ROS Toolbox

User's Guide

MATLAB&SIMULINK®

MathWorks®

How to Contact MathWorks

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

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

Contents

3

4

ROS Featured Examples

Get Started with ROS

Robot Operating System (ROS) is a communication interface that enables different parts of a robot system to discover each other, and send and receive data between them. MATLAB® supports ROS with a library of functions that enables you to exchange data with ROS-enabled physical robots or robot simulators such as Gazebo®.

This example introduces how to:

- Set up ROS within MATLAB
- Get information about capabilities in a ROS network
- Get information about ROS messages

ROS Terminology

- A *ROS network* comprises different parts of a robot system (such as a planner or a camera interface) that communicate over ROS. The network can be distributed over several machines.
- A *ROS master* coordinates the different parts of a ROS network. It is identified by a *Master URI* (Uniform Resource Identifier) that specifies the hostname or IP address of the machine where the master is running.
- A *ROS node* contains a collection of related ROS capabilities (such as publishers, subscribers, and services). A ROS network can have many ROS nodes.
- *Publishers*, *subscribers*, and *services* are different kinds of ROS entities that process data. They exchange data using *messages*.
- A publisher sends messages to a specific *topic* (such as "odometry"), and subscribers to that topic receive those messages. A single topic can be associated with multiple publishers and subscribers.

For more information, see "Robot Operating System (ROS)" and the Concepts section on the ROS website.

Initialize ROS Network

Use `rosinit` to initialize ROS. By default, `rosinit` creates a *ROS master* in MATLAB and starts a *global node* that is connected to the master. The global node is automatically used by other ROS functions.

```
rosinit
```

```
Launching ROS Core...
...............................Done in 5.6997 seconds.
Initializing ROS master on http://192.168.203.1:58232.
Initializing global node /matlab_global_node_79626 with NodeURI http://ah-sradford:49876/
```

Use `rosnode list` to see all nodes in the ROS network. Note that the only available node is the global node created by `rosinit`.

```
rosnode list
```

```
/matlab_global_node_79626
```

Use `exampleHelperROSCreateSampleNetwork` to populate the ROS network with three additional nodes and sample publishers and subscribers.

```
exampleHelperROSCreateSampleNetwork
```

Use rosnode `list` again to see the three new nodes (`node_1`, `node_2,` and `node_3`).

rosnode `list`

```
/matlab_global_node_79626
/node_1
/node_2
/node_3
```

The figure shows the current state of the ROS network. The MATLAB global node is disconnected since it currently does not have any publishers, subscribers or services.
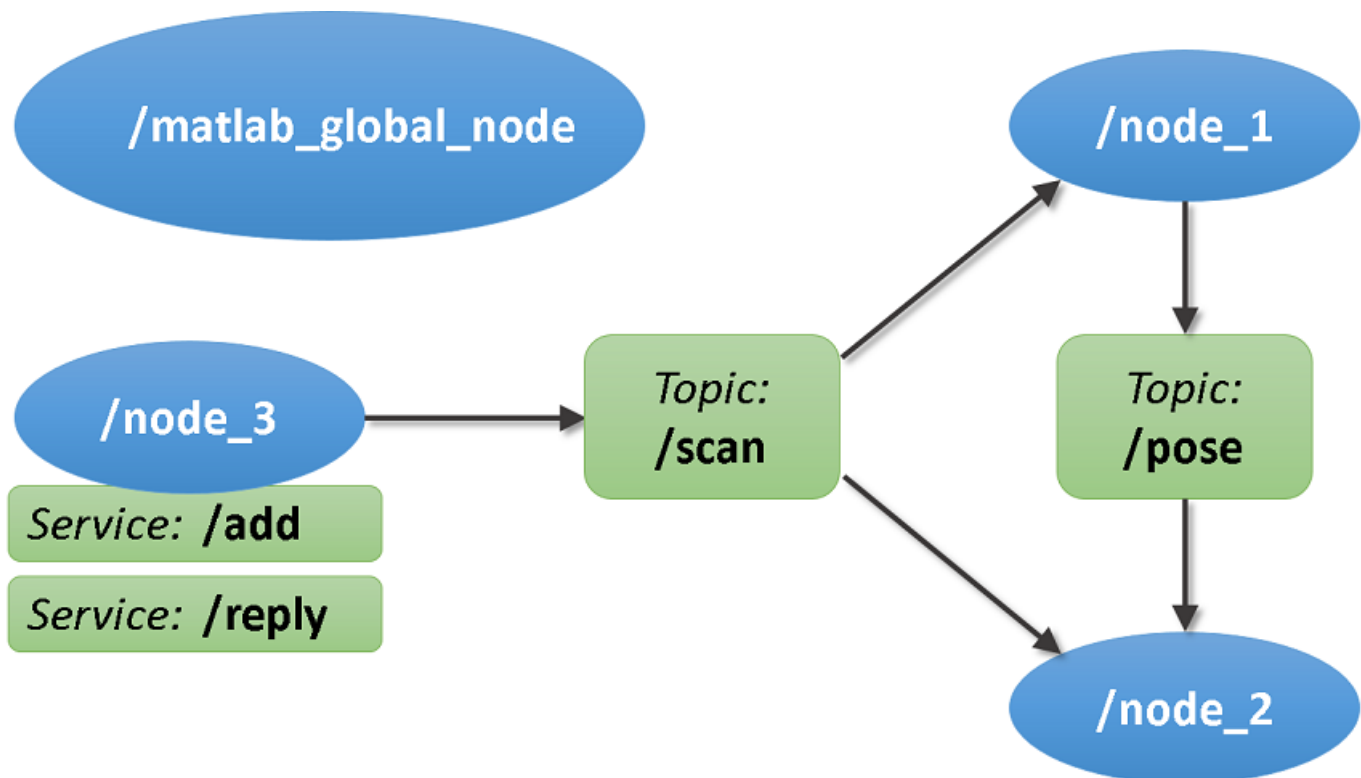
Topics

Use `rostopic list` to see available topics in the ROS network. There are four active topics: /pose, /rosout, /scan and /tf. The default topics: `rosout and tf` are always present in the ROS network. The other two topics were created as part of the sample network.

rostopic `list`

```
/pose
/rosout
/scan
/tf
```

Use `rostopic info <topicname>` to get specific information about a specific topic. The command below shows that **/node_1** publishes (sends messages to) the **/pose** topic, and **/node_2** subscribes (receives messages from) to that topic. See "Exchange Data with ROS Publishers and Subscribers" on page 1-20 for more information.

```
rostopic info /pose

Type: geometry_msgs/Twist

Publishers:
* /node_1 (http://ah-sradford:49883/)

Subscribers:
* /node_2 (http://ah-sradford:49889/)
```

Use `rosnode info <nodename>` to get information about a specific node. The command below shows that `node_1` publishes to `/pose, /rosout` and /tf topics, subscribes to the `/scan` topic and provides services: /node_1/get_loggers and /node_1/set_logger_level. The default logging services: get_loggers and set_logger_level are provided by all the nodes created in ROS network.

```
rosnode info /node_1

Node: [/node_1]
URI: [http://ah-sradford:49883/]

Publications (3 Active Topics):
 * /pose
 * /rosout
 * /tf

Subscriptions (1 Active Topics):
 * /scan

Services (2 Active):
 * /node_1/get_loggers
 * /node_1/set_logger_level
```

Services

ROS services provide a mechanism for procedure calls across the ROS network. A *service client* sends a request message to a *service server*, which processes the information in the request and returns with a response message (see "Call and Provide ROS Services" on page 1-26).

Use `rosservice list` to see all available service servers in the ROS network. The command below shows that two services (`/add` and `/reply`) are available along with the default logger services of all the nodes.

```
rosservice list

/add
/matlab_global_node_79626/get_loggers
/matlab_global_node_79626/set_logger_level
/node_1/get_loggers
/node_1/set_logger_level
/node_2/get_loggers
/node_2/set_logger_level
/node_3/get_loggers
/node_3/set_logger_level
/reply
```

Use `rosservice info <servicename>` to get information about a specific service.

```
rosservice info /add
```

```
Node: /node_3
URI: rosrpc://ah-sradford:49894
Type: roscpp_tutorials/TwoInts
Args: a b
```

Messages

Publishers, subscribers, and services use ROS messages to exchange information. Each ROS message has an associated *message type* that defines the datatypes and layout of information in that message (See "Work with Basic ROS Messages" on page 1-15).

Use `rostopic type <topicname>` to see the message type used by a topic. The command below shows that the `/pose` topic uses messages of type `geometry_msgs/Twist`.

```
rostopic type /pose
```

```
geometry_msgs/Twist
```

Use `rosmsg show <messagetype>` to view the properties of a message type. The `geometry_msgs/Twist` message type has two properties, `Linear` and `Angular`. Each property is a message of type `geometry_msgs/Vector3`, which in turn has three properties of type `double`.

```
rosmsg show geometry_msgs/Twist
```

```
% This expresses velocity in free space broken into its Linear and Angular parts.
Vector3  Linear
Vector3  Angular
```

```
rosmsg show geometry_msgs/Vector3
```

```
% This represents a vector in free space.
% It is only meant to represent a direction. Therefore, it does not
% make sense to apply a translation to it (e.g., when applying a
% generic rigid transformation to a Vector3, tf2 will only apply the
% rotation). If you want your data to be translatable too, use the
% geometry_msgs/Point message instead.

double X
double Y
double Z
```

Use `rosmsg list` to see the full list of message types available in MATLAB.

Shut Down ROS Network

Use `exampleHelperROSShutDownSampleNetwork` to remove the sample nodes, publishers, and subscribers from the ROS network. This command is only needed if the sample network was created earlier using `exampleHelperROSStartSampleNetwork`.

```
exampleHelperROSShutDownSampleNetwork
```

Use `rosshutdown` to shut down the ROS network in MATLAB. This shuts down the ROS master that was started by `rosinit` and deletes the global node. Using `rosshutdown` is the recommended procedure once you are done working with the ROS network.

```
rosshutdown
```

```
Shutting down global node /matlab_global_node_79626 with NodeURI http://ah-sradford:49876/
Shutting down ROS master on http://192.168.203.1:58232.
```

Next Steps

- "Connect to a ROS Network" on page 1-7

Connect to a ROS Network

A ROS network consists of a single *ROS master* and multiple *ROS nodes*. The ROS master facilitates the communication in the ROS network by keeping track of all active ROS entities. Every node needs to register with the ROS master to be able to communicate with the rest of the network. MATLAB® can start the ROS master, or the master can be launched outside of MATLAB (for example, on a different computer).

When you work with ROS, you typically follow these steps:

1 *Connect to a ROS network*. To connect to a ROS network, you can create the ROS master in MATLAB or connect to an existing ROS master. In both cases, MATLAB will also create and register its own ROS node (called the MATLAB *global node*) with the master. The `rosinit` function manages this process.

2 *Exchange data*. Once connected, MATLAB exchanges data with other ROS nodes through publishers, subscribers, and services.

3 *Disconnect from the ROS network*. Call the `rosshutdown` function to disconnect MATLAB from the ROS network.

This example shows you how to:

• Create a ROS master in MATLAB
• Connect to an external ROS master

Prerequisites: "Get Started with ROS" on page 1-2

Create a ROS Master in MATLAB

• To create the ROS master in MATLAB, call `rosinit` without any arguments. This function also creates the global node, which MATLAB uses to communicate with other nodes in the ROS network.

```
rosinit

Shutting down ROS master on http://172.30.68.186:52845.

Warning: Error shutting down the ROS master.

Launching ROS Core...
.............Done in 24.2735 seconds.
Initializing ROS master on http://172.30.68.186:55077.
Initializing global node /matlab_global_node_46764 with NodeURI http://bat1070102win64:58961/
```

ROS nodes that are external to MATLAB can now join the ROS network. They can connect to the ROS master in MATLAB by using the hostname or IP address of the MATLAB host computer.

You can shut down the ROS master and the global node by calling `rosshutdown`.

```
rosshutdown

Shutting down global node /matlab_global_node_46764 with NodeURI http://bat1070102win64:58961/
Shutting down ROS master on http://172.30.68.186:55077.
```

Connect to an External ROS Master

You can also use the `rosinit` command to connect to an external ROS master (for example running on a robot or a virtual machine). You can specify the address of the master in two ways: by an IP address or by hostname of the computer that runs the master.

After each call to `rosinit`, you have to call `rosshutdown` before calling `rosinit` with a different syntax. For brevity, these calls to `rosshutdown` are omitted in these examples.

`'master_host'` is an example host name and `'192.168.1.1'` is an example IP address of the external ROS master. Adjust these addresses depending on where the external master resides in your network. These commands will fail if no master is found at the specified addresses.

```
rosinit('192.168.1.1')
rosinit('master_host')
```

Both calls to `rosinit` assume that the master accepts network connections on port 11311, which is the standard ROS master port. If the master is running on a different port, you can specify it as a second argument. To connect to a ROS master running on host name `master_host` and port 12000, use the following command:

```
rosinit('master_host',12000)
```

If you know the entire Uniform Resource Identifier (URI) of the master, you can create the global node and connect to this master using this syntax:

```
rosinit('http://192.168.1.1:12000')
```

Node Host Specification

In some cases, your computer may be connected to multiple networks and have multiple IP addresses. This illustration shows an example.

Network 1 IP: 73.195.120.40

Network 1 IP: 73.195.120.50
Network 2 IP: 192.168.1.100

Network 2 IP: 192.168.1.1
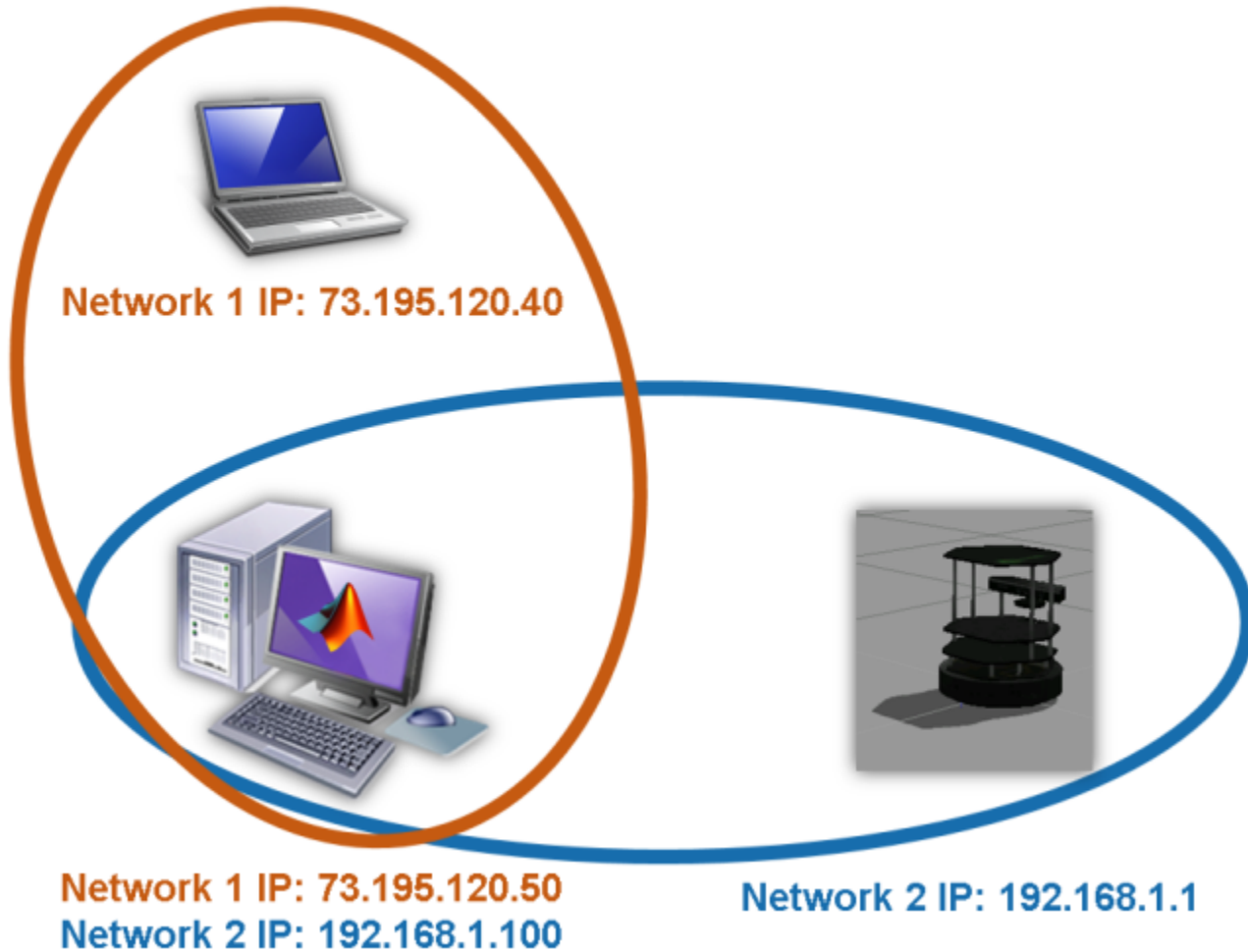
The computer on the bottom left runs MATLAB and is connected to two different networks. In one subnet, its IP address is `73.195.120.50,` and in the other, its IP is `192.168.1.100`. This computer wants to connect to the ROS master on the TurtleBot® computer at IP address `192.168.1.1`. As part of the registration with the master, the MATLAB global node has to specify the IP address or host name where other ROS nodes can reach it. All the nodes on the TurtleBot will use this address to send data to the global node in MATLAB.

When `rosinit` is invoked with the master's IP address, it tries to detect the network interface used to contact the master and use that as the IP address for the global node. If this automatic detection fails, you can explicitly specify the IP address or host name by using the `NodeHost` name-value pair in the `rosinit` call. The `NodeHost` name-value pair can be used with any of the other syntaxes already shown.

These commands advertise your computer's IP address to the ROS network as `192.168.1.100`.

```
rosinit('192.168.1.1','NodeHost','192.168.1.100')
rosinit('http://192.168.1.1:11311','NodeHost','192.168.1.100')
rosinit('master_host','NodeHost','192.168.1.100')
```

Once a node is registered in the ROS network, you can see the address that it advertises by using the command `rosnode info <nodename>`. You can see the names of all registered nodes by calling `rosnode list`.

ROS Environment Variables

In advanced use cases, you might want to specify the address of a ROS master and your advertised node address through standard ROS environment variables. The syntaxes that were explained in the previous sections should be sufficient for the majority of your use cases.

If no arguments are provided to `rosinit`, the function will also check the values of standard ROS environment variables. These variables are `ROS_MASTER_URI`, `ROS_HOSTNAME`, and `ROS_IP`. You can see their current values using the `getenv` command:

```
getenv('ROS_MASTER_URI')
getenv('ROS_HOSTNAME')
getenv('ROS_IP')
```

You can set these variables using the `setenv` command. After setting the environment variables, call `rosinit` with no arguments. The address of the ROS master is specified by `ROS_MASTER_URI,` and the global node's advertised address is given by `ROS_IP` or `ROS_HOSTNAME`. If you specify additional arguments to `rosinit`, they override the values in the environment variables.
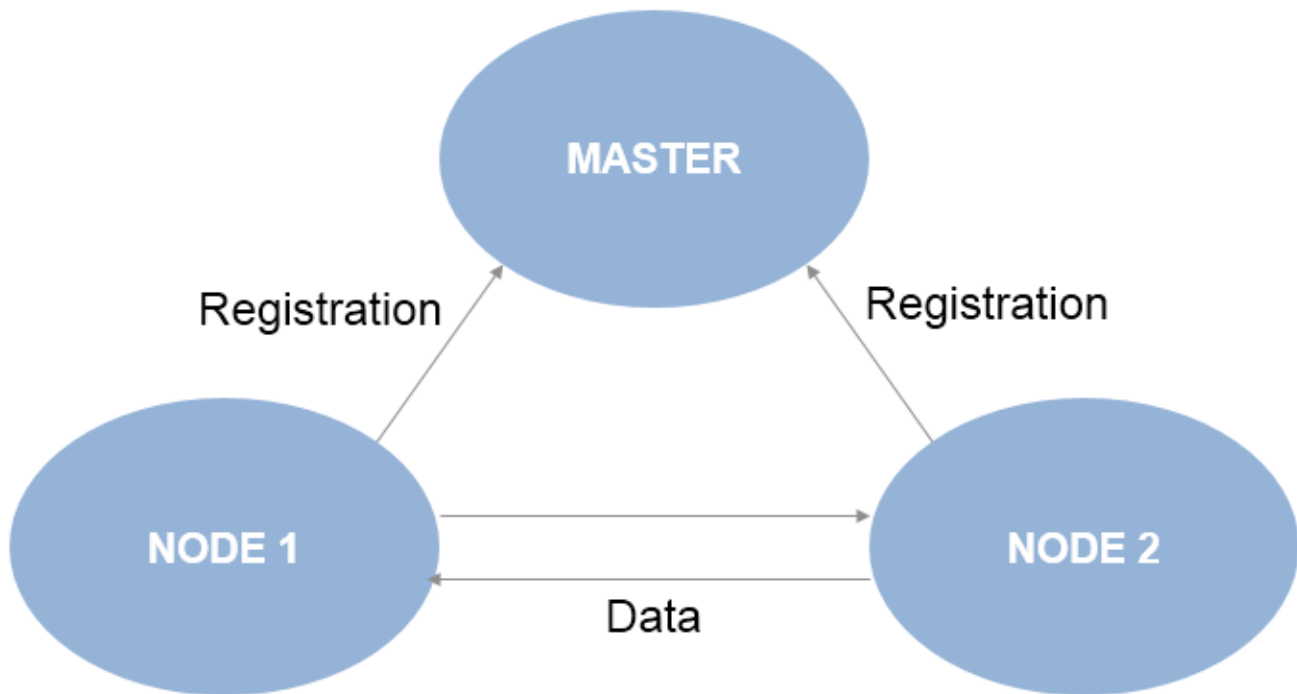
```
setenv('ROS_MASTER_URI','http://192.168.1.1:11311')
setenv('ROS_IP','192.168.1.100')
rosinit
```

You do not have to set both `ROS_HOSTNAME` and `ROS_IP`. If both are set, `ROS_HOSTNAME` takes precedence.

Verify Connection

For your ROS connection to work correctly, you must ensure that all nodes can communicate with the master and with each other. The individual nodes must communicate with the master to register subscribers, publishers, and services. They must also be able to communicate with one another to send and receive data. If your ROS network is not set up correctly, it is possible to be able to send data and be unable to receive data (or vice versa).

This diagram shows a ROS Network with a single ROS master and two different nodes that register themselves with the master. Each node contacts the master to find the advertised address of the other node in the ROS network. Once each node knows the other node's address, a data exchange can be established without involvement of the master.

Next Steps

- See "Exchange Data with ROS Publishers and Subscribers" on page 1-20 to explore publishers and subscribers in ROS.

Access the ROS Parameter Server

This example explores how to add and retrieve parameters on the ROS parameter server. The parameter server usually runs on the same device that launches the ROS master. The parameters are accessible globally over the ROS network and can be used to store static data such as configuration parameters. Supported data types include strings, integers, doubles, logicals, and cell arrays.

Prerequisites: "Get Started with ROS" on page 1-2, "Connect to a ROS Network" on page 1-7

Create Parameter Tree

Start the ROS master and parameter server in MATLAB.

```
rosinit

Launching ROS Core...
.........Done in 15.9501 seconds.
Initializing ROS master on http://172.30.68.186:59978.
Initializing global node /matlab_global_node_40279 with NodeURI http://bat1070102win64:50659/
```

Create a parameter tree object to interact with the parameter server. Use the parameter tree to interact with the parameter server and call functions such as `set`, `get`, `del`, `has` and `search`. Create a new parameter server using `rosparam`.

```
ptree = rosparam

ptree =
  ParameterTree with properties:

    AvailableParameters: {4x1 cell}
```

Add New Parameters

To set a parameter for the robot IP address, use the parameter name `ROBOT_IP`. Check if a parameter with the same name already exists. Use the `has` function.

```
has(ptree,'ROBOT_IP')

ans = logical
   0
```

If `has` returns `0` (false) as the output, then the `ROBOT_IP` name could not be found on the parameter server.

Add some parameters indicating a robot's IP address to the parameter server. Use the `set` function for this purpose.

```
set(ptree,'ROBOT_IP','192.168.1.1');
set(ptree,'/myrobot/ROBOT_IP','192.168.1.100');
```

The `ROBOT_IP` parameters are now available to all nodes connected to this ROS master. You can specify parameters within a namespace. For example, the `/myrobot/ROBOT_IP` parameter is within the `/myrobot` namespace in this example.

Set more parameters with different data types.

```
set(ptree,'MAX_SPEED',1.5);
```

Use a cell array as an input to the `set` function. Set a parameter that has the goal coordinates {x, y, z} for the robot.

```
set(ptree,'goal',{5.0,2.0,0.0});
```

Set additional parameters to populate the parameter server.

```
set(ptree,'/myrobot/ROBOT_NAME','TURTLE');
set(ptree,'/myrobot/MAX_SPEED',1.5);
set(ptree,'/newrobot/ROBOT_NAME','NEW_TURTLE');
```

Get Parameter Values

Retrieve the robot's IP address from the ROBOT_IP parameter in the `/myrobot` namespace using the `get` function:

```
robotIP = get(ptree,'/myrobot/ROBOT_IP')
```

```
robotIP =
'192.168.1.100'
```

Get List of All Parameters

To get the entire list of parameters stored on the parameter server, use dot notation to access the `AvailableParameters` property. The list contains all the parameters that you added in previous sections.

```
plist = ptree.AvailableParameters
```

```
plist = 11x1 cell
    {'/MAX_SPEED'                            }
    {'/ROBOT_IP'                             }
    {'/goal'                                 }
    {'/myrobot/MAX_SPEED'                    }
    {'/myrobot/ROBOT_IP'                     }
    {'/myrobot/ROBOT_NAME'                   }
    {'/newrobot/ROBOT_NAME'                  }
    {'/rosdistro'                            }
    {'/roslaunch/uris/host_172_30_68_186__50408'}
    {'/rosversion'                           }
    {'/run_id'                               }
```

Modify Existing Parameters

You can also use the `set` function to change parameter values. Note that the modification of a parameter is irreversible, since the parameter server will simply overwrite the parameter with the new value. You can verify if a parameter already exists by using the `has` function.

Modify the MAX_SPEED parameter:

```
set(ptree,'MAX_SPEED',1.0);
```

The modified value can have a different data type from a previously assigned value. For example, the value of the MAX_SPEED parameter is currently of type double. Set a string value for the MAX_SPEED parameter:

```
set(ptree,'MAX_SPEED','none');
```

Delete Parameters

Use the `del` function to delete a parameter from the parameter server.

Delete the `goal` parameter.

```
del(ptree,'goal');
```

Check if the `goal` parameter has been deleted. Use the `has` function.

```
has(ptree,'goal')

ans = logical
   0
```

The output is `0` (false), which means the parameter was deleted from the parameter server.

Search Parameters

Search for all the parameters that contain `'myrobot'` using the `search` command:

```
results = search(ptree,'myrobot')

results = 1x3 cell
    {'/myrobot/MAX_SPEED'}    {'/myrobot/ROBOT_IP'}    {'/myrobot/ROBOT_...'}
```

Shut Down the ROS Network

Shut down the ROS master and delete the global node.

```
rosshutdown
```

```
Shutting down global node /matlab_global_node_40279 with NodeURI http://bat1070102win64:50659/
Shutting down ROS master on http://172.30.68.186:59978.
```

Next Steps

- For application examples, see the "Get Started with Gazebo and a Simulated TurtleBot" on page 1-102 or "Get Started with a Real TurtleBot" on page 1-56 examples.

Work with Basic ROS Messages

Messages are the primary container for exchanging data in ROS. Topics and services use messages to carry data between nodes. (See "Exchange Data with ROS Publishers and Subscribers" on page 1-20 and "Call and Provide ROS Services" on page 1-26 for more information on topics and services)

To identify its data structure, each message has a *message type*. For example, sensor data from a laser scanner is typically sent in a message of type `sensor_msgs/LaserScan`. Each message type identifies the data elements that are contained in a message. Every message type name is a combination of a package name, followed by a forward slash /, and a type name:
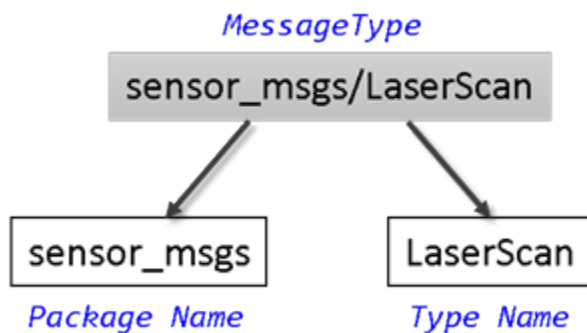
MATLAB® supports many ROS message types that are commonly encountered in robotics applications. This example shows some of the ways to create, explore, and populate ROS messages in MATLAB.

Prerequisites: "Get Started with ROS" on page 1-2, "Connect to a ROS Network" on page 1-7

Find Message Types

Initialize the ROS master and global node.

```
rosinit
```

```
Launching ROS Core...
.........Done in 17.113 seconds.
```

Use `exampleHelperROSCreateSampleNetwork` to populate the ROS network with three additional nodes and sample publishers and subscribers.

```
exampleHelperROSCreateSampleNetwork
```

There are various nodes on the network with a few topics and affiliated publishers and subscribers.

You can see the full list of available topics by calling `rostopic list`.

```
rostopic list
```

If you want to know more about the type of data that is sent through the `/scan` topic, use the `rostopic info` command to examine it. `/scan` has a message type of `sensor_msgs/LaserScan`.

```
rostopic info /scan
```

The command output also tells you which nodes are publishing and subscribing to the topic. To learn about publishers and subscribers, see "Call and Provide ROS Services" on page 1-26.

To find out more about the topic's message type, create an empty message of the same type using the `rosmessage` function. `rosmessage` supports tab completion for the message type. To complete message type names, type the first few characters of the name you want to complete, and then press the **Tab** key.

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

The created message `scandata` has many properties associated with data typically received from a laser scanner. For example, the minimum sensing distance is stored in the `RangeMin` property, and the maximum sensing distance is in `RangeMax`.

To see a complete list of all message types available for topics and services, use `rosmsg list`.

Explore Message Structure and Get Message Data

ROS messages are objects, and the message data is stored in properties. MATLAB features convenient ways to find and explore the contents of messages.

- If you subscribe to the `/pose` topic, you can receive and examine the messages that are sent.

```
posesub = rossubscriber('/pose')
```

Use `receive` to get data from the subscriber. Once a new message is received, the function will return it and store it in the `posedata` variable (the second argument is a time-out in seconds).

```
posedata = receive(posesub, 10)
```

The message has a type of `geometry_msgs/Twist`. There are two other properties in the message: `Linear` and `Angular`. You can see the values of these message properties by accessing them directly:

```
posedata.Linear
```

```
posedata.Angular
```

Each of the values of these message fields is actually a message in itself. The message type for these is `geometry_msgs/Vector3`. `geometry_msgs/Twist` is a composite message made up of two `geometry_msgs/Vector3` messages.

Data access for these nested messages works exactly the same as accessing the data in other messages. Access the X component of the `Linear` message using this command:

```
xpos = posedata.Linear.X
```

If you want a quick summary of all the data contained in a message, call the `showdetails` function. `showdetails` works on messages of any type and recursively displays all the message data properties.

```
showdetails(posedata)
```

`showdetails` helps you during debugging and when you want to quickly explore the contents of a message.

Set Message Data

You can also set message property values. Create a message with type `geometry_msgs/Twist`.

```
twist = rosmessage('geometry_msgs/Twist')
```

The numeric properties of this message are initialized to 0 by default. You can modify any of the properties of this message. Set the `Linear.Y` entry equal to 5.

```
twist.Linear.Y = 5;
```

View the message data to make sure that your change took effect.

```
twist.Linear
```

Once a message is populated with your data, you can use it with publishers, subscribers, and services. See the "Exchange Data with ROS Publishers and Subscribers" on page 1-20 and "Call and Provide ROS Services" on page 1-26 examples.

Copy Messages

There are two ways to copy the contents of a message:

- You can create a *reference copy* and the original messages share the same data.
- You can create a *deep copy*. The deep copy in which the copy and the original messages each have their own data.

A reference copy is useful if you want to share message data between different functions or objects, whereas a deep copy is necessary if you want an independent copy of a message.

Make a *reference copy* of a message by using the = sign. This creates a variable that references the same message contents as the original variable.

```
twistCopyRef = twist
```

Modify the `Linear.Z` field of `twistCopyRef`. This also changes the contents of `twist`.

```
twistCopyRef.Linear.Z = 7;
twist.Linear
```

Make a deep copy of `twist` so that you can change its contents without affecting the original data. Make a new message, `twistCopyDeep`, using the `copy` function:

```
twistCopyDeep = copy(twist)
```

Modify the `Linear.X` property of `twistCopyDeep`. The contents of `twist` remain unchanged.

```
twistCopyDeep.Linear.X = 100;
twistCopyDeep.Linear
twist.Linear
```

Save and Load Messages

You can save messages and store the contents for later use.

Get a new message from the subscriber.

```
posedata = receive(posesub,10)
```

Save the pose data to a MAT file using MATLAB's `save` function.

```
save('posedata.mat','posedata')
```

Before loading the file back into the workspace, clear the `posedata` variable.

```
clear posedata
```

Now you can load the message data by calling the `load` function. This loads the `posedata` from above into the `messageData` structure. `posedata` is a data field of the struct.

```
messageData = load('posedata.mat')
```

Examine `messageData.posedata` to see the message contents.

```
messageData.posedata
```

You can now delete the MAT file.

```
delete('posedata.mat')
```

Object Arrays in Messages

Some messages from ROS are stored in "Object Arrays". These must be handled differently from typical data arrays.

In your workspace, the variable `tf` contains a sample message. (The `exampleHelperROSCreateSampleNetwork` script created the variable.) In this case, it is a message of type `tf/tfMessage` used for coordinate transformations.

```
tf
```

`tf` has two fields: `MessageType` contains a standard data array, and `Transforms` contains an object array. There are 53 objects stored in `Transforms`, and all of them have the same structure.

Expand `tf` in `Transforms` to see the structure:

```
tf.Transforms
```

Each object in `Transforms` has four properties. You can expand to see the `Transform` field of `Transforms`.

```
tformFields = tf.Transforms.Transform
```

Note: The command output returns 53 individual answers, since each object is evaluated and returns the value of its `Transform` field. This format is not always useful, so you can convert it to a cell array with the following command:

```
cellTransforms = {tf.Transforms.Transform}
```

This puts all 53 object entries in a cell array, enabling you to access them with indexing.

In addition, you can access object array elements the same way you access standard MATLAB vectors:

```
tf.Transforms(5)
```

Access the translation component of the fifth transform in the list of 53:

```
tf.Transforms(5).Transform.Translation
```

Shut Down ROS Network

Remove the sample nodes, publishers, and subscribers from the ROS network.

`exampleHelperROSShutDownSampleNetwork`

Shut down the ROS master and delete the global node.

`rosshutdown`

Next Steps

- See "Work with Specialized ROS Messages" on page 1-44 for examples of handling images, point clouds, and laser scan messages.
- For application examples, see the "Get Started with Gazebo and a Simulated TurtleBot" on page 1-102 or "Get Started with a Real TurtleBot" on page 1-56 examples.

Exchange Data with ROS Publishers and Subscribers

The primary mechanism for ROS nodes to exchange data is sending and receiving *messages*. Messages are transmitted on a *topic,* and each topic has a unique name in the ROS network. If a node wants to share information, it uses a *publisher* to send data to a topic. A node that wants to receive that information uses a *subscriber* to that same topic. Besides its unique name, each topic also has a *message type*, which determines the types of messages that are capable of being transmitted under that topic.

This publisher and subscriber communication has the following characteristics:

- Topics are used for many-to-many communication. Many publishers can send messages to the same topic and many subscribers can receive them.
- Publishers and subscribers are decoupled through topics and can be created and destroyed in any order. A message can be published to a topic even if there are no active subscribers.

The concept of topics, publishers, and subscribers is illustrated in the figure:
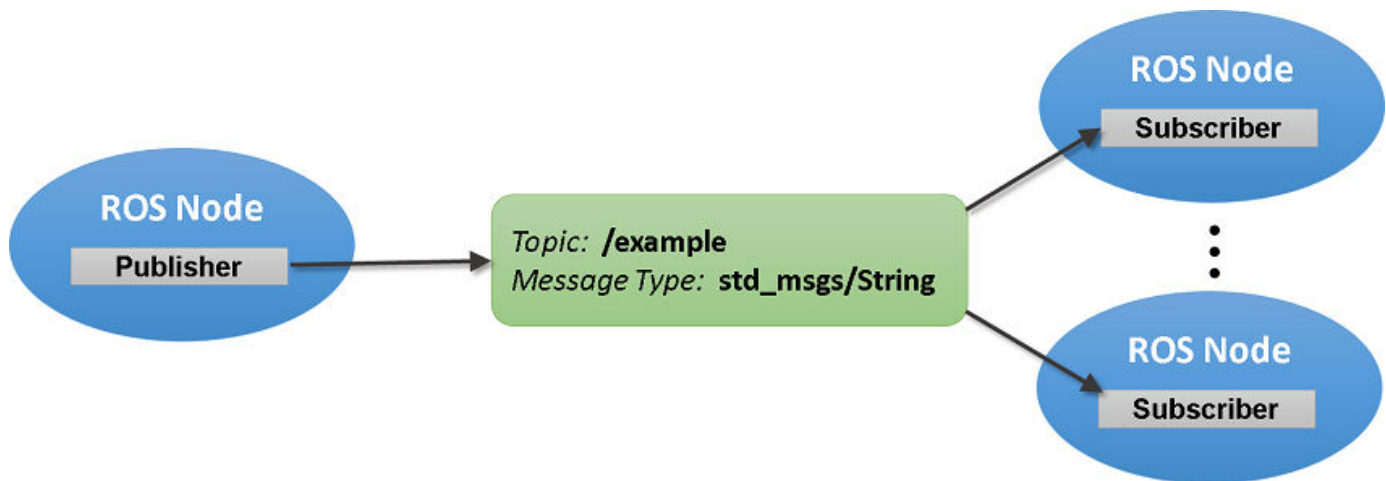
This example shows how to publish and subscribe to topics in a ROS network. It also shows how to:

- Wait until a new message is received
- Use callbacks to process new messages in the background

Prerequisites: "Get Started with ROS" on page 1-2, "Connect to a ROS Network" on page 1-7

Subscribe and Wait for Messages

Start the ROS master in MATLAB® using the `rosinit` command.

```
rosinit
```

```
Shutting down ROS master on http://172.30.68.186:54421.

Warning: Error shutting down the ROS master.

Launching ROS Core...
.......Done in 12.0108 seconds.
Initializing ROS master on http://172.30.68.186:53365.
Initializing global node /matlab_global_node_60652 with NodeURI http://bat1070102win64:59640/
```

Create a sample ROS network with several publishers and subscribers using the provided helper function `exampleHelperROSCreateSampleNetwork.`

```
exampleHelperROSCreateSampleNetwork
```

Use `rostopic list` to see which topics are available.

```
rostopic list
```

```
/pose
/rosout
/scan
/tf
```

Use `rostopic info` to check if any nodes are publishing to the `/scan` topic. The command below shows that `node_3` is publishing to it.

```
rostopic info /scan
```

```
Type: sensor_msgs/LaserScan

Publishers:
* /node_3 (http://bat1070102win64:59668/)

Subscribers:
* /node_1 (http://bat1070102win64:59650/)
* /node_2 (http://bat1070102win64:59656/)
```

Use `rossubscriber` to subscribe to the `/scan` topic. If the topic already exists in the ROS network (as is the case here), `rossubscriber` detects its message type automatically, so you do not need to specify it.

```
laser = rossubscriber('/scan');
pause(2)
```

Use `receive` to wait for a new message. (The second argument is a time-out in seconds.) The output `scandata` contains the received message data.

```
scandata = receive(laser,10)
```

```
scandata =
  ROS LaserScan message with properties:

       MessageType: 'sensor_msgs/LaserScan'
            Header: [1x1 Header]
          AngleMin: -0.5216
          AngleMax: 0.5243
    AngleIncrement: 0.0016
     TimeIncrement: 0
          ScanTime: 0.0330
          RangeMin: 0.4500
          RangeMax: 10
            Ranges: [640x1 single]
       Intensities: [0x1 single]

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

Some message types have visualizers associated with them. For the LaserScan message, `plot` plots the scan data. The `MaximumRange` name-value pair specifies the maximum plot range.

```
figure
plot(scandata,'MaximumRange',7)
```
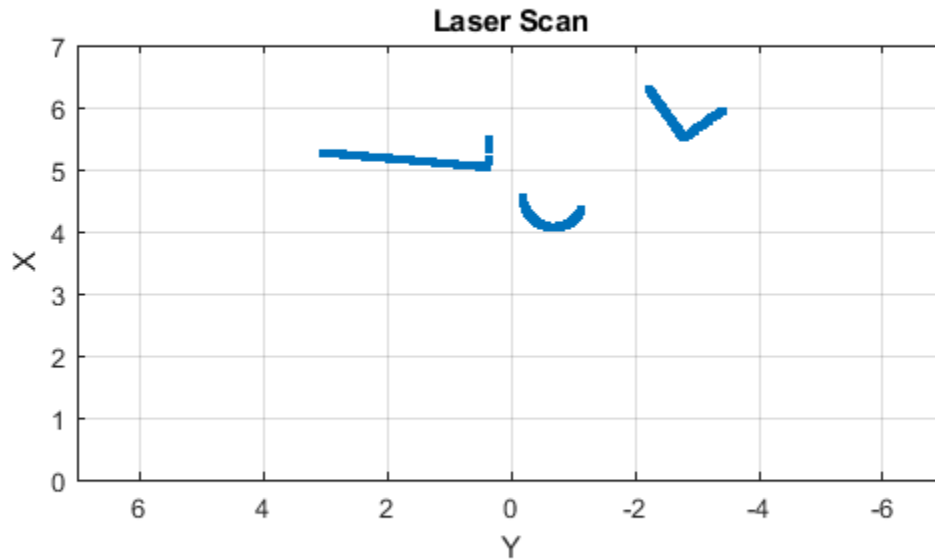

Subscribe Using Callback Functions

Instead of using `receive` to get data, you can specify a function to be called when a new message is received. This allows other MATLAB code to execute while the subscriber is waiting for new messages. Callbacks are essential if you want to use multiple subscribers.

Subscribe to the `/pose` topic, using the callback function `exampleHelperROSPoseCallback`.

```
robotpose = rossubscriber('/pose',@exampleHelperROSPoseCallback)

robotpose =
  Subscriber with properties:

        TopicName: '/pose'
    LatestMessage: [0x1 Twist]
      MessageType: 'geometry_msgs/Twist'
       BufferSize: 1
    NewMessageFcn: @exampleHelperROSPoseCallback
```

One way of sharing data between your main workspace and the callback function is to use global variables. Define two global variables `pos` and `orient`.

```
global pos
global orient
```

The global variables `pos` and `orient` are assigned in the `exampleHelperROSPoseCallback` function when new message data is received on the `/pose` topic.

Wait for a few seconds to make sure that the subscriber can receive messages. The most current position and orientation data will always be stored in the `pos` and `orient` variables.

```
pause(2)
pos
```

```
pos = 1×3

   -0.1612   -0.2381    0.0105
```

```
orient
```

```
orient = 1×3

   -0.2323    0.2905    0.1572
```

If you type in `pos` and `orient` a few times in the command line, you can see that the values are continuously updated.

Stop the pose subscriber by clearing the subscriber variable

```
clear robotpose
```

Note: There are other ways to extract information from callback functions besides using globals. For example, you can pass a handle object as additional argument to the callback function. See the "Callback Definition" documentation for more information about defining callback functions.

Publish Messages

Create a publisher that sends ROS string messages to the `/chatter` topic (see "Work with Basic ROS Messages" on page 1-15).

```
chatterpub = rospublisher('/chatter', 'std_msgs/String')
```

```
chatterpub =
  Publisher with properties:

          TopicName: '/chatter'
    NumSubscribers: 0
        IsLatching: 1
       MessageType: 'std_msgs/String'
```

```
pause(2) % Wait to ensure publisher is registered
```

Create and populate a ROS message to send to the `/chatter` topic.

```
chattermsg = rosmessage(chatterpub);
chattermsg.Data = 'hello world'
```

```
chattermsg =
  ROS String message with properties:
```

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

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

Use `rostopic list` to verify that the `/chatter` topic is available in the ROS network.

```
rostopic list

/chatter
/pose
/rosout
/scan
/tf
```

Define a subscriber for the `/chatter` topic. `exampleHelperROSChatterCallback` is called when a new message is received and displays the string content in the message.

```
chattersub = rossubscriber('/chatter', @exampleHelperROSChatterCallback)

chattersub =
  Subscriber with properties:

        TopicName: '/chatter'
    LatestMessage: [0x1 String]
      MessageType: 'std_msgs/String'
       BufferSize: 1
     NewMessageFcn: @exampleHelperROSChatterCallback
```

Publish a message to the `/chatter` topic. The string is displayed by the subscriber callback.

```
send(chatterpub,chattermsg)
pause(2)

ans =
'hello world'
```

The `exampleHelperROSChatterCallback` function was called as soon as you published the string message.

Shut Down ROS Network

Remove the sample nodes, publishers, and subscribers from the ROS network. Clear the global variables `pos` and `orient`.

```
exampleHelperROSShutDownSampleNetwork
clear global pos orient
```

Shut down the ROS master and delete the global node.

```
rosshutdown

Shutting down global node /matlab_global_node_60652 with NodeURI http://bat1070102win64:59640/
Shutting down ROS master on http://172.30.68.186:53365.
```

Next Steps

- To learn more about how ROS messages are handled in MATLAB, see "Work with Basic ROS Messages" on page 1-15 and "Work with Specialized ROS Messages" on page 1-44.
- To explore ROS services, refer to "Call and Provide ROS Services" on page 1-26.

Call and Provide ROS Services

ROS supports two main communication mechanisms: topics and services. Topics have publishers and subscribers and are used for sending and receiving messages (see "Exchange Data with ROS Publishers and Subscribers" on page 1-20). *Services*, on the other hand, implement a tighter coupling by allowing request-response communication. A *service client* sends a request message to a *service server* and waits for a response. The server will use the data in the request to construct a response message and sends it back to the client. Each service has a type that determines the structure of the request and response messages. Services also have a name that is unique in the ROS network.

This service communication has the following characteristics:

- A service request (or service call) is used for one-to-one communication. A single node will initiate the request and only one node will receive the request and send back a response.

- A service client and a service server are tightly coupled when a service call is executed. The server needs to exist at the time of the service call and once the request is sent, the client will block until a response is received.

The concept of services is illustrated in the following image:
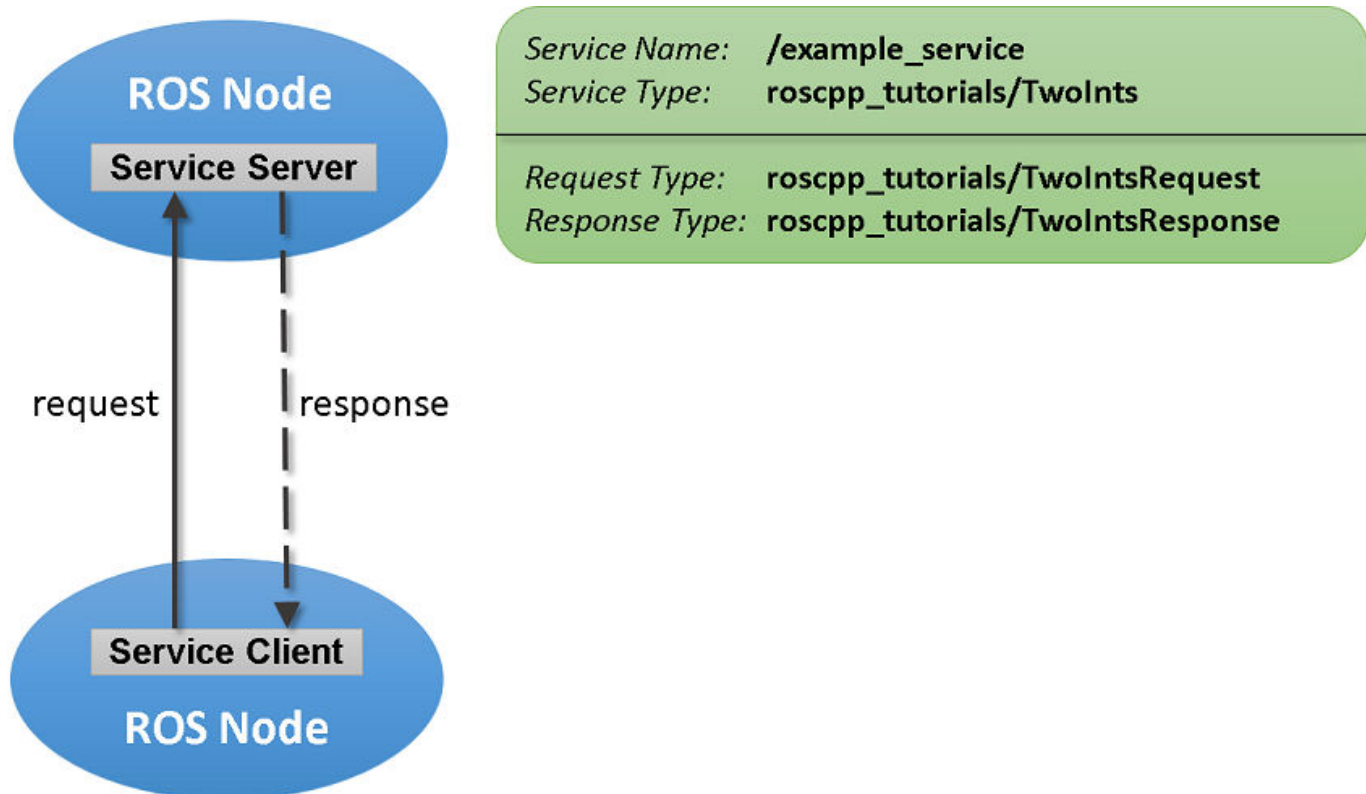
This example shows you how to set up service servers to advertise a service to the ROS network. In addition, you will learn how to use service clients to call the server and receive a response.

Prerequisites: "Get Started with ROS" on page 1-2, "Connect to a ROS Network" on page 1-7, "Exchange Data with ROS Publishers and Subscribers" on page 1-20

Create Service Server

Before examining service concepts, start the ROS master in MATLAB® and the sample ROS network. `exampleHelperROSCreateSampleNetwork` will create some service servers to simulate a realistic ROS network.

```
rosinit
```

```
Initializing ROS master on http://HYD-GDAVULUR:53363/.
Initializing global node /matlab_global_node_86569 with NodeURI http://HYD-GDAVULUR:53367/
```

```
exampleHelperROSCreateSampleNetwork
```

Suppose you want to make a simple service server that displays "*A service client is calling*" when you call the service. Create the service using the `rossvcserver` command. Specify the service name and the service message type. Also define the callback function as `exampleHelperROSEmptyCallback`. Callback functions for service servers have a very specific signature. For details, see the documentation of `rossvcserver`.

```
testserver = rossvcserver('/test', 'std_srvs/Empty', @exampleHelperROSEmptyCallback)
```

```
testserver =
  ServiceServer with properties:

      ServiceName: '/test'
      ServiceType: 'std_srvs/Empty'
    NewRequestFcn: @exampleHelperROSEmptyCallback
```

You can see your new service, `/test`, when you list all services in the ROS network.

```
rosservice list
```

```
/add
/reply
/test
```

You can get more information about your service using `rosservice info`. The global node is listed as node where the service server is reachable and you also see its `std_srvs/Empty` service type.

```
rosservice info /test
```

```
Node: /matlab_global_node_86569
URI: rosrpc://HYD-GDAVULUR:53368/
Type: std_srvs/Empty
Args:
```

Create Service Client

Use service clients to request information from a ROS service server. To create a client, use `rossvcclient` with the service name as the argument.

Create a service client for the `/test` service that we just created.

```
testclient = rossvcclient('/test')
```

```
testclient =
  ServiceClient with properties:
```

```
        ServiceName: '/test'
        ServiceType: 'std_srvs/Empty'
```

Create an empty request message for the service. Use the `rosmessage` function and pass the client as the first argument. This will create a service request function that has the message type that is specified by the service.

```
testreq = rosmessage(testclient)

testreq =
  ROS EmptyRequest message with properties:

    MessageType: 'std_srvs/EmptyRequest'

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

When you want to get a response from the server, use the `call` function, which calls the service server and returns a response. The service server you created before will return an empty response. In addition, it will call the `exampleHelperROSEmptyCallback` function and displays the string "*A service client is calling*". You can also define a `Timeout` parameter, which indicates how long the client should wait for a response.

```
testresp = call(testclient,testreq,'Timeout',3);
```

Create a Service for Adding Two Numbers

Up to now, the service server has not done any meaningful work, but you can use services for computations and data manipulation. Create a service that adds two integers.

There is an existing service type, `roscpp_tutorials/TwoInts`, that we can use for this task. You can inspect the structure of the request and response messages by calling `rosmsg show`. The request contains two integers, A and B, and the response contains their addition in Sum.

```
rosmsg show roscpp_tutorials/TwoIntsRequest

int64 A
int64 B


rosmsg show roscpp_tutorials/TwoIntsResponse

int64 Sum
```

Create the service server with this message type and a callback function that calculates the addition. For your convenience, the `exampleHelperROSSumCallback` function already implements this calculation. Specify the function as a callback.

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

sumserver =
  ServiceServer with properties:

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

To call the service server, you have to create a service client. Note that this client can be created anywhere in the ROS network. For the purposes of this example, we will create a client for the /sum service in MATLAB.

```
sumclient = rossvcclient('/sum')

sumclient =
  ServiceClient with properties:

    ServiceName: '/sum'
    ServiceType: 'roscpp_tutorials/TwoInts'
```

Create the request message. You can define the two integers, A and B, which are added together when you use the call command.

```
sumreq = rosmessage(sumclient);
sumreq.A = 2;
sumreq.B = 1

sumreq =
  ROS TwoIntsRequest message with properties:

    MessageType: 'roscpp_tutorials/TwoIntsRequest'
              A: 2
              B: 1

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

The expectation is that the sum of these two numbers will be 3. To call the service, use the following command. The service response message will contain a Sum property, which stores the addition of A and B.

```
sumresp = call(sumclient,sumreq,'Timeout',3)

sumresp =
  ROS TwoIntsResponse message with properties:

    MessageType: 'roscpp_tutorials/TwoIntsResponse'
            Sum: 3

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

Shut Down ROS Network

Remove the sample nodes and service servers from the ROS network.

```
exampleHelperROSShutDownSampleNetwork
```

Shut down the ROS master and delete the global node.

```
rosshutdown

Shutting down global node /matlab_global_node_86569 with NodeURI http://HYD-GDAVULUR:53367/
Shutting down ROS master on http://HYD-GDAVULUR:53363/.
```

Next Steps

- Refer to "Work with Basic ROS Messages" on page 1-15 to explore how ROS messages are represented in MATLAB.

Work with rosbag Logfiles

A *rosbag* or bag is a file format in ROS for storing 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. See "ROS Log Files (rosbags)" on page 3-13 for more information about rosbag support in MATLAB.

In this example, you will load a rosbag and learn how to select and retrieve the contained messages.

Prerequisites: "Work with Basic ROS Messages" on page 1-15

Load a rosbag

Load an example file using the `rosbag` command.

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

bag =
  BagSelection with properties:

            FilePath: 'C:\TEMP\Bdoc20b_1465442_5924\ib8F4264\9\tpa047c0b2\ros-ex71482057\ex_multip
           StartTime: 201.3400
             EndTime: 321.3400
         NumMessages: 36963
      AvailableTopics: [4x3 table]
      AvailableFrames: {0x1 cell}
         MessageList: [36963x4 table]
```

The object returned from the `rosbag` call is a `BagSelection` object, which is a representation of all the messages in the rosbag.

The object display shows details about how many messages are contained in the file (`NumMessages`) and the time when the first (`StartTime`) and the last (`EndTime`) message were recorded.

Evaluate the `AvailableTopics` property to see more information about the topics and message types that are recorded in the bag:

```
bag.AvailableTopics
```

```
ans=4×3 table
                         NumMessages           MessageType                    MessageDefinition
                         _____    _____    _____

    /clock                  12001       rosgraph_msgs/Clock        {0x0 char
    /gazebo/link_states     11999       gazebo_msgs/LinkStates     {'geometry_msgs/Pose[] Pose.
    /odom                   11998       nav_msgs/Odometry          {'  uint32 Seq...'
    /scan                     965       sensor_msgs/LaserScan      {'  uint32 Seq...'
```

The `AvailableTopics` table contains the sorted list of topics that are included in the rosbag. The table stores the number of messages, the message type, and the message definition for the topic. For more information on the MATLAB table data type and what operations you can perform on it, see the documentation for "Tables".

Initially the rosbag is only indexed by MATLAB and no actual message data is read.

You might want to filter and narrow the selection of messages as much as possible based on this index before any messages are loaded into MATLAB memory.

Select Messages

Before you retrieve any message data, you must select a set of messages based on criteria such as time stamp, topic name, and message type.

You can examine all the messages in the current selection:

```
bag.MessageList
```

ans=*36963×4 table*

Time	Topic	MessageType	FileOffset
201.34	/gazebo/link_states	gazebo_msgs/LinkStates	9866
201.34	/odom	nav_msgs/Odometry	7666
201.34	/clock	rosgraph_msgs/Clock	4524
201.35	/clock	rosgraph_msgs/Clock	10962
201.35	/clock	rosgraph_msgs/Clock	12876
201.35	/odom	nav_msgs/Odometry	12112
201.35	/gazebo/link_states	gazebo_msgs/LinkStates	11016
201.36	/gazebo/link_states	gazebo_msgs/LinkStates	12930
201.36	/odom	nav_msgs/Odometry	14026
201.37	/odom	nav_msgs/Odometry	14844
201.37	/gazebo/link_states	gazebo_msgs/LinkStates	15608
201.37	/clock	rosgraph_msgs/Clock	14790
201.38	/clock	rosgraph_msgs/Clock	16704
201.38	/gazebo/link_states	gazebo_msgs/LinkStates	16758
201.38	/odom	nav_msgs/Odometry	17854
201.39	/gazebo/link_states	gazebo_msgs/LinkStates	18672
⋮			

The `MessageList` table contains one row for each message in the bag (there are over 30,000 rows for the bag in this example). The rows are sorted by time stamp in the first column, which represents the time (in seconds) that the message was recorded.

Since the list is very large, you can also display a selection of rows with the familiar row and column selection syntax:

```
bag.MessageList(500:505,:)
```

ans=*6×4 table*

Time	Topic	MessageType	FileOffset
203	/clock	rosgraph_msgs/Clock	339384
203	/gazebo/link_states	gazebo_msgs/LinkStates	331944
203	/gazebo/link_states	gazebo_msgs/LinkStates	333040
203	/gazebo/link_states	gazebo_msgs/LinkStates	334136
203	/gazebo/link_states	gazebo_msgs/LinkStates	335232
203	/odom	nav_msgs/Odometry	336328

Use the `select` function to narrow the selection of messages. The `select` function operates on the `bag` object.

You can filter the message list by time, topic name, message type, or any combination of the three.

To select all messages that were published on the `/odom` topic, use the following `select` command:

```
bagselect1 = select(bag, 'Topic', '/odom')

bagselect1 =
  BagSelection with properties:

            FilePath: 'C:\TEMP\Bdoc20b_1465442_5924\ib8F4264\9\tpa047c0b2\ros-ex71482057\ex_multip
           StartTime: 201.3400
             EndTime: 321.3300
         NumMessages: 11998
      AvailableTopics: [1x3 table]
      AvailableFrames: {0x1 cell}
          MessageList: [11998x4 table]
```

Calls to the `select` function return another `BagSelection` object, which can be used to make further selections or retrieve message data. All selection objects are independent of each other, so you can clear them from the workspace once you are done.

You can make a different selection that combines two criteria. To get the list of messages that were recorded within the first 30 seconds of the rosbag and published on the `/odom` topic, enter the following command:

```
start = bag.StartTime

start = 201.3400

bagselect2 = select(bag, 'Time', [start start + 30], 'Topic', '/odom')

bagselect2 =
  BagSelection with properties:

            FilePath: 'C:\TEMP\Bdoc20b_1465442_5924\ib8F4264\9\tpa047c0b2\ros-ex71482057\ex_multip
           StartTime: 201.3400
             EndTime: 231.3200
         NumMessages: 2997
      AvailableTopics: [1x3 table]
      AvailableFrames: {0x1 cell}
          MessageList: [2997x4 table]
```

Use the last selection to narrow down the time window even further:

```
bagselect3 = select(bagselect2, 'Time', [205 206])

bagselect3 =
  BagSelection with properties:

            FilePath: 'C:\TEMP\Bdoc20b_1465442_5924\ib8F4264\9\tpa047c0b2\ros-ex71482057\ex_multip
           StartTime: 205.0200
             EndTime: 205.9900
         NumMessages: 101
      AvailableTopics: [1x3 table]
      AvailableFrames: {0x1 cell}
          MessageList: [101x4 table]
```

The selection in this last step operated on the existing `bagselect2` selection and returned a new `bagselect3` object.

If you want to save a set of selection options, store the selection elements in a cell array and then re-use it later as an input to the `select` function:

```
selectOptions = {'Time', [start, start+1; start+5, start+6], 'MessageType', {'sensor_msgs/LaserSc
bagselect4 = select(bag, selectOptions{:})

bagselect4 =
  BagSelection with properties:

           FilePath: 'C:\TEMP\Bdoc20b_1465442_5924\ib8F4264\9\tpa047c0b2\ros-ex71482057\ex_multip
          StartTime: 201.3400
            EndTime: 207.3300
        NumMessages: 209
    AvailableTopics: [2x3 table]
    AvailableFrames: {0x1 cell}
        MessageList: [209x4 table]
```

Read Selected Message Data

After you narrow your message selection, you might want to read the actual message data into MATLAB. Depending on the size of your selection, this can take a long time and consume a lot of your computer's memory.

To retrieve the messages in you selection as a cell array, use the `readMessages` function:

```
msgs = readMessages(bagselect3);
size(msgs)

ans = 1×2

   101     1
```

The resulting cell array contains as many elements as indicated in the `NumMessages` property of the selection object.

In reading message data, you can also be more selective and only retrieve messages at specific indices. Here is an example of retrieving 4 messages:

```
msgs = readMessages(bagselect3, [1 2 3 7])

msgs=4×1 cell array
    {1x1 Odometry}
    {1x1 Odometry}
    {1x1 Odometry}
    {1x1 Odometry}
```

```
msgs{2}

ans =
  ROS Odometry message with properties:

    MessageType: 'nav_msgs/Odometry'
```

```
        Header: [1x1 Header]
          Pose: [1x1 PoseWithCovariance]
         Twist: [1x1 TwistWithCovariance]
   ChildFrameId: 'base_footprint'

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

Each message in the cell array is a standard MATLAB ROS message object. For more information on messages, see the "Work with Basic ROS Messages" on page 1-15 example.

Extract Message Data as Time Series

Sometimes you are not interested in the complete messages, but only in specific properties that are common to all the messages in a selection. In this case, it is helpful to retrieve the message data as a time series instead. A time series is a data vector that is sampled over time and represents the time evolution of one or more dynamic properties. For more information on the MATLAB time series support, see the documentation for "Time Series".

In the case of ROS messages within a rosbag, a time series can help to express the change in particular message elements through time. You can extract this information through the `timeseries` function. This is memory-efficient, since the complete messages do not have to be stored in memory.

Use the same selection, but use the `timeseries` function to only extract the properties for x-position and z-axis angular velocity:

```
ts = timeseries(bagselect3, 'Pose.Pose.Position.X', 'Twist.Twist.Angular.Z')

  timeseries

  Timeseries contains duplicate times.

  Common Properties:
            Name: '/odom Properties'
            Time: [101x1 double]
        TimeInfo: tsdata.timemetadata
            Data: [101x2 double]
        DataInfo: tsdata.datametadata
```

The return of this call is a `timeseries` object that can be used for further analysis or processing.

Note that this method of extracting data is only supported if the current selection contains a single topic with a single message type.

To see the data contained within the time series, access the `Data` property:

```
ts.Data
```

ans = *101×2*

```
    0.0003    0.0003
    0.0003    0.0003
    0.0003   -0.0006
    0.0003   -0.0006
    0.0003   -0.0010
    0.0003   -0.0010
    0.0003   -0.0003
```

```
     0.0003    -0.0003
     0.0003    -0.0003
     0.0003    -0.0003
        ⋮
```

There are many other possible ways to work with the time series data. Calculate the mean of the data columns:

```
mean(ts)
```

```
ans = 1×2
10-3 ×

    0.3213    -0.4616
```

You can also plot the data of the time series:

```
figure
plot(ts, 'LineWidth', 3)
```
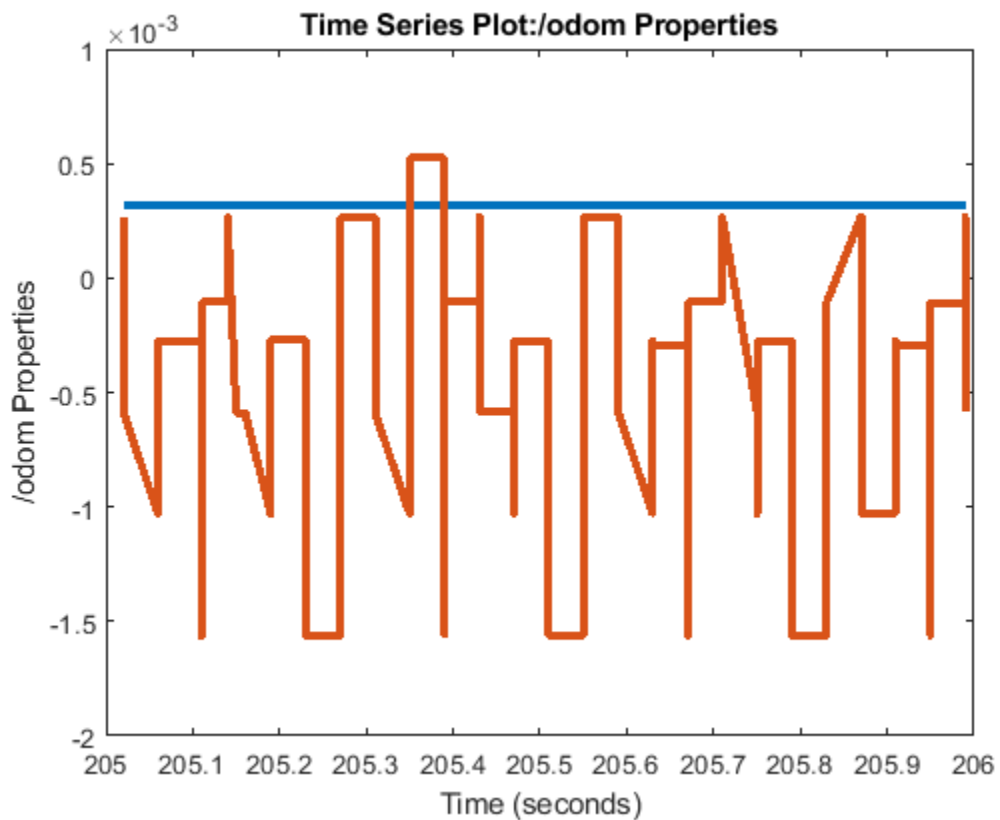
Access the tf Transformation Tree in ROS

The tf system in ROS keeps track of multiple coordinate frames and maintains the relationship between them in a tree structure. tf is distributed, so that all coordinate frame information is available to every node in the ROS network. MATLAB® enables you to access this transformation tree. This example familiarizes you with accessing the available coordinate frames, retrieving transformations between them, and transform points, vectors, and other entities between any two coordinate frames.

Prerequisites: "Get Started with ROS" on page 1-2, "Connect to a ROS Network" on page 1-7

Start up

Initialize the ROS system.

```
rosinit
```

```
Launching ROS Core...
........Done in 10.7193 seconds.
Initializing ROS master on http://172.30.68.186:49941.
Initializing global node /matlab_global_node_71987 with NodeURI http://bat1070102win64:51092/
```

To create a realistic environment for this example, use `exampleHelperROSStartTfPublisher` to broadcast several transformations. The transformations represent a camera that is mounted on a robot.

There are three coordinate frames that are defined in this transformation tree:

- the robot base frame (`robot_base`)
- the camera's mounting point (`mounting_point`)
- the optical center of the camera (`camera_center`)

Two transforms are being published:

- the transformation from the robot base to the camera's mounting point
- the transformation from the mounting point to the center of the camera

```
exampleHelperROSStartTfPublisher
```

A visual representation of the three coordinate frames looks as follows.
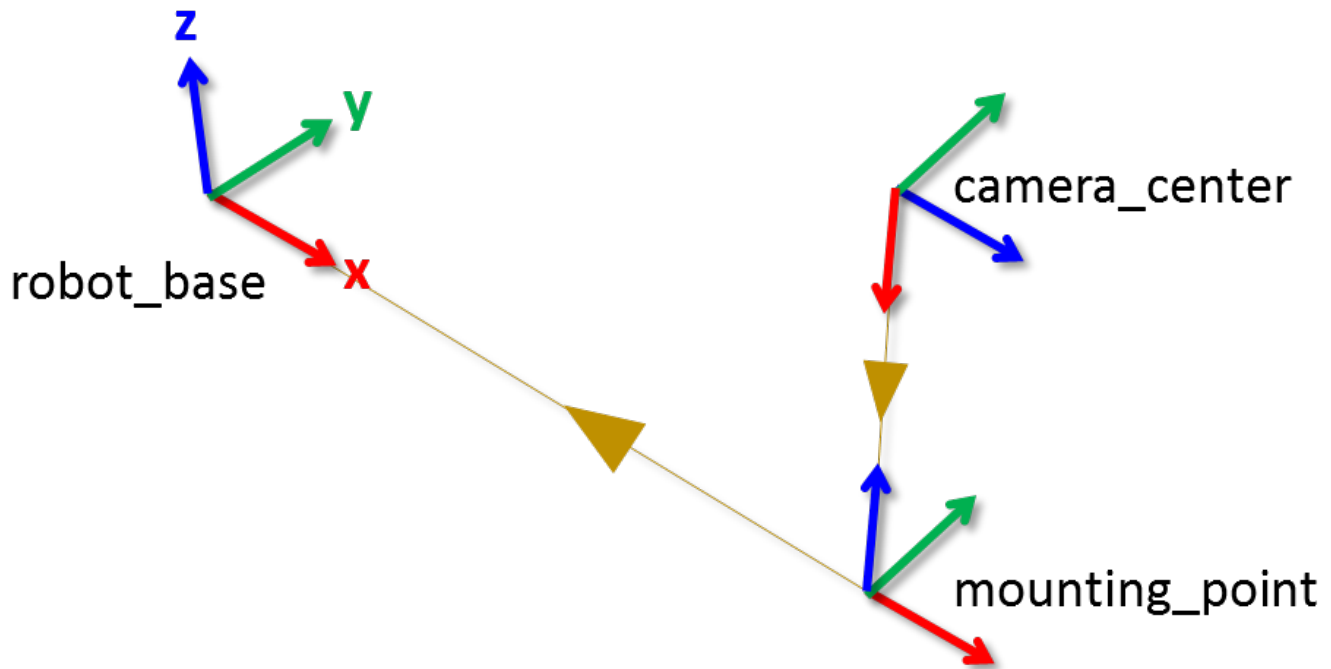
Here, the x, y, and z axes of each frame are represented by red, green, and blue lines respectively. The parent-child relationship between the coordinate frames is shown through a brown arrow pointing from the child to its parent frame.

Create a new transformation tree object with the `rostf` function. You can use this object to access all available transformations and apply them to different entities.

```
tftree = rostf
```

```
tftree =
  TransformationTree with properties:

    AvailableFrames: {0x1 cell}
     LastUpdateTime: [0x1 Time]
         BufferTime: 10
```

Once the object is created, it starts receiving tf transformations and buffers them internally. Keep the `tftree` variable in the workspace so that it continues to receive data.

Pause for a little bit to make sure that all transformations are received.

```
pause(2);
```

You can see the names of all the available coordinate frames by accessing the `AvailableFrames` property.

```
tftree.AvailableFrames
```

```
ans = 3x1 cell
    {'camera_center' }
```

```
{'mounting_point'}
{'robot_base'    }
```

This should show the three coordinate frames that describe the relationship between the camera, its mounting point, and the robot.

Receive Transformations

Now that the transformations are available, you can inspect them. Any transformation is described by a ROS `geometry_msgs/TransformStamped` message and has a translational and rotational component.

Retrieve the transformation that describes the relationship between the mounting point and the camera center. Use the `getTransform` function to do that.

```
mountToCamera = getTransform(tftree, 'mounting_point', 'camera_center');
mountToCameraTranslation = mountToCamera.Transform.Translation

mountToCameraTranslation =
  ROS Vector3 message with properties:

    MessageType: 'geometry_msgs/Vector3'
              X: 0
              Y: 0
              Z: 0.5000

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

```
quat = mountToCamera.Transform.Rotation

quat =
  ROS Quaternion message with properties:

    MessageType: 'geometry_msgs/Quaternion'
              X: 0
              Y: 0.7071
              Z: 0
              W: 0.7071

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

```
mountToCameraRotationAngles = rad2deg(quat2eul([quat.W quat.X quat.Y quat.Z]))

mountToCameraRotationAngles = 1×3

     0    90     0
```

Relative to the mounting point, the camera center is located 0.5 meters along the z-axis and is rotated by 90 degrees around the y-axis.

To inspect the relationship between the robot base and the camera's mounting point, call `getTransform` again.

```
baseToMount = getTransform(tftree, 'robot_base', 'mounting_point');
baseToMountTranslation = baseToMount.Transform.Translation
```

```
baseToMountTranslation =
  ROS Vector3 message with properties:

    MessageType: 'geometry_msgs/Vector3'
             X: 1
             Y: 0
             Z: 0

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

```
baseToMountRotation = baseToMount.Transform.Rotation
```

```
baseToMountRotation =
  ROS Quaternion message with properties:

    MessageType: 'geometry_msgs/Quaternion'
             X: 0
             Y: 0
             Z: 0
             W: 1

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

The mounting point is located at 1 meter along the robot base's x-axis.

Apply Transformations

Assume now that you have a 3D point that is defined in the `camera_center` coordinate frame and you want to calculate what the point coordinates in the `robot_base` frame are.

Use the `waitForTransform` function to wait until the transformation between the `camera_center` and `robot_base` coordinate frames becomes available. This call blocks until the transform that takes data from `camera_center` to `robot_base` is valid and available in the transformation tree.

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

Now define a point at position `[3 1.5 0.2]` in the camera center's coordinate frame. You will subsequently transform this point into `robot_base` coordinates.

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

You can transform the point coordinates by calling the `transform` function on the transformation tree object. Specify what the target coordinate frame of this transformation is. In this example, use `robot_base`.

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

```
tfpt =
  ROS PointStamped message with properties:

    MessageType: 'geometry_msgs/PointStamped'
         Header: [1x1 Header]
```

```
        Point: [1x1 Point]

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

The transformed point `tfpt` has the following 3D coordinates.

`tfpt.Point`

```
ans =
  ROS Point message with properties:

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

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

Besides `PointStamped` messages, the `transform` function allows you to transform other entities like poses (`geometry_msgs/PoseStamped`), quaternions (`geometry_msgs/QuaternionStamped`), and point clouds (`sensor_msgs/PointCloud2`).

If you want to store a transformation, you can retrieve it with the `getTransform` function.

`robotToCamera = getTransform(tftree, 'robot_base', 'camera_center')`

```
robotToCamera =
  ROS TransformStamped message with properties:

     MessageType: 'geometry_msgs/TransformStamped'
          Header: [1x1 Header]
       Transform: [1x1 Transform]
     ChildFrameId: 'camera_center'

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

This transformation can be used to transform coordinates in the `camera_center` frame into coordinates in the `robot_base` frame.

`robotToCamera.Transform.Translation`

```
ans =
  ROS Vector3 message with properties:

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

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

`robotToCamera.Transform.Rotation`

```
ans =
  ROS Quaternion message with properties:
```

```
    MessageType: 'geometry_msgs/Quaternion'
             X: 0
             Y: 0.7071
             Z: 0
             W: 0.7071
```

  Use showdetails to show the contents of the message

Send Transformations

You can also broadcast a new transformation to the ROS network.

Assume that you have a simple transformation that describes the offset of the `wheel` coordinate frame relative to the `robot_base` coordinate frame. The wheel is mounted -0.2 meters along the y-axis and -0.3 along the z-axis. The wheel has a relative rotation of 30 degrees around the y-axis.

Create the corresponding `geometry_msgs/TransformStamped` message that describes this transformation. The source coordinate frame, `wheel`, is placed to the `ChildFrameId` property. The target coordinate frame, `robot_base`, is added to the `Header.FrameId` property.

```
tfStampedMsg = rosmessage('geometry_msgs/TransformStamped');
tfStampedMsg.ChildFrameId = 'wheel';
tfStampedMsg.Header.FrameId = 'robot_base';
```

The transformation itself is described in the `Transform` property. It contains a `Translation` and a `Rotation`. Fill in the values that are listed above.

The `Rotation` is described as a quaternion. To convert the 30 degree rotation around the y-axis to a quaternion, you can use the `axang2quat` (Navigation Toolbox) function. The y-axis is described by the `[0 1 0]` vector and 30 degrees can be converted to radians with the `deg2rad` function.

```
tfStampedMsg.Transform.Translation.X = 0;
tfStampedMsg.Transform.Translation.Y = -0.2;
tfStampedMsg.Transform.Translation.Z = -0.3;

quatrot = axang2quat([0 1 0 deg2rad(30)])

quatrot = 1×4

    0.9659         0    0.2588         0


tfStampedMsg.Transform.Rotation.W = quatrot(1);
tfStampedMsg.Transform.Rotation.X = quatrot(2);
tfStampedMsg.Transform.Rotation.Y = quatrot(3);
tfStampedMsg.Transform.Rotation.Z = quatrot(4);
```

Use `rostime` to retrieve the current system time and use that to timestamp the transformation. This lets the recipients know at which point in time this transformation was valid.

```
tfStampedMsg.Header.Stamp = rostime('now');
```

Use the `sendTransform` function to broadcast this transformation.

```
sendTransform(tftree, tfStampedMsg)
```

The new `wheel` coordinate frame is now also in the list of available coordinate frames.

`tftree.AvailableFrames`

```
ans = 4x1 cell
    {'camera_center' }
    {'mounting_point'}
    {'robot_base'    }
    {'wheel'         }
```

The final visual representation of all four coordinate frames looks as follows.
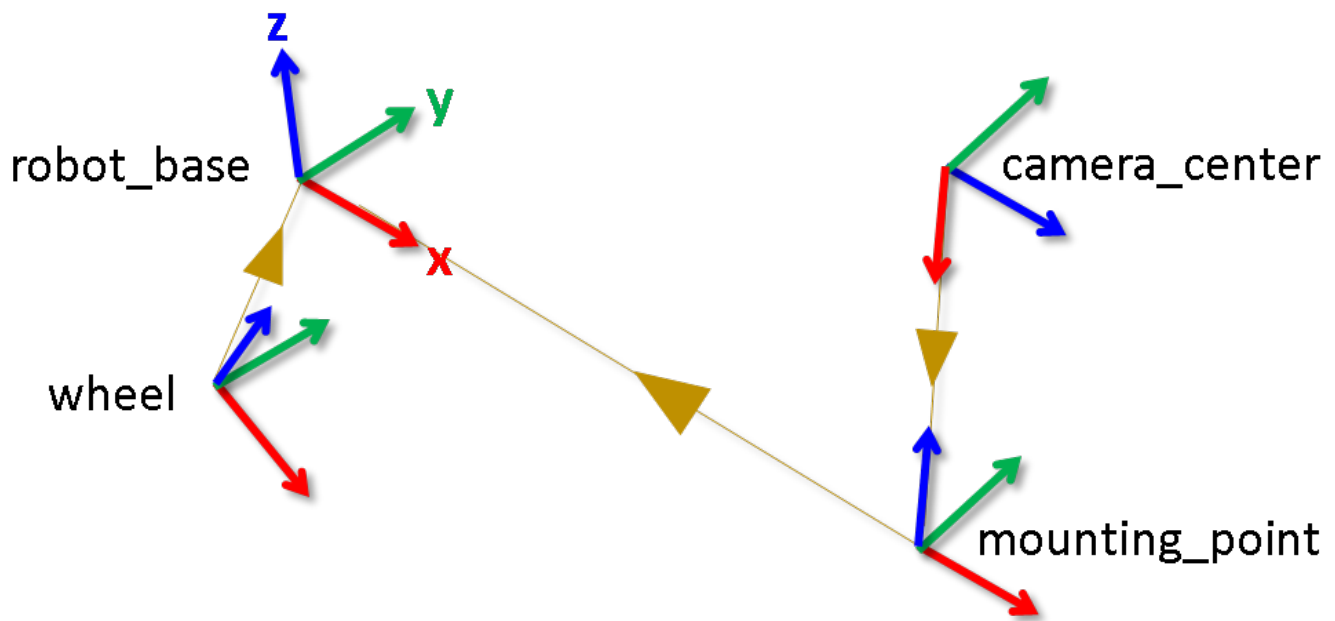
You can see that the `wheel` coordinate frame has the translation and rotation that you specified in sending the transformation.

Stop Example Publisher and Shut Down ROS Network

Stop the example transformation publisher.

`exampleHelperROSStopTfPublisher`

Shut down the ROS master and delete the global node.

`rosshutdown`

```
Shutting down global node /matlab_global_node_71987 with NodeURI http://bat1070102win64:51092/
Shutting down ROS master on http://172.30.68.186:49941.
```

Work with Specialized ROS Messages

Some commonly used ROS messages store data in a format that requires some transformation before it can be used for further processing. MATLAB® can help you by formatting these specialized ROS messages for easy use. In this example, you can explore how message types for laser scans, uncompressed and compressed images, and point clouds are handled.

Prerequisites: "Work with Basic ROS Messages" on page 1-15

Load Sample Messages

Load some sample messages. These messages are populated with data gathered from various robotics sensors.

```
exampleHelperROSLoadMessages
```

Laser Scan Messages

Laser scanners are commonly used sensors in robotics. You can see the standard ROS format for a laser scan message by creating an empty message of the appropriate type.

Use `rosmessage` to create the message.

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

emptyscan =
  ROS LaserScan message with properties:

       MessageType: 'sensor_msgs/LaserScan'
            Header: [1x1 Header]
          AngleMin: 0
          AngleMax: 0
    AngleIncrement: 0
     TimeIncrement: 0
          ScanTime: 0
          RangeMin: 0
          RangeMax: 0
            Ranges: [0x1 single]
        Intensities: [0x1 single]

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

Since you created an empty message, `emptyscan` does not contain any meaningful data. For convenience, the `exampleHelperROSLoadMessages` function loaded a laser scan message that is fully populated and is stored in the `scan` variable.

Inspect the `scan` variable. The primary data in the message is in the `Ranges` property. The data in `Ranges` is a vector of obstacle distances recorded at small angle increments.

```
scan

scan =
  ROS LaserScan message with properties:

       MessageType: 'sensor_msgs/LaserScan'
            Header: [1x1 Header]
```

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

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

You can get the measured points in Cartesian coordinates using the `readCartesian` function.

```
xy = readCartesian(scan);
```

This populates `xy` with a list of `[x,y]` coordinates that were calculated based on all valid range readings. Visualize the scan message using the `plot` function:

```
figure
plot(scan,'MaximumRange',5)
```
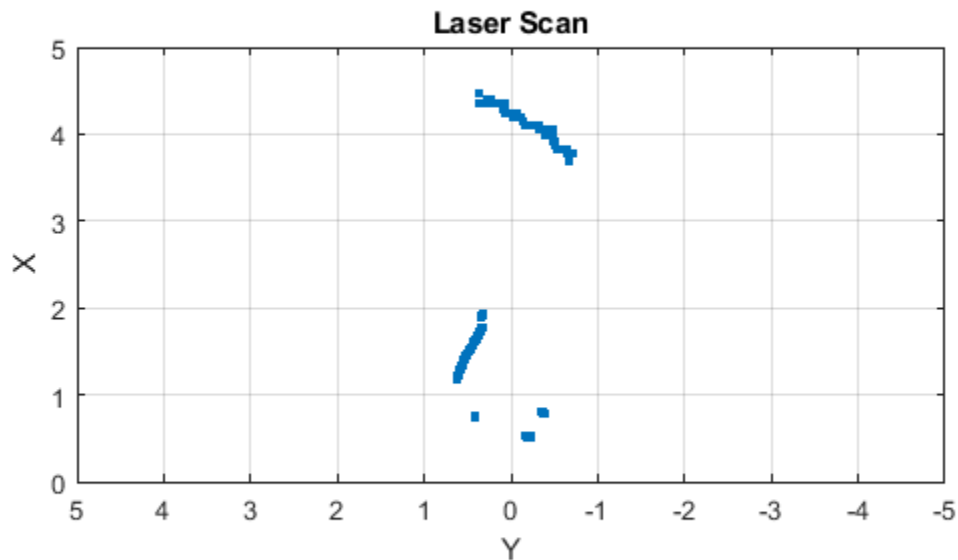

Image Messages

MATLAB also provides support for image messages, which always have the message type `sensor_msgs/Image`.

Create an empty image message using `rosmessage` to see the standard ROS format for an image message.

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

emptyimg =
  ROS Image message with properties:

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

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

For convenience, the `exampleHelperROSLoadMessages` function loaded an image message that is fully populated and is stored in the `img` variable.

Inspect the image message variable `img` in your workspace. The size of the image is stored in the `Width` and `Height` properties. ROS sends the actual image data using a vector in the `Data` property.

```
img

img =
  ROS Image message with properties:

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

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

The `Data` property stores raw image data that cannot be used directly for processing and visualization in MATLAB. You can use the `readImage` function to retrieve the image in a format that is compatible with MATLAB.

```
imageFormatted = readImage(img);
```

The original image has an 'rgb8' encoding. By default, `readImage` returns the image in a standard 480-by-640-by-3 `uint8` format. View this image using the `imshow` function.

```
figure
imshow(imageFormatted)
```
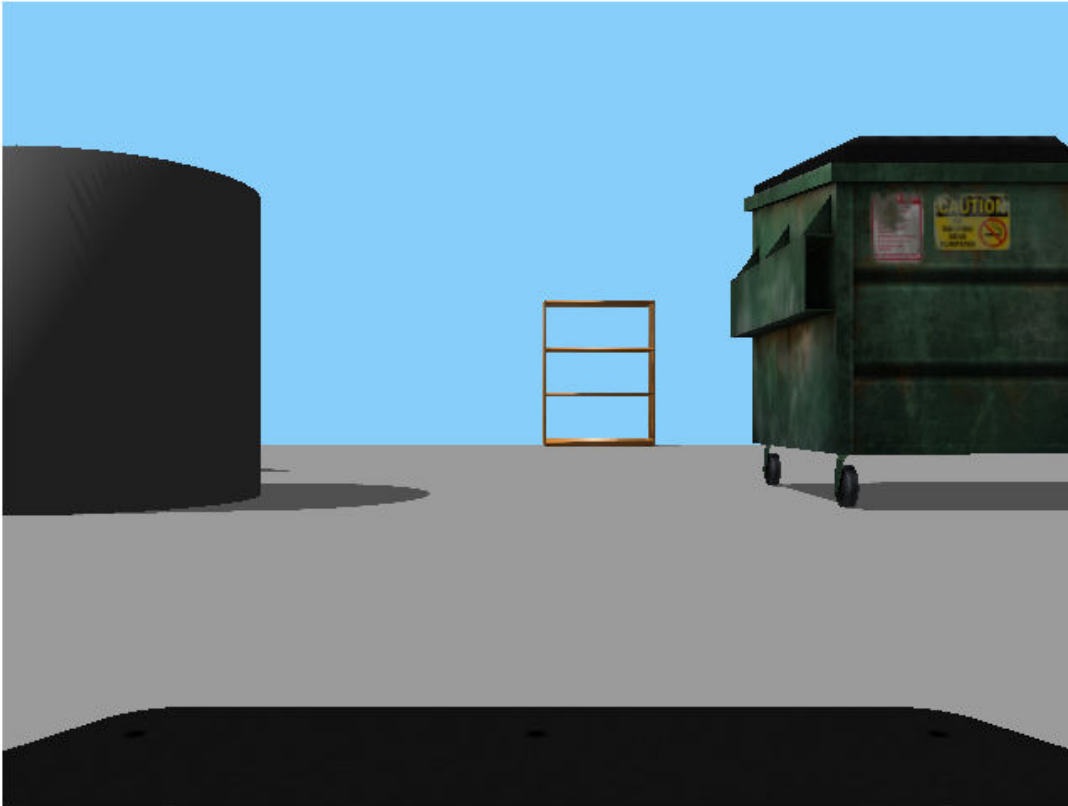
MATLAB® supports all ROS image encoding formats, and `readImage` handles the complexity of converting the image data. In addition to color images, MATLAB also supports monochromatic and depth images.

Compressed Messages

Many ROS systems send their image data in a compressed format. MATLAB provides support for these compressed image messages.

Create an empty compressed image message using `rosmessage`. Compressed images in ROS have the message type `sensor_msgs/CompressedImage` and have a standard structure.

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

emptyimgcomp =
  ROS CompressedImage message with properties:

    MessageType: 'sensor_msgs/CompressedImage'
         Header: [1x1 Header]
         Format: ''
           Data: [0x1 uint8]

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

For convenience, the `exampleHelperROSLoadMessages` function loaded a compressed image message that is already populated.

Inspect the `imgcomp` variable that was captured by a camera. The `Format` property captures all the information that MATLAB needs to decompress the image data stored in `Data`.

```
imgcomp
```

```
imgcomp =
  ROS CompressedImage message with properties:

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

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

Similar to the image message, you can use `readImage` to obtain the image in standard RGB format. Even though the original encoding for this compressed image is `bgr8`, `readImage` the conversion.

```
compressedFormatted = readImage(imgcomp);
```

Visualize the image using the `imshow` function.

```
figure
imshow(compressedFormatted)
```

Most image formats are supported for the compressed image message type. The `16UC1` and `32FC1` encodings are not supported for compressed depth images. Monochromatic and color image encodings are supported.

Point Clouds

Point clouds can be captured by a variety of sensors used in robotics, including LIDARs, Kinect®, and stereo cameras. The most common message type in ROS for transmitting point clouds is `sensor_msgs/PointCloud2` and MATLAB provides some specialized functions for you to work with this data.

You can see the standard ROS format for a point cloud message by creating an empty point cloud message.

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

emptyptcloud =
  ROS PointCloud2 message with properties:

    PreserveStructureOnRead: 0
                MessageType: 'sensor_msgs/PointCloud2'
                     Header: [1x1 Header]
                     Fields: [0x1 PointField]
```

```
                          Height: 0
                           Width: 0
                     IsBigendian: 0
                       PointStep: 0
                         RowStep: 0
                            Data: [0x1 uint8]
                         IsDense: 0

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

View the populated point cloud message that is stored in the `ptcloud` variable in your workspace:

`ptcloud`

```
ptcloud =
  ROS PointCloud2 message with properties:

    PreserveStructureOnRead: 0
                MessageType: 'sensor_msgs/PointCloud2'
                     Header: [1x1 Header]
                     Fields: [4x1 PointField]
                     Height: 480
                      Width: 640
                IsBigendian: 0
                  PointStep: 32
                    RowStep: 20480
                       Data: [9830400x1 uint8]
                    IsDense: 0

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

The point cloud information is encoded in the `Data` property of the message. You can extract the x, y, z coordinates as an N-by-3 matrix by calling the `readXYZ` function.

`xyz = readXYZ(ptcloud)`

xyz = 307200x3 single matrix

```
    NaN    NaN    NaN
    NaN    NaN    NaN
    NaN    NaN    NaN
    NaN    NaN    NaN
    NaN    NaN    NaN
    NaN    NaN    NaN
    NaN    NaN    NaN
    NaN    NaN    NaN
    NaN    NaN    NaN
    NaN    NaN    NaN
      ⋮
```

`NaN` in the point cloud data indicates that some of the x, y, z values are not valid. This is an artifact of the Kinect® sensor, and you can safely remove all `NaN` values.

`xyzvalid = xyz(~isnan(xyz(:,1)),:)`

xyzvalid = 193359x3 single matrix

```
    0.1378   -0.6705    1.6260
    0.1409   -0.6705    1.6260
    0.1433   -0.6672    1.6180
    0.1464   -0.6672    1.6180
    0.1502   -0.6705    1.6260
    0.1526   -0.6672    1.6180
    0.1556   -0.6672    1.6180
    0.1587   -0.6672    1.6180
    0.1618   -0.6672    1.6180
    0.1649   -0.6672    1.6180
      ⋮
```

Some point cloud sensors also assign RGB color values to each point in a point cloud. If these color values exist, you can retrieve them with a call to `readRGB`.

```
rgb = readRGB(ptcloud)
```

rgb = *307200×3*

```
    0.8392    0.7059    0.5255
    0.8392    0.7059    0.5255
    0.8392    0.7137    0.5333
    0.8392    0.7216    0.5451
    0.8431    0.7137    0.5529
    0.8431    0.7098    0.5569
    0.8471    0.7137    0.5569
    0.8549    0.7098    0.5569
    0.8588    0.7137    0.5529
    0.8627    0.7137    0.5490
      ⋮
```

You can visualize the point cloud with the `scatter3` function. `scatter3` automatically extracts the x,y,z coordinates and the RGB color values (if they exist) and show them in a 3-D scatter plot. The `scatter3` function ignores all NaN x,y,z coordinates, even if RGB values exist for that point.

