How to Contact MathWorks

Latest news:	www.mathworks.com
Sales and services:	www.mathworks.com/sales_and_services
User community:	www.mathworks.com/matlabcentral
Technical support:	www.mathworks.com/support/contact_us
Phone:	508-647-7000

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

Robotics System Toolbox® Reference

É COPYRIGHT 2015ð2017 by The MathWorks, Inc.

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

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

Trademarks

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

Patents

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

Revision History

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)

Contents

1 Classes ñ Alphabetical List

2 Functions ñ Alphabetical List

3 Methods ñ Alphabetical List

4 Blocks ñ Alphabetical List

Classes مم Alphabetical List

BagSelection

Create 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.

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 located at 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 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.

See `select` for other syntaxes. You can specify certain parameters to filter by such as time and topic.

Properties

`FilePath` Absolute path to the `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 the first message in the 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 the last message in the 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 the 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 selection.

Data Types: `double`

`AvailableTopics` Table of topics in the 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                                MessageDefiniti
            ___________    __________________    ____________________________________________

    /odom    99             nav_msgs/Odometry    '# This represents an estimate of a po
```

Data Types: `table`

`MessageList` ⇒ List of messages in the 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

readMessages	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

Create a `BagSelection` object from a `rosbag` log file and parse out specific messages based on the selected criteria.

Set the path to the logfile

```
filepath = fullfile(fileparts(which('ROSWorkingWithRosbagsExample')), 'data', 'ex_multi
```

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

```
bagMsgs = robotics.ros.Bag.parse(filepath);
```

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');
```

Å òWork with rosbag Logfilesó

See Also

readMessages | select | timeseries

Topics

òWork with rosbag Logfilesó
òROS Log Files (rosbags)ó

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 th 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 th ROS network

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

Å òConnect to a ROS Networkó

See Also

`Node` | `rosinit`

Topics

òConnect to a ROS Networkó
òROS Network Setupó

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 = rosmessage('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
```

Å òWork with Specialized ROS Messagesó

See Also

readImage | rosmessage | rossubscriber

Topics

òWork with Specialized ROS Messagesó

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` مم 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`.

Å `true` ñ Big endian sequence. Stores the most significant byte in the smallest address.

Å `false` ñ 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` مم 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 MATLABÈ image to ROS message.

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

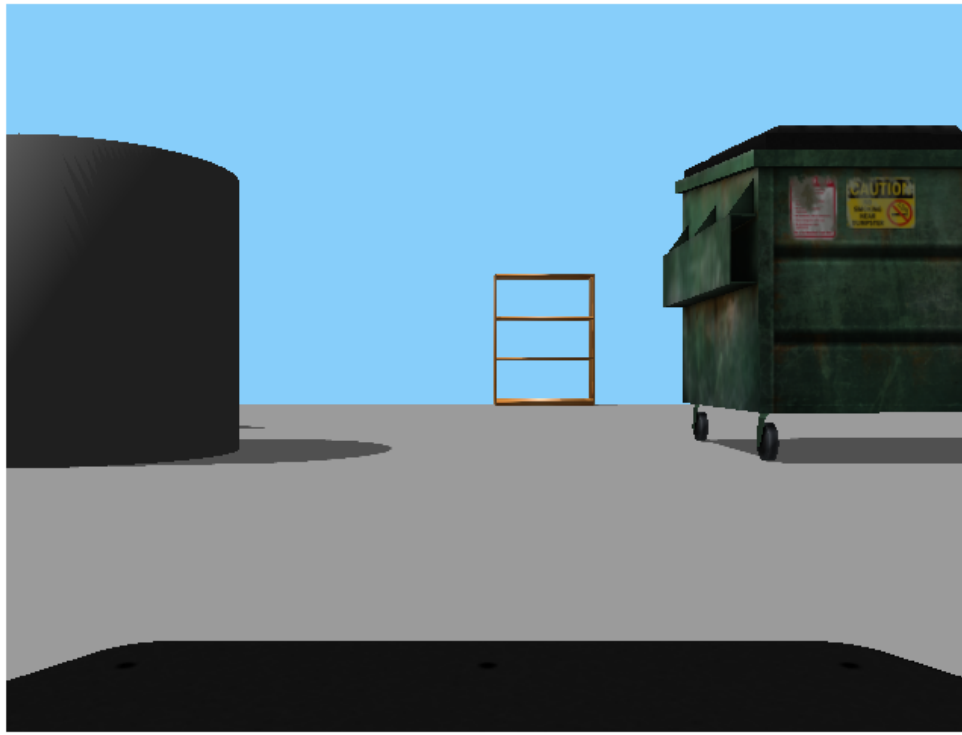
```
exampleHelperROSLoadMessages
img
```

```
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 = rosmessage('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 = rosmessage('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
```

Å òWork with Specialized ROS Messagesó

See Also

`readImage` | `rosmessage` | `rossubscriber` | `writeImage`

Topics

òWork with Specialized ROS Messagesó

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` مم 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` مم Minimum valid range value
scalar

Minimum valid range value, specified as a scalar.

`RangeMax` مم Maximum valid range value
scalar

Maximum valid range value, specified as a scalar.

`Ranges` مم Range readings from laser scan
vector

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

`Intensities` مم 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

lidarScan	Creat object for storing 2-D lidar scan
plot	Display laser or lidar scan readings
readCartesian	Read laser scan ranges in Cartesian coordinates
readScanAngles	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
```

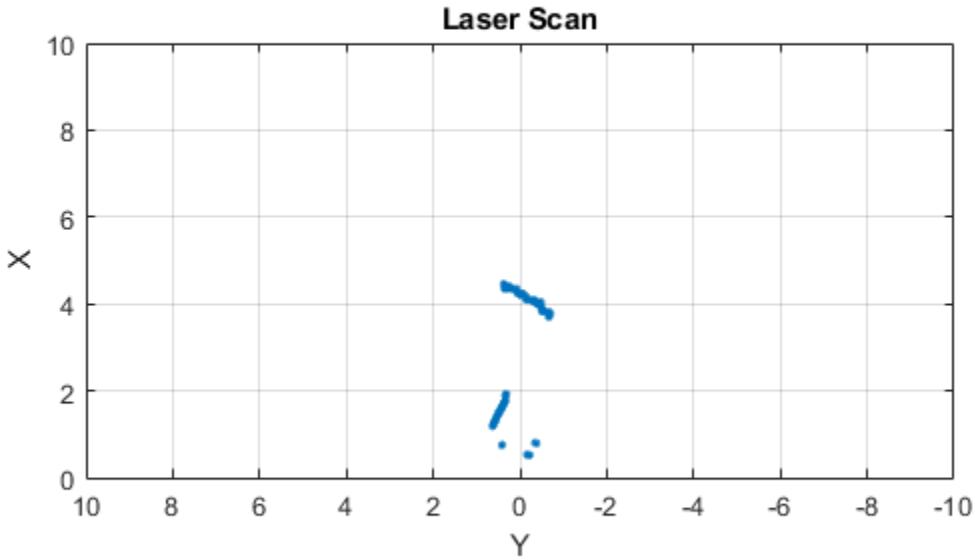
Get ranges and angles from the object properties. Check that the ranges and angles are the same size.

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

```
ans =

   640     1


ans =

   640     1
```

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

```
plot(scan)
```



Create Empty `LaserScan` Message

```
scan = rosmessage('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
```

Å òWork with Specialized ROS Messagesó

See Also

lidarScan | plot | readCartesian | readScanAngles | rosmessage | rossubscriber | showdetails

Topics

òWork with Specialized ROS Messagesó

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` ñ 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` ñ 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

character vector

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

`MasterURI` URI of the ROS master

character vector

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

`NodeURI` URI for the node

character vector

URI for the node, specified as a 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 MATLABÈ. These seperate 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` | `rosshutdown`

Topics

òROS Network Setupó

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

Description

`msg = rosmessage('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` ممم 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.

```
filePath = fullfile(fileparts(which('PathPlanningExample')),'data','exampleMaps.mat');
load(filePath)
whos *Map*
```

```
  Name              Size                Bytes  Class      Attributes

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

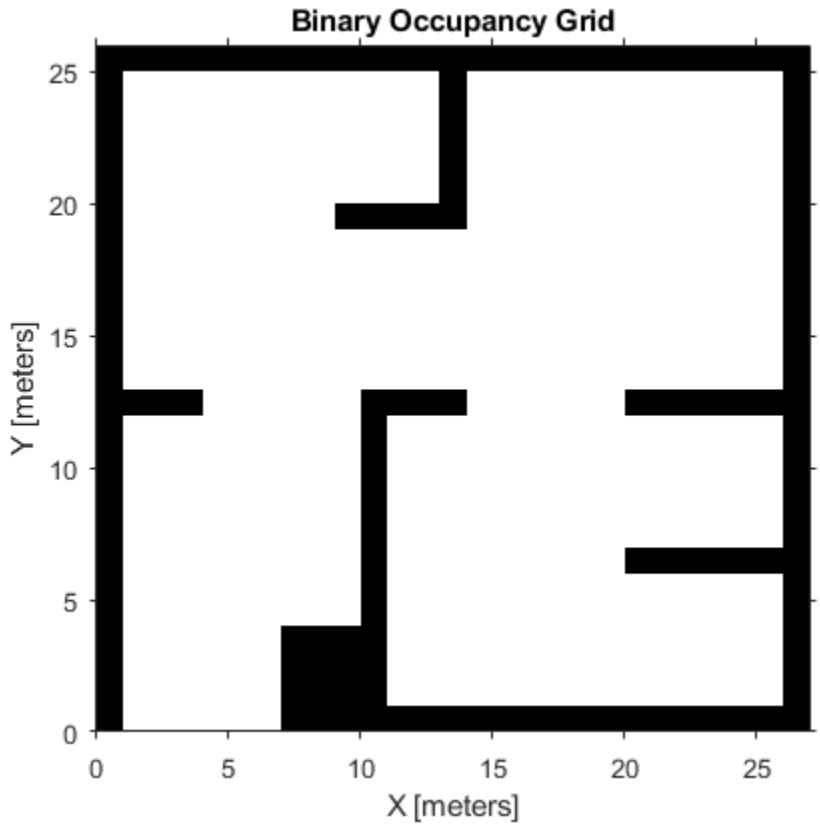
Create a ROS message from `simpleMap` using a `BinaryOccupancyGrid` object. Write the `OccupancyGrid` message using `writeBinaryOccupancyGrid`.

```
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

òOccupancy Gridsó

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 `flase` 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` ñ Big endian sequence. Stores the most significant byte in the smallest address.
- `false` ñ 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 م 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 م Point cloud data

`uint8` array

Point cloud data, specified as a `uint8` array. To access the data, use the òObject Functionsó on page 1-36.

Object Functions

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

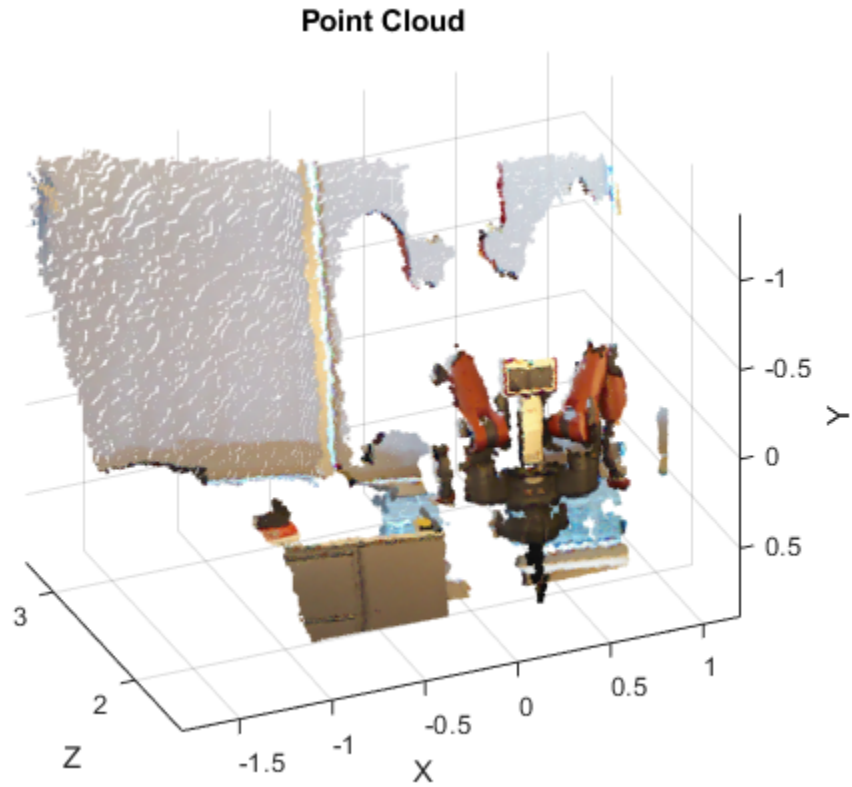
```
                MessageType: 'sensor_msgs/PointCloud2'
                     Header: [1x1 Header]
                     Height: 480
                      Width: 640
                IsBigendian: 0
                  PointStep: 32
                    RowStep: 20480
                    IsDense: 0
                     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 Ê point cloud message into a Computer Vision System Toolbox Ê `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]
```

Å òWork with Specialized ROS Messagesó

See Also

readAllFieldNames | readField | readRGB | readXYZ | rosmessage | rossubscriber | scatter3 | showdetails

Topics

òWork with Specialized ROS Messagesó

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 òGenerate a Standalone ROS Node from SimulinkÈó.

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

runNode	Start ROS node
stopNode	Stop ROS node
isNodeRunning	Determine if ROS node is running
runCore	Start ROS core
stopCore	Stop ROS core
isCoreRunning	Determine if ROS core is running
system	Execute system command on device
putFile	Copy file to device
getFile	Get file from device
deleteFile	Delete file from device
dir	List folder contents on device
openShell	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 = '192.168.154.131';
d = rosdevice(ipaddress,'user','password');
d.ROSFolder = '/opt/ros/hydro';
d.CatkinWorkspace = '~/catkin_ws_test'
```

```
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. Connect MATLABÈ to the ROS master using `rosinit`. This core enables you to run ROS nodes on your ROS device.

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

```
Another roscore / ROS master is already running on the ROS device. Use the 'stopCore' f
Initializing global node /matlab_global_node_15046 with NodeURI http://192.168.154.1:56
```

Check the available ROS nodes on the connected ROS device. These nodes were generated from SimulinkÈ models following the process in the Generate a standalone ROS node from SimulinkÈ 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_15046 with NodeURI http://192.168.154.1:5
```

Run Multiple ROS Nodes

Run multiple ROS nodes on a connected ROS device. ROS nodes can be generated using SimulinkÈ 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 SimulinkÈ). 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 = '192.168.154.131';
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. Connect MATLABÈ to the ROS master using `rosinit`. This ROS core enables you to run ROS nodes on your ROS device.

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

```
Initializing global node /matlab_global_node_68749 with NodeURI http://192.168.154.1:64
```

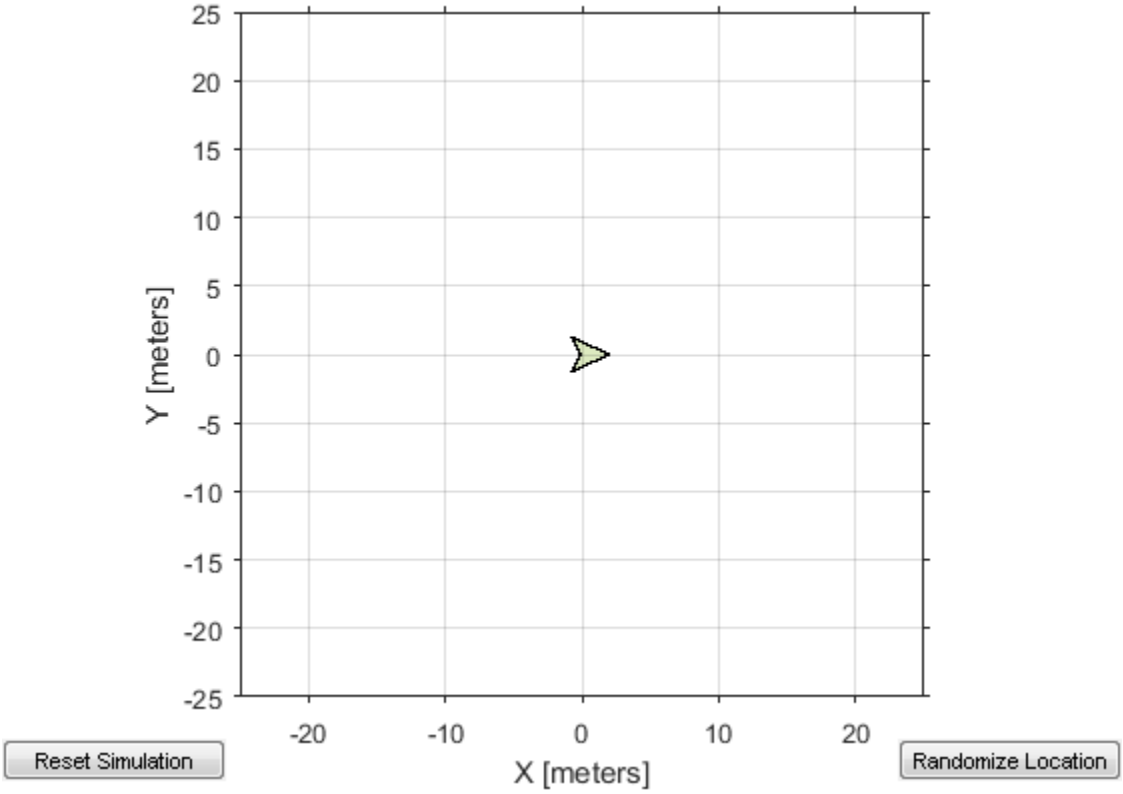
Check the available ROS nodes on the connected ROS device. These nodes were generated from SimulinkÈ models following the process in the Generate a standalone ROS node from SimulinkÈ example.

```
d.AvailableNodes
```

```
ans =

  1×2 cell array

    'robotcontroller'    'robotcontroller2'
```

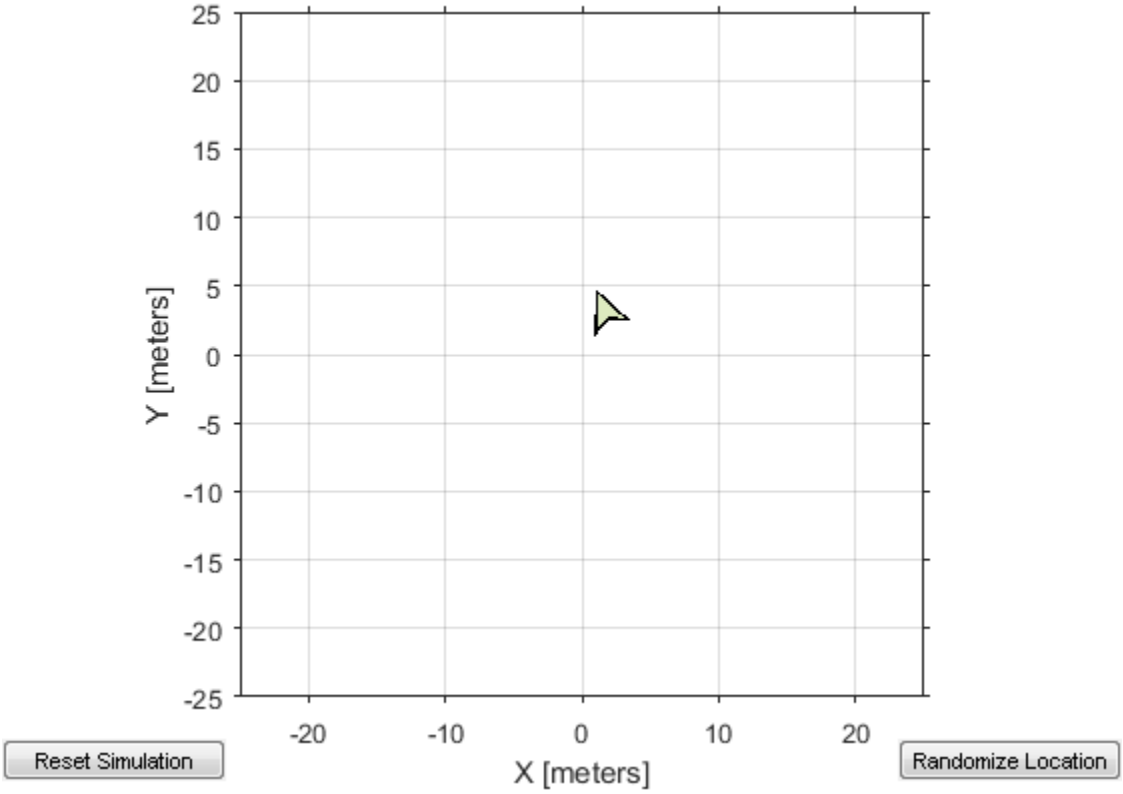
Start up the Robot Simulator using `ExampleHelperSimulinkRobotROS`. This simulator automatically connects to the ROS master on the ROS device. You will use this simulator to run a ROS node and control the robot.

```
sim = ExampleHelperSimulinkRobotROS;
```



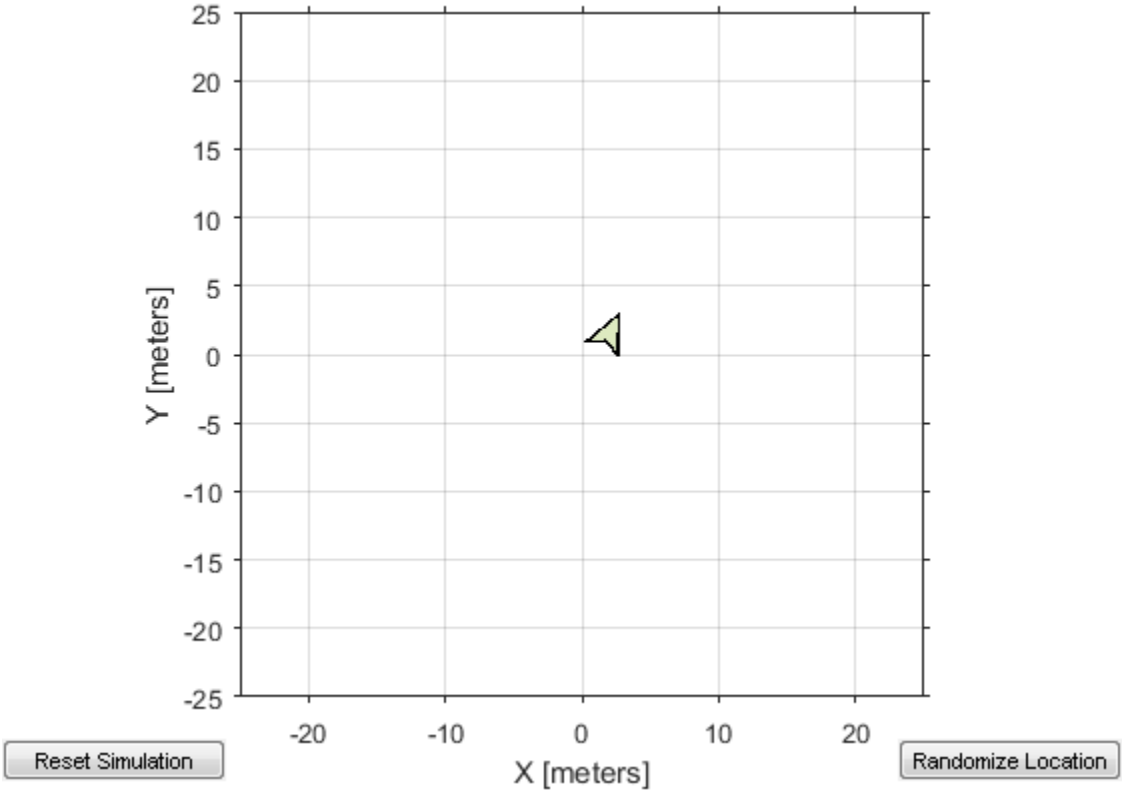
Run a ROS node, specifying the node name. The `'robotcontroller'` node commands the robot to a specific location (`[-10 10]`). Wait to see the robot drive.

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



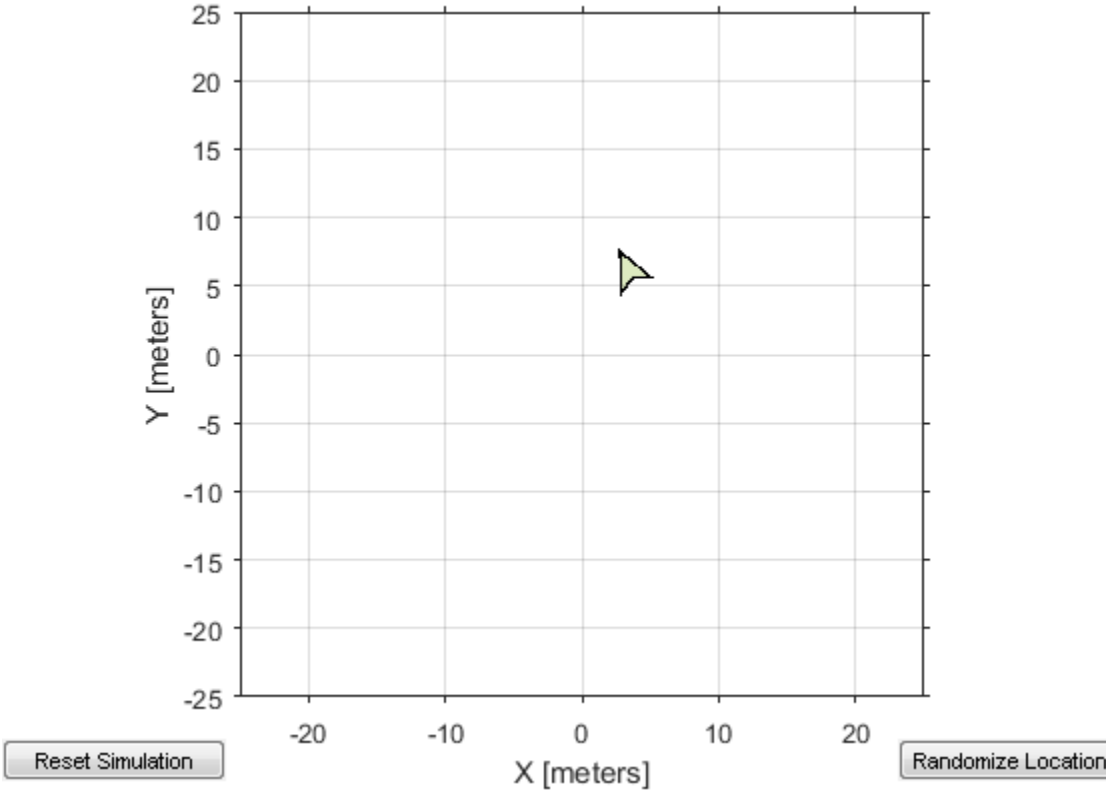
Reset the Robot Simulator to reset the robot position. Alternatively, click Reset Simulation. Because the node is still running, the robot continues back to the specific location. To stop sending commands, stop the node.

```
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.

```
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')
rosshutdown
stopCore(d)
```

```
Shutting down global node /matlab_global_node_68749 with NodeURI http://192.168.154.1:6
```

Å òGenerate a Standalone ROS Node from SimulinkÈó

See Also

`isNodeRunning` | `runCore` | `runNode` | `stopNode`

Topics

òGenerate a Standalone ROS Node from SimulinkÈó

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` مم 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
exampleHelperROSStartTfPublisher
```

```
Initializing ROS master on http://bat5823win64:54708/.
Initializing global node /matlab_global_node_07637 with NodeURI http://bat5823win64:547
```

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     :  12219
    FrameId :  camera_center
    Stamp
      Sec  :  1504299019
      Nsec :  0
  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.

```
rosshutdown
```

```
Shutting down global node /matlab_global_node_07637 with NodeURI http://bat5823win64:54
Shutting down ROS master on http://bat5823win64:54708/.
```

Apply Transformation using `TransformStamped` Object

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

Start ROS network and setup transformations.

```
rosinit
exampleHelperROSStartTfPublisher
```

```
Initializing ROS master on http://bat5823win64:54741/.
Initializing global node /matlab_global_node_33561 with NodeURI http://bat5823win64:547
```

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     :  12230
    FrameId :  camera_center
    Stamp
      Sec  :  1504299022
      Nsec :  119000064
  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 = rosmessage('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.

```
rosshutdown
```

```
Shutting down global node /matlab_global_node_33561 with NodeURI http://bat5823win64:54
Shutting down ROS master on http://bat5823win64:54741/.
```

Å òAccess the tf Transformation Tree in ROSó

See Also

Functions

apply | getTransform | rostf | transform | waitForTransform

Topics

òAccess the tf Transformation Tree in ROSó

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 òPlan a Reaching Trajectory With Multiple Kinematic Constraintsó.

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.

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

Input Arguments

`endeffector` مم End-effector name

character vector

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

Example: `'left_palm'`

Properties

`EndEffector` مم Name of the end effector

character vector

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

Example: `'left_palm'`

`ReferenceBody` مم 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`).

`TargetPoint` مم 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` مم 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 MATLABÈ Coder Ê.

See Also

Classes

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

Topics

òPlan a Reaching Trajectory With Multiple Kinematic Constraintsó

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` مم **Map width**
double in meters

Map width, specified as a double in meters.

Data Types: `double`

`height` مم **Map height**
double in meters

Map width, specified as a double in meters.

Data Types: `double`

`resolution` مم **Grid resolution**
1 (default) | double in cells per meter

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

Data Types: `double`

`p` مم **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

Examples

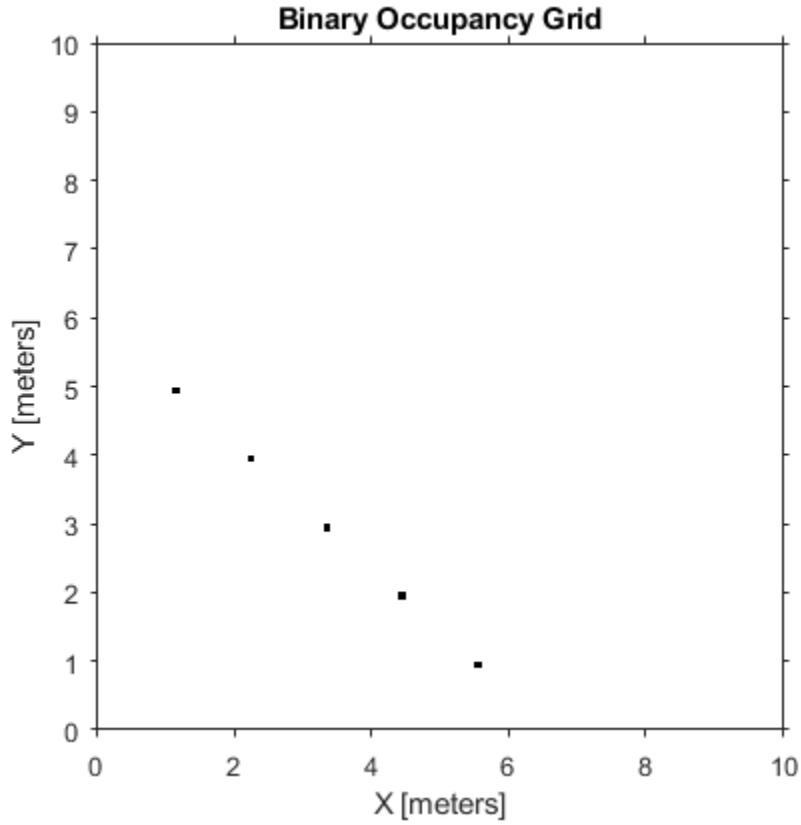
Create and Modify Binary Occupancy Grid

Create a 10m x 10m empty map.

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

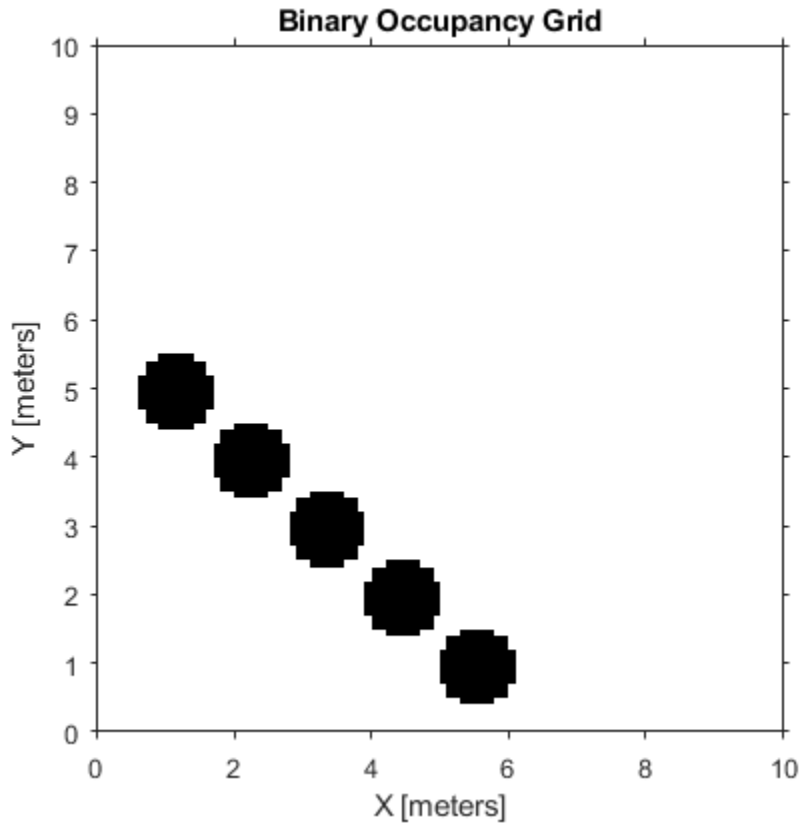
Set occupancy of world locations and show map.

