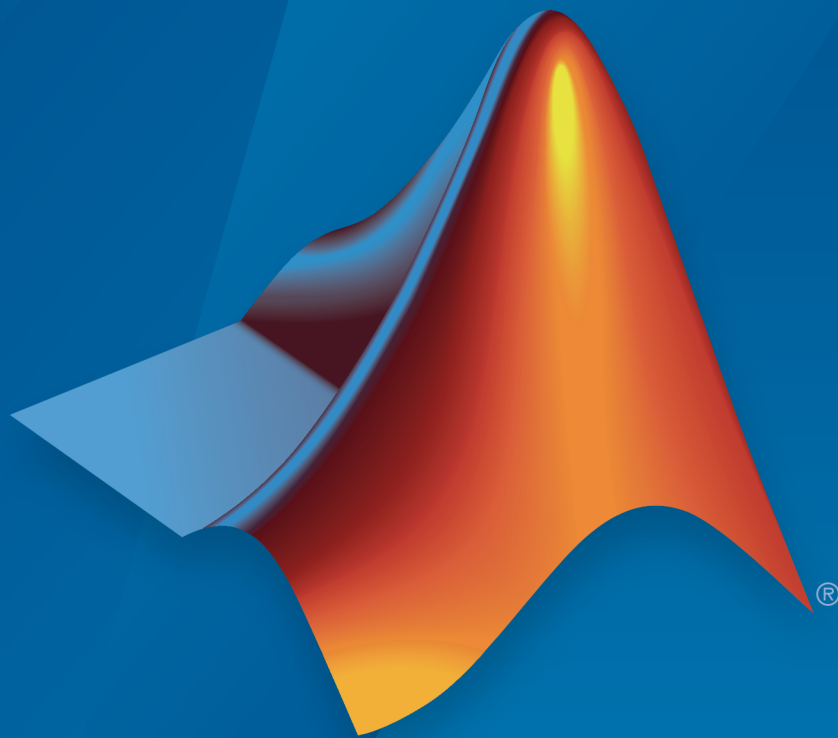


Robotics System Toolbox™

Reference



MATLAB® & SIMULINK®

R2016b



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

© COPYRIGHT 2015–2016 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)

1 | Classes — Alphabetical List

2 | Functions — Alphabetical List

3 | Methods — Alphabetical List

4 | Blocks — Alphabetical List

Classes — 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.

Create Object

`rosbag`
`select`

Open and parse rosbag log file
Select subset of messages in rosbag

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 `rosviz`, this property contains the number of messages in the `rosviz`. 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	MessageDefinition
<code>/odom</code>	99	<code>nav_msgs/Odometry</code>	<code>'# This represents an estimate of a pose'</code>

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 rosviz

select
timeseries

Select subset of messages in rosbag
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_mult...
```

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');
```

- “Work with rosbag Logfiles”

See Also

`readMessages` | `select` | `timeseries`

More About

- “ROS Log Files (rosbags)”

Introduced in R2015a

Core

Create ROS Core

Description

The ROS Core encompasses many key components and nodes that are essential for the ROS network. You must have exactly one ROS core running in the ROS network in order 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`.

Create Object

`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;
```

Create ROS Core on Specific Port

Create ROS Core on localhost and port 12000.

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

- “Connect to a ROS Network”

See Also

Node | `roscpp`

More About

- “ROS Network Setup”

External Websites

- ROS Core

Introduced in R2015a

CompressedImage

Create compressed image message

Description

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

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

Create Object

`msg = rosmesssage('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: [1×1 Header]
      Format: 'bgr8; jpeg compressed bgr8'
      Data: [30376×1 uint8]
```

Use `showdetails` to show the contents of the message

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

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



Create Blank Compressed Image Message

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

```
compImg =
```

```
ROS CompressedImage message with properties:
```

```
  MessageType: 'sensor_msgs/CompressedImage'  
    Header: [1×1 Header]  
    Format: ''  
    Data: [0×1 uint8]
```

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

- “Work with Specialized ROS Messages”

See Also

[readImage](#) | [rosmessage](#) | [rossubscriber](#)

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.

Create Object

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

Properties

MessageType — Message type of ROS message

character vector

This property is read only.

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

Data Types: `char`

Header — ROS Header message

Header object

This property is read only.

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

Height — Image height in pixels

scalar

Image height in pixels, specified as a scalar.

Width — Image width in pixels

scalar

Image width in pixels, specified as a scalar.

Encoding — Image encoding

character vector

Image encoding, specified as a character vector.

Example: 'rgb8'

IsBigendian — Image byte sequence

true | false

Image byte sequence, specified as a `true` or `false`.

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

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

Step — Full row length in bytes

integer

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

Data — Image data

uint8 array

Image data, specified as a `uint8` array.

Object Functions

`readImage`

Convert ROS image data into MATLAB
image

writeImage

Write MATLAB image to ROS image message

Examples

Read and Write Image Messages

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

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

```
exampleHelperROSLoadMessages
img
```

```
img =
```

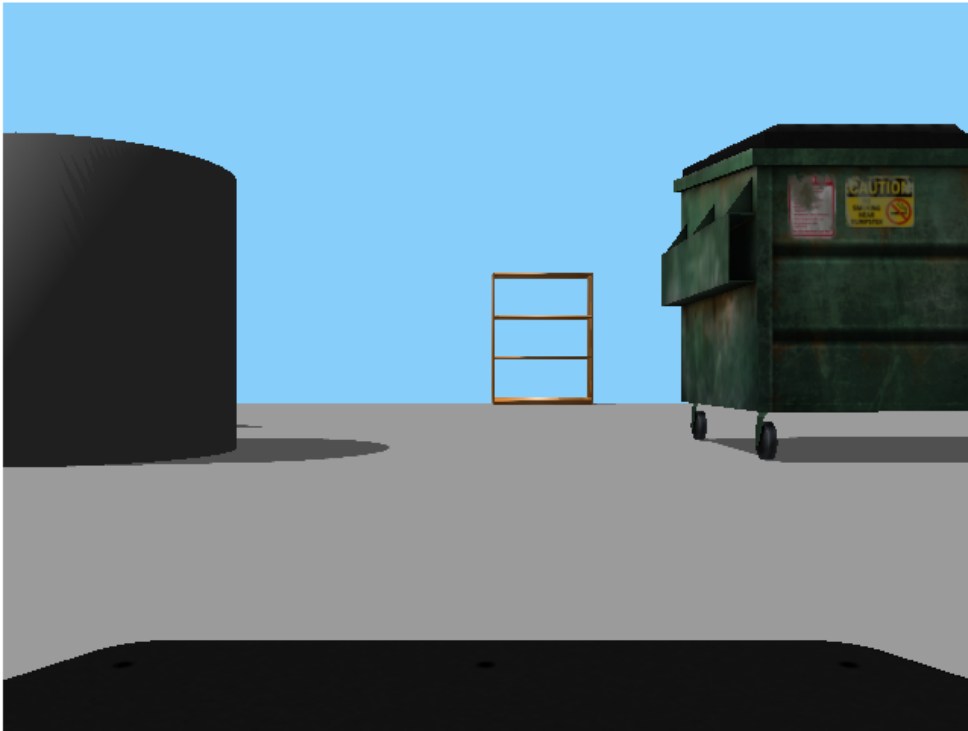
```
ROS Image message with properties:
```

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

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

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

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



Create a ROS Image message from a MATLAB image.

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

```
imgMsg =
```

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

```
  MessageType: 'sensor_msgs/Image'
```

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

Use `showdetails` to show the contents of the message

- “Work with Specialized ROS Messages”

See Also

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

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`.

Create Object

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

Properties

MessageType — Message type of ROS message

character vector

This property is read only.

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

Data Types: `char`

Header — ROS Header message

Header object

This property is read only.

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

AngleMin — Minimum angle of range data

scalar

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

AngleMax — Maximum angle of range data

scalar

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

AngleIncrement — Angle increment of range data

scalar

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

TimeIncrement — Time between individual range data points in seconds

scalar

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

ScanTime — Time to complete a full scan in seconds

scalar

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

RangeMin — Minimum valid range value

scalar

Minimum valid range value, specified as a scalar.

RangeMax — Maximum valid range value

scalar

Maximum valid range value, specified as a scalar.

Ranges — Range readings from laser scan

vector

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

Intensities — Intensity values from range readings

vector

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

Object Functions

plot	Display ROS laser scan messages on custom plot
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: [640×1 single]
Intensities: [0×1 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.

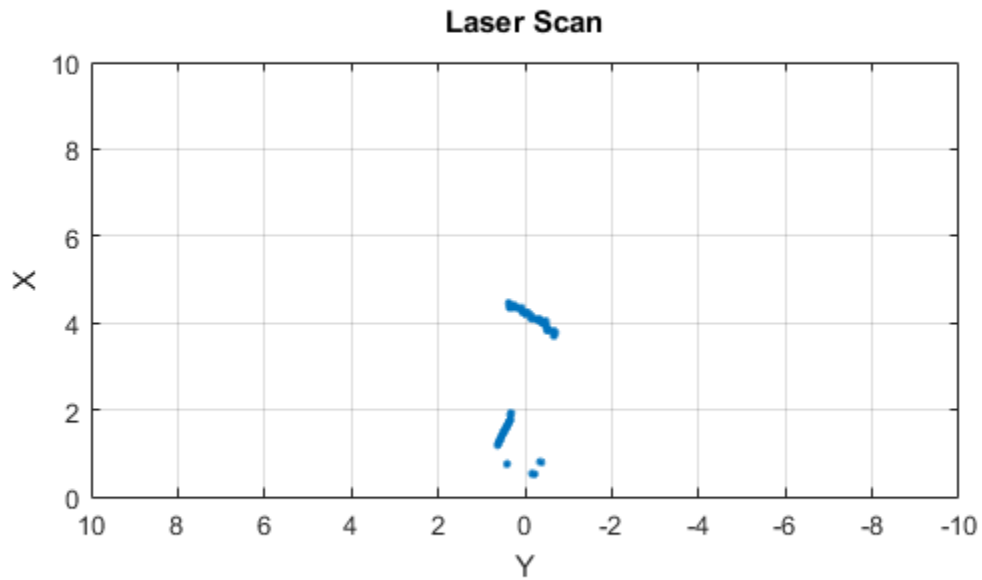
```
ranges = scan.Ranges;
angles = scan.readScanAngles;
size(ranges)
size(angles)
```

```
ans =
    640     1
```

```
ans =
    640     1
```

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

```
plot(scan)
```

Create Empty LaserScan Message

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

```
scan =
```

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

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

```
TimeIncrement: 0
  ScanTime: 0
  RangeMin: 0
  RangeMax: 0
  Ranges: [0×1 single]
  Intensities: [0×1 single]
```

Use `showdetails` to show the contents of the message

- “Work with Specialized ROS Messages”

See Also

`plot` | `readCartesian` | `readScanAngles` | `rosmessage` | `rossubscriber` | `showdetails`

Introduced in R2016a

Node

Start ROS node and connect to ROS master

Description

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

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

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

Create Object

`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 `rostopic`.

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

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

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

```
ans =
```

```
ROS String message with properties:
```

```
  MessageType: 'std_msgs/String'
           Data: 'Message from Node 1'
```

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

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

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

Connect to Multiple ROS Masters

Connecting to multiple ROS masters is possible using MATLAB. Typically, this need arises when you try to connect to multiple robots at the same time. When you connect to multiple ROS masters, you are connecting to two different ROS networks, each with their own set of nodes and topics.

MATLAB can connect to multiple ROS masters by creating multiple nodes. For example, assume that you have two robots that are wirelessly connected to your computer running MATLAB. The first robot has IP address 192.168.1.1 and the second robot has IP address 192.168.1.2.

Create one node connecting to the first robot and one node connecting to the second robot.

```
node1 = robotics.ros.Node('/test_node_1', '192.168.1.1');  
node2 = robotics.ros.Node('/test_node_2', '192.168.1.2');
```

You can now create subscribers, publishers, and other ROS entities and attach them to each node. For

Create a subscriber for a topic on robot 1.

```
sub = robotics.ros.Subscriber(node1, '/topic_robot1');
```

Create a publisher and a message to send with this publisher.

```
pub = robotics.ros.Publisher(node2, '/topic_robot2');  
msg = rosmesssage(pub);
```

The same steps apply for service clients, service servers, parameter trees, and transformation tree objects. You can now write MATLAB code that interacts with both robots at the same time.

See Also

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

More About

- “ROS Network Setup”

External Websites

- [ROS Nodes](#)

Introduced in R2015a

OccupancyGrid

Create occupancy grid message

Description

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

Create Object

`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 — 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 = rosmesssage('nav_msgs/OccupancyGrid');  
writeBinaryOccupancyGrid(mapMsg,bogMap)  
mapMsg
```

```
mapMsg =
```

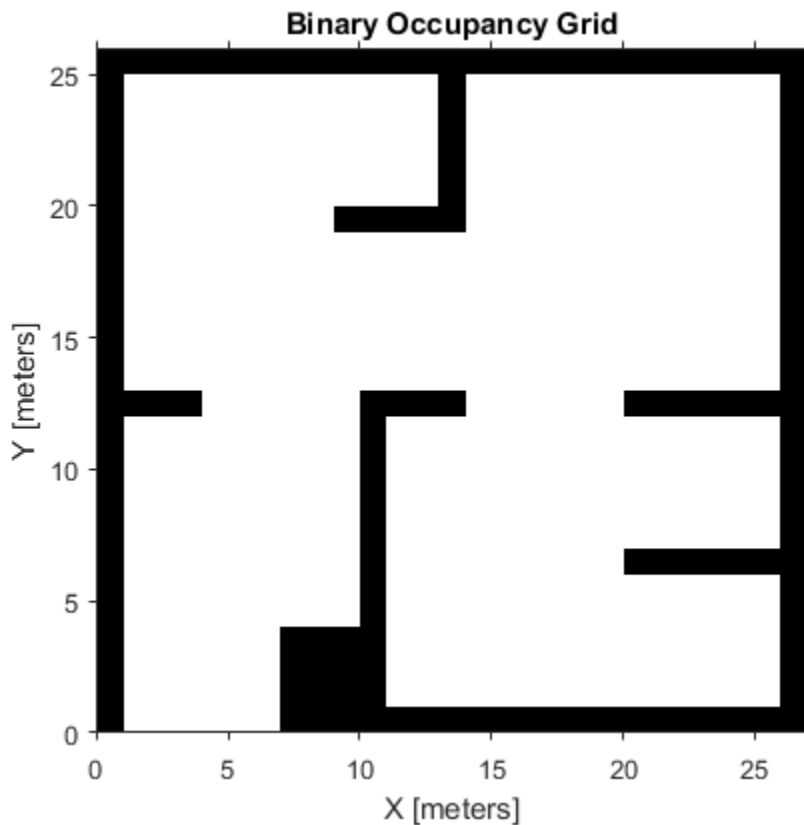
```
ROS OccupancyGrid message with properties:
```

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

[robotics.BinaryOccupancyGrid](#) | [readBinaryOccupancyGrid](#) | [rosmmessage](#) | [rossubscriber](#) | [writeBinaryOccupancyGrid](#)

More About

- “Occupancy Grids”

Introduced in R2015a

ParameterTree

Access ROS parameter server

Description

A `robotics.ros.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.

Supported parameter values are:

- `int32`
- `logical`
- `double`
- character vector
- cell array

Create Object

`ptree = robotics.ros.ParameterTree(node)` returns a `ParameterTree` object to communicate with the ROS parameter server. The parameter tree attaches to the ROS node, `node`. To connect to the global node, specify `node` as `[]`.

Properties

AvailableParameters — List of parameter names on the server

cell array

This property is read only.

List of parameter names on the server, specified as a cell array.

Example: `{ '/myParam'; '/robotSize'; '/hostname' }`

Data Types: `cell`

Object Functions

<code>get</code>	Get ROS parameter value
<code>has</code>	Check if ROS parameter name exists
<code>search</code>	Search ROS network for parameter names
<code>set</code>	Set value of ROS parameter; add new parameter
<code>del</code>	Delete a ROS parameter

Examples

Create ROS ParameterTree Object and Modify Parameters

Start the ROS master and create a ROS node.

```
master = robotics.ros.Core;  
node = robotics.ros.Node('/test1');
```

Create the parameter tree object.

```
ptree = robotics.ros.ParameterTree(node);
```

Set multiple parameters.

```
set(ptree, 'DoubleParam', 1.0)  
set(ptree, 'CharParam', 'test')  
set(ptree, 'CellParam', {'test'}, {1,2});
```

View the available parameters.

```
parameters = ptree.AvailableParameters
```

```
parameters =
```

```
3×1 cell array
```

```
  '/CellParam'  
  '/CharParam'  
  '/DoubleParam'
```

Get a parameter value.

```
data = get(ptree, 'CellParam')
```

```
data =
```

```
    1×2 cell array  
    {1×1 cell}    {1×2 cell}
```

Search for a parameter name.

```
search(ptree, 'char')
```

```
ans =
```

```
    cell  
    '/CharParam'
```

Delete the parameter tree and ROS node. Shut down the ROS master.

```
clear('ptree', 'node')  
clear('master')
```

- “Access the ROS Parameter Server”

See Also

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

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.

Create Object

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

Properties

PreserveStructureOnRead — Preserve the shape of point cloud matrix

`false` (default) | `true`

This property is read only.

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

MessageType — Message type of ROS message

character vector

This property is read only.

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

Data Types: `char`

Header — ROS Header message

Header object

This property is read only.

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

Height — Point cloud height in pixels

integer

Point cloud height in pixels, specified as an integer.

Width — Point cloud width in pixels

integer

Point cloud width in pixels, specified as an integer.

IsBigendian — Image byte sequence

true | false

Image byte sequence, specified as a `true` or `false`.

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

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

PointStep — Length of a point in bytes

integer

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

RowStep — Full row length in bytes

integer

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

Data — Point cloud data

uint8 array

Point cloud data, specified as a `uint8` array. To access the data, use the “Object Functions” on page 1-36.

Object Functions

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

Examples

Inspect Point Cloud Image

Access and visualize the data inside a point cloud message.

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

```
exampleHelperROSLoadMessages  
ptcloud
```

```
ptcloud =
```

```
ROS PointCloud2 message with properties:
```

```
  PreserveStructureOnRead: 0  
    MessageType: 'sensor_msgs/PointCloud2'  
      Header: [1×1 Header]  
      Height: 480  
      Width: 640  
    IsBigendian: 0  
    PointStep: 32  
    RowStep: 20480  
    IsDense: 0  
    Fields: [4×1 PointField]  
      Data: [9830400×1 uint8]
```

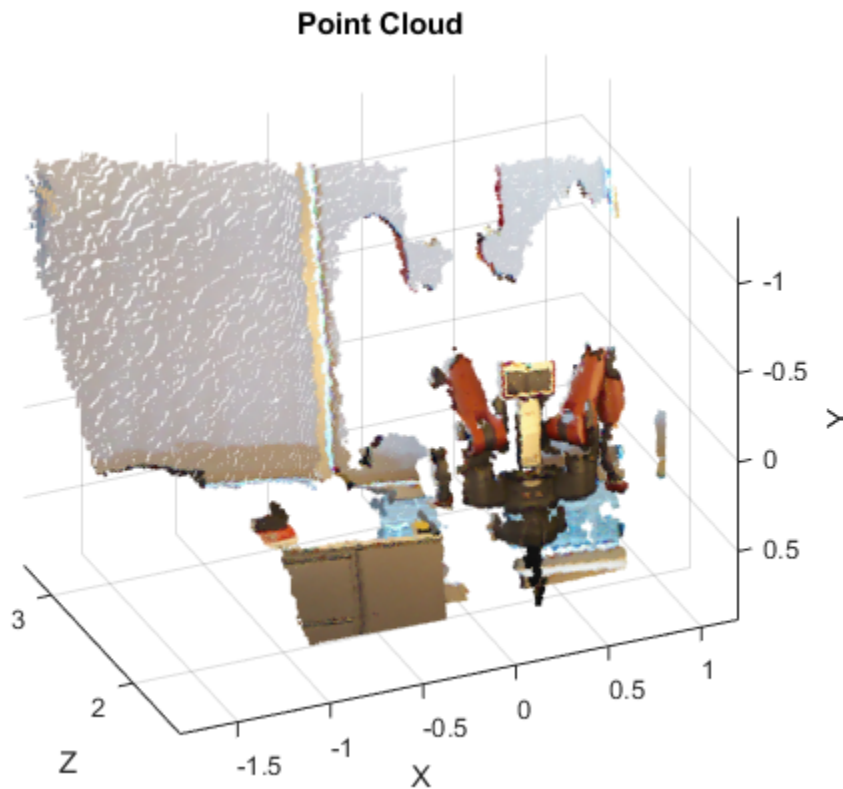
```
Use showdetails to show the contents of the message
```


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

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

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

```
scatter3(ptcloud)
```



Create pointCloud Object Using Point Cloud Message

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

Load sample messages.

`exampleHelperROSLoadMessages`

Convert a `ptcloud` message to the `pointCloud` object.

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

```
pcobj =
```

```
pointCloud with properties:
```

```
Location: [307200x3 single]  
Color: [307200x3 uint8]  
Normal: []  
Count: 307200  
XLimits: [-1.8147 1.1945]  
YLimits: [-1.3714 0.8812]  
ZLimits: [1.4190 3.3410]
```

- “Work with Specialized ROS Messages”

See Also

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

Introduced in R2015a

Publisher

Create ROS publisher

Description

The `Publisher` object represents a publisher on the ROS network. The object publishes to an available topic or to a topic that it creates. This topic has an associated message type. 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.

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.

Create Object

`pub = robotics.ros.Publisher(node, topicname)` creates a publisher for a topic with name, `topicname`. This topic must exist already so the message type can be retrieved. `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. You can use any combination of previous inputs with this syntax.

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
rosmessage

Publish ROS message to topic
Create ROS messages

Examples

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://bat5731win64:11311/.  
Initializing global node /matlab_global_node_03937 with NodeURI http://bat5731win64:60
```

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
```

```
roshutdown
```

```
subCount =
```

```
    1
```

```
Shutting down global node /matlab_global_node_03937 with NodeURI http://bat5731win64:6
Shutting down ROS master on http://bat5731win64:11311/.
```

Use ROS Publisher Object

Create a Publisher object using the class constructor.

Start the ROS master.

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

Create a ROS node, which connects to the master.

```
node = robotics.ros.Node('/test1');
```

Create a publisher and send string data. The publisher attaches to the node object in the first argument.

```
pub = robotics.ros.Publisher(node, '/robotname', 'std_msgs/String');
msg = rosmesssage(rostype.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')
```

- “Exchange Data with ROS Publishers and Subscribers”

See Also

[rosmesssage](#) | [rospublisher](#) | [rossubscriber](#) | [send](#)

Introduced in R2015a

rosdevice

Connect to remote ROS device

Description

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

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

Create Object

`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
stopNode

Start ROS node
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 =
```

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

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

```
Initializing global node /matlab_global_node_14356 with NodeURI http://192.168.154.1:6
```

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

```
d.AvailableNodes
```

```
ans =  
  
1×2 cell array  
  
'robotcontroller'    'robotcontroller2'
```

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

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

```
running =  
  
logical  
  
1
```

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

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

```
Shutting down global node /matlab_global_node_14356 with NodeURI http://192.168.154.1:6
```

Run Multiple ROS Nodes

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

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

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

```
ipaddress = '192.168.154.131';  
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 MATLAB® 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://192.168.154.1:6
```

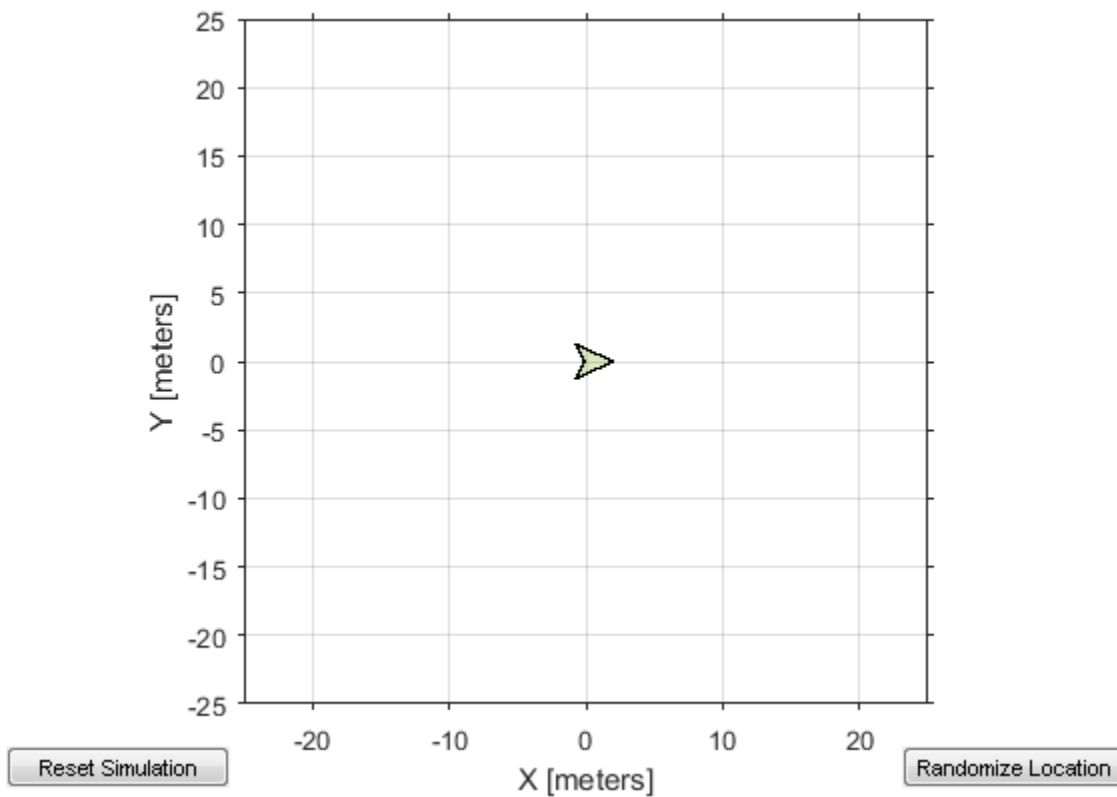
Check the available ROS nodes on the connected ROS device. These nodes were generated from Simulink® models following the process in the [Generate a standalone ROS node from Simulink®](#) example.

d.AvailableNodes

```
ans =  
  
1×2 cell array  
  
    'robotcontroller'    'robotcontroller2'
```

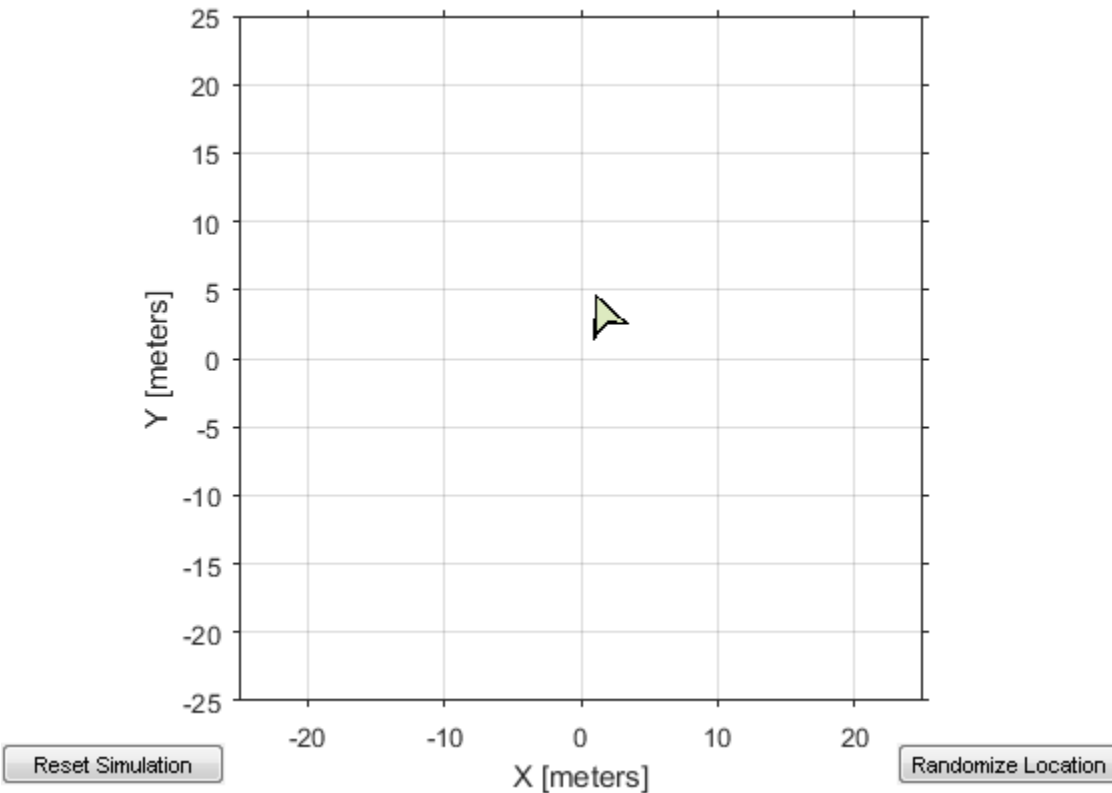
Start up the Robot Simulator using `ExampleHelperSimulinkRobotROS`. This simulator automatically connects to the ROS master on the ROS device. You will use this simulator to run a ROS node and control the robot.

```
sim = ExampleHelperSimulinkRobotROS;
```



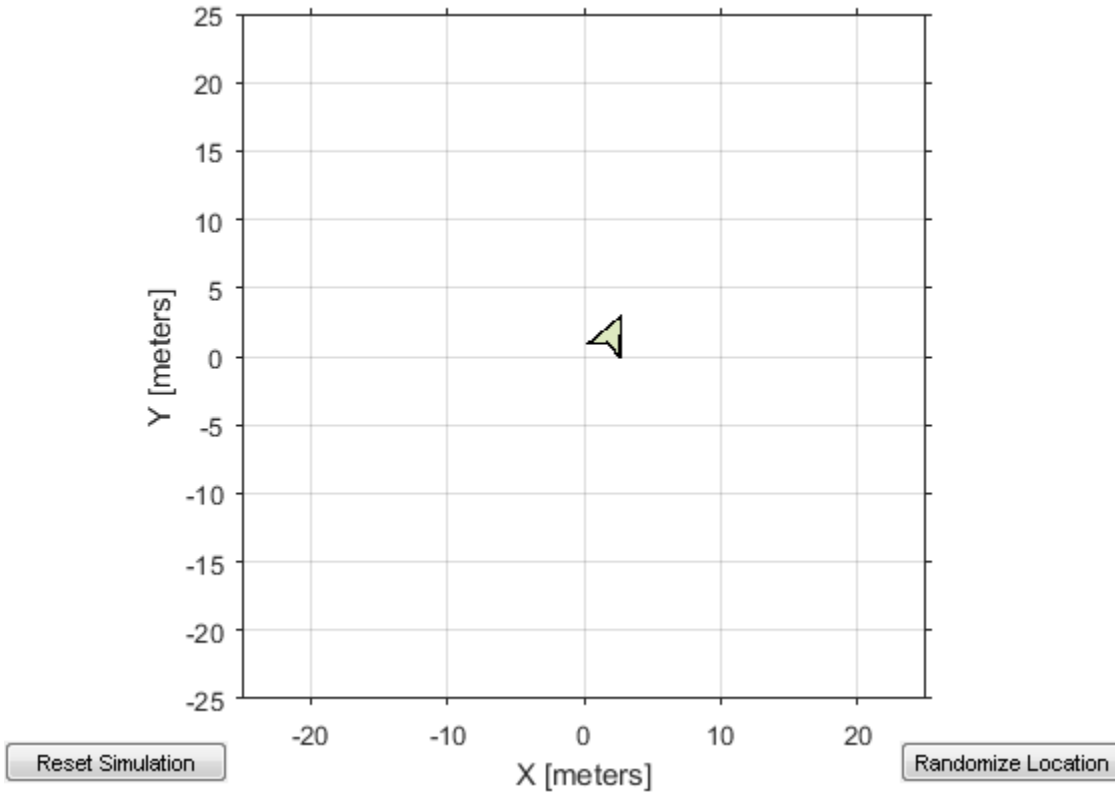
Run a ROS node, specifying the node name. The 'robotcontroller' node commands the robot to a specific location ([-10 10]). Wait to see the robot drive.

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



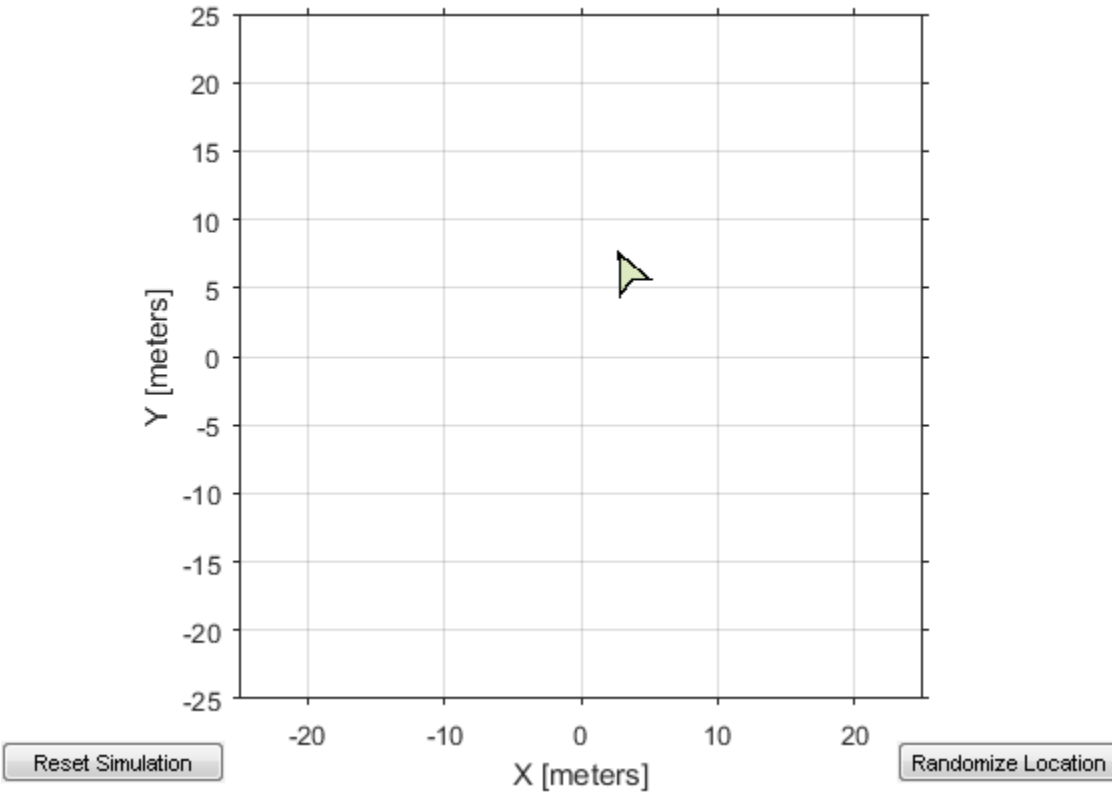
Reset the Robot Simulator to reset the robot position. Alternatively, click **Reset Simulation**. Because the node is still running, the robot continues back to the specific location. To stop sending commands, stop the node.

```
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')
roshutdown
stopCore(d)
```

Shutting down global node /matlab_global_node_68749 with NodeURI http://192.168.154.1:0

- “Generate a standalone ROS node from Simulink®”

See Also

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

Introduced in R2016b

ROS Rate

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.

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 `robotics.Rate.waitfor` in the loop to pause code execution until the next time step. Unless the enclosed code takes longer, the loop operates every `DesiredPeriod` seconds. 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 `sleep` method to wait until the next multiple of `DesiredPeriod` is reached to execute the next loop.

Tip The accuracy of the rate execution is influenced by the scheduling resolution of your operating system and by the level of other system activity. Accurate rate timing is limited to 100 Hz for execution of MATLAB code. However, the use of code generation can improve performance and execution speeds.

Create Object

`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

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

`robotics.Rate.waitFor`

Pause code execution to achieve desired execution rate

`robotics.Rate.statistics`

Statistics of past execution periods

Examples

Run Loop at Fixed Rate Using ROS Time

Initialize the ROS master and node.

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

Create a `ros.Rate` object running at 20 Hz.

```
r = robotics.ros.Rate(node,20);
```

Reset the object to restart the timer and run the loop for 30 iterations. Input user code before calling `sleep`.

```
reset(r)
for i = 1:30
    % User code goes here.
    waitFor(r)
end
```

Shutdown ROS node.

`roshutdown`

See Also

`robotics.Rate.waitFor` | `robotics.Rate.statistics` | `robotics.Rate` | `Node` | `rosclock`

More About

- [Class Attributes](#)
- [Property Attributes](#)

External Websites

- [ROS Clock](#)

Introduced in R2016a

ServiceClient

Connect to ROS service server

Description

Use `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.

Create Object

`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: 'rostype.gazebo_msgs_GetModelState'

Object Functions

rosmessage
call

Create ROS messages
Call the ROS service server and receive a response

Examples

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 MATLAB® path.

Assign the callback function for incoming service calls.

```
server.NewRequestFcn = @flipString;
```

Create a service client and connect to the service server. Create a request message based on the client.

```
client = robotics.ros.ServiceClient(node, '/data/string');
request = rosmesssage(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')
```

- “Call and Provide ROS Services”

See Also

`call` | `rosmesssage` | `rossvcserver`

Introduced in R2015a

ServiceServer

Create ROS service server

Description

Use `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.

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.

Create Object

`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: `'rostype.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

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 MATLAB® path.

Assign the callback function for incoming service calls.

```
server.NewRequestFcn = @flipString;
```

Create a service client and connect to the service server. Create a request message based on the client.

```
client = robotics.ros.ServiceClient(node, '/data/string');
```

```
request = rosmesssage(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')
```

- “Call and Provide ROS Services”

See Also

rossvcclient

Introduced in R2015a

SimpleActionClient

Create ROS action client

Description

Use `SimpleActionClient` objects to connect to an action server 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.

Use `roactionclient` to connect to the action server and create the `SimpleActionClient` object.

Create Object

`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 `roaction type actionname`.

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.

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 `roaction('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:

- `'pending'` — Goal was received, but has not yet been accepted or rejected.
- `'active'` — Goal was accepted and is running on the server.
- `'succeeded'` — Goal executed successfully.
- `'preempted'` — An action client canceled the goal before it finished executing.
- `'aborted'` — The goal was aborted before it finished executing. The action server typically aborts a goal.
- `'rejected'` — The goal was not accepted after being in the `'pending'` state. The action server typically triggers this status.
- `'recalled'` — A client canceled the goal while it was in the `'pending'` state.
- `'lost'` — An internal error occurred in the action client.

ActivationFcn — Activation function

@(~) disp('Goal is active.') (default) | function handle

Activation function, returned as a function handle. This function executes when `GoalState` is set to `'active'`. By default, the function displays `'Goal is active.'`. You can set the function to `[]` to have the action client do nothing upon activation.

FeedbackFcn — Feedback function

`@(~,msg) disp(['Feedback: ', showdetails(msg)])` (default) | function handle

Feedback function, returned as a function handle. This function executes when a new feedback message is received from the action server. By default, the function displays the details of the message. You can set the function to `[]` to have the action client not give any feedback.

ResultFcn — Result function

`@(~,msg,s,~) disp(['Result with state ' s ': ', showdetails(msg)])`
(default) | function handle

Result function, returned as a function handle. This function executes when the server finishes executing the goal and returns a result state and message. By default, the function displays the state and details of the message. You can set the function to `[]` to have the action client do nothing once the goal is completed.

Object Functions

<code>cancelGoal</code>	Cancel last goal sent by client
<code>cancelAllGoals</code>	Cancel all goals on action server
<code>rosmessage</code>	Create ROS messages
<code>sendGoal</code>	Send goal message to action server
<code>sendGoalAndWait</code>	Send goal message and wait for result
<code>waitForServer</code>	Wait for action server to start

Examples

Setup a ROS Action Client and Execute an Action

This example shows how to create a ROS action client and execute the action. Action types must be setup beforehand with an action server running.

You must have the `'/fibonacci'` action type setup. To run this action server use the following command on the ROS system:

```
roslaunch actionlib_tutorials fibonacci_server
```

Connect to a ROS network. You must be connected to a ROS network to gather information about what actions are available. Replace `ipaddress` with your network address.

```
ipaddress = '192.168.154.131';  
rosinit(ipaddress)
```

```
Initializing global node /matlab_global_node_68978 with NodeURI http://192.168.154.1:5
```

List actions available on the network. The only action setup on this network is the `/fibonacci` action.

```
roaction list
```

```
/fibonacci
```

Create an action client. Specify the action name.

```
[actClient,goalMsg] = roactionclient('/fibonacci');
```

Wait for action client to connect to server.

```
waitForServer(actClient);
```

The fibonacci action will calculate the fibonacci sequence for a given order specified in the goal message. The goal message was returned when creating the action client and can be modified to send goals to the ROS action server.

```
goalMsg.Order = 8
```

```
goalMsg =
```

```
ROS FibonacciGoal message with properties:
```

```
  MessageType: 'actionlib_tutorials/FibonacciGoal'  
  Order: 8
```

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

Send goal and wait for its completion. Specify a timeout of 10 seconds to complete the action.

```
[resultMsg,resultState] = sendGoalAndWait(actClient,goalMsg,10)
```

```
Goal active
```

```
Feedback:
  Sequence : [0, 1, 1]
Feedback:
  Sequence : [0, 1, 1, 2]
Feedback:
  Sequence : [0, 1, 1, 2, 3]
Feedback:
  Sequence : [0, 1, 1, 2, 3, 5]
Feedback:
  Sequence : [0, 1, 1, 2, 3, 5, 8]
Feedback:
  Sequence : [0, 1, 1, 2, 3, 5, 8, 13]
Feedback:
  Sequence : [0, 1, 1, 2, 3, 5, 8, 13, 21]
Feedback:
  Sequence : [0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
Final state succeeded with result:
  Sequence : [0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
```

```
resultMsg =
```

```
  ROS FibonacciResult message with properties:
```

```
    MessageType: 'actionlib_tutorials/FibonacciResult'
    Sequence: [10×1 int32]
```

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

```
resultState =
```

```
  1×9 char array
```

```
succeeded
```

Disconnect from the ROS network.

```
roshutdown
```

```
Shutting down global node /matlab_global_node_68978 with NodeURI http://192.168.154.1:3
```

See Also

[cancelGoal](#) | [rosaction](#) | [rosmessage](#) | [sendGoal](#) | [waitForServer](#)

More About

- [“ROS Actions Overview”](#)
- [“Move a Turtlebot Robot Using ROS Actions”](#)

External Websites

- [ROS Actions](#)

Introduced in R2016b

Subscriber

Create a ROS subscriber

Description

The `Subscriber` object represents a subscriber on the ROS network. The `robotics.ros.Subscriber` object subscribed 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.

Create Object

`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, type)` specifies the message type, `type`, 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` in “Properties” on page 1-71 for more information about the callback function.

`sub = robotics.ros.Subscriber(node, topicname, type, callback)` specifies the topic name, type and callback function for the subscriber.

`sub = robotics.ros.Subscriber(____, 'BufferSize', value)` specifies the queue size in `value` for incoming messages. See the `BufferSize` in “Properties” on page

1-71 for more information. You can use any combination of previous inputs with this syntax.

Properties

TopicName — 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 — 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 — Latest message sent to the topic

Message object

Latest message sent to the topic, specified as a `Message` object. The `Message` object is specific to the given `MessageType`. If the subscriber has not received a message, then the `Message` object is empty.

BufferSize — Buffer size

1 (default) | scalar

Buffer size of the incoming message queue, specified as the comma-separated pair consisting of `'BufferSize'` and a scalar. If messages arrive faster and than your callback can process them, they are deleted once the incoming queue is full.

NewMessageFcn — Callback property

function handle | cell array

Callback property, specified as a function handle or cell array. In the first element of the cell array, specify either a function handle or a 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
rosmessage

Wait for new ROS message
Create ROS messages

Examples

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(rostype.std_msgs_String);
msg.Data = 'hello world';
send(pub, msg)
```

View the message data using the `LatestMessage` property of the `Subscriber` object.

```
pause(1)
sub.LatestMessage
```

```
ans =
```

```
ROS String message with properties:
```

```
  MessageType: 'std_msgs/String'
      Data: 'hello world'
```

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

Clear the publisher, subscriber, and ROS node. Shut down the ROS master.

```
clear('pub', 'sub', 'node')
clear('master')
```

- “Exchange Data with ROS Publishers and Subscribers”

See Also

[receive](#) | [rosmessage](#) | [rospublisher](#) | [rossubscriber](#)

Introduced in R2015a

TransformationTree

Receive, send, and apply ROS transformations

Description

ROS uses the tf transform library to store the relationship between multiple coordinate frames. The relative transformations between these coordinate frames are maintained in a tree structure. Querying this tree lets you transform entities like poses and points between any two coordinate frames.

Create 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.

Object Functions

<code>waitForTransform</code>	Wait until a transformation is available
<code>getTransform</code>	Retrieve the transformation between two coordinate frames
<code>transform</code>	Transform message entities into target coordinate frame
<code>sendTransform</code>	Send transformation to ROS network

Examples

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://bat5731win64:11311/.
Initializing global node /matlab_global_node_15043 with NodeURI http://bat5731win64:49
Using Master URI http://localhost:11311 from the global node to connect to the ROS master
```

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 =
    3×1 cell array
    'camera_center'
    'mounting_point'
```

```
'robot_base'  
  
updateTime =  
  ROS Time with properties:  
    Sec: 1.4726e+09  
    Nsec: 441000000
```

Wait for the transform between two frames, 'camera_center' and 'robot_base'. This will wait until the transformation is valid and block all other operations. A time out of 5 seconds is also given.

```
waitForTransform(tftree, 'robot_base', 'camera_center', 5)
```

Define a point in the camera's coordinate frame.

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

Transform the point into the 'base_link' frame.

```
tfpt = transform(tftree, 'robot_base', pt)
```

```
tfpt =
```

```
  ROS PointStamped message with properties:  
    MessageType: 'geometry_msgs/PointStamped'  
    Header: [1×1 Header]  
    Point: [1×1 Point]
```

Use `showdetails` to show the contents of the message

Display the transformed point coordinates.

```
tfpt.Point
```



```
ans =
```

```
ROS Point message with properties:
```

```
  MessageType: 'geometry_msgs/Point'  
             X: 1.2000  
             Y: 1.5000  
             Z: -2.5000
```

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

Clear ROS node. Shut down ROS master.

```
clear('node')  
roshutdown
```

```
Shutting down global node /matlab_global_node_15043 with NodeURI http://bat5731win64:4  
Shutting down ROS master on http://bat5731win64:11311/.
```

- “Access the tf Transformation Tree in ROS”

See Also

`getTransform` | `rostopic` | `sendTransform` | `transform` | `waitForTransform`

Introduced in R2015a

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.

Create Object

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

Properties

MessageType — Message type of ROS message

character vector

This property is read only.

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

Data Types: char

Header — ROS Header message

Header object

This property is read only.

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

ChildFrameID — Second coordinate frame to transform point into

character vector

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

Transform — Transformation message

Transform object

This property is read only.

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

Object Functions

apply

Transform message entities into target frame

Examples

Inspect Sample TransformStamped Object

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

Start ROS network and setup transformations.

```
rosinit
exampleHelperROSstartTfPublisher
```

```
Initializing ROS master on http://AH-SRADFORD:11311/.
Initializing global node /matlab_global_node_88892 with NodeURI http://AH-SRADFORD:5323
```

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          : 7483
FrameId      : camera_center
Stamp
  Sec        : 1441806033
  Nsec       : 206000000
Transform
Translation
  X          : 0.5
  Y          : 0
  Z          : -1
Rotation
  X          : -0
  Y          : -0.7071067812
  Z          : -0
  W          : 0.7071067812
```

Access the `Translation` vector inside the `Transform` object.

```
tform.Transform.Translation
```

```
ans =
```

```
ROS Vector3 message with properties:

MessageType: 'geometry_msgs/Vector3'
  X: 0.5000
  Y: 0
  Z: -1.0000
```

Use `showdetails` to show the contents of the message

Apply Transformation from the TransformationTree Using TransformStamped Object

This example shows how to apply the transformation from a `TransformStamped` object to a `PointStamped` message. It is assumed you already have a transformation saved under the variable `tform`.

`tform` is a saved transformation from a transformation tree. This can be retrieved from a ROS network and saved for offline use.

```
showdetails(tform)

ChildFrameId : robot_base
Header
Seq      : 7483
FrameId  : camera_center
Stamp
Sec      : 1441806033
Nsec     : 206000000
Transform
Translation
X : 0.5
Y : 0
Z : -1
Rotation
X : -0
Y : -0.7071067812
Z : -0
W : 0.7071067812
```

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 transformation to point.

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

- “Access the tf Transformation Tree in ROS”

See Also

Functions

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

Using Objects

[TransformationTree](#)

Introduced in R2015a

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.

Code Generation Support:

Supports MATLAB Function block: No

“Code Generation Support, Usage Notes and Limitations”

Construction

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

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

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

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

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

Input Arguments

width — Map width

double in meters

Map width, specified as a double in meters.

Data Types: double

height — Map height

double in meters

Map width, specified as a double in meters.

Data Types: double

resolution — Grid resolution

1 (default) | double in cells per meter

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

Data Types: double

p — Input occupancy grid

matrix of ones and zeros

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

Properties

GridSize — Number of rows and columns in grid

two-element horizontal vector

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

Resolution — Grid resolution

1 (default) | scalar in cells per meter

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

Data Types: `double`

XWorldLimits — Minimum and maximum values of x-coordinates

two-element vector

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

YWorldLimits — Minimum and maximum values of y-coordinates

two-element vector

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

GridLocationWorld — `[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

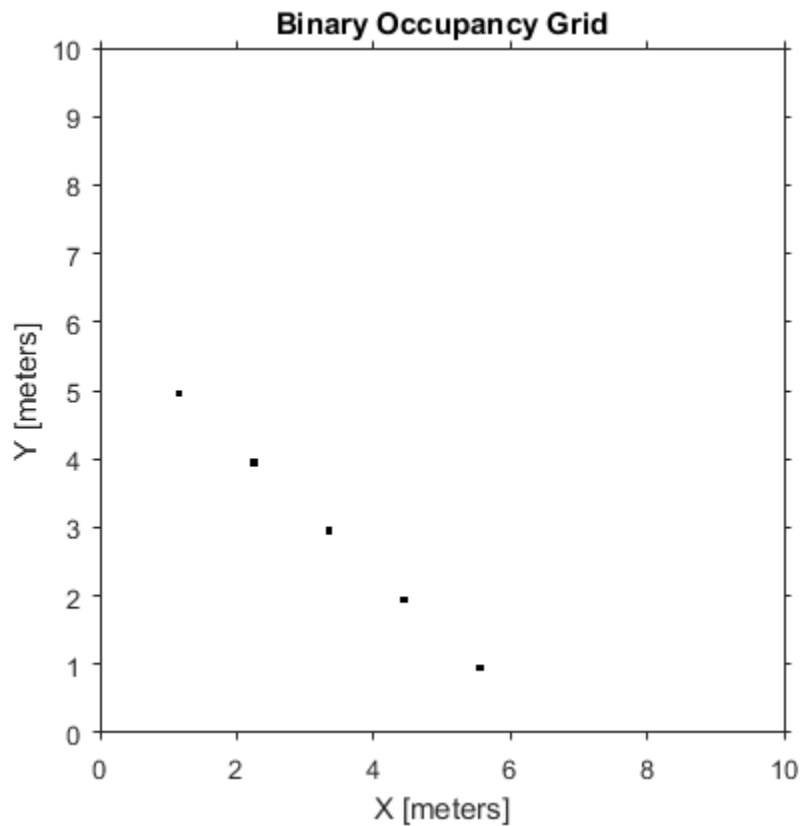
Create and Modify Binary Occupancy Grid

Create a 10m x 10m empty map.