```
figure
scatter3(ptcloud)
```

Point Cloud
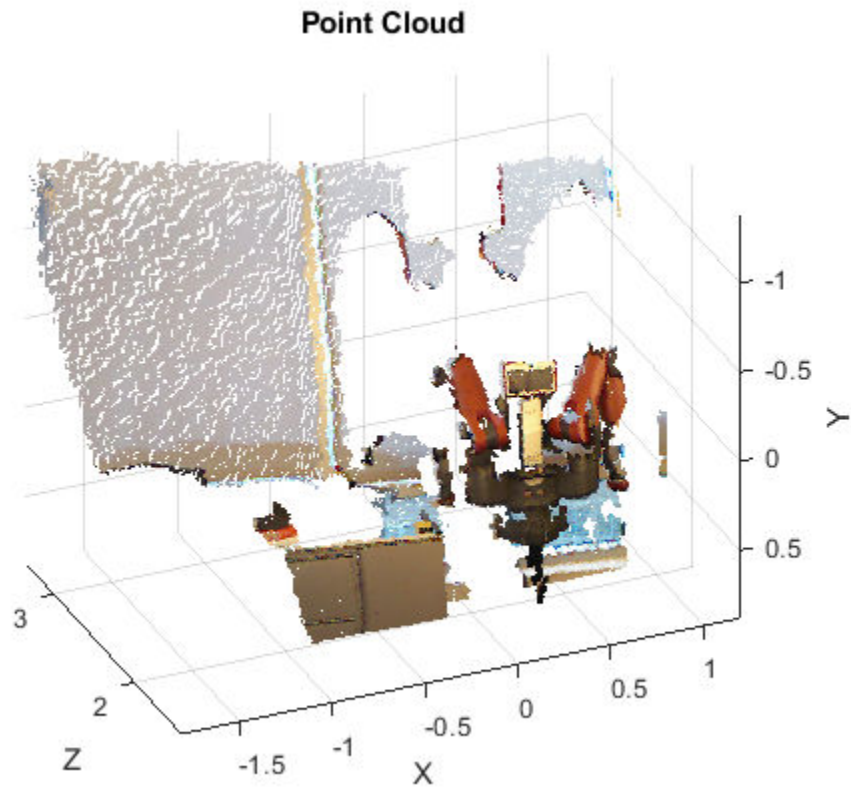
Work with Velodyne ROS Messages

Velodyne ROS messages store data in a format that requires some interpretation before it can be used for further processing. MATLAB® can help you by formatting Velodyne ROS messages for easy use. In this example, you can explore how VelodyneScan messages from a Velodyne LiDAR are handled.

Prerequisites: "Work with Basic ROS Messages" on page 1-15

Load Sample Messages

Load sample Velodyne messages. These messages are populated with data gathered from Velodyne LiDAR sensor.

```
load('lidarData_ConstructionRoad.mat')
```

VelodyneScan Messages

VelodyneScan messages are ROS messages that contain Velodyne LIDAR scan packets. You can see the standard ROS format for a VelodyneScan message by creating an empty message of the appropriate type.

```
emptyveloScan = rosmessage('velodyne_msgs/VelodyneScan')
```

```
emptyveloScan =
  ROS VelodyneScan message with properties:

    MessageType: 'velodyne_msgs/VelodyneScan'
         Header: [1x1 Header]
        Packets: [0x1 VelodynePacket]

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

Since you created an empty message, emptyveloScan does not contain any meaningful data. For convenience, the loaded lidarData_ConstructionRoad.mat filecontains a set of VelodyneScan messages that is fully populated and is stored in the msgs variable. Each element in the msgs cell array is a VelodyneScan ROS message struct. The primary data in each VelodyneScan message is in the Packets property, it contains multiple VelodynePacket messages. You can see the standard ROS format for a VelodynePacket message by creating an empty message of the appropriate type.

```
emptyveloPkt = rosmessage('velodyne_msgs/VelodynePacket')
```

```
emptyveloPkt =
  ROS VelodynePacket message with properties:

    MessageType: 'velodyne_msgs/VelodynePacket'
          Stamp: [1x1 Time]
           Data: [1206x1 uint8]

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

Create Velodyne ROS Message Reader

The `velodyneROSMessageReader` object reads point cloud data from `VelodyneScan` ROS messages based on their specified model type. Note that providing an incorrect device model may result in improperly calibrated point clouds. This example uses messages from the `'HDL32E'` model.

```
veloReader = velodyneROSMessageReader(msgs,'HDL32E')
```

```
veloReader =
  velodyneROSMessageReader with properties:

    VelodyneMessages: {28x1 cell}
         DeviceModel: 'HDL32E'
     CalibrationFile: 'B:\matlab\toolbox\shared\pointclouds\utilities\velodyneFileReaderConfigura
      NumberOfFrames: 55
            Duration: 2.7477 sec
           StartTime: 1145.2 sec
             EndTime: 1147.9 sec
          Timestamps: [1x55 duration]
         CurrentTime: 1145.2 sec
```

Extract Point Clouds

You can extract point clouds from the raw packets message with the help of this `velodyneROSMessageReader` object. By providing a specific frame number or timestamp, one point cloud can be extracted from `velodyneROSMessageReader` object using the `readFrame` object function. If you call `readFrame` without a frame number or timestamp, it extracts the next point cloud in the sequence based on the `CurrentTime` property.

Create a duration scalar that represents one second after the first point cloud reading.

```
timeDuration = veloReader.StartTime + duration(0,0,1,'Format','s');
```

Read the first point cloud recorded at or after the given time duration.

```
ptCloudObj = readFrame(veloReader,timeDuration);
```

Access `Location` data in the point cloud.

```
ptCloudLoc = ptCloudObj.Location;
```

Reset the `CurrentTime` property of `veloReader` to the default value

```
reset(veloReader)
```

Display All Point Clouds

You can also loop through all point clouds in the input Velodyne ROS messages.

Define *x-, y-,* and *z*-axes limits for `pcplayer` in meters. Label the axes.

```
xlimits = [-60 60];
ylimits = [-60 60];
zlimits = [-20 20];
```

Create the point cloud player.

```
player = pcplayer(xlimits,ylimits,zlimits);
```

Label the axes.

```
xlabel(player.Axes,'X (m)');
ylabel(player.Axes,'Y (m)');
zlabel(player.Axes,'Z (m)');
```

The first point cloud of interest is captured at 0.3 second into the input messages. Set the CurrentTime property to that time to begin reading point clouds from there.

```
veloReader.CurrentTime = veloReader.StartTime + seconds(0.3);
```

Display the point cloud stream for 2 seconds. To check if a new frame is available and continue past 2 seconds, remove the last while condition. Iterate through the file by calling readFrame to read in point clouds. Display them using the point cloud player.

```
while(hasFrame(veloReader) && isOpen(player) && (veloReader.CurrentTime < veloReader.StartTime +
    ptCloudObj = readFrame(veloReader);
    view(player,ptCloudObj.Location,ptCloudObj.Intensity);
    pause(0.1);
end
```
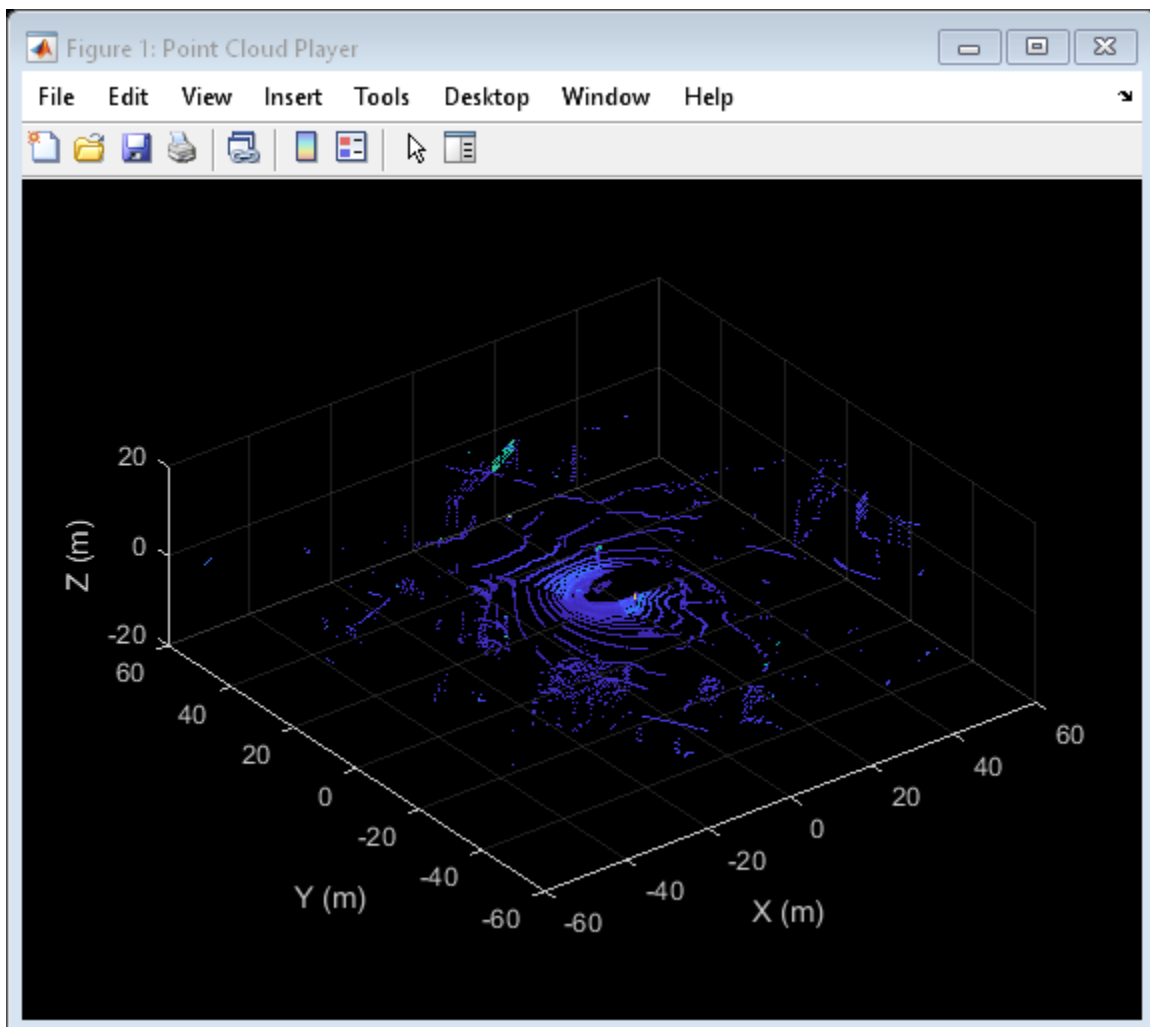
Get Started with a Real TurtleBot

This example shows how to connect to a TurtleBot® using the MATLAB® ROS interface. You can use this interface to connect to a wide range of ROS-supported hardware from MATLAB. If you are using a TurtleBot in Gazebo® refer to the "Get Started with Gazebo and a Simulated TurtleBot" on page 1-102 example.

Set Up New TurtleBot Hardware

The following steps use the *TurtleBot 3 Waffle Pi* platform (https://www.turtlebot.com/). The kit comes with a Raspberry Pi that has a pre-installed copy of ROS with the appropriate TurtleBot software. This procedure assumes that you are using a new TurtleBot of similar configuration. If you are already using a TurtleBot and communicating with it through an external computer, do not perform this procedure.
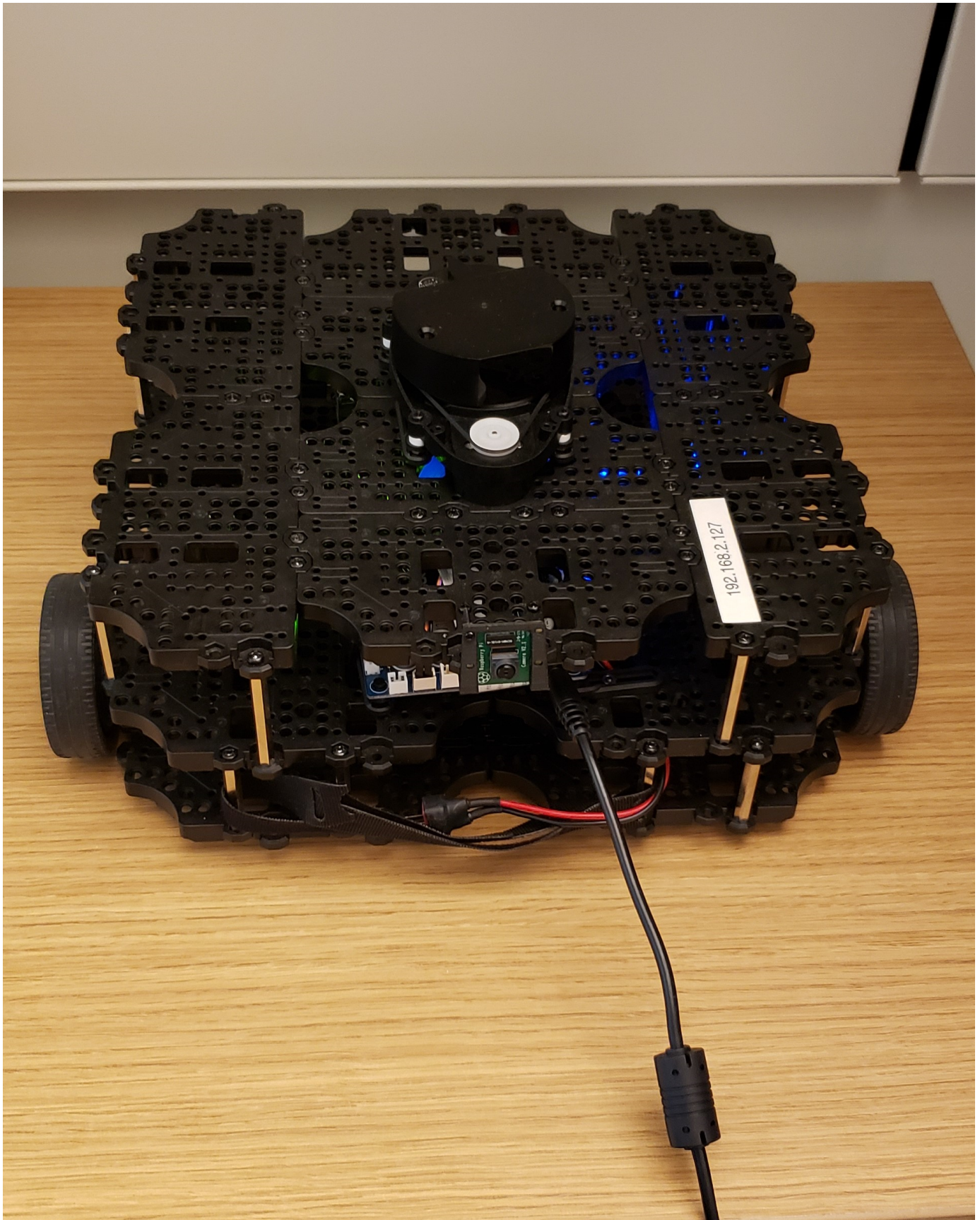
- Unpack the TurtleBot and make sure the power source is connected.
- Turn on the Raspberry Pi.
- Make sure that you have a network set up to connect the host computer (the one with MATLAB) to the Raspberry Pi on this TurtleBot. Use a wireless router or an Ethernet cable.
- Open a terminal on the Raspberry Pi and run `ifconfig`. The IP address associated with the network that you connected to is displayed.

- Set the appropriate environment variables on the TurtleBot by executing the following commands. Execute these command only once.

```
echo export ROS_IP=IP_OF_TURTLEBOT >> ~/.bashrc
echo export ROS_MASTER_URI=http://IP_OF_TURTLEBOT:11311 >> ~/.bashrc
sudo sh -c 'echo export ROS_IP=IP_OF_TURTLEBOT >> /etc/ros/setup.sh'
```

Make sure that you can ping the host machine from the Raspberry Pi:

```
ping IP_OF_HOST_COMPUTER
```

A successful `ping` is shown on the left. An unsuccessful `ping` is shown on the right.

Note: These environment variables must always have the correct IP address assigned to the TurtleBot. If the IP address of the TurtleBot Raspberry Pi changes, you must also change the environment variables using the preceding commands.
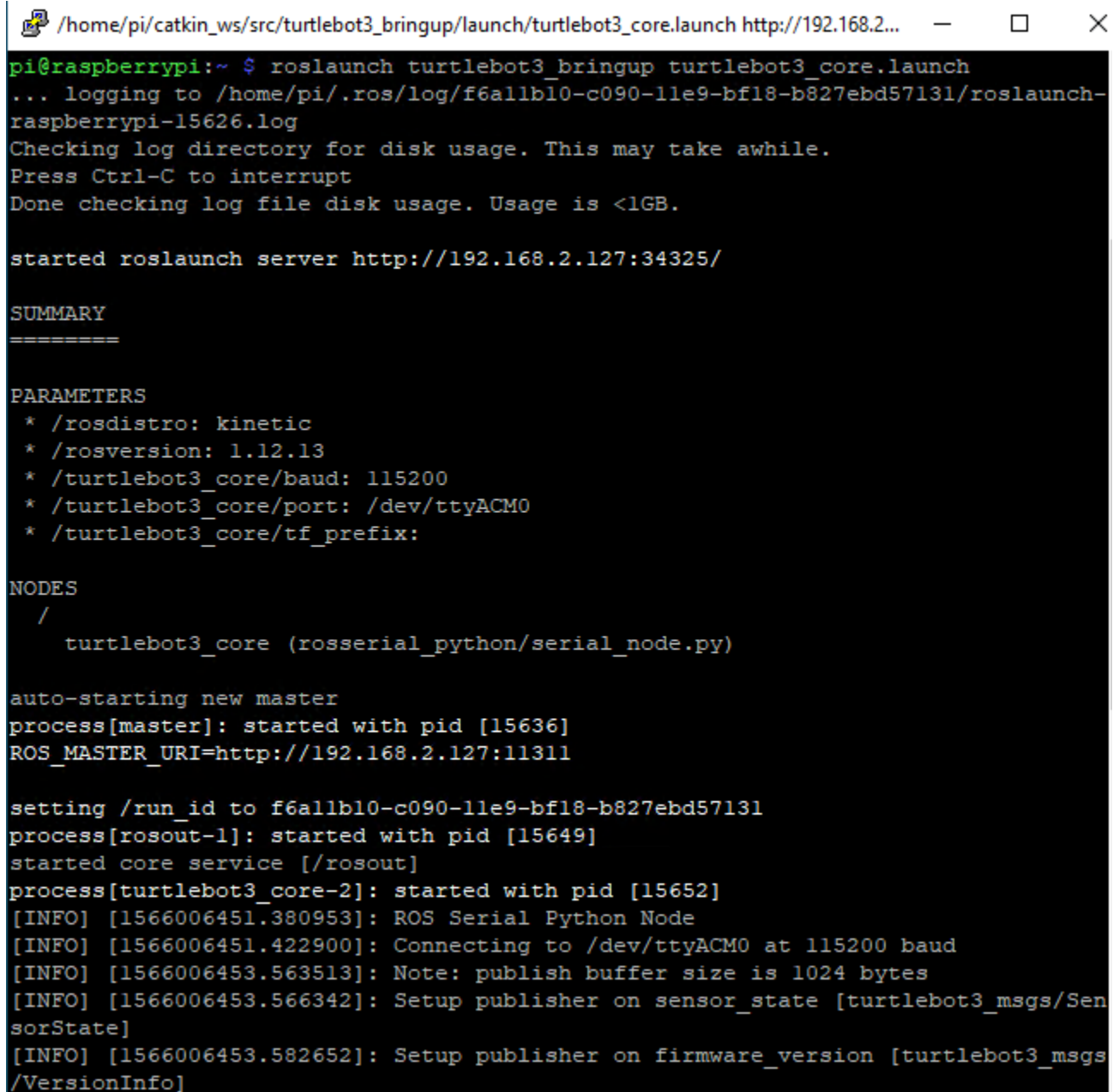
Here is a diagram illustrating the proper assignment of environment variables:
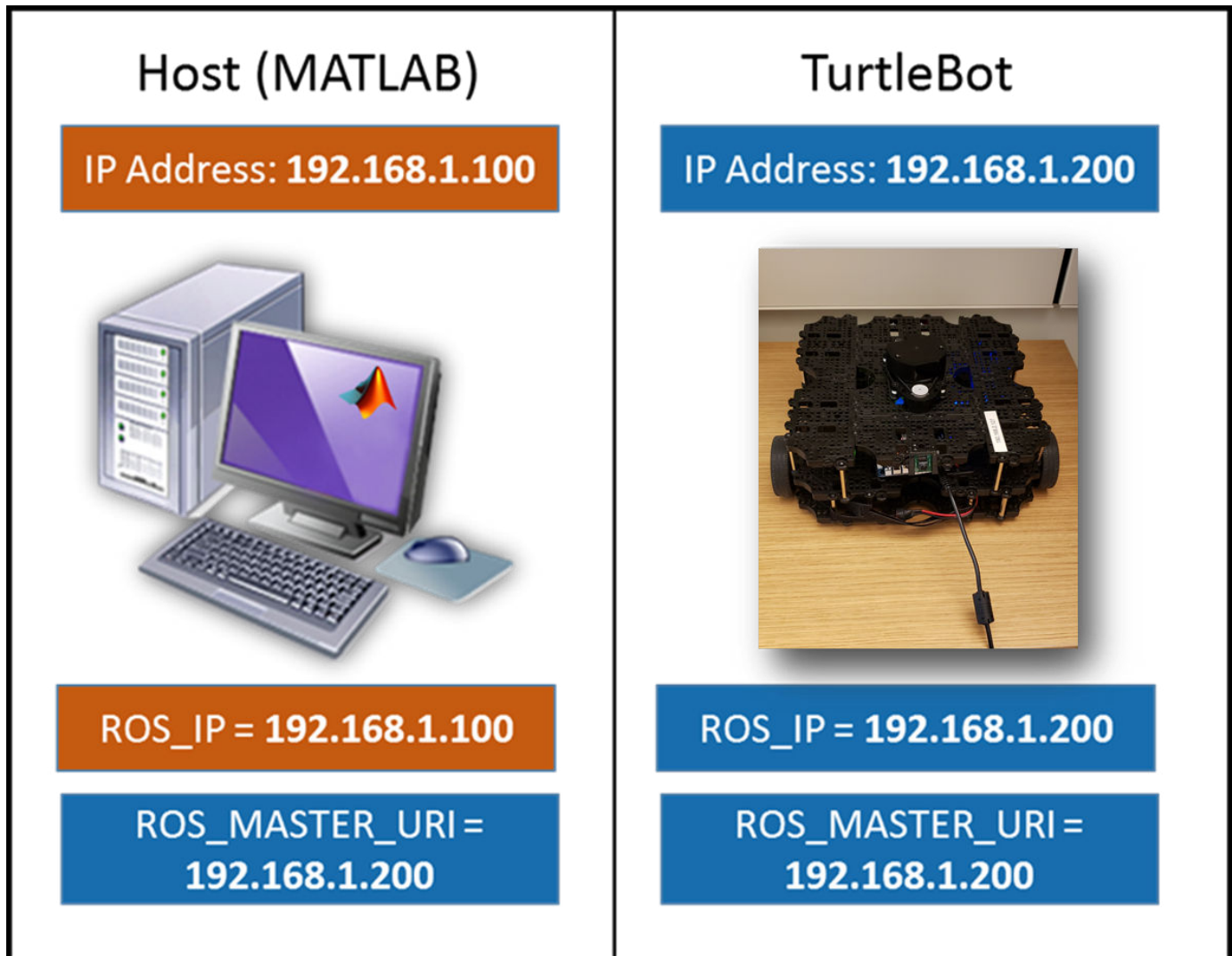
Type the following commands in separate terminals on the TurtleBot Raspberry Pi to launch LiDAR and camera sensors in TurtleBot 3:

```
roslaunch turtlebot3_bringup turtlebot3_core.launch
roslaunch turtlebot3_bringup turtlebot3_lidar.launch
roslaunch turtlebot3_bringup turtlebot3_rpicamera.launch
```

A possible output is shown below.

```
/home/pi/catkin_ws/src/turtlebot3_bringup/launch/turtlebot3_core.launch http://192.168.2...    —    □    ×

pi@raspberrypi:~ $ roslaunch turtlebot3_bringup turtlebot3_core.launch
... logging to /home/pi/.ros/log/f6a11b10-c090-11e9-bf18-b827ebd57131/roslaunch-
raspberrypi-15626.log
Checking log directory for disk usage. This may take awhile.
Press Ctrl-C to interrupt
Done checking log file disk usage. Usage is <1GB.

started roslaunch server http://192.168.2.127:34325/

SUMMARY
========


PARAMETERS
 * /rosdistro: kinetic
 * /rosversion: 1.12.13
 * /turtlebot3_core/baud: 115200
 * /turtlebot3_core/port: /dev/ttyACM0
 * /turtlebot3_core/tf_prefix:

NODES
  /
    turtlebot3_core (rosserial_python/serial_node.py)

auto-starting new master
process[master]: started with pid [15636]
ROS_MASTER_URI=http://192.168.2.127:11311

setting /run_id to f6a11b10-c090-11e9-bf18-b827ebd57131
process[rosout-1]: started with pid [15649]
started core service [/rosout]
process[turtlebot3_core-2]: started with pid [15652]
[INFO] [1566006451.380953]: ROS Serial Python Node
[INFO] [1566006451.422900]: Connecting to /dev/ttyACM0 at 115200 baud
[INFO] [1566006453.563513]: Note: publish buffer size is 1024 bytes
[INFO] [1566006453.566342]: Setup publisher on sensor_state [turtlebot3_msgs/Sen
sorState]
[INFO] [1566006453.582652]: Setup publisher on firmware_version [turtlebot3_msgs
/VersionInfo]
```

Set Up Existing TurtleBot Hardware

If you have a TurtleBot with a different setup from the setup previously described, before trying to communicate through MATLAB make sure that the following information is true:

- You have set up your network so that you can ping the host machine.
- You have access to the following topics. On the TurtleBot Raspberry Pi, type `rostopic list` to see the topics.

```
/odom
/cmd_vel
/reset
/scan
```

Host Computer Setup

- On the network, find the IP address of your host computer. On a Windows® machine, at the command prompt, type `ipconfig`. On a Mac or Linux® machine, open a terminal and type `ifconfig`. Here is an example of `ipconfig`:

Make sure that you can ping the notebook:

```
ping IP_OF_TURTLEBOT
```

A successful `ping` is shown first, followed by an unsuccessful `ping`.

```
Command Prompt                                           _  □  ✕

C:\Users\jgreco>ping 192.168.84.128

Pinging 192.168.84.128 with 32 bytes of data:
Reply from 192.168.84.128: bytes=32 time<1ms TTL=64
Reply from 192.168.84.128: bytes=32 time<1ms TTL=64
Reply from 192.168.84.128: bytes=32 time<1ms TTL=64
Reply from 192.168.84.128: bytes=32 time<1ms TTL=64

Ping statistics for 192.168.84.128:
    Packets: Sent = 4, Received = 4, Lost = 0 (0% loss),
Approximate round trip times in milli-seconds:
    Minimum = 0ms, Maximum = 0ms, Average = 0ms

C:\Users\jgreco>ping 192.168.1.130

Pinging 192.168.1.130 with 32 bytes of data:
Request timed out.
Request timed out.
Request timed out.
Request timed out.

Ping statistics for 192.168.1.130:
    Packets: Sent = 4, Received = 0, Lost = 4 (100% loss),

C:\Users\jgreco>_
```

Next Steps

- Refer to the next example: "Communicate with the TurtleBot" on page 1-130

Get Started with ROS in Simulink®

This example shows how to use Simulink blocks for ROS to send and receive messages from a local ROS network.

Introduction

Simulink support for Robot Operating System (ROS) enables you to create Simulink models that work with a ROS network. ROS is a communication layer that allows different components of a robot system to exchange information in the form of *messages*. A component sends a message by *publishing* it to a particular *topic*, such as /odometry. Other components receive the message by *subscribing* to that topic.

Simulink support for ROS includes a library of Simulink blocks for sending and receiving messages for a designated topic. When you simulate the model, Simulink connects to a ROS network, which can be running on the same machine as Simulink or on a remote system. Once this connection is established, Simulink exchanges messages with the ROS network until the simulation is terminated. If Simulink Coder™ is installed, you can also generate C++ code for a standalone ROS component, or *node*, from the Simulink model.

This example shows how to:

- Set up the ROS environment
- Create and run a Simulink model to send and receive ROS messages
- Work with data in ROS messages

Prerequisites: "Create a Simple Model" (Simulink), "Get Started with ROS" on page 1-2

Model

You will use Simulink to publish the X and Y location of a robot. You will also subscribe to the same location topic and display the received X,Y location.

Enter the following command to open the completed model created in example.

```
open_system('robotROSGetStartedExample');
```

Initialize ROS

Every ROS network has a *ROS master* that coordinates all the parts of the ROS network. For this example, use MATLAB® to create a ROS master on your local system. Simulink automatically detects and uses the local ROS master.

On the MATLAB command line, execute the following:

```
rosinit
```

Create a Publisher

Configure a block to send a geometry_msgs/Point message to a topic named /location (the "/" is standard ROS syntax).

- From the MATLAB Toolstrip, select **Home > Simulink** to open Simulink Start Page.
- On Simulink Start Page, under **Simulink**, click **Blank Model** to create and open a new Simulink Model.

- From the Simulink Toolstrip, select **Simulation > Library Browser** to open the Simulink Library Browser. Click on the **ROS Toolbox** tab (you can also type `roslib` in MATLAB command window). Select the **ROS** Library.

- Drag a **Publish** block to the model. Double-click on the block to configure the topic and message type.

- Select **Specify your own** for the **Topic source**, and enter `/location` in **Topic**.

- Click **Select** next to **Message type**. A pop-up window will appear. Select `geometry_msgs/Point` and click **OK** to close the pop-up window.
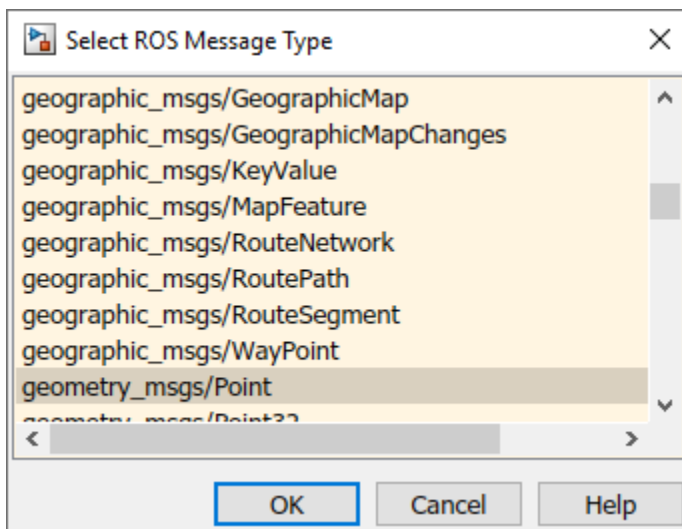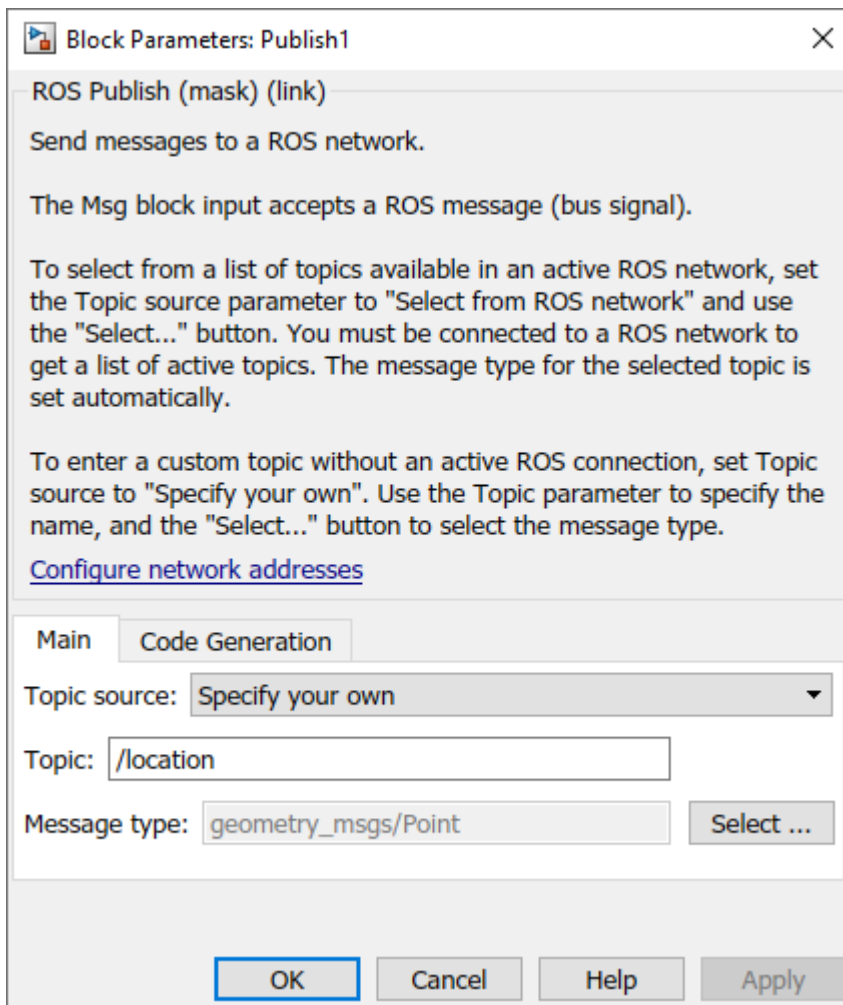
Create a ROS Message

Create a blank ROS message and populate it with the X and Y location for the robot path. Then publish the updated ROS message to the ROS network.

A ROS message is represented as a *bus signal* in Simulink. A bus signal is a bundle of Simulink signals, and can also include other bus signals (see the "Simulink Bus Signals" (Simulink) example for an overview). The ROS **Blank Message** block outputs a Simulink bus signal corresponding to a ROS message.

- Click **ROS Toolbox** tab in the Library Browser, or type `roslib` at the MATLAB command line. Select the **ROS** Library.

- Drag a **Blank Message** block to the model. Double-click on the block to open the block mask.

- Click on **Select** next to the **Message type** box, and select `geometry_msgs/Point` from the resulting pop-up window. Click **OK** to close the block mask.

- From the **Simulink > Signal Routing** tab in the Library Browser, drag a **Bus Assignment** block.

- Connect the output port of the **Blank Message** block to the Bus input port of the **Bus Assignment** block. Connect the output port of the **Bus Assignment** block to the input port of **ROS Publish** block.

- Double-click on the **Bus Assignment** block. You should see X, Y and Z (the signals comprising a `geometry_msgs/Point` message) listed on the left. Select `??? signal1` in the right listbox and click **Remove**. Select both X and Y signals in the left listbox and click **Select**. Click **OK** to apply changes.

NOTE: If you do not see X, Y and Z listed, close the block mask for the Bus Assignment block, and under the **Modeling** tab, click **Update Model** to ensure that the bus information is correctly propagated. If you see the error, "Selected signal 'signal1' in the Bus Assignment block cannot be found", it indicates that the bus information has not been propagated. Close the Diagnostic Viewer, and repeat the above step.

You can now populate the bus signal with the robot location.

- From the **Simulink > Sources** tab in the Library Browser, drag two **Sine Wave** blocks into the model.
- Connect the output ports of each **Sine Wave** block to the assignment input ports X and Y of the **Bus Assignment** block.
- Double-click on the **Sine Wave** block that is connected to input port X. Set the **Phase** parameter to -pi/2 and click **OK**. Leave the **Sine Wave** block connected to input port Y as default.

Your publisher should look like this:

At this point, the model is set up to publish messages to the ROS network. You can verify this as follows:

- Under the **Simulation** tab, set the simulation stop time to inf.
- Click **Run** to start simulation. Simulink creates a dedicated ROS node for the model and a ROS publisher corresponding to the **Publish** block.
- While the simulation is running, type rosnode list in the MATLAB command window. This lists all the nodes available in the ROS network, and includes a node with a name like / untitled_81473 (the name of the model along with a random number to make it unique).
- While the simulation is running, type rostopic list in the MATLAB command window. This lists all the topics available in the ROS network, and it includes /location.

```
>> rosnode list
/matlab_global_node_48012
/untitled_81473  ←——— Node created by Simulink

>> rostopic list
/location  ←——— Topic published from Simulink
/rosout
```

Observe that the base MATLAB workspace has one or more variables whose name starts with `SL_Bus_`. These are temporary bus objects created by Simulink and should not be modified. However, it is safe to clear them from the workspace, as they will be re-created if needed.

- Click **Stop** to stop the simulation. Simulink deletes the ROS node and ROS publisher. In general, the ROS node for a model and any associated publishers and subscribers are automatically deleted at the end of a simulation; no additional clean-up steps are required.

Create a Subscriber

Use Simulink to receive messages sent to the `/location` topic. You will extract the X and Y location from the message and plot it in the XY-plane.

- From the **ROS Toolbox** tab in the Library Browser, drag a **Subscribe** block to the model. Double-click on the block.
- Select **Specify your own** in the **Topic source** box, and enter `/location` in the **Topic** box.
- Click **Select** next to the **Message type** box, and select `geometry_msgs/Point` from the pop-up window. Click **OK** to close the block mask.

The **Subscribe** block outputs a Simulink bus signal, so you need to extract the X and Y signals from it.

- From the **Simulink > Signal Routing** tab in the Library Browser, drag a **Bus Selector** block to the model.
- Connect the **Msg** output of the **Subscribe** block to the input port of the **Bus Selector** block.
- From the **Modeling** tab, select **Update Model** to ensure that the bus information is propagated. You may get an error, "Selected signal 'signal1' in the Bus Selector block 'untitled/Bus Selector' cannot be found in the input bus signal". This error is expected, and will be resolved by the next step.
- Double-click on the **Bus Selector** block. Select `??? signal1` and `??? signal2` in the right listbox and click **Remove**. Select both X and Y signals in the left listbox and click **Select**. Click **OK**.

The **Subscribe** block will output the most-recently received message for the topic on every time step. The **IsNew** output indicates whether the message has been received during the prior time step. For the current task, the **IsNew** output is not needed, so do the following:

- From the **Simulink > Sinks** tab in the Library Browser, drag a **Terminator** block to the model.
- Connect the **IsNew** output of the **Subscribe** block to the input of the **Terminator** block.

The remaining steps configure the display of the extracted X and Y signals.

- From the **Simulink > Sinks** tab in the Library Browser, drag an **XY Graph** block to the model. Connect the output ports of the **Bus Selector** block to the input ports of the **XY Graph** block.

- From the **Simulink > Sinks** tab in the Library Browser, drag two **Display** blocks to the model. Connect each output of the **Bus Selector** block to each **Display** block.
- Save your model.

Your entire model should look like this:

Configure and Run the Model

- From the **Modeling** tab, select **Model Settings**. In the **Solver** pane, set **Type** to **Fixed-step** and **Fixed-step size** to `0.01`.
- Set simulation stop time to `10.0`.
- Click **Run** to start simulation. An XY plot will appear.

The first time you run the model in Simulink, the XY plot may look more jittery than the one above due to delays caused by loading ROS libraries. Once you rerun the simulation a few times, the plot should look smoother.

Note that the simulation **does not** work in actual or "real" time. The blocks in the model are evaluated in a loop that only simulates the progression of time, and is not intended to track actual clock time (for details, see "Simulation Loop Phase" (Simulink)).

Modify the Model to React Only to New Messages

In the above model, the **Subscribe** block outputs a message (bus signal) on every time step; if no messages have been received at all, it outputs a blank message (i.e., a message with zero values). Consequently, the XY coordinates are initially plotted at (0,0).

In this task, you will modify the model to use an **Enabled Subsystem**, so that it plots the location only when a new message is received (for more information, see "Using Enabled Subsystems" (Simulink)). A pre-configured model is included for your convenience.

- In the model, click and drag to select the **Bus Selector** block and **XY Graph** blocks. Right-click on the selection and select **Create Subsystem from Selection**.
- From the **Simulink > Ports & Subsystems** tab in the Library Browser, drag an **Enable** block into the newly-created subsystem.
- Connect the **IsNew** output of the **Subscribe** block to the enabled input of the subsystem as shown in the picture below. Delete the **Terminator** block. Note that the **IsNew** output is true only if a new message was received during the previous time step.

- Save your model.
- Click **Run** to start simulation. You should see the following XY plot.

The blocks in the enabled subsystem are only executed when a new ROS message is received by the **Subscribe** block. Hence, the initial `(0,0)` value will not be displayed in the XY plot.

Work with ROS Messages in Simulink®

This example illustrates how to work with complex ROS messages in Simulink, such as messages with nested sub-messages and variable-length arrays.

Introduction

In ROS Simulink Models, bus signals represent ROS Messages. Each field of a ROS message is corresponds to a field in a Simulink bus, with the following limitations:

- **Constants** are not supported, and are excluded from the Simulink bus.
- **64-bit Integers** (ROS types `int64` and `uint64`) convert to doubles in the Simulink bus, as Simulink does not natively support 64-bit integer datatypes.
- **Variable-length arrays** (ROS type `...[]`) convert to fixed-length array with customizable maximum lengths. By default, the fixed length is 128 for primitive types (e.g., `uint8[]`, `float32[]`), and 16 for nested arrays of messages (e.g., `geometry_msgs/Point[]`).
- **Strings** (ROS type `string`) convert to fixed-length `uint8` arrays with customizable maximum lengths, with a default maximum length of 128 characters.
- **String arrays** (ROS type `string[]`) convert to a fixed-length array of `std_msgs/String` with a customizable maximum length. The default maximum length is 16 strings.

When a Simulink bus converts to a ROS message, the message fields restore to their native ROS types. For example, the ROS message type `std_msgs/Header` has a field, `FrameId`, which is a string. In the corresponding Simulink bus, the `FrameId` field is a `uint8` array. When the bus converts to a ROS message, `FrameId` converts back to a string.

Model

The following model has several examples of working with complex ROS messages in Simulink. The rest of the tasks in this example focus on specific scenarios.

```
open_system('robotROSMessageUsageExample');
```

Access Data in a Variable-length Array

A ROS message can have arrays whose length (number of elements) cannot be pre-determined. For example, the `Position` field in a `sensor_msgs/JointState` message is a variable-length array of 64-bit floats. In any given `sensor_msgs/JointState` message, the `Position` array can have no elements or it can have an arbitrarily large number of elements. In Simulink, such arrays are required to have a maximum length.

Open the example model and explore how variable-length arrays in ROS messages are handled in Simulink in the following steps.

```
open_system('robotROSMessageUsageExample/Work with Variable-length Arrays');
```

- Double-click the **Work with Variable-length Arrays** subsystem. Note that the **Subscribe** block is configured to receive messages sent to topic `/my_joint_topic` as message type, `sensor_msgs/JointState`.
- Under the **Modeling** tab, click **Update Model.**
- Double-click on the **Bus Selector** block. There are three variable-length arrays in the message (`Position`, `Velocity`, and `Effort`).

- Observe that there is a `Position_SL_Info` field in the bus. `Position_SL_Info.ReceivedLength` holds the length of the `Position` array in the original received ROS message. This value can be arbitrarily large. `Position_SL_Info.CurrentLength` holds the length of the `Position` array in the Simulink bus signal. This can vary between 0 and the maximum length of the array (128, in this case).
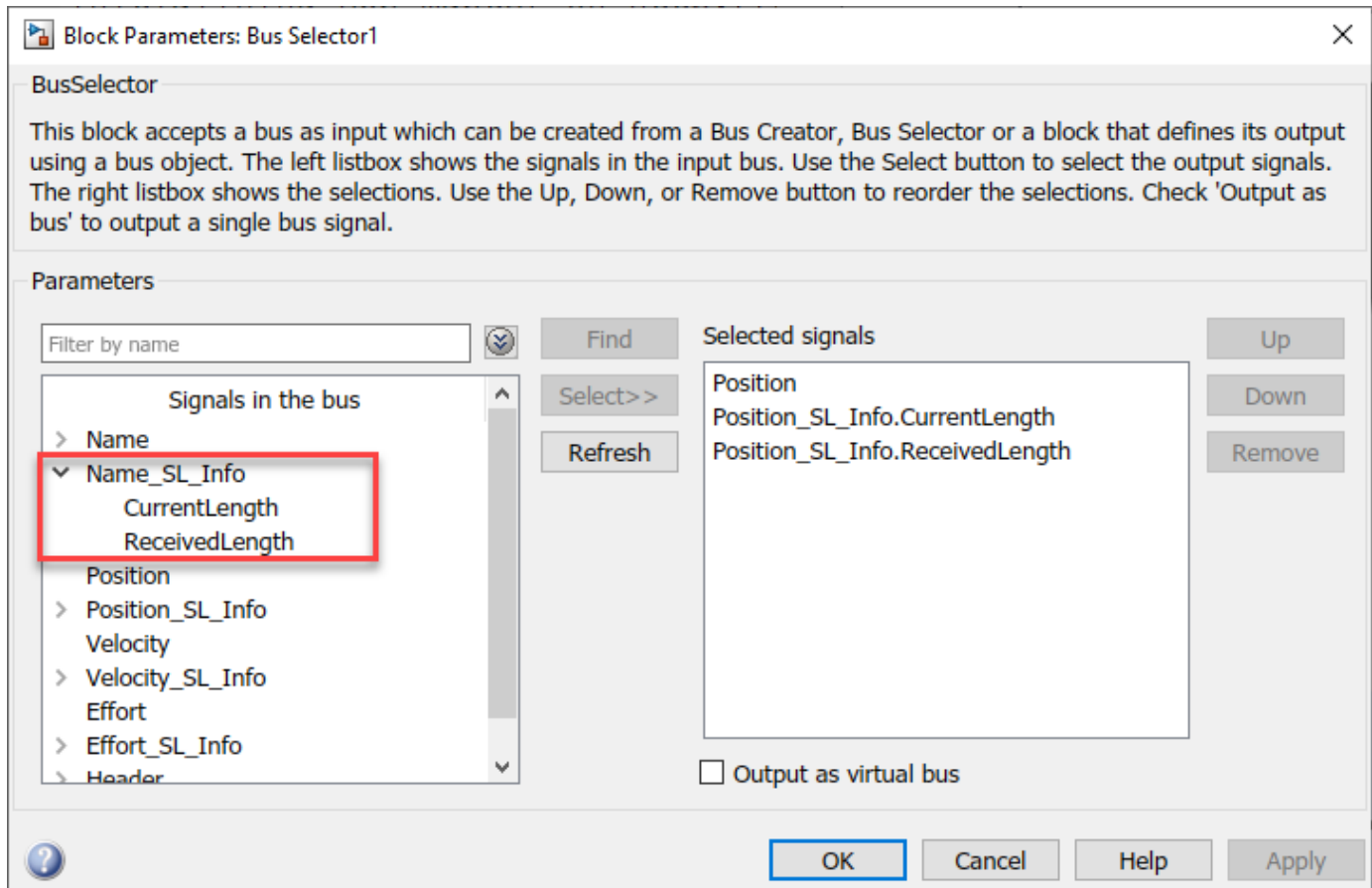
Configure ROS Network

- Under the **Simulation** tab, select **ROS Toolbox > ROS Network.** If you do not see ROS Toolbox, select **Model Settings** under the **Modeling** tab. In the **Configuration Parameters** dialog, select `Robot Operating System (ROS)` for the **Hardware Board** dropdown. Click **OK** to close the dialog box.
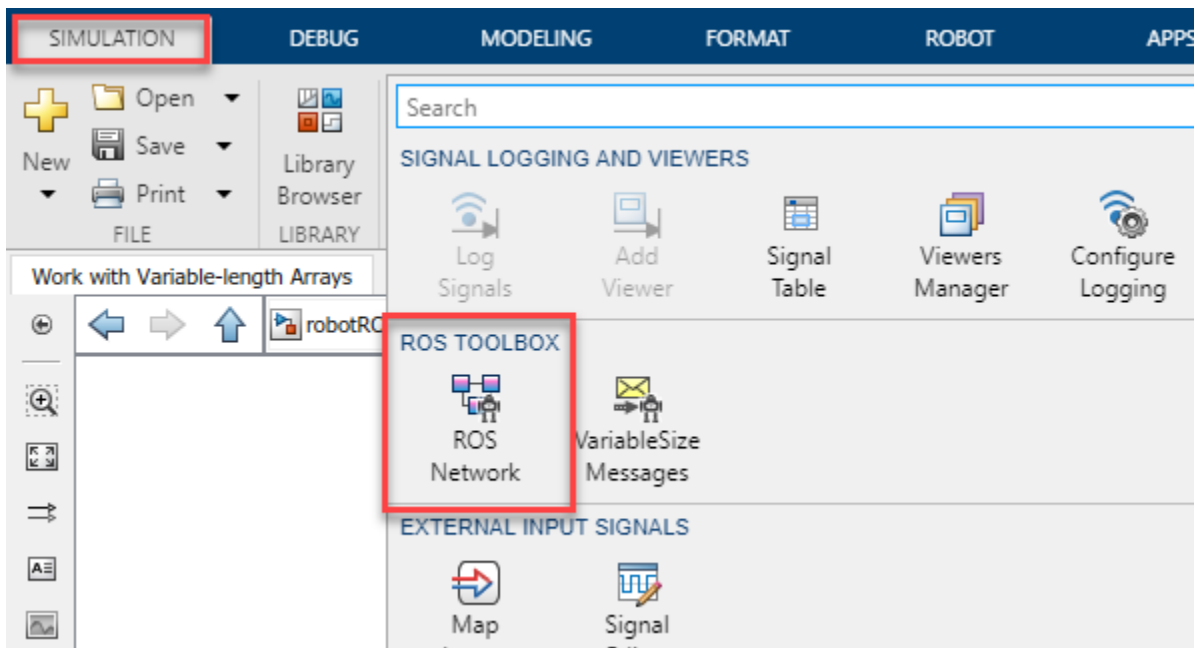
- Set the **Network Address** for both **ROS Master (ROS 1)** and **Node Host (ROS 1)** to `Default`.
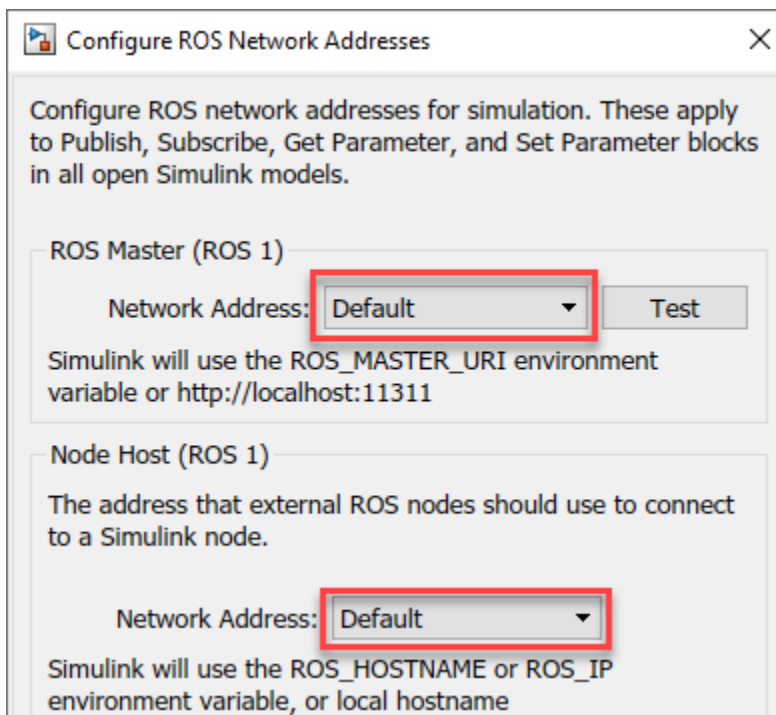
- Enter `rosinit` at the MATLAB® command line.

Run Simulation

- Under the Simulation tab, set **Stop Time** to `Inf`, and click **Play** to start simulation.
- Execute the following at the MATLAB command line.

```
[pub, msg] = rospublisher('/my_joint_state', 'sensor_msgs/JointState');
msg.Position = [11:2:25]; % array of length 8
send(pub, msg);
```

- Observe the **Display** outputs in the **Work with Variable-length Arrays** subsystem. Note that `Current Length` and `Received Length` are equal.
- Execute the following at the MATLAB command line.

```
msg.Position = 1:130; % array of length 130
send(pub, msg);
```

- Observe that a warning is emitted, indicating that a truncation has happened. The `Received Length` is now `130` and `Current Length` is `128`.
- Under the **Debug** tab, select **Diagnostics > Diagnostic Viewer**. Warnings are typically routed here to the Simulink Diagnostic Viewer (see "View Diagnostics" (Simulink)).

Modify Maximum Size of a Variable-length Array

Change the maximum size of a variable-length array in Simulink. The default maximum of the `Position` array in the `sensor_msgs/JointState` message type is `128`. You will change this limit to `256`.

- Open the example model, and double-click on the **Work with Variable-length Arrays** subsystem.
- From the **Simulation** tab, select **ROS Toolbox > Variable Size Messages**.
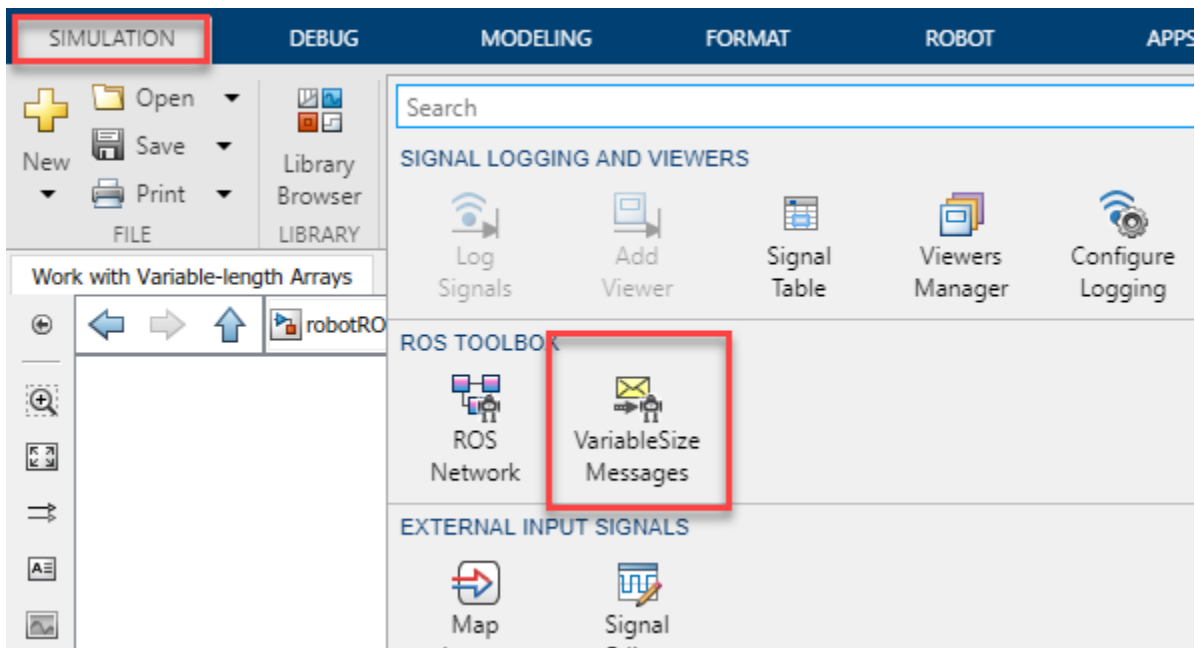
- From the list box on the left, click on `sensor_msgs/JointState`. Then, unselect the **Use default limits for this message type** checkbox. Finally, enter the new value (256) in the row for the `Position` array property, and click **OK** to close the dialog.
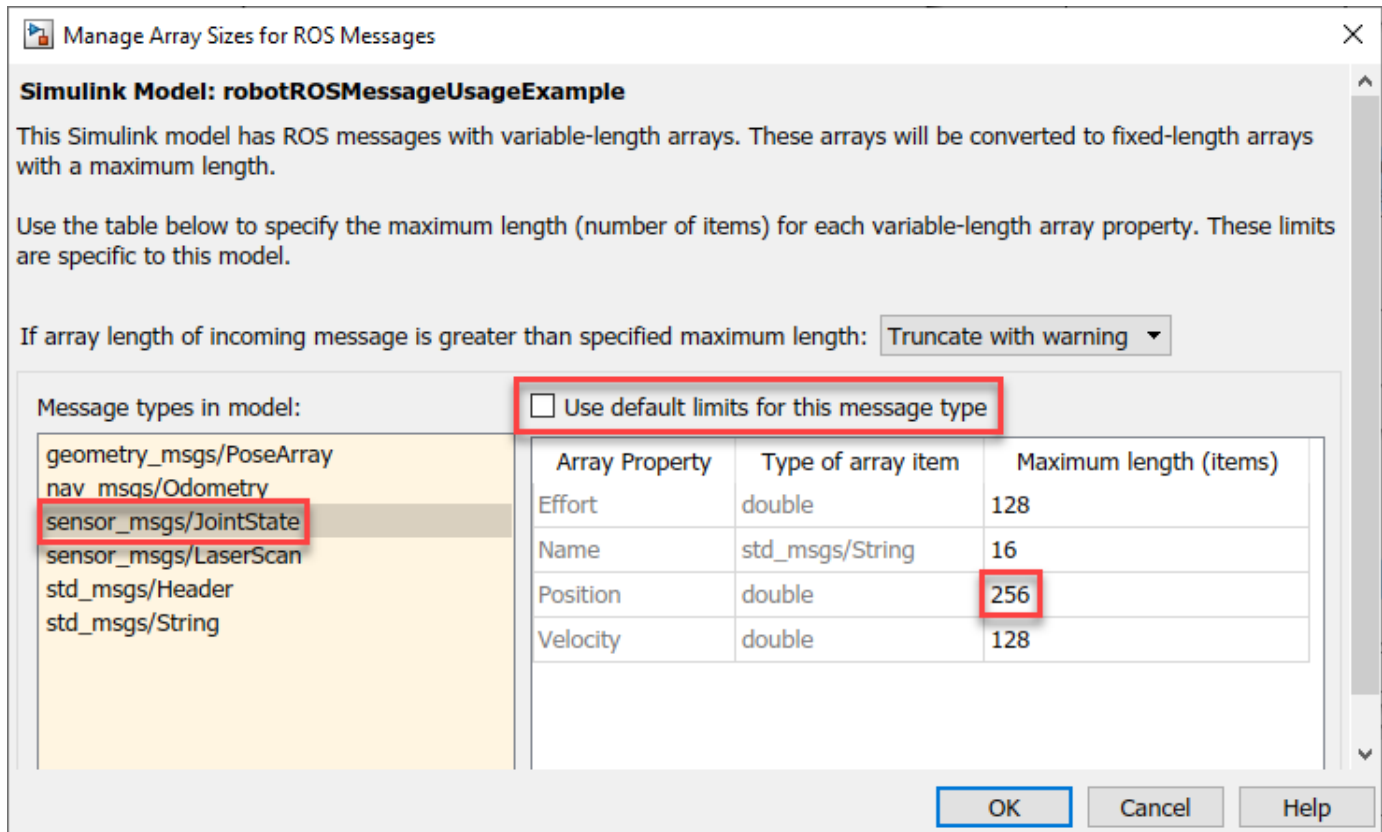
- Click **Play** to start simulation.
- Run the following at the MATLAB command line. Observe that a warning is not emitted in the **Diagnostic Viewer**.

```
msg.Position = 1:200; % array of length 200
send(pub, msg);
```

- Run the following at the MATLAB command line. Observe that a warning is emitted in the **Diagnostic Viewer**.

```
msg.Position = 1:300; % array of length 300
send(pub, msg);
```

- Close the model without saving.

Note:

- The maximum size information applies to all instances of the `sensor_msgs/JointState` message type. For example, if other messages used in the model include a `sensor_msgs/JointState` message, the updated limit of 256 will apply to all those nested instances as well.
- The maximum size information is specific to the model, and is saved with the model. You can have two models open that use `sensor_msgs/JointState`, with one model using the default limit of 128, and another using a custom limit of 256.

Work with Messages Using MATLAB Function Block

The Bus Assignment block in Simulink does not support assigning to an element inside an array of buses.

For example, a `geometry_msgs/PoseArray` message has a `Poses` property, which is required to be an array of `geometry_msgs/Pose` messages. If you want to assign to specific elements of the `Poses` array, that is not possible with the Bus Assignment block.

Explore how to use the MATLAB Function block for advanced message manipulation such as assignment of nested messages.

- Open the example model. Select the **Work with Nested Arrays of Messages** subsystem and copy.
- Open a new Simulink model. Paste and save the new model to a temporary location, with the name `FunctionTest.slx`.
- Close all models, and clear the base workspace by typing `clear` in the MATLAB command line.

Configure the MATLAB Assign Block

- Open the FunctionTest.slx model, double-click on the **Work with Nested Arrays of Messages** subsystem, and open the **MATLAB Function - Assign** block. Observe that it uses MATLAB notation to assign values inside a nested array.
- The Function Block requires the datatype of bus outputs (in this case, `msg`) to be explicitly specified. Create all buses required for this model by typing the following at the MATLAB command line. Note that the bus objects are created in the MATLAB workspace using the name `SL_Bus_<modelname>_<messageType>`. (This may be abbreviated if the model name is too long.)

`ros.createSimulinkBus(gcs)`

- Double-click the **MATLAB Function - Assign** block. In the MATLAB Editor, click **Edit Data**. In **Ports and Data Manager**, select `msg`, and set its type to `SL_Bus_FunctionTest_geometry_msgs_PoseArray`. Click **Apply** and close *Ports and Data Manager*.
- If you do not see `SL_Bus_FunctionTest_geometry_msgs_PoseArray` listed as an option in the **Type** dropdown, select **Refresh data types**.
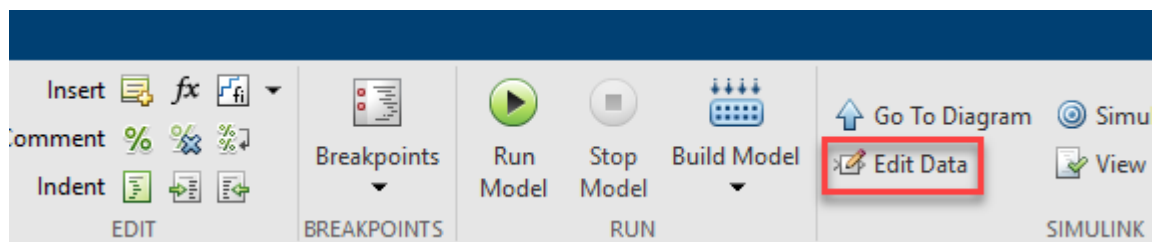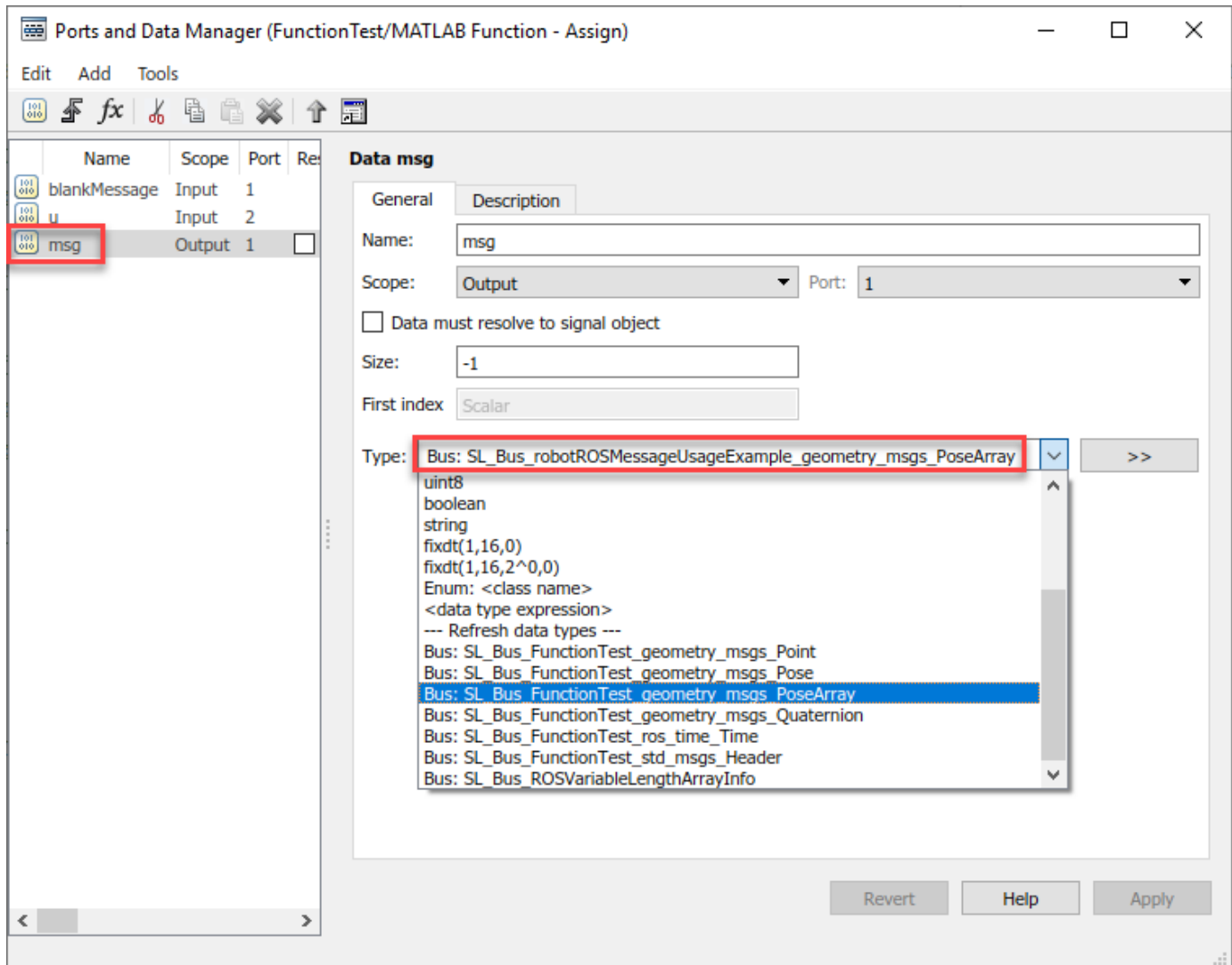
Configure the ROS Network

- Under the **Simulation** tab, select **ROS Toolbox > ROS Network**.
- Set the **Network Address** for both **ROS Master (ROS 1)** and **Node Host (ROS 1)** to `Default`.
- Enter `rosinit` at the MATLAB command line.

Run Simulation

- Under the **Simulation** tab, set **Stop Time** to `1.0`, and click **Play** to run the simulation. Verify that the values in the **Display** blocks are equal to `pi/2` and `pi/2 + 1`.
- The `ros.createSimulinkBus(gcs)` statement has to be re-run each time the model is loaded or if the workspace is cleared. To avoid these issues, include this statement in the InitFcn callback for the model (see "Model Callback Parameters" (Simulink)).

Work with String Arrays

A string array in a ROS message is represented in Simulink as an array of `std_msgs/String` messages. Each `std_msgs/String` message has a `Data` property that has the actual characters in the string. Each string is represented as an array of `uint8` values.

By default, the maximum number of `std_msgs/String` messages in a string array is `16`, and the maximum length of an individual string is `128` characters. The following steps show how to change these defaults:

Open the example model, and double-click the **Work with Strings and String Arrays** subsystem.

Change Maximum Array Lengths

- From the **Simulation** tab, select **ROS Toolbox > Variable Size Messages**.
- In the **Message types in model** column, click on the `sensor_msgs/JointState` entry. Observe that the right-hand pane shows a `Name` property that is an array of `std_msgs/String`, with a maximum length of `16`. To change the maximum number of strings in `Name`, deselect the **Use default limits for this message type** checkbox and enter the desired value.

- In the **Message types in model** column, click on the `std_msgs/String` entry. Observe that the right-hand pane shows a `Data` property that is an array of `uint8`, with a maximum length of `128`. To change the maximum length of the string, deselect the **Use default limits for this message type** checkbox and enter the desired value.
- Once you change the default values, open the **Work with Strings and String Arrays** subsystem and simulate the model. The **Display** blocks should now reflect the updated maximum values.

Note: The maximum length of `Data` applies to all instances of `std_msgs/String` in the model. For example, the **Blank String** block in **Work with Strings and String Arrays** subsystem uses a `std_msgs/String` message, so these messages would inherit the updated maximum length. Likewise, if the model has another ROS message type with a string array property, the individual strings in that array will also inherit the updated maximum length.

Connect to a ROS-enabled Robot from Simulink®

You can use Simulink to connect to a ROS-enabled physical robot or to a ROS-enabled robot simulator such as Gazebo. This example shows how to:

- Configure Simulink to connect to a separate robot simulator using ROS
- Send velocity commands to a simulated robot
- Receive position information from a simulated robot

You can follow the steps in the example to create your own model, or you can use this completed version instead.

```
open_system('robotROSConnectToRobotExample');
```

Prerequisites: "Get Started with ROS" on page 1-2, "Exchange Data with ROS Publishers and Subscribers" on page 1-20, "Get Started with ROS in Simulink®" on page 1-64.

Robot Simulator

Start a ROS-based simulator for a differential-drive robot. The simulator receives and sends messages on the following topics:

- Sends `nav_msgs/Odometry` messages to the `/odom` topic
- Receives `geometry_msgs/Twist` velocity command messages on the `/mobile_base/commands` or `/cmd_vel` topic, based on the ROS-based simulator

You can choose one of two options for setting up the ROS-based simulator.

Option A: Simulator in MATLAB®

Use a simple MATLAB-based simulator to plot the current location of the robot in a separate figure window.

- Enter `rosinit` at the MATLAB command line. This creates a local *ROS master* with network address (URI) of `http://localhost:11311`.
- Enter `ExampleHelperSimulinkRobotROS` to start the Robot Simulator:
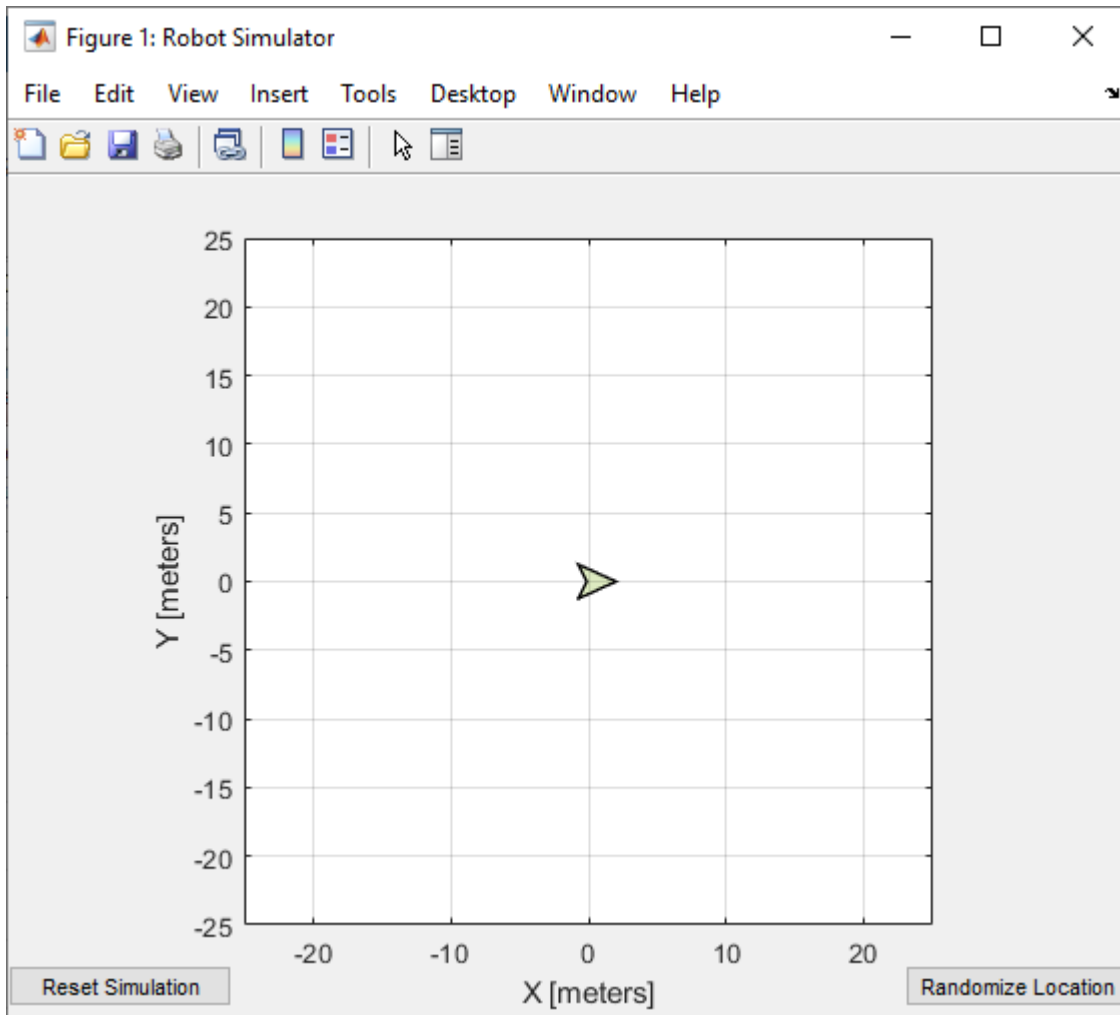
- **Note:** The `geometry_msgs/Twist` velocity command messages are received on the `/mobile_base/commands/velocity` topic.

Option B: Gazebo Simulator

Use a simulated TurtleBot® in Gazebo.

- See "Add, Build, and Remove Objects in Gazebo" on page 1-108 for instructions on setting up the Gazebo environment. In the Ubuntu® desktop in the virtual machine, click the "Gazebo Empty" icon.
- Note the network address (URI) of the ROS master. It will look like `http://192.168.84.128:11311`, but with your specific IP address.
- Verify that the Gazebo environment is properly set up by typing the `rostopic list` in the Ubuntu terminal window. You should see a list of topics, including `/cmd_vel` and `/odom`.
- **Note:** The `geometry_msgs/Twist` velocity command messages are received on the /cmd_vel topic.

Configure Simulink to Connect to the ROS Network

1. From the **Simulation** tab, select **ROS Toolbox > ROS Network**.
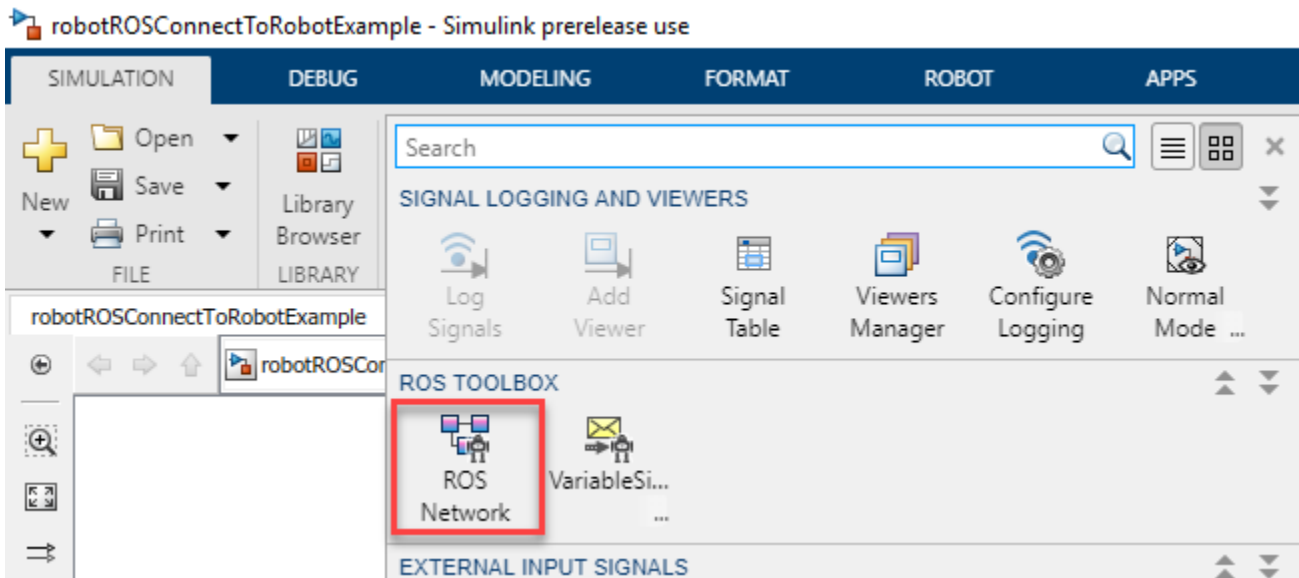
2. In the **ROS Master (ROS 1)** section, select `Custom` from the **Network Address** dropdown.

- Option A (MATLAB Simulator): Ensure that the **Hostname/IP Address** is set to `localhost`, and **Port** is set to `11311`.

- Option B (Gazebo Simulator): Specify the IP address and port number of the ROS master in Gazebo. For example, if it is `http://192.168.60.165:11311`, then enter `192.168.60.165` in the **Hostname/IP address** box and `11311` in the **Port** box.
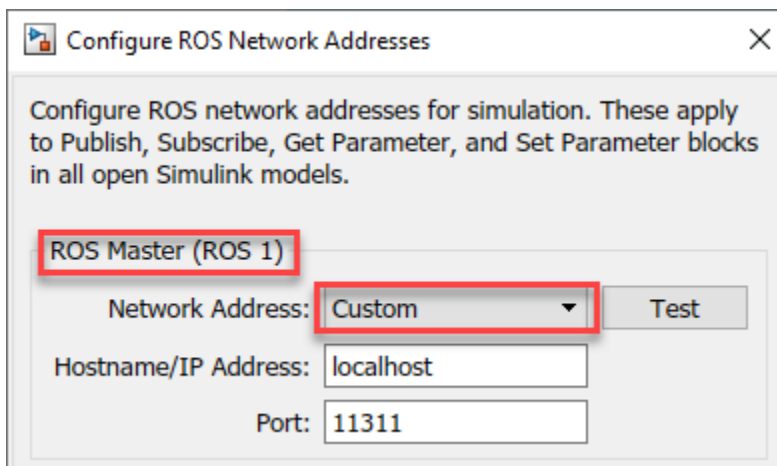
Send Velocity Commands To the Robot

Create a publisher that sends control commands (linear and angular velocities) to the simulator. Make these velocities adjustable by using **Slider Gain** blocks.

ROS uses a right-handed coordinate system, so X-axis is forward, Y-axis is left, and Z-axis is up. Control commands are sent using a `geometry_msgs/Twist` message, where `Linear.X` indicates linear forward velocity (in m/s), and `Angular.Z` indicates angular velocity around the Z-axis (in rad/s).

Configure a Publisher Block

1   Open a new Simulink model.

2   From the **ROS Toolbox > ROS** tab in the Library Browser, drag a **Publish** block to the model. Double-click the block.

3   Set **Topic source** field to **Select From ROS network**. Select a topic based on the simulator as shown below.

•   Option A (MATLAB Simulator): Click **Select** next to **Topic**, select `/mobile_base/commands/velocity`, and click **OK**. Note that the message type (`geometry_msgs/Twist`) is set automatically.

•   Option B (Gazebo Simulator): Click **Select** next to **Topic**, select `/cmd_vel`, and click **OK**. Note that the message type (`geometry_msgs/Twist`) is set automatically.

Configure a Message Block

1   From the **ROS Toolbox > ROS** tab in the Library Browser, drop a **Blank Message** block to the model. Double-click the block.

2   Click **Select** next to **Message type** and select `geometry_msgs/Twist`.

3   Set **Sample time** to `0.01` and click **OK**.

Configure Message Inputs

1   From the **Simulink > Signal Routing** tab in the Library Browser, drag a **Bus Assignment** block to the model.

2   Connect the `Msg` output of the **Blank Message** block to the `Bus` input of the **Bus Assignment** block, and the `Bus` output to the `Msg` input of the **Publish** block.

3   From the **Modeling** tab, click **Update Model** to ensure that the bus information is correctly propagated. Ignore the error, "Selected signal 'signal1' in the Bus Assignment block 'untitled/Bus Assignment' cannot be found in the input bus signal", if it appears. The next step will resolve this error.

4   Double-click the **Bus Assignment** block. Select `??? signal1` in the right list box and click **Remove**. In the left list box, expand both Linear and Angular properties. Select **Linear > X** and **Angular > Z** and click **Select**. Click **OK** to close the block mask.
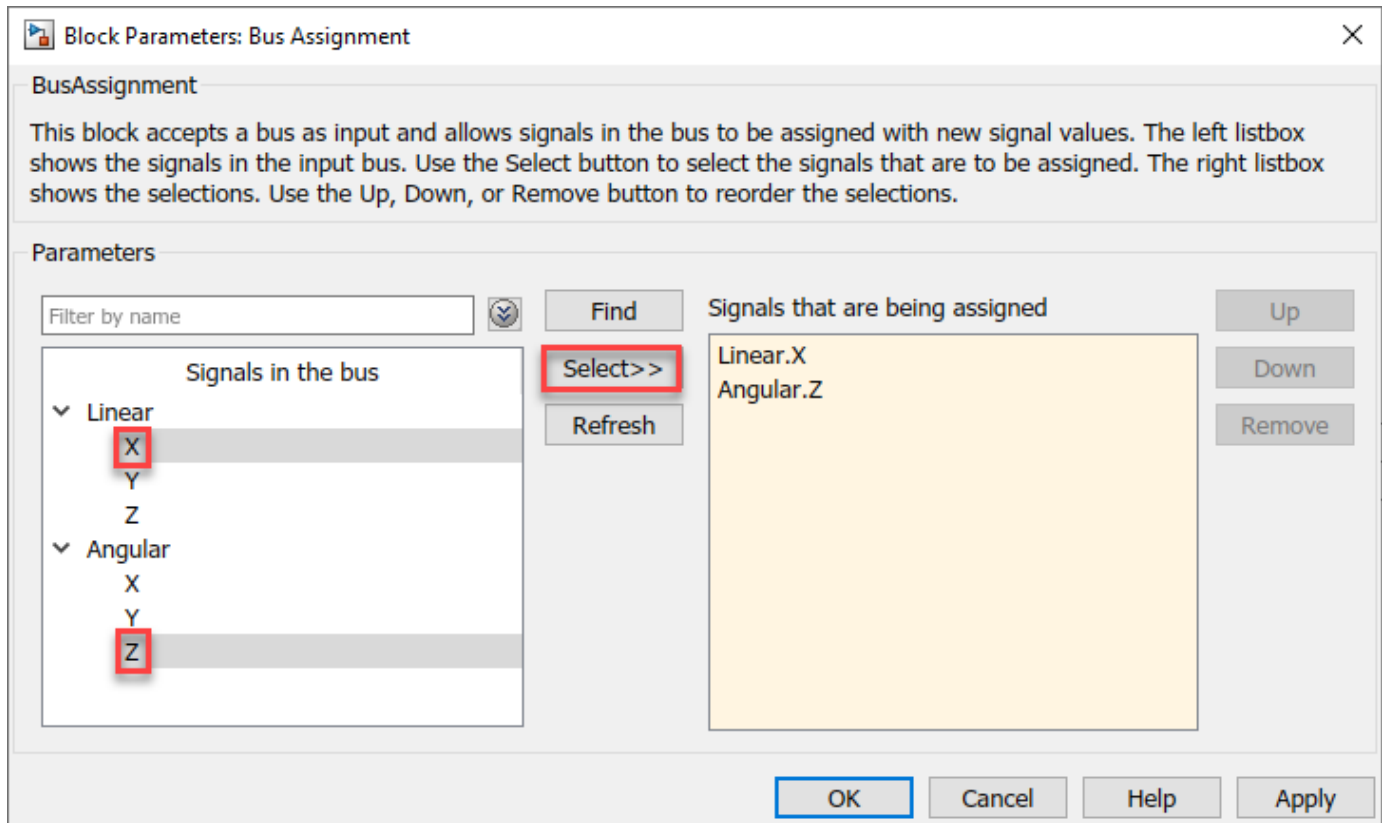
- Add a **Constant** block, a **Gain** block, and two **Slider Gain** blocks. Connect them together as shown in the figure, and set the **Gain** value to `-1.`
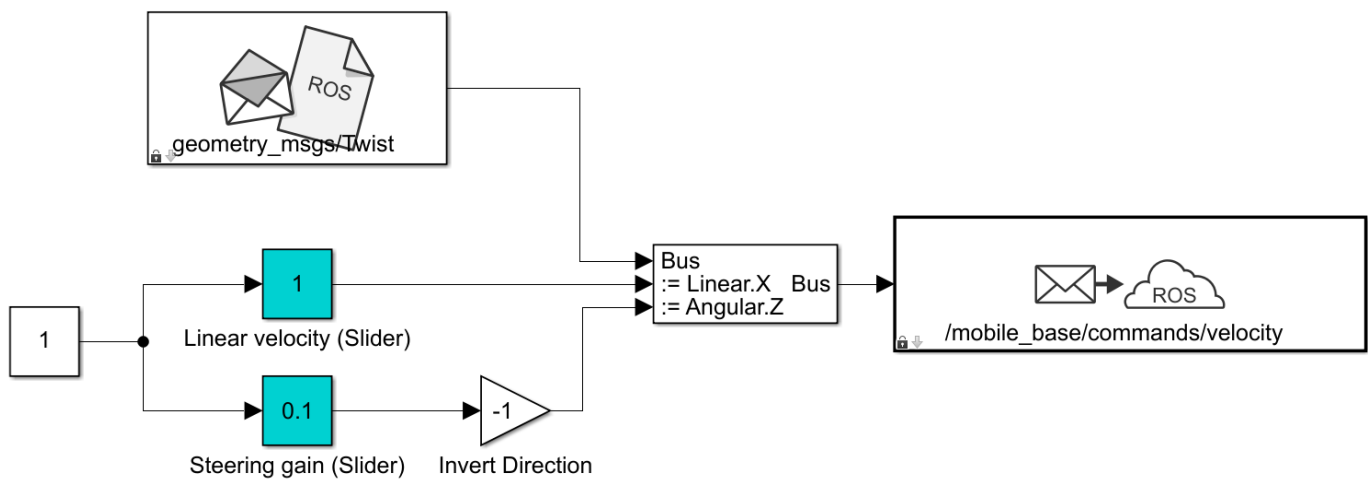
- Set the limits and current parameters of the linear velocity slider to `0.0` to `2.0`, and `1.0` respectively. Set the corresponding parameters of the steering gain slider to `-1.0` to `1.0`, and `0.1`.

Receive Location Information from the Robot

Create a subscriber to receive messages sent to the `/odom` topic. Extract the location of the robot and plot it's path in the XY-plane.

Configure a Subscriber block

1 From the **ROS Toolbox > ROS** tab in the Library Browser, drag a **Subscribe** block to the model. Double-click the block.

2 Set **Topic source** to **Select From ROS network**, and click **Select** next to the **Topic** box. Select "/odom" for the topic and click **OK**. Note that the message type `nav_msgs/Odometry` is set automatically.

Read Message Output

1 From the **Simulink > Signal Routing** tab in the Library Browser, drag a **Bus Selector** block to the model.

2 Connect the output port of the **Subscribe** block to the input port of the **Bus Selector** block. In the **Modeling** tab, click **Update Model** to ensure that the bus information is correctly propagated.

3 Double-click the **Bus Selector** block. Select `??? signal1` and `??? signal2` in the right listbox and click **Remove**. In the left listbox, expand **Pose > Pose > Position** and select **X** and **Y**. Click **Select** and then **OK**.

4 From the **Simulink > Sinks** tab in the Library Browser, drag an **XY Graph** block to the model. Connect the X and Y output ports of the **Bus Selector** block to the input ports of the **XY Graph** block.
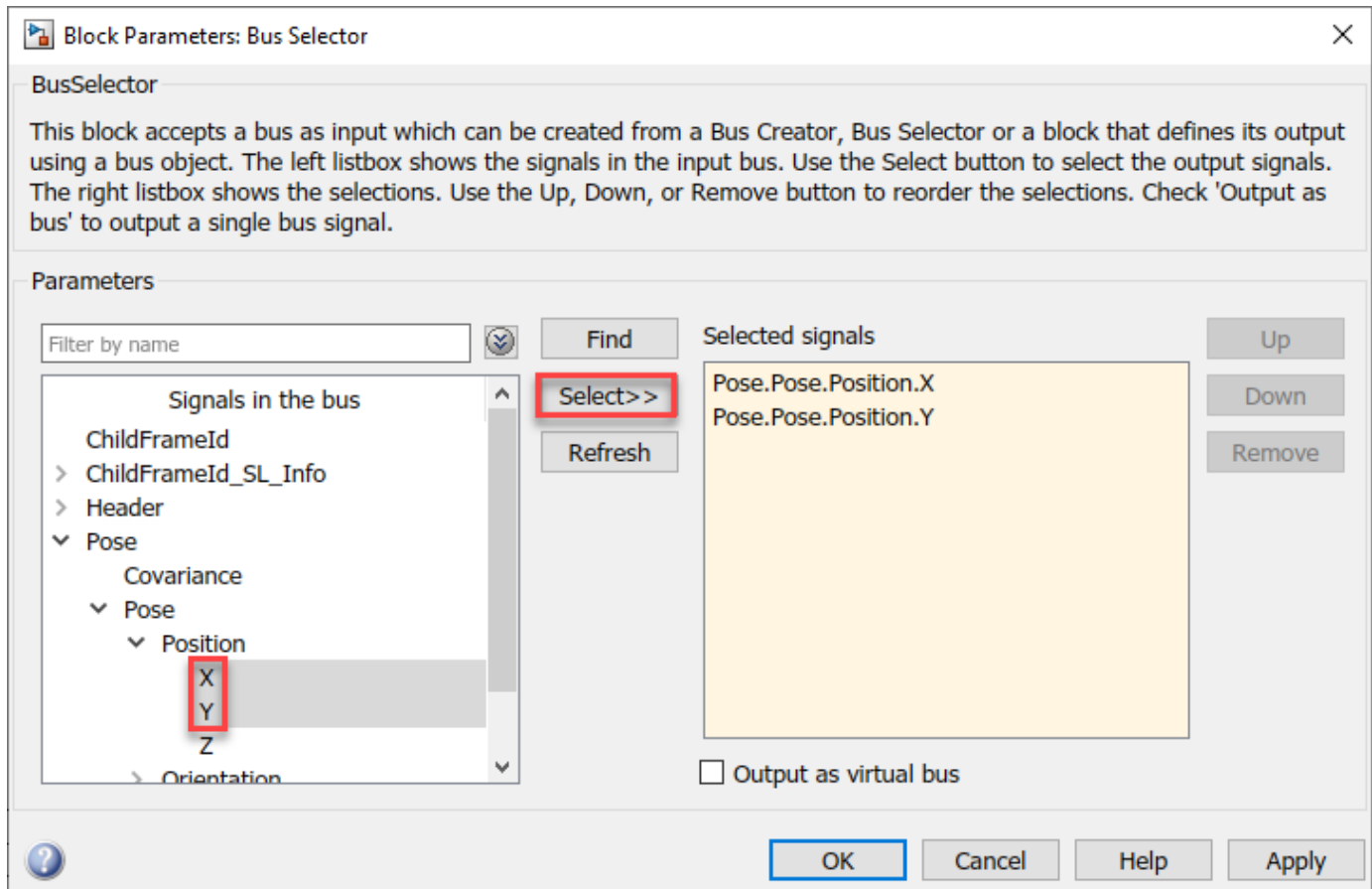
This figure shows the completed model. A pre-configured model is included for your convenience.

- **Note:** The Publisher block in this model uses the `/mobile_base/commands/velocity` topic for use with MATLAB simulator option. For Gazebo simulator option, select the `/cmd_vel` topic as shown above on page 1-0 .
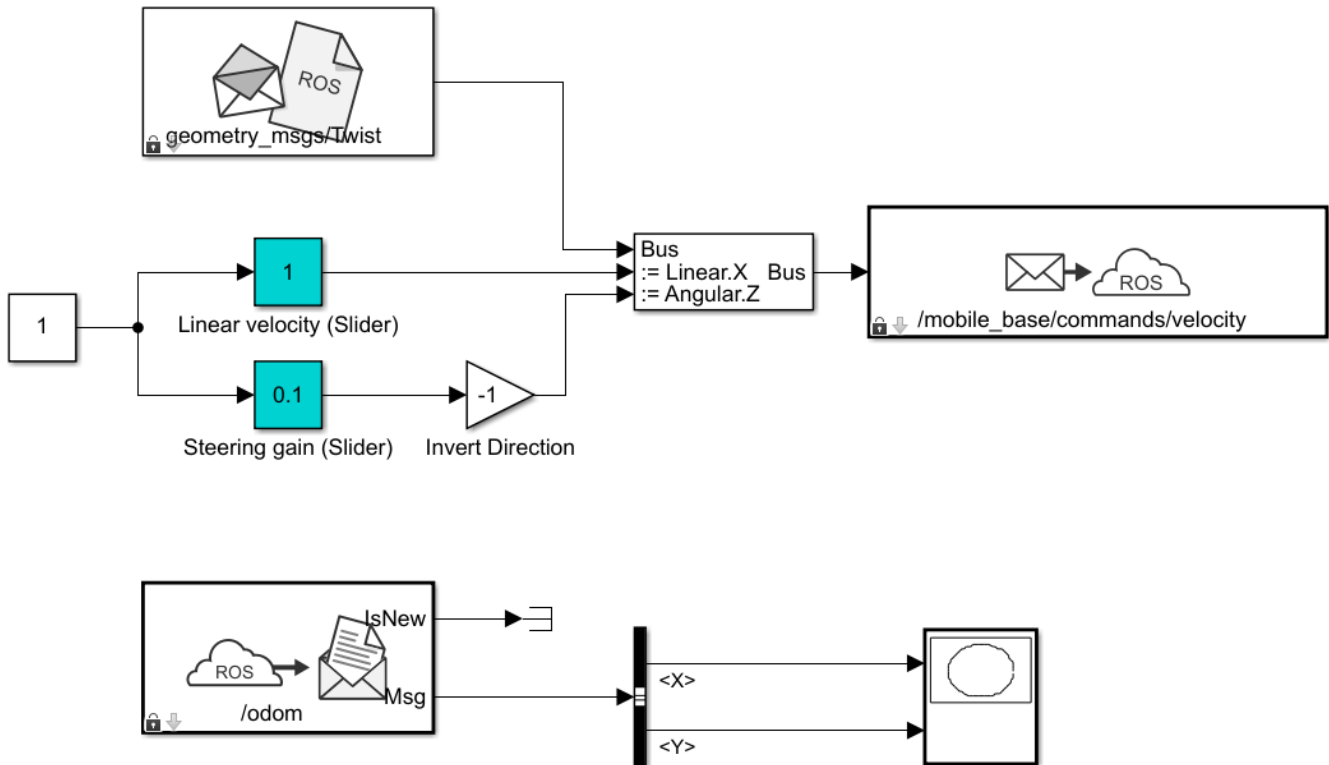
Configure and Run the Model

1. From the **Modeling** tab, click **Model Settings**. In the **Solver** pane, set **Type** to **Fixed-step** and **Fixed-step size** to `0.01`.

2. Set simulation Stop time to `Inf`.

3. Click **Run** to start the simulation.

4. In both the simulator and XY plot, you should see the robot moving in a circle.

5. While the simulation is running, change the values of **Slider Gain** blocks to control the robot. Double-click the **XY Graph** block and change the X and Y axis limits if needed. (You can do this while the simulation is running.)

6. To stop the simulation, click **Stop**.

Feedback Control of a ROS-Enabled Robot

Use Simulink® to control a simulated robot running in a separate ROS-based simulator.

This example involves a model that implements a simple closed-loop proportional controller. The controller receives location information from a simulated robot (running in a separate ROS-based simulator) and sends velocity commands to drive the robot to a specified location. Adjust parameters while the model is running and observe the effect on the simulated robot.

The following figure summarizes the interaction between Simulink and the robot simulator (the arrows in the figure indicate ROS message transmission). The /odom topic conveys location information, and the /mobile_base/commands/velocity topic conveys velocity commands.
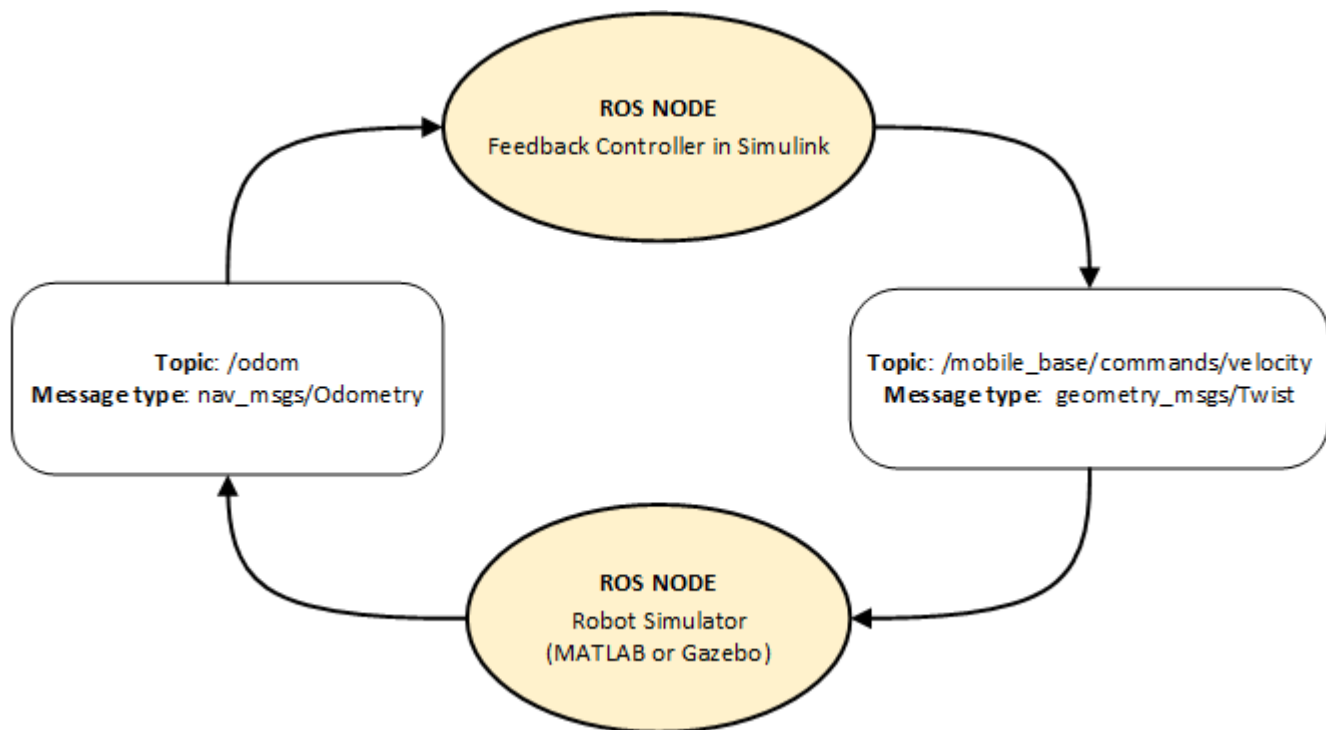
Start a Robot Simulator and Configure Simulink

Follow the steps in the "Connect to a ROS-enabled Robot from Simulink®" on page 1-81 example to do the following:

- Start a MATLAB® or Gazebo® robot simulator.
- Configure Simulink to connect to the ROS network.

Open Existing Model

After connecting to the ROS network, open the example model.

