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	

Trademarks

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

Patents

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

Contents

ROS 2 Featured Examples

2

ROS Topics

3

1

ROS Featured Examples

Sign Following Robot with Time Synchronization Using ROS and Gazebo Co-Simulation

This example shows how to use Simulink® to enable synchronized simulation between ROS and the Gazebo robot simulator using the Gazebo Pacer (Robotics System Toolbox) block.

This example builds upon the "Sign Following Robot with ROS in Simulink" on page 2-115 example, which does not support synchronized simulation. Unsynchronized simulation can result in sensor data loss or delayed messages and simulation time mismatch. Gazebo Co-Simulation is ROS-independent, which enables it to synchronize the simulation times between Gazebo and Simulink when using the Gazebo Pacer block. For more details about Gazebo Co-Simulation, see "Gazebo Simulation for Robotics System Toolbox" (Robotics System Toolbox).

Start Robot Simulator

To follow along with this example:

1 Download and install the virtual machine with ROS and Gazebo. See **Setup Gazebo Simulation Environment** in "Perform Co-Simulation between Simulink and Gazebo" (Robotics System Toolbox).
2 Start the Ubuntu® virtual machine (VM) desktop.
3 On the Ubuntu desktop, click **Gazebo Sign Follower ROS Co-Sim** to start the Gazebo world built for this example.

Open Simulink Model

The Simulink model used in this example is based on the Simulink model used in the "Sign Following Robot with ROS in Simulink" on page 2-115 example, with the addition of the Gazebo Pacer block.

Open the Simulink model.

```
open_system("signFollowingRobotROSCoSim.slx")
```

Sign Following Robot with ROS Co-Sim
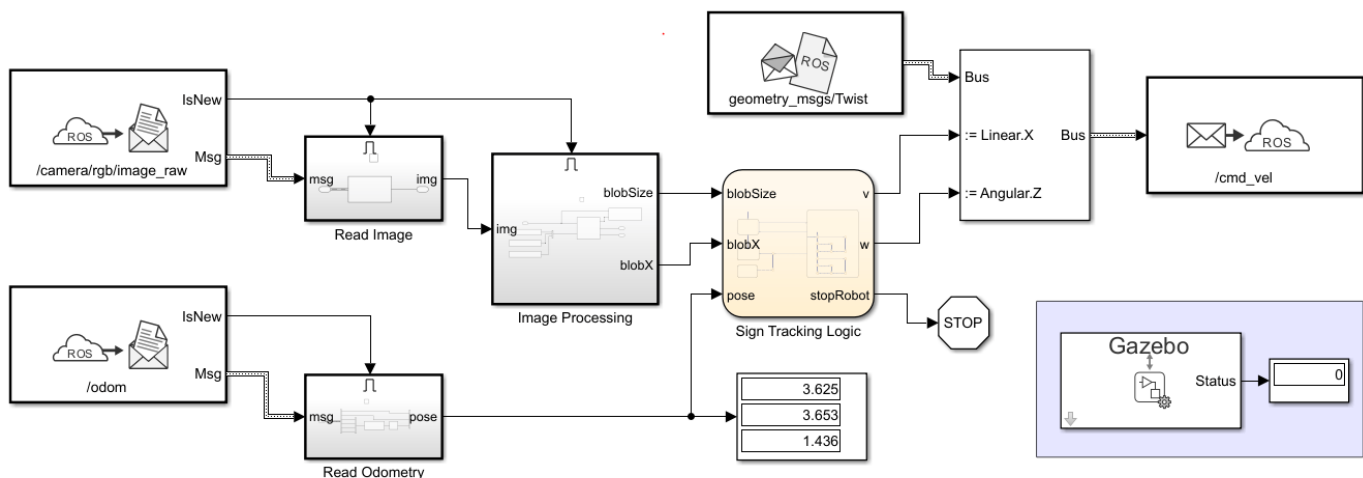
You can add the Gazebo Pacer block to an existing ROS–Gazebo Simulink model, as shown in the model diagram, to enable synchronized simulation.

Configure Gazebo Pacer Block

Before simulating the model, configure Gazebo Co-Simulation using the **Gazebo Pacer** block.

```
open_system("signFollowingRobotROSCoSim/Gazebo Pacer")
```

Set **Reset behavior** to Reset Gazebo simulation time and scene.
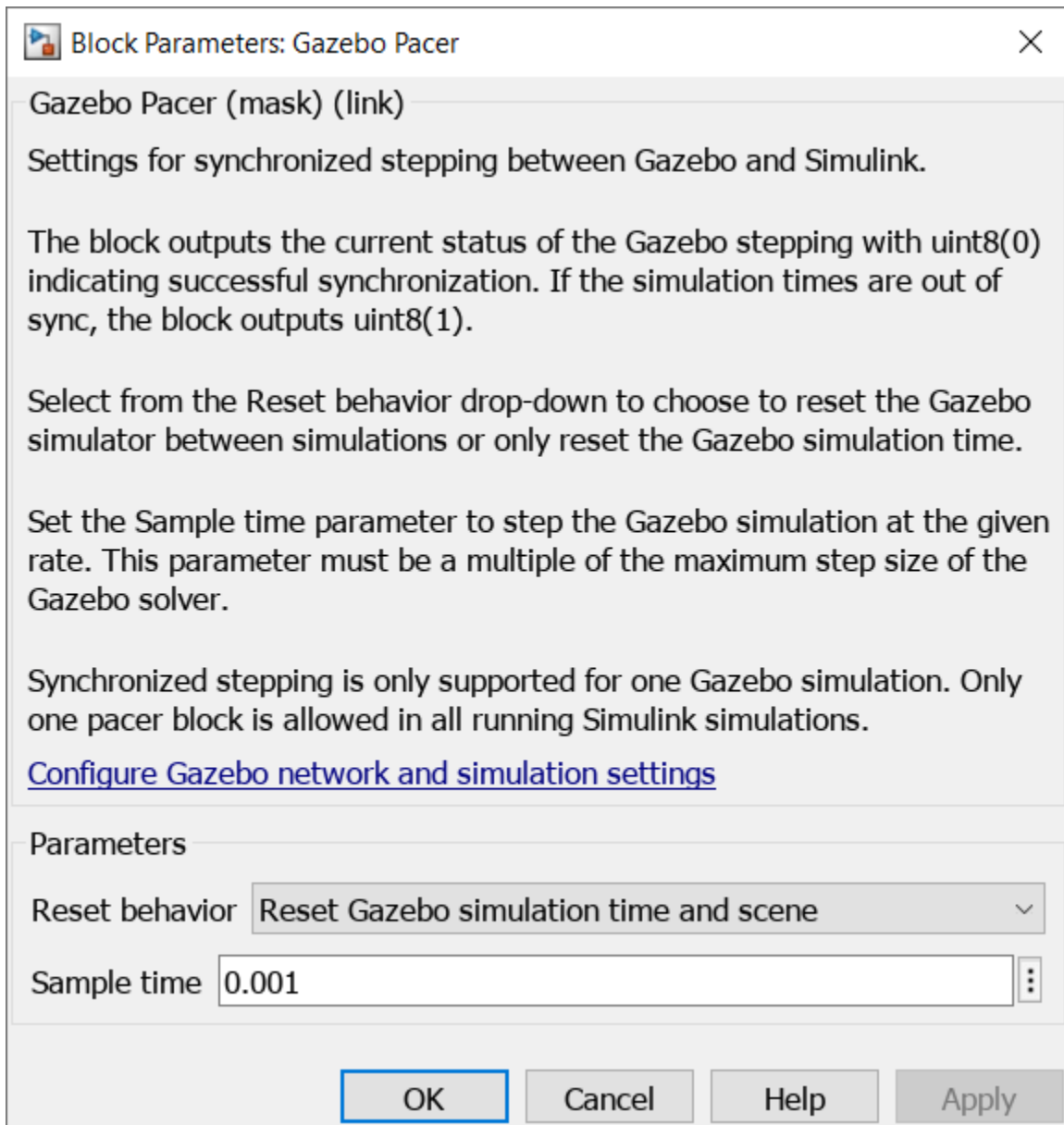
To configure the connection to the Gazebo simulation, click **Configure Gazebo network and simulation settings**.

1. Set **Network Address** to `Custom`.
2. Enter the **Hostname/IP Address** of the VM.
3. The default **Port** number for Gazebo is `14581`. Check that the **Port** value is the same as in the `<portnumber>` tag in the world file.
4. Set **Response timeout** to `10` seconds.
5. Click **Test** to test the connection to the running Gazebo simulator.
6. Click **OK** to apply your changes.
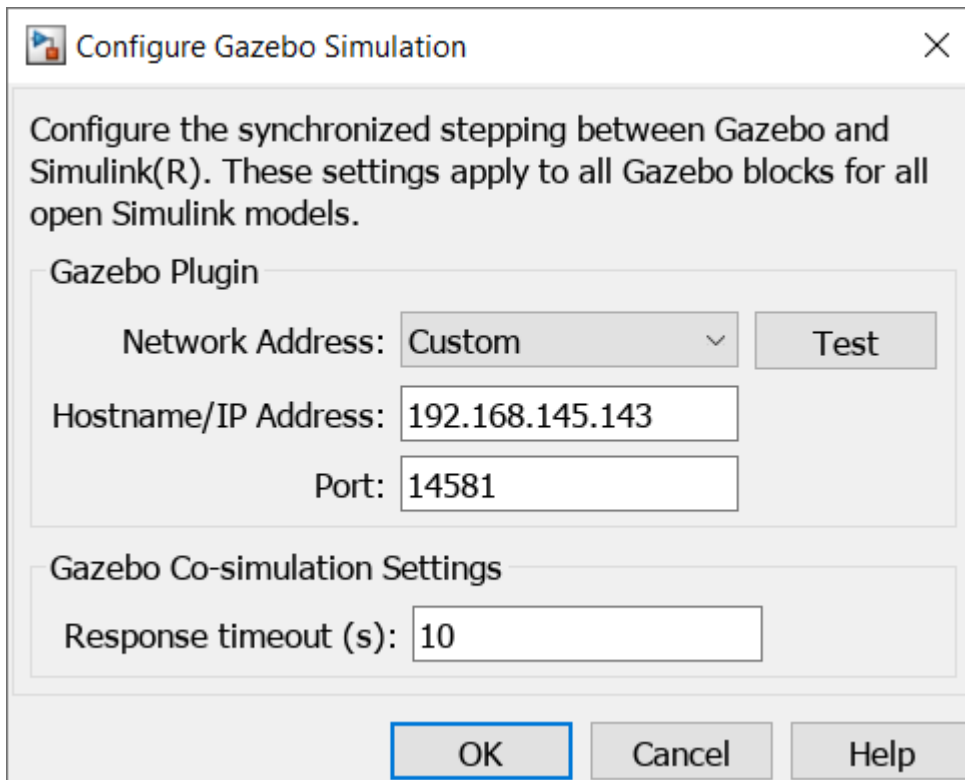
Configure ROS Network

To configure the network settings for ROS:

1. From the **Simulation** tab, in the **Prepare** group, select **ROS Network**.
2. Specify the IP address and port number of the ROS master in Gazebo. For this example, the ROS master in Gazebo is `192.168.145.143:11311`. Set **Hostname/IP Address** to `192.168.145.143` and **Port** to `11311`.
3. Click **OK** to apply your changes.

Configure ROS Network Addresses   ✕

Configure ROS network addresses for simulation. These apply to Publish, Subscribe, Get Parameter, and Set Parameter blocks in all open Simulink models.

ROS Master (ROS)

Network Address: `Custom` ⌄   `Test`

Hostname/IP Address: `192.168.145.143`

Port: `11311`

Node Host (ROS)

The address that external ROS nodes should use to connect to a Simulink node.

Network Address: `Default` ⌄

Simulink will use the ROS_HOSTNAME or ROS_IP environment variable, or local hostname

Domain ID (ROS 2)

ID: `0`

Simulink will use the Domain ID to connect to ROS 2 network

ROS Middleware (ROS 2)

RMW Implementation: `rmw_fastrtps_cpp`

Simulink will use the RMW Implementation to switch ROS Middleware

`OK`   `Cancel`   `Help`

Run Model

Run the model and observe this simulation behavior, in addition to the behavior of the robot in the robot simulator from the "Sign Following Robot with ROS in Simulink" on page 2-115 example.

- Simulink Time and Gazebo Time are synchronized.
- The image readings from the camera are synchronized with the movement of the robot.
- When you pause the Simulink model, the Gazebo simulation preserves the pause state, and the robot maintains its current position.

Enable Synchronization in Other ROS–Gazebo Examples in VM

To enable synchronization in other ROS–Gazebo examples, follow these steps.

Set Up Gazebo Plugin

- In the VM for your ROS–Gazebo co-simulation, locate the Gazebo plugin required for synchronized simulation in `/home/user/src/GazeboPlugin`.
- To rebuild or replace the plugin, follow the Gazebo plugin manual installation steps in "Perform Co-Simulation between Simulink and Gazebo" (Robotics System Toolbox).

Update Gazebo World File

- Locate the shared library file of the synchronization plugin in `/home/user/src/GazeboPlugin/export/lib/libGazeboCoSimPlugin.so`.
- To activate the Gazebo plugin in any Gazebo world, add the shared library file details to the world file.
- For the "Sign Following Robot with ROS in Simulink" on page 2-115 example, the world file `mw_vision_world_newstopsign.world` is located in `/home/user/catkin_ws/src/mw_vision_example/worlds`.
- Open the `mw_vision_world_newstopsign.world` and add these shared library details.

```
<plugin name="GazeboPlugin" filename="/home/user/src/GazeboPlugin/export/lib/libGazeboCoSimPlugi
```

- The code snippet of the updated world file shows the Gazebo plugin tag.

```
    </link>
    <pose frame=''>3.69568 4.31208 0 0 -0 0</pose>
  </model>
  <plugin name="GazeboPlugin" filename="/home/user/src/GazeboPlugin/export/lib/libGazeboCoSimPlugin.so"><portNumber>14581</portNumber></plugin>
  </world>
</sdf>
```

Pick-and-Place Workflow in Unity 3D Using ROS

This example shows how to set up an end-to-end pick-and-place workflow for a robotic manipulator like the KINOVA® Gen3, and simulate the robot in the Unity® game engine.

Overview

This example identifies and places objects into a bin using a KINOVA Gen3 manipulator. The example uses tools from three toolboxes:

- **Robotics System Toolbox™** — Models and simulates the manipulator.
- **Stateflow®** — Schedules the high-level tasks in the example, and steps the simulation from task to task.
- **ROS Toolbox™** — Connects MATLAB® to Unity.

This example builds on key concepts from these related examples:

- "Plan and Execute Task- and Joint-Space Trajectories Using KINOVA Gen3 Manipulator" (Robotics System Toolbox) — Shows how to generate and simulate interpolated joint trajectories to move from an initial to a desired end-effector pose.
- "Pick-and-Place Workflow Using Stateflow for MATLAB" (Robotics System Toolbox) — Shows how to set up an end-to-end pick-and-place workflow for a robotic manipulator like the KINOVA Gen3.
- "Get Started with Gazebo and Simulated TurtleBot" on page 1-206 — Shows how to set up the connection between MATLAB and Gazebo.
- "Pick-and-Place Workflow in Gazebo Using ROS" (Robotics System Toolbox) — Shows how to set up an end-to-end pick-and-place workflow for a robotic manipulator like the KINOVA Gen3, and simulate the robot in the Gazebo physics simulator.
- "Model and Control Manipulator Arm with Robotics and Simscape" (Robotics System Toolbox) — Shows how to design robot algorithms in Simulink®, and then simulate the action in a test environment using Simscape™.

Pick-and-Place Workflow
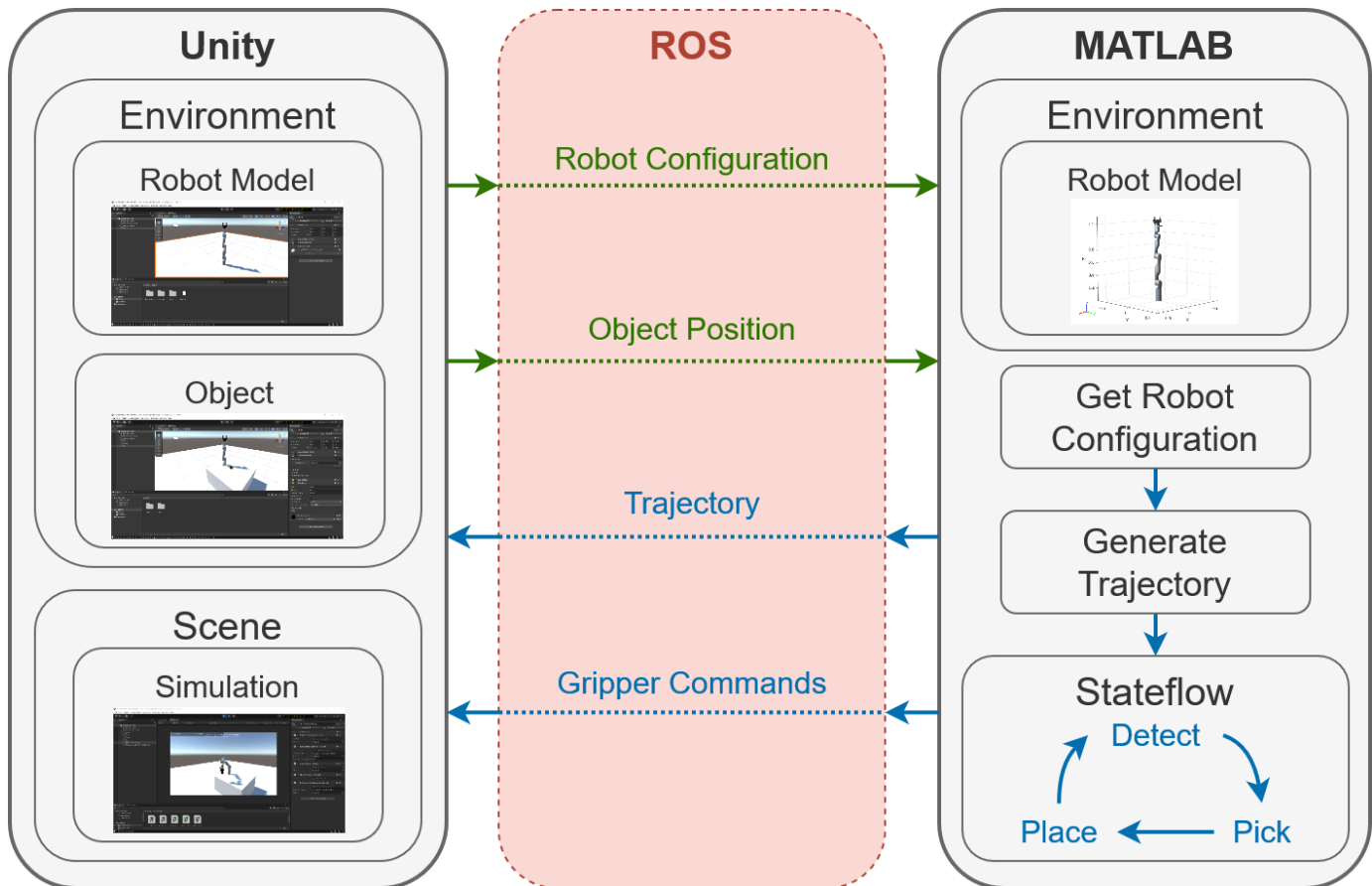
Install Unity and Create New Project

Follow these steps to install Unity and create a new project:

1 Download and install Unity Hub.

2 Launch **Unity Hub** and sign in with your account.

3 Install **Unity Editor** from Unity Hub. This example has been tested with Unity Editor version 2021.3.6f1 LTS.

4 In Unity Hub, select **Create project**. Unity Editor opens with a new project.
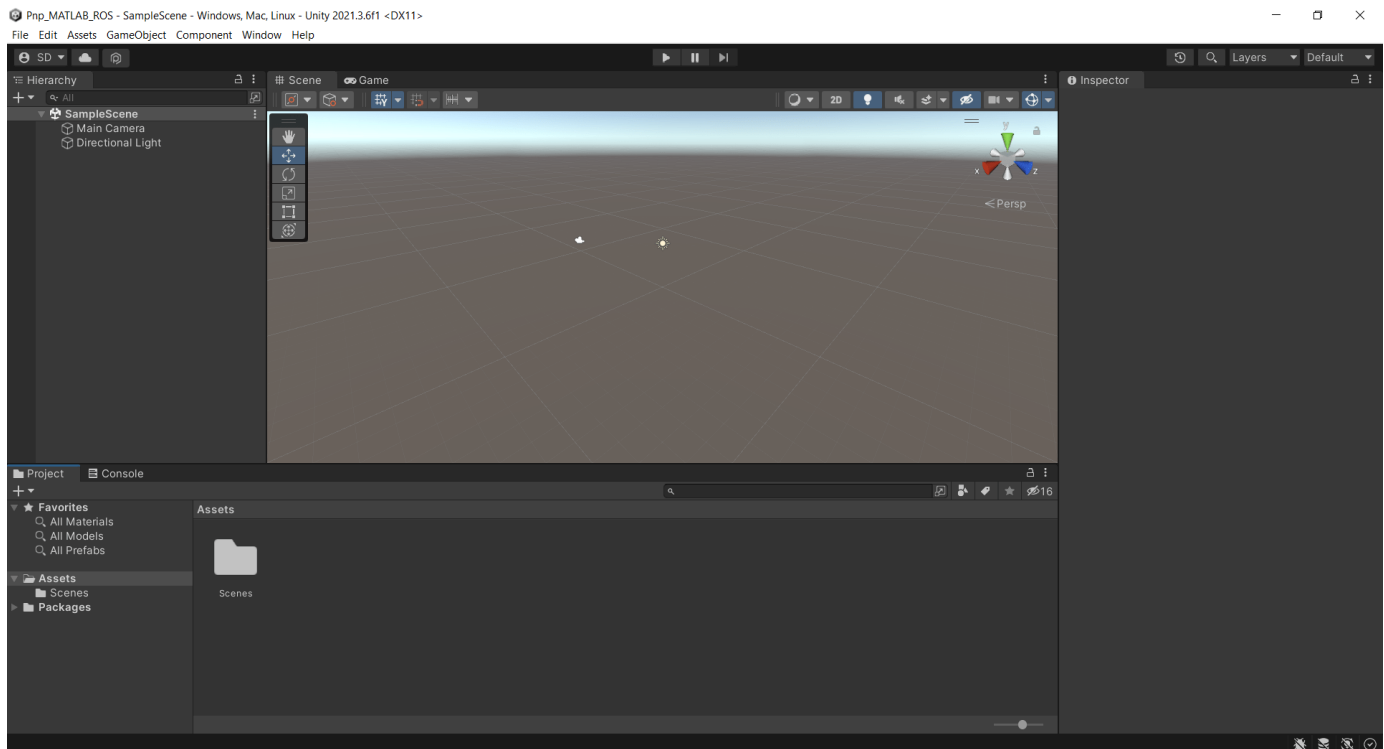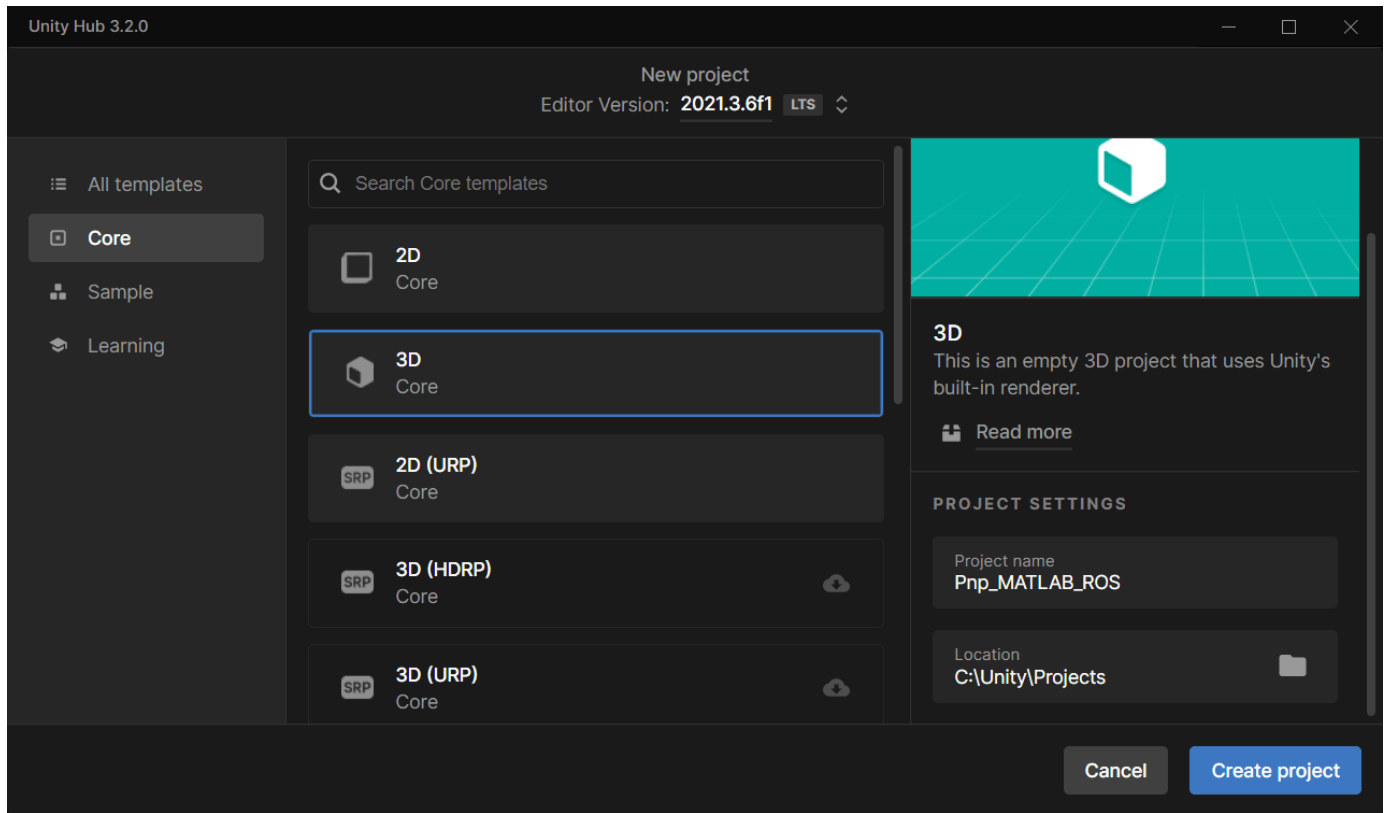
Install Unity Packages

Install these packages using the package manager in Unity.

- **URDF Importer** — To import the robot model from URDF files. Download this package as a ZIP file from the URDF Importer GitHub repository. This example has been tested with URDF Importer version 0.5.2.

- **ROS TCP Connector** — To connect Unity with the ROS server. Download this package as a ZIP file from the ROS TCP Connector GitHub repository. This example has been tested with ROS TCP Connector version 0.7.0.

Follow these steps to install the packages:

1 Extract the ZIP files containing the packages.

2 In Unity Editor from the menu, select **Window > Package Manager** to open the **Package Manager** window.

3 In the **Package Manager** window, click the **+** sign and select **Add package from disk**.

4 Browse and open the `package.json` files for the two packages. The screenshot indicates the locations of the `package.json` files each of the packages.
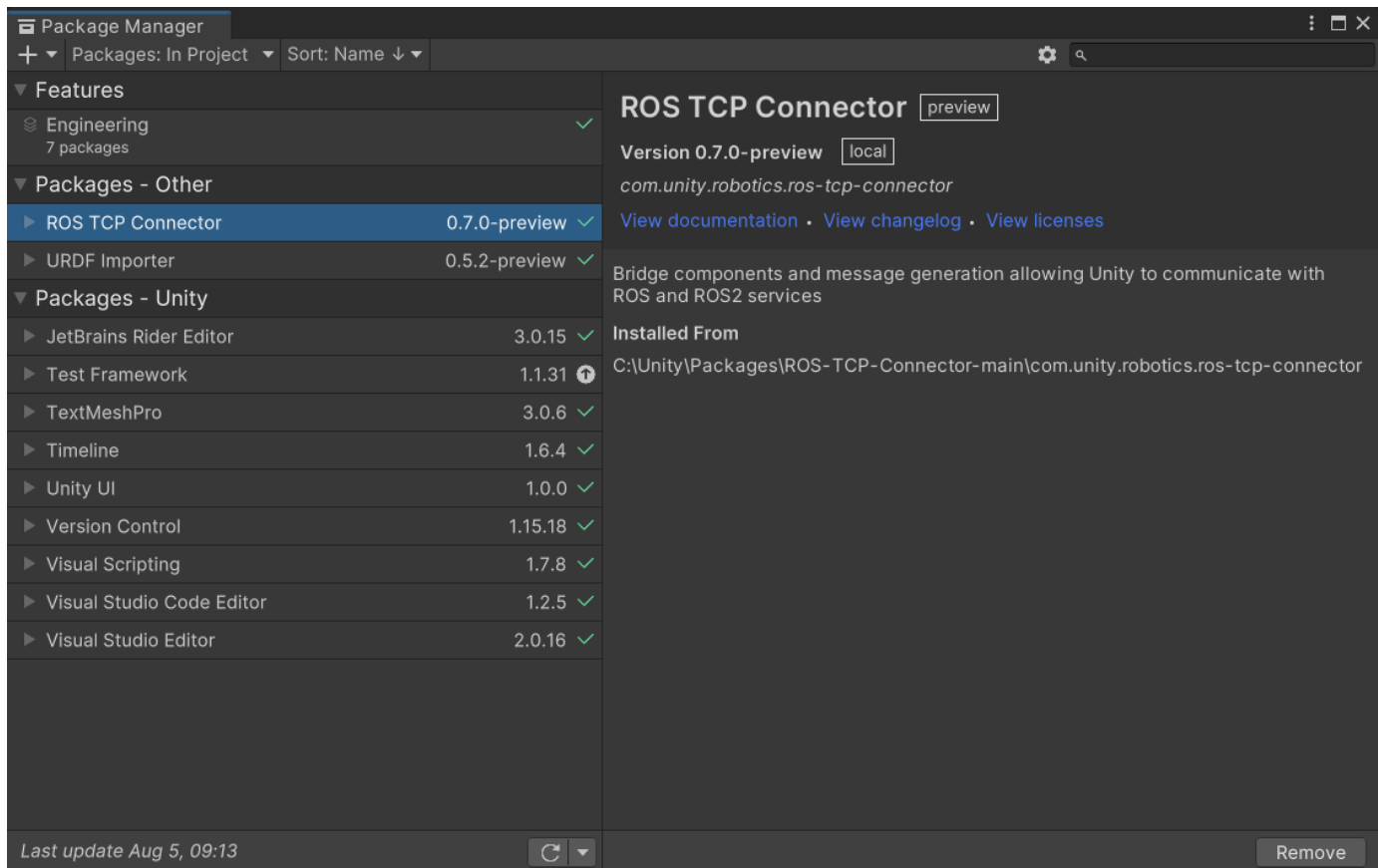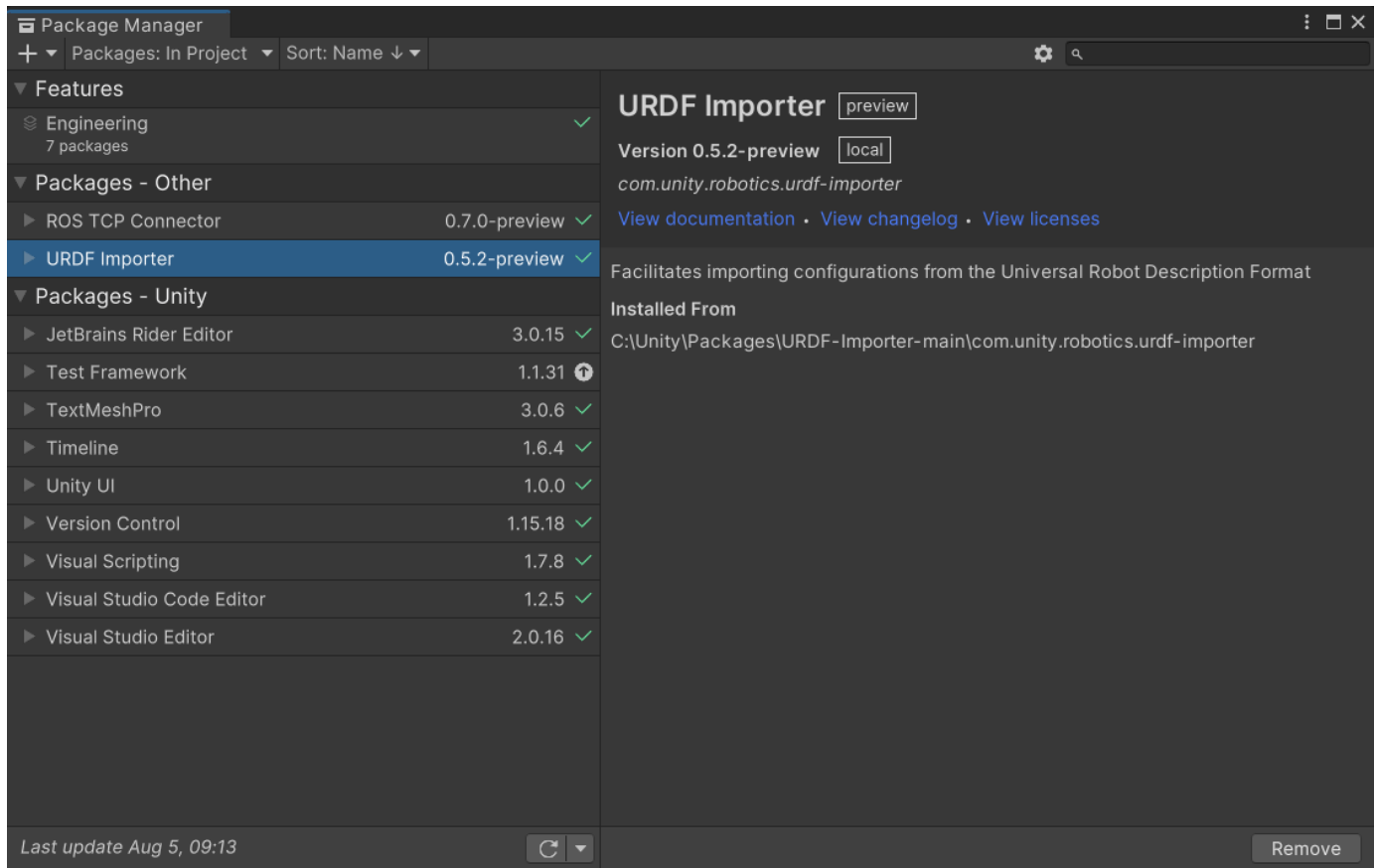
Import Robot Model into Unity

The `Robot.zip` file included with this example contains the URDF file of the KINOVA Gen3 robot with a custom gripper attached to it. The file also contains the relevant mesh files for the robot model.

To import the URDF model into Unity:

1 Extract the contents of the `Robot.zip` file to the `Assets` folder in the Unity project directory.

2 Right-click the URDF file in the **Project** pane and select **Import Robot from Selected URDF file**.

3 In the **URDF Import Settings** window, click **Import URDF**.
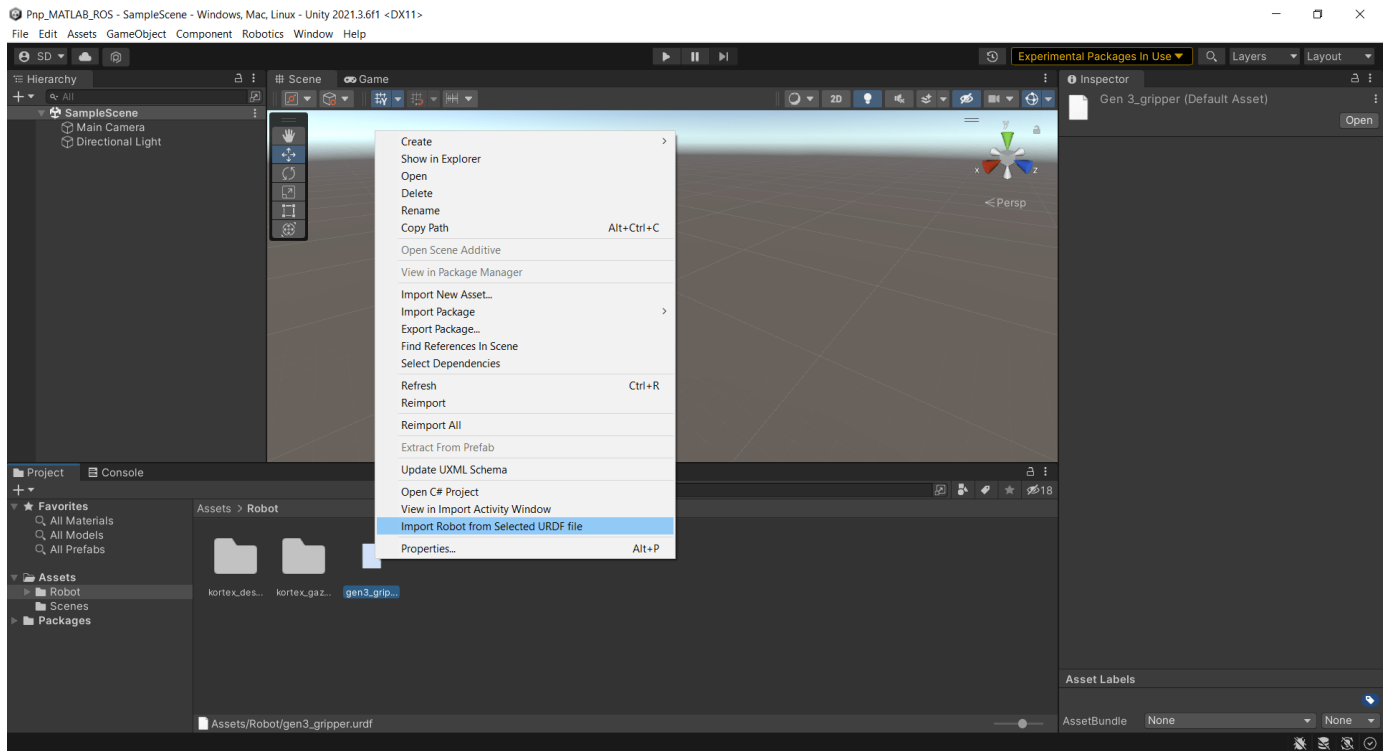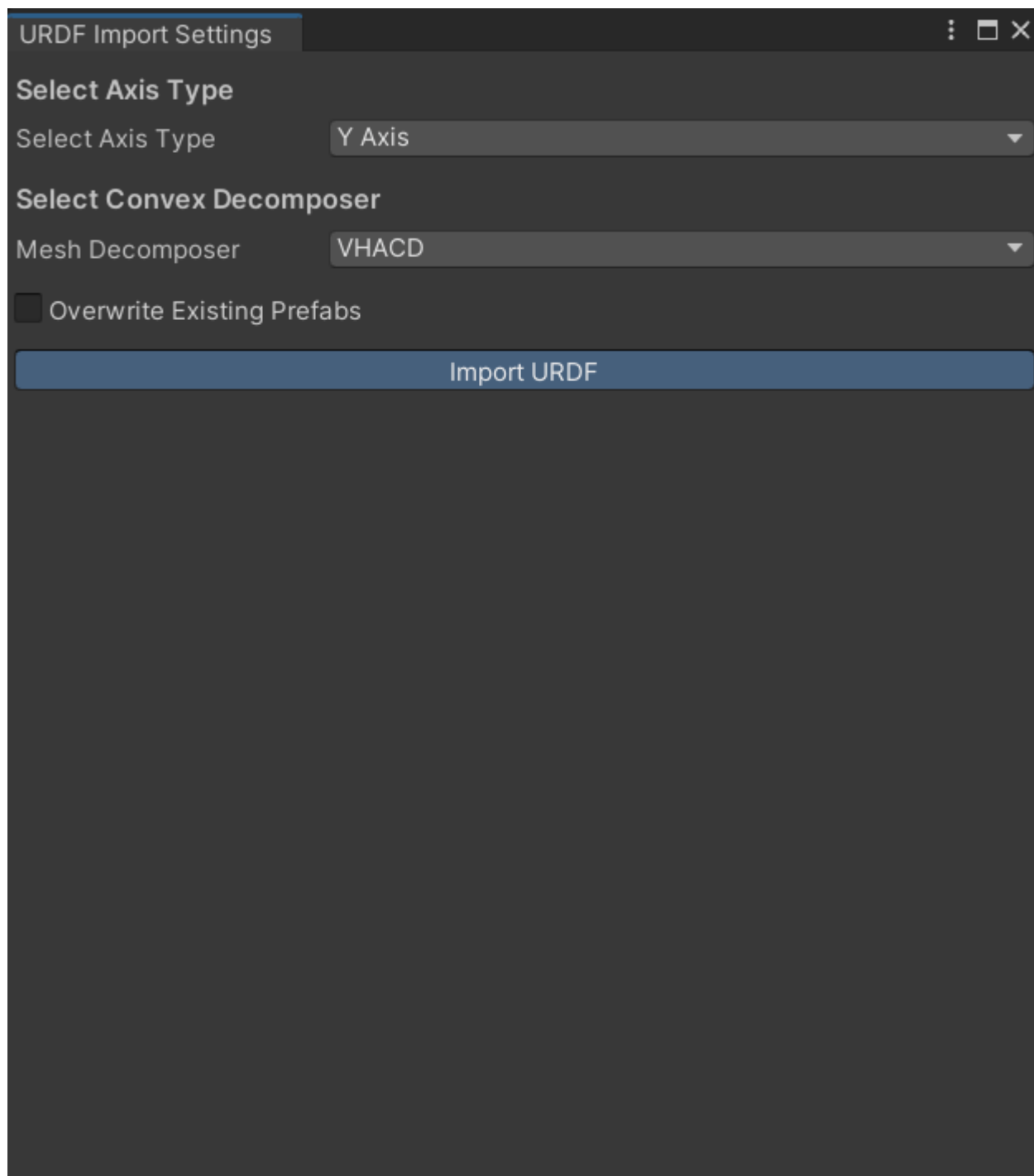
Zoom into the scene to view the imported KINOVA Gen3 robot model.
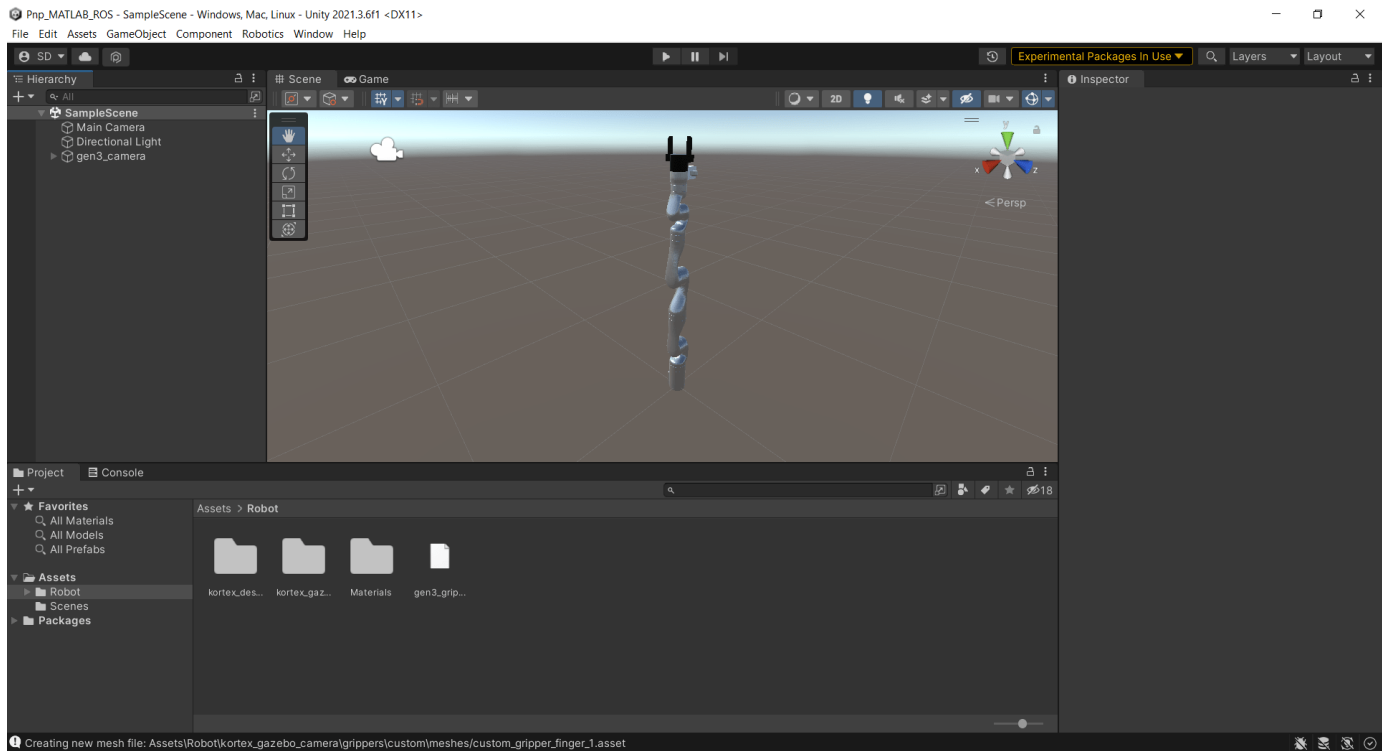
Set Up Scene in Unity

To simulate a pick-and-place workflow in Unity, create an environment in Unity containing a floor, a table, and a box to be picked.

Add Floor

1 From the Unity menu, select **GameObject > 3D Object > Plane**. Unity Editor adds a plane to the scene and lists it in the **Hierarchy** pane.

2 In the **Hierarchy** pane, right-click the plane object and select **Rename**. Rename it as `Floor`.

3 In the **Inspector** pane, under **Transform**, set the **X**, **Y**, and **Z** values of **Position** to `0`.
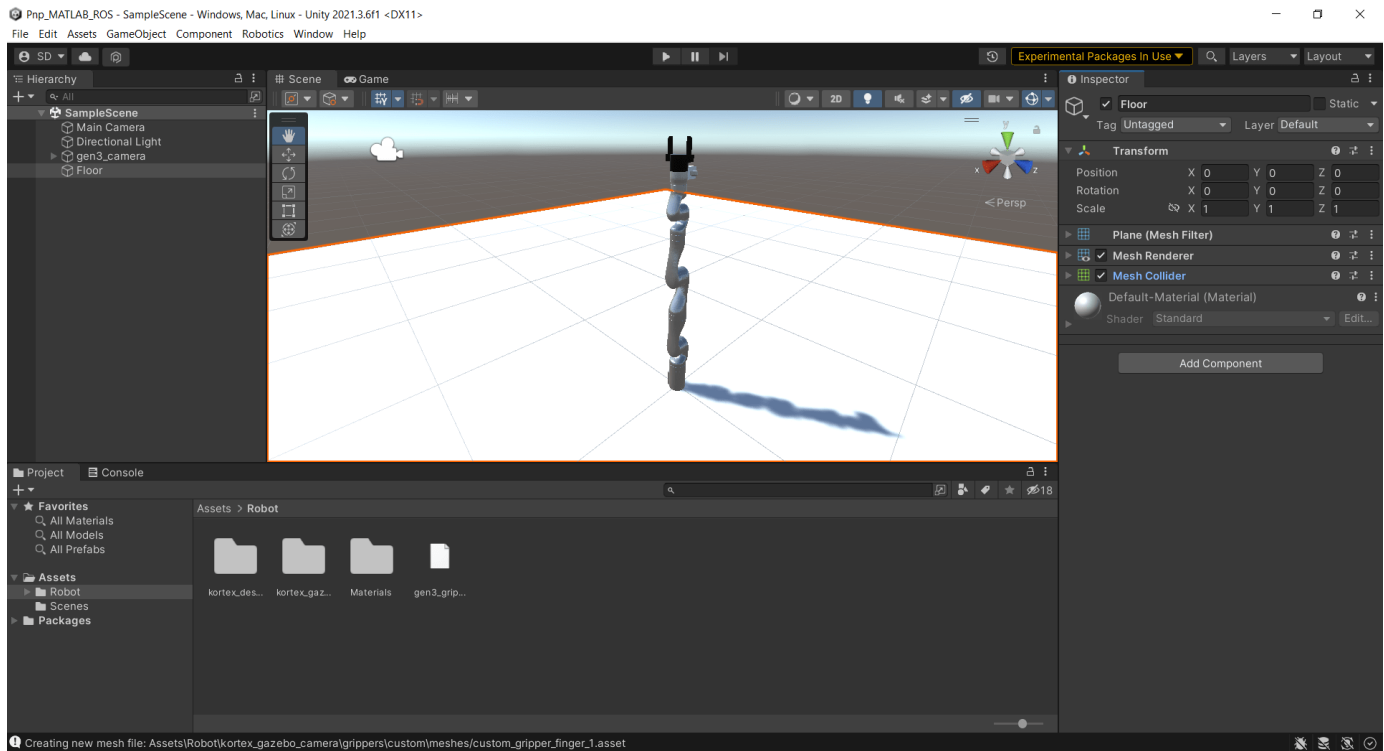
Add Table

1 From the Unity menu, select **GameObject > 3D Object > Cube**. Unity Editor adds a cube to the scene and lists it in the **Hierarchy** pane.

2 In the **Hierarchy** pane, right-click the cube object and select **Rename**. Rename it as `Table`.

3 In the **Inspector** pane, under **Transform**, set the **Y** and **Z** values of **Position** to `0.25` and set **Y** value of **Scale** to `0.5`.
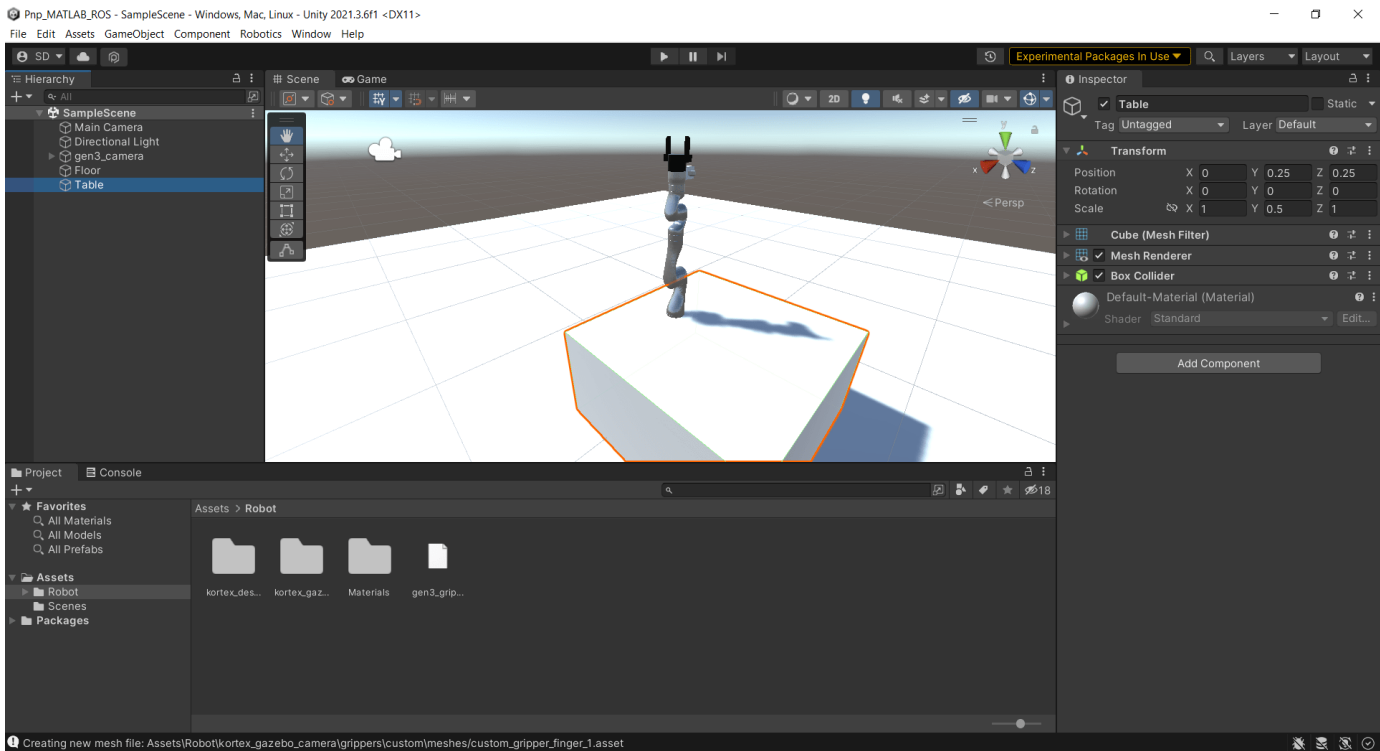
Add Box

1   From the Unity menu, select **GameObject > 3D Object > Cube**. Unity Editor adds a cube to the scene and lists it in the **Hierarchy** pane.

2   In the **Hierarchy** pane, right-click the cube object and select **Rename**. Rename it as `Box`.

3   In the **Inspector** pane, under **Transform**, set the **Y** value of **Position** to `0.525` and **Z** values of **Position** to `0.25`.

4   Set the **X**, **Y**, and **Z** values of **Scale** to `0.05`.

5   In the **Inspector** pane, under **Mesh Renderer**, change the **Material** property of the box to a non-default color for ease of visualization.

6   Click **Add Component** and search for `rigidbody`.

7   Select the **RigidBody** component to add it to the **Inspector** pane.

8   In the **Inspector** pane, under **RigidBody**, set the value of **Mass** to `0.01`. if **Use Gravity** is not selected, select it.
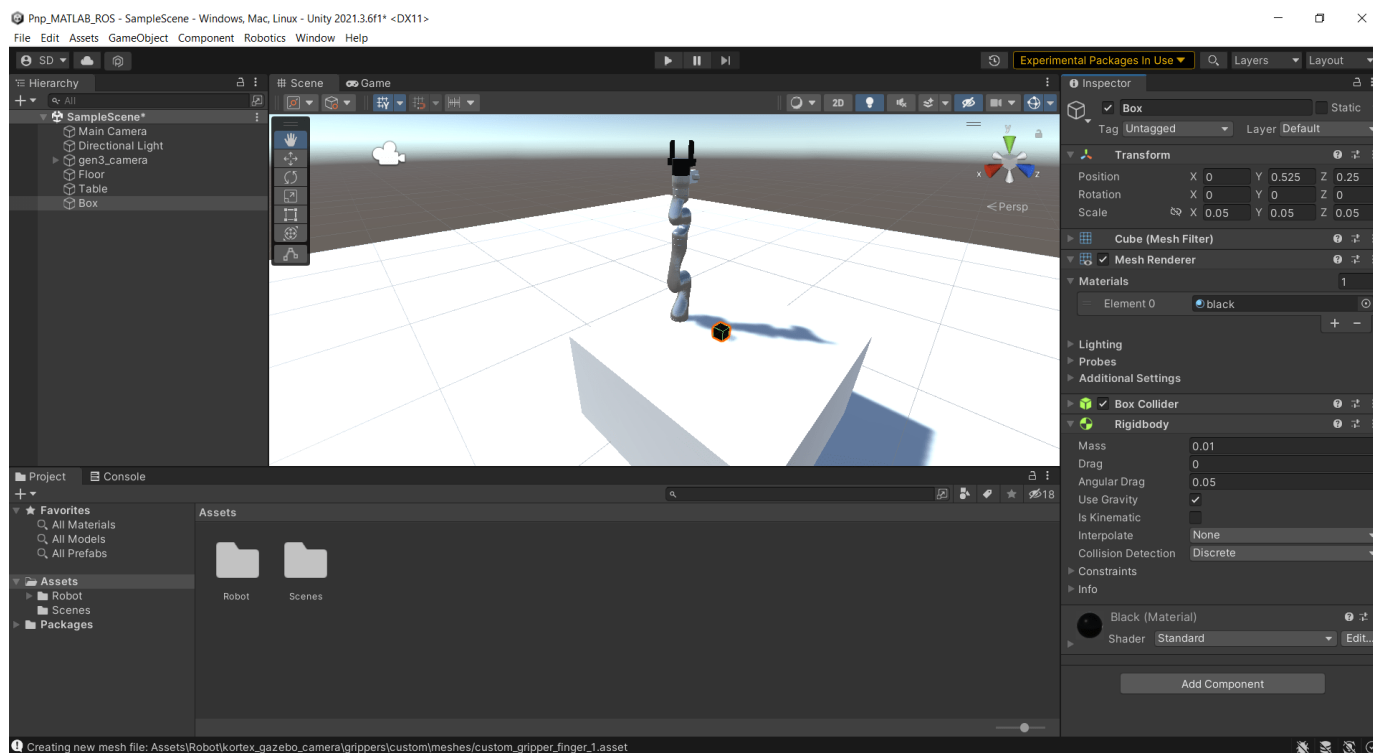
Add Tag to Box

1 In the **Hierarchy** pane, select **Box**.

2 In the **Inspector** pane, click the **Tag** drop-down and select **Add Tag**.

3 Click the **+** sign to create a tag. Name the tag `PickObject`.

4 In the **Hierarchy** pane, select **Box**.

5 In the **Inspector** pane, click the **Tag** drop-down and select **PickObject**.
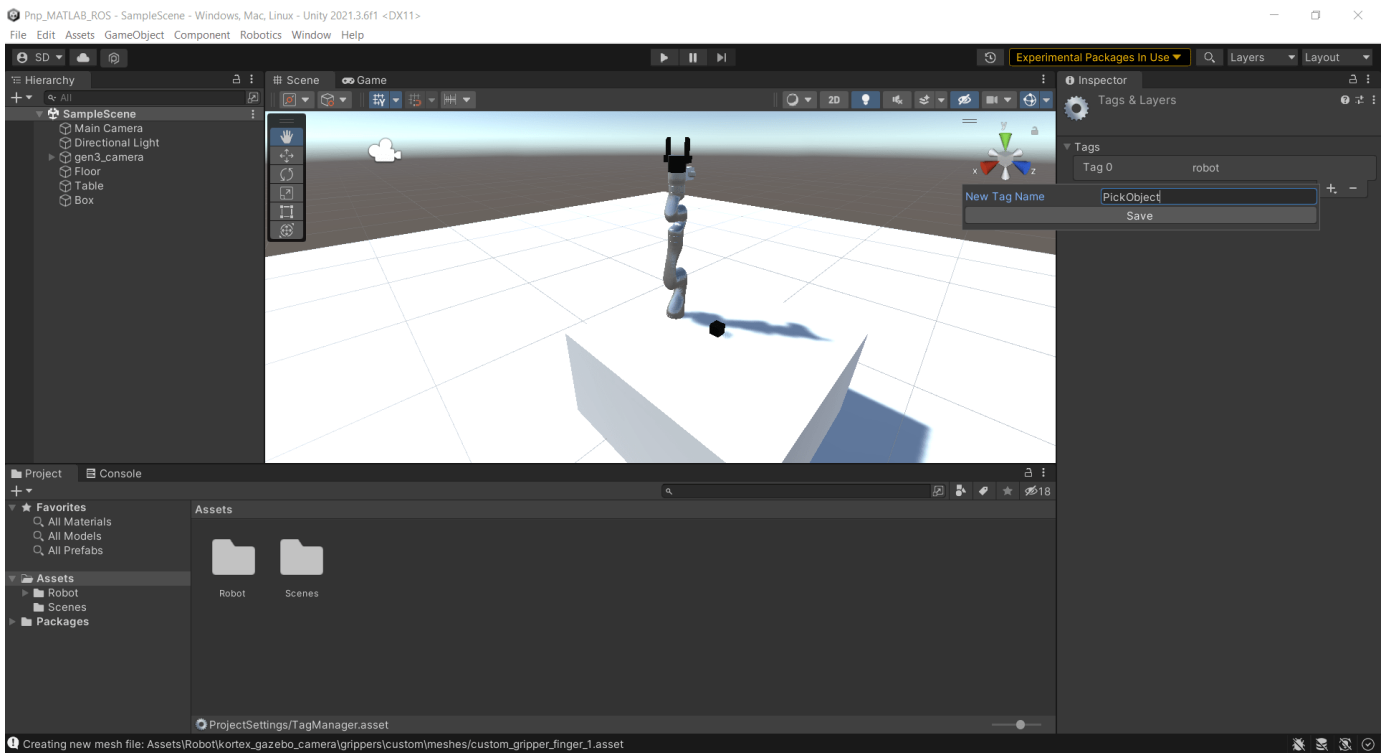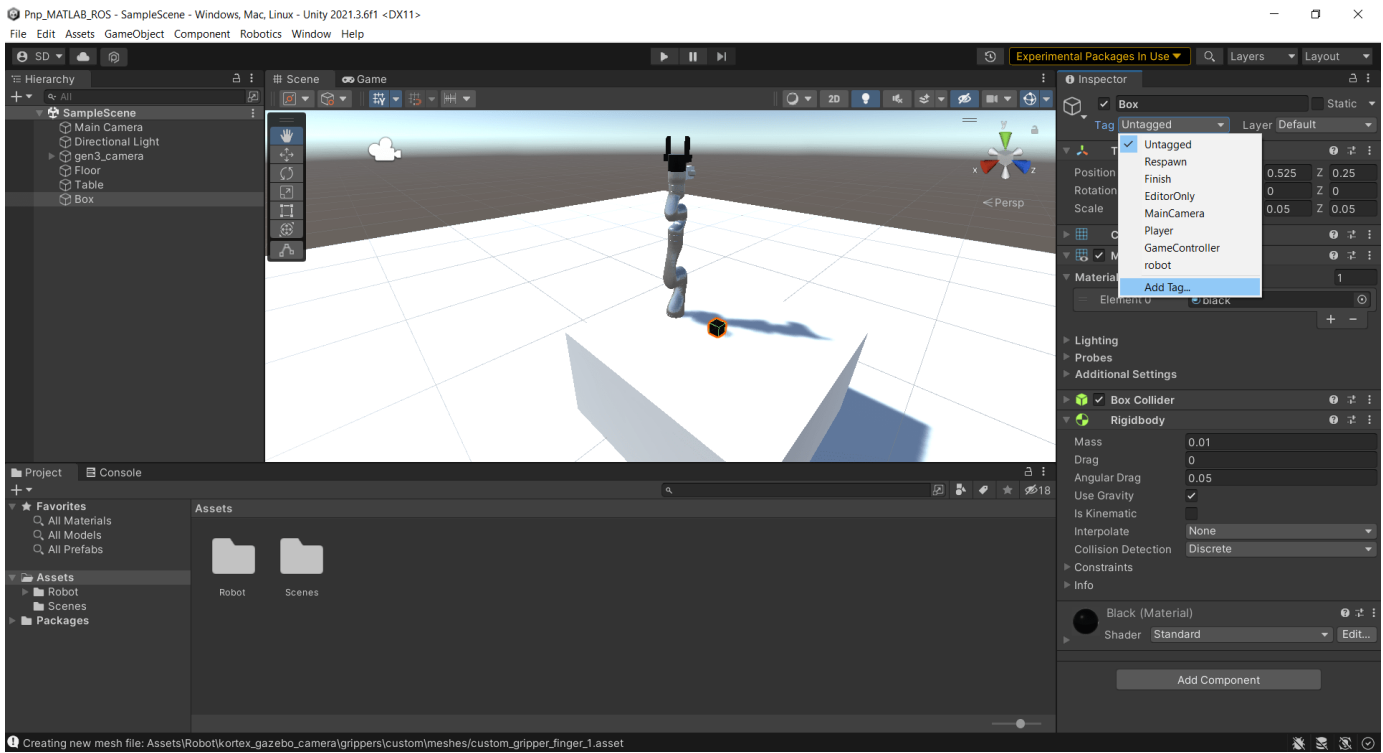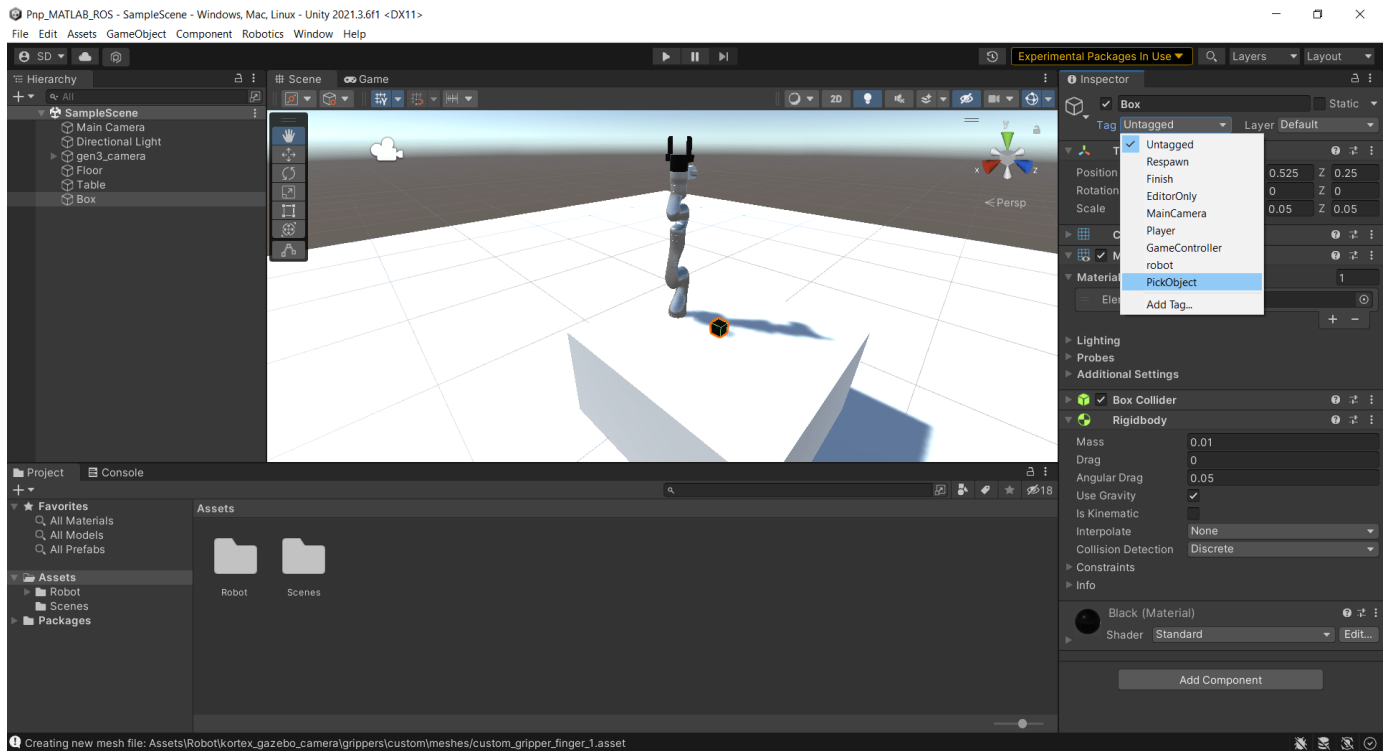
You can repeat the steps in Add Box on page 1-17 and Add Tag to Box on page 1-18 to add more objects for the robot to pick.

Adjust Camera

1 In the **Hierarchy** pane, select **Main Camera**.

2 In the **Inspector** pane, under **Transform**, adjust the **Position** and **Rotation** properties.

3 Alternatively, you can manually adjust the camera position and orientation using the tool overlay in the **Scene** pane.

4 You can preview the point of view of the camera while adjusting it.
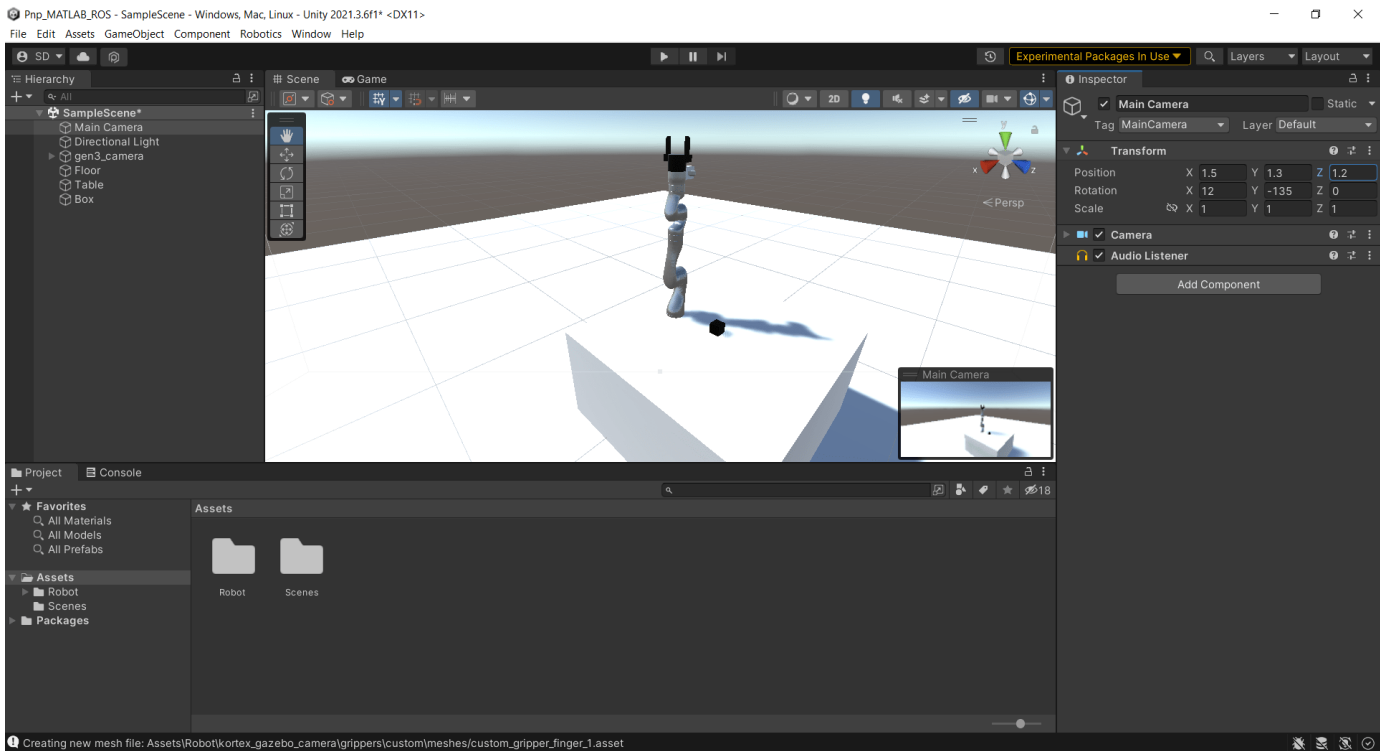
Configure Robot Model in Unity

To configure the robot model:

1   In the **Hierarchy** pane, select the robot object **gen3_camera**.

2   In the **Inspector** pane, under **Transform**, set the **Y** value of **Position** to 0.5, which positions the robot on the top of the table.

3   Under **Controller (Script)**, set **Stiffness** to 1000, **Damping** to 100, and **Force Limit** to 10.

4   Rename the robot object to **gen3**, to match the robot name used in the setCurrentRobotJConfig method of the exampleHelperCoordinatorPickPlaceROSUnity class.

The setCurrentRobotJConfig method of the exampleHelperCoordinatorPickPlaceROSUnity class uses this code.

```
function configRespUnity = setCurrentRobotJConfig(obj,JConfig)
    disp("Sending robots to initial positions...")

    %For Unity
    configReqUnity = rosmessage(obj.ROSinfo.UnityConfigClient);
    configReqUnity.ModelName = "gen3";
    configReqUnity.JointPositions = JConfig;
    configRespUnity = call(obj.ROSinfo.UnityConfigClient,configReqUnity,"Timeout",3);
end
```

1-21
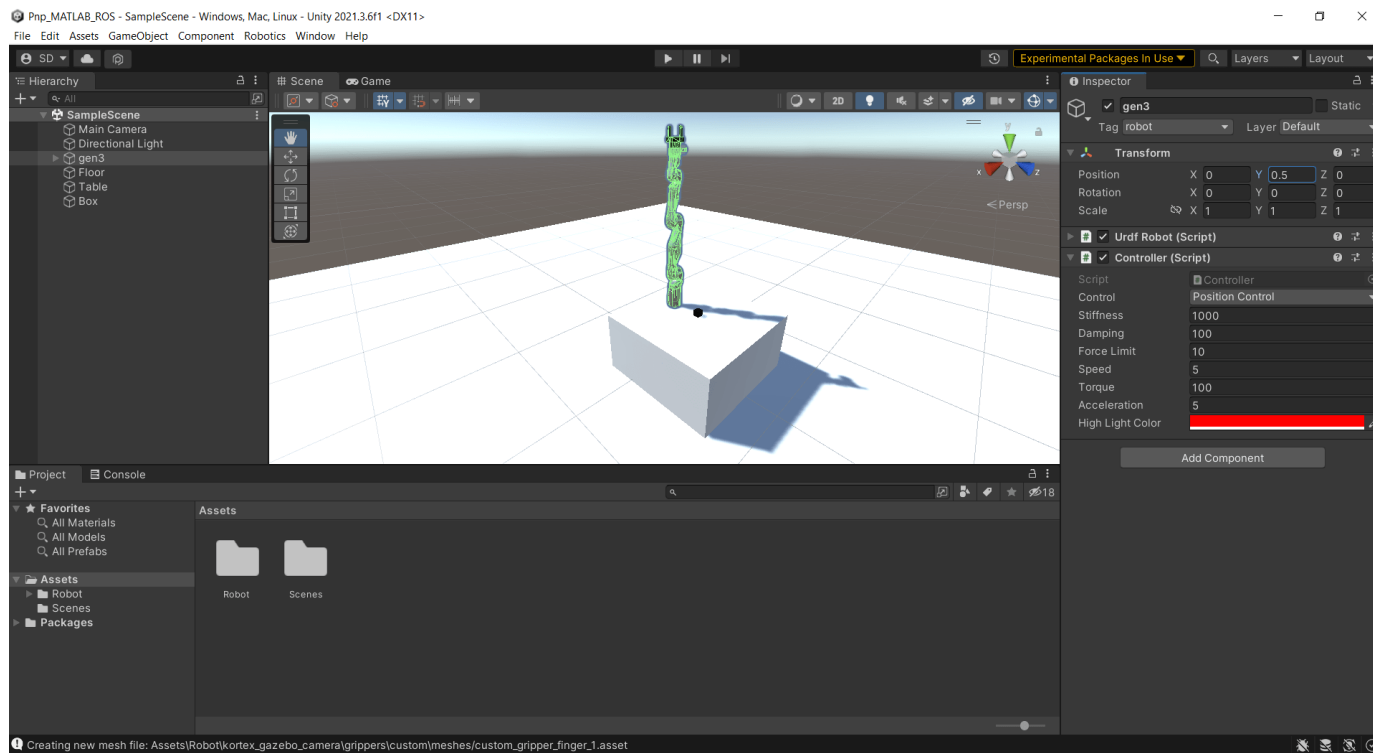
To configure the base link of the robot model:

1   In the **Hierarchy** pane, under the **gen3** robot object, select **base_link**.

2   In the **Inspector** pane, under **Articulation Body**, select **Immovable**.
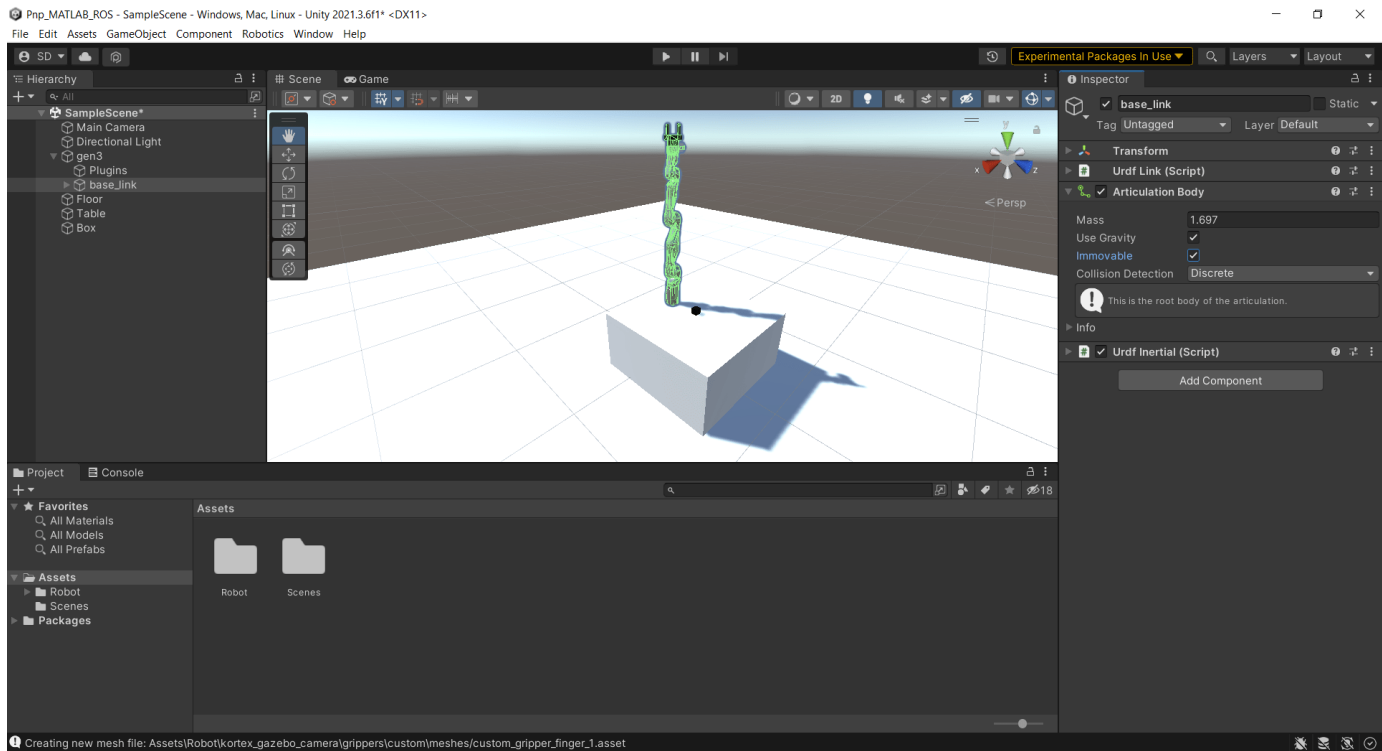
To add collision boxes to the gripper fingers:

1 In the **Hierarchy** pane, under the **gen3** robot object, select **gripper_finger1_finger_tip_link**.

2 In the **Inspector** pane, click **Add Component** and search for `box collider`.

3 Select the **Box Collider** component to add it to the **Inspector** pane.

4 Adjust the values of the **Center** and **Size** properties of the collision box to ensure that it aligns with the gripper finger mesh.

5 Repeat the previous steps for **gripper_finger2_finger_tip_link**.
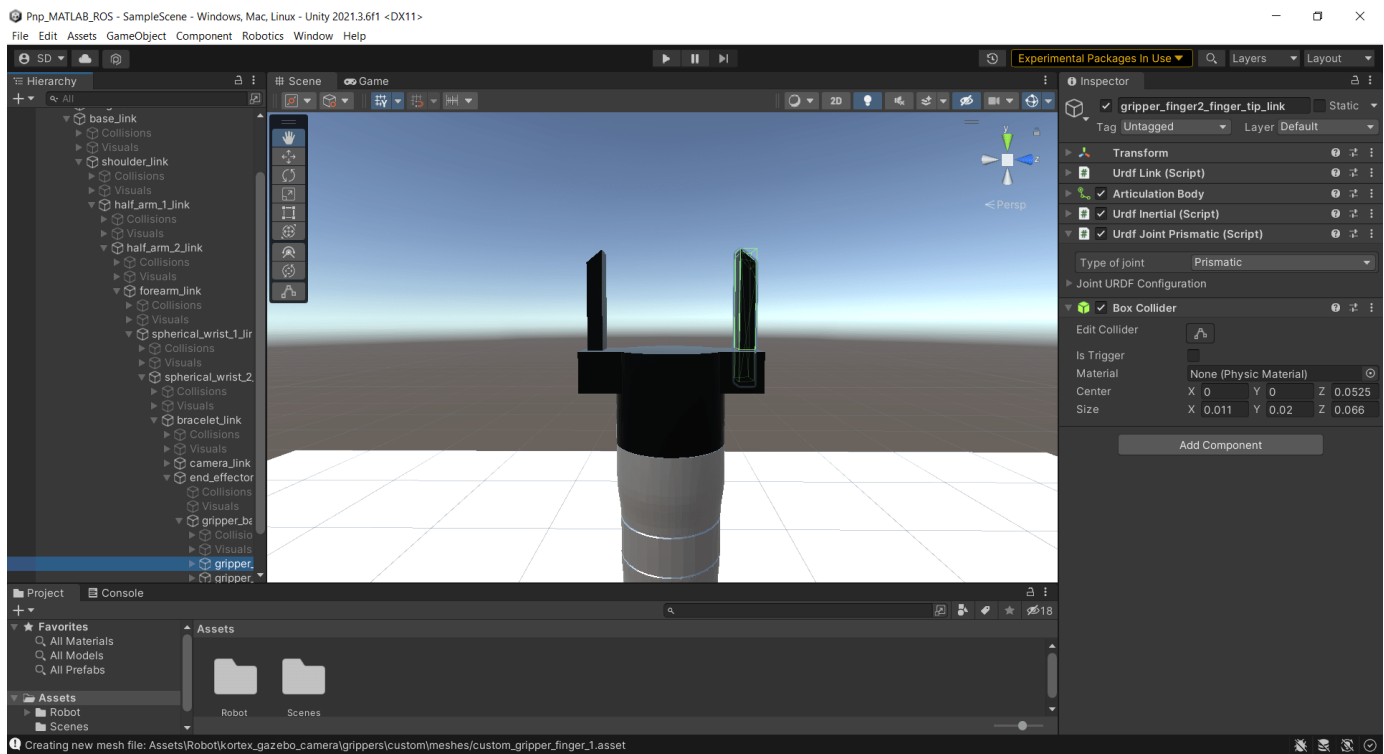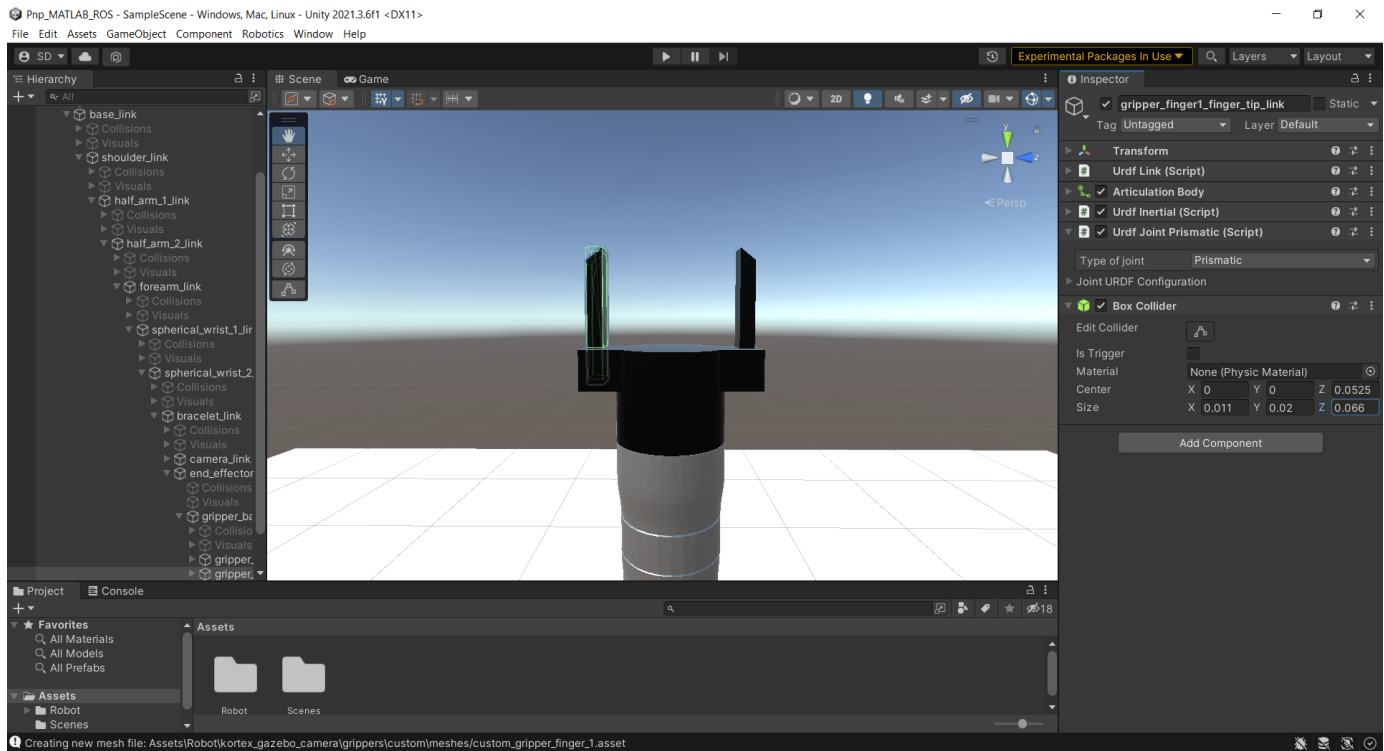
To set the physics solver type:

1 From the menu, select **Edit > Project Settings** to launch the **Project Settings** window.

2   In the left pane of the **Project Settings** window, select **Physics**.

3   In the **Physics** settings pane, set **Solver Type** to **Temporal Gauss Seidel.**
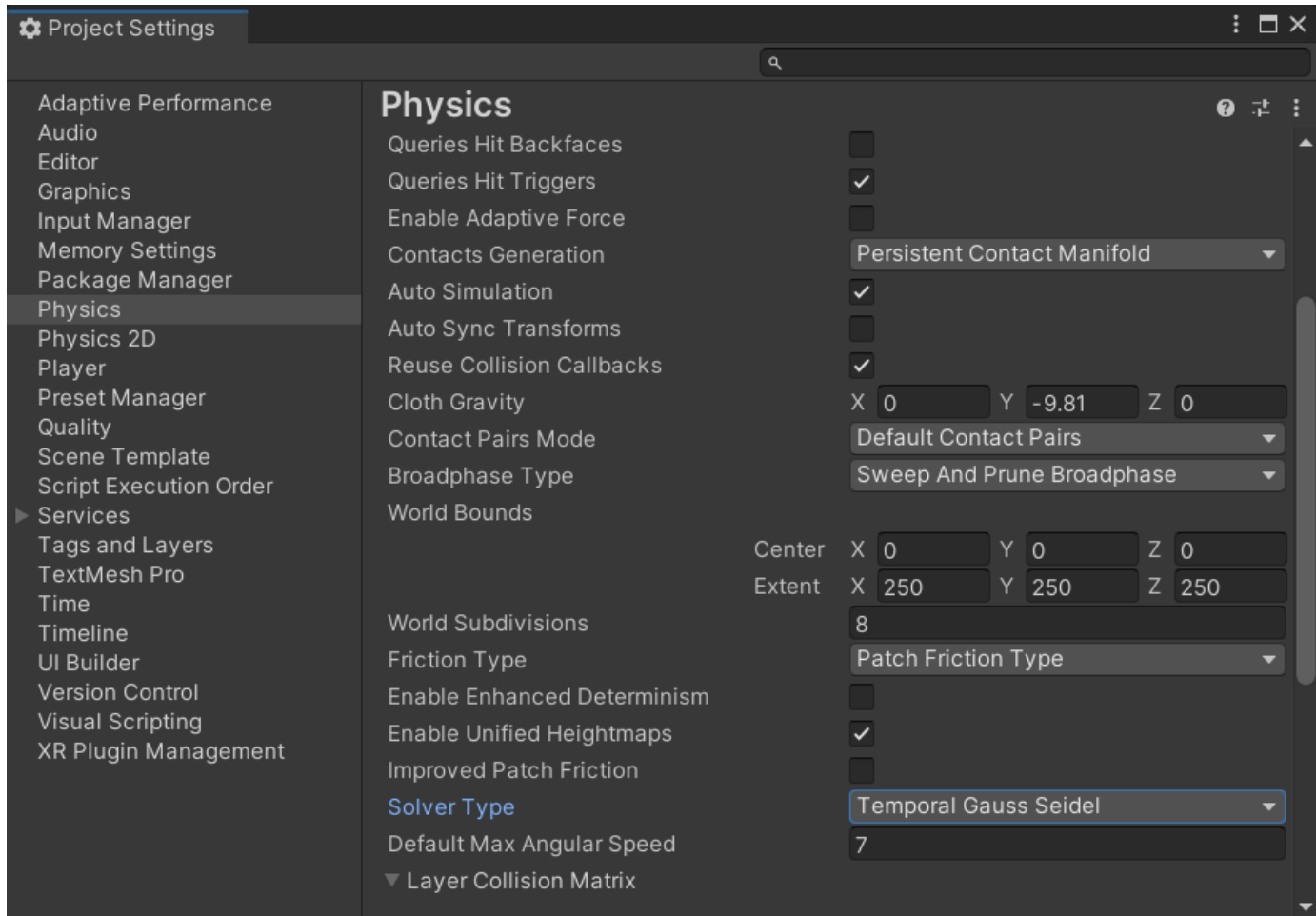
ROS Custom Services and Publisher

MATLAB needs to communicate with Unity in order to transmit and receive information regarding the robot model. The execution of the pick-and-place workflow requires channels of communication between MATLAB and Unity. To establish communication for these channels, use custom services and a publisher.

This example includes the ROS custom service definition (`.srv`) files. You can find these files inside the `unity_robotics_demo_msgs` folder. The Generate Unity Scripts for Custom Services on page 1-26 section shows how to generate ROS messages from the custom service definition files in Unity. The Generate Custom ROS Messages in MATLAB on page 1-32 section shows how to generate ROS messages from the custom service definition files in MATLAB.
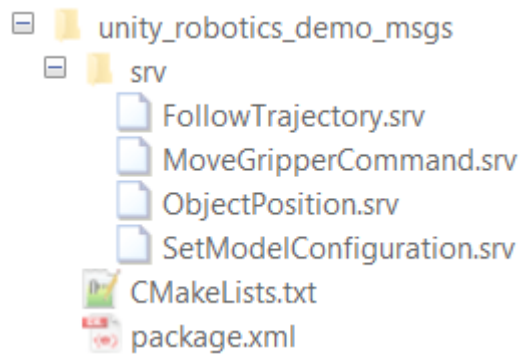
- `SetModelConfiguration.srv` — Sends the robot joint configuration from MATLAB to Unity.
- `FollowTrajectory.srv` — Sends the robot joint-space trajectory from MATLAB to Unity.
- `ObjectPosition.srv` — Receives the position of the box to be picked from the Unity scene in MATLAB.
- `MoveGripperCommand.srv` — Sends the command to open or close the gripper fingers from MATLAB.

The `PNPScripts.zip` file included with this example contains C# (`.cs`) scripts you can use to establish communication between Unity and MATLAB. The Add ROS Service Scripts in Unity on page 1-30 section shows how to add the Unity ROS service scripts to the scene. For more examples of implementing the Unity ROS service, see these Unity Scripts in the Unity Robotics Hub GitHub repository.
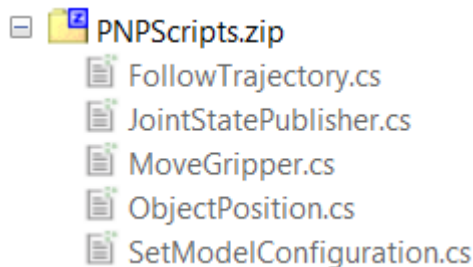
- `SetModelConfiguration.cs` — Receives the robot joint configuration from MATLAB in Unity. Unity then assigns these joint configuration values to the corresponding joints of the robot model. The script also unselects the **Use Gravity** option in all the links of the robot during simulation.
- `FollowTrajectory.cs` — Moves the robot model along a set of joint configurations provided by the trajectory from MATLAB.
- `ObjectPosition.cs` — Sends the position of the box to be picked from the Unity scene to MATLAB.
- `MoveGripper.cs` — Opens or closes the gripper fingers attached to the robot arm in Unity.
- `JointStatePublisher.cs` — Publishes the current joint configuration from Unity to the ROS network in MATLAB. For more information on publishing messages from Unity, see ROS–Unity Communication.

Generate Unity Scripts for Custom Services

You must build the ROS custom service definition files into C# scripts for use in Unity programming. The **ROS TCP Connector** package provides a tool you can use to build messages and services.

To build the custom services:

1 Copy the `unity_robotics_demo_msgs` folder, which contains the ROS custom service
       definition files, to the Unity project folder. For more information on the service definition files in
       the `unity_robotics_demo_msgs` folder, see the ROS Custom Services and Publisher on page
       1-25 section.

2 From the Unity menu bar, select **Robotics > Generate ROS Messages**, to launch the tool for
       building custom services.

3 In the **ROS Message Browser** window, browse to the the `unity_robotics_demo_msgs` folder
       inside the Unity project folder.

4 Click **Build 4 srvs**.

ROS Message Browser	⋮ ☐ ✕
ROS message path	C:/Unity/Projects/Pnp_MATLAB_ROS/unity_robotics_demo_msgs [Browse]
Built message path	RosMessages [Browse]

C:/Unity/Projects/Pnp_MATLAB_ROS/unity_robotics_demo_msgs:

▼ unity_robotics_demo_msgs

  ▼ srv   [Build 4 srvs]

    FollowTrajectory.srv   [Build srv]

    MoveGripperCommand.srv   [Build srv]

    ObjectPosition.srv   [Build srv]

    SetModelConfiguration.srv   [Build srv]

Unity generates C# service scripts from the ROS custom service definition files.
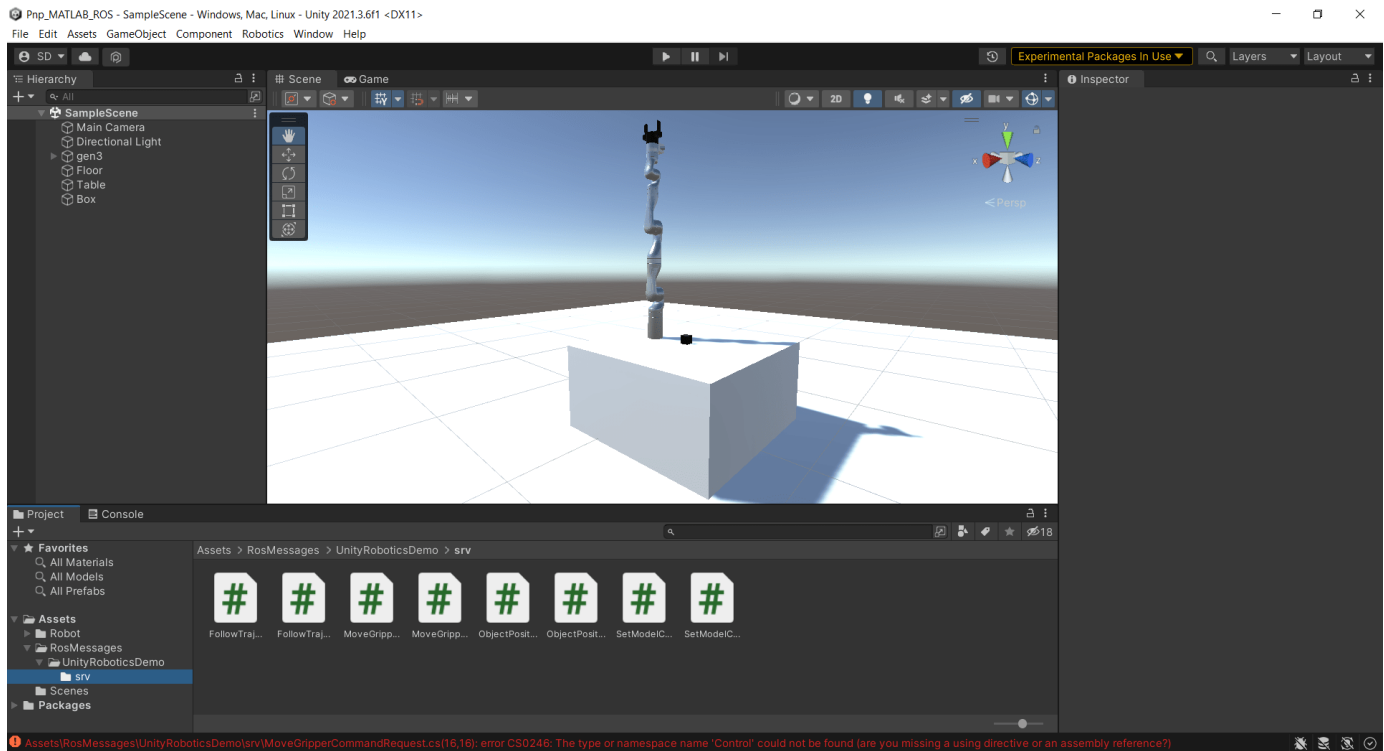
Unity also returns errors regarding the missing `control_msgs` messages.
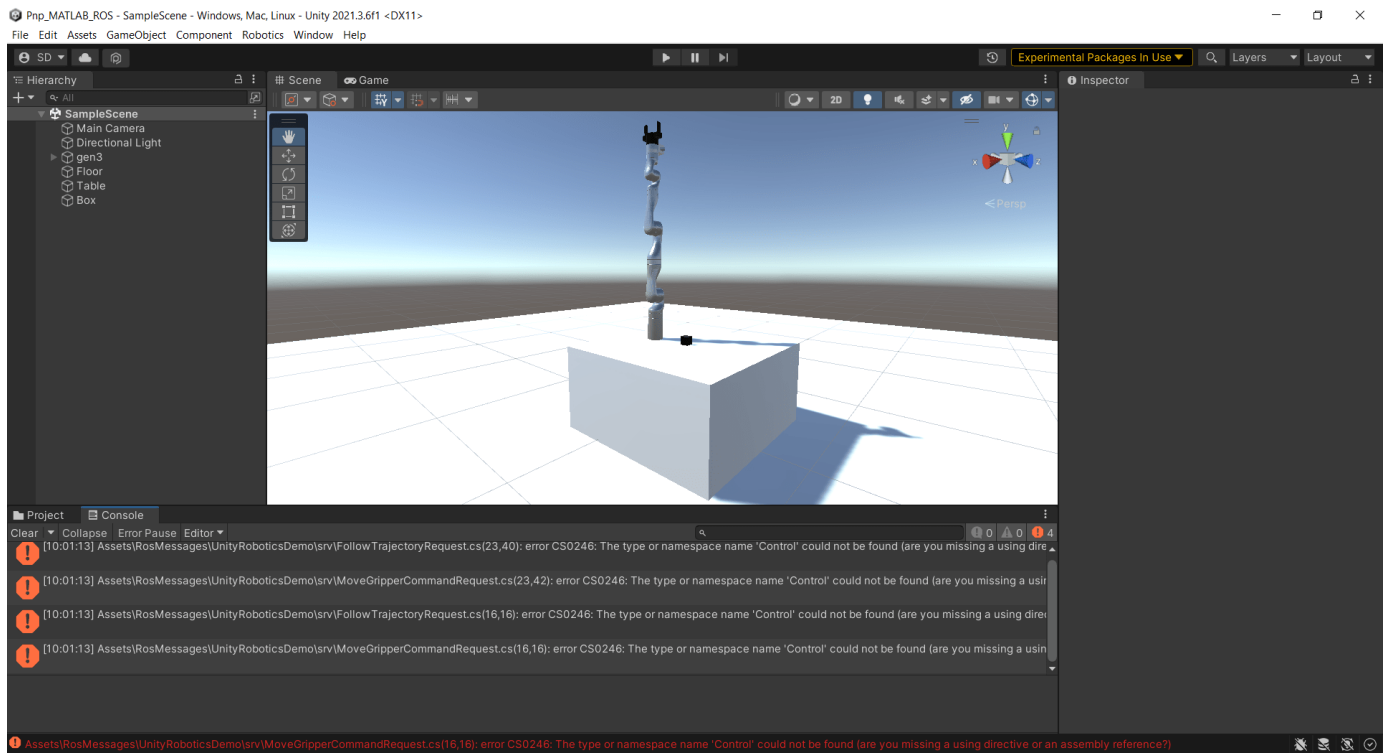
To build the required `control_msgs` messages:

1 Extract the Control_Msgs.zip file inside the Unity project folder. For more information on ROS control messages, see control_msgs Github repository.

2 From the Unity menu bar, select **Robotics > Generate ROS Messages**, to launch the tool for building these messages.

3 In the **ROS Message Browser** window, browse to the the `control_msgs` folder inside the Unity project folder.

4 Build only these messages, named in the errors returned by Unity: **FollowJointTrajectoryGoal.msg**, **GripperCommand.msg**, and **JointTolerance.msg**.

5 Click **Build 3 msgs**.

ROS Message Browser	⋮ □ ×
ROS message path	C:/Unity/Projects/Pnp_MATLAB_ROS/control_msgs Browse
Built message path	RosMessages Browse

C:/Unity/Projects/Pnp_MATLAB_ROS/control_msgs:

▼ control_msgs
  ▼ msg   **Build 3 msgs**
    FollowJointTrajectoryGoal.msg   **Build msg**
    GripperCommand.msg   **Build msg**
    JointTolerance.msg   **Build msg**
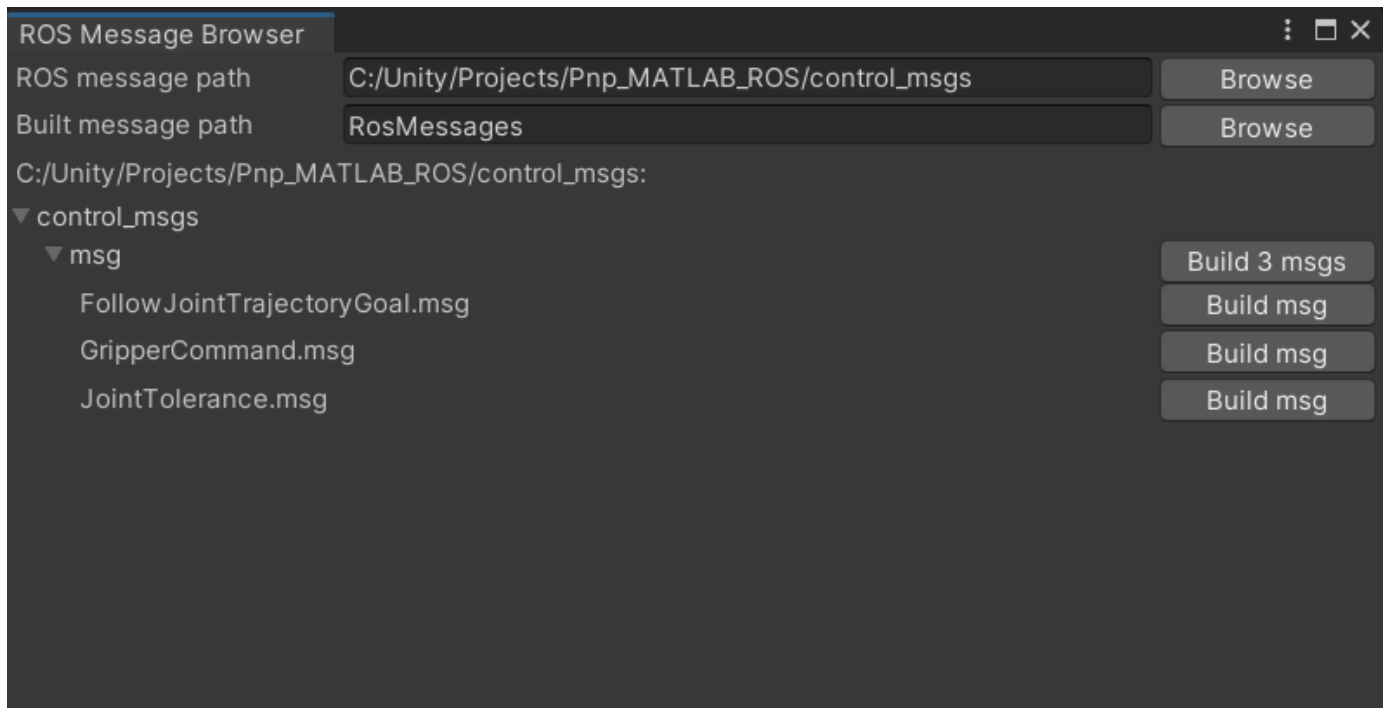
Unity generates C# message scripts from the `control_msgs` message definition files, which resolves the errors.
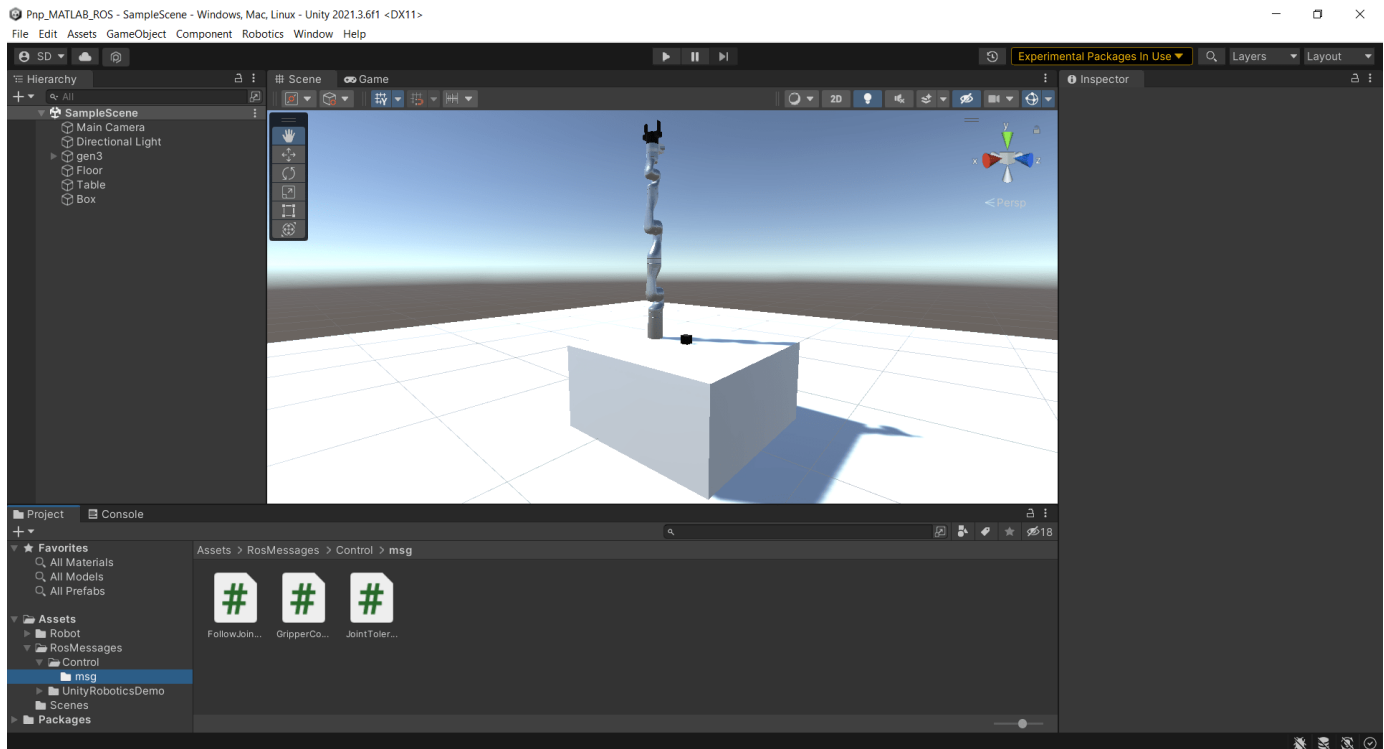
Add ROS Service Scripts in Unity

To add the Unity ROS service scripts to the scene:

1 Extract the PNPScripts.zip file to the Assets folder in the Unity project directory. For more information on the service scripts in the PNPScripts.zip file, see the ROS Custom Services and Publisher on page 1-25 section.

2 From the Unity menu bar, select **GameObject > Create Empty**. Unity adds an empty **GameObject** to the scene and lists it in the **Hierarchy** pane.

3 Rename the empty **GameObject** to **PNPScriptsObject**.

4 In the Project pane, select all of the C# script files in the PNPScripts folder.

5 Drag the files to **PNPScriptsObject** in the **Hierarchy** pane.
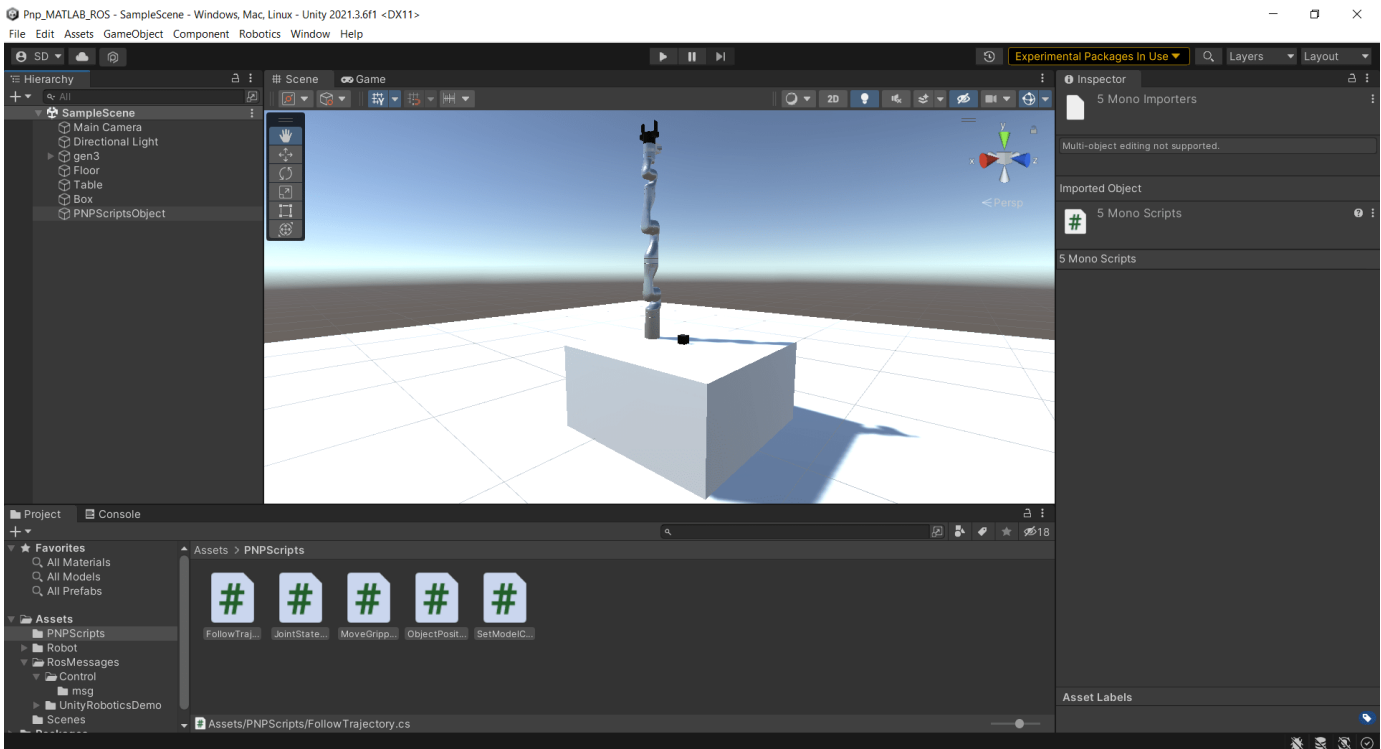
The scripts are added as components of **PNPScriptsObject**. In the **Hierarchy** pane of
PNPScriptsObject, set the **Gen 3** property of the **Follow Trajectory**, **Joint State Publisher**, **Move
Gripper**, and **Set Model Configuration** components to the **gen3** robot object.
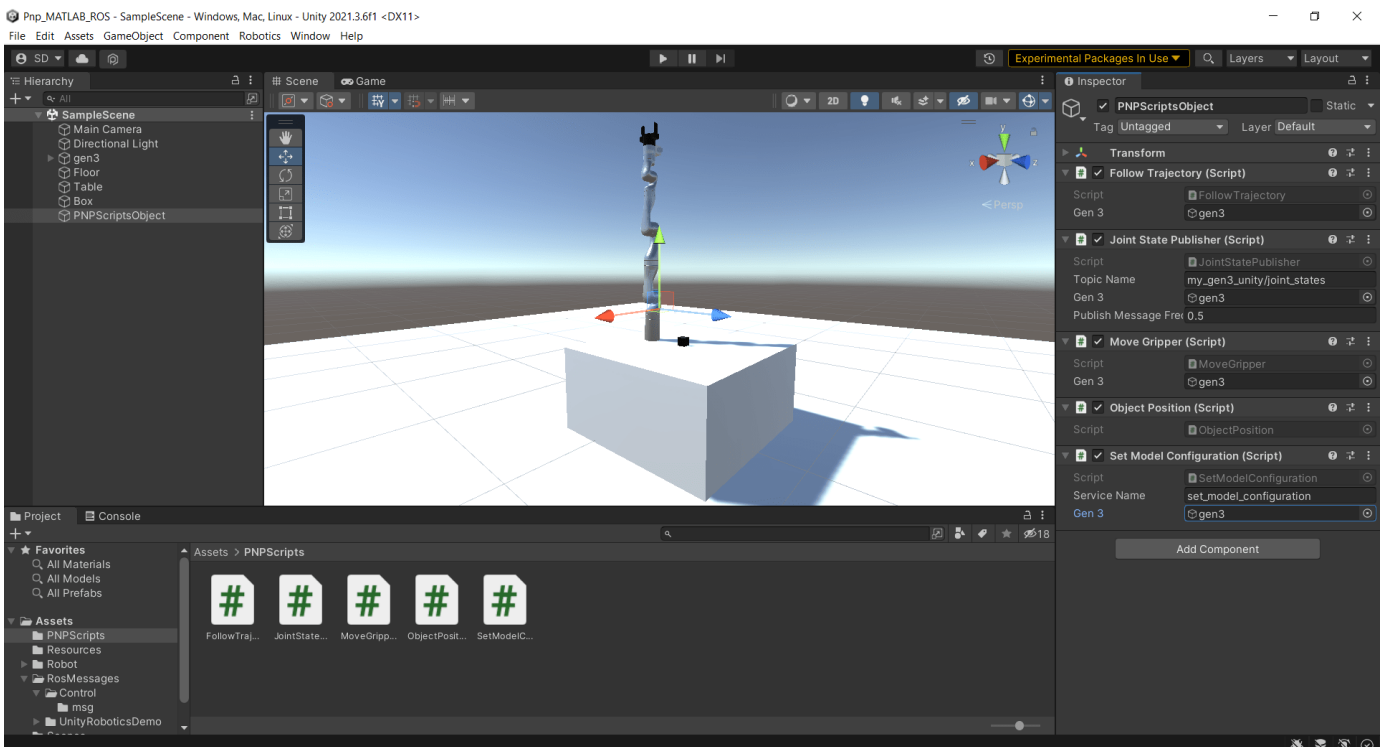
Create Pick-and-Place Workflow in MATLAB

This example uses a Stateflow chart to schedule the tasks of the pick-and-place workflow. Open the chart to examine the contents and follow the state transitions during chart execution.

edit exampleHelperFlowChartPickPlaceROSUnity.sfx

The chart dictates how the manipulator interacts with the objects, or parts. It consists of basic initialization steps, followed by two main sections:

- Identify parts and determine where to place them.
- Execute the pick-and-place workflow.

For a high-level description of the pick-and-place steps, see "Pick-and-Place Workflow Using Stateflow for MATLAB" (Robotics System Toolbox).

Opening and Closing Gripper

The exampleCommandActivateGripperROSUnity command function sends a service request to open and close the gripper implemented in Unity.

Moving Manipulator to Specified Pose

Use the exampleCommandMoveToTaskConfigROSUnity command function to move the manipulator to specified poses. After generating a joint trajectory for the robot to follow, the exampleCommandMoveToTaskConfigROSUnity function samples the trajectory at the desired sample rate, then packages it into joint-trajectory ROS messages, and sends a service request to Unity.

Trajectory Planning

Use trapveltraj (Robotics System Toolbox) and transformtraj (Robotics System Toolbox) functions to generate simple task-space trajectories from an initial to a desired task configuration. For more details on planning and executing trajectories, see "Plan and Execute Task- and Joint-Space Trajectories Using KINOVA Gen3 Manipulator" (Robotics System Toolbox).

Generate Custom ROS Messages in MATLAB

You must generate ROS custom messages in MATLAB from the custom service definition files inside the unity_robotics_demo_msgs folder. For more information on these service definition files, see the ROS Custom Services and Publisher on page 1-25 section.

Use the rosgenmsg function to generate the custom messages, and follow the instructions in the dialog box to add the messages to the MATLAB path.

```
answer = questdlg("Are the Custom ROS messages generated for MATLAB?", ...
    "Custom ROS messages for MATLAB","Yes","No", "No");
switch answer
    case "Yes"
        % Continue with pick and place
    case "No"
        % Run rosgenmsg to build custom ROS messages for MATLAB
        rosgenmsg
end

answer = questdlg("Are the Custom ROS messages added to MATLAB path?", ...
```

```matlab
    "ROS messages","Yes","Add path...", "Add path from current directory", "Yes");

switch answer
    case "Yes"
        % Continue with pick and place
    case "Add path..."
        % Add the custom ROS messages to the MATLAB path
        customROSMsgPath = uigetdir("", "Select the install\m folder created by rosgenmsg");
        addpath(customROSMsgPath)
        savepath
        clear classes
        rehash toolboxcache
    case "Add path from current directory"
        customROSMsgPath = uigetdir(fullfile(pwd, "matlab_msg_gen_ros1"), "Select the install\m
        addpath(customROSMsgPath)
        savepath
        clear classes
        rehash toolboxcache
end
```

Set Up MATLAB for Unity Connection

ROS Master Setup in MATLAB

Start the ROS master from MATLAB. Alternatively, you can connect Unity and MATLAB to a ROS master running elsewhere.

```matlab
% Close any existing ROS Master
rosshutdown
% Create a ROS Master
rosinit
```

```
Launching ROS Core...
........Done in 8.3609 seconds.
Initializing ROS master on http://172.18.250.141:11311.
Initializing global node /matlab_global_node_65508 with NodeURI http://hyd-shivarad:53031/ and Ma
```

Run TCP Endpoint Node in MATLAB

A ROS TCP endpoint node can facilitate the communication between Unity and ROS for sending and receiving messages.

Start the ROS TCP endpoint node in MATLAB.

```matlab
helperConnectROSUnity;
```

Unity ROS Connection Settings

From the Unity menu bar, select **Robotics > ROS Settings**, to open the tool to specify the IP address of the machine running the ROS master in the MATLAB session. For a detailed guide on setting up the ROS Connection, see The Unity Side.

ROS Settings	⋮ □ ✕
Connect on Startup	✓
Protocol	ROS1 ▾
Settings for a new ROSConnection.instance	
ROS IP Address	172.18.250.141
ROS Port	10000
Show HUD	✓
KeepAlive time (secs)	1
Network timeout (secs)	2
Sleep time (secs)	0.01
Listen for TF Messages	✓
Compass settings	
Unity Z Axis Direction	North ▾
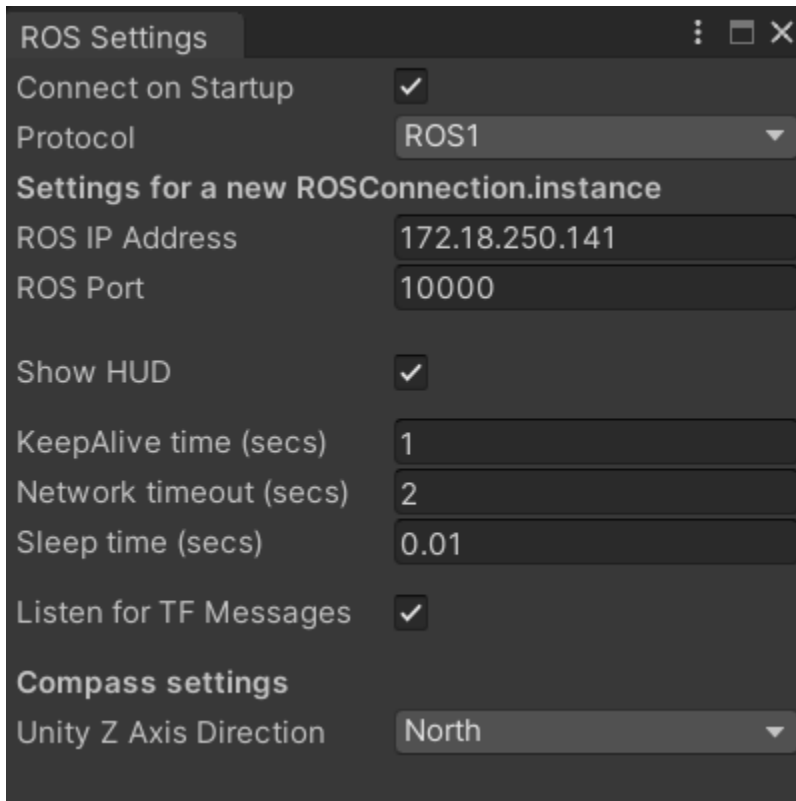
Unity Setup Verification

Click the play button in Unity to simulate the scene. The console displays any critical errors in your setup. Address the errors before proceeding with the example.

