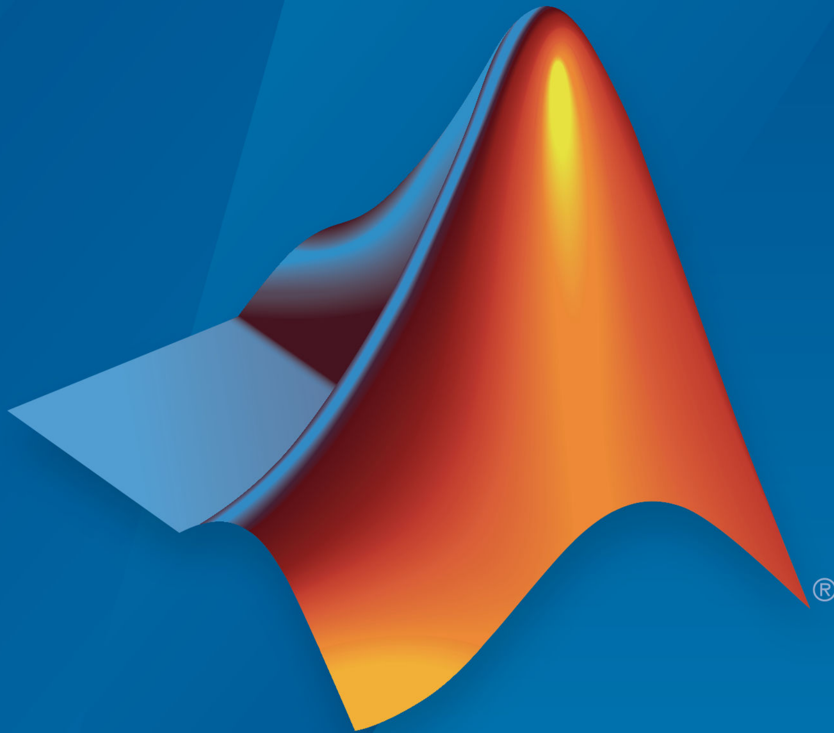
Robotics System Toolbox™

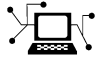
User's Guide



MATLAB® & SIMULINK®

R2018b

How to Contact MathWorks



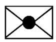
Latest news: www.mathworks.com

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™ User Guide

© COPYRIGHT 2015–2018 by The MathWorks, Inc.

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

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

Trademarks

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

Patents

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

Revision History

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

Contents

1 Coordinate System Transformations

Standard Units for Robotics System Toolbox **1-2**

Coordinate Transformations in Robotics **1-3**
Axis-Angle **1-3**
Euler Angles **1-4**
Homogeneous Transformation Matrix **1-4**
Quaternion **1-5**
Rotation Matrix **1-5**
Translation Vector **1-6**
Conversion Functions and Transformations **1-6**

Convert A ROS Pose Message To A Homogeneous Transformation **1-8**

2 ROS Network Connection

ROS Network Setup **2-2**
Introduction **2-2**
Network Connection Layout **2-2**

3 ROS Publishers, Subscribers, Services, and Actions

Built-In Message Support **3-2**
ROS Message Structure **3-2**

Limitations of ROS Messages in MATLAB **3-3**
ROS Data Type Conversions **3-3**
Supported Messages **3-4**

ROS Actions Overview **3-11**
Client to Server Relationship **3-11**
Performing Actions Workflow **3-12**
Action Messages and Functions **3-14**

Move a Turtlebot Robot Using ROS Actions **3-16**

4 ROS Log Files and Transformations

ROS Log Files (rosbags) **4-2**
Introduction **4-2**
MATLAB rosbag Structure **4-2**
Workflow for rosbag Selection **4-3**
Limitations **4-5**

5 ROS Custom Message Support

Create Custom Messages from ROS Package **5-2**

ROS Custom Message Support **5-8**
Custom Message Overview **5-8**
Custom Message Contents **5-8**
Custom Message Creation Workflow **5-11**

Install Robotics System Toolbox Add-ons **5-14**

6 Simulink ROS Concepts

Publish and Subscribe to ROS Messages in Simulink **6-2**

Selecting ROS Topics, Messages, and Parameters **6-5**
Selecting ROS Topics **6-5**
Selecting ROS Message Types **6-6**
Selecting ROS Parameter Names **6-7**

Configure ROS Network Addresses **6-9**

Managing Array Sizes in Simulink ROS **6-13**

Connect to ROS Device **6-15**

Simulink and ROS Interaction **6-16**
MATLAB ROS Information **6-16**
Simulink ROS Node **6-16**
Differences Between Simulation and Generated Code **6-17**
Publishers and Subscribers in Simulink **6-17**

ROS Parameters in Simulink **6-18**
Get and Set ROS Parameters **6-18**

ROS String Parameters **6-20**
Set String Parameter on ROS Network **6-20**
Get ROS String Parameter and Compare to Specified String **6-21**
Check Image Encoding Parameter for ROS Image Message . **6-21**

Enable ROS Time Model Stepping for Deployed ROS Nodes **6-23**

Overrun Detection with Deployed ROS Nodes **6-25**

ROS Simulink Support and Limitations **6-27**

Read A ROS Image Message In Simulink® **6-28**

Read A ROS Point Cloud Message In Simulink® **6-32**

Convert Coordinate System Transformations **6-37**

Call ROS Service in Simulink **6-38**

Play Back Data from Jackal™ rosbag Logfile in Simulink . . . **6-40**

Time Stamp A ROS Message Using Current Time in Simulink **6-42**

7 Algorithm Design

Occupancy Grids **7-2**
- Overview **7-2**
- World and Grid Coordinates **7-3**
- Inflation of Coordinates **7-6**
- Log-Odds Representation of Probability Values **7-11**

Particle Filter Parameters **7-14**
- Number of Particles **7-14**
- Initial Particle Location **7-15**
- State Transition Function **7-17**
- Measurement Likelihood Function **7-18**
- Resampling Policy **7-18**
- State Estimation Method **7-19**

Particle Filter Workflow **7-21**
- Estimation Workflow **7-22**
- Estimate Robot Position in a Loop Using Particle Filter **7-26**

Probabilistic Roadmaps (PRM) **7-30**
- Tune the Number of Nodes **7-30**
- Tune the Connection Distance **7-34**
- Create or Update PRM **7-37**

Pure Pursuit Controller **7-41**
- Reference Coordinate System **7-41**
- Look Ahead Distance **7-42**
- Limitations **7-43**

Vector Field Histogram **7-44**
Robot Dimensions **7-44**
Cost Function Weights **7-46**
Histogram Properties **7-47**
Tune Parameters Using show **7-51**

Monte Carlo Localization Algorithm **7-52**
Overview **7-52**
State Representation **7-53**
Initialization of Particles **7-55**
Resampling Particles and Updating Pose **7-57**
Motion and Sensor Model **7-58**

Compose a Series of Laser Scans with Pose Changes **7-63**

Rigid Body Tree Robot Model **7-68**
Rigid Body Tree Components **7-68**
Robot Configurations **7-71**

Build a Robot Step by Step **7-74**

Inverse Kinematics Algorithms **7-79**
Choose an Algorithm **7-79**
Solver Parameters **7-80**
Solution Information **7-81**
References **7-82**

8 Manipulator Algorithms

2-D Path Tracing With Inverse Kinematics **8-2**

Trace An End-Effector Trajectory with Inverse Kinematics in Simulink® **8-7**

Solve Inverse Kinematics for a Four-Bar Linkage **8-14**

Robot Dynamics **8-19**
Dynamics Properties **8-19**
Dynamics Functions **8-20**

Calculate Manipulator Gravity Dynamics in Simulink **8-21**

Compute Velocity Product for Manipulators in Simulink . . . **8-23**

Compute Geometric Jacobian for Manipulators in Simulink **8-26**

Get Transformations for Manipulator Bodies in Simulink . . . **8-28**

Get Mass Matrix for Manipulators in Simulink **8-31**

9 Application Design

Transform Laser Scan Data From A ROS Network **9-2**

Obstacle Avoidance with TurtleBot and VFH **9-4**

Execute Code at a Fixed-Rate **9-7**
- Introduction **9-7**
- Send Fixed-rate Control Commands To A Robot **9-7**
- Fixed-rate Publishing of ROS Image Data **9-9**
- Overrun Actions for Fixed Rate Execution **9-11**

Reduce Drift in 3-D Visual Odometry Trajectory Using Pose Graphs **9-14**

Build Occupancy Map from Depth Images Using Visual Odometry and Optimized Pose Graph **9-18**

10 Code Generation

Code Generation from MATLAB Code **10-2**

Code Generation Support, Usage Notes and Limitations **10-4**

Generate Code to Manually Deploy a ROS Node from Simulink **10-8**
Prerequisites **10-8**
Configure A Model for Code Generation **10-8**
Configure the Build Options for Code Generation **10-10**
Generate and Deploy the Code **10-11**

Accelerate Robotics Algorithms with Code Generation **10-14**
Create Separate Function for Algorithm **10-14**
Perform Code Generation for Algorithm **10-15**
Check Performance of Generated Code **10-15**
Replace Algorithm Function with MEX Function **10-16**

Enable External Mode for Robotics System Toolbox Models **10-18**

Tune Parameters and View Signals on Deployed Robot Models Using External Mode **10-19**
Set Up the Simulink Model **10-19**
Deploy and Run the Model **10-20**
Monitor Signals and Tune Parameters **10-21**

Coordinate System Transformations

- "Standard Units for Robotics System Toolbox" on page 1-2
- "Coordinate Transformations in Robotics" on page 1-3
- "Convert A ROS Pose Message To A Homogeneous Transformation" on page 1-8

Standard Units for Robotics System Toolbox

Robotics System Toolbox uses a fixed set of standards for units to ensure consistency across algorithms and applications. Unless specified otherwise, functions and classes in this toolbox represent all values in units based on the International System of Units (SI). The table below summarizes the relevant quantities and their SI derived units.

Quantity	**Unit (abbrev.)**
Length	meter (m)
Time	second (s)
Angle	radian (rad)
Velocity	meter/second (m/s)
Angular Velocity	radian/second (rad/s)
Acceleration	meter/second^2 (m/s^2)
Angular Acceleration	radian/second^2 (rad/s^2)
Mass	kilogram (kg)
Force	Newton (N)
Torque	Newton-meter (N-m)
Moment of Inertia	kilogram-meter^2 (kg-m^2)

See Also

More About

• “Coordinate Transformations in Robotics” on page 1-3

Coordinate Transformations in Robotics

In this section...
“Axis-Angle” on page 1-3
“Euler Angles” on page 1-4
“Homogeneous Transformation Matrix” on page 1-4
“Quaternion” on page 1-5
“Rotation Matrix” on page 1-5
“Translation Vector” on page 1-6
“Conversion Functions and Transformations” on page 1-6

In robotics applications, many different coordinate systems can be used to define where robots, sensors, and other objects are located. In general, the location of an object in 3-D space can be specified by position and orientation values. There are multiple possible representations for these values, some of which are specific to certain applications. Translation and rotation are alternative terms for position and orientation. Robotics System Toolbox supports representations that are commonly used in robotics and allows you to convert between them. You can transform between coordinate systems when you apply these representations to 3-D points. These supported representations are detailed below with brief explanations of their usage and numeric equivalent in MATLAB®. Each representation has an abbreviation for its name. This is used in the naming of arguments and conversion functions that are supported in this toolbox.

At the end of this section, you can find out about the conversion functions that we offer to convert between these representations.

Robotics System Toolbox assumes that positions and orientations are defined in a right-handed Cartesian coordinate system.

Axis-Angle

Abbreviation: axang

A rotation in 3-D space described by a scalar rotation around a fixed axis defined by a vector.

Numeric Representation: 1-by-3 unit vector and a scalar angle combined as a 1-by-4 vector

For example, a rotation of `pi/2` radians around the *y*-axis would be:

```
axang = [0 1 0 pi/2]
```

Euler Angles

Abbreviation: `eul`

Euler angles are three angles that describe the orientation of a rigid body. Each angle is a scalar rotation around a given coordinate frame axis. The Robotics System Toolbox supports two rotation orders. The `'ZYZ'` axis order is commonly used for robotics applications. We also support the `'ZYX'` axis order which is also denoted as "Roll Pitch Yaw (rpy)." Knowing which axis order you use is important for apply the rotation to points and in converting to other representations.

Numeric Representation: 1-by-3 vector of scalar angles

For example, a rotation around the *y* -axis of pi would be expressed as:

```
eul = [0 pi 0]
```

Note: The axis order is not stored in the transformation, so you must be aware of what rotation order is to be applied.

Homogeneous Transformation Matrix

Abbreviation: `tform`

A homogeneous transformation matrix combines a translation and rotation into one matrix.

Numeric Representation: 4-by-4 matrix

For example, a rotation of angle α around the *y* -axis and a translation of 4 units along the *y* -axis would be expressed as:

```
tform =
 cos α  0      sin α  0
 0      1      0      4
-sin α  0      cos α  0
 0      0      0      1
```

You should **pre-multiply** your transformation matrix with your homogeneous coordinates, which are represented as a matrix of row vectors (*n*-by-4 matrix of points). Utilize the transpose (') to rotate your points for matrix multiplication. For example:

```
points = rand(100,4);
tformPoints = (tform*points')';
```

Quaternion

Abbreviation: `quat`

A quaternion is a four-element vector with a scalar rotation and 3-element vector. Quaternions are advantageous because they avoid singularity issues that are inherent in other representations. The first element, *w*, is a scalar to normalize the vector with the three other values, *[x y z]* defining the axis of rotation.

Numeric Representation: 1-by-4 vector

For example, a rotation of `pi/2` around the *y* -axis would be expressed as:

```
quat = [0.7071 0 0.7071 0]
```

Rotation Matrix

Abbreviation: `rotm`

A rotation matrix describes a rotation in 3-D space. It is a square, orthonormal matrix with a determinant of 1.

Numeric Representation: 3-by-3 matrix

For example, a rotation of **α** degrees around the *x*-axis would be:

```
rotm =

     1     0         0
     0     cos α     -sin α
     0     sin α     cos α
```

You should **pre-multiply** your rotation matrix with your coordinates, which are represented as a matrix of row vectors (*n*-by-3 matrix of points). Utilize the transpose (') to rotate your points for matrix multiplication. For example:

```
points = rand(100,3);
rotPoints = (rotm*points')';
```

Translation Vector

Abbreviation: `trvec`

A translation vector is represented in 3-D Euclidean space as Cartesian coordinates. It only involves coordinate translation applied equally to all points. There is no rotation involved.

Numeric Representation: 1-by-3 vector

For example, a translation by 3 units along the *x* -axis and 2.5 units along the *z* -axis would be expressed as:

```
trvec = [3 0 2.5]
```

Conversion Functions and Transformations

Robotics System Toolbox provides conversion functions for the previously mentioned transformation representations. Not all conversions are supported by a dedicated function. Below is a table showing which conversions are supported (in blue). The abbreviations for the rotation and translation representations are shown as well.

Converting To / Converting From	Axis-Angle (axang)	Euler Angles (eul)	Quaternion (quat)	Rotation Matrix (rotm)	Homogeneous Transformation (tform)	Translation Vector (trvec)
Axis-Angle (axang)						
Euler Angles (eul)						
Quaternion (quat)						
Rotation Matrix (rotm)						
Homogeneous Transformation (tform)						
Translation Vector (trvec)						

The names of all the conversion functions follow a standard format. They follow the form `alpha2beta` where `alpha` is the abbreviation for what you are converting from and `beta`

is what you are converting to as an abbreviation. For example, converting from Euler angles to quaternion would be `eul2quat`.

All the functions expect valid inputs. If you specify invalid inputs, the outputs will be undefined.

There are other conversion functions for converting between radians and degrees, Cartesian and homogeneous coordinates, and for calculating wrapped angle differences. For a full list of conversions, see “Coordinate System Transformations”.

See Also

More About

- “Standard Units for Robotics System Toolbox” on page 1-2

Convert A ROS Pose Message To A Homogeneous Transformation

This model subscribes to a `Pose` message on the ROS network. Use bus selectors to extract the rotation and translation vectors. The Coordinate Transformation Conversion block takes the rotation vector (euler angles) and translation vector in and gives the homogeneous transformation for the message.

Connect to a ROS network. Create a publisher for the `'/pose'` topic using a `'geometry_msgs/Pose'` message type.

```
rosinit
[pub,msg] = rospublisher('/pose','geometry_msgs/Pose');
```

```
Initializing ROS master on http://bat5838win64:59974/.
Initializing global node /matlab_global_node_40051 with NodeURI http://bat5838win64:599
```

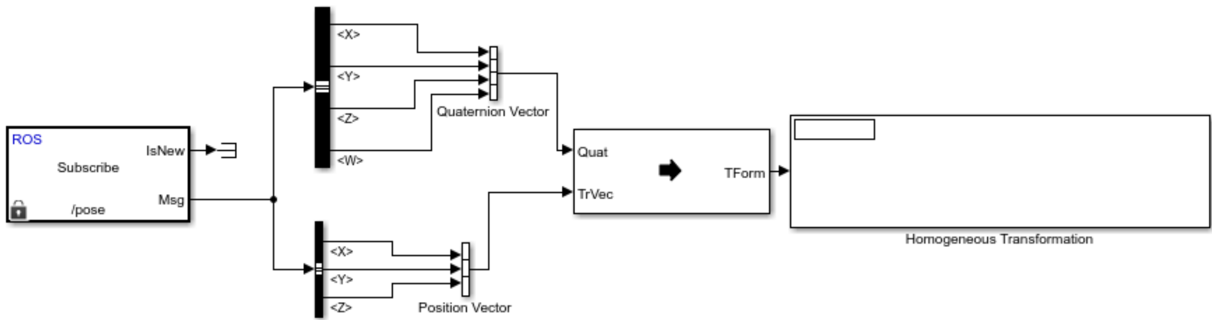
Specify the detailed pose information. The message contains a translation (`Position`) and quaternion (`Orientation`) to express the pose. Send the message via the publisher.

```
msg.Position.X = 1;
msg.Position.Y = 2;
msg.Position.Z = 3;
msg.Orientation.X = sqrt(2)/2;
msg.Orientation.Y = sqrt(2)/2;
msg.Orientation.Z = 0;
msg.Orientation.W = 0;

send(pub,msg)
```

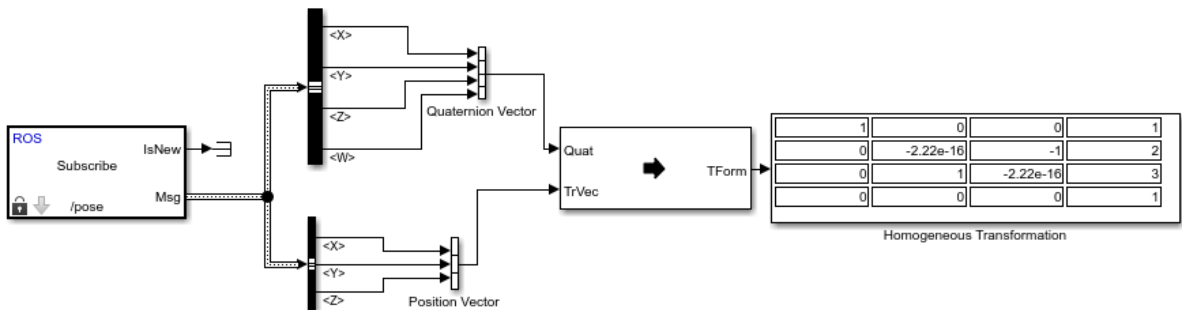
Open the `'pose_to_transformation_model'` model. This model subscribes to the `'/pose'` topic in ROS. The bus selectors extract the quaternion and position vectors from the ROS message. The Coordinate Transformation Conversion block then converts the position (translation) and quaternion to a homogeneous transformation.

For more details, inspect the bus selector in the model to see how the message information is extracted.

```
open_system('pose_to_transformation_model.slx')
```

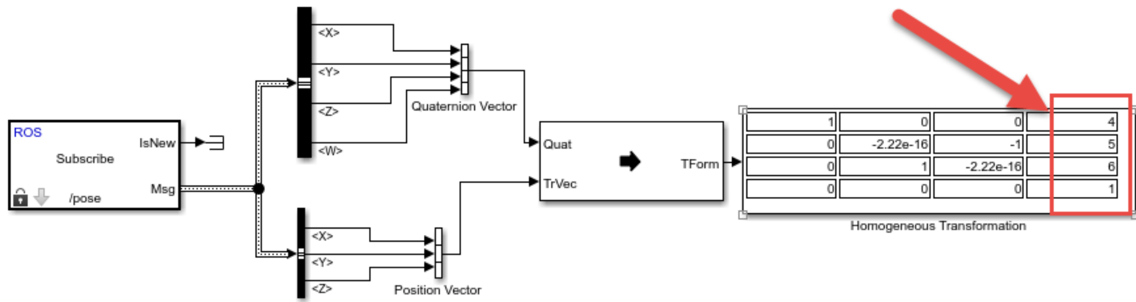


Run the model to display the homogeneous transformation.



Modify the position or orientation components of the message. Resend the message and run model to see the change in the homogeneous transformation.

```
msg.Position.X = 4;
msg.Position.Y = 5;
msg.Position.Z = 6;
send(pub,msg)
```



Shutdown the ROS network.

```
rosshutdown
```

```
Shutting down global node /matlab_global_node_40051 with NodeURI http://bat5838win64:59
Shutting down ROS master on http://bat5838win64:59974/.
```

2

ROS Network Connection

ROS Network Setup

In this section...

"Introduction" on page 2-2
"Network Connection Layout" on page 2-2

Introduction

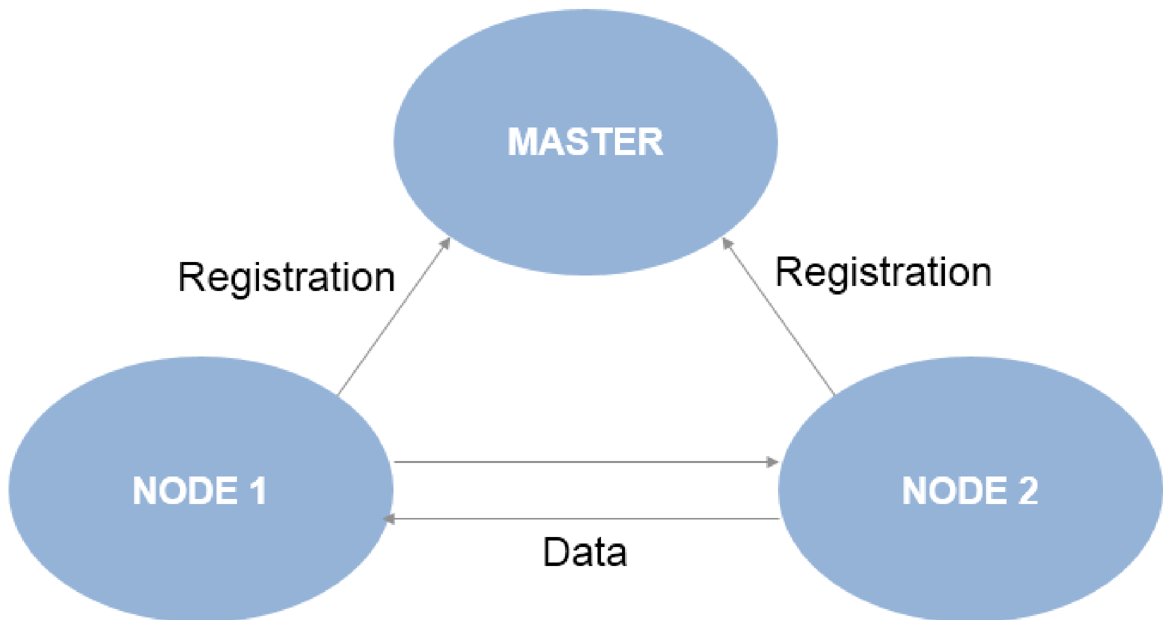
Setting up a ROS network allows for communication between different devices. Different participants or *nodes* all register with a ROS master to share information. The ROS master is unique and each ROS network only has one master. Each node is usually a separate device, although one device can have multiple nodes running. MATLAB acts as one of these nodes when using it to communicate with ROS.

All devices must be connected to the same actual or virtual network for ROS connections to work. You can create a new ROS master in MATLAB, or you can connect to an existing ROS master that is running on a different device. If you connect to an external master, you have to know the IP address or hostname of the device. The initial ROS master connection is done by calling `rosinit`. For more information on setting up and using the ROS network, see "Network Connection and Exploration".

Data communication is achieved by sending messages using entities called publishers, subscribers, and services. Publishers send data via topic names, which subscribers then receive over the network. Services use clients to request information from a server. For more information on sending messages, see "Publishers and Subscribers"

Network Connection Layout

The ROS network is a collection of nodes that are all connected to the ROS master. The number of nodes can be quite large depending on your application and devices. When nodes get registered with the master, communication with all other nodes becomes possible. Each node registers different publishers, subscribers, and services on the ROS master to send and receive information between nodes. Even though all nodes in the ROS network are registered with the master, data is exchanged directly between nodes. The following figure shows the layout of a ROS network with two ROS nodes. It is important that all nodes have bidirectional connectivity to share data across the network. Verifying these connections is important during setup.



Each node registers its own Node URI with the master. Other participants in the ROS network will use this URI to contact the node. Again, this URI must be reachable by every other node in the ROS network. To create a node in MATLAB, call `rosinit`. If a ROS master is already set up, MATLAB detects it and sets the Node URI appropriately. Otherwise, it creates both a ROS master and node that are connected.

By default, each MATLAB instance has a single "global" node. The node has a randomly-generated name assigned to it for uniqueness. All publishers, subscribers, service clients, and service servers will operate on this global node.

See Also

`rosinit` | `rosnode` | `rostopic`

Related Examples

- "Get Started with ROS"

- “Connect to a ROS Network”
- “Robot Operating System (ROS)”

ROS Publishers, Subscribers, Services, and Actions

- "Built-In Message Support" on page 3-2
- "ROS Actions Overview" on page 3-11
- "Move a Turtlebot Robot Using ROS Actions" on page 3-16

Built-In Message Support

In this section...
“ROS Message Structure” on page 3-2
“Limitations of ROS Messages in MATLAB” on page 3-3
“ROS Data Type Conversions” on page 3-3
“Supported Messages” on page 3-4

MATLAB has support for a large library of ROS message types. How messages are structured, limitations for ROS messages, and supported ROS data types are described to understand how MATLAB works with ROS messages. Also, a full list of built-in message types are shown.

ROS Message Structure

In MATLAB, ROS messages are stored as handle objects. Therefore, all the rules of handle objects apply including copying, modifying and other performance considerations. For more information on handle objects, see “Handle Object Behavior” (MATLAB). Each handle points to the object for that specific message, which contains the information relevant to that message type. The message type has a built in structure for the data it contains.

ROS messages are similar to *structure arrays* with how they store the data relevant to that message type. Each message type has a specific set of properties with their corresponding values that are individually stored and accessed. You can specifically point to and modify each property on its own. All messages have the `MessageType` property to view the message type as a character vector. Also, you can use the `showdetails` function to view the contents of the message.

Here is a sample `'geometry_msgs/Point'` created in MATLAB using `rosmessage`. It contains 3 properties corresponding to a 3-D point in XYZ coordinates.

```
pointMsg = rosmessage('geometry_msgs/Point')
```

```
pointMsg = 

  ROS Point message with properties:

    MessageType: 'geometry_msgs/Point'
```

```
              X: 0
              Y: 0
              Z: 0

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

You can access and modify each property by using the created `pointMsg` handle.

```
pointMsg.Y = 2
```

```
pointMsg = 

  ROS Point message with properties:

    MessageType: 'geometry_msgs/Point'
              X: 0
              Y: 2
              Z: 0

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

To explore further the ROS message structure in MATLAB, see “Work with Basic ROS Messages”.

Limitations of ROS Messages in MATLAB

Because ROS messages use independent properties, certain messages with multiple values cannot be validated. Because each value can be set separately, the message does not validate the properties as a whole entity. For example, a quaternion message contains *w*, *x*, *y*, and *z* properties, but the message does not enforce that the quaternion as a whole is valid. When modifying properties, you should ensure you are maintaining the rules required for that message.

Message properties can also have a variety of data types. MATLAB uses the rules set by ROS to determine what these data types are. However, if they are to be used in calculations, you might have to cast them to another value. The ROS data types do not convert directly to MATLAB data types. For a detailed list of ROS data types and their MATLAB equivalent, see “ROS Data Type Conversions” on page 3-3.

ROS Data Type Conversions

ROS message types have predetermined properties and data types for the values of those properties. These data types must be mapped to MATLAB data types to be used in

MATLAB. This table summarizes how ROS data types are converted to MATLAB data types.

ROS Data Type	Description	MATLAB
bool	Boolean / Unsigned 8-bit integer	`logical`
int8	Signed 8-bit integer	`int8`
uint8	Unsigned 8-bit integer	`uint8`
int16	Signed 16-bit integer	`int16`
uint16	Unsigned 16-bit integer	`uint16`
int32	Signed 32-bit integer	`int32`
uint32	Unsigned 32-bit integer	`uint32`
int64	Signed 64-bit integer	`int64`
uint64	Unsigned 64-bit integer	`uint64`
float32	32-bit IEEE floating point	`single`
float64	64-bit IEEE floating point	`double`
string	ASCII string (utf-8 only)	`char`
time	Seconds and nanoseconds as signed 32-bit integers	`Time` object (see `rostime`)
duration	Seconds and nanoseconds as signed 32-bit integers	`Duration` object (see `rosduration`)

Supported Messages

Here is an alphabetized list of supported ROS packages. A package can contain message types, service types, or action types.

To get a full list of supported message types, call `rosmsg list` in the MATLAB Command Window.

Robotics System Toolbox supports ROS Indigo and Hydro platforms, but your own ROS installation may have different message versions. If you would like to overwrite our current message catalog, you can utilize the “Custom Message Support” to generate new message definitions.

When specifying message types, input character vectors must match the character vector listed in `rosmsg list` exactly. To use custom message types, MATLAB also provides a custom message support package. For more information, see "Install Robotics System Toolbox Add-ons" on page 5-14.

```
ackermann_msgs
actionlib
actionlib_msgs
actionlib_tutorials
adhoc_communication
app_manager
applanix_msgs
ar_track_alvar
arbotix_msgs
ardrone_autonomy
asmach_tutorials
audio_common_msgs
axis_camera
base_local_planner
baxter_core_msgs
baxter_maintenance_msgs
bayesian_belief_networks
blob
bond
brics_actuator
bride_tutorials
bwi_planning
bwi_planning_common
calibration_msgs
capabilities
clearpath_base
cmvision
cob_base_drive_chain
cob_camera_sensors
cob_footprint_observer
cob_grasp_generation
cob_kinematics
cob_light
cob_lookat_action
cob_object_detection_msgs
cob_perception_msgs
cob_phidgets
cob_pick_place_action
cob_relayboard
```

```
cob_script_server
cob_sound
cob_srvs
cob_trajectory_controller
concert_msgs
control_msgs
control_toolbox
controller_manager_msgs
costmap_2d
create_node
data_vis_msgs
designator_integration_msgs
diagnostic_msgs
dna_extraction_msgs
driver_base
dynamic_reconfigure
dynamic_tf_publisher
dynamixel_controllers
dynamixel_msgs
epos_driver
ethercat_hardware
ethercat_trigger_controllers
ethzasl_icp_mapper
explorer
face_detector
fingertip_pressure
frontier_exploration
gateway_msgs
gazebo_msgs
geographic_msgs
geometry_msgs
gps_common
graft
graph_msgs
grasp_stability_msgs
grasping_msgs
grizzly_msgs
handle_detector
hector_mapping
hector_nav_msgs
hector_uav_msgs
hector_worldmodel_msgs
household_objects_database_msgs
hrpsys_gazebo_msgs
```

```
humanoid_nav_msgs
iai_content_msgs
iai_kinematics_msgs
iai_pancake_perception_action
image_cb_detector
image_exposure_msgs
image_view2
industrial_msgs
interaction_cursor_msgs
interactive_marker_proxy
interval_intersection
jaco_msgs
joint_states_settler
jsk_footstep_controller
jsk_footstep_msgs
jsk_gui_msgs
jsk_hark_msgs
jsk_network_tools
jsk_pcl_ros
jsk_perception
jsk_rviz_plugins
jsk_topic_tools
keyboard
kingfisher_msgs
kobuki_msgs
kobuki_testsuite
laser_assembler
laser_cb_detector
leap_motion
linux_hardware
lizi
manipulation_msgs
map_merger
map_msgs
map_store
mavros
microstrain_3dmgx2_imu
ml_classifiers
mln_robosherlock_msgs
mongodb_store
mongodb_store_msgs
monocam_settler
move_base_msgs
moveit_msgs
```

```
moveit_simple_grasps
multimaster_msgs_fkie
multisense_ros
nao_interaction_msgs
nao_msgs
nav2d_msgs
nav2d_navigator
nav2d_operator
nav_msgs
navfn
network_monitor_udp
nmea_msgs
nodelet
object_recognition_msgs
octomap_msgs
p2os_driver
pano_ros
pcl_msgs
pcl_ros
pddl_msgs
people_msgs
play_motion_msgs
polled_camera
posedetection_msgs
pr2_calibration_launch
pr2_common_action_msgs
pr2_controllers_msgs
pr2_gazebo_plugins
pr2_gripper_sensor_msgs
pr2_mechanism_controllers
pr2_mechanism_msgs
pr2_msgs
pr2_power_board
pr2_precise_trajectory
pr2_self_test_msgs
pr2_tilt_laser_interface
program_queue
ptu_control
qt_tutorials
r2_msgs
razer_hydra
rmp_msgs
robot_mechanism_controllers
robot_pose_ekf
```

```
roboteq_msgs
robotnik_msgs
rocon_app_manager_msgs
rocon_interaction_msgs
rocon_service_pair_msgs
rocon_std_msgs
rosapi
rosauth
rosbridge_library
roscpp
roscpp_tutorials
roseus
rosgraph_msgs
rospy_message_converter
rospy_tutorials
rosruby_tutorials
rosserial_arduino
rosserial_msgs
rovio_shared
rtt_ros_msgs
s3000_laser
saphari_msgs
scanning_table_msgs
scheduler_msgs
schunk_sdh
segbot_gui
segbot_sensors
segbot_simulation_apps
segway_rmp
sensor_msgs
shape_msgs
shared_serial
sherlock_sim_msgs
simple_robot_control
smach_msgs
sound_play
speech_recognition_msgs
sr_edc_ethercat_drivers
sr_robot_msgs
sr_ronex_msgs
sr_utilities
statistics_msgs
std_msgs
std_srvs
```

```
stdr_msgs
stereo_msgs
stereo_wall_detection
tf
tf2_msgs
theora_image_transport
topic_proxy
topic_tools
trajectory_msgs
turtle_actionlib
turtlebot_actions
turtlebot_calibration
turtlebot_msgs
turtlesim
um6
underwater_sensor_msgs
universal_teleop
uuid_msgs
velodyne_msgs
view_controller_msgs
visp_camera_calibration
visp_hand2eye_calibration
visp_tracker
visualization_msgs
wfov_camera_msgs
wge100_camera
wifi_ddwrt
wireless_msgs
yocs_msgs
zeroconf_msgs
```

See Also

`rosmessage` | `rosmsg` | `showdetails`

Related Examples

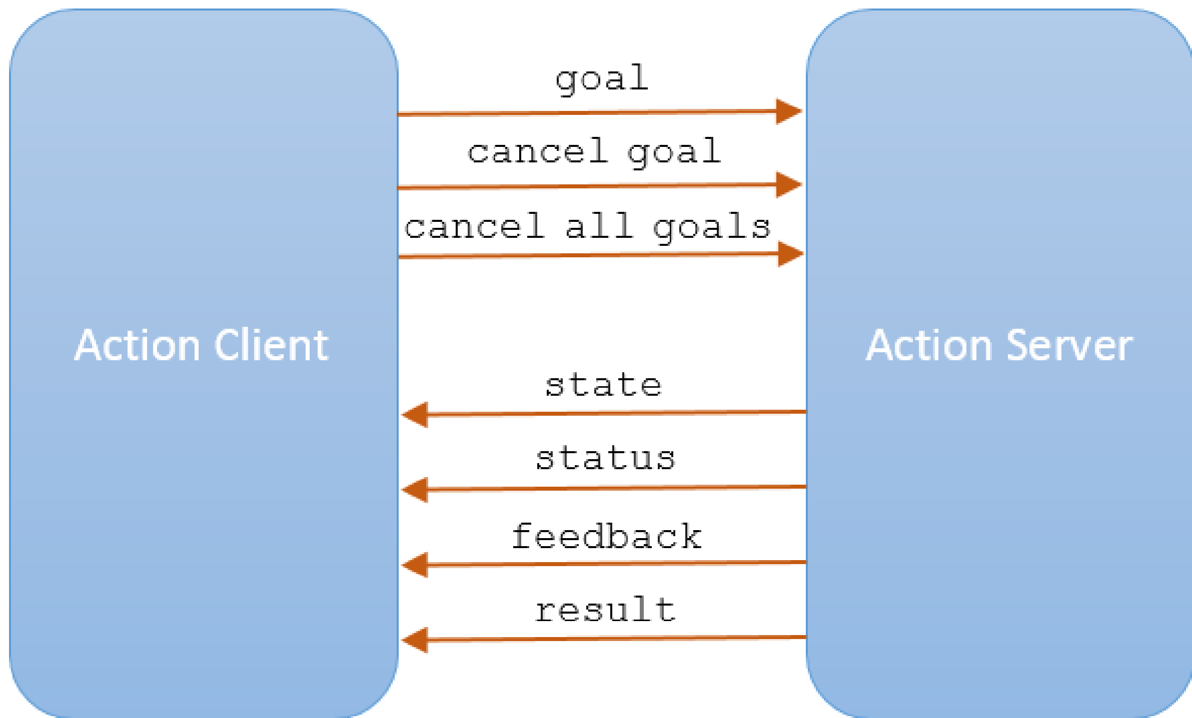
- “Work with Basic ROS Messages”
- “Exchange Data with ROS Publishers and Subscribers”
- “Work with Specialized ROS Messages”

ROS Actions Overview

In this section...
"Client to Server Relationship" on page 3-11
"Performing Actions Workflow" on page 3-12
"Action Messages and Functions" on page 3-14

Client to Server Relationship

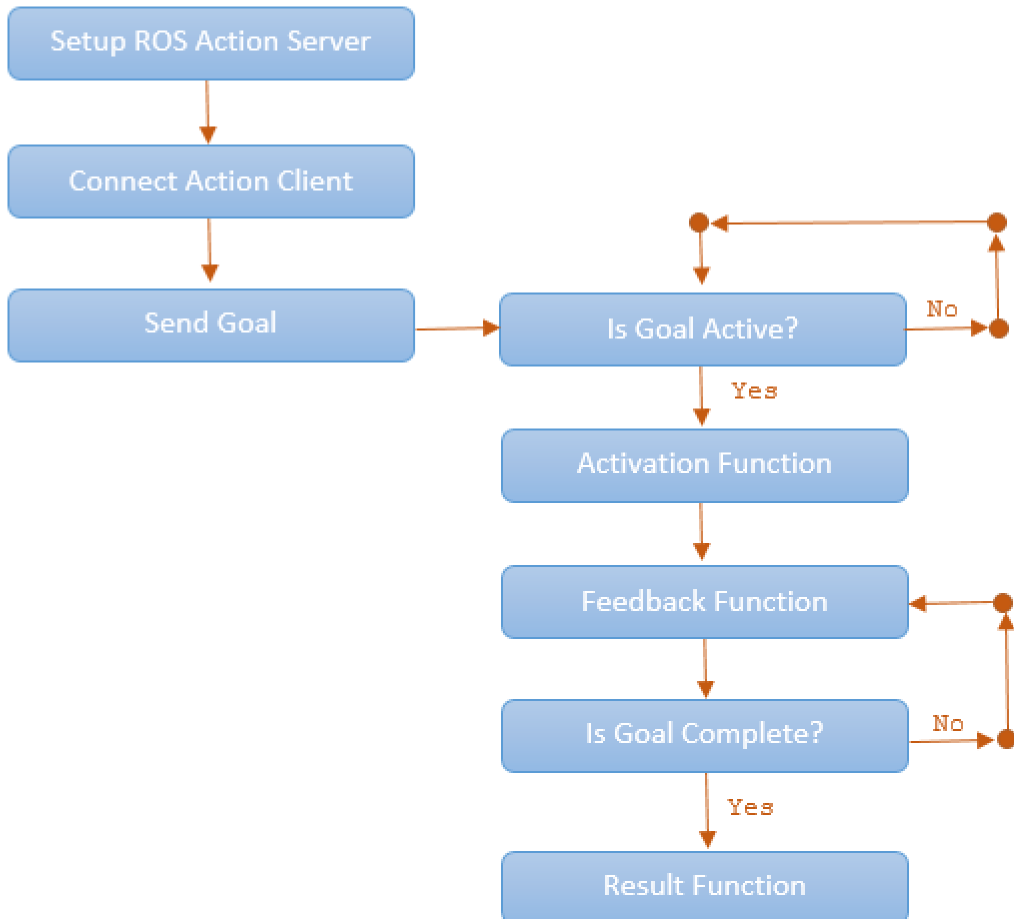
ROS Actions have a client to server communication relationship with a specified protocol. The actions utilize ROS topics to send goal messages from a client to the server. You can cancel goals using the action client. After receiving a goal, the server processes it and can give information back to the client. This information includes the status of the server, the state of the current goal, feedback on that goal during operation, and finally a result message when the goal is complete.



Use the `sendGoal` function to send goals to the server. Send the goal and wait for it to complete using `sendGoalAndWait`. This function allows you to return the result message, final state of the goal and status of the server. While the server is executing a goal, the callback function, `FeedbackFcn`, is called to provide data relevant to that goal (see `SimpleActionClient`). Cancel the current goal using `cancelGoal` or all goals on server using `cancelAllGoals`.

Performing Actions Workflow

In general, the following steps occur when creating and executing a ROS action on a ROS network.



- Setup ROS action server. Check what actions are available on a ROS network by typing `rosaction list` in the MATLAB command window.
- Use `rosactionclient` to create action clients and connect them to the server. Specify an action type currently available on the ROS network. Use `waitForServer` to wait for the action client to connect to the server.
- Send a goal using `sendGoal`. Define a `goalMsg` that corresponds to the action type. When you create an action client using `rosactionclient`, a blank `goalMsg` is returned. You can modify this message with your desired parameters.

- When a goal status becomes `'active'`, the goal begins execution and the `ActivationFcn` callback function is called. For more information on modifying this callback function, see `SimpleActionClient`.
- While the goal status remains `'active'`, the server continues to execute the goal. The feedback callback function processes information about this goals execution periodically whenever a new feedback message is received. Use the `FeedbackFcn` to access or process the message data sent from the ROS server.
- When the goal is achieved, the server returns a result message and status. Use the `ResultFcn` callback to access or process the result message and status.

Action Messages and Functions

ROS actions utilize ROS messages to send goals and receive feedback about their execution. In MATLAB, you can use callback functions to access or process the feedback and result information from these messages. After you create the `SimpleActionClient` object, specify the callback functions by assigning function handles to the properties on the object. You can create the object using `rosactionclient`.

- `GoalMsg` — The goal message contains information about the goal. To perform an action, you must send a goal message with updated goal information (see `sendGoal`). The type of goal message depends on the type of ROS action.
- `ActivationFcn` — Once a goal is received on the action server, its status goes to `'pending'` until the server decides to execute it. The status is then `'active'`. At this moment, MATLAB executes the callback function defined in the `ActivationFcn` property of the `SimpleActionClient` object. There is no ROS message or data associated with this function. By default, this function simply displays `'Goal is active'` on the MATLAB command line to notify you the goal is being executed.

  The default function handle is:

  ```
  @(~) disp('Goal is active')
  ```
- `FeedbackFcn` — The feedback function is used to process the information from the feedback message. The type of feedback message depends on the action type. The feedback function executes periodically during the goal operation whenever a new feedback message is received. By default, the function displays the details of the message using `showdetails`. You can do other processing on the feedback message in the feedback function.

  The default function handle is:

```
@(~,msg) disp(['Feedback: ',showdetails(msg)])
```

`msg` is the feedback message as an input argument to the function you define.

- `ResultFcn` — The result function executes when the goal has been completed. Inputs to this function include both the result message and the status of execution. The type of result message depends on the action type. This message, `msg`, and status, `s`, are the same as the outputs you get when using `sendGoalAndWait`. This function can also be used to trigger dependent processes after a goal is completed.

  The default function handle is:

  ```
  @(~,s,msg) disp(['Result with state ',char(s),': ',showdetails(msg)])
  ```

See Also

`rosaction` | `rosactionclient`

Related Examples

- “Move a Turtlebot Robot Using ROS Actions” on page 3-16

Move a Turtlebot Robot Using ROS Actions

This example shows how to use the `/turtlebot_move` action with a Turtlebot robot. The `/turtlebot_move` action takes a location in the robot environment and attempts to move the robot to that location.

To run the Turtlebot ROS action server, this command is used on the ROS distribution.

```
roslaunch turtlebot_actions server_turtlebot_move.launch
```

Connect to a ROS network. You must have an ROS action server setup on this network. Change `ipaddress` to the address of your ROS network.

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

```
Initializing global node /matlab_global_node_10652 with NodeURI http://192.168.154.1:62
```

View the ROS actions available on the network. You should see `/turtlebot_move` available.

```
rosaction list
```

```
/turtlebot_move
```

Create a simple action client to connect to the action server. Specify the action name. `goalMsg` is the goal message for you to specify goal parameters.

```
[client,goalMsg] = rosactionclient('/turtlebot_move');
waitForServer(client)
```

Set the parameters for the goal. The `goalMsg` contains properties for both the forward and turn distances. Specify how far forward and what angle you would like the robot to turn. This example moves the robot forward 2 meters.

```
goalMsg.ForwardDistance = 2;
goalMsg.TurnDistance = 0;
```

Set the feedback function to empty to have nothing output during the goal execution. Leave `FeedbackFcn` as the default value to get a print out of the feedback information on the goal execution.

```
client.FeedbackFcn = [];
```

Send the goal message to the server. Wait for it to execute and get the result message.

```
[resultMsg,~,~] = sendGoalAndWait(client,goalMsg)
```

```
Goal active

resultMsg =

  ROS TurtlebotMoveResult message with properties:

        MessageType: 'turtlebot_actions/TurtlebotMoveResult'
       TurnDistance: 0
    ForwardDistance: 2.0076

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

Disconnect from the ROS network.