```
open_system('robotROSFeedbackControlExample.slx');
```

The model implements a proportional controller for a differential-drive mobile robot. At each time step, the algorithm orients the robot toward the desired location and drives it forward. Once the desired location is reached, the algorithm stops the robot.

```
open_system('robotROSFeedbackControlExample/Proportional Controller');
```

Note that there are four tunable parameters in the model (indicated by colored blocks).

- Desired Position (at top level of model): The desired location in (X,Y) coordinates
- Distance Threshold: The robot stops if it is closer than this distance from the desired location
- Linear Velocity: The forward linear velocity of the robot
- Gain: The proportional gain when correcting the robot orientation

The model also has a **Simulation Rate Control** block (at top level of model). This block ensures that the simulation update intervals follow wall-clock elapsed time.

Run the Model

Run the model and observe the behavior of the robot in the robot simulator.

- Position windows on your screen so that you can observe both the Simulink model and the robot simulator.
- Click **Play** to start simulation.
- While the simulation is running, double-click the **Desired Position** block and change the Constant value to [2 3]. Observe that the robot changes its heading.
- While the simulation is running, open the **Proportional Controller** subsystem and double-click the **Linear Velocity (slider)** block. Move the slider to 2. Observe the increase in robot velocity.
- Click **Stop** to end the simulation.

Observe Rate of Incoming Messages

Use the MATLAB-based simulator to observe the timing and rate of incoming messages.

- Close any existing Robot Simulator figure windows.
- Click **Play** to start simulation.
- Open the **Scope** block. Observe that the IsNew output of the **Subscribe** block is always 0, indicating that no messages are being received for the /odom topic. The horizontal axis of the plot indicates simulation time in seconds.
- At the MATLAB command line, type ExampleHelperSimulinkRobotROS to start the MATLAB-based robot simulator. This simulator publishes /odom messages at approximately 20 Hz in wall-clock elapsed time.
- In the Scope display, observe that the **IsNew** output has the value 1 at an approximate rate of 20 times per second, in elapsed wall-clock time.

The synchronization with wall-clock time is due to the **Simulation Rate Control** block. Typically, a Simulink simulation executes in a free-running loop whose speed depends on complexity of the model and computer speed (see "Simulation Loop Phase" (Simulink)). The **Simulation Rate Control** block attempts to regulate Simulink execution so that each update takes `0.02` seconds in wall-clock time when possible. (This is equal to the fundamental sample time of the model.) See the comments inside the block for more information.

In addition, the Enabled subsystems for the **Proportional Controller** and the **Command Velocity Publisher** ensure that the model only reacts to genuinely new messages. If enabled subsystems were not used, the model would repeatedly process the same (most-recently received) message repeatedly, leading to wasteful processing and redundant publishing of command messages.

Generate a Standalone ROS Node from Simulink®

This example shows you how to generate and build a standalone ROS node from a Simulink model.

Introduction

In this example, you configure a model to generate C++ code for a standalone ROS node. You then build and run the ROS node on an Ubuntu® Linux® system.

Prerequisites

*   This example requires Simulink Coder™ and Embedded Coder™.
*   A Ubuntu Linux system with ROS and an SSH server installed 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" on page 1-102 for instructions).
*   Review the "Feedback Control of a ROS-Enabled Robot" on page 1-89 example.

Configure a Model for Code Generation

Configure a model to generate C++ code for a standalone ROS node. The model is the proportional controller introduced in the "Feedback Control of a ROS-Enabled Robot" on page 1-89 example.

*   Open *RobotController.slx*. Click the link or run `RobotController` in the Command Window.
*   In the toolstrip, click **Modeling > Model Settings.**
*   In the **Hardware Implementation** pane of the Configuration Parameters dialog, set **Hardware board** to **Robot Operating System (ROS)**, and click Apply. The **Hardware board settings** section contains settings specific to the generated ROS package, such as information to be included in the `package.xml` file. Change **Maintainer name** to `ROS Example User`.
*   The model uses variable-sized arrays. To enable code-generation of variable-sized signals, check **variable-size signals** under **Code Generation > Interface > Software environment**. If the **variable-size signals** option is not visible, check the option, **Use Embedded Coder Features** in **Hardware Implementation > Advanced parameters.**
*   In the **Solver** pane of the Configuration Parameters dialog, ensure that Solver **Type** is set to **Fixed-step**, and set **Fixed-step size** to `0.05`. In generated code, the Fixed-step size defines the actual time step, in seconds, that is used for the model update loop (see "Execution of Code Generated from a Model" (Simulink Coder)). It can be made smaller (e.g., 0.001 or 0.0001) but for current purposes 0.05 is sufficient.

Configure the Connection to the ROS Device

A **ROS device** is any Linux system that has ROS installed and is capable of building and running a ROS node. If you have Simulink Coder, you can generate code for a standalone ROS node. If your system is connected to a ROS device, Simulink can also transfer the generated code to the ROS device, build an executable, and run the resulting ROS node (this is referred to as "deploying" the ROS node).

In this task, you decide if you want to generate code for the ROS node or if you want to build and run it on a ROS device. If you are connected to a ROS device, you can configure Simulink to use it as a deployment target for your ROS node.
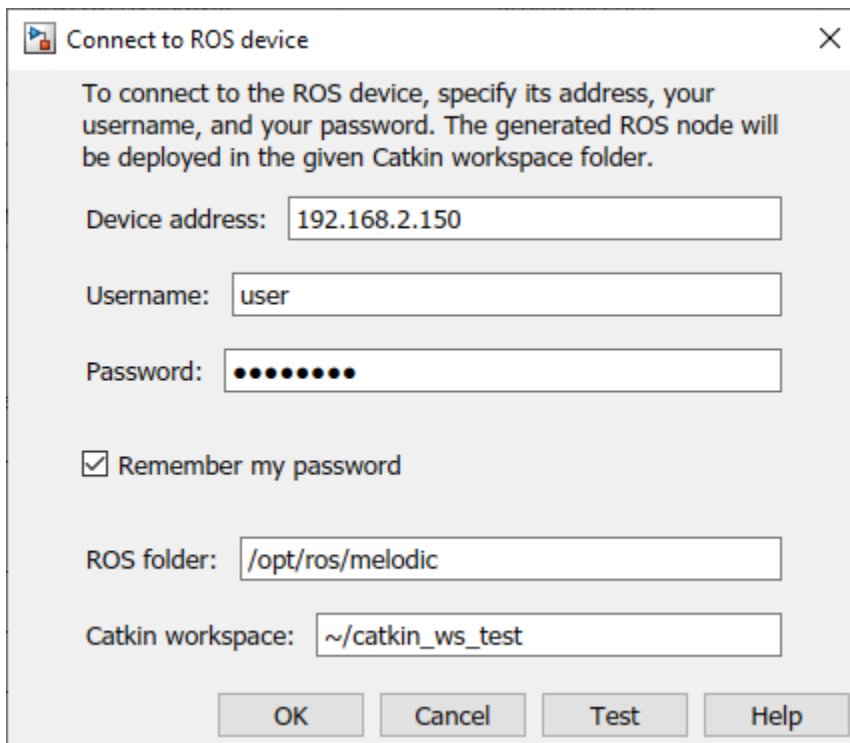
*   Under the **Modeling** tab, click **Model Settings**.

- In the **Hardware Implementation** pane of Configuration Parameters dialog, select an **Build action** under **Hardware board settings > Target hardware resources > Groups > Build Options**. The selected build action affects the behavior of Simulink when building the model. *None* (the default setting) only generates the code for the ROS node, without building it on an external ROS device. *Build and load* generates the code, transfers it to an external device and builds a ROS node executable. If you select *Build and run*, the resulting node executable is started automatically at the end of the build.
- Set the **Build action** to **Build and run.**
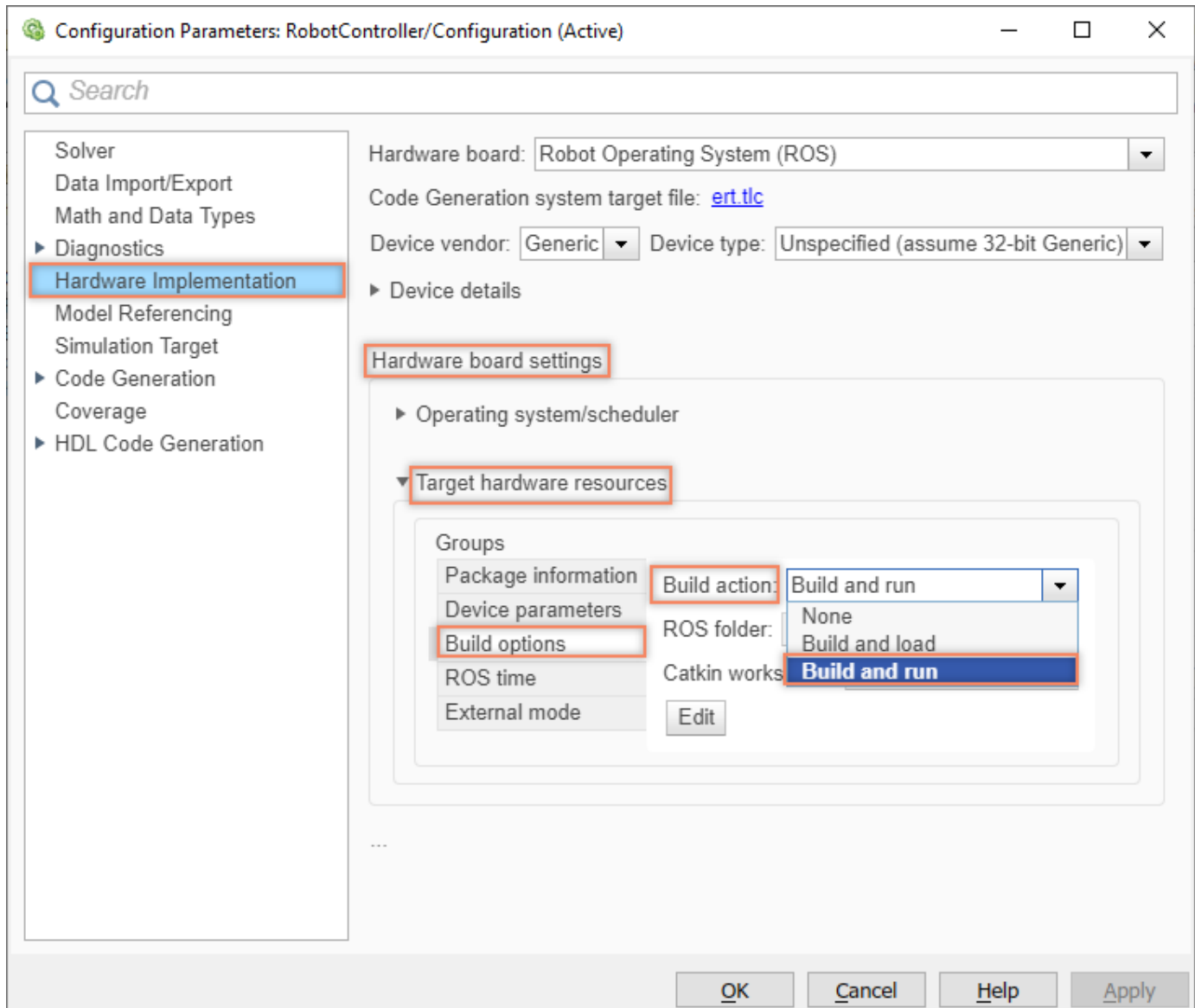
- Configure the connection to your external ROS device. Under the **Robot** tab, click **Connect to Robot** to open the **Connect to a ROS device** dialog. In this dialog, you can enter all the information that Simulink needs to deploy the ROS node. This includes the IP address or host name of your ROS device, your login credentials, and the Catkin workspace. Change **Catkin workspace** to ~/catkin_ws_test.

ROS Folder is the location of the ROS installation on the ROS device. If you do not specify this folder, the settings test (see next step) tries to determine the correct folder for you.

- If the ROS device is turned on and accessible from your computer, you can verify the connection settings by clicking **Test**. The test verifies every device setting and display warnings and errors in the Simulink Diagnostic Viewer if problems are found. If possible, the test also suggests how the problems can be fixed. Click **Test** now.

- Most likely, the Catkin workspace `~/catkin_ws_test` does not exist on the target device. The test detects this problem and suggests to create the folder and initialize the workspace. Click **Fix** to apply this action automatically. After a few seconds, you should see a green notice that the folder has been created successfully. In the following figure you can see an example of creating the folder successfully. To verify that the Catkin workspace is now available, click **Test** in the connection settings dialog again. The warning has disappeared and the Catkin workspace is ready to build your ROS node.
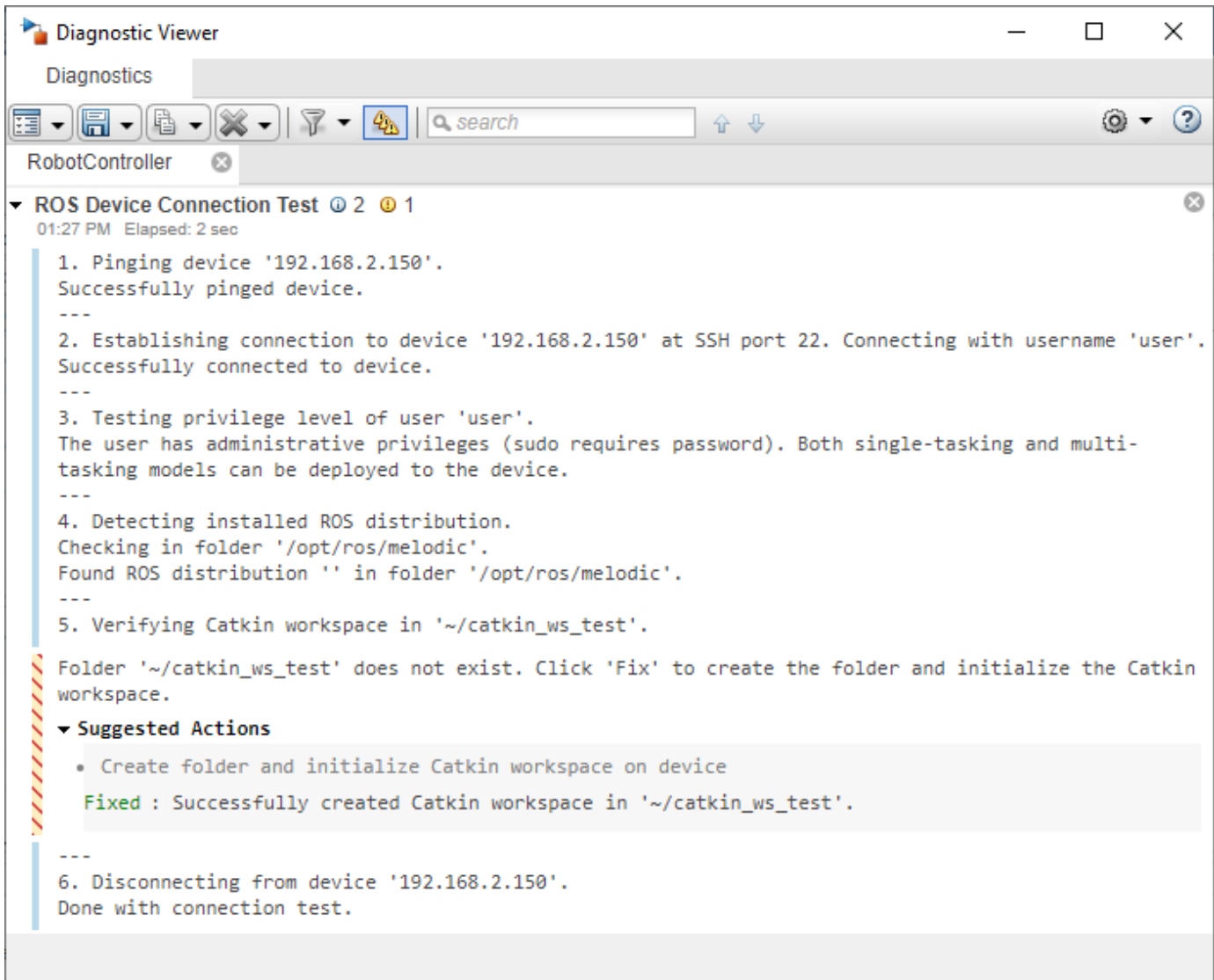
- Change the device connection settings and test them until no other warnings or errors are shown. If an automatic fix to your settings is possible, Simulink suggests it by displaying the **Fix** button. Once you have a good set of settings, click **OK** in the connection settings dialog to save the settings.

The connection settings are not specific to a single model, but apply to all ROS models in Simulink.

Generate the C++ ROS Node

Generate code for a standalone ROS node, and automatically transfer, build, and run it on the ROS device. Exercise the generated ROS node using a ROS master running on the ROS device.

1. In MATLAB®, change the current folder to a temporary location where you have write permission.

2. The code generation process first prepares the model for simulation to ensure that all blocks are properly initialized. This preparation requires a valid connection to a ROS master.

In MATLAB, you can use the `rosdevice` object to start a ROS master on the ROS device. If you provide no arguments, `rosdevice` uses the device connection settings you entered in the Simulink dialog to connect to the ROS device.

```
d = rosdevice
runCore(d);
```

```
d =

  rosdevice with properties:

        DeviceAddress: '192.168.2.150'
             Username: 'user'
            ROSFolder: '/opt/ros/melodic'
      CatkinWorkspace: '~/catkin_ws_test'
       AvailableNodes: {0×1 cell}
```

3. Use `rosinit` to connect MATLAB to the ROS master running on the ROS device:

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

4. Tell Simulink to use the same ROS connection settings as MATLAB. Under the **Simulation** tab, select **ROS Toolbox > ROS Network**. Set the ROS Master (ROS 1) and Node Host network addresses to **Default**.
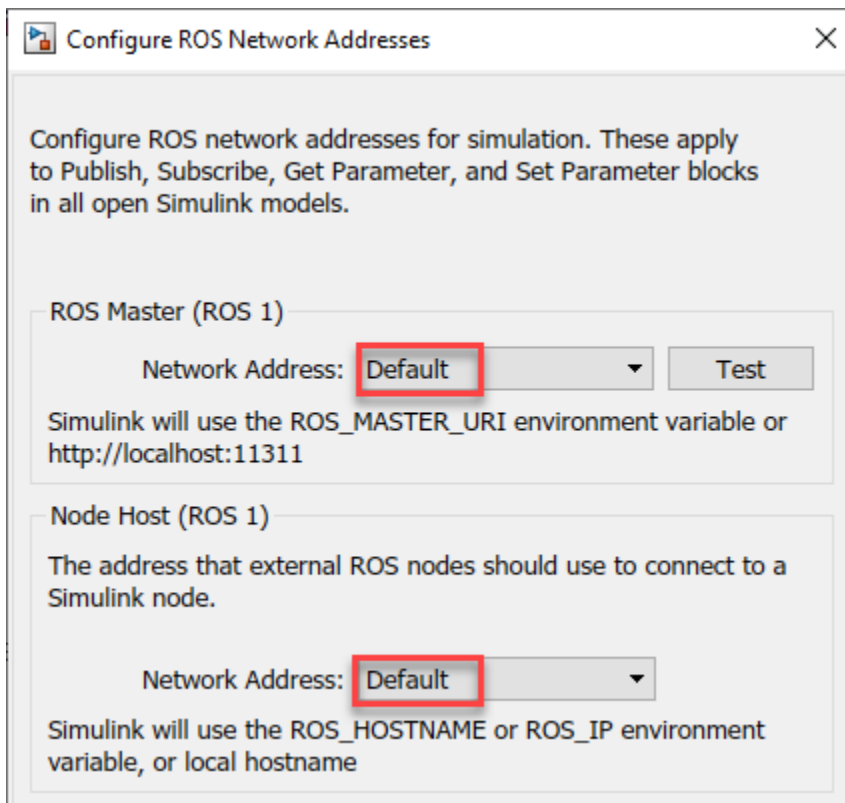
You only have to execute steps 2 - 4 once per MATLAB session, not every time you generate a ROS node.

5. Under the Simulation tab, click **Build & Run**. If you get any errors about bus type mismatch, close the model, clear all variables from the base MATLAB workspace, and re-open the model.

Click on the **View Diagnostics** link at the bottom of the model toolbar to see the output of the build process.

6. Once the code generation completes, the ROS node is transferred to the Catkin workspace on your ROS device. The node builds there and starts to run automatically.

The generated node connects to the ROS master running on the ROS device.

7. Use `rosnode` to list all running nodes is the ROS network. "robotcontroller" should be in the displayed list of nodes.

```
rosnode list
```

You can use `rostopic` to verify that the deployed node publishes data on the ROS topic to control the robot motion:

```
rostopic info /mobile_base/commands/velocity
```

Run and Verify the ROS Node

Run the newly-built ROS node and verify its behavior using a MATLAB-based robot simulator.

1. In MATLAB, type `ExampleHelperSimulinkRobotROS` to start the Robot Simulator. The simulator automatically connects to the ROS master running on the ROS device. If you want to connect to a Gazebo-based robot simulation, see "Connect to a ROS-enabled Robot from Simulink®" on page 1-81.

```
sim = ExampleHelperSimulinkRobotROS
```
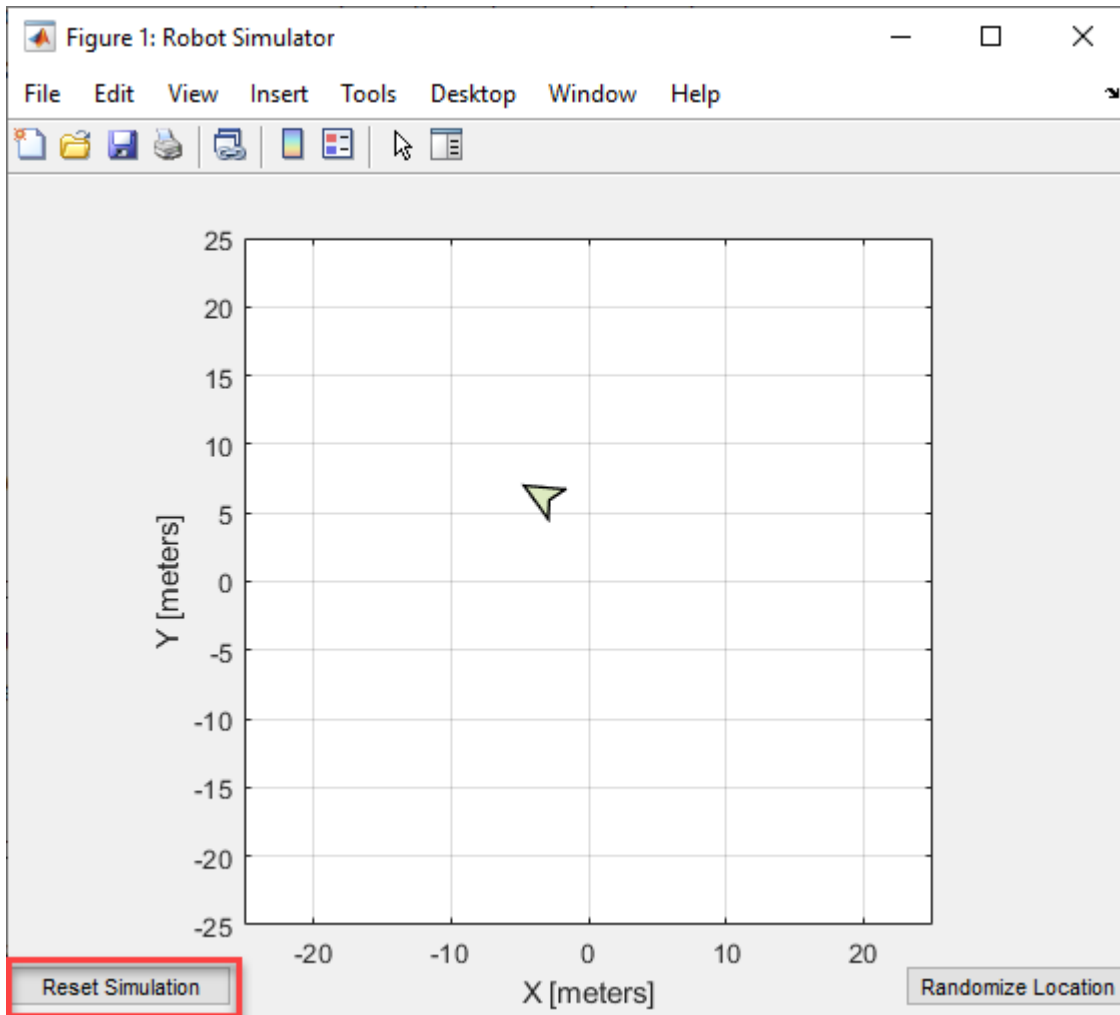
2. Verify that the simulated robot moves toward the goal (the **Desired Position** constant specified in the model). The robot stops once it reaches the goal [-10, 10].

3. Click **Reset Simulation** to reset the robot's position to [0, 0]. The robot starts to move immediately towards the goal position.

4. In MATLAB, you can manage ROS nodes generated by Simulink with the `rosdevice` object. Once a Simulink model is deployed, you can use `rosdevice` to run and stop the node at any point, without having to rebuild it in Simulink.

The `AvailableNodes` property shows the deployed robotcontroller node. You can verify that the node is running by calling the `isNodeRunning` function.

```
d = rosdevice
```

```
d =

  rosdevice with properties:

       DeviceAddress: '192.168.2.150'
            Username: 'user'
           ROSFolder: '/opt/ros/melodic'
    CatkinWorkspace: '~/catkin_ws_test'
     AvailableNodes: {'robotcontroller'}
```

```
isNodeRunning(d, 'robotcontroller')
```

5. Stop the ROS node from running.

```
stopNode(d, 'robotcontroller')
isNodeRunning(d, 'robotcontroller')
```

6. Click the **Reset Simulation** button in the simulation window. The robot stays at location [0,0] and does not move.

- Now restart the node.

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

- The robot should start moving towards the goal position again.

7. Once you are done verifying, you can clean up the system state as follows.

- Stop the node running on the target device

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

- On the host computer, close the *Robot Simulator* figure window and type `rosshutdown` at the MATLAB command line.

```
rosshutdown
```

Advanced Topics and Troubleshooting

Specify ROS network settings in Simulink: By default, Simulink uses the ROS connection settings from `rosinit` in MATLAB. To override these settings, specify ROS connection settings in Simulink. Under the **Simulation** tab, select **ROS Toolbox > ROS Network** and set the ROS Master and Node Host network addresses:
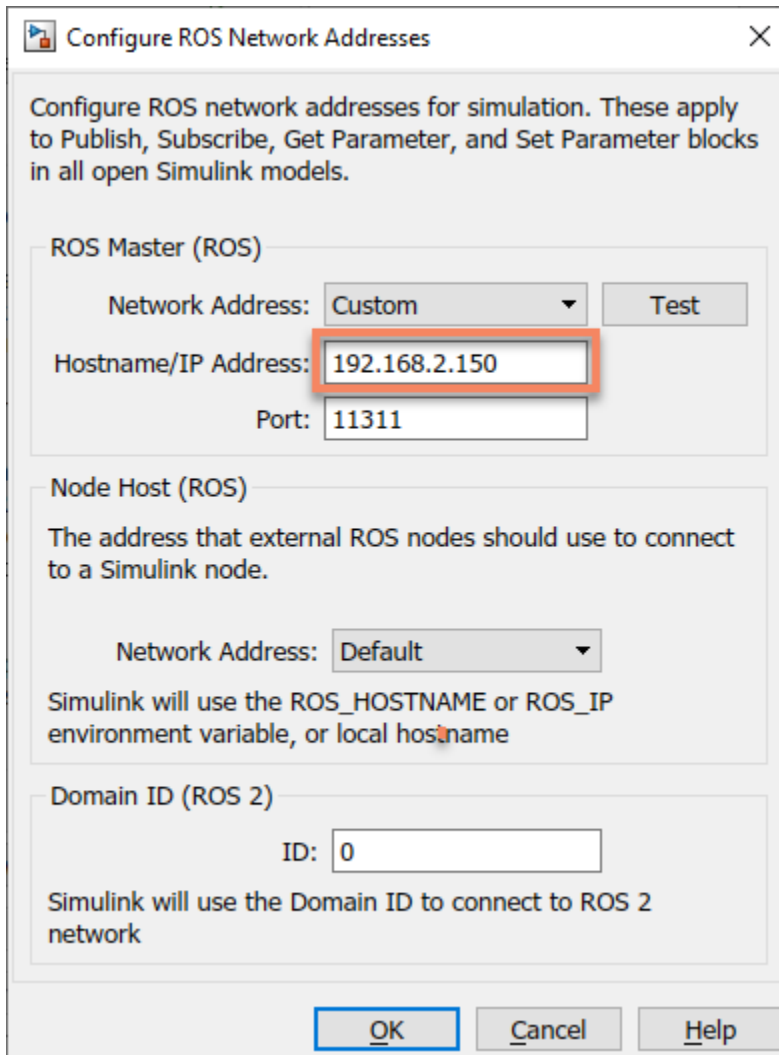
Generated C++ code archive: No matter what **Build action** you select (None, Build and load, Build and run), Simulink always generates two files in your current folder: an archive containing the C++ source code (RobotController.tgz in our example) and a shell script for extracting and building the C++ code manually (build_ros_model.sh). If your MATLAB computer is not connected to the ROS device, you can transfer the files manually and build them there.
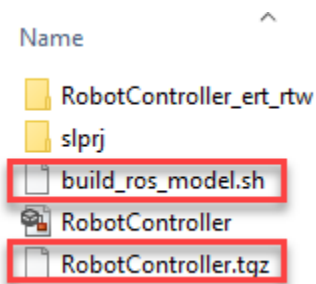
Processor-specific generated code: If you use blocks from other products (such as Computer Vision System Toolbox™), the generated code may include processor-specific optimizations that lead

to compilation problems when building the ROS node on Linux. In these cases, you need to let Simulink know the platform on which the generated code is compiled. You can do this through the **Hardware Implementation** pane of the Model Configuration Parameters dialog.

Running ROS Master in MATLAB: In the example above, you connected to a ROS master running on the ROS device. Alternatively, you can create a ROS master in MATLAB. Use `rosinit` at the MATLAB command line:

```
rosinit('NodeHost', <IP address of your computer>)
```

For example, if the IP address of your host computer is 172.28.194.92, use the following command:

```
rosinit('NodeHost', '172.28.194.92')
```

The `NodeHost` setting is important to ensure that the generated ROS node is able to communicate to the master on MATLAB. **Note**: The generated ROS node will use the `NodeHost` IP address to communicate to the global ROS node in MATLAB, so ensure that the specified IP address is accessible from the ROS device (for example, using `ping`). See the "Connect to a ROS Network" on page 1-7 example for more details on the significance of the `NodeHost` setting.

Tasking mode: Simulink can generate code for either multi-tasking or single-tasking modes (see "Time-Based Scheduling and Code Generation" (Simulink Coder)). By default, generated ROS code uses single-tasking mode (a single thread for all the rates) without real-time scheduling. This allows the generated ROS code to execute without `sudo` privileges, but can lead to less predictable performance.

If you require more predictable performance, you can configure the model to use multi-tasking. In the **Solver** pane of the Configuration Parameters dialog enable **Treat each discrete rate as a separate task** to enable multi-tasking. In generated code, this creates a separate thread for each rate in the model and uses prioritized scheduling for the threads.

To run the ROS node, you need to have administrative privileges on the ROS device. Simulink automatically detects if your privileges are insufficient when the model is deployed to the target device.

See Also

Related Examples
*   "Generate Code to Manually Deploy a ROS Node from Simulink" on page 4-26

Get Started with Gazebo and a Simulated TurtleBot

This example shows how to set up the Gazebo® simulator engine. This example prepares you for further exploration with Gazebo and also for exploration with a simulated TurtleBot®.

Gazebo is a simulator that allows you to test and experiment realistically with physical scenarios. Gazebo is a useful tool in robotics because it allows you to create and run experiments rapidly with solid physics and good graphics. MATLAB® connects to Gazebo through the ROS interface.

Download Virtual Machine

You can download a virtual machine image that already has ROS and Gazebo installed. This virtual machine is based on Ubuntu® Linux® and is pre-configured to support the examples in ROS Toolbox™.

- Download and install the ROS Virtual Machine

- Launch the virtual machine

- On the Ubuntu desktop you see multiple Gazebo world start-up scripts, as well as other utility shortcuts. For the TurtleBot® examples, use "Gazebo Empty," "Gazebo House," "Gazebo Office," or "Gazebo Sign Follower ROS."

- Click **Gazebo House**. A world opens.
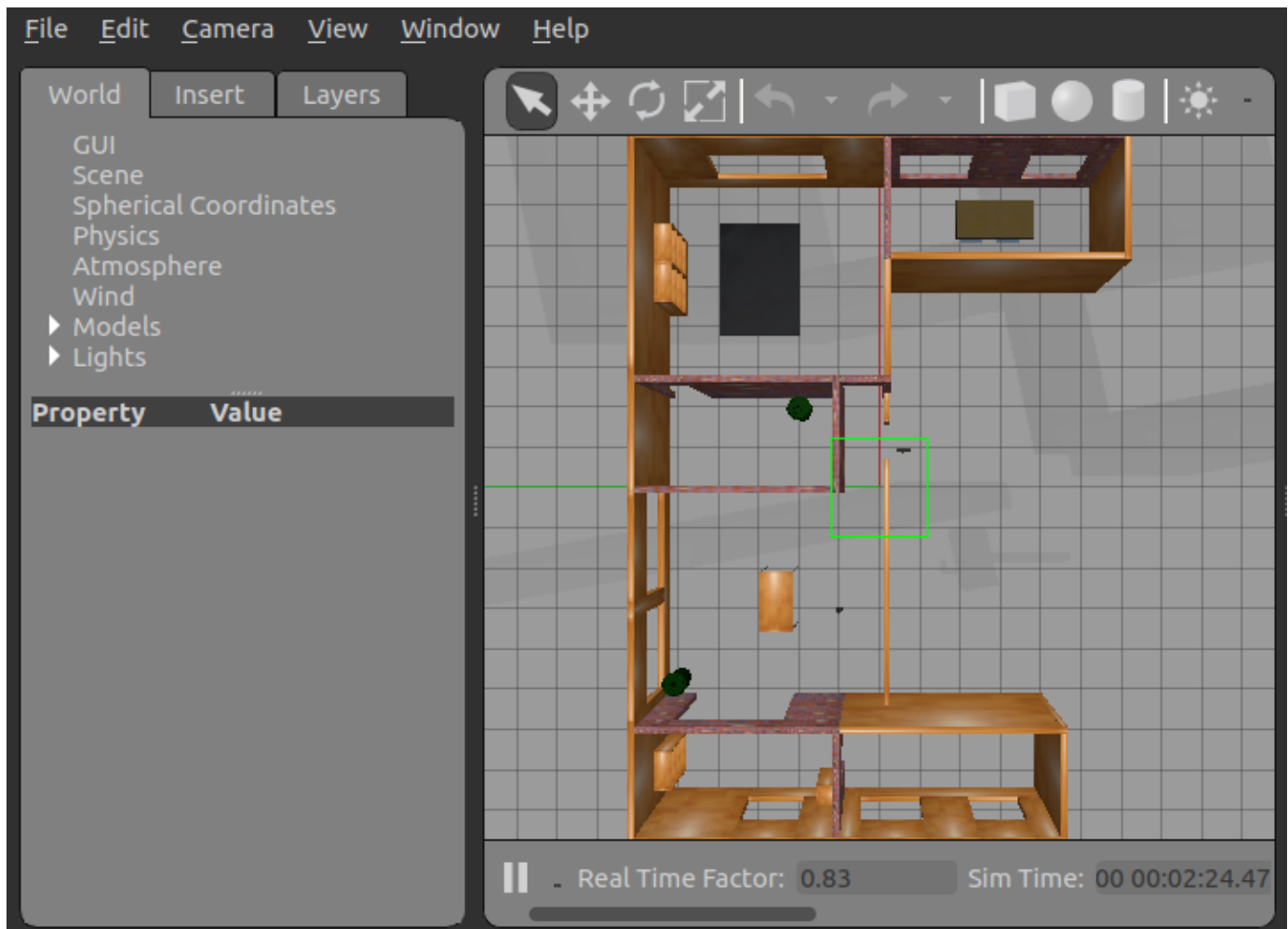
Note: If the Gazebo screen looks entirely black, refresh the image by minimizing it and then maximizing it.

- Open a new terminal in the Ubuntu virtual machine.
- Type `ifconfig` and return to see the networking information for the virtual machine.
- Under `eth0`, the `inet addr` displays the IP address for the virtual machine.

```
user@ubuntu:~$ ifconfig
ens33: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 192.168.178.128  netmask 255.255.255.0  broadcast 192.168.178.255
        inet6 fe80::7051:bc83:3912:9af6  prefixlen 64  scopeid 0x20<link>
        ether 00:0c:29:48:fb:90  txqueuelen 1000  (Ethernet)
        RX packets 3656  bytes 2862062 (2.8 MB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 5541  bytes 1017765 (1.0 MB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING>  mtu 65536
        inet 127.0.0.1  netmask 255.0.0.0
        inet6 ::1  prefixlen 128  scopeid 0x10<host>
        loop  txqueuelen 1000  (Local Loopback)
        RX packets 894775  bytes 67740927 (67.7 MB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 894775  bytes 67740927 (67.7 MB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

user@ubuntu:~$ █
```

- Two ROS environment variables must be set to set up the network: ROS_MASTER_URI and ROS_IP. If you are using the demos from the desktop of the Linux® virtual machine, these variables are usually automatically set at startup.

- **(Optional) If you are using your own virtual machine** set up the variables by executing the following commands in the terminal. **Replace `IP_OF_VM` with the IP address retrieved through `ifconfig`):**

```
echo export ROS_MASTER_URI=http://IP_OF_VM:11311 >> ~/.bashrc
echo export ROS_IP=IP_OF_VM >> ~/.bashrc
```

- Check the environment variables using `echo $ENV_VAR` (replacing ENV_VAR with the appropriate environment variable). You can close and reopen your terminal for it to take effect.

```
user@ubuntu:~$ echo $ROS_MASTER_URI
http://192.168.178.128:11311
user@ubuntu:~$ echo $ROS_IP
192.168.178.128
user@ubuntu:~$
```

- The following diagram illustrates correct environment variable assignments (with fake IP addresses)
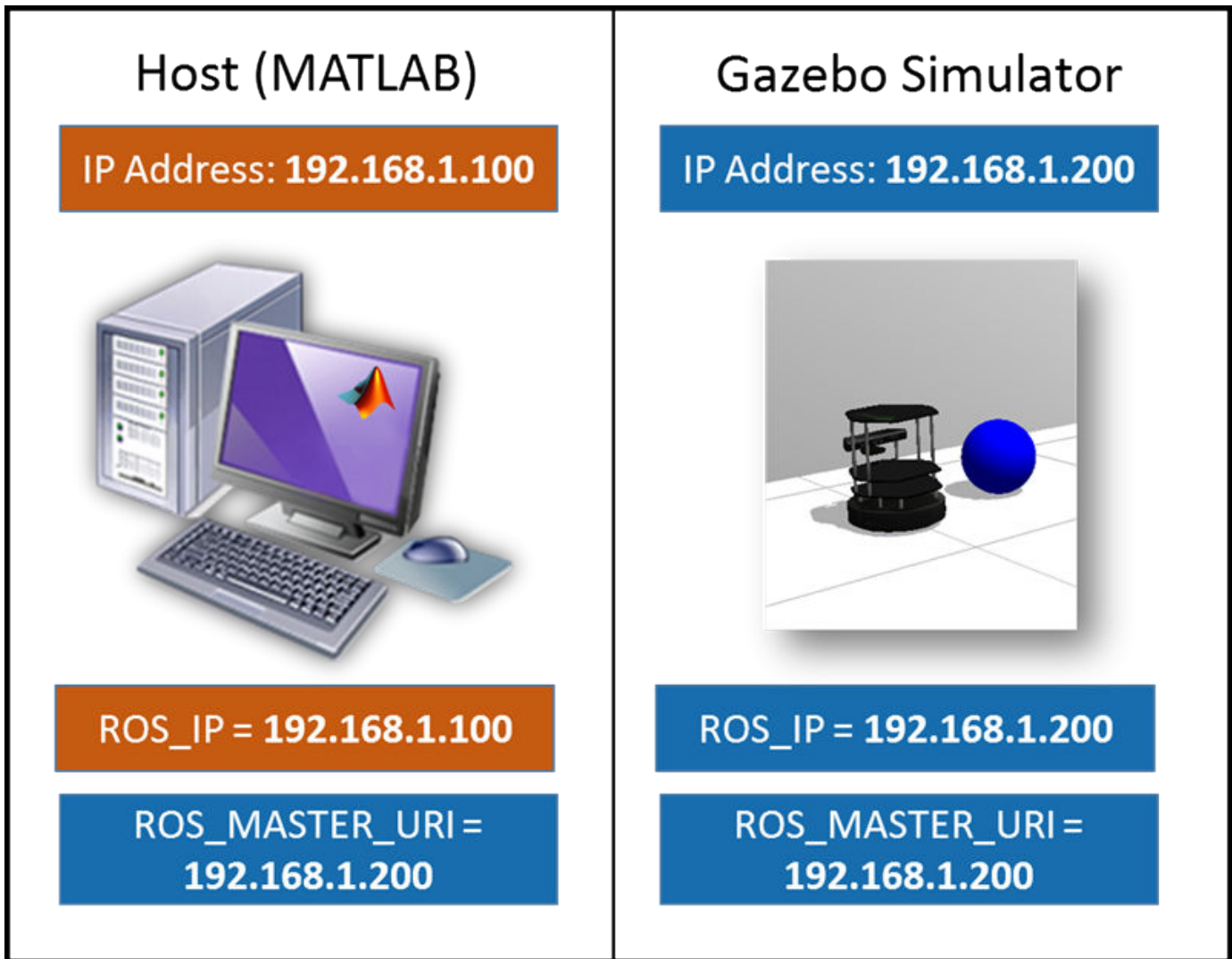
Connect to an Existing Gazebo Simulator

If you already have Gazebo running on a Linux distribution, set up the simulator as described here:

- On the ROS website, download the appropriate packages for TurtleBot.
- Follow the instructions on the ROS website to get the TurtleBot running in a simulated Gazebo environment.
- Make sure the environment variables are appropriately set and that you can ping back and forth between your host computer and the Gazebo computer. There are many ways to set up the network. The "Connect to a ROS Network" on page 1-7 example contains tips on how to verify connectivity between devices in the ROS network.
- To use any ROS commands in the Linux machine terminals, the terminal environment needs to be set to use the proper ROS installation. Source the appropriate ROS environment setup script in the terminal before running any ROS commands. In the VM, the command is: `source /opt/ros/melodic/setup.bash`
- Make sure you have access to the following topics. In the terminal on the Linux machine, enter `rostopic list` to see the at least these available topics.

```
/clock
/cmd_vel
/imu
/odom
/scan
/tf
```

Host Computer Setup

- Find the IP address of your host computer on the network. On a Windows® machine, at the command prompt, type `ipconfig`. On a Mac or Linux machine, open a terminal and type `ifconfig`. An example of `ipconfig` is shown.

```
C:\>ipconfig

Windows IP Configuration


Ethernet adapter Ethernet:

   Connection-specific DNS Suffix  . : mathworks.com
   Link-local IPv6 Address . . . . . : fe80::dc58:c7e2:a4bf:be19%10
   IPv4 Address. . . . . . . . . . . : 172.21.17.16
   Subnet Mask . . . . . . . . . . . : 255.255.255.0
   Default Gateway . . . . . . . . . : 172.21.17.1

Ethernet adapter VMware Network Adapter VMnet1:

   Connection-specific DNS Suffix  . :
   Link-local IPv6 Address . . . . . : fe80::24b3:afbc:d3f1:e969%13
   IPv4 Address. . . . . . . . . . . : 192.168.40.1
   Subnet Mask . . . . . . . . . . . : 255.255.255.0
   Default Gateway . . . . . . . . . :

Ethernet adapter VMware Network Adapter VMnet8:

   Connection-specific DNS Suffix  . :
   Link-local IPv6 Address . . . . . : fe80::69bc:103a:8a85:76c4%4
   IPv4 Address. . . . . . . . . . . : 192.168.178.1
   Subnet Mask . . . . . . . . . . . : 255.255.255.0
   Default Gateway . . . . . . . . . :
```

Note: The connection type can vary depending on how you are connected to the laptop. In this case you use the Ethernet, however, in many cases the wireless (wlan) is the appropriate connection.

- Ping the simulator machine `ping IP_OF_VM`. A successful `ping` is shown first, followed by an unsuccessful `ping`.

```
C:\>ping 192.168.178.128

Pinging 192.168.178.128 with 32 bytes of data:
Reply from 192.168.178.128: bytes=32 time<1ms TTL=64
Reply from 192.168.178.128: bytes=32 time<1ms TTL=64
Reply from 192.168.178.128: bytes=32 time<1ms TTL=64
Reply from 192.168.178.128: bytes=32 time<1ms TTL=64

Ping statistics for 192.168.178.128:
    Packets: Sent = 4, Received = 4, Lost = 0 (0% loss),
Approximate round trip times in milli-seconds:
    Minimum = 0ms, Maximum = 0ms, Average = 0ms

C:\>ping 192.168.178.130

Pinging 192.168.178.130 with 32 bytes of data:
Reply from 192.168.178.1: Destination host unreachable.
Request timed out.
Request timed out.
Request timed out.

Ping statistics for 192.168.178.130:
    Packets: Sent = 4, Received = 1, Lost = 3 (75% loss),
```

Next Steps

- For more Gazebo examples, refer to: "Pick-and-Place Workflow in Gazebo using ROS" (Robotics System Toolbox)
- For TurtleBot examples, refer to: "Communicate with the TurtleBot" on page 1-130

Add, Build, and Remove Objects in Gazebo

This example explores more in-depth interaction with the Gazebo® Simulator from MATLAB®. Topics include creating simple models, adding links and joints to models, connecting models together, and applying forces to bodies.

Prerequisites: "Get Started with Gazebo and a Simulated TurtleBot" on page 1-102

Connect to Gazebo®

On your Linux® machine, start Gazebo. If you are using the virtual machine from "Get Started with Gazebo and a Simulated TurtleBot" on page 1-102, start the **Gazebo Empty** world from the desktop.

Initialize ROS by replacing `ipaddress` with the IP address of the virtual machine. Create an instance of the `ExampleHelperGazeboCommunicator` class.

```
ipAddress = "http://192.168.178.132:11311";
rosinit(ipAddress)
```

```
Initializing global node /matlab_global_node_77778 with NodeURI http://192.168.178.1:52158/
```

```
gazebo = ExampleHelperGazeboCommunicator;
```

Spawn a Simple Sphere

To create a model, use the `ExampleHelperGazeboModel` class. Define properties (using `addLink`) and spawn a ball using the `spawnModel` function.

```
ball = ExampleHelperGazeboModel("Ball")

ball =
  ExampleHelperGazeboModel with properties:

        Name: 'Ball'
     ModelObj: [1×1 org.apache.xerces.dom.DocumentImpl]
        Links: []
       Joints: []
```

```
sphereLink = addLink(ball,"sphere",1,"color",[0 0 1 1])

sphereLink =
'link0'
```

```
spawnModel(gazebo,ball,[8.5 0 1])
```

All units for Gazebo commands are specified in SI units. Depending on your view, you might have to zoom out to see the ball, because it is placed at `[8.5, 0, 1]`. Here is an image of the scene:
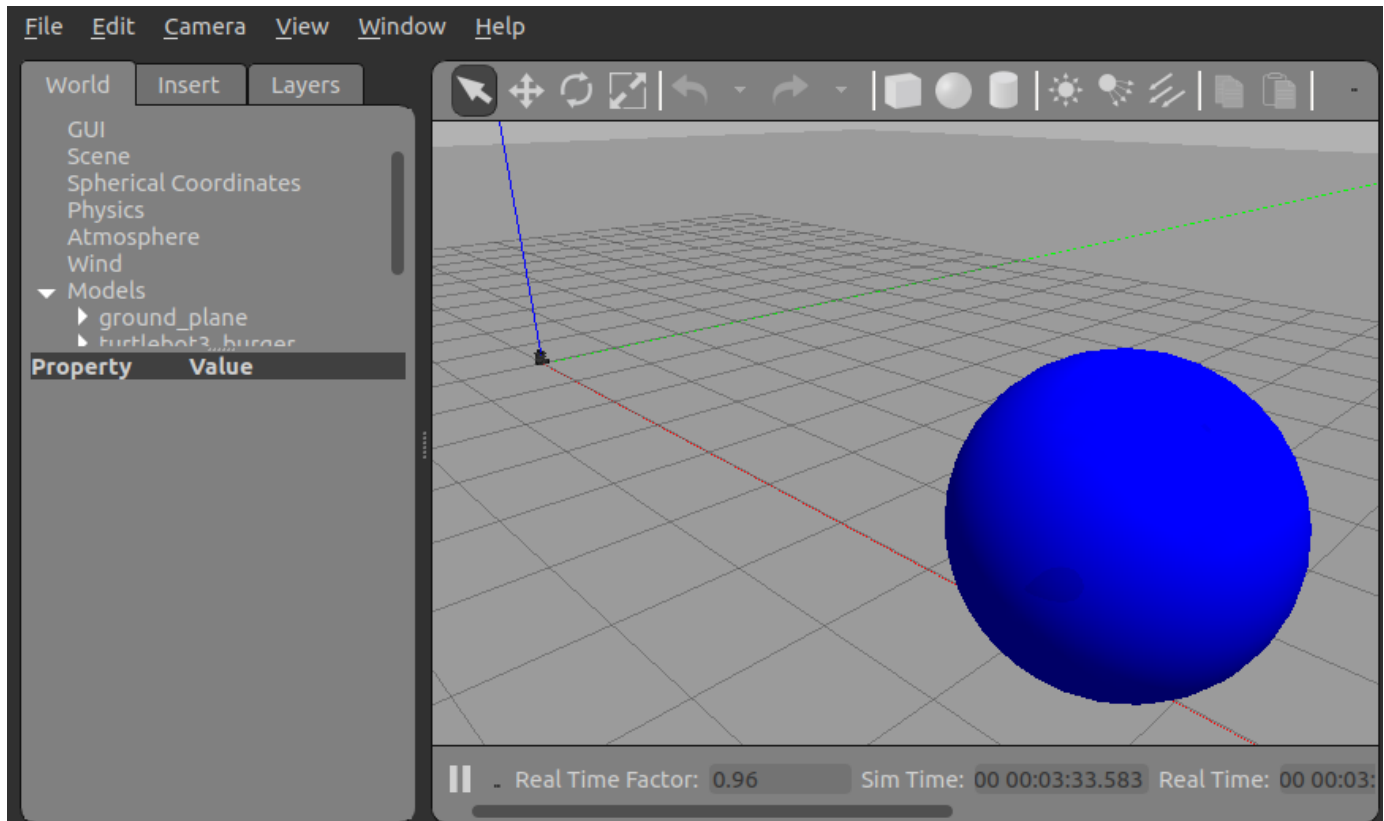
Build and Spawn Bowling Pins

Create vectors x and y for the location of the bowling pins (in meters).

```
x = [ 1.5  1.5  1.5  1.5  2.5  2.5  2.5  3.5  3.5  4.5];
y = [-1.5 -0.5  0.5  1.5 -1    0    1   -0.5  0.5  0];
```

Define a basic model for the bowling pin using the ExampleHelperGazeboModel object. Use addLink to create the cylinder and the ball.

```
pin = ExampleHelperGazeboModel("BowlPin");

link1 = addLink(pin,"cylinder",[1 0.2],"position",[0 0 0.5]);
link2 = addLink(pin,"sphere",0.2,"position",[0 0 1.2],"color",[0.7 0 0.2 1]);
```

The output of addLink produces a variable containing the assigned name of the link. These variables create the joint.

Use addJoint to define the relationship between the two links. In this case they are attached together by a revolute joint.

```
joint = addJoint(pin,link1,link2,"revolute",[0 0],[0 0 1]);
```

The arguments of the addJoint function are object, parent, child, type, limits, and axis.

After defining bowlPin once, You can create all ten bowling pins from the preceding ExampleHelperGazeboModel. The following for loop spawns the models in Gazebo using the x and y vectors.

```
for i = 1:10
    spawnModel(gazebo,pin,[x(i),y(i),0.7]);
    pause(1);
end
```

After adding the pins to the world, it looks like this:
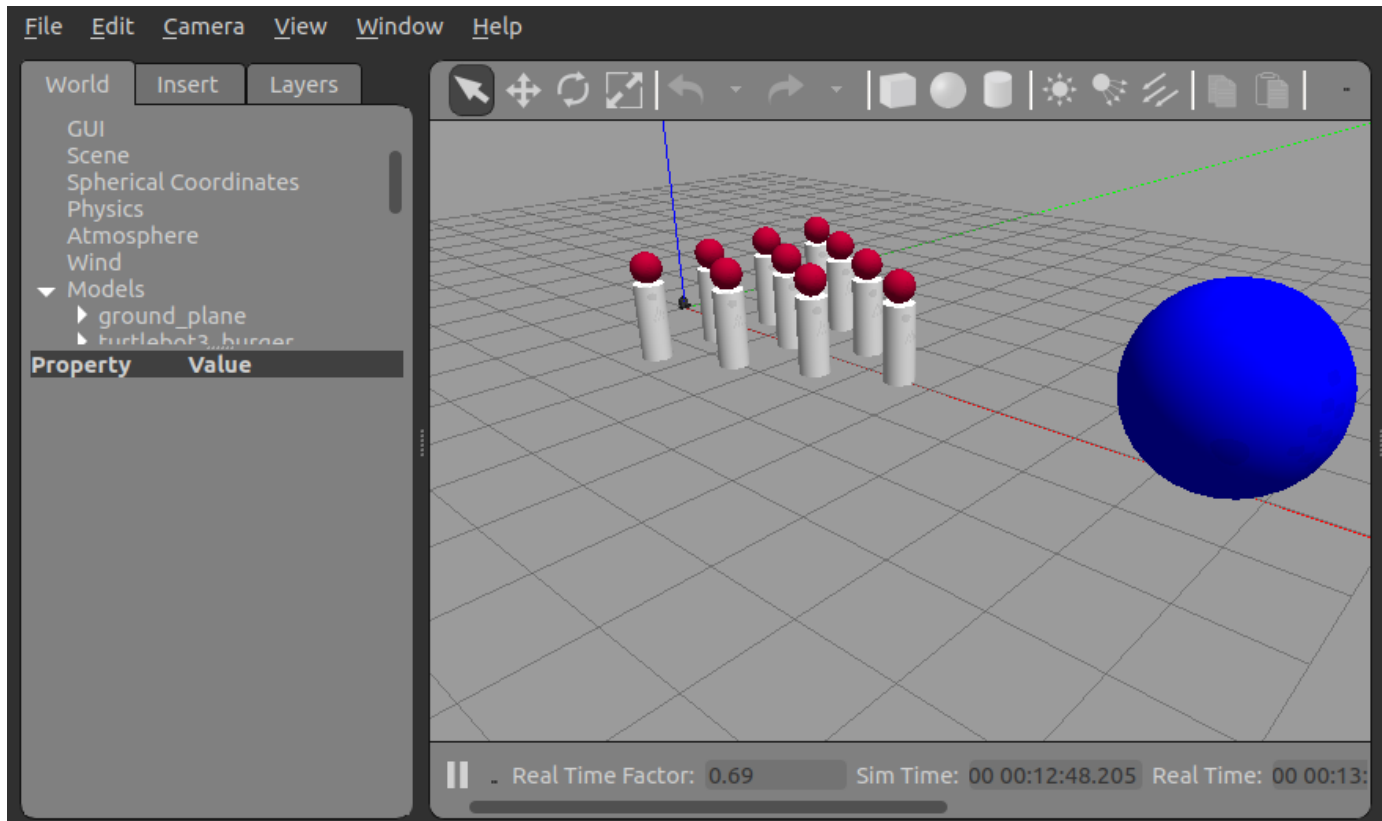
Remove Models

If the TurtleBot® exists in the scene, remove it. Look in the list of models. Remove the one named `turtlebot3_burger`, for this particular world.

```
if ismember("turtlebot3_burger",getSpawnedModels(gazebo))
    removeModel(gazebo,"turtlebot3_burger");
end
```

Spawn Built-In Models

Create an `ExampleHelperGazeboModel` for a Jersey barrier. The object finds this model on the Gazebo website.

```
barrier = ExampleHelperGazeboModel("jersey_barrier","gazeboDB");
```

Spawn two Jersey barriers in the world using `spawnModel`.

```
spawnModel(gazebo,barrier,[1.5 -3 0]); % Right barrier
pause(1);
spawnModel(gazebo,barrier,[1.5 3 0]); % Left barrier
```

Note: You need an Internet connection to spawn models that are not included in these examples. However, if you have previously spawned a model in your Gazebo simulation, it is cached, so you can spawn it later without an Internet connection.

The scene looks like this figure:
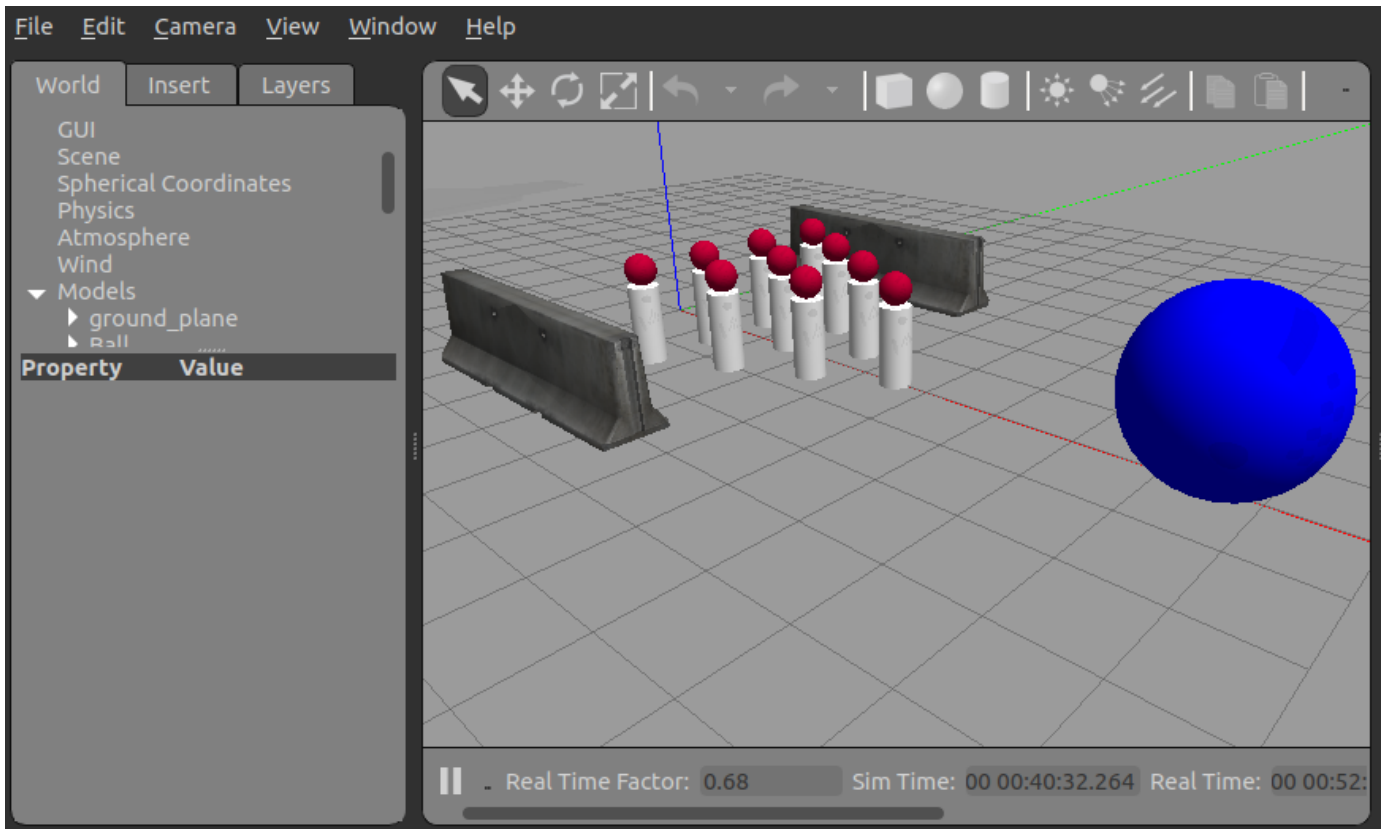
Apply Forces to the Ball

Retrieve the handle to the ball through the `ExampleHelperGazeboSpawnedModel` class.

```
spawnedBall = ExampleHelperGazeboSpawnedModel(ball.Name,gazebo)

spawnedBall =
  ExampleHelperGazeboSpawnedModel with properties:

      Name: 'Ball'
     Links: {'link0'}
    Joints: {0×1 cell}
```

Define parameters for the application of force. Here the duration is set to 1 second and the force vector is set to -75 Newtons in the x direction.

```
duration = 1; % Seconds
forceVec = [-75 0 0]; % Newtons
```

Apply the force to the model using the `applyForce` function.

```
applyForce(spawnedBall,sphereLink,duration,forceVec);
pause(5);
```

Following are images of the collision and the aftermath
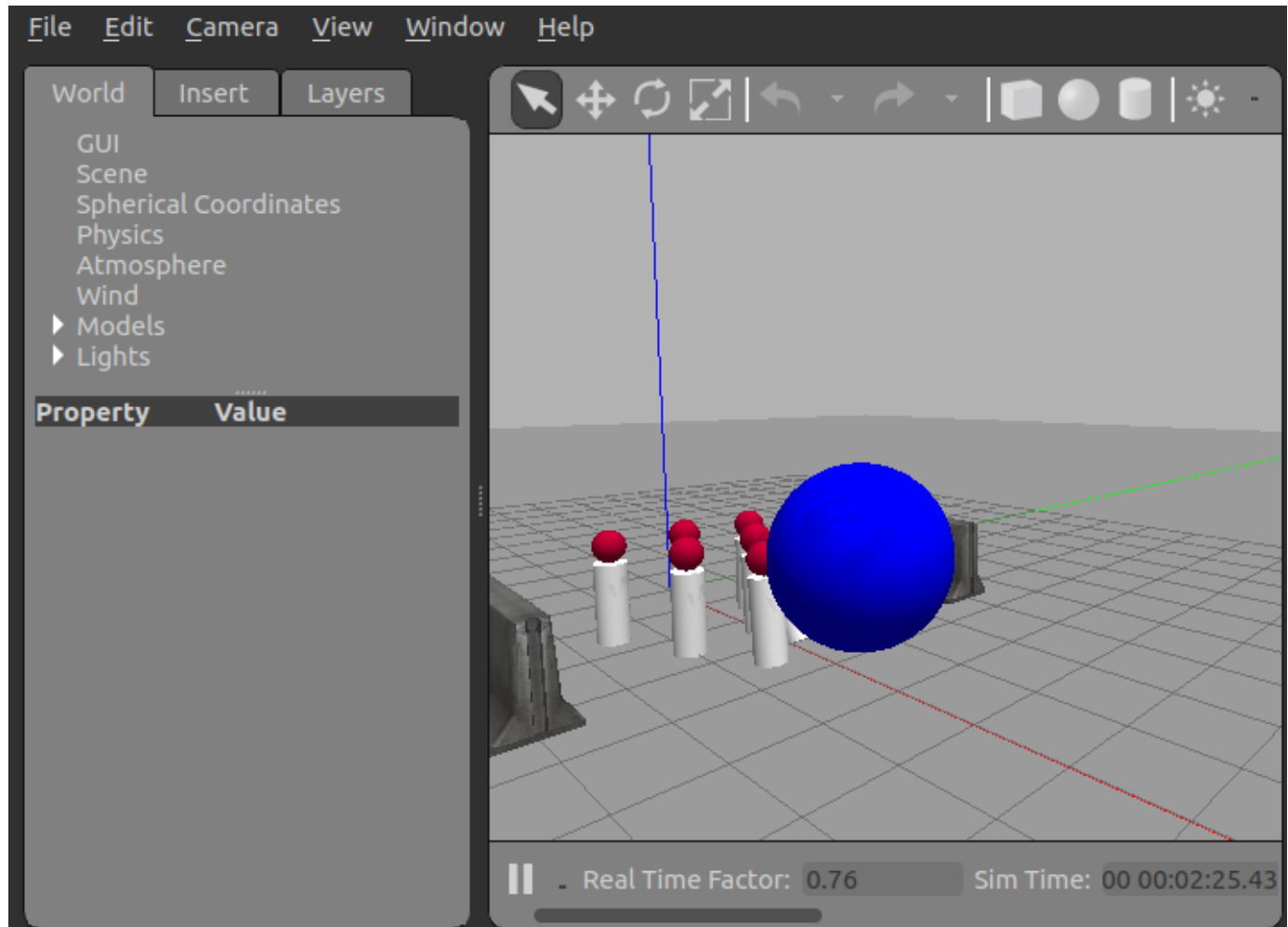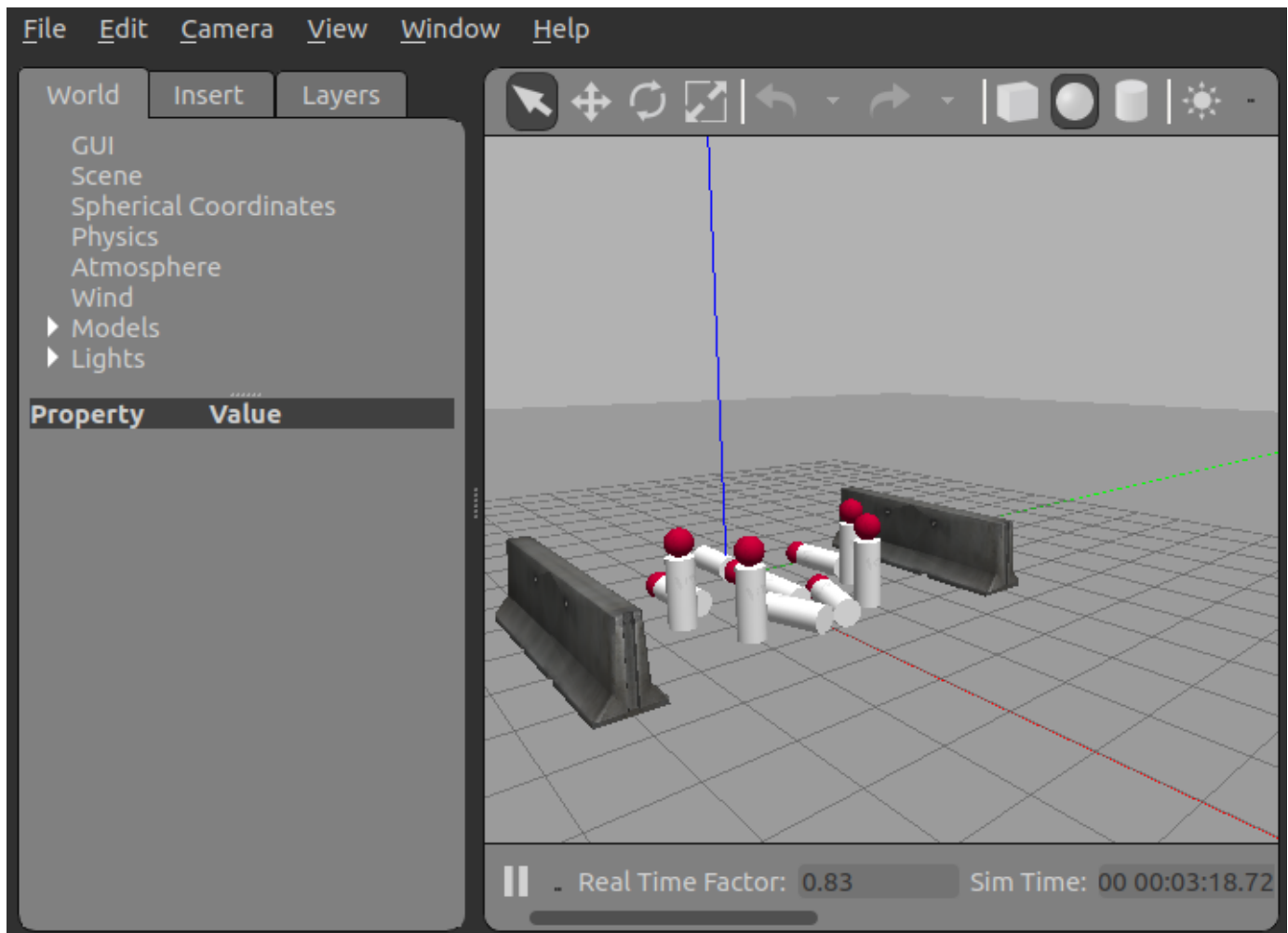
Remove Models and Shut Down

Delete the models created for this example.

```
exampleHelperGazeboCleanupBowling;
```

Clear the workspace of publishers, subscribers, and other ROS-related objects when you are finished with them.

```
clear
```

Use `rosshutdown` once you are done working with the ROS network. Shut down the global node and disconnect from Gazebo.

```
rosshutdown
```

Shutting down global node /matlab_global_node_77778 with NodeURI http://192.168.178.1:52158/

When you are finished, close the Gazebo window on your virtual machine.

See Also

- "Apply Forces and Torques in Gazebo" on page 1-115

Apply Forces and Torques in Gazebo

This example illustrates a collection of ways to apply forces and torques to models in the Gazebo®
simulator. First, application of torques is examined in three distinct ways using doors for illustration.
Second, two TurtleBot® Create models demonstrate the forcing of compound models. Finally, object
properties (bounce, in this case) are examined using basic balls.

Prerequisites: "Get Started with Gazebo and a Simulated TurtleBot" on page 1-102, "Add, Build, and
Remove Objects in Gazebo" on page 1-108

Connect to Gazebo

On your Linux® machine, start Gazebo. If you are using the virtual machine from "Get Started with
Gazebo and a Simulated TurtleBot" on page 1-102, click **Gazebo Empty** world on the desktop.

Initialize ROS by replacing `ipaddress` with the IP address of the virtual machine. Create an instance
of the `ExampleHelperGazeboCommunicator` class.

```
rosinit('http://192.168.233.133:11311')
```

Initializing global node /matlab_global_node_68978 with NodeURI http://192.168.233.1:53907/

```
gazebo = ExampleHelperGazeboCommunicator;
```

Add Moving Doors

This section demonstrates three distinct methods for applying joint torques. In this case, doors are
used.

Create a door model and spawn three instances in the simulator. Specify the spawn position and
orientation (units are meters and radians).

```
doormodel = ExampleHelperGazeboModel('hinged_door','gazeboDB');
door1 = spawnModel(gazebo,doormodel,[-1.5 2.0 0]);
door2 = spawnModel(gazebo,doormodel,[-1.5 0.5 0],[0 0 pi]);
door3 = spawnModel(gazebo,doormodel,[-1.5 -2.5 0]);
```

All units in Gazebo are specified using SI convention. With the doors added, the world looks like this
image:
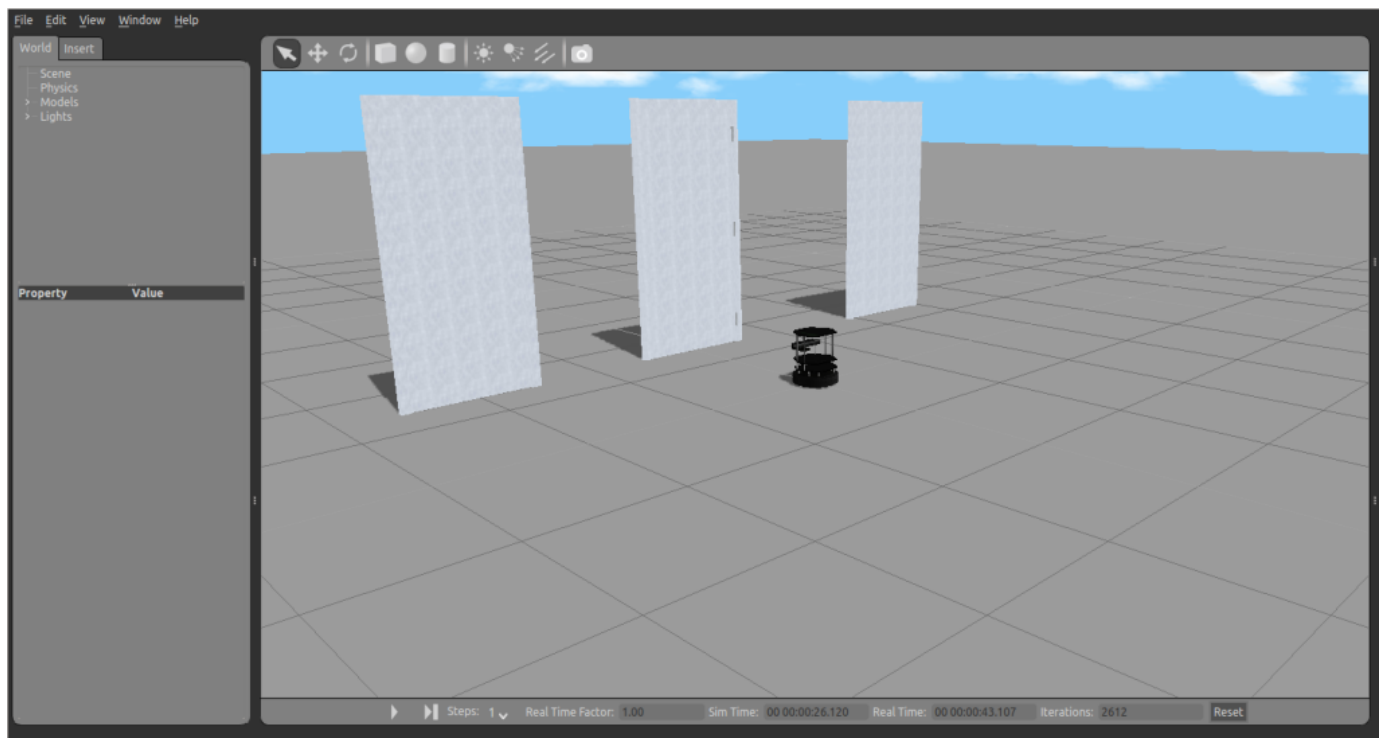
Note: When the Gazebo simulation is left idle, movable items often drift. If you see the doors moving slowly without a command, this behavior is normal. This happens because there is often more friction in the real world than there is in the ideal setting of the Gazebo simulator.

Retrieve handles for the links and joints of the first door and display them.

```
[links, joints] = getComponents(door1)

links = 3×1 cell
    {'hinged_door::frame'  }
    {'hinged_door::door'   }
    {'hinged_door::handles'}


joints = 3×1 cell
    {'hinged_door::handle'     }
    {'hinged_door::hinge'      }
    {'hinged_door::world_joint'}
```

For the first door, apply a torque directly to the `hinge` joint.

Apply the torque to the first door using `jointTorque`. Doing so makes it open and stay open during the simulation. The first two lines define the stop time and effort parameters for the torque application. The second entry in the `joints` cell array is `hinged_door::hinge`. Use this in the `jointTorque` call.

```
stopTime = 5; % Seconds
effort = 3.0; % Newton-meters
jointTorque(door1, joints{2}, stopTime, effort);
```
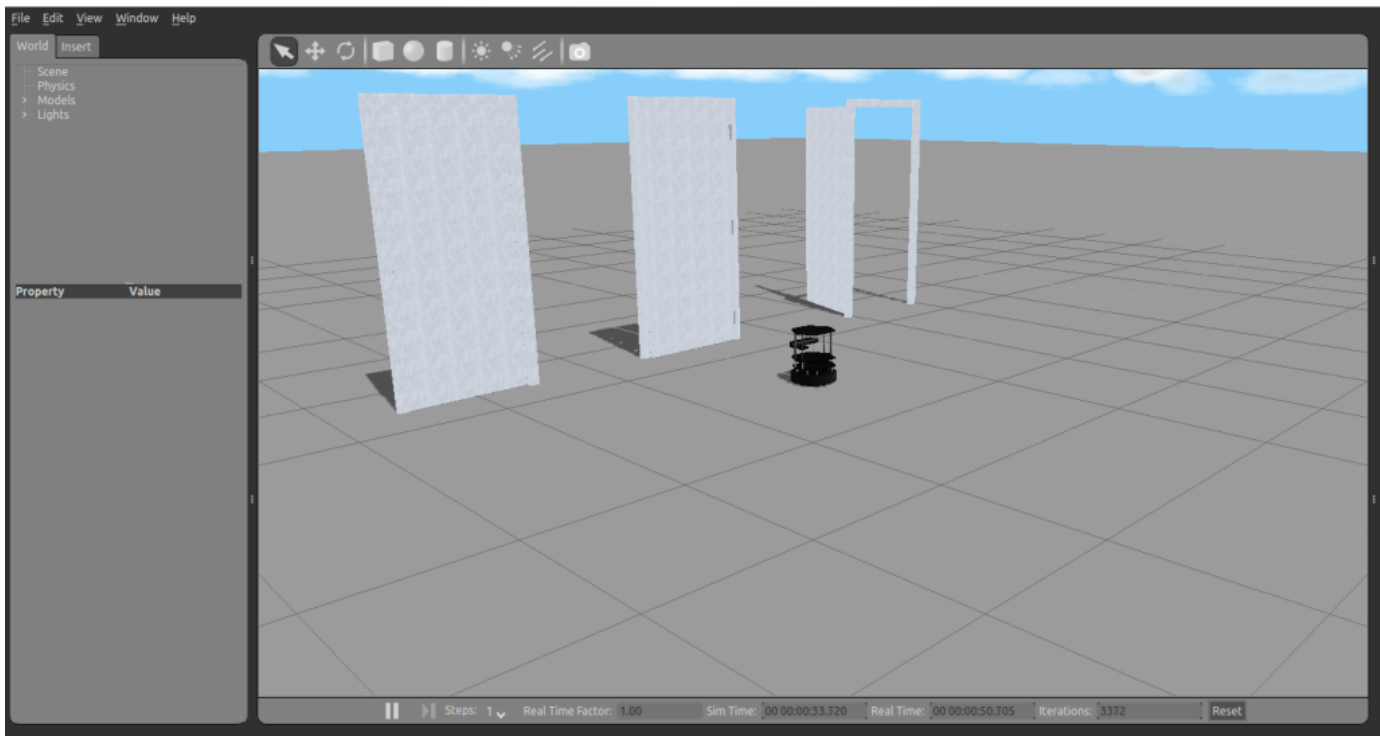
The second method is to apply a torque to the door link instead of the hinge joint. This method is not as clean because the torque is applied to the center of mass of the link (which is the door in this case) and is not applied around the axis of rotation. This method still produces a torque that moves the door.

Use the `applyForce` function. The second entry in `links` is 'hinged_door::door'. Use it in the `applyForce` call.

```
forceVector = [0 0 0];      % Newtons
torqueVector = [0 0 3];     % Newton-meters
applyForce(door2, links{2}, stopTime, forceVector, torqueVector);
```
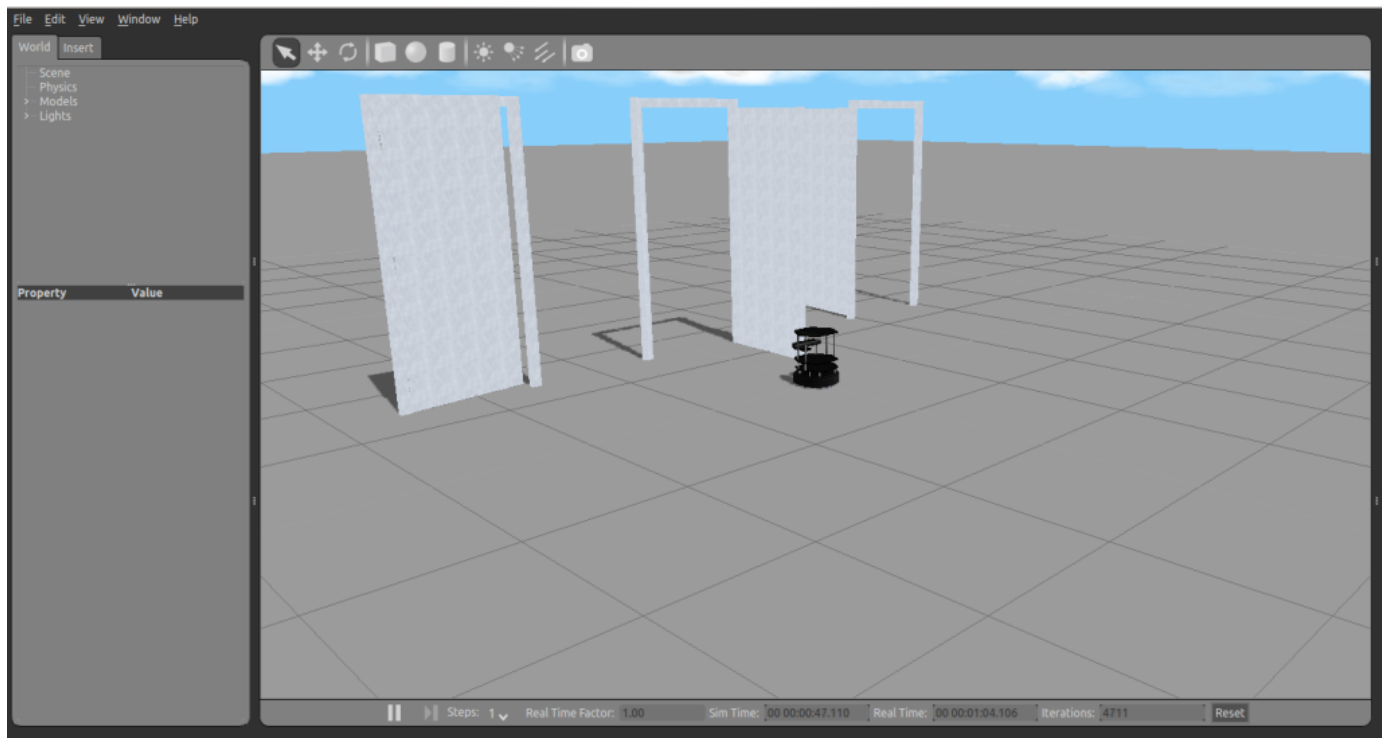
You can apply a force (instead of a torque) directly to the center of mass of the door for it to move. The commands are:

```
forceVector = [0 -2 0];      % Newtons
applyForce(door2, links{2}, stopTime, forceVector);
```

Note: The forces are always applied from the world coordinate frame and not the object frame. When you apply this force, it continually operates in the negative y direction. It does not result in a constant torque on the door.

For the third door, manually define the hinge angle without applying a force or torque.

Use a while loop to create a swinging behavior for the door. Use the `setConfig` function of the `ExampleHelperGazeboSpawnedModel` class.

```
angdelta = 0.1;     % Radians
 dt = 0;             % Seconds
 angle = 0;          % Radians
 tic
 while (toc < stopTime)

     if angle > 1.5 || angle < 0     % In radians
         angdelta = -angdelta;
     end

     angle = angle+angdelta;
     setConfig(door3,joints{2},angle);
     pause(dt);
 end
```
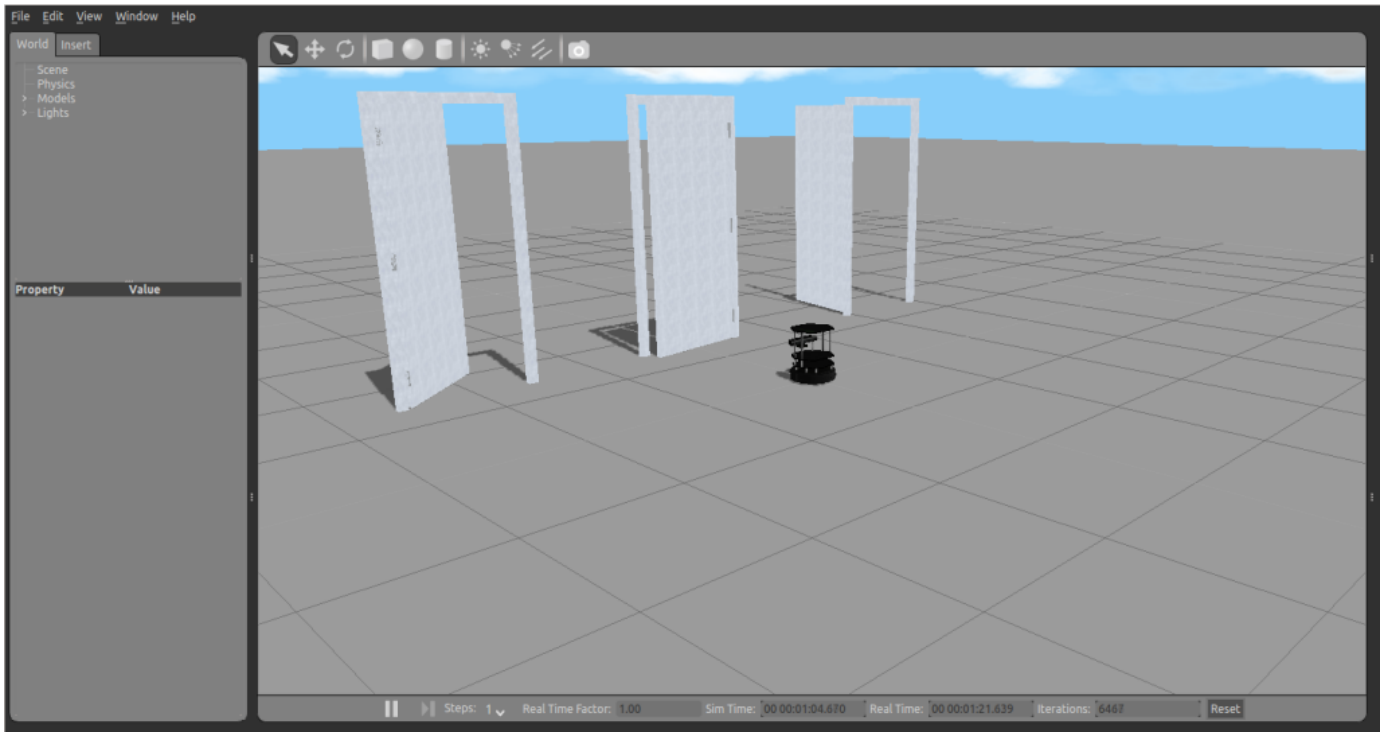
Create TurtleBot Objects for Manipulation

This section demonstrates creation and external manipulation of a TurtleBot Create. It illustrates simple control of a more complex object.

Create another TurtleBot in the world by adding the `GazeboModel` from the database (GazeboDB). The robot spawned is a TurtleBot Create, not a Kobuki. Apply an external torque to its right wheel.

Note: Spawning the Create requires an internet connection.

```
botmodel = ExampleHelperGazeboModel('turtlebot','gazeboDB');
bot = spawnModel(gazebo,botmodel,[1,0,0]);
```

The TurtleBot originally spawns facing along the x-axis with an angle of 0 degrees. Change the orientation to pi/2 radians (90 degrees) using this command:

```
setState(bot,'orientation',[0 0 pi/2]);
```
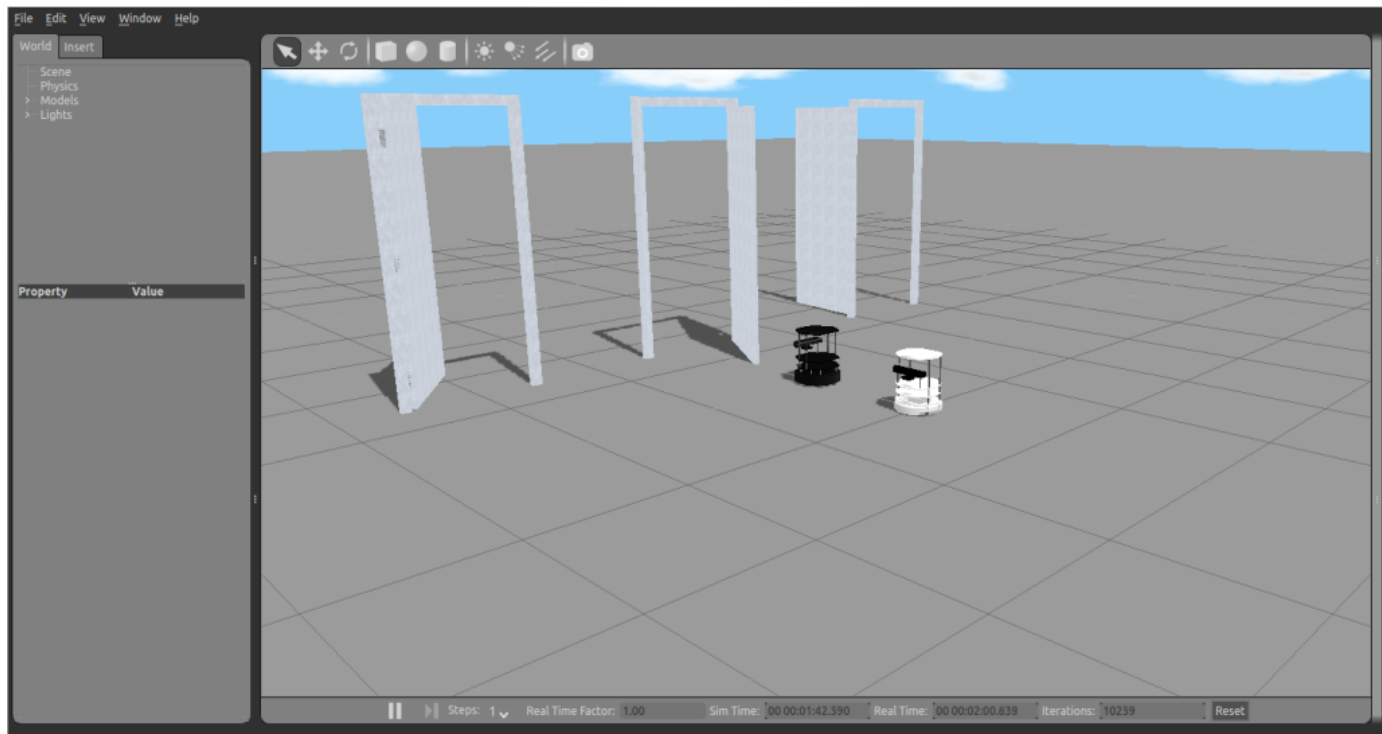
Using `applyForce`, make the right wheel of the TurtleBot Create move by applying an external torque to it from the `ExampleHelperGazeboSpawnedModel` object.

```
[botlinks, botjoints] = getComponents(bot)

botlinks = 5×1 cell
    {'turtlebot::rack'              }
    {'turtlebot::create::base'      }
    {'turtlebot::create::left_wheel' }
    {'turtlebot::create::right_wheel'}
    {'turtlebot::kinect::link'      }


botjoints = 4×1 cell
    {'turtlebot::create::left_wheel' }
    {'turtlebot::create::right_wheel'}
    {'turtlebot::create_rack'        }
    {'turtlebot::kinect_rack'        }
```

The second entry of `botjoints` is 'turtlebot::create::right_wheel' Use botjoints{2} in the `jointTorque` call.

