision Points', 'Robot Position', 'Rays', 'Location', 'SouthEast')
```



## Input Arguments

### **map** — Map representation

`OccupancyGrid` object

Map representation, specified as a `robotics.OccupancyGrid` object. This object represents the environment of the robot. The object contains a matrix grid with values representing the probability of the occupancy of that cell. Values close to 1 represent a high certainty that the cell contains an obstacle. Values close to 0 represent certainty that the cell is not occupied and obstacle free.

### **pose — Position and orientation of robot**

[x y theta] vector

Position and orientation of robot, specified as an [x y theta] vector. The robot pose is an x and y position with angular orientation (in radians) measured from the x-axis.

### **angles — Ray angles emanating from the robot**

vector in radians

Ray angles emanating from the robot, specified as a vector in radians. These angles are relative to the specified robot pose.

### **maxrange — Maximum range of sensor**

scalar

Maximum range of laser range sensor, specified as a scalar. Range values greater than or equal to maxrange are considered free along the whole length of the ray, up to maxrange.

### **threshold — Threshold for occupied cells**

scalar from 0 to 1

Threshold for occupied cells, specified as a scalar from 0 to 1. Occupancy values greater than or equal to the threshold are treated as occupied cells to trigger intersections.

## Output Arguments

### **intersectionPts — Intersection points**

*n*-by-2 matrix

Intersection points, returned as *n*-by-2 matrix, [x y], in the world coordinate frame, where *n* is the length of angles.

## See Also

robotics.BinaryOccupancyGrid | robotics.OccupancyGrid |  
robotics.OccupancyGrid.raycast |  
robotics.OccupancyGrid.updateOccupancy

## **Topics**

“Occupancy Grids”

**Introduced in R2016b**

# setOccupancy

**Class:** robotics.OccupancyGrid

**Package:** robotics

Set occupancy of a location

## Syntax

```
setOccupancy(map,xy,occval)  
setOccupancy(map,ij,occval,"grid")
```

## Description

`setOccupancy(map,xy,occval)` assigns the occupancy values to each coordinate specified in `xy`. `occval` can be an array the length of `xy` or a scalar, which is applied to all coordinates.

`setOccupancy(map,ij,occval,"grid")` assigns occupancy values to the specified grid locations, `ij`, instead of to world coordinates.

## Input Arguments

### **map — Map representation**

OccupancyGrid object

Map representation, specified as a `robotics.OccupancyGrid` object. This object represents the environment of the robot. The object contains a matrix grid with values representing the probability of the occupancy of that cell. Values close to 1 represent a high certainty that the cell contains an obstacle. Values close to 0 represent certainty that the cell is not occupied and obstacle free.

### **xy — World coordinates**

*n*-by-2 matrix

World coordinates, specified as an  $n$ -by-2 matrix of  $[x \ y]$  pairs, where  $n$  is the number of world coordinates.

Data Types: double

### **ij** — Grid positions

$n$ -by-2 vertical array

Grid positions, specified as an  $n$ -by-2 matrix of  $[i \ j]$  pairs in  $[rows \ cols]$  format, where  $n$  is the number of grid positions.

Data Types: double

### **occval** — Probability occupancy values

scalar | column vector

Probability occupancy values, specified as a scalar or a column vector the same size as either  $xy$  or  $ij$ . A scalar input is applied to all coordinates in either  $xy$  or  $ij$ .

Values close to 0 represent certainty that the cell is not occupied and obstacle free.

## Examples

### Create and Modify Occupancy Grid

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

```
map = robotics.OccupancyGrid(10,10,10);
```

Update the occupancy of world locations with specific probability values.

```
map = robotics.OccupancyGrid(10,10,10);
```

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

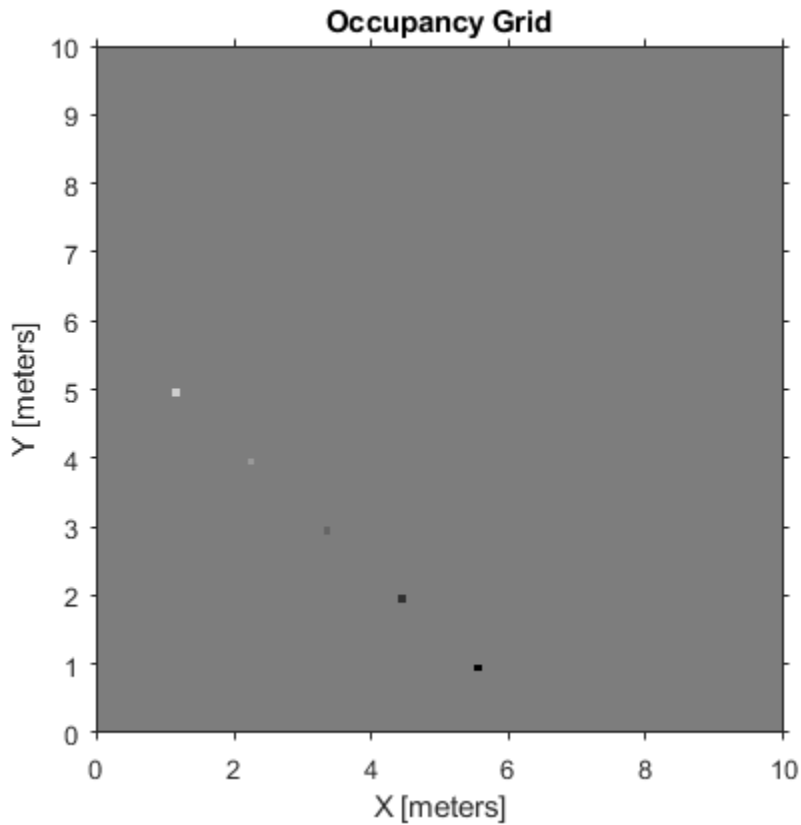
```
y = [5.0; 4.0; 3.0; 2.0; 1.0];
```

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

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

```
figure
```

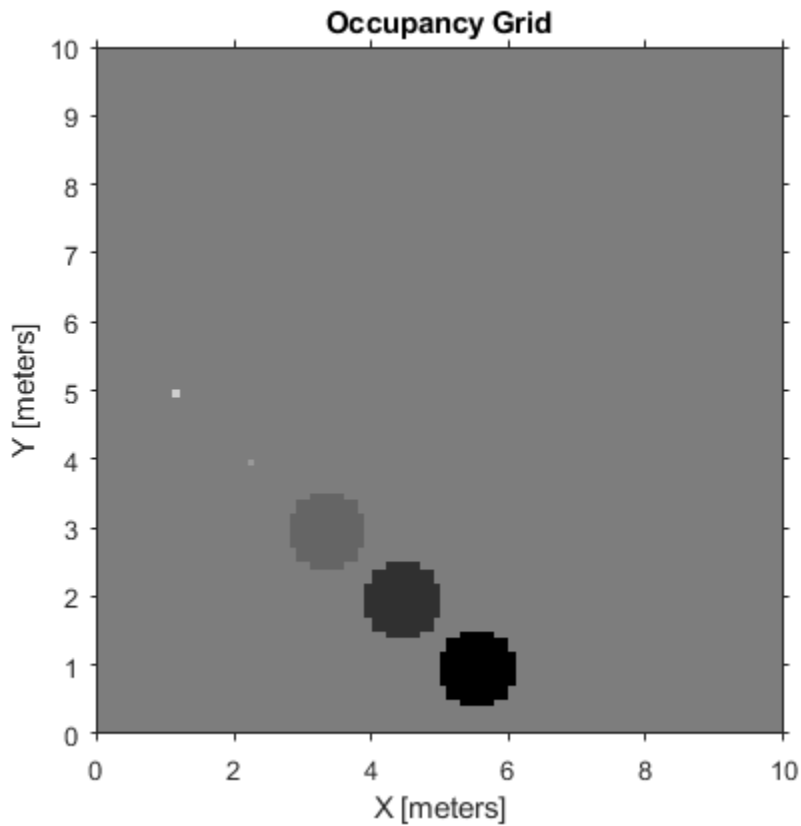
```
show(map)
```



Inflate occupied areas by a given radius. Larger occupancy values overwrite the smaller values.

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





Get grid locations from world locations.

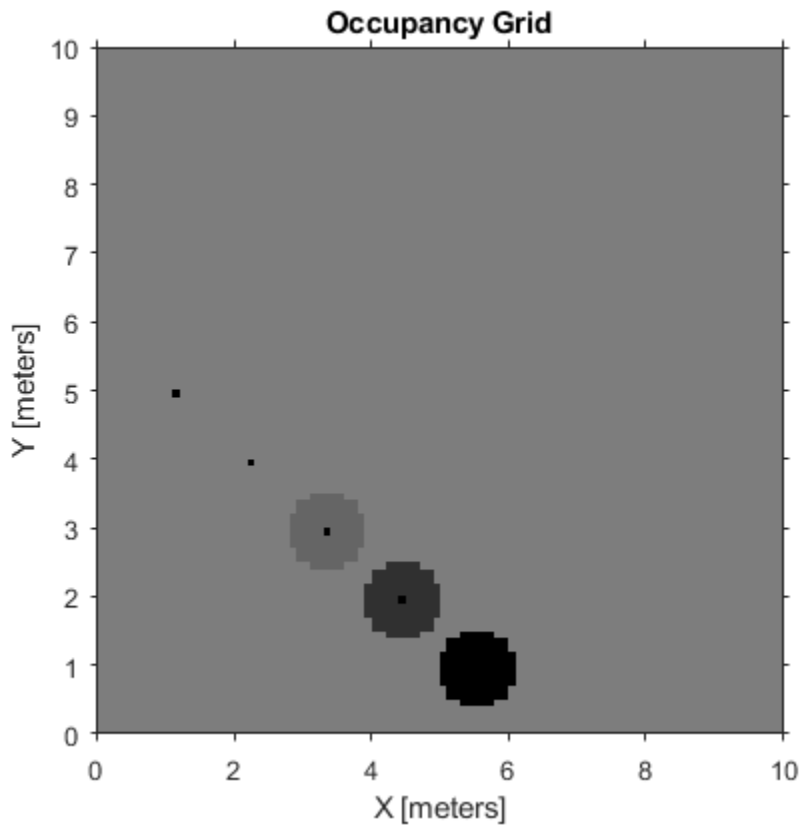
```
ij = world2grid(map,[x y]);
```

Set grid locations to occupied locations.

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

```
figure
```

```
show(map)
```



## Limitations

Occupancy values have a limited resolution of  $\pm 0.001$ . The values are stored as `int16` using a log-odds representation. This data type limits resolution, but saves you memory when storing large maps in MATLAB. When calling `set` and then `get`, the value returned might not equal the value you set. For more information, see the log-odds representations section in "Occupancy Grids".

## See Also

`robotics.BinaryOccupancyGrid` | `robotics.OccupancyGrid` |  
`robotics.OccupancyGrid.getOccupancy`

## Topics

“Occupancy Grids”

**Introduced in R2016b**

## show

**Class:** robotics.OccupancyGrid

**Package:** robotics

Show grid values in a figure

## Syntax

```
show(map)
```

```
show(map, "grid")
```

```
show( ____, "Parent", parent)
```

```
mapImage= show(map, ____)
```

## Description

`show(map)` displays the occupancy grid map in the current axes, with the axes labels representing the world coordinates.

`show(map, "grid")` displays the occupancy grid with the axes labels representing the grid coordinates.

`show( ____, "Parent", parent)` uses the axes handle specified as a parent to display the occupancy grid. Use any of the arguments from previous syntaxes.

`mapImage= show(map, ____)` returns the handle to the image object created by `show`.

## Input Arguments

### **map** — Map representation

OccupancyGrid object

Map representation, specified as a `robotics.OccupancyGrid` object. This object represents the environment of the robot. The object contains a matrix grid with values

representing the probability of the occupancy of that cell. Values close to 1 represent a high certainty that the cell contains an obstacle. Values close to 0 represent certainty that the cell is not occupied and obstacle free.

### **parent — Axes to plot the map**

Axes object | UIAxes object

Axes to plot the map specified as either an Axes or UIAxes object. See axes or uiaxes.

## **Outputs**

### **mapImage — Map image**

object handle

Map image, specified as an object handle.

## **Examples**

### **Create and Modify Occupancy Grid**

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

```
map = robotics.OccupancyGrid(10,10,10);
```

Update the occupancy of world locations with specific probability values.

```
map = robotics.OccupancyGrid(10,10,10);
```

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

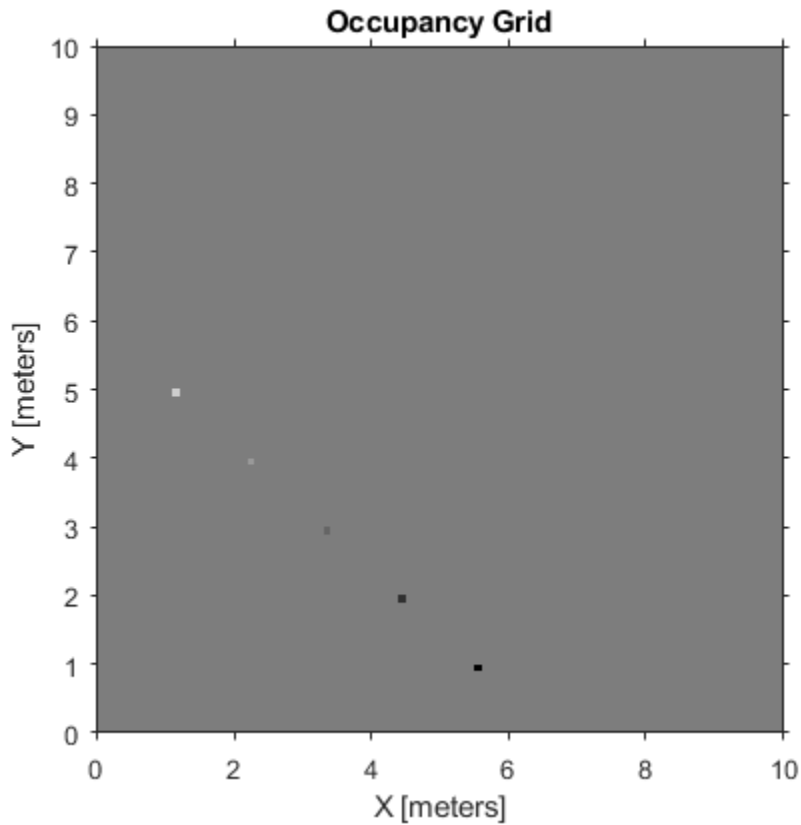
```
y = [5.0; 4.0; 3.0; 2.0; 1.0];
```

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

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

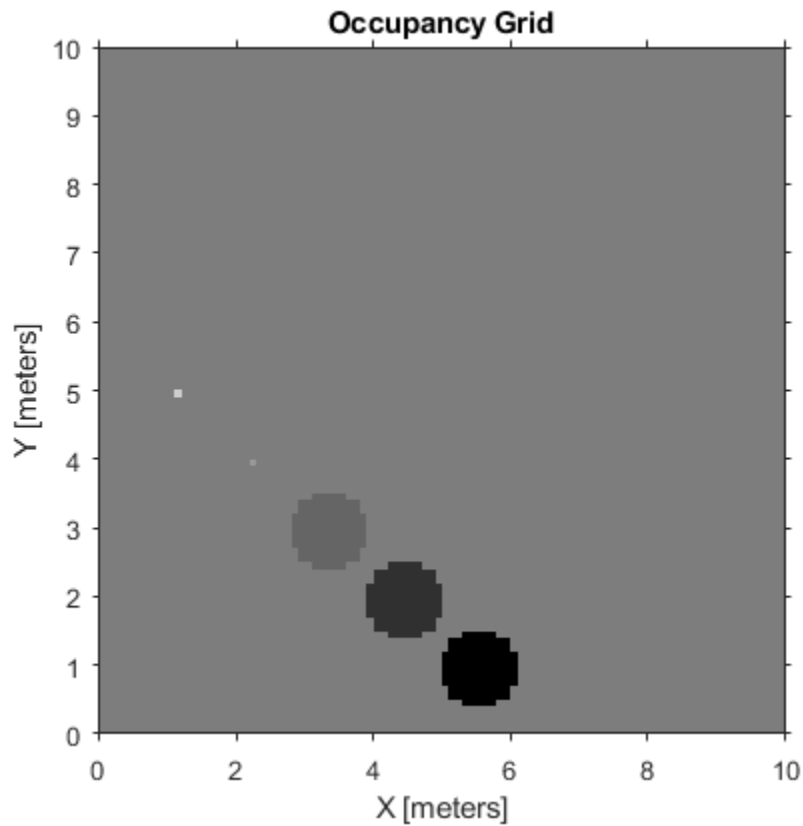
```
figure
```

```
show(map)
```



Inflate occupied areas by a given radius. Larger occupancy values overwrite the smaller values.

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



Get grid locations from world locations.

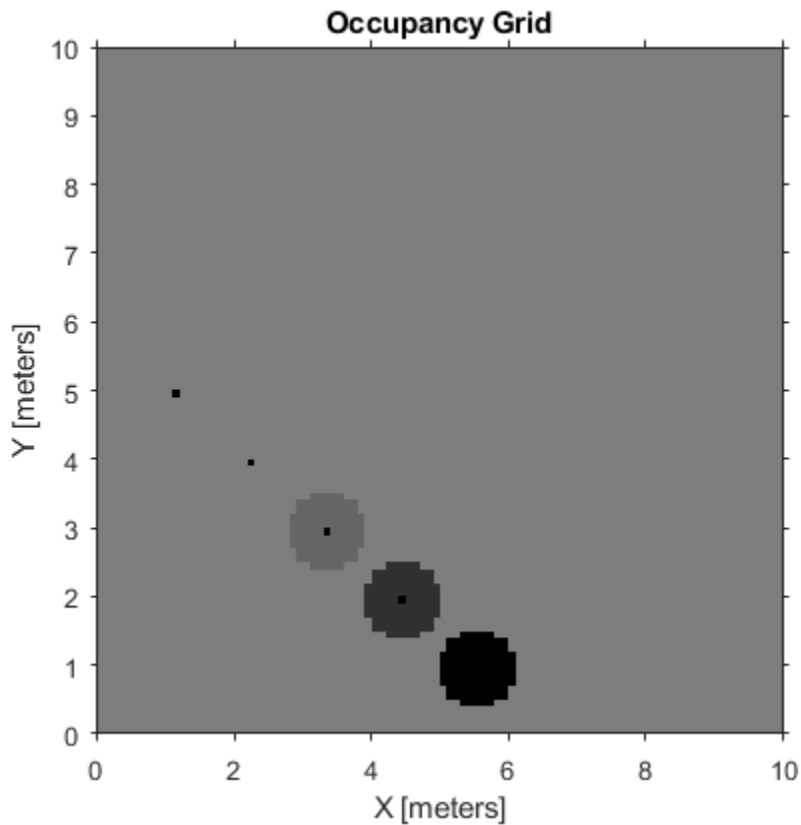
```
ij = world2grid(map,[x y]);
```

Set grid locations to occupied locations.

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

```
figure
```

```
show(map)
```



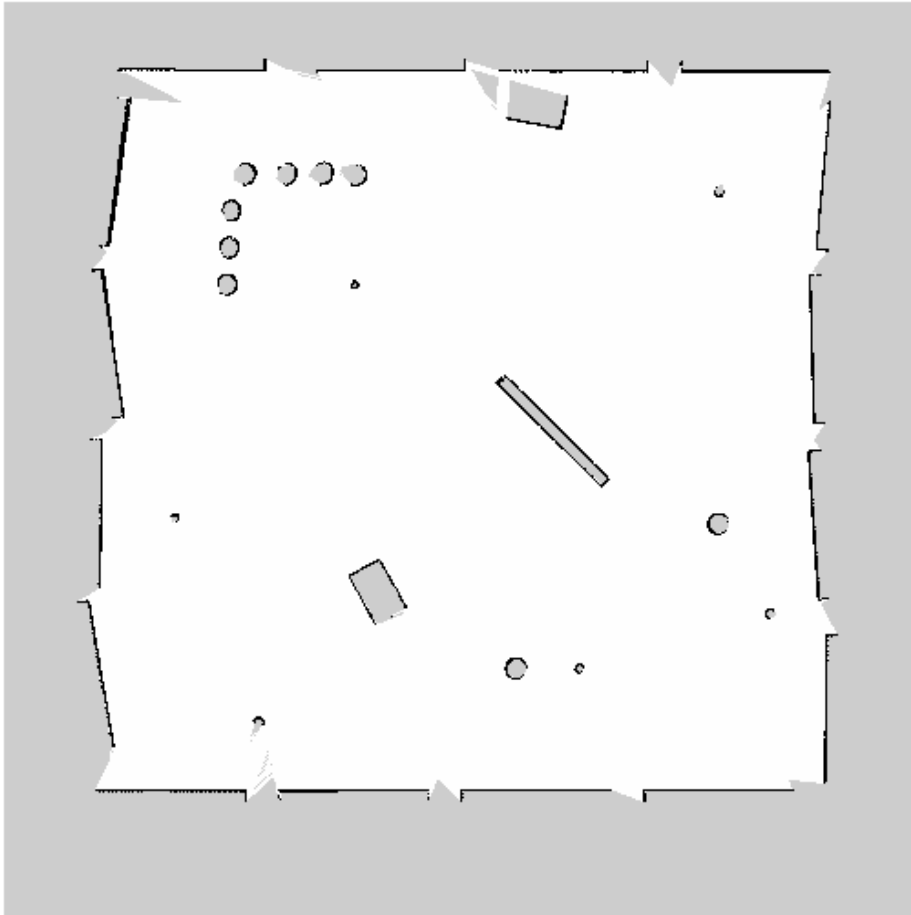
#### Convert PGM Image to Map

Convert a portable graymap (.pgm) file containing a ROS map into an `OccupancyGrid` map for use in MATLAB.

Import the image using `imread`. Crop the image to the relevant area.

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



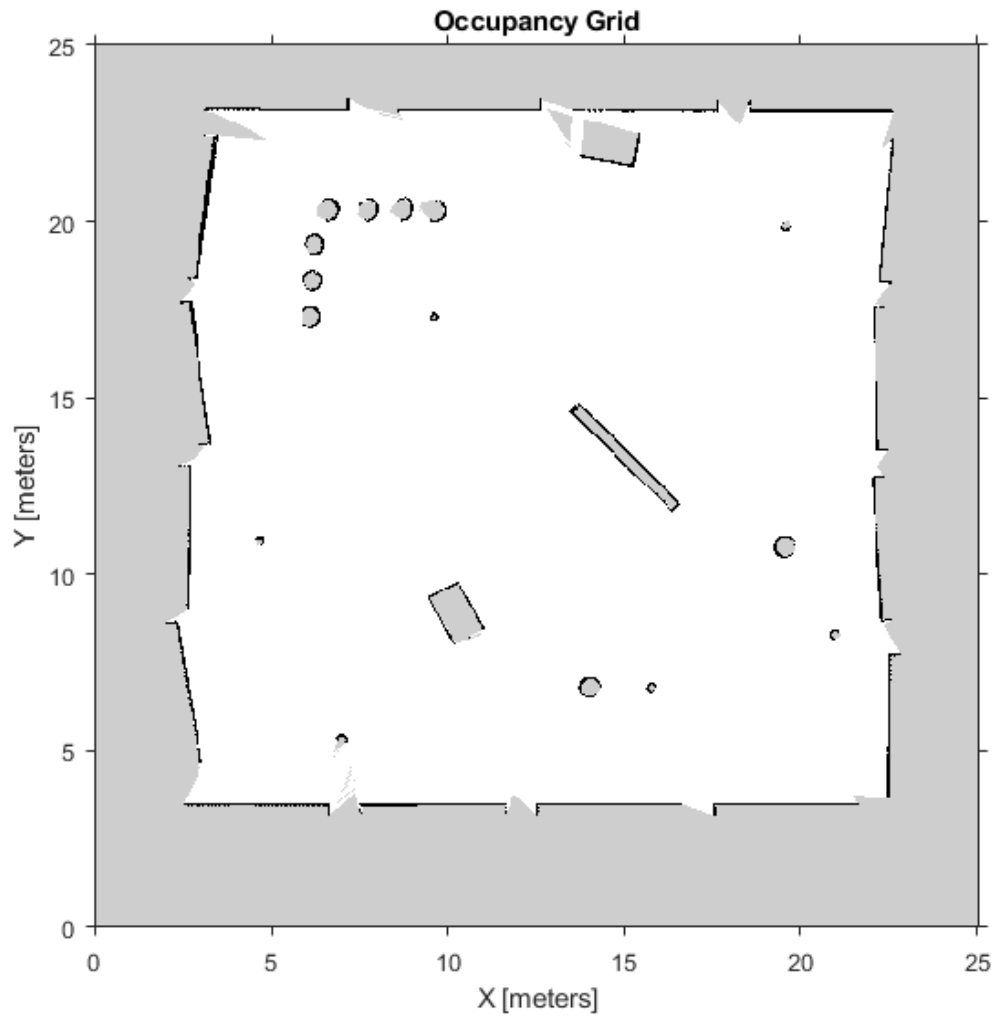


PGM values are expressed from 0 to 255 as `uint8`. Normalize these values by converting the cropped image to `double` and dividing each cell by 255. This image shows obstacles as values close to 0. Subtract the normalized image from 1 to get occupancy values with 1 representing occupied space.

```
imageNorm = double(imageCropped)/255;  
imageOccupancy = 1 - imageNorm;
```

Create the `OccupancyGrid` object using an adjusted map image. The imported map resolution is 20 cells per meter.

```
map = robotics.OccupancyGrid(imageOccupancy,20);  
show(map)
```



## See Also

[axes](#) | [occupancyMatrix](#) | [robotics.BinaryOccupancyGrid](#) | [robotics.OccupancyGrid](#)

**Introduced in R2016b**

# updateOccupancy

**Class:** robotics.OccupancyGrid

**Package:** robotics

Integrate probability observation at a location

## Syntax

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

## Description

`updateOccupancy(map, xy, obs)` probabilistically integrates the observation values, `obs`, to each coordinate specified in `xy`. Observation values are determined based on the "Inverse Sensor Model" on page 3-107.

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

## Input Arguments

### **map** — Map representation

OccupancyGrid object

Map representation, specified as a `robotics.OccupancyGrid` object. This object represents the environment of the robot. The object contains a matrix grid with values representing the probability of the occupancy of that cell. Values close to 1 represent a high certainty that the cell contains an obstacle. Values close to 0 represent certainty that the cell is not occupied and obstacle free.

### **xy** — World coordinates

*n*-by-2 matrix

World coordinates, specified as an  $n$ -by-2 vertical matrix of  $[x \ y]$  pairs, where  $n$  is the number of world coordinates.

Data Types: double

### **ij** — Grid positions

$n$ -by-2 matrix

Grid positions, specified as an  $n$ -by-2 matrix of  $[i \ j]$  pairs in  $[rows \ cols]$  format, where  $n$  is the number of grid positions.

Data Types: double

### **obs** — Probability observation values

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

Probability observation values, specified as a scalar or an  $n$ -by-1 column vector the same size as either `xy` or `ij`.

`obs` values can be any value from 0 to 1, but if `obs` is a logical array, the default observation values of 0.7 (`true`) and 0.4 (`false`) are used. These values correlate to the inverse sensor model for ray casting. If `obs` is a scalar or logical, the value is applied to all coordinates in `xy` or `ij`.

## Examples

### **Create and Modify Occupancy Grid**

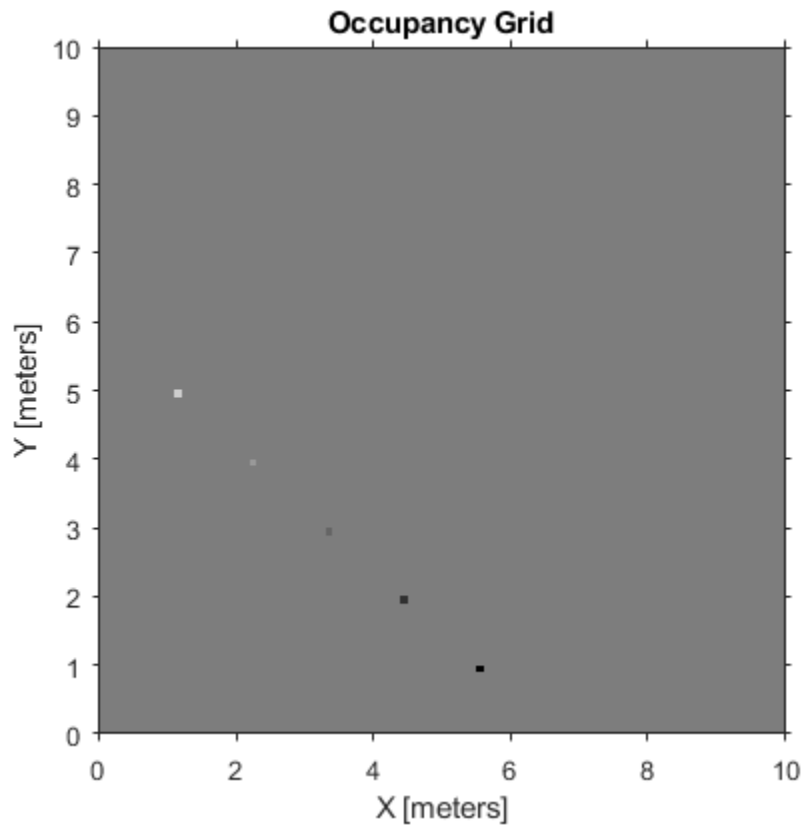
Create a 10m-by-10m empty map.

```
map = robotics.OccupancyGrid(10,10,10);
```

Update the occupancy of world locations with specific probability values.

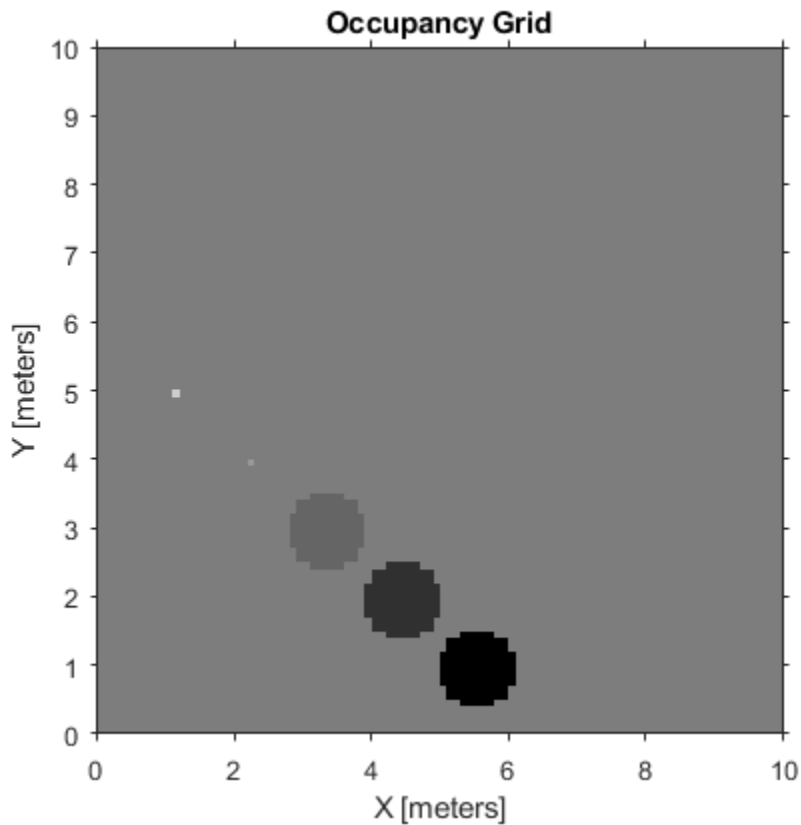
```
map = robotics.OccupancyGrid(10,10,10);  
x = [1.2; 2.3; 3.4; 4.5; 5.6];  
y = [5.0; 4.0; 3.0; 2.0; 1.0];  
  
pvalues = [0.2 0.4 0.6 0.8 1];  
  
updateOccupancy(map,[x y],pvalues)
```

```
figure  
show(map)
```



Inflate occupied areas by a given radius. Larger occupancy values overwrite the smaller values.

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



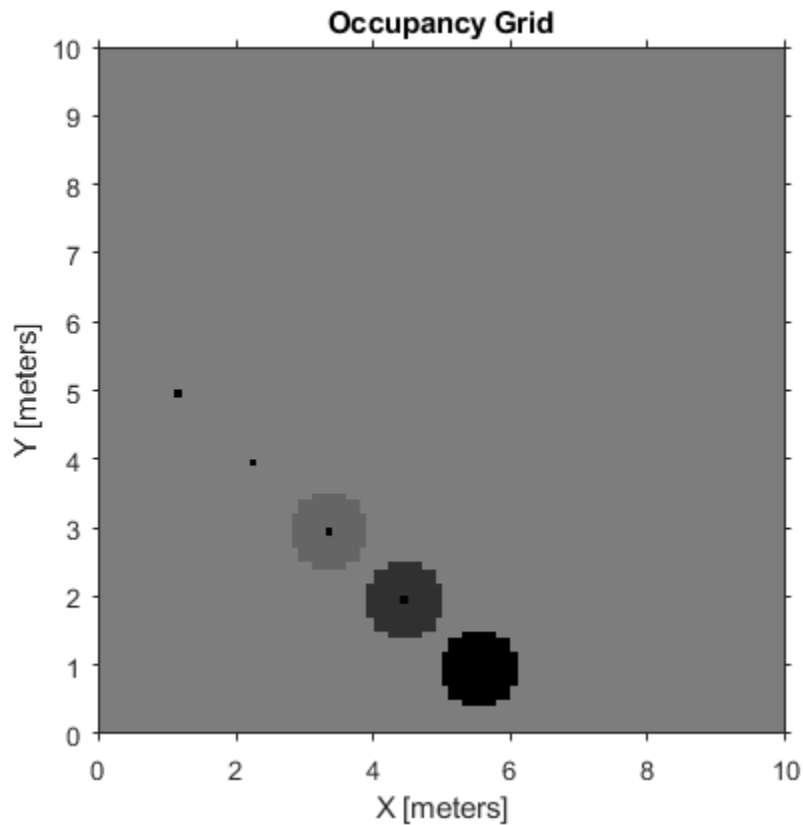
Get grid locations from world locations.

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

Set grid locations to occupied locations.

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

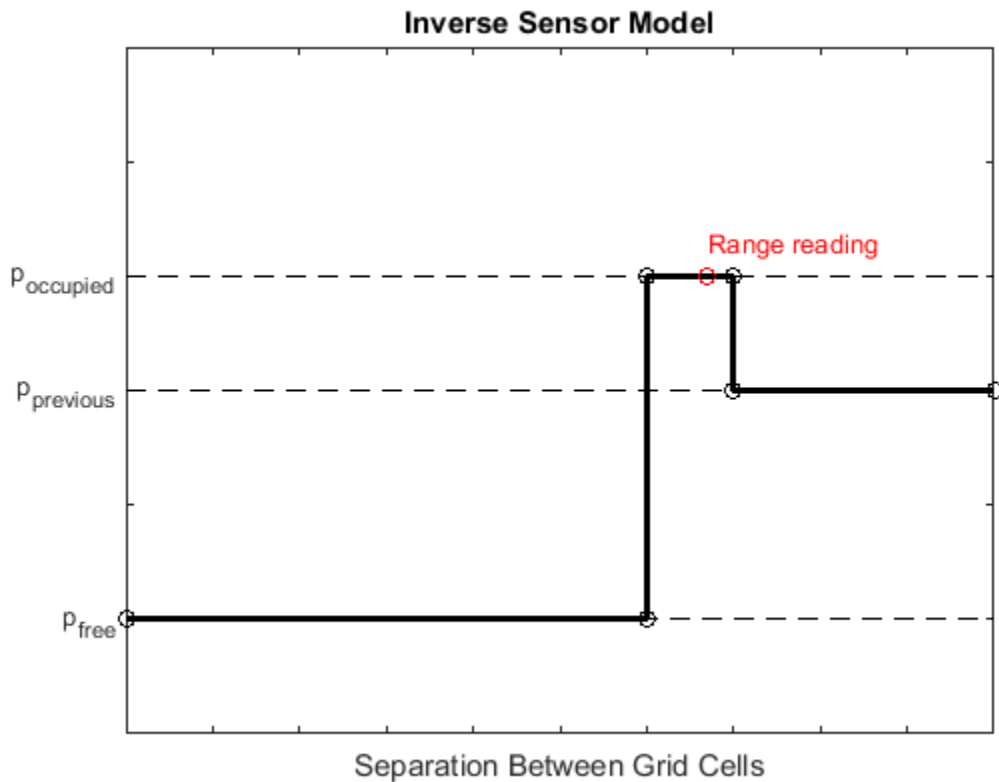




## Definitions

### Inverse Sensor Model

The inverse sensor model determines how values are set along a ray from a range sensor reading to the obstacles in the map. NaN range values are ignored. Range values greater than maxrange are not updated.



Grid locations that contain range readings are updated with the occupied probability. Locations before the reading are updated with the free probability. All locations after the reading are not updated.

## See Also

`robotics.BinaryOccupancyGrid` | `robotics.OccupancyGrid` |  
`robotics.OccupancyGrid.setOccupancy`

## Topics

“Occupancy Grids”

**Introduced in R2016b**

## world2grid

**Class:** `robotics.OccupancyGrid`

**Package:** `robotics`

Convert world coordinates to grid indices

### Syntax

```
ij = world2grid(map,xy)
```

### Description

`ij = world2grid(map,xy)` converts an array of world coordinates, `xy`, to an array of grid indices, `ij` in `[row col]` format.

### Input Arguments

**map — Map representation**

`OccupancyGrid` object

Map representation, specified as a `robotics.OccupancyGrid` object. This object represents the environment of the robot. The object contains a matrix grid with values representing the probability of the occupancy of that cell. Values close to 1 represent a high certainty that the cell contains an obstacle. Values close to 0 represent certainty that the cell is not occupied and obstacle free.

**xy — World coordinates**

*n*-by-2 matrix

World coordinates, specified as an *n*-by-2 matrix of `[x y]` pairs, where *n* is the number of world coordinates.

Data Types: `double`

## Output Arguments

### **ij** — Grid positions

*n*-by-2 matrix

Grid positions, returned as an *n*-by-2 matrix of [*i j*] pairs in [*rows cols*] format, where *n* is the number of grid positions. The grid cell locations are counted from the top left corner of the grid.

Data Types: double

## Examples

### **Create and Modify Occupancy Grid**

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

```
map = robotics.OccupancyGrid(10,10,10);
```

Update the occupancy of world locations with specific probability values.

```
map = robotics.OccupancyGrid(10,10,10);
```

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

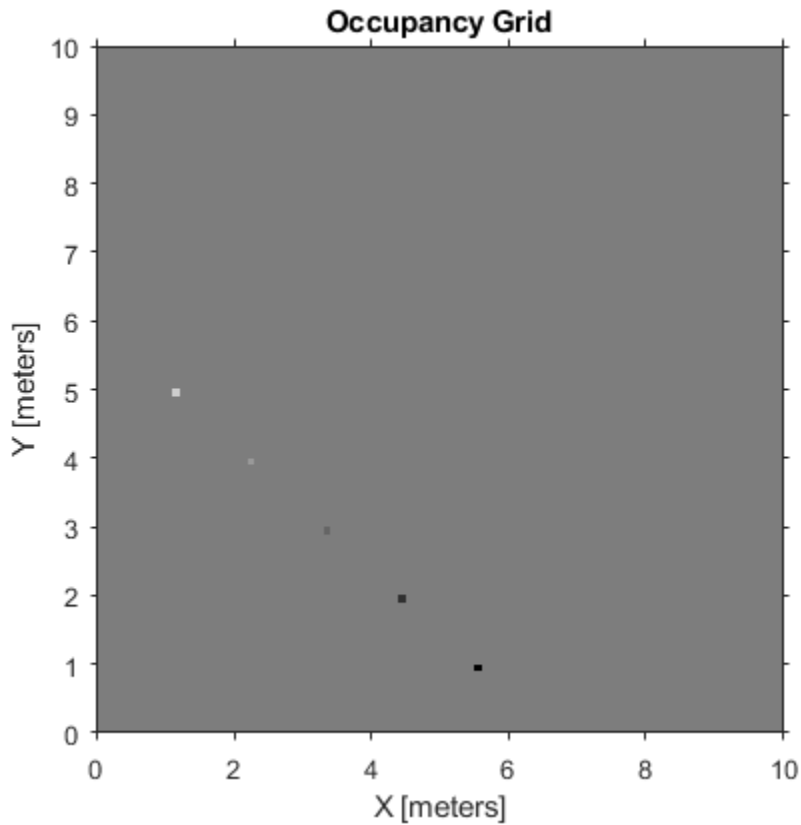
```
y = [5.0; 4.0; 3.0; 2.0; 1.0];
```

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

```
updateOccupancy(map,[x y],pvalues)
```

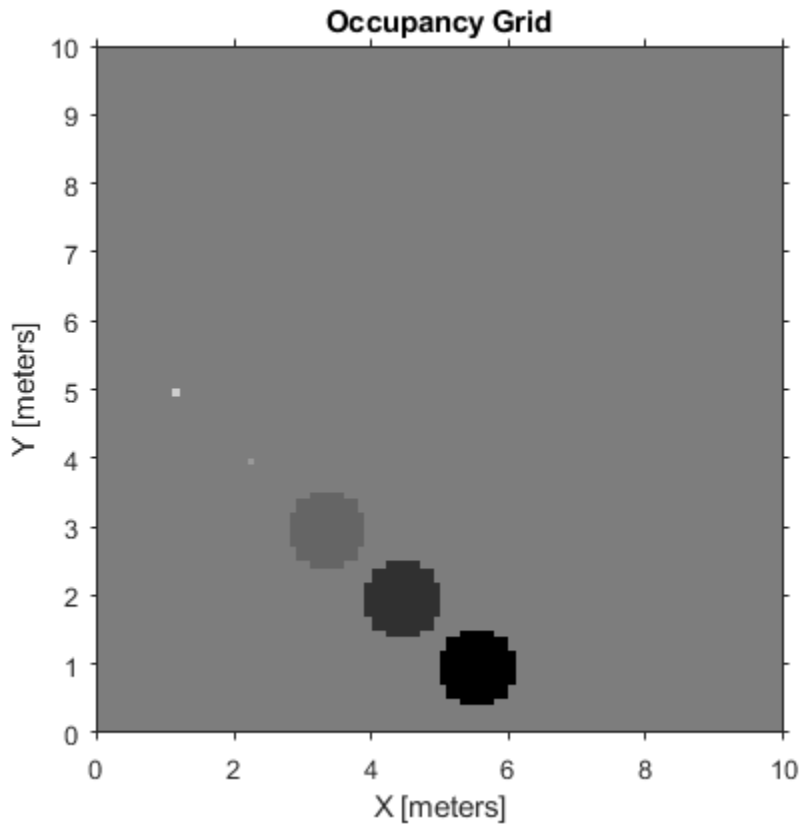
```
figure
```

```
show(map)
```



Inflate occupied areas by a given radius. Larger occupancy values overwrite the smaller values.

```
inflate(map,0.5)  
figure  
show(map)
```



Get grid locations from world locations.

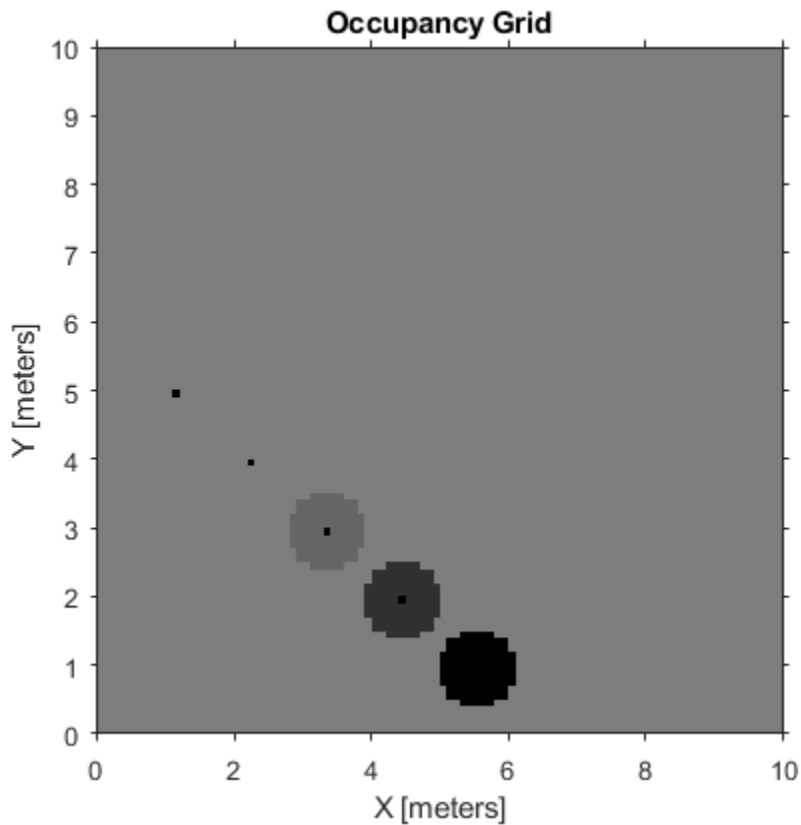
```
ij = world2grid(map,[x y]);
```

Set grid locations to occupied locations.