```
rosshutdown
```

```
Shutting down global node /matlab_global_node_10652 with NodeURI http://192.168.154.1:6
```

See Also

rosaction | rosactionclient | sendGoal | sendGoalAndWait

More About

- “ROS Actions Overview” on page 3-11

4

ROS Log Files and Transformations

ROS Log Files (rosbags)

In this section...
"Introduction" on page 4-2
"MATLAB rosbag Structure" on page 4-2
"Workflow for rosbag Selection" on page 4-3
"Limitations" on page 4-5

Introduction

A rosbag or bag is a file format in ROS for storing ROS message data. These bags are often created by subscribing to one or more ROS topics, and storing the received message data in an efficient file structure. MATLAB® can read these rosbag files and help with filtering and extracting message data. The following sections detail the structure of rosbags in MATLAB and the workflow for extracting data from them.

MATLAB rosbag Structure

When accessing rosbag log files, call `rosbag` and specify the file path to the object. MATLAB then creates a `BagSelection` object that contains an index of all the messages from the rosbag.

The `BagSelection` object has the following properties related to the rosbag:

- `FilePath`: a character vector of the absolute path to the rosbag file.
- `StartTime`: a scalar indicating the time the first message was recorded
- `EndTime`: a scalar indicating the time the last message was recorded
- `NumMessages`: a scalar indicating how many messages are contained in the file
- `AvailableTopics`: a list of what topic and message types were recorded in the bag. This is stored as table data that lists the number of messages, message type, and message definition for each topic. For more information on table data types, see "Access Data in a Table" (MATLAB). Here is an example output of this table:

```
ans =

                            NumMessages       MessageType           MessageDefinitio
                            ___________    _____________________    ________________
```

```
    /clock                   12001         rosgraph_msgs/Clock       [1x185  char]
    /gazebo/link_states      11999         gazebo_msgs/LinkStates    [1x1247 char]
    /odom                    11998         nav_msgs/Odometry         [1x2918 char]
    /scan                      965         sensor_msgs/LaserScan     [1x2123 char]
```

- `MessageList`: a list of every message in the bag with rows sorted by time stamp of when the message was recorded. This list can be indexed and you can select a portion of the list this way. Calling `select` allows you to select subsets based on time stamp, topic or message type.

Also, note that the `BagSelection` object contains an index for all the messages. However, you must still use functions to extract the data. For extracting this information, see `readMessages` for getting messages based on indices as a cell array or see `timeseries` for reading the data of specified properties as a time series.

Workflow for rosbag Selection

When working with rosbags, there is a general procedure of how you should extract data.

- **Load a rosbag**: Call `rosbag` and the file path to load file and create `BagSelection`.
- **Examine available messages**: Examine `BagSelection` properties (`AvailableTopics`, `NumMessages`, `StartTime`, `EndTime`, and `MessageList`) to determine how to select a subset of messages for analysis.
- **Select messages**: Call `select` to create a selection of messages based on your desired properties.
- **Extract message data**: Call `readMessages` or `timeseries` to get message data as either a cell array or time series data structure.
- **Visualize, analyze or process data**: Use the extracted data for your specific application. You can plot data or develop algorithms to process data.

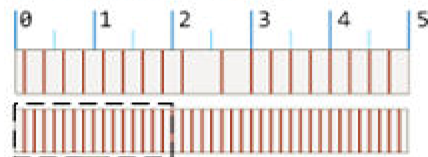
The following figure also shows the workflow.

Load a rosbag

```
>> filePath = fullfile('rosbags','bag1.bag');
>> bagSelect = rosbag(filePath);
```

Examine available messages

Topic Name	Message Type	Time-stamped Messages
/scan	sensor_msgs/LaserScan	
/odom	nav_msgs/Odometry	

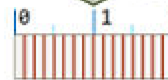


Select messages

Select Topic: /odom
Select Time Interval: [0, 2]

select()

/odom nav_msgs/Odometry



Extract message data

Select Message Indices

readMessages()

Message Object(s)
(nav_msgs/Odometry)
...

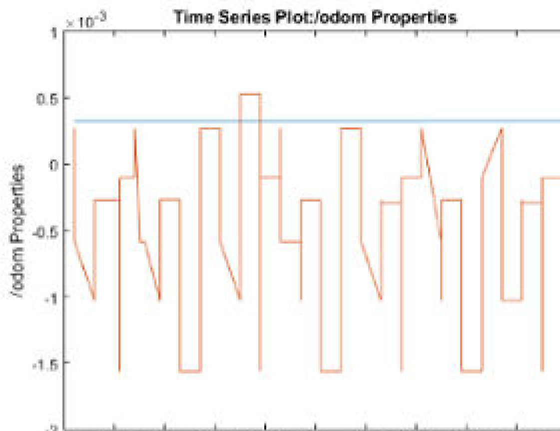
Select Property: Pose.X
Select Property: Pose.Y
Select Property: Orient.Z

timeseries()

Timeseries Object

Time	Pose.X	Pose.Y	Orient.Z
0.125	0.000	1.254	0.000
0.250	0.123	1.254	1.571
...	...	...	...
2.000	0.297	1.254	3.142

Visualize, analyze, or process data



Limitations

There are a few limitations in the rosbag support within MATLAB:

- MATLAB can only parse uncompressed rosbags. See the ROS Wiki for a tool to decompress a compressed rosbag.
- Only rosbags in the v2.0 format are supported. See the ROS Wiki for more information on different bag formats
- The file path to the rosbag must always be accessible. Because the message selection process does not retrieve any data, the file needs to be available for reading when the message data is accessed.

See Also

`BagSelection` | `readMessages` | `rosbag`

Related Examples

- "Work with rosbag Logfiles"

ROS Custom Message Support

- “Create Custom Messages from ROS Package” on page 5-2
- “ROS Custom Message Support” on page 5-8
- “Install Robotics System Toolbox Add-ons” on page 5-14

Create Custom Messages from ROS Package

In this example, you go through the procedure for creating ROS custom messages in MATLAB. It assumes you have already gone through the installation process shown in "Install Robotics System Toolbox Add-ons" on page 5-14. Also, you must have a ROS package that contains the required `msg`, `srv`, and `package.xml` files. The correct file contents and folder structure are described in "Custom Message Contents" on page 5-8. This folder structure follows the standard ROS package conventions. Therefore, if you have any existing packages, they should match this structure.

It is recommended you start this procedure after opening a new MATLAB session to ensure that there are no lingering changes to MATLAB preferences from previous work. After ensuring that your custom message package is correct, note the folder path location. Then, call `rosgenmsg` with the specified path and follow the steps output in the command window. The following example has three messages, `A`, `B`, and `C`, that have dependencies on each other. This example also illustrates that you can use a folder containing multiple messages and generate them all at the same time.

To set up custom messages in MATLAB:

- Open MATLAB in a new session
- Place your custom message folder in a location and note the folder path. In this example, a location of example packages is provided and copied to `userFolder` location. Make sure that the `userFolder` directory exists prior to running this code.

  ```
  examplePackages = fullfile(fileparts(which('rosgenmsg')), 'examples', 'packages');
  userFolder = 'c:\MATLAB\custom_msgs';
  copyfile(examplePackages, userFolder)
  ```
- Specify the folder path of the custom messages.

  ```
  folderpath = userFolder;
  ```

  (Optional) If you have an existing catkin workspace (`catkin_ws`), you can specify the path to its `src` folder instead. However, this workspace might contain a large number of packages and message generation will be run for all of them.

  ```
  folderpath = fullfile('catkin_ws','src');
  ```
- Specify the folder path for custom message files and call `rosgenmsg` to create custom messages for MATLAB.

  ```
  rosgenmsg(folderpath)
  ```

```
Checking subfolder "A" for custom messages.

Checking subfolder "B" for custom messages.

Checking subfolder "C" for custom messages.

Building custom message files for the following packages:
   A
   B
   C


Generating MATLAB classes for message packages in
C:\MATLAB\custom_msgs\matlab_gen\jar

Loading file A-1.0.jar.
Generating MATLAB code for A/DependsOnB message type.
Generating MATLAB code for B/Standalone message type.

Loading file B-1.0.jar.

Loading file C-1.0.jar.
Generating MATLAB code for C/DependsOnB message type.

To use the custom messages, follow these steps:

1. Edit javaclasspath.txt, add the following file locations as new lines, and
save the file:

C:\MATLAB\custom_msgs\matlab_gen\jar\A-1.0.jar
C:\MATLAB\custom_msgs\matlab_gen\jar\B-1.0.jar
C:\MATLAB\custom_msgs\matlab_gen\jar\C-1.0.jar

2. Add the custom message folder to the MATLAB path by executing:

addpath('C:\MATLAB\custom_msgs\matlab_gen\msggen')
savepath

3. Restart MATLAB and verify that you can use the custom messages.
   Type "rosmsg list" and ensure that the output contains the generated
   custom message types.
```

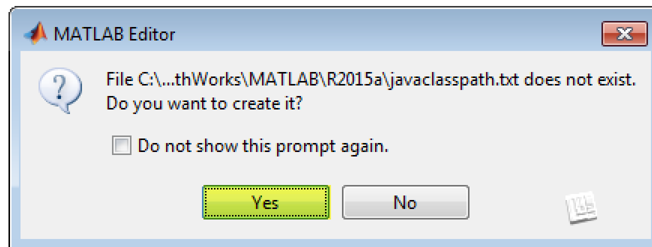
Tip If you see the following warning

```
Objects of *** class exist - not clearing java
```

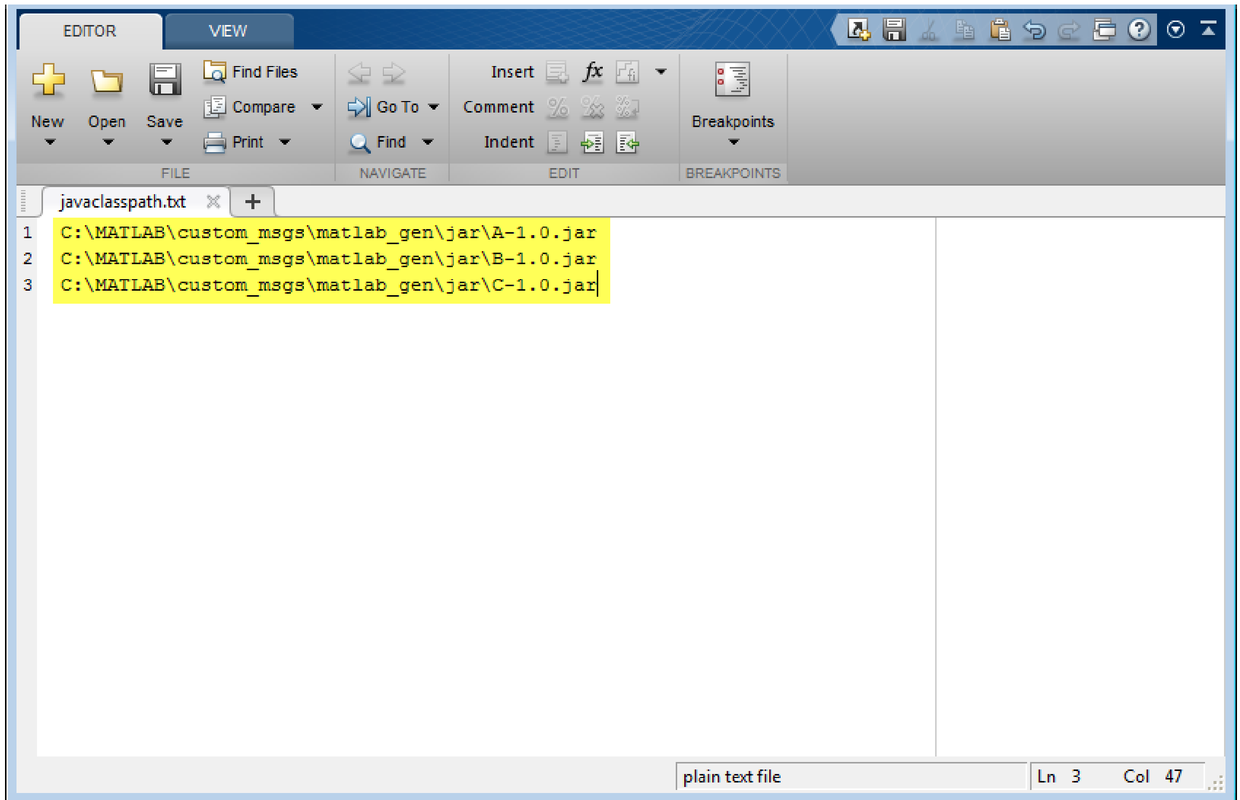
Try either calling `rosgenmsg` at the beginning of your MATLAB session or make sure that no Java objects are created with any startup functions called.

- Then, follow steps 1–3 from the output of `rosgenmsg`.

1. Click the `javaclasspath.txt` link to open the file in the Editor. Copy and paste the different jar file locations as new lines in the file. If this file does not exist, you will be prompted to create it. Click **Yes** and then copy and paste the file locations into the file.



The `javaclasspath.txt` looks like this after adding lines. Other paths may also already exist in this file.



2 Add the given files to the MATLAB path by running `addpath` and `savepath` in the command window. You can either highlight the commands shown and press **F9** or copy and paste it into the MATLAB Command Window.

```
addpath('C:\MATLAB\custom_msgs\matlab_gen\msggen')
savepath
```

3 Restart MATLAB for the path changes to be applied. You can then use the custom messages like any other ROS messages supported in Robotics System Toolbox. Verify these changes by either calling `rosmsg list` and search for your message types, or use `rosmessage` to create a new message.

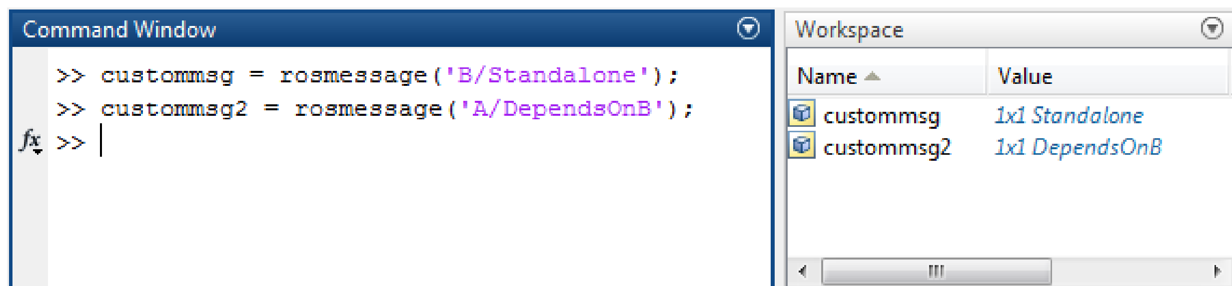
```
custommsg = rosmessage('B/Standalone')
```

```
custommsg = 

  ROS Standalone message with properties:
```

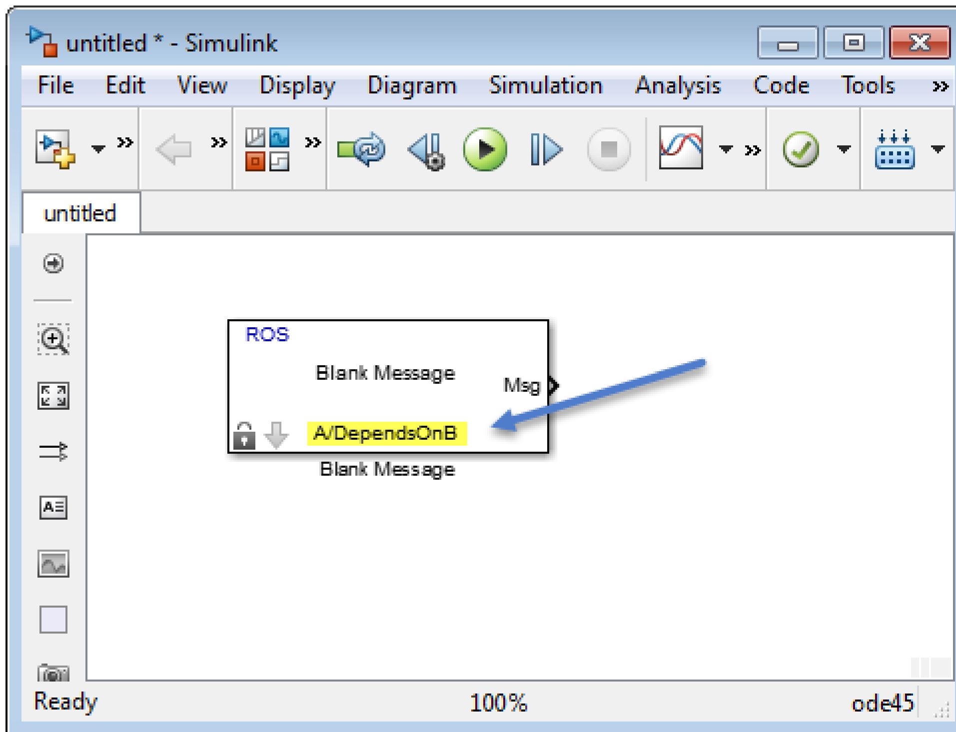
```
      MessageType: 'B/Standalone'
      IntProperty: 0
    StringPropert: ''

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

This final verification shows that you have performed the custom message generation process correctly. You can now send and receive these messages over a ROS network using MATLAB and Simulink®. The new custom messages can be used like normal message types. You should see them create objects specific to their message type and be displayed in your workspace.

```
custommsg = rosmessage('B/Standalone');
custommsg2 = rosmessage('A/DependsOnB');
```



Custom messages can also be used with the ROS Simulink blocks.



See Also

`roboticsAddons` | `rosgenmsg`

Related Examples

- "Install Robotics System Toolbox Add-ons" on page 5-14
- "ROS Custom Message Support" on page 5-8

ROS Custom Message Support

In this section...
"Custom Message Overview" on page 5-8
"Custom Message Contents" on page 5-8
"Custom Message Creation Workflow" on page 5-11

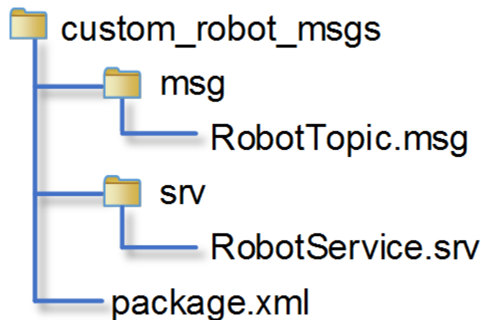
Custom Message Overview

Custom messages are user-defined messages that you can use to extend the set of message types currently supported in Robotics System Toolbox. If you are sending and receiving supported message types, you do not need to use custom messages. To see a list of supported message types, call `rosmsg list` in the MATLAB Command Window.

To install custom message support, call `roboticsAddons` and follow the instructions for installation. Custom message creation requires ROS packages, which are detailed in the ROS Wiki at Packages. After ensuring that you have valid ROS packages for custom messages, call `rosgenmsg` to generate the necessary MATLAB code to use custom messages. For an example on how to generate a ROS custom message in MATLAB, see"Create Custom Messages from ROS Package" on page 5-2.

Custom Message Contents

ROS custom messages are specified in ROS package folders that contain a `package.xml` file and optional `msg` and `srv` directories. The `msg` folder contains all your custom message type definitions. You should also add all custom service type definitions to the `srv` folder. For example, the package `custom_robot_msgs` has this folder and file structure.



The package contains one custom message type in `RobotTopic.msg` and one custom service type in `RobotService.srv`. MATLAB uses these files to generate the necessary files for using the custom messages contained in the package. For more information on creating `msg` and `srv` files, see Creating a ROS msg and srv and Defining Custom Messages on the ROS Wiki. The syntax of these files is described on the pages specific to `msg` and `srv`.

In all packages, you must define a `package.xml` file, which has the following contents:

- Name — `custom_robot_msgs`
- Version — `1.1.01`
- Dependency — `message_generation`
- Other dependencies on message packages (optional) — `geometry_msgs`, `std_msgs`

Here is a sample `package.xml` file with the previously shown contents.

```
<package>
    <name>custom_robot_msgs</name>
    <version>1.1.01</version>

    <build_depend>message_generation</build_depend>
    <build_depend>geometry_msgs</build_depend>
    <build_depend>std_msgs</build_depend>
</package>
```

Note

- You must have write access to the custom messages folder.

- At any time, there should only be one custom messages folder on the MATLAB path. This folder can contain multiple packages, but it is recommended that you keep them all in one unique folder.
- ROS actions are not currently supported and will be ignored during the custom message generation.
- ROS packages will not be processed if both of these conditions are met:
  - A package with the same name already exists
  - The version number of that existing package is the same

Property Naming From Message Fields

When ROS message definitions are converted to MATLAB, the field names are converted to properties for the message object. Object properties always begin with a capital letter and do not contain underscores. The field names are modified to fit this naming convention. The first letter and the first letter after underscores are capitalized with underscores removed. For example, the `sensor_msgs/Image` message has these fields in ROS:

```
header
height
width
encoding
is_bigendian
step
data
```

The converted MATLAB properties are:

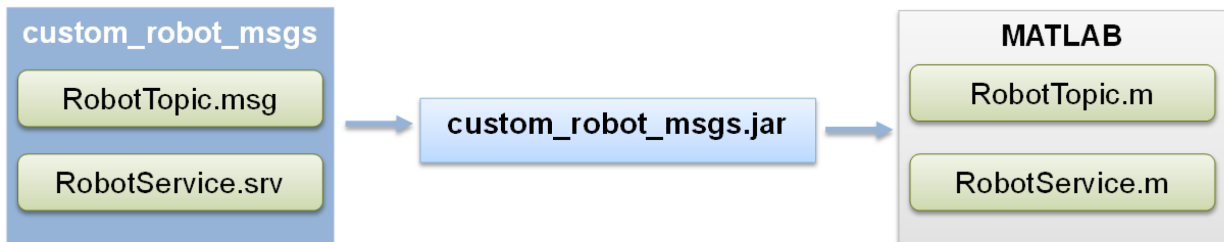
```
Header
Height
Width
Encoding
IsBigendian
Step
Data
```

This is also reflected when using ROS messages in Simulink. ROS message buses use the same properties names as MATLAB.

Custom Message Creation Workflow

Once you have your custom message structure set up as described in the previous section, you can create the code needed to use these custom messages. First, you call `rosgenmsg` with your known path to the custom message files to create MATLAB code.

Then, the two main creation steps that are handled by the `rosgenmsg` function. This function takes your custom message files (`.msg`, `.srv` and `package.xml`) and converts each message type to working MATLAB code. The `rosgenmsg` function will look for `.msg` files in the `msg` folder and for `.srv` files in the `srv` folder. This code is a group of classes that define the message properties when you create new custom messages. The basic procedure takes the custom message files and converts them to `.jar` files and then creates MATLAB program for each topic and service. Do not modify the `.jar` files because MATLAB uses them internally.



After the `rosgenmsg` function creates these files, you must add the files to the Java class path and the MATLAB path before you can use the custom messages. These steps are given as prompts in the MATLAB Command Window:

1. **Add location of files to `javaclasspath.txt`**: Add the specified paths as new lines of text in the `javaclasspath.txt` file. If this file does not exist, a message in the command window prompts you to create it. This text file defines the static class path for Java classes. For more information on the Java class path, see “Java Class Path” (MATLAB).
2. **Add location of class files to MATLAB path**: Use `addpath` to add new locations of files with the .m extension to the MATLAB path and use `savepath` to save these changes.
3. **Restart MATLAB and verify messages are available**: After restarting MATLAB, call `rosmsg list` or `rosmessage` to check that you can use the messages as expected.

For an example of this procedure, see “Create Custom Messages from ROS Package” on page 5-2. This example uses sample custom message files to create custom messages in MATLAB.

You need to complete this procedure only once for a specific set of custom messages. After that, you can use the new custom messages like any other ROS message in MATLAB and take advantage of the full ROS functionality that Robotics System Toolbox provides. Repeat this generation procedure when you would like to update or create new message types.

You must maintain the Java class path and MATLAB path that contain the files directories. Make sure that the MATLAB path has only one folder at a time that contains custom message artifacts. Also, ensure you add the correct paths to the `javaclasspath.txt`, as the prompt directs. Do not modify the path. This file is used to load Java files at the start of each MATLAB session.

Sharing Custom Messages

After creating your custom message files, you can share them with other users. Other people do not need to call `rosgenmsg` to access your messages. Instead, to share your messages, access the `_matlab_gen` folder and follow the same three steps for specifying paths as described previously. If you have access to these files, either over a network or shared drive, add the `matlab_gen/jar` folder path to the `javaclasspath.txt` file and the `matlab_gen/msggen` path to the MATLAB path. After restarting MATLAB, other users can use the custom messages like any other ROS message.

Code Generation with Custom messages

Custom message and service types can be used with ROS Simulink blocks for generating C++ code for a standalone ROS node. The generated code (`.tgz` archive) will include Simulink definitions for the custom messages, but it will not include the ROS custom message packages. When the generated code is built in the destination Linux System, it expects the custom message packages to be available in the catkin workspace or on the `ROS_PACKAGE_PATH`. Please ensure that you either install or copy the custom message package to your Linux system before building the generated code.

See Also

`roboticsAddons` | `rosgenmsg`

Related Examples

- “Install Robotics System Toolbox Add-ons” on page 5-14
- “Create Custom Messages from ROS Package” on page 5-2

Install Robotics System Toolbox Add-ons

To expand the capabilities of the Robotics System Toolbox and gain additional functionality for specific tasks and applications, use add-ons. You can find and install add-ons using the Add-On Explorer.

1. To install add-ons relevant to the Robotics System Toolbox, type in the MATLAB command window:

   ```
   roboticsAddons
   ```

2. Select the add-on that you want. For example:

   - **Robotics System Toolbox Interface for ROS Custom Messages**

3. Click **Install**, and select either:

   - **Install**
   - **Download Only...** — Downloads an install file to use offline.

4. Continue to follow the setup instructions on the **Add-Ons Explorer** to install your add-ons.

To update or manage your add-ons, call `roboticsAddons` and select **Manage Add-Ons**.

See Also

Related Examples

- “Create Custom Messages from ROS Package” on page 5-2
- “ROS Custom Message Support” on page 5-8
- “Add-Ons” (MATLAB)

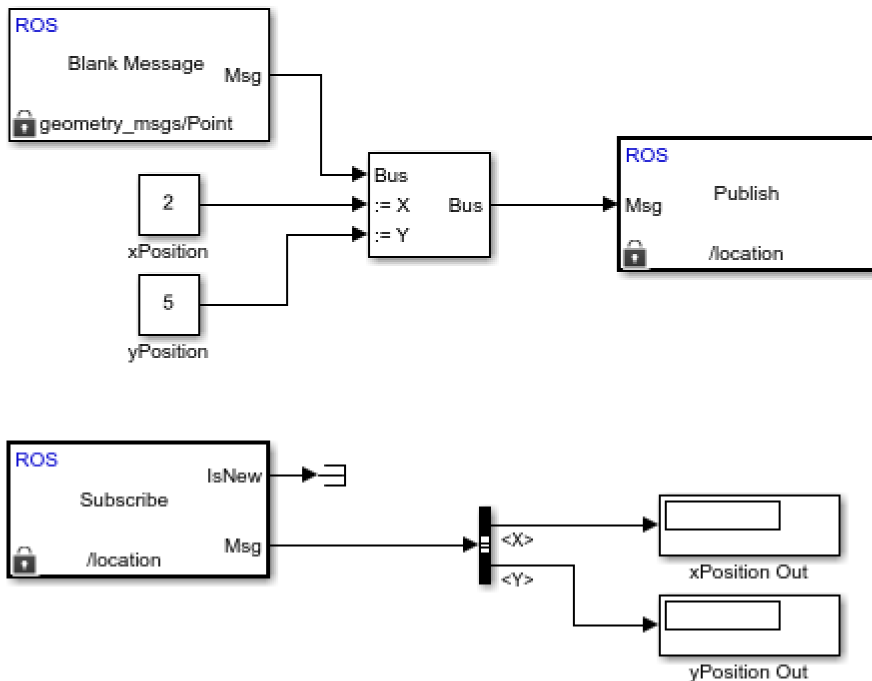
Simulink ROS Concepts

- “Publish and Subscribe to ROS Messages in Simulink” on page 6-2
- “Selecting ROS Topics, Messages, and Parameters” on page 6-5
- “Configure ROS Network Addresses” on page 6-9
- “Managing Array Sizes in Simulink ROS” on page 6-13
- “Connect to ROS Device” on page 6-15
- “Simulink and ROS Interaction” on page 6-16
- “ROS Parameters in Simulink” on page 6-18
- “ROS String Parameters” on page 6-20
- “Enable ROS Time Model Stepping for Deployed ROS Nodes” on page 6-23
- “Overrun Detection with Deployed ROS Nodes” on page 6-25
- “ROS Simulink Support and Limitations” on page 6-27
- “Read A ROS Image Message In Simulink®” on page 6-28
- “Read A ROS Point Cloud Message In Simulink®” on page 6-32
- “Convert Coordinate System Transformations” on page 6-37
- “Call ROS Service in Simulink” on page 6-38
- “Play Back Data from Jackal™ rosbag Logfile in Simulink” on page 6-40
- “Time Stamp A ROS Message Using Current Time in Simulink” on page 6-42

Publish and Subscribe to ROS Messages in Simulink

This model shows how to publish and subscribe to a ROS topic using Simulink®.

```
open_system('simulinkPubSubExample.slx')
```



Use the Blank Message and Bus Assignment blocks to specify the X and Y values of a `'geometry_msgs/Point'` message type. Open the Blank Message block mask to specify the message type. Open the Bus Assignment block mask to select the signals you want to assign. Remove any values with `'???'` from the right column. Supply the Bus Assignment block with relevant values for X and Y.

Feed the `Bus` output to the Publish block. Open the block mask and choose `Specify your own` as the topic source. Specify the topic, `'/location'`, and message type, `'geoemetry_msgs/Point'`.

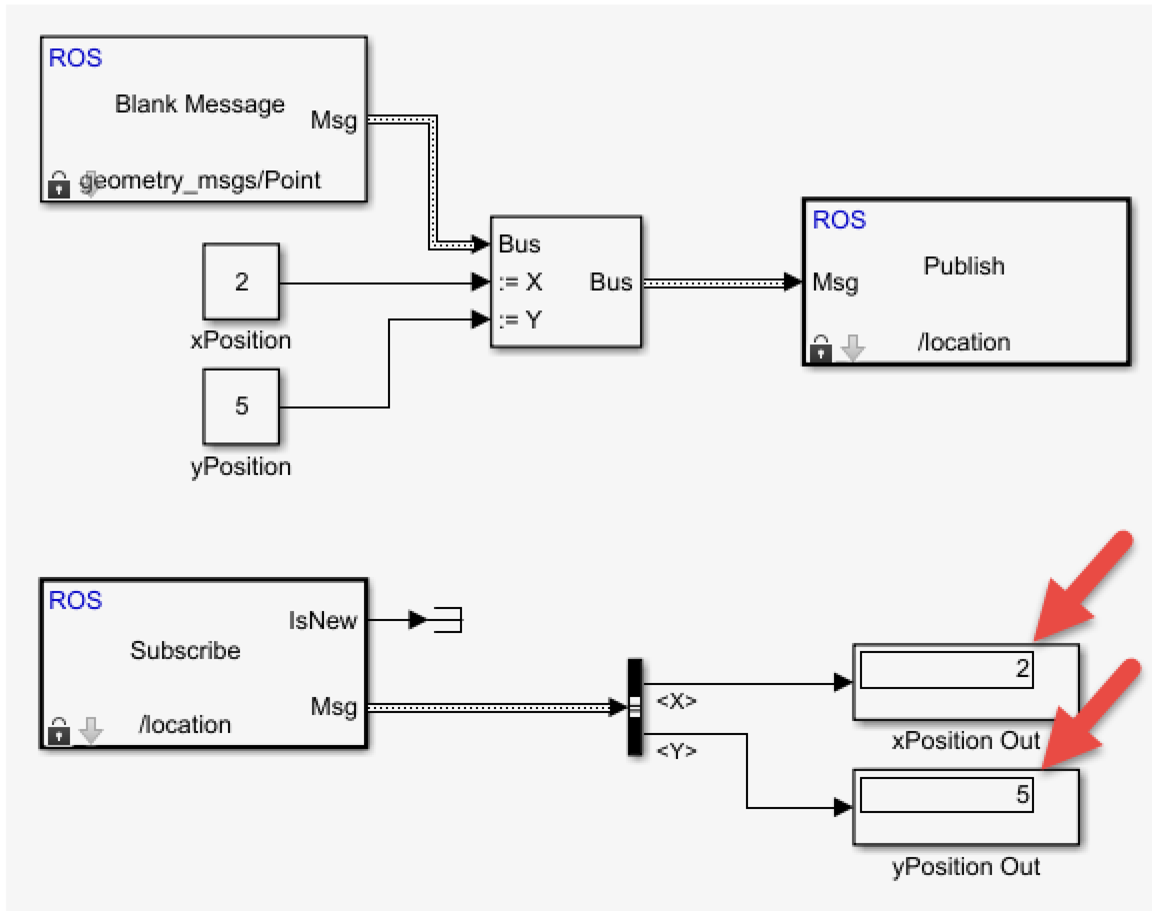
Add a Subscribe block and specify the topic and message type. Feed the output `Msg` to a Bus Selector and specify the selected signals in the block mask. Display the `X` and `Y` values.

Before running the model, call `rosinit` to connect to a ROS network.

```
rosinit
```

```
Initializing ROS master on http://bat5838win64:58946/.
Initializing global node /matlab_global_node_36800 with NodeURI http://bat5838win64:589
```

Set the simulation stop time to `Inf` and run the model. You should see the `xPosition Out` and `yPosition Out` displays show the corresponding values published to the ROS network.



Shut down the ROS network.

```
rosshutdown
```

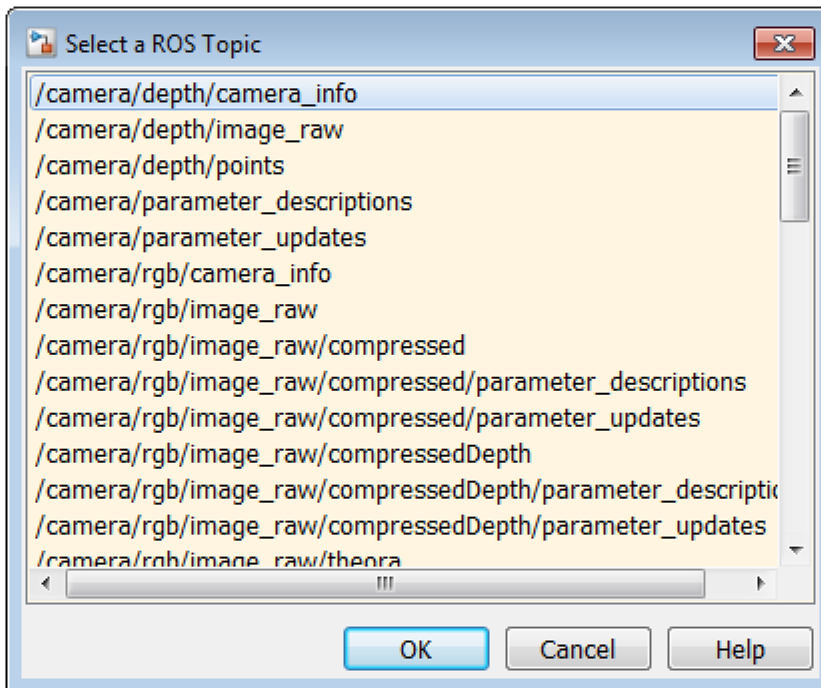
```
Shutting down global node /matlab_global_node_36800 with NodeURI http://bat5838win64:5
Shutting down ROS master on http://bat5838win64:58946/.
```

Selecting ROS Topics, Messages, and Parameters

In this section...
"Selecting ROS Topics" on page 6-5
"Selecting ROS Message Types" on page 6-6
"Selecting ROS Parameter Names" on page 6-7

Selecting ROS Topics

When using Simulink with ROS, you can publish or subscribe to topics on the ROS network. In the dialog boxes for the Publish and Subscribe blocks, you can select from a list of topics on the ROS network. You must be currently connected to a ROS network to get a list of topics. You can select a topic using the following:



This dialog shows the list of topics available on the ROS master. Selecting a topic from the list automatically populates the **Topic** and **Message type** parameters for the

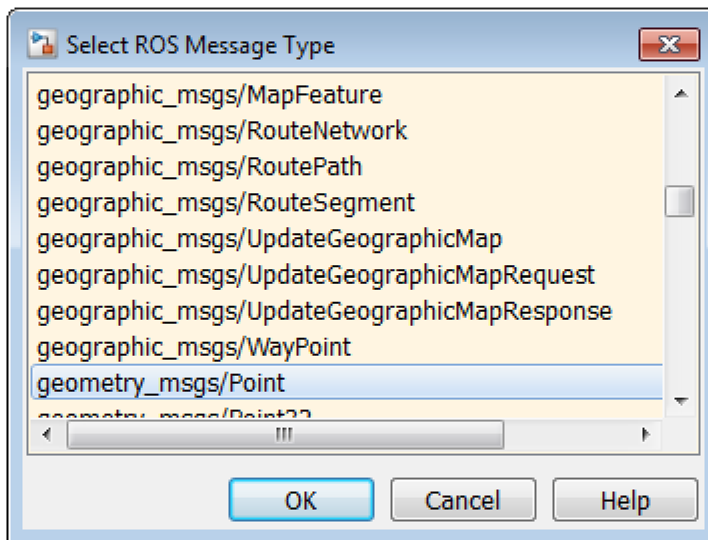
corresponding block mask dialog. If the message type is not supported in MATLAB ROS, Simulink will throw an error. Once the topic is selected, it is saved with the block. Even if the topic is not longer available on the network, the block will still use that topic name.

To refresh the list, close and open the dialog again.

To use a topic not currently posted on the ROS network or if you are not currently connected, use the “Specify your own” option under the **Topic Source** parameter in your block mask dialog.

Selecting ROS Message Types

Simulink ROS allows you to select from a list of message types currently supported by MATLAB ROS when setting the **Message type** for Publish, Subscribe, or BlankMessage blocks.



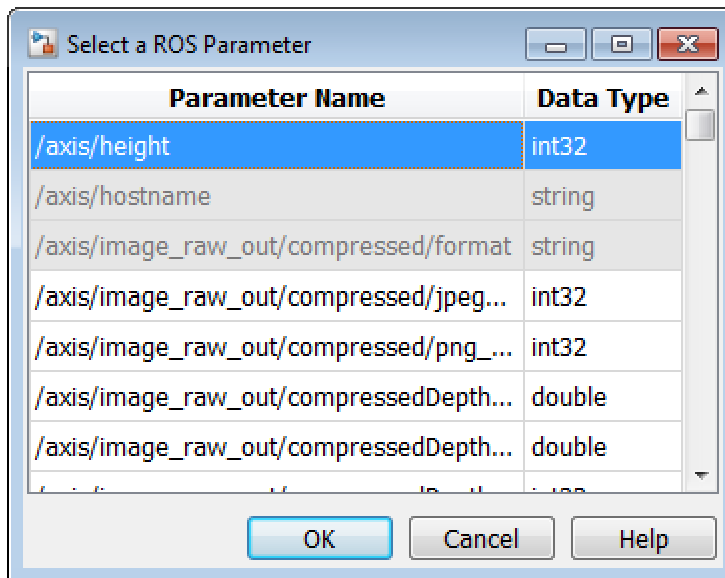
This is the list of all message types supported in MATLAB ROS including any custom message types. You can begin typing in the name of your desired message type or manually search through the list.

The selected message type is stored with the block and saved with the model.

Note: When using code generation, message type information is not included. You must ensure that your Linux ROS environment has the ROS packages installed that contain the necessary message type definitions.

Selecting ROS Parameter Names

When using the Get Parameter and Set Parameter blocks, you have the option of "Select from ROS Network" in the block parameters, which gets a list of parameters currently on the server. When clicking **Select**, you should see this dialog box.



This is the list of parameters you can select from the ROS parameter server. The parameters that are grayed out have unsupported data types. Select a parameter name that is not grayed out and click **OK**. This should auto-fill the **Name** and **Data type** into the block parameters.

See Also

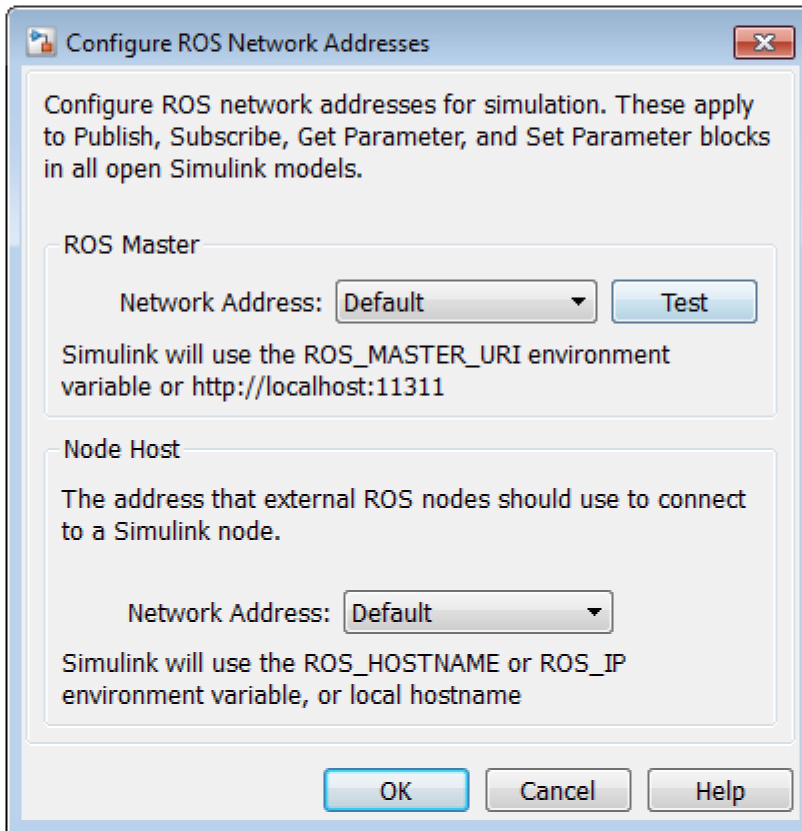
Blank Message | Get Parameter | Publish | Set Parameter | Subscribe

Related Examples

- “ROS Parameters in Simulink” on page 6-18
- “Managing Array Sizes in Simulink ROS” on page 6-13

Configure ROS Network Addresses

During model initialization, Simulink connects to a ROS master and also creates a node associated with the model. The ROS master URI and Node Host are specified in the "Configure ROS Network Addresses" dialog. You can access this in the menu under *Tools>>Robot Operating System (ROS)* by selecting "Configure ROS Network Addresses".



The **Network Address** parameter can be set to "Default" or "Custom".

For the ROS master URI, if **Network Address** is set to "Default", Simulink uses the following rules to set the ROS Master URI:

- Use `ROS_MASTER_URI` environment variable if it is set.
- If a MATLAB global ROS node exists, use the Master URI associated with the global node. The global node is created automatically when `rosinit` is called.
- Use address `http://localhost:11311` if other two rules do not apply.

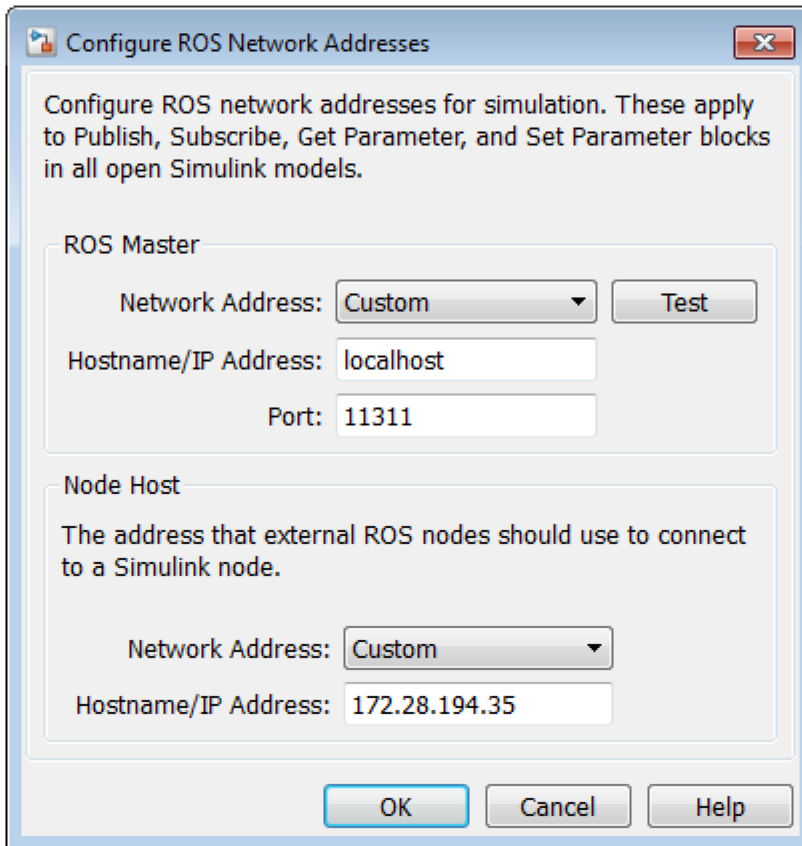
For the Node Host, if **Network Address** is set to “Default”, Simulink uses the following rules to set the ROS Node Host:

- Use `ROS_HOSTNAME` environment variable if it is set.
- Use `ROS_IP` environment variable if it is set.
- Use hostname or IP address of the first network interface on the system if available.
- Use address `http://localhost:11311` if other rules do not apply.

For both, these are the same rules that MATLAB uses to resolve its ROS network addresses.

Otherwise, if you chose “Custom”, you can set all the variables as shown below. This overrides the environment variables.

Note: These addresses are saved in MATLAB preferences, not the model. Therefore, this information is shared across all Simulink models and multiple MATLAB installs of the same release.



You can also use the “Test” button to ensure you can connect to the ROS master. If you get an error, call `rosinit` to setup a local ROS network, or if you specified a remote ROS master, check your settings are correct.

The custom ROS master or node host settings are not used in generated code when deploying a standalone node.

See Also

`rosinit`

Related Examples

- “Get Started with ROS in Simulink®”
- “Connect to a ROS-enabled Robot from Simulink®”

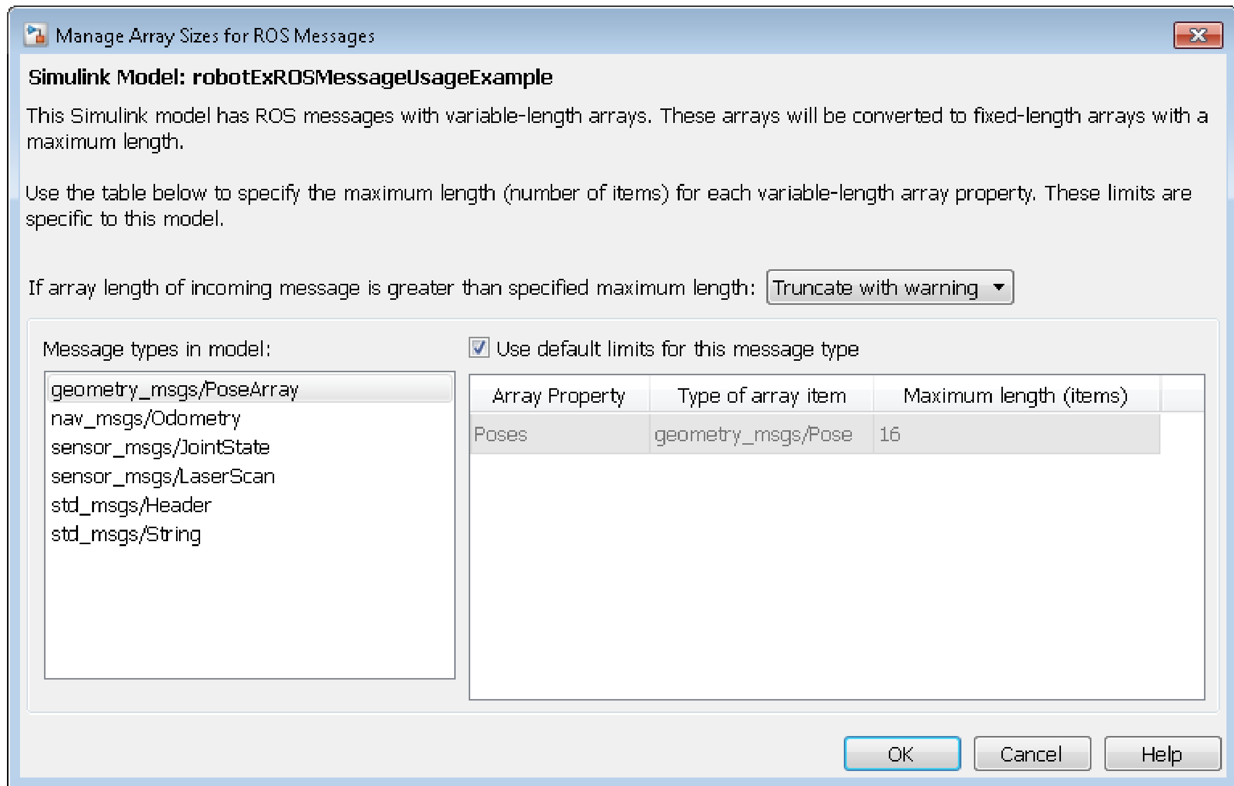
More About

- “Simulink and ROS Interaction” on page 6-16
- “Selecting ROS Topics, Messages, and Parameters” on page 6-5
- “ROS Simulink Support and Limitations” on page 6-27

Managing Array Sizes in Simulink ROS

A ROS message is represented as a bus signal. For more information on bus signals, see "Buses" (Simulink).

If you are working with variable-length signals in Simulink, the non-virtual bus used for messages cannot contain variable-length arrays as properties. All variable-length arrays are converted to fixed-length arrays for non-virtual buses. Therefore, you must manage the maximum size for these fixed-size arrays. To manage array sizes, select *Tools>>Robot Operating System* in the menu and select "Manage Array Sizes". If your model uses ROS messages with variable-length arrays, the following dialog box opens. Otherwise, Simulink displays a message.



Because the message properties have a variable length, it is possible that they can be truncated if they exceed the maximum size set for that array. You have the option of

`Truncate with warning` or `Truncate silently`. Either way, the simulation will run, but `Truncate with warning` displays a warning in the Diagnostic Viewer that the message property has been truncated. When using generated code, the warning will be emitted using Log Statements in ROS. The warning will be a `ROS_WARN_NAMED` log statement and the *name* is the model name.

The **Message types in model** section shows all the ROS message types that are currently used by Publish, Subscribe and Blank Message blocks in your Simulink model. You have the option to use the default limits for this message type by clicking the check box. Otherwise, select each message type individually to set the **Maximum length (items)** of each **Array Property** as desired. This maximum length is applied to all instances of that message type for that model. The maximum length is also stored with the model. Therefore, it is possible to have two models accessing the same message type with different maximum length limits.

Managing the size of your variable-length arrays can help improve performance. If you limit the size of the array to only include relevant data, you can process data more effectively. However, when running these models, consider possible issues associated with truncation and what could happen to your system if some data is ignored.

Note: If you would like to know the appropriate maximum lengths for different message types. You can simulate the model and observe the sizes output in the warning. To see an example of using ROS messages and working with variable-length arrays, see “Work with ROS Messages in Simulink®”.

See Also

Publish | Subscribe

Related Examples

- “Work with ROS Messages in Simulink®”
- “ROS Simulink Support and Limitations” on page 6-27

Connect to ROS Device

When connecting to a ROS device, deploying a ROS node to a ROS device, or trying to start and stop nodes on a ROS device, you must specify the login credentials. The `Connect to a ROS Device` dialog requests the following information to connect to the ROS device.

- **Device Address** — Specify the host name or IP address for the target ROS device.
- **Username** — Specify the user name that is used to log into the target ROS device.
- **Password** — Specify the password that is used to log into the target ROS device with the specified user name.
- **Remember my password** — Select this parameter for your password to be saved for all MATLAB sessions. If this parameter is not selected, MATLAB prompts for your password whenever a connection to the ROS device is established.
- **ROS folder** — Specify the location of the ROS installation folder on the ROS device. For example: `/opt/ros/indigo`
- **Catkin workspace** — Specify the location of the Catkin workspace folder on the ROS device. For example: `~/catkin_ws_test`

By clicking the **Test** button, you can verify your settings. The results of the test are displayed in the Simulink Diagnostic Viewer. Use the Diagnostic Viewer to troubleshoot any issues with connecting to your ROS device. For more information, see “View Diagnostics” (Simulink).

See Also

Related Examples

- “Generate a Standalone ROS Node from Simulink®”

Simulink and ROS Interaction

In this section...