```
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)
```

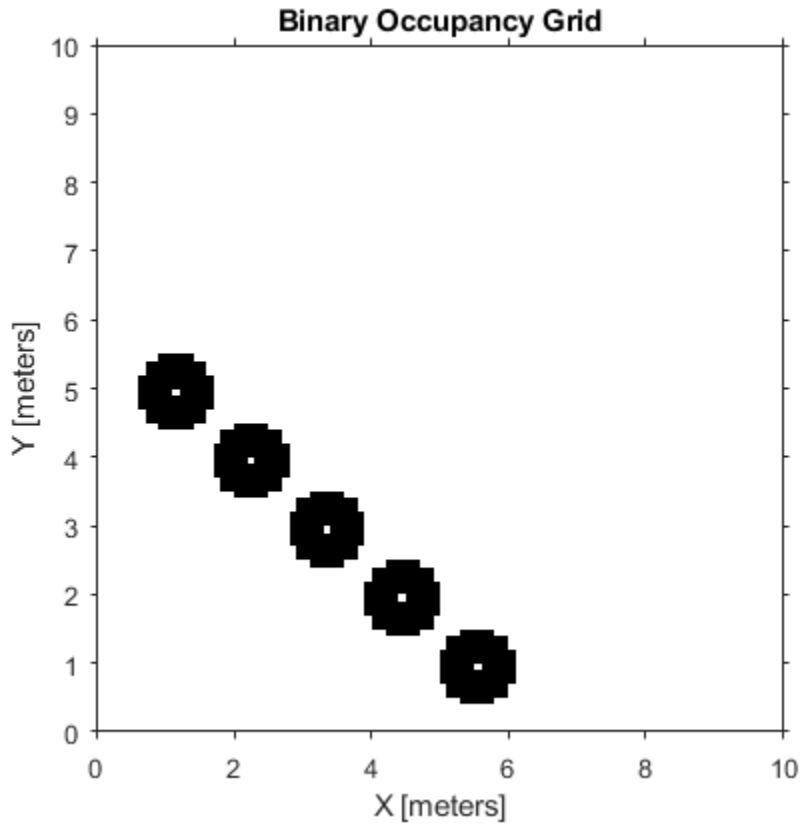
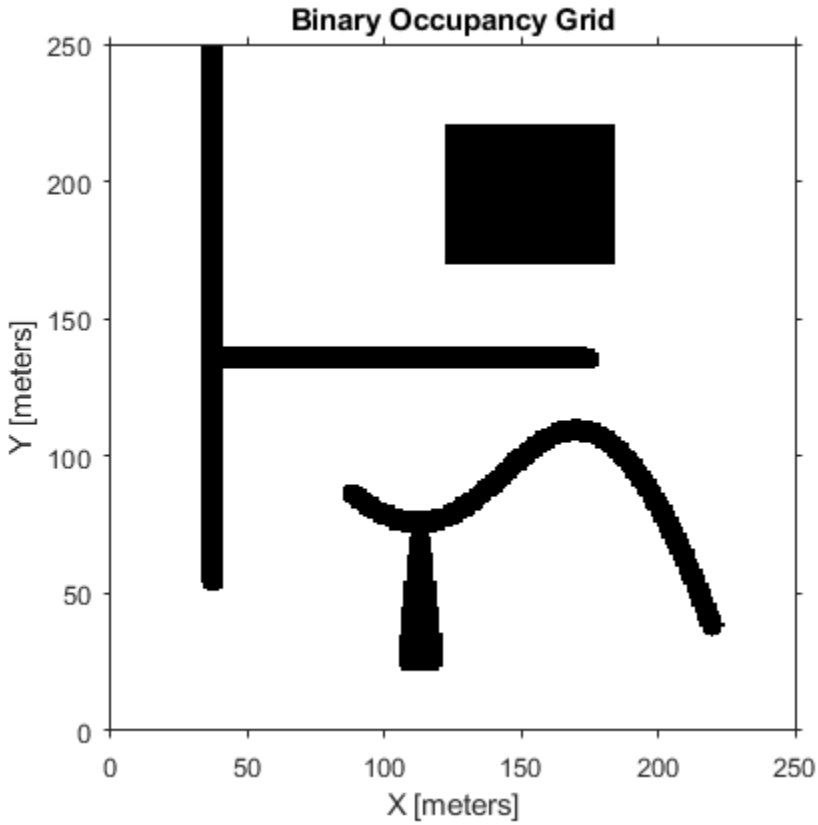


Image to Binary Occupancy Grid Example

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

```
% 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)
```

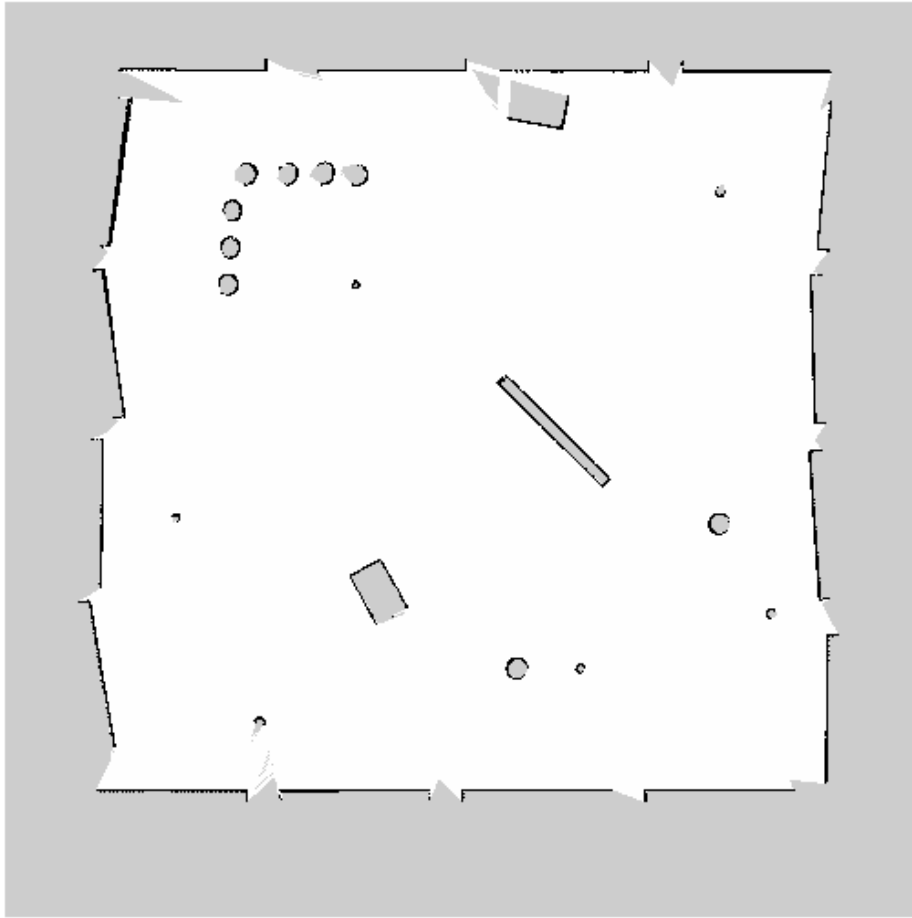


Convert PGM Image to Map

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

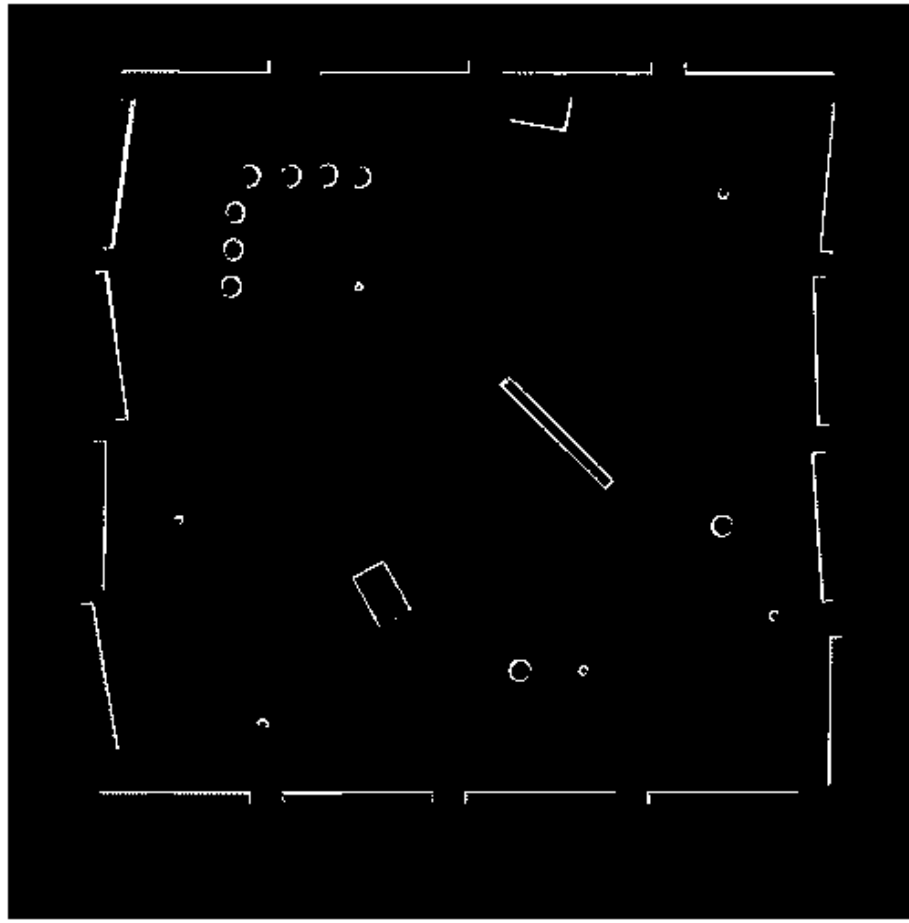
Import image using `imread`. The image is quite large and should be cropped to the relevant area.

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



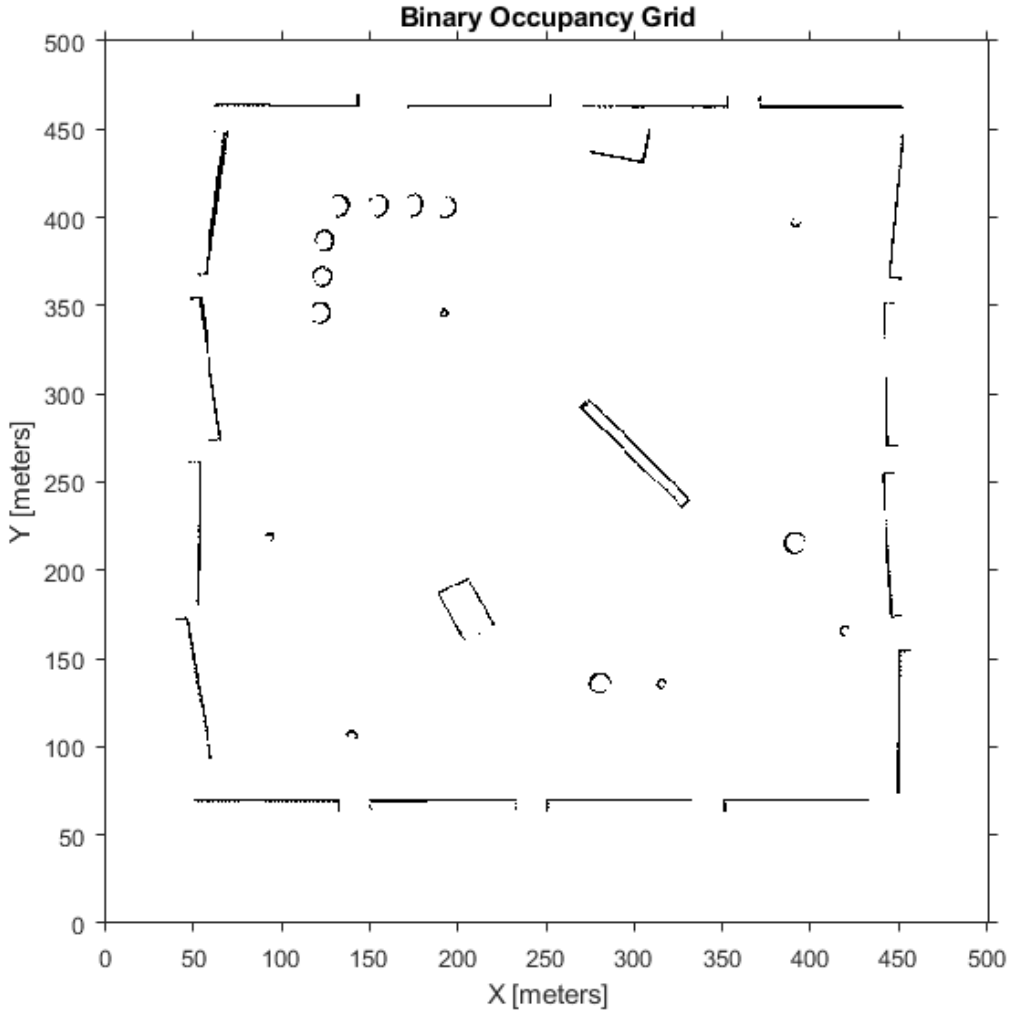
Unknown areas (gray) should be removed and treated as free space. Create a logical matrix based on a threshold. Depending on your image, this value could be different. Occupied space should be set as 1 (white in image).

```
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 MATLABÈ Coder Ê.

See Also

robotics.PRM | robotics.PurePursuit

Topics

òOccupancy Gridsó

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 òPlan a Reaching Trajectory With Multiple Kinematic Constraintsó.

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.

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

Input Arguments

`endeffector` ⌘ End-effector name

character vector

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

Example: `'left_palm'`

Properties

`EndEffector` ⌘ Name of the end effector

character vector

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

Example: `'left_palm'`

`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`).

`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 MATLABÈ Coder Ê.

See Also

Classes

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

Topics

òPlan a Reaching Trajectory With Multiple Kinematic Constraintsó

Introduced in R2017a

robotics.GeneralizedInverseKinematics System object

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 (or using the `step` method). To change constraint inputs after calling the object, call `release`.

Specify the constraint inputs as constraint objects and call `GeneralizedInverseKinematics` with these object 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 generalize 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'` ñ `OrientationTarget`
- `'position'` ñ `PositionTarget`
- `'pose'` ñ `PoseTarget`
- `'aiming'` ñ `AimingConstraint`
- `'cartesian'` ñ `CartesianBounds`
- `'joint'` ñ `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 prior to 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 òInverse Kinematics Algorithmsó.

`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 òSolver Parametersó.

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` مم **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` مم **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` ñ Character vector for the name of the joint specified in the `RigidBodyTree` robot model
- `JointPosition` ñ 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` ñ Number of iterations run by the solver.
- `NumRandomRestarts` ñ Number of random restarts because the solver got stuck in a local minimum.
- `ConstraintViolation` ñ 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.

Å `ExitFlag` ñ Code that gives more details on the solver execution and what caused it to return. For the exit flags of each solver type, see òExit Flagsó.

Å `Status` ñ Character vector describing whether the solution is within the tolerances defined by each constraint (`'success'`) or outside the tolerance and is the best possible solution that the solver 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

step	Run System object algorithm
clone	Create duplicate System object
getNumInputs	Number of inputs required to call the System object
getNumOutputs	Number of outputs from calling the System object
isLocked	Determine if System object is locked
release	Release resources and allow changes to System object property values and input characteristics
reset	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 Ê 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.

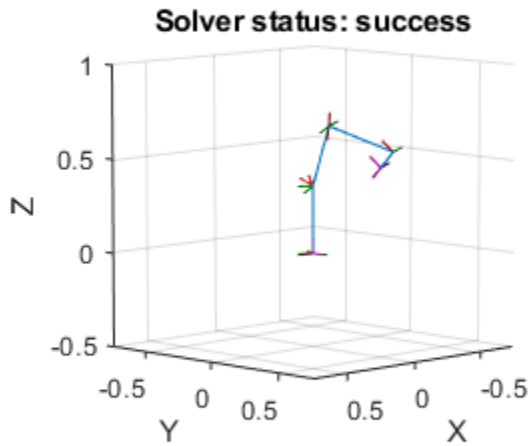
```
posTgt = robotics.PositionTarget('tool0');
posTgt.TargetPosition = [0.0 0.5 0.5];

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

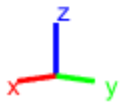
Find a configuration that satisfies the constraints. You must pass the constraint objects into the System object in the order in which they were specified in the `ConstraintInputs` property. Specify an initial guess at the robot configuration.

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

Visualize the configuration returned by the solver.

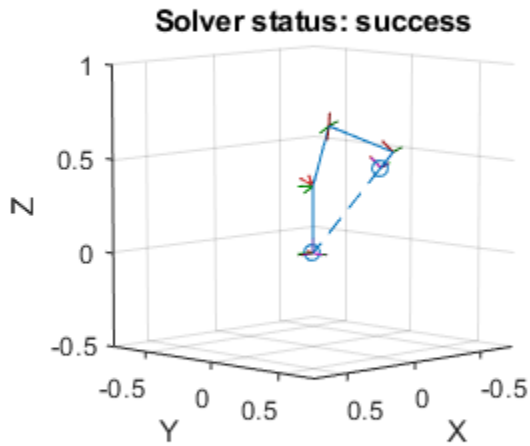
```
show(lbr,q);
title(['Solver status: ' solutionInfo.Status])
axis([-0.75 0.75 -0.75 0.75 -0.5 1])
```

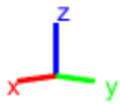




Plot a line segment from the target position to the origin of the base. The origin of the `tool0` frame coincides with one end of the segment, and its `z`-axis is aligned with the segment.

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





Å òControl PR2 Arm Movements Using ROS Actions and Inverse Kinematicsó

Å òPlan a Reaching Trajectory With Multiple Kinematic Constraintsó

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLABÈ Coder Ê.

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

```
gik = robotics.GenerlizedInverseKinematics(...
    'ConstraintInputs',{'pose','position'});
```

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

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

See Also

Classes

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

Topics

òControl PR2 Arm Movements Using ROS Actions and Inverse Kinematicsó
òPlan a Reaching Trajectory With Multiple Kinematic Constraintsó

Introduced in R2017a

robotics.InverseKinematics System object

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. You must 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 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` مم 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 prior to 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 òInverse Kinematics Algorithmsó.

`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 òSolver Parametersó.

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` م 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` م Weight for pose tolerances

6-element vector

Weight for pose tolerances, specified as a 6-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` م Initial guess of robot configuration

structure array

Initial guess of robot configuration, specified as a structure array. 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.

Output Arguments

`configSol` م Robot configuration solution

structure array

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

Å `JointName` ñ Character vector for the name of the joint specified in the `RigidBodyTree` robot model

Å `JointPosition` ñ Position of the corresponding joint

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

`solInfo` م Solution information

structure

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

- `Iterations` ñ Number of iterations run by the algorithm.
- `NumRandomRestarts` ñ Number of random restarts because algorithm got stuck in a local minimum.
- `PoseErrorNorm` ñ The magnitude of the pose error for the solution compared to the desired end-effector pose.
- `ExitFlag` ñ Code that gives more details on the algorithm execution and what caused it to return. For the exit flags of each algorithm type, see òExit Flagsó.
- `Status` ñ 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

step	Run System object algorithm
clone	Create duplicate System object
getNumInputs	Number of inputs required to call the System object
getNumOutputs	Number of outputs from calling the System object
isLocked	Determine if System object is locked
release	Release resources and allow changes to System object property values and input characteristics
reset	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.

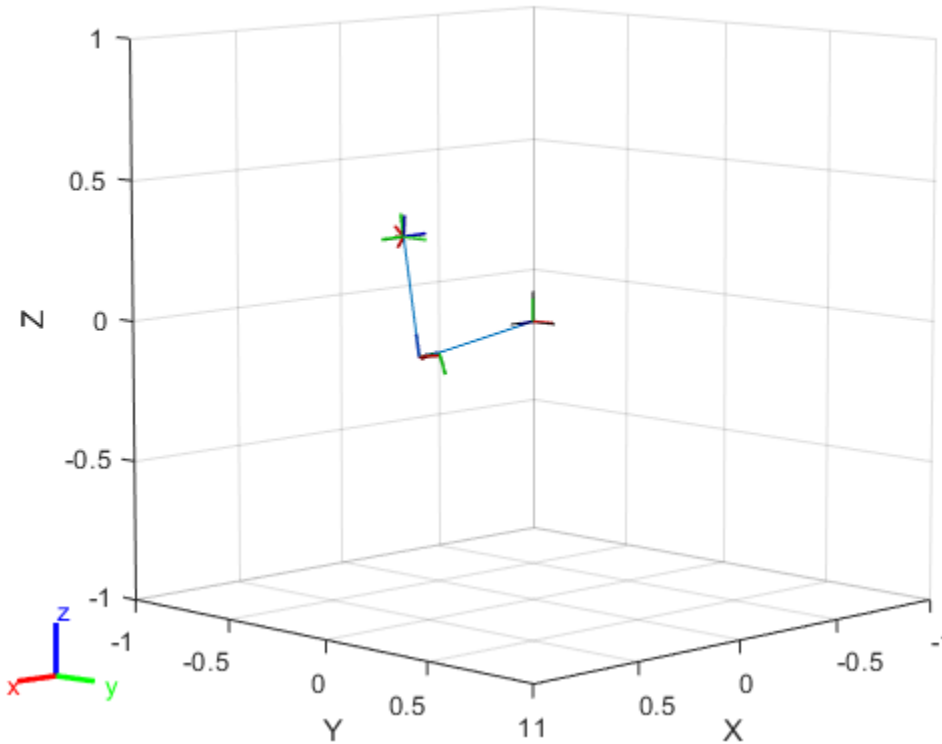
Load example robots. The `puma1` robot is a `RigidBodyTree` model of a six-axis robot arm with six revolute joints.

```
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 tranformation 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.

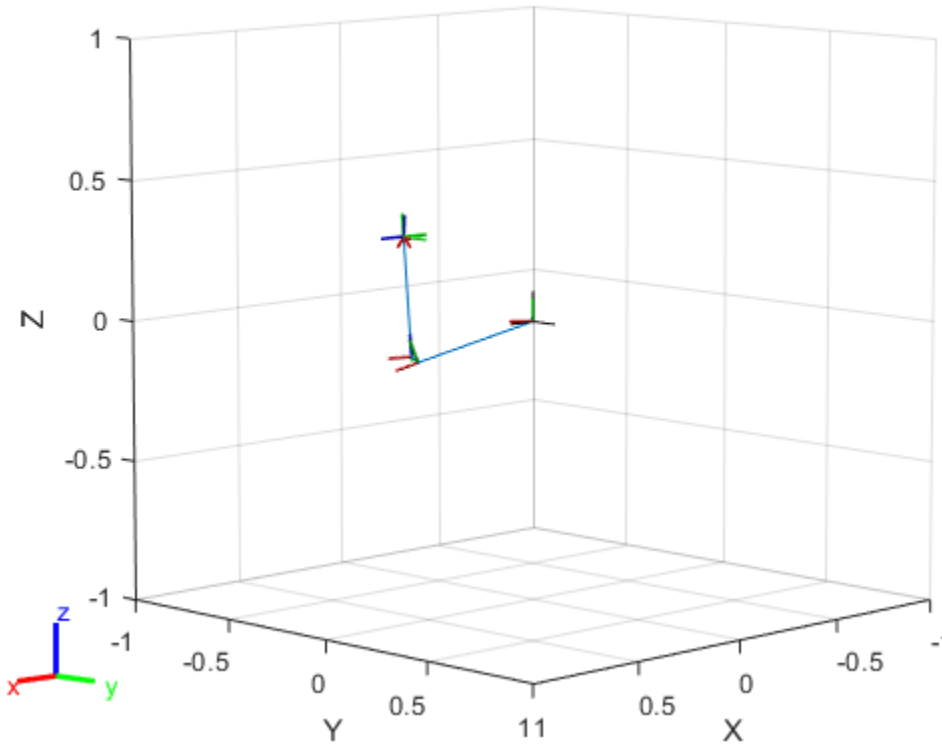
```
ik = robotics.InverseKinematics('RigidBodyTree',puma1);
weights = [0.25 0.25 0.25 1 1 1];
initialguess = puma1.homeConfiguration;
```

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);
```



Å òControl PR2 Arm Movements Using ROS Actions and Inverse Kinematicsó

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): 1ð16. doi: 10.1016/j.jcp.2013.08.044.

[2] Bertsekas, Dimitri P. *Nonlinear Programming*. Belmont, MA: Athena Scientific, 1999.

[3] Goldfarb, Donald. "Extension of Davidonîs Variable Metric Method to Maximization Under Linear Inequality and Equality Constraints." *SIAM Journal on Applied Mathematics*. Vol. 17, No. 4 (1969): 739ð64. 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 LevenbergðMarquardt Method." *IEEE Transactions on Robotics* Vol. 27, No. 5 (2011): 984ð91. 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): 313ð36. doi:10.1145/195826.195827.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLABÈ Coder Ê.

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

òControl PR2 Arm Movements Using ROS Actions and Inverse Kinematicsó
òInverse Kinematics Algorithmsó
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:

Å `'fixed'` ñ Fixed joint that prevents relative motion between two bodies.

Å `'revolute'` ñ Single degree of freedom (DOF) joint that rotates around a given axis. Also called a pin or hinge joint.

Å `'prismatic'` ñ 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 مم Joint name

character vector

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

Example: `'elbow_right'`

jtype مم Joint type

`'fixed'` (default) | character vector

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

The different joint types supported are:

Å `'fixed'` ñ Fixed joint that prevents relative motion between two bodies.

Å `'revolute'` ñ Single degree of freedom (DOF) joint that rotates around a given axis. Also called a pin or hinge joint.

Å `'prismatic'` ñ Single DOF joint that slides along a given axis. Also called a sliding joint.

Properties

Type مم Joint type

`'fixed'` (default) | character vector

This property is read-only.

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

The different joint types supported are:

Å `'fixed'` ñ Fixed joint that prevents relative motion between two bodies.

Å `'revolute'` ñ Single degree of freedom (DOF) joint that rotates around a given axis. Also called a pin or hinge joint.

Å `'prismatic'` ñ 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`.

`Name` ™ Joint name

character vector

Joint name, returned as a 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'`

`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.

Å `'fixed'` ñ `[NaN NaN]` (default). A fixed joint has no joint limits. Bodies remain fixed between each other.

Å `'revolute'` ñ `[-pi pi]` (default). The limits define the angle of rotation around the axis in radians.

Å `'prismatic'` ñ `[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.

Å `'fixed'` ñ `0` (default). A fixed joint has no relevant home position.

Å `'revolute'` ñ `0` (default). A revolute joint has a home position defined by the angle of rotation around the joint axis in radians.

Å `'prismatic'` ñ `0` (default). A prismatic joint has a home position defined by the linear motion along the joint axis in meters.

`JointAxis` م 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 defintion.

Å `'fixed'` ñ A fixed joint has no relevant axis of motion.

Å `'revolute'` ñ A revolute joint rotates the body in the plane perpendicular to the joint axis.

Å `'prismatic'` ñ A prismatic joint moves the body in a linear motion along the joint axis direction.

`JointToParentTransform` م 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` م 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

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 Puma560È 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.

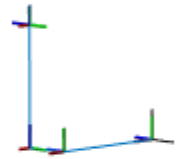
```
body2 = robotics.RigidBody('body2');
jnt2 = robotics.Joint('jnt2','revolute');
body3 = robotics.RigidBody('body3');
jnt3 = robotics.Joint('jnt3','revolute');
body4 = robotics.RigidBody('body4');
jnt4 = robotics.Joint('jnt4','revolute');
body5 = robotics.RigidBody('body5');
jnt5 = robotics.Joint('jnt5','revolute');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

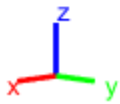
Verify that your robot was built properly by using the `showdetails` or `show` function. `showdetails` lists all the bodies in the MATLABÈ command window. `show` displays the robot with a given configuration (home by default). Calls to `axis` modify the axis limits and hide the axis labels.

```
showdetails(robot)

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

```
--------------------
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)
--------------------
```





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}
body3Copy = copy(body3);
```

```
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: {}
```

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)
--------------------
```

Å òBuild a Robot Step by Stepó

Å òRigid Body Tree Robot Modeló

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 MATLABÈ Coder Ê.

See Also

robotics.RigidBody | robotics.RigidBodyTree

Topics

òBuild a Robot Step by Stepó
òRigid Body Tree Robot Modeló
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 òPlan a Reaching Trajectory With Multiple Kinematic Constraintsó.

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.

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

Input Arguments

`robot` ممم Rigid body tree model
`RigidBodyTree` object

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

Properties

`Bounds` ممم 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` ممم 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 MATLABÈ Coder Ê.

See Also

Classes

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

Topics

òPlan a Reaching Trajectory With Multiple Kinematic Constraintsó

Introduced in R2017a

lidarScan

Creat 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

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

Examples

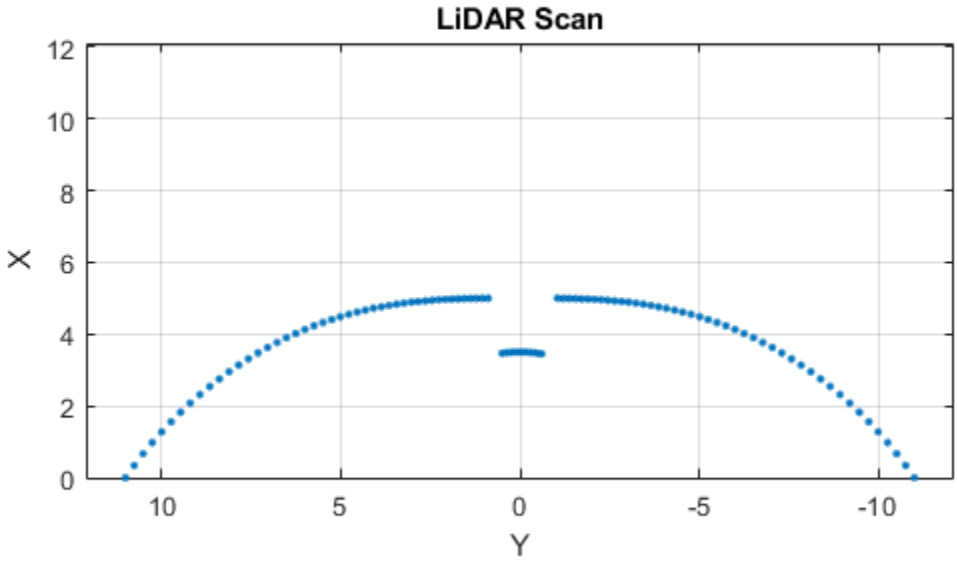
Plot Lidar Scan and Remove Invalid Points

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

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

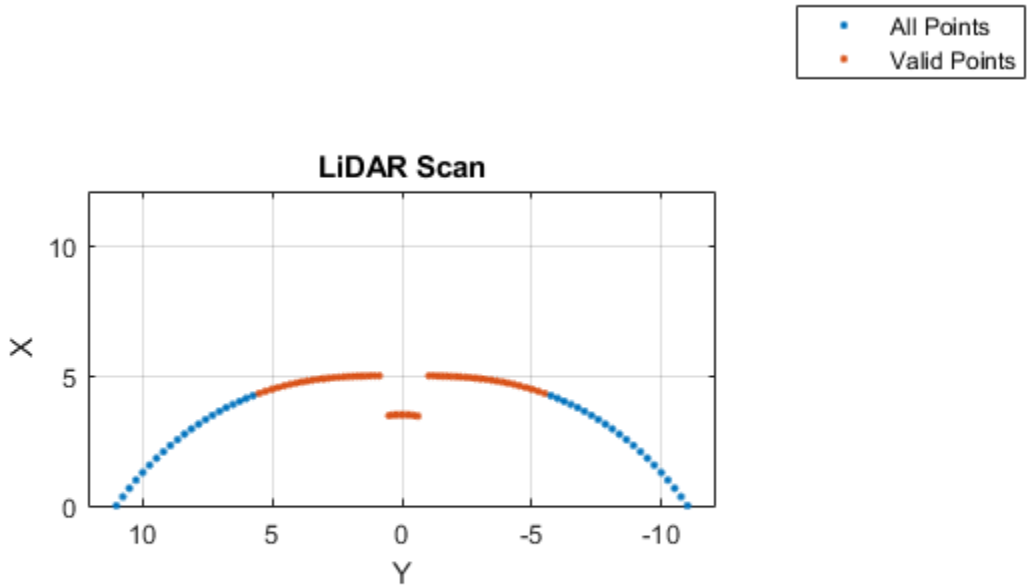
Create a lidar scan by specifying the ranges and angles. Plot all points of the lidar scan.

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



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

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



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)`.

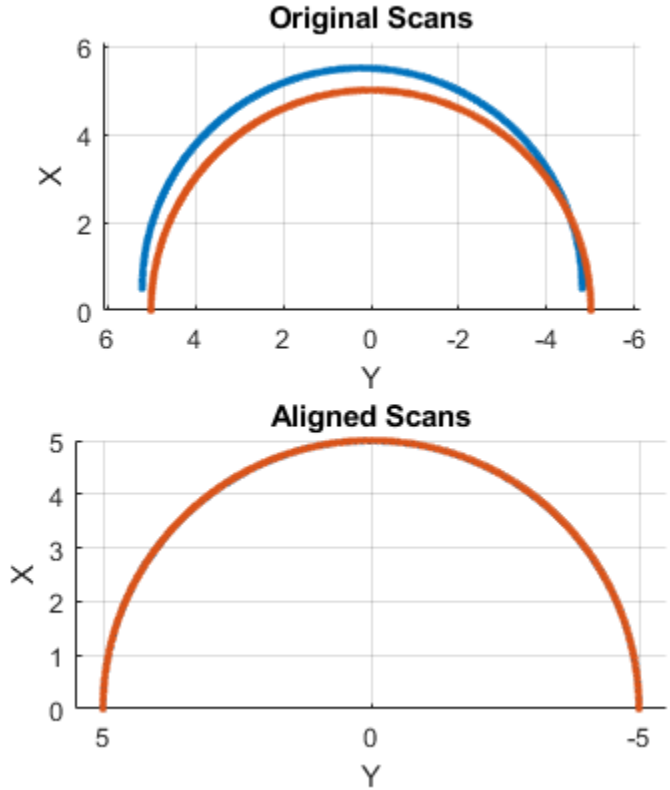
```
currScan = transformScan(refScan,[0.5 0.2 0]);
```

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 MATLABÈ Coder Ê.

See Also

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

Introduced in R2017b

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` مم 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` مم 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` مم 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 or 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 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 make changes to 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

òLocalize TurtleBot Using Monte Carlo Localizationó
òMonte Carlo Localization Algorithmó
Class Attributes (MATLAB)
Property Attributes (MATLAB)

Introduced in R2016a

robotics.MonteCarloLocalization System object

Package: robotics

Localize robot using range sensor data and map

Description

The `robotics.MonteCarloLocalization` 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
clone	Create duplicate System object
getNumInputs	Number of inputs required to call the System object
getNumOutputs	Number of outputs from calling the System object
isLocked	Determine if System object is locked
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 =

    0.0343    0.0193    0.0331


covariance =

    0.9467    0.0048         0
    0.0048    0.9025         0
         0         0    1.0011
```

Å òLocalize TurtleBot Using Monte Carlo Localizationó

References

[1] Thrun, Sebatian, Wolfram Burgard, and Dieter Fox. *Probabilistic Robotics*. MIT Press, 2005.

[2] Dellaert, F., D. Fox, W. Burgard, and S. Thrun. "Monte Carlo Localization for Mobile Robots." *Proceedings 1999 IEEE International Conference on Robotics and Automation*.

See Also

`lidarScan` | `robotics.LikelihoodFieldSensorModel` | `robotics.OdometryMotionModel`

Topics

òLocalize TurtleBot Using Monte Carlo Localizationó
òMonte Carlo Localization Algorithmó
Class Attributes (MATLAB)
Property Attributes (MATLAB)

Introduced in R2016a

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` مم Map width

scalar in meters

Map width, specified as a scalar in meters.

Data Types: `double`

`height` مم Map height

scalar in meters

Map height, specified as a scalar in meters.

Data Types: `double`

`resolution` مم Grid resolution

1 (default) | scalar in cells per meter

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

Data Types: `double`

p مم **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` مم **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` مم **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` مم **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` مم **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

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);
```

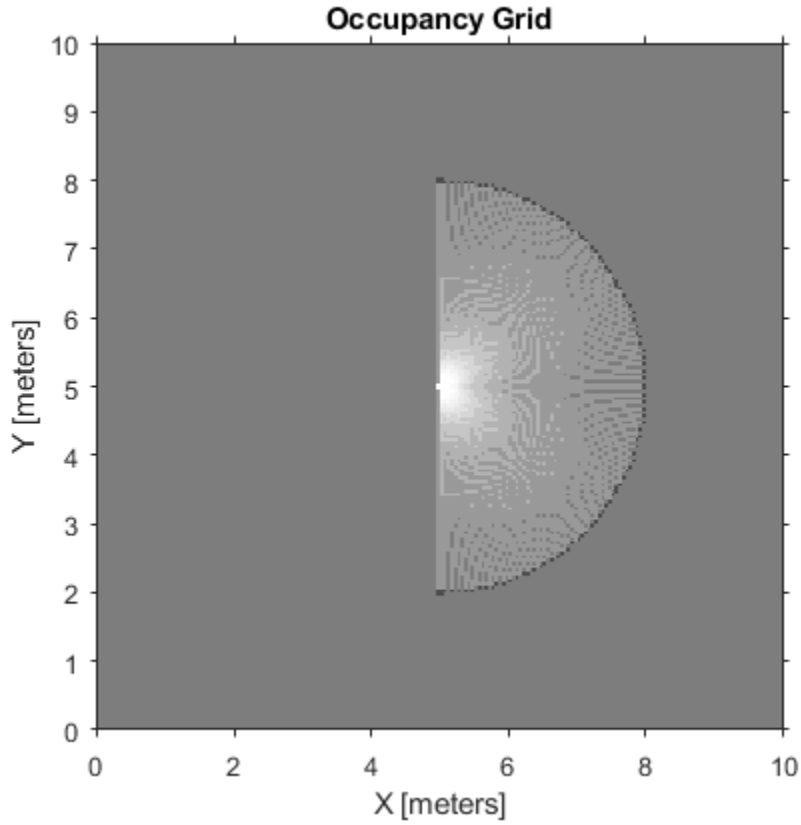
Insert a laser scan into the occupancy grid. Specify the pose of the robot ranges and angles and the max range of the laser scan.