```
setOccupancy(map,ij,ones(5,1),'grid')
```

```
figure
```

```
show(map)
```



## See Also

`robotics.BinaryOccupancyGrid` | `robotics.OccupancyGrid` | `robotics.OccupancyGrid.grid2world`

## Topics

“Occupancy Grids”

**Introduced in R2016b**



# checkOccupancy

**Class:** robotics.OccupancyMap3D

**Package:** robotics

Check if locations are free or occupied

## Syntax

```
i0ccval = checkOccupancy(map3D, xyz)
```

## Description

`i0ccval = checkOccupancy(map3D, xyz)` returns an array of occupancy values specified at the `xyz` locations using the `OccupiedThreshold` and `FreeThreshold` properties of the input `OccupancyMap3D` object. Each row is a separate `xyz` location in the map to check the occupancy of. Occupancy values can be obstacle-free (0), occupied (1), or unknown (-1).

## Input Arguments

**map3D — 3-D occupancy map**

`OccupancyMap3D` object

3-D occupancy map, specified as an `OccupancyMap3D` object.

**xyz — World coordinates**

$n$ -by-3 matrix

World coordinates, specified as an  $n$ -by-3 matrix of  $[x \ y \ z]$  points, where  $n$  is the number of world coordinates.

# Output Arguments

## **i0ccval** — Interpreted occupancy values

column vector

Interpreted occupancy values, returned as a column vector with the same length as xyz.

Occupancy values can be obstacle-free (0), occupied (1), or unknown (-1). These values are determined from the actual probability values and the `OccupiedThreshold` and `FreeThreshold` properties of the `map3D` object.

# See Also

## **Classes**

[LidarSLAM](#) | [OccupancyGrid](#) | [OccupancyMap3D](#)

## **Functions**

[inflate](#) | [insertPointCloud](#) | [setOccupancy](#) | [show](#)

**Introduced in R2018a**

# getOccupancy

**Class:** robotics.OccupancyMap3D

**Package:** robotics

Get occupancy probability of locations

## Syntax

```
occval = getOccupancy(map3D, xyz)
```

## Description

`occval = getOccupancy(map3D, xyz)` returns an array of probability occupancy values at the specified `xyz` locations in the `OccupancyMap3D` object. Values close to 1 represent a high certainty that the cell contains an obstacle. Values close to 0 represent certainty that the cell is not occupied and obstacle-free.

## Input Arguments

**map3D** — 3-D occupancy map

`OccupancyMap3D` object

3-D occupancy map, specified as an `OccupancyMap3D` object.

**xyz** — World coordinates

*n*-by-3 matrix

World coordinates, specified as an *n*-by-3 matrix of [x y z] points, where *n* is the number of world coordinates.

## Output Arguments

**occval** — Probability occupancy values

column vector

Probability occupancy values, returned as a column vector with the same length as xyz.

Values close to 0 represent certainty that the cell is not occupied and obstacle-free.

## See Also

### Classes

LidarSLAM | OccupancyGrid | OccupancyMap3D

### Functions

inflate | insertPointCloud | setOccupancy | show

**Introduced in R2018a**

# inflate

**Class:** robotics.OccupancyMap3D

**Package:** robotics

Inflate map

## Syntax

```
inflate(map3D, radius)
```

## Description

`inflate(map3D, radius)` inflates each occupied position of the specified in the input `OccupancyMap3D` object by the `radius` specified in meters. `radius` is rounded up to the nearest equivalent cell based on the resolution of the map. This inflation increases the size of the occupied locations in the map.

## Input Arguments

**map3D — 3-D occupancy map**

`OccupancyMap3D` object

3-D occupancy map, specified as an `OccupancyMap3D` object.

**radius — Amount to inflate occupied locations**

scalar

Amount to inflate occupied locations, specified as a scalar. `radius` is rounded up to the nearest cell value.

## See Also

### Classes

LidarSLAM | OccupancyGrid | OccupancyMap3D

**Functions**

insertPointCloud | setOccupancy | show

**Introduced in R2018a**

# insertPointCloud

**Class:** robotics.OccupancyMap3D

**Package:** robotics

Insert 3-D points or point cloud observation into map

## Syntax

```
insertPointCloud(map3D,pose,points,maxrange)
insertPointCloud(map3D,pose,ptcloud,maxrange)
```

## Description

`insertPointCloud(map3D,pose,points,maxrange)` inserts one or more sensor observations at the given `points` in the occupancy map, `map3D`. Occupied points are updated with an observation of 0.7. All other points between the sensor `pose` and `points` are treated as obstacle-free and updated with an observation of 0.4. Points outside `maxrange` are not updated. NaN values are ignored.

`insertPointCloud(map3D,pose,ptcloud,maxrange)` inserts a `ptcloud` object into the map.

## Input Arguments

**map3D** — 3-D occupancy map

OccupancyMap3D object

3-D occupancy map, specified as a `OccupancyMap3D` object.

**points** — Points of point cloud

*n*-by-3 matrix

Points of point cloud in sensor coordinates, specified as an *n*-by-3 matrix of [x y z] points, where *n* is the number of points in the point cloud.

### **ptcCloud — Point cloud reading**

pointCloud object

Point cloud reading, specified as a pointCloud object.

---

**Note** Using pointCloud objects requires Computer Vision Toolbox™.

---

### **pose — Position and orientation of robot**

[x y z qw qx qy qz] vector

Position and orientation of robot, specified as an [x y z qw qx qy qz] vector. The robot pose is an xyz-position vector with a quaternion orientation vector specified as [qw qx qy qz].

### **maxrange — Maximum range of sensor**

scalar

Maximum range of point cloud sensor, specified as a scalar. Points outside this range are ignored.

## **See Also**

### **Classes**

LidarSLAM | OccupancyGrid | OccupancyMap3D

### **Functions**

inflate | setOccupancy | show

**Introduced in R2018a**



# setOccupancy

**Class:** robotics.OccupancyMap3D

**Package:** robotics

Set occupancy probability of locations

## Syntax

```
setOccupancy(map3D, xyz, occval)
```

## Description

`setOccupancy(map3D, xyz, occval)` assigns the occupancy values to each specified `xyz` coordinate in the 3-D occupancy map.

## Input Arguments

**map3D** — 3-D occupancy map

OccupancyMap3D object

3-D occupancy map, specified as an OccupancyMap3D object.

**xyz** — World coordinates

$n$ -by-3 matrix

World coordinates, specified as an  $n$ -by-3 matrix of  $[x \ y \ z]$  points, where  $n$  is the number of world coordinates.

**occval** — Probability occupancy values

scalar | column vector

Probability occupancy values, specified as a scalar or a column vector with the same length as `xyz`. A scalar input is applied to all coordinates in `xyz`.

Values close to 0 represent certainty that the cell is not occupied and obstacle-free.

## See Also

### Classes

LidarSLAM | OccupancyGrid | OccupancyMap3D

### Functions

inflate | insertPointCloud | setOccupancy | show

**Introduced in R2018a**

# show

**Class:** robotics.OccupancyMap3D

**Package:** robotics

Show occupancy map

## Syntax

```
axes = show(map3D)
show(map3D, "Parent" , parent)
```

## Description

`axes = show(map3D)` displays the occupancy map, `map3D`, in the current axes, with the axes labels representing the world coordinates.

The function displays the 3-D environment using 3-D voxels for areas with occupancy values greater than the `OccupiedThreshold` property value specified in `map3D`. The color of the 3-D plot is strictly height-based.

`show(map3D, "Parent" , parent)` displays the occupancy map in the axes handle specified by `parent`.

## Input Arguments

**map3D — 3-D occupancy map**

OccupancyMap3D object

3-D occupancy map, specified as an `OccupancyMap3D` object.

**parent — Axes used to plot the map**

Axes object | UIAxes object

Axes used to plot the map, specified as either an `Axes` or `UIAxes` object. See `axes` or `uiaxes`.

## Output Arguments

### **axes — Axes handle for map**

Axes object | UIAxes object

Axes handle for map, returned as either an Axes or UIAxesobject. See axes or uiaxes.

## See Also

### **Classes**

LidarSLAM | OccupancyGrid | OccupancyMap3D

### **Functions**

insertPointCloud | setOccupancy | show

**Introduced in R2018a**

# updateOccupancy

**Class:** robotics.OccupancyMap3D

**Package:** robotics

Update occupancy probability at locations

## Syntax

```
updateOccupancy(map3D, xyz, obs)
```

## Description

`updateOccupancy(map3D, xyz, obs)` probabilistically integrates the observation values, `obs`, to each specified `xyz` coordinate in the `OccupancyMap3D` object, `map3D`.

## Input Arguments

**map3D — 3-D occupancy map**

`OccupancyMap3D` object

3-D occupancy map, specified as an `OccupancyMap3D` object.

**xyz — World coordinates**

*n*-by-3 matrix

World coordinates, specified as an *n*-by-3 matrix of `[x y z]` points, where *n* is the number of world coordinates.

**obs — Probability observation values**

numeric scalar | logical scalar | *n*-by-1 column vector

Probability observation values, specified as a numeric or logical scalar, or as an *n*-by-1 column vector with the same size as `xyz`.

obs values can be from 0 to 1, but if obs is a logical array, the function uses the default observation values of 0.7 (true) and 0.4 (false). If obs is a numeric or logical scalar, the value is applied to all coordinates in xyz.

## See Also

### Classes

LidarSLAM | OccupancyGrid | OccupancyMap3D

### Functions

inflate | insertPointCloud | setOccupancy | show

**Introduced in R2018a**

# showNoiseDistribution

**Class:** robotics.OdometryMotionModel

**Package:** robotics

Display noise parameter effects

## Syntax

```
showNoiseDistribution(ommObj)
showNoiseDistribution(ommObj)
showNoiseDistribution(ommObj,Name,Value)
```

## Description

`showNoiseDistribution(ommObj)` shows the noise distribution for a default odometry pose update, number of samples and the current noise parameters on the input object.

`axes = showNoiseDistribution(ommObj)` shows the noise distribution and returns the axes handle.

`showNoiseDistribution(ommObj,Name,Value)` provides additional options specified by one or more `Name,Value` pairs. `Name` is the property name and `Value` is the corresponding value. `Name` must appear inside single quotes ( ' '). You can specify several name-value pair arguments in any order as `Name1,Value1, . . . ,NameN,ValueN`. Properties not specified retain their default values.

## Input Arguments

**ommObj — OdometryMotionModel object**  
handle

OdometryMotionModel object, specified as a handle. Create this object using `robotics.OdometryMotionModel`.

### Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

#### **OdometryPoseChange** — Change in odometry

three-element vector

Change in odometry of the robot, specified as a comma-separated pair consisting of 'OdometryPoseChange' and a three-element vector, `[x y theta]`.

#### **NumSamples** — Number of particles to display

scalar

Number of particles to display, specified as a specified as a comma-separated pair consisting of 'NumSamples' and a scalar.

#### **Parent** — Axes to plot the map

Axes object | UIAxes object

Axes to plot the map specified as a comma-separated pair consisting of 'Parent' and either an Axes or UIAxes object. See axes or uiaxes.

## Examples

### Show Noise Distribution Effects for Odometry Motion Model

This example shows how to visualize the effect of different noise parameters on the `robotics.OdometryMotionModel` class. An `OdometryMotionModel` object contains the motion model noise parameters for a differential drive robot. Use `showNoiseDistribution` to visualize how changing these values affect the distribution of predicted poses.

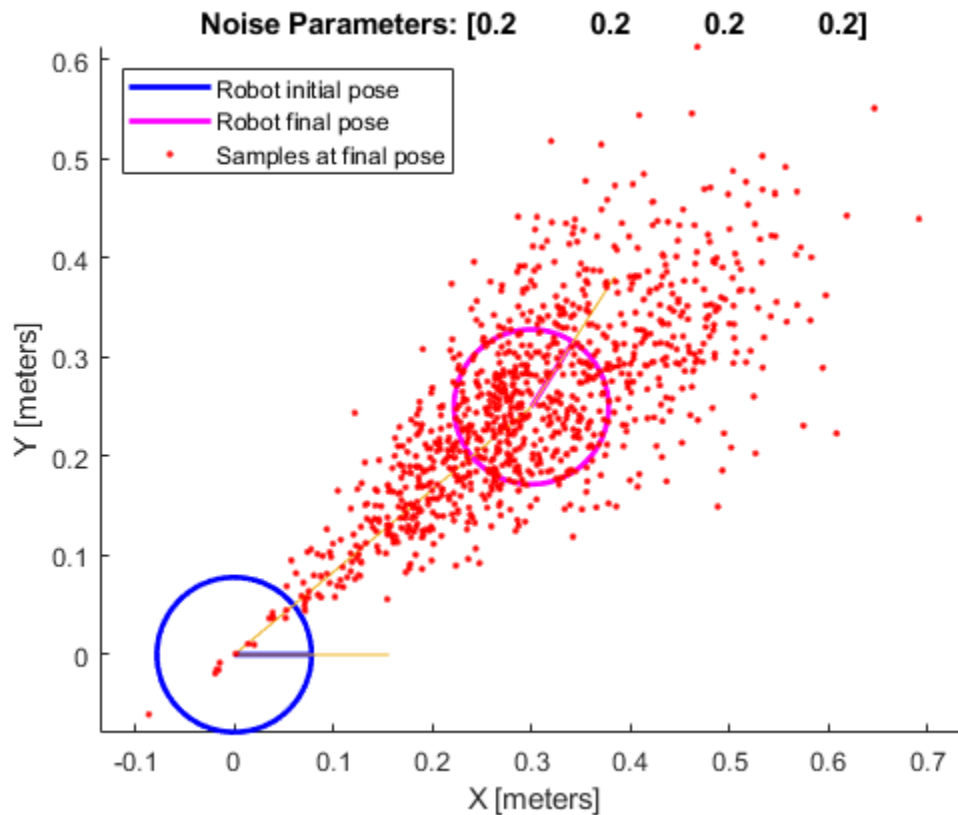
Create a motion model object.

```
motionModel = robotics.OdometryMotionModel;
```



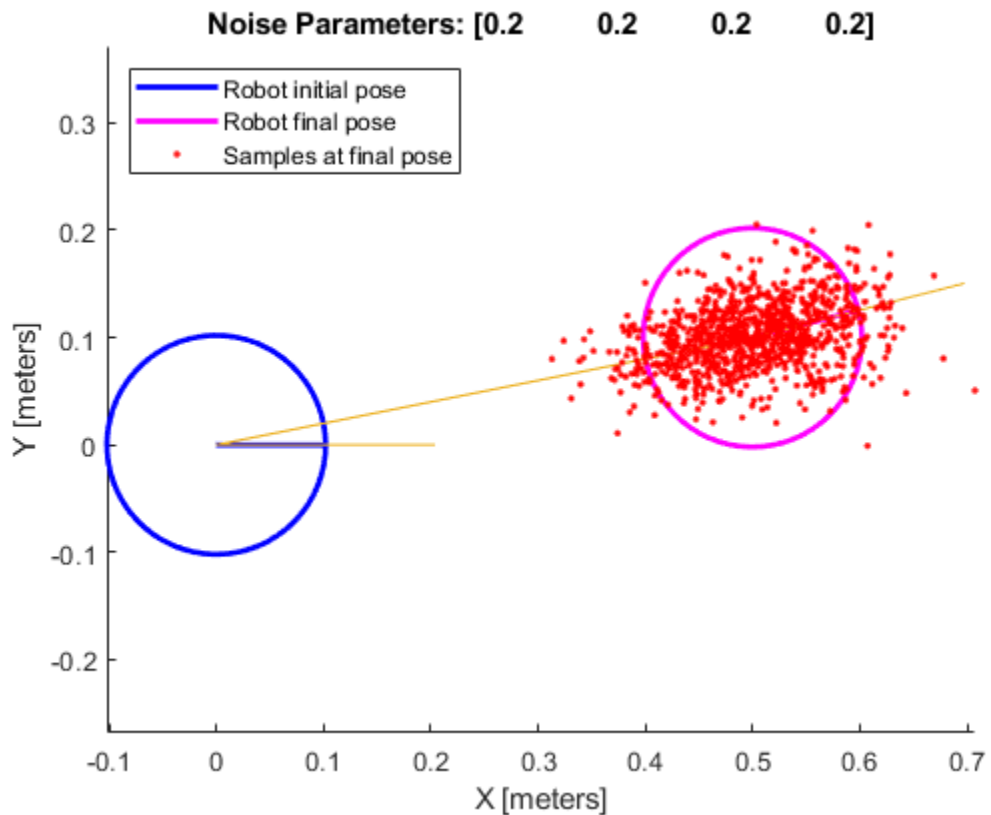
Show the distribution of particles with the existing noise parameters. Each particle is a hypothesis for the predicted pose.

```
showNoiseDistribution(motionModel);
```



Show the distribution with a specified odometry pose change and number of samples. The change in odometry is used as the final pose with hypotheses distributed around based on the Noise parameters.

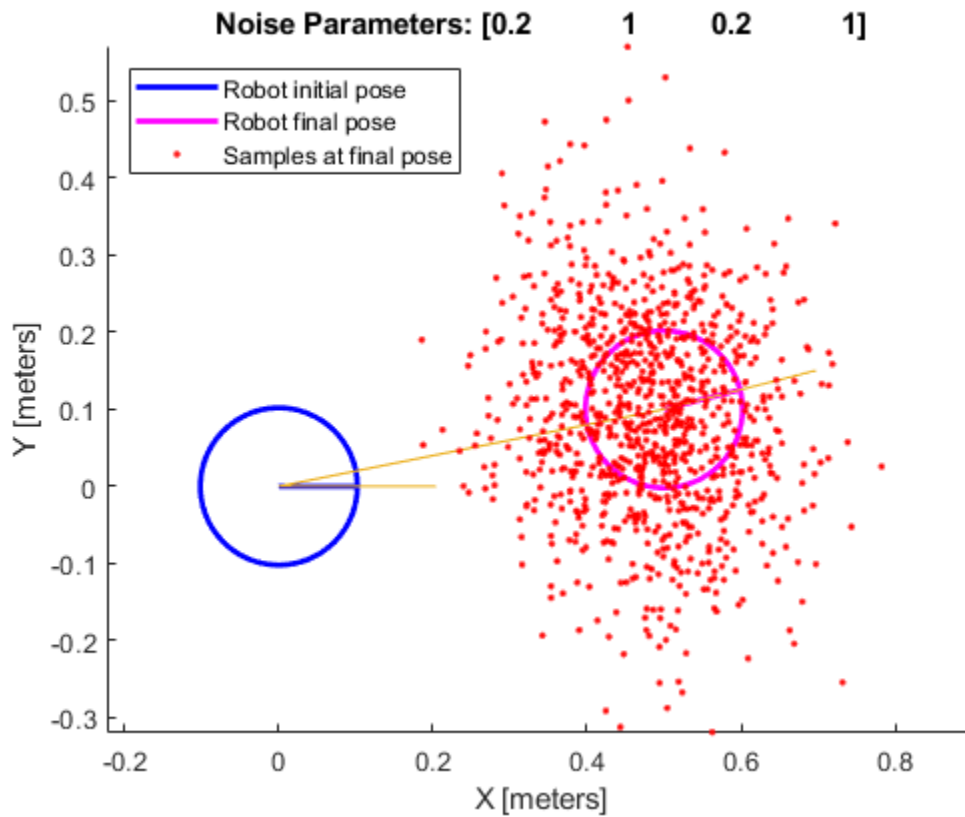
```
showNoiseDistribution(motionModel, ...
    'OdometryPoseChange', [0.5 0.1 0.25], ...
    'NumSamples', 1000);
```



Change the Noise parameters and visualize the effects. Use the same odometry pose change and number of samples.

```
motionModel.Noise = [0.2 1 0.2 1];

showNoiseDistribution(motionModel, ...
    'OdometryPoseChange', [0.5 0.1 0.25], ...
    'NumSamples', 1000);
```



## See Also

`robotics.LikelihoodFieldSensorModel` | `robotics.MonteCarloLocalization` | `robotics.OdometryMotionModel`

Introduced in R2016b

## step

**Class:** robotics.OdometryMotionModel

**Package:** robotics

Computer next pose from previous pose

## Syntax

```
currentPoses = step(ommObj,previousPoses,odomPose)
```

## Description

`currentPoses = step(ommObj,previousPoses,odomPose)` returns the current poses by propagating the previous poses using a sampling-based odometry motion model, which uses the difference between the specified `odomPose` and the `LastOdometryPose` property of the `ommObj`. The first `step` call instantiates the object and sets the `LastOdometryPose` property.

## Input Arguments

**ommObj — OdometryMotionModel object**

handle

OdometryMotionModel object, specified as a handle. Create this object using `robotics.OdometryMotionModel`.

**previousPoses — Previous poses**

*n*-by-3 array

Previous poses, specified as an *n*-by-3 array, [*x* *y* *theta*]. Each row of the `previousPoses` vector is treated as a separate robot and a corresponding predicted pose is present in `currentPoses`

**odomPose — Current robot pose**

three-element vector

Current robot pose, specified as a three-element vector, [x y theta].

## Output Arguments

### **currentPoses** — Current poses

*n*-by-3 array

Current poses, returned as an *n*-by-3 array, [x y theta]. Each row of the `previousPoses` vector is treated as a separate robot and a corresponding predicted pose is present in `currentPoses`.

## Examples

### **Predict Poses Based On An Odometry Motion Model**

This example shows how to use the `robotics.OdometryMotionModel` class to predict the pose of a robot. An `OdometryMotionModel` object contains the motion model parameters for a differential drive robot. Use the object to predict the pose of a robot based on its current and previous poses and the motion model parameters.

Create odometry motion model object.

```
motionModel = robotics.OdometryMotionModel;
```

Define previous poses and the current odometry reading. Each pose prediction corresponds to a row in `previousPoses` vector.

```
previousPoses = rand(10,3);  
currentOdom = [0.1 0.1 0.1];
```

The first call to the object initializes values and returns the previous poses as the current poses.

```
currentPoses = motionModel(previousPoses, currentOdom);
```

Subsequent calls to the object with updated odometry poses returns the predicted poses based on the motion model.

```
currentOdom = currentOdom + [0.1 0.1 0.05];  
predPoses = motionModel(previousPoses, currentOdom);
```

## **See Also**

`robotics.LikelihoodFieldSensorModel` | `robotics.MonteCarloLocalization` |  
`robotics.OdometryMotionModel`

**Introduced in R2016b**

## copy

**Class:** robotics.ParticleFilter

**Package:** robotics

Create copy of particle filter

## Syntax

```
b = copy(a)
```

## Description

`b = copy(a)` copies each element in the array of handles, `a`, to the new array of handles, `b`.

The `copy` method does not copy dependent properties. MATLAB does not call `copy` recursively on any handles contained in property values. MATLAB also does not call the class constructor or property-set methods during the copy operation.

## Input Arguments

**a** — Object array

handle

Object array, specified as a handle.

## Output Arguments

**b** — Object array containing copies of the objects in `a`

handle

Object array containing copies of the object in `a`, specified as a handle.

`b` has the same number of elements and is the same size and class of `a`. `b` is the same class as `a`. If `a` is empty, `b` is also empty. If `a` is heterogeneous, `b` is also heterogeneous. If `a` contains deleted handles, then `copy` creates deleted handles of the same class in `b`. Dynamic properties and listeners associated with objects in `a` are not copied to objects in `b`.

### See Also

`robotics.ParticleFilter`

### Topics

“Particle Filter Parameters”

“Particle Filter Workflow”

**Introduced in R2016a**



## correct

**Class:** robotics.ParticleFilter

**Package:** robotics

Adjust state estimate based on sensor measurement

## Syntax

```
[stateCorr, stateCov] = correct(pf, measurement)
[stateCorr, stateCov] = correct(pf, measurement, varargin)
```

## Description

`[stateCorr, stateCov] = correct(pf, measurement)` calculates the corrected system state and its associated uncertainty covariance based on a sensor measurement at the current time step. `correct` uses the `MeasurementLikelihoodFcn` property from the particle filter object, `pf`, as a function to calculate the likelihood of the sensor measurement for each particle. The two inputs to the `MeasurementLikelihoodFcn` function are:

- 1 `pf` - The `ParticleFilter` object, which contains the particles of the current iteration
- 2 `measurement` - The sensor measurements used to correct the state estimate

The `MeasurementLikelihoodFcn` function then extracts the best state estimate and covariance based on the setting in the `StateEstimationMethod` property.

`[stateCorr, stateCov] = correct(pf, measurement, varargin)` passes all additional arguments in `varargin` to the underlying `MeasurementLikelihoodFcn` after the first three required inputs.

## Input Arguments

**pf** — **ParticleFilter** object  
handle

ParticleFilter object, specified as a handle. See `robotics.ParticleFilter` for more information.

**measurement — Sensor measurements**

array

Sensor measurements, specified as an array. This input is passed directly into the `MeasurementLikelihoodFcn` property of `pf`. It is used to calculate the likelihood of the sensor measurement for each particle.

**varargin — Variable-length input argument list**

comma-separated list

Variable-length input argument list, specified as a comma-separated list. This input is passed directly into the `MeasurementLikelihoodFcn` property of `pf`. It is used to calculate the likelihood of the sensor measurement for each particle. When you call:

```
correct(pf,measurement,arg1,arg2)
```

MATLAB essentially calls `measurementLikelihoodFcn` as:

```
measurementLikelihoodFcn(pf,measurement,arg1,arg2)
```

## Output Arguments

**stateCorr — Corrected system state**

vector with length `NumStateVariables`

Corrected system state, returned as a row vector with length `NumStateVariables`. The corrected state is calculated based on the `StateEstimationMethod` algorithm and the `MeasurementLikelihoodFcn`.

**stateCov — Corrected system covariance**

$N$ -by- $N$  matrix | []

Corrected system variance, returned as an  $N$ -by- $N$  matrix, where  $N$  is the value of `NumStateVariables` property from `pf`. The corrected state is calculated based on the `StateEstimationMethod` algorithm and the `MeasurementLikelihoodFcn`. If you specify a state estimate method that does not support covariance, then the function returns `stateCov` as [].

## Examples

### Particle Filter Prediction and Correction

Create a `ParticleFilter` object, and execute a prediction and correction step for state estimation. The particle filter gives a predicted state estimate based on the return value of `StateTransitionFcn`. It then corrects the state based on a given measurement and the return value of `MeasurementLikelihoodFcn`.

Create a particle filter with the default three states.

```
pf = robotics.ParticleFilter

pf =
  ParticleFilter with properties:
      NumStateVariables: 3
      NumParticles: 1000
      StateTransitionFcn: @robotics.algs.gaussianMotion
      MeasurementLikelihoodFcn: @robotics.algs.fullStateMeasurement
      IsStateVariableCircular: [0 0 0]
      ResamplingPolicy: [1x1 robotics.ResamplingPolicy]
      ResamplingMethod: 'multinomial'
      StateEstimationMethod: 'mean'
      StateOrientation: 'row'
      Particles: [1000x3 double]
      Weights: [1000x1 double]
      State: 'Use the getStateEstimate function to see the value.'
      StateCovariance: 'Use the getStateEstimate function to see the value.'
```

Specify the mean state estimation method and systematic resampling method.

```
pf.StateEstimationMethod = 'mean';
pf.ResamplingMethod = 'systematic';
```

Initialize the particle filter at state `[4 1 9]` with unit covariance (`eye(3)`). Use 5000 particles.

```
initialize(pf,5000,[4 1 9],eye(3));
```

Assuming a measurement `[4.2 0.9 9]`, run one predict and one correct step.

```
[statePredicted, stateCov] = predict(pf);  
[stateCorrected, stateCov] = correct(pf, [4.2 0.9 9]);
```

Get the best state estimate based on the StateEstimationMethod algorithm.

```
stateEst = getStateEstimate(pf)
```

```
stateEst = 1×3
```

```
    4.1562    0.9185    9.0202
```

## See Also

[robotics.ParticleFilter](#) | [robotics.ParticleFilter.getStateEstimate](#) | [robotics.ParticleFilter.initialize](#) | [robotics.ParticleFilter.predict](#)

## Topics

[“Track a Car-Like Robot Using Particle Filter”](#)

[“Particle Filter Parameters”](#)

[“Particle Filter Workflow”](#)

**Introduced in R2016a**

# getStateEstimate

**Class:** robotics.ParticleFilter

**Package:** robotics

Extract best state estimate and covariance from particles

## Syntax

```
stateEst = getStateEstimate(pf)
[stateEst, stateCov] = getStateEstimate(pf)
```

## Description

`stateEst = getStateEstimate(pf)` returns the best state estimate based on the current set of particles. The estimate is extracted based on the `StateEstimationMethod` property from the `ParticleFilter` object, `pf`.

`[stateEst, stateCov] = getStateEstimate(pf)` also returns the covariance around the state estimate. The covariance is a measure of the uncertainty of the state estimate. Not all state estimate methods support covariance output. In this case, `getStateEstimate` returns `stateCov` as `[]`.

## Input Arguments

**pf** — ParticleFilter object

handle

`ParticleFilter` object, specified as a handle. See `robotics.ParticleFilter` for more information.

## Output Arguments

### **stateEst** — Best state estimate

vector

Best state estimate, returned as a row vector with length `NumStateVariables`. The estimate is extracted based on the `StateEstimationMethod` algorithm specified in `pf`.

### **stateCov** — Corrected system covariance

*N*-by-*N* matrix | []

Corrected system variance, returned as an *N*-by-*N* matrix, where *N* is the value of `NumStateVariables` property from `pf`. The corrected state is calculated based on the `StateEstimationMethod` algorithm and the `MeasurementLikelihoodFcn`. If you specify a state estimate method that does not support covariance, then the function returns `stateCov` as [].

## Examples

### Particle Filter Prediction and Correction

Create a `ParticleFilter` object, and execute a prediction and correction step for state estimation. The particle filter gives a predicted state estimate based on the return value of `StateTransitionFcn`. It then corrects the state based on a given measurement and the return value of `MeasurementLikelihoodFcn`.

Create a particle filter with the default three states.

```
pf = robotics.ParticleFilter
```

```
pf =
```

```
ParticleFilter with properties:
```

```
    NumStateVariables: 3
      NumParticles: 1000
    StateTransitionFcn: @robotics.algs.gaussianMotion
MeasurementLikelihoodFcn: @robotics.algs.fullStateMeasurement
  IsStateVariableCircular: [0 0 0]
    ResamplingPolicy: [1x1 robotics.ResamplingPolicy]
    ResamplingMethod: 'multinomial'
```

```

StateEstimationMethod: 'mean'
  StateOrientation: 'row'
    Particles: [1000x3 double]
    Weights: [1000x1 double]
      State: 'Use the getStateEstimate function to see the value.'
    StateCovariance: 'Use the getStateEstimate function to see the value.'

```

Specify the mean state estimation method and systematic resampling method.

```

pf.StateEstimationMethod = 'mean';
pf.ResamplingMethod = 'systematic';

```

Initialize the particle filter at state [4 1 9] with unit covariance (`eye(3)`). Use 5000 particles.

```

initialize(pf,5000,[4 1 9],eye(3));

```

Assuming a measurement [4.2 0.9 9], run one predict and one correct step.

```

[statePredicted,stateCov] = predict(pf);
[stateCorrected,stateCov] = correct(pf,[4.2 0.9 9]);

```

Get the best state estimate based on the `StateEstimationMethod` algorithm.

```

stateEst = getStateEstimate(pf)

```

```

stateEst = 1×3

```

```

    4.1562    0.9185    9.0202

```

## See Also

`robotics.ParticleFilter` | `robotics.ParticleFilter.correct` |  
`robotics.ParticleFilter.initialize` | `robotics.ParticleFilter.predict`

## Topics

“Track a Car-Like Robot Using Particle Filter”  
 “Particle Filter Parameters”  
 “Particle Filter Workflow”

**Introduced in R2016a**



# initialize

**Class:** robotics.ParticleFilter

**Package:** robotics

Initialize the state of the particle filter

## Syntax

```
initialize(pf,numParticles,mean,covariance)
initialize(pf,numParticles,stateBounds)
initialize( __ ,Name,Value)
```

## Description

`initialize(pf,numParticles,mean,covariance)` initializes the particle filter object, `pf`, with a specified number of particles, `numParticles`. The initial states of the particles in the state space are determined by sampling from the multivariate normal distribution with the specified mean and covariance.

`initialize(pf,numParticles,stateBounds)` determines the initial location of the particles by sample from the multivariate uniform distribution within the specified `stateBounds`.

`initialize( __ ,Name,Value)` initializes the particles with additional options specified by one or more `Name, Value` pair arguments.

## Input Arguments

**pf** — ParticleFilter object

handle

ParticleFilter object, specified as a handle. See `robotics.ParticleFilter` for more information.

**numParticles — Number of particles used in the filter**

scalar

Number of particles used in the filter, specified as a scalar.

**mean — Mean of particle distribution**

vector

Mean of particle distribution, specified as a vector. The `NumStateVariables` property of `pf` is set based on the length of this vector.

**covariance — Covariance of particle distribution**

*N*-by-*N* matrix

Covariance of particle distribution, specified as an *N*-by-*N* matrix, where *N* is the value of `NumStateVariables` property from `pf`.

**stateBounds — Bounds of state variables**

*n*-by-2 matrix

Bounds of state variables, specified as an *n*-by-2 matrix. The `NumStateVariables` property of `pf` is set based on the value of *n*. Each row corresponds to the lower and upper limit of the corresponding state variable.

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1, Value1, . . . , NameN, ValueN`.

**CircularVariables — Circular variables**

logical vector

Circular variables, specified as a logical vector. Each state variable that uses circular or angular coordinates is indicated with a 1. The length of the vector is equal to the `NumStateVariables` property of `pf`.

## Examples

## Particle Filter Prediction and Correction

Create a `ParticleFilter` object, and execute a prediction and correction step for state estimation. The particle filter gives a predicted state estimate based on the return value of `StateTransitionFcn`. It then corrects the state based on a given measurement and the return value of `MeasurementLikelihoodFcn`.

Create a particle filter with the default three states.

```
pf = robotics.ParticleFilter
```

```
pf =
  ParticleFilter with properties:

    NumStateVariables: 3
    NumParticles: 1000
    StateTransitionFcn: @robotics.algs.gaussianMotion
    MeasurementLikelihoodFcn: @robotics.algs.fullStateMeasurement
    IsStateVariableCircular: [0 0 0]
    ResamplingPolicy: [1x1 robotics.ResamplingPolicy]
    ResamplingMethod: 'multinomial'
    StateEstimationMethod: 'mean'
    StateOrientation: 'row'
    Particles: [1000x3 double]
    Weights: [1000x1 double]
    State: 'Use the getStateEstimate function to see the value.'
    StateCovariance: 'Use the getStateEstimate function to see the value.'
```

Specify the mean state estimation method and systematic resampling method.

```
pf.StateEstimationMethod = 'mean';
pf.ResamplingMethod = 'systematic';
```

Initialize the particle filter at state [4 1 9] with unit covariance (`eye(3)`). Use 5000 particles.

```
initialize(pf,5000,[4 1 9],eye(3));
```

Assuming a measurement [4.2 0.9 9], run one predict and one correct step.

```
[statePredicted,stateCov] = predict(pf);
[stateCorrected,stateCov] = correct(pf,[4.2 0.9 9]);
```

Get the best state estimate based on the `StateEstimationMethod` algorithm.

```
stateEst = getStateEstimate(pf)
```

```
stateEst = 1x3
```

```
    4.1562    0.9185    9.0202
```

### See Also

[robotics.ParticleFilter.correct](#) |  
[robotics.ParticleFilter.getStateEstimate](#) |  
[robotics.ParticleFilter.predict](#) | [robotics.ParticleFilter.predict](#)

### Topics

[“Track a Car-Like Robot Using Particle Filter”](#)  
[“Particle Filter Parameters”](#)  
[“Particle Filter Workflow”](#)

**Introduced in R2016a**

# predict

**Class:** robotics.ParticleFilter

**Package:** robotics

Predict state of robot in next time step

## Syntax

```
[statePred, stateCov] = predict(pf)
[statePred, stateCov] = predict(pf, varargin)
```

## Description

`[statePred, stateCov] = predict(pf)` calculates the predicted system state and its associated uncertainty covariance. `predict` uses the `StateTransitionFcn` property of `ParticleFilter` object, `pf`, to evolve the state of all particles. It then extracts the best state estimate and covariance based on the setting in the `StateEstimationMethod` property.

`[statePred, stateCov] = predict(pf, varargin)` passes all additional arguments specified in `varargin` to the underlying `StateTransitionFcn` property of `pf`. The first input to `StateTransitionFcn` is the set of particles from the previous time step, followed by all arguments in `varargin`.

## Input Arguments

**pf** — ParticleFilter object

handle

ParticleFilter object, specified as a handle. See `robotics.ParticleFilter` for more information.

**varargin** — Variable-length input argument list

comma-separated list

Variable-length input argument list, specified as a comma-separated list. This input is passed directly into the `StateTransitionFcn` property of `pf` to evolve the system state for each particle. When you call:

```
predict(pf, arg1, arg2)
```

MATLAB essentially calls the `stateTransitionFcn` as:

```
stateTransitionFcn(pf, prevParticles, arg1, arg2)
```

## Output Arguments

### **statePred** — Predicted system state

vector

Predicted system state, returned as a vector with length `NumStateVariables`. The predicted state is calculated based on the `StateEstimationMethod` algorithm.

### **stateCov** — Corrected system covariance

$N$ -by- $N$  matrix | []

Corrected system variance, returned as an  $N$ -by- $N$  matrix, where  $N$  is the value of `NumStateVariables` property from `pf`. The corrected state is calculated based on the `StateEstimationMethod` algorithm and the `MeasurementLikelihoodFcn`. If you specify a state estimate method that does not support covariance, then the function returns `stateCov` as [].

## Examples

### **Particle Filter Prediction and Correction**

Create a `ParticleFilter` object, and execute a prediction and correction step for state estimation. The particle filter gives a predicted state estimate based on the return value of `StateTransitionFcn`. It then corrects the state based on a given measurement and the return value of `MeasurementLikelihoodFcn`.

Create a particle filter with the default three states.

```
pf = robotics.ParticleFilter
```

```

pf =
  ParticleFilter with properties:

      NumStateVariables: 3
      NumParticles: 1000
      StateTransitionFcn: @robotics.algs.gaussianMotion
      MeasurementLikelihoodFcn: @robotics.algs.fullStateMeasurement
      IsStateVariableCircular: [0 0 0]
      ResamplingPolicy: [1x1 robotics.ResamplingPolicy]
      ResamplingMethod: 'multinomial'
      StateEstimationMethod: 'mean'
      StateOrientation: 'row'
      Particles: [1000x3 double]
      Weights: [1000x1 double]
      State: 'Use the getStateEstimate function to see the value.'
      StateCovariance: 'Use the getStateEstimate function to see the value.'

```

Specify the mean state estimation method and systematic resampling method.

```

pf.StateEstimationMethod = 'mean';
pf.ResamplingMethod = 'systematic';

```

Initialize the particle filter at state [4 1 9] with unit covariance (eye(3)). Use 5000 particles.

```

initialize(pf,5000,[4 1 9],eye(3));

```

Assuming a measurement [4.2 0.9 9], run one predict and one correct step.

```

[statePredicted,stateCov] = predict(pf);
[stateCorrected,stateCov] = correct(pf,[4.2 0.9 9]);

```

Get the best state estimate based on the StateEstimationMethod algorithm.

```

stateEst = getStateEstimate(pf)

```

```

stateEst = 1x3

```

```

    4.1562    0.9185    9.0202

```

## See Also

robotics.ParticleFilter | robotics.ParticleFilter.correct |  
robotics.ParticleFilter.getStateEstimate |  
robotics.ParticleFilter.initialize

## Topics

“Track a Car-Like Robot Using Particle Filter”  
“Particle Filter Parameters”  
“Particle Filter Workflow”

**Introduced in R2016a**



# findpath

**Class:** robotics.PRM

**Package:** robotics

Find path between start and goal points on roadmap

## Syntax

```
xy = findpath(prm,start,goal)
```

## Description

`xy = findpath(prm,start,goal)` finds an obstacle-free path between `start` and `goal` locations within `prm`, a roadmap object that contains a network of connected points.

If any properties of `prm` change, or if the roadmap is not created, `update` is called.

## Input Arguments

**prm** — Roadmap path planner

PRM object

Roadmap path planner, specified as a `robotics.PRM` object.

**start** — Start location of path

1-by-2 vector

Start location of path, specified as a 1-by-2 vector representing an `[x y]` pair.

Example: `[0 0]`

**goal** — Final location of path

1-by-2 vector

Final location of path, specified as a 1-by-2 vector vector representing an `[x y]` pair.

Example: [10 10]

## Output Arguments

### **xy** — Waypoints for a path between start and goal

*n*-by-2 column vector

Waypoints for a path between start and goal, specified as a *n*-by-2 column vector of [x y] pairs, where *n* is the number of waypoints. These pairs represent the solved path from the start and goal locations, given the roadmap from the `prm` input object.

## See Also

`robotics.PRM` | `robotics.PRM.show` | `robotics.PRM.update`

**Introduced in R2015a**

---

# show

**Class:** robotics.PRM

**Package:** robotics

Show map, roadmap, and path

## Syntax

```
show(prm)
show(prm,Name,Value)
```

## Description

`show(prm)` shows the map and the roadmap, specified as `prm` in a figure window. If no roadmap exists, `update` is called. If a path is computed before calling `show`, the path is also plotted on the figure.

`show(prm,Name,Value)` sets the specified `Value` to the property `Name`.

## Input Arguments

**prm — Roadmap path planner**

PRM object

Roadmap path planner, specified as a `robotics.PRM` 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`.

### **Parent — Axes to plot the map**

Axes object | UIAxes object

Axes to plot the map specified as a comma-separated pair consisting of "Parent" and either an Axes or UIAxes object. See `axes` or `uiaxes`.

### **Map — Map display option**

"on" (default) | "off"

Map display option, specified as the comma-separated pair consisting of "Map" and either "on" or "off".

### **Roadmap — Roadmap display option**

"on" (default) | "off"

Roadmap display option, specified as the comma-separated pair consisting of "Roadmap" and either "on" or "off".

### **Path — Path display option**

"on" (default) | "off"

Path display option, specified as "on" or "off". This controls whether the computed path is shown in the plot.

## **See Also**

`robotics.PRM` | `robotics.PRM.findpath` | `robotics.PRM.update`

## **Topics**

"Path Following for a Differential Drive Robot"

## **Introduced in R2015a**

# update

**Class:** robotics.PRM

**Package:** robotics

Create or update roadmap

## Syntax

```
update(prm)
```

## Description

`update(prm)` creates a roadmap if called for the first time after creating the PRM object, `prm`. Subsequent calls of `update` recreate the roadmap by resampling the map. `update` creates the new roadmap using the `Map`, `NumNodes`, and `ConnectionDistance` property values specified in `prm`.

## Input Arguments

**prm — Roadmap path planner**

PRM object

Roadmap path planner, specified as a `robotics.PRM` object.

## See Also

`robotics.PRM` | `robotics.PRM.findpath` | `robotics.PRM.show`

**Introduced in R2015a**

## addVisual

**Class:** robotics.RigidBody

**Package:** robotics

Add visual geometry data to rigid body

### Syntax

```
addVisual(body, "Mesh", filename)
addVisual(body, "Mesh", filename, tform)
```

### Description

`addVisual(body, "Mesh", filename)` adds a polygon mesh on top of any current visual geometry using the specified `.stl` file, `filename`. Multiple visual geometries can be added to a single body. The coordinate frame is assumed to coincide with the frame of `body`. You can view the meshes for an entire rigid body tree using `robotics.RigidBodyTree.show`.

`addVisual(body, "Mesh", filename, tform)` specifies a homogeneous transformation for the polygon mesh relative to the body frame.

### Input Arguments

**body** — RigidBody object

handle

RigidBody object, specified as a handle. Create a rigid body object using `robotics.RigidBody`.

**filename** — .stl file name

string scalar | character vector

.stl file name, specified as a string scalar or character vector.

Data Types: `char` | `string`

### **tform — Polygon mesh transformation**

4-by-4 homogeneous transformation

Mesh transformation relative to the body coordinate frame, specified as a 4-by-4 homogeneous transformation.

### **See Also**

`robotics.RigidBody.clearVisual` | `robotics.RigidBodyTree` |  
`robotics.RigidBodyTree.show`

**Introduced in R2017b**

## clearVisual

**Class:** robotics.RigidBody

**Package:** robotics

Clear all visual geometries

## Syntax

```
clearVisual(body)
```

## Description

`clearVisual(body)` clears all visual geometries attached to the given rigid body object.

## Input Arguments

**body — RigidBody object**

*handle*

RigidBody object, specified as a handle. Create a rigid body object using `robotics.RigidBody`.

## See Also

`robotics.RigidBody.addVisual` | `robotics.RigidBodyTree` |  
`robotics.RigidBodyTree.show`

**Introduced in R2017b**



## copy

**Class:** robotics.RigidBody

**Package:** robotics

Create a deep copy of rigid body

## Syntax

```
copyObj = copy(bodyObj)
```

## Description

`copyObj = copy(bodyObj)` creates a copy of the rigid body object with the same properties.

## Input Arguments

**bodyObj** — RigidBody object

handle

RigidBody object, specified as a handle. Create a rigid body object using `robotics.RigidBody`.

## Output Arguments

**copyObj** — RigidBody object

handle

RigidBody object, returned as a handle. Create a rigid body object using `robotics.RigidBody`.

## **See Also**

`robotics.Joint` | `robotics.RigidBodyTree`

**Introduced in R2016b**

# addBody

**Class:** robotics.RigidBodyTree

**Package:** robotics

Add body to robot

## Syntax

```
addBody( robot , body , parentname )
```

## Description

`addBody( robot , body , parentname )` adds a rigid body to the robot object and is attached to the rigid body parent specified by `parentname`. The `body.Joint` property defines how this body moves relative to the parent body.

## Input Arguments

### **robot — Robot model**

RigidBodyTree object

Robot model, specified as a RigidBodyTree object.

### **body — Rigid body**

RigidBody object

Rigid body, specified as a RigidBody object.

### **parentname — Parent body name**

string scalar | character vector

Parent body name, specified as a string scalar or character vector. This parent body must already exist in the robot model. The new body is attached to this parent body.

Data Types: char | string

## Examples

### Attach Rigid Body and Joint to Rigid Body Tree

Add a rigid body and corresponding joint to a rigid body tree. Each `RigidBody` object contains a `Joint` object and must be added to the `RigidBodyTree` using `addBody`.

Create a rigid body tree.