“MATLAB ROS Information” on page 6-16
“Simulink ROS Node” on page 6-16
“Differences Between Simulation and Generated Code” on page 6-17
“Publishers and Subscribers in Simulink” on page 6-17

When using Simulink to communicate with a ROS network or work with ROS functionality, there are several points to note regarding its interaction with MATLAB and the ROS network.

MATLAB ROS Information

Simulink uses the functionality built into MATLAB to communicate with the ROS network during simulation. When trying to debug issues in Simulink, you can use MATLAB to view topics or messages available on the ROS master. For more information on ROS topics and messages, see `rosnode`, `rostopic`, or `rosmsg`.

By default, Simulink uses MATLAB ROS capabilities to resolve network information such as the address of the ROS master. This network information can also be specified in Simulink using the “Configure ROS Network Addresses” on page 6-9 dialog.

Simulink ROS Node

Each model is associated with a unique ROS node. At the start of each simulation, Simulink creates the node and deletes it when the simulation is terminated. If multiple models are open and being simulated, each model will get its own dedicated node, but all the nodes will connect to the same ROS master. This is because all the models use the same ROS network address settings.

In simulation, the Simulink ROS node name is `<modelName>_<random#>`. This takes the model name and adds a random number to the end to avoid node name conflicts.

In generated code, the node name is `<modelName>` (casing preserved). The model name is also used in the archive used for generated code. Do **not** rename the tgz file from code generation (e.g. ModelName.tgz). The file name is used to get the ROS package name and initiate the build.

Differences Between Simulation and Generated Code

In simulation, the model execution does not match real elapsed time. The blocks in the model are evaluated in a loop that only simulates the progression of time, and whose speed depends on complexity of the model and computer speed. It is not intended to track actual clock time.

In generated code, the model execution attempts to match actual elapsed time (the Fixed-step size defines the actual time step, in seconds, that is used for the model update loop). However, this does not guarantee real-time performance, as it is dependent on other processes running on the Linux system and the complexity of the model. If the deployed model is too slow to meet the execution frequency, tasks are dropped. This drop is called an "overrun" and the model waits for the next scheduled task. For more information, see the *Tasking Mode* section in the “Generate a Standalone ROS Node from Simulink®” example.

You can also modify how your generated code runs for a deployed ROS node using `rosdevice`. The `rosdevice` object allows you to connect to a ROS device, run nodes that are deployed, and modify files on the device.

Publishers and Subscribers in Simulink

All publishers and subscribers created using Publish and Subscribe blocks will connect with the ROS node for that model. They are created during the model initialization and topic names are resolved at the same time. The publishers and subscribers are deleted when the simulation is terminated.

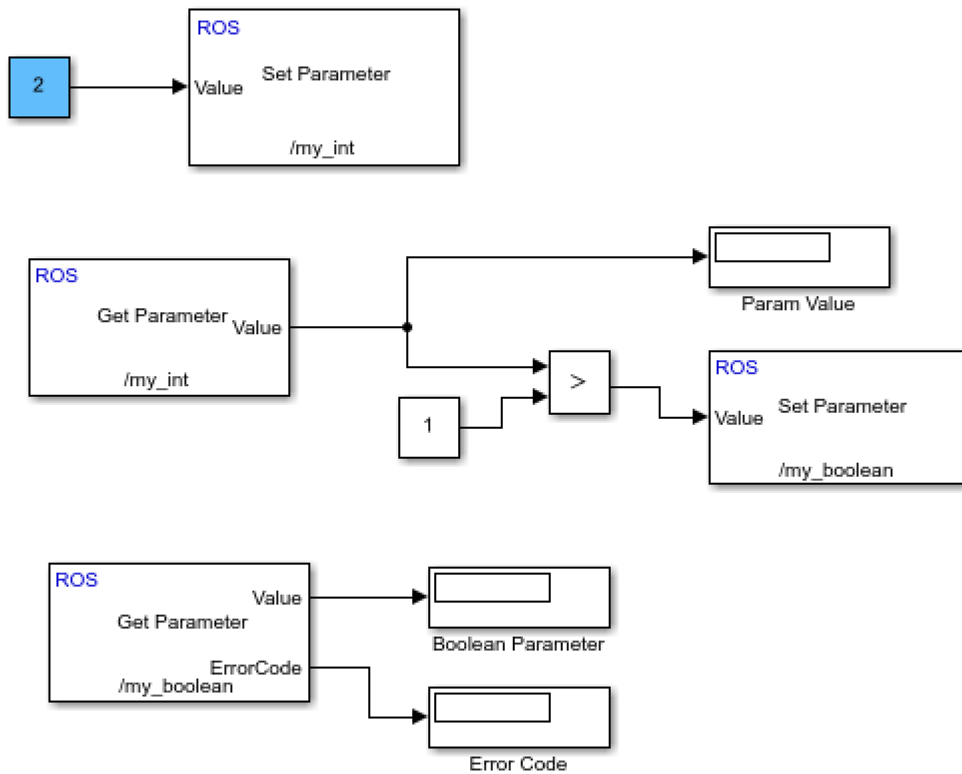
NOTE: If a custom topic name is specified for a Subscribe block, the topic is not required to exist when the model is initialized. The Subscribe block will output blank messages until it receives a message on the topic name you specify. This allows you to setup and test models before the rest of the network has been setup.

ROS Parameters in Simulink

Get and Set ROS Parameters

This model gets and sets ROS parameters using Simulink®. This example illustrates how to use ROS parameters in Simulink and to share data over the ROS network. An integer value is set as a parameter on the ROS network. This integer is retrieved from the parameter server and compared to a constant. The output Boolean from the comparison is also set on the network. Change the constant block in the top left (blue) when you run the model to set network parameters based on user input conditions.

You must be connected to a ROS network. Call `rosinit` in the MATLAB® command line.



See Also

Get Parameter | Set Parameter

More About

- “ROS String Parameters” on page 6-20

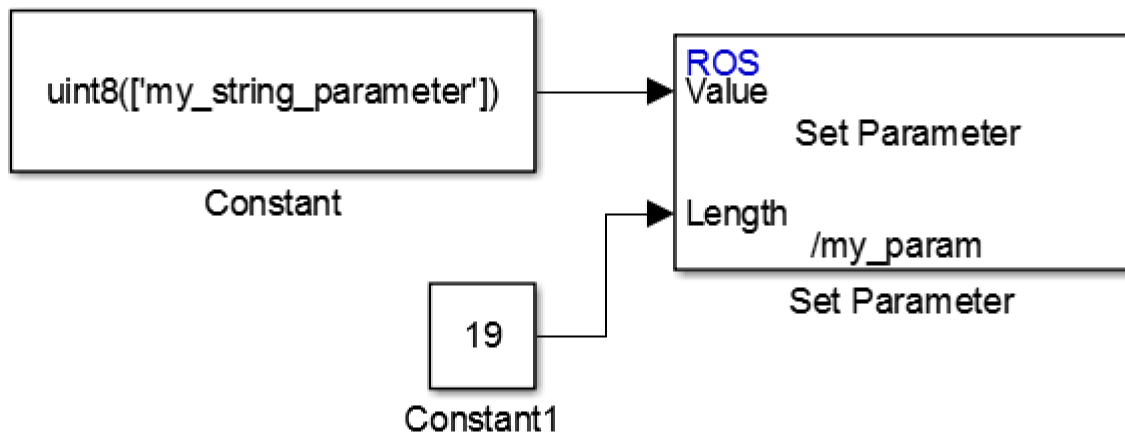
ROS String Parameters

In this section...
“Set String Parameter on ROS Network” on page 6-20
“Get ROS String Parameter and Compare to Specified String” on page 6-21
“Check Image Encoding Parameter for ROS Image Message” on page 6-21

To use ROS string parameters in Simulink, cast them to `uint8` arrays. These examples show how to get, set, compare, and manipulate strings for ROS parameters. To run these examples, you must first set up a ROS network. Use `rosinit`.

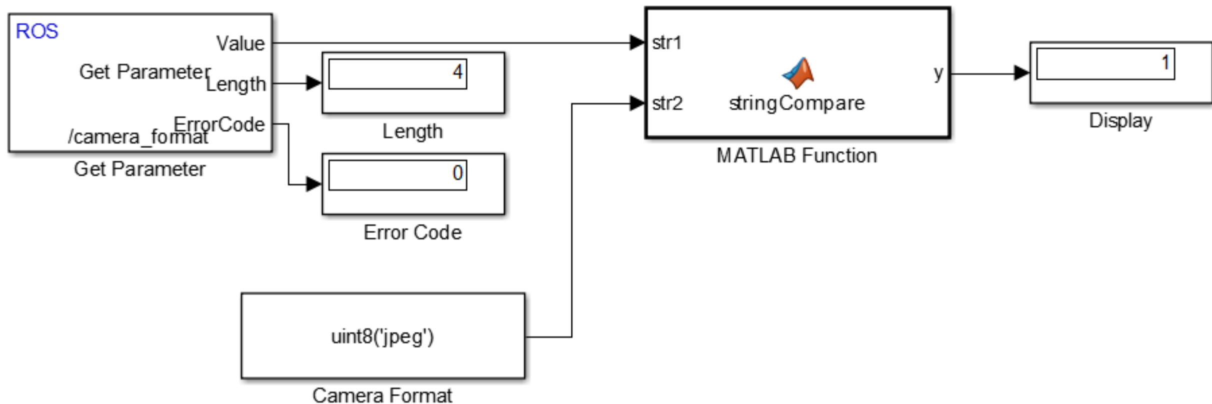
Set String Parameter on ROS Network

To create your string parameter, use a Constant block and cast it to `uint8` by specifying `uint8(['my_string_parameter'])` in **Constant Value** of the block mask. The string is passed into the Set Parameter block along with the extra input, Length, specified with a second Constant block. The Length refers to the maximum expected string length and is required for all string parameters. For more information, see the Set Parameter block.



Get ROS String Parameter and Compare to Specified String

You can compare string parameters to specified strings to validate settings or trigger subsystems. To get the parameter off the server, use the Get Parameter block. Then, use the MATLAB Function block to compare the parameter to a `uint8` string from a Constant block. This model checks to see if a previously set camera format parameter is named `'jpeg'`.



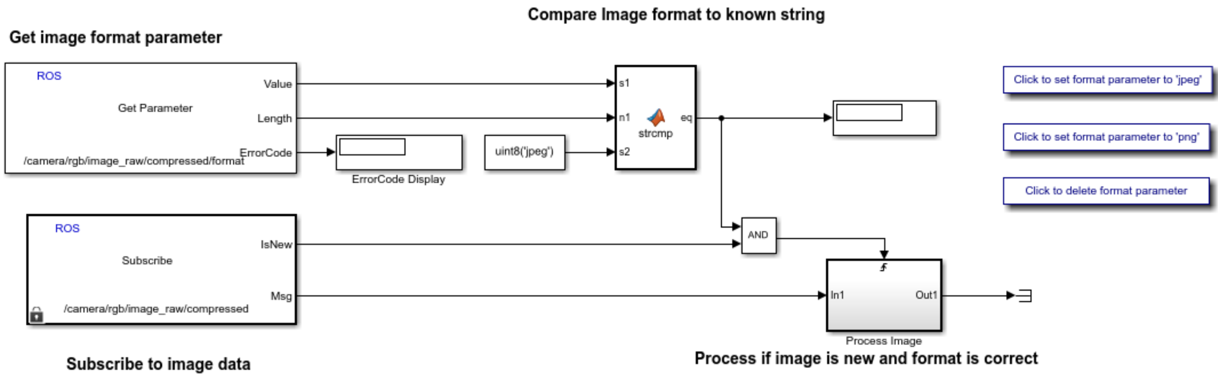
The following code is used inside the `stringCompare` MATLAB Function block. The function compares each character of its two input strings to see if they match. The output is a single Boolean indicating whether the strings match.

```
function y = stringCompare(str1,str2)
%#codegen
minLength = min(length(str1),length(str2));
st1 = str1(1:minLength);
st2 = str2(1:minLength);
y = all(st1(:)==st2(:));
```

Check Image Encoding Parameter for ROS Image Message

This model shows how to access string parameters and use them to trigger subsystem operations. It gets an image format off the set up ROS parameter server. It is retrieved as a `uint8` array that is compared using the `strcmp` MATLAB function block. When a new image is received from the Subscribe block and the format is `uint8('jpeg')`, it triggers the "Process Image" block to perform a task on the image data.

This model requires you to be connected to a ROS network. Call `rosinit` in the MATLAB® command window. The `'/camera/rgb/image_raw/compressed/format'` parameter must be set and the `'/camera/rgb/image_raw/compressed'` topic must have image messages being published. Use the buttons in the model to set the image format parameters to check the `strcmp` block. The `eq` output should be 1 when the parameter is set to `'jpeg'`.



See Also

Blocks

Get Parameter | Set Parameter

More About

- “ROS Parameters in Simulink” on page 6-18

Enable ROS Time Model Stepping for Deployed ROS Nodes

You can enable a deployed ROS node to execute based on the time published on the `/clock` topic on a ROS network. To deploy a ROS node from Simulink, see “Generate a Standalone ROS Node from Simulink®”.

When you enable ROS time model stepping, the deployed ROS node executes when the published ROS time is a multiple of the base rate of the model. To enable model stepping based on ROS time:

1. Open the **Model Configuration Parameters** for your model.
2. Under **Hardware Implementation**, set **Hardware board** to `Robot Operating System (ROS)`.
3. Under **Target Hardware resources** > **ROS time**, select **Enable ROS time model stepping**.

To specify a topic to publish a notification when the model executes, check the **Enable notification after step** check box, and use the **Notification topic** (default is `/step_notify`). Subscribe to the topic to get a message every time ROS time is published. The ROS node publishes a `std_msgs/String` message type with a string containing a `'+'` or `'-'` and the model name (`+rostime_test`, for example). A `'+'` indicates the model was stepped. A `'-'` indicates the published ROS time was not a multiple of the base rate of the model.

After enabling model stepping and setting a notification topic, you can re-build and deploy your model. When starting the ROS node, the model waits for the ROS time to be published.

You can also enable overrun detection if the model execution is still processing when the next step is triggered by the ROS time. For more information, see “Overrun Detection with Deployed ROS Nodes” on page 6-25.

See Also

Current Time | Subscribe

Related Examples

- “Generate a Standalone ROS Node from Simulink®”
- “Overrun Detection with Deployed ROS Nodes” on page 6-25
- “Get Started with ROS in Simulink®”
- “Exchange Data with ROS Publishers and Subscribers”

Overrun Detection with Deployed ROS Nodes

You can enable overrun detection for a deployed ROS node. To deploy a ROS node from Simulink, see "Generate a Standalone ROS Node from Simulink®".

An overrun occurs when the deployed Simulink model is still processing the last step, but the next step is requested.

When you enable overrun detection, the deployed ROS node notifies the user through the `ROS_ERROR` logging mechanism (see ROS Logging). The error is output to the ROS console command line. To enable overrun detection on ROS time:

1. Open the **Model Configuration Parameters** for your model.
2. Under **Hardware Implementation**, set **Hardware board** to `Robot Operating System (ROS)`.
3. Under **Operating system/scheduler settings** > **Operating system options**, select **Detect task overruns**.

After enabling **Detect task overruns**, you can re-build and deploy your model. When starting the ROS node, the model waits for the ROS time to be published. When an overrun is detected, an error is output to the ROS console command line, recorded in the log file, and published via `/rosout`. A typical error is:

```
[ERROR [1518780859.389633256, 214281.990000000]: !!! Overrun 1 !!!
```

The model continues executing when the previous step finishes, and waits for the next time step.

When an overrun condition occurs, you can correct it using one of the following approaches:

- Simplify the model
- Increase the sample times for the model and the blocks in it. For example, change the **Same time** parameter in all of your data source blocks from `0.1` to `0.2`.

See Also

Current Time | Subscribe

Related Examples

- “Generate a Standalone ROS Node from Simulink®”
- “Enable ROS Time Model Stepping for Deployed ROS Nodes” on page 6-23
- “Get Started with ROS in Simulink®”
- “Exchange Data with ROS Publishers and Subscribers”

ROS Simulink Support and Limitations

To see a full list of ROS support in Simulink, see “ROS Access with Simulink”.

Robotics System Toolbox does not support the following ROS features in Simulink:

- ROS Services
- ROS Actions
- Transformation trees

If your application requires these features, consider using MATLAB ROS functionality. You can write a ROS node using MATLAB that can publish services, actions, and transformation trees to a topic as ROS messages. Simulink can then subscribe to that topic to work with those messages. The following functions are used in MATLAB to work with these features:

- ROS Services: `rosservice`, `rossvcserver`, `rossvcclient`, `call`
- ROS Actions: `rosaction`, `rosactionclient`
- Transformation trees: `rostf`, `transform`, `getTransform`

See Also

Related Examples

- “ROS String Parameters” on page 6-20
- “Simulink and ROS Interaction” on page 6-16
- “Managing Array Sizes in Simulink ROS” on page 6-13

Read A ROS Image Message In Simulink®

This example requires Computer Vision System Toolbox® and Robotics System Toolbox®.

Start a ROS network.

```
rosinit
```

```
Initializing ROS master on http://bat5838win64:62242/.
Initializing global node /matlab_global_node_16670 with NodeURI http://bat5838win64:622
```

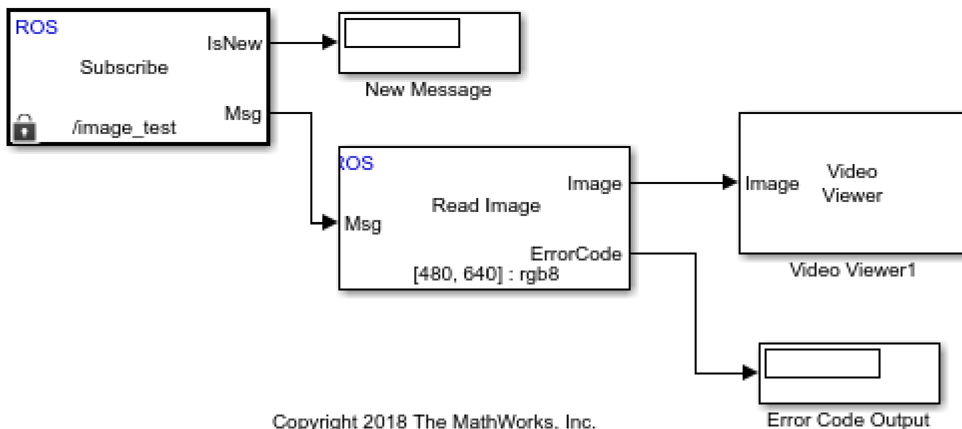
Load sample messages to send including a sample image message, `img`. Create a publisher to send a ROS `Image` message on the `'/image_test'` topic. Specify the message type as `'/sensor_msgs/Image'`. Send the image message.

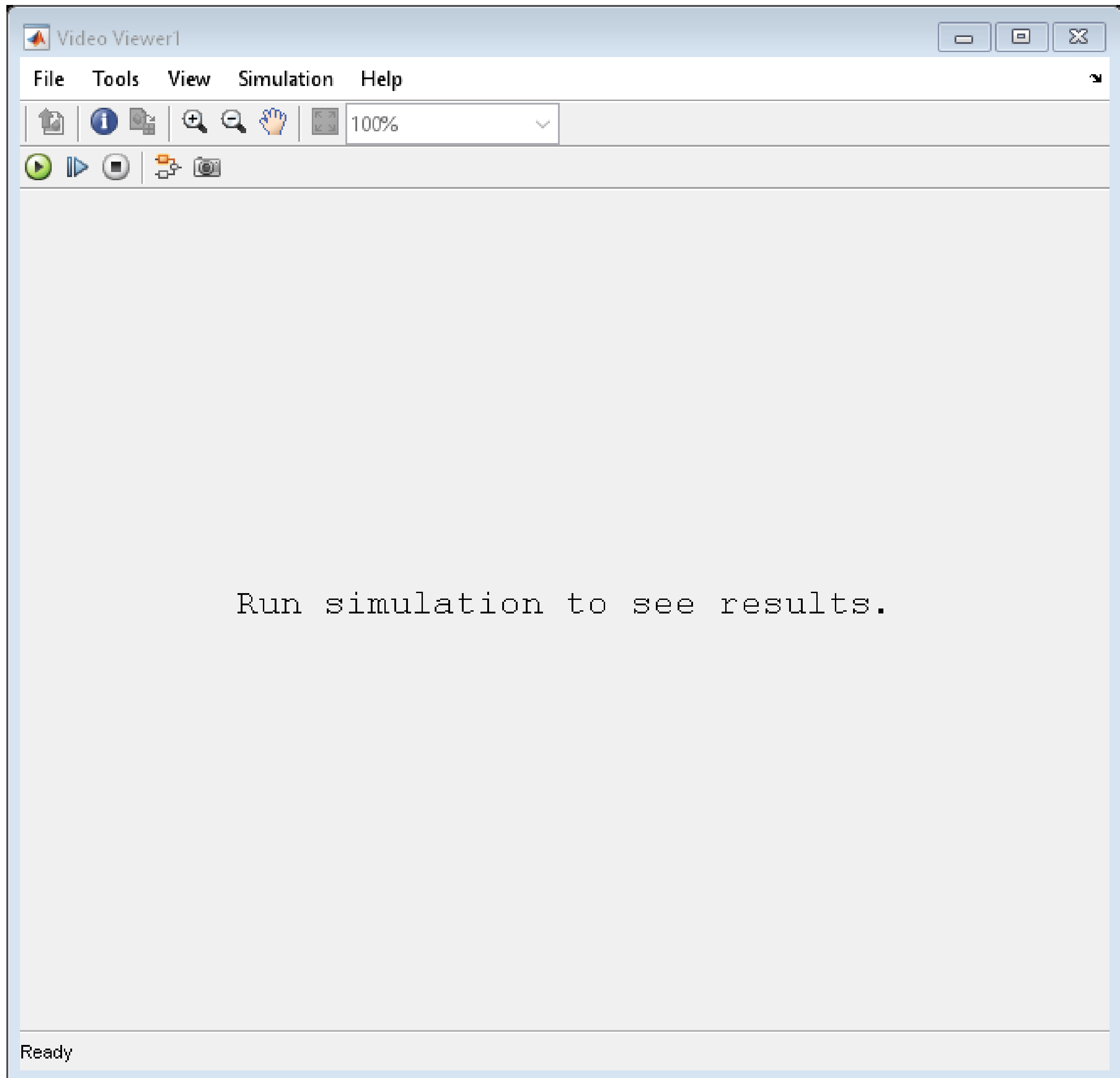
```
exampleHelperROSLoadMessages
pub = rospublisher('/image_test','sensor_msgs/Image');
send(pub,img)
```

Open the Simulink® model for subscribing to the ROS message and reading in the image from the ROS.

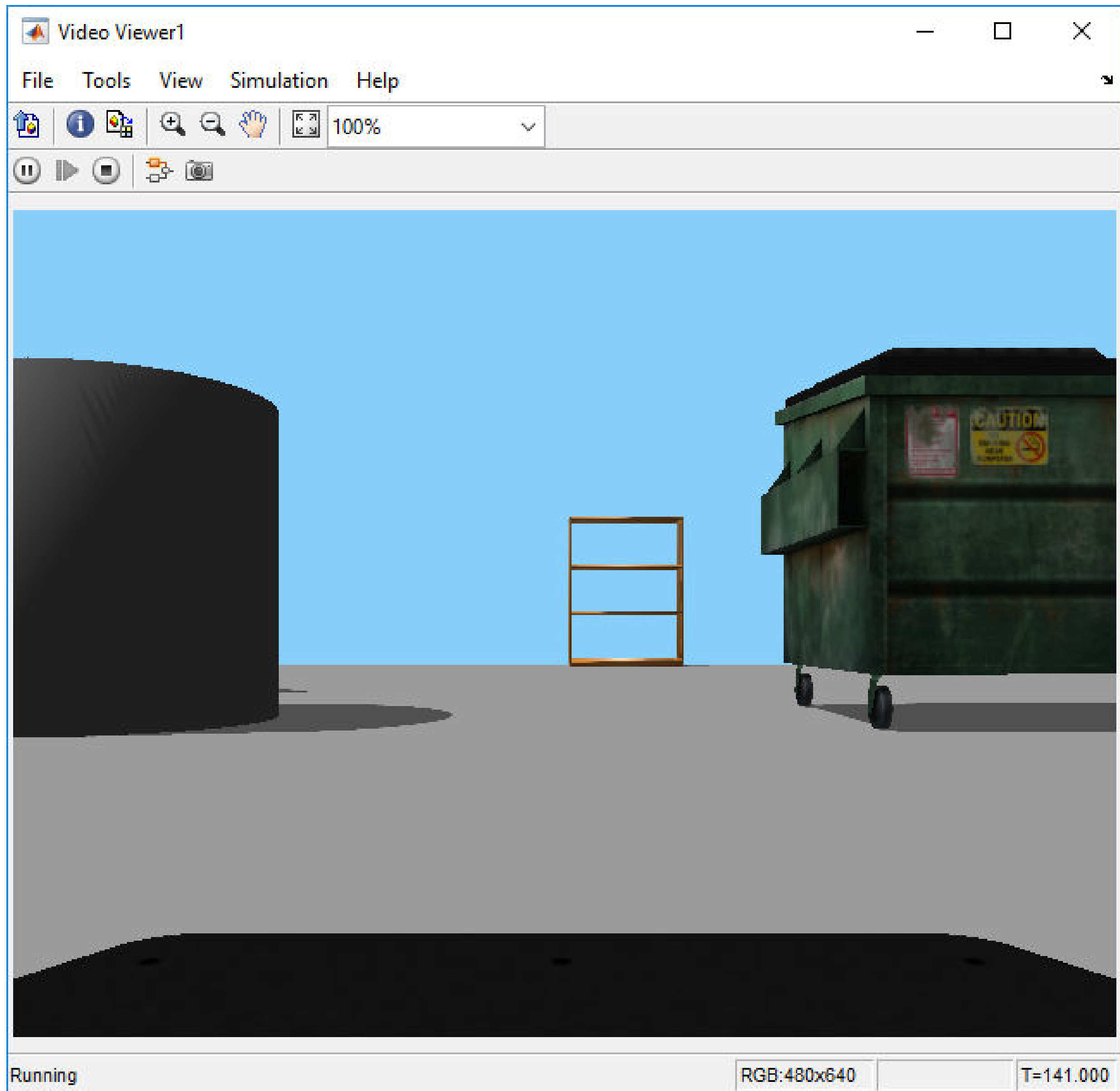
Ensure that the `Subscribe` block is subscribing to the `'/image_test'` topic. In the menu under **Tools** > **Robot Operating System** > **Manage Array Lengths**, verify the `Data` array has a maximum length greater than the sample image (921,600 pixels).

```
open_system('read_image_example_model.slx')
```





Run the model. The `Video Viewer` shows the sample image.



Stop the simulation and shut down the ROS network.

```
rosshutdown
```

```
Shutting down global node /matlab_global_node_16670 with NodeURI http://bat5838win64:62
Shutting down ROS master on http://bat5838win64:62242/.
```

Read A ROS Point Cloud Message In Simulink®

Read in a point cloud message from a ROS network. Calculate the center of mass of the coordinates and display the point cloud as an image.

This example requires Computer Vision System Toolbox® and Robotics System Toolbox®.

Start a ROS network.

```
rosinit
```

```
Initializing ROS master on http://bat5838win64:60908/.
Initializing global node /matlab_global_node_26616 with NodeURI http://bat5838win64:609
```

Load sample messages to send including a sample point cloud message, `ptcloud`. Create a publisher to send an ROS `PointCloud2` message on the `'/ptcloud_test'` topic. Specify the message type as `'sensor_msgs/PointCloud2'`. Send the point cloud message.

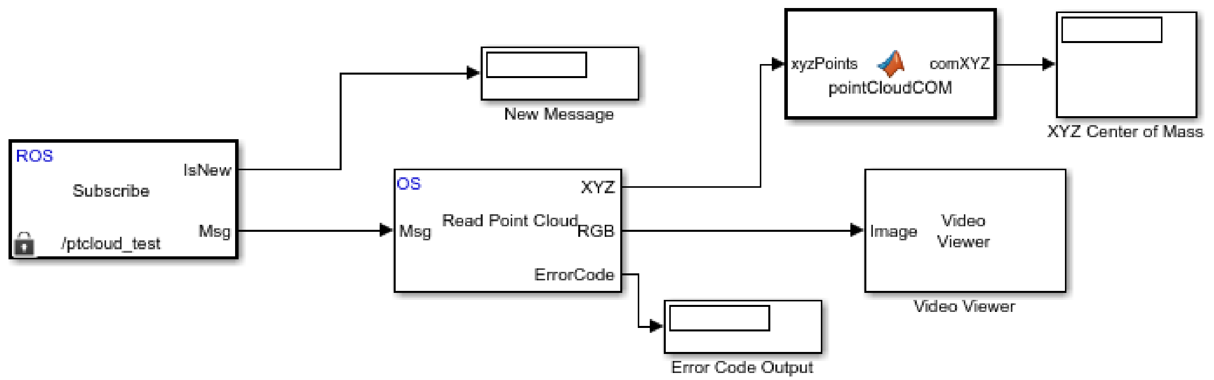
```
exampleHelperROSLoadMessages
pub = rospublisher('/ptcloud_test','sensor_msgs/PointCloud2');
send(pub,ptcloud)
```

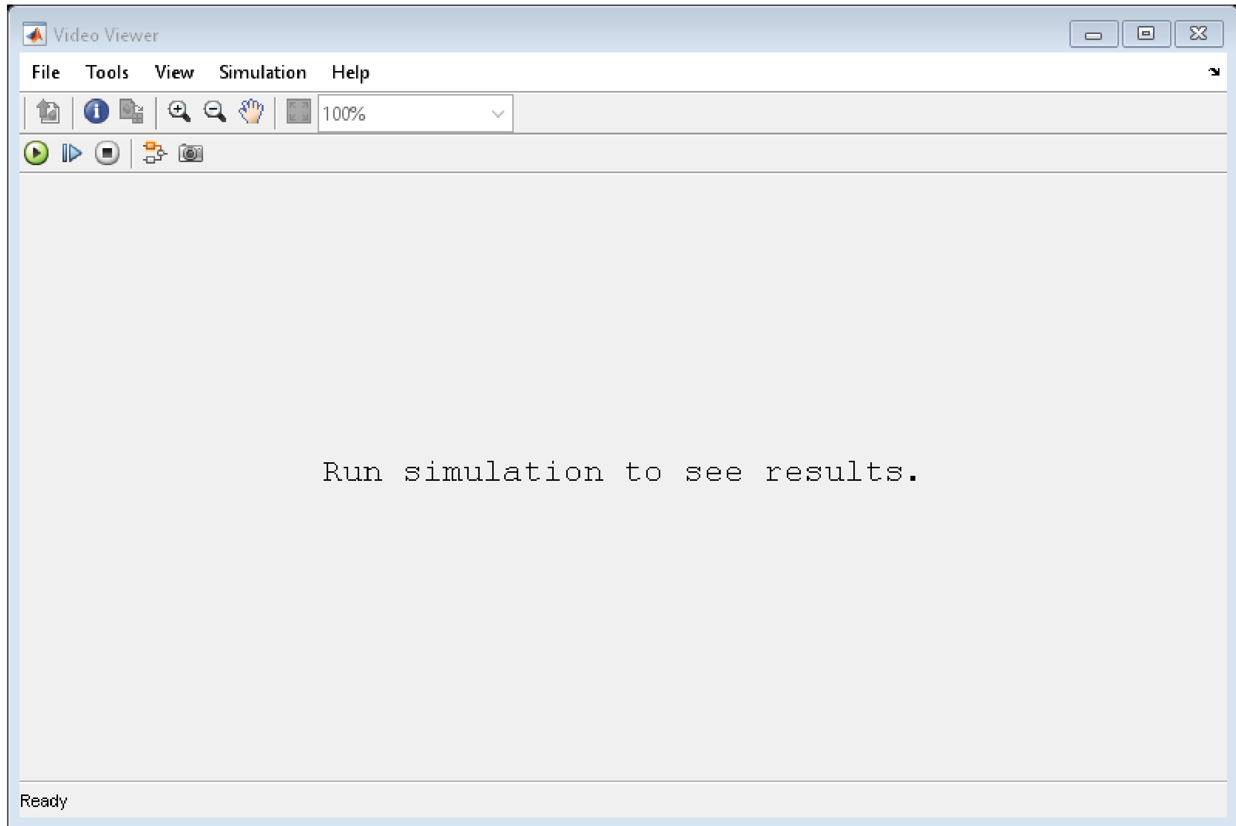
Open the Simulink® model for subscribing to the ROS message and reading in the point cloud from the ROS.

Ensure that the `Subscribe` block is subscribing to the `'/ptcloud_test'` topic. In the menu under **Tools** > **Robot Operating System** > **Manage Array Lengths**, verify the `Data` array has a maximum length greater than the sample image (9,830,400 points).

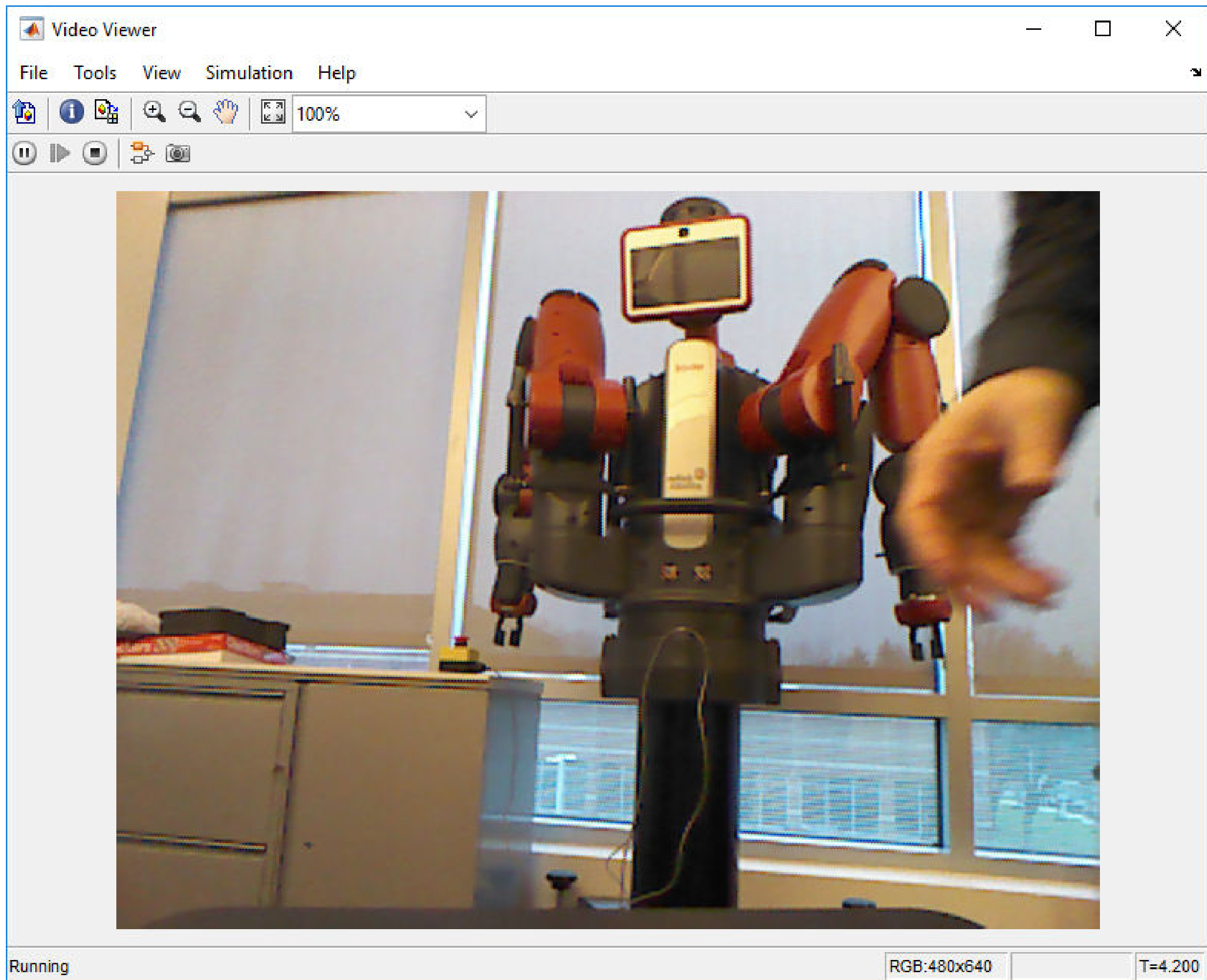
The model only displays the RGB values of the point cloud as an image. The `XYZ` output is used to calculate the center of mass (mean) of the coordinates using a MATLAB Function block. All `NaN` values are ignored.

```
open_system('read_point_cloud_example_model.slx')
```


ROS
Subscribe
IsNew
Msg
/ptcloud_test
New Message
OS
Read Point Cloud
Msg
XYZ
RGB
ErrorCode
xyzPoints
comXYZ
pointCloudCOM
XYZ Center of Mass
Image
Video
Viewer
Video Viewer
Error Code Output
Copyright 2018 The MathWorks, Inc.



Run the model. The `Video Viewer` shows the sample point cloud as an image. The output center of mass is `[-0.2869 -0.0805 2.232]` for this point cloud.



Stop the simulation and shut down the ROS network.

```
rosshutdown
```

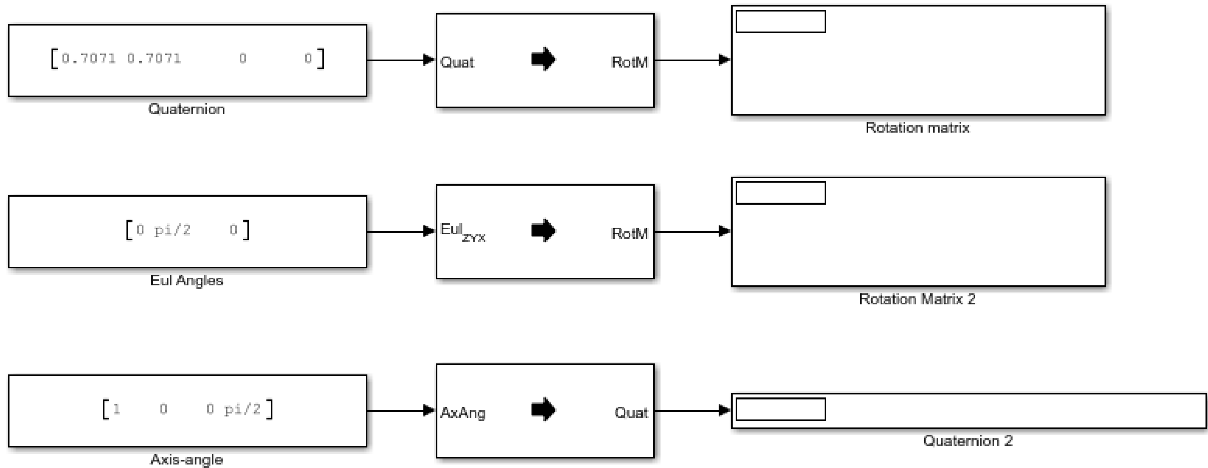
```
Shutting down global node /matlab_global_node_26616 with NodeURI http://bat5838win64:6
Shutting down ROS master on http://bat5838win64:60908/.
```

The `pointCloudCOM` function block contains the following code for calculating the center of mass of the coordinates.

```
function comXYZ = pointCloudCOM(xyzPoints)
% Compute the center of mass of a point cloud based on the input NxMx3
% matrix.

% Turn matrix into vectors.
xPoints = reshape(xyzPoints(:,:,1),numel(xyzPoints(:,:,1)),1);
yPoints = reshape(xyzPoints(:,:,2),numel(xyzPoints(:,:,2)),1);
zPoints = reshape(xyzPoints(:,:,3),numel(xyzPoints(:,:,3)),1);

% Calculate the mean for each set of coordinates.
xMean = mean(xPoints,'omitnan');
yMean = mean(yPoints,'omitnan');
zMean = mean(zPoints,'omitnan');

comXYZ = [xMean,yMean,zMean];

end
```

Convert Coordinate System Transformations

This model shows how to convert some basic coordinate system transformations into other coordinate systems. Input vectors are expected to be vertical vectors.



Call ROS Service in Simulink

Use the Call Service block to call a service on the ROS service server.

Connect to a ROS network.

```
rosinit
```

```
Initializing ROS master on http://bat5838win64:51982/.
Initializing global node /matlab_global_node_31711 with NodeURI http://bat5838win64:51
```

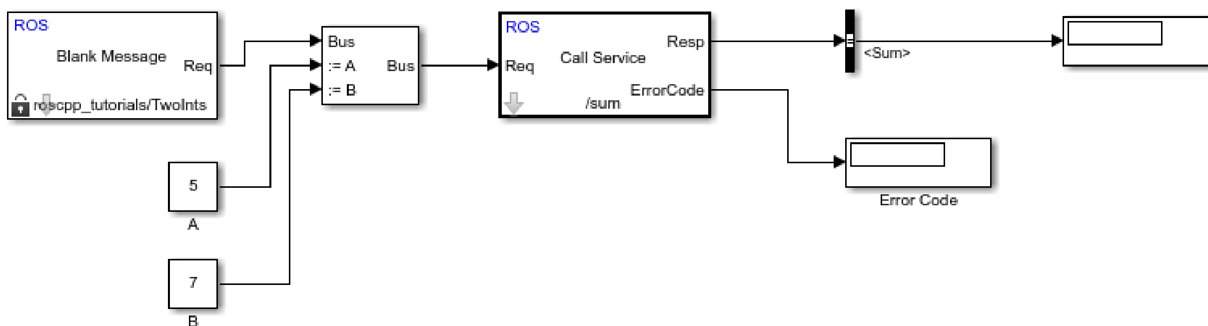
Set up a `roscpp_tutorials/TwoInts` service server message type and specify a example helper callback function. The call back function sums the `A` and `B` elements of a `roscpp_tutorials/TwoIntsRequest message`. The service server must be set up before you can call a service client.

```
sumserver = rossvcserver('/sum','roscpp_tutorials/TwoInts',@exampleHelperROSSumCallbac
```

```
sumserver = 
  ServiceServer with properties:

      ServiceName: '/sum'
      ServiceType: 'roscpp_tutorials/TwoInts'
    NewRequestFcn: @exampleHelperROSSumCallback
```

Open a Simulink® model with the **Call Service** block. Use the **Blank Message** block to output a request message with the `roscpp_tutorials/TwoIntsRequest` message type. Populate the bus with two values to sum togther.

```
open_system('ros_twoint_service_simulink_example.slx')
```



Run the model. The service call should return 0 in the `Resp` output as part of the response message. An error code of 0 indicates the service call was successful. You can ignore warnings about converting data types.

```
sim('ros_twoint_service_simulink_example.slx')
```

```
Warning: The property "A" in ROS message type "roscpp_tutorials/TwoIntsRequest" has an

Warning: The property "B" in ROS message type "roscpp_tutorials/TwoIntsRequest" has an

Warning: The property "Sum" in ROS message type "roscpp_tutorials/TwoIntsResponse" has
```

Shut down the ROS network to disconnect.

```
rosshutdown
```

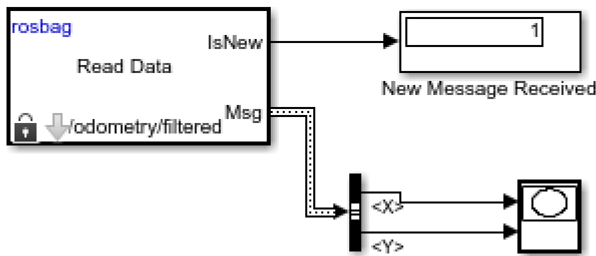
```
Shutting down global node /matlab_global_node_31711 with NodeURI http://bat5838win64:5
Shutting down ROS master on http://bat5838win64:51982/.
```

Play Back Data from Jackal™ rosbag Logfile in Simulink

Use the **Read Data** block to play back data from a rosbag logfile recorded from a Jackal™ robot from ClearPath Robotics™.

Load the model.

```
open_system('read_jackal_pose_log.slx')
```



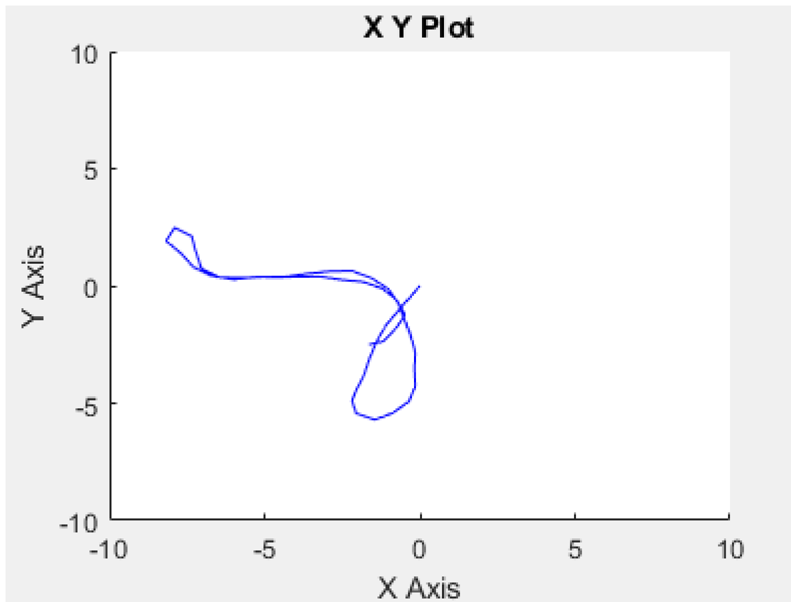
Open the **Read Data** block mask to load a rosbag logfile. Click the **Load logfile data** link. Browse for the logfile and specify a time offset or limited duration if needed.

Select the desired topic, `/odometry/filtered`, which contains `nav_msgs/Odometry` messages.

The **Read Data** block outputs the messages from the rosbag logfile. A bus selector extracts the *xy*-position from the `nav_msgs/Odometry` messages

Run the model. The block plays back data in sync with the simulation time. The **XY Graph** plot displays the robot position over time.

```
sim(gcs)
```



See Also

Blocks

Publish | Read Image | Read Point Cloud | Subscribe

Functions

`readMessages` | `rosbag` | `select`

Related Examples

- “Work with ROS Messages in Simulink®”
- “Work with rosbag Logfiles”

Time Stamp A ROS Message Using Current Time in Simulink

This example shows how to specify the time stamp a ROS message with the current system time of your computer. Use the **Current Time** block and assign the output to the `std_msgs/Header` message in the `Stamp` field. Publish the message on a desired topic.

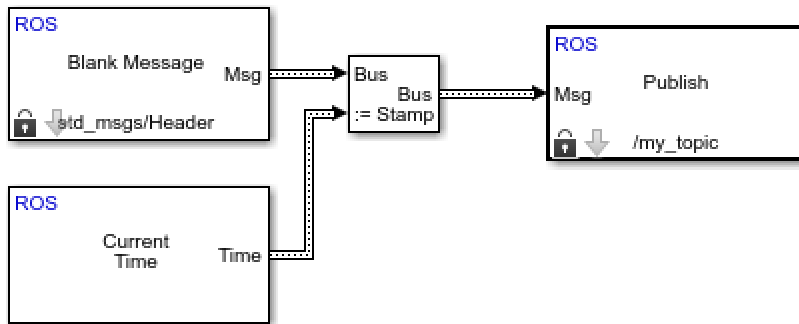
Connect to a ROS network.

```
rosinit
```

```
Initializing ROS master on http://bat5838win64:62345/.
Initializing global node /matlab_global_node_87729 with NodeURI http://bat5838win64:623
```

Open the Simulink model.

```
open_system(fullfile(matlabroot,'/examples/robotics/','current_time_stamp_message.slx')
```



Run the model. The **Publish** block should publish the `Header` message with the current system time.

```
sim(fullfile(matlabroot,'/examples/robotics/','current_time_stamp_message.slx'))
```

Shut down the ROS network.

```
rosshutdown
```

```
Shutting down global node /matlab_global_node_87729 with NodeURI http://bat5838win64:62
Shutting down ROS master on http://bat5838win64:62345/.
```

See Also

Blocks
Get Parameter | Publish | Set Parameter

Functions
`get` | `rosparam` | `rospublisher` | `rostime` | `set`

External Websites

- ROS Time

Algorithm Design

- “Occupancy Grids” on page 7-2
- “Particle Filter Parameters” on page 7-14
- “Particle Filter Workflow” on page 7-21
- “Probabilistic Roadmaps (PRM)” on page 7-30
- “Pure Pursuit Controller” on page 7-41
- “Vector Field Histogram” on page 7-44
- “Monte Carlo Localization Algorithm” on page 7-52
- “Compose a Series of Laser Scans with Pose Changes” on page 7-63
- “Rigid Body Tree Robot Model” on page 7-68
- “Build a Robot Step by Step” on page 7-74
- “Inverse Kinematics Algorithms” on page 7-79

Occupancy Grids

In this section...

"Overview" on page 7-2

"World and Grid Coordinates" on page 7-3

"Inflation of Coordinates" on page 7-6

"Log-Odds Representation of Probability Values" on page 7-11

Overview

Occupancy grids are used to represent a robot workspace as a discrete grid. Information about the environment can be collected from sensors in real time or be loaded from prior knowledge. Laser range finders, bump sensors, cameras, and depth sensors are commonly used to find obstacles in your robot's environment.

Occupancy grids are used in robotics algorithms such as path planning (see `robotics.PRM`). They are used in mapping applications for integrating sensor information in a discrete map, in path planning for finding collision-free paths, and for localizing robots in a known environment (see `robotics.MonteCarloLocalization`). You can create maps with different sizes and resolutions to fit your specific application.