```
turnStopTime = 1;        % Seconds
turnEffort = 0.2;        % Newton-meters
jointTorque(bot, botjoints{2}, turnStopTime, turnEffort)
```
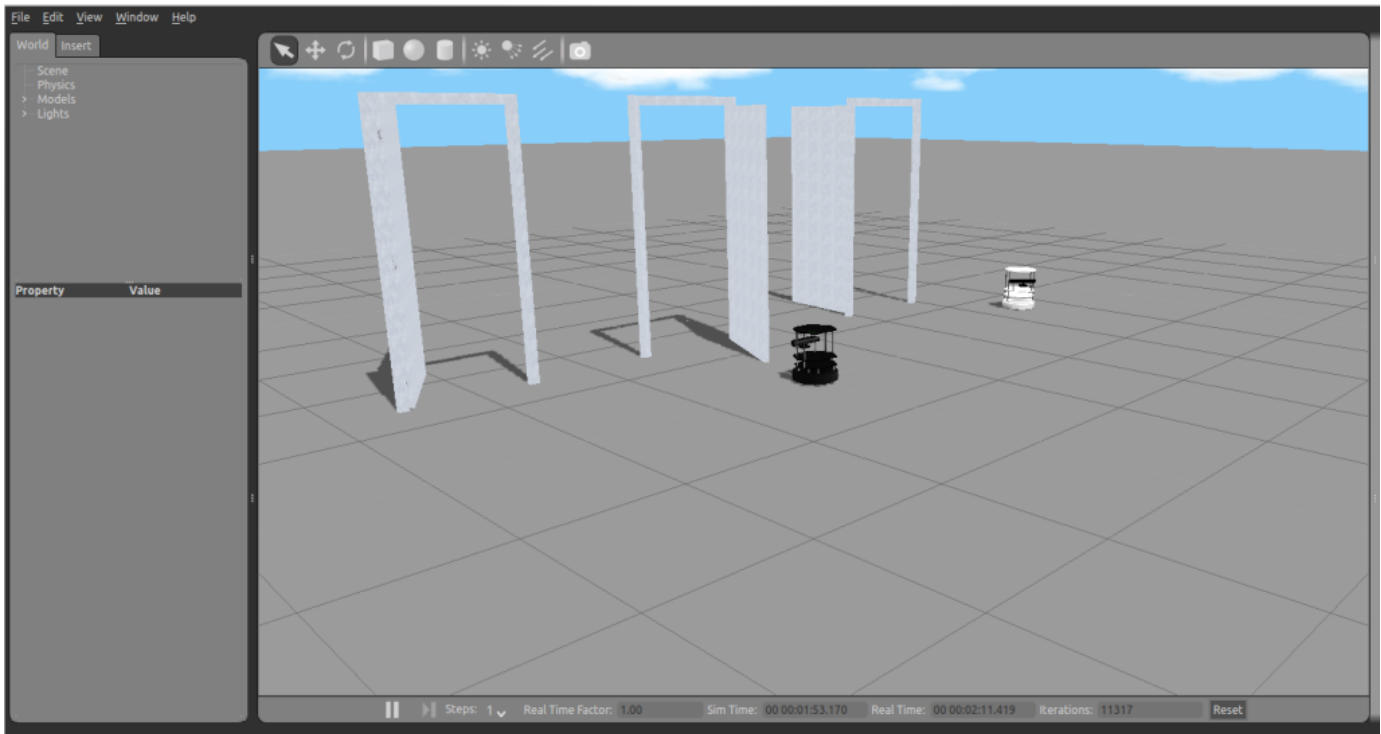
You can experiment with application of forces to a TurtleBot base instead of to the wheels.

Make a second TurtleBot Create with `spawnModel`:

```
bot2 = spawnModel(gazebo,botmodel,[2,0,0]);
[botlinks2, botjoints2] = getComponents(bot2)

botlinks2 = 5×1 cell
    {'turtlebot::rack'               }
    {'turtlebot::create::base'       }
    {'turtlebot::create::left_wheel' }
    {'turtlebot::create::right_wheel'}
    {'turtlebot::kinect::link'       }


botjoints2 = 4×1 cell
    {'turtlebot::create::left_wheel' }
    {'turtlebot::create::right_wheel'}
    {'turtlebot::create_rack'        }
    {'turtlebot::kinect_rack'        }
```
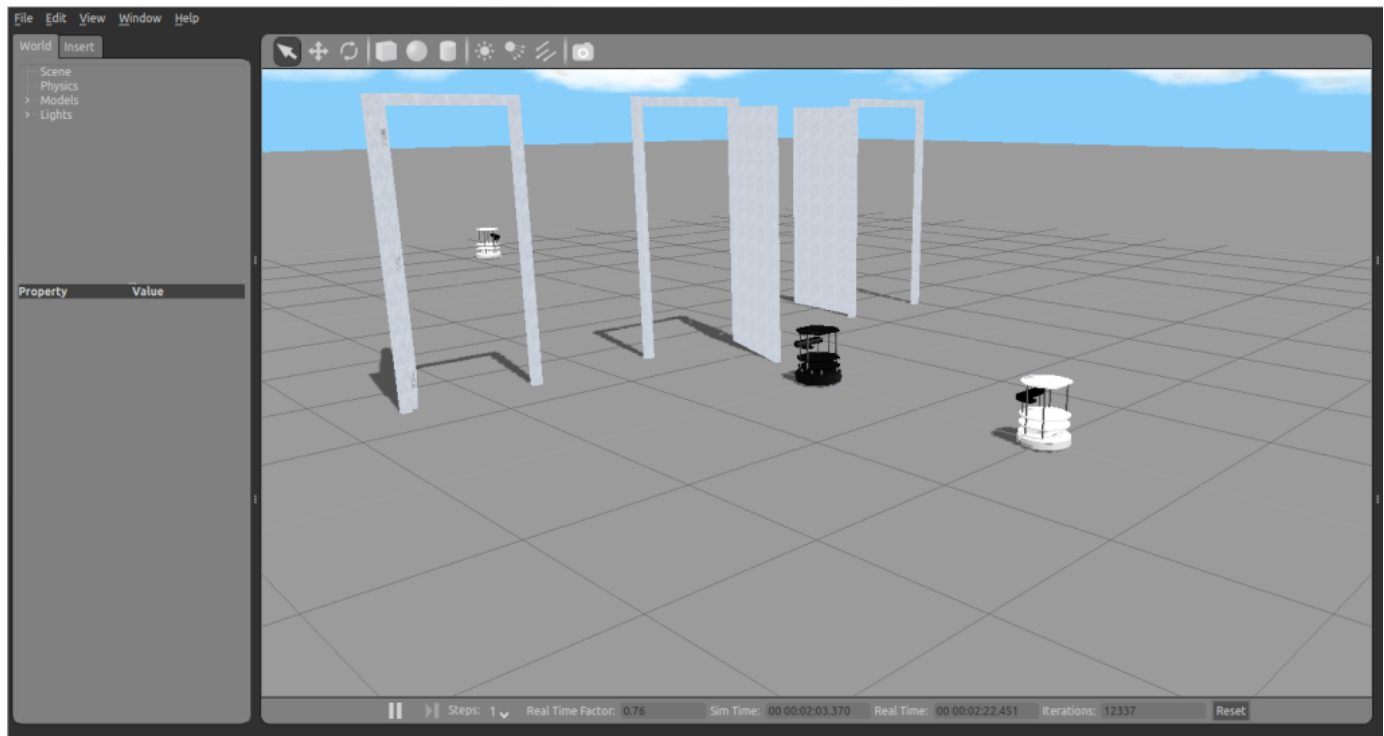
Apply a force to the base in the y direction. See that the base barely moves. The force is acting perpendicular to the wheel orientation.

The first entry of `botlinks2` is 'turtlebot::create::base'. Use `botlinks2{1}` in the `applyForce` call.

```
applyForce(bot2,botlinks2{1},2,[0 1 0]);
```

Apply a force in the x direction. The robot moves more substantially.

```
applyForce(bot2,botlinks2{1},2,[1 0 0]);
```

Apply a torque to the TurtleBot base to make it spin.

```
applyForce(bot2,botlinks2{1},2,[0 0 0],[0 0 1]);
```
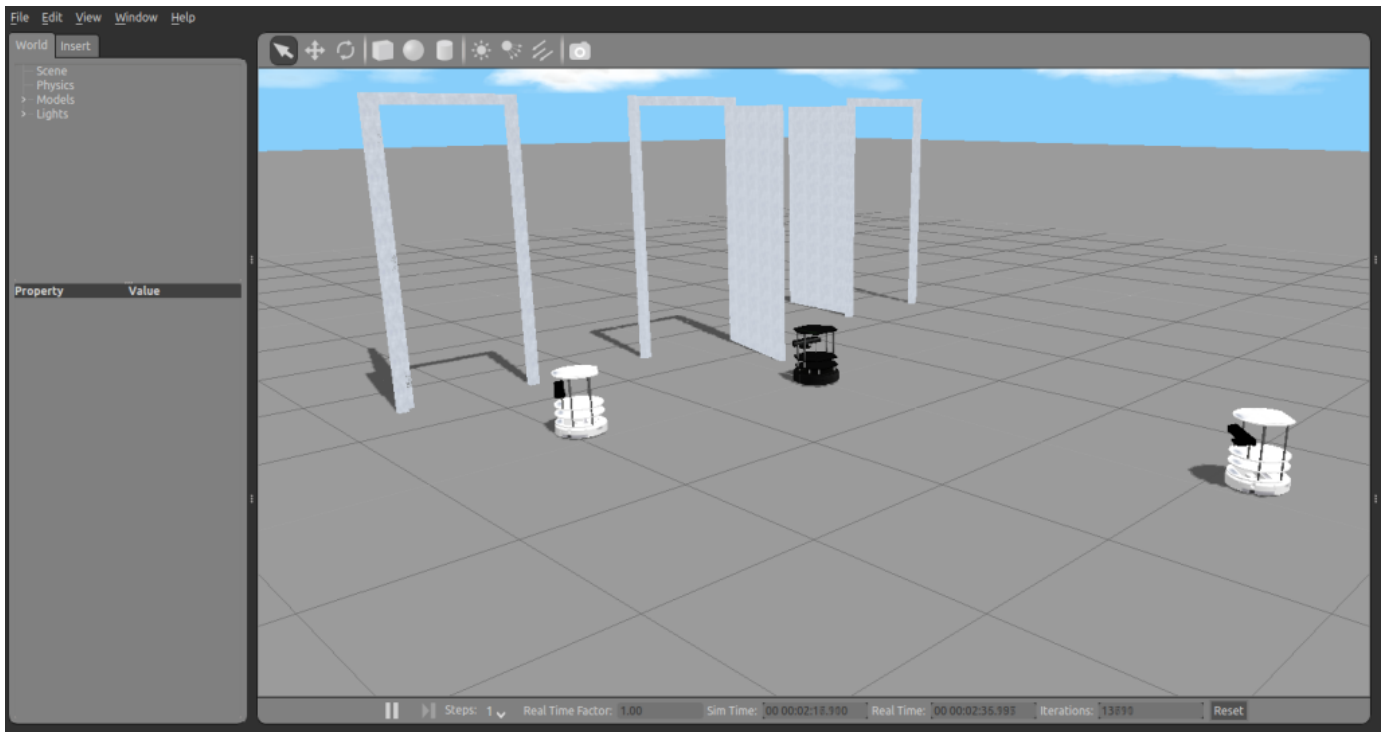
Add Bouncing Balls

This section demonstrates the creation of two balls and exposes the 'bounce' property.

Use the `ExampleHelperGazeboModel` class to create two balls in the simulation. Specify the parameters of bouncing by using `addLink`.

```
bounce = 1;      % Unitless coefficient
maxCorrectionVelocity = 10;      % Meters per second
ballmodel = ExampleHelperGazeboModel('ball');
addLink(ballmodel,'sphere',0.2,'color',[0.3 0.7 0.7 0.5],'bounce',[bounce maxCorrectionVelocity]
```

Spawn two balls, one on top of the other, to illustrate bouncing.

```
spawnModel(gazebo,ballmodel,[0 1 2]);
spawnModel(gazebo,ballmodel,[0 1 3]);

pause(5);
```

After adding the balls, the world looks like this:
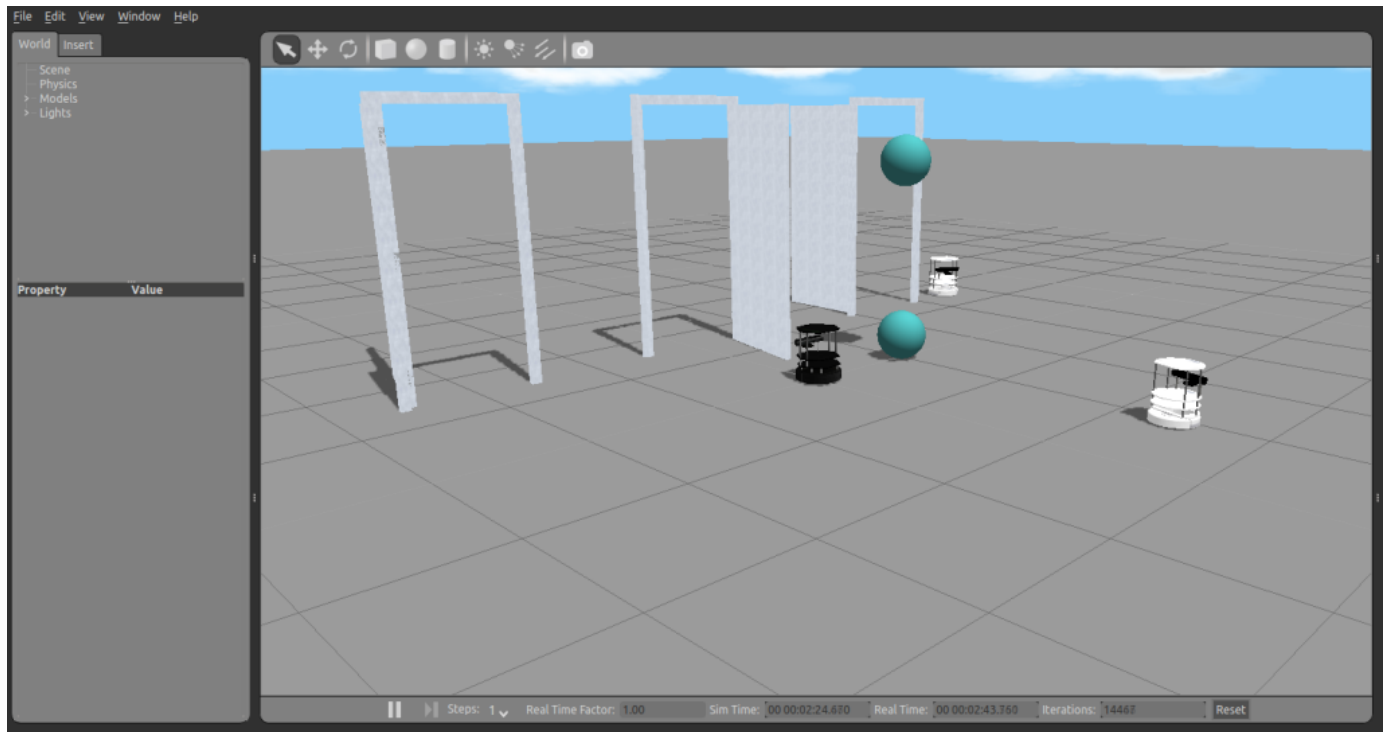
Remove Models and Shut Down

Clean up the models.

```
exampleHelperGazeboCleanupApplyForces;
```

Clear the workspace of publishers, subscribers, and other ROS-related objects when you finish with them.

```
clear
```

Use `rosshutdown` once you are done working with the ROS network. Shut down the global node and disconnect from Gazebo.

```
rosshutdown
```

Shutting down global node /matlab_global_node_68978 with NodeURI http://192.168.233.1:53907/

When finished, close the Gazebo window on your virtual machine

Next Steps

• Refer to the next example: "Test Robot Autonomy in Simulation" on page 1-125

Test Robot Autonomy in Simulation

This example explores MATLAB® control of the Gazebo® Simulator.

When using robot simulators, it is important to test autonomous algorithms and dynamically alter the surroundings in the world while the simulation is running. This example shows how to create basic robot autonomy with Gazebo and how to interact with it. In this example the robot is the TurtleBot® platform. For specific examples involving the TurtleBot, see the "Communicate with the TurtleBot" on page 1-130 example.

In this example, you use a `timer` to control the autonomous aspects of TurtleBot movement. Timers allow processes to run in the background in regular execution intervals without blocking the MATLAB® command line. While you can use loops and other methods to examine basic autonomy, the scheduled execution and non-blocking nature of timers make them the best choice for achieving autonomous behavior.

Prerequisites: "Get Started with Gazebo and a Simulated TurtleBot" on page 1-102, "Add, Build, and Remove Objects in Gazebo" on page 1-108, "Apply Forces and Torques in Gazebo" on page 1-115

Connect to Gazebo

On your Linux® machine, start Gazebo. If you are using the virtual machine from "Get Started with Gazebo and a Simulated TurtleBot" on page 1-102, use the **Gazebo Empty** world.

Initialize ROS by replacing the sample IP address with the IP address of the virtual machine. Create an instance of the `ExampleHelperGazeboCommunicator` class.

```
rosinit("http://192.168.178.132:11311")
```

Initializing global node /matlab_global_node_19208 with NodeURI http://192.168.178.1:53310/

```
gazebo = ExampleHelperGazeboCommunicator;
```

Build a wall in the world.

```
 wall = ExampleHelperGazeboModel("grey_wall","gazeboDB");
 spawnModel(gazebo,wall,[-2 4 0]);
```

All units in Gazebo are specified using SI convention.

Create a `ExampleHelperGazeboSpawnedModel` object for the mobile base and change its orientation state. Manually rotate the TurtleBot by 90 degrees ($\pi/2$ radians) so that it is directly facing the wall.

```
 turtleBot = ExampleHelperGazeboSpawnedModel("turtlebot3_burger",gazebo);
 setState(turtleBot,"orientation",[0 0 pi/2]);
```
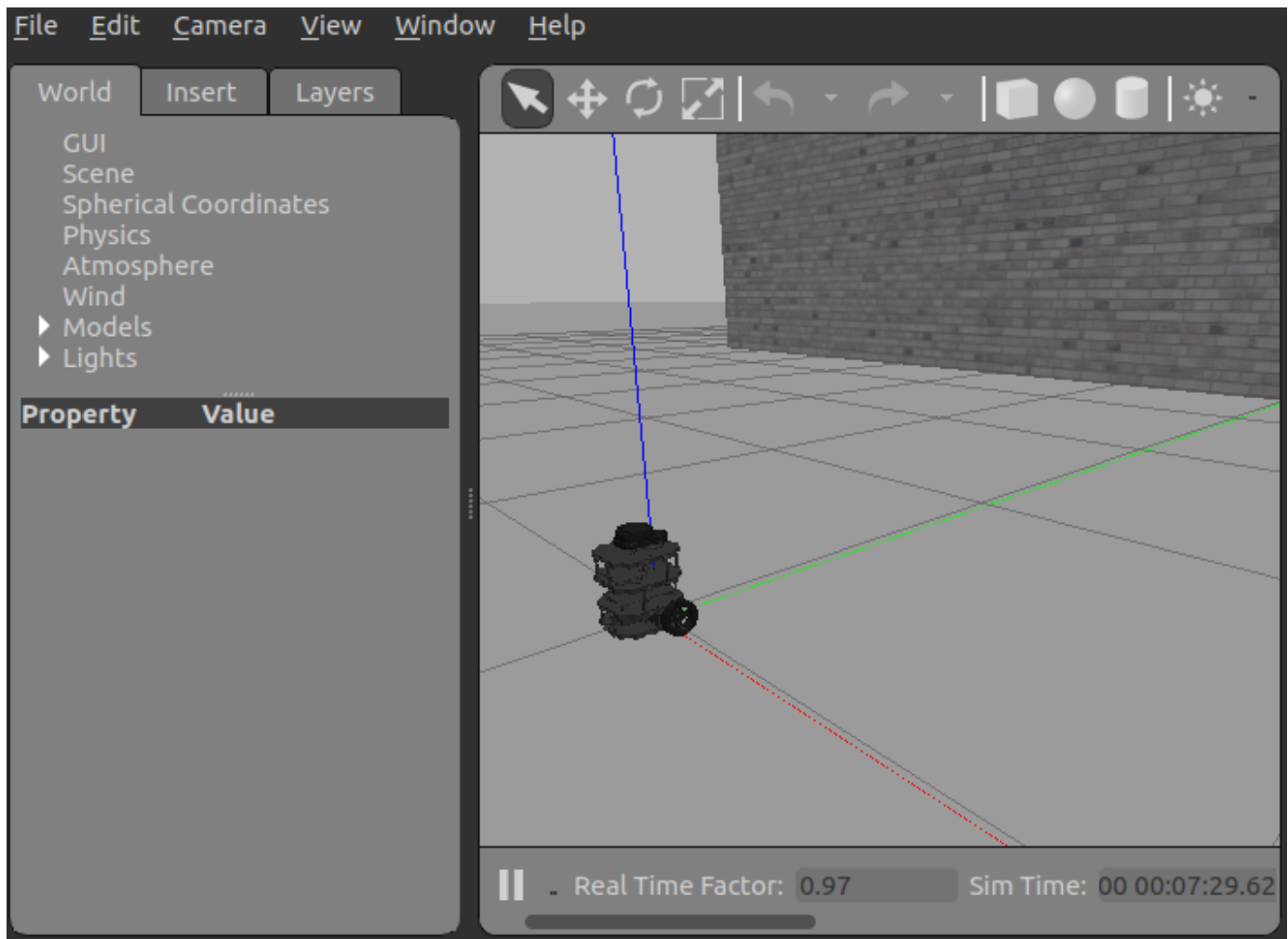
Start TurtleBot Obstacle Avoidance

This section describes a simple way to create autonomous behavior on a TurtleBot in Gazebo. Use a basic obstacle avoidance behavior for the TurtleBot. The behavior is to drive forward and turn when the robot gets very close to an obstacle detected by the laser scanner.

Create global variables for the publisher and publisher message so they can be accessed by the control algorithm.

```
global robot
global velmsg
```

Create the publisher for velocity and the ROS message to carry the information.

```
robot = rospublisher("/cmd_vel");
velmsg = rosmessage(robot);
```

Subscribe to the laser scan topic.

```
timerHandles.sub = rossubscriber("/scan");
```

Create a timer to control the main control loop of the TurtleBot.

```
t = timer("TimerFcn",{@exampleHelperGazeboAvoidanceTimer,timerHandles},"Period",0.1,"ExecutionM
```

The timer callback function, `exampleHelperGazeboAvoidanceTimer` defines the laser scan callback function and executes a basic algorithm to allow the TurtleBot to avoid hitting objects as it moves.

Start the timer.

```
start(t)
```

The TurtleBot drives toward the wall. Once it gets very close to the wall, it must turn left to avoid running into it.
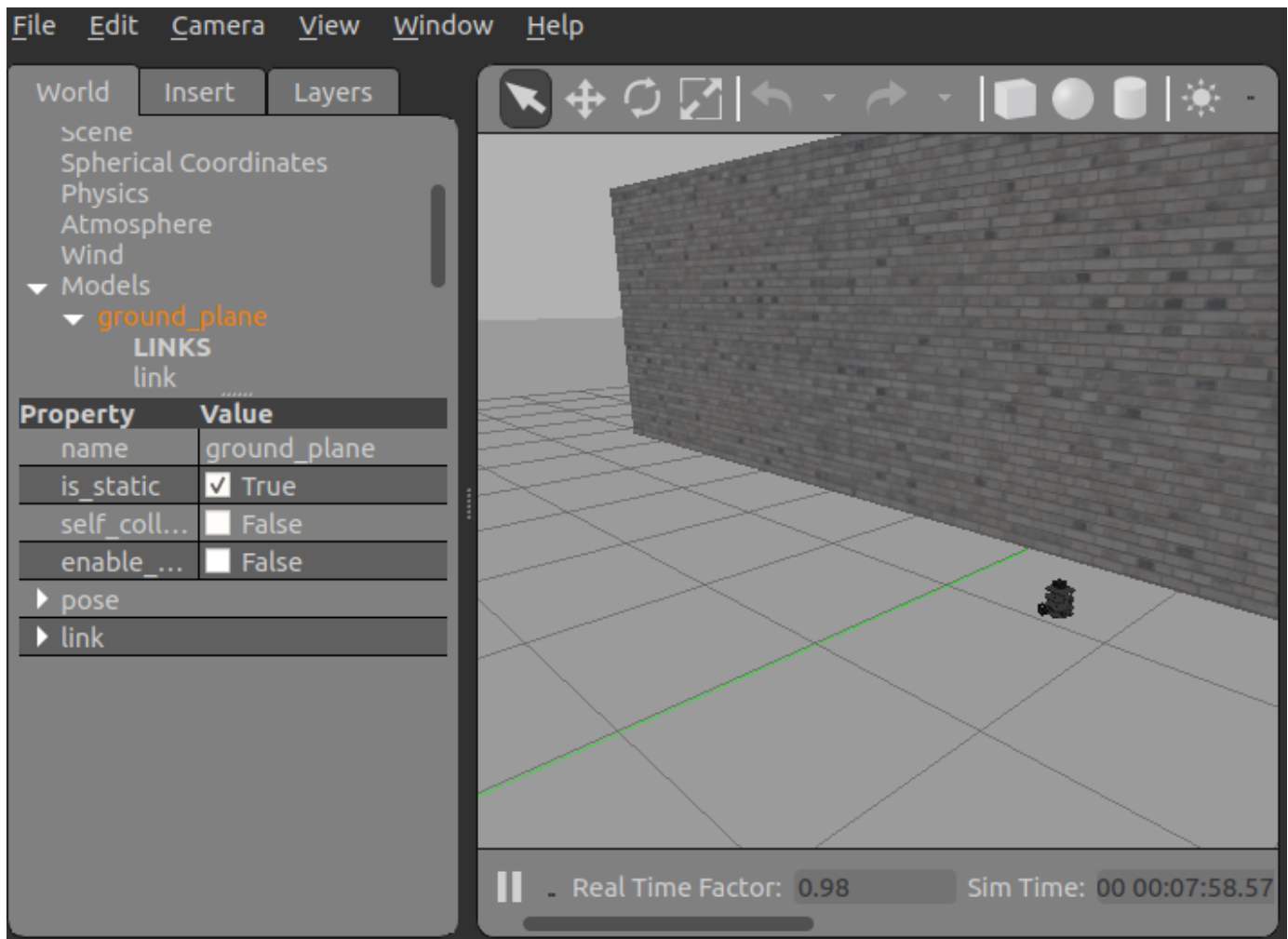
Note: If the TurtleBot crashes into the wall, the laser scan is probably not being published through Gazebo. Restart your Gazebo session and try again.

Add Objects

You can still make changes to the world while the TurtleBot is moving. Add a new wall to the world. If you add it soon enough, it can block the TurtleBot so that it avoids hitting the wall.

```
spawnModel(gazebo,wall,[-5.85 0.15 0],[0, 0, pi/2]);
pause(20)       % TurtleBot avoids walls for 20 seconds
```
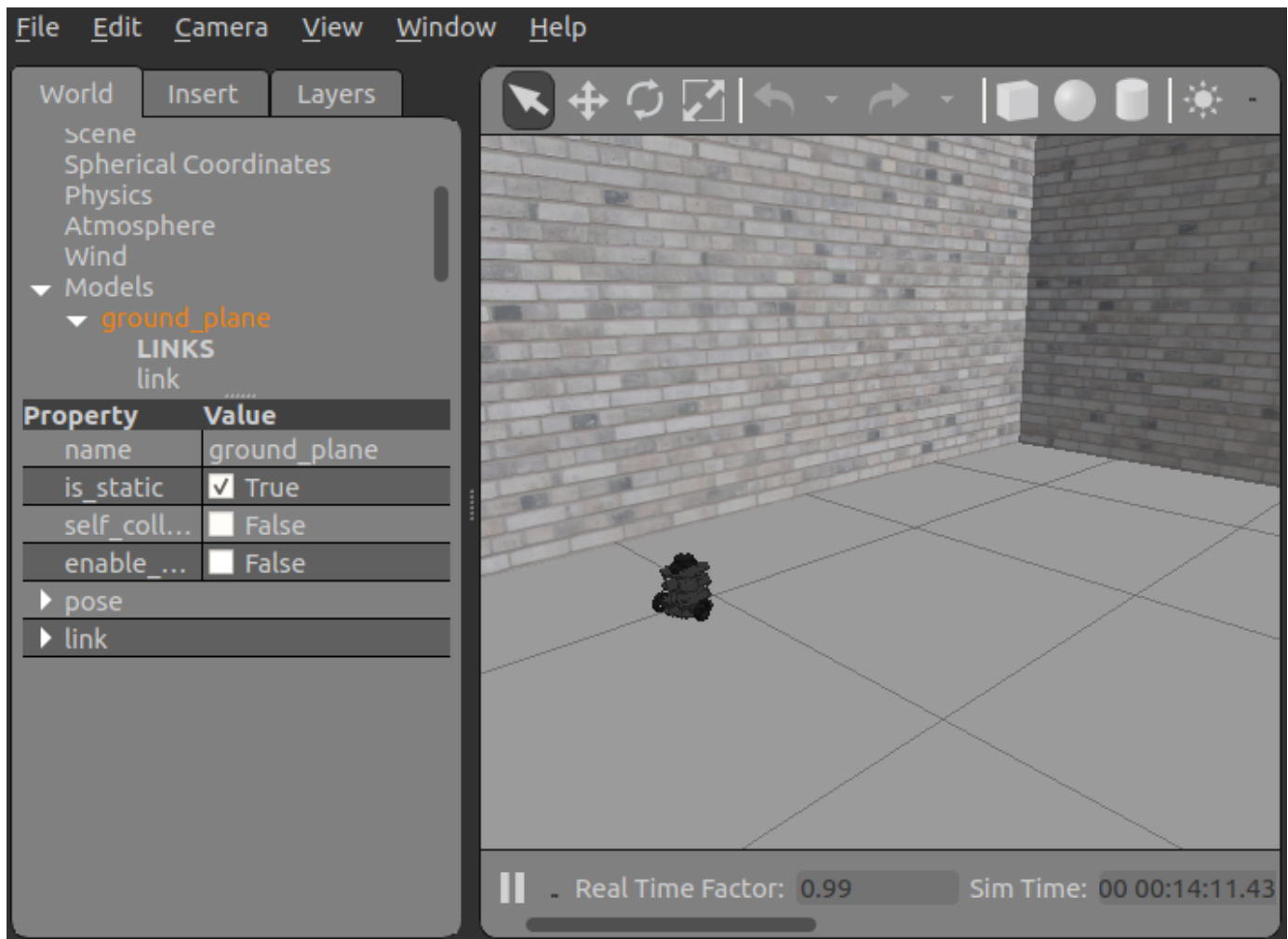

Remove Models and Shut Down

Stop the timer to halt the robot algorithm.

```
stop(t)
delete(t)
```

Find all objects in the world and remove the ones added manually.

```
list = getSpawnedModels(gazebo)
```

```
list = 4×1 cell
    {'ground_plane'    }
    {'turtlebot3_burger'}
    {'grey_wall'       }
    {'grey_wall_0'     }
```

Remove the two walls, using the following commands:

```
removeModel(gazebo,"grey_wall");
removeModel(gazebo,"grey_wall_0");
```

Clear the workspace of publishers, subscribers, and other ROS-related objects when you are finished with them.

```
clear
```

Use `rosshutdown` once you are done working with the ROS network. Shut down the global node and disconnect from Gazebo.

```
rosshutdown
```

Shutting down global node /matlab_global_node_19208 with NodeURI http://192.168.178.1:53310/

When finished, close the Gazebo window on your virtual machine.

Communicate with the TurtleBot

This example introduces the TurtleBot® platform and the ways in which MATLAB® users can interact with it. Specifically, the code in this example demonstrates how to publish messages to the TurtleBot (such as velocities) and how to subscribe to topics that the TurtleBot publishes (such as odometry).

The TurtleBot must be running for this example to work.

Prerequisites: "Get Started with Gazebo and a Simulated TurtleBot" on page 1-102 or "Get Started with a Real TurtleBot" on page 1-56

Connect to the TurtleBot

The TurtleBot must be running. If you are using a real TurtleBot and followed the hardware setup steps in "Get Started with a Real TurtleBot" on page 1-56, the robot is running. If you are using a TurtleBot in simulation and followed the setup steps in "Get Started with Gazebo and a Simulated TurtleBot" on page 1-102, launch one of the Gazebo® worlds from the desktop (`Gazebo Office`, for instance).

In your MATLAB instance on the host computer, run the following command. Replace `ipaddress` with the IP address of the TurtleBot. This line initializes ROS and connects to the TurtleBot.

```
ipaddress = "http://192.168.178.132:11311";
rosinit(ipaddress)
```

```
Initializing global node /matlab_global_node_00696 with NodeURI http://192.168.178.1:64340/
```

If the network you are using to connect to the TurtleBot is not your default network adapter, you can manually specify the IP address of the adapter that is used to connect to the robot. This might happen if you use a Wireless network, but also have an active Ethernet connection. Replace `IP_OF_TURTLEBOT` with the IP address of the TurtleBot and `IP_OF_HOST_COMPUTER` with the IP address of the host adapter that is used to connect to the robot:

```
rosinit("IP_OF_TURTLEBOT","NodeHost","IP_OF_HOST_COMPUTER");
```

Display all the available ROS topics using:

```
rostopic list
```

If you do not see any topics, then the network has not been set up properly. Refer to the beginning of this document for network setup steps.

Move the Robot

You can control the movement of the TurtleBot by publishing a message to the `/cmd_vel` topic. The message has to be of type `geometry_msgs/Twist`, which contains data specifying desired linear and angular velocities. The TurtleBot's movements can be controlled through two different values: the linear velocity along the *X*-axis controls forward and backward motion and the angular velocity around the *Z*-axis controls the rotation speed of the robot base.

Set a variable `velocity` to use for a brief TurtleBot movement.

```
velocity = 0.1;      % meters per second
```

Create a publisher for the `/cmd_vel` topic and the corresponding message containing the velocity values.

```
robotCmd = rospublisher("/cmd_vel") ;
velMsg = rosmessage(robotCmd);
```

Set the forward velocity (along the *X*-axis) of the robot based on the `velocity` variable and publish the command to the robot. Let it move for a moment, and then bring it to a stop.

```
velMsg.Linear.X = velocity;
send(robotCmd,velMsg)
pause(4)
velMsg.Linear.X = 0;
send(robotCmd,velMsg)
```
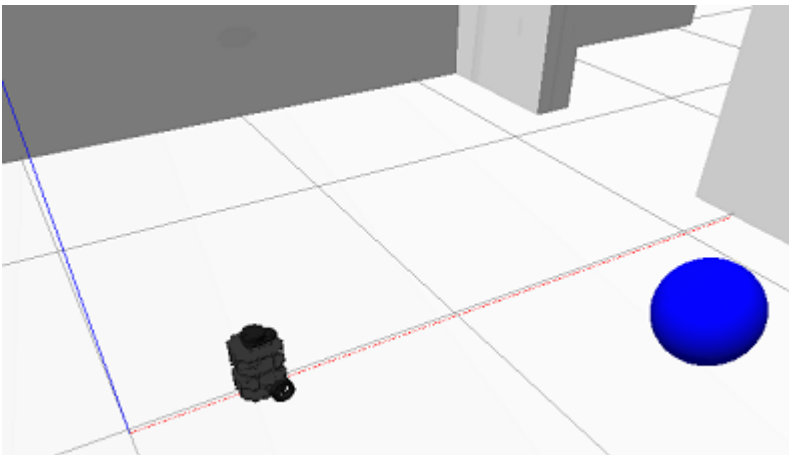

To view the type of the message published by the velocity topic, execute the following:

```
rostopic type /cmd_vel
```

```
geometry_msgs/Twist
```

The topic expects messages of type `geometry_msgs/Twist`, which is exactly the type of the `velMsg` created above.

To view which nodes are publishing and subscribing to a given topic, use the command: `rostopic info TOPICNAME`. The following command lists the publishers and subscribers for the velocity topic. MATLAB is listed as one of the publishers.

```
rostopic info /cmd_vel
```

```
Type: geometry_msgs/Twist

Publishers:
* /matlab_global_node_00696 (http://192.168.178.1:64340/)

Subscribers:
* /gazebo (http://192.168.178.132:41095/)
```

Receive Robot Position and Orientation

The TurtleBot uses the `/odom` topic to publish its current position and orientation (collectively denoted as pose). Since the TurtleBot is not equipped with a GPS system, the pose will be relative to the pose that the robot had when it was first turned on.

Create a subscriber for the odometry messages

```
odomSub = rossubscriber("/odom");
```

Wait for the subscriber to return data, then extract the data and assign it to variables `x`, `y`, and `z`:

```
odomMsg = receive(odomSub,3);
pose = odomMsg.Pose.Pose;
x = pose.Position.X;
y = pose.Position.Y;
z = pose.Position.Z;
```

Note: If you see an error, then it is likely that the `receive` command timed out. Make sure that odometry is being published and that your network is set up properly.

Display the *x*, *y*, and *z* values

```
[x y z]
```

```
ans = 1×3

    0.3883    0.0003   -0.0010
```

The orientation of the TurtleBot is stored as a quaternion in the `Orientation` property of `pose`. Use `quat2eul` (Robotics System Toolbox) to convert into the more convenient representation of Euler angles. To display the current orientation, `theta`, of the robot in degrees, execute the following lines.

```
quat = pose.Orientation;
angles = quat2eul([quat.W quat.X quat.Y quat.Z]);
theta = rad2deg(angles(1))
```

```
theta = -0.0274
```

Receive Lidar Data

Subscribe to the lidar topic:

```
lidarSub = rossubscriber("/scan");
```

After subscribing to the lidar topic, wait for the data and then display it with plot.

```
scanMsg = receive(lidarSub);
figure
plot(scanMsg)
```

Laser Scan
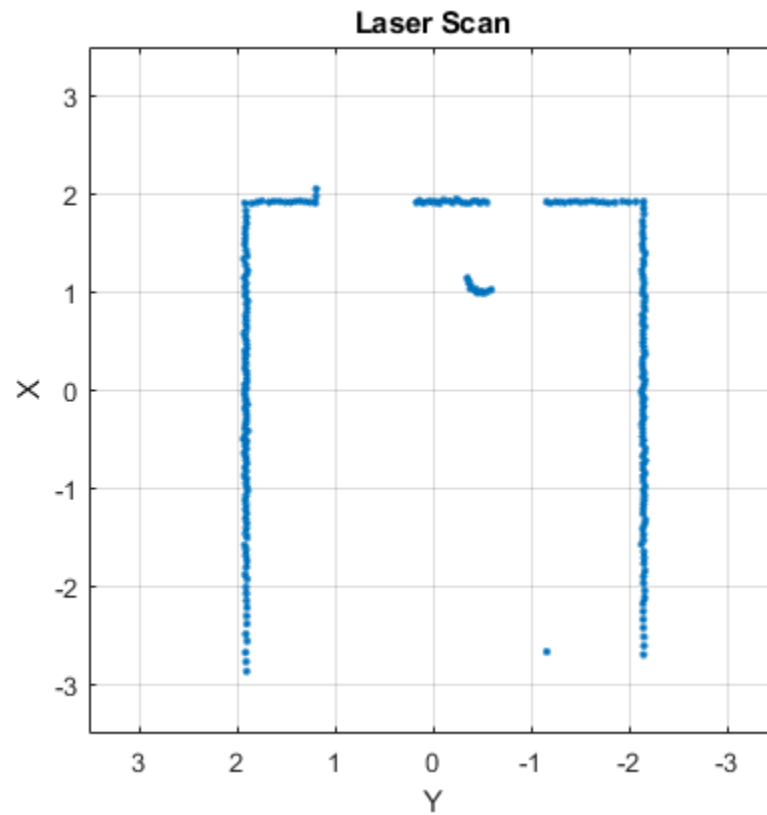
To continuously display updating lidar scans while the robot turns for a short duration, use the following while loop:

```
velMsg.Angular.Z = velocity;
send(robotCmd,velMsg)
tic
while toc < 20
  scanMsg = receive(lidarSub);
  plot(scanMsg)
end
```

Laser Scan
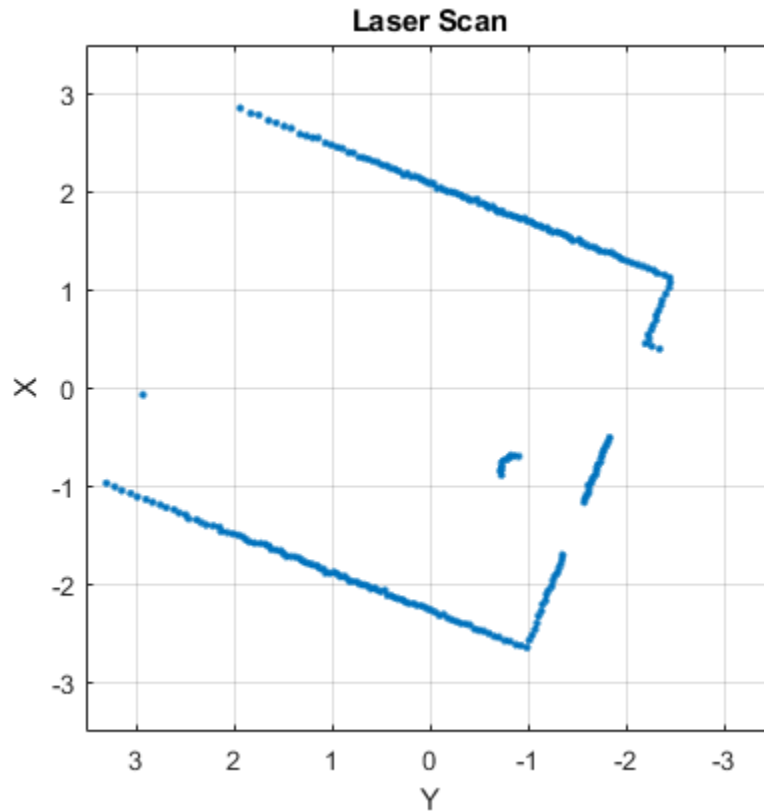

```
velMsg.Angular.Z = 0;
send(robotCmd,velMsg)
```

Disconnect from the Robot

Clear the workspace of publishers, subscribers, and other ROS-related objects when you are finished with them.

```
clear
```

Use `rosshutdown` once you are done working with the ROS network. Shut down the global node and disconnect from the TurtleBot.

```
rosshutdown
```

Shutting down global node /matlab_global_node_00696 with NodeURI http://192.168.178.1:64340/

Next Steps

Refer to the next example: "Explore Basic Behavior of the TurtleBot" on page 1-135

Explore Basic Behavior of the TurtleBot

This example helps you to explore basic autonomy with the TurtleBot®. The described behavior drives the robot forward and changes its direction when there is an obstacle. You will subscribe to the laser scan topic and publish the velocity topic to control the TurtleBot.

Prerequisites: "Communicate with the TurtleBot" on page 1-130

Connect to the TurtleBot

Make sure you have a TurtleBot running either in simulation through Gazebo® or on real hardware. Refer to "Get Started with Gazebo and a Simulated TurtleBot" on page 1-102 or "Get Started with a Real TurtleBot" on page 1-56 for the startup procedure. This example uses the Gazebo-simulated Turtlebot.

In the downloaded virtual machine, click the **Gazebo Office** shortcut to startup the world.

Initialize ROS. Connect to the TurtleBot by replacing `ipaddress` with the IP address of the TurtleBot.

```
ipaddress = 'http://192.168.203.132:11311'

ipaddress =
'http://192.168.203.132:11311'
```

```
rosinit(ipaddress)
```

Initializing global node /matlab_global_node_48605 with NodeURI http://192.168.203.1:60009/

Create a publisher for the robot's velocity and create a message for that topic.

```
robot = rospublisher('/cmd_vel');
velmsg = rosmessage(robot);
```

Receive Scan Data

Make sure that you start the lidar and camera if you are working with real TurtleBot hardware. The command to start the lidar and camera is:

```
 roslaunch turtlebot3_bringup turtlebot3_core.launch
 roslaunch turtlebot3_bringup turtlebot3_lidar.launch
 roslaunch turtlebot3_bringup turtlebot3_rpicamera.launch
```

You must execute the command in a terminal on the TurtleBot. The TurtleBot uses the LDS-01 Lidar to construct a laser scan that is published on the `/scan` topic. For the remainder of this example, the term *laser scan* refers to data published on this topic.

Subscribe to the topic `/scan`.

```
laser = rossubscriber('/scan');
```

Wait for one laser scan message to arrive and then display it.

```
scan = receive(laser,3)

scan =
  ROS LaserScan message with properties:
```

```
    MessageType: 'sensor_msgs/LaserScan'
         Header: [1×1 Header]
       AngleMin: 0
       AngleMax: 6.2832
 AngleIncrement: 0.0175
  TimeIncrement: 0
       ScanTime: 0
       RangeMin: 0.1200
       RangeMax: 3.5000
         Ranges: [360×1 single]
    Intensities: [360×1 single]

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

```
figure
plot(scan);
```
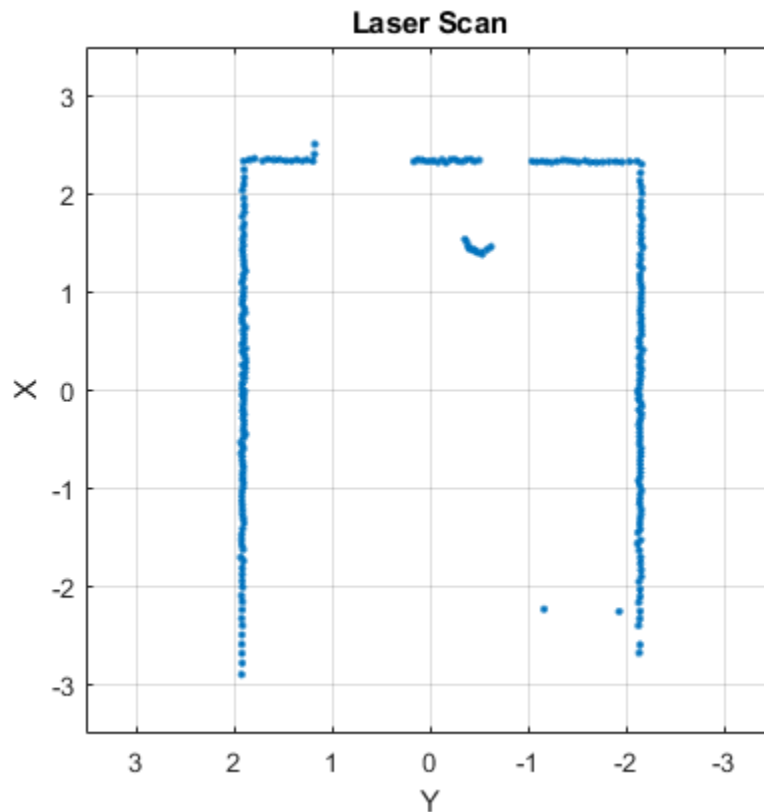

If you see an error, it is possible that the laser scan topic is not receiving any data. If you are running in simulation, try restarting Gazebo. If you are using hardware, make sure that you started the lidar and camera properly.

Run the following lines of code, which plot a live laser scan feed for ten seconds. Move an object in front of the TurtleBot and bring it close enough until it no longer shows up in the plot window. The laser scan has a limited range because of hardware limitations. The LDS-01 lidar has a minimum

sensing range of 0.12 meters and a maximum range of 3.5 meters. Any objects outside these limits will not be detected by the sensor.

```
tic;
while toc < 10
  scan = receive(laser,3);
  plot(scan);
end
```
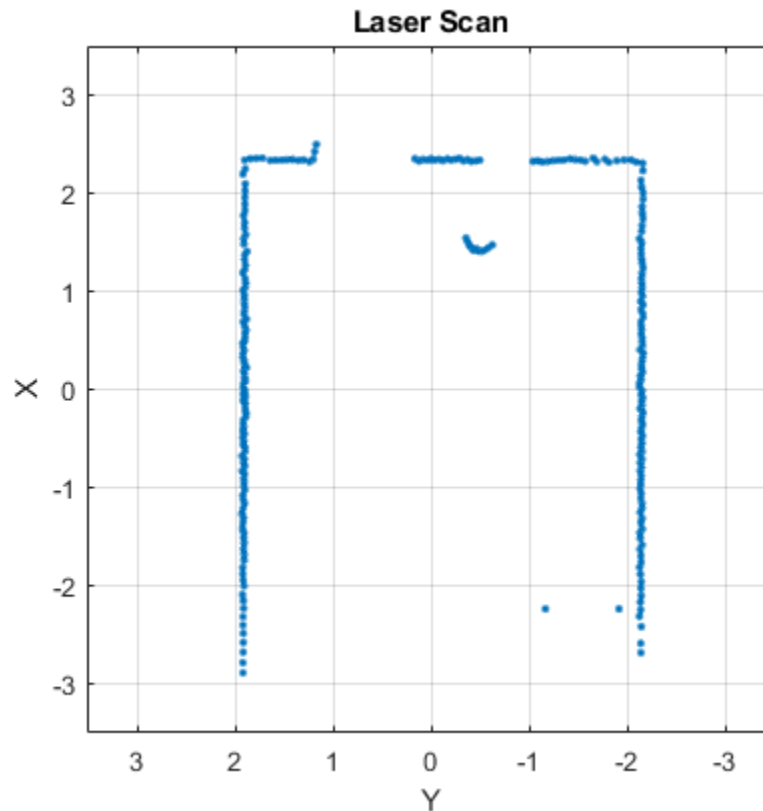

Simple Obstacle Avoidance

Based on the distance readings from the laser scan, you can implement a simple obstacle avoidance algorithm. You can use a simple `while` loop to implement this behavior.

Set some parameters that will be used in the processing loop. You can modify these values for different behavior.

```
spinVelocity = 0.6;        % Angular velocity (rad/s)
forwardVelocity = 0.1;     % Linear velocity (m/s)
backwardVelocity = -0.02;  % Linear velocity (reverse) (m/s)
distanceThreshold = 0.6;   % Distance threshold (m) for turning
```

Run a loop to move the robot forward and compute the closest obstacles to the robot. When an obstacle is within the limits of the `distanceThreshold`, the robot turns. This loop stops after 20 seconds of run time. CTRL+C (or Control+C on the Mac) also stops this loop.

```
  tic;
  while toc < 20
```

```
% Collect information from laser scan
scan = receive(laser);
plot(scan);
data = readCartesian(scan);
x = data(:,1);
y = data(:,2);
% Compute distance of the closest obstacle
dist = sqrt(x.^2 + y.^2);
minDist = min(dist);
% Command robot action
if minDist < distanceThreshold
    % If close to obstacle, back up slightly and spin
    velmsg.Angular.Z = spinVelocity;
    velmsg.Linear.X = backwardVelocity;
else
    % Continue on forward path
    velmsg.Linear.X = forwardVelocity;
    velmsg.Angular.Z = 0;
end
send(robot,velmsg);
end
```
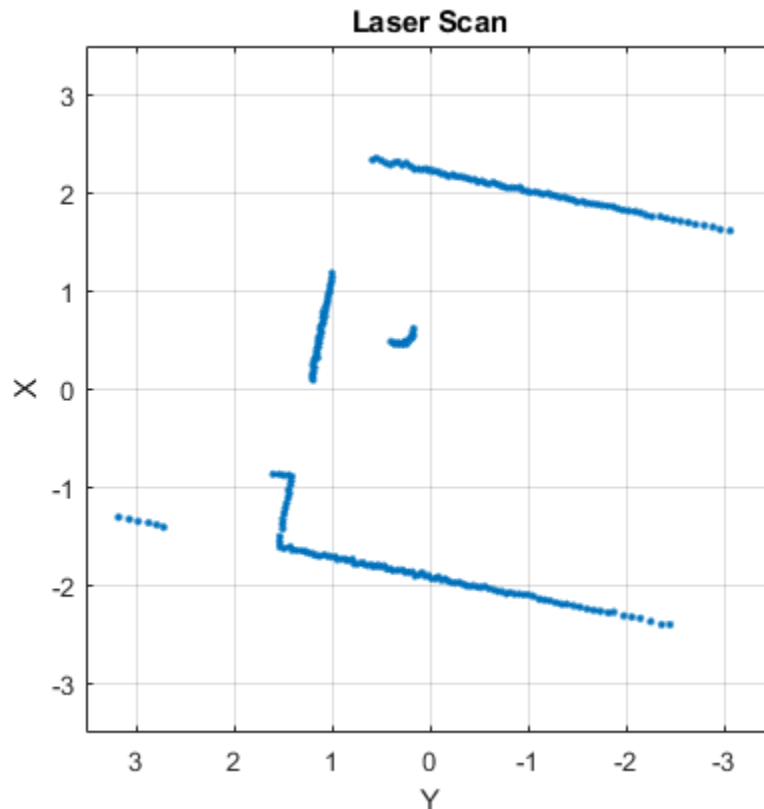

Laser Scan

Disconnect from the Robot

Clear the workspace of publishers, subscribers, and other ROS related objects when you are finished with them.

```
clear
```

Use `rosshutdown` once you are done working with the ROS network. Shut down the global node and disconnect from the TurtleBot.

`rosshutdown`

Shutting down global node /matlab_global_node_48605 with NodeURI http://192.168.203.1:60009/

More Information

The laser scan has a minimum range at which it no longer sees objects in its way. That minimum is somewhere around 0.12 meters from the lidar.

The laser scan cannot detect glass walls. Following is an image from the camera:
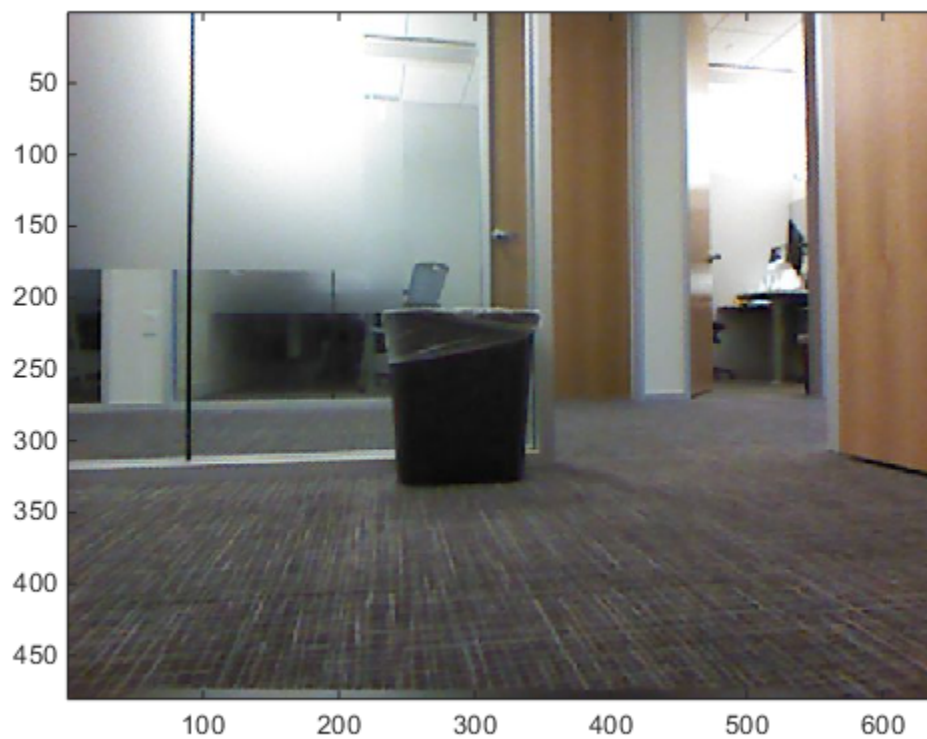
Here is the corresponding laser scan:

Laser Scan

The trash can is visible, but you cannot see the glass wall. When you use the TurtleBot in areas with windows or walls that the TurtleBot might not be able to detect, be aware of the limitations of the laser scan.

Next Steps

Refer to the next example: "Control the TurtleBot with Teleoperation" on page 1-141

Control the TurtleBot with Teleoperation

This example shows keyboard control of the TurtleBot® through the use of the `ExampleHelperTurtleBotCommunicator` class. The instructions describe how to set up the object and how to start the keyboard control. Instructions on how to use keyboard control are displayed when the function is launched. To change parameters of the function, edit the `exampleHelperTurtleBotKeyboardControl` function or the `ExampleHelperTurtleBotKeyInput` class. For an introduction to using the TurtleBot with MATLAB®, see the getting started examples ("Get Started with a Real TurtleBot" on page 1-56 or "Get Started with Gazebo and a Simulated TurtleBot" on page 1-102)

Prerequisites: "Communicate with the TurtleBot" on page 1-130, "Explore Basic Behavior of the TurtleBot" on page 1-135

Hardware Support Package for TurtleBot

This example gives an overview of working with a TurtleBot using its native ROS interface. The ROS Toolbox™ Support Package for TurtleBot based Robots provides a more streamlined interface to TurtleBot2 hardware.

To install the support package, open **Add-Ons > Get Hardware Support Packages** on the MATLAB **Home** tab and select **ROS Toolbox Support Package for TurtleBot based Robots**. Alternatively, use the `rosAddons` command.

Connect to the TurtleBot

Make sure you have a TurtleBot running either in simulation through Gazebo® or on real hardware. Refer to "Get Started with Gazebo and a Simulated TurtleBot" on page 1-102 or "Get Started with a Real TurtleBot" on page 1-56 for the startup procedure. If you are using simulation, `Gazebo Office` is good for exploring.

Initialize ROS. Connect to the TurtleBot by replacing `ipaddress` with the IP address of the TurtleBot.

```
ipaddress = "http://192.168.178.132:11311";
rosinit(ipaddress)
```

```
Initializing global node /matlab_global_node_45834 with NodeURI http://192.168.178.1:50976/
```

If you are working with real TurtleBot2 hardware, make sure that you start the Kinect® camera. Run the following in a terminal on the TurtleBot:

```
roslaunch turtlebot_bringup 3dsensor.launch
```

There may be some functionality, sensor, and topic name differences between TurtleBot versions. Be sure to check which version is being used and the expected topics when controlling it.

```
turtleBotVersion = 3; % Gazebo Office world uses TurtleBot3 Burger model
```

Subscribe to the odometry and laser scan topics and make sure that you can receive messages on these topics.

```
handles.odomSub = rossubscriber("/odom","BufferSize",25);
receive(handles.odomSub,3);
handles.laserSub = rossubscriber("/scan","BufferSize",5);
receive(handles.laserSub,3);
```

Create a publisher for controlling the robot velocity.

```
if turtleBotVersion == 3
    velTopic = "/cmd_vel";
else
    velTopic = "/mobile_base/commands/velocity";
end
handles.velPub = rospublisher(velTopic);
```

Control the Robot

Run the `exampleHelperTurtleBotKeyboardControl` function, which allows you to control the TurtleBot with the keyboard. Mark the inserted code example as code (highlight and press 'Alt +Enter') to execute the function.

```
exampleHelperTurtleBotKeyboardControl(handles);
```

Following are samples of the Command Window, the world plot, and the Gazebo world after some keyboard teleoperation by the user:

i = Forward

j = Left   l = Right

k = Backward

q = Quit
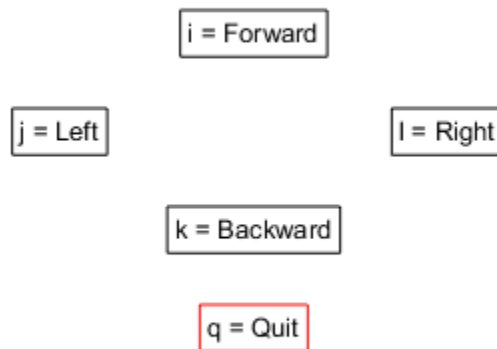
Keep this figure in scope to give commands
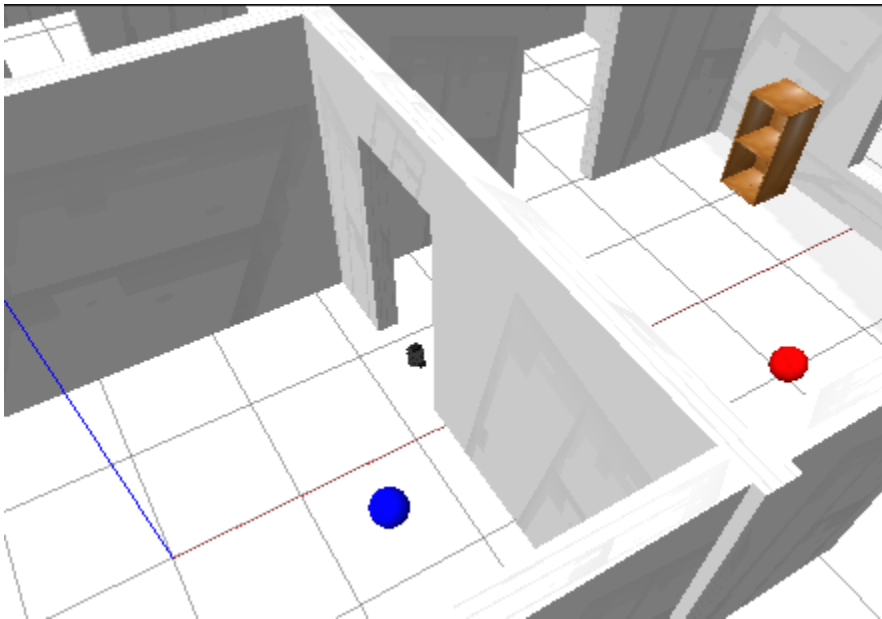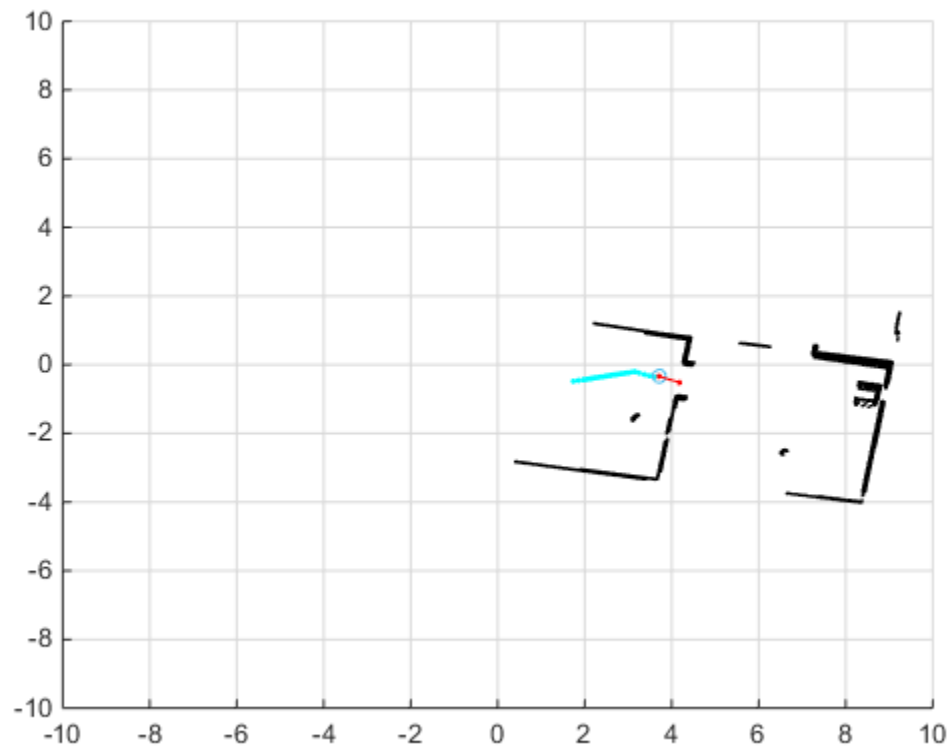
If you move the TurtleBot too quickly, the obstacle plotting can become messy because of relative inaccuracies in the odometry topic at high speeds. Here is an example of a messy world plot:
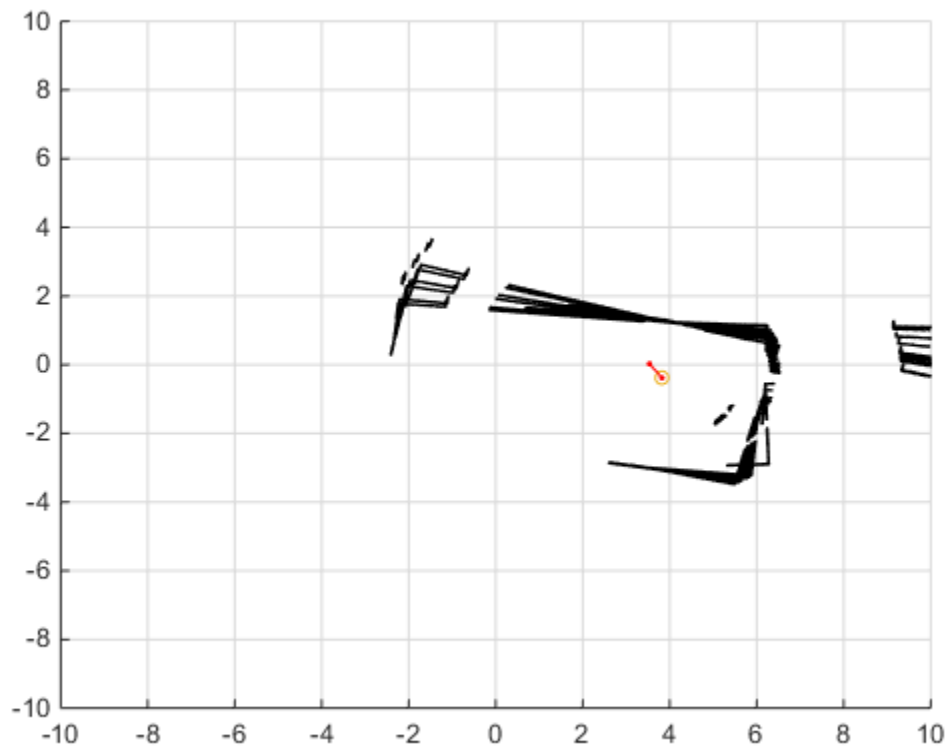
A sample plot of a real TurtleBot moving around an office space is shown:
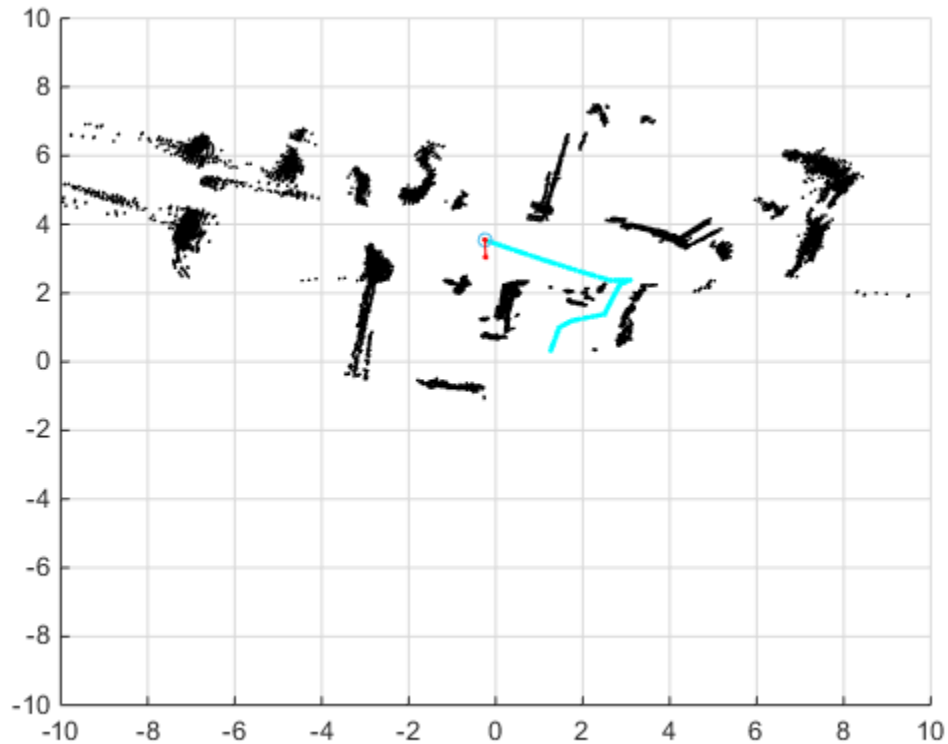
Disconnect from the Robot

Once you have exited the function by pressing q, clear the publishers and subscribers on the host.

```
clear
```

Use rosshutdown once you are done working with the ROS network. Shut down the global node and disconnect from the TurtleBot.

```
rosshutdown
```

Shutting down global node /matlab_global_node_45834 with NodeURI http://192.168.178.1:50976/

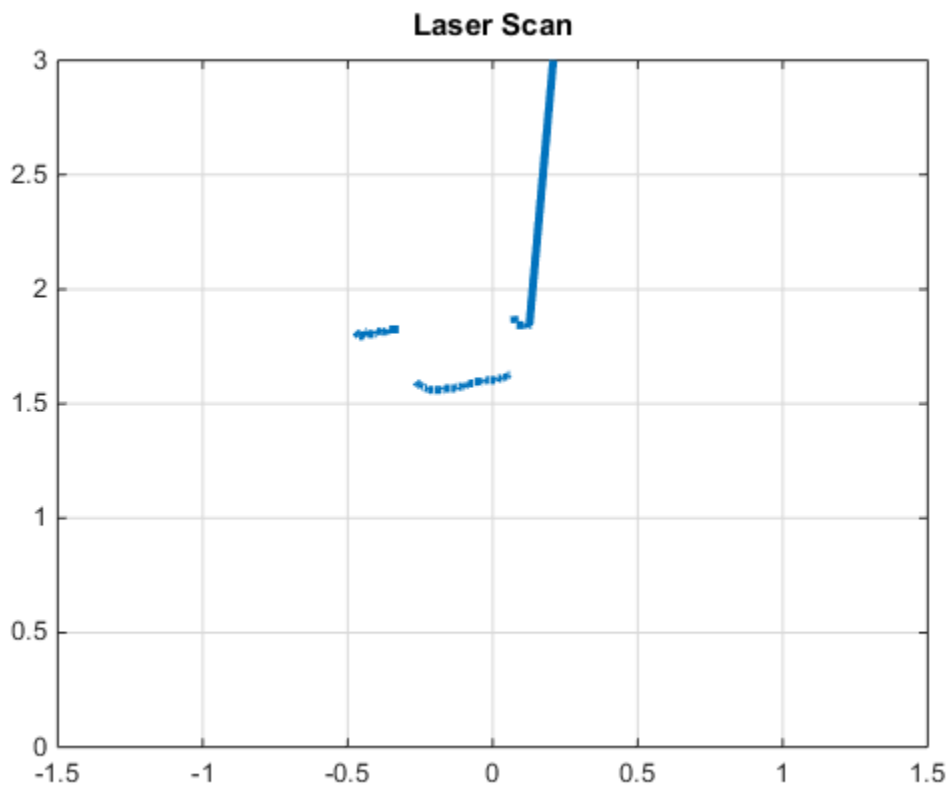**Next Steps**

• Refer to the next example: "Obstacle Avoidance with TurtleBot and VFH" on page 1-146

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 `controllerVFH` (Navigation Toolbox) object 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" on page 1-102 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',11311)
```

Initializing global node /matlab_global_node_41200 with NodeURI http://192.168.203.1:54964/

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. Set the `UseLidarScan` property to `true`. Specify algorithm properties for robot specifications. Set target direction to `0` in order to drive straight.

```
vfh = controllerVFH;
vfh.UseLidarScan = true;
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 `rateControl` (Navigation Toolbox), which can track the timing of your loop. This object can be used to control the rate the loop operates as well.

```
rate = rateControl(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. Create a `lidarScan` object by specifying the ranges and angles. 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);

    % Create a lidarScan object from the ranges and angles
        scan = lidarScan(ranges,angles);

    % Call VFH object to computer steering direction
    steerDir = vfh(scan, 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 Navigation 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_41200 with NodeURI http://192.168.203.1:54964/

Track and Follow an Object

In this example, you explore autonomous behavior that incorporates the Kinect® camera. This algorithm involves the TurtleBot® looking for a blue ball and then staying at a fixed distance from the ball. You incorporate safety features, such as bump and cliff sensing.

Running this example requires the Image Processing Toolbox™.

Prerequisites: "Communicate with the TurtleBot" on page 1-130, "Explore Basic Behavior of the TurtleBot" on page 1-135, "Control the TurtleBot with Teleoperation" on page 1-141, "Obstacle Avoidance with TurtleBot and VFH" on page 1-146

Hardware Support Package for TurtleBot

This example gives an overview of working with a TurtleBot using its native ROS interface. The **ROS Toolbox™ Support Package for TurtleBot based Robots** provides a more streamlined interface to TurtleBot. It allows you to:

Acquire sensor data and send control commands without explicitly calling ROS commands

Communicate transparently with a simulated robot in Gazebo or with a physical TurtleBot

To install the support package, open **Add-Ons > Get Hardware Support Packages** on the MATLAB® **Home** tab and select **ROS Toolbox™ Support Package for TurtleBot based Robots**. Alternatively, use the `roboticsAddons` (Robotics System Toolbox) command.

Connect to the TurtleBot

Make sure you have a TurtleBot running either in simulation through Gazebo® or on real hardware. Refer to "Get Started with Gazebo and a Simulated TurtleBot" on page 1-102 or "Get Started with a Real TurtleBot" on page 1-56 for the startup procedure. If you are using hardware, find a blue ball to use for tracking. If you are using Gazebo®, the blue ball must be in the world in front of the robot (make sure that you are using **Gazebo Office** world).

Initialize ROS. Connect to the TurtleBot by replacing `ipaddress` with the IP address of the TurtleBot

```
ipaddress = '192.168.2.150';
rosinit(ipaddress,11311)
```

Initializing global node /matlab_global_node_54170 with NodeURI http://192.168.2.1:52893/

Make sure that you have started the Kinect camera if you are working with real TurtleBot hardware. The command to start the camera is:

`roslaunch turtlebot_bringup 3dsensor.launch.`

You must enter this in a terminal on the TurtleBot.

Create subscribers for the color camera, the cliff sensor, and the bumper sensor.

Create publishers for emitting sound and for controlling the robot velocity messages.

`handles.colorImgSub = exampleHelperTurtleBotEnableColorCamera;`

Successfully Enabled Camera (raw image)

```
useHardware = exampleHelperTurtleBotIsPhysicalRobot;
if useHardware
    handles.cliffSub = rossubscriber('/mobile_base/events/cliff','BufferSize', 5);
    handles.bumpSub = rossubscriber('/mobile_base/sensors/bumper_pointcloud', 'BufferSize', 5);
    handles.soundPub = rospublisher('/mobile_base/commands/sound', 'kobuki_msgs/Sound');
    handles.velPub = rospublisher('/mobile_base/commands/velocity');
else
    % Cliff sensor, bumper sensor and sound emitter are only present in
    % real TurtleBot hardware
    handles.cliffSub = [];
    handles.bumpSub = [];
    handles.soundPub = [];
    handles.velPub = rospublisher('/cmd_vel');
end
```

Tune the Blue Ball Detection

Set the parameters for image filtering. Add them to a data structure that will be used in the algorithm.

```
blueBallParams.blueMax = 120;   % Maximum permissible deviation from pure blue
blueBallParams.darkMin = 30;    % Minimum acceptable darkness value
```

Try to visualize the ball to make sure that the ball-finding parameters can locate it. Run the `exampleHelperTurtleBotFindBlueBall` function to see if a circle is found. If so, `c` and `m` are assigned values. `ball` is a binary image created by applying blueness and darkness filters on the image. View `ball` to see if the blue ball was properly isolated:

```
latestImg = readImage(handles.colorImgSub.LatestMessage);
[c,~,ball] = exampleHelperTurtleBotFindBlueBall(latestImg,blueBallParams,useHardware);
```

Use this example helper to display the real and binary image in a figure and plot a red plus at the center of the ball.

```
exampleHelperTurtleBotPlotObject(latestImg,ball,c);
```
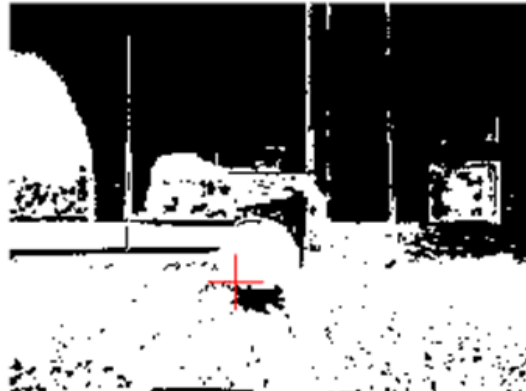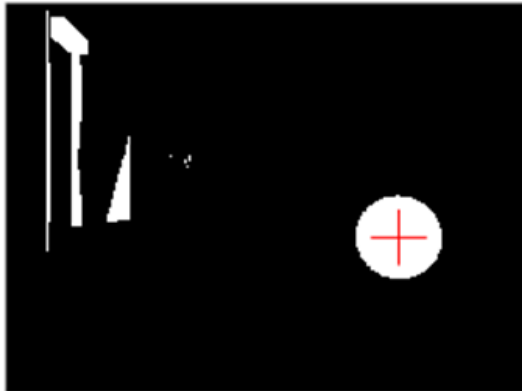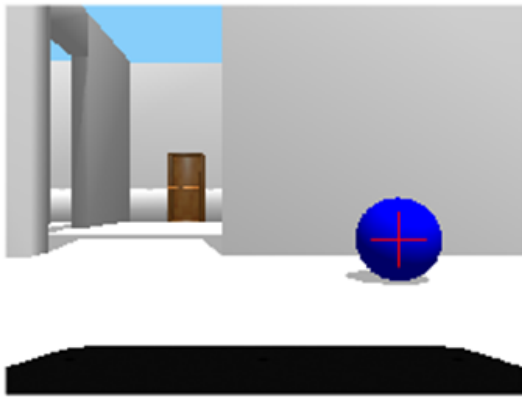
If the ball is not found, try increasing or decreasing `blueBallParams.blueMax` and `blueBallParams.darkMin`. View the plot again until the ball is found. This method is a good way to fine tune the ball-finding algorithm before using the controller.

In Gazebo, the parameters used might not find the ball, because the threshold values are too generous. The Gazebo image (left figures) includes parts of the wall and other objects in the white space. The real image (right figures) looks very saturated with white. Try changing the parameters so that they are more restrictive:

```
blueBallParams.blueMax = 200; % Maximum permissible deviation from pure blue
blueBallParams.darkMin = 220; % Minimum acceptable darkness value
latestImg = readImage(handles.colorImgSub.LatestMessage);
[c,~,ball] = exampleHelperTurtleBotFindBlueBall(latestImg,blueBallParams,useHardware);
```

Use this example helper to display the figures.

```
exampleHelperTurtleBotPlotObject(latestImg,ball,c);
```
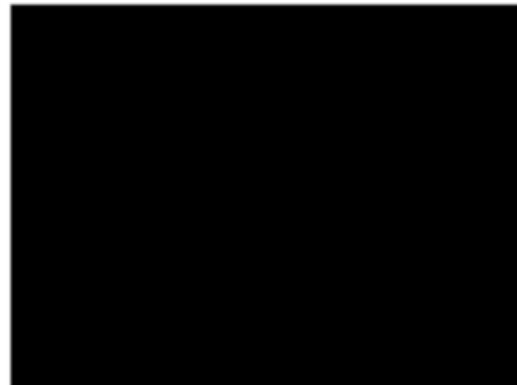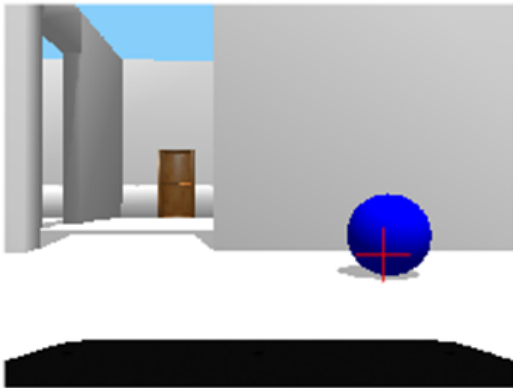
Now the parameters are too restrictive. Part of the ball does not even show up in the Gazebo image, and you see nothing in the real image. If you tune the parameters further you can find a middle ground. In Gazebo, the following parameters should work well. With hardware, ambient lighting might require you to spend more time fine tuning the parameters.

```
blueBallParams.blueMax = 30; % Maximum permissible deviation from pure blue
blueBallParams.darkMin = 90; % Minimum acceptable darkness value
latestImg = readImage(handles.colorImgSub.LatestMessage);
[c,~,ball] = exampleHelperTurtleBotFindBlueBall(latestImg,blueBallParams, useHardware);
```

Use this example helper to display the figures.

```
exampleHelperTurtleBotPlotObject(latestImg,ball,c);
```
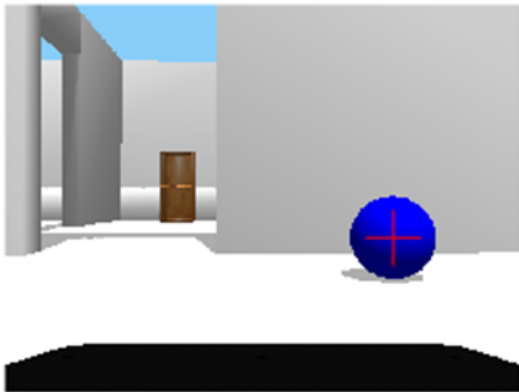
Tuning the color thresholds is challenging when compared to tuning them in a simulated environment like Gazebo.

After you have fine-tuned the parameters, add them to the `handles` object, which will be used by the ball tracking algorithm.

```
handles.params = blueBallParams;
```

Test Fixed-Distance Controller

Set controller gains for the TurtleBot. The TurtleBot uses a PID controller to stay at a constant distance from the ball.

The first set of controller gains is good for a TurtleBot in Gazebo. The second set is good for a TurtleBot in real hardware. Adjust the gains as you see fit.

Here is a compact way to assign the struct values.

Effective gains for Gazebo simulation:

```
gains.lin = struct('pgain',1/100,'dgain',1/100,'igain',0,'maxwindup',0','setpoint',0.65);
gains.ang = struct('pgain',1/400,'dgain',1/500,'igain',0,'maxwindup',0','setpoint',0.5);
```

Effective gains for TurtleBot hardware:

```
gains.lin = struct('pgain',1/100,'dgain',1/1000,'igain',0,'maxwindup',0','setpoint',0.75);
gains.ang = struct('pgain',1/100,'dgain',1/3000,'igain',0,'maxwindup',0','setpoint',0.5);
```

Make sure to add the `gains` struct to the `handles` variable.

```
handles.gains = gains;
```

Define a timer to execute the ball tracking behavior through the callback. Define the stop function to shut down ROS. Include the handles in the callback function for the timer:

```
timer2 = timer('TimerFcn',{@exampleHelperTurtleBotTrackingTimer,handles,useHardware},'Period',0.1
timer2.StopFcn = {@exampleHelperTurtleBotStopCallback};
```

Start the timer using the following command. You see the TurtleBot begin to move around the world, searching for the ball. When it finds it in the Kinect image, the robot will use the controller to stay at a fixed distance.

```
start(timer2);
pause(1);
```

The bump sensor does not activate in simulation, so the TurtleBot might not recover when it hits a wall.

If you want to move the blue ball around, use the following commands to apply a force:

```
g = ExampleHelperGazeboCommunicator();
    ballhandle = ExampleHelperGazeboSpawnedModel('unit_sphere_1',g)
    duration = 2;
    forceVector = [0 4 0];
    applyForce(ballhandle,'link',duration,forceVector)
```

If you want to further explore Gazebo control of the simulation refer to "Add, Build, and Remove Objects in Gazebo" on page 1-108.

Stop Robot Motion

To stop the timer and autonomous behavior, use the following command:

```
stop(timer2);
```

If the timer is cleared from the workspace before it is stopped, you must delete it another way. To stop all timers (even timers in the background) execute the following command:

```
delete(timerfindall)
```

Clear the workspace of publishers, subscribers, and other ROS related objects when you are finished with them

```
clear
```

More Information

NOTE: Code in this section is not for MATLAB command line execution

In this example, the organization of supporting files allows you great flexibility in customizing and re-purposing the code. You can alter the ball-finding parameters and the controller gains by changing values in the `handles` struct. This example incorporates a `timer` that manages all aspects of the control algorithm. This timer is the `exampleHelperTurtleBotTrackingTimer`. This timer has Name-Value pairs of `Period` and `ExecutionMode` that are set to determine how often the timer callback is called. Additionally, the stop callback is used. You can incorporate additional callback functions if you want.

The handles passed into the timer include `params` for ball-finding and `gains` for the controller.

The structure of `exampleHelperTurtleBotTrackingTimer` is simple. It is a basic state machine with some initialization steps. The initialization function determines which tracking algorithm and which controller to use when not in a cliff or bumper recovery state. The function is:

```
function [objectTrack, imgControl] = initControl()
    % INITCONTROL - Initialization function to determine which control
    % and object detection algorithms to use
    objectTrack = @exampleHelperTurtleBotFindBlueBall;
    imgControl = @exampleHelperTurtleBotPointController;
```

In this example the tracking function is `exampleHelperTurtleBotFindBlueBall` and the controller is `exampleHelperTurtleBotPointController` You can replace this function and controller with any user-defined functions that have the same input and output argument structure. The input arguments for `exampleHelperTurtleBotFindBlueBall` are a color image and a struct of ball-finding parameters. The output arguments are a center, magnitude, and binary image of the sought object. The input arguments for `exampleHelperTurtleBotPointController` are object center, magnitude (though magnitude is not used in the example), image size, and controller gains (a struct). The output arguments are linear and angular velocities.

The basic state machine used in `exampleHelperTurtleBotTrackingTimer` is:

```
switch state
        case ExampleHelperTurtleBotStates.Seek
            % Object-finding state
            [center, scale] = findObject(handles.Tbot.ImColor,handles.params);
            % Wander if no circle is found, target the circle if it exists
            if isempty(center)
                [linearV, angularV] = exampleHelperTurtleBotWanderController();
            else
                [linearV, angularV] = imageControl(center, scale, size(handles.Tbot.ImColor),hand
                setSound(handles.Tbot,2);
            end
            state = ExampleHelperTurtleBotStates.Seek;
        case ExampleHelperTurtleBotStates.Bumper
            % Bumper contact state
        case ExampleHelperTurtleBotStates.Spin
            % Spin state
        case ExampleHelperTurtleBotStates.Cliff
            % Cliff avoidance
end
```
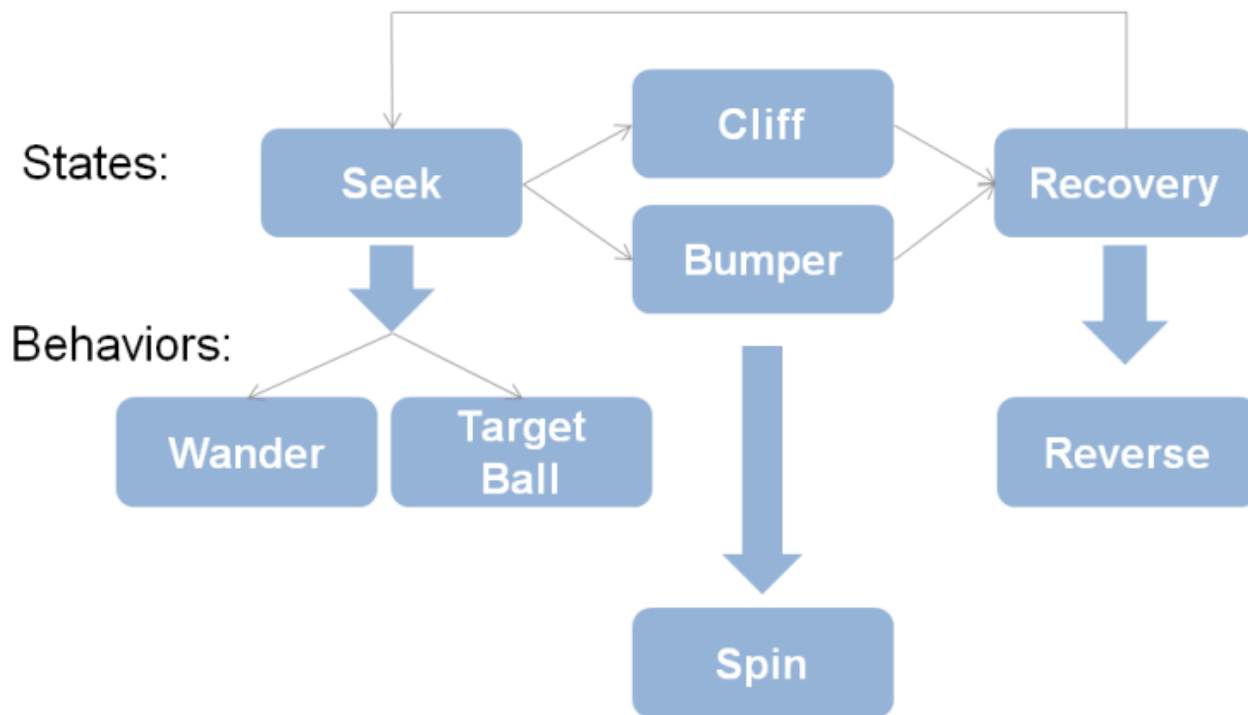
You can add or remove cases from the state machine. If you want to change the state names, use the `ExampleHelperTurtleBotStates` class.

The ball-finding algorithm is modular and alterable. It uses two image filters (one on darkness and one on blueness) masked together to isolate the blue ball. You can change the masks to find a red or green ball instead. If you want to explore other forms of shape-tracking, the basic workflow remains the same.

The blue channel is isolated (with some scaling factors) and a threshold is applied to produce a binary image mask.

```
blueImg = img(:,:,1)/2 + img(:,:,2)/2 - img(:,:,3)/2;
blueThresh = blueImg < params.blueMax;
```

These commands isolate the inverse of the blue (with different scaling) and emphasize darkness. A threshold is applied.

```
darkIso = -img(:,:,1)/2 - img(:,:,2)/2 + 3*img(:,:,3) - 2*rgb2gray(img);
darkThresh = darkIso > params.darkMin;
```

Mask the two binary images together to isolate the dark blue ball.

```
ball1 = blueThresh & darkThresh;
```

The constants and scaling factors on the image are user-determined to isolate a specific color. You can experiment with various combinations.

You can also find contiguous regions in the filtered image using `regionprops`, which is part of the Image Processing Toolbox.

```
s = regionprops(ball1, {'Centroid','Area','EquivDiameter'});
```

There are additional steps to find the ball from this region, which you can find in `exampleHelperTurtleBotFindBlueBall`.

The `exampleHelperTurtleBotPointController` function uses the `ExampleHelperPIDControl` class to keep a specified point (in this case the location of the center of the ball) at an exact location within the image.

The modularity and flexibility of the example code allows you to experiment with your own algorithms and functions.

ROS 2 Featured Examples

Get Started with ROS 2

Robot Operating System 2 (ROS 2) is the second version of ROS, which is a communication interface that enables different parts of a robot system to discover, send, and receive data. MATLAB® support for ROS 2 is a library of functions that allows you to exchange data with ROS 2 enabled physical robots or robot simulators such as Gazebo®. ROS 2 is built on Data Distribution Standard (DDS) which is an end-to-end middleware that provides features such as discovery, serialization and transportation. These features align with the design principles of ROS 2 such as *distributed discovery* and control over different "*Quality of Service*" options for transportation. DDS uses Real Time Publish-Subscribe (RTPS) protocol which provides communication over unreliable network protocols such as UDP. For more information, see RTPS.

This example shows how to:

- Set up ROS 2 within MATLAB
- Get information about capabilities in a ROS 2 network
- Get information about ROS 2 messages

To learn about ROS, see "Get Started with ROS" on page 1-2.

ROS 2 Terminology

- A *ROS 2 network* comprises different parts of a robot system (such as a planner or a camera interface) that communicate over ROS 2 network. The network can be distributed over several machines.
- A *ROS 2 node* is an entity that contains a collection of related ROS 2 capabilities (such as publishers and subscribers). A ROS 2 network can have many ROS 2 nodes.
- *Publishers* and *subscribers* are different kinds of ROS 2 entities that process data. They exchange data using *messages*.
- A publisher sends messages to a specific *topic* (such as "odometry"), and subscribers to that topic receive those messages. There can be multiple publishers and subscribers associated with a single topic.
- A *Domain* is the physical segmentation of network. It is identified by a unique integer value known as *Domain ID.* By default the *Domain ID* is 0.
- Every node in ROS 2 network on creation advertises its presence to other nodes in the same *Domain ID* only.
- *ROS 2* network is built on Data Distribution Standard (DDS) which makes it possible to connect multiple nodes across distributed network.
- *RTPS* (Real Time publisher-subscriber) protocol provides ROS 2 network with capabilities to send messages in unreliable network conditions.
- ROS 2 offers variety of *Quality of Service (QoS)* policies that allow you to tune your communication between nodes. For more information, see "Manage Quality of Service Policies in ROS 2" on page 2-21.

For more information, see Robot Operating System2 (ROS 2) and the Concepts section on the ROS 2 website.

Initialize ROS 2 Network

Unlike ROS, ROS 2 does not require initialization in MATLAB. The ROS 2 network automatically starts with creation of nodes.

Use `ros2node` to create a node.

```
test1 = ros2node("/test1")

test1 =
  ros2node with properties:

    Name: '/test1'
      ID: 0
```

Use `ros2 node list` to see all nodes in the ROS 2 network.

```
ros2 node list
```

```
/test1
```

Use `clear` to shutdown the node in ROS 2 network.

```
clear test1
```

Use `exampleHelperROS2CreateSampleNetwork` to populate the ROS network with three additional nodes with sample publishers and subscribers.

```
exampleHelperROS2CreateSampleNetwork
```

Use `ros2 node list` again, and observe that there are three new nodes, `node_1`, `node_2`, and `node_3`).