```
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])
```

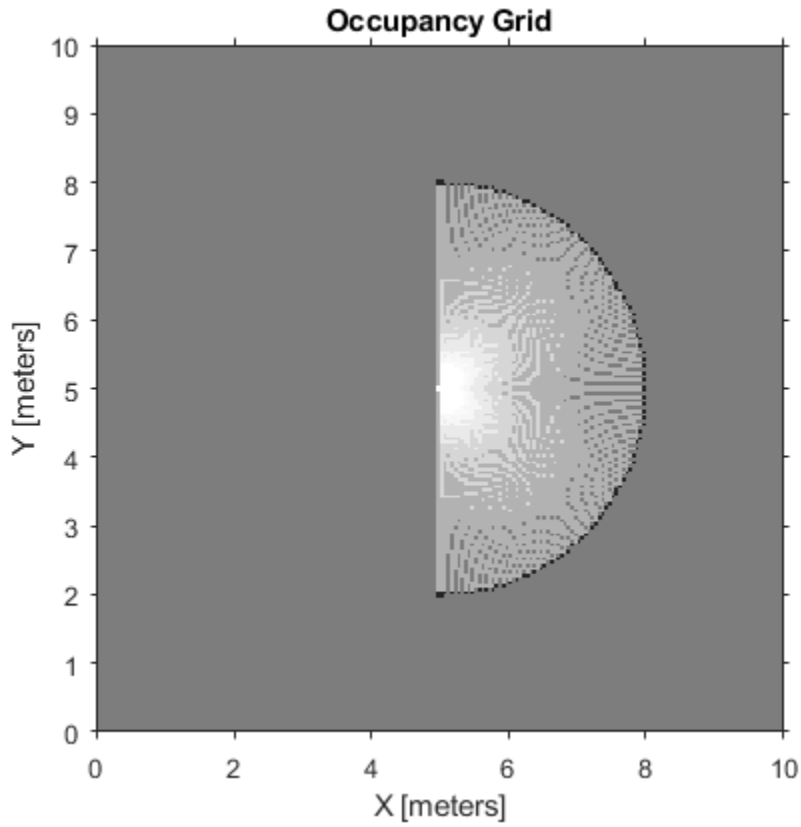
```
ans =

    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
```

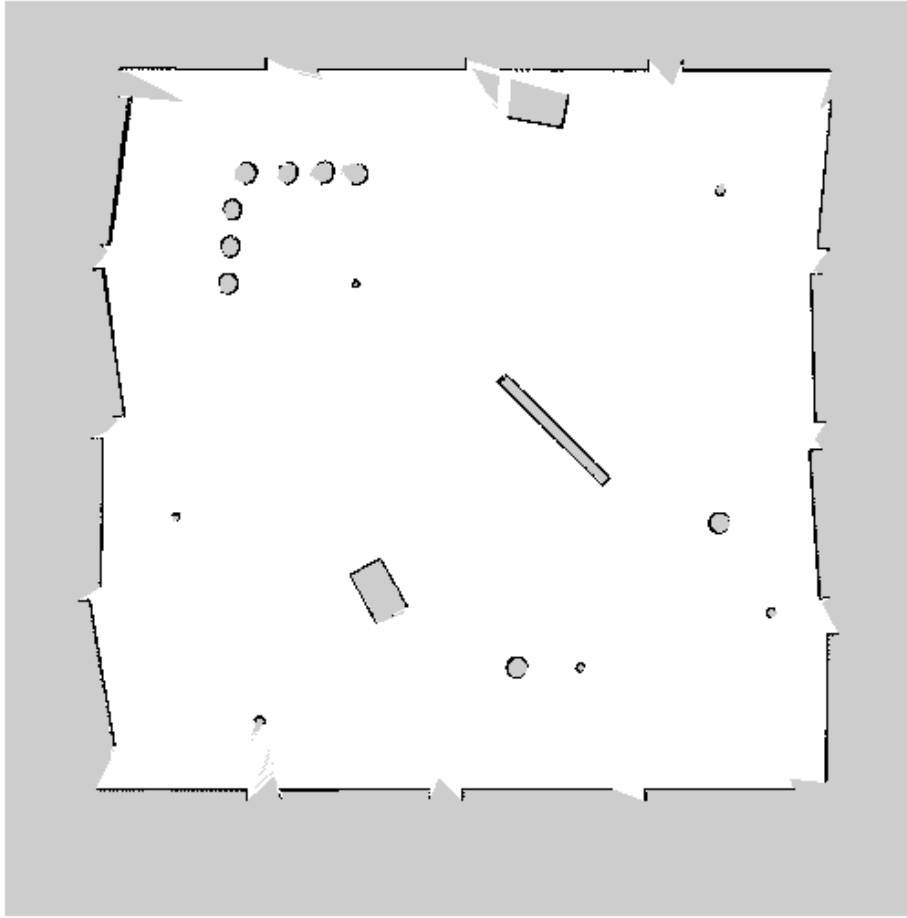


Convert PGM Image to Map

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(fullfile(matlabroot,'examples','robotics','playpen_map.pgm'));
imageCropped = image(750:1250,750:1250);
imshow(imageCropped)
```

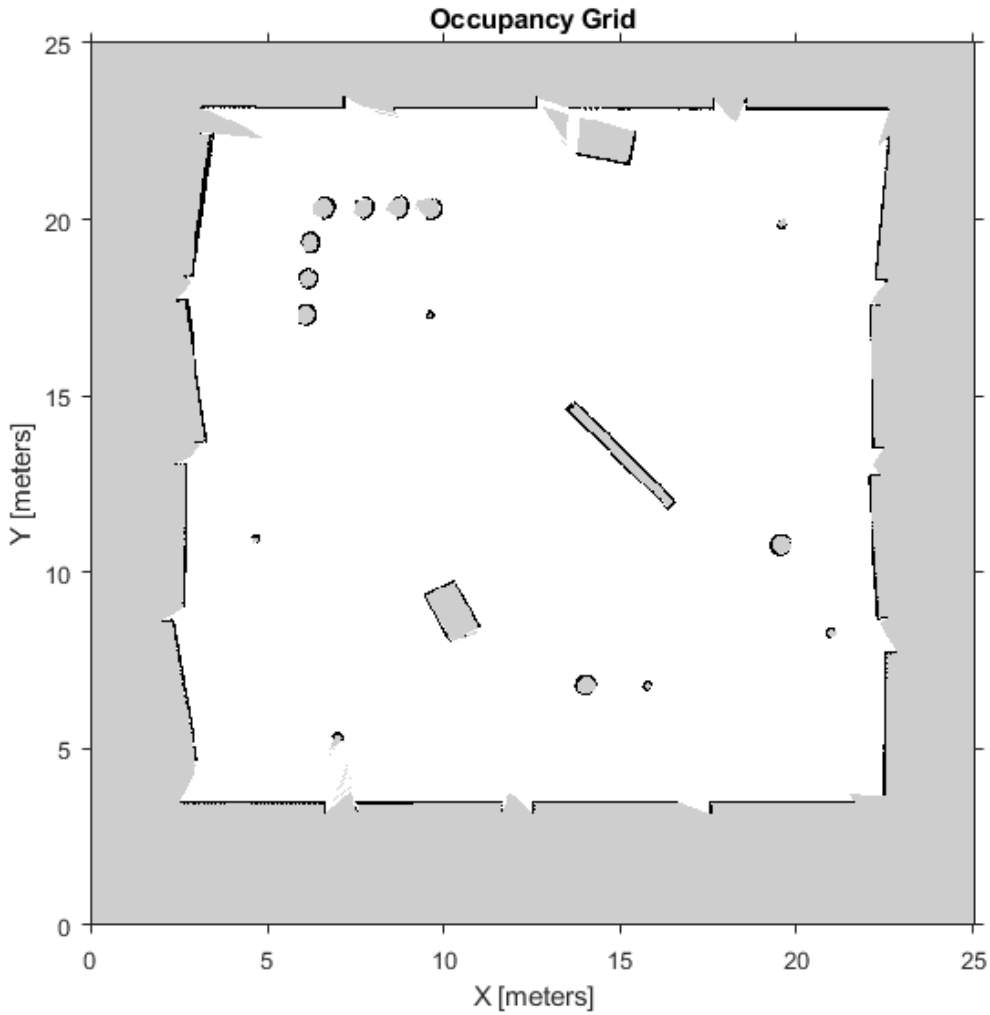


PGM values are expressed from 0 to 255 as `uint8`. Normalize these values by converting the cropped image to `double` and dividing each cell by 255. This image shows obstacles as values close to 0. Subtract the normalized image from 1 to get occupancy values with 1 representing occupied space.

```
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)
```



Å òMapping With Known Posesó

Limitations

Occupancy values have a limited resolution of Ñ0.001. 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 òOccupancy Gridsó.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLABÈ Coder Ê.

See Also

`robotics.BinaryOccupancyGrid` | `robotics.PRM` | `robotics.PurePursuit`

Topics

òMapping With Known Posesó
òOccupancy Gridsó
Class Attributes (MATLAB)
Property Attributes (MATLAB)

Introduced in R2016b

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:

Å 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) | character vector

This property is read-only.

Type of the odometry motion model, returned as a character vector. 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 mdoel 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

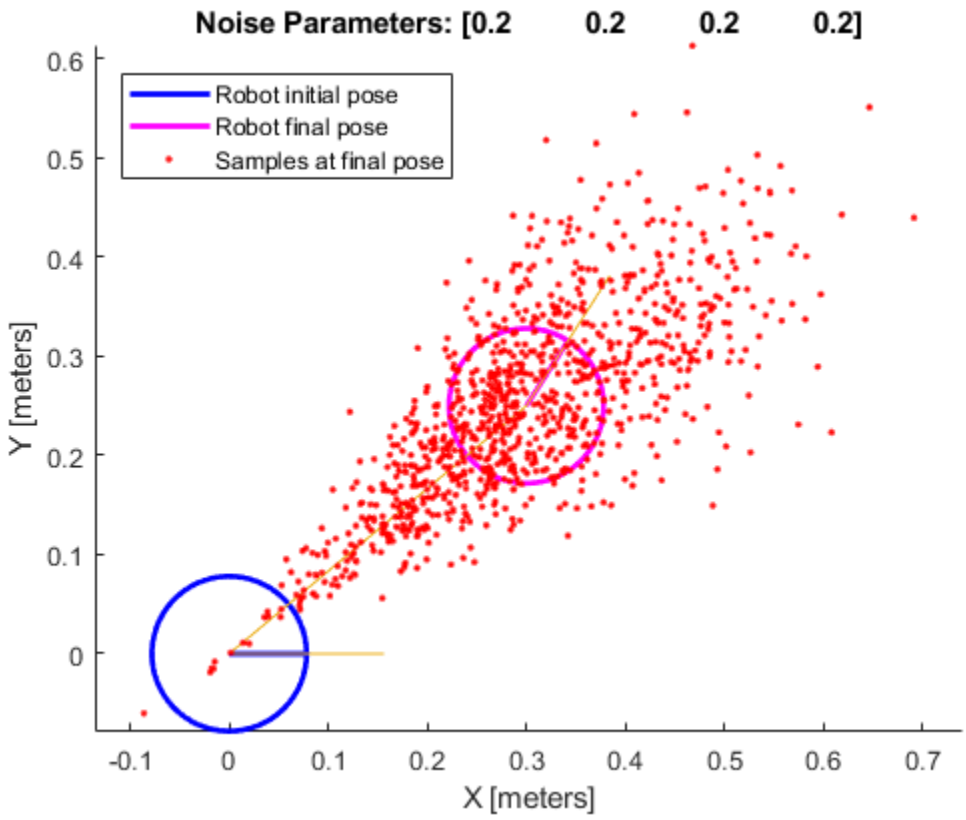
This example shows how to visualize the effect of different noise parameters on the `robotics.OdometryMotionModel` class. An `OdometryMotionModel` object contains the motion model noise parameters for a differential drive robot. Use `showNoiseDistribution` to visualize how changing these values affect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

Create a motion model object.

```
motionModel = robotics.OdometryMotionModel;
```

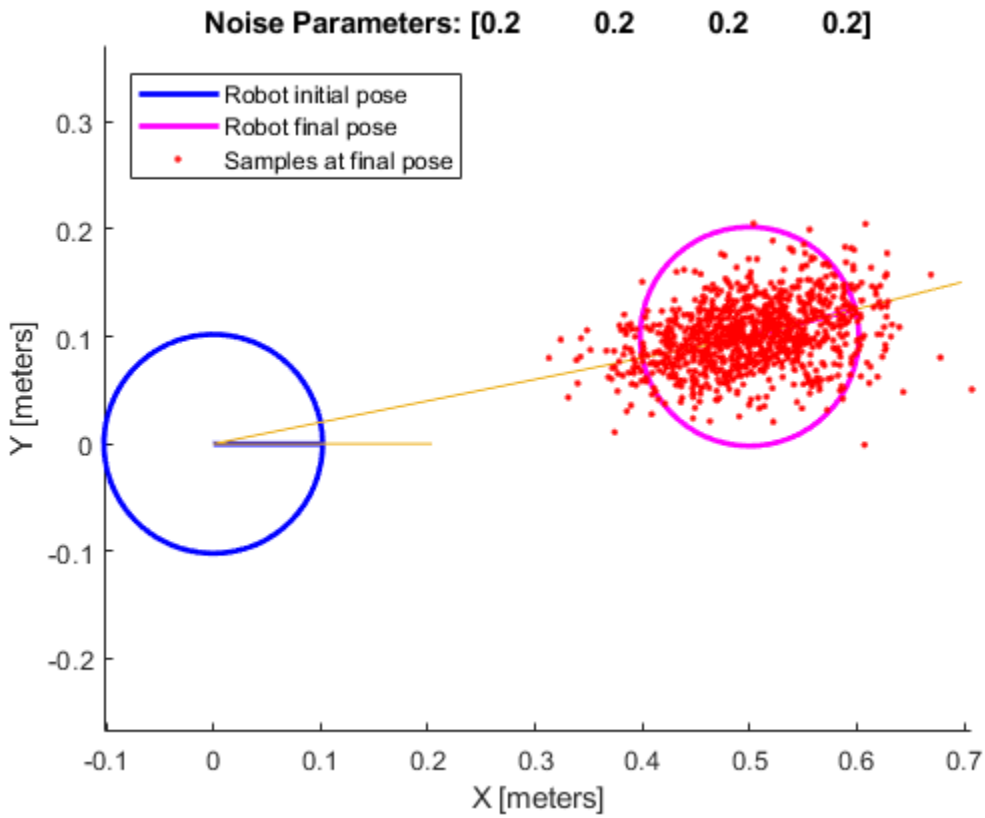
Show the distribution of particles with the existing noise parameters. Each particle is a hypothesis for the predicted pose.

```
showNoiseDistribution(motionModel);
```



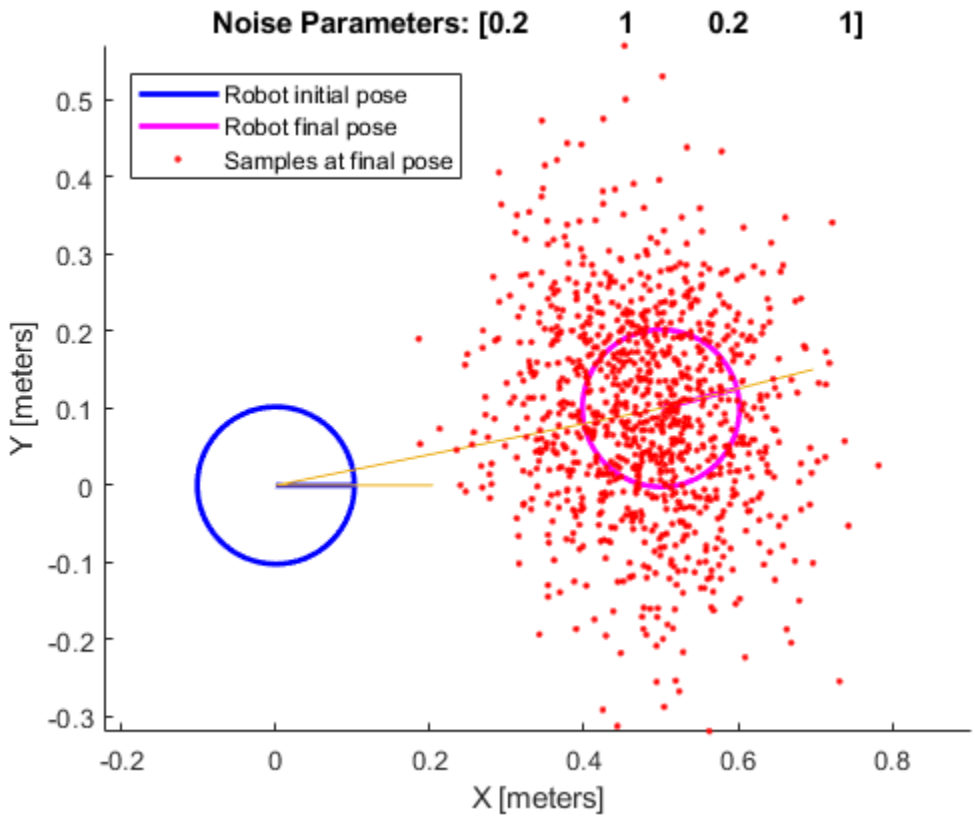
Show the distribution with a specified odometry pose change and number of samples. The change in odometry is used as the final pose with hypotheses distributed around based on the `Noise` parameters.

```
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);
```



Å òLocalize TurtleBot Using Monte Carlo Localizationó

Methods

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, Sebatian, Wolfram Burgard, and Dieter Fox. *Probabilistic Robotics*. MIT Press, 2005.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLABÈ Coder Ê.

See Also

`robotics.LikelihoodFieldSensorModel` | `robotics.MonteCarloLocalization`

Topics

òLocalize TurtleBot Using Monte Carlo Localizationó

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 òPlan a Reaching Trajectory With Multiple Kinematic Constraintsó.

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.

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

Input Arguments

`endeffector` End-effector name
character vector

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

Example: `'left_palm'`

Properties

`EndEffector` Name of the end effector
character vector

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

Example: `'left_palm'`

`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`).

`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` م 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 MATLABÈ Coder Ê.

See Also

Classes

`robotics.GeneralizedInverseKinematics` | `robotics.JointPositionBounds` | `robotics.PoseTarget` | `robotics.PositionTarget`

Topics

òPlan a Reaching Trajectory With Multiple Kinematic Constraintsó

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	`int32`
boolean	`logical`
double	`double`
string	character vector (`char`)
list	cell array (`cell`)
dictionary	structure (`struct`)

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

get	Get ROS parameter value
has	Check if ROS parameter name exists
search	Search ROS network for parameter names
set	Set value of ROS parameter or add new parameter
del	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.

```
rosinit
```

```
Initializing ROS master on http://bat5823win64:54437/.
Initializing global node /matlab_global_node_95144 with NodeURI http://bat5823win64:544
```

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_95144 with NodeURI http://bat5823win64:54
Shutting down ROS master on http://bat5823win64:54437/.
```

Å òAccess the ROS Parameter Serveró

See Also

del | get | has | rosparam | search | set

Topics

òAccess the ROS Parameter Serveró

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:

Å òParticle Filter Workflowó

Å òParticle Filter Parametersó

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 filterõs 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 `stateTranstionFcn` as:

```
stateTransitionFcn(pf,prevParticles,arg1,arg2)
```

`MeasurementLikelihoodFcn` ⌘ Callback function calculating the likelihood of sensor measurements

function handle

Callback function calculating the likelihood of sensor measurements, specified as a function handle. Once a sensor measurement is available, this callback function calculates the likelihood that the measurement is consistent with the state hypothesis of each particle. You must implement this function based on your measurement model. The function signature is:

```
function likelihood = measurementLikelihoodFcn(PF,predictParticles,measurement,varargin
```

The callback function accepts at least three input arguments:

1. `pf` ð The associated `ParticleFilter` object
2. `predictParticles` ð The particles that represent the predicted system state at the current time step as an array of size `NumParticles`-by-`NumStateVariables`
3. `measurement` ð 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.

Methods

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 <a href="matlab:helpPopup robotics.ParticleFilte
              StateCovariance: 'Use the <a href="matlab:helpPopup robotics.ParticleFilte
```

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 =

    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);
```

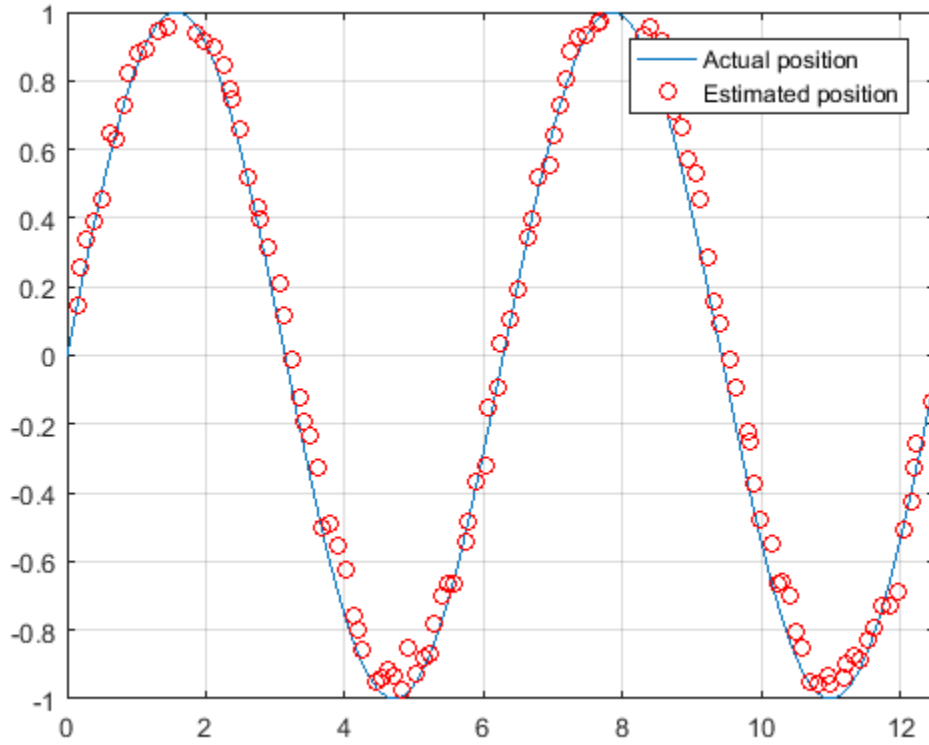
```
robotCorrected = zeros(length(t),2);
noise = 0.1;
```

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.

Å òTrack a Car-Like Robot Using Particle Filteró

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 MATLABÈ Coder Ê.

See Also

`robotics.ParticleFilter.correct` | `robotics.ParticleFilter.predict` | `robotics.ResamplingPolicy`

Topics

òTrack a Car-Like Robot Using Particle Filteró
òParticle Filter Parametersó
òParticle Filter Workflowó
Class Attributes (MATLAB)
Property Attributes (MATLAB)

Introduced in R2016a

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 òPlan a Reaching Trajectory With Multiple Kinematic Constraintsó.

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.

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

Input Arguments

`endeffector` — **End-effector name**
character vector

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

Example: `'left_palm'`

Properties

`EndEffector` ⋯ Name of the end effector

character vector

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

Example: `'left_palm'`

`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`).

`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` ℳ **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` ℳ **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 MATLABÈ Coder Ê.

See Also

Classes

`robotics.CartesianBounds` | `robotics.GeneralizedInverseKinematics` | `robotics.OrientationTarget` | `robotics.PositionTarget`

Topics

òPlan a Reaching Trajectory With Multiple Kinematic Constraintsó

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 òPlan a Reaching Trajectory With Multiple Kinematic Constraintsó.

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.

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

Input Arguments

`endeffector` م End-effector name

character vector

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

Example: `'left_palm'`

Properties

`EndEffector` Name of the end effector

character vector

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

Example: `'left_palm'`

`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 MATLABÈ Coder Ê.

See Also

Classes

`robotics.CartesianBounds` | `robotics.GeneralizedInverseKinematics` | `robotics.OrientationTarget` | `robotics.PoseTarget`

Topics

òPlan a Reaching Trajectory With Multiple Kinematic Constraintsó

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

Map representation, specified as the comma-separated pair consisting of 'Map' and 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'` ⌘ 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

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLABÈ Coder Ê.

Usage notes and limitations:

The `map` input must be specified on creation of the `PRM` object.

See Also

`robotics.BinaryOccupancyGrid` | `robotics.PurePursuit`

Topics

òPath Planning in Environments of Different Complexityó
òProbabilistic Roadmaps (PRM)ó

Introduced in R2015a

robotics.PurePursuit System object

Package: robotics

Create controller to follow set of waypoints

Description

The `robotics.PurePursuit` 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 robotôs 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 òPure Pursuit Controlleró.

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` مم **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` مم **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` مم **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 robotîs pose is an *x* and *y* position with angular orientation ê (in radians) measured from the *x*-axis.

Output Arguments

`vel` مم **Linear velocity**
scalar in meters per second

Linear velocity, specified as a scalar in meters per second.

Data Types: `double`

`angvel` مم **Angular velocity**
scalar in radians per second

Angular velocity, specified as a scalar in radians per second.

Data Types: `double`

`lookaheadpoint` مم **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
clone	Create duplicate System object
getNumInputs	Number of inputs required to call the System object
getNumOutputs	Number of outputs from calling the System object
isLocked	Determine if System object is locked
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]
```

Å òPath Following for a Differential Drive Robotó

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLABÈ Coder Ê.

Usage notes and limitations:

Use in a MATLAB Function block is not supported.

For additional information about code generation for System objects, see òSystem Objects in MATLAB Code Generationó (MATLAB Coder)

See Also

`robotics.BinaryOccupancyGrid` | `robotics.OccupancyGrid` | `robotics.PRM`

Topics

òPath Following for a Differential Drive Robotó
òPure Pursuit Controlleró

Introduced in R2015a

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

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`.

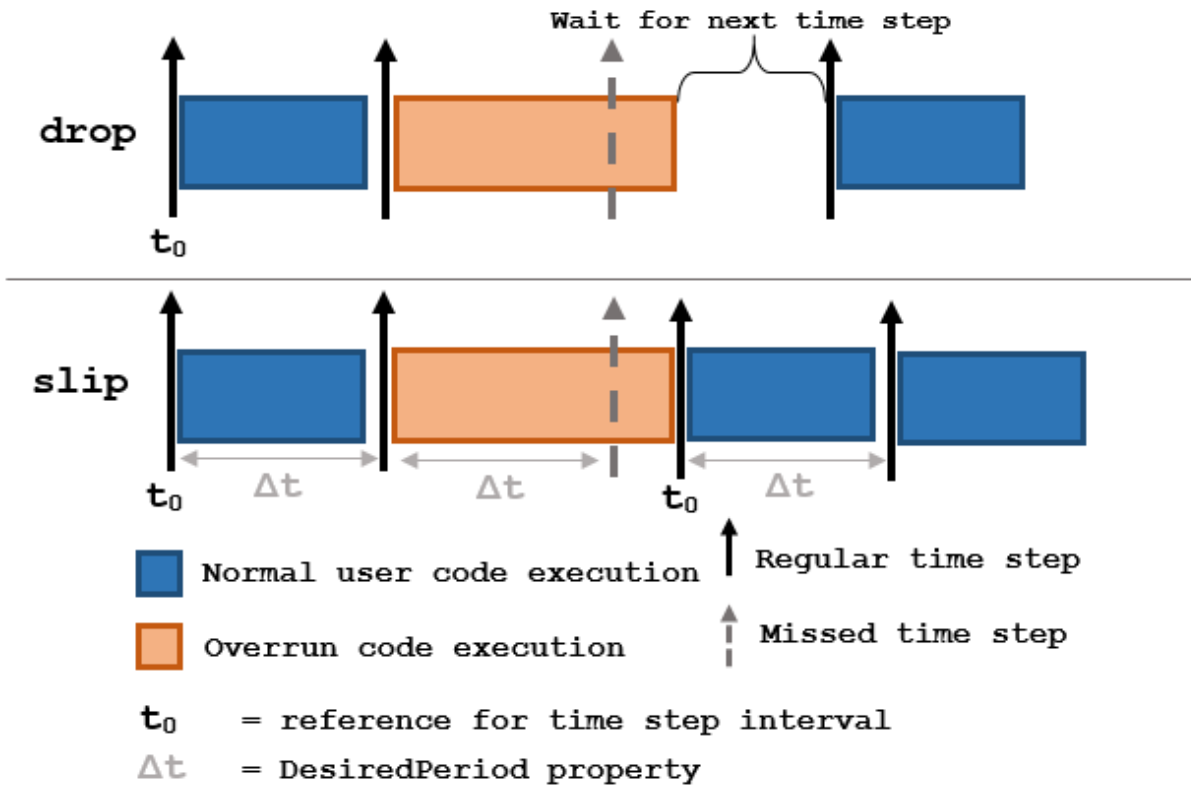
`OverrunAction` Method for handling overruns

`'slip'` (default) | `'drop'`

Method for handling overruns, specified as one of these character vectors:

Å `'drop'` ñ waits until the next time interval equal to a multiple of `DesiredPeriod`

Å `'slip'` ñ immediately executes the loop again



Each code section calls `waitfor` at the end of execution.

Object Functions

waitfor	Pause code execution to achieve desired execution rate
statistics	Statistics of past execution periods
reset	Reset Rate object

Examples

Run Loop at Fixed Rate

Create a rate object that runs at 1 Hz.

```
r = 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: 0.006148
Iteration: 2 - Time Elapsed: 1.007045
Iteration: 3 - Time Elapsed: 2.004028
Iteration: 4 - Time Elapsed: 3.003421
Iteration: 5 - Time Elapsed: 4.005034
Iteration: 6 - Time Elapsed: 5.012938
Iteration: 7 - Time Elapsed: 6.000512
Iteration: 8 - Time Elapsed: 7.000391
Iteration: 9 - Time Elapsed: 8.000306
Iteration: 10 - Time Elapsed: 9.001195
```

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: 0.0050
          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: 15.0247
          LastPeriod: 0.4998
```

Reset the object to restart the time statistics.

```
reset(r);
disp(r)
```

```
  Rate with properties:
```

```
        DesiredRate: 2
      DesiredPeriod: 0.5000
      OverrunAction: 'slip'
  TotalElapsedTime: 0.0013
        LastPeriod: NaN
```

Å òExecute Code at a Fixed-Rateó

See Also

rosrate | waitfor

Topics

òExecute Code at a Fixed-Rateó

Introduced in R2016a

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` مم 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` مم 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

òTrack a Car-Like Robot Using Particle Filteró
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 مم Name of rigid body

character vector

Name of the rigid body, specified as a 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

character vector

Name of the rigid body, specified as a character vector. This name must be unique to the body so that it can be found in a `RigidBodyTree` object.

`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]` kgÅm^2 (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 character vectors

Visual geomteries, specified as a cell array of 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 character vectour would be `'Mesh:link_0.stl'`. Visual geometries are added to the rigid body using `robotics.RigidBody.addVisual`.

Methods

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 Puma560È 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.

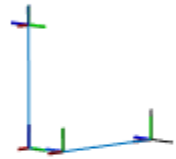
```
body2 = robotics.RigidBody('body2');
jnt2 = robotics.Joint('jnt2','revolute');
body3 = robotics.RigidBody('body3');
jnt3 = robotics.Joint('jnt3','revolute');
body4 = robotics.RigidBody('body4');
jnt4 = robotics.Joint('jnt4','revolute');
body5 = robotics.RigidBody('body5');
jnt5 = robotics.Joint('jnt5','revolute');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

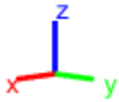
```
body5.Joint = jnt5;
body6.Joint = jnt6;

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 MATLABÈ 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)

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

```
--------------------
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)
--------------------
```





Å òBuild a Robot Step by Stepó

Å òRigid Body Tree Robot Modeló

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 MATLABÈ Coder Ê.

See Also

`robotics.Joint` | `robotics.RigidBodyTree` | `robotics.RigidBodyTree.addBody` | `robotics.RigidBodyTree.replaceJoint`

Topics

òBuild a Robot Step by Stepó
òRigid Body Tree Robot Modeló
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 specfiied 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 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) | character vector

Name of robot base, returned as a character vector.

`Gravity` ⌘ Gravitational acceleration experienced by robot

`[0 0 0]` m/s^2 (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

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 Puma560È 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.

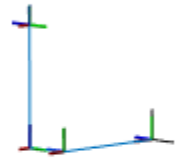
```
body2 = robotics.RigidBody('body2');
jnt2 = robotics.Joint('jnt2','revolute');
body3 = robotics.RigidBody('body3');
jnt3 = robotics.Joint('jnt3','revolute');
body4 = robotics.RigidBody('body4');
jnt4 = robotics.Joint('jnt4','revolute');
body5 = robotics.RigidBody('body5');
jnt5 = robotics.Joint('jnt5','revolute');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

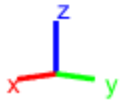
```
addBody(robot,body5,'body4')
addBody(robot,body6,'body5')
```

Verify that your robot was built properly by using the `showdetails` or `show` function. `showdetails` lists all the bodies in the MATLABÈ 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)

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

```
--------------------
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)
--------------------
```





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}
body3Copy = copy(body3);
```

```
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: {}
```

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 dyanmics 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;
```

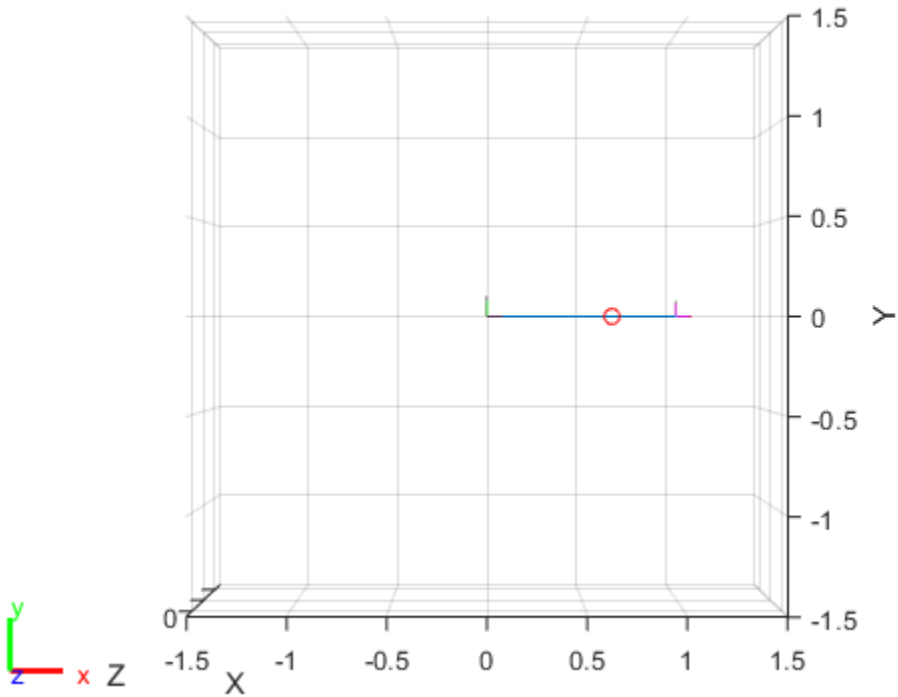
Specify dynamics properties for the two bodies. Add the bodies to the robot model. For this example, basic values for a rod (`body1`) with an attached spherical mass (`body2`) are given.

```
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');
```

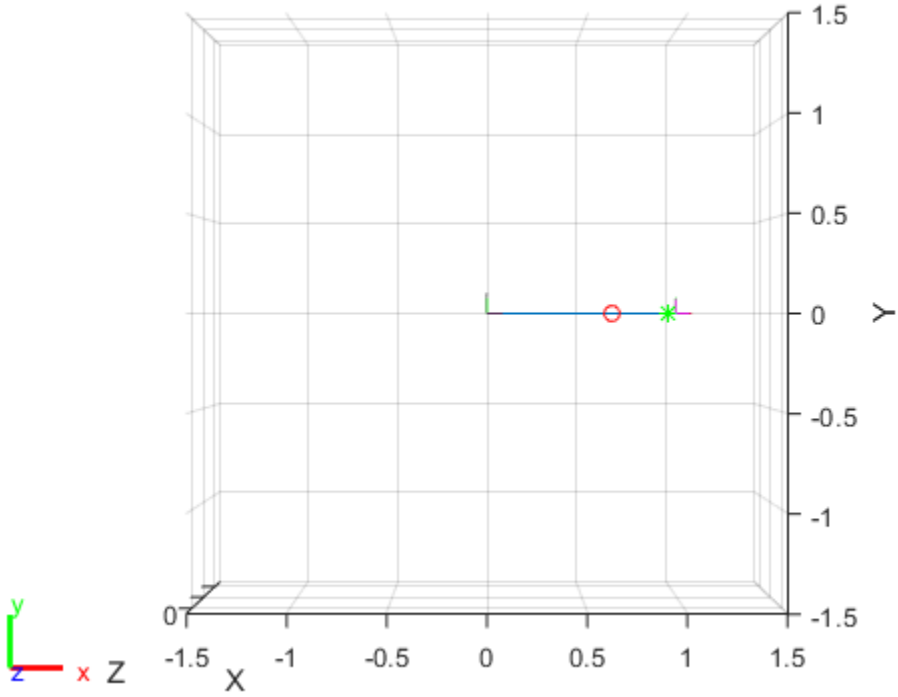
Compute the center of mass position of the whole robot. Plot the position on the robot. Move the view to the *xy* plane.

```
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 predeinfed 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 predeinfed 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` funnction.

Load a predeinfed 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 a 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 to 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.

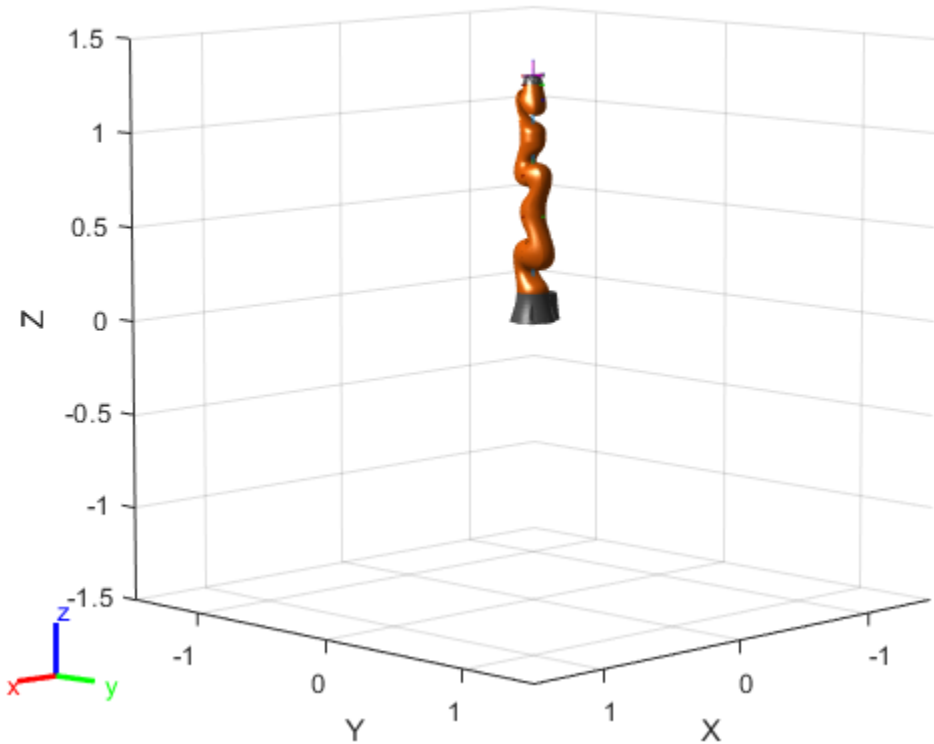
Import a robot model as a URDF file. The .stl file locations must be properly specified in this URDF. To add other .stl files to individual rigid bodies, see `robotics.RigidBody.addVisual`.

```
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)
```

```
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'

  Use GET to show all properties
```



- Å òBuild a Robot Step by Stepó
- Å òRigid Body Tree Robot Modeló
- Å òSolve Inverse Kinematics for a Four-Bar Linkageó
- Å òControl PR2 Arm Movements Using ROS Actions and Inverse Kinematicsó
- Å òPlan a Reaching Trajectory With Multiple Kinematic Constraintsó
- Å òControl LBR Manipulator Motion Through Joint Torque Commandsó

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 MATLABÈ Coder Ê.

Usage notes and limitations:

Use the syntax that specifies the `'MaxNumBodies'` as a 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

òBuild a Robot Step by Stepó
òRigid Body Tree Robot Modeló

òSolve Inverse Kinematics for a Four-Bar Linkageó
òControl PR2 Arm Movements Using ROS Actions and Inverse Kinematicsó
òPlan a Reaching Trajectory With Multiple Kinematic Constraintsó
òControl LBR Manipulator Motion Through Joint Torque Commandsó

Introduced in R2016b

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 no 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

character vector

Action type for a ROS action, returned as a 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:

Å `'pending'` ñ Goal was received, but has not yet been accepted or rejected.

Å `'active'` ñ Goal was accepted and is running on the server.

Å `'succeeded'` ñ Goal executed successfully.

Å `'preempted'` ñ An action client canceled the goal before it finished executing.

Å `'aborted'` ñ The goal was aborted before it finished executing. The action server typically aborts a goal.

Å `'rejected'` ñ The goal was not accepted after being in the `'pending'` state. The action server typically triggers this status.

Å `'recalled'` ñ A client canceled the goal while it was in the `'pending'` state.

Å `'lost'` ñ 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` ممم 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:

```
rosrun 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://192.168.154.1:51
```

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: [10×1 int32]

  Use showdetails to show the contents of the message


resultState =

  1×9 char array

succeeded
```

Disconnect from the ROS network.

```
rosshutdown
```

```
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://192.168.154.1:57
```

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: [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.

```
rosshutdown
```

```
Shutting down global node /matlab_global_node_40739 with NodeURI http://192.168.154.1:5
```

See Also

cancelGoal | rosaction | rosmessage | sendGoal | waitForServer

Topics

òROS Actions Overviewó
òMove a Turtlebot Robot Using ROS Actionsó

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

Description

`pub = rospublisher(topicname)` creates a publisher for a topic specified as a character vector and sets the `TopicName`. 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. `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.

`[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 `rosmessage` 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

character vector

This property is read-only.

Name of the published topic, specified as a 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

character vector

This property is read-only.