2-D occupancy grids have two representations:

- Binary occupancy grid (see `robotics.BinaryOccupancyGrid`)
- Probability occupancy grid (see `robotics.OccupancyGrid`)

A binary occupancy grid uses `true` values to represent the occupied workspace (obstacles) and `false` values to represent the free workspace. This grid shows where obstacles are and whether a robot can move through that space. Use a binary occupancy grid if memory size is a factor in your application.

A probability occupancy grid uses probability values to create a more detailed map representation. This representation is the preferred method for using occupancy grids. This grid is commonly referred to as simply an occupancy grid. 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. The probabilistic values can give better fidelity of objects and improve performance of certain algorithm applications.

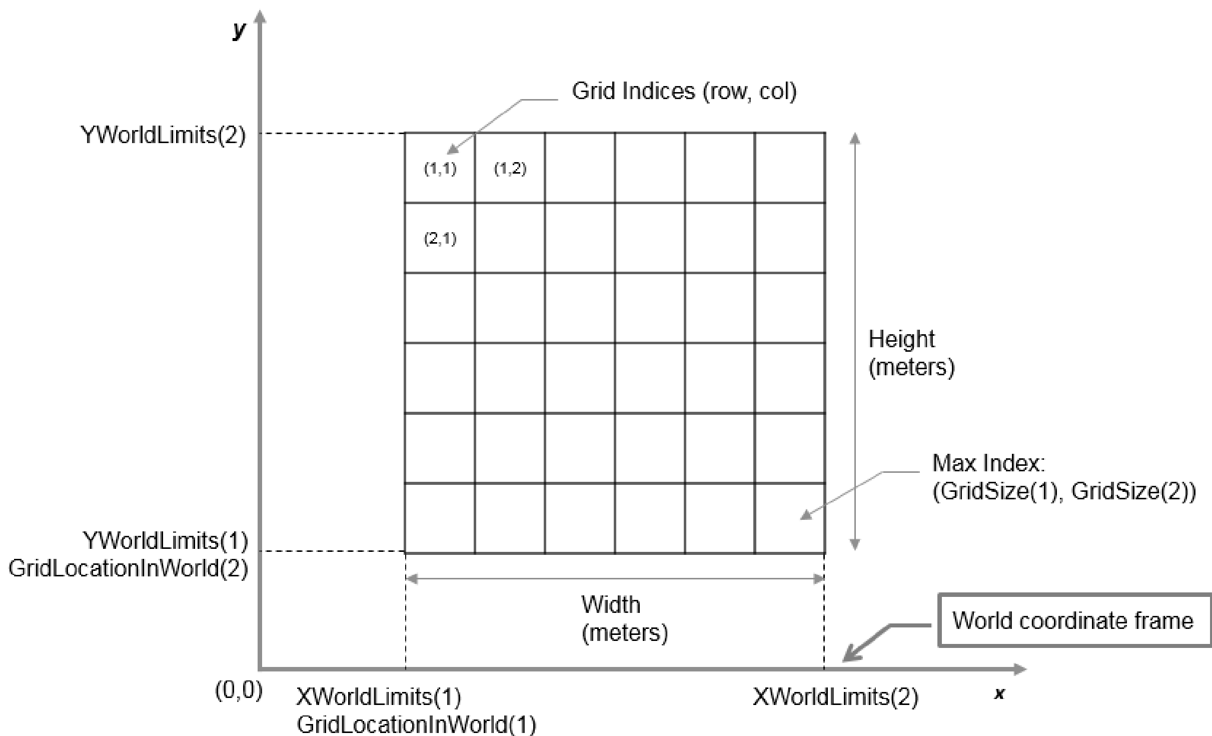
Binary and probability occupancy grids share several properties and algorithm details. Grid and word coordinates apply to both types of occupancy grids. The inflation method also applies to both grids, but each grid implements it differently. The effects of the log-odds representation and probability saturation apply to probability occupancy grids only.

World and Grid Coordinates

When working with occupancy grids in MATLAB, you can use either world or grid coordinates.

The absolute reference frame in which the robot operates is referred to as the world frame in the occupancy grid. Most Robotics System Toolbox operations are performed in the world frame, and it is the default selection when using MATLAB functions in this toolbox. World coordinates are used as an absolute coordinate frame with a fixed origin, and points can be specified with any resolution. However, all locations are converted to grid locations because of data storage and resolution limits on the map itself.

Grid coordinates define the actual resolution of the occupancy grid and the finite locations of obstacles. The origin of grid coordinates is in the top-left corner of the grid, with the first location having an index of `(1,1)`. However, the `GridLocationInWorld` property of the occupancy grid in MATLAB defines the bottom-left corner of the grid in world coordinates. When creating an occupancy grid object, properties such as `XWorldLimits` and `YWorldLimits` are defined by the input `width`, `height`, and `resolution`. This figure shows a visual representation of these properties and the relation between world and grid coordinates.



Grid Coordinates from World Coordinate Inputs

When setting occupancy locations, you can input the locations in either grid or world coordinates. However, based on the limits of the grid, the locations are set to the closest grid locations. Edges of the grid belong to the lower-left grid location.

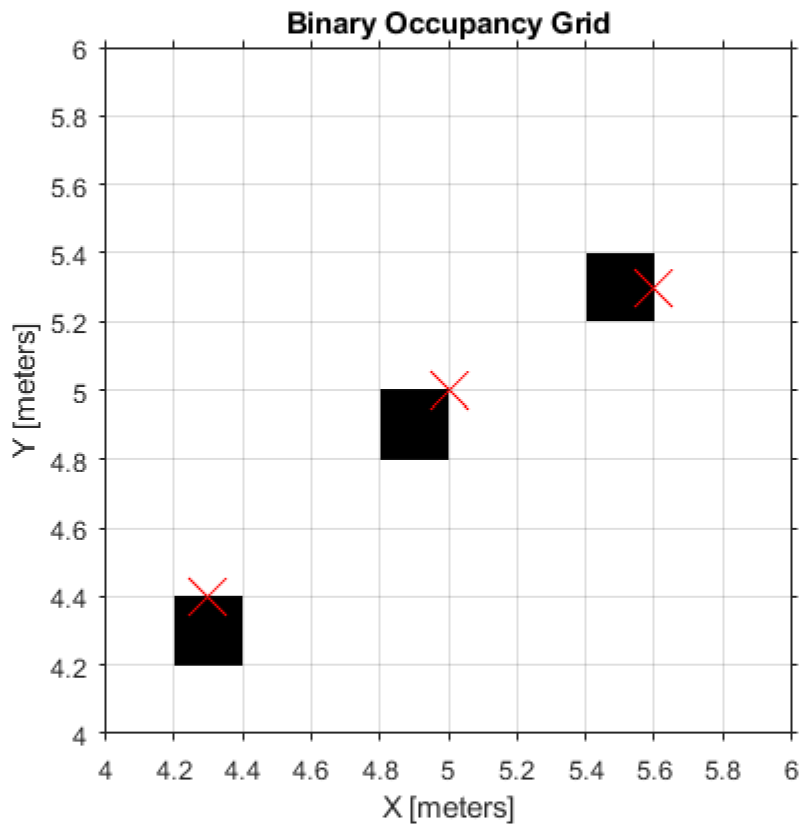
Show how locations are interpreted on the grid by creating an occupancy grid and setting points as occupied by obstacles. Then, plot the original input points over the map to show how they are interpreted. If any point within the grid cell is set as occupied, the entire grid cell is set as occupied.

Create an occupancy grid map and set obstacle locations.

```
map = robotics.BinaryOccupancyGrid(10,10,5);
xy = [5 5; 4.3 4.4; 5.6 5.3];
setOccupancy(map,xy,1);
```

Display the map, original points, and set the axes limits to zoom in. You can see how the edge points affect the entire grid location status.

```
show(map);
hold on
plot(xy(:,1),xy(:,2),'xr','MarkerSize', 20)
grid on
set(gca,'XTick',0:0.2:10,'YTick',0:0.2:10)
xlim([4 6])
ylim([4 6])
```



Inflation of Coordinates

Both the binary and normal occupancy grids have an option for inflating obstacles. This inflation is used to add a factor of safety on obstacles and create buffer zones between the robot and obstacle in the environment. The `inflate` method of an occupancy grid object converts the specified `radius` to the number of cells rounded up from the `resolution*radius` value. Each algorithm uses this cell value separately to modify values around obstacles.

Binary Occupancy Grid

The `robotics.BinaryOccupancyGrid.inflate` method takes each occupied cell and directly inflates it by adding occupied space around each point. This basic inflation example illustrates how the radius value is used.

Inflate Obstacles in a Binary Occupancy Grid

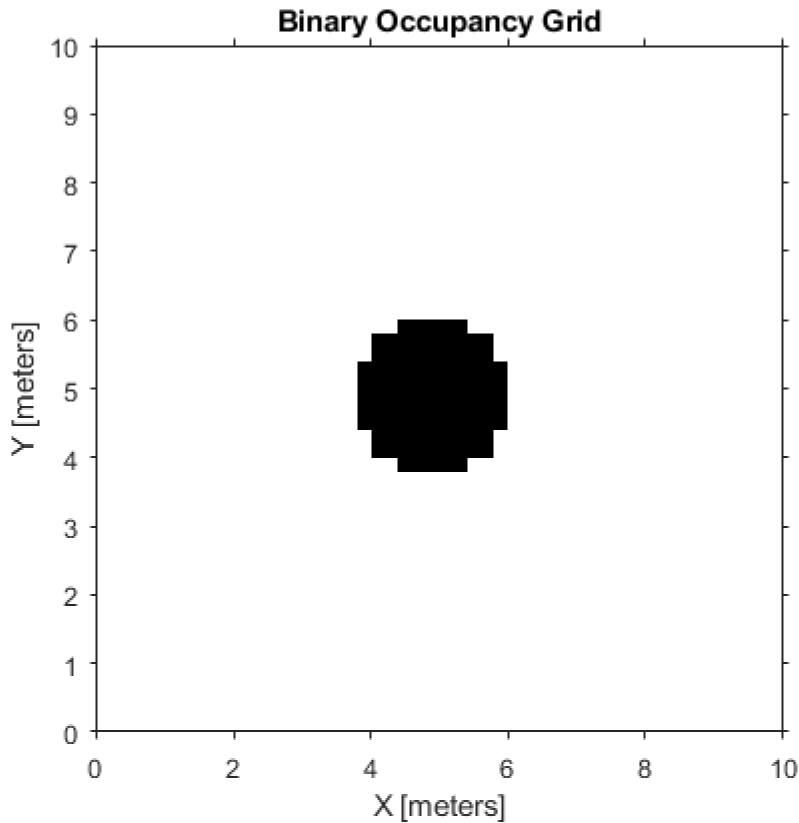
This example shows how to create the map, set the obstacle locations and inflate it by a radius of 1m. Extra plots on the figure help illustrate the inflation and shifting due to conversion to grid locations.

Create binary occupancy grid. Set occupancy of position [5,5].

```
map = robotics.BinaryOccupancyGrid(10,10,5);
setOccupancy(map,[5 5], 1);
```

Inflate occupied spaces on map by 1m.

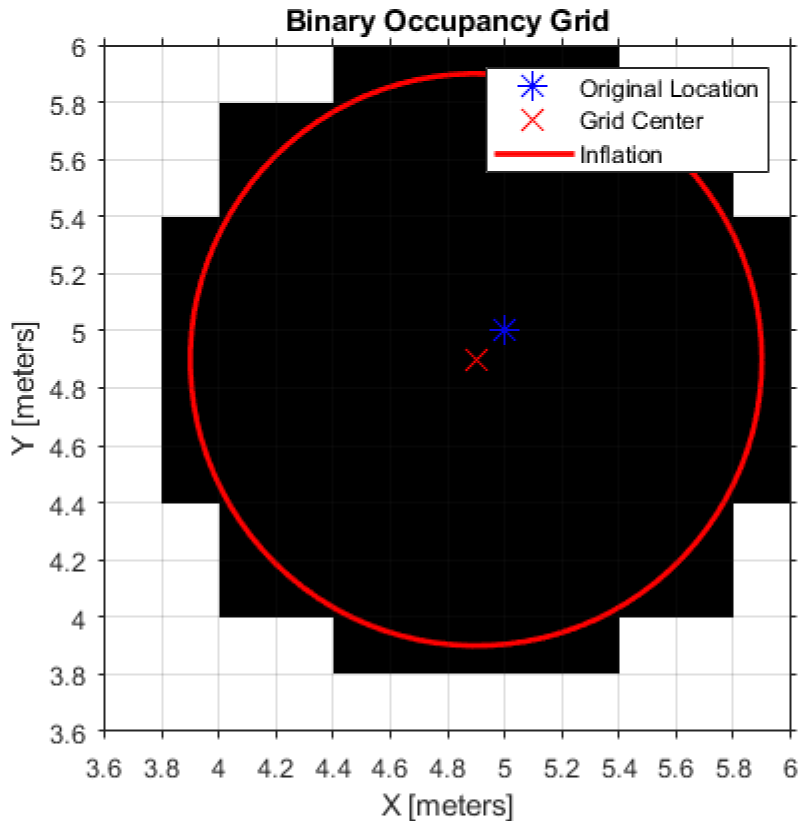
```
inflate(map,1);
show(map)
```



Plot original location, converted grid position and draw the original circle. You can see from this plot, that the grid center is [4.9 4.9], which is shifted from the [5 5] location. A 1m circle is drawn from there and notice that any cells that touch this circle are marked as occupied. The figure is zoomed in to the relevant area.

```
hold on
theta = linspace(0,2*pi);
x = 4.9+cos(theta); % x circle coordinates
y = 4.9+sin(theta); % y circle coordinates
plot(5,5,'*b','MarkerSize',10) % Original location
plot(4.9,4.9,'xr','MarkerSize',10) % Grid location center
plot(x,y,'-r','LineWidth',2); % Circle of radius 1m.
axis([3.6 6 3.6 6])
ax = gca;
```

```
ax.XTick = [3.6:0.2:6];
ax.YTick = [3.6:0.2:6];
grid on
legend('Original Location','Grid Center','Inflation')
```



As you can see from the above figure, even cells that barely overlap with the inflation radius are labeled as occupied.

Occupancy Grids

The `robotics.OccupancyGrid.inflate` method uses the inflation radius to perform *probabilistic inflation*. Probabilistic 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 navigate around obstacles. This example shows how the inflation works with a range of probability values.

Inflate Obstacles in an Occupancy Grid

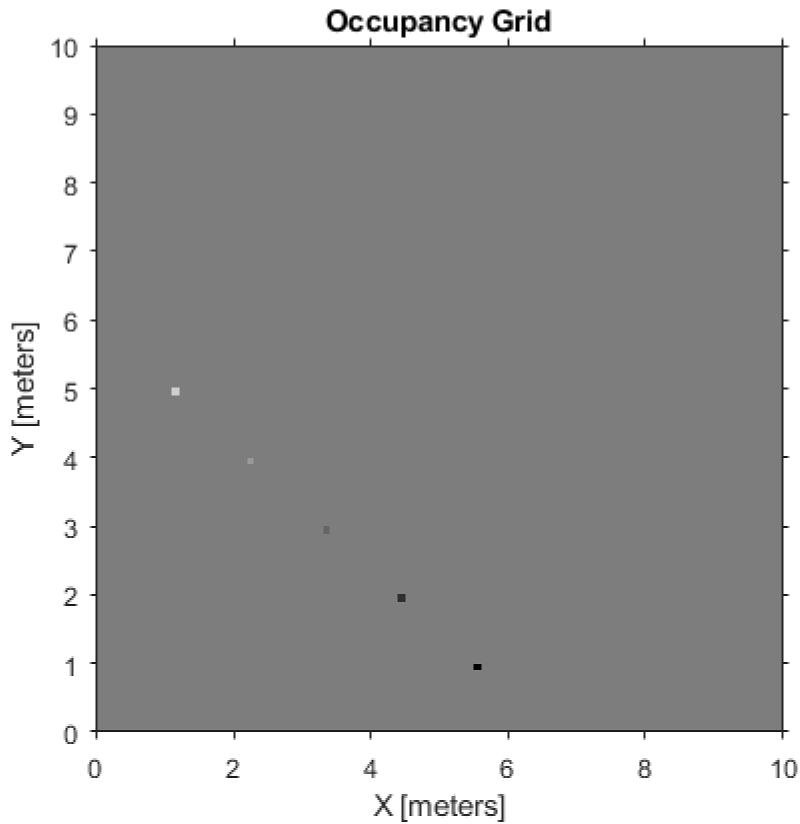
This example shows how the inflate method performs probabilistic inflation on obstacles to inflate their size and create a buffer zone for areas with a higher probability of obstacles.

Create a 10m x 10m empty map.

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

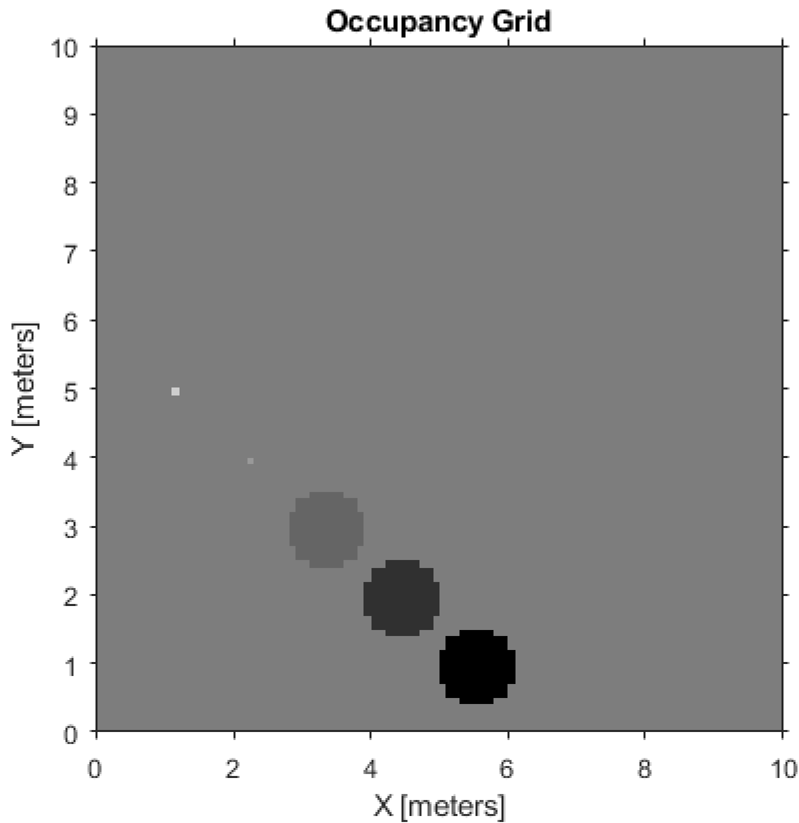
Update occupancy of world locations with specific values in `pvalues`.

```
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 are written over smaller values. You can copy your map beforehand to revert any unwanted changes.

```
savedMap = copy(map);
inflate(map,0.5)
figure
show(map)
```

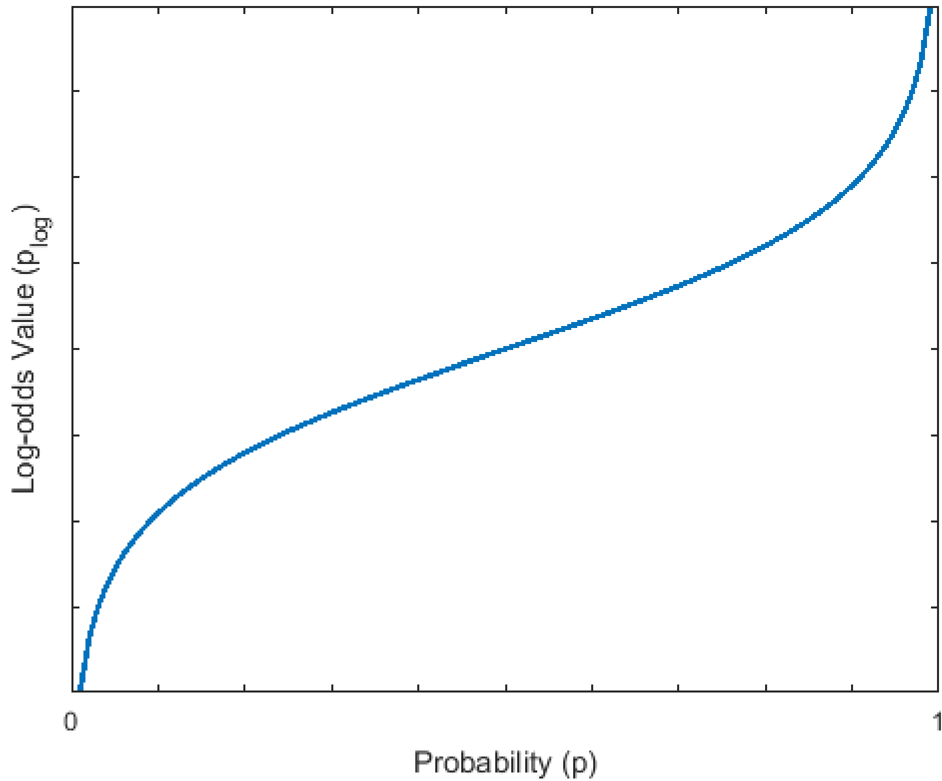


Log-Odds Representation of Probability Values

When using occupancy grids with probability values, the goal is to estimate the probability of obstacle locations for use in real-time robotics applications. The `OccupancyGrid` class uses a *log-odds* representation of the probability values for each cell. Each probability value is converted to a corresponding log-odds value for internal storage. The value is converted back to probability when accessed. This representation efficiently updates probability values with the fewest operations. Therefore, you can quickly integrate sensor data into the map.

The log-odds representation uses the following equation:

$$p_{\log} = \log(\frac{p}{1-p})$$



Note Log-odds values are stored as `int16` values. This data type limits the resolution of probability values to ±0.001 but greatly improves memory size and allows for creation of larger maps.

Probability Saturation

When updating an occupancy grid with observations using the log-odds representation, the values have a range of $-\infty$ to ∞. This range means if a robot observes a location such

as a closed door multiple times, the log-odds value for this location becomes unnecessarily high, or the value probability gets saturated. If the door then opens, the robot needs to observe the door open many times before the probability changes from occupied to free. In dynamic environments, you want the map to react to changes to more accurately track dynamic objects.

To prevent this saturation, update the `ProbabilitySaturation` property, which limits the minimum and maximum probability values allowed when incorporating multiple observations. This property is an upper and lower bound on the log-odds values and enables the map to update quickly to changes in the environment. The default minimum and maximum values of the saturation limits are `[0.001 0.999]`. For dynamic environments, the suggested values are at least `[0.12 0.97]`. Consider modifying this range if the map does not update rapidly enough for multiple observations.

See Also

`readBinaryOccupancyGrid` | `readOccupancyGrid` | `robotics.BinaryOccupancyGrid` | `robotics.OccupancyGrid` | `robotics.OccupancyMap3D` | `writeBinaryOccupancyGrid` | `writeOccupancyGrid`

Particle Filter Parameters

In this section...

"Number of Particles" on page 7-14
"Initial Particle Location" on page 7-15
"State Transition Function" on page 7-17
"Measurement Likelihood Function" on page 7-18
"Resampling Policy" on page 7-18
"State Estimation Method" on page 7-19

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

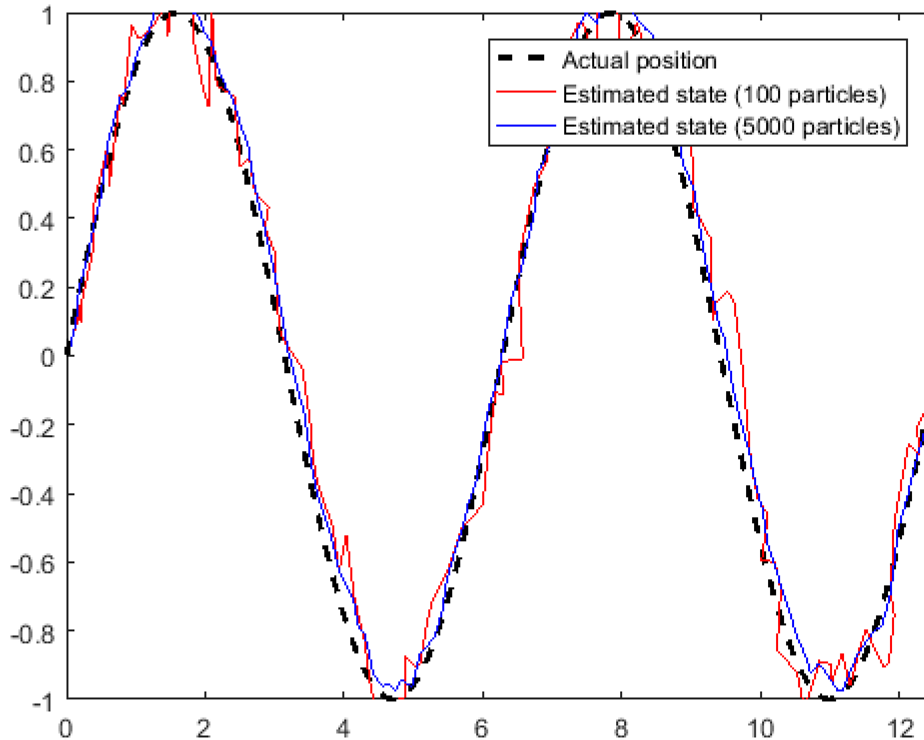
To use the particle filter properly, you must specify parameters such as the number of particles, the initial particle location, and the state estimation method. Also, if you have a specific motion and sensor model, you specify these parameters in the state transition function and measurement likelihood function, respectively. The details of these parameters are detailed on this page. For more information on the particle filter workflow, see "Particle Filter Workflow" on page 7-21.

Number of Particles

To specify the number of particles, use the `initialize` method. Each particle is a hypothesis of the current state. The particles are distributed across your state space based on either a specified mean and covariance, or on the specified state bounds. Depending on the `StateEstimationMethod` property, either the particle with the highest weight or the mean of all particles is taken to determine the best state estimate.

The default number of particles is 1000. Unless performance is an issue, do not use fewer than 1000 particles. A higher number of particles can improve the estimate but sacrifices performance speed, because the algorithm has to process more particles. Tuning the number of particles is the best way to affect your particle filters performance.

These results, which are based on the "Estimate Robot Position in a Loop Using Particle Filter" on page 7-26 example, show the difference in tracking accuracy when using 100 particles and 5000 particles.



Initial Particle Location

When you initialize your particle filter, you can specify the initial location of the particles using:

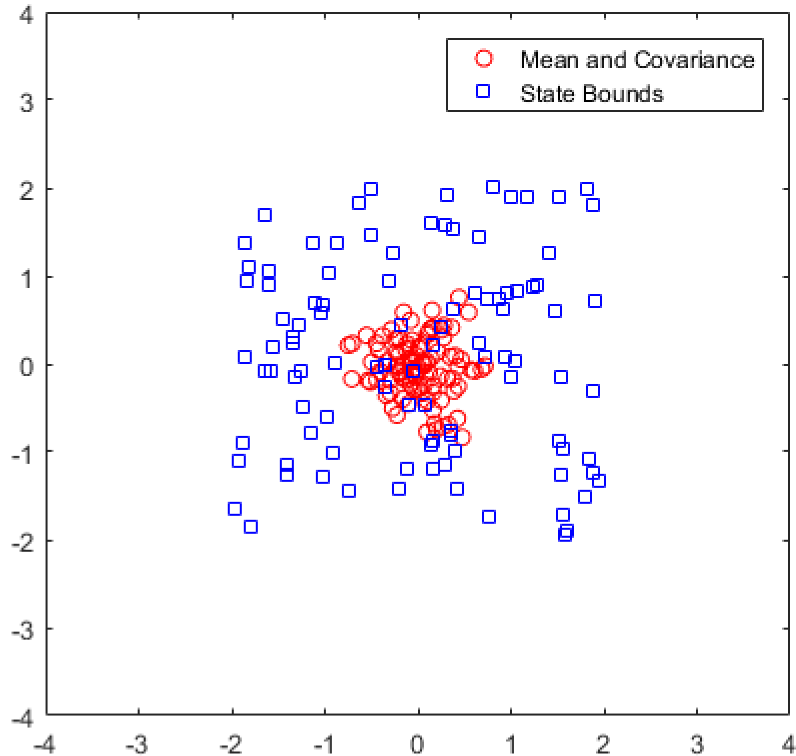
- Mean and covariance
- State bounds

Your initial state is defined as a mean with a covariance relative to your system. This mean and covariance correlate to the initial location and uncertainty of your system. The `ParticleFilter` object distributes particles based on your covariance around the given mean. The algorithm uses this distribution of particles to get the best estimation of state,

so an accurate initialization of particles helps to converge to the best state estimation quickly.

If an initial state is unknown, you can evenly distribute your particles across a given state bounds. The state bounds are the limits of your state. For example, when estimating the position of a robot, the state bounds are limited to the environment that the robot can actually inhabit. In general, an even distribution of particles is a less efficient way to initialize particles to improve the speed of convergence.

The plot shows how the mean and covariance specification can cluster particles much more effectively in a space rather than specifying the full state bounds.



State Transition Function

The state transition function, `StateTransitionFcn`, of a particle filter helps to evolve the particles to the next state. It is used during the prediction step of the “Particle Filter Workflow” on page 7-21. In the `ParticleFilter` object, the state transition function is specified as a callback function that takes the previous particles, and any other necessary parameters, and outputs the predicted location. The function header syntax is:

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

By default, the state transition function assumes a Gaussian motion model with constant velocities. The function uses a Gaussian distribution to determine the position of the particles in the next time step.

For your application, it is important to have a state transition function that accurately describes how you expect the system to behave. To accurately evolve all the particles, you must develop and implement a motion model for your system. If particles are not distributed around the next state, the `ParticleFilter` object does not find an accurate estimate. Therefore, it is important to understand how your system can behave so that you can track it accurately.

You also must specify system noise in `StateTransitionFcn`. Without random noise applied to the predicted system, the particle filter does not function as intended.

Although you can predict many systems based on their previous state, sometimes the system can include extra information. The use of `varargin` in the function enables you to input any extra parameters that are relevant for predicting the next state. When you call `predict`, you can include these parameters using:

```
predict(pf,param1,param2)
```

Because these parameters match the state transition function you defined, calling `predict` essentially calls the function as:

```
predictParticles = stateTransitionFcn(pf,prevParticles,param1,param2)
```

The output particles, `predictParticles`, are then either used by the “Measurement Likelihood Function” on page 7-18 to correct the particles, or used in the next prediction step if correction is not required.

Measurement Likelihood Function

After predicting the next state, you can use measurements from sensors to correct your predicted state. By specifying a MeasurementLikelihoodFcn in the ParticleFilter object, you can correct your predicted particles using the correct function. This measurement likelihood function, by definition, gives a weight for the state hypotheses (your particles) based on a given measurement. Essentially, it gives you the likelihood that the observed measurement actually matches what each particle observes. This likelihood is used as a weight on the predicted particles to help with correcting them and getting the best estimation. Although the prediction step can prove accurate for a small number of intermediate steps, to get accurate tracking, use sensor observations to correct the particles frequently.

The specification of the MeasurementLikelihoodFcn is similar to the StateTransitionFcn. It is specified as a function handle in the properties of the ParticleFilter object. The function header syntax is:

```
function likelihood = measurementLikelihoodFcn(pf,predictParticles,measurement,varargi
```

The output is the likelihood of each predicted particle based on the measurement given. However, you can also specify more parameters in varargin. The use of varargin in the function enables you to input any extra parameters that are relevant for correcting the predicted state. When you call correct, you can include these parameters using:

```
correct(pf,measurement,param1,param2)
```

These parameters match the measurement likelihood function you defined:

```
likelihood = measurementLikelihoodFcn(pf,predictParticles,measurement,param1,param2)
```

The correct function uses the likelihood output for particle resampling and giving the final state estimate.

Resampling Policy

The resampling of particles is a vital step for continuous tracking of objects. It enables you to select particles based on the current state, instead of using the particle distribution given at initialization. By continuously resampling the particles around the current estimate, you can get more accurate tracking and improve long-term performance.

When you call correct, the particles used for state estimation can be resampled depending on the ResamplingPolicy property specified in the ParticleFilter

object. This property is specified as a `robotics.ResamplingPolicy` object. The `TriggerMethod` property on that object tells the particle filter which method to use for resampling.

You can trigger resampling at either a fixed interval or when a minimum effective particle ratio is reached. The fixed interval method resamples at a set number of iterations, which is specified in the `SamplingInterval` property. The minimum effective particle ratio is a measure of how well the current set of particles approximates the posterior distribution. The number of effective particles is calculated by:

$$N_{eff} = \frac{1}{\sum_{i=1}^{N} \left(w^i\right)^2}$$

In this equation, N is the number of particles, and w is the normalized weight of each particle. The effective particle ratio is then N_{eff} / `NumParticles`. Therefore, the effective particle ratio is a function of the weights of all the particles. After the weights of the particles reach a low enough value, they are not contributing to the state estimation. This low value triggers resampling, so the particles are closer to the current state estimation and have higher weights.

State Estimation Method

The final step of the particle filter workflow is the selection of a single state estimate. The particles and their weights sampled across the distribution are used to give the best estimation of the actual state. However, you can use the particles information to get a single state estimate in multiple ways. With the `ParticleFilter` object, you can either choose the best estimate based on the particle with the highest weight or take a mean of all the particles. Specify the estimation method in the `StateEstimationMethod` property as either `'mean'`(default) or `'maxweight'`.

Because you can estimate the state from all of the particles in many ways, you can also extract each particle and its weight from the `robotics.ParticleFilter` using the `Particles` property.

See Also

`robotics.ParticleFilter` | `robotics.ParticleFilter.correct` | `robotics.ParticleFilter.initialize` | `robotics.ParticleFilter.predict`

Related Examples

- “Track a Car-Like Robot Using Particle Filter”
- “Estimate Robot Position in a Loop Using Particle Filter”

More About

- “Particle Filter Workflow” on page 7-21

Particle Filter Workflow

In this section...
"Estimation Workflow" on page 7-22
"Estimate Robot Position in a Loop Using Particle Filter" on page 7-26

A 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 – The algorithm uses the previous state to predict the current state based on a given system model.
- Correction – The algorithm uses the current sensor measurement to correct the state estimate.

The algorithm also periodically redistributes, or resamples, the particles in the state space to match the posterior distribution of the estimated state.

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

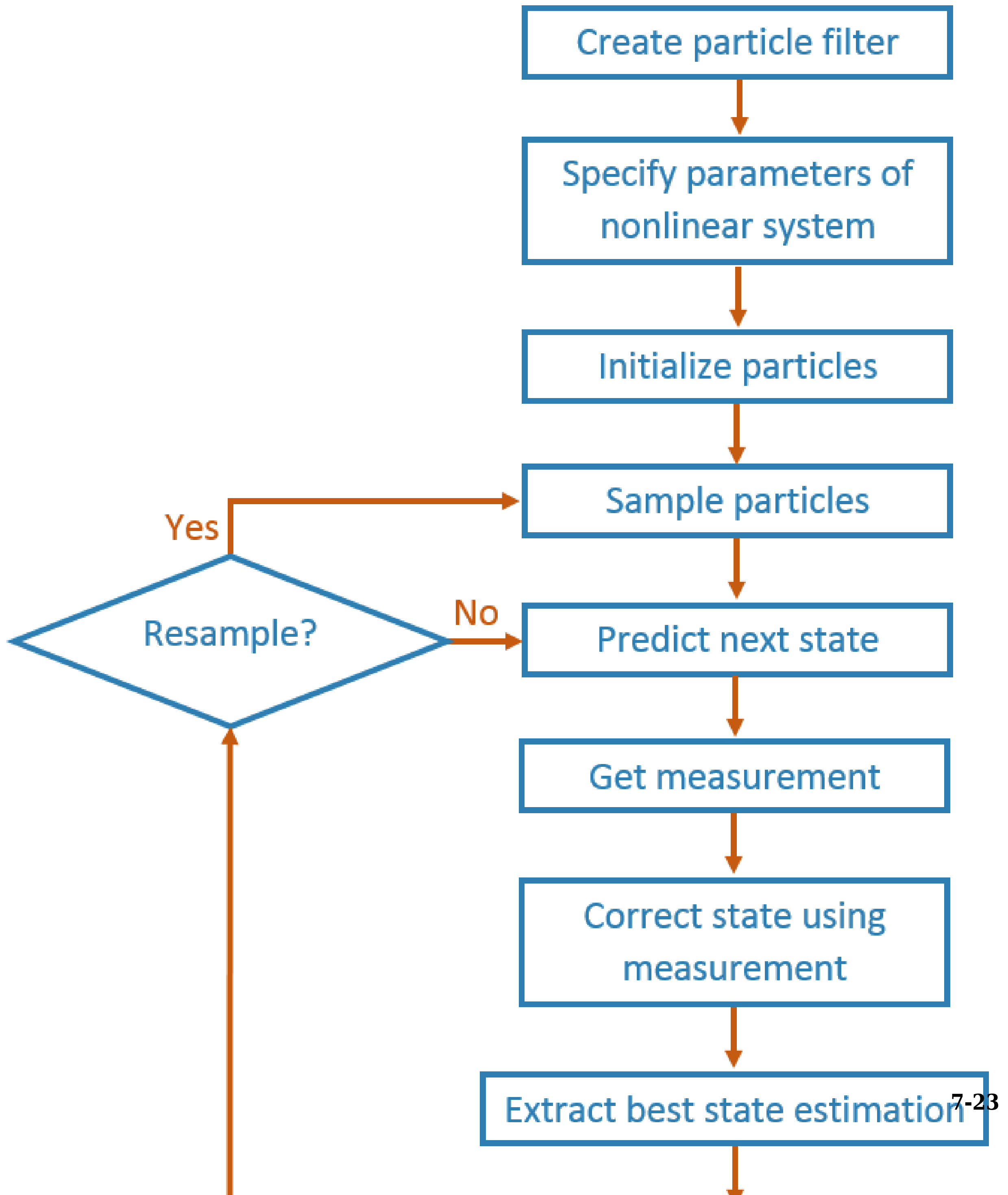
You can apply the particle filter to arbitrary nonlinear system models. Process and measurement noise can follow arbitrary non-Gaussian distributions.

To use the particle filter properly, you must specify parameters such as the number of particles, the initial particle location, and the state estimation method. Also, if you have a specific motion and sensor model, you specify these parameters in the state transition function and measurement likelihood function, respectively. For more information, see "Particle Filter Parameters" on page 7-14.

Follow this basic workflow to create and use a particle filter. This page details the estimation workflow and shows an example of how to run a particle filter in a loop to continuously estimate state.

Estimation Workflow

When using a particle filter, there is a required set of steps to create the particle filter and estimate state. The prediction and correction steps are the main iteration steps for continuously estimating state.


Create particle filter
Specify parameters of nonlinear system
Initialize particles
Sample particles
Yes
Resample?
No
Predict next state
Get measurement
Correct state using measurement
Extract best state estimation

Create Particle Filter

Create a `ParticleFilter` object by calling `robotics.ParticleFilter`.

Set Parameters of Nonlinear System

Modify these `ParticleFilter` parameters to fit for your specific system or application:

- `StateTransitionFcn`
- `MeasurementLikelihoodFcn`
- `ResamplingPolicy`
- `ResamplingMethod`
- `StateEstimationMethod`

Default values for these parameters are given for basic operation.

The `StateTransitionFcn` and `MeasurementLikelihoodFcn` functions define the system behavior and measurement integration. They are vital for the particle filter to track accurately. For more information, see “Particle Filter Parameters” on page 7-14.

Initialize Particles

Use the `initialize` method to set the number of particles and the initial state. See `robotics.ParticleFilter.initialize`.

Sample Particles from a Distribution

You can sample the initial particle locations in two ways:

- Initial pose and covariance — If you have an idea of your initial state, it is recommended you specify the initial pose and covariance. This specification helps to cluster particles closer to your estimate so tracking is more effective from the start.
- State bounds — If you do not know your initial state, you can specify the possible limits of each state variable. Particles are uniformly distributed across the state bounds for each variable. Widely distributed particles are not as effective at tracking, because fewer particles are near the actual state. Using state bounds usually requires more particles, computation time, and iterations to converge to the actual state estimate.

Predict

Based on a specified state transition function, particles evolve to estimate the next state. Use `predict` to execute the state transition function specified in the `StateTransitionFcn` property. See `robotics.ParticleFilter.predict`.

Get Measurement

The measurements collected from sensors are used in the next step to correct the current predicted state.

Correct

Measurements are then used to adjust the predicted state and correct the estimate. Specify your measurements using the `correct` function. `robotics.ParticleFilter.correct` uses the `MeasurementLikelihoodFcn` to calculate the likelihood of sensor measurements for each particle. Resampling of particles is required to update your estimation as the state changes in subsequent iterations. This step triggers resampling based on the `ResamplingMethod` and `ResamplingPolicy` properties.

Extract Best State Estimation

After calling `correct`, the best state estimate is automatically extracted based on the `Weights` of each particle and the `StateEstimationMethod` property specified in `robotics.ParticleFilter`. The best estimated state and covariance is output by the `correct` function.

Resample Particles

This step is not separately called, but is executed when you call `correct`. Once your state has changed enough, resample your particles based on the newest estimate. The `correct` method checks the `ResamplingPolicy` for the triggering of particle resampling according to the current distribution of particles and their weights. If resampling is not triggered, the same particles are used for the next estimation. If your state does not vary by much or if your time step is low, you can call the predict and correct methods without resampling.

Continuously Predict and Correct

Repeat the previous prediction and correction steps as needed for estimating state. The correction step determines if resampling of the particles is required. Multiple calls for `predict` or `correct` might be required when:

- No measurement is available but control inputs and time updates are occur at a high frequency. Use the `predict` method to evolve the particles to get the updated predicted state more often.
- Multiple measurement reading are available. Use `correct` to integrate multiple readings from the same or multiple sensors. The function corrects the state based on each set of information collected.

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;
```

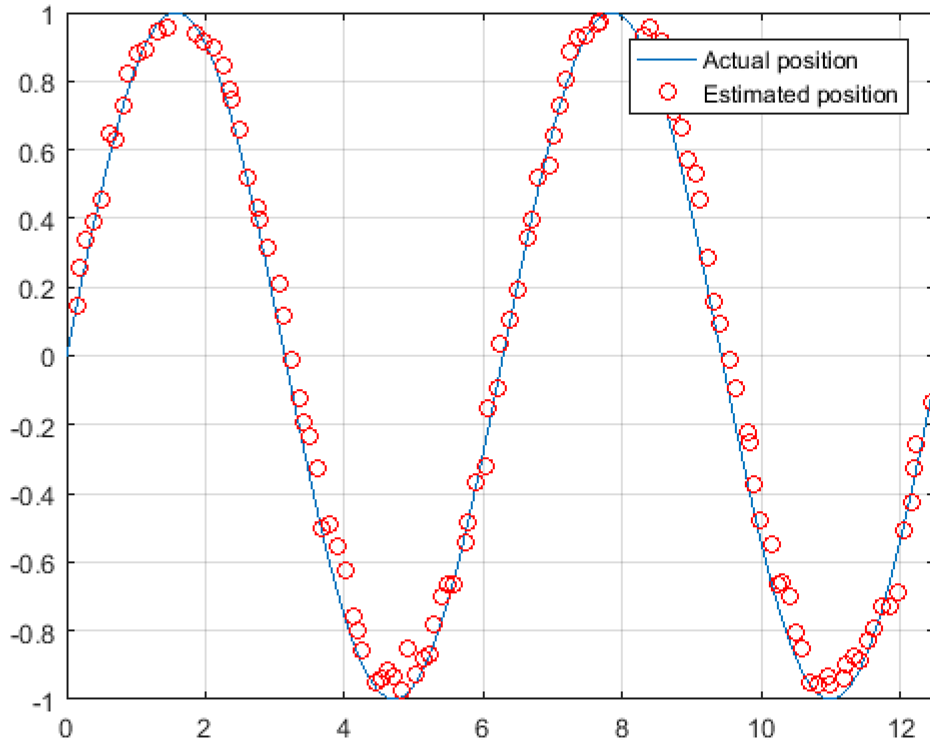
Begin the loop for predicting and correcting the estimated position based on measurements. The resampling of particles occurs based on the `ResamplingPolicy` property. The robot moves based on a sine wave function with random noise added to the measurement.

```
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.

See Also

`robotics.ParticleFilter` | `robotics.ParticleFilter.correct` | `robotics.ParticleFilter.initialize` | `robotics.ParticleFilter.predict`

Related Examples

- "Track a Car-Like Robot Using Particle Filter"

- “Estimate Robot Position in a Loop Using Particle Filter”

More About

- “Particle Filter Parameters” on page 7-14

Probabilistic Roadmaps (PRM)

In this section...
"Tune the Number of Nodes" on page 7-30
"Tune the Connection Distance" on page 7-34
"Create or Update PRM" on page 7-37

A probabilistic roadmap (PRM) is a network graph of possible paths in a given map based on free and occupied spaces. The `robotics.PRM` class randomly generates nodes and creates connections between these nodes based on the PRM algorithm parameters. Nodes are connected based on the obstacle locations specified in `Map`, and on the specified `ConnectionDistance`. You can customize the number of nodes, `NumNodes`, to fit the complexity of the map and the desire to find the most efficient path. The PRM algorithm uses the network of connected nodes to find an obstacle-free path from a start to an end location. To plan a path through an environment effectively, tune the `NumNodes` and `ConnectionDistance` properties.

When creating or updating the `robotics.PRM` class, the node locations are randomly generated, which can affect your final path between multiple iterations. This selection of nodes occurs when you specify `Map` initially, change the parameters, or `update` is called. To get consistent results with the same node placement, use `rng` to save the state of the random number generation. See "Tune the Connection Distance" on page 7-34 for an example using `rng`.

Tune the Number of Nodes

Use the `NumNodes` property on the `PRM` object to tune the algorithm. `NumNodes` specifies the number of points, or nodes, placed on the map, which the algorithm uses to generate a roadmap. Using the `ConnectionDistance` property as a threshold for distance, the algorithm connects all points that do not have obstacles blocking the direct path between them.

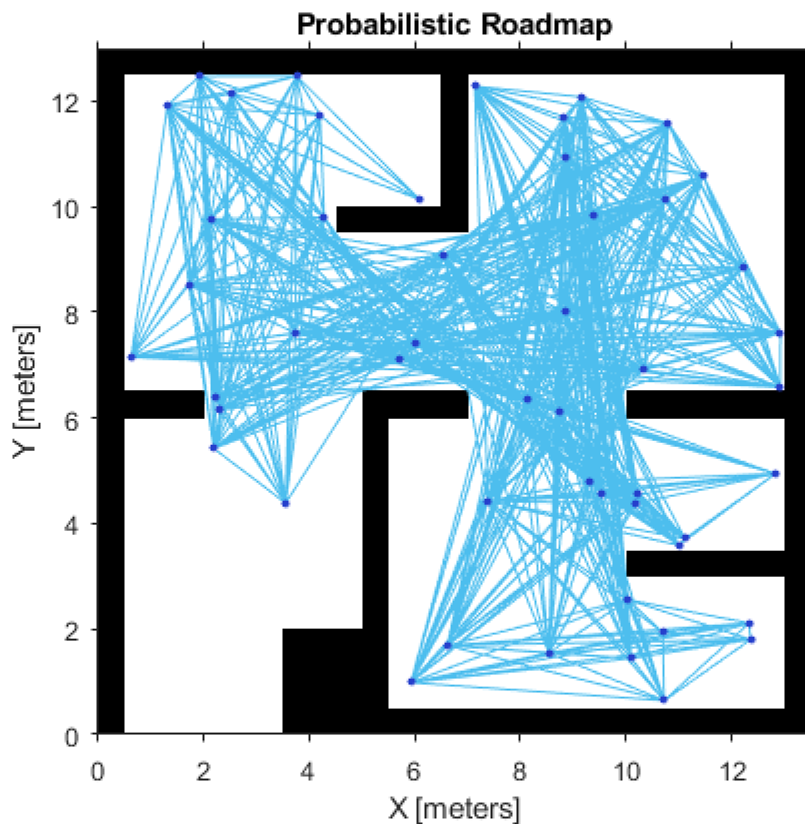
Increasing the number of nodes can increase the efficiency of the path by giving more feasible paths. However, the increased complexity increases computation time. To get good coverage of the map, you might need a large number of nodes. Due to the random placement of nodes, some areas of the map may not have enough nodes to connect to the rest of the map. In this example, you create a large and small number of nodes in a roadmap.

Load a map file as a logical matrix, `simpleMaps`, and create an occupancy grid.

```
load exampleMaps.mat
map = robotics.OccupancyGrid(simpleMap,2);
```

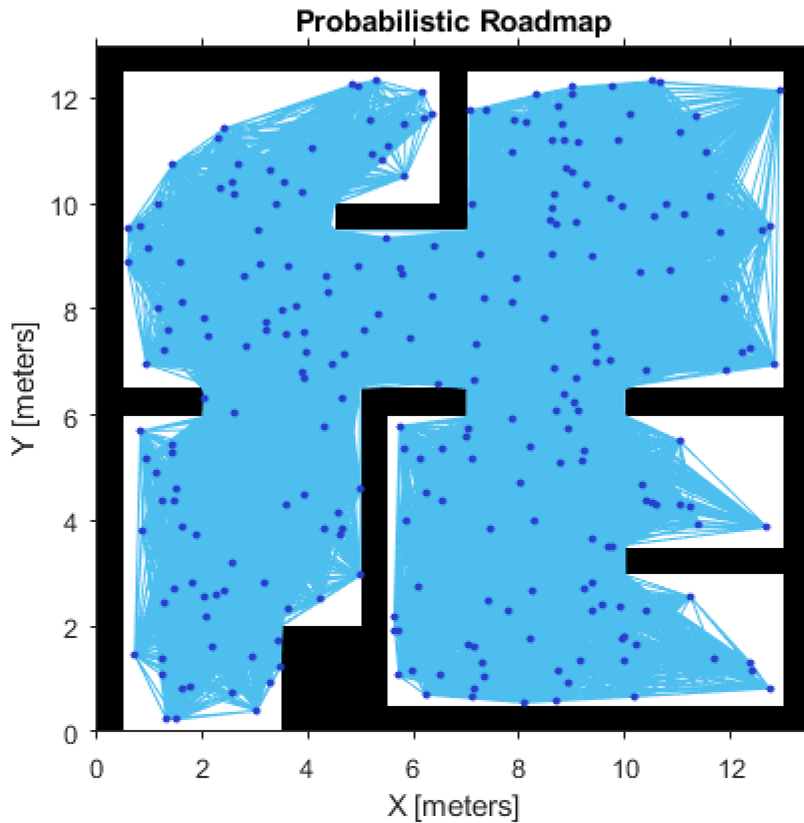
Create a simple roadmap with 50 nodes.