Ensure the Unity scene is running and that the connection to MATLAB is functioning. When the Unity sceneis playing, you should see an overlay on the **Game** pane. The blue and green arrows next to **ROS IP** indicate that there is a connection between Unity and the ROS master.
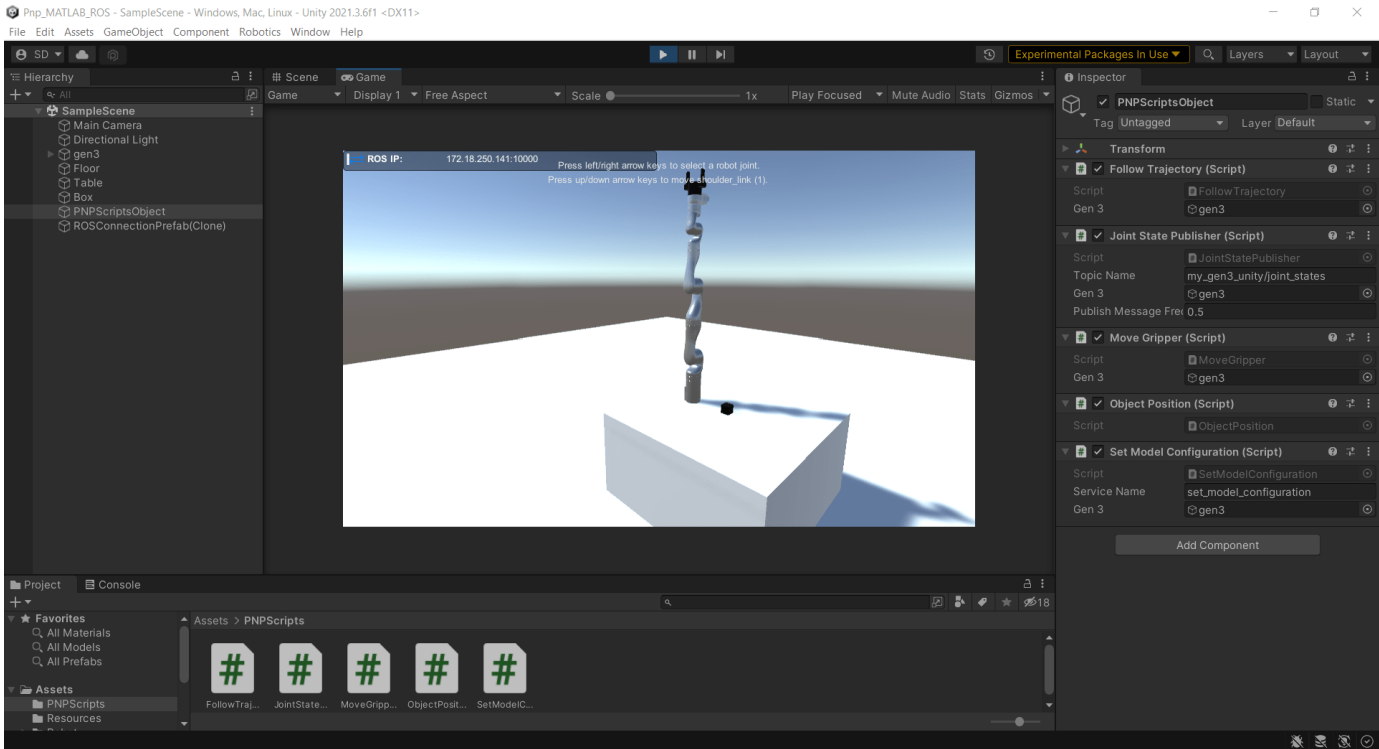
Unity Setup Check in MATLAB

Use a dialog to confirm the Unity is running. Click **Yes** in the dialog to begin the simulation.

```
answer = questdlg("Click the Play button in Unity. Select Yes once the connection is established
                  "Run Unity Scene","Yes","No", "No");
switch answer
    case "Yes"
        % Continue with pick and place
    case "No"
        % Stop the pick and place job
        coordinator.FlowChart.endPickPlace;
        delete(coordinator.FlowChart)
        delete(coordinator);
end
```

To check the connection status in MATLAB is functioning, run the following `rostopic` commands and verify if MATLAB is receiving the Unity published ROS topic `/my_gen3_unity/joint_states`.

```
rostopic list

/my_gen3_unity/joint_states
/rosout
/tf

rostopic echo /my_gen3_unity/joint_states


  Header
    Stamp
      Sec  :  0
```

```
    Nsec :  0
   Seq   :  1
   FrameId :  1
  Name    : {1}  shoulder_link   {2}  half_arm_1_link   {3}  half_arm_2_link   {4}  forearm_li
  Position : [3.84349641535664e-06, -5.377179604693083e-06, 5.302986664901255e-06, -1.0956663572
  Velocity : [1.649751438306241e-05, -3.894936167378246e-05, 5.202663783397106e-05, 8.829219636;
  Effort   : []
---
```

Start Pick-and-Place Task

Load Robot Model in MATLAB

This simulation uses a KINOVA Gen3 manipulator with a Robotiq gripper attached. Load the robot model from a `.mat` file as a `rigidBodyTree` (Robotics System Toolbox) object.

```
load("exampleHelperKINOVAGen3GripperROSUnity.mat");
```

Initialize Pick-and-Place Coordinator

Set the initial robot configuration. Initialize the coordinator, which handles the robot control, by specifying the robot model, initial configuration, and end-effector name.

```
initialRobotJConfig =  [3.5797 -0.6562 -1.2507 -0.7008 0.7303 -2.0500 -1.9053];
endEffectorFrame = "gripper";
coordinator = exampleHelperCoordinatorPickPlaceROSUnity(robot,initialRobotJConfig,endEffectorFram
```

```
Sending robots to initial positions...
```

In Unity, verify that the robot model has moved to the position specified by `initialRobotJConfig`.
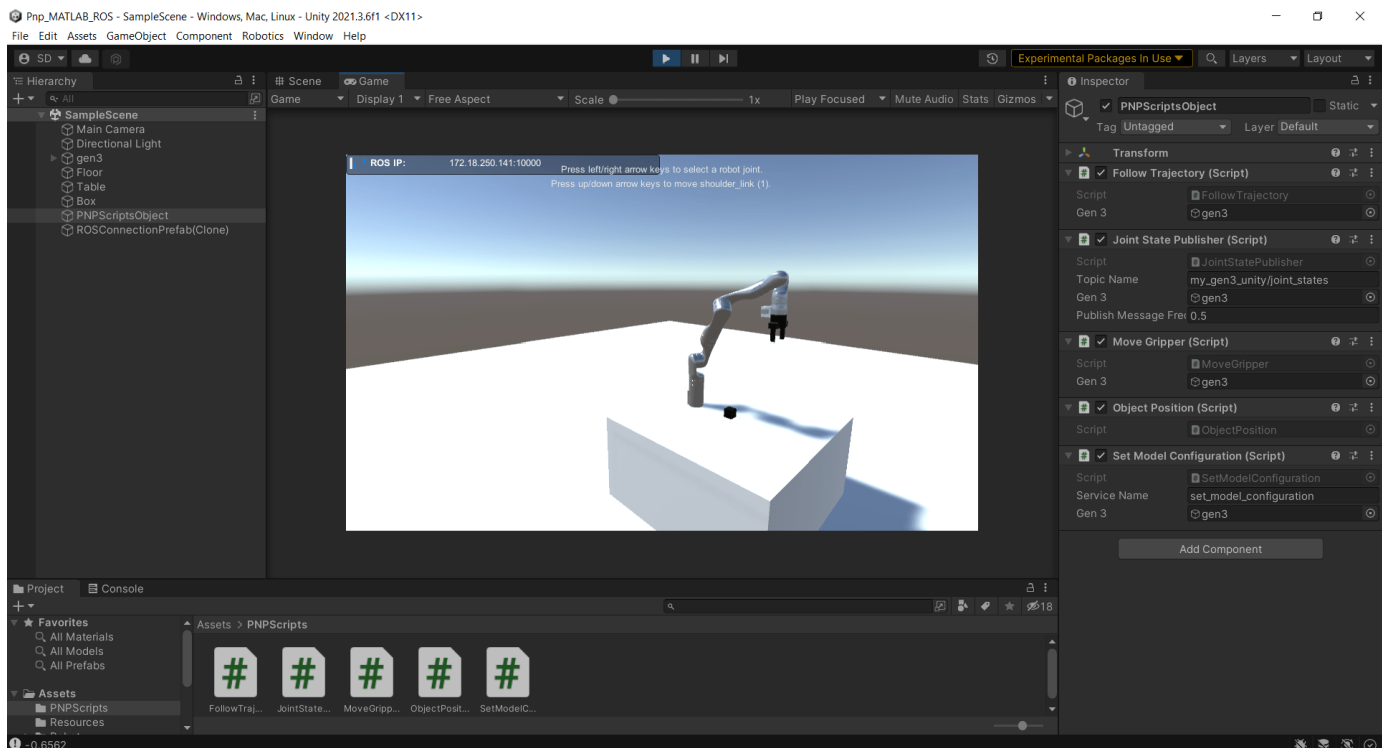
Specify the home configuration and two poses for placing objects.

```
coordinator.HomeRobotTaskConfig = getTransform(robot,initialRobotJConfig,endEffectorFrame);
coordinator.PlacingPose{1} = trvec2tform([0.2 0.55 0.26])*axang2tform([0 0 1 pi/2])*axang2tform(
coordinator.PlacingPose{2} = trvec2tform([0.2 -0.55 0.26])*axang2tform([0 0 1 pi/2])*axang2tform
```

Run and Visualize Simulation

Connect the coordinator to the Stateflow chart. Once started, the Stateflow chart is responsible for continuously going through the states of detecting objects, picking them up, and placing them in the correct staging area.

```
coordinator.FlowChart = exampleHelperFlowChartPickPlaceROSUnity("coordinator",coordinator);
```

Create a dialog box for starting the pick-and-place task execution. To begin the simulation, click **Yes** in the dialog box.

```
answer = questdlg("Do you want to start the pick-and-place job now?", ...
    "Start job","Yes","No", "No");

switch answer
    case "Yes"
        % Trigger event to start Pick and Place in the Stateflow Chart
        coordinator.FlowChart.startPickPlace;
    case "No"
        coordinator.FlowChart.endPickPlace;
        delete(coordinator.FlowChart)
        delete(coordinator)
end
```

Observe the robot model in Unity, and check that it performs the pick-and-place actions.
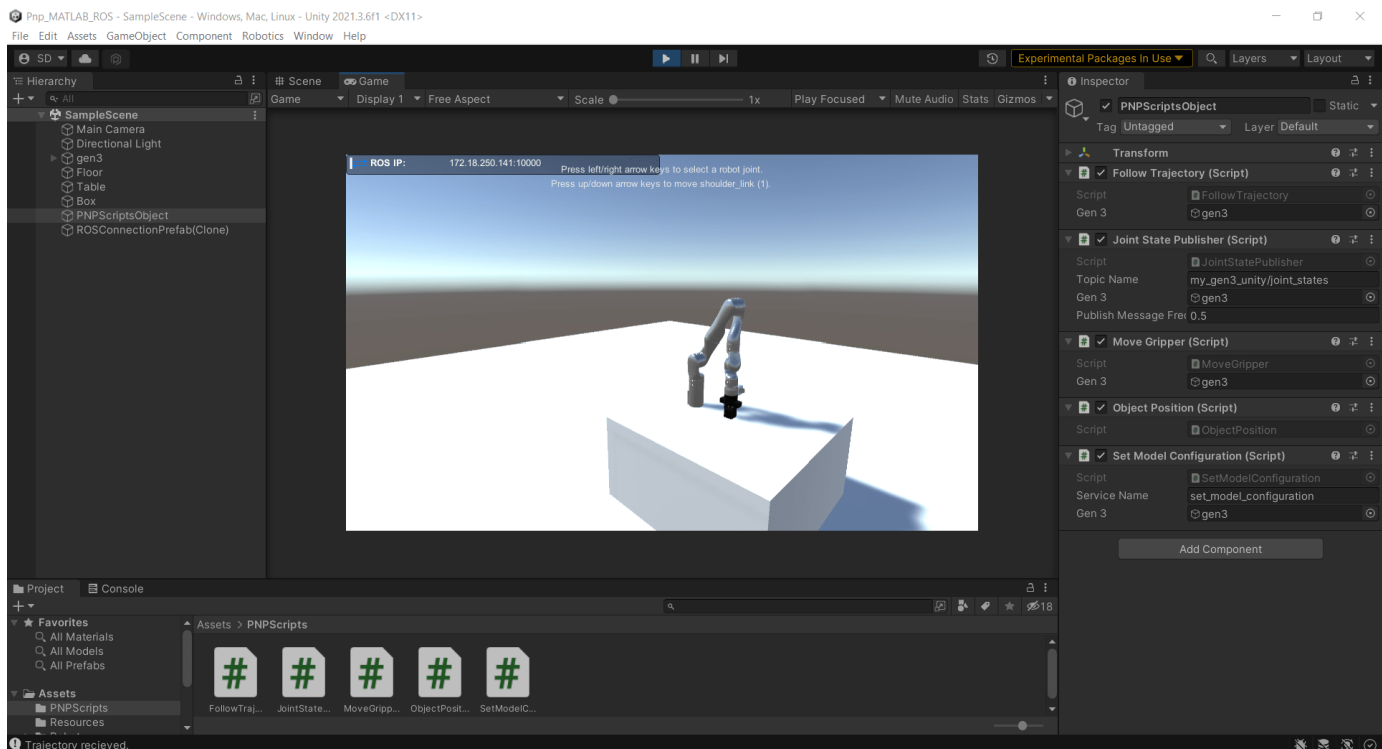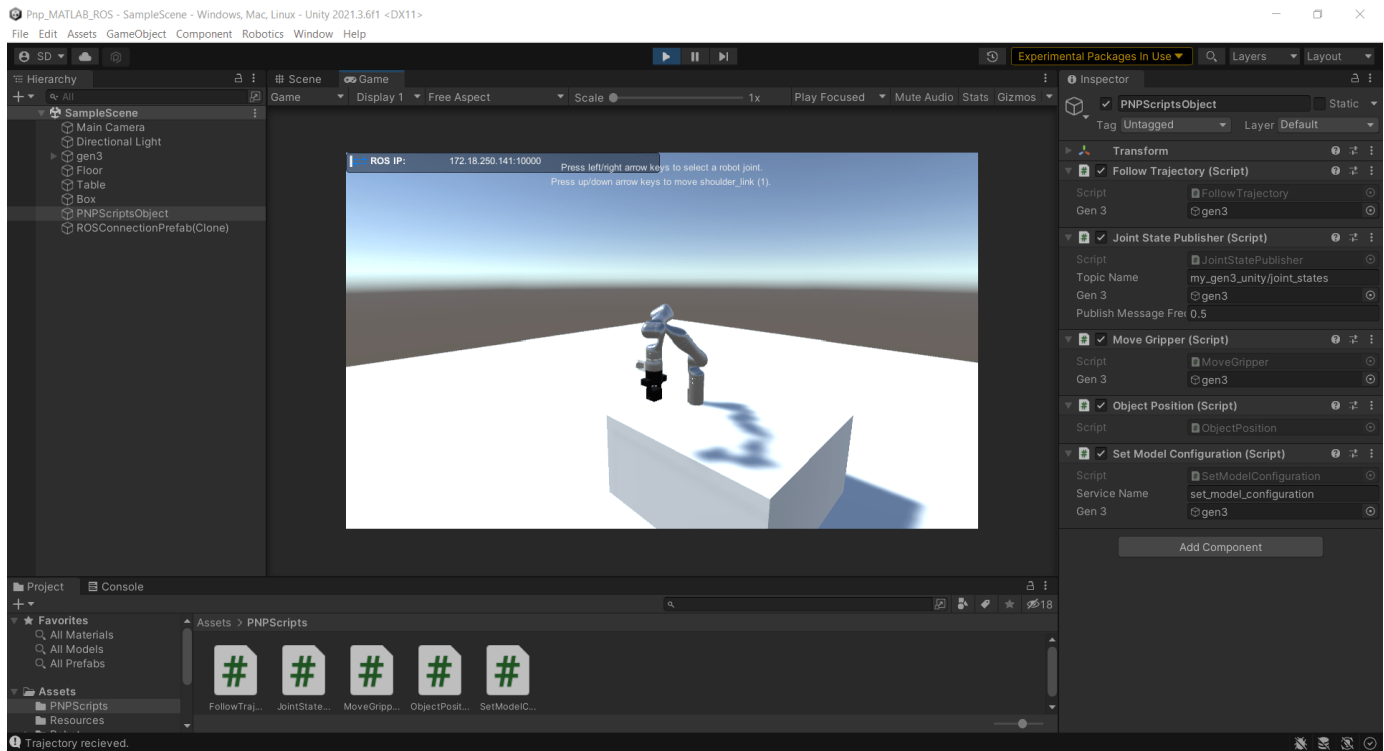
Ending Pick-and-Place Task

The Stateflow chart stops automatically after four failed attempts to detect new objects. To stop the pick-and-place task prematurely, uncomment and execute these lines of code or select the command window and press **Ctrl+C**.

```
% coordinator.FlowChart.endPickPlace;
% delete(coordinator.FlowChart)
% delete(coordinator)
```

Visualize Pick-and-Place Action in Unity

The Unity scene shows the robot in the working area as it moves the objects to the desired location. The robot continues working until all objects have been placed. When the detection step does not find any more objects four times, the Stateflow chart ends.

```
if strcmp(answer,"Yes")
    while  coordinator.NumDetectionRuns <  4
        % Wait for no objects to be detected.
    end
end
```

Kill the ROS TCP endpoint node and shut down the ROS network.

```
helperKillTCPEndpointNode;
rosshutdown
```

```
Shutting down global node /matlab_global_node_37032 with NodeURI http://hyd-shivarad:64473/ and N
Shutting down ROS master on http://172.18.250.141:11311.
```

Get Started with ROS

This example introduces how to set up ROS within MATLAB, and get information about ROS network and ROS messages.

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®.

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 5.7786 seconds.
Initializing ROS master on http://172.21.16.85:51063.
Initializing global node /matlab_global_node_13423 with NodeURI http://ah-avijayar:52604/ and Mas
```

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_13423
```

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_13423
/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.
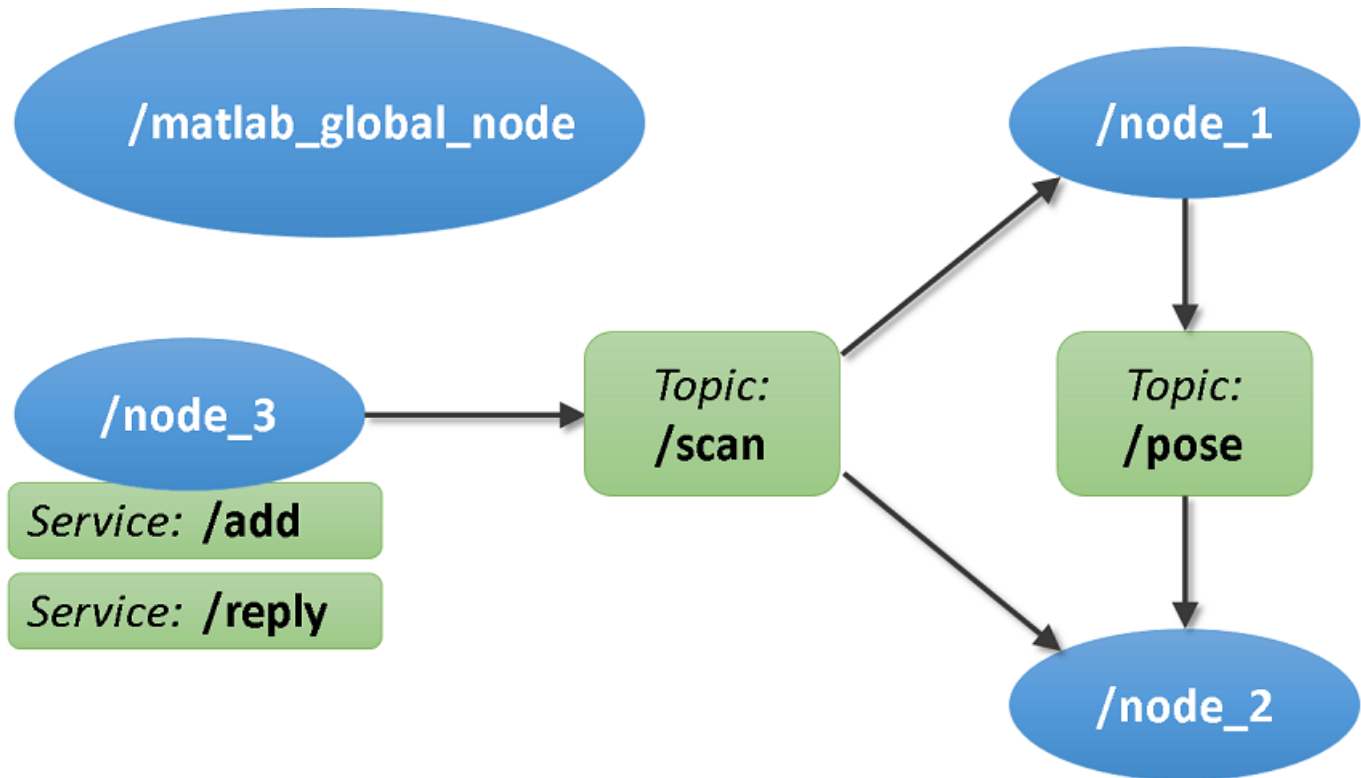
Topics

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 1-59 for more information.

```
rostopic info /pose
```

```
Type: geometry_msgs/Twist

Publishers:
* /node_1 (http://ah-avijayar:52609/)

Subscribers:
* /node_2 (http://ah-avijayar:52614/)
```

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-avijayar:52609/]

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 1-74).

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_13423/get_loggers
/matlab_global_node_13423/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-avijayar:52618
```

```
Type: roscpp_tutorials/TwoInts
Args: MessageType 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 1-52).

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_13423 with NodeURI http://ah-avijayar:52604/ and Ma
Shutting down ROS master on http://172.21.16.85:51063.
```

Next Steps

- "Connect to a ROS Network" on page 1-44

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 1-39

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

Launching ROS Core...
..Done in 2.7436 seconds.
Initializing ROS master on http://172.30.131.134:55031.
Initializing global node /matlab_global_node_70602 with NodeURI http://bat6234win64:55370/ and Ma
```

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_70602 with NodeURI http://bat6234win64:55370/ and N
Shutting down ROS master on http://172.30.131.134:55031.
```

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
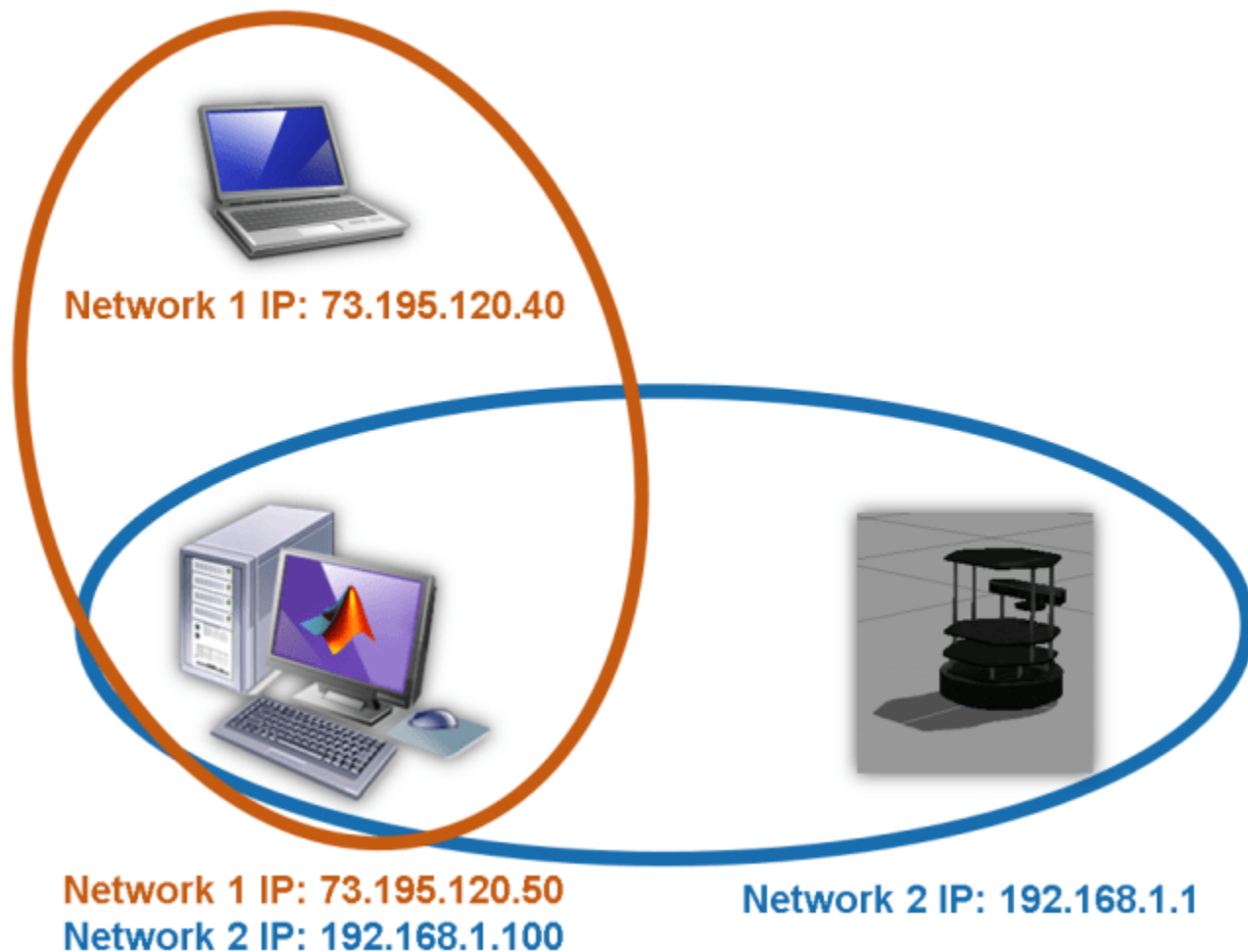
The computer on the bottom left runs MATLAB and is connected to two different networks. In one subnet, its IP address is `73.195.120.50`, and in the other, its IP is `192.168.1.100`. This computer wants to connect to the ROS master on the TurtleBot® computer at IP address `192.168.1.1`. As part of the 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.
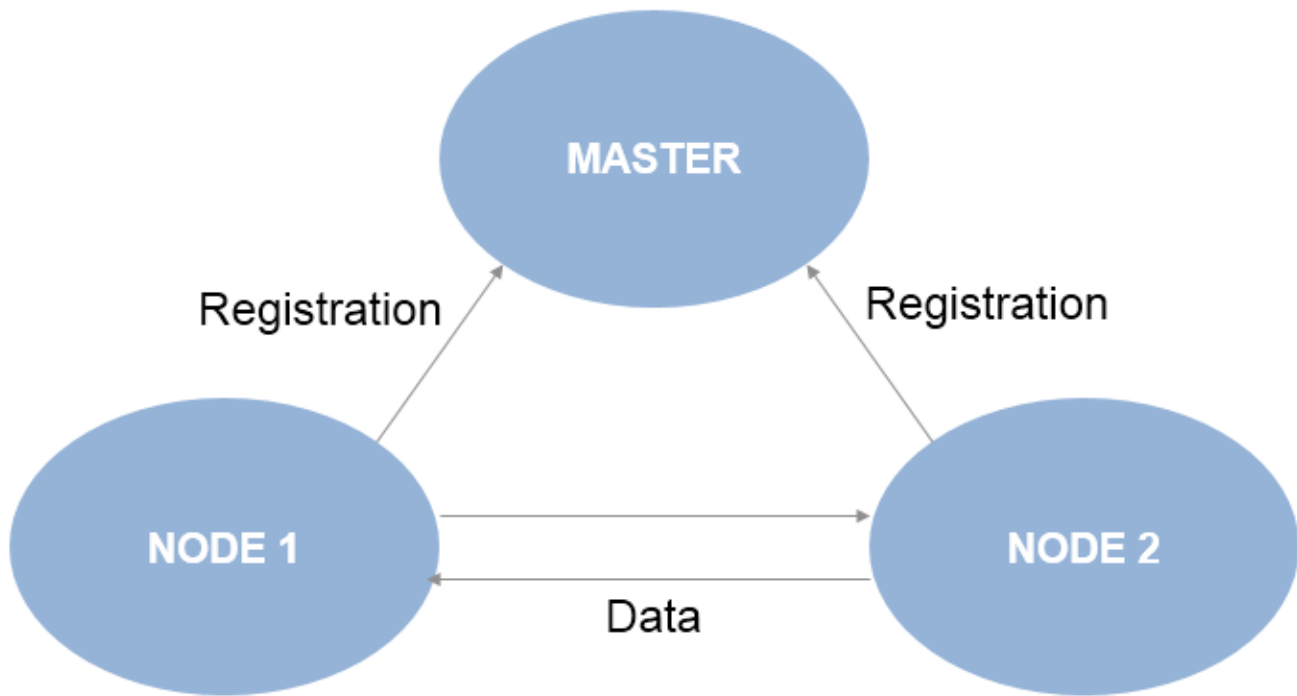
Next Steps

- See "Exchange Data with ROS Publishers and Subscribers" on page 1-59 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 1-39, "Connect to a ROS Network" on page 1-44

Create Parameter Tree

Start the ROS master and parameter server in MATLAB.

```
rosinit
```

```
Launching ROS Core...
..Done in 2.3345 seconds.
Initializing ROS master on http://172.30.131.134:58639.
Initializing global node /matlab_global_node_47220 with NodeURI http://bat6234win64:51805/ and Ma
```

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: {0x1 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 = 7x1 cell
    {'/MAX_SPEED'          }
    {'/ROBOT_IP'           }
    {'/goal'               }
    {'/myrobot/MAX_SPEED'  }
    {'/myrobot/ROBOT_IP'   }
    {'/myrobot/ROBOT_NAME' }
    {'/newrobot/ROBOT_NAME'}
```

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_NAME'}
```

Shut Down the ROS Network

Shut down the ROS master and delete the global node.

```
rosshutdown

Shutting down global node /matlab_global_node_47220 with NodeURI http://bat6234win64:51805/ and I
Shutting down ROS master on http://172.30.131.134:58639.
```

Next Steps

- For application examples, see the "Get Started with Gazebo and Simulated TurtleBot" on page 1-206 or "Get Started with a Real TurtleBot" on page 1-117 examples.

Work with Basic ROS Messages

This example shows various ways to create, explore, and populate ROS messages in MATLAB®, that are commonly encountered in robotics applications.

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 1-59 and "Call and Provide ROS Services" on page 1-74 for more information on topics and services)

Prerequisites: "Get Started with ROS" on page 1-39, "Connect to a ROS Network" on page 1-44

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:
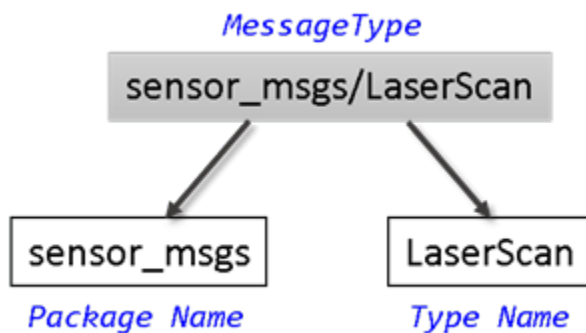
Find Message Types

Initialize the ROS master and global node.

```
rosinit
```

```
Launching ROS Core...
...Done in 3.6551 seconds.
Initializing ROS master on http://192.168.178.1:54553.
Initializing global node /matlab_global_node_11113 with NodeURI http://ah-csalzber:65499/
```

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
```

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

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

Type: sensor_msgs/LaserScan

Publishers:
* /node_3 (http://ah-csalzber:63490/)

Subscribers:
* /node_1 (http://ah-csalzber:63478/)
* /node_2 (http://ah-csalzber:63484/)
```

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 1-74.

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.

For better efficiency when creating messages or communicating, use messages in structure format.

```
scandata = rosmessage("sensor_msgs/LaserScan","DataFormat","struct")

scandata = struct with fields:
      MessageType: 'sensor_msgs/LaserScan'
           Header: [1×1 struct]
         AngleMin: 0
         AngleMax: 0
   AngleIncrement: 0
    TimeIncrement: 0
         ScanTime: 0
         RangeMin: 0
         RangeMax: 0
           Ranges: [0×1 single]
      Intensities: [0×1 single]
```

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` field, 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","DataFormat","struct")

posesub =
  Subscriber with properties:
```

```
      TopicName: '/pose'
  LatestMessage: []
    MessageType: 'geometry_msgs/Twist'
     BufferSize: 1
  NewMessageFcn: []
     DataFormat: 'struct'
```

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)
```

```
posedata = struct with fields:
    MessageType: 'geometry_msgs/Twist'
         Linear: [1×1 struct]
        Angular: [1×1 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:
    MessageType: 'geometry_msgs/Vector3'
              X: -0.0139
              Y: 0.0120
              Z: 0.0311
```

```
posedata.Angular
```

```
ans = struct with fields:
    MessageType: 'geometry_msgs/Vector3'
              X: -0.0481
              Y: -0.0416
              Z: 0.0475
```

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
```

```
xpos = -0.0139
```

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

```
rosShowDetails(posedata)
```

```
ans =
    '
```

```
MessageType :  geometry_msgs/Twist
Linear
  MessageType :  geometry_msgs/Vector3
  X           :  -0.01389779508053391
  Y           :  0.01202784270710855
  Z           :  0.03111508851002852
Angular
  MessageType :  geometry_msgs/Vector3
  X           :  -0.04807425225858586
  Y           :  -0.04161264917171002
  Z           :  0.04748016671848904'
```

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

Set Message Data

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

```
twist = rosmessage("geometry_msgs/Twist","DataFormat","struct")

twist = struct with fields:
    MessageType: 'geometry_msgs/Twist'
         Linear: [1×1 struct]
        Angular: [1×1 struct]
```

The numeric fields 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

ans = struct with fields:
    MessageType: 'geometry_msgs/Vector3'
              X: 0
              Y: 5
              Z: 0
```

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 1-59 and "Call and Provide ROS Services" on page 1-74 examples.

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:
    MessageType: 'geometry_msgs/Twist'
         Linear: [1×1 struct]
```

```
        Angular: [1×1 struct]
```

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')

messageData = struct with fields:
    posedata: [1×1 struct]
```

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

```
messageData.posedata

ans = struct with fields:
    MessageType: 'geometry_msgs/Twist'
         Linear: [1×1 struct]
        Angular: [1×1 struct]
```

You can now delete the MAT file.

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

Arrays in Messages

Some messages from ROS are stored in or contain arrays of other messages.

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 = struct with fields:
    MessageType: 'tf/tfMessage'
     Transforms: [1×53 struct]
```

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

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

```
tf.Transforms

ans=1×53 struct array with fields:
    MessageType
    Header
```

```
        ChildFrameId
        Transform
```

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}
```

```
cellTransforms=1×53 cell array
    {1×1 struct}    {1×1 struct}    {1×1 struct}    {1×1 struct}    {1×1 struct}    {1×1 struct}
```

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

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

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

```
ans = struct with fields:
      MessageType: 'geometry_msgs/TransformStamped'
           Header: [1×1 struct]
      ChildFrameId: '/imu_link'
        Transform: [1×1 struct]
```

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

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

```
ans = struct with fields:
    MessageType: 'geometry_msgs/Vector3'
              X: 0.0599
              Y: 0
              Z: -0.0141
```

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
```

```
Shutting down global node /matlab_global_node_11113 with NodeURI http://ah-csalzber:65499/
Shutting down ROS master on http://192.168.178.1:54553.
```

Next Steps

- See "Work with Specialized ROS Messages" on page 1-92 for examples of handling images, point clouds, and laser scan messages.

- For application examples, see the "Get Started with Gazebo and Simulated TurtleBot" on page 1-206 or "Get Started with a Real TurtleBot" on page 1-117 examples.

Exchange Data with ROS Publishers and Subscribers

This example shows how to publish and subscribe to topics in a ROS network.

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:
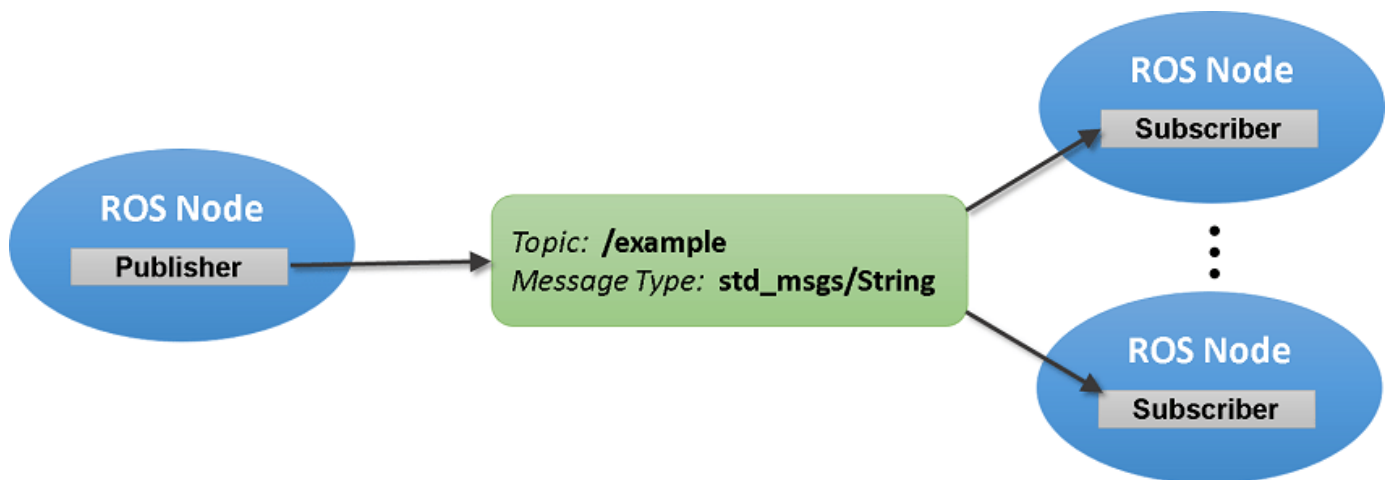
Besides how to publish and subscribe to topics in a ROS network, this example also shows how to:

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

Prerequisites: "Get Started with ROS" on page 1-39, "Connect to a ROS Network" on page 1-44

Subscribe and Wait for Messages

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

```
rosinit
```

```
Launching ROS Core...
.....Done in 5.4209 seconds.
Initializing ROS master on http://172.21.16.85:56764.
Initializing global node /matlab_global_node_69902 with NodeURI http://ah-avijayar:62300/ and Mas
```

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://ah-avijayar:62315/)

Subscribers:
* /node_1 (http://ah-avijayar:62305/)
* /node_2 (http://ah-avijayar:62310/)
```

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. Use messages in struct format for better efficiency.

```
laser = rossubscriber("/scan","DataFormat","struct");
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 = struct with fields:
      MessageType: 'sensor_msgs/LaserScan'
           Header: [1×1 struct]
         AngleMin: -0.5467
         AngleMax: 0.5467
   AngleIncrement: 0.0017
    TimeIncrement: 0
         ScanTime: 0.0330
         RangeMin: 0.4500
         RangeMax: 10
           Ranges: [640×1 single]
      Intensities: []
```

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

```
figure
rosPlot(scandata,"MaximumRange",7)
```
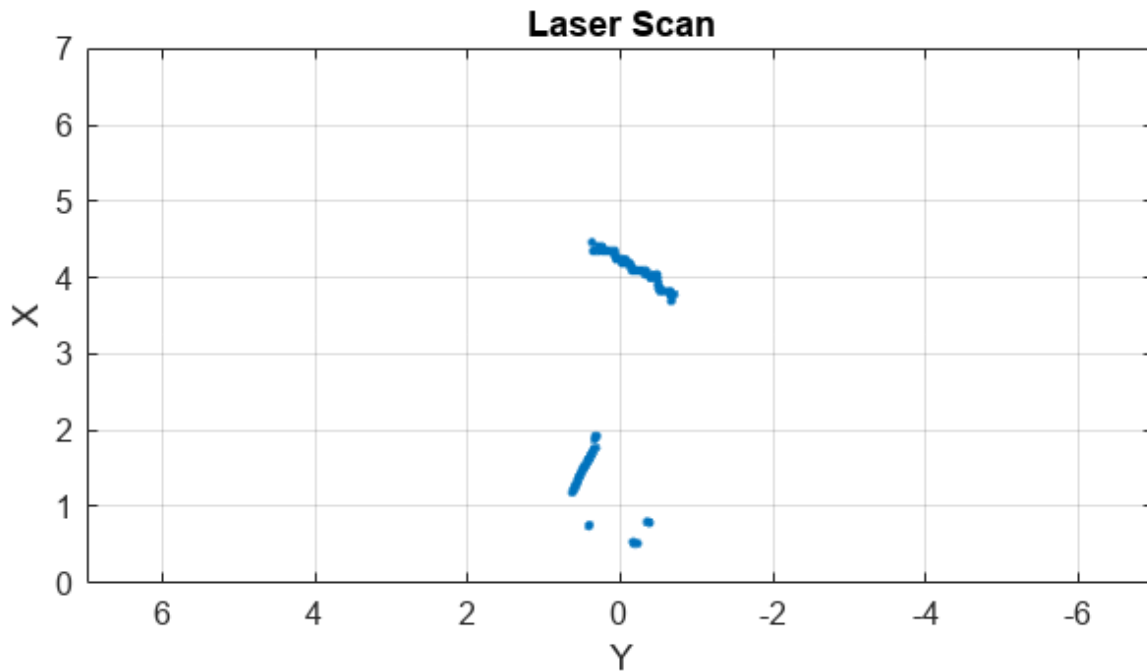
Subscribe Using Callback Functions

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,"DataFormat","struct")
```

```
robotpose =
  Subscriber with properties:

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

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

   -0.0292   -0.0199   -0.0029
```

```
orient
```

```
orient = 1×3

   -0.0270    0.0344   -0.0305
```

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 "Create Callbacks for Graphics Objects" 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 1-52).

```
chatterpub = rospublisher("/chatter","std_msgs/String","DataFormat","struct")
```

```
chatterpub =
  Publisher with properties:

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

```
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 = struct with fields:
    MessageType: 'std_msgs/String'
           Data: 'hello world'
```

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,"DataFormat","struct")

chattersub =
  Subscriber with properties:

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

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_69902 with NodeURI http://ah-avijayar:62300/ and Ma
Shutting down ROS master on http://172.21.16.85:56764.
```

Next Steps

- To learn more about how ROS messages are handled in MATLAB, see "Work with Basic ROS Messages" on page 1-52 and "Work with Specialized ROS Messages" on page 1-92.
- To explore ROS services, refer to "Call and Provide ROS Services" on page 1-74.

Improve Performance of ROS Using Message Structures

This example demonstrates the use of ROS message structures, and their benefits and differences from message objects.

Message structures have better performance over objects when performing initial creation, reading them from rosbag files, accessing nested properties, and performing communication operations over the ROS network. Also, message structures are the only supported message format when generating code through MATLAB Coder™.

Message Structure Basics

ROS message objects are instances of classes defined specifically for each message type.

```
msgObj = rosmessage("nav_msgs/Path");
class(msgObj)

ans =
'ros.msggen.nav_msgs.Path'
```

The object properties contain the data of the message, and each object type has functions defined that are specific to the ROS message.

```
showdetails(msgObj)

  Header
    Stamp
      Sec  :  0
      Nsec :  0
    Seq      :  0
    FrameId :
  Poses
```

ROS message structures have been introduced to improve the performance of using ROS messages. Each message is a MATLAB® structure data type with the same fields as the properties of the ROS message objects.

```
msgStruct = rosmessage("nav_msgs/Path","DataFormat","struct")

msgStruct = struct with fields:
    MessageType: 'nav_msgs/Path'
         Header: [1x1 struct]
          Poses: [0x1 struct]


class(msgStruct)

ans =
'struct'
```

Update Existing Code to Use Structures

To update existing code that uses objects, two common workflows are provided with the steps required to update them.

Communication Workflow

This example code shows how to send and receive messages over the ROS network.

```
% Setup ROS network
rosinit

Launching ROS Core...
..Done in 2.3876 seconds.
Initializing ROS master on http://172.30.131.134:60399.
Initializing global node /matlab_global_node_46994 with NodeURI http://bat6234win64:56496/ and M

stringPub = rospublisher("/chatter","std_msgs/String");
stringSub = rossubscriber("/chatter","std_msgs/String");

% Set message field and send message
stringMsg = rosmessage("std_msgs/String");
stringMsg.Data = 'Hello World!';
send(stringPub,stringMsg)

% Wait for message to be received and then check the value
pause(2)
showdetails(stringSub.LatestMessage)

  Data :  Hello World!
```

How to Update

Set the data format name-value argument of the publisher and subscriber.

```
stringPub = rospublisher("/chatter","std_msgs/String","DataFormat","struct");
stringSub = rossubscriber("/chatter","std_msgs/String","DataFormat","struct");
```

Update the data format for the `rosmessage` function as well.

```
stringMsg = rosmessage("std_msgs/String","DataFormat","struct");
stringMsg.Data = 'Hello World!';
send(stringPub,stringMsg)
```

Alternatively, the `rosmessage` object function for the publisher can be used. This syntax produces a message that follows the format set in the publisher, and is the most efficient way to ensure compatibility between the message and the publisher.

```
stringMsg = rosmessage(stringPub);
stringMsg.Data = 'Hello World!';
send(stringPub,stringMsg)
```

Because structures do not have object functions, new functions are provided to handle common ROS message tasks. To show details a structure message, use the `rosShowDetails` function. To see all the new functions provided, go to Message Handling Functions on page 1-67.

```
% Wait for message to be received and then check the value
pause(2)
rosShowDetails(stringSub.LatestMessage)

ans =
    '
```

```
        MessageType :  std_msgs/String
        Data        :  Hello World!'
```

Read rosbag Workflow

For an example that reads messages from a `rosbag`, specifiy the `DataFormat` name-value argument for the `readMessages` function and any publishers you use to send those messages.

```matlab
% Extract message from rosbag
msgType = "nav_msgs/Odometry";
bag = rosbag("ex_multiple_topics.bag");
bagSelect = select(bag,"MessageType",msgType);
odomMsgs = readMessages(bagSelect,"DataFormat","struct");
odomMsg = odomMsgs{1}

odomMsg = struct with fields:
    MessageType: 'nav_msgs/Odometry'
         Header: [1x1 struct]
    ChildFrameId: 'base_footprint'
           Pose: [1x1 struct]
          Twist: [1x1 struct]


% Create publisher and send first message
odomPub = rospublisher("/odom",msgType,"DataFormat","struct");
send(odomPub,odomMsg)
```

Message Handling Functions

Because functions on the ROS message objects are not usable with message structures, new functions have been introduced for handling messages. This list includes functions for reading data from or writing data to specialized messages.

Message Type	Object method	Structure Function
All messages	showdetails	rosShowDetails
	definition	rosmsg("show",____)
sensor_msgs/Image sensor_msgs/CompressedImage	readImage	rosReadImage
sensor_msgs/Image	writeImage	rosWriteImage
sensor_msgs/LaserScan	readCartesian	rosReadCartesian
	readScanAngles	rosReadScanAngles
	lidarScan	rosReadLidarScan
	plot	rosPlot
sensor_msgs/PointCloud2	readXYZ	rosReadXYZ
	readRGB	rosReadRGB
	readAllFieldNames	rosReadAllFieldNames
	readField	rosReadField
	scatter3	rosPlot
geometry_msgs/Quaternion	readQuaternion	rosReadQuaternion
nav_msgs/OccupancyGrid	readBinaryOccupancyGrid	rosReadBinaryOccupancyGrid
	readOccupancyGrid	rosReadOccupancyGrid
	writeBinaryOccupancyGrid	rosWriteBinaryOccupancyGrid
	writeOccupancyGrid	rosWriteOccupancyGrid
octomap_msgs/Octomap	readOccupancyMap3D	rosReadOccupancyMap3D
geometry_msgs/TransformStamped	apply	rosApplyTransform

Behavior Changes

Handle Class vs. Structure Behavior

An important consideration when converting code is that ROS message objects are handles, which means that message objects are passed by reference when provided as inputs to functions. If a message is modified within a function, the modification applies to the message in the MATLAB® workspace as well.

```
msgObj = rosmessage("geometry_msgs/Pose2D");
pose = [1 2 3];
exampleHelperWritePoseToMsgObj(msgObj,pose)
disp(msgObj)

  ROS Pose2D message with properties:

    MessageType: 'geometry_msgs/Pose2D'
              X: 1
              Y: 2
          Theta: 3

  Use showdetails to show the contents of the message

function exampleHelperWritePoseToMsgObj(pointMsg,pose)
pointMsg.X = pose(1);
pointMsg.Y = pose(2);
pointMsg.Theta = pose(3);
end
```

Message structures only pass their value when input into functions. If a message structure is modified within a function, that modification will only apply to the structure within the scope of that function. To make the modification available outside of the function, the message structure must be returned.

```
msgStruct = rosmessage("geometry_msgs/Pose2D","DataFormat","struct");
pose = [1 2 3];

% With no return, the message structure will not change
exampleHelperWritePoseToMsgObj(msgStruct, pose)
disp(msgStruct)

    MessageType: 'geometry_msgs/Pose2D'
              X: 0
              Y: 0
          Theta: 0

% When returned from the function, the message can be overwritten.
msgStruct = exampleHelperWritePoseToMsgStruct(msgStruct, pose);
disp(msgStruct)

    MessageType: 'geometry_msgs/Pose2D'
              X: 1
              Y: 2
          Theta: 3

function pointMsg = exampleHelperWritePoseToMsgStruct(pointMsg,pose)
pointMsg.X = pose(1);
pointMsg.Y = pose(2);
pointMsg.Theta = pose(3);
end
```

This applies to the specialized message handling functions as well. The write functions that update message values now have output arguments to supply the updated message structure.

```
image = imread('imageMap.png');

% Message object
msg = rosmessage("sensor_msgs/Image");
msg.Encoding = 'rgb8';
writeImage(msg,image)
imshow(readImage(msg))
```
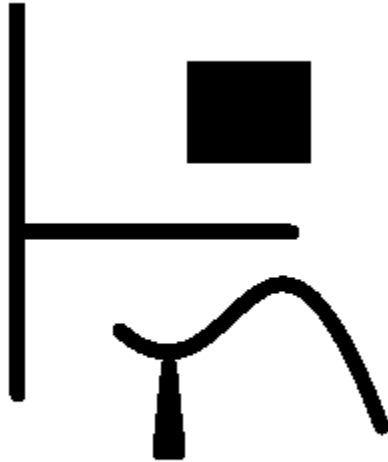
```
% Message structure
msg = rosmessage("sensor_msgs/Image","DataFormat","struct");
msg.Encoding = 'rgb8';
msg = rosWriteImage(msg,image);
imshow(rosReadImage(msg))
close
```

Time and Duration Arithmatic

ROS time and duration message structures are unable to support operator overloading in the same way that the time and duration objects do. Arithmetic and comparison operations should be done by converting the time or duration structures to a numerical seconds value, performing the operation, and then recreating the time or duration structure if necessary.

```
% Periodically update message timestamp with objects
msg = rosmessage("std_msgs/Header");
runFor = rosduration(2);
tNow = rostime("now");
tEnd = tNow + runFor;
while tNow < tEnd
    msg.Stamp = tNow;
    % Message may be sent here
    pause(1)
    tNow = rostime("now");
end
% Periodically update message timestamp with structures
msg = rosmessage("std_msgs/Header","DataFormat","struct");
runFor = 2;
tNow = rostime("now","DataFormat","struct");
tNowSec = tNow.Sec + tNow.Nsec*1e-9;
tEndSec = tNowSec + runFor;
while tNowSec < tEndSec
```

```
    msg.Stamp = tNow;
    % Message may be sent here
    pause(1)
    tNow = rostime("now","DataFormat","struct");
    tNowSec = tNow.Sec + tNow.Nsec*1e-9;
end
```

Data Field Coercion

With ROS message objects, data fields have specific types. When a data field value is set, the input is converted to the correct type if possible. Otherwise, if conversion is not possible, an error is returned.

```
msg = rosmessage("std_msgs/Int8");
msg.Data = 20;
class(msg.Data)

ans =
'int8'
```

ROS message structures inherently accept any data type or field name without error.

```
msg = rosmessage("std_msgs/Int8","DataFormat","struct");
msg.Data = 'Test'

msg = struct with fields:
    MessageType: 'std_msgs/Int8'
           Data: 'Test'


msg.Data = 20;
class(msg.Data)

ans =
'double'
```

Instead, invalid data types error when attempting to send the message over the ROS network.

```
pub = rospublisher("/int_topic","std_msgs/Int8","DataFormat","struct");
send(pub,msg)

Error using ros.Publisher/send (line 290)
Error publishing a message with type std_msgs/Int8 on topic name /int_topic.

Caused by:
    Error using ros.internal.Node/publish
    Field 'Data' is wrong type; expected a int8.
```

To prevent errors, ensure that messages are are using the correct data type from the message definition.

```
rosmsg show std_msgs/Int8

int8 Data
```

Other Performance Tips

Using message structures is a good first step to speed up the sending and retrieving of ROS messages. Structures also improves the performance for setting and accessing data in nested mesages.The following code demonstrates sending multiple messages with nested fields.

```matlab
% Set up network (reuse publisher for all examples)
pub = rospublisher("/goal_path","nav_msgs/Path","DataFormat","struct");

% Send robot new paths to follow
nPtsOnPath = 100;
for iPaths = 1:15
    pathMsg = rosmessage(pub);
    for iPts = 1:nPtsOnPath
        pathMsg.Poses(iPts) = rosmessage("geometry_msgs/PoseStamped","DataFormat","struct");
        pathMsg.Poses(iPts).Pose.Position.X = iPaths+iPts;
        pathMsg.Poses(iPts).Pose.Position.Y = iPaths-iPts;
        pathMsg.Poses(iPts).Pose.Position.Z = (iPaths+iPts)/10;
    end
    send(pub,pathMsg)
end
```

Reuse Messages

If messages are being created and modified in a loop, and the same data fields are being set each iteration, it is faster to create the message only once. Move the creation of the messages outside the loop, and reuse the same messages inside the loop for each iteration.

```matlab
% Set up messages for use
pathMsg = rosmessage(pub);
poseMsg = rosmessage("geometry_msgs/PoseStamped","DataFormat","struct");

% Send robot new paths to follow
nPtsOnPath = 100;
for iPaths = 1:15
    for iPts = 1:nPtsOnPath
        pathMsg.Poses(iPts) = poseMsg;
        pathMsg.Poses(iPts).Pose.Position.X = iPaths+iPts;
        pathMsg.Poses(iPts).Pose.Position.Y = iPaths-iPts;
        pathMsg.Poses(iPts).Pose.Position.Z = (iPaths+iPts)/10;
    end
    send(pub,pathMsg)
end
```

Extract Nested Messages for Manipulation

When reading or setting multiple fields in a nested message, extract the nested message before reading or setting the fields.

```matlab
% Set up messages for use
pathMsg = rosmessage(pub);
poseMsg = rosmessage("geometry_msgs/PoseStamped","DataFormat","struct");
ptMsg = poseMsg.Pose.Position;  % Extract nested message

% Send robot new paths to follow
nPtsOnPath = 100;
for iPaths = 1:15
    for iPts = 1:nPtsOnPath
        % Set fields before setting nested message
        ptMsg.X = iPaths+iPts;
        ptMsg.Y = iPaths-iPts;
        ptMsg.Z = (iPaths+iPts)/10;
        poseMsg.Pose.Position = ptMsg;
        pathMsg.Poses(iPts) = poseMsg;
```

```
    end
    send(pub,pathMsg)
end
```

Preallocate Message Struct Arrays

For relatively large arrays of messages, preallocating a structure array can improve performance when setting values in a loop. Use this method when the the array is a fixed length every iteration.

```
% Set up messages for use
pathMsg = rosmessage(pub);
poseMsg = rosmessage("geometry_msgs/PoseStamped","DataFormat","struct");
ptMsg = poseMsg.Pose.Position;  % Extract nested message

% Preallocate path array
nPtsOnPath = 100;
pathMsg.Poses(nPtsOnPath) = poseMsg;

% Send robot new paths to follow
for iPaths = 1:15
    for iPts = 1:nPtsOnPath
        % Set fields before setting nested message
        ptMsg.X = iPaths+iPts;
        ptMsg.Y = iPaths-iPts;
        ptMsg.Z = (iPaths+iPts)/10;
        poseMsg.Pose.Position = ptMsg;
        pathMsg.Poses(iPts) = poseMsg;
    end
    send(pub,pathMsg)
end
```

The ROS network can now be shut down.

```
rosshutdown
```

```
Shutting down global node /matlab_global_node_46994 with NodeURI http://bat6234win64:56496/ and N
Shutting down ROS master on http://172.30.131.134:60399.
```

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 1-59). *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 1-39, "Connect to a ROS Network" on page 1-44, "Exchange Data with ROS Publishers and Subscribers" on page 1-59

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
```

```
Launching ROS Core...
...Done in 3.16 seconds.
Initializing ROS master on http://192.168.178.1:55438.
Initializing global node /matlab_global_node_55791 with NodeURI http://ah-csalzber:51640/
```

```
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`.

For faster performance, use services with messages in structure format.

```
testserver = rossvcserver("/test","std_srvs/Empty",@exampleHelperROSEmptyCallback,"DataFormat","s
```

```
testserver =
  ServiceServer with properties:

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

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

```
rosservice list
```

```
/add
/matlab_global_node_55791/get_loggers
/matlab_global_node_55791/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
/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_55791
URI: rosrpc://ah-csalzber:51639
Type: std_srvs/Empty
Args: MessageType
```

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","DataFormat","struct")

testclient =
  ServiceClient with properties:

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

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 and format that is specified by the service.

```
testreq = rosmessage(testclient)

testreq = struct with fields:
    MessageType: 'std_srvs/EmptyRequest'
```

Ensure that the service is connected to the client, waiting for them to connect if necessary.

```
waitForServer(testclient,"Timeout",3)
```

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);
```

If the call function above fails, it will error. Instead of an error, if you would prefer to react to a call failure using conditionals, return the `status` and `statustext` outputs from the call function. The `status` output indicates if the call succeeded, while `statustext` provides additional information. Similar outputs can be returned from `waitForServer`.

```
numCallFailures = 0;
[testresp,status,statustext] = call(testclient,testreq,"Timeout",3);
if ~status
    numCallFailures = numCallFailues + 1;
    fprintf("Call failure number %d. Error cause: %s\n",numCallFailures,statustext)
else
    disp(testresp)
end

    MessageType: 'std_srvs/EmptyResponse'
```

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,"DataForm
```

```
sumserver =
  ServiceServer with properties:

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

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","DataFormat","struct")
```

```
sumclient =
  ServiceClient with properties:

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

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 = int64(2);
sumreq.B = int64(1)
```

```
sumreq = struct with fields:
    MessageType: 'roscpp_tutorials/TwoIntsRequest'
              A: 2
```

```
        B: 1
```

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. Ensure that the service server is available before making a call, and react appropriately if it is not.

```
if isServerAvailable(sumclient)
    sumresp = call(sumclient,sumreq,"Timeout",3)
else
    error("Service server not available on network")
end
```

```
sumresp = struct with fields:
    MessageType: 'roscpp_tutorials/TwoIntsResponse'
            Sum: 3
```

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_55791 with NodeURI http://ah-csalzber:51640/
Shutting down ROS master on http://192.168.178.1:55438.
```

Next Steps

- Refer to "Work with Basic ROS Messages" on page 1-52 to explore how ROS messages are represented in MATLAB.

Work with rosbag Logfiles

This example enables you to load a rosbag, and learn how to select and retrieve the contained messages.

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 3-14 for more information about rosbag support in MATLAB.

Prerequisites: "Work with Basic ROS Messages" on page 1-52

Load a rosbag

Load an example file using the `rosbag` command.

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

bag =
  BagSelection with properties:

            FilePath: 'C:\Users\csalzber\OneDrive - MathWorks\Documents\MATLAB\Examples\ros-ex7148
           StartTime: 201.3400
             EndTime: 321.3400
         NumMessages: 36963
      AvailableTopics: [4×3 table]
      AvailableFrames: {0×1 cell}
          MessageList: [36963×4 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

                          _____    _____    _____

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

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:\Users\csalzber\OneDrive - MathWorks\Documents\MATLAB\Examples\ros-ex7148
           StartTime: 201.3400
             EndTime: 321.3300
         NumMessages: 11998
      AvailableTopics: [1×3 table]
      AvailableFrames: {0×1 cell}
          MessageList: [11998×4 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:\Users\csalzber\OneDrive - MathWorks\Documents\MATLAB\Examples\ros-ex7148
           StartTime: 201.3400
             EndTime: 231.3200
         NumMessages: 2997
      AvailableTopics: [1×3 table]
      AvailableFrames: {0×1 cell}
          MessageList: [2997×4 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:\Users\csalzber\OneDrive - MathWorks\Documents\MATLAB\Examples\ros-ex7148
           StartTime: 205.0200
             EndTime: 205.9900
         NumMessages: 101
```

```
     AvailableTopics: [1×3 table]
     AvailableFrames: {0×1 cell}
         MessageList: [101×4 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/LaserScan"
bagselect4 = select(bag,selectOptions{:})
```

```
bagselect4 =
  BagSelection with properties:

           FilePath: 'C:\Users\csalzber\OneDrive - MathWorks\Documents\MATLAB\Examples\ros-ex7148
          StartTime: 201.3400
            EndTime: 207.3300
        NumMessages: 209
     AvailableTopics: [2×3 table]
     AvailableFrames: {0×1 cell}
         MessageList: [209×4 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 improve time efficiency, read the messages from the rosbag in structure format.

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

```
msgs = readMessages(bagselect3,"DataFormat","struct");
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],"DataFormat","struct")
```

```
msgs=4×1 cell array
    {1×1 struct}
    {1×1 struct}
    {1×1 struct}
    {1×1 struct}
```

```
msgs{2}
```

```
ans = struct with fields:
     MessageType: 'nav_msgs/Odometry'
          Header: [1×1 struct]
     ChildFrameId: 'base_footprint'
            Pose: [1×1 struct]
           Twist: [1×1 struct]
```

Each message in the cell array is a standard MATLAB ROS message structure. For more information on messages, see the "Work with Basic ROS Messages" on page 1-52 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: [1x1 tsdata.timemetadata]
            Data: [101x2 double]
        DataInfo: [1x1 tsdata.datametadata]

  More properties, Methods
```

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)
```
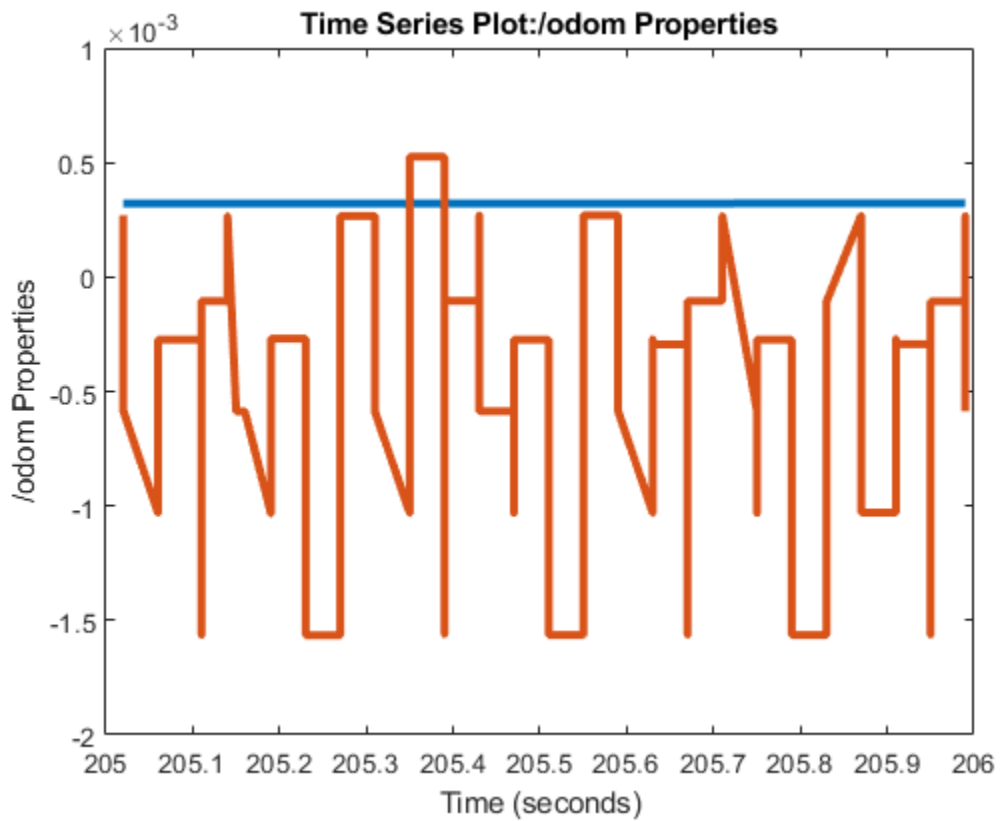
Access the tf Transformation Tree in ROS

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 1-39, "Connect to a ROS Network" on page 1-44

Start up

Initialize the ROS system.

```
rosinit
```

```
Launching ROS Core...
..Done in 2.6282 seconds.
Initializing ROS master on http://172.30.131.134:54252.
Initializing global node /matlab_global_node_86339 with NodeURI http://bat6234win64:64836/ and Ma
```

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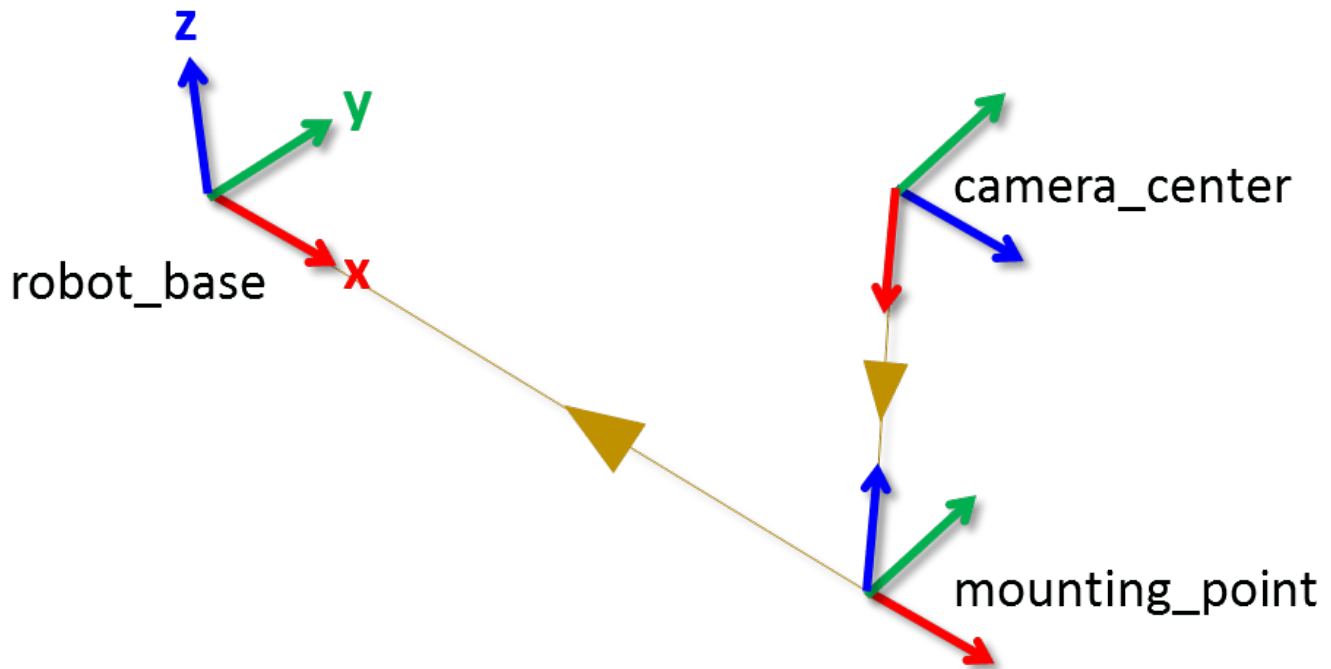
Here, the x, y, and z axes of each frame are represented by red, green, and blue lines respectively. The parent-child relationship between the coordinate frames is shown through a brown arrow pointing from the child to its parent frame.

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
         DataFormat: 'object'
```

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.
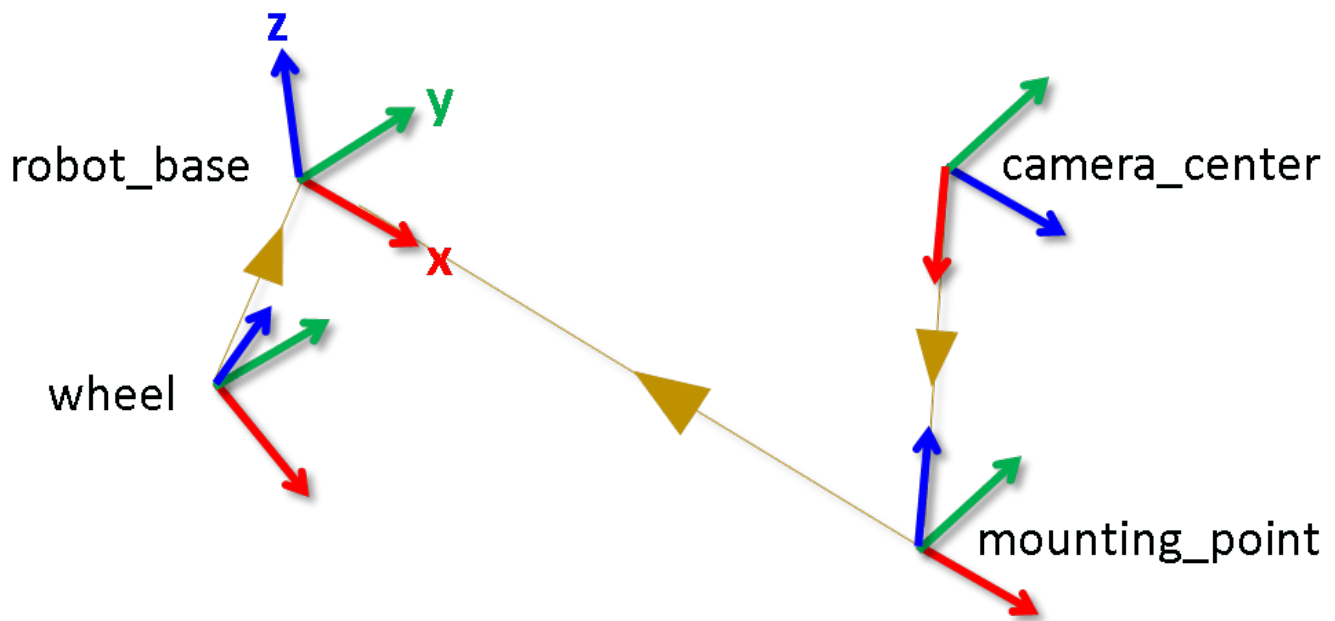
You can see that the `wheel` coordinate frame has the translation and rotation that you specified in sending the transformation.

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_86339 with NodeURI http://bat6234win64:64836/ and I
Shutting down ROS master on http://172.30.131.134:54252.
```

Work with Specialized ROS Messages

This example shows how to handle message types for laser scans, uncompressed and compressed images, point clouds, camera info, occcupancy grid, and octomap messages.

These are some commonly used ROS messages stored 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.

Prerequisites: "Work with Basic ROS Messages" on page 1-52

Load Sample Messages

Load some sample messages. These messages are populated with real and synthetic data from various robotics sensors.

```
load("specialROSMessageData.mat")
```

Image Messages

MATLAB 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",DataFormat="struct")
```

```
emptyimg = struct with fields:
    MessageType: 'sensor_msgs/Image'
         Header: [1×1 struct]
         Height: 0
          Width: 0
       Encoding: ''
    IsBigendian: 0
           Step: 0
           Data: [0×1 uint8]
```

For convenience, an image message that is fully populated and is stored in the `img` variable is loaded from `specialROSMessageData.mat`.

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 = struct with fields:
    MessageType: 'sensor_msgs/Image'
         Header: [1×1 struct]
         Height: 480
          Width: 640
       Encoding: 'rgb8'
    IsBigendian: 0
           Step: 1920
```

```
Data: [921600×1 uint8]
```

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

```
imageFormatted = rosReadImage(img);
```

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

```
imshow(imageFormatted)
```
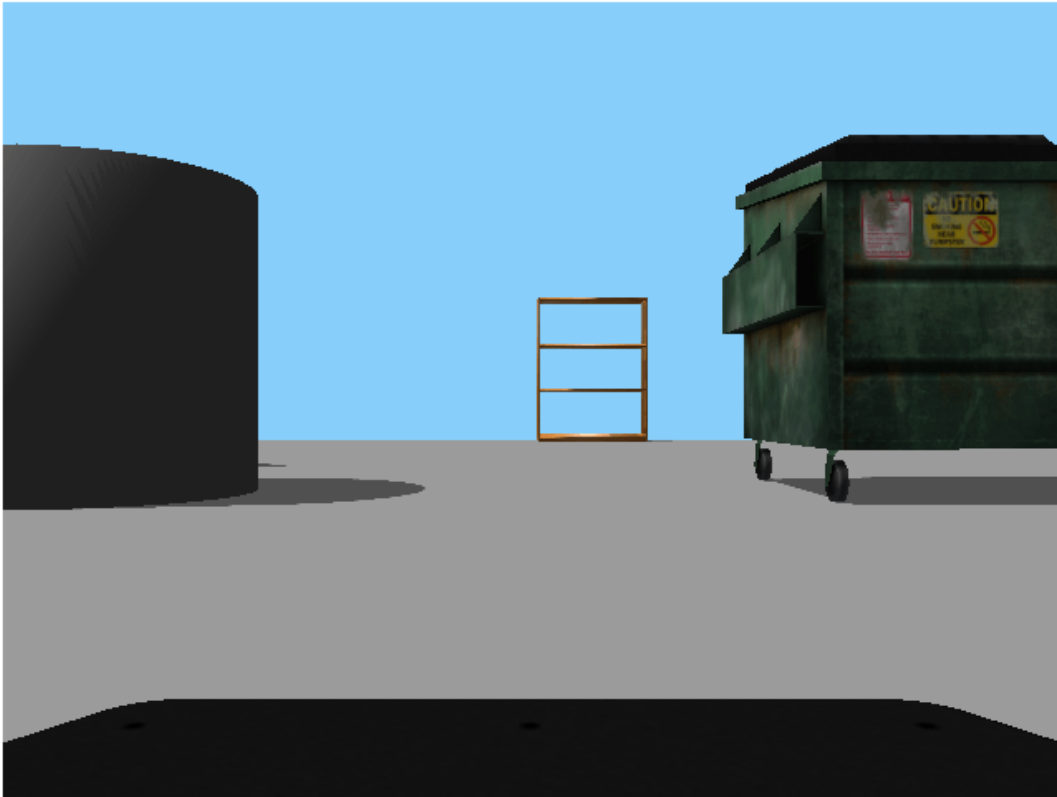

MATLAB® supports all ROS image encoding formats, and `rosReadImage` handles the complexity of converting the image data. In addition to color images, MATLAB also supports monochromatic and depth images.

Additionally, MATLAB provides the `rosWriteImage` function to convert a MATLAB image to a ROS message using the function. Apply rudimentary object detection on the sample image with color thresholding. Visualize the modified image.

```
greenPercentage = 100*double(imageFormatted(:,:,2))./sum(imageFormatted,3);
thresholdImg = 255*uint8(greenPercentage > 35);
imshow(thresholdImg)
```
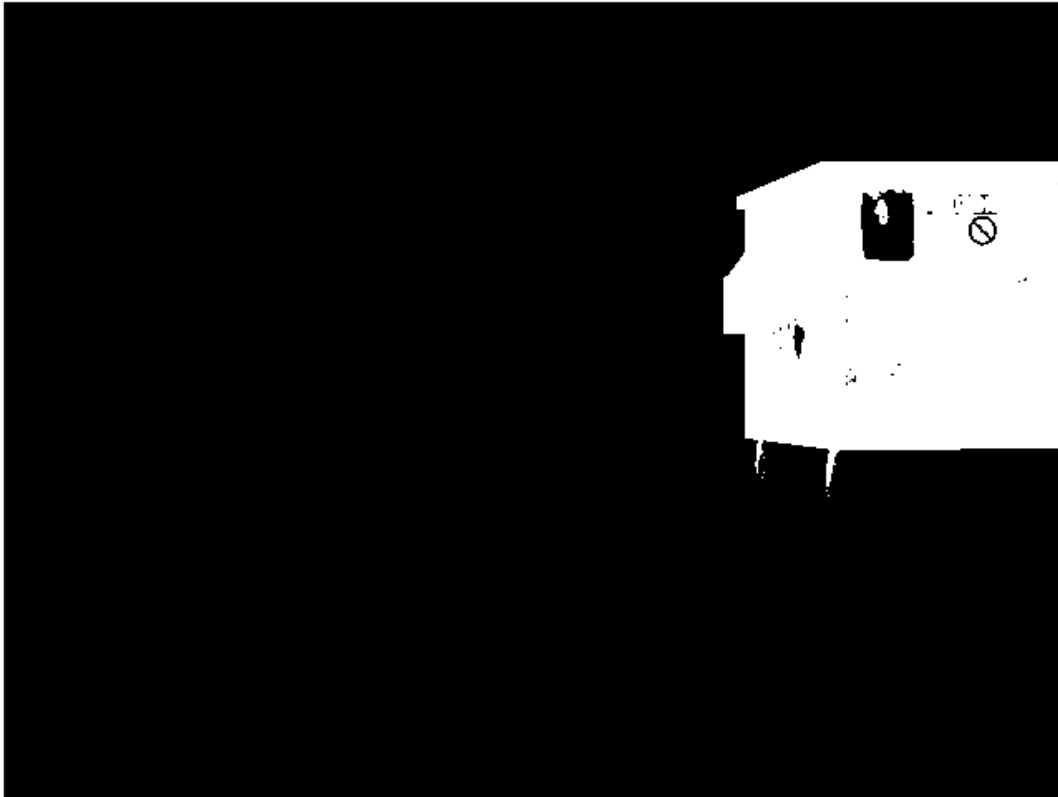

Write the modified image to a ROS message with the `rosWriteImage` function. Since the modified image only has 1 channel and is of type `uint8`, use the `mono8` encoding.

```
imageMsg = rosWriteImage(emptyimg,thresholdImg,Encoding="mono8");
```

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",DataFormat="struct")
```

```
emptyimgcomp = struct with fields:
    MessageType: 'sensor_msgs/CompressedImage'
         Header: [1×1 struct]
         Format: ''
```

```
        Data: [0×1 uint8]
```

For convenience, a compressed image message that is already populated was loaded from `specialROSMessageData.mat`.

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 = struct with fields:
    MessageType: 'sensor_msgs/CompressedImage'
         Header: [1×1 struct]
         Format: 'bgr8; jpeg compressed bgr8'
           Data: [30376×1 uint8]
```

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

```
compressedFormatted = rosReadImage(imgcomp);
```

Visualize the image using the `imshow` function.

```
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",DataFormat="struct")
```

```
emptyptcloud = struct with fields:
    MessageType: 'sensor_msgs/PointCloud2'
         Header: [1×1 struct]
         Height: 0
          Width: 0
         Fields: [0×1 struct]
```

```
        IsBigendian: 0
          PointStep: 0
            RowStep: 0
               Data: [0×1 uint8]
            IsDense: 0
```

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

```
ptcloud
```

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

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 `rosReadXYZ` function.

```
xyz = rosReadXYZ(ptcloud)
```

```
xyz = 307200×3 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 = 193359×3 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 rosReadRGB.

rgb = rosReadRGB(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 rosPlot function. rosPlot 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 rosPlot function ignores all NaN x,y,z coordinates, even if RGB values exist for that point.

rosPlot(ptcloud)
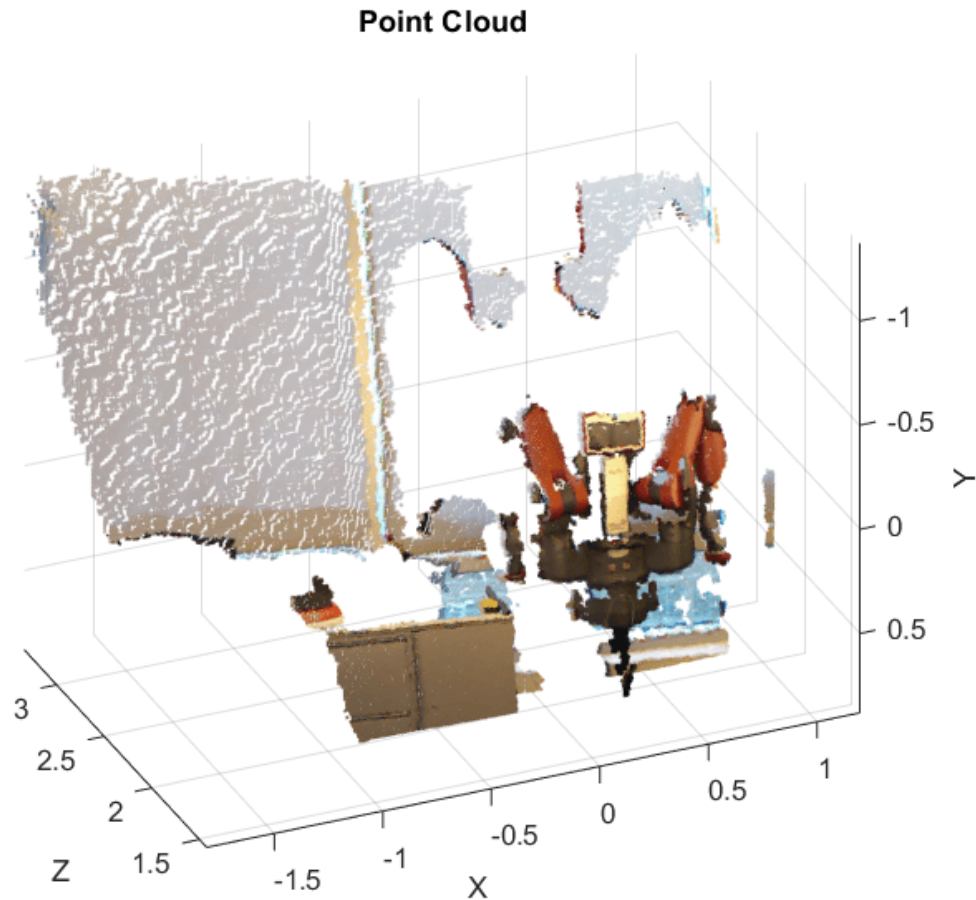
Point Cloud

Examine all the stored fields in the point cloud message using the `rosReadAllFieldNames` function. The loaded point cloud message contains four fields x, y, z, and rgb.

```
fieldNames = rosReadAllFieldNames(ptcloud)

fieldNames = 1×4 cell
    {'x'}    {'y'}    {'z'}    {'rgb'}
```

You can access the corresponding data for any field using the `rosReadField` function. You must to unpack the returned data manually, depending on how it is formatted. For example, the RGB image can be extracted by type casting the data to `uint8` and reshaping the result. Use the result from the `rosReadAllFieldNames` function for input validation.

```
if any(contains(fieldNames,"rgb"))
    rawData = typecast(rosReadField(ptcloud,"rgb"),"uint8");
    tmp = reshape(permute(reshape(rawData,4,[]),[3,2,1]),ptcloud.Width,ptcloud.Height,4);
    pcImg = permute(tmp(:,:,[3,2,1]),[2 1 3]);
    imshow(pcImg)
end
```
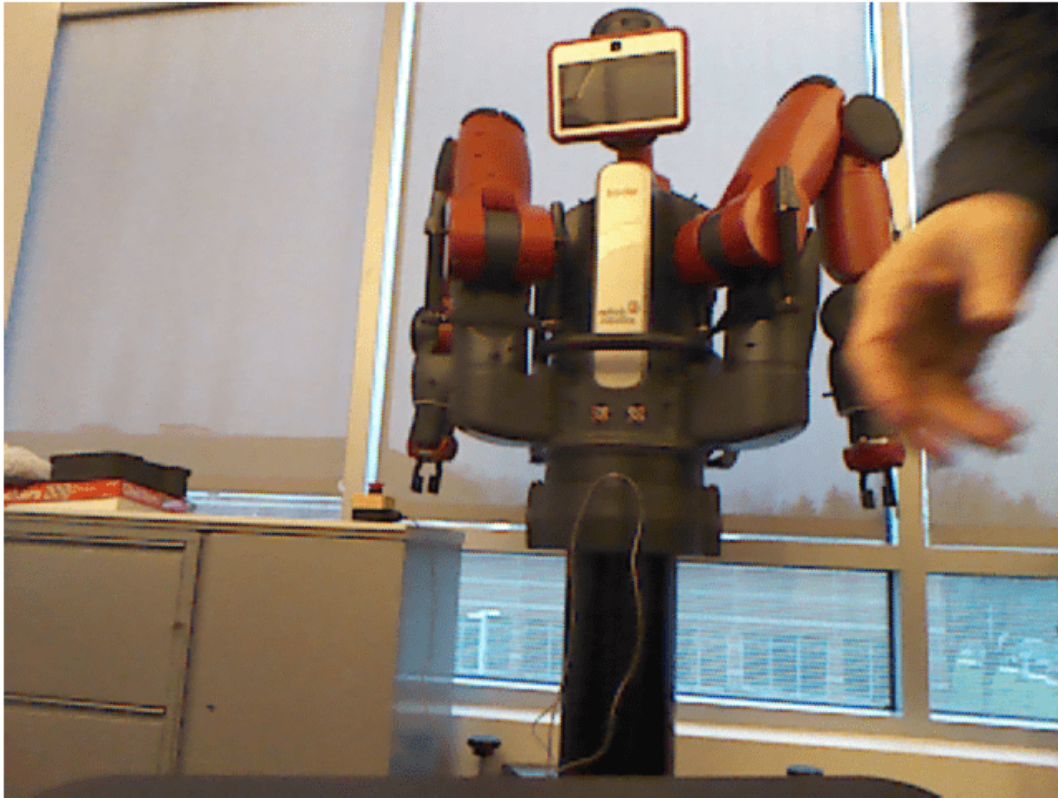
Octomap Messages

ROS uses Octomap messages to implement 3D occupancy grids. Octomap messages are commonly used in robotics applications, such as 3D navigation. You can see the standard ROS format for an octomap message by creating an empty message of the appropriate type.

Use `rosmessage` to create the message.

```
emptyoctomap = rosmessage("octomap_msgs/Octomap",DataFormat="struct")

emptyoctomap = struct with fields:
    MessageType: 'octomap_msgs/Octomap'
         Header: [1×1 struct]
         Binary: 0
             Id: ''
     Resolution: 0
           Data: [0×1 int8]
```

For convenience, an `octomap` message that is fully populated and is stored in the `octomap` variable loaded from `specialROSMessageData.mat`.

Inspect the variable `octomap` in your workspace. The `Data` field contains the octomap structure in a serialized format.

`octomap`

```
octomap = struct with fields:
    MessageType: 'octomap_msgs/Octomap'
         Header: [1×1 struct]
         Binary: 1
             Id: 'OcTree'
     Resolution: 0.0250
           Data: [3926×1 int8]
```

Create an `occupancyMap3D` (Navigation Toolbox) object from the ROS message using the `rosReadOccupancyMap3D` function. Display the 3D occupancy map using the `show` function.

```
occupancyMap3DObj = rosReadOccupancyMap3D(octomap);
show(occupancyMap3DObj)
```

Occupancy Map
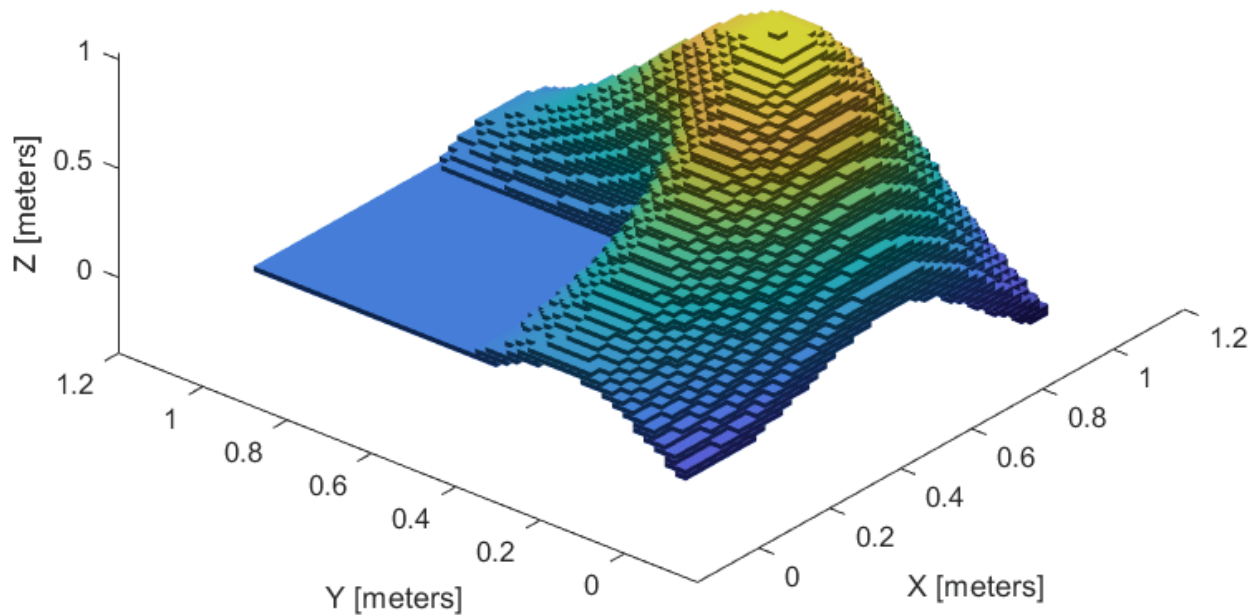
Quaternion Messages

Quaternions are commonly used in robotics to express orientation. Use `rosmessage` to create a quaternion message and observe the fields.

```
emptyquatmsg = rosmessage("geometry_msgs/Quaternion",DataFormat="struct")

emptyquatmsg = struct with fields:
    MessageType: 'geometry_msgs/Quaternion'
              X: 0
              Y: 0
              Z: 0
              W: 0
```

For convenience, a quaternion message that represents a 90 degree rotation about the z-axis was loaded from `specialROSMessageData.mat`. Inspect the variable `quatMsg` in your workspace.

```
quatmsg

quatmsg = struct with fields:
    MessageType: 'geometry_msgs/Quaternion'
              X: 0
              Y: 0
              Z: 0.7071
              W: 0.7071
```

Create a `quaternion` object from a ROS message using the `rosReadQuaternion` function. The `quaternion` object contains the x, y, z, and w components and provides additional functionalities, such as rotating a point.

```
quat = rosReadQuaternion(quatmsg);
```

Define a point in three-dimensional space and rotate it using `rotatepoint` function. Visualize the two points

```
cartesianPoint = [1,0,1];
plot3(cartesianPoint(1),cartesianPoint(2),cartesianPoint(3),"bo")
hold on
plot3([0;cartesianPoint(1)],[0;cartesianPoint(2)],[0;cartesianPoint(3)],"k")
rotationResult = rotatepoint(quat,cartesianPoint);
plot3(rotationResult(1),rotationResult(2),rotationResult(3),"ro")
plot3([0;rotationResult(1)],[0;rotationResult(2)],[0;rotationResult(3)],"k")
xlabel("x")
ylabel("y")
zlabel("z")
grid on
```
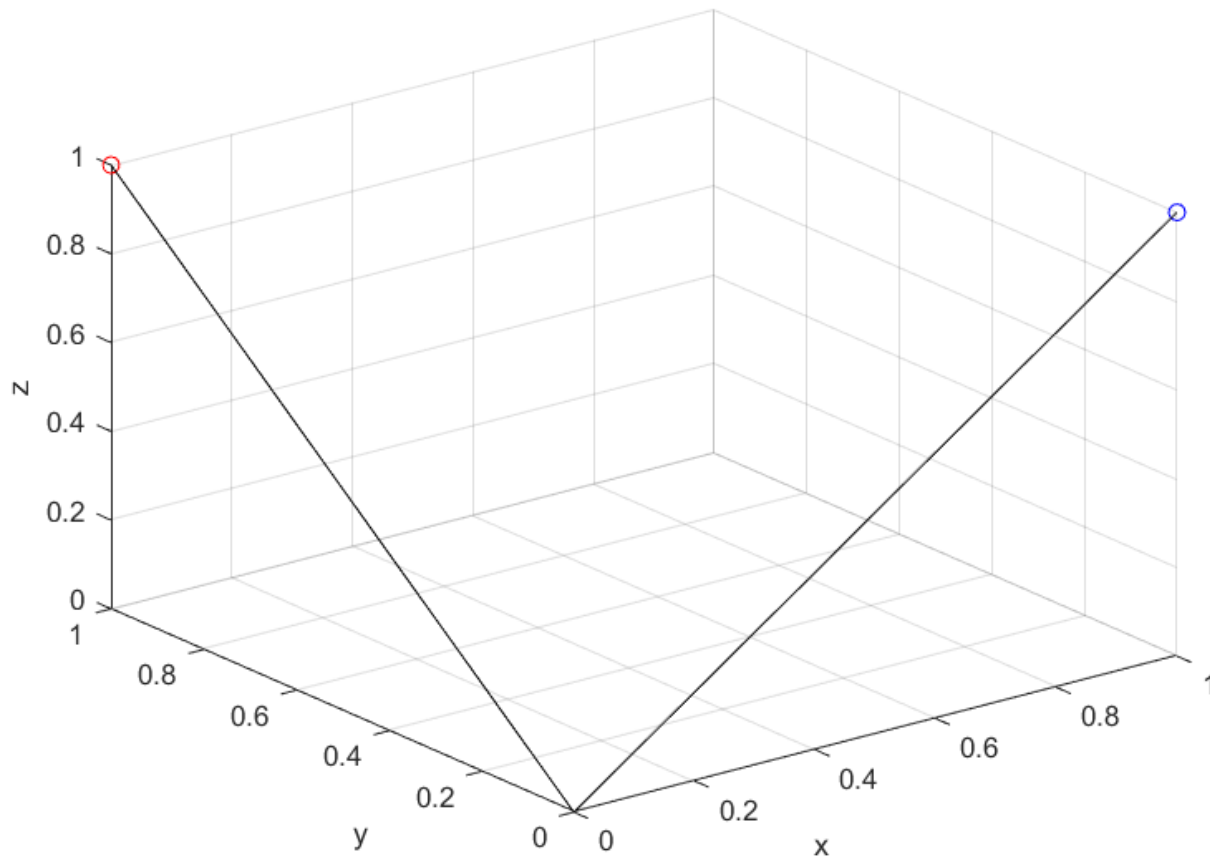
Camera Info Messages

Camera calibration is a commonly used procedure in robotics vision applications. ROS provides
`sensor_msgs/CameraInfo` message type to publish calibration information. Use `rosmessage` to
create a camera info message and observe the fields.

```
emptycamerainfomsg = rosmessage("sensor_msgs/CameraInfo",DataFormat="struct")
```

```
emptycamerainfomsg = struct with fields:
        MessageType: 'sensor_msgs/CameraInfo'
             Header: [1×1 struct]
             Height: 0
              Width: 0
    DistortionModel: ''
                  D: [0×1 double]
                  K: [9×1 double]
                  R: [9×1 double]
                  P: [12×1 double]
           BinningX: 0
           BinningY: 0
                Roi: [1×1 struct]
```

Notably, the message stores the matrices K and P as vectors. ROS requires these matrices to be stored in row-major format. MATLAB stores matrices in column-major, hence extracting the K and P matrices requires reshaping and transposing.

The `estimateCameraParameters` (Computer Vision Toolbox) function can be used to create `cameraParameters` (Computer Vision Toolbox) and `stereoParameters` (Computer Vision Toolbox) objects. You can create `sensor_msgs/CameraInfo` messages from these objects using the `rosWriteCameraInfo` function. The objects must be converted to structures before use. Load the camera calibration structures.

```
load("calibrationStructs.mat")
```

For convenience, the variable `params` loaded from `calibrationStructs.mat` is a fully populated `cameraParameters` struct. Write the `cameraParameters` struct to a new ROS message using the `rosWriteCameraInfo` function.

```
msg = rosWriteCameraInfo(emptycamerainfomsg,params);
```

The following table shows the correspondence between the cameraParameters object and the ROS message.

```
exampleHelperShowCameraParametersTable
```

```
ans=5×2 table
                           ROS message        camera parameters
                          _____      _____

    Intrinsic matrix        "K"                "IntrinsicMatrix"
    Radial distortion       "D(1:2)"           "RadialDistortion"
    Tangential distortion   "D(3:5)"           "TangentialDistortion"
    Height                  "Height"           "ImageSize(1)"
    Width                   "Width"            "ImageSize(2)"
```

Verify that the intrinsic matrix of the ROS message matches the intrinsic matrix of `params`.

```
 K = reshape(msg.K,3,3)'
```

```
K = 3×3

  714.1885         0  563.6481
        0  710.3785  355.7254
        0         0    1.0000
```

```
 intrinsicMatrix = params.IntrinsicMatrix'
```

```
intrinsicMatrix = 3×3

  714.1885         0  563.6481
        0  710.3785  355.7254
        0         0    1.0000
```

For convenience, the variable `stereoParams` loaded from `calibrationStructs.mat` is a fully populated `stereoParameters` struct. Write the `stereoParameters` struct to two new ROS messages using the `rosWriteCameraInfo` function.

```
[msg1,msg2] = rosWriteCameraInfo(msg,stereoParams);
```

The following table shows the correspondence between the stereoParameters object and the ROS message.

```
exampleHelperShowStereoParametersTable
```

ans=*2×2 table*

	ROS message	stereoParameters
Translation of camera 2	"P(:,1:2)"	"TranslationOfCamera2(1:2)"
Rotation of camera 2	"inv(R1)*R2"	"RotationOfCamera2"

Verify that the camera 2 rotation matrices of the ROS message and `stereoParams` match.

```
R1 = reshape(msg1.R,3,3)';
R2 = reshape(msg2.R,3,3)';
R = R1\R2
```

R = *3×3*

```
    1.0000   -0.0002   -0.0050
    0.0002    1.0000   -0.0037
    0.0050    0.0037    1.0000
```

```
rotationOfCamera2 = stereoParams.RotationOfCamera2
```

rotationOfCamera2 = *3×3*

```
    1.0000   -0.0002   -0.0050
    0.0002    1.0000   -0.0037
    0.0050    0.0037    1.0000
```

Verify that the camera 2 translation vectors of the ROS message and `stereoParams` match.

```
P = reshape(msg2.P,4,3)';
P(1:2,end)'
```

ans = *1×2*

```
 -119.8720   -0.4005
```

```
translationOfCamera2 = stereoParams.TranslationOfCamera2(1:2)
```

translationOfCamera2 = *1×2*

```
 -119.8720   -0.4005
```

Laser Scan Messages

Laser scanners are commonly used sensors in robotics. ROS provides `sensor_msgs/LaserScan` message type to publish laser scan messages. Use `rosmessage` to create a laser scan message and observe the fields.

```
emptyscan = rosmessage("sensor_msgs/LaserScan","DataFormat","struct")

emptyscan = struct with fields:
        MessageType: 'sensor_msgs/LaserScan'
             Header: [1×1 struct]
           AngleMin: 0
           AngleMax: 0
     AngleIncrement: 0
      TimeIncrement: 0
           ScanTime: 0
           RangeMin: 0
           RangeMax: 0
             Ranges: [0×1 single]
        Intensities: [0×1 single]
```

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

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 = struct with fields:
        MessageType: 'sensor_msgs/LaserScan'
             Header: [1×1 struct]
           AngleMin: -0.5467
           AngleMax: 0.5467
     AngleIncrement: 0.0017
      TimeIncrement: 0
           ScanTime: 0.0330
           RangeMin: 0.4500
           RangeMax: 10
             Ranges: [640×1 single]
        Intensities: [0×1 single]
```

You can get the scan angles from the ROS message using the `rosReadScanAngles` function. Visualize the scan data in polar coordinates using the `polarPlot` function.

```
angles = rosReadScanAngles(scan);
figure
polarplot(angles,scan.Ranges,LineWidth=2)
title("Laser Scan")
```

Laser Scan
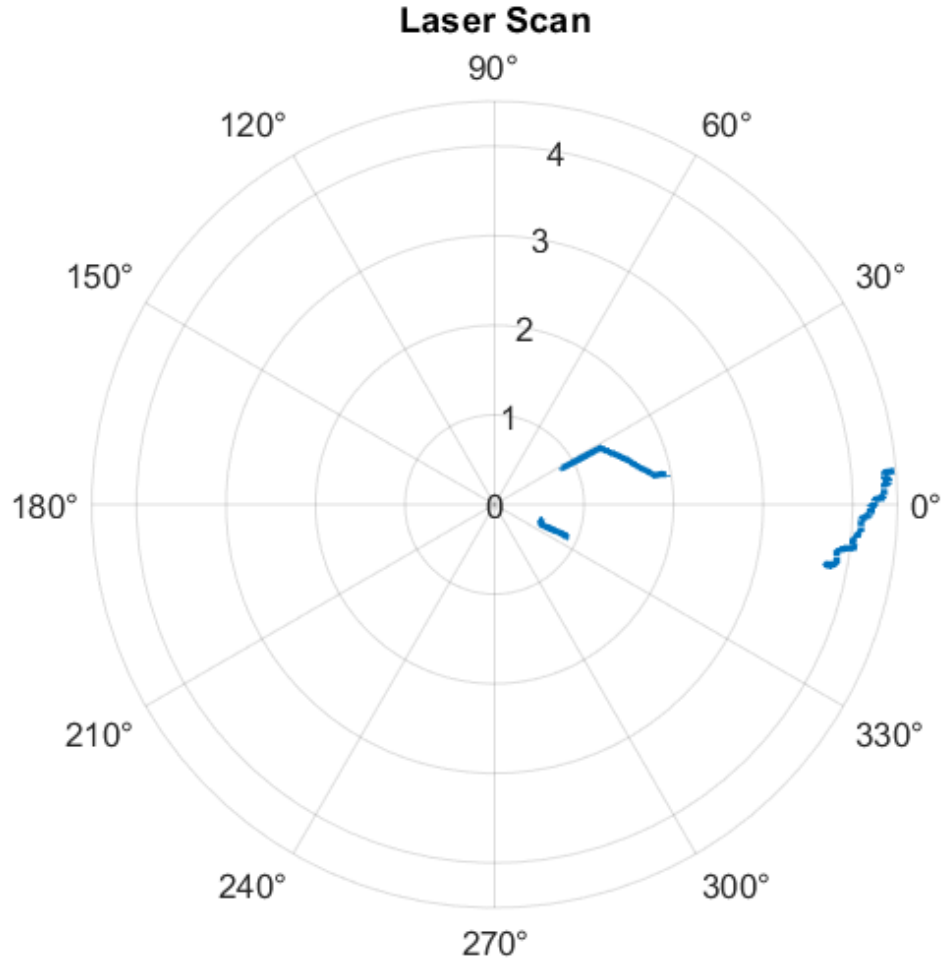
You can get the measured points in Cartesian coordinates using the `rosReadCartesian` function.

```
xy = rosReadCartesian(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 `rosPlot` function:

```
rosPlot(scan,"MaximumRange",5)
```

Laser Scan
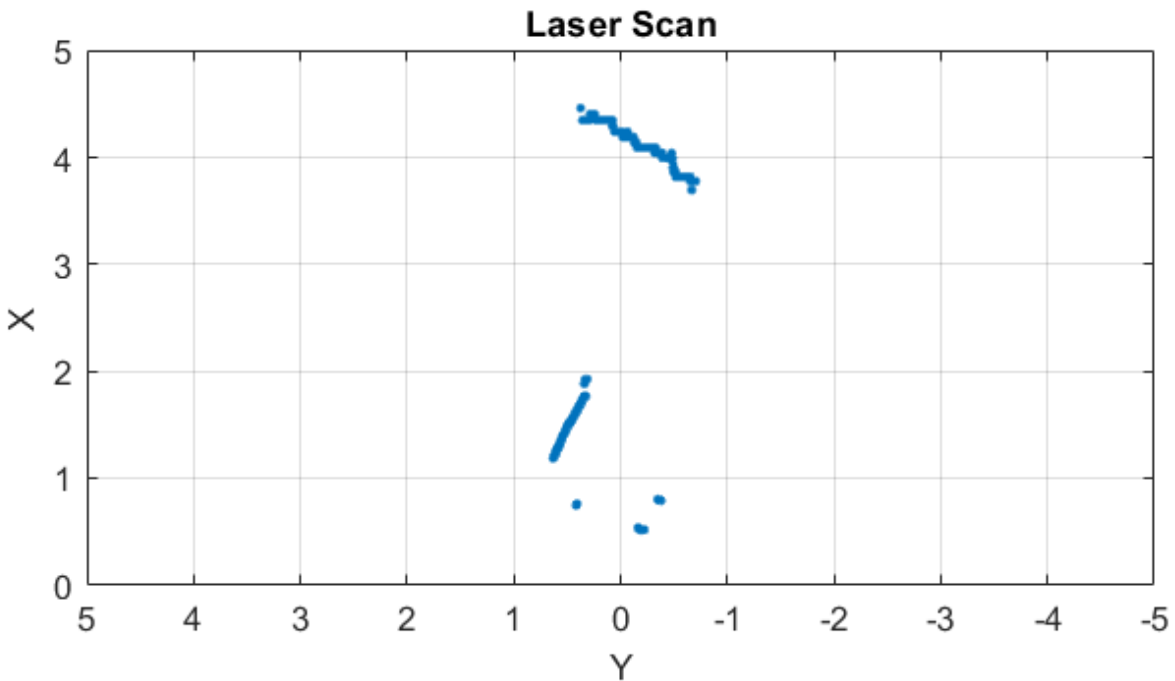
Create a `lidarScan` object from a ROS message using the `rosReadLidarScan` function. The `lidarScan` object contains ranges, angles, and Cartesian points and provides additional functionalities, such as transforming the scanned points. Use the `transformScan` function to rotate the scan point and visualize it using `plot`.

```
lidarScanObj = rosReadLidarScan(scan)

lidarScanObj =
  lidarScan with properties:

       Ranges: [640×1 double]
       Angles: [640×1 double]
    Cartesian: [640×2 double]
        Count: 640


rotateScan = transformScan(lidarScanObj,[0,0,pi/2]);
plot(rotateScan)
```

LiDAR Scan
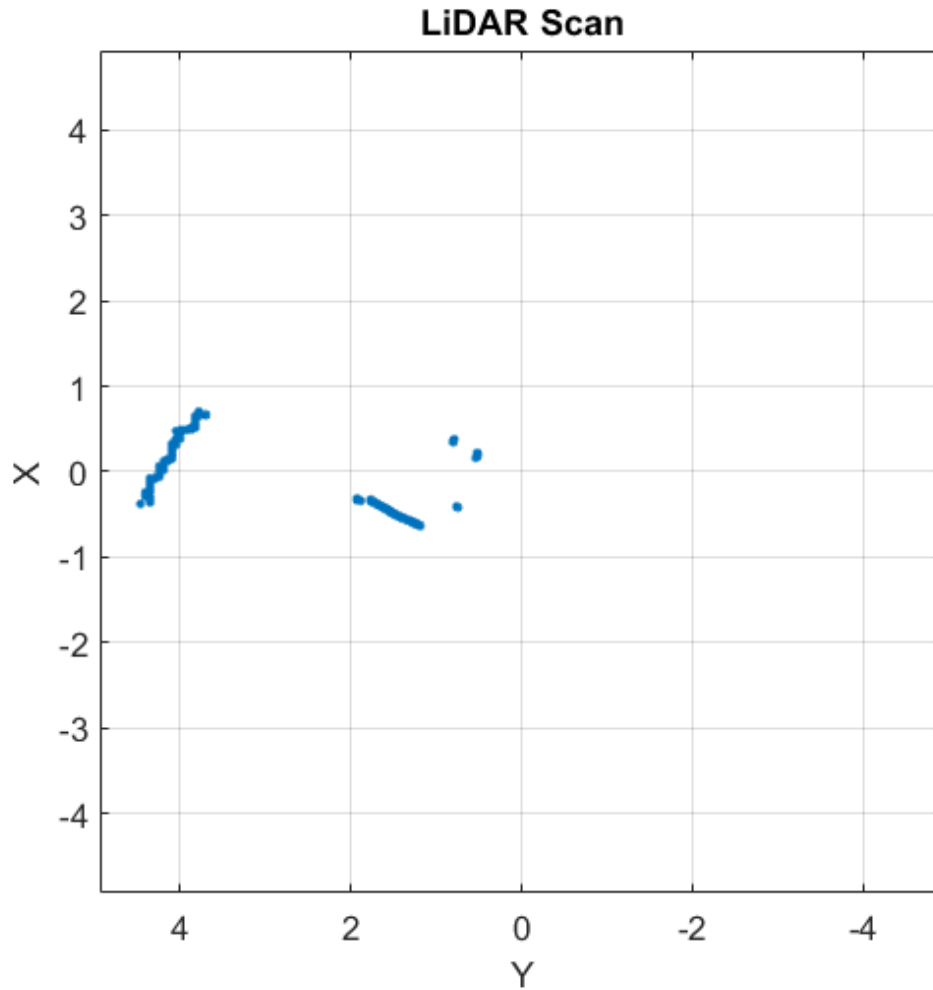
Occupancy Grid Messages

Occupancy grid messages are commonly used in robotics for 2D navigation applications. ROS provides `nav_msgs/OccupancyGrid` message type to publish laser scan messages. Use `rosmessage` to create an occupancy grid message and observe the fields.

```
emptyMap = rosmessage("nav_msgs/OccupancyGrid",DataFormat="struct")
```

```
emptyMap = struct with fields:
    MessageType: 'nav_msgs/OccupancyGrid'
         Header: [1×1 struct]
           Info: [1×1 struct]
           Data: [0×1 int8]
```

Notice that `emptyMap` does not contain any meaningful data. For convenience, an occupancy grid message that is fully populated and is stored in the `mapMsg` variable loaded from `specialROSMessageData.mat`.

Inspect the `mapMsg` variables. The occupancy grid values are encoded in the `Data` field.

```
mapMsg

mapMsg = struct with fields:
    MessageType: 'nav_msgs/OccupancyGrid'
         Header: [1×1 struct]
           Info: [1×1 struct]
           Data: [251001×1 int8]
```

Use the `rosReadOccupancyGrid` function to convert the ROS message to an `occupancyMap` (Navigation Toolbox) object. Use the `show` function to display the occupancy grid.

```
occupancyMapObj = rosReadOccupancyGrid(mapMsg);
show(occupancyMapObj);
```
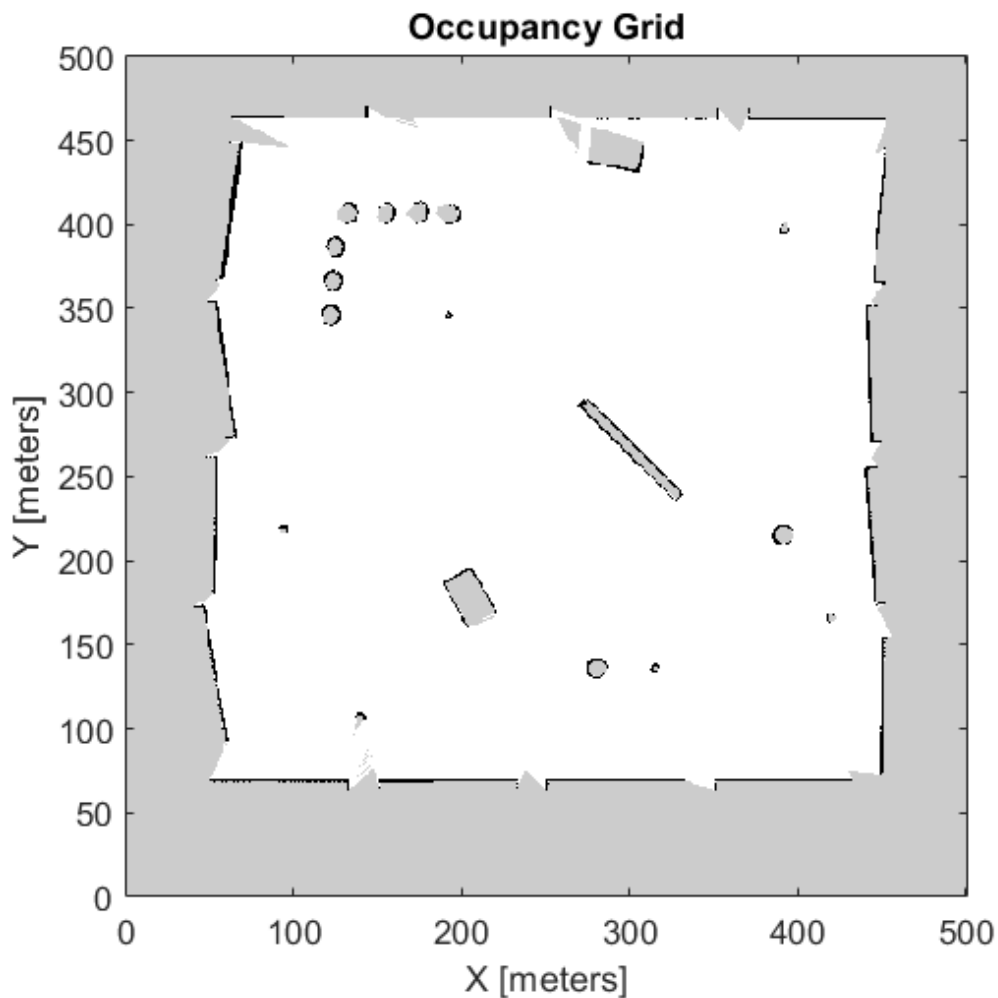

You can use the `occupancyMap` (Navigation Toolbox) object functions to manipulate the occupancy grid. Use the `inflate` function to expand the occupied regions. Write the occupancyMap object to a new ROS message using the `rosWriteOccupancyGrid` function. Use the `show` function to display the new occupancy grid.

```
inflate(occupancyMapObj,5)
occupancyMapInflatedMsg = rosWriteOccupancyGrid(mapMsg,occupancyMapObj);
show(occupancyMapObj);
```
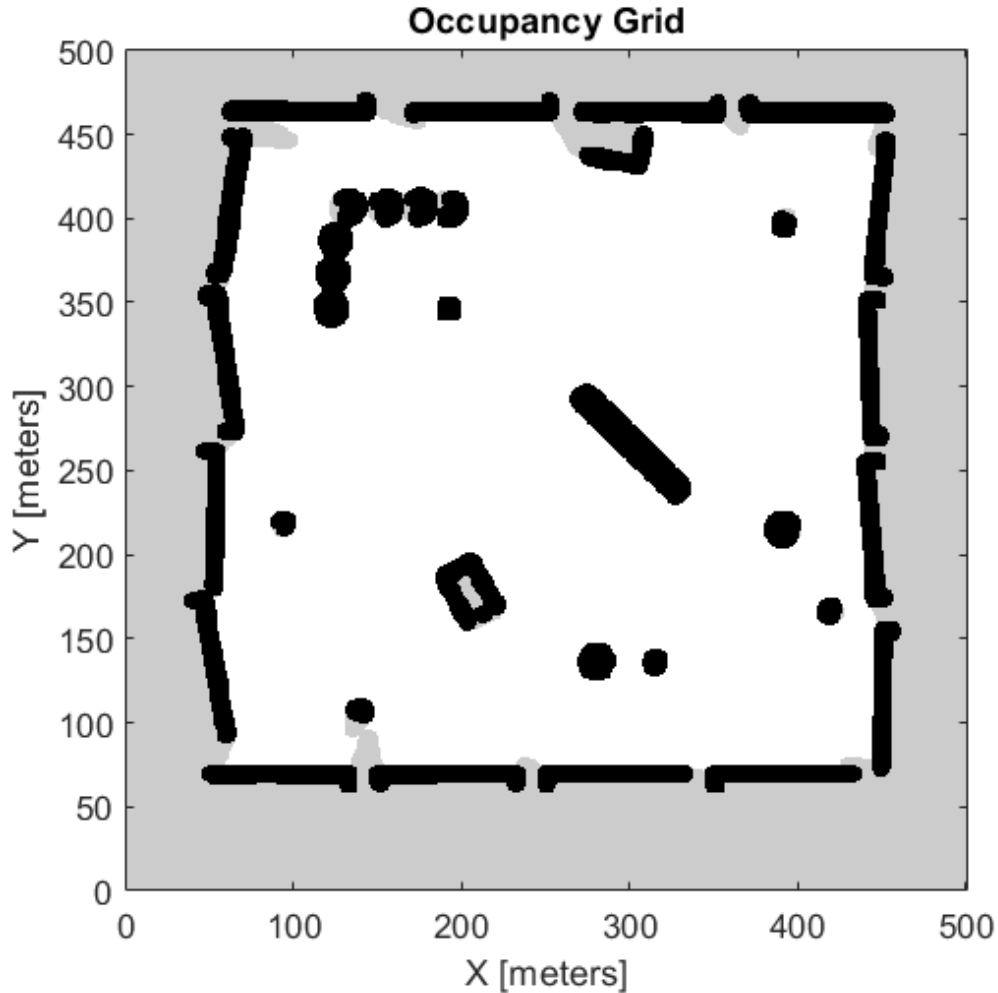

Occupancy Grid

Alternatively, you can create a `binaryOccupancyMap` (Navigation Toolbox) object from a ROS occupancy grid message using the `rosReadBinaryOccupancyGrid` function. A binary occupancy map holds discrete occupancy values of `0` or `1` at each cell whereas an occupancy map holds probability occupancy values that range between `0` and `1` at each cell. Use the `show` function to display the binary occupancy grid.

```
binaryMapObj = rosReadBinaryOccupancyGrid(mapMsg);
show(binaryMapObj);
```

Binary Occupancy Grid
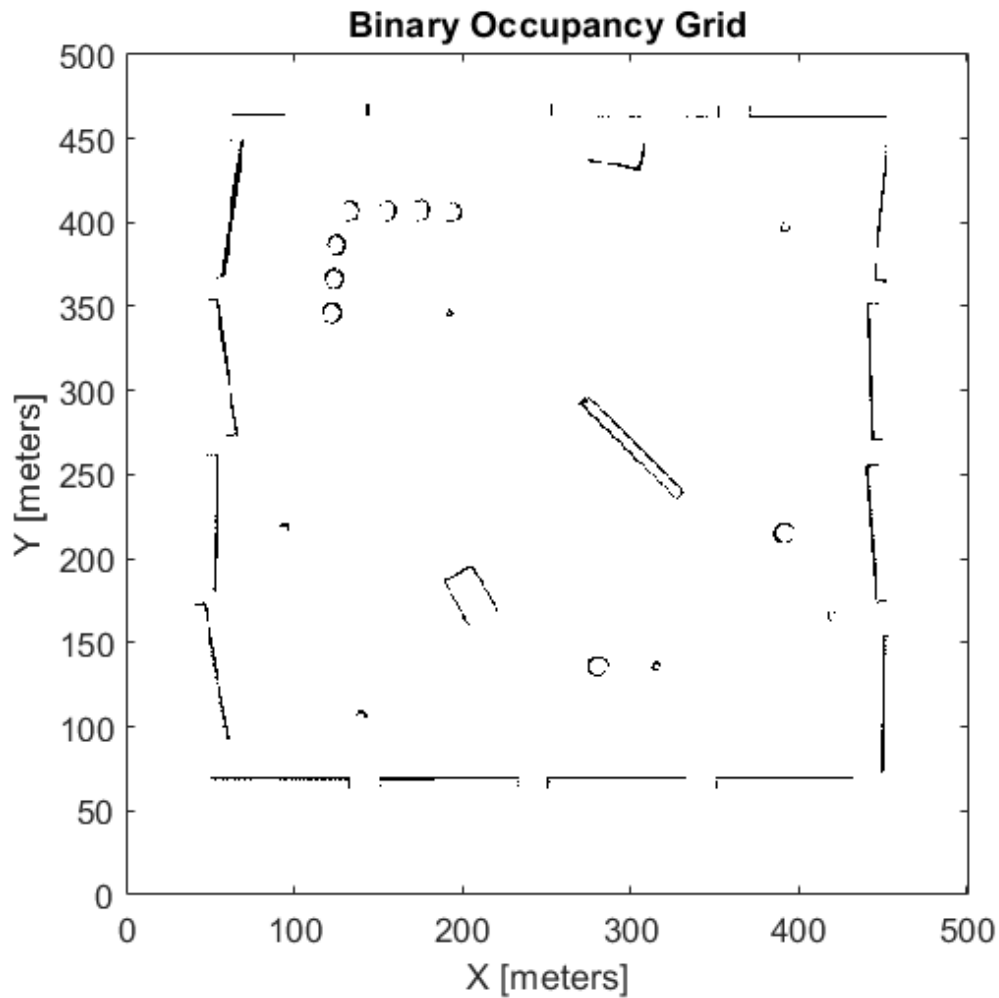
Similarly, you can use the `binaryOccupancyMap` (Navigation Toolbox) object functions to manipulate the binary occupancy grid. Use the `inflate` function to alter the binary occupancy grid and create a new ROS message using the `rosWriteBinaryOccupancyGrid` function. Use the `show` function to display the new binary occupancy grid.

```
inflate(binaryMapObj,5)
binaryMapInflatedMsg = rosWriteBinaryOccupancyGrid(mapMsg,binaryMapObj);
show(binaryMapObj);
```

Binary Occupancy Grid
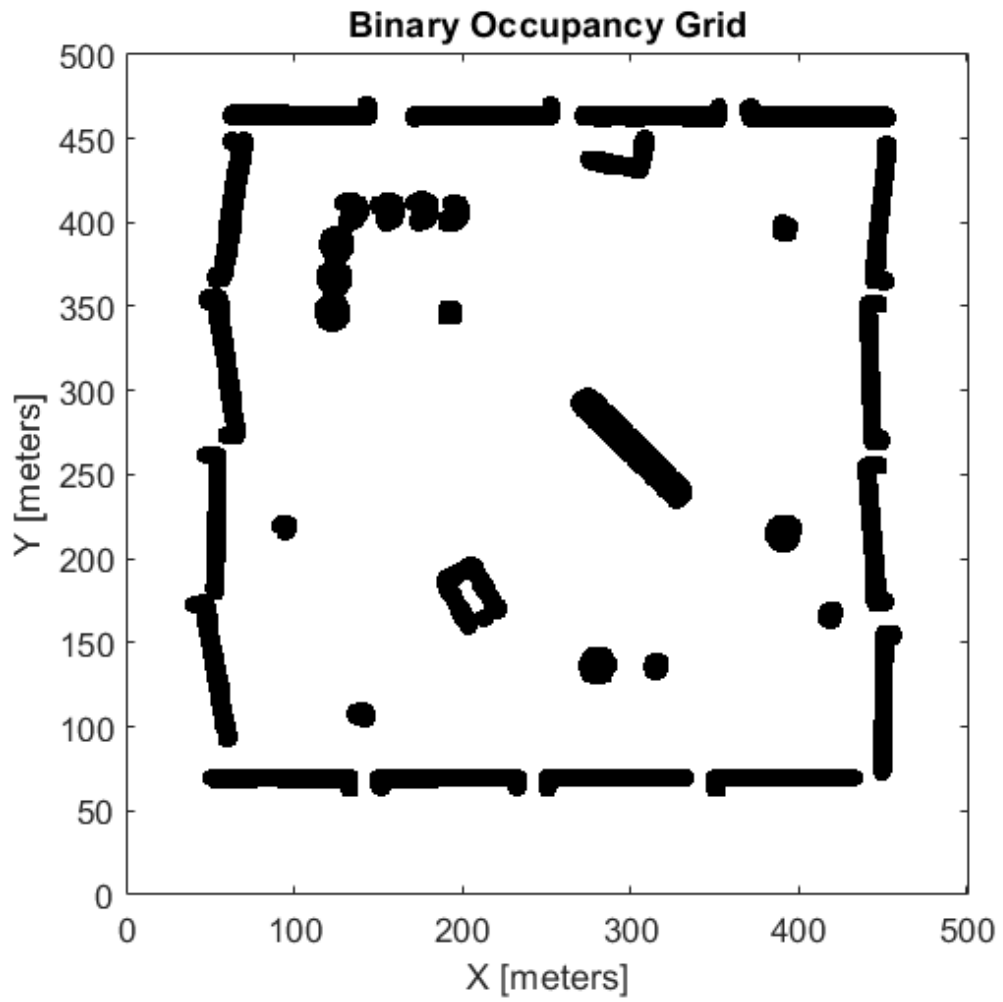
Work with Velodyne ROS Messages

This example shows how to handle `VelodyneScan` messages from a Velodyne LiDAR.

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.

Prerequisites: "Work with Basic ROS Messages" on page 1-52

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. Use messages in structure format for better performance.