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

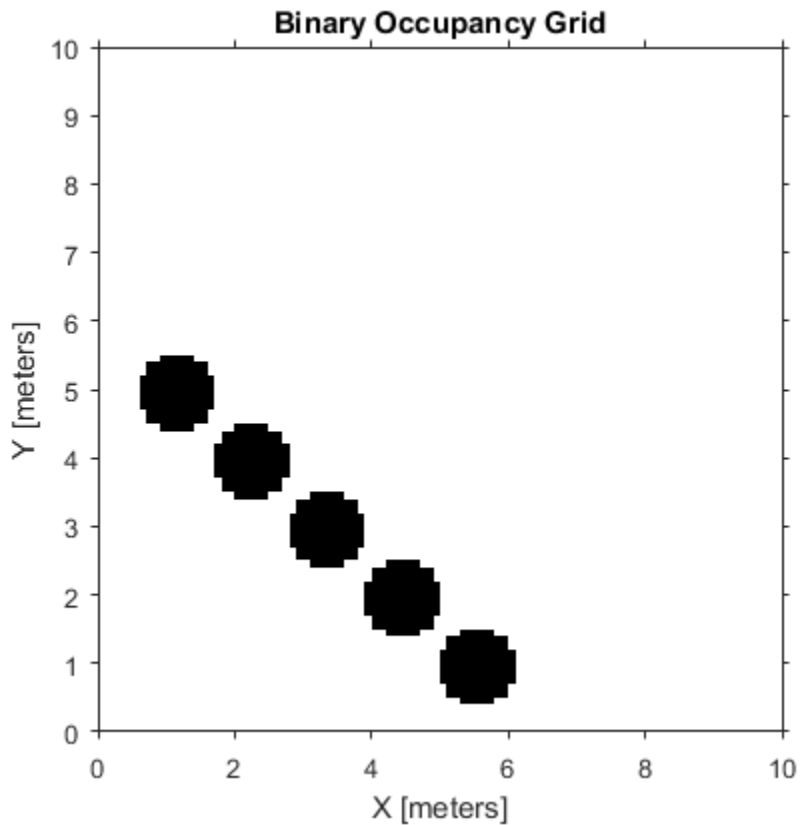
Set occupancy of world locations and show map.

```
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)
```

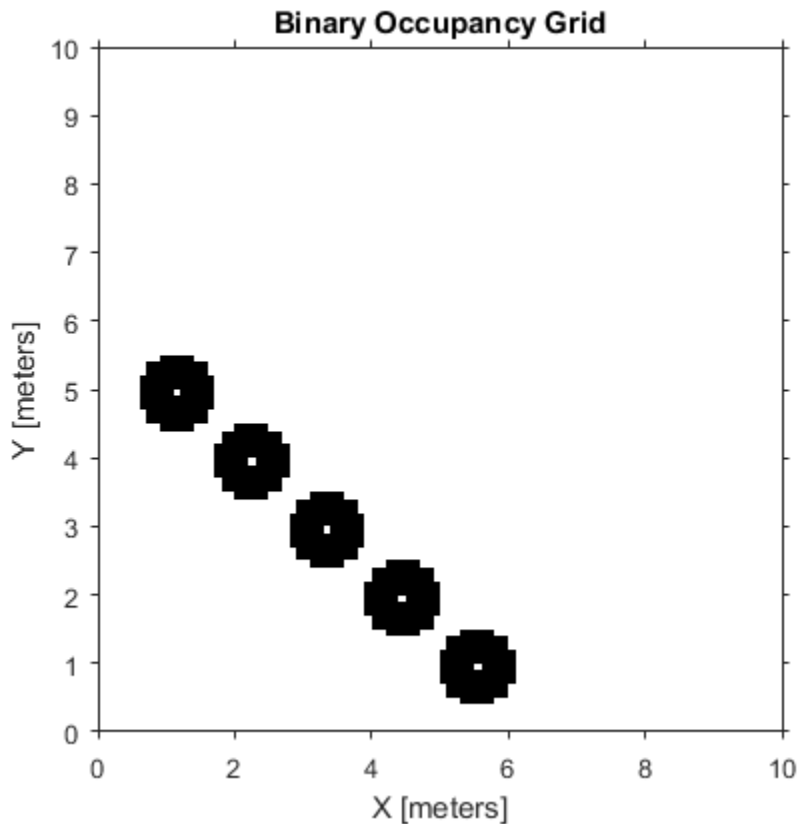


Image to Binary Occupancy Grid Example

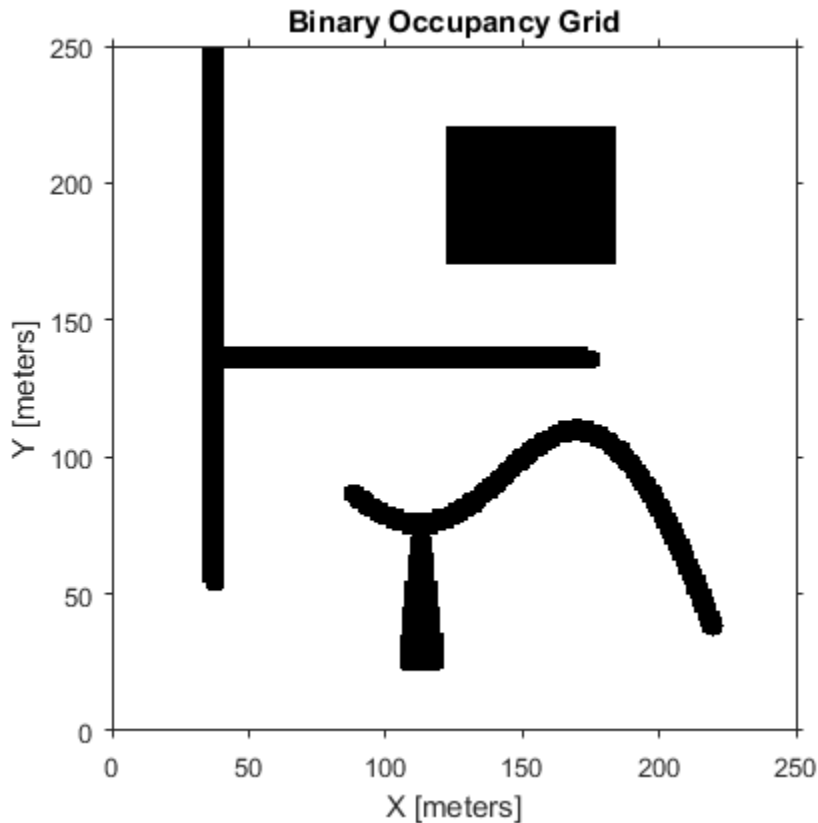
This example shows how to convert an image to a binary occupancy grid for using with the Robotics System Toolbox®

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

% Convert to grayscale and then black and white image based on arbitrary
% threshold.
grayimage = rgb2gray(image);
bwimage = grayimage < 0.5;
```

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

show(grid)
```



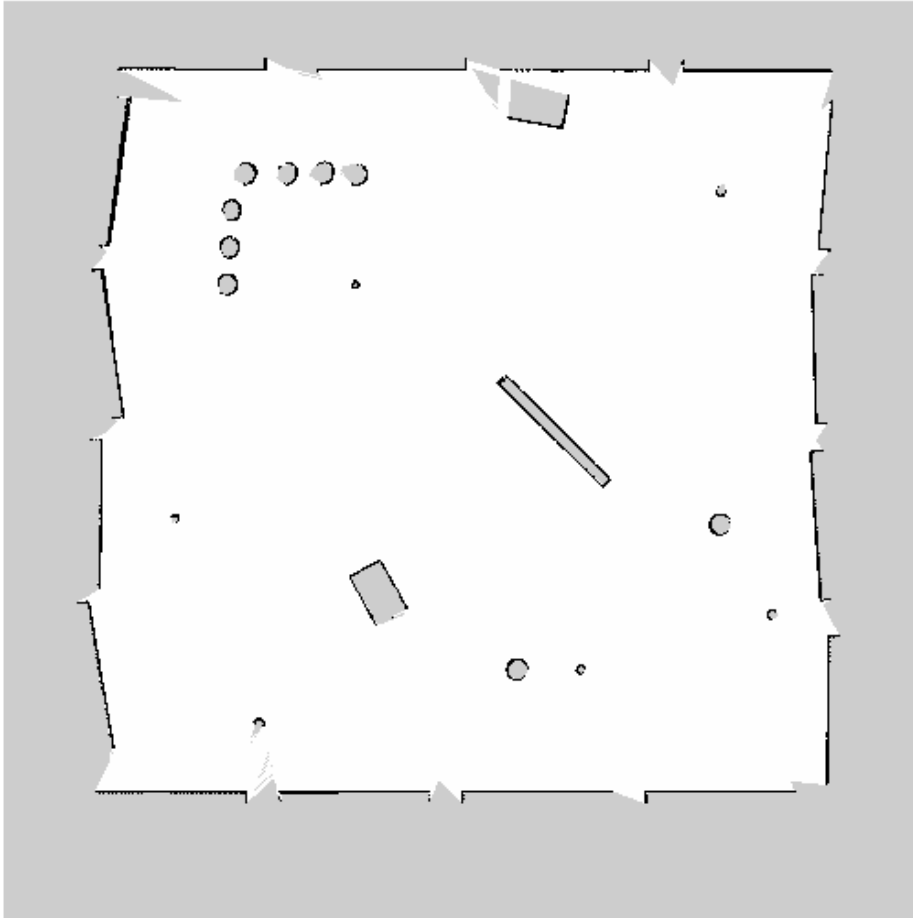
Convert PGM Image to Map

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

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

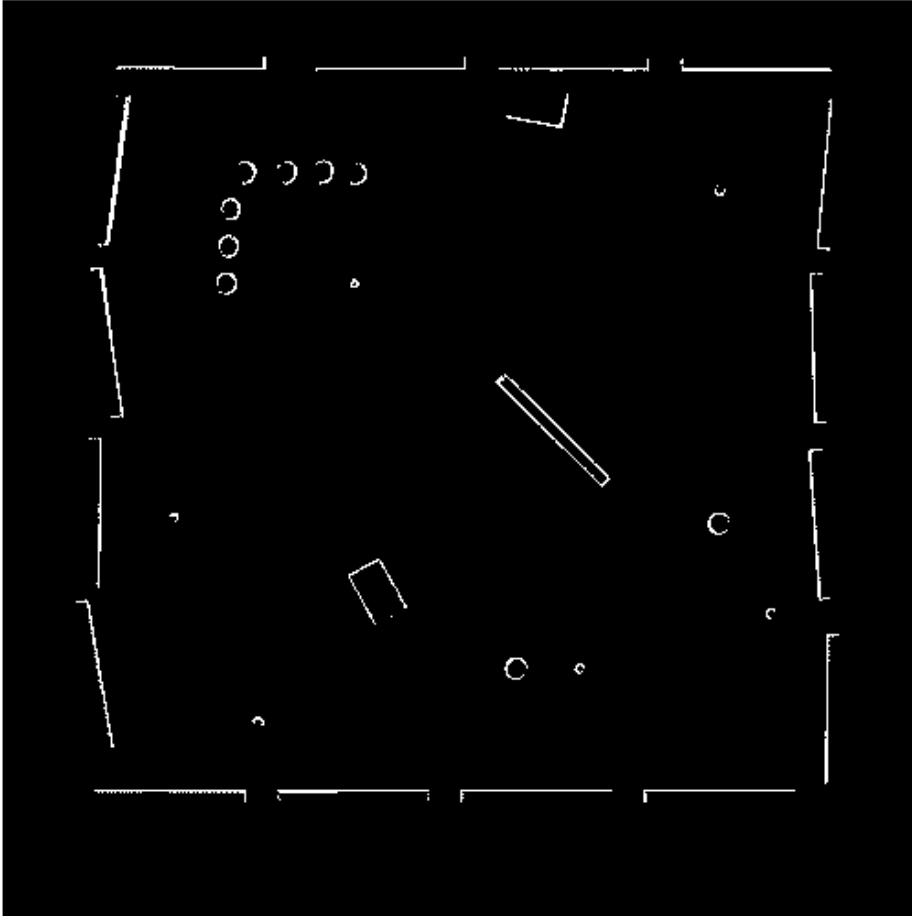
```
image = imread(fullfile(matlabroot, 'examples', 'robotics', 'playpen_map.pgm'));
imageCropped = image(750:1250, 750:1250);
```

```
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).

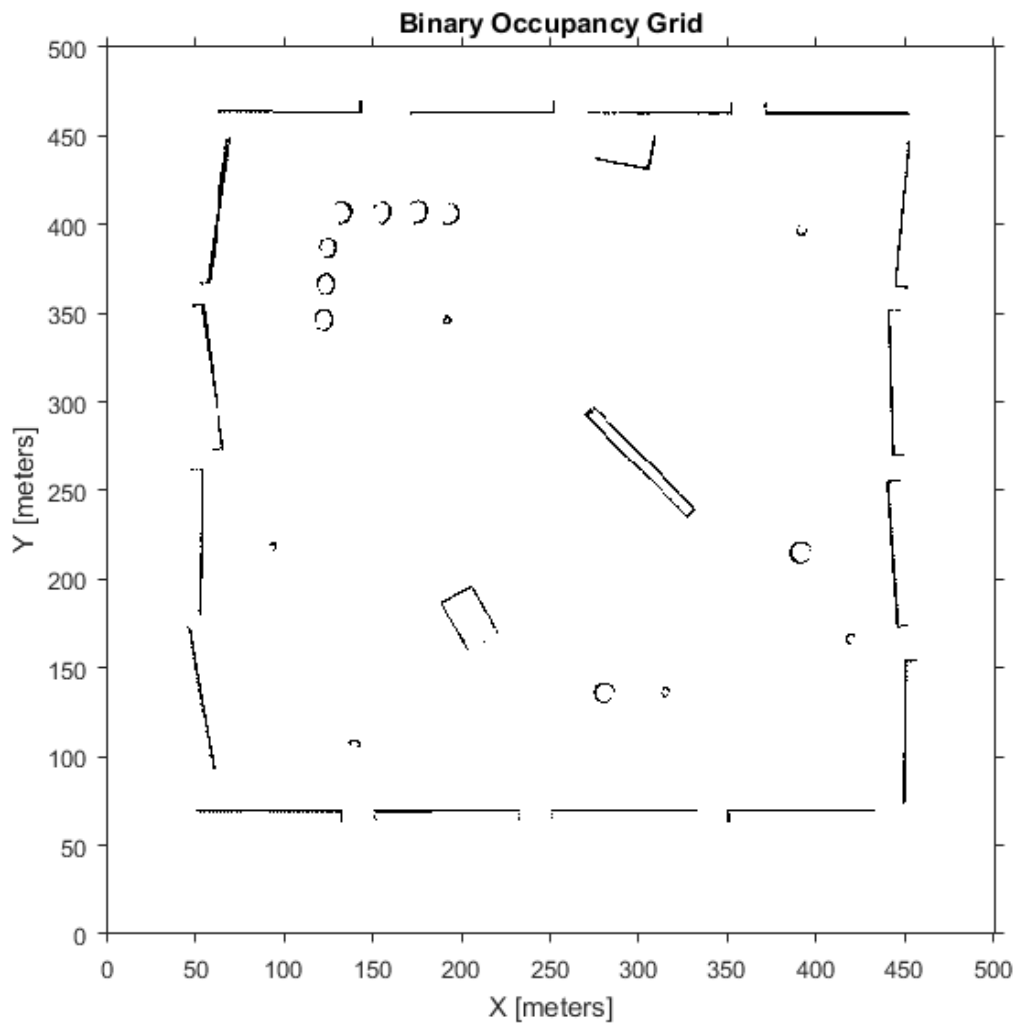
```
imageBW = imageCropped < 100;  
imshow(imageBW)
```



Create `BinaryOccupancyGrid` object using adjusted map image.

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

```
show(map)
```



See Also

[robotics.PRM](#) | [robotics.PurePursuit](#)

More About

- “Occupancy Grids”

Introduced in R2015a

robotics.InverseKinematics class

Package: robotics

Create inverse kinematic solver

Description

The `InverseKinematics` class 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 `robotics.RigidBodyTree`. 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.

Construction

`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

RigidBodyTree — Rigid body tree model

`RigidBodyTree` object

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

SolverAlgorithm — Algorithm for solving inverse kinematics

'BFGSGradientProjection' | 'LevenbergMarquardt'

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

SolverParameters — Parameters associated with algorithm

structure

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

Methods

Examples

Achieve EndEffector Position Using Inverse Kinematics

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

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

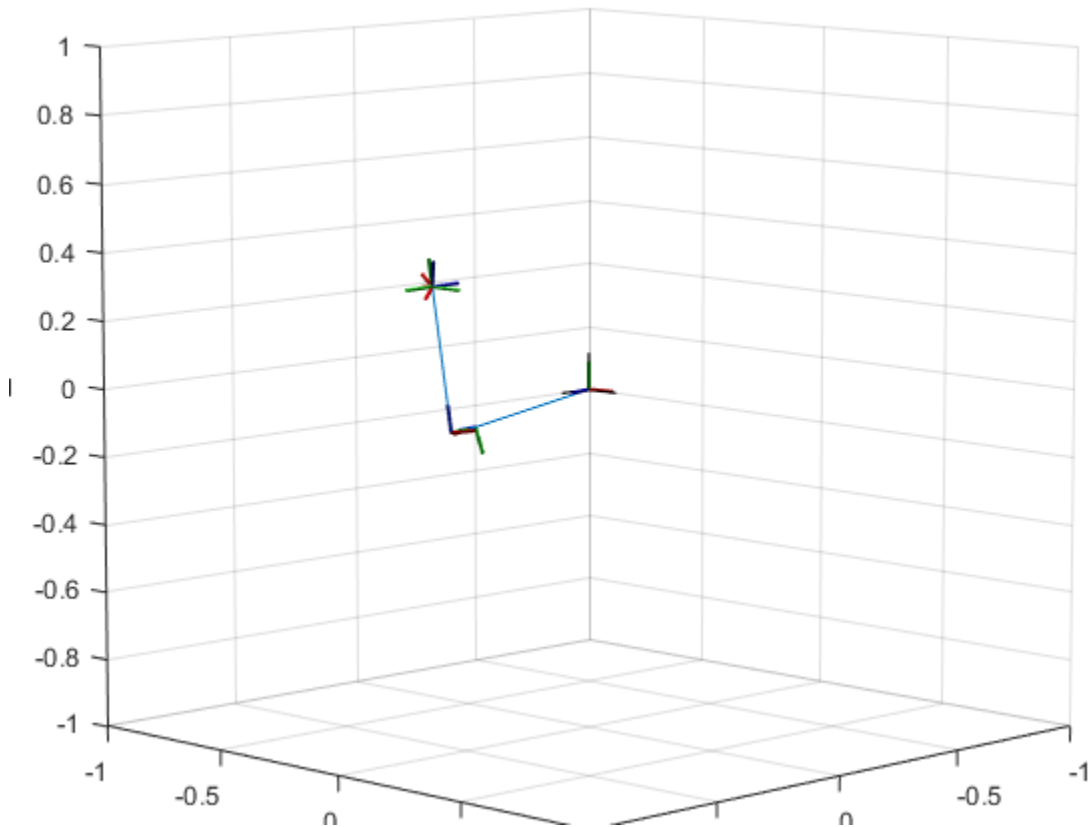
```
load exampleRobots.mat
showdetails(puma1)
```

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

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

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

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



Create an `InverseKinematics` object for the `puma1` model. Use this object to calculate a solution for the given goal. Specify weights for the different components of the pose. Use the home configuration of the robot as an initial guess.

```
ik = robotics.InverseKinematics('RigidBodyTree',puma1);
```

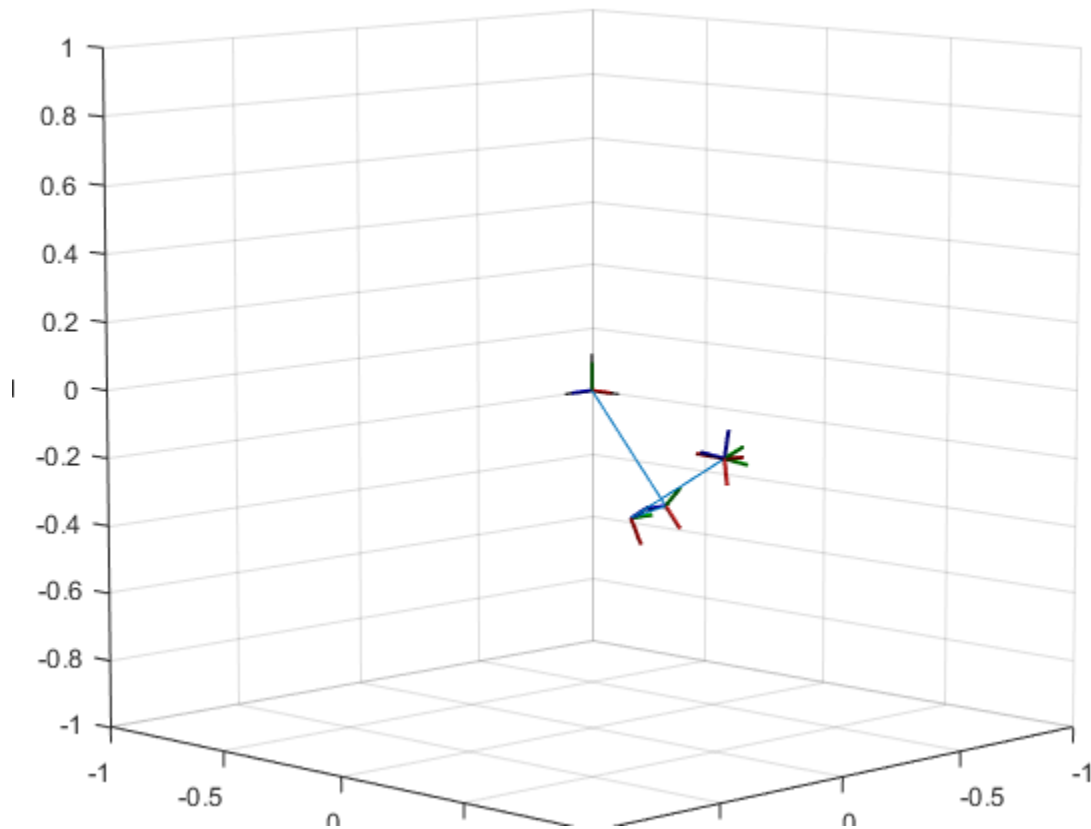
```
weights = ones(6,1);  
initialguess = puma1.homeConfiguration;
```

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);
```



- “Control PR2 Arm Movements using ROS Actions and Inverse Kinematics”

References

- [1] Badreddine, Hassan, Stefan Vandewalle, and Johan Meyers. "Sequential Quadratic Programming (SQP) for Optimal Control in Direct Numerical Simulation of Turbulent Flow." *Journal of Computational Physics*. 256 (2014): 1–16. doi:10.1016/j.jcp.2013.08.044.
- [2] Bertsekas, Dimitri P. *Nonlinear Programming*. Belmont, MA: Athena Scientific, 1999.
- [3] Goldfarb, Donald. "Extension of Davidon's Variable Metric Method to Maximization Under Linear Inequality and Equality Constraints." *SIAM Journal on Applied Mathematics*. Vol. 17, No. 4 (1969): 739–64. doi:10.1137/0117067.
- [4] Nocedal, Jorge, and Stephen Wright. *Numerical Optimization*. New York, NY: Springer, 2006.
- [5] Sugihara, Tomomichi. "Solvability-Unconcerned Inverse Kinematics by the Levenberg–Marquardt Method." *IEEE Transactions on Robotics* Vol. 27, No. 5 (2011): 984–91. doi:10.1109/tro.2011.2148230.
- [6] Zhao, Jianmin, and Norman I. Badler. "Inverse Kinematics Positioning Using Nonlinear Programming for Highly Articulated Figures." *ACM Transactions on Graphics* Vol. 13, No. 4 (1994): 313–36. doi:10.1145/195826.195827.

See Also

robotics.Joint | robotics.RigidBody | robotics.RigidBodyTree

More About

- “Inverse Kinematics Algorithms”
- Class Attributes
- Property Attributes

Introduced in R2016b

robotics.Joint class

Package: robotics

Create a joint

Description

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

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

The different joint types supported are:

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

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

Construction

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

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

Input Arguments

jname — Joint name

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 — 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:

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

Properties

Type — 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:

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

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

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.

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

HomePosition — Home position of joint

scalar

Home position of joint, specified as a scalar that depends on your joint type. The home position must fall in the range set by `PositionLimits`. This property is used by `robotics.RigidBodyTree.homeConfiguration` to generate the predefined home configuration for an entire rigid body tree.

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

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

JointAxis — Axis of motion for joint`[NaN NaN NaN]` (default) | three-element unit vector

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

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

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

JointToParentTransform — Fixed transform from joint to parent frame

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

This property is read only.

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

ChildToJointTransform — Fixed transform from child body to joint frame

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

This property is read only.

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

Methods

Examples

Attach Rigid Body and Joint to Rigid Body Tree

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

Create a rigid body tree.

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

Create a rigid body with a unique name.

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

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

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

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

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

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

```
showdetails(rbtree)
```

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

  Idx   Body Name   Joint Name   Joint Type   Parent Name(Idx)   Children Name(s)
  ---   -
    1     b1         jnt1        revolute        base(0)
-----
```

Build Manipulator Robot Using Denavit-Hartenberg Parameters

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

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

```
dhparams = [0    pi/2  0    0;
```

```
0.4318 0      0      0
0.0203 -pi/2 0.15005 0;
0      pi/2 0.4318 0;
0      -pi/2 0      0;
0      0      0      0];
```

Create a rigid body tree object to build the robot.

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

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

- 1 Create a `RigidBody` object and give it a unique name.
- 2 Create a `Joint` object and give it a unique name.
- 3 Use `setFixedTransform` to specify the body-to-body transformation using DH parameters.
- 4 Call `addBody` to attach the first body joint to the base frame of the robot.

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

setFixedTransform(jnt1,dhparams(1,:), 'dh');
body1.Joint = jnt1;

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

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

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

setFixedTransform(jnt2,dhparams(2,:), 'dh');
```

```

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

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

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

```

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

```

showdetails(robot)

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

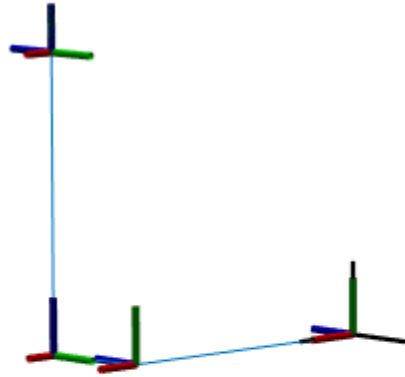
```

```

-----
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: [1×1 robotics.Joint]
    Parent: [1×1 robotics.RigidBody]
    Children: {[1×1 robotics.RigidBody]}
```

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'

```

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

```

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

```

```
showdetails(puma1)
```

```
-----
```

Robot: (6 bodies)

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

6

L6

jnt6

revolute

L5(5)

-
- “Build a Robot Step by Step”
 - “Rigid Body Tree Robot Model”

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.RigidBody](#) | [robotics.RigidBodyTree](#)

More About

- Class Attributes
- Property Attributes

Introduced in R2016b

robotics.LikelihoodFieldSensorModel class

Package: robotics

Create a likelihood field range sensor model

Description

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

Construction

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

Properties

Map — Occupancy grid representing the map

`BinaryOccupancyGrid` object (default)

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

SensorPose — Pose of the range sensor relative to the robot

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

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

SensorLimits — Minimum and maximum range of sensor

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

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

NumBeams — Number of beams used for likelihood computation

60 (default) | scalar

Number of beams used 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

Related Examples

- “Localize TurtleBot using Monte Carlo Localization”

More About

- “Monte Carlo Localization Algorithm”
- Class Attributes
- Property Attributes

Introduced in R2016a

robotics.MonteCarloLocalization System object

Package: robotics

Localize robot using range sensor data and map

Description

`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, range sensor 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 range sensor data as inputs. The input range 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. 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, repeat this process in a loop to continuously propagate particles, evaluate their likelihood, and get the best state estimate.

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.

Construction

`mcl = robotics.MonteCarloLocalization` returns a Monte Carlo localization (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`.

`mcl = robotics.MonteCarloLocalization(map,Name,Value)` specifies the map and other property values as `Name,Value` pairs.

Input Arguments

map — Map of the robot environment

BinaryOccupancyGrid object

Map of the robot environment, specified as a `BinaryOccupancyGrid` object. The map is assigned to the `Map` property of the `SensorModel` object. You must specify this input before the MCL object can begin localization. See `robotics.BinaryOccupancyGrid` for more information.

Properties

InitialPose — Initial pose of the 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

robotics.LikelihoodFieldSensorModel | 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

`robotics.OdometryMotionModel` (default) | `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 then 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) | scalar

Number of filter updates between resampling of particles, specified as a scalar.

Methods

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, $y = \text{step}(\text{obj}, x)$ and $y = \text{obj}(x)$ perform equivalent operations.

Create a Monte Carlo localization 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
```

- “Localize TurtleBot using Monte Carlo Localization”

References

- [1] Thrun, Sebastian, Wolfram Burgard, and Dieter Fox. *Probabilistic Robotics*. MIT Press, 2005.
- [2] Dellaert, F., D. Fox, W. Burgard, and S. Thrun. "Monte Carlo Localization for Mobile Robots." *Proceedings 1999 IEEE International Conference on Robotics and Automation*.

Limitations

- Calling `load`, `save`, or `clone` methods is not supported when MCL object is in the locked state. Use `robotics.MonteCarloLocalization.release` to unlock the object.

See Also

`robotics.LikelihoodFieldSensorModel` | `robotics.OdometryMotionModel`

More About

- “Monte Carlo Localization Algorithm”
- Class Attributes
- Property Attributes

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.

Code Generation Support:

Supports MATLAB Function block: No

“Code Generation Support, Usage Notes and Limitations”

Input Arguments

width — Map width

scalar in meters

Map width, specified as a scalar in meters.

Data Types: `double`

height — Map height

scalar in meters

Map height, specified as a scalar in meters.

Data Types: `double`

resolution — Grid resolution

1 (default) | scalar in cells per meter

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

Data Types: `double`

p — Input occupancy grid

matrix of probability values from 0 to 1

Input occupancy grid, specified as a matrix of probability values from 0 to 1. The size of the grid matches the size of the matrix. Each matrix element corresponds to the probability of the grid cell location being occupied. Values close to 0 represent a high certainty that the cell contains an obstacle. Values close to 1 represent certainty that the cell is not occupied and obstacle free.

Data Types: `double`

Properties

FreeThreshold — Threshold to consider cells as obstacle-free

scalar

Threshold to consider cells as obstacle-free, specified as a scalar. Probability values below this threshold are considered obstacle free. This property also defines the free locations for path planning when using `robotics.PRM`.

OccupiedThreshold — Threshold to consider cells as occupied

scalar

Threshold to consider cells as occupied, specified as a scalar. Probability values above this threshold are considered occupied.

ProbabilitySaturation — Saturation limits for probability

`[0.001 0.999]` (default) | `[min max]` vector

Saturation limits for probability, specified as a `[min max]` vector. Values above or below these saturation values are set to the `min` and `max` values. This property reduces oversaturating of cells when incorporating multiple observations.

GridSize — Number of rows and columns in grid

`[rows cols]` vector

This property is read only.

Number of rows and columns in grid, stored as a `[rows cols]` vector.

Resolution — Grid resolution

1 (default) | scalar in cells per meter

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

XWorldLimits — Minimum and maximum world range values of x-coordinates

[min max] vector

Minimum and maximum world range values of x -coordinates, stored as a [min max] vector. This value is read only.

YWorldLimits — Minimum and maximum world range values of y-coordinates

[min max] vector

Minimum and maximum world range values of y -coordinates, stored as a [min max] vector. This value is read only.

GridLocationInWorld — [x,y] world coordinates of grid

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

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

Methods

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);
```

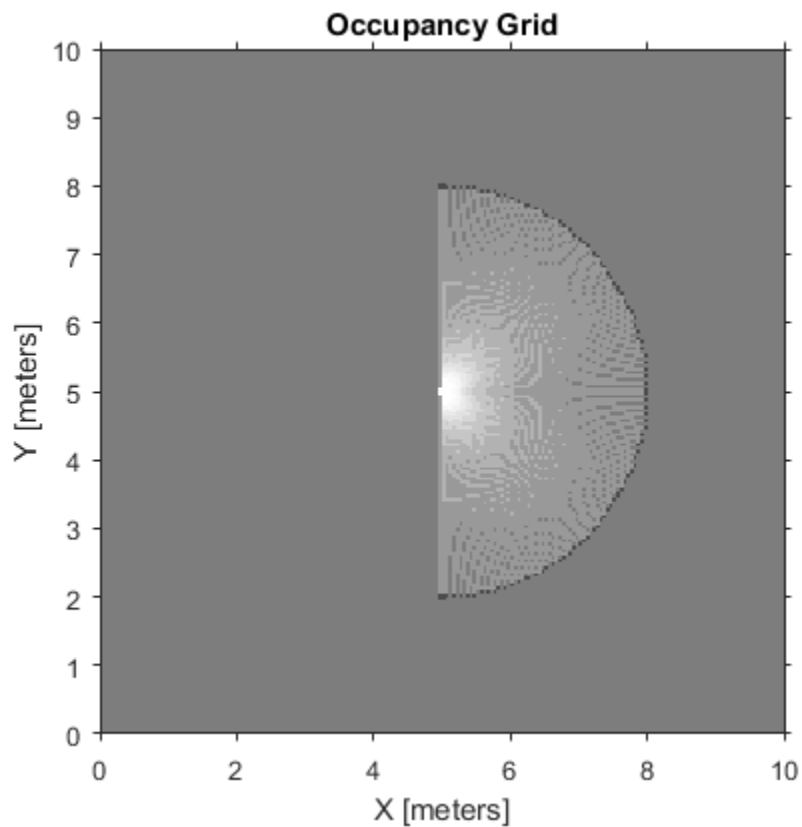
Show the map to see the results of inserting the laser scan. Check the occupancy of the spot directly in front of the robot.

```
show(map)
```

```
getOccupancy(map,[8 5])
```

```
ans =
```

```
0.7000
```

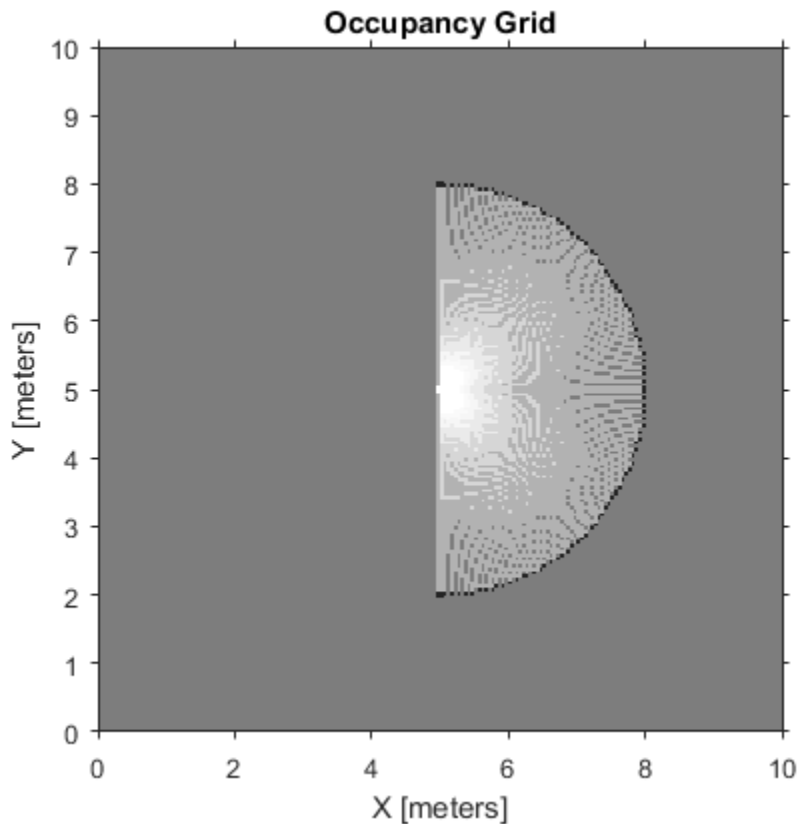


Add a second reading and view the update to the occupancy values. The additional reading increases the confidence in the readings. The free and occupied values become more distinct.

```
insertRay(map,pose,ranges,angles,maxrange);  
show(map)  
getOccupancy(map,[8 5])
```

```
ans =
```

```
0.8448
```

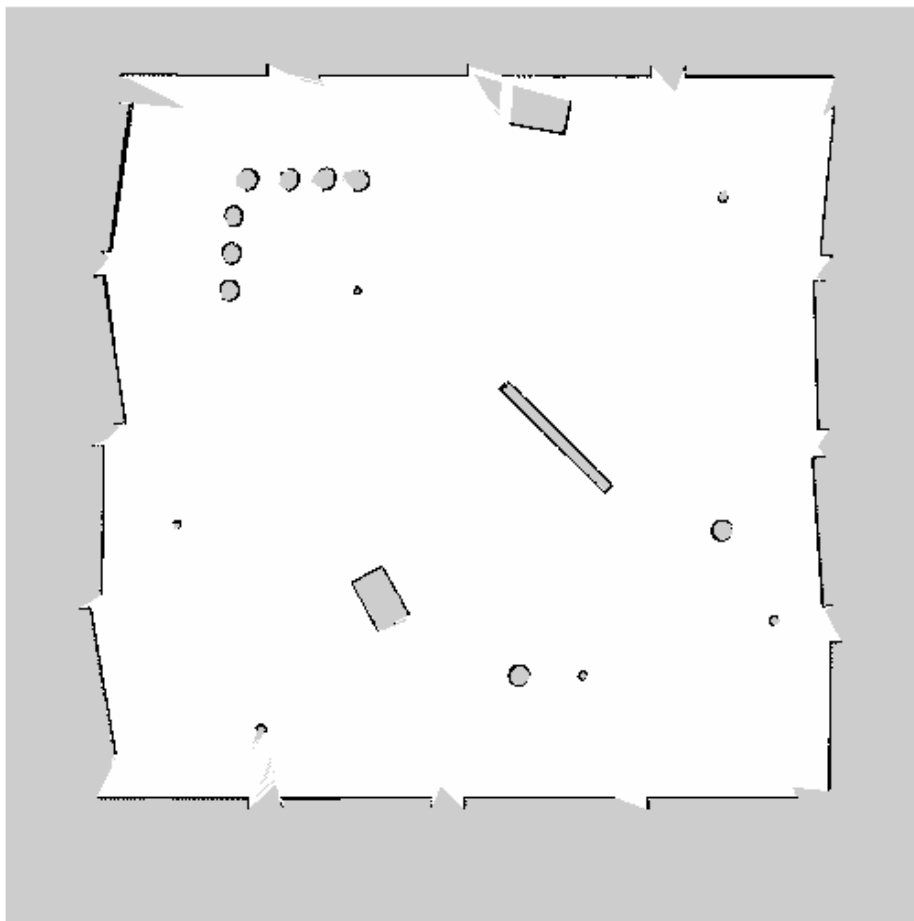



Convert PGM Image to Map

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)
```

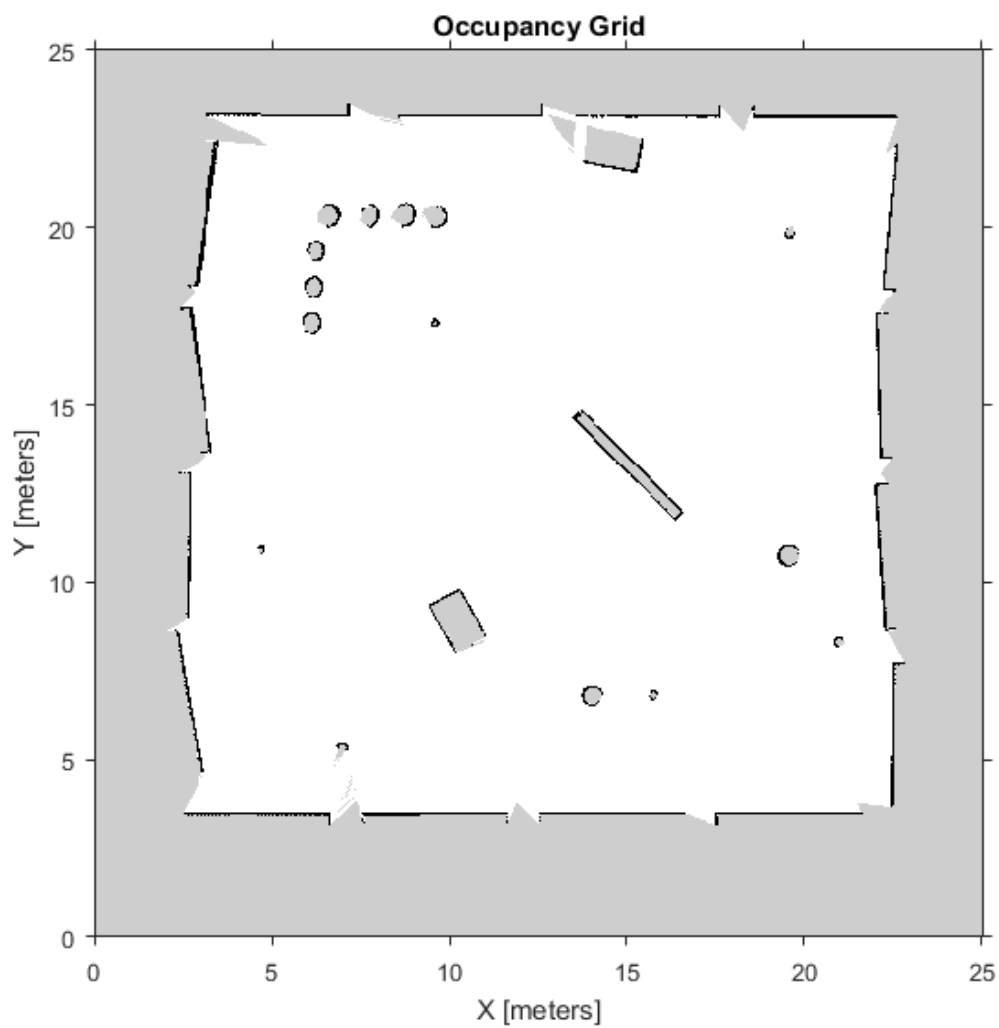


PGM values are expressed from 0 to 255 as `uint8`. Normalize these values by converting the cropped image to `double` and dividing each cell by 255. This image shows obstacles as values close to 0. Subtract the normalized image from 1 to get occupancy values with 1 representing occupied space.

```
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)
```



- “Mapping With Known Poses”

Limitations

Occupancy values have a limited resolution of ± 0.001 . The values are stored as `int16` using a log-odds representation. This data type limits resolution, but saves you memory when storing large maps in MATLAB. When calling `set` and then `get`, the value returned might not equal the value you set. For more information, see the log-odds representations section in “Occupancy Grids”.

See Also

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

More About

- “Occupancy Grids”
- Class Attributes
- Property Attributes

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 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`.

Code Generation Support:

Supports MATLAB Function block: No

“Code Generation Support, Usage Notes and Limitations”

Construction

`omm = robotics.OdometryMotionModel` creates an odometry motion model object for differential drive robots.

Properties

Noise — Gaussian noise for robot motion

[0.2 0.2 0.2 0.2] (default) | 4-element vector

Gaussian noise for robot motion, specified as a 4-element vector. This property represents the variance parameters for Gaussian noise applied to robot motion. The elements of the vector correspond to the following errors in order:

- Rotational error due to rotational motion
- Rotational error due to translational motion
- Translational error due to translation motion

- Translational error due to rotational motion

Type — Type of the odometry motion model

'DifferentialDrive' (default) | 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

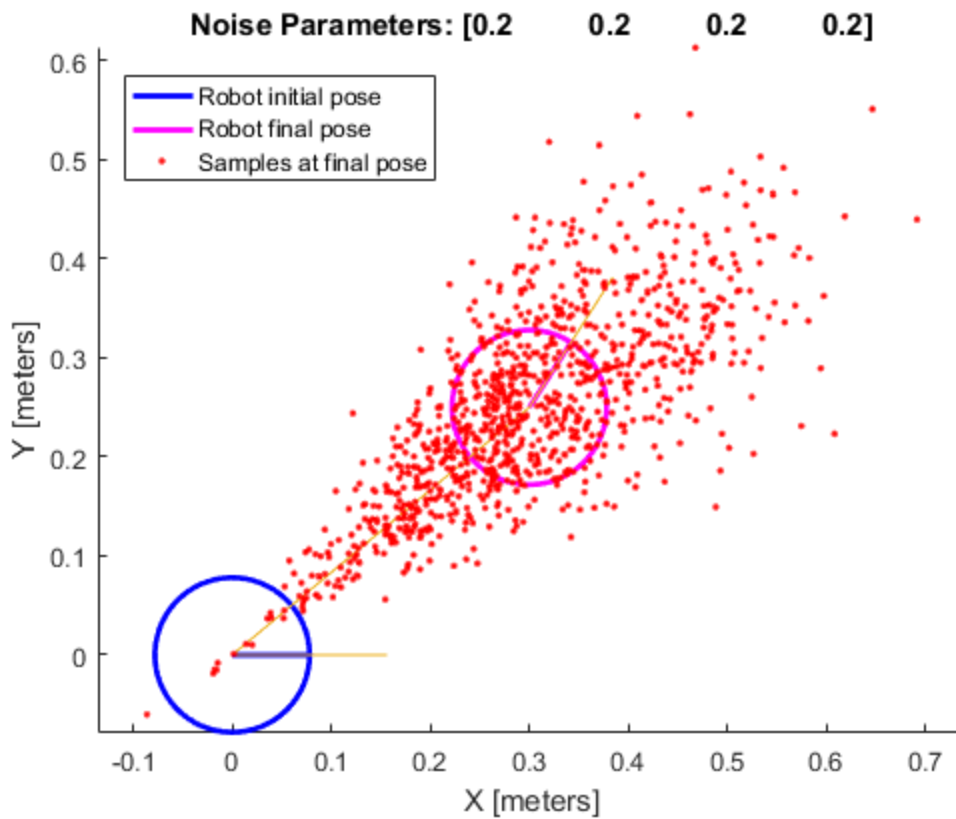
contains the motion model noise parameters for a differential drive robot. Use `showNoiseDistribution` to visualize how changing these values affect the distribution of predicted poses.

Create a motion model object.

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

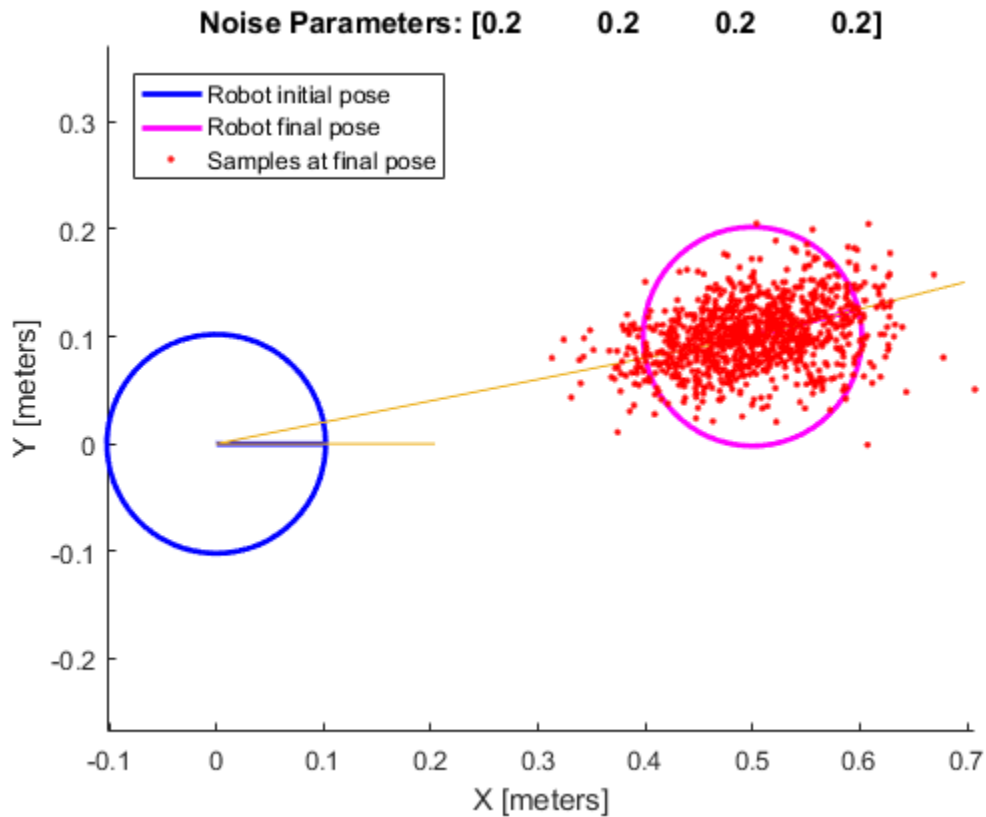
Show the distribution of particles with the existing noise parameters. Each particle is a hypothesis for the predicted pose.

```
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.

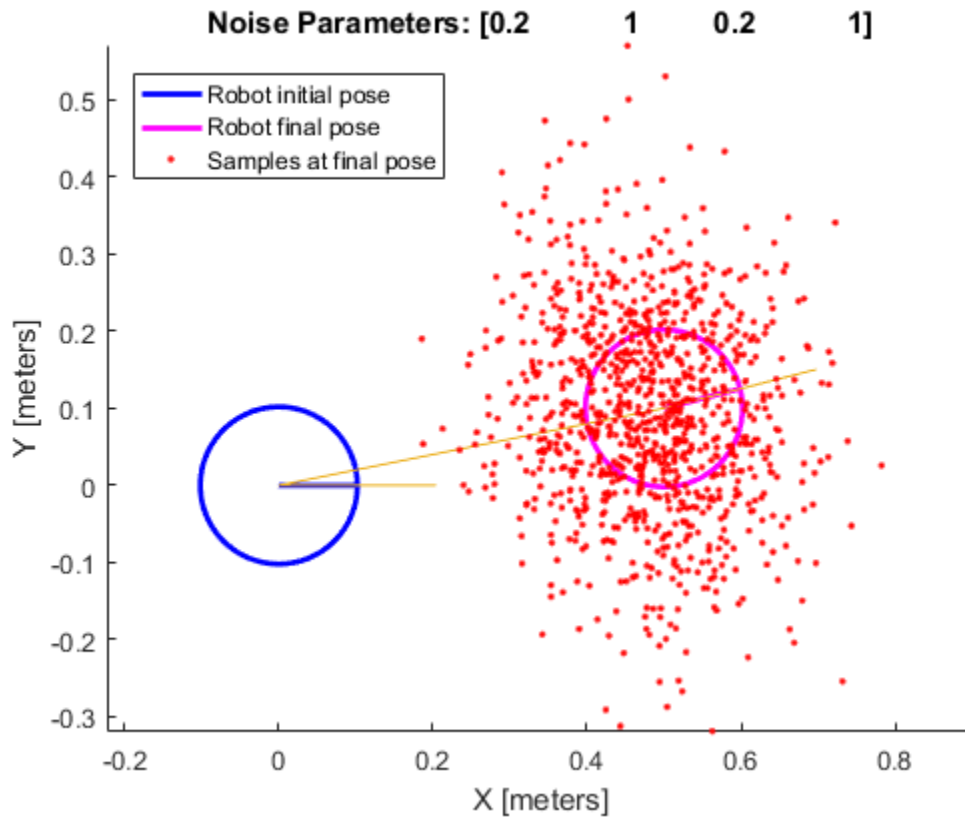
```
showNoiseDistribution(motionModel, ...
                    'OdometryPoseChange', [0.5 0.1 0.25], ...
                    'NumSamples', 1000);
```



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);
```



- “Localize TurtleBot using Monte Carlo Localization”

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

See Also

`robotics.MonteCarloLocalization` | `robotics.LikelihoodFieldSensorModel`

Introduced in R2016a

robotics.ParticleFilter class

Package: robotics

Create particle filter state estimator

Description

The particle filter is a recursive, Bayesian state estimator that uses discrete particles to approximate the posterior distribution of the estimated state.

The particle filter algorithm computes the state estimate recursively and involves two steps: prediction and correction. The prediction step uses the previous state to predict the current state based on a given system model. The correction step uses the current sensor measurement to correct the state estimate. The algorithm periodically redistributes, or resamples, the particles in the state space to match the posterior distribution of the estimated state.

The estimated state consists of state variables. Each particle represents a discrete state hypothesis of these state variables. The set of all particles is used to help determine the final state estimate.

You can apply the particle filter to arbitrary nonlinear system models. Process and measurement noise can follow arbitrary non-Gaussian distributions.

For more information on the particle filter workflow and setting specific parameters, see:

- “Particle Filter Workflow”
- “Particle Filter Parameters”

Code Generation Support:

Supports MATLAB Function block: No

“Code Generation Support, Usage Notes and Limitations”

Construction

`pf = robotics.ParticleFilter` creates a `ParticleFilter` object that enables the state estimation for a simple system with three state variables. Use the

`initialize` method to initialize the particles with a known mean and covariance or uniformly distributed particles within defined bounds. To customize the particle filter's system and measurement models, modify the `StateTransitionFcn` and `MeasurementLikelihoodFcn` properties.

After you create the `ParticleFilter` object, use `robotics.ParticleFilter.initialize` to initialize the `NumStateVariables` and `NumParticles` properties. The `initialize` function sets these two properties based on your inputs.

Properties

NumStateVariables — Number of state variables

3 (default) | scalar

This property is read only.

Number of state variables, specified as a scalar. This property is set based on the inputs to the `initialize` method. The number of states is implicit based on the specified matrices for initial state and covariance.

NumParticles — Number of particles used in the filter

1000 (default) | scalar

This property is read only.

Number of particles using in the filter, specified as a scalar. You can specify this property only by calling the `initialize` method.

StateTransitionFcn — Callback function for determining the state transition between particle filter steps

function handle

Callback function for determining the state transition between particle filter steps, specified as a function handle. The state transition function evolves the system state for each particle. The function signature is:

```
function predictParticles = stateTransitionFcn(pf,prevParticles,varargin)
```

The callback function accepts at least two input arguments: the `ParticleFilter` object, `pf`, and the particles at the previous time step, `prevParticles`. These specified

particles are the `predictParticles` returned from the previous `step` call of the `ParticleFilter` object. `predictParticles` and `prevParticles` are the same size: `NumParticles-by-NumStateVariables`.

You can also use `varargin` to pass in a variable number of arguments from the `predict` function. When you call:

```
predict(pf, arg1, arg2)
```

MATLAB essentially calls `stateTransitionFcn` as:

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

MeasurementLikelihoodFcn — Callback function calculating the likelihood of sensor measurements

(default) | function handle

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

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

The callback function accepts at least three input arguments:

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

You can also use `varargin` to pass in a variable number of arguments. These arguments are passed by the `correct` function. When you call:

```
correct(pf, measurement, arg1, arg2)
```

MATLAB essentially calls `measurementLikelihoodFcn` as:

```
measurementLikelihoodFcn(pf, predictParticles, measurement, arg1, arg2)
```

The callback needs to return exactly one output, `likelihood`, which is the likelihood of the given `measurement` for each particle state hypothesis.

IsStateVariableCircular — Indicator if state variables have a circular distribution

[0 0 0] (default) | logical array

Indicator if state variables have a circular distribution, specified as a logical array. Circular (or angular) distributions use a probability density function with a range of $[-\pi, \pi]$. If the `ParticleFilter` object has multiple state variables, then `IsStateVariableCircular` is a row vector. Each vector element indicates if the associated state variable is circular. If the object has only one state variable, then `IsStateVariableCircular` is a scalar.

ResamplingPolicy — Policy settings that determine when to trigger resampling

object

Policy settings that determine when to trigger resampling, specified as an object. You can trigger resampling either at fixed intervals, or you can trigger it dynamically, based on the number of effective particles. See `robotics.ResamplingPolicy` for more information.

ResamplingMethod — Method used for particle resampling

'multinomial' (default) | 'residual' | 'stratified' | 'systematic'

Method used for particle resampling, specified as 'multinomial', 'residual', 'stratified', and 'systematic'.

StateEstimationMethod — Method used for state estimation

'mean' (default) | 'smaxweight'

Method used for state estimation, specified as 'mean' and 'smaxweight'.

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: [1×1 robotics.ResamplingPolicy]
      ResamplingMethod: 'multinomial'
    StateEstimationMethod: 'mean'
      Particles: [1000×3 double]
      Weights: [1000×1 double]
```

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;
```

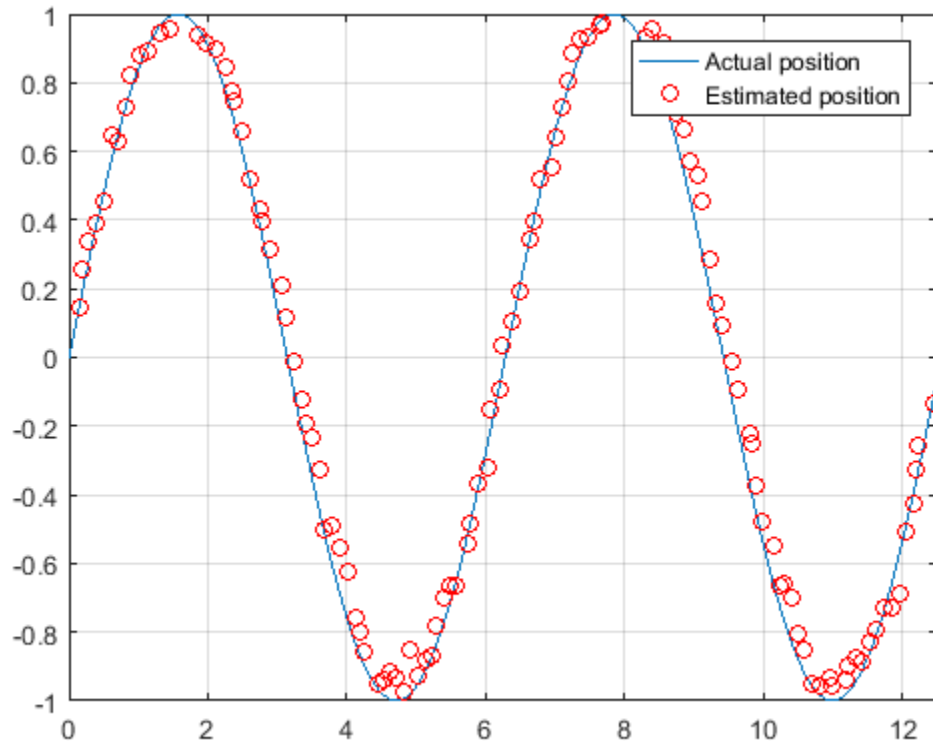
Begin the loop for predicting and correcting the estimated position based on measurements. The resampling of particles occurs based on the `ResamplingPolicy`

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.

- “Track a Car-Like Robot using Particle Filter”

References

- [1] Arulampala, 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.

See Also

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

More About

- “Particle Filter Parameters”
- “Particle Filter Workflow”
- Class Attributes
- Property Attributes

Introduced in R2016a

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.

Code Generation Support:

Supports MATLAB Function block: No

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

“Code Generation Support, Usage Notes and Limitations”

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

See Also

[robotics.BinaryOccupancyGrid](#) | [robotics.PurePursuit](#)

Related Examples

- “Path Planning in Environments of Different Complexity”

More About

- “Probabilistic Roadmaps (PRM)”

Introduced in R2015a

robotics.PurePursuit class

Package: robotics

Create controller to follow set of waypoints

Description

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 object with updated poses provide updated velocity commands for the robot to follow a path along a desired set of waypoints. Use the `MaxAngularVelocity` and `DesiredLinearVelocity` properties to update the velocities based on the robot's performance.

The `LookaheadDistance` property computes a look-ahead point on the path, which is a local goal for the robot. The angular velocity command is computed based on this point. Changing `LookaheadDistance` has a significant impact on the performance of the algorithm. A higher look-ahead distance results in a smoother trajectory for the robot, but can cause the robot to cut corners along the path. Too low of a look-ahead distance can result in oscillations in tracking the path, causing unstable behavior. For more information on the pure pursuit algorithm, see “Pure Pursuit Controller”.

Code Generation Support:

Supports MATLAB Function block: No

“Code Generation Support, Usage Notes and Limitations”

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.

Construction

`controller = robotics.PurePursuit` creates a pure pursuit object, `controller`, 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.

Properties

'DesiredLinearVelocity' — Desired constant linear velocity

0.1 (default) | scalar in meters per second

Desired constant linear velocity, specified as the comma-separated pair consisting of `'DesiredLinearVelocity'` and a scalar in meters per second. The controller assumes that the robot drives at a constant linear velocity and that the computed angular velocity is independent of the linear velocity.

Data Types: `double`

'LookaheadDistance' — Look-ahead distance

1.0 (default) | scalar in meters

Look-ahead distance, specified as the comma-separated pair consisting of `'LookaheadDistance'` and a scalar in meters. The look-ahead distance changes the response of the controller. A robot with higher look-ahead distance produces smooth paths but takes larger turns at corners. A robot with smaller look-ahead distance follows the path closely and takes sharp turns, but can produce oscillations in the path.

Data Types: `double`

'MaxAngularVelocity' — Maximum angular velocity

1.0 (default) | scalar in radians per second

Maximum angular velocity, specified as the comma-separated pair consisting of `'MaxAngularVelocity'` and 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`

Methods

See Also

`robotics.BinaryOccupancyGrid` | `robotics.PRM`

Related Examples

- “Path Following for a Differential Drive Robot”

More About

- “Pure Pursuit Controller”

Introduced in R2015a

robotics.Rate class

Package: robotics

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 `robotics.Rate.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.

Construction

`rateObj = robotics.Rate(desiredRate)` creates a `Rate` object that operates loops at a fixed-rate based on your system time.

Input Arguments

desiredRate — Desired execution rate

scalar

Desired execution rate of loop, specified as a scalar in Hz. When using `robotics.Rate.waitFor`, the loop operates every `desiredRate` seconds, unless the loop takes longer. It then begins the next loop based on the specified `OverrunAction`.

Properties

DesiredRate — Desired execution rate

scalar

Desired execution rate of loop, specified as a scalar in Hz. When using `robotics.Rate.waitFor`, the loop operates every `desiredRate` seconds, unless the loop takes longer. It then begins the next loop based on the specified `OverrunAction`.

DesiredPeriod — Desired time period between executions

scalar

Desired time period between executions, specified as a scalar in seconds. This property is equal to the inverse of `DesiredRate`.

TotalElapsedTime — Elapsed time since construction or reset

scalar

Elapsed time since construction or reset, specified as a scalar in seconds.

LastPeriod — Elapsed time between last two calls to `waitFor`

NaN (default) | scalar

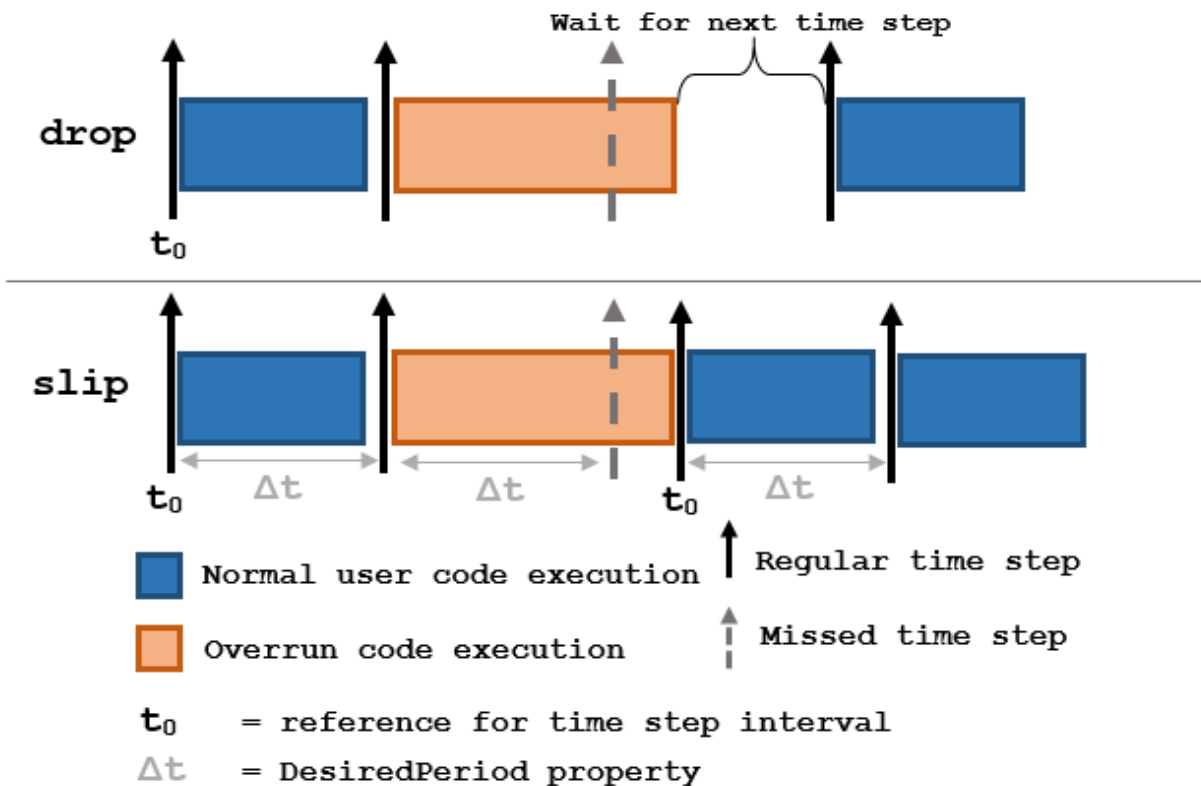
Elapsed time between last two calls to `waitFor`, specified as a scalar. By default, `LastPeriod` is set to NaN until `waitFor` is called for the first time. After the first call, `LastPeriod` equals `TotalElapsedTime`.

OverrunAction — Method for handling overruns

'slip' (default) | 'drop'

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

- 'drop' — waits until the next time interval equal to a multiple of `DesiredPeriod`
- 'slip' — immediately executes the loop again



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

Methods

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. Print the iteration and the time elapsed.

```
for i = 1:10;
    time = r.TotalElapsedTime;
    fprintf('Iteration: %d - Time Elapsed: %f\n',i,time)
    waitfor(r);
end
```

```
Iteration: 1 - Time Elapsed: 5.344408
Iteration: 2 - Time Elapsed: 5.345697
Iteration: 3 - Time Elapsed: 6.345708
Iteration: 4 - Time Elapsed: 7.345741
Iteration: 5 - Time Elapsed: 8.345797
Iteration: 6 - Time Elapsed: 9.345820
Iteration: 7 - Time Elapsed: 10.345850
Iteration: 8 - Time Elapsed: 11.344907
Iteration: 9 - Time Elapsed: 12.344948
Iteration: 10 - Time Elapsed: 13.344967
```

Each iteration executes at a 1-second interval.