Message type of published messages, specified as a 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
rosmessage	Create ROS messages

Examples

Create a ROS Publisher and Send Data

Connect to a ROS network.

```
rosinit
```

```
Initializing ROS master on http://bat5823win64:54364/.
Initializing global node /matlab_global_node_55095 with NodeURI http://bat5823win64:543
```

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 = rosmessage(chatpub);
msg.Data = 'test phrase';
```

Send message via the publisher.

```
send(chatpub,msg);
```

Shutdown ROS network.

```
rosshutdown
```

```
Shutting down global node /matlab_global_node_55095 with NodeURI http://bat5823win64:54
Shutting down ROS master on http://bat5823win64:54364/.
```

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://bat5823win64:54402/.
Initializing global node /matlab_global_node_72221 with NodeURI http://bat5823win64:544
```

Create a publisher and view its properties.

```
pub = rospublisher('/chatter','std_msgs/String');

topic = pub.TopicName
subCount = pub.NumSubscribers
```

```
topic =

    '/chatter'


subCount =

     0
```

Subscriber to the publisher topic and view the changes in the `NumSubscribers` property.

```
sub = rossubscriber('/chatter');
pause(1)

subCount = pub.NumSubscribers

rosshutdown
```

```
subCount =
```

```
     1

Shutting down global node /matlab_global_node_72221 with NodeURI http://bat5823win64:54
Shutting down ROS master on http://bat5823win64:54402/.
```

Publish Data Without A ROS Publisher

Connect to a ROS network.

```
rosinit
```

```
Initializing ROS master on http://bat5823win64:55061/.
Initializing global node /matlab_global_node_96994 with NodeURI http://bat5823win64:550
```

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.

```
rosshutdown
```

```
Shutting down global node /matlab_global_node_96994 with NodeURI http://bat5823win64:55
Shutting down ROS master on http://bat5823win64:55061/.
```

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 = rosmessage('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')
```

Å òExchange Data with ROS Publishers and Subscribersó

See Also

Functions

`rosmessage` | `send`

Topics

òExchange Data with ROS Publishers and Subscribersó

Introduced in R2015a

rosrate

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 = rosrate(desiredRate)
rate = robotics.ros.Rate(node,desiredRate)
```

Description

`rate = rosrate(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`.

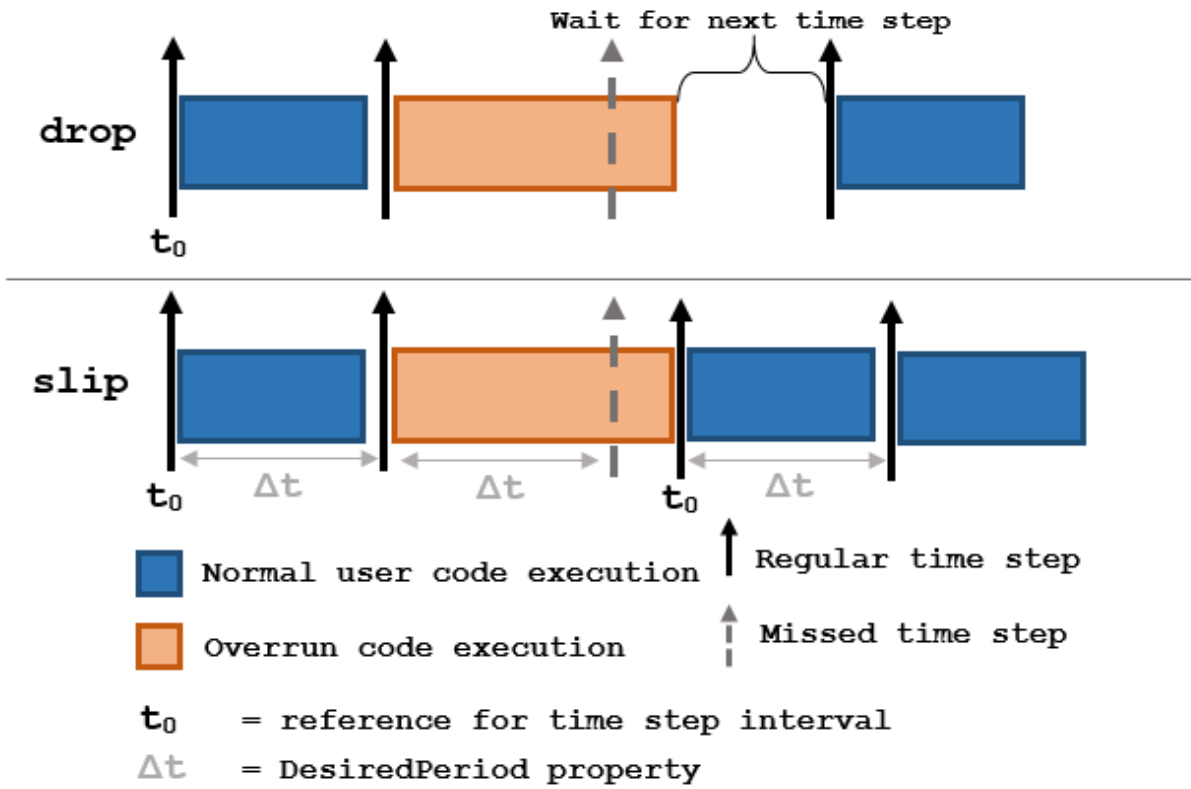
`OverrunAction` Method for handling overruns

`'slip'` (default) | `'drop'`

Method for handling overruns, specified as one of these character vectors:

Å `'drop'` ñ waits until the next time interval equal to a multiple of `DesiredPeriod`

Å `'slip'` ñ 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 `rosrate`

Initialize the ROS master and node.

```
rosinit
```

```
Initializing ROS master on http://bat5823win64:55342/.
Initializing global node /matlab_global_node_46531 with NodeURI http://bat5823win64:553
```

Create a rate object that runs at 1 Hz.

```
r = rosrate(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: 0.001501
Iteration: 2 - Time Elapsed: 1.000506
Iteration: 3 - Time Elapsed: 2.000596
Iteration: 4 - Time Elapsed: 3.000775
Iteration: 5 - Time Elapsed: 4.001104
Iteration: 6 - Time Elapsed: 5.000941
Iteration: 7 - Time Elapsed: 6.000530
Iteration: 8 - Time Elapsed: 7.000623
Iteration: 9 - Time Elapsed: 8.000273
Iteration: 10 - Time Elapsed: 9.000623
```

Shut down the ROS network.

```
rosshutdown
```

```
Shutting down global node /matlab_global_node_46531 with NodeURI http://bat5823win64:55
Shutting down ROS master on http://bat5823win64:55342/.
```

Run Loop At Fixed Rate Using ROS Time

Initialize the ROS master and node.

```
rosinit
node = robotics.ros.Node('/testTime');
```

```
Initializing ROS master on http://bat5823win64:54640/.
Initializing global node /matlab_global_node_57644 with NodeURI http://bat5823win64:546
Using Master URI http://localhost:54640 from the global node to connect to the ROS mast
```

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 iteratons. 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.

```
rosshutdown
```

```
Shutting down global node /matlab_global_node_57644 with NodeURI http://bat5823win64:54
Shutting down ROS master on http://bat5823win64:54640/.
```

Å òExecute Code at a Fixed-Rateó

See Also

`robotics.Rate` | `waitfor`

Topics

òExecute Code at a Fixed-Rateó

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 `rosinit`. 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

Description

`sub = rossubscriber(topicname)` subscribes to a topic with the given `TopicName` specified as a character vector.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 character vector 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. `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.

`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` ممم Name of the subscribed topic

character vector

This property is read-only.

Name of the subscribed topic, specified as a character vector. If the topic does not exist, the object creates the topic using its associated message type.

Example: `'/chatter'`

Data Types: `char`

`MessageType` ممم Message type of subscribed messages

character vector

This property is read-only.

Message type of subscribed messages, specified as a character vector. This message type remains associated with the topic.

Example: `'std_msgs/String'`

Data Types: `char`

`LatestMessage` ممم 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` ممم 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` ممم 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 character vector 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
exampleHelperROSCreateSampleNetwork
```

```
Initializing ROS master on http://bat5823win64:55081/.
Initializing global node /matlab_global_node_85183 with NodeURI http://bat5823win64:550
```

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 mesasge. 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_85183 with NodeURI http://bat5823win64:55
Shutting down ROS master on http://bat5823win64:55081/.
```

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://bat5823win64:54604/.
Initializing global node /matlab_global_node_36111 with NodeURI http://bat5823win64:546
```

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)
```

```
Chatter Callback message data:

ans =

    'hello world'
```

Shutdown ROS network.

```
rosshutdown
```

```
Shutting down global node /matlab_global_node_36111 with NodeURI http://bat5823win64:54
Shutting down ROS master on http://bat5823win64:54604/.
```

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 = rosmessage('std_msgs/String');
msg.Data = 'hello world';
send(pub,msg)
```

View the message data using the `LatestMesasge` 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')
```

Å òExchange Data with ROS Publishers and Subscribersó

See Also

`receive` | `rosmessage` | `rospublisher`

Topics

òExchange Data with ROS Publishers and Subscribersó

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. 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.

[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

character vector

This property is read-only.

Name of the service, specified as a character vector.

Example: '/gazebo/get_model_state'

ServiceType — Type of service

character vector

This property is read-only.

Type of service, specified as a 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_02319 with NodeURI http://AH-SRADFORD:6100
```

Set up a sevice 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_02319 with NodeURI http://AH-SRADFORD:610
Shutting down ROS master on http://AH-SRADFORD:11311/.
```

Use ROS Service Server with `ServiceServer` and `ServiceClient` Objects

Create a ROS service serve 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 MATLABÈ 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')
```

Å òCall and Provide ROS Servicesó

See Also

call | rosmessage | rosservice | rossvcserver

Topics

òCall and Provide ROS Servicesó

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

character vector

This property is read-only.

Name of the service, specified as a character vector.

Example: `'/gazebo/get_model_state'`

`ServiceType` ⌘ Type of service

character vector

This property is read-only.

Type of service, specified as a character vector.

Example: `'gazebo_msgs/GetModelState'`

`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 character vector 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

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_02319 with NodeURI http://AH-SRADFORD:6100
```

Set up a sevice 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_02319 with NodeURI http://AH-SRADFORD:610
Shutting down ROS master on http://AH-SRADFORD:11311/.
```

Å òCall and Provide ROS Servicesó

See Also

call | rosmessage | rossvcclient

Topics

òCall and Provide ROS Servicesó

Introduced in R2015a

rostf

Receive, send, and apply ROS transformations

Description

Calling `rostf` 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 `rostf` to create the transformation tree.

Creation

Syntax

```
tfTree = rostf
```

```
trtree = robotics.ros.TransformationTree(node)
```

Description

`tfTree = rostf` 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 reinitalized. You must wait the buffer time to get a fully buffered transformation tree.

Object Functions

waitForTransform	Wait until a transformation is available
getTransform	Retrieve the 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('192.168.154.131')
```

```
Initializing global node /matlab_global_node_67363 with NodeURI http://192.168.154.1:49
```

Create a transformation tree. Use the `AvailableFrames` property to see the transformation frames available. These transformations were speicfied separately prior to connecting to the network.

```
tree = rostf;
pause(1);
tree.AvailableFrames
```

```
ans =

  35×1 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'
    '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'
```

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
node = robotics.ros.Node('/testTf');
exampleHelperROSStartTfPublisher
```

```
Initializing ROS master on http://bat5823win64:55379/.
Initializing global node /matlab_global_node_70602 with NodeURI http://bat5823win64:553
Using Master URI http://localhost:55379 from the global node to connect to the ROS mast
```

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
updateTime = tftree.LastUpdateTime
```

```
frames =

  3x1 cell array

    {'camera_center' }
    {'mounting_point'}
    {'robot_base'    }


updateTime =

  ROS Time with properties:

     Sec: 1.5043e+09
    Nsec: 401000000
```

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 camer's coordinate frame.

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

Tranform 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')
rosshutdown
```

```
Shutting down global node /matlab_global_node_70602 with NodeURI http://bat5823win64:55
Shutting down ROS master on http://bat5823win64:55379/.
```

Å òAccess the tf Transformation Tree in ROSó

See Also

getTransform | sendTransform | transform | waitForTransform

Topics

òAccess the tf Transformation Tree in ROSó

Introduced in R2015a

rostime

Access ROS time functionality

Description

A ROS `Time` object representing and instance of time in seconds and nanoseconds. This time can be based off your system time, the ROS simulation time, or an arbitary time.

Creation

Syntax

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. It 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:6439
```

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 = rosmessage('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 MATLABÈ 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
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')

% Shuts down ROS network
rosshutdown
```

```
Initializing ROS master on http://bat5823win64:55025/.
Initializing global node /matlab_global_node_33332 with NodeURI http://bat5823win64:550

time =

  datetime

   01-Sep-2017 20:57:02

Shutting down global node /matlab_global_node_33332 with NodeURI http://bat5823win64:55
Shutting down ROS master on http://bat5823win64:55025/.
```

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 | rosmessage | seconds

Introduced in R2015a

robotics.VectorFieldHistogram System object

Package: robotics

Avoid obstacles using vector field histogram

Description

The `robotics.VectorFieldHistogram` System objectenables 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` مم 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` مم 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` مم 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` مم 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` مم 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
clone	Create duplicate System object
getNumInputs	Number of inputs required to call the System object
getNumOutputs	Number of outputs from calling the System object
isLocked	Determine if System object is locked
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;
```

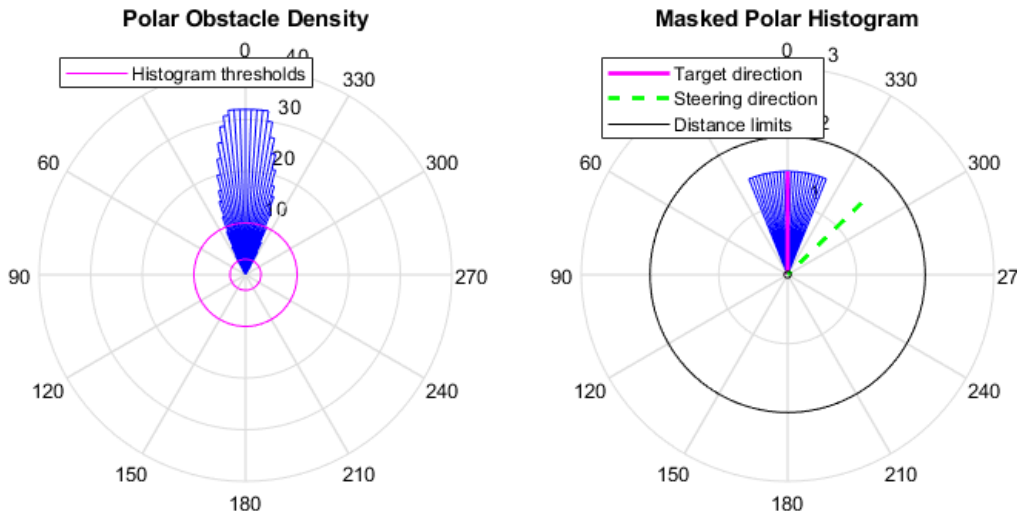
Compute an obstacle-free steering direction.

```
steeringDir = vfh(ranges,angles,targetDir)
```

```
steeringDir =

   -0.8014
```

Visualize the `VectorFieldHistogram` computation.

```
h = figure;
set(h,'Position',[50 50 800 400])
show(vfh)
```



Å òObstacle Avoidance Using TurtleBotó

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.278ð88.

[2] Ulrich, I., and J. Borenstein. "VFH : Reliable Obstacle Avoidance for Fast Mobile Robots." *Proceedings. 1998 IEEE International Conference on Robotics and Automation*. (1998): 1572ð1577.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLABÈ Coder Ê.

Usage notes and limitations:

Use in a MATLAB Function block is not supported.

For additional information about code generation for System objects, see òSystem Objects in MATLAB Code Generationó (MATLAB Coder)

See Also

`lidarScan` | `robotics.VectorFieldHistogram.reset` | `robotics.VectorFieldHistogram.show`

Topics

òObstacle Avoidance Using TurtleBotó
òVector Field Histogramó

Introduced in R2015b

Functions مم Alphabetical List

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 =

    1.5708   -0.7854   -3.1416
```

Calculate Angle Differences of Adjacent Elements

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

```
d =

   -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 MATLABÈ Coder Ê.

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 apples 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 trnasformation tree and get the transformation between desired frames.

```
rosinit('192.168.154.131')
tftree = rostf;
pause(1);
tform = getTransform(tftree,'base_link','camera_link',...
                     rostime('now'),'Timeout',5);
```

```
Initializing global node /matlab_global_node_18977 with NodeURI http://192.168.154.1:56
```

Create a ROS Point message and apply the transformation. You could also get point messages off the ROS network.

```
pt = rosmessage('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.

```
rosshutdown
```

```
Shutting down global node /matlab_global_node_18977 with NodeURI http://192.168.154.1:5
```

Input Arguments

`tfmsg` م 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` م 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 =

    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` مم **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 MATLABÈ Coder Ê.

See Also

`quat2axang`

Introduced in R2015a

axang2rotm

Convert axis-angle rotation to rotation matrix

Syntax

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

Description

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

Examples

Convert Axis-Angle Rotation to Rotation Matrix

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

```
rotm =

    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 MATLABÈ Coder Ê.

See Also

`rotm2axang`

Introduced in R2015a

axang2tform

Convert axis-angle rotation to homogeneous transformation

Syntax

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

Description

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

Examples

Convert Axis-Angle Rotation to Homogeneous Transformation

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

```
tform =

    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 مم **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 MATLABÈ Coder Ê.

See Also

`tform2axang`

Introduced in R2015a

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

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_02319 with NodeURI http://AH-SRADFORD:6100
```

Set up a sevice 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_02319 with NodeURI http://AH-SRADFORD:610
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_75695 with NodeURI http://AH-SRADFORD:6153
```

Set up a sevice server and client. This server takes 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 = rosmessage(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.

```
rosshutdown
```

```
Shutting down global node /matlab_global_node_75695 with NodeURI http://AH-SRADFORD:615
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 single 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 `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://192.168.154.1:57
```

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: [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.

```
rosshutdown
```

```
Shutting down global node /matlab_global_node_40739 with NodeURI http://192.168.154.1:5
```

Input Arguments

`client` م 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

òROS Actions Overviewó
òMove a Turtlebot Robot Using ROS Actionsó

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 `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://192.168.154.1:57
```

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: [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.

```
rosshutdown
```

```
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.

See Also

`cancelAllGoals` | `rosaction` | `sendGoal` | `sendGoalAndWait`

Topics

òROS Actions Overviewó
òMove a Turtlebot Robot Using ROS Actionsó

Introduced in R2016b

canTransform

Verify if transformation is available

Syntax

```
isAvailable = canTransform(tftree,targetframe,sourceframe)
isAvailable = canTransform(tftree,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.

`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`.

Examples

Send A Transformation to the ROS Network

This example shows how to create a transformation and send it over the ROS network.

Create a ROS transformation tree. You must be connected to a ROS network using `rosinit`. Replace `ipaddress` with your ROS network address.

```
ipaddress = '172.28.194.91';
rosinit(ipaddress)
tftree = rostf;
pause(2);
```

```
Initializing global node /matlab_global_node_61809 with NodeURI http://172.28.194.90:55
```

Verify the transformation you want does not exist. `canTransform` returns false if the transformation is not immediately available.

```
canTransform(tftree,'new_frame','base_link')
```

```
ans =

   0
```

Create a `TransformStamped` message. Populate 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;
```

```
tform =

  ROS TransformStamped message with properties:

     MessageType: 'geometry_msgs/TransformStamped'
          Header: [1×1 Header]
    ChildFrameId: ''
       Transform: [1×1 Transform]

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

Send the transformation over the ROS network.

```
sendTransform(tftree,tform)
```

Check if the transformation is now on the ROS network

```
canTransform(tftree,'new_frame','base_link')
```

```
ans =

   1
```

Shut down the ROS network.

```
rosshutdown
```

```
Shutting down global node /matlab_global_node_61809 with NodeURI http://172.28.194.90:5
```

Get ROS Transformations and Apply to ROS Messages

This example shows how to setup 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. You must be connected to a ROS network using `rosinit`. Replace `ipaddress` with your ROS network address.

```
ipaddress = '172.28.194.91';
rosinit(ipaddress)
tftree = rostf;
pause(1);
```

```
Initializing global node /matlab_global_node_93523 with NodeURI http://172.28.194.90:64
```

Look at the available frames on the transformation tree.

```
tftree.AvailableFrames
```

```
ans =

    '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'
    '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. This example is looking for the transformation from `'camera_link'` to `'base_link'`.

```
canTransform(tftree,'base_link','camera_link')
```

```
ans =

   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 be retrieved off the ROS network as well.

```
pt = rosmessage('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.

```
rosshutdown
```

```
Shutting down global node /matlab_global_node_93523 with NodeURI http://172.28.194.90:6
```

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

character vector

Target coordinate frame that entity transforms into, specified as a character vector. You can view the available frames for transformation calling `tftree.AvailableFrames`.

`sourceframe` ⌘ Initial coordinate frame

character vector

Initial coordinate frame, specified as a character vector. You can view the available frames for transformation by calling `tftree.AvailableFrames`.

`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

`isAvailable` Indicator if transform exists
boolean

Indicator if transform exists, returned as a boolean. The function returns false are if:

- `sourcetime` is outside the buffer window.
- `sourcetime` is in the future.
- the transformation has not be published yet.

See Also

`getTransform` | `transform`

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 =

    0.8147    0.1270    0.6324    1.0000
    0.9058    0.9134    0.0975    1.0000
```

Input Arguments

`cart` ⌂ Cartesian coordinates

n-by-(*k*ð1) matrix

Cartesian coordinates, specified as an *n*-by-(*k*ð1) matrix, containing *n* points. Each row of `cart` represents a point in (*k*ð1)-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 MATLABÈ Coder Ê.

See Also

`hom2cart`

Introduced in R2015a

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` | `rosmsg`

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
ptree = rosparam;
set(ptree,'MyParam','test')
has(ptree,'MyParam')
```

```
Initializing ROS master on http://bat5823win64:54494/.
Initializing global node /matlab_global_node_57635 with NodeURI http://bat5823win64:544

ans =
```

```
  logical

   1
```

Delete the parameter. Verify it was deleted. Shut down the ROS network.

```
del(ptree,'MyParam')
has(ptree,'MyParam')
rosshutdown
```

```
ans =

  logical

   0

Shutting down global node /matlab_global_node_57635 with NodeURI http://bat5823win64:54
Shutting down ROS master on http://bat5823win64:54494/.
```

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**
character vector

ROS parameter name, specified as a character vector. This character vector must match the parameter name exactly.

`namespace` — **ROS parameter namespace**
character vector

ROS parameter namespace, specified as a character vector. All parameter names starting with this character vector 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 MATLABÈ 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 MATLABÈ. 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 MATLABÈ. 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

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

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 =

    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 =

    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 character vectors:

Å `'ZYX'` (default) ð The order of rotation angles is *z*-axis, *y*-axis, *x*-axis.

Å `'ZYZ'` ð The order of rotation angles is *z*-axis, *y*-axis, *z*-axis.

Å `'XYZ'` ð The order of rotation angles is *x*-axis, *y*-axis, *z*-axis.

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 MATLABÈ Coder Ê.

See Also

`quat2eul`

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 =

    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 =

    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 character vectors:

Å `'ZYX'` (default) ð The order of rotation angles is *z*-axis, *y*-axis, *x*-axis.

Å `'ZYZ'` ð The order of rotation angles is *z*-axis, *y*-axis, *z*-axis.

Å `'XYZ'` ð The order of rotation angles is *x*-axis, *y*-axis, *z*-axis.

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 MATLABÈ Coder Ê.

See Also

`rotm2eul`

Introduced in R2015a

eul2tform

Convert Euler angles to homogeneous transformation

Syntax

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

Description

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

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

Examples

Convert Euler Angles to Homogeneous Transformation Matrix

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

```
tformZYX =

    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 =

    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` م 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` م Axis rotation sequence

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

Axis rotation sequence for the Euler angles, specified as one of these character vectors:

Å `'ZYX'` (default) ð The order of rotation angles is *z*-axis, *y*-axis, *x*-axis.

Å `'ZYZ'` ð The order of rotation angles is *z*-axis, *y*-axis, *z*-axis.

Å `'XYZ'` ð The order of rotation angles is *x*-axis, *y*-axis, *z*-axis.

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 MATLABÈ Coder Ê.

See Also

`tform2eul`

Introduced in R2015a

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.

Å 32-bit integer ñ `int32`

Å boolean ñ `logical`

Å double ñ `double`

Å string ñ character vector (`char`)

Å list ñ cell array (`cell`)

Å dictionary ñ structure (`struct`)

Examples

Set and Get Parameter Value

Create the parameter tree. A ROS network must be available using `rosinit`.

```
rosinit
ptree = rosparam;
```

```
Initializing ROS master on http://bat5823win64:54816/.
Initializing global node /matlab_global_node_40051 with NodeURI http://bat5823win64:548
```

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.

```
rosshutdown
```

```
Shutting down global node /matlab_global_node_40051 with NodeURI http://bat5823win64:54
Shutting down ROS master on http://bat5823win64:54816/.
```

Input Arguments

`ptree` — Parameter tree

`ParameterTree` object handle

Parameter tree, specified as a `ParameterTree` object handle. Create this object using the `rosparam` function.

`paramname` ممممم ROS parameter name
character vector

ROS parameter name, specified as a character vector. This character vector must match the parameter name exactly.

`namespace` ممممم ROS parameter namespace
character vector

ROS parameter namespace, specified as a character vector. All parameter names starting with this character vector are listed when calling `rosparam('list',namespace)`.

Output Arguments

`pvalue` ممممم 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.

Å 32-bit integer ñ `int32`

Å boolean ñ `logical`

Å double ñ `double`

Å string ñ character vector (`char`)

Å list ñ cell array (`cell`)

Å dictionary ñ structure (`struct`)

Limitations

Base64-encoded binary data and iso8601 data from ROS are not supported.

See Also

`rosparam` | `set`

Introduced in R2015a

getParticlesrobotics.MonteCarloLocalization.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` | `robotics.MonteCarloLocalization.step`

Topics

òMonte Carlo Localization Algorithmó

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 MATLABÈ 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 MATLABÈ. 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 MATLABÈ. 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 the transformation between two coordinate frames

Note The behavior of `getTransform` will change in a future release. The function will no longer return 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)
```

Description

`tf = 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).

`tf = getTransform(tftree,targetframe,sourceframe,sourcetime)` returns the transformation at the given source time. An error is displayed if the transformation at that time is not available.

`tf = getTransform(___,'Timeout',timeout)` specifies a timeout period, in seconds to wait for the transformation to be available. Otherwise, it returns an error. Use any of the previous syntaxes.

Examples

Get ROS Transformations and Apply to ROS Messages

This example shows how to setup 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. You must be connected to a ROS network using `rosinit`. Replace `ipaddress` with your ROS network address.

```
ipaddress = '172.28.194.91';
rosinit(ipaddress)
tftree = rostf;
pause(1);
```

```
Initializing global node /matlab_global_node_93523 with NodeURI http://172.28.194.90:64
```

Look at the available frames on the transformation tree.

```
tftree.AvailableFrames
```

```
ans =

    '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'
    '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. This example is looking for the transformation from `'camera_link'` to `'base_link'`.

```
canTransform(tftree,'base_link','camera_link')
```

```
ans =

   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 be retrieved off the ROS network as well.

```
pt = rosmessage('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.

```
rosshutdown
```

```
Shutting down global node /matlab_global_node_93523 with NodeURI http://172.28.194.90:6
```

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. You must be connected to a ROS network using `rosinit`. Replace `ipaddress` with your ROS network address.

```
ipaddress = '192.168.154.131';
rosinit(ipaddress)
tftree = rostf;
pause(2);
```

```
Initializing global node /matlab_global_node_83561 with NodeURI http://192.168.154.1:64
```

Get the transformation from 1 seconds 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.

```
rosshutdown
```

```
Shutting down global node /matlab_global_node_83561 with NodeURI http://192.168.154.1:6
```

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

character vector

Target coordinate frame, specified as a character vector. You can view the available frames for transformation by calling `tftree.AvailableFrames`.

`sourceframe` ⌘ Initial coordinate frame

character vector

Initial coordinate frame, specified as a 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, `time` is the ROS simulation time published on the `clock` topic. If the `use_sim_time` ROS parameter is set to `true`, `time` 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 (3-element vector) and a 3-D rotation (quaternion).

See Also

`transform` | `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
ptree = rosparam;
has(ptree,'MyParam')
```

```
Initializing ROS master on http://bat5823win64:55040/.
Initializing global node /matlab_global_node_62265 with NodeURI http://bat5823win64:550

ans =

  logical

   0
```

Set the `'MyParam'` parameter and verify it exists. Disconnect from ROS network.

```
set(ptree,'MyParam','test')
has(ptree,'MyParam')
rosshutdown
```

```
ans =

  logical

   1

Shutting down global node /matlab_global_node_62265 with NodeURI http://bat5823win64:55
Shutting down ROS master on http://bat5823win64:55040/.
```

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
character vector

ROS parameter name, specified as a character vector. This character vector 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 =

    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` م Cartesian coordinates
n-by-(*k*ð1) matrix

Cartesian coordinates, returned as an *n*-by-(*k*ð1) matrix, containing *n* points. Each row of `cart` represents a point in (*k*ð1)-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 MATLABÈ Coder Ê.

See Also

`cart2hom`

Introduced in R2015a

`info`robotics.PurePursuit.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:

Å `RobotPose` ð 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.

Å `LookaheadPoint`ð 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 `robotics.PurePursuit.step`.

See Also

`robotics.PurePursuit` | `robotics.PurePursuit.step`

Topics

òPure Pursuit Controlleró

Introduced in R2015a

importrobot

Import rigid body tree model from URDF file or text

Syntax

```
robot = importrobot(filename)
robot = importrobot(URDFtext)
robot = importrobot(filename,format)
robot = 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 character vector.

`robot = importrobot(filename,format)` specifies the type of robot description file for validation. The only supported format is `'urdf'`.

`robot = importrobot(___,Name,Value)` provides additional options specified by `Name,Value` pair arguments.

Examples

Import Robot from URDF File

Import the 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'
```

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 a 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 to 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.

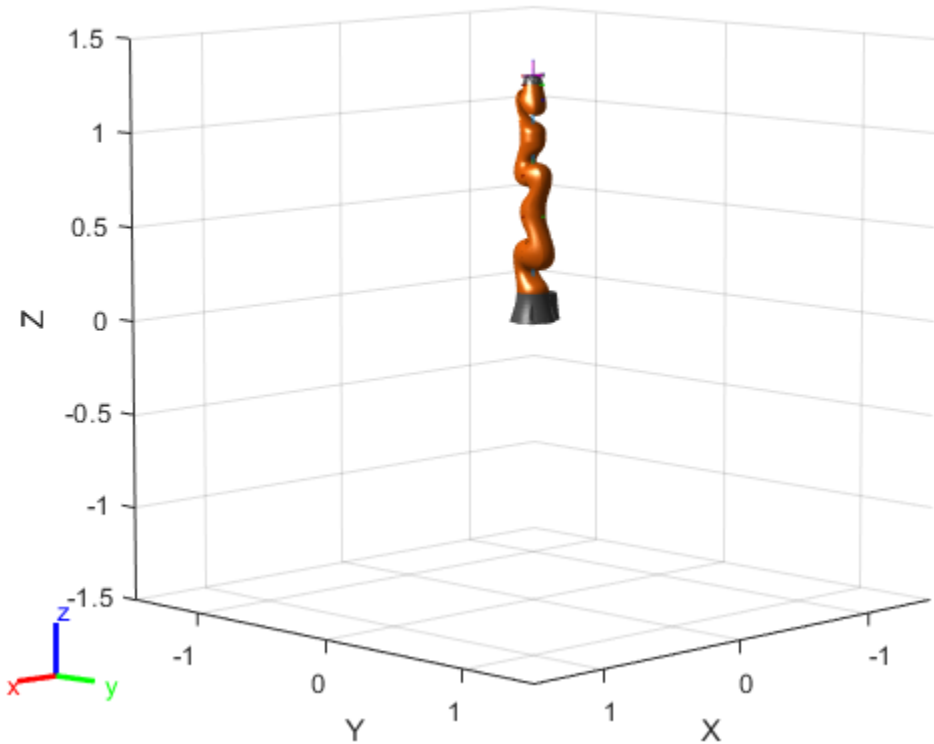
Import a robot model as a URDF file. The .stl file locations must be properly specified in this URDF. To add other .stl files to individual rigid bodies, see `robotics.RigidBody.addVisual`.

```
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)
```

```
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'

  Use GET to show all properties
```



Input Arguments

`filename` Name of URDF file
character vector

Name of URDF file, specified as a character vector. This file must be a valid URDF robot description.

Example: `'robot_file.urdf'`

`URDFtext` URDF text
character vector

URDF robot text, specified as a character vector.

Example: `<?xml version="1.0" ?><robot name="min"><link name="L0"/></robot>`

`format` ⌘ File format of robot description

`'urdf'` | character vector

File format of robot description, specified as a character vector. The only supported file format is `'urdf'`. Use this input to validate the file type.

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 single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'MeshPath',{'../arm_meshes','../body_meshes'}`

`MeshPath` ⌘ Relative search paths for mesh files

character vector | cell array of character vectors

Relative search paths for mesh files, specified as a character vector or cell array of 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.

Output Arguments

`robot` ⌘ Robot model

`RigidBodyTree` object

Robot model, returned as a `robotics.RigidBodyTree` object.

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 directory
  - MATLAB path
  - The folder containing the URDF file
  - One level above the folder containing the URDF file

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`

Topics

òRigid Body Tree Robot Modeló

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
```

Å òGenerate a Standalone ROS Node from SimulinkÈó

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

òGenerate a Standalone ROS Node from SimulinkÈó

Introduced in R2016b

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 = '192.168.154.131';
d = rosdevice(ipaddress,'user','password');
d.ROSFolder = '/opt/ros/hydro';
d.CatkinWorkspace = '~/catkin_ws_test'
```

```
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. Connect MATLABÈ to the ROS master using `rosinit`. This core enables you to run ROS nodes on your ROS device.

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

```
Another roscore / ROS master is already running on the ROS device. Use the 'stopCore' f
Initializing global node /matlab_global_node_15046 with NodeURI http://192.168.154.1:56
```

Check the available ROS nodes on the connected ROS device. These nodes were generated from SimulinkÈ models following the process in the Generate a standalone ROS node from SimulinkÈ 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_15046 with NodeURI http://192.168.154.1:5
```

Å òGenerate a Standalone ROS Node from SimulinkÈó

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

òGenerate a Standalone ROS Node from SimulinkÈó

Introduced in R2016b

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)`.

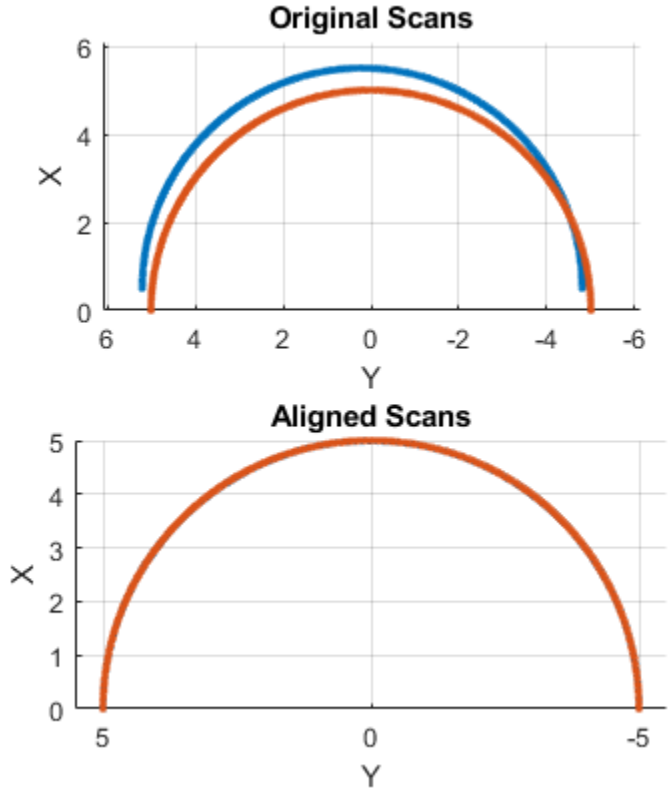
```
currScan = transformScan(refScan,[0.5 0.2 0]);
```

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 Ê 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.

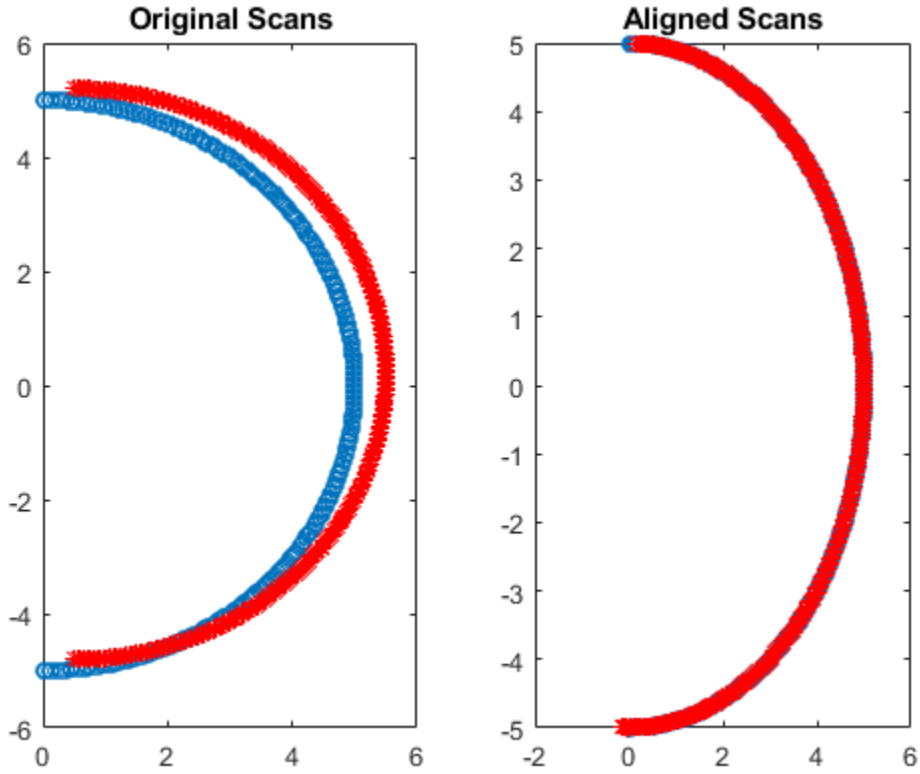
```
pose = matchScans(currRanges,currAngles,refRanges,refAngles,'SolverAlgorithm','fminunc'
```

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')
```



Å òEstimate Robot Pose with Scan Matchingó

Å òCompose a Series of Laser Scans with Pose Changesó

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` مم **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` مم **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` مم **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` مم **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` مم **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 single 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` مم Optimization algoerithm

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

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

`InitialPose` مم 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` مم 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` مم 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` مم 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` ñ 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` ñ 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. ¥rebro University, School of Science and Technology, 2009.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLABÈ Coder Ê.