```
rbtree = robotics.RigidBodyTree;
```

Create a rigid body with a unique name.

```
body1 = robotics.RigidBody('b1');
```

Create a revolute joint. By default, the `RigidBody` object comes with a fixed joint. Replace the joint by assigning a new `Joint` object to the `body1.Joint` property.

```
jnt1 = robotics.Joint('jnt1','revolute');  
body1.Joint = jnt1;
```

Add the rigid body to the tree. Specify the body name that you are attaching the rigid body to. Because this is the first body, use the base name of the tree.

```
basename = rbtree.BaseName;  
addBody(rbtree,body1,basename)
```

Use `showdetails` on the tree to confirm the rigid body and joint were added properly.

```
showdetails(rbtree)
```

```
-----  
Robot: (1 bodies)
```

Idx	Body Name	Joint Name	Joint Type	Parent Name(Idx)	Children Name(s)
1	b1	jnt1	revolute	base(0)	

```
-----
```

## Build Manipulator Robot Using Denavit-Hartenberg Parameters

Use the Denavit-Hartenberg (DH) parameters of the Puma560® robot to build a robot. Each rigid body is added one at a time, with the child-to-parent transform specified by the joint object.

The DH parameters define the geometry of the robot with relation to how each rigid body is attached to its parent. For convenience, setup the parameters for the Puma560 robot in a matrix. The Puma robot is a serial chain manipulator. The DH parameters are relative to the previous line in the matrix, corresponding to the previous joint attachment.

```
dhparams = [0      pi/2    0      0;
            0.4318  0      0      0
            0.0203  -pi/2   0.15005  0;
            0      pi/2    0.4318  0;
            0      -pi/2   0      0;
            0      0      0      0];
```

Create a rigid body tree object to build the robot.

```
robot = robotics.RigidBodyTree;
```

Create the first rigid body and add it to the robot. To add a rigid body:

- 1 Create a `RigidBody` object and give it a unique name.
- 2 Create a `Joint` object and give it a unique name.
- 3 Use `setFixedTransform` to specify the body-to-body transformation using DH parameters. The last element of the DH parameters, `theta`, is ignored because the angle is dependent on the joint position.
- 4 Call `addBody` to attach the first body joint to the base frame of the robot.

```
body1 = robotics.RigidBody('body1');
jnt1 = robotics.Joint('jnt1','revolute');

setFixedTransform(jnt1,dhparams(1,:), 'dh');
body1.Joint = jnt1;

addBody(robot,body1,'base')
```

Create and add other rigid bodies to the robot. Specify the previous body name when calling `addBody` to attach it. Each fixed transform is relative to the previous joint coordinate frame.

```

body2 = robotics.RigidBody('body2');
jnt2 = robotics.Joint('jnt2','revolute');
body3 = robotics.RigidBody('body3');
jnt3 = robotics.Joint('jnt3','revolute');
body4 = robotics.RigidBody('body4');
jnt4 = robotics.Joint('jnt4','revolute');
body5 = robotics.RigidBody('body5');
jnt5 = robotics.Joint('jnt5','revolute');
body6 = robotics.RigidBody('body6');
jnt6 = robotics.Joint('jnt6','revolute');

setFixedTransform(jnt2,dhparams(2,:), 'dh');
setFixedTransform(jnt3,dhparams(3,:), 'dh');
setFixedTransform(jnt4,dhparams(4,:), 'dh');
setFixedTransform(jnt5,dhparams(5,:), 'dh');
setFixedTransform(jnt6,dhparams(6,:), 'dh');

body2.Joint = jnt2;
body3.Joint = jnt3;
body4.Joint = jnt4;
body5.Joint = jnt5;
body6.Joint = jnt6;

addBody(robot,body2,'body1')
addBody(robot,body3,'body2')
addBody(robot,body4,'body3')
addBody(robot,body5,'body4')
addBody(robot,body6,'body5')

```

Verify that your robot was built properly by using the `showdetails` or `show` function. `showdetails` lists all the bodies in the MATLAB® command window. `show` displays the robot with a given configuration (home by default). Calls to `axis` modify the axis limits and hide the axis labels.

```
showdetails(robot)
```

```

-----
Robot: (6 bodies)

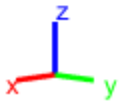
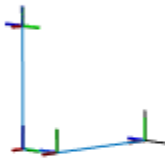
```

Idx	Body Name	Joint Name	Joint Type	Parent Name(Idx)	Children Name(s)
1	body1	jnt1	revolute	base(0)	body2(2)
2	body2	jnt2	revolute	body1(1)	body3(3)
3	body3	jnt3	revolute	body2(2)	body4(4)
4	body4	jnt4	revolute	body3(3)	body5(5)

```
5      body5      jnt5      revolute      body4(4)  body6(6)
6      body6      jnt6      revolute      body5(5)
```

-----

```
show(robot);
axis([-0.5,0.5,-0.5,0.5,-0.5,0.5])
axis off
```



### Modify a Robot Rigid Body Tree Model

Make changes to an existing `RigidBodyTree` object. You can get replace joints, bodies and subtrees in the rigid body tree.

Load example robots as `RigidBodyTree` objects.

```
load exampleRobots.mat
```

View the details of the Puma robot using `showdetails`.

```
showdetails(puma1)
```

```
-----  
Robot: (6 bodies)  
  
  Idx   Body Name   Joint Name   Joint Type   Parent Name(Idx)   Children Name(s)  
-----  
    1      L1       jnt1    revolute      base(0)      L2(2)  
    2      L2       jnt2    revolute      L1(1)      L3(3)  
    3      L3       jnt3    revolute      L2(2)      L4(4)  
    4      L4       jnt4    revolute      L3(3)      L5(5)  
    5      L5       jnt5    revolute      L4(4)      L6(6)  
    6      L6       jnt6    revolute      L5(5)  
-----
```

Get a specific body to inspect the properties. The only child of the L3 body is the L4 body. You can copy a specific body as well.

```
body3 = getBody(puma1, 'L3');  
childBody = body3.Children{1}
```

```
childBody =  
  RigidBody with properties:  
  
      Name: 'L4'  
      Joint: [1x1 robotics.Joint]  
      Mass: 1  
  CenterOfMass: [0 0 0]  
      Inertia: [1 1 1 0 0 0]  
      Parent: [1x1 robotics.RigidBody]  
  Children: {[1x1 robotics.RigidBody]}  
  Visuals: {}
```

```
body3Copy = copy(body3);
```



Replace the joint on the L3 body. You must create a new `Joint` object and use `replaceJoint` to ensure the downstream body geometry is unaffected. Call `setFixedTransform` if necessary to define a transform between the bodies instead of with the default identity matrices.

```
newJoint = robotics.Joint('prismatic');
replaceJoint(puma1, 'L3', newJoint);
```

```
showdetails(puma1)
```

```
-----
Robot: (6 bodies)
```

Idx	Body Name	Joint Name	Joint Type	Parent Name(Idx)	Children Name
1	L1	jnt1	revolute	base(0)	L2(2)
2	L2	jnt2	revolute	L1(1)	L3(3)
3	L3	prismatic	fixed	L2(2)	L4(4)
4	L4	jnt4	revolute	L3(3)	L5(5)
5	L5	jnt5	revolute	L4(4)	L6(6)
6	L6	jnt6	revolute	L5(5)	

Remove an entire body and get the resulting subtree using `removeBody`. The removed body is included in the subtree.

```
subtree = removeBody(puma1, 'L4')
```

```
subtree =
```

```
RigidBodyTree with properties:
```

```
  NumBodies: 3
  Bodies: {1x3 cell}
  Base: [1x1 robotics.RigidBody]
  BodyNames: {'L4' 'L5' 'L6'}
  BaseName: 'L3'
  Gravity: [0 0 0]
  DataFormat: 'struct'
```

Remove the modified L3 body. Add the original copied L3 body to the L2 body, followed by the returned subtree. The robot model remains the same. See a detailed comparison through `showdetails`.

```
removeBody(puma1, 'L3');  
addBody(puma1, body3Copy, 'L2')  
addSubtree(puma1, 'L3', subtree)
```

```
showdetails(puma1)
```

```
-----  
Robot: (6 bodies)
```

Idx	Body Name	Joint Name	Joint Type	Parent Name(Idx)	Children Name(s)
---	---	---	---	---	-----
1	L1	jnt1	revolute	base(0)	L2(2)
2	L2	jnt2	revolute	L1(1)	L3(3)
3	L3	jnt3	revolute	L2(2)	L4(4)
4	L4	jnt4	revolute	L3(3)	L5(5)
5	L5	jnt5	revolute	L4(4)	L6(6)
6	L6	jnt6	revolute	L5(5)	

```
-----
```

## See Also

[robotics.Joint](#) | [robotics.RigidBody](#) | [robotics.RigidBodyTree.removeBody](#)  
| [robotics.RigidBodyTree.replaceBody](#)

**Introduced in R2016b**

# addSubtree

**Class:** robotics.RigidBodyTree

**Package:** robotics

Add subtree to robot

## Syntax

```
addSubtree(robot, parentname, subtree)
```

## Description

`addSubtree(robot, parentname, subtree)` attaches the robot model, `subtree`, to an existing robot model, `robot`, at the body specified by `parentname`. The subtree base is not added as a body.

## Input Arguments

### **robot — Robot model**

RigidBodyTree object

Robot model, specified as a RigidBodyTree object.

### **parentname — Parent body name**

string scalar | character vector

Parent body name, specified as a string scalar or character vector. This parent body must already exist in the robot model. The new body is attached to this parent body.

Data Types: char | string

### **subtree — Subtree robot model**

RigidBodyTree object

Subtree robot model, specified as a RigidBodyTree object.

## Examples

### Modify a Robot Rigid Body Tree Model

Make changes to an existing RigidBodyTree object. You can get replace joints, bodies and subtrees in the rigid body tree.

Load example robots as RigidBodyTree objects.

```
load exampleRobots.mat
```

View the details of the Puma robot using showdetails.

```
showdetails(puma1)
```

```
-----  
Robot: (6 bodies)
```

Idx	Body Name	Joint Name	Joint Type	Parent Name(Idx)	Children Name(s)
1	L1	jnt1	revolute	base(0)	L2(2)
2	L2	jnt2	revolute	L1(1)	L3(3)
3	L3	jnt3	revolute	L2(2)	L4(4)
4	L4	jnt4	revolute	L3(3)	L5(5)
5	L5	jnt5	revolute	L4(4)	L6(6)
6	L6	jnt6	revolute	L5(5)	

```
-----  
Get a specific body to inspect the properties. The only child of the L3 body is the L4 body.  
You can copy a specific body as well.
```

```
body3 = getBody(puma1, 'L3');  
childBody = body3.Children{1}
```

```
childBody =
```

```
  RigidBody with properties:
```

```
      Name: 'L4'  
      Joint: [1x1 robotics.Joint]  
      Mass: 1  
CenterOfMass: [0 0 0]  
      Inertia: [1 1 1 0 0 0]  
      Parent: [1x1 robotics.RigidBody]
```

```
Children: {[1x1 robotics.RigidBody]}
Visuals: {}
```

```
body3Copy = copy(body3);
```

Replace the joint on the L3 body. You must create a new `Joint` object and use `replaceJoint` to ensure the downstream body geometry is unaffected. Call `setFixedTransform` if necessary to define a transform between the bodies instead of with the default identity matrices.

```
newJoint = robotics.Joint('prismatic');
replaceJoint(puma1, 'L3', newJoint);
```

```
showdetails(puma1)
```

```
-----
Robot: (6 bodies)
```

Idx	Body Name	Joint Name	Joint Type	Parent Name(Idx)	Children Name
1	L1	jnt1	revolute	base(0)	L2(2)
2	L2	jnt2	revolute	L1(1)	L3(3)
3	L3	prismatic	fixed	L2(2)	L4(4)
4	L4	jnt4	revolute	L3(3)	L5(5)
5	L5	jnt5	revolute	L4(4)	L6(6)
6	L6	jnt6	revolute	L5(5)	

Remove an entire body and get the resulting subtree using `removeBody`. The removed body is included in the subtree.

```
subtree = removeBody(puma1, 'L4')
```

```
subtree =
  RigidBodyTree with properties:
    NumBodies: 3
    Bodies: {1x3 cell}
    Base: [1x1 robotics.RigidBody]
    BodyNames: {'L4' 'L5' 'L6'}
    BaseName: 'L3'
    Gravity: [0 0 0]
    DataFormat: 'struct'
```

Remove the modified L3 body. Add the original copied L3 body to the L2 body, followed by the returned subtree. The robot model remains the same. See a detailed comparison through `showdetails`.

```
removeBody(puma1, 'L3');  
addBody(puma1, body3Copy, 'L2')  
addSubtree(puma1, 'L3', subtree)
```

```
showdetails(puma1)
```

```
-----  
Robot: (6 bodies)
```

Idx	Body Name	Joint Name	Joint Type	Parent Name(Idx)	Children Name(s)
---	-----	-----	-----	-----	-----
1	L1	jnt1	revolute	base(0)	L2(2)
2	L2	jnt2	revolute	L1(1)	L3(3)
3	L3	jnt3	revolute	L2(2)	L4(4)
4	L4	jnt4	revolute	L3(3)	L5(5)
5	L5	jnt5	revolute	L4(4)	L6(6)
6	L6	jnt6	revolute	L5(5)	
-----	-----	-----	-----	-----	-----

## See Also

`robotics.Joint` | `robotics.RigidBody` | `robotics.RigidBodyTree.addBody` |  
`robotics.RigidBodyTree.removeBody` | `robotics.RigidBodyTree.replaceBody`

**Introduced in R2016b**

# centerOfMass

**Class:** robotics.RigidBodyTree

**Package:** robotics

Center of mass position and Jacobian

## Syntax

```
com = centerOfMass(robot)
com = centerOfMass(robot, configuration)
[com, comJac] = centerOfMass(robot, configuration)
```

## Description

`com = centerOfMass(robot)` computes the center of mass position of the robot model at its home configuration, relative to the base frame.

`com = centerOfMass(robot, configuration)` computes the center of mass position of the robot model at the specified joint configuration, relative to the base frame.

`[com, comJac] = centerOfMass(robot, configuration)` also returns the center of mass Jacobian, which relates the center of mass velocity to the joint velocities.

## Input Arguments

### **robot — Robot model**

RigidBodyTree object

Robot model, specified as a RigidBodyTree object. To use the centerOfMass function, set the DataFormat property to either 'row' or 'column'.

### **configuration — Robot configuration**

vector

Robot configuration, specified as a vector with positions for all nonfixed joints in the robot model. You can generate a configuration using `homeConfiguration(robot)`,

`randomConfiguration(robot)`, or by specifying your own joint positions. To use the vector form of configuration, set the `DataFormat` property for the robot to either `'row'` or `'column'`.

## Output Arguments

### **com** — Center of mass location

[x y z] vector

Center of mass location, returned as an [x y z] vector. The vector describes the location of the center of mass for the specified configuration relative to the body frame, in meters.

### **comJac** — Center of mass Jacobian

3-by-*n* matrix

Center of mass Jacobian, returned as a 3-by-*n* matrix, where *n* is the robot velocity degrees of freedom.

## Examples

### **Calculate Center of Mass and Jacobian for Robot Configuration**

Load a predefined KUKA LBR robot model, which is specified as a `RigidBodyTree` object.

```
load exampleRobots.mat lbr
```

Set the data format to `'row'`. For all dynamics calculations, the data format must be either `'row'` or `'column'`.

```
lbr.DataFormat = 'row';
```

Compute the center of mass position and Jacobian at the home configuration of the robot.

```
[comLocation,comJac] = centerOfMass(lbr);
```



## **See Also**

[RigidBodyTree](#) | [gravityTorque](#) | [massMatrix](#) | [velocityProduct](#)

**Introduced in R2017a**

# copy

**Class:** robotics.RigidBodyTree

**Package:** robotics

Copy robot model

## Syntax

```
newrobot = copy(robot)
```

## Description

`newrobot = copy(robot)` creates a deep copy of `robot` with the same properties. Any changes in `newrobot` are not reflected in `robot`.

## Input Arguments

**robot — Robot model**

RigidBodyTree object

Robot model, specified as a RigidBodyTree object.

## Output Arguments

**newrobot — Robot model**

RigidBodyTree object

Robot model, returned as a RigidBodyTree object.

## Examples

## Modify a Robot Rigid Body Tree Model

Make changes to an existing RigidBodyTree object. You can get replace joints, bodies and subtrees in the rigid body tree.

Load example robots as RigidBodyTree objects.

```
load exampleRobots.mat
```

View the details of the Puma robot using showdetails.

```
showdetails(puma1)
```

```
-----
Robot: (6 bodies)
```

Idx	Body Name	Joint Name	Joint Type	Parent Name(Idx)	Children Name(s)
1	L1	jnt1	revolute	base(0)	L2(2)
2	L2	jnt2	revolute	L1(1)	L3(3)
3	L3	jnt3	revolute	L2(2)	L4(4)
4	L4	jnt4	revolute	L3(3)	L5(5)
5	L5	jnt5	revolute	L4(4)	L6(6)
6	L6	jnt6	revolute	L5(5)	

```
-----
```

Get a specific body to inspect the properties. The only child of the L3 body is the L4 body. You can copy a specific body as well.

```
body3 = getBody(puma1, 'L3');
childBody = body3.Children{1}
```

```
childBody =
  RigidBody with properties:
      Name: 'L4'
      Joint: [1x1 robotics.Joint]
      Mass: 1
      CenterOfMass: [0 0 0]
      Inertia: [1 1 1 0 0 0]
      Parent: [1x1 robotics.RigidBody]
      Children: {[1x1 robotics.RigidBody]}
      Visuals: {}
```

```
body3Copy = copy(body3);
```

Replace the joint on the L3 body. You must create a new `Joint` object and use `replaceJoint` to ensure the downstream body geometry is unaffected. Call `setFixedTransform` if necessary to define a transform between the bodies instead of with the default identity matrices.

```
newJoint = robotics.Joint('prismatic');
replaceJoint(puma1, 'L3', newJoint);
```

```
showdetails(puma1)
```

```
-----
Robot: (6 bodies)
```

Idx	Body Name	Joint Name	Joint Type	Parent Name(Idx)	Children Name
---	-----	-----	-----	-----	-----
1	L1	jnt1	revolute	base(0)	L2(2)
2	L2	jnt2	revolute	L1(1)	L3(3)
3	L3	prismatic	fixed	L2(2)	L4(4)
4	L4	jnt4	revolute	L3(3)	L5(5)
5	L5	jnt5	revolute	L4(4)	L6(6)
6	L6	jnt6	revolute	L5(5)	

Remove an entire body and get the resulting subtree using `removeBody`. The removed body is included in the subtree.

```
subtree = removeBody(puma1, 'L4')
```

```
subtree =
```

```
RigidBodyTree with properties:
```

```
  NumBodies: 3
    Bodies: {1x3 cell}
      Base: [1x1 robotics.RigidBody]
  BodyNames: {'L4' 'L5' 'L6'}
  BaseName: 'L3'
  Gravity: [0 0 0]
  DataFormat: 'struct'
```

Remove the modified L3 body. Add the original copied L3 body to the L2 body, followed by the returned subtree. The robot model remains the same. See a detailed comparison through `showdetails`.

```
removeBody(puma1, 'L3');
addBody(puma1, body3Copy, 'L2')
addSubtree(puma1, 'L3', subtree)
```

```
showdetails(puma1)
```

```
-----
Robot: (6 bodies)
```

Idx	Body Name	Joint Name	Joint Type	Parent Name(Idx)	Children Name(s)
1	L1	jnt1	revolute	base(0)	L2(2)
2	L2	jnt2	revolute	L1(1)	L3(3)
3	L3	jnt3	revolute	L2(2)	L4(4)
4	L4	jnt4	revolute	L3(3)	L5(5)
5	L5	jnt5	revolute	L4(4)	L6(6)
6	L6	jnt6	revolute	L5(5)	

## See Also

[robotics.Joint](#) | [robotics.RigidBody](#) | [robotics.RigidBodyTree](#)

**Introduced in R2016b**

## externalForce

**Class:** robotics.RigidBodyTree

**Package:** robotics

Compose external force matrix relative to base

### Syntax

```
fext = externalForce(robot,bodyname,wrench)
```

```
fext = externalForce(robot,bodyname,wrench,configuration)
```

### Description

`fext = externalForce(robot,bodyname,wrench)` composes the external force matrix, which you can use as inputs to `inverseDynamics` and `forwardDynamics` to apply an external force, `wrench`, to the body specified by `bodyname`. The `wrench` input is assumed to be in the base frame.

`fext = externalForce(robot,bodyname,wrench,configuration)` composes the external force matrix assuming that `wrench` is in the `bodyname` frame for the specified configuration. The force matrix `fext` is given in the base frame.

### Input Arguments

**robot — Robot model**

RigidBodyTree object

Robot model, specified as a RigidBodyTree object. To use the externalForce function, set the DataFormat property to either "row" or "column".

**bodyname — Name of body to which external force is applied**

string scalar | character vector

Name of body to which the external force is applied, specified as a string scalar or character vector. This body name must match a body on the robot object.

Data Types: char | string

### **wrench — Torques and forces applied to body**

[Tx Ty Tz Fx Fy Fz] vector

Torques and forces applied to the body, specified as a [Tx Ty Tz Fx Fy Fz] vector. The first three elements of the wrench correspond to the moments around xyz-axes. The last three elements are linear forces along the same axes. Unless you specify the robot configuration, the wrench is assumed to be relative to the base frame.

### **configuration — Robot configuration**

vector

Robot configuration, specified as a vector with positions for all nonfixed joints in the robot model. You can generate a configuration using `homeConfiguration(robot)`, `randomConfiguration(robot)`, or by specifying your own joint positions. To use the vector form of configuration, set the `DataFormat` property for the robot to either "row" or "column" .

## **Output Arguments**

### **fext — External force matrix**

*n*-by-6 matrix | 6-by-*n* matrix

External force matrix, returned as either an *n*-by-6 or 6-by-*n* matrix, where *n* is the velocity number (degrees of freedom) of the robot. The shape depends on the `DataFormat` property of robot. The "row" data format uses an *n*-by-6 matrix. The "column" data format uses a 6-by-*n* .

The composed matrix lists only values other than zero at the locations relevant to the body specified. You can add force matrices together to specify multiple forces on multiple bodies. Use the external force matrix to specify external forces to dynamics functions `inverseDynamics` and `forwardDynamics`.

## **Examples**

### Compute Forward Dynamics Due to External Forces on Rigid Body Tree Model

Calculate the resultant joint accelerations for a given robot configuration with applied external forces and forces due to gravity. A wrench is applied to a specific body with the gravity being specified for the whole robot.

Load a predefined KUKA LBR robot model, which is specified as a `RigidBodyTree` object.

```
load exampleRobots.mat lbr
```

Set the data format to 'row'. For all dynamics calculations, the data format must be either 'row' or 'column'.

```
lbr.DataFormat = 'row';
```

Set the gravity. By default, gravity is assumed to be zero.

```
lbr.Gravity = [0 0 -9.81];
```

Get the home configuration for the `lbr` robot.

```
q = homeConfiguration(lbr);
```

Specify the wrench vector that represents the external forces experienced by the robot. Use the `externalForce` function to generate the external force matrix. Specify the robot model, the end effector that experiences the wrench, the wrench vector, and the current robot configuration. `wrench` is given relative to the 'tool0' body frame, which requires you to specify the robot configuration, `q`.

```
wrench = [0 0 0.5 0 0 0.3];  
fext = externalForce(lbr, 'tool0', wrench, q);
```

Compute the resultant joint accelerations due to gravity, with the external force applied to the end-effector 'tool0' when `lbr` is at its home configuration. The joint velocities and joint torques are assumed to be zero (input as an empty vector []).

```
qddot = forwardDynamics(lbr, q, [], [], fext);
```

### Compute Joint Torque to Counter External Forces

Use the `externalForce` function to generate force matrices to apply to a rigid body tree model. The force matrix is an  $m$ -by-6 vector that has a row for each joint on the robot to



apply a six-element wrench. Use the `externalForce` function and specify the end effector to properly assign the wrench to the correct row of the matrix. You can add multiple force matrices together to apply multiple forces to one robot.

To calculate the joint torques that counter these external forces, use the `inverseDynamics` function.

Load a predefined KUKA LBR robot model, which is specified as a `RigidBodyTree` object.

```
load exampleRobots.mat lbr
```

Set the data format to 'row'. For all dynamics calculations, the data format must be either 'row' or 'column'.

```
lbr.DataFormat = 'row';
```

Set the `Gravity` property to give a specific gravitational acceleration.

```
lbr.Gravity = [0 0 -9.81];
```

Get the home configuration for `lbr`.

```
q = homeConfiguration(lbr);
```

Set external force on `link1`. The input wrench vector is expressed in the base frame.

```
fext1 = externalForce(lbr, 'link_1', [0 0 0.0 0.1 0 0]);
```

Set external force on the end effector, `tool0`. The input wrench vector is expressed in the `tool0` frame.

```
fext2 = externalForce(lbr, 'tool0', [0 0 0.0 0.1 0 0], q);
```

Compute the joint torques required to balance the external forces. To combine the forces, add the force matrices together. Joint velocities and accelerations are assumed to be zero (input as []).

```
tau = inverseDynamics(lbr, q, [], [], fext1+fext2);
```

## See Also

`RigidBodyTree` | `forwardDynamics` | `inverseDynamics`

## **Topics**

“Control LBR Manipulator Motion Through Joint Torque Commands”

**Introduced in R2017a**

# forwardDynamics

**Class:** robotics.RigidBodyTree

**Package:** robotics

Joint accelerations given joint torques and states

## Syntax

```
jointAccel = forwardDynamics(robot)
jointAccel = forwardDynamics(robot, configuration)
jointAccel = forwardDynamics(robot, configuration, jointVel)
jointAccel = forwardDynamics(robot, configuration, jointVel, jointTorq)
jointAccel = forwardDynamics(robot, configuration, jointVel, jointTorq,
fext)
```

## Description

`jointAccel = forwardDynamics(robot)` computes joint accelerations due to gravity at the robot home configuration, with zero joint velocities and no external forces.

`jointAccel = forwardDynamics(robot, configuration)` also specifies the joint positions of the robot configuration.

`jointAccel = forwardDynamics(robot, configuration, jointVel)` also specifies the joint velocities of the robot.

`jointAccel = forwardDynamics(robot, configuration, jointVel, jointTorq)` also specifies the joint torques applied to the robot.

`jointAccel = forwardDynamics(robot, configuration, jointVel, jointTorq, fext)` also specifies an external force matrix that contains forces applied to each joint.

To specify the home configuration, zero joint velocities, or zero torques, use `[]` for that input argument.

## Input Arguments

### **robot — Robot model**

RigidBodyTree object

Robot model, specified as a RigidBodyTree object. To use the `forwardDynamics` function, set the `DataFormat` property to either `'row'` or `'column'`.

### **configuration — Robot configuration**

vector

Robot configuration, specified as a vector with positions for all nonfixed joints in the robot model. You can generate a configuration using `homeConfiguration(robot)`, `randomConfiguration(robot)`, or by specifying your own joint positions. To use the vector form of configuration, set the `DataFormat` property for the robot to either `'row'` or `'column'`.

### **jointVel — Joint velocities**

vector

Joint velocities, specified as a vector. The number of joint velocities is equal to the degrees of freedom of the robot. To use the vector form of `jointVel`, set the `DataFormat` property for the robot to either `'row'` or `'column'`.

### **jointTorq — Joint torques**

vector

Joint torques, specified as a vector. Each element corresponds to a torque applied to a specific joint. To use the vector form of `jointTorq`, set the `DataFormat` property for the robot to either `'row'` or `'column'`.

### **fext — External force matrix**

$n$ -by-6 matrix | 6-by- $n$  matrix

External force matrix, specified as either an  $n$ -by-6 or 6-by- $n$  matrix, where  $n$  is the number of bodies of the robot. The shape depends on the `DataFormat` property of `robot`. The `'row'` data format uses an  $n$ -by-6 matrix. The `'column'` data format uses a 6-by- $n$ .

The matrix lists only values other than zero at the locations relevant to the body specified. You can add force matrices together to specify multiple forces on multiple bodies.

To create the matrix for a specified force or torque, see `externalForce`.

## Output Arguments

### **jointAccel** – Joint accelerations

vector

Joint accelerations, returned as a vector. The dimension of the joint accelerations vector is equal to the degrees of freedom of the robot. Each element corresponds to a specific joint on the robot.

## Examples

### **Compute Forward Dynamics Due to External Forces on Rigid Body Tree Model**

Calculate the resultant joint accelerations for a given robot configuration with applied external forces and forces due to gravity. A wrench is applied to a specific body with the gravity being specified for the whole robot.

Load a predefined KUKA LBR robot model, which is specified as a `RigidBodyTree` object.

```
load exampleRobots.mat lbr
```

Set the data format to 'row'. For all dynamics calculations, the data format must be either 'row' or 'column'.

```
lbr.DataFormat = 'row';
```

Set the gravity. By default, gravity is assumed to be zero.

```
lbr.Gravity = [0 0 -9.81];
```

Get the home configuration for the `lbr` robot.

```
q = homeConfiguration(lbr);
```

Specify the wrench vector that represents the external forces experienced by the robot. Use the `externalForce` function to generate the external force matrix. Specify the robot

model, the end effector that experiences the wrench, the wrench vector, and the current robot configuration. `wrench` is given relative to the `'tool0'` body frame, which requires you to specify the robot configuration, `q`.

```
wrench = [0 0 0.5 0 0 0.3];  
fext = externalForce(lbr, 'tool0', wrench, q);
```

Compute the resultant joint accelerations due to gravity, with the external force applied to the end-effector `'tool0'` when `lbr` is at its home configuration. The joint velocities and joint torques are assumed to be zero (input as an empty vector `[]`).

```
qddot = forwardDynamics(lbr, q, [], [], fext);
```

## See Also

[RigidBodyTree](#) | [externalForce](#) | [inverseDynamics](#)

## Topics

“Control LBR Manipulator Motion Through Joint Torque Commands”

**Introduced in R2017a**

# geometricJacobian

**Class:** robotics.RigidBodyTree

**Package:** robotics

Geometric Jacobian for robot configuration

## Syntax

```
jacobian = geometricJacobian(robot, configuration, endeffectorname)
```

## Description

```
jacobian = geometricJacobian(robot, configuration, endeffectorname)
```

computes the geometric Jacobian relative to the base for the specified end-effector name and configuration for the robot model.

## Input Arguments

**robot — Robot model**

RigidBodyTree object

Robot model, specified as a RigidBodyTree object.

**configuration — Robot configuration**

vector | structure

Robot configuration, specified as a vector of joint positions or a structure with joint names and positions for all the bodies in the robot model. You can generate a configuration using `homeConfiguration(robot)`, `randomConfiguration(robot)`, or by specifying your own joint positions in a structure. To use the vector form of `configuration`, set the `DataFormat` property for the robot to either "row" or "column" .

**endeffectorname — End-effector name**

string scalar | character vector

End-effector name, specified as a string scalar or character vector. An end effector can be any body in the robot model.

Data Types: `char` | `string`

## Output Arguments

### **jacobian** — Geometric Jacobian

6-by- $n$  matrix

Geometric Jacobian of the end effector with the specified `configuration`, returned as a 6-by- $n$  matrix, where  $n$  is the number of degrees of freedom for the end effector. The Jacobian maps the joint-space velocity to the end-effector velocity, relative to the base coordinate frame. The end-effector velocity equals:

$$V_{EE} = \begin{bmatrix} \omega_x \\ \omega_y \\ \omega_z \\ v_x \\ v_y \\ v_z \end{bmatrix} = J\dot{q} = J \begin{bmatrix} \dot{q}_1 \\ \vdots \\ \dot{q}_n \end{bmatrix}$$

$\omega$  is the angular velocity,  $v$  is the linear velocity, and  $\dot{q}$  is the joint-space velocity.

## Examples

### **Geometric Jacobian for Robot Configuration**

Calculate the geometric Jacobian for a specific end effector and configuration of a robot.

Load a Puma robot, which is specified as a `RigidBodyTree` object.

```
load exampleRobots.mat puma1
```

Calculate the geometric Jacobian of body 'L6' on the Puma robot for a random configuration.



```
geoJacob = geometricJacobian(puma1, randomConfiguration(puma1), 'L6')
```

```
geoJacob = 6×6
```

```
-0.0000    0.9826    0.9826    0.0286   -0.9155    0.2045  
-0.0000    0.1859    0.1859   -0.1512    0.3929    0.2690  
1.0000   -0.0000   -0.0000    0.9881    0.0866    0.9412  
0.4175    0.0530    0.0799    0.0000         0         0  
0.2317   -0.2802   -0.4223    0.0000         0         0  
         0   -0.4532   -0.0464    0.0000         0         0
```

## See Also

[Joint](#) | [RigidBody](#) | [getTransform](#) | [homeConfiguration](#) | [randomConfiguration](#)

**Introduced in R2016b**

## gravityTorque

**Class:** robotics.RigidBodyTree

**Package:** robotics

Joint torques that compensate gravity

### Syntax

```
gravTorq = gravityTorque(robot)
gravTorq = gravityTorque(robot, configuration)
```

### Description

`gravTorq = gravityTorque(robot)` computes the joint torques required to hold the robot at its home configuration.

`gravTorq = gravityTorque(robot, configuration)` specifies a joint configuration for calculating the gravity torque.

### Input Arguments

**robot — Robot model**

RigidBodyTree object

Robot model, specified as a RigidBodyTree object. To use the gravityTorque function, set the DataFormat property to either 'row' or 'column'.

**configuration — Robot configuration**

vector

Robot configuration, specified as a vector with positions for all nonfixed joints in the robot model. You can generate a configuration using homeConfiguration(robot), randomConfiguration(robot), or by specifying your own joint positions. To use the vector form of configuration, set the DataFormat property for the robot to either 'row' or 'column'.

## Output Arguments

**gravTorq** — Gravity-compensating torque for each joint  
vector

Gravity-compensating torque for each joint, returned as a vector.

## Examples

### Compute Gravity Torque for Robot Configuration

Load a predefined KUKA LBR robot model, which is specified as a `RigidBodyTree` object.

```
load exampleRobots.mat lbr
```

Set the data format to 'row'. For all dynamics calculations, the data format must be either 'row' or 'column'. Set the `Gravity` property.

```
lbr.DataFormat = 'row';  
lbr.Gravity = [0 0 -9.81];
```

Get a random configuration for `lbr`.

```
q = randomConfiguration(lbr);
```

Compute the gravity-compensating torques for each joint.

```
gtau = gravityTorque(lbr,q);
```

## See Also

`RigidBodyTree` | `inverseDynamics` | `velocityProduct`

## Topics

“Control LBR Manipulator Motion Through Joint Torque Commands”

**Introduced in R2017a**

# getBody

**Class:** robotics.RigidBodyTree

**Package:** robotics

Get robot body handle by name

## Syntax

```
body = getBody(robot, bodyname)
```

## Description

`body = getBody(robot, bodyname)` gets a body handle by name from the robot model.

## Input Arguments

**robot — Robot model**

RigidBodyTree object

Robot model, specified as a RigidBodyTree object.

**bodyname — Body name**

string scalar | character vector

Body name, specified as a string scalar or character vector. A body with this name must be on the robot model specified by `robot`.

Data Types: char | string

## Output Arguments

**body — Rigid body**

RigidBody object

Rigid body, returned as a `RigidBody` object. The returned `RigidBody` object is still a part of the `RigidBodyTree` robot model. Use `robotics.RigidBodyTree.replaceBody` with a new body to modify the body in the robot model.

## Examples

### Modify a Robot Rigid Body Tree Model

Make changes to an existing `RigidBodyTree` object. You can get replace joints, bodies and subtrees in the rigid body tree.

Load example robots as `RigidBodyTree` objects.

```
load exampleRobots.mat
```

View the details of the Puma robot using `showdetails`.

```
showdetails(puma1)
```

```
-----  
Robot: (6 bodies)
```

Idx	Body Name	Joint Name	Joint Type	Parent Name(Idx)	Children Name(s)
1	L1	jnt1	revolute	base(0)	L2(2)
2	L2	jnt2	revolute	L1(1)	L3(3)
3	L3	jnt3	revolute	L2(2)	L4(4)
4	L4	jnt4	revolute	L3(3)	L5(5)
5	L5	jnt5	revolute	L4(4)	L6(6)
6	L6	jnt6	revolute	L5(5)	

```
-----  
Get a specific body to inspect the properties. The only child of the L3 body is the L4 body. You can copy a specific body as well.
```

```
body3 = getBody(puma1, 'L3');  
childBody = body3.Children{1}
```

```
childBody =  
  RigidBody with properties:
```

```

    Name: 'L4'
    Joint: [1x1 robotics.Joint]
    Mass: 1
    CenterOfMass: [0 0 0]
    Inertia: [1 1 1 0 0 0]
    Parent: [1x1 robotics.RigidBody]
    Children: {[1x1 robotics.RigidBody]}
    Visuals: {}

```

```
body3Copy = copy(body3);
```

Replace the joint on the L3 body. You must create a new `Joint` object and use `replaceJoint` to ensure the downstream body geometry is unaffected. Call `setFixedTransform` if necessary to define a transform between the bodies instead of with the default identity matrices.

```
newJoint = robotics.Joint('prismatic');
replaceJoint(puma1, 'L3', newJoint);
```

```
showdetails(puma1)
```

```
-----
Robot: (6 bodies)
```

Idx	Body Name	Joint Name	Joint Type	Parent Name(Idx)	Children Name
1	L1	jnt1	revolute	base(0)	L2(2)
2	L2	jnt2	revolute	L1(1)	L3(3)
3	L3	prismatic	fixed	L2(2)	L4(4)
4	L4	jnt4	revolute	L3(3)	L5(5)
5	L5	jnt5	revolute	L4(4)	L6(6)
6	L6	jnt6	revolute	L5(5)	

Remove an entire body and get the resulting subtree using `removeBody`. The removed body is included in the subtree.

```
subtree = removeBody(puma1, 'L4')
```

```
subtree =
  RigidBodyTree with properties:
```

```

  NumBodies: 3
  Bodies: {1x3 cell}

```

```
Base: [1x1 robotics.RigidBody]
BodyNames: {'L4' 'L5' 'L6'}
BaseName: 'L3'
Gravity: [0 0 0]
DataFormat: 'struct'
```

Remove the modified L3 body. Add the original copied L3 body to the L2 body, followed by the returned subtree. The robot model remains the same. See a detailed comparison through `showdetails`.

```
removeBody(puma1, 'L3');
addBody(puma1, body3Copy, 'L2')
addSubtree(puma1, 'L3', subtree)
```

```
showdetails(puma1)
```

```
-----
Robot: (6 bodies)
```

Idx	Body Name	Joint Name	Joint Type	Parent Name(Idx)	Children Name(s)
1	L1	jnt1	revolute	base(0)	L2(2)
2	L2	jnt2	revolute	L1(1)	L3(3)
3	L3	jnt3	revolute	L2(2)	L4(4)
4	L4	jnt4	revolute	L3(3)	L5(5)
5	L5	jnt5	revolute	L4(4)	L6(6)
6	L6	jnt6	revolute	L5(5)	

## See Also

`robotics.Joint` | `robotics.RigidBody` | `robotics.RigidBodyTree.addBody` | `robotics.RigidBodyTree.replaceBody`

**Introduced in R2016b**



# getTransform

**Class:** robotics.RigidBodyTree

**Package:** robotics

Get transform between body frames

## Syntax

```
transform = getTransform(robot,configuration,bodyname)
```

```
transform = getTransform(robot,configuration,sourcebody,targetbody)
```

## Description

`transform = getTransform(robot,configuration,bodyname)` computes the transform that converts points in the `bodyname` frame to the robot base frame, using the specified robot configuration.

`transform = getTransform(robot,configuration,sourcebody,targetbody)` computes the transform that converts points from the source body frame to the target body frame, using the specified robot configuration.

## Input Arguments

**robot — Robot model**

RigidBodyTree object

Robot model, specified as a RigidBodyTree object.

**configuration — Robot configuration**

structure array

Robot configuration, specified as a structure array with joint names and positions for all the bodies in the robot model. You can generate a configuration using `homeConfiguration(robot)`, `randomConfiguration(robot)`, or by specifying your own joint names and positions in a structure array.

### **bodyname — Body name**

string scalar | character vector

Body name, specified as a string scalar or character vector. This body must be on the robot model specified in `robot`.

Data Types: `char` | `string`

### **targetbody — Target body name**

string scalar | character vector

Target body name, specified as a character vector. This body must be on the robot model specified in `robot`. The target frame is the coordinate system you want to transform points into.

Data Types: `char` | `string`

### **sourcebody — Body name**

string scalar | character vector

Body name, specified as a string scalar or character vector. This body must be on the robot model specified in `robot`. The source frame is the coordinate system you want points transformed from.

Data Types: `char` | `string`

## Output Arguments

### **ttransform — Homogeneous transform**

4-by-4 matrix

Homogeneous transform, returned as a 4-by-4 matrix.

## Examples

### **Get Transform Between Frames for Robot Configuration**

Get the transform between two frames for a specific robot configuration.

Load a sample robots that include the `puma1` robot.

```
load exampleRobots.mat
```

Get the transform between the 'L2' and 'L6' bodies of the puma1 robot given a specific configuration. The transform converts points in 'L6' frame to the 'L2' frame.