- “Execute Code at a Fixed-Rate”

See Also

[robotics.Rate.waitfor](#) | [robotics.Rate.statistics](#) | [ROS Rate](#)

More About

- [Class Attributes](#)
- [Property Attributes](#)

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.

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 your

Properties

TriggerMethod — Method for determining if resampling should occur

'ratio' (default) | character vector

Method for determining if resampling should occur, specified as a character vector. Possible choices are 'ratio' and 'interval'. The 'interval' method triggers resampling at regular intervals of operating the particle filter. The 'ratio' method triggers resampling based on the ratio of effective total particles.

SamplingInterval — Fixed interval between resampling

1 (default) | scalar

Fixed interval between resampling, specified as a scalar. This interval determines during which correction steps the resampling is executed. For example, a value of 2 means the resampling is executed every second correction step. A value of `inf` means that resampling is never executed.

This property only applies with the `TriggerMethod` is set to 'interval'.

MinEffectiveParticleRatio — Minimum desired ratio of effective to total particles

0.5 (default) | scalar

Minimum desired ratio of effective to total particles, specified as a scalar. The effective number of particles is a measure of how well the current set of particles approximates the posterior distribution. A lower effective particle ratio means less particles are contributing to the estimation and resampling might be required. If the ratio of effective particles to total particles falls below the `MinEffectiveParticleRatio`, a resampling step is triggered.

See Also

`robotics.ParticleFilter` | `robotics.ParticleFilter.correct`

Related Examples

- “Track a Car-Like Robot using Particle Filter”

More About

- Class Attributes
- Property Attributes

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 appropriately modify the entire tree structure.

Construction

`body = robotics.RigidBody(name)` creates a rigid body with the specified name. By default, the body comes with a fixed joint.

Input Arguments

name — Name of rigid body
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.

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.

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 Puma560® robot to build a robot. Each rigid body is added one at a time, with the child-to-parent transform specified by the joint object.

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

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

Create a rigid body tree object to build the robot.

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

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

- 1 Create a `RigidBody` object and give it a unique name.
- 2 Create a `Joint` object and give it a unique name.
- 3 Use `setFixedTransform` to specify the body-to-body transformation using DH parameters.
- 4 Call `addBody` to attach the first body joint to the base frame of the robot.

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

setFixedTransform(jnt1,dhparams(1,:), 'dh');
body1.Joint = jnt1;

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

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

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

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

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

```

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

```

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

```
showdetails(robot)
```

```

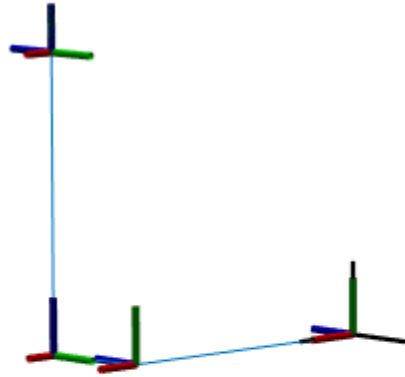
show(robot);
axis([-0.5,0.5,-0.5,0.5,-0.5,0.5])
axis off

```

```
-----
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)	

```
-----
```



- “Build a Robot Step by Step”
- “Rigid Body Tree Robot Model”

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.RigidBodyTree](#) | [robotics.RigidBodyTree.addBody](#) | [robotics.RigidBodyTree.replaceJoint](#)

More About

- [Class Attributes](#)
- [Property Attributes](#)

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.

A rigid body tree model is made up of rigid bodies as `robotics.RigidBody` objects. Each rigid body has a `robotics.Joint` object associated with it that defines how it can move relative to its parent body. Use `robotics.Joint.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.

You can also use this robot model to calculate joint angles for desired end-effector positions using inverse kinematics. Specify your rigid body tree model when using `robotics.InverseKinematics`.

Construction

`robot = robotics.RigidBodyTree` creates a tree-structured robot object. Add rigid bodies to it using `robotics.RigidBodyTree.addBody`.

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.

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 Puma560® robot to build a robot. Each rigid body is added one at a time, with the child-to-parent transform specified by the joint object.

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

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

Create a rigid body tree object to build the robot.

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

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

- 1 Create a `RigidBody` object and give it a unique name.
- 2 Create a `Joint` object and give it a unique name.
- 3 Use `setFixedTransform` to specify the body-to-body transformation using DH parameters.
- 4 Call `addBody` to attach the first body joint to the base frame of the robot.

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

setFixedTransform(jnt1,dhparams(1,:), 'dh');
body1.Joint = jnt1;

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

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

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

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

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

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

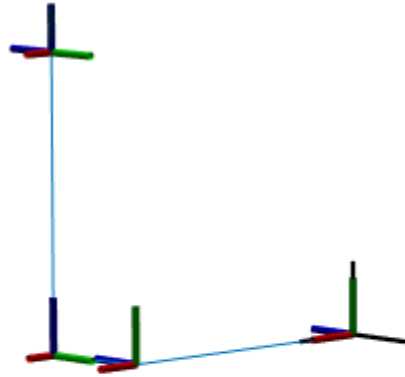
Verify that your robot was built properly by using the `showdetails` or `show` function. `showdetails` lists all the bodies in the MATLAB® command window. `show` displays the

robot with a given configuration (home by default). Calls to `axis` modify the axis limits and hide the axis labels.

```
showdetails(robot)
```

```
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]
Parent: [1x1 robotics.RigidBody]
Children: {[1x1 robotics.RigidBody]}
```

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'
```

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

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

```
showdetails(puma1)
```

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

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

6

L6

jnt6

revolute

L5(5)

-
- “Build a Robot Step by Step”
 - “Rigid Body Tree Robot Model”

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.InverseKinematics](#)

More About

- Class Attributes
- Property Attributes

Introduced in R2016b

robotics.VectorFieldHistogram System object

Package: robotics

Avoid obstacles using vector field histogram

Description

The vector field histogram (VFH) class enables your robot to avoid obstacles based on range sensor data. Given a range sensor reading in terms of ranges and angles, and a target direction to drive toward, the VFH controller computes an obstacle-free steering direction.

The class uses the VFH+ algorithm to compute the obstacle-free direction. First, the algorithm takes the ranges and angles from range sensor data and builds a polar histogram for obstacle locations. Then, it uses the input histogram thresholds to calculate a binary histogram that indicates occupied and free directions. Finally, the algorithm computes a masked histogram, which is computed from the binary histogram based on the minimum turning radius of the robot.

The algorithm selects multiple steering directions based on the open space and possible driving directions. A cost function, with weights corresponding to the previous, current, and target directions, calculates the cost of different possible directions. The algorithm then returns an obstacle-free direction with minimal cost. Using the obstacle-free direction, you can input commands to move your robot in that direction.

To use this class 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.

Code Generation Support:

Supports MATLAB Function block: No

“Code Generation Support, Usage Notes and Limitations”

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.

Construction

`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 each specified property name set to the specified value. You can specify multiple properties in any order as `Name1, Value1, . . . NameN, ValueN`.

Properties

NumAngularSectors — Number of angular sectors in histogram

180 (default) | scalar

Number of angular sectors, 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. 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 the robot in meters

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 the robot

0.1 (default) | scalar

Safety distance around the robot, specified as a scalar. This is a safety distance to leave around the robot position in addition to `RobotRadius`. The robot radius and safety distance are used to compute the obstacle-free direction.

MinTurningRadius — Minimum turning radius at current speed

0.1 (default) | scalar

Minimum turning radius for the robot moving at its current speed, specified as a scalar.

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 `CurrentDirectionWeight` and `PreviousDirectionWeight`. 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 produces 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).

Methods

Examples

Create a Vector Field Histogram Object

This example shows how to create a Vector Field Histogram (VFH) object and calculate a steering direction based on input laser scan data.

Create VFH 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 obstacle-free steering direction

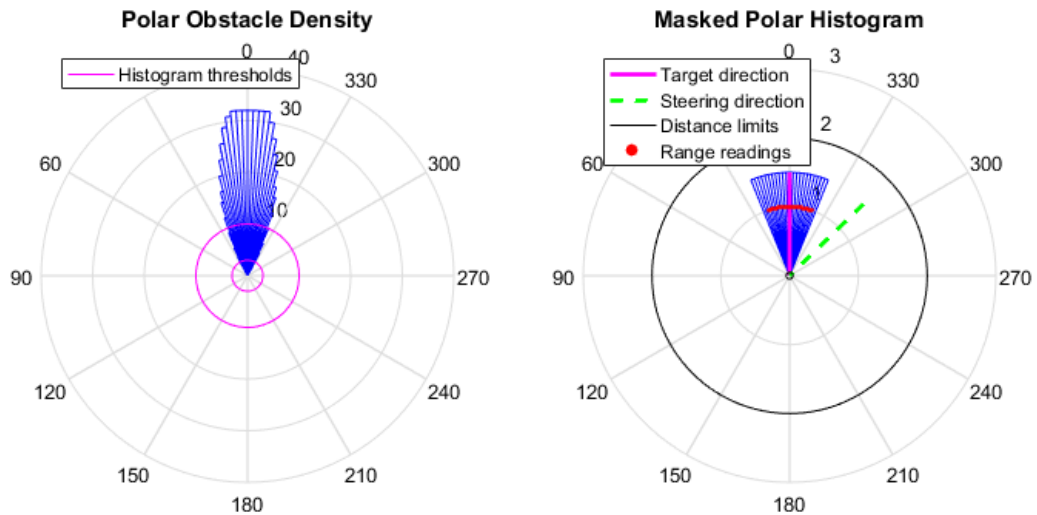
```
steeringDir = vfh(ranges,angles,targetDir)
```

```
steeringDir =
```

```
-0.8014
```

Visualize the VectorFieldHistogram computation

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



- “Obstacle Avoidance using TurtleBot”

References

- [1] Borenstein, J., and Y. Koren. "The Vector Field Histogram - Fast Obstacle Avoidance for Mobile Robots." *IEEE Journal of Robotics and Automation* 7, no. 3 (1991) pp.278–88.
- [2] Ulrich, I., and J. Borenstein. "VFH : Reliable Obstacle Avoidance for Fast Mobile Robots." *Proceedings. 1998 IEEE International Conference on Robotics and Automation.* (1998): 1572–1577.

See Also

robotics.PRM

More About

- “Vector Field Histogram”

Introduced in R2015b

Functions — 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 .

Code Generation Support:

Supports MATLAB Function block: Yes

“Code Generation Support, Usage Notes and Limitations”

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 $[-\pi, \pi]$.

Introduced in R2015a

apply

Transform message entities into target frame

Syntax

```
tfentity = apply(tfmsg,entity)
```

Description

`tfentity = apply(tfmsg,entity)` applies the transformation represented by the 'TransformStamped' ROS message to the input message object `entity`.

This function determines the message type of `entity` and applies the appropriate transformation method to it. If the object cannot handle a particular message type, then MATLAB displays an error message.

If you only want to use the most current transformation, call `transform` instead. If you want to store a transformation message for later use, call `getTransform` and then call `apply`.

Examples

Apply Transformation to a Point

```
tfPoint = apply(transform,point);
```

Input Arguments

tfmsg — Transformation message

TransformStamped ROS message handle

Transformation message, specified as a TransformStamped ROS message handle. The `tfmsg` is a ROS message of type: `geometry_msgs/TransformStamped`.

entity — ROS message

Message object handle

ROS message, specified as a Message object handle.

Supported messages are:

- `geometry_msgs/PointStamped`
- `geometry_msgs/PoseStamped`
- `geometry_msgs/PointCloud2Stamped`
- `geometry_msgs/QuaternionStamped`
- `geometry_msgs/Vector3Stamped`

Output Arguments

tfentity — Transformed ROS message

Message object handle

Transformed ROS message, returned as a Message object handle.

See Also

`getTransform` | `transform`

Introduced in R2015a

axang2quat

Convert axis-angle rotation to quaternion

Syntax

```
quat = axang2quat(axang)
```

Description

`quat = axang2quat(axang)` converts a rotation given in axis-angle form, `axang`, to quaternion, `quat`.

Code Generation Support:

Supports MATLAB Function block: Yes

“Code Generation Support, Usage Notes and Limitations”

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 — 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]`

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).

Code Generation Support:

Supports MATLAB Function block: Yes

“Code Generation Support, Usage Notes and Limitations”

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]`

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).

Code Generation Support:

Supports MATLAB Function block: Yes

“Code Generation Support, Usage Notes and Limitations”

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 — 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]`

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

Create Service Client and Call for Response Using Default Message

```
client = rossvcclient('/gazebo/get_model_state');
response = call(client);
```

Call for Response Using Specific Request Message

```
reqmessage = rosmesssage(client);
```

```
response = call(client,reqmessage);
```

Wait for Response Using Timeout of Five Seconds

```
response = call(client,reqmessage,'Timeout',5);
```

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://192.168.154.1:5
```

Send a goal message with modified parameters. Wait for the goal to finish executing.

```
goalMsg.Order = 4;  
sendGoalAndWait(actClient,goalMsg)
```

```
Goal active  
Feedback:  
  Sequence : [0, 1, 1]
```

```

Feedback:
  Sequence : [0, 1, 1, 2]
Feedback:
  Sequence : [0, 1, 1, 2, 3]
Feedback:
  Sequence : [0, 1, 1, 2, 3, 5]

ans =

ROS FibonacciResult message with properties:

  MessageType: 'actionlib_tutorials/FibonacciResult'
  Sequence: [6x1 int32]

Use showdetails to show the contents of the message

```

Send a new goal message without waiting.

```

goalMsg.Order = 5;
sendGoal(actClient,goalMsg)

```

Cancel the goal on the ROS action client, `actClient`.

```

cancelGoal(actClient)

```

Cancel all the goals on the action server that `actClient` is connected to.

```

cancelAllGoals(actClient)

```

Delete the action client.

```

delete(actClient)

```

Disconnect from the ROS network.

```

roshutdown

```

```

Shutting down global node /matlab_global_node_40739 with NodeURI http://192.168.154.1:30000

```

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.

More About

- “ROS Actions Overview”
- “Move a Turtlebot Robot Using ROS Actions”

See Also

`cancelGoal` | `roaction` | `sendGoal` | `sendGoalAndWait`

Introduced in R2016b

cancelGoal

Cancel last goal sent by client

Syntax

```
cancelGoal(client)
```

Description

`cancelGoal(client)` sends a cancel request for the tracked goal, which is the last one sent to the action server. The specified client sends the request.

If the goal is in the 'active' state, the server preempts the execution of the goal. If the goal is 'pending', it is recalled. If this client has not sent a goal, or if the previous goal was achieved, this function returns immediately.

Examples

Send and Cancel ROS Action Goals

Send and cancel goals for ROS actions. First, setup a ROS action client. Then send a goal message with modified parameters. Finally, cancel your goal and all goals on the action server.

Connect to a ROS network with a specified IP address. Create a ROS action client connected using `roactionclient`. Specify the action name. Wait for the client to be connected to the server.

```
rosinit('192.168.154.131')
[actClient,goalMsg] = roactionclient('/fibonacci');
waitForServer(actClient);
```

```
Initializing global node /matlab_global_node_40739 with NodeURI http://192.168.154.1:5
```

Send a goal message with modified parameters. Wait for the goal to finish executing.

```
goalMsg.Order = 4;  
sendGoalAndWait(actClient,goalMsg)
```

```
Goal active  
Feedback:  
  Sequence : [0, 1, 1]  
Feedback:  
  Sequence : [0, 1, 1, 2]  
Feedback:  
  Sequence : [0, 1, 1, 2, 3]  
Feedback:  
  Sequence : [0, 1, 1, 2, 3, 5]
```

```
ans =
```

```
ROS FibonacciResult message with properties:
```

```
  MessageType: 'actionlib_tutorials/FibonacciResult'  
  Sequence: [6×1 int32]
```

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

Send a new goal message without waiting.

```
goalMsg.Order = 5;  
sendGoal(actClient,goalMsg)
```

Cancel the goal on the ROS action client, `actClient`.

```
cancelGoal(actClient)
```

Cancel all the goals on the action server that `actClient` is connected to.

```
cancelAllGoals(actClient)
```

Delete the action client.

```
delete(actClient)
```

Disconnect from the ROS network.

```
roshutdown
```

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

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.

More About

- “ROS Actions Overview”
- “Move a Turtlebot Robot Using ROS Actions”

See Also

`cancelAllGoals` | `roaction` | `sendGoal` | `sendGoalAndWait`

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.

```
roshutdown
```

```
Shutting down global node /matlab_global_node_61809 with NodeURI http://172.28.194.90:6
```

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://172.28.194.90:6
```

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

Transform the ROS message to the 'base_link' frame using the desired time saved from before.

```
tfpt = transform(tftree,'base_link',pt,desiredTime);
```

Optional: You can also use `apply` with the stored `tform` to apply this transformation to the `pt` message.

```
tfpt2 = apply(tform,pt);
```

Shut down the ROS network.

```
roshutdown
```

```
Shutting down global node /matlab_global_node_93523 with NodeURI http://172.28.194.90:0
```

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.

Code Generation Support:

Supports MATLAB Function block: Yes

“Code Generation Support, Usage Notes and Limitations”

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-1)* matrix

Cartesian coordinates, specified as an n -by- $(k-1)$ matrix, containing n points. Each row of `cart` represents a point in $(k-1)$ -dimensional space. k must be greater than or equal to 2.

Example: [0.8147 0.1270 0.6324; 0.9058 0.9134 0.0975]

Output Arguments

hom — Homogeneous points

n -by- k matrix

Homogeneous points, returned as an n -by- k matrix, containing n points. k must be greater than or equal to 2.

Example: [0.2785 0.9575 0.1576 0.5; 0.5469 0.9649 0.9706 0.5]

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 `rosmesssage` 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

`rosmesssage` | `rosmmsg`

Introduced in R2015a

del

Delete a ROS parameter

Syntax

```
del(ptree, paramname)
```

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.

Examples

Delete Parameter on ROS Master

Create parameter tree, 'MyParam' parameter, and check existence.

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

```
ans =
```

```
1
```

Delete parameter and check existence.

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

```
ans =
```

0

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.

See Also

`has` | `rosparam` | `set`

Introduced in R2015a

deleteFile

Delete file from device

Syntax

```
deleteFile(device,filename)
```

Description

`deleteFile(device,filename)` deletes the specified file from the ROS device.

Examples

Put, Get, and Delete Files on ROS Device

Put a file from your host computer onto a ROS device, get it back, and then delete it.

Connect to a ROS device. Specify the device address, user name, and password of your ROS device.

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

Put a new text file that is in the MATLAB® current folder onto the ROS device. The destination folder must exist.

```
putFile(d,'test_file.txt','/home/user/test_folder')
```

Get a text file from the ROS device. You can get any file, not just ones added from MATLAB®. By default, the file is added to the MATLAB current folder.

```
getFile(d,'/home/user/test_folder/test_file.txt')
```

Delete the text file on the ROS device.

```
deleteFile(d,'/home/user/test_folder/test_file.txt')
```

Put, Get, and Delete Files on ROS Device Using Wildcards

Put a file from your host computer onto a ROS device, get it back, and then delete it. Use wildcards to search for all matching files.

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 all text files at the specified path onto the ROS device. The destination folder must exist.

```
putFile(d, 'C:/MATLAB/*.txt', '/home/user/test_folder')
```

Get all text files from the ROS device. You can get any files, not just ones added from MATLAB®. By default, the files are added to the MATLAB current folder.

```
getFile(d, '/home/user/test_folder/*.txt')
```

Delete all text files on the ROS device at the specified folder.

```
deleteFile(d, '/home/user/test_folder/*.txt')
```

Input Arguments

device — ROS device

rosdevice object

ROS device, specified as a rosdevice object.

filename — File to delete

character vector

File to delete, specified as a character vector. When you specify the file name, you can use path information and wildcards.

Example: '/home/user/image.jpg'

Example: '/home/user/*.jpg'

Data Types: cell

See Also

dir | getFile | openShell | putFile | rosdevice | system

Introduced in R2016b

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'.

Code Generation Support:

Supports MATLAB Function block: Yes

“Code Generation Support, Usage Notes and Limitations”

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(eu1, 'ZYZ')
```

```
qZYZ =
```

```
  0.7071      0      0      0.7071
```

Input Arguments

eu1 — 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'

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

- 'ZYX' (default) – The order of rotation angles is *z*-axis, *y*-axis, *x*-axis.
- 'ZYZ' – The order of rotation angles is *z*-axis, *y*-axis, *z*-axis.

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]

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'.

Code Generation Support:

Supports MATLAB Function block: Yes

“Code Generation Support, Usage Notes and Limitations”

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

eu1 — 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'

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

- 'ZYX' (default) — The order of rotation angles is *z*-axis, *y*-axis, *x*-axis.
- 'ZYZ' — The order of rotation angles is *z*-axis, *y*-axis, *z*-axis.

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]`

See Also

`rotm2eu1`

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'.

Code Generation Support:

Supports MATLAB Function block: Yes

“Code Generation Support, Usage Notes and Limitations”

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

eu1 — 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'

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

- 'ZYX' (default) – The order of rotation angles is *z*-axis, *y*-axis, *x*-axis.
- 'ZYZ' – The order of rotation angles is *z*-axis, *y*-axis, *z*-axis.

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]

See Also

tform2eul

Introduced in R2015a

get

Get ROS parameter value

Syntax

```
pvalue = get(ptree,paramname)
```

Description

`pvalue = get(ptree,paramname)` gets the value of the parameter with the name `paramname` from the parameter tree object `ptree`.

Supported parameter values are:

- `int32`
- `logical`
- `double`
- character vector
- cell array

Examples

Set and Get Parameter Value

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

```
ptree = rosparam;
```

Set the parameter value.

```
set(ptree, 'DoubleParam', 1.0)
```

Get the parameter value.

```
get(ptree, 'DoubleParam')
```

```
ans =  
    1
```

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

pvalue — Parameter value

int32 | logical | char | double | cell array

Parameter value, returned as either a `int32`, `logical`, `double`, `char`, or `cell` array. `pvalue` matches the value of the specified `paramname` and the supported data type in `ParameterTree`. Base64–encoded binary data, iso8601 data, and dictionaries from ROS are not supported.

Limitations

Base64–encoded binary data, iso8601 data, and dictionaries from ROS are not supported.

See Also

`rosparam` | `set`

Introduced in R2015a

getFile

Get file from device

Syntax

```
getFile(device,remoteSource)
getFile(device,remoteSource,localDestination)
```

Description

`getFile(device,remoteSource)` copies the specified file from the ROS device to the MATLAB current folder. Wildcards are supported.

`getFile(device,remoteSource,localDestination)` copies the remote file to a destination path. Specify a file name at the end of the destination path to copy with a custom file name.

Examples

Put, Get, and Delete Files on ROS Device

Put a file from your host computer onto a ROS device, get it back, and then delete it.

Connect to a ROS device. Specify the device address, user name, and password of your ROS device.

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

Put a new text file that is in the MATLAB® current folder onto the ROS device. The destination folder must exist.

```
putFile(d,'test_file.txt','/home/user/test_folder')
```

Get a text file from the ROS device. You can get any file, not just ones added from MATLAB®. By default, the file is added to the MATLAB current folder.

```
getFile(d,'/home/user/test_folder/test_file.txt')
```

Delete the text file on the ROS device.

```
deleteFile(d, '/home/user/test_folder/test_file.txt')
```

Put, Get, and Delete Files on ROS Device Using Wildcards

Put a file from your host computer onto a ROS device, get it back, and then delete it. Use wildcards to search for all matching files.

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 all text files at the specified path onto the ROS device. The destination folder must exist.

```
putFile(d, 'C:/MATLAB/*.txt', '/home/user/test_folder')
```

Get all text files from the ROS device. You can get any files, not just ones added from MATLAB®. By default, the files are added to the MATLAB current folder.

```
getFile(d, '/home/user/test_folder/*.txt')
```

Delete all text files on the ROS device at the specified folder.

```
deleteFile(d, '/home/user/test_folder/*.txt')
```

Input Arguments

device — ROS device

rosdevice object

ROS device, specified as a rosdevice object.

remoteSource — Path and name of file on ROS device

source path

Path and name of the file on the ROS device. Specify the path as a character vector. You can use an absolute path or a relative path from the MATLAB Current Folder. Use the path and file naming conventions of the operating system on your host computer.

Example: '/home/user/test_folder/test_file.txt'

Data Types: char

localDestination — Destination folder path and optional file name

character vector

Destination folder path and optional file name, specified as a character vector. Specify a file name at the end of the destination path to copy with a custom file name. Use the host computer path and file naming conventions.

Example: 'C:/User/username/test_folder'

Data Types: char

See Also

`deleteFile` | `dir` | `openShell` | `putFile` | `rosdevice` | `system`

Introduced in R2016b

getTransform

Retrieve the transformation between two coordinate frames

Compatibility

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 `roscpp`. Replace `ipaddress` with your ROS network address.

```
ipaddress = '172.28.194.91';  
roscpp(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 = rosmesssage('geometry_msgs/PointStamped');
pt.Header.FrameId = 'camera_link';
pt.Point.X = 3;
pt.Point.Y = 1.5;
pt.Point.Z = 0.2;
```

Transform the ROS message to the 'base_link' frame using the desired time saved from before.

```
tfpt = transform(tftree, 'base_link', pt, desiredTime);
```

Optional: You can also use `apply` with the stored `tform` to apply this transformation to the `pt` message.

```
tfpt2 = apply(tform, pt);
```

Shut down the ROS network.

```
roshutdown
```

```
Shutting down global node /matlab_global_node_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://192.168.154.1:6
```

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.

```
roshutdown
```

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

Input Arguments

tftree — ROS transformation tree

TransformationTree object handle

ROS transformation tree, specified as a TransformationTree object handle. You can create a transformation tree by calling the `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

Create a parameter tree and check for the 'MyParam' parameter.

```
ptree = rosparam;  
has(ptree, 'MyParam')  
ans =  
    0
```

Create a 'MyParam' parameter and verify that it exists.

```
set(ptree, 'MyParam', 'test')  
has(ptree, 'MyParam')  
ans =
```

1

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.

Code Generation Support:

Supports MATLAB Function block: Yes

“Code Generation Support, Usage Notes and Limitations”

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 — Cartesian coordinates

n-by- $(k-1)$ matrix

Cartesian coordinates, returned as an *n*-by- $(k-1)$ matrix, containing *n* points. Each row of **cart** represents a point in $(k-1)$ -dimensional space. *k* must be greater than or equal to 2.

Example: [0.8147 0.1270 0.6324; 0.9058 0.9134 0.0975]

See Also

[cart2hom](#)

Introduced in R2015a

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
```

- “Generate a standalone ROS node from Simulink®”

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`

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 =
```

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

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

```
Initializing global node /matlab_global_node_14356 with NodeURI http://192.168.154.1:6
```

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

```
d.AvailableNodes
```

```
ans =
```

```
1×2 cell array
```

```
'robotcontroller' 'robotcontroller2'
```

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

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

```
running =
```

```
logical
```

```
1
```

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

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

Shutting down global node /matlab_global_node_14356 with NodeURI http://192.168.154.1:

- “Generate a standalone ROS node from Simulink®”

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`

Introduced in R2016b

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

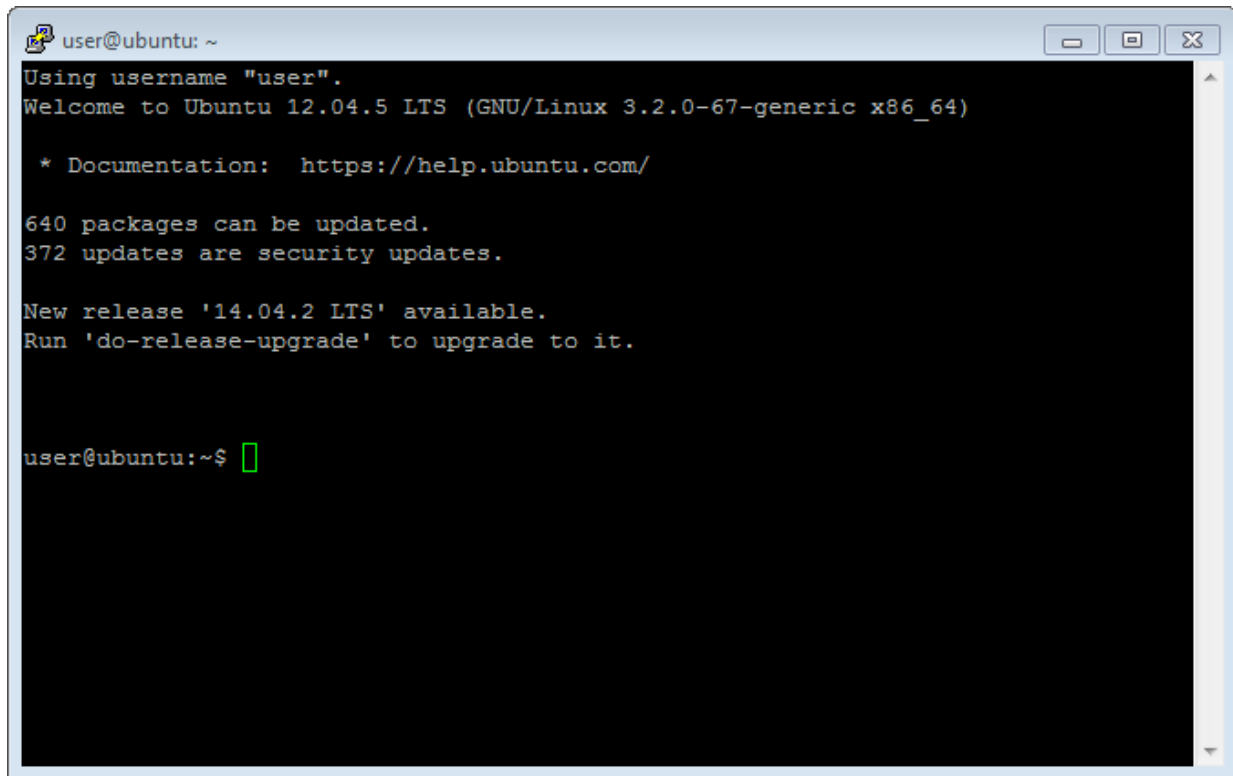
Connect to a ROS device and open the command shell on your host computer.

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);
```



```
user@ubuntu: ~  
Using username "user".  
Welcome to Ubuntu 12.04.5 LTS (GNU/Linux 3.2.0-67-generic x86_64)  
  
* Documentation:  https://help.ubuntu.com/  
  
640 packages can be updated.  
372 updates are security updates.  
  
New release '14.04.2 LTS' available.  
Run 'do-release-upgrade' to upgrade to it.  
  
user@ubuntu:~$ █
```

Input Arguments

device — ROS device

rosdevice object

ROS device, specified as a rosdevice object.

See Also

deleteFile | dir | getFile | putFile | rosdevice | system

Introduced in R2016b

plot

Display ROS laser scan messages on custom plot

Syntax

```
plot(scan)
plot(scan,Name,Value)
linehandle = plot( ___ )
```

Description

`plot(scan)` creates a line plot of the laser scan in xy -coordinates that is based on the input `LaserScan` object message. Axes are automatically scaled to the maximum range that the laser scanner supports.

`plot(scan,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`.

`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

```
plot(scan);
```

Plot Laser Scan with Maximum Range Specified

```
plot(scan, 'MaximumRange', 10);
```

Save Line Handle for Laser Scan Plot

```
linehandle = plot(scan);
```

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: 'MaximumRange', 5

'Parent' — Parent of axes

axes object

Parent of axes, specified as the comma-separated pair consisting of 'Parent' and an axes object in which the laser scan is drawn. By default, the laser scan is plotted in the currently active axes.

'MaximumRange' — Range of laser scan

scan.RangeMax (default) | scalar

Range of laser scan, specified as the comma-separated pair consisting of 'MaximumRange' and a scalar. When you specify this name-value pair argument, the

minimum and maximum x -axis 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.

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 MATLAB® current folder onto the ROS device. The destination folder must exist.

```
putFile(d,'test_file.txt','/home/user/test_folder')
```

Get a text file from the ROS device. You can get any file, not just ones added from MATLAB®. By default, the file is added to the MATLAB current folder.

```
getFile(d, '/home/user/test_folder/test_file.txt')
```

Delete the text file on the ROS device.

```
deleteFile(d, '/home/user/test_folder/test_file.txt')
```

Put, Get, and Delete Files on ROS Device Using Wildcards

Put a file from your host computer onto a ROS device, get it back, and then delete it. Use wildcards to search for all matching files.

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 all text files at the specified path onto the ROS device. The destination folder must exist.

```
putFile(d, 'C:/MATLAB/*.txt', '/home/user/test_folder')
```

Get all text files from the ROS device. You can get any files, not just ones added from MATLAB®. By default, the files are added to the MATLAB current folder.

```
getFile(d, '/home/user/test_folder/*.txt')
```

Delete all text files on the ROS device at the specified folder.

```
deleteFile(d, '/home/user/test_folder/*.txt')
```

Input Arguments

device — ROS device

rosdevice object

ROS device, specified as a rosdevice object.

localSource — Path and name of file on host computer

character vector

Path and name of the file on the host computer, specified as a character vector. You can use an absolute path or a path relative from the MATLAB current folder. Use the path and file naming conventions of the operating system on your host computer.

Example: 'C:\Work\.profile'

Data Types: char

remoteDestination — Destination folder path and optional file name

character vector

Destination folder path and optional file name, specified as a character vector. Specify a file name at the end of the destination path to copy with a custom file name. Use the Linux path and file naming conventions.

Example: '/home/user/.profile'

Data Types: char

See Also

`deleteFile` | `dir` | `getFile` | `openShell` | `rosdevice` | `system`

Introduced in R2016b

quat2axang

Convert quaternion to axis-angle rotation

Syntax

```
axang = quat2axang(quat)
```

Description

`axang = quat2axang(quat)` converts a quaternion, `quat`, to the equivalent axis-angle rotation, `axang`.

Code Generation Support:

Supports MATLAB Function block: Yes

“Code Generation Support, Usage Notes and Limitations”

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]

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'.

Code Generation Support:

Supports MATLAB Function block: Yes

“Code Generation Support, Usage Notes and Limitations”

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 — 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 — Axis rotation sequence

'ZYX' (default) | 'ZYZ'

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

- 'ZYX' (default) – The order of rotation angles is *z*-axis, *y*-axis, *x*-axis.
- 'ZYZ' – The order of rotation angles is *z*-axis, *y*-axis, *z*-axis.

Output Arguments

eu1 — 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]

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).

Code Generation Support:

Supports MATLAB Function block: Yes

“Code Generation Support, Usage Notes and Limitations”

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]

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).

Code Generation Support:

Supports MATLAB Function block: Yes

“Code Generation Support, Usage Notes and Limitations”

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]`

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

```
fieldnames = readAllFieldNames(pcloud);
```

Input Arguments

pcloud — Point cloud

`PointCloud2` object handle

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

Output Arguments

fieldnames — List of field names in `PointCloud2` object

cell array of character vectors

List of field names in `PointCloud2` object, returned as a cell array of character vectors. If no fields exist in the object, `fieldname` returns an empty cell array.

See Also

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 Data from 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

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

Read Message Data with Threshold

Threshold for occupied values is set to 65 and greater.

```
map = readBinaryOccupancyGrid(msg,65);
```

Read Message Data with Threshold and Unknown Value Replacement

```
map = readBinaryOccupancyGrid(msg,65,1);
```

Input Arguments

msg — 'nav_msgs/OccupancyGrid' ROS message

OccupancyGrid object handle

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

thresh — Threshold for occupied values

50 (default) | scalar

Threshold for occupied values, specified as a scalar. Any value greater than or equal to the threshold is set to occupied, 1. All other values are set to unoccupied, 0.

Data Types: double

va1 — Value to replace unknown values

0 (default) | 1

Value to replace unknown values, specified as either 0 or 1. Unknown message values (- 1) are set to the given value.

Data Types: double | logical

Output Arguments

map — Binary occupancy grid

BinaryOccupancyGrid object handle

Binary occupancy grid, returned as a `BinaryOccupancyGrid` object handle. `map` is converted from a `'nav_msgs/OccupancyGrid'` message on the ROS network. It is an object with a grid of binary values, where 1 indicates an occupied location and 0 indicates 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-clockwise around the positive z -axis, with the zero angle along the x -axis. `angles` is returned in radians and wrapped to the $[-\pi, \pi]$ interval.

Examples

Read Laser Scan and Convert to Cartesian Coordinates

```
cart = readCartesian(scan);
```

Read Laser Scan and Specify Scan Range

```
cart = readCartesian(scan, 'RangeLimit', [0 10]);
```

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-by-2 matrix in meters

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

angles — Scan angles for laser scan data

n-by-1 matrix in radians

Scan angles for laser scan data, returned as an *n*-by-1 matrix in radians. Angles are measured counter-clockwise around the positive *z*-axis, with the zero angle along the *x*-axis. `angles` is returned in radians and wrapped to the `[-pi, pi]` interval.

See Also

`plot` | `readScanAngles`

Introduced in R2015a

readField

Read point cloud data based on field name

Syntax

```
fielddata = readField(pcloud,fieldname)
```

Description

`fielddata = readField(pcloud,fieldname)` reads the point field from the 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 “Preserving Point Cloud Structure” on page 2-92.

Examples

Read x Coordinates for All Points

```
x = readField(pcloud, 'x');
```

Input Arguments

pcloud — Point cloud

PointCloud2 object handle

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

fieldname — Field name of point cloud data

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 “Preserving Point Cloud Structure” on page 2-92.

More About

Tips

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

```
[img,alpha] = readImage(obj);
```

Input Arguments

msg — ROS image message

Image object handle | `CompressedImage` object handle

`'sensor_msgs/Image'` or `'sensor_msgs/CompressedImage'` ROS image message, specified as an `Image` or `CompressedImage` object handle.

Output Arguments

img — Image

grayscale image matrix | RGB image matrix | *m*-by-*n*-by-3 array

Image, returned as a matrix representing a grayscale or RGB image or as *m*-by-*n*-by-3 array, depending on the sensor image.

alpha — Alpha channel

uint8 grayscale image

Alpha channel, returned as a uint8 grayscale image. If no alpha channel exists, alpha is empty.

More About

Tips

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 *MxN* 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 *MxNx3*. 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 *MxNx3*. 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 *MxNx1*. Each pixel value is represented as a **uint8**.
- **mono16** images are returned as grayscale images of size *MxNx1*. Each pixel value is represented as a **uint16**.
- **32fcX** images are returned as floating-point images of size *MxNxD*, 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 *MxNxD*, where *D* is 1, 2, 3, or 4. Each pixel value is represented as a **double**.
- **8ucX** images are returned as matrices of size *MxNxD*, where *D* is 1, 2, 3, or 4. Each pixel value is represented as a **uint8**.

- **8scX** images are returned as matrices of size $M \times N \times D$, where D is 1, 2, 3, or 4. Each pixel value is represented as a `int8`.
- **16ucX** images are returned as matrices of size $M \times N \times D$, where D is 1, 2, 3, or 4. Each pixel value is represented as a `int16`.
- **16scX** images are returned as matrices of size $M \times N \times D$, where D is 1, 2, 3, or 4. Each pixel value is represented as a `int16`.
- **32scX** images are returned as matrices of size $M \times N \times D$, where D is 1, 2, 3, or 4. Each pixel value is represented as a `int32`.
- **bayer_X** images are returned as either Bayer matrices of size $M \times N \times 1$, or as a converted image of size $M \times N \times 3$ (Image Processing Toolbox™ is required).

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

- `rgb8`, `rgba8`, `bgr8`, `bgra8`: `img` is an `rgb` image of size $M \times N \times 3$. The alpha channel is returned in `alpha`. Each output value is represented as a `uint8`.

See Also

`writeImage`

Introduced in R2015a

readMessages

Read messages from rosbag

Syntax

```
msgs = readMessages(bag)
msgs = readMessages(bag, rows)
```

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('ROSTestingWithRosbagsExample')), 'data', 'ex_mult...
```

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 “Preserving Point Cloud Structure” on page 2-99.

Examples

Read RGB Values from Point Cloud Object

```
rgb = readRGB(pcloud);
```

Input Arguments

pcloud — Point cloud

PointCloud2 object handle

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

Output Arguments

rgb — List of RGB values from point cloud

matrix

List of RGB values from point cloud, returned as a matrix. By default, this is an n -by-3 matrix. If the point cloud object being read has the `PreserveStructureOnRead` property set to true, the points are returned as an h -by- w -by-3 matrix. For more information, see “Preserving Point Cloud Structure” on page 2-99.

More About

Tips

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

Examples

Return Laser Scan Angles from Range Data

```
angles = readScanAngles(scan);
```

Input Arguments

scan — Laser scan message

LaserScan object handle

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

Output Arguments

angles — Scan angles for laser scan data

n -by-1 matrix in radians

Scan angles for laser scan data, returned as an n -by-1 matrix in radians. Angles are measured counter-clockwise around the positive z -axis, with the zero angle along the x -axis. `angles` is returned in radians and wrapped to the $[-\pi, \pi]$ interval.

See Also

`plot` | `readCartesian`

Introduced in R2015a

readXYZ

Extract XYZ coordinates from point cloud data

Syntax

```
xyz = readXYZ(pcloud)
```

Description

`xyz = readXYZ(pcloud)` extracts the `[x y z]` coordinates from all points in the 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 “Preserving Point Cloud Structure” on page 2-103.

Examples

Read XYZ Coordinates from Point Cloud

```
xyz = readXYZ(pcloud);
```

Input Arguments

pcloud — Point cloud

PointCloud2 object handle

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

Output Arguments

xyz — List of XYZ values from point cloud

matrix

List of XYZ values from point cloud, returned as a matrix. By default, this is a n -by-3 matrix. If the point cloud object being read has the `PreserveStructureOnRead` property set to true, the points are returned as an h -by- w -by-3 matrix. For more information, see “Preserving Point Cloud Structure” on page 2-103.

More About

Tips

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 Subscriber and Receive Data

```
laser = rossubscriber('/scan', rostype.sensor_msgs_LaserScan);
scan = receive(laser);
```

Receive Data with a Two Second Timeout

```
scan = receive(sub,2);
```

Input Arguments

sub — ROS subscriber

Subscriber object handle

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

timeout — Timeout for receiving a message

scalar in seconds

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

Output Arguments

msg — ROS message

Message object handle

ROS message, returned as a Message object handle.

See Also

`rosmessage` | `rossubscriber` | `rostopic`

Introduced in R2015a

roboticsAddons

Install add-ons for robotics

Syntax

`roboticsAddons`

Description

`roboticsAddons` allows you to download and install add-ons for Robotics System Toolbox.

Examples

Open Add-ons for Robotics System Toolbox

`roboticsAddons`

Introduced in R2016a

roboticsSupportPackages

Download and install support packages for Robotics System Toolbox

Compatibility

roboticsSupportPackages has been removed. Use roboticsAddons instead.

Syntax

roboticsSupportPackages

Description

roboticsSupportPackages opens the Support Package Installer to download and install support packages for Robotics System Toolbox. For more details, see “Install Robotics System Toolbox Add-ons”

Examples

Open Robotics System Toolbox Support Package Installer

roboticsSupportPackages

Introduced in R2015a

rosaction

Retrieve information about ROS actions

Syntax

```
rosaction list
rosaction info actionname
rosaction type actionname
```

```
actionlist = rosaction('list')
actioninfo = rosaction('info',actionname)
actiontype = rosaction('type',actionname)
```

Description

`rosaction list` returns a list of available ROS actions from the ROS network.

`rosaction info actionname` returns the action type, message types, action server, and action clients for the specified action name.

`rosaction type actionname` returns the action type for the specified action name.

`actionlist = rosaction('list')` returns a list of available ROS actions from the ROS network.

`actioninfo = rosaction('info',actionname)` returns a structure containing the action type, message types, action server, and action clients for the specified action name.

`actiontype = rosaction('type',actionname)` returns the action type for the specified action name.

Examples

Get Information About ROS Actions

Get information about ROS actions that are available from the ROS network. You must be connected to a ROS network using `roslint`.

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://192.168.154.1:6
```

List the actions available on the network. The only action setup on this network is the `'/fibonacci'` action.

```
rosaction list
```

```
/fibonacci
```

Get information about a specific ROS action type. The action type, message types, action server, and clients are displayed.

```
rosaction info /fibonacci
```

```
Action Type: actionlib_tutorials/Fibonacci
```

```
Goal Message Type: actionlib_tutorials/FibonacciGoal
```

```
Feedback Message Type: actionlib_tutorials/FibonacciFeedback
```

```
Result Message Type: actionlib_tutorials/FibonacciResult
```

```
Action Server:
```

```
* /fibonacci (http://192.168.154.131:38213/)
```

```
Action Clients: None
```

Disconnect from the ROS network.

```
roshutdown
```

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

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 `roaction('list')` outputs.

Output Arguments

actionlist — List of actions available

cell array of character vectors

List of actions available on the ROS network, returned as a cell array of character vectors.

actioninfo — Information about a ROS action

structure

Information about a ROS action, returned as a structure. `actioninfo` contains the following fields:

- `ActionType`
- `GoalMessageType`
- `FeedbackMessageType`
- `ResultMessageType`
- `ActionServer`
- `ActionClients`

For more information about ROS actions, see “ROS Actions Overview”.

actiontype — Type of ROS action

character vector

Type of ROS action, returned as a character vector.

More About

- “ROS Actions Overview”
- “Move a Turtlebot Robot Using ROS Actions”

See Also

`cancelGoal` | `rosmesssage` | `rostopic` | `sendGoal` | `waitForServer`

Introduced in R2016b

roactionclient

Create ROS action client

Syntax

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

Description

`client = roactionclient(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 `roactionclient` 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 = roactionclient(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] = roactionclient(___)` 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:

```
roslaunch actionlib_tutorials fibonacci_server
```

Connect to a ROS network. You must be connected to a ROS network to gather information about what actions are available. Replace `ipaddress` with your network address.

```
ipaddress = '192.168.154.131';  
rosinit(ipaddress)
```

```
Initializing global node /matlab_global_node_68978 with NodeURI http://192.168.154.1:5
```

List actions available on the network. The only action setup on this network is the `'/fibonacci'` action.

```
rosaction list  
  
/fibonacci
```

Create an action client. Specify the action name.

```
[actClient,goalMsg] = rosactionclient('/fibonacci');
```

Wait for action client to connect to server.

```
waitForServer(actClient);
```

The fibonacci action will calculate the fibonacci sequence for a given order specified in the goal message. The goal message was returned when creating the action client and can be modified to send goals to the ROS action server.

```
goalMsg.Order = 8
```

```
goalMsg =
```

```
ROS FibonacciGoal message with properties:
```

```
  MessageType: 'actionlib_tutorials/FibonacciGoal'  
  Order: 8
```

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

Send goal and wait for its completion. Specify a timeout of 10 seconds to complete the action.

```
[resultMsg,resultState] = sendGoalAndWait(actClient,goalMsg,10)
```

```
Goal active
```

```
Feedback:
```

```
  Sequence : [0, 1, 1]
```

```
Feedback:
```

```
  Sequence : [0, 1, 1, 2]
```

```
Feedback:
```

```
  Sequence : [0, 1, 1, 2, 3]
```

```
Feedback:
```

```
  Sequence : [0, 1, 1, 2, 3, 5]
```

```
Feedback:
```

```
  Sequence : [0, 1, 1, 2, 3, 5, 8]
```

```
Feedback:
```

```
  Sequence : [0, 1, 1, 2, 3, 5, 8, 13]
```

```
Feedback:
```

```
  Sequence : [0, 1, 1, 2, 3, 5, 8, 13, 21]
```

```
Feedback:
```

```
  Sequence : [0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
```

```
Final state succeeded with result:
```

```
  Sequence : [0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
```

```
resultMsg =
```

```
  ROS FibonacciResult message with properties:
```

```
    MessageType: 'actionlib_tutorials/FibonacciResult'
```

```
    Sequence: [10×1 int32]
```

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

```
resultState =
```

```
  1×9 char array
```

```
succeeded
```

Disconnect from the ROS network.

```
roshutdown
```


Shutting down global node /matlab_global_node_68978 with NodeURI http://192.168.154.1:5

Send and Cancel ROS Action Goals

Send and cancel goals for ROS actions. First, setup a ROS action client. Then send a goal message with modified parameters. Finally, cancel your goal and all goals on the action server.

Connect to a ROS network with a specified IP address. Create a ROS action client connected using `rosactionclient`. Specify the action name. Wait for the client to be connected to the server.

```
rosinit('192.168.154.131')
[actClient,goalMsg] = rosactionclient('/fibonacci');
waitForServer(actClient);
```

Initializing global node /matlab_global_node_40739 with NodeURI http://192.168.154.1:5

Send a goal message with modified parameters. Wait for the goal to finish executing.

```
goalMsg.Order = 4;
sendGoalAndWait(actClient,goalMsg)
```

```
Goal active
Feedback:
  Sequence : [0, 1, 1]
Feedback:
  Sequence : [0, 1, 1, 2]
Feedback:
  Sequence : [0, 1, 1, 2, 3]
Feedback:
  Sequence : [0, 1, 1, 2, 3, 5]
```

```
ans =
```

```
ROS FibonacciResult message with properties:
```

```
  MessageType: 'actionlib_tutorials/FibonacciResult'
  Sequence: [6×1 int32]
```

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

Send a new goal message without waiting.

```
goalMsg.Order = 5;  
sendGoal(actClient,goalMsg)
```

Cancel the goal on the ROS action client, `actClient`.

```
cancelGoal(actClient)
```

Cancel all the goals on the action server that `actClient` is connected to.

```
cancelAllGoals(actClient)
```

Delete the action client.

```
delete(actClient)
```

Disconnect from the ROS network.

```
roshutdown
```

```
Shutting down global node /matlab_global_node_40739 with NodeURI http://192.168.154.1:8080
```

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 `roaction('list')` outputs.

actiontype — Type of ROS action

character vector

Type of ROS action, returned as a character vector.

Output Arguments

client — 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 — 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`.

More About

- “ROS Actions Overview”
- “Move a Turtlebot Robot Using ROS Actions”

See Also

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

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('ROSTestingWithRosbagsExample')), 'data', 'ex_mult...
```

Retrieve information from the rosbag

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

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

```
bagselect2 = select(bagselect, 'Time', ...
```

```
[bagselct.StartTime bagselct.StartTime + 1], 'Topic', '/odom');
```

Input Arguments

filename — Name of rosvbag file and its path

character vector

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

Output Arguments

bag — Selection of rosvbag messages

BagSelection object handle

Selection of rosvbag messages, returned as a BagSelection object handle.

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: 1.4726e+09
        Nsec: 641000000
```

```
timeFuture =
```

```
    ROS Time with properties:
```

```
        Sec: 1.4726e+09
        Nsec: 143000000
```


Input Arguments

totalSecs — Total time

0 (default) | scalar

Total time, specified as a floating-point scalar. The integer portion is set to the `Sec` property with the remainder applied to `Nsec` property of the `Duration` object.

secs — Whole seconds

0 (default) | integer

Whole seconds, specified as an integer. This value is directly set to the `Sec` property of the `Duration` object.

Note: The maximum and minimum values for `secs` are `[-2147483648, 2147483647]`.

nsecs — Nanoseconds

0 (default) | positive integer

Nanoseconds, specified as a positive integer. This value is directly set to the `Nsec` property of the `Duration` object unless it is greater than or equal to 10^9 . The value is then wrapped and the remainders are added to the value of `secs`.

Output Arguments

dur — Duration

ROS `Duration` object

Duration, returned as a ROS `Duration` object with `Sec` and `Nsec` properties.

See Also

`rosmmessage` | `rostime` | `seconds`

Introduced in R2016b

rosgenmsg

Generate custom messages from ROS definitions

Syntax

```
rosgenmsg(folderpath)
```

Description

`rosgenmsg(folderpath)` generates ROS custom messages in MATLAB by reading ROS custom message and service definitions in the specified folder path. The function expects ROS package folders inside the folder path. These packages contain the message definitions in `.msg` files and the service definitions in `.srv` files. Also, the packages require a `package.xml` file to define its contents.

After calling this function, you can send and receive your custom messages in MATLAB like all other supported messages. You can create these messages using `rosmmessage` or view the list of messages by calling `rosmmsg list`.

Examples

Generate MATLAB Code for ROS Custom Messages

```
folderpath = 'C:/Users/user1/Documents/robot_custom_msg/';  
rosgenmsg(folderpath)
```

- “Create Custom Messages from ROS Package”

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.

More About

- “ROS Custom Message Support”
- ROS Tutorials: Defining Custom Messages
- ROS Tutorials: Creating a ROS msg and srv

See Also

roboticsAddons

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.

Examples

Start ROS Core and Global Node

```
rosinit
```

```
Initializing ROS master on http://hostname.mathworks.com:11311/.  
Initializing global node /matlab_global_node_9152 with NodeURI http://hostname:54194/
```

Start Node and Connect to ROS Master at Specified IP Address

```
rosinit('192.168.1.10')
```

```
Initializing global node /matlab_tped50a5c2_4448_4d11_a523_9829a6b3b5af with NodeURI ht
```

Start Global Node at Given IP and Node Name

```
rosinit('192.168.1.10', 'NodeHost', '192.168.1.1', 'NodeName', '/test_node')
```

```
Initializing global node /test_node with NodeURI http://192.168.1.1:64053/
```

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 `roslaunch` is registered on the ROS network with this name.

Example: `'NodeName', '/test_node'`

See Also

`roslaunch`

Introduced in R2015a

rosmesssage

Create ROS messages

Syntax

```
msg = rosmesssage(messagetype)
```

```
msg = rosmesssage(pub)
```

```
msg = rosmesssage(sub)
```

```
msg = rosmesssage(client)
```

```
msg = rosmesssage(server)
```

Description

`msg = rosmesssage(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 `rosmesssage('list')`. To avoid errors in entering the message type, you can use `rostype` with tab completion to browse the list of all available types.

`msg = rosmesssage(pub)` creates an empty message determined by the topic published by `pub`.

`msg = rosmesssage(sub)` creates an empty message determined by the subscribed topic of `sub`.

`msg = rosmesssage(client)` creates an empty message determined by the service associated with `client`.

`msg = rosmesssage(server)` creates an empty message determined by the service type of `server`.

Examples

Create Empty String Message

```
strMsg = rosmesssage('std_msgs/String')
```

```
strMsg =
```

```
ROS String message with properties:
```

```
  MessageType: 'std_msgs/String'  
    Data: ''
```

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

Create Laser Scan Message Using rostype

```
scan = rosmesssage(rostype.sensor_msgs_LaserScan)
```

```
scan =
```

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

```
  MessageType: 'sensor_msgs/LaserScan'  
    Header: [1×1 Header]  
    AngleMin: 0  
    AngleMax: 0  
AngleIncrement: 0  
TimeIncrement: 0  
    ScanTime: 0  
    RangeMin: 0  
    RangeMax: 0  
    Ranges: [0×1 single]  
Intensities: [0×1 single]
```

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

Create Message to Publish using ROS Publisher

```
chatpub = rospublisher('/ chatter', 'std_msgs/String');  
chatmsg = rosmesssage(chatpub);
```

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 `rosmg('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 — ROS message

`Message` object handle

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

More About

- “Built-In Message Support”

See Also

roboticsSupportPackages | rosmg | rostype

Introduced in R2015a

rosmg

Retrieve information about ROS messages and message types

Syntax

```
rosmg show msgtype  
rosmg md5 msgtype  
rosmg list
```

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

Description

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

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

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

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

`msgmd5 = rosmg('md5', msgtype)` returns the 'MD5' checksum of the `msgtype` message as a character vector.

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

Examples

Retrieve Message Type Definition

```
msgInfo = rosmg('show', 'geometry_msgs/Point')
```

```
msgInfo =  
  
% This contains the position of a point in free space  
double X  
double Y  
double Z
```

Get the MD5 Checksum of Message Type

```
msgMd5 = rosmg('md5', 'geometry_msgs/Point')
```

```
msgMd5 =  
  
4a842b65f413084dc2b10fb484ea7f17
```

Input Arguments

msgtype — ROS message type
character vector

ROS message type, specified as a character vector. `msgType` must be a valid ROS message type from ROS that MATLAB supports.

Example: 'std_msgs/Int8'

Output Arguments

msginfo — Details of message definition
character vector

Details of the information inside the ROS message definition, returned as a character vector.

msgmd5 — MD5 checksum hash value
character vector

MD5 checksum hash value, returned as a character vector. The MD5 output is a character vector representation of the 16-byte hash value that follows the MD5 standard.

msglist — List of all message types available in MATLAB

cell array of character vectors

List of all message types available in MATLAB, returned as a cell array of character vectors.

Introduced in R2015a

rostopic

Retrieve information about ROS network nodes

Syntax

```
rostopic list
rostopic info nodename
rostopic ping nodename

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

Description

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

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

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

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

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

`rostopic('ping',nodename)` pings a specific node, `nodename` and displays the response time.

Examples

Retrieve List of ROS Nodes

```
rostopic list
```

```
/bumper2pointcloud
/cmd_vel_mux
/depthimage_to_laserscan
/gazebo
/laserscan_nodelet_manager
/matlab_tp8cc35a0e_35fd_4f70_9886_9e489b95b611
/mobile_base_nodelet_manager
/robot_state_publisher
/rosout
```

Retrieve ROS Node Info

```
nodeinfo = rosnode('info', '/robot_state_publisher')
```

```
nodeinfo =
```

```
      NodeName: '/robot_state_publisher'
      URI: 'http://192.168.154.132:58140/'
      Publications: [2x1 struct]
      Subscriptions: [2x1 struct]
      Services: [2x1 struct]
```

Ping ROS Node

```
rosnode('ping', '/robot_state_publisher')
```

```
Pinging the /robot_state_publisher node with a timeout of 3 seconds.
Ping reply from http://192.168.154.132:58140/, response time = 2.920 ms.
Ping reply from http://192.168.154.132:58140/, response time = 2.138 ms.
Ping reply from http://192.168.154.132:58140/, response time = 2.194 ms.
Ping reply from http://192.168.154.132:58140/, response time = 4.607 ms.
Ping average time: 2.965 ms
```

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`.

odelist — List of node names available

cell array of character vectors

List of node names available, returned as a cell array of character vectors.

See Also

`rosinit` | `rostopic`

Introduced in R2015a

rosparam

Access ROS parameter server values

Syntax

```
ptree = rosparam
```

Description

`ptree = rosparam` creates a parameter tree object, `ptree`. Once `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.

Supported parameter values are:

- `int32`
- `logical`
- `double`
- `character vector`
- `cell array`

Examples

Create Parameter Tree Object and View Parameters

```
ptree = rosparam
```

```
ptree =
```

```
ParameterTree with properties:
  AvailableParameters: {40x1 cell}
ptree.AvailableParameters
ans =
    '/bumper2pointcloud/pointcloud_radius'
    '/camera/imager_rate'
    '/camera/rgb/image_raw/compressed/format'
    ...
```

Output Arguments

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`.

Limitations

Base64–encoded binary data, iso8601 data, and dictionaries from ROS are not supported.

See Also

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

Introduced in R2015a

rospublisher

Publish messages on a topic

Syntax

```
pub = rospublisher(topicname)
pub = rospublisher(topicname,msgtype)
pub = rospublisher( ____,Name,Value)
[pub,msg] = rospublisher( ____ )

rospublisher(topicname,msg)
```

Description

`pub = rospublisher(topicname)` creates a publisher, `pub`, for a topic, `topicname`, that already exists on the ROS master topic list. 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. To see a list of available topic names, at the MATLAB command prompt, type `rostopic list/`

`pub = rospublisher(topicname,msgtype)` creates a publisher for a topic and adds that topic to the ROS master topic list. 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. If `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 argument 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.

`rospublisher(topicname, msg)` publishes a message, `msg`, to the specified topic without creating a publisher.

Examples

Create a Publisher with Specified Message Type and Send Data

```
chatpub = rospublisher('/chatter', 'std_msgs/String');  
msg = rosmessage(chatpub);  
msg.Data = 'test phrase';  
send(chatpub, msg);
```

Send Single Message Without Creating a Publisher

```
rospublisher('/chatter', msg)
```

Input Arguments

topicname — ROS topic name

character vector

ROS topic name, specified as a character vector.

msgtype — Message type for ROS topic

character vector

ROS message type, specified as a character vector.

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: 'IsLatching', false

'IsLatching' — Latch property

true (default) | logical

Latch property, specified as the comma-separated pair consisting of `'isLatching'` and a logical. If enabled, latch mode saves the last message sent by the publisher and resends it to new subscribers. By default, latch mode is disabled (`false`). To enable latch mode, set `'IsLatching'` to `true`.

Output Arguments

pub — ROS publisher

Publisher object handle

ROS publisher, returned as a `Publisher` object handle.

Properties: When you call `rospublisher`, `pub` is returned as a `Publisher` object with the following properties:

- `TopicName` (read-only): Name of the published topic
- `MessageType` (read-only): Message type of published messages
- `IsLatching`: Indicates if publisher is latching
- `NumSubscribers` (read-only): Number of current subscribers for the published topic

To access these properties, use `pub.TopicName`, `pub.MessageType`, `pub.IsLatching`, or `pub.NumSubscribers`.

msg — ROS message

Message object handle

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

See Also

`rosmmessage` | `rossubscriber`

Introduced in R2015a

rostrate

Execute loop at fixed frequency

Syntax

```
rate = rostrate(desiredRate)
```

Description

`rate = rostrate(desiredRate)` creates a `robotics.ros.Rate` object, which lets you 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`.

Examples

Run Loop at Fixed Rate

Initialize the ROS master and node.