```
prmSimple = robotics.PRM(map,50);
show(prmSimple)
```



Create a dense roadmap with 250 nodes.

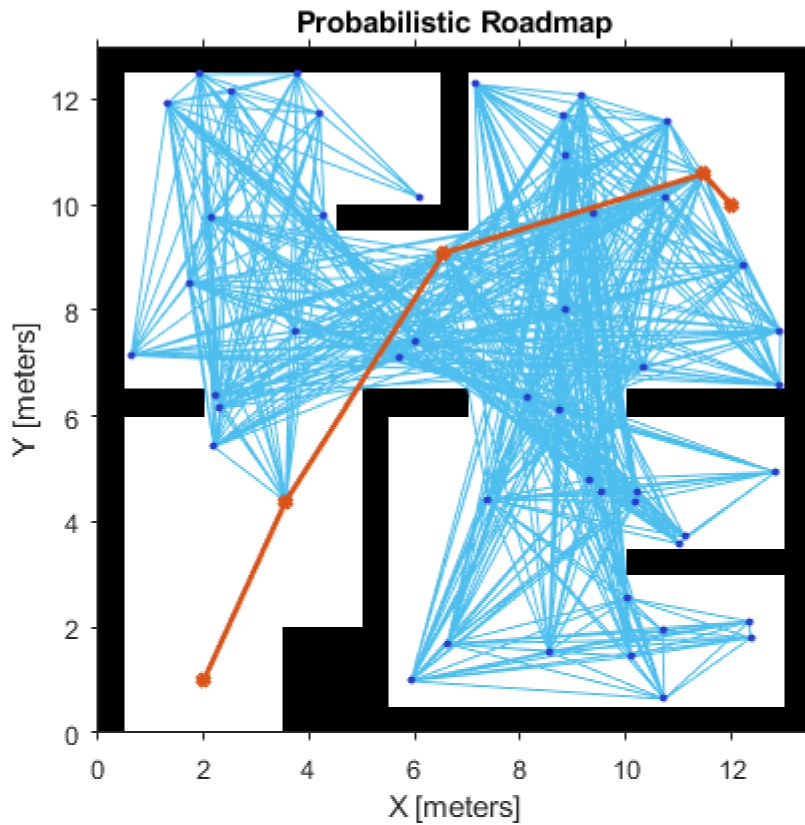
```
prmComplex = robotics.PRM(map,250);
show(prmComplex)
```



The additional nodes increase the complexity but yield more options to improve the path. Given these two maps, you can calculate a path using the PRM algorithm and see the effects.

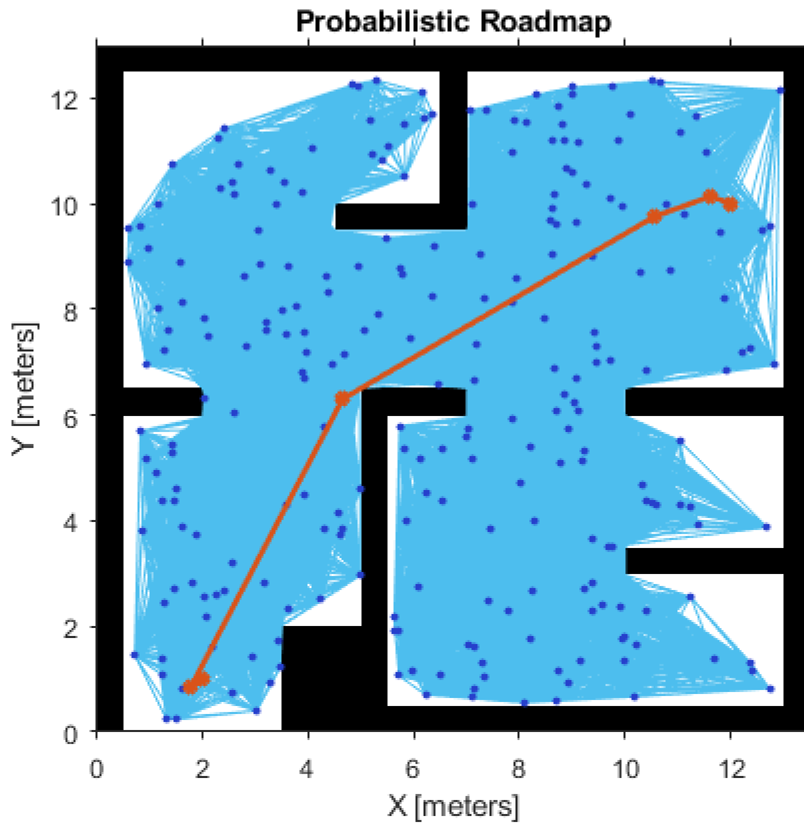
Calculate a simple path.

```
startLocation = [2 1];
endLocation = [12 10];
path = findpath(prmSimple,startLocation,endLocation);
show(prmSimple)
```



Calculate a complex path.

```
path = findpath(prmComplex, startLocation, endLocation);
show(prmComplex)
```



Increasing the nodes allows for a more direct path, but adds more computation time to finding a feasible path. Because of the random placement of points, the path is not always more direct or efficient. Using a small number of nodes can make paths worse than depicted and even restrict the ability to find a complete path.

Tune the Connection Distance

Use the `ConnectionDistance` property on the `PRM` object to tune the algorithm. `ConnectionDistance` is an upper threshold for points that are connected in the roadmap. Each node is connected to all nodes within this connection distance that do not have obstacles between them. By lowering the connection distance, you can limit the number of connections to reduce the computation time and simplify the map. However, a

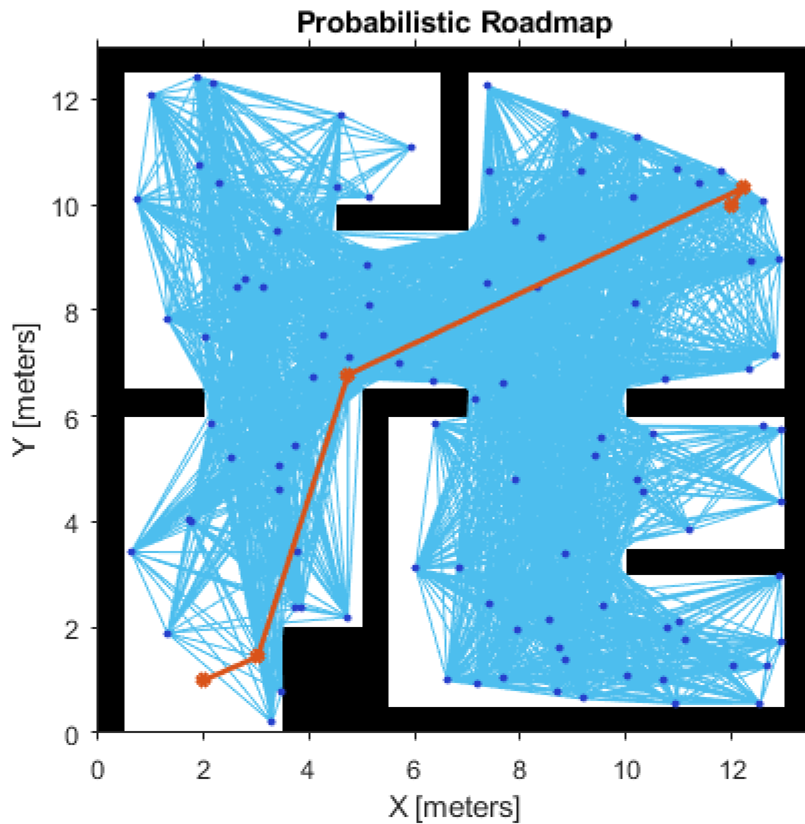
lowered distance limits the number of available paths from which to find a complete obstacle-free path. When working with simple maps, you can use a higher connection distance with a small number of nodes to increase efficiency. For complex maps with lots of obstacles, a higher number of nodes with a lowered connection distance increases the chance of finding a solution.

Load a map as a logical matrix, `simpleMap`, and create an occupancy grid.

```
load exampleMaps.mat
map = robotics.OccupancyGrid(simpleMap,2);
```

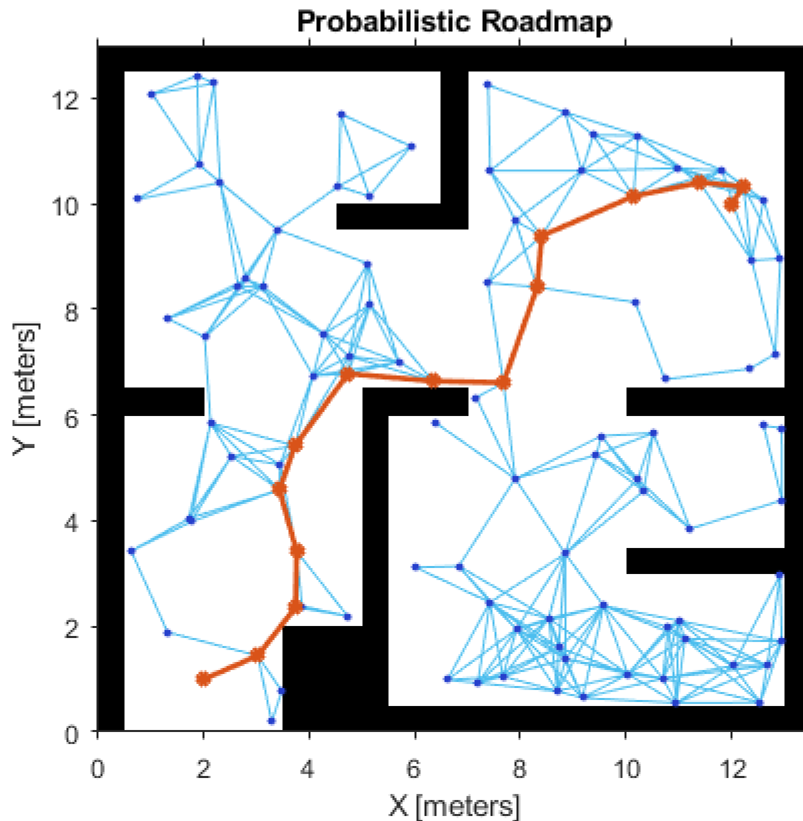
Create a roadmap with 100 nodes and calculate the path. The default `ConnectionDistance` is set to inf. Save the random number generation settings using the rng function. The saved settings enable you to reproduce the same points and see the effect of changing `ConnectionDistance`.

```
rngState = rng;
prm = robotics.PRM(map,100);
startLocation = [2 1];
endLocation = [12 10];
path = findpath(prm,startLocation,endLocation);
show(prm)
```



Reload the random number generation settings to have PRM use the same nodes. Lower `ConnectionDistance` to 2 m. Show the calculated path.

```
rng(rngState);
prm.ConnectionDistance = 2;
path = findpath(prm,startLocation,endLocation);
show(prm)
```



Create or Update PRM

Create or update your roadmap. To create the roadmap, call `prm = robotics.PRM(map,___)` or specify the `Map` property on the `PRM` object. Then, call the `update`, `findpath`, or `show` method. At this point, the nodes are randomly generated and the connections are made.

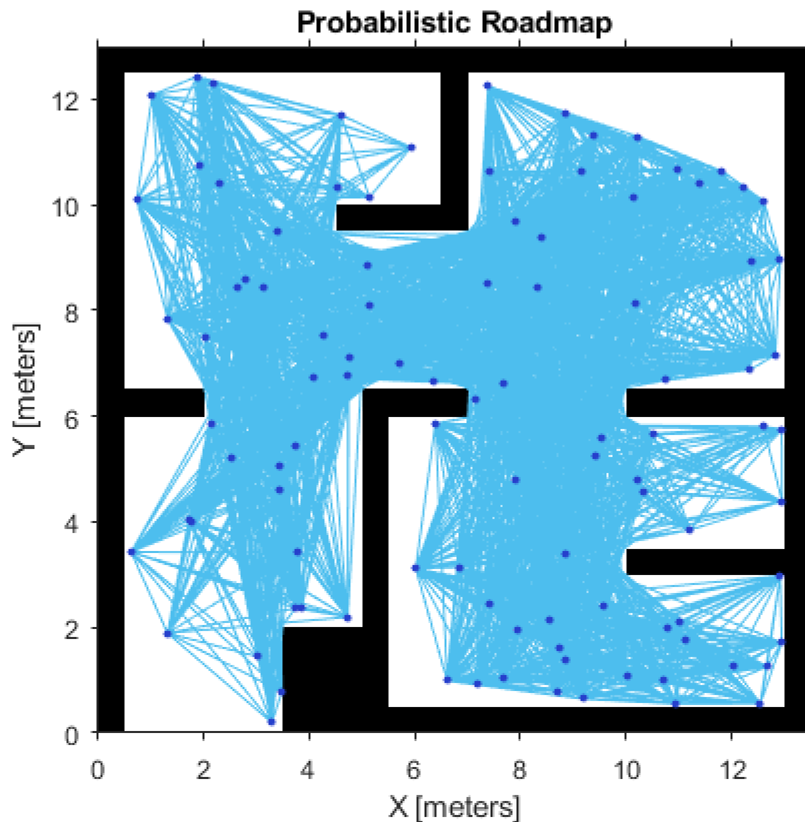
This roadmap changes only if you call update or change the properties in the `PRM` object. When properties change, any method (`update`, `findpath`, or `show`) called on the object triggers the roadmap points and connections to be recalculated. Because recalculating the map can be computationally intensive, you can reuse the same roadmap by calling `findpath` with different starting and ending locations.

Load the map (`simpleMap`) from a `.mat` file as a logical matrix and create an occupancy grid.

```
load('exampleMaps.mat')
map = robotics.BinaryOccupancyGrid(simpleMap,2);
```

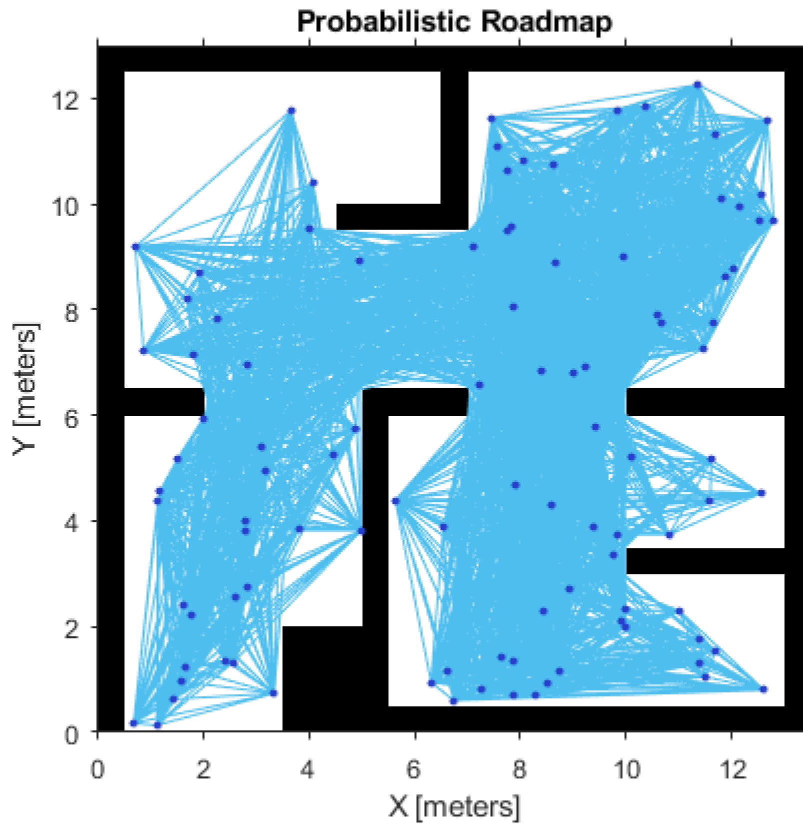
Create a roadmap. Your nodes and connections might look different due to the random placement of nodes.

```
prm = robotics.PRM(map,100);
show(prm)
```



Call `update` or change a parameter to update the `PRM` nodes and connections.

```
update(prm)
show(prm)
```



The PRM algorithm recalculates the node placement and generates a new network of nodes.

References

[1] Kavraki, L.E., P. Svestka, J.-C. Latombe, and M.H. Overmars. "Probabilistic roadmaps for path planning in high-dimensional configuration spaces," *IEEE Transactions on Robotics and Automation*. Vol. 12, No. 4, Aug 1996 pp. 566—580.

See Also

`robotics.PRM` | `robotics.PRM.findpath` | `robotics.PRM.show` | `robotics.PRM.update`

Pure Pursuit Controller

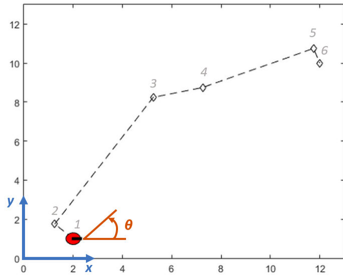
In this section...
“Reference Coordinate System” on page 7-41
“Look Ahead Distance” on page 7-42
“Limitations” on page 7-43

PurePursuit is a path tracking algorithm. It computes the angular velocity command that moves the robot from its current position to reach some look-ahead point in front of the robot. The linear velocity is assumed constant, hence you can change the linear velocity of the robot at any point. The algorithm then moves the look-ahead point on the path based on the current position of the robot until the last point of the path. You can think of this as the robot constantly chasing a point in front of it. The property LookAheadDistance decides how far the look-ahead point is placed.

The `PurePursuit` class (`robotics.PurePursuit`) is not a traditional controller, but acts as a tracking algorithm for path following purposes. In the Robotics System Toolbox, you create a `PurePursuit` controller and specify a list of waypoints. The desired linear and maximum angular velocities can be specified. These properties are determined based on the robot’s specifications. Given the pose (position and orientation) of the robot as an input, the object can be used to calculate the linear and angular velocities commands for the robot. How the robot uses these commands is dependent on the system you are using, so consider how robots can execute a motion given these commands. The final important property is the `LookAheadDistance`, which tells the robot how far along on the path to track towards. This property is explained in more detail in a section below.

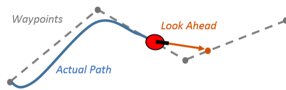
Reference Coordinate System

It is important to understand the reference coordinate frame used by the `PurePursuit` algorithm for its inputs and outputs. The figure below shows the reference coordinate system. The input waypoints are `[x y]` coordinates, which are used to compute the robot velocity commands. The robot’s pose is input as a pose and orientation (theta) list of points as `[x y theta]`. The positive *x* and *y* directions are in the right and up directions respectively (blue in figure). The *theta* value is the angular orientation of the robot measured counterclockwise in radians from the *x*-axis (robot currently at `0` radians).

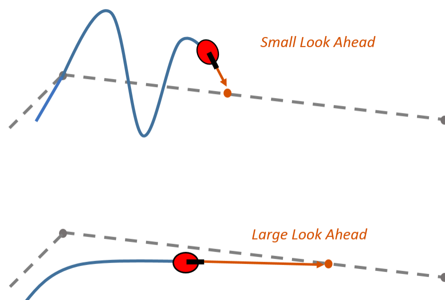


Look Ahead Distance

The `LookAheadDistance` property is the main tuning property for the `PurePursuit` controller. The look ahead distance is how far along the path the robot should look from the current location to compute the angular velocity commands. The figure below shows the robot and the look-ahead point. As displayed in this image, note that the actual path does not match the direct line between waypoints.



The effect of changing this parameter can change how your robot tracks the path and there are two major goals: regaining the path and maintaining the path. In order to quickly regain the path between waypoints, a small `LookAheadDistance` will cause your robot to move quickly towards the path. However, as can be seen in the figure below, the robot overshoots the path and oscillates along the desired path. In order to reduce the oscillations along the path, a larger look ahead distance can be chosen, however, it might result in larger curvatures near the corners.



The `LookAheadDistance` property should be tuned for your application and robot system. Different linear and angular velocities will affect this response as well and should be considered for the path following controller.

Limitations

There are a few limitations to note about this `PurePursuit` algorithm:

- As shown above, the controller cannot exactly follow direct paths between waypoints. Parameters must be tuned to optimize the performance and to converge to the path over time.
- This `PurePursuit` algorithm does not stabilize the robot at a point. In your application, a distance threshold for a goal location should be applied to stop the robot near the desired goal. This can be seen in the path following example: “Path Following for a Differential Drive Robot”.

References

[1] Coulter, R. *Implementation of the Pure Pursuit Path Tracking Algorithm*. Carnegie Mellon University, Pittsburgh, Pennsylvania, Jan 1990.

See Also

`robotics.PRM` | `robotics.PurePursuit` | `robotics.VectorFieldHistogram`

Related Examples

- “Path Following for a Differential Drive Robot”

Vector Field Histogram

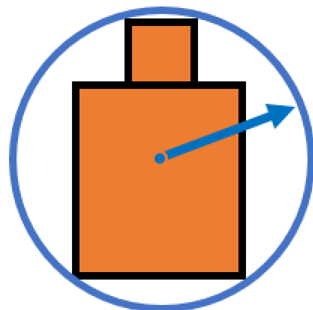
In this section...
“Robot Dimensions” on page 7-44
“Cost Function Weights” on page 7-46
“Histogram Properties” on page 7-47
“Tune Parameters Using show” on page 7-51

The vector field histogram (VFH) algorithm computes obstacle-free steering directions for a robot based on range sensor readings. Range sensor readings are used to compute polar density histograms to identify obstacle location and proximity. Based on the specified parameters and thresholds, these histograms are converted to binary histograms to indicate valid steering directions for the robot. The VFH algorithm factors in robot size and turning radius to output a steering direction for the robot to avoid obstacles and follow a target direction.

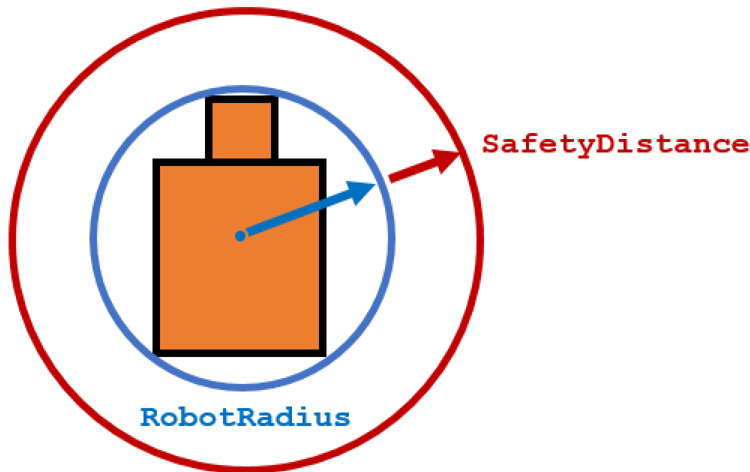
Robot Dimensions

To calculate steering directions, you must specify information about the robot size and its driving capabilities. The VFH algorithm requires only four input parameters for the robot. These parameters are properties of the `robotics.VectorFieldHistogram` class: `RobotRadius`, `SafetyDistance`, `MinTurningRadius`, and `DistanceLimits`.

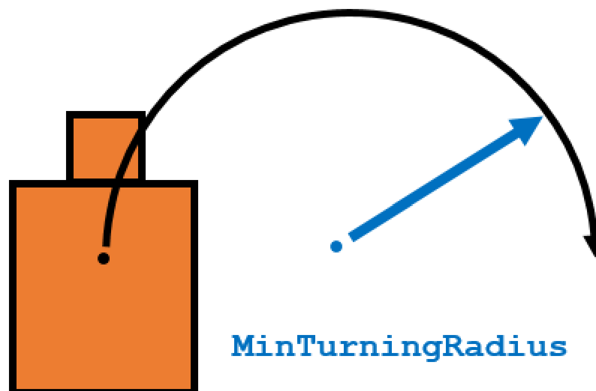
- `RobotRadius` specifies the radius of the smallest circle that can encircle all parts of the robot. This radius ensures that the robot avoids obstacles based on its size.



- `SafetyDistance` optionally specifies an added distance on top of the `RobotRadius`. You can use this property to add a factor of safety when navigating an environment.

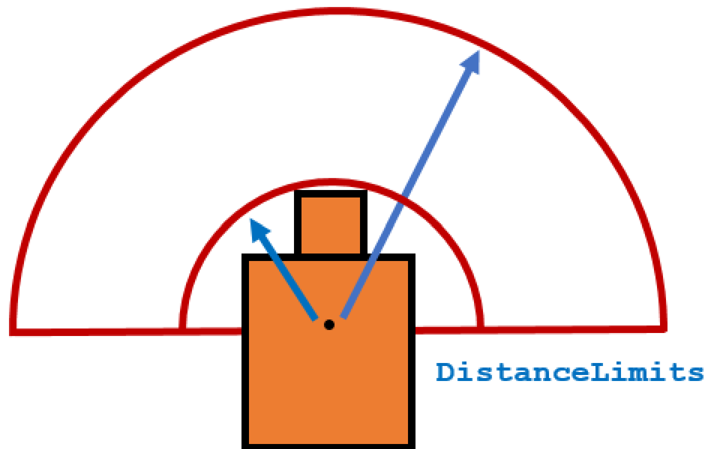


- `MinTurningRadius` specifies the minimum turning radius for the robot traveling at the desired velocity. The robot may not be able to make sharp turns at high velocities. This property factors in navigating around obstacles and gives it enough space to maneuver.



- `DistanceLimits` specifies the distance range that you want to consider for obstacle avoidance. You specify the limits in a two-element vector, `[lower upper]`. The `lower`

limit is used to ignore sensor readings that intersect with parts on the robot, sensor inaccuracies at short distances, or sensor noise. The `upper` limit is the effective range of the sensor or is based on your application. You might not want to consider all obstacles in the full sensor range.



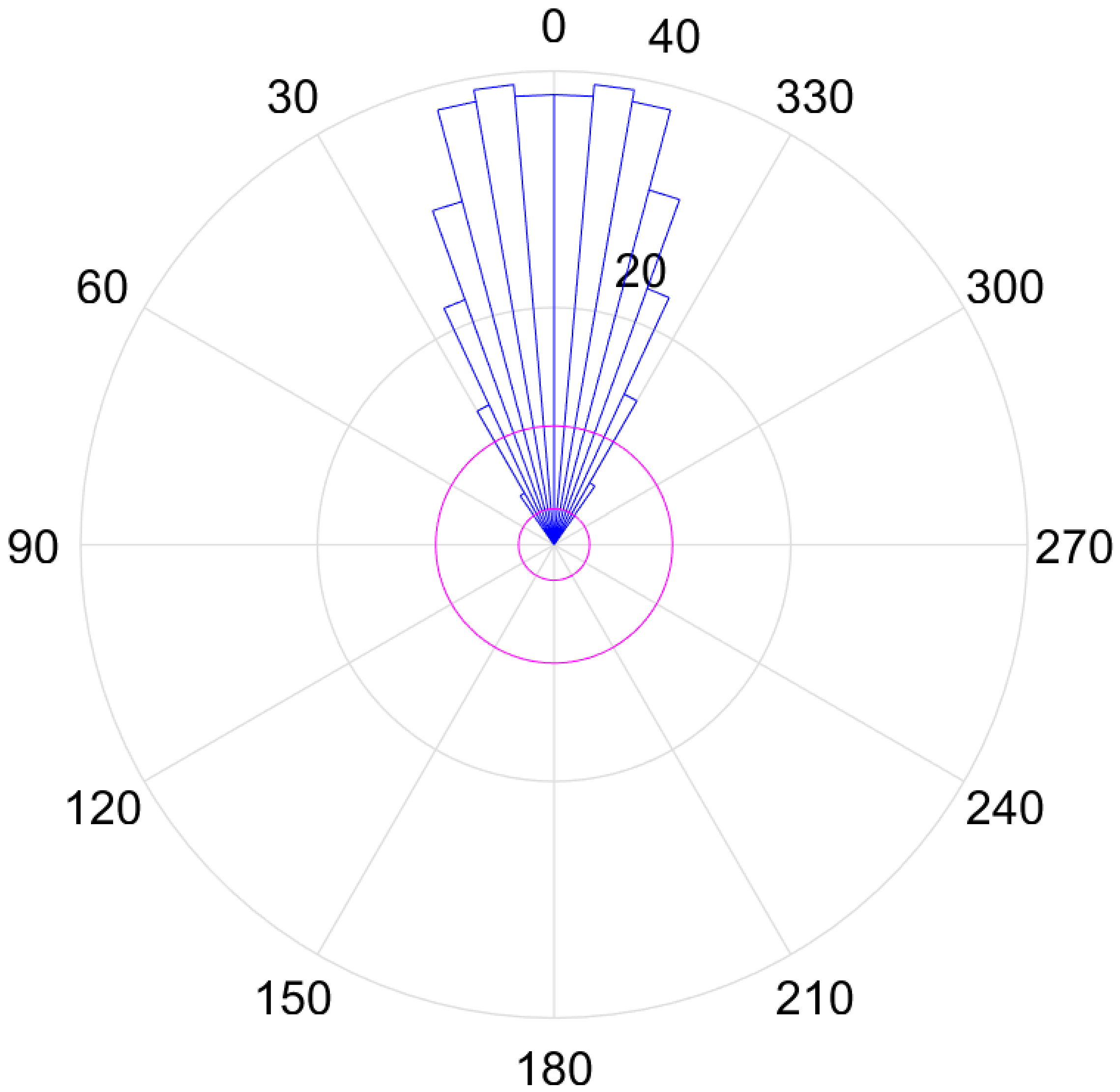
Note All information about the range sensor readings assumes that your range finder is mounted in the center of your robot. If the range sensor is mounted elsewhere, transform your range sensor readings from the laser coordinate frame to the robot base frame. See "Transform Laser Scan Data From A ROS Network" on page 9-2 for an example.

Cost Function Weights

Cost function weights are used to calculate the final steering directions. The VFH algorithm considers multiple steering directions based on your current, previous, and target directions. By setting the `CurrentDirectionWeight`, `PreviousDirectionWeight`, and `TargetDirectionWeight` properties, you can modify the steering behavior of your robot. Changing these weights affects the responsiveness of the robot and how it reacts to obstacles. To make the robot head towards its goal location, set `TargetDirectionWeight` higher than the sum of the other weights. This high `TargetDirectionWeight` value helps to ensure the computed steering direction is close to the target direction. Depending on your application, you might need to tune these weights.

Histogram Properties

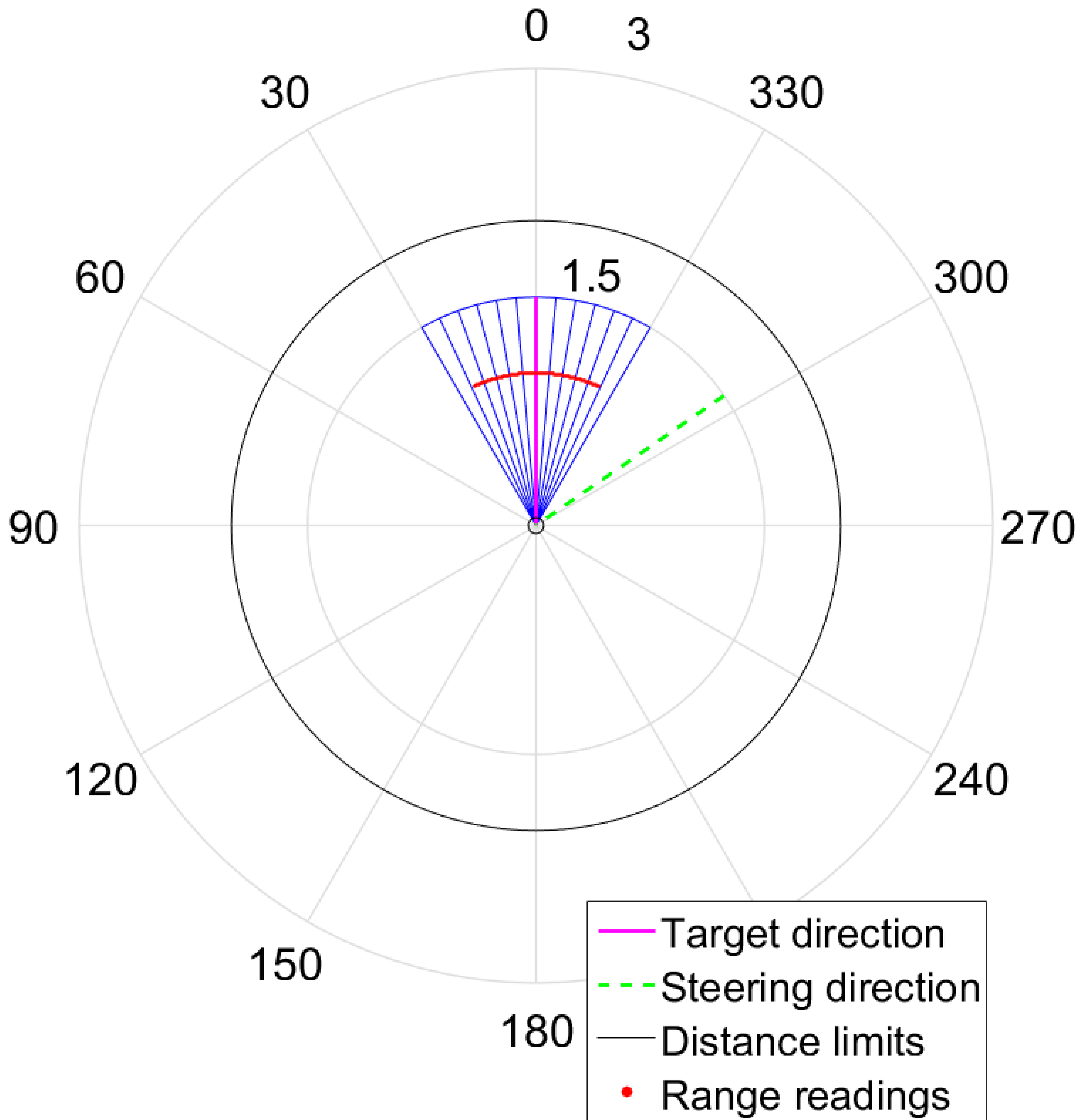
The VFH algorithm calculates a histogram based on the given range sensor data. It takes all directions around the robot and converts them to angular sectors that are specified by the `NumAngularSectors` property. This property is non-tunable and remains fixed once the `robotics.VectorFieldHistogram` object is called. The range sensor data is used to calculate a polar density histogram over these angular sectors.


Polar Obstacle Density
0
40
30
330
20
60
300
90
270
120
240
150
210
180

Note Using a small `NumAngularSectors` value can cause the VFH algorithm to miss smaller obstacles. Missed obstacles do not appear on the histogram.

This histogram displays the angular sectors in blue and the histogram thresholds in pink. The `HistogramThresholds` property is a two-element vector that determines the values of the masked histogram, specified as `[lower upper]`. Polar obstacle density values higher than the upper threshold are represented as occupied space (1) in the masked 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). The masked histogram also factors in the `MinTurningRadius`, `RobotSize`, and `SafetyDistance`.

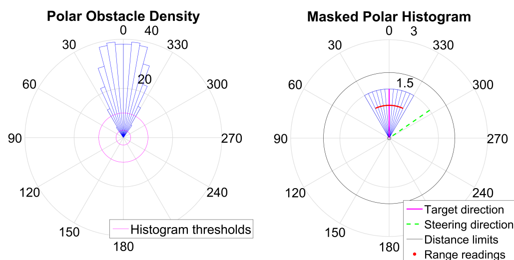
The polar density plot has the following corresponding masked histogram plot. This plot shows the target and steering directions, range readings, and distance limits.



Masked Polar Histogram
0
3
30
330
60
1.5
300
90
270
120
240
150
180
Target direction
Steering direction
Distance limits
Range readings

Tune Parameters Using show

When working with a `robotics.VectorFieldHistogram` object, you can visualize the properties and parameters of the algorithm using the `robotics.VectorFieldHistogram.show` method. This method displays the polar density plot and masked binary histogram. It also displays the algorithm parameters and the output steering direction for the VFH.



You can then tune parameters to help you prototype your obstacle avoidance application. For example, if you see that certain obstacles do not appear in the **Masked Polar Histogram** plot (right), then in the **Polar Obstacle Density** plot, consider adjusting the histogram thresholds to appropriate values. After you make the adjustments in the **Masked Polar Histogram** plot, the range sensor readings, shown in red, should match up with locations in the masked histogram (blue). Also, you can see the target and steering directions. You specify the target direction. The steering direction is the main output from the VFH algorithm. Adjusting the “Cost Function Weights” on page 7-46 can help you tune the output of the final steering direction.

Although you can use the `show` method in a loop, it slows computation speed due to the graphical plotting. If you are running this algorithm for real-time applications, get and display the VFH data in separate operations.

See Also

`robotics.VectorFieldHistogram` | `robotics.VectorFieldHistogram.show`

Monte Carlo Localization Algorithm

In this section...

"Overview" on page 7-52
"State Representation" on page 7-53
"Initialization of Particles" on page 7-55
"Resampling Particles and Updating Pose" on page 7-57
"Motion and Sensor Model" on page 7-58

Overview

The Monte Carlo Localization (MCL) algorithm is used to estimate the position and orientation of a robot. The algorithm uses a known map of the environment, range sensor data, and odometry sensor data. To see how to construct an object and use this algorithm, see `robotics.MonteCarloLocalization`.

To localize the robot, the MCL algorithm uses a particle filter to estimate its position. The particles represent the distribution of the likely states for the robot. 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. The robot motion is sensed using an odometry sensor.

The particles are updated in this process:

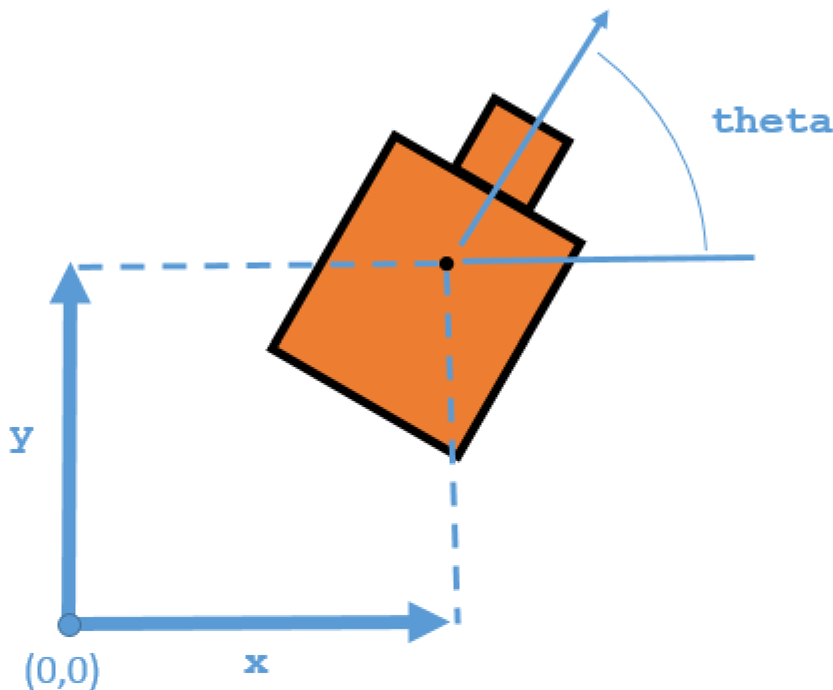
1. Particles are propagated based on the change in the pose and the specified motion model, `MotionModel`.
2. The particles are assigned weights based on the likelihood of receiving the range sensor reading for each particle. This reading is based on the sensor model you specify in `SensorModel`.
3. Based on these weights, a robot state estimate is extracted based on the particle weights. The group of particles with the highest weight is used to estimate the position of the robot.
4. Finally, the particles are resampled based on the specified `ResamplingInterval`. Resampling adjusts particle positions and improves performance by adjusting the number of particles used. It is a key feature for adjusting to changes and keeping particles relevant for estimating the robot state.

The algorithm outputs the estimated pose and covariance. These estimates are the mean and covariance of the highest weighted cluster of particles. For continuous tracking, repeat these steps in a loop to propagate particles, evaluate their likelihood, and get the best state estimate.

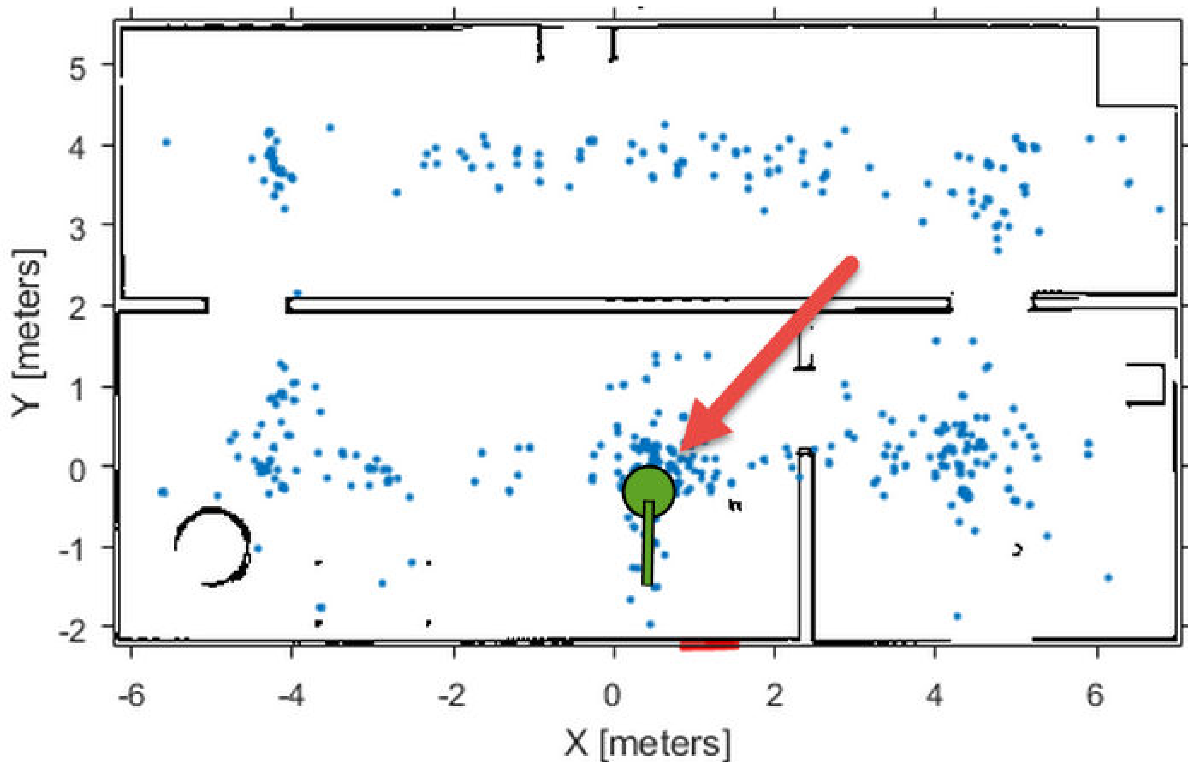
For more information on particle filters as a general application, see “Particle Filter Workflow” on page 7-21.

State Representation

When working with a localization algorithm, the goal is to estimate the state of your system. For robotics applications, this estimated state is usually a robot pose. For the `MonteCarloLocalization` object, you specify this pose as a three-element vector. The pose corresponds to an *x*-*y* position, `[x y]`, and an angular orientation, `theta`.



The MCL algorithm estimates these three values based on sensor inputs of the environment and a given motion model of your system. The output from using the `MonteCarloLocalization` object includes the `pose`, which is the best estimated state of the `[x y theta]` values. Particles are distributed around an initial pose, `InitialPose`, or sampled uniformly using global localization. The pose is computed as the mean of the highest weighted cluster of particles once these particles have been corrected based on measurements.



This plot shows the highest weighted cluster and the final robot pose displayed over the samples particles in green. With more iterations of the MCL algorithm and measurement corrections, the particles converge to the true location of the robot. However, it is possible that particle clusters can have high weights for false estimates and converge on the wrong location. If the wrong convergence occurs, resample the particles by resetting the MCL algorithm with an updated `InitialPose`.

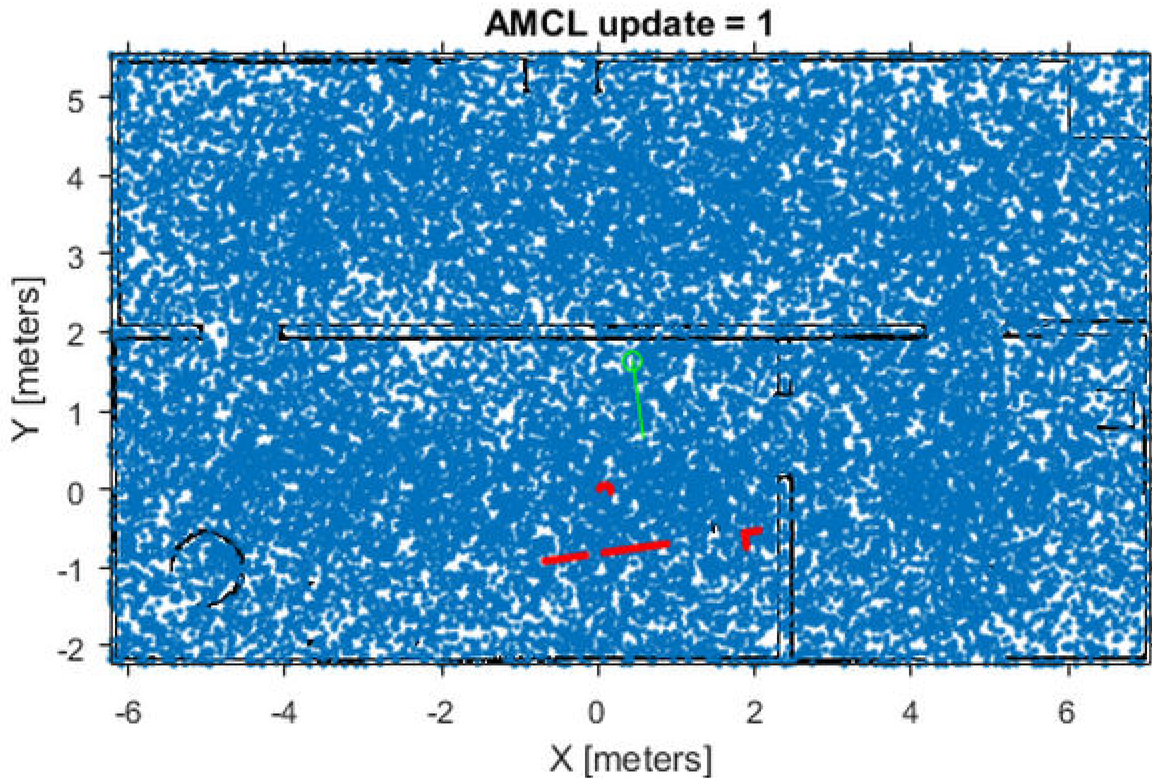
Initialization of Particles

When you first create the `MonteCarloLocalization` algorithm, specify the minimum and maximum particle limits by using the `ParticleLimits` property. A higher number of particles increases the likelihood that the particles converge on the actual location. However, a lower particle number is faster. The number of particles adjusts dynamically within the limits based on the weights of particle clusters. This adjustment helps to reduce the number of particles over time so localization can run more efficiently.

Particle Distribution

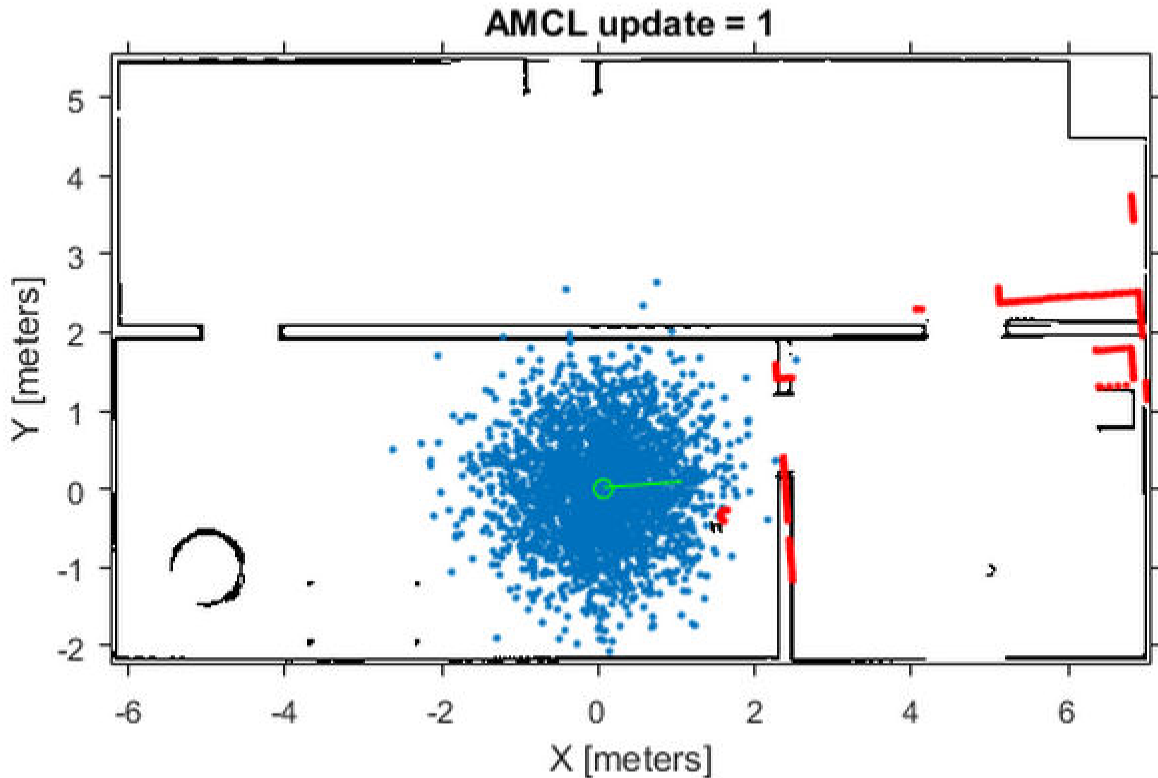
Particles must be sampled across a specified distribution. To initialize particles in the state space, you can use either an initial pose or global localization. With global localization, you can uniformly distribute particles across your expected state space (pulled from the `Map` property of your`SensorModel` object). In the default MCL object, set the `GlobalLocalization` property to `true`.