```
ros2 node list
```

```
/node_1
/node_2
/node_3
```

A visual representation of the current state of the ROS 2 network is shown below. Use it as a reference when you explore this sample network in the remainder of the example.
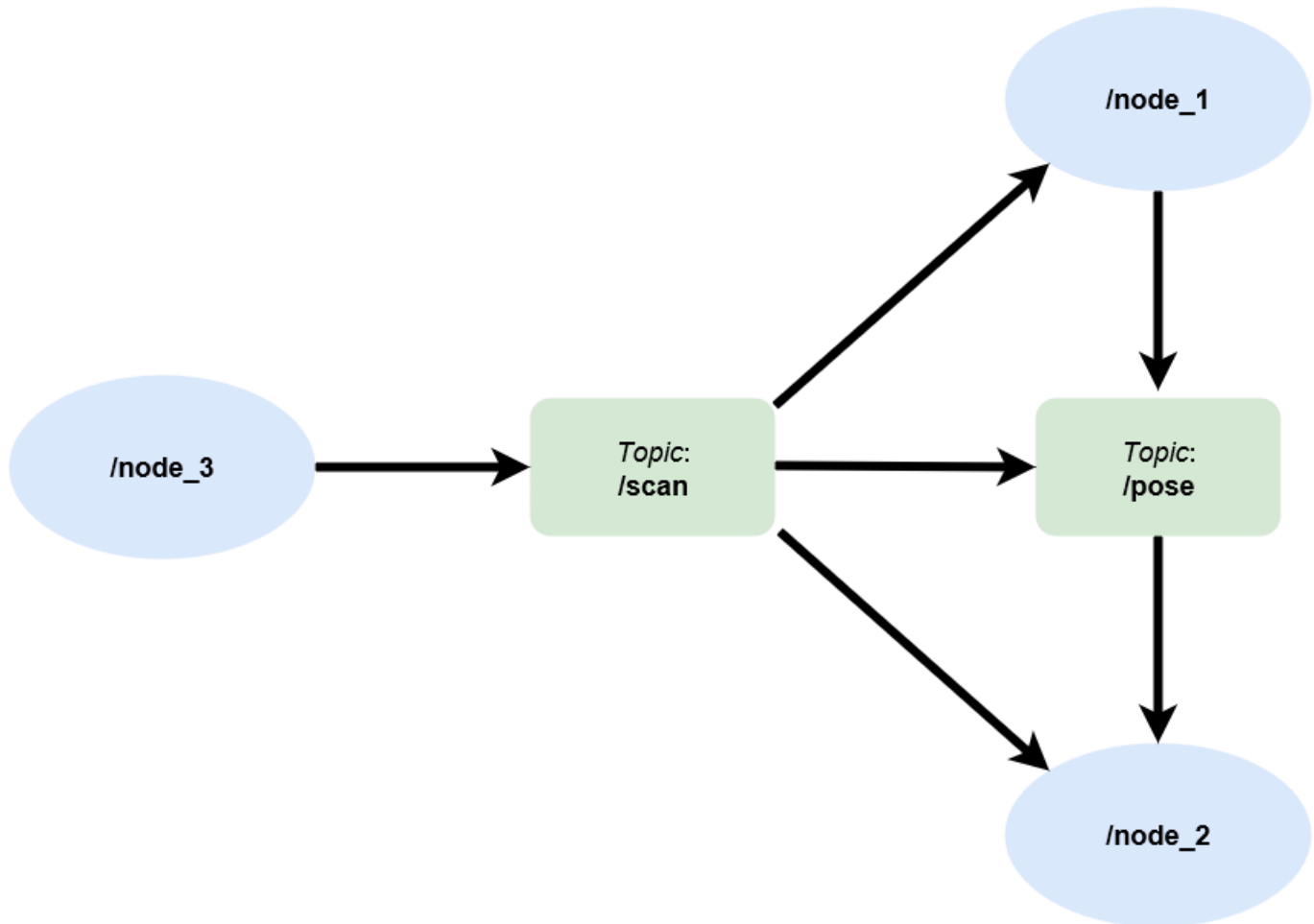
Topics

Use `ros2 topic list` to see available topics in the ROS 2 network. Observe that there are four active topics: `/pose`, `/clock`, `/parameter_events` and `/scan`.The topics `/clock` and `/parameter_events` are global topics which always present in the ROS 2 network. Topic `/clock` is used for synchronization whereas topic `/parameter_events` is used by nodes to monitor or change parameters in the network. The other two topics `/scan` and `/pose` were created as part of the sample network.

`ros2 topic list`

```
/parameter_events
/pose
/scan
```

Each topic is associated with a m*essage type*. Use `ros2 topic list -t` to see the message type of the topics.

`ros2 topic list -t`

```
         Topic                      MessageType
  _____      _____
```

```
{'/parameter_events'}    {'rcl_interfaces/ParameterEvent'}
{'/pose'            }    {'geometry_msgs/Twist'          }
{'/scan'            }    {'sensor_msgs/LaserScan'        }
```

Messages

Publishers and subscribers use ROS 2 messages to exchange information. Each ROS 2 message has an associated message type that defines the datatypes and layout of information in that message. For more information, see "Work with Basic ROS 2 Messages" on page 2-11.

Use `ros2 msg show` to view the properties of a message type. The `geometry_msgs/Twist` message type has two properties, `Linear` and `Angular`. Each property is a message of type `geometry_msgs/Vector3`, which in turn has three properties of type `double`.

```
ros2 msg show geometry_msgs/Twist

# This expresses velocity in free space broken into its linear and angular parts.

Vector3  linear
Vector3  angular
```

```
ros2 msg show geometry_msgs/Vector3

# This represents a vector in free space.

float64 x
float64 y
float64 z
```

Use `ros2 msg list` to see the full list of message types available in MATLAB.

Disconnect From ROS 2 Network

Use `exampleHelperROS2ShutDownSampleNetwork` to remove the sample nodes, publishers, and subscribers from the ROS 2 network. To remove your own nodes, use `clear` with the node, publisher, or subscriber object.

```
exampleHelperROS2ShutDownSampleNetwork
```

Next Steps

- "Connect to a ROS 2 Network" on page 2-6

Connect to a ROS 2 Network

A ROS 2 network consists of a multiple *ROS 2 nodes*. Unlike ROS where the ROS master facilitates the communication by keeping track of all active ROS entities, ROS 2 is based on Data Distribution Standard (DDS) which is an end-to-end middleware that provides features such as discovery, serialization, and transportation. These features align with the design principles of ROS 2 such as *distributed discovery* and control over different "*Quality of Service*" options for transportation.

To connect to a ROS 1 network, see "Connect to a ROS Network" on page 1-7.

When you work with ROS 2, you typically follow these steps:

- *Connect to a ROS 2 network.* To connect to a ROS 2 network, you have to create a ROS 2 node in MATLAB specifying the network domain ID.
- *Exchange Data.* Once connected, MATLAB exchanges data with other ROS 2 nodes in the same domain ID through publishers and subscribers.
- *Disconnect from the ROS 2 network.* Clearing all references to the nodes, publishers, and subscribers removes MATLAB from the ROS 2 network.

Create a ROS 2 Node in the Default Domain

Use `ros2node` to create a node in the default domain, which uses the ID of 0. Nodes communicate with other nodes in the same domain, and are unaware of nodes in other domains.

```
defaultNode = ros2node("/default_node")

defaultNode =
  ros2node with properties:

    Name: '/default_node'
      ID: 0
```

Use `clear` to remove the reference to the node, allowing it to be deleted from the ROS 2 network.

```
clear defaultNode
```

Create a ROS 2 Node on a Different Domain

To create a node in non-default domain, explicitly specify the domain ID as a second input argument to `ros2node`. Below `newDomainNode` is created in the domain specified by ID 25.

```
newDomainNode = ros2node("/new_domain_node",25)

newDomainNode =
  ros2node with properties:

    Name: '/new_domain_node'
      ID: 25
```

To view network information on a specific domain, provide the ID as a parameter to the `ros2` function. The following command displays all nodes with domain ID 25.

```
ros2("node","list","DomainID",25)

/new_domain_node
```

Change Default Domain ID

If the domain ID is not provided explicitly to the node or `ros2` command, they use the value of the `ROS_DOMAIN_ID` environment variable by default. Use `getenv` to see the current value. If that environment variable is unset, or not set to a valid value, the default domain ID of `0` will be used.

```
getenv("ROS_DOMAIN_ID")

ans =

  0x0 empty char array
```

You can set `ROS_DOMAIN_ID` using the `setenv` command.

```
setenv("ROS_DOMAIN_ID","25")
envDomainNode = ros2node("/env_domain_node")

envDomainNode =
  ros2node with properties:

    Name: '/env_domain_node'
      ID: 25
```

The ros2 function provides information on the network specified by that environment variable. Use `ros2 node list` to view nodes with domain ID 25.

```
ros2 node list

/env_domain_node
/new_domain_node
```

Reset the `ROS_DOMAIN_ID` to default.

```
setenv("ROS_DOMAIN_ID","")
```

Communication in ROS 2 Network

To connect to an existing ROS 2 network, create a node in the desired domain. The ROS 2 network automatically detects any new nodes created in the same domain in a mechanism called *discovery*.

Upon starting, each node in ROS 2 advertises its presence to other nodes in the same domain. The other nodes respond to this advertisement by providing their information to the new node. Nodes with communication objects like publishers and subscribers establish connections with other nodes if they have corresponding objects with compatible Quality of Service (QoS) settings. For more information on QoS settings, see "Manage Quality of Service Policies in ROS 2" on page 2-21.

Discovery is an ongoing process, which enables new nodes to join the network as they are created. Each node is monitoring the ROS 2 network and act similarly to the ROS master in a ROS network. Nodes also advertise their absence to other nodes when they go offline.

Advertising (STEP 1)

Begin Communication

Ongoing Communication
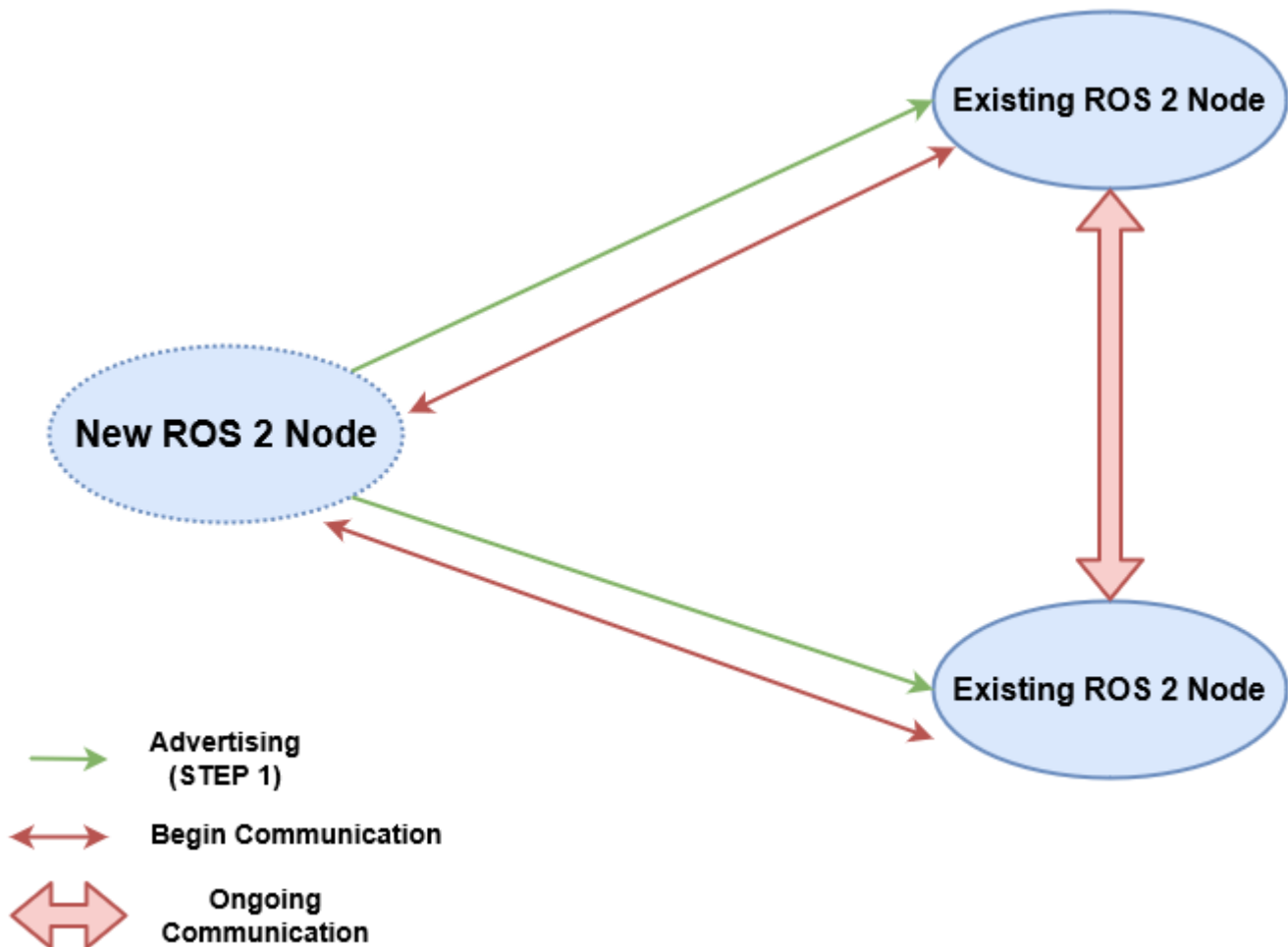
The new ROS 2 node sends its advertisement to the existing nodes. The existing nodes respond to the advertisement and then set up for ongoing communication.

ROS Communication Outside Subnet

A subnet is a logical partition of an IP network into multiple, smaller network segments. ROS 2 nodes can communicate with other nodes within the same subnet. To detect the nodes present outside the subnet, create a *DEFAULT_FASTRTPS_PROFILE.xml* file to configure the specific DDS implementation MATLAB uses. Add the list of IP address of systems outside of the subnet with which to communicate inside `address` elements. Note that for both systems to communicate, they each must specify the other system's address in their respective *DEFAULT_FASTRTPS_PROFILE.xml* files. Set the `domainId` element to the appropriate value for the network that is used for communication.

Keep this file in the MATLAB *Current Working Directory*. Systems using ROS 2 outside of MATLAB should place this file in the same directory from which the ROS 2 application is run. Below is an example *DEFAULT_FASTRTPS_PROFILES.xml* file.

```
<?xml version="1.0" encoding="UTF-8" ?>
<profiles>
    <participant profile_name="participant_win" is_default_profile="true">
        <rtps>
            <builtin>
```

```
                        <domainId>0</domainId>
                         <initialPeersList>
                             <locator>
                                 <kind>UDPv4</kind>
                                 <address>192.34.17.36</address>
                             </locator>
                             <locator>
                                 <kind>UDPv4</kind>
                                 <address>182.30.45.12</address>
                             </locator>
                             <locator>
                                 <kind>UDPv4</kind>
                                 <address>194.158.78.29</address>
                             </locator>
                         </initialPeersList>
                    </builtin>
                </rtps>
            </participant>
    </profiles>
```

ROS 2 advertises information to the nodes present in the systems with IP addresses listed inside the *DEFAULT_FASTRTPS_PROFILES.xml*. No information from the nodes in the other machine outside the subnet will be received if *DEFAULT_FASTRTPS_PROFILES.xml* is either not present or does not contain the correct IP addresses.
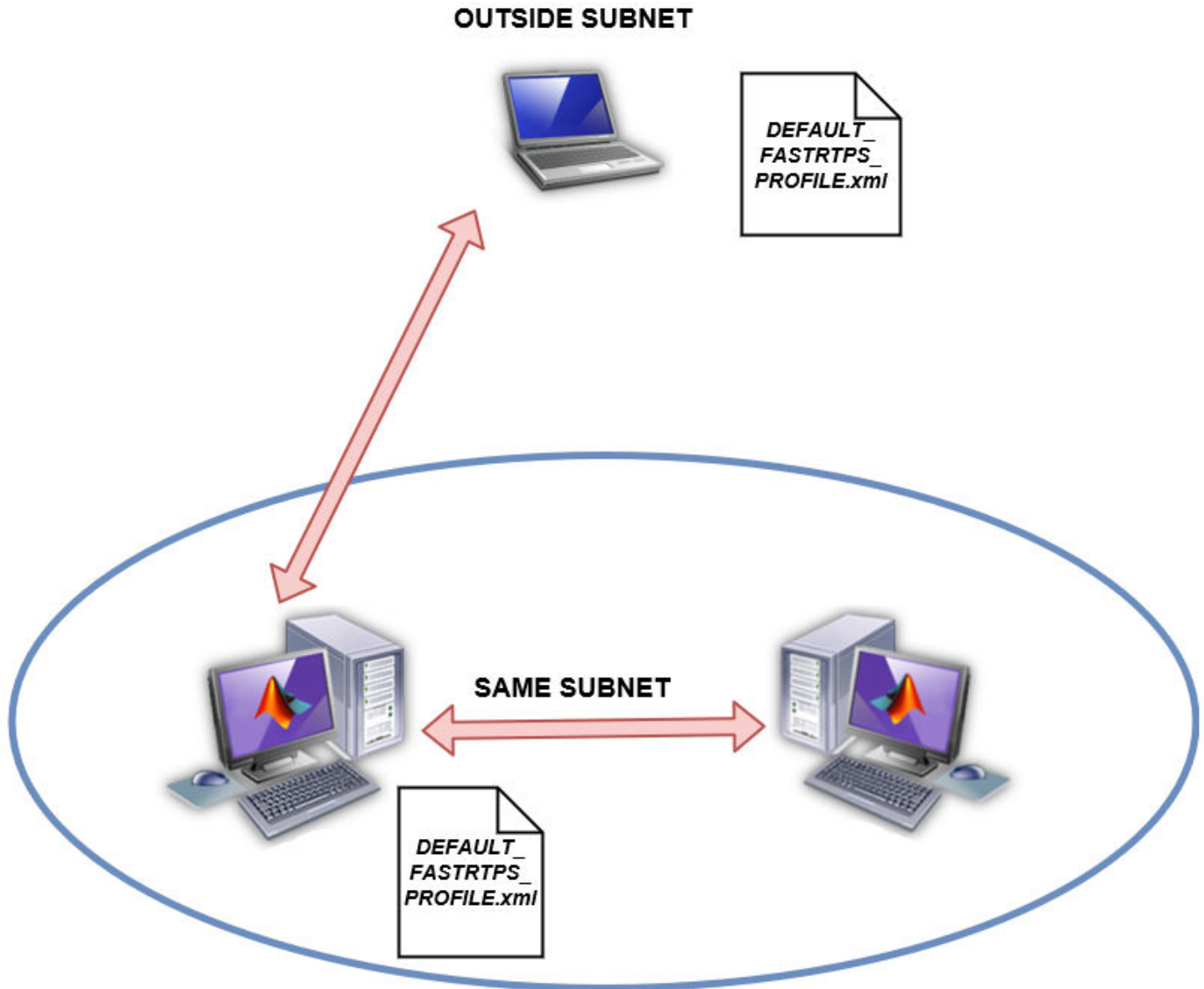
Next Steps

- "Exchange Data with ROS 2 Publishers and Subscribers" on page 2-17

Work with Basic ROS 2 Messages

ROS *messages* are the primary container for exchanging data in ROS 2. Publishers and subscribers exchange data using messages on specified *topics* to carry data between nodes. For more information on sending and receiving messages, see "Exchange Data with ROS 2 Publishers and Subscribers" on page 2-17.

To identify its data structure, each message has a *message type*. For example, sensor data from a laser scanner is typically sent in a message of type `sensor_msgs/LaserScan`. Each message type identifies the data elements that are contained in a message. Every message type name is a combination of a package name, followed by a forward slash /, and a type name:
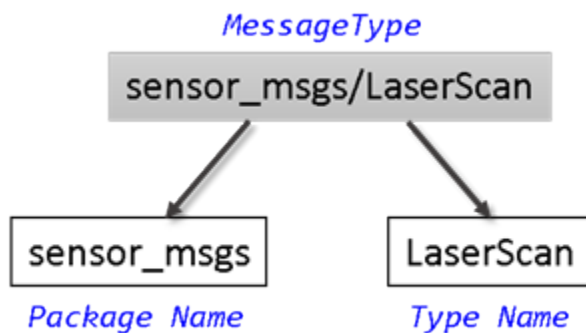
MATLAB® supports many ROS 2 message types that are commonly encountered in robotics applications. This example examines some of the ways to create, inspect, and populate ROS 2 messages in MATLAB.

Prerequisites: "Get Started with ROS 2" on page 2-2, "Connect to a ROS 2 Network" on page 2-6

Find Message Types

Use `exampleHelperROS2CreateSampleNetwork` to populate the ROS 2 network with three nodes and setup sample publishers and subscribers on specific topics.

```
exampleHelperROS2CreateSampleNetwork
```

Use `ros2 topic list -t` to find the available topics and their associated message type.

```
ros2 topic list -t
```

```
        Topic                        MessageType
   _____    _____

   {'/parameter_events'}    {'rcl_interfaces/ParameterEvent'}
```

To find out more about the topic message type, use `ros2message` to create an empty message of the same type. `ros2message` supports tab completion for the message type. To quickly complete message type names, type the first few characters of the name you want to complete, and then press the **Tab** key.

```
scanData = ros2message("sensor_msgs/LaserScan")
```

```
scanData = struct with fields:
            header: [1x1 struct]
```

```
            angle_min: 0
            angle_max: 0
      angle_increment: 0
       time_increment: 0
            scan_time: 0
            range_min: 0
            range_max: 0
               ranges: 0
          intensities: 0
```

The created message, `scanData`, has many fields associated with data that you typically received from a laser scanner. For example, the minimum sensing distance is stored in the `range_min` property and the maximum sensing distance in `range_max` property.

You can now delete the created message.

```
clear scanData
```

To see a complete list of all message types available for topics and services, use `ros2 msg list`.

Explore Message Structure and Get Message Data

ROS 2 messages are represented as structures and the message data is stored in fields. MATLAB provides convenient ways to find and explore the contents of messages.

Use `ros2 msg show` to view the definition of the message type.

```
ros2 msg show geometry_msgs/Twist

# This expresses velocity in free space broken into its linear and angular parts.

Vector3  linear
Vector3  angular
```

If you subscribe to the `/pose` topic, you can receive and examine the messages that are sent.

```
controlNode = ros2node("/base_station");
poseSub = ros2subscriber(controlNode,"/pose","geometry_msgs/Twist")

poseSub =
  ros2subscriber with properties:

        TopicName: '/pose'
    LatestMessage: []
      MessageType: 'geometry_msgs/Twist'
    NewMessageFcn: []
          History: 'keeplast'
            Depth: 10
      Reliability: 'reliable'
       Durability: 'volatile'
```

Use `receive` to acquire data from the subscriber. Once a new message is received, the function returns it and stores it in the `posedata` variable. Specify a timeout of 10 seconds for receiving messages.

```
poseData = receive(poseSub,10)
```

```
poseData = struct with fields:
     linear: [1x1 struct]
    angular: [1x1 struct]
```

The message has a type of `geometry_msgs/Twist`. There are two other fields in the message: `linear` and `angular`. You can see the values of these message fields by accessing them directly.

`poseData.linear`

```
ans = struct with fields:
    x: 0.0315
    y: 0.0406
    z: -0.0373
```

`poseData.angular`

```
ans = struct with fields:
    x: 0.0413
    y: 0.0132
    z: -0.0402
```

You can see that each of the values of these message fields is actually a message in itself. `geometry_msgs/Twist` is a composite message made up of two `geometry_msgs/Vector3` messages.

Data access for these nested messages works exactly the same as accessing the data in other messages. Access the x component of the `linear` message using this command:

`xPose = poseData.linear.x`

`xPose = 0.0315`

Set Message Data

You can also set message property values. Create a message with type `geometry_msgs/Twist`.

`twist = ros2message("geometry_msgs/Twist")`

```
twist = struct with fields:
     linear: [1x1 struct]
    angular: [1x1 struct]
```

The numeric properties of this message are initialized to 0 by default. You can modify any of the properties of this message. Set the `linear.y` field to 5.

`twist.linear.y = 5;`

You can view the message data to make sure that your change took effect.

`twist.linear`

```
ans = struct with fields:
    x: 0
    y: 5
    z: 0
```

Once a message is populated with your data, you can use it with publishers and subscribers.

Copy Messages

ROS 2 messages are structures. They can be copied directly to make a new message. The copy and the original messages each have their own data.

Make a new empty message to convey temperature data, then make a copy for modification.

```matlab
tempMsgBlank = ros2message("sensor_msgs/Temperature");
tempMsgCopy = tempMsgBlank

tempMsgCopy = struct with fields:
        header: [1x1 struct]
    temperature: 0
       variance: 0
```

Modify the `temperature` property of tempMsg and notice that the contents of `tempMsgBlank` remain unchanged.

```matlab
tempMsgCopy.temperature = 100

tempMsgCopy = struct with fields:
        header: [1x1 struct]
    temperature: 100
       variance: 0


tempMsgBlank

tempMsgBlank = struct with fields:
        header: [1x1 struct]
    temperature: 0
       variance: 0
```

It may be useful to keep a blank message structure around, and only set the specific fields when there is data for it before sending the message.

```matlab
thermometerNode = ros2node("/thermometer");
tempPub = ros2publisher(thermometerNode,"/temperature","sensor_msgs/Temperature");
tempMsgs(10) = tempMsgBlank;     % Pre-allocate message structure array
for iMeasure = 1:10
    % Copy blank message fields
    tempMsgs(iMeasure) = tempMsgBlank;

    % Record sample temperature
    tempMsgs(iMeasure).temperature = 20+randn*3;

    % Only calculate the variation once sufficient data observed
    if iMeasure >= 5
        tempMsgs(iMeasure).variance = var([tempMsgs(1:iMeasure).temperature]);
    end

    % Pass the data to subscribers
    send(tempPub,tempMsgs(iMeasure))
end
errorbar([tempMsgs.temperature],[tempMsgs.variance])
```
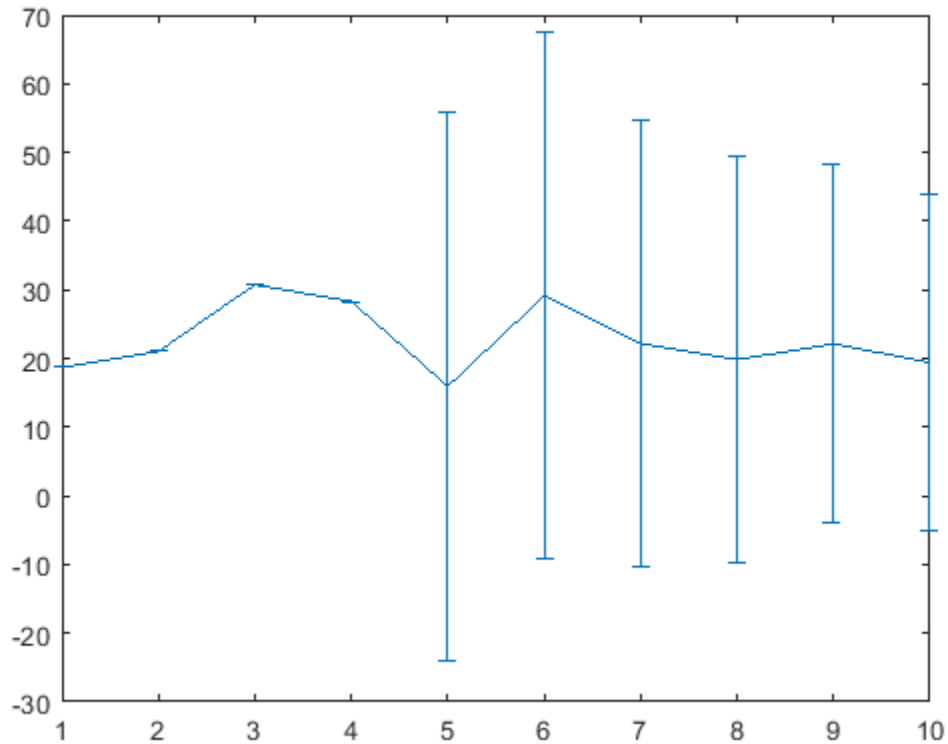
Save and Load Messages

You can save messages and store the contents for later use.

Get a new message from the subscriber.

```
poseData = receive(poseSub,10)
```

```
poseData = struct with fields:
     linear: [1x1 struct]
    angular: [1x1 struct]
```

Save the pose data to a MAT file using the `save` function.

```
save("poseFile.mat","poseData")
```

Before loading the file back into the workspace, clear the `poseData` variable.

```
clear poseData
```

Now you can load the message data by calling the `load` function. This loads the `poseData` from above into the `messageData` structure. `poseData` is a data field of the struct.

```
messageData = load("poseFile.mat")
```

```
messageData = struct with fields:
    poseData: [1x1 struct]
```

Examine `messageData.poseData` to see the message contents.

```
messageData.poseData
```

```
ans = struct with fields:
     linear: [1x1 struct]
    angular: [1x1 struct]
```

You can now delete the MAT file.

```
delete("poseFile.mat")
```

Disconnect From ROS 2 Network

Remove the sample nodes, publishers, and subscribers from the ROS 2 network.

```
exampleHelperROS2ShutDownSampleNetwork
```

Next Steps

- "Exchange Data with ROS 2 Publishers and Subscribers" on page 2-17
- "ROS 2 Custom Message Support" on page 2-29

Exchange Data with ROS 2 Publishers and Subscribers

The primary mechanism for ROS 2 nodes to exchange data is to send and receive *messages*. Messages are transmitted on a *topic* and each topic has a unique name in the ROS 2 network. If a node wants to share information, it must use a *publisher* to send data to a topic. A node that wants to receive that information must use a *subscriber* for that same topic. Besides its unique name, each topic also has a *message type*, which determines the type of messages that are allowed to be transmitted in the specific topic.

This publisher-subscriber communication has the following characteristics:

- Topics are used for many-to-many communication. Multiple publishers can send messages to the same topic and multiple subscribers can receive them.
- Publisher and subscribers are decoupled through topics and can be created and destroyed in any order. A message can be published to a topic even if there are no active subscribers.
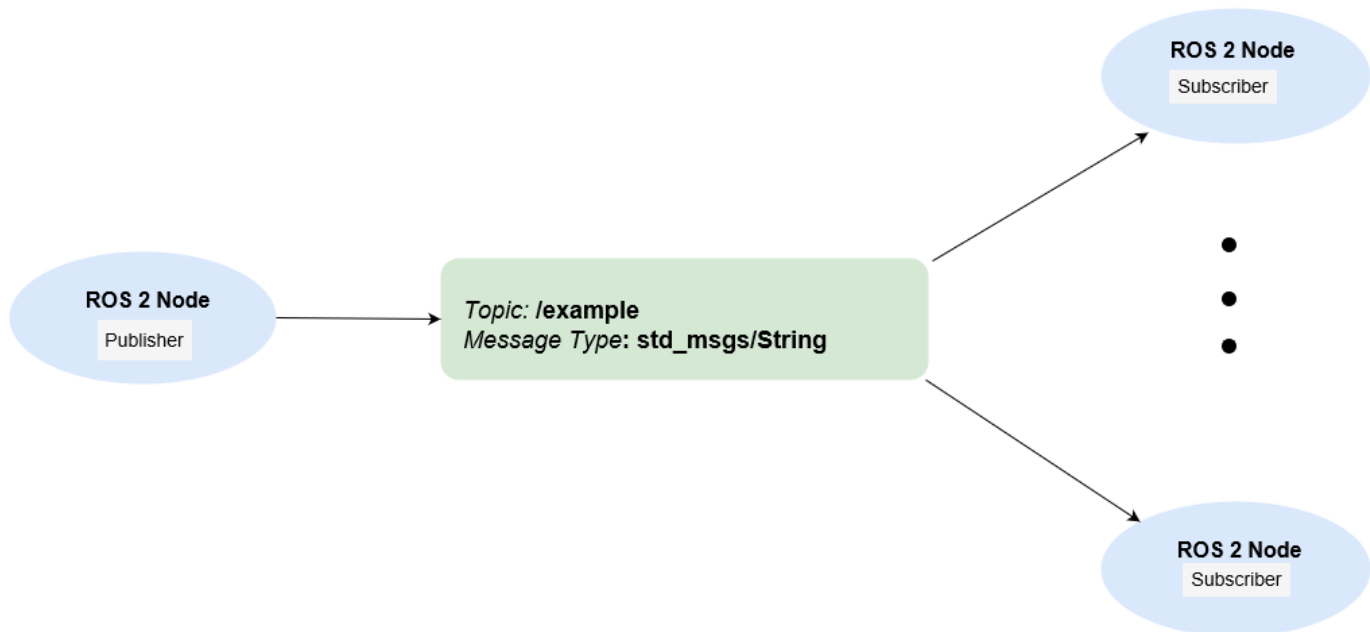
This example shows how to publish and subscribe to topics in a ROS 2 network. It also shows how to:

- Wait until a new message is received, or
- Use callbacks to process new messages in the background

Prerequisites: "Get Started with ROS 2" on page 2-2, "Connect to a ROS 2 Network" on page 2-6

Subscribe and Wait for Messages

Create a sample ROS 2 network with several publishers and subscribers.

exampleHelperROS2CreateSampleNetwork

Use `ros2 topic list` to see which topics are available.

ros2 `topic list`

```
/parameter_events
/pose
/scan
```

Assume you want to subscribe to the `/scan` topic. Use `ros2subscriber` to subscribe to the `/scan` topic. Specify the name of the node with the subscriber. If the topic already exists in the ROS 2 network, `ros2subscriber` detects its message type automatically, so you do not need to specify it.

```
detectNode = ros2node("/detection");
pause(2)
laserSub = ros2subscriber(detectNode,"/scan");
pause(2)
```

Use `receive` to wait for a new message. Specify a timeout of 10 seconds. The output `scanData` contains the received message data.

```
scanData = receive(laserSub,10);
```

You can now remove the subscriber `laserSub` and the node associated to it.

```
clear laserSub
clear detectNode
```

Subscribe Using Callback Functions

Instead of using `receive` to get data, you can specify a function to be called when a new message is received. This allows other MATLAB code to execute while the subscriber is waiting for new messages. Callbacks are essential if you want to use multiple subscribers.

Subscribe to the `/pose` topic, using the callback function `exampleHelperROS2PoseCallback`, which takes a received message as the input. One way of sharing data between your main workspace and the callback function is to use global variables. Define two global variables `pos` and `orient`.

```
controlNode = ros2node("/base_station");
poseSub = ros2subscriber(controlNode,"/pose",@exampleHelperROS2PoseCallback);
global pos
global orient
```

The global variables `pos` and `orient` are assigned in the `exampleHelperROS2PoseCallback` function when new message data is received on the `/pose` topic.

```
function exampleHelperROS2PoseCallback(message)
    % Declare global variables to store position and orientation
    global pos
    global orient

    % Extract position and orientation from the ROS message and assign the
    % data to the global variables.
    pos = [message.linear.x message.linear.y message.linear.z];
    orient = [message.angular.x message.angular.y message.angular.z];
end
```

Wait a moment for the network to publish another `/pose` message. Display the updated values.

```
pause(3)
disp(pos)

    -0.0027   -0.0347   -0.0159
```

```
disp(orient)
```

```
    0.0107   -0.0308    0.0238
```

If you type in `pos` and `orient` a few times in the command line you can see that the values are continuously updated.

Stop the pose subscriber by clearing the subscriber variable

```
clear poseSub
clear controlNode
```

Note: There are other ways to extract information from callback functions besides using globals. For example, you can pass a handle object as additional argument to the callback function. See the "Callback Definition" documentation for more information about defining callback functions.

Publish Messages

Create a publisher that sends ROS 2 string messages to the `/chatter` topic.

```
chatterPub = ros2publisher(node_1,"/chatter","std_msgs/String");
```

Create and populate a ROS 2 message to send to the `/chatter` topic.

```
chatterMsg = ros2message(chatterPub);
chatterMsg.data = 'hello world';
```

Use `ros2 topic list` to verify that the `/chatter` topic is available in the ROS 2 network.

```
ros2 topic list
```

```
/chatter
/parameter_events
/pose
/scan
```

Define a subscriber for the `/chatter` topic. `exampleHelperROS2ChatterCallback` is called when a new message is received, and displays the string content in the message.

```
chatterSub = ros2subscriber(node_2,"/chatter",@exampleHelperROS2ChatterCallback)
```

```
chatterSub =
  ros2subscriber with properties:

        TopicName: '/chatter'
    LatestMessage: []
      MessageType: 'std_msgs/String'
    NewMessageFcn: @exampleHelperROS2ChatterCallback
          History: 'keeplast'
            Depth: 10
      Reliability: 'reliable'
       Durability: 'volatile'
```

Publish a message to the `/chatter` topic. Observe that the string is displayed by the subscriber callback.

```
send(chatterPub,chatterMsg)
pause(3)
```

```
ans =
'hello world'
```

The `exampleHelperROS2ChatterCallback` function was called when the subscriber received the string message.

Disconnect From ROS 2 Network

Remove the sample nodes, publishers and subscribers from the ROS 2 network. Also clear the global variables `pos` and `orient`

```
clear global pos orient
clear
```

Next Steps

- "Work with Basic ROS 2 Messages" on page 2-11
- "ROS 2 Custom Message Support" on page 2-29

Manage Quality of Service Policies in ROS 2

Quality of Service (QoS) policy options allow for changing the behavior of communication within a ROS 2 network. QoS policies are modified for specific communication objects, such as publishers and subscribers, and change the way that messages are handled in the object and transported between them. For any messages to pass between two communication objects, their QoS policies must be compatible.

The available Quality of Service policies in ROS 2 are:

- `History` - Message queue mode
- `Depth` - Message queue size
- `Reliability` - Delivery guarantee of messages
- `Durability` - Persistence of messages

For more information, see About Quality of Service Settings.

History and Depth

The *history* and *depth* QoS policies determine the behavior of communication objects when messages are being made available faster than they can be processed. This is primarily a concern for subscribers that are receiving messages while still processing the previous message. Messages are placed into a processing queue, which can affect publishers as well. History has the options of:

- `"keeplast"` - The message processing queue has a maximum size equal to the `Depth` value. If the queue is full, the oldest messages are dropped to make room for newer ones.
- `"keepall"` - The message processing queue attempts to keep all messages received in the queue until processed.

Under either history setting, the queue size is subject to hardware resource limits. If the subscriber calls a callback when new messages are received, the queue size is also limited by the maximum recursion limit.

In situations where it is important to process all messages, increasing the `Depth` value or using `History,"keepall"` is recommended.

This example shows how to set up a publisher and subscriber for sending and receiving point cloud messages. The publisher `Depth` is 20 and the subscriber history is set to `"keepall"`. The subscriber uses a call back to plot the time stamp for each message to show the timing of processing each message. The initial messages take longer to process, but all the messages are eventually processed from the queue.

```matlab
% Create a publisher to provide sensor data
robotNode = ros2node("/simple_robot");
lidarPub = ros2publisher(robotNode,"/laser_scan","sensor_msgs/PointCloud2",...
    "History","keeplast","Depth",20);

% Create a subscriber representing localization, requiring all scan data
hFig = figure;
hAxesLidar = axes("Parent",hFig);
title("Message Timeline (Keep All)")
localizationSub = ros2subscriber(robotNode,"/laser_scan",...
    @(msg)exampleHelperROS2PlotTimestamps(msg,hAxesLidar),...
    "History","keepall");
```

```
% Send messages, simulating an extremely fast sensor
load robotPoseLidarData.mat lidarScans
for iMsg = 1:numel(lidarScans)
    send(lidarPub,lidarScans(iMsg))
end

% Allow messages to arrive, then remove the localization subscriber
pause(3)
```

Message Timeline (Keep All)
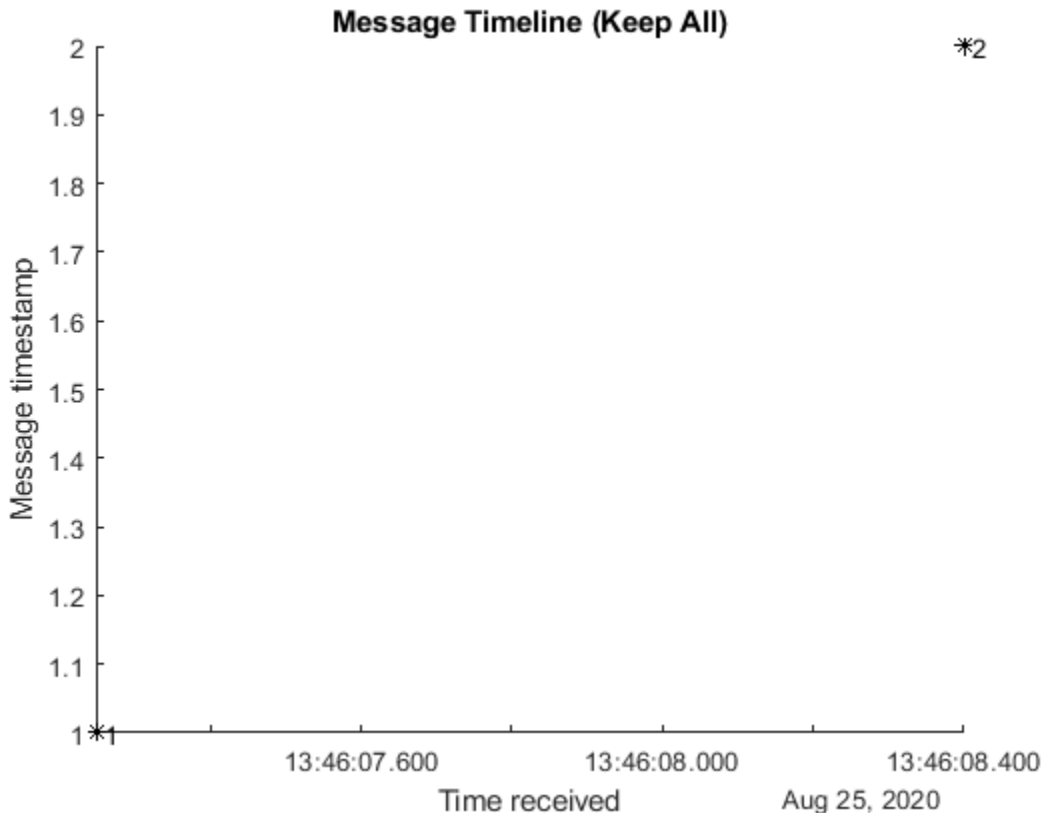

```
clear localizationSub
```

In situations where messages being dropped is less important, and only the most up-to-date information really matters, a smaller queue is recommended to improve performance and ensure the most recent information is being used. This example shows quicker processing of the first messages and still gets all the messages. Depending on your resources however, you may see messages get dropped.

```
% Create a subscriber representing user interface display
hFig = figure;
hAxesLidar2 = axes("Parent",hFig);
title("Message Timeline (Keep Last 1)")
scanDisplaySub = ros2subscriber(robotNode,"/laser_scan",...
    @(msg)exampleHelperROS2PlotTimestamps(msg,hAxesLidar2),...
    "History","keeplast","Depth",1);
for iMsg = 1:numel(lidarScans)
    send(lidarPub,lidarScans(iMsg))
```

```
end
```

```
% Allow messages to arrive, then remove the subscriber and publisher
pause(3)
```

Message Timeline (Keep Last 1)
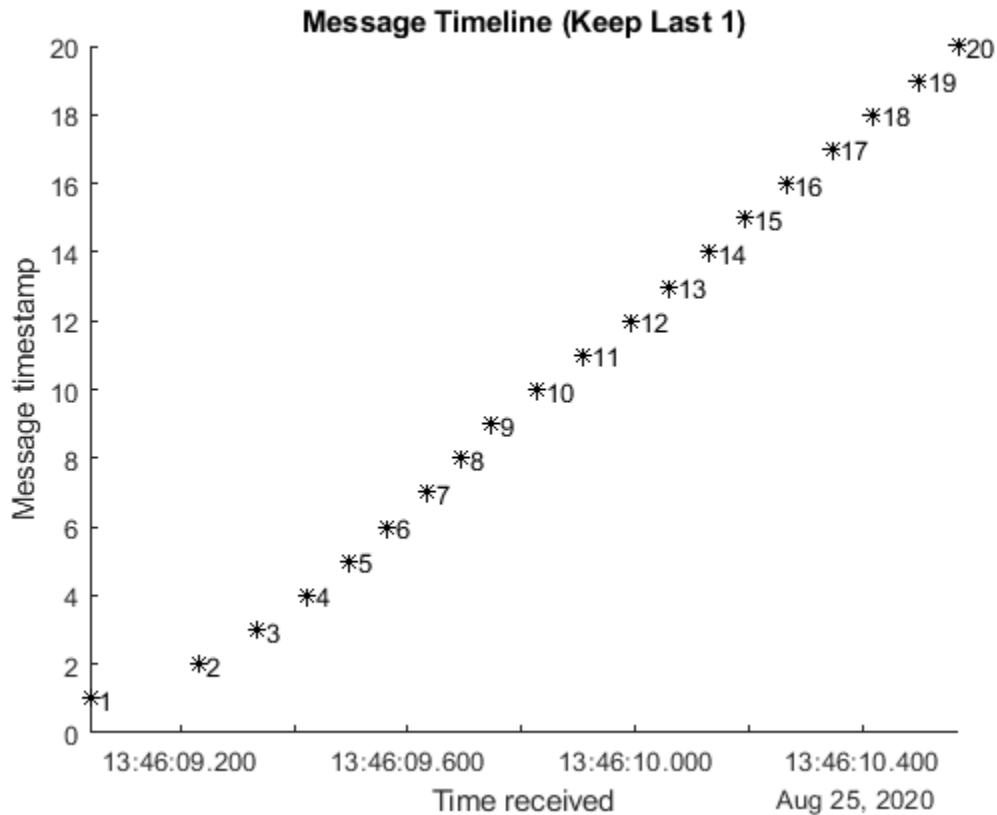
```
clear lidarPub scanDisplaySub
```

Reliability

The *reliability* QoS policy determines whether to guarantee delivery of messages, and has the options:

- `"reliable"` - The publisher continuously sends the message to the subscriber until the subscriber confirms receipt of the message.
- `"besteffort"` - The publisher sends the message only once, and does not confirm that the subscriber receives it.

Reliable

A `"reliable"` connection is useful when all of the data must be processed, and any dropped messages may impact the result. This example publishes `Odometry` messages and uses a subscriber callback to plot the position. Because for the `"reliable"` setting, all the positions are plotted in the figure.

```
% Create a publisher for odometry data
odomPub = ros2publisher(robotNode,"/odom","nav_msgs/Odometry",...
    "Reliability","reliable");
```

```
% Create a subscriber for localization
hFig = figure;
hAxesReliable = axes("Parent",hFig);
title("Robot Position (Reliable Connection)")
xlabel("X (m)")
ylabel("Y (m)")
odomPlotSub = ros2subscriber(robotNode,"/odom",...
    @(msg)exampleHelperROS2PlotOdom(msg,hAxesReliable,"ok"),...
    "Reliability","reliable");

% Send messages, simulating an extremely fast sensor
load robotPoseLidarData.mat odomData
for iMsg = 1:numel(odomData)
    send(odomPub,odomData(iMsg))
end

pause(3)    % Allow messages to arrive and be plotted
```
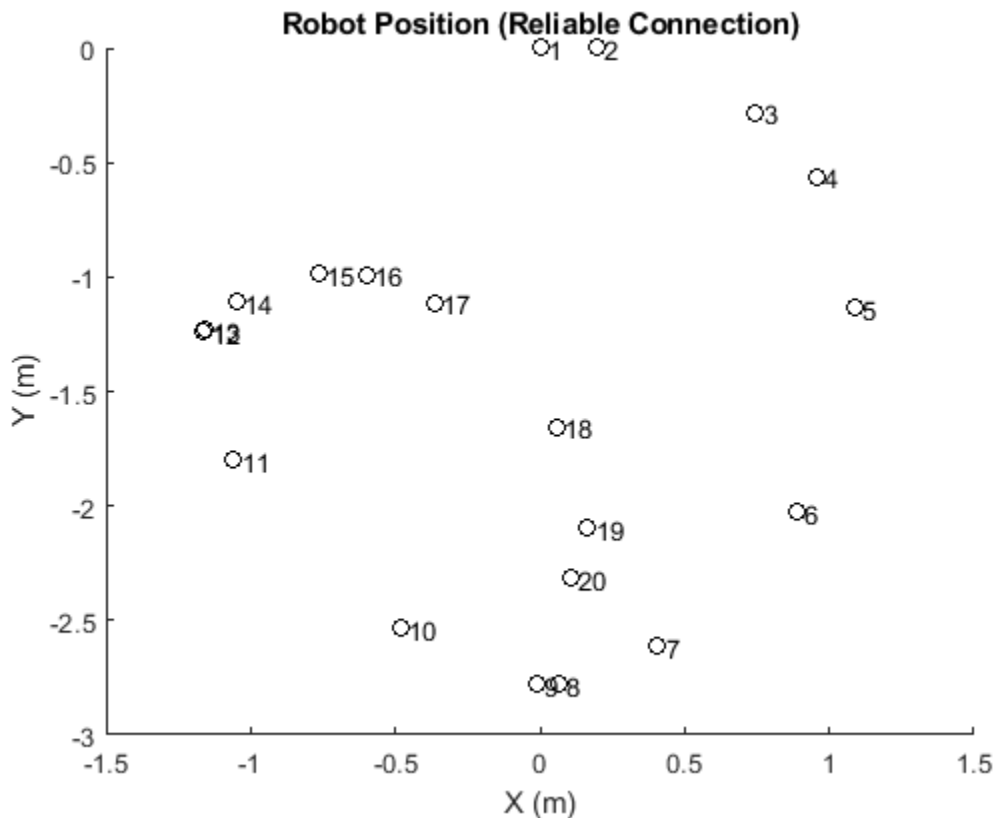


```
% Temporarily prevent reliable subscriber from reacting to new messages
odomPlotSub.NewMessageFcn = [];
```

Best Effort

A "besteffort" connection is useful to avoid impacting performance if dropped messages are acceptable. If a publisher is set to "reliable", and a subscriber is set to "besteffort", the publisher treats that connection as only requiring "besteffort", and does not confirm delivery.

Connections with `"reliable"` subscribers on the same topic are guaranteed delivery from the same publisher.

This example uses a `"besteffort"` subscriber, but still receives all messages due to the low impact on the network.

```
hFig = figure;
hAxesBestEffort = axes("Parent",hFig);
title("Message Timeline (Best Effort Connection)")
odomTimingSub = ros2subscriber(robotNode,"/odom",...
    @(msg)exampleHelperROS2PlotTimestamps(msg,hAxesBestEffort),...
    "Reliability","besteffort");
for iMsg = 1:numel(odomData)
    send(odomPub,odomData(iMsg))
end

pause(3)    % Allow messages to arrive and be plotted
```
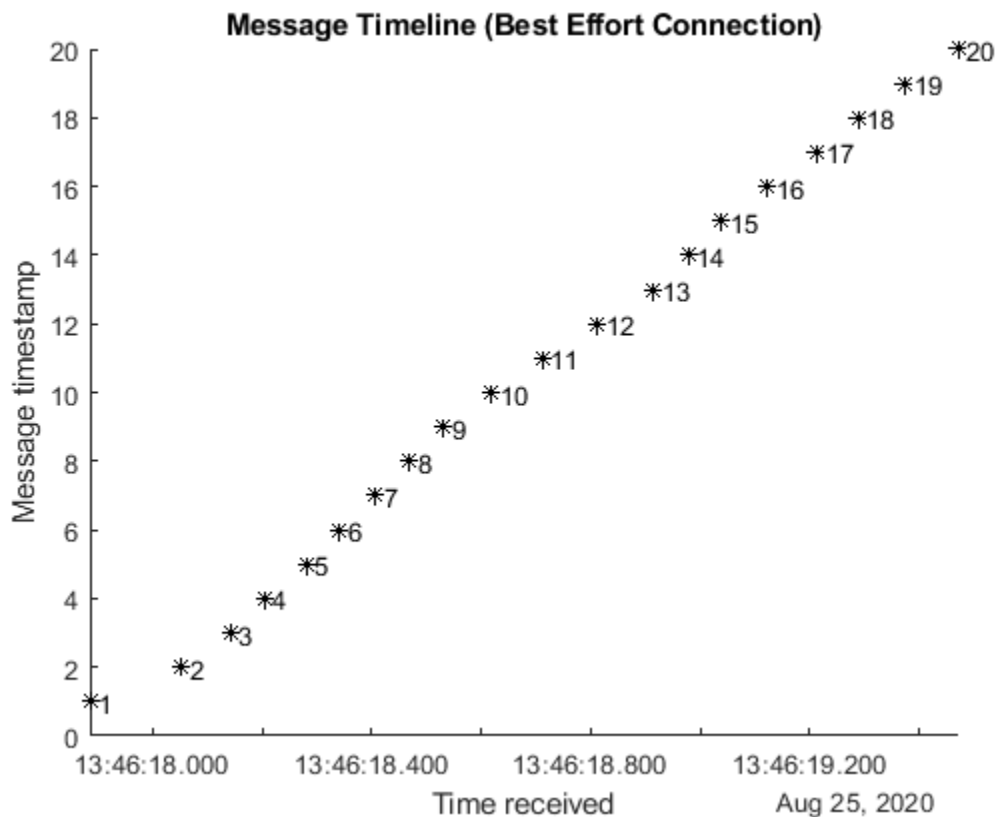

Compatibility

Ensuring compatibility is an important consideration when setting reliability. A subscriber with a `"reliable"` option set requires a publisher that meets that standard. Any `"besteffort"` publishers do not connect to a `"reliable"` subscriber because messages are not guaranteed to be delivered. In the opposite situation, a `"reliable"` publisher and a `"besteffort"` subscriber do connect, but the connection behaves as `"besteffort"` with no confirmation when receiving messages. This example shows a `"besteffort"` publisher sending messages to the `"besteffort"`

subscriber already set up. Again, due to the low impact on the network, the `"besteffort"` connection is sufficient to process all the messages.

```
% Reactivate reliable subscriber to show no messages received
odomPlotSub.NewMessageFcn = @(msg)exampleHelperROS2PlotOdom(msg,hAxesReliable,"*r");

% Send messages from a best-effort publisher
bestEffortOdomPub = ros2publisher(robotNode,"/odom","nav_msgs/Odometry",...
    "Reliability","besteffort");
for iMsg = 1:numel(odomData)
    send(bestEffortOdomPub,odomData(iMsg))
end

% Allow messages to arrive, then remove odometry publishers and subscribers
pause(3)    % Allow messages to arrive and be plotted
```
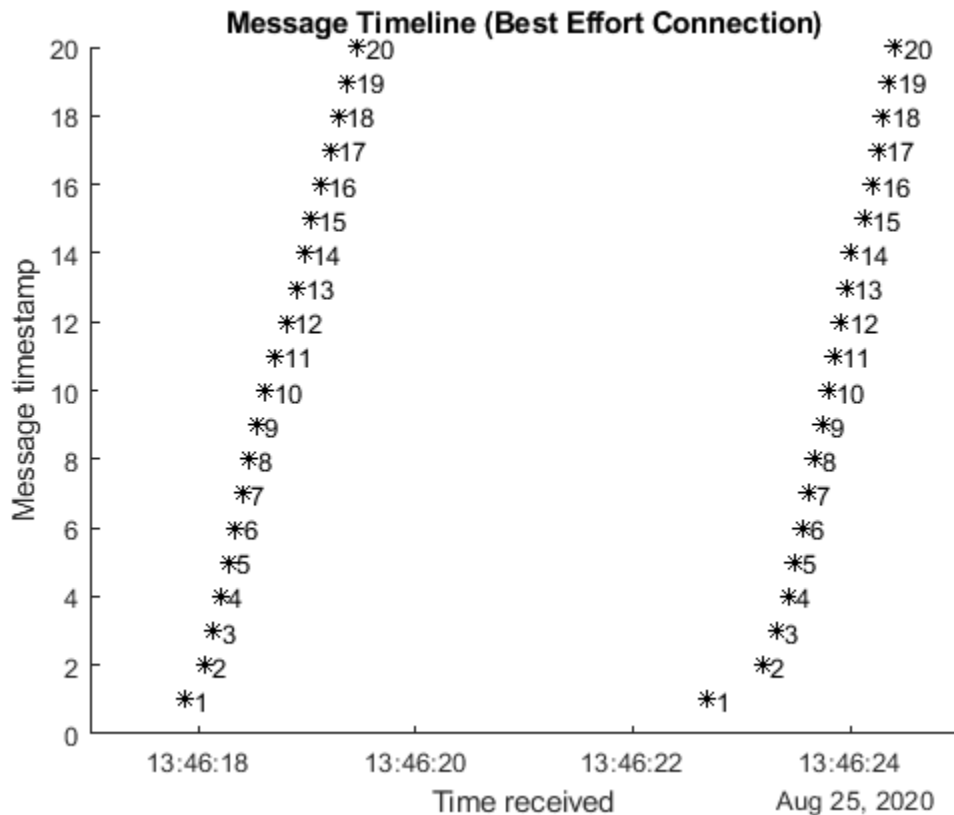


```
clear odomPub bestEffortOdomPub odomPlotSub odomTimingSub
```

Durability and Depth

The *durability* QoS policy controls the persistence of messages for late-joining connections, and has the options:

- `"transientlocal"` - For a publisher, messages that have already been sent are maintained. If a subscriber joins the network with `"transientlocal"` durability after that, then the publisher sends the persisted messages to the subscriber.

- `"volatile"` - Publishers do not persist messages after sending them, and subscribers do not request persisted messages from publishers.

The number of messages persisted by publishers with `"transientlocal"` durability is also controlled by the `Depth` input. Subscribers only request the number of recent messages based on their individual `Depth` settings. Publishers can still store more messages for other subscribers to get more. For example, a full list of the robot position may be useful for visualizing its path, but a localization algorithm may only be interested in the last known location. This example illustrates that by using a localization subscriber to display the current position and a plotting subscriber to show all positions in the queue.

```matlab
% Publish robot location information
posePub = ros2publisher(robotNode,"/bot_position","geometry_msgs/Pose2D",...
    "Durability","transientlocal","Depth",100);
load robotPoseLidarData.mat robotPositions
for iMsg = 1:numel(robotPositions)
    send(posePub,robotPositions(iMsg))
    pause(0.2)      % Allow for processing time
end

% Create a localization update subscriber that only needs current position
localUpdateSub = ros2subscriber(robotNode,"/bot_position",@disp,...
    "Durability","transientlocal","Depth",1);
pause(1)      % Allow message to arrive

        x: 0.1047
        y: -2.3168
    theta: -8.5194

% Create a visualization subscriber to show where the robot has been
hFig = figure;
hAxesMoreMsgs = axes("Parent",hFig);
title("Robot Position (Transient Local Connection)")
xlabel("X (m)")
ylabel("Y (m)")
hold on
posePlotSub = ros2subscriber(robotNode,"/bot_position",...
    @(msg)plot(hAxesMoreMsgs,msg.x,msg.y,"ok"),...
    "Durability","transientlocal","Depth",20);
pause(3)      % Allow messages to arrive and be plotted
```

Robot Position (Transient Local Connection)
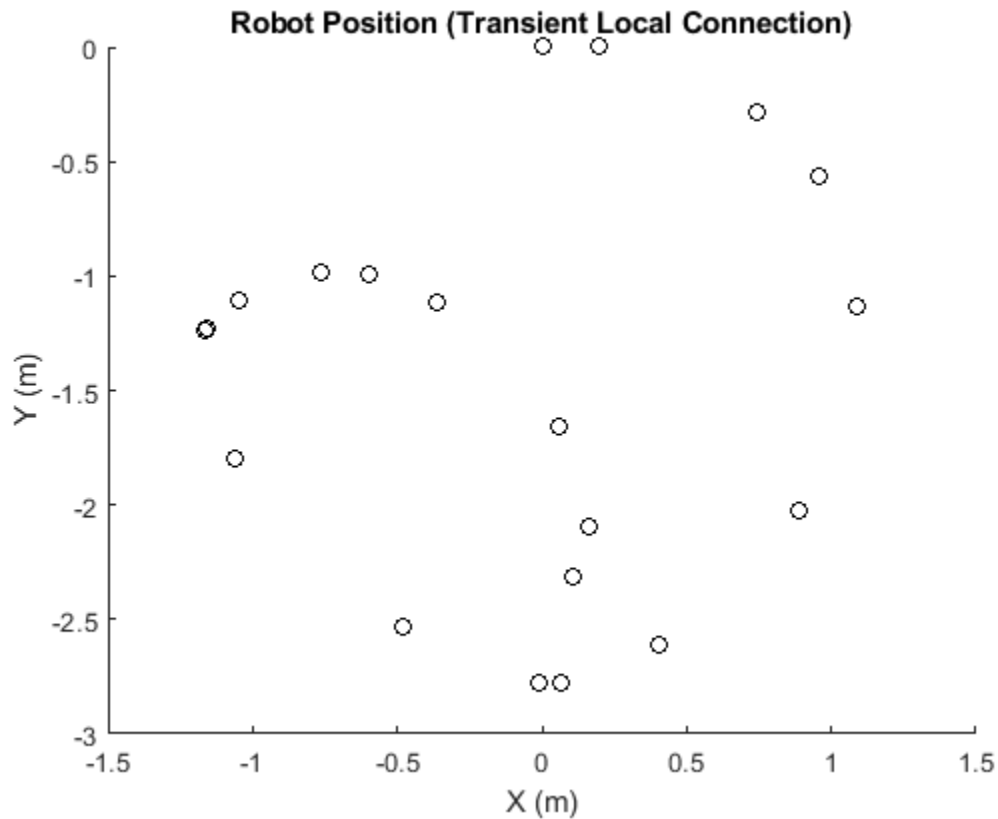
Compatibility

Similar to reliability, incompatible durability settings can prevent communication between publishers and subscribers. A subscriber with `"transientlocal"` durability requires a publisher with `"transientlocal"` durability. If a publisher is `"volatile"`, no connection is established with `"transientlocal"` subscribers. If a publisher is `"transientlocal"` and the subscriber `"volatile"`, then that connection is created, without sending persisting messages to the subscriber.

```
% Reset plotting behavior
posePlotSub.NewMessageFcn = @(msg)plot(hAxesMoreMsgs,msg.x,msg.y,"xr");

% Send messages from volatile publisher
volatilePosePub = ros2publisher(robotNode,"/bot_position",...
    "Durability","volatile");
for iMsg = 1:numel(robotPositions)
    send(volatilePosePub,robotPositions(iMsg))
    pause(0.2)      % Allow for processing time
end
```

No messages are received by either `"transientlocal"` subscriber.

```
% Remove pose publishers and subscribers
clear posePub volatilePosePub localUpdateSub posePlotSub robotNode
```

ROS 2 Custom Message Support

Custom messages are user-defined messages that you can use to extend the set of message types currently supported in ROS 2. 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 `ros2 msg list` in the MATLAB® Command Window. For more information about supported ROS 2 messages, see "Work with Basic ROS 2 Messages" on page 2-11.

If this if your first time working with ROS 2 custom messages, check the "ROS System Requirements".

Custom messages Contents

ROS 2 custom messages are specified in ROS 2 package folders that contain a `msg` directory. The `msg` folder contains all your custom message type definitions. For example, the package `example_b_msgs`, within the `custom` folder, has the below folder and file structure.

```
⊟ 📁 custom
   ⊞ 📁 example_a_msgs
   ⊟ 📁 example_b_msgs
      ⊟ 📁 msg
         ✉ Standalone.msg
   ⊞ 📁 example_c_msgs
```
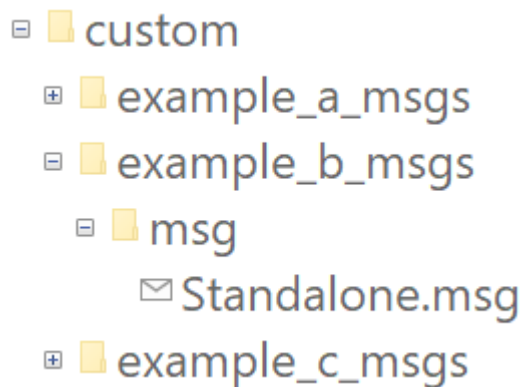
The package contains one custom message type, `Standalone.msg`. MATLAB uses these files to generate the necessary files for using the custom messages contained in the package. For more information on message naming conventions, see ROS 2 Interface Definition.

In this example, you go through the procedure for creating ROS 2 custom messages in MATLAB®. You must have a ROS 2 package that contains the required `msg` file.

After ensuring that your custom message package is correct, note the folder path location, and then, call `ros2genmsg` with the specified path. The following example provided three messages `example_package_a`, `example_package_b`, and `example_package_c` that have dependencies. 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, the custom message interface folder is present in the current directory. If you are creating custom message packages in a separate location, provide the appropriate path to the folder that contains the custom message packages.

```
folderPath = fullfile(pwd,"custom");
copyfile("example_*_msgs",folderPath);
```

Specify the folder path for custom message files and call `ros2genmsg` to create custom messages for MATLAB.

```
ros2genmsg(folderPath)
```

```
Identifying message files in folder 'U:/Documents/MATLAB/Examples/ros-ex44405863/custom'.Done.
Validating message files in folder 'U:/Documents/MATLAB/Examples/ros-ex44405863/custom'.Done.
[3/3] Generating MATLAB interfaces for custom message packages... Done.
Running colcon build in folder 'U:/Documents/MATLAB/Examples/ros-ex44405863/custom/matlab_msg_ger
Build in progress. This may take several minutes...
Build succeeded.build log
```

Call `ros2 msg list` to verify creation of new custom messages.

You can now use the above created custom message as the standard messages. For more information on sending and receiving messages, see "Exchange Data with ROS 2 Publishers and Subscribers" on page 2-17.

Create a publisher to use `example_package_b/Standalone` message.

```
node = ros2node("/node_1");
pub = ros2publisher(node,"/example_topic","example_b_msgs/Standalone");
```

Create a subscriber on the same topic.

```
sub = ros2subscriber(node,"/example_topic");
```

Create a message and send the message.

```
custom_msg = ros2message("example_b_msgs/Standalone");
custom_msg.int_property = uint32(12);
custom_msg.string_property='This is ROS 2 custom message example';
send(pub,custom_msg);
pause(3)     % Allow a few seconds for the message to arrive
```

Use `LatestMessage` field to know the recent message received by the subscriber.

```
sub.LatestMessage
```

```
ans = struct with fields:
        int_property: 12
     string_property: 'This is ROS 2 custom message example'
```

Remove the created ROS objects.

```
clear node pub sub
```

Using ROS Bridge to Establish Communication between ROS and ROS 2

ROS 2 is newer version of ROS with different architecture. Both the networks are separate and there is no direct communication between the nodes in ROS and ROS 2. The `ros1_bridge` package provides a network bridge which enables the exchange of messages between ROS and ROS 2. The bridge manages all the conversion required and sends messages across both the networks. For more information, see `ros1_bridge`. This example uses a virtual machine which may be downloaded by following the instructions in "Get Started with Gazebo and a Simulated TurtleBot" on page 1-102. The `ros1_bridge` package is installed on this virtual machine.

This example shows how to control the TurtleBot3 in Gazebo using keyboard commands from the MATLAB®. The Gazebo Simulator is available in ROS 1 networks only. You can use `ros1_bridge` to exchange the Gazebo topics such as `'/odom'` or `'/cmd_vel'` to ROS 2.

The below diagram depicts the message exchange between ROS 1 and ROS 2 networks using `ros1_bridge`. The `'/odom'` topic contains `nav_msgs/Odometry` messages sent from the ROS 1 network with Gazebo. The ROS 2 node subscribes to the `/odom` topic that has been bridged from ROS 1 and publishes a `'/cmd_vel'` message based on the robot pose. The bridge then takes the `'/cmd_vel'` message and publishes it on the ROS 1 network.
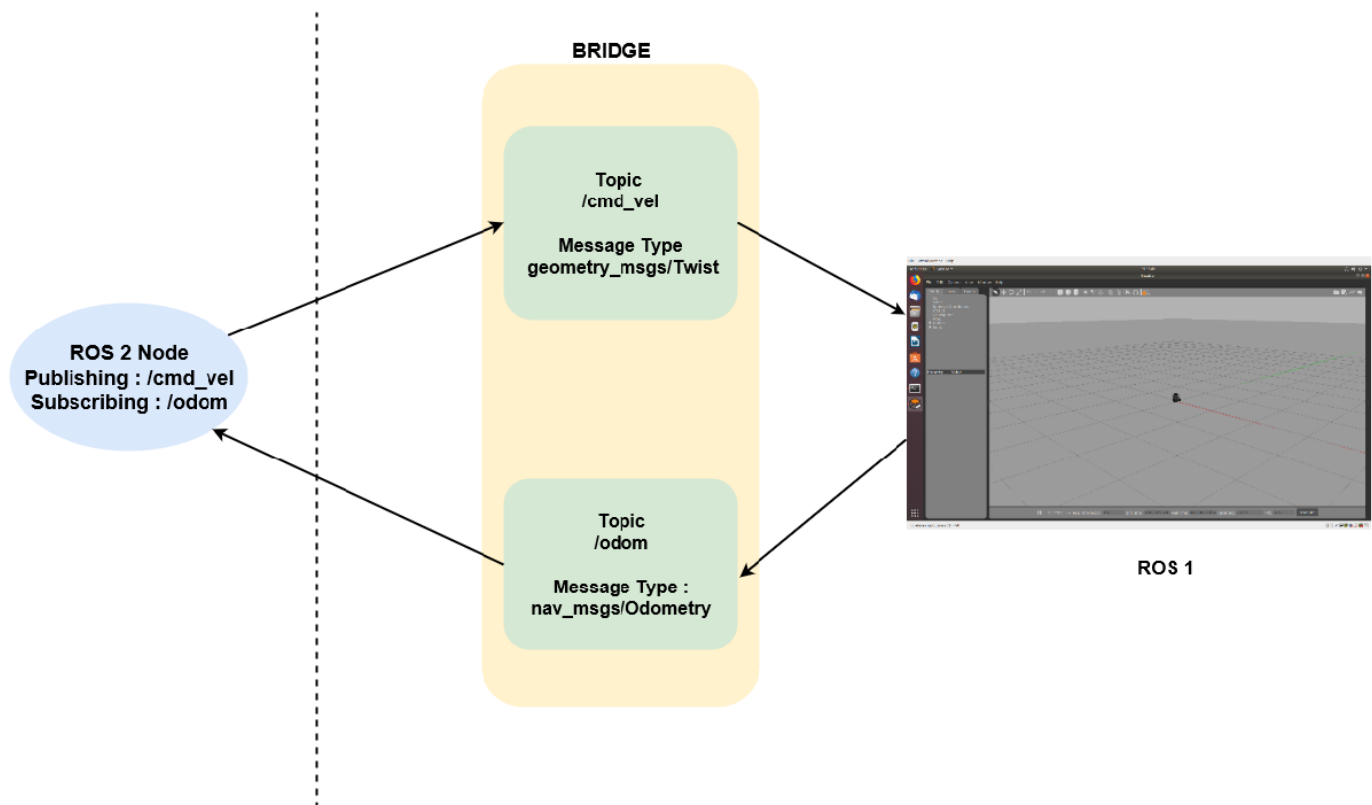
Connecting ROS 1 and ROS 2 Using ROS Bridge

Prerequisites:

- Download the Virtual Machine using instructions in "Get Started with Gazebo and a Simulated TurtleBot" on page 1-102
- "Connect to a ROS 2 Network" on page 2-6

Set Up Virtual Machine

Communicate Outside Subnet

You may need to create an XML file on the VM named *DEFAULT_FASTRTPS_PROFILE.xml* to configure IP addresses to communicate under different subnets (see **Communicate Outside Subnet** section in "Connect to a ROS 2 Network" on page 2-6). In the example XML file replace <address> entries with host and VM IP addresses and replace <domainId> entry with your specified domain. Create the same file, with the same contents, on your host computer in the MATLAB current working directory.

Example file:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<profiles>
   <participant profile_name="participant_win" is_default_profile="true">
      <rtps>
         <builtin>
            <metatrafficUnicastLocatorList>
               <locator/>
            </metatrafficUnicastLocatorList>
            <domainId>25</domainId>
            <initialPeersList>
               <locator>
                  <udpv4>
                     <address>192.168.2.147</address>
                  </udpv4>
               </locator>
               <locator>
                  <udpv4>
                     <address>192.168.2.1</address>
                  </udpv4>
               </locator>
            </initialPeersList>
         </builtin>
      </rtps>
   </participant>
</profiles>
```

Launch Gazebo

On the VM desktop, click **Gazebo Empty**. This Gazebo world contains a Turtlebot robot, which publishes and subscribes to messages on a ROS 1 network.

Start the Bridge

Click the **ROS Bridge** shortcut. This bridge setups publishers and subscribers for all the ROS 1 topics on a ROS 2 network.
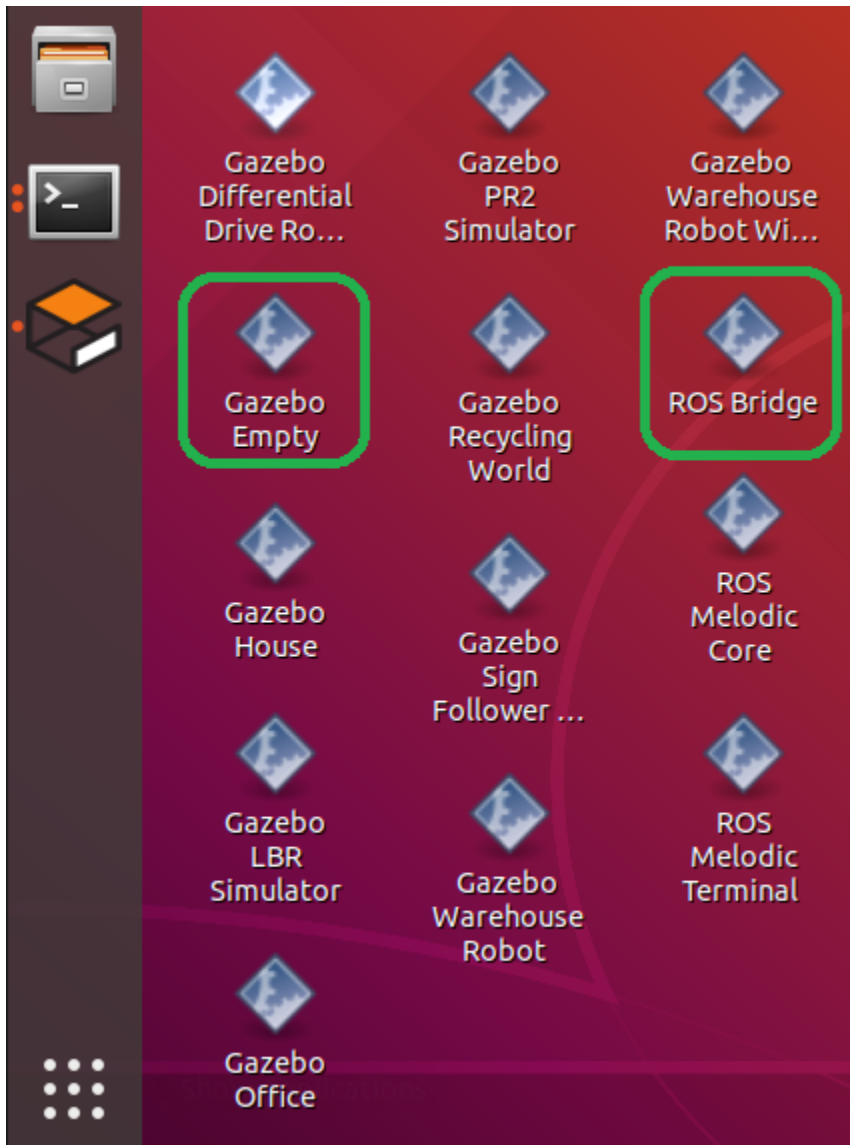
In the Terminal window, notice that the bridge is up and running.

```
user@ubuntu:~$ ros2 run ros1_bridge dynamic_bridge --bridge-all-topics
created 1to2 bridge for topic '/clock' with ROS 1 type 'rosgraph_msgs/Clock' and ROS 2 type 'rosgraph_msgs/Clock'
created 1to2 bridge for topic '/imu' with ROS 1 type 'sensor_msgs/Imu' and ROS 2 type 'sensor_msgs/Imu'
created 1to2 bridge for topic '/joint_states' with ROS 1 type 'sensor_msgs/JointState' and ROS 2 type 'sensor_msgs/JointState'
created 1to2 bridge for topic '/odom' with ROS 1 type 'nav_msgs/Odometry' and ROS 2 type 'nav_msgs/Odometry'
created 1to2 bridge for topic '/scan' with ROS 1 type 'sensor_msgs/LaserScan' and ROS 2 type 'sensor_msgs/LaserScan'
created 1to2 bridge for topic '/tf' with ROS 1 type 'tf2_msgs/TFMessage' and ROS 2 type 'tf2_msgs/TFMessage'
Created 2 to 1 bridge for service /gazebo/pause_physics
Created 2 to 1 bridge for service /gazebo/reset_simulation
Created 2 to 1 bridge for service /gazebo/reset_world
Created 2 to 1 bridge for service /gazebo/unpause_physics
Created 2 to 1 bridge for service /imu_service
[INFO] [ros1_bridge]: Passing message from ROS 1 rosgraph_msgs/Clock to ROS 2 rosgraph_msgs/Clock (showing msg only once per type)
[INFO] [ros1_bridge]: Passing message from ROS 1 sensor_msgs/Imu to ROS 2 sensor_msgs/Imu (showing msg only once per type)
[INFO] [ros1_bridge]: Passing message from ROS 1 sensor_msgs/JointState to ROS 2 sensor_msgs/JointState (showing msg only once per type)
[INFO] [ros1_bridge]: Passing message from ROS 1 tf2_msgs/TFMessage to ROS 2 tf2_msgs/TFMessage (showing msg only once per type)
[INFO] [ros1_bridge]: Passing message from ROS 1 nav_msgs/Odometry to ROS 2 nav_msgs/Odometry (showing msg only once per type)
[INFO] [ros1_bridge]: Passing message from ROS 1 sensor_msgs/LaserScan to ROS 2 sensor_msgs/LaserScan (showing msg only once per type)
created 2to1 bridge for topic '/cmd_vel' with ROS 2 type 'geometry_msgs/Twist' and ROS 1 type ''
[INFO] [ros1_bridge]: Passing message from ROS 2 geometry_msgs/Twist to ROS 1 geometry_msgs/Twist (showing msg only once per type)
```

Open one more terminal and enter the following commands

```
export ROS_DOMAIN_ID=25
source /opt/ros/dashing/setup.bash
```

Now check that Gazebo topics are present in ROS 2.

```
ros2 topic list
```


Echo the **/odom** topic to see messages being published.

```
ros2 topic echo /odom
```


Control the TurtleBot3 from ROS 2

In MATLAB on your host machine, set the proper domain ID for the ROS 2 network using the 'ROS_DOMAIN_ID' environment variable. The ID must be a character vector.

```
setenv("ROS_DOMAIN_ID","25");
```

Create a ROS 2 node. Subscribe to the odometry topic that is bridged from ROS 1.

```
ros2Node = ros2node("/example_node");
handles.odomSub = ros2subscriber(ros2Node,"/odom","nav_msgs/Odometry")
```

handles = struct with fields:
    odomSub: [1×1 ros2subscriber]

Receive the odometry messages from the bridge and use the `exampleHelperGet2DPose` function to unpack the message into a 2D pose. Get the start position of the robot.

```
odomMsg = receive(handles.odomSub);
poseStart = exampleHelperGet2DPose(odomMsg)
```

poseStart = 1×3

    0.2038   0.0140   -0.8517

```
handles.poses = poseStart;
```

Create a publisher for controlling the robot velocity. The bridge takes these messages and sends them on the ROS 1 network.

```
handles.velPub = ros2publisher(ros2Node,'/cmd_vel','geometry_msgs/Twist')
```

handles = struct with fields:
    odomSub: [1×1 ros2subscriber]
      poses: [0.2038 0.0140 -0.8517]
     velPub: [1×1 ros2publisher]

Run the `exampleHelperROS2TurtleBotKeyboardControl` function, which allows you to control the TurtleBot3 with the keyboard. The `handles` input contains the ROS 2 subscriber, ROS 2 publisher, and poses as a structure. The function sends control commands on the ROS 2 network based on the keyboard inputs. The bridge transfers those messages to the ROS 1 network for the Gazebo simulator.

```
poses = exampleHelperROS2TurtleBotKeyboardControl(handles);
```

The figure that opens listens to keyboard inputs for controlling the robot in Gazebo. Hit the keys and watch the robot move. Press **Q** to exit.

i = Forward

j = Left   l = Right

k = Backward

q = Quit
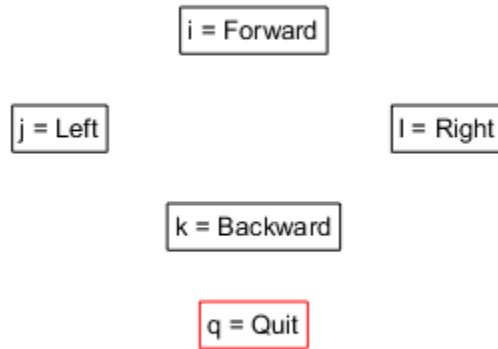
Keep this figure in scope to give commands
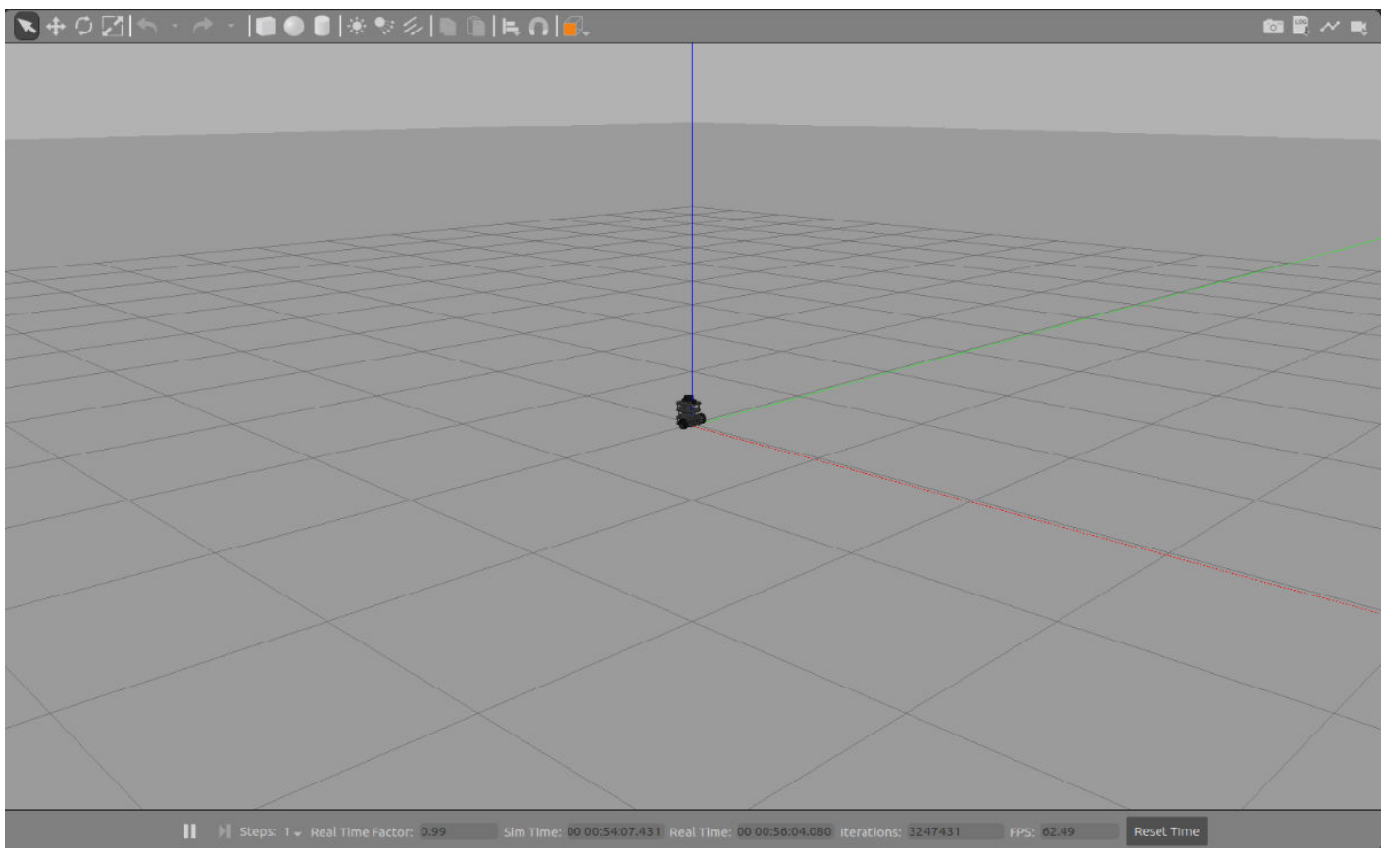
Plot Data Received from ROS

Plot the results to show how TurtleBot3 moved in Gazebo. The `poses` variable has stored all the updated `/odom` messages that were received from the ROS 1 network.

```
odomMsg = receive(handles.odomSub);
poseEnd = exampleHelperGet2DPose(odomMsg)
```

poseEnd = *1×3*

```
    0.8522    0.1618   -1.6255
```

```
poses = [poses;poseEnd];
figure
plot(poses(:,1),poses(:,2),'b-', ...
     poseStart(1),poseStart(2),'go', ...
     poseEnd(1),poseEnd(2),'ro');
xlabel('X [m]');
ylabel('Y [m]');
legend('Trajectory','Start','End');
```
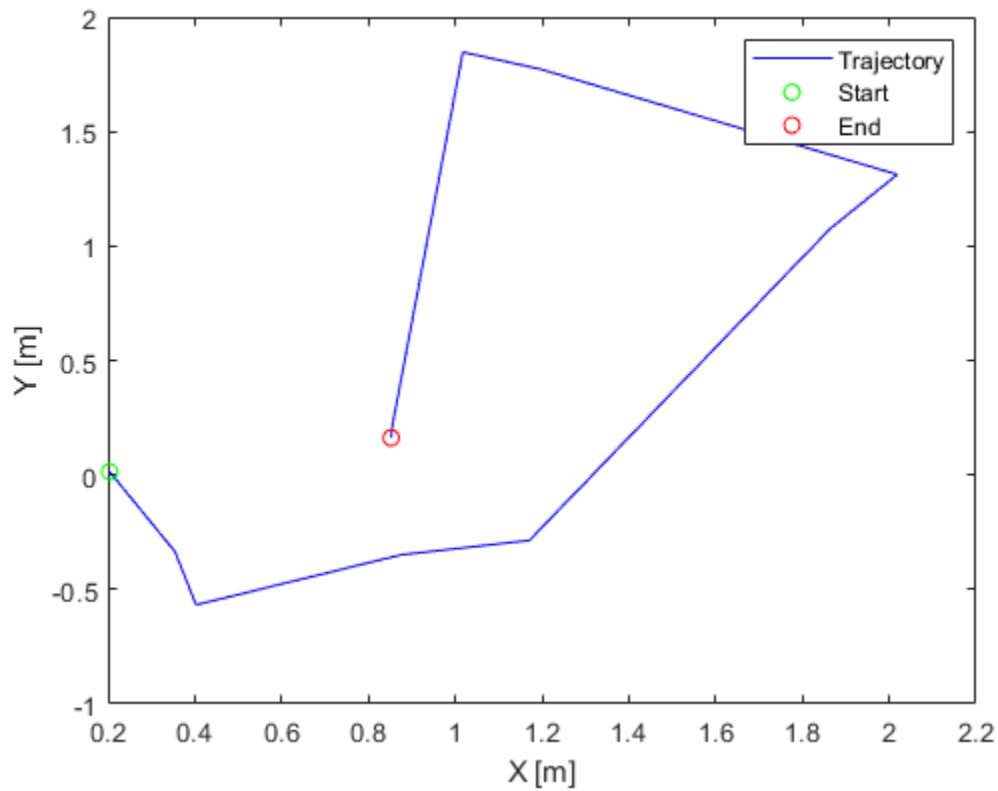

Clear the publishers and subscribers on the host.

```
clear
```

Get Started with ROS 2 in Simulink®

This example shows how to use Simulink blocks for ROS 2 to send and receive messages from a local ROS 2 network.

Introduction

Simulink support for Robot Operating System 2 (ROS 2) enables you to create Simulink models that work with a ROS 2 network. ROS 2 is a communication layer that allows different components of a robot system to exchange information in the form of *messages*. A component sends a message by *publishing* it to a particular *topic*, such as /odometry. Other components receive the message by *subscribing* to that topic.

Simulink support for ROS 2 includes a library of Simulink blocks for sending and receiving messages for a designated topic. When you simulate the model, Simulink connects to a ROS 2 network, which can be running on the same machine as Simulink or on a remote system. Once this connection is established, Simulink exchanges messages with the ROS 2 network until the simulation is terminated. If Simulink Coder™ is installed, you can also generate C++ code for a standalone ROS 2 node, from the Simulink model.

This example shows how to:

- Create and run a Simulink model to send and receive ROS 2 messages
- Work with data in ROS 2 messages

Prerequisites: Create a Simple Model, "Get Started with ROS 2" on page 2-2

Model

You will use Simulink to publish the X and Y location of a robot. You will also subscribe to the same location topic and display the received X,Y location.

Enter the following command to open the completed model created in example.

```
open_system('ros2GetStartedExample');
```

Create a Publisher

Configure a block to send a geometry_msgs/Point message to a topic named /location (the "/" is standard ROS syntax).

- From the MATLAB Toolstrip, select **Home > New > Simulink Model** to open a new Simulink model.
- From the Simulink Toolstrip, select **Simulation > Library Browser** to open the Simulink Library Browser. Click on the **ROS Toolbox** tab (you can also type roslib in MATLAB command window). Select the **ROS 2** Library.
- Drag a **Publish** block to the model. Double-click on the block to configure the topic and message type.
- Select **Specify your own** for the **Topic source**, and enter /location in **Topic**.
- Click **Select** next to **Message type**. A pop-up window will appear. Select geometry_msgs/Point and click **OK** to close the pop-up window.
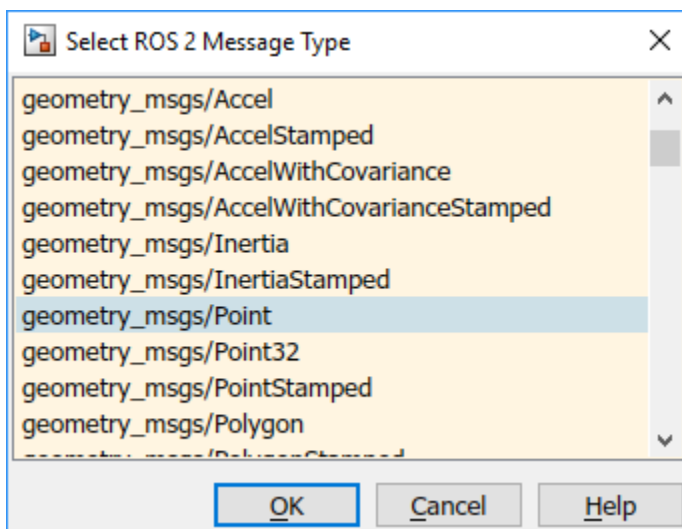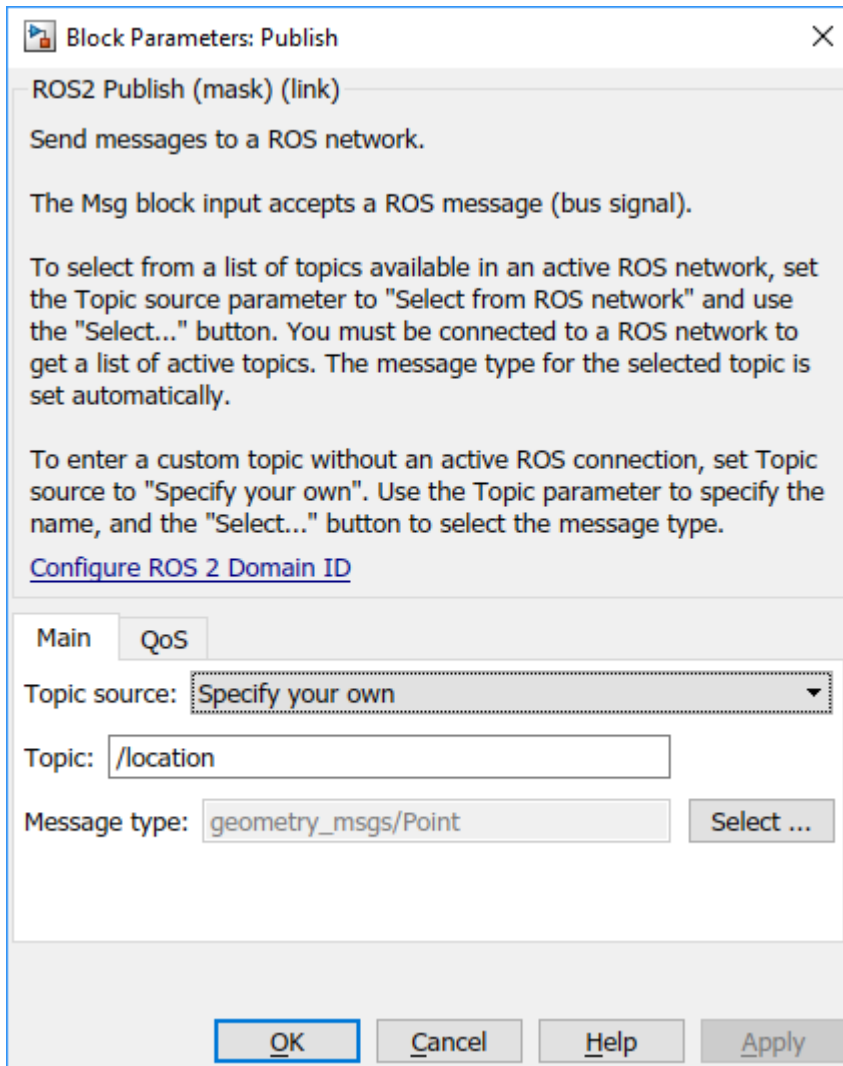
Create a ROS 2 Message

Create a blank ROS 2 message and populate it with the x and y location for the robot path. Then publish the updated ROS 2 message to the ROS 2 network.

A ROS 2 message is represented as a *bus signal* in Simulink. A bus signal is a bundle of Simulink signals, and can also include other bus signals (see the "Simulink Bus Signals" (Simulink) example for an overview). The ROS 2 **Blank Message** block outputs a Simulink bus signal corresponding to a ROS 2 message.

- Click **ROS Toolbox** tab in the Library Browser, or type `roslib` at the MATLAB command line. Select the **ROS 2** Library.

- Drag a **Blank Message** block to the model. Double-click on the block to open the block mask.

- Click on **Select** next to the **Message type** box, and select `geometry_msgs/Point` from the resulting pop-up window. set **Sample time** to 0.01. Click **OK** to close the block mask.

- From the **Simulink > Signal Routing** tab in the Library Browser, drag a **Bus Assignment** block.

- Connect the output port of the **Blank Message** block to the Bus input port of the **Bus Assignment** block. Connect the output port of the **Bus Assignment** block to the input port of **Publish** block.

- Double-click on the **Bus Assignment** block. You should see x, y and z (the signals comprising a `geometry_msgs/Point` message) listed on the left. Select `??? signal1` in the right listbox and click **Remove**. Select both X and Y signals in the left listbox and click **Select**. Click **OK** to apply changes.

NOTE: If you do not see x, y and z listed, close the block mask for the Bus Assignment block, and under the **Modeling** tab, click **Update Model** to ensure that the bus information is correctly propagated. If you see the error, "Selected signal 'signal1' in the Bus Assignment block cannot be found", it indicates that the bus information has not been propagated. Close the Diagnostic Viewer, and repeat the above step.

You can now populate the bus signal with the robot location.