```
emptyveloScan = rosmessage("velodyne_msgs/VelodyneScan","DataFormat","struct")

emptyveloScan = struct with fields:
    MessageType: 'velodyne_msgs/VelodyneScan'
         Header: [1×1 struct]
        Packets: [0×1 struct]
```

Since you created an empty message, `emptyveloScan` does not contain any meaningful data. For convenience, the loaded `lidarData_ConstructionRoad.mat` file contains a set of `VelodyneScan` messages that are fully populated and 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","DataFormat","struct")

emptyveloPkt = struct with fields:
    MessageType: 'velodyne_msgs/VelodynePacket'
          Stamp: [1×1 struct]
           Data: [1206×1 uint8]
```

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: {28×1 cell}
     DeviceModel: 'HDL32E'
 CalibrationFile: 'M:\jobarchive\Bdoc21b\2021_06_16_h16m50s15_job1697727_pass\matlab\toolbox\
  NumberOfFrames: 55
        Duration: 2.7477 sec
       StartTime: 1145.2 sec
         EndTime: 1147.9 sec
      Timestamps: [1145.2 sec    1145.2 sec    1145.3 sec    1145.3 sec    1145.4 sec    1145
     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 + seconds(1);
```

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
```
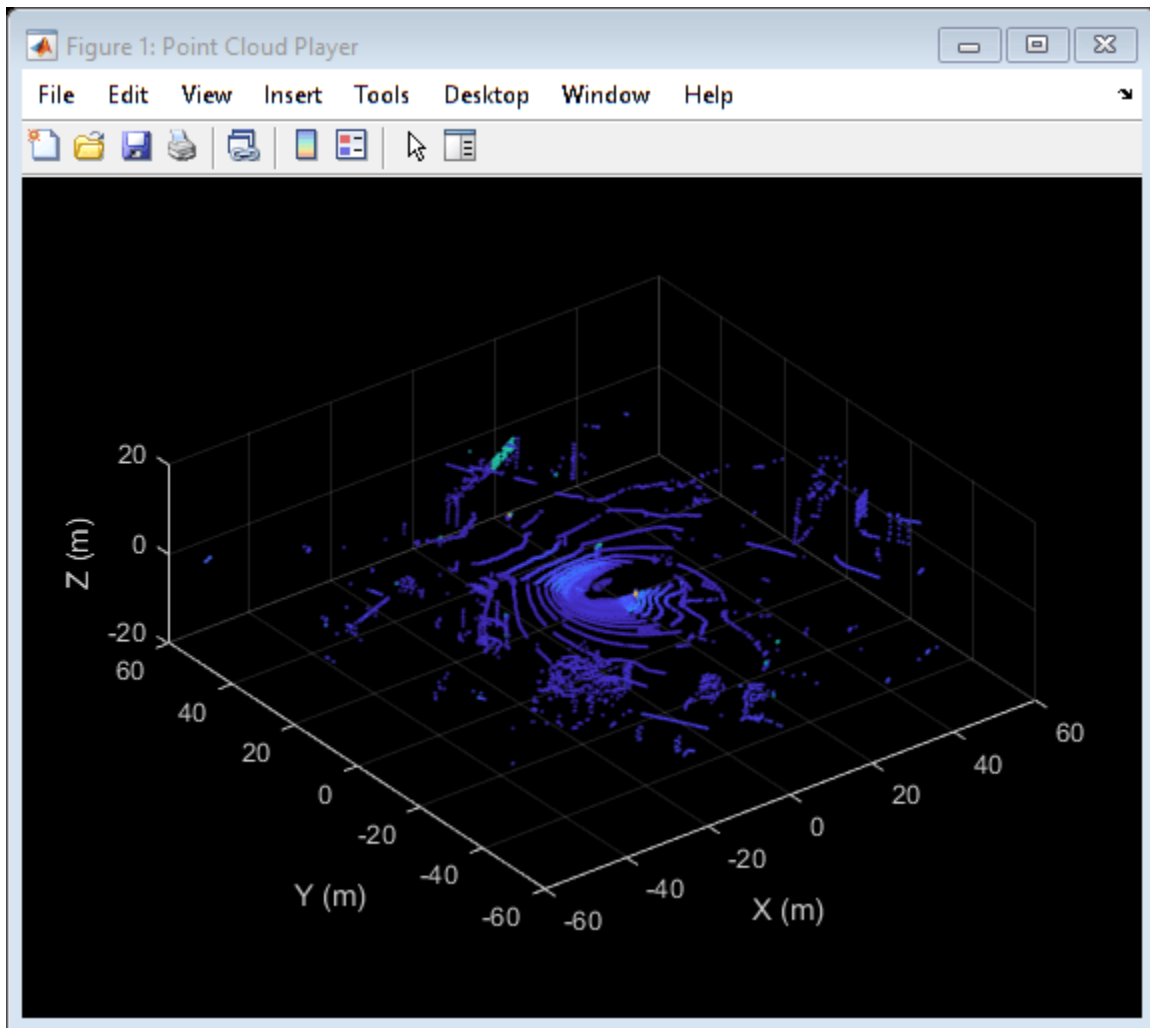
Get Started with a Real TurtleBot

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 Simulated TurtleBot" on page 1-206 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.
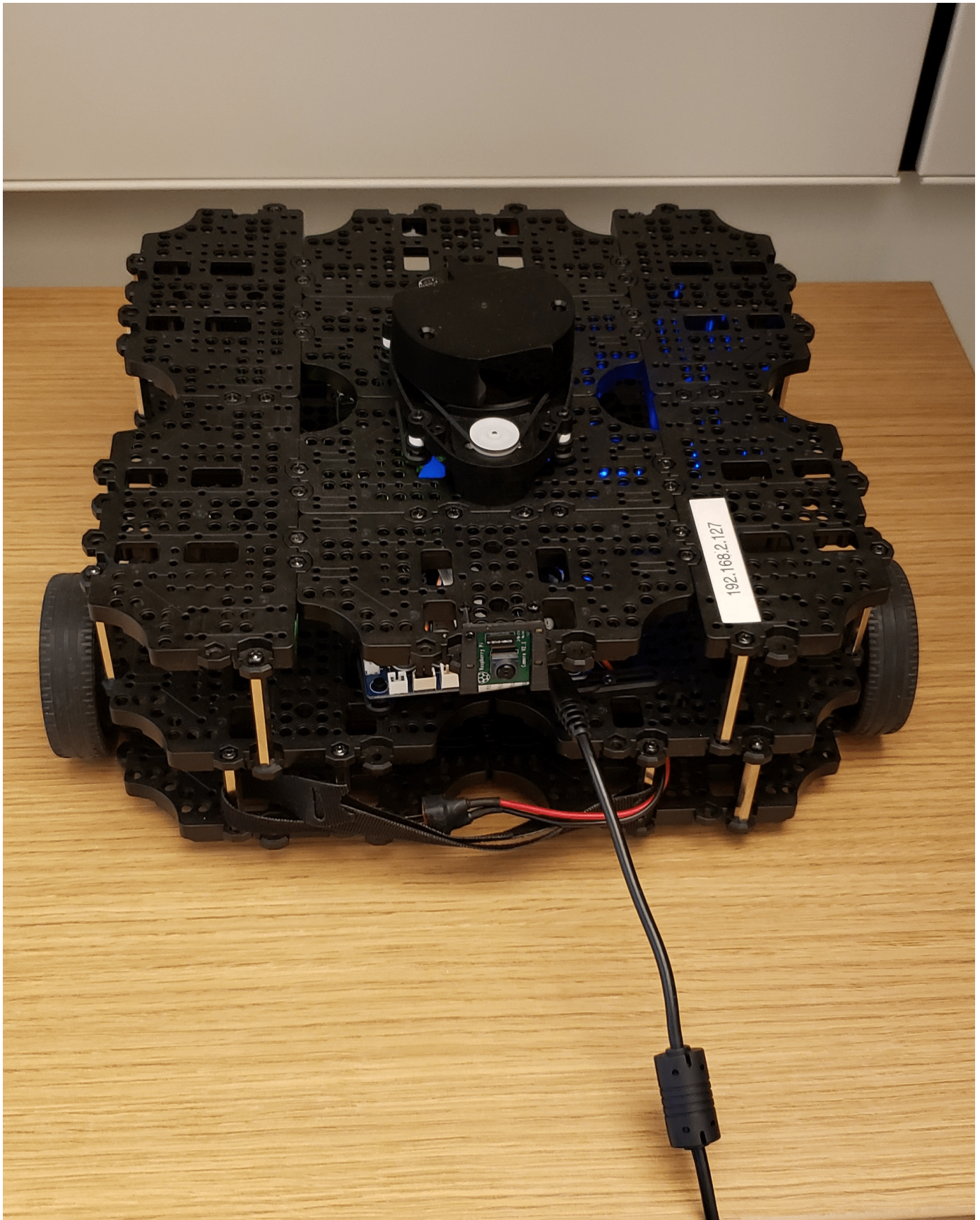
- Unpack the TurtleBot and make sure the power source is connected.
- Turn on the Raspberry Pi.
- Make sure that you have a network set up to connect the host computer (the one with MATLAB) to the Raspberry Pi on this TurtleBot. Use a wireless router or an Ethernet cable.
- Open a terminal on the Raspberry Pi and run `ifconfig`. The IP address associated with the network that you connected to is displayed.

```
turtlebot@turtlebot-1015E: ~

turtlebot@turtlebot-1015E:~$ ifconfig
lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:16436  Metric:1
          RX packets:20125 errors:0 dropped:0 overruns:0 frame:0
          TX packets:20125 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:4773546 (4.7 MB)  TX bytes:4773546 (4.7 MB)

wlan0     Link encap:Ethernet  HWaddr 6c:71:d9:7d:2c:05
          inet addr:192.168.1.117  Bcast:192.168.1.255  Mask:255.255.255.0
          inet6 addr: fe80::6e71:d9ff:fe7d:2c05/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:4  errors:0 dropped:0 overruns:0 frame:0
          TX packets:97 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:3145 (3.1 KB)  TX bytes:15589 (15.5 KB)

turtlebot@turtlebot-1015E:~$
```

- 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.
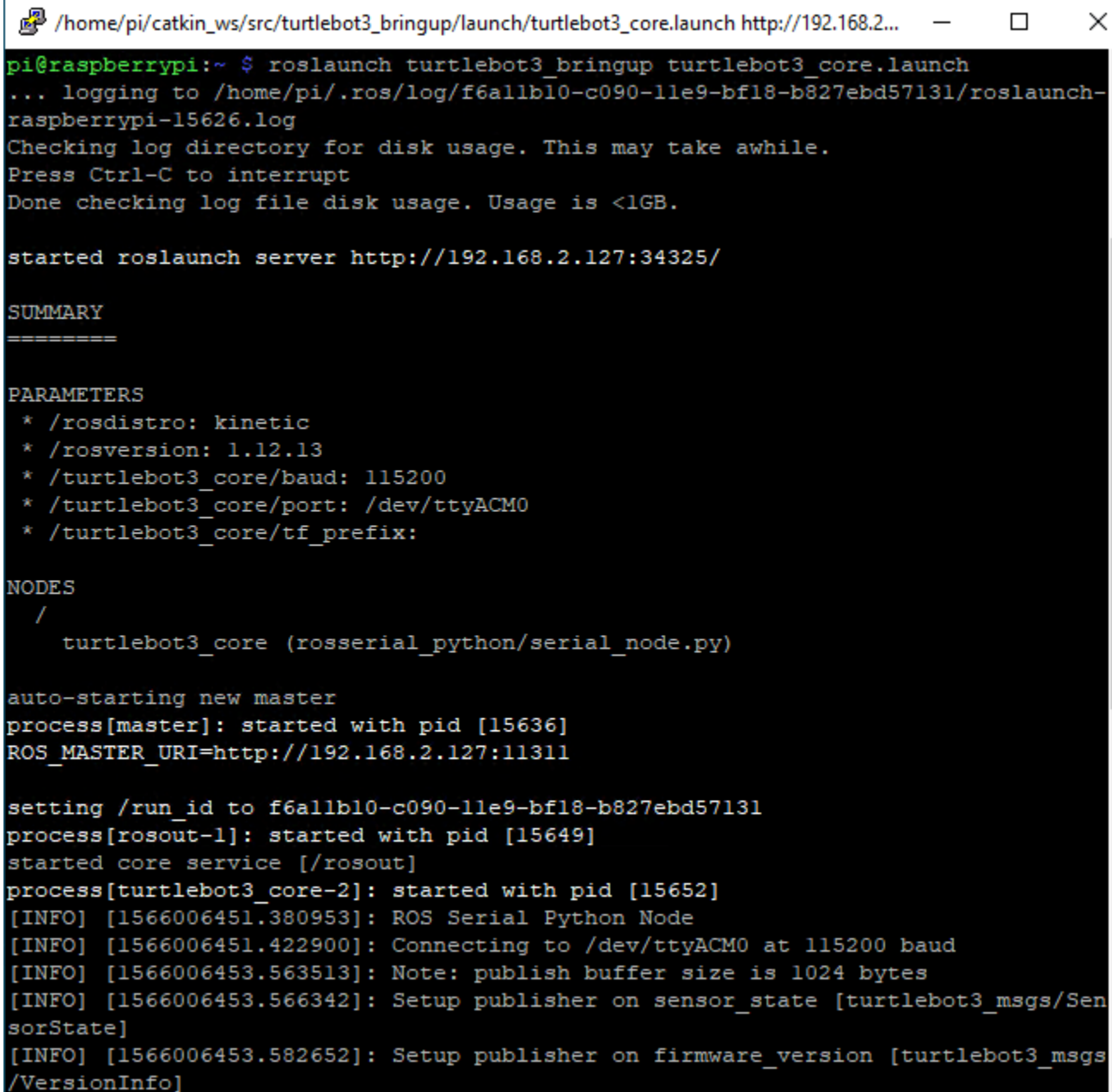
Here is a diagram illustrating the proper assignment of environment variables:
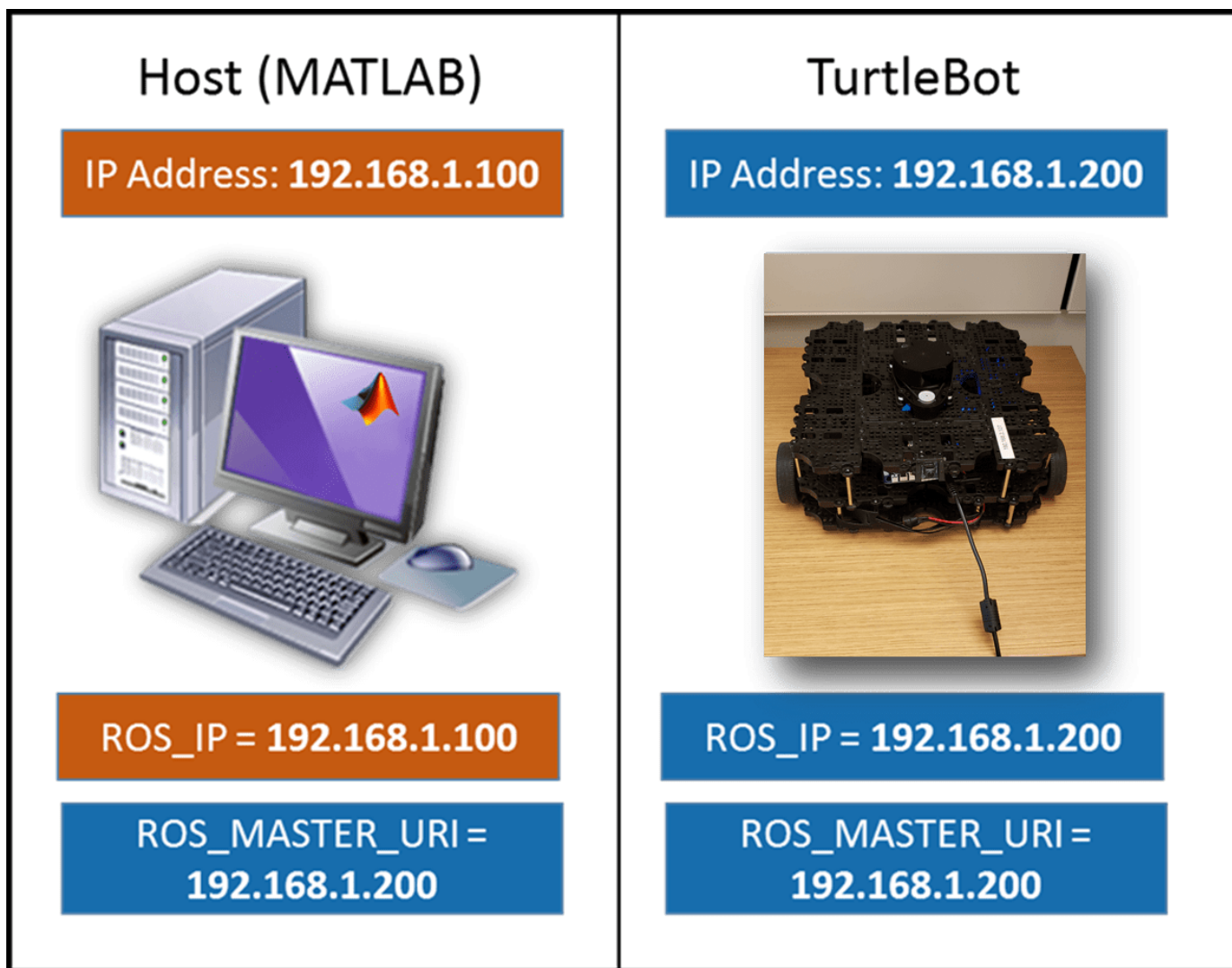
Type the following commands in separate terminals on the TurtleBot Raspberry Pi to launch LiDAR and camera sensors in TurtleBot 3:

```
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 1-236

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 1-39

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.
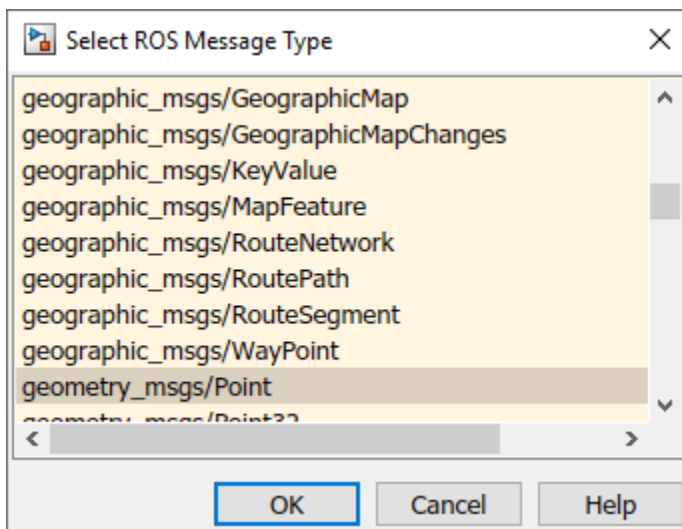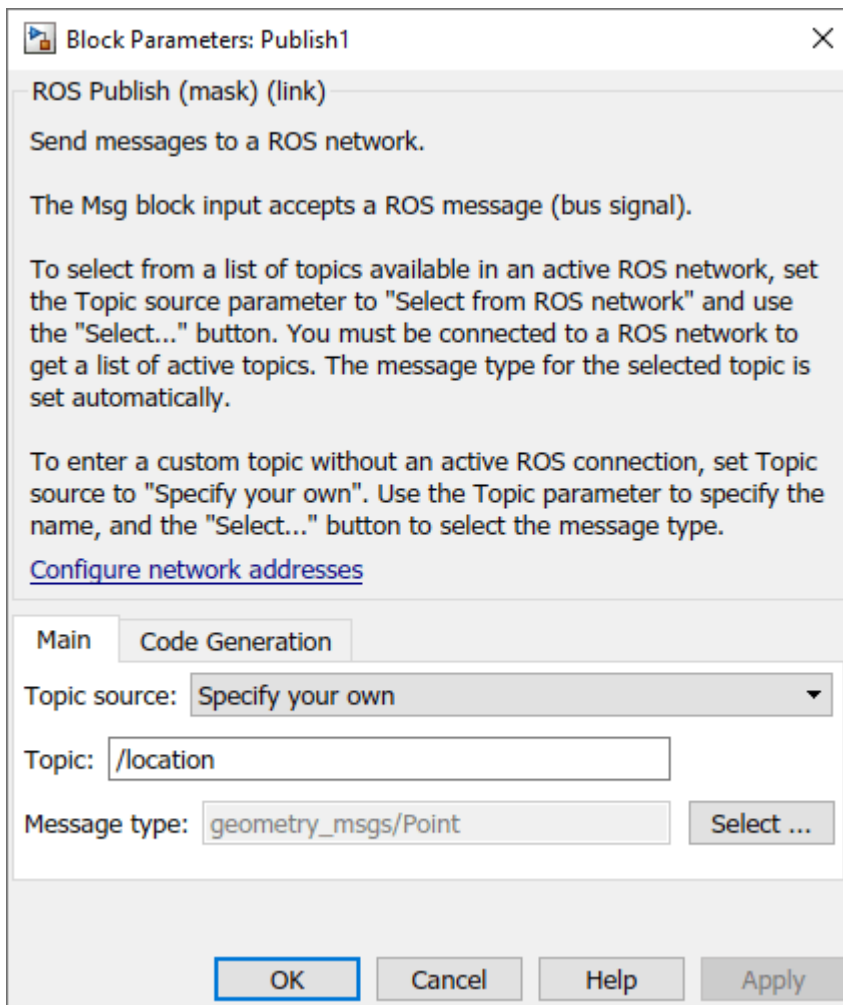
Create a ROS Message

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 "Explore Simulink Bus Capabilities" (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
```

- 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.
- **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).
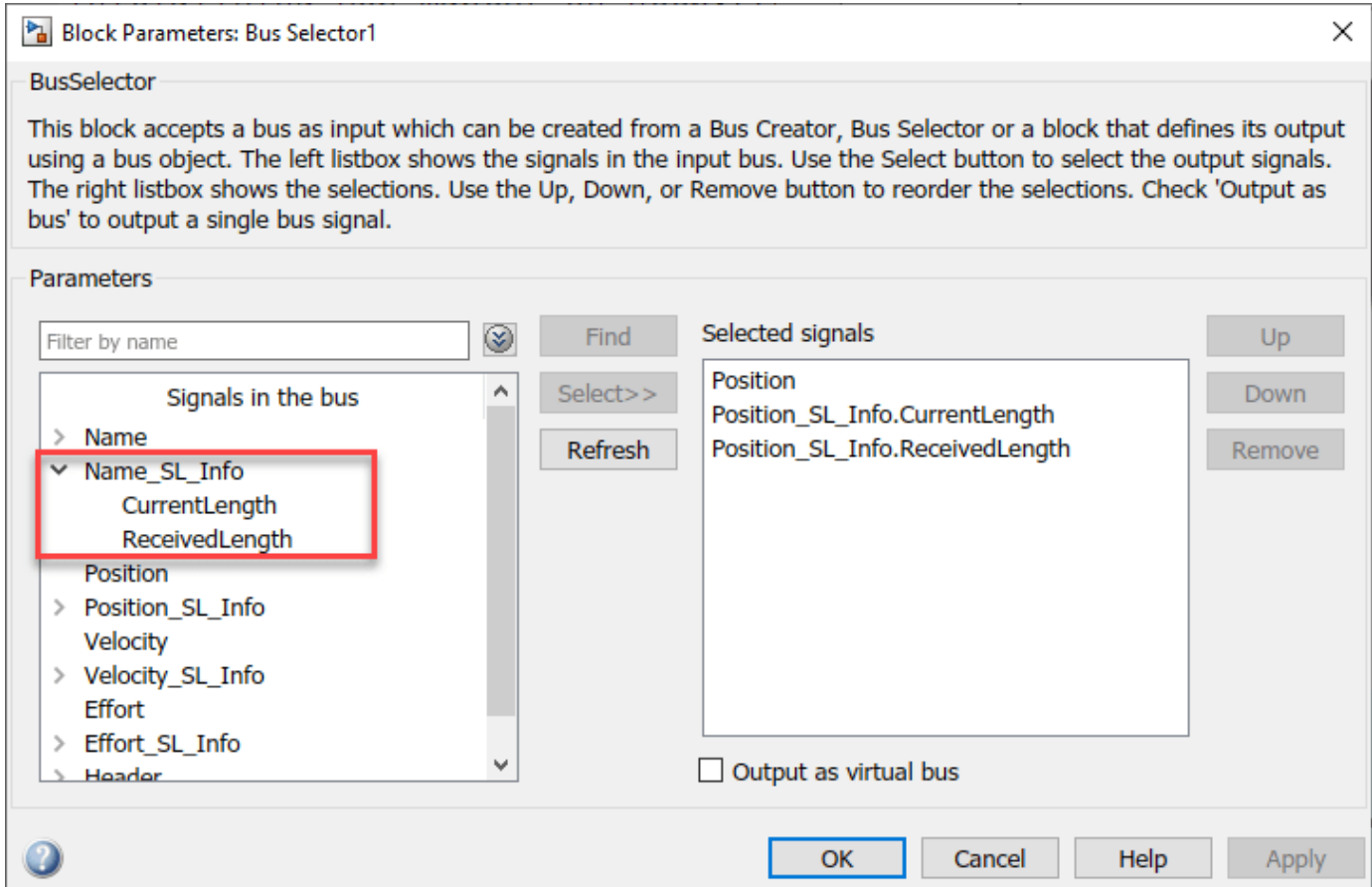
Configure ROS Network

- Under the **Simulation** tab, select **ROS Network** from the **Prepare** section**.** If you do not see ROS Toolbox, select **Robot Operating System (ROS)** on the Apps tab, under **Control Systems**. In the dialog box that opens up, select **Robot Operating System (ROS)** from the **ROS Network** drop-down.
- Set the **Network Address** for both **ROS Master (ROS 1)** and **Node Host (ROS 1)** to `Default`.
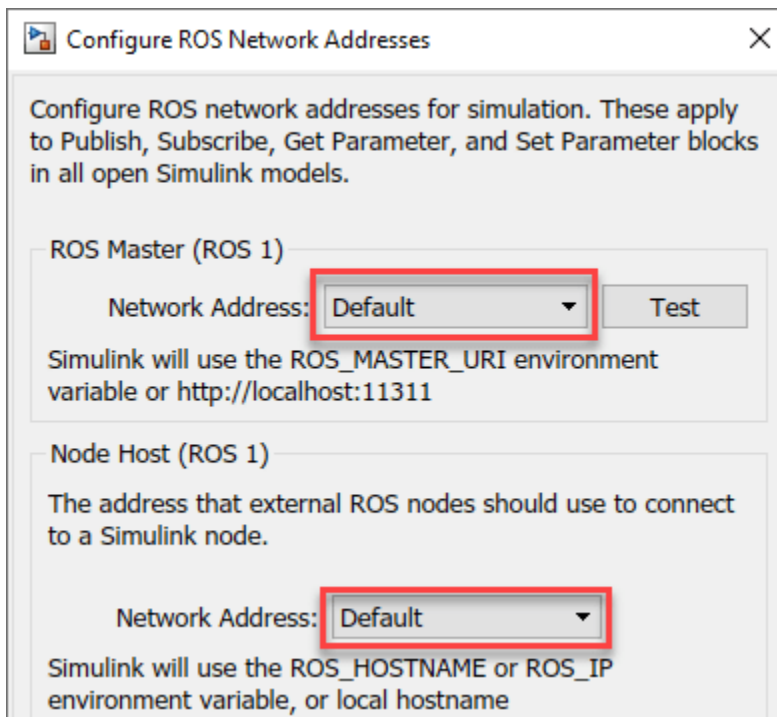
- Enter `rosinit` at the MATLAB® command line.

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 using the name `SL_Bus_<modelname>_<messageType>` and stored in the Simulink data dictionary `robotlib`. You can find this data dictionary under **External Data > From Libraries** in **Model Explorer**. (The bus names may be abbreviated if the model name is too long.)

`ros.createSimulinkBus(gcs)`

- Double-click the **MATLAB Function - Assign** block. In the MATLAB Editor toolstrip, under **Modeling** tab, click **Symbols Pane**. Then right click on `msg`, select **Inspect** 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**.
- If you are using variable-size messages within MATLAB Function block, you must enable **Dynamic memory allocation in MATLAB functions** setting under **Model Settings > Simulation Target > Advanced parameters**.
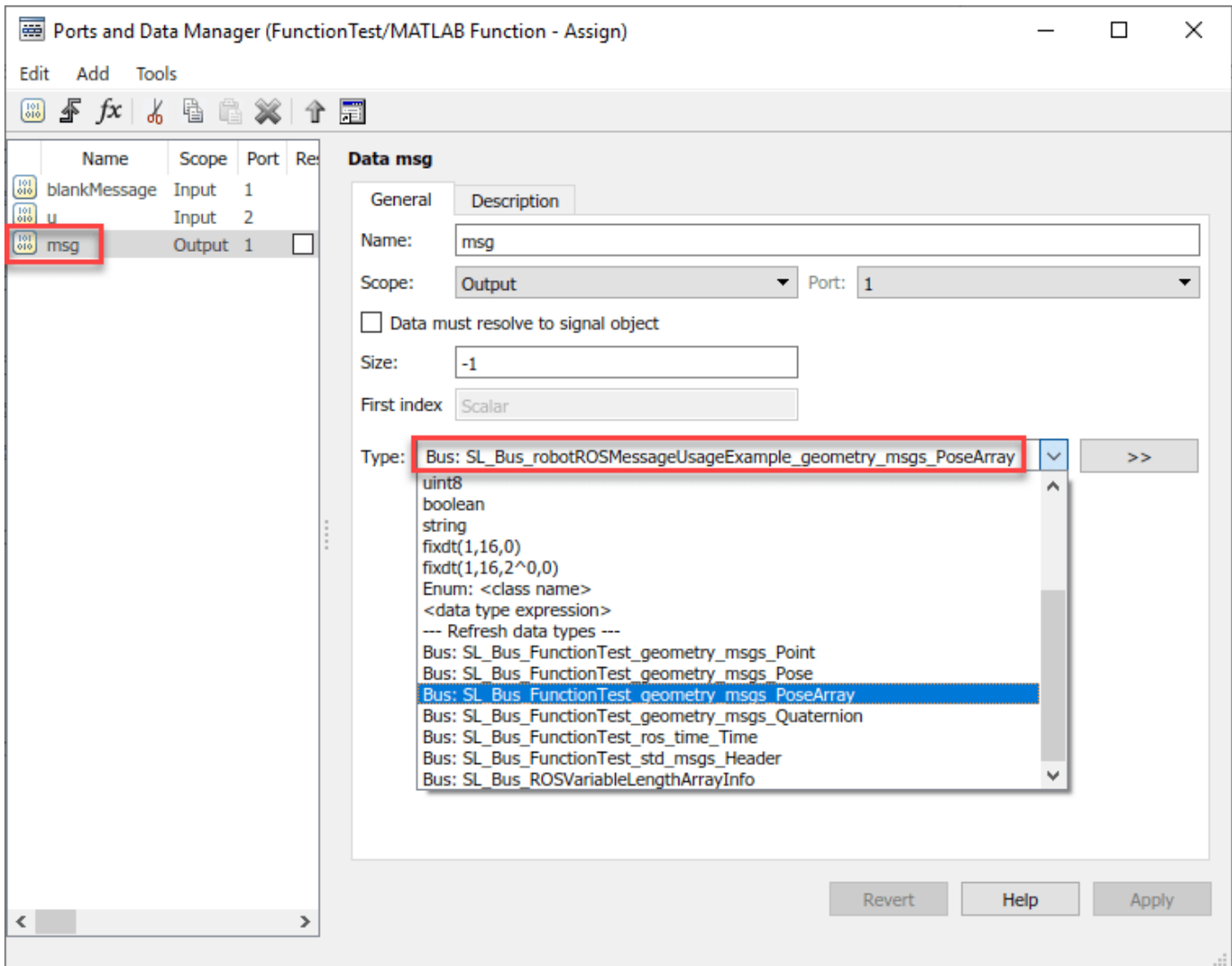
Configure the ROS Network

- 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. 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.

Message types in model:

☑ Use default limits for this message type

Array Property	Type of array item	Maximum length (items)
Effort	double	128
Name	std_msgs/String	16
Position	double	128
Velocity	double	128

geometry_msgs/PoseArray
nav_msgs/Odometry
sensor_msgs/JointState
sensor_msgs/LaserScan
std_msgs/Header
std_msgs/String

- 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. It then shows how to send velocity commands and 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 1-39, "Exchange Data with ROS Publishers and Subscribers" on page 1-59, "Get Started with ROS in Simulink" on page 1-125.

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:
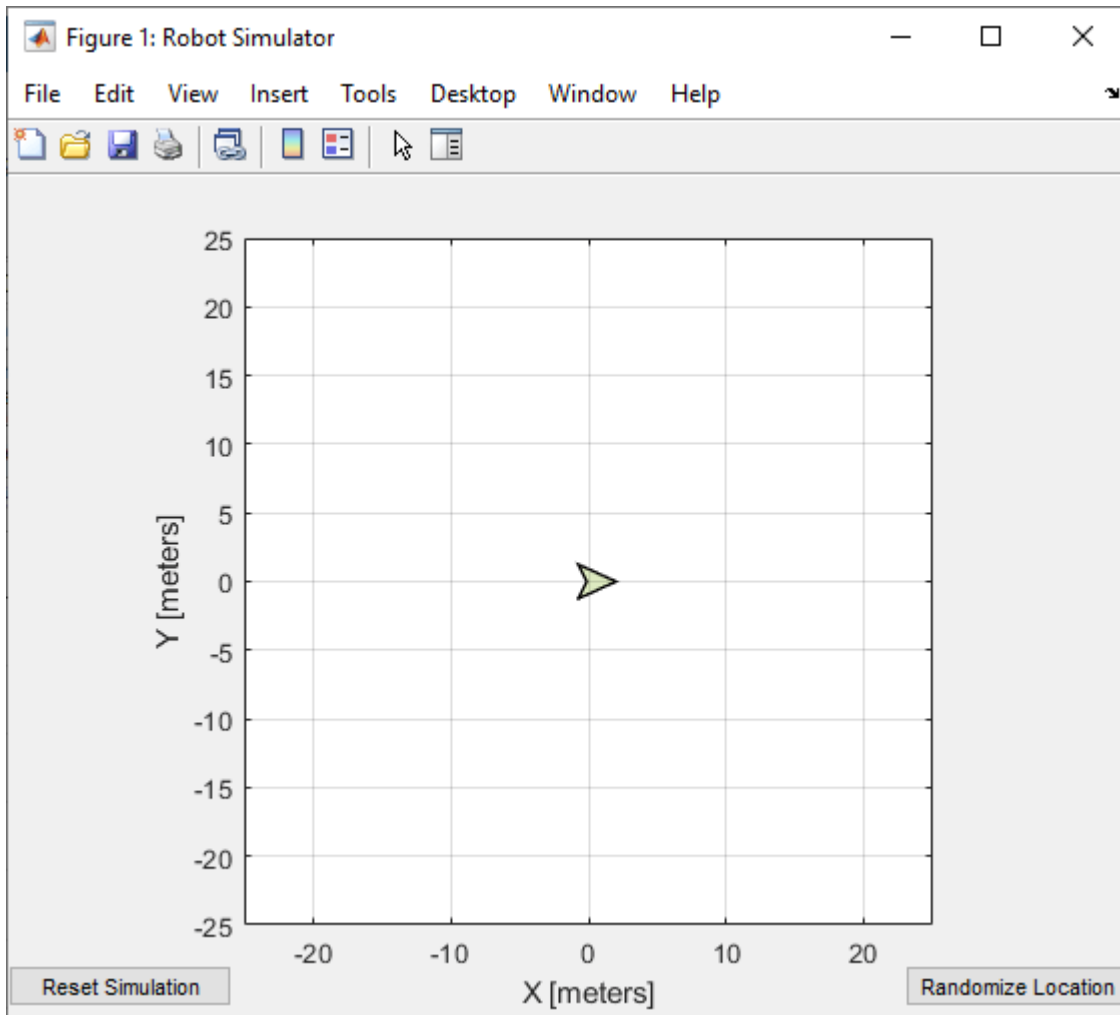
- **Note:** The `geometry_msgs/Twist` velocity command messages are received on the `/mobile_base/commands/velocity` topic.

Option B: Gazebo Simulator

Use a simulated TurtleBot® in Gazebo.

- See "Add, Build, and Remove Objects in Gazebo" on page 1-214 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**.
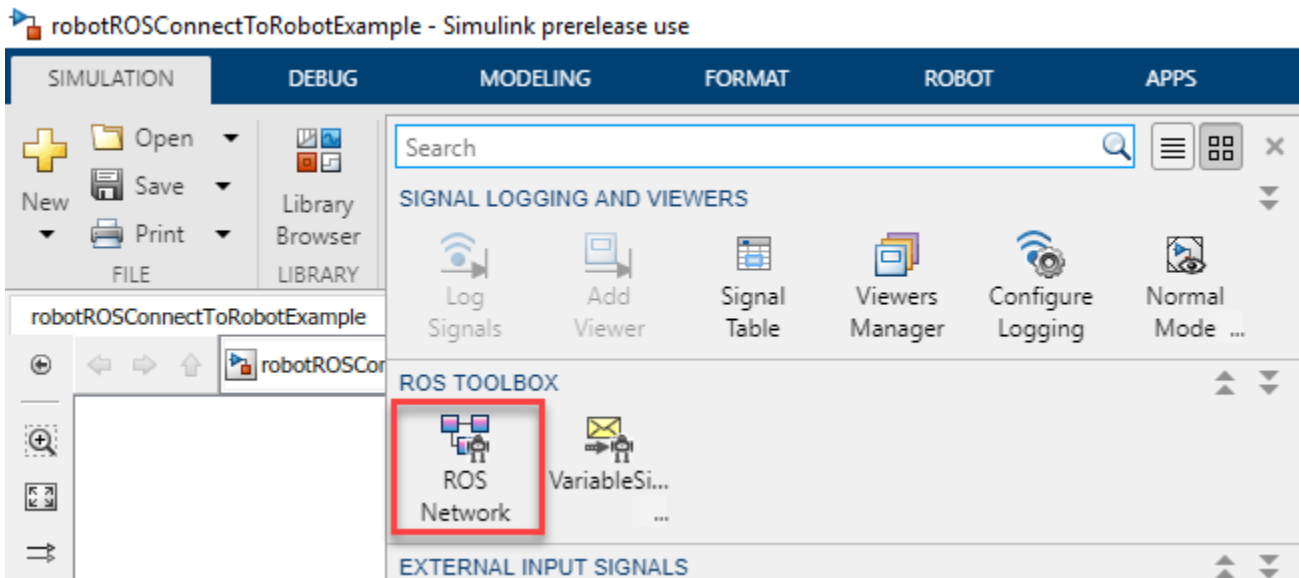
2. In the **ROS Master (ROS 1)** section, select `Custom` from the **Network Address** dropdown.

- 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.
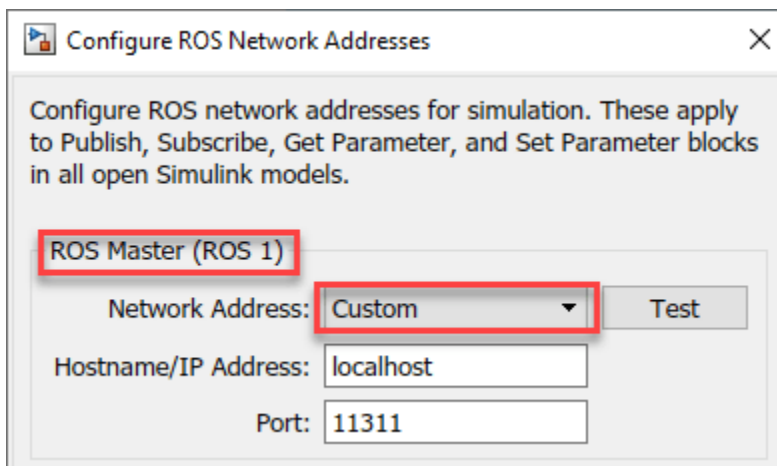
Send Velocity Commands To the Robot

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.
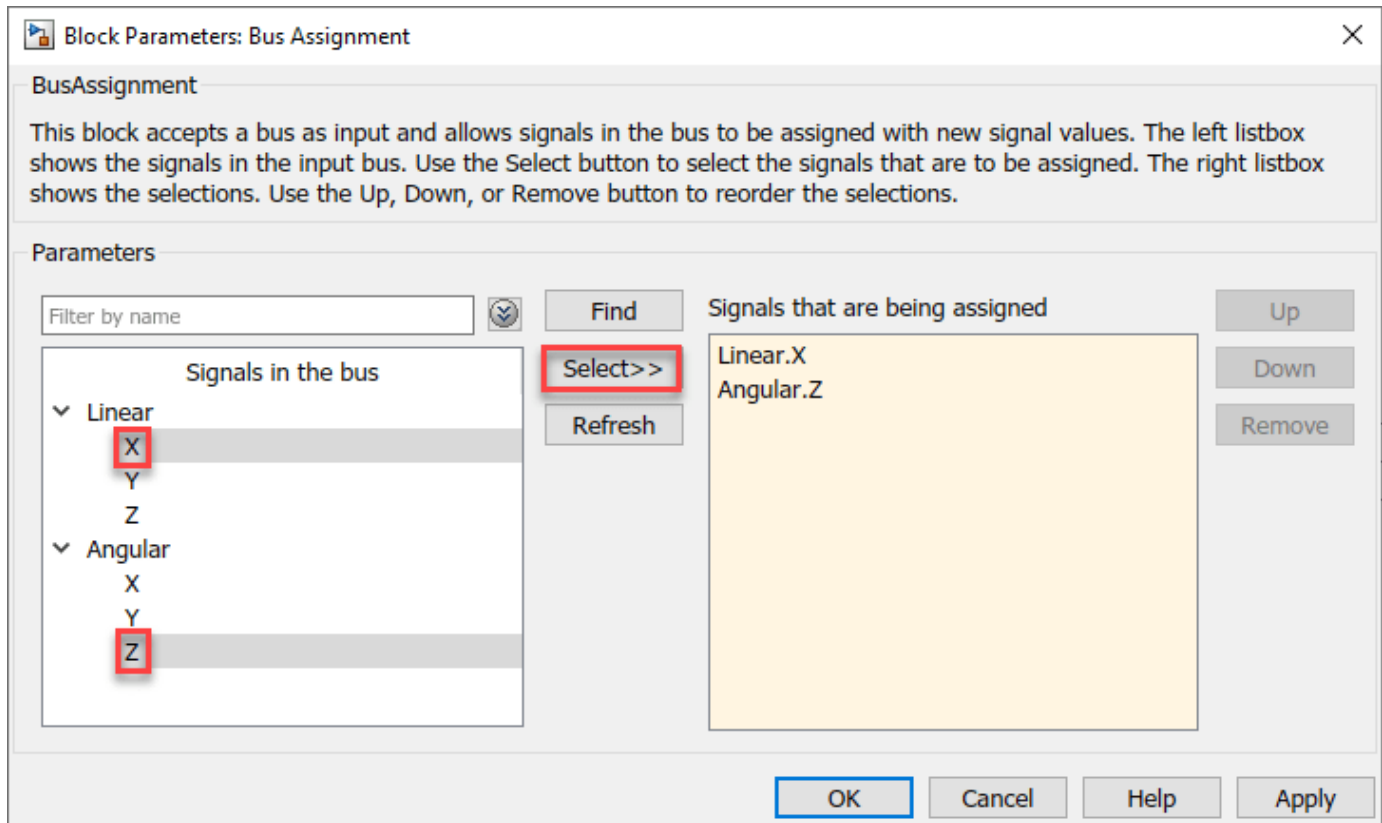
- Add a **Constant** block, a **Gain** block, and two **Slider Gain** blocks. Connect them together as shown in the figure, and set the **Gain** value to -1.
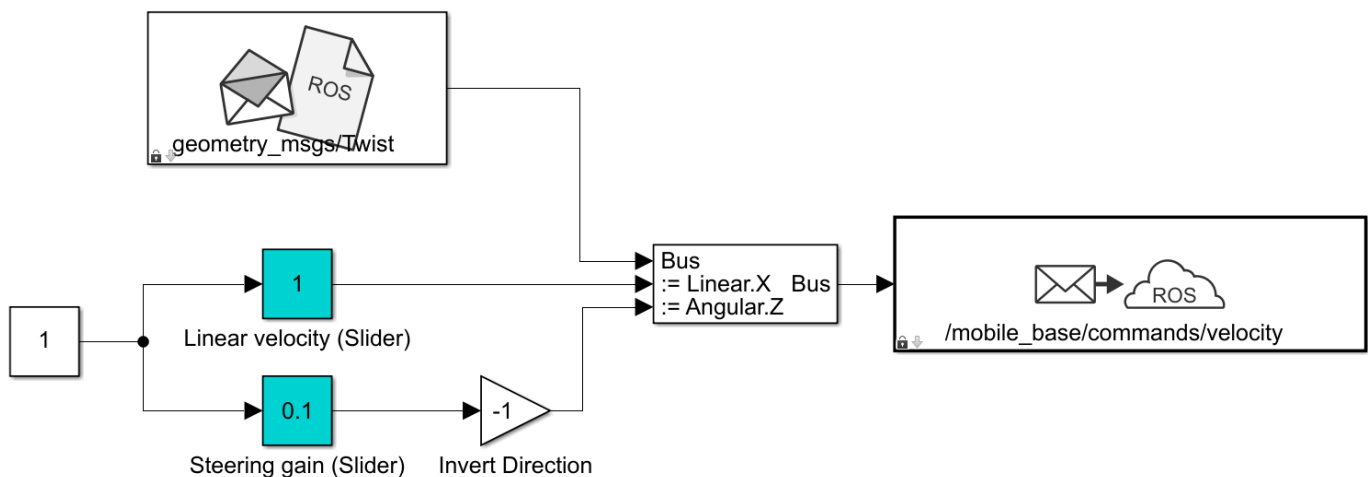
- Set the limits and current parameters of the linear velocity slider to 0.0 to 2.0, and 1.0 respectively. Set the corresponding parameters of the steering gain slider to -1.0 to 1.0, and 0.1.

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.
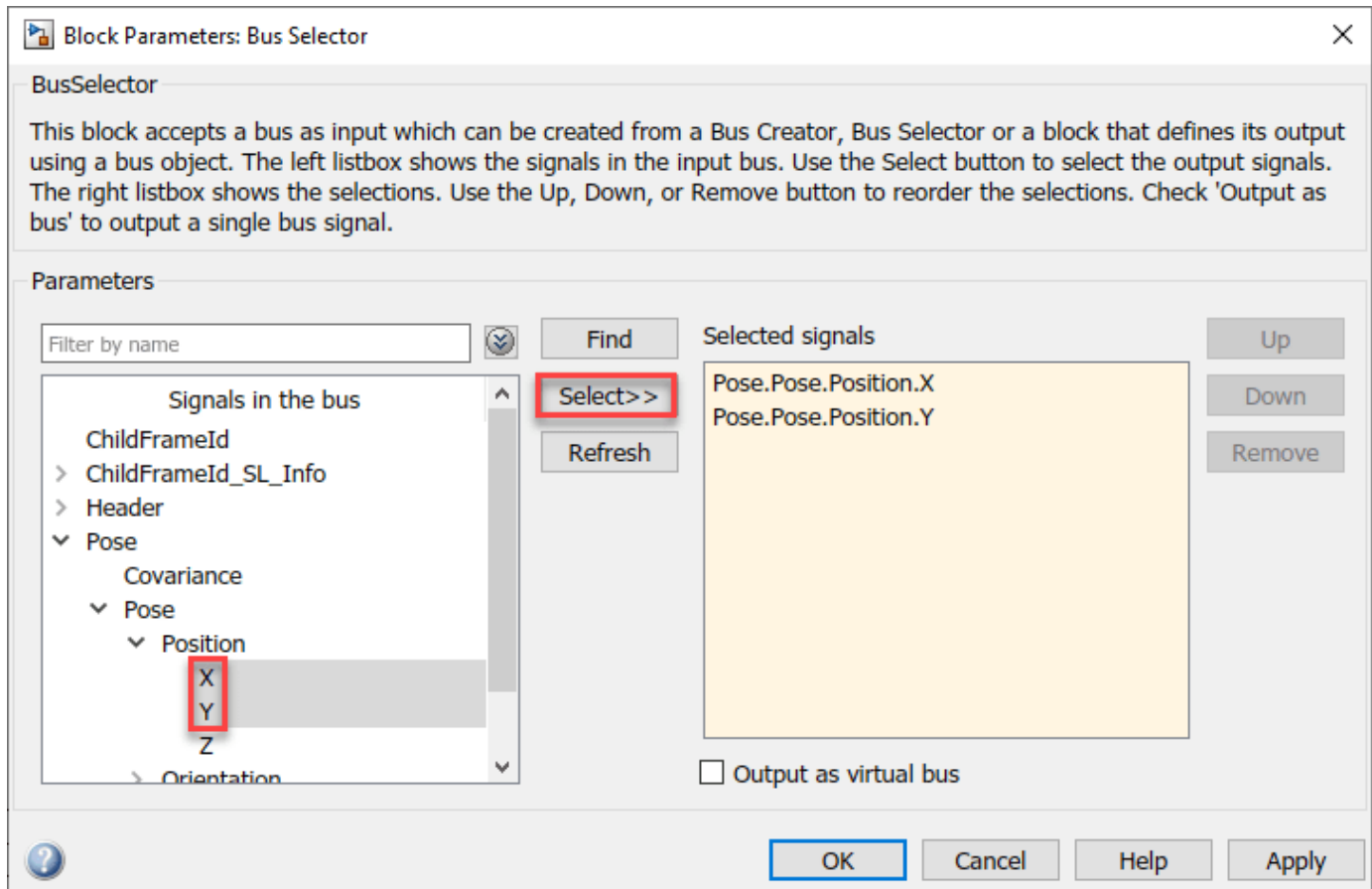
This figure shows the completed model. A pre-configured model is included for your convenience.

- **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 1-144.
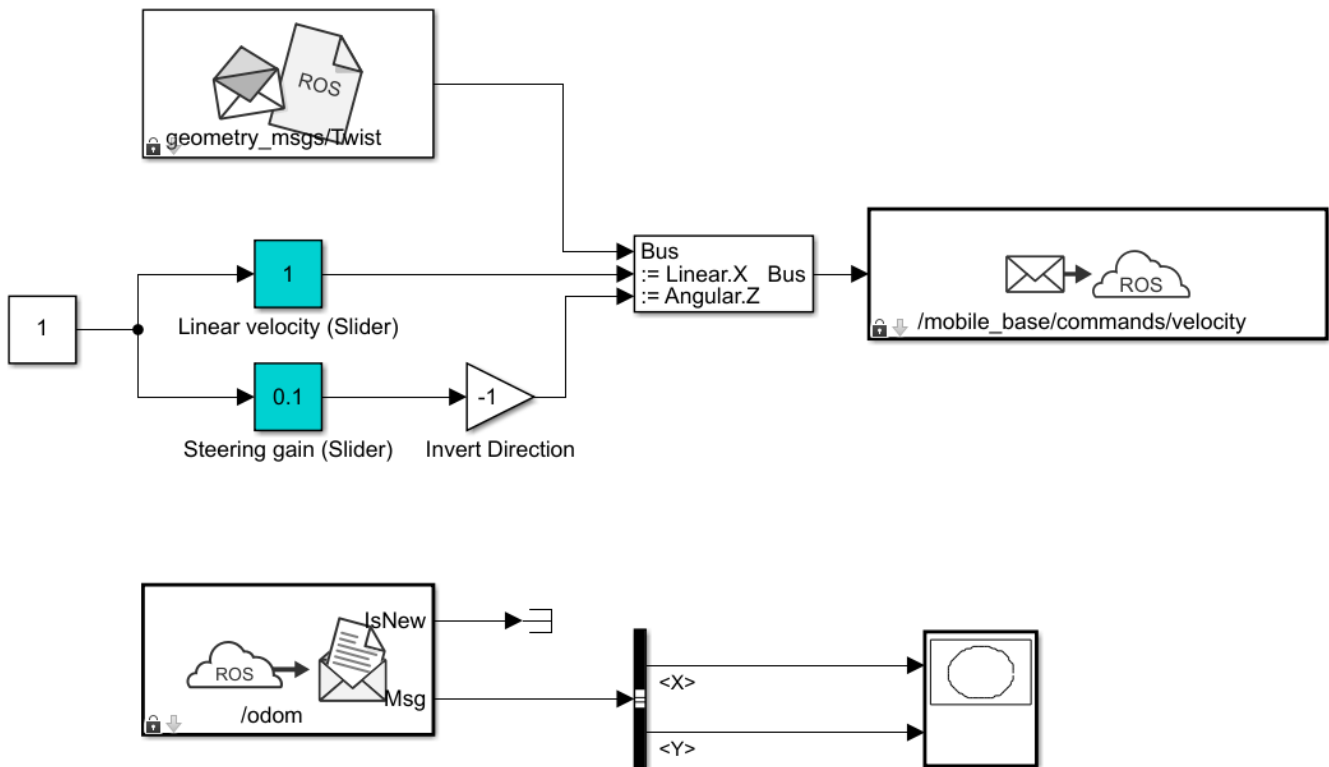
Configure and Run the Model

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.
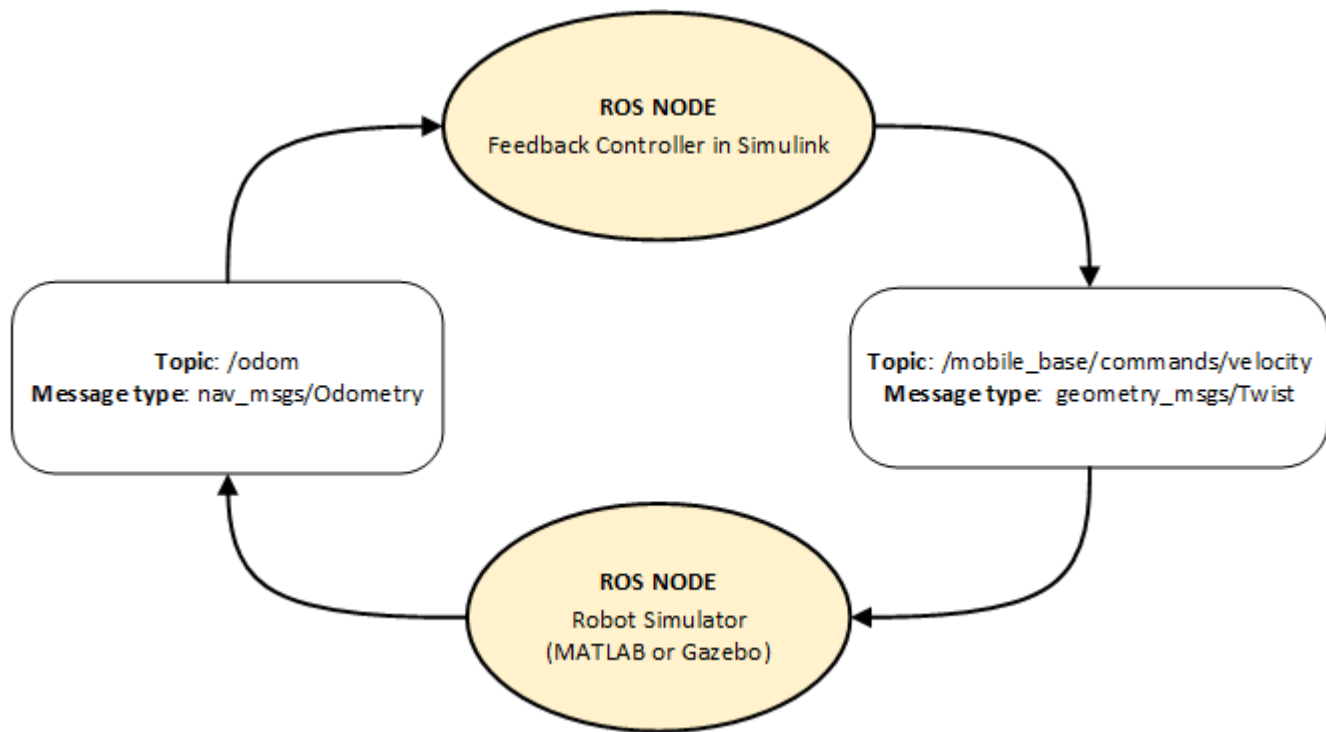
Start a Robot Simulator and Configure Simulink

Follow the steps in the "Connect to a ROS-enabled Robot from Simulink®" on page 1-141 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.

Fusion of Radar and Lidar Data Using ROS

Perform track-level sensor fusion on recorded lidar sensor data for a driving scenario recorded on a rosbag. This example uses the same driving scenario and sensor fusion as the "Track-Level Fusion of Radar and Lidar Data" (Sensor Fusion and Tracking Toolbox) example, but uses a prerecorded rosbag instead of the driving scenario simulation.

Extract Sensor Data from Rosbag

This provides an example rosbag containing lidar, radar, and vehicle data, and is approximately 33MB in size. Download the rosbag from the MathWorks website.

```
bagFile = matlab.internal.examples.downloadSupportFile("ros","rosbags/simulated_lidar_radar_driv
```

Access the rosbag and view the available topics.

```
bag = rosbag(bagFile);
disp(bag.AvailableTopics(:,["NumMessages", "MessageType"]))
```

	NumMessages	MessageType
/clock	114	rosgraph_msgs/Clock
/ego/lidar_scan	114	sensor_msgs/PointCloud2
/ego/radar_1/detections	114	cob_object_detection_msgs/DetectionArray
/ego/radar_2/detections	114	cob_object_detection_msgs/DetectionArray
/ego/radar_3/detections	114	cob_object_detection_msgs/DetectionArray
/ego/radar_4/detections	114	cob_object_detection_msgs/DetectionArray
/ego/state	114	nav_msgs/Odometry

The ego vehicle has one lidar and four radar sensors, as well as absolute positional information for itself. These messages can be extracted into separate arrays for later fusion. Because this rosbag is compressed to reduce file size, reading the messages may take a few seconds. Extract the messages as structures using the `DataFormat` name-value argument, which improves overall performance for ROS messages.

```
lidarBagSel = select(bag,"Topic","/ego/lidar_scan");
lidarMsgs = readMessages(lidarBagSel,"DataFormat","struct");
stateBagSel = select(bag,"Topic","/ego/state");
stateMsgs = readMessages(stateBagSel,"DataFormat","struct");
radarMsgs = cell(bag.AvailableTopics{"/ego/radar_1/detections","NumMessages"},4);
for idxRadar = 1:4
    radarBagSel = select(bag,"Topic",sprintf("/ego/radar_%d/detections",idxRadar));
    radarMsgs(:,idxRadar) = readMessages(radarBagSel,"DataFormat","struct");
end
```

Make timestamps to be used with fusion relative to the first message timestamp. Note that the bag `StartTime` cannot be used since that is the timestamp the first message was recorded at, which is later than the message timestamp.

```
clockBagSel = select(bag,"Topic","/clock");
clockMsg = readMessages(bag,1,"DataFormat","struct");
startTime = double(clockMsg{1}.Clock_.Sec) + double(clockMsg{1}.Clock_.Nsec)*1e-9;
```

Set Up Sensor Tracking and Fusion

Information about the sensors is known based on the vehicle set up, and needs to be put in a format usable by the tracking algorithm.

```
[lidarInfo, radarInfo, radarParameters] = helperRadarLidarInfo;
```

Radar and lidar tracking algorithms are necessary to process the high-resolution scans and determine the objects viewed in the scans without repeats. These algorithms are defined as helper functions. More details on the algorithms can be seen in the "Track-Level Fusion of Radar and Lidar Data" (Sensor Fusion and Tracking Toolbox) example.

```
radarTrackingAlgorithm = helperROSRadarTracker(radarInfo);
radarConfig = fuserSourceConfiguration("SourceIndex",1,...
    "IsInitializingCentralTracks",true,...
    "CentralToLocalTransformFcn",@central2radar,...
    "LocalToCentralTransformFcn",@radar2central);

lidarTrackingAlgorithm = helperLidarTrackingAlgorithm(lidarInfo);
lidarConfig = fuserSourceConfiguration("SourceIndex",2,...
    "IsInitializingCentralTracks",true);

fuser = trackFuser("SourceConfigurations",{radarConfig;lidarConfig},...
    "StateTransitionFcn",lidarTrackingAlgorithm.StateTransitionFcn,...
    "ProcessNoise",diag([1 3 1]),...
    "HasAdditiveProcessNoise",false,...
    "AssignmentThreshold",[250 inf],...
    "ConfirmationThreshold",[3 5],...
    "DeletionThreshold",[5 5],...
    "StateFusion","Custom",...
    "CustomStateFusionFcn",@helperRadarLidarFusionFcn);
```

Visualization

Set up figure and properties for visualization of sensor data using a helper function.

```
displayHelper = helperROSSensorFusionDisplay;
```

Perform Fusion on Sensor Messages

Iterate through the messages and run the sensor fusion algorithm. Watch the visualization to see the rosbag played back and the tracking of vehicles using the sensor fusion.

The sensors were triggered to all measure simultaneously, and all radar sensors published a message at each triggering, even if no detections were present. Because all the message arrays are all the same length, the processing of the sensor data is far simpler. If sensors triggered at distinct rates, connection issue occurred, or messages were received out of order, an intermediate step of synchronizing the sensor data may be necessary.

```
for idx = 1:numel(lidarMsgs)
    % Extract time on first sensor reading.
    % Make time relative to rosbag start time for easier tracking.
    lidarTimeStamp = lidarMsgs{idx}.Header.Stamp;
    lidarTime = double(lidarTimeStamp.Sec) + ...
        double(lidarTimeStamp.Nsec)*1e-9 - startTime;
    radarTimeStamp = radarMsgs{idx,1}.Header.Stamp;
    radarTime = double(radarTimeStamp.Sec) + ...
```

```matlab
            double(radarTimeStamp.Nsec)*1e-9 - startTime;

    % Extract vehicle state and modify structures for processing.
    egoPose = struct;
    stateMsg = stateMsgs{idx};
    positionMsg = stateMsg.Pose.Pose.Position;
    egoPose.Position = [positionMsg.X ; positionMsg.Y ; positionMsg.Z];
    % Orientation in degrees.
    orientQuat = rosReadQuaternion(stateMsg.Pose.Pose.Orientation);
    orientEul = eulerd(orientQuat,"XYZ","point");
    egoPose.Roll = orientEul(1);
    egoPose.Pitch = orientEul(2);
    egoPose.Yaw = orientEul(3);
    % By convension, nav_msgs/Odometry velocity is provided in the child
    % reference frame (the vehicle). The fusion requires velocity in the world
    % reference frame.
    velMsg = stateMsg.Twist.Twist.Linear;
    egoPose.Velocity = rotatepoint(orientQuat,...
        [velMsg.X velMsg.Y velMsg.Z]);

    % Extract point cloud from lidar for processing
    % This lidar provided no RGB or intensity information
    lidarXYZPoints = rosReadXYZ(lidarMsgs{idx});
    ptCloud = pointCloud(lidarXYZPoints);

    % Extract radar detections into a single array using metadata to
    % specify the source sensor.
    nDetections = sum(cellfun(@(msg) numel(msg.Detections),radarMsgs(idx,:)));
    radarDetections = cell(nDetections,1);  % Preallocate
    idxDetection = 1;
    for idxRadar = 1:size(radarMsgs,2)
        for idxRadarDetection = 1:numel(radarMsgs{idx,idxRadar}.Detections)
            detMsg = radarMsgs{idx,idxRadar}.Detections(idxRadarDetection);
            detTime = double(detMsg.Header.Stamp.Sec) + ...
                double(detMsg.Header.Stamp.Nsec)*1e-9 - startTime;
            measureMsg = detMsg.Pose.Pose.Position;
            measurement = [measureMsg.X ; measureMsg.Y ; measureMsg.Z];
            % Measurement noise is stored in the bounding box field due to
            % this message type containing Pose instead of PoseCovariance.
            measureNoise = diag([detMsg.BoundingBoxLwh.X detMsg.BoundingBoxLwh.Y detMsg.BoundingB
            % Store signal-to-noise ratio in Score field.
            objectAttributes = struct("TargetIndex",detMsg.Id,"SNR",detMsg.Score);
            radarDetections{idxDetection} = objectDetection(detTime,measurement,...
                "MeasurementNoise",measureNoise,...
                "SensorIndex",idxRadar,...
                "ObjectClassID",0,...
                "ObjectAttributes",{objectAttributes},...
                "MeasurementParameters",{radarParameters(idxRadar)});
            idxDetection = idxDetection + 1;
        end
    end

    % Generate sensor tracks and analysis information like the bounding box
    % detections and point cloud segmentation information.
    radarTracks = radarTrackingAlgorithm(egoPose, radarDetections, radarTime);
    [lidarTracks, lidarDetections, segmentationInfo] = ...
        lidarTrackingAlgorithm(egoPose, ptCloud, lidarTime);
    localTracks = [radarTracks ; lidarTracks];
```

```matlab
    % Update the fuser. The first call must contain one local track.
    if ~(isempty(localTracks) && ~isLocked(fuser))
        fusedTracks = fuser(localTracks,lidarTime);
    else
        fusedTracks = objectTrack.empty(0,1);
    end

    % Update the display
    updateSensorData(displayHelper,ptCloud,radarDetections)
    plotTracks(displayHelper,radarTracks,lidarTracks,fusedTracks,egoPose)
end
```
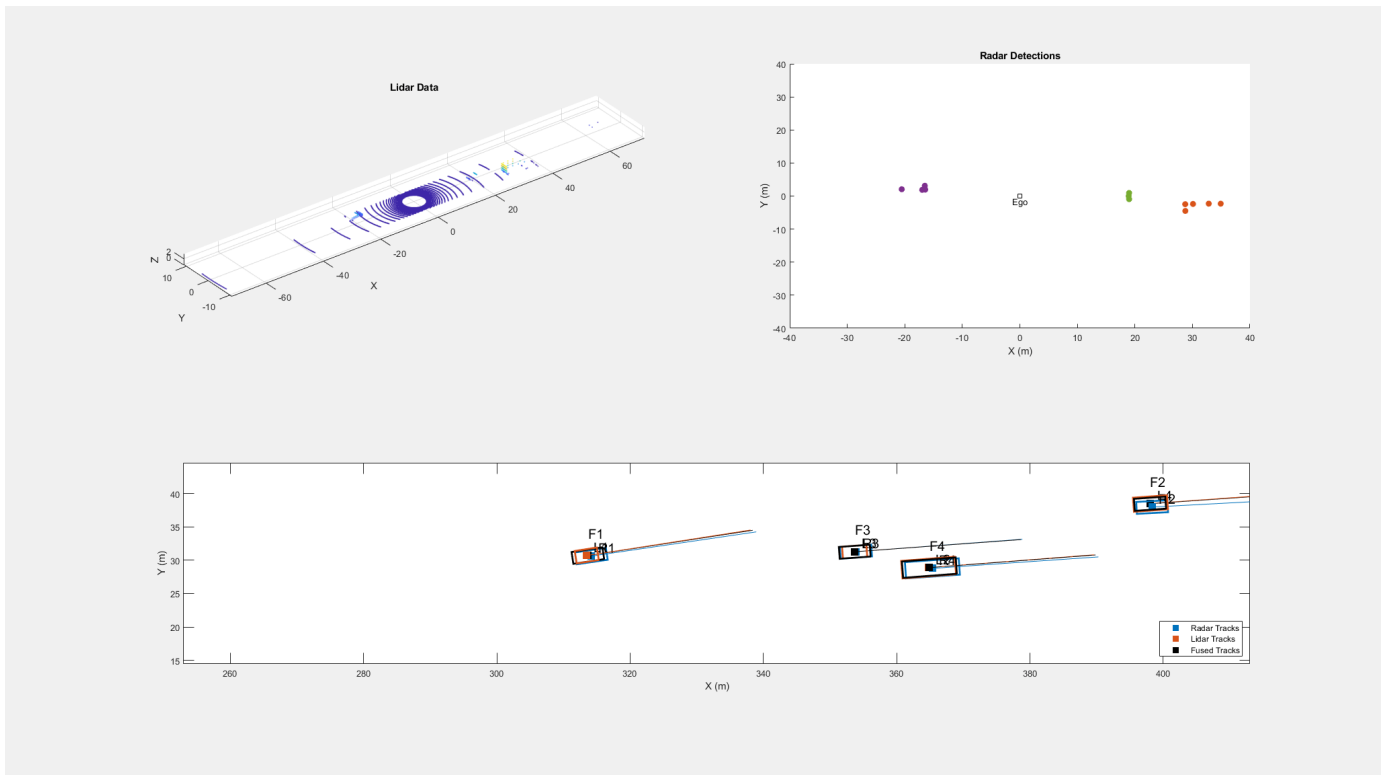

Utility Functions

The following function definitions are used in the above script.

radar2central

```matlab
function centralTrack = radar2central(radarTrack)
% Initialize a track of the correct state size
centralTrack = objectTrack('State',zeros(10,1),...
    'StateCovariance',eye(10));

% Sync properties of radarTrack except State and StateCovariance with
% radarTrack See syncTrack defined below.
centralTrack = syncTrack(centralTrack,radarTrack);

xRadar = radarTrack.State;
PRadar = radarTrack.StateCovariance;
```

```
H = zeros(10,7); % Radar to central linear transformation matrix
H(1,1) = 1;
H(2,2) = 1;
H(3,3) = 1;
H(4,4) = 1;
H(5,5) = 1;
H(8,6) = 1;
H(9,7) = 1;

xCentral = H*xRadar;  % Linear state transformation
PCentral = H*PRadar*H'; % Linear covariance transformation

PCentral([6 7 10],[6 7 10]) = eye(3); % Unobserved states

% Set state and covariance of central track
centralTrack.State = xCentral;
centralTrack.StateCovariance = PCentral;
end
```

central2radar

```
function radarTrack = central2radar(centralTrack)
% Initialize a track of the correct state size
radarTrack = objectTrack('State',zeros(7,1),...
    'StateCovariance',eye(7));

% Sync properties of centralTrack except State and StateCovariance with
% radarTrack See syncTrack defined below.
radarTrack = syncTrack(radarTrack,centralTrack);

xCentral = centralTrack.State;
PCentral = centralTrack.StateCovariance;

H = zeros(7,10); % Central to radar linear transformation matrix
H(1,1) = 1;
H(2,2) = 1;
H(3,3) = 1;
H(4,4) = 1;
H(5,5) = 1;
H(6,8) = 1;
H(7,9) = 1;

xRadar = H*xCentral;  % Linear state transformation
PRadar = H*PCentral*H'; % Linear covariance transformation

% Set state and covariance of radar track
radarTrack.State = xRadar;
radarTrack.StateCovariance = PRadar;
end
```

syncTrack

```
function tr1 = syncTrack(tr1,tr2)
props = properties(tr1);
notState = ~strcmpi(props,'State');
notCov = ~strcmpi(props,'StateCovariance');
```

```
props = props(notState & notCov);
for i = 1:numel(props)
    tr1.(props{i}) = tr2.(props{i});
end
end
```

MATLAB Programming for Code Generation

This example shows the recommended workflow for generating a standalone executable from MATLAB® code that contains ROS interfaces.

Prerequisites

- This example requires MATLAB Coder™.
- You must have access to a C/C++ compiler that is configured properly. You can use `mex -setup cpp` to view and change the default compiler. For more details, see Change Default Compiler.

Overview

A subset of ROS MATLAB functions such as `rossubscriber`, `rospublisher`, and `rosrate` support C++ code generation. To create a standalone ROS node from MATLAB code, follow these steps:

- Create the entry-point function for creating a standalone application. The entry-point function must not have any inputs and must not return any outputs.
- Add the `%#codegen` directive to the MATLAB function to indicate that it is intended for code generation. This directive also enables the MATLAB code analyzer to identify warnings and errors specific to MATLAB for code generation.
- Modify the ROS functions to use message structures.
- Identify MATLAB code that does not support C++ code generation and modify the MATLAB code to use functions or constructs that support code generation.
- Create a MATLAB Coder configuration object and specify the hardware as `'Robot Operating System (ROS)'`.
- Use `codegen` command to generate a stand-alone executable.

Generate Code for Subscriber

Consider the following MATLAB code.

```matlab
function mynode(numIterations)
%mynode Receive and display sensor_msgs/JointState messages
rosinit;

% Trajectory points to be published
sub = rossubscriber("/servo");

% Display position
for k = 1:numIterations
    msg = receive(sub);
    if ~isempty(msg.Position > 0)
        disp(msg.Position(1))
    else
        disp("Received empty message..");
    end
end
rosshutdown;
end
```

This MATLAB code receives `sensor_msgs/JointState` messages published to the `/servo` topic and displays the first element of the position in a loop. The function has an input argument that sets the number of iterations. To create a stand-alone ROS node, modify the code as follows:

- Eliminate the input argument `numIterations` using a `while` loop.
- Add the `%#codegen` directive.
- Specify the message type for the `rossubscriber` call.
- Use message structures instead of message classes.
- Eliminate `rosinit` and `rosshutdown` calls that do not generate code.
- Replace the `disp` function which does not support code generation, with `fprintf`.

Note that you execute the rosinit and rosshutdown functions only once in a MATLAB session. Avoid `rosinit` and `rosshutdown` functions for code intended for standalone deployment. In stand-alone deployment, individual ROS nodes are not expected to start or stop the ROS master. If you need to include `rosinit` and `roshutdown` calls in your MATLAB code for interpreted execution, declare them as `coder.extrinsic` (MATLAB Coder) functions at the top of your entry-point function.

The entry-point function cannot take any inputs or return outputs. A standalone ROS node executable is intended to be launched outside MATLAB, such as from a system command terminal, and therefore cannot take any MATLAB inputs or return MATLAB outputs.

To eliminate the input argument `numIterations`, replace the `for` loop with an infinite `while` loop. For stand-alone deployment, the generated node is expected to run until you terminate it externally. A good programming practice is to replace the `for` loops with a `while` loop for standalone deployment.

The `rossubscriber` function needs the message type to be specified as a compile time constant for code generation. The function uses this information to create the return message type for `receive` calls. The `rossubscriber` function supports code generation for message structures only. To return a message structure, specify the name-value pair argument, `"DataFormat","struct"`, when creating a subscriber.

After you modify the code, the MATLAB code for the entry-point function is as follows:

```matlab
function mynode
%mynode Receive and display sensor_msgs/JointState messages
%#codegen

% Trajectory points to publish
sub = rossubscriber("/servo","sensor_msgs/JointState","DataFormat","struct");

% Display position
while (1)
    msg = receive(sub);
    if ~isempty(msg.Position > 0)
        fprintf("Position = %f\n",msg.Position(1))
    else
        fprintf("Received empty message..\n");
    end
end
end
```

Create a MATLAB Coder configuration that uses `"Robot Operating System (ROS)"` hardware. Set the `HardwareImplementation.ProdHWDeviceType` parameter of the MATLAB Coder configuration object for the intended deployment hardware. For example, if you are deploying

generated code to a Windows computer set `HardwareImplementation.ProdHWDeviceType` to
`"Intel->x86-64 (Windows64)"`. To generate code, execute the following commands:

```
cfg = coder.config("exe");
cfg.Hardware = coder.hardware("Robot Operating System (ROS)");
cfg.Hardware.DeployTo = "Localhost";
cfg.Hardware.BuildAction = "Build and run";
cfg.HardwareImplementation.ProdHWDeviceType = "Intel->x86-64 (Windows64)";
cfg.HardwareImplementation.ProdLongLongMode = true; % Use 'long long' for Int64 or Uint64 data ty
codegen mynode -config cfg
```

You can send messages to `mynode` using the following command on a ROS terminal:

```
rostopic pub /servo sensor_msgs/JointState -r 1 "{header:{seq: 0, stamp:{secs: 0,nsecs: 0},frame_
```

Generate Code for Publisher

Consider the following MATLAB code.

```
function mypubnode(updateRate)
%mypubnode Publish joint trajectory messages

% Create publisher
pub = rospublisher("/traj","trajectory_msgs/JointTrajectory");

% Create a message
msg = rosmessage("trajectory_msgs/JointTrajectory");
msg.JointNames{1} = 'Left';
msg.JointNames{2} = 'Right';
trajMsg = rosmessage("trajectory_msgs/JointTrajectoryPoint");
r = rosrate(updateRate);
while (1)
    msg.Header.Stamp = rostime("now");
    x = rand;
    y = rand;
    trajMsg.Positions = [x y -x -y];
    msg.Points = [trajMsg trajMsg];
    send(pub,msg);
    waitfor(r);
end
end
```

This MATLAB code publishes `trajectory_msgs/JointTrajectory` messages to the `/traj` topic.
You can set the message publishing rate externally. The `trajectory_msgs/JointTrajectory`
message is a nested message that has the following subfields:

```
>> rosmsg show trajectory_msgs/JointTrajectory
std_msgs/Header Header
char[] JointNames
JointTrajectoryPoint[] Points
```

The message can accommodate multiple joints and multiple trajectory points. If the number of joints
is M and the number of trajectory points for each joint is N, `trajMsg` has the dimensions M-by-1 and
`trajMsg.Positions` has the dimensions N-by-1. For this example, publish two trajectory points per
joint.

The function has an input argument that sets the number of iterations. To create a stand-alone ROS
node, modify the code as follows:

- Eliminate the input argument `updateRate` by directly specifying it within function body.
- Add the `%#codegen` directive.
- Use message structures instead of message classes.
- Eliminate any message fields that use cell strings to prepare for code generation.
- Grow variable-size fields of message structures in the correct dimension.

To eliminate the input argument `updateRate`, specify the update rate using a constant literal in the entry-point function body. To modify the `updateRate`, you must re-generate code from the entry-point function.

Modify the `rospublisher`, `rosmessage` and `rosrate` functions to use message structures. The `trajectory_msgs/JointTrajectory` message structure has a field, `JointNames`, which is of type cell string. Message fields of this type are not supported for code generation due to MATLAB Coder limitations. In order to generate code for `mypubnode` function, avoid the use of `JointNames` fields in code generation using `coder.target` (MATLAB Coder) function.

The original `mypubnode` function grows variable-size fields of the message structs in the wrong dimension. As an illustration, create a message of type `trajectory_msgs/JointTrajectory` at the MATLAB command line. Note that the first dimension of the variable-size fields is of size 0 while the second dimension is of size 1:

```
msg = rosmessage('trajectory_msgs/JointTrajectory')

msg =

  ROS JointTrajectory message with properties:

    MessageType: 'trajectory_msgs/JointTrajectory'
         Header: [1×1 Header]
         Points: [0×1 JointTrajectoryPoint]
     JointNames: {0×1 cell}

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

The fields with dimensions 0-by-1 grow in the first dimension. The original code grows the `Points` and `Positions` fields in the second dimension, which works in interpreted mode, but is not supported for code generation. You get the following error message if you attempt to grow variable-size fields of a message structure in the wrong dimension:

```
??? This assignment writes a 'trajectory_msgs_JointTrajectoryPointStruct_T' value into a 'struct
does not support changing types through assignment. Check preceding assignments or input type sp
mismatches.
```

After you modify the code, the MATLAB code for the entry-point function is as follows:

```
function mypubnode
%mypubnode Publish joint trajectory messages
%#codegen

% Create publisher
pub = rospublisher("/traj","trajectory_msgs/JointTrajectory","DataFormat","struct");

% Create a msg
msg = rosmessage("trajectory_msgs/JointTrajectory","DataFormat","struct");
if isempty(coder.target)
```

```
    msg.JointNames{1} = 'Left';
    msg.JointNames{2} = 'Right';
end
trajMsg = rosmessage("trajectory_msgs/JointTrajectoryPoint","DataFormat","struct");
r = rosrate(1);
while (1)
    msg.Header.Stamp = rostime("now","DataFormat","struct");
    x = rand;
    y = rand;
    trajMsg.Positions = [x; y; -x; -y]; % Grow variable-size fields in the correct dimension
    msg.Points = [trajMsg; trajMsg];    % Grow variable-size fields in the correct dimension
    send(pub,msg);
    waitfor(r);
end
end
```

Create a MATLAB Coder configuration that uses `"Robot Operating System (ROS)"` hardware. Set the `HardwareImplementation.ProdHWDeviceType` parameter of the MATLAB Coder configuration object for the intended deployment hardware. For example, if you are deploying generated code to a Windows computer set `HardwareImplementation.ProdHWDeviceType` to `"Intel->x86-64 (Windows64)"`. To generate code execute the following commands:

```
cfg = coder.config("exe");
cfg.Hardware = coder.hardware("Robot Operating System (ROS)");
cfg.Hardware.DeployTo = "Localhost";
cfg.Hardware.BuildAction = "Build and run";
cfg.HardwareImplementation.ProdHWDeviceType = "Intel->x86-64 (Windows64)";
cfg.HardwareImplementation.ProdLongLongMode = true; % Use 'long long' for Int64 or Uint64 data ty
codegen mypubnode -config cfg
```

You can examine the contents of the messages published by `mypubnode` using `rostopic echo /traj` command on a ROS terminal.

Additional Considerations

Code generation for messages containing cell strings: MATLAB Coder does not support code generation for cell strings. You can send and receive ROS messages containing cell strings using `rossubsriber` and `rospublisher,` but you cannot access the fields that are of type cell string. For instance, in a `sensor_msgs/JointState` message, you cannot read or write to the field, `Name`, which is of type cell string in MATLAB representation.

Variable-size message fields: Do not use indexing past the array dimension pattern to grow the size of variable-size arrays in message fields. Use the following form:

```
msg = rosmessage('sensor_msgs/JointState','DataFormat','struct');
msg.Position = [0.1; -0.1];
...
msg.Position = [msg.Position; 0.2]; % Add new data point
```

For more information, see "Incompatibilities with MATLAB in Variable-Size Support for Code Generation" (MATLAB Coder). Also, for the MATLAB Coder configuration object, you must set

`EnableVariableSizing` property to `true` and enable dynamic memory allocation for variable-size arrays.

See Also

Related Examples

*   "Generate a Standalone ROS Node from MATLAB" on page 1-164

Generate a Standalone ROS Node from MATLAB

This example shows how to generate C++ code for a standalone ROS node from a MATLAB function. It then shows how to build and run the ROS node on a Windows® machine.

Prerequisites

- This example requires MATLAB 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 for ROS Toolbox examples (see "Get Started with Gazebo and Simulated TurtleBot" on page 1-206 for instructions).
- You must have access to a C/C++ compiler that is configured properly. You can use `mex -setup cpp` to view and change the default compiler. For more details, see Change Default Compiler.

Control a ROS-Enabled Robot with a Function

Open the function `robotROSFeedbackControl`, which contains a proportional controller introduced in the "Feedback Control of a ROS-Enabled Robot" on page 1-149 example. This function subscribes to the `/odom` topic to get the current odometry status of the robot, and then publishes the output of a proportional controller as a `geometry_msgs/Twist` message to the `/cmd_vel` topic. Doing so provides the control commands for the robot to move towards the desired position.

Copy the `robotROSFeedbackControl` function to your local directory and change the `defaultDesiredPos` variable to the desired coordinates.

Start an a Gazebo Empty World from the Linux virtual machine by opening the **Gazebo Empty** application on the desktop. In the Linux virtual machine for ROS Toolbox, the robot is located at the `[0,0]` location by default.

Create a `rosdevice` object specifying the `deviceAddress`, `username`, and `password` values of the Virtual Machine running Gazebo. This establishes an SSH connection between the ROS device and MATLAB.

```
gazeboVMDevice = rosdevice('172.18.250.139','user','password');
```

Create a MATLAB ROS node in the same ROS network as the virtual machine. See the list of available ROS nodes.

```
rosinit(gazeboVMDevice.DeviceAddress,11311)
```

Initializing global node /matlab_global_node_38424 with NodeURI http://172.18.250.141:65043/ and

```
rosnode list
```

```
/gazebo
/gazebo_gui
/matlab_global_node_38424
/rosout
```

Run the controller and observe that the robot is moving towards the published destination. At the same time, observe the trajectory of the robot that shows up in a MATLAB figure. Keep this figure open to compare the behavior of MATLAB execution and the generated executable node.

```
robotROSFeedbackControl
```
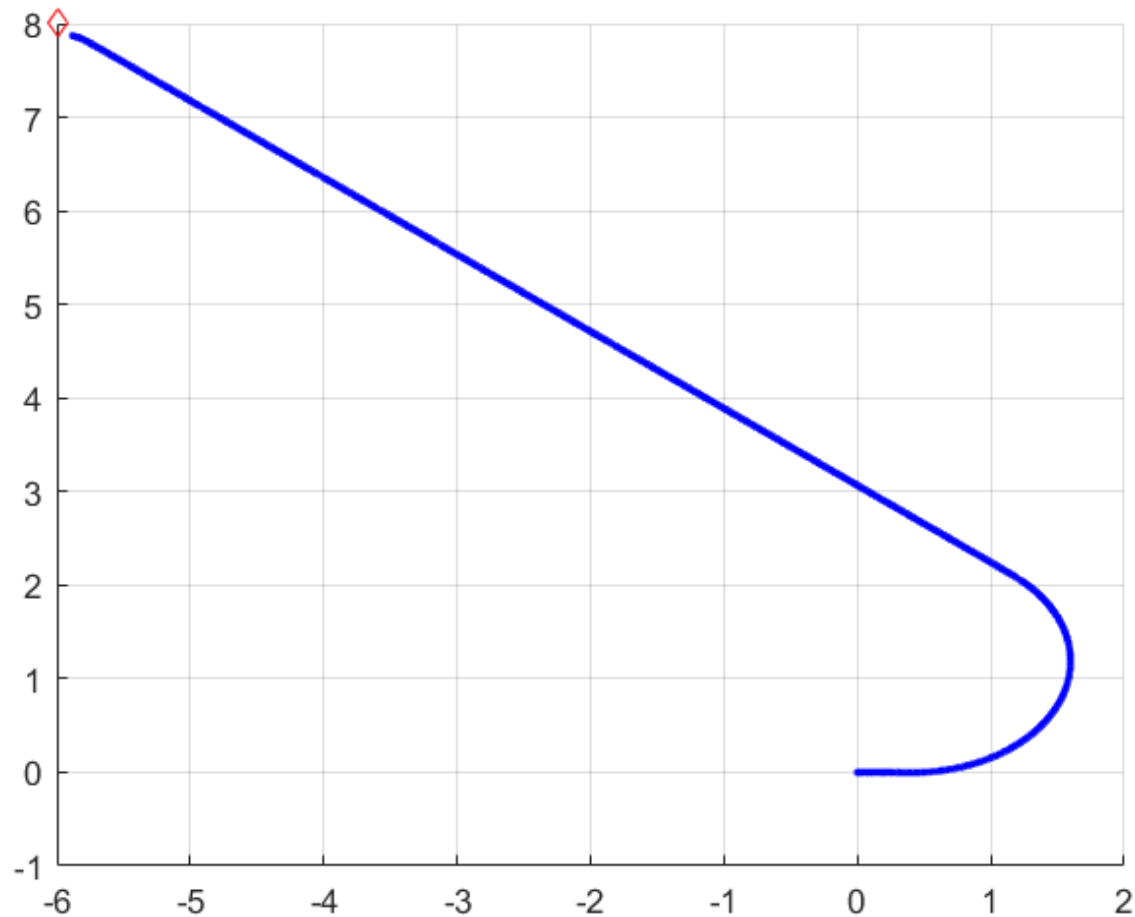
You can change the desired destination while the robot is moving. To do so, open a new terminal from the virtual machine, source the ROS repository, and publish the new destination coordinates in the form of a `std_msgs/Float64MultiArray` message to the `/dest` topic.

```
~$ source /opt/ros/noetic/local_setup.bash
~$ rostopic pub -1 /dest std_msgs/Float64MultiArray "{data:[0,0]}"
```

You can also tweak the `distanceThre`, `linearVelocity`, and `rotationGain` values in `robotROSFeedbackControl.m` to obtain the desired robot behavior. For the proportional controller in this example, the following parameter ranges provide robust performance. Alternatively, you can replace the proportional controller with a custom controller for performance comparison. To observe the behavior, reset the robot on the virtual machine by pressing **Ctrl-R** in Gazebo.

```
distanceThre: 0<x<1
linearVelocity: 0<x<3
rotationGain: 0<x<6
```

The controller loop will terminate after the goal is reached, if more than 40 seconds has elapsed since the start time. Alternatively, you can terminate the controller any time using **Ctrl-C** or typing in the

following command in the terminal from the virtual machine. Note that if you open a new terminal in the virtual machine, you must source the ROS repository.

```
~$ rostopic pub -1 /stop std_msgs/Bool "1"
```

Create a Function for Code Generation

To generate a standalone C++ node, modify the function to make it compatible for code generation.

- Because objects do not support code generation, replace them with structs for `rospublisher`, `rossubscriber`, and `rosmessage`. Specify the name-value pair `"DataFormat","struct" in the` respective function calls to create them as structures.
- Save the modified MATLAB function to `robotROSFeedbackControlCodegen.m`. Ensure any other modifications that you made in `robotROSFeedbackControl` function are reflected in `robotROSFeedbackControlCodegen`.

Generate Executable for `robotROSFeedbackControlCodegen`

Reset the Gazebo scene after simulation using the `/gazebo/reset_simulation` service. Create a `rossvcclient` object for the service and use the `call` object function to call the service and reset the Gazebo simulation scene.

```
gazeboResetClient = rossvcclient('/gazebo/reset_simulation','DataFormat','struct');
call(gazeboResetClient);
```

Generate an executable node for the `robotROSFeedbackControlCodegen` function. Specify the hardware as `'Robot Operating System (ROS)'`. Set the build action to `Build and run` so that the ROS node starts running after you generate it. Specify the coder configuration parameters for remote deployment. This example specifies the remote device parameters of the virtual machine running Gazebo. Note that the actual values might be different for your remote device. Verify them before deployment.

```
cfg = coder.config('exe');
cfg.Hardware = coder.hardware('Robot Operating System (ROS)');
cfg.Hardware.BuildAction = 'Build and run';
cfg.Hardware.RemoteDeviceAddress = '172.18.250.139';
cfg.Hardware.RemoteDeviceUsername = 'user';
cfg.Hardware.RemoteDevicePassword = 'password';
cfg.Hardware.DeployTo = 'Remote Device';
codegen robotROSFeedbackControlCodegen -args {} -config cfg

Connecting to ROS device '172.18.250.139'.
Using Catkin workspace '~/catkin_ws' to build ROS node.
---
Transferring generated code for 'robotROSFeedbackControlCodegen' to ROS device.
Starting build for ROS node.
---
tar: Ignoring unknown extended header keyword 'SCHILY.fflags'
Catkin project directory: /home/user/catkin_ws/src/robotrosfeedbackcontrolcodegen
tar: Ignoring unknown extended header keyword 'SCHILY.fflags'
tar: Ignoring unknown extended header keyword 'SCHILY.fflags'
tar: Ignoring unknown extended header keyword 'SCHILY.fflags'
-- Using CATKIN_DEVEL_PREFIX: /home/user/catkin_ws/devel
-- Using CMAKE_PREFIX_PATH: /home/user/catkin_ws/devel;/opt/ros/noetic
-- This workspace overlays: /home/user/catkin_ws/devel;/opt/ros/noetic
-- Found PythonInterp: /usr/bin/python3 (found suitable version "3.8.10", minimum required is "3'
-- Using PYTHON_EXECUTABLE: /usr/bin/python3
```

```
-- Using Debian Python package layout
-- Using empy: /usr/lib/python3/dist-packages/em.py
-- Using CATKIN_ENABLE_TESTING: ON
-- Call enable_testing()
-- Using CATKIN_TEST_RESULTS_DIR: /home/user/catkin_ws/build/test_results


-- Forcing gtest/gmock from source, though one was otherwise available.
-- Found gtest sources under '/usr/src/googletest': gtests will be built
-- Found gmock sources under '/usr/src/googletest': gmock will be built
-- Found PythonInterp: /usr/bin/python3 (found version "3.8.10")
-- Using Python nosetests: /usr/bin/nosetests3
-- catkin 0.8.10
-- BUILD_SHARED_LIBS is on
-- BUILD_SHARED_LIBS is on


-- ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
-- ~~  traversing 41 packages in topological order:
-- ~~  - husky_desktop (metapackage)
-- ~~  - husky_simulator (metapackage)
-- ~~  - kortex_move_it_config (metapackage)
-- ~~  - mav_comm (metapackage)
-- ~~  - rotors_description
-- ~~  - rotors_simulator (metapackage)
-- ~~  - husky_msgs
-- ~~  - mav_state_machine_msgs
-- ~~  - mav_system_msgs
-- ~~  - rotors_comm
-- ~~  - mav_msgs
-- ~~  - robotrosfeedbackcontrolcodegen
-- ~~  - husky_control
-- ~~  - husky_description
-- ~~  - husky_gazebo
-- ~~  - husky_navigation
-- ~~  - husky_viz
-- ~~  - kortex_control
-- ~~  - kortex_description
-- ~~  - kortex_gazebo
-- ~~  - rotors_evaluation
-- ~~  - rqt_rotors
-- ~~  - mav_planning_msgs
-- ~~  - opencv_node
-- ~~  - rotors_control
-- ~~  - rotors_hil_interface
-- ~~  - rotors_joy_interface
-- ~~  - gazebo_version_helpers
-- ~~  - gazebo_grasp_plugin
-- ~~  - roboticsgroup_upatras_gazebo_plugins
-- ~~  - rotors_gazebo_plugins
-- ~~  - kortex_driver
-- ~~  - gen3_lite_gen3_lite_2f_move_it_config
-- ~~  - gen3_move_it_config
-- ~~  - gen3_robotiq_2f_140_move_it_config
-- ~~  - gen3_robotiq_2f_85_move_it_config
-- ~~  - kortex_examples
-- ~~  - kortex_gazebo_camera
-- ~~  - kortex_gazebo_depth
-- ~~  - rotors_gazebo
```

1-167

```
-- ~~  - mw_vision_example
-- ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
-- +++ processing catkin metapackage: 'husky_desktop'
-- ==> add_subdirectory(husky/husky_desktop)
-- +++ processing catkin metapackage: 'husky_simulator'
-- ==> add_subdirectory(husky/husky_simulator)
-- +++ processing catkin metapackage: 'kortex_move_it_config'
-- ==> add_subdirectory(ros_kortex/kortex_move_it_config/kortex_move_it_config)
-- +++ processing catkin metapackage: 'mav_comm'
-- ==> add_subdirectory(mav_comm/mav_comm)
-- +++ processing catkin package: 'rotors_description'
-- ==> add_subdirectory(rotors_simulator/rotors_description)

-- +++ processing catkin metapackage: 'rotors_simulator'
-- ==> add_subdirectory(rotors_simulator/rotors_simulator)
-- +++ processing catkin package: 'husky_msgs'
-- ==> add_subdirectory(husky/husky_msgs)
-- Using these message generators: gencpp;geneus;genlisp;gennodejs;genpy
-- husky_msgs: 1 messages, 0 services
-- +++ processing catkin package: 'mav_state_machine_msgs'
-- ==> add_subdirectory(mav_comm/mav_state_machine_msgs)
-- Using these message generators: gencpp;geneus;genlisp;gennodejs;genpy
-- mav_state_machine_msgs: 1 messages, 1 services

-- +++ processing catkin package: 'mav_system_msgs'
-- ==> add_subdirectory(mav_comm/mav_system_msgs)
-- Using these message generators: gencpp;geneus;genlisp;gennodejs;genpy
-- mav_system_msgs: 2 messages, 0 services
-- +++ processing catkin package: 'rotors_comm'
-- ==> add_subdirectory(rotors_simulator/rotors_comm)
-- Using these message generators: gencpp;geneus;genlisp;gennodejs;genpy
-- rotors_comm: 1 messages, 2 services

-- +++ processing catkin package: 'mav_msgs'
-- ==> add_subdirectory(mav_comm/mav_msgs)
-- Using these message generators: gencpp;geneus;genlisp;gennodejs;genpy
-- mav_msgs: 8 messages, 0 services
-- +++ processing catkin package: 'robotrosfeedbackcontrolcodegen'
-- ==> add_subdirectory(robotrosfeedbackcontrolcodegen)
-- Using these message generators: gencpp;geneus;genlisp;gennodejs;genpy

-- +++ processing catkin package: 'husky_control'
-- ==> add_subdirectory(husky/husky_control)
-- +++ processing catkin package: 'husky_description'
-- ==> add_subdirectory(husky/husky_description)
-- +++ processing catkin package: 'husky_gazebo'
-- ==> add_subdirectory(husky/husky_gazebo)
-- +++ processing catkin package: 'husky_navigation'
-- ==> add_subdirectory(husky/husky_navigation)
-- +++ processing catkin package: 'husky_viz'
-- ==> add_subdirectory(husky/husky_viz)

-- +++ processing catkin package: 'kortex_control'
-- ==> add_subdirectory(ros_kortex/kortex_control)
-- +++ processing catkin package: 'kortex_description'
-- ==> add_subdirectory(ros_kortex/kortex_description)
-- +++ processing catkin package: 'kortex_gazebo'
-- ==> add_subdirectory(ros_kortex/kortex_gazebo)
```

```
-- Found DART: /usr/include (Required is at least version "6.6") found components: dart
-- Found Protobuf: /usr/lib/x86_64-linux-gnu/libprotobuf.so;-lpthread (found version "3.6.1")
-- Looking for ignition-math6 -- found version 6.8.0
-- Searching for dependencies of ignition-math6
-- Looking for OGRE...
-- Found Ogre Ghadamon (1.9.0)
-- Found OGRE: optimized;/usr/lib/x86_64-linux-gnu/libOgreMain.so;debug;/usr/lib/x86_64-linux-gnu
-- Looking for OGRE_Paging...
-- Found OGRE_Paging: optimized;/usr/lib/x86_64-linux-gnu/libOgrePaging.so;debug;/usr/lib/x86_64-
-- Looking for OGRE_Terrain...
-- Found OGRE_Terrain: optimized;/usr/lib/x86_64-linux-gnu/libOgreTerrain.so;debug;/usr/lib/x86_6
-- Looking for OGRE_Property...
-- Found OGRE_Property: optimized;/usr/lib/x86_64-linux-gnu/libOgreProperty.so;debug;/usr/lib/x86
-- Looking for OGRE_RTShaderSystem...
-- Found OGRE_RTShaderSystem: optimized;/usr/lib/x86_64-linux-gnu/libOgreRTShaderSystem.so;debug
-- Looking for OGRE_Volume...
-- Found OGRE_Volume: optimized;/usr/lib/x86_64-linux-gnu/libOgreVolume.so;debug;/usr/lib/x86_64-
-- Looking for OGRE_Overlay...
-- Found OGRE_Overlay: optimized;/usr/lib/x86_64-linux-gnu/libOgreOverlay.so;debug;/usr/lib/x86_6
-- Looking for ignition-math6 -- found version 6.8.0
-- Looking for ignition-transport8 -- found version 8.2.0
-- Searching for dependencies of ignition-transport8

-- Found Protobuf: /usr/lib/x86_64-linux-gnu/libprotobuf.so;-lpthread (found suitable version "3
-- Config-file not installed for ZeroMQ -- checking for pkg-config
-- Checking for module 'libzmq >= 4'
--   Found libzmq , version 4.3.2
-- Checking for module 'uuid'
--   Found uuid, version 2.34.0
-- Looking for ignition-msgs5 -- found version 5.7.0
-- Searching for dependencies of ignition-msgs5
-- Looking for ignition-math6 -- found version 6.8.0
-- Checking for module 'tinyxml2'
--   Found tinyxml2, version 6.2.0
-- Looking for ignition-msgs5 -- found version 5.7.0
-- Looking for ignition-common3 -- found version 3.13.1
-- Searching for dependencies of ignition-common3
-- Looking for dlfcn.h - found
-- Looking for libdl - found
-- Searching for <ignition-common3> component [graphics]
-- Looking for ignition-common3-graphics -- found version 3.13.1
-- Searching for dependencies of ignition-common3-graphics
-- Looking for ignition-math6 -- found version 6.8.0
-- Looking for ignition-fuel_tools4 -- found version 4.3.0
-- Searching for dependencies of ignition-fuel_tools4
-- Checking for module 'jsoncpp'
--   Found jsoncpp, version 1.7.4
-- Checking for module 'yaml-0.1'
--   Found yaml-0.1, version 0.2.2
-- Checking for module 'libzip'
--   Found libzip, version 1.5.1
-- Looking for ignition-common3 -- found version 3.13.1
-- Looking for ignition-msgs5 -- found version 5.7.0

-- +++ processing catkin package: 'rotors_evaluation'
-- ==> add_subdirectory(rotors_simulator/rotors_evaluation)
-- Installing devel-space wrapper /home/user/catkin_ws/src/rotors_simulator/rotors_evaluation/src
-- Installing devel-space wrapper /home/user/catkin_ws/src/rotors_simulator/rotors_evaluation/src
```

```
-- Installing devel-space wrapper /home/user/catkin_ws/src/rotors_simulator/rotors_evaluation/src
-- +++ processing catkin package: 'rqt_rotors'
-- ==> add_subdirectory(rotors_simulator/rqt_rotors)
-- mavros_msgs not found, skipping rqt_rotors package.

-- +++ processing catkin package: 'mav_planning_msgs'
-- ==> add_subdirectory(mav_comm/mav_planning_msgs)
-- Using these message generators: gencpp;geneus;genlisp;gennodejs;genpy
-- mav_planning_msgs: 9 messages, 3 services
-- +++ processing catkin package: 'opencv_node'
-- ==> add_subdirectory(opencv_node)

-- +++ processing catkin package: 'rotors_control'
-- ==> add_subdirectory(rotors_simulator/rotors_control)
-- Using these message generators: gencpp;geneus;genlisp;gennodejs;genpy
CMake Warning at /opt/ros/noetic/share/catkin/cmake/catkin_package.cmake:166 (message):
  catkin_package() DEPENDS on 'Eigen3' but neither 'Eigen3_INCLUDE_DIRS' nor
  'Eigen3_LIBRARIES' is defined.
Call Stack (most recent call first):
  /opt/ros/noetic/share/catkin/cmake/catkin_package.cmake:102 (_catkin_package)
  rotors_simulator/rotors_control/CMakeLists.txt:17 (catkin_package)


-- +++ processing catkin package: 'rotors_hil_interface'
-- ==> add_subdirectory(rotors_simulator/rotors_hil_interface)
-- Mavlink not found. Skipping HIL_INTERFACE package.
-- +++ processing catkin package: 'rotors_joy_interface'
-- ==> add_subdirectory(rotors_simulator/rotors_joy_interface)

-- +++ processing catkin package: 'gazebo_version_helpers'
-- ==> add_subdirectory(ros_kortex/third_party/gazebo-pkgs/gazebo_version_helpers)
-- Using these message generators: gencpp;geneus;genlisp;gennodejs;genpy
-- Found DART: /usr/include (Required is at least version "6.6") found components: dart
-- Found Protobuf: /usr/lib/x86_64-linux-gnu/libprotobuf.so;-lpthread (found version "3.6.1")
-- Looking for ignition-math6 -- found version 6.8.0
-- Searching for dependencies of ignition-math6
-- Looking for OGRE...
-- Found Ogre Ghadamon (1.9.0)
-- Found OGRE: optimized;/usr/lib/x86_64-linux-gnu/libOgreMain.so;debug;/usr/lib/x86_64-linux-gnu
-- Looking for OGRE_Paging...
-- Found OGRE_Paging: optimized;/usr/lib/x86_64-linux-gnu/libOgrePaging.so;debug;/usr/lib/x86_64-
-- Looking for OGRE_Terrain...
-- Found OGRE_Terrain: optimized;/usr/lib/x86_64-linux-gnu/libOgreTerrain.so;debug;/usr/lib/x86_6
-- Looking for OGRE_Property...
-- Found OGRE_Property: optimized;/usr/lib/x86_64-linux-gnu/libOgreProperty.so;debug;/usr/lib/x86
-- Looking for OGRE_RTShaderSystem...
-- Found OGRE_RTShaderSystem: optimized;/usr/lib/x86_64-linux-gnu/libOgreRTShaderSystem.so;debug
-- Looking for OGRE_Volume...
-- Found OGRE_Volume: optimized;/usr/lib/x86_64-linux-gnu/libOgreVolume.so;debug;/usr/lib/x86_64-
-- Looking for OGRE_Overlay...
-- Found OGRE_Overlay: optimized;/usr/lib/x86_64-linux-gnu/libOgreOverlay.so;debug;/usr/lib/x86_6
-- Looking for ignition-math6 -- found version 6.8.0
-- Looking for ignition-transport8 -- found version 8.2.0
-- Searching for dependencies of ignition-transport8
-- Found Protobuf: /usr/lib/x86_64-linux-gnu/libprotobuf.so;-lpthread (found suitable version "3

-- Config-file not installed for ZeroMQ -- checking for pkg-config
-- Checking for module 'libzmq >= 4'
```

```
--    Found libzmq , version 4.3.2
-- Checking for module 'uuid'
--    Found uuid, version 2.34.0
-- Looking for ignition-msgs5 -- found version 5.7.0
-- Searching for dependencies of ignition-msgs5
-- Looking for ignition-math6 -- found version 6.8.0
-- Checking for module 'tinyxml2'
--    Found tinyxml2, version 6.2.0
-- Looking for ignition-msgs5 -- found version 5.7.0
-- Looking for ignition-common3 -- found version 3.13.1
-- Searching for dependencies of ignition-common3
-- Looking for dlfcn.h - found
-- Looking for libdl - found
-- Searching for <ignition-common3> component [graphics]
-- Looking for ignition-common3-graphics -- found version 3.13.1
-- Searching for dependencies of ignition-common3-graphics
-- Looking for ignition-math6 -- found version 6.8.0
-- Looking for ignition-fuel_tools4 -- found version 4.3.0
-- Searching for dependencies of ignition-fuel_tools4
-- Checking for module 'jsoncpp'
--    Found jsoncpp, version 1.7.4
-- Checking for module 'yaml-0.1'
--    Found yaml-0.1, version 0.2.2
-- Checking for module 'libzip'
--    Found libzip, version 1.5.1
-- Looking for ignition-common3 -- found version 3.13.1
-- Looking for ignition-msgs5 -- found version 5.7.0
CMake Warning at /opt/ros/noetic/share/catkin/cmake/catkin_package.cmake:166 (message):
  catkin_package() DEPENDS on 'gazebo' but neither 'gazebo_INCLUDE_DIRS' nor
  'gazebo_LIBRARIES' is defined.
Call Stack (most recent call first):
  /opt/ros/noetic/share/catkin/cmake/catkin_package.cmake:102 (_catkin_package)
  ros_kortex/third_party/gazebo-pkgs/gazebo_version_helpers/CMakeLists.txt:26 (catkin_package)


-- +++ processing catkin package: 'gazebo_grasp_plugin'
-- ==> add_subdirectory(ros_kortex/third_party/gazebo-pkgs/gazebo_grasp_plugin)
-- Using these message generators: gencpp;geneus;genlisp;gennodejs;genpy
-- Found DART: /usr/include (Required is at least version "6.6") found components: dart
-- Found Protobuf: /usr/lib/x86_64-linux-gnu/libprotobuf.so;-lpthread (found version "3.6.1")
-- Looking for ignition-math6 -- found version 6.8.0
-- Searching for dependencies of ignition-math6
-- Looking for OGRE...
-- Found Ogre Ghadamon (1.9.0)
-- Found OGRE: optimized;/usr/lib/x86_64-linux-gnu/libOgreMain.so;debug;/usr/lib/x86_64-linux-gn
-- Looking for OGRE_Paging...
-- Found OGRE_Paging: optimized;/usr/lib/x86_64-linux-gnu/libOgrePaging.so;debug;/usr/lib/x86_64-
-- Looking for OGRE_Terrain...
-- Found OGRE_Terrain: optimized;/usr/lib/x86_64-linux-gnu/libOgreTerrain.so;debug;/usr/lib/x86_6
-- Looking for OGRE_Property...
-- Found OGRE_Property: optimized;/usr/lib/x86_64-linux-gnu/libOgreProperty.so;debug;/usr/lib/x86
-- Looking for OGRE_RTShaderSystem...
-- Found OGRE_RTShaderSystem: optimized;/usr/lib/x86_64-linux-gnu/libOgreRTShaderSystem.so;debug
-- Looking for OGRE_Volume...
-- Found OGRE_Volume: optimized;/usr/lib/x86_64-linux-gnu/libOgreVolume.so;debug;/usr/lib/x86_64-
-- Looking for OGRE_Overlay...
-- Found OGRE_Overlay: optimized;/usr/lib/x86_64-linux-gnu/libOgreOverlay.so;debug;/usr/lib/x86_6
```

1-171

```
-- Looking for ignition-math6 -- found version 6.8.0
-- Looking for ignition-transport8 -- found version 8.2.0
-- Searching for dependencies of ignition-transport8
-- Found Protobuf: /usr/lib/x86_64-linux-gnu/libprotobuf.so;-lpthread (found suitable version "3
-- Config-file not installed for ZeroMQ -- checking for pkg-config
-- Checking for module 'libzmq >= 4'
--   Found libzmq , version 4.3.2
-- Checking for module 'uuid'
--   Found uuid, version 2.34.0

-- Looking for ignition-msgs5 -- found version 5.7.0
-- Searching for dependencies of ignition-msgs5
-- Looking for ignition-math6 -- found version 6.8.0
-- Checking for module 'tinyxml2'
--   Found tinyxml2, version 6.2.0
-- Looking for ignition-msgs5 -- found version 5.7.0
-- Looking for ignition-common3 -- found version 3.13.1
-- Searching for dependencies of ignition-common3
-- Looking for dlfcn.h - found
-- Looking for libdl - found
-- Searching for <ignition-common3> component [graphics]
-- Looking for ignition-common3-graphics -- found version 3.13.1
-- Searching for dependencies of ignition-common3-graphics
-- Looking for ignition-math6 -- found version 6.8.0
-- Looking for ignition-fuel_tools4 -- found version 4.3.0
-- Searching for dependencies of ignition-fuel_tools4
-- Checking for module 'jsoncpp'
--   Found jsoncpp, version 1.7.4
-- Checking for module 'yaml-0.1'
--   Found yaml-0.1, version 0.2.2
-- Checking for module 'libzip'
--   Found libzip, version 1.5.1
-- Looking for ignition-common3 -- found version 3.13.1
-- Looking for ignition-msgs5 -- found version 5.7.0
CMake Warning at /opt/ros/noetic/share/catkin/cmake/catkin_package.cmake:166 (message):
  catkin_package() DEPENDS on 'gazebo' but neither 'gazebo_INCLUDE_DIRS' nor
  'gazebo_LIBRARIES' is defined.
Call Stack (most recent call first):
  /opt/ros/noetic/share/catkin/cmake/catkin_package.cmake:102 (_catkin_package)
  ros_kortex/third_party/gazebo-pkgs/gazebo_grasp_plugin/CMakeLists.txt:31 (catkin_package)


-- +++ processing catkin package: 'roboticsgroup_upatras_gazebo_plugins'
-- ==> add_subdirectory(ros_kortex/third_party/roboticsgroup_upatras_gazebo_plugins)
-- Using these message generators: gencpp;geneus;genlisp;gennodejs;genpy

-- Found DART: /usr/include (Required is at least version "6.6") found components: dart
-- Found Protobuf: /usr/lib/x86_64-linux-gnu/libprotobuf.so;-lpthread (found version "3.6.1")
-- Looking for ignition-math6 -- found version 6.8.0
-- Searching for dependencies of ignition-math6
-- Looking for OGRE...
-- Found Ogre Ghadamon (1.9.0)
-- Found OGRE: optimized;/usr/lib/x86_64-linux-gnu/libOgreMain.so;debug;/usr/lib/x86_64-linux-gnu
-- Looking for OGRE_Paging...
-- Found OGRE_Paging: optimized;/usr/lib/x86_64-linux-gnu/libOgrePaging.so;debug;/usr/lib/x86_64-
-- Looking for OGRE_Terrain...
-- Found OGRE_Terrain: optimized;/usr/lib/x86_64-linux-gnu/libOgreTerrain.so;debug;/usr/lib/x86_6
-- Looking for OGRE_Property...
```

```
-- Found OGRE_Property: optimized;/usr/lib/x86_64-linux-gnu/libOgreProperty.so;debug;/usr/lib/x86
-- Looking for OGRE_RTShaderSystem...
-- Found OGRE_RTShaderSystem: optimized;/usr/lib/x86_64-linux-gnu/libOgreRTShaderSystem.so;debug
-- Looking for OGRE_Volume...
-- Found OGRE_Volume: optimized;/usr/lib/x86_64-linux-gnu/libOgreVolume.so;debug;/usr/lib/x86_64-
-- Looking for OGRE_Overlay...
-- Found OGRE_Overlay: optimized;/usr/lib/x86_64-linux-gnu/libOgreOverlay.so;debug;/usr/lib/x86_6
-- Looking for ignition-math6 -- found version 6.8.0
-- Looking for ignition-transport8 -- found version 8.2.0
-- Searching for dependencies of ignition-transport8
-- Found Protobuf: /usr/lib/x86_64-linux-gnu/libprotobuf.so;-lpthread (found suitable version "3
-- Config-file not installed for ZeroMQ -- checking for pkg-config
-- Checking for module 'libzmq >= 4'
--   Found libzmq , version 4.3.2
-- Checking for module 'uuid'
--   Found uuid, version 2.34.0
-- Looking for ignition-msgs5 -- found version 5.7.0
-- Searching for dependencies of ignition-msgs5
-- Looking for ignition-math6 -- found version 6.8.0
-- Checking for module 'tinyxml2'
--   Found tinyxml2, version 6.2.0
-- Looking for ignition-msgs5 -- found version 5.7.0
-- Looking for ignition-common3 -- found version 3.13.1
-- Searching for dependencies of ignition-common3
-- Looking for dlfcn.h - found
-- Looking for libdl - found
-- Searching for <ignition-common3> component [graphics]
-- Looking for ignition-common3-graphics -- found version 3.13.1
-- Searching for dependencies of ignition-common3-graphics
-- Looking for ignition-math6 -- found version 6.8.0
-- Looking for ignition-fuel_tools4 -- found version 4.3.0
-- Searching for dependencies of ignition-fuel_tools4
-- Checking for module 'jsoncpp'

--   Found jsoncpp, version 1.7.4
-- Checking for module 'yaml-0.1'
--   Found yaml-0.1, version 0.2.2
-- Checking for module 'libzip'
--   Found libzip, version 1.5.1
-- Looking for ignition-common3 -- found version 3.13.1
-- Looking for ignition-msgs5 -- found version 5.7.0
-- +++ processing catkin package: 'rotors_gazebo_plugins'
-- ==> add_subdirectory(rotors_simulator/rotors_gazebo_plugins)
-- BUILD_OCTOMAP_PLUGIN variable not provided, setting to FALSE.
-- BUILD_OPTICAL_FLOW_PLUGIN variable not provided, setting to FALSE.
-- NO_ROS variable not provided, setting to FALSE.
-- CMAKE_BINARY_DIR = /home/user/catkin_ws/build

-- Could NOT find MAVLink (missing: MAVLINK_INCLUDE_DIRS) (found version "2.0")
--  mavlink not found, not building MAVLINK_INTERFACE_PLUGIN.
-- ADDITIONAL_INCLUDE_DIRS =
-- BUILD_OCTOMAP_PLUGIN = FALSE, NOT building gazebo_octomap_plugin.
-- BUILD_OPTICAL_FLOW_PLUGIN = FALSE, NOT building gazebo_optical_flow_plugin.
-- NO_ROS = FALSE, building rotors_gazebo_plugins WITH ROS dependancies.
-- Using these message generators: gencpp;geneus;genlisp;gennodejs;genpy
-- CMAKE_BINARY_DIR = /home/user/catkin_ws/build
-- Eigen found (include: /usr/include/eigen3)
-- Found DART: /usr/include (Required is at least version "6.6") found components: dart
```

```
-- Found Protobuf: /usr/lib/x86_64-linux-gnu/libprotobuf.so;-lpthread (found version "3.6.1")
-- Looking for ignition-math6 -- found version 6.8.0

-- Searching for dependencies of ignition-math6
-- Looking for OGRE...
-- Found Ogre Ghadamon (1.9.0)
-- Found OGRE: optimized;/usr/lib/x86_64-linux-gnu/libOgreMain.so;debug;/usr/lib/x86_64-linux-gnu
-- Looking for OGRE_Paging...
-- Found OGRE_Paging: optimized;/usr/lib/x86_64-linux-gnu/libOgrePaging.so;debug;/usr/lib/x86_64-
-- Looking for OGRE_Terrain...
-- Found OGRE_Terrain: optimized;/usr/lib/x86_64-linux-gnu/libOgreTerrain.so;debug;/usr/lib/x86_6
-- Looking for OGRE_Property...
-- Found OGRE_Property: optimized;/usr/lib/x86_64-linux-gnu/libOgreProperty.so;debug;/usr/lib/x86
-- Looking for OGRE_RTShaderSystem...
-- Found OGRE_RTShaderSystem: optimized;/usr/lib/x86_64-linux-gnu/libOgreRTShaderSystem.so;debug
-- Looking for OGRE_Volume...
-- Found OGRE_Volume: optimized;/usr/lib/x86_64-linux-gnu/libOgreVolume.so;debug;/usr/lib/x86_64-
-- Looking for OGRE_Overlay...
-- Found OGRE_Overlay: optimized;/usr/lib/x86_64-linux-gnu/libOgreOverlay.so;debug;/usr/lib/x86_6
-- Looking for ignition-math6 -- found version 6.8.0
-- Looking for ignition-transport8 -- found version 8.2.0
-- Searching for dependencies of ignition-transport8
-- Found Protobuf: /usr/lib/x86_64-linux-gnu/libprotobuf.so;-lpthread (found suitable version "3
-- Config-file not installed for ZeroMQ -- checking for pkg-config
-- Checking for module 'libzmq >= 4'
--   Found libzmq , version 4.3.2
-- Checking for module 'uuid'
--   Found uuid, version 2.34.0
-- Looking for ignition-msgs5 -- found version 5.7.0
-- Searching for dependencies of ignition-msgs5
-- Looking for ignition-math6 -- found version 6.8.0
-- Checking for module 'tinyxml2'
--   Found tinyxml2, version 6.2.0
-- Looking for ignition-msgs5 -- found version 5.7.0
-- Looking for ignition-common3 -- found version 3.13.1
-- Searching for dependencies of ignition-common3
-- Looking for dlfcn.h - found
-- Looking for libdl - found
-- Searching for <ignition-common3> component [graphics]
-- Looking for ignition-common3-graphics -- found version 3.13.1
-- Searching for dependencies of ignition-common3-graphics
-- Looking for ignition-math6 -- found version 6.8.0
-- Looking for ignition-fuel_tools4 -- found version 4.3.0
-- Searching for dependencies of ignition-fuel_tools4
-- Checking for module 'jsoncpp'
--   Found jsoncpp, version 1.7.4
-- Checking for module 'yaml-0.1'
--   Found yaml-0.1, version 0.2.2

-- Checking for module 'libzip'
--   Found libzip, version 1.5.1
-- Looking for ignition-common3 -- found version 3.13.1
-- Looking for ignition-msgs5 -- found version 5.7.0
-- Gazebo version: 11.5
-- No yaml_cpp_catkin, using yaml-cpp system library instead.
-- PROTOBUF_IMPORT_DIRS = /usr/include/gazebo-11/gazebo/msgs/proto
-- Found Protobuf: /usr/lib/x86_64-linux-gnu/libprotobuf.so;-lpthread (found version "3.6.1")
CMake Warning at /opt/ros/noetic/share/catkin/cmake/catkin_package.cmake:166 (message):
```

```
  catkin_package() DEPENDS on 'octomap' but neither 'octomap_INCLUDE_DIRS'
  nor 'octomap_LIBRARIES' is defined.
Call Stack (most recent call first):
  /opt/ros/noetic/share/catkin/cmake/catkin_package.cmake:102 (_catkin_package)
  rotors_simulator/rotors_gazebo_plugins/CMakeLists.txt:229 (catkin_package)


-- GAZEBO 11.5.1
--
-- BUILD_MAVLINK_INTERFACE_PLUGIN = FALSE, NOT adding mavros dependency and NOT building mavlink
-- CMAKE_INSTALL_PREFIX = /home/user/catkin_ws/install
-- +++ processing catkin package: 'kortex_driver'
-- ==> add_subdirectory(ros_kortex/kortex_driver)
CONAN_TARGET_PLATFORM is x86
-- Using these message generators: gencpp;geneus;genlisp;gennodejs;genpy

-- Found Boost: /usr/lib/x86_64-linux-gnu/cmake/Boost-1.71.0/BoostConfig.cmake (found version "1
-- Generating .msg files for action kortex_driver/FollowCartesianTrajectory /home/user/catkin_ws,

-- kortex_driver: 432 messages, 259 services

-- Conan: checking conan executable in path
-- Conan: Found program /usr/local/bin/conan

-- Conan: Version found Conan version 1.39.0

-- Conan: Adding kinova_public remote repository (https://artifactory.kinovaapps.com/artifactory,

-- Conan: Automatic detection of conan settings from cmake
-- Conan: Settings= -s;build_type=Release;-s;compiler=gcc;-s;compiler.version=9;-s;compiler.libcx
-- Conan executing: conan install /home/user/catkin_ws/src/ros_kortex/kortex_driver/conanfile.py
Configuration:
[settings]
arch=x86_64
arch_build=x86_64
build_type=Release
compiler=gcc
compiler.libcxx=libstdc++11
compiler.version=9
os=Linux
os_build=Linux
kortex_api_cpp:compiler=gcc
kortex_api_cpp:compiler.version=5
kortex_api_cpp:compiler.libcxx=libstdc++11
[options]
[build_requires]
[env]


conanfile.py: Installing package
Requirements
    kortex_api_cpp/2.3.0-r.34@kortex/stable from 'kinova_public' - Cache
Packages
    kortex_api_cpp/2.3.0-r.34@kortex/stable:c023db9fc677d4d0b3bd0c20f71385e4cf8a1220 - Cache

Installing (downloading, building) binaries...
kortex_api_cpp/2.3.0-r.34@kortex/stable: Already installed!
conanfile.py: Generator cmake created conanbuildinfo.cmake
```

1-175

```
conanfile.py: Generator txt created conanbuildinfo.txt
conanfile.py: Aggregating env generators
conanfile.py: Generated conaninfo.txt
conanfile.py: Generated graphinfo
-- Conan: Loading conanbuildinfo.cmake
-- Conan: Using cmake targets configuration
-- Library KortexApiCpp found /home/user/.conan/data/kortex_api_cpp/2.3.0-r.34/kortex/stable/pacl
-- Conan: Adjusting default RPATHs Conan policies
-- Conan: Adjusting language standard
-- Current conanbuildinfo.cmake directory: /home/user/catkin_ws/build/ros_kortex/kortex_driver
-- WARN: CONAN_COMPILER variable not set, please make sure yourself that your compiler and versic
-- +++ processing catkin package: 'gen3_lite_gen3_lite_2f_move_it_config'
-- ==> add_subdirectory(ros_kortex/kortex_move_it_config/gen3_lite_gen3_lite_2f_move_it_config)

-- +++ processing catkin package: 'gen3_move_it_config'
-- ==> add_subdirectory(ros_kortex/kortex_move_it_config/gen3_move_it_config)
-- +++ processing catkin package: 'gen3_robotiq_2f_140_move_it_config'
-- ==> add_subdirectory(ros_kortex/kortex_move_it_config/gen3_robotiq_2f_140_move_it_config)
-- +++ processing catkin package: 'gen3_robotiq_2f_85_move_it_config'
-- ==> add_subdirectory(ros_kortex/kortex_move_it_config/gen3_robotiq_2f_85_move_it_config)
-- +++ processing catkin package: 'kortex_examples'
-- ==> add_subdirectory(ros_kortex/kortex_examples)
-- Using these message generators: gencpp;geneus;genlisp;gennodejs;genpy

-- Installing devel-space wrapper /home/user/catkin_ws/src/ros_kortex/kortex_examples/src/actuatc
-- Installing devel-space wrapper /home/user/catkin_ws/src/ros_kortex/kortex_examples/src/full_ar
-- Installing devel-space wrapper /home/user/catkin_ws/src/ros_kortex/kortex_examples/src/full_ar
-- Installing devel-space wrapper /home/user/catkin_ws/src/ros_kortex/kortex_examples/src/full_ar
-- Installing devel-space wrapper /home/user/catkin_ws/src/ros_kortex/kortex_examples/src/move_it
-- Installing devel-space wrapper /home/user/catkin_ws/src/ros_kortex/kortex_examples/src/vision_
-- +++ processing catkin package: 'kortex_gazebo_camera'
-- ==> add_subdirectory(kortex_gazebo_camera)
-- +++ processing catkin package: 'kortex_gazebo_depth'
-- ==> add_subdirectory(kortex_gazebo_depth)
-- +++ processing catkin package: 'rotors_gazebo'
-- ==> add_subdirectory(rotors_simulator/rotors_gazebo)
-- Found DART: /usr/include (Required is at least version "6.6") found components: dart
-- Found Boost: /usr/lib/x86_64-linux-gnu/cmake/Boost-1.71.0/BoostConfig.cmake (found suitable ve
-- Looking for ignition-math6 -- found version 6.8.0
-- Searching for dependencies of ignition-math6
-- Looking for OGRE...
-- Found Ogre Ghadamon (1.9.0)
-- Found OGRE: optimized;/usr/lib/x86_64-linux-gnu/libOgreMain.so;debug;/usr/lib/x86_64-linux-gnu
-- Looking for OGRE_Paging...
-- Found OGRE_Paging: optimized;/usr/lib/x86_64-linux-gnu/libOgrePaging.so;debug;/usr/lib/x86_64-
-- Looking for OGRE_Terrain...
-- Found OGRE_Terrain: optimized;/usr/lib/x86_64-linux-gnu/libOgreTerrain.so;debug;/usr/lib/x86_€
-- Looking for OGRE_Property...
-- Found OGRE_Property: optimized;/usr/lib/x86_64-linux-gnu/libOgreProperty.so;debug;/usr/lib/x8€
-- Looking for OGRE_RTShaderSystem...
-- Found OGRE_RTShaderSystem: optimized;/usr/lib/x86_64-linux-gnu/libOgreRTShaderSystem.so;debug
-- Looking for OGRE_Volume...
-- Found OGRE_Volume: optimized;/usr/lib/x86_64-linux-gnu/libOgreVolume.so;debug;/usr/lib/x86_64-
-- Looking for OGRE_Overlay...
-- Found OGRE_Overlay: optimized;/usr/lib/x86_64-linux-gnu/libOgreOverlay.so;debug;/usr/lib/x86_€
-- Looking for ignition-math6 -- found version 6.8.0
-- Looking for ignition-transport8 -- found version 8.2.0
-- Searching for dependencies of ignition-transport8
```

```
-- Found Protobuf: /usr/lib/x86_64-linux-gnu/libprotobuf.so;-lpthread (found suitable version "3
-- Config-file not installed for ZeroMQ -- checking for pkg-config
-- Checking for module 'libzmq >= 4'
--   Found libzmq , version 4.3.2

-- Checking for module 'uuid'
--   Found uuid, version 2.34.0
-- Looking for ignition-msgs5 -- found version 5.7.0
-- Searching for dependencies of ignition-msgs5
-- Looking for ignition-math6 -- found version 6.8.0
-- Checking for module 'tinyxml2'
--   Found tinyxml2, version 6.2.0
-- Looking for ignition-msgs5 -- found version 5.7.0
-- Looking for ignition-common3 -- found version 3.13.1
-- Searching for dependencies of ignition-common3
-- Looking for dlfcn.h - found
-- Looking for libdl - found
-- Searching for <ignition-common3> component [graphics]
-- Looking for ignition-common3-graphics -- found version 3.13.1
-- Searching for dependencies of ignition-common3-graphics
-- Looking for ignition-math6 -- found version 6.8.0
-- Looking for ignition-fuel_tools4 -- found version 4.3.0
-- Searching for dependencies of ignition-fuel_tools4
-- Checking for module 'jsoncpp'
--   Found jsoncpp, version 1.7.4
-- Checking for module 'yaml-0.1'
--   Found yaml-0.1, version 0.2.2
-- Checking for module 'libzip'
--   Found libzip, version 1.5.1
-- Looking for ignition-common3 -- found version 3.13.1
-- Looking for ignition-msgs5 -- found version 5.7.0
-- Building iris.sdf.

-- +++ processing catkin package: 'mw_vision_example'
-- ==> add_subdirectory(mw_vision_example)
-- Using these message generators: gencpp;geneus;genlisp;gennodejs;genpy
-- Configuring done

-- Generating done

-- Build files have been written to: /home/user/catkin_ws/build
cd /home/user/catkin_ws/build && /usr/bin/cmake -S/home/user/catkin_ws/src -B/home/user/catkin_ws
cd /home/user/catkin_ws/build && /usr/bin/cmake -E cmake_progress_start /home/user/catkin_ws/buil
cd /home/user/catkin_ws/build && make -f CMakeFiles/Makefile2 robotrosfeedbackcontrolcodegen/all
make[1]: Entering directory '/home/user/catkin_ws/build'
make[2]: Entering directory '/home/user/catkin_ws/build'
Scanning dependencies of target nav_msgs_generate_messages_lisp
make[2]: Leaving directory '/home/user/catkin_ws/build'
[  0%] Built target nav_msgs_generate_messages_lisp
make[2]: Entering directory '/home/user/catkin_ws/build'
make[2]: Leaving directory '/home/user/catkin_ws/build'
[  0%] Built target std_msgs_generate_messages_cpp
make[2]: Entering directory '/home/user/catkin_ws/build'
make[2]: Leaving directory '/home/user/catkin_ws/build'
[  0%] Built target std_msgs_generate_messages_eus
make[2]: Entering directory '/home/user/catkin_ws/build'
make[2]: Leaving directory '/home/user/catkin_ws/build'
[  0%] Built target std_msgs_generate_messages_lisp
```

```
make[2]: Entering directory '/home/user/catkin_ws/build'
make[2]: Leaving directory '/home/user/catkin_ws/build'
[  0%] Built target std_msgs_generate_messages_nodejs
make[2]: Entering directory '/home/user/catkin_ws/build'
make[2]: Leaving directory '/home/user/catkin_ws/build'
[  0%] Built target std_msgs_generate_messages_py
make[2]: Entering directory '/home/user/catkin_ws/build'
make[2]: Leaving directory '/home/user/catkin_ws/build'
[  0%] Built target geometry_msgs_generate_messages_nodejs
make[2]: Entering directory '/home/user/catkin_ws/build'
make[2]: Leaving directory '/home/user/catkin_ws/build'
[  0%] Built target geometry_msgs_generate_messages_py
make[2]: Entering directory '/home/user/catkin_ws/build'
make[2]: Leaving directory '/home/user/catkin_ws/build'
[  0%] Built target geometry_msgs_generate_messages_lisp
make[2]: Entering directory '/home/user/catkin_ws/build'
make[2]: Leaving directory '/home/user/catkin_ws/build'
[  0%] Built target geometry_msgs_generate_messages_eus
make[2]: Entering directory '/home/user/catkin_ws/build'
make[2]: Leaving directory '/home/user/catkin_ws/build'
[  0%] Built target geometry_msgs_generate_messages_cpp
make[2]: Entering directory '/home/user/catkin_ws/build'
Scanning dependencies of target actionlib_msgs_generate_messages_py
make[2]: Leaving directory '/home/user/catkin_ws/build'
[  0%] Built target actionlib_msgs_generate_messages_py
make[2]: Entering directory '/home/user/catkin_ws/build'
Scanning dependencies of target actionlib_msgs_generate_messages_lisp
make[2]: Leaving directory '/home/user/catkin_ws/build'
[  0%] Built target actionlib_msgs_generate_messages_lisp
make[2]: Entering directory '/home/user/catkin_ws/build'
Scanning dependencies of target actionlib_msgs_generate_messages_cpp
make[2]: Leaving directory '/home/user/catkin_ws/build'
[  0%] Built target actionlib_msgs_generate_messages_cpp
make[2]: Entering directory '/home/user/catkin_ws/build'
Scanning dependencies of target nav_msgs_generate_messages_py
make[2]: Leaving directory '/home/user/catkin_ws/build'
[  0%] Built target nav_msgs_generate_messages_py
make[2]: Entering directory '/home/user/catkin_ws/build'
Scanning dependencies of target nav_msgs_generate_messages_nodejs
make[2]: Leaving directory '/home/user/catkin_ws/build'

[  0%] Built target nav_msgs_generate_messages_nodejs
make[2]: Entering directory '/home/user/catkin_ws/build'
Scanning dependencies of target roscpp_generate_messages_eus
make[2]: Leaving directory '/home/user/catkin_ws/build'
[  0%] Built target roscpp_generate_messages_eus
make[2]: Entering directory '/home/user/catkin_ws/build'
Scanning dependencies of target rosgraph_msgs_generate_messages_nodejs
make[2]: Leaving directory '/home/user/catkin_ws/build'
[  0%] Built target rosgraph_msgs_generate_messages_nodejs
make[2]: Entering directory '/home/user/catkin_ws/build'
Scanning dependencies of target roscpp_generate_messages_py
make[2]: Leaving directory '/home/user/catkin_ws/build'
[  0%] Built target roscpp_generate_messages_py
make[2]: Entering directory '/home/user/catkin_ws/build'
Scanning dependencies of target roscpp_generate_messages_nodejs
make[2]: Leaving directory '/home/user/catkin_ws/build'
[  0%] Built target roscpp_generate_messages_nodejs
```

```
make[2]: Entering directory '/home/user/catkin_ws/build'
Scanning dependencies of target rosgraph_msgs_generate_messages_cpp
make[2]: Leaving directory '/home/user/catkin_ws/build'
[  0%] Built target rosgraph_msgs_generate_messages_cpp
make[2]: Entering directory '/home/user/catkin_ws/build'
Scanning dependencies of target rosgraph_msgs_generate_messages_eus
make[2]: Leaving directory '/home/user/catkin_ws/build'
[  0%] Built target rosgraph_msgs_generate_messages_eus
make[2]: Entering directory '/home/user/catkin_ws/build'
Scanning dependencies of target actionlib_msgs_generate_messages_nodejs
make[2]: Leaving directory '/home/user/catkin_ws/build'
[  0%] Built target actionlib_msgs_generate_messages_nodejs
make[2]: Entering directory '/home/user/catkin_ws/build'
Scanning dependencies of target rosgraph_msgs_generate_messages_lisp
make[2]: Leaving directory '/home/user/catkin_ws/build'
[  0%] Built target rosgraph_msgs_generate_messages_lisp
make[2]: Entering directory '/home/user/catkin_ws/build'
Scanning dependencies of target actionlib_msgs_generate_messages_eus
make[2]: Leaving directory '/home/user/catkin_ws/build'
[  0%] Built target actionlib_msgs_generate_messages_eus
make[2]: Entering directory '/home/user/catkin_ws/build'
Scanning dependencies of target roscpp_generate_messages_lisp
make[2]: Leaving directory '/home/user/catkin_ws/build'
[  0%] Built target roscpp_generate_messages_lisp
make[2]: Entering directory '/home/user/catkin_ws/build'
Scanning dependencies of target rosgraph_msgs_generate_messages_py
make[2]: Leaving directory '/home/user/catkin_ws/build'
[  0%] Built target rosgraph_msgs_generate_messages_py
make[2]: Entering directory '/home/user/catkin_ws/build'
Scanning dependencies of target nav_msgs_generate_messages_eus
make[2]: Leaving directory '/home/user/catkin_ws/build'
[  0%] Built target nav_msgs_generate_messages_eus
make[2]: Entering directory '/home/user/catkin_ws/build'
Scanning dependencies of target roscpp_generate_messages_cpp
make[2]: Leaving directory '/home/user/catkin_ws/build'
[  0%] Built target roscpp_generate_messages_cpp
make[2]: Entering directory '/home/user/catkin_ws/build'
Scanning dependencies of target nav_msgs_generate_messages_cpp
make[2]: Leaving directory '/home/user/catkin_ws/build'
[  0%] Built target nav_msgs_generate_messages_cpp
make[2]: Entering directory '/home/user/catkin_ws/build'
Scanning dependencies of target robotROSFeedbackControlCodegen
make[2]: Leaving directory '/home/user/catkin_ws/build'
make[2]: Entering directory '/home/user/catkin_ws/build'
[  0%] Building CXX object robotrosfeedbackcontrolcodegen/CMakeFiles/robotROSFeedbackControlCodeg

[  0%] Building CXX object robotrosfeedbackcontrolcodegen/CMakeFiles/robotROSFeedbackControlCodeg

[  0%] Building CXX object robotrosfeedbackcontrolcodegen/CMakeFiles/robotROSFeedbackControlCodeg

[  0%] Building CXX object robotrosfeedbackcontrolcodegen/CMakeFiles/robotROSFeedbackControlCodeg
[  0%] Building CXX object robotrosfeedbackcontrolcodegen/CMakeFiles/robotROSFeedbackControlCodeg
[  0%] Building CXX object robotrosfeedbackcontrolcodegen/CMakeFiles/robotROSFeedbackControlCodeg

[  0%] Building CXX object robotrosfeedbackcontrolcodegen/CMakeFiles/robotROSFeedbackControlCodeg
[  0%] Building CXX object robotrosfeedbackcontrolcodegen/CMakeFiles/robotROSFeedbackControlCodeg

[  0%] Building CXX object robotrosfeedbackcontrolcodegen/CMakeFiles/robotROSFeedbackControlCodeg
```

```
[  0%] Building CXX object robotrosfeedbackcontrolcodegen/CMakeFiles/robotROSFeedbackControlCodeg

[  0%] Building CXX object robotrosfeedbackcontrolcodegen/CMakeFiles/robotROSFeedbackControlCodeg
[  0%] Building CXX object robotrosfeedbackcontrolcodegen/CMakeFiles/robotROSFeedbackControlCodeg
[  0%] Building CXX object robotrosfeedbackcontrolcodegen/CMakeFiles/robotROSFeedbackControlCodeg

[100%] Building CXX object robotrosfeedbackcontrolcodegen/CMakeFiles/robotROSFeedbackControlCodeg
[100%] Building CXX object robotrosfeedbackcontrolcodegen/CMakeFiles/robotROSFeedbackControlCodeg
[100%] Building CXX object robotrosfeedbackcontrolcodegen/CMakeFiles/robotROSFeedbackControlCodeg
[100%] Building CXX object robotrosfeedbackcontrolcodegen/CMakeFiles/robotROSFeedbackControlCodeg
[100%] Building CXX object robotrosfeedbackcontrolcodegen/CMakeFiles/robotROSFeedbackControlCodeg

[100%] Building CXX object robotrosfeedbackcontrolcodegen/CMakeFiles/robotROSFeedbackControlCodeg
[100%] Building CXX object robotrosfeedbackcontrolcodegen/CMakeFiles/robotROSFeedbackControlCodeg
[100%] Building CXX object robotrosfeedbackcontrolcodegen/CMakeFiles/robotROSFeedbackControlCodeg
[100%] Building CXX object robotrosfeedbackcontrolcodegen/CMakeFiles/robotROSFeedbackControlCodeg

[100%] Building CXX object robotrosfeedbackcontrolcodegen/CMakeFiles/robotROSFeedbackControlCodeg
[100%] Building CXX object robotrosfeedbackcontrolcodegen/CMakeFiles/robotROSFeedbackControlCodeg
[100%] Building CXX object robotrosfeedbackcontrolcodegen/CMakeFiles/robotROSFeedbackControlCodeg

[100%] Building CXX object robotrosfeedbackcontrolcodegen/CMakeFiles/robotROSFeedbackControlCodeg
[100%] Building CXX object robotrosfeedbackcontrolcodegen/CMakeFiles/robotROSFeedbackControlCodeg

[100%] Linking CXX executable /home/user/catkin_ws/devel/lib/robotrosfeedbackcontrolcodegen/robot
/usr/bin/c++  -O3 -DNDEBUG   CMakeFiles/robotROSFeedbackControlCodegen.dir/src/Publisher.cpp.o CM
make[2]: Leaving directory '/home/user/catkin_ws/build'
[100%] Built target robotROSFeedbackControlCodegen
make[1]: Leaving directory '/home/user/catkin_ws/build'
/usr/bin/cmake -E cmake_progress_start /home/user/catkin_ws/build/CMakeFiles 0
Base path: /home/user/catkin_ws
Source space: /home/user/catkin_ws/src
Build space: /home/user/catkin_ws/build
Devel space: /home/user/catkin_ws/devel
Install space: /home/user/catkin_ws/install
####
#### Running command: "cmake /home/user/catkin_ws/src -DCATKIN_DEVEL_PREFIX=/home/user/catkin_ws/
####
####
#### Running command: "make -j2 -l2" in "/home/user/catkin_ws/build/robotrosfeedbackcontrolcodege
####

---
Running ROS node.
---
The node will connect to the ROS master at 'http://172.18.250.139:11311' and advertise its addres
Use the 'rosdevice' object to stop the node or restart it with different settings.
Code generation successful.
```

Reset the Gazebo scene and call `plotPath` to plot the robot trajectory.

```
call(gazeboResetClient);
plotPath
```

The configuration generates and runs the ROS node on your remote device. You can opt to deploy and run the ROS node on the local machine by modifying the coder configuration object, `cfg`.

```
cfg = coder.config('exe');
cfg.Hardware = coder.hardware('Robot Operating System (ROS)');
cfg.Hardware.BuildAction = 'Build and run';
cfg.Hardware.DeployTo = 'Localhost';
codegen robotROSFeedbackControlCodegen -args {} -config cfg
plotPath
```

Verify Generated ROS Node

After the generated executable starts running, for the same destination coordinates, verify that the trajectory of the robot is similar to what you observe during MATLAB execution. You can also observe the robot moving in Gazebo on the virtual machine. You can publish a different destination coordinate while the robot is in motion. Refer to the Control a ROS-Enabled Robot with a Function on page 1-164 section, which shows how to publish a new set of destination coordinates through a virtual machine terminal.

Terminate the generated ROS node by using the `stopNode` function of the `rosDevice` object. Alternatively, you can terminate the execution by pressing **Ctrl-C** or sending a message to the `/stop` topic. Reset the Gazebo scene.

```
stopNode(gazeboVMDevice,'robotROSFeedbackControlCodegen')
call(gazeboResetClient);
```

Shut down the ROS system.