```
mcl = robotics.MonteCarloLocalization;
mcl.GlobalLocalization = true;
```



Global localization requires a larger number of particles to effectively sample particles across the state space. More particles increase the likelihood of successful convergence on the actual state. This large distribution greatly reduces initial performance until particles begin to converge and particle number can be reduced.

By default, global localization is set to `false`. Without global localization, you must specify the `InitialPose` and `InitialCovariance` properties, which helps to localize the particles. Using this initial pose, particles are more closely grouped around an estimated state. A close grouping of particles enables you to use fewer of them, and increases the speed and accuracy of tracking during the first iterations.



These images were taken from the “Localize TurtleBot Using Monte Carlo Localization” example, which shows how to use the MCL algorithm with the TurtleBot® in a known environment.

Resampling Particles and Updating Pose

To localize your robot continuously, you must resample the particles and update the algorithm. Use the `UpdateThreshold` and `ResamplingInterval` properties to control when resampling and updates to the estimated state occur.

The `UpdateThreshold` is a three-element vector that defines the minimum change in the robot pose, `[x y theta]`, to trigger an update. Changing a variable by more than this minimum triggers an update, causing the object to return a new state estimate. This change in robot pose is based on the odometry, which is specified in the functional form of

the object. Tune these thresholds based on your sensor properties and the motion of your robot. Random noise or minor variations greater than your threshold can trigger an unnecessary update and affect your performance. The `ResamplingInterval` property defines the number of updates to trigger particle resampling. For example, a resampling interval of `2` resamples at every other update.

The benefit of resampling particles is that you update the possible locations that contribute to the final estimate. Resampling redistributes the particles based on their weights and evolves particles based on the “Motion Model” on page 7-60. In this process, the particles with lower weight are eliminated, helping the particles converge to the true state of the robot. The number of particles dynamically changes to improve speed or tracking.

The performance of the algorithm depends on proper resampling. If particles are widely dispersed and the initial pose of the robot is not known, the algorithm maintains a high particle count. As the algorithm converges on the true location, it reduces the number of particles and increases the speed of performance. You can tune your `ParticleLimits` property to limit the minimum and maximum particles used to help with the performance.

Motion and Sensor Model

The motion and sensor models for the MCL algorithm are similar to the `StateTransitionFcn` and `MeasurementLikelihoodFcn` functions for the `robotics.ParticleFilter` object, which are described in “Particle Filter Parameters” on page 7-14. For the MCL algorithm, these models are more specific to robot localization. After calling the object, to change the `MotionModel` or `SensorModel` properties, you must first call `release` on your object.

Sensor Model

By default, the `MonteCarloLocalization` uses a `robotics.LikelihoodFieldSensorModel` object as the sensor model. This sensor model contains parameters specific to the range sensor used, 2-D map information for the robot environment, and measurement noise characteristics. The sensor model uses the parameters with range measurements to compute the likelihood of the measurements given the current position of the robot. Without factoring in these parameters, some measurement errors can skew the state estimate or increase weight on irrelevant particles.

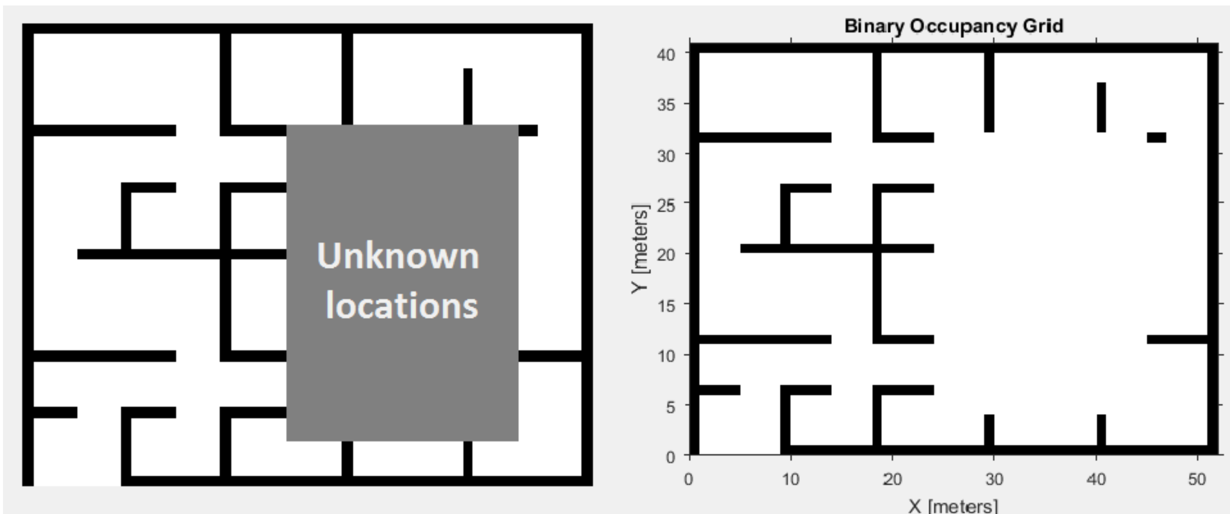
The range sensor properties are:

- `SensorPose` – The pose of the range sensor relative to the robot location. This pose is used to transform the range readings into the robot coordinate frame.
- `SensorLimits` – The minimum and maximum range limits. Measurement outside of these ranges are not factored into the likelihood calculation.
- `NumBeams` – Number of beams used to calculate likelihood. You can improve performance speed by reducing the number of beams used.

Range measurements are also known to give false readings due to system noise or other environmental interference. To account for the sensor error, specify these parameters:

- `MeasurementNoise` – Standard deviation for measurement noise. This deviation applies to the range reading and accounts for any interference with the sensor. Set this value based on information from your range sensor.
- `RandomMeasurementWeight` — Weight for probability of random measurement. Set a low probability for random measurements. The default is 0.05.
- `ExpectedMeasurementWeight` — Weight for probability of expected measurement. Set a high probability for expected measurements. The default is 0.95.

The sensor model also stores a map of the robot environment as an occupancy grid. Use `robotics.BinaryOccupancyGrid` to specify your map with occupied and free spaces. Set any unknown spaces in the map as free locations. Setting them to free locations prevents the algorithm from matching detected objects to these areas of the map.



Also, you can specify `MaximumLikelihoodDistance`, which limits the area for searching for obstacles. The value of `MaximumLikelihoodDistance` is the maximum distance to the nearest obstacle that is used for likelihood computation.

Motion Model

The motion model for robot localization helps to predict how particles evolve throughout time when resampling. It is a representation of robot kinematics. The motion model included by default with the MCL algorithm is an odometry-based differential drive motion model (`robotics.OdometryMotionModel`). Without a motion model, predicting the next step is more difficult. It is important to know the capabilities of your system so that the localization algorithm can plan particle distributions to get better state estimates. Be sure to consider errors from the wheel encoders or other sensors used to measure the odometry. The errors in the system define the spread of the particle distribution.

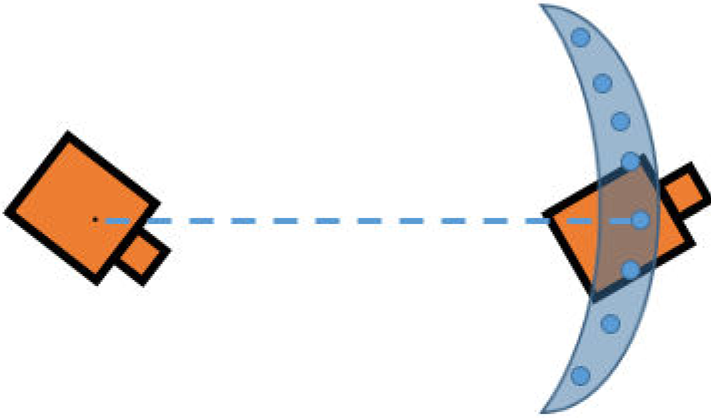
You can specify the error expected based on the motion of your robot as a four-element vector, `Noise`. These four elements are specified as weights on the standard deviations for [1]:

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

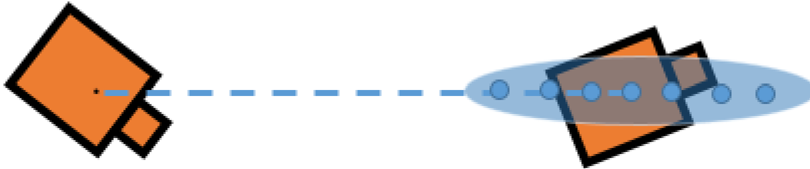
For differential drive robots, when a robot moves from a starting pose to a final pose, the change in pose can be treated as:

1. Rotation to the final position
2. Translation in a direct line to the final position
3. Rotation to the goal orientation

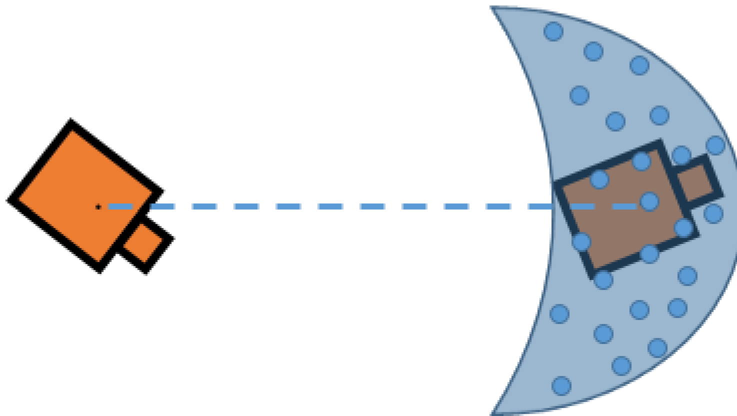
Assuming these steps, you can visualize the effect of errors in rotation and translation. Errors in the initial rotation result in your possible positions being spread out in a C-shape around the final position.



Large translational errors result in your possible positions being spread out around the direct line to the final position.



Large errors in both translation and rotation can result in wider-spread positions.



Also, rotational errors affect the orientation of the final pose. Understanding these effects helps you to define the Gaussian noise in the `Noise` property of the `MotionModel` object for your specific application. As the images show, each parameter does not directly control the dispersion and can vary with your robot configuration and geometry. Also, multiple pose changes as the robot navigates through your environment can increase the effects of these errors over many different steps. By accurately defining these parameters, particles are distributed appropriately to give the MCL algorithm enough hypotheses to find the best estimate for the robot location.

References

[1] Thrun, Sebastian, and Dieter Fox. *Probabilistic Robotics*. 3rd ed. Cambridge, Mass: MIT Press, 2006. p.136.

See Also

`robotics.LikelihoodFieldSensorModel` | `robotics.MonteCarloLocalization` | `robotics.OdometryMotionModel`

Related Examples

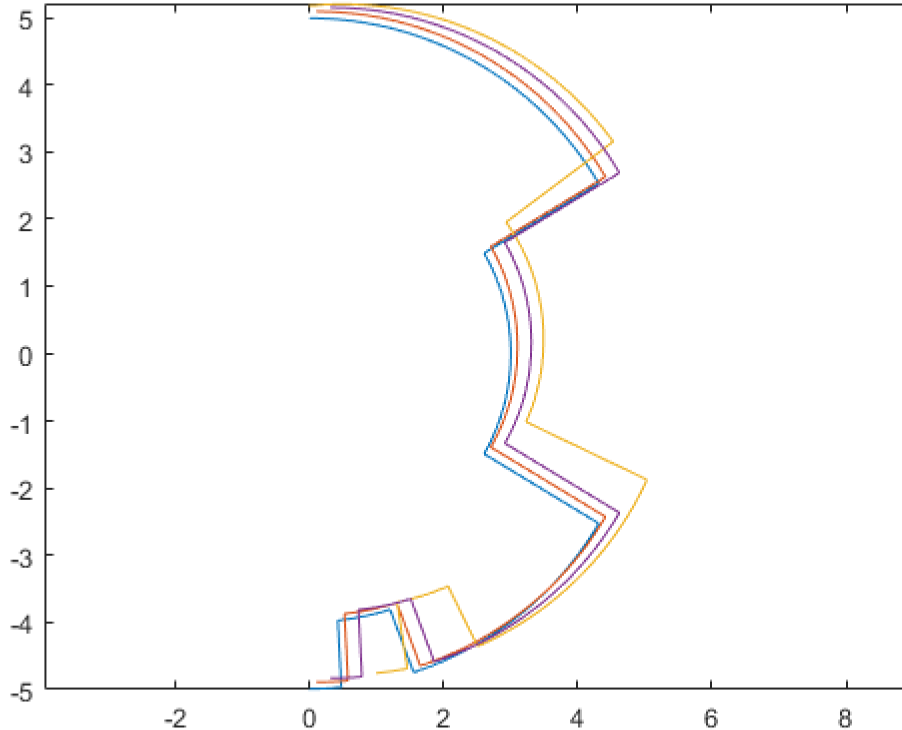
- "Localize TurtleBot Using Monte Carlo Localization"

Compose a Series of Laser Scans with Pose Changes

Use the `matchScans` function to compute the pose difference between a series of laser scans. Compose the relative poses by using a defined `composePoses` function to get a transformation to the initial frame. Then, transform all laser scans into the initial frame using these composed poses.

Specify the original laser scan and offsets to generate a series of shifted laser scans. Iterate through the scans and transform the original scan based on each offset. Plot the laser scans to see the shifted data.

```
ranges = zeros(300,4);
angles = zeros(300,4);
ranges(:,1) = 5*ones(300,1);
ranges(11:30,1) = 4*ones(1,20);
ranges(101:200,1) = 3*ones(1,100);
angles(:,1) = linspace(-pi/2,pi/2,300);
offset(1,:) = [0.1 0.1 0];
offset(2,:) = [0.4 0.1 0.1];
offset(3,:) = [-0.2 0 -0.1];

for i = 2:4
    [ranges(:,i),angles(:,i)] = transformScan(ranges(:,i-1),angles(:,i-1),offset(i-1,:)
end

[x,y] = pol2cart(angles,ranges);
plot(x,y)
axis equal
```



Perform scan matching on each laser scan set to get the relative pose between each scan. The outputs from the `matchScans` function are close to the specified offsets. The initial scan is in the initial frame, so the pose difference is `[0 0 0]`.

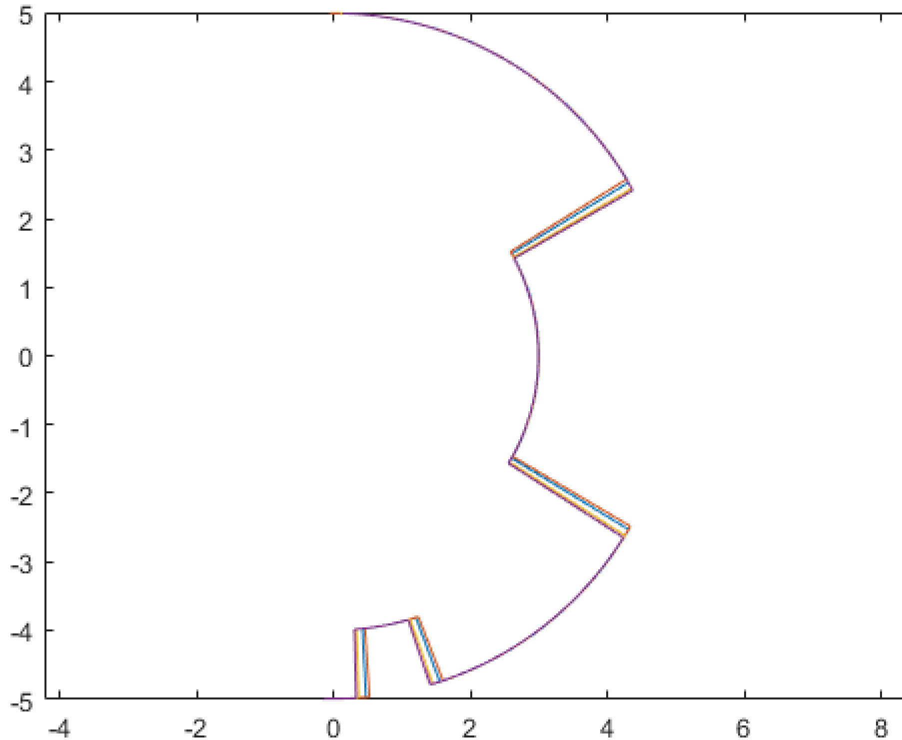
```
relPoses(1,:) = [0 0 0];

for i = 2:4
    relPoses(i,:) = matchScans(ranges(:,i),angles(:,i),ranges(:,i-1),angles(:,i-1),...
        'SolverAlgorithm','fminunc','CellSize',1);
end
```

Use the `composePoses` function in a loop to get the absolute transformation for each laser scan. This function is defined at the end of the example. Transform each scan to get them all in the initial frame.

```
transRanges = zeros(300,4);
transAngles = zeros(300,4);
transRanges(:,1) = ranges(:,1);
transAngles(:,1) = angles(:,1);
composedPoses(1,:) = [0 0 0];
for i = 2:4
    composedPoses(i,:) = composePoses(relPoses(i,:),composedPoses(i-1,:));
    [transRanges(:,i),transAngles(:,i)] = transformScan(ranges(:,i),angles(:,i),compose
end
```

Plot the transformed ranges and angles. They overlap well, based on the calculated transformations from `matchScans`.

```
[x,y] = pol2cart(transAngles,transRanges);
plot(x,y)
axis equal
```



Define the `composePoses` function. This function takes in the transformation of the initial frame to the base frame and the relative transformation from the initial frame to a second frame. For a series of laser scans, the `relative` input is the relative pose between the last two frames, and the `base` input is the composed pose over all previous scans.

You can also define this function in a separate script and save to the current folder.

```
function composedPose = composePoses(relative,base)
    %Convert both poses (3-by-1 vector) to transformations (4-by-4 matrix) and multiply
    %together using pose2tform function.
    tform = pose2tform(base)*pose2tform(relative);

    % Extract translational vector and Euler angles as ZYX.
```

```
    trvec = tform2trvec(tform);
    eul = tform2eul(tform);

    % Concatenate the elements of the transform as [x y theta].
    composedPose = [trvec(1:2) eul(1)];

    % Function to convert pose to transform.
    function tform = pose2tform(pose)
        x = pose(1);
        y = pose(2);
        th = wrapTo2Pi(pose(3));
        tform = trvec2tform([x y 0])*eul2tform([th 0 0]);
    end
end
```

See Also

matchScans | transformScan

Rigid Body Tree Robot Model

In this section...
"Rigid Body Tree Components" on page 7-68
"Robot Configurations" on page 7-71

The rigid body tree model is a representation of a robot structure. You can use it to represent robots such as manipulators or other kinematic trees. Use the `RigidBodyTree` class to create these models.

A rigid body tree is made up of rigid bodies (`RigidBody`) that are attached via joints (`Joint`). Each rigid body has a joint that defines how that body moves relative to its parent in the tree. Specify the transformation from one body to the next by setting the fixed transformation on each joint (`setFixedTransform`).

You can add, replace, or remove bodies from the rigid body tree model. You can also replace joints for specific bodies. The `RigidBodyTree` object maintains the relationships and updates the `RigidBody` object properties to reflect this relationship. You can also get transformations between different body frames using `getTransform`.

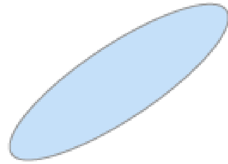
Rigid Body Tree Components

Base

Every rigid body tree has a base. The base defines the world coordinate frame and is the first attachment point for a rigid body. The base cannot be modified, except for the `Name` property. You can do so by modifying the `BaseName` property of the rigid body tree.

Rigid Body

The rigid body is the basic building block of rigid body tree model and is created using `RigidBody`. A rigid body, sometimes called a link, represents a solid body that cannot deform. The distance between any two points on a single rigid body remains constant.



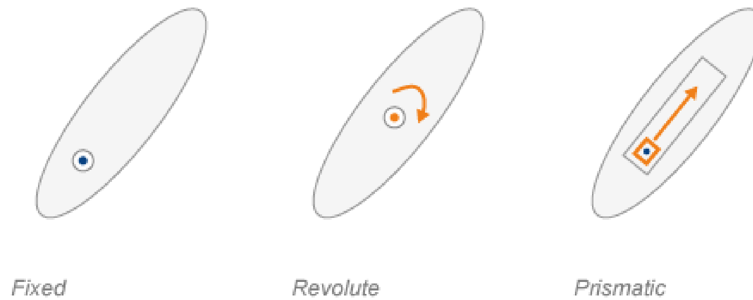
When added to a rigid body tree with multiple bodies, rigid bodies have parent or children bodies associated with them (`Parent` or `Children` properties). The parent is the body that this rigid body is attached to, which can be the robot base. The children are all the bodies attached to this body downstream from the base of the rigid body tree.

Each rigid body has a coordinate frame associated with them, and contains a `Joint` object.

Joint

Each rigid body has one joint, which defines the motion of that rigid body relative to its parent. It is the attachment point that connects two rigid bodies in a robot model. To represent a single physical body with multiple joints or different axes of motion, use multiple `RigidBody` objects.

The `Joint` class supports fixed, revolute, and prismatic joints.

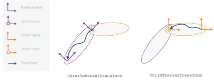


These joints allow the following motion, depending on their type:

- `'fixed'` — No motion. Body is rigidly connected to its parent.
- `'revolute'` — Rotational motion only. Body rotates around this joint relative to its parent. Position limits define the minimum and maximum angular position in radians around the axis of motion.
- `'prismatic'` — Translational motion only. The body moves linearly relative to its parent along the axis of motion.

Each joint has an axis of motion defined by the `JointAxis` property. The joint axis is a 3-D unit vector that either defines the axis of rotation (revolute joints) or axis of translation (prismatic joints). The `HomePosition` property defines the home position for that specific joint, which is a point within the position limits. Use `homeConfiguration` to return the home configuration for the robot, which is a collection of all the joints home positions in the model.

Joints also have properties that define the fixed transformation between parent and children body coordinate frames. These properties can only be set using the`setFixedTransform` method. Depending on your method of inputting transformation parameters, either the `JointToParentTransform` or `ChildToJointTransform` property is set using this method. The other property is set to the identity matrix. The following images depict what each property signifies.



- The `JointToParentTransform` defines where the joint of the child body is in relationship to the parent body frame. When `JointToParentTransform` is an identity matrix, the parent body and joint frames coincide.
- The `ChildToJointTransform` defines where the joint of the child body is in relationship to the child body frame. When `ChildToJointTransform` is an identity matrix, the child body and joint frames coincide.

Note The actual joint positions are not part of this `Joint` object. The robot model is stateless. There is an intermediate transformation between the parent and child joint frames that defines the position of the joint along the axis of motion. This transformation is defined in the robot configuration. See “Robot Configurations” on page 7-71.

Robot Configurations

After fully assembling your robot and defining transformations between different bodies, you can create robot configurations. A configuration defines all the joint positions of the robot by their joint names.

Use `homeConfiguration` to get the `HomePosition` property of each joint and create the home configuration.

Robot configurations are given as an array of structures.

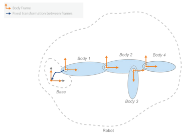
```
config = homeConfiguration(robot)
```

```
config = 

  1×6 struct array with fields:

    JointName
    JointPosition
```

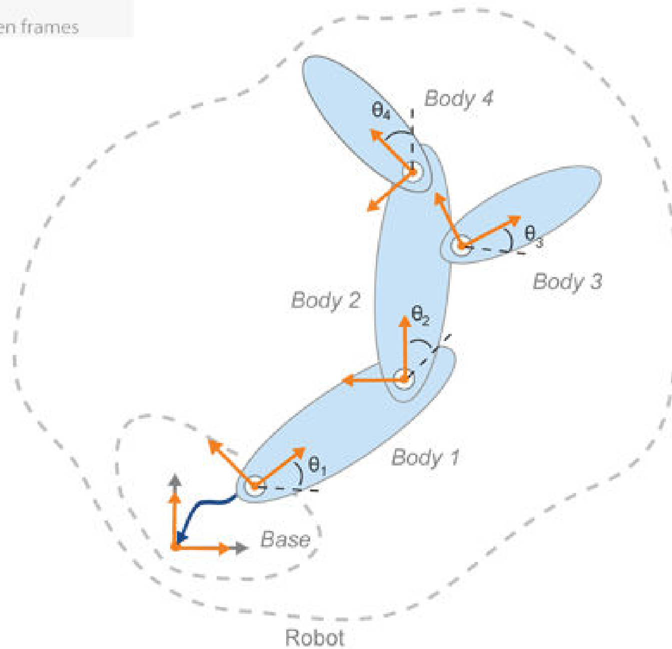
Each element in the array is a structure that contains the name and position of one of the robot joints.

```
config(1)
```

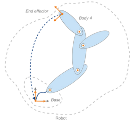
```
ans = 

  struct with fields:

         JointName: 'jnt1'
     JointPosition: 0
```



You can also generate a random configuration that obeys all the joint limits using `randomConfiguration`.



Use robot configurations when you want to plot a robot in a figure using `show`. Also, you can get the transformation between two body frames with a specific configuration using `getTransform`.



To get the robot configuration with a specified end-effector pose, use `InverseKinematics`. This algorithm solves for the required joint angles to achieve a specific pose for a specified rigid body.

See Also

`robotics.InverseKinematics` | `robotics.RigidBodyTree`

Related Examples

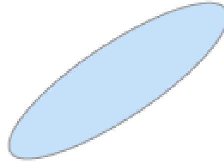
- “Build a Robot Step by Step” on page 7-74
- “Inverse Kinematics Algorithms” on page 7-79

Build a Robot Step by Step

This example goes through the process of building a robot step by step, showing you the different robot components and how functions are called to build it. Code sections are shown, but actual values for dimensions and transformations depend on your robot.

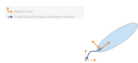
1. Create a rigid body object.

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



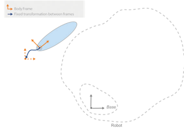
2. Create a joint and assign it to the rigid body. Define the home position property of the joint, `HomePosition`. Set the joint-to-parent transform using a homogeneous transformation, `tform`. Use the `trvec2tform` function to convert from a translation vector to a homogenous transformation.`ChildToJointTransform` is set to an identity matrix. For information on homogenous transformations and other coordinate transformations, see “Coordinate System Transformations”.

```
jnt1 = robotics.Joint('jnt1','revolute');
jnt1.HomePosition = pi/4;
tform = trvec2tform([0.25, 0.25, 0]); % User defined
setFixedTransform(jnt1,tform);
body1.Joint = jnt1;
```



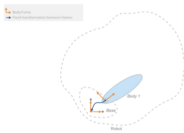
3. Create a rigid body tree. This tree is initialized with a base coordinate frame to attach bodies to.

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



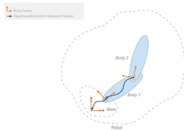
4 Add the first body to the tree. Specify that you are attaching it to the base of the tree. The fixed transform defined previously is from the base (parent) to the first body.

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



5 Create a second body. Define properties of this body and attach it to the first rigid body. Define the transformation relative to the previous body frame.

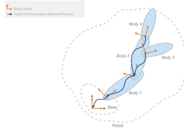
```
body2 = robotics.RigidBody('body2');
jnt2 = robotics.Joint('jnt2','revolute');
jnt2.HomePosition = pi/6; % User defined
tform2 = trvec2tform([1, 0, 0]); % User defined
setFixedTransform(jnt2,tform2);
body2.Joint = jnt2;
addBody(robot,body2,'body1'); % Add body2 to body1
```



6 Add other bodies. Attach body 3 and 4 to body 2.

```
body3 = robotics.RigidBody('body3');
body4 = robotics.RigidBody('body4');
jnt3 = robotics.Joint('jnt3','revolute');
jnt4 = robotics.Joint('jnt4','revolute');
tform3 = trvec2tform([0.6, -0.1, 0])*eul2tform([-pi/2, 0, 0]); % User defined
tform4 = trvec2tform([1, 0, 0]); % User defined
setFixedTransform(jnt3,tform3);
setFixedTransform(jnt4,tform4);
jnt3.HomePosition = pi/4; % User defined
body3.Joint = jnt3
body4.Joint = jnt4
```

```
addBody(robot,body3,'body2'); % Add body3 to body2
addBody(robot,body4,'body2'); % Add body4 to body2
```



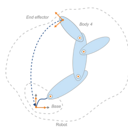
7. If you have a specific end effector that you care about for control, define it as a rigid body with a fixed joint. For this robot, add an end effector to `body4` so that you can get transformations for it.

```
bodyEndEffector = robotics.RigidBody('endeffector');
tform5 = trvec2tform([0.5, 0, 0]); % User defined
setFixedTransform(bodyEndEffector.Joint,tform5);
addBody(robot,bodyEndEffector,'body4');
```

8. Now that you have created your robot, you can generate robot configurations. With a given configuration, you can also get a transformation between two body frames using `robotics.RigidBodyTree.getTransform`. Get a transformation from the end effector to the base.

```
config = robot.randomConfiguration
tform = getTransform(robot,config,'endeffector','base')
```

```
config = 

  1×2 struct array with fields:

    JointName
    JointPosition


tform =

   -0.5484    0.8362         0         0
   -0.8362   -0.5484         0         0
         0         0    1.0000         0
         0         0         0    1.0000
```

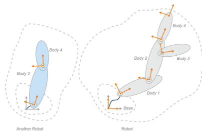


Note This transform is specific to the dimensions specified in this example. Values for your robot vary depending on the transformations you define.

---

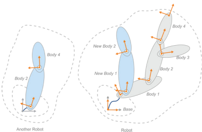
9 You can create a subtree from your existing robot or other robot models by using `robotics.RigidBodyTree.subtree`. Specify the body name to use as the base for the new subtree. You can modify this subtree by adding, changing, or removing bodies.

```
newArm = subtree(robot,'body2');
removeBody(newArm,'body3');
removeBody(newArm,'endeffector')
```



10 You can also add these subtrees to the robot. Adding a subtree is similar to adding a body. The specified body name acts as a base for attachment, and all transformations on the subtree are relative to that body frame. Before you add the subtree, you must ensure all the names of bodies and joints are unique. Create copies of the bodies and joints, rename them, and replace them on the subtree. Call `robotics.RigidBodyTree.addSubtree` to attach the subtree to a specified body.

```
newBody1 = copy(getBody(newArm,'body2'));
newBody2 = copy(getBody(newArm,'body4'));
newBody1.Name = 'newBody1';
newBody2.Name = 'newBody2';
newBody1.Joint = robotics.Joint('newJnt1','revolute');
newBody2.Joint = robotics.Joint('newJnt2','revolute');
tformTree = trvec2tform([0.2, 0, 0]); % User defined
setFixedTransform(newBody1,tformTree);
replaceBody(newArm,'body2',newBody1);
replaceBody(newArm,'body4',newBody2);

addSubtree(robot,'body1',newArm);
```



11 Finally, you can use `showdetails` to look at the robot you built. Verify that the joint types are correct.

```
showdetails(robot)

 Idx          Body Name               Joint Name              Joint Type          Pare
 ---          ---------               ----------              ----------          ----
   1              body1                     jnt1                revolute
   2              body2                     jnt2                revolute
   3              body3                     jnt3                revolute
   4              body4                     jnt4                revolute
   5        endeffector          endeffector_jnt                   fixed
   6           newBody1                  newJnt1                revolute
   7           newBody2                  newJnt2                revolute
--------------------
```

See Also

`robotics.InverseKinematics` | `robotics.RigidBodyTree`

Related Examples

- "Rigid Body Tree Robot Model" on page 7-68

Inverse Kinematics Algorithms

In this section...
“Choose an Algorithm” on page 7-79
“Solver Parameters” on page 7-80
“Solution Information” on page 7-81
“References” on page 7-82

The `robotics.InverseKinematics` and `robotics.GeneralizedInverseKinematics` classes give you access to inverse kinematics (IK) algorithms. You can use these algorithms to generate a robot configuration that achieves specified goals and constraints for the robot. This robot configuration is a list of joint positions that are within the position limits of the robot model and do not violate any constraints the robot has.

Choose an Algorithm

MATLAB supports two algorithms for achieving an IK solution: the BFGS projection algorithm and the Levenberg-Marquardt algorithm. Both algorithms are iterative, gradient-based optimization methods that start from an initial guess at the solution and seek to minimize a specific cost function. If either algorithm converges to a configuration where the cost is close to zero within a specified tolerance, it has found a solution to the inverse kinematics problem. However, for some combinations of initial guesses and desired end effector poses, the algorithm may exit without finding an ideal robot configuration. To handle this, the algorithm utilizes a random restart mechanism. If enabled, the random restart mechanism restarts the iterative search from a random robot configuration whenever that search fails to find a configuration that achieves the desired end effector pose. These random restarts continue until either a qualifying IK solution is found, the maximum time has elapsed, or the iteration limit is reached.

To set your algorithm, specify the `SolverAlgorithm` property as either `'BFGSGradientProjection'` or `'LevenbergMarquardt'`.

BFGS Gradient Projection

The Broyden-Fletcher-Goldfarb-Shanno (BFGS) gradient projection algorithm is a quasi-Newton method that uses the gradients of the cost function from past iterations to generate approximate second-derivative information. The algorithm uses this second-

derivative information in determining the step to take in the current iteration. A gradient projection method is used to deal with boundary limits on the cost function that the joint limits of the robot model create. The direction calculated is modified so that the search direction is always valid.

This method is the default algorithm and is more robust at finding solutions than the Levenberg-Marquardt method. It is more effective for configurations near joint limits or when the initial guess is not close to the solution. If your initial guess is close to the solution and a quicker solution is needed, consider the “Levenberg-Marquardt” on page 7-80 method.

Levenberg-Marquardt

The Levenberg-Marquardt (LM) algorithm variant used in the `InverseKinematics` class is an error-damped least-squares method. The error-damped factor helps to prevent the algorithm from escaping a local minimum. The LM algorithm is optimized to converge much faster if the initial guess is close to the solution. However the algorithm does not handle arbitrary initial guesses well. Consider using this algorithm for finding IK solutions for a series of poses along a desired trajectory of the end effector. Once a robot configuration is found for one pose, that configuration is often a good initial guess at an IK solution for the next pose in the trajectory. In this situation, the LM algorithm may yield faster results. Otherwise, use the “BFGS Gradient Projection” on page 7-79 instead.

Solver Parameters

Each algorithm has specific tunable parameters to improve solutions. These parameters are specified in the `SolverParameters` property of the object.

BFGS Gradient Projection

The solver parameters for the BFGS algorithm have the following fields:

- `MaxIterations` — Maximum number of iterations allowed. The default is 1500.
- `MaxTime` — Maximum number of seconds that the algorithm runs before timing out. The default is 10.
- `GradientTolerance` — Threshold on the gradient of the cost function. The algorithm stops if the magnitude of the gradient falls below this threshold. Must be a positive scalar.
- `SolutionTolerance` — Threshold on the magnitude of the error between the end-effector pose generated from the solution and the desired pose. The weights specified

for each component of the pose in the object are included in this calculation. Must be a positive scalar.

- EnforceJointLimits — Indicator if joint limits are considered in calculating the solution. JointLimits is a property of the robot model in robotics.RigidBodyTree. By default, joint limits are enforced.
- AllowRandomRestarts — Indicator if random restarts are allowed. Random restarts are triggered when the algorithm approaches a solution that does not satisfy the constraints. A randomly generated initial guess is used. MaxIteration and MaxTime are still obeyed. By default, random restarts are enabled.
- StepTolerance — Minimum step size allowed by the solver. Smaller step sizes usually mean that the solution is close to convergence. The default is 10^{-14}.

Levenberg-Marquardt

The solver parameters for the LM algorithm have the following extra fields in addition to what the “BFGS Gradient Projection” on page 7-80 method requires:

- ErrorChangeTolerance — Threshold on the change in end-effector pose error between iterations. The algorithm returns if the changes in all elements of the pose error are smaller than this threshold. Must be a positive scalar.
- DampingBias — A constant term for damping. The LM algorithm has a damping feature controlled by this constant that works with the cost function to control the rate of convergence. To disable damping, use the UseErrorDamping parameter.
- UseErrorDamping — 1 (default), Indicator of whether damping is used. Set this parameter to false to disable dampening.

Solution Information

While using the inverse kinematics algorithms, each call on the object returns solution information about how the algorithm performed. The solution information is provided as a structure with the following 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” on page 7-82.
- `Status` — Character vector describing whether the solution is within the tolerance (`'success'`) or the best possible solution the algorithm could find (`'best available'`).

Exit Flags

In the solution information, the exit flags give more details on the execution of the specific algorithm. Look at the `Status` property of the object to find out if the algorithm was successful. Each exit flag code has a defined description.

`'BFGSGradientProjection'` algorithm exit flags:

- `1` — Local minimum found.
- `2` — Maximum number of iterations reached.
- `3` — Algorithm timed out during operation.
- `4` — Minimum step size. The step size is below the `StepToleranceSize` field of the `SolverParameters` property.
- `5` — No exit flag. Relevant to `'LevenbergMarquardt'` algorithm only.
- `6` — Search direction invalid.
- `7` — Hessian is not positive semidefinite.

`'LevenbergMarquardt'` algorithm exit flags:

- `1` — Local minimum found.
- `2` — Maximum number of iterations reached.
- `3` — Algorithm timed out during operation.
- `4` — Minimum step size. The step size is below the `StepToleranceSize` field of the `SolverParameters` property.
- `5` — The change in end-effector pose error is below the `ErrorChangeTolerance` field of the `SolverParameters` property.

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.GeneralizedInverseKinematics` | `robotics.InverseKinematics` | `robotics.RigidBodyTree`

Related Examples

- "2-D Path Tracing With Inverse Kinematics" on page 8-2
- "Control PR2 Arm Movements Using ROS Actions and Inverse Kinematics"
- "Rigid Body Tree Robot Model" on page 7-68

Manipulator Algorithms

- "2-D Path Tracing With Inverse Kinematics" on page 8-2
- "Trace An End-Effector Trajectory with Inverse Kinematics in Simulink®" on page 8-7
- "Solve Inverse Kinematics for a Four-Bar Linkage" on page 8-14
- "Robot Dynamics" on page 8-19
- "Calculate Manipulator Gravity Dynamics in Simulink" on page 8-21
- "Compute Velocity Product for Manipulators in Simulink" on page 8-23
- "Compute Geometric Jacobian for Manipulators in Simulink" on page 8-26
- "Get Transformations for Manipulator Bodies in Simulink" on page 8-28
- "Get Mass Matrix for Manipulators in Simulink" on page 8-31

ex45526735

ex45526735

2-D Path Tracing With Inverse Kinematics

Introduction

This example shows how to calculate inverse kinematics for a simple 2D manipulator using the `robotics.InverseKinematics` class. The manipulator robot is a simple 2-degree-of-freedom planar manipulator with revolute joints which is created by assembling rigid bodies into a `robotics.RigidBodyTree` object. A circular trajectory is created in a 2-D plane and given as points to the inverse kinematics solver. The solver calculates the required joint positions to achieve this trajectory. Finally, the robot is animated to show the robot configurations that achieve the circular trajectory.

Construct The Robot

Create a `RigidBodyTree` object and rigid bodies with their associated joints. Specify the geometric properties of each rigid body and add it to the robot.

Start with a blank rigid body tree model.

```
robot = robotics.RigidBodyTree('DataFormat','column','MaxNumBodies',3);
```

Specify arm lengths for the robot arm.

```
L1 = 0.3;
L2 = 0.3;
```

Add `'link1'` body with `'joint1'` joint.

```
body = robotics.RigidBody('link1');
joint = robotics.Joint('joint1', 'revolute');
setFixedTransform(joint,trvec2tform([0 0 0]));
joint.JointAxis = [0 0 1];
body.Joint = joint;
addBody(robot, body, 'base');
```

Add `'link2'` body with `'joint2'` joint.

```
body = robotics.RigidBody('link2');
joint = robotics.Joint('joint2','revolute');
setFixedTransform(joint, trvec2tform([L1,0,0]));
joint.JointAxis = [0 0 1];
body.Joint = joint;
addBody(robot, body, 'link1');
```

Add `'tool'` end effector with `'fix1'` fixed joint.

```
body = robotics.RigidBody('tool');
joint = robotics.Joint('fix1','fixed');
setFixedTransform(joint, trvec2tform([L2, 0, 0]));
body.Joint = joint;
addBody(robot, body, 'link2');
```

Show details of the robot to validate the input properties. The robot should have two non-fixed joints for the rigid bodies and a fixed body for the end-effector.

```
showdetails(robot)
```

```
--------------------
Robot: (3 bodies)

 Idx    Body Name    Joint Name    Joint Type    Parent Name(Idx)   Children Name(s)
 ---    ---------    ----------    ----------    ----------------   ----------------
   1        link1        joint1      revolute             base(0)   link2(2)
   2        link2        joint2      revolute            link1(1)   tool(3)
   3         tool          fix1         fixed            link2(2)
--------------------
```

Define The Trajectory

Define a circle to be traced over the course of 10 seconds. This circle is in the *xy* plane with a radius of 0.15.

```
t = (0:0.2:10)'; % Time
count = length(t);
center = [0.3 0.1 0];
radius = 0.15;
theta = t*(2*pi/t(end));
points = center + radius*[cos(theta) sin(theta) zeros(size(theta))];
```

Inverse Kinematics Solution

Use an `InverseKinematics` object to find a solution of robotic configurations that achieve the given end-effector positions along the trajectory.

Pre-allocate configuration solutions as a matrix `qs`.

```
q0 = homeConfiguration(robot);
ndof = length(q0);
qs = zeros(count, ndof);
```

Create the inverse kinematics solver. Because the *xy* Cartesian points are the only important factors of the end-effector pose for this workflow, specify a non-zero weight for the fourth and fifth elements of the `weight` vector. All other elements are set to zero.

```
ik = robotics.InverseKinematics('RigidBodyTree', robot);
weights = [0, 0, 0, 1, 1, 0];
endEffector = 'tool';
```

Loop through the trajectory of points to trace the circle. Call the `ik` object for each point to generate the joint configuration that achieves the end-effector position. Store the configurations to use later.

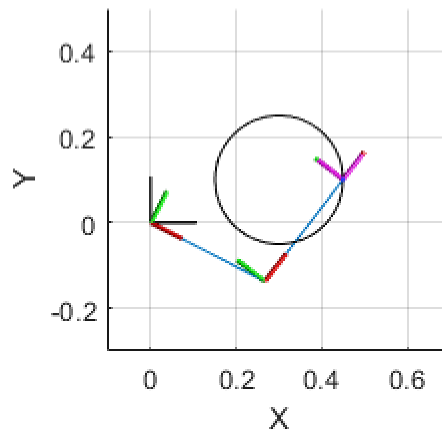
```
qInitial = q0; % Use home configuration as the initial guess
for i = 1:count
    % Solve for the configuration satisfying the desired end effector
    % position
    point = points(i,:);
    qSol = ik(endEffector,trvec2tform(point),weights,qInitial);
    % Store the configuration
    qs(i,:) = qSol;
    % Start from prior solution
    qInitial = qSol;
end
```

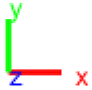
Animate The Solution

Plot the robot for each frame of the solution using that specific robot configuration. Also, plot the desired trajectory.

Show the robot in the first configuration of the trajectory. Adjust the plot to show the 2-D plane that circle is drawn on. Plot the desired trajectory.

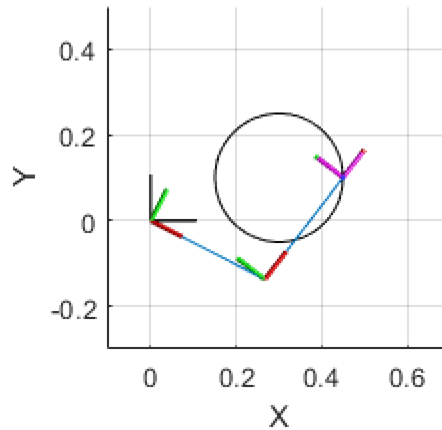
```
figure
show(robot,qs(1,:)');
view(2)
ax = gca;
ax.Projection = 'orthographic';
hold on
plot(points(:,1),points(:,2),'k')
axis([-0.1 0.7 -0.3 0.5])
```

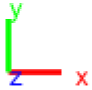




Set up a `robotics.Rate` object to display the robot trajectory at a fixed rate of 15 frames per second. Show the robot in each configuration from the inverse kinematic solver. Watch as the arm traces the circular trajectory shown.

```
framesPerSecond = 15;
r = robotics.Rate(framesPerSecond);
for i = 1:count
    show(robot,qs(i,:)','PreservePlot',false);
    drawnow
    waitfor(r);
end
```





See Also

`InverseKinematics` | `Joint` | `RigidBody` | `RigidBodyTree`

Related Examples

- "Control PR2 Arm Movements Using ROS Actions and Inverse Kinematics"
- "Inverse Kinematics Algorithms" on page 7-79

Trace An End-Effector Trajectory with Inverse Kinematics in Simulink®

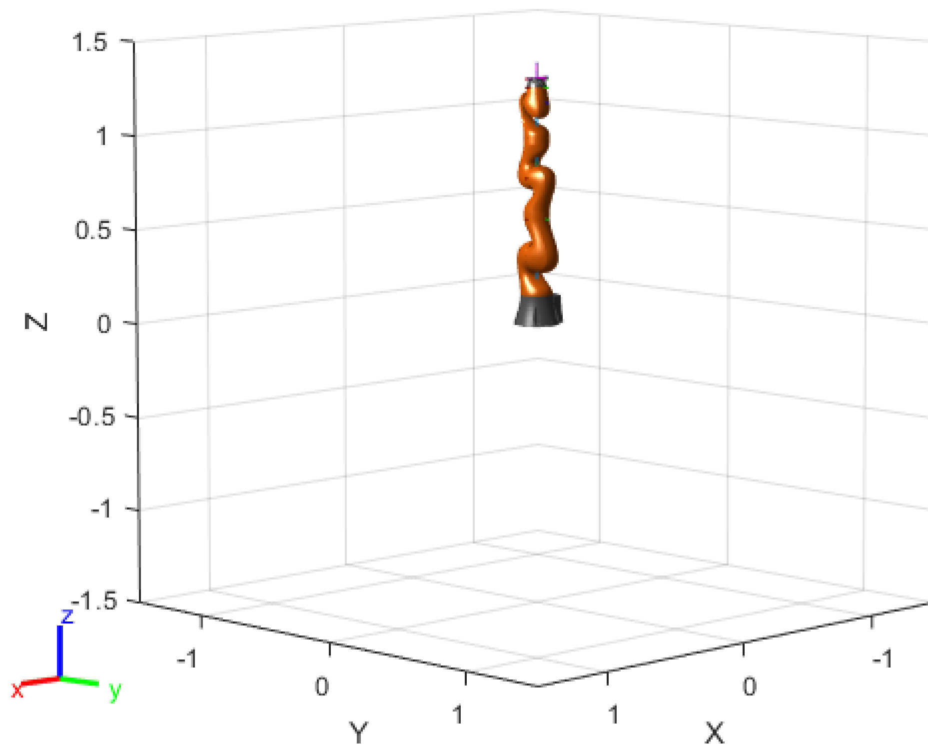
Use a rigid body robot model to compute inverse kinematics using Simulink®. Define a trajectory for the robot end effector and loop through the points to solve robot configurations that trace this trajectory.

Import a robot model from a URDF (unified robot description format) file as a RigidBodyTree object.

```
robot = importrobot('iiwa14.urdf');
robot.DataFormat = 'column';
```

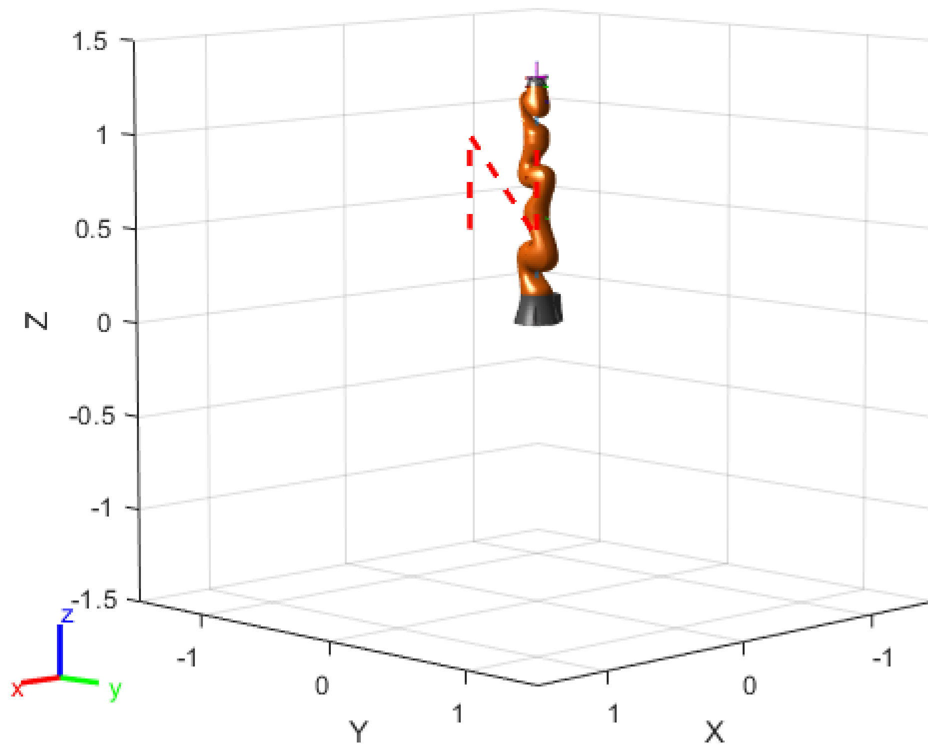
View the robot.