- From the **Simulink > Sources** tab in the Library Browser, drag two **Sine Wave** blocks into the model.
- Connect the output ports of each **Sine Wave** block to the assignment input ports x and y of the **Bus Assignment** block.
- Double-click on the **Sine Wave** block that is connected to input port X. Set the **Phase** parameter to `-pi/2` and click **OK**. Leave the **Sine Wave** block connected to input port Y as default.

Your publisher should look like this:

At this point, the model is set up to publish messages to the ROS 2 network. You can verify this as follows:

- Under the **Simulation** tab, set the simulation stop time to `inf`.
- Click **Run** to start simulation. Simulink creates a dedicated ROS 2 node for the model and a ROS 2 publisher corresponding to the **Publish** block.
- While the simulation is running, type `ros2 node list` in the MATLAB command window. This lists all the nodes available in the ROS network, and includes a node with a name like `/untitled_90580` (the name of the model along with a random number to make it unique).
- While the simulation is running, type `ros2 topic list` in the MATLAB command window. This lists all the topics available in the ROS 2 network, and it includes `/location`.

```
>> ros2 node list
untitled_90580   ←————————— Node created by Simulink
>>
>>
>>
>> ros2 topic list
/clock
/location       ←————————— Topic published from Simulink
/parameter_events
```

- Click **Stop** to stop the simulation. Simulink deletes the ROS 2 node and publisher. In general, the ROS 2 node for a model and any associated publishers and subscribers are automatically deleted at the end of a simulation; no additional clean-up steps are required.

Create a Subscriber

Use Simulink to receive messages sent to the `/location` topic. You will extract the x and y location from the message and plot it in the xy-plane.

- From the **ROS Toolbox** tab in the Library Browser, drag a **Subscribe** block to the model. Double-click on the block.
- Select **Specify your own** in the **Topic source** box, and enter `/location` in the **Topic** box.
- Click **Select** next to the **Message type** box, and select `geometry_msgs/Point` from the pop-up window. set **Sample time** to 0.01. Click **OK** to close the block mask.

The **Subscribe** block outputs a Simulink bus signal, so you need to extract the x and y signals from it.

- From the **Simulink > Signal Routing** tab in the Library Browser, drag a **Bus Selector** block to the model.
- Connect the **Msg** output of the **Subscribe** block to the input port of the **Bus Selector** block.
- From the **Modeling** tab, select **Update Model** to ensure that the bus information is propagated. You may get an error, "Selected signal 'signal1' in the Bus Selector block 'untitled/Bus Selector' cannot be found in the input bus signal". This error is expected, and will be resolved by the next step.
- Double-click on the **Bus Selector** block. Select `??? signal1` and `??? signal2` in the right listbox and click **Remove**. Select both x and y signals in the left listbox and click **Select**. Click **OK**.
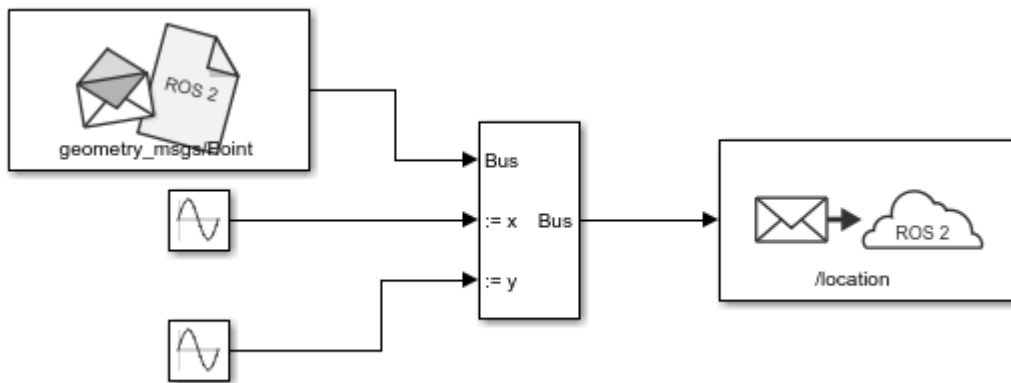
The **Subscribe** block will output the most-recently received message for the topic on every time step. The **IsNew** output indicates whether the message has been received during the prior time step. For the current task, the **IsNew** output is not needed, so do the following:

- From the **Simulink > Sinks** tab in the Library Browser, drag a **Terminator** block to the model.
- Connect the **IsNew** output of the **Subscribe** block to the input of the **Terminator** block.

The remaining steps configure the display of the extracted X and Y signals.

- From the **Simulink > Sinks** tab in the Library Browser, drag an **XY Graph** block to the model. Connect the output ports of the **Bus Selector** block to the input ports of the **XY Graph** block.
- From the **Simulink > Sinks** tab in the Library Browser, drag two **Display** blocks to the model. Connect each output of the **Bus Selector** block to each **Display** block.

- Save your model.

Your entire model should look like this:

Configure and Run the Model

- From the **Modeling** tab, select **Model Settings**. In the **Solver** pane, set **Type** to **Fixed-step** and **Fixed-step size** to 0.01.
- Set simulation stop time to 10.0.
- Click **Run** to start simulation. An XY plot will appear.

The first time you run the model in Simulink, the XY plot may look more jittery than the one above due to delays caused by loading ROS libraries. Once you rerun the simulation a few times, the plot should look smoother.

Note that the simulation **does not** work in actual or "real" time. The blocks in the model are evaluated in a loop that only simulates the progression of time, and is not intended to track actual clock time (for details, see "Simulation Loop Phase" (Simulink)).

Modify the Model to React Only to New Messages

In the above model, the **Subscribe** block outputs a message (bus signal) on every time step; if no messages have been received at all, it outputs a blank message (i.e., a message with zero values). Consequently, the XY coordinates are initially plotted at (0,0).

In this task, you will modify the model to use an **Enabled Subsystem**, so that it plots the location only when a new message is received (for more information, see "Using Enabled Subsystems" (Simulink)). A pre-configured model is included for your convenience.

- In the model, click and drag to select the **Bus Selector** block and **XY Graph** blocks. Right-click on the selection and select **Create Subsystem from Selection**.

- From the **Simulink > Ports & Subsystems** tab in the Library Browser, drag an **Enable** block into the newly-created subsystem.

- Connect the **IsNew** output of the **Subscribe** block to the enabled input of the subsystem as shown in the picture below. Delete the **Terminator** block. Note that the **IsNew** output is true only if a new message was received during the previous time step.

- Save your model.
- Click **Run** to start simulation. You should see the following XY plot.

The blocks in the enabled subsystem are only executed when a new ROS 2 message is received by the **Subscribe** block. Hence, the initial `(0,0)` value will not be displayed in the XY plot.

Connect to a ROS-enabled Robot from Simulink® over ROS 2

This example shows you how to configure a Simulink model to send and receive information from a separate ROS-based simulator such as Gazebo® over ROS 2.

Introduction

You can use Simulink to connect to a ROS-enabled physical robot or to a ROS-enabled robot simulator such as Gazebo. This example shows how to:

- Configure Simulink to connect to a separate robot simulator using ROS 2
- Send velocity commands to a simulated robot
- Receive position information from a simulated robot

Here is the model you will be creating in this example.

```
open_system('robotROS2ConnectToRobotExample');
```

Prerequisites: "Get Started with ROS 2" on page 2-2 , "Exchange Data with ROS 2 Publishers and Subscribers" on page 2-17, "Get Started with ROS 2 in Simulink®" on page 2-38

Task 1 - Start a Gazebo Robot Simulator

In this task, you will start a ROS-based simulator for a differential-drive robot along with a ROS bridge that replays ROS messages over ROS 2 network.

1 Start the Ubuntu® virtual machine desktop from Virtual Machine with ROS 2 Melodic and Gazebo.

2 In the Ubuntu desktop, click the "**Gazebo Empty**" icon to start the Gazebo world.

3 Click on the "**ROS Bridge**" icon, to relay messages between ROS and ROS 2 network.

The simulator receives and sends messages on the following topics:

- Receives `geometry_msgs/Twist` velocity command messages on the /cmd_vel topic.
- Sends `nav_msgs/Odometry` messages to the `/odom` topic.

Task 2 - Configure Simulink to Connect to the ROS 2 Network

- From the **Simulation** tab **Prepare** gallery, click **ROS Network** under **ROS TOOLBOX**.
- In the **Configure ROS Network Addresses** dialog, under **Domain ID (ROS 2)**, set **ID** to 25. This ID value matches the domain ID of the ROS 2 network on the Ubuntu virtual machine, where the messages from Gazebo in ROS network are received.

Task 3 - Send Velocity Commands To the Robot

Create a publisher that sends control commands (linear and angular velocities) to the simulator. Make these velocities adjustable by using **Slider Gain** blocks.

ROS uses a right-handed coordinate system, so X-axis is forward, Y-axis is left and Z-axis is up. Control commands are sent using a `geometry_msgs/Twist` message, where `linear.x` indicates linear forward velocity (in meters/sec), and `angular.z` indicates angular velocity around the Z-axis (in radians/sec).

- Open a new Simulink model.
- On the **Apps** tab, under **CONTROL SYSTEMS**, click **Robot Operating System**.
- In the **ROBOT OPERATING SYSTEM (ROS)** dialog, click **Select a ROS Network** and select **Robot Operating System 2 (ROS 2)**.
- From the **ROS Toolbox > ROS 2** tab in the Library Browser, drag a **Publish** block to the model. Double-click the block.
- Set **Topic source** to **Specify your own**. Enter /cmd_vel in the **Topic** field. Click **Select** next to **Message Type**, select geometry_msgs/Twist from drop down list, and click **OK**.
- From the **ROS Toolbox > ROS 2** tab in the Library Browser, drop a **Blank Message** block to the model. Double-click the block.
- Click **Select** next to **Message type** and select geometry_msgs/Twist.
- Set **Sample time** to 0.01 and click **OK**.
- From the **Simulink > Signal Routing** tab in the Library Browser, drag a **Bus Assignment** block to the model.
- Connect the output of the **Blank Message** block to the Bus input of the **Bus Assignment** block, and the Bus output to the input of the **Publish** block.
- From the **Modeling** tab, click **Update Model** to ensure that the bus information is correctly propagated. Ignore the error, "Selected signal 'signal1' in the Bus Assignment block 'untitled/Bus Assignment' cannot be found in the input bus signal", if it appears. The next step will resolve this error.
- Double-click the **Bus Assignment** block. Select ??? signal1 in the right listbox and click **Remove**. In the left listbox, expand both linear and angular properties. Select **linear > x** and **angular > z** and click **Select**. Click **OK** to close the block mask.

Block Parameters: Bus Assignment ✕

BusAssignment

This block accepts a bus as input and allows signals in the bus to be assigned with new signal values. The left listbox shows the signals in the input bus. Use the Select button to select the signals that are to be assigned. The right listbox shows the selections. Use the Up, Down, or Remove button to reorder the selections.

Parameters

Filter by name		Find	Signals that are being assigned	Up

Signals in the bus   Select>>   linear.x   Down

✓ linear   Refresh   angular.z   Remove
   x
   y
   z
✓ angular
   x
   y
   z

OK   Cancel   Help   Apply
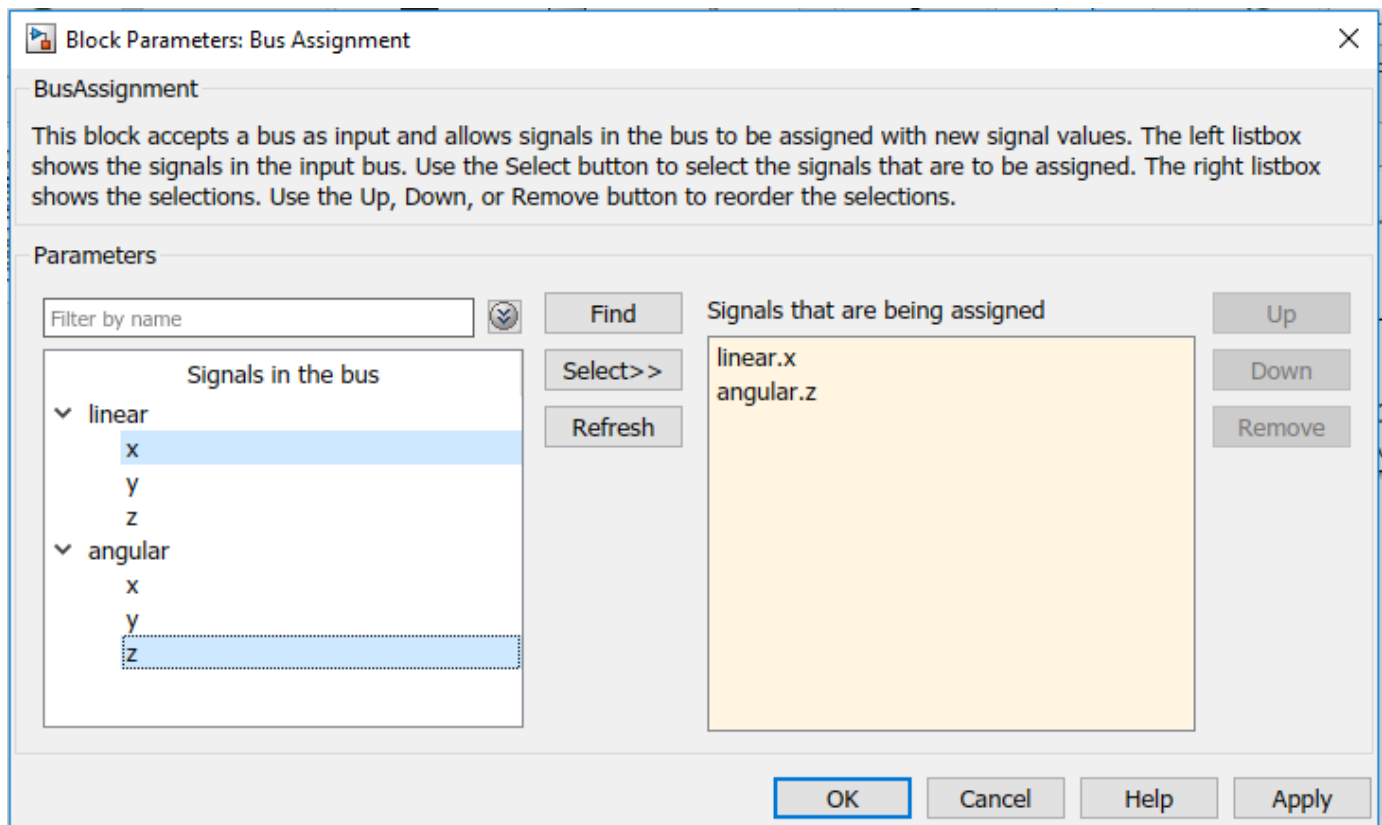
- Add a **Constant** block, a **Gain** block, and two **Slider Gain** blocks. Connect them together as shown in picture below, and set the **Gain** value to -1.
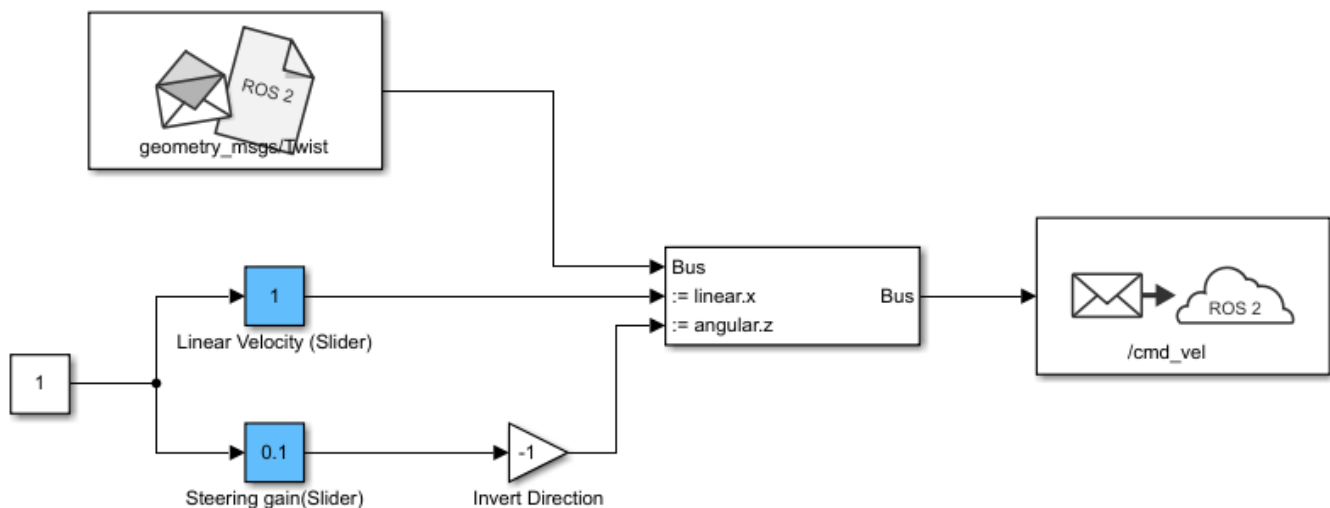
- Set the limits, and current parameters of the linear velocity slider to 0.0 to 2.0, and 1.0 respectively. Set the corresponding parameters of the steering gain slider to -1.0 to 1.0, and 0.1.

Task 4 - Receive Location Information From the Robot

Create a subscriber to receive messages sent to the `/odom` topic. You will also extract the location of the robot and plot it's path in the XY-plane.

- From the **ROS Toolbox > ROS 2** tab in the Library Browser, drag a **Subscribe** block to the model. Double-click the block.

- Set **Topic source** to **Select From ROS network**, and click **Select** next to the **Topic** box. Select "/odom" for the topic and click **OK**. Note that the message type `nav_msgs/Odometry` is set automatically.

- From the **Simulink > Signal Routing** tab in the Library Browser, drag a **Bus Selector** block to the model.

- Connect the `Msg` output port of the **Subscribe** block to the input port of the **Bus Selector** block. In the **Modeling** tab, click **Update Model** to ensure that the bus information is correctly propagated.

- Double-click the **Bus Selector** block. Select `??? signal1` and `??? signal2` in the right listbox and click **Remove**. In the left listbox, expand **pose > pose > position** and select **x** and **y**. Click **Select** and then **OK**.

- From the **Simulink > Sinks** tab in the Library Browser, drag an **XY Graph** block to the model. Connect the X and Y output ports of the **Bus Selector** block to the input ports of the **XY Graph** block.

2-49
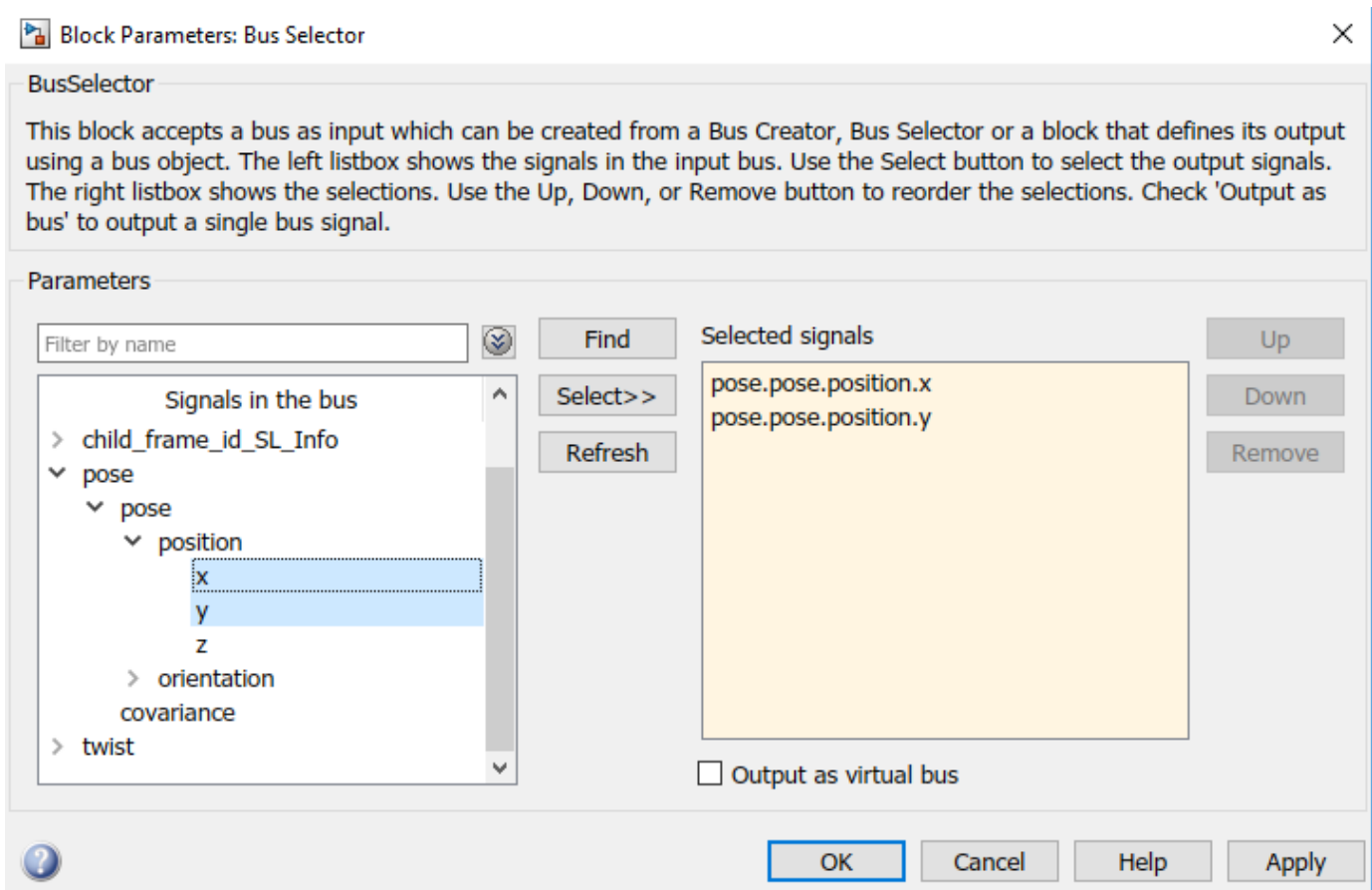
The following figure shows the completed model. A pre-configured model is included for your convenience.
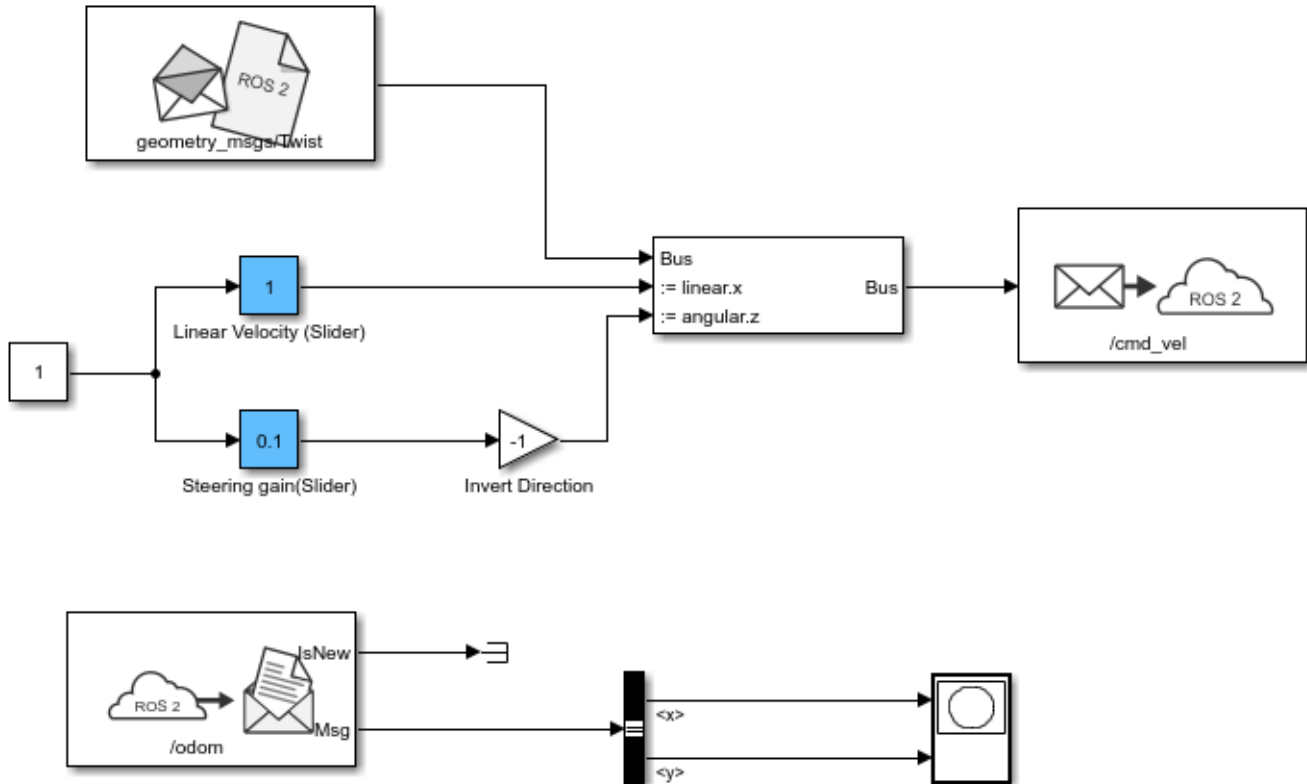
Task 5 - Configure and Run the Model

- From the **Modeling** tab, click **Model Settings**. In the **Solver** pane, set **Type** to **Fixed-step** and **Fixed-step size** to `0.01`.

- Set simulation Stop time to `inf`.

- Click **Run** to start the simulation.

- In both the simulator and XY plot, you should see the robot moving in a circle.

- While the simulation is running, change the values of **Slider Gain** blocks to control the robot. Double-click the **XY Graph** block and change the X and Y axis limits if needed (you can do this while the simulation is running).

- To stop the simulation, click **Stop**.

Feedback Control of a ROS-enabled Robot over ROS 2

This example shows you how to use Simulink® to control a simulated robot running in a Gazebo® robot simulator over ROS 2 network.

Introduction

In this example, you will run a model that implements a simple closed-loop proportional controller. The controller receives location information from a simulated robot and sends velocity commands to drive the robot to a specified location. You will adjust some parameters while the model is running and observe the effect on the simulated robot.

The following diagram summarizes the interaction between Simulink and the robot simulator (the arrows in the diagram indicate ROS 2 message transmission). The /odom topic conveys location information, and the /cmd_vel topic conveys velocity commands.
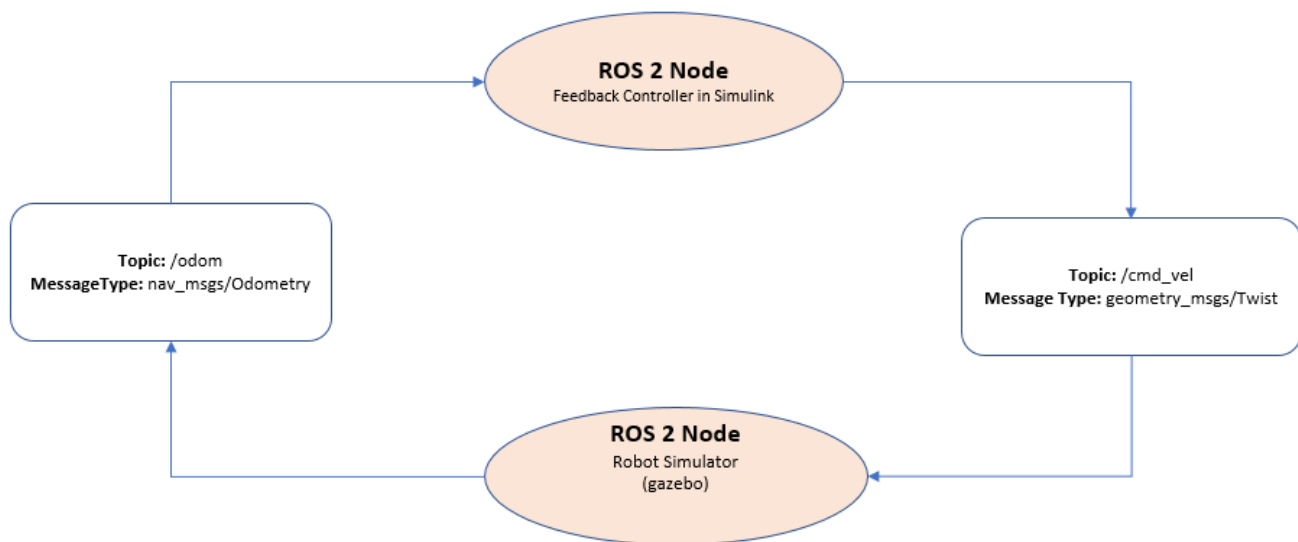
Task 1 - Start a Robot Simulator and Configure Simulink

In this task, you will start a ROS-based simulator for a differential-drive robot, start the ROS bridge configure MATLAB® connection with the robot simulator.

1   Start the Ubuntu® virtual machine desktop from <u>Virtual Machine with ROS 2 Bouncy and Gazebo</u>.

2   In the Ubuntu desktop, click the **Gazebo Empty** icon to start the empty Gazebo world.

3   Click the **ROS Bridge (Dashing)** icon to start the ROS bridge to relay messages between Simulink ROS 2 node and Turtlebot3 ROS-enabled robot.

4   In MATLAB Command Window, set the ROS_DOMAIN_ID environment variable to 25 to match the robot simulator ROS bridge settings and run ros2 topic list to verify that the topics from the robot simulator are visible in MATLAB.

```
setenv('ROS_DOMAIN_ID','25')
ros2('topic','list')
```

```
/clock
/cmd_vel
/gazebo/link_states
/gazebo/model_states
/imu
/joint_states
/odom
/parameter_events
/rosout
/rosout_agg
/scan
/tf
```

Task 2 - Open Existing Model

After connecting to the ROS 2 network, open the example model.

```
open_system('robotROS2FeedbackControlExample.slx');
```

The model implements a proportional controller for a differential-drive mobile robot. On each time step, the algorithm orients the robot toward the desired location and drives it forward. Once the desired location is reached, the algorithm stops the robot.

```
open_system('robotROS2FeedbackControlExample/Proportional Controller');
```

Note that there are four tunable parameters in the model (indicated by colored blocks).

- Desired Position (at top level of model): The desired location in (X,Y) coordinates
- Distance Threshold: The robot is stopped if it is closer than this distance from the desired location
- Linear Velocity: The forward linear velocity of the robot
- Gain: The proportional gain when correcting the robot orientation

The model also has a **Simulation Rate Control** block (at top level of model). This block ensures that the simulation update intervals follow wall-clock elapsed time.

Task 3 - Configure Simulink and Run the Model

In this task, you will configure Simulink to communicate with ROS-enabled robot simulator over ROS 2, run the model and observe the behavior of the robot in the robot simulator.

To configure the network settings for ROS 2.

- Under the **Simulation** tab, in **PREPARE**, select **ROS Toolbox > ROS Network**.
- In **Configure ROS Network Addresses**, set the ROS 2 Domain ID value to *25*.
- Click *OK* to apply changes and close the dialog.

To run the model.

- Position windows on your screen so that you can observe both the Simulink model and the robot simulator.
- Click the Play button in Simulink to start simulation.
- While the simulation is running, double-click on the **Desired Position** block and change the Constant value to [2 3]. Observe that the robot changes its heading.

- While the simulation is running, open the **Proportional Controller** subsystem and double-click on the **Linear Velocity (slider)** block. Move the slider to 2. Observe the increase in robot velocity.
- Click the Stop button in Simulink to stop the simulation.

Task 4 - Observe Rate of Incoming Messages

In this task, you will observe the timing and rate of incoming messages.

- Click the Play button in Simulink to start simulation.
- Open the Scope block. Observe that the **IsNew** output of the **Subscribe** block is always zero, indicating that no messages are being received for the /odom topic. The horizontal axis of the plot indicates simulation time.
- Start Gazebo Simulator in ROS network and start ROS Bridge in ROS 2, so that ROS 2 network able receive messages published by Gazebo Simulator.
- In the Scope display, observe that the **IsNew** output has the value 1 at an approximate rate of 20 times per second, in elapsed wall-clock time.

The synchronization with wall-clock time is due to the **Simulation Rate Control** block. Typically, a Simulink simulation executes in a free-running loop whose speed depends on complexity of the model and computer speed (see Simulation Loop Phase (Simulink)). The **Simulation Rate Control** block attempts to regulate Simulink execution so that each update takes 0.02 seconds in wall-clock time when possible (this is equal to the fundamental sample time of the model). See the comments inside the block for more information.

In addition, the Enabled subsystems for the Proportional Controller and the Command Velocity Publisher ensure that the model only reacts to genuinely new messages. If enabled subsystems were not used, the model would repeatedly process the same (most-recently received) message over and over, leading to wasteful processing and redundant publishing of command messages.

Summary

This example showed you how to use Simulink for simple closed-loop control of a simulated robot. It also showed how to use Enabled subsystems to reduce overhead in the ROS 2 network.

Publish and Subscribe to ROS 2 Messages in Simulink

This model shows how to publish and subscribe to a ROS 2 topic using Simulink®.

Prerequisites: "Get Started with ROS 2 in Simulink®" on page 2-38

```
open_system('simulinkPubSubROS2Example');
```
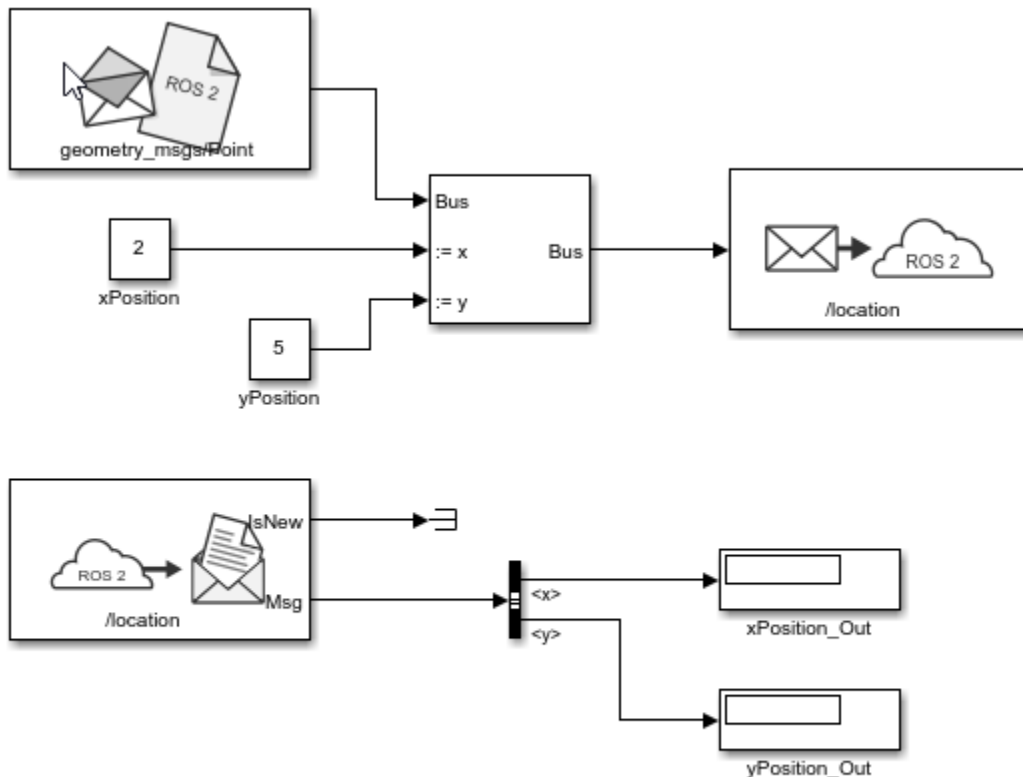

Copyright 2019 The MathWorks, Inc.

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. set Sample time to 0.01. 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'`. Set Sample time to 0.01.

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.

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 2 network.
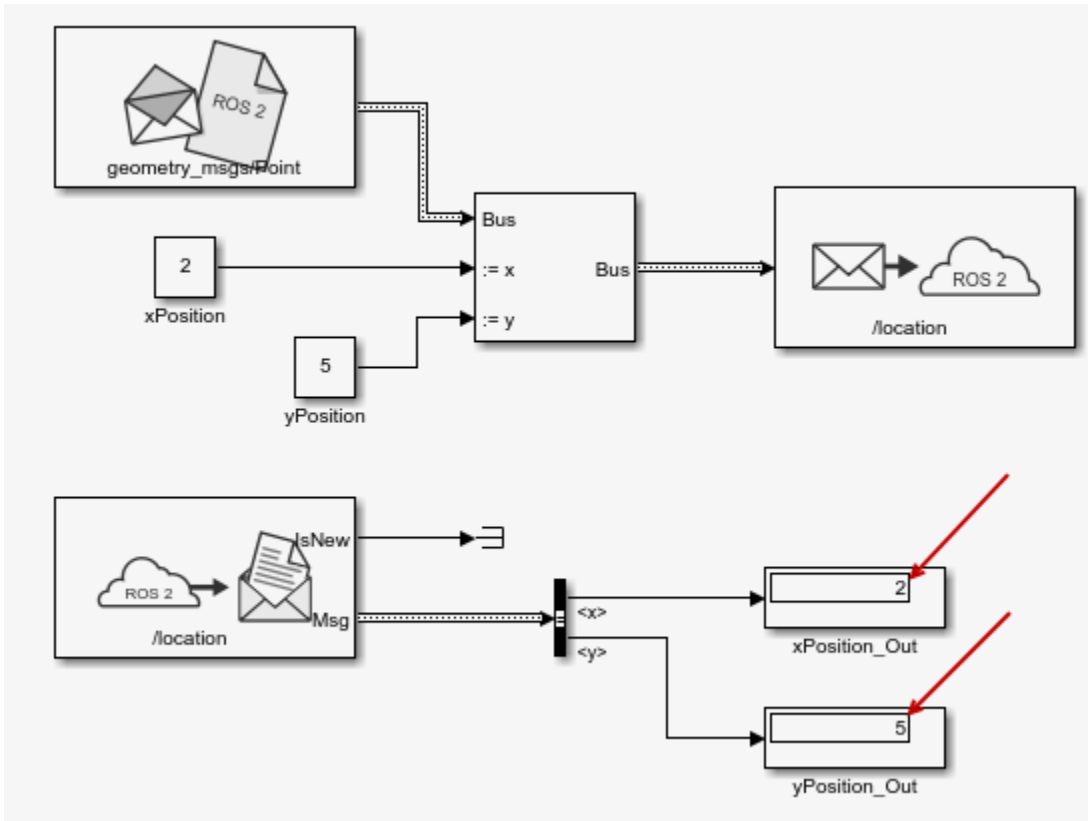
Generate a Standalone ROS 2 Node from Simulink®

This example shows you how to generate and build a standalone ROS 2 node from a Simulink® model. You configure a model to generate C++ code for a standalone ROS 2 node. Then, build and run the ROS 2 node on your host computer.

Prerequisites

- This example requires Simulink Coder™ and Embedded Coder™.
- See Virtual Machine with ROS 2 Bouncy and Gazebo for instructions on setting up a simulated robot.
- Review the "Feedback Control of a ROS-enabled Robot over ROS 2" on page 2-51 example.
- See ROS 2 Model Build Failure in "ROS Simulink Support and Limitations" on page 4-2.
- To ensure you have the proper third-party software, see "ROS System Requirements".

Configure a Model for Code Generation

Configure a model to generate C++ code for a standalone ROS 2 node. The model is the proportional controller introduced in the "Feedback Control of a ROS-enabled Robot over ROS 2" on page 2-51 example.

- Open the robot feedback control model configured for ROS 2. Alternatively, call `open_system("robotControllerROS2")`.
- Under **Robot** tab, click **Hardware settings**. In the **Hardware implementation** pane, **Hardware board settings** section contains settings specific to the generated ROS 2 package, such as information to be included in the `package.xml` file. Change **Maintainer name** to `ROS 2 Example User`.
- The model requires variable-sized arrays. To enable this option, check **variable-size signals** under **Code Generation > Interface > Software environment**.
- In the **Solver** pane, ensure that Solver **Type** is set to **Fixed-step**, and set **Fixed-step size** to `0.05`. In generated code, the Fixed-step size defines the actual time step, in seconds, that is used for the model update loop (see "Execution of Code Generated from a Model" (Simulink Coder)). It can be made smaller (e.g., 0.001 or 0.0001) but for current purposes 0.05 is sufficient.
- Click **OK**.

Generate the C++ ROS 2 Node

In this task, you generate code for a standalone ROS 2 node, and automatically build, and run it on the host computer.

- In MATLAB®, change the current folder to a location where you have write permission.
- Under the **Simulation** tab, in **Prepare**, select **ROS Toolbox > ROS Network**.
- Set the Domain ID (ROS 2) of ROS 2 network. This example uses Domain ID as 25.
- In **Robot** tab, from the **Deploy** section dropdown, click **Build & Run**. If you get any errors about bus type mismatch, close the model, clear all variables from the base MATLAB workspace, and re-open the model. Click on the **View Diagnostics** link at the bottom of the model toolbar to see the output of the build process.

Once the code generation completes, the ROS 2 node builds in the present working folder starts to run automatically. When running in Windows®, a Command window opens. Do not close the window, but use **Ctrl+C** to shutdown the ROS 2 node.

Use `ros2 node` to list all running nodes is the ROS 2 network. `robotControllerROS2` should be in the displayed list of nodes.

```
ros2('node','list')
```

```
/robotControllerROS2
/ros_bridge
```

Verify that the deployed node publishes data on the ROS 2 topic, `/cmd_vel`, to control the motion of simulated robot.

```
ros2('topic','list')
```

```
/clock
/cmd_vel
/gazebo/link_states
/gazebo/model_states
/imu
/joint_states
/odom
/parameter_events
/rosout
/rosout_agg
/scan
/tf
```

Generate Code to Manually Deploy a ROS 2 Node from Simulink®

This example shows you how to generate C++ code from a Simulink® model to deploy as a standalone ROS 2 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 2 node, see "Generate a Standalone ROS 2 Node from Simulink®" on page 2-57.

Prerequisites

- This example requires Simulink Coder™ and Embedded Coder™ .
- If this is your first time deploying a ROS node, check the "ROS System Requirements".
- 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 ROS Toolbox™ examples. See "Get Started with Gazebo and a Simulated TurtleBot" on page 1-102 for instructions on setting up a simulated robot.
- Review the "Feedback Control of a ROS-enabled Robot over ROS 2" on page 2-51 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 2 node. The model is the proportional controller introduced in the "Feedback Control of a ROS-enabled Robot over ROS 2" on page 2-51 example.

- Open the robot feedback control model configured for ROS 2.

```
open_system("robotFeedbackControllerROS2");
```

- Under **Robot** tab, click **Hardware settings**. In the **Hardware implementation** pane, **Hardware board settings** section contains settings specific to the generated ROS 2 package, such as information to be included in the `package.xml` file. Change **Maintainer name** to `ROS 2 Example User`, click **Apply**.
- The model requires variable-sized arrays. To enable this option, check **variable-size signals** under **Code Generation > Interface > Software environment**.
- In the **Solver** pane, ensure that the solver **Type** is set to `Fixed-step`, and set **Fixed-step size** to `0.05`. In generated code, the Fixed-step size defines the actual time step, in seconds, that is used for the model update loop (see "Execution of Code Generated from a Model" (Simulink Coder) (Simulink Coder)). It can be made smaller (e.g., 0.001 or 0.0001) but for current purposes 0.05 is sufficient.
- Click **OK**.

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. Under the **Modeling** tab, click **Model Settings**.

In the **Hardware Implementation** tab, under **Target hardware resources**, click the **Build options** group. Set the **Build action** to Build. This setting ensures that code generated for the ROS 2 node without building it on an external ROS 2 device.

Generate and Deploy the Code

In this task, you generate source code for ROS 2 node, manually deploy to Ubuntu Linux system, and build it on the Linux system.

- In MATLAB®, change the current folder to a location where you have write permission.
- Under the **Simulation** tab, in **Prepare**, select **ROS Toolbox > ROS Network**.
- Set the Domain ID (ROS 2) of ROS 2 network. This example uses Domain ID as 25.
- In **Robot** tab, from the **Deploy** section, click **Build Model**. If you get any errors about bus type mismatch, close the model, clear all variables from the base MATLAB workspace, and re-open the model. Click on the **View Diagnostics** link at the bottom of the model toolbar to see the output of the build process.

Once the build completes, a src folder which contains the package source code will be written to your folder.
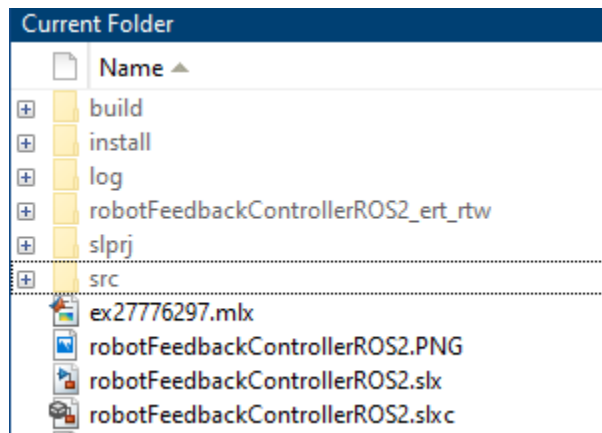
Compress the src folder to a tar file by executing the following command in MATLAB Command Window:

```
>> tar('src.tar','src');
```

After generating the tar file, manually transfer it to the target machine. This example assumes you are using the virtual machine from "Get Started with Gazebo and a Simulated TurtleBot" on page 1-102. 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 src.tar file) 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 src.tar user@<virtual_machine_ip>:
```

- Linux or macOS host systems:

```
scp src.tar user@<virtual_machine_ip>:
```

Build and Run the ROS 2 Node

On the Linux system, execute the following commands to create a Catkin workspace and decompress the source code. You may use an existing Catkin workspace.

```
mkdir ~/ros2_ws_simulink
tar -C ~/ros2_ws_simulink/ -xvf ~/src.tar
```

Build the ROS 2 node using the following command in Linux. Replace `<path_to_catkin_ws>` with the path to your catkin workspace. In this example, the `<path_to_catkin_ws>` would be `~/ros2_ws_simulink`. (Note: There might be some warnings such as unused parameters during the build process. These parameters are needed only for Simulink environment, it would not affect the build process.)

```
cd <path_to_catkin_ws>
source /opt/ros/dashing/local_setup.sh
colcon build
```

Verify that the node executable was created using:

```
file ~/ros2_ws_simulink/install/robotfeedbackcontrollerros2/lib/robotfeedbackcontrollerros2/robot
```

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 2 node on your device.

(Optional) You can then run the node using these commands. Replace `<path_to_catkin_ws>` with the path to your catkin workspace.

Double click **Gazebo Empty** and **ROS Bridge** on virtual machine desktop to set up the Gazebo environment. Setup environment variables and run the ROS 2 node using:

```
export ROS_DOMAIN_ID=25
source /opt/ros/dashing/local_setup.sh
~/<path_to_catkin_ws>/install/robotfeedbackcontrollerros2/lib/robotfeedbackcontrollerros2/robotFe
```

Note: It is possible that the robot spins at an unexpected location, this is because the pose and world is offset in Gazebo. Restart virtual machine and rerun Gazebo and the node.

You can also use `ros2 node` to list all running nodes in the ROS 2 network. `robotFeedbackControllerROS2` should be in the displayed list of nodes.

```
ros2('node','list')
```

Verify that this ROS 2 node publishes data on the ROS 2 topic, `/cmd_vel`, to control the motion of simulated robot.

```
ros2('topic','list')
```

If you cannot see the expected node and topic, try to set `ROS_DOMAIN_ID` using the `setenv` command in MATLAB Command Window.

```
setenv("ROS_DOMAIN_ID","25")
```

Sign Following Robot with ROS in MATLAB

This example shows you how to use MATLAB® to control a simulated robot running on a separate ROS-based simulator over a ROS network. The example shown here uses ROS and MATLAB. For the other examples with ROS 2 or Simulink®, see:

- "Sign Following Robot with ROS in Simulink" on page 2-68
- "Sign Following Robot with ROS 2 in MATLAB" on page 2-70
- "Sign Following Robot with ROS 2 in Simulink" on page 2-75

In this example, you run MATLAB script that implements a sign-following algorithm and controls the simulated robot to follow a path based on signs in the environment. The algorithm receives the location information and camera information from the simulated robot, which is running in a separate ROS-based simulator. The algorithm detects the color of the sign and sends the velocity commands to turn the robot based on the color. In this example, the algorithm is designed to turn left when robot encounters a blue sign and turn right when robot encounters a green sign. Finally the robot stops when it encounters a red sign.

Connect to a Robot Simulator

Start a ROS-based simulator for a differential-drive robot and configure MATLAB® connection with the robot simulator.

This example uses a virtual machine (VM) available for download at <u>Virtual Machine with ROS 2 and Gazebo</u>.

- Start the Ubuntu® virtual machine desktop.
- In the Ubuntu desktop, click the **Gazebo Sign Follower ROS** icon to start the Gazebo world built for this example.
- Specify the IP address and port number of the ROS master in Gazebo so that MATLAB® can communicate with the robot simulator. For this example, the ROS master in Gazebo is `http://192.168.203.131:11311` and your host computer address is `192.168.31.1`.
- Start the ROS 1 network using `rosinit`.

```
masterIP = '192.168.203.131';
setenv('ROS_IP','192.168.31.1');
setenv('ROS_MASTER_URI',['http://' masterIP ':11311']);

rosinit(masterIP,11311)
```

```
The value of the ROS_IP environment variable, 192.168.31.1, will be used to set the advertised ad
Initializing global node /matlab_global_node_46296 with NodeURI http://192.168.31.1:57058/
```

Setup ROS Communication

Create publishers and subscribers to relay messages to and from the robot simulator over ROS network. You need subscribers for the image and odometry data. To control the robot, set up a publisher to send velocity commands using `/cmd_vel`.

```
imgSub = rossubscriber("/camera/rgb/image_raw");

odomSub = rossubscriber("/odom");

[velPub, velMsg] = rospublisher("/cmd_vel", "geometry_msgs/Twist");
```

Define the image processing color threshold parameters. Each row defines the threshold values for the different colors.

```
colorThresholds = [100 255 0 55 0 50; ... % Red
                     0 50 50 255 0 50; ... % Green
                     0 40 0 55 50 255]';    % Blue
```

Create Sign Following Controller Using Stateflow® Chart

This example provides an example helper MATLAB Stateflow® chart that takes in the image size, coordinates from processed image, and the robot odometry poses. The chart provides linear and angular velocity to drive the robot based on these inputs.

```
controller = ExampleHelperSignFollowingControllerChart;
open('ExampleHelperSignFollowingControllerChart');
```

Run Control Loop

This section runs the controller to receive images and move the robot to follow the sign. The controller does the following steps:

- Gets the latest image and odometry message from the ROS network.
- Runs the algoritm for detecting image features (`ExampleHelperSignFollowingProcessImg`).
- Generates control commands from the Stateflow® chart using `step`.
- Publishes the velocity control commands to the ROS network.

To visualize the masked image the robot sees, change the value of `doVisualization` variable to `true`.

```
ExampleHelperSignFollowingSetupPreferences;

% Control the visualization of the mask
doVisualization = false;

r = rateControl(10);
receive(imgSub); % Wait to receive an image message before starting the loop
receive(odomSub);
while(~controller.done)
    % Get latest sensor messages and process them
    imgMsg = imgSub.LatestMessage;
    odomMsg = odomSub.LatestMessage;
    [img,pose] = ExampleHelperSignFollowingROSProcessMsg(imgMsg, odomMsg);

    % Run vision and control functions
    [mask,blobSize,blobX] = ExampleHelperSignFollowingProcessImg(img, colorThresholds);
    step(controller,'blobSize',blobSize,'blobX',blobX,'pose',pose);
    v = controller.v;
    w = controller.w;

    % Publish velocity commands
    velMsg.Linear.X = v;
    velMsg.Angular.Z = w;
    send(velPub,velMsg);

    % Optionally visualize
    % NOTE: Visualizing data will slow down the execution loop.
    % If you have Computer Vision Toolbox, we recommend using
    % vision.DeployableVideoPlayer instead of imshow.
    if doVisualization
        imshow(mask);
        title(['Linear Vel: ' num2str(v) ' Angular Vel: ' num2str(w)]);
        drawnow('limitrate');
    end
    % Pace the execution loop.
    waitfor(r);
end
```

You should see the robot moving in the ROS-based robot simulator as shown below.

Sign Following Robot with ROS in Simulink

This example shows how to use Simulink® to control a simulated robot running on a separate ROS-based simulator.

In this example, you run a model that implements a sign-following algorithm and controls the simulated robot to follow a path based on signs in the environment. The algorithm receives the location information and camera information from the simulated robot, which is running in a separate ROS-based simulator. The algorithm detects the color of the sign and sends the velocity commands to turn the robot based on the color. In this example, the algorithm is designed to turn left when robot encounters a blue sign and turn right when robot encounters a green sign. FInally the robot stops when it encounters a red sign.

To see this example using ROS 2 or MATLAB®, see "Sign Following Robot with ROS in MATLAB" on page 2-63.

Start Robot Simulator

Start a ROS-based simulator for a differential-drive robot and configure the Simulink® connection with the robot simulator.

This example uses a virtual machine (VM) available for download at <u>Virtual Machine with ROS 2 Melodic and Gazebo</u>.

- Start the Ubuntu® virtual machine desktop.
- In the Ubuntu desktop, click the **Gazebo Sign Follower ROS** icon to start the Gazebo world built for this example.

Open Model and Configure Simulink

Setup the Simulink ROS preferences to communicate with the robot simulator.

Open the <u>example model</u>.

```
open_system('signFollowingRobotROS.slx');
```

To configure the network settings for ROS.

- From **Simulation** tab, **Prepare** group, select **ROS Network**.
- Specify the IP address and port number of the ROS master in Gazebo. For this example, the ROS master in Gazebo is `192.168.203.128:11311`. Enter `192.168.203.128` in the **Hostname/IP address** box and `11311` in the **Port Number** box.
- Click *OK* to apply changes and close the dialog.

On each time step, the algorithm detects a sign from the camera feed, decides on turn and drives it forward. The sign detection is done in the **Image Proecessing** subsystem of the model.

```
open_system('signFollowingRobotROS/Image Processing');
```

The **Sign Tracking Logic** subsystem implements a Stateflow® chart that takes in the detected image size and coordinates from **Image Processing** and provides linear and angular velocity to drive the robot.

```
open_system('signFollowingRobotROS/Sign Tracking Logic');
```

Run the Model

Run the model and observe the behavior of the robot in the robot simulator.

- The video viewers show the actual camera feed and the detected sign image.
- In the simulator, the robot follows the sign and turns based on the color.
- The simulation stops automatically once the robot reaches the red sign at the end.
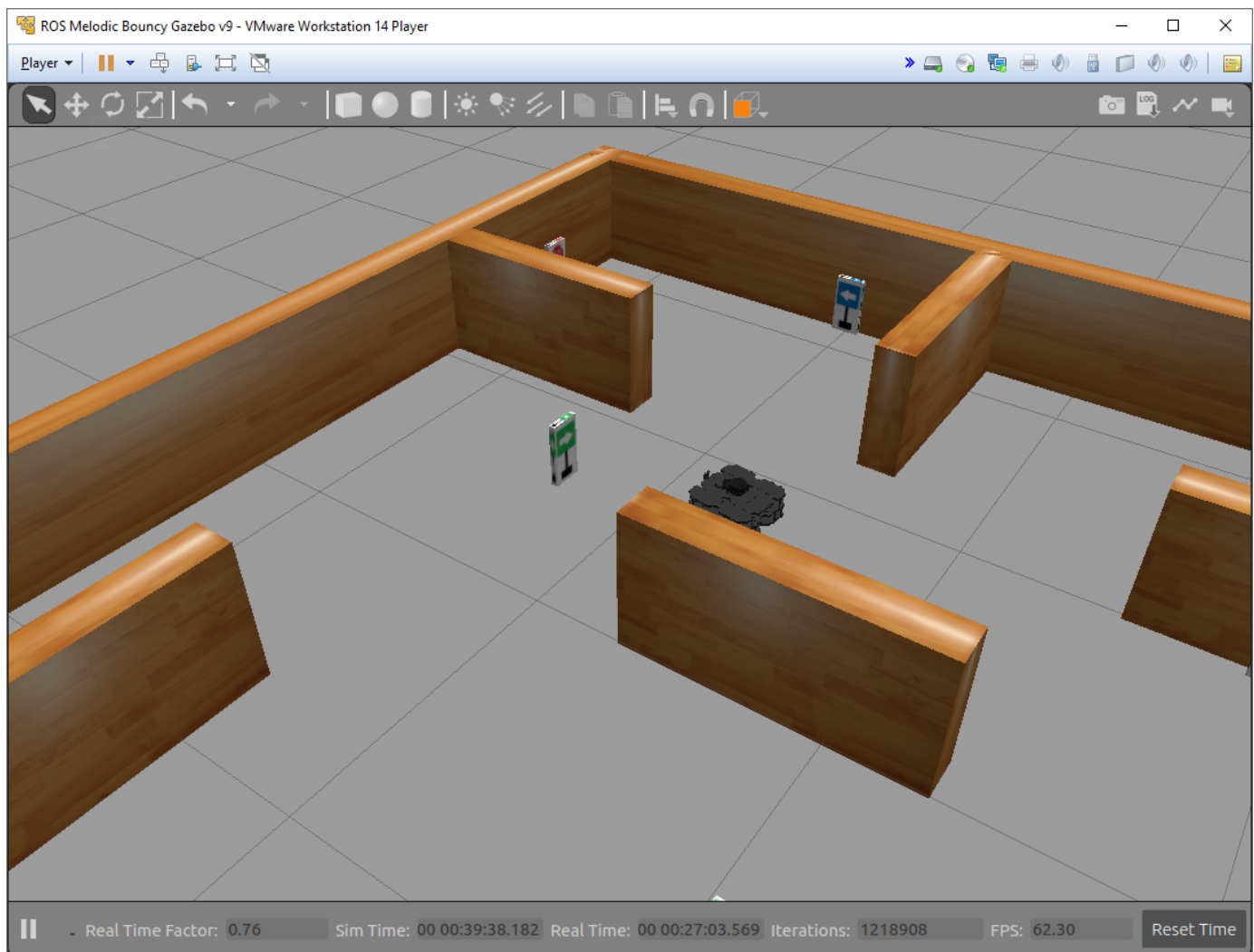
Sign Following Robot with ROS 2 in MATLAB

This example shows you how to use MATLAB® to control a simulated robot running on a separate ROS-based simulator over ROS 2 network.

In this example, you run a MATLAB script that implements a sign-following algorithm and controls the simulated robot to follow a path based on signs in the environment. The algorithm receives the location information and camera information from the simulated robot, which is running in a separate ROS-based simulator. The algorithm detects the color of the sign and sends the velocity commands to turn the robot based on the color. In this example, the algorithm is designed to turn left when robot encounters a blue sign and turn right when robot encounters a green sign. FInally the robot stops when it encounters a red sign.

To see this example using ROS 1 or Simulink®, see "Sign Following Robot with ROS in MATLAB" on page 2-63.

Start a Robot Simulator and ROS Bridge

Start a ROS-based simulator for a differential-drive robot and configure the MATLAB® connection with the robot simulator. Because the simulator uses ROS 1 topics, start the ROS bridge to share the topics with a ROS 2 network.

This example uses a virtual machine (VM) available for download at <u>Virtual Machine with ROS 2 Melodic and Gazebo</u>.

- Start the Ubuntu® virtual machine desktop.
- In the Ubuntu desktop, click the **Gazebo Sign Follower ROS** icon to start the Gazebo world built for this example.
- Click the **ROS Bridge (Dashing)** icon to start the ROS bridge to relay messages between your ROS 2 node and the Turtlebot3 ROS-enabled robot.
- In MATLAB Command Window, set the `ROS_DOMAIN_ID` environment variable to 25 to match the robot simulator ROS bridge settings and run `ros2 topic list` to verify that the topics from the robot simulator are visible in MATLAB.

```
setenv('ROS_DOMAIN_ID','25')
ros2('topic','list')
```

```
/camera/rgb/camera_info
/camera/rgb/image_raw
/camera/rgb/image_raw/compressed
/clock
/gazebo/link_states
/gazebo/model_states
/imu
/joint_states
/odom
/parameter_events
/rosout
/rosout_agg
/scan
/tf
```

Setup ROS 2 Communication

Create a ROS 2 node using the specified domain ID.

```
domainID = 25;
n = ros2node("matlab_example_robot",domainID);
```

Create publishers and subscribers to relay messages to and from the robot simulator over the ROS 2 network. You need subscribers for the image and odometry data. To control the robot, set up a publisher to send velocity commands using the /cmd_vel topic.

```
imgSub = ros2subscriber(n, "/camera/rgb/image_raw");
```

```
odomSub = ros2subscriber(n, "/odom");
```

```
[velPub, velMsg] = ros2publisher(n, "/cmd_vel", "geometry_msgs/Twist");
```

Define the image processing color threshold parameters. Each row defines the threshold values for the different colors.

```
colorThresholds = [100 255 0 55 0 50; ...  % Red
                   0 50 50 255 0 50; ...   % Green
                   0 40 0 55 50 255]';     % Blue
```

Create a sign following controller stateflow chart

This example provides an example helper MATLAB Stateflow® chart that takes in the image size, coordinates from processed image, and the robot odometry poses. The chart provides linear and angular velocity to drive the robot based on these inputs.

```
controller = ExampleHelperSignFollowingControllerChart;
open('ExampleHelperSignFollowingControllerChart');
```

Run Control Loop

This section runs the controller to receive images and move the robot to follow the sign. The controller does the following steps:

- Gets the latest image and odometry message from the ROS network.
- Runs the algoritm for detecting image features (`ExampleHelperSignFollowingProcessImg`).
- Generates control commands from the Stateflow® chart using `step`.
- Publishes the velocity control commands to the ROS network.

To visualize the masked image the robot sees, change the value of `doVisualization` variable to `true`.

```matlab
ExampleHelperSignFollowingSetupPreferences;

% Control the visualization of the mask
doVisualization = false;

r = rateControl(10);
receive(imgSub); % Wait to receive an image message before starting the loop
receive(odomSub);
while(~controller.done)
    % Get latest sensor messages and process them
    imgMsg = imgSub.LatestMessage;
    odomMsg = odomSub.LatestMessage;
    [img,pose] = ExampleHelperSignFollowingProcessMsg(imgMsg, odomMsg);

    % Run vision and control functions
    [mask,blobSize,blobX] = ExampleHelperSignFollowingProcessImg(img, colorThresholds);
    step(controller,'blobSize',blobSize,'blobX',blobX,'pose',pose);
    v = controller.v;
    w = controller.w;

    % Publish velocity commands
    velMsg.linear.x = v;
    velMsg.angular.z = w;
    send(velPub,velMsg);

    % Optionally visualize
    % NOTE: Visualizing data will slow down the execution loop.
    % If you have Computer Vision Toolbox, we recommend using
    % vision.DeployableVideoPlayer instead of imshow.
    if doVisualization
        imshow(mask);
        title(['Linear Vel: ' num2str(v) ' Angular Vel: ' num2str(w)]);
        drawnow('limitrate');
    end
    % Pace the execution loop.
    waitfor(r);
end
```

You should see the robot moving in the ROS-based robot simulator as shown below.

Sign Following Robot with ROS 2 in Simulink

Use Simulink® to control a simulated robot running on a separate ROS-based simulator over ROS 2 network.

In this example, you run a model that implements a sign-following algorithm and controls the simulated robot to follow a path based on signs in the environment. The algorithm receives the location information and camera information from the simulated robot, which is running in a separate ROS-based simulator. The algorithm detects the color of the sign and sends the velocity commands to turn the robot based on the color. In this example, the algorithm is designed to turn left when robot encounters a blue sign and turn right when robot encounters a green sign. FInally the robot stops when it encounters a red sign.

To see this example using ROS 1 or MATLAB®, see "Sign Following Robot with ROS in MATLAB" on page 2-63.

Start Robot Simulator

Start a ROS-based simulator for a differential-drive robot and configure the Simulink® connection with the robot simulator.

This example uses a virtual machine (VM) available for download at <u>Virtual Machine with ROS 2 Bouncy and Gazebo</u>.

- Start the Ubuntu® virtual machine desktop.
- In the Ubuntu desktop, click the **Gazebo Sign Follower ROS** icon to start the Gazebo world built for this example.
- Click the **ROS Bridge (Dashing)** icon to start the ROS bridge to relay messages between your ROS 2 node and the Turtlebot3 ROS-enabled robot.
- In MATLAB Command Window, set the ROS_DOMAIN_ID environment variable to 25 to match the robot simulator ROS bridge settings and run ros2 topic list to verify that the topics from the robot simulator are visible in MATLAB.

```
setenv('ROS_DOMAIN_ID','25')
ros2('topic','list')
```

```
/parameter_events
```

Open Model and Configure Simulink

Setup the Simulink ROS preferences to communicate with the robot simulator.

Open the <u>example model</u>.

```
open_system('signFollowingRobotROS2.slx');
```

To configure the network settings for ROS 2.

- Under the **Simulation** tab, in **PREPARE**, select **ROS Toolbox > ROS Network**.
- In **Configure ROS Network Addresses**, set the ROS 2 Domain ID value to *25*.
- Click *OK* to apply changes and close the dialog.

On each time step, the algorithm detects a sign from the camera feed, decides on turn and drives it forward. The sign detection is done in the **Image Proecessing** subsystem of the model.

```
open_system('signFollowingRobotROS2/Image Processing');
```

The **Sign Tracking Logic** subsystem implements a Stateflow® chart that takes in the detected image size and coordinates from **Image Processing** and provides linear and angular velocity to drive the robot.

```
open_system('signFollowingRobotROS2/Sign Tracking Logic');
```

Run the Model

Run the model and observe the behavior of the robot in the robot simulator.

- The video viewers show the actual camera feed and the detected sign image.
- In the simulator, the robot follows the sign and turns based on the color.
- The simulation stops automatically once the robot reaches the red sign at the end.
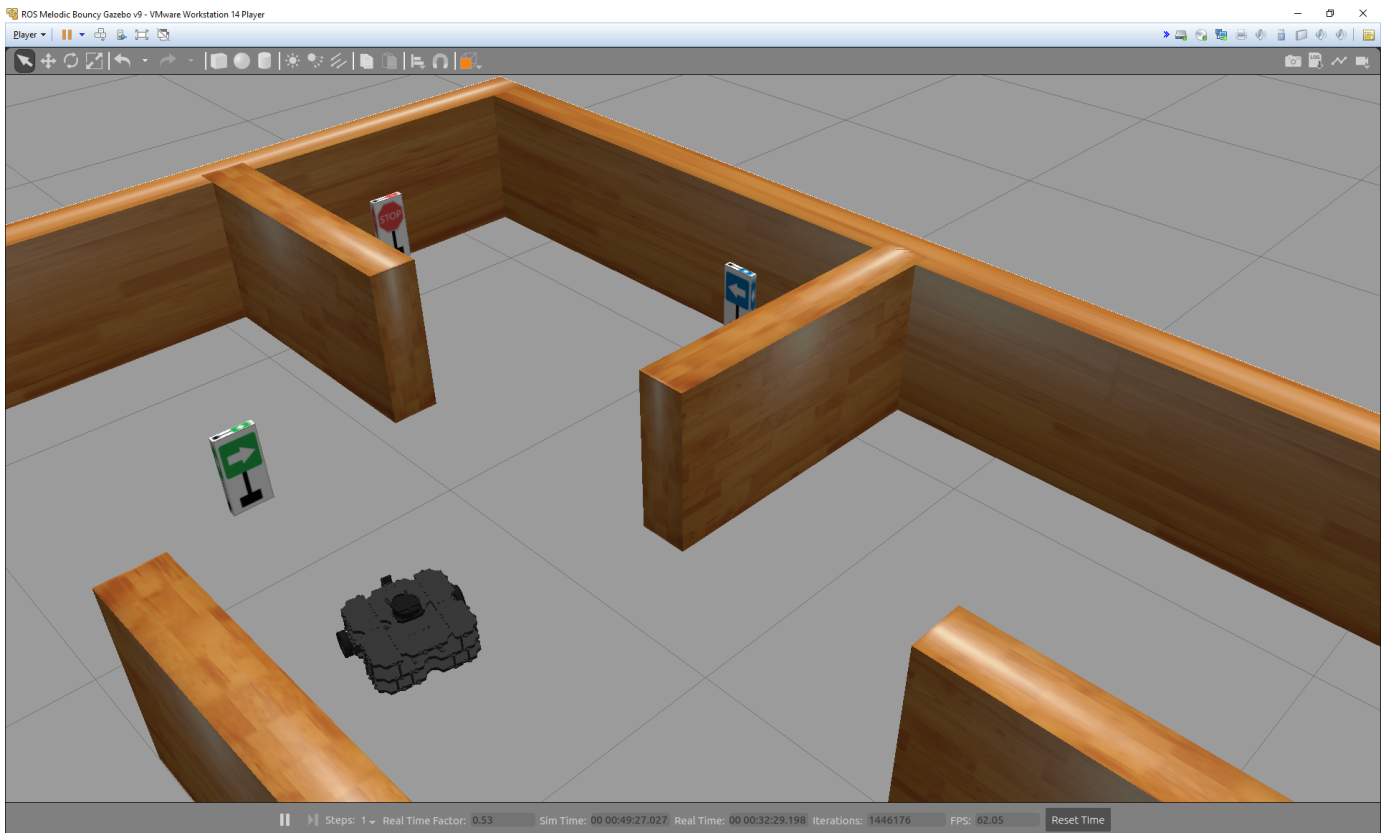
Automated Parking Valet with ROS in MATLAB

This example shows how to distribute the "Automated Parking Valet" (Automated Driving Toolbox) application among various nodes in a ROS network. Depending on your system, this example is provided for ROS and ROS 2 networks using either MATLAB® or Simulink® . The example shown here uses ROS and MATLAB. For the other examples, see:
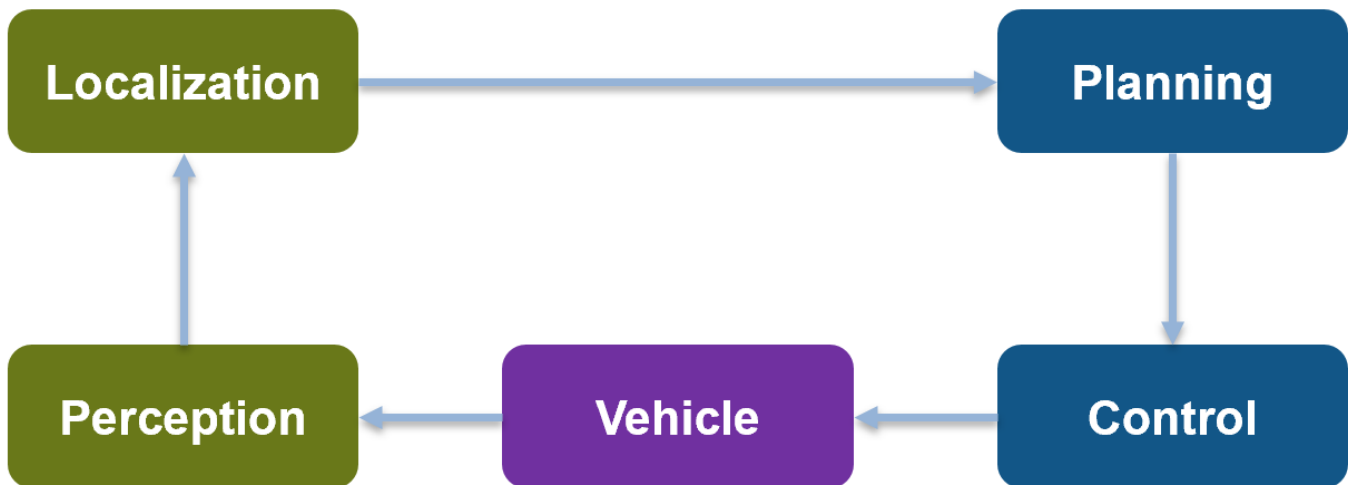
Overview

This example is an extension of the "Automated Parking Valet" (Automated Driving Toolbox) example in Automated Driving Toolbox™. A typical autonmous application has the following components.

For simplicity, this example concentrates on Planning, Control, and a simplified Vehicle Model. The example uses prerecorded data to substitute localization information.

This application demonstrates a typical split of various functions into ROS nodes. The following picture shows how the above example is split into various nodes. Each node: Planning, Control and Vehicle is a ROS node implementing the functionalities shown as below. The interconnections between the nodes show the topics used on each interconnection of the nodes.

Setup

First, load a route plan and a given costmap used by the behavior planner and path analyzer. Behavior Planner, Path Planner, Path Analyzer, Lateral and Lognitudinal Controllers are implemented by helper classes, which are setup with this example helper function call.

```
exampleHelperROSValetSetupGlobals;
```

The initialized globals are organized as fields in the global structure, `valet`.

```
disp(valet)
```

```
            mapLayers: [1×1 struct]
              costmap: [1×1 vehicleCostmap]
           vehicleDims: [1×1 vehicleDimensions]
      maxSteeringAngle: 35
                 data: [1×1 struct]
             routePlan: [4×3 table]
           currentPose: [4 12 0]
            vehicleSim: [1×1 ExampleHelperROSValetVehicleSimulator]
      behavioralPlanner: [1×1 ExampleHelperROSValetBehavioralPlanner]
         motionPlanner: [1×1 pathPlannerRRT]
              goalPose: [56 11 0]
               refPath: [1×1 driving.Path]
       transitionPoses: [14×3 double]
            directions: [522×1 double]
            currentVel: 0
       approxSeparation: 0.1000
         numSmoothPoses: 522
               maxSpeed: 5
             startSpeed: 0
               endSpeed: 0
               refPoses: [522×3 double]
            cumLengths: [522×1 double]
            curvatures: [522×1 double]
          refVelocities: [522×1 double]
             sampleTime: 0.1000
          lonController: [1×1 ExampleHelperROSValetLongitudinalController]
```

```
       controlRate: [1×1 ExampleHelperROSValetFixedRate]
       pathAnalyzer: [1×1 ExampleHelperROSValetPathAnalyzer]
           parkPose: [36 44 90]
```

Initialize the ROS network.

```
rosinit;
```

```
Launching ROS Core...
..................................................................................Done in 5.3625
```
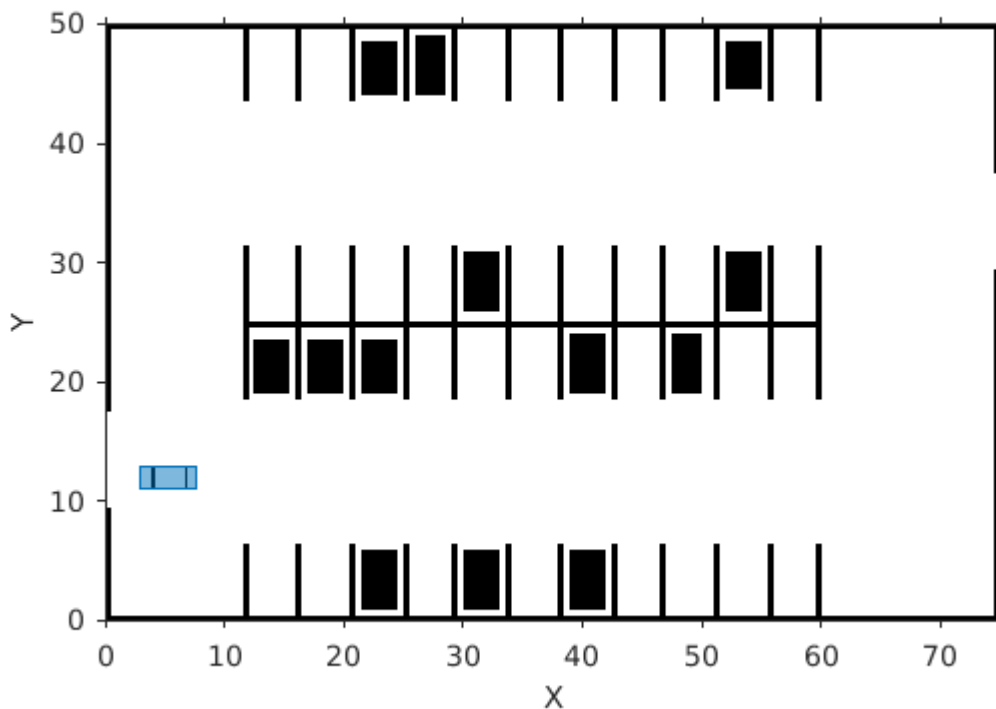


```
Initializing ROS master on http://192.168.0.10:60903.
Initializing global node /matlab_global_node_49911 with NodeURI http://sbd508773glnxa64:42335/
```

```
masterHost = 'localhost';
```

The functions in the application are distributed amongst ROS nodes. This example uses three ROS nodes: `planningNode`, `controlNode`, and `vehicleNode`.

Planning

The Planning node calculates each path segment based on the current vehicle position. This node is responsible for generating the smooth path and publishes the path to the network.

This node publishes these topics:

- /smoothpath

- /velprofile
- /directions
- /speed
- /nextgoal

The node subscribes to these topics:

- /currentvel
- /currentpose
- /desiredvel
- /reachgoal

On receiving a /reachgoal message, the node runs the exampleHelperROS2ValetPlannerCallback callback, which plans the next segment.
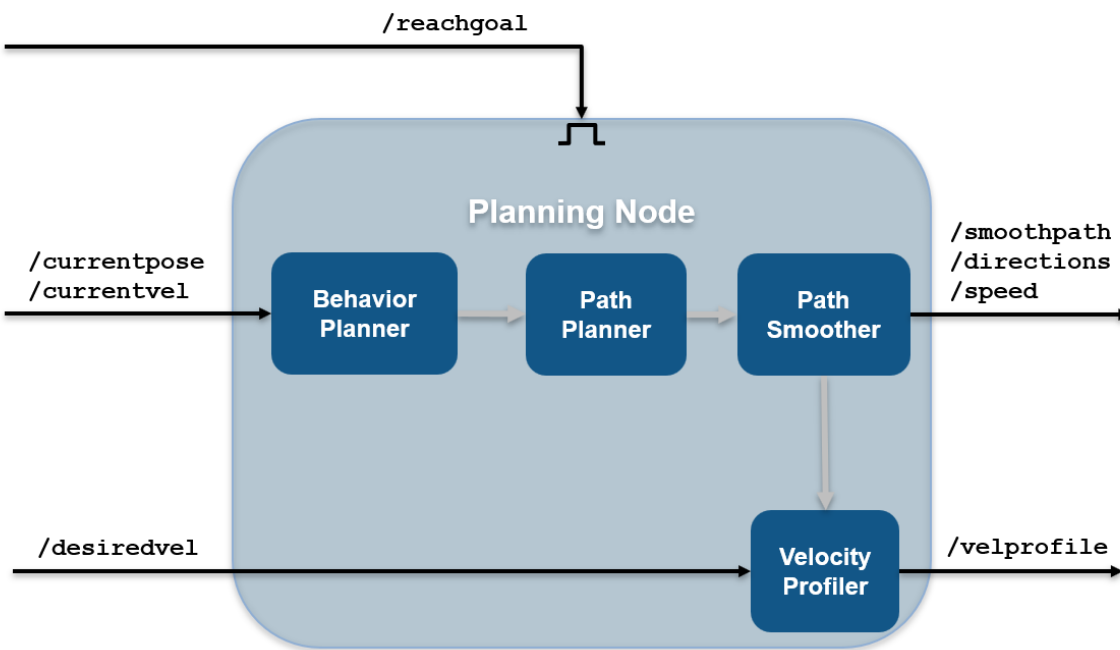
Create the planning node

```
planningNode = ros.Node('planning', masterHost);
```

Create publishers for planning node. Specify the message types for the publisher or subscriber for a topic that is not present on ROS network.

```
planning.PathPub = ros.Publisher(planningNode, '/smoothpath', 'std_msgs/Float64MultiArray');
planning.VelPub = ros.Publisher(planningNode, '/velprofile', 'std_msgs/Float64MultiArray');
planning.DirPub = ros.Publisher(planningNode, '/directions', 'std_msgs/Float64MultiArray');
```

```
planning.SpeedPub = ros.Publisher(planningNode,'/speed','std_msgs/Float64MultiArray');
planning.NxtPub = ros.Publisher(planningNode, '/nextgoal', 'geometry_msgs/Pose2D');
```

Create the subscribers for the planner, `planningNode`.

```
planning.CurVelSub = ros.Subscriber(planningNode, '/currentvel', 'std_msgs/Float64');
planning.CurPoseSub = ros.Subscriber(planningNode, '/currentpose', 'geometry_msgs/Pose2D');
planning.DesrVelSub = ros.Subscriber(planningNode, '/desiredvel', 'std_msgs/Float64');
```

Create subscriber, `GoalReachSub`, to listen to the `/reachgoal` topic of planning node and specify the callback action.

```
GoalReachSub = ros.Subscriber(planningNode, '/reachgoal', 'std_msgs/Bool');
GoalReachSub.NewMessageFcn = @(~,msg)exampleHelperROSValetPlannerCallback(msg, planning, valet);
```

Control

The Control node is responsible for longitudinal and lateral controllers. This node publishes these topics:

- `/steeringangle`
- `/accelcmd`
- `/decelcmd`
- `/vehdir`
- `/reachgoal`

The node subscribes to these topics:

- `/smoothpath`
- `/directions`
- `/speed`
- `/currentpose`
- `/currentvel`
- `/nextgoal`
- `/velprofile`

On receiving a `/velprofile` message, the node runs the `exampleHelperROS2ValetControlCallback` callback, which sends control messages to the vehicle

/velprofile

Control Node

Lateral Controller

/steeringangle

/directions
/currentpose
/smoothpath

Path Analyzer

/reachgoal
/vehdir

/currentvel

Longitudinal Controller

/accelcmd
/decelcmd
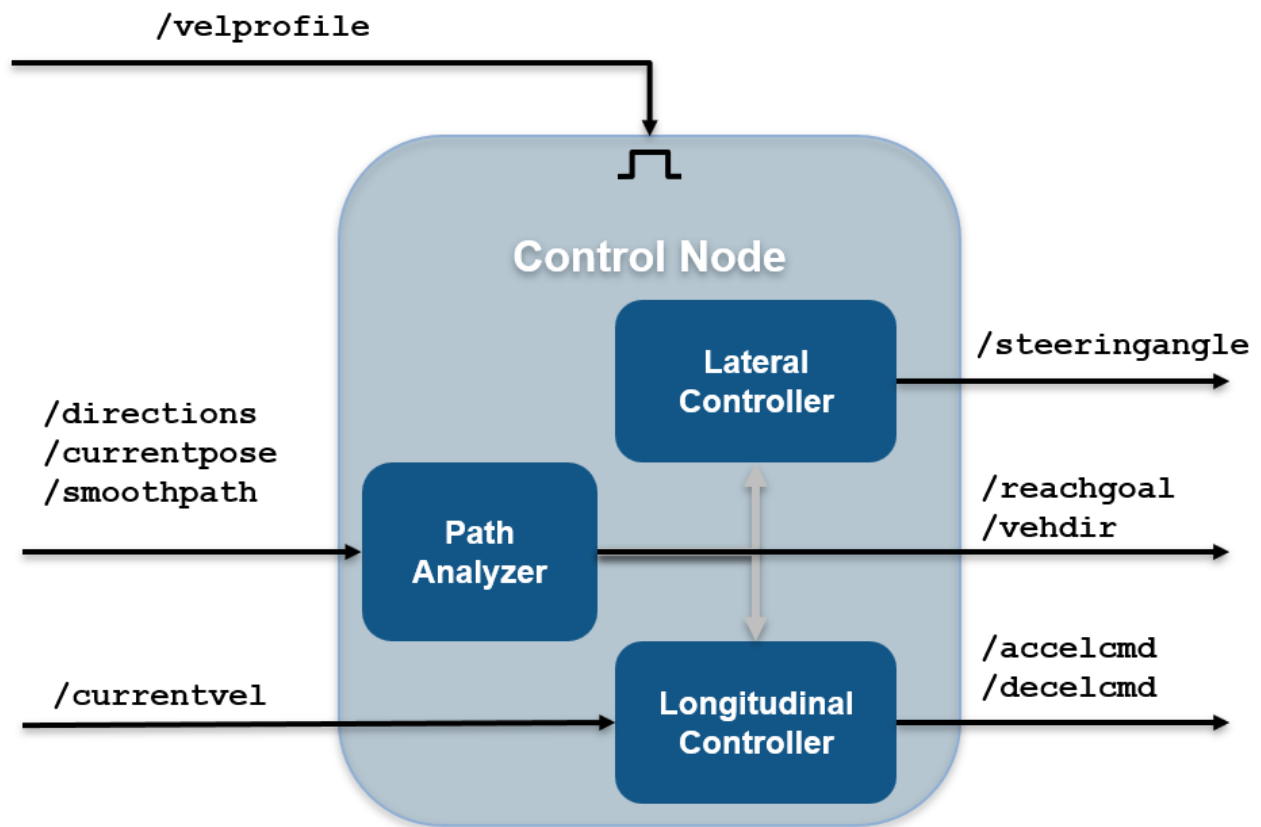
Create the controller, `controlNode`, and setup the publishers and subscribers in the node.

```
controlNode = ros.Node('control', masterHost);

% Publishers for controlNode
control.SteeringPub = ros.Publisher(controlNode, '/steeringangle', 'std_msgs/Float64');
control.AccelPub = ros.Publisher(controlNode, '/accelcmd', 'std_msgs/Float64');
control.DecelPub = ros.Publisher(controlNode, '/decelcmd', 'std_msgs/Float64');
control.VehDirPub = ros.Publisher(controlNode, '/vehdir', 'std_msgs/Float64');
control.VehGoalReachPub = ros.Publisher(controlNode, '/reachgoal');

% Subscribers for controlNode
control.PathSub = ros.Subscriber(controlNode, '/smoothpath');
control.DirSub = ros.Subscriber(controlNode, '/directions');
control.SpeedSub = ros.Subscriber(controlNode, '/speed');
control.CurPoseSub = ros.Subscriber(controlNode, '/currentpose');
control.CurVelSub = ros.Subscriber(controlNode, '/currentvel');
control.NextGoalSub = ros.Subscriber(controlNode, '/nextgoal');

% Create subscriber for /velprofile for control node and provide the callback function.
```

```
VelProfSub = ros.Subscriber(controlNode, '/velprofile');
VelProfSub.NewMessageFcn = @(~,msg)exampleHelperROSValetControlCallback(msg, control, valet);
```

Vehicle

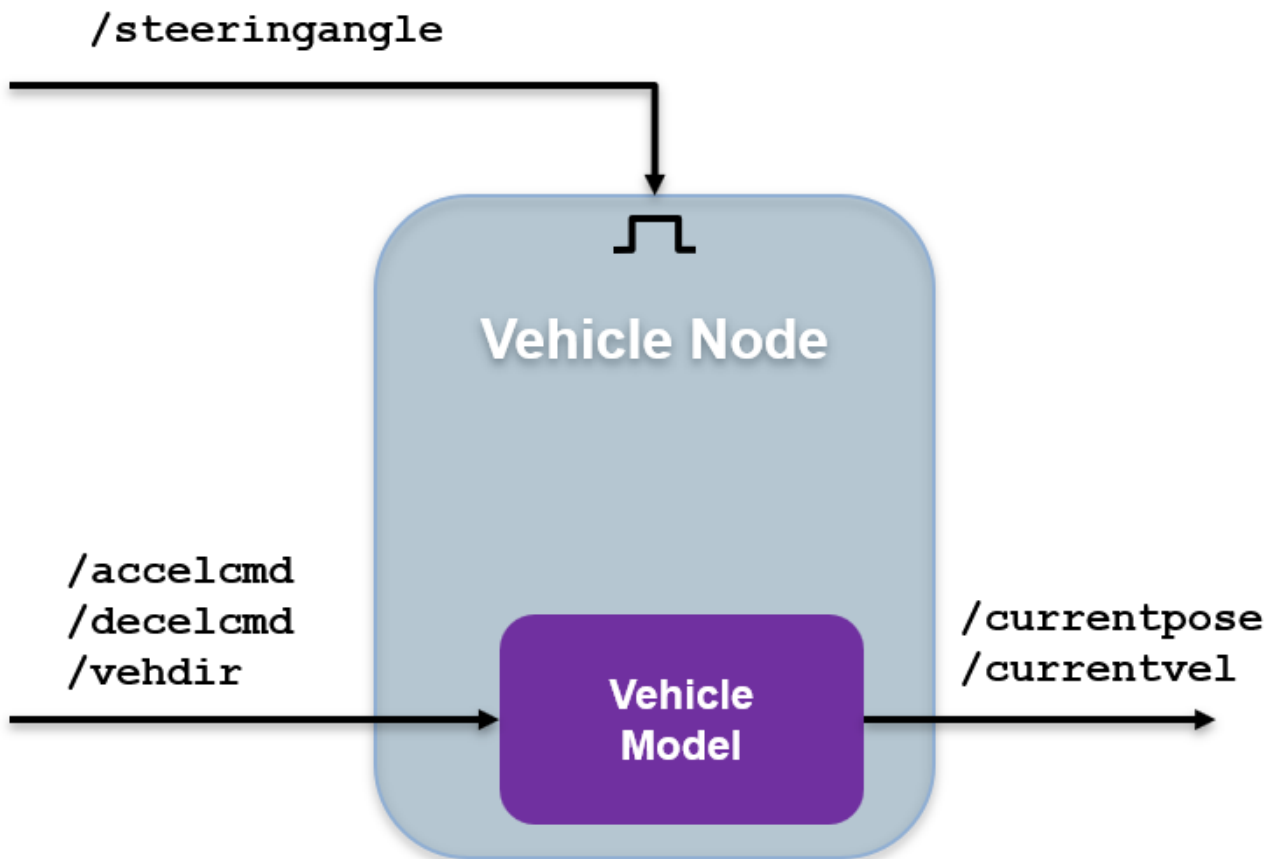The Vehicle node is responsible for simulating the vehicle model. This node publishes these topics:

- /currentvel
- /currentpose

The node subscribes to these topics:

- /accelcmd
- /decelcmd
- /vehdir
- /steeringangle

On receiving a /steeringangle message, the vehicle simulator is run in the callback function, exampleHelperROSValetVehicleCallback.

```
% Create vehicle node.
vehicleNode = ros.Node('vehicle', masterHost);

% Create publishers for vehicle node.
vehicle.CurVelPub = ros.Publisher(vehicleNode, '/currentvel');
vehicle.CurPosePub = ros.Publisher(vehicleNode, '/currentpose');

% Create subscribers for vehicle node.
vehicle.AccelSub = ros.Subscriber(vehicleNode, '/accelcmd');
vehicle.DecelSub = ros.Subscriber(vehicleNode, '/decelcmd');
vehicle.DirSub = ros.Subscriber(vehicleNode, '/vehdir');

% Create subscriber for |/steeringangle|, which runs the vehicle simulator
% callback.
SteeringSub = ros.Subscriber(vehicleNode, '/steeringangle', ...
    @(~,msg)exampleHelperROSValetVehicleCallback(msg, vehicle, valet));
```

Initialize Simulation

To initialize the simulation, send the first velocity message and current pose message. This message causes the planner to start the planning loop.

```
curVelMsg = getROSMessage(vehicle.CurVelPub.MessageType);
curVelMsg.Data = valet.vehicleSim.getVehicleVelocity;
send(vehicle.CurVelPub, curVelMsg);

curPoseMsg = getROSMessage(vehicle.CurPosePub.MessageType);
curPoseMsg.X = valet.currentPose(1);
curPoseMsg.Y = valet.currentPose(2);
curPoseMsg.Theta = valet.currentPose(3);
send(vehicle.CurPosePub, curPoseMsg);

reachMsg = getROSMessage(control.VehGoalReachPub.MessageType);
reachMsg.Data = true;
send(control.VehGoalReachPub, reachMsg);
```
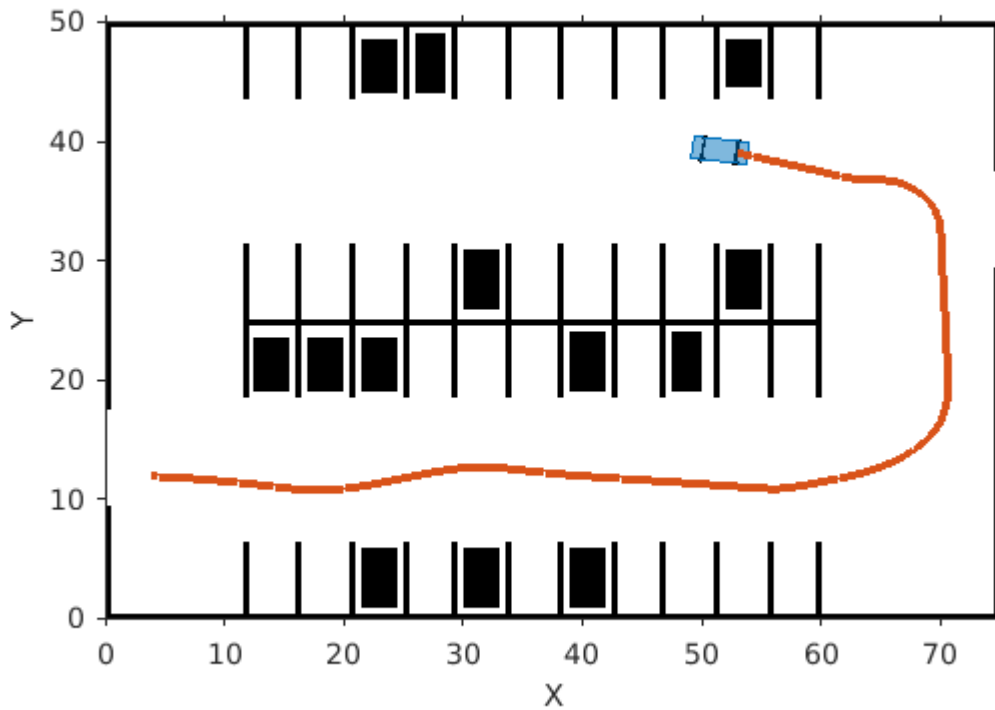
Main Loop

The main loop waits for the `behavioralPlanner` to say the vehicle reached the prepark position.

```
while ~reachedDestination(valet.behavioralPlanner)
    pause(1);
end

% Show the vehicle simulation figure.
showFigure(valet.vehicleSim);
```

Park Maneuver

The parking maneuver callbacks are slightly different from the normal driving manueuver. Replace the callbacks for the /velprofile and /reachgoal subscribers.

```
VelProfSub.NewMessageFcn = @(~,msg)exampleHelperROSValetParkControlCallback(msg, control, valet)
GoalReachSub.NewMessageFcn = @(~,msg)exampleHelperROSValetParkManeuver(msg, planning, valet);

pause(1);
reachMsg = getROSMessage(control.VehGoalReachPub.MessageType);
reachMsg.Data = false;
send(control.VehGoalReachPub, reachMsg);

% Receive a message from the |/reachgoal| topic using the subcriber. This
% waits until a new message is received. Display the figure. The vehicle
% has completed the full automated valet manuever.
receive(GoalReachSub);

exampleHelperROSValetCloseFigures;
snapnow;
```

Delete the simulator and shutdown all the nodes by clearing publishers, subscribers and node handles.