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

òEstimate Robot Pose with Scan Matchingó
òCompose a Series of Laser Scans with Pose Changesó

Introduced in R2017a

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® command shell on the ROS device. When prompted, enter a user name and password.

Examples

Open Command Shell on ROS Device

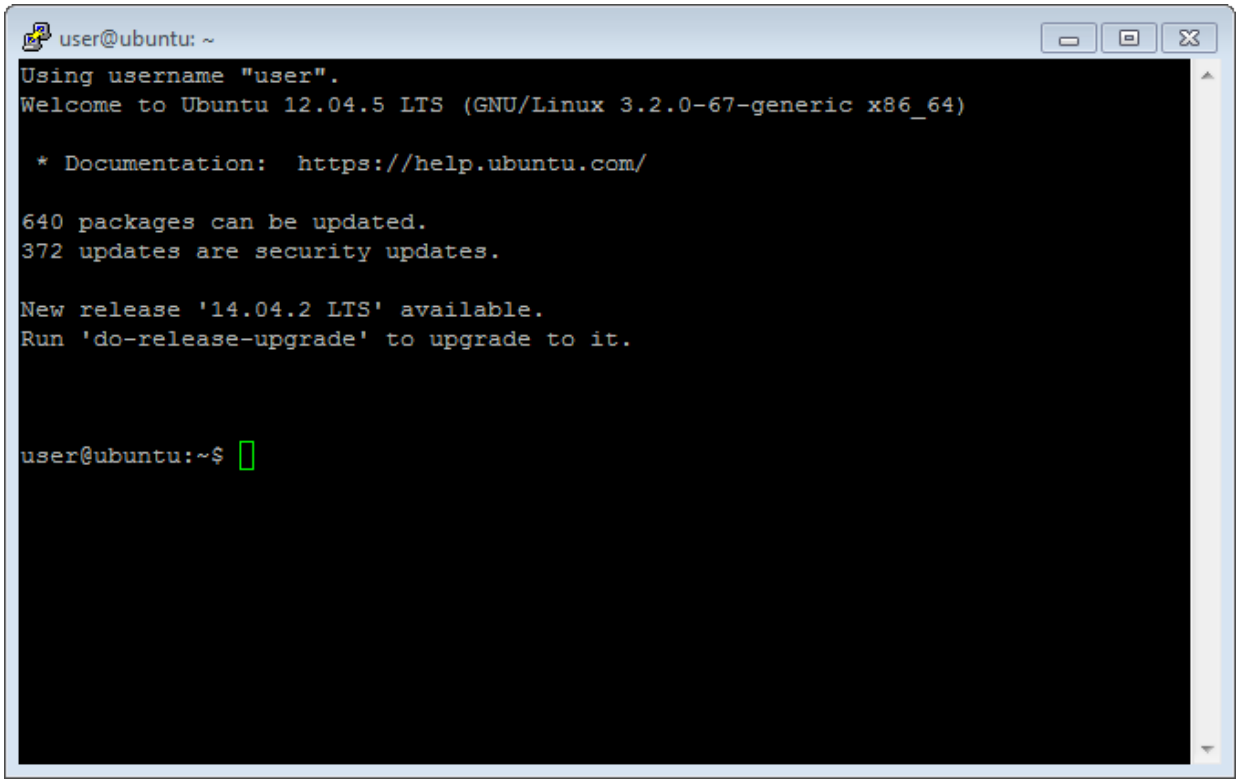
Connect to a ROS device and open the command shell on your host computer.

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);
```



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

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. Name must appear inside single quotes `('')`. You can specify several name-value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`. You can use either a laser scan

`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

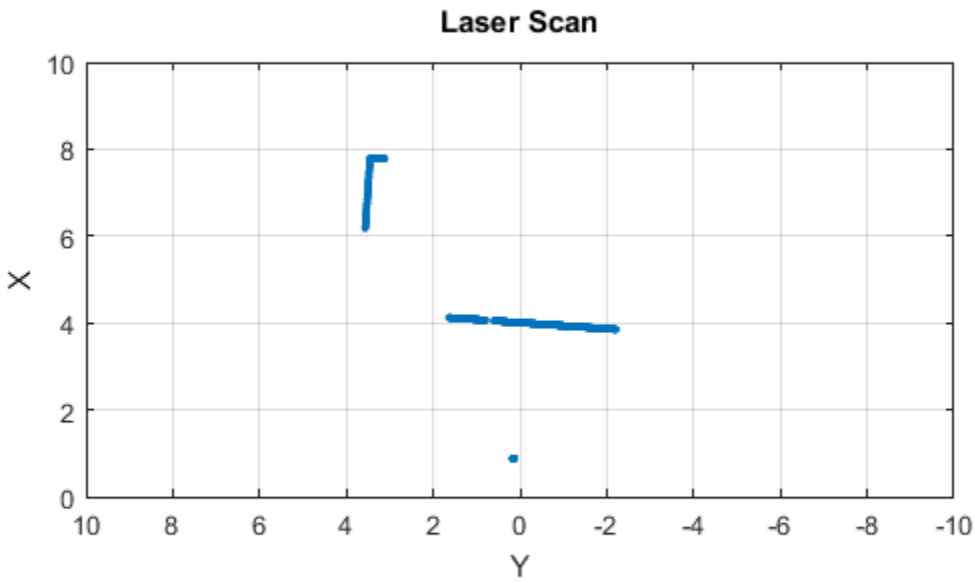
Connect to ROS network. Subscribe to a laser scan topic, and receive a message.

```
rosinit('192.168.154.131')
sub = rossubscriber('/scan');
scan = receive(sub);
```

```
Initializing global node /matlab_global_node_06485 with NodeURI http://192.168.154.1:60
```

Plot the laser scan.

```
plot(scan)
```



Shutdown ROS network.

```
rosshutdown
```

```
Shutting down global node /matlab_global_node_06485 with NodeURI http://192.168.154.1:6
```

Plot Laser Scan Message With Maximum Range

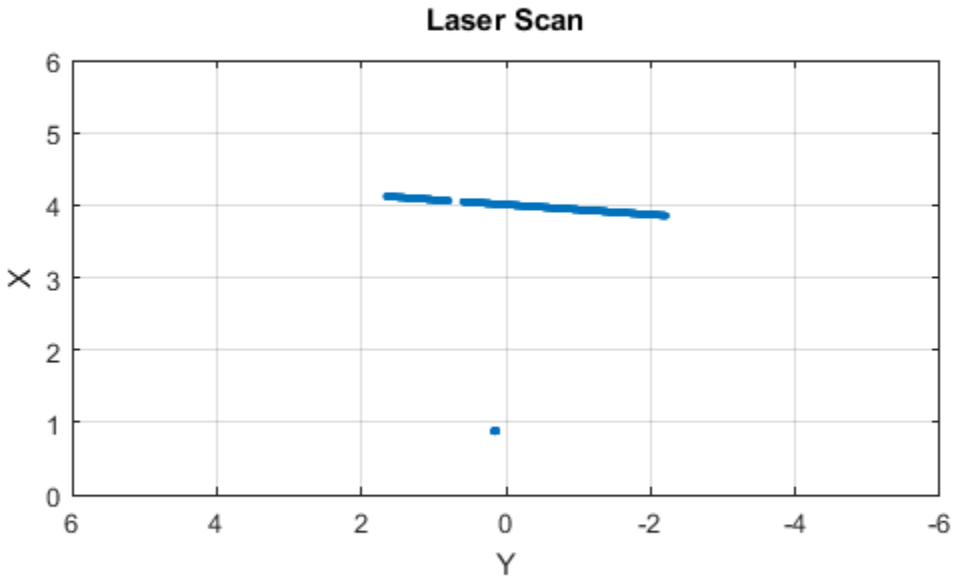
Connect to ROS network. Subscribe to a laser scan topic, and receive a message.

```
rosinit('192.168.154.131')
sub = rossubscriber('/scan');
scan = receive(sub);
```

```
Initializing global node /matlab_global_node_29862 with NodeURI http://192.168.154.1:60
```

Plot the laser scan specifying the maximum range.

```
plot(scan,'MaximumRange',6)
```



Shutdown ROS network.

```
rosshutdown
```

```
Shutting down global node /matlab_global_node_29862 with NodeURI http://192.168.154.1:6
```

Plot Lidar Scan and Remove Invalid Points

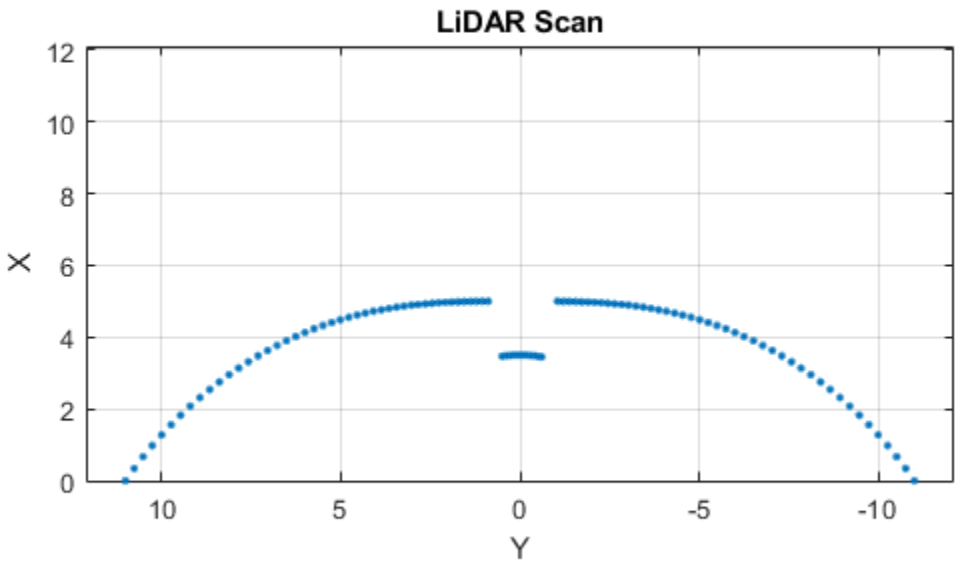
Specify lidar data as vectors of ranges and angles. These values include readings outside of the sensors range.

```
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.

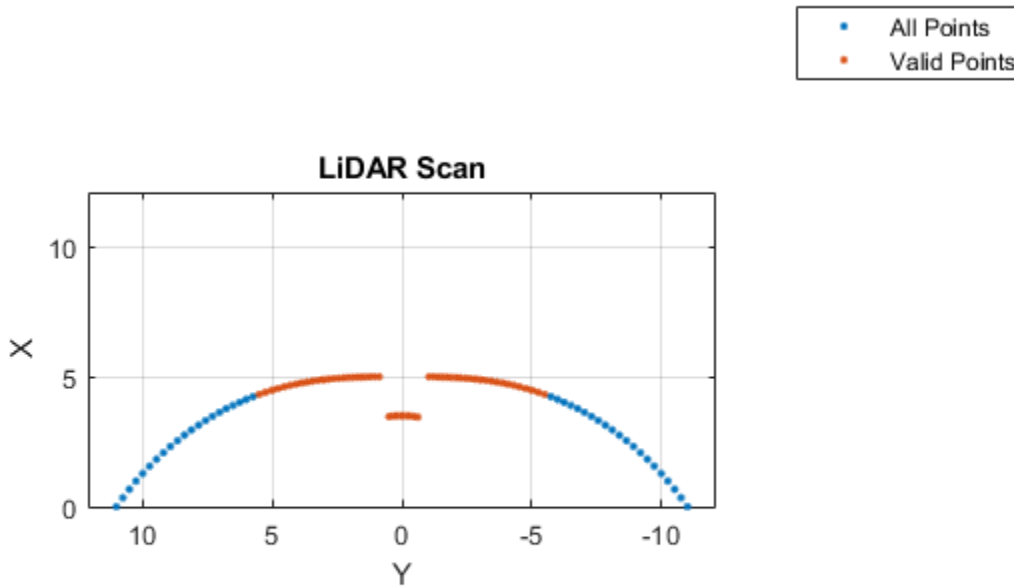
```
scan = lidarScan(ranges,angles);
plot(scan)
```



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 مم **Laser scan message**
`LaserScan` object handle

`'sensor_msgs/LaserScan'` ROS message, specified as a `LaserScan` object handle.

scanObj مم **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 single 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-sepearated pair consisting of `'MaximumRange'` and a scalar. This is only supported when you specify a When you specify this name-value pair argument, the minimum and maximum *x*-axis limits and the maximum *y*-axis limit 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

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 MATLABÈ 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 MATLABÈ. 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 MATLABÈ. 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 =

    1.0000         0         0    1.5708
```

Input Arguments

`quat` ⌘ Unit quaternion
n-by-4 matrix

Unit quaternion, specified 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]`

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 MATLABÈ Coder Ê.

See Also

`axang2quat`

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 =

         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 =

    1.5708   -1.5708   -1.5708
```

Input Arguments

`quat` ممم Unit quaternion

n-by-4 matrix

Unit quaternion, specified 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]`

`sequence` ممم Axis rotation sequence

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

Axis rotation sequence for the Euler angles, specified as one of these character vectors:

Å `'ZYX'` (default) ð The order of rotation angles is *z*-axis, *y*-axis, *x*-axis.

Å `'ZYZ'` ð The order of rotation angles is *z*-axis, *y*-axis, *z*-axis.

Å `'XYZ'` ð The order of rotation angles is *x*-axis, *y*-axis, *z*-axis.

Output Arguments

`eul` ممم 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 MATLABÈ Coder Ê.

See Also

`eul2quat`

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 =

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

Input Arguments

`quat` — **Unit quaternion**
n-by-4 matrix

Unit quaternion, specified 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]`

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 MATLABÈ Coder Ê.

See Also

`rotm2quat`

Introduced in R2015a

quat2tform

Convert quaternion to homogeneous transformation

Syntax

```
tform = quat2tform(quat)
```

Description

`tform = quat2tform(quat)` converts a quaternion, `quat`, to a homogeneous transformation matrix, `tform`. When using the transformation matrix, premultiply it with the coordinates to be transformed (as opposed to postmultiplying).

Examples

Convert Quaternion to Homogeneous Transformation

```
quat = [0.7071 0.7071 0 0];
tform = quat2tform(quat)
```

```
tform =

    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

Unit quaternion, specified 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]`

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 MATLABÈ Coder Ê.

See Also

`tform2quat`

Introduced in R2015a

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 مم Point cloud

PointCloud2 object handle

Point cloud, specified as a PointCloud2 object handle for a 'sensor_msgs/PointCloud2' ROS message.

Output Arguments

fieldnames مم 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

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

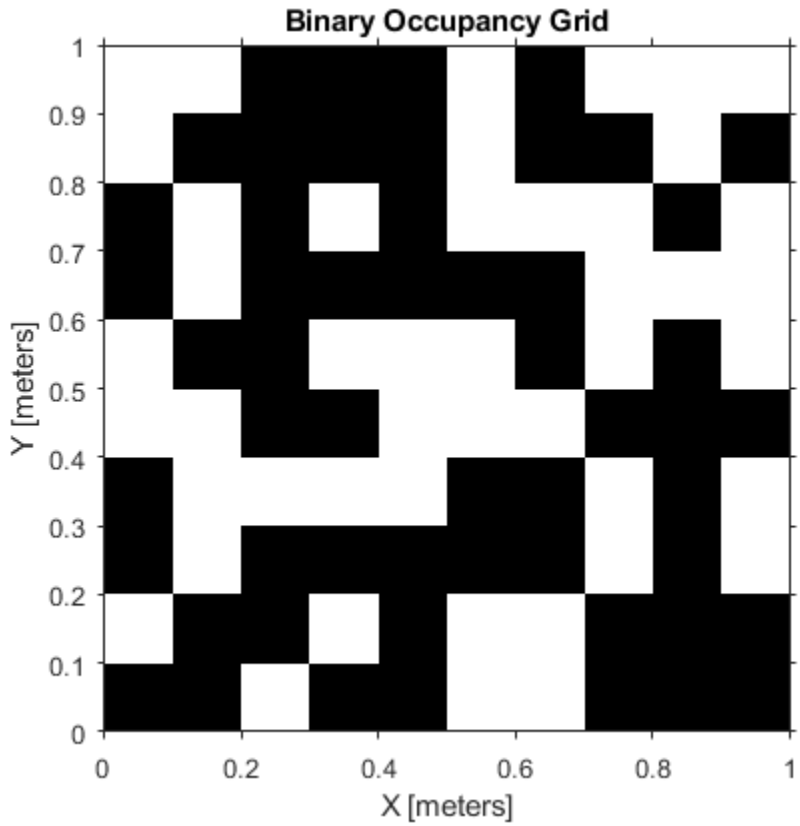
Create a occupancy grid message and populate it with data.

```
msg = rosmessage('nav_msgs/OccupancyGrid');
msg.Info.Height = 10;
msg.Info.Width = 10;
```

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

Read data from message. Show the map.

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



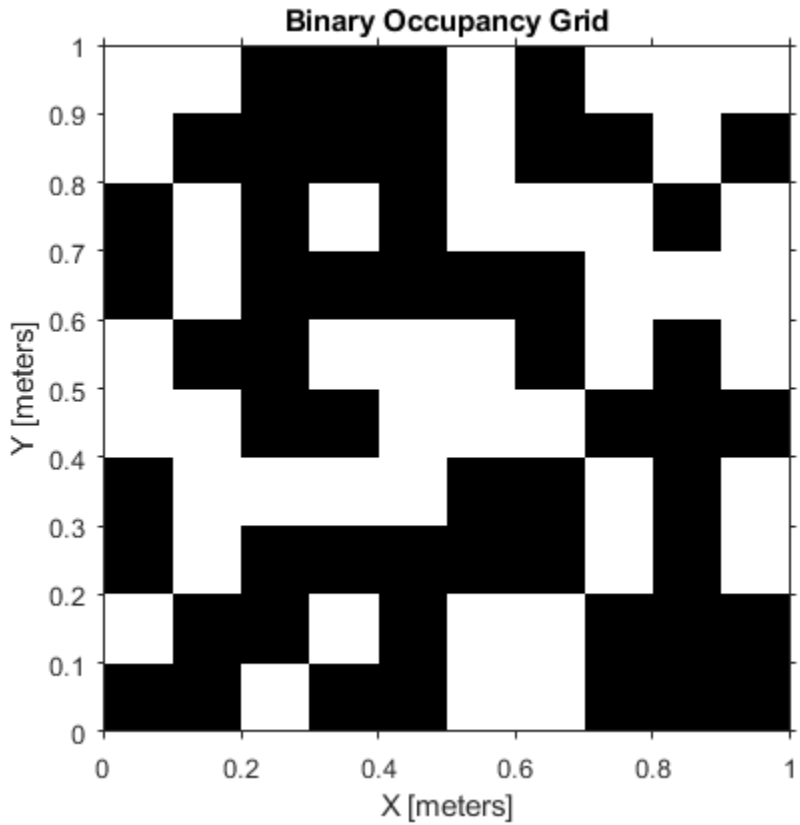
Read Binary Occupancy Data from ROS Message

Create a occupancy grid message and populate it with data.

```
msg = rosmessage('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)
```



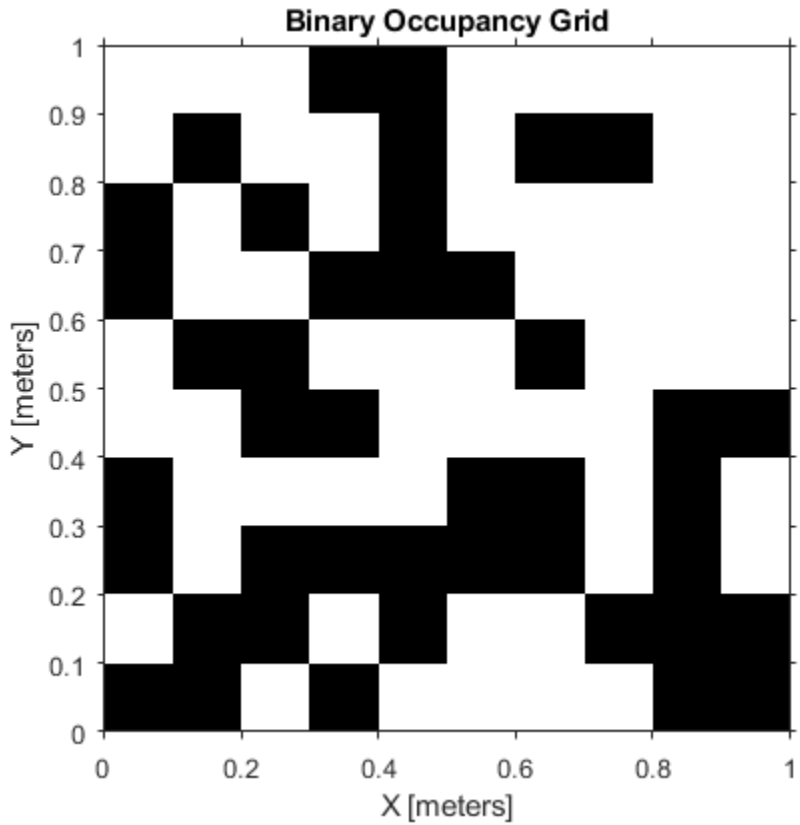
Read Binary Occupancy Data from ROS Message Using Threshold and Unknown Value Replacement

Create a occupancy grid message and populate it with data.

```
msg = rosmessage('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` مم **`'nav_msgs/OccupancyGrid'`** ROS message
`OccupancyGrid` object handle

`'nav_msgs/OccupancyGrid'` ROS message, specified as a `OccupancyGrid` object handle.

`thresh` مم 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` مم 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` مم 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

`robotics.BinaryOccupancyGrid` | `writeBinaryOccupancyGrid`

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-clockerwise around the positive *z*-axis, with the zero angle along the *x*-axis. `angles` is returned in radians and wrapped to the [ð`pi`, `pi`] interval.

Examples

Get Cartesian Coordinates from Laser Scan

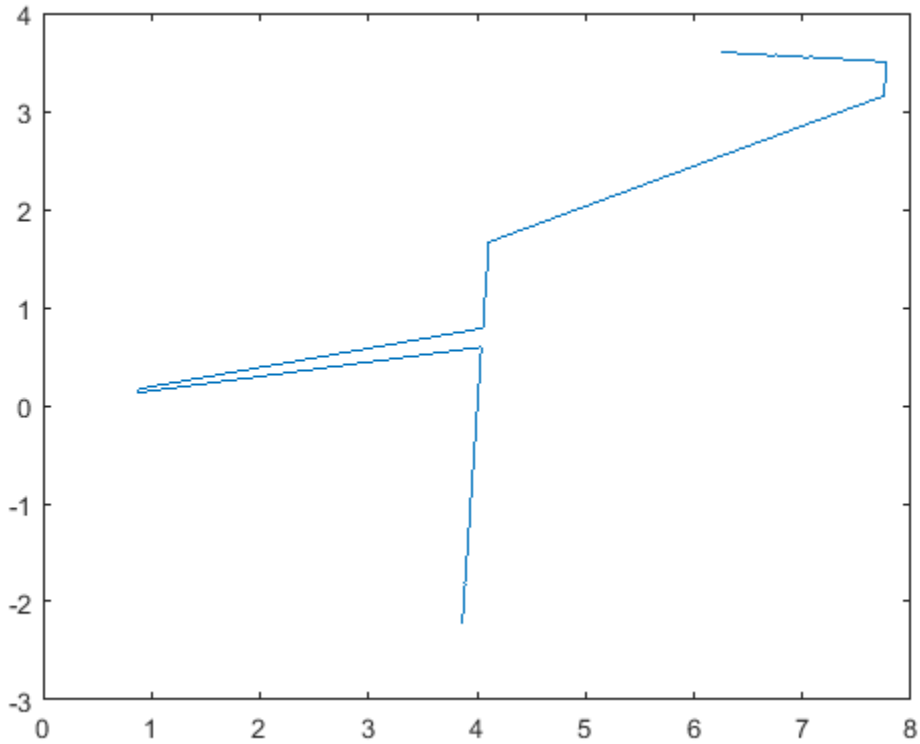
Connect to ROS network. Subscribe to a laser scan topic, and receive a message.

```
rosinit('192.168.154.131')
sub = rossubscriber('/scan');
scan = receive(sub);
```

```
Initializing global node /matlab_global_node_60179 with NodeURI http://192.168.154.1:60
```

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://192.168.154.1:6
```

Get Cartesian Coordinates from Laser Scan With Scan Range

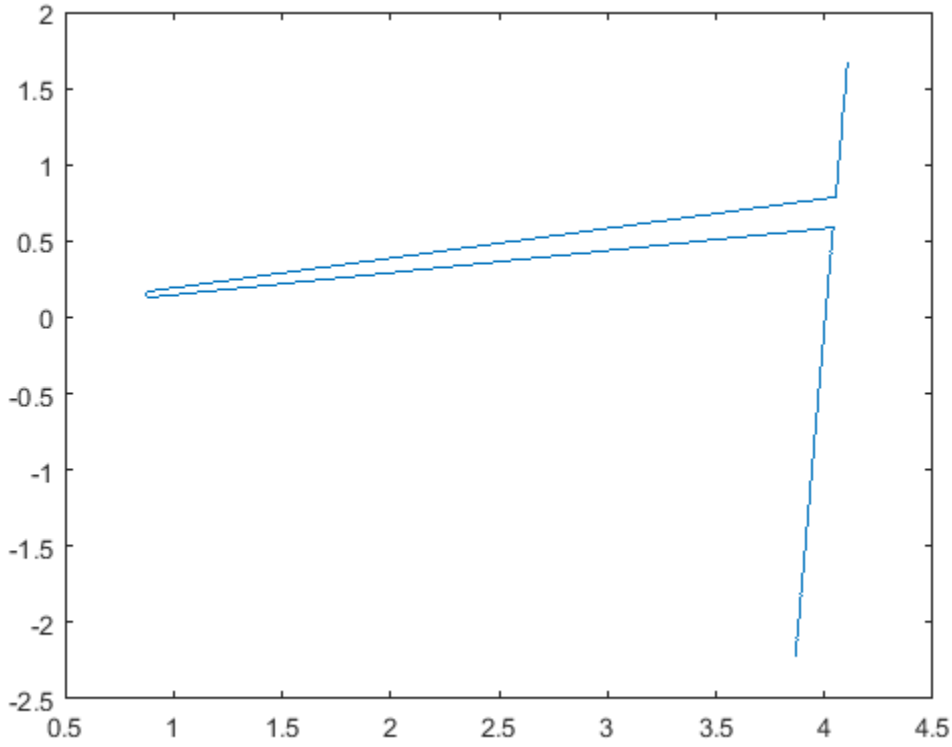
Connect to ROS network. Subscribe to a laser scan topic, and receive a message.

```
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.

```
rosshutdown
```

```
Shutting down global node /matlab_global_node_98143 with NodeURI http://192.168.154.1:6
```

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 single 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*ðbyð2 matrix in meters

Cortesian coordinates of laser scan, returned as an *n*-by-2 matrix in meters.

`angles` Scan angles for laser scan data
*n*ðbyð1 matrix in radians

Scan angles for laser scan data, returned as an *n*-by-1 matrix in radians. Angles are measured counter-clockerwise around the positive *z*-axis, with the zero angle along the *x*-axis. `angles` is returned in radians and wrapped to the [ð`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 point cloud, `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 òPreserving Point Cloud Structureó on page 2-133.

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` م 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
character vector

Field name of point cloud data, specified as a character vector. This character vector 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 òPreserving Point Cloud Structureó on page 2-133.

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

`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`.

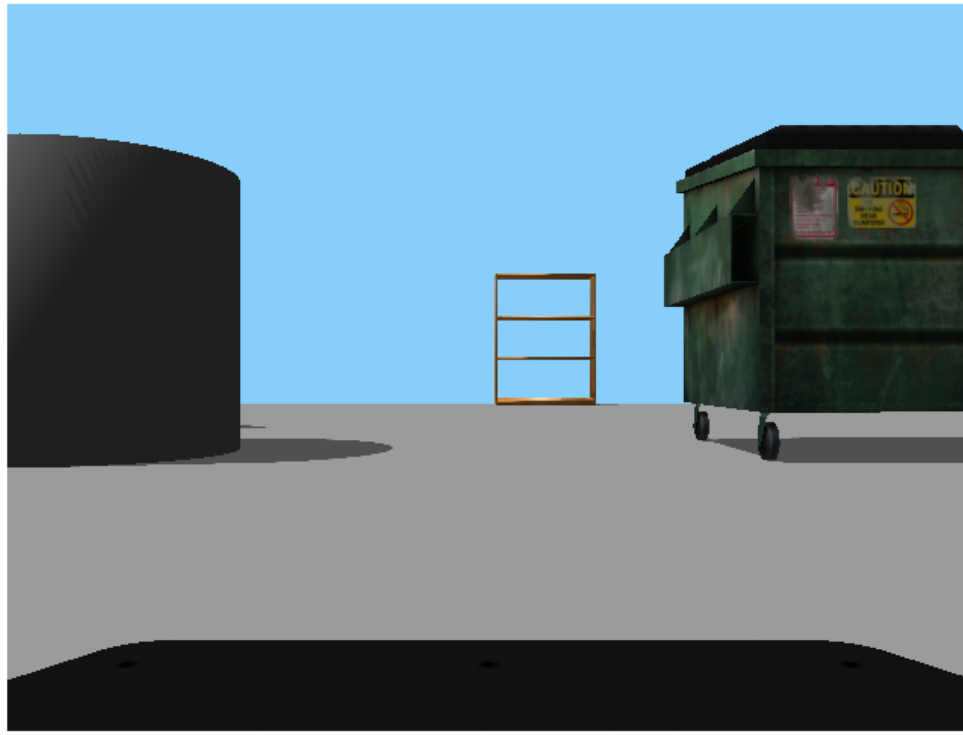
```
exampleHelperROSLoadMessages
```

Read the ROS image message as a MATLABÈ image.

```
image = readImage(img);
```

Display the image.

```
imshow(image)
```



Input Arguments

msg مم 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 | RBG image matrix | *m*-by-*n*-by-3 array

Image, returned as a matrix representing a grayscale or RGB image or as a *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*x*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*x*N*x3. 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*x*N*x3. 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*x*N*x1. Each pixel value is represented as a `uint8`.
- `mono16` images are returned as grayscale images of size *M*x*N*x1. Each pixel value is represented as a `uint16`.
- `32fcX` images are returned as floating-point images of size *M*x*N*x*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*x*N*x*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*x*N*x*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*x*N*x*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*x*N*x*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*x*N*x*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*x*N*x*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*x*N*x1, or as a converted image of size *M*x*N*x3 (Image Processing Toolbox™ is required).

The following encoding for raw images of size *M*x*N* is supported using the **`'sensor_msgs/CompressedImage'`** message type:

- `rgb8, rgba8, bgr8, bgra8`: `img` is an `rgb` image of size *M*x*N*x3. 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)
```

Description

`msgs = readMessages(bag)` returns data from all of the messages in the `BagSelection` object, `bag`. The messages are returned in a cell array of messages.

`msgs = readMessages(bag,rows)` returns data from messages in the rows specified by `rows`. The maximum range of the rows is `[1,bag.NumMessages]`.

Examples

Return ROS Messages as a Cell Array

Set file path.

```
filePath = fullfile(fileparts(which('ROSWorkingWithRosbagsExample')), 'data', 'ex_multi
```

Read rosbag and filter by topic and time.

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

Return All Messages as a Cell Array

```
allMsgs = readMessages(bagselect2);
```

Return The First 10 Messages as a Cell Array

```
firstMsgs = readMessages(bagselect2,1:10);
```

Input Arguments

bag ⌘ Message of a rosbag

`BagSelection` object

All the messages contained within a rosbag, specified as a `BagSelection` object.

rows ⌘ Rows of BagSelection object

n-by-2 matrix

Rows of `BagSelection` object, specified as an *n*-by-2 matrix, where *n* is the number of rows to retrieve messages from. The maximum range of the rows is [1, `bag.NumMessage`].

Output Arguments

msgs ⌘ ROS message object handle

handle | cell array

ROS message object handle, returned as a handle or cell array. ROS messages are retrieved from the `BagSelection` object.

See Also

`rosbag` | `select` | `timeseries`

Introduced in R2015a

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 point cloud 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 òPreserving Point Cloud Structureó on page 2-142.

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 òPreserving Point Cloud Structureó on page 2-142.

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

`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-clockerwise around the positive *z*-axis, with the zero angle along the *x*-axis. `angles` is returned in radians and wrapped to the [ðpi, 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 ᵐ **Laser scan message**
`LaserScan` object handle

`'sensor_msgs/LaserScan'` ROS message, specified as a `LaserScan` object handle.

Output Arguments

`angles` مم Scan angles for laser scan data

*n*ðbyð1 matrix in radians

Scan angles for laser scan data, returned as an *n*-by-1 matrix in radians. Angles are measured counter-clockerwise around the positive *z*-axis, with the zero angle along the *x*-axis. `angles` is returned in radians and wrapped to the [ð`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 point cloud 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 òPreserving Point Cloud Structureó on page 2-146.

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 òPreserving Point Cloud Structureó on page 2-146.

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

`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
exampleHelperROSCreateSampleNetwork
```

```
Initializing ROS master on http://bat5823win64:55081/.
Initializing global node /matlab_global_node_85183 with NodeURI http://bat5823win64:550
```

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 mesasge. 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_85183 with NodeURI http://bat5823win64:55
Shutting down ROS master on http://bat5823win64:55081/.
```

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: [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.

```
rosshutdown
```

```
Shutting down global node /matlab_global_node_10876 with NodeURI http://AH-SRADFORD:651
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');
```

Å òExchange Data with ROS Publishers and Subscribersó

Input Arguments

`sub` م ROS subscriber
`Subscriber` object handle

ROS subscriber, specified as a `Subscriber` object handle. You can create the subscriber using `rossubscriber`.

`timeout` م Timeout for receiving a message
scalar in seconds

Timeout for receiving a message, specified as a scalar in seconds.

Output Arguments

`msg` م ROS message
`Message` object handle

ROS message, returned as a `Message` object handle.

See Also

`rosmessage` | `rospublisher` | `rossubscriber` | `rostopic` | `send`

Topics

òExchange Data with ROS Publishers and Subscribersó

Introduced in R2015a

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

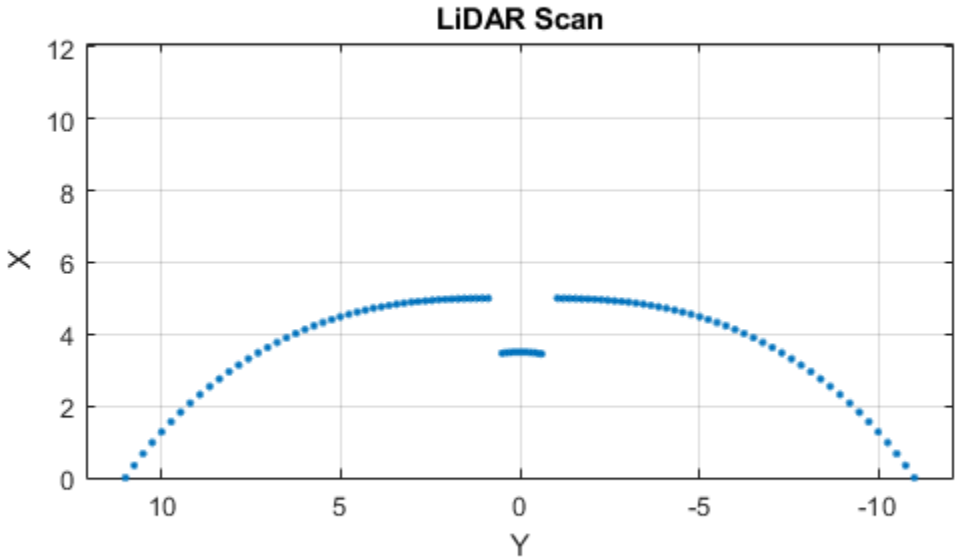
Plot Lidar Scan and Remove Invalid Points

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));
```

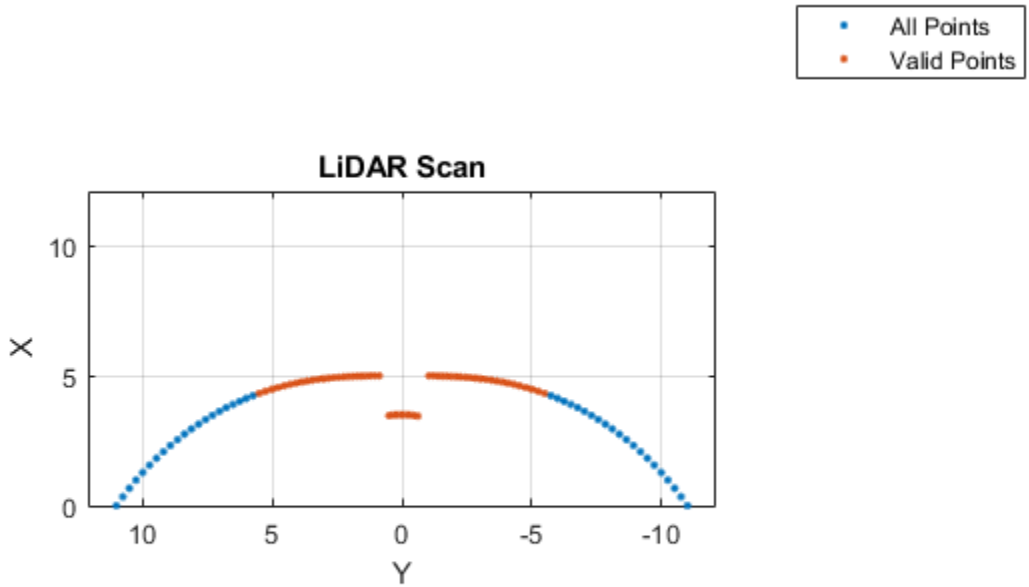
Create a lidar scan by specifying the ranges and angles. Plot all points of the lidar scan.

```
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 مم 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 single

quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `['RangeLimits',[0.05 2]`

`RangeLimits` مهم **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` مهم **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 limitslimits are removed.

Angles are measured counter-clockwise around the positivez-axis.

Data Types: `single` | `double`

Output Arguments

`validScan` مهم **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: 15.0247
          LastPeriod: 0.4998
```

Reset the object to restart the time statistics.

```
reset(r);
disp(r)
```

```
  Rate with properties:

         DesiredRate: 2
       DesiredPeriod: 0.5000
       OverrunAction: 'slip'
    TotalElapsedTime: 0.0013
          LastPeriod: NaN
```

Å òExecute Code at a Fixed-Rateó

See Also

`robotics.Rate` | `rosrate` | `waitfor`

Topics

òExecute Code at a Fixed-Rateó

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
```

- òInstall Robotics System Toolbox Add-onsó
- òROS Custom Message Supportó
- òGet Add-Onsó (MATLAB)
- òManage Your Add-Onsó (MATLAB)

See Also

Topics

òInstall Robotics System Toolbox Add-onsó
òROS Custom Message Supportó
òGet Add-Onsó (MATLAB)
òManage Your Add-Onsó (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 òInstall Robotics System Toolbox Add-onsó

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.