```
rosinit;
```

Create a rate object that runs at 1 Hz.

```
r = rostrate(1);
```

Start loop that prints iteration and time elapsed.

```
for i = 1:10;
    time = r.TotalElapsedTime;
    fprintf('Iteration: %d - Time Elapsed: %f\n',i,time)
    sleep(r)
end
```

```
Iteration: 1 - Time Elapsed: 5.344408
Iteration: 2 - Time Elapsed: 5.345697
Iteration: 3 - Time Elapsed: 6.345708
```

```
Iteration: 4 - Time Elapsed: 7.345741
Iteration: 5 - Time Elapsed: 8.345797
Iteration: 6 - Time Elapsed: 9.345820
Iteration: 7 - Time Elapsed: 10.345850
Iteration: 8 - Time Elapsed: 11.344907
Iteration: 9 - Time Elapsed: 12.344948
Iteration: 10 - Time Elapsed: 13.344967
```

Notice that each iteration executes at a 1-second interval.

Shut down the ROS network.

```
roshutdown;
```

- “Execute Code at a Fixed-Rate”

Input Arguments

desiredRate — Desired execution rate

scalar

Desired execution rate of loop, specified as a scalar in Hz. When using `robotics.Rate.waitFor`, the loop operates every `desiredRate` seconds, unless the loop takes longer. It then begins the next loop based on the specified `OverrunAction`.

Note: The performance of the `robotics.ros.Rate` object and the ability to maintain the desired rate depends on the publishing of the clock information in ROS.

Output Arguments

rate — Rate object

handle

Rate object, specified as an object handle. This object contains the information for the `DesiredRate` and other information about the execution.

See Also

`robotics.Rate.waitFor`

Introduced in R2016a

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

```
rosservice list  
  
/bumper2pointcloud/get_loggers  
/bumper2pointcloud/set_logger_level  
/camera/rgb/image_raw/compressed/set_parameters  
...
```

Get Information, Type and URI for ROS Service

Get the service information.

```
svcinfo = rosservice('info', 'gazebo/pause_physics')  
  
svcinfo =  
  
    Node: '/gazebo'  
    URI: 'rosrpc://192.168.154.132:33953'  
    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://192.168.154.132:33953
```

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

roshutdown

Shut down ROS system

Syntax

```
roshutdown
```

Description

`roshutdown` shuts down the global node and, if it is running, the ROS master. When you finish working with the ROS network, use `roshutdown` 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 `roshutdown`, any ROS entities that depend on the global node, for example, subscribers created with `rossubscriber`, are deleted and become unstable.

Examples

Shut Down Global ROS Node

```
roshutdown
```

```
Shutting down global node /matlab_global_node_9220 with NodeURI http://hostname:54335/  
Shutting down ROS master on http://hostname.mathworks.com:11311/.
```

See Also

`rosinit`

Introduced in R2015a

rossubscriber

Subscribe to messages on a topic

Syntax

```
sub = rossubscriber(topicname)
sub = rossubscriber(topicname,msgtype)

sub = rossubscriber(topicname,callback)
sub = rossubscriber(topicname, msgtype,callback)

sub = rossubscriber( ____,Name,Value)
```

Description

`sub = rossubscriber(topicname)` subscribes to a topic with name `topicname`. If the ROS master topic list includes `topicname`, this syntax returns a subscriber object handle, `sub`. If the ROS master topic list does not include the topic, this syntax displays an error. `rossubscriber` enables you to transfer data by subscribing to messages. 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, `msgtype`. If the topic list on the ROS master does not include a topic with that specified name and type, a topic with the specific name and type is added to the topic list. Use this syntax to avoid errors when it is possible for the subscriber to subscribe 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, `msgtype`.

`sub = rossubscriber(____, Name, Value)` provides additional options specified by one or more `Name, Value` pair arguments using any of the argument 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.

Examples

Create Subscriber

```
sub = rossubscriber('/scan');
```

Create Subscriber Using rostype for Message Type

Create the subscriber.

```
sub = rossubscriber('/scan', rostype.sensor_msgs_LaserScan);
```

Get the last message from the topic.

```
scan = sub.LatestMessage;
```

Wait to receive the next message and store in `scan`.

```
scan = receive(sub);
```

Create Subscriber Using Callback Function

Create the publisher and subscriber.

```
chatpub = rospublisher('/chatter', rostype.std_msgs_String);  
chatsub = rossubscriber('/chatter', @testCallback);
```

Change the Callback Function of Existing Subscriber

```
chatsub = rossubscriber('/chatter', @testCallback);  
userData = [5 1; 1 5];
```

```
chatsub.NewMessageFcn = {@func1, userData};
```

Create Subscriber with Specified Buffer Size

```
chatbuf = rossubscriber('/chatter', 'BufferSize', 5);
```

Input Arguments

topicname — ROS topic name

character vector

ROS topic name, specified as a character vector.

msgtype — Message type for ROS topic

character vector

Message type for ROS topic, specified as a character vector.

callback — Callback function

function handle | cell array

Callback function, 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)
```

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.

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: 'BufferSize',25

'BufferSize' — Buffer size

1 (default) | scalar

Buffer size, specified as the comma-separated pair consisting of 'BufferSize' and a scalar. If messages arrive faster and than your callback can process them, they will be deleted once the incoming queue is full.

'NewMessageFcn' — Callback property

function handle | cell array

Callback property, specified as a function handle or cell array. In the first element of the cell array, specify either a function handle or a 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)
```

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.

Output Arguments

sub — ROS subscriber

Subscriber object handle

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

Properties: When you call `rossubscriber`, `sub` is returned as a `Subscriber` object with the following properties:

- `TopicName` (read-only): Name of the published topic
- `MessageType` (read-only): Message type of published messages
- `LatestMessage` (read-only): Latest message received
- `BufferSize` (read-only): Buffer size of the incoming queue

- `NewMessageFcn`: Callback property for subscriber callbacks

To access these properties, use `sub.TopicName`, `sub.MessageType`, `sub.LatestMessage`, `sub.BufferSize`, or `sub.NewMessageFcn`.

See Also

`rosmessage` | `rospublisher`

Introduced in R2015a

rossvcclient

Create ROS service client

Syntax

```
client = rossvcclient(servicename)  
client = rossvcclient(servicename,Name,Value)
```

```
[client,reqmsg] = rossvcclient( ___ )
```

Description

`client = rossvcclient(servicename)` creates a service client that connects to, and gets its service type from, a service server. This command syntax blocks the current MATLAB program from running until it can connect to the service server.

Use `rossvcclient` to create a ROS service client. 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.

`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`.

`[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.

Examples

Create Service Client and Wait to Connect to Service

```
client = rossvcclient('/gazebo/get_model_state');
```

Connect to Service Server with Timeout

```
client = rossvcclient('/gazebo/get_model_state', 'Timeout', 3);
```

Create Service Request Message and Call for Response

Create the service request message.

```
request = rosmessage(client);
```

Send the service request and wait for a response.

```
request.ModelName = 'SomeModel';
response = call(client, request);
```

Create a Service Client and Get a Request Message

```
[client, reqmsg] = rossvcclient('/gazebo/get_model_state');
```

Input Arguments

servicename — Service name

character vector

Service name, specified as a character vector. To access information about active services, such as the service name, use the `rosservice` function.

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', 10

'Timeout' — Timeout period in seconds

`inf` (default) | scalar

Timeout period in seconds, specified as a scalar. If the service client does not connect to the service server by the end of the timeout period, `rossvcclient` displays an error message, and MATLAB keeps running the current program. The default value of `inf` blocks MATLAB from running the current program until the service client is connected to the service server.

Output Arguments

`client` — ROS service client

`ServiceClient` object handle

ROS service client, returned as a `ServiceClient` object handle. This service client uses a persistent connection to send requests to, and receive responses from, a ROS service server.

ROS message, returned as a `Message` object handle that matches the request type of the service.

Properties: When you call `rossubscriber`, `client` is returned as a `ServiceClient` object with the following properties:

- `ServerName` (read-only): Name of the service
- `ServiceType` (read-only): Type of the service

To access these properties, use `client.ServerName` or `client.ServiceType`.

`reqmsg` — ROS message

`Message` object handle

See Also

`call` | `rosservice` | `rossvcserver`

Introduced in R2015a

rossvcserver

Create ROS service server

Syntax

```
server = rossvcserver(servicename,svctype)
server = rossvcserver(servicename,svctype,callback)

servicename = rossvcserver(servicename,svctype,Name,Value)
```

Description

`server = rossvcserver(servicename,svctype)` creates a service server object of type `svctype` available in the ROS network under the name `servicename`. The service object cannot respond to service requests until you specify a function handle `callback`.

Use `rossvcserver` to create a ROS service server that can receive requests from, and send responses to, a ROS service client. The service server must exist before creating the service client. When you create the client, it establishes a connection to the server. The connection persists while both client and server exist and can reach each other.

`server = rossvcserver(servicename,svctype,callback)` specifies the function handle `callback`, that constructs a response when the server receives a request. `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.

`servicename = rossvcserver(servicename,svctype,Name,Value)` provides additional options specified by one or more `Name, Value` pair arguments using any of the argument 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.

Examples

Create Service Server

```
server = rossvcserver('/gazebo/get_model_state', rostype.gazebo_msgs_GetModelState)
```

Create Service Server with Callback Function and User Data

Create user data.

```
userData = randi(20);
```

Create a service server.

```
server = rossvcserver('/gazebo/get_model_state2', rostype.gazebo_msgs_GetModelState, {@func1, userData});
```

Change the callback for a incoming service calls.

```
server.NewRequestFcn = @func2;
```

Input Arguments

servicename — Service name

character vector

Service name, specified as a character vector. You can access information about active services, such as the service name, using `rosservice`.

svctype — Service message type

character vector

Service message type, specified as a character vector. You can access information about service message types using `rostype`. Use tab completion to select the message.

callback — Callback function and inputs

function handle | cell array

Callback function and inputs, specified as a 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. The service server callback function requires at least three input arguments and one output. The first argument, `server`, is the associated service server object. The second argument, `reqmsg`, is the request message object sent by the service client. The third argument is the default response message object, `defaultrespmsg`. Use `defaultrespmsg` as a starting point for constructing the function output `response`, which is sent back to the service client.

```
function response = serviceCallback(server,reqmsg,defaultrespmsg)
    response = defaultrespmsg;
    % Build the response message here
end
```

While setting the callback, to construct a callback that accepts additional parameters, use a cell array that includes the function handle callback and the parameters.

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: `'NewMessageFcn',{@func1,userDate}`

'NewMessageFcn' — Callback property

function handle | cell array

Callback property, specified as a 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. The service server callback function requires at least three input arguments and one output. The first argument, `server`, is the associated service server object. The second argument, `reqmsg`, is the request message object sent by the service client. The third argument is the default response message object, `defaultrespmsg`. Use `defaultrespmsg` as a starting point for constructing the function output `response`, which is sent back to the service client.

```
function response = serviceCallback(server,reqmsg,defaultrespmsg)
    response = defaultrespmsg;
    % Build the response message here
end
```

While setting the callback, to construct a callback that accepts additional parameters, use a cell array that includes the function handle callback and the parameters.

Output Arguments

server — Service server

`ServiceServer` object handle

Service server, returned as a `ServiceServer` object handle. This service server registers with the ROS master, which enables service clients to send it requests.

Properties: When you call `rossubscriber`, `server` is returned as a `ServiceServer` object with the following properties:

- `ServerName` (read-only): Name of the service
- `ServiceType` (read-only): Type of the service
- `NewRequestFcn`: Callback property for service request callbacks

To access these properties, use `client.ServerName`, `client.ServerType`, or `client.NewRequestFcn`.

See Also

`rossvcclient`

Introduced in R2015a

rostf

Access ROS transformations

Syntax

```
tfTree = rostf
```

Description

`tfTree = rostf` creates a ROS transformation tree object. The object 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
```

MATLAB can only keep track of the most current information between different frames. ROS tf allows for “time-traveling” or retrieving transformations from specific time instances.

Examples

Create Transformation Tree

```
tree = rostf;
```

Output Arguments

tfTree — ROS transformation tree

TransformationTree object handle

ROS transformation tree, returned as a `TransformationTree` object handle.

See Also

`getTransform` | `transform`

Introduced in R2015a

rostime

Access ROS time functionality

Syntax

```
time = rostime('now')  
[time,issimtime] = rostime('now')  
time = rostime('now','system')
```

```
time = rostime(totalSecs)  
time = rostime(secs,nsecs)
```

Description

`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.

`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`.

Examples

Show Current ROS Time

```
t = rostime('now')
t =
    ROS Time with properties:
        Sec: 1417812065
        Nsec: 368000000
```

Indicate Whether Time Is System Time

```
[t,issim] = rostime('now');
t =
    ROS Time with properties:
        Sec: 1417812173
        Nsec: 171000000

issim =
    0
```

Timestamp Message Data

```
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 MATLAB® standard time. The ROS Time object is first converted to a double in seconds, then to the specified MATLAB time.

```

% Sets up ROS network and stores ROS time
rosinit
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://bat5731win64:11311/.
Initializing global node /matlab_global_node_84957 with NodeURI http://bat5731win64:62

time =

    datetime

    30-Aug-2016 16:57:26

Shutting down global node /matlab_global_node_84957 with NodeURI http://bat5731win64:62
Shutting down ROS master on http://bat5731win64:11311/.

```

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

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.

secs — Whole seconds

0 (default) | positive integer

Whole seconds, specified as a positive integer. This value is directly set to the `Sec` property of the `Time` object.

Note: The maximum and minimum values for `secs` are [0, 4294967294].

nsecs — Nanoseconds

0 (default) | positive integer

Nanoseconds, specified as a positive integer. This value is directly set to the `NSEC` property of the `Time` 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

time — Current ROS or system time

`Time` object handle

ROS or system time, returned 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.

issimtime — System time indicator

boolean

System time indicator, returned as a boolean. This boolean indicates whether the `time` argument is in simulation time (`true`) or system time (`false`), returned as a Boolean.

See Also

rosduration | rosmesssage | seconds

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 Topics Available on ROS Master

```
rostopic list

/camera/depth/camera_info
/camera/depth/image_raw
/camera/depth/points
/camera/parameter_descriptions
...
```

Get Topic Info for Specified ROS Topic

```
topicinfo = rostopic('info', 'camera/depth/points')

topicinfo =

    MessageType: 'sensor_msgs/PointCloud2'
    Publishers: [1x1 struct]
    Subscribers: [0x0 struct]
```

Get Message Type for Specified ROS Topic

```
msgtype = rostopic('type', 'camera/depth/points')

msgtype =

sensor_msgs/PointCloud2
```

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 `rosmmessage` as an input argument.

Examples

Create ROS Message Type and ROS Message

Create Message Type String

```
t = rostype.std_msgs_String
```

```
t =
```

```
std_msgs/String
```

Create ROS Message from ROS Type

```
msg = rosmmessage(rostype.std_msgs_String)
```

```
msg =
```

ROS String message with properties:

```
MessageType: 'std_msgs/String'  
Data: ''
```

Use `showdetails` to show the contents of the message

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.

Code Generation Support:

Supports MATLAB Function block: Yes

“Code Generation Support, Usage Notes and Limitations”

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 is orthonormal. The input rotation matrix must be in the premultiply form for rotations.

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

Output Arguments

axang — Rotation given in axis-angle form

n -by-4 matrix

Rotation given in axis-angle form, returned as an n -by-4 matrix of n axis-angle rotations. The first three elements of every row specify the rotation axis, and the last element defines the rotation angle (in radians).

Example: [1 0 0 pi/2]

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'.

Code Generation Support:

Supports MATLAB Function block: Yes

“Code Generation Support, Usage Notes and Limitations”

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.

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

sequence — Axis rotation sequence

'ZYX' (default) | 'ZYZ'

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

- 'ZYX' (default) — The order of rotation angles is *z*-axis, *y*-axis, *x*-axis.
- 'ZYZ' — The order of rotation angles is *z*-axis, *y*-axis, *z*-axis.

Output Arguments

eu1 — 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]

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.

Code Generation Support:

Supports MATLAB Function block: Yes

“Code Generation Support, Usage Notes and Limitations”

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.

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

Output Arguments

quat — Unit quaternion

n -by-4 matrix

Unit quaternion, returned as an n -by-4 matrix containing n quaternions. Each quaternion, one per row, is of the form $q = [w \ x \ y \ z]$, with w as the scalar number.

Example: [0.7071 0.7071 0 0]

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).

Code Generation Support:

Supports MATLAB Function block: Yes

“Code Generation Support, Usage Notes and Limitations”

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.

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]`

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
```

- “Generate a standalone ROS node from Simulink®”

Input Arguments

device — ROS device

rosdevice object

ROS device, specified as a rosdevice object.

See Also

isCoreRunning | rosdevice | stopCore

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 =
```

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

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

```
Initializing global node /matlab_global_node_14356 with NodeURI http://192.168.154.1:6
```

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

```
d.AvailableNodes
```

```
ans =
```

```
1×2 cell array
'robotcontroller' 'robotcontroller2'
```

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

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

```
running =
```

```
logical
```

```
1
```

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

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

```
Shutting down global node /matlab_global_node_14356 with NodeURI http://192.168.154.1:
```

Run Multiple ROS Nodes

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

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

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

```
ipaddress = '192.168.154.131';  
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 MATLAB® 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://192.168.154.1:6
```

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

```
d.AvailableNodes
```

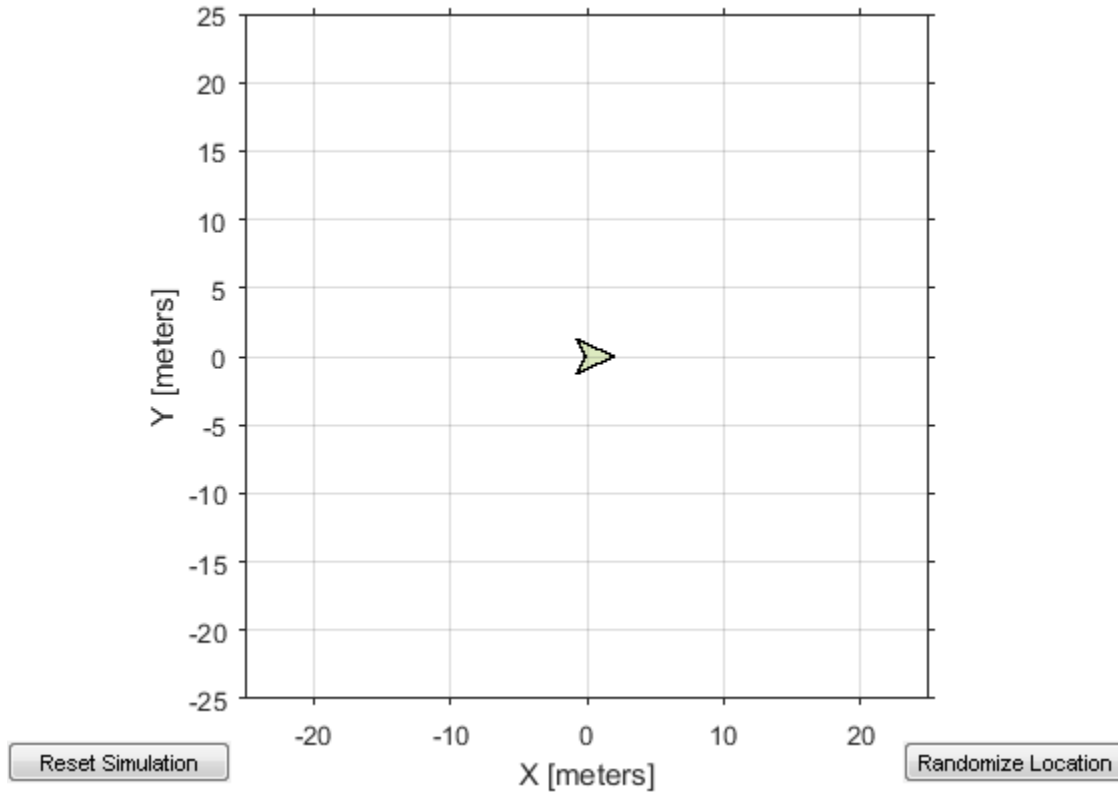
```
ans =
```

```
1×2 cell array
```

```
'robotcontroller'    'robotcontroller2'
```

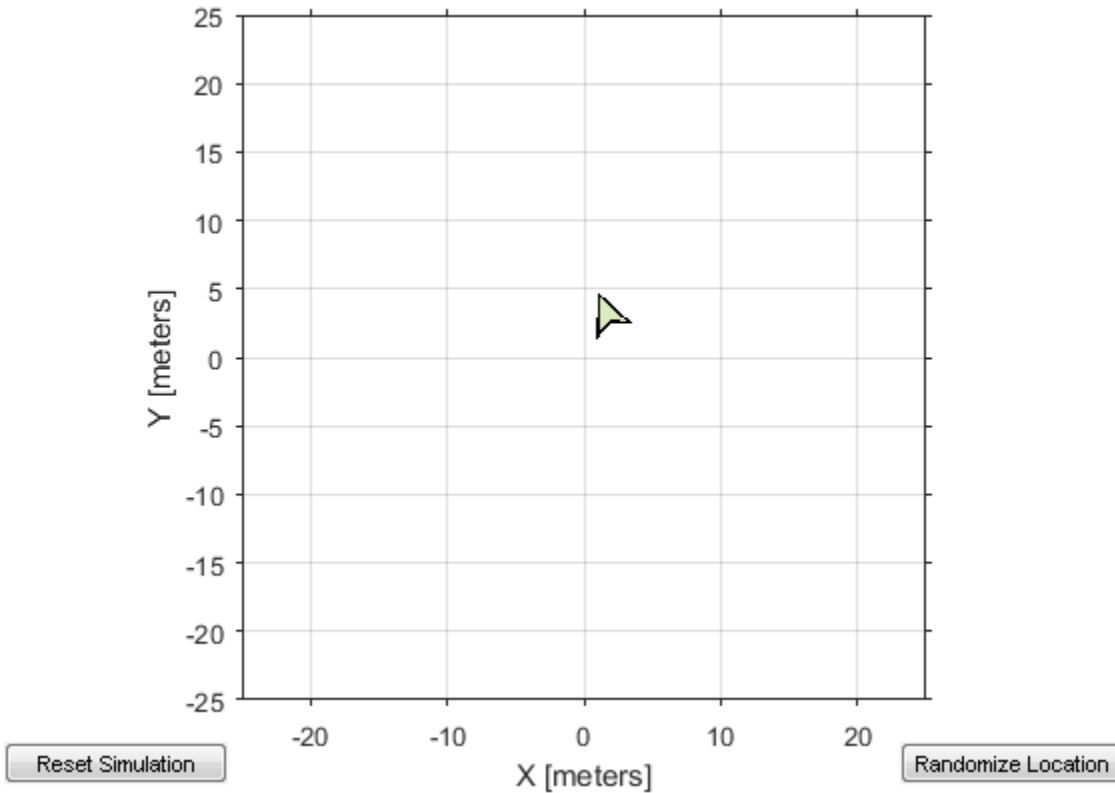
Start up the Robot Simulator using `ExampleHelperSimulinkRobotROS`. This simulator automatically connects to the ROS master on the ROS device. You will use this simulator to run a ROS node and control the robot.

```
sim = ExampleHelperSimulinkRobotROS;
```



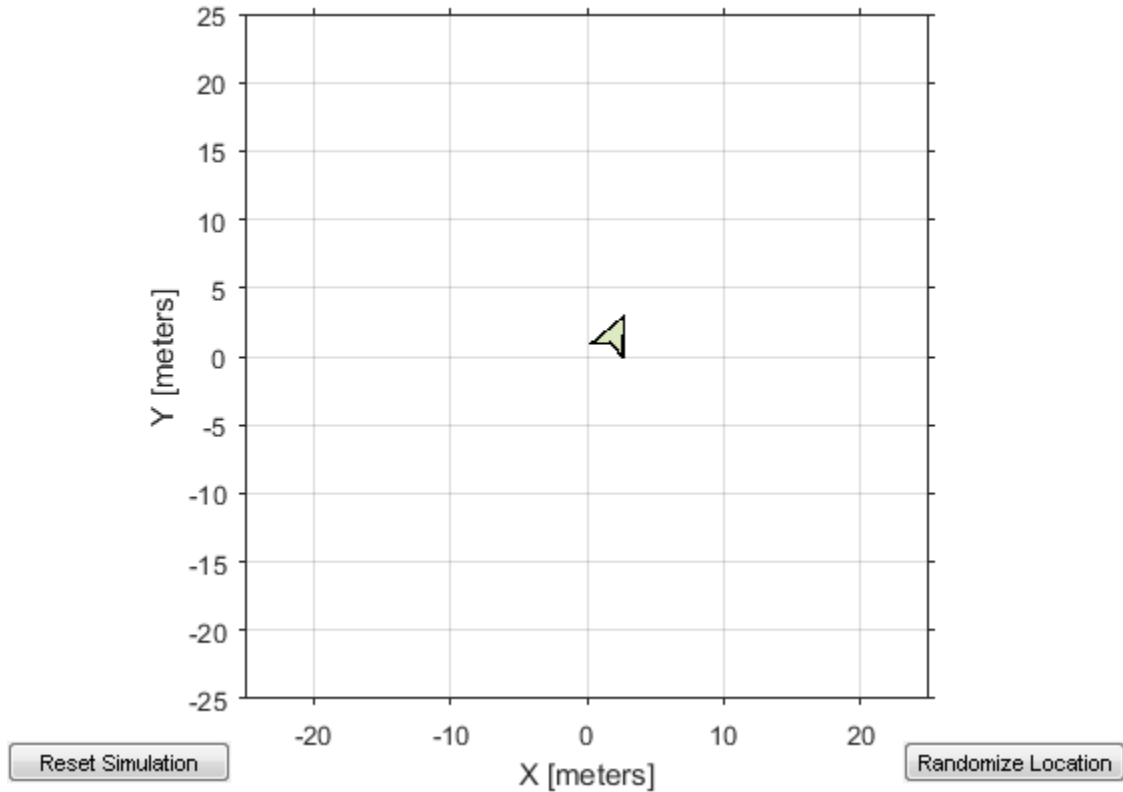
Run a ROS node, specifying the node name. The 'robotcontroller' node commands the robot to a specific location ([-10 10]). Wait to see the robot drive.

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



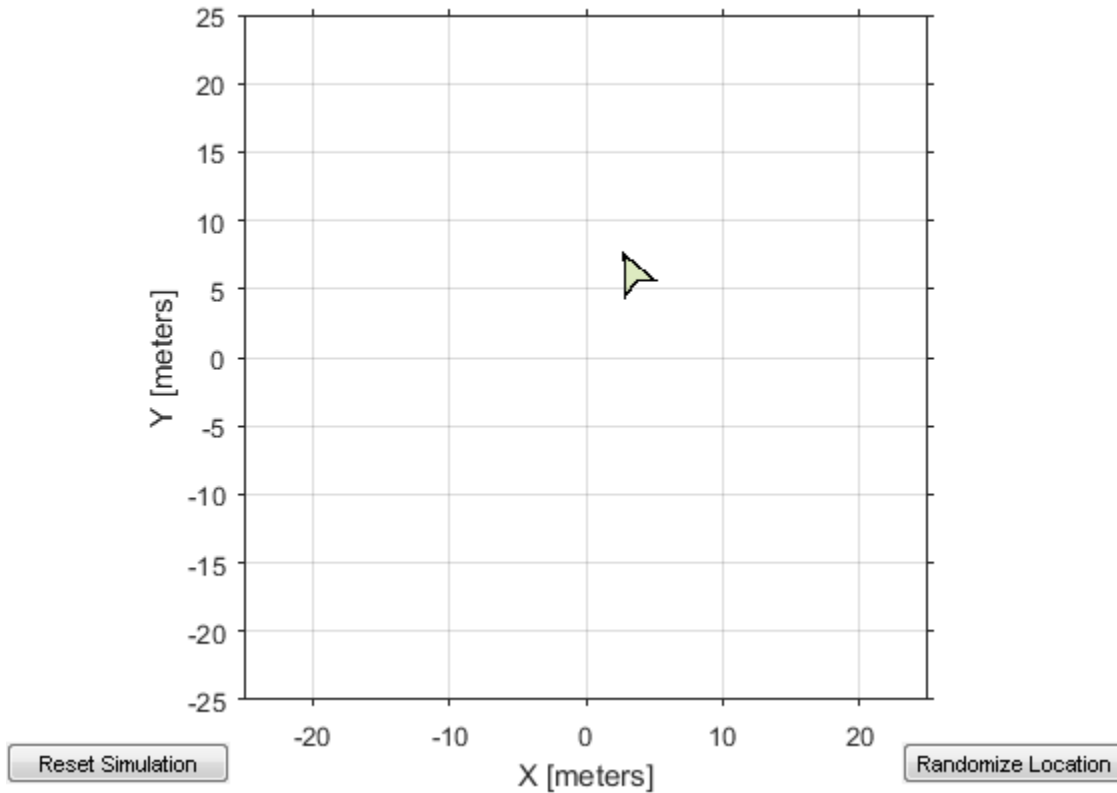
Reset the Robot Simulator to reset the robot position. Alternatively, click **Reset Simulation**. Because the node is still running, the robot continues back to the specific location. To stop sending commands, stop the node.

```
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')
roshutdown
stopCore(d)
```

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

- “Connect to a ROS Network”
- “Generate a standalone ROS node from Simulink®”

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`

Introduced in R2016b

scatter3

Display point cloud in scatter plot

Syntax

```
scatter3(pcloud)
scatter3(pcloud,Name,Value)
h = scatter3( ___ )
```

Description

`scatter3(pcloud)` plots the input `pcloud` point cloud as a 3-D scatter plot in the current axes handle. If the data contains RGB information for each point, the scatter plot is colored accordingly.

`scatter3(pcloud,Name,Value)` provides additional options specified by one or more `Name,Value` pair arguments. `Name` must appear inside single quotes (`' '`). You can specify several name-value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`

`h = scatter3(___)` returns the scatter series object, using any of the arguments from previous syntaxes. Use `h` to modify properties of the scatter series after it is created.

When plotting ROS point cloud messages, MATLAB follows the standard ROS convention for axis orientation. This convention states that **positive x is forward, positive y is left, and positive z is up**. However, if cameras are used, a second frame is defined with an “_optical” suffix which changes the orientation of the axis. In this case, positive z is forward, positive x is right, and positive y is down. MATLAB looks for the “_optical” suffix and will adjust the axis orientation of the scatter plot accordingly. For more information, see [Axis Orientation](#) on the ROS Wiki.

Examples

Show 3-D Point Cloud

```
scatter3(pcloud);
```

Show 3-D Ppoint Cloud with Uniform Red Points

```
scatter3(pcloud, 'MarkerEdgeColor', [1 0 0]);
```

Input Arguments

pc1oud — 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 | character vector of color name

Marker outline color, specified as one of these values:

- 'flat' — Colors defined by the CData property.
- 'none' — No color, which makes unfilled markers invisible.
- RGB triplet or character vector of color name — Specify a custom color.

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]. This table lists the long and short color name options and the equivalent RGB triplet values.

Long Name	Short Name	RGB Triplet
'yellow'	'y'	[1 1 0]
'magenta'	'm'	[1 0 1]
'cyan'	'c'	[0 1 1]
'red'	'r'	[1 0 0]
'green'	'g'	[0 1 0]
'blue'	'b'	[0 0 1]
'white'	'w'	[1 1 1]
'black'	'k'	[0 0 0]

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.

Supported parameter values are:

- `int32`
- `logical`
- `double`
- character vector
- cell array

Examples

Search for Parameter Names and Values Using Partial Character Vector

```
[pnames,pvalues] = search(ptree,'gravity')
pnames =
    '/gazebo/gravity_x'    '/gazebo/gravity_y'    '/gazebo/gravity_z'
```

```
pvalues =  
  
    [      0]  
    [      0]  
    [-9.8000]
```

Input Arguments

p_{tree} — Parameter tree

ParameterTree object handle

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

search_{str} — 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

p_{names} — 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.

p_{values} — Parameter values

cell array

Parameter values, returned as a cell array. These values vary, but it should match the value expected for each parameter name in the array. Supported values are:

- `int32`
- `logical`
- `double`
- character vector

- cell array

Base64–encoded binary data, iso8601 data, and dictionaries from ROS are not supported.

Limitations

Base64–encoded binary data, iso8601 data, and dictionaries 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

Retrieve a rosbag file.

```
bag = rosbag(filepath);
```

Copy the bag using the `select` function.

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

Select Message Based on Time

Get the messages from the first full second of the rosbag.

```
bagMsgs = select(bagMsgs, 'Time', [bagMsgs.StartTime, ...
    bagMsgs.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

bagse1 — 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

Publish Message Using send

```
send(pub, msg);
```

Create, Send and Receive Message

Set up a topic, publisher, and subscriber to share and receive a message.

Create a topic and publisher.

```
msgtype = rostype.geometry_msgs_Point;  
pub = rospublisher('position', msgtype);
```

Create a message.

```
msg = rosmessage(msgtype);  
msg.Y = 2
```

```
msg =
```

```
ROS Point message with properties:
```

```
MessageType: 'geometry_msgs/Point'
```

```
X: 0
Y: 2
Z: 0
```

Use `showdetails` to show the contents of the message

Send the message.

```
send(pub,msg)
```

Subscribe to the publisher.

```
sub = rossubscriber('position',msgtype)
```

```
sub =
```

Subscriber with properties:

```
  TopicName: '/position'
  MessageType: 'geometry_msgs/Point'
  LatestMessage: [1x1 Point]
  BufferSize: 25
  NewMessageFcn: []
```

Verify that the latest message received is correct.

```
sub.LatestMessage
```

```
ans =
```

ROS Point message with properties:

```
  MessageType: 'geometry_msgs/Point'
  X: 0
  Y: 2
  Z: 0
```

Use `showdetails` to show the contents of the message

Input Arguments

pub — ROS publisher

Publisher object handle

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

msg — ROS message

Message object handle

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

See Also

`rospublisher` | `rostopic`

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 `rosmesssage` and the action client object. This message sends linear and angular velocity commands to a Turtlebot® robot.

```
goalMsg = rosmesssage(actClient);  
disp(goalMsg)
```

ROS TurtlebotMoveGoal message with properties:

```
    MessageType: 'turtlebot_actions/TurtlebotMoveGoal'  
    TurnDistance: 0  
    ForwardDistance: 0
```

Use `showdetails` to show the contents of the message

Modify the goal message parameters and send the goal to the action server.

```
goalMsg.ForwardDistance = 2;  
sendGoal(actClient,goalMsg)
```

Send and Cancel ROS Action Goals

Send and cancel goals for ROS actions. First, setup a ROS action client. Then send a goal message with modified parameters. Finally, cancel your goal and all goals on the action server.

Connect to a ROS network with a specified IP address. Create a ROS action client connected using `roactionclient`. Specify the action name. Wait for the client to be connected to the server.

```
rosinit('192.168.154.131')  
[actClient,goalMsg] = roactionclient('/fibonacci');  
waitForServer(actClient);
```

Initializing global node /matlab_global_node_40739 with NodeURI http://192.168.154.1:57

Send a goal message with modified parameters. Wait for the goal to finish executing.

```
goalMsg.Order = 4;  
sendGoalAndWait(actClient,goalMsg)
```

```
Goal active  
Feedback:
```



```

    Sequence : [0, 1, 1]
Feedback:
    Sequence : [0, 1, 1, 2]
Feedback:
    Sequence : [0, 1, 1, 2, 3]
Feedback:
    Sequence : [0, 1, 1, 2, 3, 5]

```

```
ans =
```

```
ROS FibonacciResult message with properties:
```

```

  MessageType: 'actionlib_tutorials/FibonacciResult'
  Sequence: [6×1 int32]

```

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

Send a new goal message without waiting.

```
goalMsg.Order = 5;
sendGoal(actClient,goalMsg)
```

Cancel the goal on the ROS action client, `actClient`.

```
cancelGoal(actClient)
```

Cancel all the goals on the action server that `actClient` is connected to.

```
cancelAllGoals(actClient)
```

Delete the action client.

```
delete(actClient)
```

Disconnect from the ROS network.

```
roshutdown
```

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

Input Arguments

client — ROS action client

SimpleActionClient object handle

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

goalMsg — ROS action goal message

Message object handle

ROS action goal message, specified as a `Message` object handle. Update this message with your goal details and send it to the ROS action client using `sendGoal` or `sendGoalAndWait`.

More About

- “ROS Actions Overview”
- “Move a Turtlebot Robot Using ROS Actions”

See Also

`cancelGoal` | `roaction` | `roactionclient` | `sendGoalAndWait`

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] = roactionclient('/fibonacci');
waitForServer(actClient);
```

Initializing global node /matlab_global_node_40739 with NodeURI http://192.168.154.1:57

Send a goal message with modified parameters. Wait for the goal to finish executing.

```
goalMsg.Order = 4;
sendGoalAndWait(actClient,goalMsg)
```

```
Goal active
Feedback:
  Sequence : [0, 1, 1]
Feedback:
  Sequence : [0, 1, 1, 2]
Feedback:
  Sequence : [0, 1, 1, 2, 3]
Feedback:
  Sequence : [0, 1, 1, 2, 3, 5]
```

```
ans =
```

```
ROS FibonacciResult message with properties:
```

```
  MessageType: 'actionlib_tutorials/FibonacciResult'
  Sequence: [6×1 int32]
```

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

Send a new goal message without waiting.

```
goalMsg.Order = 5;
sendGoal(actClient,goalMsg)
```

Cancel the goal on the ROS action client, `actClient`.

```
cancelGoal(actClient)
```

Cancel all the goals on the action server that `actClient` is connected to.

```
cancelAllGoals(actClient)
```

Delete the action client.

```
delete(actClient)
```

Disconnect from the ROS network.

```
roshutdown
```

```
Shutting down global node /matlab_global_node_40739 with NodeURI http://192.168.154.1:11311
```

Input Arguments

client — ROS action client

SimpleActionClient object handle

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

goalMsg — ROS action goal message

Message object handle

ROS action goal message, specified as a Message object handle. Update this message with your goal details and send it to the ROS action client using `sendGoal` or `sendGoalAndWait`.

timeout — Timeout period

scalar in seconds

Timeout period for receiving a result message, specified as a scalar in seconds. If the client does not receive a new result message in that time period, an error is displayed.

Output Arguments

resultMsg — Result message

ROS Message object

Result message, returned as a ROS Message object. The result message contains the result data sent by the action server. This data depends on the action type.

state — Final goal state

character vector

Final goal state, returned as one of the following:

- 'pending' — Goal was received, but has not yet been accepted or rejected.
- 'active' — Goal was accepted and is running on the server.
- 'succeeded' — Goal executed successfully.
- 'preempted' — An action client canceled the goal before it finished executing.
- 'aborted' — The goal was aborted before it finished executing. The action server typically aborts a goal.
- 'rejected' — The goal was not accepted after being in the 'pending' state. The action server typically triggers this status.
- 'recalled' — A client canceled the goal while it was in the 'pending' state.
- 'lost' — An internal error occurred in the action client.

status — Status text

character vector

Status text that the server associated with the final goal state, returned as a character vector.

More About

- “ROS Actions Overview”
- “Move a Turtlebot Robot Using ROS Actions”

See Also

`cancelGoal` | `rosaction` | `rosactionclient` | `sendGoal`

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://172.28.194.90:5
```

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.

```
roshutdown
```

```
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 `rostop` 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; add new parameter

Syntax

```
set(ptree, paramname, pvalue)
```

Description

`set(ptree, paramname, pvalue)` assigns the value `pvalue` to the parameter with the name `paramname`, which is contained in the parameter tree `ptree`.

Supported parameter values are:

- `int32`
- `logical`
- `double`
- character vector
- cell array

Examples

Set and Get Parameter Value

```
ptree = rosparam;  
set(ptree, 'DoubleParam', 1.0)  
get(ptree, 'DoubleParam')  
  
ans =  
  
    1
```

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 — Parameter value

int32 | logical | char | double | cell array

Parameter value, returned as either a `int32`, `logical`, `double`, `char`, or `cell array`. `pvalue` matches the value of the specified `paramname` and the supported data type in `ParameterTree`. Base64–encoded binary data, iso8601 data, and dictionaries from ROS are not supported.

Limitations

Base64–encoded binary data, iso8601 data, and dictionaries from ROS are not supported.

See Also

`get` | `rosparam`

Introduced in R2015a

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.

```
msg = rosmessage(rostype.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
```

- “Generate a standalone ROS node from Simulink®”

Input Arguments

device — ROS device

rosdevice object

ROS device, specified as a rosdevice object.

See Also

isCoreRunning | rosdevice | runCore

Introduced in R2016b

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 =
```

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

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

```
Initializing global node /matlab_global_node_14356 with NodeURI http://192.168.154.1:6
```

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

```
d.AvailableNodes
```

```
ans =
```

```
1×2 cell array
```

```
'robotcontroller' 'robotcontroller2'
```

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

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

```
running =
```

```
logical
```

```
1
```

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

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

```
Shutting down global node /matlab_global_node_14356 with NodeURI http://192.168.154.1:6
```

Run Multiple ROS Nodes

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

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

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

```
ipaddress = '192.168.154.131';  
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 MATLAB® 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://192.168.154.1:6
```

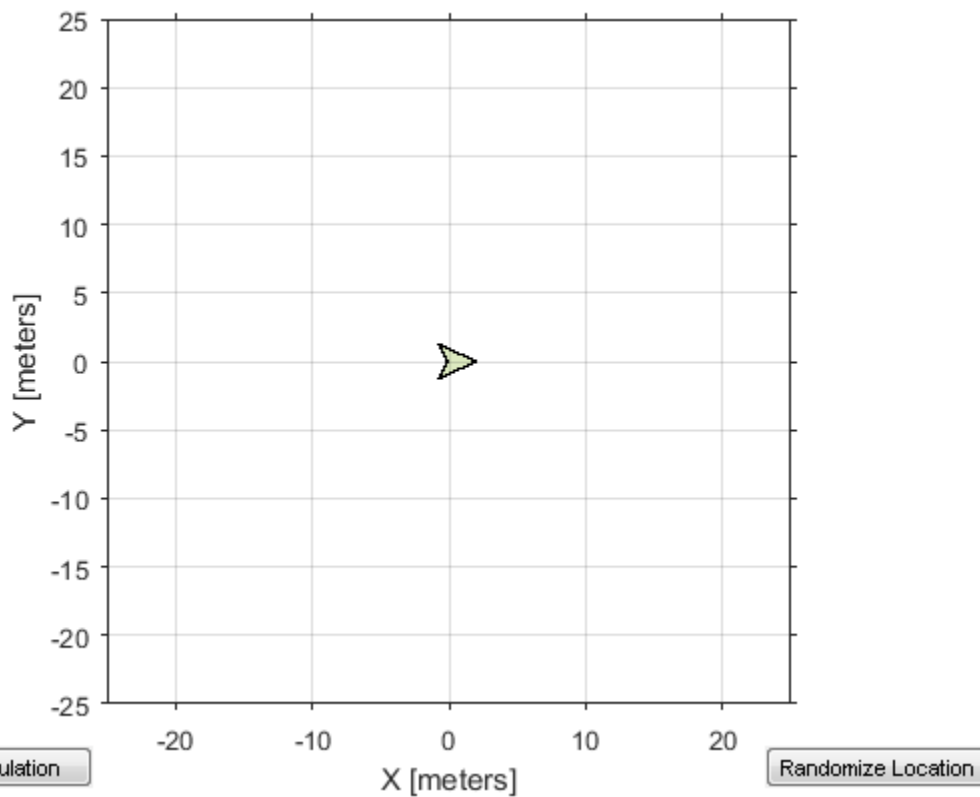
Check the available ROS nodes on the connected ROS device. These nodes were generated from Simulink® models following the process in the [Generate a standalone ROS node from Simulink®](#) example.

d.AvailableNodes

```
ans =  
  
1x2 cell array  
  
    'robotcontroller'    'robotcontroller2'
```

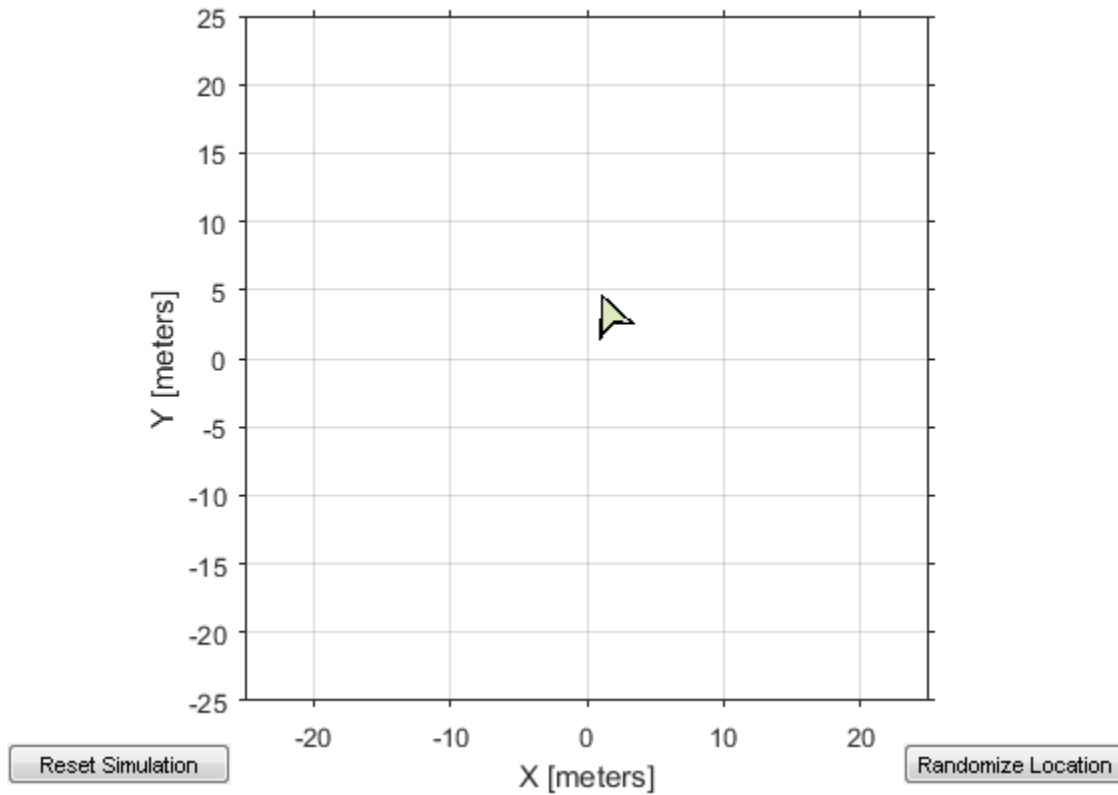
Start up the Robot Simulator using `ExampleHelperSimulinkRobotROS`. This simulator automatically connects to the ROS master on the ROS device. You will use this simulator to run a ROS node and control the robot.

```
sim = ExampleHelperSimulinkRobotROS;
```



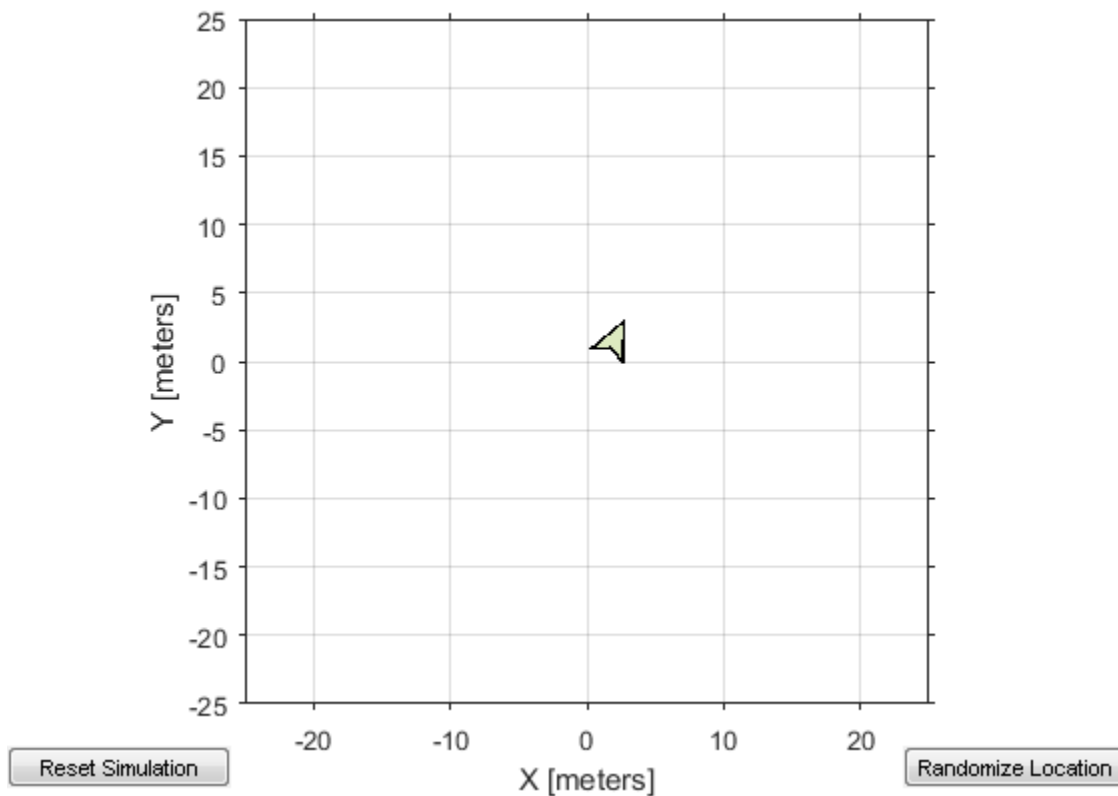
Run a ROS node, specifying the node name. The 'robotcontroller' node commands the robot to a specific location ([-10 10]). Wait to see the robot drive.

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



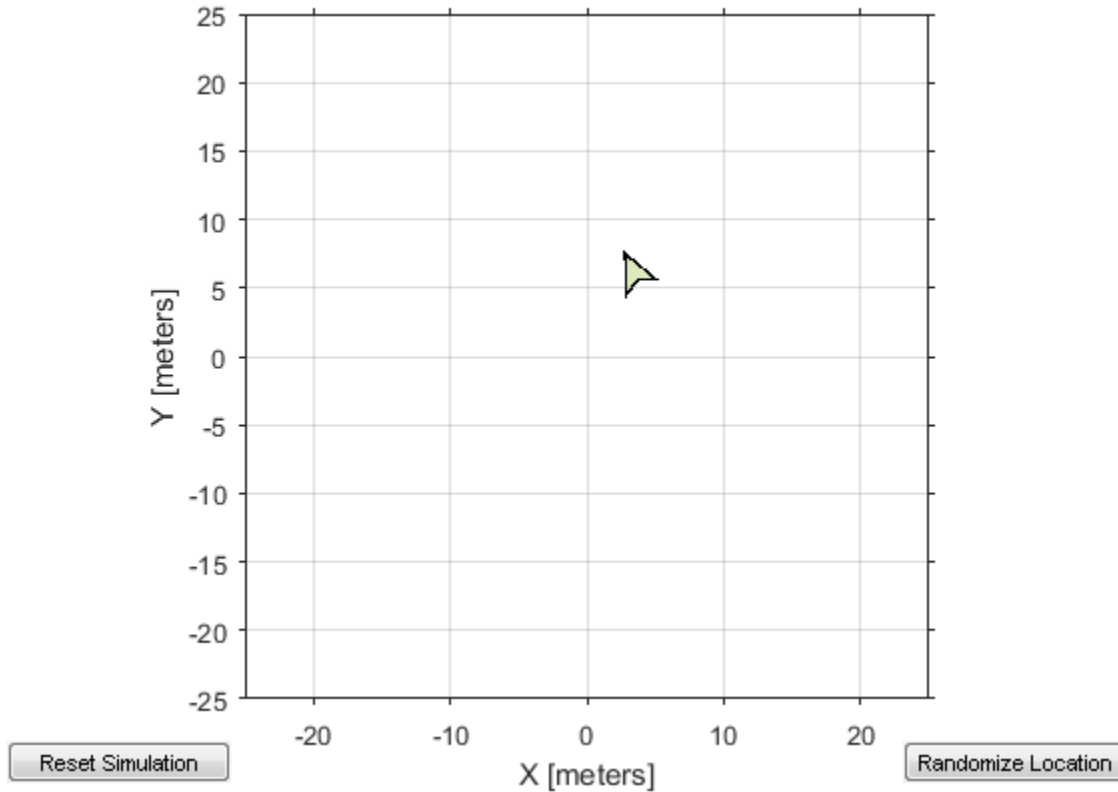
Reset the Robot Simulator to reset the robot position. Alternatively, click **Reset Simulation**. Because the node is still running, the robot continues back to the specific location. To stop sending commands, stop the node.

```
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')
roshutdown
stopCore(d)
```

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

- “Generate a standalone ROS node from Simulink®”

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`

Introduced in R2016b

system

Execute system command on device

Syntax

```
system(device,command)
system(device,command,'sudo')
response = system(____)
```

Description

`system(device,command)` runs a command in the Linux command shell on the ROS device. This function does not allow you to run interactive commands.

`system(device,command,'sudo')` runs a command with superuser privileges.

`response = system(____)` runs a command using any of the previous syntaxes with the command shell output returned in `response`.

Examples

Run Linux Commands on ROS Device

Connect to a ROS device and run commands on the Linux® command shell.

Connect to a ROS device. Specify the device address, user name, and password of your ROS device.

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

Run a command that lists the contents of the Catkin workspace folder.

```
system(d,'ls /home/user/catkin_ws_test')
```

```
ans =
```



```
build
devel
robotcontroller2_node.log
robotcontroller_node.log
src
```

Input Arguments

device — ROS device

rosdevice object

ROS device, specified as a rosdevice object.

command — Linux command

character vector

Linux command, specified as a character vector.

Example: 'ls -al'

Output Arguments

response — Output from Linux shell

character vector

Output from Linux shell, returned as a character vector.

See Also

`deleteFile` | `dir` | `getFile` | `openShell` | `putFile` | `rosdevice`

Introduced in R2016b

tform2axang

Convert homogeneous transformation to axis-angle rotation

Syntax

```
axang = tform2axang(tform)
```

Description

`axang = tform2axang(tform)` converts the rotational component of a homogeneous transformation, `tform`, to an axis-angle rotation, `axang`. The translational components of `tform` are ignored. The input homogeneous transformation must be in the premultiply form for transformations.

Code Generation Support:

Supports MATLAB Function block: Yes

“Code Generation Support, Usage Notes and Limitations”

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]

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'.

Code Generation Support:

Supports MATLAB Function block: Yes

“Code Generation Support, Usage Notes and Limitations”

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 — Homogeneous transformation

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

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

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

sequence — Axis rotation sequence

'ZYX' (default) | 'ZYZ'

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

- 'ZYX' (default) – The order of rotation angles is *z*-axis, *y*-axis, *x*-axis.
- 'ZYZ' – The order of rotation angles is *z*-axis, *y*-axis, *z*-axis.

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]

See Also

eu12tform

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.

Code Generation Support:

Supports MATLAB Function block: Yes

“Code Generation Support, Usage Notes and Limitations”

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 — Homogeneous transformation

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

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

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

Output Arguments

quat — Unit quaternion

n -by-4 matrix

Unit quaternion, returned as an n -by-4 matrix containing n quaternions. Each quaternion, one per row, is of the form $q = [w \ x \ y \ z]$, with w as the scalar number.

Example: [0.7071 0.7071 0 0]

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).

Code Generation Support:

Supports MATLAB Function block: Yes

“Code Generation Support, Usage Notes and Limitations”

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]

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.

Code Generation Support:

Supports MATLAB Function block: Yes

“Code Generation Support, Usage Notes and Limitations”

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]

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

```
ts = timeseries(bagMsgs);
```

Create Time Series with Single Property

```
ts = timeseries(bagMsgs, 'Pose.Pose.Position.X');
```

Create Time Series with Multiple Properties

```
ts = timeseries(bagMsgs, '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.

More About

- “Time Series Basics”

See Also

`readMessages` | `rosviz` | `select`

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

Transform PointStamped Message

Define a point in the coordinate frame of a camera.

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

Transform the point to the `base_link` frame. You must be connected to a ROS network.


```
tftp = transform(tftree, 'base_link', pt);
```

You can also use the source time of the message to get the transformation for that specific time. Use the message `Header` to get the time stamp, `Stamp`.

```
tftp = transform(tftree, 'base_link', pt, pt.Header.Stamp);
```

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 `roscpp`. Replace `ipaddress` with your ROS network address.

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

```
Initializing global node /matlab_global_node_93523 with NodeURI http://172.28.194.90:6
```

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.

```
roshutdown
```

```
Shutting down global node /matlab_global_node_93523 with NodeURI http://172.28.194.90:0
```

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://192.168.154.1:6
```

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.

```
roshutdown
```

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

Input Arguments

tftree — ROS transformation tree

TransformationTree object handle

ROS transformation tree, specified as a `TransformationTree` object handle. You can create a transformation tree by calling the `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

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).

Code Generation Support:

Supports MATLAB Function block: Yes

“Code Generation Support, Usage Notes and Limitations”

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]`

See Also

`tform2trvec`

Introduced in R2015a

waitForServer

Wait for action server to start

Syntax

```
waitForServer(client)
waitForServer(client, timeout)
```

Description

`waitForServer(client)` waits until the action server is started up and available to send goals. The `IsServerConnected` property of the `SimpleActionClient` shows the status of the server connection. Press **Ctrl+C** to abort the wait.