```
rosshutdown
```

Shutting down global node /matlab_global_node_12273 with NodeURI http://192.168.192.1:57204/ and

See Also

Related Examples

*   "MATLAB Programming for Code Generation" on page 1-158

Generate ROS Node for UAV Waypoint Follower

This example shows how to use MATLAB® code generation to create a ROS node to move an unmanned aerial vehicle (UAV) along a predefined circular path and a set of specified custom waypoints.

In this example, you deploy a MATLAB function as a standalone ROS node to control a simulated PX4® UAV with Gazebo. For more information on ROS node generation from MATLAB code, see "MATLAB Programming for Code Generation" on page 1-158.

Start PX4 SITL Gazebo Simulator

Download a virtual machine using instructions in "Get Started with Gazebo and Simulated TurtleBot" on page 1-206.

- Start the Ubuntu® virtual machine desktop.
- In the Ubuntu desktop, click the **Gazebo PX4 SITL** icon to start the simulated PX4 Gazebo world.
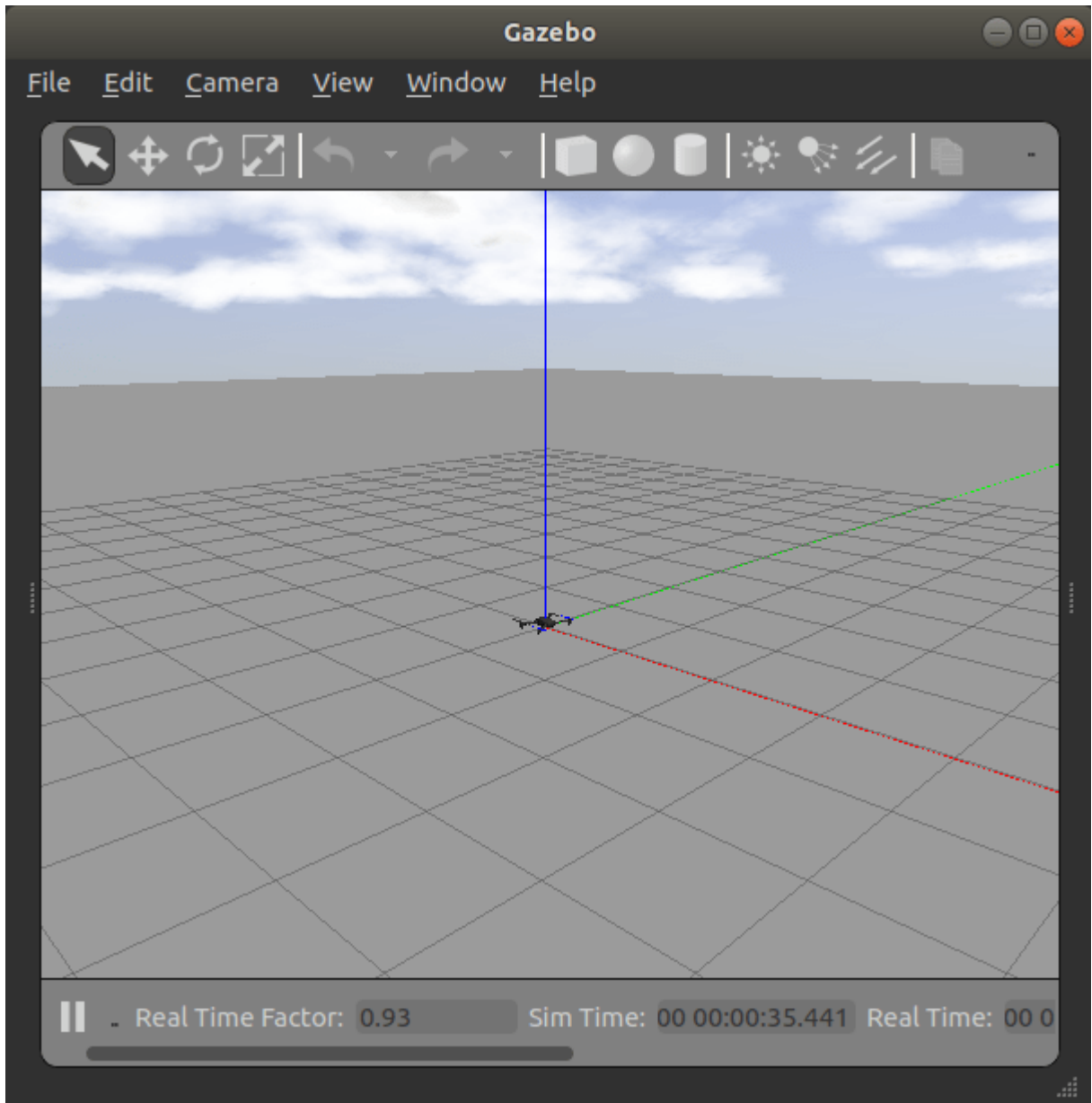
- Specify the IP address and port number of the ROS master in Gazebo so that MATLAB® can communicate with the PX4 simulator. For this example, the ROS master in Gazebo is `http:// 192.168.192.136:11311` and your host computer address is `192.168.31.1`.
- Start the ROS network using `rosinit`.

```
ipaddress = '192.168.93.136';
%Shut down global node if one is already running
rosshutdown();
```

```
Shutting down global node /matlab_global_node_29172 with NodeURI http://192.168.93.1:62022/
```

```
rosinit(ipaddress,11311);
```

```
Initializing global node /matlab_global_node_22217 with NodeURI http://192.168.93.1:62084/
```

View the ROS topics available on the network. Verify that the topics with the namespace `/mavros` are available.

```
rostopic("list");
```

```
/clock
/diagnostics
/gazebo/link_states
/gazebo/model_states
/gazebo/parameter_descriptions
/gazebo/parameter_updates
/gazebo/set_link_state
/gazebo/set_model_state
/mavlink/from
/mavlink/gcs_ip
/mavlink/to
/mavros/actuator_control
/mavros/adsb/send
/mavros/adsb/vehicle
/mavros/altitude
/mavros/battery
/mavros/cam_imu_sync/cam_imu_stamp
/mavros/companion_process/status
/mavros/debug_value/debug
/mavros/debug_value/debug_vector
/mavros/debug_value/named_value_float
/mavros/debug_value/named_value_int
/mavros/debug_value/send
/mavros/esc_info
/mavros/esc_status
/mavros/estimator_status
/mavros/extended_state
/mavros/fake_gps/mocap/tf
/mavros/geofence/waypoints
/mavros/global_position/compass_hdg
/mavros/global_position/global
/mavros/global_position/gp_lp_offset
/mavros/global_position/gp_origin
/mavros/global_position/home
/mavros/global_position/local
/mavros/global_position/raw/fix
/mavros/global_position/raw/gps_vel
/mavros/global_position/raw/satellites
/mavros/global_position/rel_alt
/mavros/global_position/set_gp_origin
/mavros/gps_rtk/rtk_baseline
/mavros/gps_rtk/send_rtcm
/mavros/gpsstatus/gps1/raw
/mavros/gpsstatus/gps1/rtk
/mavros/gpsstatus/gps2/raw
/mavros/gpsstatus/gps2/rtk
/mavros/hil/actuator_controls
/mavros/hil/controls
/mavros/hil/gps
/mavros/hil/imu_ned
/mavros/hil/optical_flow
/mavros/hil/rc_inputs
/mavros/hil/state
```

```
/mavros/home_position/home
/mavros/home_position/set
/mavros/imu/data
/mavros/imu/data_raw
/mavros/imu/diff_pressure
/mavros/imu/mag
/mavros/imu/static_pressure
/mavros/imu/temperature_baro
/mavros/imu/temperature_imu
/mavros/landing_target/lt_marker
/mavros/landing_target/pose
/mavros/landing_target/pose_in
/mavros/local_position/accel
/mavros/local_position/odom
/mavros/local_position/pose
/mavros/local_position/pose_cov
/mavros/local_position/velocity_body
/mavros/local_position/velocity_body_cov
/mavros/local_position/velocity_local
/mavros/log_transfer/raw/log_data
/mavros/log_transfer/raw/log_entry
/mavros/manual_control/control
/mavros/manual_control/send
/mavros/mission/reached
/mavros/mission/waypoints
/mavros/mocap/pose
/mavros/mount_control/command
/mavros/mount_control/orientation
/mavros/obstacle/send
/mavros/odometry/in
/mavros/odometry/out
/mavros/onboard_computer/status
/mavros/param/param_value
/mavros/play_tune
/mavros/px4flow/ground_distance
/mavros/px4flow/raw/optical_flow_rad
/mavros/px4flow/raw/send
/mavros/px4flow/temperature
/mavros/radio_status
/mavros/rallypoint/waypoints
/mavros/rc/in
/mavros/rc/out
/mavros/rc/override
/mavros/setpoint_accel/accel
/mavros/setpoint_attitude/cmd_vel
/mavros/setpoint_attitude/thrust
/mavros/setpoint_position/global
/mavros/setpoint_position/global_to_local
/mavros/setpoint_position/local
/mavros/setpoint_raw/attitude
/mavros/setpoint_raw/global
/mavros/setpoint_raw/local
/mavros/setpoint_raw/target_attitude
/mavros/setpoint_raw/target_global
/mavros/setpoint_raw/target_local
/mavros/setpoint_trajectory/desired
/mavros/setpoint_trajectory/local
/mavros/setpoint_velocity/cmd_vel
```

```
/mavros/setpoint_velocity/cmd_vel_unstamped
/mavros/state
/mavros/statustext/recv
/mavros/statustext/send
/mavros/target_actuator_control
/mavros/time_reference
/mavros/timesync_status
/mavros/trajectory/desired
/mavros/trajectory/generated
/mavros/trajectory/path
/mavros/vfr_hud
/mavros/vision_pose/pose
/mavros/vision_pose/pose_cov
/mavros/vision_speed/speed_twist_cov
/mavros/wind_estimation
/rosout
/rosout_agg
/tf
/tf_static
```

Generate Custom Messages

This example uses MAVROS to control the PX4 drone using the mavros_msgs package.

Verify that the messages are visible in MATLAB:

```
msgs = rosmsg("list");
mavrosMsgs = msgs(startsWith(msgs,"mavros_msgs"))
```

```
mavrosMsgs = 127×1 cell
    {'mavros_msgs/ADSBVehicle'                  }
    {'mavros_msgs/ActuatorControl'              }
    {'mavros_msgs/Altitude'                     }
    {'mavros_msgs/AttitudeTarget'               }
    {'mavros_msgs/BatteryStatus'                }
    {'mavros_msgs/CamIMUStamp'                  }
    {'mavros_msgs/CommandBoolRequest'           }
    {'mavros_msgs/CommandBoolResponse'          }
    {'mavros_msgs/CommandCode'                  }
    {'mavros_msgs/CommandHomeRequest'           }
    {'mavros_msgs/CommandHomeResponse'          }
    {'mavros_msgs/CommandIntRequest'            }
    {'mavros_msgs/CommandIntResponse'           }
    {'mavros_msgs/CommandLongRequest'           }
    {'mavros_msgs/CommandLongResponse'          }
    {'mavros_msgs/CommandTOLRequest'            }
    {'mavros_msgs/CommandTOLResponse'           }
    {'mavros_msgs/CommandTriggerControlRequest' }
    {'mavros_msgs/CommandTriggerControlResponse'}
    {'mavros_msgs/CommandTriggerIntervalRequest'}
    {'mavros_msgs/CommandTriggerIntervalResponse'}
    {'mavros_msgs/CommandVtolTransitionRequest' }
    {'mavros_msgs/CommandVtolTransitionResponse'}
    {'mavros_msgs/CompanionProcessStatus'       }
    {'mavros_msgs/DebugValue'                   }
    {'mavros_msgs/ESCInfo'                      }
    {'mavros_msgs/ESCInfoItem'                  }
    {'mavros_msgs/ESCStatus'                    }
```

```
{'mavros_msgs/ESCStatusItem'                    }
{'mavros_msgs/EstimatorStatus'                  }
    ⋮
```

Fly the PX4 in a Circular Path

Open the MATLAB function px4sitlCircularLoop and examine the code.

The function has three sections:

1   The initialization sequence

2   The controller loop that moves the PX4 along a circular path

3   The landing sequence.

Initialization and Circular Path Waypoint Setup

Declare all the required ROS publishers, subscribers, and service clients to communicate over MAVROS.

```
stateSub = rossubscriber("/mavros/state","mavros_msgs/State","DataFormat","struct");
% Create a SetMode service client
setModeCli = rossvcclient("/mavros/set_mode","mavros_msgs/SetMode","DataFormat","struct");
% Create a arm command service client
armingCli = rossvcclient("/mavros/cmd/arming","mavros_msgs/CommandBool","DataFormat","struct");
% Subscribe to landing message
landNowSub = rossubscriber("/land_message","std_msgs/Bool","DataFormat","struct");
% Subscribe to current position of PX4
currPosSub = rossubscriber("/mavros/local_position/pose","geometry_msgs/PoseStamped","DataFormat"
% Create a publisher for PX4 position command
posePub = rospublisher("/mavros/setpoint_position/local","geometry_msgs/PoseStamped","DataFormat"
poseMsg = rosmessage(posePub);
```

Create a circular path along the origin.

```
radius = 5.0; % radius in meters
% Set the number of steps to divide the circular path
numSteps = 100.0;
numPoints = radius * numSteps;
% Create [x,y] points along a circle with specified radius
xpoints = radius * sin(linspace(-pi,pi,numPoints));
ypoints = radius * cos(linspace(-pi,pi,numPoints));
altitude = 2.0; % meters

if isempty(coder.target)
    figure("Name","Circular trajectory");
    plot3(xpoints,ypoints,altitude*ones(1,numPoints));
    grid on;
end
```
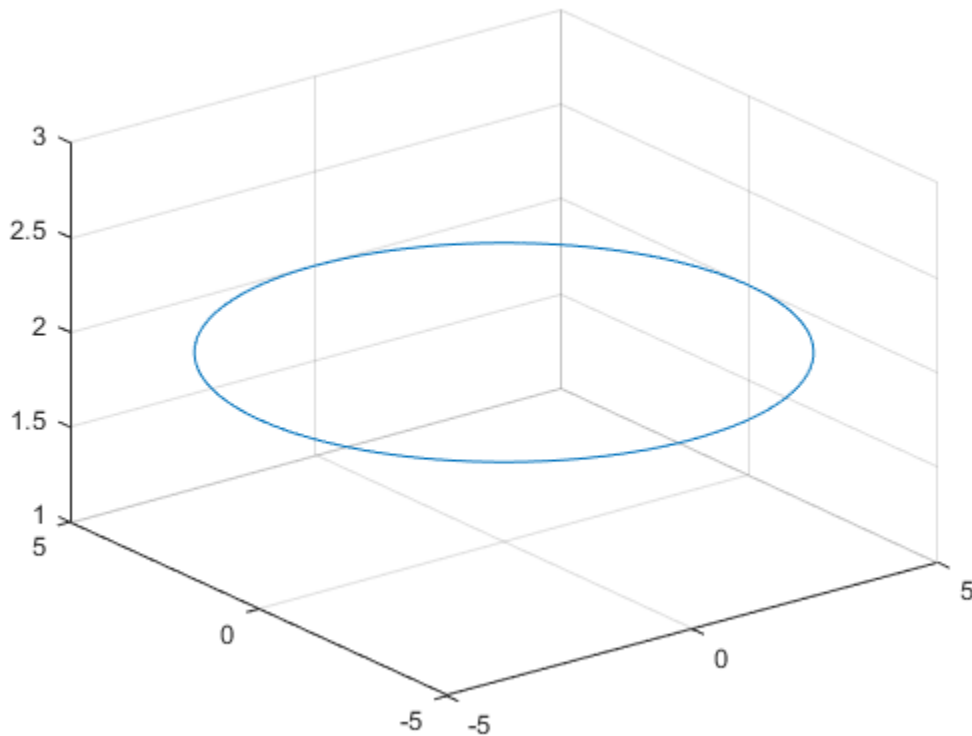
The `numSteps` variable controls the granularity of the waypoints.

The rate with which the topic is published is controlled by the `rosrate` function. For this example, set the rate to 20 Hz.

```
% Publish the messages at 20 Hz
r = rosrate(20);
```

Circular Path Follower Control Loop

In `px4sitlCircularLoop` function, the control loop initializes the takeoff sequence by changing the mode and arming PX4. If the PX4 is ready to accept commands, this function returns immediately.

```
exampleHelperUAVTakeOff(stateSub,setModeCli,armingCli,5.0);
```

PX4 OFFBOARD mode set

The control loop sets the X and Y fields of the pose message of type `geometry_msgs/PoseStamped` and sends the message over the `/mavros/setpoint_position/local` topic. The altitude is maintained at 2 meters.

In MATLAB, the control loop exits after PX4 follows the circular path three times.

In C++ ROS node deployed from `px4sitlCircularLoop` function, the control loop subscribes to the `/land_message` topic of type `std_msgs/Bool` and exits when the message value is set to `true`.

After exiting the control loop, the function runs the landing sequence.

Landing Sequence

During landing, the MATLAB function calculates the waypoints from current position on the circular path back to the origin. The PX4 follows the waypoints to reach the origin position while maintaining the altitude. After it reaches the origin, the `exampleHelperUAVLand` function lands the PX4.

```
exampleHelperUAVLand(posePub,poseMsg,r);
```

```
Landed successfully
```

Prepare Workspace on Ubuntu VM

Before deploying the node on the VM, create a Catkin workspace folder by running the following commands in MATLAB.

```
vmdev = rosdevice(ipaddress,'user','password');
vmdev.CatkinWorkspace = '/tmp/px4stil_catkinws';
system(vmdev,['mkdir -p ', vmdev.CatkinWorkspace, '/src; source /opt/ros/noetic/setup.bash; cd '
```

Generate and Deploy ROS Node

Create a MATLAB Coder™ configuration object.

```
cfg = coder.config("exe");
cfg.Hardware = coder.hardware("Robot Operating System (ROS)");
cfg.Hardware.DeployTo = "Remote Device";
cfg.Hardware.BuildAction = "Build and run";
cfg.HardwareImplementation.ProdHWDeviceType = "Intel->x86-64 (Linux 64)";
cfg.HardwareImplementation.ProdLongLongMode = true; % Use 'long long' for int64 or uint64 data ty
```

Set the deployment device properties of the Ubuntu VM.

```
cfg.Hardware.RemoteDeviceAddress = ipaddress;
cfg.Hardware.CatkinWorkspace = vmdev.CatkinWorkspace;
cfg.Hardware.RemoteDevicePassword = 'password';
cfg.Hardware.RemoteDeviceUsername = 'user';
cfg.Hardware

ans =
  Hardware with properties:

                       Name: 'Robot Operating System (ROS)'
               CPUClockRate: 1000
        PackageMaintainerName: 'ROS User'
       PackageMaintainerEmail: 'rosuser@test.com'
          RemoteDeviceAddress: '192.168.93.136'
               PackageLicense: 'BSD'
        RemoteDeviceUsername: 'user'
              CatkinWorkspace: '/tmp/px4stil_catkinws'
        RemoteDevicePassword: 'password'
               PackageVersion: '1.0.0'
                  BuildAction: 'Build and run'
                    ROSFolder: '/opt/ros/melodic'
                     DeployTo: 'Remote Device'
```

Deploy the MATLAB function as a standalone ROS node.

```
codegen("px4sitlCircularLoop","-config",cfg);
```

The generated node starts running on the ROS device. You can verify this by using the `rosdevice` object.

```
isNodeRunning(vmdev,"px4sitlCircularLoop")
```

Simulated PX4 takes off and starts flying in a circular pattern in Gazebo.
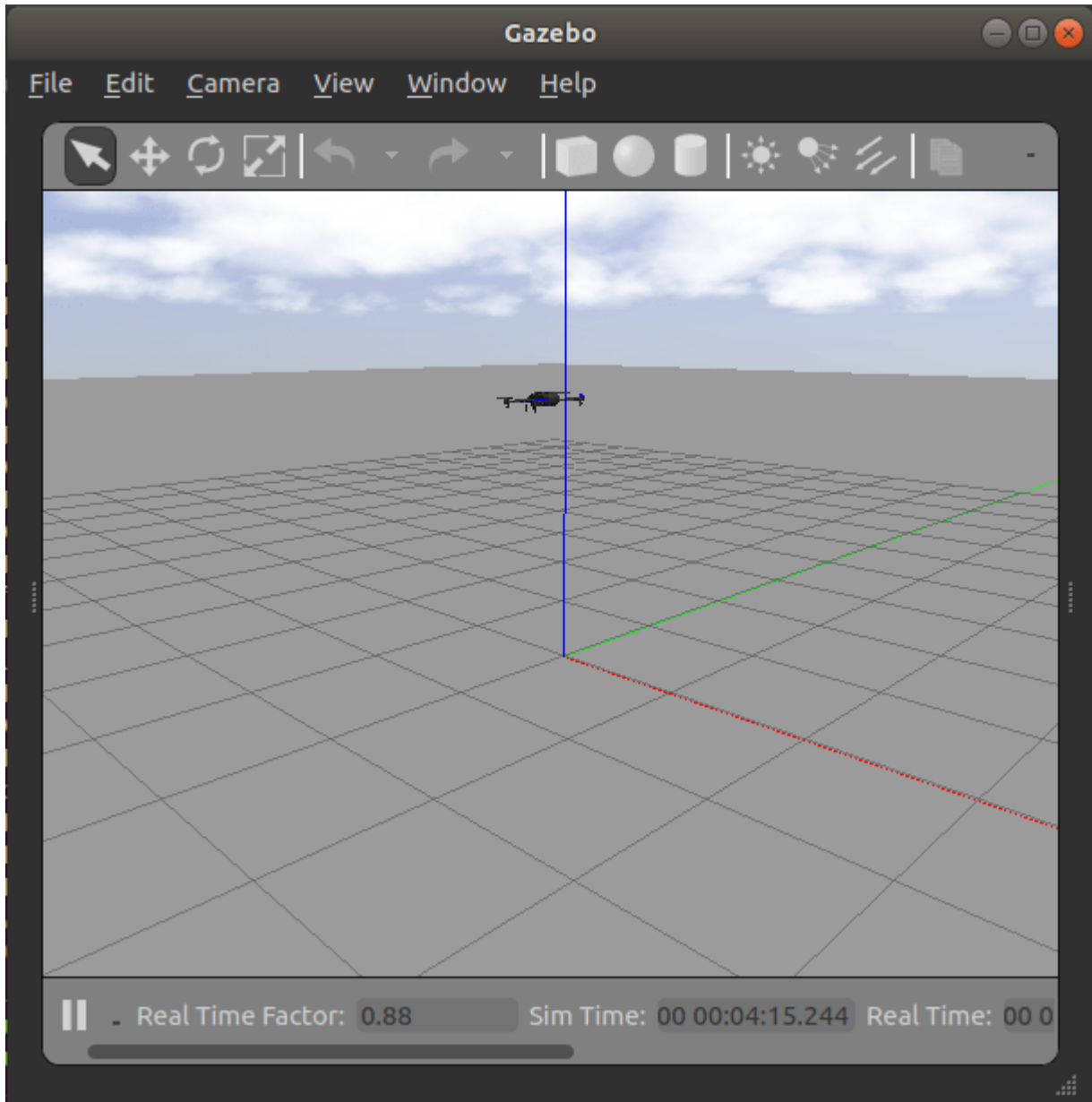
Stop the ROS node by sending the landing message from MATLAB.

```
pubLandMsg = rospublisher("/land_message","DataFormat","struct")
```

```
pubLandMsg =
  Publisher with properties:
```

```
        TopicName: '/land_message'
    NumSubscribers: 1
        IsLatching: 1
      MessageType: 'std_msgs/Bool'
       DataFormat: 'struct'
```

```
msg = rosmessage(pubLandMsg);
msg.Data = true;
send(pubLandMsg,msg);
```

Deploy a UAV Waypoint Follower ROS Node

Open the `px4sitlWaypointFollower` function and examine the code.

Similar to the `px4sitlCircularLoop` function, this function also has three sections: the initialization sequence, the trajectory follower controller loop that moves the PX4 along the desired waypoints, and the landing sequence.

Create Reference Trajectory and Waypoints

The helper function `exampleHelperCreateReferencePath` generates a trajectory along the provided waypoints in latitude, longitude, and altitude format in the ENU frame.

It uses `minsnappolytraj` (Robotics System Toolbox) to return an array with the position, velocity, and acceleration at various points along the trajectory.

```
latlonhome = [42.301389286674166 -71.37609390148994 0];

latlonpoints = [latlonhome;
    42.301389286674166 -71.37609390148994 10;
    42.302289545648726 -71.376093901489938 10;
    42.302289539236234 -71.374881155270842 10;
    42.301389280261866 -71.374881172546949 10;
    42.301389286674166 -71.37609390148994 10];
timeOfFlight = 100; % seconds
sampleSize = 100; % samples per segment
[q,qd,qdd] = exampleHelperCreateReferencePath(latlonhome,latlonpoints,timeOfFlight,sampleSize);
```
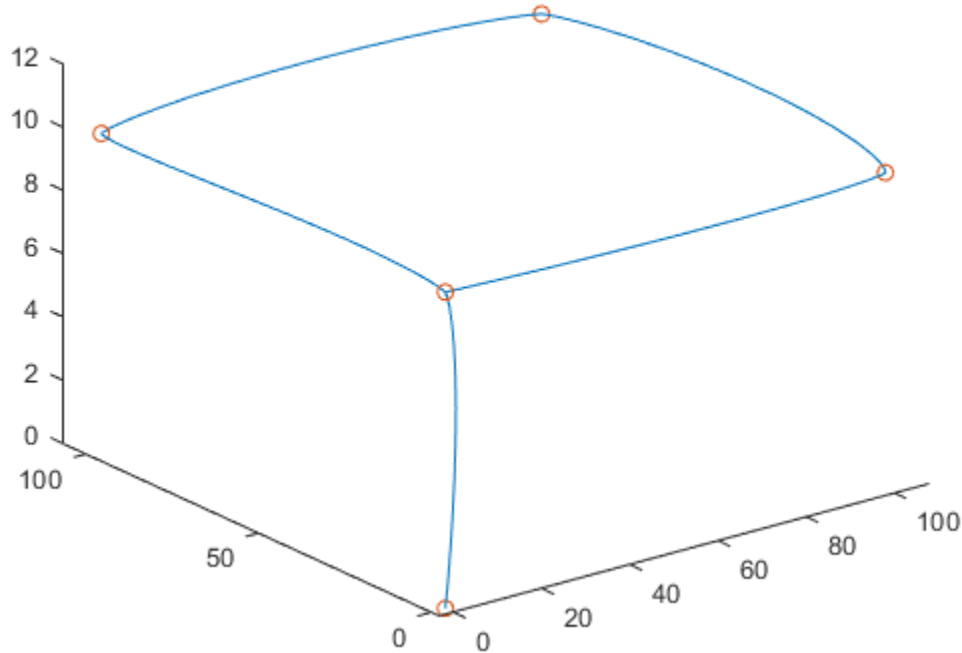
Note the precision of the latitude and longitude values. If you modify the function to use a different set of waypoints, you must provide waypoints with similar precision.

Trajectory Control Loop

The control loop starts with the takeoff sequence using `exampleHelperUAVTakeOff` helper function.

The control loop calls the `exampleHelperTrajectoryController` function, which accepts the current position and velocity values, the reference position, reference velocity, and reference accelerations generated by the `exampleHelperCreateReferencePath` function, and the PD (proportional-derivative) gain values and returns the desired acceleration values along the X, Y and Z axes.

```
accCMD = exampleHelperTrajectoryController([0 0 0]',[0 0 0]',q,qd,qdd,-4,-2)
```

accCMD = *3×1*

```
    0.0195
   -0.1541
    0.1840
```

The control loop converts these acceleration values into the acceleration message of type `geometry_msgs/Vector3Stamped`, and sends the message over the `/mavros/setpoint_accel/accel` topic.

The control loop tracks the index of the trajectory segments and the proximity to the final waypoint. The control loop exits once the PX4 reaches within 1 meter radius of the final waypoint.

Deploy Waypoint Follower ROS Node

Deploy the MATLAB function as a standalone ROS node using the `codegen` command:

```
codegen("px4sitlWaypointFollower","-config",cfg);
```

Observe the PX4 SITL in Gazebo. Once the node starts running, the PX4 UAV takes off and starts following the waypoints.

The deployed ROS node stops once it reaches the final waypoint.

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™.
- 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 Simulated TurtleBot" on page 1-206 for instructions).
- Review the "Feedback Control of a ROS-Enabled Robot" on page 1-149 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 1-149 example.

- Open `RobotController.slx`. Click the link or run `RobotController` in the Command Window.
- In the **Prepare** section under **ROS** tab, click **Hardware Settings** to open the **Hardware Implementation** pane of the Configuration Parameters dialog. 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. If you have Embedded Coder™ installed, you must enable code-generation for 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.
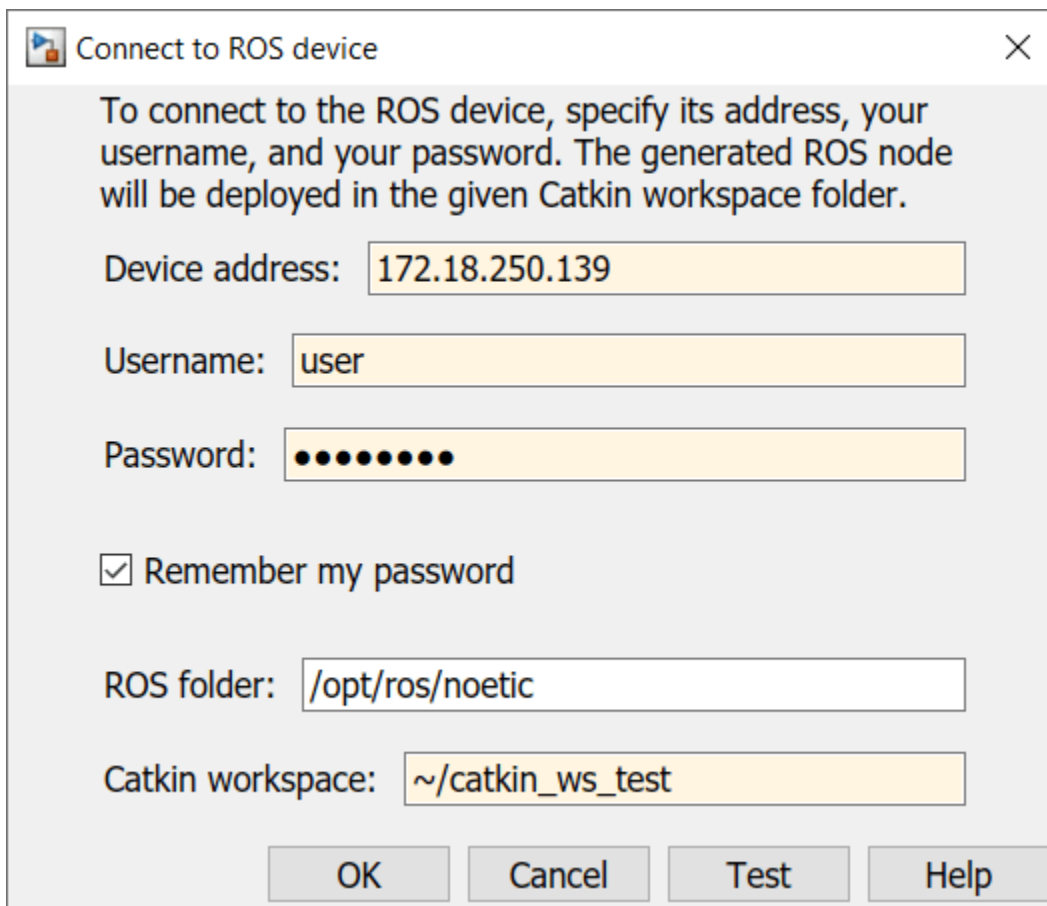
Configure the Connection to the ROS Device

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 **ROS** tab, from the **Deploy to** drop-down, click **Manage Remote Device**. This opens 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.
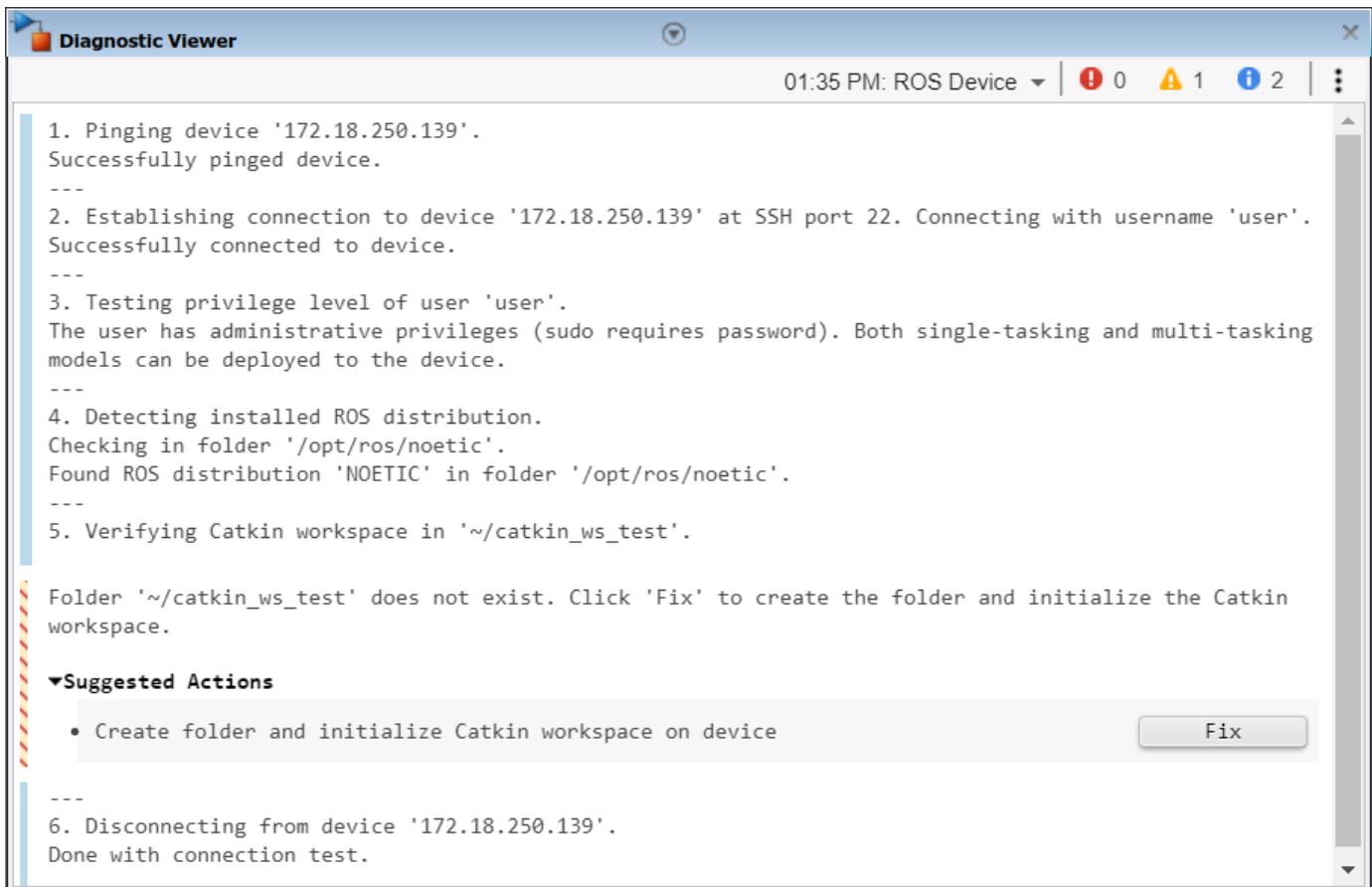
- Change the device connection settings and test them until no other warnings or errors are shown. If an automatic fix to your settings is possible, Simulink suggests it by displaying the **Fix** button. Once you have a good set of settings, click **OK** in the connection settings dialog to save the 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: '172.18.250.139'
               Username: 'user'
         AvailableNodes: {0×1 cell}
              ROSFolder: '/opt/ros/noetic'
        CatkinWorkspace: '~/catkin_ws_test'
```

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, in **Prepare** section, select **ROS Network**. Set the ROS Master (ROS 1) and Node Host network addresses to **Default**.
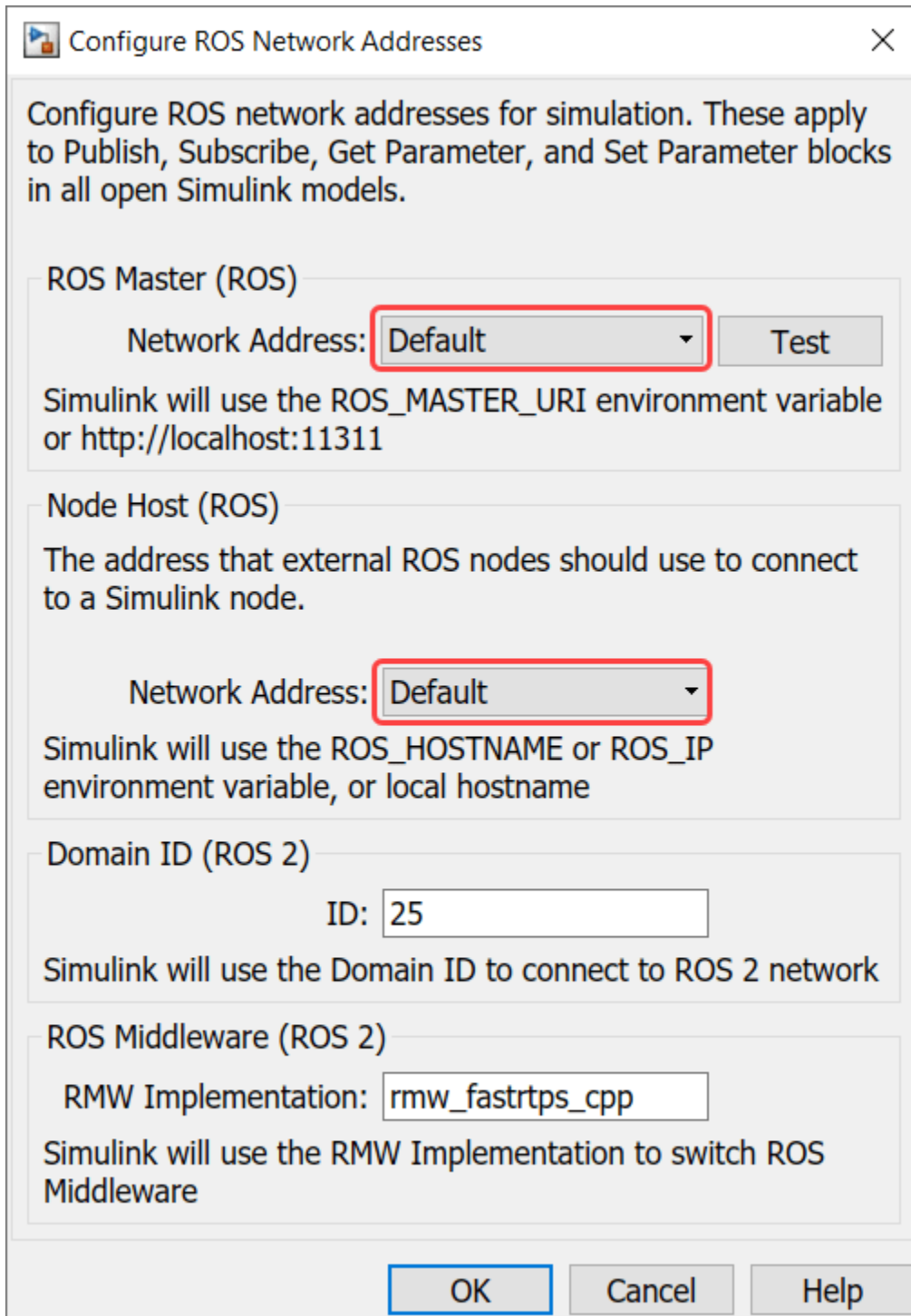
You only have to execute steps 2 - 4 once per MATLAB session, not every time you generate a ROS node.

5. Under the **ROS** tab, click **Deploy > 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 in a synchronous fashion based on the sample time of the model.

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 1-141.

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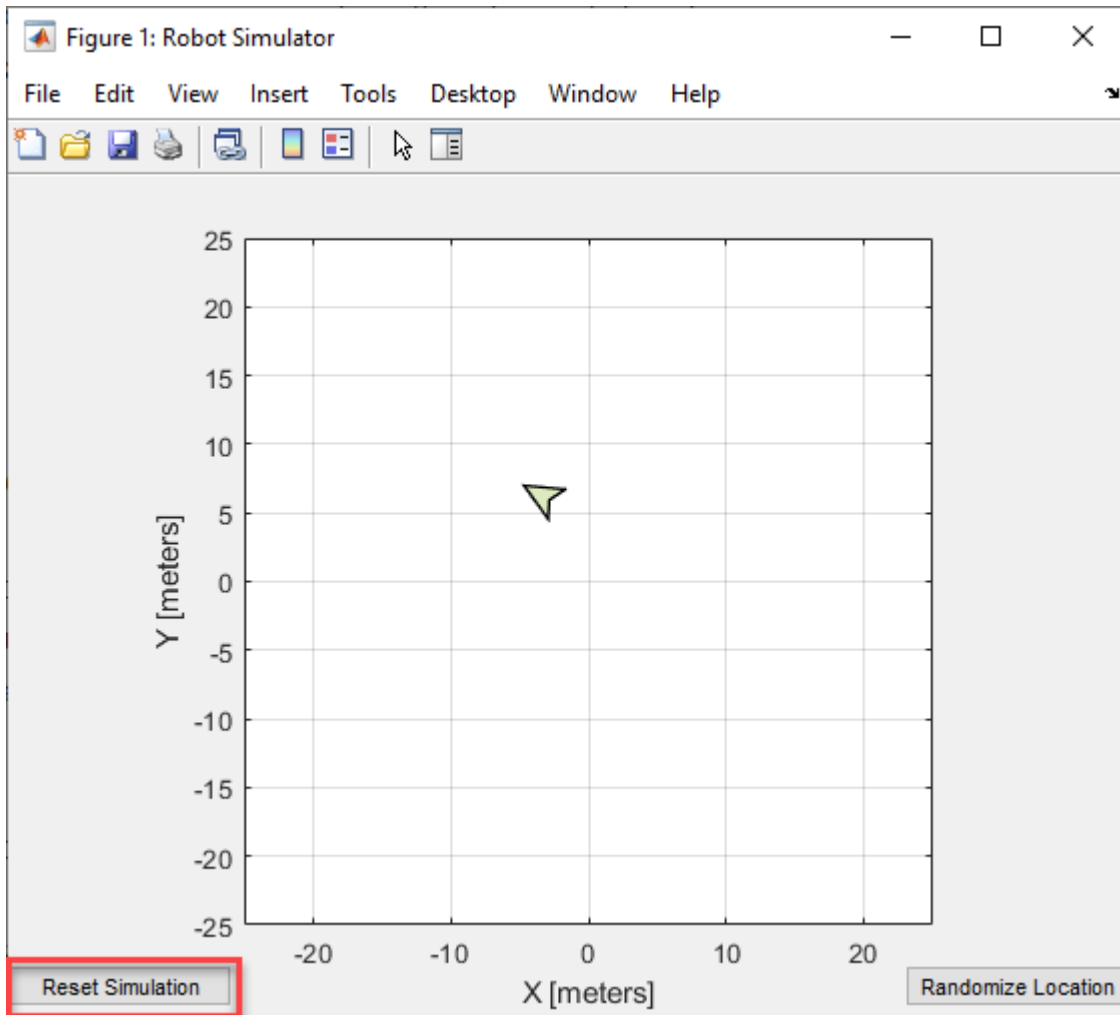
2. Verify that the simulated robot moves toward the goal (the **Desired Position** constant specified in the model). The robot stops once it reaches the goal [-10, 10].

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: '172.18.250.139'
             Username: 'user'
       AvailableNodes: {'robotcontroller'}
            ROSFolder: '/opt/ros/noetic'
      CatkinWorkspace: '~/catkin_ws_test'
```

```
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 code generation and build options: You can choose different code generation and build behaviors by specifying one of these **Deployment** options from the toolstrip under **ROS** tab.

- **Generate code** — Generates the ROS package source code on localhost or remote device.
- **Build Model** — Generates the ROS package source code and builds the standalone executable on localhost or remote device.
- **Build & Run** — Generates the ROS package source code, builds the standalone executable and starts running it on localhost or remote device.

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, in **Prepare** section, select **ROS Network** and set the ROS Master and Node Host network addresses:

Configure ROS Network Addresses   ✕

Configure ROS network addresses for simulation. These apply to Publish, Subscribe, Get Parameter, and Set Parameter blocks in all open Simulink models.

ROS Master (ROS)

Network Address: `Custom` ▾   `Test`

Hostname/IP Address: `172.18.250.139`

Port: `11311`

Node Host (ROS)

The address that external ROS nodes should use to connect to a Simulink node.

Network Address: `Default` ▾

Simulink will use the ROS_HOSTNAME or ROS_IP environment variable, or local hostname

Domain ID (ROS 2)

ID: `25`

Simulink will use the Domain ID to connect to ROS 2 network

ROS Middleware (ROS 2)

RMW Implementation: `rmw_fastrtps_cpp`

Simulink will use the RMW Implementation to switch ROS Middleware

`OK`   `Cancel`   `Help`

Specify external ROS packages as dependencies: To specify external ROS packages as dependencies to the generated ROS node, specify appropriate toolchain options. In the Configuration parameters, under **Code Generation > Toolchain Settings**, specify the **Build configuration** as **Specify** from the drop-down. Then, you can specify the **Required Packages, Include Directories, Link Libraries, Library Paths** and **Defines** based on the external ROS packages that you wish to integrate with the generated ROS node.

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.

Name

- RobotController_ert_rtw
- slprj
- build_ros_model.sh
- RobotController
- RobotController.tgz
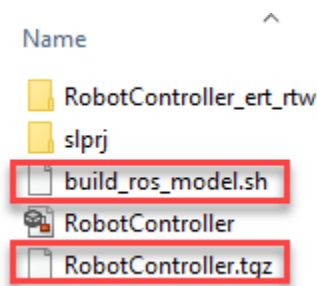
Processor-specific generated code: If you use blocks from other products (such as Computer Vision System Toolbox™), the generated code may include processor-specific optimizations that lead to compilation problems when building the ROS node on Linux. In these cases, you need to let Simulink know the 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 172.18.250.92, use the following command:

```
rosinit('NodeHost', '172.18.250.92')
```

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 1-44 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 6-30

Get Started with Gazebo and 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 the **Gazebo Empty**, **Gazebo House**, **Gazebo Office**, or **Gazebo Sign Follower ROS** icons.
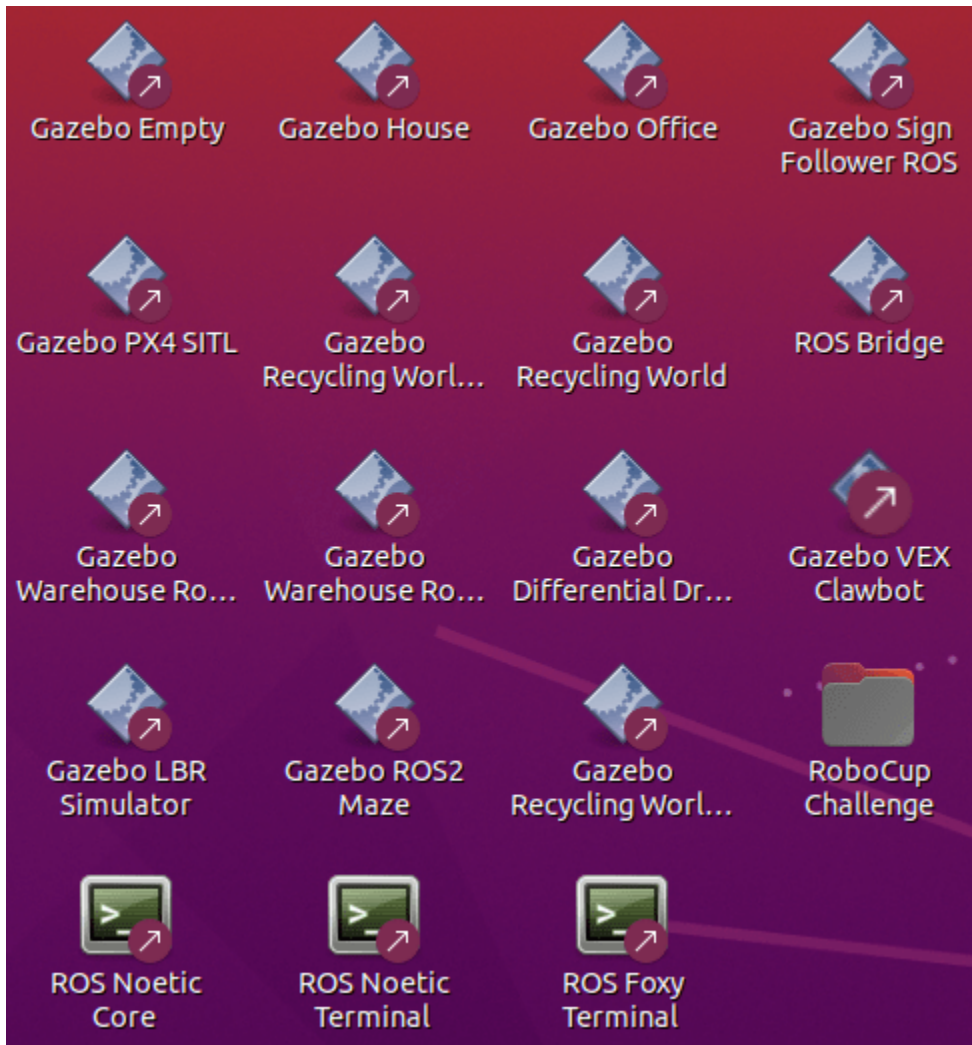
- Click **Gazebo House**. A world opens.
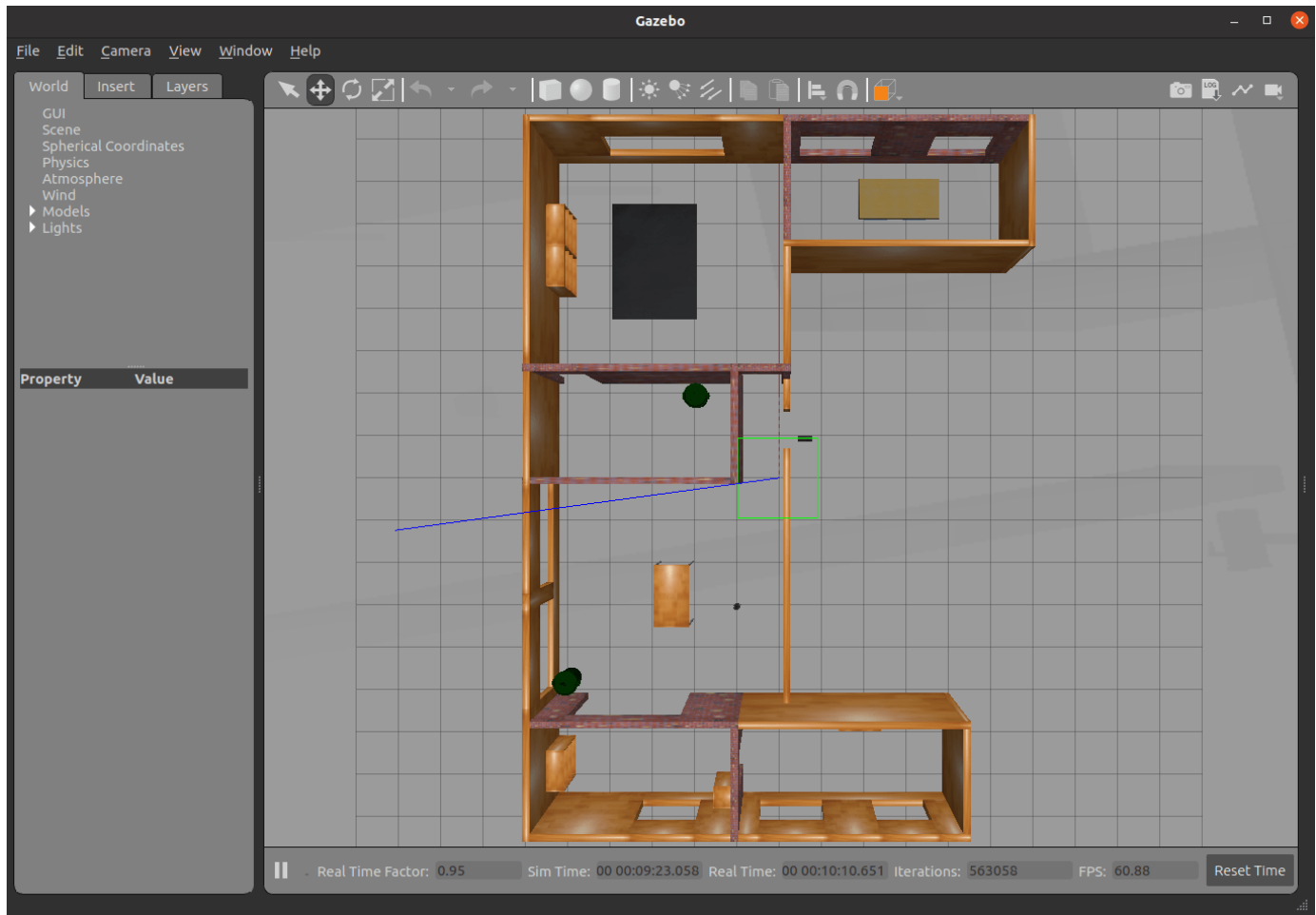
Note: If the Gazebo screen looks entirely black, refresh the image by minimizing it and then maximizing it.

- 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)

1-209
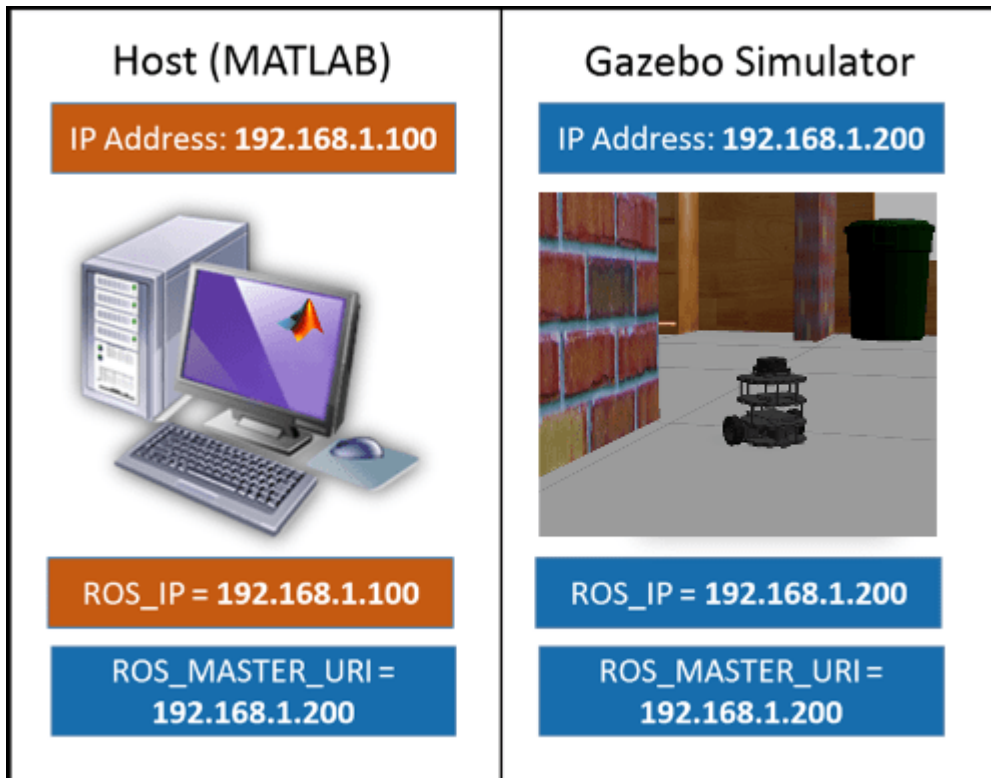
Connect to an Existing Gazebo Simulator

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 1-44 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/noetic/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.

```
user@ubuntu:~$ source /opt/ros/noetic/setup.bash
user@ubuntu:~$ rostopic list
/clock
/cmd_vel
/gazebo/link_states
/gazebo/model_states
/gazebo/parameter_descriptions
/gazebo/parameter_updates
/gazebo/performance_metrics
/gazebo/set_link_state
/gazebo/set_model_state
/imu
/joint_states
/odom
/rosout
/rosout_agg
/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 1-236

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 Simulated TurtleBot" on page 1-206

Connect to Gazebo®

On your Linux® machine, start Gazebo. If you are using the virtual machine from "Get Started with Gazebo and Simulated TurtleBot" on page 1-206, 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://192.168.93.144:11311";
rosinit(ipAddress)
```

```
Initializing global node /matlab_global_node_56950 with NodeURI http://192.168.93.1:50980/ and Ma
```

```
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 matlab.io.xml.dom.Document]
        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:
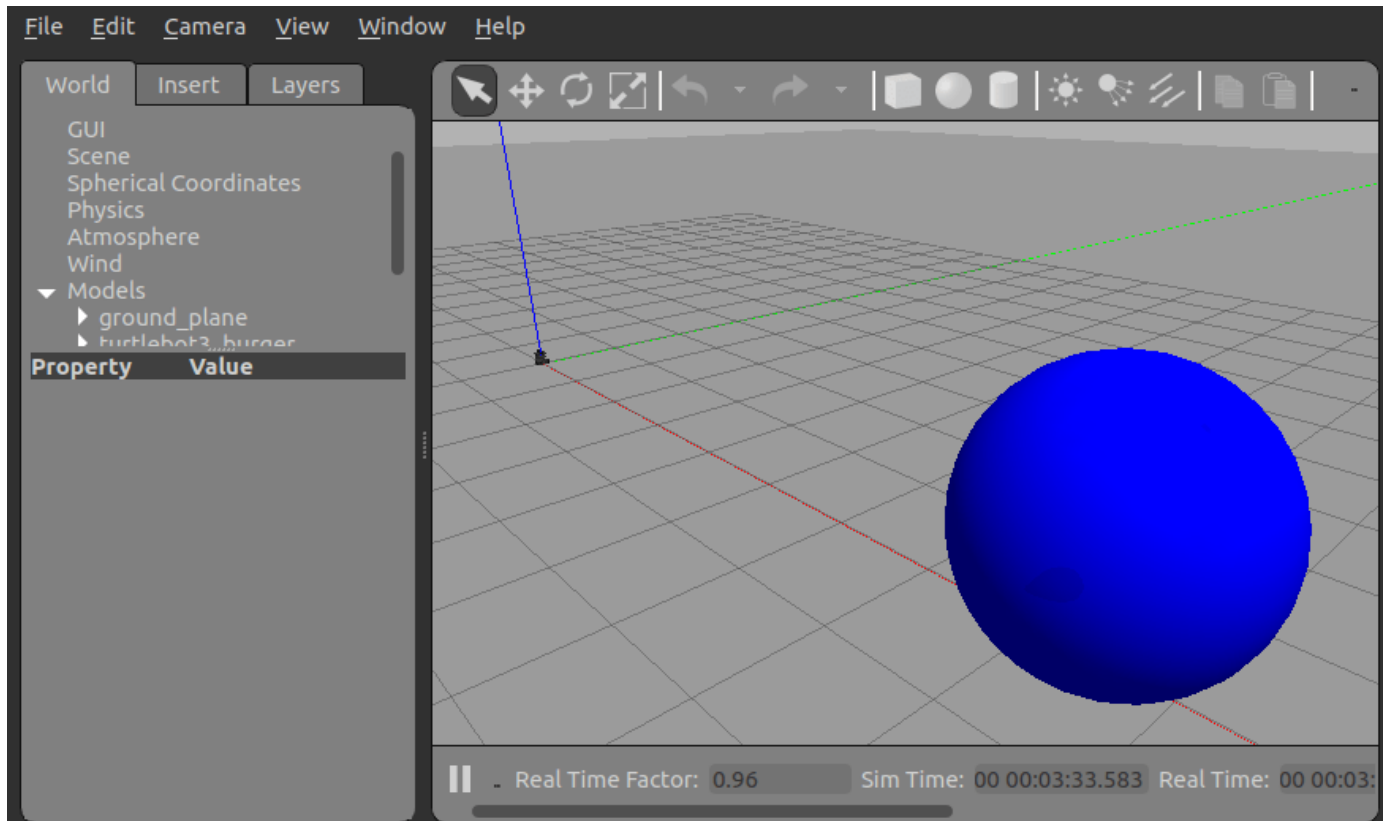
Build and Spawn Bowling Pins

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:
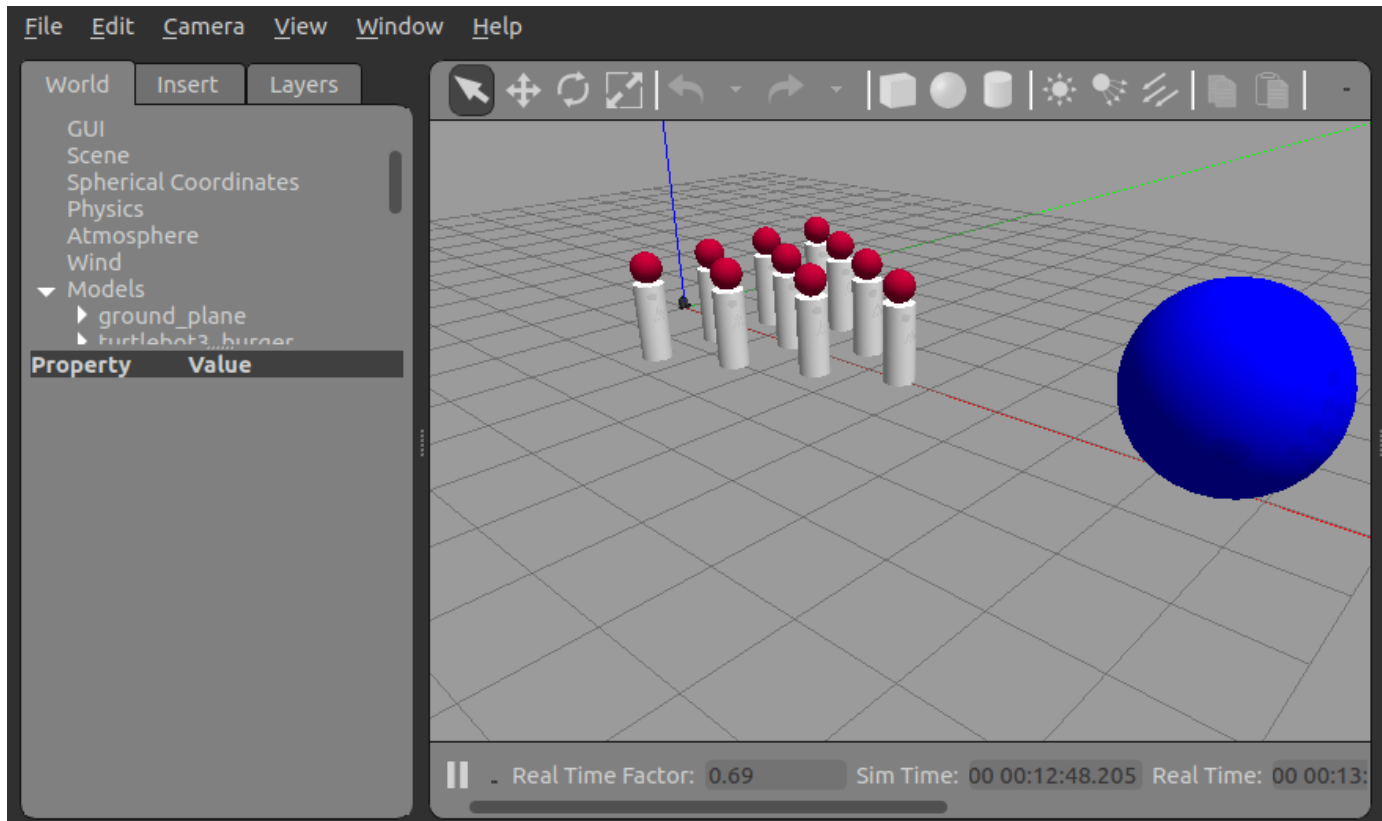
Remove Models

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:
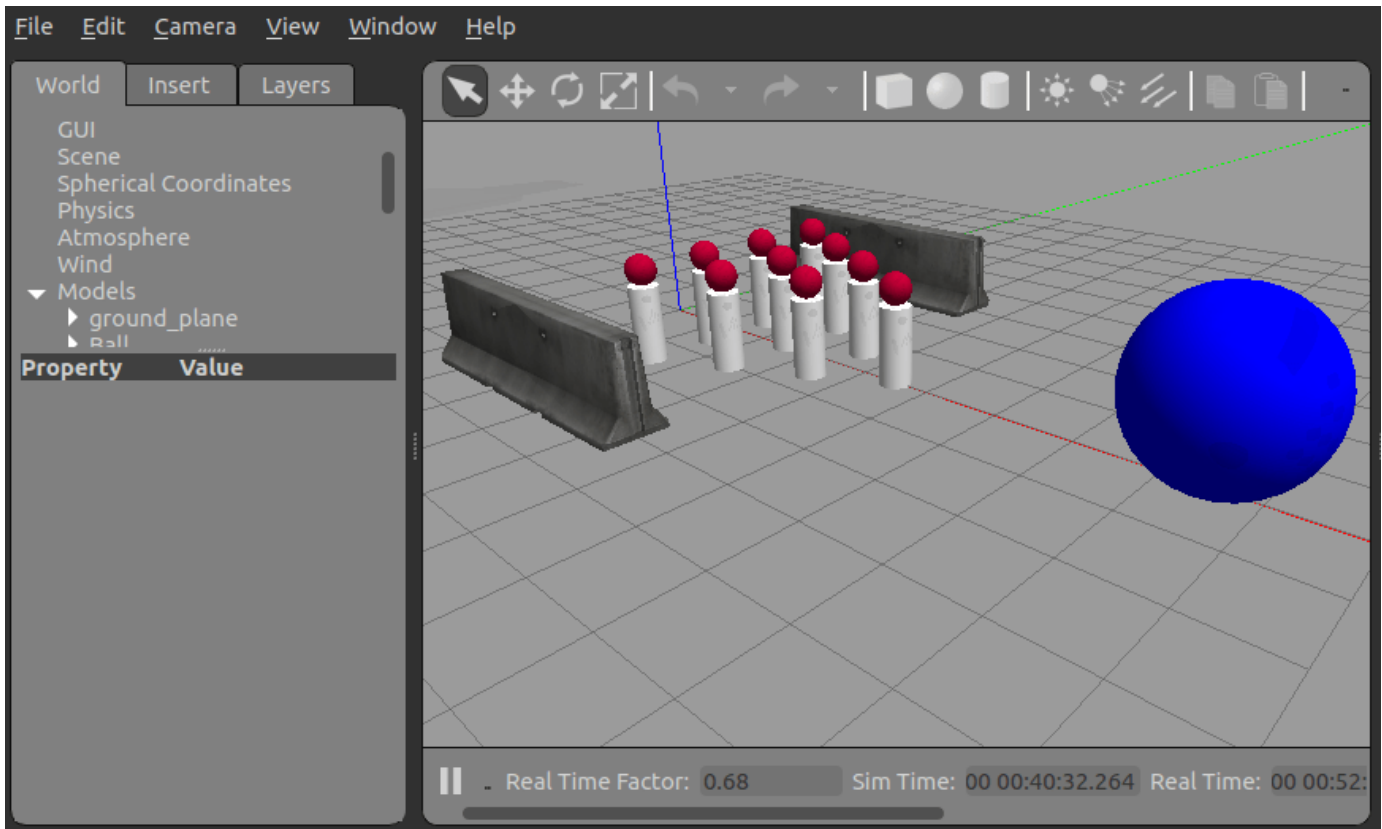
Apply Forces to the Ball

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
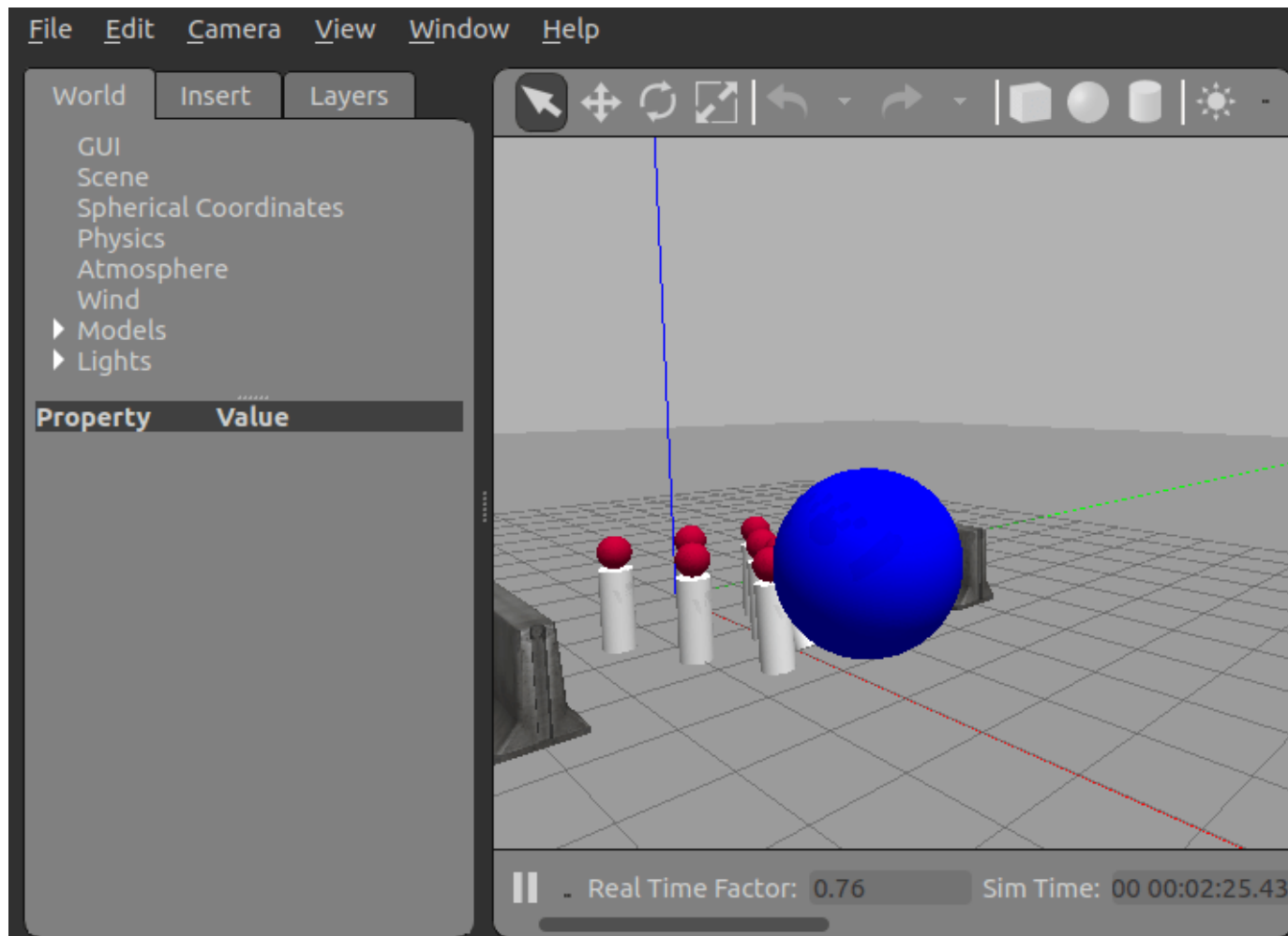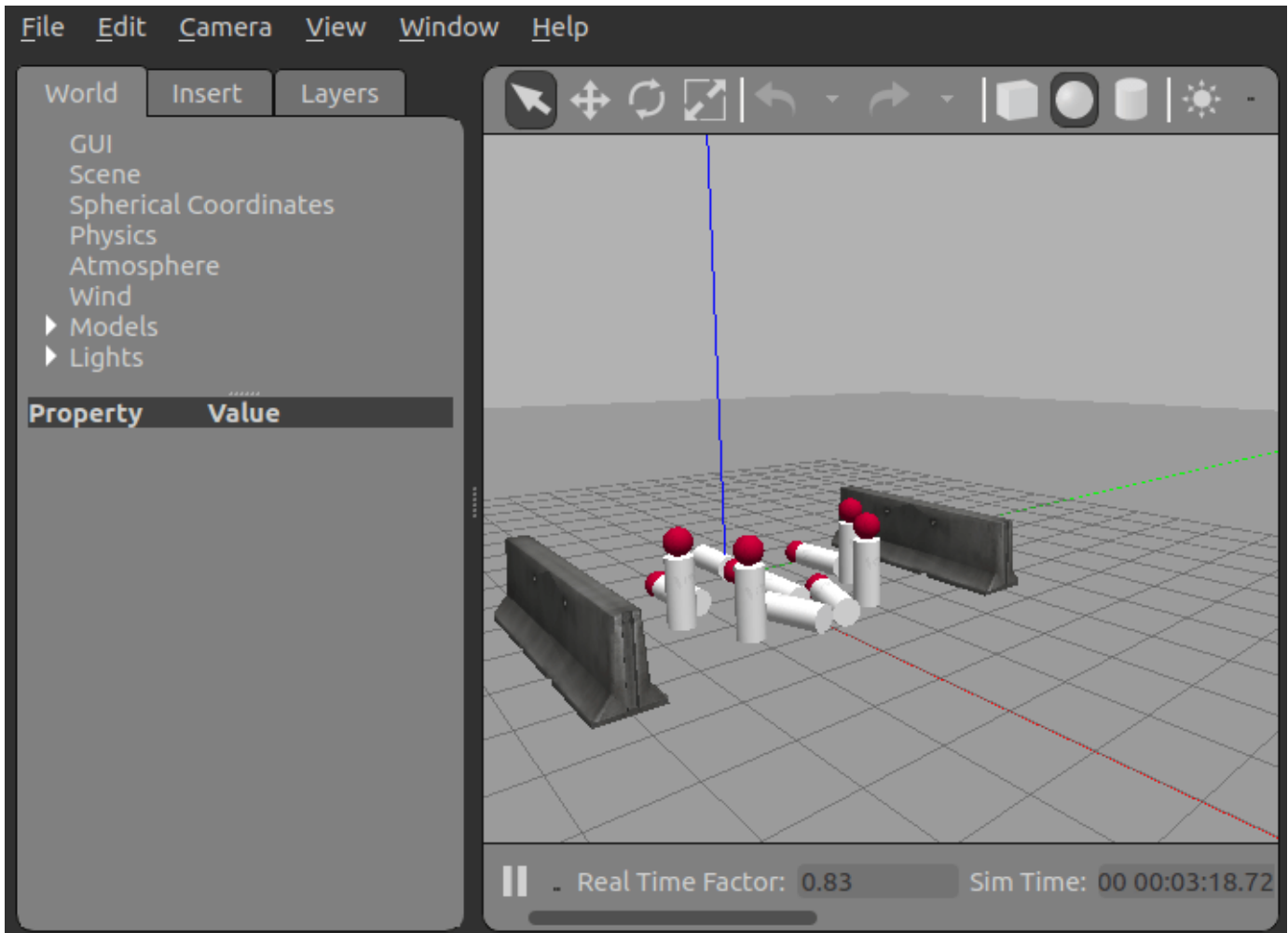
Remove Models and Shut Down

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_56950 with NodeURI http://192.168.93.1:50980/ and N

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

See Also

- "Apply Forces and Torques in Gazebo" on page 1-221

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 Simulated TurtleBot" on page 1-206, "Add, Build, and Remove Objects in Gazebo" on page 1-214

Connect to Gazebo

On your Linux® machine, start Gazebo. If you are using the virtual machine from "Get Started with Gazebo and Simulated TurtleBot" on page 1-206, 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:
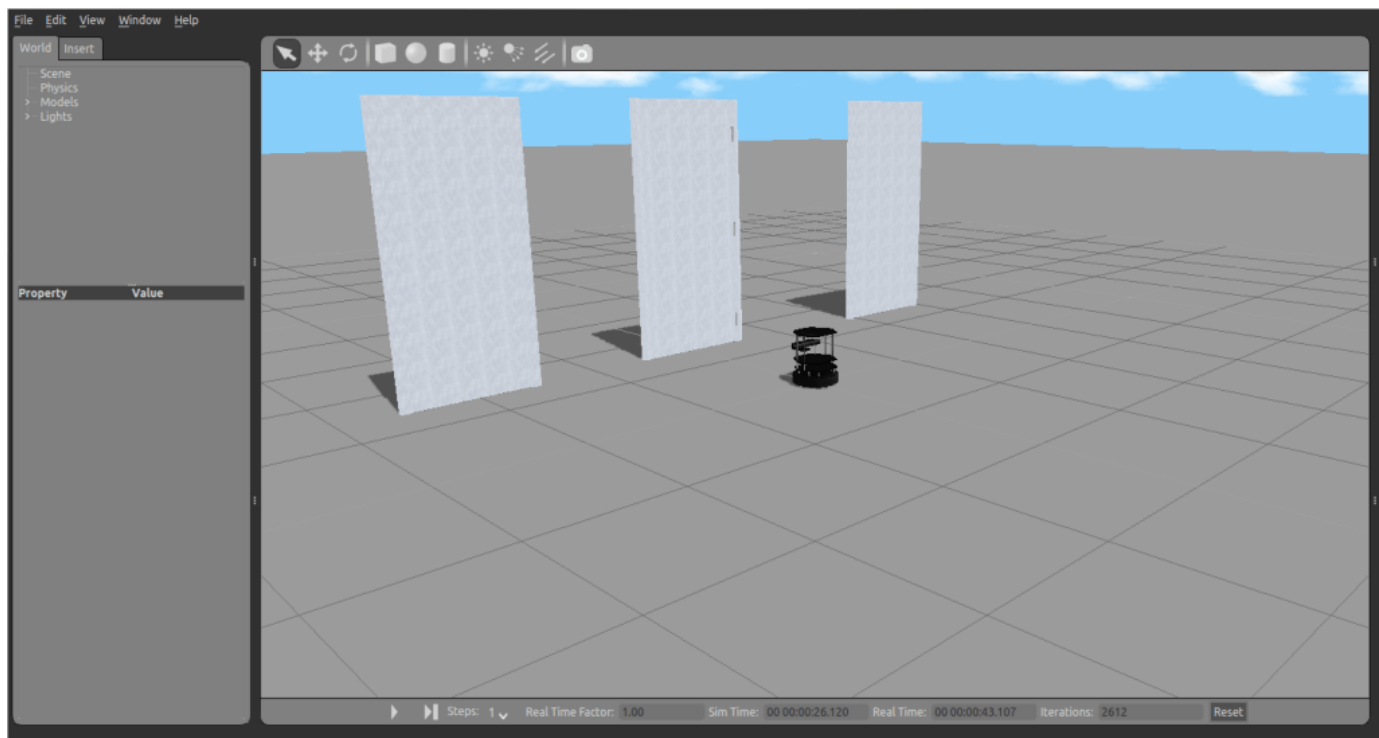
Note: When the Gazebo simulation is left idle, movable items often drift. If you see the doors moving slowly without a command, this behavior is normal. This happens because there is often more friction in the real world than there is in the ideal setting of the Gazebo simulator.

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);
```
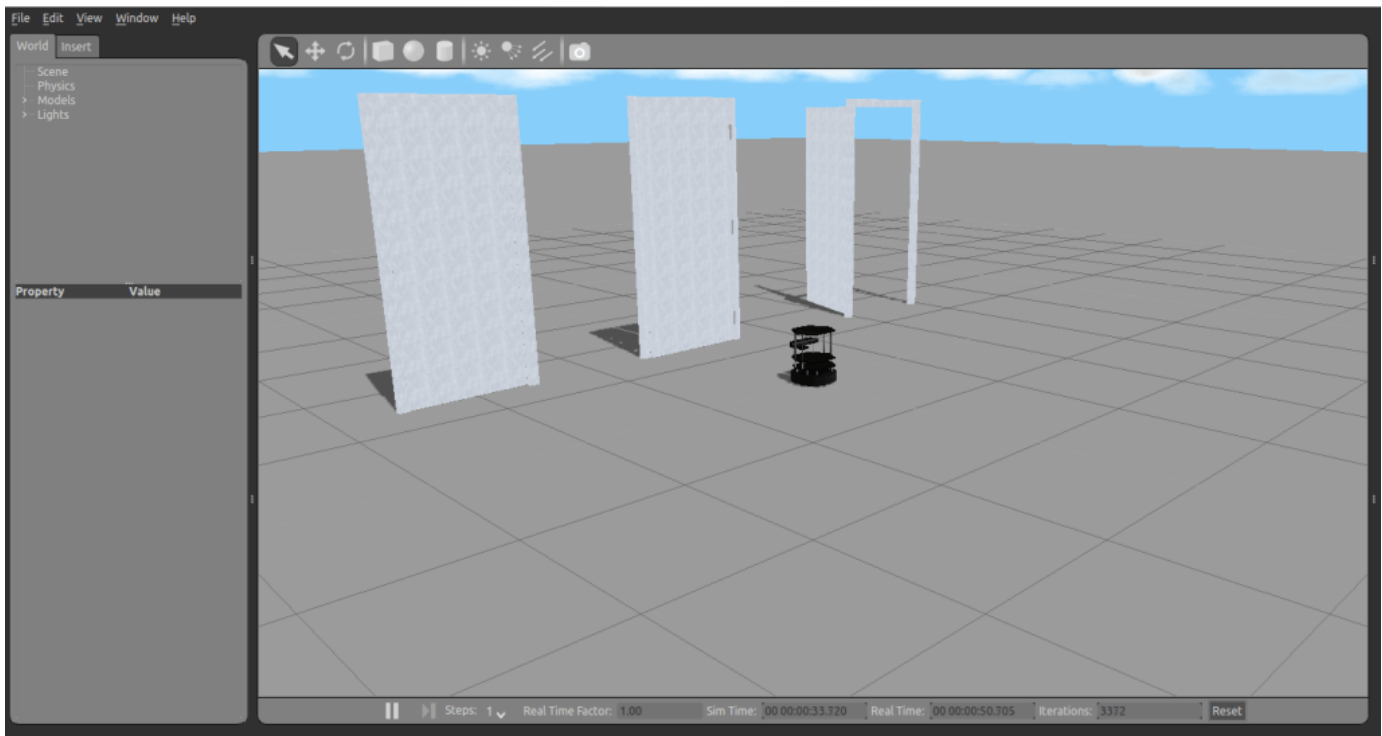
The second method is to apply a torque to the door link instead of the hinge joint. This method is not as clean because the torque is applied to the center of mass of the link (which is the door in this case) and is not applied around the axis of rotation. This method still produces a torque that moves 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);
```
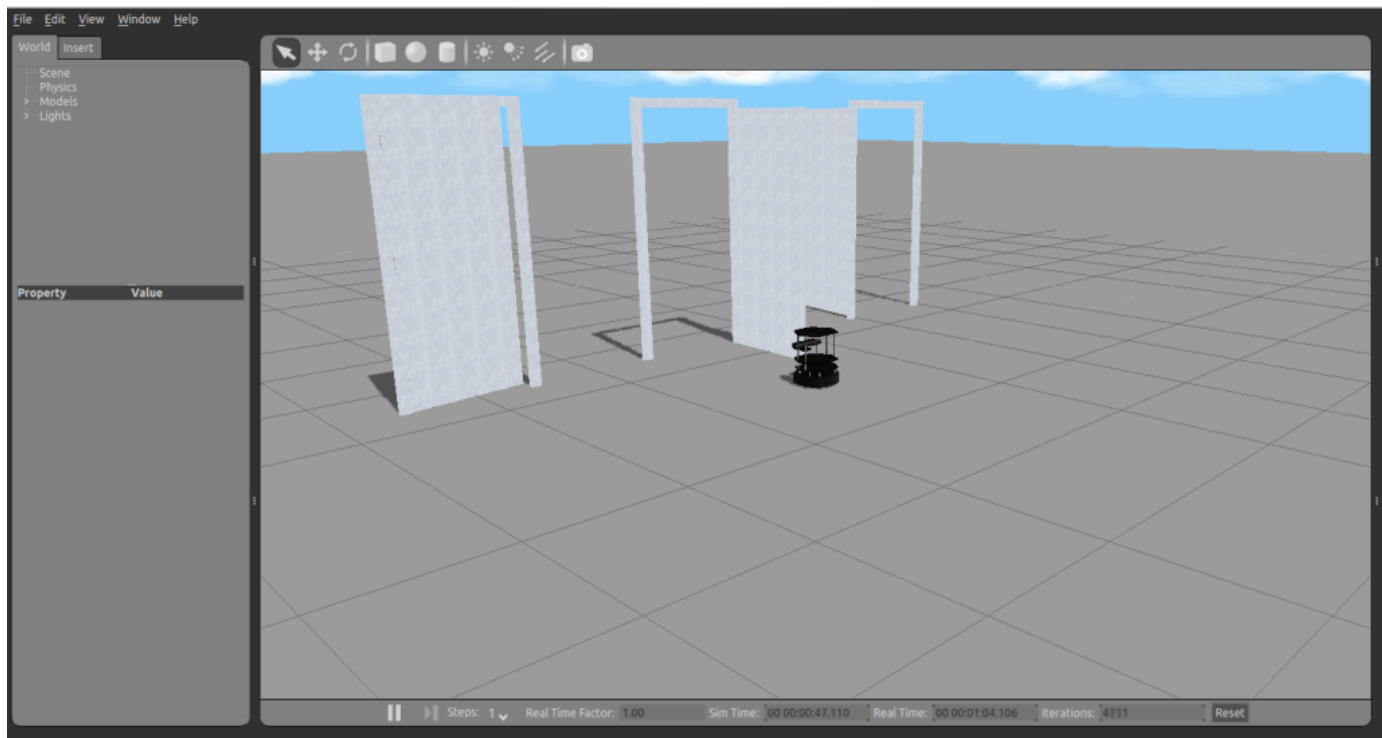
You can apply a force (instead of a torque) directly to the center of mass of the door for it to move. The commands are:

```
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
```
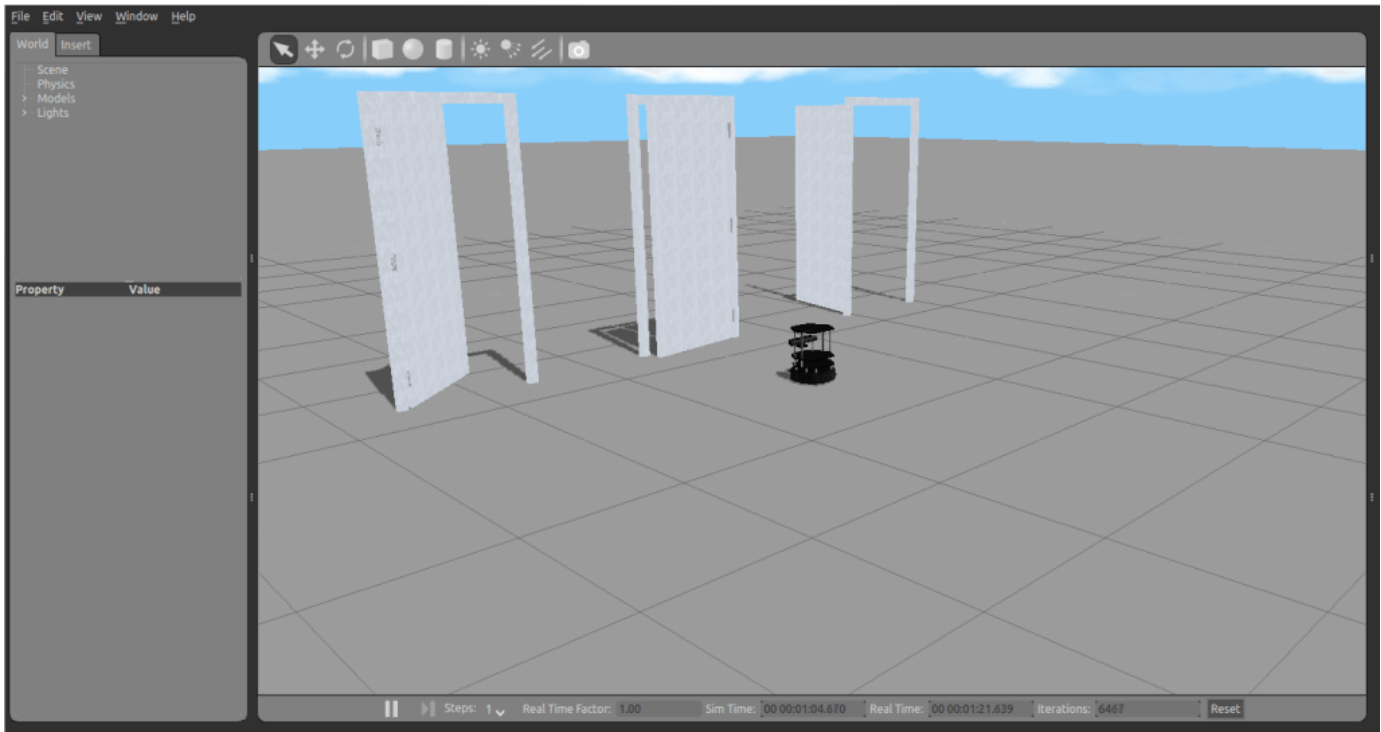
Create TurtleBot Objects for Manipulation

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]);
```
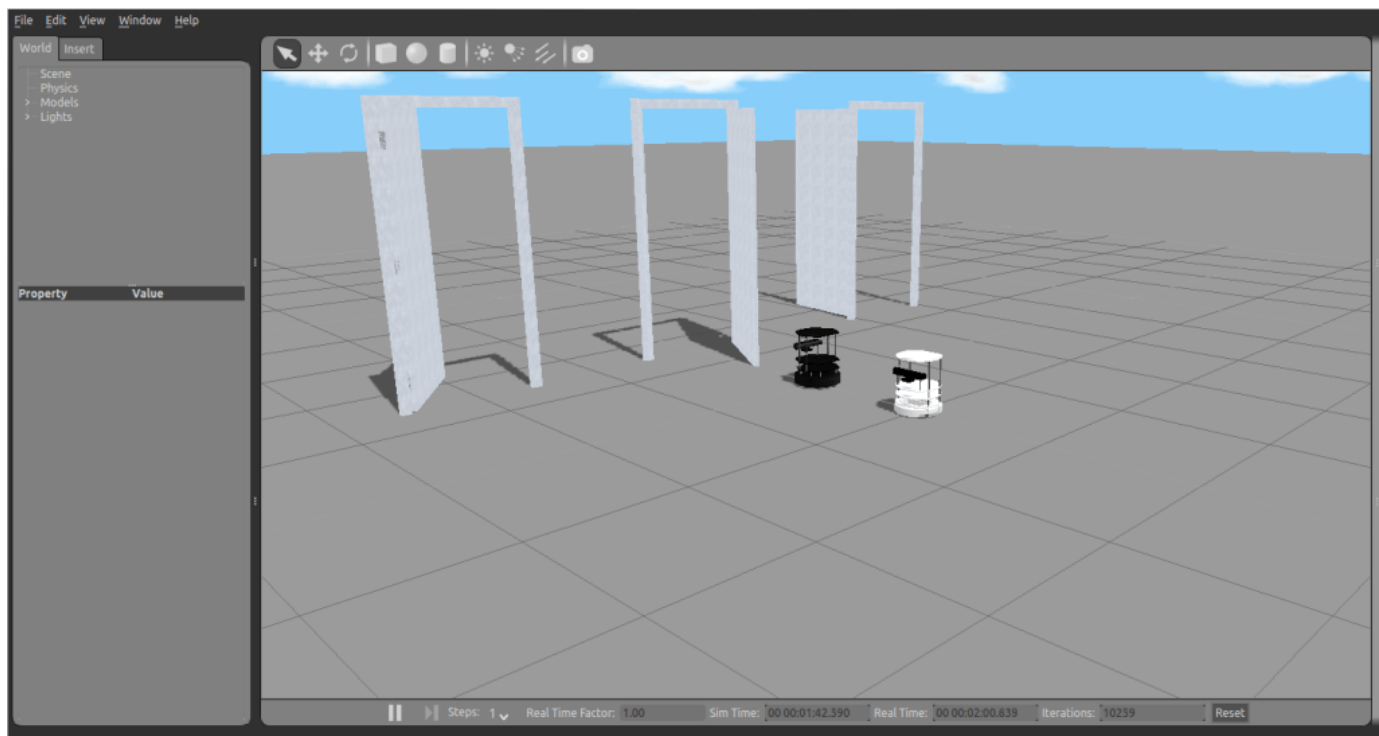
Using `applyForce`, make the right wheel of the TurtleBot Create move by applying an external torque to it from the `ExampleHelperGazeboSpawnedModel` object.

```
[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)
```
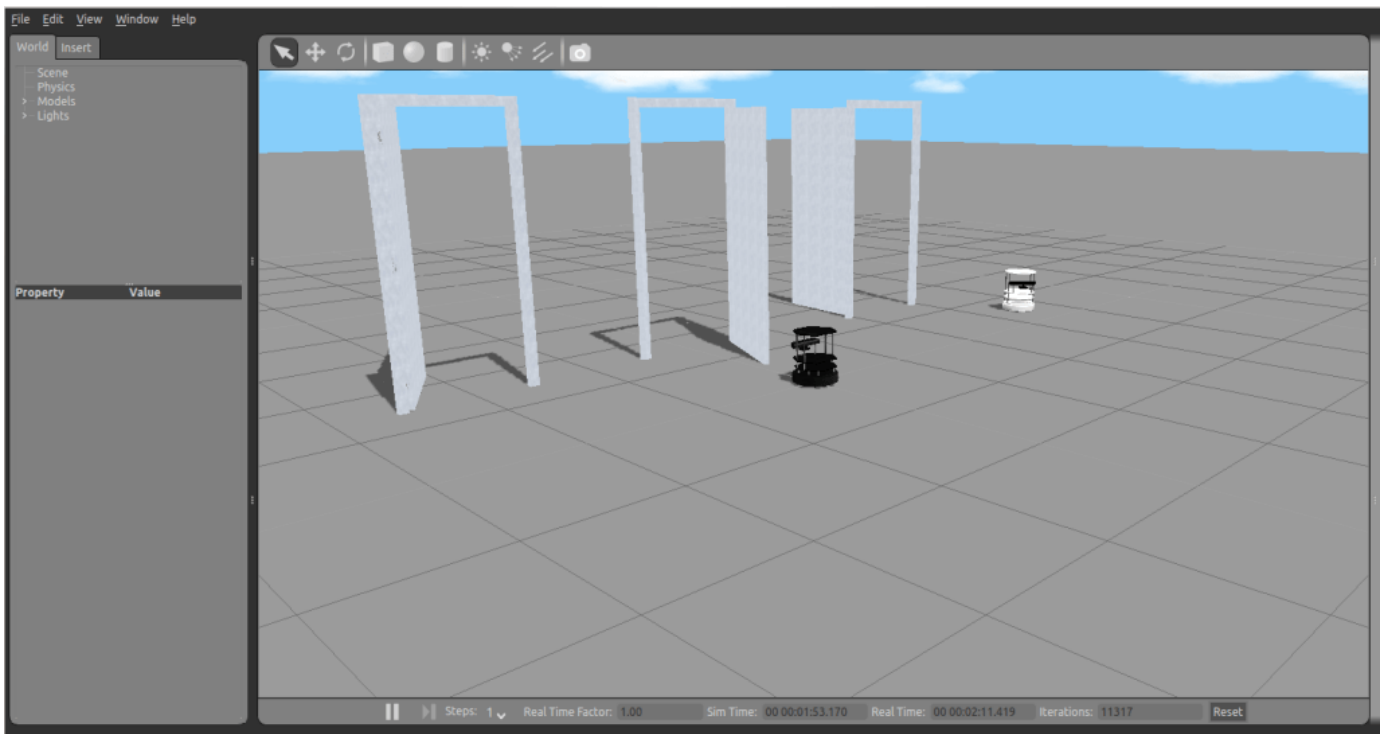
You can experiment with application of forces to a TurtleBot base instead of to the wheels.

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'        }
```
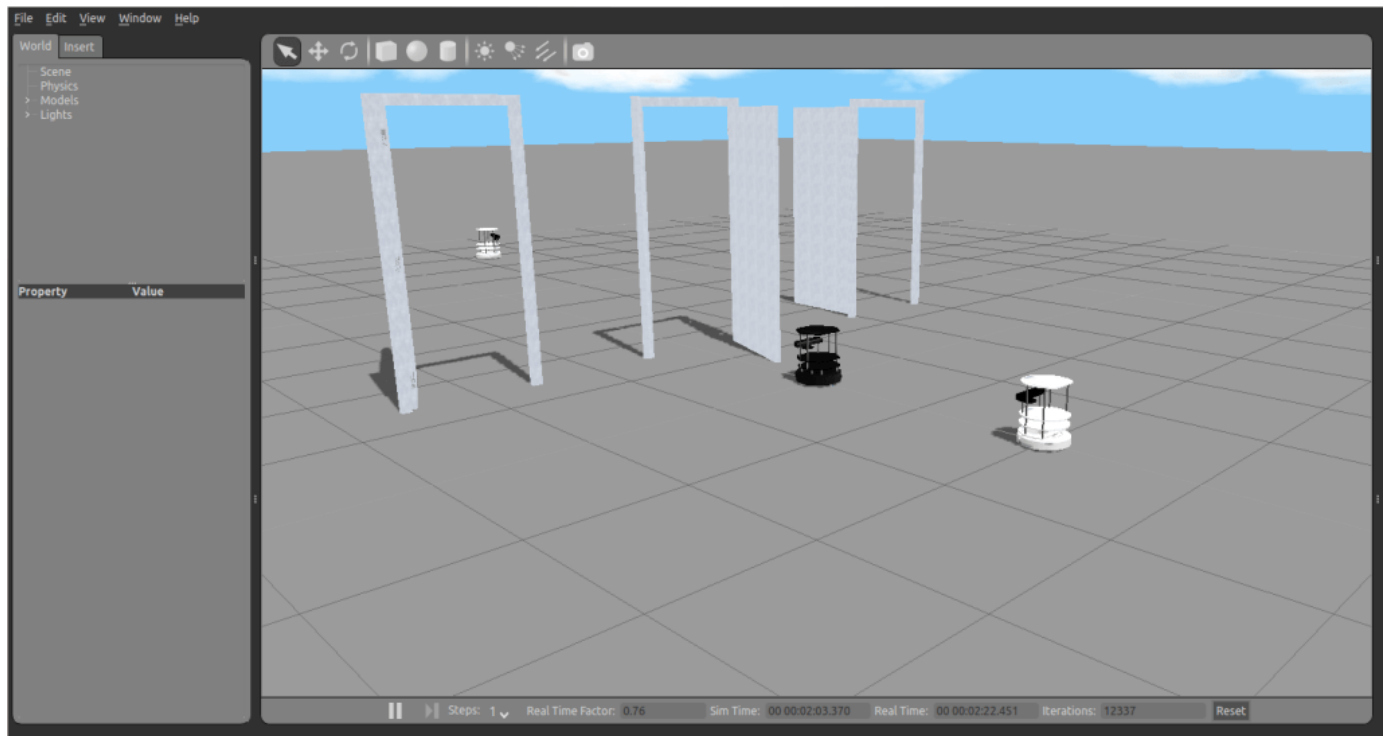
Apply a force to the base in the y direction. See that the base barely moves. The force is acting perpendicular to the wheel orientation.

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]);
```
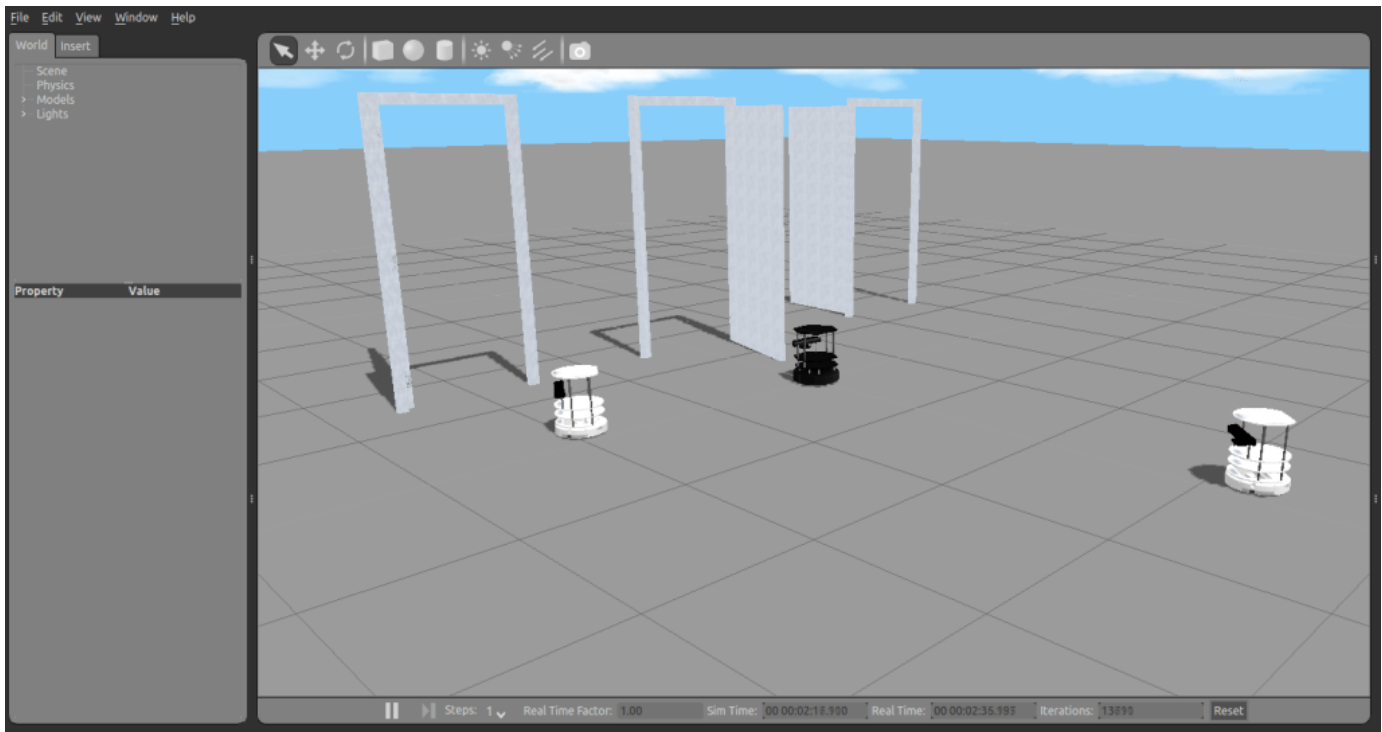
Add Bouncing Balls

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:
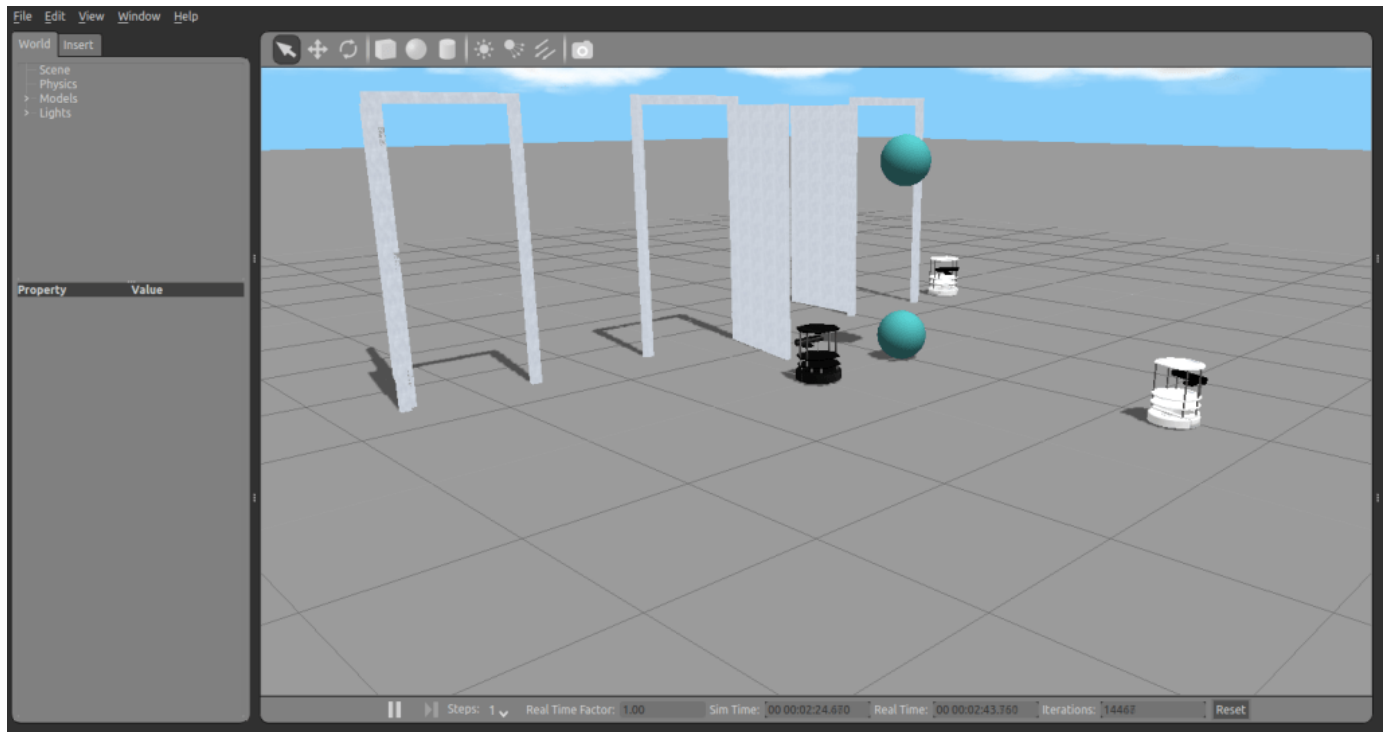
Remove Models and Shut Down

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 1-231

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 1-236 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 Simulated TurtleBot" on page 1-206, "Add, Build, and Remove Objects in Gazebo" on page 1-214, "Apply Forces and Torques in Gazebo" on page 1-221

Connect to Gazebo

On your Linux® machine, start Gazebo. If you are using the virtual machine from "Get Started with Gazebo and Simulated TurtleBot" on page 1-206, 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://192.168.178.133:11311")
```

```
Initializing global node /matlab_global_node_07643 with NodeURI http://192.168.178.1:57635/
```

```
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]);
```
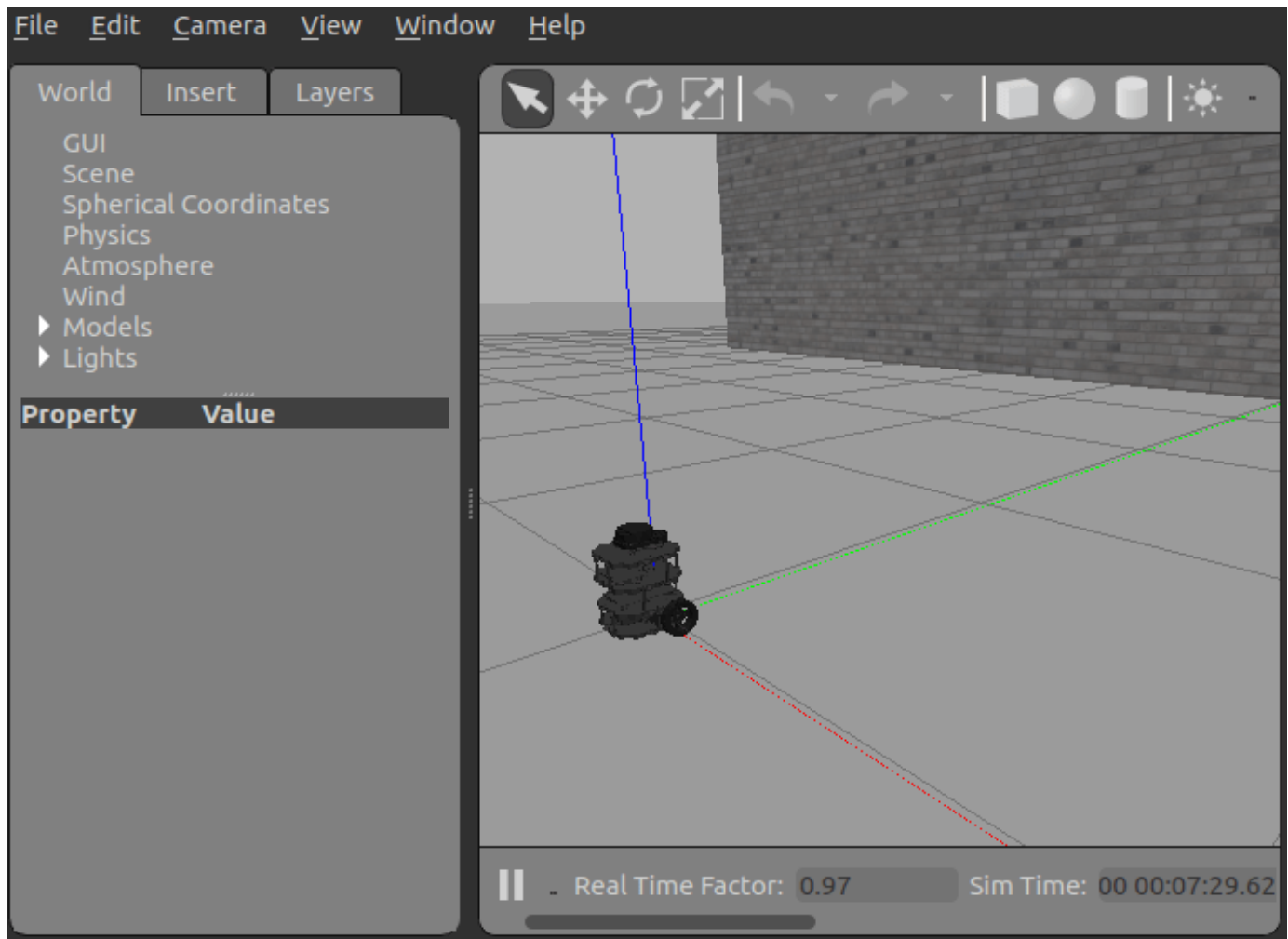
Start TurtleBot Obstacle Avoidance

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","DataFormat","struct");
velmsg = rosmessage(robot);
```

Subscribe to the laser scan topic.

```
timerHandles.sub = rossubscriber("/scan","DataFormat","struct");
```

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.
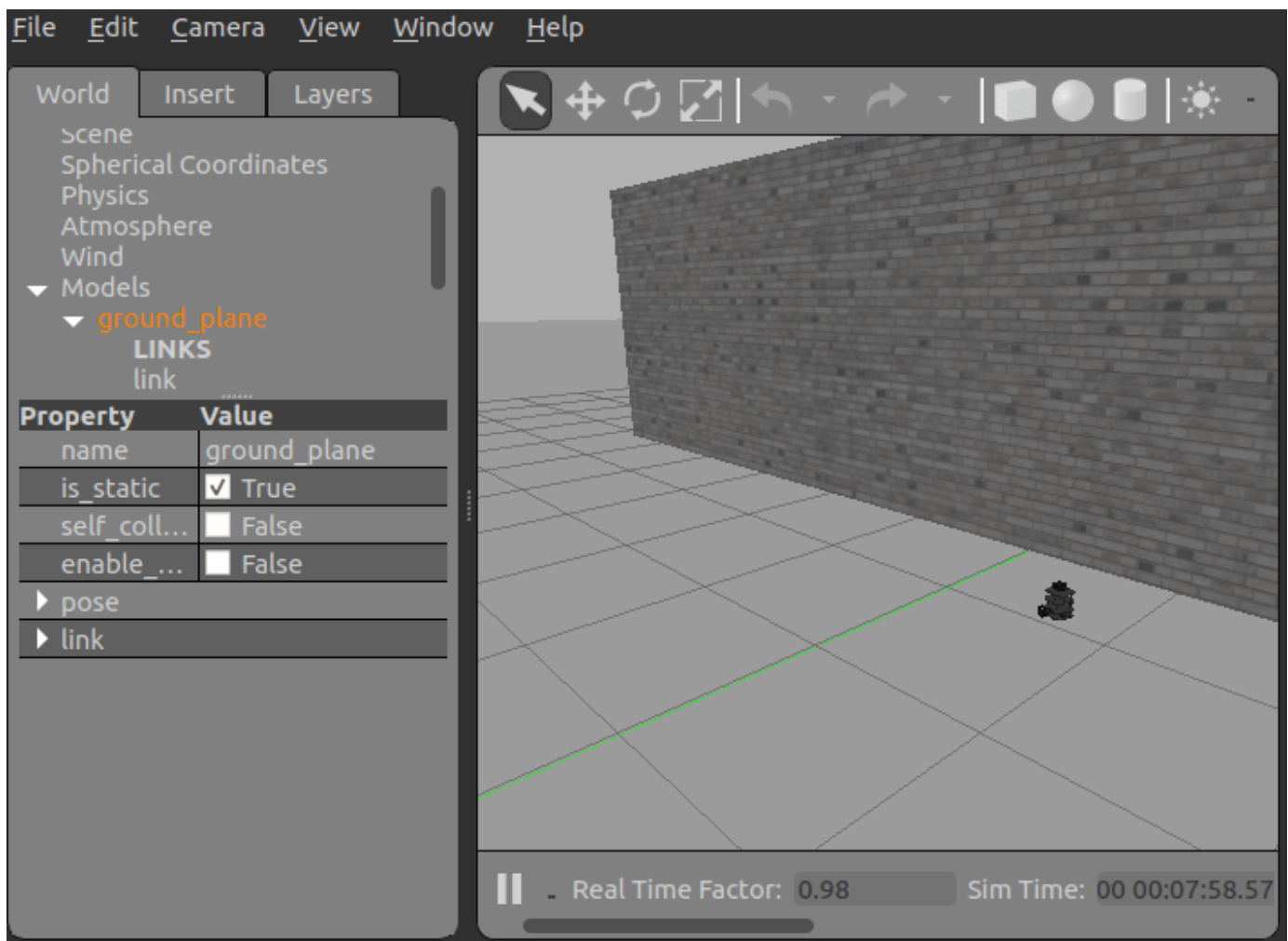
Note: If the TurtleBot crashes into the wall, the laser scan is probably not being published through Gazebo. Restart your Gazebo session and try again.

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
```
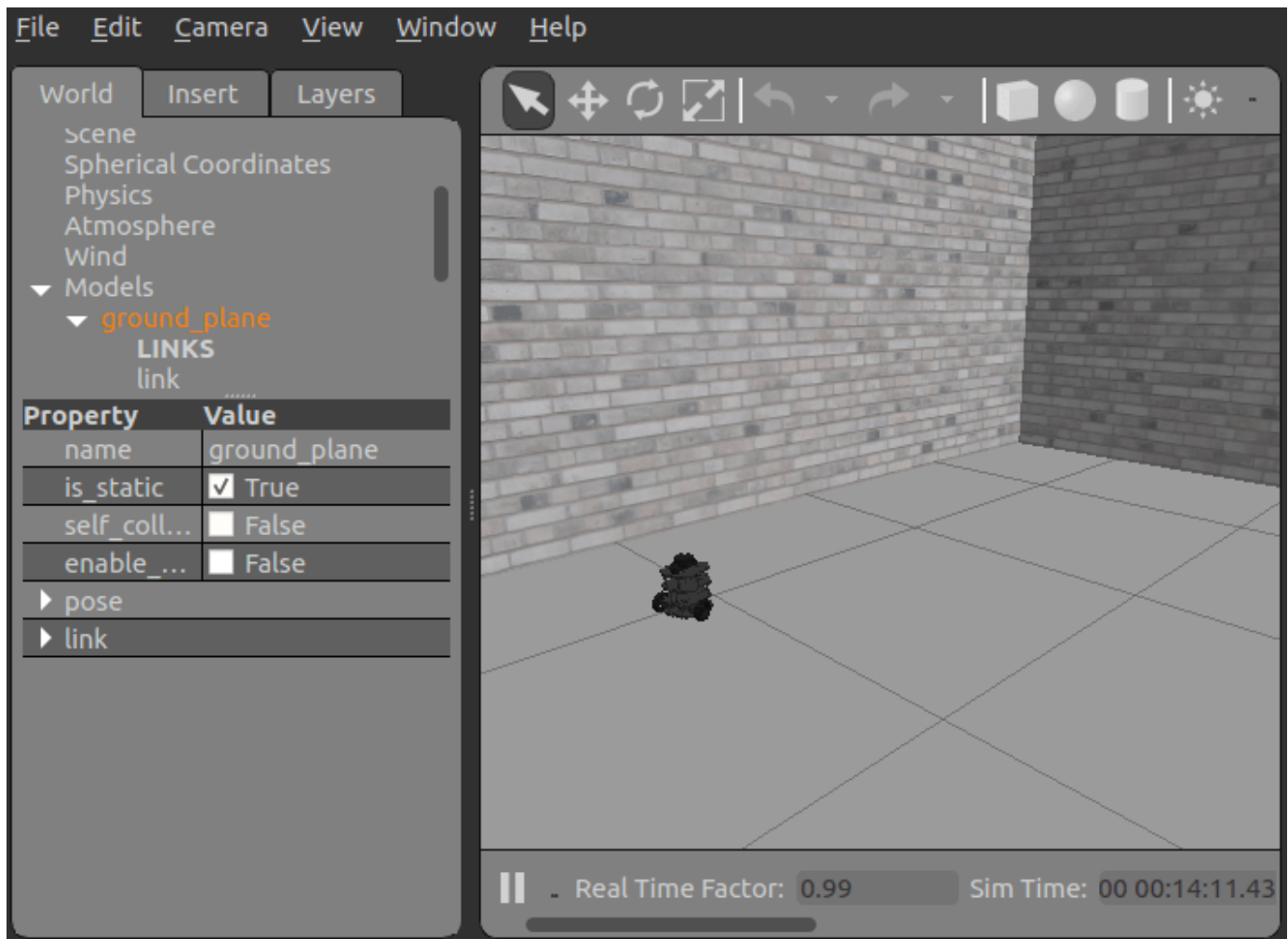

Remove Models and Shut Down

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_07643 with NodeURI http://192.168.178.1:57635/

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 Simulated TurtleBot" on page 1-206 or "Get Started with a Real TurtleBot" on page 1-117

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 1-117, the robot is running. If you are using a TurtleBot in simulation and followed the setup steps in "Get Started with Gazebo and Simulated TurtleBot" on page 1-206, 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://192.168.178.133:11311";
rosinit(ipaddress)
```

```
Initializing global node /matlab_global_node_88195 with NodeURI http://192.168.178.1:59934/
```

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. Use the publisher with structure format messages for better performance.

```
robotCmd = rospublisher("/cmd_vel","DataFormat","struct") ;
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)
```
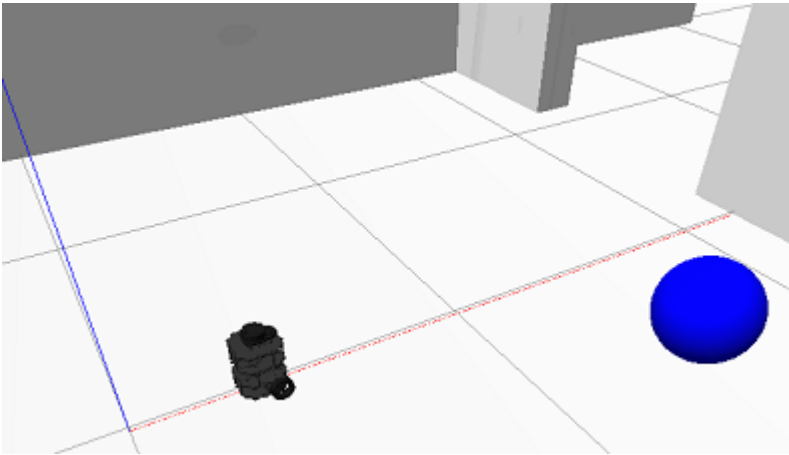

To view the type of the message published by the velocity topic, execute the following:

```
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_88195 (http://192.168.178.1:59934/)

Subscribers:
* /gazebo (http://192.168.178.133:39227/)
```

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","DataFormat","struct");
```

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

   -2.6176    1.0007   -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 = 0.1556
```

Receive Lidar Data

Subscribe to the lidar topic:

```
lidarSub = rossubscriber("/scan","DataFormat","struct");
```

After subscribing to the lidar topic, wait for the data and then display it with rosPlot.

```
scanMsg = receive(lidarSub);
figure
rosPlot(scanMsg)
```

Laser Scan
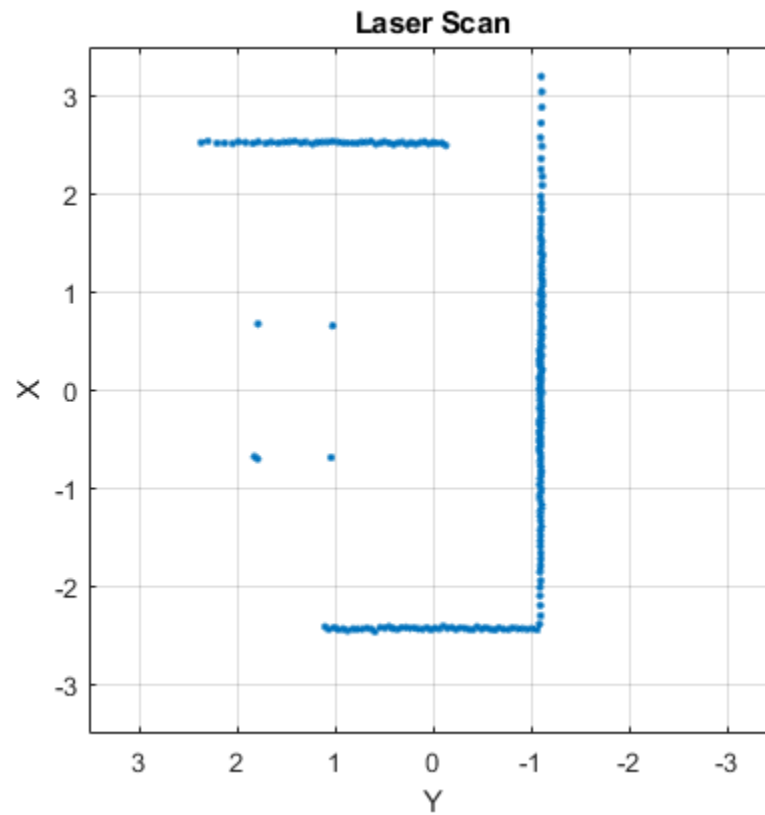
To continuously display updating lidar scans while the robot turns for a short duration, use the following while loop:

```
velMsg.Angular.Z = velocity;
send(robotCmd,velMsg)
tic
while toc < 20
  scanMsg = receive(lidarSub);
  rosPlot(scanMsg)
end
```

Laser Scan
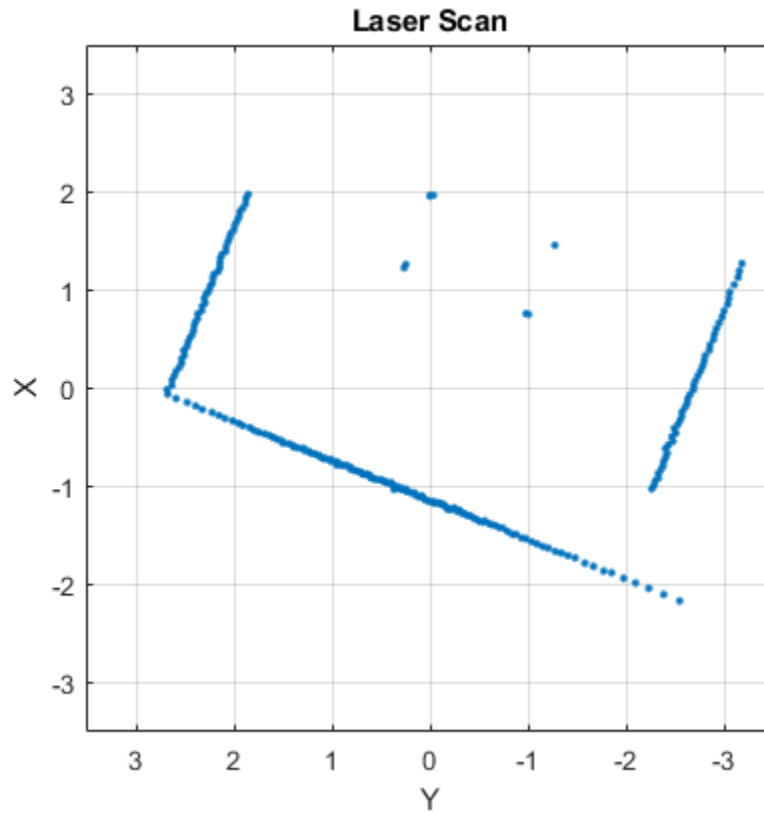

```
velMsg.Angular.Z = 0;
send(robotCmd,velMsg)
```

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_88195 with NodeURI http://192.168.178.1:59934/

Next Steps

Refer to the next example: "Explore Basic Behavior of the TurtleBot" on page 1-241

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 1-236

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 Simulated TurtleBot" on page 1-206 or "Get Started with a Real TurtleBot" on page 1-117 for the startup procedure. This example uses the Gazebo-simulated Turtlebot.

In the downloaded virtual machine, click the **Gazebo House** shortcut to start up the world.

Initialize ROS. Connect to the TurtleBot by replacing `ipaddress` with the IP address of the TurtleBot.

```
ipaddress = "http://192.168.178.133:11311";
rosinit(ipaddress)
```

```
Initializing global node /matlab_global_node_05319 with NodeURI http://192.168.178.1:51558/
```

Create a publisher for the robot's velocity and create a message for that topic. Communicate using structure format messages for improved performance.

```
robot = rospublisher("/cmd_vel","DataFormat","struct");
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","DataFormat","struct");
```

Wait for one laser scan message to arrive and then display it.

```
scan = receive(laser,3)
```

```
scan = struct with fields:
      MessageType: 'sensor_msgs/LaserScan'
           Header: [1×1 struct]
         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]
```

```
figure
rosPlot(scan);
```
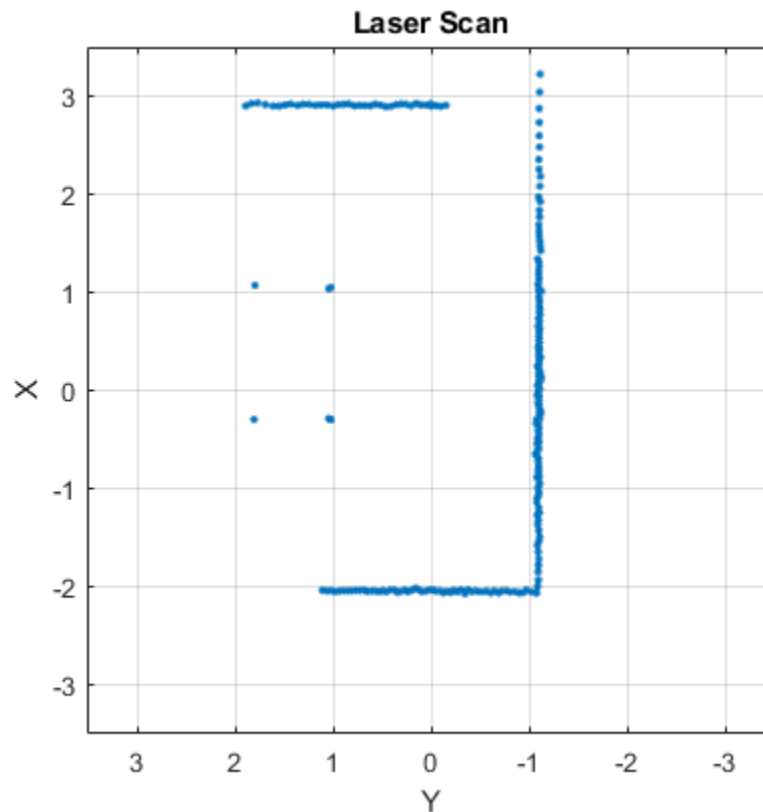

If you see an error, it is possible that the laser scan topic is not receiving any data. If you are running in simulation, try restarting Gazebo. If you are using hardware, make sure that you started the lidar and camera properly.