```
rosshutdown
```

```
Shutting down global node /matlab_global_node_87036 with NodeURI http://192.168.154.1:6
```

Input Arguments

`actionname` — ROS action name

character vector

ROS action name, specified as a 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 òROS Actions Overviewó.

`actiontype` — Type of ROS action

character vector

Type of ROS action, returned as a character vector.

See Also

`cancelGoal` | `rosmessage` | `rostopic` | `sendGoal` | `waitForServer`

Topics

òROS Actions Overviewó
òMove a Turtlebot Robot Using ROS Actionsó

Introduced in R2016b

rosactionclient

Create ROS action client

Syntax

```
client = rosactionclient(actionname)
client = rosactionclient(actionname,actiontype)
[client,goalMsg] = rosactionclient(___)
```

Description

`client = rosactionclient(actionname)` creates a client for the specified ROS action name. 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. If the action is not available, or the name and type do no 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.

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:

```
rosrun 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://192.168.154.1:51
```

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: [10×1 int32]

  Use showdetails to show the contents of the message


resultState =

  1×9 char array
```

```
succeeded
```

Disconnect from the ROS network.

```
rosshutdown
```

```
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://192.168.154.1:57
```

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: [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.

```
rosshutdown
```

```
Shutting down global node /matlab_global_node_40739 with NodeURI http://192.168.154.1:5
```

Input Arguments

`actionname` — **ROS action name**
character vector

ROS action name, specified as a character vector. The action name must match one of the topics that `rosaction('list')` outputs.

`actiontype` — **Type of ROS action**
character vector

Type of ROS action, returned as a character vector.

Output Arguments

`client` مم ROS action client
`SimpleActionClient` object handle

ROS action client, returned as a `SimpleActionClient` object handle. This simple action client enables you to track a single goal at a time.

`goalMsg` مم ROS action goal message
`Message` object handle

ROS action goal message, returned 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

`SimpleActionClient` | `cancelGoal` | `rosaction` | `rosmessage` | `sendGoal` | `waitForServer`

Topics

òROS Actions Overviewó
òMove a Turtlebot Robot Using ROS Actionsó

Introduced in R2016b

rosbag

Open and parse rosbag log file

Syntax

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

Description

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

A rosbag, 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.

This function supports version 2.0 of the rosbag file format. It also supports only uncompressed rosbags. See the ROS Wiki page for more information about rosbags and Bag version 2.0.

Examples

Retrieve information from rosbag

Set the path to a rosbag file.

```
filePath = fullfile(fileparts(which('ROSWorkingWithRosbagsExample')), 'data', 'ex_multi
```

Retrieve information from the rosbag

```
bagselect = rosbag(filePath);
```

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

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

Input Arguments

`filename` ⌘ Name of rosbag file and its path

character vector

Name of file and its path, for the rosbag you want to access, specified as a character vector. This path can be relative or absolute.

Output Arguments

`bag` ⌘ Selection of rosbag messages

`BagSelection` object handle

Selection of rosbag messages, returned as a `BagSelection` object handle.

See Also

`readMessages` | `select` | `timeseries`

Introduced in R2015a

rosduration

Create a ROS duration object

Syntax

```
dur = rosduration
dur = rosduration(totalSecs)
dur = rosduration(secs,nsecs)
```

Description

`dur = rosduration` returns a default ROS duration object. The properties for seconds and nanoseconds are set to 0.

`dur = rosduration(totalSecs)` initializes the time values for seconds and nanoseconds based on `totalSecs`, which represents the time in seconds as a floating-point number.

`dur = rosduration(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 = rosduration(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
dur5 = rosduration(-1,2000000)
```

```
dur4 =
```

```
  ROS Duration with properties:

     Sec: -80
    Nsec: 498000000


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')
timeFuture = time + dur3
```

```
time =

  ROS Time with properties:

     Sec: 1.5043e+09
    Nsec: 251000000


timeFuture =

  ROS Time with properties:

     Sec: 1.5043e+09
```

```
Nsec: 753000000
```

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

`rosmessage` | `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 `rosmsg 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)
```

Å òCreate Custom Messages from ROS Packageó

Input Arguments

folderpath — Path to ROS package folders

character vector

Path to package folders, specified as a 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

Å You must install the Robotics System Toolbox Interface for ROS Custom Messages add-on using `roboticsAddons` to use this function.

See Also

`roboticsAddons`

Topics

òCreate Custom Messages from ROS Packageó
òROS Custom Message Supportó

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

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://bat5823win64:54548/.
Initializing global node /matlab_global_node_20607 with NodeURI http://bat5823win64:545
```

When you are finished, shut down the ROS network.

```
rosshutdown
```

```
Shutting down global node /matlab_global_node_20607 with NodeURI http://bat5823win64:54
Shutting down ROS master on http://bat5823win64:54548/.
```

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://192.168.154.1:50
```

Shut down the ROS network when you are finished.

```
rosshutdown
```

```
Shutting down global node /matlab_global_node_76850 with NodeURI http://192.168.154.1:5
```

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.

```
rosshutdown
```

```
Shutting down global node /test_node with NodeURI http://192.168.1.1:59577/
```

Å òConnect to a ROS Networkó

Input Arguments

`hostname` ∞ Host name or IP address

character vector

Host name or IP address, specified as a character vector.

`port` ∞ Port number

scalar

Port number used to connect to the ROS master, specified as a scalar.

`URI` ∞ URI for ROS master

character vector

URI for ROS master, specified as a 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 single 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` | `rosshutdown`

Topics

Connect to a ROS Network

Introduced in R2015a

rosmessage

Create ROS messages

Syntax

```
msg = rosmessage(messagetype)
```

```
msg = rosmessage(pub)
msg = rosmessage(sub)
msg = rosmessage(client)
msg = rosmessage(server)
```

Description

`msg = rosmessage(messagetype)` creates an empty ROS message object with message type. The `messagetype` character vector is case-sensitive and no partial matches are allowed. It must match a message on the list given by calling `rosmsg('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 = rosmessage('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://bat5823win64:54364/.
Initializing global node /matlab_global_node_55095 with NodeURI http://bat5823win64:543
```

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 = rosmessage(chatpub);
msg.Data = 'test phrase';
```

Send message via the publisher.

```
send(chatpub,msg);
```

Shutdown ROS network.

```
rosshutdown
```

```
Shutting down global node /matlab_global_node_55095 with NodeURI http://bat5823win64:54
Shutting down ROS master on http://bat5823win64:54364/.
```

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 = [rosmessage('std_msgs/String') rosmessage('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(@(~) rosmessage('std_msgs/String'),zeros(1,50));
```

Preallocate a cell array of ROS messages.

```
msgCell = cellfun(@(~) rosmessage('std_msgs/String'),cell(1,50),'UniformOutput',false);
```

Input Arguments

`messagetype` — Message type

character vector

Message type, specified as a character vector. The character vector is case-sensitive and no partial matches are allowed. It must match a message on the list given by calling `rosmsg('list')`. To avoid errors in entering the message type, you can use `rostype` with tab completion to browse the list of all available types.

`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 م **ROS message**
`Message` object handle

ROS message, returned as a `Message` object handle.

See Also

`roboticsAddons` | `rosmsg` | `rostopic`

Topics

òWork with Basic ROS Messagesó
òBuilt-In Message Supportó

Introduced in R2015a

rosmsg

Retrieve information about ROS messages and message types

Syntax

```
rosmsg show msgtype
rosmsg md5 msgtype
rosmsg list

msginfo = rosmsg('show', msgtype)
msgmd5 = rosmsg('md5', msgtype)
msglist = rosmsg('list')
```

Description

`rosmsg show msgtype` returns the definition of the `msgtype` message.

`rosmsg md5 msgtype` returns the MD5 checksum of the `msgtype` message.

`rosmsg list` returns all available message types that you can use in MATLAB.

`msginfo = rosmsg('show', msgtype)` returns the definition of the `msgtype` message as a character vector.

`msgmd5 = rosmsg('md5', msgtype)` returns the ìMD5î checksum of the `msgtype` message as a character vector.

`msglist = rosmsg('list')` returns a cell array containing all available message types that you can use in MATLAB.

Examples

Retrieve Message Type Definition

```
msgInfo = rosmsg('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 = rosmsg('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` مم 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` مم 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

rosnode

Retrieve information about ROS network nodes

Syntax

```
rosnode list
rosnode info nodename
rosnode ping nodename
```

```
nodelist = rosnode('list')
nodeinfo = rosnode('info',nodename)
rosnode('ping',nodename)
```

Description

`rosnode list` returns a list of all nodes registered on the ROS network. Use these nodes to exchange data between MATLAB and the ROS network.

`rosnode info nodename` returns a structure containing the name, URI, publications, subscriptions, and services of a specific ROS node,`nodename`.

`rosnode ping nodename` pings a specific node, `nodename`, and displays the response time.

`nodelist = rosnode('list')` returns a cell array of character vectors containing the nodes registered on the ROS network.

`nodeinfo = rosnode('info',nodename)` returns a structure containing the name, URI, publications, subscriptions, and services of a specific ROS node, `nodename`.

`rosnode('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.

```
rosshutdown
```

```
Shutting down global node /matlab_global_node_90274 with NodeURI http://192.168.203.1:6
```

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://192.168.154.1:61
```

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.

```
rosshutdown
```

```
Shutting down global node /matlab_global_node_40513 with NodeURI http://192.168.154.1:6
```

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.

```
rosshutdown
```

```
Shutting down global node /matlab_global_node_88195 with NodeURI http://192.168.154.1:5
```

Input Arguments

nodename — Name of node

character vector

Name of node, specified as a 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 properties: `'NodeName'`, `'URI'`, `'Publications'`, `'Subscriptions'`, and `'Services'`. Access these properties using dot syntax, for example, `nodeinfo.NodeName`.

nodelist — 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 òLimitationsó on page 2-203 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.

Å 32-bit integer ñ `int32`

Å boolean ñ `logical`

Å double ñ `double`

Å string ñ character vector (`char`)

Å list ñ cell array (`cell`)

Å dictionary ñ 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://bat5823win64:54836/.
Initializing global node /matlab_global_node_21754 with NodeURI http://bat5823win64:548
```

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.

```
rosshutdown
```

```
Shutting down global node /matlab_global_node_21754 with NodeURI http://bat5823win64:54
Shutting down ROS master on http://bat5823win64:54836/.
```

Create Parameter Tree Object and View Parameters

Connect to the ROS network. ROS parameters should already be available on the ROS master.

```
rosinit('192.168.154.131')
```

```
Initializing global node /matlab_global_node_82870 with NodeURI http://192.168.154.1:58
```

Create a `ParameterTree` object using `rosparam`.

```
ptree = rosparam;
```

List the available parameters on the ROS master.

```
ptree.AvailableParameters
```

```
ans =

  33×1 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'
    '/rosdistro'
    '/roslaunch/uris/host_192_168_154_131__41131'
    '/rosversion'
    '/run_id'
    '/use_sim_time'
```

Shut down the ROS network.

```
rosshutdown
```

```
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.

```
rosinit
```

```
Initializing ROS master on http://bat5823win64:54437/.
Initializing global node /matlab_global_node_95144 with NodeURI http://bat5823win64:544
```

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_95144 with NodeURI http://bat5823win64:54
Shutting down ROS master on http://bat5823win64:54437/.
```

Input Arguments

namespace — **ROS parameter namespace**

character vector

ROS parameter namespace, specified as a character vector. All parameter names starting with this character vector are listed when calling `rosparam('list',namespace)`.

`pname` ⌘ ROS parameter name
character vector

ROS parameter name, specified as a character vector.

`pval` ⌘ ROS parameter value or dictionary of values
`int32` | `logical` | `double` | 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:

Å 32-bit integers ñ `int32`

Å booleans ñ `logical`

Å doubles ñ `double`

Å strings ñ character vector `char`

Å lists ñ cell array

Å dictionaries ñ 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	`int32`
boolean	`logical`
double	`double`
string	character vector (`char`)
list	cell array (`cell`)
dictionary	structure (`struct`)

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` ñ If `pval` is `'true'` or `'false'`
  - `int32` ñ If `pval` is an integer, for example, `5`
  - `double` ñ If `pval` is a fractional number, for example, `1.256`
  - character vector ñ If `pval` is any other value

See Also

Functions

`del` | `get` | `has` | `search` | `set`

Using 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://192.168.154.1:56
```

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.

```
rosshutdown
```

```
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://192.168.154.1:55
```

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.

```
rosshutdown
```

```
Shutting down global node /matlab_global_node_76389 with NodeURI http://192.168.154.1:5
```

Input Arguments

svcname ⌕ Name of service

character vector

Name of service, specified as a 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. After calling `rosshutdown`, any ROS entities that depend on the global node, for example, subscribers created with `rossubscriber`, are deleted and become unstable.

Examples

Start ROS Core and Global Node

```
rosinit
```

```
Initializing ROS master on http://bat5823win64:54548/.
Initializing global node /matlab_global_node_20607 with NodeURI http://bat5823win64:545
```

When you are finished, shut down the ROS network.

```
rosshutdown
```

```
Shutting down global node /matlab_global_node_20607 with NodeURI http://bat5823win64:54
Shutting down ROS master on http://bat5823win64:54548/.
```

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://192.168.154.1:56
```

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.

```
rosinit('192.168.154.131')
```

```
Initializing global node /matlab_global_node_28473 with NodeURI http://192.168.154.1:57
```

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://192.168.154.1:58
```

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

character vector

ROS topic name, specified as a 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 `rosmessage` 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 = rosmessage(rostype.std_msgs_String)
```

```
msg =

  ROS String message with properties:

    MessageType: 'std_msgs/String'
           Data: ''

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

Å òBuilt-In Message Supportó

Å òWork with Basic ROS Messagesó

See Also

rosmessage | rostopic

Topics

òBuilt-In Message Supportó
òWork with Basic ROS Messagesó

Introduced in R2015a

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 =

    1.0000         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 م 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 MATLABÈ Coder Ê.

See Also

`axang2rotm`

Introduced in R2015a

rotm2eul

Convert rotation matrix to Euler angles

Syntax

```
eul = rotm2eul(rotm)
eul = rotm2eul(rotm,sequence)
```

Description

`eul = rotm2eul(rotm)` converts a rotation matrix, `rotm`, to the corresponding Euler angles, `eul`. The input rotation matrix must be in the premultiply form for rotations. The default order for Euler angle rotations is `'ZYX'`.

`eul = rotm2eul(rotm,sequence)` converts a rotation matrix to Euler angles. The Euler angles are specified in the axis rotation sequence, `sequence`. The default order for Euler angle rotations is `'ZYX'`.

Examples

Convert Rotation Matrix to Euler Angles

```
rotm = [0 0 1; 0 1 0; -1 0 0];
eulZYX = rotm2eul(rotm)
```

```
eulZYX =

         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 =

   -3.1416   -1.5708   -3.1416
```

Input Arguments

`rotm` ⌘ Rotation matrix

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

Rotation matrix, specified as a 3-by-3-by-*n* matrix containing *n* rotation matrices. Each rotation matrix has a size of 3-by-3 and is orthonormal. The input rotation matrix must be in the premultiply form for rotations.

Note Rotation matrices that are slightly non-orthonormal can give complex outputs. Consider validating your matrix before inputting to the function.

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

`sequence` ⌘ Axis rotation sequence

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

Axis rotation sequence for the Euler angles, specified as one of these character vectors:

Å `'ZYX'` (default) ð The order of rotation angles is *z*-axis, *y*-axis, *x*-axis.

Å `'ZYZ'` ð The order of rotation angles is *z*-axis, *y*-axis, *z*-axis.

Å `'XYZ'` ð The order of rotation angles is *x*-axis, *y*-axis, *z*-axis.

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 MATLABÈ Coder Ê.

See Also

`eul2rotm`

Introduced in R2015a

rotm2quat

Convert rotation matrix to quaternion

Syntax

```
quat = rotm2quat(rotm)
```

Description

`quat = rotm2quat(rotm)` converts a rotation matrix, `rotm`, to the corresponding unit quaternion representation, `quat`. The input rotation matrix must be in the premultiply form for rotations.

Examples

Convert Rotation Matrix to Quaternion

```
rotm = [0 0 1; 0 1 0; -1 0 0];
quat = rotm2quat(rotm)
```

```
quat =

    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` مم **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 MATLABÈ Coder Ê.

See Also

`quat2rotm`

Introduced in R2015a

rotm2tform

Convert rotation matrix to homogeneous transformation

Syntax

```
tform = rotm2tform(rotm)
```

Description

`tform = rotm2tform(rotm)` converts the rotation matrix, `rotm`, into a homogeneous transformation matrix, `tform`. The input rotation matrix must be in the premultiply form for rotations. When using the transformation matrix, premultiply it with the coordinates to be transformed (as opposed to postmultiplying).

Examples

Convert Rotation Matrix to Homogeneous Transformation

```
rotm = [1 0 0 ; 0 -1 0; 0 0 -1];
tform = rotm2tform(rotm)
```

```
tform =

     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 MATLABÈ Coder Ê.

See Also

`tform2rotm`

Introduced in R2015a

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
```

Å òGenerate a Standalone ROS Node from SimulinkÈó

Input Arguments

device م ROS device

rosdevice object

ROS device, specified as a rosdevice object.

See Also

isCoreRunning | rosdevice | stopCore

Topics

òGenerate a Standalone ROS Node from SimulinkÈó

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 = '192.168.154.131';
d = rosdevice(ipaddress,'user','password');
d.ROSFolder = '/opt/ros/hydro';
d.CatkinWorkspace = '~/catkin_ws_test'
```

```
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. Connect MATLABÈ to the ROS master using `rosinit`. This core enables you to run ROS nodes on your ROS device.

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

```
Another roscore / ROS master is already running on the ROS device. Use the 'stopCore' f
Initializing global node /matlab_global_node_15046 with NodeURI http://192.168.154.1:56
```

Check the available ROS nodes on the connected ROS device. These nodes were generated from SimulinkÈ models following the process in the Generate a standalone ROS node from SimulinkÈ 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_15046 with NodeURI http://192.168.154.1:5
```

Run Multiple ROS Nodes

Run multiple ROS nodes on a connected ROS device. ROS nodes can be generated using SimulinkÈ 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 SimulinkÈ). 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 = '192.168.154.131';
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. Connect MATLABÈ to the ROS master using `rosinit`. This ROS core enables you to run ROS nodes on your ROS device.

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

```
Initializing global node /matlab_global_node_68749 with NodeURI http://192.168.154.1:64
```

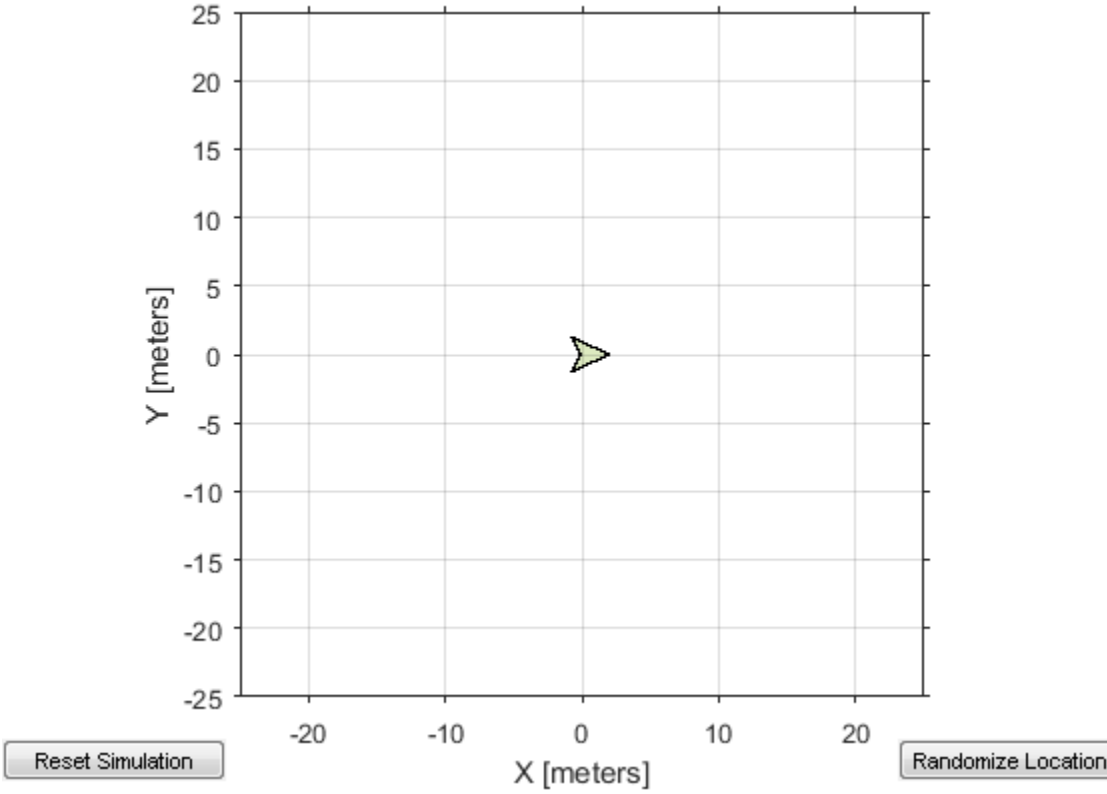
Check the available ROS nodes on the connected ROS device. These nodes were generated from SimulinkÈ models following the process in the Generate a standalone ROS node from SimulinkÈ example.

```
d.AvailableNodes
```

```
ans =

  1×2 cell array

    'robotcontroller'    'robotcontroller2'
```

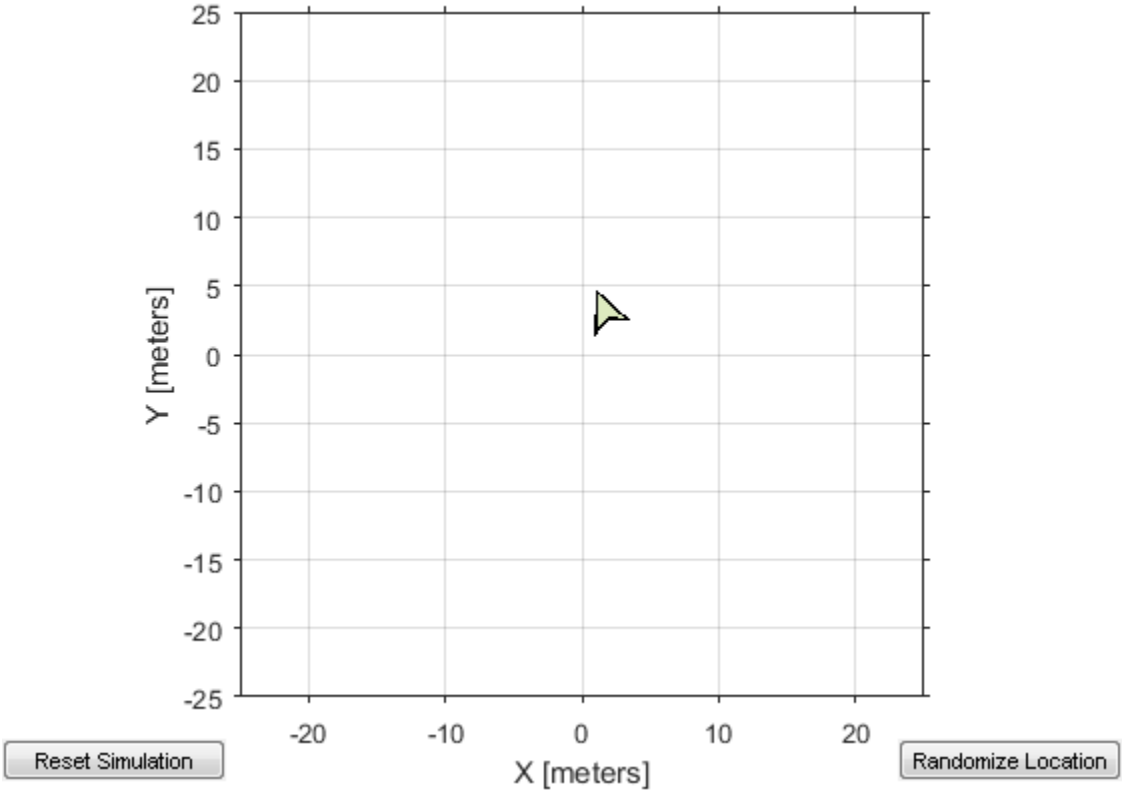
Start up the Robot Simulator using `ExampleHelperSimulinkRobotROS`. This simulator automatically connects to the ROS master on the ROS device. You will use this simulator to run a ROS node and control the robot.

```
sim = ExampleHelperSimulinkRobotROS;
```



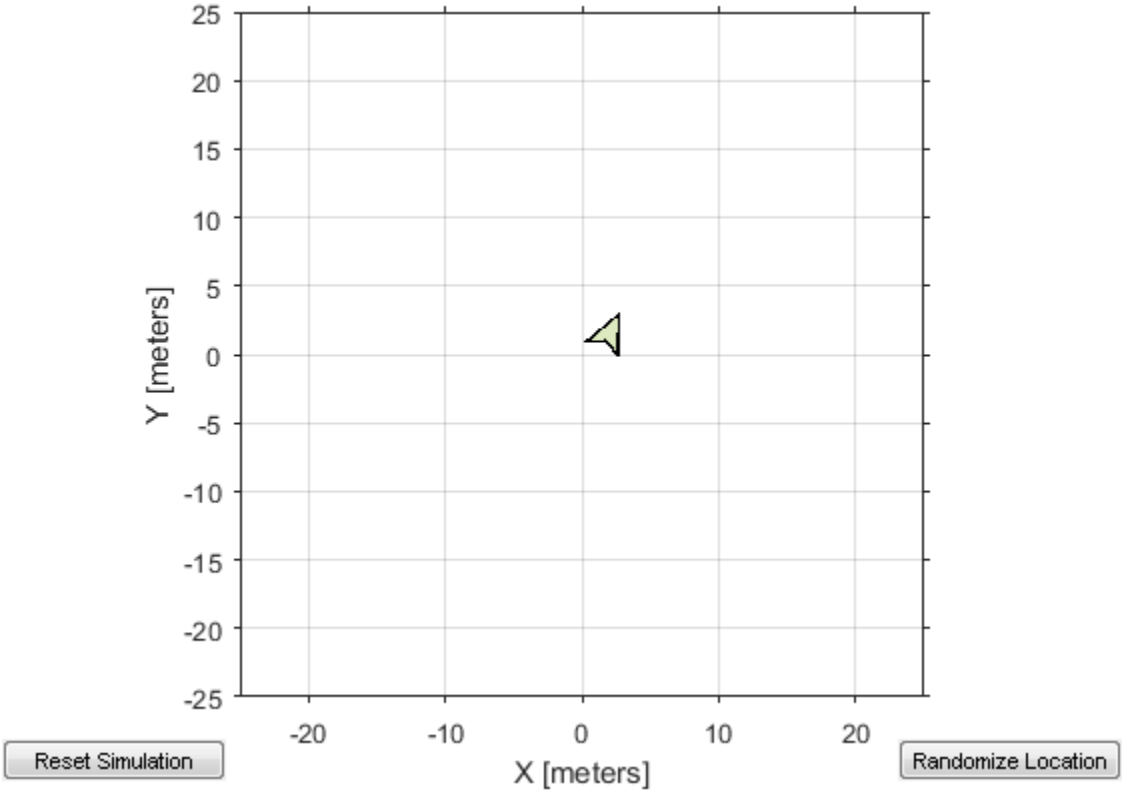
Run a ROS node, specifying the node name. The `'robotcontroller'` node commands the robot to a specific location (`[-10 10]`). Wait to see the robot drive.

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



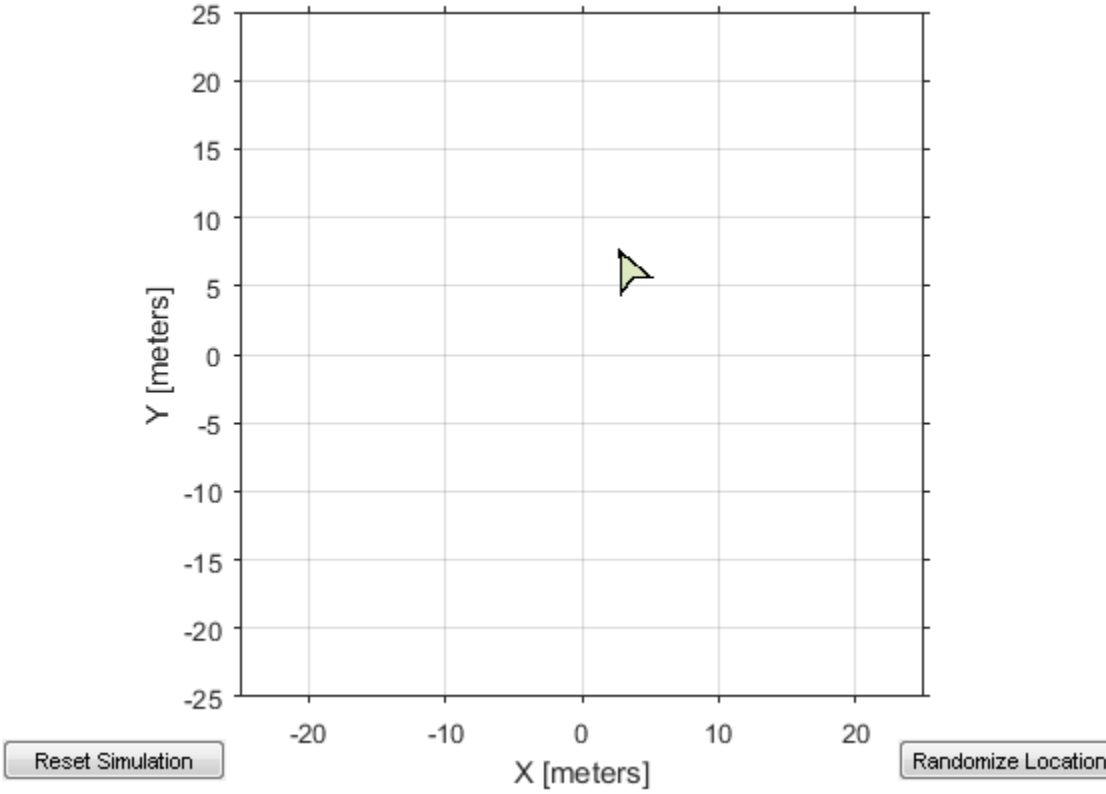
Reset the Robot Simulator to reset the robot position. Alternatively, click Reset Simulation. Because the node is still running, the robot continues back to the specific location. To stop sending commands, stop the node.

```
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.

```
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')
rosshutdown
stopCore(d)
```

```
Shutting down global node /matlab_global_node_68749 with NodeURI http://192.168.154.1:6
```

Å òConnect to a ROS Networkó

Å òGenerate a Standalone ROS Node from SimulinkÈó

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

òConnect to a ROS Networkó
òGenerate a Standalone ROS Node from SimulinkÈó

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 ò_opticaló 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 ò_opticaló 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

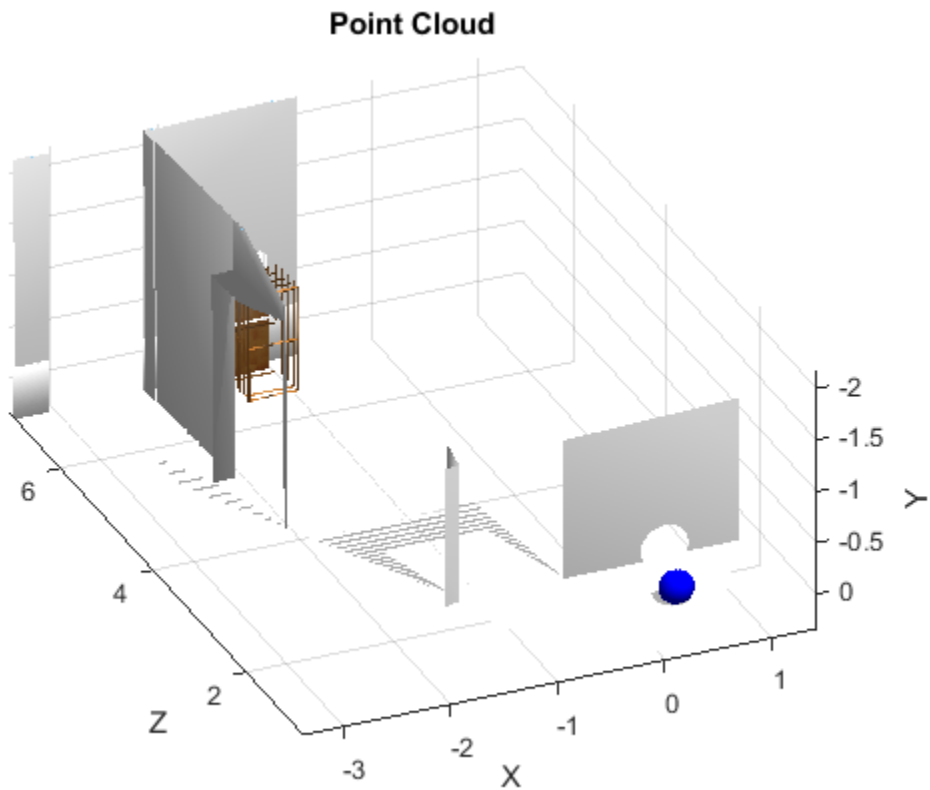
Connect to a ROS network. Subscribe to a point cloud message topic.

```
rosinit('192.168.154.131')
sub = rossubscriber('/camera/depth/points');
pause(1)
```

```
Initializing global node /matlab_global_node_47682 with NodeURI http://192.168.154.1:60
```

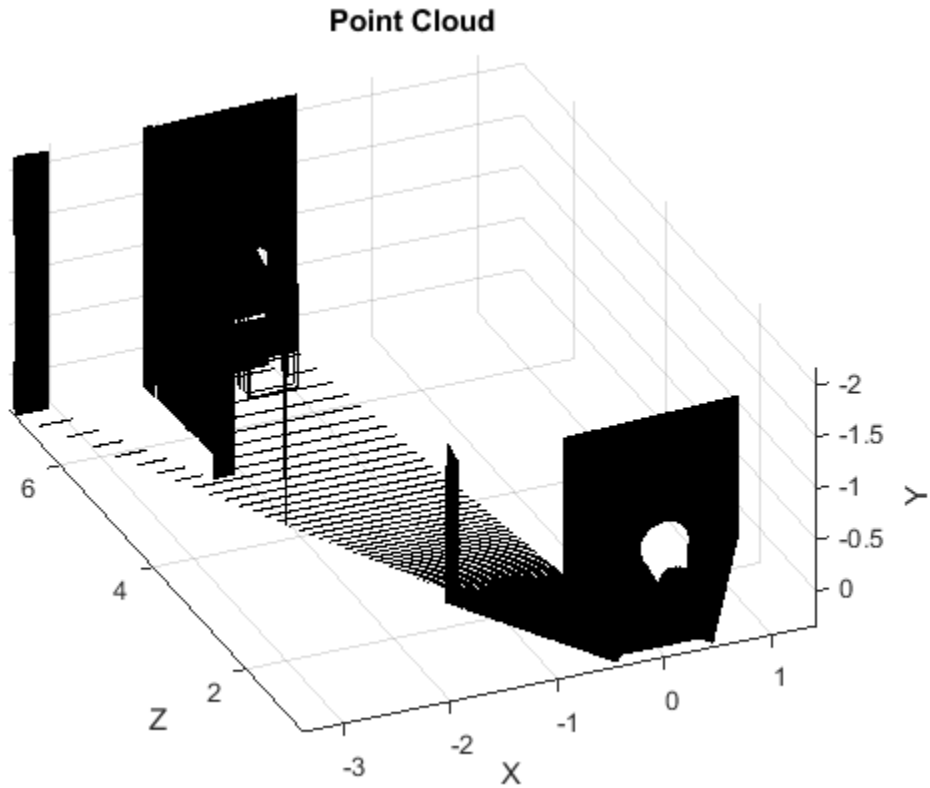
Get the latest point cloud message. Plot the point cloud.

```
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 single 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) | `'none'` | RGB triplet | `'r'` | `'g'` | `'b'` | ...

Marker outline color, specified `'flat'`, an RGB triplet, or one of the color options listed in the table. The default value of `'flat'` uses colors from the `CData` property.

For a custom color, specify an RGB triplet. 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]`. Alternatively, you can specify some common colors by name. This table lists the long and short color name options and the equivalent RGB triplet values.

Option	Description	Equivalent RGB Triplet
`'red'` or `'r'`	Red	`[1 0 0]`
`'green'` or `'g'`	Green	`[0 1 0]`
`'blue'` or `'b'`	Blue	`[0 0 1]`
`'yellow'` or `'y'`	Yellow	`[1 1 0]`
`'magenta'` or `'m'`	Magenta	`[1 0 1]`
`'cyan'` or `'c'`	Cyan	`[0 1 1]`
`'white'` or `'w'`	White	`[1 1 1]`
`'black'` or `'k'`	Black	`[0 0 0]`
`'none'`	No color	Not applicable

Example: `[0.5 0.5 0.5]`

Example: `'blue'`

`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 character vector `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:

Å 32-bit integers ñ `int32`

Å booleans ñ `logical`

Å doubles ñ `double`

Å strings ñ character vector `char`

Å lists ñ cell array

Å dictionaries ñ 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://192.168.154.1:58
```

Create a parameter tree.

```
ptree = rosparam;
```

Search for parameter names that contain `'gravity'`.

```
[pnames,pvalues] = search(ptree,'gravity')
```

```
pnames =

  1×3 cell array

    '/gazebo/gravity_x'    '/gazebo/gravity_y'    '/gazebo/gravity_z'


pvalues =

  3×1 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

character vector

ROS parameter search string specified as a character vector. `search` returns all parameters that contain this character vector.

Output Arguments

pnames م 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 م 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:

Å 32-bit integers ñ `int32`

Å booleans ñ `logical`

Å doubles ñ `double`

Å strings ñ character vector `char`

Å lists ñ cell array

Å dictionaries ñ structure

Base64ðencoded binary data and iso8601 data from ROS are not supported.

Limitations

Base64ðencoded 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. `Name`must appear inside single quotes `('')`. You can specify several name-value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Examples

Create Copy of rosbag

Retreive the rosbag. Specify the file path.