`waitForServer(client, timeout)` specifies a timeout period in seconds. If the server does not start up in the timeout period, this function displays an error.

Examples

Setup a ROS Action Client and Execute an Action

This example shows how to create a ROS action client and execute the action. Action types must be setup beforehand with an action server running.

You must have the `'/fibonacci'` action type setup. To run this action server use the following command on the ROS system:

```
roslaunch actionlib_tutorials fibonacci_server
```

Connect to a ROS network. You must be connected to a ROS network to gather information about what actions are available. Replace `ipaddress` with your network address.

```
ipaddress = '192.168.154.131';
rosinit(ipaddress)
```

```
Initializing global node /matlab_global_node_68978 with NodeURI http://192.168.154.1:5
```


List actions available on the network. The only action setup on this network is the '/ fibonacci' action.

```
roaction list
```

```
/fibonacci
```

Create an action client. Specify the action name.

```
[actClient,goalMsg] = roactionclient('/fibonacci');
```

Wait for action client to connect to server.

```
waitForServer(actClient);
```

The fibonacci action will calculate the fibonacci sequence for a given order specified in the goal message. The goal message was returned when creating the action client and can be modified to send goals to the ROS action server.

```
goalMsg.Order = 8
```

```
goalMsg =
```

```
ROS FibonacciGoal message with properties:
```

```
  MessageType: 'actionlib_tutorials/FibonacciGoal'  
    Order: 8
```

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

Send goal and wait for its completion. Specify a timeout of 10 seconds to complete the action.

```
[resultMsg,resultState] = sendGoalAndWait(actClient,goalMsg,10)
```

```
Goal active
```

```
Feedback:
```

```
  Sequence : [0, 1, 1]
```

```
Feedback:
```

```
  Sequence : [0, 1, 1, 2]
```

```
Feedback:
```

```
  Sequence : [0, 1, 1, 2, 3]
```

```
Feedback:
```

```
Sequence : [0, 1, 1, 2, 3, 5]
Feedback:
Sequence : [0, 1, 1, 2, 3, 5, 8]
Feedback:
Sequence : [0, 1, 1, 2, 3, 5, 8, 13]
Feedback:
Sequence : [0, 1, 1, 2, 3, 5, 8, 13, 21]
Feedback:
Sequence : [0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
Final state succeeded with result:
Sequence : [0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
```

```
resultMsg =
```

```
ROS FibonacciResult message with properties:
```

```
MessageType: 'actionlib_tutorials/FibonacciResult'
Sequence: [10×1 int32]
```

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

```
resultState =
```

```
1×9 char array
```

```
succeeded
```

Disconnect from the ROS network.

```
roshutdown
```

```
Shutting down global node /matlab_global_node_68978 with NodeURI http://192.168.154.1:5
```

Send and Cancel ROS Action Goals

Send and cancel goals for ROS actions. First, setup a ROS action client. Then send a goal message with modified parameters. Finally, cancel your goal and all goals on the action server.

Connect to a ROS network with a specified IP address. Create a ROS action client connected using `rosactionclient`. Specify the action name. Wait for the client to be connected to the server.

```
rosinit('192.168.154.131')
[actClient,goalMsg] = rosactionclient('/fibonacci');
waitForServer(actClient);
```

Initializing global node /matlab_global_node_40739 with NodeURI http://192.168.154.1:57

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.

```
roshutdown
```

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

Input Arguments

client — ROS action client

SimpleActionClient object handle

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

timeout — Timeout period

scalar in seconds

Timeout period for setting up ROS action server, specified as a scalar in seconds. If the client does not connect to the server in the specified time period, an error is displayed.

More About

- “ROS Actions Overview”
- “Move a Turtlebot Robot Using ROS Actions”

See Also

cancelGoal | roaction | roactionclient | sendGoalAndWait

Introduced in R2016b

waitForTransform

Wait until a transformation is available

Compatibility

`waitForTransform` will be removed in a future release. Use `getTransform` with a specified `timeout` instead. Use `inf` to wait indefinitely.

Syntax

```
waitForTransform(tftree, targetframe, sourceframe)  
waitForTransform(tftree, targetframe, sourceframe, timeout)
```

Description

`waitForTransform(tftree, targetframe, sourceframe)` waits until the transformation between `targetframe` and `sourceframe` is available in the transformation tree, `tftree`. This function disables the command prompt until a transformation becomes available on the ROS network.

`waitForTransform(tftree, targetframe, sourceframe, timeout)` specifies a timeout period in seconds. If the transformation does not become available, MATLAB displays an error, but continues running the current program.

Examples

Wait for Transform

```
waitForTransform(tftree, '/camera_depth_frame', '/base_link');
```

Specify Timeout of Five Seconds to Wait for Transform

```
waitForTransform(tftree, '/camera_depth_frame', '/base_link', 5);
```

Input Arguments

tftree — ROS transformation tree

TransformationTree object handle

ROS transformation tree, specified as a TransformationTree object handle. You can create a transformation tree by calling the `rostopic` function.

targetframe — Target coordinate frame

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

```
map = robotics.BinaryOccupancyGrid(randi([0,1], 10));  
msg = rosmessage('nav_msgs/OccupancyGrid');  
writeBinaryOccupancyGrid(msg, map);
```

Input Arguments

map — Binary occupancy grid

BinaryOccupancyGrid object handle

Binary occupancy grid, specified as a BinaryOccupancyGrid object handle. `map` is converted to a 'nav_msgs/OccupancyGrid' message on the ROS network. `map` is an object with a grid of binary values, where 1 indicates an occupied location and 0 indicates an unoccupied location.

msg — 'nav_msgs/OccupancyGrid' ROS message

OccupancyGrid object handle

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

See Also

`robotics.BinaryOccupancyGrid` | `readBinaryOccupancyGrid`

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

```
msg = rosmessages('sensor_msgs/Image')
writeImage(msg, img);
```

Write Message Using Alpha Channel

```
writeImage(msg, img, alpha);
```

Input Arguments

msg — ROS image message
Image object handle

'sensor_msgs/Image' ROS image message, specified as an Image object handle.
'sensor_msgs/Image' image messages are not supported.

img — Image

grayscale image matrix | RGB image matrix | *m*-by-*n*-by-3 array

Image, specified as a matrix representing a grayscale or RGB image or as *m*-by-*n*-by-3 array, depending on the sensor image.

alpha — Alpha channel

uint8 grayscale image

Alpha channel, specified as a uint8 grayscale image. Alpha must be the same size and data type as `img`.

More About

Tips

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 — 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 — Object array

handle

Object array, specified as a handle.

Output Arguments

b — Object array containing copies of the objects in **a**
handle

Object array containing copies of the object in **a**, specified as a handle.

See Also

robotics.BinaryOccupancyGrid

Introduced in R2015a

getOccupancy

Class: robotics.BinaryOccupancyGrid

Package: robotics

Get occupancy value for one or more positions

Syntax

```
occval = getOccupancy(map,xy)
occval = getOccupancy(map,ij,'grid')
```

Description

`occval = getOccupancy(map,xy)` returns an array of occupancy values for an input array of world coordinates, `xy`. Each row of `xy` is a point in the world, represented as an `[x y]` coordinate pair. `occval` is the same length as `xy` and a single column array. An occupied location is represented as `true` (1), and a free location is represented as `false` (0).

`occval = getOccupancy(map,ij,'grid')` returns an array of occupancy values based on a `[rows cols]` input array of grid positions, `ij`.

Input Arguments

map — Map representation

BinaryOccupancyGrid object

Map representation, specified as a robotics.BinaryOccupancyGrid object. This object represents the environment of the robot. The object contains a matrix grid with binary values indicating obstacles as `true` (1) and free locations as `false` (0).

xy — World coordinates

n-by-2 vertical array

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

Data Types: double

ij — Grid positions

n-by-2 vertical array

Grid positions, specified as an *n*-by-2 vertical array of [*i* *j*] pairs in [rows cols] format, where *n* is the number of grid positions.

Data Types: double

Output Arguments

occva1 — Occupancy values

n-by-1 vertical array

Occupancy values of the same length as either *xy* or *ij*, returned as an *n*-by-1 vertical array, where *n* is the same *n* in either *xy* or *ij*.

See Also

[robotics.BinaryOccupancyGrid](#) | [robotics.BinaryOccupancyGrid.setOccupancy](#)

Introduced in R2015a

grid2world

Class: robotics.BinaryOccupancyGrid

Package: robotics

Convert grid indices to world coordinates

Syntax

```
xy = grid2world(map,ij)
```

Description

`xy = grid2world(map,ij)` converts a `[row col]` array of grid indices, `ij`, to an array of world coordinates, `xy`.

Input Arguments

map — Map representation

BinaryOccupancyGrid object

Map representation, specified as a robotics.BinaryOccupancyGrid object. This object represents the environment of the robot. The object contains a matrix grid with binary values indicating obstacles as `true` (1) and free locations as `false` (0).

ij — Grid positions

n-by-2 vertical array

Grid positions, specified as an *n*-by-2 vertical array of `[i j]` pairs in `[rows cols]` format, where *n* is the number of grid positions.

Output Arguments

xy — World coordinates

n-by-2 vertical array

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

See Also

`robotics.BinaryOccupancyGrid` | `robotics.BinaryOccupancyGrid.world2grid`

Introduced in R2015a

inflate

Class: robotics.BinaryOccupancyGrid

Package: robotics

Inflate each occupied grid location

Syntax

```
inflate(map, radius)
inflate(map, gridradius, 'grid')
```

Description

`inflate(map, radius)` inflates each occupied position of the `map` by the radius given in meters. `radius` is rounded up to the nearest cell equivalent based on the resolution of the map. Every cell within the radius is set to `true` (1).

`inflate(map, gridradius, 'grid')` inflates each occupied position by the radius given in number of cells.

Input Arguments

map — Map representation

BinaryOccupancyGrid object

Map representation, specified as a robotics.BinaryOccupancyGrid object. This object represents the environment of the robot. The object contains a matrix grid with binary values indicating obstacles as `true` (1) and free locations as `false` (0).

radius — Dimension the defines how much to inflate occupied locations

scalar

Dimension that defines how much to inflate occupied locations, specified as a scalar. `radius` is rounded up to the nearest cell value.

Data Types: double

gridradius — Dimension that 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 — Map representation

BinaryOccupancyGrid object

Map representation, specified as a robotics.BinaryOccupancyGrid object. This object represents the environment of the robot. The object contains a matrix grid with binary values indicating obstacles as `true` (1) and free locations as `false` (0).

xy — World coordinates

n-by-2 vertical array

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

Data Types: double

ij — Grid positions

n-by-2 vertical array

Grid positions, specified as an *n*-by-2 vertical array of [*i* *j*] pairs in [rows cols] format, where *n* is the number of grid positions.

Data Types: double

occval — Occupancy values

n-by-1 vertical array

Occupancy values of the same length as either *xy* or *ij*, returned as an *n*-by-1 vertical array, where *n* is the same *n* in either *xy* or *ij*.

Examples

Set Occupancy Values

Set the occupancy of grid locations using `setOccupancy`.

Initialize an occupancy grid object using `BinaryOccupancyGrid`.

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

Set the occupancy of a specific location using `setOccupancy`.

```
setOccupancy(map,[8 8],1);
```

Set the occupancy of an array of locations.

```
[x,y] = meshgrid(2:5);  
setOccupancy(map,[x(:) y(:)],1);
```

See Also

`robotics.BinaryOccupancyGrid` | `robotics.BinaryOccupancyGrid.getOccupancy`

Introduced in R2015a

show

Class: robotics.BinaryOccupancyGrid

Package: robotics

Show occupancy grid values

Syntax

```
show(map)
```

```
show(map, 'grid')
```

```
show( ____, 'Parent', parent)
```

```
h = show(map, ____)
```

Description

`show(map)` displays the binary occupancy grid `map` in the current axes, with the axes labels representing the world coordinates.

`show(map, 'grid')` displays the binary occupancy grid `map` in the current axes, with the axes labels representing the grid coordinates.

`show(____, 'Parent', parent)` sets the specified axes handle `parent` to the axes, using any of the arguments from previous syntaxes.

`h = show(map, ____)` returns the figure object handle created by `show`.

Input Arguments

map — Map representation

BinaryOccupancyGrid object

Map representation, specified as a `robotics.BinaryOccupancyGrid` object. This object represents the environment of the robot. The object contains a matrix grid with binary values indicating obstacles as `true` (1) and free locations as `false` (0).

parent — Axes properties

handle

Axes properties, specified as a handle.

See Also

robotics.BinaryOccupancyGrid

Introduced in R2015a

world2grid

Class: robotics.BinaryOccupancyGrid

Package: robotics

Convert world coordinates to grid indices

Syntax

```
ij = world2grid(map,xy)
```

Description

`ij = world2grid(map,xy)` converts an array of world coordinates, `xy`, to a [rows cols] array of grid indices, `ij`.

Input Arguments

map — Map representation

BinaryOccupancyGrid object

Map representation, specified as a robotics.BinaryOccupancyGrid object. This object represents the environment of the robot. The object contains a matrix grid with binary values indicating obstacles as `true` (1) and free locations as `false` (0).

xy — World coordinates

n-by-2 vertical array

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

Output Arguments

ij — Grid positions

n-by-2 vertical array

Grid positions, specified as an n -by-2 vertical array of $[i\ j]$ pairs in `[rows cols]` format, where n is the number of grid positions.

See Also

`robotics.BinaryOccupancyGrid` | `robotics.BinaryOccupancyGrid.grid2world`

Introduced in R2015a

release

Class: robotics.InverseKinematics

Package: robotics

Allow property value changes

Syntax

```
release(ik)
```

Description

`release(ik)` releases the `InverseKinematics` object so that you can change its properties. Use `release` to change the underlying algorithm specified in the `SolverAlgorithm` property of the object.

Input Arguments

ik — Inverse kinematics solver

`InverseKinematics` object

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

See Also

`robotics.InverseKinematics` | `robotics.InverseKinematics.step`

Introduced in R2016b

step

Class: 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 — Inverse kinematics solver

InverseKinematics object

Inverse kinematics solver, specified as an InverseKinematics object.

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 `ik`.

pose — End-effector pose

4-by-4 homogeneous transform

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

weights — Weight for pose tolerances

six-element vector

Weight for pose tolerances, specified as a six-element vector. The first three elements correspond to weight of the *xyz* positions. The last three elements correspond to the weight of the *yaw-pitch-roll* angles.

initialguess — 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 — Robot configuration solution

structure array

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

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

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

solInfo — Solution information

structure

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

- `Iterations` — Number of iterations run by the algorithm.

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

Examples

Achieve EndEffector Position Using Inverse Kinematics

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

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

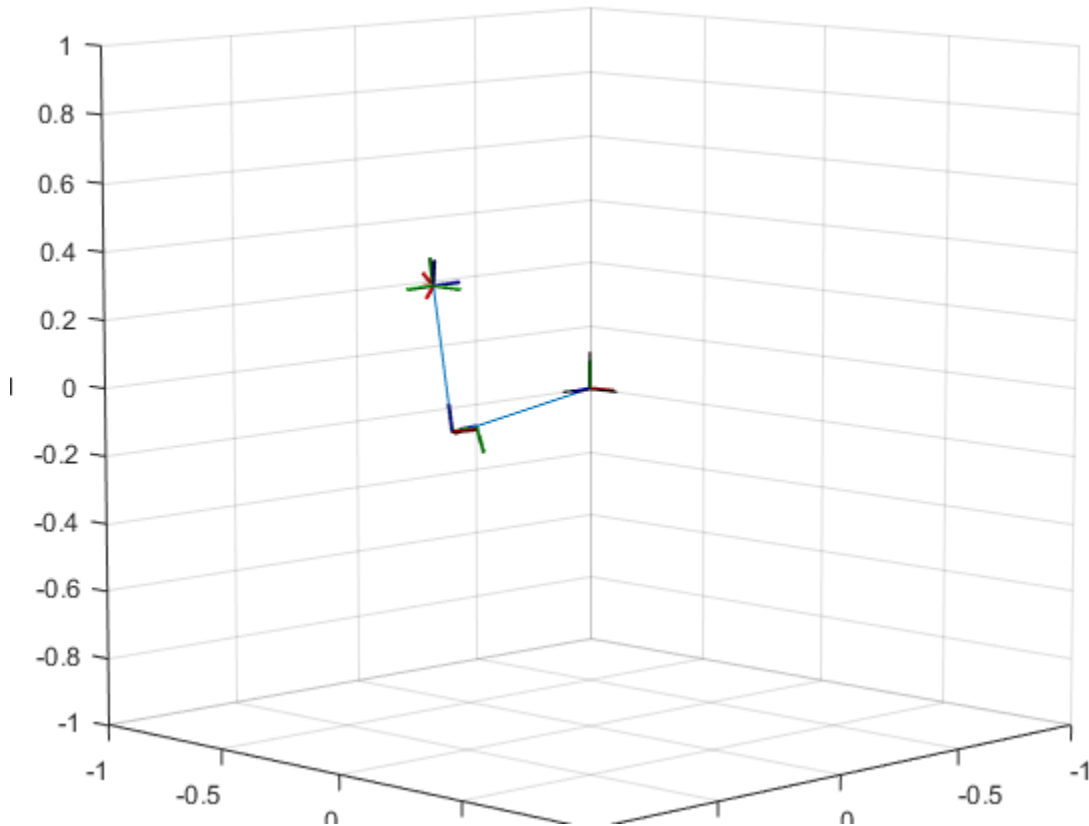
```
load exampleRobots.mat
showdetails(puma1)
```

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

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

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

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



Create an `InverseKinematics` object for the `puma1` model. Use this object to calculate a solution for the given goal. Specify weights for the different components of the pose. Use the home configuration of the robot as an initial guess.

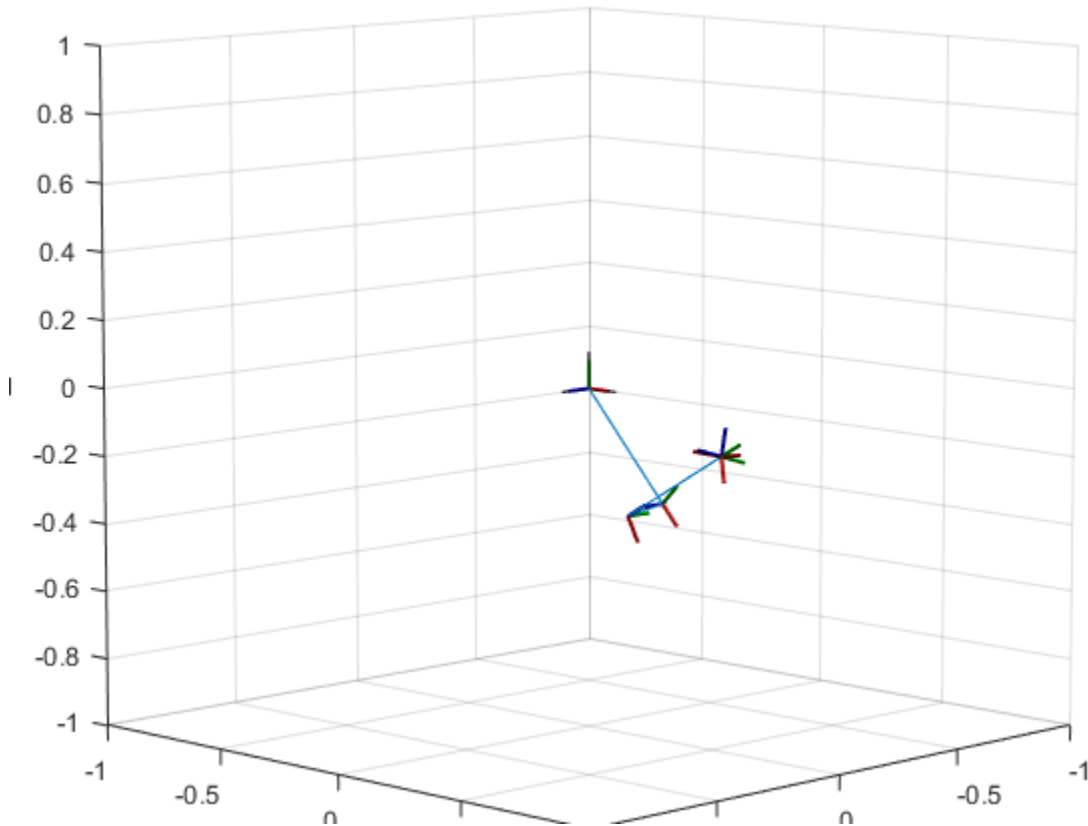
```
ik = robotics.InverseKinematics('RigidBodyTree',puma1);  
weights = ones(6,1);  
initialguess = puma1.homeConfiguration;
```

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);
```



- “Control PR2 Arm Movements using ROS Actions and Inverse Kinematics”

See Also

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

More About

- “Inverse Kinematics Algorithms”

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 — Joint object

handle

Joint object, specified as a handle. Create a joint object using `robotics.Joint`.

Output Arguments

jCopy — Joint object

handle

Joint object, returned as a handle. Create a joint object using `robotics.Joint`. This copy has the same properties.

See Also

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

Introduced in R2016b

setFixedTransform

Class: robotics.Joint

Package: robotics

Set fixed transform properties of joint

Syntax

```
setFixedTransform(jointObj, tform)
```

```
setFixedTransform(jointObj, dhparams, 'dh')
```

```
setFixedTransform(jointObj, mdhparams, 'mdh')
```

Description

`setFixedTransform(jointObj, tform)` sets the `JointToParentTransform` property of the `Joint` object directly with the supplied homogenous transformation.

`setFixedTransform(jointObj, dhparams, 'dh')` sets the `ChildToJointTransform` property using Denavit-Hartenberg (DH) parameters. The `JointToParentTransform` property is set to an identity matrix. DH parameters are given in the order [a alpha d theta].

`setFixedTransform(jointObj, mdhparams, 'mdh')` sets the `JointToParentTransform` property using modified DH parameters. The `ChildToJointTransform` property is set to an identity matrix. Modified DH parameters are given in the order [a alpha d theta].

Input Arguments

jointObj — Joint object

handle

Joint object, specified as a handle. Create a joint object using `robotics.Joint`.

tform — Homogeneous transform

4-by-4 matrix

Homogeneous transform, specified as a 4-by-4 matrix. The transform is set to the `ChildToJointTransform` property. The `JointToParentTransform` property is set to an identity matrix.

dhparams — Denavit-Hartenberg (DH) parameters

four-element vector

Denavit-Hartenberg (DH) parameters, specified as a four-element vector, [`a` `alpha` `d` `theta`]. These parameters are used to set the `ChildToJointTransform` property. The `JointToParentTransform` property is set to an identity matrix.

mdhparams — Modified Denavit-Hartenberg (DH) parameters

four-element vector

Modified Denavit-Hartenberg (DH) parameters, specified as a four-element vector, [`a` `alpha` `d` `theta`]. These parameters are used to set the `JointToParentTransform` property. The `ChildToJointTransform` is set to an identity matrix.

Examples

Build Manipulator Robot Using Denavit-Hartenberg Parameters

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

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

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

Create a rigid body tree object to build the robot.

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

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

- 1 Create a `RigidBody` object and give it a unique name.
- 2 Create a `Joint` object and give it a unique name.
- 3 Use `setFixedTransform` to specify the body-to-body transformation using DH parameters.
- 4 Call `addBody` to attach the first body joint to the base frame of the robot.

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

setFixedTransform(jnt1, dhparams(1,:), 'dh');
body1.Joint = jnt1;

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

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

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

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

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

addBody(robot, body2, 'body1')
addBody(robot, body3, 'body2')
```

```

addBody(robot,body4,'body3')
addBody(robot,body5,'body4')
addBody(robot,body6,'body5')

```

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

```
showdetails(robot)
```

```

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

```

```

-----
Robot: (6 bodies)

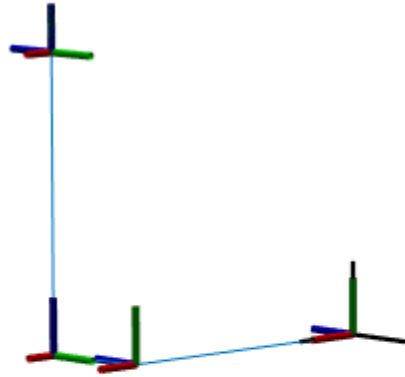
```

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

clone

System object: robotics.MonteCarloLocalization

Package: robotics

Create MonteCarloLocalization object having same property values

Syntax

```
copy = clone(mcl)
```

Description

`copy = clone(mcl)` creates another instance of the `MonteCarloLocalization` object, `mcl`, having the same property values. `clone` creates a new unlocked object with uninitialized states.

Input Arguments

mcl — MonteCarloLocalization object

handle

robotics.MonteCarloLocalization object, specified as an object handle.

Output Arguments

copy — MonteCarloLocalization object

handle

robotics.MonteCarloLocalization object, returned as an object handle. The object is unlocked with uninitialized states.

Examples

Copy Monte Carlo Localization Object

```
mcl = robotics.MonteCarloLocalization;  
copy = clone(mcl)
```

```
copy =
```

```
robotics.MonteCarloLocalization with properties:
```

```
    InitialPose: [0 0 0]  
    InitialCovariance: [3x3 double]  
    GlobalLocalization: 0  
    ParticleLimits: [500 5000]  
    SensorModel: [1x1 robotics.LikelihoodFieldSensorModel]  
    MotionModel: [1x1 robotics.OdometryMotionModel]  
    UpdateThresholds: [0.2000 0.2000 0.2000]  
    ResamplingInterval: 1
```

See Also

[robotics.MonteCarloLocalization](#) | [robotics.MonteCarloLocalization.reset](#) | [robotics.MonteCarloLocalization.step](#)

More About

- “Monte Carlo Localization Algorithm”

Introduced in R2016a

getParticles

System object: robotics.MonteCarloLocalization

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.

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.

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);
```

See Also

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

More About

- “Monte Carlo Localization Algorithm”

Introduced in R2016a

release

System object: robotics.MonteCarloLocalization

Package: robotics

Enable property value and input characteristic changes

Syntax

```
release(mcl)
```

Description

`release(mcl)` releases system resources (such as memory, file handles, or hardware connection) of the object, `mcl`. After `release` is called, nontunable properties and input characteristics of `mcl` can change.

Input Arguments

mcl — MonteCarloLocalization object

handle

robotics.MonteCarloLocalization object, specified as an object handle.

Examples

Release 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);
```

Release object to adjust nontunable properties.

```
release(mcl)
```

See Also

[robotics.MonteCarloLocalization](#) | [robotics.MonteCarloLocalization.step](#) | [robotics.MonteCarloLocalization.getParticles](#)

More About

- “Monte Carlo Localization Algorithm”

Introduced in R2016a

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.step](#) | [robotics.MonteCarloLocalization.getParticles](#)

More About

- “Monte Carlo Localization Algorithm”

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 — 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 — 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 — 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 — 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.

Examples

Localize Robot Using Global Localization

Create the Monte Carlo localization object with the `GlobalLocalization` property set to `true`.

```
mcl = robotics.MonteCarloLocalization(map, 'GlobalLocalization', true)
```

Call `step` to get first state estimate.

```
[isUpdated, pose, covariance] = step(mcl, odomPose, ranges, angles);
```

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 a Monte Carlo localization 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
```

- “Localize TurtleBot using Monte Carlo Localization”

See Also

[robotics.MonteCarloLocalization](#) | [robotics.MonteCarloLocalization.getParticles](#)

More About

- “Monte Carlo Localization Algorithm”

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 (-1).

`iOccval = checkOccupancy(map,ij,'grid')` specifies `ij` grid cell indices instead of `xy` locations.

Input Arguments

map — Map representation

OccupancyGrid object

Map representation, specified as a `robotics.OccupancyGrid` object. This object represents the environment of the robot. The object contains a matrix grid with values representing the probability of the occupancy of that cell. Values close to 1 represent a high certainty that the cell contains an obstacle. Values close to 0 represent certainty that the cell is not occupied and obstacle free.

xy — World coordinates

n-by-2 matrix

World coordinates, specified as an n -by-2 matrix of [x y] pairs, where n is the number of world coordinates.

Data Types: `double`

`ij` — Grid positions

n -by-2 vertical array

Grid positions, specified as an n -by-2 matrix of [i j] pairs in [`rows` `cols`] format, where n is the number of grid positions.

Data Types: `double`

Output Arguments

`i0ccva1` — Interpreted occupancy values

n -by-1 column vector

Interpreted occupancy values, returned as an n -by-1 column vector equal in length to `xy` or `ij`.

Occupancy values can be obstacle free (0), occupied (1), or unknown (-1). These values are determined from the actual probability values and the `OccupiedThreshold` and `FreeThreshold` properties of the `map` object.

Examples

Get Occupancy Values and Check Occupancy Status

Access occupancy values and check their occupancy status based on the occupied and free thresholds of the `OccupancyGrid` object.

Create a matrix and populate it with values. Use this matrix to create an occupancy grid.

```
p = 0.5*ones(20,20);  
p(11:20,11:20) = 0.75*ones(10,10);  
map = robotics.OccupancyGrid(p,10);
```

Get the occupancy of different locations and check their occupancy statuses. The occupancy status returns 0 for free space and 1 for occupied space. Unknown values return -1.

```
pocc = getOccupancy(map,[1.5 1])
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.OccupancyGrid](#) | [robotics.OccupancyGrid.getOccupancy](#) | [robotics.BinaryOccupancyGrid](#)

Introduced in R2016b

copy

Class: robotics.OccupancyGrid

Package: robotics

Create copy of occupancy grid

Syntax

```
copyMap = copy(map)
```

Description

`copyMap = copy(map)` creates a deep copy of the `OccupancyGrid` object with the same properties.

Input Arguments

map — Map representation

`OccupancyGrid` object

Map representation, specified as a `robotics.OccupancyGrid` object. This object represents the environment of the robot. The object contains a matrix grid with values representing the probability of the occupancy of that cell. Values close to 1 represent a high certainty that the cell contains an obstacle. Values close to 0 represent certainty that the cell is not occupied and obstacle free.

Output Arguments

copyMap — Copied map representation

`OccupancyGrid` object

Map representation, specified as a `robotics.OccupancyGrid` object. The properties are the same as the input object, `map`, but they have a different object handle.

Examples

Copy Occupancy Grid Map

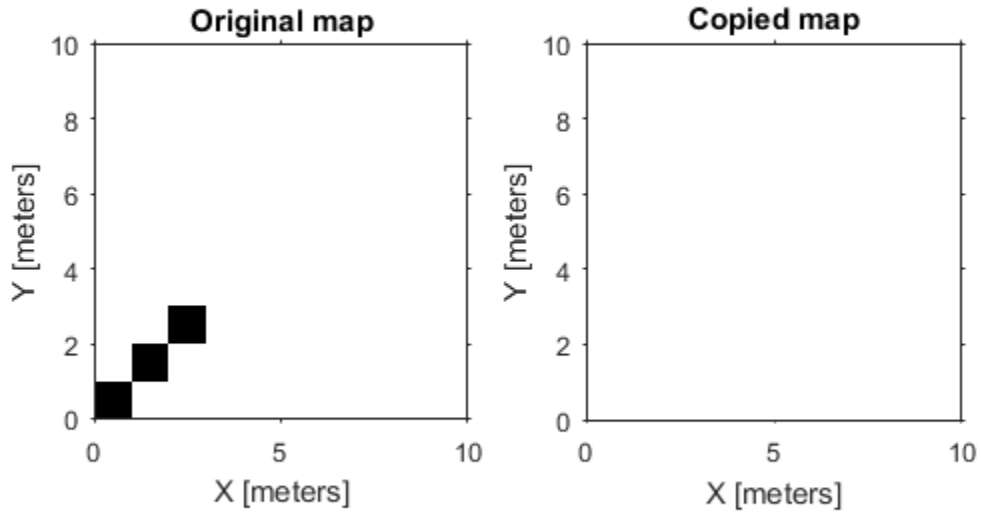
Copy an occupancy grid object. Once copied, the original object can be modified without affecting the copied map.

Create an occupancy grid map. Show the 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.occupancyMatrix](#) | [robotics.OccupancyGrid.getOccupancy](#)

More About

- “Occupancy Grids”

Introduced in R2016b

getOccupancy

Class: robotics.OccupancyGrid

Package: robotics

Get occupancy of a location

Syntax

```
occval = getOccupancy(map, xy)
occval = getOccupancy(map, ij, 'grid')
```

Description

`occval = getOccupancy(map, xy)` returns an array of probability occupancy values at the `xy` locations. Values close to 1 represent a high certainty that the cell contains an obstacle. Values close to 0 represent certainty that the cell is not occupied and obstacle free.

`occval = getOccupancy(map, ij, 'grid')` specifies `ij` grid cell indices instead of `xy` locations.

Input Arguments

map — Map representation

OccupancyGrid object

Map representation, specified as a `robotics.OccupancyGrid` object. This object represents the environment of the robot. The object contains a matrix grid with values representing the probability of the occupancy of that cell. Values close to 1 represent a high certainty that the cell contains an obstacle. Values close to 0 represent certainty that the cell is not occupied and obstacle free.

xy — World coordinates

n-by-2 matrix

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

Data Types: double

ij — Grid positions

n-by-2 matrix

Grid positions, specified as an *n*-by-2 matrix of [i j] pairs in [rows cols] format, where *n* is the number of grid positions.

Data Types: double

Output Arguments

occva1 — 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.

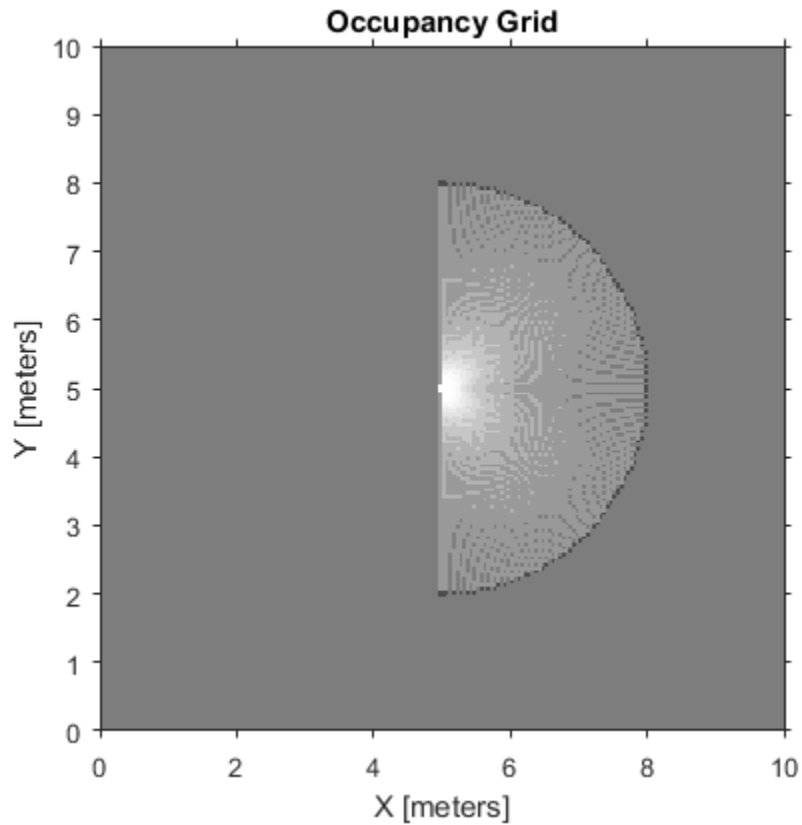
```
pose = [5,5,0];
ranges = 3*ones(100, 1);
angles = linspace(-pi/2, pi/2, 100);
maxrange = 20;
```

```
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
```



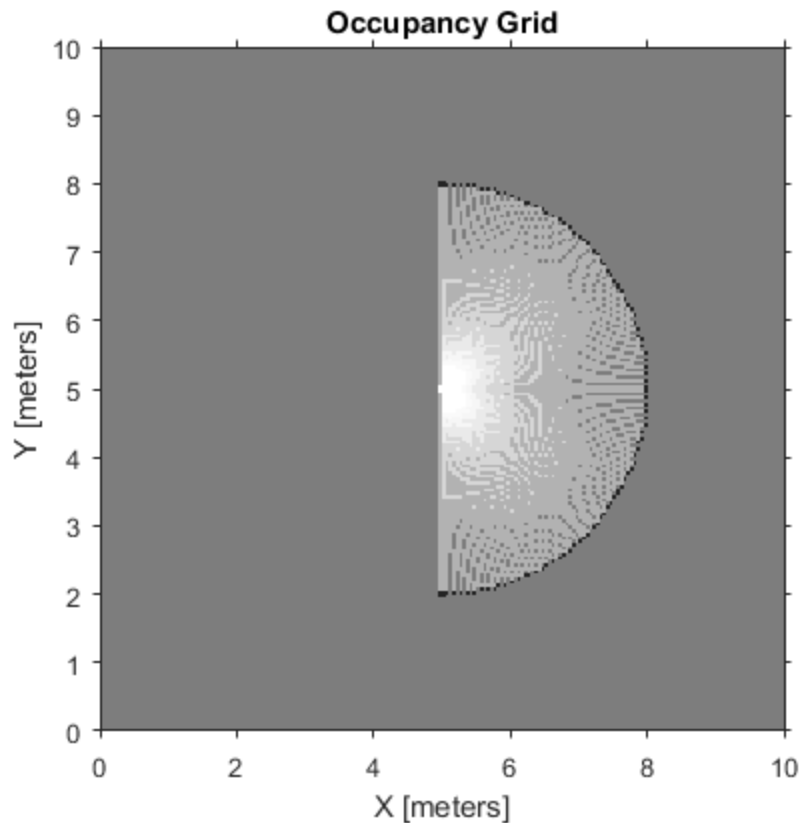
Add a second reading and view the update to the occupancy values. The additional reading increases the confidence in the readings. The free and occupied values become more distinct.

```
insertRay(map,pose,ranges,angles,maxrange);
```

```
show(map)  
getOccupancy(map,[8 5])
```

```
ans =
```

```
0.8448
```



Limitations

Occupancy values have a limited resolution of ± 0.001 . The values are stored as `int16` using a log-odds representation. This data type limits resolution, but saves you memory

when storing large maps in MATLAB. When calling `set` and then `get`, the value returned might not equal the value you set. For more information, see the log-odds representations section in “Occupancy Grids”.

See Also

`robotics.BinaryOccupancyGrid` | `robotics.OccupancyGrid` |
`robotics.OccupancyGrid.checkOccupancy`

More About

- “Occupancy Grids”

Introduced in R2016b

grid2world

Class: robotics.OccupancyGrid

Package: robotics

Convert grid indices to world coordinates

Syntax

```
xy = grid2world(map,ij)
```

Description

`xy = grid2world(map,ij)` converts a `[row col]` array of grid indices, `ij`, to an array of world coordinates, `xy`.

Input Arguments

map — Map representation

OccupancyGrid object

Map representation, specified as a robotics.OccupancyGrid object. This object represents the environment of the robot. The object contains a matrix grid with values representing the probability of the occupancy of that cell. Values close to 1 represent a high certainty that the cell contains an obstacle. Values close to 0 represent certainty that the cell is not occupied and obstacle free.

ij — Grid positions

n-by-2 matrix

Grid positions, specified as an *n*-by-2 matrix of `[i j]` pairs in `[rows cols]` format, where *n* is the number of grid positions. The grid cell locations are counted from the top left corner of the grid.

Data Types: double

Output Arguments

xy — World coordinates

n-by-2 matrix

World coordinates, returned as an *n*-by-2 matrix of [x y] pairs, where *n* is the number of world coordinates.

Data Types: double

See Also

robotics.OccupancyGrid.world2grid | robotics.BinaryOccupancyGrid |
robotics.OccupancyGrid

More About

- “Occupancy Grids”

Introduced in R2016b

inflate

Class: robotics.OccupancyGrid

Package: robotics

Inflate each occupied grid location

Syntax

```
inflate(map, radius)
inflate(map, gridradius, 'grid')
```

Description

`inflate(map, radius)` inflates each occupied position of the specified `map` by the `radius` specified in meters. `radius` is rounded up to the nearest equivalent cell based on the resolution of the map. Values are modified using *grayscale inflation* to inflate higher probability values across the grid. This inflation increases the size of the occupied locations in the map.

`inflate(map, gridradius, 'grid')` inflates each occupied position by the `gridradius` in number of cells.

Input Arguments

map — Map representation

OccupancyGrid object

Map representation, specified as a robotics.OccupancyGrid object. This object represents the environment of the robot. The object contains a matrix grid with values representing the probability of the occupancy of that cell. Values close to 1 represent a high certainty that the cell contains an obstacle. Values close to 0 represent certainty that the cell is not occupied and obstacle free.

radius — Dimension that defines by how much to inflate occupied locations

scalar

Dimension that defines by how much to inflate occupied locations, specified as a scalar. `radius` is rounded up to the nearest cell value.

Data Types: `double`

gridradius — Number of cells by which to inflate the occupied locations

positive scalar

Number of cells by which to inflate the occupied locations, specified as a positive scalar.

Data Types: `double`

Examples

Create and Modify Occupancy Grid

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

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

Update the occupancy of world locations with specific probability values.

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

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

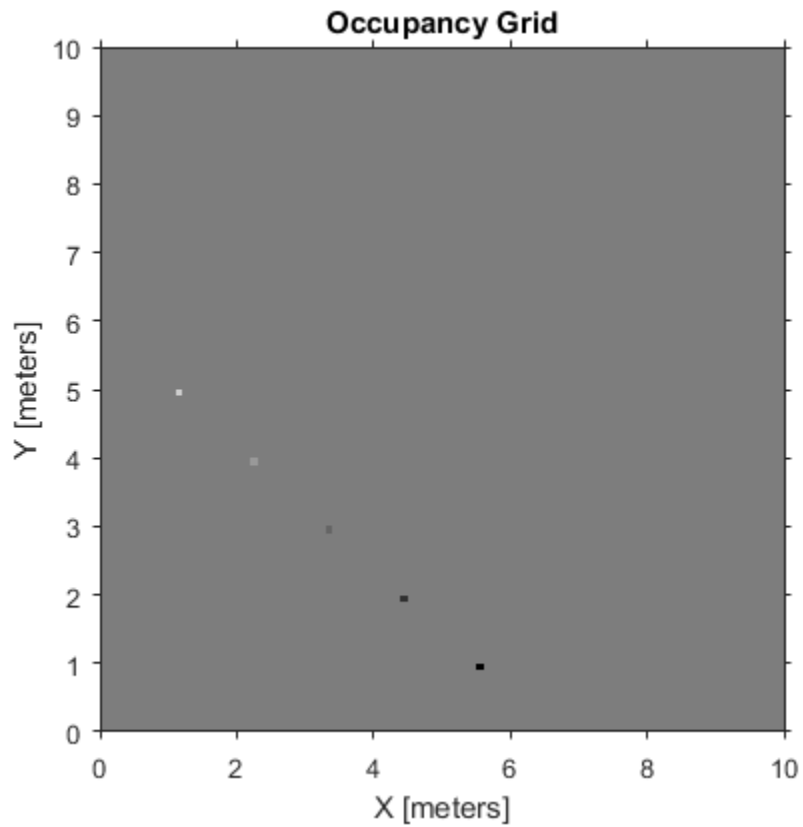
```
y = [5.0; 4.0; 3.0; 2.0; 1.0];
```

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

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

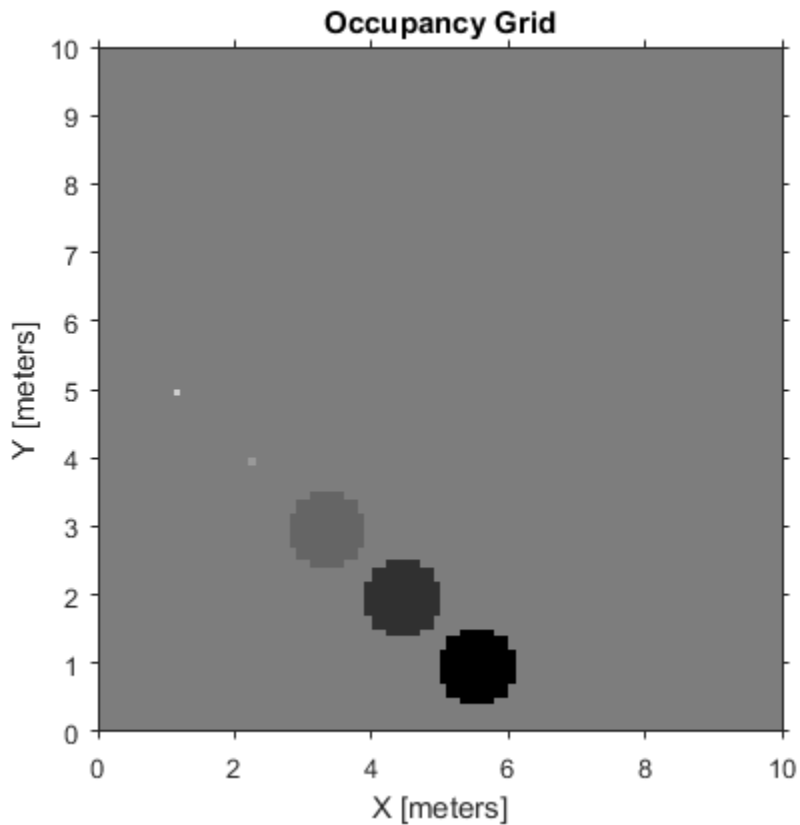
```
figure
```

```
show(map)
```



Inflate occupied areas by a given radius. Larger occupancy values overwrite the smaller values.

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

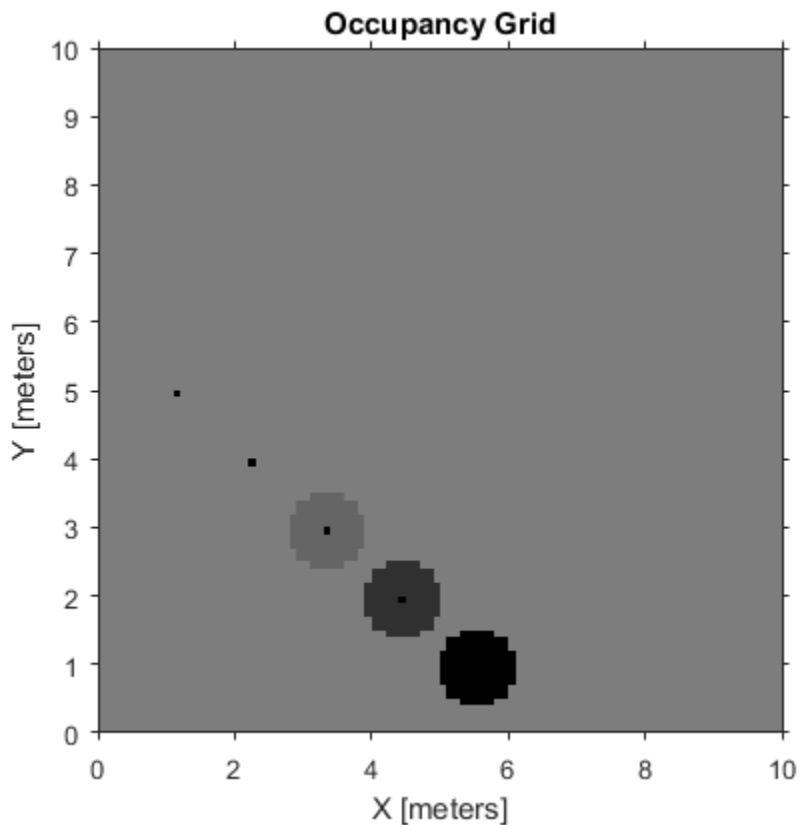


Get grid locations from world locations.

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

Set grid locations to occupied locations.

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



Definitions

In *grayscale inflation*, the `strel` class function creates a circular structuring element using the inflation radius. The grayscale inflation of $A(x, y)$ by $B(x, y)$ is defined as:

$$(A \oplus B)(x, y) = \max \{A(x - x', y - y') + B(x', y') \mid (x', y') \in D_B\}.$$

D_B is the domain of the probability values in the structuring element B . $A(x, y)$ is assumed to be $+\infty$ outside the domain of the grid.

Grayscale inflation acts as a local maximum operator and finds the highest probability values for nearby cells. The `inflate` method uses this definition to inflate the higher

probability values throughout the grid. This inflation increases the size of any occupied locations and creates a buffer zone for robots to use as they navigate.

See Also

`robotics.BinaryOccupancyGrid` | `robotics.OccupancyGrid` |
`robotics.OccupancyGrid.getOccupancy`

More About

- “Occupancy Grids”

Introduced in R2016b

insertRay

Class: robotics.OccupancyGrid

Package: robotics

Insert ray from laser scan observation

Syntax

```
insertRay(map, pose, ranges, angles, maxrange)
```

```
insertRay(map, startpt, endpoints)
```

```
insertRay( ____, invModel)
```

Description

`insertRay(map, pose, ranges, angles, maxrange)` inserts one or more range sensor observations in the occupancy grid, `map` using the input `ranges` and `angles` to get ray endpoints. The ray endpoints are considered free space if the input `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, 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.

ranges — Range values from scan data

vector of scalars

Range values from scan data, specified as a vector of scalars in meters. These range values are distances from a sensor at given `angles`. The vector must be the same length as the corresponding `angles` vector.

angles — Angle values from scan data

vector of scalars

Angle values from scan data, specified as a vector of scalars in radians. These angle values are the specific angles of the given `ranges`. The vector must be the same length as the corresponding `ranges` vector.

maxrange — Maximum range of sensor

scalar

Maximum range of laser range sensor, specified as a scalar. Range values greater than or equal to `maxrange` are considered free along the whole length of the ray, up to `maxrange`.

startpt — Start point for rays

two-element vector

Start point for rays, specified as a two-element vector, [`x` `y`], in the world coordinate frame. All rays are line segments that originate at this point.

endpoints — Endpoints for rays

n -by-2 matrix

Endpoints for rays, specified as an n -by-2 matrix, $[x \ y]$, in the world coordinate frame, where n is the length of `ranges` or `angles`. All rays are line segments that originate at `startpt`.

invModel — Inverse sensor model values

two-element vector

Inverse sensor model values, specified as a two-element vector corresponding to obstacle-free and occupied probabilities. Points along the ray are updated according to the inverse sensor model and the specified range readings. NaN range values are ignored. Range values greater than `maxrange` are not updated. See “Inverse Sensor Model” on page 3-65.

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;
```

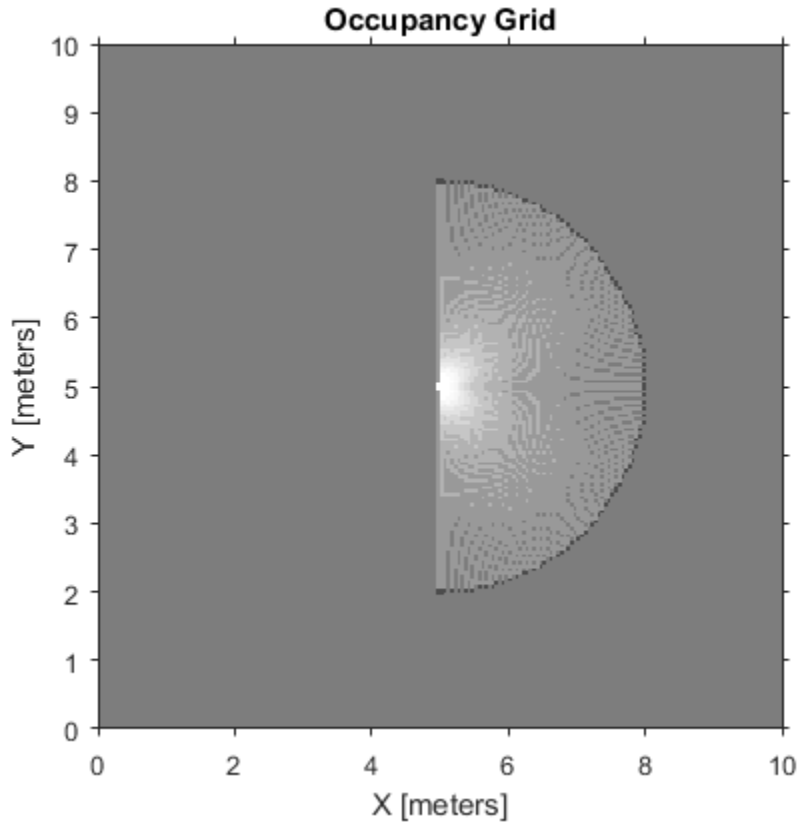
```
insertRay(map,pose,ranges,angles,maxrange);
```

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

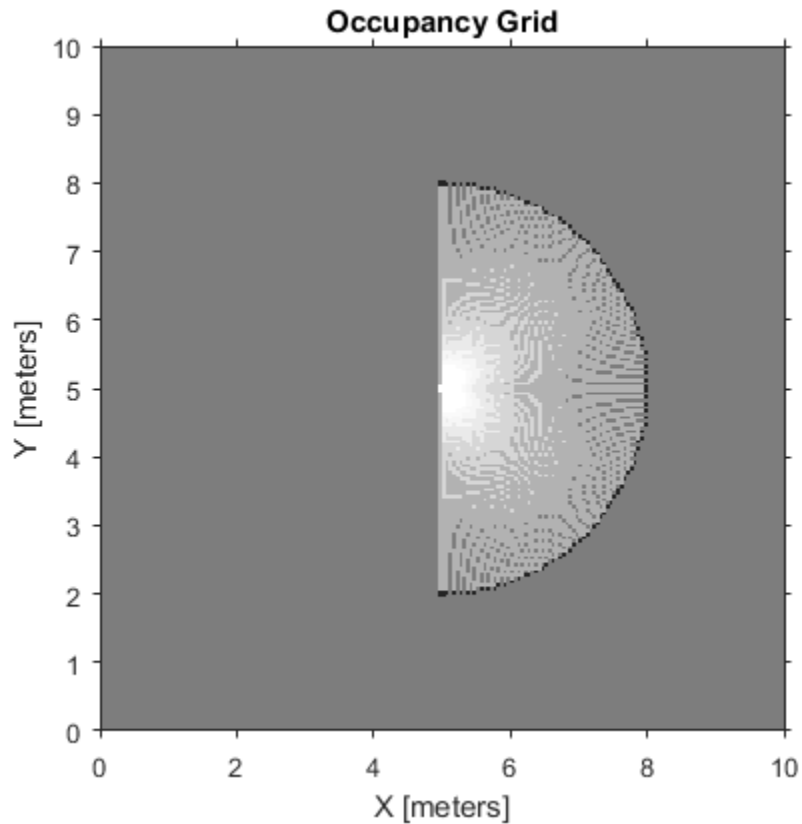


Add a second reading and view the update to the occupancy values. The additional reading increases the confidence in the readings. The free and occupied values become more distinct.

```
insertRay(map,pose,ranges,angles,maxrange);  
show(map)  
getOccupancy(map,[8 5])
```

ans =

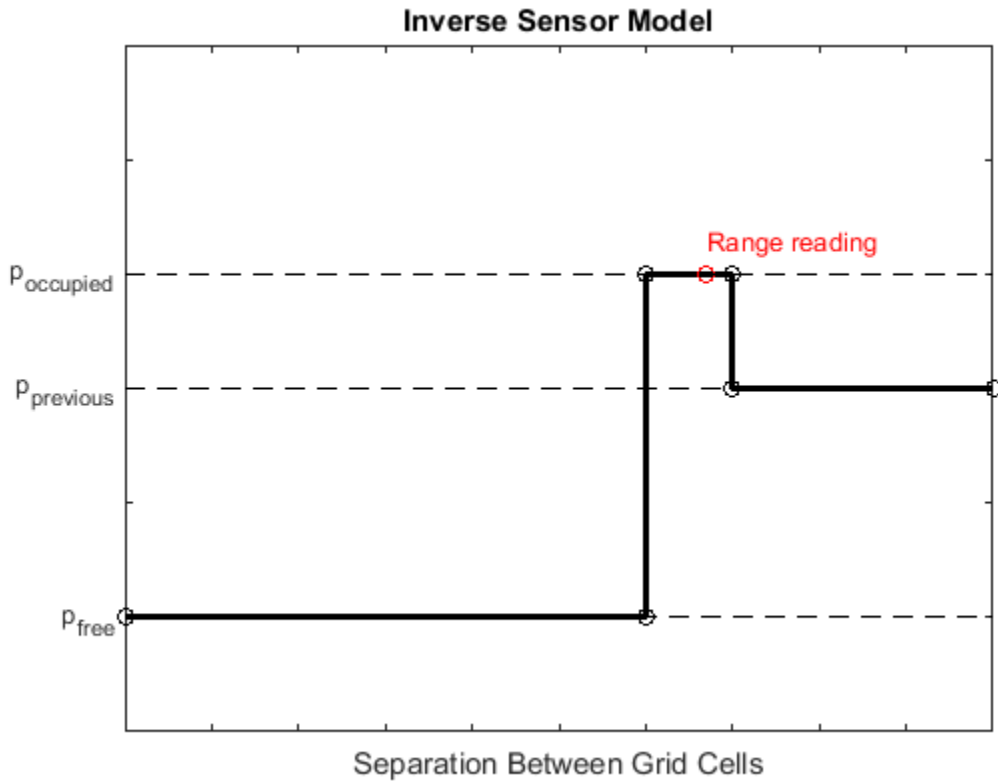
0.8448



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

`robotics.OccupancyGrid` | `robotics.OccupancyGrid.raycast` | `robotics.BinaryOccupancyGrid`

More About

- “Occupancy Grids”

Introduced in R2016b

occupancyMatrix

Class: robotics.OccupancyGrid

Package: robotics

Convert occupancy grid to double matrix

Syntax

```
mat = occupancyMatrix(map)
mat = occupancyMatrix(map, 'ternary')
```

Description

`mat = occupancyMatrix(map)` returns probability values stored in the occupancy grid object as a matrix.

`mat = occupancyMatrix(map, 'ternary')` returns the occupancy status of each grid cell as a matrix. The `OccupiedThreshold` and `FreeThreshold` properties on the occupancy grid determine the obstacle free cells (0) and occupied cells (1). Unknown values are returned as `-1`.

Input Arguments

map — Map representation

OccupancyGrid object

Map representation, specified as a `robotics.OccupancyGrid` object. This object represents the environment of the robot. The object contains a matrix grid with values representing the probability of the occupancy of that cell. Values close to 1 represent a high certainty that the cell contains an obstacle. Values close to 0 represent certainty that the cell is not occupied and obstacle free.

Output Arguments

mat — Occupancy grid values

matrix

Occupancy grid values, returned as an h -by- w matrix, where h and w are defined by the two elements of the `GridSize` property of the occupancy grid object.

See Also

[robotics.OccupancyGrid](#) | [robotics.OccupancyGrid.getOccupancy](#) | [robotics.OccupancyGrid.show](#) | [robotics.BinaryOccupancyGrid](#)

More About

- “Occupancy Grids”

Introduced in R2016b

raycast

Class: `robotics.OccupancyGrid`

Package: `robotics`

Compute cell indices along a ray

Syntax

```
[endpoints,midpoints] = raycast(map,pose,range,angle)
[endpoints,midpoints] = raycast(map,p1,p2)
```

Description

`[endpoints,midpoints] = raycast(map,pose,range,angle)` returns cell indices of the specified map for all cells traversed by a ray originating from the specified `pose` at the specified `angle` and `range` values. `endpoints` contains all indices touched by the end of the ray, with all other points included in `midpoints`.

`[endpoints,midpoints] = raycast(map,p1,p2)` returns the cell indices of the line segment between the two specified points.

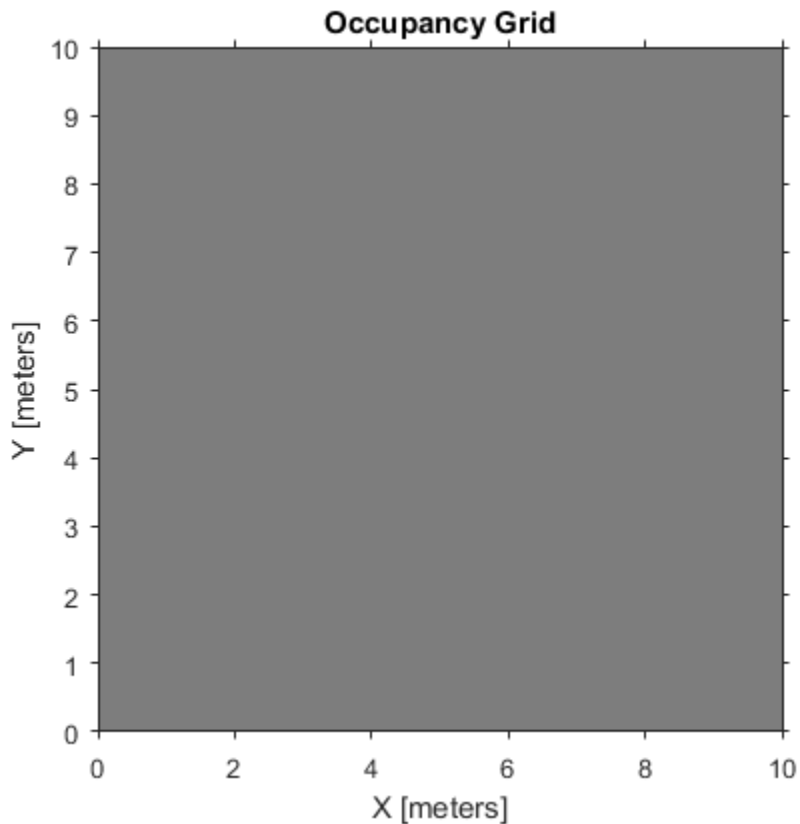
Examples

Get Grid Cells Along A Ray

Use the `raycast` method to generate cell indices for all cells traversed by a ray.

Create an empty map. A low resolution map is used to illustrate the affect of grid locations.