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);
```

```
    rosPlot(scan);
end
```
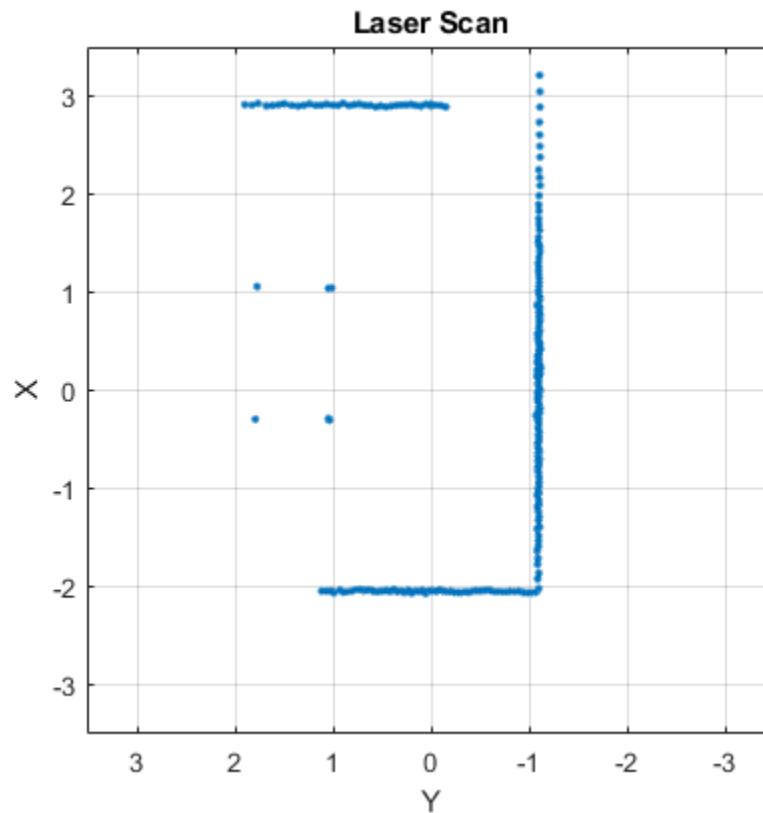

Laser Scan

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);
    rosPlot(scan);
    data = rosReadCartesian(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
```
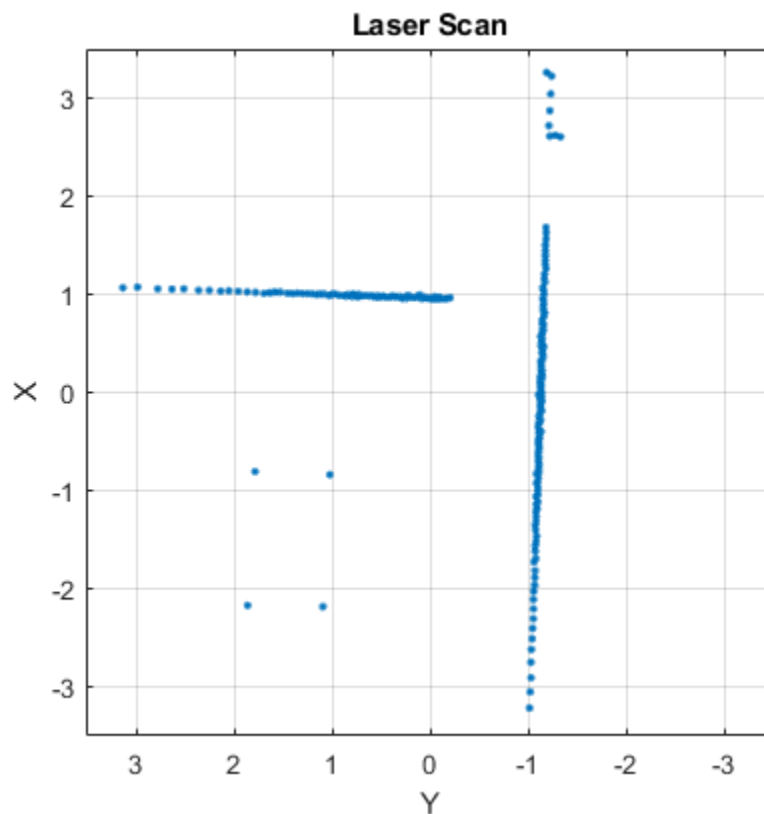

Disconnect from the Robot

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_05319 with NodeURI http://192.168.178.1:51558/

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:
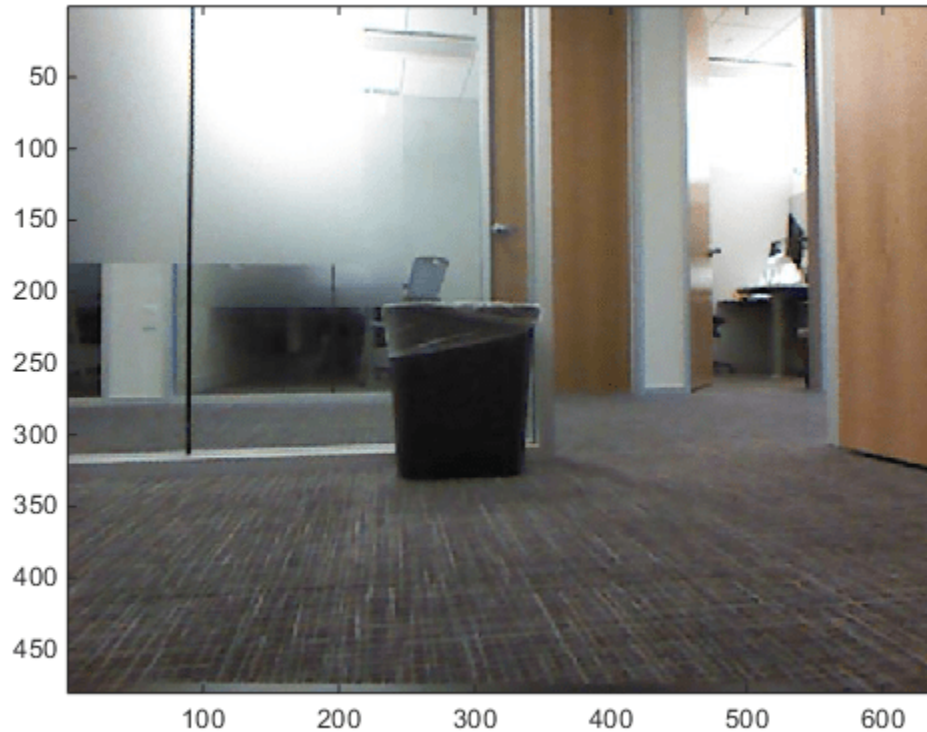
Here is the corresponding laser scan:
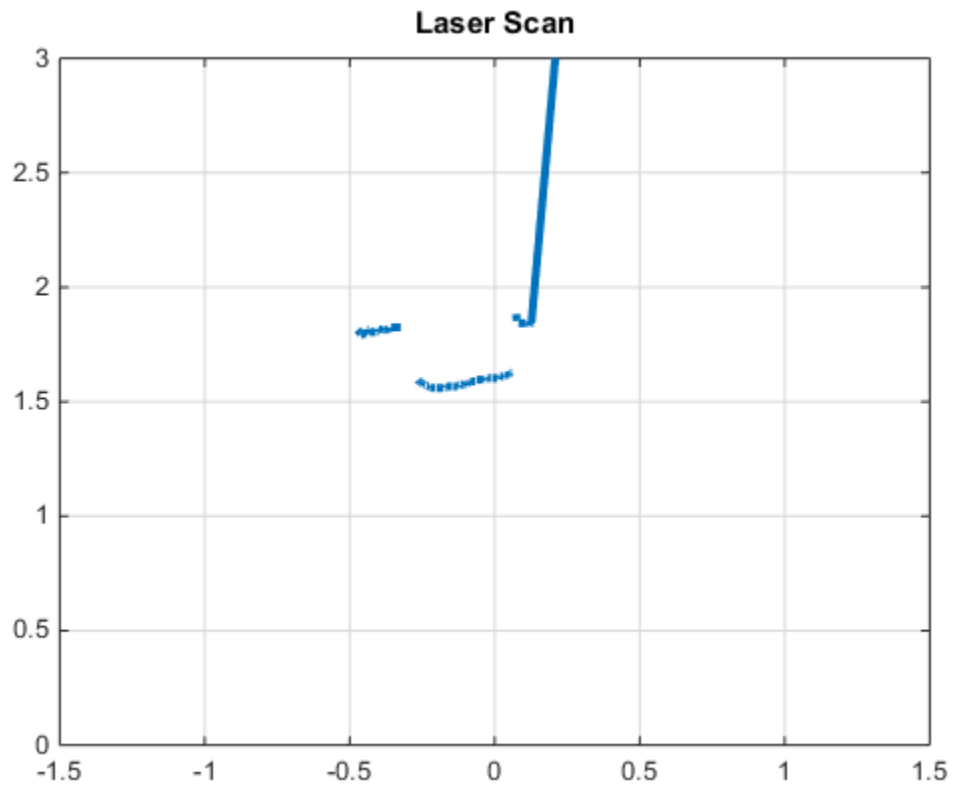
The trash can is visible, but you cannot see the glass wall. When you use the TurtleBot in areas with windows or walls that the TurtleBot might not be able to detect, be aware of the limitations of the laser scan.

Next Steps

Refer to the next example: "Control the TurtleBot with Teleoperation" on page 1-247

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 1-117 or "Get Started with Gazebo and Simulated TurtleBot" on page 1-206)

Prerequisites: "Communicate with the TurtleBot" on page 1-236, "Explore Basic Behavior of the TurtleBot" on page 1-241

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 Simulated TurtleBot" on page 1-206 or "Get Started with a Real TurtleBot" on page 1-117 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://192.168.178.133:11311";
rosinit(ipaddress)
```

```
Initializing global node /matlab_global_node_59490 with NodeURI http://192.168.178.1:55912/
```

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. Communicate using structure message format for better performance.

```
handles.odomSub = rossubscriber("/odom","BufferSize",25,"DataFormat","struct");
receive(handles.odomSub,3);
```

```
handles.laserSub = rossubscriber("/scan","BufferSize",5,"DataFormat","struct");
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,"DataFormat","struct");
```

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
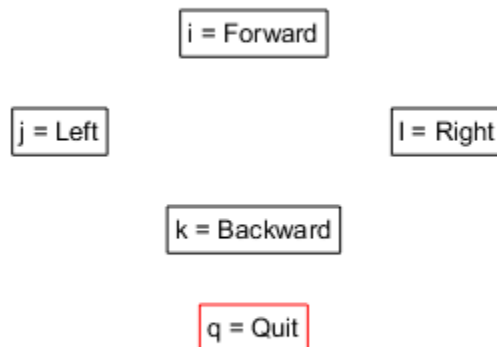
Keep this figure in scope to give commands
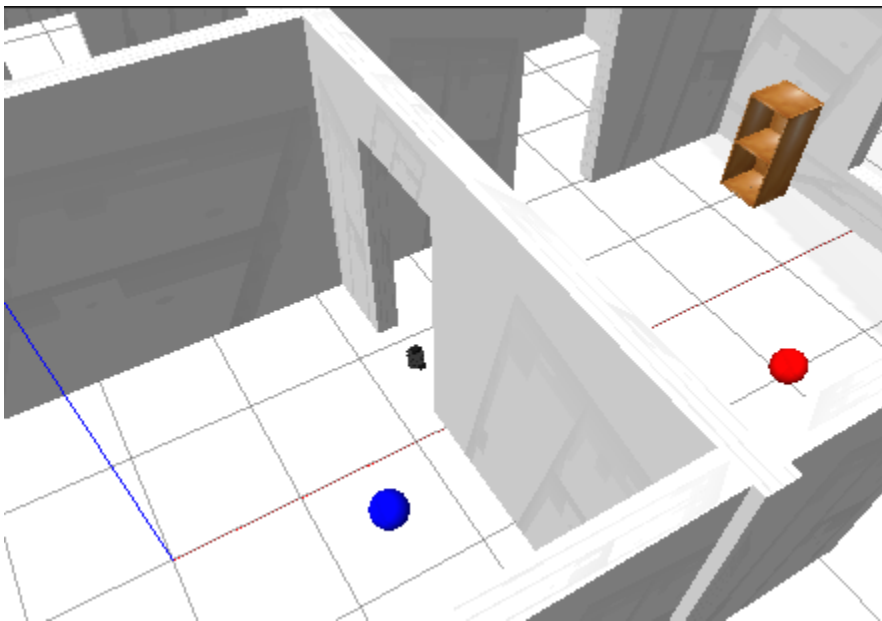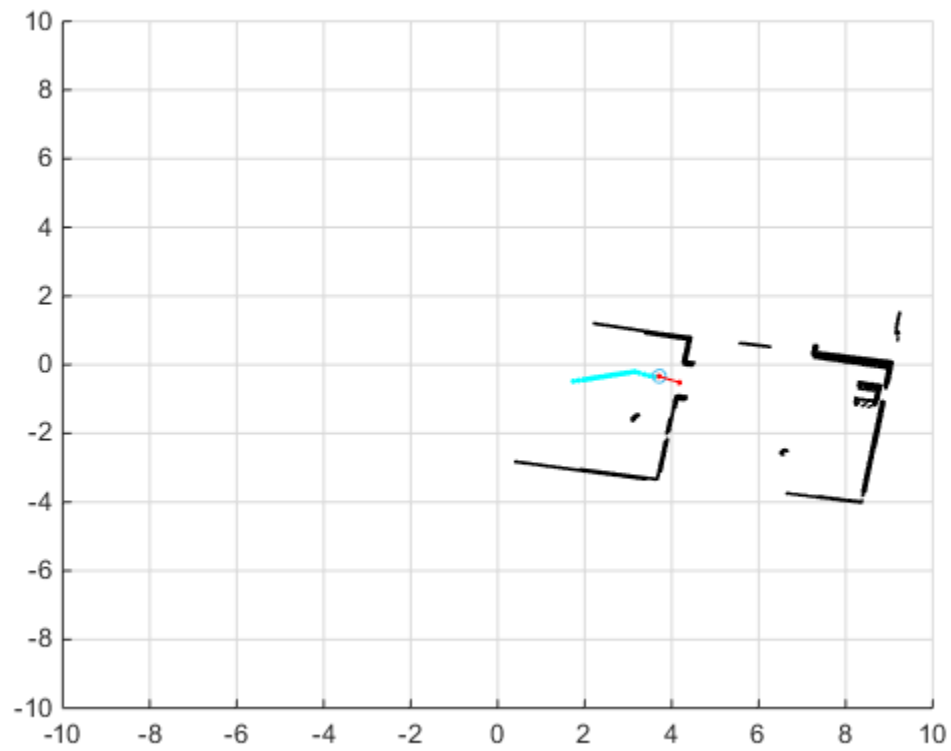
If you move the TurtleBot too quickly, the obstacle plotting can become messy because of relative inaccuracies in the odometry topic at high speeds. Here is an example of a messy world plot:
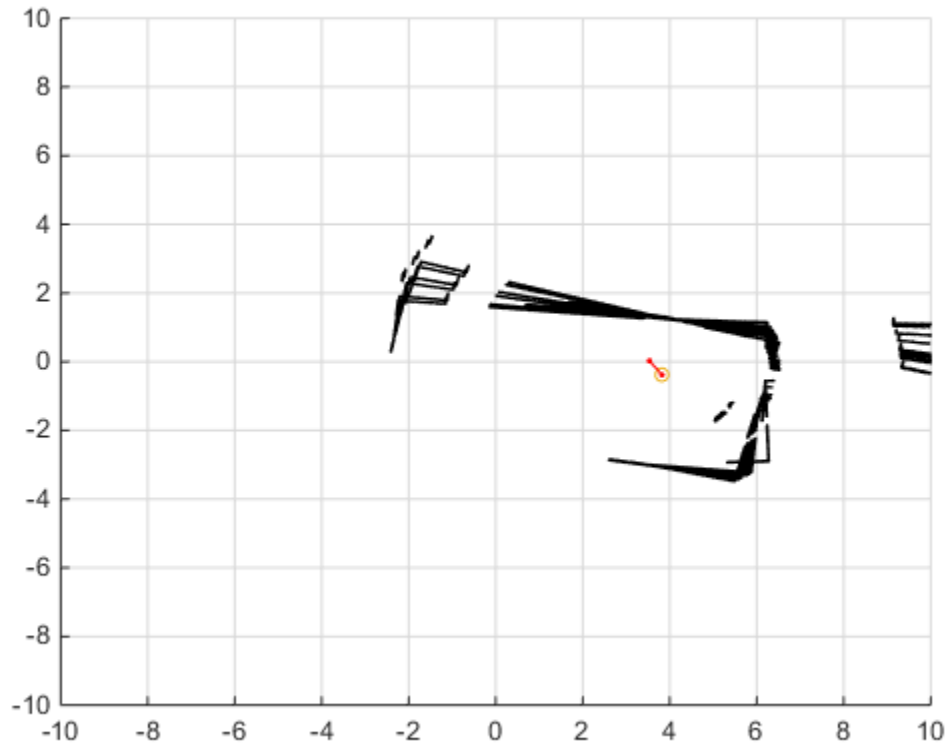
A sample plot of a real TurtleBot moving around an office space is shown:
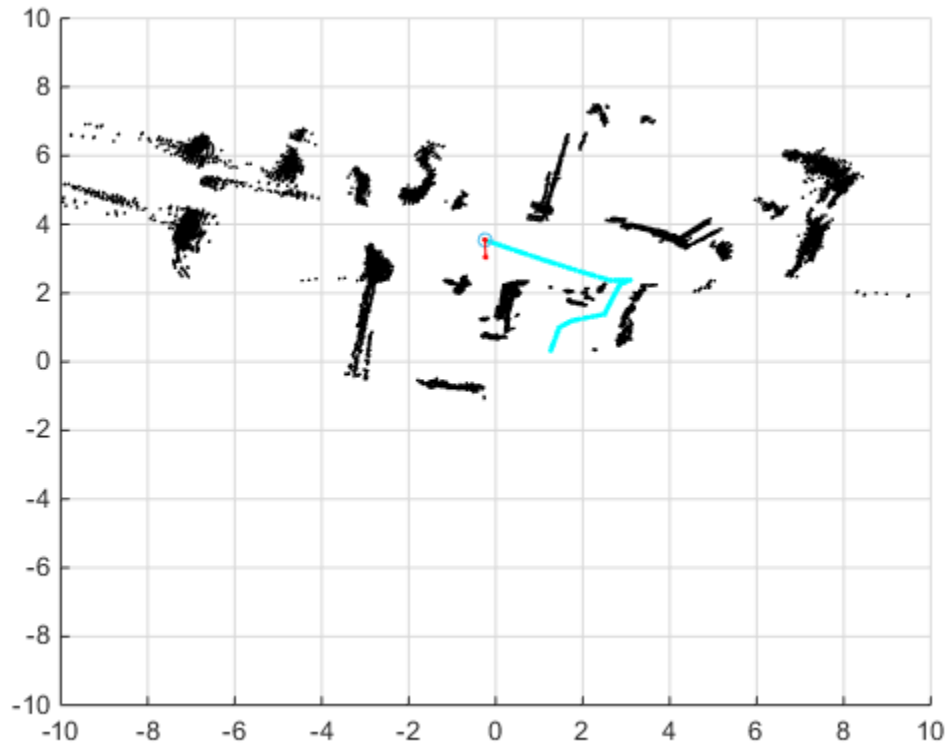
Disconnect from the Robot

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_59490 with NodeURI http://192.168.178.1:55912/

Next Steps

- Refer to the next example: "Obstacle Avoidance with TurtleBot and VFH" on page 1-252

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 Simulated TurtleBot" on page 1-206 to install and set up a TurtleBot in Gazebo.

Connect to the TurtleBot using the IP address obtained from setup.

```
rosinit("192.168.178.133",11311)
```

Initializing global node /matlab_global_node_05320 with NodeURI http://192.168.178.1:63605/

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. Communicate using structure format messages for better performance.

```
laserSub = rossubscriber("/scan","DataFormat","struct");
[velPub, velMsg] = rospublisher("/cmd_vel","DataFormat","struct");
```

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 and create a lidarScan object
scanMsg = receive(laserSub);
scan = rosReadLidarScan(scanMsg);

% 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_05320 with NodeURI http://192.168.178.1:63605/

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 1-236, "Explore Basic Behavior of the TurtleBot" on page 1-241, "Control the TurtleBot with Teleoperation" on page 1-247, "Obstacle Avoidance with TurtleBot and VFH" on page 1-252

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 Simulated TurtleBot" on page 1-206 or "Get Started with a Real TurtleBot" on page 1-117 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 = "192.168.178.133";
rosinit(ipaddress,11311)
```

Initializing global node /matlab_global_node_76155 with NodeURI http://192.168.178.1:49316/

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. Communicate using messages in structure format for better performance.

```
handles.colorImgSub = exampleHelperTurtleBotEnableColorCamera;
```

Successfully Enabled Camera (raw image)

```matlab
useHardware = exampleHelperTurtleBotIsPhysicalRobot;
if useHardware
    handles.cliffSub = rossubscriber("/mobile_base/events/cliff",...
        "BufferSize",5,"DataFormat","struct");
    handles.bumpSub = rossubscriber("/mobile_base/sensors/bumper_pointcloud",...
        "BufferSize",5,"DataFormat","struct");
    handles.soundPub = rospublisher("/mobile_base/commands/sound", "kobuki_msgs/Sound",...
        "DataFormat","struct");
    handles.velPub = rospublisher("/mobile_base/commands/velocity",...
        "DataFormat","struct");
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","DataFormat","struct");
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.

```matlab
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:

```matlab
latestImg = rosReadImage(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.

```matlab
exampleHelperTurtleBotPlotObject(latestImg,ball,c);
```
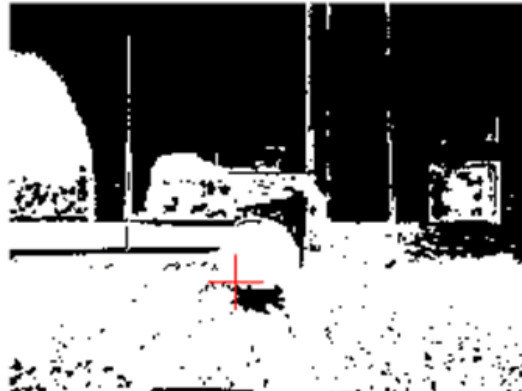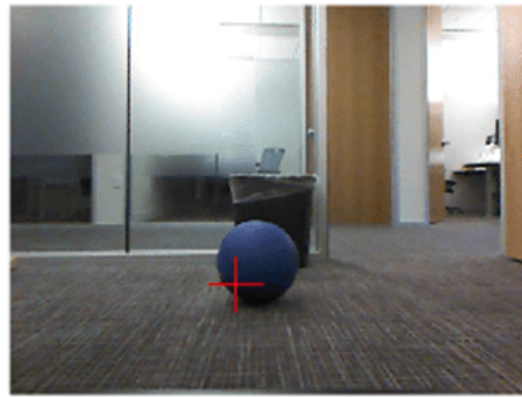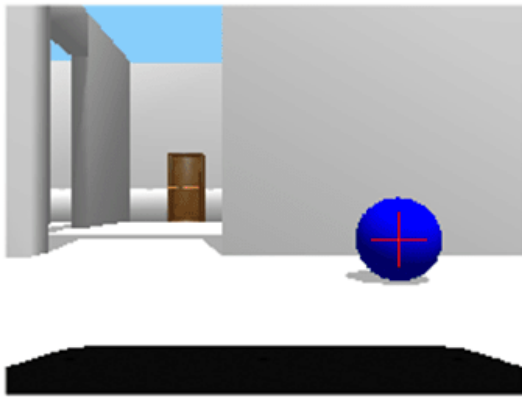
If the ball is not found, try increasing or decreasing `blueBallParams.blueMax` and `blueBallParams.darkMin`. View the plot again until the ball is found. This method is a good way to fine tune the ball-finding algorithm before using the controller.

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 = rosReadImage(handles.colorImgSub.LatestMessage);
[c,~,ball] = exampleHelperTurtleBotFindBlueBall(latestImg,blueBallParams,useHardware);
```

Use this example helper to display the figures.

```
exampleHelperTurtleBotPlotObject(latestImg,ball,c);
```
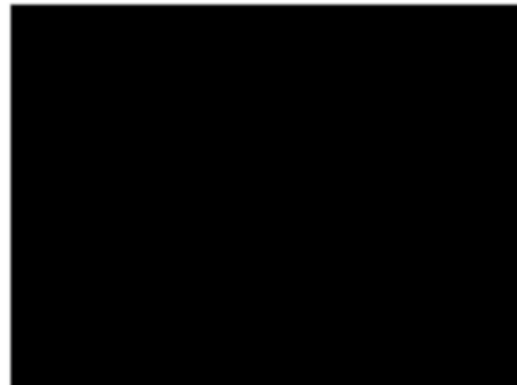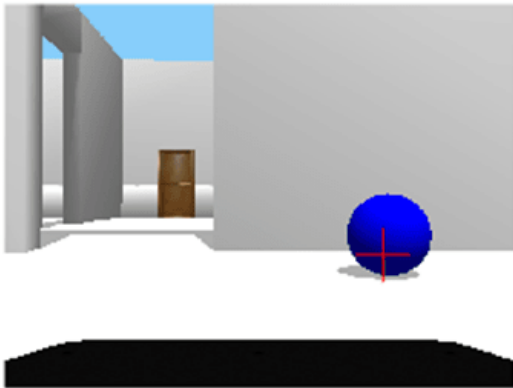
Now the parameters are too restrictive. Part of the ball does not even show up in the Gazebo image, and you see nothing in the real image. If you tune the parameters further you can find a middle ground. In Gazebo, the following parameters should work well. With hardware, ambient lighting might require 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 = rosReadImage(handles.colorImgSub.LatestMessage);
[c,~,ball] = exampleHelperTurtleBotFindBlueBall(latestImg,blueBallParams, useHardware);
```

Use this example helper to display the figures.

```
exampleHelperTurtleBotPlotObject(latestImg,ball,c);
```
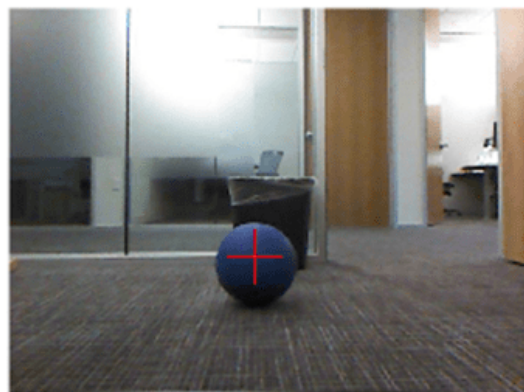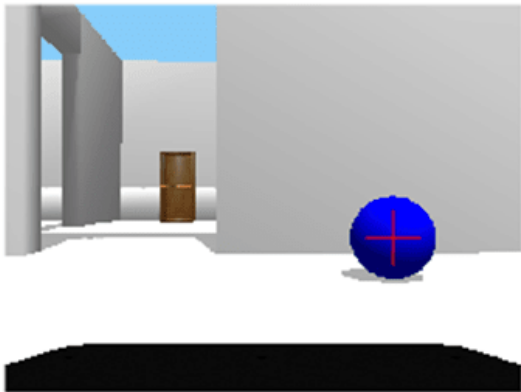
Tuning the color thresholds is challenging when compared to tuning them in a simulated environment like Gazebo.

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 1-214.

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
```
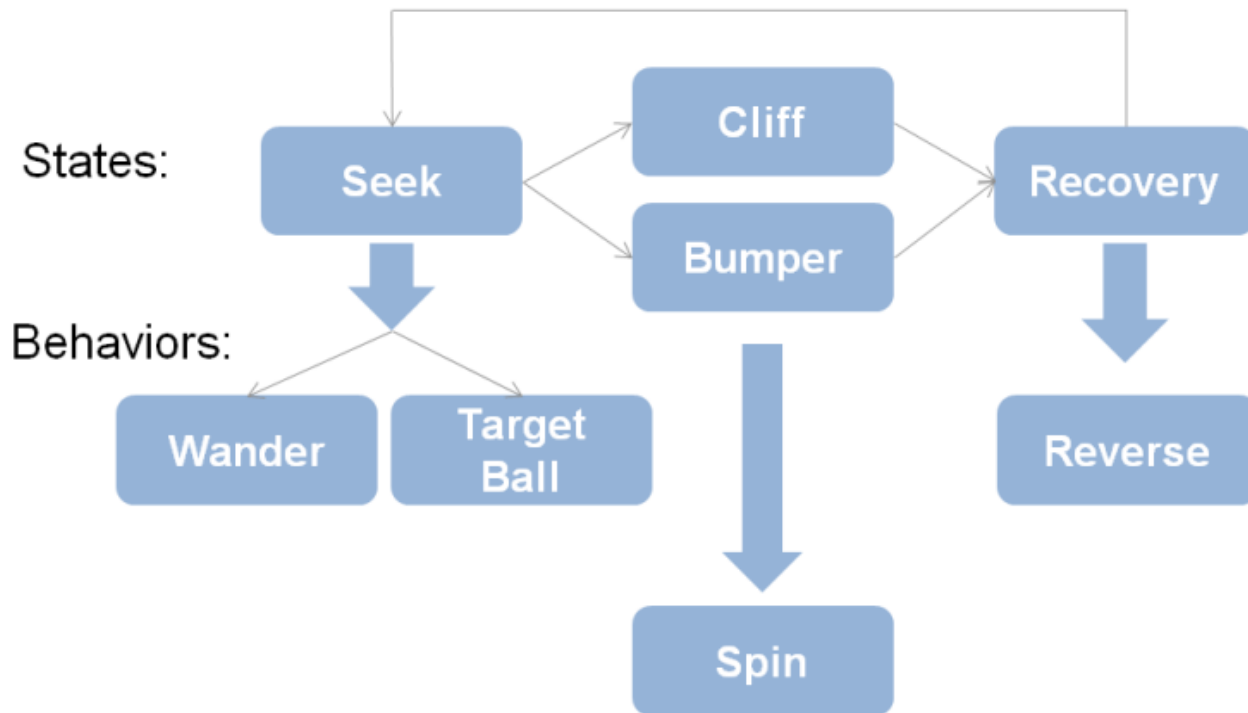
You can add or remove cases from the state machine. If you want to change the state names, use the `ExampleHelperTurtleBotStates` class.

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

This example shows how to set up ROS 2 within MATLAB, and get information about ROS 2 network and ROS 2 messages.

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 Service (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.

To learn about ROS, see "Get Started with ROS" on page 1-39.

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 Service (DDS) which makes it possible to connect multiple nodes across distributed network. For more information, see "Switching Between ROS Middleware Implementations" on page 2-172.

- *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 2-23.

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.
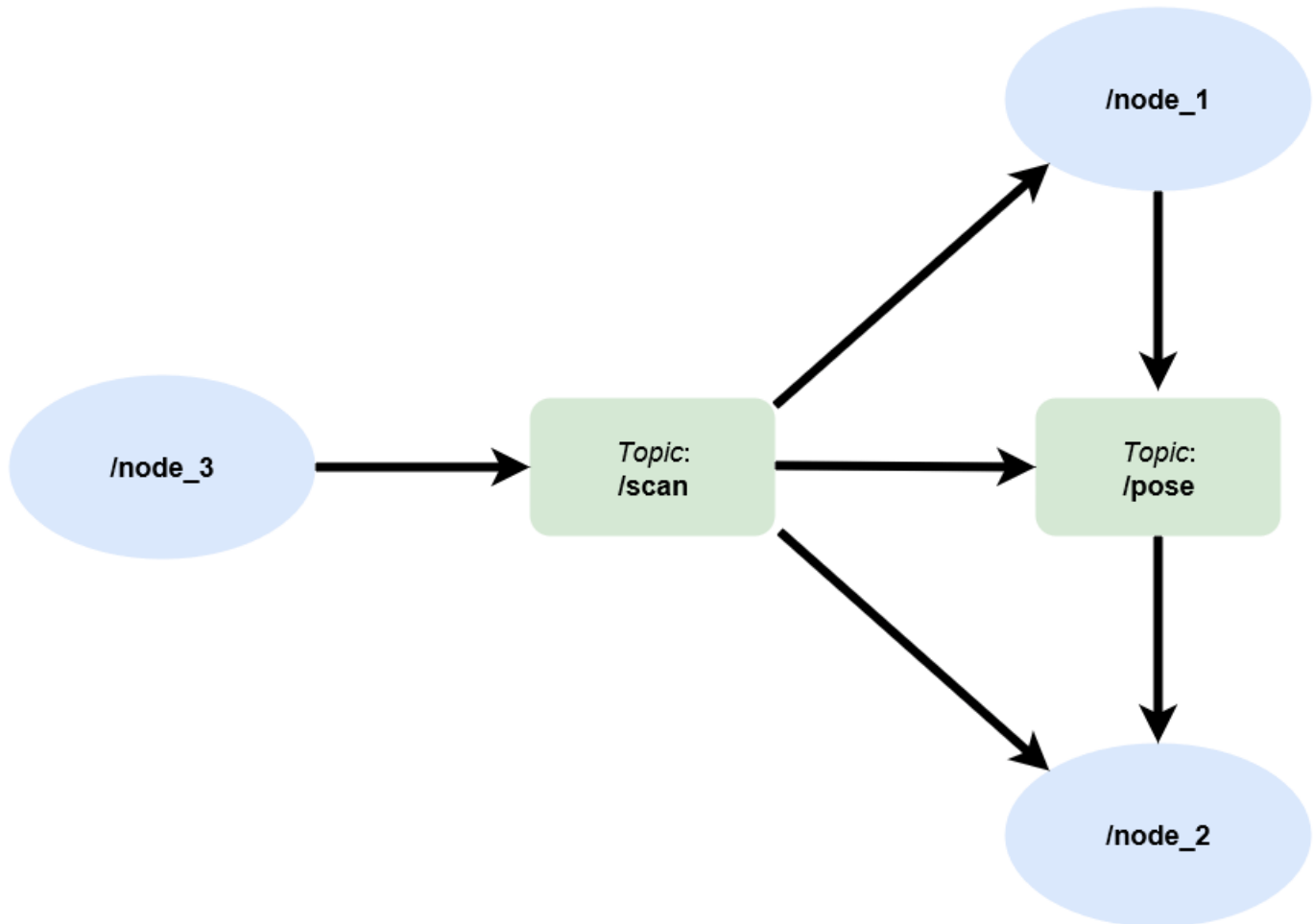
Topics and Quality of Service Policies

Run the command `ros2 topic list` to see available topics in the ROS 2 network. This command returns three active topics: `/pose`, `/parameter_events,` and `/scan`. The topic `/parameter_events` is a global topic which is always present in the ROS 2 network. The nodes use the `/paramater_events` topic to monitor or change parameters in the network. You created the `/scan` and `/pose` topics as part of the sample network.

```
ros2 topic list
```

```
/parameter_events
/pose
/rosout
/scan
```

Each topic is associated with a *message type*. Run the command `ros2 topic list -t` to see the message type of each topic.

```
ros2 topic list -t
```

```
          Topic                          MessageType
    _____        _____
```

```
{'/parameter_events'}    {'rcl_interfaces/ParameterEvent'}
{'/pose'            }    {'geometry_msgs/Twist'           }
{'/rosout'          }    {'rcl_interfaces/Log'            }
{'/scan'            }    {'sensor_msgs/LaserScan'         }
```

The Quality of Service (QoS) policy options change the way the publishers and subscribers handle and exchange messages.

1. Use the `History` and `Depth` QoS policies to determine the behavior of communication objects based on how you want to place messages in the processing queue. Specify `History` as one of these options.

- `"keeplast"` — Drop messages and retain only the most up-to-date information.
- `"keepall"` — Keep all the messages received in the queue until they are processed.

2. Use the `Reliability` QoS policy to guarantee delivery of the messages to the subscriber. Specify `Reliability` as one of these options.

- `"reliable"` — Ensure that the publisher continuously sends the message to the subscriber until the subscriber confirms receipt of the message.
- `"besteffort"` — Allow the publisher to send the message only once.

3. Use the `Durability` QoS policy with the `Depth` input to control the persistence of messages for late-joining connections. Specify `Durability` as one of these options.

- `"transientlocal"` — The publisher sends the persisted messages to the subscriber if the subscriber joins the network after the publisher initially sends the messages.
- `"volatile"` — The publisher does not persist messages after sending them. Subscribers do not request persisted messages from publishers.

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 2-13.

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.

# This is semantically different than a point.
# A vector is always anchored at the origin.
# When a transform is applied to a vector, only the rotational component is applied.
```

```
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
```

DDS and RMW Implementations

You can switch between RMW Implementations for using the data distribution service (DDS), when you work with ROS 2 in MATLAB® and Simulink®. To configure the required RMW Implementation, follow these steps.

1. Under the **Home** tab in the **Environment** section of the MATLAB toolstrip, open **Preferences.**

2. Configure the Python™ environment in the **ROS Toolbox Preferences** dialog box and select an implementation from the **ROS Middleware (RMW) Implementation** dropdown list. The default implementation is `rmw_fastrtps_cpp`.
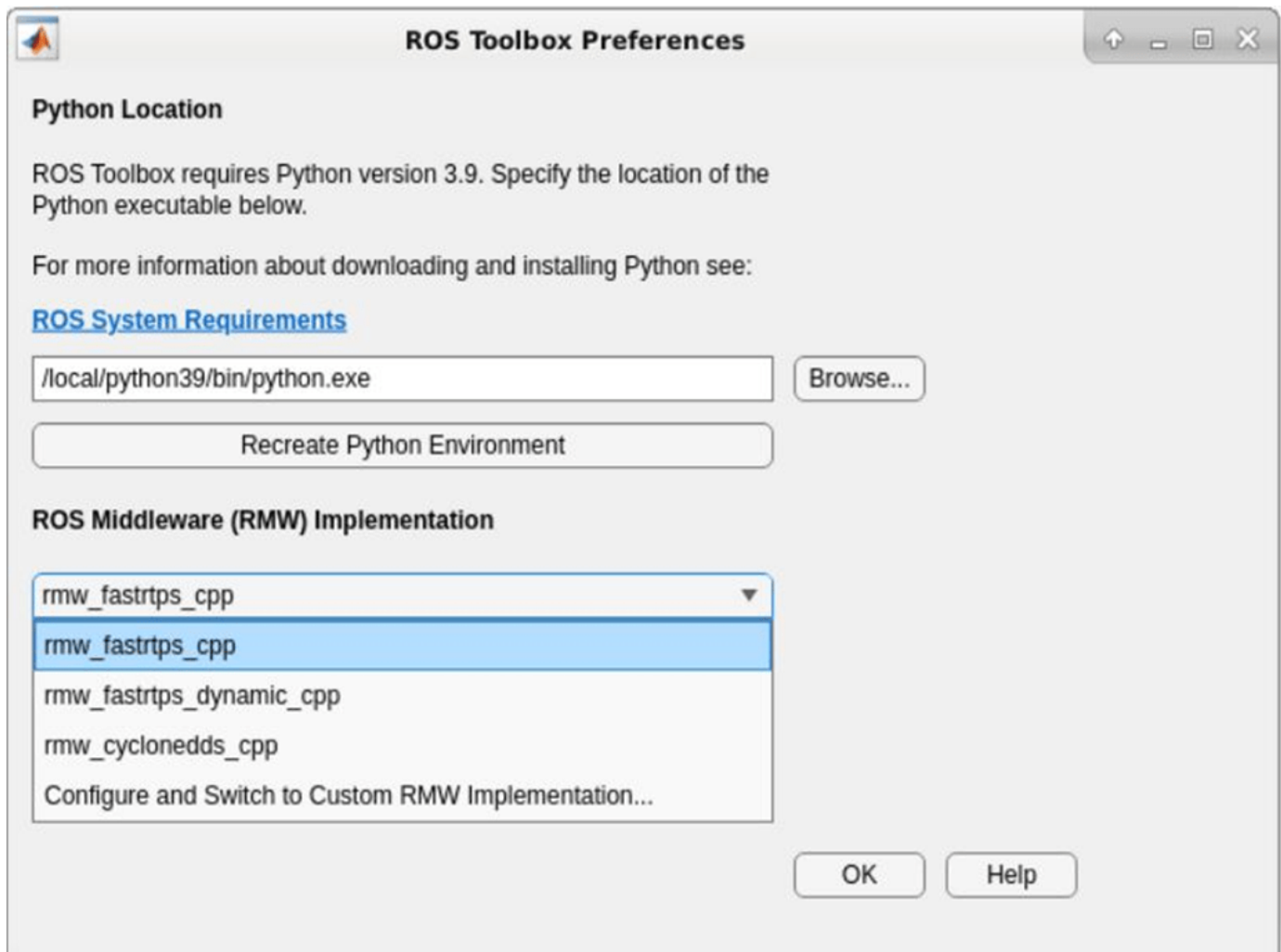
3. Switch to a custom RMW implementation by clicking **Configure and Switch to Custom RMW Implementation**, which launches the **ROS Middleware Configuration** dialogue box.

4. Install and build the custom RMW Implementation package to validate the creation of a ROS 2 node with the selected custom RMW implementation.

Next Steps

- "Connect to a ROS 2 Network" on page 2-8

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 1-44.

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 2-23.

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.
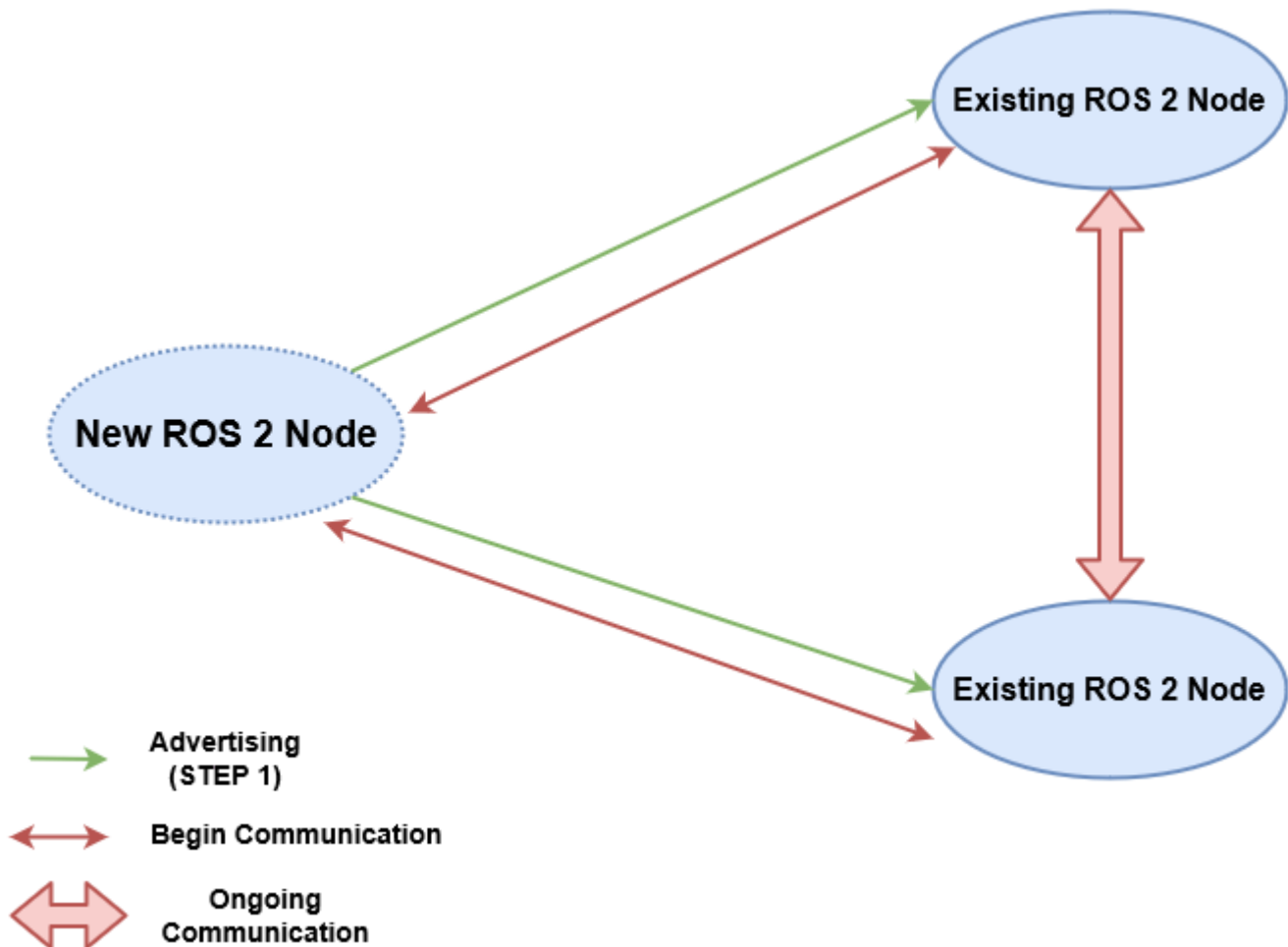
The new ROS 2 node sends its advertisement to the existing nodes. The existing nodes respond to the advertisement and then set up for ongoing communication.

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.

OUTSIDE SUBNET

*DEFAULT_
FASTRTPS_
PROFILE.xml*

SAME SUBNET

*DEFAULT_
FASTRTPS_
PROFILE.xml*
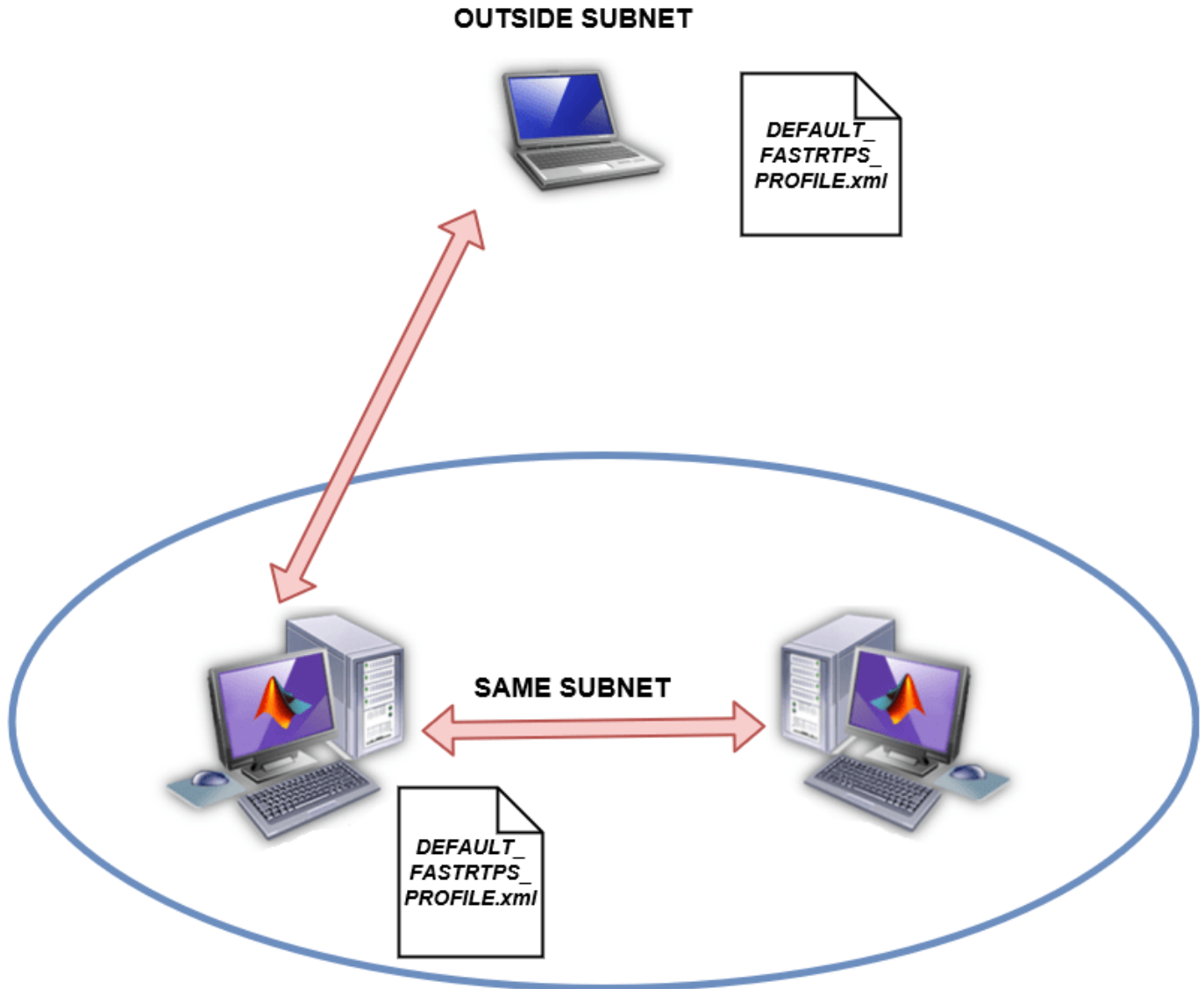
Next Steps

- "Exchange Data with ROS 2 Publishers and Subscribers" on page 2-19

Work with Basic ROS 2 Messages

This example examines various ways to create, inspect, and populate ROS 2 messages in MATLAB, that are commonly encountered in robotics applications.

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 2-19.

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:
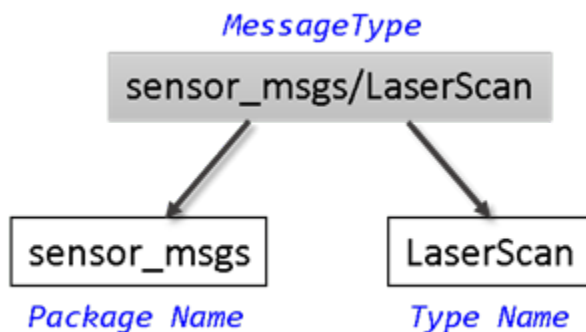
MATLAB® supports many ROS 2 message types that are commonly encountered in robotics applications. This example examines some of the ways to create, inspect, and populate ROS 2 messages in MATLAB.

Prerequisites: "Get Started with ROS 2" on page 2-2, "Connect to a ROS 2 Network" on page 2-8

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'}
   {'/pose'            }   {'geometry_msgs/Twist'          }
   {'/rosout'          }   {'rcl_interfaces/Log'           }
   {'/scan'            }   {'sensor_msgs/LaserScan'        }
```

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:
          MessageType: 'sensor_msgs/LaserScan'
               header: [1×1 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:
    MessageType: 'geometry_msgs/Twist'
         linear: [1×1 struct]
        angular: [1×1 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:
    MessageType: 'geometry_msgs/Vector3'
              x: 0.0206
              y: -0.0468
              z: -0.0223
```

```
poseData.angular
```

```
ans = struct with fields:
    MessageType: 'geometry_msgs/Vector3'
              x: -0.0454
              y: -0.0403
              z: 0.0323
```

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 = 0.0206
```

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:
    MessageType: 'geometry_msgs/Twist'
         linear: [1×1 struct]
        angular: [1×1 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:
    MessageType: 'geometry_msgs/Vector3'
              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.

```
tempMsgBlank = ros2message("sensor_msgs/Temperature");
tempMsgCopy = tempMsgBlank
```

```
tempMsgCopy = struct with fields:
    MessageType: 'sensor_msgs/Temperature'
         header: [1×1 struct]
    temperature: 0
       variance: 0
```

Modify the `temperature` property of tempMsg and notice that the contents of `tempMsgBlank` remain unchanged.

```
tempMsgCopy.temperature = 100
```

```
tempMsgCopy = struct with fields:
    MessageType: 'sensor_msgs/Temperature'
         header: [1×1 struct]
    temperature: 100
       variance: 0
```

```
tempMsgBlank
```

```
tempMsgBlank = struct with fields:
    MessageType: 'sensor_msgs/Temperature'
         header: [1×1 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])
```
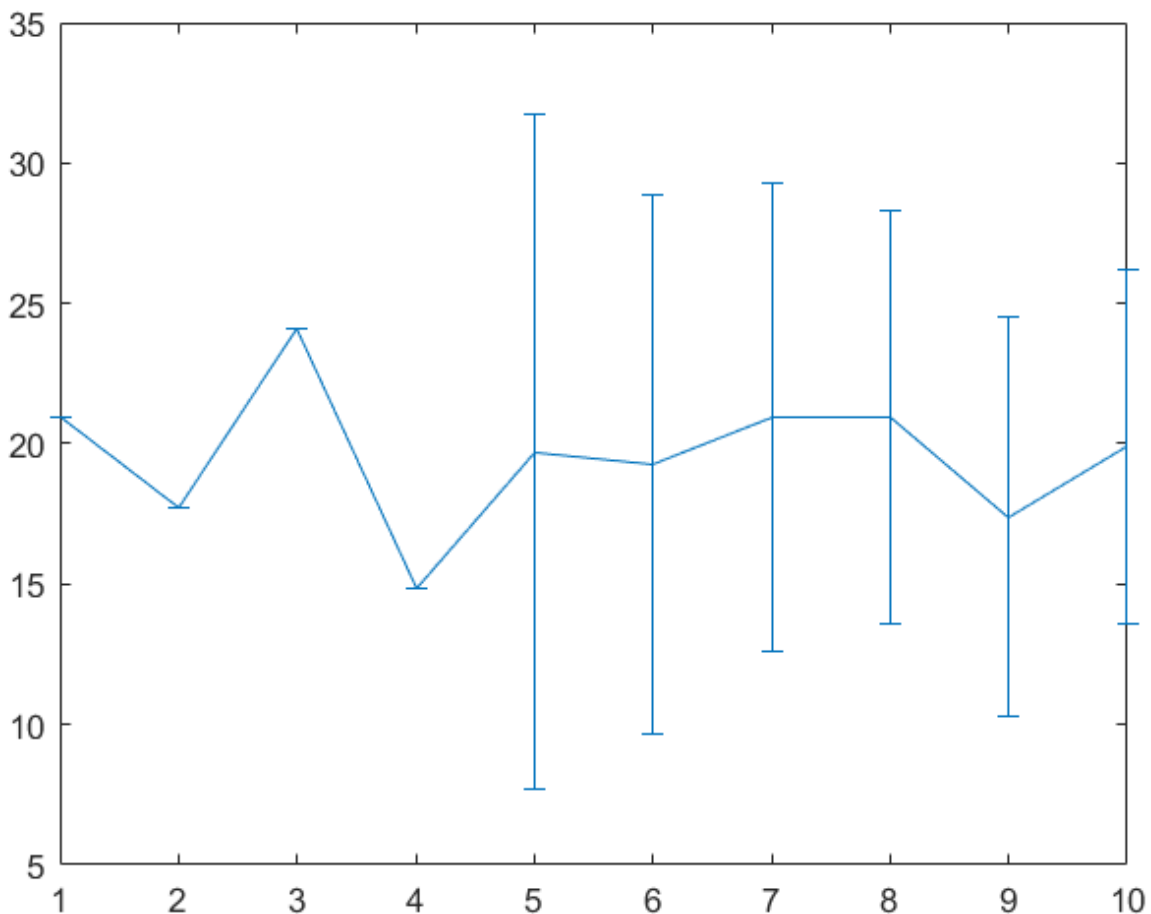
Save and Load Messages

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:
    MessageType: 'geometry_msgs/Twist'
         linear: [1×1 struct]
        angular: [1×1 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: [1×1 struct]
```

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

```
messageData.poseData

ans = struct with fields:
    MessageType: 'geometry_msgs/Twist'
         linear: [1×1 struct]
        angular: [1×1 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 2-19
- "Generate ROS 2 Custom Messages in MATLAB" on page 2-54

Exchange Data with ROS 2 Publishers and Subscribers

This example shows how to publish and subscribe to topics in a ROS 2 network.

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.
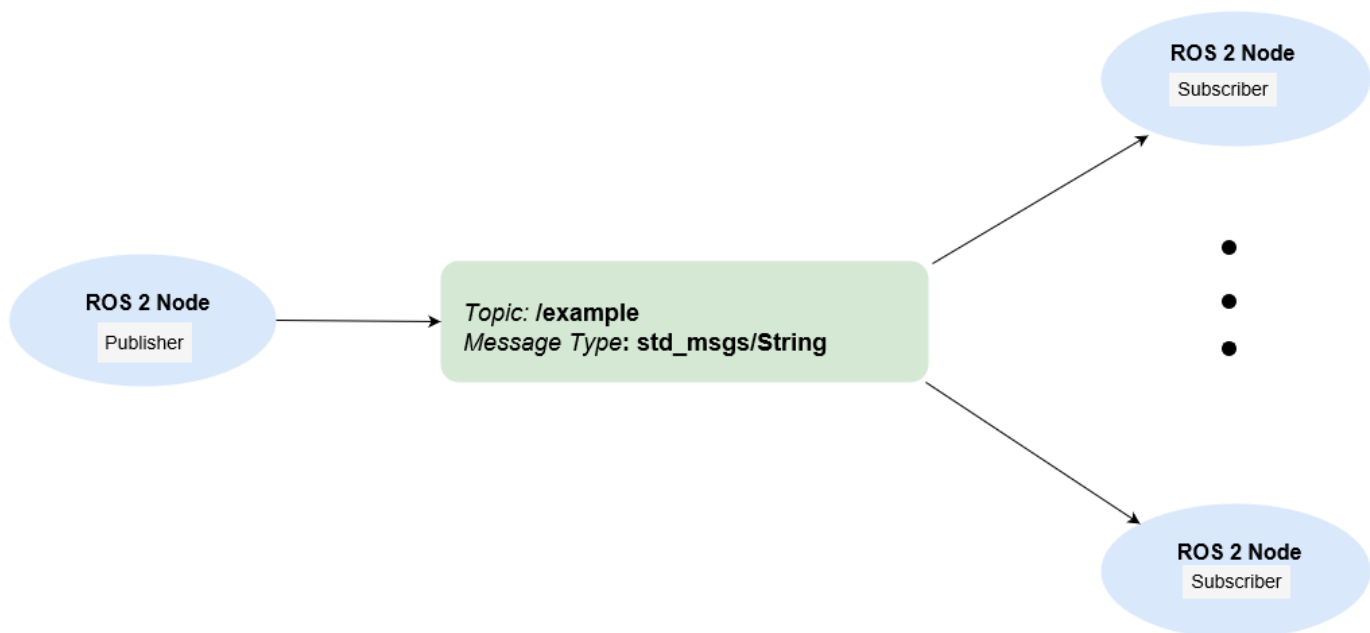
Besides how to publish and subscribe to topics in a ROS 2 network, this example also shows how to:

- 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 2-8

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
/rosout
/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(5)
laserSub = ros2subscriber(detectNode,"/scan");
pause(5)
```

Use `receive` to wait for a new message. Specify a timeout of 10 seconds. The output `scanData` contains the received message data. `status` indicates whether a message was received successfully and `statustext` provides additional information about the `status`.

```
[scanData,status,statustext] = 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");
pause(5)
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)
```

```
    0.00235920447111606        -0.0201184589892978         0.0203969078651195
```

```
disp(orient)
```

```
    -0.0118389124011118        0.00676849978014866         0.0387860955311228
```

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 "Create Callbacks for Graphics Objects" 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
/rosout
/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'
```

2-21

```
        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 2-13
*   "Generate ROS 2 Custom Messages in MATLAB" on page 2-54

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.

```
% 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)")
```
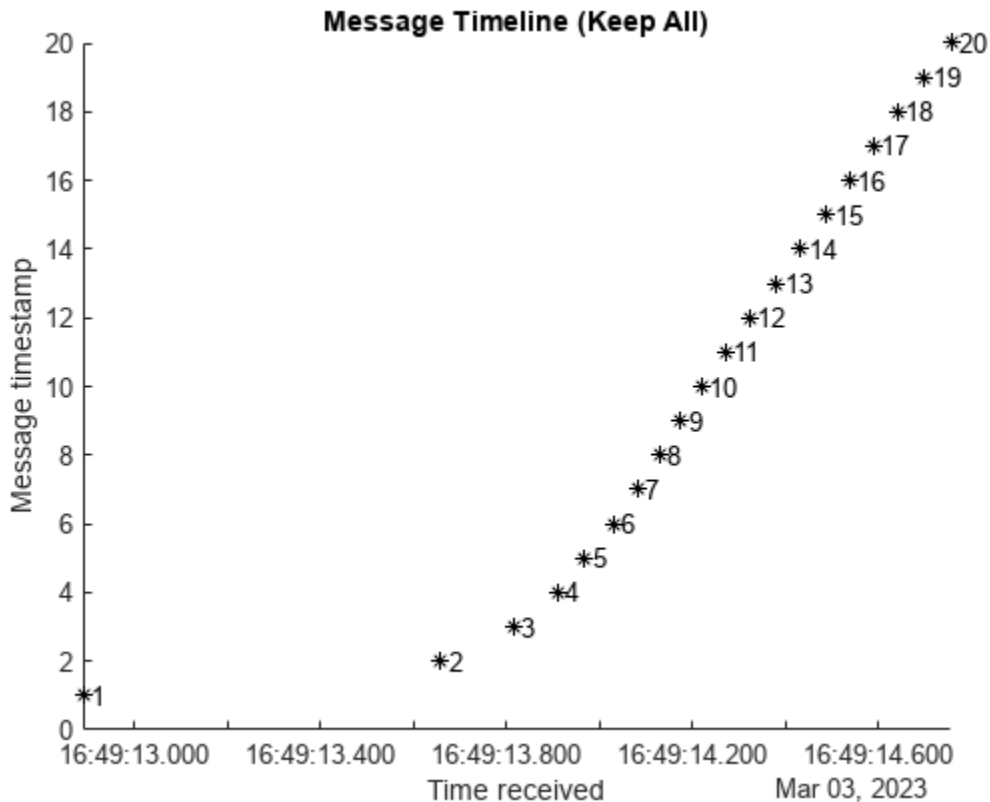
```
localizationSub = ros2subscriber(robotNode,"/laser_scan",...
    @(msg)exampleHelperROS2PlotTimestamps(msg,hAxesLidar),...
    "History","keepall");

% 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)
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)")
```

Message Timeline (Keep Last 1)
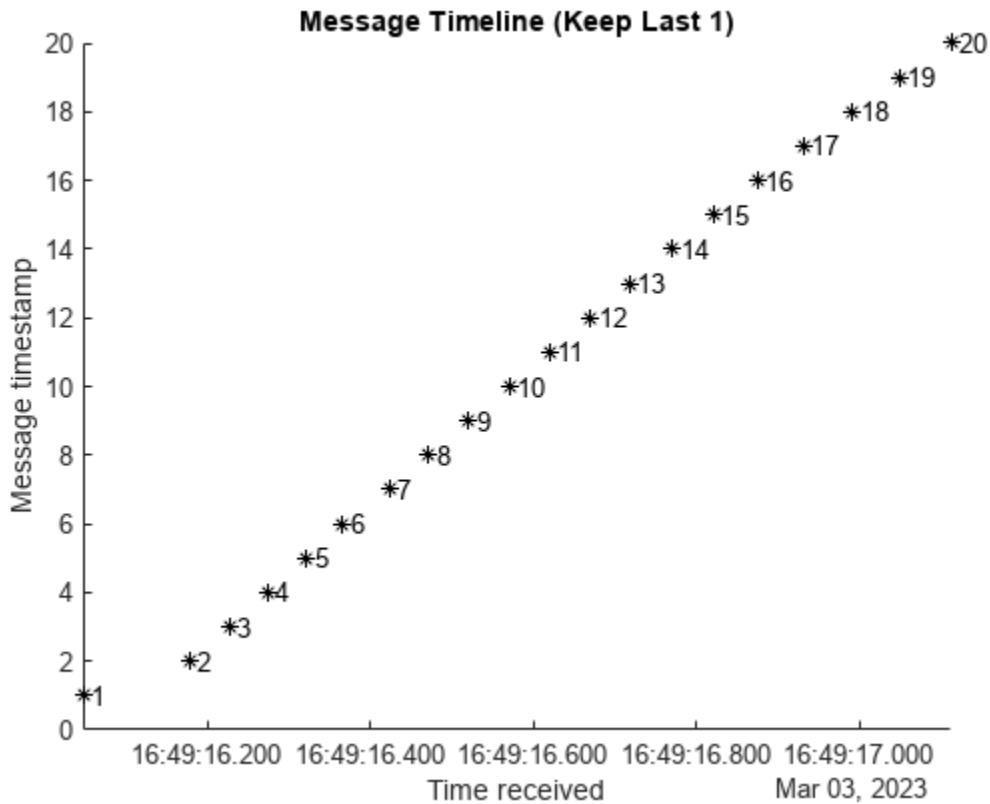
```
scanDisplaySub = ros2subscriber(robotNode,"/laser_scan",...
    @(msg)exampleHelperROS2PlotTimestamps(msg,hAxesLidar2),...
    "History","keeplast","Depth",1);
for iMsg = 1:numel(lidarScans)
    send(lidarPub,lidarScans(iMsg))
end

% Allow messages to arrive, then remove the subscriber and publisher
pause(3)
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)")
```


Robot Position (Reliable Connection)
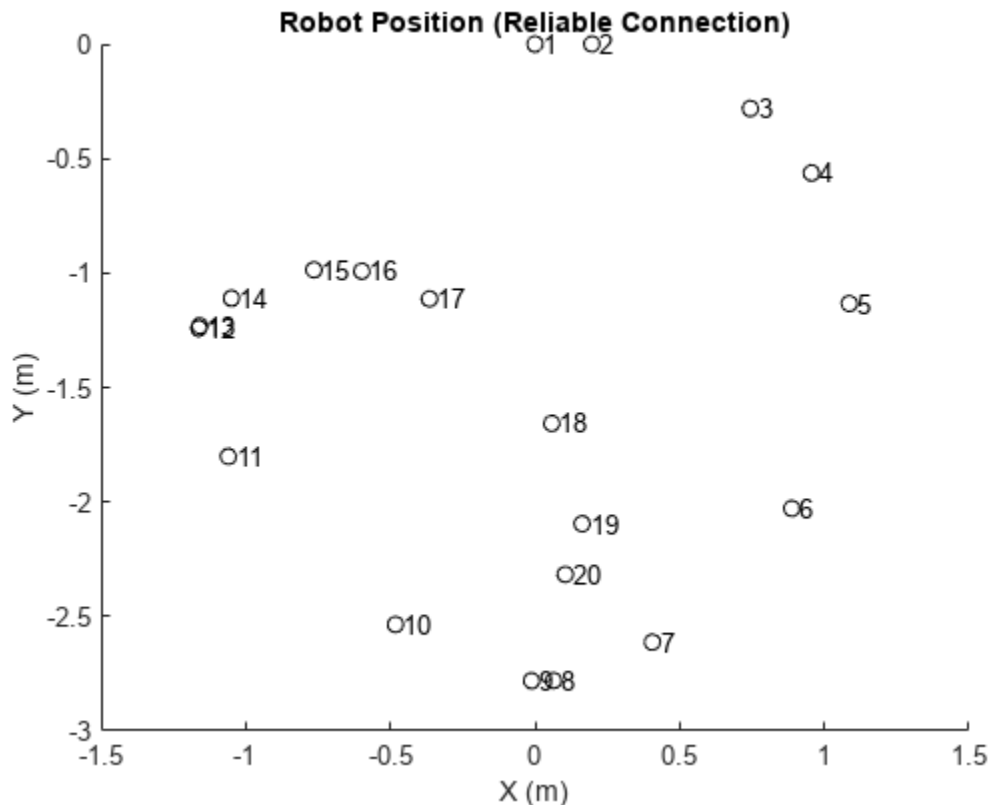
```
odomPlotSub = ros2subscriber(robotNode,"/odom",...
    @(msg)exampleHelperROS2PlotOdom(msg,hAxesReliable,"ok"),...
    "Reliability","reliable");

% Send messages, simulating an extremely fast sensor
load robotPoseLidarData.mat odomData
for iMsg = 1:numel(odomData)
    send(odomPub,odomData(iMsg))
end

pause(5)     % 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)")
```
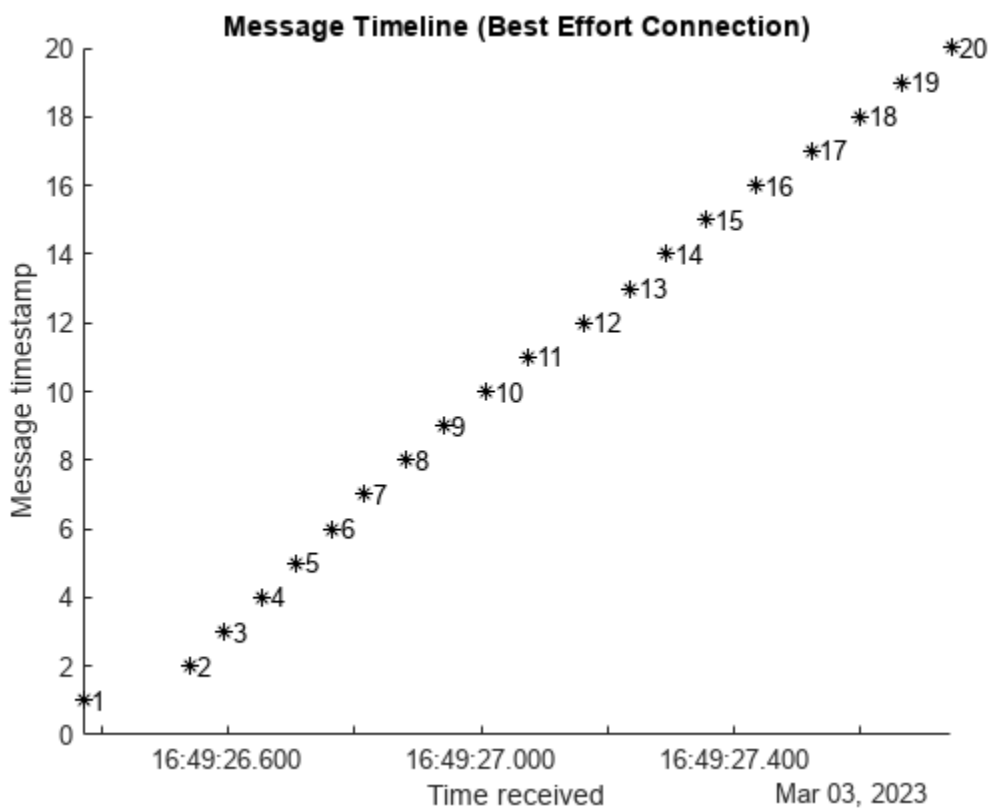


```
odomTimingSub = ros2subscriber(robotNode,"/odom",...
    @(msg)exampleHelperROS2PlotTimestamps(msg,hAxesBestEffort),...
    "Reliability","besteffort");
for iMsg = 1:numel(odomData)
    send(odomPub,odomData(iMsg))
end
```

```
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.

```
% 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
```


Robot Position (Transient Local Connection)
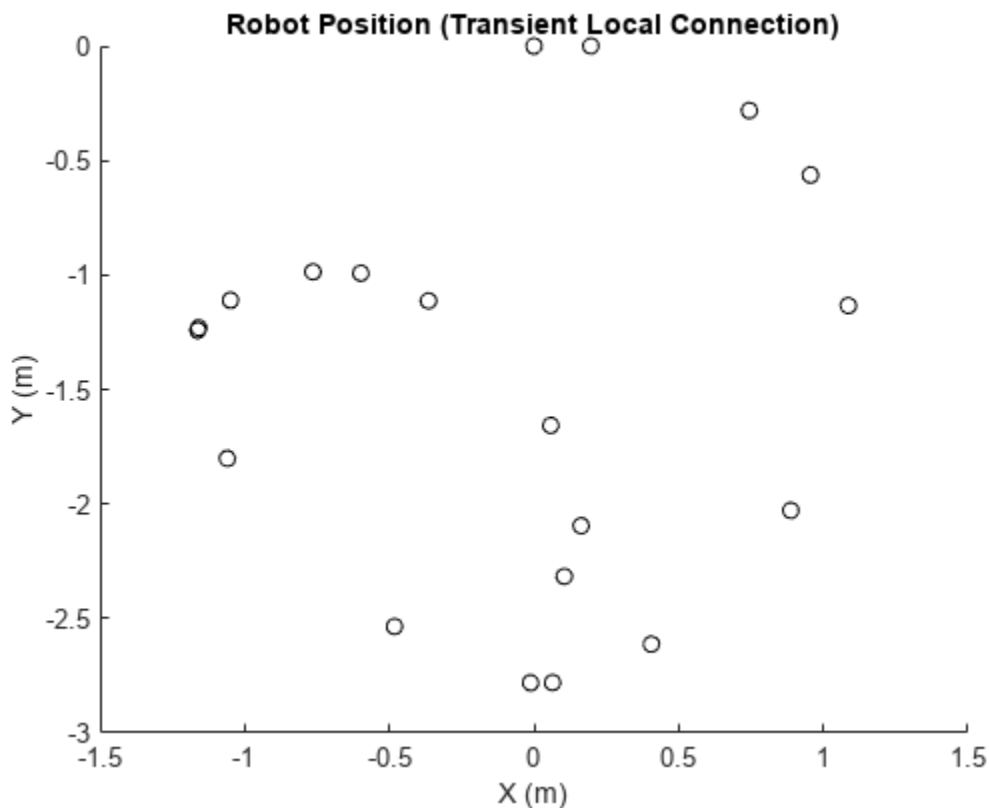
```
posePlotSub = ros2subscriber(robotNode,"/bot_position",...
    @(msg)plot(hAxesMoreMsgs,msg.x,msg.y,"ok"),...
    "Durability","transientlocal","Depth",20);
pause(3)    % Allow messages to arrive and be plotted
```

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
```

Manage Quality of Service Policies in ROS 2 Application with TurtleBot

This example demonstrates best practices in managing Quality of Service (QoS) policies for an application using ROS 2. QoS policies allow for the flexible tuning of communication behavior between publishers and subscribers, and change the way that messages are transported within a ROS 2 network. For more information, see "Manage Quality of Service Policies in ROS 2" on page 2-23.

In this example, you use a MATLAB® script to launch a teleoperation controller for a simulated TurtleBot® to follow a path in based on instructions in the environment.

Start Robot Simulator

Start a ROS 2 simulator for a TurtleBot® and configure the MATLAB connection with the robot simulator.

This example uses a virtual machine (VM). Download the VM using the instructions in "Get Started with Gazebo and Simulated TurtleBot" on page 1-206.

- Start the Ubuntu® virtual machine.
- Select **Gazebo ROS2 Maze** on the Ubuntu desktop to start the Gazebo world built for this example.
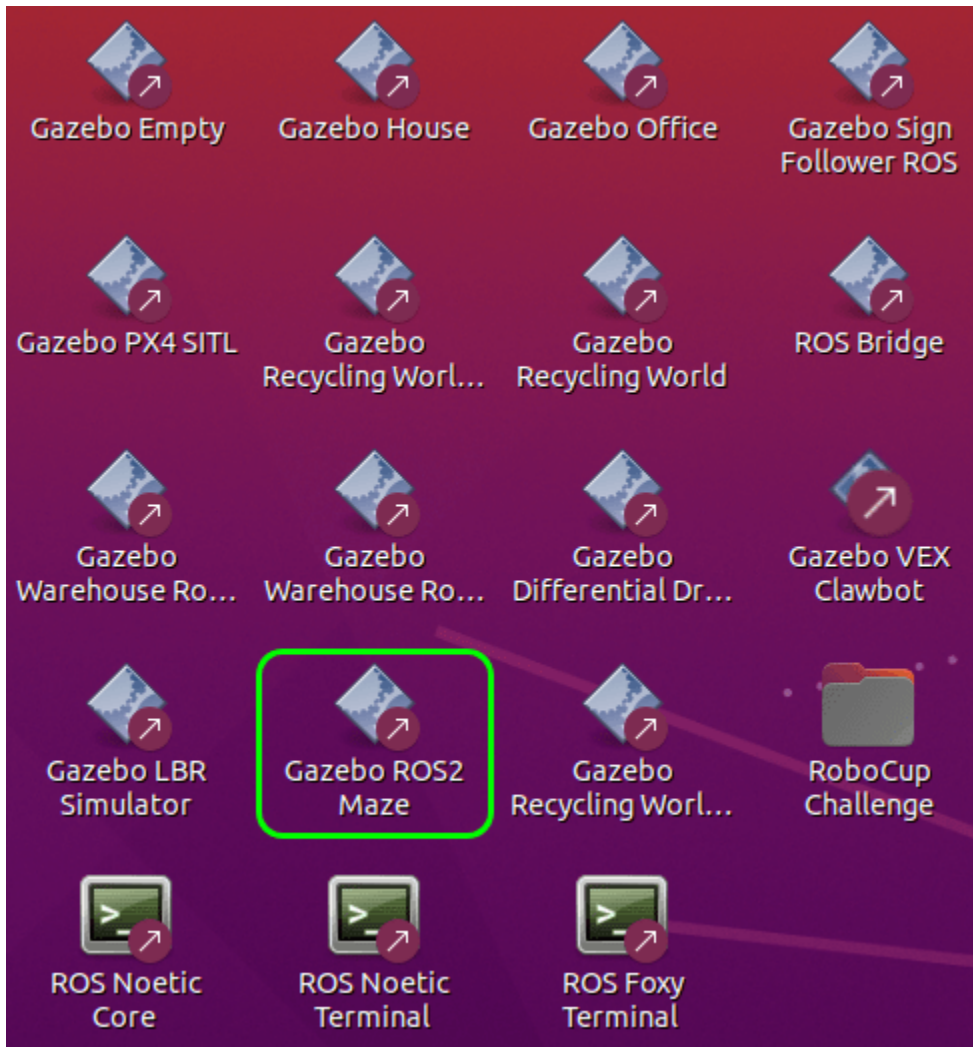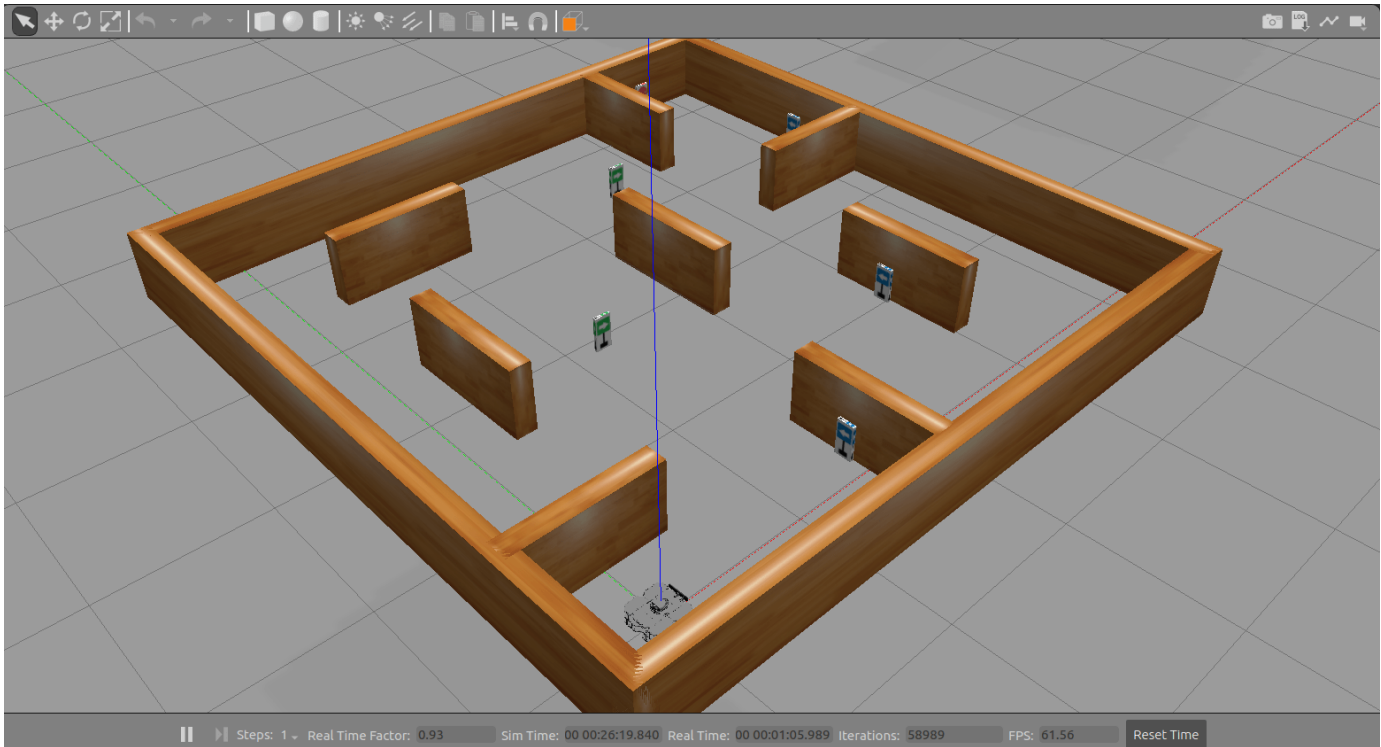
- Enter these commands in the MATLAB Command Window to verify that the topics from the robot simulator are visible in MATLAB.

```
setenv('ROS_DOMAIN_ID','25')
ros2('topic','list')

/camera/camera_info
/camera/image_raw
/clock
/cmd_vel
/imu
/joint_states
/odom
/parameter_events
/rosout
/scan
/tf
```

Set Up ROS 2 Communication

Create two ROS 2 nodes: `/robotDataProcessingNode` and `/humanOperatorNode`. The `/robotDataProcessingNode` receives sensor data to process and publishes messages to keep track of the number of signs detected. The `/humanOperatorNode` sends velocity commands to drive the TurtleBot around the environment and receives a confirmation whenever a sign is detected.

```
domainID = 25;
robotDataProcessingNode = ros2node("/robotDataProcessingNode",domainID);
humanOperatorNode = ros2node("/humanOperatorNode",domainID);
```

This diagram summarizes the interaction between MATLAB and the robot simulator.
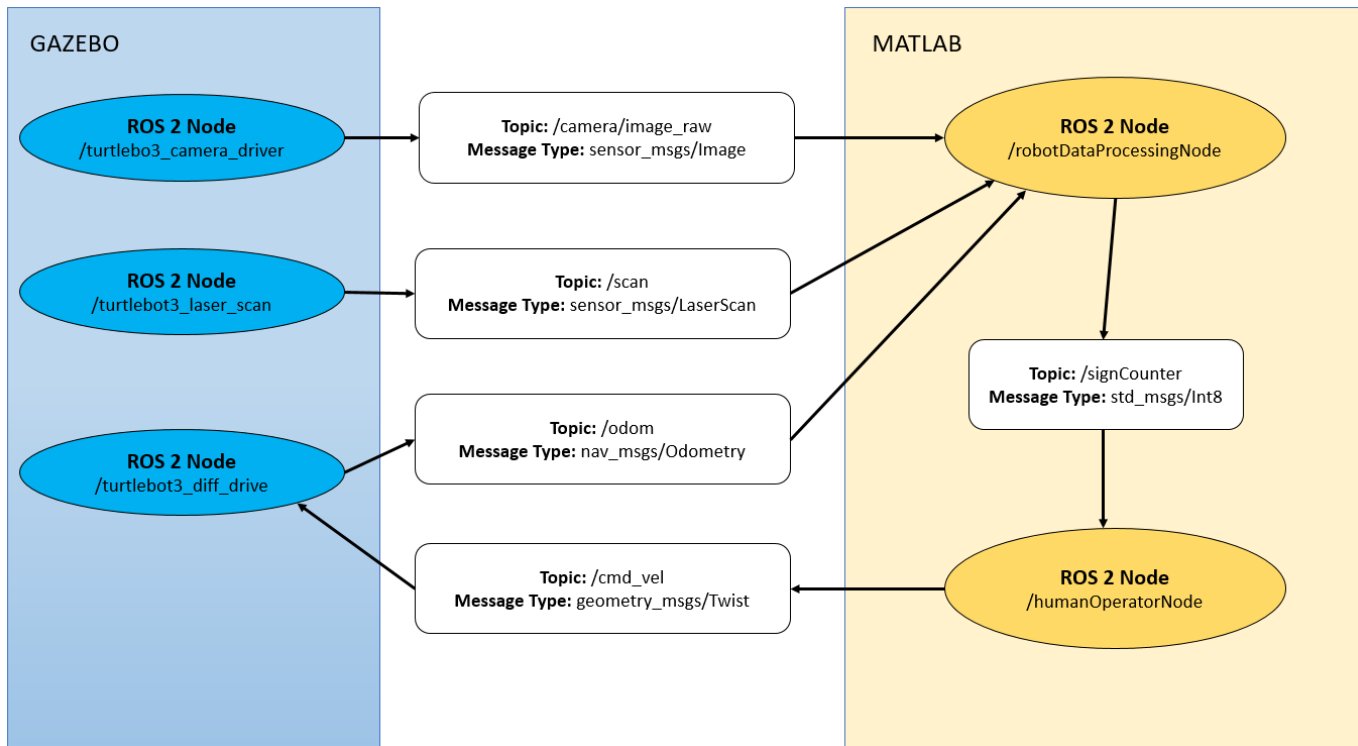
Quality of Service Policies for Control Commands

Create publishers and subscribers to relay messages to and from the robot simulator over the ROS 2 network. A publisher and subscriber pair can have compatible, but different QoS policies unless any QoS policies of the subscriber are more stringent than those of the publisher. For example, you must relay the velocity commands in a reliable channel from the publisher to the subscriber. To ensure compatibility, specify the `"Reliability"` and `"Durability"` QoS policies of the publisher as `"reliable"` and `"transientlocal"`, respectively. This configuration indicates the maximum quality that the controller provides for sending messages reliably. If the receiver on the robot is not equipped with good hardware to reliably process the messages, you can set a lower QoS standard for the subscriber. Thus, specify the `"Reliability"` and `"Durability"` QoS policies of the subscriber to `"besteffort"` and `"volatile"` respectively, which is the minimum quality that the receiver is willing to accept. These QoS settings demonstrate the best practice for specifying `"Reliability"`, and `"Durability"` parameters. Publishers with policies of `"besteffort"` or `"volatile"` do not connect to subscribers with policies of `"reliable"`, or `"transientlocal"`. Because the subscriber is asking for a higher QoS standard than the publisher is offering, delivery of the publisher messages is not guaranteed.

```
velPub = ros2publisher(humanOperatorNode,"/cmd_vel","geometry_msgs/Twist","Reliability","reliable
```
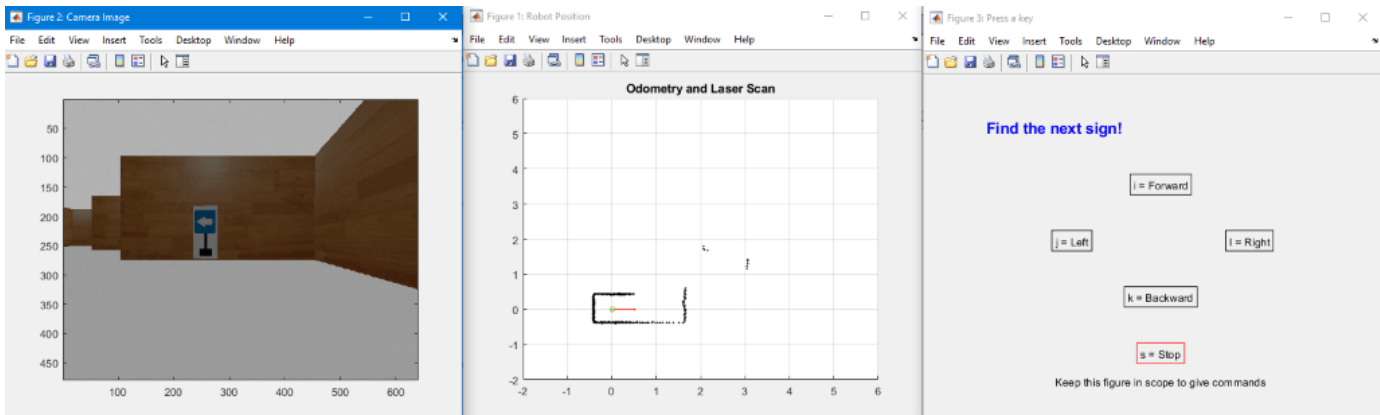
Quality of Service Policies for High-Frequency Sensor Data

To receive the latest reading of sensor data being published at a high rate, set the `"Reliability"` QoS policy of the subscriber to `"besteffort"` and, the `"Durability"` policy to `"volatile"`, with a small `"Depth"` value. These settings enable high-speed communication by reducing the overhead time for sending and receiving message confirmation and ensure that the subscribers process the most recent messages. These settings can result in subscribers not receiving messages in lossy or high-traffic networks, or not processing all messages received if the processing cannot keep up. Apply this QoS policy to both the camera and lidar sensor.

```
imageSub = ros2subscriber(robotDataProcessingNode,"/camera/image_raw","sensor_msgs/Image","Reliab
laserSub = ros2subscriber(robotDataProcessingNode,"/scan","sensor_msgs/LaserScan","Reliability","
```

Because odometry is critical in placing the lidar scans in context, dropped odometry messages result in misleading lidar readings. To prevent dropped messages, the reliability and durability policies for the odometry publisher in the Gazebo node are `"reliable"` and `"transientlocal"`, respectively. As this particular algorithm does not need past messages, specify QoS policies for the odometry subscriber as `"reliable"` and `"volatile"`.

```
odomSub = ros2subscriber(robotDataProcessingNode,"/odom","nav_msgs/Odometry","Reliability","relia
```

Quality of Service Policies for Robot States and Operational Modes

In this example, the information about the current stage of the robot updates at a low frequency, and the value in the latest message received applies until the subscriber receives the next message. Create a publisher that sends messages reliably and stores them for late-joining subscribers. If the `"Depth"` value of the publisher is large enough, it is possible for subscribers to request the entire history of the publisher when they join the network. Configure the publisher of the `/signCounter` topic with `"Reliability"` policy set to `"reliable"`, `"Durability"` policy set to `"transientlocal"`, and `"Depth"` value set to `100`.

```
stagePub = ros2publisher(robotDataProcessingNode,"/signCounter","std_msgs/Int8","Reliability","re
```

This table summarizes the QoS policies for the five pairs of publishers and subscribers.

	Topic	Reliable	Best Effort	Transient Local	Volatile	Depth
Publisher	Camera Sensor	X			X	5
Subscriber	Camera Sensor		X		X	5
Publisher	Lidar Sensor	X			X	5
Subscriber	Lidar Sensor		X		X	5
Publisher	Odometry	X		X		5
Subscriber	Odometry	X			X	5
Publisher	Velocity Command	X		X		5
Subscriber	Velocity Command		X		X	5
Publisher	Sign Counter	X		X		100
Subscriber	Sign Counter	X		X		100

Compatibility in Quality of Service Policies

For messages to pass successfully from publisher to subscriber over a topic, their QoS policies must be compatible. The publishers in the table do not always have the same QoS parameters as their corresponding subscribers, but they are still compatible. For example, in the camera sensor, velocity command, and laser scan topics, the `"reliable"` publishers and `"besteffort"` subscribers are able to connect. The connection behaves as a `"besteffort"` connection, with no confirmation when a subscriber receives a message. Similarly, in the odometry and velocity command topics, the `"transientlocal"` publishers and `"volatile"` subscribers have compatible QoS policies. The publishers retain the published messages while the subscribers do not request any previously sent messages.

Control Robot Using Teleoperation

Run the teleoperation controller to move the robot . Process the sensor data to help the robot visualize and navigate in the environment. When the robot moves close to a sign, the sign-detecting algorithm outputs a confirmation message with the instruction to find the next sign. This task repeats until the robot reaches the stop sign. For information on running the robot in autonomous mode, see "Sign Following Robot with ROS 2 in MATLAB" on page 2-121.

```
[laserPlotObj,imageAxesHandle,signText,axesHandle] = ExampleHelperQoSTurtleBotSetupVisualizer(ve
% Wait to receive sensor messages before starting the control loop
receive(laserSub,5);
receive(odomSub,5);
receive(imageSub,5);
% Set callback functions for subscribers
imageSub.NewMessageFcn = @(msg)ExampleHelperQoSTurtleBotPlotImage(msg,imageAxesHandle);
```

```
laserSub.NewMessageFcn = @(msg)ExampleHelperQoSTurtleBotPlotScan(msg,laserPlotObj,odomSub);
r = rateControl(10);
LastStage = false;
while ~LastStage
    [~,blobSize,blobX] = ExampleHelperQoSTurtleBotProcessImg(imageSub.LatestMessage); % Process
    [nextStage,LastStage,stageMsg,textHandle] = ExampleHelperQoSTurtleBotSignDetection(LastStage
    if nextStage && ~LastStage  % When the algorithm detects a sign, publish a message to keep t
        send(stagePub,stageMsg);
    end
    waitfor(r);
end
```
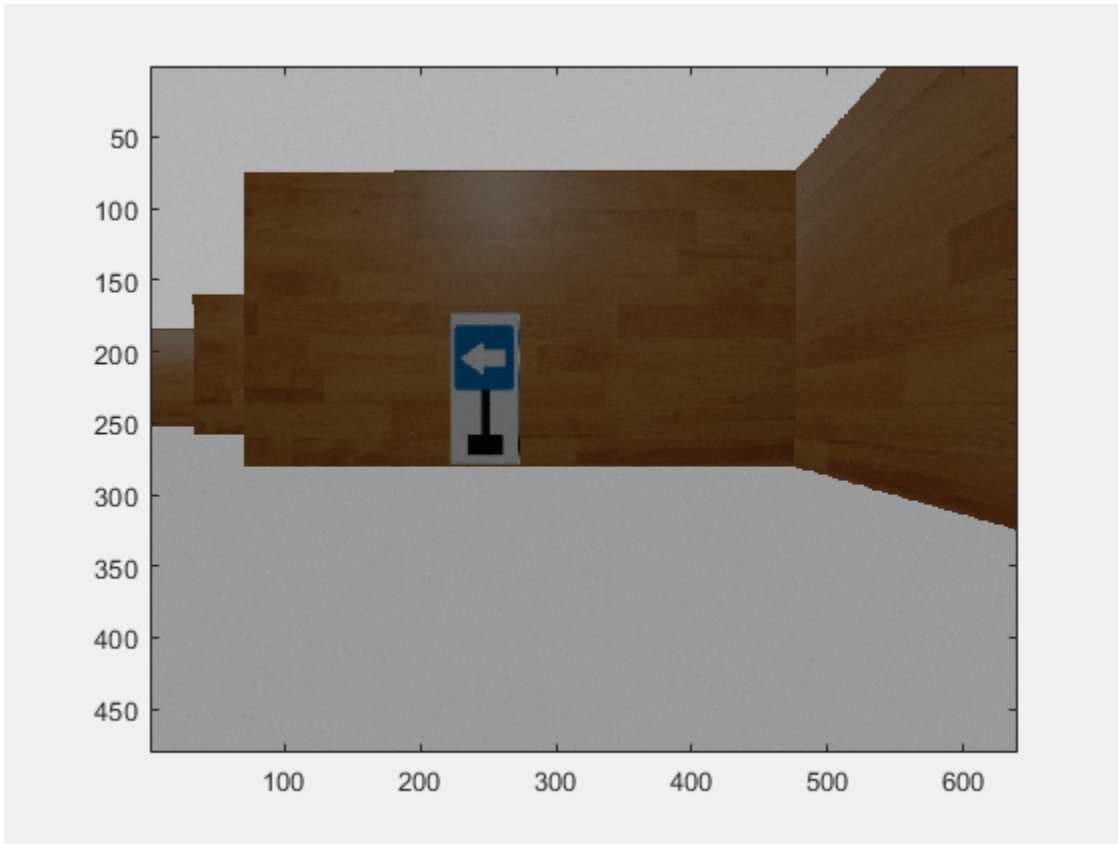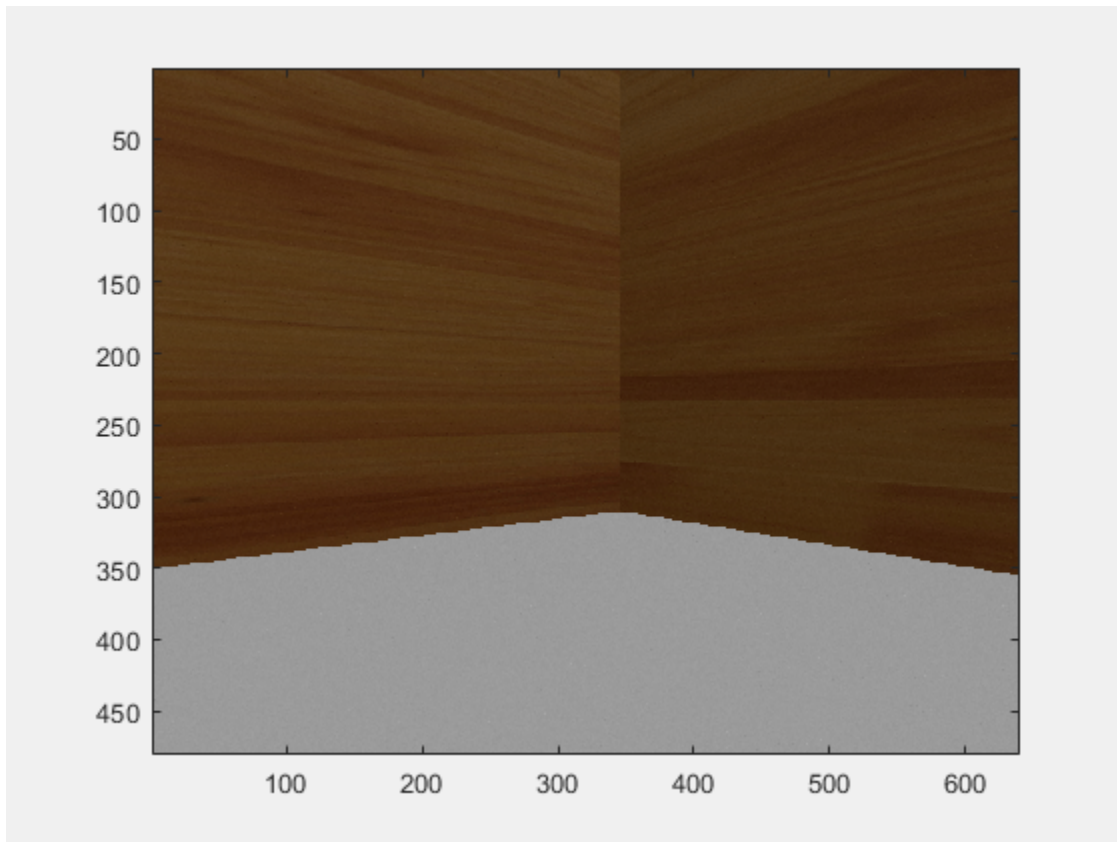
When sensors detect the stop sign, create a new subscriber in the `/humanOperatorNode` node to request the past messages in the history of the publisher. Extract information on all the detected signs.

```
send(stagePub,stageMsg);
stageSub = ros2subscriber(humanOperatorNode,"/signCounter","std_msgs/Int8","Reliability","reliabl
stageSub.NewMessageFcn = @(msg)ExampleHelperQoSTurtleBotSignCountUpdate(msg,textHandle);
pause(2); % Allow time for the persisting messages to be received and processed
```
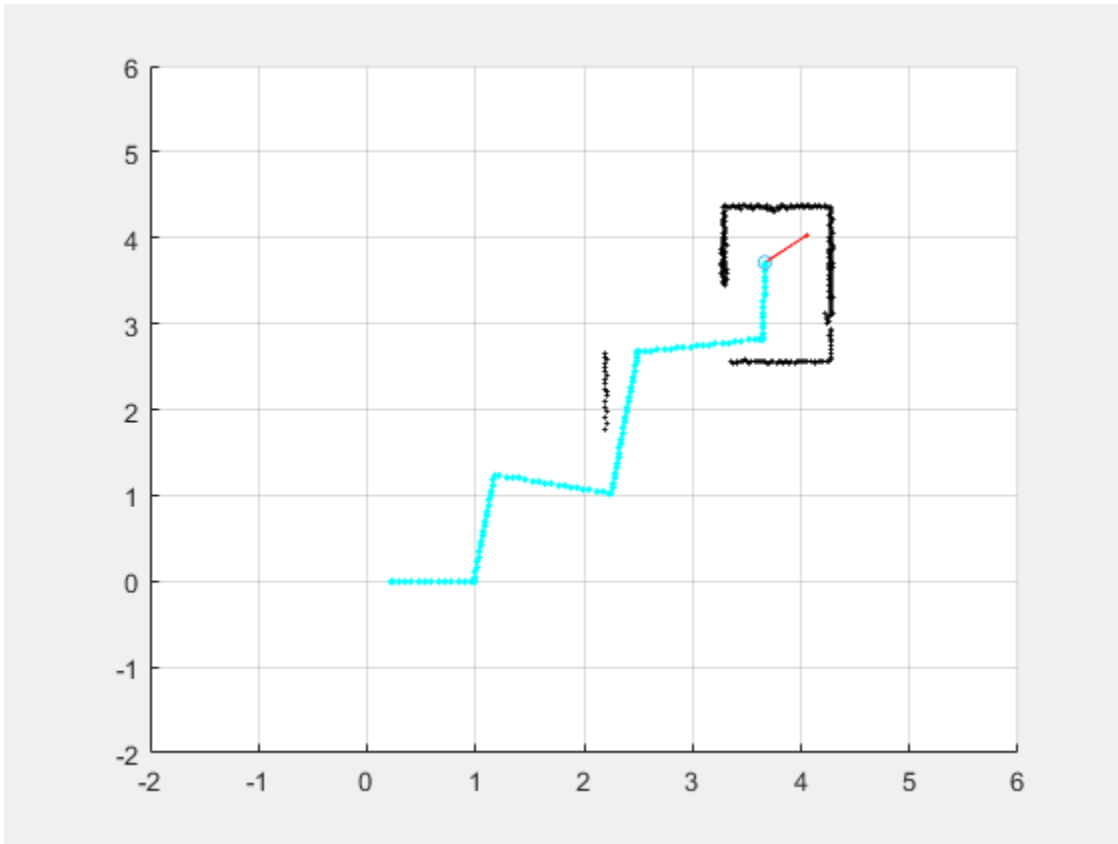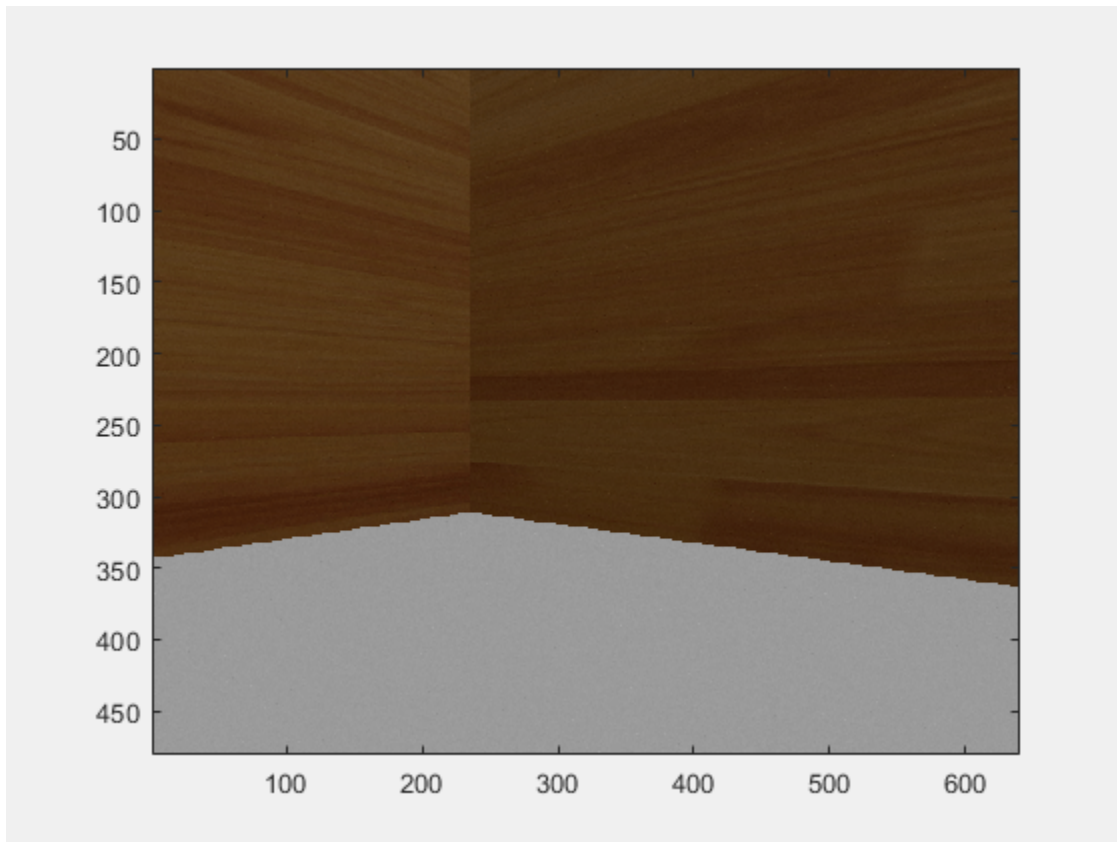
```
% Clean up entities in ROS 2 to remove them from the network.
clear laserSub odomSub velPub imageSub stagePub stageSub robotDataProcessingNode humanOperatorNode
```

Observe Effects of Quality of Service Policies in Lossy Networks

In low-traffic and lossless networks, there is little difference between `reliable` and `best-effort` communication. To visualize how different QoS policies handle lossy or high-traffic network connections, use the traffic control utility to simulate a network with delay. On the VM, open a new terminal and enter this command.

```
sudo tc qdisc add dev ens33 root netem delay 0.5ms
```

The traffic control utility simulates a fixed amount of delay to all packets on the `ens33` network interface. Run the example again, observing the stuttering effect of the image stream due to some dropped frames in `"best-effort"` communication. If you change the image subscriber to `"reliable"`, the image stream smooths out, but lags behind the actual robot viewpoint slightly due to the network delay.

To clean up, remove the artificial network lag by entering this command.

```
sudo tc qdisc delete dev ens33 root netem delay 0.5ms
```
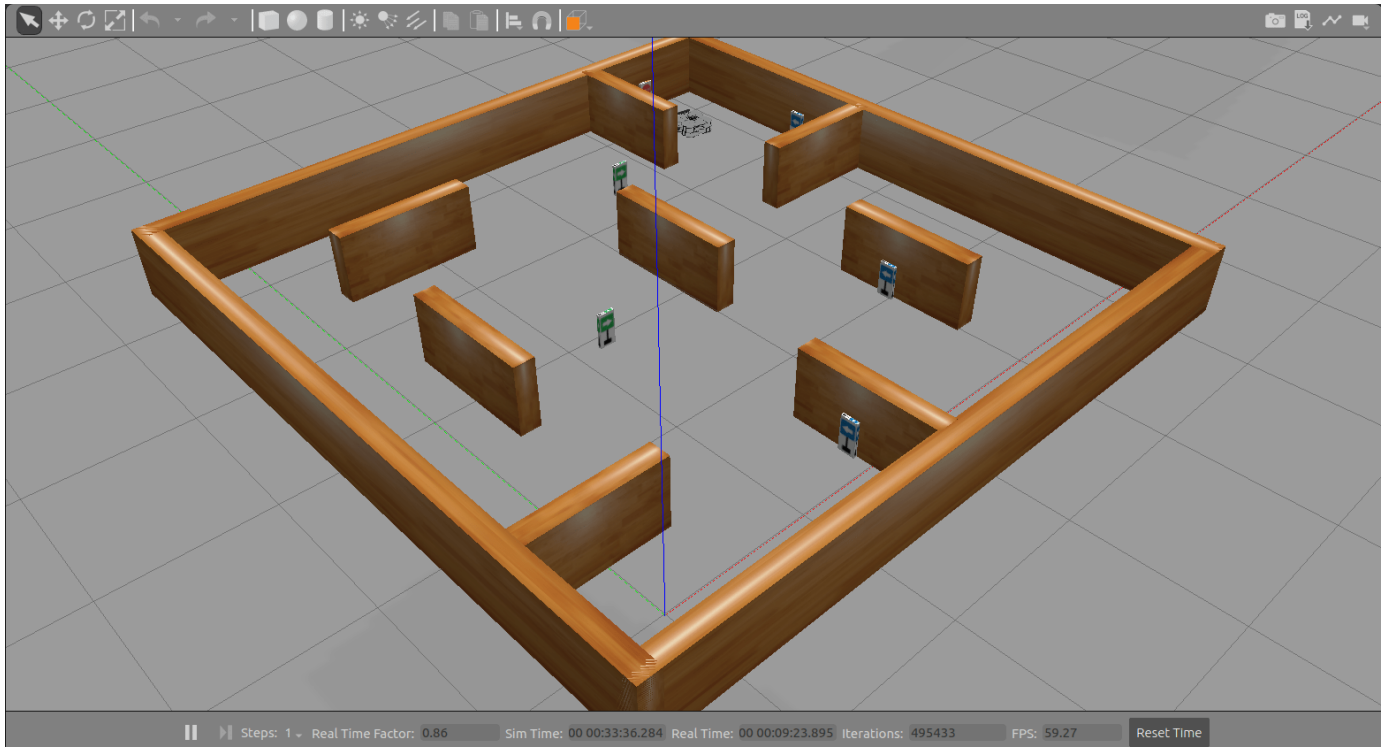
Call and Provide ROS 2 Services

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 2 Publishers and Subscribers" on page 2-19). *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.

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 2" on page 2-2, "Connect to a ROS 2 Network" on page 2-8, "Exchange Data with ROS 2 Publishers and Subscribers" on page 2-19

Create Service Server

Before examining service concepts, create the sample ROS 2 network with three nodes to simulate a realistic network.

```
node_1 = ros2node("node_1");
node_2 = ros2node("node_2");
node_3 = ros2node("node_3");
```

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 `ros2svcserver` command. Specify the service name, the service message type and the node to attach the server. Also define the callback function as `exampleHelperROSEmptyCallback`. Callback functions for service servers have a very specific signature. For details, see the documentation of `ros2svcserver`.

```
testserver = ros2svcserver(node_1,"/service_example","std_srvs/Empty",@exampleHelperROS2EmptyCal
```

```
testserver =
  ros2svcserver with properties:

      ServiceType: 'std_srvs/Empty'
      ServiceName: '/service_example'
    NewRequestFcn: @exampleHelperROS2EmptyCallback
          History: 'keeplast'
            Depth: 10
      Reliability: 'reliable'
       Durability: 'volatile'
```

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

```
ros2("service","list");
```

```
/_matlab_introspec__0_rmw_fastrtps_cpp/describe_parameters
/_matlab_introspec__0_rmw_fastrtps_cpp/get_parameter_types
/_matlab_introspec__0_rmw_fastrtps_cpp/get_parameters
/_matlab_introspec__0_rmw_fastrtps_cpp/list_parameters
/_matlab_introspec__0_rmw_fastrtps_cpp/set_parameters
/_matlab_introspec__0_rmw_fastrtps_cpp/set_parameters_atomically
/node_1/describe_parameters
/node_1/get_parameter_types
/node_1/get_parameters
/node_1/list_parameters
/node_1/set_parameters
/node_1/set_parameters_atomically
/node_2/describe_parameters
/node_2/get_parameter_types
/node_2/get_parameters
/node_2/list_parameters
/node_2/set_parameters
/node_2/set_parameters_atomically
/node_3/describe_parameters
/node_3/get_parameter_types
/node_3/get_parameters
/node_3/list_parameters
/node_3/set_parameters
/node_3/set_parameters_atomically
/service_example
```

Create Service Client

Use service clients to request information from a ROS 2 service server. To create a client, use `ros2svcclient` with the service name as the argument.

Create a service client for the `/test` service that we just created and attach it to a different node.

```
testclient = ros2svcclient(node_2,"/service_example","std_srvs/Empty")

testclient =
  ros2svcclient with properties:

    ServiceType: 'std_srvs/Empty'
    ServiceName: '/service_example'
        History: 'keeplast'
          Depth: 10
    Reliability: 'reliable'
     Durability: 'volatile'
```

Create an empty request message for the service. Use the `ros2message` 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 = ros2message(testclient)

testreq = struct with fields:
    MessageType: 'std_srvs/EmptyRequest'
```

Ensure that the service is connected to the client, waiting for them to connect if necessary.

```
waitForServer(testclient,"Timeout",3)
```

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. `status` and `statustext` outputs provide additional information about the status of the response.

```
testresp = call(testclient,testreq,"Timeout",3)

testresp = struct with fields:
    MessageType: 'std_srvs/EmptyResponse'
```

If the call function above fails, it will error. Instead of an error, if you would prefer to react to a call failure using conditionals, return the status and statustext outputs from the call function. The status output indicates if the call succeeded, while statustext provides additional information. Similar outputs can be returned from waitForServer.

```
numCallFailures = 0;
[testresp,status,statustext] = call(testclient,testreq,"Timeout",3);
if ~status
    numCallFailures = numCallFailues + 1;
    fprintf("Call failure number %d. Error cause: %s\n",numCallFailures,statustext)
else
```

```
      disp(testresp)
end

    MessageType: 'std_srvs/EmptyResponse'
```

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, `example_interfaces/AddTwoInts`, that we can use for this task. You can inspect the structure of the request and response messages by calling `ros2 msg show`. The request contains two integers, `a` and `b`, and the response contains their addition in `sum`.

```
ros2 msg show example_interfaces/AddTwoIntsRequest

int64 a
int64 b

ros2 msg show example_interfaces/AddTwoIntsResponse

int64 sum
```

Create the service server with this message type and a callback function that calculates the addition. For your convenience, the `exampleHelperROS2SumCallback` function already implements this calculation. Specify the function as a callback.

```
sumserver = ros2svcserver(node_1,"/sum","example_interfaces/AddTwoInts",@exampleHelperROS2SumCal

sumserver =
  ros2svcserver with properties:

      ServiceType: 'example_interfaces/AddTwoInts'
      ServiceName: '/sum'
    NewRequestFcn: @exampleHelperROS2SumCallback
          History: 'keeplast'
            Depth: 10
      Reliability: 'reliable'
       Durability: 'volatile'
```

To call the service server, you have to create a service client. Note that this client can be created anywhere in the ROS 2 network. For the purposes of this example, we will create a client for the `/sum` service in MATLAB. Then we will wait to ensure the client has connected to the server.

```
sumclient = ros2svcclient(node_3,"/sum","example_interfaces/AddTwoInts")

sumclient =
  ros2svcclient with properties:

    ServiceType: 'example_interfaces/AddTwoInts'
    ServiceName: '/sum'
        History: 'keeplast'
          Depth: 10
    Reliability: 'reliable'
     Durability: 'volatile'


waitForServer(sumclient,"Timeout",3);
```

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

```
sumreq = ros2message(sumclient);
sumreq.a = int64(2);
sumreq.b = int64(1);
```

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 = struct with fields:
    MessageType: 'example_interfaces/AddTwoIntsResponse'
            sum: 3
```

Shut Down ROS 2 Network

Remove the sample nodes and service servers from the ROS 2 network.

```
clear("node_1");
clear("node_2");
clear("node_3");
```

Next Steps

- Refer to "Work with Basic ROS 2 Messages" on page 2-13 to explore how ROS 2 messages are represented in MATLAB.

Access the tf Transformation Tree in ROS 2

The `tf` system in ROS 2 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 2 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.

Create a ROS 2 node on domain ID 25.

```
node = ros2node("/matlabNode",25);
```

To create a realistic environment for this example, use `exampleHelperROS2StartTfPublisher` 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 transformations 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

```
exampleHelperROS2StartTfPublisher
```

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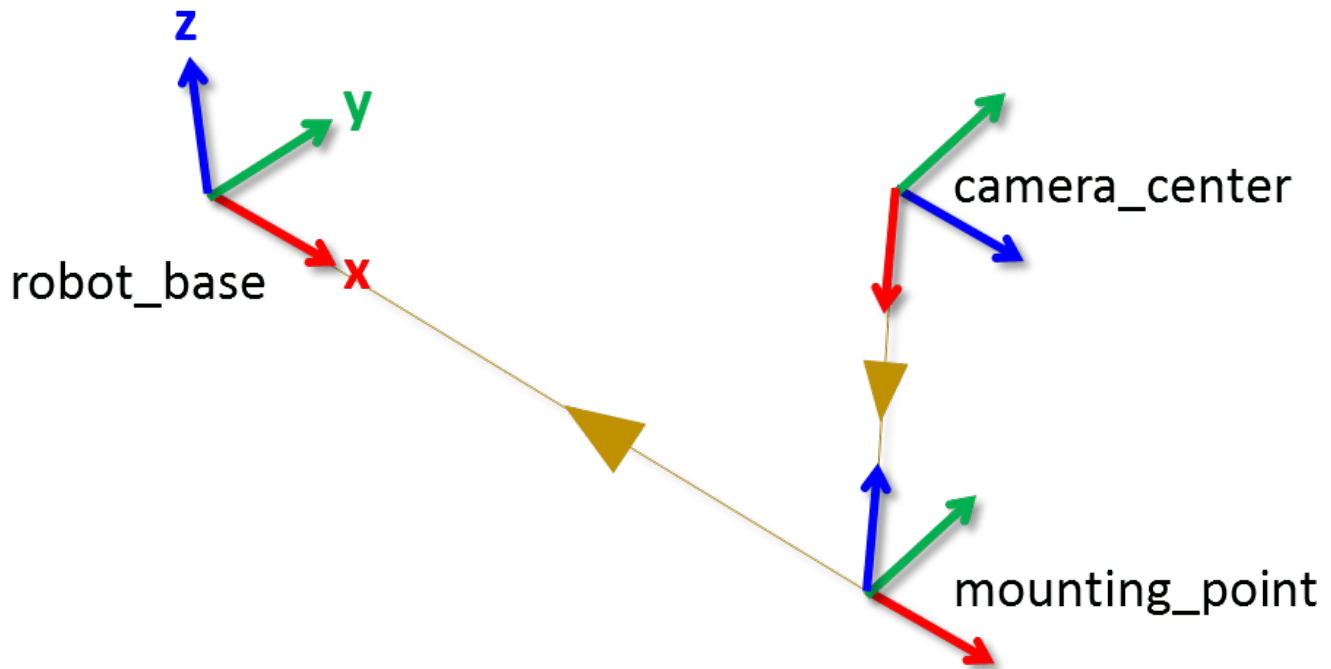
Here, the x, y, and z axes of each frame are represented by red, green, and blue lines respectively. The parent-child relationship between the coordinate frames is shown through a brown arrow pointing from the child to its parent frame.

Create a new transformation tree using `ros2tf` object. You can use this object to access all available transformations and apply them to different entities.

```
tftree = ros2tf(node,...
"DynamicBroadcasterQoS", struct('Depth', 50), ...
"DynamicListenerQoS", struct('Reliability','besteffort'), ...
"StaticBroadcasterQoS", struct('Depth',10), ...
"StaticListenerQoS", struct('Durability','volatile'));
```

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 = 3×1 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 2 `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 = struct with fields:
    MessageType: 'geometry_msgs/Vector3'
              x: 0
              y: 0
              z: 0.5000


quat = mountToCamera.transform.rotation

quat = struct with fields:
    MessageType: 'geometry_msgs/Quaternion'
              x: 0
              y: 0.7071
              z: 0
              w: 0.7071


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 = struct with fields:
    MessageType: 'geometry_msgs/Vector3'
              x: 1
              y: 0
              z: 0


baseToMountRotation = baseToMount.transform.rotation

baseToMountRotation = struct with fields:
    MessageType: 'geometry_msgs/Quaternion'
```

```
               x: 0
               y: 0
               z: 0
               w: 1
```

```
baseToMountRotationRotationAngles = rad2deg(quat2eul([baseToMountRotation.w baseToMountRotation.x
```

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

```
getTransform(tftree,'robot_base','camera_center','Timeout', Inf);
```

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 = ros2message('geometry_msgs/PointStamped');
pt.header.frame_id = '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 = struct with fields:
    MessageType: 'geometry_msgs/PointStamped'
         header: [1×1 struct]
          point: [1×1 struct]
```

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 = struct with fields:
       MessageType: 'geometry_msgs/TransformStamped'
            header: [1×1 struct]
     child_frame_id: 'camera_center'
         transform: [1×1 struct]
```

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

```
robotToCamera.transform.translation
```

```
ans = struct with fields:
    MessageType: 'geometry_msgs/Vector3'
              x: 1
              y: 0
              z: 0.5000
```

```
robotToCamera.transform.rotation
```

```
ans = struct with fields:
    MessageType: 'geometry_msgs/Quaternion'
              x: 0
              y: 0.7071
              z: 0
              w: 0.7071
```

Send Transformations

You can also broadcast a new transformation to the ROS 2 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 `child_frame_id` property. The target coordinate frame, `robot_base`, is added to the `header.FrameId` property.

```
tfStampedMsg = ros2message('geometry_msgs/TransformStamped');
tfStampedMsg.child_frame_id = 'wheel';
tfStampedMsg.header.frame_id = '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)]);
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 `ros2time` 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 = ros2time(node,'now');
```

Broadcast this transformation over the ROS 2 network.

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

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

```
tftree.AvailableFrames
```

```
ans = 4×1 cell
    {'camera_center' }
    {'mounting_point'}
    {'robot_base'    }
    {'wheel'         }
```

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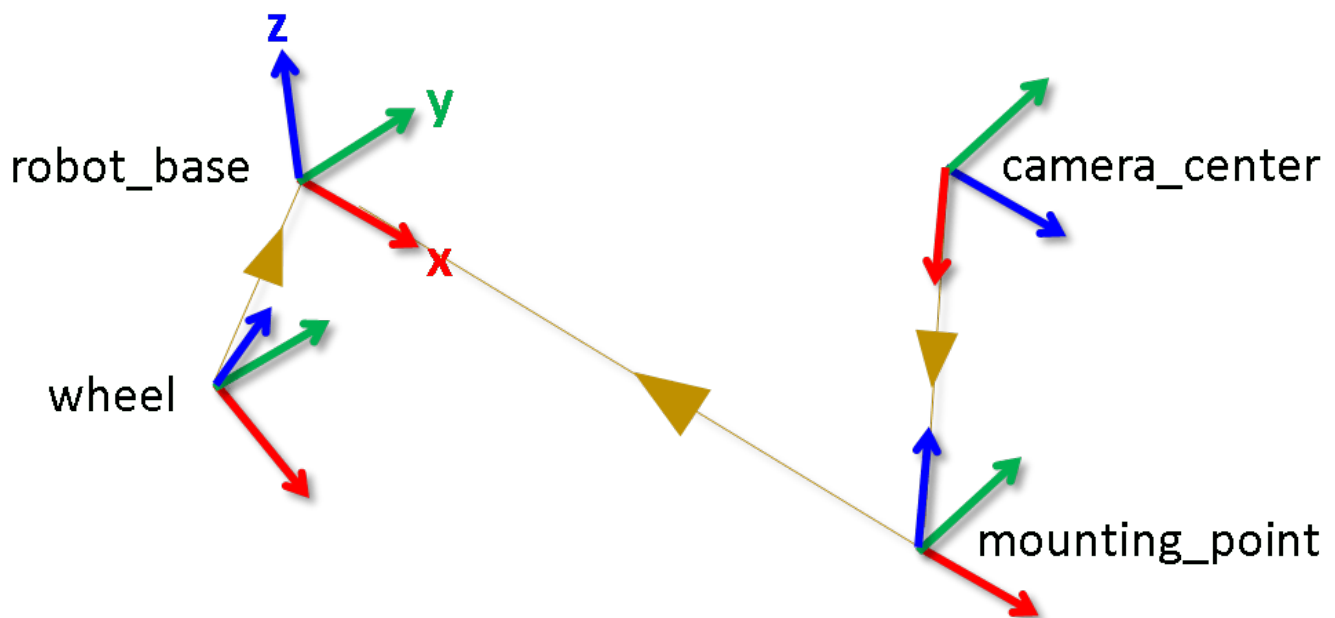
You can see that the `wheel` coordinate frame has the translation and rotation that you specified in sending the transformation.

Stop Example Publisher

Stop the example transformation publisher.

```
exampleHelperROS2StopTfPublisher
```

Clear the node.

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

Generate ROS 2 Custom Messages in MATLAB

Use custom messages to extend the set of message types currently supported in ROS 2. Custom messages are messages that you define. 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, enter `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 2-13.

If this if your first time working with ROS 2 custom messages, see "ROS Toolbox System Requirements".

ROS 2 custom messages are specified in ROS 2 package folders that contain a folder named `msg`. The `msg` folder contains all your custom message type definitions. For example, the `example_b_msgs` package in the `custom` folder, has this folder and file structure.

```
⊟ 📁 custom
   ⊞ 📁 example_a_msgs
   ⊟ 📁 example_b_msgs
      ⊟ 📁 msg
           ✉ Standalone.msg
   ⊞ 📁 example_c_msgs
```

The package contains the custom message type `Standalone.msg`. MATLAB uses these files to generate the necessary files for using the custom messages contained in the package.

In this example, you create 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, you specify the path to the parent folder and 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.

Open a new MATLAB session and create a custom message folder in a local folder.

```
folderPath = fullfile(pwd,"custom");
copyfile("example_*_msgs",folderPath);
```

Specify the folder path for custom message files and use `ros2genmsg` to create custom messages.

```
ros2genmsg(folderPath)
```

```
Identifying message files in folder 'C:/Work/custom'.Done.
Removing previous version of Python virtual environment.Done.
Creating a Python virtual environment.Done.
Adding required Python packages to virtual environment.Done.
Copying include folders.Done.
Copying libraries.Done.
Validating message files in folder 'C:/Work/custom'.Done.
[3/3] Generating MATLAB interfaces for custom message packages... Done.
Running colcon build in folder 'C:/Work/custom/matlab_msg_gen/win64'.
```

```
Build in progress. This may take several minutes...
Build succeeded.build log
```

Call `ros2 msg list` to verify creation of new custom messages.

`ros2 msg list`

```
action_msgs/CancelGoalRequest
action_msgs/CancelGoalResponse
action_msgs/GoalInfo
action_msgs/GoalStatus
action_msgs/GoalStatusArray
actionlib_msgs/GoalID
actionlib_msgs/GoalStatus
actionlib_msgs/GoalStatusArray
builtin_interfaces/Duration
builtin_interfaces/Time
composition_interfaces/ListNodesRequest
composition_interfaces/ListNodesResponse
composition_interfaces/LoadNodeRequest
composition_interfaces/LoadNodeResponse
composition_interfaces/UnloadNodeRequest
composition_interfaces/UnloadNodeResponse
diagnostic_msgs/AddDiagnosticsRequest
diagnostic_msgs/AddDiagnosticsResponse
diagnostic_msgs/DiagnosticArray
diagnostic_msgs/DiagnosticStatus
diagnostic_msgs/KeyValue
diagnostic_msgs/SelfTestRequest
diagnostic_msgs/SelfTestResponse
example_a_msgs/DependsOnB
example_b_msgs/Standalone
example_c_msgs/DependsOnB
example_interfaces/AddTwoIntsRequest
example_interfaces/AddTwoIntsResponse
example_interfaces/Bool
example_interfaces/Byte
example_interfaces/ByteMultiArray
example_interfaces/Char
example_interfaces/Empty
example_interfaces/Float32
example_interfaces/Float32MultiArray
example_interfaces/Float64
example_interfaces/Float64MultiArray
example_interfaces/Int16
example_interfaces/Int16MultiArray
example_interfaces/Int32
example_interfaces/Int32MultiArray
example_interfaces/Int64
example_interfaces/Int64MultiArray
example_interfaces/Int8
example_interfaces/Int8MultiArray
example_interfaces/MultiArrayDimension
example_interfaces/MultiArrayLayout
example_interfaces/SetBoolRequest
example_interfaces/SetBoolResponse
example_interfaces/String
example_interfaces/TriggerRequest
```

```
example_interfaces/TriggerResponse
example_interfaces/UInt16
example_interfaces/UInt16MultiArray
example_interfaces/UInt32
example_interfaces/UInt32MultiArray
example_interfaces/UInt64
example_interfaces/UInt64MultiArray
example_interfaces/UInt8
example_interfaces/UInt8MultiArray
example_interfaces/WString
geometry_msgs/Accel
geometry_msgs/AccelStamped
geometry_msgs/AccelWithCovariance
geometry_msgs/AccelWithCovarianceStamped
geometry_msgs/Inertia
geometry_msgs/InertiaStamped
geometry_msgs/Point
geometry_msgs/Point32
geometry_msgs/PointStamped
geometry_msgs/Polygon
geometry_msgs/PolygonStamped
geometry_msgs/Pose
geometry_msgs/Pose2D
geometry_msgs/PoseArray
geometry_msgs/PoseStamped
geometry_msgs/PoseWithCovariance
geometry_msgs/PoseWithCovarianceStamped
geometry_msgs/Quaternion
geometry_msgs/QuaternionStamped
geometry_msgs/Transform
geometry_msgs/TransformStamped
geometry_msgs/Twist
geometry_msgs/TwistStamped
geometry_msgs/TwistWithCovariance
geometry_msgs/TwistWithCovarianceStamped
geometry_msgs/Vector3
geometry_msgs/Vector3Stamped
geometry_msgs/Wrench
geometry_msgs/WrenchStamped
lifecycle_msgs/ChangeStateRequest
lifecycle_msgs/ChangeStateResponse
lifecycle_msgs/GetAvailableStatesRequest
lifecycle_msgs/GetAvailableStatesResponse
lifecycle_msgs/GetAvailableTransitionsRequest
lifecycle_msgs/GetAvailableTransitionsResponse
lifecycle_msgs/GetStateRequest
lifecycle_msgs/GetStateResponse
lifecycle_msgs/State
lifecycle_msgs/Transition
lifecycle_msgs/TransitionDescription
lifecycle_msgs/TransitionEvent
logging_demo/ConfigLoggerRequest
logging_demo/ConfigLoggerResponse
map_msgs/GetMapROIRequest
map_msgs/GetMapROIResponse
map_msgs/GetPointMapROIRequest
map_msgs/GetPointMapROIResponse
map_msgs/GetPointMapRequest
```

```
map_msgs/GetPointMapResponse
map_msgs/OccupancyGridUpdate
map_msgs/PointCloud2Update
map_msgs/ProjectedMap
map_msgs/ProjectedMapInfo
map_msgs/ProjectedMapsInfoRequest
map_msgs/ProjectedMapsInfoResponse
map_msgs/SaveMapRequest
map_msgs/SaveMapResponse
map_msgs/SetMapProjectionsRequest
map_msgs/SetMapProjectionsResponse
nav_msgs/GetMapRequest
nav_msgs/GetMapResponse
nav_msgs/GetPlanRequest
nav_msgs/GetPlanResponse
nav_msgs/GridCells
nav_msgs/MapMetaData
nav_msgs/OccupancyGrid
nav_msgs/Odometry
nav_msgs/Path
nav_msgs/SetMapRequest
nav_msgs/SetMapResponse
pendulum_msgs/JointCommand
pendulum_msgs/JointState
pendulum_msgs/RttestResults
rcl_interfaces/DescribeParametersRequest
rcl_interfaces/DescribeParametersResponse
rcl_interfaces/FloatingPointRange
rcl_interfaces/GetParameterTypesRequest
rcl_interfaces/GetParameterTypesResponse
rcl_interfaces/GetParametersRequest
rcl_interfaces/GetParametersResponse
rcl_interfaces/IntegerRange
rcl_interfaces/ListParametersRequest
rcl_interfaces/ListParametersResponse
rcl_interfaces/ListParametersResult
rcl_interfaces/Log
rcl_interfaces/Parameter
rcl_interfaces/ParameterDescriptor
rcl_interfaces/ParameterEvent
rcl_interfaces/ParameterEventDescriptors
rcl_interfaces/ParameterType
rcl_interfaces/ParameterValue
rcl_interfaces/SetParametersAtomicallyRequest
rcl_interfaces/SetParametersAtomicallyResponse
rcl_interfaces/SetParametersRequest
rcl_interfaces/SetParametersResponse
rcl_interfaces/SetParametersResult
rosgraph_msgs/Clock
sensor_msgs/BatteryState
sensor_msgs/CameraInfo
sensor_msgs/ChannelFloat32
sensor_msgs/CompressedImage
sensor_msgs/FluidPressure
sensor_msgs/Illuminance
sensor_msgs/Image
sensor_msgs/Imu
sensor_msgs/JointState
```

```
sensor_msgs/Joy
sensor_msgs/JoyFeedback
sensor_msgs/JoyFeedbackArray
sensor_msgs/LaserEcho
sensor_msgs/LaserScan
sensor_msgs/MagneticField
sensor_msgs/MultiDOFJointState
sensor_msgs/MultiEchoLaserScan
sensor_msgs/NavSatFix
sensor_msgs/NavSatStatus
sensor_msgs/PointCloud
sensor_msgs/PointCloud2
sensor_msgs/PointField
sensor_msgs/Range
sensor_msgs/RegionOfInterest
sensor_msgs/RelativeHumidity
sensor_msgs/SetCameraInfoRequest
sensor_msgs/SetCameraInfoResponse
sensor_msgs/Temperature
sensor_msgs/TimeReference
shape_msgs/Mesh
shape_msgs/MeshTriangle
shape_msgs/Plane
shape_msgs/SolidPrimitive
simple_msgs/AddTwoIntsRequest
simple_msgs/AddTwoIntsResponse
simple_msgs/Num
statistics_msgs/MetricsMessage
statistics_msgs/StatisticDataPoint
statistics_msgs/StatisticDataType
std_msgs/Bool
std_msgs/Byte
std_msgs/ByteMultiArray
std_msgs/Char
std_msgs/ColorRGBA
std_msgs/Empty
std_msgs/Float32
std_msgs/Float32MultiArray
std_msgs/Float64
std_msgs/Float64MultiArray
std_msgs/Header
std_msgs/Int16
std_msgs/Int16MultiArray
std_msgs/Int32
std_msgs/Int32MultiArray
std_msgs/Int64
std_msgs/Int64MultiArray
std_msgs/Int8
std_msgs/Int8MultiArray
std_msgs/MultiArrayDimension
std_msgs/MultiArrayLayout
std_msgs/String
std_msgs/UInt16
std_msgs/UInt16MultiArray
std_msgs/UInt32
std_msgs/UInt32MultiArray
std_msgs/UInt64
std_msgs/UInt64MultiArray
```

```
std_msgs/UInt8
std_msgs/UInt8MultiArray
std_srvs/EmptyRequest
std_srvs/EmptyResponse
std_srvs/SetBoolRequest
std_srvs/SetBoolResponse
std_srvs/TriggerRequest
std_srvs/TriggerResponse
stereo_msgs/DisparityImage
test_msgs/Arrays
test_msgs/ArraysRequest
test_msgs/ArraysResponse
test_msgs/BasicTypes
test_msgs/BasicTypesRequest
test_msgs/BasicTypesResponse
test_msgs/BoundedSequences
test_msgs/Builtins
test_msgs/Constants
test_msgs/Defaults
test_msgs/Empty
test_msgs/EmptyRequest
test_msgs/EmptyResponse
test_msgs/MultiNested
test_msgs/Nested
test_msgs/Strings
test_msgs/UnboundedSequences
test_msgs/WStrings
trajectory_msgs/JointTrajectory
trajectory_msgs/JointTrajectoryPoint
trajectory_msgs/MultiDOFJointTrajectory
trajectory_msgs/MultiDOFJointTrajectoryPoint
unique_identifier_msgs/UUID
visualization_msgs/GetInteractiveMarkersRequest
visualization_msgs/GetInteractiveMarkersResponse
visualization_msgs/ImageMarker
visualization_msgs/InteractiveMarker
visualization_msgs/InteractiveMarkerControl
visualization_msgs/InteractiveMarkerFeedback
visualization_msgs/InteractiveMarkerInit
visualization_msgs/InteractiveMarkerPose
visualization_msgs/InteractiveMarkerUpdate
visualization_msgs/Marker
visualization_msgs/MarkerArray
visualization_msgs/MenuEntry
```

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 2-19.