```
filepath = fullfile(fileparts(which('ROSWorkingWithRosbagsExample')),...
                    'data','ex_multiple_topics.bag');
bag = rosbag(filepath);
```

Use `select` with no selection criteria to create a copy of the rosbag.

```
bagCopy = select(bag);
```

Select Subset of Messages In rosbag

Retreive the rosbag. Specify the file path.

```
filepath = fullfile(fileparts(which('ROSWorkingWithRosbagsExample')),...
                    'data','ex_multiple_topics.bag');
bag = rosbag(filepath);
```

Select all messages within the first second of the rosbag.

```
bag = select(bag,'Time',[bag.StartTime,bag.StartTime + 1]);
```

Input Arguments

`bag` ⌘ Message of a rosbag

`BagSelection` object

All the 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 single 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

character vector | cell array

ROS message type, specified as a character vector or cell array. Multiple message types can be specified with a cell array of character vectors.

`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
character vector | cell array

ROS topic name, specified as a character vector or cell array. Multiple topic names can be specified with a cell array of character vectors.

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: [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.

```
rosshutdown
```

```
Shutting down global node /matlab_global_node_10876 with NodeURI http://AH-SRADFORD:651
Shutting down ROS master on http://AH-SRADFORD:11311/.
```

Å ðExchange Data with ROS Publishers and Subscribersó

Input Arguments

pub مم ROS publisher

`Publisher` object handle

ROS publisher, specified as a `Publisher` object handle. You can create the object using `rospublisher`.

msg مم ROS message

`Message` object handle

ROS message, specified as a `Message` object handle.

See Also

`receive` | `rosmessage` | `rospublisher` | `rossubscriber` | `rostopic`

Topics

òExchange Data with ROS Publishers and Subscribersó

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.

Create a ROS action client and get a goal message. The `actClient` object connects to the ROS action server. `goalMsg` is a valid goal message. Update the message parameters with your specific goal.

```
[actClient, goalMsg] = rosactionclient('/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 `rosmessage` and the action client object. This mesasge sends linear and angular velocity commands to a TurtlebotÈ robot.

```
goalMsg = rosmessage(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 `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://192.168.154.1:57
```

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: [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.

```
rosshutdown
```

```
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

òROS Actions Overviewó
òMove a Turtlebot Robot Using ROS Actionsó

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://192.168.154.1:57
```

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: [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.

```
rosshutdown
```

```
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:

Å `'pending'` ñ Goal was received, but has not yet been accepted or rejected.

Å `'active'` ñ Goal was accepted and is running on the server.

Å `'succeeded'` ñ Goal executed successfully.

Å `'preempted'` ñ An action client canceled the goal before it finished executing.

Å `'aborted'` ñ The goal was aborted before it finished executing. The action server typically aborts a goal.

Å `'rejected'` ñ The goal was not accepted after being in the `'pending'` state. The action server typically triggers this status.

Å `'recalled'` ñ A client canceled the goal while it was in the `'pending'` state.

Å `'lost'` ñ An internal error occurred in the action client.

`status` م 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

òROS Actions Overviewó
òMove a Turtlebot Robot Using ROS Actionsó

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 the ROS Network

This example shows how to create a transformation and send it over the ROS network.

Create a ROS transformation tree. You must be connected to a ROS network using `rosinit`. Replace `ipaddress` with your ROS network address.

```
ipaddress = '172.28.194.91';
rosinit(ipaddress)
tftree = rostf;
pause(2);
```

```
Initializing global node /matlab_global_node_61809 with NodeURI http://172.28.194.90:55
```

Verify the transformation you want does not exist. `canTransform` returns false if the transformation is not immediately available.

```
canTransform(tftree,'new_frame','base_link')
```

```
ans =
```

```
   0
```

Create a `TransformStamped` message. Populate 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;
```

```
tform = 

  ROS TransformStamped message with properties:

     MessageType: 'geometry_msgs/TransformStamped'
          Header: [1×1 Header]
    ChildFrameId: ''
       Transform: [1×1 Transform]

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

Send the transformation over the ROS network.

```
sendTransform(tftree,tform)
```

Check if the transformation is now on the ROS network

```
canTransform(tftree,'new_frame','base_link')
```

```
ans = 

   1
```

Shut down the ROS network.

```
rosshutdown
```

```
Shutting down global node /matlab_global_node_61809 with NodeURI http://172.28.194.90:5
```

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.

`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.

Å 32-bit integer ñ `int32`

Å boolean ñ `logical`

Å double ñ `double`

Å string ñ character vector (`char`)

Å list ñ cell array (`cell`)

Å dictionary ñ 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:603
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://bat5823win64:54437/.
Initializing global node /matlab_global_node_95144 with NodeURI http://bat5823win64:544
```

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_95144 with NodeURI http://bat5823win64:54
Shutting down ROS master on http://bat5823win64:54437/.
```

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

character vector

ROS parameter name, specified as a character vector. This character vector must match the parameter name exactly.

`pvalue` ROS parameter value or dictionary of values

`int32` | `logical` | `double` | 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.

ROS Data Type	MATLAB Data Type
32-bit integer	`int32`
boolean	`logical`
double	`double`
string	character vector (`char`)
list	cell array (`cell`)
dictionary	structure (`struct`)

`namespace` ROS parameter namespace

character vector

ROS parameter namespace, specified as a character vector. All parameter names starting with this character vector 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

showrobotics.VectorFieldHistogram.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 `VectorFieldHistrogram` object and range values from the last `step` input.

`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;
```

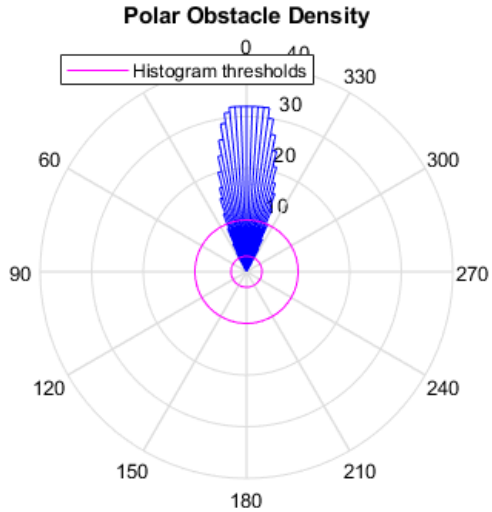
Compute an obstacle-free steering direction.

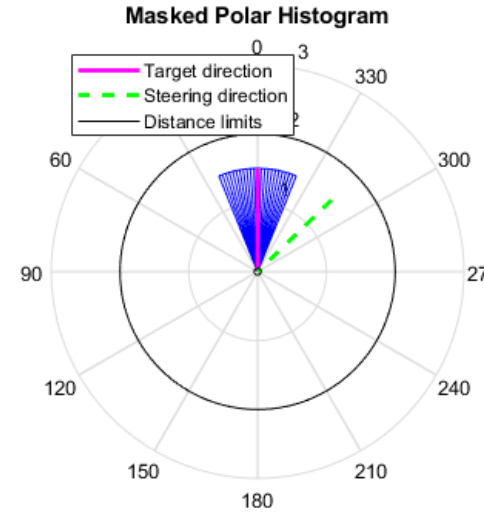
```
steeringDir = vfh(ranges,angles,targetDir)
```

```
steeringDir =

   -0.8014
```

Visualize the `VectorFieldHistogram` computation.

```
h = figure;
set(h,'Position',[50 50 800 400])
show(vfh)
```





Input Arguments

`vfh` مم 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` مم Axes properties

handle

Axes properties, specified as a handle.

Output Arguments

h مم 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` | `robotics.VectorFieldHistogram.step`

Introduced in R2015b

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

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
```

Å òGenerate a Standalone ROS Node from SimulinkÈó

Input Arguments

`device` مم ROS device
`rosdevice` object

ROS device, specified as a `rosdevice` object.

See Also

`isCoreRunning` | `rosdevice` | `runCore`

Topics

òGenerate a Standalone ROS Node from SimulinkÈó

Introduced in R2016b

statistics

Statistics of past execution periods

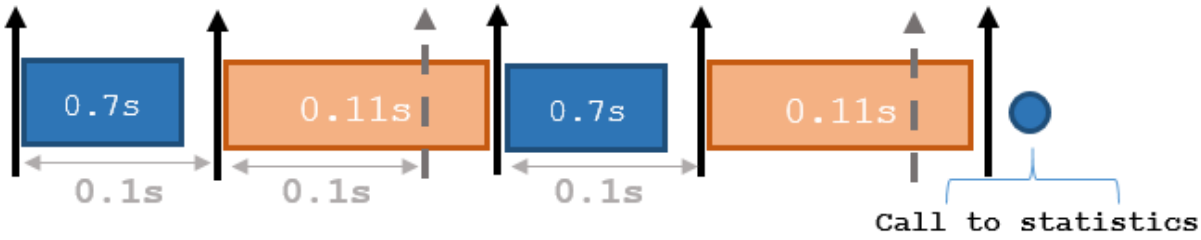
Syntax

```
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` ñ All time periods (returned in seconds) used to calculate statistics as an indexed array. `stats.Period(end)` is the most recent period.
- `NumPeriods` ñ Number of elements in `Periods`
- `AveragePeriod` ñ Average time in seconds
- `StandardDeviation` ñ Standard deviation of all periods in seconds, centered around the mean stored in `AveragePeriod`
- `NumOverruns` ñ 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: 0.0050
           NumOverruns: 0
```

Å òExecute Code at a Fixed-Rateó

See Also

`robotics.Rate` | `rosrate` | `waitfor`

Topics

òExecute Code at a Fixed-Rateó

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 = '192.168.154.131';
d = rosdevice(ipaddress,'user','password');
d.ROSFolder = '/opt/ros/hydro';
d.CatkinWorkspace = '~/catkin_ws_test'
```

```
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. Connect MATLABÈ to the ROS master using `rosinit`. This core enables you to run ROS nodes on your ROS device.

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

```
Another roscore / ROS master is already running on the ROS device. Use the 'stopCore' f
Initializing global node /matlab_global_node_15046 with NodeURI http://192.168.154.1:56
```

Check the available ROS nodes on the connected ROS device. These nodes were generated from SimulinkÈ models following the process in the Generate a standalone ROS node from SimulinkÈ 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_15046 with NodeURI http://192.168.154.1:5
```

Run Multiple ROS Nodes

Run multiple ROS nodes on a connected ROS device. ROS nodes can be generated using SimulinkÈ 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 SimulinkÈ). 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 = '192.168.154.131';
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. Connect MATLABÈ to the ROS master using `rosinit`. This ROS core enables you to run ROS nodes on your ROS device.

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

```
Initializing global node /matlab_global_node_68749 with NodeURI http://192.168.154.1:64
```

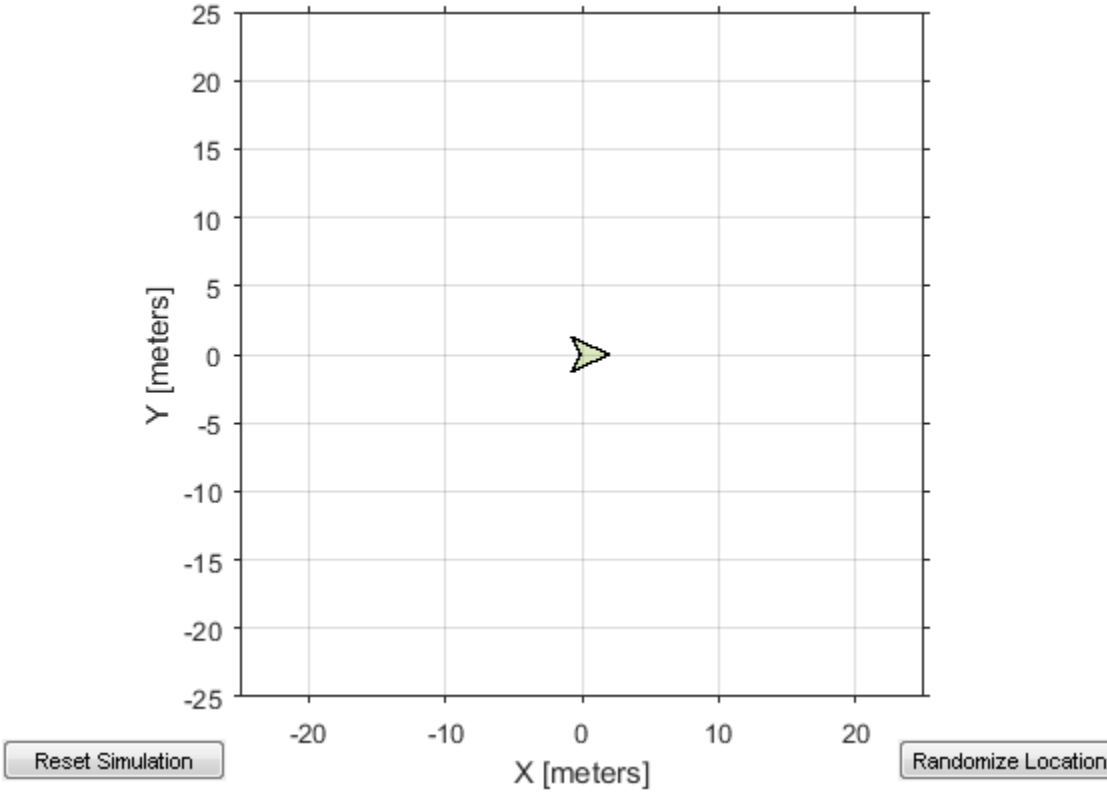
Check the available ROS nodes on the connected ROS device. These nodes were generated from SimulinkÈ models following the process in the Generate a standalone ROS node from SimulinkÈ example.

```
d.AvailableNodes
```

```
ans =

  1×2 cell array

    'robotcontroller'    'robotcontroller2'
```

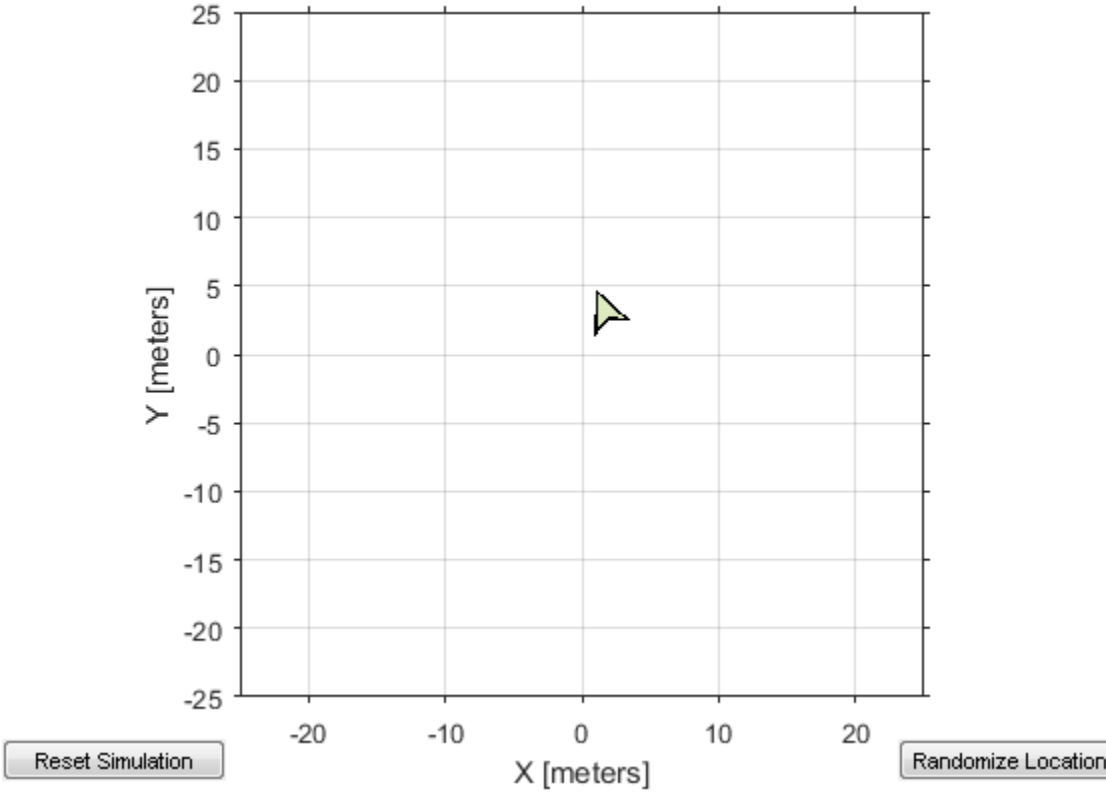
Start up the Robot Simulator using `ExampleHelperSimulinkRobotROS`. This simulator automatically connects to the ROS master on the ROS device. You will use this simulator to run a ROS node and control the robot.

```
sim = ExampleHelperSimulinkRobotROS;
```



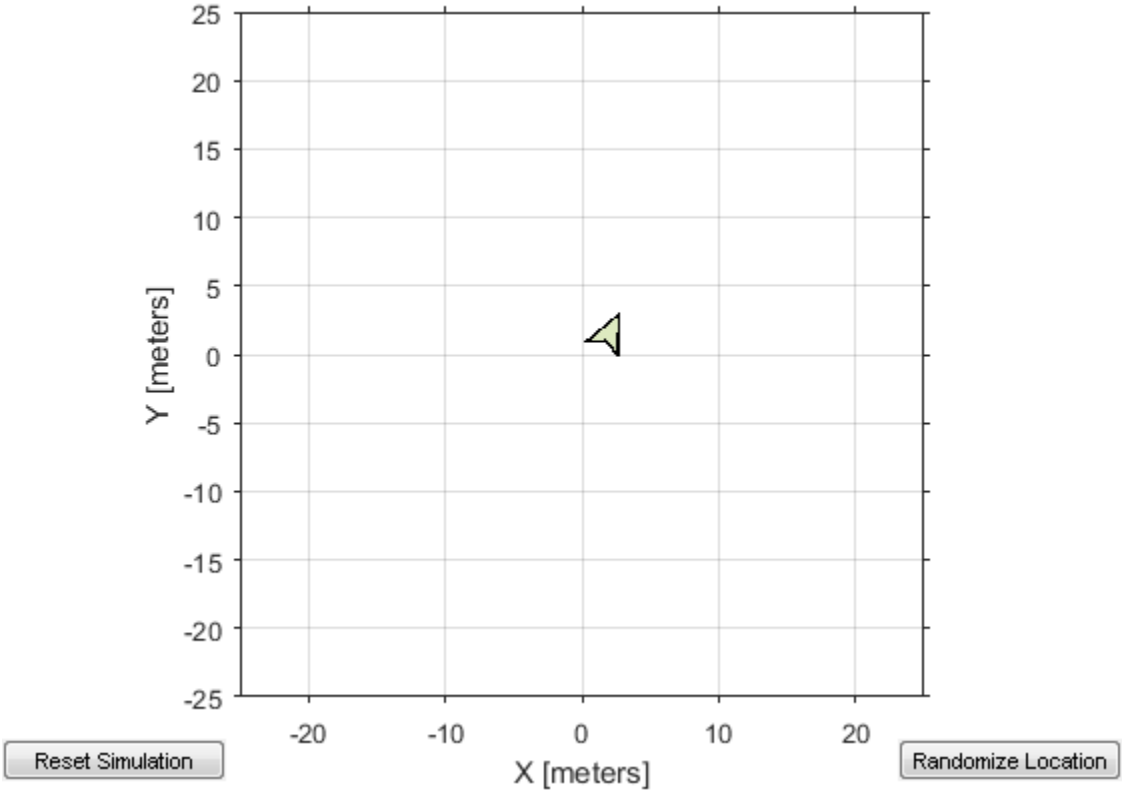
Run a ROS node, specifying the node name. The `'robotcontroller'` node commands the robot to a specific location (`[-10 10]`). Wait to see the robot drive.

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



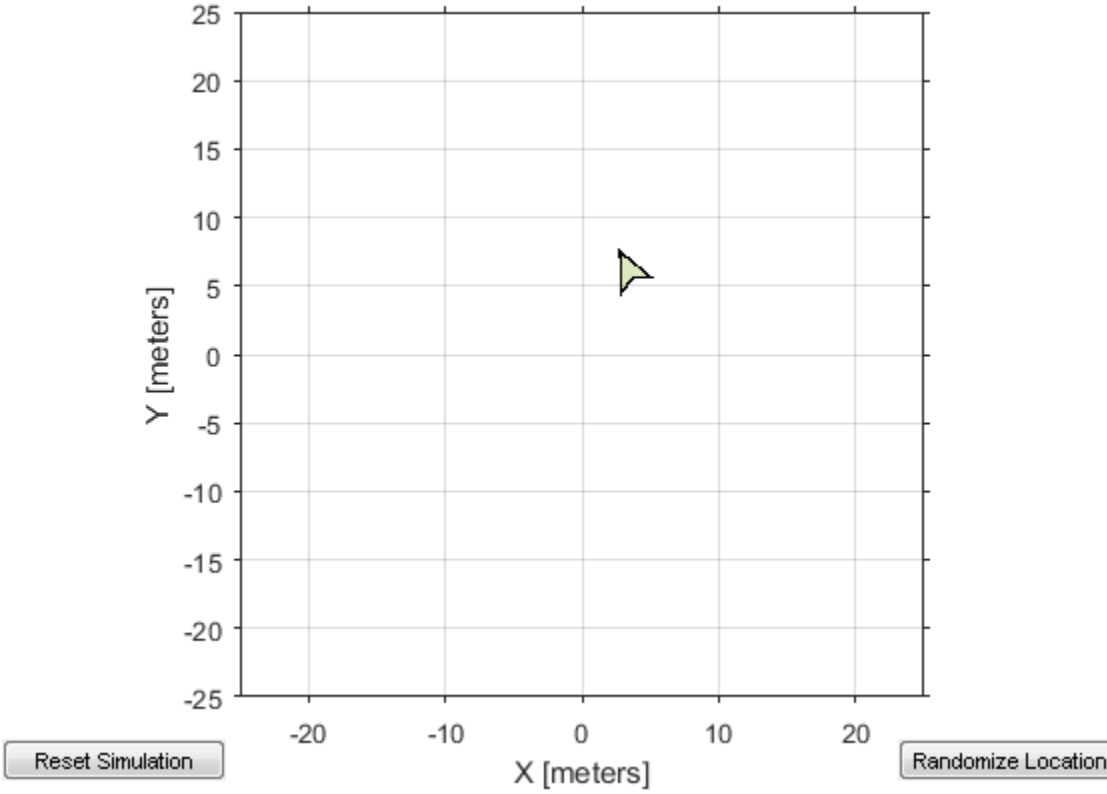
Reset the Robot Simulator to reset the robot position. Alternatively, click Reset Simulation. Because the node is still running, the robot continues back to the specific location. To stop sending commands, stop the node.

```
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.

```
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')
rosshutdown
stopCore(d)
```

```
Shutting down global node /matlab_global_node_68749 with NodeURI http://192.168.154.1:6
```

Å òGenerate a Standalone ROS Node from SimulinkÈó

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

òGenerate a Standalone ROS Node from SimulinkÈó

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 LinuxÈ 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 =

    1.0000         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 MATLABÈ Coder Ê.

See Also

`axang2tform`

Introduced in R2015a

tform2eul

Extract Euler angles from homogeneous transformation

Syntax

```
eul = tform2eul(tform)
eul = tform2eul(tform, sequence)
```

Description

`eul = tform2eul(tform)` extracts the rotational component from a homogeneous transformation, `tform`, and returns it as Euler angles, `eul`. The translational components of `tform` are ignored. The input homogeneous transformation must be in the premultiply form for transformations. The default order for Euler angle rotations is `'ZYX'`.

`eul = tform2eul(tform, sequence)` extracts the Euler angles, `eul`, from a homogeneous transformation, `tform`, using the specified rotation sequence, `sequence`. The default order for Euler angle rotations is `'ZYX'`.

Examples

Extract Euler Angles from Homogeneous Transformation Matrix

```
tform = [1 0 0 0.5; 0 -1 0 5; 0 0 -1 -1.2; 0 0 0 1];
eulZYX = tform2eul(tform)
```

```
eulZYX =

         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 =

         0   -3.1416    3.1416
```

Input Arguments

`tform` م 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` م Axis rotation sequence

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

Axis rotation sequence for the Euler angles, specified as one of these character vectors:

Å `'ZYX'` (default) ð The order of rotation angles is *z*-axis, *y*-axis, *x*-axis.

Å `'ZYZ'` ð The order of rotation angles is *z*-axis, *y*-axis, *z*-axis.

Å `'XYZ'` ð The order of rotation angles is *x*-axis, *y*-axis, *z*-axis.

Output Arguments

`eul` م 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 MATLABÈ Coder Ê.

See Also

`eul2tform`

Introduced in R2015a

tform2quat

Extract quaternion from homogeneous transformation

Syntax

```
quat = tform2quat(tform)
```

Description

`quat = tform2quat(tform)` extracts the rotational component from a homogeneous transformation, `tform`, and returns it as a quaternion, `quat`. The translational components of `tform` are ignored. The input homogeneous transformation must be in the premultiply form for transformations.

Examples

Extract Quaternion from Homogeneous Transformation

```
tform = [1 0 0 0; 0 -1 0 0; 0 0 -1 0; 0 0 0 1];
quat = tform2quat(tform)
```

```
quat =

     0     1     0     0
```

Input Arguments

`tform` مم 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` مم **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 MATLABÈ Coder Ê.

See Also

`quat2tform`

Introduced in R2015a

tform2rotm

Extract rotation matrix from homogeneous transformation

Syntax

```
rotm = tform2rotm(tform)
```

Description

`rotm = tform2rotm(tform)` extracts the rotational component from a homogeneous transformation, `tform`, and returns it as an orthonormal rotation matrix, `rotm`. The translational components of `tform` are ignored. The input homogeneous transformation must be in the pre-multiply form for transformations. When using the rotation matrix, premultiply it with the coordinates to be rotated (as opposed to postmultiplying).

Examples

Convert Homogeneous Transformation to Rotation Matrix

```
tform = [1 0 0 0; 0 -1 0 0; 0 0 -1 0; 0 0 0 1];
rotm = tform2rotm(tform)
```

```
rotm =

     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 MATLABÈ Coder Ê.

See Also

`rotm2tform`

Introduced in R2015a

tform2trvec

Extract translation vector from homogeneous transformation

Syntax

```
trvec = tform2trvec(tform)
```

Description

`trvec = tform2trvec(tform)` extracts the Cartesian representation of translation vector, `trvec` , from a homogeneous transformation, `tform`. The rotational components of `tform` are ignored. The input homogeneous transformation must be in the premultiply form for transformations.

Examples

Extract Translation Vector from Homogeneous Transformation

```
tform = [1 0 0 0.5; 0 -1 0 5; 0 0 -1 -1.2; 0 0 0 1];
trvec = tform2trvec(tform)
```

```
trvec =

    0.5000    5.0000   -1.2000
```

Input Arguments

`tform` — Homogeneous transformation

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

Homogeneous transformation, specified by a 4-by-4-by-*n* matrix of *n* homogeneous transformations. The input homogeneous transformation must be in the premultiply form for transformations.

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

Output Arguments

`trvec` ⌘ Cartesian representation of a translation vector
n-by-3 matrix

Cartesian representation of a translation vector, returned as an *n*-by-3 matrix containing *n* translation vectors. Each vector is of the form $t = [x\ y\ z]$.

Example: `[0.5 6 100]`

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLABÈ Coder Ê.

See Also

`trvec2tform`

Introduced in R2015a

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.

```
filepath = fullfile(fileparts(which('ROSWorkingWithRosbagsExample')), 'data', 'ex_multi
bag = rosbag(filepath);
```

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.

```
filepath = fullfile(fileparts(which('ROSWorkingWithRosbagsExample')), 'data', 'ex_multi
bag = rosbag(filepath);
```

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.

```
filepath = fullfile(fileparts(which('ROSWorkingWithRosbagsExample')), 'data', 'ex_multi
bag = rosbag(filepath);
```

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

character vector

Property names, specified as a 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

òTime Series Basicsó (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 setup 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. You must be connected to a ROS network using `rosinit`. Replace `ipaddress` with your ROS network address.

```
ipaddress = '172.28.194.91';
rosinit(ipaddress)
tftree = rostf;
pause(1);
```

```
Initializing global node /matlab_global_node_93523 with NodeURI http://172.28.194.90:64
```

Look at the available frames on the transformation tree.

```
tftree.AvailableFrames
```

```
ans =

    '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'
    '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. This example is looking for the transformation from `'camera_link'` to `'base_link'`.

```
canTransform(tftree,'base_link','camera_link')
```

```
ans =

   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 be retrieved off the ROS network as well.

```
pt = rosmessage('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.

```
rosshutdown
```

```
Shutting down global node /matlab_global_node_93523 with NodeURI http://172.28.194.90:6
```

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. You must be connected to a ROS network using `rosinit`. Replace `ipaddress` with your ROS network address.

```
ipaddress = '192.168.154.131';
rosinit(ipaddress)
tftree = rostf;
pause(2);
```

```
Initializing global node /matlab_global_node_83561 with NodeURI http://192.168.154.1:64
```

Get the transformation from 1 seconds 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.

```
rosshutdown
```

```
Shutting down global node /matlab_global_node_83561 with NodeURI http://192.168.154.1:6
```

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

character vector

Target coordinate frame that entity transforms into, specified as a 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 Ê 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);
```

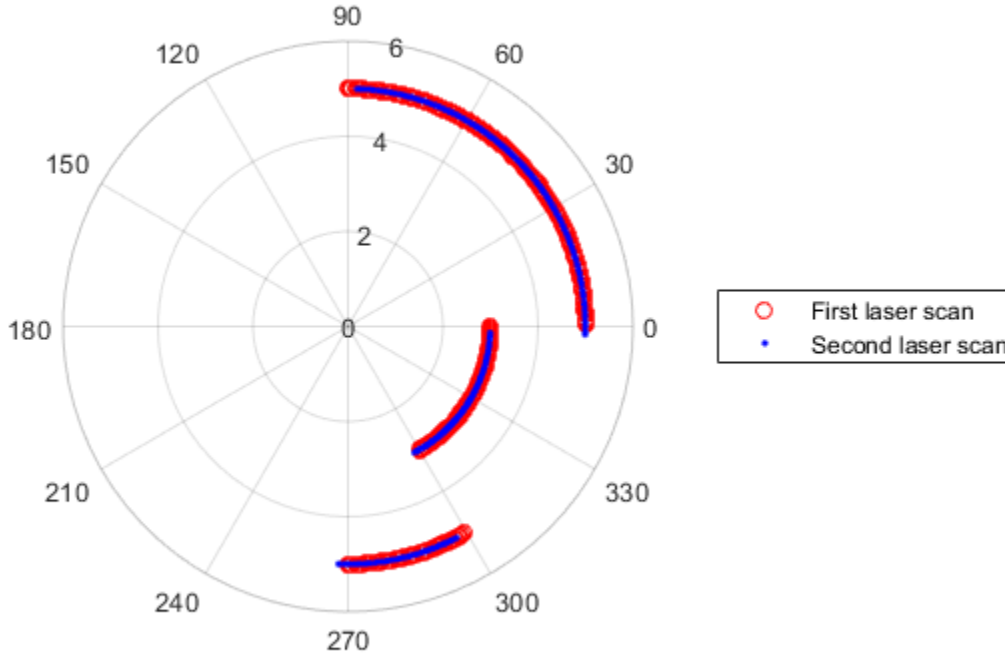
Use scan matching to find the relative pose between the two laser scans. This pose is close to the specified `offset`. You must have an Optimization Toolbox Ê license to use the `matchScans` function.

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

```
pose =

   -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
```



Å òEstimate Robot Pose with Scan Matchingó

Å òCompose a Series of Laser Scans with Pose Changesó

Input Arguments

scan مم **Lidar scan readings**

`lidarScan` object

Lidar scan readings, specified as a `lidarScan` object.

ranges مم **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` مم 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` مم 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` مم Transformed lidar scan readings
`lidarScan` object

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

`transRanges` مم 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` مم 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 MATLABÈ Coder Ê.

See Also

Functions

`LaserScan` | `lidarScan` | `matchScans` | `readCartesian` | `readScanAngles`

Classes

`MonteCarloLocalization` | `OccupancyGrid`

Topics

òEstimate Robot Pose with Scan Matchingó
òCompose a Series of Laser Scans with Pose Changesó

Introduced in R2017a

trvec2tform

Convert translation vector to homogeneous transformation

Syntax

```
tform = trvec2tform(trvec)
```

Description

`tform = trvec2tform(trvec)` converts the Cartesian representation of a translation vector, `trvec`, to the corresponding homogeneous transformation, `tform`. When using the transformation matrix, premultiply it with the coordinates to be transformed (as opposed to postmultiplying).

Examples

Convert Translation Vector to Homogeneous Transformation

```
trvec = [0.5 6 100];
tform = trvec2tform(trvec)
```

```
tform =

    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 MATLABÈ Coder Ê.

See Also

`tform2trvec`

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: 0.006148
Iteration: 2 - Time Elapsed: 1.007045
Iteration: 3 - Time Elapsed: 2.004028
Iteration: 4 - Time Elapsed: 3.003421
Iteration: 5 - Time Elapsed: 4.005034
Iteration: 6 - Time Elapsed: 5.012938
Iteration: 7 - Time Elapsed: 6.000512
Iteration: 8 - Time Elapsed: 7.000391
Iteration: 9 - Time Elapsed: 8.000306
Iteration: 10 - Time Elapsed: 9.001195
```

Each iteration executes at a 1-second interval.

Å òExecute Code at a Fixed-Rateó

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` | `rosrate` | `waitfor`

Topics

òExecute Code at a Fixed-Rateó

Introduced in R2016a

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:

```
rosrun 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://192.168.154.1:51
```

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: [10×1 int32]

  Use showdetails to show the contents of the message


resultState =

  1×9 char array

succeeded
```

Disconnect from the ROS network.

```
rosshutdown
```

```
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://192.168.154.1:57
```

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: [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.

```
rosshutdown
```

```
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` | `rosaction` | `rosactionclient` | `sendGoalAndWait`

Topics

òROS Actions Overviewó
òMove a Turtlebot Robot Using ROS Actionsó

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 functions 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://192.168.154.1:59
```

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.

```
rosshutdown
```

```
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

character vector

Target coordinate frame, specified as a character vector. You can view the available frames for transformation by calling `tftree.AvailableFrames`.

`sourceframe` ⌘ Initial coordinate frame

character vector

Initial coordinate frame, specified as a character vector. You can view the available frames for transformation using `tftree.AvailableFrames`.

`timeout` ⌘ Timeout period

scalar in seconds

Timeout period, specified as a 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` indications an unoccupied location.

msg مم **'nav_msgs/OccupancyGrid'** ROS message
`OccupancyGrid` object handle

`'nav_msgs/OccupancyGrid'` ROS message, specified as a `OccupancyGrid` object handle.

See Also

`readBinaryOccupancyGrid` | `robotics.BinaryOccupancyGrid`

Introduced in R2015a

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.

`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 = rosmessage('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 | RBG image matrix | *m*-by-*n*-by-3 array

Image, specified as a matrix representing a grayscale or RGB image or as a *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 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).

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

Methods مم 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 `a`.

`copy` is a sealed and public method in class `matlab.mixin.Copyable`.

Input Arguments

a مم 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 مم 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 مم World coordinates

n-by-2 vertical array

World coordiantes, specified as an *n*-by-2 vertical array of `[x y]` pairs, where *n* is the number of world coordinates.

Data Types: `double`

`ij` مم 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`

Output Arguments

`occval` مم 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` مم 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` مم 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 مم World coordinates

n-by-2 vertical array

World coordiantes, 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` مم 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` مم 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` مم **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

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 مم 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 مم World coordinates

n-by-2 vertical array

World coordiantes, specified as an *n*-by-2 vertical array of `[x y]` pairs, where *n* is the number of world coordinates.

Data Types: `double`

`ij` مم 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` مم 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` مم 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 coordiantes, specified as an *n*-by-2 vertical array of `[x y]` pairs, where *n* is the number of world coordinates.

Output Arguments

`ij` مم 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

step

System object: robotics.GeneralizedInverseKinematics
Package: robotics

Solve generalized inverse kinematics

Syntax

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

Description

Note Alternatively, 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.

`[configSol,solInfo] = step(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 specified in the generalized inverse kinematics solver, `gik`.

Input Arguments

`gik` مم **Generalized inverse kinematics solver**
`GeneralizedInverseKinematics` object

Inverse kinematics solver, specified as a `GeneralizedInverseKinematics` object.