```
ax = show(robot);
```



Specify a robot trajectory. These *xyz*-coordinates draw an N-shape in front of the robot.

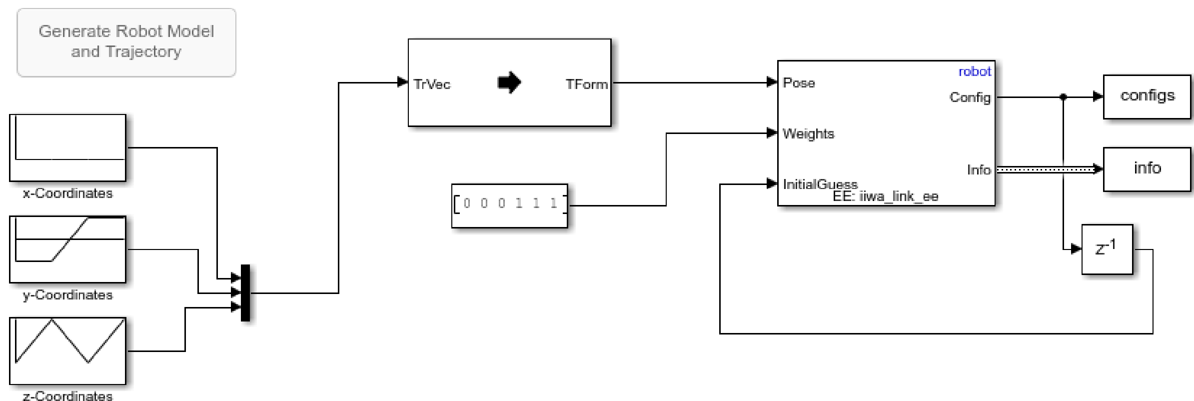
```
x = 0.5*zeros(1,4)+0.25;
y = 0.25*[-1 -1 1 1];
z = 0.25*[-1 1 -1 1] + 0.75;

hold on
plot3(x,y,z,'--r','LineWidth',2,'Parent',ax)
hold off
```



Open a model that performs inverse kinematics. The *xyz*-coordinates defined in MATLAB® are converted to homogeneous transformations and input as the desired `Pose`. The output inverse-kinematic solution is fed back as the initial guess for the next solution. This initial guess helps track the end-effector pose and generate smooth configurations.

You can press the callback button to regenerate the robot model and trajectory you just defined.

```
close
open_system('sm_ik_trajectory_model.slx')
```

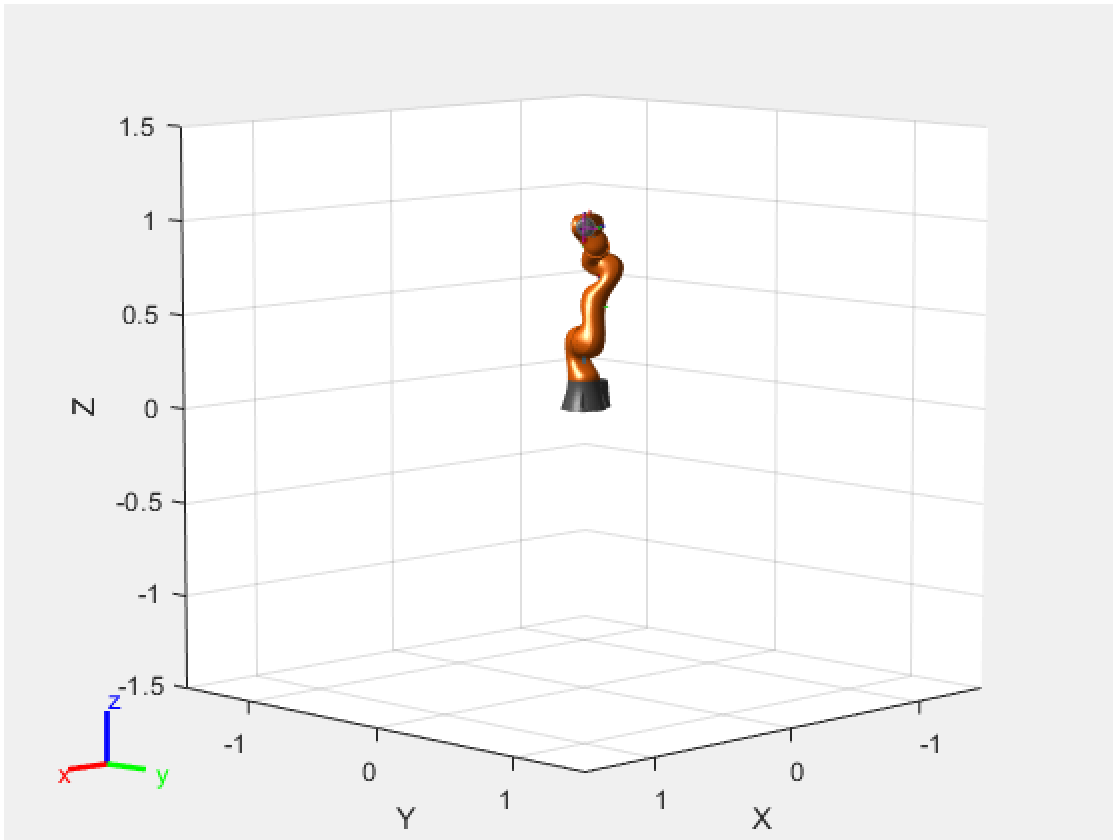


Run the simulation. The model should generate the robot configurations (`configs`) that follow the specified trajectory for the end effector.

```
sim('sm_ik_trajectory_model.slx')
```

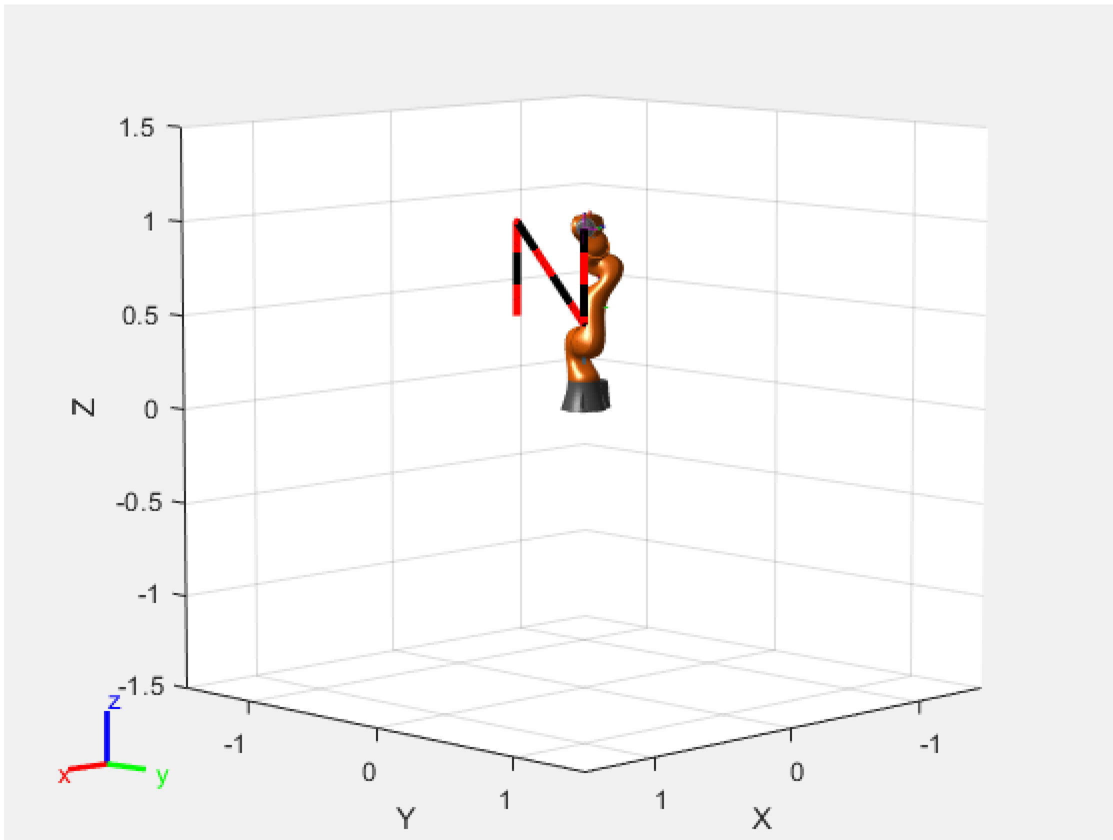
Loop through the robot configurations and display the robot for each time step. Store the end-effector positions in `xyz`.

```
figure('Visible','on');
tformIndex = 1;
for i = 1:10:numel(configs.Data)/7
    currConfig = configs.Data(:,1,i);
    show(robot,currConfig);
    drawnow

    xyz(tformIndex,:) = tform2trvec(getTransform(robot,currConfig,'iiwa_link_ee'));
    tformIndex = tformIndex + 1;
end
```



Draw the final trajectory of the end effector as a black line. The figure shows the end effector tracing the N-shape originally defined (red dotted line).

```
figure('Visible','on')
show(robot,configs.Data(:,1,end));

hold on
plot3(xyz(:,1),xyz(:,2),xyz(:,3),'-k','LineWidth',3);
plot3(x,y,z,'--r','LineWidth',3)
hold off
```



See Also

Objects
`robotics.GeneralizedInverseKinematics` | `robotics.InverseKinematics` | `robotics.RigidBodyTree`

Blocks
Get Transform | Inverse Dynamics | Inverse Kinematics

Related Examples

- “Control PR2 Arm Movements Using ROS Actions and Inverse Kinematics”
- “Inverse Kinematics Algorithms” on page 7-79

Solve Inverse Kinematics for a Four-Bar Linkage

This example shows how to solve inverse kinematics for a four-bar linkage, a simple planar closed-chain linkage. Robotics System Toolbox™ doesn't directly support closed-loop mechanisms. However, the loop-closing joints can be approximated using kinematic constraints. This example shows how to setup a rigid body tree for a four-bar linkage, specify the kinematic constraints, and solve for a desired end-effector position.

Initialize the four-bar linkage rigid body tree model.

```
robot = robotics.RigidBodyTree('Dataformat','column','MaxNumBodies',7);
```

Define body names, parent names, joint names, joint types, and fixed transforms in cell arrays. The fixed transforms define the geometry of the four-bar linkage. The linkage rotates in the *xz*-plane. An offset of `-0.1` is used in the *y*-axis on the `'b4'` body to isolate the motion of the overlapping joints for `'b3'` and `'b4'`.

```
bodyNames = {'b1','b2','b3','b4','b5','b6'};
parentNames = {'base','b1','b2','base','b4','b5'};
jointNames = {'j1','j2','j3','j4','j5','j6'};
jointTypes = {'revolute','revolute','fixed','revolute','revolute','fixed'};
fixedTforms = {eye(4), ...
                trvec2tform([0 0 0.5]), ...
                trvec2tform([0.8 0 0]), ...
                trvec2tform([0.0 -0.1 0]), ...
                trvec2tform([0.8 0 0]), ...
                trvec2tform([0 0 0.5])};
```

Use a `for` loop to assemble the four-bar linkage:

- Create a rigid body and specify the joint type.
- Specify the `JointAxis` property for any non-fixed joints.
- Specify the fixed transformation.
- Add the body to the rigid body tree.

```
for k = 1:6

    b = robotics.RigidBody(bodyNames{k});
    b.Joint = robotics.Joint(jointNames{k},jointTypes{k});

    if ~strcmp(jointTypes{k},'fixed')
        b.Joint.JointAxis = [0 1 0];
```

```
    end

    b.Joint.setFixedTransform(fixedTforms{k});

    robot.addBody(b,parentNames{k});
end
```

Add a final body to function as the end-effector (handle) for the four-bar linkage.

```
bn = 'handle';
b = robotics.RigidBody(bn);
b.Joint.setFixedTransform(trvec2tform([0 -0.15 0]));
robot.addBody(b,'b6');
```

Specify kinematic constraints for the `GeneralizedInverseKinematics` object:

- **Position constraint 1** : The origins of `'b3'` body frame and `'b6'` body frame should always overlap. This keeps the handle in line with the approximated closed-loop mechanism. Use the `-0.1` offset for the *y*-coordinate.
- **Position constraint 2** : End-effector should target the desired position.
- **Joint limit bounds** : Satisfy the joint limits in the rigid body tree model.

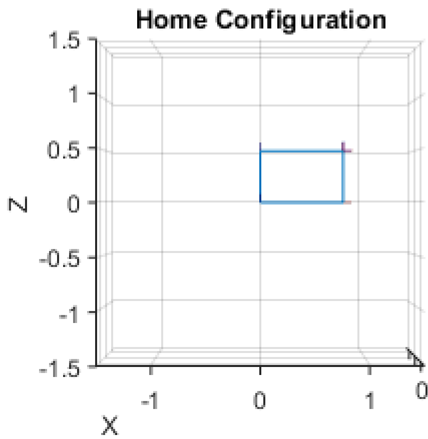
```
gik = robotics.GeneralizedInverseKinematics('RigidBodyTree',robot);
gik.ConstraintInputs = {'position',...  % Position constraint for closed-loop mechanism
                        'position',...  % Position constraint for end-effector
                        'joint'};       % Joint limits
gik.SolverParameters.AllowRandomRestart = false;

% Position constraint 1
positionTarget1 = robotics.PositionTarget('b6','ReferenceBody','b3');
positionTarget1.TargetPosition = [0 -0.1 0];
positionTarget1.Weights = 50;
positionTarget1.PositionTolerance = 1e-6;

% Joint limit bounds
jointLimBounds = robotics.JointPositionBounds(gik.RigidBodyTree);
jointLimBounds.Weights = ones(1,size(gik.RigidBodyTree.homeConfiguration,1))*10;

% Position constraint 2
desiredEEPosition = [0.9 -0.1 0.9]'; % Position is relative to base.
positionTarget2 = robotics.PositionTarget('handle');
positionTarget2.TargetPosition = desiredEEPosition;
positionTarget2.PositionTolerance = 1e-6;
positionTarget2.Weights = 1;
```

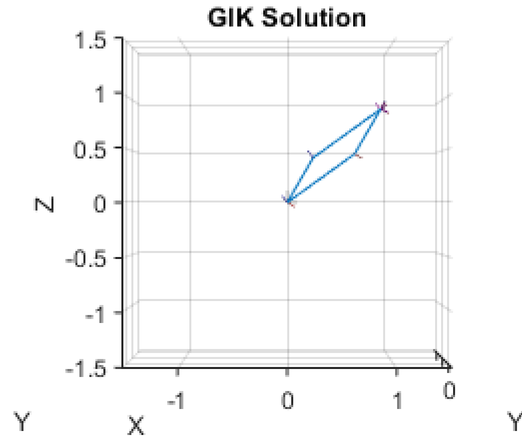
Compute the kinematic solution using the `gik` object. Specify the initial guess and the different kinematic constraints in the proper order.

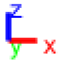
```
iniGuess = homeConfiguration(robot);
[q, solutionInfo] = gik(iniGuess,positionTarget1,positionTarget2,jointLimBounds);
```

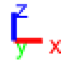
Examine the results in `solutionInfo`. Show the kinematic solution compared to the home configuration. Plots are shown in the *xz*-plane.

```
loopClosingViolation = solutionInfo.ConstraintViolations(1).Violation;
jointBndViolation = solutionInfo.ConstraintViolations(2).Violation;
eePositionViolation = solutionInfo.ConstraintViolations(3).Violation;

subplot(1,2,1)
show(robot,homeConfiguration(robot));
title('Home Configuration')
view([0 -1 0]);
subplot(1,2,2)
show(robot,q);
title('GIK Solution')
view([0 -1 0]);
```









See Also

Classes

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

Related Examples

- "Rigid Body Tree Robot Model" on page 7-68

- “Plan a Reaching Trajectory With Multiple Kinematic Constraints”
- “Control LBR Manipulator Motion Through Joint Torque Commands”

Robot Dynamics

In this section...
“Dynamics Properties” on page 8-19
“Dynamics Functions” on page 8-20

Robot dynamics is the relationship between the forces acting on a robot and the resulting motion of the robot. In Robotics System Toolbox, manipulator dynamics information is contained within a `RigidBodyTree` object. This object describes a rigid body tree model that has multiple `RigidBody` objects connected through `Joint` objects. The `Joint`, `RigidBody`, and `RigidBodyTree` objects all contain information related to the robot kinematics and dynamics.

Note To use dynamics functions, you must set the `DataFormat` property to `'row'` or `'column'`. This setting takes inputs and gives outputs as row or column vectors for relevant robotics calculations, such as robot configurations or joint torques.

Dynamics Properties

When working with robot dynamics, specify the information for individual bodies of your manipulator robot using properties on the `RigidBody` objects:

- `Mass` — Mass of the rigid body in kilograms.
- `CenterOfMass` — Center of mass position of the rigid body, specified as an `[x y z]` vector. The vector describes the location of the center of mass relative to the body frame in meters.
- `Inertia` — Inertia of rigid body, specified as an `[Ixx Iyy Izz Iyz Ixz Ixy]` vector relative to the body frame in kilogram square meters. The first three elements of the vector are the diagonal elements of the inertia tensor (moment of inertia). The last three elements are the off-diagonal elements of the inertia tensor (product of inertia). The inertia tensor is a positive definite matrix:

$$\begin{pmatrix} I_{xx} & I_{xy} & I_{xz} \\ I_{xy} & I_{yy} & I_{yz} \\ I_{xz} & I_{yz} & I_{zz} \end{pmatrix}$$

For information related to your whole manipulator robot model, specify these `RigidBodyTree` object properties:

- `Gravity` — Gravitational acceleration experienced by the robot, specified as an `[x y z]` vector in meters per second squared. By default, there is no gravitational acceleration.
- `DataFormat` — The input and output data format for the kinematics and dynamics functions. Set this property to `'row'` or `'column'` to use dynamics functions. This setting takes inputs and gives outputs as row or column vectors for relevant robotics calculations, such as robot configurations or joint torques.

Dynamics Functions

The following dynamics functions are available for robot manipulators. You can use these functions after specifying all the relevant dynamics properties on your `RigidBodyTree` robot model.

- `forwardDynamics` — Compute joint accelerations given joint torques and states
- `inverseDynamics` — Compute required joint torques given desired motion
- `externalForce` — Compose external force matrix relative to base
- `gravityTorque` — Compute joint torques that compensate gravity
- `centerOfMass` — Compute center of mass position and Jacobian
- `massMatrix` — Compute joint-space mass matrix
- `velocityProduct` — Compute joint torques that cancel velocity-induced forces

See Also

`GeneralizedInverseKinematics` | `InverseKinematics` | `RigidBodyTree`

Related Examples

- “Control LBR Manipulator Motion Through Joint Torque Commands”
- “Control PR2 Arm Movements Using ROS Actions and Inverse Kinematics”

Calculate Manipulator Gravity Dynamics in Simulink

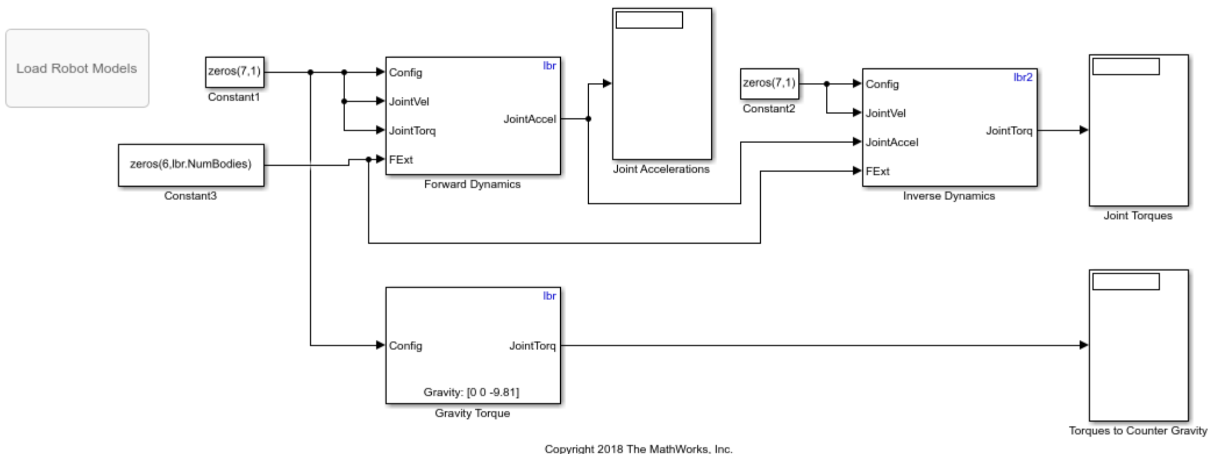
This example shows how to use the manipulator algorithm blocks to compute and compare dynamics due to gravity for a manipulator robot.

Specify two similar robot models with different gravity accelerations. Load the KUKA LBR robot model into the MATLAB® workspace and create a copy of it. For the first robot model, `lbr`, specify a normal gravity vector, `[0 0 -9.81]`. For the copy, lbr2, use the default gravity vector, `[0 0 0]`. These robot models are also specified in the **Rigid body tree** parameters of the blocks in the model.

```
load('exampleLBR.mat','lbr')
lbr.DataFormat = 'column';
lbr2 = copy(lbr);
lbr.Gravity = [0 0 -9.81];
```

Open the gravity dynamics model. If needed, reload the robot models specified by the MATLAB code using the **Load Robot Models** callback button.

```
open_system('gravity_dynamics_model.slx')
```



The Forward Dynamics block calculates the joint accelerations due to gravity for a given `lbr` robot configuration with no initial velocity, torque, or external force. The Inverse Dynamics block then computes the torques needed for the joint to create those same accelerations with no gravity by using the `lbr2` robot. Finally, the Gravity Torque block calculates the torque required to counteract gravity for the `lbr` robot.

Run the model. Besides some small numerical differences, the gravity torque and the torque required for accelerations due to gravity are the same value with opposite directions.

See Also

Blocks
Forward Dynamics | Get Jacobian | Gravity Torque | Inverse Dynamics | Joint Space Mass Matrix | Velocity Product Torque

Classes
`RigidBodyTree`

Functions
`externalForce` | `homeConfiguration` | `importrobot` | `randomConfiguration`

Related Examples

- "Perform Safe Trajectory Tracking Control Using Robotics Manipulator Blocks"

Compute Velocity Product for Manipulators in Simulink

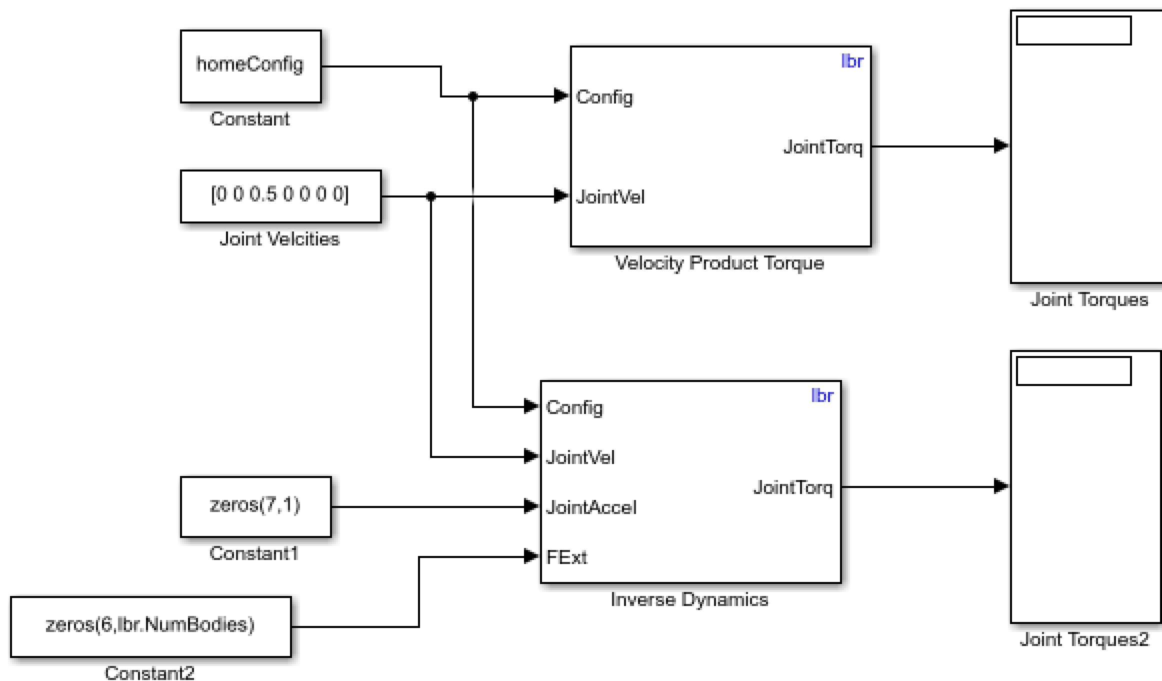
This example shows how to calculate the velocity-induced torques for a robot manipulator by using a `robotics.RigidBodyTree` model. In this example, you define a robot model and robot configuration in MATLAB® and pass them to Simulink® to be used with the manipulator algorithm blocks.

Load a `RigidBodyTree` object that models a KUKA LBR robot. Use the `homeConfiguration` function to get the home configuration or home joint positions of the robot.

```
load('exampleLBR.mat','lbr')
lbr.DataFormat = 'column';

homeConfig = homeConfiguration(lbr);
```

Open the model. If necessary, use the **Load Robot Model** callback button to reload the robot model and configuration vector.

```
open_system('velocity_product_example.slx')
```


homeConfig
Constant
[0 0 0.5 0 0 0 0]
Joint Velcities
lbr
Config
JointTorq
JointVel
Velocity Product Torque
Joint Torques
lbr
Config
JointVel
JointTorq
JointAccel
FExt
Inverse Dynamics
zeros(7,1)
Constant1
zeros(6,lbr.NumBodies)
Constant2
Joint Torques2
Copyright 2018 The MathWorks, Inc.

Run the model. The Velocity Product block calculates the torques induced by the given velocities. Verify these values by passing the same velocities to the Inverse Dynamics block with no acceleration or external forces.

See Also

Blocks
Forward Dynamics | Get Jacobian | Gravity Torque | Inverse Dynamics | Joint Space Mass Matrix

Classes
`RigidBodyTree`

Functions
`externalForce` | `homeConfiguration` | `importrobot` | `randomConfiguration`

Related Examples

- "Perform Safe Trajectory Tracking Control Using Robotics Manipulator Blocks"

Compute Geometric Jacobian for Manipulators in Simulink

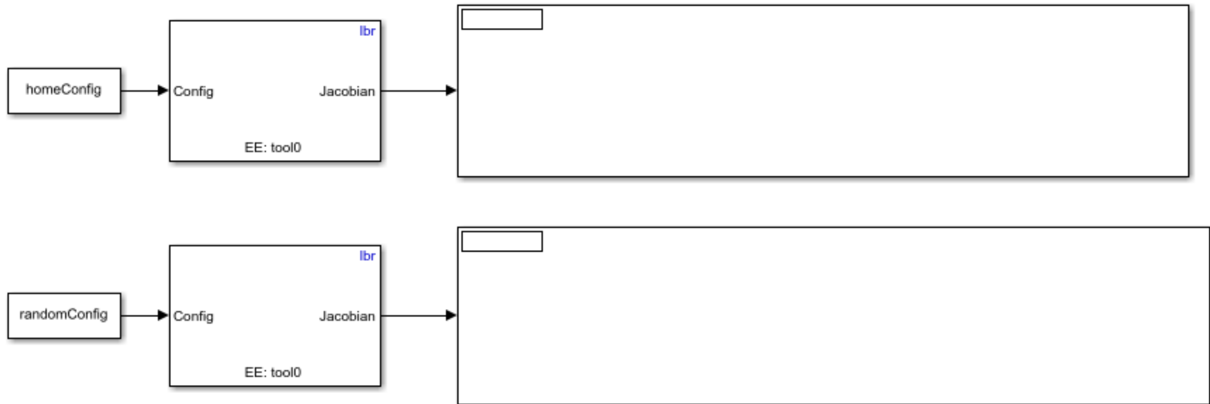
This example shows how to calculate the geometric Jacobian for a robot manipulator by using a `robotics.RigidBodyTree` model. The Jacobian maps the joint-space velocity to the end-effector velocity relative to the base coordinate frame. In this example, you define a robot model and robot configurations in MATLAB® and pass them to Simulink® to be used with the manipulator algorithm blocks.

Load a `RigidBodyTree` object that models a KUKA LBR robot. Use the `homeConfiguration` function to get the home configuration or home joint positions of the robot. Use the `randomConfiguration` function to generate a random configuration within the specified joint limits.

```
load('exampleRobots.mat','lbr')
lbr.DataFormat = 'column';
homeConfig = homeConfiguration(lbr);
randomConfig = randomConfiguration(lbr);
```

Open the model. If necessary, use the **Load Robot Model** callback button to reload the robot model and the configuration vectors. The `'tool0'` body is selected as the end-effector in both blocks.

```
open_system('get_jacobian_example.slx')
```



Run the model to display the Jacobian for each configuration.

See Also

Blocks
Forward Dynamics | Get Jacobian | Gravity Torque | Inverse Dynamics | Joint Space Mass Matrix

Classes
`RigidBodyTree`

Functions
`externalForce` | `homeConfiguration` | `importrobot` | `randomConfiguration`

Related Examples

- "Perform Safe Trajectory Tracking Control Using Robotics Manipulator Blocks"

Get Transformations for Manipulator Bodies in Simulink

This example shows how to get the transformation between bodies in a `robotics.RigidBodyTree` robot model. In this example, you define a robot model and robot configuration in MATLAB® and pass them to Simulink® to be used with the manipulator algorithm block.

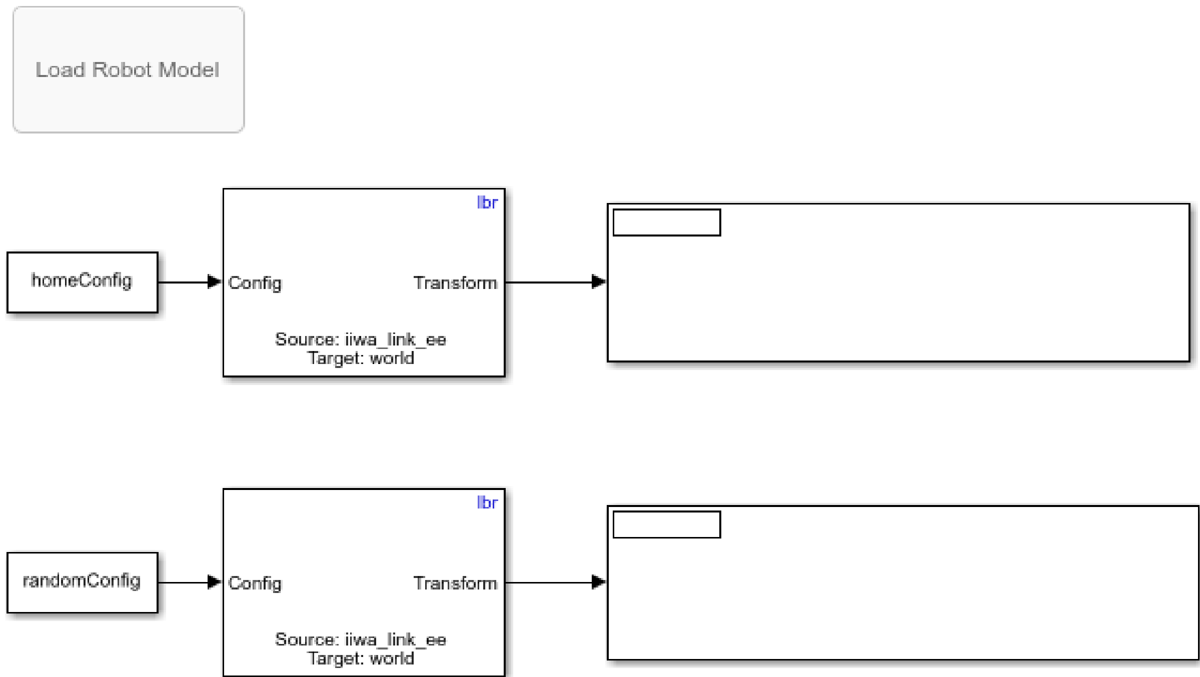
Load the robot model of the KUKA LBR robot as a `RigidBodyTree` object. Use the `homeConfiguration` function to get the home configuration as joint positions of the robot.

```
load('exampleLBR.mat','lbr')
lbr.DataFormat = 'column';

homeConfig = homeConfiguration(lbr);
randomConfig = randomConfiguration(lbr);
```

Open the model. If necessary, use the **Load Robot Model** callback button to reload the robot model and configuration vectors.

The Get Transform block calculates the transformation from the source body to the target body. This transformation converts coordinates from the source body frame to the given target body frame. This example gives you transformations to convert coordinates from the `'iiwa_link_ee'` end effector into the `'world'` base coordinates.

```
open_system('get_transform_example.slx')
```



Run the model to get the transformations.

See Also

Blocks
Forward Dynamics | Get Jacobian | Get Transform | Gravity Torque | Inverse Dynamics | Joint Space Mass Matrix

Classes
`RigidBodyTree`

Functions
`homeConfiguration` | `importrobot` | `randomConfiguration`

Related Examples

- "Perform Safe Trajectory Tracking Control Using Robotics Manipulator Blocks"

Get Mass Matrix for Manipulators in Simulink

This example shows how to calculate the mass matrix for a robot manipulator using a `robotics.RigidBodyTree` model. In this example, you define a robot model and robot configurations in MATLAB® and pass them to Simulink® to be used with the manipulator algorithm blocks.

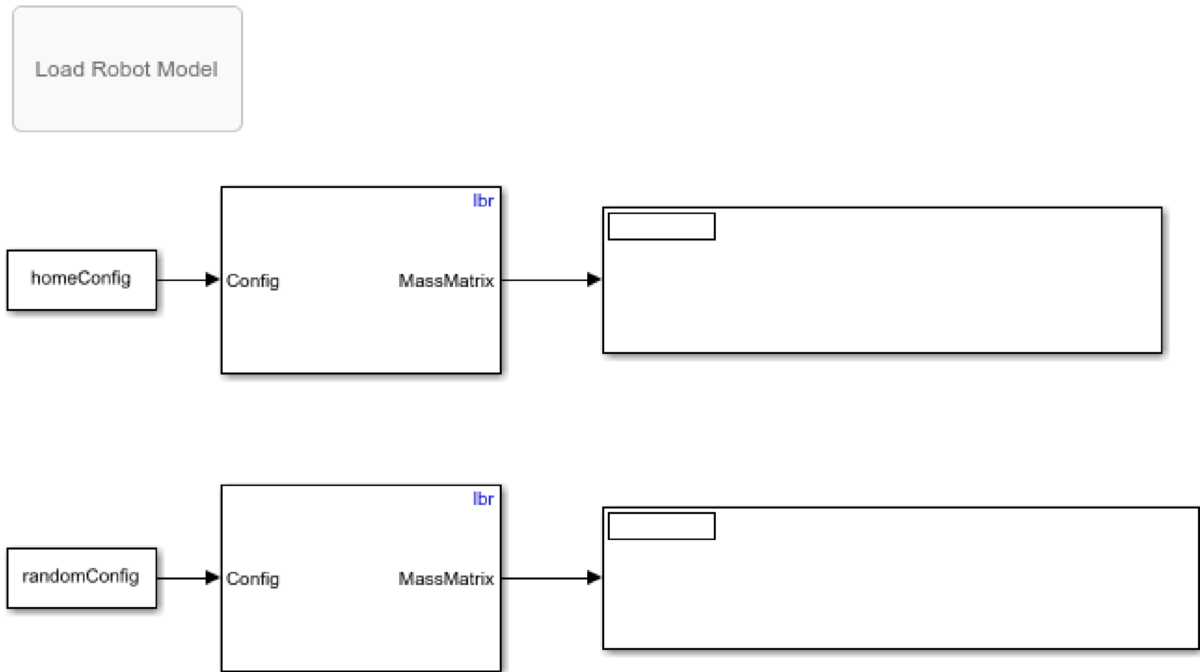
Load a `RigidBodyTree` object that models a KUKA LBR robot. Use the `homeConfiguration` functions to get the home configuration or home joint positions of the robot. Use the `randomConfiguration` function to generate a random configuration within the robot joint limits.

```
load('exampleRobots.mat','lbr')
lbr.DataFormat = 'column';

homeConfig = homeConfiguration(lbr);
randomConfig = randomConfiguration(lbr);
```

Open the model. If necessary, use the **Load Robot Model** callback button to reload the robot model and configuration vectors.

The Joint Space Mass Matrix block calculates the mass matrix for the given configuration.

```
open_system('mass_matrix_example.slx')
```



Run the model to display the mass matrices for each configuration.

See Also

Blocks
Forward Dynamics | Get Jacobian | Get Transform | Gravity Torque | Inverse Dynamics

Classes
`RigidBodyTree`

Functions
`homeConfiguration` | `importrobot` | `randomConfiguration`

Related Examples

- "Perform Safe Trajectory Tracking Control Using Robotics Manipulator Blocks"

Application Design

- “Transform Laser Scan Data From A ROS Network” on page 9-2
- “Obstacle Avoidance with TurtleBot and VFH” on page 9-4
- “Execute Code at a Fixed-Rate” on page 9-7
- “Reduce Drift in 3-D Visual Odometry Trajectory Using Pose Graphs” on page 9-14
- “Build Occupancy Map from Depth Images Using Visual Odometry and Optimized Pose Graph” on page 9-18

Transform Laser Scan Data From A ROS Network

Transform laser scan data using a ROS transformation tree. When working with laser scan data, your sensor might not be mounted in the center of the robot. Many algorithms make this assumption, so that you might need to transform your data so it is relative to the robot center. This example uses a ROS transformation tree to receive the transformations between different coordinate frames. To transform the sensor data, you must be connected to a ROS network and have transformations available.

Setup and connect to a ROS network. Specify the IP address of the ROS device.

```
rosinit('192.168.203.129')
```

```
Initializing global node /matlab_global_node_43056 with NodeURI http://192.168.203.1:5
```

Create the ROS transformation tree using `rostf`. Get the transform between the `'/camera_link'` and `'/base_link'` coordinate frames. These coordinate frame names are dependent on your robot configuration.

```
tftree = rostf;
pause(1);
tf = getTransform(tftree,'/camera_link','/base_link',rostime('now'));
```

Extract the rotation and translation matrices from the transform.

```
quat = [tf.Transform.Rotation.W,...
        tf.Transform.Rotation.X,...
        tf.Transform.Rotation.Y,...
        tf.Transform.Rotation.Z];
rotm = quat2rotm(quat);
trvec = [tf.Transform.Translation.X,...
         tf.Transform.Translation.Y ...
         tf.Transform.Translation.Z];
```

Create a homogeneous transform by combining the translation and rotation matrices.

```
tform = trvec2tform(trvec);
tform(1:3,1:3) = rotm(1:3,1:3);
```

Set up a subscriber to get laser scan data. Get the laser scan data as Cartesian points. Pad the points with zeros for the z-axis and convert them to homogeneous coordinates.

```
scansub = rossubscriber('/scan');
scan = receive(scansub)
```

```
cartScanData = scan.readCartesian;
cartScanData(:,3) = 0;
homScanData = cart2hom(cartScanData);
```

```
scan = 

  ROS LaserScan message with properties:

       MessageType: 'sensor_msgs/LaserScan'
            Header: [1×1 Header]
          AngleMin: -0.5216
          AngleMax: 0.5243
    AngleIncrement: 0.0016
     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
```

Apply the homogeneous transform and convert scan data back to Cartesian points.

```
trPts = tform*homScanData';
cartScanDataTransformed = hom2cart(trPts');
```

Get the polar angles and ranges from the Cartesian Points.

```
[angles,ranges] = cart2pol(cartScanDataTransformed(:,1),...
                           cartScanDataTransformed(:,2));
```

Shutdown ROS network.

```
rosshutdown
```

```
Shutting down global node /matlab_global_node_43056 with NodeURI http://192.168.203.1:5
```

See Also

apply | getTransform | robotics.VectorFieldHistogram | rostf

Obstacle Avoidance with TurtleBot and VFH

This example shows how to use a TurtleBot® with Vector Field Histograms (VFH) to perform obstacle avoidance when driving a robot in an environment. The robot wanders by driving forward until obstacles get in the way. The `robotics.VectorFieldHistogram` class computes steering directions to avoid objects while trying to drive forward.

Optional: If you do not already have a TurtleBot (simulated or real) set up, install a virtual machine with the Gazebo simulator and TurtleBot package. See "Get Started with Gazebo and a Simulated TurtleBot" to install and set up a TurtleBot in Gazebo.

Connect to the TurtleBot using the IP address obtained from setup.

```
rosinit('192.168.203.129')
```

```
Initializing global node /matlab_global_node_96529 with NodeURI http://192.168.203.1:60
```

Create a publisher and subscriber to share information with the VFH class. The subscriber receives the laser scan data from the robot. The publisher sends velocity commands to the robot.

The topics used are for the simulated TurtleBot. Adjust the topic names for your specific robot.

```
laserSub = rossubscriber('/scan');
[velPub, velMsg] = rospublisher('/mobile_base/commands/velocity');
```

Set up VFH object for obstacle avoidance. Specify algorithm properties for robot specifications. Set target direction to `0` in order to drive straight.

```
vfh = robotics.VectorFieldHistogram;
vfh.DistanceLimits = [0.05 1];
vfh.RobotRadius = 0.1;
vfh.MinTurningRadius = 0.2;
vfh.SafetyDistance = 0.1;

targetDir = 0;
```

Set up a Rate object using `robotics.Rate`, which can track the timing of your loop. This object can be used to control the rate the loop operates as well.

```
rate = robotics.Rate(10);
```

Create a loop that collects data, calculates steering direction, and drives the robot. Set a loop time of 30 seconds.

Use the ROS subscriber to collect laser scan data. Calculate the steering direction with the VFH object based on the input laser scan data. Convert the steering direction to a desired linear and an angular velocity. If a steering direction is not found, the robot stops and searches by rotating in place.

Drive the robot by sending a message containing the angular velocity and the desired linear velocity using the ROS publisher.

```
while rate.TotalElapsedTime < 30

    % Get laser scan data
    laserScan = receive(laserSub);
    ranges = double(laserScan.Ranges);
    angles = double(laserScan.readScanAngles);

    % Call VFH object to computer steering direction
    steerDir = vfh(ranges, angles, targetDir);

    % Calculate velocities
    if ~isnan(steerDir) % If steering direction is valid
        desiredV = 0.2;
        w = exampleHelperComputeAngularVelocity(steerDir, 1);
    else % Stop and search for valid direction
        desiredV = 0.0;
        w = 0.5;
    end

    % Assign and send velocity commands
    velMsg.Linear.X = desiredV;
    velMsg.Angular.Z = w;
    velPub.send(velMsg);
end
```

This code shows how you can use the Robotics System Toolbox™ algorithms to control robots and react to dynamic changes in their environment. Currently the loop ends after 30 seconds, but other conditions can be set to exit the loop based on information on the ROS network (i.e. robot position or number of laser scan messages).

Disconnect from the ROS network

```
rosshutdown
```

```
Shutting down global node /matlab_global_node_96529 with NodeURI http://192.168.203.1:6
```

See Also

robotics.VectorFieldHistogram | rospublisher | rossubscriber

Related Examples

- “Get Started with Gazebo and a Simulated TurtleBot”
- “Communicate with the TurtleBot”

Execute Code at a Fixed-Rate

In this section...

"Introduction" on page 9-7

"Send Fixed-rate Control Commands To A Robot" on page 9-7

"Fixed-rate Publishing of ROS Image Data" on page 9-9

"Overrun Actions for Fixed Rate Execution" on page 9-11

Introduction

Using the `robotics.Rate` object or the `rosrate` function allows you to time iterations of your robotics applications. By executing code at constant intervals, you can accurately time and schedule tasks. These examples show different applications for the `Rate` object including its uses with ROS and sending commands for robot control.

Depending on your application, the `rosrate` function and `robotics.Rate` object offer different options. If you would like to execute code based on the system time of your computer, create an object using `robotics.Rate`. However, if you are connected to a ROS network and want to base code execution on the ROS time, you can use the `rosrate` function.

Send Fixed-rate Control Commands To A Robot

This example shows to send regular commands to a robot at a fixed rate. It uses the `Rate` object to execute a loop that publishes `std_msgs/Twist` messages to the network at a regular interval.

Setup ROS network. Specify the IP address if your ROS network already exists.

```
rosinit
```

```
Initializing ROS master on http://AH-SRADFORD:11311/.
Initializing global node /matlab_global_node_26621 with NodeURI http://AH-SRADFORD:5008
```

Setup publisher and message for sending `Twist` commands.

```
[pub,msg] = rospublisher('/cmd_vel','geometry_msgs/Twist');
msg.Linear.X = 0.5;
msg.Angular.Z = -0.5;
```

Create `Rate` object with specified loop parameters.

```
desiredRate = 10;
rate = robotics.Rate(desiredRate);
rate.OverrunAction = 'drop'
```

```
rate = 

  Rate with properties:

         DesiredRate: 10
       DesiredPeriod: 0.1000
       OverrunAction: 'drop'
    TotalElapsedTime: 0.0300
          LastPeriod: NaN
```

Run loop and hold each iteration using `waitfor(rate)`. Send the `Twist` message inside the loop. Reset the `Rate` object before the loop to reset timing.

```
reset(rate)

while rate.TotalElapsedTime < 10
   send(pub,msg)
   waitfor(rate);
end
```

View statistics of fixed-rate execution. Look at `AveragePeriod` to verify the desired rate was maintained.

```
statistics(rate)
```

```
ans = 

  struct with fields:

              Periods: [1×100 double]
           NumPeriods: 100
        AveragePeriod: 0.1000
    StandardDeviation: 3.4189e-04
          NumOverruns: 0
```

Shut down ROS network

```
rosshutdown
```

```
Shutting down global node /matlab_global_node_26621 with NodeURI http://AH-SRADFORD:500
Shutting down ROS master on http://AH-SRADFORD:11311/.
```

Fixed-rate Publishing of ROS Image Data

This example shows how to regularly publish and receive image messages using ROS and the `rosrate` function. The `rosrate` function creates a `Rate` object to regularly access the `/camera/rgb/image_raw` topic on the ROS network using a subscriber. The rgb image is converted to a grayscale using `rgb2gray` and republished at a regular interval. Parameters such as the IP address and topic names vary with your robot and setup.

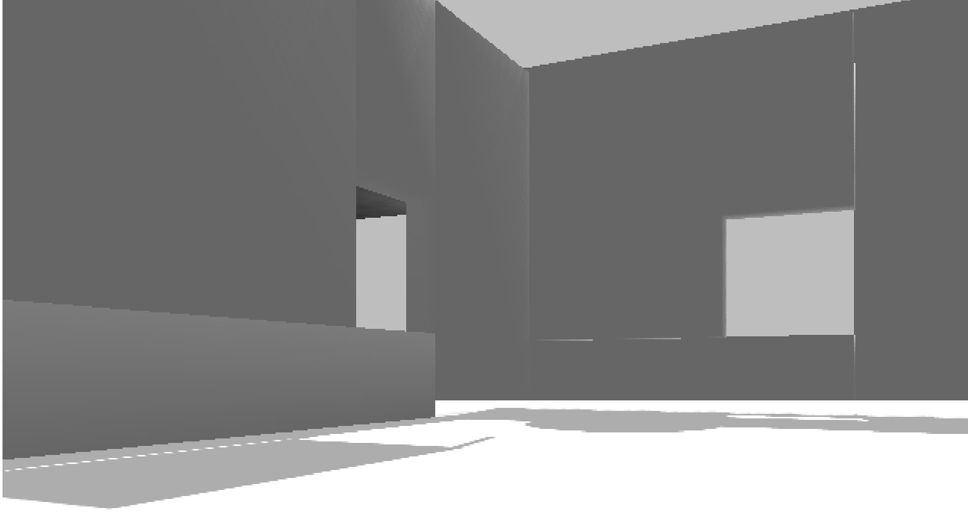
Connect to ROS network. Setup subscriber, publisher, and data message.

```
ipaddress = '172.28.194.188'; % Replace with your network address
rosinit(ipaddress)
sub = rossubscriber('/camera/rgb/image_raw');
pub = rospublisher('/camera/gray/image_gray','sensor_msgs/Image');
msgGray = rosmessage('sensor_msgs/Image');
msgGray.Encoding = 'mono8';
```

```
Initializing global node /matlab_global_node_04172 with NodeURI http://172.28.194.235:6
```

Receive the first image message. Extract image and convert to a grayscale image. Display grayscale image and publish the message.