Create a publisher to use `example_b_msgs/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:
      MessageType: 'example_b_msgs/Standalone'
     int_property: 12
  string_property: 'This is ROS 2 custom message example'
```

Remove the created ROS objects.

```
clear node pub sub
```

Replacing Definitions of Built-In Messages with Custom Definitions

MATLAB provides a lot of built-in ROS 2 message types. You can replace the definitions of those message types with new definitions using the same custom message creation workflow detailed above. When you are replacing the definitions of a built-in message package, you must ensure that the custom message package folder contains new definitions (`.msg` files) for all the message types in the corresponding built-in message package.

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 Simulated TurtleBot" on page 1-206. 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.
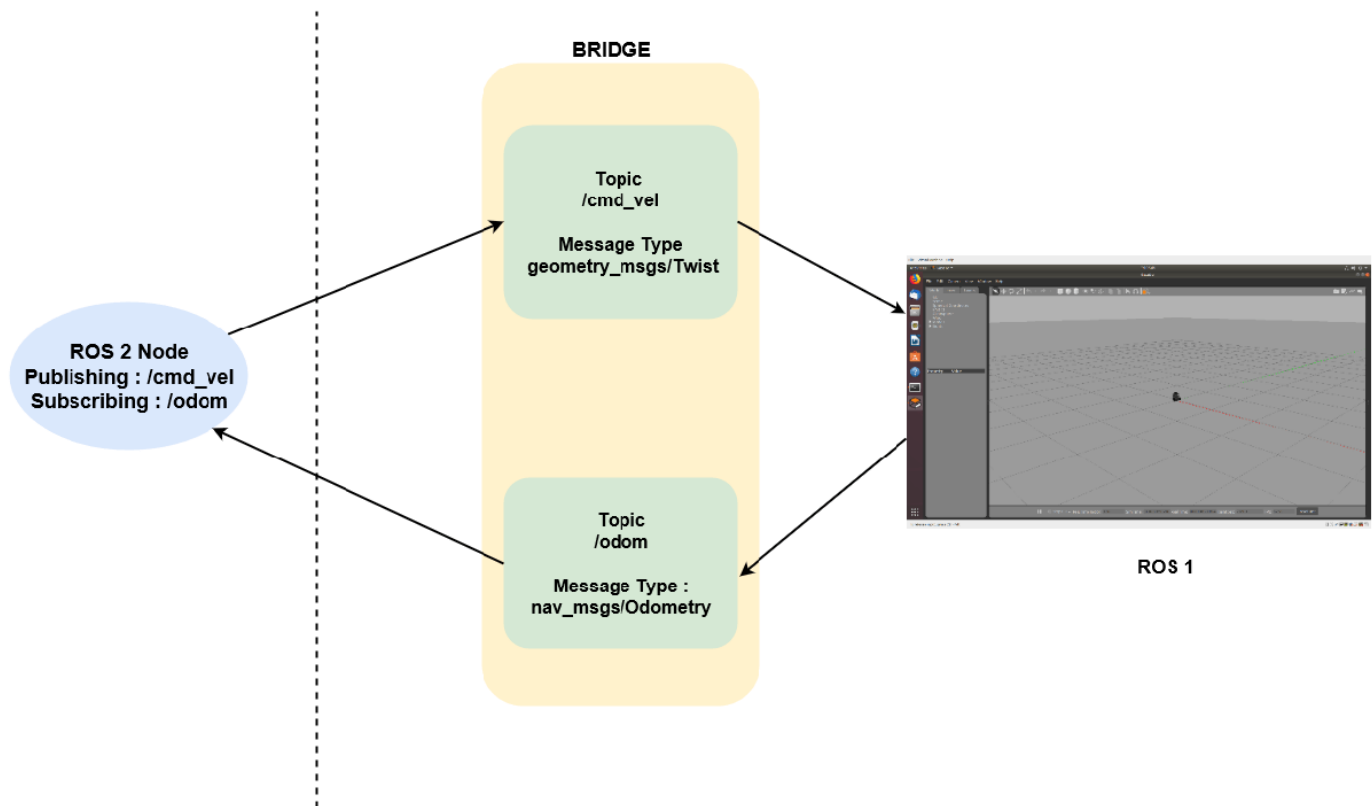
Connecting ROS 1 and ROS 2 Using ROS Bridge

Prerequisites:

- Download the Virtual Machine using instructions in "Get Started with Gazebo and Simulated TurtleBot" on page 1-206
- "Connect to a ROS 2 Network" on page 2-8

Set Up Virtual Machine

Communicate Outside Subnet

Open the *DEFAULT_FASTRTPS_PROFILE.xml* file in the home directory of the VM and replace `<address>` entries with host and VM IP addresses. To communicate under different subnets (see **Communicate Outside Subnet** section in "Connect to a ROS 2 Network" on page 2-8).

```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.
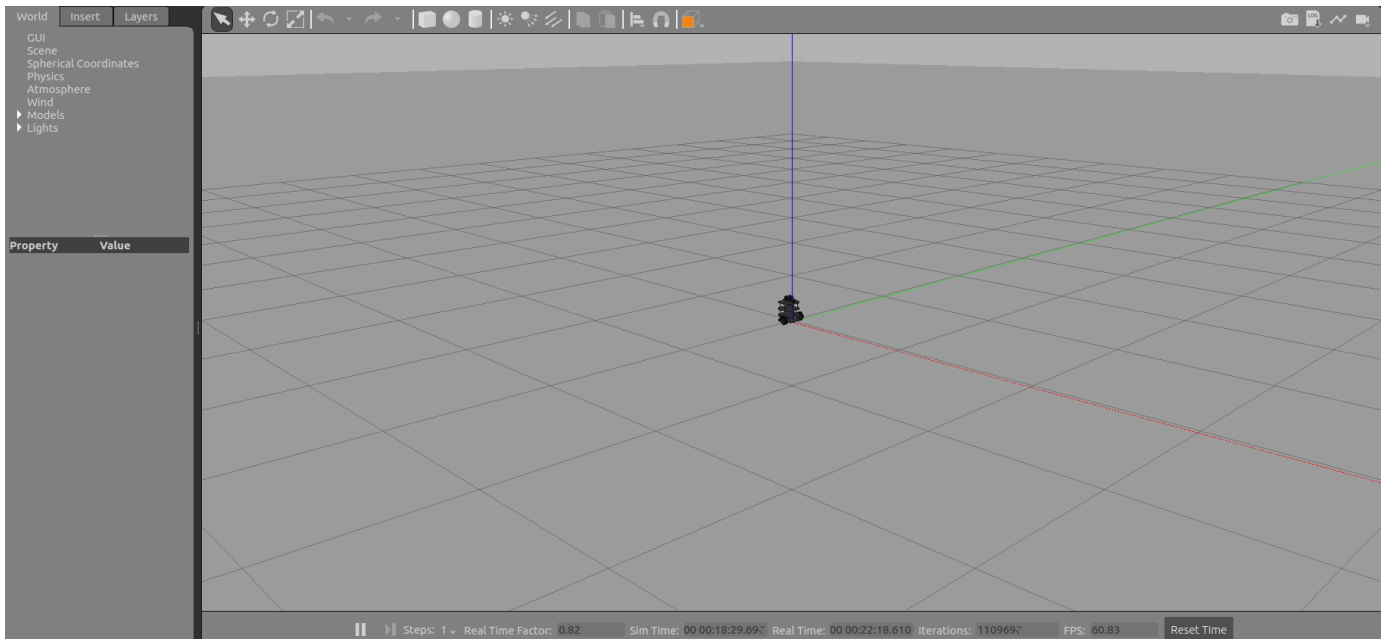
Start the Bridge

Click the **ROS Bridge** shortcut. This bridge setups publishers and subscribers for all the ROS 1 topics on a ROS 2 network.
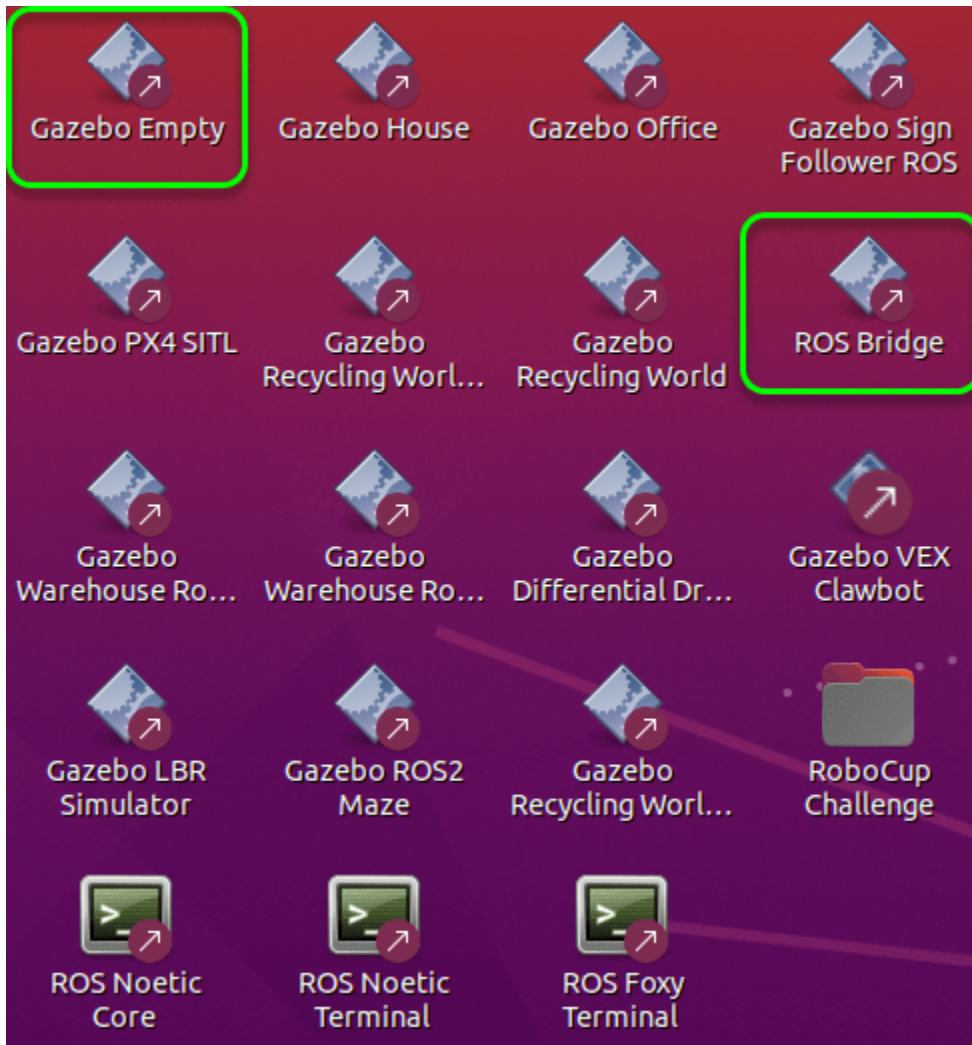
In the Terminal window, notice that the bridge is up and running.

```
ROS_DISTRO was set to 'noetic' before. Please make sure that the environment does not mix paths from different distributions.
created 1to2 bridge for topic '/clock' with ROS 1 type 'rosgraph_msgs/Clock' and ROS 2 type 'rosgraph_msgs/msg/Clock'
created 1to2 bridge for topic '/gazebo/link_states' with ROS 1 type 'gazebo_msgs/LinkStates' and ROS 2 type 'gazebo_msgs/msg/LinkStates'
created 1to2 bridge for topic '/gazebo/model_states' with ROS 1 type 'gazebo_msgs/ModelStates' and ROS 2 type 'gazebo_msgs/msg/ModelStates'
created 1to2 bridge for topic '/gazebo/performance_metrics' with ROS 1 type 'gazebo_msgs/PerformanceMetrics' and ROS 2 type 'gazebo_msgs/msg/PerformanceMetrics'
created 1to2 bridge for topic '/imu' with ROS 1 type 'sensor_msgs/Imu' and ROS 2 type 'sensor_msgs/msg/Imu'
created 1to2 bridge for topic '/joint_states' with ROS 1 type 'sensor_msgs/JointState' and ROS 2 type 'sensor_msgs/msg/JointState'
created 1to2 bridge for topic '/odom' with ROS 1 type 'nav_msgs/Odometry' and ROS 2 type 'nav_msgs/msg/Odometry'
created 1to2 bridge for topic '/rosout' with ROS 1 type 'rosgraph_msgs/Log' and ROS 2 type 'rcl_interfaces/msg/Log'
created 1to2 bridge for topic '/rosout_agg' with ROS 1 type 'rosgraph_msgs/Log' and ROS 2 type 'rcl_interfaces/msg/Log'
created 1to2 bridge for topic '/scan' with ROS 1 type 'sensor_msgs/LaserScan' and ROS 2 type 'sensor_msgs/msg/LaserScan'
created 1to2 bridge for topic '/tf' with ROS 1 type 'tf2_msgs/TFMessage' and ROS 2 type 'tf2_msgs/msg/TFMessage'
Created 2 to 1 bridge for service /gazebo/clear_body_wrenches
Created 2 to 1 bridge for service /gazebo/clear_joint_forces
Created 2 to 1 bridge for service /gazebo/delete_light
Created 2 to 1 bridge for service /gazebo/delete_model
Created 2 to 1 bridge for service /gazebo/get_joint_properties
Created 2 to 1 bridge for service /gazebo/get_light_properties
Created 2 to 1 bridge for service /gazebo/get_link_properties
Created 2 to 1 bridge for service /gazebo/get_link_state
Created 2 to 1 bridge for service /gazebo/get_model_properties
Created 2 to 1 bridge for service /gazebo/get_model_state
Created 2 to 1 bridge for service /gazebo/get_physics_properties
Created 2 to 1 bridge for service /gazebo/get_world_properties
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/set_joint_properties
Created 2 to 1 bridge for service /gazebo/set_link_properties
Created 2 to 1 bridge for service /gazebo/set_link_state
Created 2 to 1 bridge for service /gazebo/set_model_configuration
Created 2 to 1 bridge for service /gazebo/set_model_state
Created 2 to 1 bridge for service /gazebo/set_physics_properties
Created 2 to 1 bridge for service /gazebo/spawn_sdf_model
Created 2 to 1 bridge for service /gazebo/spawn_urdf_model
Created 2 to 1 bridge for service /gazebo/unpause_physics
Created 2 to 1 bridge for service /imu_service
[INFO] [1640937684.677792824] [ros_bridge]: Passing message from ROS 1 rosgraph_msgs/Clock to ROS 2 rosgraph_msgs/msg/Clock (showing msg only once per type)
[INFO] [1640937684.680416182] [ros_bridge]: Passing message from ROS 1 gazebo_msgs/PerformanceMetrics to ROS 2 gazebo_msgs/msg/PerformanceMetrics (showing msg only once per type)
[INFO] [1640937684.680518075] [ros_bridge]: Passing message from ROS 1 sensor_msgs/Imu to ROS 2 sensor_msgs/msg/Imu (showing msg only once per type)
[INFO] [1640937684.680604478] [ros_bridge]: Passing message from ROS 1 gazebo_msgs/LinkStates to ROS 2 gazebo_msgs/msg/LinkStates (showing msg only once per type)
[INFO] [1640937684.680683298] [ros_bridge]: Passing message from ROS 1 gazebo_msgs/ModelStates to ROS 2 gazebo_msgs/msg/ModelStates (showing msg only once per type)
[INFO] [1640937684.680828935] [ros_bridge]: Passing message from ROS 1 nav_msgs/Odometry to ROS 2 nav_msgs/msg/Odometry (showing msg only once per type)
[INFO] [1640937684.680956259] [ros_bridge]: Passing message from ROS 1 sensor_msgs/JointState to ROS 2 sensor_msgs/msg/JointState (showing msg only once per type)
[INFO] [1640937684.799564124] [ros_bridge]: Passing message from ROS 1 tf2_msgs/TFMessage to ROS 2 tf2_msgs/msg/TFMessage (showing msg only once per type)
[INFO] [1640937684.889330211] [ros_bridge]: Passing message from ROS 1 sensor_msgs/LaserScan to ROS 2 sensor_msgs/msg/LaserScan (showing msg only once per type)
created 2to1 bridge for topic '/rosout' with ROS 2 type 'rcl_interfaces/msg/Log' and ROS 1 type 'rosgraph_msgs/Log'
```

Open one more terminal and enter the following commands

```
export ROS_DOMAIN_ID=25
source /opt/ros/foxy/setup.bash
```

Now check that Gazebo topics are present in ROS 2.

```
ros2 topic list
```

```
user@ubuntu:~$ export ROS_DOMAIN_ID=25
user@ubuntu:~$ source /opt/ros/foxy/setup.bash
user@ubuntu:~$ ros2 topic list
/clock
/gazebo/link_states
/gazebo/model_states
/gazebo/performance_metrics
/imu
/joint_states
/odom
/parameter_events
/rosout
/rosout_agg
/scan
/tf
```

Echo the **/odom** topic to see messages being published.

```
ros2 topic echo /odom
```

```
user@ubuntu:~$ ros2 topic echo /odom
header:
  stamp:
    sec: 288
    nanosec: 887000000
  frame_id: odom
child_frame_id: base_footprint
pose:
  pose:
    position:
      x: -4.788269541574665e-05
      y: 0.00016720098305932177
      z: -0.0010013931907763597
    orientation:
      x: -1.2123567868320229e-05
      y: 0.0038530909528901335
      z: 0.001313973112444194
      w: 0.9999917134745606
  covariance:
  - 1.0e-05
  - 0.0
  - 0.0
  - 0.0
  - 0.0
  - 0.0
  - 0.0
  - 1.0e-05
  - 0.0
  - 0.0
  - 0.0
  - 0.0
  - 0.0
  - 0.0
  - 1000000000000.0
  - 0.0
  - 0.0
  - 0.0
  - 0.0
  - 0.0
  - 0.0
  - 1000000000000.0
  - 0.0
  - 0.0
  - 0.0
  - 0.0
  - 0.0
  - 0.0
```

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 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
/gazebo/link_states
/gazebo/model_states
/gazebo/performance_metrics
/imu
/joint_states
/odom
/parameter_events
/rosout
/rosout_agg
/scan
/tf
```

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
10⁻³ ×

   -0.0256    0.0374    0.5800
```

```
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: [-2.5609e-05 3.7370e-05 5.7996e-04]
     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
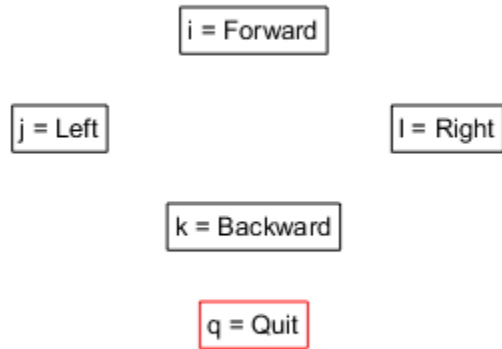
Keep this figure in scope to give commands
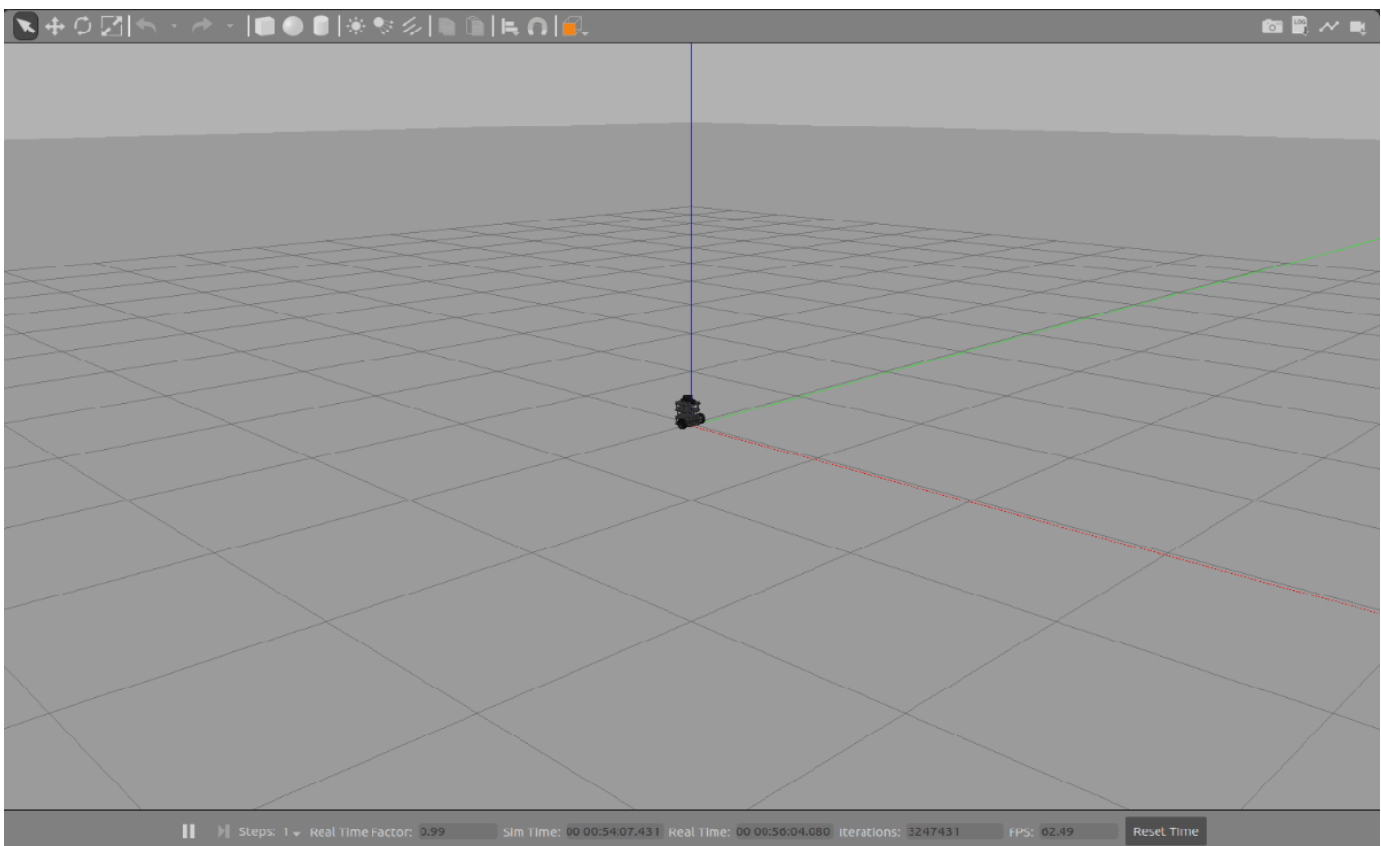
Plot Data Received from ROS

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');
```
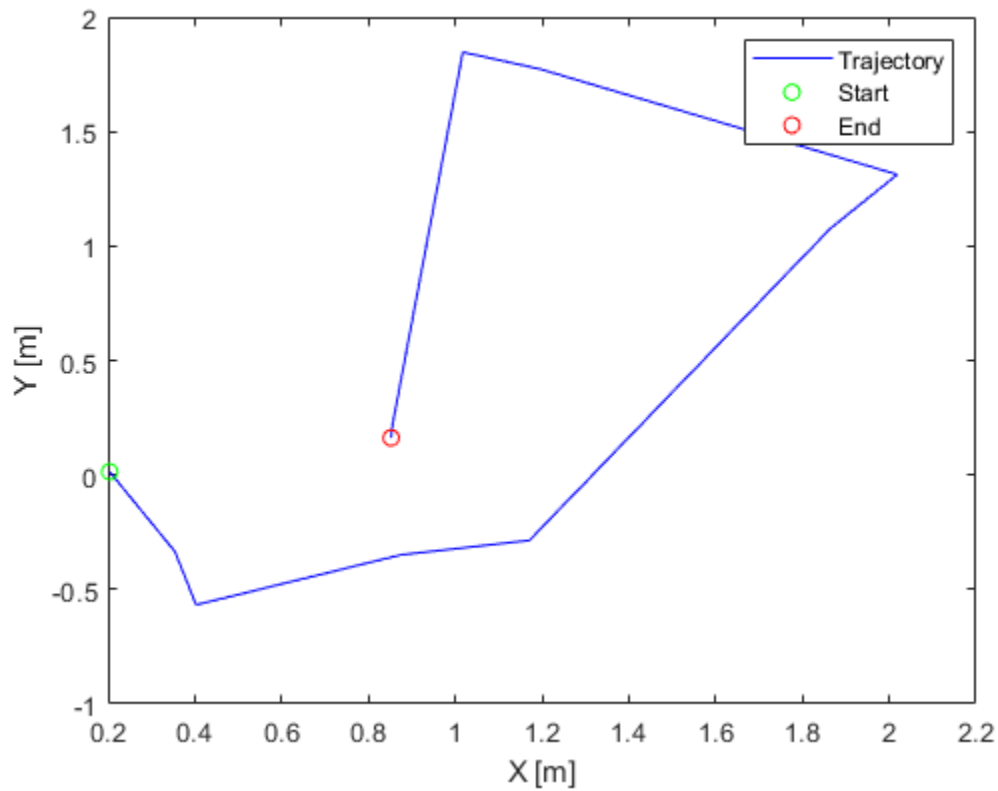

Clear the publishers and subscribers on the host.

```
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.
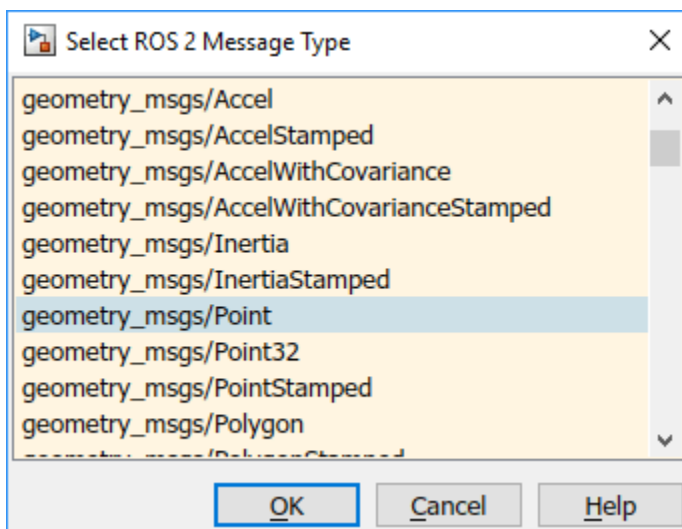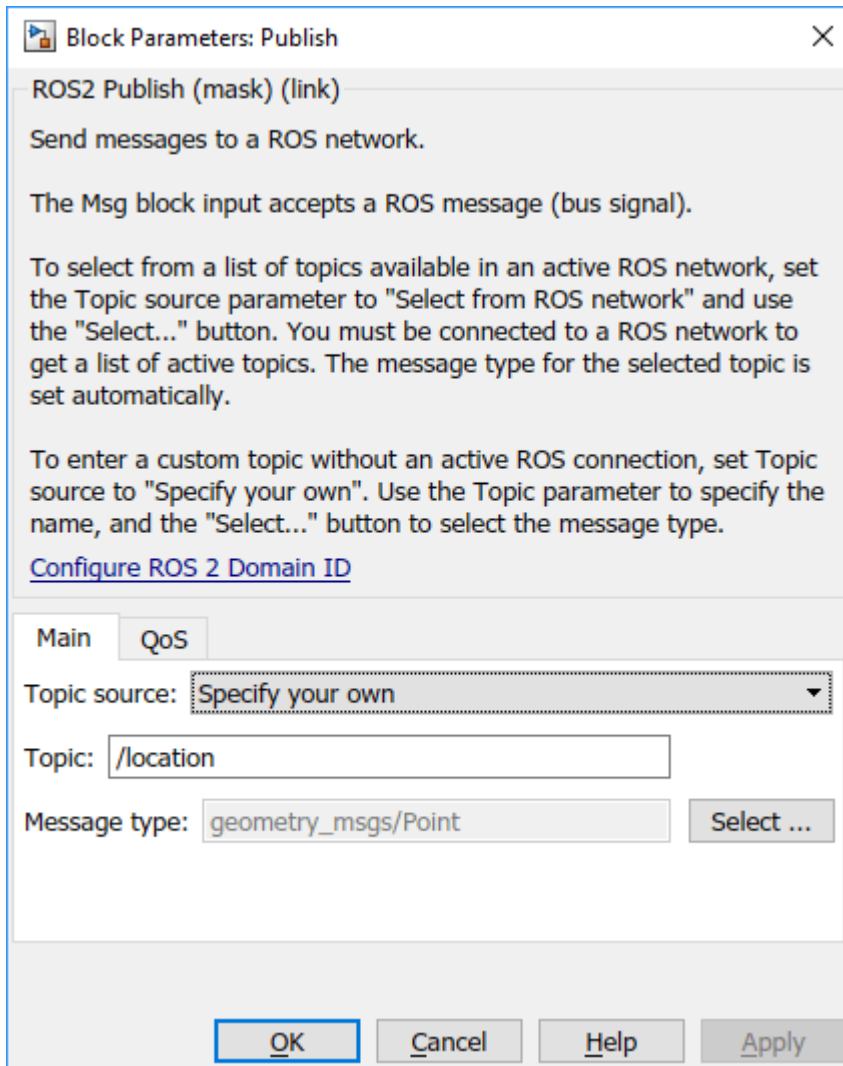
Create a ROS 2 Message

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 "Explore Simulink Bus Capabilities" (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.

2-75

- 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

Prerequisites

- "Get Started with ROS 2" on page 2-2
- "Exchange Data with ROS 2 Publishers and Subscribers" on page 2-19
- "Get Started with ROS 2 in Simulink®" on page 2-71

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.

- Download a virtual machine using the instructions in "Get Started with Gazebo and Simulated TurtleBot" on page 1-206.
- Open the *DEFAULT_FASTRTPS_PROFILE.xml* file in the home directory of the VM and replace <address> entries with host and VM IP addresses. To communicate under different subnets (see **Communicate Outside Subnet** section in "Connect to a ROS 2 Network" on page 2-8).

```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>
```

- In the Ubuntu desktop, click the "**Gazebo Empty**" icon to start the Gazebo world.
- Click on the "**ROS Bridge**" icon, to relay messages between ROS and ROS 2 network.
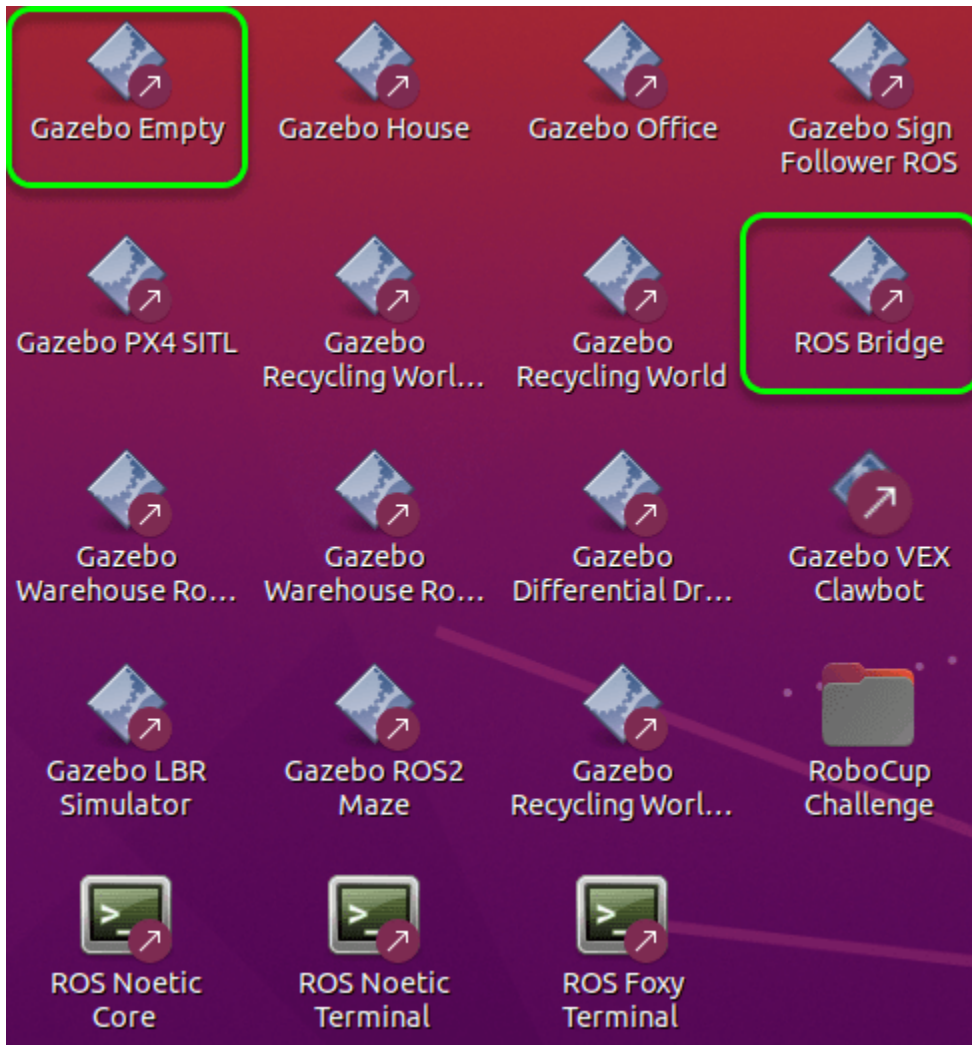
- In MATLAB on your host machine, set the proper domain ID for the ROS 2 network using the `'ROS_DOMAIN_ID'` environment variable to 25 to match the robot simulator ROS bridge settings and run `ros2 topic list` to verify that the topics from the robot simulator are visible in MATLAB.

```
setenv('ROS_DOMAIN_ID','25')
ros2 topic list
```
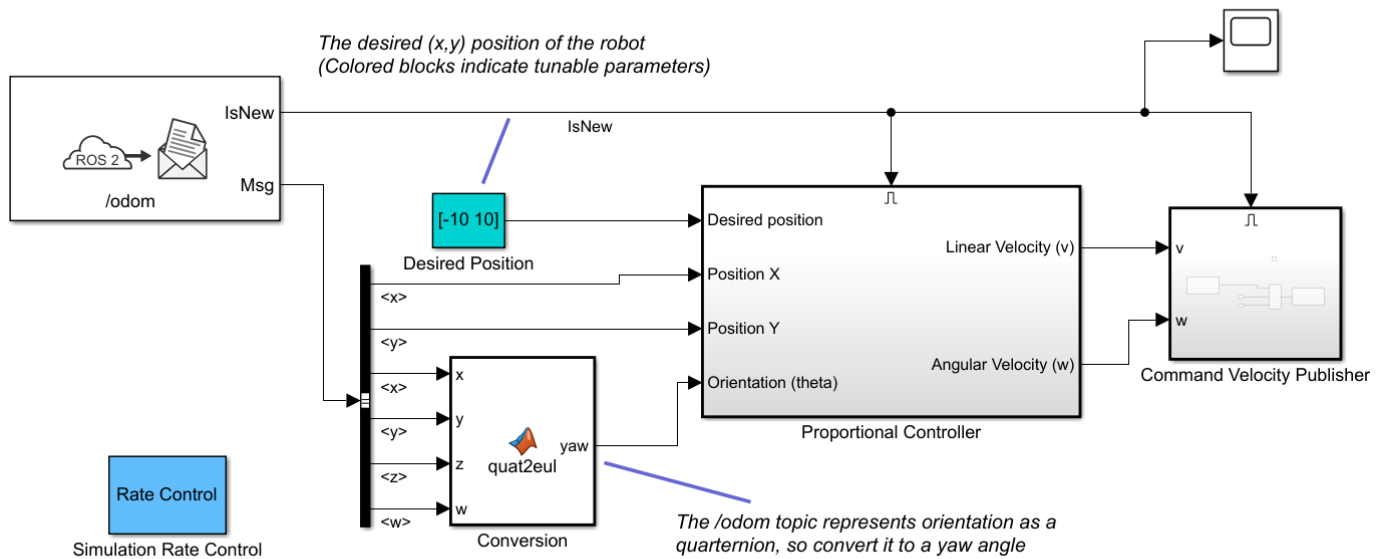
```
/clock
/gazebo/link_states
/gazebo/model_states
/gazebo/performance_metrics
/imu
/joint_states
/odom
/parameter_events
/rosout
/rosout_agg
/scan
/tf
```

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.

Open Existing Model

After connecting to the ROS 2 network, open the example model.

```
open_system('robotROS2ConnectToRobotExample');
```
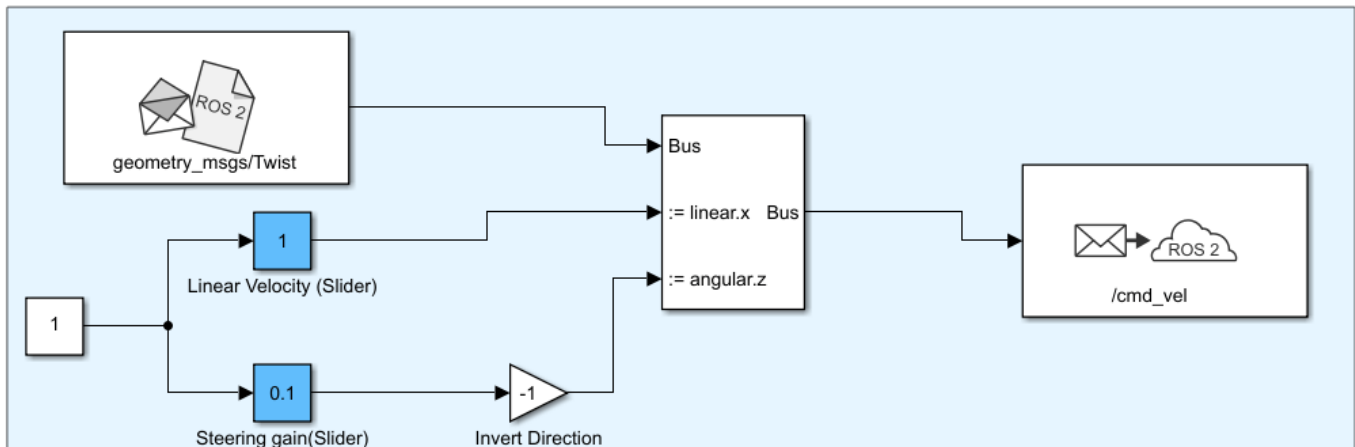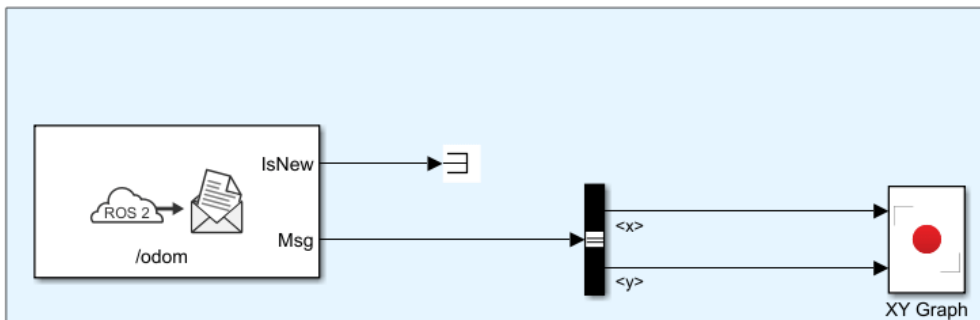

Copyright 2019 The MathWorks, Inc.

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 **ROS** tab, under **CONNECT**, 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
		Select>>	linear.x	Down
Signals in the bus			angular.z	
∨ linear		Refresh		Remove
x				
y				
z				
∨ angular				
x				
y				
z				

OK   Cancel   Help   Apply
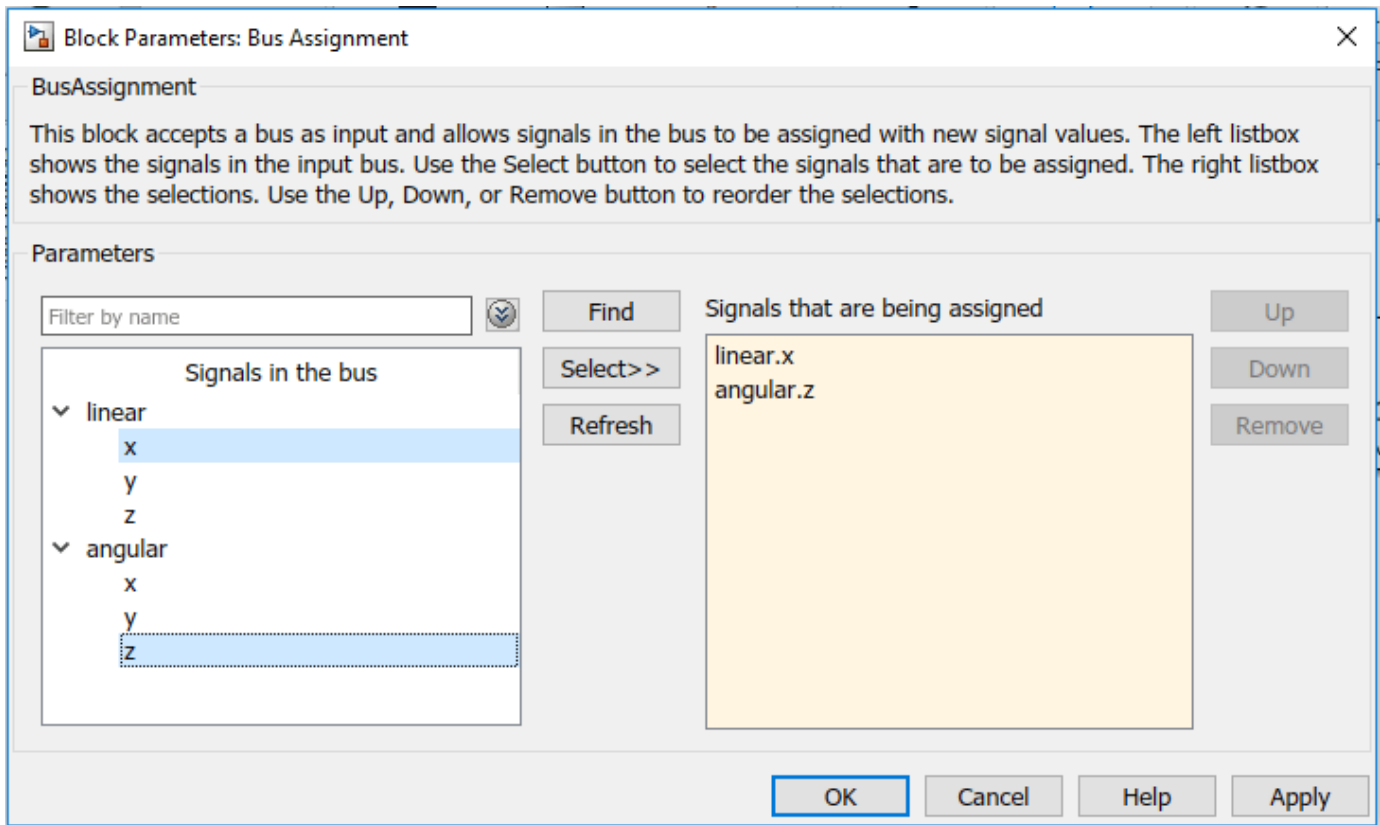
- Add a **Constant** block, a **Gain** block, and two **Slider Gain** blocks. Connect them together as shown in picture below, and set the **Gain** value to -1.
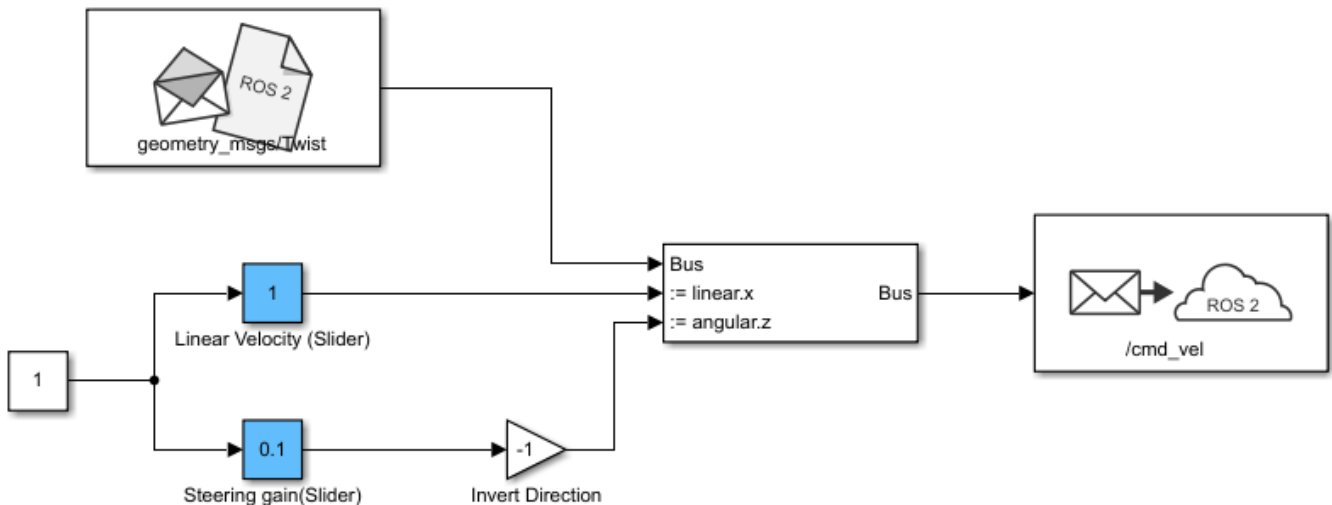
- Set the limits, and current parameters of the linear velocity slider to 0.0 to 2.0, and 1.0 respectively. Set the corresponding parameters of the steering gain slider to -1.0 to 1.0, and 0.1.

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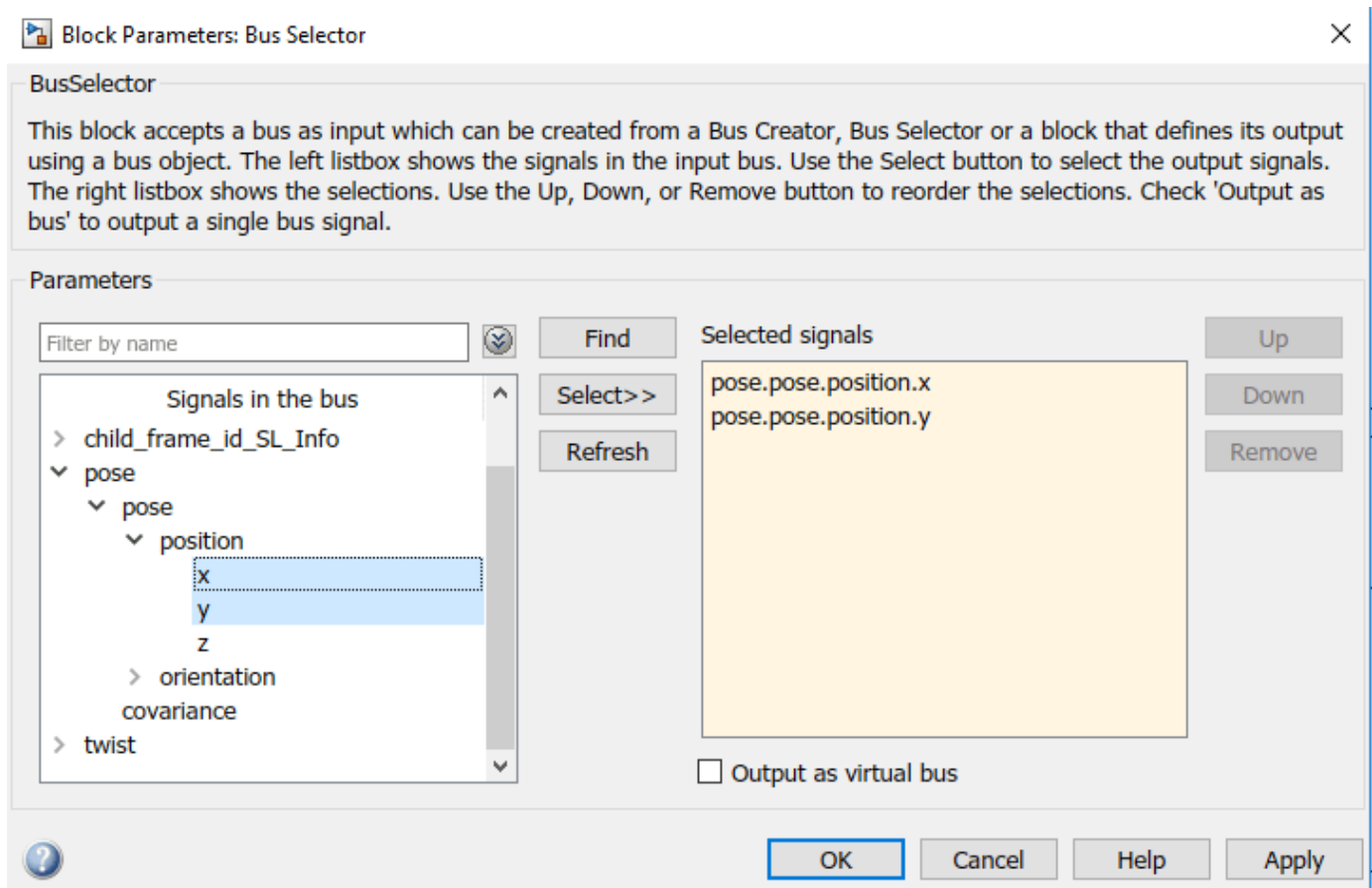
- From the **Simulink > Sinks** tab in the Library Browser, drag an **XY Graph** block to the model. Connect the X and Y output ports of the **Bus Selector** block to the input ports of the **XY Graph** block.

The following figure shows the completed model. A pre-configured model is included for your convenience.
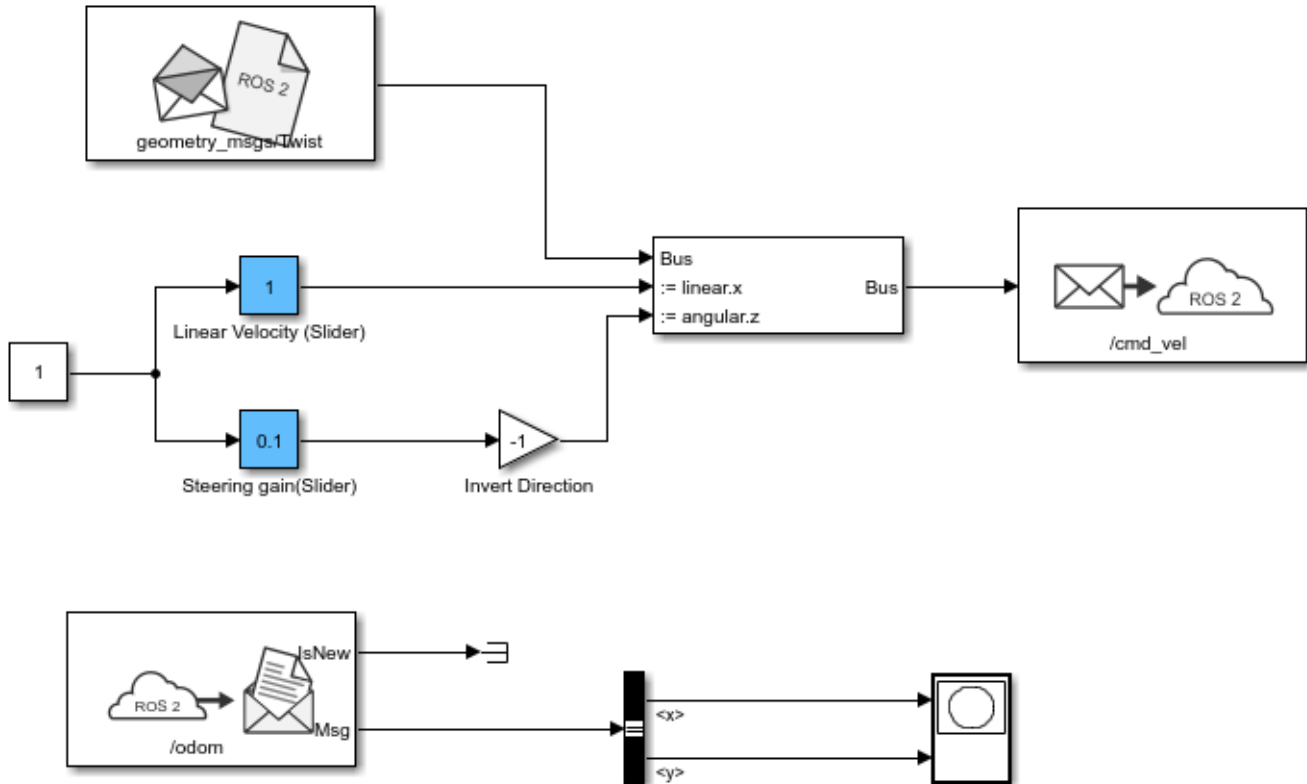
Task 5 - Configure and Run the Model

- 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.
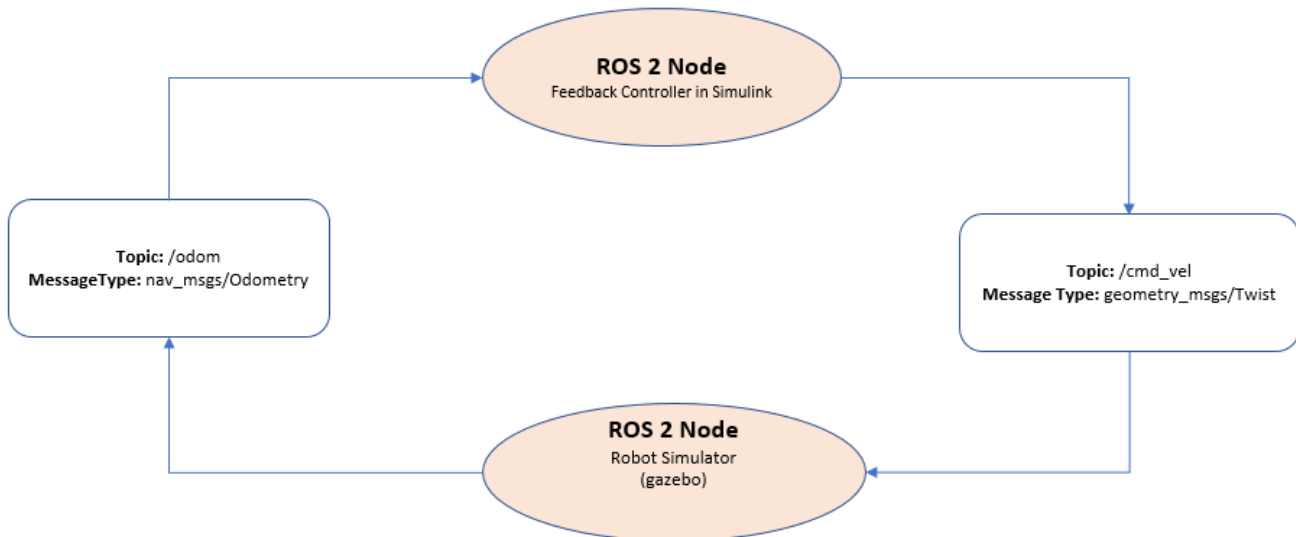
Start a Robot Simulator and Configure Simulink

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.

- Download a virtual machine using the instructions in "Get Started with Gazebo and Simulated TurtleBot" on page 1-206.
- Open the *DEFAULT_FASTRTPS_PROFILE.xml* file in the home directory of the VM and replace <address> entries with host and VM IP addresses. To communicate under different subnets, see **Communicate Outside Subnet** section in "Connect to a ROS 2 Network" on page 2-8.

```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>
```

- In the Ubuntu desktop, click the **Gazebo Empty** icon to start the empty Gazebo world.
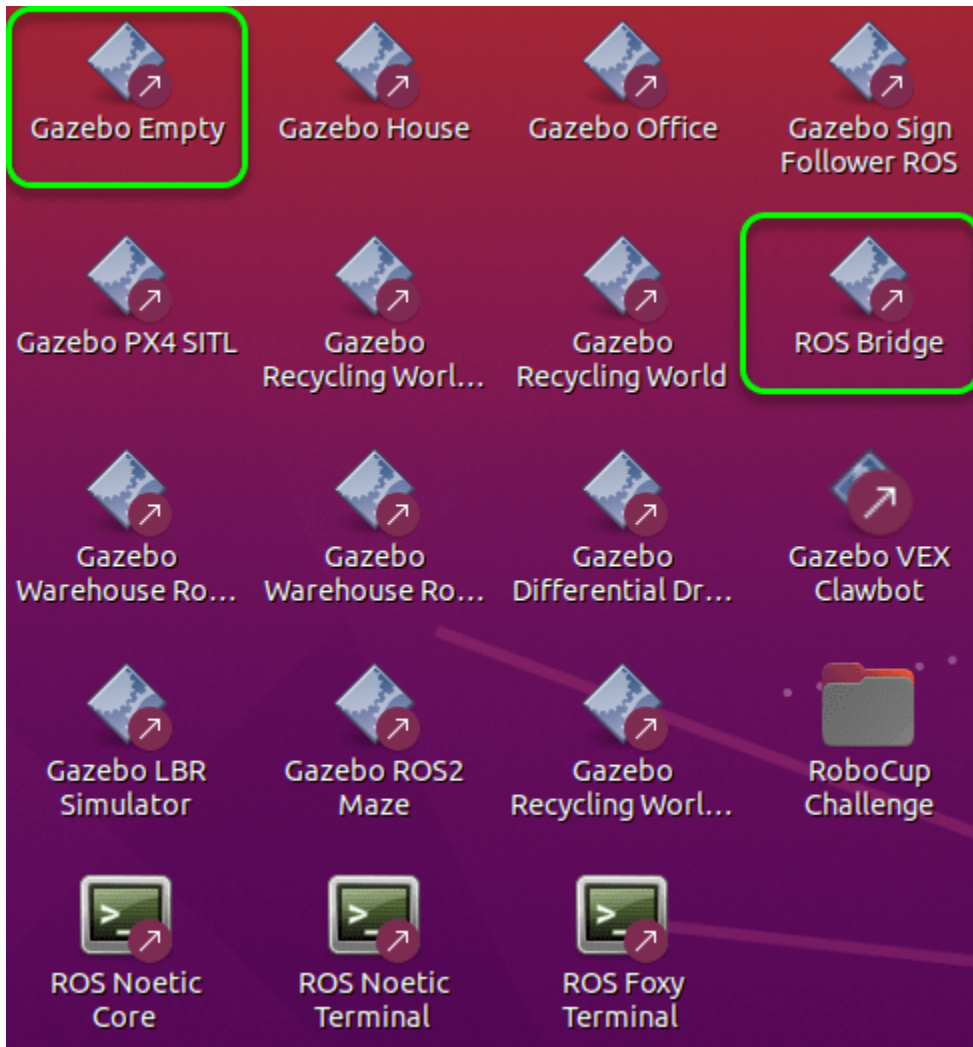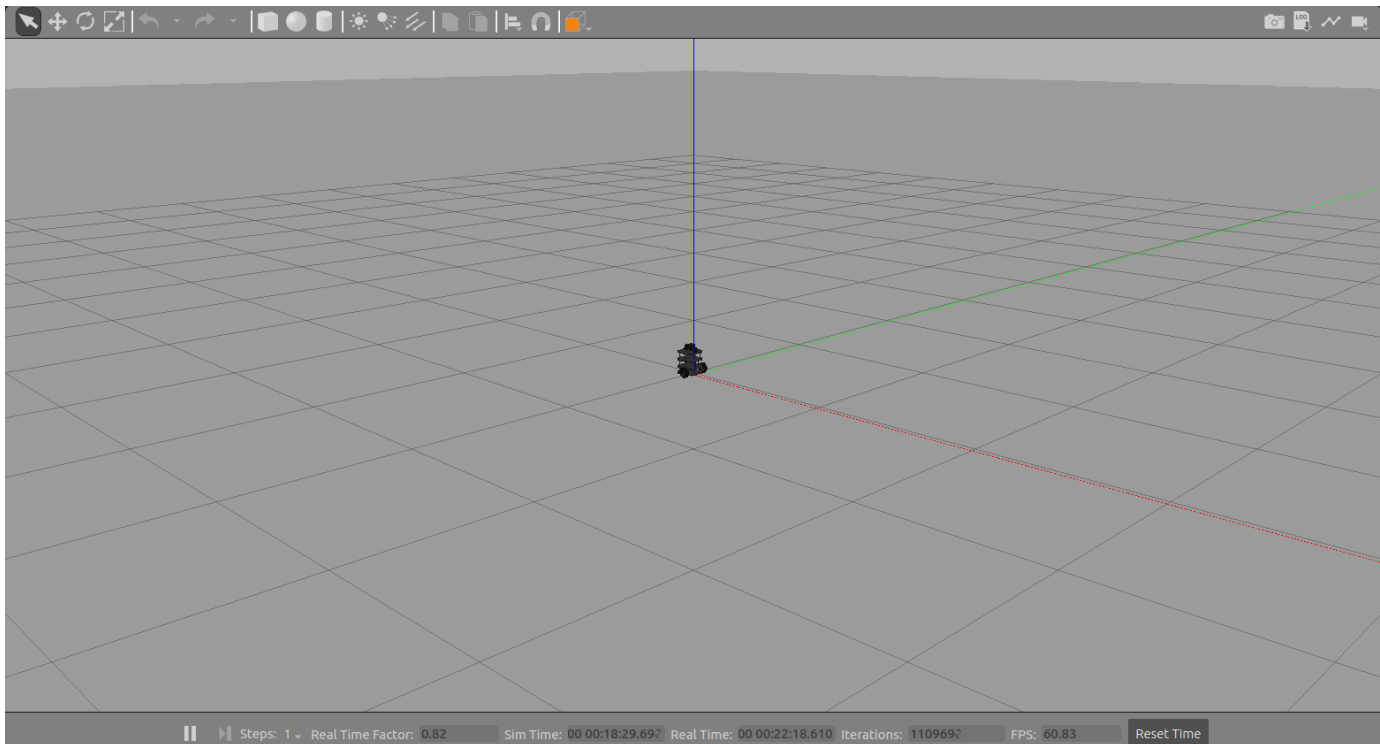
- Click the **ROS Bridge** icon to start the ROS bridge to relay messages between Simulink ROS 2 node and Turtlebot3 ROS-enabled robot.
- In MATLAB on your host machine, set the proper domain ID for the ROS 2 network using the `'ROS_DOMAIN_ID'` environment variable to 25 to match the robot simulator 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
/gazebo/link_states
/gazebo/model_states
/gazebo/performance_metrics
/imu
/joint_states
/odom
/parameter_events
/rosout
/rosout_agg
/scan
/tf
```

Open Existing Model

After connecting to the ROS 2 network, open the example model.

```
open_system('robotROS2FeedbackControlExample.slx');
```

Feedback Control of a ROS-enabled Robot over ROS 2

*The desired (x,y) position of the robot
(Colored blocks indicate tunable parameters)*

*The /odom topic represents orientation as a
quarternion, so convert it to a yaw angle*

Copyright 2019 The MathWorks, Inc.

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');
```

Proportional Controller
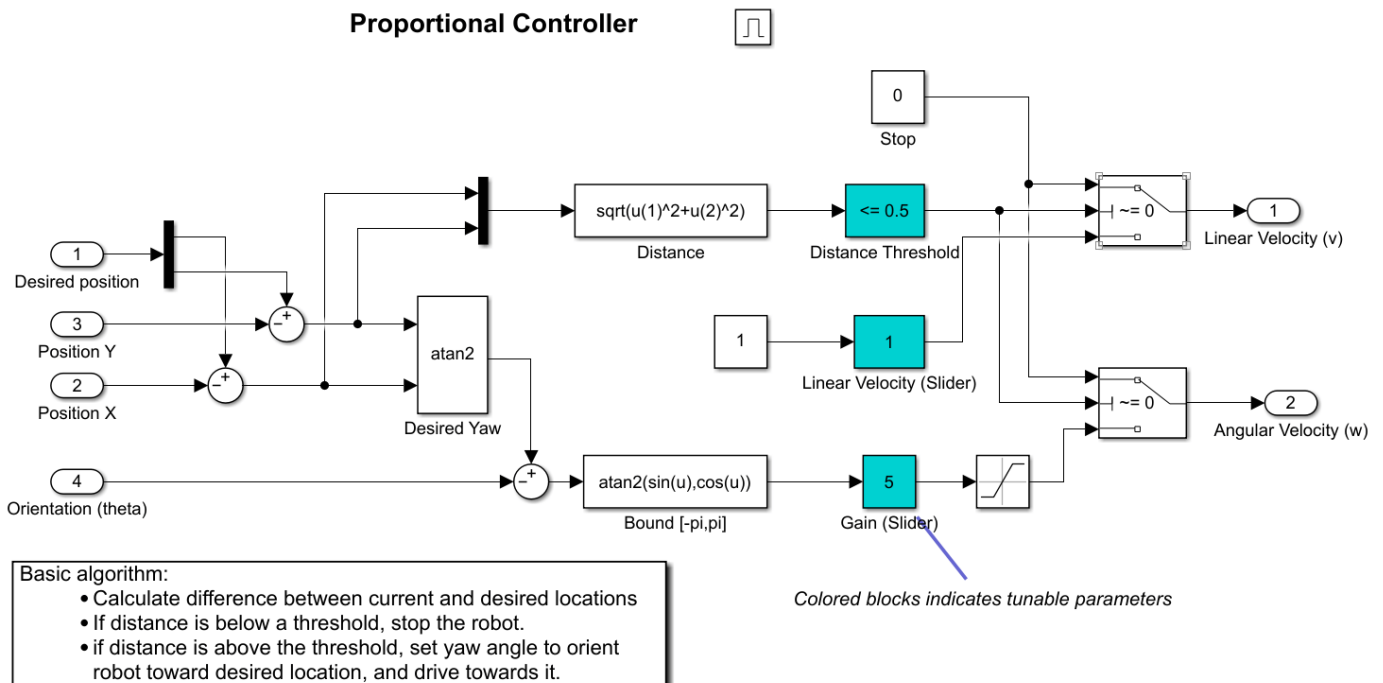
Colored blocks indicates tunable parameters

Basic algorithm:
• Calculate difference between current and desired locations
• If distance is below a threshold, stop the robot.
• if distance is above the threshold, set yaw angle to orient robot toward desired location, and drive towards it.

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.

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* and set the ROS 2 RMW Implementation as `rmw_fastrtps_cpp`.
- Click *OK* to apply changes and close the dialog.
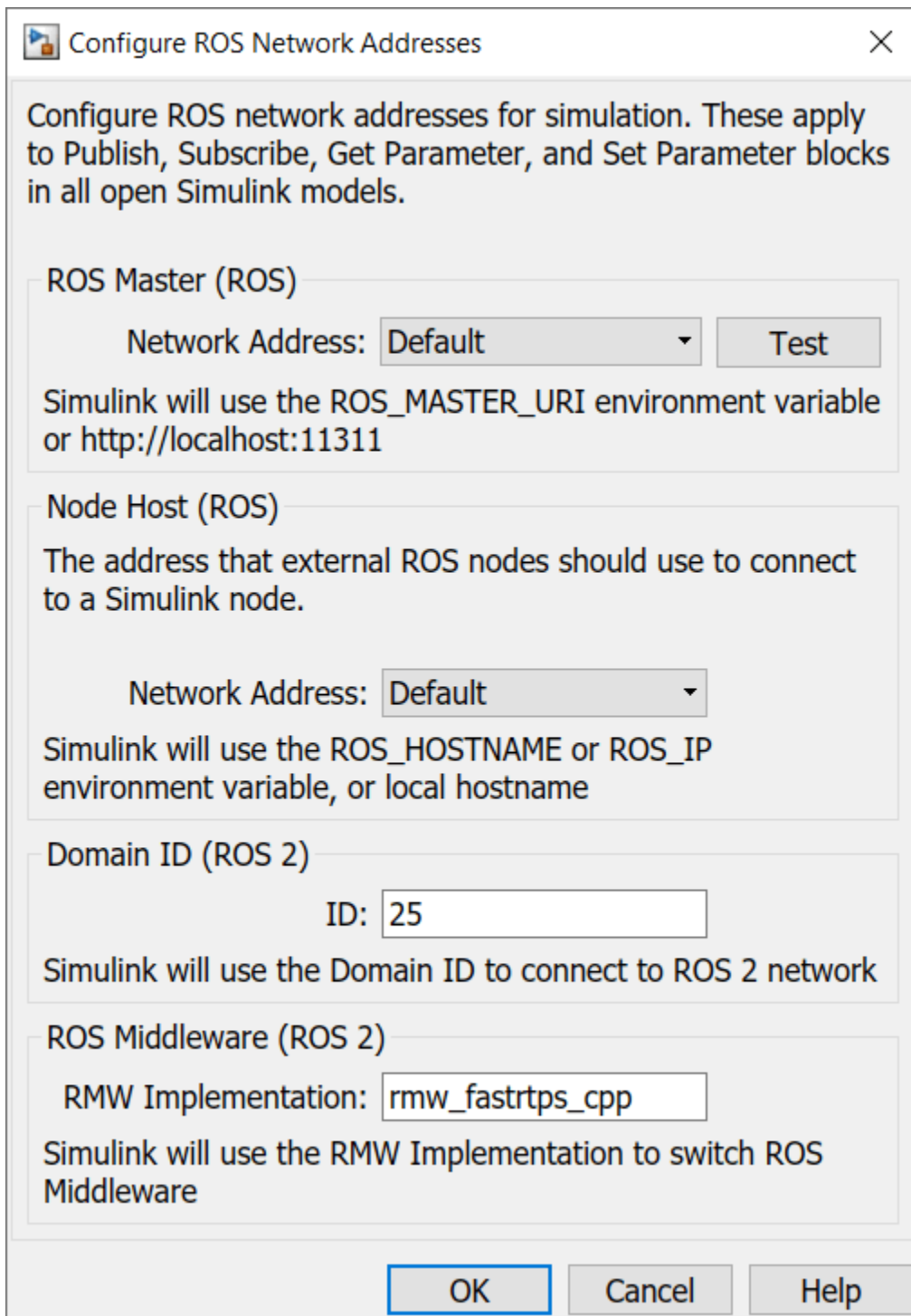
To run the model.

- 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.

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 2-71

```
open_system('simulinkPubSubROS2Example');
```
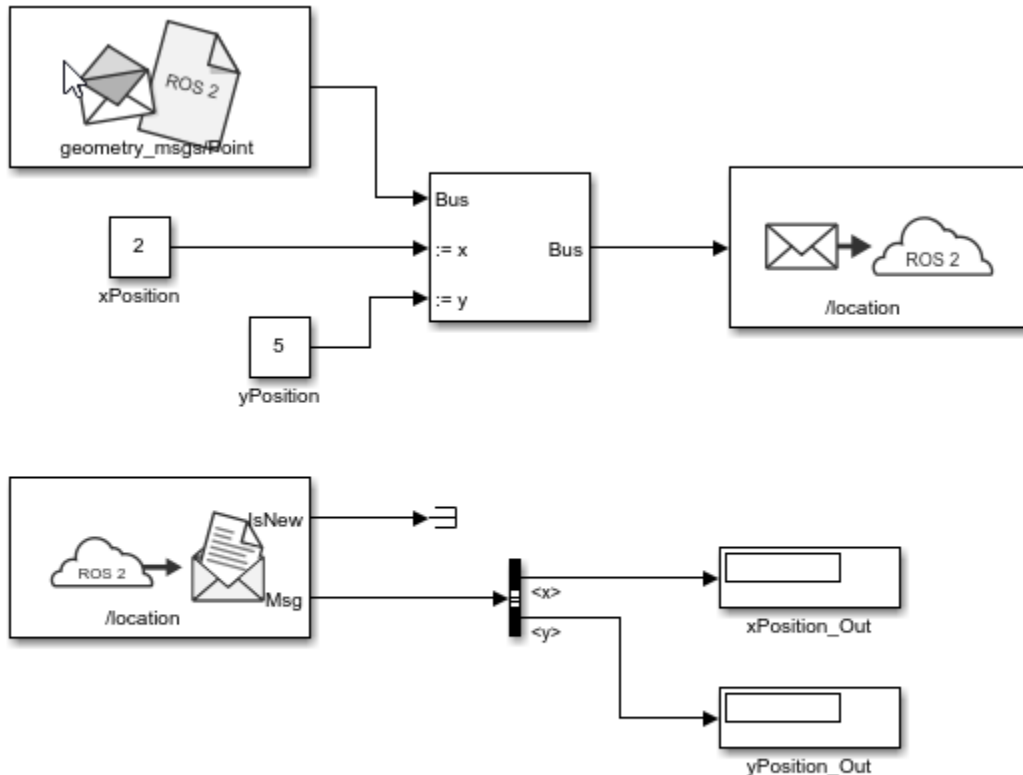
Copyright 2019 The MathWorks, Inc.

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.
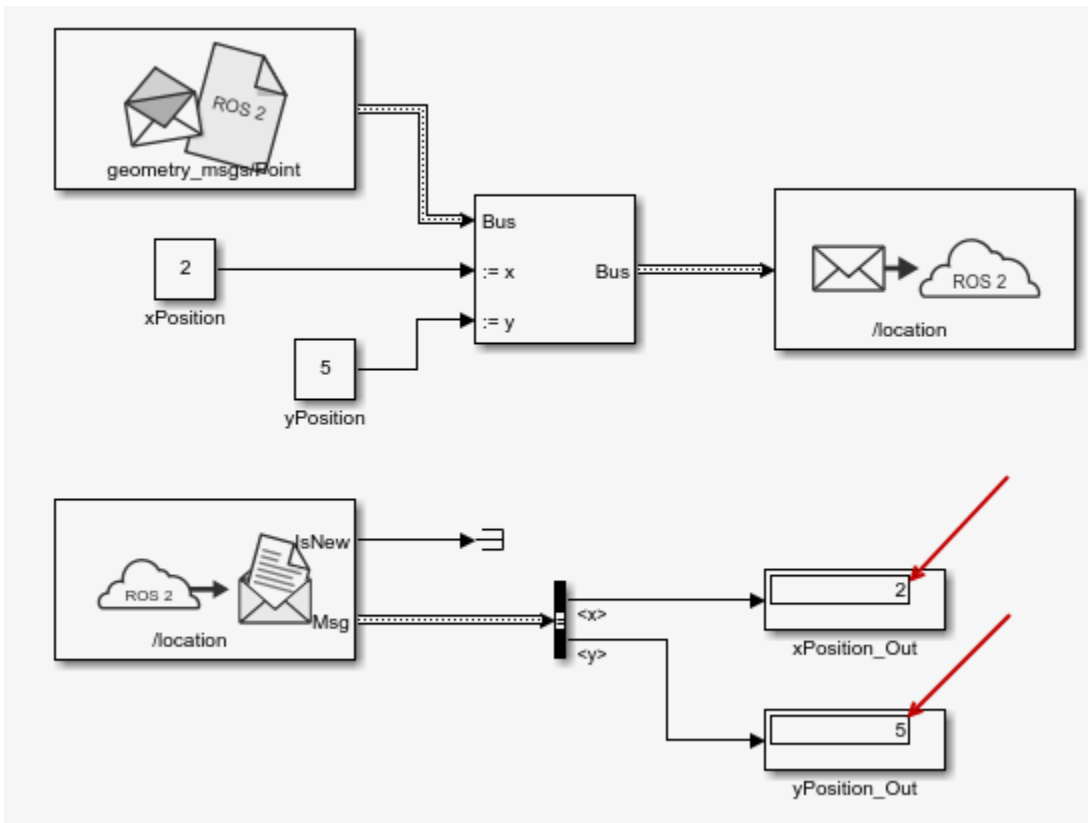
Generate a Standalone ROS 2 Node from Simulink

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™.
- Download a virtual machine using instructions in "Get Started with Gazebo and Simulated
  TurtleBot" on page 1-206.
- Review the "Feedback Control of a ROS-Enabled Robot Over ROS 2" on page 2-88 example.
- See ROS 2 Model Build Failure in "ROS Simulink Support and Limitations" on page 6-2.
- To ensure you have the proper third-party software, see "ROS Toolbox 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 2-88
example.

- Open the robot feedback control model configured for ROS 2. Alternatively, call
  `open_system("robotControllerROS2")`.
- Under **ROS** tab, in **Prepare**, 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.
- **Specify external ROS packages as dependencies:** To specify external ROS 2 packages as
  dependencies to the generated ROS 2 node, specify appropriate toolchain options. In the
  Configuration parameters, under **Code Generation > Toolchain Settings**, specify the **Build
  configuration** as **Specify** from the drop-down. Then, you can specify the **Required Packages,
  Include Directories, Link Libraries, Library Paths** and **Defines** based on the external ROS 2
  packages that you wish to integrate with the generated ROS 2 node.
- Click **OK**.

Configure the Connection to the ROS 2 Device

A **ROS 2 device** is any Linux system that has ROS 2 installed and is capable of building and running
a ROS 2 node. If you have Simulink Coder, you can generate code for a standalone ROS 2 node. If
your system is connected to a ROS 2 device, Simulink can also transfer the generated code to the
ROS 2 device, build an executable, and run the resulting ROS 2 node (this is referred to as
"deploying" the ROS node).

In this task, you decide if you want to generate code for the ROS 2 node or if you want to build and run it on a ROS 2 device. If you are connected to a ROS 2 device, you can configure Simulink to use it as a deployment target for your ROS 2 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 2 node, without building it on an external ROS 2 device. *Build and load* generates the code, transfers it to an external device and builds a ROS 2 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 2 device. Under the **ROS** tab, from the **Deploy to** drop-down, click **Manage Remote Device**. This opens the **Connect to a ROS device** dialog. In this dialog, you can enter all the information that Simulink needs to deploy the ROS 2 node. This includes the IP address or host name of your ROS 2 device, your login credentials, and the Catkin workspace. Change **Catkin workspace** to ~/ros2_ws_test.

ROS 2 Folder is the location of the ROS 2 installation on the ROS 2 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 2 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 ~/ros2_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 2 node.
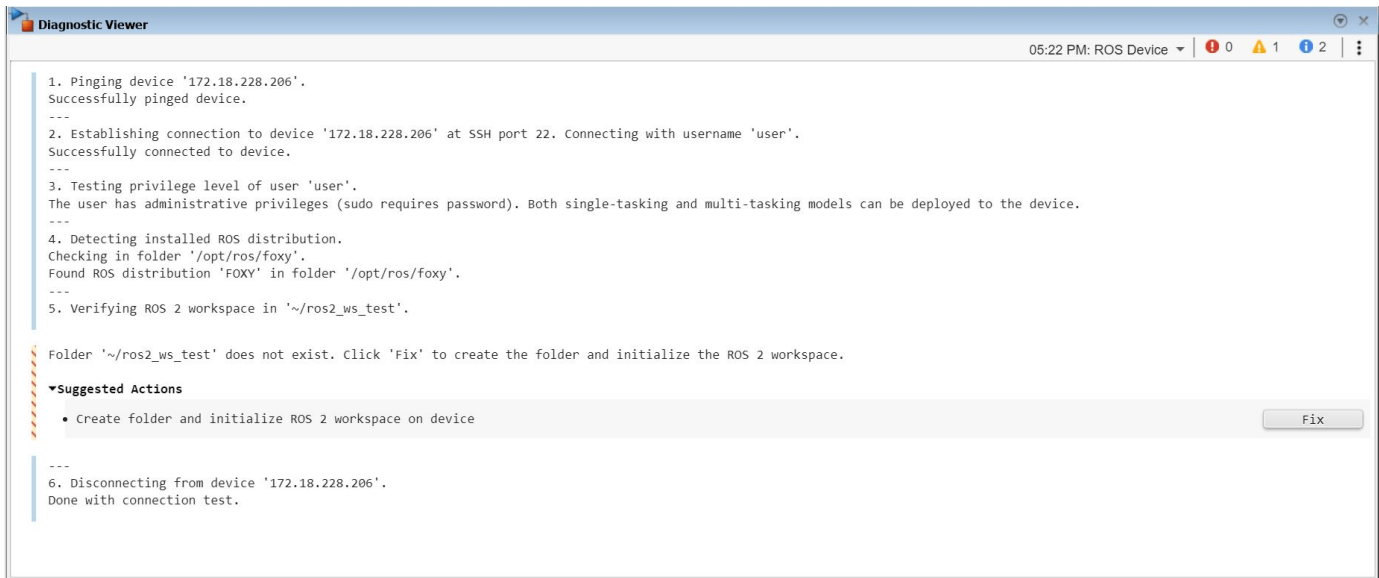
- Change the device connection settings and test them until no other warnings or errors are shown. If an automatic fix to your settings is possible, Simulink suggests it by displaying the **Fix** button. Once you have a good set of settings, click **OK** in the connection settings dialog to save the settings.

The connection settings are not specific to a single model, but apply to all ROS 2 models in Simulink.

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.
- If you are deploying to a local device, set the environment variable ROS_DOMAIN_ID to the value 25, using setenv in MATLAB.
- Set the ROS Middleware (ROS 2) of ROS 2 network. This example uses ROS Middleware as rmw_cyclonedds_cpp.
- If you are deploying to a local device, set the environment variable RMW_IMPLEMENTATION to rmw_cyclonedds_cpp, using setenv in MATLAB.
- In **ROS** 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 in a synchronous fashion based on the sample time of the model. 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
```

Specify Code Generation and Build Options

You can choose different code generation and build behaviors by specifying one of these **Deployment** options from the toolstrip under **ROS** tab.

- **Generate code** — Generates the ROS package source code on localhost or remote device.
- **Build Model** — Generates the ROS package source code and builds the standalone executable on localhost or remote device.
- **Build & Run** — Generates the ROS package source code, builds the standalone executable and starts running it on localhost or remote device.

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 2-99.

Prerequisites

- This example requires Simulink Coder™ and Embedded Coder™ .
- If this is your first time deploying a ROS node, check the "ROS Toolbox 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 Simulated TurtleBot" on page 1-206 for instructions on setting up a simulated robot.
- Review the "Feedback Control of a ROS-Enabled Robot Over ROS 2" on page 2-88 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 2-88 example.

- Open the robot feedback control model configured for ROS 2.

```
open_system("robotFeedbackControllerROS2");
```

- Under **ROS** 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 **ROS** 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.
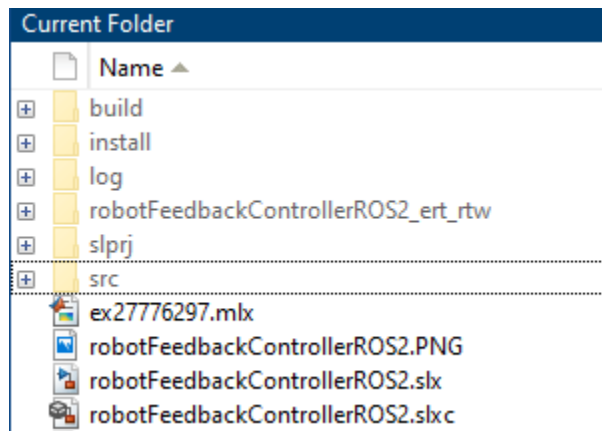
Compress the `src` folder to a tar file by executing the following command in MATLAB Command Window:

```
>> 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 Simulated TurtleBot" on page 1-206. 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 ROS 2 workspace and decompress the source code. You may use an existing ROS 2 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_ros2_ws>` with the path to your ROS 2 workspace. In this example, the `<path_to_ros2_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). If the node uses custom messages, you must manually copy the necessary custom message packages in `<path_to_ros2_ws>/msg` folder before building the node.

```
cd <path_to_ros2_ws>
source /opt/ros/foxy/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_ros2_ws>` with the path to your ROS 2 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/foxy/local_setup.sh
~/<path_to_ros2_ws>/install/robotfeedbackcontrollerros2/lib/robotfeedbackcontrollerros2/robotFeed
```

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 how to control a simulated robot running on a ROS-based simulator over a ROS network. You generate a ROS node for the control algorithm and deploy it to the remote device running ROS. The example shown here uses ROS and MATLAB for simulation, and MATLAB Coder™ for code generation and deployment. For the other examples with ROS 2 or Simulink®, see:

- "Sign Following Robot with ROS in Simulink" on page 2-115
- "Sign Following Robot with ROS 2 in MATLAB" on page 2-121
- "Sign Following Robot with ROS 2 in Simulink" on page 2-127

In this example, you implement a sign-following algorithm and control the simulated robot to follow a path based on signs in the environment. The algorithm receives the location and camera information from the simulated robot. The algorithm detects the color of the sign and sends velocity commands to turn the robot based on the color. In this example, the robot turns left when it encounters a blue sign and right when it encounters a green sign. The robot stops when it encounters a red sign.

Connect to Robot Simulator

Start a ROS-based simulator for a differential-drive robot and configure a MATLAB® connection with the robot simulator.

Download a virtual machine using instructions in "Get Started with Gazebo and Simulated TurtleBot" on page 1-206, then follow these steps.

1. Start the Ubuntu® virtual machine desktop.
2. Click the **Gazebo Sign Follower ROS** icon to start the Gazebo world for this example.
3. 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://172.18.228.122:11311` and your host computer address is `192.168.31.1`.
4. Start the ROS 1 network using `rosinit`

```
masterIP = '172.18.228.122';
rosinit(masterIP,11311);
```

Initializing global node /matlab_global_node_12033 with NodeURI http://172.18.228.249:60144/ and

Configure 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`. The `rossubscriber` and `rospublisher` functions support code generation for message structures only. To return a message structure, specify the name-value pair argument, `"DataFormat","struct"`, when creating subscribers and publishers.

```
imgSub = rossubscriber("/camera/rgb/image_raw","sensor_msgs/Image",DataFormat = "struct");
odomSub = rossubscriber("/odom","nav_msgs/Odometry",DataFormat = "struct");
[velPub, velMsg] = rospublisher("/cmd_vel", "geometry_msgs/Twist",DataFormat = "struct");
```

Create a `rosbagwriter` object to log the subscribed image and odometry data that the robot publishes.

```
bagWriter = rosbagwriter("sign_following_data.bag")

bagWriter =
  rosbagwriter with properties:

       FilePath: 'Z:\32\psahu.Bdoc23a.j2115093\sign_following_data.bag'
      StartTime: 0
        EndTime: 0
    NumMessages: 0
    Compression: 'uncompressed'
      ChunkSize: 786432                                              Bytes
       FileSize: 4117                                                Bytes
```

Define the image processing color threshold parameters. Each row defines the threshold values for the corresponding color.

```
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 a helper MATLAB Stateflow® chart that takes as inputs the image size, coordinates from a processed image, and robot odometry poses. The chart returns the linear and angular velocity to drive the robot.

```
controller = ExampleHelperSignFollowingControllerChart;
open("ExampleHelperSignFollowingControllerChart");
```
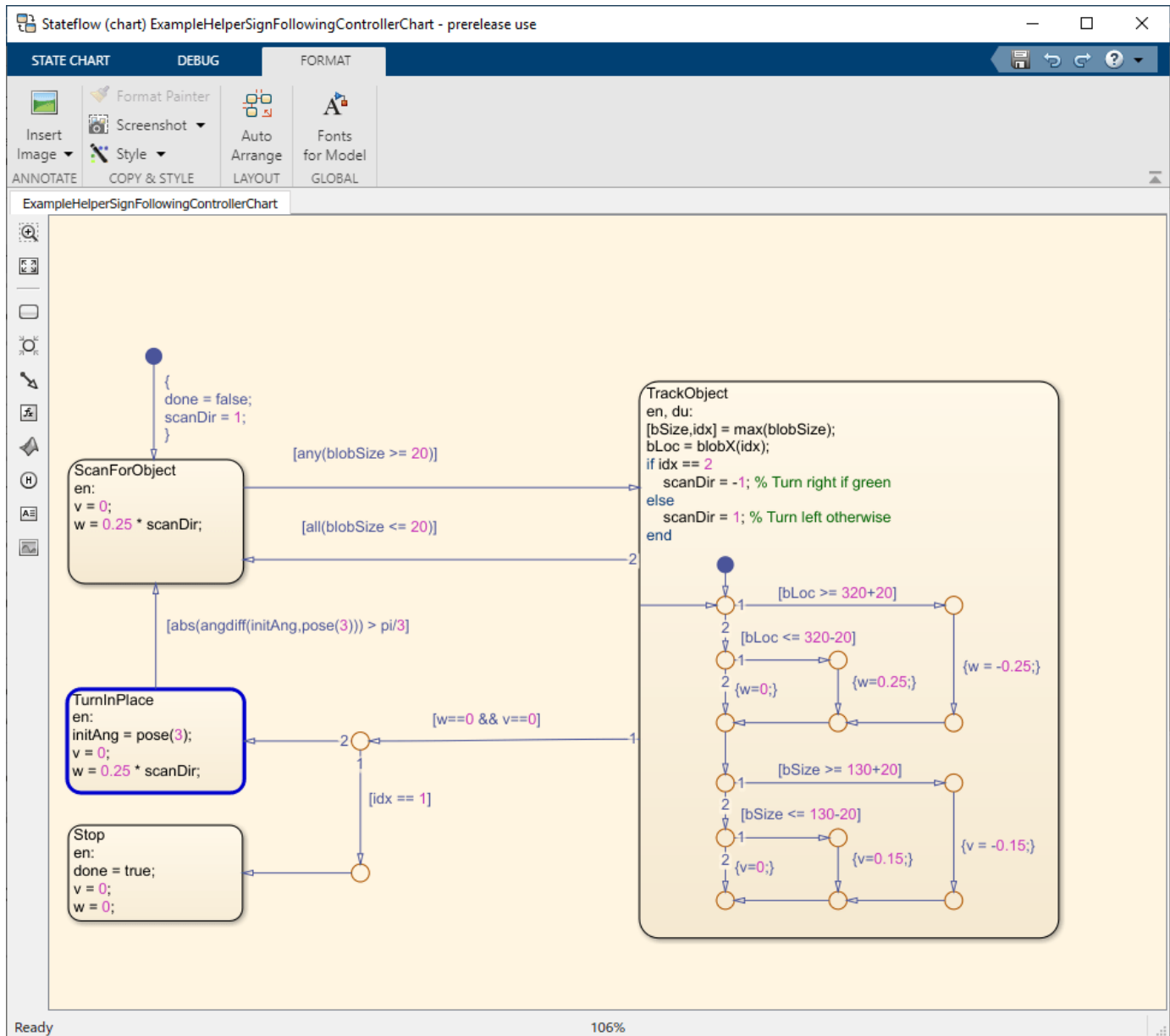
Run Control Loop

Run the controller to receive images and move the robot to follow the sign. The controller performs these steps:

- Get the latest image and odometry message from the ROS network.
- Run the algorithm for detecting image features (`ExampleHelperSignFollowingProcessImg`) function.
- Generate control commands from the Stateflow® chart using `step`.
- Publish the velocity control commands to the ROS network.

To visualize the masked image that the robot sees, change the value of `doVisualization` to `true`.

```matlab
ExampleHelperSignFollowingSetupPreferences;
doVisualization = false;
r = rosrate(10);
receive(imgSub);
receive(odomSub);
while(~controller.done)
    % Get the latest sensor messages and process them.
    imgMsg = imgSub.LatestMessage;
    odomMsg = odomSub.LatestMessage;
    [img,pose] = ExampleHelperSignFollowingROSProcessMsg(imgMsg, odomMsg);

    % Log the subscribed image and odometry data into a bag file.
    currentTime = rostime("now");
    write(bagWriter,imgSub.TopicName,currentTime,imgMsg);
    write(bagWriter,odomSub.TopicName,currentTime,odomMsg);

    % Run the 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 the 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
```

The robot follows the signs and stops at the final STOP sign. This figure shows the robot in motion.
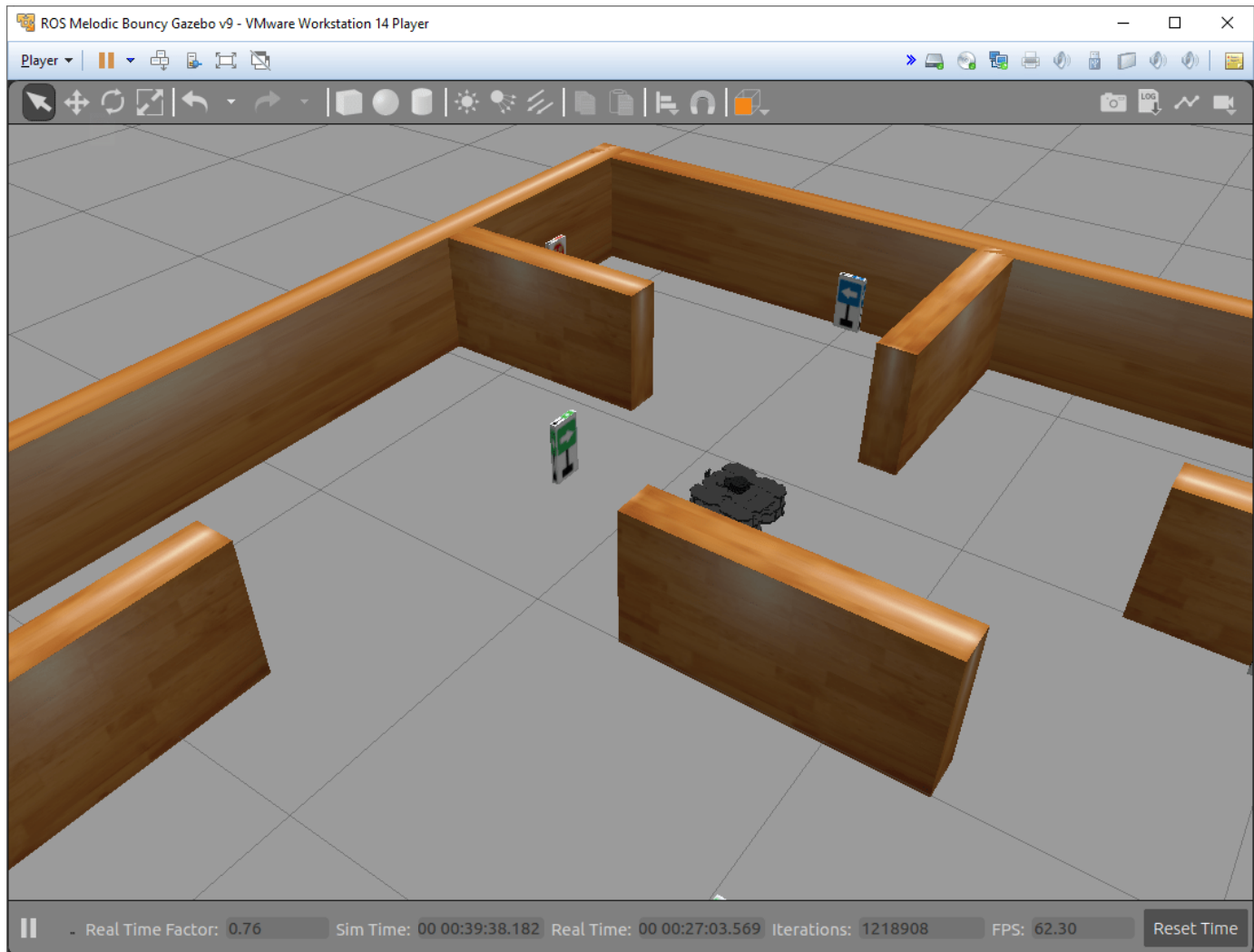
Delete the `rosbagwriter` object to close the bag file and reset the Gazebo scene after simulation using the `/gazebo/reset_simulation` service. Create a `rossvcclient` object for the service. Use the `call` object function to call the service and reset the Gazebo simulation scene.

```
delete(bagWriter);
gazeboResetClient = rossvcclient('/gazebo/reset_simulation',DataFormat = "struct");
call(gazeboResetClient);
```

Visualize Logged ROS Messages

Open the ROS Bag Viewer app by entering `rosbagViewer` in the MATLAB® Command Window. You can also launch the app from the Apps section from the MATLAB toolstrip.

Click **Open** and load the generated `sign_following_data.bag` file. Inspect the **Topic List** and **Source Details** panel on the left side of the app to check the bag file information. Launch **Image** and **Odometry** visualizers from the toolstrip and select data sources. Use the **Playback** panel controls to play the bag file and visualize the logged messages.
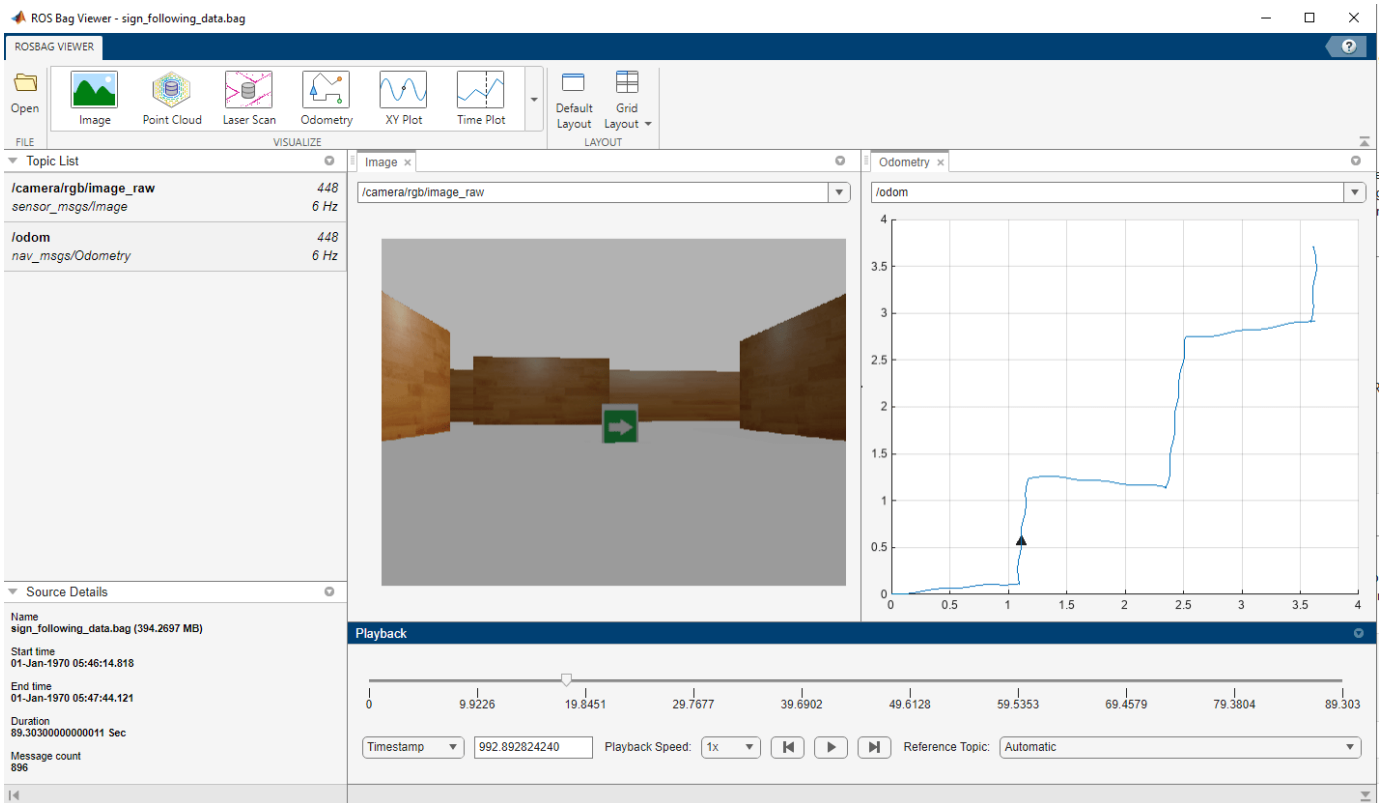
Generate and Deploy ROS Node

After you verify the controller, you can generate a ROS node for the sign-following robot algorithm using MATLAB Coder™ software. You can then deploy this node on the remote virtual machine running Gazebo. Using deployment, you can run ROS nodes on remote machines directly, resulting in faster executions. Create a MATLAB Coder configuration object that uses "Robot Operating System (ROS)" hardware.

```
cfg = coder.config("exe");
cfg.Hardware = coder.hardware("Robot Operating System (ROS)");
```

Set configuration parameters. You can adjust these values so they are suitable for your remote device and verify them before deployment. The robot follows the signs and stops at the final STOP sign.

```
cfg.Hardware.DeployTo = "Remote Device";
cfg.Hardware.RemoteDeviceAddress = '192.168.192.129';
cfg.Hardware.RemoteDeviceUsername = "user";
cfg.Hardware.RemoteDevicePassword = "password";
```

Set the build action to "Build and Run" to ensure that the deployed ROS node starts running after code generation.

```
cfg.Hardware.BuildAction = "Build and run";
```

Generate the ROS node and deploy the controller. The DeploySignFollowingRobotROS function contains the controller algorithm code.

```
codegen DeploySignFollowingRobotROS -config cfg
```

2-113

Reset the Gazebo scene after the node executes by calling the `rossvcclient` object, `gazeboResetClient`.

```
call(gazeboResetClient);
```

Rerun Deployed Node Using `rosdevice`

To rerun the deployed ROS node, create a `rosdevice` object. Specify the `deviceAddress,` `username,` and `password` inputs to establish an SSH connection between the ROS device and MATLAB.

```
gazeboVMDevice = rosdevice('192.168.192.129',"user","password");
```

Check the available nodes on the connected remote device. Verify that the deployed ROS node, `deploysignfollowingrobotros,` exists on the remote device .

```
gazeboVMDevice.AvailableNodes
```

Run the ROS node on the remote device using `runNode` function. The robot follows the signs and stops at the final STOP sign.

```
runNode(gazeboVMDevice,'deploysignfollowingrobotros')
```

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 2-108.

Start Robot Simulator

Start a ROS-based simulator for a differential-drive robot and configure the Simulink® connection with the robot simulator.

To follow along with this example, download a virtual machine using instructions in "Get Started with Gazebo and Simulated TurtleBot" on page 1-206.

- 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 example model.

```
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');
```

2-115

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 Using YOLOv2 Detection Algorithm with ROS in Simulink

This example shows how to use Simulink® to control a simulated robot running on a separate ROS-based simulator. It then shows how to generate CUDA-optimized code for the ROS node, from the Simulink model and deploy it to the localhost device.

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 using a YOLO v2 detector 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 2-108.

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). For instructions on downloading the virtual machine, see "Get Started with Gazebo and Simulated TurtleBot" on page 1-206.

- 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.

Get Pretrained Sign Detection Network

This function downloads the YoloV2DirectionSignRecognitionModel.mat file if is is not already present.

exampleHelperGetYoloV2DirectionSignRecognitionNetwork()

Downloading pretrained lane detection network (58b MB)...

Open Model and Configure Simulink

Setup the Simulink ROS preferences to communicate with the robot simulator.

Open the example model.

open_system('signFollowingRobotYOLOv2ROS.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 using a pretrained YOLO v2 object detector in the **Deep Learning Object Detection** subsystem of the model.

```
open_system('signFollowingRobotYOLOv2ROS/Deep Learning Object Detection');
```

The **Sign Tracking Logic** subsystem implements a Stateflow® chart that takes in the detected image size and coordinates from **Deep Learning Object Detection** and provides linear and angular velocity to drive the robot.

```
open_system('signFollowingRobotYOLOv2ROS/Sign Tracking Logic');
```

Run the Model

Open Configuration Parameters dialog of the `signFollowingRobotYOLOv2ROS` model. In **Simulation Target** pane, select **GPU acceleration**. In the **Deep Learning** group, select the target library as **cuDNN**.

```
set_param('signFollowingRobotYOLOv2ROS','GPUAcceleration','on');
set_param('signFollowingRobotYOLOv2ROS','SimDLTargetLibrary','cudnn');
set_param('signFollowingRobotYOLOv2ROS','DLTargetLibrary','cudnn');
```

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.
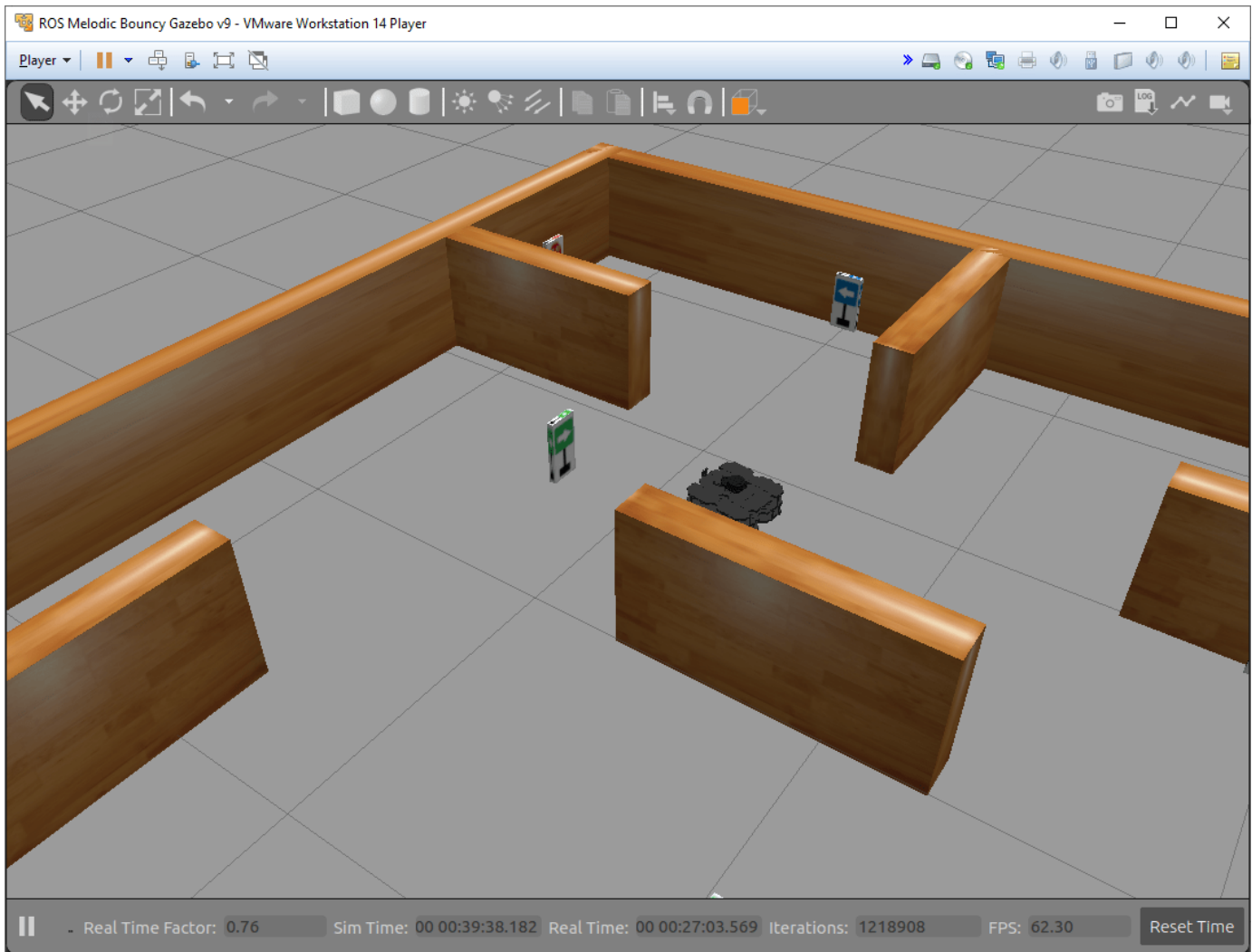
Generate and Deploy CUDA-Optimized ROS Node

Third-Party Prerequisities

- CUDA enabled NVIDIA GPU.
- NVIDIA CUDA toolkit and driver.
- NVIDIA cuDNN library.
- Environment variables for the compilers and libraries. For more information, see "Installing Prerequisite Products" (GPU Coder) and "Setting Up the Prerequisite Products" (GPU Coder).

Configure the Simulink Model for GPU code generation

Open Configuration Parameters dialog of the `signFollowingRobotYOLOv2ROS` model. In **Hardware Implementation** pane, select **Robot Operating System (ROS)** for **Hardware Board** and specify the appropriate **Device Vendor** and **Device Type**. Then, in **Code Generation** pane, select the **Language** as **C++** and enable **Generate GPU code**.

```
set_param('signFollowingRobotYOLOv2ROS','TargetLang','C++');
set_param('signFollowingRobotYOLOv2ROS','GenerateGPUCode','CUDA');
```

In **Code Generation > Libraries > GPU Code** pane, enable **cuBLAS**, **cuSOLVER** and **cuFFT**.

```
set_param('signFollowingRobotYOLOv2ROS','GPUcuBLAS','on');
set_param('signFollowingRobotYOLOv2ROS','GPUcuSOLVER','on');
set_param('signFollowingRobotYOLOv2ROS','GPUcuFFT','on');
```

Deploy ROS Node

To configure ROS node deployment on local host machine, in the **Connect** section of **ROS** tab, set **Deploy to** to `Localhost`. Select the **Build and Run** button from **ROS** tab to deploy the node. At the end of build process, you will see the ROS node running on local host machine.

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. It then shows how to generate a ROS 2 node for the control algorithm and deploy it to a remote device. The example shown here uses ROS 2 and MATLAB for simulation, and MATLAB Coder™ for code generation and deployment.

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 2-108.

Connect to a Robot Simulator

Start a ROS-based simulator for a differential-drive robot and configure the MATLAB® connection with the robot simulator.

To follow along with this example, download a virtual machine using instructions in "Get Started with Gazebo and Simulated TurtleBot" on page 1-206.

- Start the Ubuntu® virtual machine desktop.
- In the Ubuntu desktop, click the **Gazebo ROS2 Maze** icon to start the Gazebo world built for this example.
- 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/camera_info
/camera/image_raw
/clock
/cmd_vel
/imu
/joint_states
/odom
/parameter_events
/rosout
/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. Update the Quality of Service (QoS) policies of publisher and subscriber. 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/image_raw","sensor_msgs/Image","Reliability","besteffort","Du

odomSub = ros2subscriber(n, "/odom","nav_msgs/Odometry","Reliability","besteffort","Durability",'

[velPub, velMsg] = ros2publisher(n, "/cmd_vel", "geometry_msgs/Twist","Reliability","besteffort"
```

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');
```
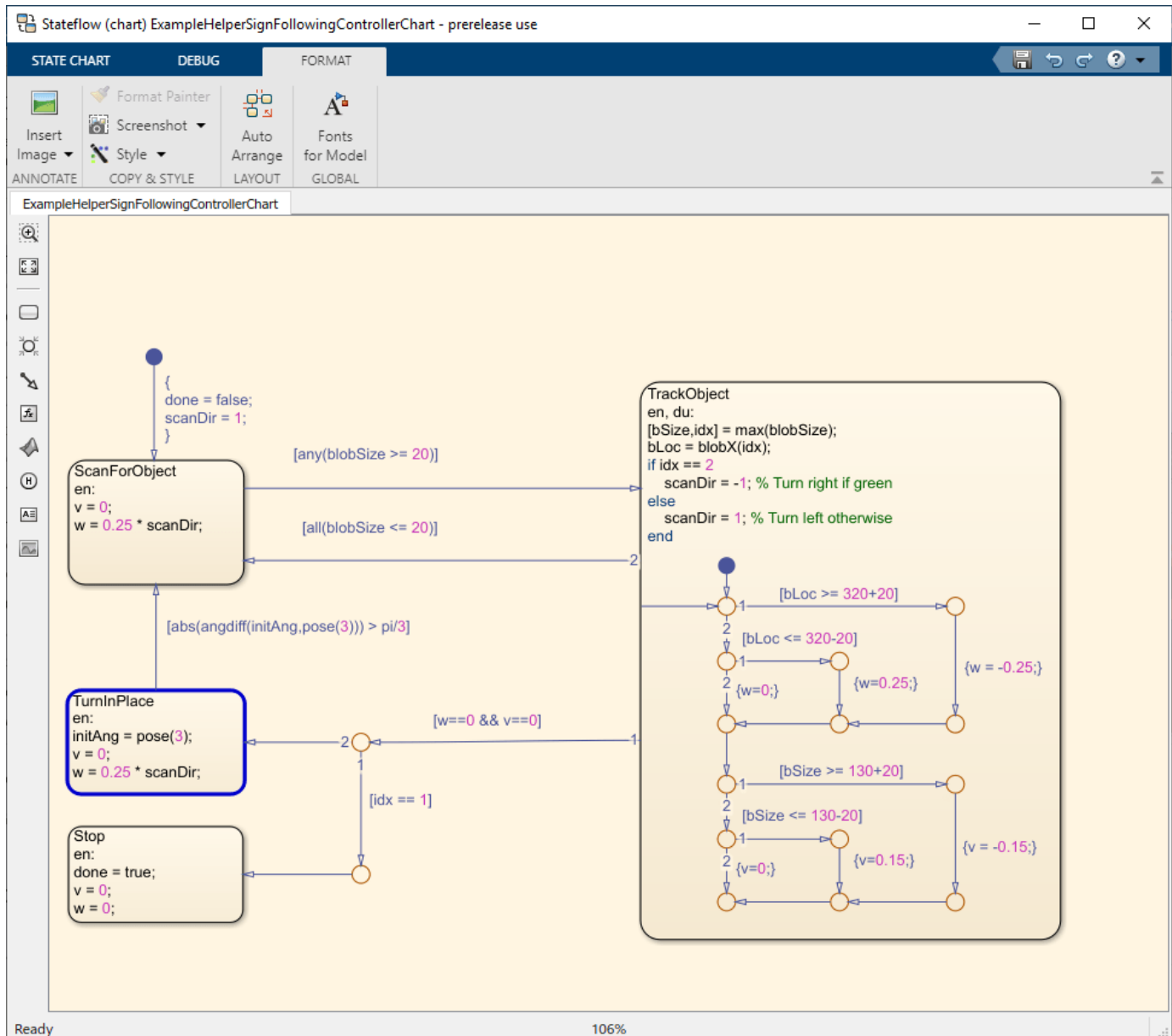
Run Control Loop

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] = 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.
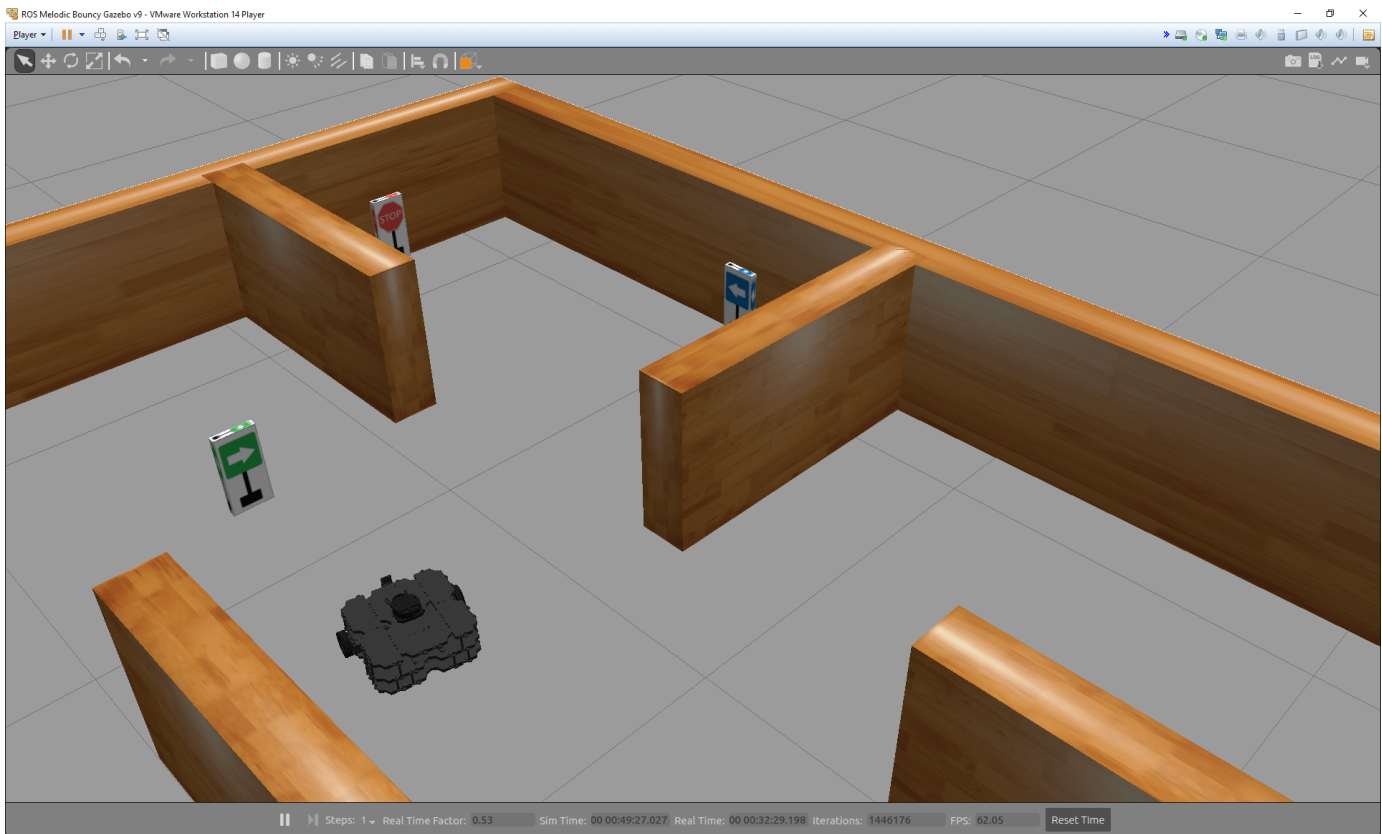
The robot follows the signs and stops at the final STOP sign. Reset the Gazebo scene after simulation using the /reset_simulation service. Create a ros2svcclient object for the service and use the call object function to call the service and reset the Gazebo simulation scene.

```
gazeboResetClient = ros2svcclient(n,'/reset_simulation','std_srvs/Empty');
call(gazeboResetClient);
```

Generate and Deploy ROS 2 Node

After the controller is verified, the next step is to generate a ROS 2 node for the sign following robot algorithm using MATLAB Coder™ and deploy it on the remote Virtual Machine running Gazebo. Deployment enables ROS nodes to run on the remote machines directly, resulting in faster executions. Create a MATLAB Coder configuration object that uses "Robot Operating System 2 (ROS 2)" hardware. For the Linux virtual machine for ROS Toolbox, set the following configuration parameters before remote deployment. Note that the actual values might be different for your remote device. Verify them before deployment. Set the build action to 'Build and Run' so that the deployed ROS node starts running after code generation.

```
cfg = coder.config("exe");
cfg.Hardware = coder.hardware("Robot Operating System 2 (ROS 2)");
cfg.Hardware.DeployTo = 'Remote Device';
cfg.Hardware.RemoteDeviceAddress = '192.168.192.129';
cfg.Hardware.RemoteDeviceUsername = 'user';
cfg.Hardware.RemoteDevicePassword = 'password';
cfg.Hardware.BuildAction = "Build and run";
```

Use `DeploySignFollowingRobotROS2` function that contains the controller algorithm code verified in the previous section. Run the following command to generate the ROS node and deploy the controller. You should see the robot moving in the Gazebo world.

```
codegen DeploySignFollowingRobotROS2 -config cfg

Connecting to ROS 2 device at '192.168.192.129'.
Using ROS 2 workspace '~/ros2_ws' to build ROS 2 node.
---
Transferring generated code for 'DeploySignFollowingRobotROS2' to ROS device.
Starting build for ROS node.
---
ROS 2 project directory: /home/user/ros2_ws/src
tar: Ignoring unknown extended header keyword 'SCHILY.fflags'
tar: Ignoring unknown extended header keyword 'SCHILY.fflags'
tar: Ignoring unknown extended header keyword 'SCHILY.fflags'
tar: Ignoring unknown extended header keyword 'SCHILY.fflags'


Starting >>> deploysignfollowingrobotros2

Finished <<< deploysignfollowingrobotros2 [20.5s]

Summary: 1 package finished [20.7s]


---
Running ROS 2 node.
---
Use the 'ros2device' object to stop or start the generated node.
Code generation successful.
```

The robot follows the signs and stops at the final STOP sign. Reset the Gazebo scene after the node executes by calling the `rossvcclient` object, `gazeboResetClient`.

```
call(gazeboResetClient);
```

Rerun Deployed Node Using `ros2device`

To rerun the deployed ROS node from MATLAB, create a `ros2device` object specifying the `deviceAddress, username,` and `password` values of the Virtual Machine running Gazebo. This establishes an SSH connection between the ROS device and MATLAB. Check the available nodes on the connected remote device. Verify that the deployed ROS node, `DeploySignFollowingRobotROS2`, exists.

```
gazeboVMDevice = ros2device('192.168.192.129','user','password');
gazeboVMDevice.AvailableNodes

ans = 1×3 cell
    {'DeploySignFollowingRobotROS2'}    {'gazebo_ros_paths.py'}    {'spawn_entity.py'}
```

Run the ROS node deployed on the remote device using `runNode` function.

```
runNode(gazeboVMDevice,'DeploySignFollowingRobotROS2')
```

```
The node 'DeploySignFollowingRobotROS2' is already running on the ROS device. Use the 'stopNode'
```

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 2-108.

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 using instructions in "Get Started with Gazebo and Simulated TurtleBot" on page 1-206.

- Start the Ubuntu® virtual machine desktop.
- In the Ubuntu desktop, click the **Gazebo ROS2 Maze** icon to start the Gazebo world built for this example.
- In MATLAB Command Window, set the ROS_DOMAIN_ID environment variable to 25 to match the robot simulator 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/camera_info
/camera/image_raw
/clock
/cmd_vel
/imu
/joint_states
/odom
/parameter_events
/rosout
/scan
/tf
```

Open Model and Configure Simulink

Setup the Simulink ROS preferences to communicate with the robot simulator.

Open the example model.