```
transform = getTransform(puma1, randomConfiguration(puma1), 'L2', 'L6')
```

```
transform = 4x4
```

```
-0.2232    0.4179    0.8807    0.0212  
-0.8191    0.4094   -0.4018    0.1503  
-0.5284   -0.8111    0.2509   -0.4317  
         0         0         0         1.0000
```

## See Also

[robotics.Joint](#) | [robotics.RigidBody](#) |  
[robotics.RigidBodyTree.geometricJacobian](#) |  
[robotics.RigidBodyTree.homeConfiguration](#) |  
[robotics.RigidBodyTree.randomConfiguration](#)

**Introduced in R2016b**

## homeConfiguration

**Class:** robotics.RigidBodyTree

**Package:** robotics

Get home configuration of robot

### Syntax

```
configuration = homeConfiguration(robot)
```

### Description

`configuration = homeConfiguration(robot)` returns the home configuration of the robot model. The home configuration is the ordered list of `HomePosition` properties of each nonfixed joint.

### Input Arguments

**robot — Robot model**

`RigidBodyTree` object

Robot model, specified as a `RigidBodyTree` object.

### Output Arguments

**configuration — Robot configuration**

vector | structure

Robot configuration, returned as a vector of joint positions or a structure with joint names and positions for all the bodies in the robot model. You can generate a configuration using `homeConfiguration(robot)`, `randomConfiguration(robot)`, or by specifying your own joint positions in a structure. To use the vector form of `configuration`, set the `DataFormat` property for the robot to either `'row'` or `'column'`.

## Examples

### Visualize Robot Configurations

Show different configurations of a robot created using a `RigidBodyTree` model. Use the `homeConfiguration` or `randomConfiguration` functions to generate the structure that defines all the joint positions.

Load example robots as `RigidBodyTree` objects.

```
load exampleRobots.mat
```

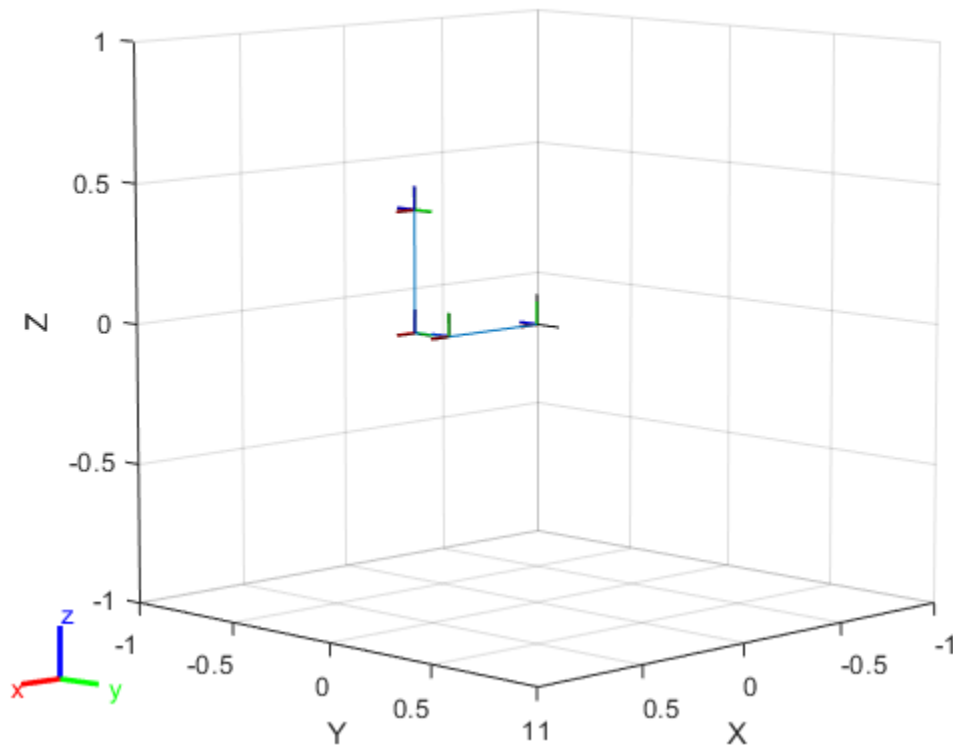
Create a structure for the home configuration of a Puma robot. The structure has joint names and positions for each body on the robot model.

```
config = homeConfiguration(puma1)

config = 1x6 struct array with fields:
    JointName
    JointPosition
```

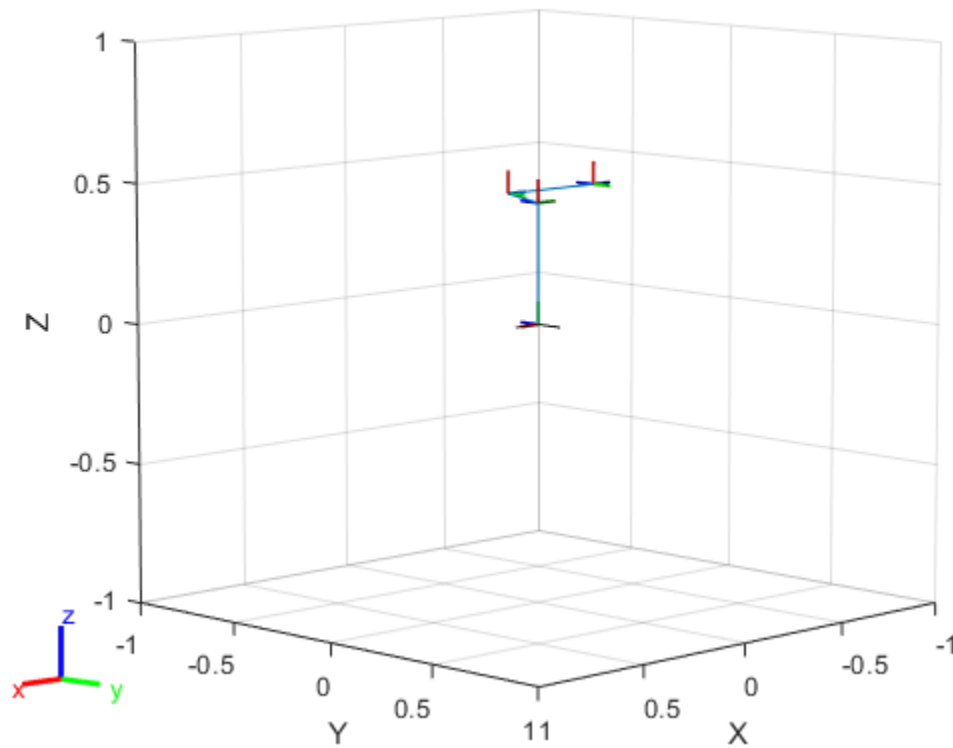
Show the home configuration using `show`. You do not need to specify a configuration input.

```
show(puma1);
```



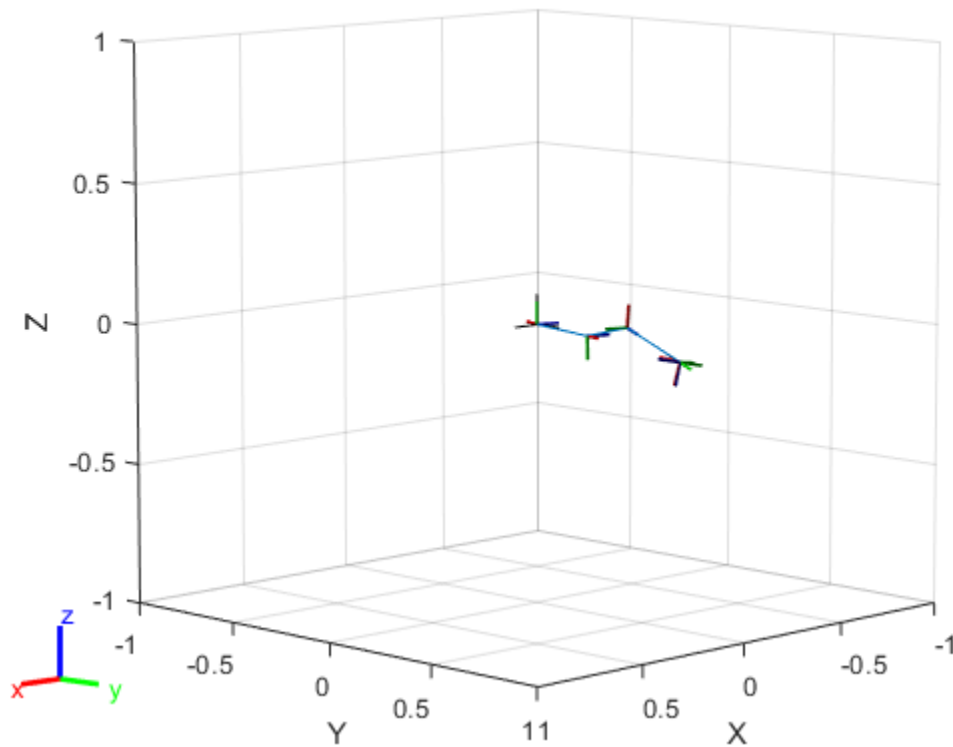
Modify the configuration and set the second joint position to  $\pi/2$ . Show the resulting change in the robot configuration.

```
config(2).JointPosition = pi/2;  
show(puma1,config);
```



Create random configurations and show them.

```
show(puma1,randomConfiguration(puma1));
```



## See Also

`robotics.RigidBodyTree.geometricJacobian` |  
`robotics.RigidBodyTree.getTransform` |  
`robotics.RigidBodyTree.randomConfiguration`

**Introduced in R2016b**



# inverseDynamics

**Class:** robotics.RigidBodyTree

**Package:** robotics

Required joint torques for given motion

## Syntax

```
jointTorq = inverseDynamics(robot)
jointTorq = inverseDynamics(robot, configuration)
jointTorq = inverseDynamics(robot, configuration, jointVel)
jointTorq = inverseDynamics(robot, configuration, jointVel, jointAccel)
jointTorq = inverseDynamics(robot, configuration, jointVel, jointAccel,
fext)
```

## Description

`jointTorq = inverseDynamics(robot)` computes joint torques required for the robot to statically hold its home configuration with no external forces applied.

`jointTorq = inverseDynamics(robot, configuration)` computes joint torques to hold the specified robot configuration.

`jointTorq = inverseDynamics(robot, configuration, jointVel)` computes joint torques for the specified joint configuration and velocities with zero acceleration and no external forces.

`jointTorq = inverseDynamics(robot, configuration, jointVel, jointAccel)` computes joint torques for the specified joint configuration, velocities, and accelerations with no external forces.

`jointTorq = inverseDynamics(robot, configuration, jointVel, jointAccel, fext)` computes joint torques for the specified joint configuration, velocities, accelerations, and external forces. Use the `externalForce` function to generate `fext`.

To specify the home configuration, zero joint velocities, or zero accelerations, use [ ] for that input argument.

## Input Arguments

### **robot — Robot model**

RigidBodyTree object

Robot model, specified as a RigidBodyTree object. To use the `inverseDynamics` function, set the `DataFormat` property to either 'row' or 'column'.

### **configuration — Robot configuration**

vector

Robot configuration, specified as a vector with positions for all nonfixed joints in the robot model. You can generate a configuration using `homeConfiguration(robot)`, `randomConfiguration(robot)`, or by specifying your own joint positions. To use the vector form of configuration, set the `DataFormat` property for the robot to either 'row' or 'column'.

### **jointVel — Joint velocities**

vector

Joint velocities, specified as a vector. The number of joint velocities is equal to the velocity degrees of freedom of the robot. To use the vector form of `jointVel`, set the `DataFormat` property for the robot to either 'row' or 'column'.

### **jointAccel — Joint accelerations**

vector

Joint accelerations, returned as a vector. The dimension of the joint accelerations vector is equal to the velocity degrees of freedom of the robot. Each element corresponds to a specific joint on the robot. To use the vector form of `jointAccel`, set the `DataFormat` property for the robot to either 'row' or 'column'.

### **fext — External force matrix**

$n$ -by-6 matrix | 6-by- $n$  matrix

External force matrix, specified as either an  $n$ -by-6 or 6-by- $n$  matrix, where  $n$  is the velocity degrees of freedom of the robot. The shape depends on the `DataFormat`

property of robot. The 'row' data format uses an  $n$ -by-6 matrix. The 'column' data format uses a 6-by- $n$ .

The matrix lists only values other than zero at the locations relevant to the body specified. You can add force matrices together to specify multiple forces on multiple bodies.

To create the matrix for a specified force or torque, see `externalForce`.

## Output Arguments

### **jointTorq** — Joint torques

vector

Joint torques, returned as a vector. Each element corresponds to a torque applied to a specific joint.

## Examples

### **Compute Inverse Dynamics from Static Joint Configuration**

Use the `inverseDynamics` function to calculate the required joint torques to statically hold a specific robot configuration. You can also specify the joint velocities, joint accelerations, and external forces using other syntaxes.

Load a predefined KUKA LBR robot model, which is specified as a `RigidBodyTree` object.

```
load exampleRobots.mat lbr
```

Set the data format to 'row'. For all dynamics calculations, the data format must be either 'row' or 'column'.

```
lbr.DataFormat = 'row';
```

Set the Gravity property to give a specific gravitational acceleration.

```
lbr.Gravity = [0 0 -9.81];
```

Generate a random configuration for `lbr`.

```
q = randomConfiguration(lbr);
```

Compute the required joint torques for `lbr` to statically hold that configuration.

```
tau = inverseDynamics(lbr,q);
```

#### Compute Joint Torque to Counter External Forces

Use the `externalForce` function to generate force matrices to apply to a rigid body tree model. The force matrix is an  $m$ -by-6 vector that has a row for each joint on the robot to apply a six-element wrench. Use the `externalForce` function and specify the end effector to properly assign the wrench to the correct row of the matrix. You can add multiple force matrices together to apply multiple forces to one robot.

To calculate the joint torques that counter these external forces, use the `inverseDynamics` function.

Load a predefined KUKA LBR robot model, which is specified as a `RigidBodyTree` object.

```
load exampleRobots.mat lbr
```

Set the data format to 'row'. For all dynamics calculations, the data format must be either 'row' or 'column'.

```
lbr.DataFormat = 'row';
```

Set the `Gravity` property to give a specific gravitational acceleration.

```
lbr.Gravity = [0 0 -9.81];
```

Get the home configuration for `lbr`.

```
q = homeConfiguration(lbr);
```

Set external force on `link1`. The input wrench vector is expressed in the base frame.

```
fext1 = externalForce(lbr, 'link_1', [0 0 0.0 0.1 0 0]);
```

Set external force on the end effector, `tool0`. The input wrench vector is expressed in the `tool0` frame.

```
fext2 = externalForce(lbr, 'tool0', [0 0 0.0 0.1 0 0], q);
```

Compute the joint torques required to balance the external forces. To combine the forces, add the force matrices together. Joint velocities and accelerations are assumed to be zero (input as []).

```
tau = inverseDynamics(lbr, q, [], [], fext1+fext2);
```

## See Also

[RigidBodyTree](#) | [externalForce](#) | [forwardDynamics](#)

## Topics

“Control LBR Manipulator Motion Through Joint Torque Commands”

**Introduced in R2017a**

## massMatrix

**Class:** robotics.RigidBodyTree

**Package:** robotics

Joint-space mass matrix

## Syntax

`H = massMatrix(robot)`

`H = massMatrix(robot, configuration)`

## Description

`H = massMatrix(robot)` returns the joint-space mass matrix of the home configuration of a robot.

`H = massMatrix(robot, configuration)` returns the mass matrix for a specified robot configuration.

## Input Arguments

### **robot — Robot model**

RigidBodyTree object

Robot model, specified as a RigidBodyTree object. To use the massMatrix function, set the DataFormat property to either 'row' or 'column'.

### **configuration — Robot configuration**

vector

Robot configuration, specified as a vector with positions for all nonfixed joints in the robot model. You can generate a configuration using homeConfiguration(robot), randomConfiguration(robot), or by specifying your own joint positions. To use the vector form of configuration, set the DataFormat property for the robot to either 'row' or 'column'.

## Output Arguments

### **H** — Mass matrix

positive-definite symmetric matrix

Mass matrix of the robot, returned as a positive-definite symmetric matrix with size  $n$ -by- $n$ , where  $n$  is the velocity degrees of freedom of the robot.

## Examples

### Calculate The Mass Matrix For A Robot Configuration

Load a predefined KUKA LBR robot model, which is specified as a `RigidBodyTree` object.

```
load exampleRobots.mat lbr
```

Set the data format to 'row'. For all dynamics calculations, the data format must be either 'row' or 'column'.

```
lbr.DataFormat = 'row';
```

Generate a random configuration for `lbr`.

```
q = randomConfiguration(lbr);
```

Get the mass matrix at configuration `q`.

```
H = massMatrix(lbr,q);
```

## See Also

[RigidBodyTree](#) | [gravityTorque](#) | [homeConfiguration](#) | [velocityProduct](#)

## Topics

“Control LBR Manipulator Motion Through Joint Torque Commands”

**Introduced in R2017a**



# randomConfiguration

**Class:** robotics.RigidBodyTree

**Package:** robotics

Generate random configuration of robot

## Syntax

```
configuration = randomConfiguration(robot)
```

## Description

`configuration = randomConfiguration(robot)` returns a random configuration of the specified robot. Each joint position in this configuration respects the joint limits set by the `PositionLimits` property of the corresponding `Joint` object in the robot model.

## Input Arguments

**robot — Robot model**

RigidBodyTree object

Robot model, specified as a RigidBodyTree object.

## Output Arguments

**configuration — Robot configuration**

vector | structure

Robot configuration, returned as a vector of joint positions or a structure with joint names and positions for all the bodies in the robot model. You can generate a configuration using `homeConfiguration(robot)`, `randomConfiguration(robot)`, or by specifying your own joint positions in a structure. To use the vector form of `configuration`, set the `DataFormat` property for the robot to either `'row'` or `'column'`.

## Examples

### Visualize Robot Configurations

Show different configurations of a robot created using a `RigidBodyTree` model. Use the `homeConfiguration` or `randomConfiguration` functions to generate the structure that defines all the joint positions.

Load example robots as `RigidBodyTree` objects.

```
load exampleRobots.mat
```

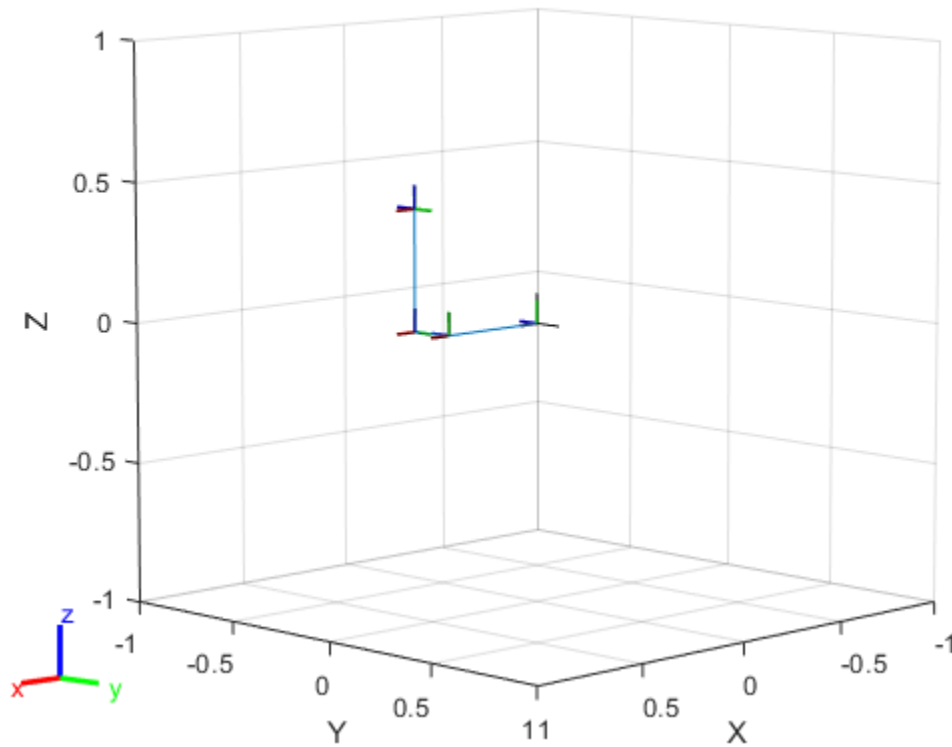
Create a structure for the home configuration of a Puma robot. The structure has joint names and positions for each body on the robot model.

```
config = homeConfiguration(puma1)

config = 1x6 struct array with fields:
    JointName
    JointPosition
```

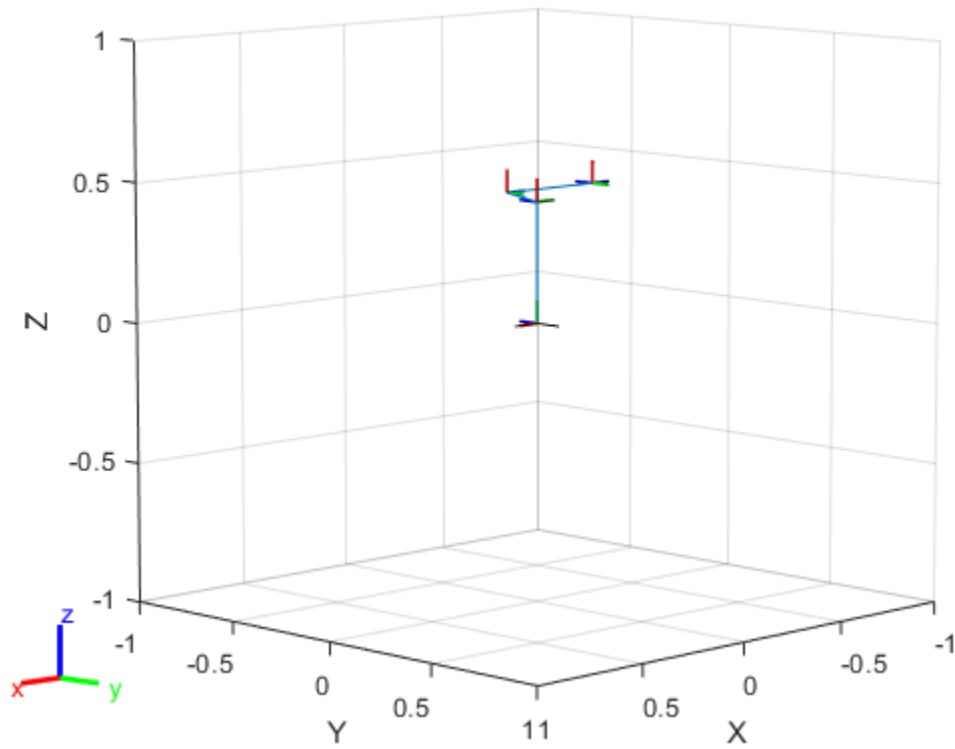
Show the home configuration using `show`. You do not need to specify a configuration input.

```
show(puma1);
```



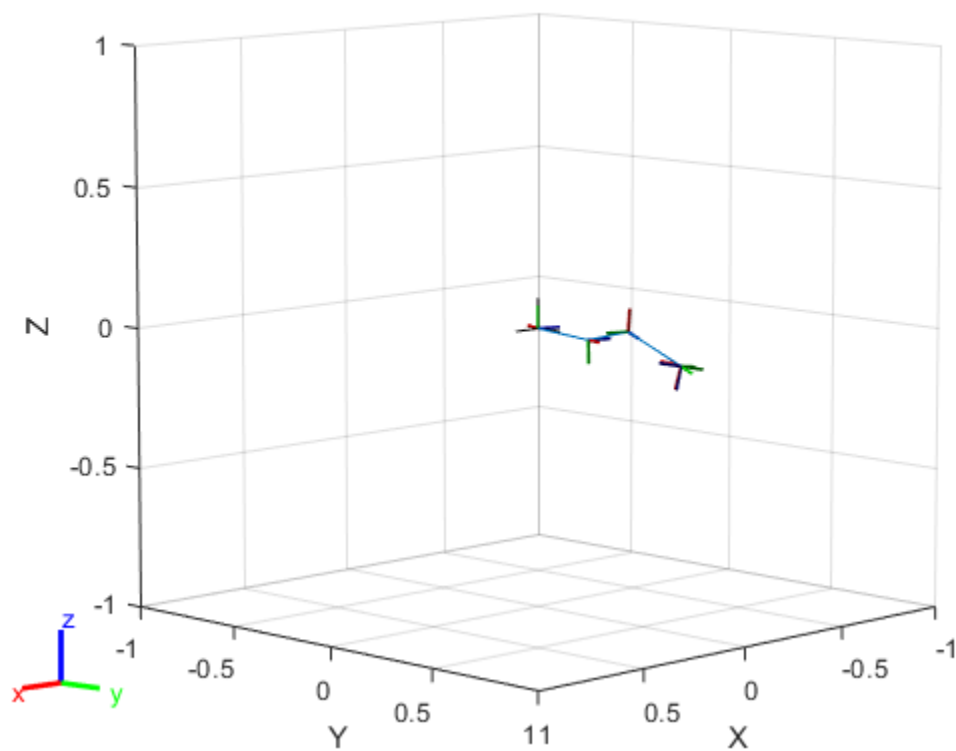
Modify the configuration and set the second joint position to  $\pi/2$ . Show the resulting change in the robot configuration.

```
config(2).JointPosition = pi/2;  
show(puma1,config);
```



Create random configurations and show them.

```
show(puma1,randomConfiguration(puma1));
```



## See Also

`robotics.RigidBodyTree.geometricJacobian` |  
`robotics.RigidBodyTree.getTransform` |  
`robotics.RigidBodyTree.homeConfiguration`

**Introduced in R2016b**

# removeBody

**Class:** robotics.RigidBodyTree

**Package:** robotics

Remove body from robot

## Syntax

```
removeBody(robot, bodyname)  
newSubtree = removeBody(robot, bodyname)
```

## Description

`removeBody(robot, bodyname)` removes the body and all subsequently attached bodies from the robot model.

`newSubtree = removeBody(robot, bodyname)` returns the subtree created by removing the body and all subsequently attached bodies from the robot model.

## Input Arguments

**robot — Robot model**

RigidBodyTree object

Robot model, specified as a RigidBodyTree object.

**bodyname — Body name**

string scalar | character vector

Body name, specified as a string scalar character vector. This body must be on the robot model specified in `robot`.

Data Types: char | string

## Output Arguments

### newSubtree — Robot subtree

RigidBodyTree object

Robot subtree, returned as a RigidBodyTree object. This new subtree uses the parent name of the body specified by `bodyname` as the base name. All bodies that are attached in the previous robot model (including the body with `bodyname` specified) are added to the subtree.

## Examples

### Modify a Robot Rigid Body Tree Model

Make changes to an existing RigidBodyTree object. You can get replace joints, bodies and subtrees in the rigid body tree.

Load example robots as RigidBodyTree objects.

```
load exampleRobots.mat
```

View the details of the Puma robot using `showdetails`.

```
showdetails(puma1)
```

```
-----
Robot: (6 bodies)

  Idx   Body Name   Joint Name   Joint Type   Parent Name(Idx)   Children Name(s)
  ---   -
  1     L1             jnt1         revolute     base(0)            L2(2)
  2     L2             jnt2         revolute     L1(1)              L3(3)
  3     L3             jnt3         revolute     L2(2)              L4(4)
  4     L4             jnt4         revolute     L3(3)              L5(5)
  5     L5             jnt5         revolute     L4(4)              L6(6)
  6     L6             jnt6         revolute     L5(5)
-----
```

Get a specific body to inspect the properties. The only child of the L3 body is the L4 body. You can copy a specific body as well.

```
body3 = getBody(puma1, 'L3');
childBody = body3.Children{1}

childBody =
  RigidBody with properties:
      Name: 'L4'
      Joint: [1x1 robotics.Joint]
      Mass: 1
      CenterOfMass: [0 0 0]
      Inertia: [1 1 1 0 0 0]
      Parent: [1x1 robotics.RigidBody]
      Children: {[1x1 robotics.RigidBody]}
      Visuals: {}
```

```
body3Copy = copy(body3);
```

Replace the joint on the L3 body. You must create a new `Joint` object and use `replaceJoint` to ensure the downstream body geometry is unaffected. Call `setFixedTransform` if necessary to define a transform between the bodies instead of with the default identity matrices.

```
newJoint = robotics.Joint('prismatic');
replaceJoint(puma1, 'L3', newJoint);
```

```
showdetails(puma1)
```

```
-----
Robot: (6 bodies)
```

Idx	Body Name	Joint Name	Joint Type	Parent Name(Idx)	Children Name
---	-----	-----	-----	-----	-----
1	L1	jnt1	revolute	base(0)	L2(2)
2	L2	jnt2	revolute	L1(1)	L3(3)
3	L3	prismatic	fixed	L2(2)	L4(4)
4	L4	jnt4	revolute	L3(3)	L5(5)
5	L5	jnt5	revolute	L4(4)	L6(6)
6	L6	jnt6	revolute	L5(5)	
-----	-----	-----	-----	-----	-----

Remove an entire body and get the resulting subtree using `removeBody`. The removed body is included in the subtree.

```
subtree = removeBody(puma1, 'L4')
```



```

subtree =
  RigidBodyTree with properties:

    NumBodies: 3
    Bodies: {1x3 cell}
    Base: [1x1 robotics.RigidBody]
    BodyNames: {'L4' 'L5' 'L6'}
    BaseName: 'L3'
    Gravity: [0 0 0]
    DataFormat: 'struct'

```

Remove the modified L3 body. Add the original copied L3 body to the L2 body, followed by the returned subtree. The robot model remains the same. See a detailed comparison through `showdetails`.

```

removeBody(puma1, 'L3');
addBody(puma1, body3Copy, 'L2')
addSubtree(puma1, 'L3', subtree)

```

```
showdetails(puma1)
```

```
-----
Robot: (6 bodies)
```

Idx	Body Name	Joint Name	Joint Type	Parent Name(Idx)	Children Name(s)
1	L1	jnt1	revolute	base(0)	L2(2)
2	L2	jnt2	revolute	L1(1)	L3(3)
3	L3	jnt3	revolute	L2(2)	L4(4)
4	L4	jnt4	revolute	L3(3)	L5(5)
5	L5	jnt5	revolute	L4(4)	L6(6)
6	L6	jnt6	revolute	L5(5)	

## See Also

`robotics.Joint` | `robotics.RigidBody` | `robotics.RigidBodyTree.addBody` | `robotics.RigidBodyTree.replaceBody`

**Introduced in R2016b**

## replaceBody

**Class:** robotics.RigidBodyTree

**Package:** robotics

Replace body on robot

### Syntax

```
replaceBody( robot , bodyname , newbody )
```

### Description

`replaceBody( robot , bodyname , newbody )` replaces the body in the robot model with the new body. All properties of the body are updated accordingly, except the `Parent` and `Children` properties. The rest of the robot model is unaffected.

### Input Arguments

**robot — Robot model**

RigidBodyTree object

Robot model, specified as a RigidBodyTree object. The rigid body is added to this object and attached at the rigid body specified by `bodyname`.

**bodyname — Body name**

string scalar | character vector

Body name, specified as a string scalar or character vector. This body must be on the robot model specified in `robot`.

Data Types: char | string

**newbody — Rigid body**

RigidBody object

Rigid body, specified as a RigidBody object.

## See Also

`robotics.Joint` | `robotics.RigidBody` | `robotics.RigidBodyTree.addBody` |  
`robotics.RigidBodyTree.removeBody` |  
`robotics.RigidBodyTree.replaceJoint`

**Introduced in R2016b**

## replaceJoint

**Class:** robotics.RigidBodyTree

**Package:** robotics

Replace joint on body

### Syntax

```
replaceJoint(robot, bodyname, joint)
```

### Description

`replaceJoint(robot, bodyname, joint)` replaces the joint on the specified body in the robot model if the body is a part of the robot model. This method is the only way to change joints in a robot model. You cannot directly assign the `Joint` property of a rigid body.

### Input Arguments

**robot — Robot model**

RigidBodyTree object

Robot model, specified as a RigidBodyTree object.

**bodyname — Body name**

string scalar | character vector

Body name, specified as a string scalar or character vector. This body must be on the robot model specified in `robot`.

Data Types: char | string

**joint — Replacement joint**

Joint object

Replacement joint, specified as a Joint object.

## Examples

### Modify a Robot Rigid Body Tree Model

Make changes to an existing RigidBodyTree object. You can get replace joints, bodies and subtrees in the rigid body tree.

Load example robots as RigidBodyTree objects.

```
load exampleRobots.mat
```

View the details of the Puma robot using showdetails.

```
showdetails(puma1)
```

```
-----  
Robot: (6 bodies)
```

Idx	Body Name	Joint Name	Joint Type	Parent Name(Idx)	Children Name(s)
1	L1	jnt1	revolute	base(0)	L2(2)
2	L2	jnt2	revolute	L1(1)	L3(3)
3	L3	jnt3	revolute	L2(2)	L4(4)
4	L4	jnt4	revolute	L3(3)	L5(5)
5	L5	jnt5	revolute	L4(4)	L6(6)
6	L6	jnt6	revolute	L5(5)	

Get a specific body to inspect the properties. The only child of the L3 body is the L4 body. You can copy a specific body as well.

```
body3 = getBody(puma1, 'L3');  
childBody = body3.Children{1}
```

```
childBody =
```

```
  RigidBody with properties:
```

```
      Name: 'L4'  
      Joint: [1x1 robotics.Joint]  
      Mass: 1  
      CenterOfMass: [0 0 0]  
      Inertia: [1 1 1 0 0 0]  
      Parent: [1x1 robotics.RigidBody]
```

```
Children: {[1x1 robotics.RigidBody]}
Visuals: {}
```

```
body3Copy = copy(body3);
```

Replace the joint on the L3 body. You must create a new `Joint` object and use `replaceJoint` to ensure the downstream body geometry is unaffected. Call `setFixedTransform` if necessary to define a transform between the bodies instead of with the default identity matrices.

```
newJoint = robotics.Joint('prismatic');
replaceJoint(puma1, 'L3', newJoint);
```

```
showdetails(puma1)
```

```
-----
Robot: (6 bodies)
```

Idx	Body Name	Joint Name	Joint Type	Parent Name(Idx)	Children Name
---	-----	-----	-----	-----	-----
1	L1	jnt1	revolute	base(0)	L2(2)
2	L2	jnt2	revolute	L1(1)	L3(3)
3	L3	prismatic	fixed	L2(2)	L4(4)
4	L4	jnt4	revolute	L3(3)	L5(5)
5	L5	jnt5	revolute	L4(4)	L6(6)
6	L6	jnt6	revolute	L5(5)	

Remove an entire body and get the resulting subtree using `removeBody`. The removed body is included in the subtree.

```
subtree = removeBody(puma1, 'L4')
```

```
subtree =
  RigidBodyTree with properties:
    NumBodies: 3
    Bodies: {1x3 cell}
    Base: [1x1 robotics.RigidBody]
    BodyNames: {'L4' 'L5' 'L6'}
    BaseName: 'L3'
    Gravity: [0 0 0]
    DataFormat: 'struct'
```

Remove the modified L3 body. Add the original copied L3 body to the L2 body, followed by the returned subtree. The robot model remains the same. See a detailed comparison through `showdetails`.

```
removeBody(puma1, 'L3');
addBody(puma1, body3Copy, 'L2')
addSubtree(puma1, 'L3', subtree)
```

```
showdetails(puma1)
```

```
-----
Robot: (6 bodies)
```

Idx	Body Name	Joint Name	Joint Type	Parent Name(Idx)	Children Name(s)
1	L1	jnt1	revolute	base(0)	L2(2)
2	L2	jnt2	revolute	L1(1)	L3(3)
3	L3	jnt3	revolute	L2(2)	L4(4)
4	L4	jnt4	revolute	L3(3)	L5(5)
5	L5	jnt5	revolute	L4(4)	L6(6)
6	L6	jnt6	revolute	L5(5)	

## See Also

`robotics.Joint` | `robotics.RigidBody` | `robotics.RigidBodyTree.addBody` | `robotics.RigidBodyTree.replaceBody`

**Introduced in R2016b**

## show

**Class:** robotics.RigidBodyTree

**Package:** robotics

Show robot model in a figure

## Syntax

```
show(robot)
show(robot, configuration)
show( ____, Name, Value)
ax = show( ____ )
```

## Description

`show(robot)` plots the body frames of the robot model in a figure with the predefined home configuration. Both `Frames` and `Visuals` are displayed automatically.

`show(robot, configuration)` uses the joint positions specified in `configuration` to show the robot body frames.

`show( ____, Name, Value)` provides additional options specified by one or more `Name, Value` pair arguments.

`ax = show( ____ )` returns the axes handle the robot is plotted on.

## Input Arguments

**robot — Robot model**

RigidBodyTree object

Robot model, specified as a RigidBodyTree object.

**configuration — Robot configuration**

vector | structure



Robot configuration, specified as a vector of joint positions or a structure with joint names and positions for all the bodies in the robot model. You can generate a configuration using `homeConfiguration(robot)`, `randomConfiguration(robot)`, or by specifying your own joint positions in a structure. To use the vector form of configuration, set the `DataFormat` property for the robot to either "row" or "column".

## 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`.

### Parent — Parent of axes

Axes object

Parent of axes, specified as the comma-separated pair consisting of `Parent` and an `Axes` object in which to draw the robot. By default, the robot is plotted in the active axes.

### PreservePlot — Preserve robot plot

`true` (default) | `false`

Option to reserve robot plot, specified as the comma-separated pair consisting of "PreservePlot" and `true` or `false`. When this property is set to `true`, previous plots displayed by calling `show` are not overwritten. This setting functions similar to calling `hold on` for a standard MATLAB figure, but is limited to the robot body frames. When this property is set to `false`, previous plots of the robot are overwritten.

### Frames — Display body frames

"on" (default) | "off"

Display body frames, specified as "on" or "off". These frames are the coordinate frames of individual bodies on the rigid body tree.

### Visuals — Display visual geometries

"on" (default) | "off"

Display visual geometries, specified as "on" or "off". Individual visual geometries can also be turned off by right-clicking them in the figure.

You can either specify individual visual geometries using `robotics.RigidBody.addVisual` or by using the `importrobot` to import a robot model with `.stl` files specified.

## Output Arguments

### **ax** — Axes graphic handle

Axes object

Axes graphic handle, returned as an Axes object. This object contains the properties of the figure that the robot is plotted onto.

## Examples

### **Display Robot Model with Visual Geometries**

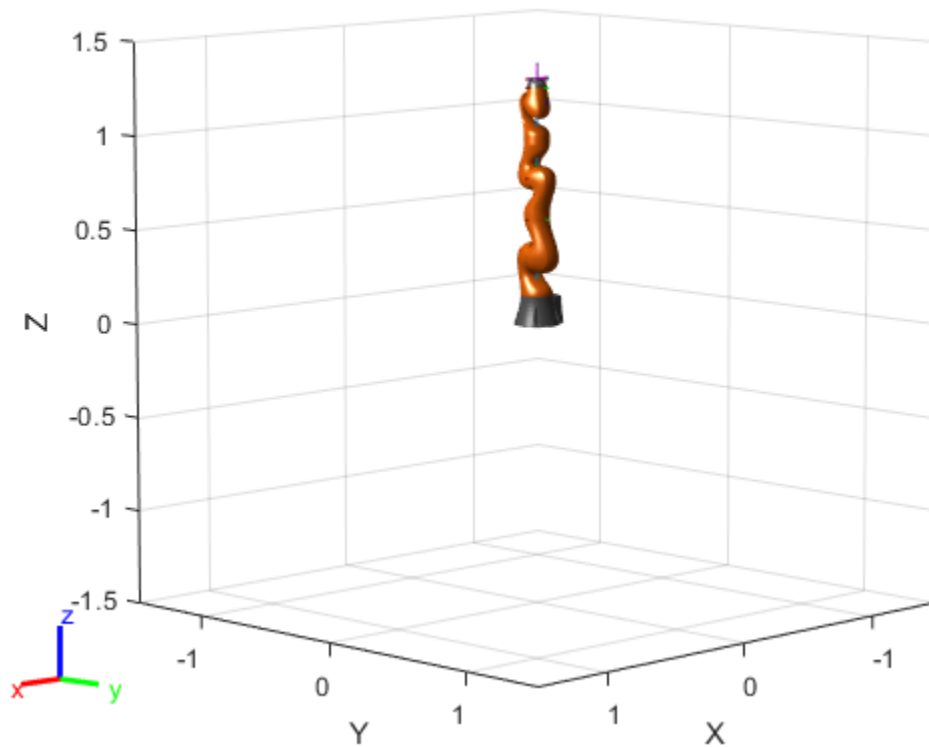
You can import robots that have `.stl` files associated with the Unified Robot Description format (URDF) file to describe the visual geometries of the robot. Each rigid body has an individual visual geometry specified. The `importrobot` function parses the URDF file to get the robot model and visual geometries. Use the `show` function to visualize the robot model in a figure. You can then interact with the model by clicking components to inspect them and right-clicking to toggle visibility.

Import a robot model as a URDF file. The `.stl` file locations must be properly specified in this URDF. To add other `.stl` files to individual rigid bodies, see `addVisual`.

```
robot = importrobot('iiwa14.urdf');
```

Visualize the robot with the associated visual model. Click bodies or frames to inspect them. Right-click bodies to toggle visibility for each visual geometry.

```
show(robot);
```



### Visualize Robot Configurations

Show different configurations of a robot created using a `RigidBodyTree` model. Use the `homeConfiguration` or `randomConfiguration` functions to generate the structure that defines all the joint positions.

Load example robots as `RigidBodyTree` objects.

```
load exampleRobots.mat
```

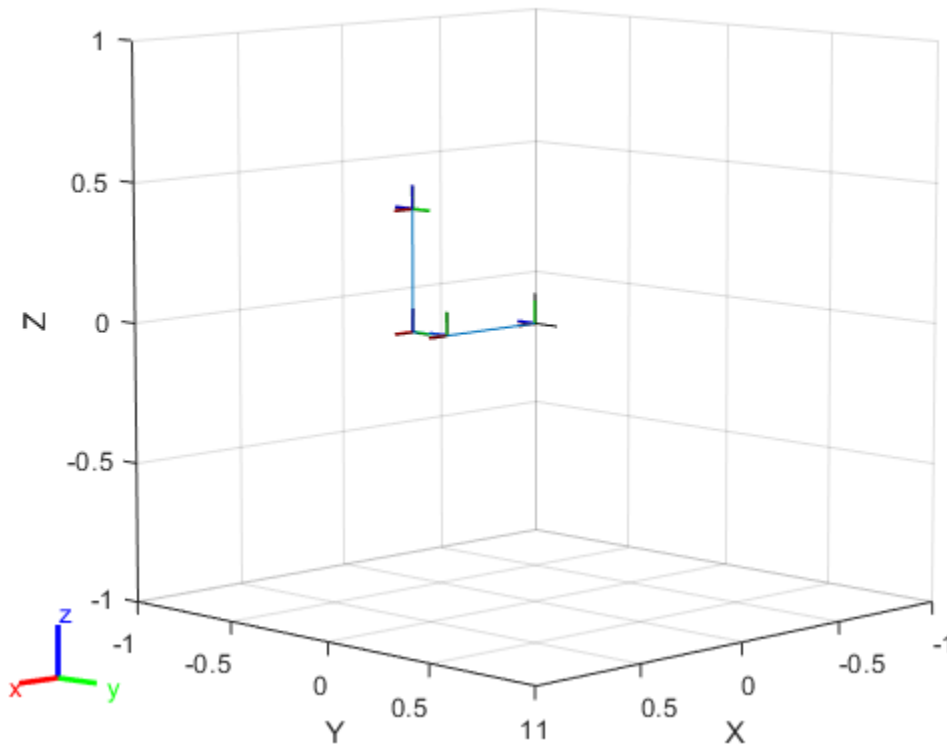
Create a structure for the home configuration of a Puma robot. The structure has joint names and positions for each body on the robot model.