`initialguess` مم **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 the `DataFormat` property of the `RigidBodyTree` object 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` م Constraint descriptions
constraint objects

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

Å `AimingConstraint`

Å `CartesianBounds`

Å `JointPositionBounds`

Å `OrientationTarget`

Å `PoseTarget`

Å `PositionTarget`

Output Arguments

`configSol` م Robot configuration solution
structure array | vector

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

The structure array contains these fields:

Å `JointName` ñ Character vector for the name of the joint specified in the `RigidBodyTree` robot model

Å `JointPosition` ñ 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` ñ Number of iterations run by the solver.
- `NumRandomRestarts` ñ Number of random restarts because the solver got stuck in a local minimum.
- `ConstraintViolation` ñ 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 of `gik`.
  - `Violation`: Vector of constraint violations for the corresponding constraint type. `0` indicates that the constraint is satisfied.
- `ExitFlag` ñ Code that gives more details on the solver execution and what caused it to return. For the exit flags of each solver type, see òExit Flagsó.
- `Status` ñ Character vector describing whether the solution is within the tolerance (`'success'`) or outside the tolerance and is the best possible solution that the solver could find (`'best available'`).

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 Ê 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.

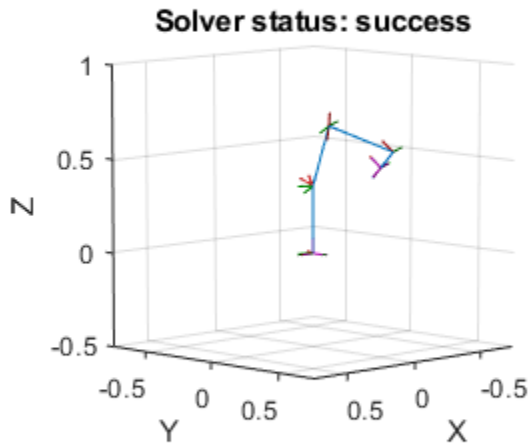
```
posTgt = robotics.PositionTarget('tool0');
posTgt.TargetPosition = [0.0 0.5 0.5];

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

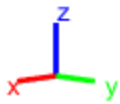
Find a configuration that satisfies the constraints. You must pass the constraint objects into the System object in the order in which they were specified in the `ConstraintInputs` property. Specify an initial guess at the robot configuration.

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

Visualize the configuration returned by the solver.

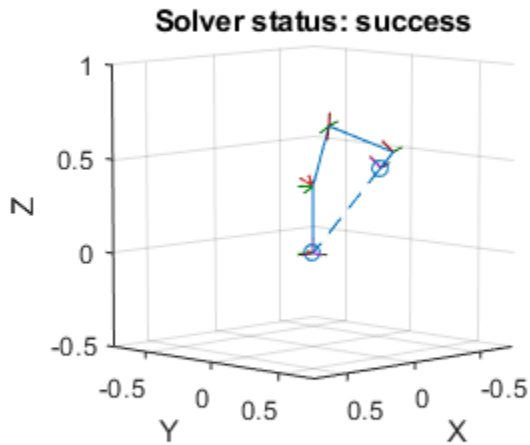
```
show(lbr,q);
title(['Solver status: ' solutionInfo.Status])
axis([-0.75 0.75 -0.75 0.75 -0.5 1])
```

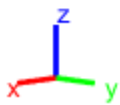




Plot a line segment from the target position to the origin of the base. The origin of the `tool0` frame coincides with one end of the segment, and its `z`-axis is aligned with the segment.

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





- Å òControl PR2 Arm Movements Using ROS Actions and Inverse Kinematicsó
- Å òPlan a Reaching Trajectory With Multiple Kinematic Constraintsó

See Also

Classes

`AimingConstraint` | `CartesianBounds` | `InverseKinematics` | `JointPositionBounds` | `OrientationTarget` | `PoseTarget` | `PositionTarget`

Topics

òControl PR2 Arm Movements Using ROS Actions and Inverse Kinematicsó

òPlan a Reaching Trajectory With Multiple Kinematic Constraintsó

Introduced in R2017a

step

System object: robotics.InverseKinematics
Package: robotics

Joint configurations for desired end-effector pose

Syntax

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

Description

Note Alternatively, 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.

`[configSol,solInfo] = step(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

`ik` مم **Inverse kinematics solver**
`InverseKinematics` object

Inverse kinematics solver, specified as an `InverseKinematics` object.

`endeffector` مم **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 `ik`.

`pose` ᵐ **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 `endeffector`.

`weights` ᵐ **Weight for pose tolerances**
six-element vector

Weight for pose tolerances, specified as a six-element vector. The first three elements correspond to error in orientation. The last three elements correspond to the error of the *xyz* positions.

`initialguess` ᵐ **Initial guess of robot configuration**
structure array

Initial guess of robot configuration, specified as a structure array. 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.

Output Arguments

`configSol` ᵐ **Robot configuration solution**
structure array

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

Å `JointName` ñ Character vector for the name of the joint specified in the `RigidBodyTree` robot model

Å `JointPosition` ñ Position of the corresponding joint

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

`solInfo` ᵐ **Solution information**
structure

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

Å `Iterations` ñ Number of iterations run by the algorithm.

Å `NumRandomRestarts` ñ Number of random restarts because algorithm got stuck in a local minimum.

Å `PoseErrorNorm` ñ The magnitude of the pose error for the solution compared to the desired end effector pose.

Å `ExitFlag` ñ Code that gives more details on the algorithm execution and what caused it to return. For the exit flags of each algorithm type, see òExit Flagsó.

Å `Status` ñ Character vector describing whether the solution is within the tolerance (`'success'`) or the best possible solution the algorithm could find (`'best available'`).

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.

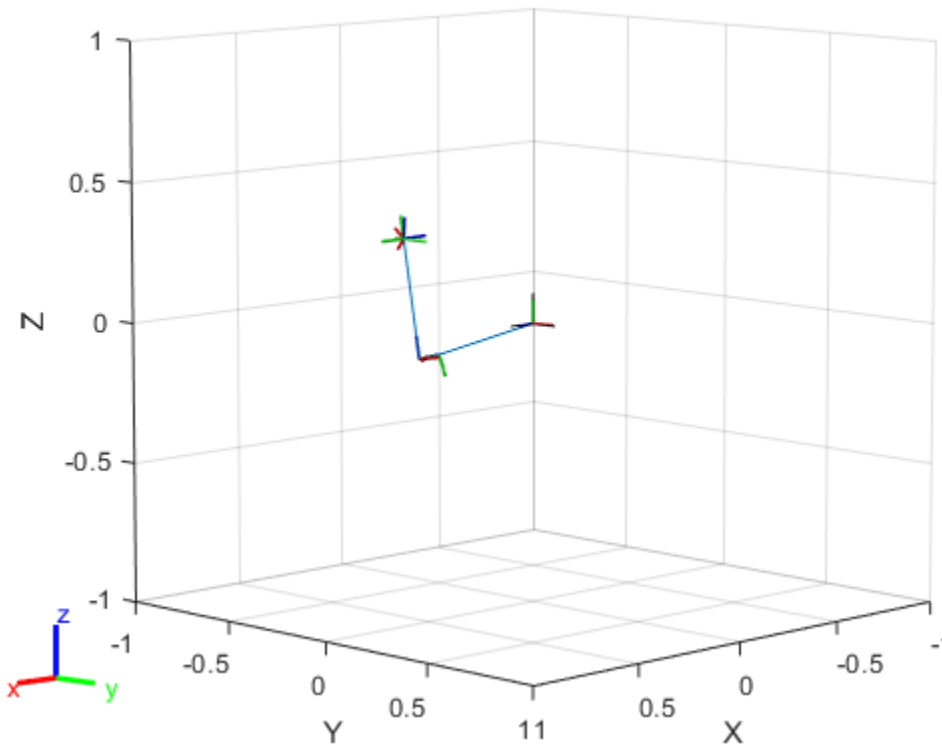
```
load exampleRobots.mat
showdetails(puma1)
```

```
--------------------
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 tranformation 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.

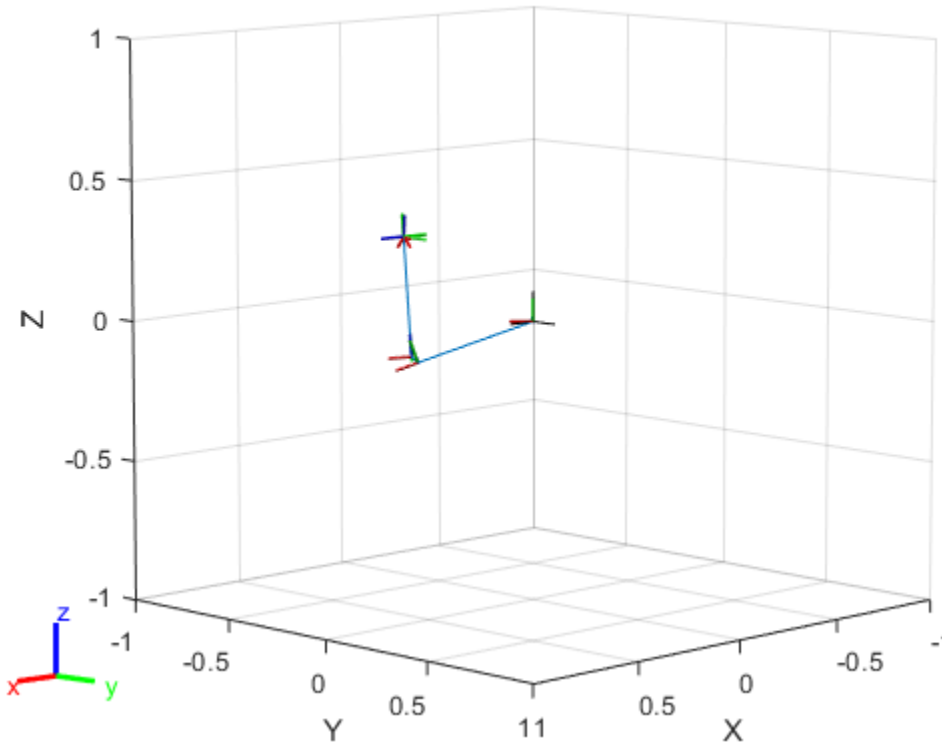
```
ik = robotics.InverseKinematics('RigidBodyTree',puma1);
weights = [0.25 0.25 0.25 1 1 1];
initialguess = puma1.homeConfiguration;
```

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);
```



Å òControl PR2 Arm Movements Using ROS Actions and Inverse Kinematicsó

See Also

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

Topics

òControl PR2 Arm Movements Using ROS Actions and Inverse Kinematicsó
òInverse Kinematics Algorithmsó

Introduced in R2016b

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` مم `Joint` object
handle

`Joint` object, specified as a handle. Create a joint object using `robotics.Joint`.

Output Arguments

`jCopy` مم `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 depedent on the joint configuration. For more information, see òRigid Body Tree Robot Modeló.

`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` مم `Joint` object
handle

`Joint` object, specified as a handle. Create a joint object using `robotics.Joint`.

`tform` مم 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` مم 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 depedent on the joint configuration. For more information, see òRigid Body Tree Robot Modeló.

`mdhparams` مم 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 depedent on the joint configuration. For more information, see òRigid Body Tree Robot Modeló.

Examples

Build Manipulator Robot Using Denavit-Hartenberg Parameters

Use the Denavit-Hartenberg (DH) parameters of the Puma560È 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.

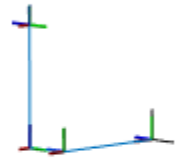
```
body2 = robotics.RigidBody('body2');
jnt2 = robotics.Joint('jnt2','revolute');
body3 = robotics.RigidBody('body3');
jnt3 = robotics.Joint('jnt3','revolute');
body4 = robotics.RigidBody('body4');
jnt4 = robotics.Joint('jnt4','revolute');
body5 = robotics.RigidBody('body5');
jnt5 = robotics.Joint('jnt5','revolute');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

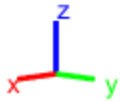
Verify that your robot was built properly by using the `showdetails` or `show` function. `showdetails` lists all the bodies in the MATLABÈ command window. `show` displays the robot with a given configuration (home by default). Calls to `axis` modify the axis limits and hide the axis labels.

```
showdetails(robot)

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

```
--------------------
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)
--------------------
```





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

reset

System object: robotics.MonteCarloLocalization
Package: robotics

Reinitialize `MonteCarloLocalization` object

Syntax

```
reset(mcl)
```

Description

`reset(mcl)` reinitializes the Monte Carlo localization object, `mcl`, to the initial values.

Input Arguments

`mcl` — **`MonteCarloLocalization` object**
handle

`robotics.MonteCarloLocalization` object, specified as an object handle.

Examples

Reset 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);
```

Reset the MCL object to reinitialize states.

```
reset(mcl)
```

See Also

`robotics.MonteCarloLocalization` | `robotics.MonteCarloLocalization.getParticles` | `robotics.MonteCarloLocalization.step`

Topics

òMonte Carlo Localization Algorithmó

Introduced in R2016a

step

System object: robotics.MonteCarloLocalization
Package: robotics

Estimate robot pose and covariance using range data

Syntax

```
[isUpdated,pose,covariance] = step(mcl,odomPose,ranges,angles)
```

Description

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.

`[isUpdated,pose,covariance] = step(mcl,odomPose,ranges,angles)` estimates the pose and covariance of a robot using the Monte Carlo localization (MCL) algorithm. The estimates are based on the pose calculated from the specified robot odometry, `odomPose`, and the specified range sensor data, `ranges` and `angles`. `mcl` is the `robotics.MonteCarloLocalization` object. `isUpdated` indicates whether the estimate is updated based on the `UpdateThreshold` property.

Input Arguments

`mcl` — `MonteCarloLocalization` object
handle

`robotics.MonteCarloLocalization` object, specified as an object handle.

`odomPose` — Pose based on odometry
three-element vector

Posed based on odometry, specified as a three-element vector, `[x y theta]`. This pose is calculated by integrating the odometry over time.

`ranges` مم Range values from scan data
vector

Range values from scan data, specified as a vector in meters. These range values are distances from a laser scan sensor at the specified `angles`. The `ranges` vector must be the same length as the corresponding `angles` vector.

`angles` مم Angle values from scan data
vector

Angle values from scan data, specified as a vector in radians. These angle values are the specific angles from a laser scan sensor of the specified `ranges`. The `angles` vector must be the same length as the corresponding `ranges` vector.

Output Arguments

`isUpdated` مم 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 returned as `true`. Otherwise, it is `false`. The `true` output means that updated pose and covariance are returned. The `false` output means that pose and covariance are not updated and are the same as at the last update.

`pose` مم 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` مم 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.

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 =
```

```
    0.0343    0.0193    0.0331


covariance =

    0.9467    0.0048         0
    0.0048    0.9025         0
         0         0    1.0011
```

Å òLocalize TurtleBot Using Monte Carlo Localizationó

See Also

robotics.MonteCarloLocalization | robotics.MonteCarloLocalization.getParticles

Topics

òLocalize TurtleBot Using Monte Carlo Localizationó
òMonte Carlo Localization Algorithmó

Introduced in R2016a

checkOccupancy

Class: robotics.OccupancyGrid
Package: robotics

Check locations for free, occupied, or unknown values

Syntax

```
iOccval = checkOccupancy(map,xy)
iOccval = checkOccupancy(map,ij,'grid')
```

Description

`iOccval = 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 (ð1).

`iOccval = checkOccupancy(map,ij,'grid')` specifies `ij` grid cell indices instead of `xy` locations.

Input Arguments

`map` مم **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` مم **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` مم 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` مم 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 (ð1). 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])
occupied = checkOccupancy(map,[1.5 1])

pocc2 = getOccupancy(map,[5 5],'grid')
occupied2 = checkOccupancy(map,[5 5],'grid')
```

```
pocc =

    0.7500


occupied =

     1


pocc2 =

    0.5000


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` مم 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` مم 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

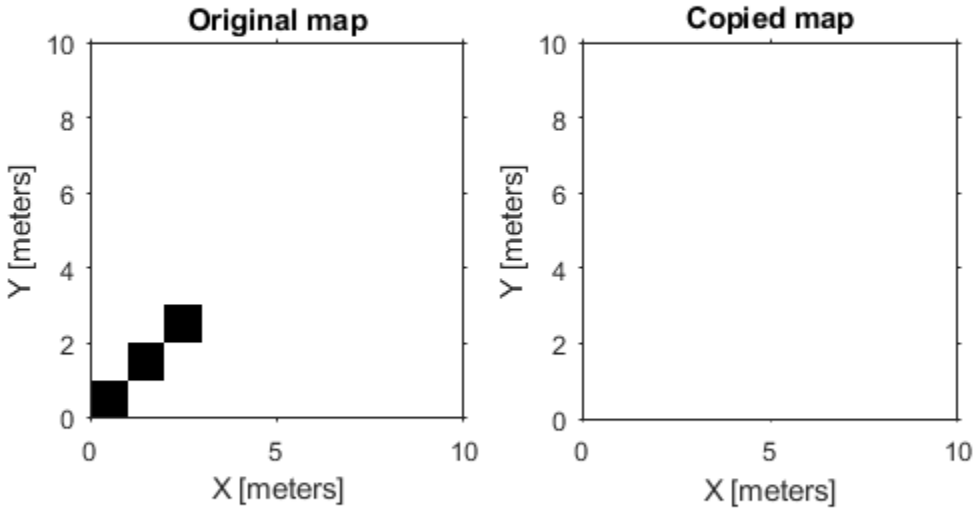
Copy an occupancy grid object. Once copied, the original object can be modified without affecting the copied map.

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

`robotics.BinaryOccupancyGrid` | `robotics.OccupancyGrid` | `robotics.OccupancyGrid.getOccupancy` | `robotics.OccupancyGrid.occupancyMatrix`

Topics

òOccupancy Gridsó

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` مم 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` مم 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])
occupied = checkOccupancy(map,[1.5 1])

pocc2 = getOccupancy(map,[5 5],'grid')
occupied2 = checkOccupancy(map,[5 5],'grid')
```

```
pocc =

    0.7500


occupied =

     1


pocc2 =

    0.5000


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.

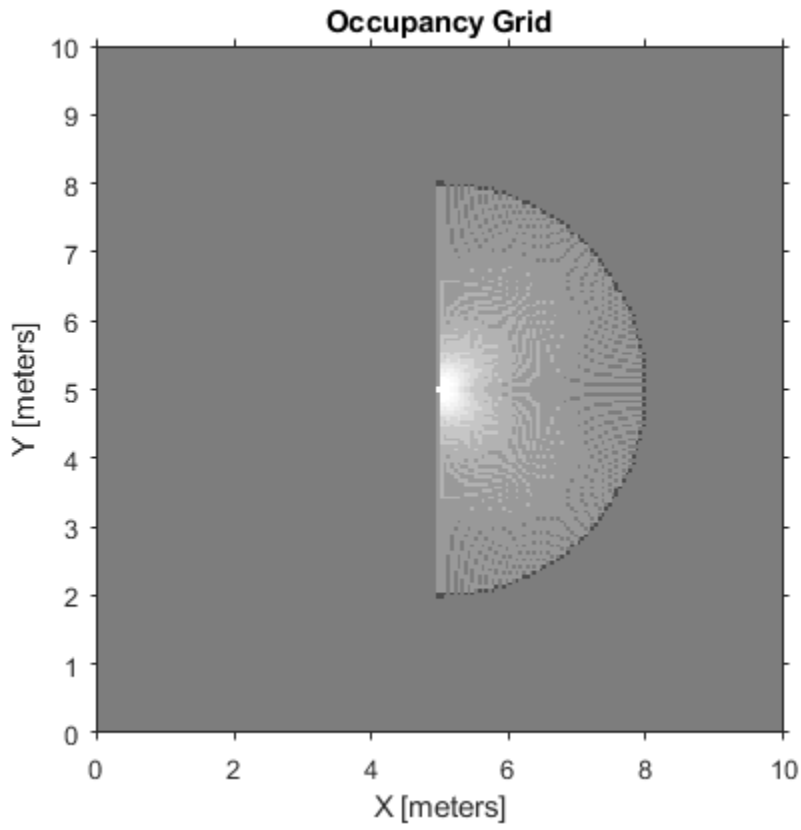
```
pose = [5,5,0];
ranges = 3*ones(100, 1);
angles = linspace(-pi/2, pi/2, 100);
maxrange = 20;
```

```
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])
```

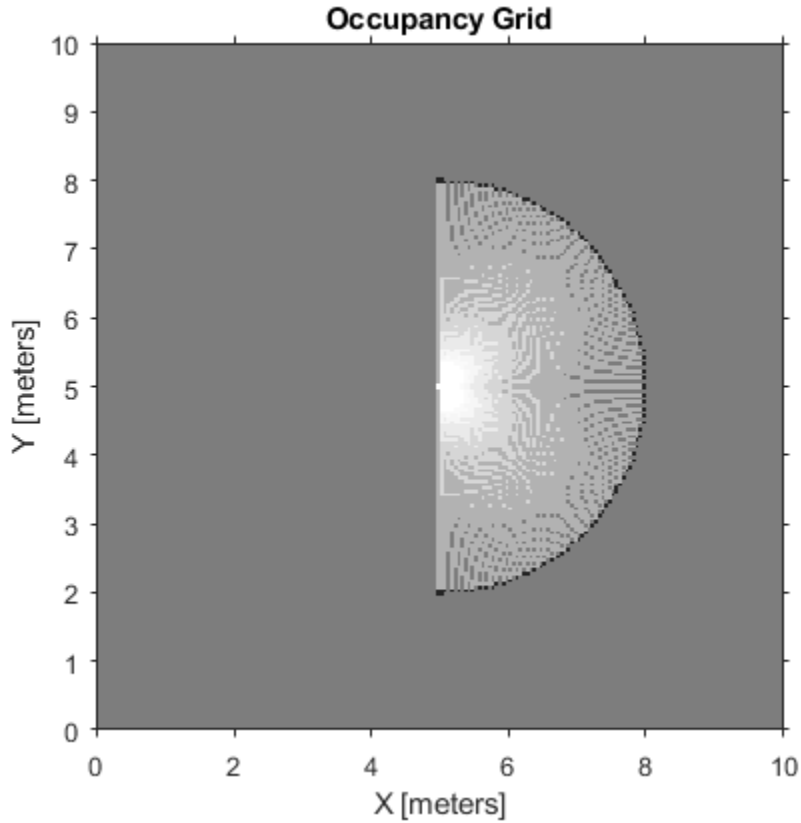
```
ans =

    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 Ñ0.001. 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 òOccupancy Gridsó.

See Also

`robotics.BinaryOccupancyGrid` | `robotics.OccupancyGrid` | `robotics.OccupancyGrid.checkOccupancy`

Topics

òOccupancy Gridsó

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` مم 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` مم 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 [illegible] 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

òOccupancy Gridsó

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` مم 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

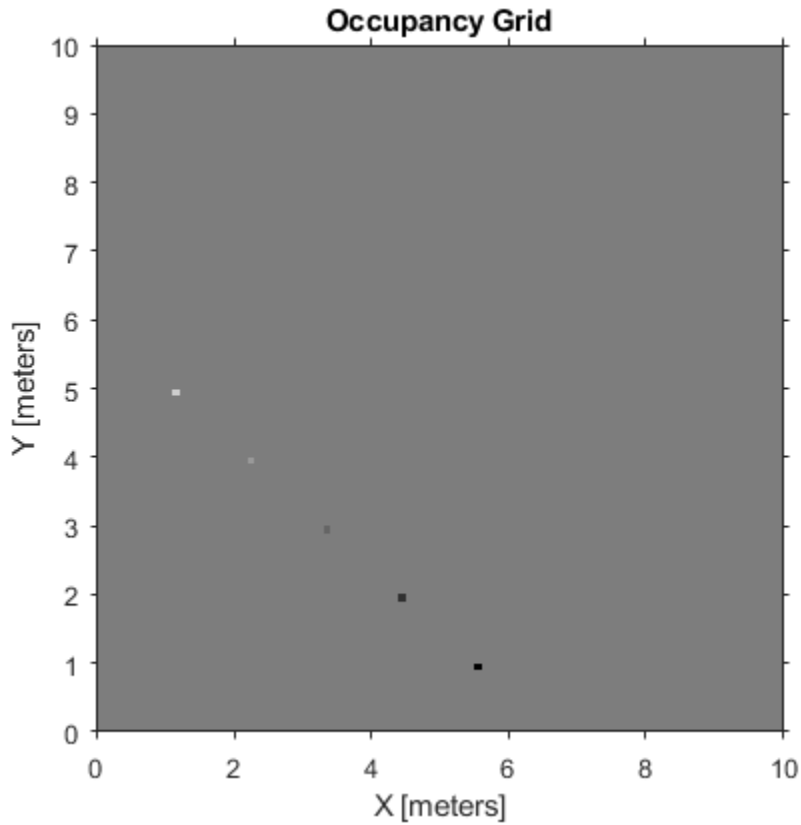
Create and Modify Occupancy Grid

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

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

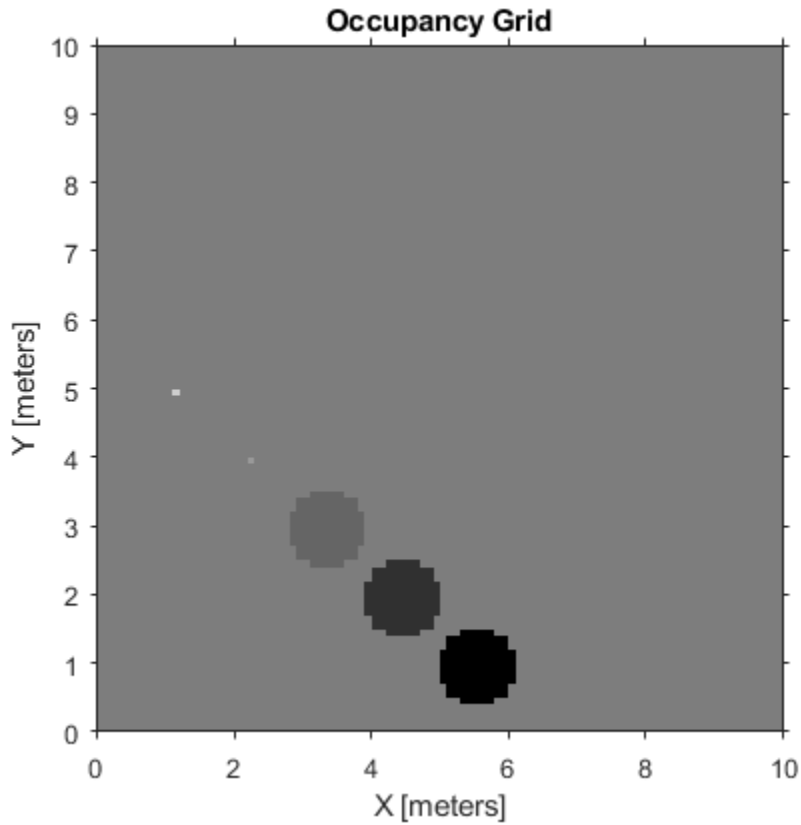
Update the occupancy of world locations with specific probability values.

```
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)
```

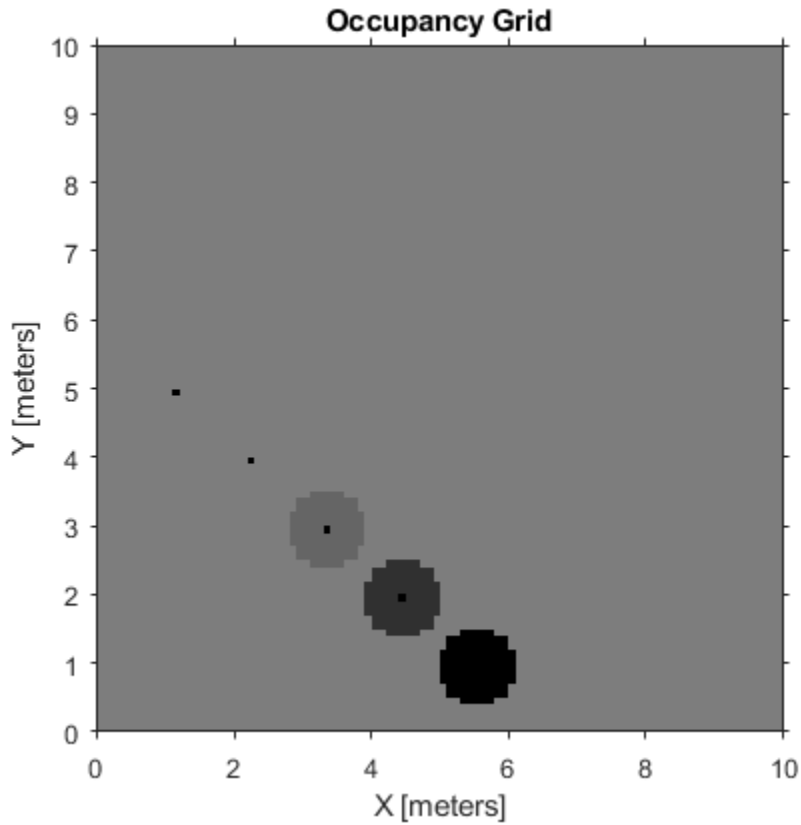


Get grid locations from world locations.

```
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:

$(A \quad B)(x, y) = \max \{A(x$ ð $x\dot{l}, y\dot{o}$ ð $y\dot{l}) + B(x', y') \mid (x\dot{l}, y\dot{l})$ نمِ $D_B\}$.

D_B is the domain of the probability values in the structuring element B. $A(x,y)$ is assumed to be +Ð 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

òOccupancy Gridsó

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 òInverse Sensor Modeló on page 3-71.

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);
```

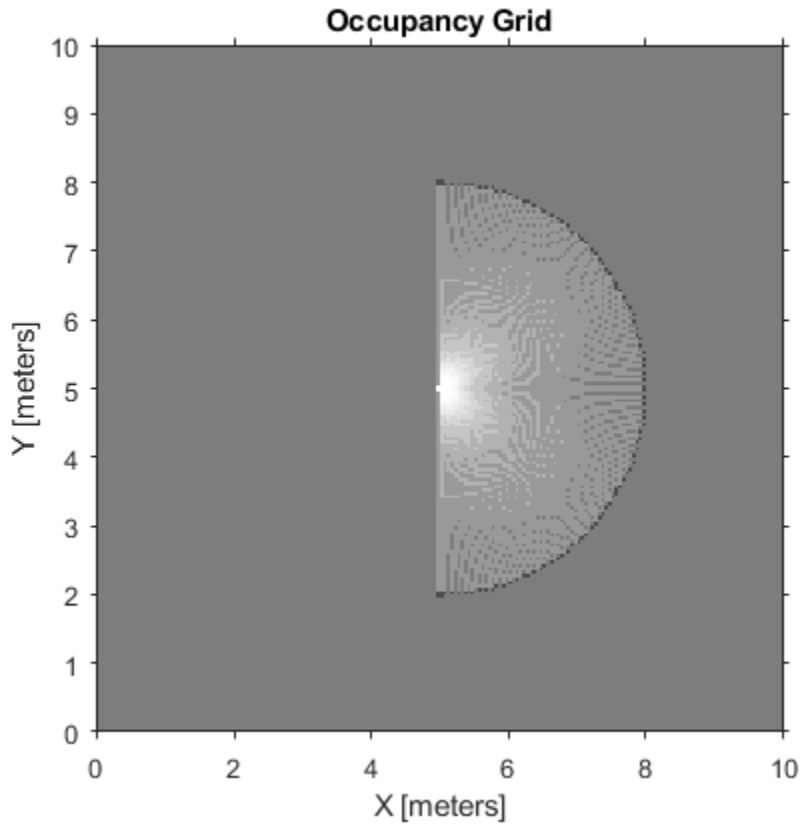
Insert a laser scan into the occupancy grid. Specify the pose of the robot ranges and angles and the max range of the laser scan.

```
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])
```

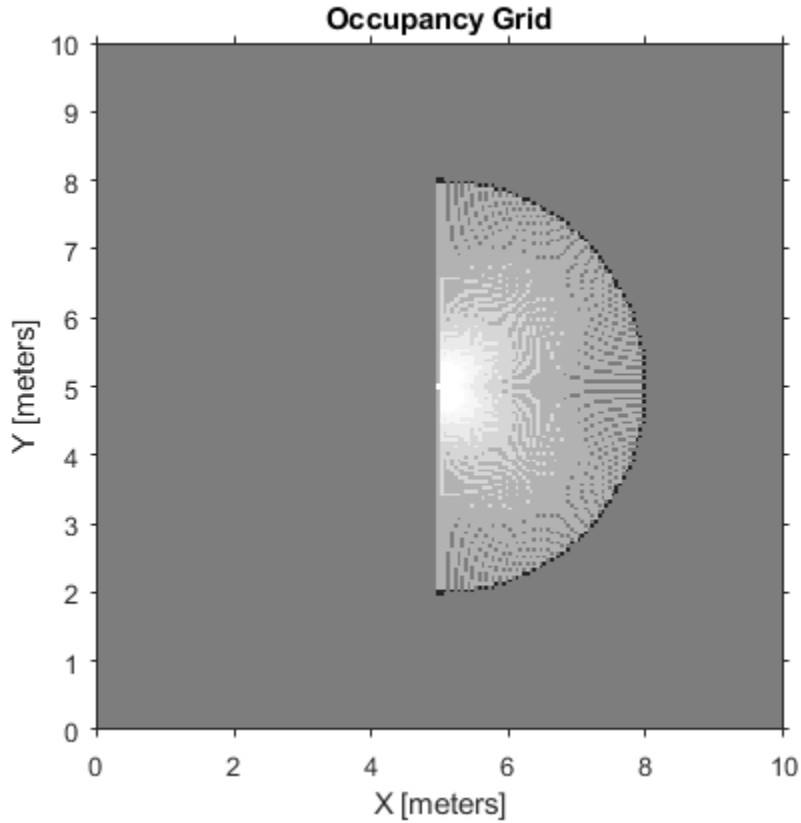
```
ans =

    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])
```

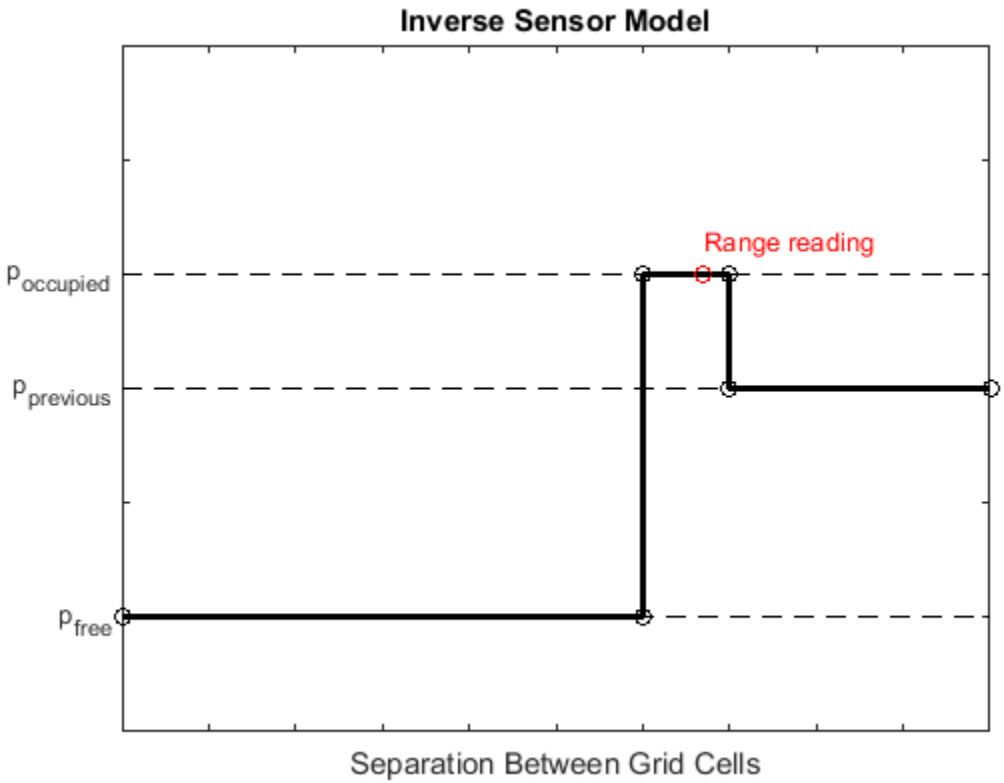
```
ans =

    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

òOccupancy Gridsó

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 ð1.

Input Arguments

map مم 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

òOccupancy Gridsó

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.

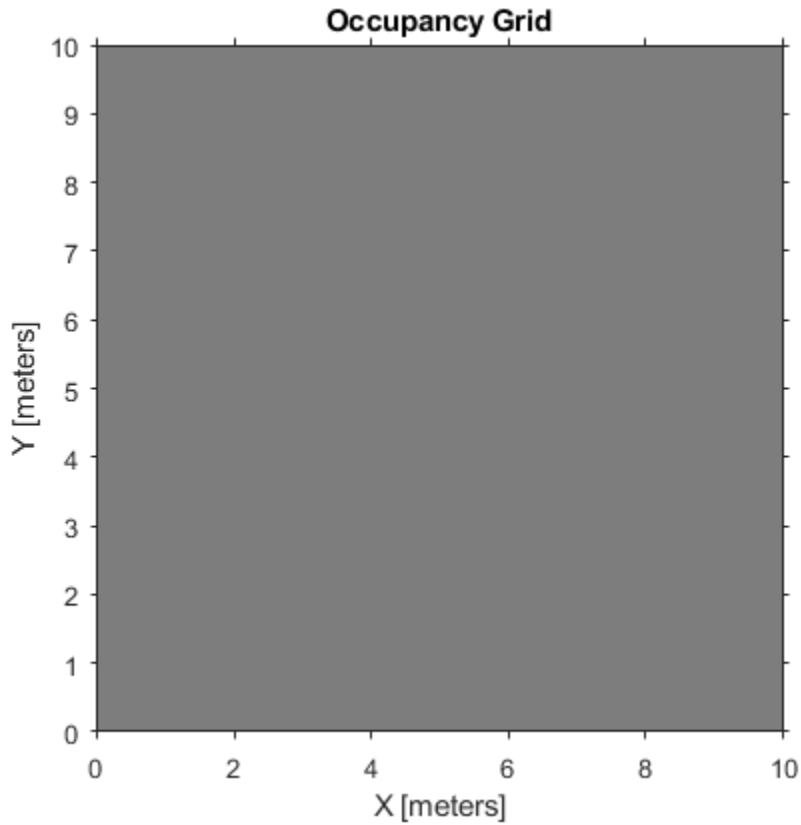
Examples

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 `p1` to `p2`. 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.