```
delete(valet.vehicleSim);

% Clear variables that were created above.
clear('valet');

GoalReachSub.NewMessageFcn = [];
VelProfSub.NewMessageFcn = [];

clear('planning', 'planningNode', 'GoalReachSub');
clear('control', 'controlNode', 'VelProfSub');
clear('vehicle', 'vehicleNode', 'SteeringSub');
clear('curPoseMsg', 'curVelMsg', 'reachMsg');
clear('masterHost');

% Shutdown the ROS network.
rosshutdown;
```

```
Shutting down global node /matlab_global_node_49911 with NodeURI http://sbd508773glnxa64:42335/
Shutting down ROS master on http://192.168.0.10:60903.
.....
```

Automated Parking Valet with ROS in Simulink

Distribute an automated parking valet application among various nodes in a ROS network in Simulink®. This example extends the "Automated Parking Valet" (Automated Driving Toolbox) example in the Automated Driving Toolbox™. Using the Simulink model in the Automated Parking Valet in Simulink example, tune the planner, controller and vehicle dynamics parameters before partitioning the model into ROS nodes.

Prerequisite: "Automated Parking Valet" (Automated Driving Toolbox), "Generate a Standalone ROS Node from Simulink®" on page 1-92

Introduction

A typical autonomous vehicle application has the following workflow.
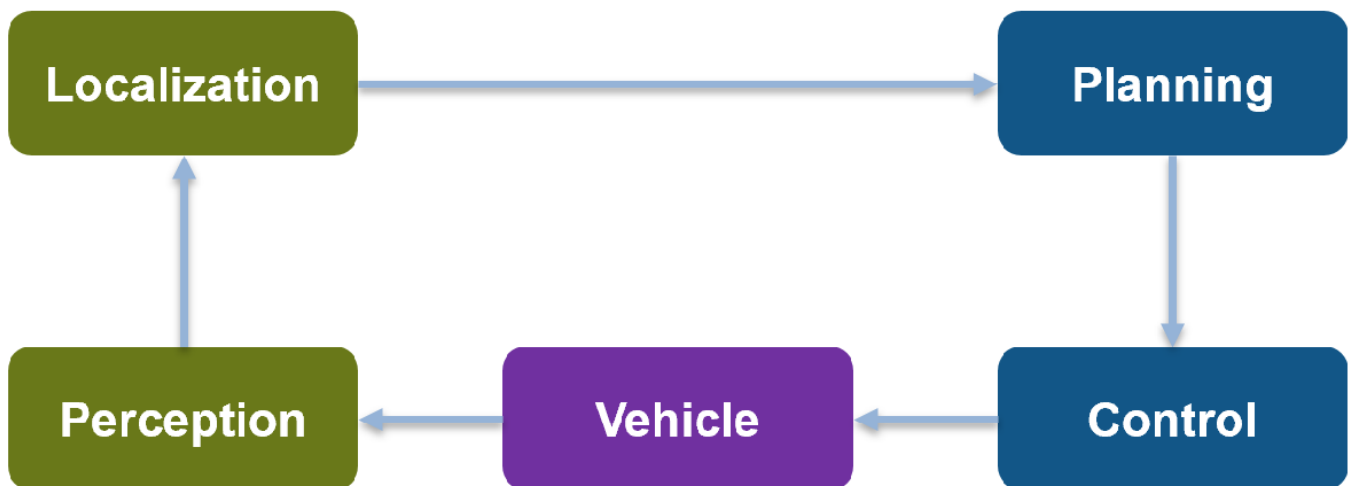
This example concentrates on simulating the *Planning*, *Control* and the *Vehicle* components. For *Localization*, this example uses pre-recorded map localization data. The *Planning* component are further divided into *Behavior planner* and *Path Planner* components. This results in a ROS network comprised of four ROS nodes: `Behavioral Planner`, `Path Planner`, `Controller` and `Vehicle Sim`. The following figure shows the relationships between each ROS node in the network and the topics used in each.
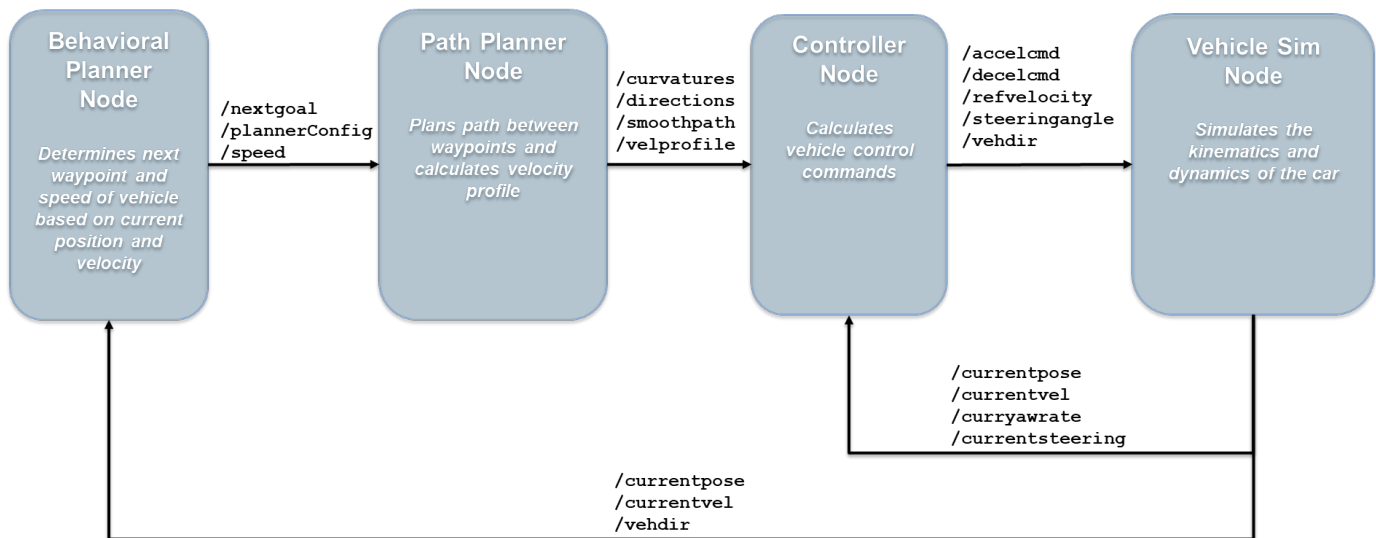
Explore the Simulink ROS nodes and connectivity

Observe the division of the components into four separate Simulink models. Each Simulink model represents a ROS node that sends and receives messages on different topics.

Vehicle Sim Node
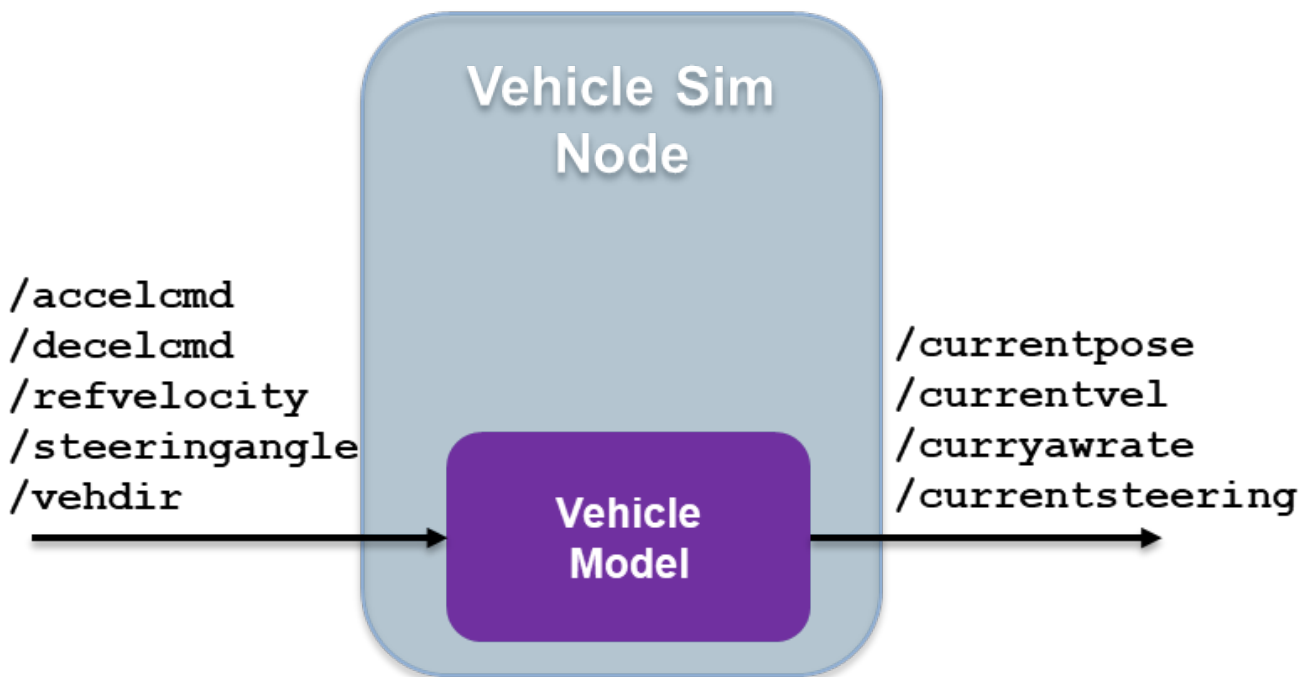
1. Open the vehicle model.

```
open_system('ROSValetVehicleExample');
```

2. The Subscribe subsystem contains the ROS Subscribe blocks that read input data from the Controller on page 2-0   node.

3. The `Vehicle model` subsystem contains a Bicycle Model (Automated Driving Toolbox) block, `Vehicle Body 3DOF`, to simulate the vehicle controller effects and sends the vehicle information over the ROS network through ROS Publish blocks in the `Publish` subsystem.

Behavioral Planner Node

/currentpose
/currentvel

Behavioral Planner Node

/vehdir

Behavior Planner & Goal Checker
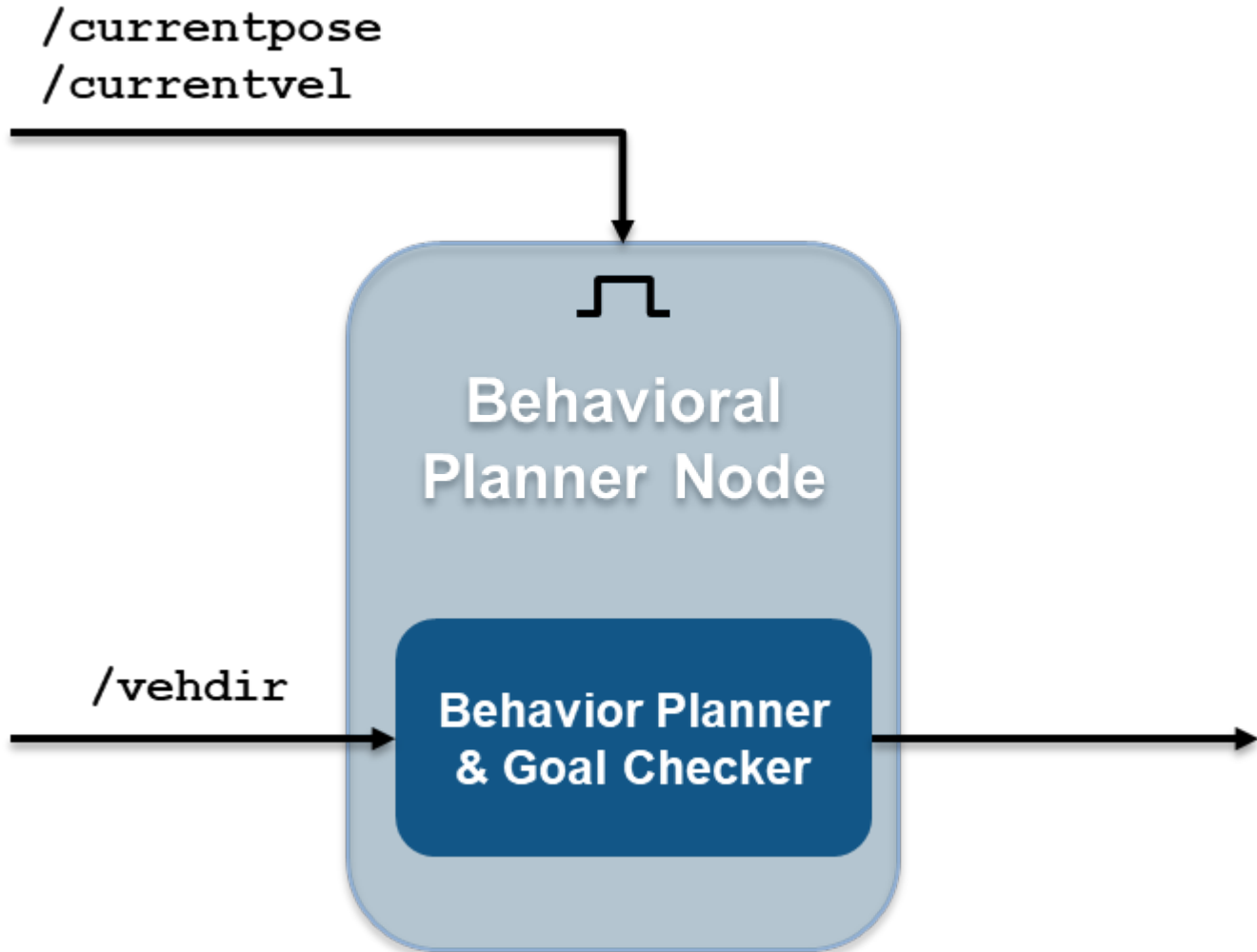
1. Open the behavioral planner model.

```
open_system('ROSValetBehavioralPlannerExample');
```

2. This model reads the current vehicle pose, velocity, and direction from the ROS network, and sends the next goal. It checks if the vehicle has reached the goal pose of the segment using `exampleHelperROSValetGoalChecker`.

3. The `Behavior Planner and Goal Checker` model runs when a new message is available on either `/currentpose` or `/currentvel`.

4. The model sends the status if the vehicle has reached the parking goal using the `/reachgoal` topic, which uses a `std_msgs/Bool` message. All the models stop simulation when this message is `true`.
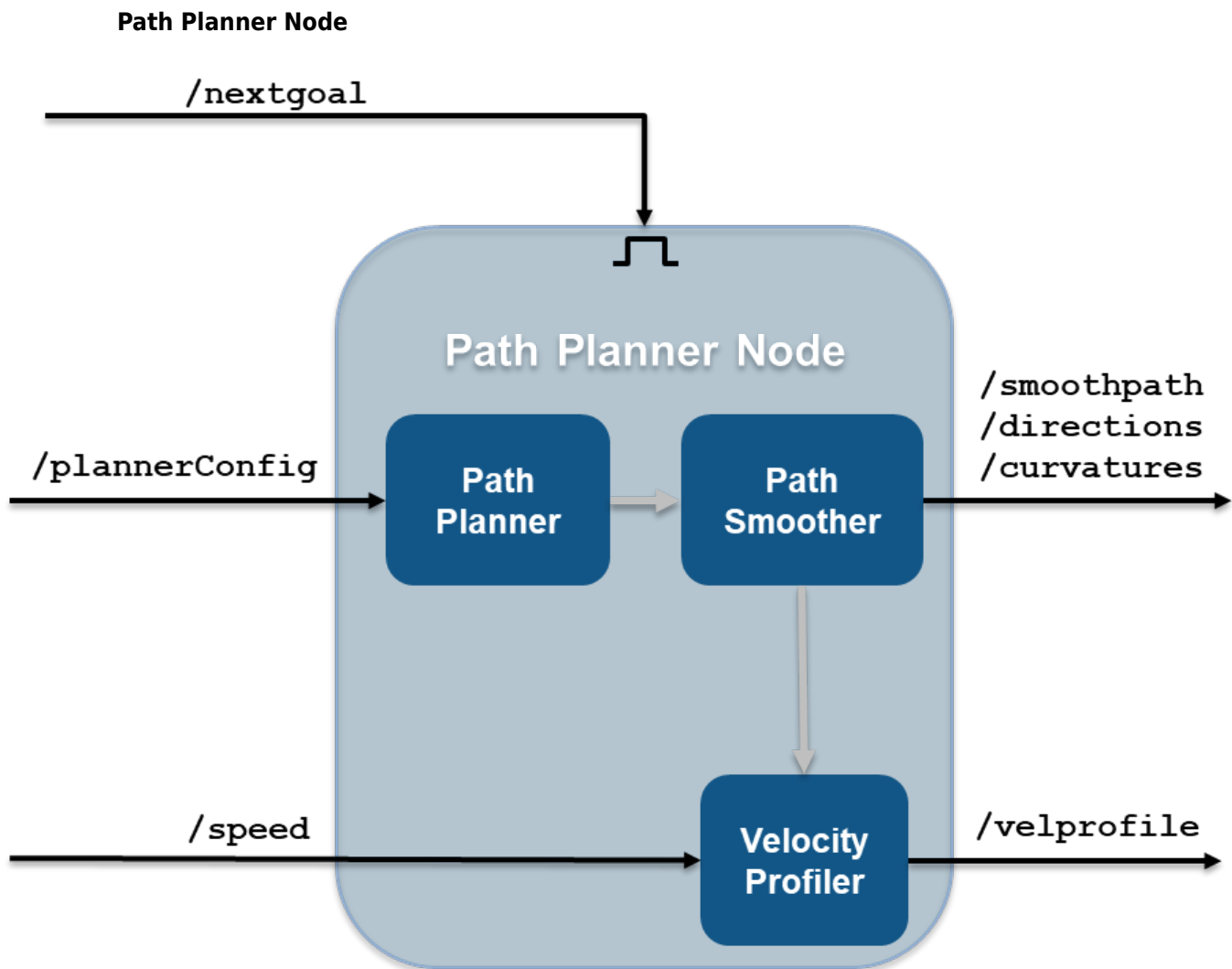
Path Planner Node

1. Open the path planner model.

```
open_system('ROSValetPathPlannerExample');
```

2. This model plans a feasible path through the environment map using a `pathPlannerRRT` (Automated Driving Toolbox) object, which implements the *optimal rapidly exploring random tree* (RRT*) algorithm and sends the plan to the controller over the ROS network.

3. The `Path Planner` subsystem runs when a new message is available on `/plannerConfig` or `/nextgoal` topics.
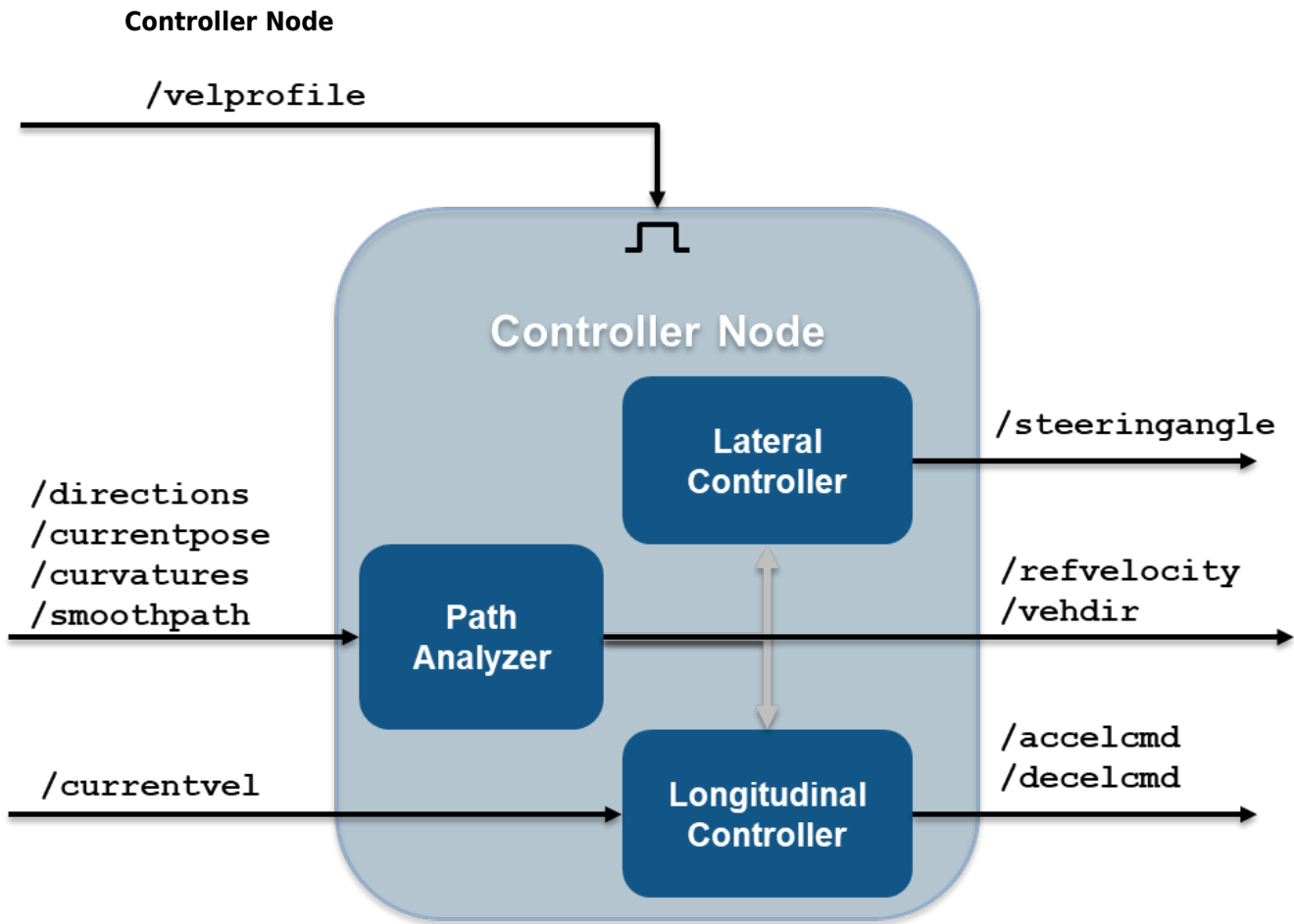
Controller Node

1. Open the vehicle controller model.

```
open_system('ROSValetControllerExample');
```

2. This model calculates and sends the steering and velocity commands over the ROS network.

3. The Controller subsystem runs when a new message is available on the `/velprofile` topic.

Simulate the ROS nodes to verify partitioning

Verify that the behavior of the model remains the same after partitioning the system into four ROS nodes.

1. Run `rosinit` in MATLAB® Command Window to initialize the global node and ROS master

```
rosinit
```

```
Launching ROS Core...
...................................Done in 7.169 seconds.
Initializing ROS master on http://192.168.203.1:51612.
Initializing global node /matlab_global_node_70835 with NodeURI http://ah-sradford:58388/
```

2. Load the pre-recorded localization map data in MATLAB base workspace using the `exampleHelperROSValetLoadLocalizationData` helper function.

```
exampleHelperROSValetLoadLocalizationData;
```

3. Open the simulation model.

```
open_system('ROSValetSimulationExample.slx');
```

In the left parking selection area, you can also select a spot. The default parking spot is the sixth spot at the top row.

4. In the SIMULATION tab, click **Run** from **SIMULATE** section or run `sim('ROSValetSimulationExample.slx')` in MATLAB Command Window. A figure opens and shows how the vehicle tracks the reference path. The blue line represents the reference path while the red line is the actual path traveled by the vehicle. Simulation for all the models stop when the vehicle reaches the final parking spot.

```
sim('ROSValetSimulationExample.slx');
```
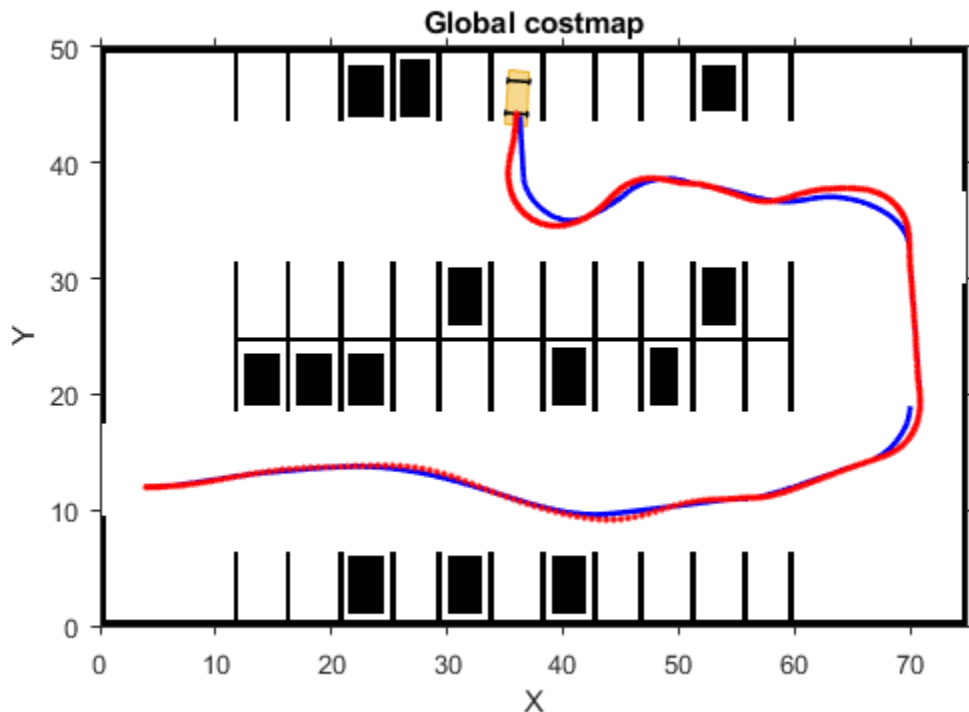

Simulation Results

The `Visualization` subsystem in the vehicle model generates the results for this example.

```
open_system('ROSValetVehicleExample/Vehicle model/Visualization');
```

The `visualizePath` block is responsible for creating and updating the plot of the vehicle paths shown previously. The vehicle speed and steering commands are displayed in a scope.

```
open_system("ROSValetVehicleExample/Vehicle model/Visualization/Commands")
```
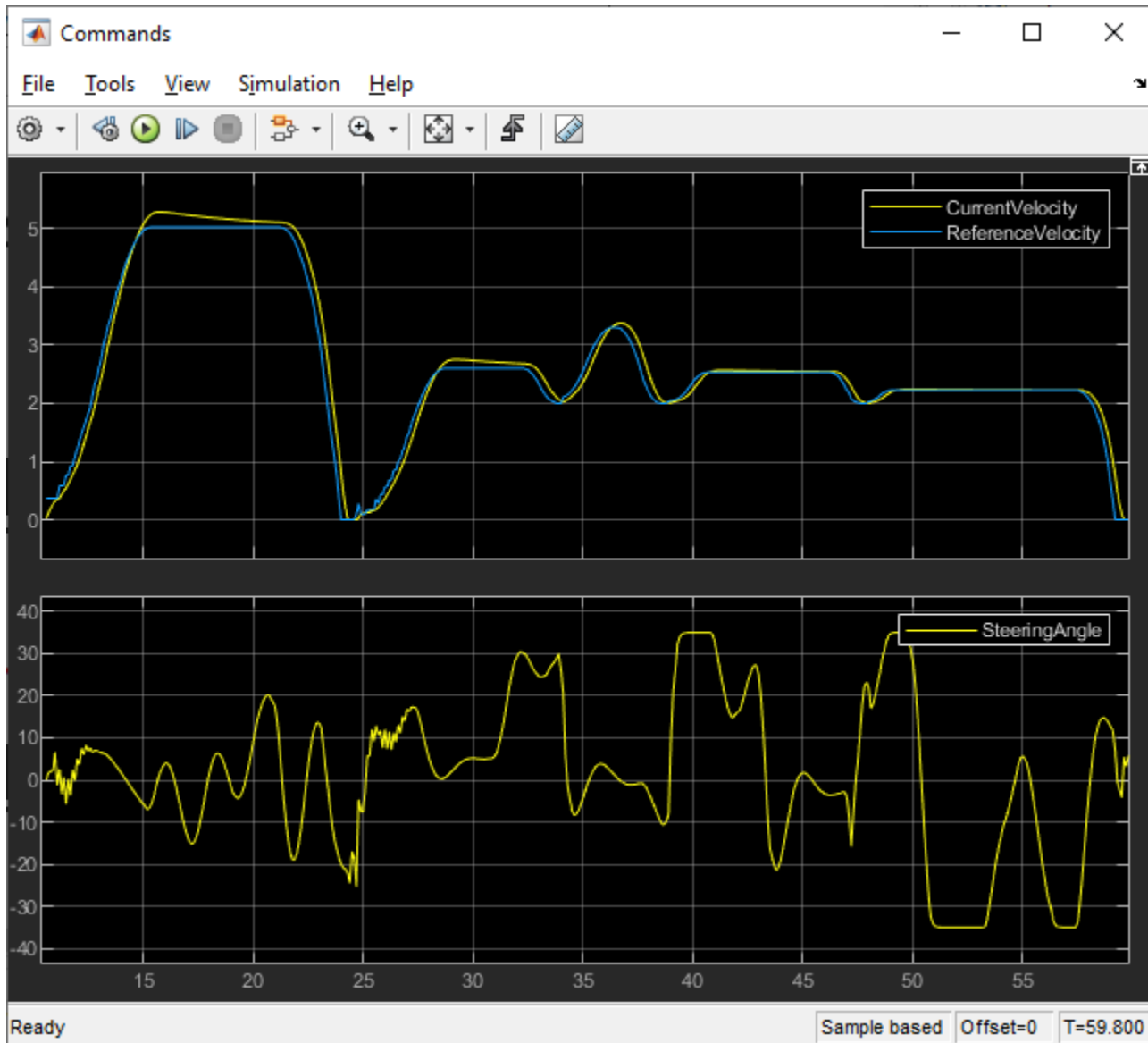
Deploy ROS Nodes

Generate ROS applications for `Behavioral planner`, `Path planner`, `Controller` nodes, and simulate the `Vehicle` node in MATLAB and compare the results with simulation. For more informaiton on generating ROS notes, see "Generate a Standalone ROS Node from Simulink®" on page 1-92.

1. Deploy the `Behavioral planner`, `Path planner` and `Controller` ROS nodes.

2. Open the vehicle model.

```
open_system('ROSValetVehicleExample');
```

3. From the **Simulation** tab, click **Run** to start the simulation.

4. Observe the vehicle movement on the plot and compare the results from simulation run.

5. Shut down the ROS network using `rosshutdown`.

```
rosshutdown
```

```
Shutting down global node /matlab_global_node_70835 with NodeURI http://ah-sradford:58388/
Shutting down ROS master on http://192.168.203.1:51612.
```

Automated Parking Valet with ROS 2 in MATLAB

This example shows how you can distribute "Automated Parking Valet" (Automated Driving Toolbox) application among various nodes in a ROS 2 network.

Overview

This example is an extension of the "Automated Parking Valet" (Automated Driving Toolbox) example in Automated Driving Toolbox™. A typical autonmous application has the following components.

For simplicity, this example concentrates on Planning, Control, and a simplified Vehicle Model. The example uses prerecorded data to substitute localization information.

This application demonstrates a typical split of various functions into ROS nodes. The following picture shows how the above example is split into various nodes. Each node: Planning, Control and Vehicle is a ROS node implementing the functionalities shown as below. The interconnections between the nodes show the topics used on each interconnection of the nodes.

Setup

First, load a route plan and a given costmap used by the behavior planner and path analyzer.
Behavior Planner, Path Planner, Path Analyzer, Lateral and Lognitudinal Controllers are implemented
by helper classes, which are setup with this example helper function call.

```
exampleHelperROSValetSetupGlobals;

% The initialized globals are organized as fields in the global structure
% |valet|.
disp(valet)

            mapLayers: [1×1 struct]
              costmap: [1×1 vehicleCostmap]
          vehicleDims: [1×1 vehicleDimensions]
     maxSteeringAngle: 35
                 data: [1×1 struct]
             routePlan: [4×3 table]
          currentPose: [4 12 0]
           vehicleSim: [1×1 ExampleHelperROSValetVehicleSimulator]
    behavioralPlanner: [1×1 ExampleHelperROSValetBehavioralPlanner]
        motionPlanner: [1×1 pathPlannerRRT]
             goalPose: [56 11 0]
              refPath: [1×1 driving.Path]
      transitionPoses: [14×3 double]
           directions: [522×1 double]
           currentVel: 0
       approxSeparation: 0.1000
        numSmoothPoses: 522
             maxSpeed: 5
           startSpeed: 0
             endSpeed: 0
             refPoses: [522×3 double]
           cumLengths: [522×1 double]
           curvatures: [522×1 double]
         refVelocities: [522×1 double]
           sampleTime: 0.1000
        lonController: [1×1 ExampleHelperROSValetLongitudinalController]
          controlRate: [1×1 ExampleHelperROSValetFixedRate]
         pathAnalyzer: [1×1 ExampleHelperROSValetPathAnalyzer]
             parkPose: [36 44 90]
```

Use nodes to split the functions in the application. This example uses three nodes: `planningNode`, `controlNode`, and `vehicleNode`.

Planning

The Planning node calculates each path segment based on the current vehicle position. This node is responsible for generating the smooth path and publishes the path to the network.

This node publishes these topics:

- `/smoothpath`
- `/velprofile`
- `/directions`
- `/speed`
- `/nextgoal`

The node subscribes to these topics:

- `/currentvel`
- `/currentpose`
- `/desiredvel`
- `/reachgoal`

On receiving a `/reachgoal` message, the node runs the
`exampleHelperROS2ValetPlannerCallback` callback, which plans the next segment.
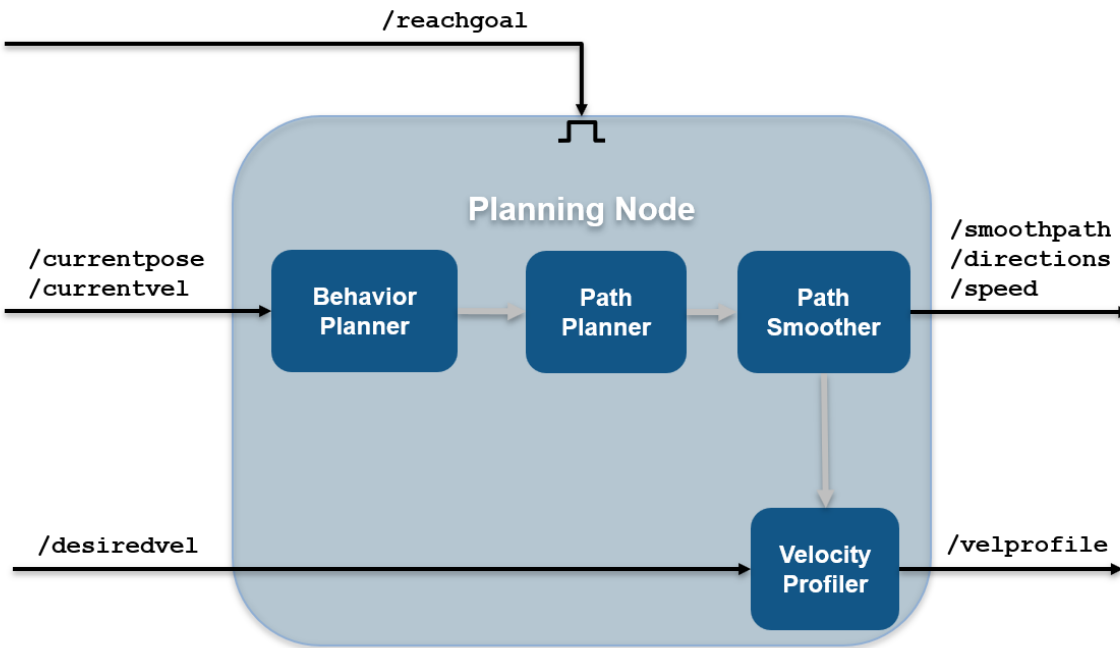

```matlab
% Create planning node.
planningNode = ros2node('planning');

% Create publishers for planning node. Specify the message types the first
% time you create a publisher or subscriber for a topic.
planning.PathPub = ros2publisher(planningNode, '/smoothpath', 'std_msgs/Float64MultiArray');
planning.VelPub = ros2publisher(planningNode, '/velprofile', 'std_msgs/Float64MultiArray');
planning.DirPub = ros2publisher(planningNode, '/directions', 'std_msgs/Float64MultiArray');
planning.SpeedPub = ros2publisher(planningNode,'/speed','std_msgs/Float64MultiArray');
planning.NxtPub = ros2publisher(planningNode, '/nextgoal', 'geometry_msgs/Pose2D');

% Create subscribers for planning node.
planning.CurVelSub = ros2subscriber(planningNode, '/currentvel', 'std_msgs/Float64');
planning.CurPoseSub = ros2subscriber(planningNode, '/currentpose', 'geometry_msgs/Pose2D');
planning.DesrVelSub = ros2subscriber(planningNode, '/desiredvel', 'std_msgs/Float64');

% Create GoalReachSub part of planning node and provide the callback.
GoalReachSub = ros2subscriber(planningNode, '/reachgoal', 'std_msgs/Bool');
GoalReachSub.NewMessageFcn = @(msg)exampleHelperROS2ValetPlannerCallback(msg, planning, valet);
```

Control

The Control node is responsible for longitudinal and lateral controllers. This node publishes these topics:

*   `/steeringangle`
*   `/accelcmd`
*   `/decelcmd`
*   `/vehdir`
*   `/reachgoal`

The node subscribes to these topics:

*   `/smoothpath`
*   `/directions`
*   `/speed`
*   `/currentpose`
*   `/currentvel`
*   `/nextgoal`
*   `/velprofile`

On receiving a `/velprofile` message, the node runs the
`exampleHelperROS2ValetControlCallback` callback, which sends control messages to the
vehicle
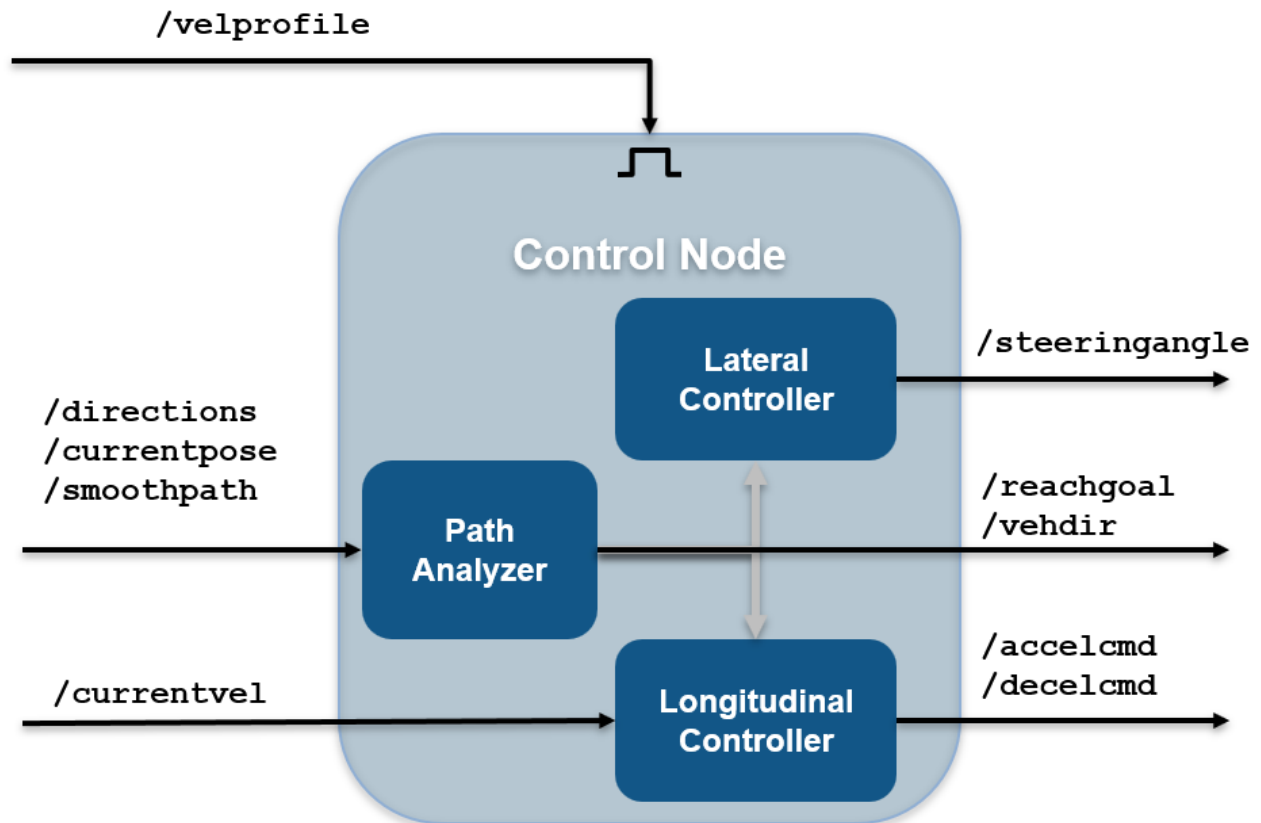

```matlab
% Create control node
controlNode = ros2node('control');

% Create publishers for control node
control.SteeringPub = ros2publisher(controlNode, '/steeringangle', 'std_msgs/Float64');
control.AccelPub = ros2publisher(controlNode, '/accelcmd', 'std_msgs/Float64');
control.DecelPub = ros2publisher(controlNode, '/decelcmd', 'std_msgs/Float64');
control.VehDirPub = ros2publisher(controlNode, '/vehdir', 'std_msgs/Float64');
control.VehGoalReachPub = ros2publisher(controlNode, '/reachgoal');

% Create subscribers for control node. Since all the message types for the
% topics are determined above, we can use the shorter version to create
% subscribers
control.PathSub = ros2subscriber(controlNode, '/smoothpath');
control.DirSub = ros2subscriber(controlNode, '/directions');
control.SpeedSub = ros2subscriber(controlNode, '/speed');
control.CurPoseSub = ros2subscriber(controlNode, '/currentpose');
```

```
control.CurVelSub = ros2subscriber(controlNode, '/currentvel');
control.NextGoalSub = ros2subscriber(controlNode, '/nextgoal');

% Create VelProfSub for control node and provide the callback
VelProfSub = ros2subscriber(controlNode, '/velprofile');
VelProfSub.NewMessageFcn = @(msg)exampleHelperROS2ValetControlCallback(msg, control, valet);
```
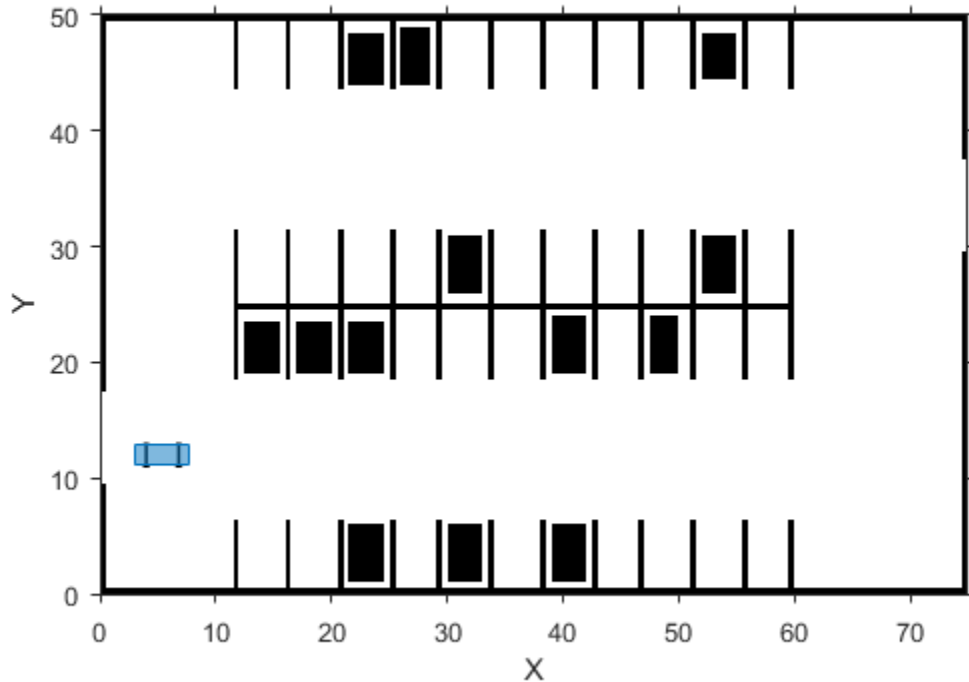

Vehicle

The Vehicle node is responsible for simulating the vehicle model. This node publishes these topics:

- /currentvel
- /currentpose

The node subscribes to these topics:

- /accelcmd
- /decelcmd
- /vehdir
- /steeringangle

```
% On receiving a |/steeringangle| message, the vehicle simulator is run in
% the |exampleHelperROS2ValetVehicleCallback| callback.
%
% <<../exampleHelperROSValetVehicleNode.PNG>>
%
```

```
% Create vehicle node.
vehicleNode = ros2node('vehicle');

% Create publishers for vehicle node.
vehicle.CurVelPub = ros2publisher(vehicleNode, '/currentvel');
vehicle.CurPosePub = ros2publisher(vehicleNode, '/currentpose');

% Create subscribers for vehicle node.
vehicle.AccelSub = ros2subscriber(vehicleNode, '/accelcmd');
vehicle.DecelSub = ros2subscriber(vehicleNode, '/decelcmd');
vehicle.DirSub = ros2subscriber(vehicleNode, '/vehdir');

% Create SteeringSub which runs the vehicle simulator as part of callback.
SteeringSub = ros2subscriber(vehicleNode, '/steeringangle', ...
    @(msg)exampleHelperROS2ValetVehicleCallback(msg, vehicle, valet));
```
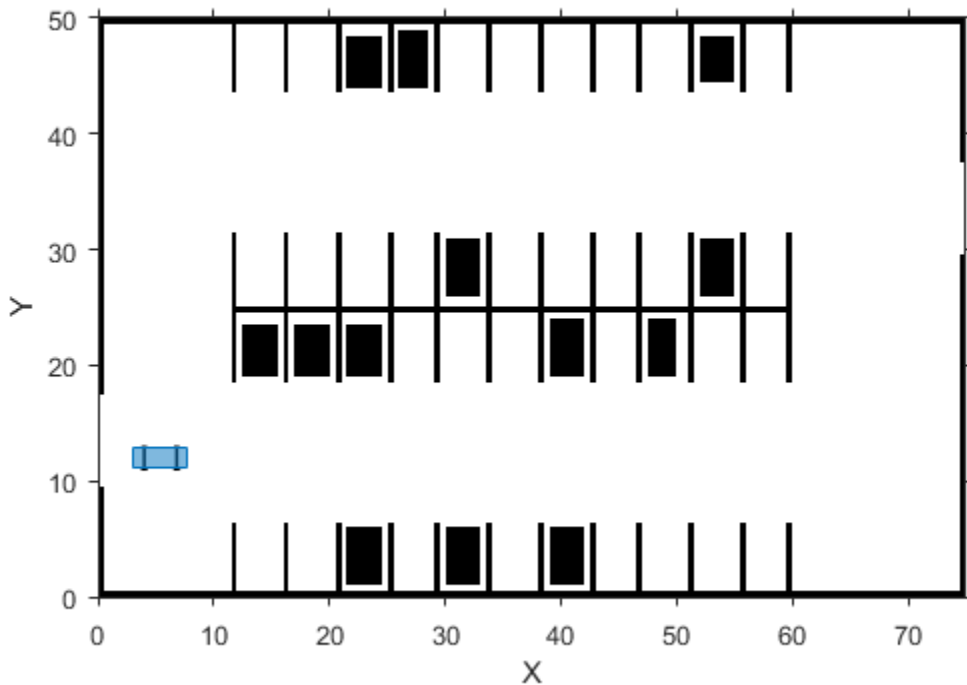

Initialize Simulation

To initialize the simulation, send the first velocity message and current pose message. This message causes the planner to start the planning loop.

```
curVelMsg = ros2message(vehicle.CurVelPub);
curVelMsg.data = valet.vehicleSim.getVehicleVelocity;
send(vehicle.CurVelPub, curVelMsg);

curPoseMsg = ros2message(vehicle.CurPosePub);
curPoseMsg.x = valet.currentPose(1);
```

```
curPoseMsg.y = valet.currentPose(2);
curPoseMsg.theta = valet.currentPose(3);
send(vehicle.CurPosePub, curPoseMsg);

reachMsg = ros2message(control.VehGoalReachPub);
reachMsg.data = true;
send(control.VehGoalReachPub, reachMsg);
```
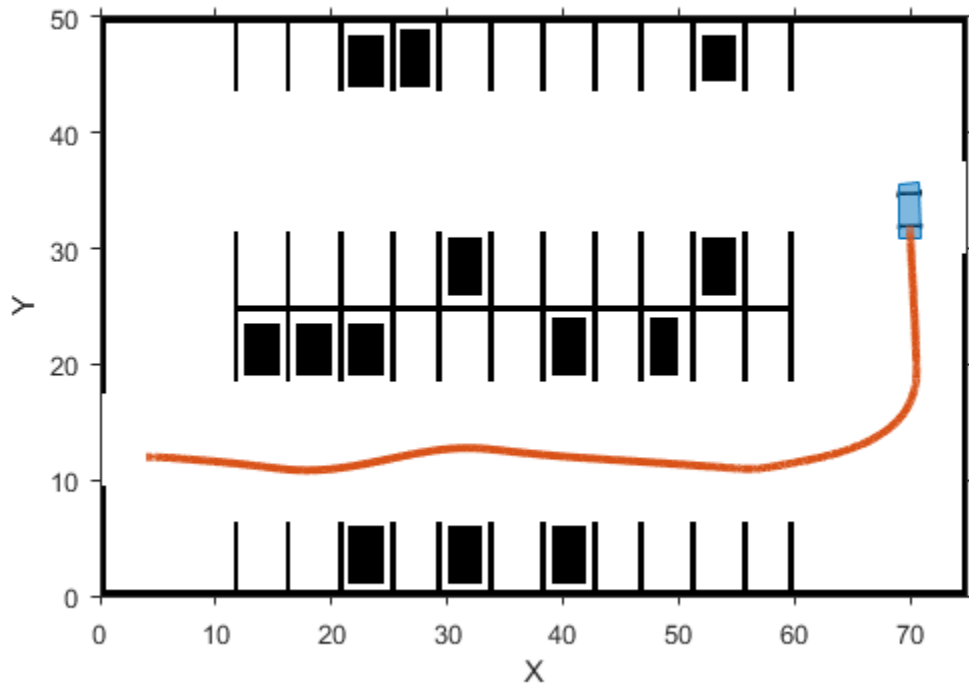

Main Loop

The main loop waits for the behavioralPlanner to say the vehicle reached the prepark position.

```
while ~reachedDestination(valet.behavioralPlanner)
    pause(1);
end

% Show vehicle simulation figure
showFigure(valet.vehicleSim);
```
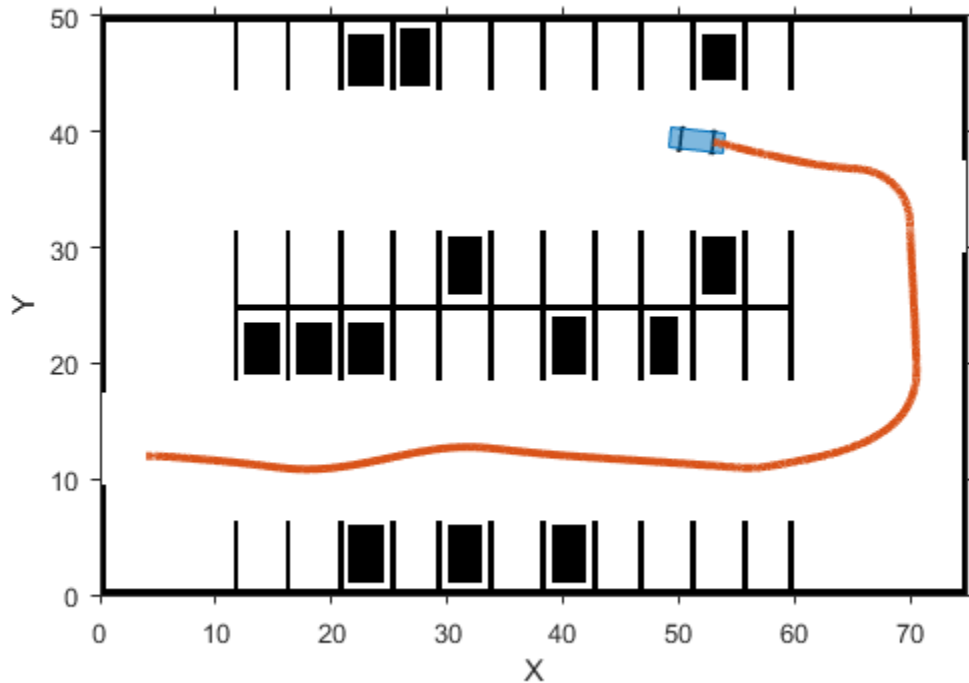
Park Maneuver

The parking maneuver callbacks are slightly different from the normal driving manueuver. Replace the callbacks for the /velprofile and /reachgoal subscribers.

```
VelProfSub.NewMessageFcn = @(msg)exampleHelperROS2ValetParkControlCallback(msg, control, valet);
GoalReachSub.NewMessageFcn = @(msg)exampleHelperROS2ValetParkManeuver(msg, planning, valet);

reachMsg = ros2message(control.VehGoalReachPub);
reachMsg.data = false;
send(control.VehGoalReachPub, reachMsg);

% Receive a message from the |/reachgoal| topic using the subcriber. This
% waits until a new message is received. Display the figure. The vehicle
% has completed the full automated valet manuever.

receive(GoalReachSub);

exampleHelperROSValetCloseFigures;
snapnow;
```

Delete the simulator and shutdown all the nodes by clearing publishers, subscribers and node handles.

```
delete(valet.vehicleSim);

% Clear variables that were created above.
clear('valet');

GoalReachSub.NewMessageFcn = [];
VelProfSub.NewMessageFcn = [];

clear('planning', 'planningNode', 'GoalReachSub');
clear('control', 'controlNode', 'VelProfSub');
clear('vehicle', 'vehicleNode', 'SteeringSub');
clear('curPoseMsg', 'curVelMsg', 'reachMsg');
```

Automated Parking Valet with ROS 2 in Simulink

This example shows how to distribute the Automated Parking Valet application among various nodes in a ROS 2 network in Simulink® and deploy them as standalone ROS 2 nodes. This example extends the "Automated Parking Valet" (Automated Driving Toolbox) example in the Automated Driving Toolbox™. Using the Simulink model in the Automated Parking Valet in Simulink example, tune the planner, controller and vehicle dynamic parameters before partitioning the model into ROS 2 nodes.

Prerequisites: "Automated Parking Valet" (Automated Driving Toolbox), "Generate a Standalone ROS 2 Node from Simulink®" on page 2-57

Introduction

This autonomous vehicle application has the following components.
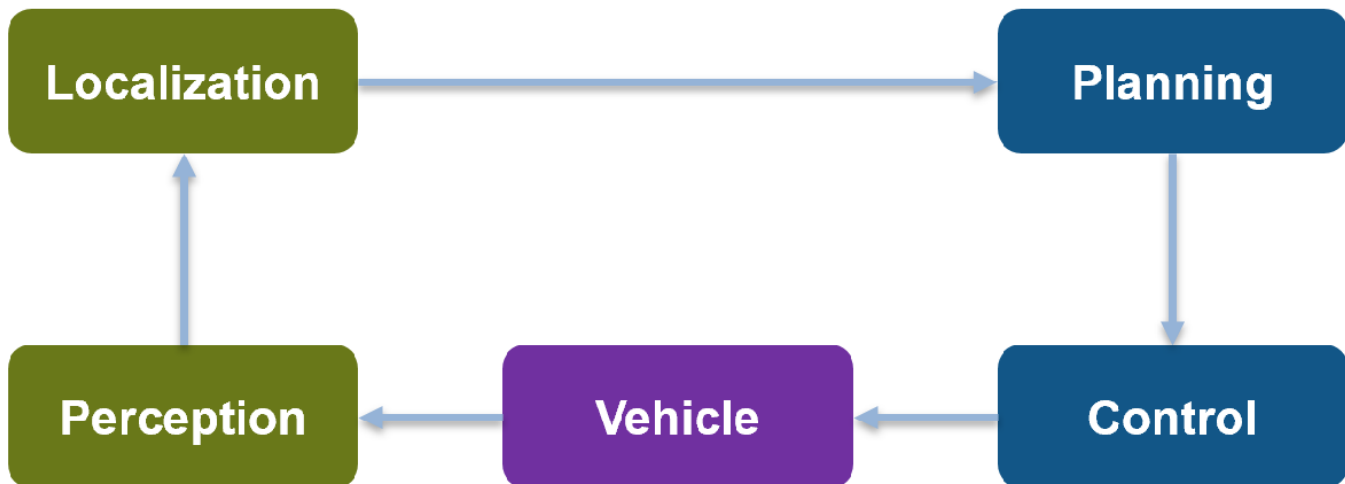
This example concentrates on simulating the *Planning*, *Control* and the *Vehicle* components. For *Localization*, this example uses pre-recorded localization map data. The *Planning* component is further divided into *Behavior planner* and *Path Planner* components. This results in a ROS 2 network comprised of four ROS 2 nodes: `Behavioral Planner`, `Path Planner`, `Controller` and `Vehicle`. The following figure shows the relationships between each ROS 2 node in the network and the topics used in each.
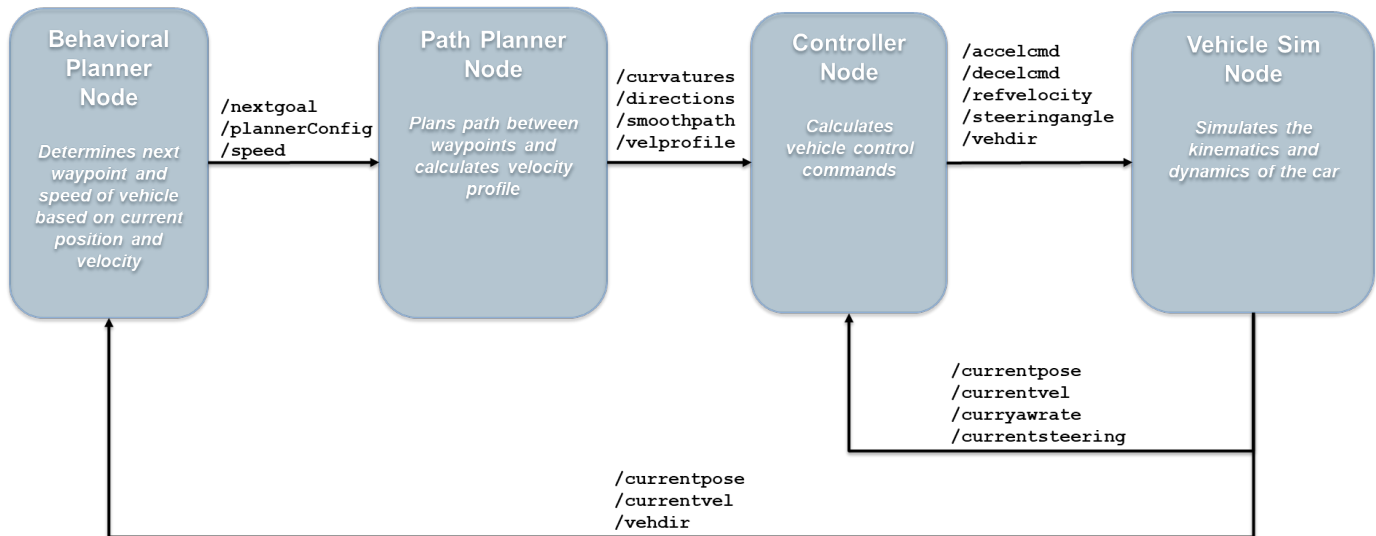
Explore the Simulink ROS 2 nodes and connectivity

Observe the division of the components into four separate Simulink models. Each Simulink model represents a ROS 2 node.

Vehicle Node
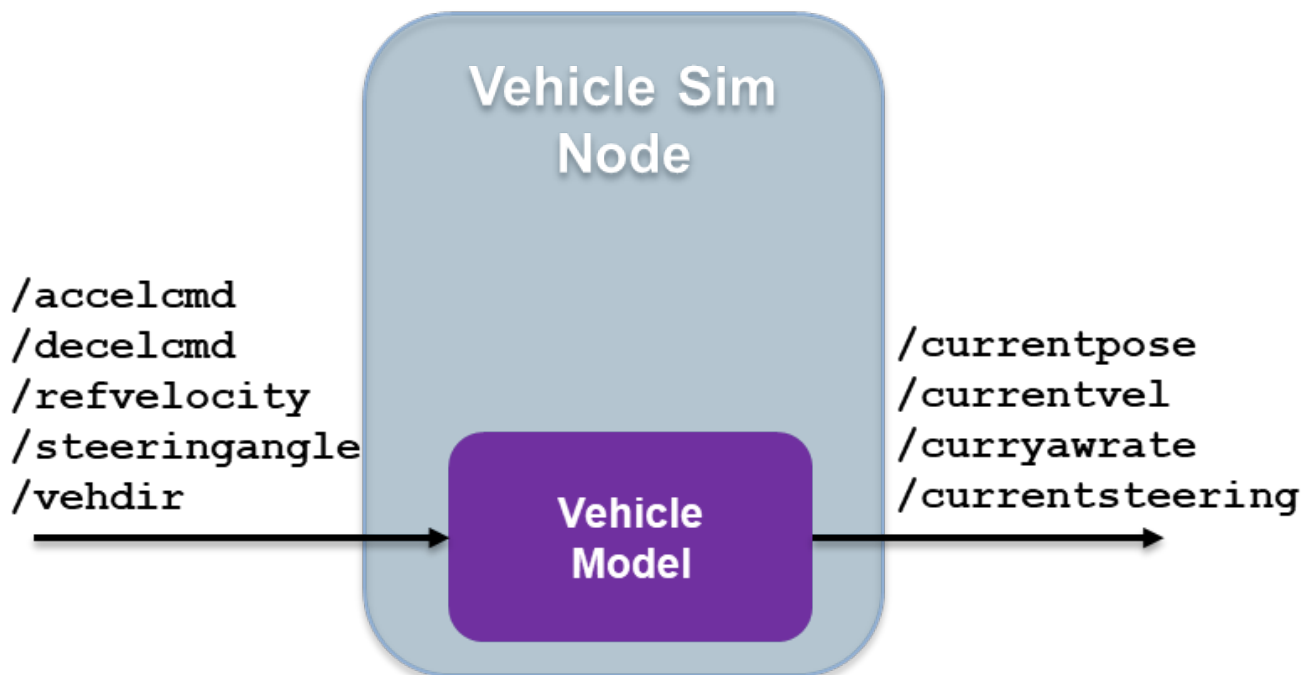
1. Open the vehicle model.

```
open_system('ROS2ValetVehicleExample');
```

2. The Subscribe subsystem contains the ROS 2 Subscribe blocks that read input data from the Controller on page 2-0   node.

3. The `Vehicle model` subsystem contains a Bicycle Model (Automated Driving Toolbox) block, `Vehicle Body 3DOF`, to simulate the vehicle controller effects and sends the vehicle information over the ROS 2 network through ROS 2 Publish blocks in the `Publish` subsystem.

Behavioral Planner Node
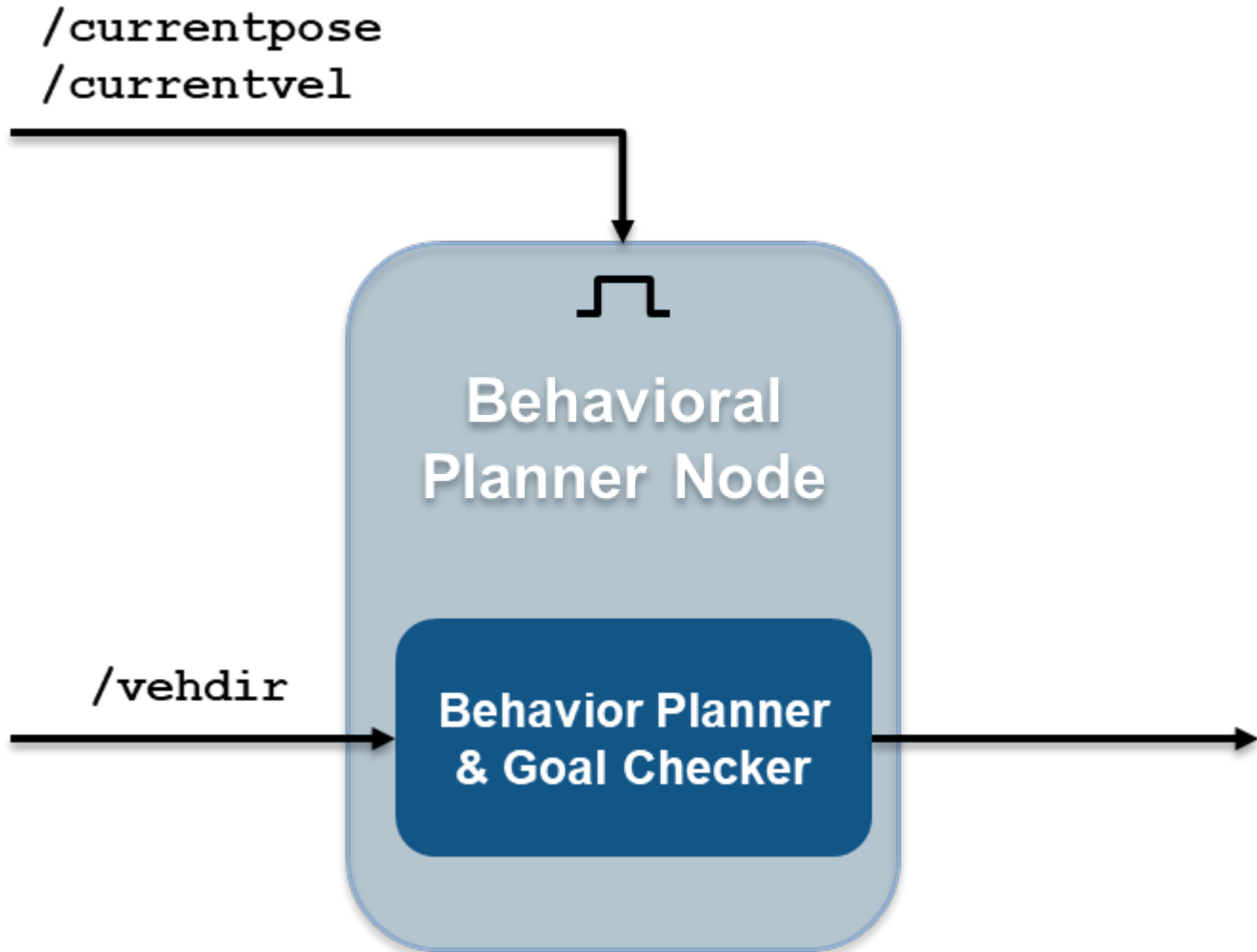
1. Open the behavioral planner model.

```
open_system('ROS2ValetBehavioralPlannerExample');
```

2. This model reads the current vehicle information from ROS 2 network, sends the next goal and checks if the vehicle has reached the final pose of the segment using `rosAutomatedValetHelperGoalChecker`.

3. The `Behavioral Planner and Goal Checker` subsystem runs when a new message is available on either `/currentvel` or `/currentpose`.

4. The model sends the status if the vehicle has reached the final parking goal using the `/reachgoal` topic, which uses a `std_msgs/Bool` message type. All the models stop simulation when this message is `true`.
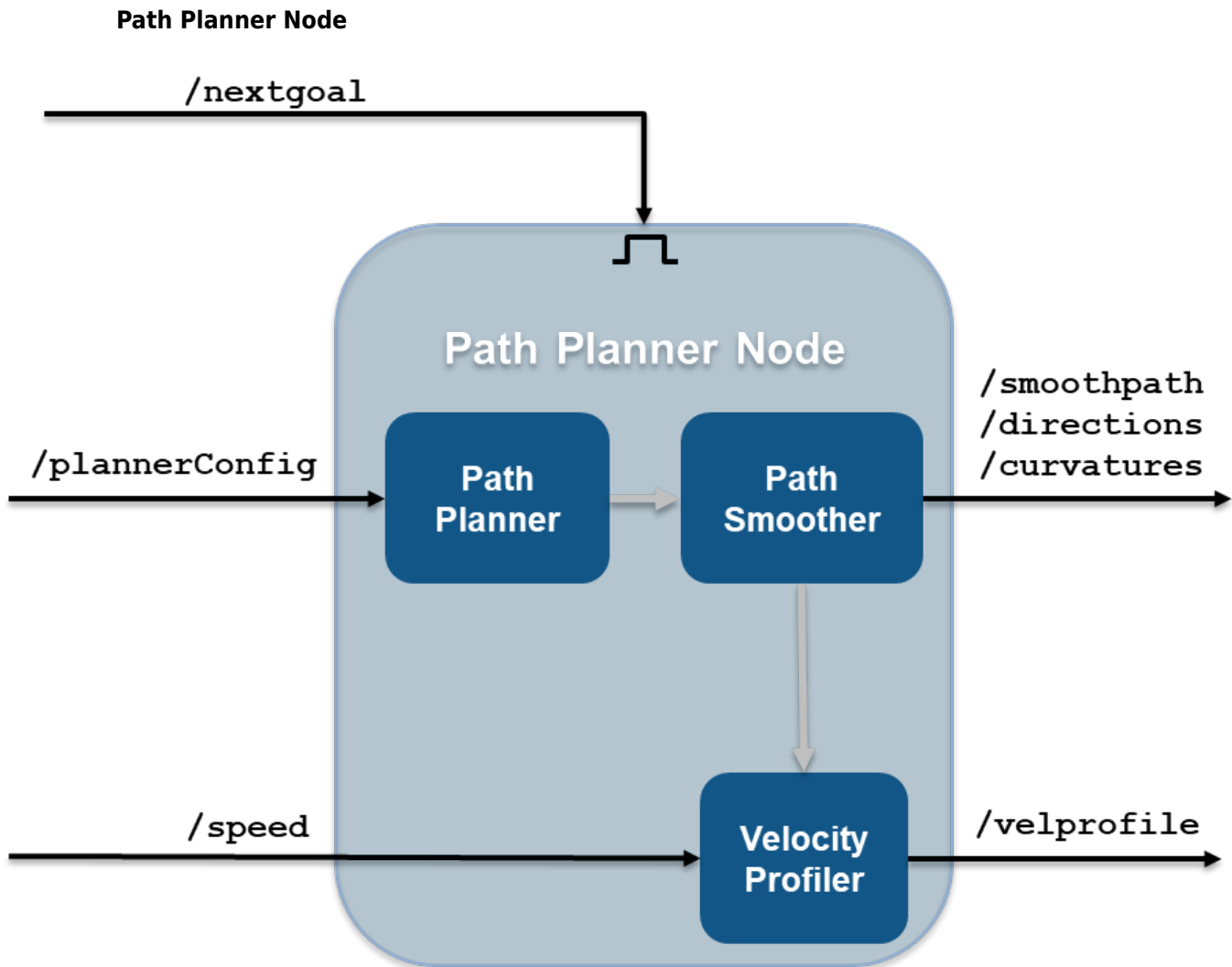
Path Planner Node

1. Open the path planner model.

```
open_system('ROS2ValetPathPlannerExample');
```

2. This model plans a feasible path through the environment map using a `pathPlannerRRT` (Automated Driving Toolbox) object, which implements the *optimal rapidly exploring random tree (RRT*)* algorithm and sends the plan to the controller over the ROS 2 network.

3. The `Path Planner` subsystem runs when a new message is available on `/plannerConfig` or `/nextgoal` topics.
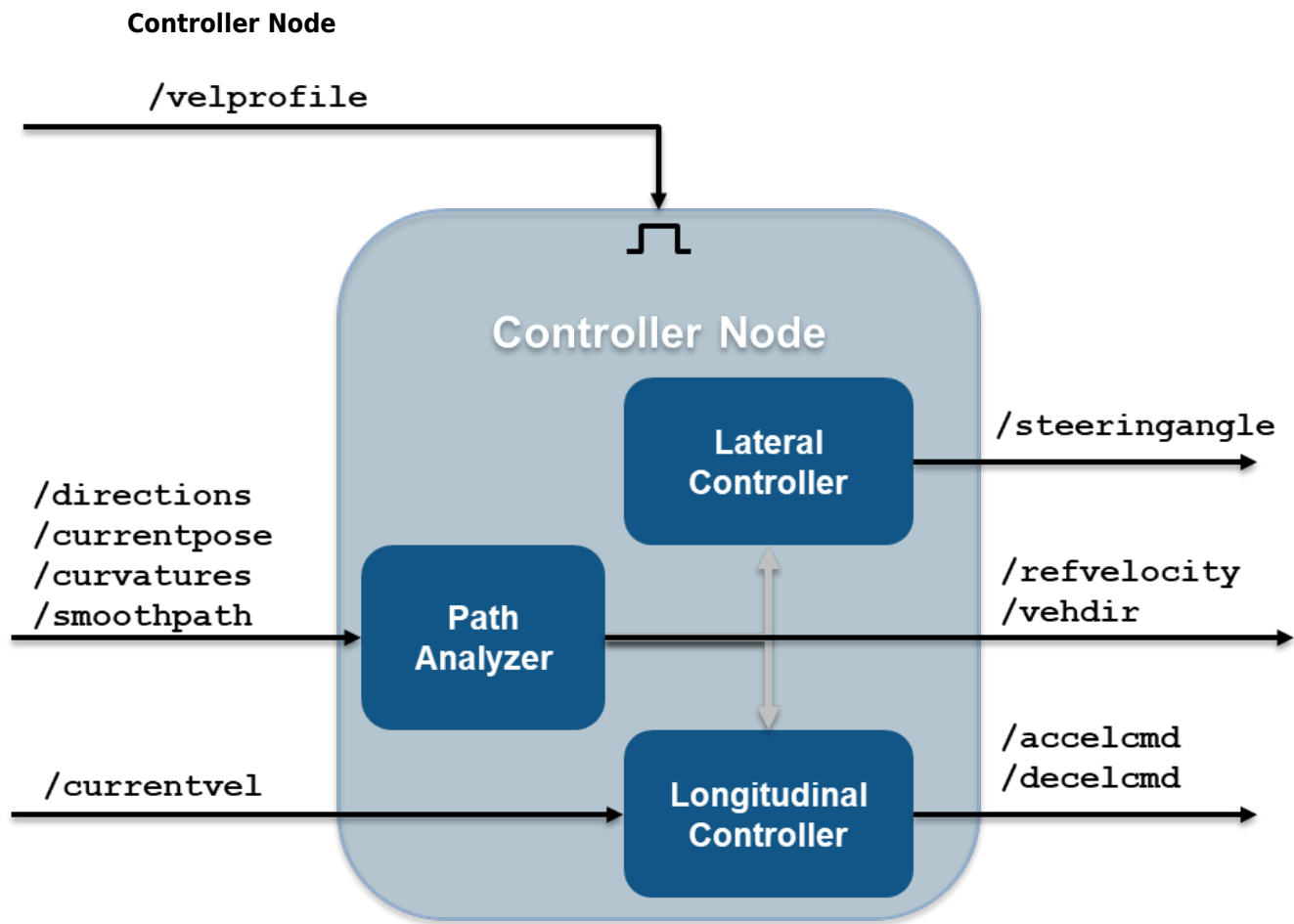
Controller Node

1. Open the vehicle controller model.

```
open_system('ROS2ValetControllerExample');
```

2. This model calculates and sends the steering and velocity commands over the ROS 2 network.

3. The Controller subsystem runs when a new message is available on the `/velprofile` topic.

Simulate the ROS 2 nodes to verify partitioning

Verify that the behavior of the model remains the same after partitioning the system into four ROS 2 nodes.

1. Load the pre-recorded localization map data in MATLAB base workspace using the `exampleHelperROSValetLoadLocalizationData` helper function.

```
exampleHelperROSValetLoadLocalizationData;
```

2. Open the simulation model.

```
open_system('ROS2ValetSimulationExample.slx');
```

In the left parking selection area, you can also select a spot. The default parking spot is the sixth spot at the top row.

3. In the SIMULATION tab, click **Run** from **SIMULATE** section or run
`sim('ROS2ValetSimulationExample.slx')` in MATLAB Command Window. A figure opens and
shows how the vehicle tracks the reference path. The blue line represents the reference path while
the red line is the actual path traveled by the vehicle. Simulation for all the models stop when the
vehicle reaches the final parking spot.

`sim('ROS2ValetSimulationExample.slx');`
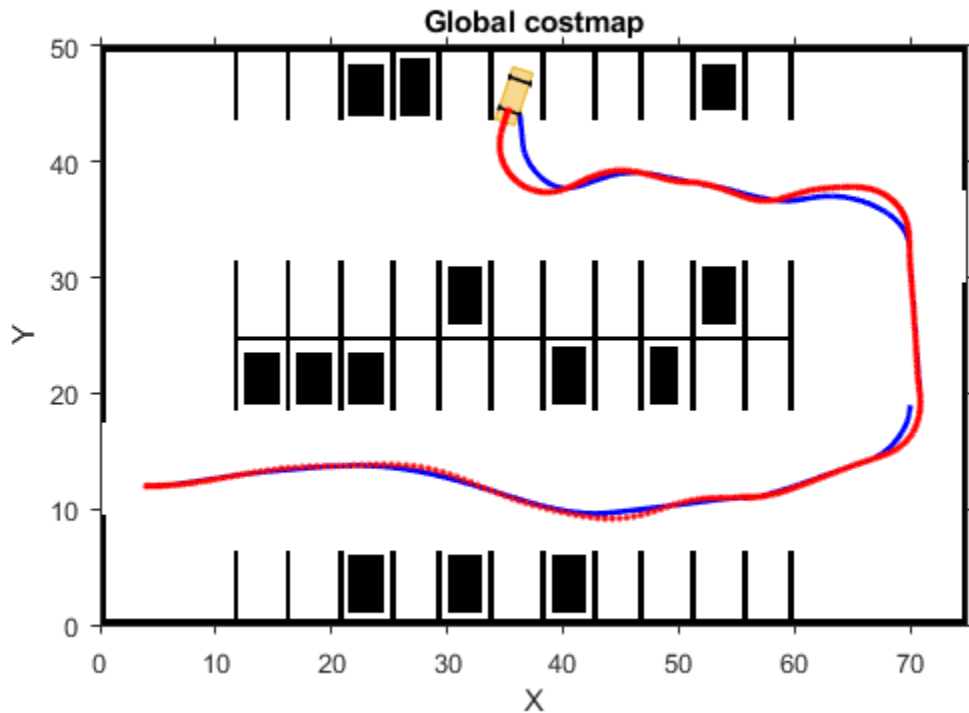
Simulation Results

The `Visualization` subsystem in vehicle model generates the results for this example.

`open_system('ROS2ValetVehicleExample/Vehicle model/Visualization');`

The `visualizePath` block is responsible for creating and updating the plot of the vehicle paths
shown previously. The vehicle speed and steering commands are displayed in a scope.

`open_system("ROS2ValetVehicleExample/Vehicle model/Visualization/Commands")`
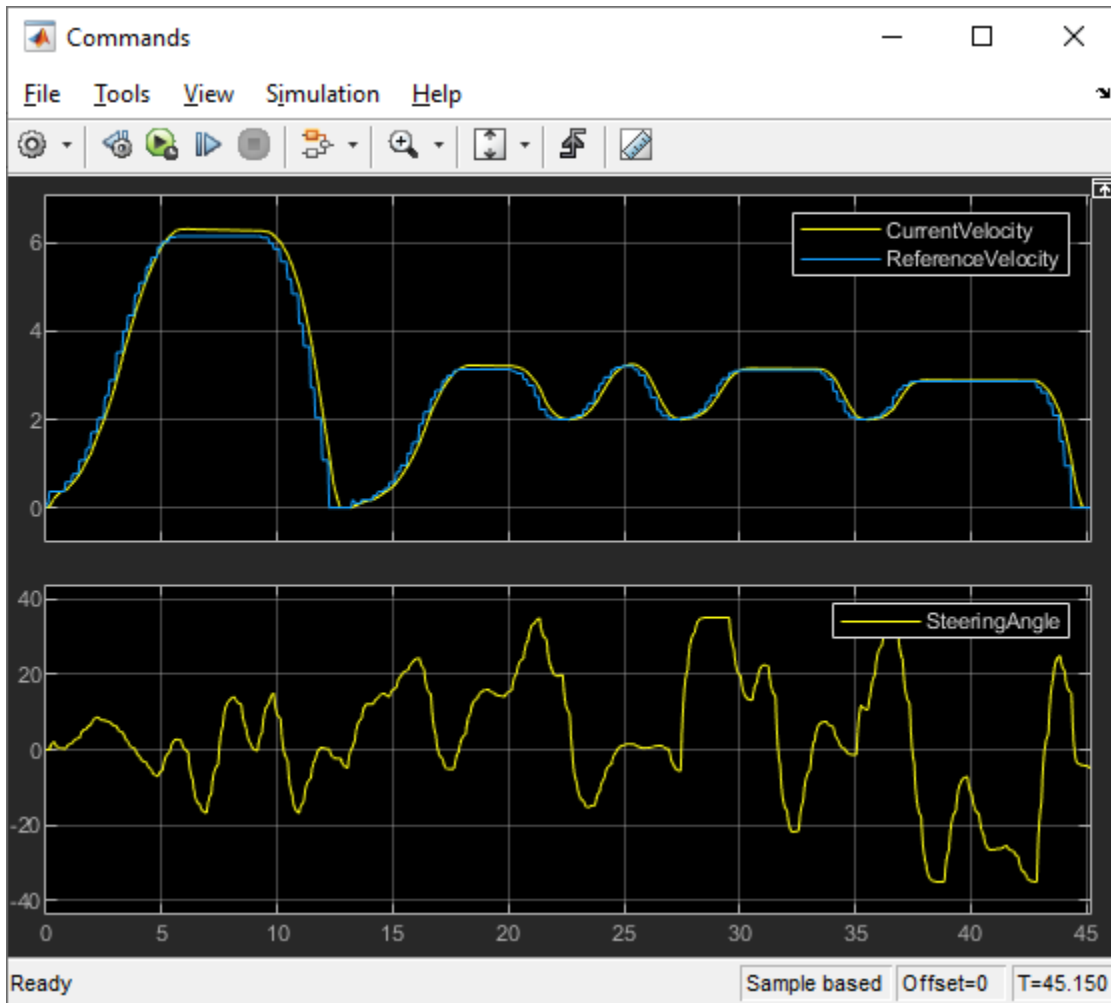
Deploy ROS 2 Nodes

Generate ROS 2 applications for `Behavioral Planner`, `Path planner` and `Controller` nodes.
Simulate the `Vehicle` node in MATLAB and compare the results with simulation.

Generate and deploy `Behavioral Planner`, `Path Planner` and `Controller` node applications
using `exampleHelperROS2ValetDeployNodes` helper function. The helper function calls `slbuild`
(Simulink) command with the name of the Simulink model as input argument, for each model, to
generate C++ code and deploy the application on the host computer.

```
exampleHelperROS2ValetDeployNodes(); % generate C++ code and deploy the application for ROS 2 no

### Starting build procedure for: ROS2ValetBehavioralPlannerExample
### Generating code and artifacts to 'Model specific' folder structure
### Generating code into build folder: C:\Users\joshchen\OneDrive - MathWorks\Documents\MATLAB\E>

### Generated code for 'ROS2ValetBehavioralPlannerExample' is up to date because no structural, p
### Evaluating PostCodeGenCommand specified in the model
### Using toolchain: Colcon Tools
### Building 'ROS2ValetBehavioralPlannerExample': all
Running colcon build in folder 'C:/Users/joshchen/OneDrive - MathWorks/Documents/MATLAB/Examples,
Success
```

```
### Successfully generated all binary outputs.
### Successful completion of build procedure for: ROS2ValetBehavioralPlannerExample
### Creating HTML report file ROS2ValetBehavioralPlannerExample_codegen_rpt.html

Build Summary

Top model targets built:

Model                          Action        Rebuild Reason
=========================================================================================
ROS2ValetBehavioralPlannerExample  Code compiled  Compilation artifacts were out of date.

1 of 1 models built (0 models already up to date)
Build duration: 0h 1m 32.504s
### Starting build procedure for: ROS2ValetPathPlannerExample
### Generating code and artifacts to 'Model specific' folder structure
### Generating code into build folder: C:\Users\joshchen\OneDrive - MathWorks\Documents\MATLAB\E>
### Generated code for 'ROS2ValetPathPlannerExample' is up to date because no structural, paramet
### Evaluating PostCodeGenCommand specified in the model
### Using toolchain: Colcon Tools
### Building 'ROS2ValetPathPlannerExample': all
Running colcon build in folder 'C:/Users/joshchen/OneDrive - MathWorks/Documents/MATLAB/Examples,
Success
### Successfully generated all binary outputs.
### Successful completion of build procedure for: ROS2ValetPathPlannerExample
### Creating HTML report file ROS2ValetPathPlannerExample_codegen_rpt.html

Build Summary

Top model targets built:

Model                          Action        Rebuild Reason
=========================================================================================
ROS2ValetPathPlannerExample  Code compiled  Compilation artifacts were out of date.

1 of 1 models built (0 models already up to date)
Build duration: 0h 1m 53.874s
### Starting build procedure for: ROS2ValetControllerExample
### Generating code and artifacts to 'Model specific' folder structure
### Generating code into build folder: C:\Users\joshchen\OneDrive - MathWorks\Documents\MATLAB\E>
### Generated code for 'ROS2ValetControllerExample' is up to date because no structural, paramete
### Evaluating PostCodeGenCommand specified in the model
### Using toolchain: Colcon Tools
### Building 'ROS2ValetControllerExample': all
Running colcon build in folder 'C:/Users/joshchen/OneDrive - MathWorks/Documents/MATLAB/Examples,
Success
### Successfully generated all binary outputs.
### Successful completion of build procedure for: ROS2ValetControllerExample
### Creating HTML report file ROS2ValetControllerExample_codegen_rpt.html

Build Summary

Top model targets built:

Model                          Action        Rebuild Reason
=========================================================================================
ROS2ValetControllerExample  Code compiled  Compilation artifacts were out of date.
```

```
1 of 1 models built (0 models already up to date)
Build duration: 0h 2m 1.851s
```

Open the vehicle model and start simulation.

```
open_system("ROS2ValetVehicleExample");
set_param("ROS2ValetVehicleExample","SimulationCommand","start");
```

Verify that the results from simulation match with the deployed ROS 2 nodes.

ROS Topics

ROS Network Setup

Introduction

Setting up a ROS network enables communication between different devices. These participants, or *nodes*, all register with a *ROS master* to share information. Each ROS network has only one, unique master. Each node is usually a separate device, although one device can have multiple nodes running. MATLAB® acts as one of these nodes when communicating on an existing ROS Network.

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 created by calling `rosinit`. For more information on setting up and using the ROS network, see "Network Connection and Exploration".

Nodes communicate by sending messages using entities called publishers, subscribers, and services. Publishers send data using 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. Nodes that are registered with the master can communicate with all other registered nodes. 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. All nodes must 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 operate on this global node.

See Also
rosinit | rosnode | rostopic

Related Examples
- "Get Started with ROS" on page 1-2
- "Connect to a ROS Network" on page 1-7
- "Robot Operating System (ROS)"

Built-In Message Support

In this section...
"ROS Message Structure" on page 3-4
"Limitations of ROS Messages in MATLAB" on page 3-5
"ROS Data Type Conversions" on page 3-5
"Supported Messages" on page 3-6

MATLAB supports a large library of ROS message types. This topic covers how MATLAB works with ROS Messages by describing message structure, limitations for ROS messages, and supported ROS data types. Refer to the full list of built-in message types at the end of this article.

For information about ROS 2 messages, see "Work with Basic ROS 2 Messages" on page 2-11.

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". 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 store the data relevant to that message type in a way similar to structure arrays. 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. The `MessageType` property of each message contains 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 `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
```

For more information on the ROS message structure in MATLAB, see "Work with Basic ROS Messages" on page 1-15.

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 the data types 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 3-5.

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.

ROS Toolbox supports ROS Indigo and Hydro platforms, but your own ROS installation may have different message versions. To overwrite our current message catalog, you can utilize the "ROS 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 "ROS Custom Message Support" on page 3-17.

```
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
nav_msgs
nav2d_msgs
nav2d_navigator
nav2d_operator
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_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" on page 1-15
- "Exchange Data with ROS Publishers and Subscribers" on page 1-20
- "Work with Specialized ROS Messages" on page 1-44

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 localization algorithms make the assumption that your sensor is mounted in the center of the robot. So depending on your robot configuration, you must transform your laser scan data so it is relative to the robots 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. For this example, a sample network is already set up with an existing transformation tree.

```
rosinit('192.168.233.131')
```

```
Initializing global node /matlab_global_node_68056 with NodeURI http://192.168.233.1:62899/
```

Create the ROS transformation tree using `rostf`. The function connects to the ROS parameter server for the network. 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)
```

```
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
```

```
cartScanData = scan.readCartesian;
cartScanData(:,3) = 0;
homScanData = cart2hom(cartScanData);
```

Ensure that there is something within scanning distance of your robot. If nothing is detected, a laser scan will contain only NaN values, resulting in an error from cart2hom.

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_68056 with NodeURI http://192.168.233.1:62899/
```

ROS Log Files (rosbags)

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 Tables". Here is an example output of this table:

```
ans =

                        NumMessages       MessageType              MessageDefinition
                        _____       _____      _____

    /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()
```
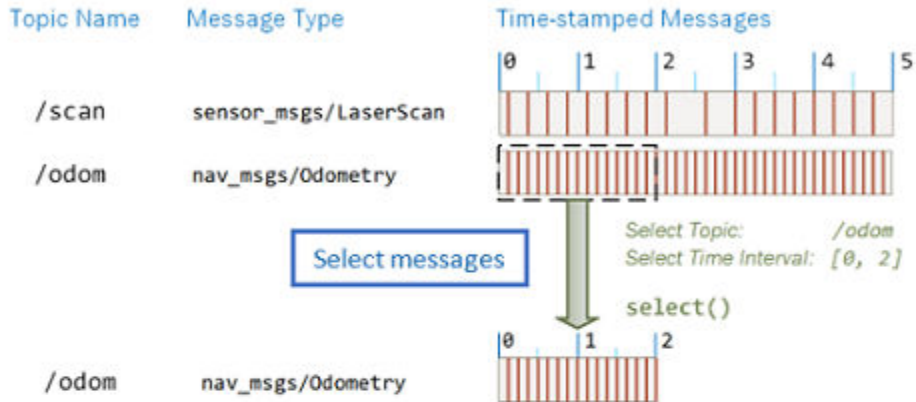
Timeseries Object

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

Time Series Plot:/odom Properties
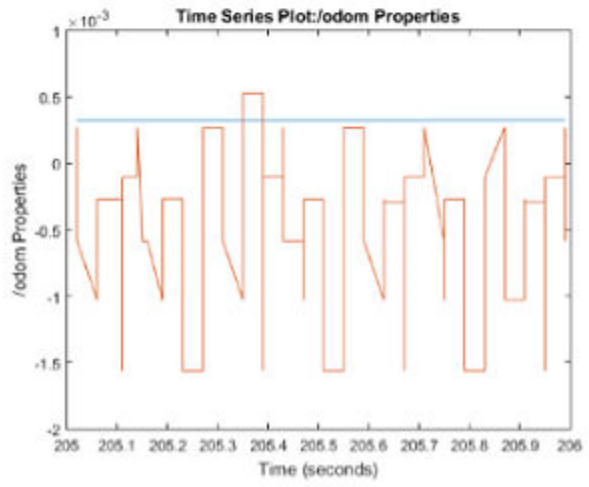
Limitations

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" on page 1-31

ROS Custom Message Support

In this section...
"Custom Message Overview" on page 3-17
"Custom Message Contents" on page 3-17
"Custom Message Creation Workflow" on page 3-18

Custom Message Overview

Custom messages are user-defined messages that you can use to extend the set of message types currently supported in ROS 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.

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` with the file path to your custom message package 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 3-20.

If this is your first time using ROS custom messages, check "ROS System Requirements".

Custom Message Contents

ROS custom messages are specified in ROS package folders that contains `msg` and `srv` directories.

Note At any time, there should only be one custom messages folder on the MATLAB path. This folder can contain multiple packages. It is recommended that you keep them all in one unique folder.

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.
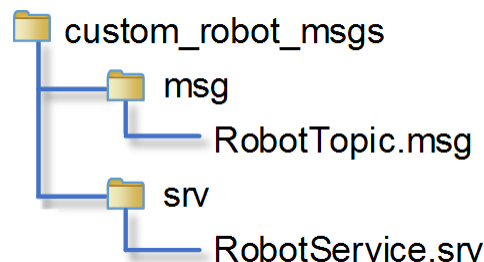
The package contains one custom message type in `RobotTopic.msg` and one custom service type in `RobotService.srv`. MATLAB uses these files to generate the necessary files for using the custom messages contained in the package. For more information on creating `msg` and `srv` files, see Creating a ROS 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`.

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.

The `rosgenmsg` function takes your custom message files (`.msg` and `.srv`) and converts each message type to working MATLAB code. The `rosgenmsg` function looks 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 function then creates the required MATLAB M-files for the different message classes.

After the `rosgenmsg` function creates these files, you must add the class files to the MATLAB path. These steps are given as prompts in the MATLAB Command Window.

1   **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.

2 **Refresh all message class definitions**, which requires clearing the workspace:

```
clear classes
rehash toolboxcache
```

3 **Verify messages are available**: Use `rosmsg list` or the `rosmessage` function to check that the new custom messages are available.

For an example of this procedure, see "Create Custom Messages from ROS Package" on page 3-20. 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 ROS Toolbox provides. Repeat this generation procedure when you would like to update or create new message types.

You must maintain the 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.

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. Ensure that you either install or copy the custom message package to your Linux system before building the generated code.

See Also
ros2genmsg | rosgenmsg

Related Examples
- "Create Custom Messages from ROS Package" on page 3-20
- "ROS 2 Custom Message Support" on page 2-29
- "ROS System Requirements"

Create Custom Messages from ROS Package

In this example, you go through the procedure for creating ROS custom messages in MATLAB. You must have a ROS package that contains the required `msg` and `srv` files. The correct file contents and folder structure are described in "Custom Message Contents" on page 3-17. This folder structure follows the standard ROS package conventions. Therefore, if you have any existing packages, they should match this structure.

To ensure you have the proper third-party software, see "ROS System Requirements".

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 messages in a location and note the folder path. We recommend you put all your custom message definitions in a single packages folder.

  ```
  folderpath = 'c:\MATLAB\custom_msgs\packages';
  ```
- **(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 the `rosgenmsg` function to create custom messages for MATLAB.