```
map = robotics.OccupancyGrid(10,10,1);
show(map)
```

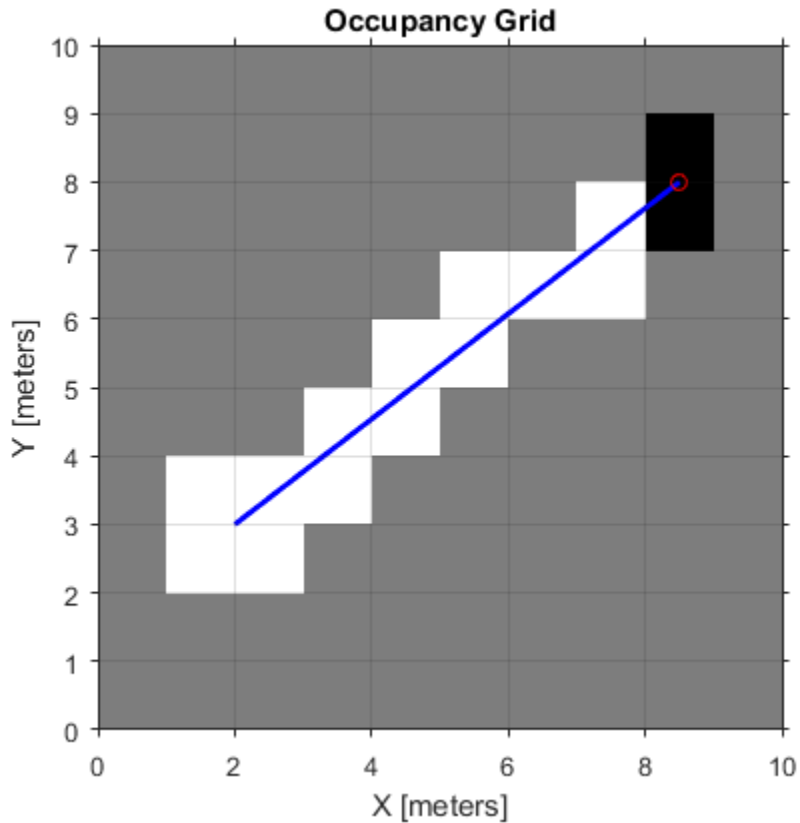



Get the grid indices of the midpoints and end points of a ray from p_1 to p_2 . Set occupancy values for these grid indices. Midpoints are treated as open space. Endpoints are updated with an occupied observation.

```
p1 = [2 3];
p2 = [8.5 8];
[endPts,midPts] = raycast(map,p1,p2);
setOccupancy(map,midPts,zeros(length(midPts),1),'grid');
setOccupancy(map,endPts,ones(length(endPts),1),'grid');
```

Plot the original ray over the map. Notice that each grid cell touched by the line is updated. The starting point overlaps multiple cells and the line touches the edge of certain cells, but all the cells are still updated.

```
show(map);  
hold on  
plot([p1(1) p2(1)],[p1(2) p2(2)], '-b', 'LineWidth',2)  
plot(p2(1),p2(2), 'or')  
grid on
```



Input Arguments

map — Map representation
OccupancyGrid object

Map representation, specified as a `robotics.OccupancyGrid` object. This object represents the environment of the robot. The object contains a matrix grid with values representing

the probability of the occupancy of that cell. Values close to 1 represent a high certainty that the cell contains an obstacle. Values close to 0 represent certainty that the cell is not occupied and obstacle free.

pose — Position and orientation of robot

[*x* *y* *theta*] vector

Position and orientation of robot, specified as an [*x* *y* *theta*] vector. The robot pose is an *x* and *y* position with angular orientation (in radians) measured from the *x*-axis.

range — Range value

scalar

Range value, specified as a scalar in meters.

angle — Angle value

scalar

Angle value, specified as a scalar in radians. The angle value is the specific angle orientation of the given **range**.

p1 — Starting point of ray

[*x* *y*] two-element vector

Starting point of ray, specified as an [*x* *y*] two-element vector. The point is defined in the robot coordinate frame.

p2 — Endpoint of ray

[*x* *y*] two-element vector

Endpoint of ray, specified as an [*x* *y*] two-element vector. The point is defined in the robot coordinate frame.

Output Arguments

endpoints — Endpoint grid indices

[*i* *j*] matrix

Endpoint indices, returned as an [*i* *j*] matrix. The endpoints are where the **range** value hits at the specified **angle**. Multiple indices are only given if the point intersect grid locations.

midpoints — Midpoint grid indices

[i j] matrix

Midpoint indices, returned as an [i j] matrix. This argument includes all grid indices the ray intersects, excluding the endpoint.

See Also

robotics.OccupancyGrid | robotics.OccupancyGrid.insertRay |
robotics.BinaryOccupancyGrid

More About

- “Occupancy Grids”

Introduced in R2016b

rayIntersection

Class: robotics.OccupancyGrid

Package: robotics

Compute map intersection points of rays

Syntax

```
intersectionPts = rayIntersection(map,pose,angles,maxrange)
intersectionPts = rayIntersection(map,pose,angles,maxrange,
threshold)
```

Description

`intersectionPts = rayIntersection(map,pose,angles,maxrange)` returns intersection points in the world coordinate frame of the specified `map` for rays emanating from the specified `pose` with the specified `angles`. If there is no intersection up to the specified `maxrange`, `[NaN NaN]` is returned. By default, the `OccupiedThreshold` property is used to determine occupied cells.

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

Examples

Get Ray Intersection Points on Occupancy Grid

This example shows how to get the ray intersection points on an occupancy grid that has obstacles in the map. The rays are defined ranges and angles from a starting robot pose.

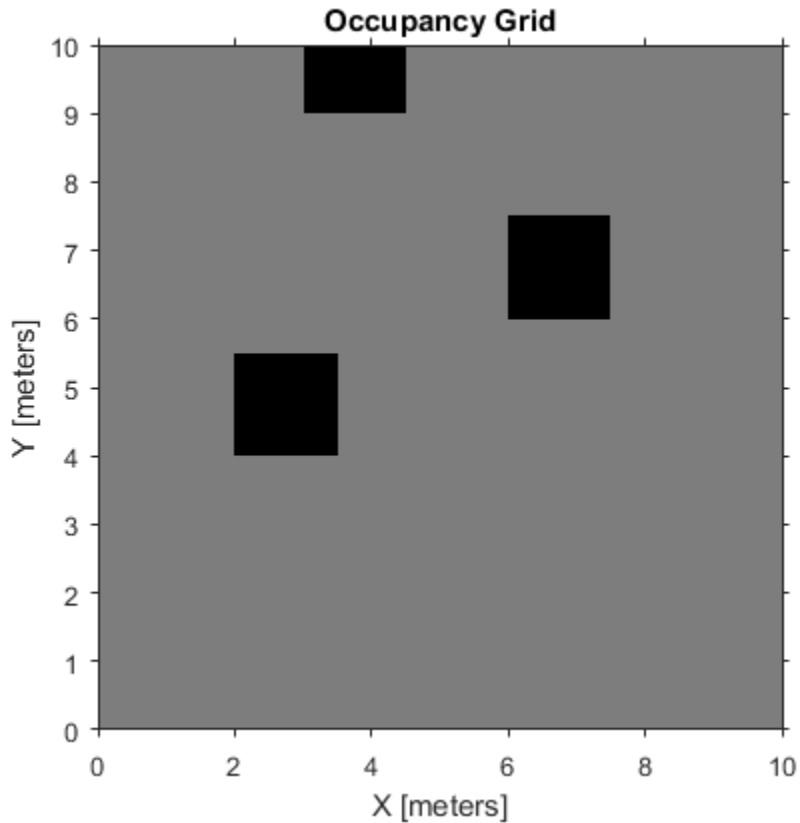
Create occupancy grid. Add obstacles and inflate them. A lower resolution map is used to illustrate the importance of using grid cells. Show the map.

```
map = robotics.OccupancyGrid(10,10,2);
```

```

obstacles = [4 10; 3 5; 7 7];
setOccupancy(map,obstacles,ones(length(obstacles),1))
inflate(map,0.25)
show(map)

```



Find the collision point for rays that emit from the given robot pose. The maxrange and angles for these rays are specified. The last ray does not intersect with an obstacle within the max range, thus it has no collision point.

```

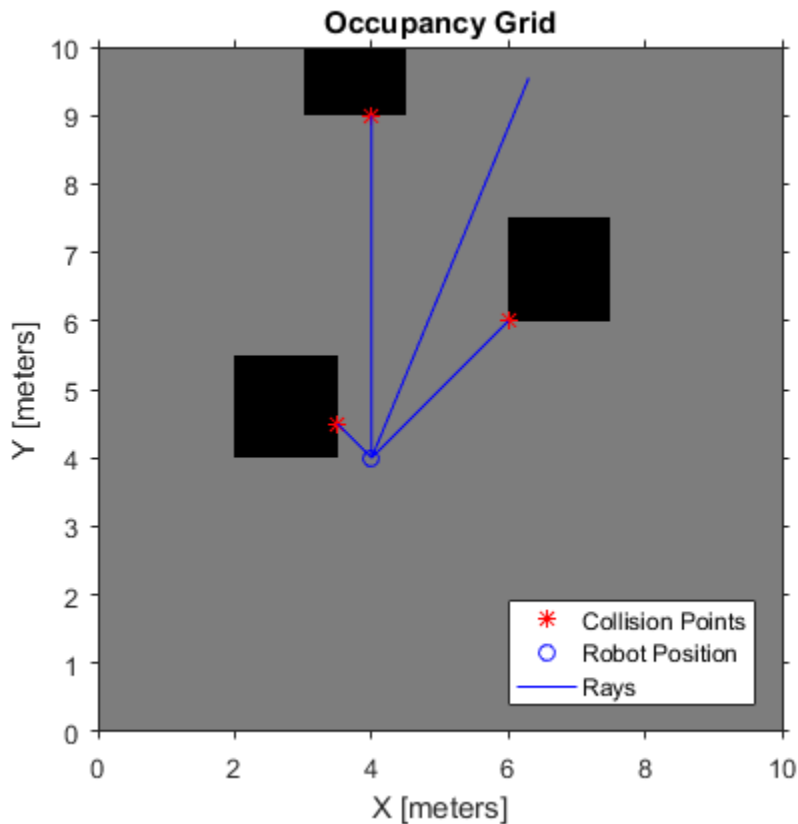
maxrange = 6;
angles = [pi/4,-pi/4,0,-pi/8];
robotPose = [4,4,pi/2];
collisionPts = rayIntersection(map,robotPose,angles,maxrange,0.7)

```

```
collisionPts =  
  
    3.5000    4.5000  
    6.0000    6.0000  
    4.0000    9.0000  
     NaN     NaN
```

Plot the collision points and rays from the pose.

```
hold on  
plot(collisionPts(:,1),collisionPts(:,2) , '*r') % Collision points  
plot(robotPose(1),robotPose(2),'ob') % Robot pose  
for i = 1:3  
    plot([robotPose(1),collisionPts(i,1)],...  
         [robotPose(2),collisionPts(i,2)], '-b') % Plot collision rays  
end  
plot([robotPose(1),robotPose(1)-6*sin(angles(4))],...  
     [robotPose(2),robotPose(2)+6*cos(angles(4))], '-b') % No collision ray  
legend('Collision Points','Robot Position','Rays','Location','SouthEast')
```



Input Arguments

map — Map representation

OccupancyGrid object

Map representation, specified as a `robotics.OccupancyGrid` object. This object represents the environment of the robot. The object contains a matrix grid with values representing the probability of the occupancy of that cell. Values close to 1 represent a high certainty that the cell contains an obstacle. Values close to 0 represent certainty that the cell is not occupied and obstacle free.

pose — Position and orientation of robot

[*x y theta*] vector

Position and orientation of robot, specified as an [*x y theta*] vector. The robot pose is an *x* and *y* position with angular orientation (in radians) measured from the *x*-axis.

angles — Ray angles emanating from the robot

vector in radians

Ray angles emanating from the robot, specified as a vector in radians. These angles are relative to the specified robot **pose**.

maxrange — Maximum range of sensor

scalar

Maximum range of laser range sensor, specified as a scalar. Range values greater than or equal to **maxrange** are considered free along the whole length of the ray, up to **maxrange**.

threshold — Threshold for occupied cells

scalar from 0 to 1

Threshold for occupied cells, specified as a scalar from 0 to 1. Occupancy values greater than or equal to the threshold are treated as occupied cells to trigger intersections.

Output Arguments

intersectionPts — Intersection points

n-by-2 matrix

Intersection points, returned as *n*-by-2 matrix, [*x y*], in the world coordinate frame, where *n* is the length of **angles**.

See Also

[robotics.OccupancyGrid](#) | [robotics.OccupancyGrid.raycast](#) |
[robotics.OccupancyGrid.updateOccupancy](#) | [robotics.BinaryOccupancyGrid](#)

More About

- “Occupancy Grids”

Introduced in R2016b

setOccupancy

Class: robotics.OccupancyGrid

Package: robotics

Set occupancy of a location

Syntax

```
setOccupancy(map, xy, occval)  
setOccupancy(map, ij, occval, 'grid')
```

Description

`setOccupancy(map, xy, occval)` assigns the occupancy values to each coordinate specified in `xy`. `occval` can be an array the length of `xy` or a scalar, which is applied to all coordinates.

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

Input Arguments

map — Map representation

OccupancyGrid object

Map representation, specified as a `robotics.OccupancyGrid` object. This object represents the environment of the robot. The object contains a matrix grid with values representing the probability of the occupancy of that cell. Values close to 1 represent a high certainty that the cell contains an obstacle. Values close to 0 represent certainty that the cell is not occupied and obstacle free.

xy — World coordinates

n-by-2 matrix

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

Data Types: `double`

`ij` — Grid positions

n-by-2 vertical array

Grid positions, specified as an *n*-by-2 matrix of [`i j`] pairs in [`rows cols`] format, where *n* is the number of grid positions.

Data Types: `double`

`occva1` — Probability occupancy values

scalar | column vector

Probability occupancy values, specified as a scalar or a column vector the same size as either `xy` or `ij`. A scalar input is applied to all coordinates in either `xy` or `ij`.

Values close to 0 represent certainty that the cell is not occupied and obstacle free.

Examples

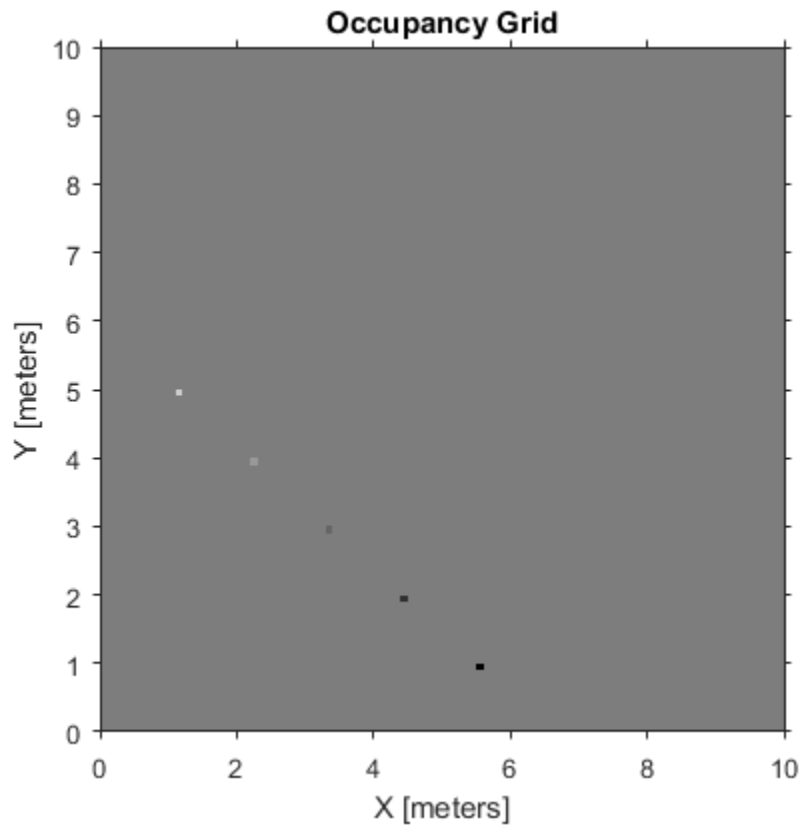
Create and Modify Occupancy Grid

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

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

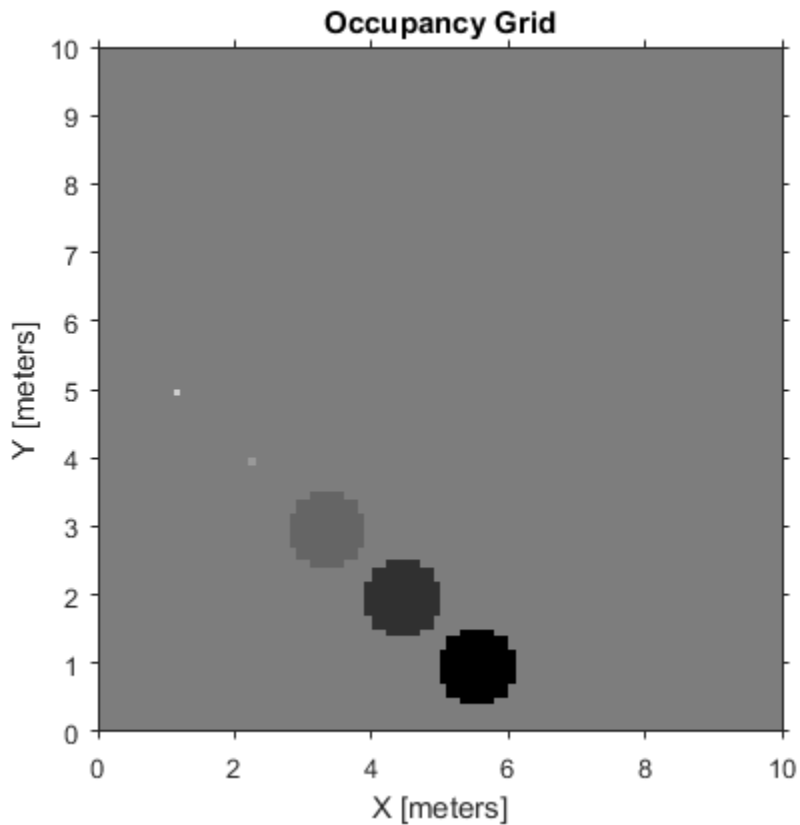
Update the occupancy of world locations with specific probability values.

```
map = robotics.OccupancyGrid(10,10,10);  
x = [1.2; 2.3; 3.4; 4.5; 5.6];  
y = [5.0; 4.0; 3.0; 2.0; 1.0];  
  
pvalues = [0.2 0.4 0.6 0.8 1];  
  
updateOccupancy(map,[x y],pvalues)  
figure  
show(map)
```



Inflate occupied areas by a given radius. Larger occupancy values overwrite the smaller values.

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

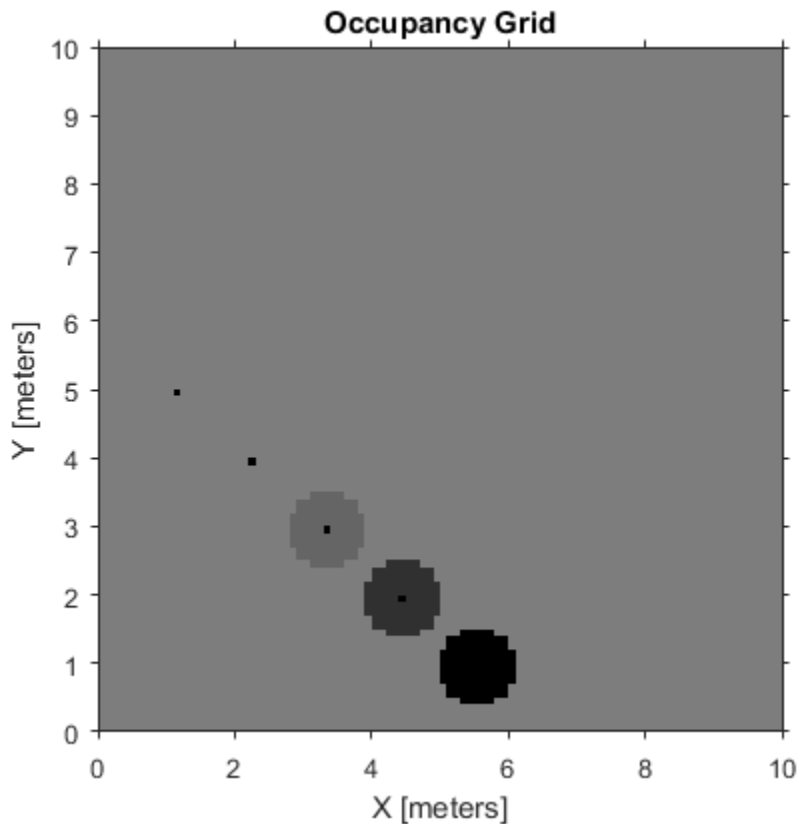


Get grid locations from world locations.

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

Set grid locations to occupied locations.

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



Limitations

Occupancy values have a limited resolution of ± 0.001 . The values are stored as `int16` using a log-odds representation. This data type limits resolution, but saves you memory when storing large maps in MATLAB. When calling `set` and then `get`, the value returned might not equal the value you set. For more information, see the log-odds representations section in “Occupancy Grids”.

See Also

`robotics.OccupancyGrid` | `robotics.OccupancyGrid.getOccupancy` |
`robotics.BinaryOccupancyGrid`

More About

- “Occupancy Grids”

Introduced in R2016b

show

Class: robotics.OccupancyGrid

Package: robotics

Show grid values in a figure

Syntax

```
show(map)
```

```
show(map, 'grid')
```

```
show( ____, 'Parent', parent)
```

```
mapImage= show(map, ____)
```

Description

`show(map)` displays the occupancy grid `map` in the current axes, with the axes labels representing the world coordinates.

`show(map, 'grid')` displays the occupancy grid with the axes labels representing the grid coordinates.

`show(____, 'Parent', parent)` uses the axes handle specified as a parent to display the occupancy grid. Use any of the arguments from previous syntaxes.

`mapImage= show(map, ____)` returns the handle to the image object created by `show`.

Input Arguments

map — Map representation

OccupancyGrid object

Map representation, specified as a `robotics.OccupancyGrid` object. This object represents the environment of the robot. The object contains a matrix grid with values representing

the probability of the occupancy of that cell. Values close to 1 represent a high certainty that the cell contains an obstacle. Values close to 0 represent certainty that the cell is not occupied and obstacle free.

parent — Axes handle

handle

Axes handle to plot the map on.

Outputs

mapImage — Map image

object handle

Map image, specified as an object handle.

Examples

Create and Modify Occupancy Grid

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

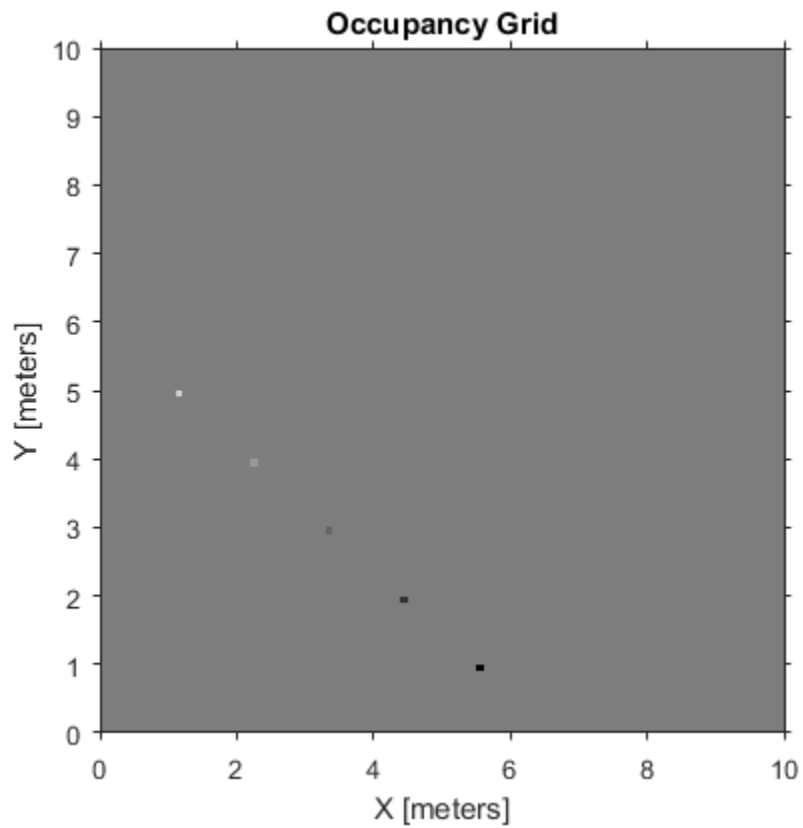
```
map = robotics.OccupancyGrid(10,10,10);
```

Update the occupancy of world locations with specific probability values.

```
map = robotics.OccupancyGrid(10,10,10);  
x = [1.2; 2.3; 3.4; 4.5; 5.6];  
y = [5.0; 4.0; 3.0; 2.0; 1.0];
```

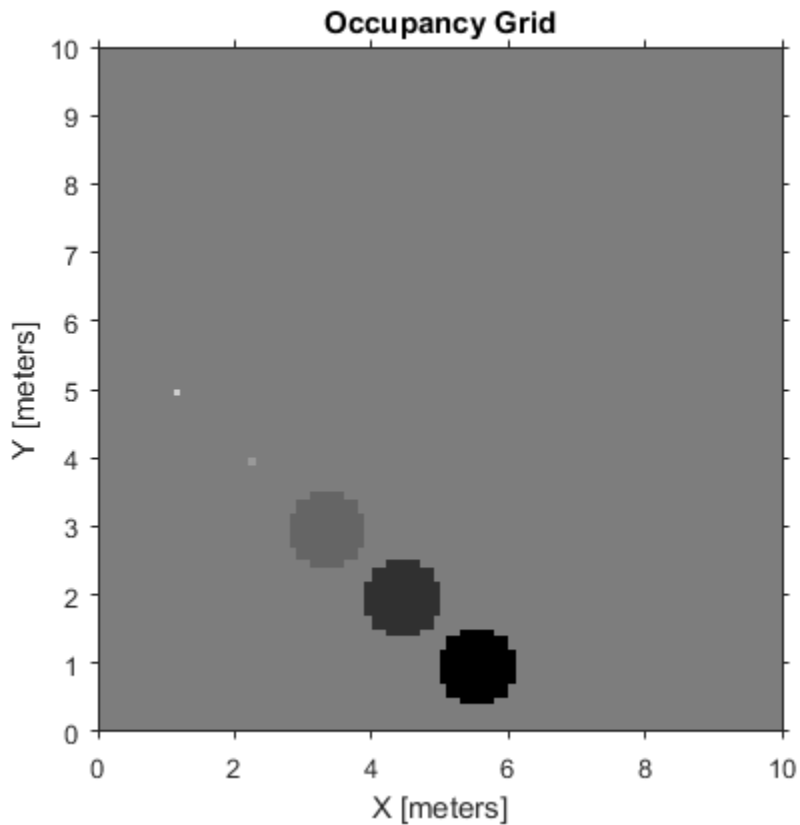
```
pvalues = [0.2 0.4 0.6 0.8 1];
```

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



Inflate occupied areas by a given radius. Larger occupancy values overwrite the smaller values.

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

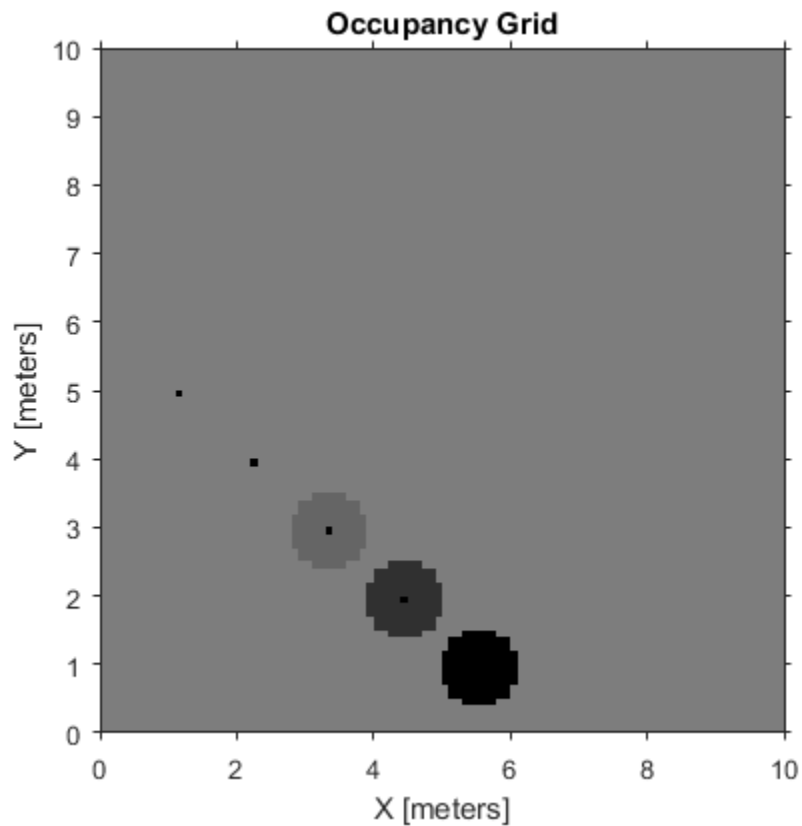


Get grid locations from world locations.

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

Set grid locations to occupied locations.

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

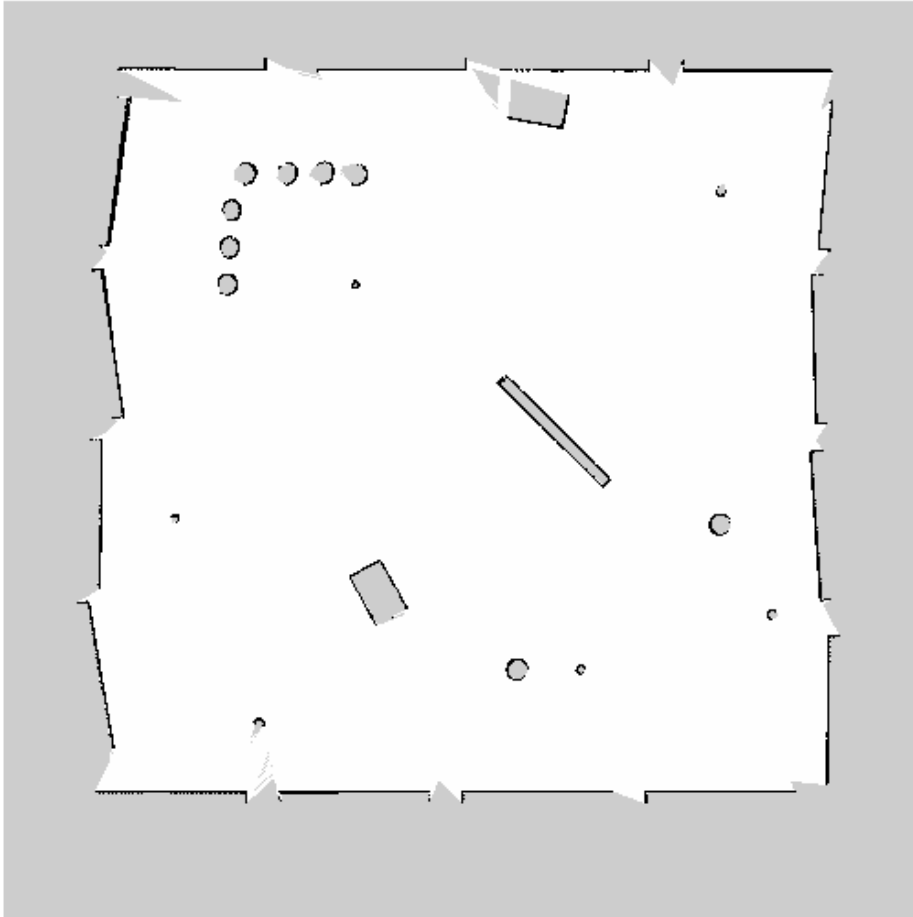


Convert PGM Image to Map

Convert a portable graymap (.pgm) file containing a ROS map into an `OccupancyGrid` map for use in MATLAB.

Import the image using `imread`. Crop the image to the relevant area.

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

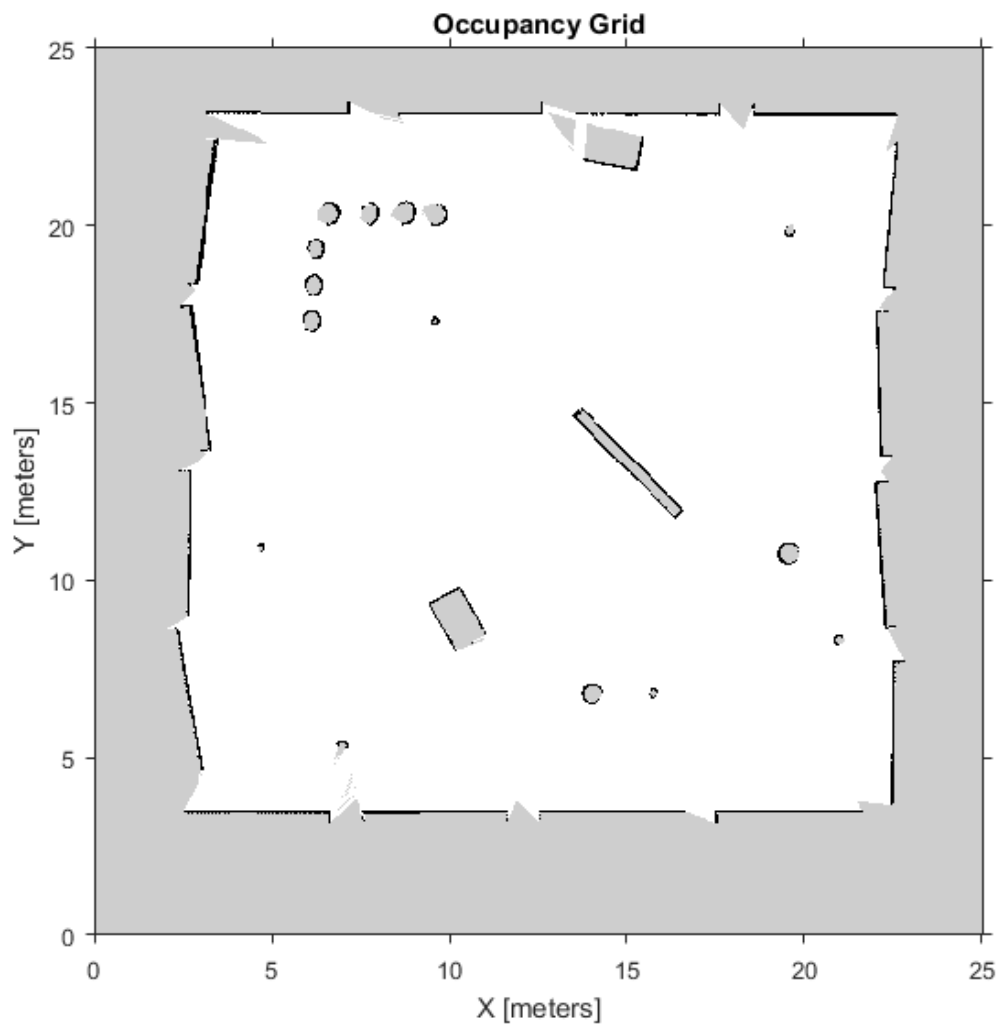


PGM values are expressed from 0 to 255 as `uint8`. Normalize these values by converting the cropped image to `double` and dividing each cell by 255. This image shows obstacles as values close to 0. Subtract the normalized image from 1 to get occupancy values with 1 representing occupied space.

```
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)
```



See Also

[robotics.OccupancyGrid](#) | [robotics.OccupancyGrid.occupancyMatrix](#) | [robotics.BinaryOccupancyGrid](#)

Introduced in R2016b

updateOccupancy

Class: robotics.OccupancyGrid

Package: robotics

Integrate probability observation at a location

Syntax

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

Description

`updateOccupancy(map, xy, obs)` probabilistically integrates the observation values, `obs`, to each coordinate specified in `xy`. Observation values are determined based on the “Inverse Sensor Model” on page 3-100.

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

Input Arguments

map — Map representation

OccupancyGrid object

Map representation, specified as a `robotics.OccupancyGrid` object. This object represents the environment of the robot. The object contains a matrix grid with values representing the probability of the occupancy of that cell. Values close to 1 represent a high certainty that the cell contains an obstacle. Values close to 0 represent certainty that the cell is not occupied and obstacle free.

xy — World coordinates

n-by-2 matrix

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

Data Types: `double`

ij — Grid positions

n-by-2 matrix

Grid positions, specified as an *n*-by-2 matrix of [*i* *j*] pairs in [`rows` `cols`] format, where *n* is the number of grid positions.

Data Types: `double`

obs — Probability observation values

n-by-1 column vector | scalar | logical

Probability observation values, specified as a scalar or an *n*-by-1 column vector the same size as either `xy` or `ij`.

`obs` values can be any value from 0 to 1, but if `obs` is a logical array, the default observation values of 0.7 (`true`) and 0.4 (`false`) are used. These values correlate to the inverse sensor model for ray casting. If `obs` is a scalar or logical, the value is applied to all coordinates in `xy` or `ij`.

Examples

Create and Modify Occupancy Grid

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

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

Update the occupancy of world locations with specific probability values.

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

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

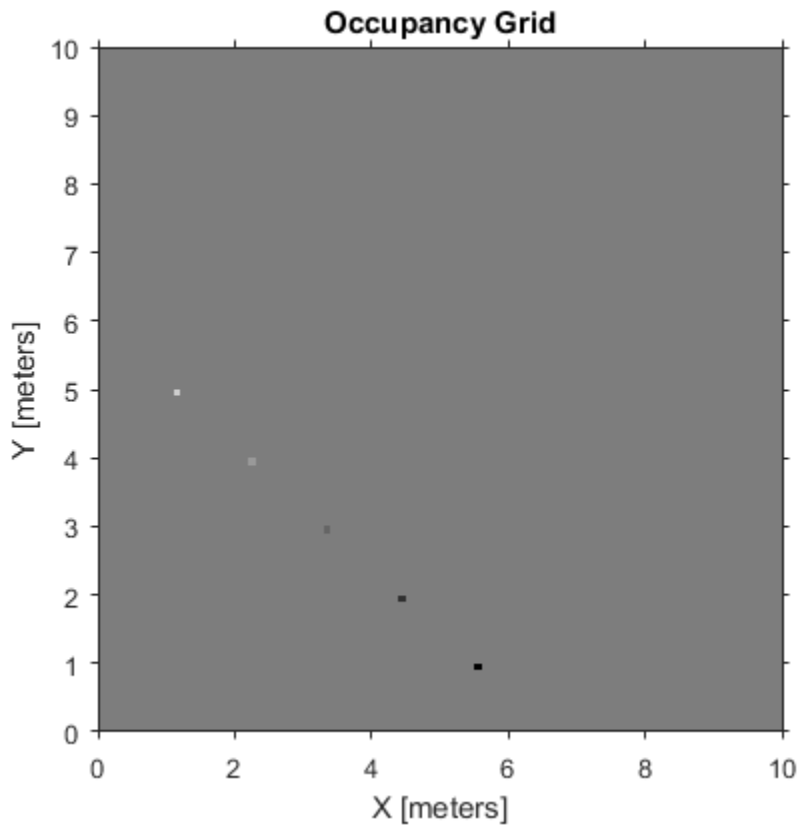
```
y = [5.0; 4.0; 3.0; 2.0; 1.0];
```

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

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

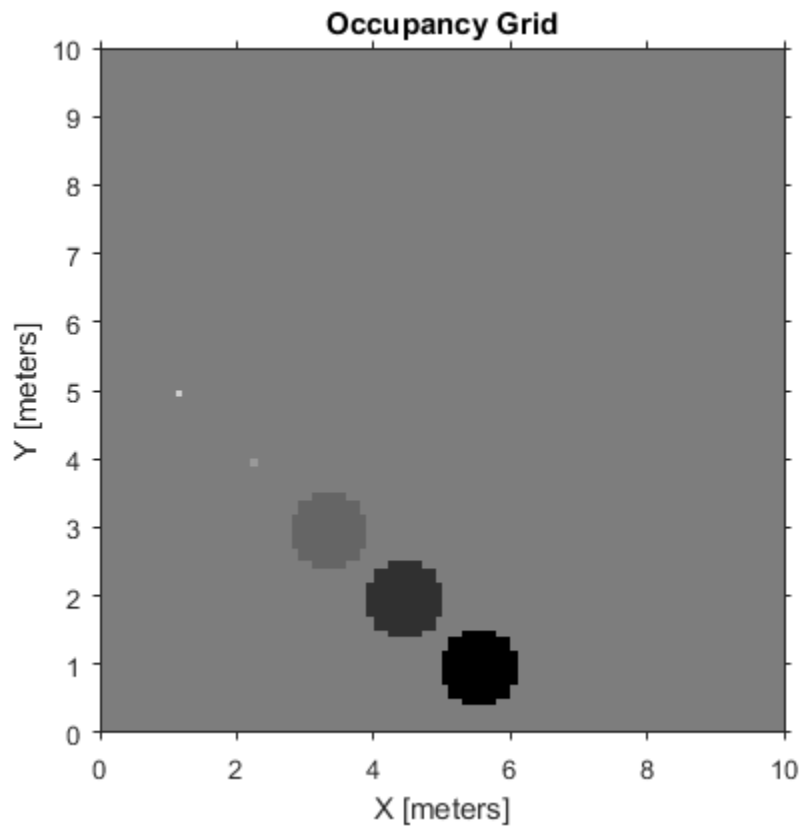
```
figure
```

```
show(map)
```



Inflate occupied areas by a given radius. Larger occupancy values overwrite the smaller values.

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

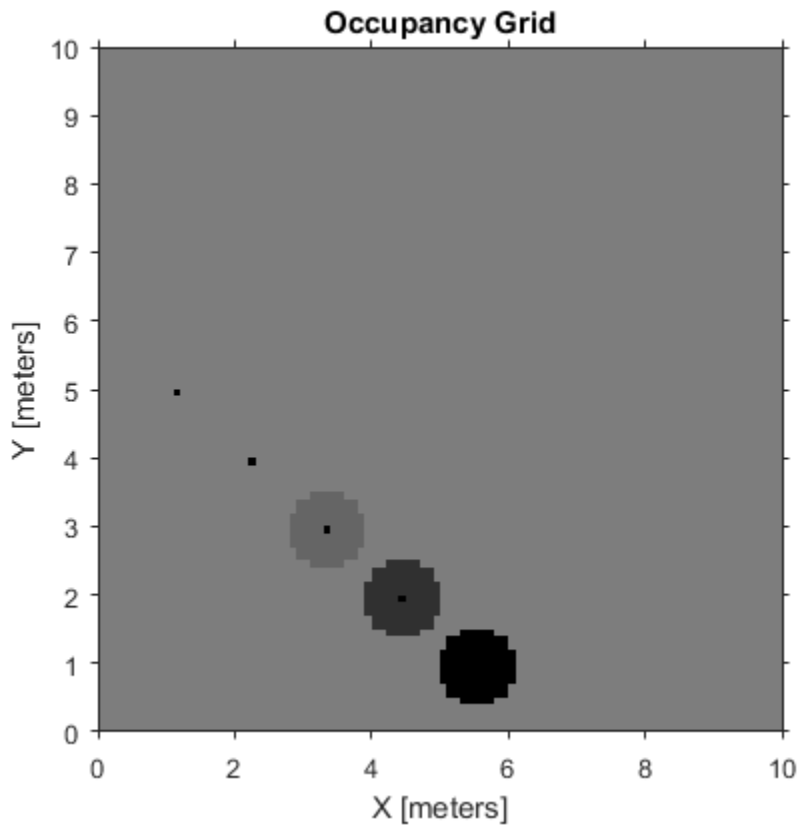


Get grid locations from world locations.

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

Set grid locations to occupied locations.

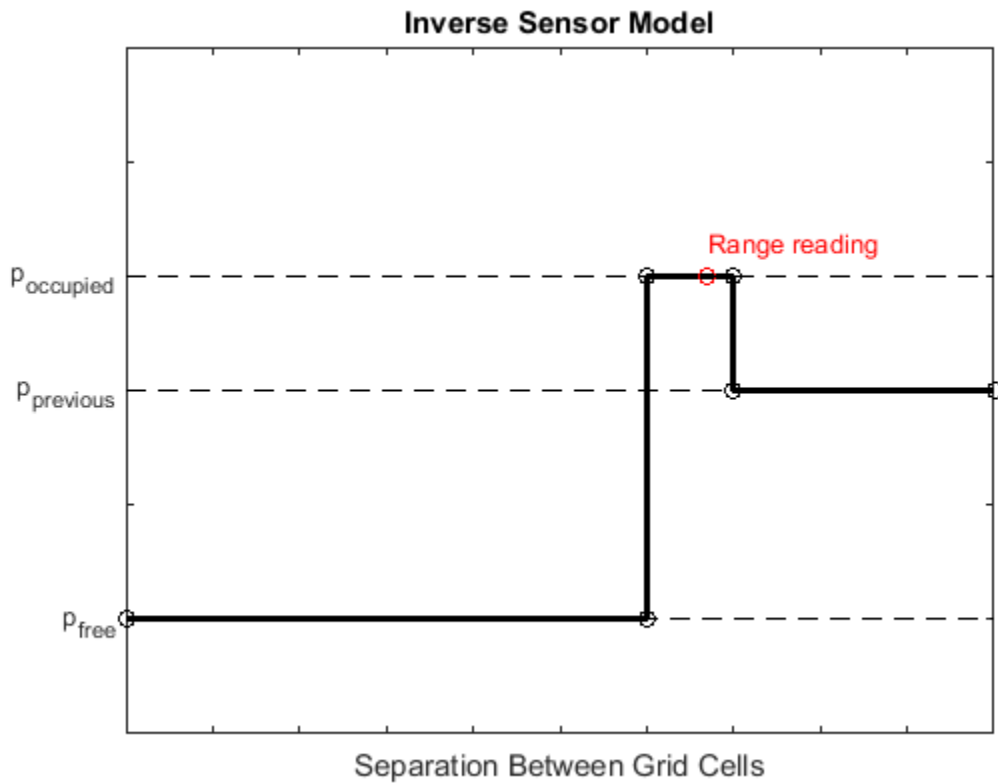
```
setOccupancy(map,ij,ones(5,1), 'grid')  
figure  
show(map)
```



Definitions

Inverse Sensor Model

The inverse sensor model determines how values are set along a ray from a range sensor reading to the obstacles in the map. NaN range values are ignored. Range values greater than `maxrange` are not updated.



Grid locations that contain range readings are updated with the occupied probability. Locations before the reading are updated with the free probability. All locations after the reading are not updated.

See Also

`robotics.OccupancyGrid` | `robotics.OccupancyGrid.setOccupancy` |
`robotics.BinaryOccupancyGrid`

More About

- “Occupancy Grids”

Introduced in R2016b

world2grid

Class: robotics.OccupancyGrid

Package: robotics

Convert world coordinates to grid indices

Syntax

```
ij = world2grid(map,xy)
```

Description

`ij = world2grid(map,xy)` converts an array of world coordinates, `xy`, to an array of grid indices, `ij` in `[row col]` format.

Input Arguments

map — Map representation

OccupancyGrid object

Map representation, specified as a `robotics.OccupancyGrid` object. This object represents the environment of the robot. The object contains a matrix grid with values representing the probability of the occupancy of that cell. Values close to 1 represent a high certainty that the cell contains an obstacle. Values close to 0 represent certainty that the cell is not occupied and obstacle free.

xy — World coordinates

n-by-2 matrix

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

Data Types: `double`

Output Arguments

ij — Grid positions

n-by-2 matrix

Grid positions, returned as an *n*-by-2 matrix of [*i* *j*] pairs in [*rows* *cols*] format, where *n* is the number of grid positions. The grid cell locations are counted from the top left corner of the grid.

Data Types: double

Examples

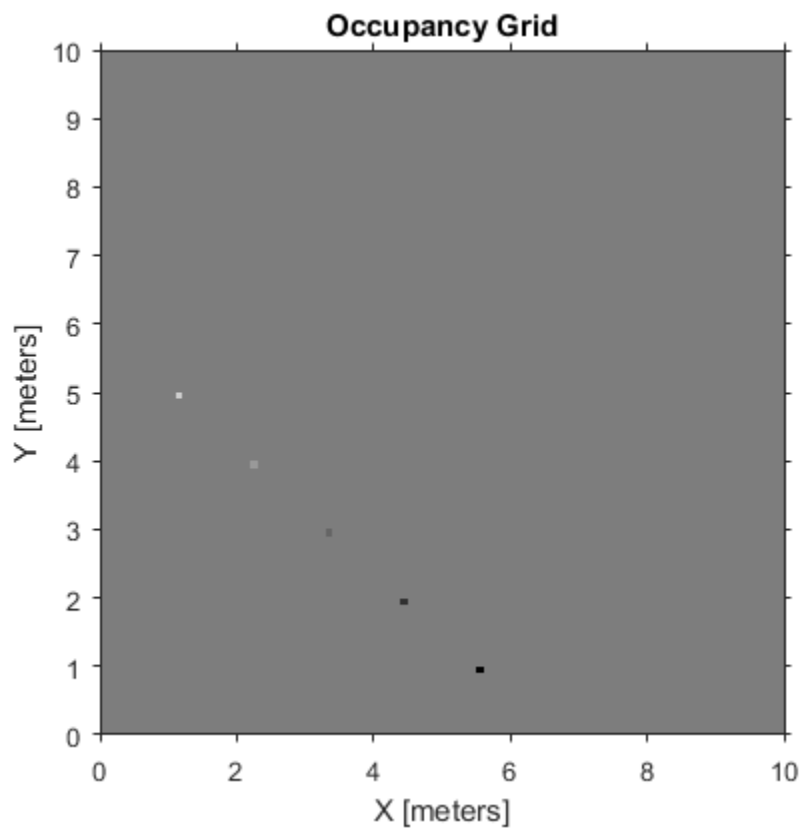
Create and Modify Occupancy Grid

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

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

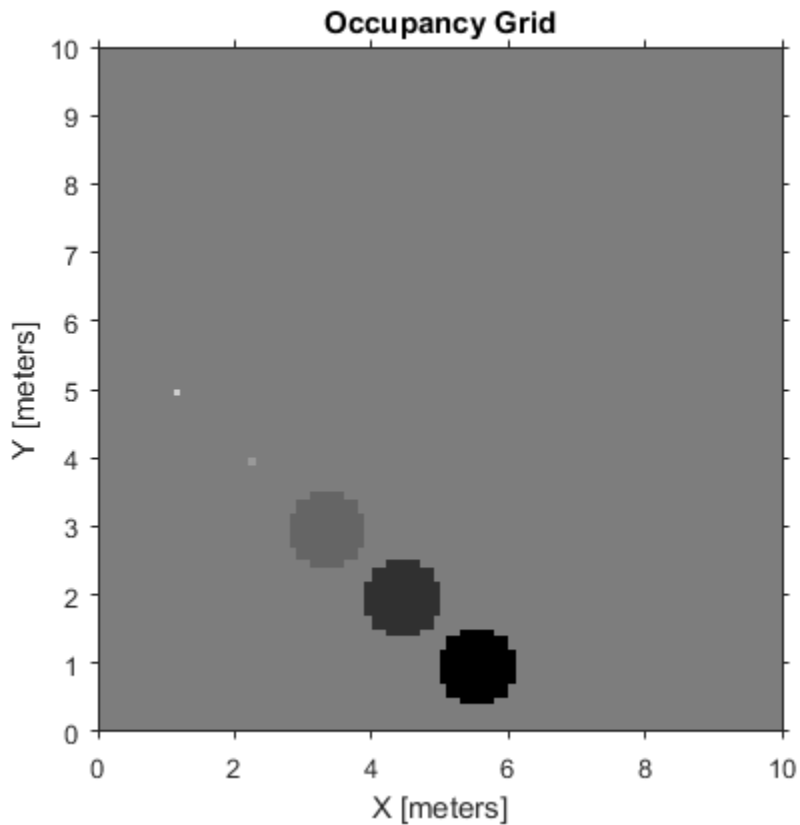
Update the occupancy of world locations with specific probability values.