```
msgImg = receive(sub);

img = readImage(msgImg);
grayImg = rgb2gray(img);
imshow(grayImg)

writeImage(msgGray,grayImg)
send(pub,msgGray)
```



Create ROS `Rate` object to execute at 10 Hz. Set a loop time and the `OverrunAction` for handling

```
desiredRate = 10;
loopTime = 5;
overrunAction = 'slip';
rate = rosrate(desiredRate);
r.OverrunAction = overrunAction;
```

Begin loop to receive, process and send messages every 0.1 seconds (10 Hz). Reset the `Rate` object before beginning.

```
reset(rate)

for i = 1:desiredRate*loopTime

    msgImg = receive(sub);

    img = readImage(msgImg);
    grayImg = rgb2gray(img);
    writeImage(msgGray,grayImg)

    send(pub,msgGray)

    waitfor(rate);
end
```

View the statistics for the code execution. `AveragePeriod` and `StandardDeviation` show how well the code maintained the `desiredRate`. `OverRuns` occur when data processing takes longer than the desired period length.

```
statistics(rate)
```

```
ans = 

              Periods: [1x50 double]
           NumPeriods: 50
        AveragePeriod: 0.1024
    StandardDeviation: 0.0193
          NumOverruns: 1
```

Shut down ROS node

```
rosshutdown
```

```
Shutting down global node /matlab_global_node_04172 with NodeURI http://172.28.194.235
```

Overrun Actions for Fixed Rate Execution

The `Rate` object uses the `OverrunAction` property to decide how to handle code that takes longer than the desired period to operate. The options are `'slip'` (default) or `'drop'`. This example shows how the `OverrunAction` affects code execution.

Setup desired rate and loop time. `slowFrames` is an array of times when the loop should be stalled longer than the desired rate.

```
desiredRate = 1;
loopTime = 20;
slowFrames = [3 7 12 18];
```

Create the `Rate` object and specify the `OverrunAction` property. `'slip'` indicates that the `waitfor` function will return immediately if the time for `LastPeriod` is greater than the `DesiredRate` property.

```
rate = robotics.Rate(desiredRate);
rate.OverrunAction = 'slip';
```

Reset `Rate` object and begin loop. This loop will execute at the desired rate until the loop time is reached. When the `TotalElapsedTime` reaches a slow frame time, it will stall for longer than the desired period.

```
reset(rate);

while rate.TotalElapsedTime < loopTime
    if ~isempty(find(slowFrames == floor(rate.TotalElapsedTime)))
        pause(desiredRate + 0.1)
    end
    waitfor(rate);
end
```

View statistics on the `Rate` object. Notice the number of periods.

```
stats = statistics(rate)
```

```
stats = struct with fields:
              Periods: [1x20 double]
           NumPeriods: 20
        AveragePeriod: 1.0209
    StandardDeviation: 0.0430
          NumOverruns: 4
```

Change the `OverrunAction` to `'drop'`. `'drop'` indicates that the `waitfor` function will return at the next time step, even if the `LastPeriod` is greater than the `DesiredRate` property. This effectively drops the iteration that was missed by the slower code execution.

```
rate.OverrunAction = 'drop';
```

Reset `Rate` object and begin loop.

```
reset(rate);

while rate.TotalElapsedTime < loopTime
    if ~isempty(find(slowFrames == floor(rate.TotalElapsedTime)))
        pause(1.1)
    end
    waitfor(rate);
end
stats2 = statistics(rate)
```

```
stats2 = struct with fields:
              Periods: [1x16 double]
           NumPeriods: 16
        AveragePeriod: 1.2501
    StandardDeviation: 0.4494
          NumOverruns: 4
```

Using the `'drop'` over run action resulted in 16 periods when the `'slip'` resulted in 20 periods. This difference is because the `'slip'` did not wait until the next interval based on the desired rate. Essentially, using `'slip'` tries to keep the `AveragePeriod` property as close to the desired rate. Using `'drop'` ensures the code will execute at an even interval relative to `DesiredRate` with some iterations being skipped.

See Also

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

Reduce Drift in 3-D Visual Odometry Trajectory Using Pose Graphs

This example shows how to reduce the drift in the estimated trajectory (location and orientation) of a monocular camera using 3-D pose graph optimization.

Visual odometry estimates the current global pose of the camera (current frame). Because of poor matching or errors in 3-D point triangulation, robot trajectories often tends to drift from the ground truth. Loop closure detection and pose graph optimization reduce this drift and correct for errors.

Load Estimated Poses for Pose Graph Optimization

Load the estimated camera poses and loop closure edges. Estimated camera poses are computed using visual odometry. Loop closure edges are computed by finding previous frame which saw the current scene and estimating the relative pose between the current frame and the loop closure candidate. Camera frames are sampled from [1].

```
% Estimated poses
load('estimatedpose.mat');
% Loopclosure edge
load('loopedge.mat');
% Groundtruth camera locations
load('groundtruthlocations.mat');
```

Build 3-D Pose Graph

Create an empty pose graph.

```
posegraph3D = robotics.PoseGraph3D;
```

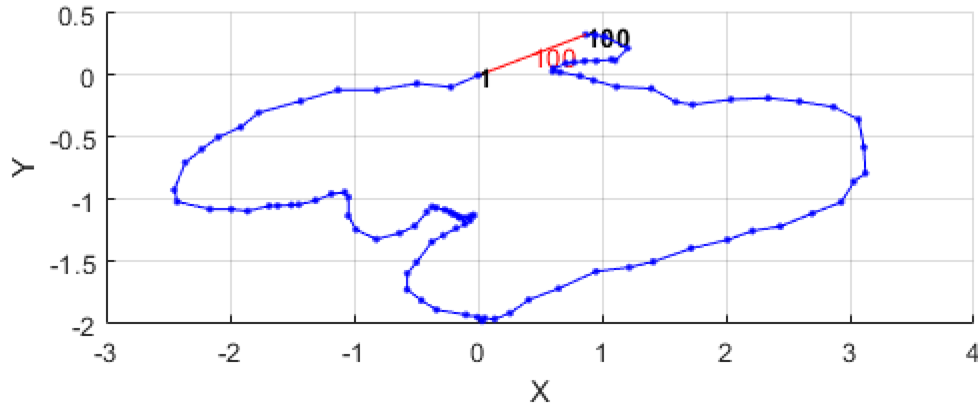
Add nodes to the pose graph, with edges defining the relative pose and information matrix for the pose graph. Convert the estimated poses, given as transformations, to relative poses as an `[x y theta qw qx qy qz]` vector. An identity matrix is used for the information matrix for each pose.

```
len = size(estimatedPose,2);
informationmatrix = [1 0 0 0 0 0 1 0 0 0 0 1 0 0 0 1 0 0 1 0 1];
% Insert relative poses between all successive frames
for k = 2:len
    % Relative pose between current and previous frame
    relativePose = estimatedPose{k-1}/estimatedPose{k};
```

```
    % Relative orientation represented in quaternions
    relativeQuat = tform2quat(relativePose);
    % Relative pose as [x y theta qw qx qy qz]
    relativePose = [tform2trvec(relativePose),relativeQuat];
    % Add pose to pose graph
    addRelativePose(posegraph3D,relativePose,informationmatrix);
end
```

Add a loop closure edge. Add this edge between two existing nodes from the current frame to a previous frame.

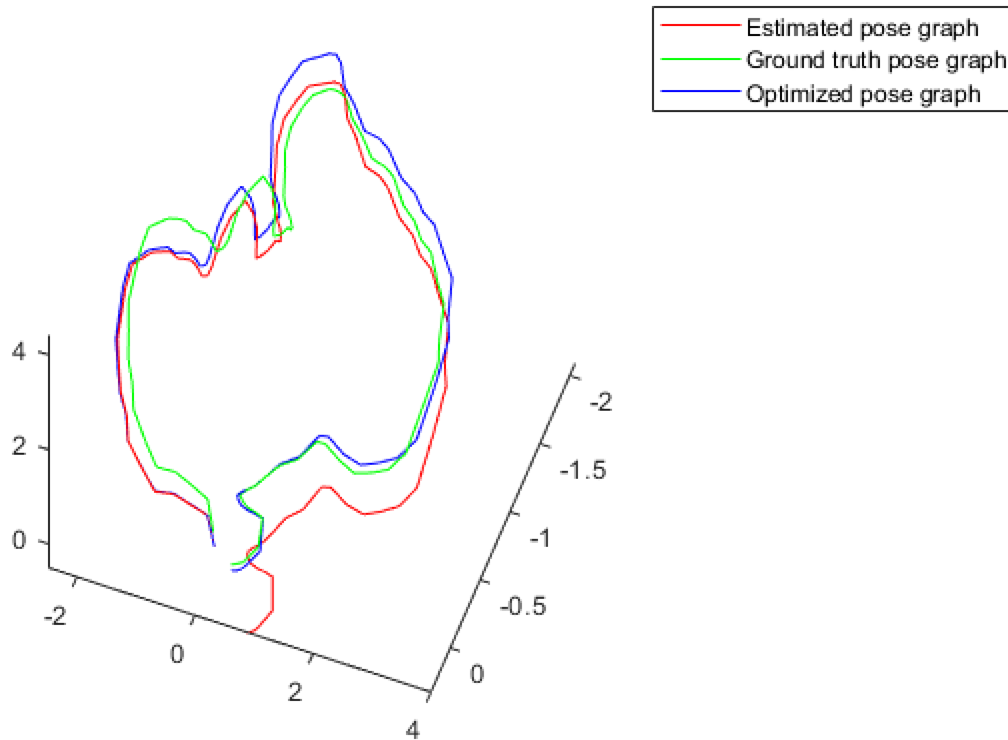
```
% Convert pose from transformation to pose vector.
relativeQuat = tform2quat(loopedge);
relativePose = [tform2trvec(loopedge),relativeQuat];
% Loop candidate
loopcandidateframeid = 1;
% Current frame
currentframeid = 100;

addRelativePose(posegraph3D,relativePose,informationmatrix,...
                loopcandidateframeid,currentframeid);

figure
show(posegraph3D);
```



Optimize the pose graph. The nodes are adjusted based on the edge constraints to improve the overall pose graph. To see the change in drift, plot the estimated poses and the new optimized poses against the ground truth.

```
% Pose graph optimization
optimizedPosegraph = optimizePoseGraph(posegraph3D);
optimizedposes = nodes(optimizedPosegraph);
% Camera trajectory plots
figure
estimatedposes = nodes(posegraph3D);
plot3(estimatedposes(:,1),estimatedposes(:,2),estimatedposes(:,3),'r');
hold on
plot3(groundtruthlocations(:,1),groundtruthlocations(:,2),groundtruthlocations(:,3),'g
plot3(optimizedposes(:,1),optimizedposes(:,2),optimizedposes(:,3),'b');
```

```
hold off
legend('Estimated pose graph','Ground truth pose graph', 'Optimized pose graph');
view(-20.8,-56.4);
```



References

[1] Galvez-López, D., and J. D. Tardós. "Bags of Binary Words for Fast Place Recognition in Image Sequences." *IEEE Transactions on Robotics.* Vol. 28, No. 5, 2012, pp. 1188-1197.

See Also

`addRelativePose` | `optimizePoseGraph` | `robotics.PoseGraph3D`

Build Occupancy Map from Depth Images Using Visual Odometry and Optimized Pose Graph

This example shows how to reduce the drift in the estimated trajectory (location and orientation) of a monocular camera using 3-D pose graph optimization. In this example, you build an occupancy map from the depth images, which can be used for path planning while navigating in that environment.

Load Estimated Poses for Pose Graph Optimization

Load the estimated camera poses and loop closure edges. The estimated camera poses were computed using visual odometry. The loop closure edges were computed by finding the previous frame that saw the current scene and estimating the relative pose between the current frame and the loop closure candidate. Camera frames are sampled from a data set that contains depth images, camera poses, and ground truth locations [1].

```
load('estimatedpose.mat');          % Estimated poses
load('loopedge.mat');               % Loopclosure edge
load('groundtruthlocations.mat');   % Ground truth camera locations
```

Build 3-D Pose Graph

Create an empty pose graph.

```
posegraph3D = robotics.PoseGraph3D;
```

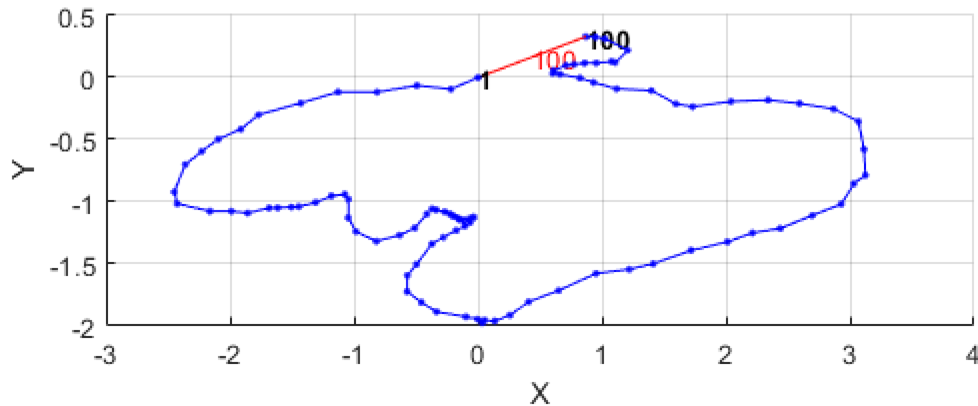
Add nodes to the pose graph, with edges defining the relative pose and information matrix for the pose graph. Convert the estimated poses, given as transformations, to relative poses as an `[x y theta qw qx qy qz]` vector. An identity matrix is used for the information matrix for each pose.

```
len = size(estimatedPose,2);
informationmatrix = [1 0 0 0 0 0 1 0 0 0 0 1 0 0 0 1 0 0 1 0 1];
% Insert relative poses between all successive frames
for k = 2:len
    % Relative pose between current and previous frame
    relativePose = estimatedPose{k-1}/estimatedPose{k};
    % Relative orientation represented in quaternions
    relativeQuat = tform2quat(relativePose);
    % Relative pose as [x y theta qw qx qy qz]
    relativePose = [tform2trvec(relativePose),relativeQuat];
    % Add pose to pose graph
```

```
    addRelativePose(posegraph3D,relativePose,informationmatrix);
end
```

Add a loop closure edge. Add this edge between two existing nodes from the current frame to a previous frame. Optimize the pose graph to adjust nodes based on the edge constraints and this loop closure. Store the optimized poses.

```
% Convert pose from transformation to pose vector.
relativeQuat = tform2quat(loopedge);
relativePose = [tform2trvec(loopedge),relativeQuat];
% Loop candidate
loopcandidateframeid = 1;
% Current frame
currentframeid = 100;

addRelativePose(posegraph3D,relativePose,informationmatrix,...
                loopcandidateframeid,currentframeid);

optimizedPosegraph = optimizePoseGraph(posegraph3D);
optimizedposes = nodes(optimizedPosegraph);

figure;
show(posegraph3D);
```



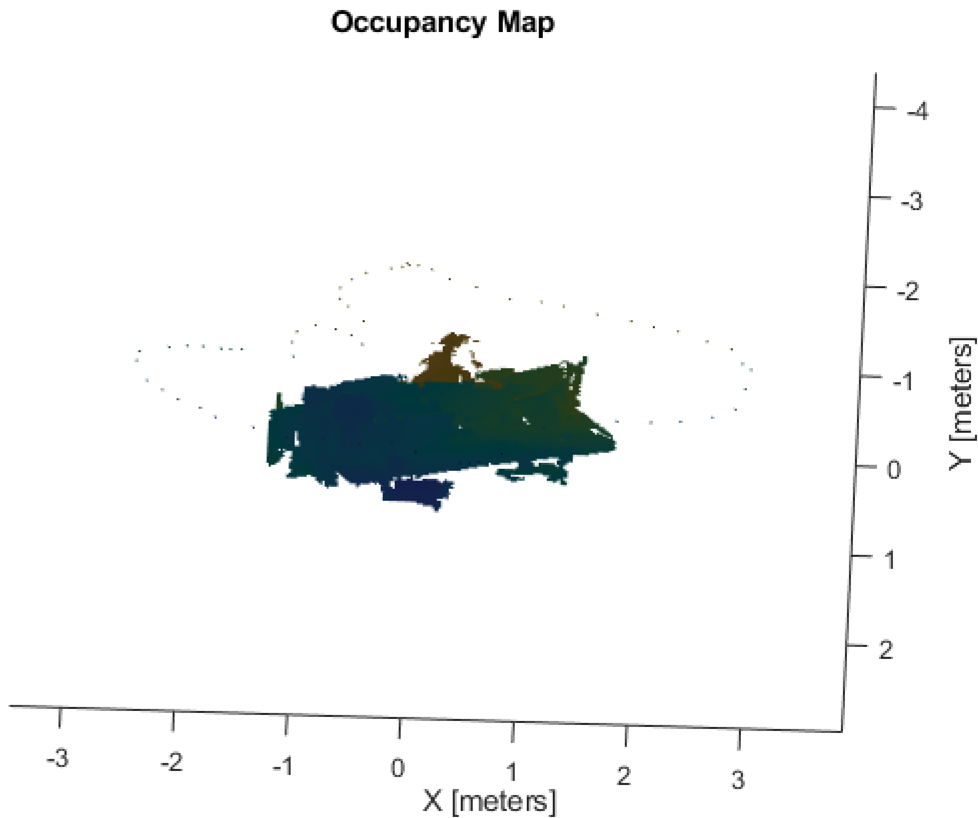
Create Occupancy Map from Depth Images and Optimized Poses

Load the depth images and camera parameters from the dataset [1].

```
load('depthimagearray.mat');    % variable depthImages
load('freburgK.mat');           % variable K
```

Create a 3-D occupancy map with a resolution of 50 cells per meter. Read in the depth images iteratively and convert the points in the depth image using the camera parameters and the optimized poses of the camera. Insert the points as point clouds at the optimized poses to build the map. Show the map after adding all the points. Because there are many depth images, this step can take several minutes. Consider uncommenting the `fprintf` command to print the progress the image processing.

```
occupancymap = robotics.OccupancyMap3D(50);

for k = 1:length(depthImages)
    points3D = exampleHelperExtract3DPointsFromDepthImage(depthImages{k},K);
    % fprintf('Processing Image %d\n', k);
    insertPointCloud(occupancymap,optimizedposes(k,:),points3D,1.5);
end
figure;
show(occupancymap);
view(-2.4,-90);
```



References

[1] Galvez-López, D., and J. D. Tardós. "Bags of Binary Words for Fast Place Recognition in Image Sequences." *IEEE Transactions on Robotics*. Vol. 28, No. 5, 2012, pp. 1188-1197.

See Also

`addRelativePose` | `insertPointCloud` | `optimizePoseGraph` | `robotics.OccupancyMap3D` | `robotics.PoseGraph3D`

Code Generation

- “Code Generation from MATLAB Code” on page 10-2
- “Code Generation Support, Usage Notes and Limitations” on page 10-4
- “Generate Code to Manually Deploy a ROS Node from Simulink” on page 10-8
- “Accelerate Robotics Algorithms with Code Generation” on page 10-14
- “Enable External Mode for Robotics System Toolbox Models” on page 10-18
- “Tune Parameters and View Signals on Deployed Robot Models Using External Mode” on page 10-19

Code Generation from MATLAB Code

Several Robotics System Toolbox functions are enabled to generate C/C++ code. Code generation from MATLAB code requires the MATLAB Coder™ product. To generate code from robotics functions, follow these steps:

- Write your function or application that uses Robotics System Toolbox functions that are enabled for code generation. For code generation, some of these functions have requirements that you must follow. See “Code Generation Support, Usage Notes and Limitations” on page 10-4.
- Add the `%#codegen` directive to your MATLAB code.
- Follow the workflow for code generation from MATLAB code using either the MATLAB Coder app or the command-line interface.

Using the app, the basic workflow is:

1. Set up a project. Specify your top-level functions and define input types.

   The app screens your code for code generation readiness. It reports issues such as a function that is not supported for code generation.
2. Check for run-time issues.

   The app generates and runs a MEX version of your function. This step detects issues that can be hard to detect in the generated C/C++ code.
3. Configure the code generation settings for your application.
4. Generate C/C++ code.
5. Verify the generated C/C++ code. If you have an Embedded Coder® license, you can use software-in-the-loop execution (SIL) or processor-in-the-loop (PIL) execution.

For a tutorial, see “C Code Generation Using the MATLAB Coder App” (MATLAB Coder).

Using the command-line interface, the basic workflow is:

- To detect issues and verify the behavior of the generated code, generate a MEX version of your function.
- Use `coder.config` to create a code configuration object for a library or executable.
- Modify the code configuration object properties as required for your application.
- Generate code using the `codegen` command.

- Verify the generated code. If you have an Embedded Coder license, you can use software-in-the-loop execution (SIL) or processor-in-the-loop (PIL) execution.

For a tutorial, see “C Code Generation at the Command Line” (MATLAB Coder).

See Also

More About

- “Code Generation Support, Usage Notes and Limitations” on page 10-4

Code Generation Support, Usage Notes and Limitations

To generate code from MATLAB code that contains Robotics System Toolbox functions, classes, or System objects, you must have the MATLAB Coder software.

The following functions support code generation using MATLAB Coder, but may have some limitations.

An asterisk (*) indicates that the reference page has usage notes and limitations for C/C++ code generation.

Algorithm Design
robotics.AimingConstraint
robotics.BinaryOccupancyGrid
robotics.CartesianBounds
robotics.GeneralizedInverseKinematics*
robotics.InverseKinematics*
robotics.Joint
robotics.JointPositionBounds
lidarScan
matchScans
matchScansGrid
robotics.OccupancyGrid
robotics.OdometryMotionModel
robotics.OrientationTarget
robotics.ParticleFilter*
robotics.PoseTarget
robotics.PositionTarget
robotics.PRM
robotics.PurePursuit
robotics.RigidBody
robotics.RigidBodyTree*

transformScan
robotics.VectorFieldHistogram
Coordinate System Transformations
angdiff
axang2quat
axang2rotm
axang2tform
cart2hom
classUnderlying of quaternion
compact of quaternion
conj of quaternion
ctranspose, ' of quaternion
dist of quaternion
eul2quat
eul2rotm
eul2tform
euler of quaternion
eulerd of quaternion
exp of quaternion
hom2cart
ldivide, .\ of quaternion
log of quaternion
meanrot of quaternion
minus, - of quaternion
mtimes, * of quaternion
norm of quaternion
normalize of quaternion
ones of quaternion
parts of quaternion

power, .^ of quaternion
prod of quaternion
quat2axang
quat2eul
quat2rotm
quat2tform
quaternion
rdivide, ./ of quaternion
rotateframe of quaternion
rotatepoint of quaternion
rotm2axang
rotm2eul
rotm2quat
rotm2tform
rotmat of quaternion
rotvec of quaternion
rotvecd of quaternion
slerp of quaternion
times, .* of quaternion
tform2axang
tform2eul
tform2quat
tform2rotm
tform2trvec
transpose, .' of quaternion
trvec2tform
uminus, - of quaternion
zeros of quaternion
UAV Algorithms

control
derivative
derivative
fixedwing
multirotor
state
uavWaypointFollower

See Also

More About

- “Code Generation from MATLAB Code” on page 10-2

Generate Code to Manually Deploy a ROS Node from Simulink

This example shows you how to generate C++ code from a Simulink model to deploy as a standalone ROS node. The code is generated on your computer and must be manually transferred to the target ROS device. No connection to the hardware is necessary for generated the code. For an automated deployment of a ROS node, see “Generate a Standalone ROS Node from Simulink®”.

Prerequisites

- This example requires Simulink Coder.
- A Ubuntu Linux system with ROS is necessary for building and running the generated C++ code. You can use your own Ubuntu ROS system, or you can use the Linux virtual machine used for Robotics System Toolbox examples. See “Get Started with Gazebo and a Simulated TurtleBot” for instructions on how to install and use the virtual machine.
- Review the “Feedback Control of a ROS-enabled Robot” example, which details the Simulink model that the code is being generated from.

Configure A Model for Code Generation

Configure a model to generate C++ code for a standalone ROS node using the **Configuration Parameters**. The model used here is the proportional controller introduced in the “Feedback Control of a ROS-enabled Robot” example.

Open the proportional controller model.

```
edit robotROSFeedbackControlExample
```

Copy the entire model to a new blank Simulink model. In the menu, Click **Edit > Select All**, then **Edit > Copy**.

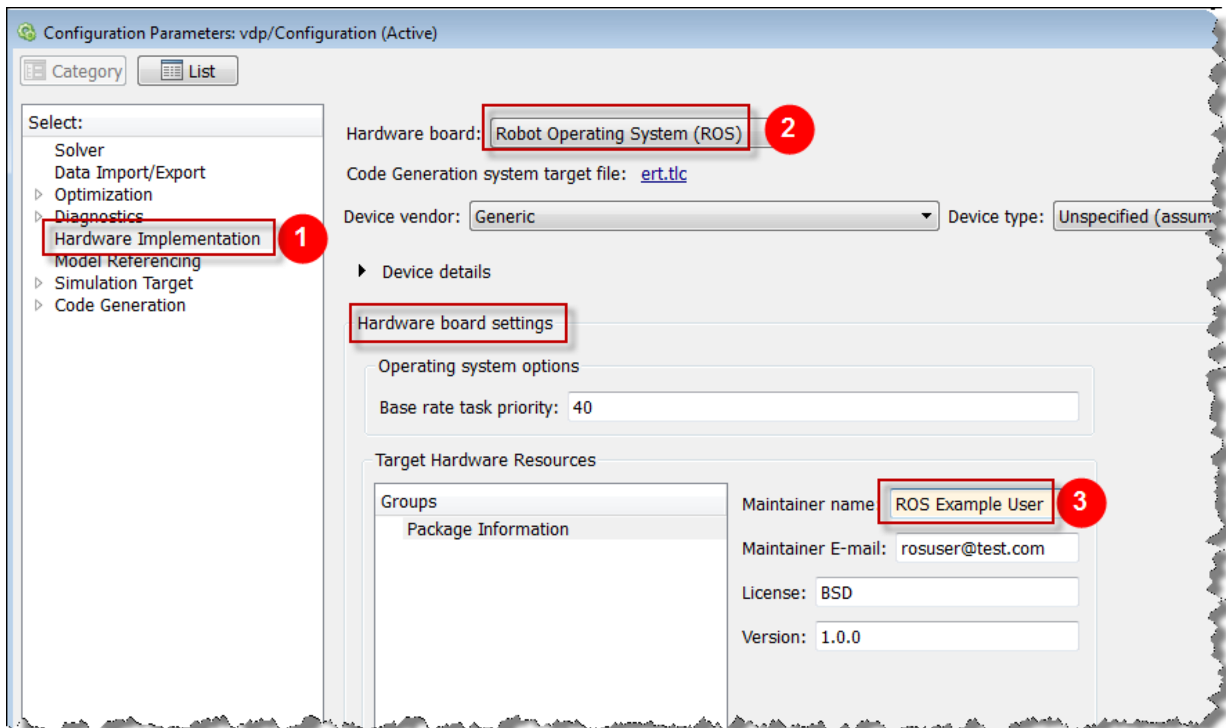
Open a new Simulink model. In the menu, Click **Edit > Paste**.

Delete the Simulation Rate Control block.

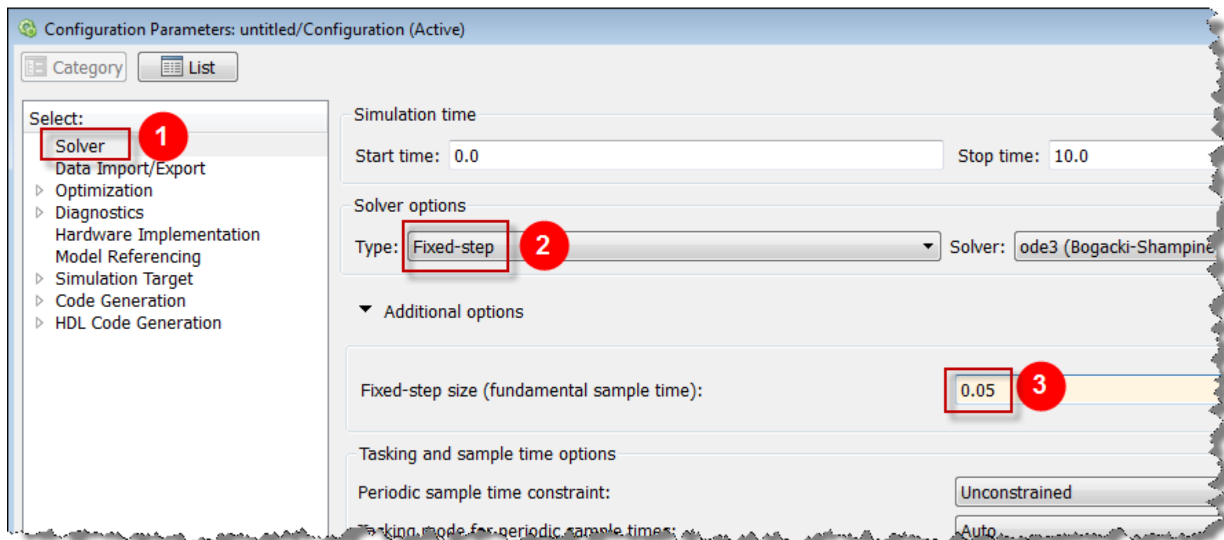
Open the **Configuration Parameters** dialog. Click **Simulation > Model Configuration Parameters**.

In the **Hardware Implementation** pane, set the **Hardware board** to `Robot Operating System (ROS)`.

The **Hardware board settings** section contains settings specific to the generated ROS package, such as information included in the `package.xml` file. Change **Maintainer name** to `ROS Example User` and click **OK**.



In the **Solver** pane of the **Configuration Parameters** dialog, ensure the **Type** is set to `Fixed-step`, and the **Fixed-step size** is set to `0.05`. In generated code, the fixed-step size defines the actual time step that is used for the model update loop. See “Execution of Code Generated from a Model” (Simulink Coder) for more information.



Click **OK** to close the **Configuration Parameters** dialog. Save the model as `RobotController.slx`.

Configure the Build Options for Code Generation

After configuring the model, you must specify the build options for the target hardware and set the folder or building the generated code.

Open the **Configuration Parameters** dialog. Click **Simulation** > **Model Configuration Parameters**.

In the **Hardware Implementation** tab, under **Hardware board settings**, click the **Build options** group. Set the **Build action** to None. This setting ensures that code generated for the ROS node without building it on an external ROS device.

▼ Target hardware resources

Groups
Package information
Device parameters
Build options
External mode

Build action: None
ROS folder:
Catkin workspace:
Edit

Generate and Deploy the Code

Start a ROS master in MATLAB. This ROS master is used by Simulink for the code generation steps.

In the MATLAB command window type:

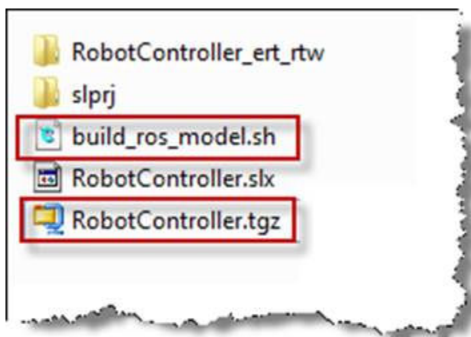
```
rosinit
```

Set the current folder to a writable directory. This folder is the location that generate code will be stored when you build the model.

In Simulink, click **Code** > **C/C++ Code** > **Deploy to Hardware** or press `Ctrl+B` to start code generation for the model.

Once the build completes, two new files are written to your folder.

- `RobotController.tgz`-- An archive containing the C++ code
- `build_ros_model.sh` -- A shell script for extracting and building the C++ code



Manually transfer the two files to the target machine. If you connect to a ROS device using `rosdevice`, you can send files using `putFile`. Otherwise, this step assumes you are using the Linux virtual machine used for Robotics System Toolbox examples. The virtual machine is configured to accept SSH and SCP connections. *If you are using your own Linux system, consult your system administrator for a secure way to transfer files.*

Ensure your host system (the system with your `RobotController.tgz` and `build_ros_model.sh` files) has an SCP client. For Windows® systems, the next step assumes that PuTTY SCP client (`pcsp.exe`) is installed.

Use SCP to transfer the files to the user home director on the Linux virtual machine. Username is `user` and password is `password`. Replace `<virtual_machine_ip>` with your virtual machines IP address.

- Windows host systems:

  ```
  pscp.exe RobotController.tgz build_ros_model.sh user@<virtual_machine_ip>:
  ```
- Linux or macOS host systems:

  ```
  scp RobotController.tgz build_ros_model.sh user@<virtual_machine_ip>:
  ```

The `build_ros_model.sh` file is not specific to this model. It only needs to be transferred once for multiple models.

On the Linux system, execute the following commands to create a Catkin workspace. You may use an existing Catkin workspace.

```
mkdir -p ~/catkin_ws_simulink/src
cd ~/catkin_ws_simulink/src
catkin_init_workspace
```

Decompress and build the node there using the following command in Linux. Replace `<path_to_catkin_ws>` with the path to your catkin workspace.

```
cd ~
./build_ros_model.sh RobotController.tgz <path_to_catkin_ws>
```

The generated source code is under `~/catkin_ws_simulink/src/robotcontroller/`. Review the contents of the `package.xml` file. Verify that the node executable was created using:

```
file ~/catkin_ws_simulink/devel/lib/robotcontroller/robotcontroller_node
```

If the executable was created successfully, the command lists information about the file.

The model is now ready to be run as a standalone ROS node on your device.

Optional: You can then run the node using this command. Replace `<path_to_catkin_ws>` with the path to your catkin workspace.

```
~/<path_to_catkin_workspace>/devel/lib/robotcontroller/robotcontroller_node
```

See Also

More About

- "Feedback Control of a ROS-enabled Robot"
- "Generate a Standalone ROS Node from Simulink®"
- "Tune Parameters and View Signals on Deployed Robot Models Using External Mode" on page 10-19

Accelerate Robotics Algorithms with Code Generation

In this section...

"Create Separate Function for Algorithm" on page 10-14
"Perform Code Generation for Algorithm" on page 10-15
"Check Performance of Generated Code" on page 10-15
"Replace Algorithm Function with MEX Function" on page 10-16

You can generate code for select Robotics System Toolbox algorithms to speed up their execution. Set up the algorithm that supports code generation as a separate function that you can insert into your workflow. To use code generation, you must have a MATLAB Coder license. For a summary of code generation support in Robotics System Toolbox, see "Code Generation".

For this example, a robot is wandering in an environment using the `VectorFieldHistogram` class with laser scan data to perform obstacle avoidance. The goal is to replace the vector field histogram (VFH) algorithm with a MEX file created from code generation.

To see this example without code generation, see "Obstacle Avoidance with TurtleBot and VFH" on page 9-4.

Create Separate Function for Algorithm

Create a separate function, `vfhCodeGen`, that runs the vector field algorithm. Create a VFH object and specify the algorithm parameters. Call the object as the main function to use the VFH algorithm. Specify `%#codegen` inside the function to identify it as a function for code generation.

```
function steerDir = vfhCodeGen(ranges,angles,targetDir)
    %#codegen
    vfh = robotics.VectorFieldHistogram;
    vfh.DistanceLimits = [0.05 1];
    vfh.RobotRadius = 0.1;
    vfh.MinTurningRadius = 0.2;
    vfh.SafetyDistance = 0.1;

    steerDir = vfh(ranges,angles,targetDir);
end
```

Save the function in your current folder.

Perform Code Generation for Algorithm

You can use either the `codegen` function or the **MATLAB Coder** app to generate code. In this example, you generate a MEX file by calling`codegen` on the MATLAB command line. Specify sample input arguments for each input to the function using the `-args` input argument

Specify sample values for the input arguments. Create a sample of the ranges, angles, and target directions. The TurtleBot laser scan gives 640 scans.

```
ranges = zeros(640,1);
angles = zeros(640,1);
targetDir = 0;
```

Call the `codegen` function and specify the input arguments in a cell array. This function creates a separate `vfhCodeGen_mex` function to use. You can also produce C code by using the `options` input argument.

```
codegen vfhCodeGen -args {ranges,angles,targetDir}
```

If your laser scan can come from different sources with variable-size lengths, specify the canonical type of the `ranges` and `angles` inputs by using `coder.typeof` with the `codegen` function.

```
codegen vfhCodeGen -args {coder.typeof(ranges,[Inf 1]), ...
                          coder.typeof(angles,[Inf 1]),targetDir}
```

Check Performance of Generated Code

Compare the timing of the generated MEX function to the timing of your original function by using `timeit`.

```
time = timeit(@() vfhCodeGen(ranges,angles,targetDir))
mexTime = timeit(@() vfhCodeGen_mex(ranges,angles,targetDir))
```

```
time =

    0.0039
```

```
mexTime =

   7.6490e-05
```

The MEX function runs over 50 times faster in this example. Results might vary in your system.

Replace Algorithm Function with MEX Function

Open the main function for running your robotics workflow. Replace the `vfh` object call with the MEX function that you created using code generation.

Open the “Obstacle Avoidance with TurtleBot and VFH” on page 9-4 example.

```
openExample('robotics/ObstacleAvoidanceWithTurtleBotAndVFHExample')
```

Modify the example code to use the new `vfhCodeGen_mex` function. The code that follows is a copy of the example with modified comments and use of the new MEX function. The definition of the VFH object is also removed.

Connect to the TurtleBot. Set up the laser scan subscriber, the velocity publisher, and the rate control object. Specify a starting target direction.

```
rosinit('192.168.154.131')
laserSub = rossubscriber('/scan');
[velPub, velMsg] = rospublisher('/mobile_base/commands/velocity');
rate = robotics.Rate(10);
targetDir = 0;
```

Create a loop that collects data, calculates the steering direction, and drives the robot. Set a loop time of 30 seconds. Replace the `step` call with `vfhCodeGen_mex`.

```
while rate.TotalElapsedTime < 30

    % Get laser scan data
    laserScan = receive(laserSub);
    ranges = double(laserScan.Ranges);
    angles = double(readScanAngles(laserScan));

    % Call MEX function created using code generation
    steerDir = vfhCodeGen_mex(ranges,angles,targetDir);

    % Calculate velocities
```

```
    if ~isnan(steerDir) % If steering direction is valid
        desiredV = 0.2;
        w = exampleHelperComputeAngularVelocity(steerDir,1);
    else % Stop and search for valid direction
        desiredV = 0.0;
        w = 0.5;
    end

    % Assign and send velocity commands
    velMsg.Linear.X = desiredV;
    velMsg.Angular.Z = w;
    send(velPub,velMsg);
end
```

The robot performs 30 seconds of obstacle avoidance using the MEX function. For this application, the time difference of the VFH algorithm is minimal, but you can use this example to help you replace other algorithms with generated code. Consider using code generation in areas of your code that slow down the workflow.

Disconnect from the ROS network.

```
rosshutdown
```

See Also

`VectorFieldHistogram` | `codegen` | `timeit`

Related Examples

- "Obstacle Avoidance with TurtleBot and VFH" on page 9-4
- "Code Generation Support, Usage Notes and Limitations" on page 10-4
- "Generate a Standalone ROS Node from Simulink®"
- "C Code Generation at the Command Line" (MATLAB Coder)
- "C Code Generation Using the MATLAB Coder App" (MATLAB Coder)

Enable External Mode for Robotics System Toolbox Models

External mode enables Simulink on your host computer to communicate with a deployed model on your robotics hardware during runtime. External mode allows you to tune block mask parameters and to visualize signals on your model while your model is running. For Robotics System Toolbox, deployed models are ROS nodes running on the target hardware that communicates with Simulink over TCP/IP.

To use external mode with Robotics System Toolbox models:

1. Open the Configuration Parameters dialog box.
2. On the **Hardware Implementation** pane, specify the **Hardware board** as `Robot Operating System (ROS)`. Specify related parameters in **Target Hardware Resources**.
3. In **Target Hardware Resources**, set the **External mode** parameters. Click **OK**.
4. In the model, set the **Simulation mode** to `External` for the model.
5. Run the model.

Your model connects to the **Device Address** specified in the “Connect to ROS Device” on page 6-15 dialog box which is used to connect to your ROS device when deploying the model.

To configure signal monitoring and data archiving, go to the **Code** menu and select **External Mode Control Panel**. You can also connect to the target program and start and stop execution of the model code. For more information, see “Host-Target Communication with External Mode Simulation” (Simulink Coder).

See Also

Related Examples

- “Generate a Standalone ROS Node from Simulink®”
- “Tune Parameters and View Signals on Deployed Robot Models Using External Mode” on page 10-19
- “Host-Target Communication with External Mode Simulation” (Simulink Coder)

Tune Parameters and View Signals on Deployed Robot Models Using External Mode

In this section...
"Set Up the Simulink Model" on page 10-19
"Deploy and Run the Model" on page 10-20
"Monitor Signals and Tune Parameters" on page 10-21

External mode enables Simulink models on your host computer to communicate with a deployed model on your robot hardware during runtime. Use external mode to view signals or modify block mask parameters on your deployed Simulink model. Parameter tuning with external mode helps you make adjustments to your algorithms as they run on the hardware as opposed to in simulation in Simulink itself. This example shows how to use external mode with the "Feedback Control of a ROS-enabled Robot" example when the model is deployed to the robot hardware.

Set Up the Simulink Model

Configure the Simulink model to deploy to the robot hardware and enable external mode.

Open the model.

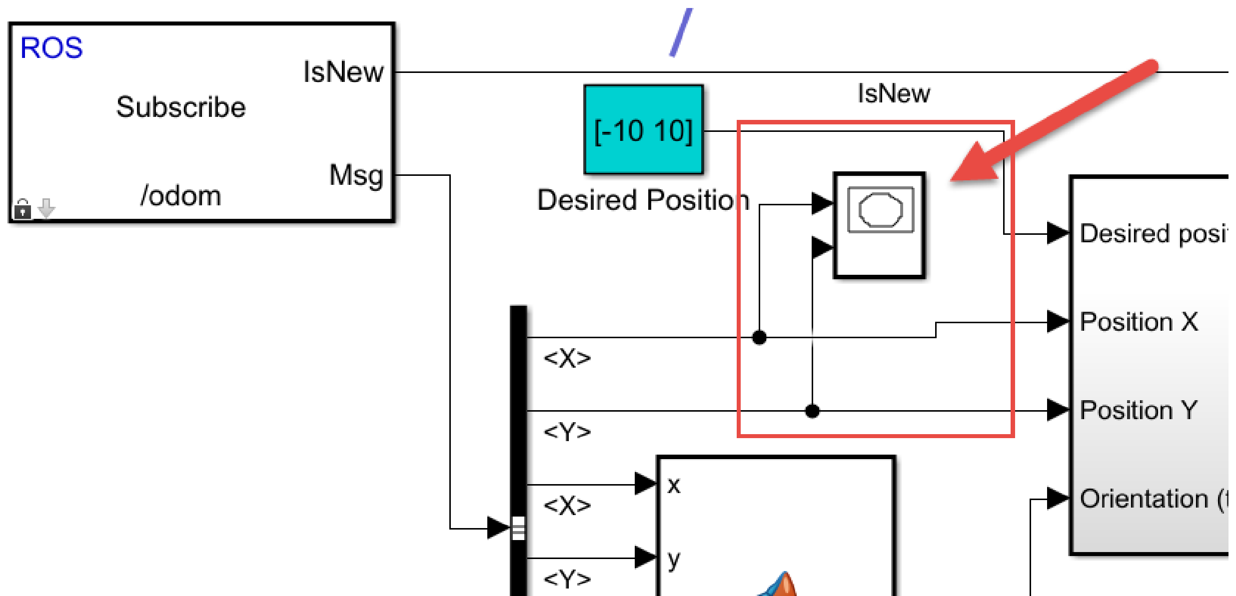
```
robotROSFeedbackControlExample
```

Set the configuration parameters of the model.

1. In the **Simulation** menu, click **Model Configuration Parameters** to open the Configuration Parameters dialog box.
2. On the **Solver** pane, set **Type** to `Fixed-step` and the **Fixed-step size** to `0.05`.
3. On the **Hardware Implementation** pane, specify the **Hardware board** as `Robot Operating System (ROS)`. Specify related parameters in **Target Hardware Resources**.
4. In **Build Options**, set the **Build action** to `Build and run`. By default, Simulink always uses `Build and run` when using external mode.
5. In **Target Hardware Resources**, set the **External mode** parameters. To prioritize model execution speed, enable **Run external mode in a background thread**.

6 In the model, set the **Simulation mode** to `External`.

In the model, add scope blocks to the signals you want to view. For this example, add an XY Graph scope to the X and Y signals that are attached to the ROS subscriber that monitors the robot position. Open the block and change the minimum and maximum values for each axis to `[-10 10]`.



Deploy and Run the Model

Now that the model is configured, you can deploy and run the model on the robot hardware.

Connect to the ROS network by setting the network address. The network must be running on your target robotics hardware. This example uses the "Gazebo Empty" simulator environment is used from the Virtual Machine with ROS Hydro and Gazebo for Robotics System Toolbox example. In the **Tools** menu, under **Robotics Operating System**, select **Configure Network Addresses**. Specify your device address by selecting `Custom` under **Network Address** and specifying the IP address or host name under **Hostname/IP Address**. For this virtual machine, the IP address is `192.168.154.131`.

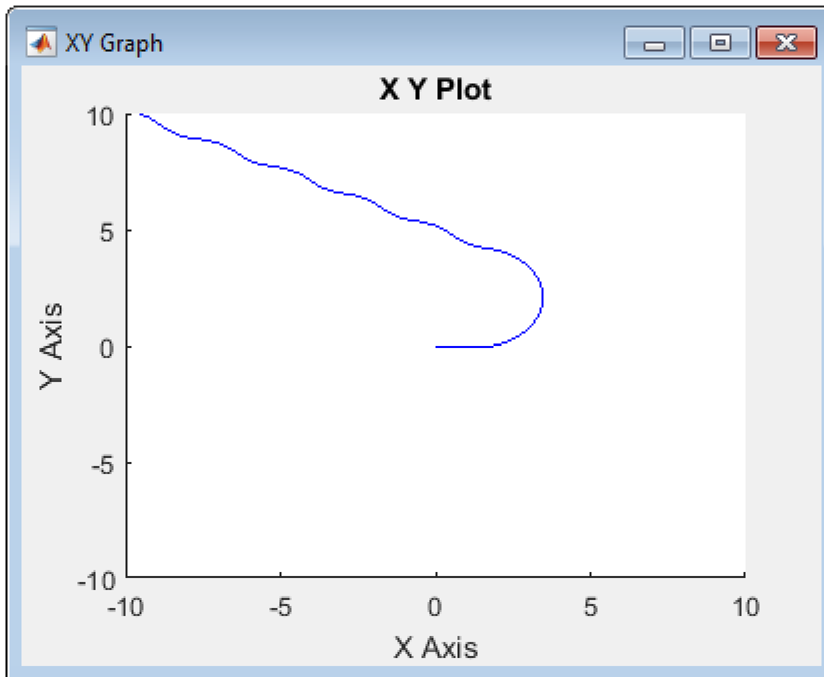
Run the model. The model is deployed to the robot hardware and runs after the build process is complete. This step might take some time.

Monitor Signals and Tune Parameters

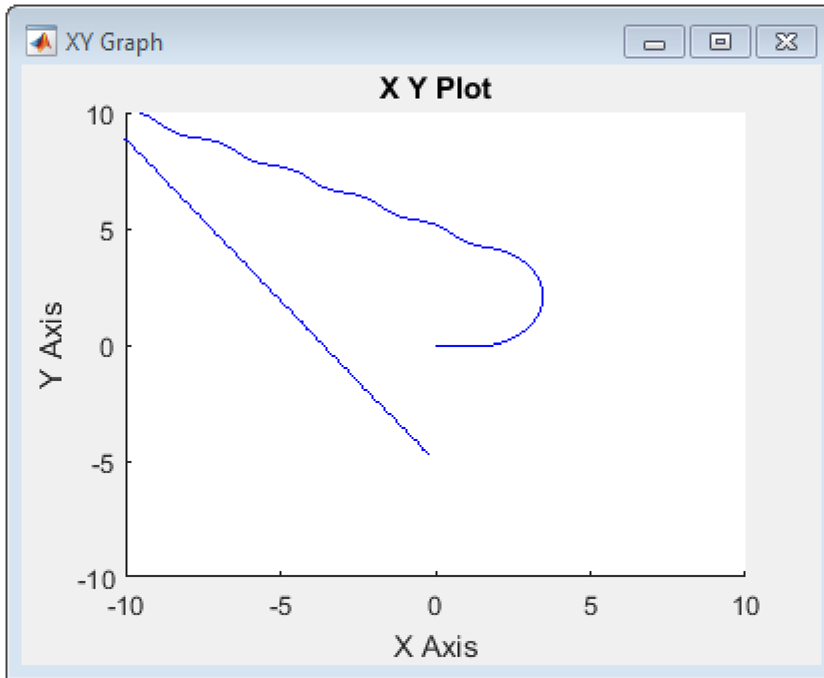
After you deploy the model and the model is running, you can view its signals and modify its parameters.

While the model runs on the hardware, view the XY Graph window to monitor the robot position over time.

The path has a slight wobble, which is due to the high velocity of the robot as it tracks the path.



While the model is still running, you can also tune parameters. Open the `Proportional Controller` subsystem and change the Linear Velocity slider to `0.25`. Back in the main model, change the Desired Position constant block to a new position, `[0 -5]`. The robot drives to the new position slower.



The lowered velocity reduces the wobble along the path. All these modifications were done while the model was deployed on the hardware.

See Also

Related Examples

- “Feedback Control of a ROS-enabled Robot”
- “Enable External Mode for Robotics System Toolbox Models” on page 10-18
- “Generate a Standalone ROS Node from Simulink®”
- “Host-Target Communication with External Mode Simulation” (Simulink Coder)