```
config = homeConfiguration(puma1)

config = 1x6 struct array with fields:
    JointName
    JointPosition
```

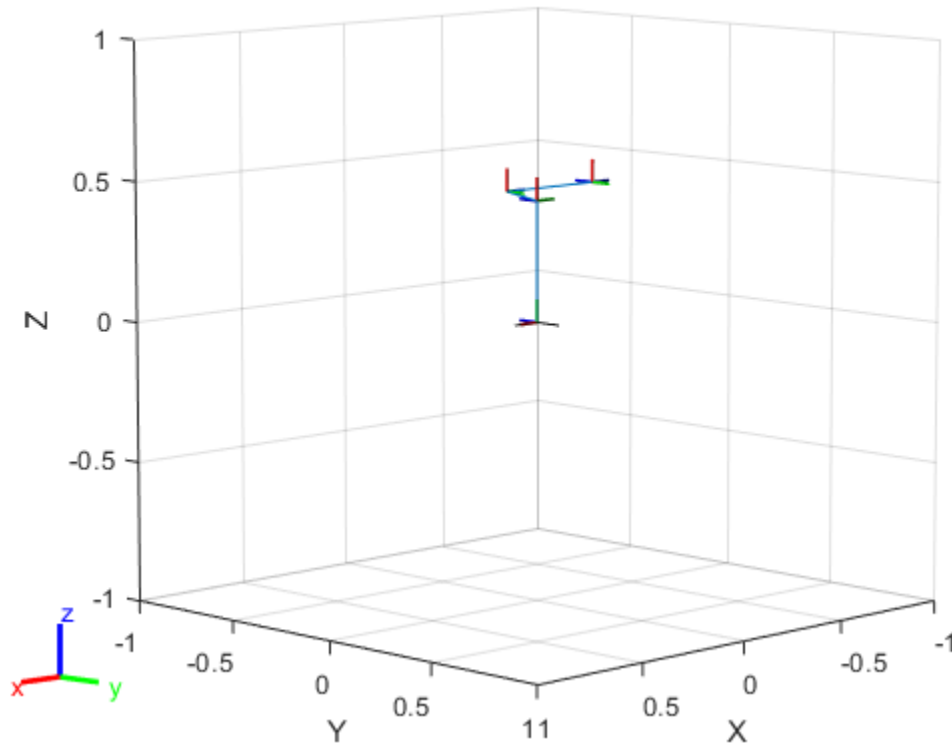
Show the home configuration using `show`. You do not need to specify a configuration input.

```
show(puma1);
```



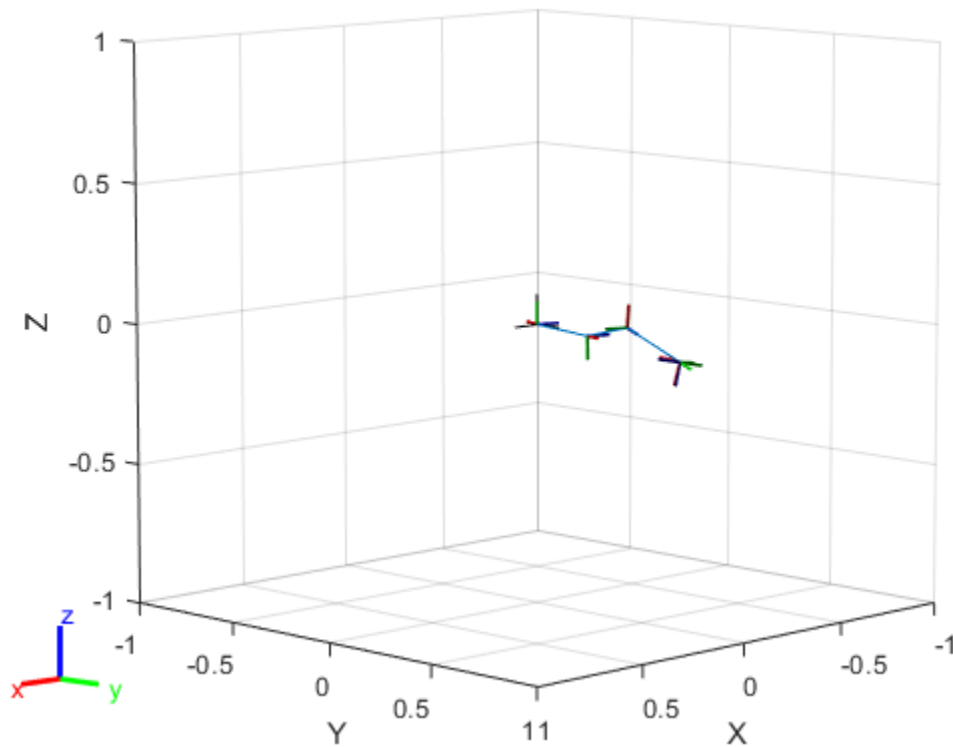
Modify the configuration and set the second joint position to  $\pi/2$ . Show the resulting change in the robot configuration.

```
config(2).JointPosition = pi/2;  
show(puma1,config);
```



Create random configurations and show them.

```
show(puma1,randomConfiguration(puma1));
```



### **Build Manipulator Robot Using Denavit-Hartenberg Parameters**

Use the Denavit-Hartenberg (DH) parameters of the Puma560® robot to build a robot. Each rigid body is added one at a time, with the child-to-parent transform specified by the joint object.

The DH parameters define the geometry of the robot with relation to how each rigid body is attached to its parent. For convenience, setup the parameters for the Puma560 robot in a matrix. The Puma robot is a serial chain manipulator. The DH parameters are relative to the previous line in the matrix, corresponding to the previous joint attachment.

```

dhparams = [0      pi/2    0      0;
            0.4318  0      0      0;
            0.0203 -pi/2   0.15005 0;
            0      pi/2    0.4318  0;
            0      -pi/2   0      0;
            0      0      0      0];

```

Create a rigid body tree object to build the robot.

```
robot = robotics.RigidBodyTree;
```

Create the first rigid body and add it to the robot. To add a rigid body:

- 1 Create a `RigidBody` object and give it a unique name.
- 2 Create a `Joint` object and give it a unique name.
- 3 Use `setFixedTransform` to specify the body-to-body transformation using DH parameters. The last element of the DH parameters, `theta`, is ignored because the angle is dependent on the joint position.
- 4 Call `addBody` to attach the first body joint to the base frame of the robot.

```

body1 = robotics.RigidBody('body1');
jnt1 = robotics.Joint('jnt1','revolute');

setFixedTransform(jnt1,dhparams(1,:), 'dh');
body1.Joint = jnt1;

addBody(robot,body1,'base')

```

Create and add other rigid bodies to the robot. Specify the previous body name when calling `addBody` to attach it. Each fixed transform is relative to the previous joint coordinate frame.

```

body2 = robotics.RigidBody('body2');
jnt2 = robotics.Joint('jnt2','revolute');
body3 = robotics.RigidBody('body3');
jnt3 = robotics.Joint('jnt3','revolute');
body4 = robotics.RigidBody('body4');
jnt4 = robotics.Joint('jnt4','revolute');
body5 = robotics.RigidBody('body5');
jnt5 = robotics.Joint('jnt5','revolute');
body6 = robotics.RigidBody('body6');
jnt6 = robotics.Joint('jnt6','revolute');

```

```

setFixedTransform(jnt2,dhparams(2,:), 'dh');
setFixedTransform(jnt3,dhparams(3,:), 'dh');
setFixedTransform(jnt4,dhparams(4,:), 'dh');
setFixedTransform(jnt5,dhparams(5,:), 'dh');
setFixedTransform(jnt6,dhparams(6,:), 'dh');

body2.Joint = jnt2;
body3.Joint = jnt3;
body4.Joint = jnt4;
body5.Joint = jnt5;
body6.Joint = jnt6;

addBody(robot,body2, 'body1')
addBody(robot,body3, 'body2')
addBody(robot,body4, 'body3')
addBody(robot,body5, 'body4')
addBody(robot,body6, 'body5')

```

Verify that your robot was built properly by using the `showdetails` or `show` function. `showdetails` lists all the bodies in the MATLAB® command window. `show` displays the robot with a given configuration (home by default). Calls to `axis` modify the axis limits and hide the axis labels.

```
showdetails(robot)
```

```

-----
Robot: (6 bodies)

  Idx   Body Name   Joint Name   Joint Type   Parent Name(Idx)   Children Name(s)
  ---   -
  1     body1         jnt1        revolute    base(0)            body2(2)
  2     body2         jnt2        revolute    body1(1)           body3(3)
  3     body3         jnt3        revolute    body2(2)           body4(4)
  4     body4         jnt4        revolute    body3(3)           body5(5)
  5     body5         jnt5        revolute    body4(4)           body6(6)
  6     body6         jnt6        revolute    body5(5)
-----

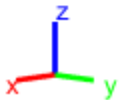
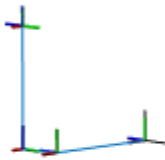
```

```

show(robot);
axis([-0.5,0.5,-0.5,0.5,-0.5,0.5])
axis off

```





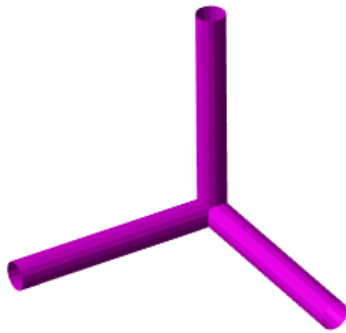
## Tips

### Visual Components

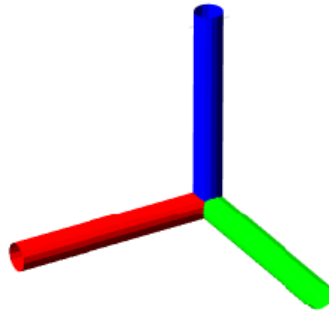
Your robot model has visual components associated with it. Each `RigidBody` object contains a coordinate frame that is displayed as the body frame. Each body also can have visual meshes associated with them. By default, both of these components are displayed automatically. You can inspect or modify the visual components of the rigid body tree display. Click body frames or visual meshes to highlight them in yellow and see the

associated body name, index, and joint type. Right-click to toggle visibility of individual components.

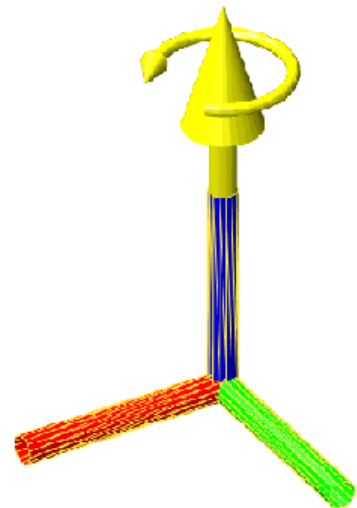
- **Body Frames:** Individual body frames are displayed as a 3-axis coordinate frame. Fixed frames are pink frames. Movable joint types are displayed as RGB axes. You can click a body frame to see the axis of motion. Prismatic joints show a yellow arrow in the direction of the axis of motion and, revolute joints show a circular arrow around the rotation axis.



Fixed Joint Frame

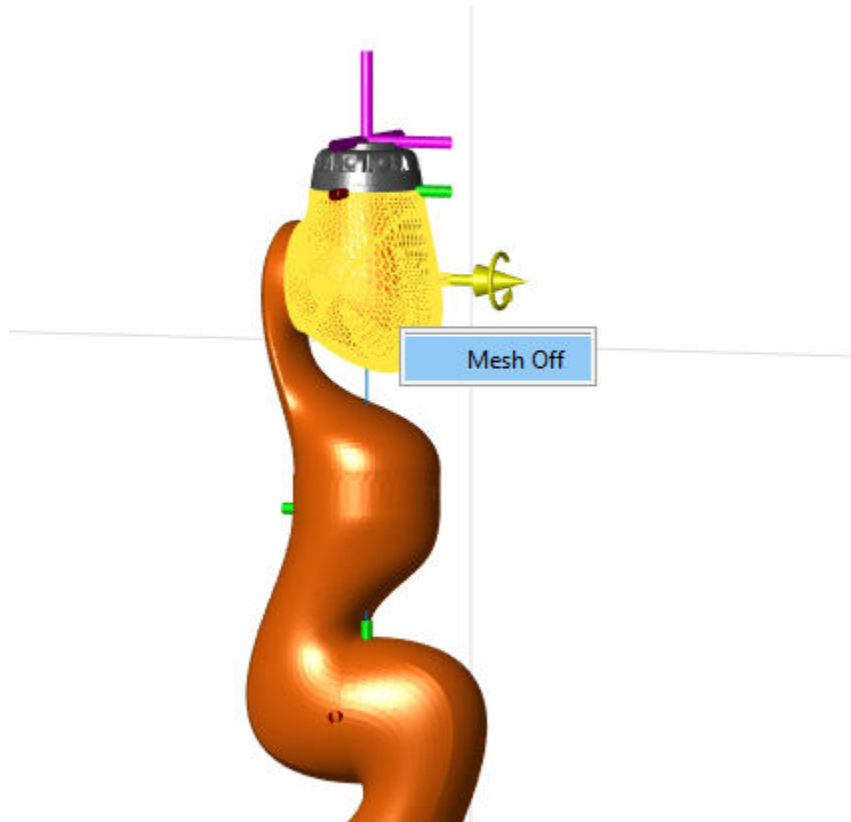


Moveable Joint Frame



Selected Revolute Joint

- **Visual Meshes:** Individual visual geometries are specified using `robotics.RigidBody.addVisual` or by using the `importrobot` to import a robot model with `.stl` files specified. By right-clicking individual bodies in a figure, you can turn off their meshes or specify the `Visuals` name-value pair to hide all visual geometries.



## See Also

`importrobot | robotics.RigidBodyTree.randomConfiguration |  
robotics.RigidBodyTree.showdetails`

**Introduced in R2016b**

## showdetails

**Class:** robotics.RigidBodyTree

**Package:** robotics

Show details of robot model

## Syntax

```
showdetails(robot)
```

## Description

`showdetails(robot)` displays in the MATLAB command window the details of each body in the robot model. These details include the body name, associated joint name, joint type, parent name, and children names.

## Input Arguments

**robot — Robot model**

RigidBodyTree object

Robot model, specified as a RigidBodyTree object.

## Examples

### Attach Rigid Body and Joint to Rigid Body Tree

Add a rigid body and corresponding joint to a rigid body tree. Each RigidBody object contains a Joint object and must be added to the RigidBodyTree using `addBody`.

Create a rigid body tree.

```
rbtree = robotics.RigidBodyTree;
```

Create a rigid body with a unique name.

```
body1 = robotics.RigidBody('b1');
```

Create a revolute joint. By default, the `RigidBody` object comes with a fixed joint. Replace the joint by assigning a new `Joint` object to the `body1.Joint` property.

```
jnt1 = robotics.Joint('jnt1','revolute');
body1.Joint = jnt1;
```

Add the rigid body to the tree. Specify the body name that you are attaching the rigid body to. Because this is the first body, use the base name of the tree.

```
basename = rbtree.BaseName;
addBody(rbtree,body1,basename)
```

Use `showdetails` on the tree to confirm the rigid body and joint were added properly.

```
showdetails(rbtree)
```

```
-----
```

```
Robot: (1 bodies)
```

Idx	Body Name	Joint Name	Joint Type	Parent Name(Idx)	Children Name(s)
1	b1	jnt1	revolute	base(0)	

```
-----
```

## Modify a Robot Rigid Body Tree Model

Make changes to an existing `RigidBodyTree` object. You can get replace joints, bodies and subtrees in the rigid body tree.

Load example robots as `RigidBodyTree` objects.

```
load exampleRobots.mat
```

View the details of the Puma robot using `showdetails`.

```
showdetails(puma1)
```

```
-----
```

```
Robot: (6 bodies)
```

Idx	Body Name	Joint Name	Joint Type	Parent Name(Idx)	Children Name(s)
1	L1	jnt1	revolute	base(0)	L2(2)
2	L2	jnt2	revolute	L1(1)	L3(3)
3	L3	jnt3	revolute	L2(2)	L4(4)
4	L4	jnt4	revolute	L3(3)	L5(5)
5	L5	jnt5	revolute	L4(4)	L6(6)
6	L6	jnt6	revolute	L5(5)	

Get a specific body to inspect the properties. The only child of the L3 body is the L4 body. You can copy a specific body as well.

```
body3 = getBody(puma1, 'L3');
childBody = body3.Children{1}
```

```
childBody =
```

```
  RigidBody with properties:
```

```
      Name: 'L4'
      Joint: [1x1 robotics.Joint]
      Mass: 1
  CenterOfMass: [0 0 0]
      Inertia: [1 1 1 0 0 0]
      Parent: [1x1 robotics.RigidBody]
  Children: {[1x1 robotics.RigidBody]}
  Visuals: {}
```

```
body3Copy = copy(body3);
```

Replace the joint on the L3 body. You must create a new `Joint` object and use `replaceJoint` to ensure the downstream body geometry is unaffected. Call `setFixedTransform` if necessary to define a transform between the bodies instead of with the default identity matrices.

```
newJoint = robotics.Joint('prismatic');
replaceJoint(puma1, 'L3', newJoint);
```

```
showdetails(puma1)
```

```
-----
Robot: (6 bodies)
```

Idx	Body Name	Joint Name	Joint Type	Parent Name(Idx)	Children Name
1	L1	jnt1	revolute	base(0)	L2(2)
2	L2	jnt2	revolute	L1(1)	L3(3)
3	L3	prismatic	fixed	L2(2)	L4(4)
4	L4	jnt4	revolute	L3(3)	L5(5)
5	L5	jnt5	revolute	L4(4)	L6(6)
6	L6	jnt6	revolute	L5(5)	

Remove an entire body and get the resulting subtree using `removeBody`. The removed body is included in the subtree.

```
subtree = removeBody(puma1, 'L4')
```

```
subtree =
```

```
RigidBodyTree with properties:
```

```

  NumBodies: 3
  Bodies: {1x3 cell}
  Base: [1x1 robotics.RigidBody]
  BodyNames: {'L4' 'L5' 'L6'}
  BaseName: 'L3'
  Gravity: [0 0 0]
  DataFormat: 'struct'
```

Remove the modified L3 body. Add the original copied L3 body to the L2 body, followed by the returned subtree. The robot model remains the same. See a detailed comparison through `showdetails`.

```
removeBody(puma1, 'L3');
addBody(puma1, body3Copy, 'L2')
addSubtree(puma1, 'L3', subtree)
```

```
showdetails(puma1)
```

```
Robot: (6 bodies)
```

Idx	Body Name	Joint Name	Joint Type	Parent Name(Idx)	Children Name(s)
1	L1	jnt1	revolute	base(0)	L2(2)
2	L2	jnt2	revolute	L1(1)	L3(3)
3	L3	jnt3	revolute	L2(2)	L4(4)

4	L4	jnt4	revolute	L3(3)	L5(5)
5	L5	jnt5	revolute	L4(4)	L6(6)
6	L6	jnt6	revolute	L5(5)	

-----

### Build Manipulator Robot Using Denavit-Hartenberg Parameters

Use the Denavit-Hartenberg (DH) parameters of the Puma560® robot to build a robot. Each rigid body is added one at a time, with the child-to-parent transform specified by the joint object.

The DH parameters define the geometry of the robot with relation to how each rigid body is attached to its parent. For convenience, setup the parameters for the Puma560 robot in a matrix. The Puma robot is a serial chain manipulator. The DH parameters are relative to the previous line in the matrix, corresponding to the previous joint attachment.

```
dhparams = [0      pi/2    0      0;
            0.4318  0      0      0;
            0.0203  -pi/2   0.15005  0;
            0      pi/2    0.4318  0;
            0      -pi/2   0      0;
            0      0      0      0];
```

Create a rigid body tree object to build the robot.

```
robot = robotics.RigidBodyTree;
```

Create the first rigid body and add it to the robot. To add a rigid body:

- 1 Create a `RigidBody` object and give it a unique name.
- 2 Create a `Joint` object and give it a unique name.
- 3 Use `setFixedTransform` to specify the body-to-body transformation using DH parameters. The last element of the DH parameters, `theta`, is ignored because the angle is dependent on the joint position.
- 4 Call `addBody` to attach the first body joint to the base frame of the robot.

```
body1 = robotics.RigidBody('body1');
jnt1 = robotics.Joint('jnt1','revolute');

setFixedTransform(jnt1,dhparams(1,:), 'dh');
```



```
body1.Joint = jnt1;

addBody(robot,body1,'base')
```

Create and add other rigid bodies to the robot. Specify the previous body name when calling `addBody` to attach it. Each fixed transform is relative to the previous joint coordinate frame.

```
body2 = robotics.RigidBody('body2');
jnt2 = robotics.Joint('jnt2','revolute');
body3 = robotics.RigidBody('body3');
jnt3 = robotics.Joint('jnt3','revolute');
body4 = robotics.RigidBody('body4');
jnt4 = robotics.Joint('jnt4','revolute');
body5 = robotics.RigidBody('body5');
jnt5 = robotics.Joint('jnt5','revolute');
body6 = robotics.RigidBody('body6');
jnt6 = robotics.Joint('jnt6','revolute');

setFixedTransform(jnt2,dhparams(2,:), 'dh');
setFixedTransform(jnt3,dhparams(3,:), 'dh');
setFixedTransform(jnt4,dhparams(4,:), 'dh');
setFixedTransform(jnt5,dhparams(5,:), 'dh');
setFixedTransform(jnt6,dhparams(6,:), 'dh');

body2.Joint = jnt2;
body3.Joint = jnt3;
body4.Joint = jnt4;
body5.Joint = jnt5;
body6.Joint = jnt6;

addBody(robot,body2,'body1')
addBody(robot,body3,'body2')
addBody(robot,body4,'body3')
addBody(robot,body5,'body4')
addBody(robot,body6,'body5')
```

Verify that your robot was built properly by using the `showdetails` or `show` function. `showdetails` lists all the bodies in the MATLAB® command window. `show` displays the robot with a given configuration (home by default). Calls to `axis` modify the axis limits and hide the axis labels.

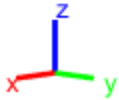
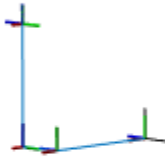
```
showdetails(robot)
```

-----  
Robot: (6 bodies)

Idx	Body Name	Joint Name	Joint Type	Parent Name(Idx)	Children Name(s)
1	body1	jnt1	revolute	base(0)	body2(2)
2	body2	jnt2	revolute	body1(1)	body3(3)
3	body3	jnt3	revolute	body2(2)	body4(4)
4	body4	jnt4	revolute	body3(3)	body5(5)
5	body5	jnt5	revolute	body4(4)	body6(6)
6	body6	jnt6	revolute	body5(5)	

-----

```
show(robot);  
axis([-0.5,0.5,-0.5,0.5,-0.5,0.5])  
axis off
```



## See Also

`robotics.RigidBodyTree.replaceBody` |  
`robotics.RigidBodyTree.replaceJoint` | `robotics.RigidBodyTree.show`

**Introduced in R2016b**

# subtree

**Class:** robotics.RigidBodyTree

**Package:** robotics

Create subtree from robot model

## Syntax

```
newSubtree = subtree(robot, bodyname)
```

## Description

`newSubtree = subtree(robot, bodyname)` creates a new robot model using the parent name of the body specified by `bodyname` as the base name. All subsequently attached bodies (including the body with `bodyname` specified) are added to the subtree. The original robot model is unaffected.

## Input Arguments

**robot — Robot model**

RigidBodyTree object

Robot model, specified as a RigidBodyTree object.

**bodyname — Body name**

string scalar | character vector

Body name, specified as a string scalar or character vector. This body must be on the robot model specified in `robot`.

Data Types: char | string

## Output Arguments

### **newSubtree — Robot subtree**

RigidBodyTree object

Robot subtree, returned as a RigidBodyTree object. This new subtree uses the parent name of the body specified by `bodyname` as the base name. All bodies that are attached in the previous robot model (including the body with `bodyname` specified) are added to the subtree.

## See Also

`robotics.Joint` | `robotics.RigidBody` | `robotics.RigidBodyTree.addBody` | `robotics.RigidBodyTree.replaceBody`

**Introduced in R2016b**

## velocityProduct

**Class:** robotics.RigidBodyTree

**Package:** robotics

Joint torques that cancel velocity-induced forces

### Syntax

```
jointTorq = velocityProduct(robot, configuration, jointVel)
```

### Description

`jointTorq = velocityProduct(robot, configuration, jointVel)` computes the joint torques required to cancel the forces induced by the specified joint velocities under a certain joint configuration. Gravity torque is not included in this calculation.

### Input Arguments

**robot — Robot model**

RigidBodyTree object

Robot model, specified as a RigidBodyTree object. To use the `velocityProduct` function, set the `DataFormat` property to either `'row'` or `'column'`.

**configuration — Robot configuration**

vector

Robot configuration, specified as a vector with positions for all nonfixed joints in the robot model. You can generate a configuration using `homeConfiguration(robot)`, `randomConfiguration(robot)`, or by specifying your own joint positions. To use the vector form of `configuration`, set the `DataFormat` property for the robot to either `'row'` or `'column'`.

**jointVel — Joint velocities**

vector

Joint velocities, specified as a vector. The number of joint velocities is equal to the velocity degrees of freedom of the robot. To use the vector form of `jointVel`, set the `DataFormat` property for the robot to either `'row'` or `'column'`.

## Output Arguments

### **jointTorq** – Joint torques

vector

Joint torques, specified as a vector. Each element corresponds to a torque applied to a specific joint.

## Examples

### **Compute Velocity-Induced Joint Torques**

Load a predefined KUKA LBR robot model, which is specified as a `RigidBodyTree` object.

```
load exampleRobots.mat lbr
```

Set the data format to `'row'`. For all dynamics calculations, the data format must be either `'row'` or `'column'`.

```
lbr.DataFormat = 'row';
```

Set the joint velocity vector.

```
qdot = [0 0 0.2 0.3 0 0.1 0];
```

Compute the joint torques required to cancel the velocity-induced joint torques at the robot home configuration (`[]` input). The velocity-induced joint torques equal the negative of the `velocityProduct` output.

```
tau = -velocityProduct(lbr,[],qdot);
```

## **See Also**

`RigidBodyTree` | `gravityTorque` | `inverseDynamics` | `massMatrix`

## **Topics**

“Control LBR Manipulator Motion Through Joint Torque Commands”

**Introduced in R2017a**



# Blocks — Alphabetical List

---

## Blank Message

Create blank message using specified message type

**Library:** Robotics System Toolbox / ROS



## Description

The Blank Message block creates a Simulink nonvirtual bus corresponding to the selected ROS message type. The block creates ROS message buses that work with Publish, Subscribe, or Call Service blocks. On each sample hit, the block outputs a blank or “zero” signal for the designated message type. All elements of the bus are initialized to 0. The lengths of the variable-length arrays are also initialized to 0.

## Limitations

Before R2016b, models using ROS message types with certain reserved property names could not generate code. In 2016b, this limitation has been removed. The property names are now appended with an underscore (e.g. `Vector3Stamped_`). If you use models created with a pre-R2016b release, update the ROS message types using the new names with an underscore. Redefine custom maximum sizes for variable length arrays.

The affected message types are:

- 'geometry\_msgs/Vector3Stamped'
- 'jsk\_pcl\_ros/TransformScreenpointResponse'
- 'pddl\_msgs/PDDLAction'
- 'rocon\_interaction\_msgs/Interaction'
- 'capabilities/GetRemappingsResponse'
- 'dynamic\_reconfigure/Group'

# Input/Output Ports

## Output

### Msg — Blank ROS message

nonvirtual bus

Blank ROS message, returned as a nonvirtual bus. To specify the type of ROS message, use the **Message type** parameter. All elements of the bus are initialized to 0. The lengths of the variable-length arrays are also initialized to 0.

Data Types: bus

## Parameters

### Class — Class of ROS message

Message (default) | Service Request | Service Response

Class of ROS message, specified as Message, Service Request, or Service Response. For basic publishing and subscribing, use the Message class.

### Type — ROS message type

'geometry\_msgs/Point' (default) | character vector | dialog box selection

ROS message type, specified as a character vector or a dialog box selection. Use **Select** to select from a list of supported ROS messages. The list of messages given depends on the **Class** of message you select.

### Sample time — Interval between outputs

Inf (default) | numeric scalar

Interval between outputs, specified as a numeric scalar. The default value indicates that the block output never changes. Using this value speeds simulation and code generation by eliminating the need to recompute the block output. Otherwise, the block outputs a new blank message at each interval of **Sample time**.

For more information, see “Specify Sample Time” (Simulink).

## **Extended Capabilities**

### **C/C++ Code Generation**

Generate C and C++ code using Simulink® Coder™.

### **See Also**

[Call Service](#) | [Publish](#) | [Subscribe](#)

### **Topics**

[“Get Started with ROS in Simulink®”](#)

[“Work with ROS Messages in Simulink®”](#)

[“Connect to a ROS-enabled Robot from Simulink®”](#)

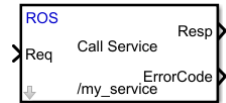
[“Virtual and Nonvirtual Buses” \(Simulink\)](#)

**Introduced in R2015a**

# Call Service

Call service in ROS network

**Library:** Robotics System Toolbox / ROS



## Description

The Call Service block takes a ROS service request message, sends it to the ROS service server, and waits for a response. Connect to a ROS network using `rosinit`. A ROS server should be set up somewhere on the network before using this block. Check the available services on a ROS network using `rosservice`. Use `rossvcserver` to set up a service server in MATLAB.

Specify the name for your ROS service and the service type in the block mask. If connected to a ROS network, you can select from a list of available services. You can create a blank service request or response message to populate with data using the Blank Message block.

## Ports

### Input

#### Req — Request message

nonvirtual bus

Request message, specified as a nonvirtual bus. The request message type corresponds to your service type. To generate an empty request message bus to populate with data, use the Blank Message block.

Data Types: bus

## Output

### Resp — Response message

nonvirtual bus

Response message, returned as a nonvirtual bus. The response is based on the input **Req** message. The response message type corresponds to your service type. To generate an empty response message bus to populate with data, use the Blank Message block.

Data Types: bus

### ErrorCode — Error conditions for service call

integer

Error conditions for service call, specified as an integer. Each integer corresponds to a different error condition for the service connection or the status of the service call. If an error condition occurs, **Resp** outputs the last response message or a blank message if a response was not previously received.

#### Error Codes:

Error Code	Condition
0	The service response was successfully retrieved and is available in the Resp output.
1	The connection was not established within the specified Connection timeout.
2	The response from the server was not received within the specified Call timeout
3	The service call failed for unknown reasons.

#### Dependencies

This output is enabled when the **Show ErrorCode output port** check box is on.

Data Types: uint8

## Parameters

### Source — Source for specifying service name

Select from ROS network | Specify your own

Source for specifying the service name:

- **Select from ROS network** — Use **Select** to select a service name. The **Name** and **Type** parameters are set automatically. You must be connected to a ROS network.
- **Specify your own** — Enter a service name in **Name** and specify its service type in **Type**. You must match a service name exactly.

**Name — Service name**

character vector

Service name, specified as a character vector. The service name must match a service name available on the ROS service server. To see a list of valid services in a ROS network, see `rosservice`.

**Type — Service type**

character vector

Service type, specified as a character vector. Each service name has a corresponding type.

**Connection timeout — Timeout for service server connection**

5 (default) | positive numeric scalar

Timeout for service server connection, specified as a positive numeric scalar in seconds. If a connection cannot be established with the ROS service server in this time, then **ErrorCode** outputs 1.

**Keep persistent connection — Keep connection to service server**

off (default) | on

Check this box to maintain a persistent connection with the ROS service server. When **off**, the block creates a service client every time a request message is input into **Req**.

**Show ErrorCode output port — Enable error code output port**

on (default) | off

Check this box to output the **ErrorCode** output. If an error condition occurs, **Resp** outputs the last response message or a blank message if response was not previously received.

## **Extended Capabilities**

### **C/C++ Code Generation**

Generate C and C++ code using Simulink® Coder™.

### **See Also**

#### **Blocks**

[Blank Message](#) | [Publish](#) | [Subscribe](#)

#### **Functions**

[rosservice](#) | [rossvcclient](#) | [rossvcserver](#)

#### **Topics**

[“Call and Provide ROS Services”](#)

[“Publish and Subscribe to ROS Messages in Simulink”](#)

#### **Introduced in R2018b**



# Coordinate Transformation Conversion

Convert to a specified coordinate transformation representation

**Library:** Robotics System 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 (Eu1) - [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 Eu1, 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.

For more information about the different coordinate transformation representations, see “Coordinate Transformations in Robotics”.

## 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 (Eu1) - [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 Eu1, 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 (EuI) - [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 EuI, for example), you can specify two inputs or outputs to handle all transformation information. When you select the Homogeneous Transformation as an input or output, an optional `Show TrVec input/output port` parameter can be selected on the block mask to toggle the multiple ports.

### TrVec — Translation vector

three-element column vector

Translation vector, specified as a three-element column vector, [x y z], which corresponds to a translation in the x, y, and z axes respectively. This port can be used to input or output the translation information separately from the rotation vector.

### Dependencies

You must select Homogeneous Transformation (TForm) for the opposite transformation port to get the option to show the additional TrVec port. Enable the port by clicking `Show TrVec input/output port`.

## Parameters

### Representation — Input or output representation

Axis-Angle | Euler Angles | Homogeneous Transformation | Rotation Matrix | Translation Vector | Quaternion

Select the representation for both the input and output port for the block. If you are using a transformation with only orientation information, you can also select the Show TrVec input/output port when converting to or from a homogeneous transformation.

### **Show TrVec input/output port — Toggle TrVec port**

off (default) | on

Toggle the TrVec input or output port when you want to specify or receive a separate translation vector for position information along with an orientation representation.

### **Dependencies**

You must select Homogeneous Transformation (TForm) for the opposite transformation port to get the option to show the additional TrVec port.

## **Extended Capabilities**

### **C/C++ Code Generation**

Generate C and C++ code using Simulink® Coder™.

### **See Also**

[axang2quat](#) | [eul2tform](#) | [trvec2tform](#)

### **Topics**

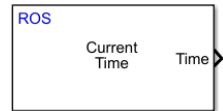
“Coordinate Transformations in Robotics”

**Introduced in R2017b**

# Current Time

Retrieve current ROS time or system time

**Library:** Robotics System Toolbox / ROS

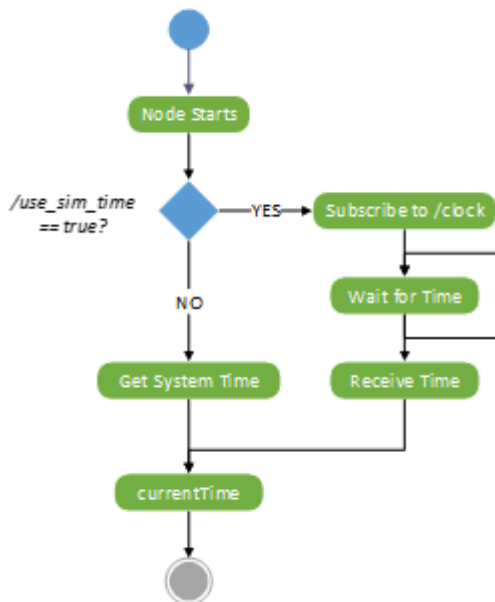


## Description

The Current Time block outputs the current ROS or system time. ROS Time is based on the system clock of your computer or the `/clock` topic being published on the ROS node.

Use this block to synchronize your simulation time with your connected ROS node.

If the `use_sim_time` ROS parameter is set to `true`, the block returns the simulation time published on the `/clock` topic. Otherwise, the block returns the system time of your machine.



# Ports

## Output

### Time — ROS time

bus | scalar

ROS time, returned as a bus signal or a scalar. The bus represents a `rosgraph_msgs/Clock` ROS message with `Sec` and `NSec` elements. The scalar is the ROS time in seconds. If no time has been received on the `/clock` topic, the block outputs 0.

Data Types: bus | double

# Parameters

### Output format — Format of ROS time

bus (default) | double

Format of ROS Time output, specified as either bus or double.

### Sample time — Interval between outputs

-1 (default) | numeric scalar

Interval between outputs, specified as a numeric scalar.

For more information, see “Specify Sample Time” (Simulink).

# Tips

- To set the `use_sim_time` parameters and get time from a `/clock` topic:

Connect to a ROS network, then use the Set Parameter block or set the parameter in the MATLAB command window:

```
ptree = rosparam;  
set(ptree, '/use_sim_time', true)
```

Usually, the ROS node that publishes on the `/clock` topic sets up the parameter.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

### See Also

#### Blocks

Get Parameter | Publish | Set Parameter

#### Functions

get | rosparam | rospublisher | rostime | set

### External Websites

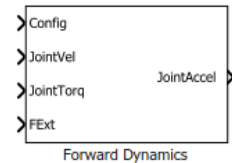
ROS Time

**Introduced in R2018b**

## Forward Dynamics

Joint accelerations given joint torques and states

**Library:** Robotics System Toolbox / Manipulator Algorithms



## Description

The Forward Dynamics block computes joint accelerations for a robot model given a robot state that is made up of joint torques, joint states, and external forces. To get the joint accelerations, specify the robot configuration (joint positions), joint velocities, applied torques, and external forces.

Specify the robot model in the **Rigid body tree** parameter as a `RigidBodyTree` object, and set the Gravity property on the object. You can also import a robot model from an URDF (Unified Robot Description Formation) file using `importrobot`.

## Ports

### Input

#### **Config** — Robot configuration

vector

Robot configuration, specified as a vector of positions for all nonfixed joints in the robot model, as set by the **Rigid body tree** parameter. You can also generate this vector for a complex robot using the `homeConfiguration` or `randomConfiguration` functions inside a Constant or MATLAB Function block.

#### **JointVel** — Joint velocities

vector



Joint velocities, specified as a vector. The number of joint velocities is equal to the degrees of freedom (number of nonfixed joints) of the robot.

### **JointTorq — Joint torques**

vector

Joint torques, specified as a vector. Each element corresponds to a torque applied to a specific joint. The number of joint torques is equal to the degrees of freedom (number of nonfixed joints) of the robot.

### **FExt — External force matrix**

6-by- $n$  matrix

External force matrix, specified as a 6-by- $n$  matrix, where  $n$  is the number of bodies in the robot model. The matrix contains nonzero values in the rows corresponding to specific bodies. Each row is a vector of applied forces and torques that act as a wrench for that specific body. Generate this matrix using `externalForce` with a MATLAB Function block

## **Output**

### **JointAccel — Joint accelerations**

vector

Joint accelerations, returned as a vector. The number of joint accelerations is equal to the degrees of freedom of the robot.

## **Parameters**

### **Rigid body tree — Robot model**

`twoJointRigidBodyTree` (default) | `RigidBodyTree` object

Robot model, specified as a `RigidBodyTree` object. You can also import a robot model from an URDF (Unified Robot Description Formation) file using `importrobot`.

The default robot model, `twoJointRigidBodyTree`, is a robot with revolute joints and two degrees of freedom.

### **Simulate using — Type of simulation to run**

Interpreted execution (default) | Code generation

- **Interpreted execution** — Simulate model using the MATLAB interpreter. This option shortens startup time but has a slower simulation speed than **Code generation**. In this mode, you can debug the source code of the block.
- **Code generation** — Simulate model using generated C code. The first time you run a simulation, Simulink generates C code for the block. The C code is reused for subsequent simulations, as long as the model does not change. This option requires additional startup time, but the speed of the subsequent simulations is comparable to **Interpreted execution**.

**Tunable:** No

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

## See Also

### Blocks

[Get Jacobian](#) | [Get Transform](#) | [Gravity Torque](#) | [Inverse Dynamics](#) | [Joint Space Mass Matrix](#) | [Velocity Product Torque](#)

### Classes

[RigidBodyTree](#)

### Functions

[externalForce](#) | [forwardDynamics](#) | [homeConfiguration](#) | [importrobot](#) | [randomConfiguration](#)

### Topics

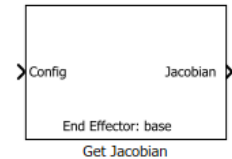
[“Perform Safe Trajectory Tracking Control Using Robotics Manipulator Blocks”](#)

**Introduced in R2018a**

# Get Jacobian

Geometric Jacobian for robot configuration

**Library:** Robotics System Toolbox / Manipulator Algorithms



## Description

The Get Jacobian block returns the geometric Jacobian relative to the base for the specified end effector at the given configuration of a RigidBodyTree robot model.

The Jacobian maps the joint-space velocity to the end-effector velocity relative to the base coordinate frame. The end-effector velocity equals:

$$V_{EE} = \begin{bmatrix} \omega_x \\ \omega_y \\ \omega_z \\ v_x \\ v_y \\ v_z \end{bmatrix} = J\dot{q} = J \begin{bmatrix} \dot{q}_1 \\ \vdots \\ \dot{q}_n \end{bmatrix}$$

$\omega$  is the angular velocity,  $v$  is the linear velocity, and  $\dot{q}$  is the joint-space velocity.

## Ports

### Input

**Config — Robot configuration**

vector

Robot configuration, specified as a vector of positions for all nonfixed joints in the robot model, as set by the **Rigid body tree** parameter. You can also generate this vector for a complex robot using the `homeConfiguration` or `randomConfiguration` functions inside a Constant or MATLAB Function block.

## Output

### Jacobian — Geometric Jacobian of end effector

6-by- $n$  matrix

Geometric jacobian of the end effector with the specified configuration, **Config**, returned as a 6-by- $n$  matrix, where  $n$  is the number of degrees of freedom of the end effector. The Jacobian maps the joint-space velocity to the end-effector velocity relative to the base coordinate frame. The end-effector velocity equals:

$$V_{EE} = \begin{bmatrix} \omega_x \\ \omega_y \\ \omega_z \\ v_x \\ v_y \\ v_z \end{bmatrix} = J\dot{q} = J \begin{bmatrix} \dot{q}_1 \\ \vdots \\ \dot{q}_n \end{bmatrix}$$

$\omega$  is the angular velocity,  $v$  is the linear velocity, and  $\dot{q}$  is the joint-space velocity.

## Parameters

### Rigid body tree — Robot model

`twoJointRigidBodyTree` (default) | `RigidBodyTree` object

Robot model, specified as a `RigidBodyTree` object. You can also import a robot model from an URDF (Unified Robot Description Formation) file using `importrobot`.

The default robot model, `twoJointRigidBodyTree`, is a robot with revolute joints and two degrees of freedom.

### End effector — End effector for Jacobian

body name

End effector for `Jacobian`, specified as a body name from the **Rigid body tree** robot model. To access body names from the robot model, click **Select body**.

### **Simulate using — Type of simulation to run**

Interpreted execution (default) | Code generation

- **Interpreted execution** — Simulate model using the MATLAB interpreter. This option shortens startup time but has a slower simulation speed than **Code generation**. In this mode, you can debug the source code of the block.
- **Code generation** — Simulate model using generated C code. The first time you run a simulation, Simulink generates C code for the block. The C code is reused for subsequent simulations, as long as the model does not change. This option requires additional startup time, but the speed of the subsequent simulations is comparable to **Interpreted execution**.

**Tunable:** No

## **Extended Capabilities**

### **C/C++ Code Generation**

Generate C and C++ code using Simulink® Coder™.

## **See Also**

### **Blocks**

[Forward Dynamics](#) | [Get Transform](#) | [Gravity Torque](#) | [Inverse Dynamics](#) | [Joint Space Mass Matrix](#) | [Velocity Product Torque](#)

### **Classes**

[RigidBodyTree](#)

### **Functions**

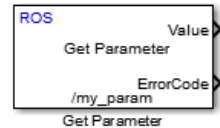
[geometricJacobian](#) | [homeConfiguration](#) | [importrobot](#) | [randomConfiguration](#)

**Introduced in R2018a**

# Get Parameter

Get values from ROS parameter server

**Library:** Robotics System Toolbox / ROS



## Description

The Get Parameter block outputs the value of the specified ROS parameter. The block uses the ROS node of the Simulink model to connect to the ROS network. This node is created when you run the model and is deleted when the model terminates. If the model does not have a node, the block creates one.

On each sample hit, the block checks the ROS parameter server for the specified ROS parameter and outputs its value.

## Input/Output Ports

### Output

#### **Value — Parameter value**

scalar | logical | uint8 array

Parameter value from the ROS network. The value depends on the **Data type** parameter.

#### **ErrorCode — Status of ROS parameter**

0 | 1 | 2 | 3

Status of ROS parameter, specified as one of the following:

- **0** — ROS parameter retrieved successfully. The retrieved value is output in the **Value** port.
- **1** — No ROS parameter with specified name found. If there is no known value, **Value** is set to the last received value or to **Initial value**.

- **2** — ROS parameter retrieved, but its type is different than the specified **Data type**. If there is no known value, Value is set to the last received value or to **Initial value**.
- **3** — For string parameters, the incoming string has been truncated based on the specified length.

### Length — Length of string parameter

integer

Length of the string parameter, returned as an integer. This length is the number of elements of the uint8 array or the number of characters in the string that you cast to uint8.

---

**Note** When getting string parameters from the ROS network, an ASCII value of 13 returns an error due to its incompatible character type.

---

### Dependencies

To enable this port, set the **Data type** to uint8[] (string).

## Parameters

### Source — Source for specifying the parameter name

Select from ROS network | Specify your own

Source for specifying the parameter name as one of the following:

- **Select from ROS network** — Use **Select** to select a parameter name. The **Data type** parameter is set automatically. You must be connected to a ROS network.
- **Specify your own** — Enter a parameter name in **Name** and specify its data type in **Data type**. You must match a parameter name exactly.

### Name — Parameter name

string

Parameter name to get from the ROS network, specified as a string. When **Source** is set to **Select from ROS network**, use **Select** to select an existing parameter. You must be connected to a ROS network to get a list of parameters. Otherwise, specify the parameter and data type.

Parameter name strings must follow the rules of ROS graph names. Valid names have these characteristics:

- The first character is an alpha character ([a-z|A-Z]), tilde (~), or forward slash (/).
- Subsequent characters are alphanumeric ([0-9|a-z|A-Z]), underscores(\_), or forward slashes (/).

### **Data type — Data type of your parameter**

double | int32 | boolean | uint8[] (string)

Data type of your parameter, specified as a string. The uint8[] (string) enables the **Maximum length** parameter.

---

**Note** The uint8[] (string) data type is an array of ASCII values corresponding to the characters in a string. When getting string parameters, you can create a MATLAB Function block to compare the string to a desired parameter value. For more information, see “ROS String Parameters”.

---

Data Types: double | int32 | Boolean | uint8

### **Maximum length — Maximum length of the uint8 array**

scalar

Maximum length of the uint8 array, specified as a scalar. If the parameter string has a length greater than **Maximum length**, the **ErrorCode** output is set to 3.

### **Dependencies**

To enable this port, set the **Data type** to uint8[] (string).

### **Initial value — Default parameter value output**

double | int32 | boolean | uint8

Default parameter value output from when an error occurs and no valid value has been received from the parameter server. The data type must match the specified **Data type**.

### **Sample time — Interval between outputs**

inf (default) | scalar

Interval between outputs, specified as a scalar. This default value indicates that the block output never changes. Using this value speeds simulation and code generation by



eliminating the need to recompute the block output. Otherwise, the block outputs a new blank message at each interval of `Sample time`.

For more information, see “Specify Sample Time” (Simulink).

### Show `ErrorCode` output port — Display error code output

on | off

To enable error code output, select this parameter. When you clear this parameter, the **ErrorCode** output port is removed from the block. The status options are:

- **0** — ROS parameter retrieved successfully. The retrieved value is output in the **Value** port.
- **1** — No ROS parameter with specified name found. If there is no known value, **Value** is set to the last received value or to **Initial value**.
- **2** — ROS parameter retrieved, but its type is different than the specified **Data type**. If there is no known value, **Value** is set to the last received value or to **Initial value**.
- **3** — For string parameters, the incoming string has been truncated based on the specified length.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

### See Also

Set Parameter

### Topics

“ROS Parameters in Simulink”

“ROS String Parameters”

### External Websites

ROS Parameter Server

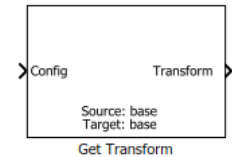
ROS Graph Names

**Introduced in R2015b**

# Get Transform

Get transform between body frames

**Library:** Robotics System Toolbox / Manipulator Algorithms



## Description

The Get Transform block returns the homogeneous transformation between body frames on the **Rigid body tree** robot model. Specify a **RigidBodyTree** object for the robot model, and select a source and target body in the block.

The block uses **Config**, the robot configuration (joint positions) input, to calculate the transformation from the source body to the target body. This transformation is used to convert coordinates from the source to the target body. To convert to base coordinates, use the base body name as the **Target body** parameter.

## Ports

### Input

#### **Config** — Robot configuration

vector

Robot configuration, specified as a vector of positions for all nonfixed joints in the robot model, as set by the **Rigid body tree** parameter. You can also generate this vector for a complex robot using the `homeConfiguration` or `randomConfiguration` functions inside a Constant or MATLAB Function block.

### Output

#### Transform — Homogeneous transform

4-by-4 matrix

Homogeneous transform, returned as a 4-by-4 matrix.

### Parameters

#### Rigid body tree — Robot model

twoJointRigidBodyTree (default) | RigidBodyTree object

Robot model, specified as a RigidBodyTree object. You can also import a robot model from an URDF (Unified Robot Description Formation) file using `importrobot`.

The default robot model, `twoJointRigidBodyTree`, is a robot with revolute joints and two degrees of freedom.

#### Target body — Target body name

body name

Target body name, specified as a body name from the robot model specified in **Rigid body tree**. To access body names from the robot model, click **Select body**. The target frame is the coordinate system you want to transform points into.

#### Source body — Source body name

body name

Source body name, specified as a body name from the robot model specified in **Rigid body tree**. To access body names from the robot model, click **Select body**. The source frame is the coordinate system you want points transformed from.

#### Simulate using — Type of simulation to run

Interpreted execution (default) | Code generation

- **Interpreted execution** — Simulate model using the MATLAB interpreter. This option shortens startup time but has a slower simulation speed than **Code generation**. In this mode, you can debug the source code of the block.
- **Code generation** — Simulate model using generated C code. The first time you run a simulation, Simulink generates C code for the block. The C code is reused for

subsequent simulations, as long as the model does not change. This option requires additional startup time, but the speed of the subsequent simulations is comparable to Interpreted execution.

**Tunable:** No

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

### See Also

#### Blocks

Get Jacobian | Gravity Torque | Inverse Dynamics | Joint Space Mass Matrix | Velocity Product Torque

#### Classes

RigidBodyTree

#### Functions

getTransform | homeConfiguration | importrobot | randomConfiguration

### Topics

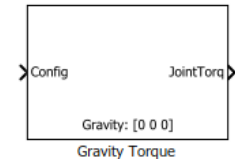
“Perform Safe Trajectory Tracking Control Using Robotics Manipulator Blocks”

**Introduced in R2018a**

# Gravity Torque

Joint torques that compensate gravity

**Library:** Robotics System Toolbox / Manipulator Algorithms



## Description

The Gravity Torque block returns the joint torques required to hold the robot at a given configuration with the current Gravity setting on the **Rigid body tree** robot model.

## Ports

### Input

#### **Config — Robot configuration**

vector

Robot configuration, specified as a vector of positions for all nonfixed joints in the robot model, as set by the **Rigid body tree** parameter. You can also generate this vector for a complex robot using the `homeConfiguration` or `randomConfiguration` functions inside a Constant or MATLAB Function block.

### Output

#### **JointTorq — Joint torques**

vector

Joint torques, specified as a vector. Each element corresponds to a torque applied to a specific joint. The number of joint torques is equal to the degrees of freedom (number of nonfixed joints) of the robot.

## Parameters

### Rigid body tree — Robot model

`twoJointRigidBodyTree` (default) | `RigidBodyTree` object

Robot model, specified as a `RigidBodyTree` object. You can also import a robot model from an URDF (Unified Robot Description Formation) file using `importrobot`.

The default robot model, `twoJointRigidBodyTree`, is a robot with revolute joints and two degrees of freedom.

### Simulate using — Type of simulation to run

`Interpreted execution` (default) | `Code generation`

- `Interpreted execution` — Simulate model using the MATLAB interpreter. This option shortens startup time but has a slower simulation speed than `Code generation`. In this mode, you can debug the source code of the block.
- `Code generation` — Simulate model using generated C code. The first time you run a simulation, Simulink generates C code for the block. The C code is reused for subsequent simulations, as long as the model does not change. This option requires additional startup time, but the speed of the subsequent simulations is comparable to `Interpreted execution`.

**Tunable:** No

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

## See Also

### Blocks

[Forward Dynamics](#) | [Get Jacobian](#) | [Inverse Dynamics](#) | [Joint Space Mass Matrix](#) | [Velocity Product Torque](#)

**Classes**

RigidBodyTree

**Functions**

gravityTorque | homeConfiguration | importrobot | randomConfiguration

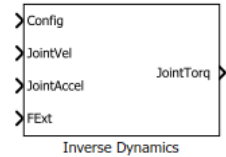
**Introduced in R2018a**



# Inverse Dynamics

Required joint torques for given motion

**Library:** Robotics System Toolbox / Manipulator Algorithms



## Description

The Inverse Dynamics block returns the joint torques required for the robot to maintain the specified robot state. To get the required joint torques, specify the robot configuration (joint positions), joint velocities, joint accelerations, and external forces.

## Ports

### Input

#### **Config — Robot configuration**

vector

Robot configuration, specified as a vector of positions for all nonfixed joints in the robot model, as set by the **Rigid body tree** parameter. You can also generate this vector for a complex robot using the `homeConfiguration` or `randomConfiguration` functions inside a Constant or MATLAB Function block.

#### **JointVel — Joint velocities**

vector

Joint velocities, specified as a vector. The number of joint velocities is equal to the degrees of freedom (number of nonfixed joints) of the robot.

#### **JointAccel — Joint accelerations**

vector

Joint accelerations, specified as a vector. The number of joint accelerations is equal to the degrees of freedom of the robot.

### **FExt — External force matrix**

6-by- $n$  matrix

External force matrix, specified as a 6-by- $n$  matrix, where  $n$  is the number of bodies in the robot model. The matrix contains nonzero values in the rows corresponding to specific bodies. Each row is a vector of applied forces and torques that act as a wrench for that specific body. Generate this matrix using `externalForce` with a MATLAB Function block

## Output

### **JointTorq — Joint torques**

vector

Joint torques, returned as a vector. Each element corresponds to a torque applied to a specific joint. The number of joint torques is equal to the degrees of freedom (number of nonfixed joints) of the robot.

## Parameters

### **Rigid body tree — Robot model**

`twoJointRigidBodyTree` (default) | `RigidBodyTree` object

Robot model, specified as a `RigidBodyTree` object. You can also import a robot model from an URDF (Unified Robot Description Formation) file using `importrobot`.

The default robot model, `twoJointRigidBodyTree`, is a robot with revolute joints and two degrees of freedom.

### **Simulate using — Type of simulation to run**

`Interpreted execution` (default) | `Code generation`

- `Interpreted execution` — Simulate model using the MATLAB interpreter. This option shortens startup time but has a slower simulation speed than `Code generation`. In this mode, you can debug the source code of the block.
- `Code generation` — Simulate model using generated C code. The first time you run a simulation, Simulink generates C code for the block. The C code is reused for

subsequent simulations, as long as the model does not change. This option requires additional startup time, but the speed of the subsequent simulations is comparable to Interpreted execution.

**Tunable:** No

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

### See Also

#### Blocks

[Forward Dynamics](#) | [Get Jacobian](#) | [Gravity Torque](#) | [Joint Space Mass Matrix](#) | [Velocity Product Torque](#)

#### Classes

[RigidBodyTree](#)

#### Functions

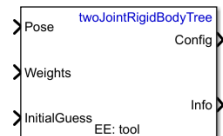
[externalForce](#) | [homeConfiguration](#) | [importrobot](#) | [inverseDynamics](#) | [randomConfiguration](#)

**Introduced in R2018a**

## Inverse Kinematics

Compute joint configurations to achieve an end-effector pose

**Library:** Robotics System Toolbox / Manipulator Algorithms



### Description

The Inverse Kinematics block uses an inverse kinematic (IK) solver to calculate joint configurations for a desired end-effector pose based on a specified rigid body tree model. Create a rigid body tree model for your robot using the `robotics.RigidBodyTree` class. The rigid body tree model defines all the joint constraints that the solver enforces.

Specify the `RigidBodyTree` object and the desired end effector inside the block mask. You can also tune the algorithm parameters in the **Solver Parameters** tab.

Input the desired end-effector **Pose**, the **Weights** on pose tolerance, and an **InitialGuess** for the joint configuration. The solver outputs a robot configuration, **Config**, that satisfies the end-effector pose within the tolerances specified in the **Solver Parameters** tab.

### Ports

#### Input

##### **Pose — End-effector pose**

4-by-4 homogeneous transform

End-effector pose, specified as a 4-by-4 homogeneous transform. This transform defines the desired position and orientation of the rigid body specified in the **End effector** parameter.

Data Types: `single` | `double`

### **Weights — Weights for pose tolerances**

six-element vector

Weights for pose tolerances, specified as a six-element vector. The first three elements of the vector correspond to the weights on the error in orientation for the desired pose. The last three elements of the vector correspond to the weights on the error in the xyz position for the desired pose.

Data Types: `single` | `double`

### **InitialGuess — Initial guess of robot configuration**

vector

Initial guess of robot configuration, specified as a vector of joint positions. The number of positions is equal to the number of nonfixed joints in the **Rigid body tree** parameter. Use this initial guess to help guide the solver to a desired robot configuration. However, the solution is not guaranteed to be close to this initial guess.

Data Types: `single` | `double`

## **Output**

### **Config — Robot configuration solution**

vector

Robot configuration that solves the desired end-effector pose, specified as a vector. A robot configuration is a vector of joint positions for the rigid body tree model. The number of positions is equal to the number of nonfixed joints in the **Rigid body tree** parameter.

Data Types: `single` | `double`

### **Info — Solution information**

bus

Solution information, returned as a bus. The solution information bus contains these elements:

- **Iterations** — Number of iterations run by the algorithm.
- **PoseErrorNorm** — The magnitude of the error between the pose of the end effector in the solution and the desired end-effector pose.
- **ExitFlag** — Code that gives more details on the algorithm execution and what caused it to return. For the exit flags of each algorithm type, see “Exit Flags”.

- **Status** — Character vector describing whether the solution is within the tolerance (1) or is the best possible solution the algorithm could find (2).

## Parameters

### Block Parameters

#### **Rigid body tree — Rigid body tree model**

twoJointRigidBodyTree (default) | RigidBodyTree object

Rigid body tree model, specified as a RigidBodyTree object. Create the robot model in the MATLAB workspace before specifying in the block mask.

#### **End effector — End-effector name**

'tool' | Select body

End-effector name for desired pose. To see a list of bodies on the RigidBodyTree object, specify the **Rigid body tree** parameter, then click **Select body**.

#### **Show solution diagnostic outputs — Enable info port**

on (default) | off

Select to enable the **Info** port and get diagnostic info for the solver solution.

### Solver Parameters

#### **Solver — Algorithm for solving inverse kinematics**

'BFGSGradientProjection' (default) | 'LevenbergMarquardt'

Algorithm for solving inverse kinematics, specified as either 'BFGSGradientProjection' or 'LevenbergMarquardt'. For details of each algorithm, see “Inverse Kinematics Algorithms”.

#### **Enforce joint limits — Enforce rigid body tree joint limits**

on (default) | off

Select to enforce the joint limits specified in the **Rigid body tree** model.

#### **Maximum iterations — Maximum number of iterations**

1500 (default) | positive integer

Maximum number of iterations to optimize the solution, specified as a positive integer. Increasing the number of iterations can improve the solution at the cost of execution time.

**Maximum time — Maximum time**

10 (default) | positive scalar

Maximum number of seconds that the algorithm runs before timing out, specified as a positive scalar. Increasing the maximum time can improve the solution at the cost of execution time.

**Gradient tolerance — Threshold on gradient of cost function**

1e-7 (default) | positive scalar

Threshold on the gradient of the cost function, specified as a positive scalar. The algorithm stops if the magnitude of the gradient falls below this threshold. A low gradient magnitude usually indicates that the solver has converged to a solution.

**Solution tolerance — Threshold on pose error**

1e-6 (default) | positive scalar

Threshold on the magnitude of the error between the end-effector pose generated from the solution and the desired pose, specified as a positive scalar. The `Weights` specified for each component of the pose are included in this calculation.

**Step tolerance — Minimum step size**

1e-14 (default) | positive scalar

Minimum step size allowed by the solver, specified as a positive scalar. Smaller step sizes usually mean that the solution is close to convergence.

**Error change tolerance — Threshold on change in pose error**

1e-12 (default) | positive scalar

Threshold on the change in end-effector pose error between iterations, specified as a positive scalar. The algorithm returns if the changes in all elements of the pose error are smaller than this threshold.

**Dependencies**

This parameter is enabled when the `Solver` is Levenberg-Marquadt.

### Use error damping — Enable error damping

on (default) | off

Select the check box to enable error damping, then specify the **Damping bias** parameter.

#### Dependencies

This parameter is enabled when the **Solver** is Levenberg-Marquadt.

### Damping bias — Damping on cost function

0.0025 (default) | positive scalar

Damping on cost function, specified as a positive scalar. The Levenberg-Marquadt algorithm has a damping feature controlled by this scalar that works with the cost function to control the rate of convergence.

#### Dependencies

This parameter is enabled when the **Solver** is Levenberg-Marquadt and **Use error damping** is on.

### Simulate using — Type of simulation to run

Interpreted execution (default) | Code generation

- **Interpreted execution** — Simulate model using the MATLAB interpreter. This option shortens startup time but has a slower simulation speed than **Code generation**. In this mode, you can debug the source code of the block.
- **Code generation** — Simulate model using generated C code. The first time you run a simulation, Simulink generates C code for the block. The C code is reused for subsequent simulations, as long as the model does not change. This option requires additional startup time, but the speed of the subsequent simulations is comparable to **Interpreted execution**.

**Tunable:** No

## References

- [1] Badreddine, Hassan, Stefan Vandewalle, and Johan Meyers. "Sequential Quadratic Programming (SQP) for Optimal Control in Direct Numerical Simulation of Turbulent Flow." *Journal of Computational Physics*. 256 (2014): 1-16. doi:10.1016/j.jcp.2013.08.044.



- [2] Bertsekas, Dimitri P. *Nonlinear Programming*. Belmont, MA: Athena Scientific, 1999.
- [3] Goldfarb, Donald. "Extension of Davidon's Variable Metric Method to Maximization Under Linear Inequality and Equality Constraints." *SIAM Journal on Applied Mathematics*. Vol. 17, No. 4 (1969): 739-64. doi:10.1137/0117067.
- [4] Nocedal, Jorge, and Stephen Wright. *Numerical Optimization*. New York, NY: Springer, 2006.
- [5] Sugihara, Tomomichi. "Solvability-Unconcerned Inverse Kinematics by the Levenberg-Marquardt Method." *IEEE Transactions on Robotics*. Vol. 27, No. 5 (2011): 984-91. doi:10.1109/tro.2011.2148230.
- [6] Zhao, Jianmin, and Norman I. Badler. "Inverse Kinematics Positioning Using Nonlinear Programming for Highly Articulated Figures." *ACM Transactions on Graphics*. Vol. 13, No. 4 (1994): 313-36. doi:10.1145/195826.195827.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

### See Also

#### Objects

`robotics.GeneralizedInverseKinematics` | `robotics.InverseKinematics` | `robotics.RigidBodyTree`

#### Blocks

`Get Transform` | `Inverse Dynamics`

### Topics

"Trajectory Control Modeling With Inverse Kinematics"

"Control PR2 Arm Movements Using ROS Actions and Inverse Kinematics"

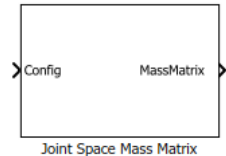
"Inverse Kinematics Algorithms"

**Introduced in R2018b**

# Joint Space Mass Matrix

Joint-space mass matrix for robot configuration

**Library:** Robotics System Toolbox / Manipulator Algorithms



## Description

The Joint Space Mass Matrix block returns the joint-space mass matrix for the given robot configuration (joint positions) for the **Rigid body tree** robot model.

## Ports

### Input

#### **Config** — Robot configuration

vector

Robot configuration, specified as a vector of positions for all nonfixed joints in the robot model, as set by the **Rigid body tree** parameter. You can also generate this vector for a complex robot using the `homeConfiguration` or `randomConfiguration` functions inside a Constant or MATLAB Function block.

### Output

#### **MassMatrix** — Joint-space mass matrix for configuration

positive-definite symmetric matrix

Joint-space mass matrix for the given robot configuration, returned as a positive-definite symmetric matrix.

## Parameters

### Rigid body tree — Robot model

`twoJointRigidBodyTree` (default) | `RigidBodyTree` object

Robot model, specified as a `RigidBodyTree` object. You can also import a robot model from an URDF (Unified Robot Description Formation) file using `importrobot`.

The default robot model, `twoJointRigidBodyTree`, is a robot with revolute joints and two degrees of freedom.

### Simulate using — Type of simulation to run

`Interpreted execution` (default) | `Code generation`

- `Interpreted execution` — Simulate model using the MATLAB interpreter. This option shortens startup time but has a slower simulation speed than `Code generation`. In this mode, you can debug the source code of the block.
- `Code generation` — Simulate model using generated C code. The first time you run a simulation, Simulink generates C code for the block. The C code is reused for subsequent simulations, as long as the model does not change. This option requires additional startup time, but the speed of the subsequent simulations is comparable to `Interpreted execution`.

**Tunable:** No

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

## See Also

### Blocks

[Forward Dynamics](#) | [Get Jacobian](#) | [Gravity Torque](#) | [Inverse Dynamics](#) | [Velocity Product Torque](#)

**Classes**

RigidBodyTree

**Functions**

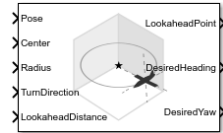
homeConfiguration | importrobot | massMatrix | randomConfiguration

**Introduced in R2018a**

# Orbit Follower

Orbit location of interest using UAV

**Library:** Robotics System Toolbox UAV Library



## Description

The Orbit Follower block generates heading 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 heading] vector

Current UAV pose, specified as an [x y z heading] vector. [x y z] is the UAV's position in NED coordinates (north-east-down) specified in meters. heading 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

#### **DesiredHeading — Desired heading**

numeric scalar

Desired heading, returned as numeric scalar in radians in the range of  $[-\pi, \pi]$ . The UAV heading is the angle of direction of the velocity vector relative to north measured in radians.

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.

Data Types: double

#### **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.

### 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 | roboticsAddons  
| state

#### Objects

fixedwing | multicopter | uavOrbitFollower | uavWaypointFollower

### Topics

“Approximate High-Fidelity UAV model with UAV Guidance Model block”

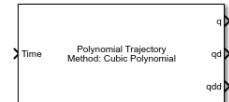
“Tuning Waypoint Follower for Fixed-Wing UAV”

**Introduced in R2019a**

# Polynomial Trajectory

Generate polynomial trajectories through waypoints

**Library:** Robotics System Toolbox / Utilities



## Description

The Polynomial Trajectory block generates trajectories to travel through waypoints at the given time points using either cubic, quintic, or B-spline polynomials. The block outputs positions, velocities, and accelerations for achieving this trajectory based on the **Time** input. For B-spline polynomials, the waypoints actually define the control points for the convex hull of the B-spline instead of the actual waypoints, but the first and last waypoint are still met.

The initial and final values are held constant outside the time period defined in **Time points**.

## Ports

### Input

#### **Time** — Time point along trajectory

scalar | vector

Time point along the trajectory, specified as a scalar or vector. In general, when specified as a scalar, this value is synced with simulation time and is used to specify the time point for sampling the trajectory. The block outputs a vector of the trajectory variables at that instant in time. If the time is specified as a vector, the block outputs a matrix with each column corresponding to each element of the vector.

Data Types: single | double

#### **Waypoints** — Waypoint positions along trajectory

$n$ -by- $p$  matrix

Positions of waypoints of the trajectory at given time points, specified as an  $n$ -by- $p$  matrix, where  $n$  is the dimension of the trajectory and  $p$  is the number of waypoints. If you specify the **Method** as B-spline, these waypoints actually define the control points for the convex hull of the B-spline, but the first and last waypoint are still met.

### Dependencies

To enable this input, set **Waypoint Source** to External.

### TimePoints — Time points for waypoints of trajectory

$p$ -element vector

Time points for waypoints of trajectory, specified as a  $p$ -element vector.

### Dependencies

To enable this input, set **Waypoint Source** to External.

### VelBC — Velocity boundary conditions for waypoints

$n$ -by- $p$  matrix

Velocity boundary conditions for waypoints, specified as an  $n$ -by- $p$  matrix. Each row corresponds to the velocity at each of the  $p$  waypoints for the respective variable in the trajectory.

### Dependencies

To enable this input, set **Method** to Cubic Polynomial or Quintic Polynomial and **Parameter Source** to External.

### AccelBC — Acceleration boundary conditions for trajectory

$n$ -by- $p$  matrix

Acceleration boundary conditions for waypoints, specified as an  $n$ -by- $p$  matrix. Each row corresponds to the acceleration at each of the  $p$  waypoints for the respective variable in the trajectory.

### Dependencies

To enable this parameter, set **Method** to Quintic Polynomial and **Parameter Source** to External.

## Output

### **q** — Position of trajectory

scalar | vector | matrix

Position of the trajectory, specified as a scalar, vector, or matrix. If you specify a scalar for the **Time** input with an  $n$ -dimensional trajectory, the output is a vector with  $n$  elements. If you specify a vector of  $m$  elements for the **Time** input, the output is an  $n$ -by- $m$  matrix.

Data Types: single | double

### **qd** — Velocity of trajectory

scalar | vector | matrix

Velocity of the trajectory, specified as a scalar, vector, or matrix. If you specify a scalar for the **Time** input with an  $n$ -dimensional trajectory, the output is a vector with  $n$  elements. If you specify a vector of  $m$  elements for the **Time** input, the output is an  $n$ -by- $m$  matrix.

Data Types: single | double

### **qdd** — Acceleration of trajectory

scalar | vector | matrix

Acceleration of the trajectory, specified as a scalar, vector, or matrix. If you specify a scalar for the **Time** input with an  $n$ -dimensional trajectory, the output is a vector with  $n$  elements. If you specify a vector of  $m$  elements for the **Time** input, the output is an  $n$ -by- $m$  matrix.

Data Types: single | double

## Parameters

### **Waypoint source** — Source for waypoints

Internal (default) | External

Specify **External** to specify the **Waypoints** and **Time points** parameters as block inputs instead of block parameters.

### **Waypoints** — Waypoint positions along trajectory

$n$ -by- $p$  matrix

Positions of waypoints of the trajectory at given time points, specified as an  $n$ -by- $p$  matrix, where  $n$  is the dimension of the trajectory and  $p$  is the number of waypoints. If you specify

the **Method** as B-spline, these waypoints actually define the control points for the convex hull of the B-spline, but the first and last waypoint are still met.

### **Dependencies**

To specify this parameter in the block mask, set **Waypoint Source** to Internal.

### **Time points — Time points for waypoints of trajectory**

*p*-element vector

Time points for waypoints of trajectory, specified as a *p*-element vector, where *p* is the number of waypoints.

### **Dependencies**

To specify this parameter in the block mask, set **Waypoint Source** to Internal.

### **Method — Method for trajectory generation**

Cubic Polynomial (default) | Quintic Polynomial | B-Spline

Method for trajectory generation, specified as either Cubic Polynomial, Quintic Polynomial, or B-Spline.

### **Parameter source — Source for waypoints**

Internal (default) | External

Specify External to specify the **Velocity boundary conditions** and **Acceleration boundary conditions** parameters as block inputs instead of block parameters.

### **Velocity boundary conditions — Velocity boundary conditions for waypoints**

zeroes(2,5) (default) | *n*-by-*p* matrix

Velocity boundary conditions for waypoints, specified as an *n*-by-*p* matrix. Each row corresponds to the velocity at each of the *p* waypoints for the respective variable in the trajectory.

### **Dependencies**

To enable this input, set **Method** to Cubic Polynomial or Quintic Polynomial.

### **Acceleration boundary conditions — Acceleration boundary conditions for trajectory**

*n*-by-*p* matrix

Acceleration boundary conditions for waypoints, specified as an  $n$ -by- $p$  matrix. Each row corresponds to the acceleration at each of the  $p$  waypoints for the respective variable in the trajectory.

### Dependencies

To enable this parameter, set **Method** to `Quintic Polynomial`.

### 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`.

## References

- [1] Farin, Gerald E. *Curves and Surfaces for Computer Aided Geometric Design: A Practical Guide*. San Diego, CA: Academic Press, 1993.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

## See Also

### Blocks

`Rotation Trajectory` | `Transform Trajectory` | `Trapezoidal Velocity Profile Trajectory`

**Functions**

bsplinepolytraj | cubicpolytraj | quinticpolytraj | rottraj |  
transformtraj | trapveltraj

**Introduced in R2019a**



# Publish

Send messages to ROS network

**Library:** Robotics System Toolbox / ROS



## Description

The Publish block takes in as its input a Simulink nonvirtual bus that corresponds to the specified ROS message type and publishes it to the ROS network. It uses the node of the Simulink model to create a ROS publisher for a specific topic. This node is created when the model runs and is deleted when the model terminates. If the model does not have a node, the block creates one.

On each sample hit, the block converts the **Msg** input from a Simulink bus signal to a ROS message and publishes it. The block does not distinguish whether the input is a new message but merely publishes it on every sample hit. For simulation, this input is a MATLAB ROS message. In code generation, it is a C++ ROS message.

## Input/Output Ports

### Input

#### **Msg — ROS message**

nonvirtual bus

ROS message, specified as a nonvirtual bus. To specify the type of ROS message, use the **Message type** parameter.

Data Types: bus

## Parameters

### Topic source — Source for specifying topic name

Select from ROS network | Specify your own

Source for specifying the topic name, specified as one of the following:

- **Select from ROS network** — Use **Select** to select a topic name. The **Topic** and **Message type** parameters are set automatically. You must be connected to a ROS network.
- **Specify your own** — Enter a topic name in **Topic** and specify its message type in **Message type**. You must match a topic name exactly.

### Topic — Topic name to publish to

string

Topic name to publish to, specified as a string. When **Topic source** is set to **Select from ROS network**, use **Select** to select a topic from the ROS network. You must be connected to a ROS network to get a list of topics. Otherwise, set **Topic source** to **Specify your own** and specify the topic you want.

### Message type — ROS message type

string

ROS message type, specified as a string. Use **Select** to select from a full list of supported ROS messages. Service message types are not supported and are not included in the list.

### Length of publish queue — Message queue length

1 (default) | integer

Message queue length in code generation, specified as an integer. In simulation, the message queue is always 1 and cannot be adjusted. To ensure each message is processed, use a smaller model step or only execute the model when publishing a new message.

## Tips

You can also set the addresses for the ROS master and node host by clicking the **Configure network addresses** link in the block.

## **Extended Capabilities**

### **C/C++ Code Generation**

Generate C and C++ code using Simulink® Coder™.

### **See Also**

[Blank Message](#) | [Subscribe](#)

### **Topics**

[“Virtual and Nonvirtual Buses” \(Simulink\)](#)

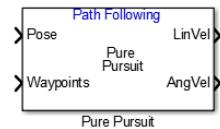
[“Simulink and ROS Interaction”](#)

**Introduced in R2015a**

## Pure Pursuit

Linear and angular velocity control commands

**Library:** Robotics System Toolbox / Mobile Robot Algorithms



## Description

The Pure Pursuit block computes linear and angular velocity commands for following a path using a set of waypoints and the current pose of a differential drive robot. The block takes updated poses to update velocity commands for the robot to follow a path along a desired set of waypoints. Use the **Max angular velocity** and **Desired linear velocity** parameters to update the velocities based on the performance of the robot.

The **Lookahead distance** parameter computes a look-ahead point on the path, which is an instantaneous local goal for the robot. The angular velocity command is computed based on this point. Changing **Lookahead distance** has a significant impact on the performance of the algorithm. A higher look-ahead distance results in a smoother trajectory for the robot, but can cause the robot to cut corners along the path. Too low of a look-ahead distance can result in oscillations in tracking the path, causing unstable behavior. For more information on the pure pursuit algorithm, see “Pure Pursuit Controller”.

## Input/Output Ports

### Input

#### Pose — Current robot pose

[x y theta] vector

Current robot pose, specified as an [x y theta] vector, which corresponds to the x-y position and orientation angle, *theta*. Positive angles are measured counterclockwise from the positive x-axis.

### Waypoints — Waypoints

[ ] (default) |  $n$ -by-2 array

Waypoints, specified as an  $n$ -by-2 array of [x y] pairs, where  $n$  is the number of waypoints. You can generate the waypoints from the `robotics.PRM` class or specify them as an array in Simulink.

## Output

### LinVel — Linear velocity

scalar in meters per second

Linear velocity, specified as a scalar in meters per second.

Data Types: double

### AngVel — Angular velocity

scalar in radians per second

Angular velocity, specified as a scalar in radians per second.

Data Types: double

### TargetDir — Target direction for robot

scalar in radians

Target direction for the robot, specified as a scalar in radians. The forward direction of the robot is considered zero radians, with positive angles measured counterclockwise. This output can be used as the input to the **TargetDir** port for the Vector Field Histogram block.

### Dependencies

To enable this port, select the **Show TargetDir output port** parameter.

## Parameters

### Desired linear velocity (m/s) — Linear velocity

0.1 (default) | scalar

Desired linear velocity, specified as a scalar in meters per second. The controller assumes that the robot drives at a constant linear velocity and that the computed angular velocity is independent of the linear velocity.

### **Maximum angular velocity (rad/s) — Angular velocity**

1.0 (default) | scalar

Maximum angular velocity, specified as a scalar in radians per second. The controller saturates the absolute angular velocity output at the given value.

### **Lookahead distance (m) — Look-ahead distance**

1.0 (default) | scalar

Look-ahead distance, specified as a scalar in meters. The look-ahead distance changes the response of the controller. A robot with a higher look-ahead distance produces smooth paths but takes larger turns at corners. A robot with a smaller look-ahead distance follows the path closely and takes sharp turns, but oscillate along the path. For more information on the effects of look-ahead distance, see “Pure Pursuit Controller”.

### **Show TargetDir output port — Target direction indicator**

off (default) | on

Select this parameter to enable the **TargetDir** out port. This port gives the target direction as an angle in radians from the forward position, with positive angles measured counterclockwise.

## **Extended Capabilities**

### **C/C++ Code Generation**

Generate C and C++ code using Simulink® Coder™.

## **See Also**

### **Blocks**

[Publish](#) | [Subscribe](#) | [Vector Field Histogram](#)

### **Classes**

robotics.PRM | robotics.PurePursuit

### **Topics**

“Path Following for a Differential Drive Robot”

“Path Following with Obstacle Avoidance in Simulink®”

“Pure Pursuit Controller”

**Introduced in R2016b**

## Read Data

Play back data from log file

**Library:** Robotics System Toolbox / ROS



## Description

The Read Data block plays back rosbag logfiles by outputting the most recent message from the log file based on the current simulation time. You must load a rosbag log file (.bag) and specify the **Topic** in the block mask to get a stream of messages from the file. Messages on this topic are output from the file in sync with the simulation time.

In the Read Data block mask, click **Load log file data** to specify a rosbag log file (.bag) to load. In the **Load Log File** window, specify a **Start time offset**, in seconds, to start playback at a certain point in the file. **Duration** specifies how long the block should play back this file in seconds. By default, the block outputs all messages for the specific **Topic** in the file.

## Ports

### Output

#### IsNew — New message indicator

0 | 1

New message indicator, returned as a logical. If the output is 1, then a new message was loaded from the rosbag file at that time. This output can be used to trigger subsystems for processing new messages received.

#### Msg — ROS message

nonvirtual bus



ROS message, returned as a nonvirtual bus. Messages are output in the order they are stored in the rosbag and synced with the simulation time.

Data Types: bus

## Parameters

### Topic — Topic name to extract from log file

string

Topic name to extract from log file, specified as a string. This topic must exist in the loaded rosbag. Click the **Load rosbag file** Use **Select ...** to inspect the topics available and select a specific topic.

### 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).

## See Also

### Blocks

[Publish](#) | [Read Image](#) | [Read Point Cloud](#) | [Subscribe](#)

### Functions

[readMessages](#) | [rosbag](#) | [select](#)

### Topics

“Work with ROS Messages in Simulink®”

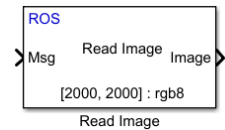
“Work with rosbag Logfiles”

### Introduced in R2018b

## Read Image

Extract image from ROS Image message

**Library:** Robotics System Toolbox / ROS



## Description

The Read Image block extracts an image from a ROS `Image` or `CompressedImage` message. You can select the ROS message parameters of a topic active on a live ROS network or specify the message parameters separately. The ROS messages are specified as a nonvirtual bus. Use the Subscribe block output to receive a message from a ROS network and input the message to the Read Image block.

---

**Note** When reading ROS image messages from the network, the `Data` property of the message can exceed the maximum array length set in Simulink. To increase the maximum array length, click **Tools > Manage Array Lengths > Robot Operating System**, select the **Data** array, and increase the size based on the number of points in the image.

---

## Ports

### Input

**Msg — ROS Image or CompressedImage message**

nonvirtual bus

ROS `Image` or `CompressedImage` message, specified as a nonvirtual bus. You can use the Subscribe block to get a message from an active ROS network.

Data Types: bus

## Output

### Image — Extracted image signal

*M*-by-*N*-by-3 matrix | *M*-by-*N* matrix

Extracted image signal from ROS message, returned as an *M*-by-*N*-by-3 matrix for color images, and an *M*-by-*N* matrix for grayscale images. The matrix contains the pixel data from the `Data` property of the ROS message.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16`

### AlphaChannel — Alpha channel for image

*M*-by-*N* matrix

Alpha channel for image, returned as an *M*-by-*N* matrix. This matrix is the same height and width as the image output and has values [0 1] to indicate the opacity of each corresponding pixel, with a value of 0 being completely transparent.

---

**Note** For `CompressedImage` messages, the Alpha channel returns all zeros if the `Show Alpha output port` is enabled.

---

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16`

### ErrorCode — Error code for image conversion

scalar

Error code for image conversion, returned as a scalar. The error code values are:

- 0 - Successfully converted the image message.
- 1 - Incorrect image encoding. Check that the incoming message encoding matches the `ImageEncoding` parameter.
- 2 - The dimensions of the image message exceed the limits specified in the `Maximum Image Size` parameter.
- 3 - The `Data` field of the image message was truncated. See “Managing Array Sizes in Simulink ROS” to increase the maximum length of the array.
- 04 - Image decompression failed.

Data Types: `uint8`

## Parameters

### Maximum Image Size — Maximum image size

[2000 2000] (default) | two-element vector

Maximum image size, specified as a two-element [height width] vector.

Click **Configure using ROS ...** to set this parameter automatically using an active topic on a ROS network. You must be connected to the ROS network.

### Image Encoding — Image encoding

rgb8 (default) | rgba8 | ...

Image encoding for the input ImageMsg. Select the supported encoding type which matches the Encoding property of the message. For more information about encoding types, see readImage.

### Show Alpha output port — Toggle AlphaChannel port

off (default) | on

Toggle Alpha channel output port if your encoding supports an Alpha channel.

### Dependencies

Only certain encoding types support alpha channels. The ImageEncoding parameter determines if this parameter appears in the block mask.

### Show error code output port — Toggle ErrorCode port

on (default) | off

Toggle the ErrorCode port to monitor errors.

### Output variable-size signals — Toggle variable-size signal output

off (default) | on

Toggle variable-size signal output. Variable-sized signals should only be used if the image size is expected to change over time. For more information about variable sized signals, see “Variable-Size Signal Basics” (Simulink).

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

### See Also

[Blank Message](#) | [CompressedImage](#) | [Image](#) | [Subscribe](#) | [readImage](#)

### Topics

[“Managing Array Sizes in Simulink ROS”](#)

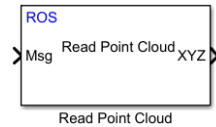
[“Variable-Size Signal Basics” \(Simulink\)](#)

**Introduced in R2017b**

## Read Point Cloud

Extract point cloud from ROS PointCloud2 message

**Library:** Robotics System Toolbox / ROS



### Description

The Read Point Cloud block extracts a point cloud from a ROS `PointCloud2` message. You can select the ROS message parameters of a topic active on a live ROS network or specify the message parameters separately. The ROS messages are specified as a nonvirtual bus. Use the Subscribe block to receive a message from a ROS network and input the message to the Read Point Cloud block.

---

**Note** When reading ROS point cloud messages from the network, the `Data` property of the message can exceed the maximum array length set in Simulink. To increase the maximum array length, click **Tools > Manage Array Lengths > Robot Operating System**, select the **Data** array, and increase the size based on the number of points in the point cloud.

---

### Ports

#### Input

**Msg — ROS PointCloud2 message**

nonvirtual bus

ROS `PointCloud2` message, specified as a nonvirtual bus. You can use the Subscribe block to get a message from the ROS network.

Data Types: bus

## Output

### XYZ — XYZ coordinates

matrix | multidimensional array

$x$ ,  $y$ , and  $z$  coordinates of the point cloud data, output as either an  $N$ -by-3 matrix or  $h$ -by- $w$ -by-3 multidimensional array.  $N$  is the number of points.  $h$  and  $w$  are the height and width of the image in pixels. To get the  $x$ ,  $y$ , and  $z$  coordinates as a multidimensional array, select the `Preserve point cloud structure` check box in the block mask parameters.

Data Types: `single`

### RGB — RGB values for each point

matrix | multidimensional array

RGB values for each point of the point cloud data, output as either an  $N$ -by-3 matrix or  $h$ -by- $w$ -by-3 multidimensional array.  $N$  is the number of points.  $h$  and  $w$  are the height and width of the image in pixels. The RGB values correspond to the red, green, and blue color intensities with a range of  $[0\ 1]$ . To get the RGB values as a multidimensional array, select the `Preserve point cloud structure` check box in the block mask parameters.

Data Types: `double`

### Intensity — Intensity values for each point

array | matrix

Intensity values for each point of the point cloud data, output as either an array or a  $h$ -by- $w$  matrix.  $h$  and  $w$  are the height and width of the image in pixels. To get the intensity values as a matrix, select the `Preserve point cloud structure` check box in the block mask parameters.

Data Types: `single`

### ErrorCode — Error code for image conversion

scalar

Error code for image conversion, returned as a scalar. The error code values are:

- 0 - Successfully converted the point cloud message.
- 1 - The dimensions of the incoming point cloud exceed the limits set in `Maximum point cloud size`.

- 2 - One of the variable-length arrays in the incoming message was truncated. See “Managing Array Sizes in Simulink ROS” to increase the maximum length of the array.
- 3 - The X, Y, or Z field of the point cloud message is missing.
- 4 -The point cloud does not contain any RGB color data. You must have toggled Show RGB output port to on to get this error .
- 5 -The point cloud does not contain any intensity data. You must have toggled Show Intensity output port to on to get this error.
- 6 - The X, Y, or Z field of the point cloud message does not have the correct data type (float32).
- 7 - The RGB field of the point cloud message does not have the correct data type (float32).
- 8 - The Intensity field of the point cloud message does not have the correct data type (float32).

For certain error codes, data is truncated or populated with NaN values where appropriate.

Data Types: uint8

## Parameters

### Maximum point cloud size — Maximum point cloud image size

[480 640] (default) | two-element vector

Maximum point cloud image size, specified as a two-element [height width] vector.

Click **Configure using ROS ...** to set this parameter automatically using an active topic on a ROS network. You must be connected to the ROS network.

### Preserve point cloud structure — Preserve point cloud data output shape

off (default) | on

When this check box is selected, the cloud data output shape for XYZ, RGB, and Intensity are preserved. The outputs maintain the structure of the original image. Therefore, XYZ and RGB are output as multidimensional arrays, and Intensity is output as a matrix.

### Show RGB output port — Toggle RGB port

on (default) | off



Select this check box to get RGB values for each point of the point cloud message from the RGB port. The RGB data must be supplied by the message.

**Show Intensity output port — Toggle Intensity port**

off (default) | on

Select this check box to get intensity values for each point of the point cloud message from the Intensity port. The intensity data must be supplied by the message.

**Show error code output port — Toggle ErrorCode port**

off (default) | on

Select this check box to monitor errors with the ErrorCode port.

**Output variable-size signals — Toggle variable-size signal output**

off (default) | on

Select this check box to output variable-size signals. Variable-sized signals should only be used if the image size is expected to change over time. For more information about variable sized signals, see “Variable-Size Signal Basics” (Simulink).

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

### See Also

[Blank Message](#) | [PointCloud2](#) | [Subscribe](#)

### Topics

“Managing Array Sizes in Simulink ROS”

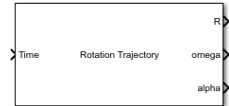
“Variable-Size Signal Basics” (Simulink)

**Introduced in R2017b**

# Rotation Trajectory

Generate trajectory between two orientations

**Library:** Robotics System Toolbox / Utilities



## Description

The Rotation Trajectory block generates an interpolated trajectory between two rotation matrices. The block outputs the rotation at the times given by the **Time** input, which can be a scalar or vector.

The trajectory is computed using quaternion spherical linear interpolation (SLERP) and finds the shortest path between points. Select the **Use custom time scaling** check box to compute using a custom time scaling. The block uses linear time scaling by default.

The initial and final values are held constant outside the time period defined in the **Time interval** parameter.

## Ports

### Input

#### Time — Time point along trajectory

scalar | vector

Time point along the trajectory, specified as a scalar or vector. In general, when specified as a scalar, this value is synced with simulation time and is used to specify the time point for sampling the trajectory. The block outputs a vector of the trajectory variables at that instant in time. If the time is specified as a vector, the block outputs a matrix with each column corresponding to each element of the vector.

Data Types: single | double

**R0 — Initial orientation**

four-element quaternion vector | 3-by-3 rotation matrix

Initial orientation, specified as a four-element quaternion vector or 3-by-3 rotation matrix. The function generates a trajectory that starts at the initial orientation, **R0**, and goes to the final orientation, **RF**.

Example: `[1 0 0 0]'`**Dependencies**

To enable this input, set the **Waypoint source** to External.

To specify quaternions, set **Rotation Format** parameter to Quaternion.

To specify rotation matrices, set **Rotation Format** parameter to Rotation.

Data Types: `single` | `double`**RF — Final orientation**

four-element vector | 3-by-3 rotation matrix

Initial orientation, specified as a four-element vector or 3-by-3 rotation matrix. The function generates a trajectory that starts at the initial orientation, **R0**, and goes to the final orientation, **RF**.

Example: `[0 0 1 0]'`**Dependencies**

To enable this input, set the **Waypoint source** to External.

To specify quaternions, set **Rotation Format** parameter to Quaternion.

To specify rotation matrices, set **Rotation Format** parameter to Rotation.

Data Types: `single` | `double`**TimeInterval — Start and end times for trajectory**

two-element vector

Start and end times for the trajectory, specified as a two-element vector.

Example: `[0 10]`

**Dependencies**

To enable this input, set the **Waypoint source** to External.

Data Types: single | double

**TSTime — Time scaling time points**

scalar |  $p$ -element vector

Time scaling time points, specified as a scalar or  $n$   $p$ -element vector, where  $p$  is the number of points for time scaling. By default, the time scaling is a linear time scaling spanning the **TimeInterval**. Specify the actual time scaling values in **TimeScaling**.

If the **Time** input is specified at a time not specified by these points, interpolation is used to find the right scaling time.

**Dependencies**

To enable this parameter, select the **Use custom time scaling** check box and set **Parameter source** to External.

To specify a scalar, the **Time** input must be a scalar.

Data Types: single | double

**TimeScaling — Time scaling vector and first two derivatives**

three-element vector | 3-by- $p$  matrix

Time scaling vector and its first two derivatives, specified as a three element vector or a 3-by- $p$  matrix, where  $m$  is the length of **TSTime**. By default, the time scaling is a linear time scaling spanning the **TimeInterval**.

For a nonlinear time scaling, specify the values of the time points in the first row. The second and third rows are the velocity and acceleration of the time points, respectively. For example, to follow the path with a linear velocity to the halfway point, and then jump to the end, the time-scaling would be:

```
s(1,:) = [0 0.25 0.5 1 1 1] % Position  
s(2,:) = [1 1 1 0 0 0] % Velocity  
s(3,:) = [0 0 0 0 0 0] % Acceleration
```

**Dependencies**

To enable this parameter, select the **Use custom time scaling** check box and set **Parameter source** to External.

To specify a three-element vector, the **Time** and **TSTime** inputs must be a scalar.

Data Types: `single` | `double`

## Output

### **R** — Orientation vectors

4-by- $m$  quaternion array | 3-by-3-by- $m$  rotation matrix array

Orientation vectors, returned as a 4-by- $m$  quaternion array or 3-by-3-by- $m$  rotation matrix array, where  $m$  is the number of points in the input to **Time**.

### Dependencies

To get a quaternion array, set **Rotation Format** parameter to `Quaternion`.

To get a rotation matrix array, set **Rotation Format** parameter to `Rotation`.

### **omega** — Orientation angular velocity

3-by- $m$  matrix

Orientation angular velocity, returned as a 3-by- $m$  matrix, where  $m$  is the number of points in the input to **Time**.

### **alpha** — Orientation angular acceleration

3-by- $m$  matrix

Orientation angular acceleration, returned as a 3-by- $m$  matrix, where  $m$  is the number of points in the input to **Time**.

## Parameters

### **Rotation format** — Format for orientations

`Quaternion` (default) | `Rotation Matrix`

Select `Rotation Matrix` to specify the **Initial rotation** and **Final rotation** as 3-by-3 rotation matrices and get the orientation output (port **R**) as a rotation matrix array. By default, the initial and final rotations are specified as four-element quaternion vectors.

### **Waypoint source** — Source for waypoints

`Internal` (default) | `External`

Specify **External** to specify the **Initial rotation**, **Final rotation**, and **Time interval** parameters as block inputs instead of block parameters.

### **Initial rotation — Initial orientation**

[1 0 0 0]' (default) | four-element quaternion vector | 3-by-3 rotation matrix

Initial orientation, specified as a four-element quaternion vector or 3-by-3 rotation matrix. The function generates a trajectory that starts at the **Initial rotation** and goes to the **Final rotation**.

#### **Dependencies**

To specify quaternions, set **Rotation Format** parameter to Quaternion.

To specify rotation matrices, set **Rotation Format** parameter to Rotation.

Data Types: single | double

### **Final rotation — Final orientation**

[0 0 1 0]' (default) | four-element vector | 3-by-3 rotation matrix

Final orientation, specified as a four-element vector or 3-by-3 rotation matrix. The function generates a trajectory that starts at the **Initial rotation** and goes to the **Final rotation**.

#### **Dependencies**

To specify quaternions, set **Rotation Format** parameter to Quaternion.

To specify rotation matrices, set **Rotation Format** parameter to Rotation.

Data Types: single | double

### **Time interval — Start and end times for trajectory**

[0 10] (default) | two-element vector

Start and end times for the trajectory, specified as a two-element vector.

Data Types: single | double

### **Use custom time scaling — Enable custom time scaling**

off (default) | on

Enable to specify custom time scaling for the trajectory using the **Parameter Source**, **Time scaling time**, and **Time scaling values** parameters.

**Parameter source — Source for waypoints**

Internal (default) | External

Specify External to specify the **Time scaling time** and **Time scaling values** parameters as block inputs instead of block parameters.

**Dependencies**

To enable this parameter, select the **Use custom time scaling** check box.

**Time scaling time — Time scaling time points**2:0.1:3 (default) | scalar |  $p$ -element vector

Time scaling time points, specified as a scalar or  $p$ -element vector, where  $p$  is the number of points for time scaling. By default, the time scaling is a linear time scaling spanning the **Time interval**. Specify the actual time scaling values in **Time scaling values**.

If the **Time** input is specified at a time not specified by these points, interpolation is used to find the right scaling time.

**Dependencies**

To enable this parameter, select the **Use custom time scaling** check box.

To specify a scalar, the **Time** input must be a scalar.

Data Types: single | double

**Time scaling values — Time scaling vector and first two derivatives**[0:0.1:1; ones(1,11); zeros(1,11)] (default) | three-element vector | 3-by- $m$  matrix

Time scaling vector and its first two derivatives, specified as a three-element vector or 3-by- $p$  matrix, where  $p$  is the length of **Time scaling time**. By default, the time scaling is a linear time scaling spanning the **Time interval**.

For a nonlinear time scaling, specify the values of the time points in the first row. The second and third rows are the velocity and acceleration of the time points, respectively. For example, to follow the path with a linear velocity to the halfway point, and then jump to the end, the time-scaling would be:

```
s(1,:) = [0 0.25 0.5 1 1 1] % Position
s(2,:) = [1 1 1 0 0 0] % Velocity
s(3,:) = [0 0 0 0 0 0] % Acceleration
```

### Dependencies

To enable this parameter, select the **Use custom time scaling** checkbox.

To specify a three-element vector, the **Time** and **TSTime** inputs must be a scalar.

Data Types: `single` | `double`

### Simulate using — Type of simulation to run

`Interpreted execution (default)` | `Code generation`

- `Interpreted execution` — Simulate model using the MATLAB interpreter. This option shortens startup time but has a slower simulation speed than `Code generation`. In this mode, you can debug the source code of the block.
- `Code generation` — Simulate model using generated C code. The first time you run a simulation, Simulink generates C code for the block. The C code is reused for subsequent simulations, as long as the model does not change. This option requires additional startup time, but the speed of the subsequent simulations is comparable to `Interpreted execution`.

**Tunable:** No

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

## See Also

### Blocks

`Polynomial Trajectory` | `Transform Trajectory` | `Trapezoidal Velocity Profile Trajectory`

### Functions

`bsplinepolytraj` | `cubicpolytraj` | `quinticpolytraj` | `rottraj` | `transformtraj` | `trapveltraj`

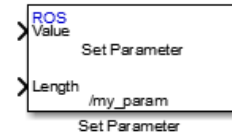


**Introduced in R2019a**

# Set Parameter

Set values on ROS parameter server

**Library:** Robotics System Toolbox / ROS



## Description

The Set Parameter block sets the **Value** input to the specified name on the ROS parameter server. The block uses the ROS node of the Simulink model to connect to the ROS network. This node is created when you run the model and is deleted when the model terminates. If the model does not have a node, the block creates one.

## Input/Output Ports

### Input

#### **Value — Parameter value**

scalar | logical | uint8 array

Parameter value from the ROS network. The value depends on the **Data type** parameter.

#### **Length — Length of string parameter**

integer

Length of the string parameter, specified as an integer. This length is the number of elements of the `uint8` array or the number of characters in the string that you cast to `uint8`.

---

**Note** When casting your string parameters to `uint8`, ASCII values 0-31 (control characters) return an error due to their incompatible character type.

---

**Dependencies**

To enable this port, set the **Data type** to `uint8[]` (string).

**Parameters****Source — Source for specifying the parameter name**

Select from ROS network | Specify your own

Source for specifying the parameter name as one of the following:

- **Select from ROS network** — Use **Select** to select a parameter name. The **Data type** parameter is set automatically. You must be connected to a ROS network.
- **Specify your own** — Enter a parameter name in **Name** and specify its data type in **Data type**. You must match a parameter name exactly.

**Name — Parameter name**

string

Parameter name to get from the ROS network, specified as a string. When **Source** is set to **Select from ROS network**, use **Select** to select an existing parameter. You must be connected to a ROS network to get a list of parameters. Otherwise, specify the parameter and data type.

Parameter name strings must follow the rules of ROS graph names. Valid names have these characteristics:

- The first character is an alpha character ([a-z|A-Z]), tilde (~), or forward slash (/).
- Subsequent characters are alphanumeric ([0-9|a-z|A-Z]), underscores(\_), or forward slashes (/).

**Data type — Data type of your parameter**

double | int32 | boolean | uint8[] (string)

Data type of your parameter, specified as a string.

---

**Note** The `uint8[]` (string) data type is an array of ASCII values corresponding to the characters in a string. When getting string parameters, you can create a MATLAB Function block to compare the string to a desired parameter value. For more information, see “ROS String Parameters”.

---

Data Types: `double` | `int32` | `Boolean` | `uint8`

## **Extended Capabilities**

### **C/C++ Code Generation**

Generate C and C++ code using Simulink® Coder™.

### **See Also**

Get Parameter

### **Topics**

“ROS Parameters in Simulink”

“ROS String Parameters”

### **External Websites**

ROS Parameter Servers

ROS Graph Names

**Introduced in R2015b**

# Subscribe

Receive messages from ROS network

**Library:** Robotics System Toolbox / ROS



## Description

The Subscribe block creates a Simulink nonvirtual bus that corresponds to the specified ROS message type. The block uses the node of the Simulink model to create a ROS subscriber for a specific topic. This node is created when the model runs and is deleted when the model terminates. If the model does not have a node, the block creates one.

On each simulation step, the block checks if a new message is available on the specific topic. If a new message is available, the block retrieves the message and converts it to a Simulink bus signal. The **Msg** port outputs this new message. If a new message is not available, **Msg** outputs the last received ROS message. If a message has not been received since the start of the simulation, **Msg** outputs a blank message.

## Input/Output Ports

### Output

#### IsNew — New message indicator

0 | 1

New message indicator, returned as a logical. If the output is **1**, then a new message was received since the last sample hit. This output can be used to trigger subsystems for processing new messages received in the ROS network.

#### Msg — ROS message

nonvirtual bus

ROS message, returned as a nonvirtual bus. The type of ROS message is specified in the **Message type** parameter. The Subscribe block outputs blank messages until it receives a

message on the topic name you specify. These blank messages allow you to create and test full models before the rest of the network has been setup.

Data Types: bus

## Parameters

### Topic source — Source for specifying topic name

Select from ROS network | Specify your own

Source for specifying the topic name, specified as one of the following:

- **Select from ROS network** — Use **Select** to select a topic name. The **Topic** and **Message type** parameters are set automatically. You must be connected to a ROS network.
- **Specify your own** — Enter a topic name in **Topic** and specify its message type in **Message type**. You must match a topic name exactly.

### Topic — Topic name to subscribe to

string

Topic name to subscribe to, specified as a string. When **Topic source** is set to **Select from ROS network**, use **Select** to select a topic from the ROS network. You must be connected to a ROS network to get a list of topics. Otherwise, set **Topic source** to **Specify your own** and specify the topic you want.

### Message type — ROS message type

string

ROS message type, specified as a string. Use **Select** to select from a full list of supported ROS messages. Service message types are not supported and are not included in the list.

### 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-block 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).

### **Length of subscribe callback queue — Message queue length**

1 (default) | integer

Message queue length in code generation, specified as an integer. In simulation, the message queue is always 1 and cannot be adjusted. To ensure each message is caught, use a smaller model step or only execute the model if `IsNew` returns 1.

## **Tips**

You can also set the addresses for the ROS master and node host by clicking the **Configure network addresses** link in the block.

## **Extended Capabilities**

### **C/C++ Code Generation**

Generate C and C++ code using Simulink® Coder™.

### **See Also**

[Blank Message](#) | [Publish](#)

### **Topics**

“Virtual and Nonvirtual Buses” (Simulink)  
“Simulink and ROS Interaction”

**Introduced in R2015a**

# Transform Trajectory

Generate trajectory between two homogeneous transforms

**Library:** Robotics System Toolbox / Utilities



## Description

The Transform Trajectory block generates an interpolated trajectory between two homogenous transformation matrices. The block outputs the transform at the times given by the **Time** input, which can be a scalar or vector.

The trajectory is computed using quaternion spherical linear interpolation (SLERP) for the rotation and linear interpolation for the translation. This method finds the shortest path between positions and rotations of the transformation. Select the **Use custom time scaling** check box to compute the trajectory using a custom time scaling. The block uses linear time scaling by default.

The initial and final values are held constant outside the time period defined in **Time interval**.

## Ports

### Input

#### Time — Time point along trajectory

scalar | vector

Time point along trajectory, specified as a scalar or vector. In general, when specified as a scalar, this value is synced with simulation time and is used to specify the time point for sampling the trajectory. The block outputs a vector of the trajectory variables at that instant in time. If the time is specified as a vector, the block outputs a matrix with each column corresponding to each element of the vector.

Data Types: single | double



**T0 — Initial transformation matrix**

4-by-4 homogeneous transformation

Initial transformation matrix, specified as a 4-by-4 homogeneous transformation. The function generates a trajectory that starts at the initial orientation, **T0**, and goes to the final orientation, **TF**.

Example: `trvec2tform([1 10 -1])`

**Dependencies**

To enable this parameter, set the **Waypoint source** to External.

Data Types: `single` | `double`

**TF — Final transformation matrix**

4-by-4 homogeneous transformation

Final transformation matrix, specified as a 4-by-4 homogeneous transformation. The function generates a trajectory that starts at the initial orientation, **T0**, and goes to the final orientation, **TF**.

Example: `trvec2tform([1 10 -1])`

**Dependencies**

To enable this parameter, set the **Waypoint source** to External.

Data Types: `single` | `double`

**TimeInterval — Start and end times for trajectory**

two-element vector

Start and end times for the trajectory, specified as a two-element vector.

Example: `[0 10]`

**Dependencies**

To enable this parameter, set the **Waypoint source** to External.

Data Types: `single` | `double`

**TSTime — Time scaling time points**scalar |  $p$ -element vector

Time scaling time points, specified as a scalar or  $n$   $p$ -element vector, where  $p$  is the number of points for time scaling. By default, the time scaling is a linear time scaling spanning the **TimeInterval**. Specify the actual time scaling values in **TimeScaling**.

If the **Time** input is specified at a time not specified by these points, interpolation is used to find the right scaling time.

#### Dependencies

To enable this parameter, select the **Use custom time scaling** check box and set **Parameter source** to External.

To specify a scalar, the **Time** input must be a scalar.

Data Types: `single` | `double`

#### **TimeScaling** — Time scaling vector and first two derivatives

three-element vector | 3-by- $p$  matrix

Time scaling vector and its first two derivatives, specified as a three element vector or a 3-by- $p$  matrix, where  $m$  is the length of **TSTime**. By default, the time scaling is a linear time scaling spanning the **TimeInterval**.

For a nonlinear time scaling, specify the values of the time points in the first row. The second and third rows are the velocity and acceleration of the time points, respectively. For example, to follow the path with a linear velocity to the halfway point, and then jump to the end, the time-scaling would be:

```
s(1,:) = [0 0.25 0.5 1 1 1] % Position
s(2,:) = [1 1 1 0 0 0] % Velocity
s(3,:) = [0 0 0 0 0 0] % Acceleration
```

#### Dependencies

To enable this parameter, select the **Use custom time scaling** check box and set **Parameter source** to External.

To specify a three-element vector, the **Time** and **TSTime** inputs must be a scalar.

Data Types: `single` | `double`

## Output

### **tform** — Homogeneous transformation matrices

4-by-4-by- $m$  homogenous matrix array

Homogeneous transformation matrices, returned as a 4-by-4-by- $m$  homogenous matrix array, where  $m$  is the number of points input to **Time**.

### **vel** — Transform velocities

6-by- $m$  matrix

Transform velocities, returned as a 6-by- $m$  matrix, where  $m$  is the number of points input to **Time**. Each row of the vector is the angular and linear velocity of the transform as  $[w_x \ w_y \ w_z \ v_x \ v_y \ v_z]$ .  $w$  represents an angular velocity and  $v$  represents a linear velocity.

### **alpha** — Transform accelerations

6-by- $m$  matrix

Transform velocities, returned as a 6-by- $m$  matrix, where  $m$  is the number of points input to **Time**. Each row of the vector is the angular and linear acceleration of the transform as  $[\alpha_x \ \alpha_y \ \alpha_z \ a_x \ a_y \ a_z]$ .  $\alpha$  represents an angular acceleration and  $a$  represents a linear acceleration.

## Parameters

### **Waypoint source** — Source for waypoints

Internal (default) | External

Specify **External** to specify the **Waypoints** and **Time points** parameters as block inputs instead of block parameters.

### **Initial transform** — Initial transformation matrix

`trvec2tform([1 10 -1])` (default) | 4-by-4 homogeneous transformation

Initial transformation matrix, specified as a 4-by-4 homogeneous transformation. The function generates a trajectory that starts at the **Initial transform** and goes to the **Final transform**.

Data Types: single | double

### **Final transform — Final transformation matrix**

`eul2tform([0 pi pi/2])` (default) | 4-by-4 homogeneous transformation

Final transformation matrix, specified as a 4-by-4 homogeneous transformation. The function generates a trajectory that starts at the **Initial transform** and goes to the **Final transform**.

Data Types: `single` | `double`

### **Time interval — Start and end times for trajectory**

`[2 3]` | two-element vector

Start and end times for the trajectory, specified as a two-element vector in seconds.

Data Types: `single` | `double`

### **Use custom time scaling — Enable custom time scaling**

`off` (default) | `on`

Enable to specify custom time scaling for the trajectory using the **Parameter Source**, **Time scaling time**, and **Time scaling values** parameters.

### **Parameter source — Source for waypoints**

`Internal` (default) | `External`

Specify `External` to specify the **Time scaling time** and **Time scaling values** parameters as block inputs instead of block parameters.

### **Dependencies**

To enable this parameter, select the **Use custom time scaling** check box.

### **Time scaling time — Time scaling time points**

`2:0.1:3` (default) | scalar |  $p$ -element vector

Time scaling time points, specified as a scalar or  $p$ -element vector, where  $p$  is the number of points for time scaling. By default, the time scaling is a linear time scaling spanning the **Time interval**. Specify the actual time scaling values in **Time scaling values**.

If the **Time** input is specified at a time not specified by these points, interpolation is used to find the right scaling time.

### **Dependencies**

To enable this parameter, select the **Use custom time scaling** check box.

To specify a scalar, the **Time** input must be a scalar.

Data Types: `single` | `double`

### **Time scaling values — Time scaling vector and first two derivatives**

`[0:0.1:1; ones(1,11); zeros(1,11)]` (default) | three-element vector | 3-by-*m* matrix

Time scaling vector and its first two derivatives, specified as a three-element vector or 3-by-*p* matrix, where *p* is the length of **Time scaling time**. By default, the time scaling is a linear time scaling spanning the **Time interval**.

For a nonlinear time scaling, specify the values of the time points in the first row. The second and third rows are the velocity and acceleration of the time points, respectively. For example, to follow the path with a linear velocity to the halfway point, and then jump to the end, the time-scaling would be:

```
s(1,:) = [0 0.25 0.5 1 1 1] % Position
s(2,:) = [1 1 1 0 0 0] % Velocity
s(3,:) = [0 0 0 0 0 0] % Acceleration
```

### **Dependencies**

To enable this parameter, select the **Use custom time scaling** checkbox.

To specify a three-element vector, the **Time** and **TSTime** inputs must be a scalar.

Data Types: `single` | `double`

### **Simulate using — Type of simulation to run**

`Interpreted execution` (default) | `Code generation`

- **Interpreted execution** — Simulate model using the MATLAB interpreter. This option shortens startup time but has a slower simulation speed than **Code generation**. In this mode, you can debug the source code of the block.
- **Code generation** — Simulate model using generated C code. The first time you run a simulation, Simulink generates C code for the block. The C code is reused for subsequent simulations, as long as the model does not change. This option requires additional startup time, but the speed of the subsequent simulations is comparable to **Interpreted execution**.

**Tunable:** No

## **Extended Capabilities**

### **C/C++ Code Generation**

Generate C and C++ code using Simulink® Coder™.

### **See Also**

#### **Blocks**

Polynomial Trajectory | Rotation Trajectory | Trapezoidal Velocity Profile Trajectory

#### **Functions**

bsplinepolytraj | cubicpolytraj | quinticpolytraj | rottraj |  
transformtraj | trapveltraj

**Introduced in R2019a**

# Trapezoidal Velocity Profile Trajectory

Generate trajectories through multiple waypoints using trapezoidal velocity profiles

**Library:** Robotics System Toolbox / Utilities



## Description

The Trapezoidal Velocity Profile Trajectory block generates a trajectory through a given set of waypoints that follow a trapezoidal velocity profile. The block outputs positions, velocities, and accelerations for a trajectory based on the given waypoints and velocity profile parameters.

## Ports

### Input

#### Time — Time point along trajectory

scalar | vector

Time point along trajectory, specified as a scalar or vector. In general, when specified as a scalar, this value is synced with simulation time and is used to specify the time point for sampling the trajectory. The block outputs a vector of the trajectory variables at that instant in time. If the time is specified as a vector, the block outputs a matrix with each column corresponding to each element of the vector.

Data Types: single | double

#### Waypoints — Waypoint positions along trajectory

$n$ -by- $p$  matrix

Positions of waypoints of the trajectory at given time points, specified as an  $n$ -by- $p$  matrix, where  $n$  is the dimension of the trajectory and  $p$  is the number of waypoints.

### Dependencies

To enable this input, set **Waypoint source** to External.

### PeakVelocity — Peak velocity of the velocity profile

[1;2] (default) | scalar |  $n$ -element vector |  $n$ -by- $(p - 1)$  matrix

Peak velocity of the profile segment, specified as a scalar, vector, or matrix. This peak velocity is the highest velocity achieved during the trapezoidal velocity profile.

A scalar value is applied to all elements of the trajectory and between all waypoints. An  $n$ -element vector is applied to each element of the trajectory between all waypoints. An  $n$ -by- $(p - 1)$  matrix is applied to each element of the trajectory for each waypoint.

### Dependencies

To enable this parameter, set **Number of parameters** to 1 or 2. Set **Parameter 1** or **Parameter 2** to Peak Velocity. Then, set **Parameter source** to External.

Data Types: single | double

### Acceleration — Acceleration of the velocity profile

[2;2] (default) | scalar |  $n$ -element vector |  $n$ -by- $(p - 1)$  matrix

Acceleration of the velocity profile, specified as a scalar, vector, or matrix. This acceleration defines the constant acceleration from zero velocity to the **PeakVelocity** value.

A scalar value is applied to all elements of the trajectory and between all waypoints. An  $n$ -element vector is applied to each element of the trajectory between all waypoints. An  $n$ -by- $(p - 1)$  matrix is applied to each element of the trajectory for each waypoint.

### Dependencies

To enable this parameter, set **Number of parameters** to 1 or 2. Set **Parameter 1** or **Parameter 2** to Acceleration. Then, set **Parameter source** to External.

Data Types: single | double

### EndTime — Duration of trajectory segment

[1;2] (default) | scalar |  $n$ -element vector |  $n$ -by- $(p - 1)$  matrix

Duration of trajectory segment, specified as a scalar, vector, or matrix.



A scalar value is applied to all elements of the trajectory and between all waypoints. An  $n$ -element vector is applied to each element of the trajectory between all waypoints. An  $n$ -by- $(p - 1)$  matrix is applied to each element of the trajectory for each waypoint.

### Dependencies

To enable this parameter, set **Number of parameters** to 1 or 2. set **Parameter 1** or **Parameter 2** to End Time. Then, set **Parameter source** to External.

Data Types: single | double

### Acceleration Time — Duration of acceleration phase of velocity profile

[1;1] (default) | scalar |  $n$ -element vector |  $n$ -by- $(p - 1)$  matrix

Duration of acceleration phase of velocity profile, specified as a scalar, vector, or matrix.

A scalar value is applied to all elements of the trajectory and between all waypoints. An  $n$ -element vector is applied to each element of the trajectory between all waypoints. An  $n$ -by- $(p - 1)$  matrix is applied to each element of the trajectory for each waypoint.

### Dependencies

To enable this parameter, set **Number of parameters** to 1 or 2. set **Parameter 1** or **Parameter 2** to Acceleration Time. Then, set **Parameter source** to External.

Data Types: single | double

## Output

### q — Position of trajectory

scalar | vector | matrix

Position of the trajectory, specified as a scalar, vector, or matrix. If you specify a scalar for the Time input with an  $n$ -dimensional trajectory, the output is a vector with  $n$  elements. If you specify a vector of  $m$  elements for the Time input, the output is an  $n$ -by- $m$  matrix.

Data Types: single | double

### qd — Velocity of trajectory

scalar | vector | matrix

Velocity of the trajectory, specified as a scalar, vector, or matrix. If you specify a scalar for the Time input with an  $n$ -dimensional trajectory, the output is a vector with  $n$  elements. If you specify a vector of  $m$  elements for the Time input, the output is an  $n$ -by- $m$  matrix.

Data Types: `single` | `double`

### **qdd — Acceleration of trajectory**

`scalar` | `vector` | `matrix`

Acceleration of the trajectory, specified as a scalar, vector, or matrix. If you specify a scalar for the `Time` input with an  $n$ -dimensional trajectory, the output is a vector with  $n$  elements. If you specify a vector of  $m$  elements for the `Time` input, the output is an  $n$ -by- $m$  matrix.

Data Types: `single` | `double`

## Parameters

### **Waypoint source — Source for waypoints**

`Internal` (default) | `External`

Specify `External` to specify the **Waypoints** and **Time points** parameters as block inputs instead of block parameters.

### **Waypoints — Waypoint positions along trajectory**

$n$ -by- $p$  matrix

Positions of waypoints of the trajectory at given time points, specified as an  $n$ -by- $p$  matrix, where  $n$  is the dimension of the trajectory and  $p$  is the number of waypoints.

### **Number of parameters — Number of velocity profile parameters**

0 (default) | 1 | 2

Number of velocity profile parameters, specified as 0, 1, or 2. Increasing this value adds **Parameter 1** and **Parameter 2** for specifying parameters for the velocity profile.

### **Parameter 1 — Velocity profile parameter**

`Peak Velocity` | `Acceleration` | `End Time` | `Acceleration Time`

Velocity profile parameter, specified as `Peak Velocity`, `Acceleration`, `End Time`, or `Acceleration Time`. Setting this parameter creates a parameter in the mask with this value as its name.

### **Dependencies**

To enable this parameter, set **Number of parameters** to 1 or 2.

If **Parameter Source** is set to `Internal`, this parameter creates a parameter in the mask with this value as its name.

If **Parameter Source** is set to `External`, this parameter creates an input port based on this value.

### Parameter 2 – Velocity profile parameter

Peak Velocity | Acceleration | End Time | Acceleration Time

Velocity profile parameter, specified as `Peak Velocity`, `Accleration`, `End Time`, or `Acceleration Time`. Setting this parameter creates a parameter in the mask with this value as its name.

#### Dependencies

To enable this parameter, set **Number of parameters** to 2.

If **Parameter Source** is set to `Internal`, this parameter creates a parameter in the mask with this value as its name.

If **Parameter Source** is set to `External`, this parameter creates an input port based on this value.

### Parameter source – Source for waypoints

`Internal` (default) | `External`

Specify `External` to specify the velocity profile parameters as block inputs instead of block parameters.

#### Dependencies

To enable this parameter, set **Number of parameters** to 1 or 2.

### PeakVelocity – Peak velocity of the velocity profile

[1; 2] (default) | scalar |  $n$ -element vector |  $n$ -by- $(p - 1)$  matrix

Peak velocity of the profile segment, specified as a scalar, vector, or matrix. This peak velocity is the highest velocity achieved during the trapezoidal velocity profile.

A scalar value is applied to all elements of the trajectory and between all waypoints. An  $n$ -element vector is applied to each element of the trajectory between all waypoints. An  $n$ -by- $(p - 1)$  matrix is applied to each element of the trajectory for each waypoint.

### Dependencies

To enable this parameter, set **Number of parameters** to 1 or 2. Then, set **Parameter 1** or **Parameter 2** to Peak Velocity.

Data Types: single | double

### Acceleration — Acceleration of the velocity profile

[2;2] (default) | scalar |  $n$ -element vector |  $n$ -by- $(p - 1)$  matrix

Acceleration of the velocity profile, specified as a scalar, vector, or matrix. This acceleration defines the constant acceleration from zero velocity to the **PeakVelocity** value.

A scalar value is applied to all elements of the trajectory and between all waypoints. An  $n$ -element vector is applied to each element of the trajectory between all waypoints. An  $n$ -by- $(p - 1)$  matrix is applied to each element of the trajectory for each waypoint.

### Dependencies

To enable this parameter, set **Number of parameters** to 1 or 2. Then, set **Parameter 1** or **Parameter 2** to Acceleration.

Data Types: single | double

### EndTime — Duration of trajectory segment

[1;2] (default) | scalar |  $n$ -element vector |  $n$ -by- $(p - 1)$  matrix

Duration of trajectory segment, specified as a scalar, vector, or matrix.

A scalar value is applied to all elements of the trajectory and between all waypoints. An  $n$ -element vector is applied to each element of the trajectory between all waypoints. An  $n$ -by- $(p - 1)$  matrix is applied to each element of the trajectory for each waypoint.

### Dependencies

To enable this parameter, set **Number of parameters** to 1 or 2. Then, set **Parameter 1** or **Parameter 2** to End Time.

Data Types: single | double

### Acceleration Time — Duration of acceleration phase of velocity profile

[1;1] (default) | scalar |  $n$ -element vector |  $n$ -by- $(p - 1)$  matrix

Duration of acceleration phase of velocity profile, specified as a scalar, vector, or matrix.

A scalar value is applied to all elements of the trajectory and between all waypoints. An  $n$ -element vector is applied to each element of the trajectory between all waypoints. An  $n$ -by- $(p - 1)$  matrix is applied to each element of the trajectory for each waypoint.

### Dependencies

To enable this parameter, set **Number of parameters** to 1 or 2. Then, set **Parameter 1** or **Parameter 2** to Acceleration Time.

Data Types: single | double

### Simulate using — Type of simulation to run

Interpreted execution (default) | Code generation

- **Interpreted execution** — Simulate model using the MATLAB interpreter. This option shortens startup time but has a slower simulation speed than **Code generation**. In this mode, you can debug the source code of the block.
- **Code generation** — Simulate model using generated C code. The first time you run a simulation, Simulink generates C code for the block. The C code is reused for subsequent simulations, as long as the model does not change. This option requires additional startup time, but the speed of the subsequent simulations is comparable to **Interpreted execution**.

**Tunable:** No

## References

- [1] Lynch, Kevin M., and Frank C. Park. *Modern Robotics: Mechanics, Planning and Control*. Cambridge: Cambridge University Press, 2017.
- [2] Spong, Mark W., Seth Hutchinson, and M. Vidyasagar. *Robot Modeling and Control*. John Wiley & Sons, 2006.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

## See Also

### Blocks

Polynomial Trajectory | Rotation Trajectory | Transform Trajectory

### Functions

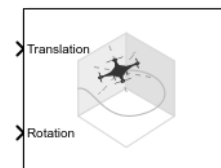
bsplinepolytraj | cubicpolytraj | quinticpolytraj | rottraj |  
transformtraj | trapveltraj

**Introduced in R2019a**

# UAV Animation

Animate UAV flight path using translations and rotations

**Library:** Robotics System Toolbox UAV Library



## Description

---

**Note** This block requires you to install the UAV Library for Robotics System Toolbox. To install add-ons, use `roboticsAddons` and select the desired add-on.

---

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-block 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).



## See Also

### Functions

`plotTransforms` | `roboticsAddons` | `state`

### Objects

`robotics.FixedWingGuidanceModel` | `robotics.MulticopterGuidanceModel` | `robotics.WaypointFollower`

### Blocks

Waypoint Follower

### Topics

“Approximate High-Fidelity UAV model with UAV Guidance Model block”

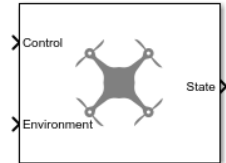
“Tuning Waypoint Follower for Fixed-Wing UAV”

**Introduced in R2018b**

## UAV Guidance Model

Reduced-order model for UAV

**Library:** Robotics System Toolbox UAV Library



### Description

---

**Note** This block requires you to install the UAV Library for Robotics System Toolbox. To install add-ons, use `roboticsAddons` and select the desired add-on.

---

The UAV 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 per second.
- **FlightPathAngle** - Angle between ground velocity and north-east plane in meters per second.
- **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 multicopter 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** - [r p q] 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 per second.
- **Flight Path Angle** - Angle between ground velocity and north-east plane in meters per second.
- **Roll Angle** - Angle of rotation along body x-axis in radians per second.
- **Roll Angle Rate** - Angular velocity of rotation along body x-axis in radians per second.

**Tunable:** No

### **Configuration — UAV controller configuration tab**

multiple table entries

UAV controller configuration tab, specified as multiple table entries. This tab allows you to configure the parameters of the internal control behaviour of the UAV. Specify the proportional (P) and derivative (D) gains for the dynamic model and the UAV mass in kilograms (for multirotor).

For multirotor UAVs, the parameters are:

- **PD Roll**
- **PD Pitch**
- **P YawRate**
- **P Thrust**
- **Mass(kg)**

For fixed-wing UAVs, the parameters are:

- **P Height**
- **P Flight Path Angle**
- **PD Roll**
- **P Air Speed**
- **Min/Max Flight Path Angle** ([min max] angle in radians)

**Tunable:** No

### **Input/Output Bus Names — Simulink bus signal names tab**

multiple entries of character vectors

Simulink bus signal names tab, specified as multiple entries of character vectors. These buses have a default name based on the UAV model and input type. To use multiple guidance models in the same Simulink model, specify different bus names that do not intersect. All entries on this tab are nontunable.

## Definitions

### UAV Coordinate Systems

The UAV Library for Robotics System 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,  $\psi$ . Then, rotate about the intermediate  $y$ -axis by the pitch angle,  $\phi$ . Then, rotate about the intermediate  $x$ -axis by the roll angle,  $\theta$ .

The angular velocity of the UAV is represented by  $[r, p, q]$  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 heading angle. The corresponding equations of motion are:

$$\begin{aligned}\dot{x}_e &= V_g \cos \chi \cos \gamma \\ \dot{y}_e &= V_g \sin \chi \cos \gamma \\ \dot{h} &= V_g \sin \gamma \\ \dot{\chi} &= \frac{g \cos(\chi - \psi)}{V_g} \tan \phi \\ V_g \sin(\gamma^c) &= \min(\max(k_h(h^c - h), -V_g), V_g) \\ \dot{\gamma} &= k_\gamma(\gamma^c - \gamma) \\ \dot{V}_a &= k_{V_a}(V_a^c - V_a) \\ \frac{g \cos(\chi - \psi)}{V_g} \tan(\phi^c) &= k_\chi(\chi^c - \chi) \\ \ddot{\phi} &= k_P\phi(\phi^c - \phi) + k_D\dot{\phi}(-\dot{\phi})\end{aligned}$$

$V_a$  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  $[x_b, y_b, z_b]$ .

When converting coordinates from the world (earth) frame to the body frame of the UAV, the rotation matrix 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:

$$J = \begin{bmatrix} 1 & \sin \phi \tan \theta & \cos \phi \tan \theta \\ 0 & \cos \phi & -\sin \phi \\ 0 & \frac{\sin \phi}{\cos \theta} & \frac{\cos \phi}{\cos \theta} \end{bmatrix}$$

$$\begin{bmatrix} \dot{\phi} \\ \dot{\theta} \\ \dot{\psi} \end{bmatrix} = J \begin{bmatrix} p \\ q \\ r \end{bmatrix}$$

$$\begin{bmatrix} \dot{p} \\ \dot{q} \\ \dot{r} \end{bmatrix} = \begin{bmatrix} 1 & 0 & -\sin \theta \\ 0 & \cos \phi & \sin \phi \cos \theta \\ 0 & -\sin \phi & \cos \phi \cos \theta \end{bmatrix} \begin{bmatrix} KP_{\phi}(\phi^c - \phi) + KD_{\phi}(-\dot{\phi}) \\ KP_{\theta}(\theta^c - \theta) + KD_{\theta}(-\dot{\theta}) \\ KP_{\psi}(\psi^c - \psi) \end{bmatrix}$$

$$\dot{F}_{thrust} = KP_F(F_{thrust}^c - F_{thrust})$$

This model assumes the autopilot takes in commanded roll, pitch, yaw angles,  $[\psi^c, \theta^c, \phi^c]$  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  $KP_{\alpha}$  and  $KD_{\alpha}$ , where  $\alpha$  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:

$$[x_e \ y_e \ z_e \ \dot{x}_e \ \dot{y}_e \ \dot{z}_e \ \psi \ \theta \ \phi \ r \ p \ q \ F_{thrust}]$$

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.

## See Also

### Functions

control | derivative | environment | ode45 | plotTransforms | roboticsAddons | state

### Objects

fixedwing | multirotor | uavWaypointFollower

### Blocks

Waypoint Follower

## Topics

"Approximate High-Fidelity UAV model with UAV Guidance Model block"

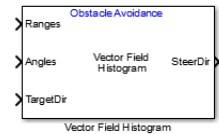
"Tuning Waypoint Follower for Fixed-Wing UAV"

## Introduced in R2018b

## Vector Field Histogram

Avoid obstacles using vector field histogram

**Library:** Robotics System Toolbox / Mobile Robot Algorithms



### Description

The Vector Field Histogram (VFH) block enables your robot to avoid obstacles based on range sensor data. Given a range sensor reading in terms of ranges and angles, and a target direction to drive toward, the VFH controller computes an obstacle-free steering direction.

For more information on the algorithm details, see “Vector Field Histogram” on page 4-119 under Algorithms.

### Limitations

- The Ranges and Angles inputs are limited to 4000 elements when generating code for models that use this block.

### Input/Output Ports

#### Input

##### **Ranges — Range values from scan data**

vector of scalars

Range values from scan data, specified as a vector of scalars in meters. These range values are distances from a sensor at specified angles. The vector must be the same length as the corresponding **Angles** vector.

**Angles — Angle values from scan data**

vector of scalars

Angle values from scan data, specified as a vector of scalars in radians. These angle values are the specific angles of the specified ranges. The vector must be the same length as the corresponding **Ranges** vector.

**TargetDir — Target direction for robot**

scalar

Target direction for the robot, specified as a scalar in radians. The forward direction of the robot is considered zero radians, with positive angles measured counterclockwise. You can use the **TargetDir** output from the Pure Pursuit block when generating controls from a set of waypoints.

**Output****steeringDir — Steering direction for robot**

scalar

Steering direction for the robot, specified as a scalar in radians. This obstacle-free direction is calculated based on the VFH+ algorithm. The forward direction of the robot is considered zero radians, with positive angles measured counterclockwise.

**Parameters****Main****Number of angular sectors — Number of bins used to create the histograms**

180 (default) | scalar

Number of bins used to create the histograms, specified as a scalar. This parameter is nontunable. You can set this parameter only when the object is initialized.

**Range distance limits (m) — Limits for range readings**

[0.05 2] (default) | two-element vector of scalars

Limits for range readings in meters, specified as a two-element vector of scalars. The range readings input are only considered if they fall within the distance limits. Use the

lower distance limit to ignore false positives from poor sensor performance at lower ranges. Use the upper limit to ignore obstacles that are too far away from the robot.

### **Histogram thresholds — Thresholds for computing binary histogram**

[3 10] (default) | two-element vector of scalars

Thresholds for computing binary histogram, specified as a two-element vector of scalars. The algorithm uses these thresholds to compute the binary histogram from the polar obstacle density. Polar obstacle density values higher than the upper threshold are represented as occupied space (1) in the binary histogram. Values smaller than the lower threshold are represented as free space (0). Values that fall between the limits are set to the values of a previous computed binary histogram if one exists from previous iterations. If a previous histogram does not exist, the value is set as free space (0).

### **Robot radius (m) — Radius of the robot**

0.1 (default) | scalar

Radius of the robot, specified as a scalar in meters. This dimension defines the smallest circle that can circumscribe your robot. The robot radius is used to account for robot size when computing the obstacle-free direction.

### **Safety distance (m) — Safety distance around the robot**

0.1 (default) | scalar

Safety distance left around the robot position in addition to **Robot radius**, specified as a scalar in meters. The robot radius and safety distance are used to compute the obstacle-free direction.

### **Minimum turning radius (m) — Minimum turning radius at current speed**

0.1 (default) | scalar

Minimum turning radius for the robot moving at its current speed, specified as a scalar in meters.

### **Simulate using — Specify type of simulation to run**

Code generation (default) | Interpreted execution

- **Code generation** — Simulate model using generated C code. The first time you run a simulation, Simulink generates C code for the block. The C code is reused for subsequent simulations, as long as the model does not change. This option requires additional startup time but the speed of the subsequent simulations is comparable to **Interpreted execution**.

- **Interpreted execution** — Simulate model using the MATLAB interpreter. This option shortens startup time but has a slower simulation speed than **Code generation**. In this mode, you can debug the source code of the block.

**Tunable:** No

### Cost Function Weights

#### **Target direction weight — Cost function weight for target direction**

5 (default) | scalar

Cost function weight for moving toward the target direction, specified as a scalar. To follow a target direction, set this weight to be higher than the sum of **Current direction weight** and **Previous direction weight**. To ignore the target direction cost, set this weight to 0.

#### **Current direction weight — Cost function weight for current direction**

2 (default) | scalar

Cost function weight for moving the robot in the current heading direction, specified as a scalar. Higher values of this weight produce efficient paths. To ignore the current direction cost, set this weight to 0.

#### **Previous direction weight — Cost function weight for previous direction**

2 (default) | scalar

Cost function weight for moving in the previously selected steering direction, specified as a scalar. Higher values of this weight produce smoother paths. To ignore the previous direction cost, set this weight to 0.

## Algorithms

### Vector Field Histogram

The block uses the VFH+ algorithm to compute the obstacle-free direction. First, the algorithm takes the ranges and angles from range sensor data and builds a polar histogram for obstacle locations. Then, it uses the input histogram thresholds to calculate a binary histogram that indicates occupied and free directions. Finally, the algorithm computes a masked histogram, which is computed from the binary histogram based on the minimum turning radius of the robot.

The algorithm selects multiple steering directions based on the open space and possible driving directions. A cost function, with weights corresponding to the previous, current, and target directions, calculates the cost of different possible directions. The algorithm then returns an obstacle-free direction with minimal cost. Using the obstacle-free direction, you can input commands to move your robot in that direction.

To use this block for your own application and environment, you must tune the algorithm parameters. Parameter values depend on the type of robot, the range sensor, and the hardware you use. For more information on the VFH algorithm, see “Vector Field Histogram”.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

### See Also

#### Blocks

[Publish](#) | [Pure Pursuit](#) | [Subscribe](#)

#### Classes

`robotics.VectorFieldHistogram`

#### Topics

[“Vector Field Histogram”](#)

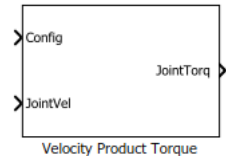
#### Introduced in R2016b



# Velocity Product Torque

Joint torques that cancel velocity-induced forces

**Library:** Robotics System Toolbox / Manipulator Algorithms



## Description

The Velocity Product Torque block returns the torques that cancel the velocity-induced forces for the given robot configuration (joint positions) and joint velocities for the **Rigid body tree** robot model.

## Ports

### Input

#### **Config** — Robot configuration

vector

Robot configuration, specified as a vector of positions for all nonfixed joints in the robot model, as set by the **Rigid body tree** parameter. You can also generate this vector for a complex robot using the `homeConfiguration` or `randomConfiguration` functions inside a Constant or MATLAB Function block.

#### **JointVel** — Joint velocities

vector

Joint velocities, specified as a vector. The number of joint velocities is equal to the degrees of freedom (number of nonfixed joints) of the robot.

### Output

#### JointTorq — Joint torques

vector

Joint torques, specified as a vector. Each element corresponds to a torque applied to a specific joint. The number of joint torques is equal to the degrees of freedom (number of nonfixed joints) of the robot.

### Parameters

#### Rigid body tree — Robot model

twoJointRigidBodyTree (default) | RigidBodyTree object

Robot model, specified as a RigidBodyTree object. You can also import a robot model from an URDF (Unified Robot Description Formation) file using `importrobot`.

The default robot model, `twoJointRigidBodyTree`, is a robot with revolute joints and two degrees of freedom.

#### Simulate using — Type of simulation to run

Interpreted execution (default) | Code generation

- **Interpreted execution** — Simulate model using the MATLAB interpreter. This option shortens startup time but has a slower simulation speed than **Code generation**. In this mode, you can debug the source code of the block.
- **Code generation** — Simulate model using generated C code. The first time you run a simulation, Simulink generates C code for the block. The C code is reused for subsequent simulations, as long as the model does not change. This option requires additional startup time, but the speed of the subsequent simulations is comparable to **Interpreted execution**.

**Tunable:** No

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

### See Also

#### Blocks

Forward Dynamics | Get Jacobian | Gravity Torque | Inverse Dynamics | Joint Space Mass Matrix

#### Classes

RigidBodyTree

#### Functions

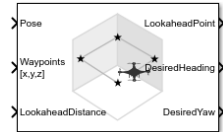
homeConfiguration | importrobot | randomConfiguration | velocityProduct

**Introduced in R2018a**

# Waypoint Follower

Follow waypoints for UAV

**Library:** Robotics System Toolbox UAV Library



## 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 heading, and desired yaw given a UAV position, a set of waypoints, and a lookahead distance. Specify a set of waypoints and tune the lookahead distance and transition radius parameters for navigating the waypoints. The block supports both multirotor and fixed-wing UAV types.

## Ports

### Input

#### Pose — Current UAV pose

[x y z chi] vector

Current UAV pose, specified as a [x y z chi] vector. This pose is used to calculate the lookahead point based on the input to the **LookaheadDistance** port. [x y z] is the current position in meters. chi is the current heading in radians.

Example: [0.5;1.75;-2.5;pi]

Data Types: single | double

#### Waypoints — Set of waypoints

$n$ -by-3 matrix |  $n$ -by-4 matrix |  $n$ -by-5 matrix

Set of waypoints for the UAV to follow, specified as a matrix with number of rows,  $n$ , equal to the number of waypoints. The number of columns depend on the **Show Yaw input variable** and the **Transition radius source** parameter.

Each row in the matrix has the first three elements as an  $[x \ y \ z]$  position in the sequence of waypoints.

If **Show Yaw input variable** is checked, specify the desired yaw angle,  $yaw$ , as the fourth element in radians.

If **Show Yaw input variable** is unchecked, and **Transition radius source** is external, the transition radius is the fourth element of the vector in meters.

If **Show Yaw input variable** is checked, and **Transition radius source** is external, the transition radius is the fifth element of the vector in meters.

The block display updates as the size of the waypoint matrix changes.

Data Types: `single` | `double`

### **LookaheadDistance** — Lookahead distance

positive numeric scalar

Lookahead distance along the path, specified as a positive numeric scalar in meters.

Data Types: `single` | `double`

## **Output**

### **LookaheadPoint** — Lookahead point on path

$[x \ y \ z]$  position vector

Lookahead point on path, returned as an  $[x \ y \ z]$  position vector in meters.

Data Types: `single` | `double`

### **DesiredHeading** — Desired heading

numeric scalar

Desired heading, returned as numeric scalar in radians in the range of  $[-\pi, \pi]$ . The UAV heading is the angle of direction of the velocity vector relative to north measured in radians.

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.

Data Types: `single` | `double`

### **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.

#### **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.

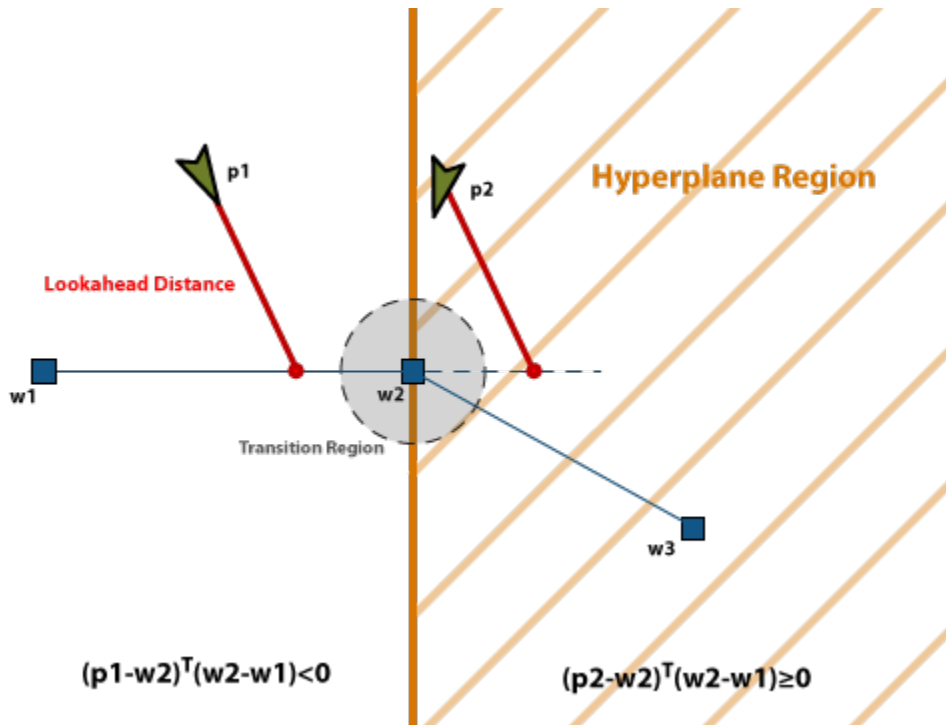
**Tunable:** No

## Definitions

### **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:

$$(p - w_1)^T (w_2 - w_1) \geq 0$$

$p$  is the UAV position, and  $w_1$  and  $w_2$  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.

## See Also

### Blocks

Orbit Follower | UAV Guidance Model

### Functions

control | derivative | environment | ode45 | plotTransforms | roboticsAddons  
| state

### Objects

robotics.FixedWingGuidanceModel | robotics.MulticopterGuidanceModel |  
uavWaypointFollower

### Topics

“Approximate High-Fidelity UAV model with UAV Guidance Model block”

“Tuning Waypoint Follower for Fixed-Wing UAV”

### Introduced in R2018b

# **Apps in Robotics System Toolbox**

---








## SLAM Map Builder

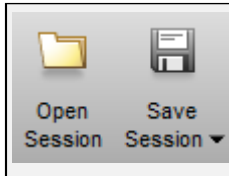
Build 2-D grid maps using lidar-based SLAM

### Description

The **SLAM Map Builder** app loads recorded lidar scans and odometry sensor data to build a 2-D occupancy grid using simultaneous localization and mapping (SLAM) algorithms. Incremental scan matching aligns and overlays scans to build the map. Loop closure detection adjusts for drift of the robot odometry by detecting previously visited locations and adjusting the overall map. Sometimes, the scan matching algorithm and loop closure detection require manual adjustment. Use the app to manually align scans and modify loop closures to improve the overall map accuracy. You can also tune the SLAM algorithm settings to improve the automatic map building.

**To use the app:**

 Import	<p>Select <b>Import</b> &gt; <b>Import from rosbag</b>. Select the rosbag file and click <b>Open</b>. This opens the <b>Import</b> tab. For more information, see Import and Filter a rosbag on page 5-13.</p> <p>You can also specify scans and odometry poses that are prefiltered in the workspace by calling <code>slamMapBuilder</code> with inputs. This skips the import process. See Programmatic Use on page 5-13.</p>
 SLAM Settings	<p>Use <b>SLAM Settings</b> to adjust the SLAM algorithm settings. Default values are provided, but your specific sensors and data may require tuning of these settings. The most important value to tune is the <b>Loop Closure Threshold</b>. For more information, see Tune SLAM Settings on page 5-14.</p>
 Build	<p>Click <b>Build</b> to begin the SLAM map building process. The building process aligns scans in the map using incremental scan matching, identifies loop closures when visiting previous locations, and adjusts poses. Click <b>Pause</b> at any time during the map building process to manually align incremental scans or modify loop closures.</p>
  Incremental Match    Loop Closure	<p>Click <b>Incremental Match</b> to modify the relative pose of the currently selected frame and align the scan with the previous scan. Click <b>Loop Closure</b> to modify or ignore the detected loop closure for the current frame. Use the slider on the bottom to scroll back to areas where scan matching or loop closures are not accurate. You can modify any number of scans or loop closures. For more information, see Modify Increment Scans and Loop Closures on page 5-15.</p>
 Sync	<p>After modifying your map, click <b>Sync</b> to update all the poses in the scan map. The two options under <b>Sync</b> are <b>Sync</b>, which searches for new loop closures, or <b>Sync Fast</b>, which skips loop closure searching and just updates the scan map. For more information, see Sync the Map on page 5-16.</p>
 Export Occupancy Grid ▼	<p>When you are satisfied with how the map looks, click <b>Export to OccupancyGrid</b> to either export the map to an m-file, or save the map in the workspace. The map is output as a 2-D probabilistic occupancy grid in a <code>robotics.OccupancyGrid</code> object.</p>



You can open existing app sessions you have saved using **Open Session**. When you are in the **Map Builder** tab, you can save your progress to an m-file using **Save Session**.

## Open the SLAM Map Builder App

- MATLAB Toolstrip: On the **Apps** tab, under **Control System Design and Analysis**,

click  **SLAM Map Builder**.

- MATLAB Command Window: Enter `slamMapBuilder`

## Examples

### Build and Tune a Map Using Lidar Scans with SLAM

The **SLAM Map Builder** app helps you build an occupancy grid from lidar scans using simultaneous localization and mapping (SLAM) algorithms. The map is built by estimating poses through scan matching and using loop closures for pose graph optimization. This example shows you the workflow for loading a rosbag of lidar scan data, filtering the data, and building the map. Tune the scan map by adjusting incremental scan matches and modifying loop closures.

### Open the App

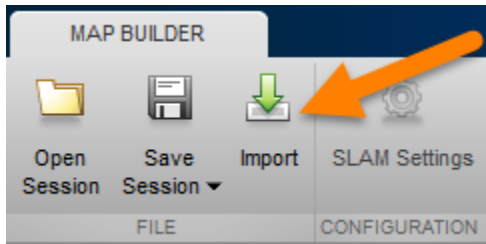
In the **Apps** tab, under **Control System Design and Analysis**, click **SLAM Map Builder**.

Also, you can call the function:

```
slamMapBuilder
```

### Import Lidar Scans from rosbag

Click **Import > Import from rosbag** to load a rosbag. The provided rosbag, `southend.bag`, contains laser scan messages. Select the file and open. The scans are shown in the **Import** tab.



The screenshot shows the 'IMPORT' dialog box in the SLAM Map Builder software. The dialog has several tabs: 'LIDAR', 'ODOMETRY', 'TF', and 'DATA SELECTION'. The 'LIDAR' tab is active, showing a 'Lidar Scan' plot. The plot displays a 2D grid with X and Y axes ranging from -15 to 15. Magenta dots represent the scanned data points, forming a shape that resembles a stylized 'A' or a similar character. The 'Raw Data Information' panel on the right provides details about the imported data:

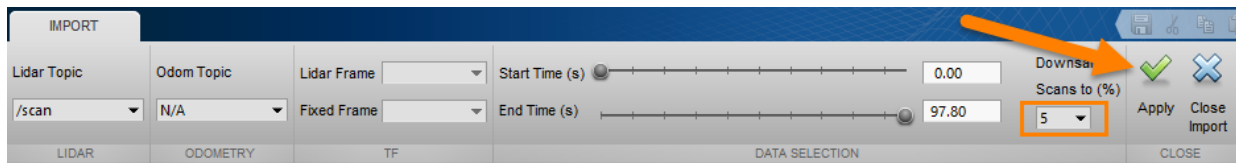
- rosbag File:** southend.bag (55.52 MB)
- Start Time:** 12-Feb-2018 00:57:25
- End Time:** 12-Feb-2018 00:59:03
- Available Topics:**
  - /axis/image\_raw\_out/compressed
  - /scan
  - /tf

Below the 'Raw Data Information' panel is the 'Odometry' panel, which contains an empty 2D plot with X and Y axes ranging from -0.5 to 1.5. The 'Import' dialog also includes a 'Start Time (s)' slider set to 0.00 and an 'End Time (s)' slider set to 97.80. There are 'Apply' and 'Close Import' buttons on the right side of the dialog.

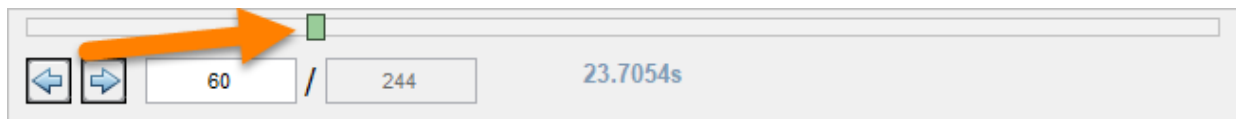
In the **Import** Tab, specify the import parameters. The **Lidar Topic** is preselected as `/scan` because that is the only `sensor_msgs/LaserScan` topic found. If odometry is available as a `tf` tree, select **Use TF**, and specify the **Lidar Frame** (sensor frame) and the **Fixed Frame** (world frame).

Select the desired **Start Time (s)** and **End Time (s)**. Because the scans are captured at a high frequency, downsample the scans to reduce data processing. Select the desired percentage of scans in **Downsample Scans to (%)**. Scans are evenly sampled. For example, 5% is every 20th scan.

Click **Apply** to apply filtering parameters.



Use the slider or arrow keys at the bottom to preview the scans.

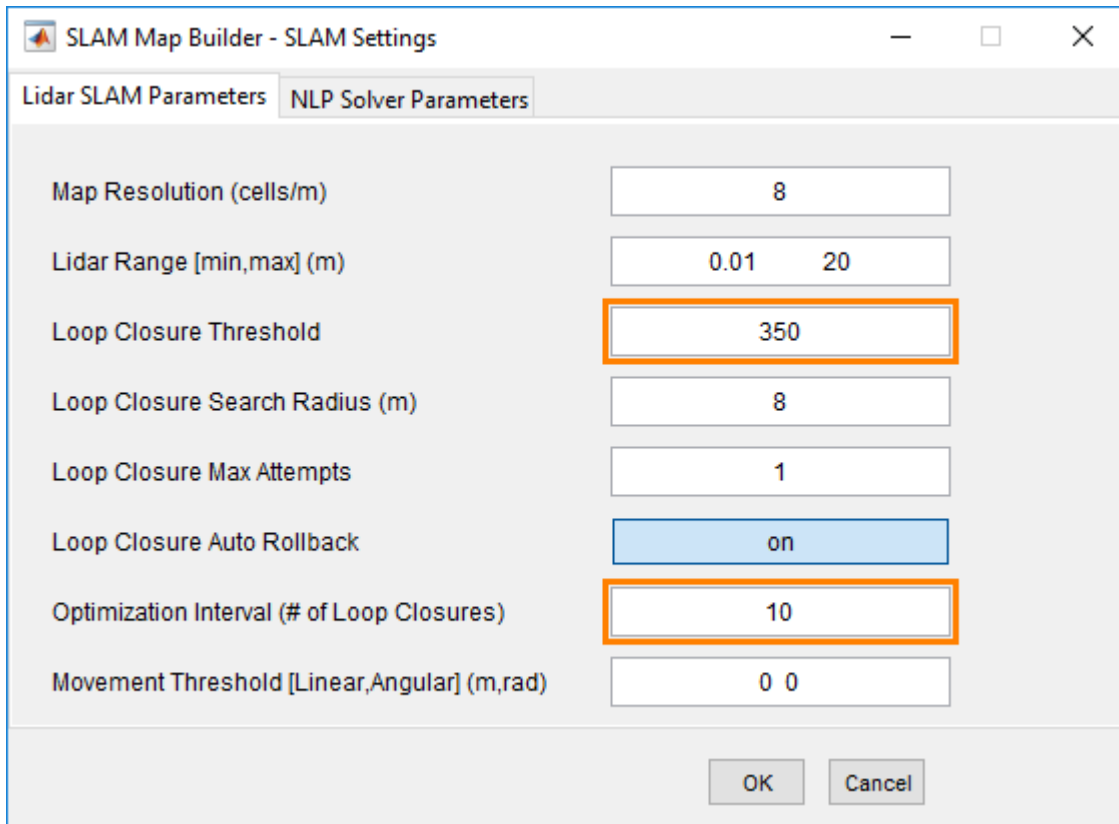


Once you are done filtering, click **Close**.

### Tune SLAM Settings

The SLAM algorithm can be tuned using the **SLAM Settings** dialog. The parameters should be adjusted based on your sensor specifications, the environment, and your robotic application. For this example, increase **Loop Closure Threshold** from 200 to 300. This increased threshold decreases the likelihood of accepting and using a detected loop closure. Set the **Optimization Interval** to 10. With every 10th loop closure accepted, the pose graph is optimized to account for drift.





SLAM Map Builder - SLAM Settings

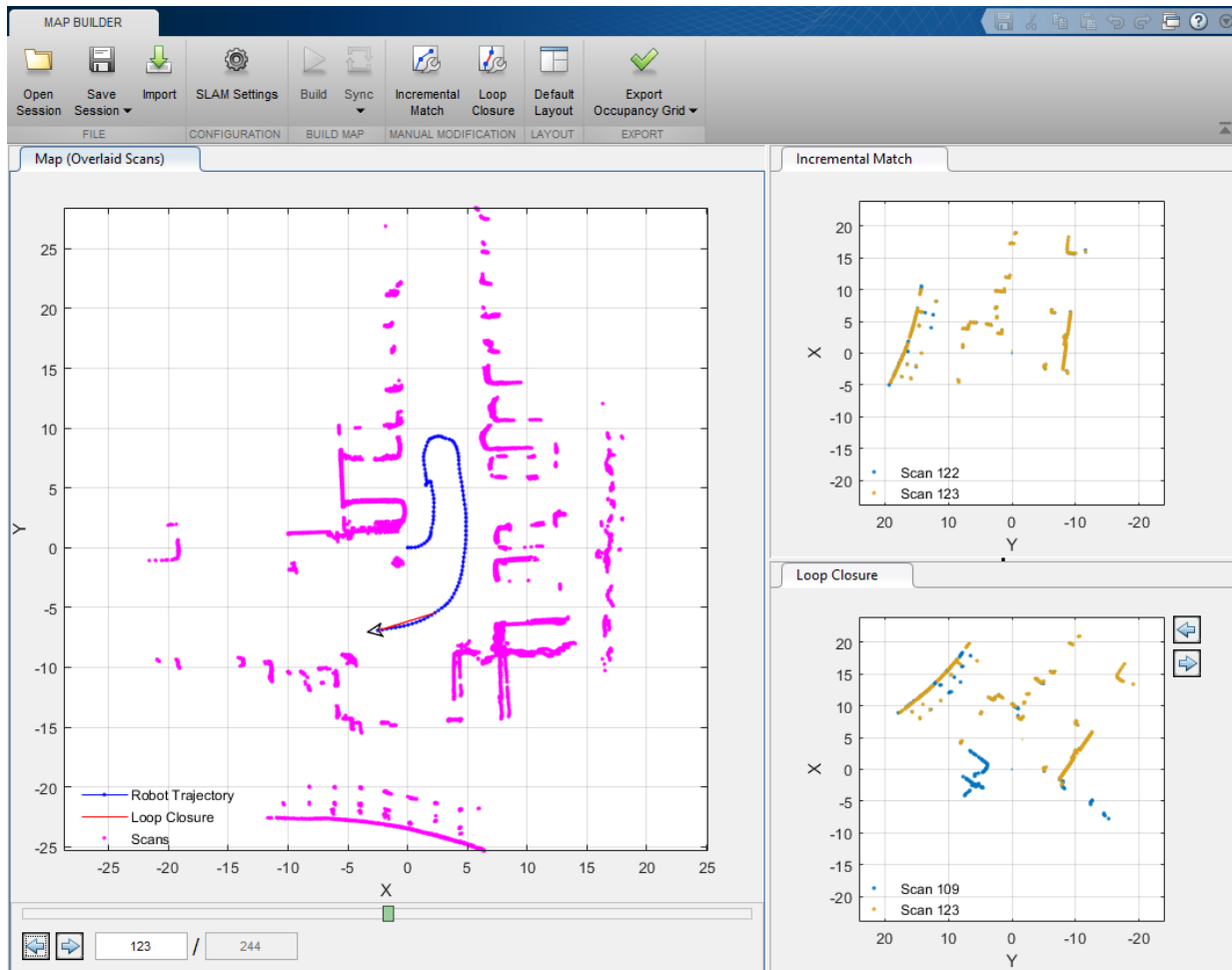
Lidar SLAM Parameters | NLP Solver Parameters

Map Resolution (cells/m)	8
Lidar Range [min,max] (m)	0.01 20
Loop Closure Threshold	350
Loop Closure Search Radius (m)	8
Loop Closure Max Attempts	1
Loop Closure Auto Rollback	on
Optimization Interval (# of Loop Closures)	10
Movement Threshold [Linear,Angular] (m,rad)	0 0

OK Cancel

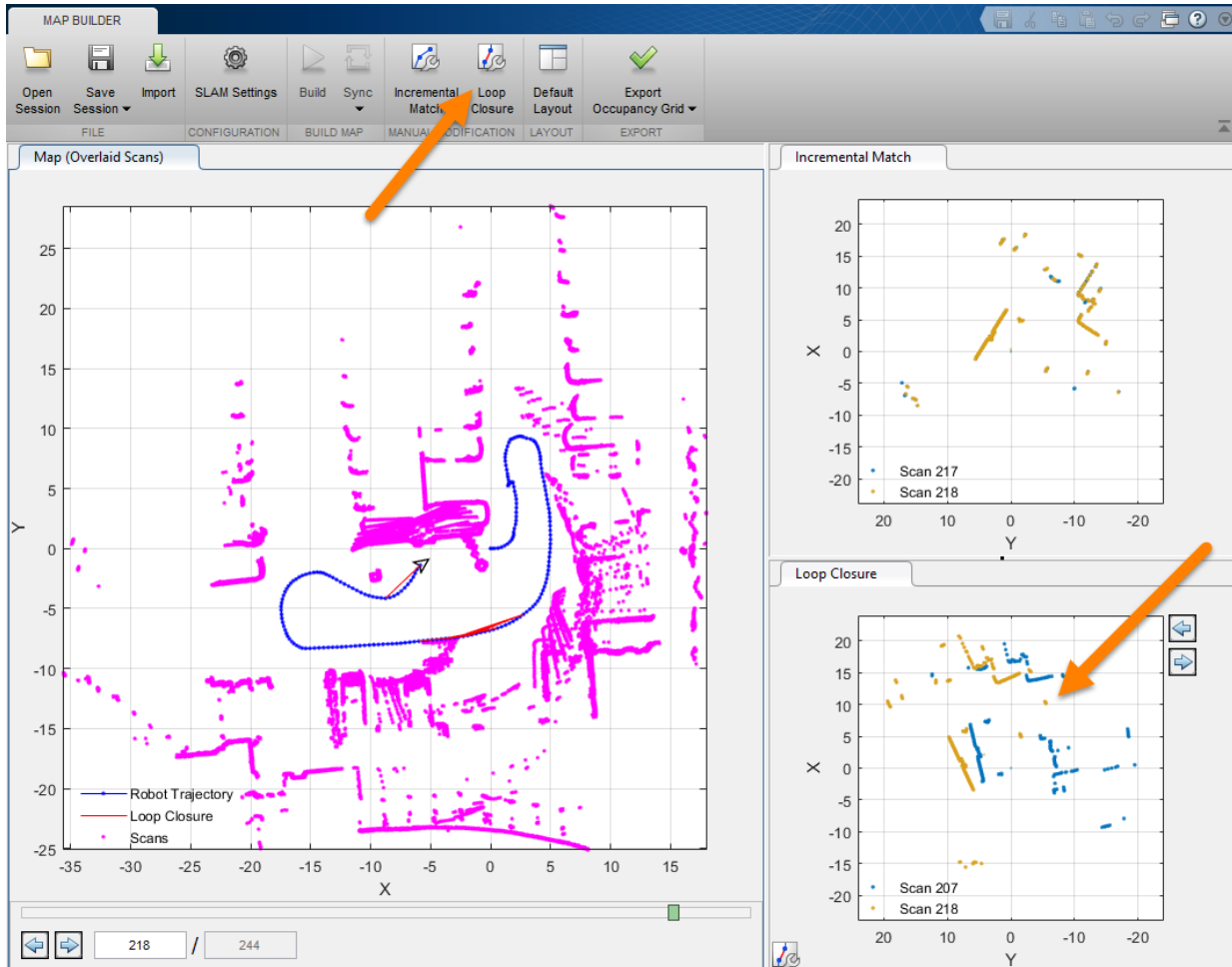
### Build the Map

After filtering your data and setting the SLAM algorithm settings, click **Build**. The app begins processing scans to build the map. You should see the slider progressing and scans being overlaid in the map. The estimated robot trajectory is plotted on the same scan map. Incremental scan matches are shown in the **Incremental Match** pane. Whenever a loop closure is detected, the **Loop Closure** pane shows the two scans overlaid on each other.



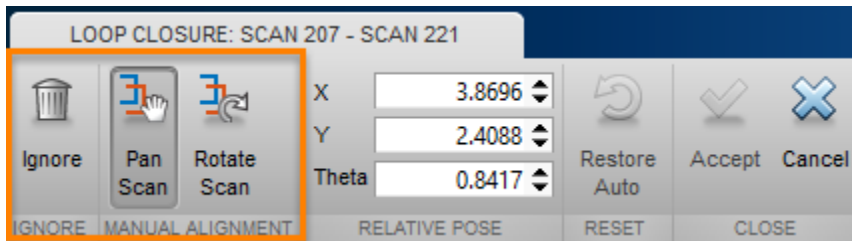
At any time during the build process, if you notice the map is distorted or an incremental match or loop closure looks off, click **Pause** to select scans for adjustment. You can modify scans at the end of the build process as well. Navigate using the arrow keys or slider to the point in the file where the distortion first occurs. Click the **Incremental Match** or **Loop Closure** buttons to adjust the currently displayed scan poses. In this example, we manually created a bad loop closure that does not normally occur with this data set at scan 218.

Click the **Loop Closure** button. This opens a tab for modifying the loop closure relative pose.

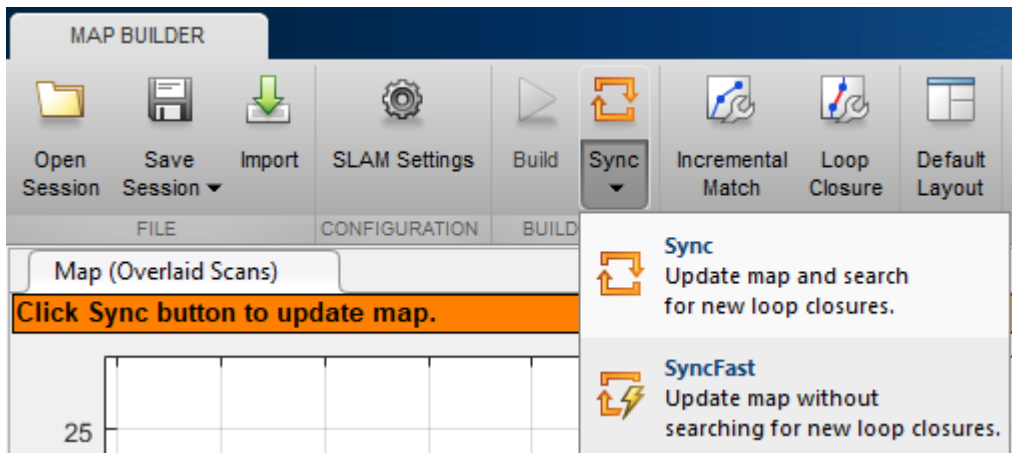


To ignore the loop closure completely, click **Ignore**. Otherwise, manually modify the relative scan pose until the scans line up.

Click **Pan Scan** or **Rotate Scan**, then click and drag in the figure to align the two scans. Click **Accept** when you are done. You can do this for multiple scans.

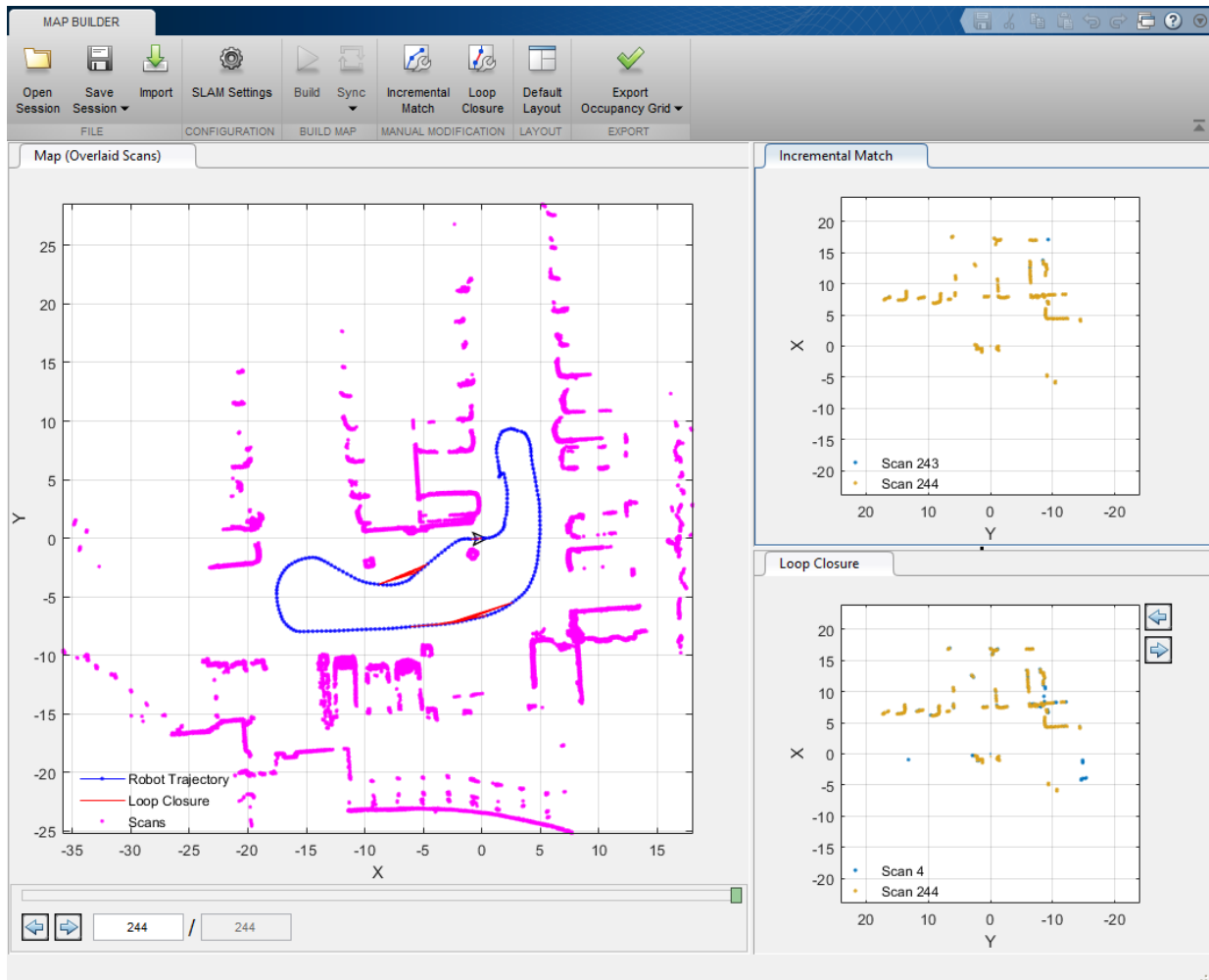


After you modify your scan poses for incremental matches and loop closures, click **Sync** to apply changes. **SyncFast** updates the map without searching for new loop closures and reduces computation time if you have already processed all the scans.



### Export Occupancy Grid

Once you have synced your changes and finished building the map, you should see a fully overlaid scan map with a robot trajectory.



Click **Export Occupancy Grid** to get a final occupancy map of your environment as a `robotics.OccupancyGrid` object. Specify the variable name to export the map to the workspace. You can create a map from a subset of scans by scrolling back to the desired frame before exporting and selecting **Up to currently selected scan**.

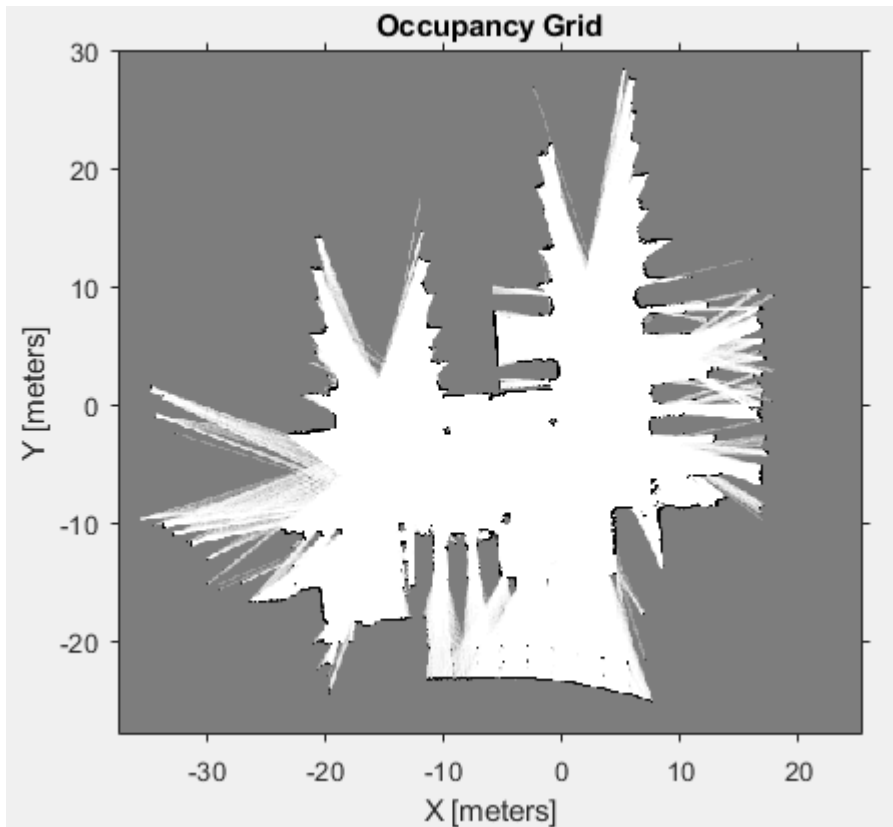
Occupancy grid map variable name:

All processed scans (Scan 1 - Scan 244)

Up to currently selected scan (Scan 1 - Scan 244)

Call `show` on the stored map to visualize the occupancy map.

```
show(myOccMap)
```



You can also save a SLAM Map Builder app session using the **Save Session** button. The app writes the current state of the app to a `.mat` file that can be loaded later using **Open Session**.

- “Implement Simultaneous Localization And Mapping (SLAM) with Lidar Scans”
- “Implement Online Simultaneous Localization And Mapping (SLAM) with Lidar Scans”

## Programmatic Use

`slamMapBuilder(bag)` opens the **SLAM Map Builder** app and imports the rosbag log file specified in `bag`, a `BagSelection` object created using the `rosbag` function. The app opens to the **Import** tab to filter the sensor data in your rosbag.

`slamMapBuilder(sessionFile)` opens the **SLAM Map Builder** app from a saved session file name, `sessionFile`. An app session file is created through the **Save Session** button in the app toolbar.

`slamMapBuilder(scans)` opens the **SLAM Map Builder** app and imports the scans specified in `scans`, a cell array of `lidarScan` objects. The app assumes you have prefiltered your scans and skips the import process. Click **Build** to start building the map.

`slamMapBuilder(scans, poses)` opens the **SLAM Map Builder** app and imports the scans and poses. `scans` is specified as a cell array of `lidarScan` objects. `poses` is a matrix of `[x y theta]` vectors that correspond to the poses of `scans`. The app assumes you have prefiltered your scans and skips the import process. Click **Build** to start building the map.

## Definitions

### Import and Filter a rosbag

When you click the **Import** button, specify the parameters for your rosbag and how you want to filter the data in the toolbar. You must **Apply** your settings to see the scans updated in the figures.

- Select the ROS topic for the lidar scans and odometry (if available).

- In **Odom Topic**, if you select **Use TF**, specify the frame of the lidar scan sensor, **Lidar Frame**, and the base fixed frame of the robot, **Fixed Frame**. The items in the drop down menu are generated based on the available frames in the tf transformation tree of the rosbag.
- Specify the **Start Time** and **End Time** if you want to trim data from rosbag. You can use the sliders or manually type in your time values.
- Select the desired downsample percentage of scans in **Downsample Scans**. This evenly downsamples the scans based on the percentage. For example, a value of 25% would only select every fourth scan.
- Click **Apply** to see the new filtered scans and apply all settings. **Close** the tab when you are done.

If you'd like more control over filtering scans in the rosbag, import your rosbag into MATLAB using `rosbag`. Filter the rosbag using `select`. To open the app using your custom filtered rosbag, see Programmatic Use on page 5-13.

## Tune SLAM Settings

To improve the automatic map building process, the SLAM algorithm has tunable parameters. Click **SLAM Settings** to tune the parameters. Use **Lidar SLAM Parameters** to affect different aspects of the scan alignment and loop closure detection processes. Also, tune the **NLP Solver Parameters** to change how the map optimization algorithm improves the overall map based on loop closures.

### Lidar SLAM Parameters:

- **Map Resolution (cells/m)** -- Resolution of the map. The resolution affects the location accuracy of the scan alignment and defines the output size of the occupancy grid.
- **Lidar Range [min,max] (m)** -- Range of lidar sensor readings. When processing the lidar scans, readings outside of the lidar range are ignored.
- **Loop Closure Threshold** -- Unitless threshold for accepting loop closures. Depending on your lidar scans, the average loop closure score varies. If the build process does not find loop closures and the robot revisits locations in the map, consider lowering this threshold.
- **Loop Closure Search Radius (m)** -- Radius to search for loop closures. Based on the odometry pose, the algorithm searches for loop closures in the existing map within the given radius in meters.



- **Loop Closure Max Attempts** -- Number of attempts at finding loop closures. When this number increases, the algorithm makes more attempts to find loop closures in the map but increases computation time.
- **Loop Closure Auto Rollback** -- Allow automatic rejection of loop closures. The algorithm tracks the residual error from the map optimization. If it detects a sudden change in the error and this parameter is set to `on`, the loop closure is rejected.
- **Optimization Interval (# of Loop Closures)** -- Number of detected loop closures accepted to trigger optimization. By default, the map is optimized with every loop closure found.
- **Movement Threshold [Linear,Angular] (m,rad)** -- Minimum change in pose required to accept a new scan. If the pose of the robot does not exceed this threshold, the next scan is discarded from the map building process.

#### NLP Solver Parameters:

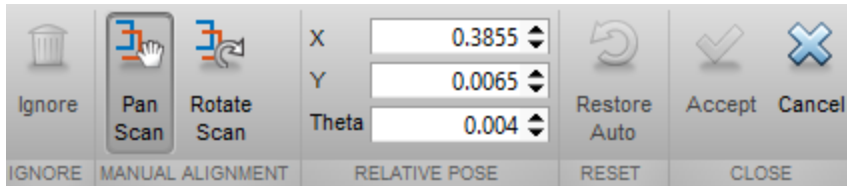
- **Max Iterations** -- Maximum number of iterations for map optimization. Increasing this value may improve map accuracy but increases computation time.
- **Max Time (s)** -- Maximum time allowed for map optimization specified in seconds. Increasing this value may improve map accuracy but increases computation time.
- **Gradient Tolerance** -- Lower bound on the norm of the gradient of the cost function for optimization. Lowering this value causes the optimization to run longer to search for a local minimum but increases the computation time.
- **Function Tolerance** -- Lower bound on the change in the cost function for optimization. Lowering this value causes the optimization to run longer to search for a local minimum but increases the computation time.
- **Step Tolerance** -- Lower bound on the step size for optimization. Lowering this value causes the optimization to run longer to search for a local minimum but increases the computation time.
- **First Node Pose [x,y,theta] (m,rad)** -- Pose of the first node in the graph. If you need to offset the position of the scans in the map, specify the position,  $[x \ y]$ , in meters and orientation,  $\theta$ , in radians.

After changing any of these settings, the map building process must be restarted to rebuild the map with the new parameters.

## Modify Incremental Matches and Loop Closures

This app allows you to manually modify incremental scans and adjust detected loop closures. If you notice scans are not properly aligned after you build the map, use the

**Incremental Match** and **Loop Closure** buttons to open their modification tabs. Use the modification toolstrip buttons to adjust the relative pose between scans.



- **Ignore** -- When modifying loop closures, you can simply ignore loop closures if they are inaccurate. The algorithm always discards ignored loop closure if detected in the same app session. You cannot ignore incremental scan matches.
- **Pan Scan** -- Click this button to manually shift the pose. After selecting, click and drag inside the map to shift the scans and overlay them properly. Align all the points of the scans until you are satisfied. You can manually specify the **X**, **Y** location in the **Relative Pose** section as well.
- **Rotate Scan** -- Click this button to manually rotate the pose. After selecting, click and drag inside the map to rotate the scans and overlay them properly. Align all the points of the scans until you are satisfied. You can manually specify the **Theta** location in the **Relative Pose** section as well.

## Sync the Map

After making modifications to the map building process using **Incremental Scans** and **Loop Closures**, you must sync the map to apply the changes. Based on the changes you make to properly align scans, the overall map shifts and alignments change for every scan after your modification. You have two options after making your modifications, **Sync** or **Sync Fast**. If you click **Sync Fast**, the changes to the poses are automatically applied and no other changes to the map occur. **Sync** restarts the entire map building and loop closure detection processes starting at the first modification. The specified modifications are applied, but the algorithm attempts to realign other scans and search for new loop closures as well.

## See Also

### Functions

`buildMap` | `matchScans` | `matchScansGrid` | `optimizePoseGraph` | `roslaunch`

**Objects**

lidarScan | robotics.LidarSLAM | robotics.OccupancyGrid |  
robotics.PoseGraph

**Topics**

“Implement Simultaneous Localization And Mapping (SLAM) with Lidar Scans”

“Implement Online Simultaneous Localization And Mapping (SLAM) with Lidar Scans”

**Introduced in R2018b**