```
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.
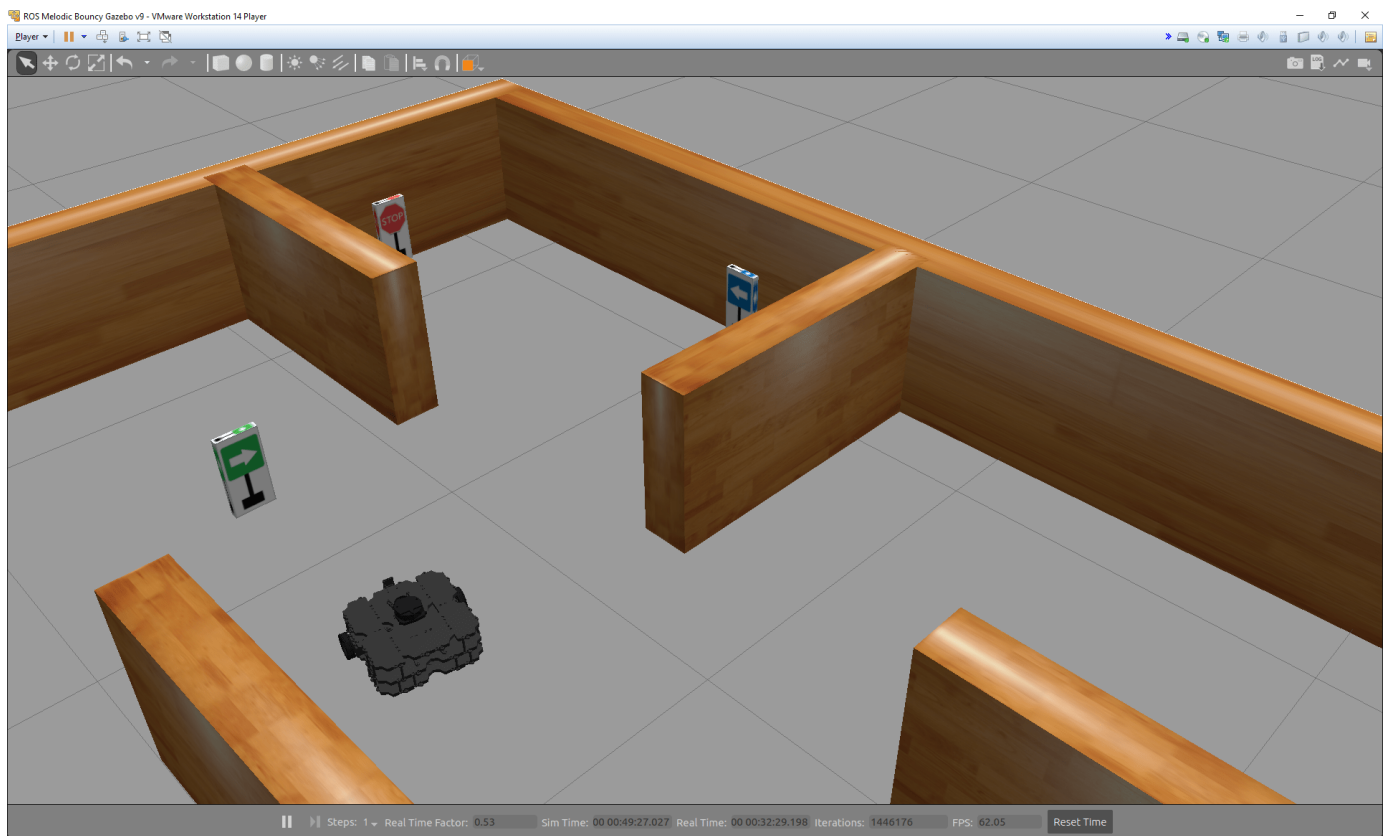
Automated Parking Valet with ROS in MATLAB

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
```
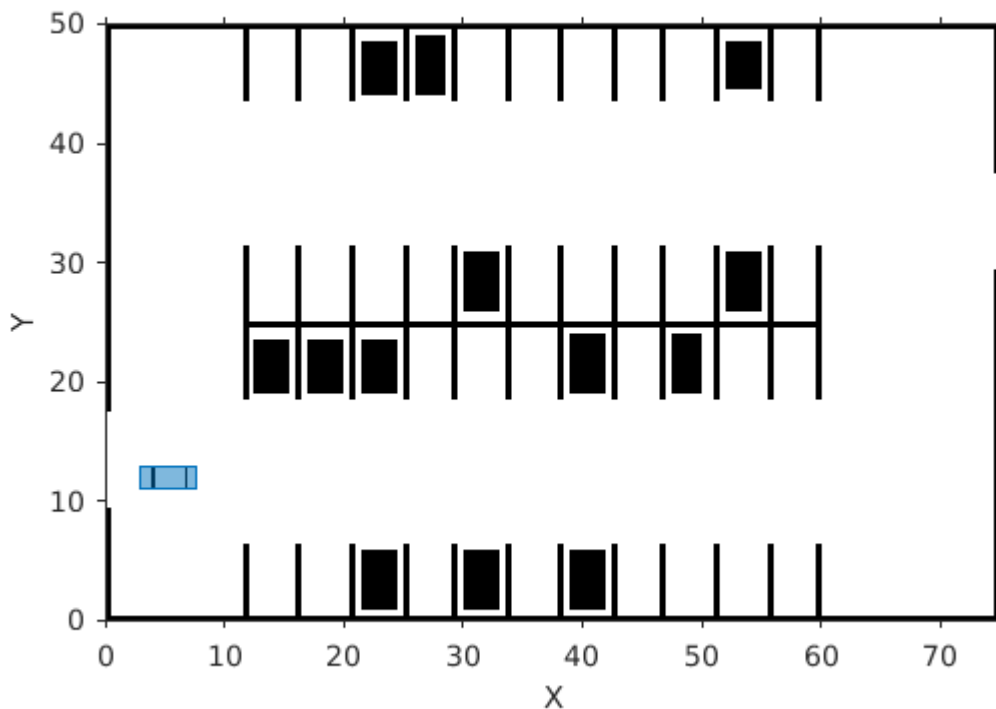


```
Initializing ROS master on http://192.168.0.10:60903.
Initializing global node /matlab_global_node_49911 with NodeURI http://sbd508773glnxa64:42335/
```

```
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.
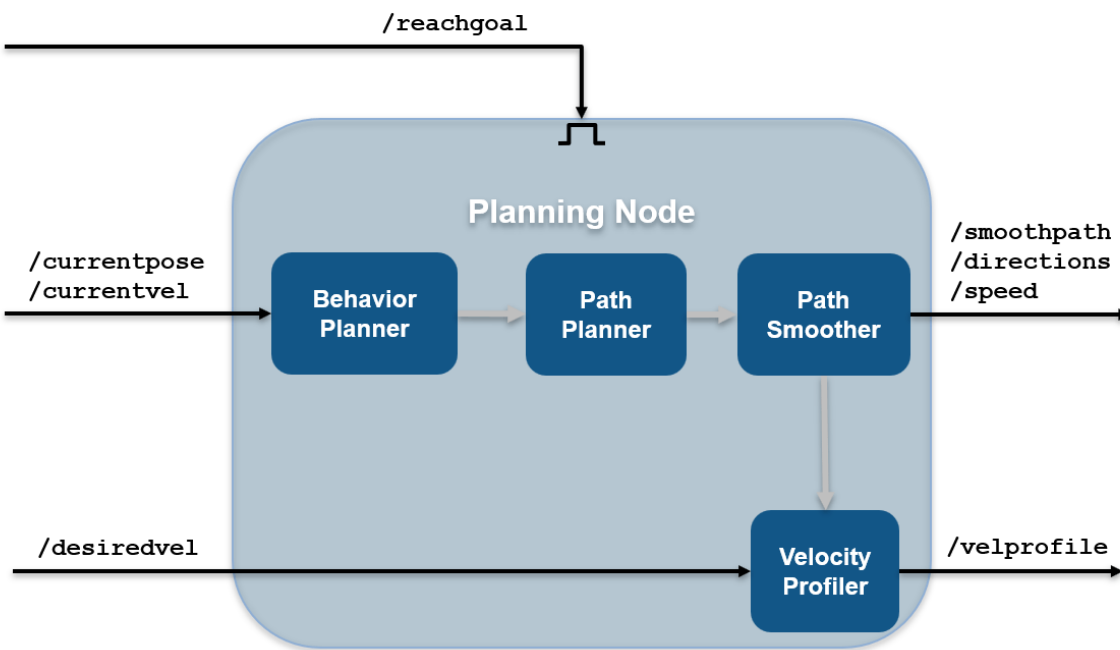
Create the planning node

```
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
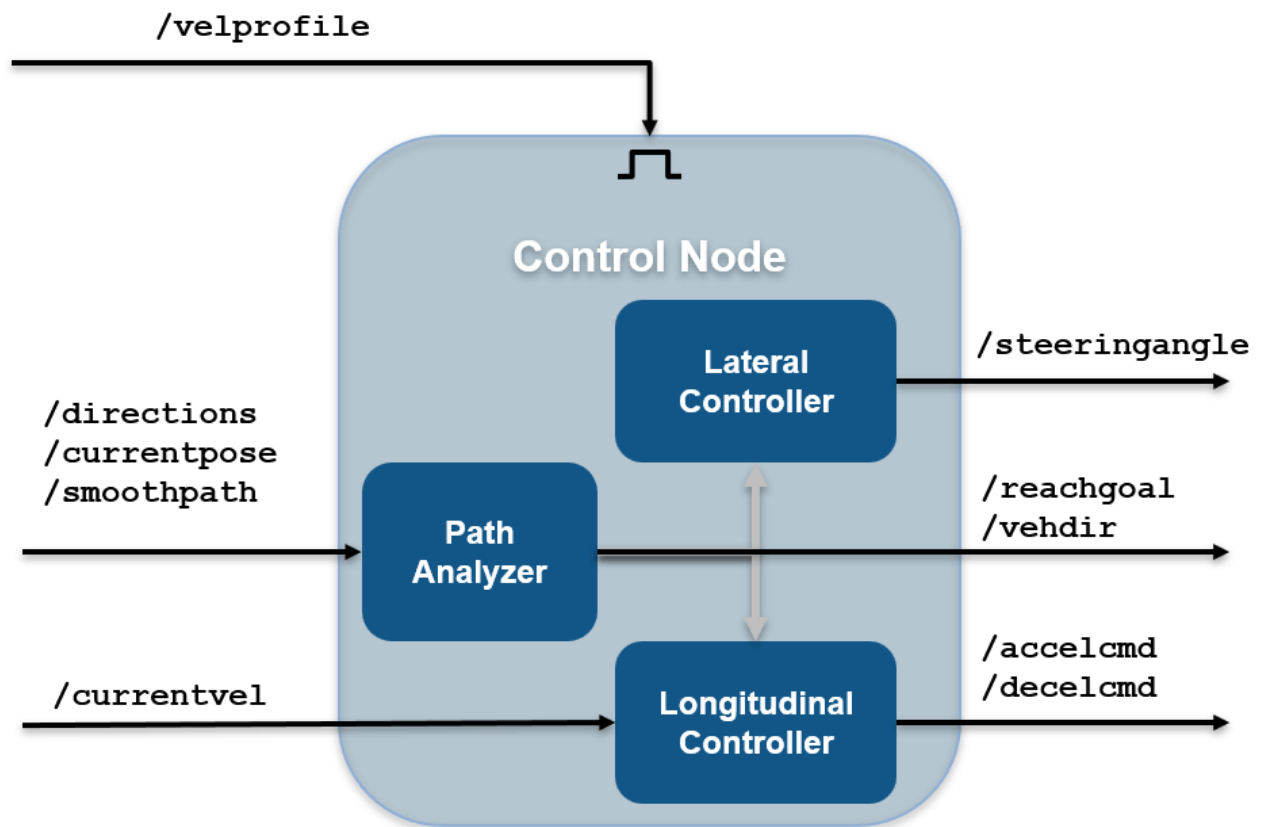
Create the controller, `controlNode`, and setup the publishers and subscribers in the node.

```
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`.
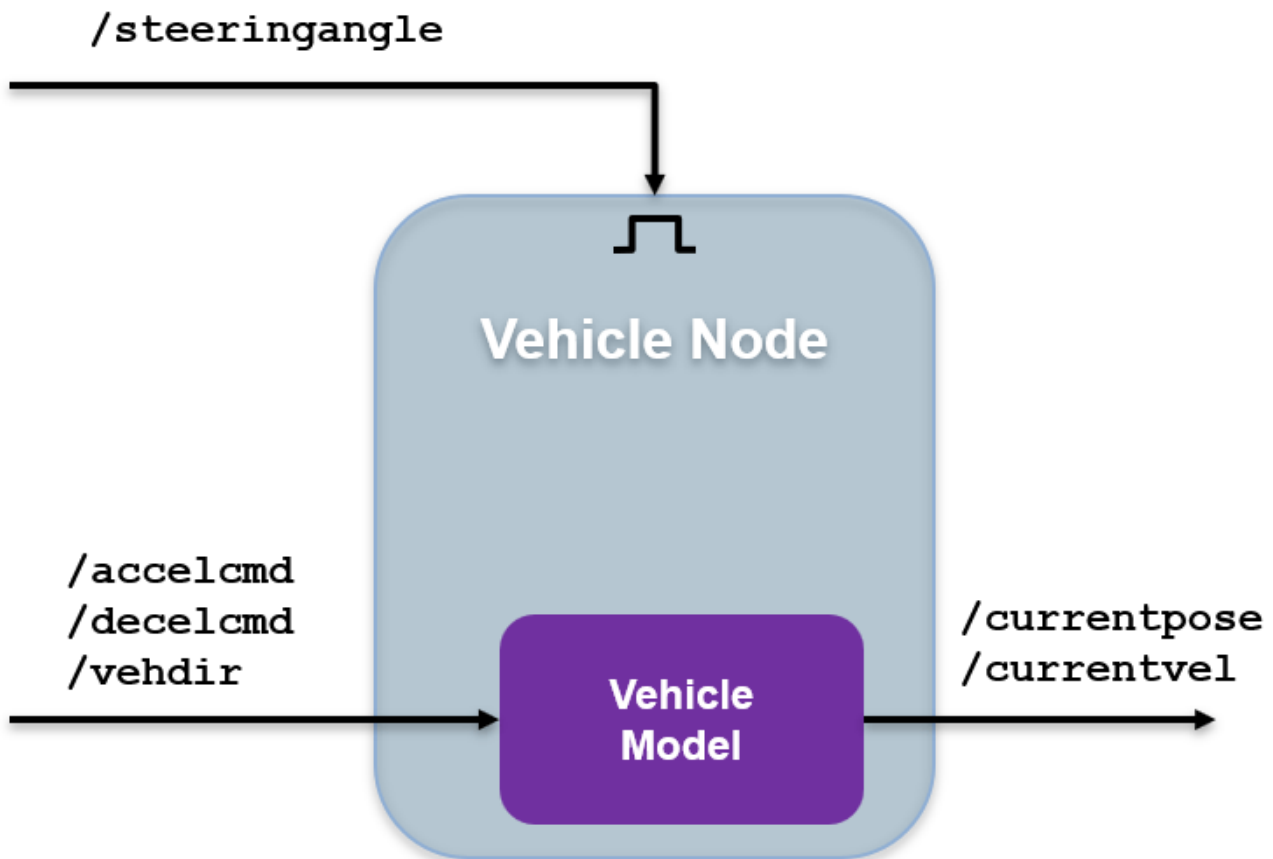
```
% Create vehicle node.
vehicleNode = ros.Node('vehicle', masterHost);

% 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);
```
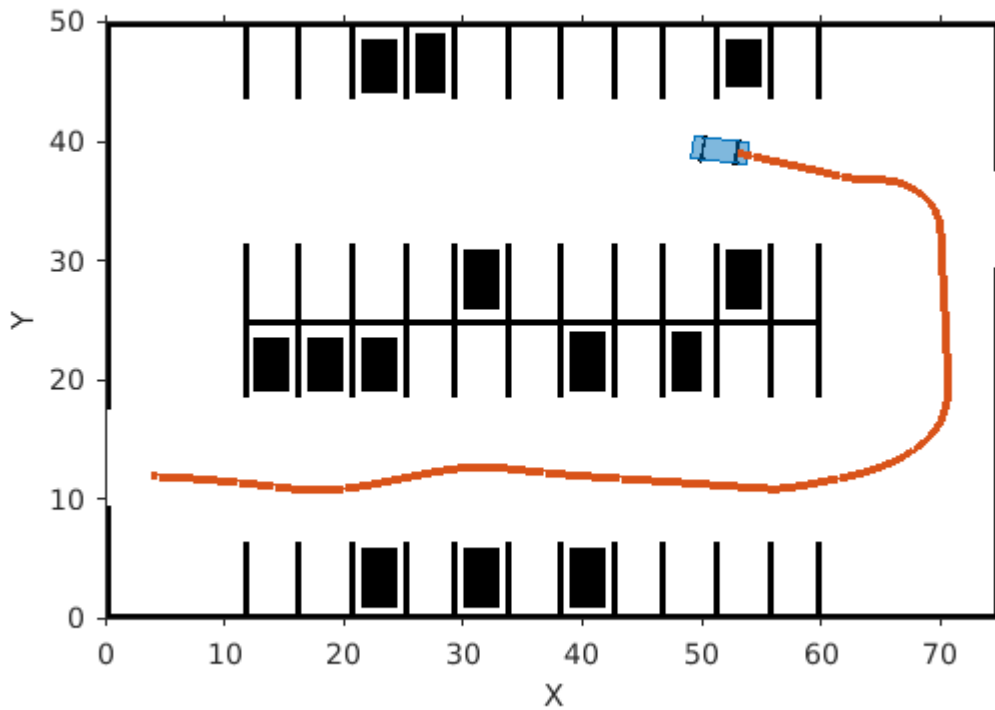
Park Maneuver

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 1-195

Introduction

A typical autonomous vehicle application has the following workflow.
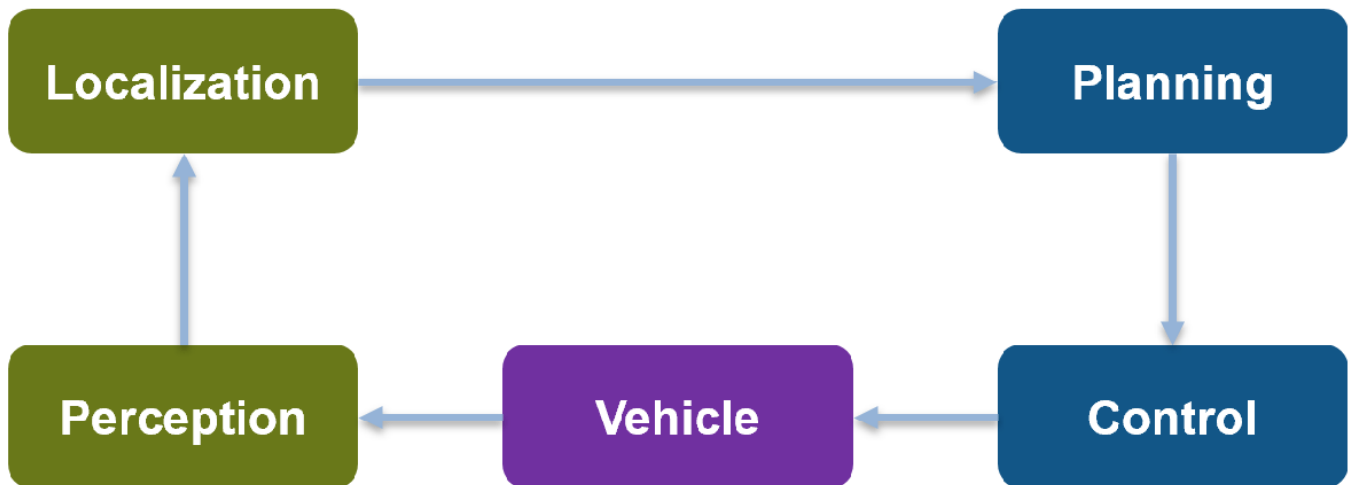
This example concentrates on simulating the *Planning*, *Control* and the *Vehicle* components. For *Localization*, this example uses pre-recorded map localization data. The *Planning* component are further divided into *Behavior planner* and *Path Planner* components. This results in a ROS network comprised 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.
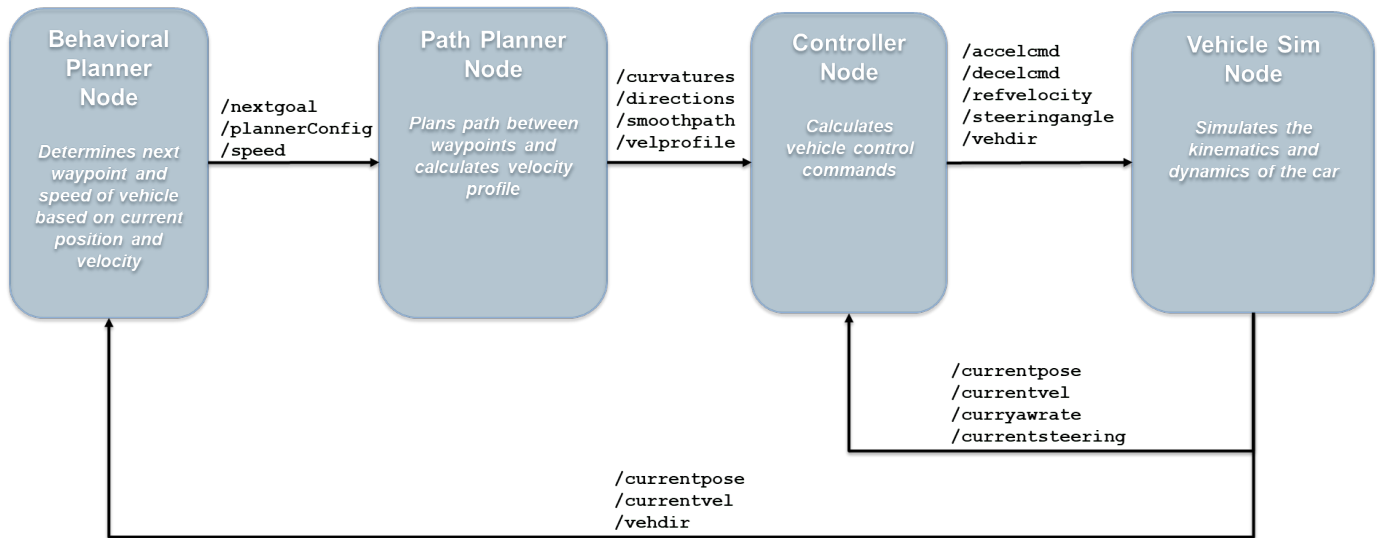
Explore the Simulink ROS nodes and connectivity

Observe the division of the components into four separate Simulink models. Each Simulink model represents a ROS node.

Vehicle Sim Node
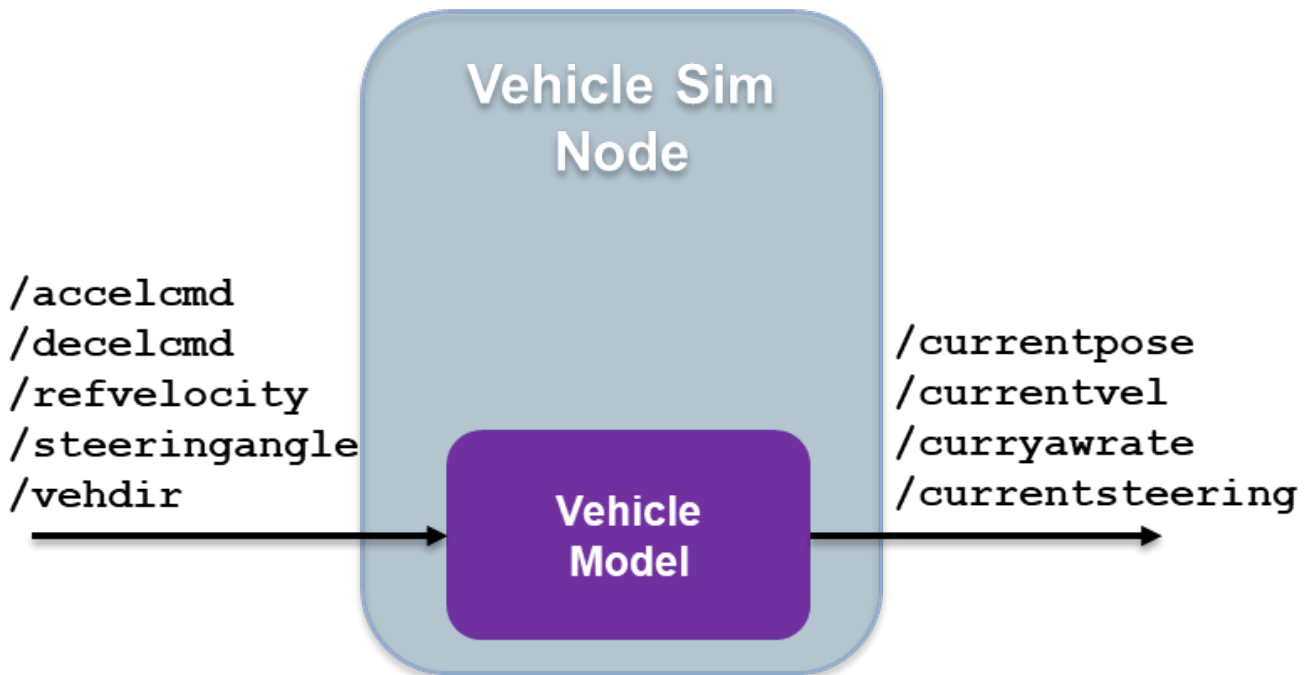
1. Open the vehicle model.

```
open_system('ROSValetVehicleExample');
```

2. 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.

Behavioral Planner Node
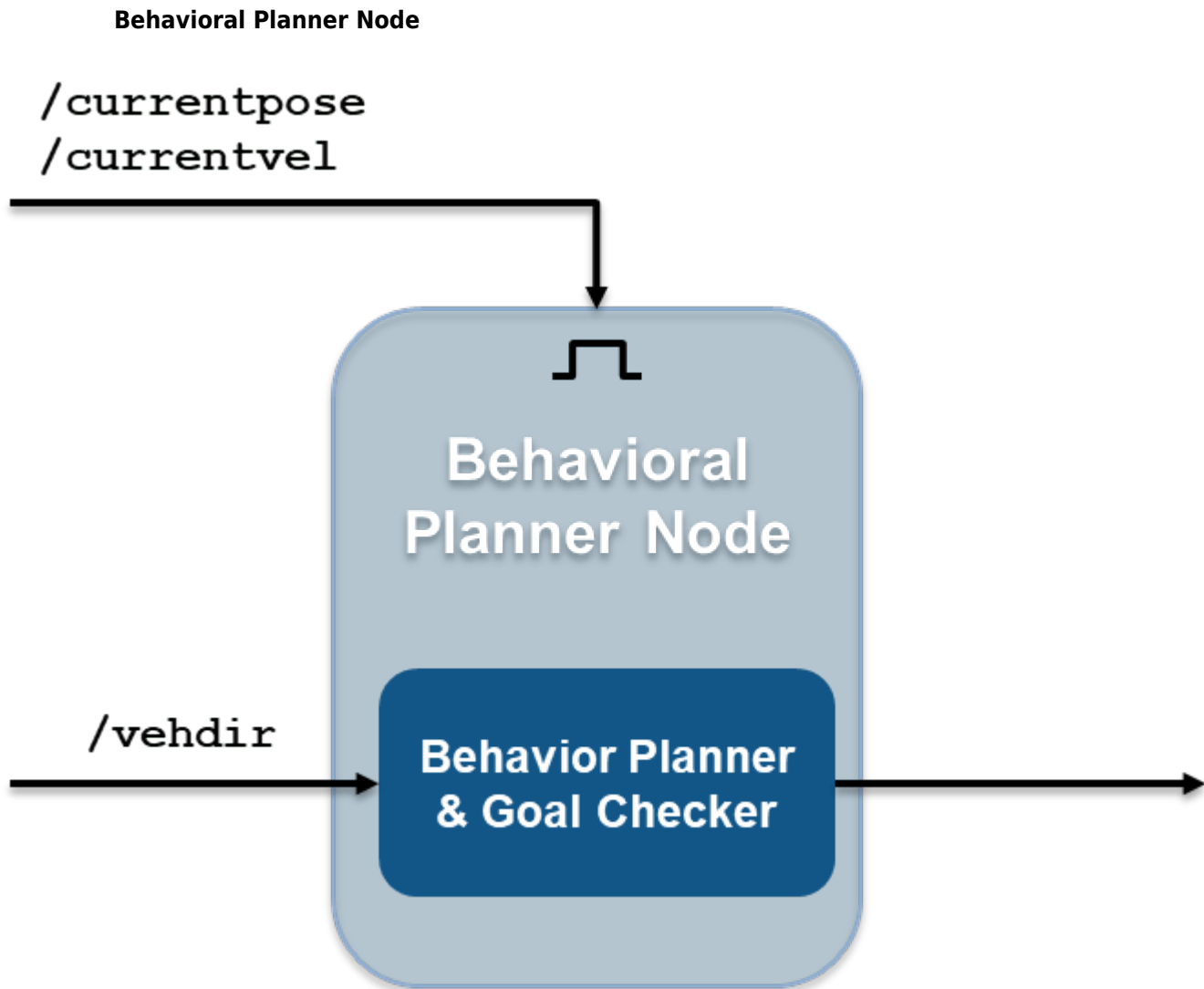
1. Open the behavioral planner model.

```
open_system('ROSValetBehavioralPlannerExample');
```

2. This model reads the current vehicle information, sends the next goal and 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`.
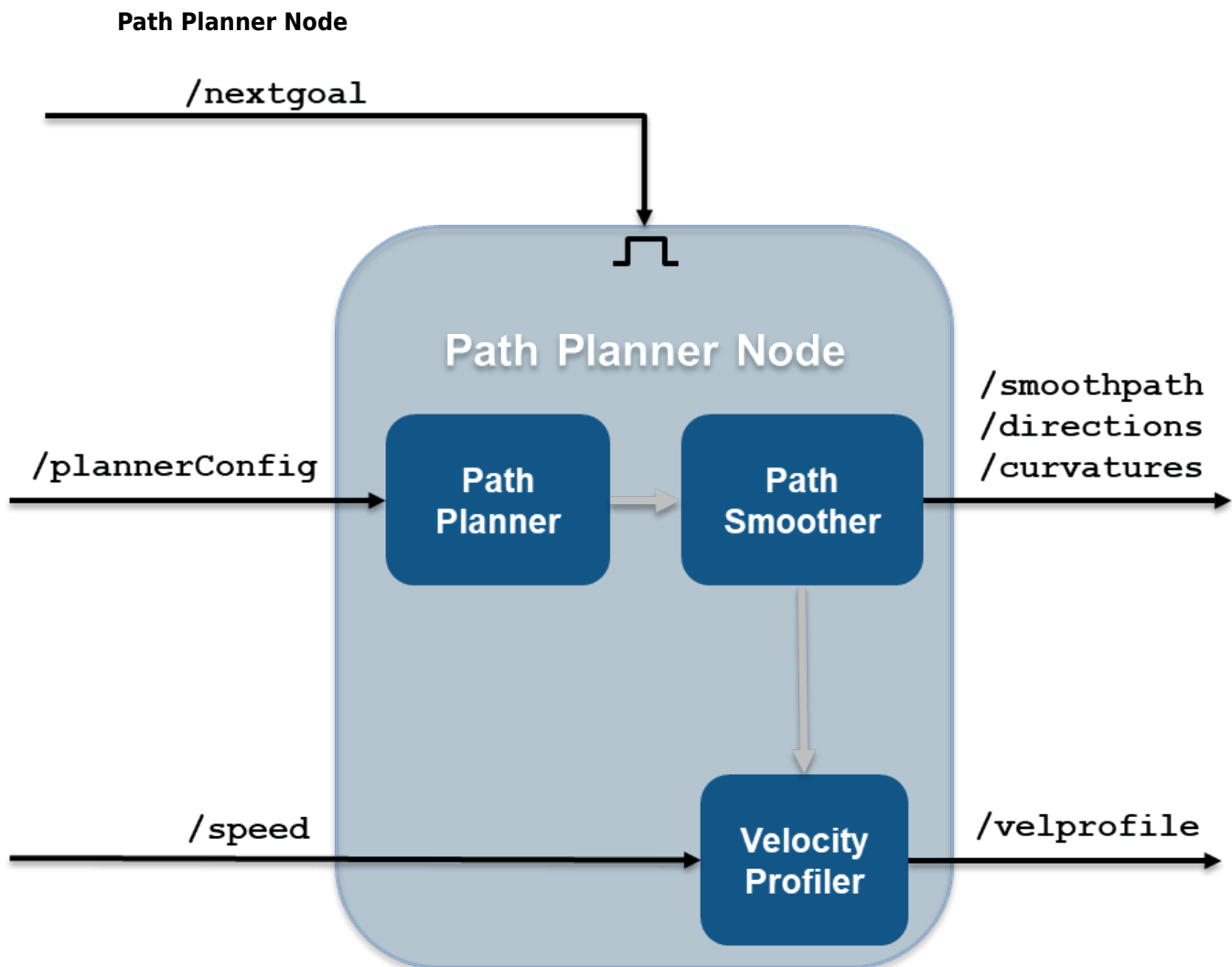
Path Planner Node

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.
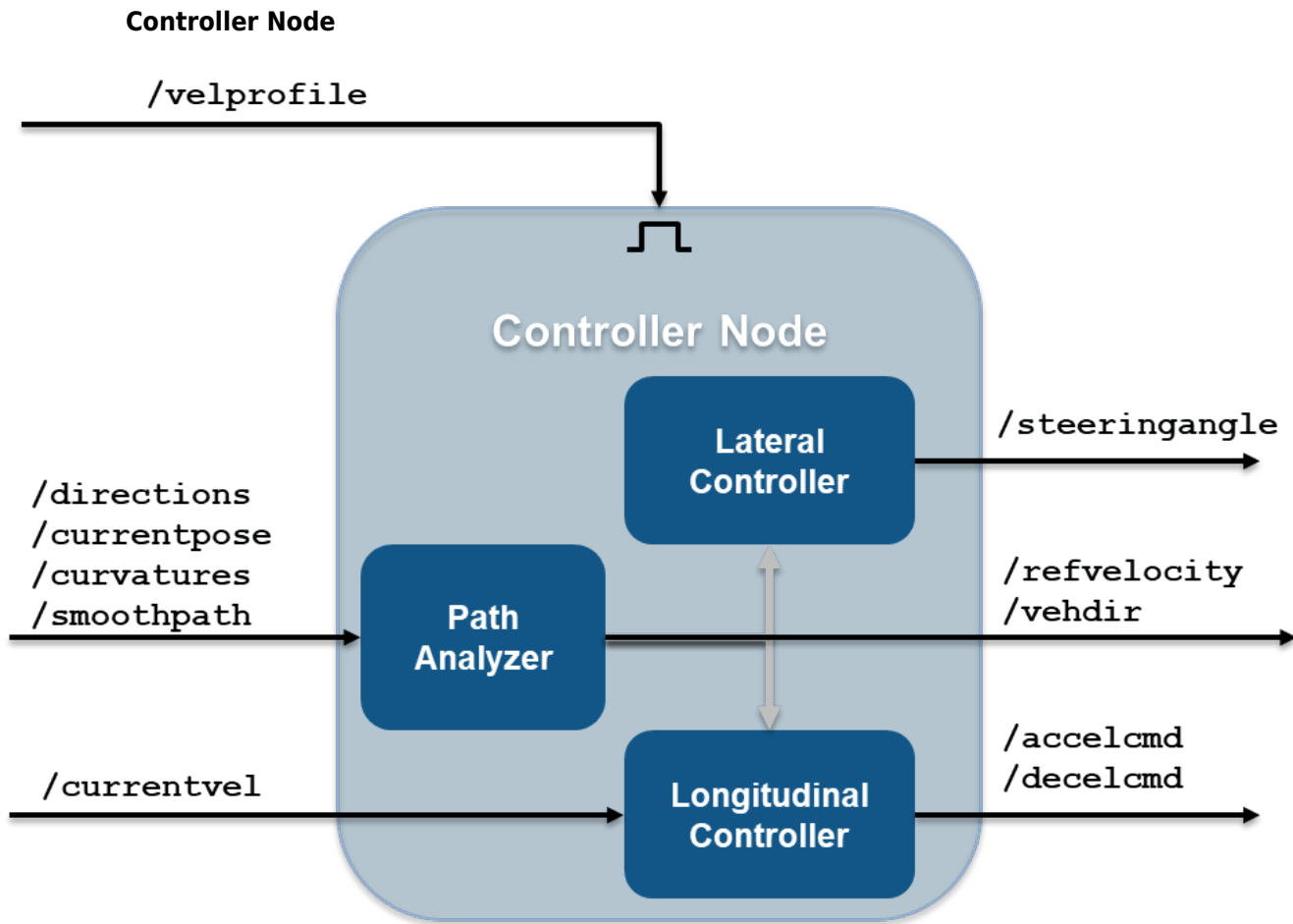
Controller Node

1. Open the vehicle controller model.

```
open_system('ROSValetControllerExample');
```

2. This model calculates and sends the steering and velocity commands.

3. The Controller subsystem runs when a new message is available on the `/velprofile` topic.

Simulate the ROS nodes to verify partitioning

Model references in this example use "Implement Variations in Separate Hierarchy Using Variant Subsystems" (Simulink) to achieve easy switch between simulation and code generation. During simulation, messages pass from one node to other directly without ROS blocks to showcase the data flow and execution order among these four ROS nodes. During code generation, variant subsystem under each node will use publish and subscribe blocks to replace input and output ports. In this way, after we lock down the control algorithm during simulation, each node can be deployed as a standalone ROS node without any changes to the model.

Verify that the behavior of the model remains the same after partitioning the system into four Simuilnk models.

1. Run `rosinit` in MATLAB® Command Window to initialize the global node and ROS master

```
rosinit
```

```
Launching ROS Core...
...Done in 3.2077 seconds.
Initializing ROS master on http://172.30.131.134:60715.
Initializing global node /matlab_global_node_37965 with NodeURI http://bat6234win64:57326/ and Ma
```

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');`

```
Warning: Inconsistent sample times. Sample time (0.05) of signal driving 'Input Port 4 (Left:4)'
Warning: Inconsistent sample times. Sample time (0.1) of signal driving 'Input Port 1 (Left:1)' 
Warning: Inconsistent sample times. Sample time (0.1) of signal driving 'Input Port 2 (Left:2)' 
Warning: Inconsistent sample times. Sample time (0.1) of signal driving 'Input Port 3 (Left:3)' 
Warning: Inconsistent sample times. Sample time (0.1) of signal driving 'Input Port 8 (Top:1)' o
Warning: Inconsistent sample times. Sample time (0.05) of signal driving 'Input Port 1 (Left:1)' 
Warning: Inconsistent sample times. Sample time (0.05) of signal driving 'Input Port 2 (Left:2)' 
Warning: Inconsistent sample times. Sample time (0.05) of signal driving 'Input Port 3 (Left:3)' 
Warning: Inconsistent sample times. Sample time (0.05) of signal driving 'Input Port 4 (Left:4)' 
Warning: Inconsistent sample times. Sample time (0.05) of signal driving 'Input Port 5 (Left:5)' 
Warning: Inconsistent sample times. Sample time (0.05) of signal driving 'Input Port 1' of 'ROSVa
Warning: Inconsistent sample times. Sample time (0.05) of signal driving 'Input Port 2' of 'ROSVa
Warning: Inconsistent sample times. Sample time (0.05) of signal driving 'Input Port 3' of 'ROSVa
```
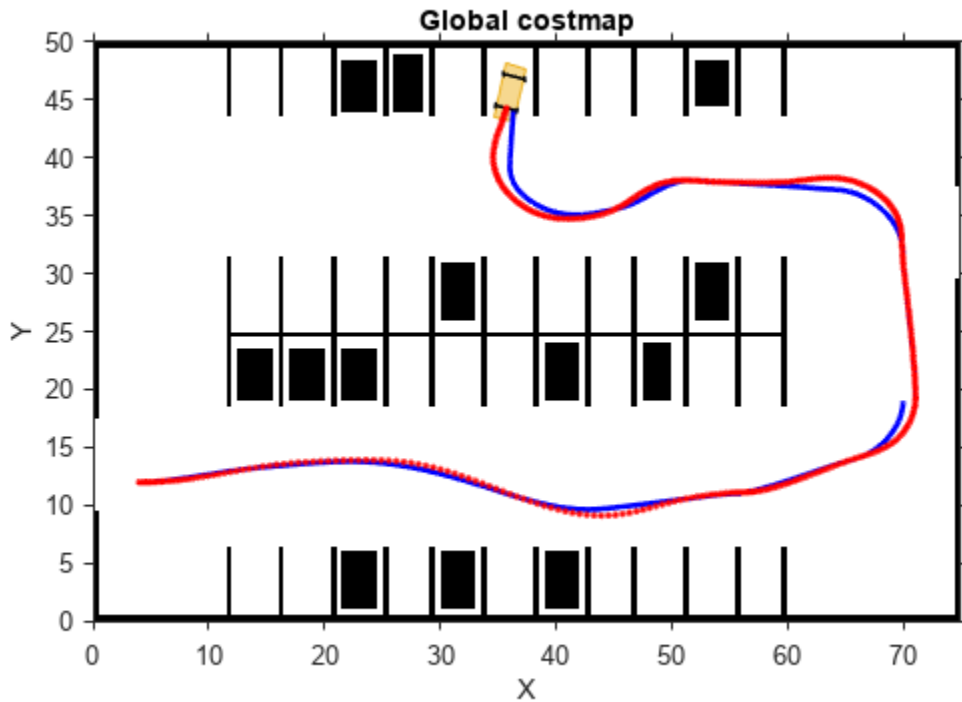
Simulation Results

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")
```
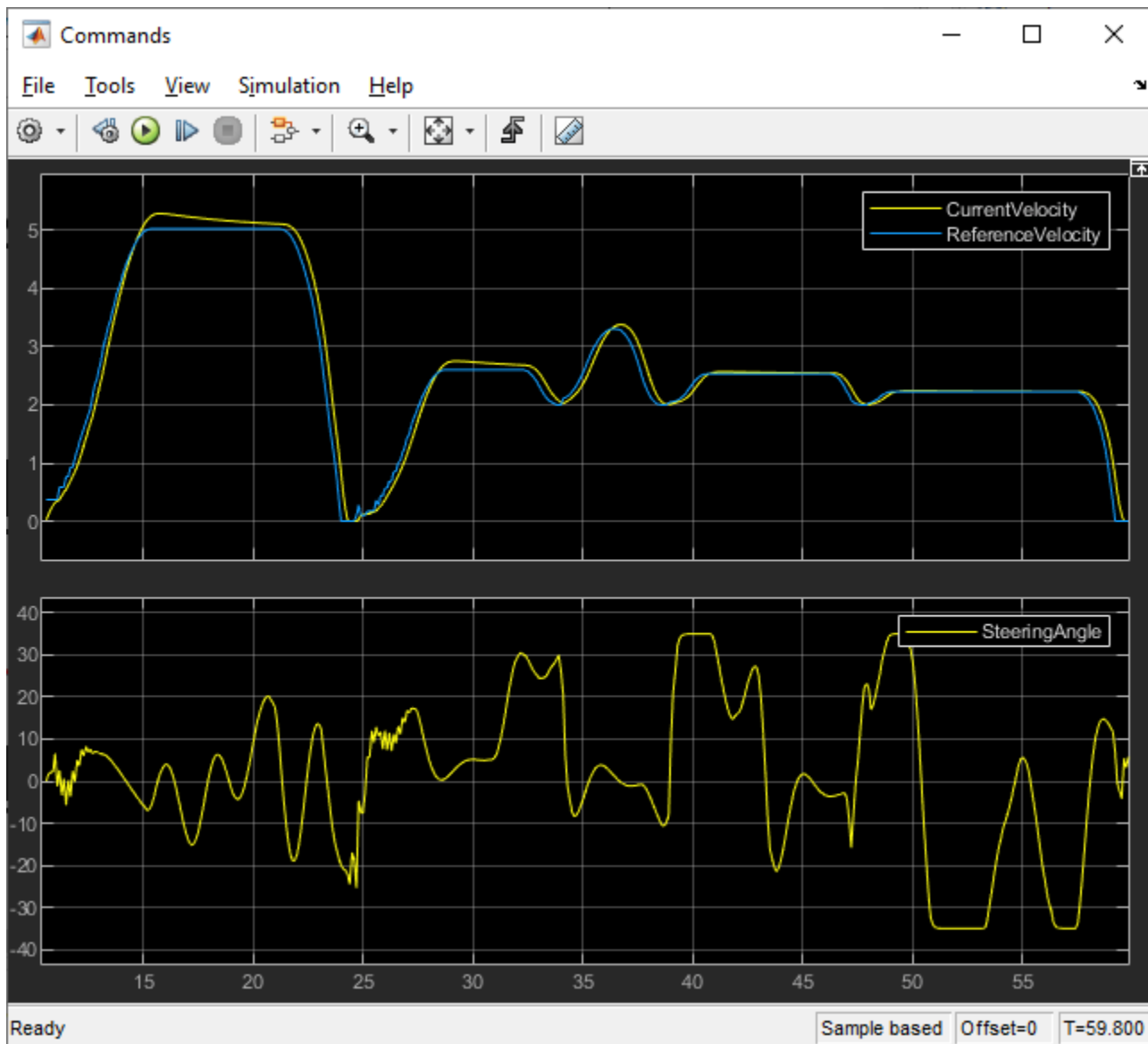
Deploy ROS Nodes

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 1-195.

1. Set `VEHICLE_MODE` to switch variant subsystem in Vehicle node for deployment.

`VEHICLE_MODE=1;`

2. Deploy the `Behavioral planner`, `Path planner` and `Controller` ROS nodes.

3. Open the vehicle model and set

`open_system('ROSValetVehicleExample');`

4. From the **Simulation** tab, click **Run** to start the simulation.

5. Observe the vehicle movement on the plot and compare the results from simulation run.

6. Shut down the ROS network using `rosshutdown`.

```
rosshutdown
```

Shutting down global node /matlab_global_node_37965 with NodeURI http://bat6234win64:57326/ and I
Shutting down ROS master on http://172.30.131.134:60715.

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.

/currentpose, /currentvel

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]
```

2-151

```
       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.
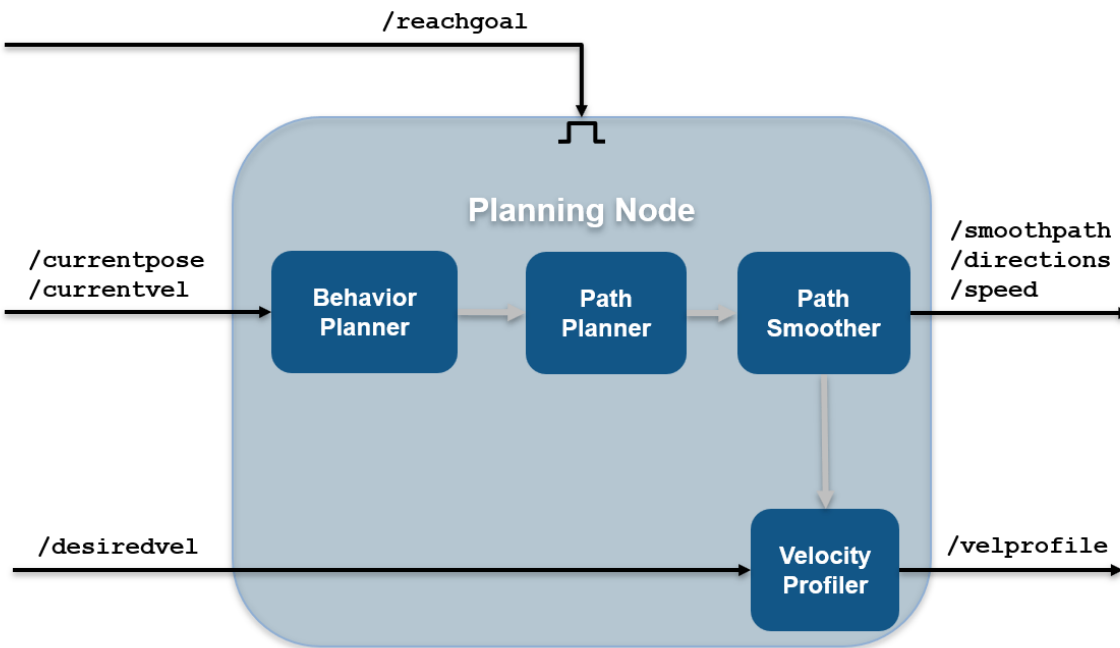

```matlab
% Create planning node.
planningNode = ros2node('planning');

% 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
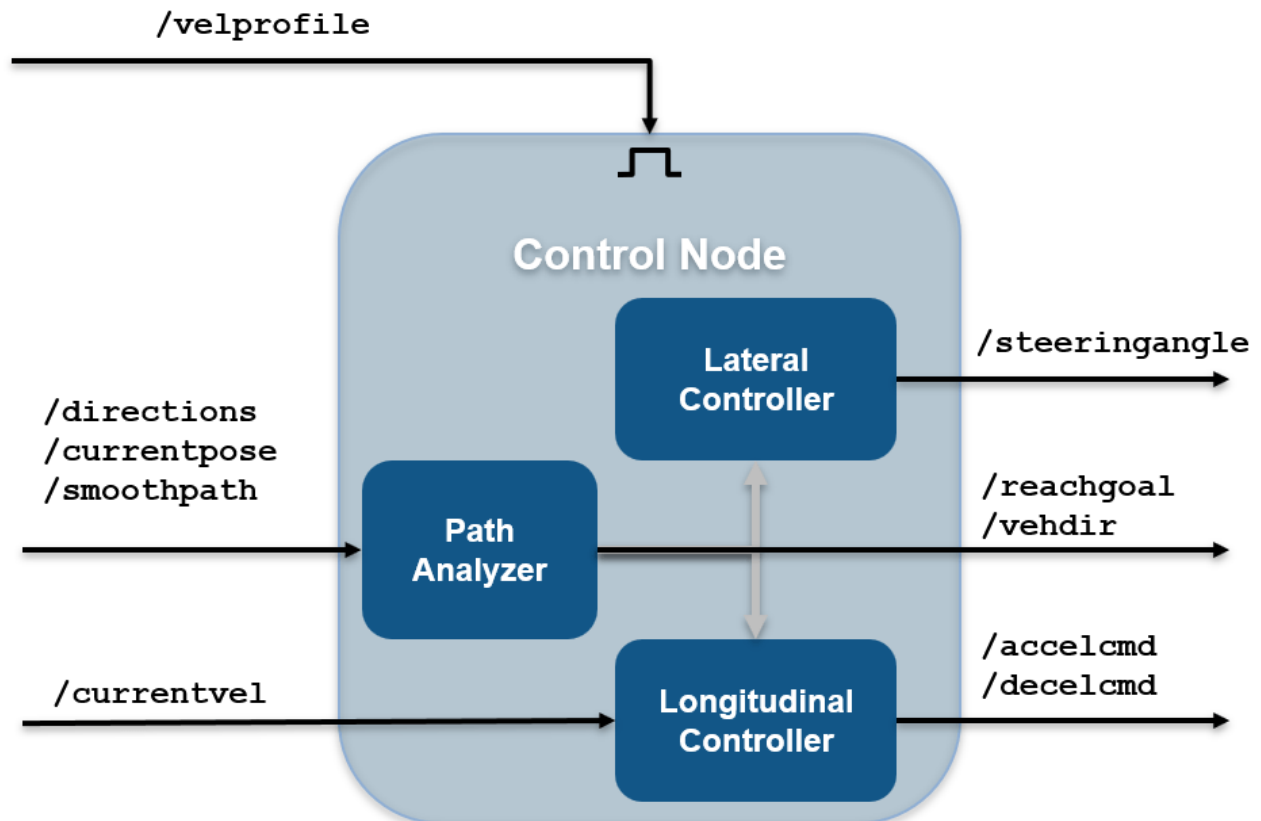

```matlab
% Create control node
controlNode = ros2node('control');

% 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);
```
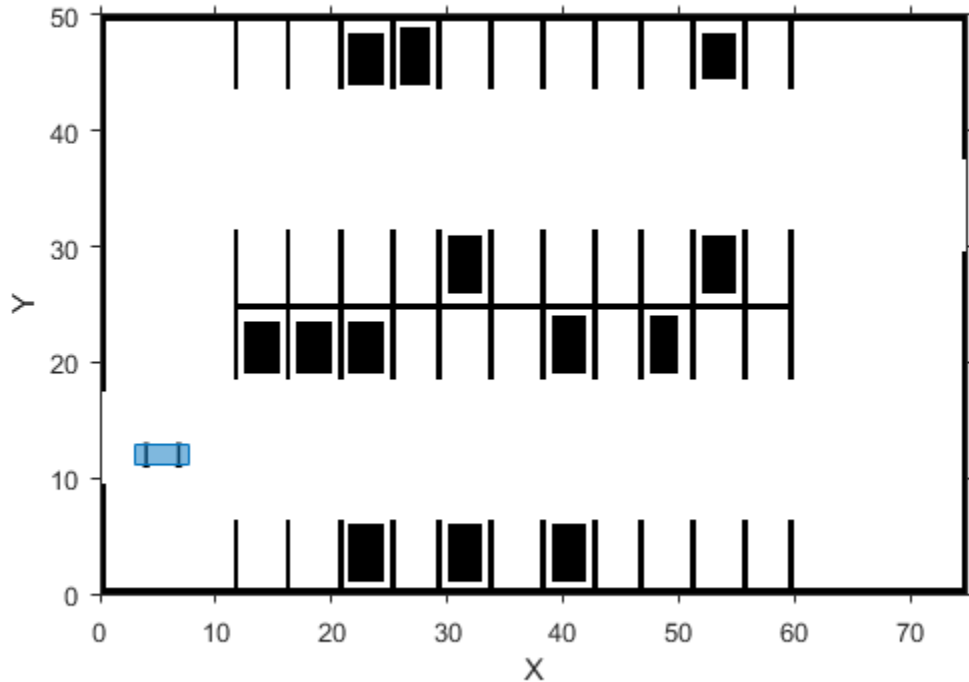

Vehicle

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));
```
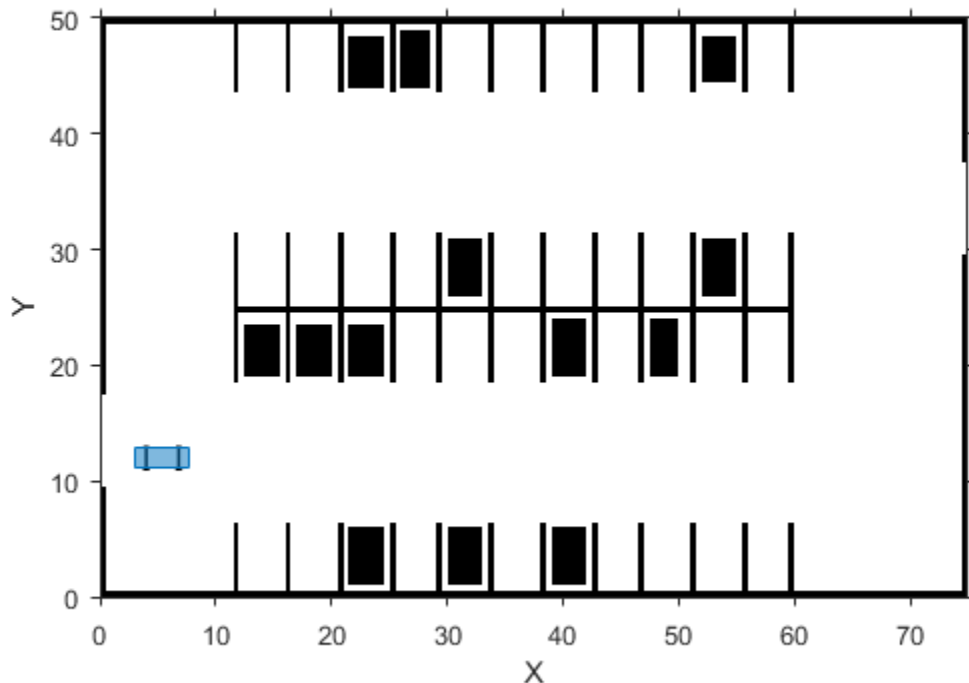

Initialize Simulation

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);
```
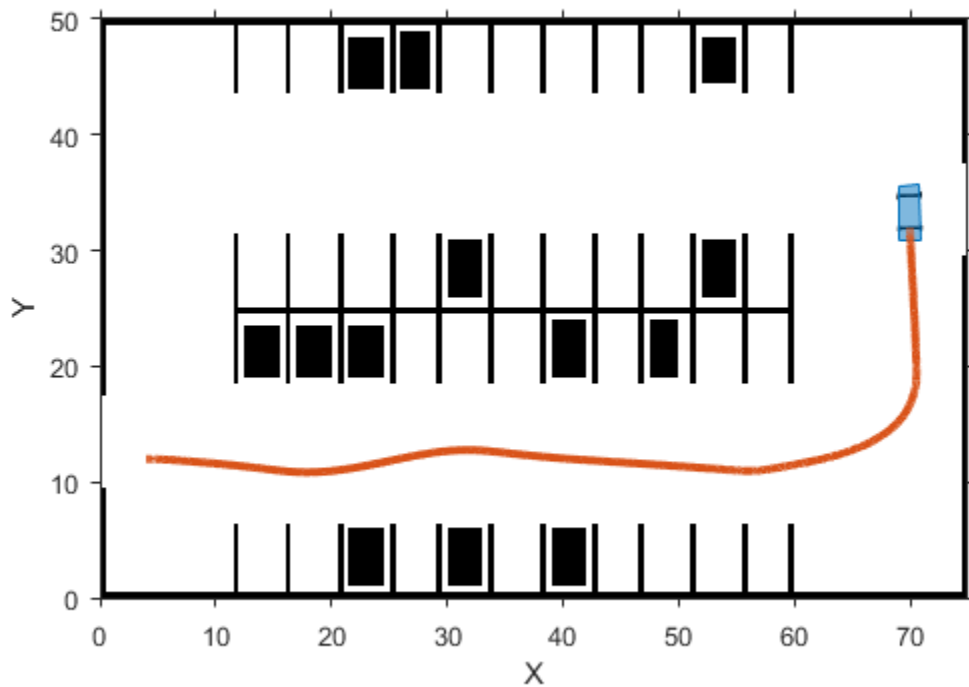

Main Loop

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);
```
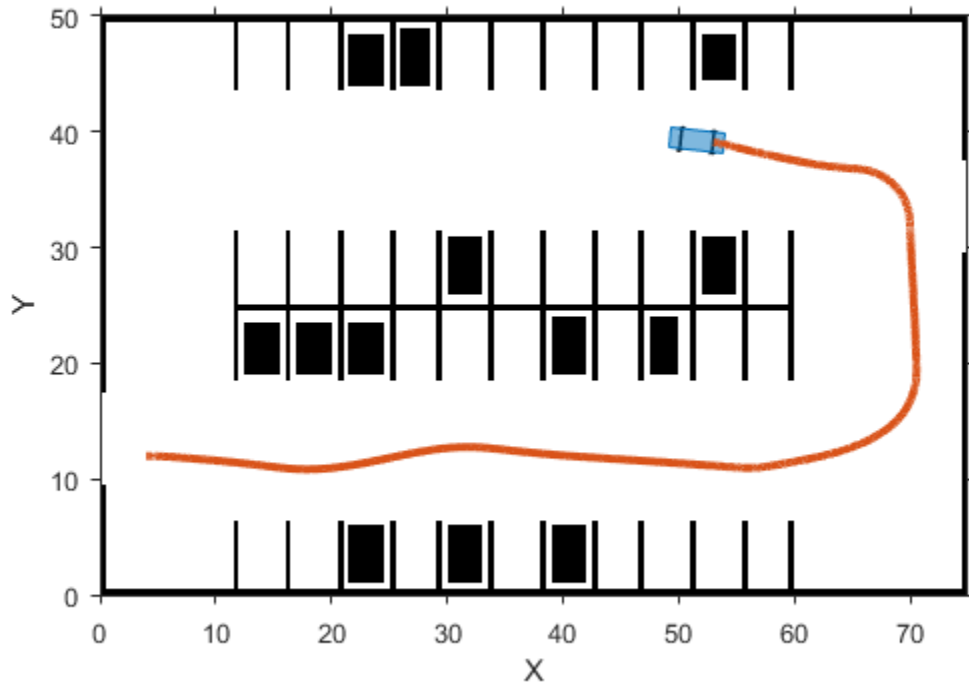
Park Maneuver

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');
```

Simulate Automated Parking Valet with ROS 2 in Simulink

This example shows how to distribute components of the Automated Parking Valet application into various Simulink® models and simulate them in a known parking lot environment.

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
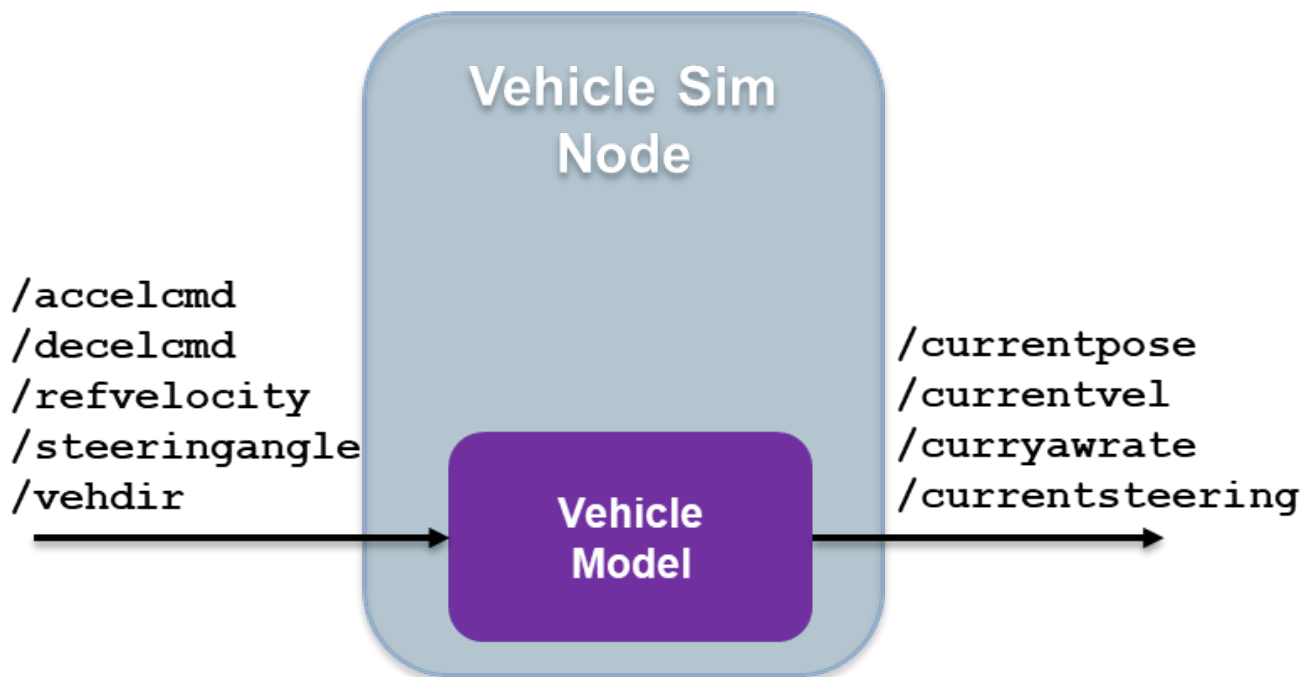
1. Open the vehicle model.

```
open_system('ROS2ValetVehicleExample');
```

2. The 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.

Behavioral Planner Node
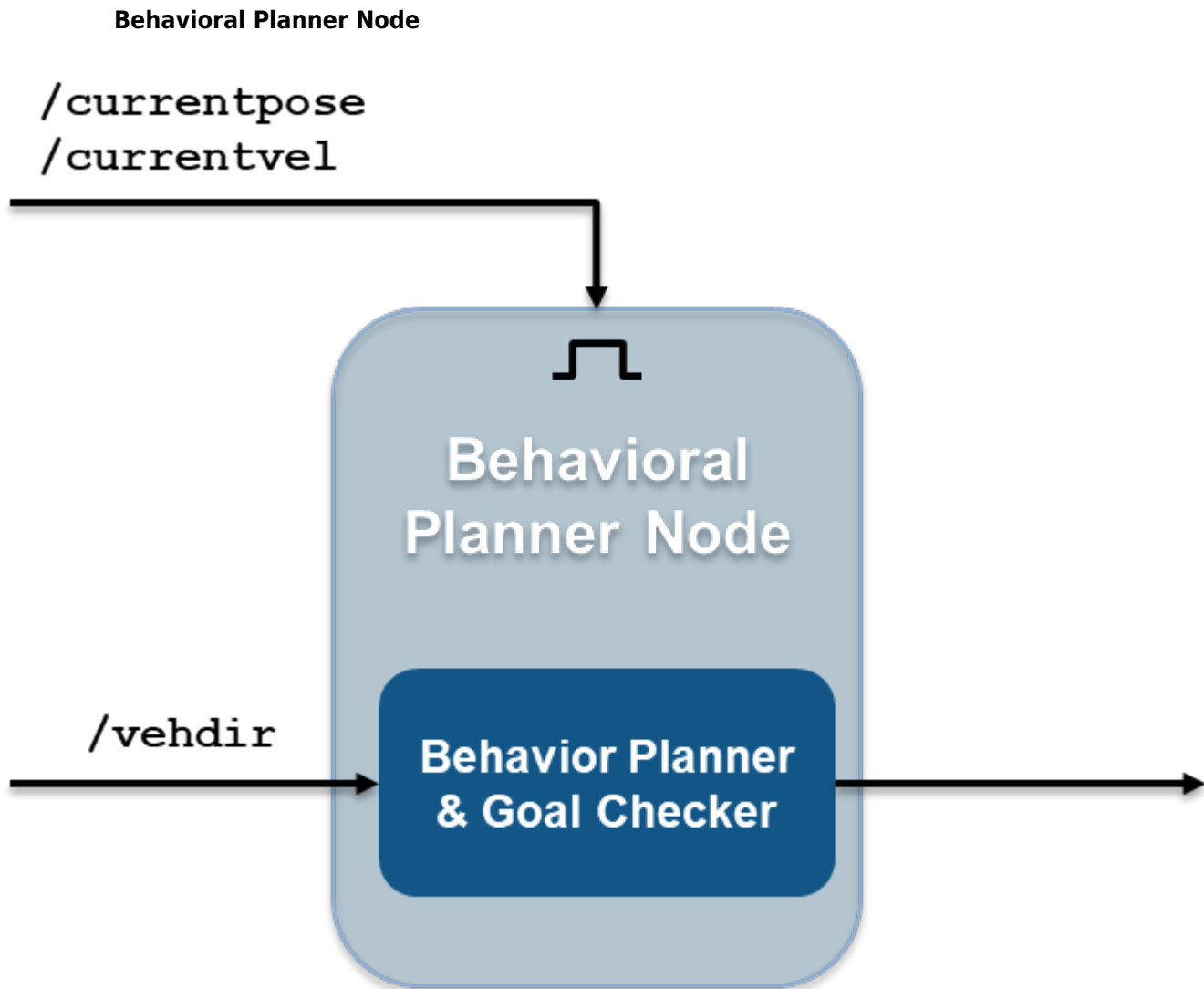
1. Open the behavioral planner model.

```
open_system('ROS2ValetBehavioralPlannerExample');
```

2. This model reads the current vehicle information, 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`.
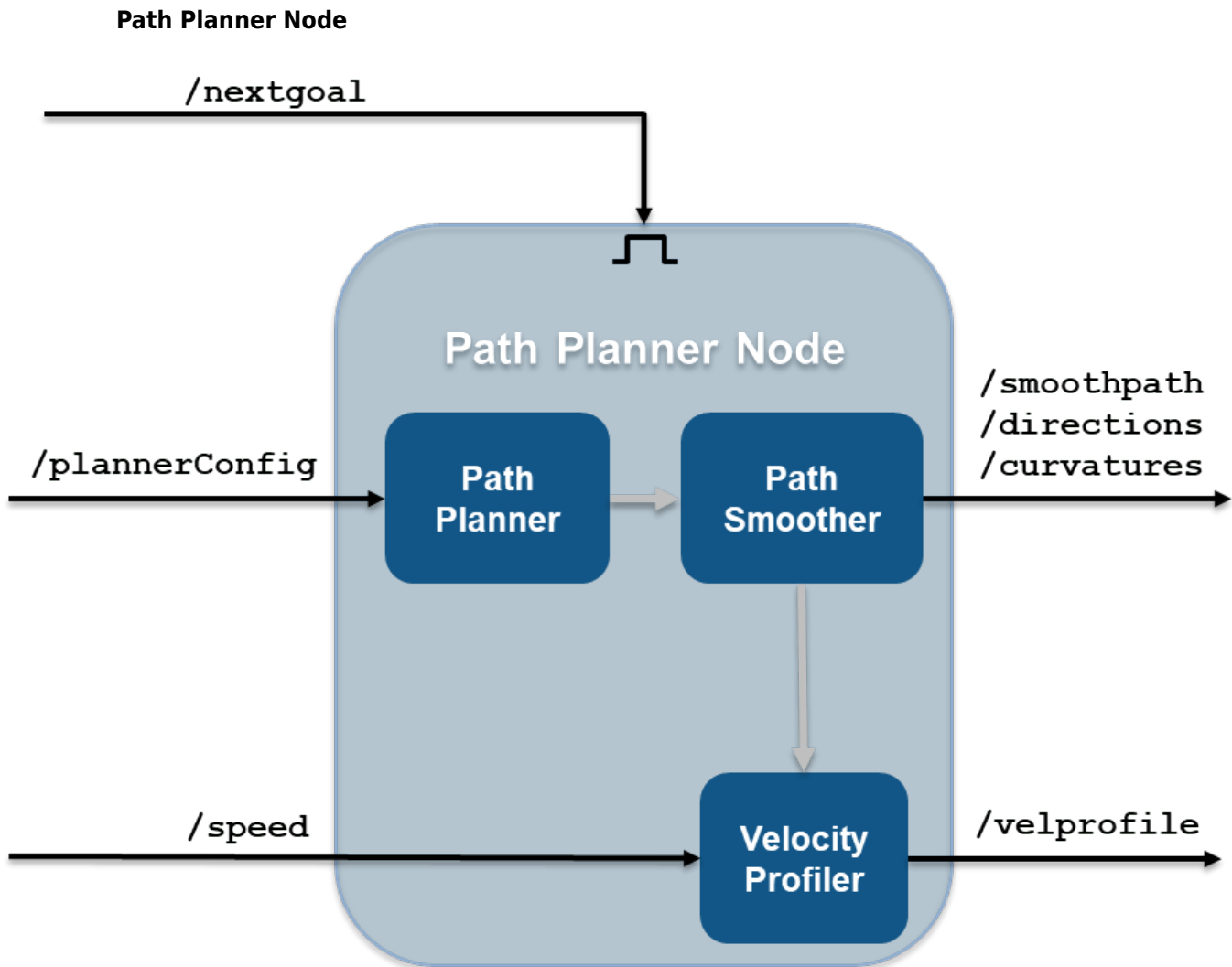
Path Planner Node

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.

3. The `Path Planner` subsystem runs when a new message is available on `/plannerConfig` or `/nextgoal` topics.
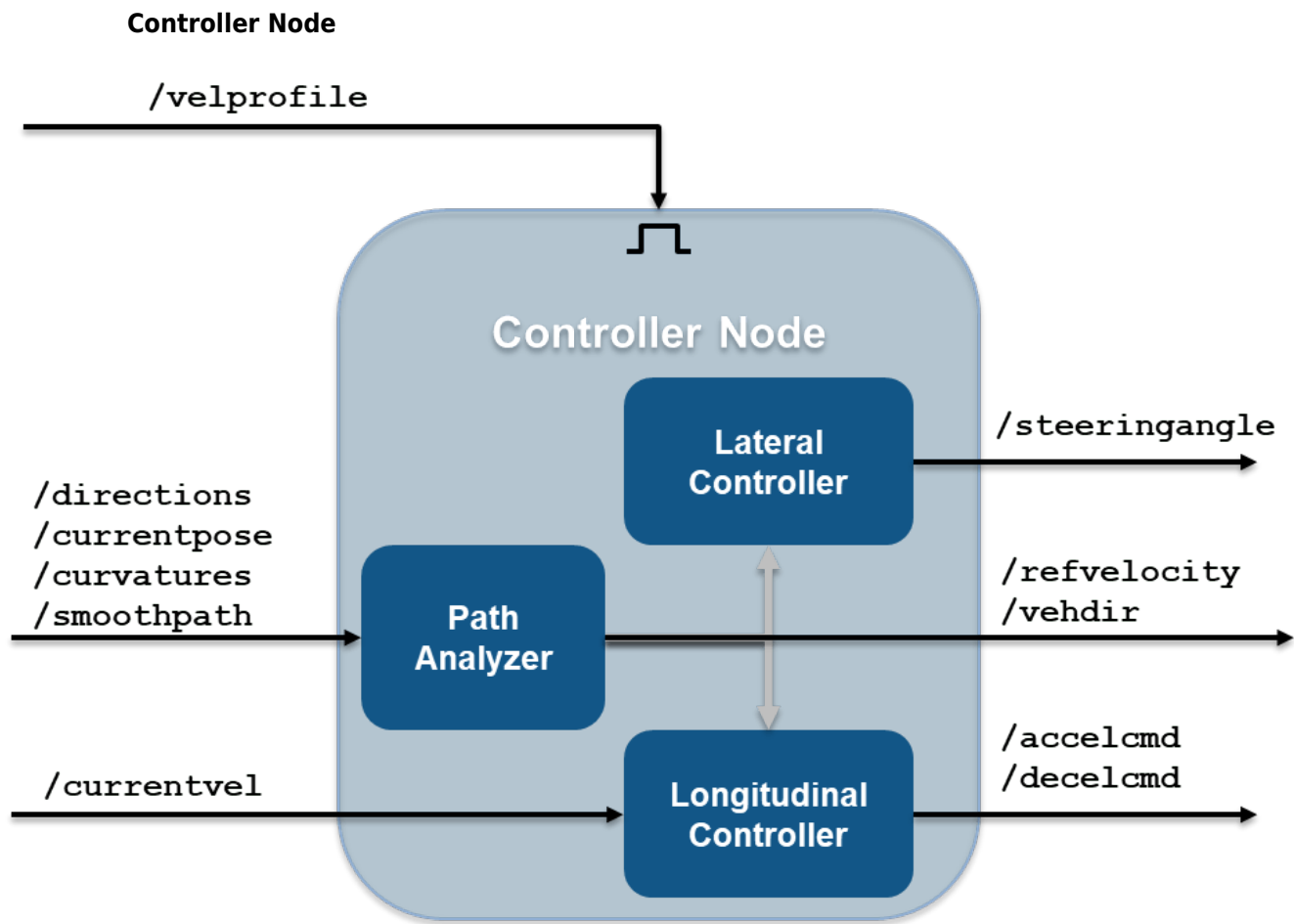
Controller Node

1. Open the vehicle controller model.

```
open_system('ROS2ValetControllerExample');
```

2. This model calculates and sends the steering and velocity commands.

3. The Controller subsystem runs when a new message is available on the /velprofile topic.

Simulate the ROS 2 nodes to verify partitioning

Model references in this example use "Implement Variations in Separate Hierarchy Using Variant Subsystems" (Simulink) to achieve easy switch between simulation and code generation. During simulation, messages pass from one node to other directly without ROS 2 blocks to showcase the data flow and execution order among these four ROS 2 nodes. During code generation, variant subsystem under each node will use publish and subscribe blocks to replace input and output ports. In this way, after we lock down the control algorithm during simulation, each node can be deployed as a standalone ROS 2 node without any changes to the model.

Verify that the behavior of the model remains the same after partitioning the system into four Simulink models.

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');
```
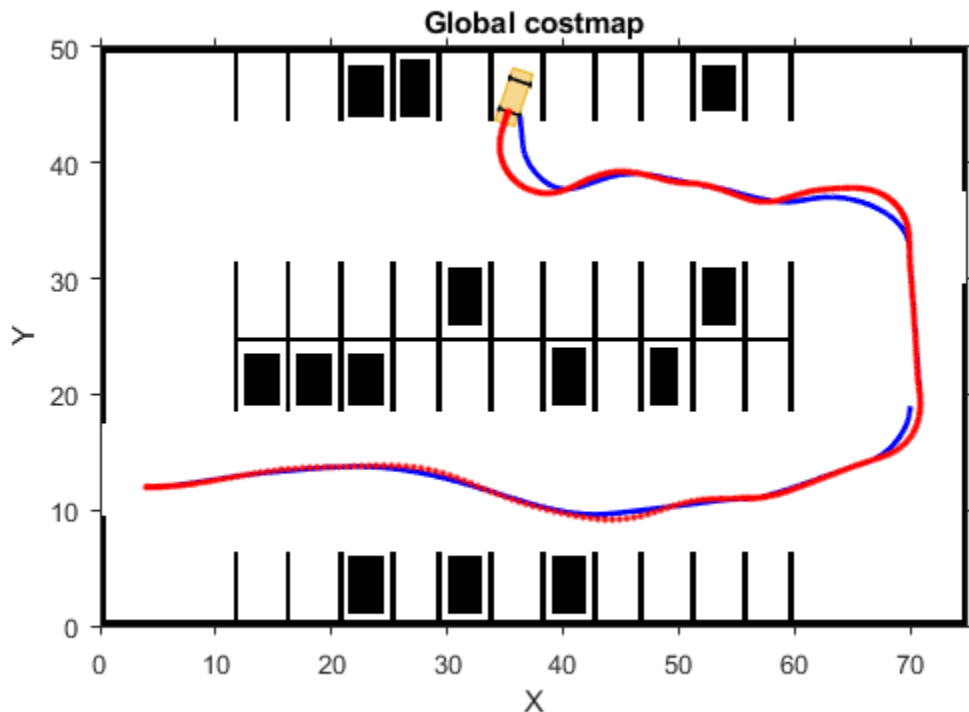

Simulation Results

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")
```
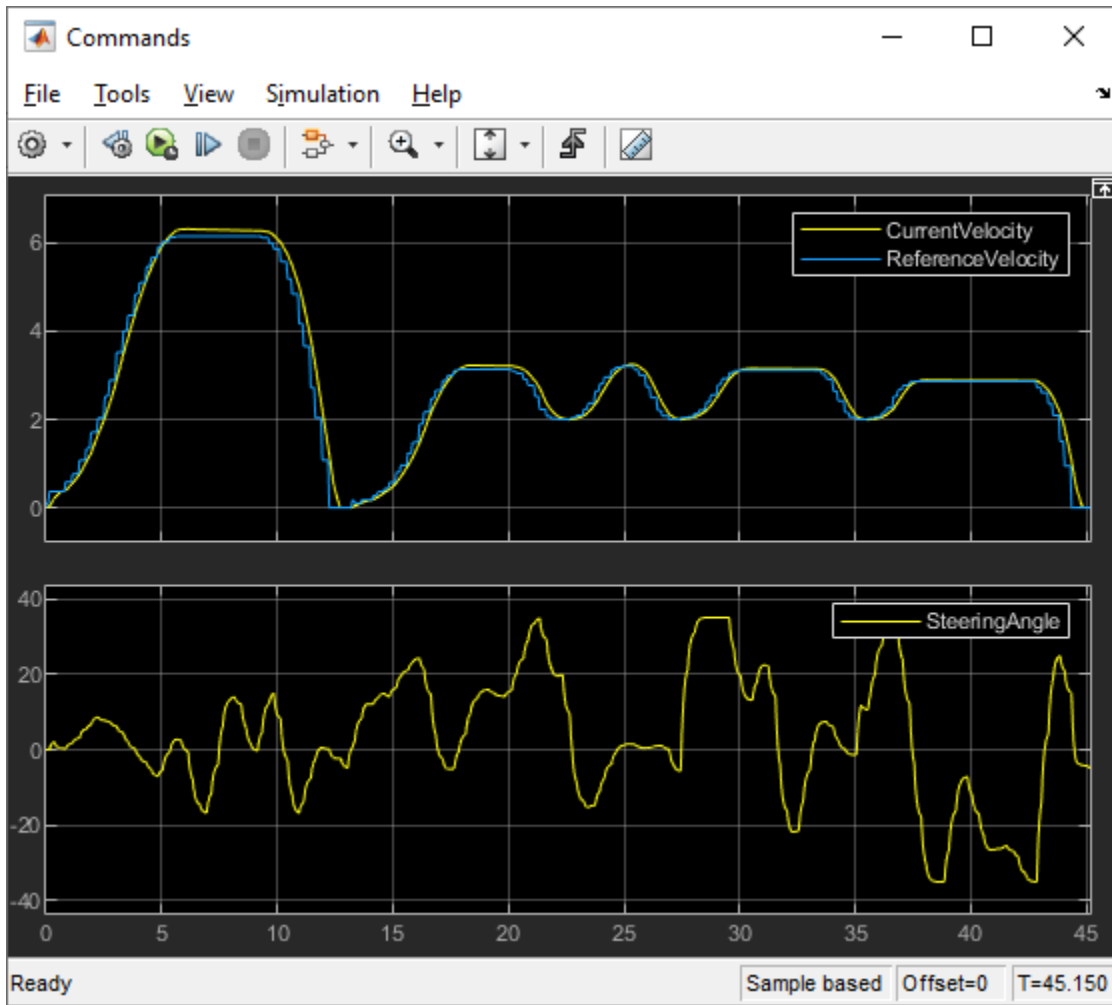
Automated Parking Valet with ROS 2 in Simulink

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. Then you generate code for the ROS 2 nodes and deploy them.

Prerequisites: "Automated Parking Valet" (Automated Driving Toolbox), "Generate a Standalone ROS 2 Node from Simulink" on page 2-99

Introduction

This autonomous vehicle application has the following components.
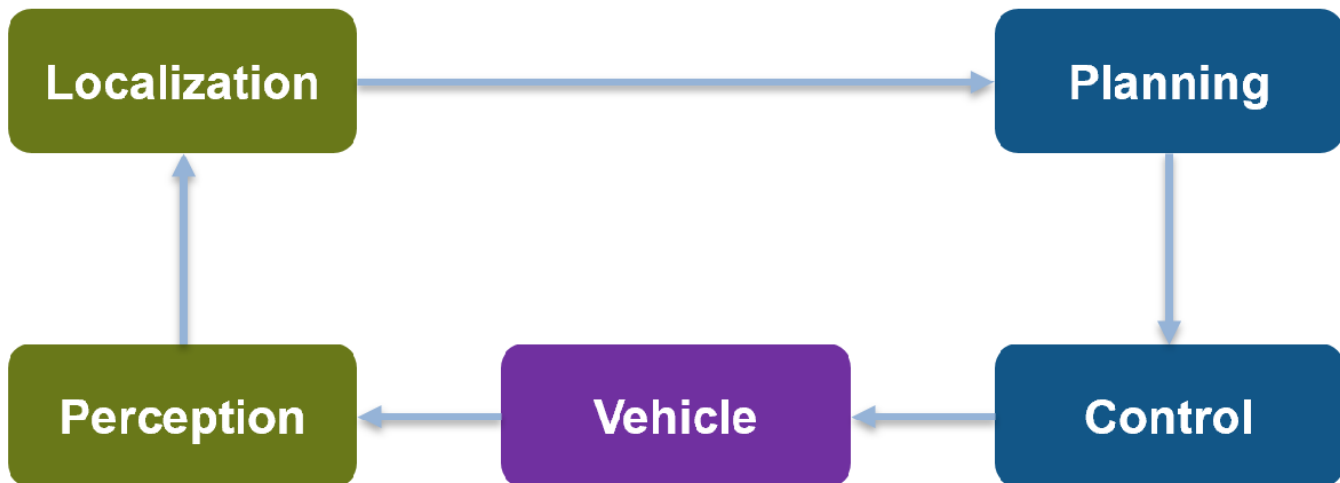
This example concentrates on simulating the *Planning*, *Control* and the *Vehicle* components. For *Localization*, this example uses pre-recorded localization map data. The *Planning* component is further divided into *Behavior planner* and *Path Planner* components. This results in a ROS 2 network 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.
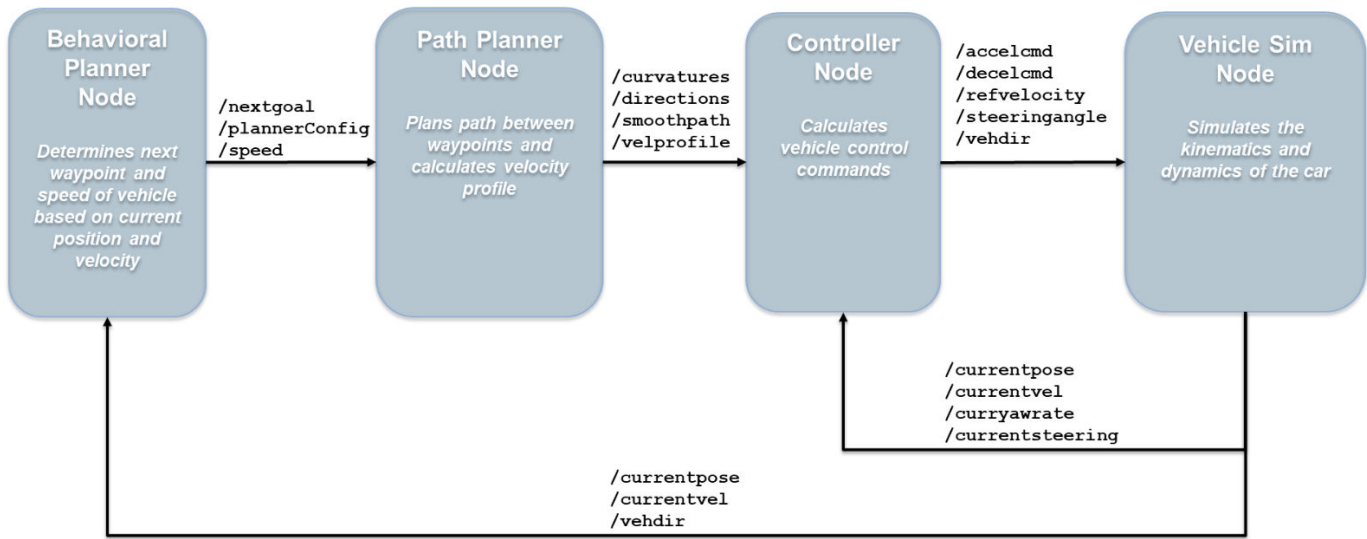
The example follows this general workflow from algorithm design to deployment:

1 Design a Simulink model without enabling ROS and verify that the controller algorithm provides accurate results

2 Replace the input and output ports with ROS 2 Subscribe and Publish blocks to prepare for actual deployment

3 Generate and deploy ROS 2 nodes

This example walks you through the simulation and code generation workflow for the automated parking valet components in these steps.

1 "Simulate Automated Parking Valet with ROS 2 in Simulink" on page 2-161 — Examine Simulink models for all the components, and simulate the automated parking valet workflow.

2 "Generate and Deploy ROS 2 Nodes for Automated Parking Valet in Simulink" on page 2-169 — Generate code and deploy the automated parking valet system components as ROS 2 nodes.

Generate and Deploy ROS 2 Nodes for Automated Parking Valet in Simulink

After simulating the ROS 2 nodes to verify all the components of the automated parking valet system as described in "Simulate Automated Parking Valet with ROS 2 in Simulink" on page 2-161, you can now generate code for those ROS 2 nodes and deploy them. Variant subsystem under each node will use publish and subscribe blocks to replace input and output ports.

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.

Recent change in GNU may result in a compilation error for ROS2ValetPathPlannerExample model. If you encounter any compilation issues, follow these steps to update the model:

1   Open the `ROS2ValetPathPlannerExample.slx` model.
2   Under the **Modeling** tab, click **Model Settings**.
3   In the **Simulation Target** panel, navigate to **Code information > Linker flags**.
4   Add `-Wl,--no-as-needed`. Click `Ok` to apply the change.

```
exampleHelperROSValetLoadLocalizationData;
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\avijayar\OneDrive - MathWorks\Documents\MATLAB\Ex
### Invoking Target Language Compiler on ROS2ValetBehavioralPlannerExample.rtw
### Using System Target File: Z:\25\avijayar.Bdoc23a.j2028781\matlab\rtw\c\ert\ert.tlc
### Loading TLC function libraries
.........
### Initial pass through model to cache user defined code
...
### Caching model source code
.............................................................................
### Writing header file ROS2ValetBehavioralPlannerExample_types.h
### Writing source file ROS2ValetBehavioralPlannerExample.cpp
### Writing header file ROS2ValetBehavioralPlannerExample_private.h
### Writing header file ROS2ValetBehavioralPlannerExample.h
### Writing header file rtwtypes.h
.
### Writing header file rtGetNaN.h
### Writing source file rtGetNaN.cpp
### Writing header file rt_nonfinite.h
### Writing source file rt_nonfinite.cpp
.
### Writing header file rtGetInf.h
### Writing source file rtGetInf.cpp
### Writing header file rtmodel.h
### Writing source file ROS2ValetBehavioralPlannerExample_data.cpp
### Writing source file ert_main.cpp
### TLC code generation complete (took 4.5s).
.### Saving binary information cache.
```

```
### Using toolchain: Colcon Tools
Creating a Python virtual environment.Done.
Adding required Python packages to virtual environment.Done.
Copying include folders.Done.
Copying libraries.Done.
### Building 'ROS2ValetBehavioralPlannerExample': all
Running colcon build in folder 'C:/Users/avijayar/OneDrive - MathWorks/Documents/MATLAB/ExampleMa
Build in progress. This may take several minutes...
Build succeeded.Success
### Successfully generated all binary outputs.
### Creating HTML report file index.html
### Successful completion of build procedure for: ROS2ValetBehavioralPlannerExample
### Simulink cache artifacts for 'ROS2ValetBehavioralPlannerExample' were created in 'C:\Users\av

Build Summary

Top model targets built:

Model                                Action                          Rebuild Reason
===================================================================================================
ROS2ValetBehavioralPlannerExample  Code generated and compiled.  Global variable #SL_Bus_ROSVaria

1 of 1 models built (0 models already up to date)
Build duration: 0h 2m 44.093s
### Starting build procedure for: ROS2ValetPathPlannerExample
### Generating code and artifacts to 'Model specific' folder structure
### Generating code into build folder: C:\Users\avijayar\OneDrive - MathWorks\Documents\MATLAB\Ex
### Generated code for 'ROS2ValetPathPlannerExample' is up to date because no structural, paramet
### Recompiling code generation target for model ROS2ValetPathPlannerExample because a referenced
### Using toolchain: Colcon Tools
### Building 'ROS2ValetPathPlannerExample': all
Running colcon build in folder 'C:/Users/avijayar/OneDrive - MathWorks/Documents/MATLAB/ExampleMa
Build in progress. This may take several minutes...
Build succeeded.Success
### Successfully generated all binary outputs.
### Creating HTML report file index.html
### Successful completion of build procedure for: ROS2ValetPathPlannerExample

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 56.515s
### Starting build procedure for: ROS2ValetControllerExample
### Generating code and artifacts to 'Model specific' folder structure
### Generating code into build folder: C:\Users\avijayar\OneDrive - MathWorks\Documents\MATLAB\Ex
### Generated code for 'ROS2ValetControllerExample' is up to date because no structural, paramete
### Recompiling code generation target for model ROS2ValetControllerExample because a referenced
### Using toolchain: Colcon Tools
### Building 'ROS2ValetControllerExample': all
Running colcon build in folder 'C:/Users/avijayar/OneDrive - MathWorks/Documents/MATLAB/ExampleMa
Build in progress. This may take several minutes...
Build succeeded.Success
```

```
### Successfully generated all binary outputs.
### Creating HTML report file index.html
### Successful completion of build procedure for: ROS2ValetControllerExample

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 7.4935s
```

```
VEHICLE_MODE=1; % switch variant subsystem in Vehicle node to use ROS 2 network
open_system("ROS2ValetVehicleExample");
set_param("ROS2ValetVehicleExample","SimulationCommand","start");
```
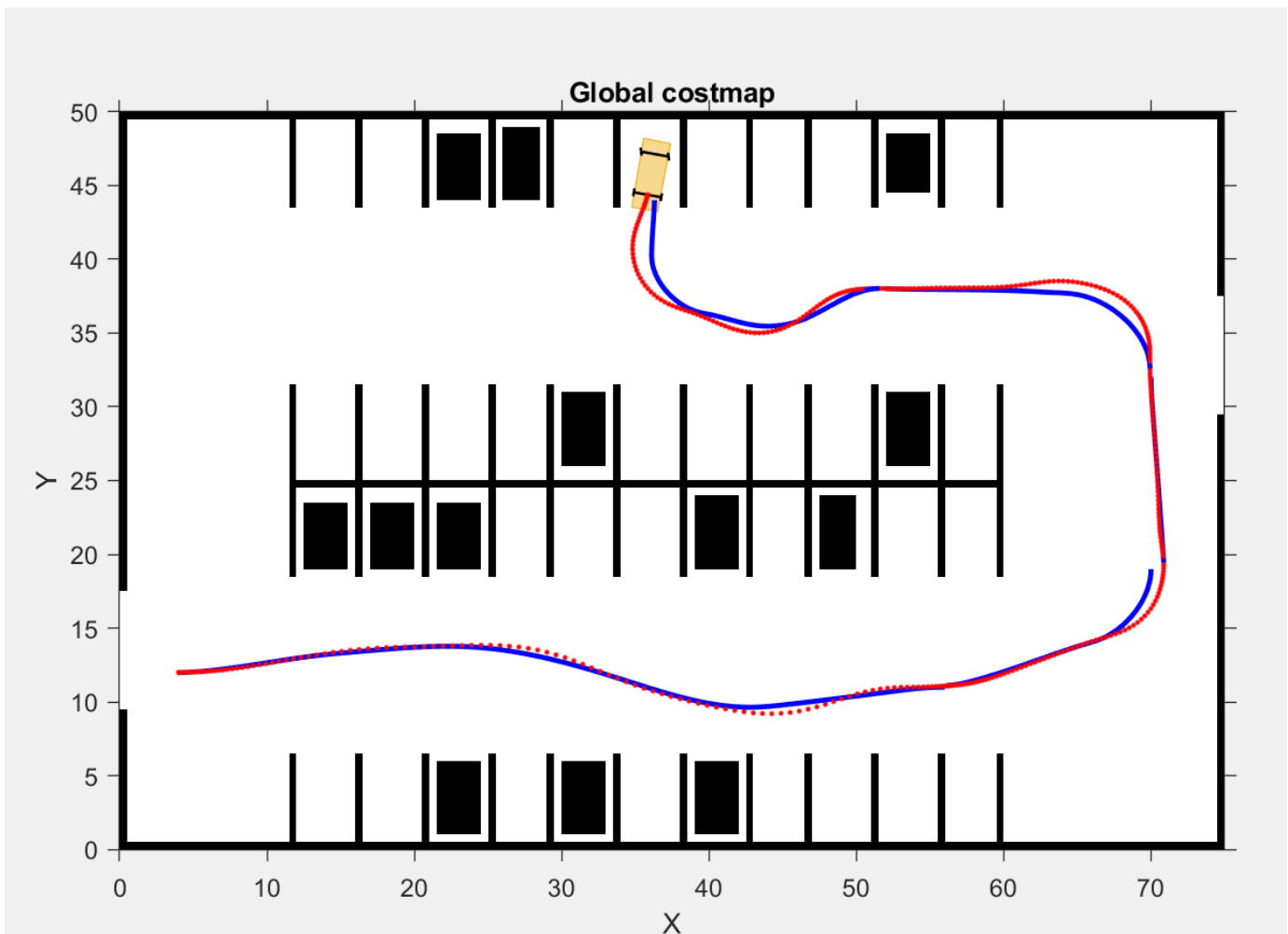
Switching Between ROS Middleware Implementations

ROS 2 is based on the Data Distribution Service (DDS™) standard, an end-to-end middleware protocol that provides features such as discovery, serialization, and transportation. ROS 2 supports multiple ROS middleware (RMW) implementations so that you can choose one that best suits your requirements. Use these factors to inform your choice of middleware implementation: platform availability, performance characteristics, computation footprint, dependencies, and license.

You can switch between RMW implementations for using DDS while working with ROS 2 in MATLAB® and Simulink®, by using ROS Toolbox Preferences. ROS Toolbox contains these RMW implementations.

1   `rmw_fastrtps_cpp` — eProsima Fast DDS (Version 2.5.0). This is the default RMW implementation.

2   `rmw_fastrtps_dynamic_cpp` — eProsima Fast DDS.

3   `rmw_cyclonedds_cpp` — Eclipse Cyclone DDS (Version 0.7.0).

You can also configure and switch to these custom RMW implementations.

1   `rmw_connextdds` — RTI Connext DDS (version 6.0.1 or later). This implementation is supported on Windows®, Linux®, and macOS® operating systems.

2   `rmw_iceoryx_cpp` — Eclipse Iceoryx Middleware (version 1.0.1). This implementation is supported on Linux® and macOS® operating systems.

Additionally, you can switch to other custom RMW implementations based on dynamic or static introspection type support in ROS 2 such as

1   `rmw_ecal_dynamic_cpp` — eCAL Middleware. This is based on dynamic introspection type support in ROS 2 and implements generic functions to pass a message to be published.

2   `rmw_ecal_proto_cpp` — eCAL Middleware. This is based on static type support in ROS 2 and requires a static ROS IDL (Interface Definition Language) type support package for building. This implements middleware specific functions for each message type to be published to a topic.

Configure RMW Implementations Shipped with ROS Toolbox

1. In the **Environment** section of the **Home** tab in the MATLAB toolstrip, click **Preferences** and select **ROS Toolbox**.
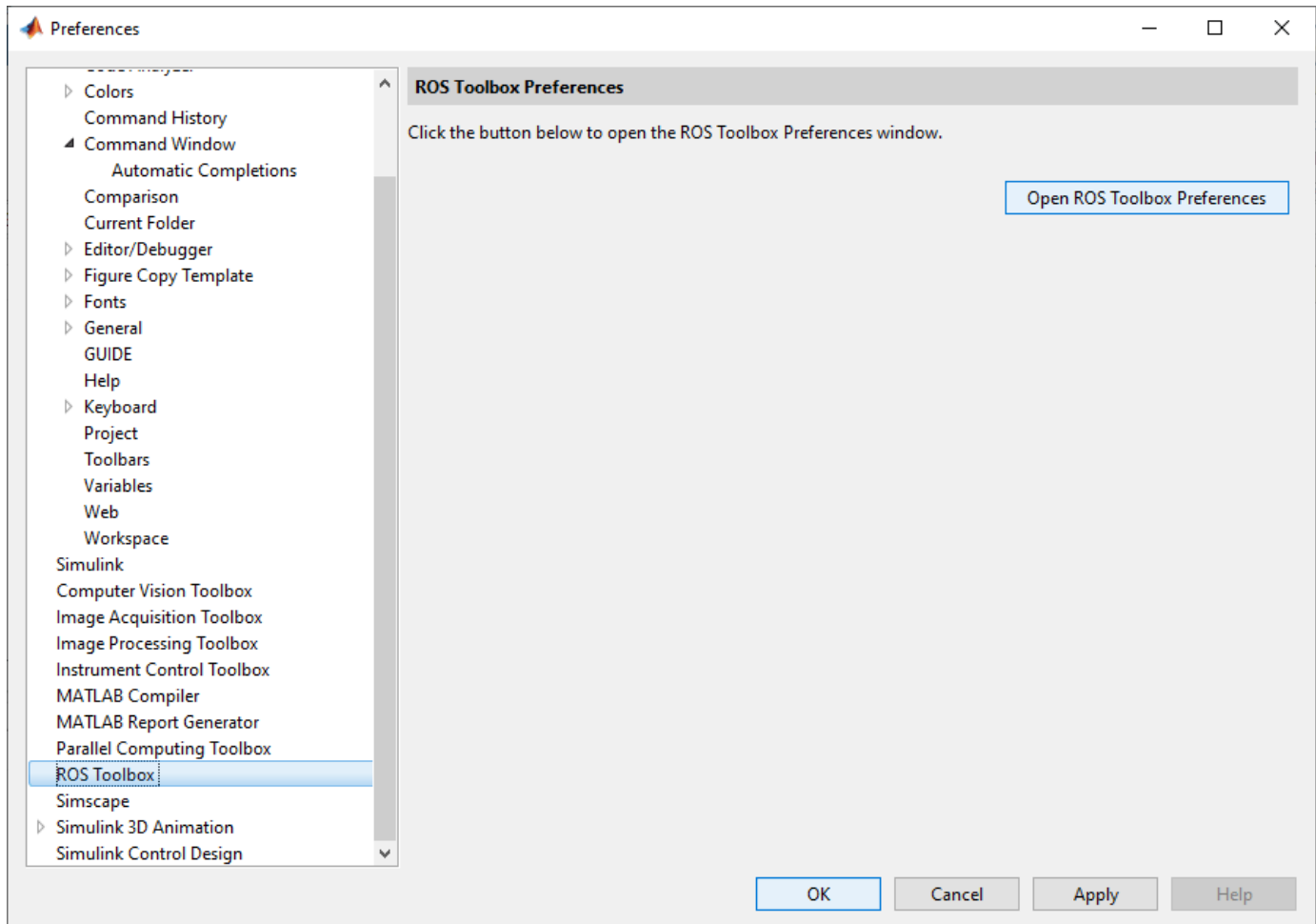
2. Click **Open ROS Toolbox Preferences**, specify the path to the Python executable, and click **Recreate Python Environment**. If you have configured Python already, proceed with the next step.

3. Open the dropdown list under **ROS Middleware (RMW) Implementation** to switch between RMW implementations.

Select an implementation from the list and click **OK**. This enables you to create a ROS 2 node with the selected implementation.
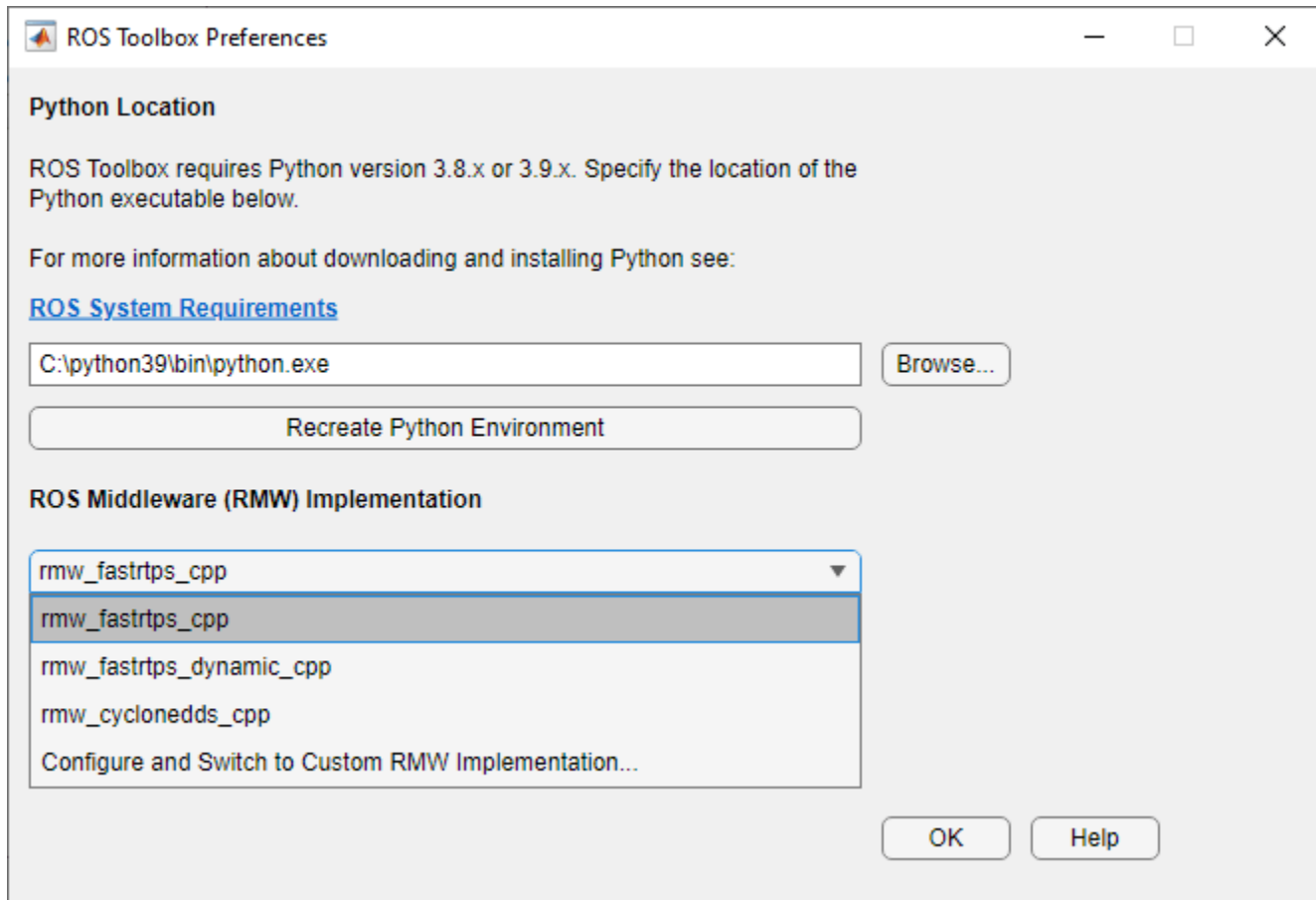
Configure Custom RMW Implementations

1. Open the **ROS Toolbox Preferences** window using the above steps and create the Python environment.

2. Select **Configure and Switch to Custom RMW Implementation** in the dropdown list under **ROS Middleware (RMW) Implementation** to launch the **ROS Middleware Configuration** dialog box.
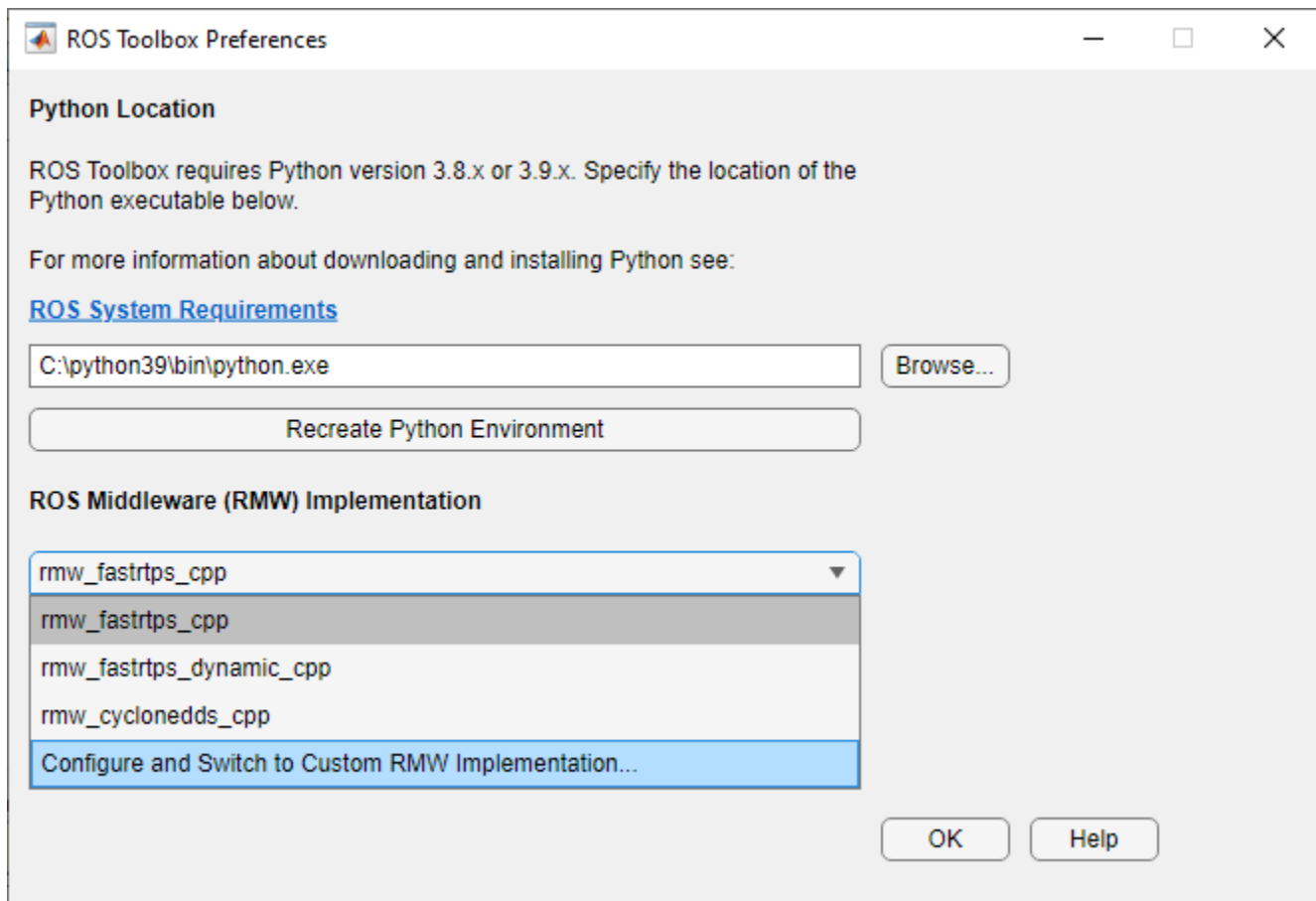
3. This enables you to switch to a custom implementation like `rmw_connextdds` or `rmw_iceoryx_cpp` from the RMW Implementation dropdown menu.
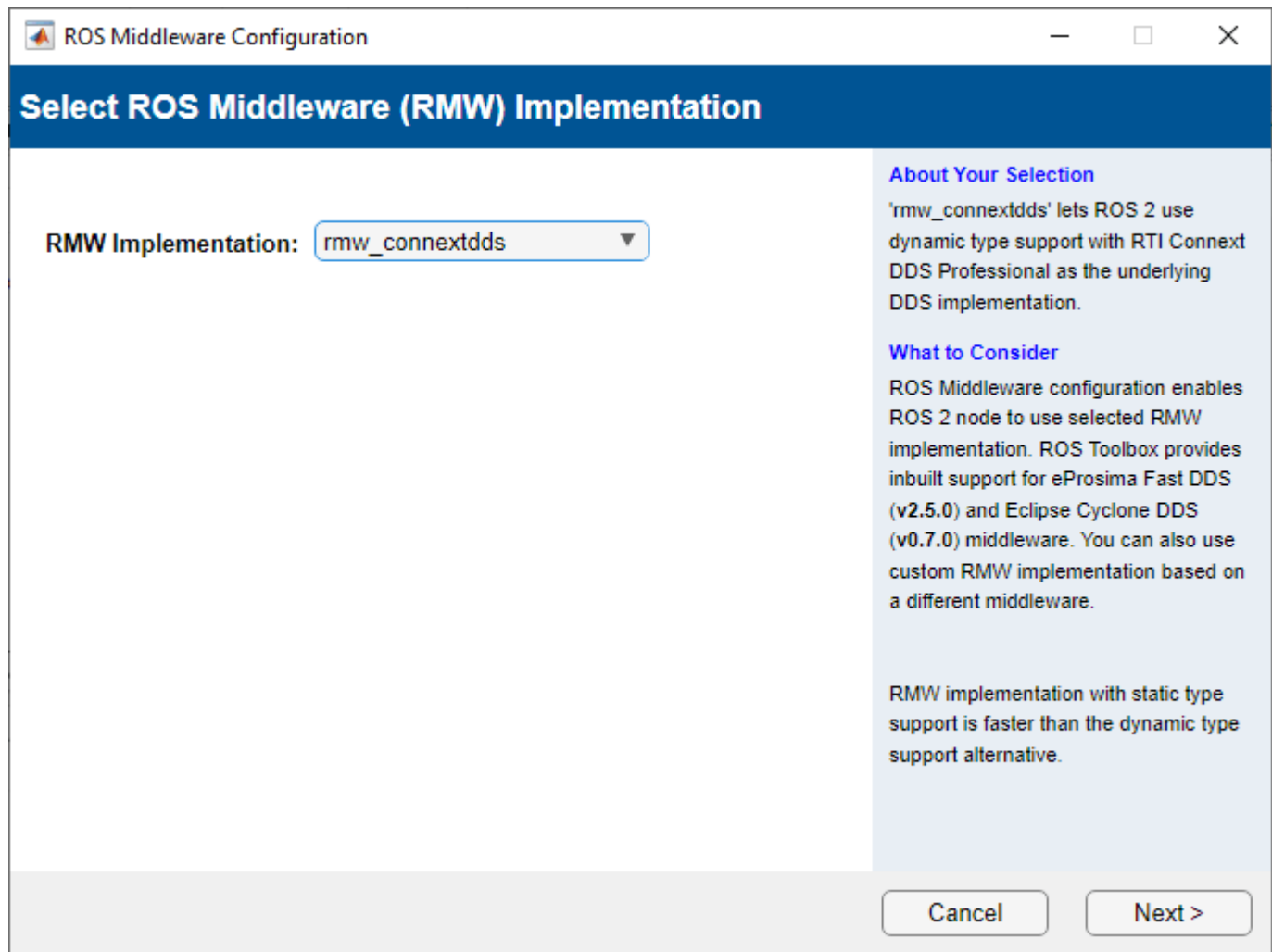
4. To set up `rmw_connextdds`, verify the underlying DDS installation (RTI Connext DDS). For more information, see the links in the validation window.

ROS Middleware Configuration — □ ✕

Validate DDS Installation

Ensure that RTI Connext DDS Professional is available on your host platform by following the instructions for Downloading Connext DDS and Setting Up a License.

Once the RTI Connext Professional installation is available, click 'Browse' to select the installation directory on your host platform and then click 'Verify installation'.

C:\Program Files\rti_connext_dds-6.0.1 | Browse

Verify Installation

✓ RTI Connext DDS Professional installation is valid. Click 'Next' to build and register 'rmw_connextdds' with MATLAB/Simulink.

What to Consider

'rmw_connextdds' requires RTI Connext DDS Professional (version 6.0.1 or later) to be installed on your host platform.

< Back   Cancel   Next >

5. Click **Browse** and navigate to the parent directory of the DDS package. Build the RMW implementation package and then click **Next** if the build is successful.
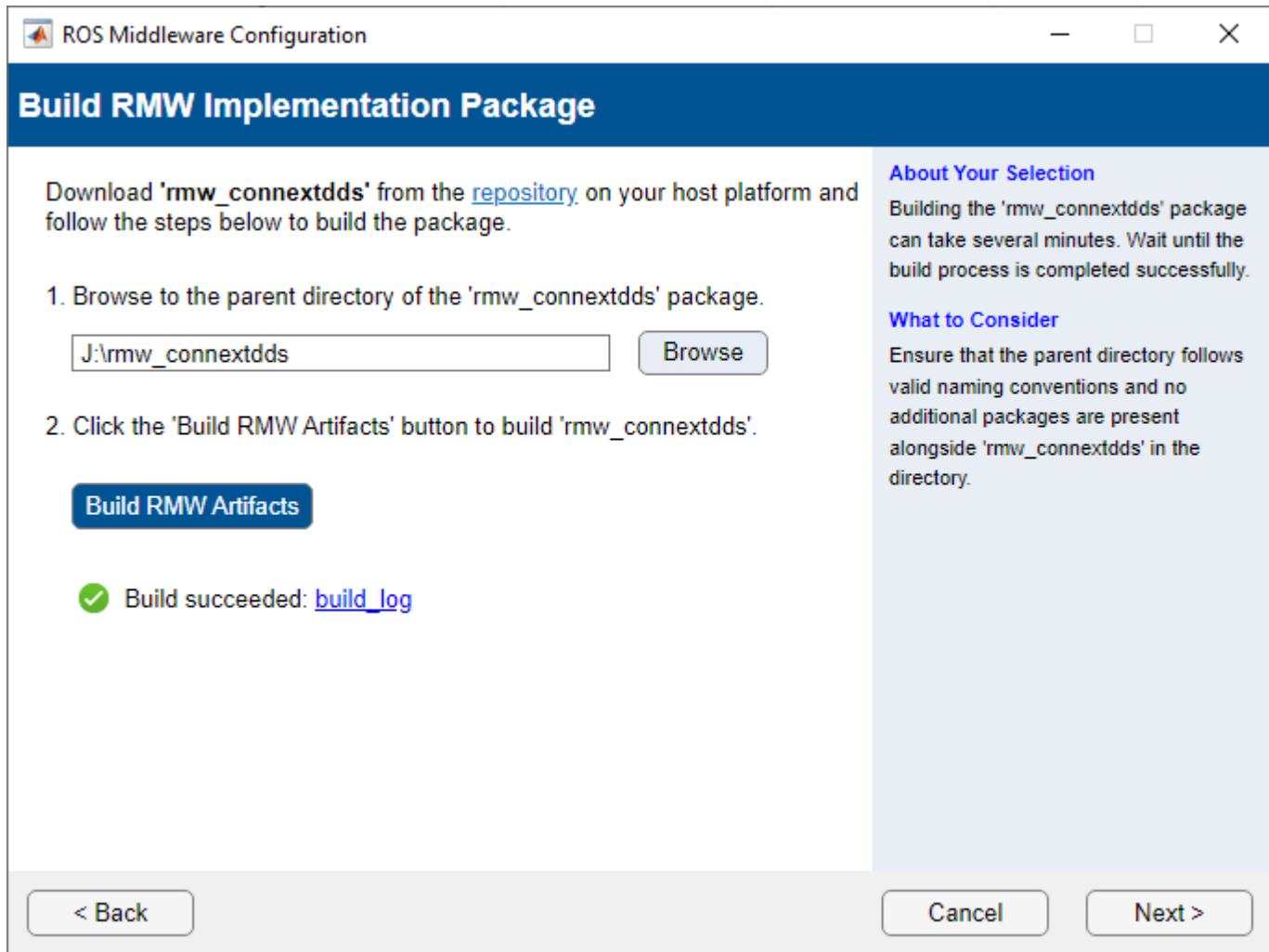
6. Click **Test RMW** to validate if ROS 2 node creation is successful with the custom RMW implementation set to `rmw_connextdds`.
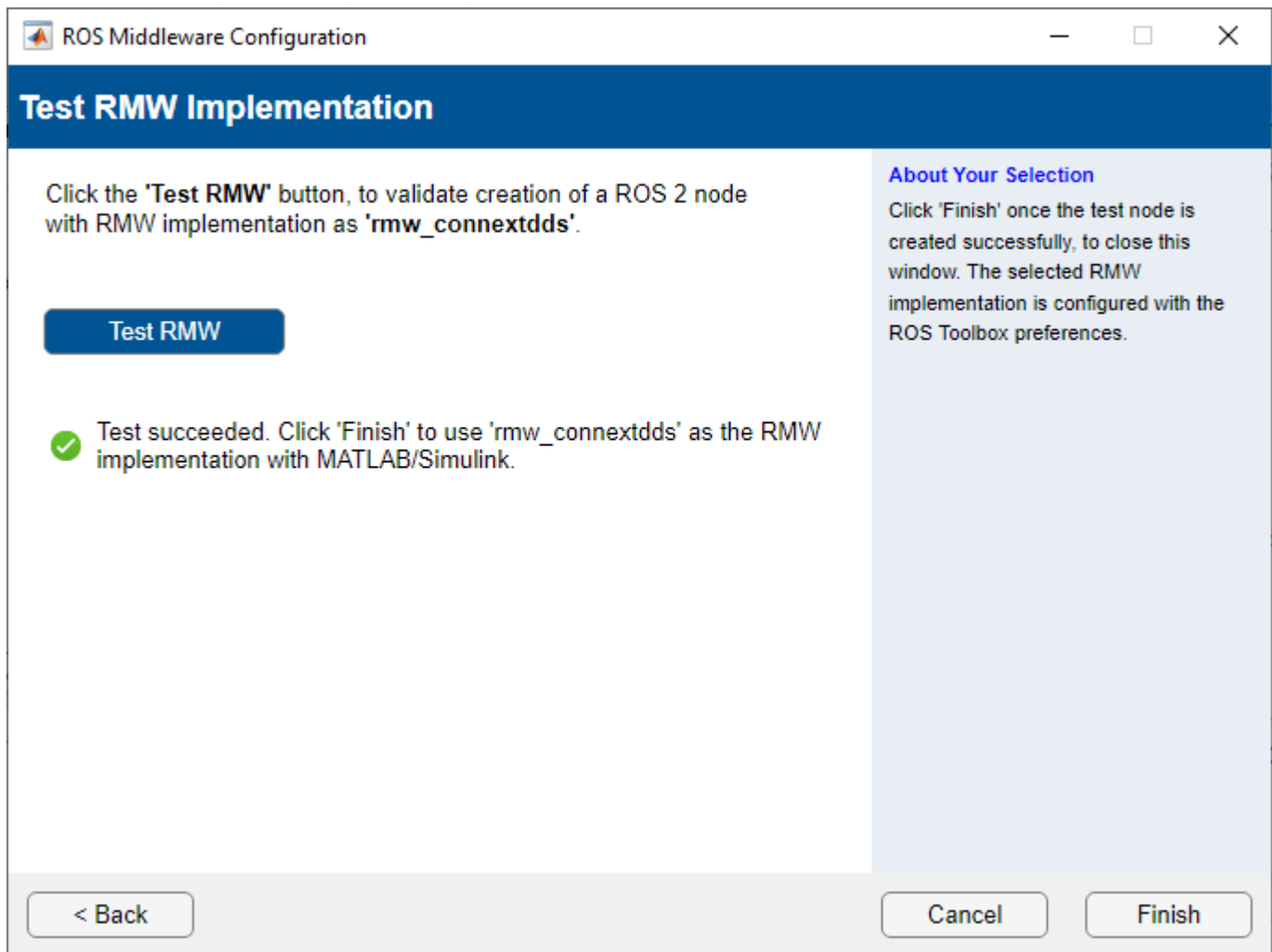
7. The selected **ROS Middleware (RMW) Implementation** is now registered as default in the dropdown menu. Click **OK** to set this RMW implementation.
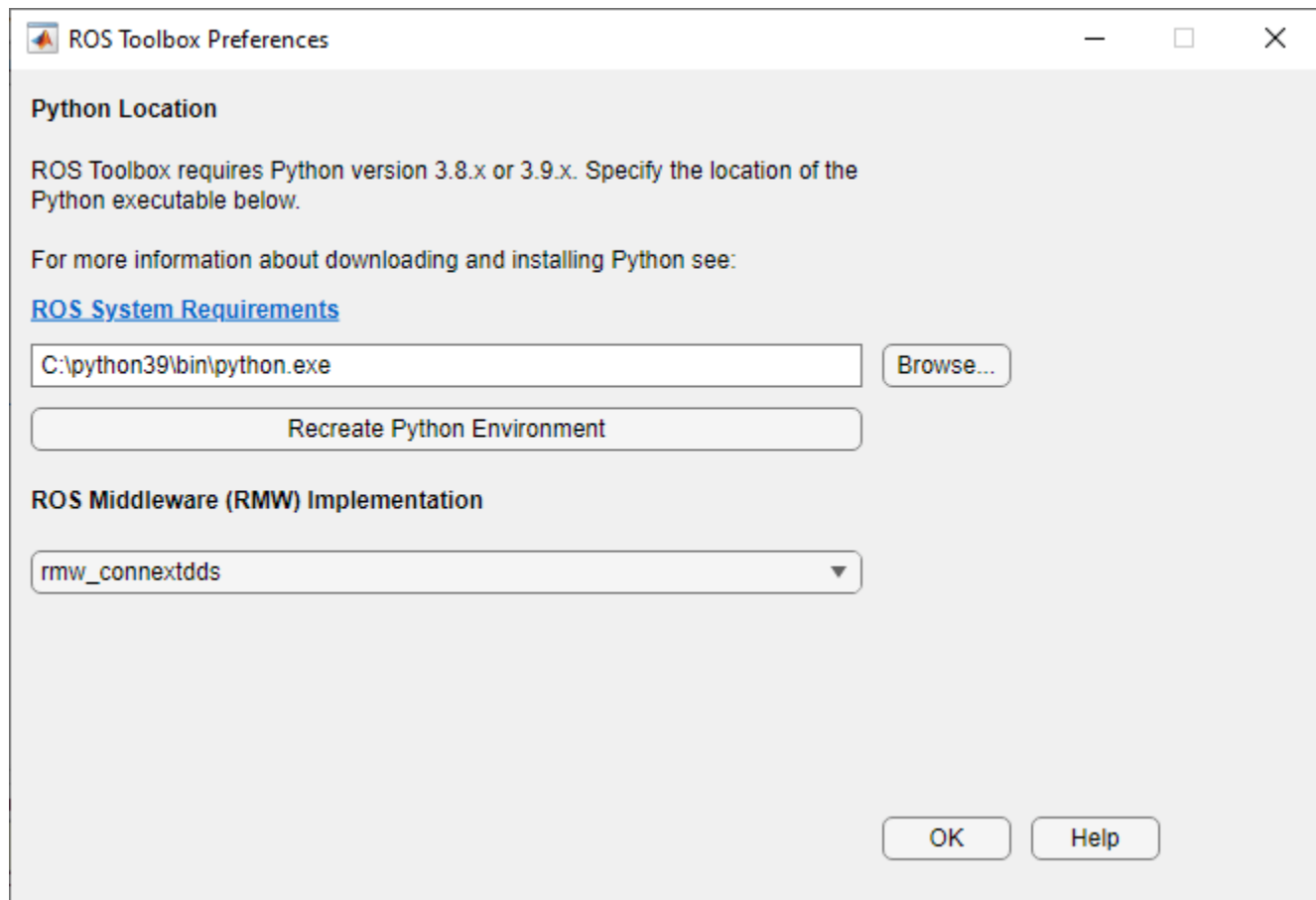
Configure Other RMW Implementations Based on Dynamic or Static Introspection Type Support

1. Open the **ROS Toolbox Preferences** window using the above steps and create the Python environment.

2. Select **Configure and Switch to Custom RMW Implementation** to launch the **ROS Middleware Configuration** dialog box. Select **other** in the dropdown to provide the folderpath containing one or multiple RMW implementations, based on dynamic or static introspection type support. Click **Next** and proceed to validate their existence in the location.
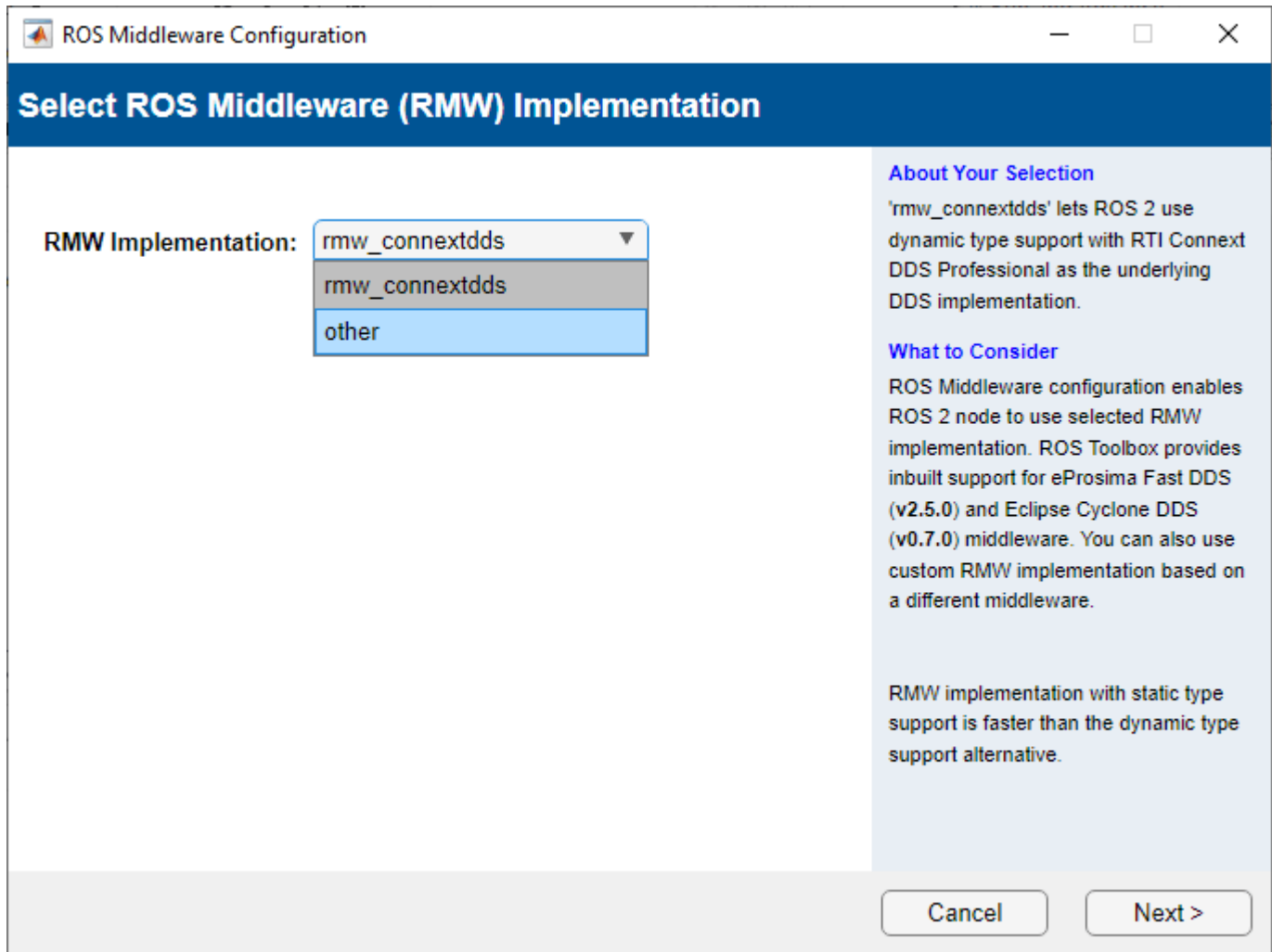
3. Providing the folderpath to a particular RMW implementation enables you to proceed with its configuration.
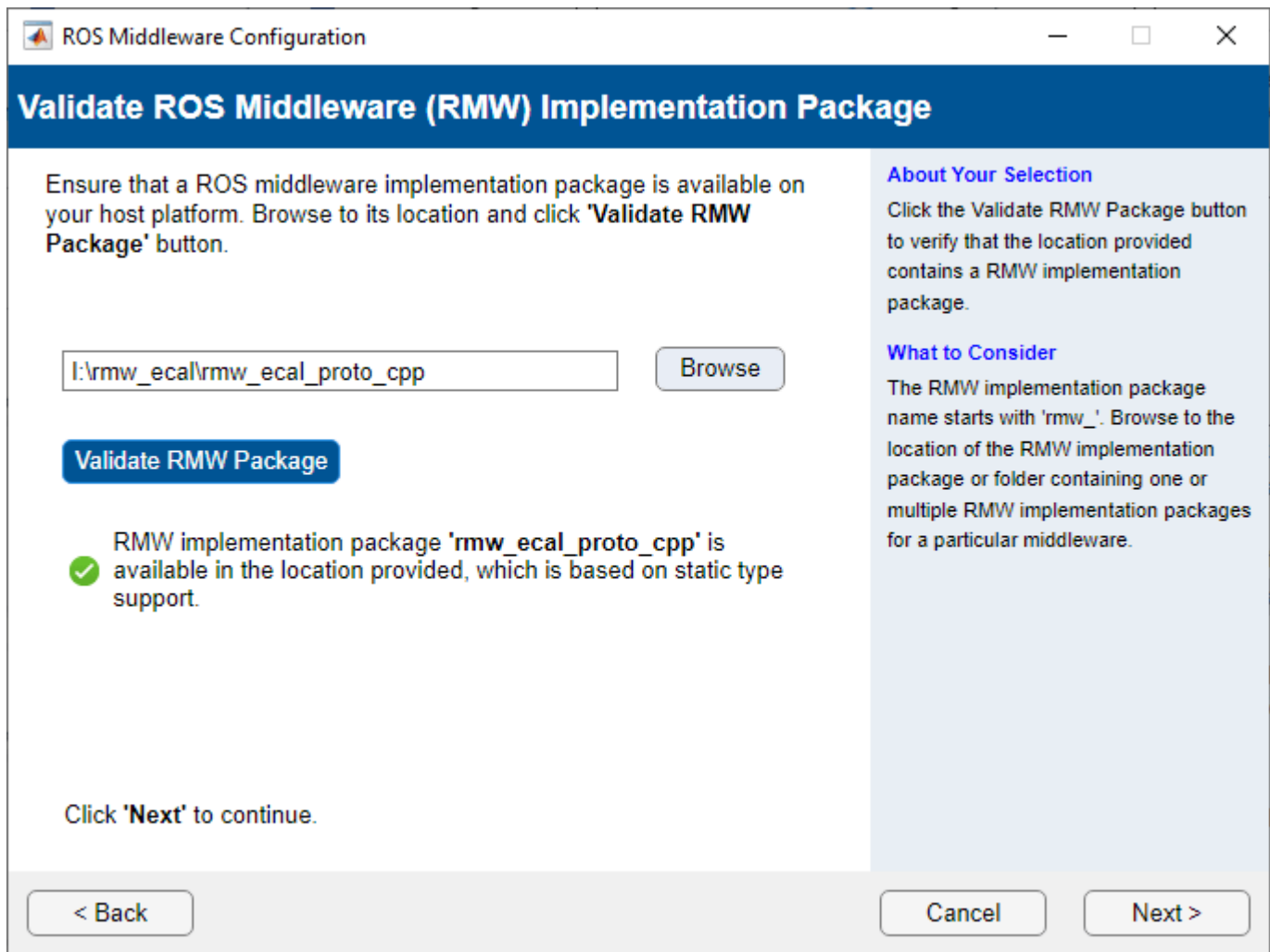
4. If the folder contains multiple RMW implementations, you can choose one or all from the dropdown in the subsequent interface. Click **Next** upon making the selection.
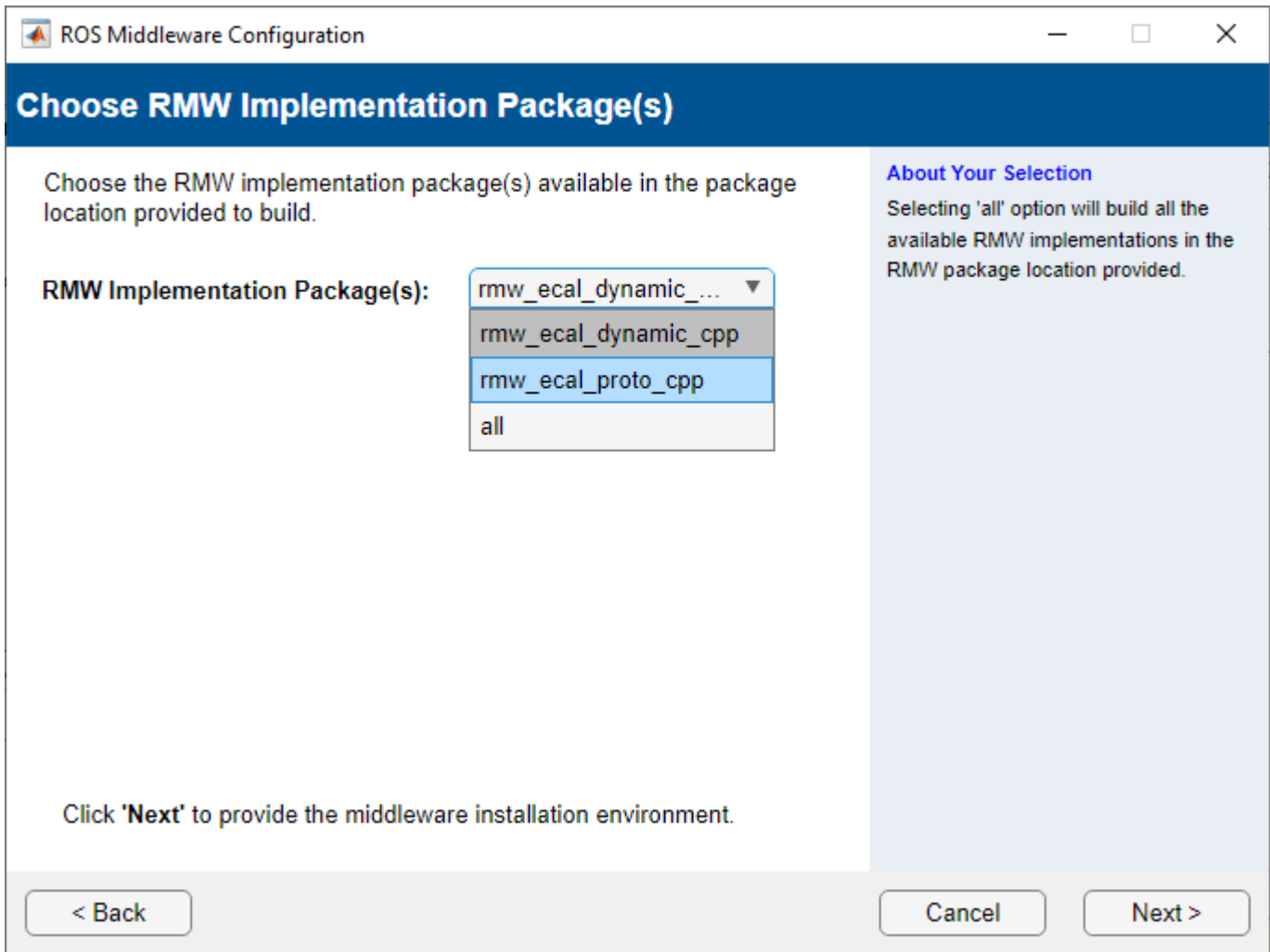
Note: Building RMW implementation based on static type support requires static ROS IDL type support package. For example, `rmw_ecal_proto_cpp` requires `rosidl_typesupport_protobuf`, which uses Protobuf for serialization and deserialization of ROS 2 messages.
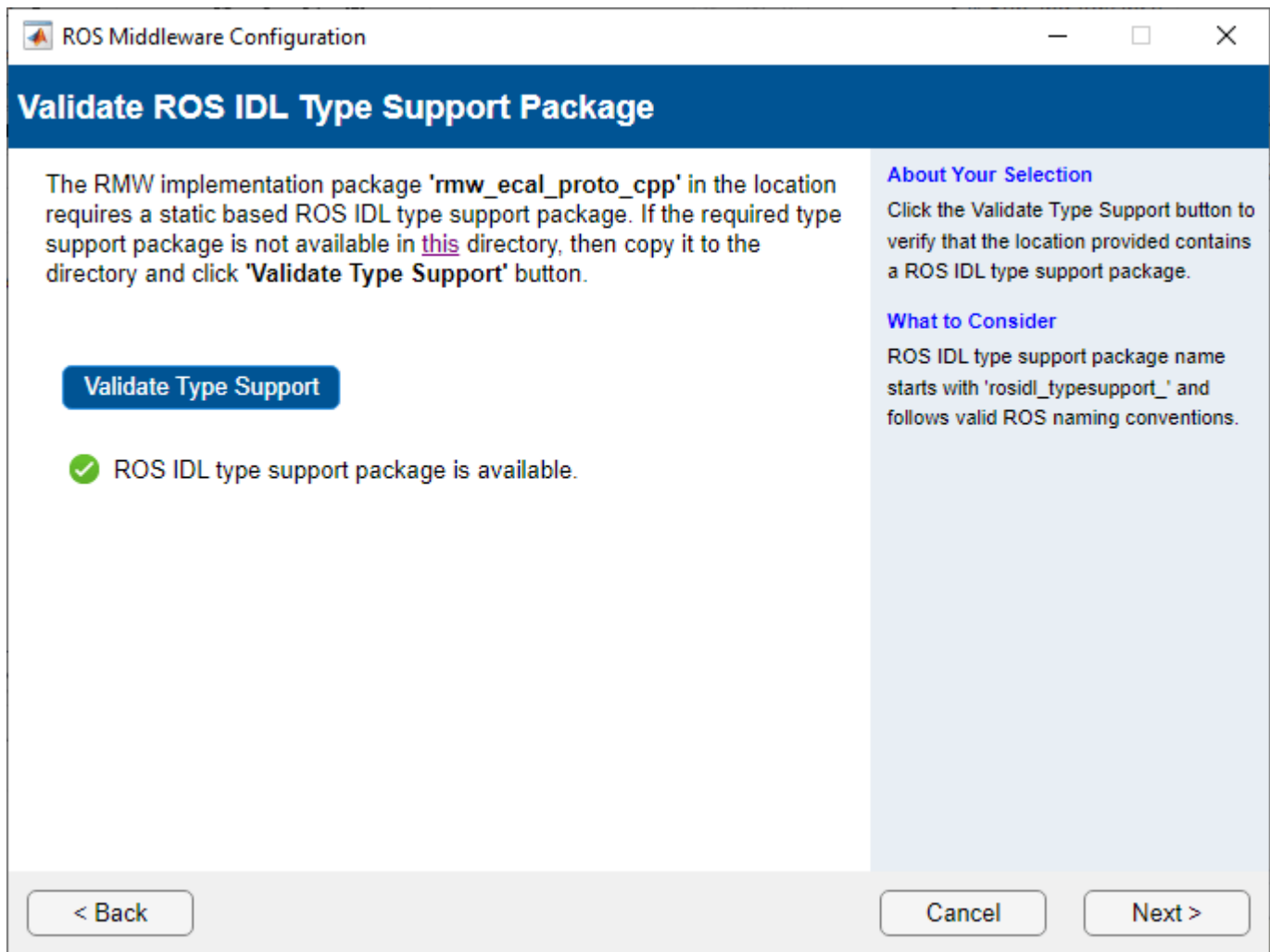
5. Provide the parent path to middleware installation and set the environment variables required to build the RMW implementation package or packages.
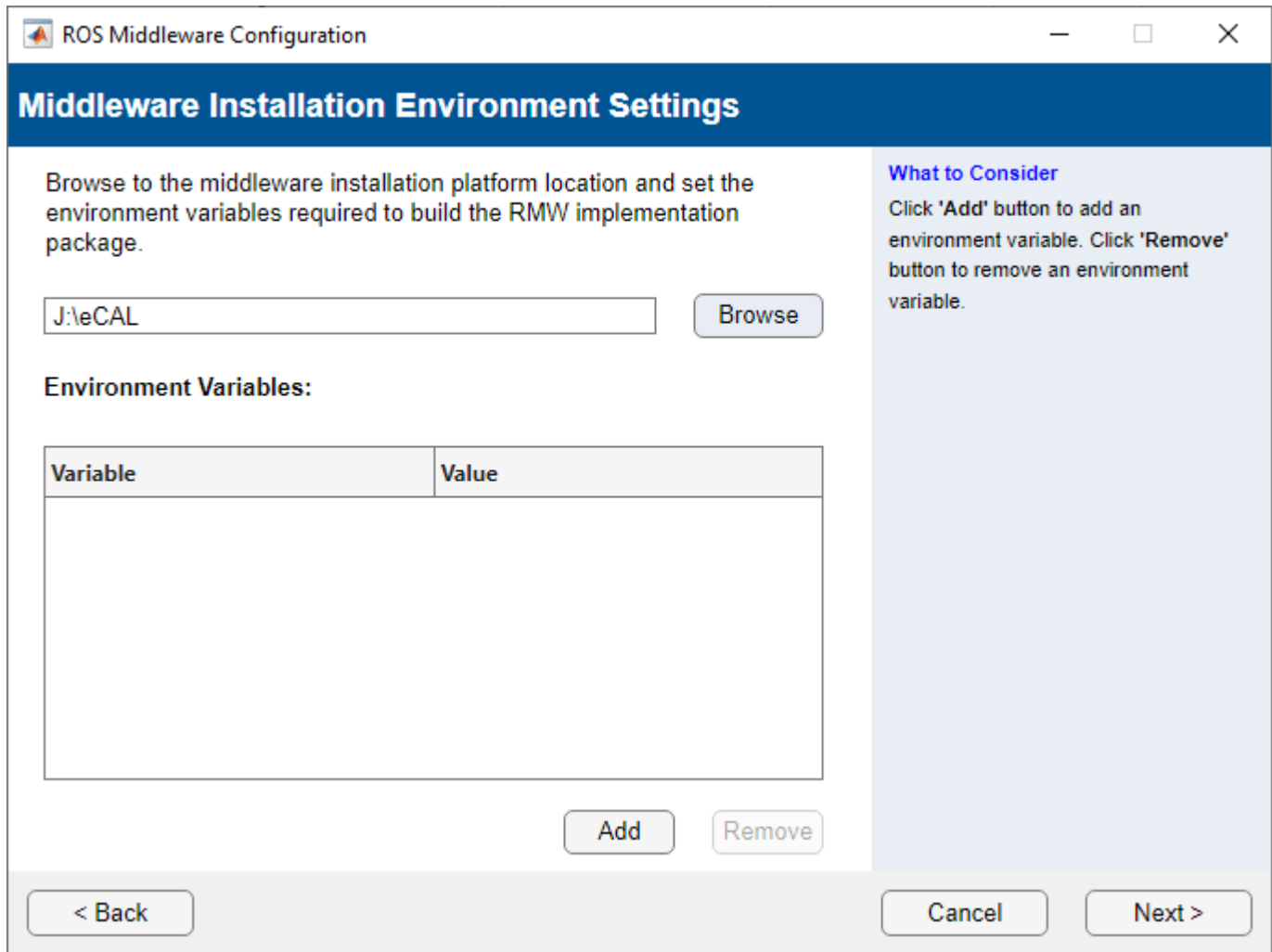
6. Follow subsequent steps as earlier to build the RMW implementation package.