  ```
  rosgenmsg(folderpath)
  ```
- Then, follow steps from the output of `rosgenmsg`.

1 Add the given files to the MATLAB path by running `addpath` and `savepath` in the command window.

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

2 Refresh all message class definitions, which requires clearing the workspace:

  ```
  clear classes
  ```

  ```
  rehash toolboxcache
  ```

3 You can then use the custom messages like any other ROS messages supported in ROS 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.
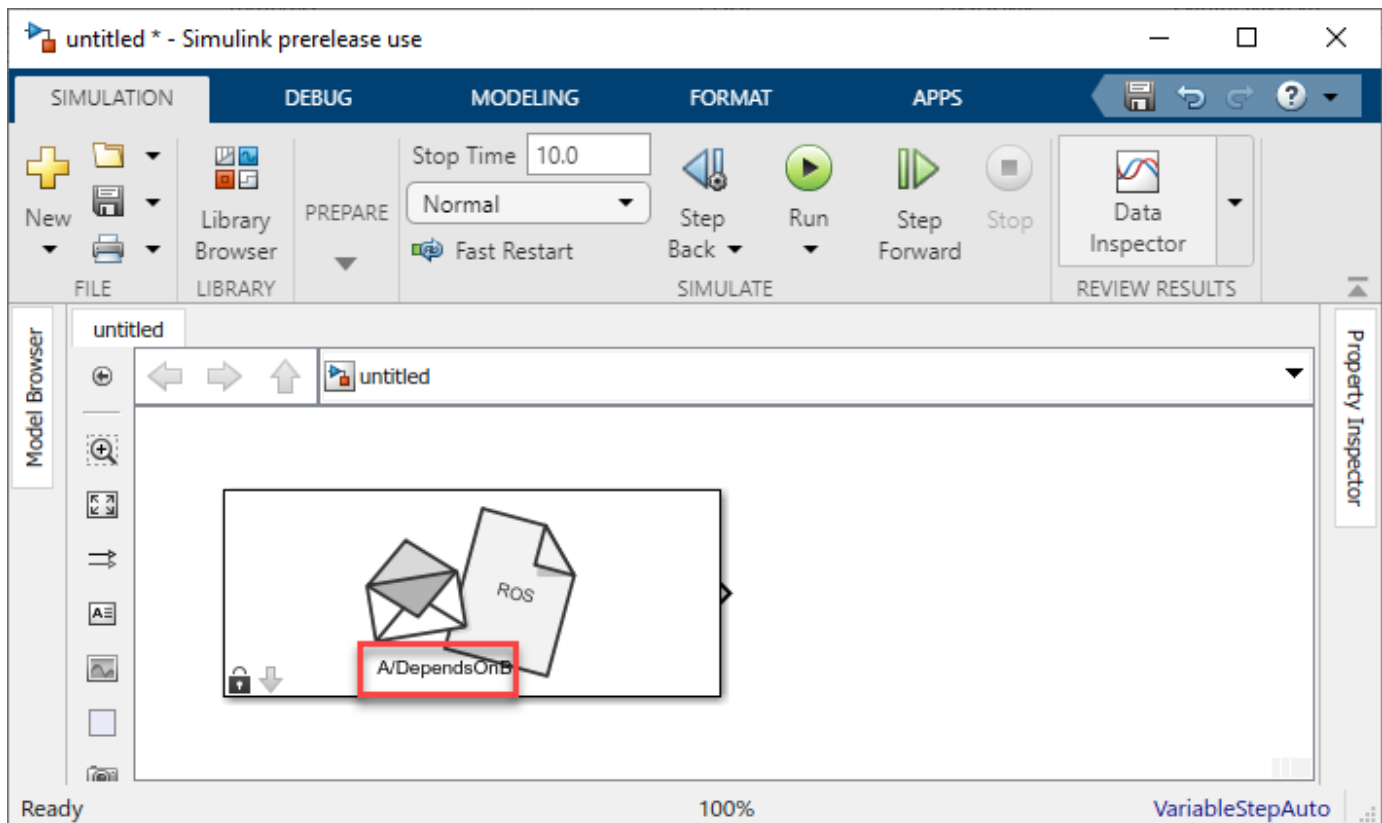
```
custommsg = rosmessage('B/Standalone');
custommsg2 = rosmessage('A/DependsOnB');
```
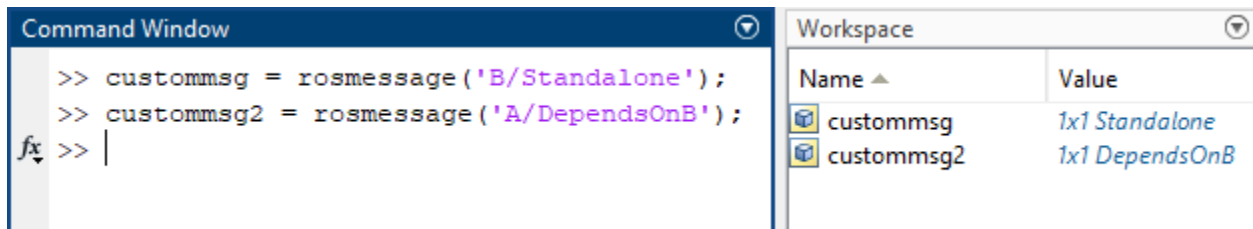

Custom messages can also be used with the ROS Simulink blocks.

See Also

ros2genmsg | rosgenmsg

Related Examples

- "Create Custom Messages from ROS Package" on page 3-20
- "ROS 2 Custom Message Support" on page 2-29
- "ROS System Requirements"

ROS Actions Overview

Client to Server Relationship

ROS Actions have a client-to-server communication relationship with a specified protocol. The actions use 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.
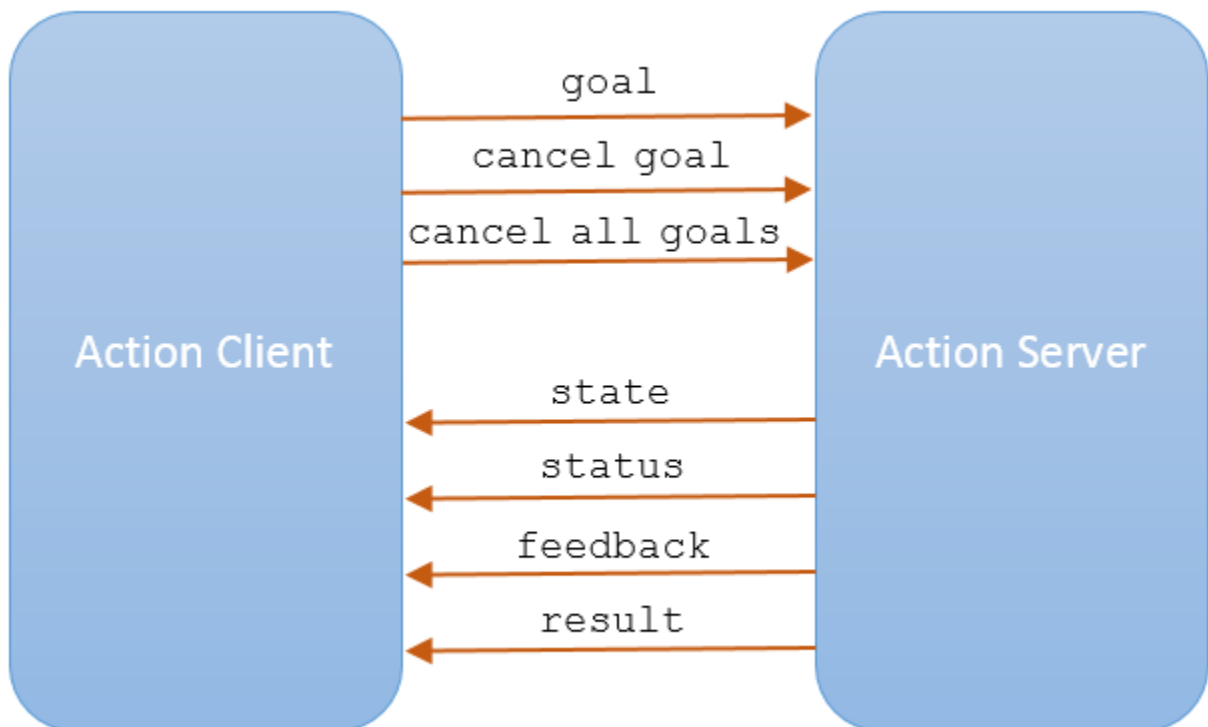
Use the `sendGoal` function to send goals to the server. Send the goal and wait for it to complete using `sendGoalAndWait`. This function enables you to return the result message, final state of the goal and status of the server. While the server is executing a goal, the callback function, `FeedbackFcn`, 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.
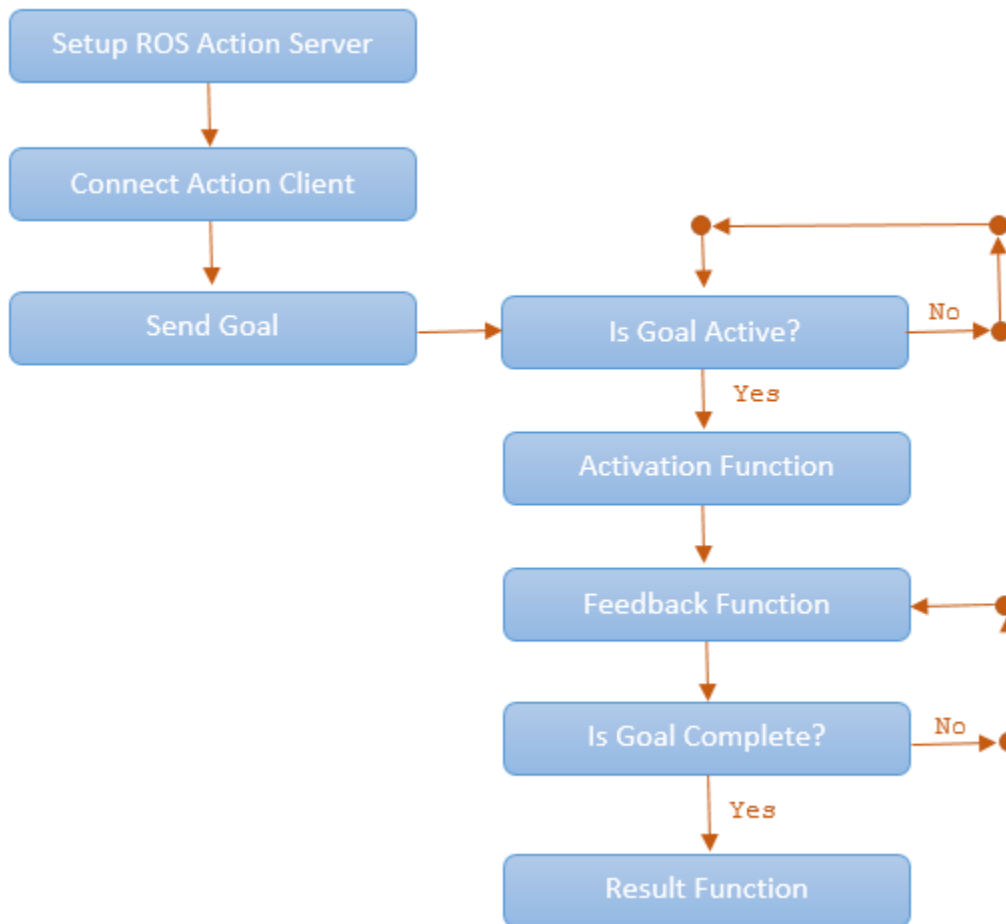
- Setup ROS action server. Check what actions are available on a ROS network by typing `rosaction list` in the MATLAB command window.

- 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 use 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 3-26

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.

Follow the steps in "Get Started with Gazebo and a Simulated TurtleBot" on page 1-102 to setup a simulated TurtleBot. After starting the virtual machine, launch **Gazebo Empty** world using desktop shortcut and open the terminal window.

To run the Turtlebot ROS action server, use this command on the ROS distribution terminal.

`~/start-turtlebot-move-action-server.sh`
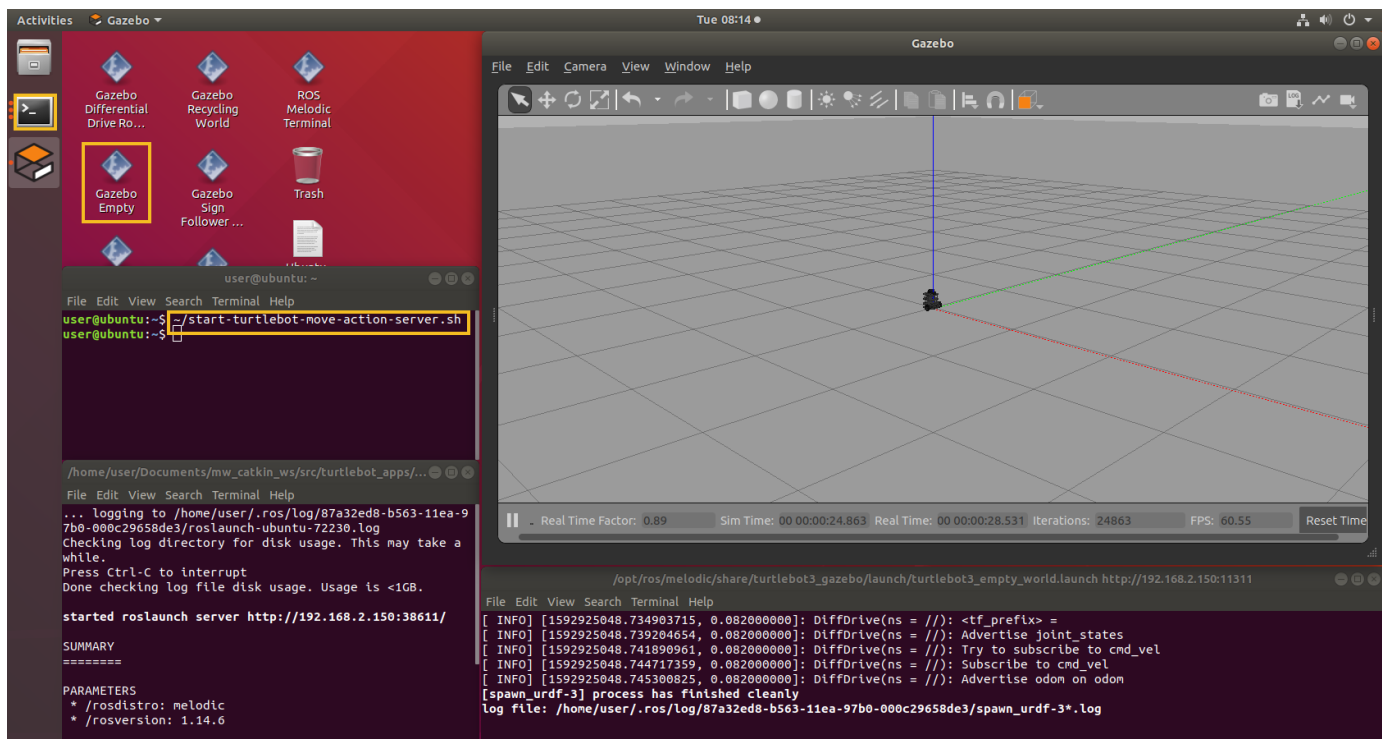
Connect to a ROS network. You must have an ROS action server setup on this network. Change `ipaddress` to the address of your ROS network.

```
ipaddress = '192.168.2.150';
rosinit(ipaddress,11311);
```

Initializing global node /matlab_global_node_94218 with NodeURI http://192.168.2.1:51650/

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)

resultMsg =
  ROS TurtlebotMoveResult message with properties:

        MessageType: 'turtlebot_actions/TurtlebotMoveResult'
       TurnDistance: 0
    ForwardDistance: 2.0022

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

Disconnect from the ROS network.

```
rosshutdown

Shutting down global node /matlab_global_node_94218 with NodeURI http://192.168.2.1:51650/
```

Execute Code Based on ROS Time

Using a `rosrate` object allows you to control the rate of your code execution based on the ROS Time `/clock` topic or system time on your computer. By executing code at constant intervals, you can accurately time and schedule tasks. These examples show different applications for the `rosrate` object including its uses with ROS image messages and sending commands for robot control.

For other applications based on system time, consider the `rateControl` object.

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
```

```
Launching ROS Core...
..........Done in 15.6689 seconds.
Initializing ROS master on http://172.30.68.186:50070.
Initializing global node /matlab_global_node_35650 with NodeURI http://bat1070102win64:57477/
```

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 =
  rateControl with properties:

          DesiredRate: 10
        DesiredPeriod: 0.1000
        OverrunAction: 'drop'
     TotalElapsedTime: 0.0933
           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: [1x100 double]
           NumPeriods: 100
        AveragePeriod: 0.1000
    StandardDeviation: 0.0157
           NumOverruns: 0
```

Shut down ROS network

```
rosshutdown
```

```
Shutting down global node /matlab_global_node_35650 with NodeURI http://bat1070102win64:57477/
Shutting down ROS master on http://172.30.68.186:50070.
```

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 = '192.168.203.129'; % Replace with your network address
rosinit(ipaddress)
```

```
Initializing global node /matlab_global_node_10650 with NodeURI http://192.168.203.1:50899/
```

```
sub = rossubscriber('/camera/rgb/image_raw');
pub = rospublisher('/camera/gray/image_gray','sensor_msgs/Image');
msgGray = rosmessage('sensor_msgs/Image');
msgGray.Encoding = 'mono8';
```
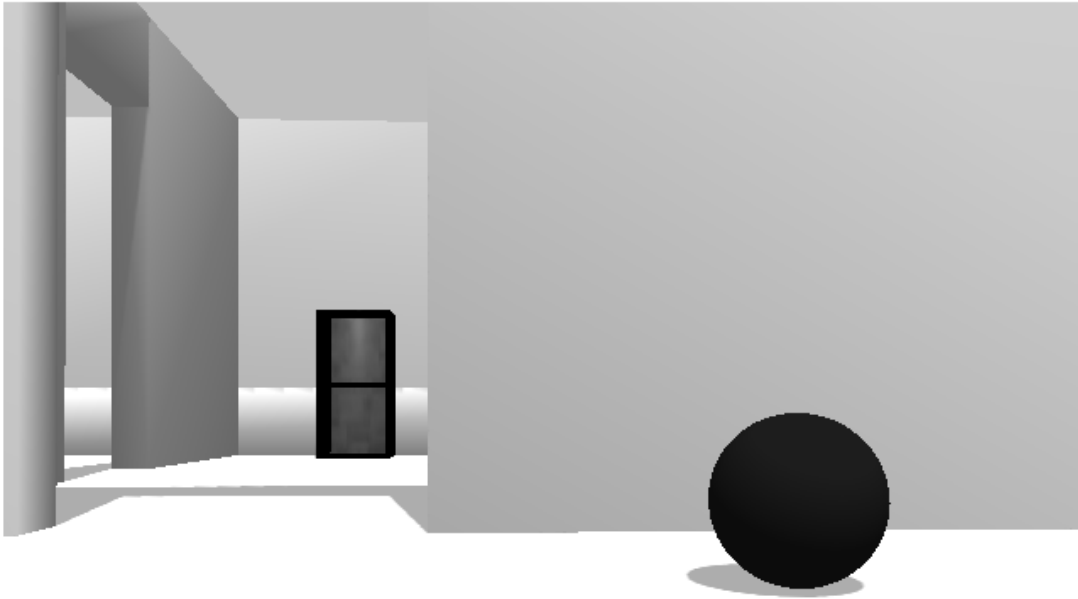
Receive the first image message. Extract image and convert to a grayscale image. Display grayscale image and publish the message.

```
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 = struct with fields:
              Periods: [1×50 double]
           NumPeriods: 50
        AveragePeriod: 0.1000
    StandardDeviation: 0.0083
          NumOverruns: 0
```

Shut down ROS node

```
rosshutdown
```

```
Shutting down global node /matlab_global_node_10650 with NodeURI http://192.168.203.1:50899/
```

See Also
rateControl | rosrate | waitfor

ROS Simulink Topics

ROS Simulink Support and Limitations

In this section...
"ROS Model Reference" on page 4-2
"Remote Desktop" on page 4-2
"ROS 2 Model Build Failure" on page 4-2

The ROS Toolbox does not support the following ROS features in Simulink:

- ROS Service Servers
- 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 Service Servers: `rosservice`, `rossvcserver`
- ROS Actions: `rosaction`, `rosactionclient`
- Transformation trees: `rostf`, `transform`, `getTransform`

For ROS 2, Simulink only supports:

- Publish
- Subscribe

To see a full list of ROS support in Simulink, see "ROS in Simulink".

ROS Model Reference

Simulink supports model reference when using ROS blocks with some limitations.

- Multiple references to the same model results in an error due to duplicate buses with the same name being created for ROS messages used by the ROS blocks. You can only reference a model once in a parent model.
- Referenced data dictionaries are not supported with variable-size ROS messages.
- Simulation Mode only supports `Normal` mode.

Remote Desktop

Running ROS networks from remote desktop applications can cause ROS communication to be interrupted. Consider executing your network without a remote connection.

ROS 2 Model Build Failure

A space in the installation path of Python 3.7 causes an error related to the creation of a Python virtual environment when generating code from a ROS 2 Simulink model. e.g. C:\Program Files \Python37\python.exe

See Also

Related Examples

- "ROS Parameters in Simulink" on page 4-9
- "ROS Simulink Interaction" on page 4-4
- "Manage Array Sizes for ROS Messages in Simulink" on page 4-24

ROS Simulink Interaction

In this section...
"MATLAB ROS Information" on page 4-4
"Simulink ROS Node" on page 4-4
"Differences Between Simulation and Generated Code" on page 4-4
"Publishers and Subscribers in Simulink" on page 4-5
"ROS Model Reference" on page 4-5

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 4-18 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®" on page 1-92 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 Model Reference

Simulink supports model reference when using ROS blocks with one limitation. Multiple references to the same model results in an error due to duplicate buses with the same name being created for ROS messages used by the ROS blocks. You can only reference a model once in a parent model.

Publish and Subscribe to ROS Messages in Simulink

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

open_system('rosPubSubExample.slx')
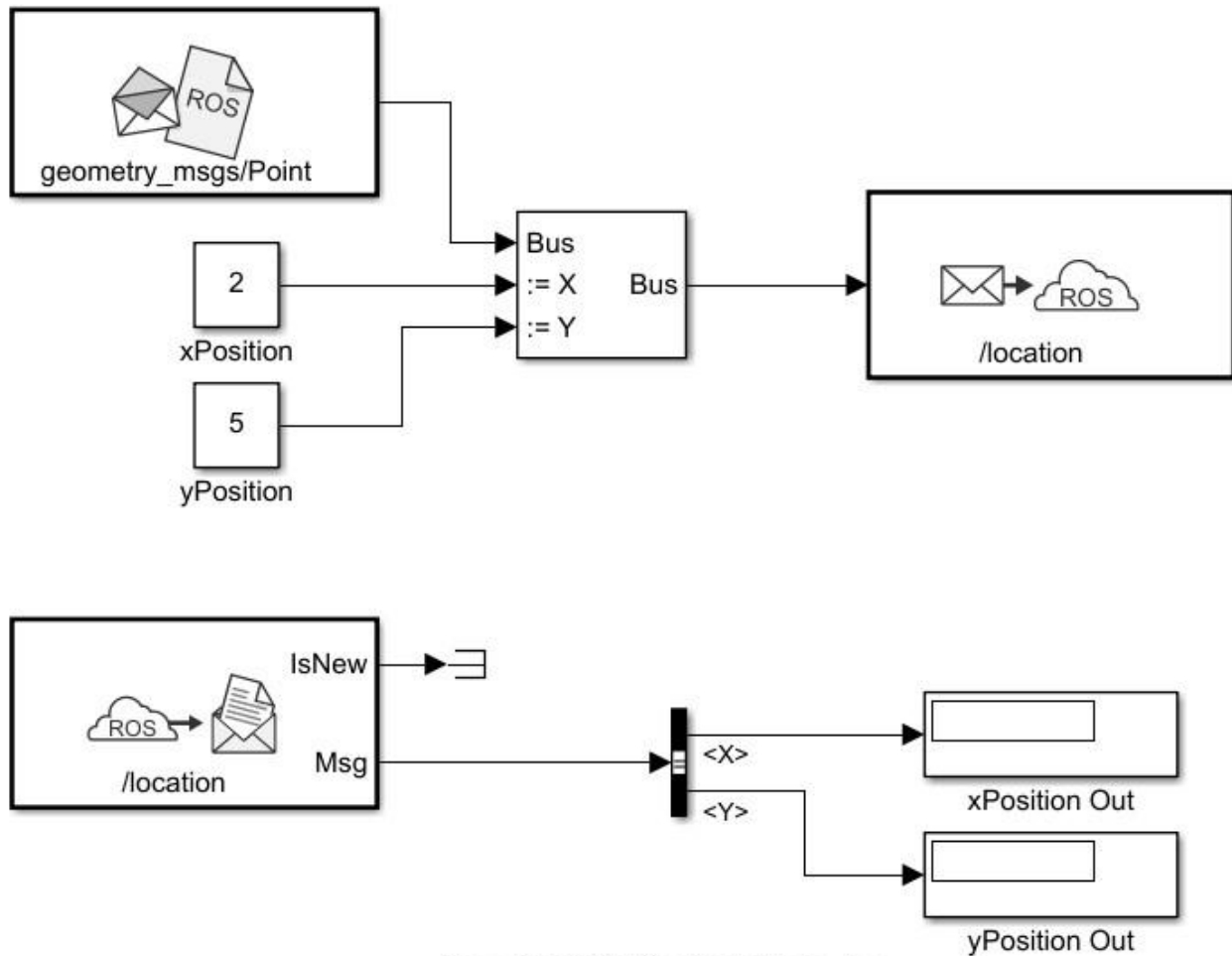
Copyright 2018 The MathWorks, Inc.

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
```

```
Launching ROS Core...
.........Done in 12.7381 seconds.
Initializing ROS master on http://172.30.68.186:50273.
Initializing global node /matlab_global_node_40275 with NodeURI http://bat1070102win64:57793/
```

Run the model. You should see the `xPosition Out` and `yPosition Out` displays show the corresponding values published to the ROS network.

```
sim('rosPubSubExample')
```

Shut down the ROS network.

```
rosshutdown
```

```
Shutting down global node /matlab_global_node_40275 with NodeURI http://bat1070102win64:57793/
Shutting down ROS master on http://172.30.68.186:50273.
```

Time Stamp a ROS Message Using Current Time in Simulink

This example shows how to 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
```

```
Shutting down ROS master on http://172.30.68.186:56317.

Warning: Error shutting down the ROS master.

Launching ROS Core...
......Done in 12.3519 seconds.
Initializing ROS master on http://172.30.68.186:55176.
```
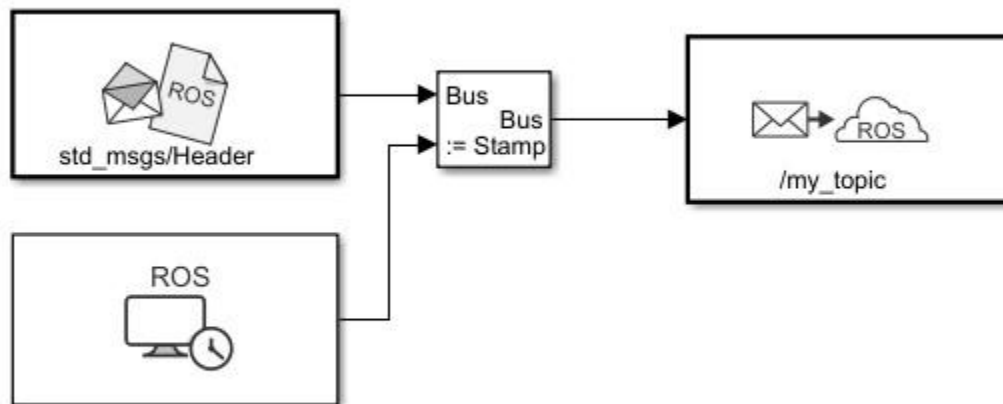
Open the Simulink model provided with this example. The model uses a **Bus Assignment** block to add the **Current Time** output to the Stamp field of the ROS message.

```
open_system('current_time_stamp_message.slx')
```


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

```
sim('current_time_stamp_message.slx')
```

Shut down the ROS network.

```
rosshutdown
```

ROS Parameters in Simulink

In this section...

These examples show how to get, set, compare, and manipulate ROS parameters in Simulink. To run these examples, you must first set up a ROS network using `rosinit`. To set network-wide settings and share values with the whole network, start a ROS parameter server using `rosparam`. Follow these examples to see how to work with parameters in Simulink, including using string parameters.

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.
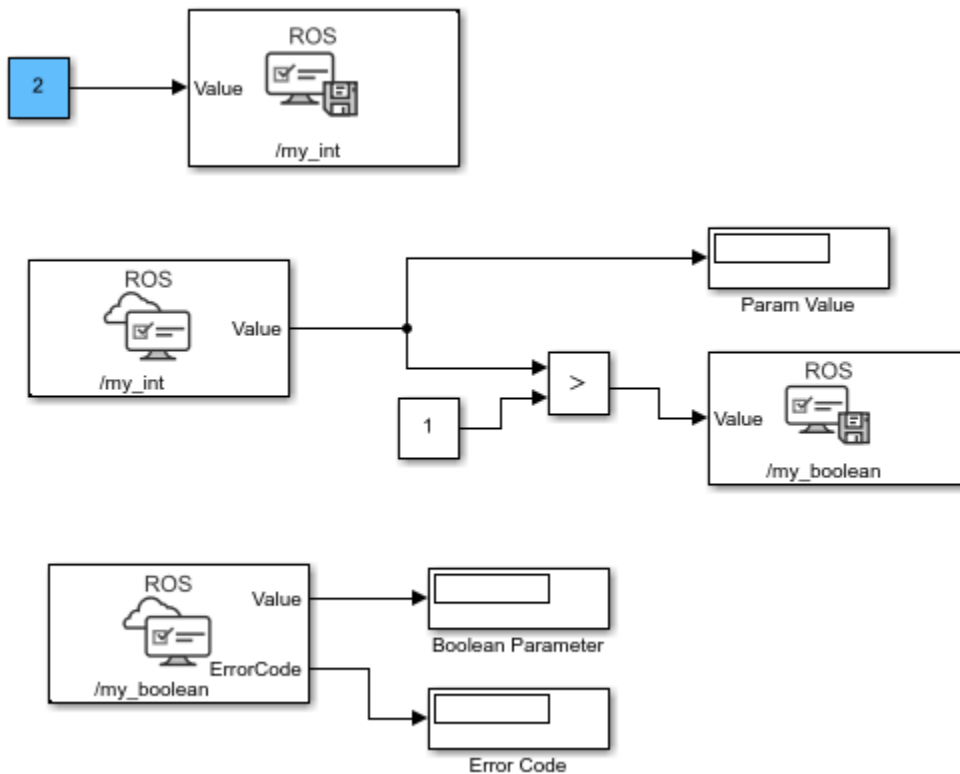
Set String Parameter on ROS Network

To create your string parameter, use a String Constant block and convert it to uint8 using a MATLAB function block. The converted uint8 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.
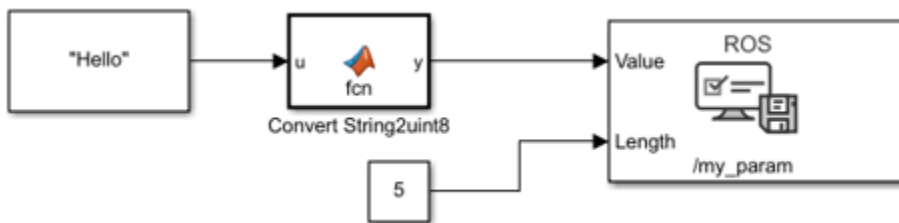
Compare ROS String Parameters

On ROS networks, strings parameters are stored as a `uint8` array. When you get from string parameters from the server, they are returned as a `char` array. In Simulink®, they are cast as `uint8`, so you must use `uint8` character vectors when comparing to the ROS string parameters. You can use this comparison to trigger subsystems for larger models or validate settings for specific algorithms.

Connect to a ROS network. Set up the ROS Parameter tree.

```
rosinit
```

```
Launching ROS Core...
......Done in 10.374 seconds.
Initializing ROS master on http://172.30.68.186:52845.
```

```
ptree = rosparam;
```

Set a ROS parameter, `/camera_format`, to a string value. You can use string scalars or character vectors. The value is stored as a `uint8` array on the ROS parameter server and returned as `'jpeg'` in MATLAB®.

```
set(ptree,"/camera_format","jpeg")
pause(1)
pvalue = get(ptree,"/camera_format")
```

Run the attached Simulink® model. This model checks to see if the previously set camera format parameter is named 'jpeg'. To get the parameter off the server, use the Get Parameter block. Then, compare the parameter to a character vector cast as `uint8` from a Constant block, using a MATLAB function block. An output of 1 means the parameters match.

```
open_system("rosStringParameterCompare")
sim("rosStringParameterCompare");
```
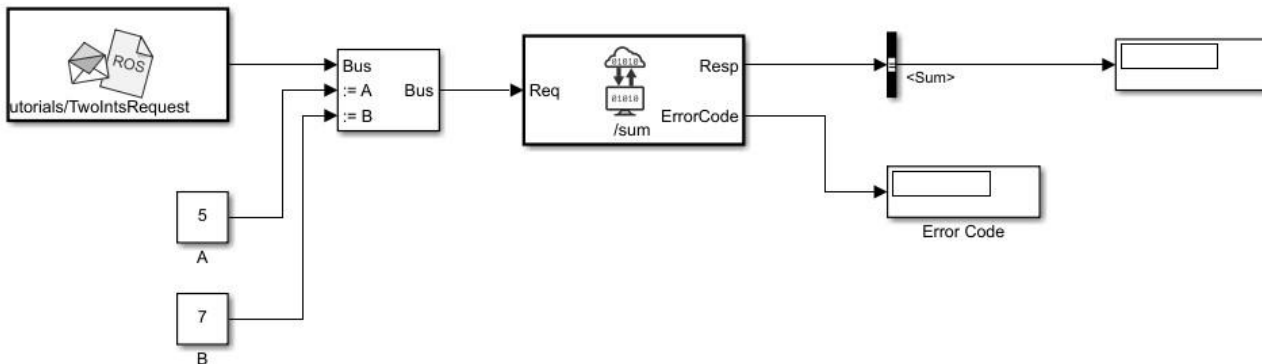

Shutdown ROS network.

```
rosshutdown
```

The `stringCompare` function is defined as:

```
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.

Connect to a ROS network and set up the ROS parameter server.

```
rosinit
```

```
Launching ROS Core...
.......Done in 10.3662 seconds.
Initializing ROS master on http://172.30.68.186:59772.
Initializing global node /matlab_global_node_39122 with NodeURI http://bat1070102win64:55810/
```

```
ptree = rosparam;
```

Set the "`/camera/rgb/image_raw/compressed/format`" parameter, and set up a publisher for the "`/camera/rgb/image_raw/compressed`" topic.

```
set(ptree,"/camera/rgb/image_raw/compressed/format","jpeg")
pub = rospublisher("/camera/rgb/image_raw/compressed","sensor_msgs/CompressedImage");
```

Open the Simulink® model. This model checks the image format parameter and compares the value to a `uint8` cast character vector, `uint8('jpeg')` using a MATLAB® Function block. The boolean output is fed to an AND operator with the IsNew output of a Subscribe block that gets the image off the network. If the parameter value is correct and a new message is received, the Subsystem "Process Image" is triggered.

Run the model and use the buttons in the model to change the image format parameter and verify the `strcmp` function works. The `eq` output should be 1 when the parameter is set to `'jpeg'`. While the model is running, it is expected that image messages are being published on the network.

```
open_system("rosImageFormatParameter")
```
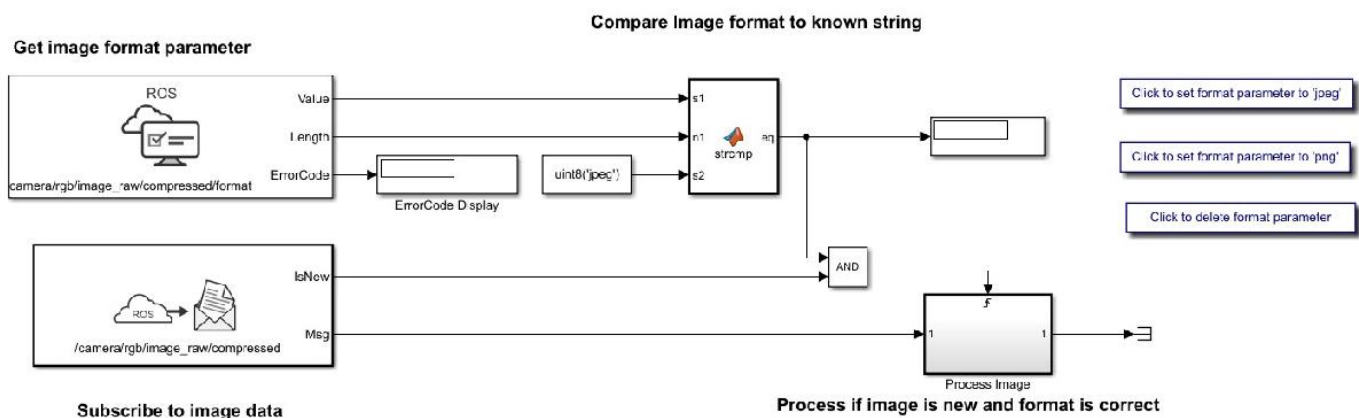

Shut down the ROS network.

```
rosshutdown
```

```
Shutting down global node /matlab_global_node_39122 with NodeURI http://bat1070102win64:55810/
Shutting down ROS master on http://172.30.68.186:59772.
```

The `strcmp` function in the MATLAB® Function block is defined as:

```matlab
function eq = strcmp(s1, n1, s2)
%#codegen

% Convert to proper strings
string1 = char(s1(1:n1));
string2 = char(s2);

eq = strcmp(string1, string2);
```
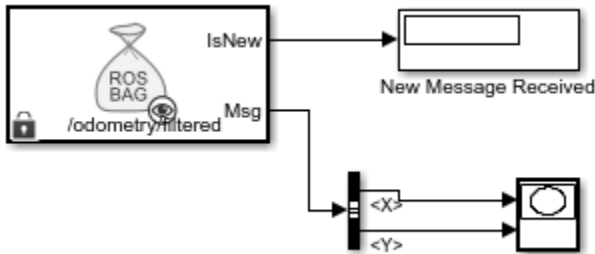
See Also

Get Parameter | Set Parameter

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. The `jackal_sim.bag` file is attached to this example.

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)
```
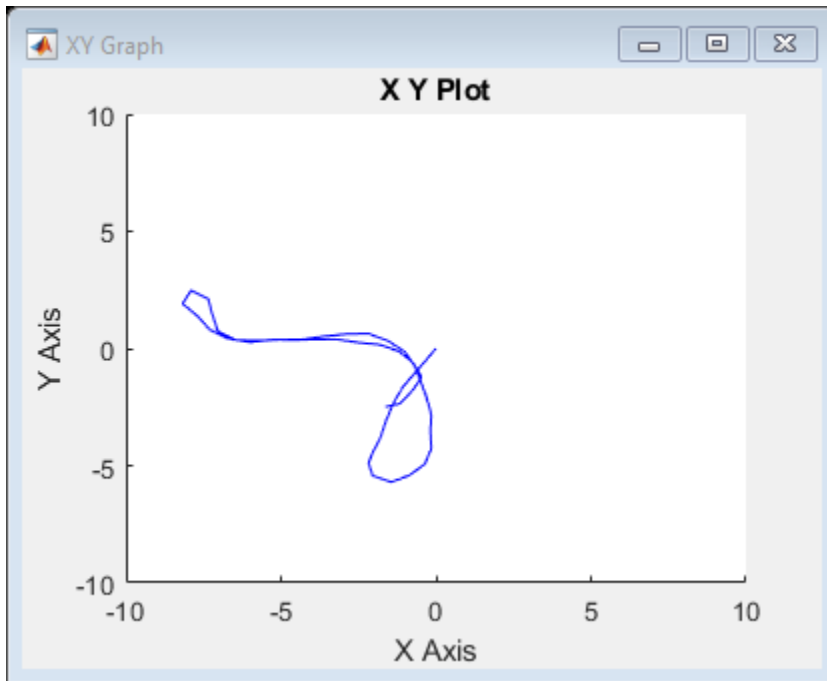
Call ROS Service in Simulink

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

Connect to a ROS network.

```
rosinit
```

```
Launching ROS Core...
.......Done in 9.3418 seconds.
Initializing ROS master on http://172.30.68.186:50742.
Initializing global node /matlab_global_node_74999 with NodeURI http://bat1070102win64:52781/
```

Set up a `roscpp_tutorials/TwoInts` service server message type and specify a example helper callback function. The call back function provided 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',@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 together.

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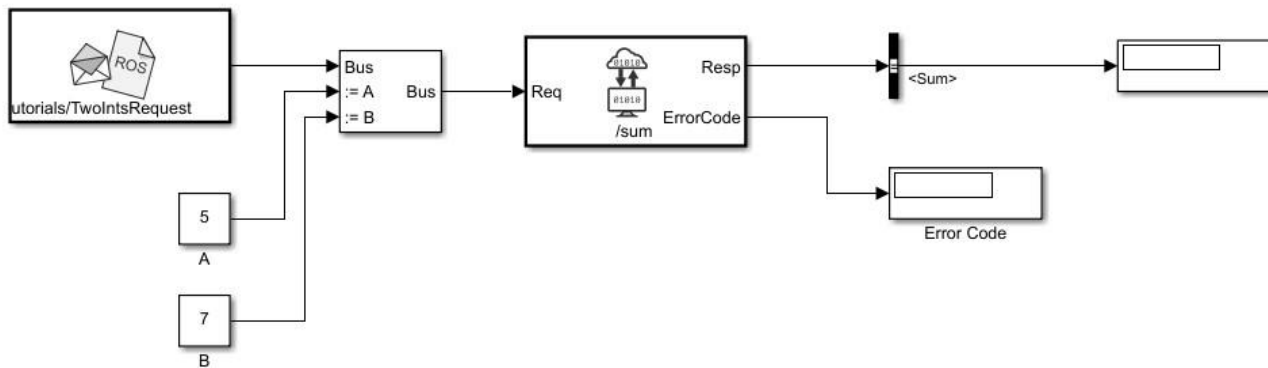

Run the model. The service call should return 0 in the Resp output as part of the response message. An error code of 0 indicates the service call was successful. You can ignore warnings about converting data types.

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

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

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

```
Warning: The property "sum" in ROS message type "roscpp_tutorials/TwoIntsResponse" has an unsuppo
```

Shut down the ROS network to disconnect.

```
rosshutdown
```

```
Shutting down global node /matlab_global_node_74999 with NodeURI http://bat1070102win64:52781/
Shutting down ROS master on http://172.30.68.186:50742.
```

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 **Simulation** tab by selecting **ROS Toolbox > ROS Network**.
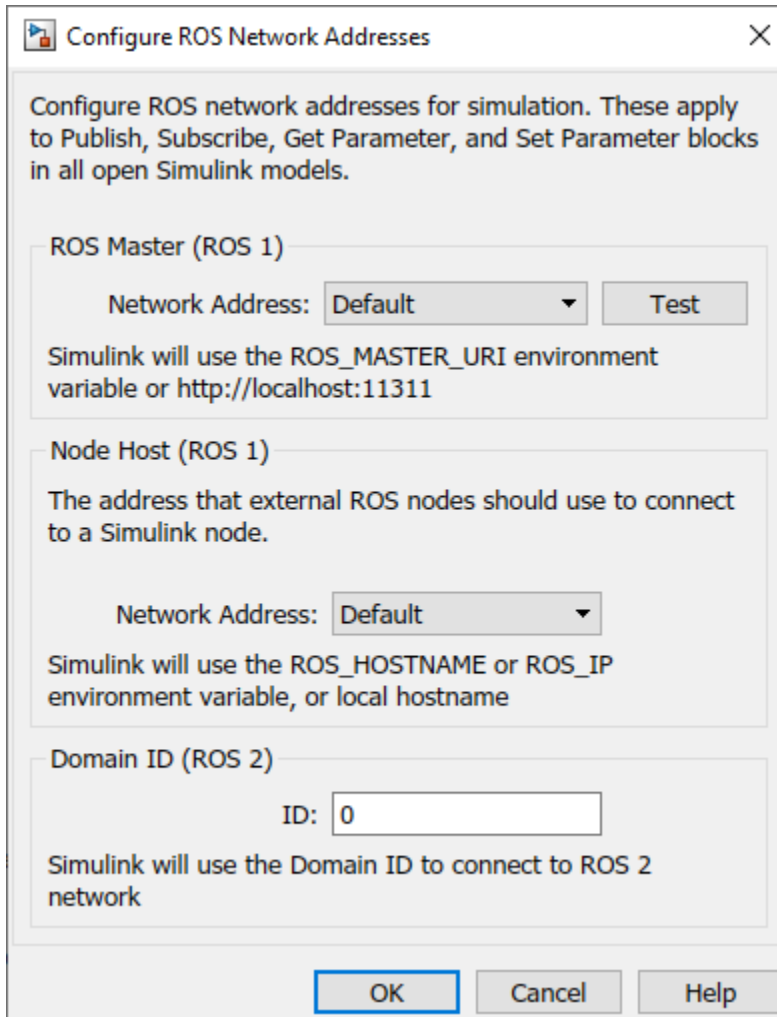
The **Network Address** parameter can be set to `Default` or `Custom`.

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" on page 1-2
- "Connect to a ROS-enabled Robot from Simulink®" on page 1-81

More About

- "ROS Simulink Interaction" on page 4-4
- "Select ROS Topics, Messages, and Parameters" on page 4-21
- "ROS Simulink Support and Limitations" on page 4-2

# Select ROS Topics, Messages, and Parameters

In this section...
"Select ROS Topics" on page 4-21
"Select ROS Message Types" on page 4-22
"Select ROS Parameter Names" on page 4-22

Select 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:
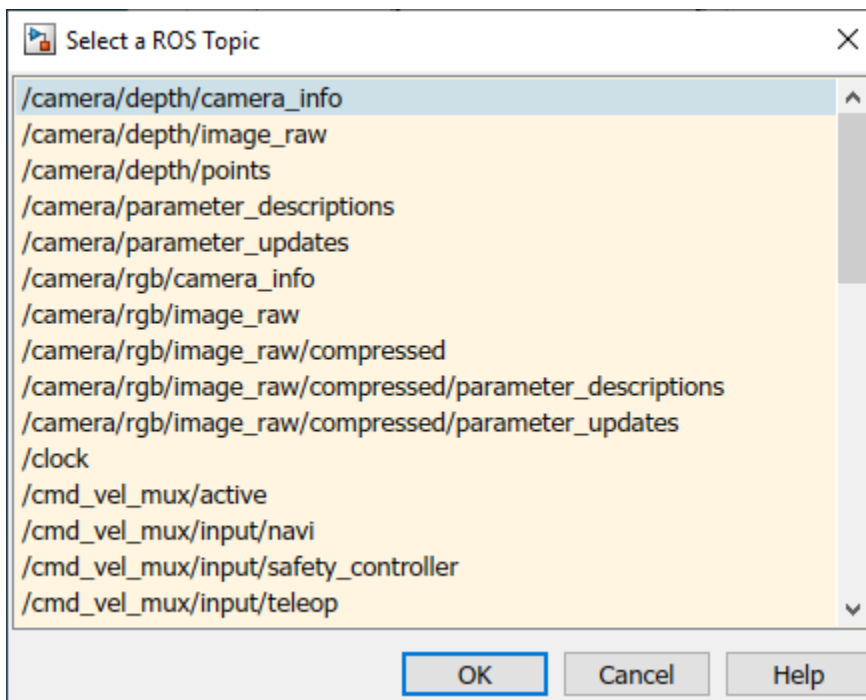
This dialog shows the list of topics available on the ROS master. Selecting a topic from the list automatically populates the **Topic** and **Message type** parameters for the corresponding block mask dialog. If the message type is not supported in MATLAB ROS, Simulink will throw an error. Once the topic 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.

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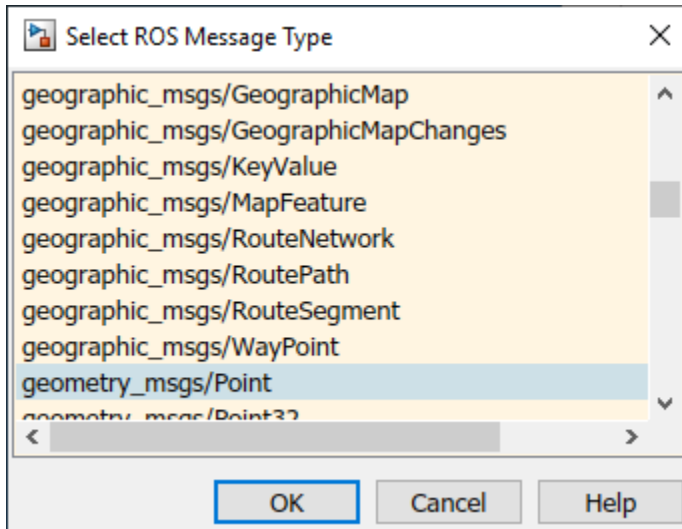
This is the list of all message types supported in MATLAB ROS including any custom message types. You can begin typing in the name of your desired message type or manually search through the list.

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.

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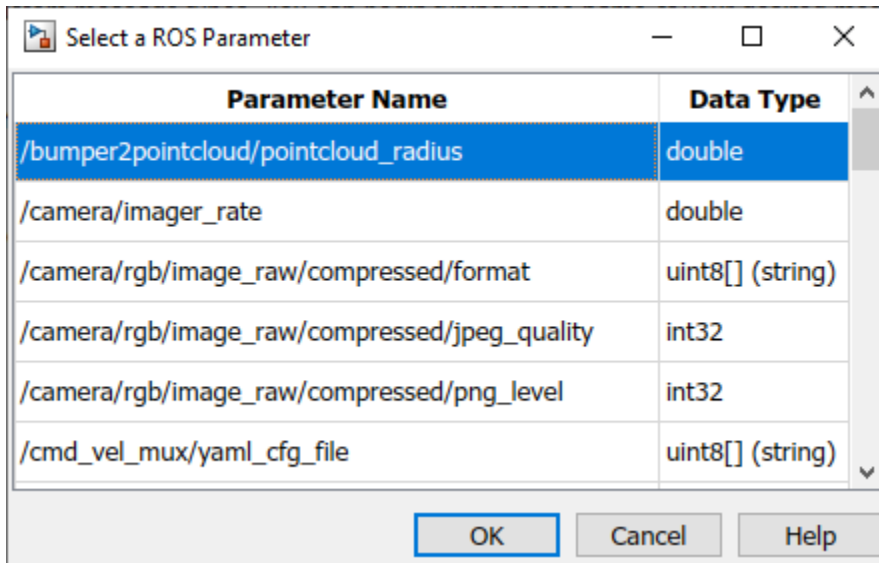
This is the list of parameters you can select from the ROS parameter server. The parameters that are grayed out have unsupported data types. Select a parameter name that is not grayed out and click **OK**. This should auto-fill the **Name** and **Data type** into the block parameters.

See Also

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

Related Examples

- "ROS Parameters in Simulink" on page 4-9
- "Manage Array Sizes for ROS Messages in Simulink" on page 4-24

Manage Array Sizes for ROS Messages in Simulink

A ROS message is represented as a bus signal. For more information on bus signals, see "Virtual Bus" (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. In the **Simulation** tab, select **ROS Toolbox > Variable Size Message** to manage array sizes. If your model uses ROS messages with variable-length arrays, the following dialog box opens. Otherwise, Simulink displays a message.
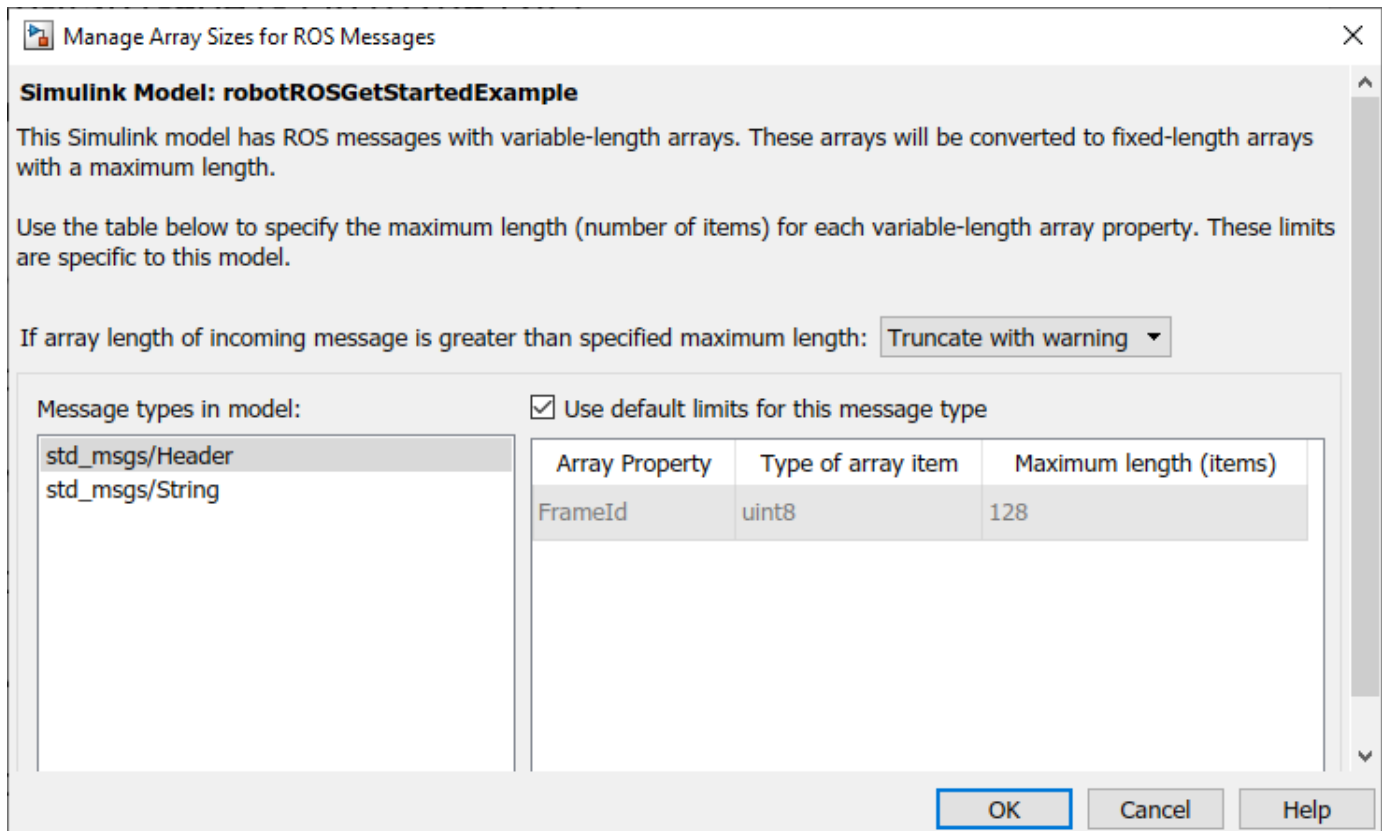
Because the message properties have a variable length, it is possible that they can be truncated if they exceed the maximum size set for that array. You have the option of `Truncate with warning` or `Truncate silently`. Either way, the simulation will run, but `Truncate with warning` displays a warning 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 "Get Started with ROS in Simulink®" on page 1-64.

See Also
Publish | Subscribe

Related Examples
- "Get Started with ROS in Simulink®" on page 1-64
- "ROS Simulink Support and Limitations" on page 4-2

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®" on page 1-92.

Prerequisites

- This example requires Simulink Coder™ and Embedded 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 ROS Toolbox examples. See "Get Started with Gazebo and a Simulated TurtleBot" on page 1-102 for instructions on how to install and use the virtual machine.
- Review the "Feedback Control of a ROS-Enabled Robot" on page 1-89 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" on page 1-89 example.

Open the proportional controller model.

edit robotROSFeedbackControlExample

Copy the entire model to a new blank Simulink model.

Delete the Simulation Rate Control block.

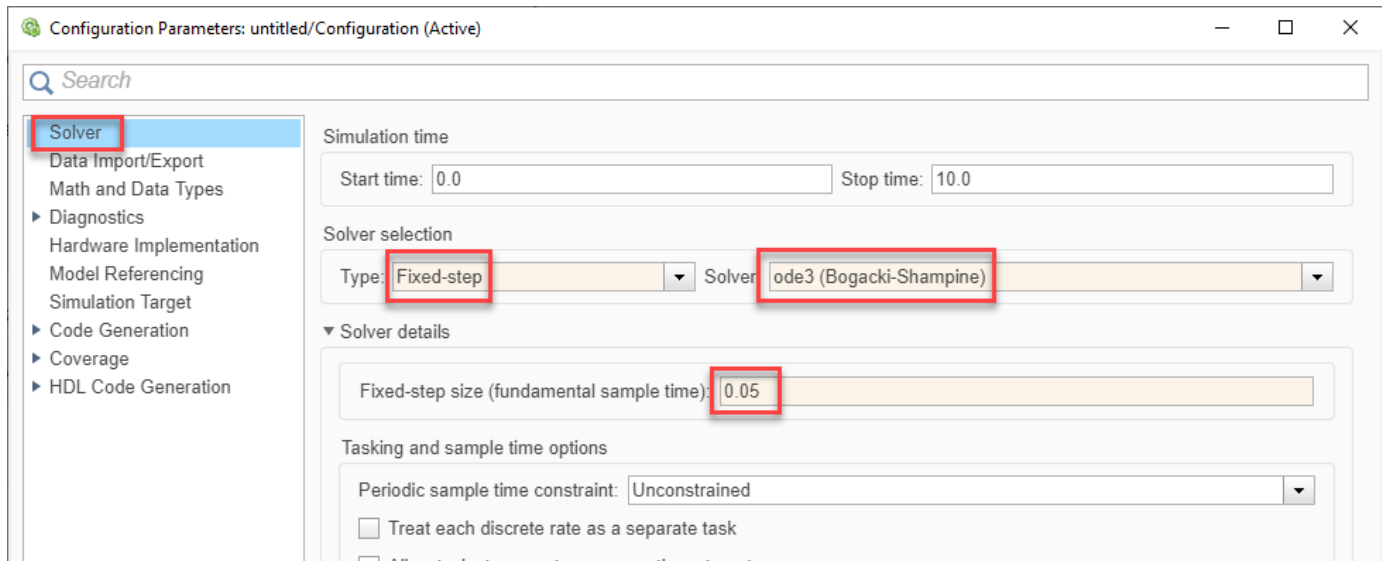Open the **Configuration Parameters** dialog. Under the **Simulation** tab, click **Model Settings**.

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**.
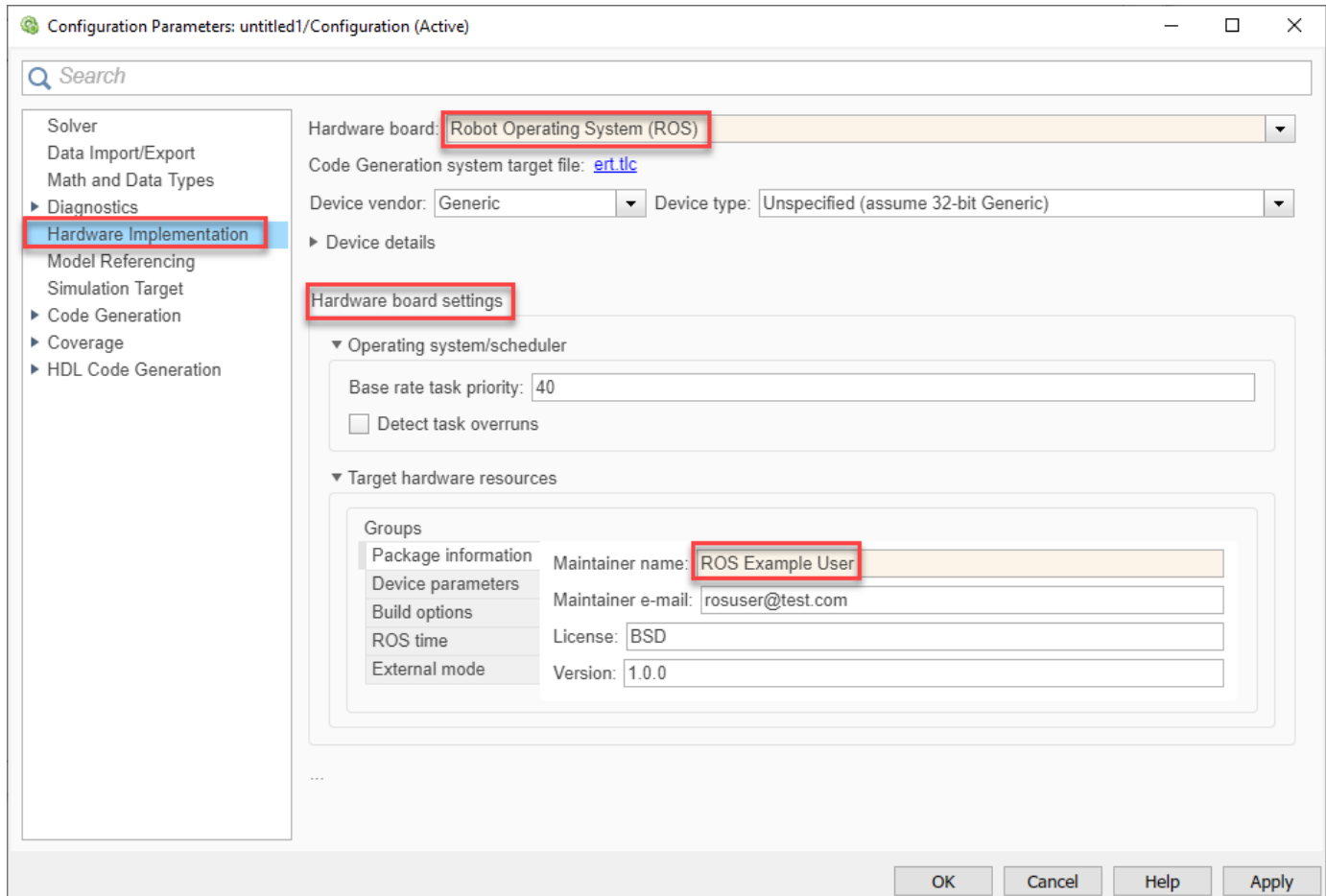
In the **Solver** pane of the **Configuration Parameters** dialog, ensure the **Type** is set to `Fixed-step`, the **Solver** is set to `ode3 (Bogacki-Shampine)`and the **Fixed-step size** is set to `0.05`. In generated code, the fixed-step size defines the actual time step that is used for the model update loop. See "Execution of Code Generated from a Model" (Simulink Coder) for more information.

In the **Code Generation** pane, ensure **variable-size signals** is enabled.

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. Under the **Simulation** tab, click **Model Settings**.

In the **Hardware Implementation** tab, under **Target hardware resources**, 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.
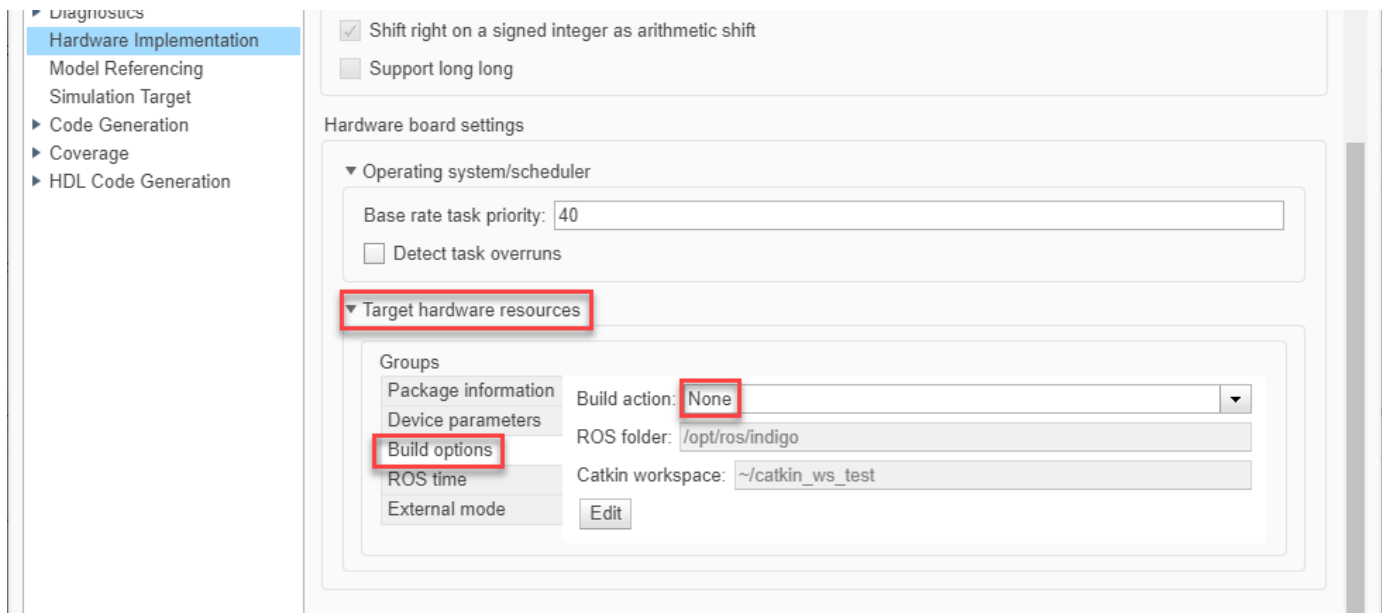
Generate and Deploy the Code

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.

Under the **C Code** tab, click **Generate Code** 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
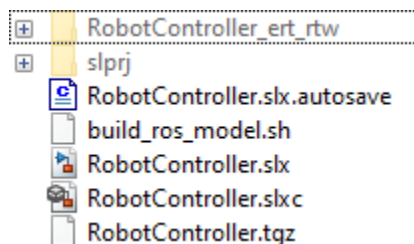
Manually transfer the two files to the target machine. If you connect to a ROS device using `rosdevice`, you can send files using `putFile`. Otherwise, this step assumes you are using the Linux virtual machine used for Robotics System Toolbox™ examples. The virtual machine is configured to accept SSH 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>
```

If that does not work, ensure that `build_ros_model.sh` is set up as an executable by entering the following command.

```
chmod +x build_ros_model.sh
```

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" on page 1-89

- "Generate a Standalone ROS Node from Simulink®" on page 1-92
- "Tune Parameters and View Signals on Deployed Robot Models Using External Mode" on page 4-32

Tune Parameters and View Signals on Deployed Robot Models Using External Mode

In this section...
"Set Up the Simulink Model" on page 4-32
"Deploy and Run the Model" on page 4-32
"Monitor Signals and Tune Parameters" on page 4-33

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" on page 1-89 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 **Modelling** tab on the toolstrip, select **Model Settings** 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 exiting the Bus Selector from the ROS subscriber that monitors the robot position. Open the XY Graph 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 example. In the **Simulations** tab select **ROS Network** to configure your ROS network address. 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.
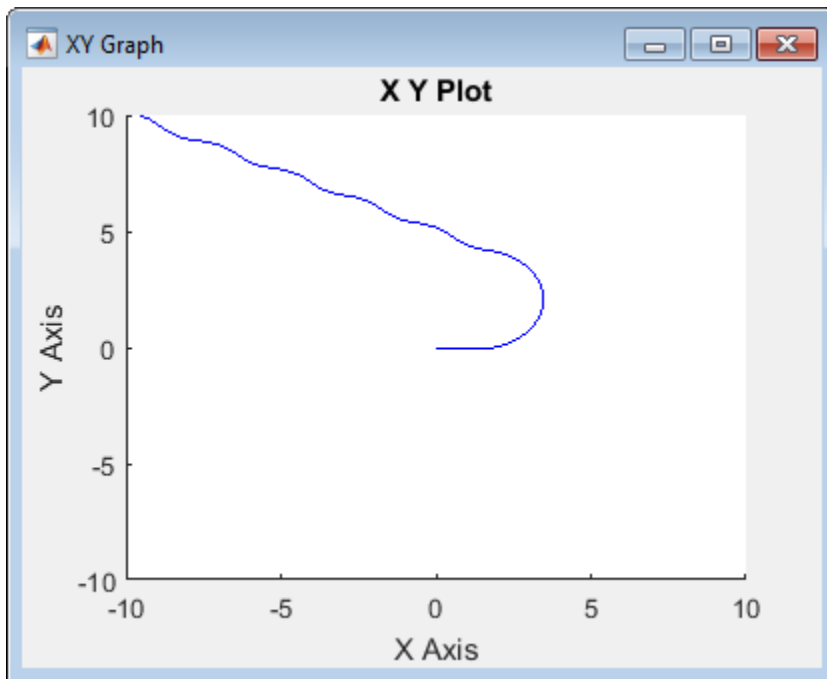
While the model is still running, you can also tune parameters. Open the `Proportional Controller` subsystem and change the Linear Velocity slider to `0.25`. Back in the main model, change the Desired Position constant block to a new position, `[0 -5]`. The robot drives to the new position slower.
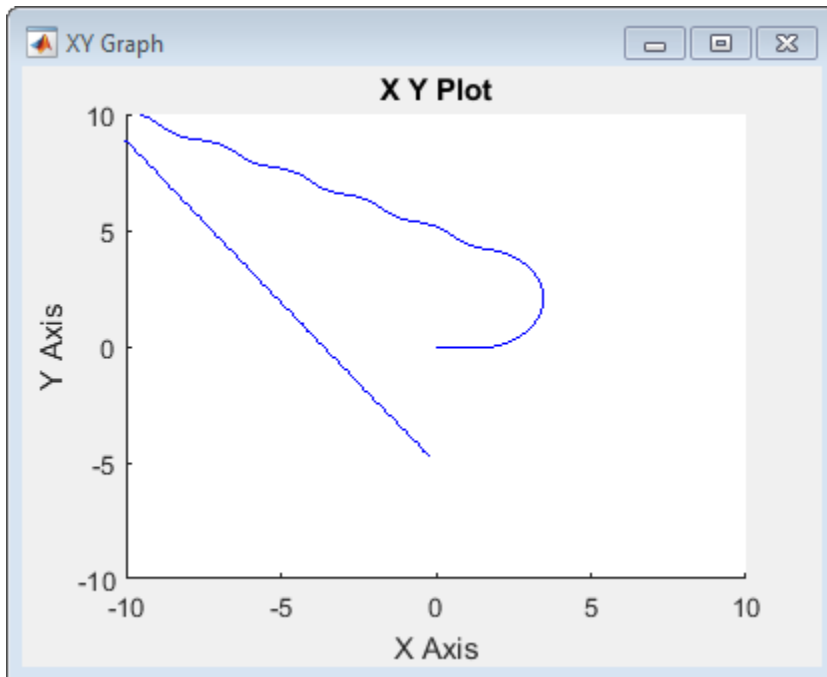
The lowered velocity reduces the wobble along the path. All these modifications were done while the model was deployed on the hardware.

See Also

Related Examples

- "Feedback Control of a ROS-Enabled Robot" on page 1-89
- "Enable External Mode for ROS Toolbox Models" on page 4-37
- "Generate a Standalone ROS Node from Simulink®" on page 1-92

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®" on page 1-92

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®" on page 1-92.

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   Under the **Modeling** tab, click **Model Settings** 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 4-38.

See Also
Current Time | Subscribe

Related Examples
- "Generate a Standalone ROS Node from Simulink®" on page 1-92
- "Overrun Detection with Deployed ROS Nodes" on page 4-38
- "Get Started with ROS in Simulink®" on page 1-64
- "Exchange Data with ROS Publishers and Subscribers" on page 1-20

Enable External Mode for ROS 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 ROS Toolbox, deployed models are ROS nodes running on the target hardware that communicates with Simulink over TCP/IP.

To use external mode with ROS 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 4-35 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 **Apps** tab 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 "External Mode Simulations for Parameter Tuning and Signal Monitoring" (Simulink Coder).

See Also

Related Examples

- "Generate a Standalone ROS Node from Simulink®" on page 1-92
- "Tune Parameters and View Signals on Deployed Robot Models Using External Mode" on page 4-32
- "External Mode Simulations for Parameter Tuning and Signal Monitoring" (Simulink Coder)

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®" on page 1-92.

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 Under the **Modeling** tab, select **Model Settings** for your model.
2 Under **Hardware Implementation**, set **Hardware board** to Robot Operating System (ROS).
3 Under **Hardware board settings** > **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 **Sample time** parameter in all of your data source blocks from 0.1 to 0.2.

See Also
Current Time | Subscribe

Related Examples

Convert a ROS Pose Message to a Homogeneous Transformation

This model subscribes to a `Pose` message on the ROS network and converts it to a homogeneous transmation. 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
```

```
Launching ROS Core...
..........Done in 19.101 seconds.
Initializing ROS master on http://172.30.68.186:59168.
Initializing global node /matlab_global_node_25466 with NodeURI http://bat1070102win64:60168/
```

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

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.
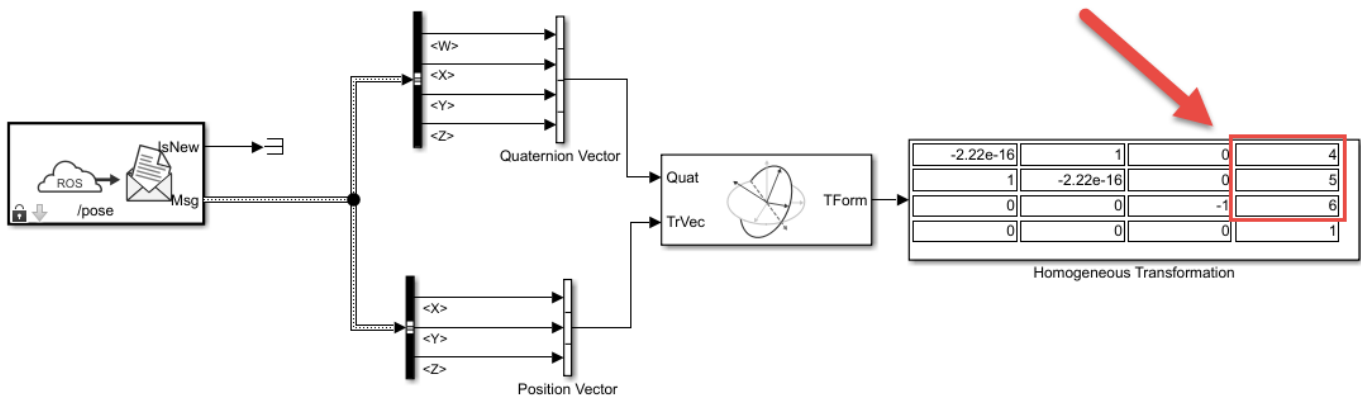
Copyright 2018 The MathWorks, Inc.

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_25466 with NodeURI http://bat1070102win64:60168/
Shutting down ROS master on http://172.30.68.186:59168.

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 Toolbox® and Robotics System Toolbox®.

Start a ROS network.

```
rosinit
```

```
Initializing ROS master on http://ah-rhosea:11311/.
Initializing global node /matlab_global_node_07639 with NodeURI http://ah-rhosea:51851/
```

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.

```
exampleHelperROSLoadPCloud
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. Under the **Simulation** tab, select **ROS Toolbox > Variable Size Arrays** and 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.
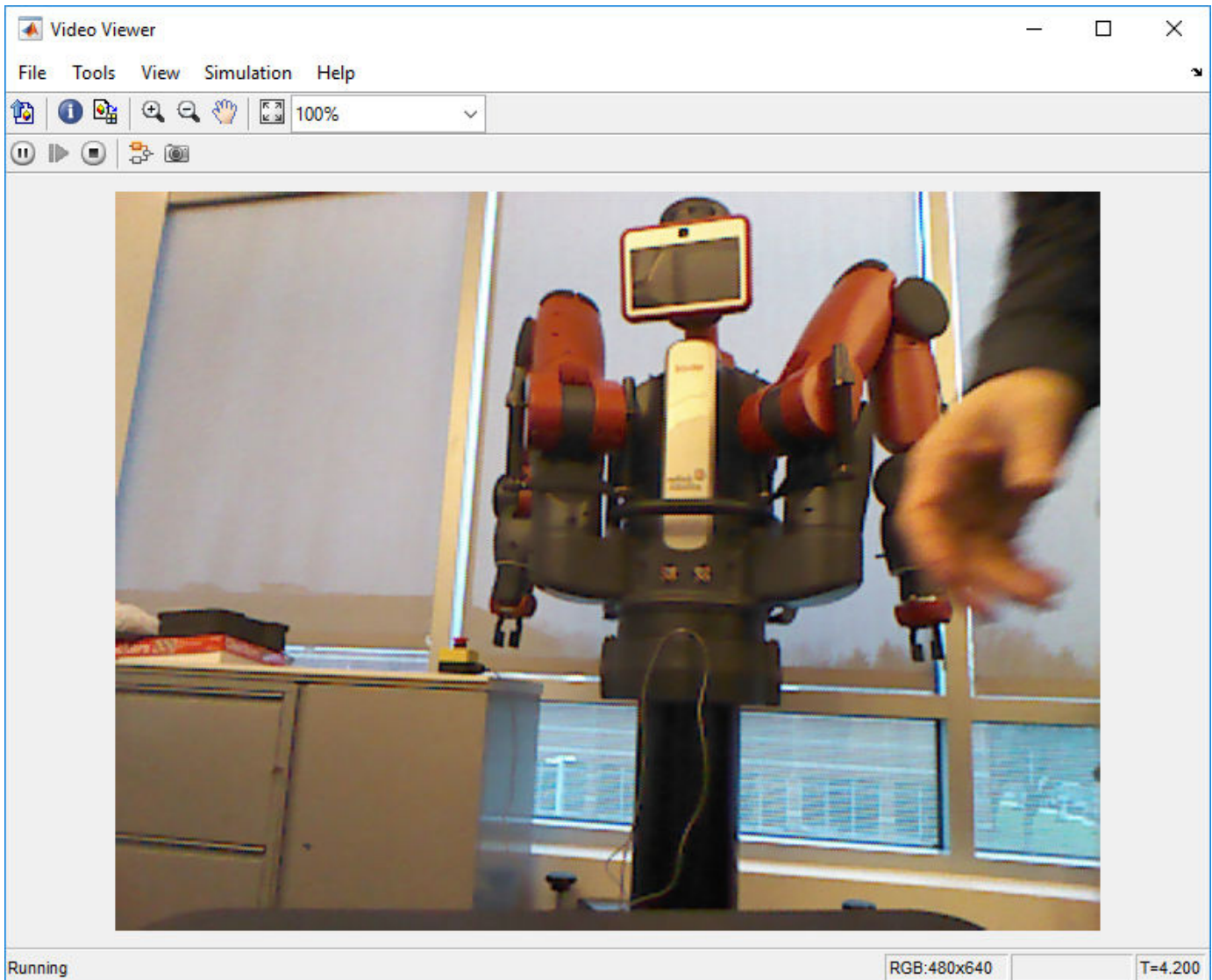
```
open_system('read_point_cloud_example_model.slx')
```
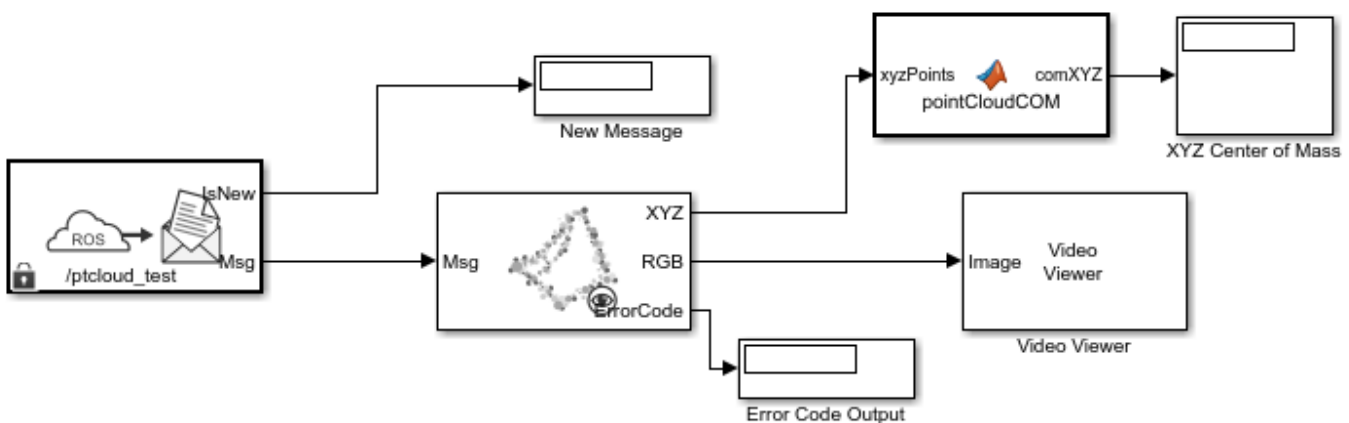

Copyright 2018 The MathWorks, Inc.
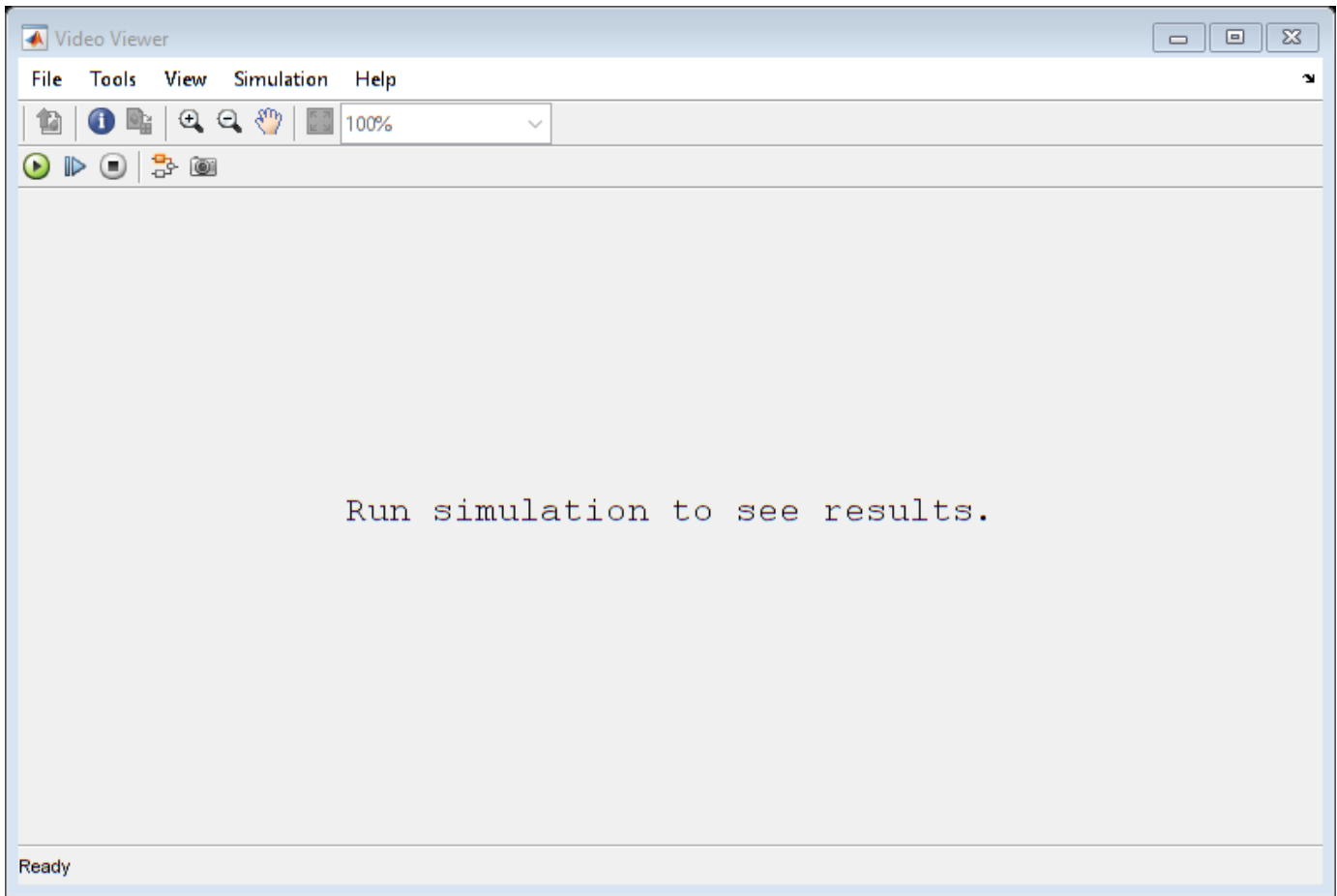
Run the model. The `Video Viewer` shows the sample point cloud as an image. The output center of mass is `[-0.2869 -0.0805 2.232]` for this point cloud.

Stop the simulation and shut down the ROS network.

```
rosshutdown
```

Shutting down global node /matlab_global_node_07639 with NodeURI http://ah-rhosea:51851/
Shutting down ROS master on http://ah-rhosea:11311/.

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);
```

```matlab
% 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
```

Read A ROS Image Message In Simulink®

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

Start a ROS network.

```
rosinit
```

```
Initializing ROS master on http://ah-rhosea:11311/.
Initializing global node /matlab_global_node_45601 with NodeURI http://ah-rhosea:49292/
```

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.

```
imgcell = load('imgdata.mat','img');
img = imgcell.img;
pub = rospublisher('/image_test','sensor_msgs/Image');
send(pub,img)
```

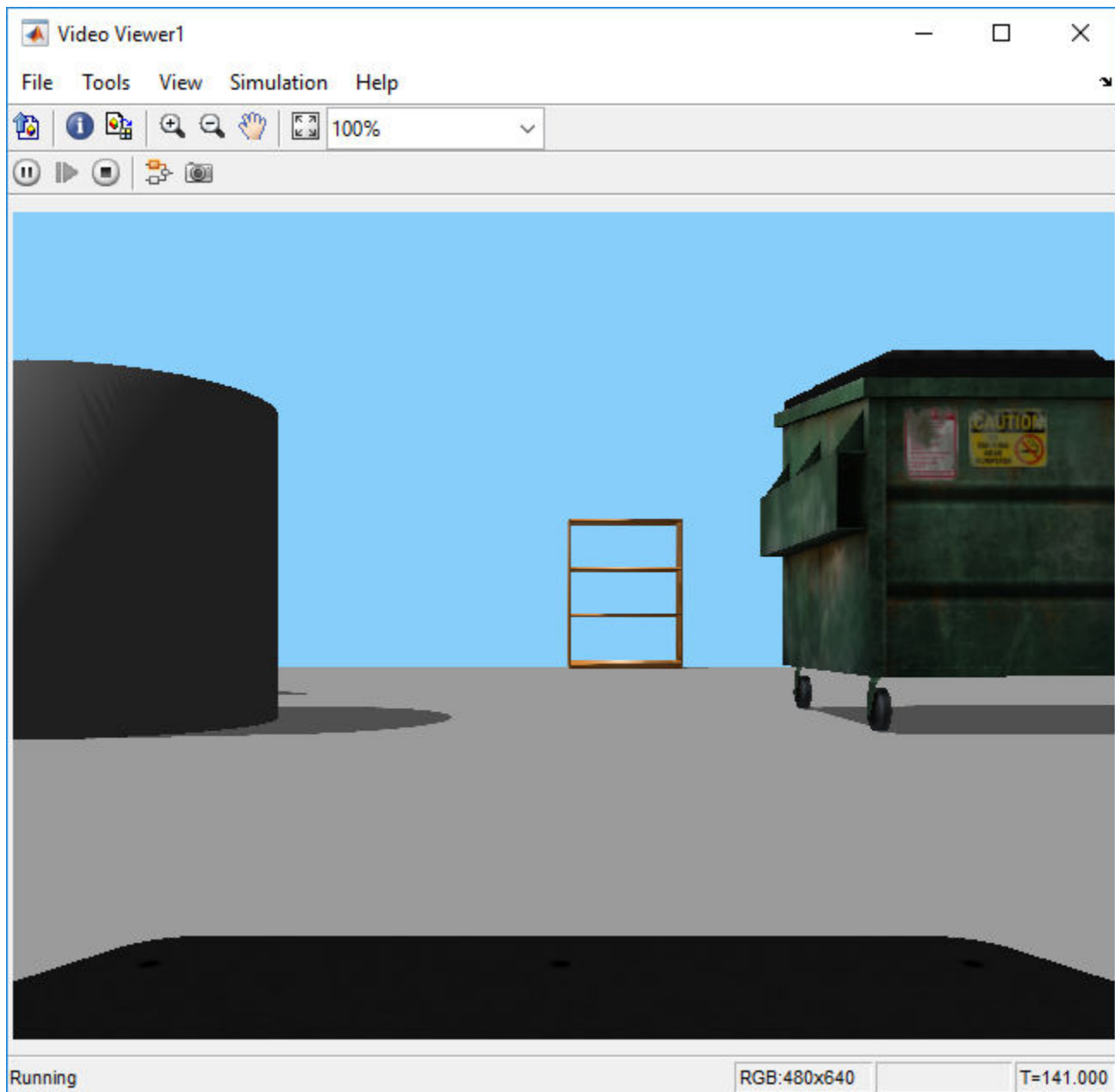Open the Simulink® model for subscribing to the ROS message and reading in the image from the ROS.

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')
```


Copyright 2018 The MathWorks, Inc.
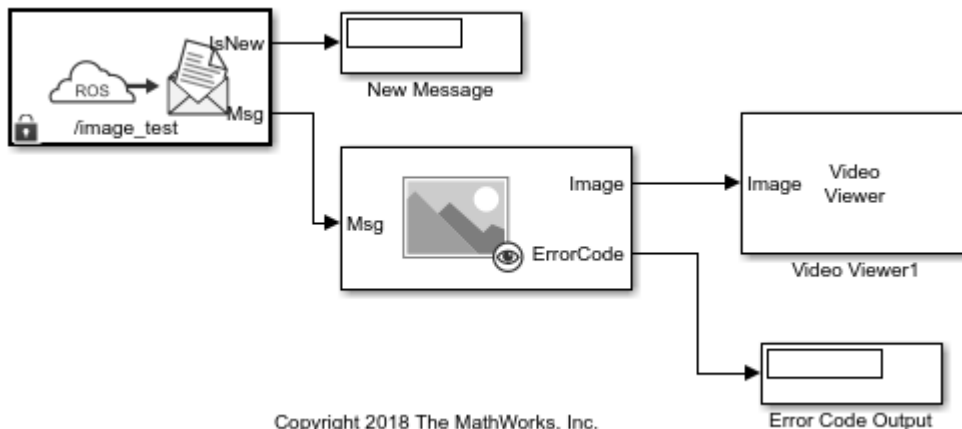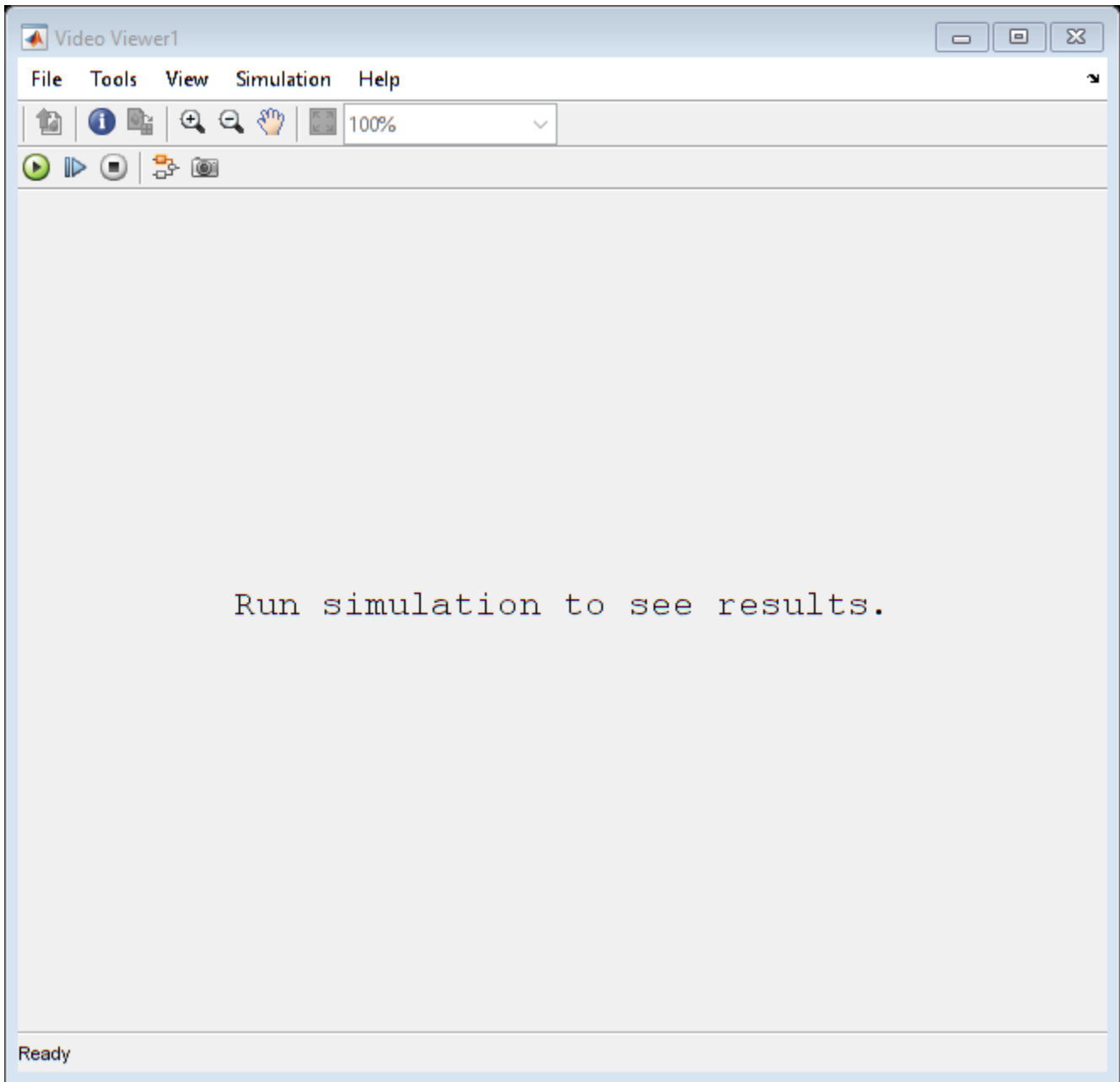
Run the model. The `Video Viewer` shows the sample image.

Stop the simulation and shut down the ROS network.

```
rosshutdown
```

```
Shutting down global node /matlab_global_node_45601 with NodeURI http://ah-rhosea:49292/
Shutting down ROS master on http://ah-rhosea:11311/.
```