```
map = robotics.OccupancyGrid(10,10,10);  
x = [1.2; 2.3; 3.4; 4.5; 5.6];  
y = [5.0; 4.0; 3.0; 2.0; 1.0];  
  
pvalues = [0.2 0.4 0.6 0.8 1];  
  
updateOccupancy(map,[x y],pvalues)  
figure  
show(map)
```



Inflate occupied areas by a given radius. Larger occupancy values overwrite the smaller values.

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

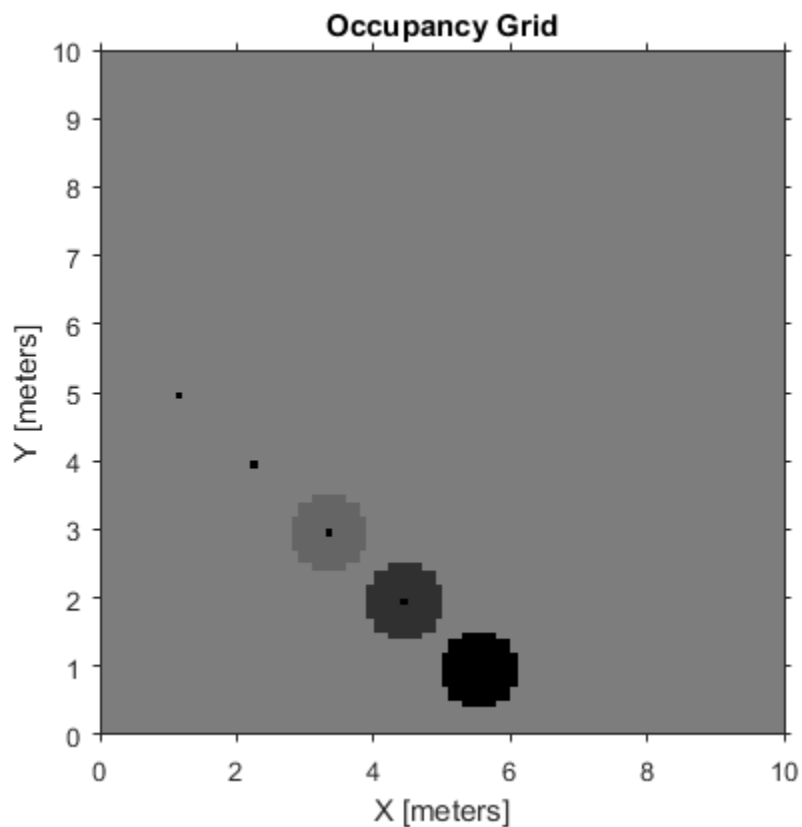


Get grid locations from world locations.

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

Set grid locations to occupied locations.

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



See Also

[robotics.OccupancyGrid.grid2world](#) | [robotics.BinaryOccupancyGrid](#) | [robotics.OccupancyGrid](#)

More About

- “Occupancy Grids”

Introduced in R2016b

showNoiseDistribution

Class: robotics.OdometryMotionModel

Package: robotics

Display noise parameter effects

Syntax

```
showNoiseDistribution(ommObj)
showNoiseDistribution(ommObj)
showNoiseDistribution(ommObj,Name,Value)
```

Description

`showNoiseDistribution(ommObj)` shows the noise distribution for a default odometry pose update, number of samples and the current noise parameters on the input object.

`axes = showNoiseDistribution(ommObj)` shows the noise distribution and returns the axes handle.

`showNoiseDistribution(ommObj,Name,Value)` provides additional options specified by one or more `Name,Value` pairs. `Name` is the property name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name-value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`. Properties not specified retain their default values.

Input Arguments

ommObj — **OdometryMotionModel** object

handle

OdometryMotionModel object, specified as a handle. Create this object using `robotics.OdometryMotionModel`.

Name-Value Pair Arguments

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

'OdometryPoseChange' — Change in odometry

three-element vector

Change in odometry of the robot, specified as a comma-separated pair consisting of 'OdometryPoseChange' and a three-element vector, `[x y theta]`.

'NumSamples' — Number of particles to display

scalar

Number of particles to display, specified as a specified as a comma-separated pair consisting of 'NumSamples' and a scalar.

'Parent' — Axes handle

handle

Axes handle that specifies the parent of the figure object created by `show`, specified as a comma-separated pair consisting of 'Parent' and a handle.

Examples

Show Noise Distribution Effects for Odometry Motion Model

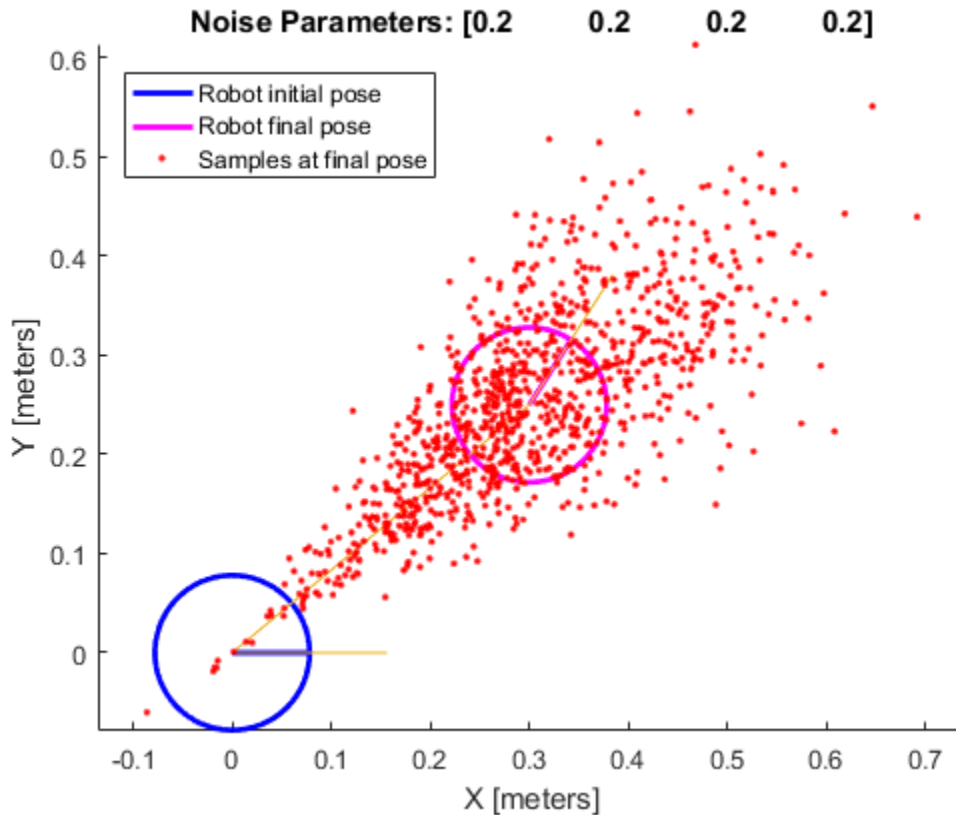
This example shows how to visualize the effect of different noise parameters on the `robotics.OdometryMotionModel` class. An `OdometryMotionModel` object contains the motion model noise parameters for a differential drive robot. Use `showNoiseDistribution` to visualize how changing these values affect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

Create a motion model object.

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

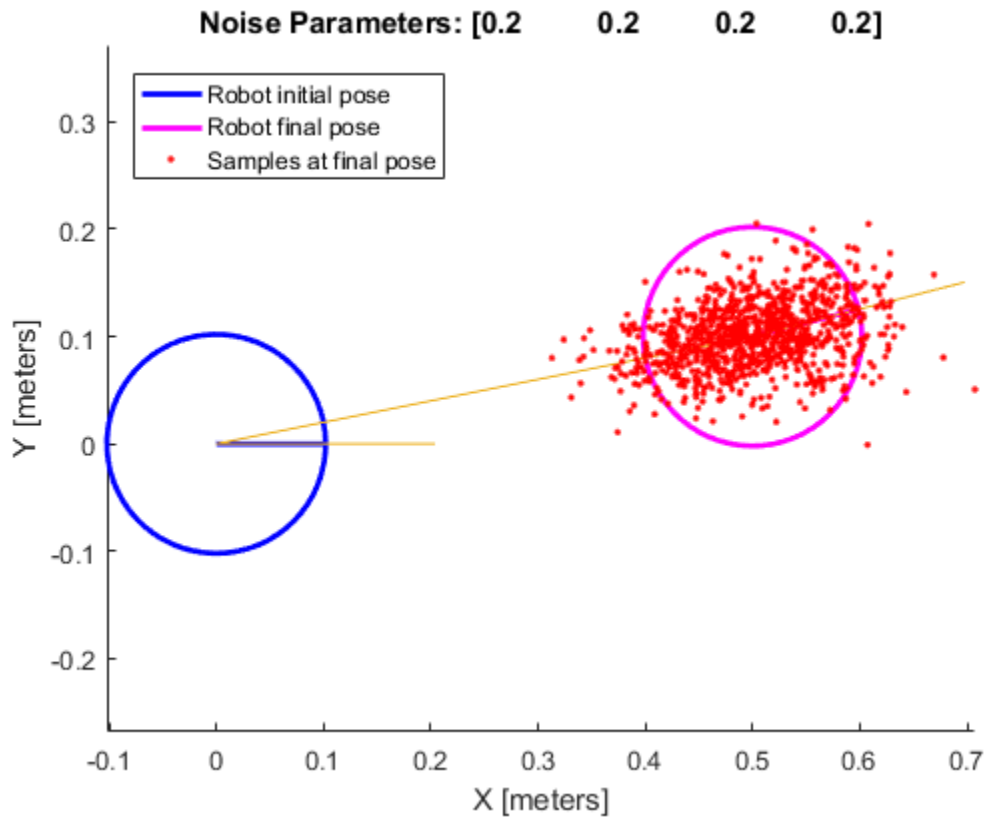
Show the distribution of particles with the existing noise parameters. Each particle is a hypothesis for the predicted pose.

```
showNoiseDistribution(motionModel);
```



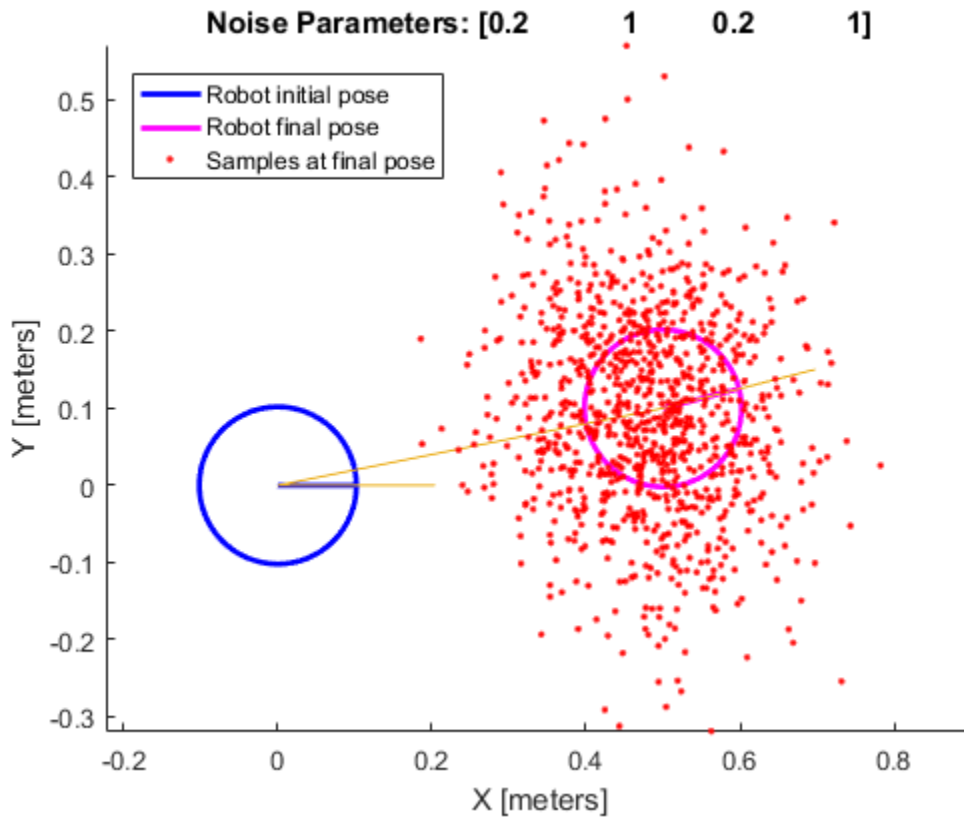
Show the distribution with a specified odometry pose change and number of samples. The change in odometry is used as the final pose with hypotheses distributed around based on the **NOISE** parameters.

```
showNoiseDistribution(motionModel, ...
    'OdometryPoseChange', [0.5 0.1 0.25], ...
    'NumSamples', 1000);
```

Change the **Noise** parameters and visualize the effects. Use the same odometry pose change and number of samples.

```
motionModel.Noise = [0.2 1 0.2 1];
showNoiseDistribution(motionModel, ...
    'OdometryPoseChange', [0.5 0.1 0.25], ...
    'NumSamples', 1000);
```



See Also

[robotics.MonteCarloLocalization](#) | [robotics.OdometryMotionModel](#) | [robotics.LikelihoodFieldSensorModel](#)

Introduced in R2016b

step

Class: robotics.OdometryMotionModel

Package: robotics

Computer next pose from previous pose

Syntax

```
currentPoses = step(ommObj,previousPoses,odomPose)
```

Description

`currentPoses = step(ommObj,previousPoses,odomPose)` returns the current poses by propagating the previous poses using a sampling-based odometry motion model, which uses the difference between the specified `odomPose` and the `LastOdometryPose` property of the `ommObj`. The first `step` call instantiates the object and sets the `LastOdometryPose` property.

Input Arguments

ommObj — OdometryMotionModel object

handle

OdometryMotionModel object, specified as a handle. Create this object using `robotics.OdometryMotionModel`.

previousPoses — Previous poses

n-by-3 array

Previous poses, specified as an *n*-by-3 array, [`x` `y` `theta`]. Each row of the `previousPoses` vector is treated as a separate robot and a corresponding predicted pose is present in `currentPoses`

odomPose — Current robot pose

three-element vector

Current robot pose, specified as a three-element vector, [x y theta].

Output Arguments

currentPoses — Current poses

n-by-3 array

Current poses, returned as an *n*-by-3 array, [x y theta]. Each row of the `previousPoses` vector is treated as a separate robot and a corresponding predicted pose is present in `currentPoses`.

Examples

Predict Poses Based On An Odometry Motion Model

This example shows how to use the `robotics.OdometryMotionModel` class to predict the pose of a robot. An `OdometryMotionModel` object contains the motion model parameters for a differential drive robot. Use the object to predict the pose of a robot based on its current and previous poses and the motion model parameters.

Create odometry motion 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);
```

See Also

[robotics.MonteCarloLocalization](#) | [robotics.OdometryMotionModel](#) | [robotics.LikelihoodFieldSensorModel](#)

Introduced in R2016b

copy

Class: robotics.ParticleFilter

Package: robotics

Create copy of particle filter

Syntax

```
b = copy(a)
```

Description

`b = copy(a)` copies each element in the array of handles, `a`, to the new array of handles, `b`.

The `copy` method does not copy dependent properties. MATLAB does not call `copy` recursively on any handles contained in property values. MATLAB also does not call the class constructor or property-set methods during the copy operation.

Input Arguments

a — Object array

handle

Object array, specified as a handle.

Output Arguments

b — Object array containing copies of the objects in `a`

handle

Object array containing copies of the object in `a`, specified as a handle.

`b` has the same number of elements and is the same size and class of `a`. `b` is the same class as `a`. If `a` is empty, `b` is also empty. If `a` is heterogeneous, `b` is also heterogeneous.

If **a** contains deleted handles, then **copy** creates deleted handles of the same class in **b**. Dynamic properties and listeners associated with objects in **a** are not copied to objects in **b**.

See Also

robotics.ParticleFilter

More About

- “Particle Filter Parameters”
- “Particle Filter Workflow”

Introduced in R2016a

correct

Class: robotics.ParticleFilter

Package: robotics

Adjust state estimate based on sensor measurement

Syntax

```
[stateCorr, stateCov] = correct(pf, measurement)
[stateCorr, stateCov] = correct(pf, measurement, varargin)
```

Description

`[stateCorr, stateCov] = correct(pf, measurement)` calculates the corrected system state and its associated uncertainty covariance based on a sensor `measurement` at the current time step. `correct` uses the `MeasurementLikelihoodFcn` property from the particle filter object, `pf`, as a function to calculate the likelihood of the sensor measurement for each particle. The two inputs to the `MeasurementLikelihoodFcn` function are:

- 1 `pf` – The `ParticleFilter` object, which contains the particles of the current iteration
- 2 `measurement` – The sensor measurements used to correct the state estimate

The `MeasurementLikelihoodFcn` function then extracts the best state estimate and covariance based on the setting in the `StateEstimationMethod` property.

`[stateCorr, stateCov] = correct(pf, measurement, varargin)` passes all additional arguments in `varargin` to the underlying `MeasurementLikelihoodFcn` after the first three required inputs.

Input Arguments

pf – **ParticleFilter** object
handle

ParticleFilter object, specified as a handle. See `robotics.ParticleFilter` for more information.

measurement — Sensor measurements

array

Sensor measurements, specified as an array. This input is passed directly into the `MeasurementLikelihoodFcn` property of `pf`. It is used to calculate the likelihood of the sensor measurement for each particle.

varargin — Variable-length input argument list

comma-separated list

Variable-length input argument list, specified as a comma-separated list. This input is passed directly into the `MeasurementLikelihoodFcn` property of `pf`. It is used to calculate the likelihood of the sensor measurement for each particle. When you call:

```
correct(pf, measurement, arg1, arg2)
```

MATLAB essentially calls `measurementLikelihoodFcn` as:

```
measurementLikelihoodFcn(pf, measurement, arg1, arg2)
```

Output Arguments

stateCorr — Corrected system state

vector with length `NumStateVariables`

Corrected system state, returned as a row vector with length `NumStateVariables`. The corrected state is calculated based on the `StateEstimationMethod` algorithm and the `MeasurementLikelihoodFcn`.

stateCov — Corrected system covariance

N -by- N matrix | []

Corrected system variance, returned as an N -by- N matrix, where N is the value of `NumStateVariables` property from `pf`. The corrected state is calculated based on the `StateEstimationMethod` algorithm and the `MeasurementLikelihoodFcn`. If you specify a state estimate method that does not support covariance, then the function returns `stateCov` as [].

Examples

Particle Filter Prediction and Correction

Create a `ParticleFilter` object, and execute a prediction and correction step for state estimation. The particle filter gives a predicted state estimate based on the return value of `StateTransitionFcn`. It then corrects the state based on a given measurement and the return value of `MeasurementLikelihoodFcn`.

Create a particle filter with the default three states.

```
pf = robotics.ParticleFilter
```

```
pf =
```

```
ParticleFilter with properties:
```

```
    NumStateVariables: 3
    NumParticles: 1000
    StateTransitionFcn: @robotics.algs.gaussianMotion
    MeasurementLikelihoodFcn: @robotics.algs.fullStateMeasurement
    IsStateVariableCircular: [0 0 0]
    ResamplingPolicy: [1×1 robotics.ResamplingPolicy]
    ResamplingMethod: 'multinomial'
    StateEstimationMethod: 'mean'
    Particles: [1000×3 double]
    Weights: [1000×1 double]
```

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
```

- “Track a Car-Like Robot using Particle Filter”

See Also

| | |

More About

- “Particle Filter Parameters”
- “Particle Filter Workflow”

Introduced in R2016a

getStateEstimate

Class: robotics.ParticleFilter

Package: robotics

Extract best state estimate and covariance from particles

Syntax

```
stateEst = getStateEstimate(pf)
[stateEst, stateCov] = getStateEstimate(pf)
```

Description

`stateEst = getStateEstimate(pf)` returns the best state estimate based on the current set of particles. The estimate is extracted based on the `StateEstimationMethod` property from the `ParticleFilter` object, `pf`.

`[stateEst, stateCov] = getStateEstimate(pf)` also returns the covariance around the state estimate. The covariance is a measure of the uncertainty of the state estimate. Not all state estimate methods support covariance output. In this case, `getStateEstimate` returns `stateCov` as `[]`.

Input Arguments

pf — **ParticleFilter** object
handle

`ParticleFilter` object, specified as a handle. See `robotics.ParticleFilter` for more information.

Output Arguments

stateEst — **Best state estimate**
vector

Best state estimate, returned as a row vector with length `NumStateVariables`. The estimate is extracted based on the `StateEstimationMethod` algorithm specified in `pf`.

stateCov — Corrected system covariance

N-by-*N* matrix | []

Corrected system variance, returned as an *N*-by-*N* matrix, where *N* is the value of `NumStateVariables` property from `pf`. The corrected state is calculated based on the `StateEstimationMethod` algorithm and the `MeasurementLikelihoodFcn`. If you specify a state estimate method that does not support covariance, then the function returns `stateCov` as [].

Examples

Particle Filter Prediction and Correction

Create a `ParticleFilter` object, and execute a prediction and correction step for state estimation. The particle filter gives a predicted state estimate based on the return value of `StateTransitionFcn`. It then corrects the state based on a given measurement and the return value of `MeasurementLikelihoodFcn`.

Create a particle filter with the default three states.

```
pf = robotics.ParticleFilter
```

```
pf =
```

```
ParticleFilter with properties:
```

```

    NumStateVariables: 3
      NumParticles: 1000
    StateTransitionFcn: @robotics.algs.gaussianMotion
    MeasurementLikelihoodFcn: @robotics.algs.fullStateMeasurement
    IsStateVariableCircular: [0 0 0]
      ResamplingPolicy: [1×1 robotics.ResamplingPolicy]
      ResamplingMethod: 'multinomial'
    StateEstimationMethod: 'mean'
      Particles: [1000×3 double]
      Weights: [1000×1 double]
```

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
```

- “Track a Car-Like Robot using Particle Filter”

See Also

| | |

More About

- “Particle Filter Parameters”
- “Particle Filter Workflow”

Introduced in R2016a

initialize

Class: robotics.ParticleFilter

Package: robotics

Initialize the state of the particle filter

Syntax

```
initialize(pf,numParticles,mean,covariance)
initialize(pf,numParticles,stateBounds)
initialize( ____,Name,Value)
```

Description

`initialize(pf,numParticles,mean,covariance)` initializes the particle filter object, `pf`, with a specified number of particles, `numParticles`. The initial states of the particles in the state space are determined by sampling from the multivariate normal distribution with the specified `mean` and `covariance`.

`initialize(pf,numParticles,stateBounds)` determines the initial location of the particles by sample from the multivariate uniform distribution within the specified `stateBounds`.

`initialize(____,Name,Value)` initializes the particles with additional options specified by one or more `Name,Value` pair arguments.

Input Arguments

pf — ParticleFilter object

handle

ParticleFilter object, specified as a handle. See `robotics.ParticleFilter` for more information.

numParticles — Number of particles used in the filter

scalar

Number of particles used in the filter, specified as a scalar.

mean — Mean of particle distribution

vector

Mean of particle distribution, specified as a vector. The `NumStateVariables` property of `pf` is set based on the length of this vector.

covariance — Covariance of particle distribution

N-by-*N* matrix

Covariance of particle distribution, specified as an *N*-by-*N* matrix, where *N* is the value of `NumStateVariables` property from `pf`.

stateBounds — Bounds of state variables

n-by-2 matrix

Bounds of state variables, specified as an *n*-by-2 matrix. The `NumStateVariables` property of `pf` is set based on the value of *n*. Each row corresponds to the lower and upper limit of the corresponding state variable.

Name-Value Pair Arguments

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

'CircularVariables' — Circular variables

logical vector

Circular variables, specified as a logical vector. Each state variable that uses circular or angular coordinates is indicated with a 1. The length of the vector is equal to the `NumStateVariables` property of `pf`.

Examples

Particle Filter Prediction and Correction

Create a `ParticleFilter` object, and execute a prediction and correction step for state estimation. The particle filter gives a predicted state estimate based on the return value

of `StateTransitionFcn`. It then corrects the state based on a given measurement and the return value of `MeasurementLikelihoodFcn`.

Create a particle filter with the default three states.

```
pf = robotics.ParticleFilter
```

```
pf =
```

```
ParticleFilter with properties:
```

```

    NumStateVariables: 3
    NumParticles: 1000
    StateTransitionFcn: @robotics.algs.gaussianMotion
    MeasurementLikelihoodFcn: @robotics.algs.fullStateMeasurement
    IsStateVariableCircular: [0 0 0]
    ResamplingPolicy: [1×1 robotics.ResamplingPolicy]
    ResamplingMethod: 'multinomial'
    StateEstimationMethod: 'mean'
    Particles: [1000×3 double]
    Weights: [1000×1 double]
```

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

- “Track a Car-Like Robot using Particle Filter”

See Also

| | |

More About

- “Particle Filter Parameters”
- “Particle Filter Workflow”

Introduced in R2016a

predict

Class: robotics.ParticleFilter

Package: robotics

Predict state of robot in next time step

Syntax

```
[statePred, stateCov] = predict(pf)
[statePred, stateCov] = predict(pf, varargin)
```

Description

`[statePred, stateCov] = predict(pf)` calculates the predicted system state and its associated uncertainty covariance. `predict` uses the `StateTransitionFcn` property of `ParticleFilter` object, `pf`, to evolve the state of all particles. It then extracts the best state estimate and covariance based on the setting in the `StateEstimationMethod` property.

`[statePred, stateCov] = predict(pf, varargin)` passes all additional arguments specified in `varargin` to the underlying `StateTransitionFcn` property of `pf`. The first input to `StateTransitionFcn` is the set of particles from the previous time step, followed by all arguments in `varargin`.

Input Arguments

pf — **ParticleFilter** object

handle

`ParticleFilter` object, specified as a handle. See `robotics.ParticleFilter` for more information.

varargin — **Variable-length input argument list**

comma-separated list

Variable-length input argument list, specified as a comma-separated list. This input is passed directly into the `StateTransitionFcn` property of `pf` to evolve the system state for each particle. When you call:

```
predict(pf, arg1, arg2)
```

MATLAB essentially calls the `stateTransitionFcn` as:

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

Output Arguments

statePred — Predicted system state

vector

Predicted system state, returned as a vector with length `NumStateVariables`. The predicted state is calculated based on the `StateEstimationMethod` algorithm.

stateCov — Corrected system covariance

N -by- N matrix | []

Corrected system variance, returned as an N -by- N matrix, where N is the value of `NumStateVariables` property from `pf`. The corrected state is calculated based on the `StateEstimationMethod` algorithm and the `MeasurementLikelihoodFcn`. If you specify a state estimate method that does not support covariance, then the function returns `stateCov` as [].

Examples

Particle Filter Prediction and Correction

Create a `ParticleFilter` object, and execute a prediction and correction step for state estimation. The particle filter gives a predicted state estimate based on the return value of `StateTransitionFcn`. It then corrects the state based on a given measurement and the return value of `MeasurementLikelihoodFcn`.

Create a particle filter with the default three states.

```
pf = robotics.ParticleFilter
```

```
pf =
```

ParticleFilter with properties:

```

    NumStateVariables: 3
      NumParticles: 1000
    StateTransitionFcn: @robotics.algs.gaussianMotion
    MeasurementLikelihoodFcn: @robotics.algs.fullStateMeasurement
    IsStateVariableCircular: [0 0 0]
      ResamplingPolicy: [1×1 robotics.ResamplingPolicy]
      ResamplingMethod: 'multinomial'
    StateEstimationMethod: 'mean'
      Particles: [1000×3 double]
      Weights: [1000×1 double]

```

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
```

- “Track a Car-Like Robot using Particle Filter”

See Also

`robotics.ParticleFilter` | `robotics.ParticleFilter.initialize` | `robotics.ParticleFilter.correct`
| `robotics.ParticleFilter.getStateEstimate`

More About

- “Particle Filter Parameters”
- “Particle Filter Workflow”

Introduced in R2016a

findpath

Class: robotics.PRM

Package: robotics

Find path between start and goal points on roadmap

Syntax

```
xy = findpath(prm,start,goal)
```

Description

`xy = findpath(prm,start,goal)` finds an obstacle-free path between `start` and `goal` locations within `prm`, a roadmap object that contains a network of connected points.

If any properties of `prm` change, or if the roadmap is not created, `update` is called.

Input Arguments

prm — Roadmap path planner

PRM object

Roadmap path planner, specified as a `robotics.PRM` object.

start — Start location of path

2-by-1 vector

Start location of path, specified as a 2-by-1 vector representing an `[x y]` pair.

Example: `[0 0]`

goal — Final location of path

2-by-1 vector

Final location of path, specified as a 2-by-1 vector representing an `[x y]` pair.

Example: `[10 10]`

Output Arguments

xy — Waypoints for a path between start and goal

2-by- n column vector

Waypoints for a path between start and goal, specified as a 2-by- n column vector of [x y] pairs, where n is the number of waypoints. These pairs represent the solved path from the start and goal locations, given the roadmap from the `prm` input object.

See Also

`robotics.PRM` | `robotics.PRM.show` | `robotics.PRM.update`

Introduced in R2015a

show

Class: robotics.PRM

Package: robotics

Show map, roadmap, and path

Syntax

show(prm)

show(prm, Name, Value)

Description

show(prm) shows the map and the roadmap, specified as prm in a figure window. If no roadmap exists, update is called. If a path is computed before calling show, the path is also plotted on the figure.

show(prm, Name, Value) sets the specified Value to the property Name.

Input Arguments

prm — Roadmap path planner

PRM object

Roadmap path planner, specified as a robotics.PRM object.

Name-Value Pair Arguments

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

'Parent' — Axes handle

handle

Axes handle that specifies the parent of the figure object created by show, specified as the comma-separated pair consisting of 'Parent' and a handle.

'Map' — Map display option

'on' (default) | 'off'

Map display option, specified as the comma-separated pair consisting of 'Map' and either 'on' or 'off'.

'Roadmap' — Roadmap display option

'on' (default) | 'off'

Roadmap display option, specified as the comma-separated pair consisting of 'Roadmap' and either 'on' or 'off'.

'Path' — Path display option

'on' (default) | 'off'

Path display option, specified as 'on' or 'off'. This controls whether the computed path is shown in the plot.

See Also

| |

Related Examples

- “Path Following for a Differential Drive Robot”

Introduced in R2015a

update

Class: robotics.PRM

Package: robotics

Create or update roadmap

Syntax

```
update(prm)
```

Description

`update(prm)` creates a roadmap if called for the first time after creating the PRM object, `prm`. Subsequent calls of `update` recreate the roadmap by resampling the map. `update` creates the new roadmap using the `Map`, `NumNodes`, and `ConnectionDistance` property values specified in `prm`.

Input Arguments

prm — Roadmap path planner

PRM object

Roadmap path planner, specified as a `robotics.PRM` object.

See Also

`robotics.PRM` | `robotics.PRM.findpath` | `robotics.PRM.show`

Introduced in R2015a

clone

Class: robotics.PurePursuit

Package: robotics

Create PurePursuit object with same property values

Syntax

```
copy = clone(controller)
```

Description

`copy = clone(controller)` creates another instance of the System object, `controller`, with the same property values. If the input object is locked, the `clone` method creates a copy that is also locked and has states initialized to the same values as the original. If the input object is not locked, the `clone` method creates a new unlocked object with uninitialized states.

Input Arguments

controller — Pure pursuit controller

PurePursuit object

Pure pursuit controller, specified as a PurePursuit object.

Output Arguments

copy — Pure pursuit controller

PurePursuit object

Copy of pure pursuit controller, returned as a PurePursuit object.

See Also

robotics.PurePursuit

Introduced in R2015a

info

Class: robotics.PurePursuit

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`.

Input Arguments

controller — Pure pursuit controller

`PurePursuit` object

Pure pursuit controller, specified as a `PurePursuit` object.

Output Arguments

controllerInfo — Information on the `PurePursuit` object

structure

Information on the `PurePursuit` object, returned as a structure. The structure contains two fields:

- `RobotPose` — A three-element vector in the form `[x y theta]` that corresponds to the x - y position and orientation of the robot. The angle, `theta`, is measured in radians with positive angles measured counterclockwise from the x -axis.

- **LookaheadPoint**– A two-element vector in the form $[x \ y]$. The location is a point on the path that was used to compute outputs of the last call to `robotics.PurePursuit.step`.

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 `step` call.

Create a `PurePursuit` object.

```
pp = robotics.PurePursuit;
```

Assign waypoints.

```
pp.Waypoints = [0 0;1 1];
```

Compute control commands using the `step` method.

```
[v, w] = step(pp, [0 0 0]);
```

Get additional information.

```
s = info(pp)
```

```
s =
```

```
struct with fields:
```

```
    RobotPose: [0 0 0]
```

```
    LookaheadPoint: [0.7071 0.7071]
```

See Also

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

More About

- “Pure Pursuit Controller”

Introduced in R2015a

isLocked

Class: robotics.PurePursuit

Package: robotics

Check locked states (logical)

Syntax

```
status = isLocked(controller)
```

Description

`status = isLocked(controller)` returns a logical value, `status`, which indicates whether input attributes and nontunable properties are locked for the object, `controller`. For the `PurePursuit` class, the only nontunable property is `Waypoints`.

Input Arguments

controller — Pure pursuit controller

PurePursuit object

Pure pursuit controller, specified as a `PurePursuit` object.

Output Arguments

status — Locked status of object

Boolean

Locked status of the object input attributes and nontunable properties, returned as a Boolean.

See Also

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

Introduced in R2015a

release

Class: robotics.PurePursuit

Package: robotics

Allow property value changes

Syntax

```
release(controller)
```

Description

`release(controller)` resets the internal properties of the `controller` object and unlocks the object so that you can modify nontunable properties. For the `PurePursuit` class, the only nontunable property is `Waypoints`. After `release` is called, you can change the properties and input characteristics of `controller`.

Input Arguments

controller — Pure pursuit controller

PurePursuit object

Pure pursuit controller, specified as a PurePursuit object.

See Also

robotics.PurePursuit | robotics.PurePursuit.isLocked | robotics.PurePursuit.step

Introduced in R2015a

reset

Class: robotics.PurePursuit

Package: robotics

Reset internal states to default

Syntax

```
reset(controller)
```

Description

`reset(controller)` resets the internal system properties of the `controller` object. All properties specific to the `PurePursuit` object are kept the same and the locked status of the object does not change.

Input Arguments

controller — Pure pursuit controller

PurePursuit object

Pure pursuit controller, specified as a PurePursuit object.

See Also

robotics.PurePursuit | robotics.PurePursuit.release

Introduced in R2015a

step

Class: robotics.PurePursuit

Package: robotics

Compute linear and angular velocity control commands

Syntax

```
[vel,angvel] = step(controller,pose)
[vel,angvel,lookaheadpoint] = step(controller,pose)
```

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.

`[vel,angvel] = step(controller,pose)` processes the robot's position and orientation, `pose`, as `[x y theta]`, and outputs the linear velocity, `vel`, and angular velocity, `angvel`, based on the specified `controller`.

`[vel,angvel,lookaheadpoint] = step(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

controller — Pure pursuit controller

PurePursuit object

Pure pursuit controller, specified as a PurePursuit object.

pose — Position and orientation of robot

3-by-1 vector in the form [x y theta]

Position and orientation of robot, specified as a 3-by-1 vector in the form [x y theta]. The robot's pose is an x and y position with angular orientation (in radians) measured from the x -axis.

Output Arguments

ve1 — Linear velocity

scalar in meters per second

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

Data Types: double

angve1 — Angular velocity

scalar in radians per second

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

Data Types: double

lookaheadpoint — Look-ahead point on path

[x y] vector

Look-ahead point on the path, returned as an [x y] vector. This value is calculated based on the LookaheadDistance property on the controller object.

See Also

robotics.PurePursuit

Introduced in R2015a

reset

Class: robotics.Rate

Package: robotics

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 Object

Create a `Rate` object running at 20 Hz.

```
r = robotics.Rate(20);
```

Start a loop using the `Rate` object.

```
for i = 1:30
```

```
% Your code goes here
waitfor(r);
end
```

Display the `Rate` object properties.

```
disp(r)

Rate with properties:

    DesiredRate: 20
   DesiredPeriod: 0.0500
  OverrunAction: 'slip'
TotalElapsedTime: 26.7246
    LastPeriod: 0.0500
```

Reset the loop to restart the time.

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

Rate with properties:

    DesiredRate: 20
   DesiredPeriod: 0.0500
  OverflowMethod: 'drop'
TotalElapsedTime: 0.0012
    LastPeriod: NaN
```

- “Execute Code at a Fixed-Rate”

See Also

[robotics.Rate](#) | [robotics.Rate.waitfor](#) | [robotics.Rate.statistics](#)

Introduced in R2016a

statistics

Class: robotics.Rate

Package: robotics

Statistics of past execution periods

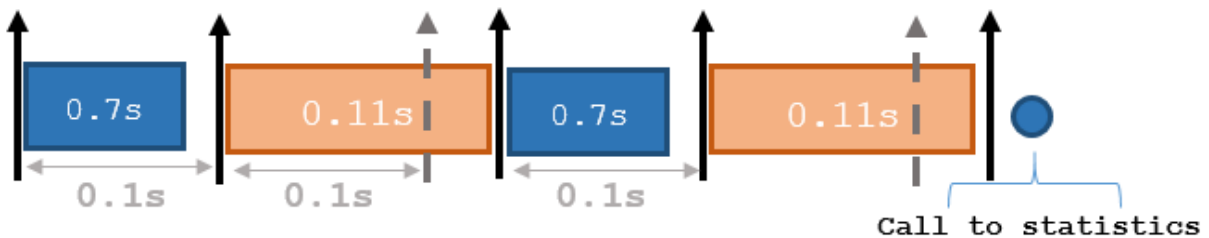
Syntax

```
stats = statistics(rate)
```

Description

`stats = statistics(rate)` returns statistics of previous periods of code execution. `stats` is a struct with these fields: `Periods`, `NumPeriods`, `AveragePeriod`, `StandardDeviation`, and `NumOverruns`.

Here is a sample execution graphic using the default setting, 'slip', for the `OverrunAction` property in the `Rate` object. See `OverrunAction` for more information on overrun code execution.



The output of `statistics` is:

```
stats =
    Periods: [0.7 0.11 0.7 0.11]
    NumPeriods: 4
    AveragePeriod: 0.09
```

StandardDeviation: 0.0231
NumOverruns: 2

Input Arguments

rate — Rate object

handle

Rate object, specified as an object handle. This object contains the information for the `DesiredRate` and other info about the execution. See `robotics.Rate` for more information.

Output Arguments

stats — Time execution statistics

structure

Time execution statistics, returned as a structure. This structure contains the following fields:

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

Examples

Get Statistics from Rate Object

```
stats = statistics(rate)
```

```
stats =
```

```
    Periods: [1x100 double]
  NumPeriods: 100
  AveragePeriod: 0.3019
  StandardDeviation: 2.5197
  NumOverruns: 0
```

- “Execute Code at a Fixed-Rate”

See Also

[robotics.Rate](#) | [robotics.Rate.waitFor](#) | [robotics.Rate.reset](#) | [ROS Rate](#)

Introduced in R2016a

waitfor

Class: robotics.Rate

Package: robotics

Pause code execution to achieve desired execution rate

Syntax

```
waitfor(rate)
numMisses = waitfor(rate)
```

Description

`numMisses = 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.

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

numMissed — 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.

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. Print the iteration and the time elapsed.

```
for i = 1:10;
    time = r.TotalElapsedTime;
    fprintf('Iteration: %d - Time Elapsed: %f\n',i,time)
    waitfor(r);
end
```

```
Iteration: 1 - Time Elapsed: 5.344408
Iteration: 2 - Time Elapsed: 5.345697
Iteration: 3 - Time Elapsed: 6.345708
Iteration: 4 - Time Elapsed: 7.345741
Iteration: 5 - Time Elapsed: 8.345797
Iteration: 6 - Time Elapsed: 9.345820
Iteration: 7 - Time Elapsed: 10.345850
Iteration: 8 - Time Elapsed: 11.344907
Iteration: 9 - Time Elapsed: 12.344948
Iteration: 10 - Time Elapsed: 13.344967
```

Each iteration executes at a 1-second interval.

- “Execute Code at a Fixed-Rate”

See Also

`robotics.Rate` | `robotics.Rate.statistics` | `robotics.Rate.reset` | ROS Rate

Introduced in R2016a

copy

Class: robotics.RigidBody

Package: robotics

Create a deep copy of rigid body

Syntax

```
copyObj = copy(bodyObj)
```

Description

`copyObj = copy(bodyObj)` creates a copy of the rigid body object with the same properties.

Input Arguments

bodyObj — **RigidBody** object

handle

RigidBody object, specified as a handle. Create a rigid body object using `robotics.RigidBody`.

Output Arguments

copyObj — **RigidBody** object

handle

RigidBody object, returned as a handle. Create a rigid body object using `robotics.RigidBody`.

See Also

`robotics.Joint` | `robotics.RigidBodyTree`

Introduced in R2016b

addBody

Class: robotics.RigidBodyTree

Package: robotics

Add a body to robot

Syntax

```
addBody( robot , body , parentname )
```

Description

`addBody(robot , body , parentname)` adds a rigid body to the robot object and is attached to the rigid body parent specified by `parentname`. The `body.Joint` property defines how this body moves relative to the parent body.

Input Arguments

robot — Robot model

RigidBodyTree object

Robot model, specified as a RigidBodyTree object.

body — Rigid body

RigidBody object

Rigid body, specified as a RigidBody object.

parentname — Parent body name

character vector

Parent body name, specified as a character vector. This parent body must already exist in the robot model. The new body is attached to this parent body.

Examples

Attach Rigid Body and Joint to Rigid Body Tree

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

Create a rigid body tree.

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

Create a rigid body with a unique name.

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

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

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

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

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

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

```
showdetails(rbtree)
```

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

  Idx   Body Name   Joint Name   Joint Type   Parent Name(Idx)   Children Name(s)
  ---   -
    1      b1         jnt1        revolute        base(0)
-----
```

Build Manipulator Robot Using Denavit-Hartenberg Parameters

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

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

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

Create a rigid body tree object to build the robot.

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

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

- 1 Create a `RigidBody` object and give it a unique name.
- 2 Create a `Joint` object and give it a unique name.
- 3 Use `setFixedTransform` to specify the body-to-body transformation using DH parameters.
- 4 Call `addBody` to attach the first body joint to the base frame of the robot.

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

setFixedTransform(jnt1, dhparams(1,:), 'dh');
body1.Joint = jnt1;

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

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

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

```

jnt5 = robotics.Joint('jnt5','revolute');
body6 = robotics.RigidBody('body6');
jnt6 = robotics.Joint('jnt6','revolute');

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

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

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

```

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

```
showdetails(robot)
```

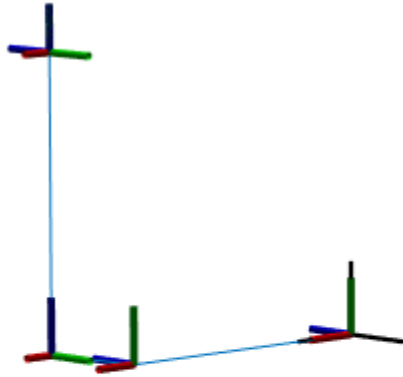
```
show(robot);
axis([-0.5,0.5,-0.5,0.5,-0.5,0.5])
axis off
```

```

-----
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]
    Parent: [1x1 robotics.RigidBody]
    Children: {[1x1 robotics.RigidBody]}
```

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'
```

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

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

```
showdetails(puma1)
```

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

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

6

L6

jnt6

revolute

L5(5)
-----**See Also**

[robotics.Joint](#) | [robotics.RigidBody](#) | [robotics.RigidBodyTree.removeBody](#) | [robotics.RigidBodyTree.replaceBody](#)

Introduced in R2016b

addSubtree

Class: robotics.RigidBodyTree

Package: robotics

Add subtree to robot

Syntax

```
addSubtree(robot, parentname, subtree)
```

Description

`addSubtree(robot, parentname, subtree)` attaches the robot model, `subtree`, to an existing robot model, `robot`, at the body specified by `parentname`. The subtree base is not added as a body.

Input Arguments

robot — Robot model

RigidBodyTree object

Robot model, specified as a RigidBodyTree object.

parentname — Parent body name

character vector

Parent body name, specified as a character vector. This parent body must already exist in the robot model. The new body is attached to this parent body.

subtree — Subtree robot model

RigidBodyTree object

Subtree robot model, specified as a RigidBodyTree object.

Examples

Modify a Robot Rigid Body Tree Model

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

Load example robots as `RigidBodyTree` objects.

```
load exampleRobots.mat
```

View the details of the Puma robot using `showdetails`.

```
showdetails(puma1)
```

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

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

```
-----
```

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

```
body3 = getBody(puma1, 'L3');
childBody = body3.Children{1}
body3Copy = copy(body3);
```

```
childBody =
```

```
RigidBody with properties:
```

```
    Name: 'L4'
    Joint: [1x1 robotics.Joint]
    Parent: [1x1 robotics.RigidBody]
    Children: {[1x1 robotics.RigidBody]}
```

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: {1×3 cell}
  Base: [1×1 robotics.RigidBody]
  BodyNames: {'L4' 'L5' 'L6'}
  BaseName: 'L3'
```

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

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

```
showdetails(puma1)
```

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

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

See Also

robotics.Joint | robotics.RigidBody | robotics.RigidBodyTree.addBody |
robotics.RigidBodyTree.removeBody | robotics.RigidBodyTree.replaceBody

Introduced in R2016b

copy

Class: robotics.RigidBodyTree

Package: robotics

Copy robot model

Syntax

```
newrobot = copy(robot)
```

Description

`newrobot = copy(robot)` creates a deep copy of `robot` with the same properties. Any changes in `newrobot` are not reflected in `robot`.

Input Arguments

robot — Robot model

RigidBodyTree object

Robot model, specified as a RigidBodyTree object.

Output Arguments

newrobot — Robot model

RigidBodyTree object

Robot model, returned as a RigidBodyTree object.

Examples

Modify a Robot Rigid Body Tree Model

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

Load example robots as `RigidBodyTree` objects.

```
load exampleRobots.mat
```

View the details of the Puma robot using `showdetails`.

```
showdetails(puma1)
```

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

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

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

```
body3 = getBody(puma1, 'L3');
childBody = body3.Children{1}
body3Copy = copy(body3);
```

```
childBody =
```

```
RigidBody with properties:

    Name: 'L4'
   Joint: [1×1 robotics.Joint]
  Parent: [1×1 robotics.RigidBody]
 Children: {[1×1 robotics.RigidBody]}
```

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: {1×3 cell}  
  Base: [1×1 robotics.RigidBody]  
  BodyNames: {'L4' 'L5' 'L6'}  
  BaseName: 'L3'
```

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

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

```
showdetails(puma1)
```

Robot: (6 bodies)

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

See Also

[robotics.Joint](#) | [robotics.RigidBody](#) | [robotics.RigidBodyTree](#)

Introduced in R2016b

geometricJacobian

Class: robotics.RigidBodyTree

Package: robotics

Geometric Jacobian for robot configuration

Syntax

```
jacobian = geometricJacobian(robot,configuration,endeffectorname)
```

Description

```
jacobian = geometricJacobian(robot,configuration,endeffectorname)
```

computes the geometric Jacobian relative to the base for the specified end effector name and configuration for the robot model.

Input Arguments

robot — Robot model

RigidBodyTree object

Robot model, specified as a RigidBodyTree object.

configuration — Robot configuration

structure

Robot configuration, specified as a structure with joint names and positions for all the bodies in the robot model. You can generate a configuration using `homeConfiguration(robot)`, `randomConfiguration(robot)`, or by specifying your own joint positions in a structure.

endeffectorname — End effector name

character vector

End effector name, specified as a character vector. An end effector can be any body in the robot model.

Output Arguments

jacobian — Geometric Jacobian

6-by- n matrix

Geometric Jacobian of the end effector with the given configuration, returned as a 6-by- n matrix, where n is the number of degrees of freedom for the end effector. The Jacobian maps the joint-space velocity to the end effector velocity, relative to the base coordinate frame. The end effector velocity equals:

$$V_{EE} = \begin{bmatrix} \omega_x \\ \omega_y \\ \omega_z \\ v_x \\ v_y \\ v_z \end{bmatrix} = Jq = J \begin{bmatrix} q_1 \\ \vdots \\ q_n \end{bmatrix}$$

ω is the angular velocity, v is the linear velocity, and $[q_1 \dots q_n]$ is the joint-space velocity.

Examples

Geometric Jacobian for Robot Configuration

Calculate the geometric Jacobian for a specific end effector and configuration of a robot.

Load example robots as `RigidBodyTree` objects.

```
load exampleRobots.mat
```

Calculate the geometric Jacobian of body 'L6' on the Puma robot for a random configuration.

```
geoJacobian = geometricJacobian(puma1,randomConfiguration(puma1),'L6')
```

```
geoJacobian =
```

```
    -0.0000    0.9826    0.9826    0.0286   -0.9155    0.2045
```

-0.0000	0.1859	0.1859	-0.1512	0.3929	0.2690
1.0000	-0.0000	-0.0000	0.9881	0.0866	0.9412
0.4175	0.0530	0.0799	0.0000	0	0
0.2317	-0.2802	-0.4223	0.0000	0	0
0	-0.4532	-0.0464	0.0000	0	0

See Also

[robotics.RigidBodyTree.getTransform](#) | [robotics.RigidBodyTree.homeConfiguration](#) | [robotics.RigidBodyTree.randomConfiguration](#) | [robotics.Joint](#) | [robotics.RigidBody](#)

Introduced in R2016b

getBody

Class: robotics.RigidBodyTree

Package: robotics

Get robot body handle by name

Syntax

```
body = getBody(robot, bodyname)
```

Description

`body = getBody(robot, bodyname)` gets a body handle by name from the robot model.

Input Arguments

robot — Robot model

RigidBodyTree object

Robot model, specified as a RigidBodyTree object.

bodyname — Body name

character vector

Body name, specified as a character vector. A body with this name must be on the robot model specified by `robot`.

Output Arguments

body — Rigid body

RigidBody object

Rigid body, returned as a RigidBody object. The returned RigidBody object is still a part of the RigidBodyTree robot model. Use `robotics.RigidBodyTree.replaceBody` with a new body to modify the body in the robot model.

Examples

Modify a Robot Rigid Body Tree Model

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

Load example robots as `RigidBodyTree` objects.

```
load exampleRobots.mat
```

View the details of the Puma robot using `showdetails`.

```
showdetails(puma1)
```

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

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

```
body3 = getBody(puma1, 'L3');  
childBody = body3.Children{1}  
body3Copy = copy(body3);
```

```
childBody =
```

```
  RigidBody with properties:
```

```
    Name: 'L4'  
    Joint: [1×1 robotics.Joint]  
    Parent: [1×1 robotics.RigidBody]  
    Children: {[1×1 robotics.RigidBody]}
```

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: {1×3 cell}
  Base: [1×1 robotics.RigidBody]
  BodyNames: {'L4' 'L5' 'L6'}
  BaseName: 'L3'
```

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

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

```
showdetails(puma1)
```

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

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

```
-----
```

See Also

[robotics.Joint](#) | [robotics.RigidBody](#) | [robotics.RigidBodyTree.addBody](#) | [robotics.RigidBodyTree.replaceBody](#)

Introduced in R2016b

getTransform

Class: robotics.RigidBodyTree

Package: robotics

Get transform between body frames

Syntax

```
transform = getTransform(robot,configuration,bodyname)
transform = getTransform(robot,configuration,targetbody,sourcebody)
```

Description

`transform = getTransform(robot,configuration,bodyname)` computes the transform that converts points in the `bodyname` frame to the robot base frame, using the specified robot configuration.

`transform = getTransform(robot,configuration,targetbody,sourcebody)` computes the transform that converts points from the source body frame to the target frame, using the specified robot configuration.

Input Arguments

robot — Robot model

RigidBodyTree object

Robot model, specified as a RigidBodyTree object.

configuration — Robot configuration

structure array

Robot configuration, specified as a structure array with joint names and positions for all the bodies in the robot model. You can generate a configuration using `homeConfiguration(robot)`, `randomConfiguration(robot)`, or by specifying your own joint names and positions in a structure array.

bodyname — Body name

character vector

Body name, specified as a character vector. This body must be on the robot model specified in `robot`.

targetbody — Target body name

character vector

Target body name, specified as a character vector. This body must be on the robot model specified in `robot`. The target frame is the coordinate system you want to transform points into.

sourcebody — Body name

character vector

Body name, specified as a character vector. This body must be on the robot model specified in `robot`. The source frame is the coordinate system you want points transformed from.

Output Arguments

transform — Homogenous transform

4-by-4 matrix

Homogeneous transform, returned as a 4-by-4 matrix.

Examples

Get Transform Between Frames for Robot Configuration

Get the transform between two frames for a specific robot configuration.

Load a sample robots that include the `puma1` robot.

```
load exampleRobots.mat
```

Get the transform between the 'L2' and 'L6' bodies of the `puma1` robot given a specific configuration. The transform converts points in 'L6' frame to the 'L2' frame.


```
transform = getTransform(puma1, randomConfiguration(puma1), 'L2', 'L6')
```

```
transform =
```

```
-0.2232    0.4179    0.8807    0.0212  
-0.8191    0.4094   -0.4018    0.1503  
-0.5284   -0.8111    0.2509   -0.4317  
         0         0         0         1.0000
```

See Also

[robotics.Joint](#) | [robotics.RigidBody](#) | [robotics.RigidBodyTree.geometricJacobian](#)
| [robotics.RigidBodyTree.homeConfiguration](#) |
[robotics.RigidBodyTree.randomConfiguration](#)

Introduced in R2016b

homeConfiguration

Class: robotics.RigidBodyTree

Package: robotics

Get home configuration of robot

Syntax

```
configuration = homeConfiguration(robot)
```

Description

`configuration = homeConfiguration(robot)` returns the home configuration of the robot model. The home configuration is the ordered list of `HomePosition` properties of each nonfixed joint.

Input Arguments

robot — Robot model

`RigidBodyTree` object

Robot model, specified as a `RigidBodyTree` object.

Output Arguments

configuration — Robot configuration

structure

Robot configuration, returned as a structure with joint names and positions for all the bodies in the robot model. Bodies with fixed joints are not included. You can generate a configuration using `homeConfiguration(robot)`, `randomConfiguration(robot)`, or by specifying your own joint positions in a structure.

Examples

Visualize Robot Configurations

Show different configurations of a robot created using a `RigidBodyTree` model. Use the `homeConfiguration` or `randomConfiguration` functions to generate the structure that defines all the joint positions.

Load example robots as `RigidBodyTree` objects.

```
load exampleRobots.mat
```

Create a structure for the home configuration of a Puma robot. The structure has joint names and positions for each body on the robot model.

```
config = homeConfiguration(puma1)
```

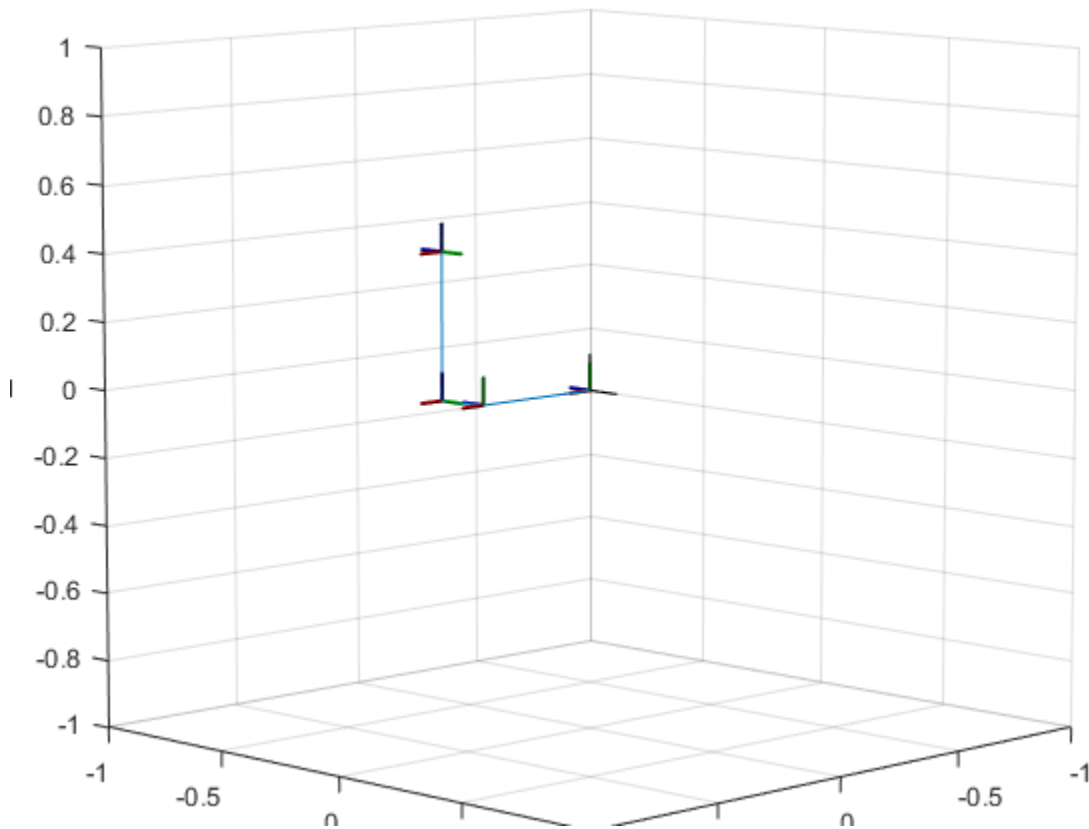
```
config =
```

```
1×6 struct array with fields:
```

```
JointName  
JointPosition
```

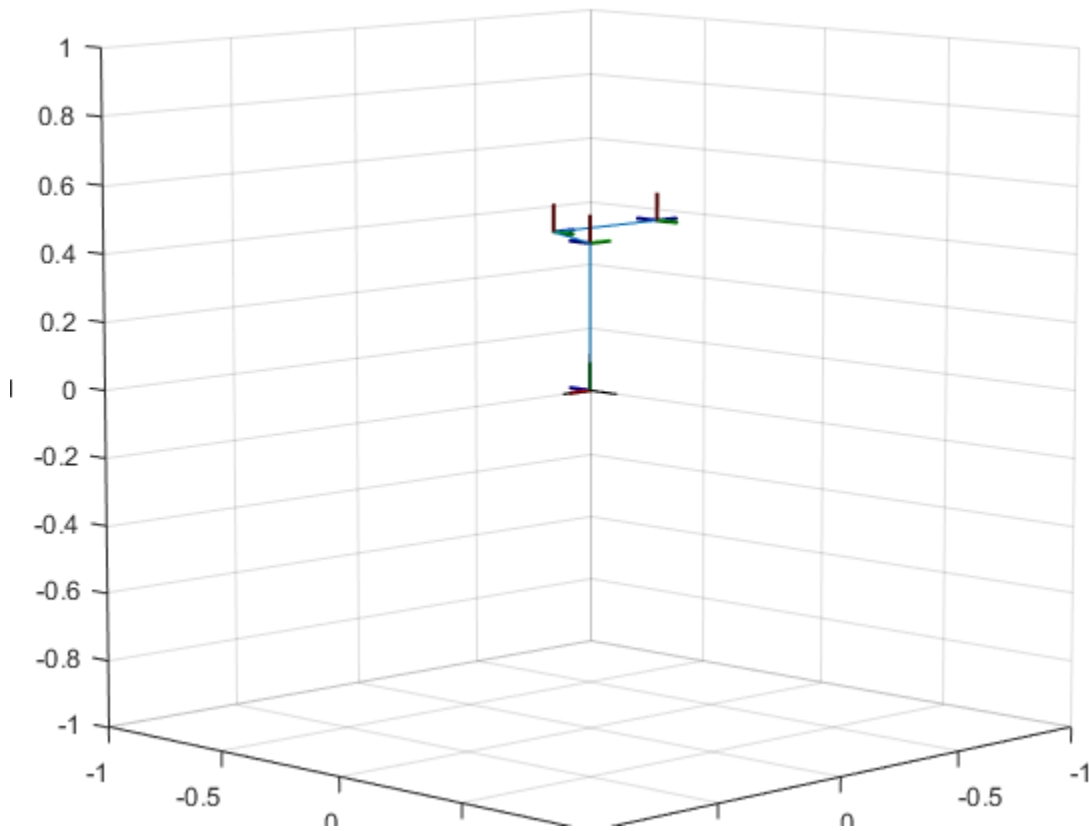
Show the home configuration using `show`. You do not need to specify a configuration input.

```
show(puma1);
```



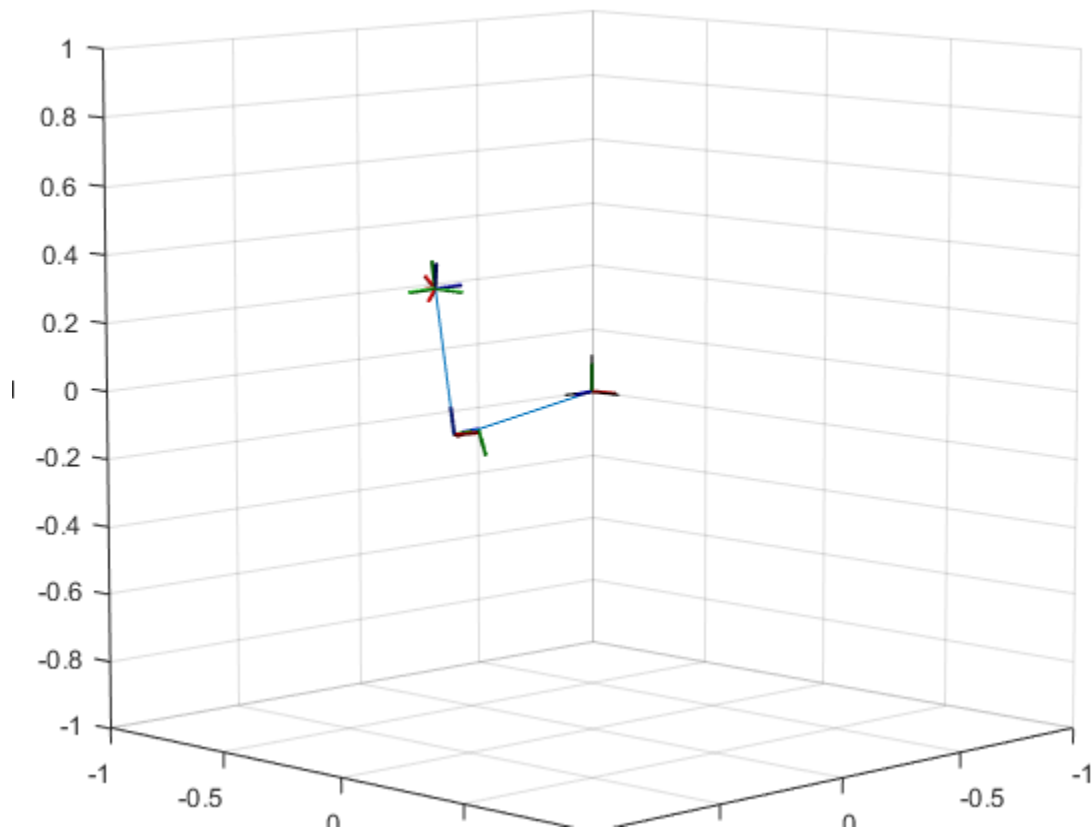
Modify the configuration and set the second joint position to $\pi/2$. Show the resulting change in the robot configuration.

```
config(2).JointPosition = pi/2;  
show(puma1,config);
```



Create random configurations and show them.

```
show(puma1,randomConfiguration(puma1));
```



See Also

[robotics.RigidBodyTree.randomConfiguration](#) | [robotics.RigidBodyTree.getTransform](#) | [robotics.RigidBodyTree.geometricJacobian](#)

Introduced in R2016b

randomConfiguration

Class: robotics.RigidBodyTree

Package: robotics

Generate random configuration of robot

Syntax

```
configuration = randomConfiguration(robot)
```

Description

`configuration = randomConfiguration(robot)` returns a random configuration of the specified robot. Each joint position in this configuration respects the joint limits set by the `PositionLimits` property of the corresponding `Joint` object in the robot model.

Input Arguments

robot — Robot model

RigidBodyTree object

Robot model, specified as a RigidBodyTree object.

Output Arguments

configuration — Robot configuration

structure

Robot configuration, returned as a structure with joint names and positions for all the bodies in the robot model. Bodies with fixed joints are not included. You can generate a configuration using `homeConfiguration(robot)`, `randomConfiguration(robot)`, or by specifying your own joint positions in a structure.

Examples

Visualize Robot Configurations

Show different configurations of a robot created using a `RigidBodyTree` model. Use the `homeConfiguration` or `randomConfiguration` functions to generate the structure that defines all the joint positions.

Load example robots as `RigidBodyTree` objects.

```
load exampleRobots.mat
```

Create a structure for the home configuration of a Puma robot. The structure has joint names and positions for each body on the robot model.

```
config = homeConfiguration(puma1)
```

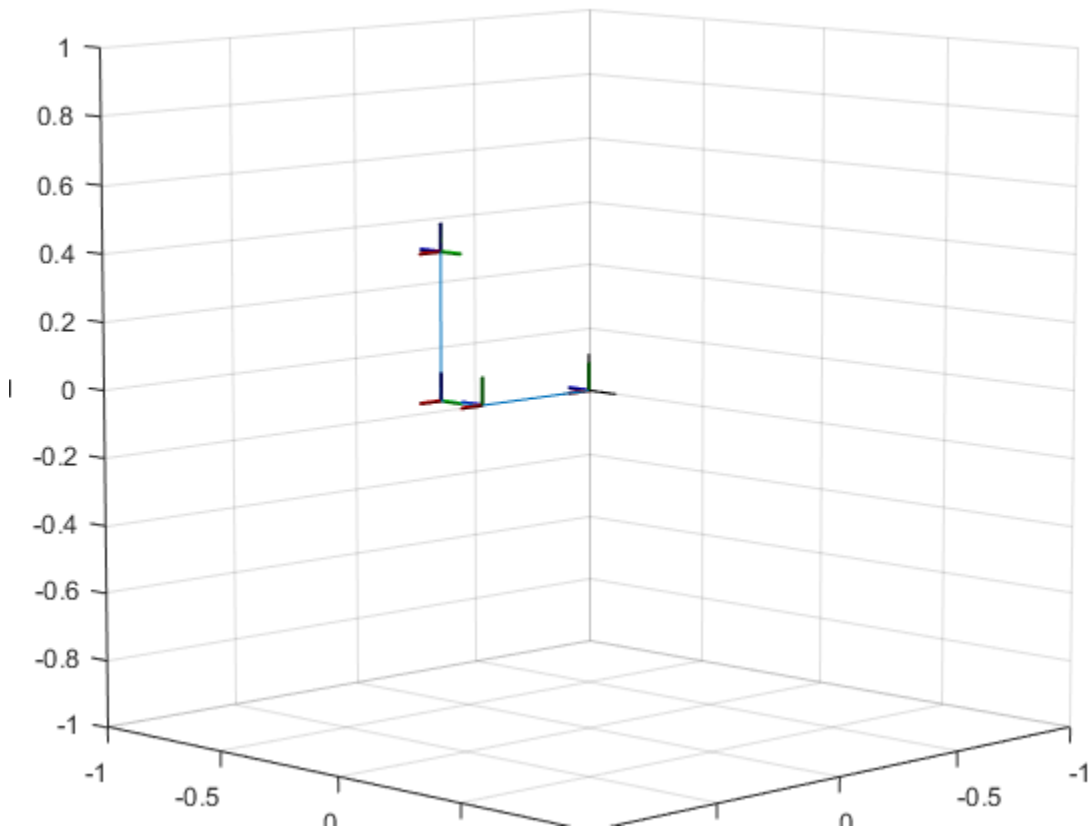
```
config =
```

```
1×6 struct array with fields:
```

```
JointName  
JointPosition
```

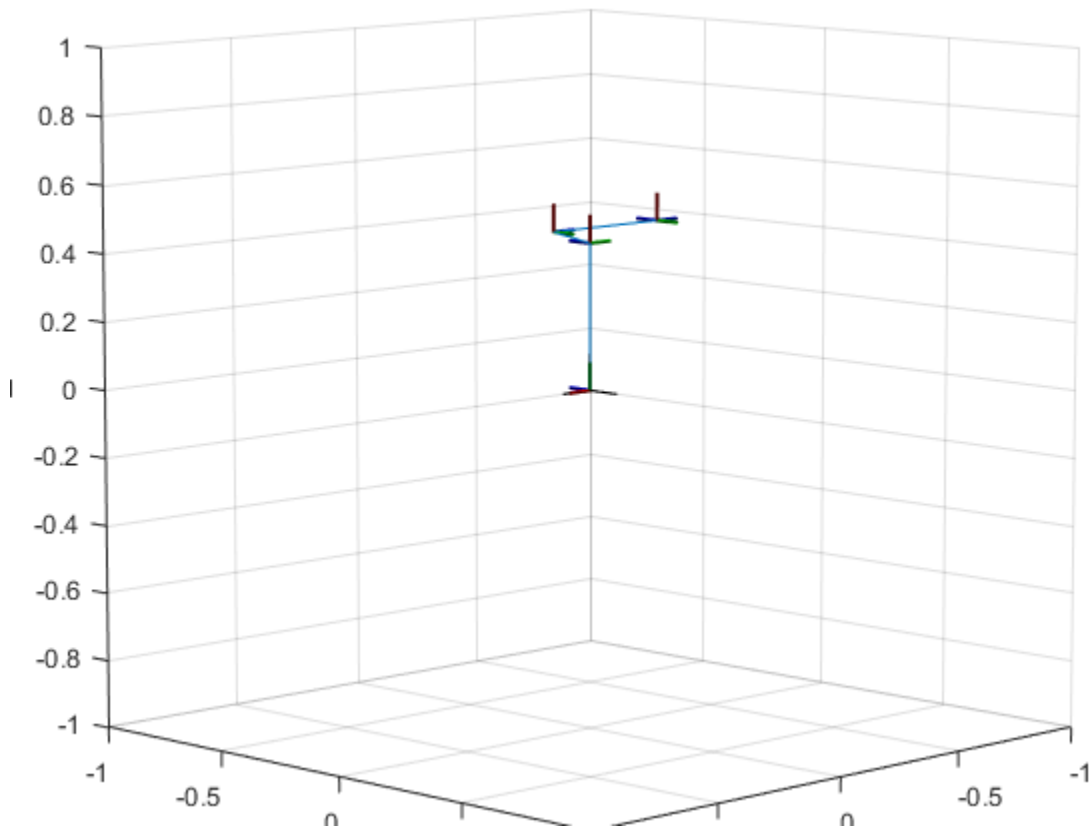
Show the home configuration using `show`. You do not need to specify a configuration input.

```
show(puma1);
```

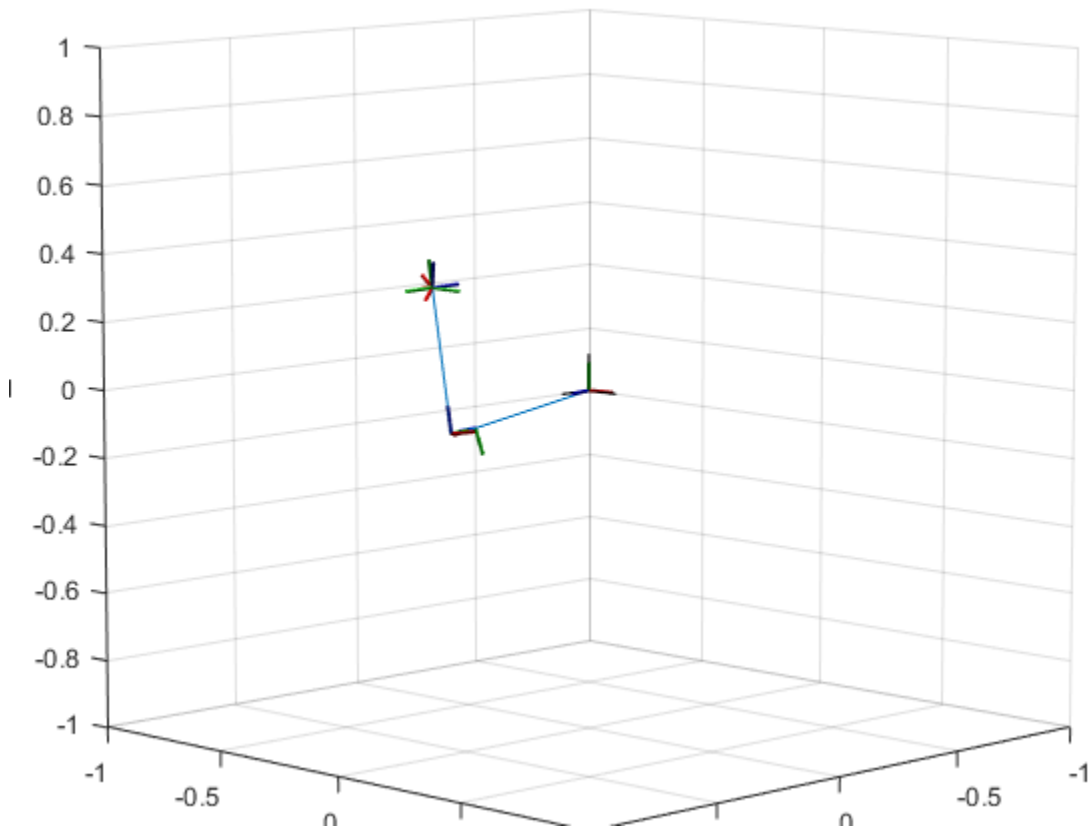
Modify the configuration and set the second joint position to $\pi/2$. Show the resulting change in the robot configuration.

```
config(2).JointPosition = pi/2;  
show(puma1,config);
```



Create random configurations and show them.

```
show(puma1,randomConfiguration(puma1));
```



See Also

[robotics.RigidBodyTree.homeConfiguration](#) | [robotics.RigidBodyTree.getTransform](#) | [robotics.RigidBodyTree.geometricJacobian](#)

Introduced in R2016b

removeBody

Class: robotics.RigidBodyTree

Package: robotics

Remove body from robot

Syntax

```
removeBody(robot, bodyname)  
newSubtree = removeBody(robot, bodyname)
```

Description

`removeBody(robot, bodyname)` removes the body and all subsequently attached bodies from the robot model.

`newSubtree = removeBody(robot, bodyname)` returns the subtree created by removing the body and all subsequently attached bodies from the robot model.

Input Arguments

robot — Robot model

RigidBodyTree object

Robot model, specified as a RigidBodyTree object.

bodyname — Body name

character vector

Body name, specified as a character vector. This body must be on the robot model specified in `robot`.

Output Arguments

newSubtree — Robot subtree

RigidBodyTree object

Robot subtree, returned as a `RigidBodyTree` object. This new subtree uses the parent name of the body specified by `bodyname` as the base name. All bodies that are attached in the previous robot model (including the body with `bodyname` specified) are added to the subtree.

Examples

Modify a Robot Rigid Body Tree Model

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

Load example robots as `RigidBodyTree` objects.

```
load exampleRobots.mat
```

View the details of the Puma robot using `showdetails`.

```
showdetails(puma1)
```

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

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

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

```
body3 = getBody(puma1, 'L3');  
childBody = body3.Children{1}  
body3Copy = copy(body3);
```

```
childBody =
```

RigidBody with properties:

```
Name: 'L4'
Joint: [1×1 robotics.Joint]
Parent: [1×1 robotics.RigidBody]
Children: {[1×1 robotics.RigidBody]}
```

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: {1×3 cell}
Base: [1×1 robotics.RigidBody]
BodyNames: {'L4' 'L5' 'L6'}
```

```
BaseName: 'L3'
```

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

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

```
showdetails(puma1)
```

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

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

See Also

`robotics.Joint` | `robotics.RigidBody` | `robotics.RigidBodyTree.addBody` | `robotics.RigidBodyTree.replaceBody`

Introduced in R2016b

replaceBody

Class: robotics.RigidBodyTree

Package: robotics

Replace body on robot

Syntax

```
replaceBody(robot, bodyname, newbody)
```

Description

`replaceBody(robot, bodyname, newbody)` replaces the body in the robot model with the new body. All properties of the body are updated accordingly, except the **Parent** and **Children** properties. The rest of the robot model is unaffected.

Input Arguments

robot — **Robot model**

RigidBodyTree object

Robot model, specified as a RigidBodyTree object. The rigid body is added to this object and attached at the rigid body specified by **bodyname**.

bodyname — **Body name**

character vector

Body name, specified as a character vector. This body must be on the robot model specified in **robot**.

newbody — **Rigid body**

RigidBody object

Rigid body, specified as a RigidBody object.

See Also

[robotics.Joint](#) | [robotics.RigidBody](#) | [robotics.RigidBodyTree.replaceJoint](#) | [robotics.RigidBodyTree.addBody](#) | [robotics.RigidBodyTree.removeBody](#)

Introduced in R2016b

replaceJoint

Class: robotics.RigidBodyTree

Package: robotics

Replace joint on body

Syntax

```
replaceJoint(robot, bodyname, joint)
```

Description

`replaceJoint(robot, bodyname, joint)` replaces the joint on the specified body in the robot model if the body is a part of the robot model. This method is the only way to change joints in a robot model. You cannot directly assign the `Joint` property of a rigid body.

Input Arguments

robot — Robot model

RigidBodyTree object

Robot model, specified as a RigidBodyTree object.

bodyname — Body name

character vector

Body name, specified as a character vector. This body must be on the robot model specified in `robot`.

joint — Replacement joint

Joint object

Replacement joint, specified as a `Joint` object.

Examples

Modify a Robot Rigid Body Tree Model

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

Load example robots as `RigidBodyTree` objects.

```
load exampleRobots.mat
```

View the details of the Puma robot using `showdetails`.

```
showdetails(puma1)
```

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

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

```
-----
```

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

```
body3 = getBody(puma1, 'L3');
childBody = body3.Children{1}
body3Copy = copy(body3);
```

```
childBody =
```

```
RigidBody with properties:
```

```
    Name: 'L4'
    Joint: [1×1 robotics.Joint]
    Parent: [1×1 robotics.RigidBody]
    Children: {[1×1 robotics.RigidBody]}
```

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: {1×3 cell}
  Base: [1×1 robotics.RigidBody]
  BodyNames: {'L4' 'L5' 'L6'}
  BaseName: 'L3'
```

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

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

```
showdetails(puma1)
```

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

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

See Also

[robotics.Joint](#) | [robotics.RigidBody](#) | [robotics.RigidBodyTree.addBody](#) | [robotics.RigidBodyTree.replaceBody](#)

Introduced in R2016b

show

Class: `robotics.RigidBodyTree`

Package: `robotics`

Show robot body frames in a figure

Syntax

```
show(robot)
show(robot, configuration)
show( ____, Name, Value)
ax = show( ____ )
```

Description

`show(robot)` plots the body frames of the robot model in a figure with the predefined home configuration.

`show(robot, configuration)` uses the joint positions specified in `configuration` to show the robot body frames.

`show(____, Name, Value)` provides additional options specified by one or more `Name, Value` pair arguments. `Name` must appear inside single quotes (`'`). You can specify several name-value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`, using any combination of previous syntaxes.

`ax = show(____)` returns the axes handle the robot is plotted on.

Input Arguments

robot — Robot model

`RigidBodyTree` object

Robot model, specified as a `RigidBodyTree` object.

configuration — Robot configuration

structure

Robot configuration, specified as a structure with joint names and positions for all the bodies in the robot model. You can generate a configuration using `homeConfiguration(robot)`, `randomConfiguration(robot)`, or by specifying your own joint positions in a structure.

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`.

'Parent' — Parent of axes

Axes object

Parent of axes, specified as the comma-separated pair consisting of **Parent** and an **Axes** object in which to draw the robot. By default, the robot is plotted in the active axes.

'PreservePlot' — Preserve robot plot

true (default) | false

Option to reserve robot plot, specified as the comma-separated pair consisting of **'PreservePlot'** and true or false. When this property is set to **true**, previous plots displayed by calling `show` are not overwritten. This setting functions similar to calling `hold on` for a standard MATLAB figure, but is limited to the robot body frames. When this property is set to **false**, previous plots of the robot are overwritten.

Output Arguments

ax — Axes graphic handle

Axes object

Axes graphic handle, returned as an **Axes** object. This object contains the properties of the figure that the robot is plotted onto.

Examples

Visualize Robot Configurations

Show different configurations of a robot created using a `RigidBodyTree` model. Use the `homeConfiguration` or `randomConfiguration` functions to generate the structure that defines all the joint positions.

Load example robots as `RigidBodyTree` objects.

```
load exampleRobots.mat
```

Create a structure for the home configuration of a Puma robot. The structure has joint names and positions for each body on the robot model.

```
config = homeConfiguration(puma1)
```

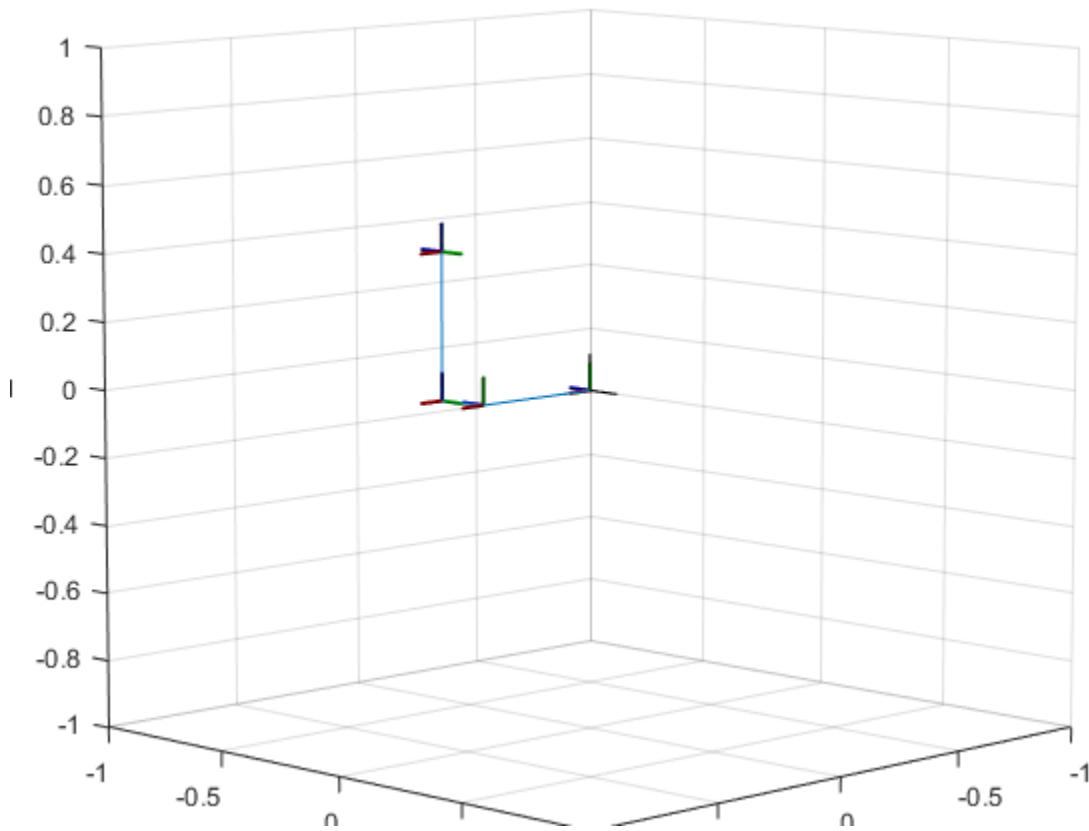
```
config =
```

```
1×6 struct array with fields:
```

```
JointName  
JointPosition
```

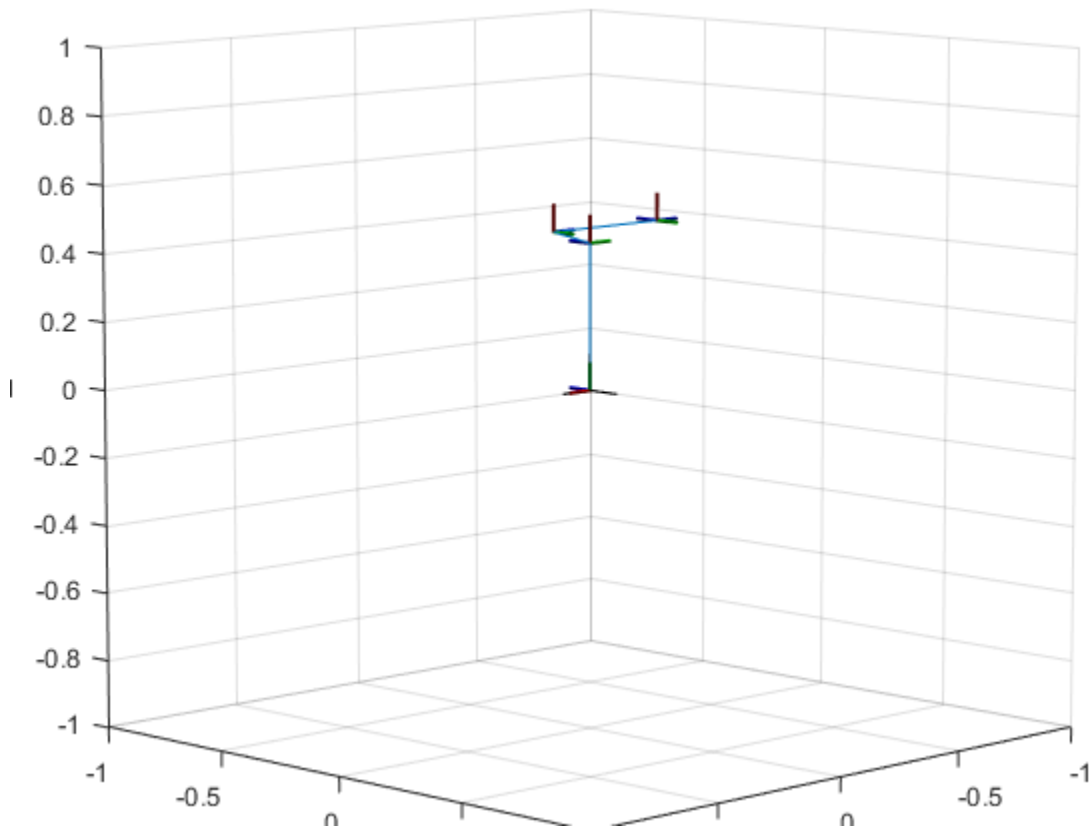
Show the home configuration using `show`. You do not need to specify a configuration input.

```
show(puma1);
```

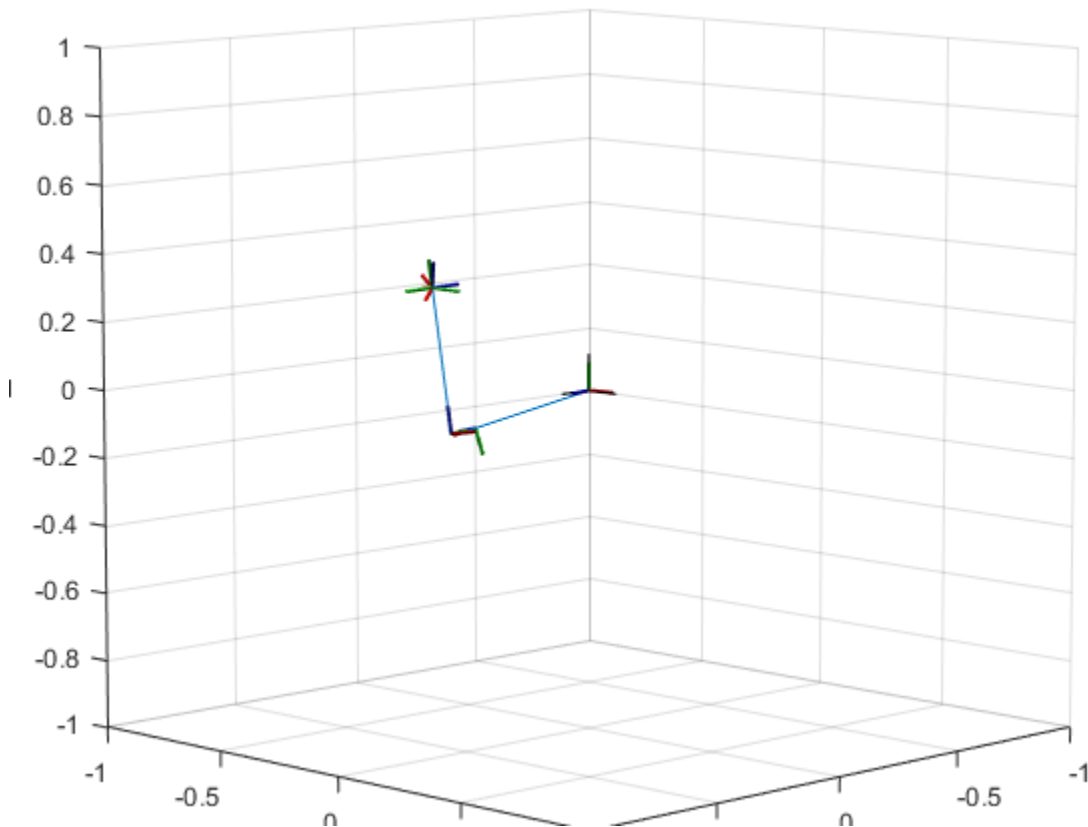
Modify the configuration and set the second joint position to $\pi/2$. Show the resulting change in the robot configuration.

```
config(2).JointPosition = pi/2;  
show(puma1,config);
```



Create random configurations and show them.

```
show(puma1,randomConfiguration(puma1));
```



Build Manipulator Robot Using Denavit-Hartenberg Parameters

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

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

```
dhparams = [0    pi/2  0    0;
            0.4318 0    0    0
```

```
0.0203 -pi/2 0.15005 0;  
0      pi/2 0.4318 0;  
0      -pi/2 0      0;  
0      0      0      0];
```

Create a rigid body tree object to build the robot.

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

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

- 1 Create a `RigidBody` object and give it a unique name.
- 2 Create a `Joint` object and give it a unique name.
- 3 Use `setFixedTransform` to specify the body-to-body transformation using DH parameters.
- 4 Call `addBody` to attach the first body joint to the base frame of the robot.

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

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

```
body2 = robotics.RigidBody('body2');  
jnt2 = robotics.Joint('jnt2','revolute');  
body3 = robotics.RigidBody('body3');  
jnt3 = robotics.Joint('jnt3','revolute');  
body4 = robotics.RigidBody('body4');  
jnt4 = robotics.Joint('jnt4','revolute');  
body5 = robotics.RigidBody('body5');  
jnt5 = robotics.Joint('jnt5','revolute');  
body6 = robotics.RigidBody('body6');  
jnt6 = robotics.Joint('jnt6','revolute');  
  
setFixedTransform(jnt2,dhparams(2,:), 'dh');  
setFixedTransform(jnt3,dhparams(3,:), 'dh');
```

```

setFixedTransform(jnt4,dhparams(4,:), 'dh' );
setFixedTransform(jnt5,dhparams(5,:), 'dh' );
setFixedTransform(jnt6,dhparams(6,:), 'dh' );

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

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

```

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

```

showdetails(robot)

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

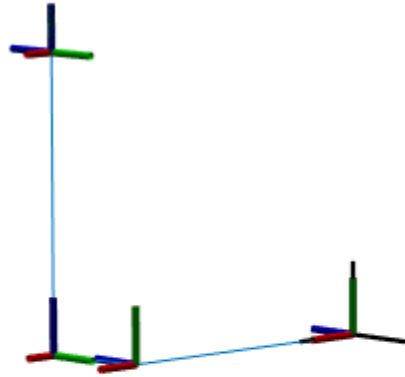
```

```

-----
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)	



See Also

`robotics.RigidBodyTree.showdetails` | `robotics.RigidBodyTree.randomConfiguration`

Introduced in R2016b

showdetails

Class: robotics.RigidBodyTree

Package: robotics

Show details of robot model

Syntax

```
showdetails(robot)
```

Description

`showdetails(robot)` displays in the MATLAB command window the details of each body in the robot model. These details include the body name, associated joint name, joint type, parent name, and children names.

Input Arguments

robot — Robot model

RigidBodyTree object

Robot model, specified as a RigidBodyTree object.

Examples

Attach Rigid Body and Joint to Rigid Body Tree

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

Create a rigid body tree.

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

Create a rigid body with a unique name.

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

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

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

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

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

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

```
showdetails(rbtree)
```

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

  Idx   Body Name   Joint Name   Joint Type   Parent Name(Idx)   Children Name(s)
  ---   -
    1      b1         jnt1        revolute        base(0)
-----
```

Modify a Robot Rigid Body Tree Model

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

Load example robots as `RigidBodyTree` objects.

```
load exampleRobots.mat
```

View the details of the Puma robot using `showdetails`.

```
showdetails(puma1)
```

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

  Idx   Body Name   Joint Name   Joint Type   Parent Name(Idx)   Children Name(s)
  ---   -
    1      L1         jnt1        revolute        base(0)           L2(2)
    2      L2         jnt2        revolute        L1(1)            L3(3)
    3      L3         jnt3        revolute        L2(2)            L4(4)
-----
```



```

4          L4          jnt4      revolute          L3(3)  L5(5)
5          L5          jnt5      revolute          L4(4)  L6(6)
6          L6          jnt6      revolute          L5(5)
-----

```

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

```

body3 = getBody(puma1, 'L3');
childBody = body3.Children{1}
body3Copy = copy(body3);

```

```

childBody =

```

```

  RigidBody with properties:

```

```

    Name: 'L4'
    Joint: [1x1 robotics.Joint]
    Parent: [1x1 robotics.RigidBody]
    Children: {[1x1 robotics.RigidBody]}

```

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: {1×3 cell}
  Base: [1×1 robotics.RigidBody]
  BodyNames: {'L4' 'L5' 'L6'}
  BaseName: 'L3'
```

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

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

```
showdetails(puma1)
```

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

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

Build Manipulator Robot Using Denavit-Hartenberg Parameters

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

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

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

Create a rigid body tree object to build the robot.

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

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

- 1 Create a `RigidBody` object and give it a unique name.
- 2 Create a `Joint` object and give it a unique name.
- 3 Use `setFixedTransform` to specify the body-to-body transformation using DH parameters.
- 4 Call `addBody` to attach the first body joint to the base frame of the robot.

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

setFixedTransform(jnt1, dhparams(1,:), 'dh');
body1.Joint = jnt1;

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

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

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

```

jnt5 = robotics.Joint('jnt5','revolute');
body6 = robotics.RigidBody('body6');
jnt6 = robotics.Joint('jnt6','revolute');

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

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

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

```

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

```
showdetails(robot)
```

```
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)	

subtree

Class: robotics.RigidBodyTree

Package: robotics

Create subtree from robot model

Syntax

```
newSubtree = subtree(robot, bodyname)
```

Description

`newSubtree = subtree(robot, bodyname)` creates a new robot model using the parent name of the body specified by `bodyname` as the base name. All subsequently attached bodies (including the body with `bodyname` specified) are added to the subtree. The original `robot` model is unaffected.

Input Arguments

robot — Robot model

RigidBodyTree object

Robot model, specified as a RigidBodyTree object.

bodyname — Body name

character vector

Body name, specified as a character vector. This body must be on the robot model specified in `robot`.

Output Arguments

newSubtree — Robot subtree

RigidBodyTree object

Robot subtree, returned as a `RigidBodyTree` object. This new subtree uses the parent name of the body specified by `bodyname` as the base name. All bodies that are attached in the previous robot model (including the body with `bodyname` specified) are added to the subtree.

See Also

`robotics.Joint` | `robotics.RigidBody` | `robotics.RigidBodyTree.addBody` | `robotics.RigidBodyTree.replaceBody`

Introduced in R2016b

robotics.VectorFieldHistogram.clone

System object: robotics.VectorFieldHistogram

Package: robotics

Create VectorFieldHistogram object with same property values

Syntax

```
vfhCopy = clone(vfh)
```

Description

`vfhCopy = clone(vfh)` creates another instance of the VectorFieldHistogram object with the same properties.

Input Arguments

vfh — Vector field histogram algorithm

VectorFieldHistogram object

Vector field histogram algorithm, specified as a VectorFieldHistogram object. This object contains all the parameters for tuning the VFH+ algorithm.

Output Arguments

vfhCopy — Vector field histogram algorithm

VectorFieldHistogram object

Vector field histogram algorithm, returned as a VectorFieldHistogram object. This object contains all the parameters for tuning the VFH+ algorithm.

Examples

Copy VectorFieldHistogram Object

Create a `VectorFieldHistogram` object.

```
vfh = robotics.VectorFieldHistogram
```

```
vfh =
```

```
System: robotics.VectorFieldHistogram
```

```
Properties:
```

```
    NumAngularSectors: 180
      DistanceLimits: [0.05 2]
        RobotRadius: 0.1
          SafetyDistance: 0.1
            MinTurningRadius: 0.1
              TargetDirectionWeight: 5
                CurrentDirectionWeight: 2
                  PreviousDirectionWeight: 2
                    HistogramThresholds: [3 10]
```

Create a copy with the same properties.

```
vfh2 = clone(vfh)
```

```
vfh2 =
```

```
System: robotics.VectorFieldHistogram
```

```
Properties:
```

```
    NumAngularSectors: 180
      DistanceLimits: [0.05 2]
        RobotRadius: 0.1
          SafetyDistance: 0.1
            MinTurningRadius: 0.1
              TargetDirectionWeight: 5
                CurrentDirectionWeight: 2
                  PreviousDirectionWeight: 2
```

```
HistogramThresholds: [3 10]
```

See Also

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

Introduced in R2015b

robotics.VectorFieldHistograms.reset

Reset internal states to default

Syntax

```
reset(vfh)
```

Description

`reset(vfh)` resets the internal states of the `VectorFieldHistogram` object to their initial values. All properties specific to the object are kept the same.

Input Arguments

vfh — Vector field histogram algorithm

`VectorFieldHistogram` object

Vector field histogram algorithm, specified as a `VectorFieldHistogram` object. This object contains all the parameters for tuning the VFH+ algorithm.

Examples

Reset VectorFieldHistogram Object

```
reset(vfh)
```

See Also

`robotics.VectorFieldHistogram` | `robotics.VectorFieldHistogram.clone` | `robotics.VectorFieldHistogram.step`

Introduced in R2015b

robotics.VectorFieldHistogram.show

System object: robotics.VectorFieldHistogram

Package: robotics

Display VectorFieldHistogram information in figure window

Syntax

```
show(vfh)
```

```
show(vfh, 'Parent', parent)
```

```
h = show( ___ )
```

Description

`show(vfh)` shows histograms calculated by the VFH+ algorithm in a figure window. The figure also includes the parameters of the `VectorFieldHistogram` object and range values from the last 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

Show Sample Data from VectorFieldHistogram Object

Create a vector field histogram object.

```
vfh = robotics.VectorFieldHistogram;  
targetDir = 0;
```

Create sample laser scan data input.

```
ranges = 10*ones(1,300);
```

```
ranges(1,110:150) = 1.0;
angles = linspace(-pi/2,pi/2,300);

Compute an obstacle-free steering direction.

steeringDir = step(vfh,ranges,angles,targetDir);

Show the data from the VectorFieldHistogram object.

show(vfh);
```

Input Arguments

vfh — Vector field histogram algorithm

VectorFieldHistogram object

Vector field histogram algorithm, specified as a VectorFieldHistogram object. This object contains all the parameters for tuning the VFH+ algorithm.

parent — Axes properties

handle

Axes properties, specified as a handle.

Output Arguments

h — Axes handles for VFH algorithm display

Axes array

Axes handles for VFH algorithm display, specified as an Axes array. The VFH histogram and HistogramThresholds are shown in the first axes. The binary histogram, range sensor readings, target direction, and steering directions are shown in the second axes.

See Also

robotics.VectorFieldHistogram | robotics.VectorFieldHistogram.step

Introduced in R2015b

robotics.VectorFieldHistogram.step

System object: robotics.VectorFieldHistogram

Package: robotics

Find obstacle-free steering direction

Syntax

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

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.

`steeringDir = step(vfh,ranges,angles,targetDir)` finds an obstacle-free steering direction using the VFH+ algorithm for input vectors, `ranges` and `angles`. The algorithm parameters are defined in the `vfh` object. A target direction is given based on the target location.

Input Arguments

vfh — Vector field histogram algorithm

VectorFieldHistogram object

Vector field histogram algorithm, specified as a VectorFieldHistogram object. This object contains all the parameters for tuning the VFH+ algorithm.

ranges — Range values from scan data

vector

Range values from scan data, specified as a vector in meters. These range values are distances from a sensor at given `angles`. The vector must be the same length as the corresponding `angles` vector.

angles — Angle values from scan data

vector

Angle values from scan data, specified as a vector in radians. These angle values are the specific angles of the given `ranges`. The vector must be the same length as the corresponding `ranges` vector.

targetDir — Target direction for robot

scalar

Target direction for the robot, specified as a scalar in radians. The forward direction of the robot is considered zero radians, with positive angles measured counterclockwise.

Output Arguments

steeringDir — Steering direction for robot

scalar

Steering direction for the robot, specified as a scalar in radians. This obstacle-free direction is calculated based on the VFH+ algorithm. The forward direction of the robot is considered zero radians, with positive angles measured counterclockwise.

Examples

Find Steering Direction of Vector Field Histogram

Create a vector field histogram object.

```
vfh = robotics.VectorFieldHistogram;
targetDir = 0;
```

Create sample laser scan data input.

```
ranges = 10*ones(1, 300);
ranges(1, 70:150) = 1.0;
angles = linspace(-pi/2, pi/2, 300);
```

Compute an obstacle-free steering direction.

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

Show the data from the `VectorFieldHistogram` object.

```
show(vfh);
```

- “Obstacle Avoidance using TurtleBot”

See Also

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

More About

- “Vector Field Histogram”

Introduced in R2015b

Blocks — Alphabetical List

Blank Message

Create blank message using specified message type

Library: Robotics System Toolbox / ROS



Description

The Blank Message block creates a Simulink nonvirtual bus corresponding to the selected ROS message type. On each sample hit, the block outputs a blank or “zero” signal for the designated message type. All elements of the bus are initialized to 0. The lengths of the variable-length arrays are also initialized to 0.

Limitations

Prior to R2016b, models using ROS message types that have certain reserved property names could not generate code. In 2016b, this limitation has been removed. These properties are renamed with an appended underscore (e.g. `Vector_`). If you use models prior to R2016b, update the ROS message types using these names and redefine custom maximum sizes for variable length arrays.

The affected message types are:

- `'geometry_msgs/Vector3Stamped'`
- `'jsk_pcl_ros/TransformScreenpointResponse'`
- `'pddl_msgs/PDDLAction'`
- `'rocon_interaction_msgs/Interaction'`
- `'capabilities/GetRemappingsResponse'`
- `'dynamic_reconfigure/Group'`

Ports

Output

Msg — Blank ROS message

nonvirtual bus

Blank ROS message, returned as a nonvirtual bus. To specify the type of ROS message, use the **Message type** parameter. All elements of the bus are initialized to 0. The lengths of the variable-length arrays are also initialized to 0.

Data Types: bus

Parameters

Message type — ROS message type

'geometry_msgs/Point' (default) | string

ROS message type, specified as a string. Use **Select** to select a message from a list of supported ROS messages. Service message types are not supported and are not included in the list.

Sample time — Interval between outputs

inf (default) | scalar

Interval between outputs, specified as a scalar. The default value indicates that the block output never changes. Using this value speeds simulation and code generation by eliminating the need to recompute the block output. Otherwise, the block outputs a new blank message at each interval of **Sample time**.

For more information, see “Specify Sample Time”.

Model Examples

See Also

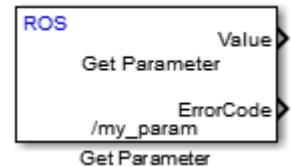
Publish | Subscribe

Introduced in R2015a

Get Parameter

Get values from ROS parameter server

Library: Robotics System Toolbox / ROS



Description

The Get Parameter block outputs the value of the specified ROS parameter. The block uses the ROS node of the Simulink model to connect to the ROS network. This node is created when you run the model and is deleted when the model terminates. If the model does not have a node, the block creates one.

On each sample hit, the block checks the ROS parameter server for the specified ROS parameter and outputs its value.

Ports

Output

Value — Parameter value

scalar | logical | uint8 array

Parameter value from the ROS network. The value depends on the **Data type** parameter.

ErrorCode — Status of ROS parameter

0 | 1 | 2 | 3

Status of ROS parameter, specified as one of the following:

- **0** — ROS parameter retrieved successfully. The retrieved value is output in the **Value** port.

- **1** — No ROS parameter with specified name found. If there is no known value, **Value** is set to the last received value or to **Initial value**.
- **2** — ROS parameter retrieved, but its type is different than the specified **Data type**. If there is no known value, Value is set to the last received value or to **Initial value**.
- **3** — For string parameters, the incoming string has been truncated based on the specified length.

Length — Length of string parameter

integer

Length of the string parameter, returned as an integer. This length is the number of elements of the `uint8` array or the number of characters in the string that you cast to `uint8`.

Note: When getting string parameters from the ROS network, an ASCII value of 13 returns an error due to its incompatible character type.

Dependencies

To enable this port, set the **Data type** to `uint8[]` (string).

Parameters

Source — Source for specifying the parameter name

Select from ROS network | Specify your own

Source for specifying the parameter name as one of the following:

- **Select from ROS network** — Use **Select** to select a parameter name. The **Data type** parameter is set automatically. You must be connected to a ROS network.
- **Specify your own** — Enter a parameter name in **Name** and specify its data type in **Data type**. You must match a parameter name exactly.

Name — Parameter name

string

Parameter name to get from the ROS network, specified as a string. When **Source** is set to **Select from ROS network**, use **Select** to select an existing parameter. You

must be connected to a ROS network to get a list of parameters. Otherwise, specify the parameter and data type.

Parameter name strings must follow the rules of ROS graph names. Valid names have these characteristics:

- The first character is an alpha character ([a-z|A-Z]), tilde (~), or forward slash (/).
- Subsequent characters are alphanumeric ([0-9|a-z|A-Z]), underscores(_), or forward slashes (/).

Data type — Data type of your parameter

double | int32 | boolean | uint8[] (string)

Data type of your parameter, specified as a string. The uint8[] (string) enables the **Maximum length** parameter.

Note: The uint8[] (string) data type is an array of ASCII values corresponding to the characters in a string. When getting string parameters, you can create a MATLAB Function block to compare the string to a desired parameter value. For more information, see “ROS String Parameters”.

Data Types: double | int32 | Boolean | uint8

Maximum length — Maximum length of the uint8 array

scalar

Maximum length of the uint8 array, specified as a scalar. If the parameter string has a length greater than **Maximum length**, the **ErrorCode** output is set to 3.

Dependencies

To enable this port, set the **Data type** to uint8[] (string).

Initial value — Default parameter value output

double | int32 | boolean | uint8

Default parameter value output from when an error occurs and no valid value has been received from the parameter server. The data type must match the specified **Data type**.

Sample time — Interval between outputs

inf (default) | scalar

Interval between outputs, specified as a scalar. This default value indicates that the block output never changes. Using this value speeds simulation and code generation by eliminating the need to recompute the block output. Otherwise, the block outputs a new blank message at each interval of **Sample time**.

For more information, see “Specify Sample Time”.

Show ErrorCode output port — Display error code output

on | off

To enable error code output, select this parameter. When you clear this parameter, the **ErrorCode** output port is removed from the block. The status options are:

- **0** — ROS parameter retrieved successfully. The retrieved value is output in the **Value** port.
- **1** — No ROS parameter with specified name found. If there is no known value, **Value** is set to the last received value or to **Initial value**.
- **2** — ROS parameter retrieved, but its type is different than the specified **Data type**. If there is no known value, **Value** is set to the last received value or to **Initial value**.
- **3** — For string parameters, the incoming string has been truncated based on the specified length.

Model Examples

More About

- ROS Parameter Server
- ROS Graph Names

See Also

Set Parameter

Introduced in R2015b

Publish

Send messages to ROS network

Library: Robotics System Toolbox / ROS



Description

The Publish block takes in as its input a Simulink nonvirtual bus that corresponds to the specified ROS message type and publishes it to the ROS network. It uses the node of the Simulink model to create a ROS publisher for a specific topic. This node is created when the model runs and is deleted when the model terminates. If the model does not have a node, the block creates one.

On each sample hit, the block converts the **Msg** input from a Simulink bus signal to a ROS message and publishes it. The block does not distinguish whether the input is a new message but merely publishes it on every sample hit. For simulation, this input is a MATLAB ROS message. In code generation, it is a C++ ROS message.

Ports

Input

Msg — ROS message

nonvirtual bus

ROS message, specified as a nonvirtual bus. To specify the type of ROS message, use the **Message type** parameter.

Data Types: bus

Parameters

Topic source — Source for specifying topic name

Select from ROS network | Specify your own

Source for specifying the topic name, specified as one of the following:

- **Select from ROS network** — Use **Select** to select a topic name. The **Topic** and **Message type** parameters are set automatically. You must be connected to a ROS network.
- **Specify your own** — Enter a topic name in **Topic** and specify its message type in **Message type**. You must match a topic name exactly.

Topic — Topic name to publish to

string

Topic name to publish to, specified as a string. When **Topic source** is set to **Select from ROS network**, use **Select** to select a topic from the ROS network. You must be connected to a ROS network to get a list of topics. Otherwise, set **Topic source** to **Specify your own** and specify the topic you want.

Message type — ROS message type

string

ROS message type, specified as a string. Use **Select** to select from a full list of supported ROS messages. Service message types are not supported and are not included in the list.

Model Examples

More About

Tips

You can also set the addresses for the ROS master and node host by clicking the **Configure network addresses** link in the block.

- “Simulink and ROS Interaction”

See Also

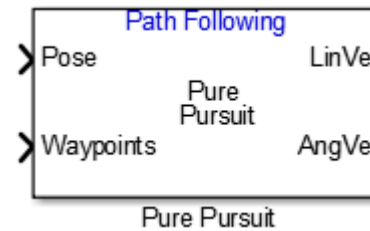
Blank Message | Subscribe

Introduced in R2015a

Pure Pursuit

Linear and angular velocity control commands

Library: Robotics System Toolbox / Mobile Robot Algorithms



Description

The Pure Pursuit block computes linear and angular velocity commands for following a path using a set of waypoints and the current pose of a differential drive robot. The block takes updated poses to update velocity commands for the robot to follow a path along a desired set of waypoints. Use the **Max angular velocity** and **Desired linear velocity** parameters to update the velocities based on the performance of the robot.

The **Lookahead distance** parameter computes a look-ahead point on the path, which is an instantaneous local goal for the robot. The angular velocity command is computed based on this point. Changing **Lookahead distance** has a significant impact on the performance of the algorithm. A higher look-ahead distance results in a smoother trajectory for the robot, but can cause the robot to cut corners along the path. Too low of a look-ahead distance can result in oscillations in tracking the path, causing unstable behavior. For more information on the pure pursuit algorithm, see “Pure Pursuit Controller”.

Ports

Input

Pose — Current robot pose

[x y theta] vector

Current robot pose, specified as an [x y theta] vector, which corresponds to the x - y position and orientation angle, θ . Positive angles are measured counterclockwise from the positive x -axis.

Waypoints — Waypoints

[] (default) | n -by-2 array

Waypoints, specified as an n -by-2 array of [x y] pairs, where n is the number of waypoints. You can generate the waypoints from the robotics.PRM class or specify them as an array in Simulink.

Output

LinVel — Linear velocity

scalar in meters per second

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

Data Types: double

AngVel — Angular velocity

scalar in radians per second

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

Data Types: double

TargetDir — Target direction for robot

scalar in radians

Target direction for the robot, specified as a scalar in radians. The forward direction of the robot is considered zero radians, with positive angles measured counterclockwise. This output can be used as the input to the **TargetDir** port for the **Vector Field Histogram** block.

Dependencies

To enable this port, select the **Show TargetDir output port** parameter.

Parameters

Desired linear velocity (m/s) – Linear velocity

0.1 (default) | scalar

Desired linear velocity, specified as a scalar in meters per second. The controller assumes that the robot drives at a constant linear velocity and that the computed angular velocity is independent of the linear velocity.

Maximum angular velocity (rad/s) – Angular velocity

1.0 (default) | scalar

Maximum angular velocity, specified as a scalar in radians per second. The controller saturates the absolute angular velocity output at the given value.

Lookahead distance (m) – Look-ahead distance

1.0 (default) | scalar

Look-ahead distance, specified as a scalar in meters. The look-ahead distance changes the response of the controller. A robot with a higher look-ahead distance produces smooth paths but takes larger turns at corners. A robot with a smaller look-ahead distance follows the path closely and takes sharp turns, but oscillate along the path. For more information on the effects of look-ahead distance, see “Pure Pursuit Controller”.

Show TargetDir output port – Target direction indicator

off (default) | on

Select this parameter to enable the **TargetDir** out port. This port gives the target direction as an angle in radians from the forward position, with positive angles measured counterclockwise.

Model Examples

More About

- “Pure Pursuit Controller”

See Also

Blocks

Publish | Subscribe | Vector Field Histogram

Classes

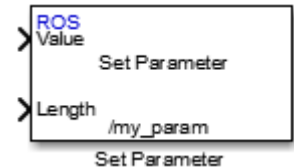
robotics.PurePursuit | robotics.PRM

Introduced in R2016b

Set Parameter

Set values on ROS parameter server

Library: Robotics System Toolbox / ROS



Description

The Set Parameter block sets the **Value** input to the specified name on the ROS parameter server. The block uses the ROS node of the Simulink model to connect to the ROS network. This node is created when you run the model and is deleted when the model terminates. If the model does not have a node, the block creates one.

Ports

Input

Value — Parameter value

scalar | logical | uint8 array

Parameter value from the ROS network. The value depends on the **Data type** parameter.

Length — Length of string parameter

integer

Length of the string parameter, specified as an integer. This length is the number of elements of the `uint8` array or the number of characters in the string that you cast to `uint8`.

Note: When casting your string parameters to `uint8`, ASCII values 0–31 (control characters) return an error due to their incompatible character type.

Dependencies

To enable this port, set the **Data type** to `uint8[]` (string).

Parameters

Source — Source for specifying the parameter name

Select from ROS network | Specify your own

Source for specifying the parameter name as one of the following:

- **Select from ROS network** — Use **Select** to select a parameter name. The **Data type** parameter is set automatically. You must be connected to a ROS network.
- **Specify your own** — Enter a parameter name in **Name** and specify its data type in **Data type**. You must match a parameter name exactly.

Name — Parameter name

string

Parameter name to get from the ROS network, specified as a string. When **Source** is set to **Select from ROS network**, use **Select** to select an existing parameter. You must be connected to a ROS network to get a list of parameters. Otherwise, specify the parameter and data type.

Parameter name strings must follow the rules of ROS graph names. Valid names have these characteristics:

- The first character is an alpha character ([a-z | A-Z]), tilde (~), or forward slash (/).
- Subsequent characters are alphanumeric ([0-9 | a-z | A-Z]), underscores(_), or forward slashes (/).

Data type — Data type of your parameter

double | int32 | boolean | uint8[] (string)

Data type of your parameter, specified as a string.

Note: The `uint8[]` (string) data type is an array of ASCII values corresponding to the characters in a string. When getting string parameters, you can create a

MATLAB Function block to compare the string to a desired parameter value. For more information, see “ROS String Parameters”.

Data Types: `double` | `int32` | `Boolean` | `uint8`

Model Examples

More About

- ROS Parameter Servers
- ROS Graph Names

See Also

Get Parameter

Introduced in R2015b

Subscribe

Receive messages from ROS network

Library: Robotics System Toolbox / ROS



Description

The Subscribe block creates a Simulink nonvirtual bus that corresponds to the specified ROS message type. The block uses the node of the Simulink model to create a ROS subscriber for a specific topic. This node is created when the model runs and is deleted when the model terminates. If the model does not have a node, the block creates one.

On each sample hit, the block checks if a new message is available on the specific topic. If a new message is available, the block retrieves the message and converts it to a Simulink bus signal. The **Msg** port outputs this new message. If a new message is not available, **Msg** outputs the last received ROS message. If a message has not been received since the start of the simulation, **Msg** outputs a blank message.

Ports

Output

IsNew — New message indicator

0 | 1

New message indicator, returned as a logical. If the output is 1, then a new message was received since the last sample hit. This output can be used to trigger subsystems for processing new messages received in the ROS network.

Msg — ROS message

nonvirtual bus

ROS message, returned as a nonvirtual bus. The type of ROS message is specified in the **Message type** parameter.

Data Types: bus

Parameters

Topic source — Source for specifying topic name

Select from ROS network | Specify your own

Source for specifying the topic name, specified as one of the following:

- **Select from ROS network** — Use **Select** to select a topic name. The **Topic** and **Message type** parameters are set automatically. You must be connected to a ROS network.
- **Specify your own** — Enter a topic name in **Topic** and specify its message type in **Message type**. You must match a topic name exactly.

Topic — Topic name to subscribe to

string

Topic name to subscribe to, specified as a string. When **Topic source** is set to **Select from ROS network**, use **Select** to select a topic from the ROS network. You must be connected to a ROS network to get a list of topics. Otherwise, set **Topic source** to **Specify your own** and specify the topic you want.

Message type — ROS message type

string

ROS message type, specified as a string. Use **Select** to select from a full list of supported ROS messages. Service message types are not supported and are not included in the list.

Sample time — Interval between outputs

-1 (default) | scalar

Interval between outputs, specified as a scalar. In simulation, the sample time follows simulation time and not actual wall-block time.

This default value indicates that the block sample time is *inherited*.

For more information about the inherited sample time type, see “Specify Sample Time”.

Model Examples

More About

Tips

You can also set the addresses for the ROS master and node host by clicking the **Configure network addresses** link in the block.

- “Simulink and ROS Interaction”

See Also

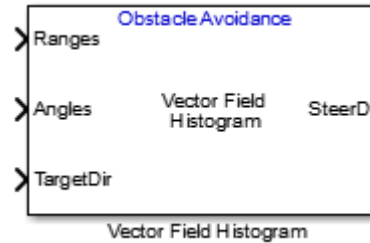
Blank Message | Publish

Introduced in R2015a

Vector Field Histogram

Avoid obstacles using vector field histogram

Library: Robotics System Toolbox / Mobile Robot Algorithms



Description

The Vector Field Histogram (VFH) block enables your robot to avoid obstacles based on range sensor data. Given a range sensor reading in terms of ranges and angles, and a target direction to drive toward, the VFH controller computes an obstacle-free steering direction.

For more information on the algorithm details, see “Vector Field Histogram” on page 4-24 under Algorithms.

Ports

Input

Ranges — Range values from scan data

vector of scalars

Range values from scan data, specified as a vector of scalars in meters. These range values are distances from a sensor at specified angles. The vector must be the same length as the corresponding **Angles** vector.

Angles — Angle values from scan data

vector of scalars

Angle values from scan data, specified as a vector of scalars in radians. These angle values are the specific angles of the specified ranges. The vector must be the same length as the corresponding **Ranges** vector.

TargetDir — Target direction for robot

scalar

Target direction for the robot, specified as a scalar in radians. The forward direction of the robot is considered zero radians, with positive angles measured counterclockwise. You can use the **TargetDir** output from the **Pure Pursuit** block when generating controls from a set of waypoints.

Output

steeringDir — Steering direction for robot

scalar

Steering direction for the robot, specified as a scalar in radians. This obstacle-free direction is calculated based on the VFH+ algorithm. The forward direction of the robot is considered zero radians, with positive angles measured counterclockwise.

Parameters

Main

Number of angular sectors — Number of bins used to create the histograms

180 (default) | scalar

Number of bins used to create the histograms, specified as a scalar. This parameter is nontunable. You can set this parameter only when the object is initialized.

Range distance limits (m) — Limits for range readings

[0.05 2] (default) | two-element vector of scalars

Limits for range readings in meters, specified as a two-element vector of scalars. The range readings specified in the step function 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 away from the robot.

Histogram thresholds — Thresholds for computing binary histogram

[3 10] (default) | two-element vector of scalars

Thresholds for computing binary histogram, specified as a two-element vector of scalars. The algorithm uses these thresholds to compute the binary histogram from the polar obstacle density. Polar obstacle density values higher than the upper threshold are represented as occupied space (1) in the binary histogram. Values smaller than the lower threshold are represented as free space (0). Values that fall between the limits are set to the values of a previous computed binary histogram if one exists from previous iterations. If a previous histogram does not exist, the value is set as free space (0).

Robot radius (m) — Radius of the robot

0.1 (default) | scalar

Radius of the robot, specified as a scalar in meters. This dimension defines the smallest circle that can circumscribe your robot. The robot radius is used to account for robot size when computing the obstacle-free direction.

Safety distance (m) — Safety distance around the robot

0.1 (default) | scalar

Safety distance left around the robot position in addition to **Robot radius**, specified as a scalar in meters. The robot radius and safety distance are used to compute the obstacle-free direction.

Minimum turning radius (m) — Minimum turning radius at current speed

0.1 (default) | scalar

Minimum turning radius for the robot moving at its current speed, specified as a scalar in meters.

Simulate using — Specify type of simulation to run

Code generation (default) | Interpreted execution

- **Code generation** — Simulate model using generated C code. The first time you run a simulation, Simulink generates C code for the block. The C code is reused for subsequent simulations, as long as the model does not change. This option requires additional startup time but the speed of the subsequent simulations is comparable to **Interpreted execution**.
- **Interpreted execution** — Simulate model using the MATLAB interpreter. This option shortens startup time but has a slower simulation speed than **Code generation**. In this mode, you can debug the source code of the block.

Tunable: No

Cost Function Weights

Target direction weight — Cost function weight for target direction

5 (default) | scalar

Cost function weight for moving toward the target direction, specified as a scalar. To follow a target direction, set this weight to be higher than the sum of **Current direction weight** and **Previous direction weight**. To ignore the target direction cost, set this weight to 0.

Current direction weight — Cost function weight for current direction

2 (default) | scalar

Cost function weight for moving the robot in the current heading direction, specified as a scalar. Higher values of this weight produce efficient paths. To ignore the current direction cost, set this weight to 0.

Previous direction weight — Cost function weight for previous direction

2 (default) | scalar

Cost function weight for moving in the previously selected steering direction, specified as a scalar. Higher values of this weight produce smoother paths. To ignore the previous direction cost, set this weight to 0.

Model Examples

More About

Algorithms

Vector Field Histogram

The block uses the VFH+ algorithm to compute the obstacle-free direction. First, the algorithm takes the ranges and angles from range sensor data and builds a polar histogram for obstacle locations. Then, it uses the input histogram thresholds to calculate

a binary histogram that indicates occupied and free directions. Finally, the algorithm computes a masked histogram, which is computed from the binary histogram based on the minimum turning radius of the robot.

The algorithm selects multiple steering directions based on the open space and possible driving directions. A cost function, with weights corresponding to the previous, current, and target directions, calculates the cost of different possible directions. The algorithm then returns an obstacle-free direction with minimal cost. Using the obstacle-free direction, you can input commands to move your robot in that direction.

To use this block for your own application and environment, you must tune the algorithm parameters. Parameter values depend on the type of robot, the range sensor, and the hardware you use. For more information on the VFH algorithm, see “Vector Field Histogram”.

- “Vector Field Histogram”

See Also

Blocks

[Publish](#) | [Pure Pursuit](#) | [Subscribe](#)

Classes

`robotics.VectorFieldHistogram`

Introduced in R2016b