7. Additionally, build the ROS 2 message packages and custom messages to create ROS 2 node and work with ROS 2 APIs or Simulink Blocks on the selected RMW implementation.
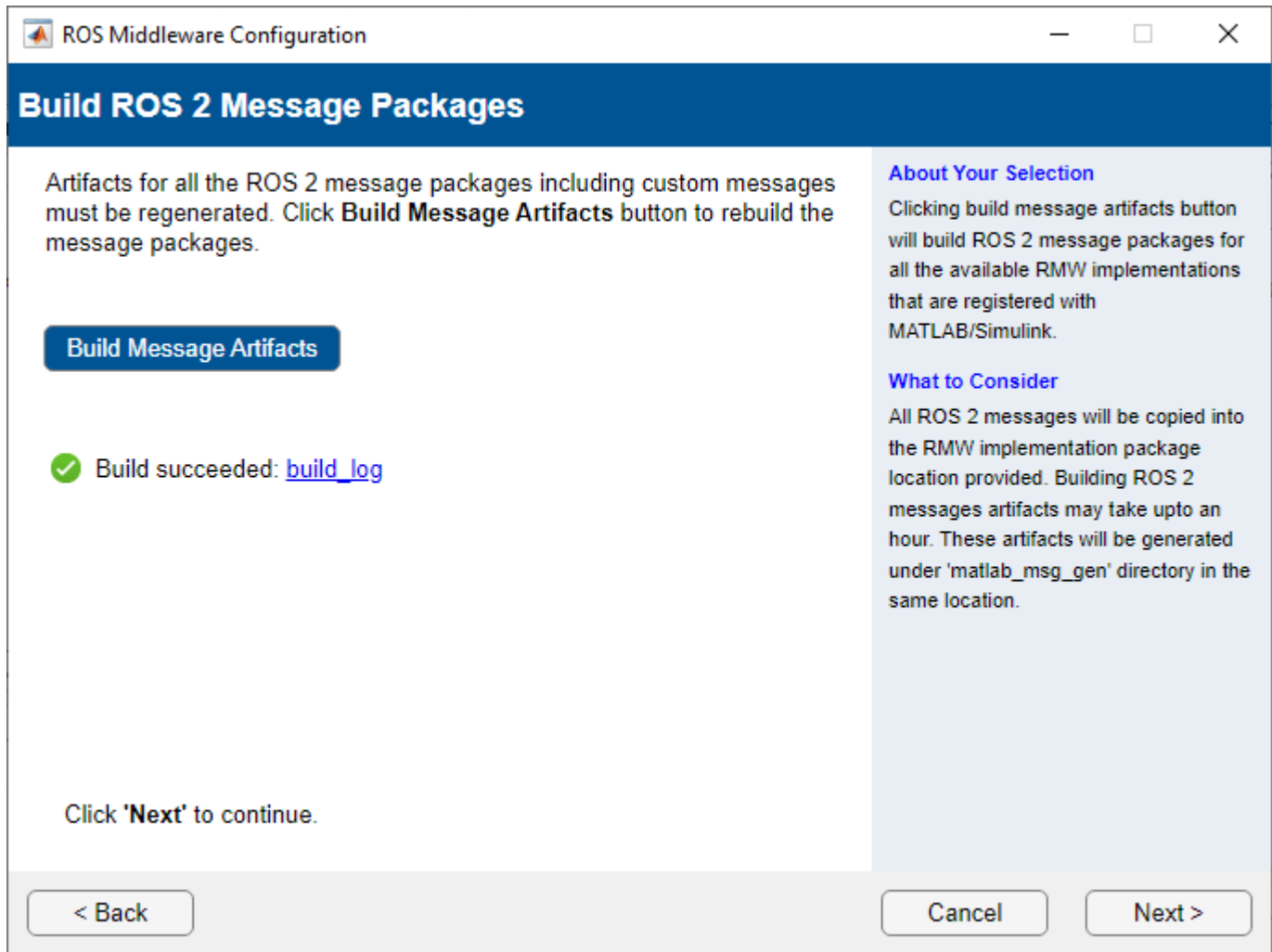
8. Test it to validate the ROS 2 node creation with the RMW implementation set to
`rmw_ecal_proto_cpp`.

9. The selected **ROS Middleware (RMW) Implementation** is now registered as default in the dropdown menu. Click **OK** to set this RMW implementation.

ROS Toolbox Preferences

Python Location

ROS Toolbox requires Python version 3.8.x or 3.9.x. Specify the location of the Python executable below.

For more information about downloading and installing Python see:

ROS System Requirements

C:\python39\bin\python.exe   Browse...

Recreate Python Environment

ROS Middleware (RMW) Implementation

rmw_ecal_proto_cpp

OK   Help
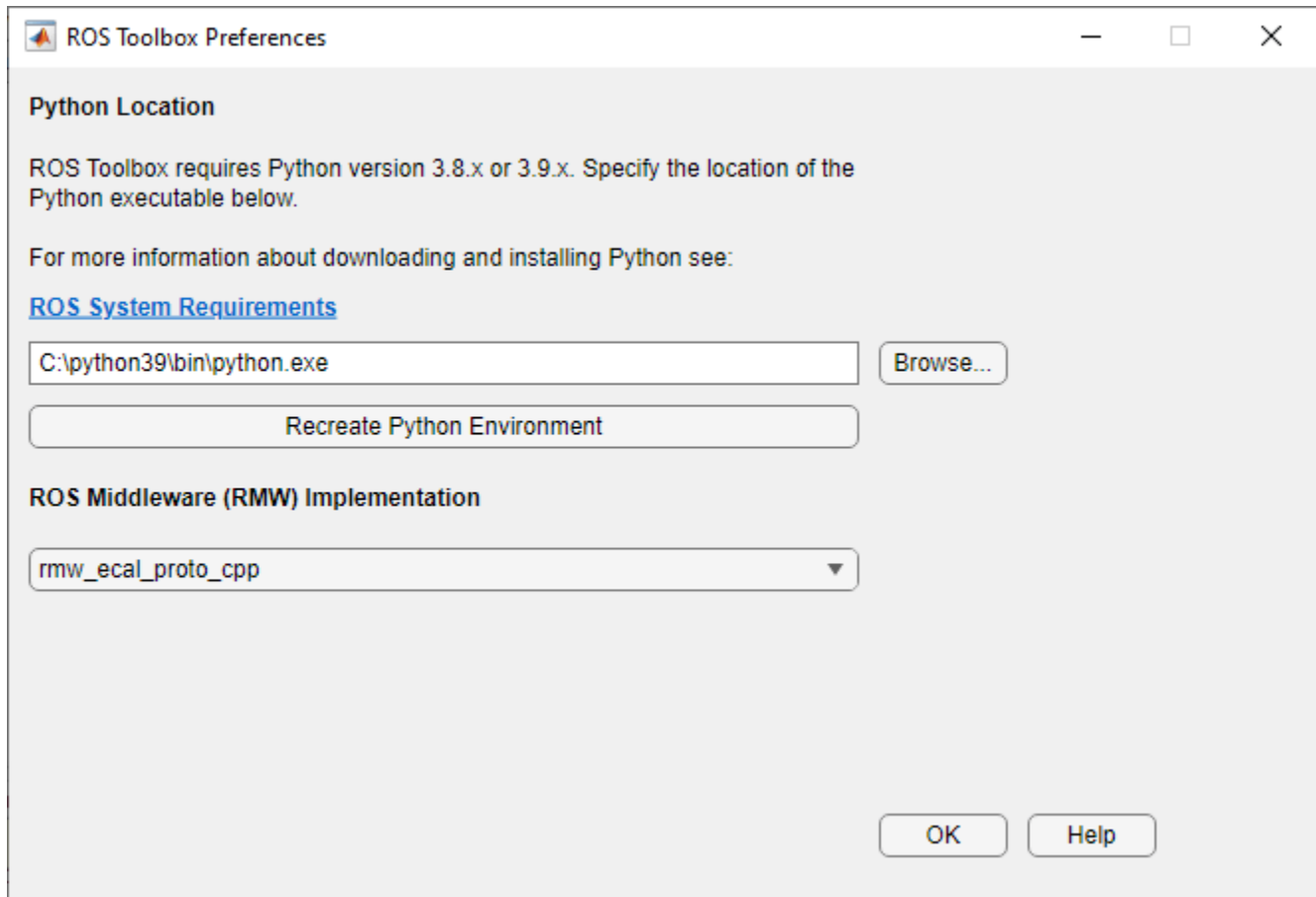
Use these steps to configure RMW implementations based on other middleware as well, such as openDDS, GurumDDS and further ones that are supported in ROS 2.

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 "ROS 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 "ROS in MATLAB".

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.
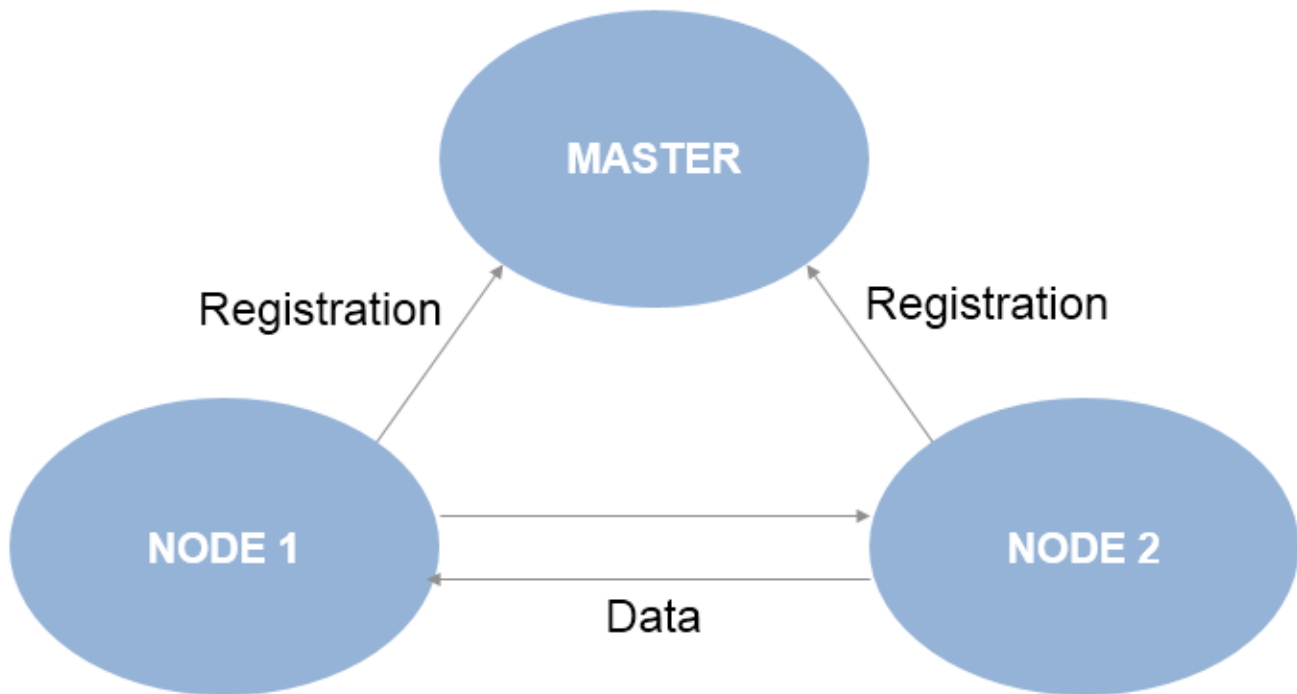
Each node registers its own *Node URI* with the master. Other participants in the ROS network will use this URI to contact the node. Again, this URI must be reachable by every other node in the ROS network. To create a node in MATLAB, call `rosinit`. If a ROS master is already set up, MATLAB detects 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 1-39
- "Connect to a ROS Network" on page 1-44
- "Robot Operating System (ROS)"

Built-In Message Support

In this section...
"ROS Messages" on page 3-4
"Limitations of ROS Messages in MATLAB" on page 3-5
"ROS Data Type Conversions" on page 3-6
"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 2-13.

ROS Messages

In MATLAB, ROS messages are stored as message structures or message objects. Message structures are the recommended format as they have better performance over objects when performing initial creation, reading them from rosbag files, accessing nested properties, and performing communication operations over the ROS network. Also, message structures are the only supported message format when generating code through MATLAB Coder™. For more information on the ROS messages in MATLAB, see "Work with Basic ROS Messages" on page 1-52.

ROS Message Structures

A ROS message structure is stored as a MATLAB structure data type with fields based on the message type. Each message type has a specific set of fields with their corresponding values that are individually stored and accessed. You can specifically point to and modify each field on its own. The `MessageType` field of each message contains the message type as a character vector. Also, you can use the `rosShowDetails` function to view the contents of the message.

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

```
pointMsg = rosmessage('geometry_msgs/Point','Dataformat','struct')

pointMsg =

  struct with fields:

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

You can access and modify each field by using the `pointMsg` structure.

```
pointMsg.Y = 2

pointMsg =

  struct with fields:
```

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

ROS Message Objects

For ROS message objects, 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
```

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 3-6.

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 3-26.

```
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

rosmsg | rosmessage | showdetails

Related Examples

* "Work with Basic ROS Messages" on page 1-52
* "Exchange Data with ROS Publishers and Subscribers" on page 1-59
* "Work with Specialized ROS Messages" on page 1-92

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. For this example, a sample network is already set up with an existing transformation tree. You must specify the IP address of your ROS device, on which the transformations are published.

```
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)

In this section...
"Introduction" on page 3-14
"MATLAB rosbag Structure" on page 3-14
"Workflow for rosbag Selection" on page 3-15
"Limitations" on page 3-17

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
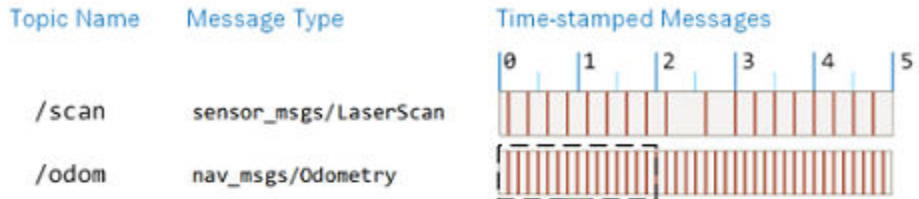
Select messages

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

select()

/odom   nav_msgs/Odometry

Extract message data

Select Message Indices

readMessages()

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

timeseries()

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

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
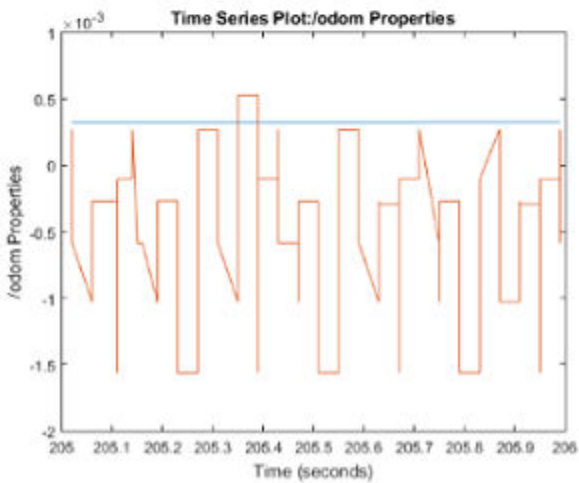
Limitations

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
rosbag | readMessages | BagSelection

Related Examples

- "Work with rosbag Logfiles" on page 1-79

Publish Variable-Length Nested ROS Messages and Deploy as ROS Node Using MATLAB

This example shows how to work with complex ROS messages in MATLAB, such as messages with nested submessages and variable-length arrays.

Some ROS message types have nested submessages that are of different message types. Such nested ROS messages can be arrays whose length (number of elements) cannot be predetermined. Examples of such message types include:

- `geometry_msgs/PoseArray` : This message type contains an array of poses of type `geometry_msgs/Pose`. It is used to send a bunch of waypoints to the robot in a specific time step.
- `nav_msgs/Path` : This message type contains an array of poses of type `geometry_msgs/PoseStamped`. It is used for the output of motion planners that send a path for the robot to follow. The path is represented as a sequence of poses, each with its own header and timestamp.

In this example, you send pose arrays of different lengths over a single topic that publishes messages of type `geometry_msgs/PoseArray`. You then deploy the same as a ROS node.

Load and View Waypoints

Load the source data, which contains waypoints of different lengths that need to be published on a single topic, for the robot to follow. The MAT file `wayPointSets.mat` loads two sets of waypoints. These can be used to specify the pose array message. The waypoints are in the form of XYZ coordinates.

```
load wayPointSets.mat;
```

Visualize the two sets of waypoints using the `plot3` function. Note that the two sets contain different numbers of waypoints.

```
figure
plot3(wayPointSet1(:,1),wayPointSet1(:,2),wayPointSet1(:,3),"*-")
grid on
xlabel("X")
ylabel("Y")
zlabel("Z")
title("Waypoint Set 1")
```
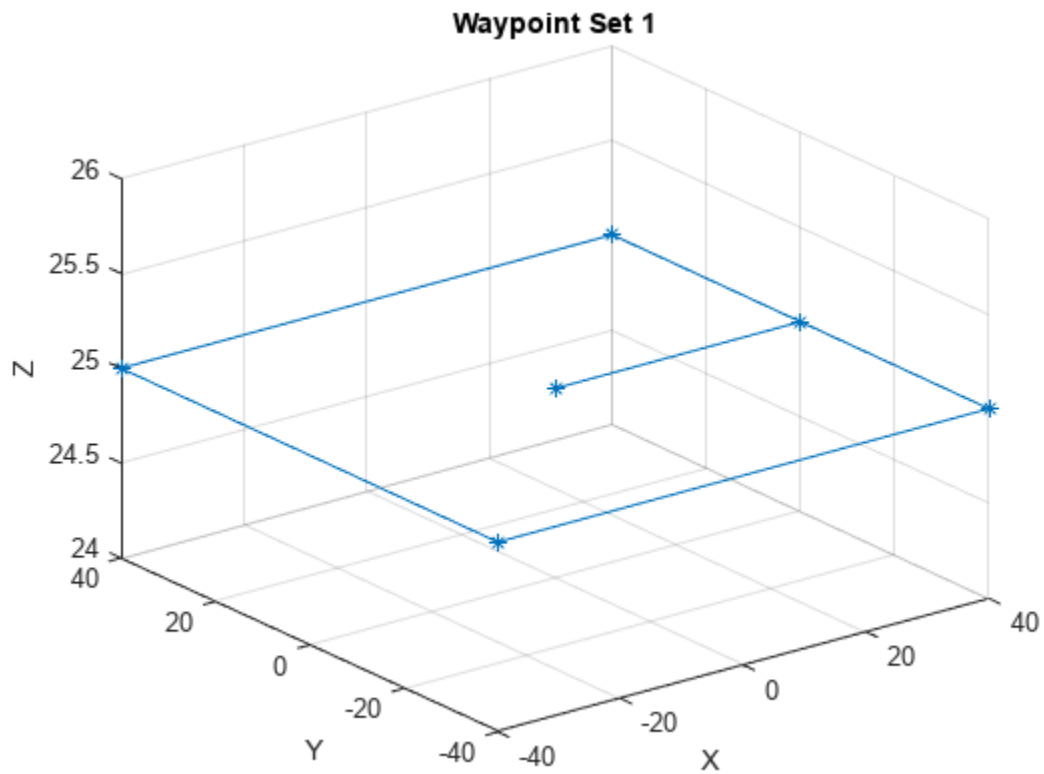
Waypoint Set 1

```
figure
plot3(wayPointSet2(:,1),wayPointSet2(:,2),wayPointSet2(:,3),"*-r")
grid on
xlabel("X")
ylabel("Y")
zlabel("Z")
title("Waypoint Set 2")
```

Waypoint Set 2
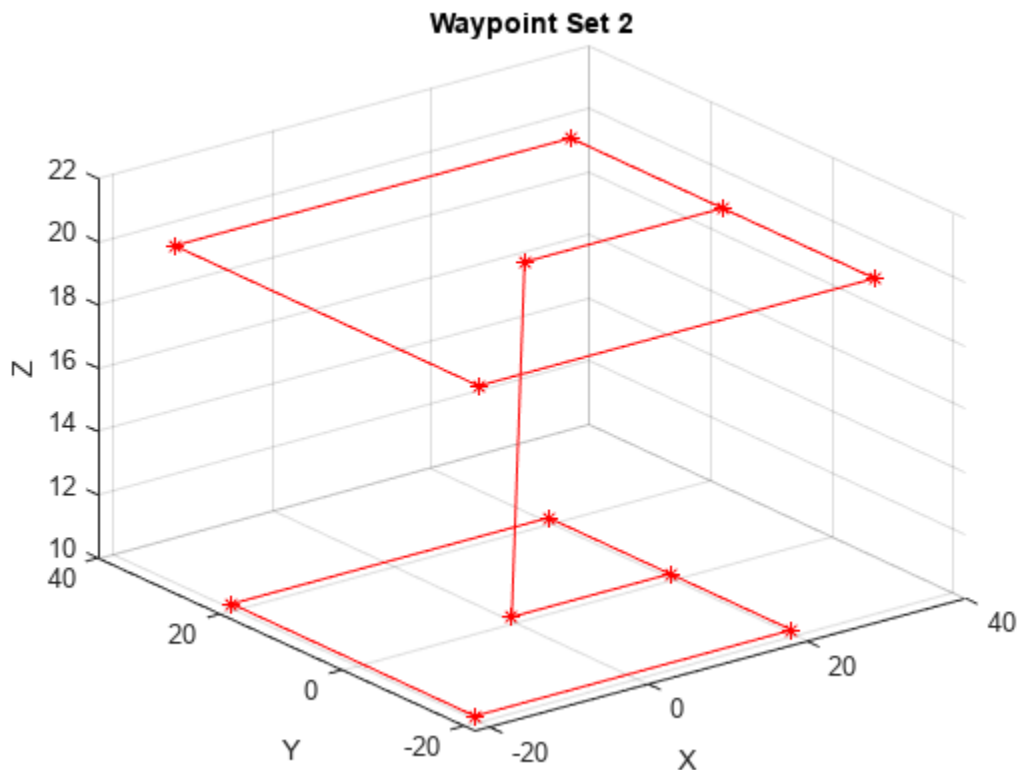
Initialize and Configure ROS Network

Use `rosinit` to create a ROS master in MATLAB and start a global node that is connected to the master.

```
rosinit
```

```
Launching ROS Core...
..Done in 2.1606 seconds.
Initializing ROS master on http://172.30.131.134:54967.
Initializing global node /matlab_global_node_17360 with NodeURI http://bat6234win64:64017/ and Ma
```

Use `rospublisher` to create a ROS publisher for sending messages of type `geometry_msgs/PoseArray`. Specify the name of the topic as `/waypoints`. Add a ROS subscriber that subscribes to the published topic using `rossubscriber`. Use messages in struct format for better efficiency.

```
pub = rospublisher("/waypoints","geometry_msgs/PoseArray","DataFormat","struct");
sub = rossubscriber("/waypoints","DataFormat","struct");
```

Use `rosmessage` to create an empty message based on the topic published by the publisher, `pub`.

```
poseArrayMsg = rosmessage(pub);
```

Populate Message and Publish

Specify the workspace variable corresponding to the waypoint set that you want to publish. Then, populate the pose array with `geometry_msgs/Pose` messages. Assign the XYZ position fields of the individual pose message elements from the waypoint set data. Continue adding new individual pose

message elements until the the pose array message contains all of the waypoint set data. Because struct messages are not handles, the same structure can be reused when populating the array of poses.

```matlab
% Specify the waypoint set to publish
wayPointsToPublish = wayPointSet1;

% Populate the pose array message
poseMsg = rosmessage("geometry_msgs/Pose","DataFormat","struct");
for k = 1:size(wayPointsToPublish,1)
    poseMsg.Position.X = wayPointsToPublish(k,1);
    poseMsg.Position.Y = wayPointsToPublish(k,2);
    poseMsg.Position.Z = wayPointsToPublish(k,3);
    poseArrayMsg.Poses(k) = poseMsg;
end
```

Use the `send` function to publish the pose array message to the topic `/waypoints`, using the ROS publisher object, `pub`.

```matlab
send(pub,poseArrayMsg);
pause(0.5)
```

View the pose array message data, as received by the subscriber, using the `LatestMessage` property of the `Subscriber` object. Use `horzcat` to concatenate the position information extracted from the received message into a structure array for the purposes of visualization. Use `plot3` to visualize the waypoints as received by the subscriber. Note that the visualization matches that of the corresponding source waypoint data set.

```matlab
receivedPoseArrayMsg1 = sub.LatestMessage;
waypointPositions1 = horzcat(receivedPoseArrayMsg1.Poses.Position);

figure
plot3([waypointPositions1.X],[waypointPositions1.Y],[waypointPositions1.Z],"*-")
grid on
xlabel("X")
ylabel("Y")
zlabel("Z")
title("Waypoint Set 1 Received Through ROS Topic")
```

Waypoint Set 1 Received Through ROS Topic
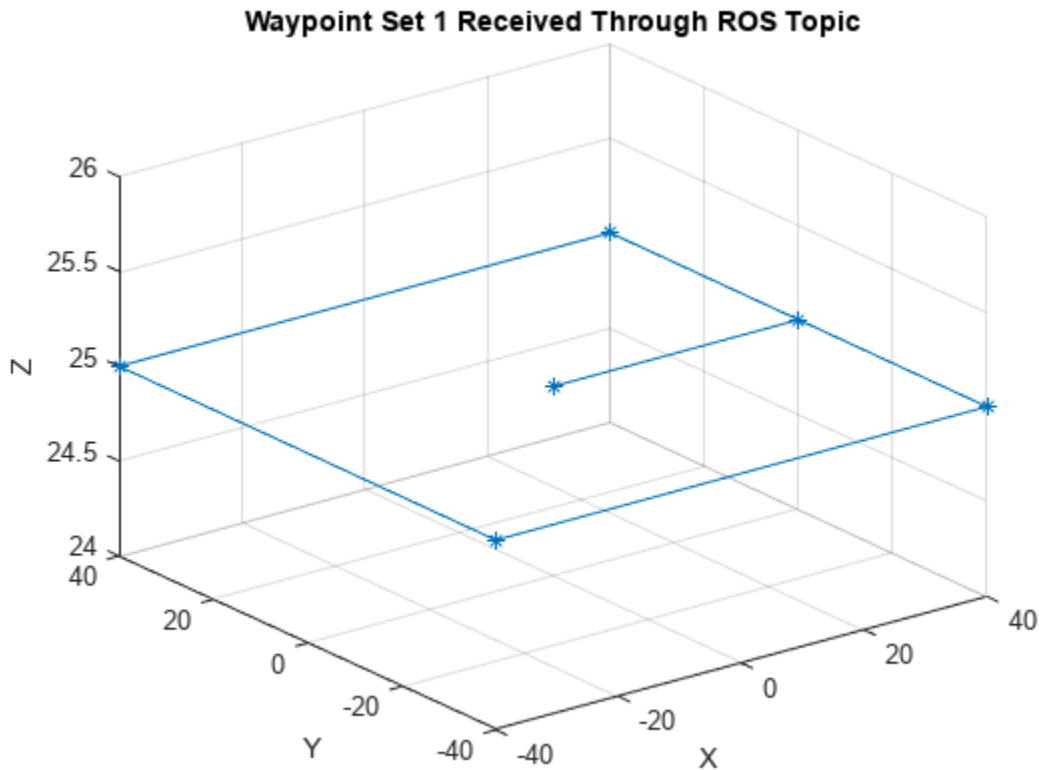
Now publish the second waypoint using the same procedure. Populate the pose array message with the new set of waypoint information.

```
% Specify the waypoint set to publish
wayPointsToPublish = wayPointSet2;

% Populate the Pose Array Message
poseMsg = rosmessage("geometry_msgs/Pose","DataFormat","struct");
for k = 1:size(wayPointsToPublish,1)
    poseMsg.Position.X = wayPointsToPublish(k,1);
    poseMsg.Position.Y = wayPointsToPublish(k,2);
    poseMsg.Position.Z = wayPointsToPublish(k,3);
    poseArrayMsg.Poses(k) = poseMsg;
end
```

Use the `send` function to publish the new pose array message to the same topic via the same ROS publisher object, `pub`.

```
send(pub,poseArrayMsg);
pause(0.5)
```

Visualize the pose array message data received by the subscriber by following the same procedure as before.

```
receivedPoseArrayMsg2 = sub.LatestMessage;
waypointPositions2 = vertcat(receivedPoseArrayMsg2.Poses.Position);
```

```
figure
plot3([waypointPositions2.X],[waypointPositions2.Y],[waypointPositions2.Z],"*-r")
grid on
xlabel("X")
ylabel("Y")
zlabel("Z")
title("Waypoint Set 2 Received Through ROS Topic")
```
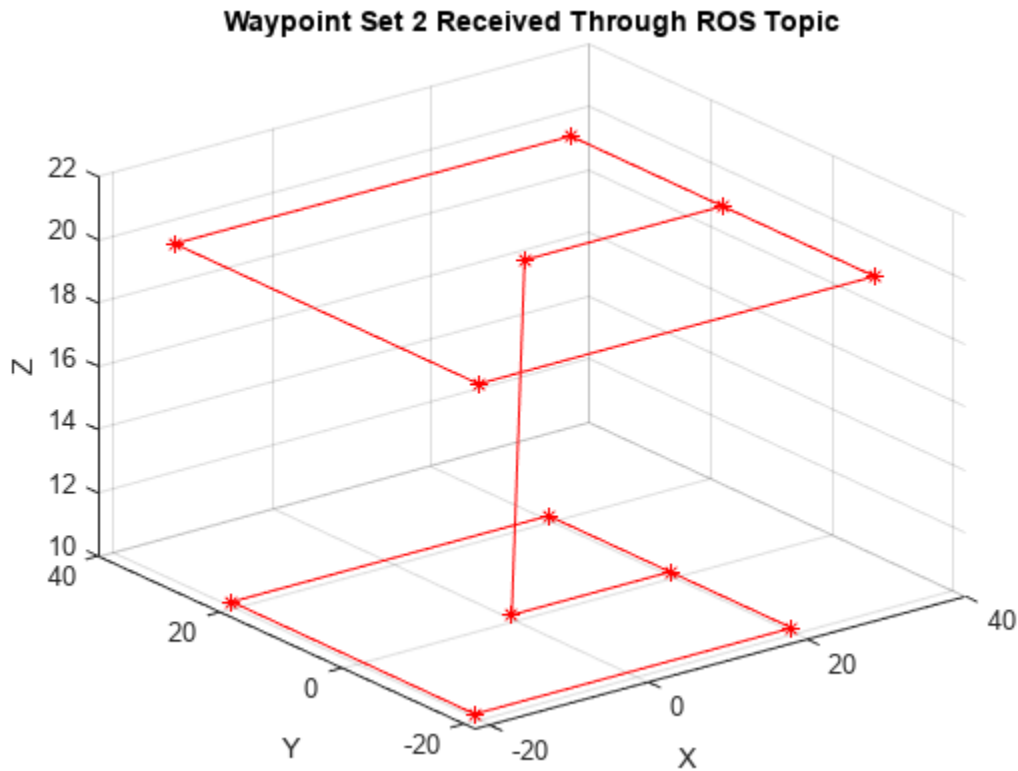

The visualization matches that of the corresponding source waypoint data set, indicating the successful broadcast of two sets of pose arrays with different lengths over a single topic.

Generate and Deploy ROS Node

You can now generate a ROS node that uses the same workflow described in the previous section to publish variable-length `geometry_msgs/PoseArray` messages. Then, you deploy the generated node on the local machine.

Configure MATLAB Coder

Create a MATLAB Coder configuration object that uses `"Robot Operating System (ROS)"` hardware. Set the build action to `"Build and Run"` and deploy to `"Localhost"` options so that the deployed ROS node starts running on the local machine after code generation.

```
cfg = coder.config("exe");
cfg.Hardware = coder.hardware("Robot Operating System (ROS)");
cfg.Hardware.DeployTo =  "Localhost";
cfg.Hardware.BuildAction = "Build and run";
```

Use the `helperPublishVariableLengthNestedROSMsg` function that contains the code verified in the previous section as the entry-point function.

Verify Initialization of Variable-Size Message Field before Deployment

Before ROS node generation, a necessary additional step is to initialize the variable-size data field in the ROS message to an array of a maximum size. This step is necessary to prevent segmentation faults as C++ requires preallocation of arrays. These lines of code perform the necessary steps.

```
% Create publisher
pub = rospublisher("/waypoints1","geometry_msgs/PoseArray","DataFormat","struct");
poseArrayMsg = rosmessage(pub);

% Specify maximum size of the pose array and initialize with empty pose messages
maxSize = 25;
poseArrayMsg.Poses = repmat(rosmessage('geometry_msgs/Pose',"DataFormat","struct"),maxSize,1);
```

Note that the variable-size field `Poses` is initialized to an array of 25 pose message structures. For any other message types, you must initialize its corresponding variable-size data field in a similar fashion.

Generate and deploy ROS Nodes.

```
codegen helperPublishVariableLengthNestedMsg -config cfg

---
Running ROS node.
Code generation successful.
```

Retrieve the latest message from the deployed node. Plot and visualize the received pose array message. You notice that an additional waypoint pose at the origin is present in the received pose array message. This is because the code initializes the position of unused pose elements to zero.

```
nodeSub = rossubscriber("/waypoints1","DataFormat","struct");
receivedPoseArrayMsg3 = receive(nodeSub,5);
waypointPositions3 = vertcat(receivedPoseArrayMsg3.Poses.Position);

figure
plot3([waypointPositions3.X],[waypointPositions3.Y],[waypointPositions3.Z],"*-r")
grid on
xlabel("X")
ylabel("Y")
zlabel("Z")
title("Waypoint Set Received Through Deployed ROS Node")
```

Waypoint Set Received Through Deployed ROS Node
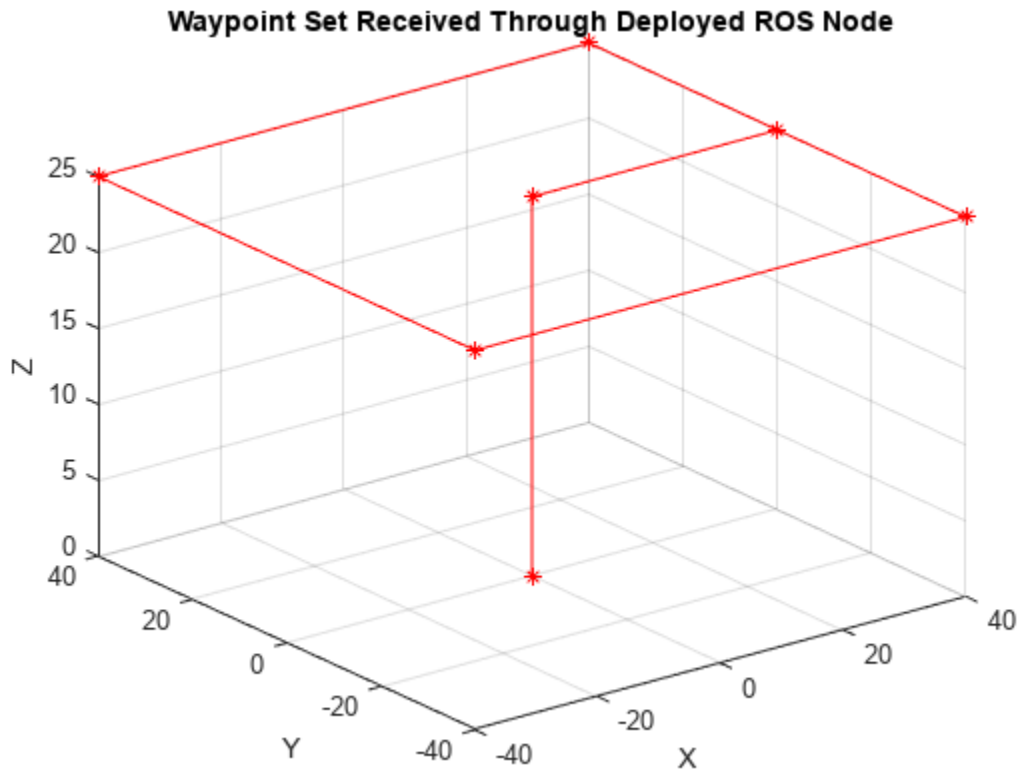
Use `rosshutdown` to shut down the ROS network in MATLAB. Doing so, shuts down the ROS master initialized by `rosinit` and deletes the global node.

`rosshutdown`

Shutting down global node /matlab_global_node_17360 with NodeURI http://bat6234win64:64017/ and Shutting down ROS master on http://172.30.131.134:54967.

If the waypoint set data has orientation information, you can populate it in the quaternion orientation fields of the individual pose message elements before publishing. To publish messages of type `nav_msgs/Path`, use the same procedure, but specify the individual pose message elements as `geometry_msgs/PoseStamped` type. To publish messages of any other type, specify the appropriate nested message type as individual array elements, and ensure that the source data set contains the required information you want to publish.

See Also

"Work with ROS Messages in Simulink" on page 1-134

"ROS Custom Message Support" on page 3-26

ROS Custom Message Support

In this section...

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 the location containing the custom message package folders 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 3-29.

If this is your first time using ROS custom messages, check "ROS Toolbox System Requirements".

Custom Message Contents

ROS custom messages are specified in ROS package folders that contains `msg`, `srv`, and `action` directories.

Note ROS Toolbox supports the existence of multiple custom messages folders on the MATLAB path at any given time. These folders can contain multiple custom message packages.

The `msg` folder contains all your custom message type definitions. You should also add all custom service type definitions to the `srv` folder and add all custom action type definitions to the `action` folder. For example, the package `custom_robot_msgs` has this folder and file structure.
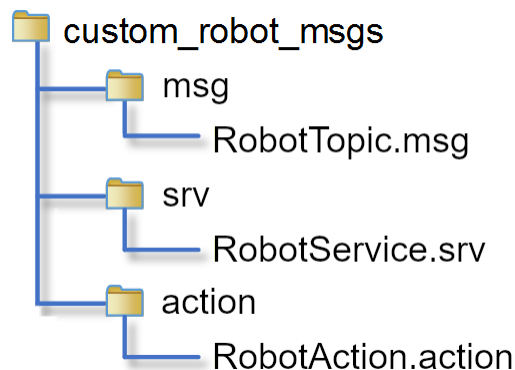
The package contains one custom message type in `RobotTopic.msg` and one custom service type in `RobotService.srv`, and one custom action type in `RobotAction.action`. MATLAB uses these

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`. For more information about ROS actions, see "ROS Actions Overview" on page 3-38.

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

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`, `.srv`, and `.action`) and converts each message type to working MATLAB code. The `rosgenmsg` function looks for `.msg` files in the `msg` folder, for `.srv` files in the `srv` folder, and for `.action` files in the `action` 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 3-29. 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.

Code Generation with Custom messages

Custom message, service, and action 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.

Replacing Definitions of Built-In Messages With Custom Definitions

MATLAB provides a lot of built-in ROS message types. You can replace the definitions of those message types with new definitions using the same custom message creation workflow detailed above. When you are replacing the definitions of a built-in message package, you must ensure that the custom message package folder contains new definitions (`.msg` files) for all the message types in the corresponding built-in message package.

See Also
`rosgenmsg` | `ros2genmsg`

Related Examples
*   "Create Custom Messages from ROS Package" on page 3-29
*   "Generate ROS 2 Custom Messages in MATLAB" on page 2-54
*   "ROS Toolbox 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`, `srv`, and `action` files. The correct file contents and folder structure are described in "Custom Message Contents" on page 3-26. 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 Toolbox 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 that contains the custom message packages and call the `rosgenmsg` function to create custom messages for MATLAB.

    rosgenmsg('c:\MATLAB\custom_msgs')

• 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_msg_gen_ros1\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

rosgenmsg | ros2genmsg

Related Examples

- "Generate ROS Custom Messages in MATLAB"
- "Generate ROS 2 Custom Messages in MATLAB"
- "ROS Toolbox System Requirements"

Generate ROS Custom Messages in MATLAB

Use custom messages to extend the set of message types currently supported in ROS. Custom messages are messages that you define. If you are sending and receiving supported message types, you do not need to use custom messages. To see the list of supported message types, enter `rosmsg list` in the MATLAB Command Window. For more information about supported ROS messages, see "Work with Basic ROS Messages" on page 1-52.

If this is your first time working with ROS custom messages, see "ROS Toolbox System Requirements".

ROS custom messages are specified in ROS package folders that contain a folder named `msg`. The `msg` folder contains all your custom message type definitions. For example, the `simple_msgs` package in the `rosCustomMessages` folder, has this folder and file structure.
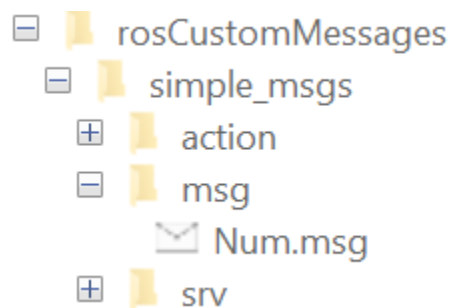
The package contains the custom message type `Num.msg`. MATLAB uses these files to generate the necessary files for using the custom messages contained in the package.

In this example, you create ROS custom messages in MATLAB and compress them in a shareable ZIP archive. You must have a ROS package that contains the required `msg` file.

After you prepare your custom message package folder, you specify the path to the parent folder and call `rosgenmsg` with the specified path.

Open a new MATLAB session and create a custom message package folder in a local folder. Choose a short folder path when you generate custom messages on a Windows machine to avoid limitations on the number of characters in the folder path. For example,

```
genDir = fullfile('C:/test/rosCustomMessages')
```

```
genDir = fullfile(pwd,'rosCustomMessages');
packagePath = fullfile(genDir,'simple_msgs');
mkdir(packagePath)
```

Create a folder named `msg` inside the custom message package folder.

```
mkdir(packagePath,'msg')
```

Create a file named `.msg` inside the `msg` folder.

```
messageDefinition = {'int64 num'};
```

```
fileID = fopen(fullfile(packagePath,'msg', ...
```

```
                  'Num.msg'),'w');
fprintf(fileID,'%s\n',messageDefinition{:});
fclose(fileID);
```

Create a folder named `srv` inside the custom message package folder.

```
mkdir(packagePath,'srv')
```

Create a file named `.srv` inside the `srv` folder.

```
serviceDefinition = {'int64 a'
                     'int64 b'
                     '---'
                     'int64 sum'};

fileID = fopen(fullfile(packagePath,'srv', ...
               'AddTwoInts.srv'),'w');
fprintf(fileID,'%s\n',serviceDefinition{:});
fclose(fileID);
```

Create a folder named `action` inside the custom message package folder.

```
mkdir(packagePath,'action')
```

Create a file named `.action` inside the `action` folder.

```
actionDefinition = {'int64 goal'
                    '---'
                    'int64 result'
                    '---'
                    'int64 feedback'};

fileID = fopen(fullfile(packagePath,'action', ...
               'Test.action'),'w');
fprintf(fileID,'%s\n',actionDefinition{:});
fclose(fileID);
```

Generate custom messages from ROS definitions in `.msg`, `.srv`, and `.action` files. Use the `CreateShareableFile` name-value argument to create a shareable ZIP archive of the generated custom messages.

For information about how to use use this ZIP archive to register the custom messages in another machine, see `rosRegisterMessages`.

```
rosgenmsg(genDir,CreateShareableFile=true)
```

```
Identifying message files in folder 'C:/test/rosCustomMessages'.Done.
Creating a Python virtual environment.Done.
Adding required Python packages to virtual environment.Done.
Copying include folders.Done.
Copying libraries.Done.
Validating message files in folder 'C:/test/rosCustomMessages'.Done.
[1/1] Generating MATLAB interfaces for custom message packages... Done.
Running catkin build in folder 'C:/test/rosCustomMessages/matlab_msg_gen_ros1/win64'.
Build in progress. This may take several minutes...
Build succeeded.build log
Generating zip file in the folder 'C:/test/rosCustomMessages'.Done.
```

To use the custom messages, follow these steps:

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

```
addpath('C:\test\rosCustomMessages\matlab_msg_gen_ros1\win64\install\m')
savepath
```

2. Refresh all message class definitions, which requires clearing the workspace, by executing:

```
clear classes
rehash toolboxcache
```

3. Verify that you can use the custom messages.
   Enter "rosmsg list" and ensure that the output contains the generated
   custom message types.

- ⊟ 📁 rosCustomMessages
  - ⊞ 📁 matlab_msg_gen_ros1
  - ⊞ 📁 simple_msgs
  - ⊞ 🗜 matlab_msg_gen_ros1.zip
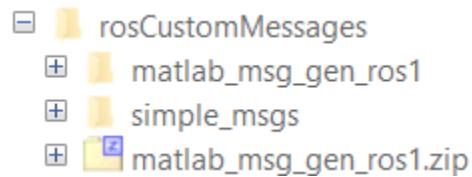
Verify creation of the new custom messages by entering `rosmsg list`.

```
simple_msgs/AddTwoIntsRequest
simple_msgs/AddTwoIntsResponse
simple_msgs/Num
simple_msgs/TestAction
simple_msgs/TestActionFeedback
simple_msgs/TestActionGoal
simple_msgs/TestActionResult
simple_msgs/TestFeedback
simple_msgs/TestGoal
simple_msgs/TestResult
```

Register ROS Custom Messages to MATLAB

In this example, you create ROS custom messages in MATLAB® to share them on another machine. This other machine must run on the same platform and use the same version of MATLAB®. You must have a ROS package that contains the required msg file, as this figure shows.

```
⊟  📁  rosCustomMessages
   ⊟  📁  simple_msgs
      ⊞  📁  action
      ⊟  📁  msg
            ✉  Num.msg
      ⊞  📁  srv
```
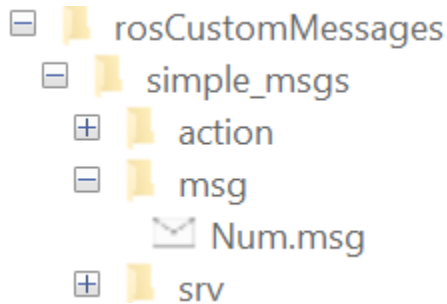
Open a new MATLAB session and create a custom message package folder in a local folder. Specify a short folder path when you generate custom messages on a Windows machine to avoid limitations on the number of characters in the folder path. For example,

```matlab
genDir = fullfile('C:/Work/rosCustomMessages')

genDir = fullfile(pwd,'rosCustomMessages');
packagePath = fullfile(genDir,'simple_msgs');
mkdir(packagePath)
```

Create a folder named msg inside the custom message package folder.

```matlab
mkdir(packagePath,'msg')
```

Create a file named .msg inside the msg folder.

```matlab
messageDefinition = {'int64 num'};

fileID = fopen(fullfile(packagePath,'msg', ...
            'Num.msg'),'w');
fprintf(fileID,'%s\n',messageDefinition{:});
fclose(fileID);
```

Create a folder named srv inside the custom message package folder.

```matlab
mkdir(packagePath,'srv')
```

Create a file named .srv inside the srv folder.

```matlab
serviceDefinition = {'int64 a'
                     'int64 b'
                     '---'
                     'int64 sum'};

fileID = fopen(fullfile(packagePath,'srv', ...
            'AddTwoInts.srv'),'w');
fprintf(fileID,'%s\n',serviceDefinition{:});
fclose(fileID);
```

Create a folder named `action` inside the custom message package folder.

```
mkdir(packagePath,'action')
```

Create a file named `.action` inside the `action` folder.

```
actionDefinition = {'int64 goal'
                    '---'
                    'int64 result'
                    '---'
                    'int64 feedback'};

fileID = fopen(fullfile(packagePath,'action', ...
            'Test.action'),'w');
fprintf(fileID,'%s\n',actionDefinition{:});
fclose(fileID);
```

Generate the custom messages from the ROS definitions in `.msg`, `.srv`, and `.action` files as a shareable ZIP archive.

```
rosgenmsg(genDir,CreateShareableFile=true)

Identifying message files in folder 'C:/Work/rosCustomMessages'.Done.
Validating message files in folder 'C:/Work/rosCustomMessages'.Done.
[1/1] Generating MATLAB interfaces for custom message packages... Done.
Running catkin build in folder 'C:/Work/rosCustomMessages/matlab_msg_gen_ros1/win64'.
Build in progress. This may take several minutes...
Build succeeded.build log
Generating zip file in the folder 'C:/Work/rosCustomMessages'.Done.

To use the custom messages, follow these steps:

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

addpath('C:\Work\rosCustomMessages\matlab_msg_gen_ros1\win64\install\m')
savepath

2. Refresh all message class definitions, which requires clearing the workspace, by executing:

clear classes
rehash toolboxcache

3. Verify that you can use the custom messages.
   Enter "rosmsg list" and ensure that the output contains the generated
   custom message types.
```

- rosCustomMessages
  - matlab_msg_gen_ros1
  - simple_msgs
  - matlab_msg_gen_ros1.zip

Copy the generated custom messages in the ZIP archive to the target computer and register it using the `rosRegisterMessages` function.

```
rosRegisterMessages(genDir)
```

```
To use the custom messages, follow these steps:

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

addpath('C:\Work\rosCustomMessages\install\m')
savepath

2. Refresh all message class definitions, which requires clearing the workspace, by executing:

clear classes
rehash toolboxcache

3. Verify that you can use the custom messages.
   Enter "rosmsg list" and ensure that the output contains the generated
   custom message types.
```

- rosCustomMessages
  - install
  - simple_msgs
  - matlab_msg_gen_ros1.zip
  - VersionInfo.mat
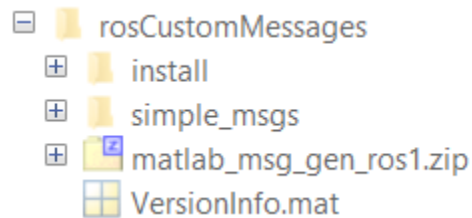
Run `rosmsg list` on the target computer to view the custom messages for using in the workflow.

```
simple_msgs/AddTwoIntsRequest
simple_msgs/AddTwoIntsResponse
simple_msgs/Num
simple_msgs/TestAction
simple_msgs/TestActionFeedback
simple_msgs/TestActionGoal
simple_msgs/TestActionResult
simple_msgs/TestFeedback
simple_msgs/TestGoal
simple_msgs/TestResult
```

ROS Actions Overview

In this section...

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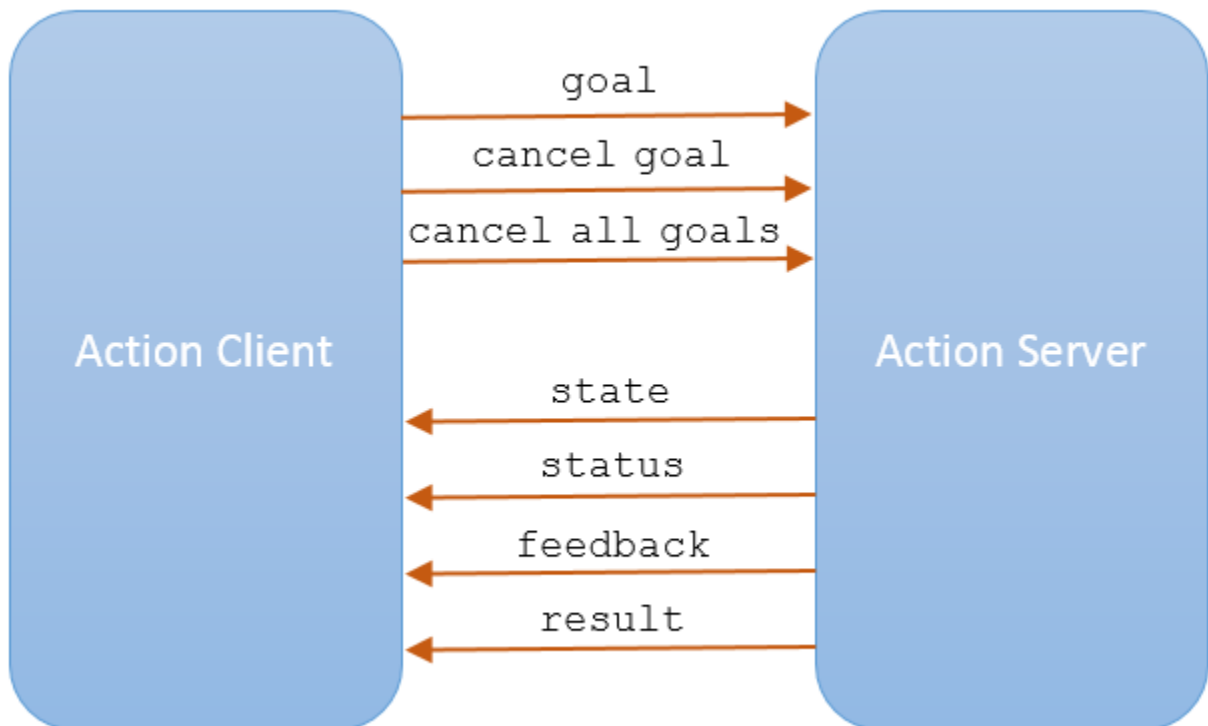
Use the `sendGoal` function to send goals to the server. Send the goal and wait for it to complete using `sendGoalAndWait`. This function enables you to return the result message, final state of the goal and status of the server. While the server is executing a goal, the callback function, `FeedbackFcn`,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

Use the `sendFeedback` function to send feedback messages to the client during goal execution. During goal execution, the server calls the `ExecuteGoalFcn` callback. You can use `isPreemptRequested` within the callback function to check whether the client has cancelled the goal or sent a new goal to execute.

Performing Actions Workflow

In general, the following steps occur when creating and executing a ROS action on a ROS network.
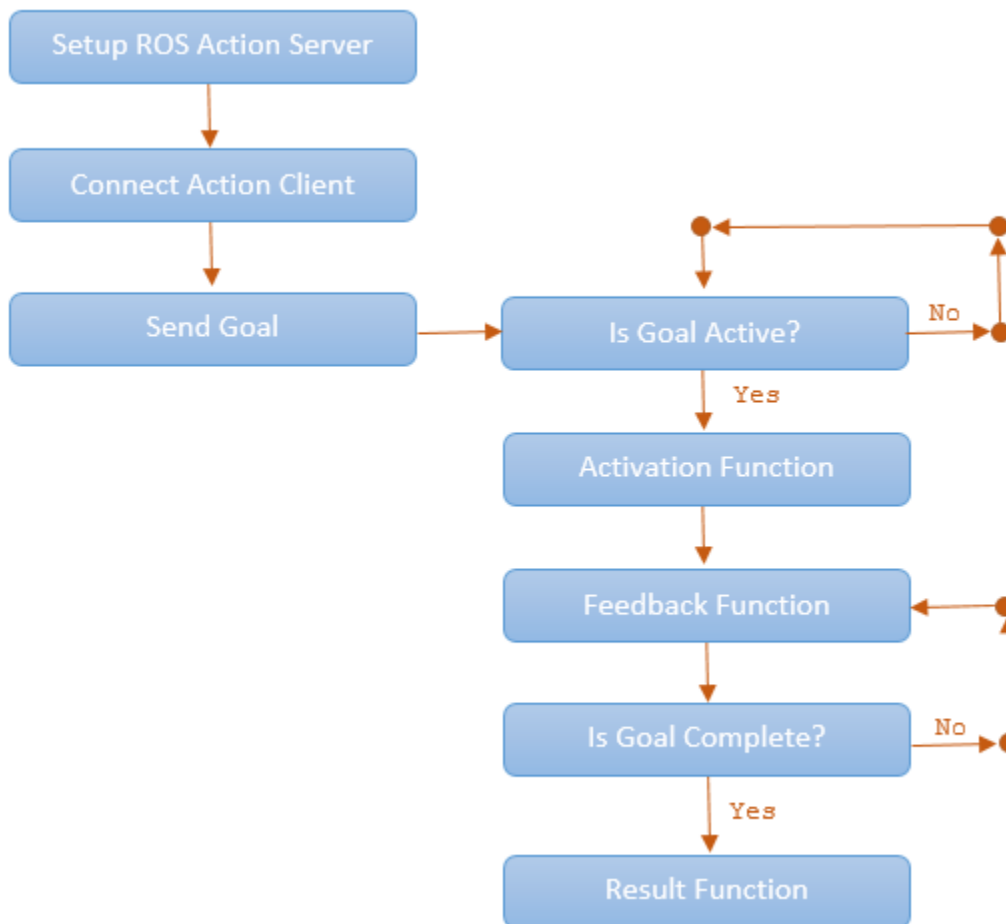
- To set up a ROS action server, use `rosactionserver`. Check what actions are available on a ROS network by typing `rosaction list` in the MATLAB command window.

- 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')`

- `FeedbackMsg` — The feedback message contains information sent to the client by the server during goal execution. The type of feedback message depends on the action type. This feedback contains the current state and any other information related to the progress towards goal execution. Use `sendFeedback` function to send the feedback message from the server.

- `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.

- `ResultMsg` — The server sends the final result message to the client after goal execution as the output argument to `sendGoal` or `sendGoalAndWait`.

- `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

rosactionserver | rosactionclient | rosaction

Related Examples

- "Move a Turtlebot Robot Using ROS Actions" on page 3-42

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 Simulated TurtleBot" on page 1-206 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
```
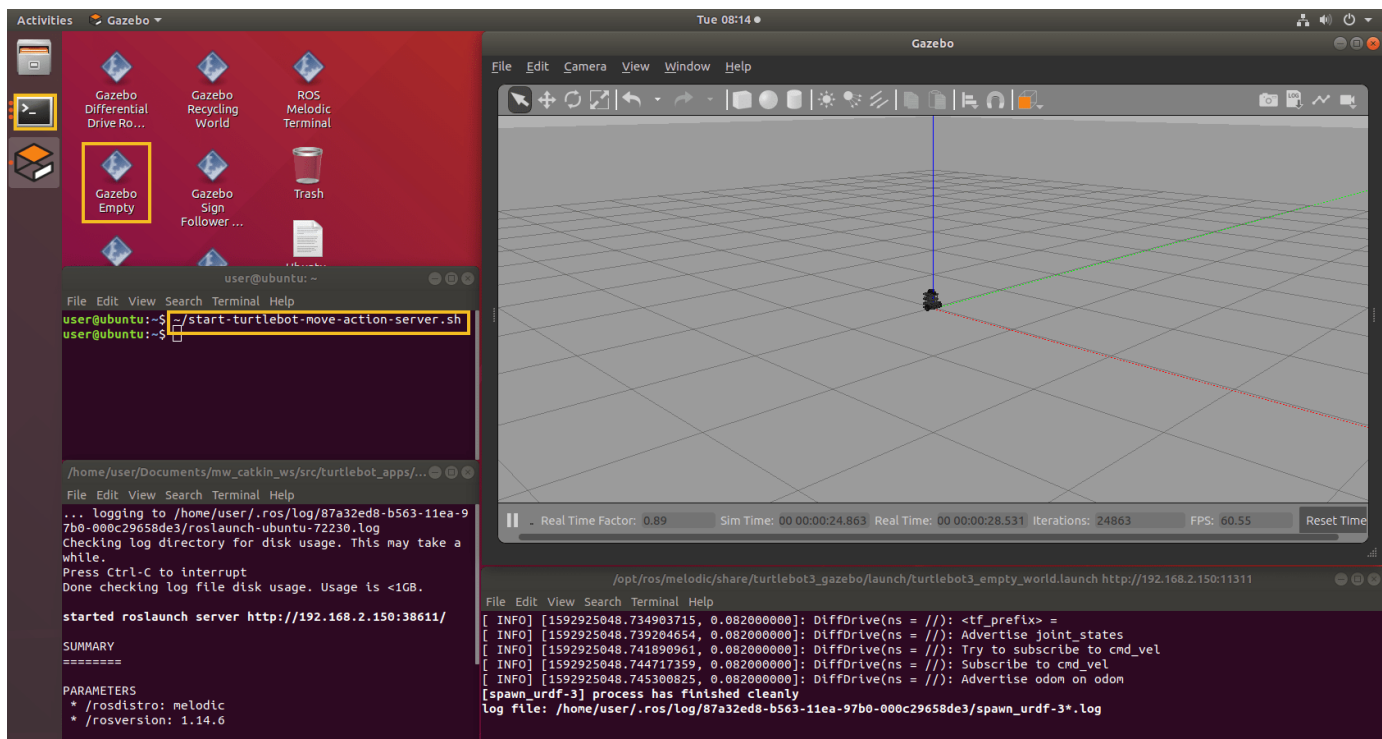

Connect to a ROS network. You must have an ROS action server setup on this network. Change `ipaddress` to the address of your ROS network.

```
ipaddress = "192.168.178.133";
rosinit(ipaddress,11311);
```

Initializing global node /matlab_global_node_88888 with NodeURI http://192.168.178.1:57929/

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. Use struct message format for better efficiency.

```
[client,goalMsg] = rosactionclient("/turtlebot_move","turtlebot_actions/TurtlebotMove","DataForma
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 = single(2);
goalMsg.TurnDistance = single(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 = struct with fields:
        MessageType: 'turtlebot_actions/TurtlebotMoveResult'
       TurnDistance: 0
    ForwardDistance: 2.0078
```

Disconnect from the ROS network.

```
rosshutdown
```

```
Shutting down global node /matlab_global_node_88888 with NodeURI http://192.168.178.1:57929/
```

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 2.2148 seconds.
Initializing ROS master on http://172.30.131.134:59451.
Initializing global node /matlab_global_node_16207 with NodeURI http://bat6234win64:57301/ and M
```

Setup publisher and message to send `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: 0.0772
          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: [0.1091 0.1042 0.0901 0.1058 0.1050 0.0909 0.1050 0.0911 0.1058 0.1049 0.0
          NumPeriods: 100
       AveragePeriod: 0.1001
   StandardDeviation: 0.0074
         NumOverruns: 0
```

Shut down ROS network.

```
rosshutdown
```

```
Shutting down global node /matlab_global_node_16207 with NodeURI http://bat6234win64:57301/ and N
Shutting down ROS master on http://172.30.131.134:59451.
```

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)
```
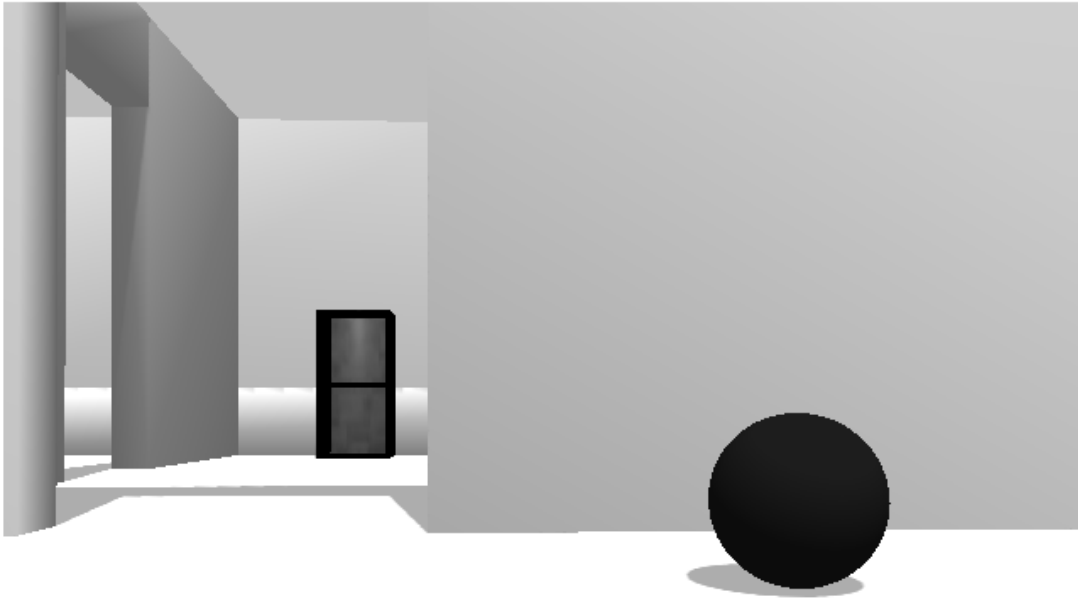
```
writeImage(msgGray,grayImg)
send(pub,msgGray)
```

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

Configure MATLAB Coder for ROS Node Generation

In this section...
"Create MATLAB Coder Configuration Object" on page 3-48
"Configure Hardware Properties" on page 3-48
"Configure Build Options" on page 3-49
"Specify Additional Hardware Properties and ROS Dependencies" on page 3-51

To generate C++ code for ROS Nodes from MATLAB functions, you must configure a MATLAB Coder configuration object. This topic shows you how to configure the properties of the object to customize ROS Node generation.

To create a MATLAB Coder configuration object, use the `coder.config` object. ROS Toolbox only supports the executable build type for the MATLAB Coder configuration object.

Create MATLAB Coder Configuration Object

To create a MATLAB Coder configuration object, use the `coder.config` object. ROS Toolbox only supports the executable build type for the MATLAB Coder configuration object.

```
cfg = coder.config('exe');
```

Configure Hardware Properties

Specify the `Hardware` property of the object as a `'Robot Operating System (ROS)'` hardware configuration object, using the `coder.hardware` function.

```
cfg.Hardware = coder.hardware('Robot Operating System (ROS)');
```

Then, you can specify these `coder.Hardware` properties specific to ROS node generation. For remote device deployment, the device parameters automatically save to MATLAB preferences, and are used the next time you set the deployment site to `'Remote Device'`.

Property	Values	Description
cfg.Hardware.DeployTo	'Localhost' (default) 'Remote Device'	Deployment site for the code, which can be a remote device or the local host device.
cfg.Hardware.BuildAction	'None' (default) 'Build and Load' 'Build and Run'	Build action for code generation.
cfg.Hardware.RemoteDeviceAddress	'192.168.128.130' (default) character vector	Remote IP address or host name of the remote device.
cfg.Hardware.RemoteDeviceUsername	'user' (default) character vector	User name for the remote device.

Property	Values	Description
`cfg.Hardware.RemoteDevicePassword`	`'password'` (default) character vector	Password for the remote device.
`cfg.Hardware.CatkinWorkspace`	`'~/catkin_ws'` (default) character vector	Path to the Catkin workspace in the remote device. For Windows devices, the default value replaces `'~'` with the user path.
`cfg.Hardware.ROSFolder`	`'/opt/ros/noetic'` (default) character vector	Path to the ROS installation folder. Leave it blank to use the MATLAB ROS distribution.
`cfg.Hardware.PackageMaintainerName`	`'ROS User'` (default) character vector	ROS package maintainer name, which is used for `package.xml` generation.
`cfg.Hardware.PackageMaintainerEmail`	`'rosuser@test.com'` (default) character vector	ROS package maintainer e-mail ID, used for `package.xml` generation.
`cfg.Hardware.PackageLicense`	`'BSD'` (default) character vector	ROS license information, used for `package.xml` generation.
`cfg.Hardware.PackageVersion`	`'1.0.0'` (default) character vector	Version number of the ROS package.

Configure Build Options

You can also specify whether you only need the code to be generated using the `GenCodeOnly` property of the coder configuration object. This table highlights how the code generation behavior changes based on `GenCodeOnly` and other hardware properties.

`cfg.GenCodeOnly`	`cfg.Hardware.DeployTo`	`cfg.Hardware.BuildAction`	Code Generation and Build Behavior
`true`	`'Localhost'`	`'None'`	Generates ROS Package source code folder `src` on local host device
		`'Build and Load'`	
		`'Build and Run'`	
	`'Remote Device'`	`'None'`	Generates these artifacts on local host device: • ROS package source code folder `src` • TAR archive of the source code in a `.tgz` file • Shell script to extract and set up
		`'Build and Load'`	

3-49

cfg.GenCodeOnly	cfg.Hardware.DeployTo	cfg.Hardware.BuildAction	Code Generation and Build Behavior
		'Build and Run'	the package on remote device
false	'Localhost'	'None'	Generates ROS Package source code folder src on local host device
		'Build and Load'	Generates ROS Package source code folder src and builds the executable on local host device
		'Build and Run'	Generates ROS Package source code folder src, builds and runs the executable on local host device
	'Remote Device'	'None'	Generates these artifacts on local host device: • ROS package source code folder src • TAR archive of the source code in a .tgz file • Shell script to extract and set up the package on remote device
		'Build and Load'	Generates these artifacts on local host device and builds the executable on remote device: • ROS package source code folder src • TAR archive of the source code in a .tgz file • Shell script to extract and set up the package on remote device

cfg.GenCodeOnly	cfg.Hardware.DeployTo	cfg.Hardware.BuildAction	Code Generation and Build Behavior
		'Build and Run'	Generates these artifacts on local host device, builds and runs the executable on remote device: • ROS package source code folder src • TAR archive of the source code in a .tgz file • Shell script to extract and set up the package on remote device

For example, the following code specifies that the generated code must be deployed to the remote device and run after build. It also specifies the remote device parameters.

```
cfg.Hardware.DeployTo = 'Remote Device';
cfg.Hardware.BuildAction = 'Build and run';
cfg.Hardware.RemoteDeviceAddress = '192.168.243.144';
cfg.Hardware.RemoteDeviceUsername = 'user';
cfg.Hardware.RemoteDevicePassword = 'password';
```

For more information on ROS Node generation, see "MATLAB Programming for Code Generation" on page 1-158.

Specify Additional Hardware Properties and ROS Dependencies

To specify additional hardware-specific configuration parameters use coder.HardwareImplementation (MATLAB Coder) object. For example, this code specifies the manufacturer and type of the hardware as well as long long data type support for int64 and uint64 values.

```
cfg.HardwareImplementation.ProdHWDeviceType = 'Intel->x86-64 (Linux 64)';
cfg.HardwareImplementation.ProdLongLongMode = true;
```

To specify external ROS packages as dependencies for the generated ROS node, specify appropriate custom toolchain options in the coder configuration object.

```
cfg.BuildConfiguration = 'Specify';
cfg.CustomToolchainOptions{4} = '"ROS_PKG1","ROS_PKG2"'; % Add ROS Required Packages
cfg.CustomToolchainOptions{6} = '/usr/include/opencv';   % Add Include Directories
cfg.CustomToolchainOptions{8} = '"-lopencv_core","-lopencv_shape"'; % Add Link Libraries
cfg.CustomToolchainOptions{10} = '/usr/lib/opencv';      % Add Library Paths
cfg.CustomToolchainOptions{12} = '-DMYVAR=1';  % Add Defines
```

See Also

Related Examples

- "MATLAB Programming for Code Generation" on page 1-158
- "Generate a Standalone ROS Node from MATLAB" on page 1-164

Get Started with ROS Bag Viewer App

The **ROS Bag Viewer** app enables you to load a ROS bag file into MATLAB and visualize all the message content in the bag file. You can create separate viewers for different message types and visualize them during playback. You can also control the playback of the bag file.

For more information on how to launch the app, see **ROS Bag Viewer**.

This example walks you through the ROS bag file loading and playback workflow by using two steps.

1 "Load and Play ROS Bag File" on page 3-54
2 "Control ROS Bag File Playback" on page 3-59
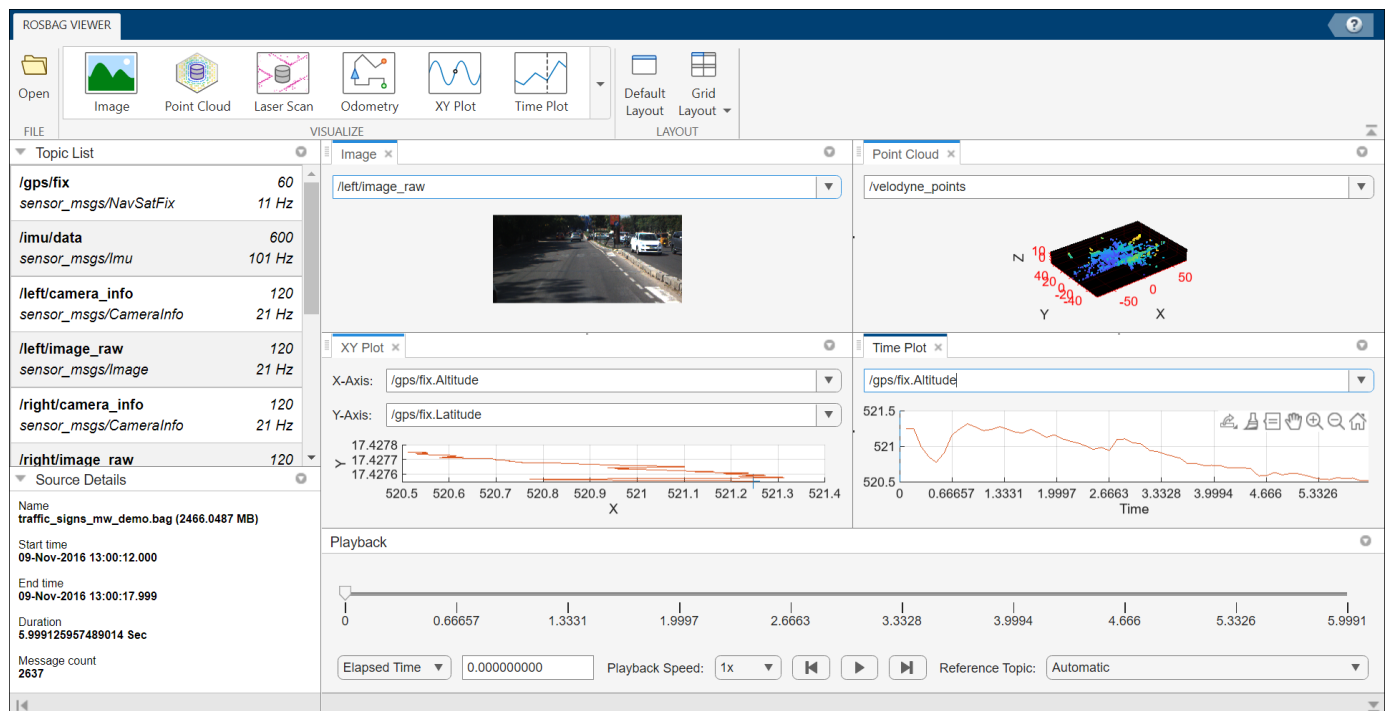
See Also
ROS Bag Viewer

Load and Play ROS Bag File

Load ROS Bag File

Load the ROS bag file by opening it from the **File** tab in ROS Bag Viewer app toolstrip. The toolstrip also contains the list of supported visualizers under the **Visualize** tab, and allows you to choose the layout between **Default** and **Grid**.

You can open multiple instances of the app to play multiple ROS bag files simultaneously.

View Bag File Details

The details of bag file content are found on the left side under the app toolstrip.

- The **Topic List** shows the available topics in the ROS bag file.
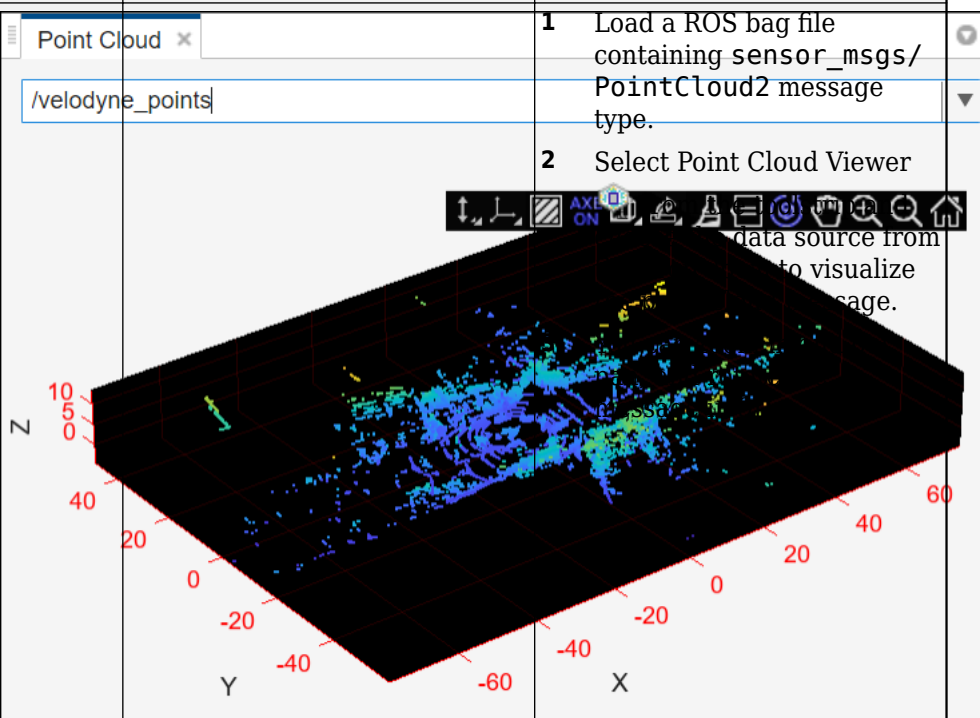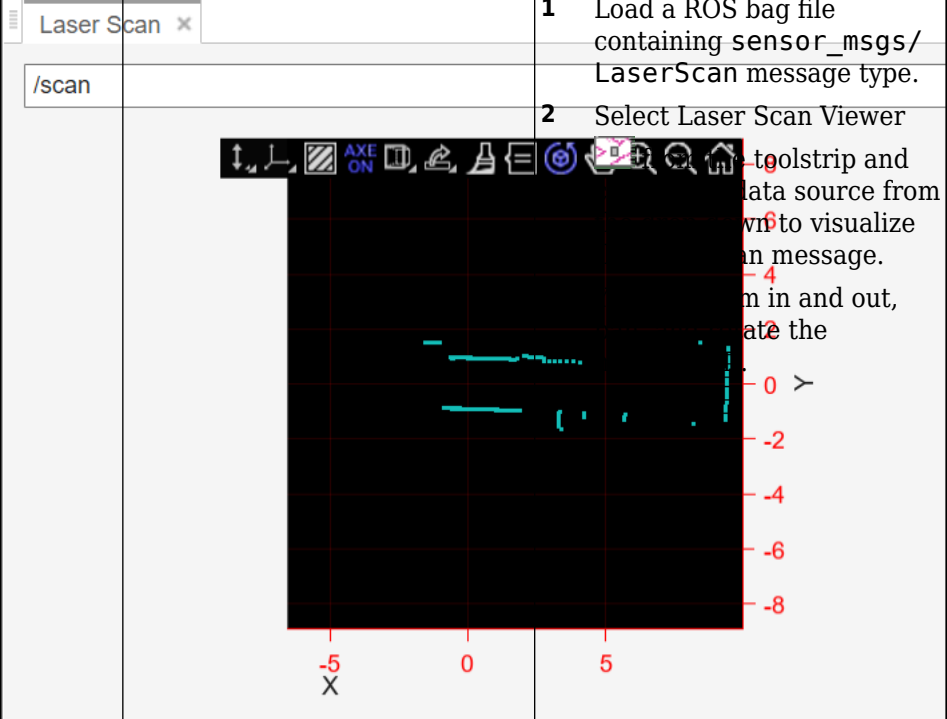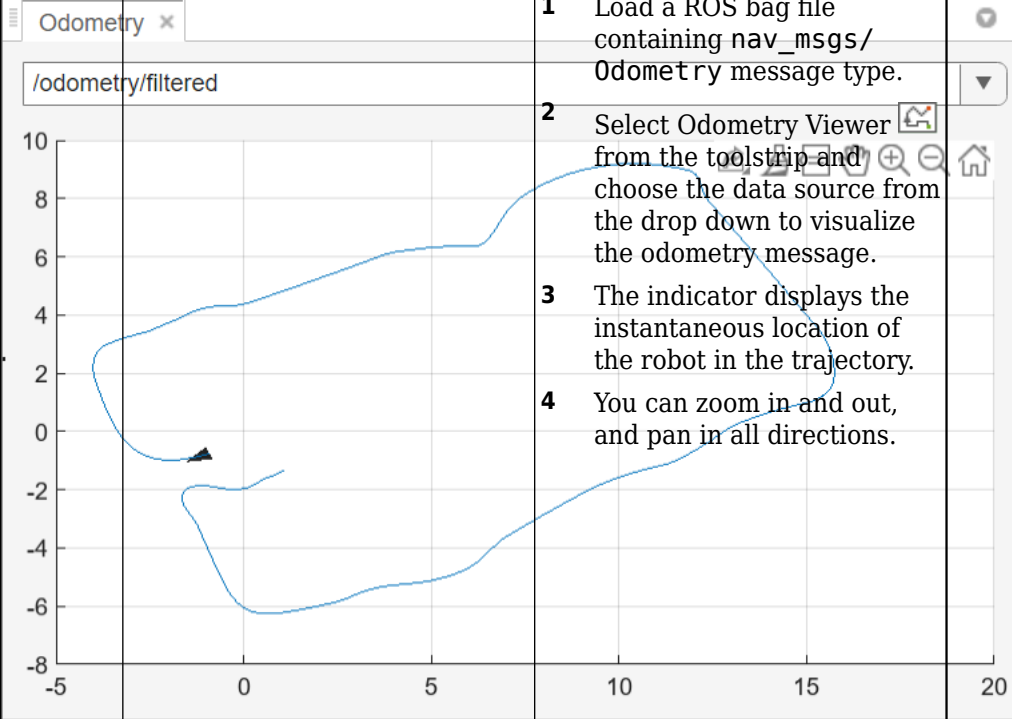
- The **Source Details** list the **Name**, **Start time**, **End time**, **Duration** and **Message count** of the bag file.
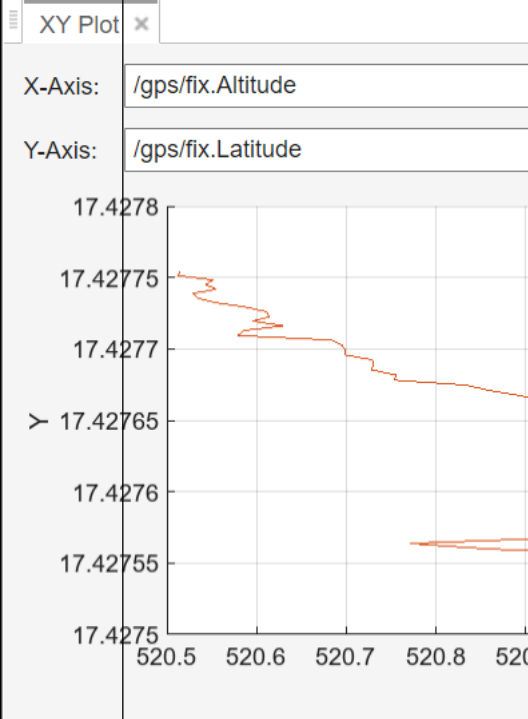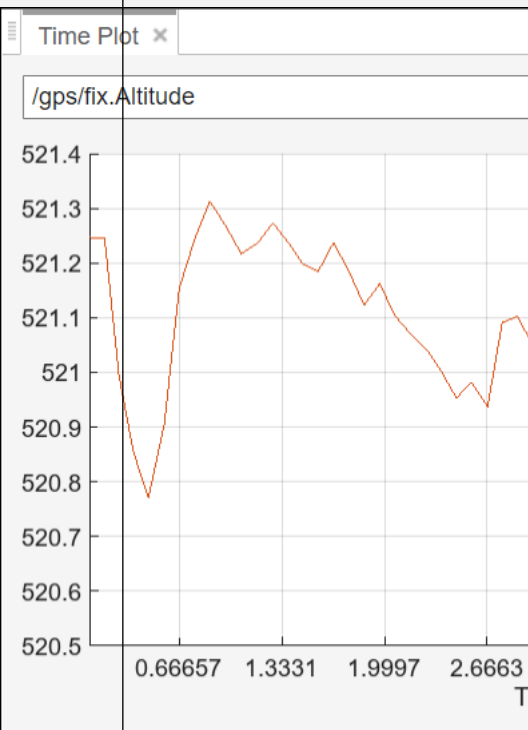
Create Visualizers and Play Bag File

Types of Viewer

Viewer	Interface	Description
Image Viewer		**1** Load a ROS bag file containing `sensor_msgs/Image` or `sensor_msgs/CompressedImage` message type. **2** Select Image Viewer from the toolstrip and choose the data source from the drop down to visualize the image. **3** You can zoom in and out, and pan the image in all directions.
Point Cloud Viewer		**1** Load a ROS bag file containing `sensor_msgs/PointCloud2` message type. **2** Select Point Cloud Viewer ... data source from ... to visualize ... sage.

Viewer	Interface	Description
Laser Scan Viewer		**1** Load a ROS bag file containing `sensor_msgs/ LaserScan` message type. **2** Select Laser Scan Viewer from the toolstrip and choose the data source from the drop down to visualize the scan message. You can zoom in and out, and pan in and rotate the view.
Odometry Viewer		**1** Load a ROS bag file containing `nav_msgs/ Odometry` message type. **2** Select Odometry Viewer from the toolstrip and choose the data source from the drop down to visualize the odometry message. **3** The indicator displays the instantaneous location of the robot in the trajectory. **4** You can zoom in and out, and pan in all directions.

Viewer	Interface	Description
XY Plot Viewer		**1** Load a ROS bag file containing `geometry_msgs/Point` or `nav_msgs/Odometry` message type. **2** Select XY Plot Viewer from the toolstrip and choose the data source from the drop down to visualize how the numeric message field changes across the XY axes. **3** The indicator displays the instantaneous location of the robot across the XY axes. **4** You can zoom in and out, and pan in all directions.
Time Plot Viewer		**1** Load a ROS bag file containing `geometry_msgs/Point` or `nav_msgs/Odometry` message type. **2** Select Time Plot Viewer from the toolstrip and choose the data source from the drop down to visualize how the numeric message field changes with respect to time. **3** The indicator displays the instantaneous location of the robot. **4** You can zoom in and out, and pan in all directions.

Viewer	Interface	Description
Message Viewer	Message ✕ /gps/fix MessageType : sensor_msgs/NavSatFix Header MessageType : std_msgs/Header Seq : 18060 Stamp Sec : 1478676612 Nsec : 81456019 FrameId : gps Status MessageType : sensor_msgs/NavSatStatus STATUSNOFIX : -1 STATUSFIX : 0 STATUSSBASFIX : 1 STATUSGBASFIX : 2 Status : 0 SERVICEGPS : 1 SERVICEGLONASS : 2 SERVICECOMPASS : 4 SERVICEGALILEO : 8	**1** Load a ROS bag file containing any message type. **2** Select Message Viewer ⊞ from the toolstrip and choose the data source from the drop down to visualize the raw message stored in the rosbag.

The **ROS Bag Viewer** app enables you to visualize messages in a ROS bag file. You can create multiple viewers within the app and visualize different ROS messages simultaneously. For each

viewer, you can filter the supported messages in the bag file and press the **Play** button ▶ from the **Playback** panel for visualization.

Playback

| 0 | 0.66657 | 1.3331 | 1.9997 | 2.6663 | 3.3328 | 3.9994 | 4.666 | 5.3326 | 5.9991 |

Elapsed Time ▼ | 0.000000000 | Playback Speed: | 1x ▼ | ⏮ ▶ ⏭ | Reference Topic: | Automatic ▼

You can choose to display the **Elapsed Time** or the **Timestamp** values associated with the bag file along the slider tick labels. You can visualize the messages by putting a specific numeric value or drag the slider to a particular value of the elapsed time or the time stamp. You can also take a snapshot and export the information for further use.

Control ROS Bag File Playback

After loading the bag file and creating visualizers for desired message types, you can play the ROS bag file. There are various options that control the playback of the bag file. All these options can be found in the **Playback** panel at the bottom of the app.

Reference Topic

Reference Topic is the topic whose frequency determines the bag file playback. The app displays all the messages in the topic that you select as the main signal and displays other messages based on the timestamps of the main signal. You can select a specific topic as the main signal using the **Reference Topic** drop down. If you want to display all messages for all the topics specified among the visualizers, select the topic with the highest frequency as the main signal. In this case, the app repeats messages for the lower frequency topics among the visualizers while displaying all messages from the highest frequency topic.

By default, the option for **Reference Topic** is `Automatic`. When you specify this option, based on the visualizers present in the current instance, the app selects the main signal automatically using this algorithm:
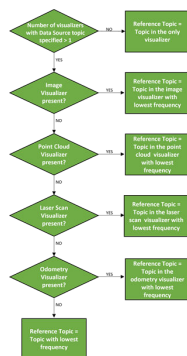
If you select the `Automatic` option, the app selects the lowest frequency topic as the main signal when there are multiple visualizers of the same data type in the app instance. The app also has a hierarchy preference for topics specified among image visualizers, followed by topics in point cloud, laser scan, odometry visualizers. When the lowest frequency topic is selected as the main signal, the app skips messages from other higher frequency topics specified among the visualizers while displaying all the messages from the lowest frequency topic.

Elapsed Time and Timestamp

In the **Playback** panel, by default, the app displays the elapsed time values along the slider tick labels. Elapsed Time denotes the time difference between the Start Time and Current Time on the **Playback** Panel. It is represented in seconds and the app displays the current value of the elapsed

time in the numeric field next to the **Elapsed Time** Time drop down. You can enter a specific value of elapsed time in this field and the app displays the latest messages corresponding to that time in all the visualizers. Alternatively, you can drag the slider to visualize all messages at a specific time value.

If you select the **Timestamp** option from the drop down, the app displays the time stamp values associated with the bag file along the slider tick labels. Timestamp is the current time of playing messages from a particular topic in a rosbag file. It is represented in hh:mm:ss format on the playback panel and displays the epoch convention in the numeric field. Similar to the **Elapsed Time** option, you can seek through the bag file using the numeric field in the **Timestamp** window or the slider. The visualizers update with the latest messages received at the selected timestamp.

Playback Options

After you play the bag file and start visualizing message data, you can change the playback direction by clicking on the forward or backward button. If you wish to step through the messages in the bag file instead, you must first pause the bag file playback and then use the forward or backward step button. You can also set the speed in the **Playback Speed** drop-down for both message playback as well as pause-and-step-through.

Deploy ROS Node for Sign Following Robot with Time Synchronization Using Simulink

This example shows how to use Simulink® to generate a ROS stepping enabled node and control a simulated robot running on a separate ROS-based simulator.

In this example, you deploy a ROS node 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 on 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. This example focuses on time source options for a ROS node generated from Simulink.

To see this example using Simulink or ROS2, see "Sign Following Robot with ROS in Simulink" on page 2-115.

ROS Time Synchronization for Simulation

During Simulink simulation, you can control the simulation pace to iterate the model at a pace similar to the robot running on the ROS device using one of these options:

1   Gazebo Pacer Block - If the robot is running on Gazebo, use a Gazebo Pacer block to achieve time synchronization with Simulink. For an example, see "Sign Following Robot with Time Synchronization Using ROS and Gazebo Co-Simulation" (Robotics System Toolbox).

2   "Simulation Pacing" (Simulink) - Use Simulation Pacing to change the simulation pace to match with robot simulation time.

ROS Time Synchronization for Code Generation and Deployment

As for code generation and node deployment, the default time source is the system clock where the deployed node is running. The generated node is expected to run at a rate that you specified in Simulink. The rate may fluctuate based on the system specifications of the ROS device.

On the other hand, if there are more than one node in your project, you can use the time source from /clock topic to achieve time synchronization. For instructions to enable ROS time model stepping for deployment, see "Enable ROS Time Model Stepping for Deployed ROS Nodes" on page 6-37.

Create Clock topic

If your simulator contains publisher to /clock topic, tune parameters and verify average rate for /clock topic. For example, you can modify **real time update rate** and **max step size** in **Gazebo**

Open a ROS terminal, and run the following command to verify the average rate for /clock topic:

```
$ rostopic hz /clock
```

If the simulator does not publish a /clock topic, consider the following alternatives:

• Use a Digital Clock (Simulink) block as a time source in Simulink and **run simulation** to publish to /clock topic.

• Use a Current Time block as a time source in Simulink and **deploy node** to publish to /clock topic.

- Write a custom MATLAB node or C++ node to publish to `/clock` topic.

You can use `rosClockSimulator` model to publish a `/clock` topic during simulation. You can adjust the **Sample time** of **Digital Clock** block to change the publishing rate of `/clock` topic.

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

In order to maintain the simulation rate you specified in simulink before deployment, `/clock` topic must be updated consistently and at least twice faster than simulation rate. Otherwise, you may experience warning or termination during run time.

This example shows you how to enable ROS time model stepping and generate node to control robot for sign following.

Connect to a Robot Simulator

Start a ROS-based simulator for a differential-drive robot and configure MATLAB® connection with the robot simulator.

To follow along with this example, download a virtual machine using instructions in "Get Started with Gazebo and Simulated TurtleBot" on page 1-206.

- 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 robto simulator.

Open the example model.

```
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 `192.168.243.158:11311`. Enter `192.168.243.158` 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');
```

Enable ROS Time Model Stepping

In the **Prepare** section under **ROS** tab, click **Hardware Settings** to open the model configuration parameters dialog box. Under **Hardware Implementation** > **Hardware board settings** > **Target**

Hardware resources > ROS time, select **Enable ROS time model stepping**. Click *OK* to apply changes and close the dialog.

(Optional) You can also select Enable notification after step check box if you wish to get a message every time ROS time is published. See more information in "Enable ROS Time Model Stepping for Deployed ROS Nodes" on page 6-37.
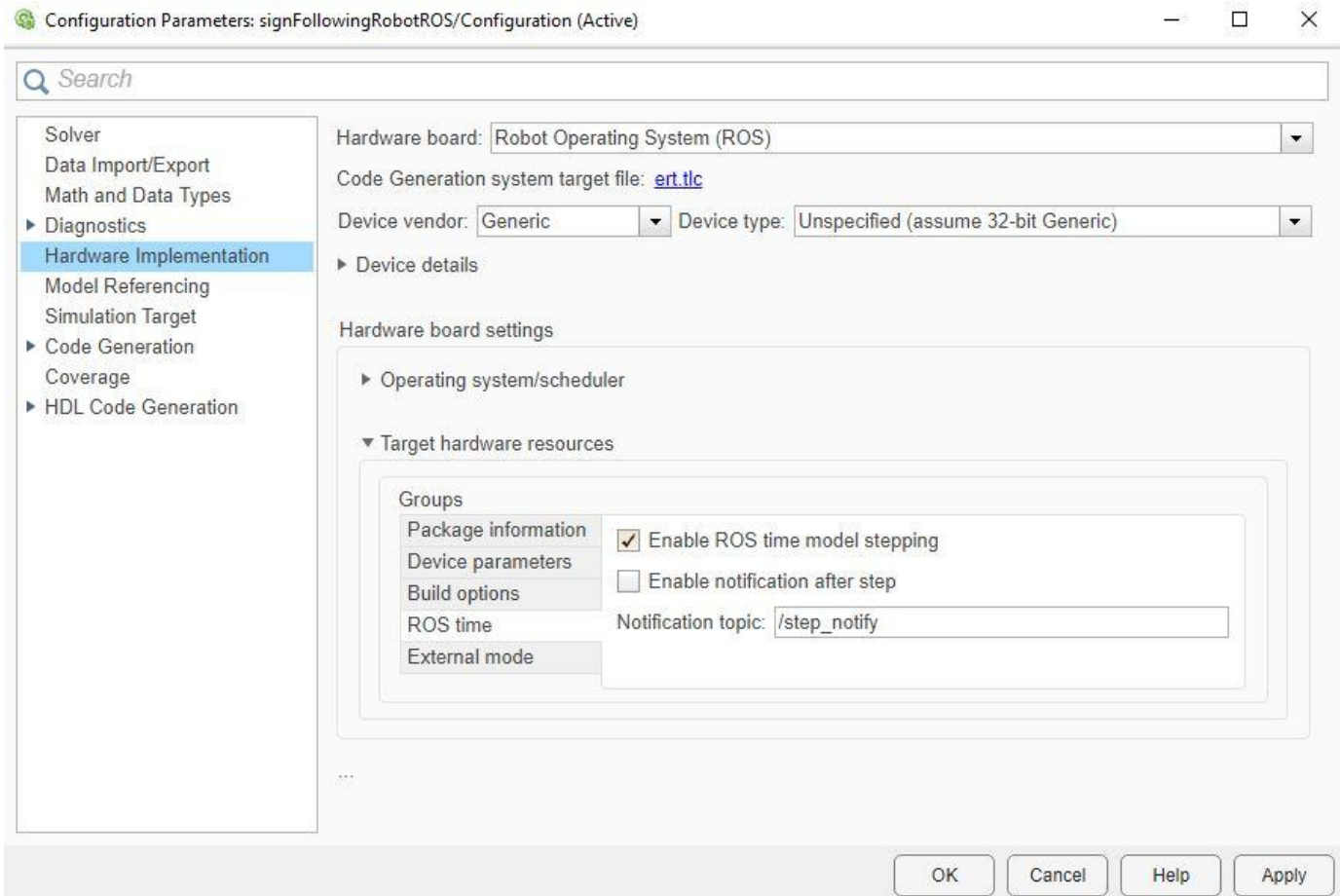
Deploy and Run Node from the Model

In the **Connect** section under **ROS** tab, make sure **Deploy to** is set to **Remote Device**. Click on **Test Connection** to verify that Simulink is able to connect with the remove device. In the **Deploy** section under **ROS** tab, click **Build & Run** to build, deploy, and start the program on remote device.

Once the deployed program starts running, the robot in the simulator will follow the sign and turn based on the color.
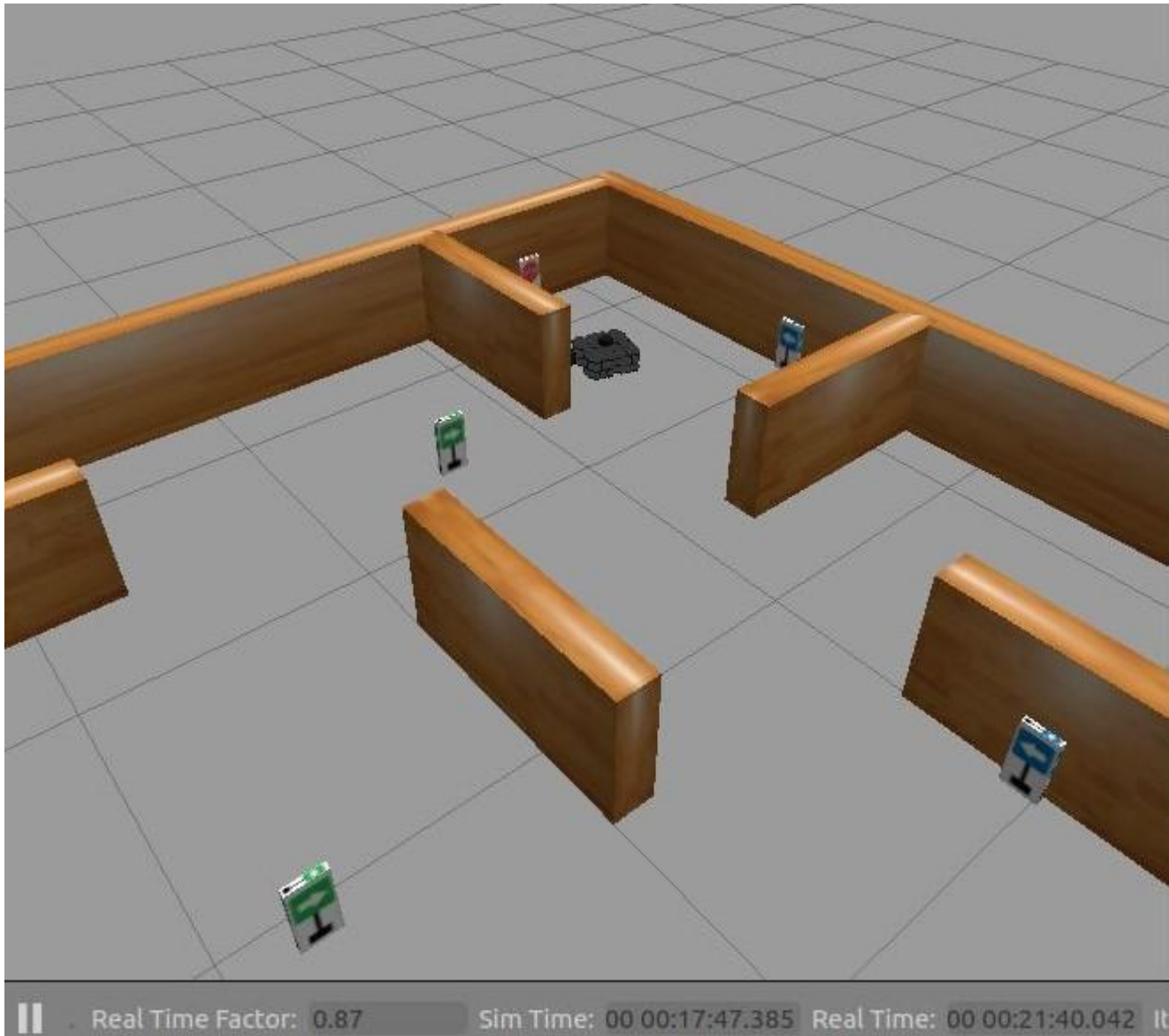
ROS 2 Topics

Generate ROS 2 Custom Messages in MATLAB

Use custom messages to extend the set of message types currently supported in ROS 2. Custom messages are messages that you define. 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, enter `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 2-13.

If this if your first time working with ROS 2 custom messages, see "ROS Toolbox System Requirements".

ROS 2 custom messages are specified in ROS 2 package folders that contain a folder named `msg`. The `msg` folder contains all your custom message type definitions. For example, the `example_b_msgs` package in the `custom` folder, has this folder and file structure.

```
☐ 📁 custom
    ⊞ 📁 example_a_msgs
    ☐ 📁 example_b_msgs
        ☐ 📁 msg
            ✉ Standalone.msg
    ⊞ 📁 example_c_msgs
```

The package contains the custom message type `Standalone.msg`. MATLAB uses these files to generate the necessary files for using the custom messages contained in the package.

In this example, you create 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, you specify the path to the parent folder and 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.

Open a new MATLAB session and create a custom message folder in a local folder.

```
folderPath = fullfile(pwd,"custom");
copyfile("example_*_msgs",folderPath);
```

Specify the folder path for custom message files and use `ros2genmsg` to create custom messages.

```
ros2genmsg(folderPath)

Identifying message files in folder 'C:/Work/custom'.Done.
Removing previous version of Python virtual environment.Done.
Creating a Python virtual environment.Done.
Adding required Python packages to virtual environment.Done.
Copying include folders.Done.
Copying libraries.Done.
Validating message files in folder 'C:/Work/custom'.Done.
[3/3] Generating MATLAB interfaces for custom message packages... Done.
Running colcon build in folder 'C:/Work/custom/matlab_msg_gen/win64'.
```

```
Build in progress. This may take several minutes...
Build succeeded.build log
```

Call `ros2 msg list` to verify creation of new custom messages.

```
ros2 msg list
```

```
action_msgs/CancelGoalRequest
action_msgs/CancelGoalResponse
action_msgs/GoalInfo
action_msgs/GoalStatus
action_msgs/GoalStatusArray
actionlib_msgs/GoalID
actionlib_msgs/GoalStatus
actionlib_msgs/GoalStatusArray
builtin_interfaces/Duration
builtin_interfaces/Time
composition_interfaces/ListNodesRequest
composition_interfaces/ListNodesResponse
composition_interfaces/LoadNodeRequest
composition_interfaces/LoadNodeResponse
composition_interfaces/UnloadNodeRequest
composition_interfaces/UnloadNodeResponse
diagnostic_msgs/AddDiagnosticsRequest
diagnostic_msgs/AddDiagnosticsResponse
diagnostic_msgs/DiagnosticArray
diagnostic_msgs/DiagnosticStatus
diagnostic_msgs/KeyValue
diagnostic_msgs/SelfTestRequest
diagnostic_msgs/SelfTestResponse
example_a_msgs/DependsOnB
example_b_msgs/Standalone
example_c_msgs/DependsOnB
example_interfaces/AddTwoIntsRequest
example_interfaces/AddTwoIntsResponse
example_interfaces/Bool
example_interfaces/Byte
example_interfaces/ByteMultiArray
example_interfaces/Char
example_interfaces/Empty
example_interfaces/Float32
example_interfaces/Float32MultiArray
example_interfaces/Float64
example_interfaces/Float64MultiArray
example_interfaces/Int16
example_interfaces/Int16MultiArray
example_interfaces/Int32
example_interfaces/Int32MultiArray
example_interfaces/Int64
example_interfaces/Int64MultiArray
example_interfaces/Int8
example_interfaces/Int8MultiArray
example_interfaces/MultiArrayDimension
example_interfaces/MultiArrayLayout
example_interfaces/SetBoolRequest
example_interfaces/SetBoolResponse
example_interfaces/String
example_interfaces/TriggerRequest
```

```
example_interfaces/TriggerResponse
example_interfaces/UInt16
example_interfaces/UInt16MultiArray
example_interfaces/UInt32
example_interfaces/UInt32MultiArray
example_interfaces/UInt64
example_interfaces/UInt64MultiArray
example_interfaces/UInt8
example_interfaces/UInt8MultiArray
example_interfaces/WString
geometry_msgs/Accel
geometry_msgs/AccelStamped
geometry_msgs/AccelWithCovariance
geometry_msgs/AccelWithCovarianceStamped
geometry_msgs/Inertia
geometry_msgs/InertiaStamped
geometry_msgs/Point
geometry_msgs/Point32
geometry_msgs/PointStamped
geometry_msgs/Polygon
geometry_msgs/PolygonStamped
geometry_msgs/Pose
geometry_msgs/Pose2D
geometry_msgs/PoseArray
geometry_msgs/PoseStamped
geometry_msgs/PoseWithCovariance
geometry_msgs/PoseWithCovarianceStamped
geometry_msgs/Quaternion
geometry_msgs/QuaternionStamped
geometry_msgs/Transform
geometry_msgs/TransformStamped
geometry_msgs/Twist
geometry_msgs/TwistStamped
geometry_msgs/TwistWithCovariance
geometry_msgs/TwistWithCovarianceStamped
geometry_msgs/Vector3
geometry_msgs/Vector3Stamped
geometry_msgs/Wrench
geometry_msgs/WrenchStamped
lifecycle_msgs/ChangeStateRequest
lifecycle_msgs/ChangeStateResponse
lifecycle_msgs/GetAvailableStatesRequest
lifecycle_msgs/GetAvailableStatesResponse
lifecycle_msgs/GetAvailableTransitionsRequest
lifecycle_msgs/GetAvailableTransitionsResponse
lifecycle_msgs/GetStateRequest
lifecycle_msgs/GetStateResponse
lifecycle_msgs/State
lifecycle_msgs/Transition
lifecycle_msgs/TransitionDescription
lifecycle_msgs/TransitionEvent
logging_demo/ConfigLoggerRequest
logging_demo/ConfigLoggerResponse
map_msgs/GetMapROIRequest
map_msgs/GetMapROIResponse
map_msgs/GetPointMapROIRequest
map_msgs/GetPointMapROIResponse
map_msgs/GetPointMapRequest
```

```
map_msgs/GetPointMapResponse
map_msgs/OccupancyGridUpdate
map_msgs/PointCloud2Update
map_msgs/ProjectedMap
map_msgs/ProjectedMapInfo
map_msgs/ProjectedMapsInfoRequest
map_msgs/ProjectedMapsInfoResponse
map_msgs/SaveMapRequest
map_msgs/SaveMapResponse
map_msgs/SetMapProjectionsRequest
map_msgs/SetMapProjectionsResponse
nav_msgs/GetMapRequest
nav_msgs/GetMapResponse
nav_msgs/GetPlanRequest
nav_msgs/GetPlanResponse
nav_msgs/GridCells
nav_msgs/MapMetaData
nav_msgs/OccupancyGrid
nav_msgs/Odometry
nav_msgs/Path
nav_msgs/SetMapRequest
nav_msgs/SetMapResponse
pendulum_msgs/JointCommand
pendulum_msgs/JointState
pendulum_msgs/RttestResults
rcl_interfaces/DescribeParametersRequest
rcl_interfaces/DescribeParametersResponse
rcl_interfaces/FloatingPointRange
rcl_interfaces/GetParameterTypesRequest
rcl_interfaces/GetParameterTypesResponse
rcl_interfaces/GetParametersRequest
rcl_interfaces/GetParametersResponse
rcl_interfaces/IntegerRange
rcl_interfaces/ListParametersRequest
rcl_interfaces/ListParametersResponse
rcl_interfaces/ListParametersResult
rcl_interfaces/Log
rcl_interfaces/Parameter
rcl_interfaces/ParameterDescriptor
rcl_interfaces/ParameterEvent
rcl_interfaces/ParameterEventDescriptors
rcl_interfaces/ParameterType
rcl_interfaces/ParameterValue
rcl_interfaces/SetParametersAtomicallyRequest
rcl_interfaces/SetParametersAtomicallyResponse
rcl_interfaces/SetParametersRequest
rcl_interfaces/SetParametersResponse
rcl_interfaces/SetParametersResult
rosgraph_msgs/Clock
sensor_msgs/BatteryState
sensor_msgs/CameraInfo
sensor_msgs/ChannelFloat32
sensor_msgs/CompressedImage
sensor_msgs/FluidPressure
sensor_msgs/Illuminance
sensor_msgs/Image
sensor_msgs/Imu
sensor_msgs/JointState
```

```
sensor_msgs/Joy
sensor_msgs/JoyFeedback
sensor_msgs/JoyFeedbackArray
sensor_msgs/LaserEcho
sensor_msgs/LaserScan
sensor_msgs/MagneticField
sensor_msgs/MultiDOFJointState
sensor_msgs/MultiEchoLaserScan
sensor_msgs/NavSatFix
sensor_msgs/NavSatStatus
sensor_msgs/PointCloud
sensor_msgs/PointCloud2
sensor_msgs/PointField
sensor_msgs/Range
sensor_msgs/RegionOfInterest
sensor_msgs/RelativeHumidity
sensor_msgs/SetCameraInfoRequest
sensor_msgs/SetCameraInfoResponse
sensor_msgs/Temperature
sensor_msgs/TimeReference
shape_msgs/Mesh
shape_msgs/MeshTriangle
shape_msgs/Plane
shape_msgs/SolidPrimitive
simple_msgs/AddTwoIntsRequest
simple_msgs/AddTwoIntsResponse
simple_msgs/Num
statistics_msgs/MetricsMessage
statistics_msgs/StatisticDataPoint
statistics_msgs/StatisticDataType
std_msgs/Bool
std_msgs/Byte
std_msgs/ByteMultiArray
std_msgs/Char
std_msgs/ColorRGBA
std_msgs/Empty
std_msgs/Float32
std_msgs/Float32MultiArray
std_msgs/Float64
std_msgs/Float64MultiArray
std_msgs/Header
std_msgs/Int16
std_msgs/Int16MultiArray
std_msgs/Int32
std_msgs/Int32MultiArray
std_msgs/Int64
std_msgs/Int64MultiArray
std_msgs/Int8
std_msgs/Int8MultiArray
std_msgs/MultiArrayDimension
std_msgs/MultiArrayLayout
std_msgs/String
std_msgs/UInt16
std_msgs/UInt16MultiArray
std_msgs/UInt32
std_msgs/UInt32MultiArray
std_msgs/UInt64
std_msgs/UInt64MultiArray
```

```
std_msgs/UInt8
std_msgs/UInt8MultiArray
std_srvs/EmptyRequest
std_srvs/EmptyResponse
std_srvs/SetBoolRequest
std_srvs/SetBoolResponse
std_srvs/TriggerRequest
std_srvs/TriggerResponse
stereo_msgs/DisparityImage
test_msgs/Arrays
test_msgs/ArraysRequest
test_msgs/ArraysResponse
test_msgs/BasicTypes
test_msgs/BasicTypesRequest
test_msgs/BasicTypesResponse
test_msgs/BoundedSequences
test_msgs/Builtins
test_msgs/Constants
test_msgs/Defaults
test_msgs/Empty
test_msgs/EmptyRequest
test_msgs/EmptyResponse
test_msgs/MultiNested
test_msgs/Nested
test_msgs/Strings
test_msgs/UnboundedSequences
test_msgs/WStrings
trajectory_msgs/JointTrajectory
trajectory_msgs/JointTrajectoryPoint
trajectory_msgs/MultiDOFJointTrajectory
trajectory_msgs/MultiDOFJointTrajectoryPoint
unique_identifier_msgs/UUID
visualization_msgs/GetInteractiveMarkersRequest
visualization_msgs/GetInteractiveMarkersResponse
visualization_msgs/ImageMarker
visualization_msgs/InteractiveMarker
visualization_msgs/InteractiveMarkerControl
visualization_msgs/InteractiveMarkerFeedback
visualization_msgs/InteractiveMarkerInit
visualization_msgs/InteractiveMarkerPose
visualization_msgs/InteractiveMarkerUpdate
visualization_msgs/Marker
visualization_msgs/MarkerArray
visualization_msgs/MenuEntry
```

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 2-19.

Create a publisher to use `example_b_msgs/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:
        MessageType: 'example_b_msgs/Standalone'
       int_property: 12
    string_property: 'This is ROS 2 custom message example'
```

Remove the created ROS objects.

```
clear node pub sub
```

Replacing Definitions of Built-In Messages with Custom Definitions

MATLAB provides a lot of built-in ROS 2 message types. You can replace the definitions of those message types with new definitions using the same custom message creation workflow detailed above. When you are replacing the definitions of a built-in message package, you must ensure that the custom message package folder contains new definitions (`.msg` files) for all the message types in the corresponding built-in message package.

Create Shareable ROS 2 Custom Message Package

In this example, you create a shareable ROS 2 custom message package in MATLAB. You must have a ROS 2 package that contains the required `msg` file. This figure shows an example of an appropriate folder structure.
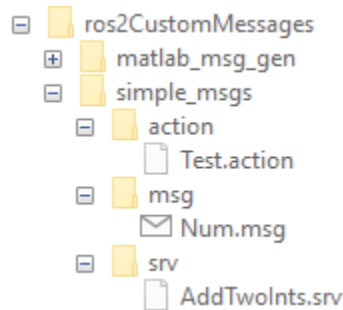
After you prepare your custom message package folder, you specify the path to the parent folder and call `ros2genmsg` with the specified path.

Open a new MATLAB session and create a custom message package folder in a local folder. Choose a short folder path when you generate custom messages on a Windows machine to avoid limitations on the number of characters in the folder path. For example,

```matlab
genDir = fullfile('C:/test/ros2CustomMessages')

genDir = fullfile(pwd,'ros2CustomMessages');
packagePath = fullfile(genDir,'simple_msgs');
mkdir(packagePath)
```

Create a folder named `msg` inside the custom message package folder.

```matlab
mkdir(packagePath,'msg')
```

Create a file named `.msg` inside the `msg` folder.

```matlab
messageDefinition = {'int64 num'};

fileID = fopen(fullfile(packagePath,'msg', ...
               'Num.msg'),'w');
fprintf(fileID,'%s\n',messageDefinition{:});
fclose(fileID);
```

Create a folder named `srv` inside the custom message package folder.

```matlab
mkdir(packagePath,'srv')
```

Create a file named `.srv` inside the `srv` folder.

```matlab
serviceDefinition = {'int64 a'
                     'int64 b'
                     '---'
                     'int64 sum'};

fileID = fopen(fullfile(packagePath,'srv', ...
```

```
                    'AddTwoInts.srv'),'w');
fprintf(fileID,'%s\n',serviceDefinition{:});
fclose(fileID);
```

Create a folder named `action` inside the custom message package folder.

```
mkdir(packagePath,'action')
```

Create a file named `.action` inside the `action` folder.

```
actionDefinition = {'int64 goal'
                    '---'
                    'int64 result'
                    '---'
                    'int64 feedback'};
```

```
fileID = fopen(fullfile(packagePath,'action', ...
            'Test.action'),'w');
fprintf(fileID,'%s\n',actionDefinition{:});
fclose(fileID);
```

Generate custom messages from ROS 2 definitions in `.msg`, `.srv` and `.action` files. Use the `CreateShareableFile` name-value argument to create a shareable ZIP archive of the generated custom messages.

For information about how to use use this ZIP archive to register the custom messages in another machine, see `ros2RegisterMessages`.

```
ros2genmsg(genDir,CreateShareableFile=true);
```

```
Identifying message files in folder 'C:/Users/echakrab/OneDrive - MathWorks/Documents/MATLAB/Exam
Creating a Python virtual environment.Done.
Adding required Python packages to virtual environment.Done.
Copying include folders.Done.
Copying libraries.Done.
Done.
[1/1] Generating MATLAB interfaces for custom message packages... Done.
Running colcon build in folder 'C:/Users/echakrab/OneDrive - MathWorks/Documents/MATLAB/ExampleMa
Build in progress. This may take several minutes...
```

```
⊟ 📁 ros2CustomMessages
   ⊞ 📁 matlab_msg_gen
   ⊞ 📁 simple_msgs
   ⊞ 📁 matlab_msg_gen.zip
```

Verify creation of the new custom messages by entering `ros2 msg list` in the Command Window.

```
simple_msgs/AddTwoIntsRequest
simple_msgs/AddTwoIntsResponse
simple_msgs/Num
simple_msgs/TestFeedback
simple_msgs/TestGoal
simple_msgs/TestResult
```

Register ROS 2 Custom Messages to MATLAB

In this example, you create ROS 2 custom messages in MATLAB® to share them on another machine. This other machine must run on the same platform and same version of MATLAB®. You must have a ROS 2 package that contains the required `msg` file, as this figure shows.
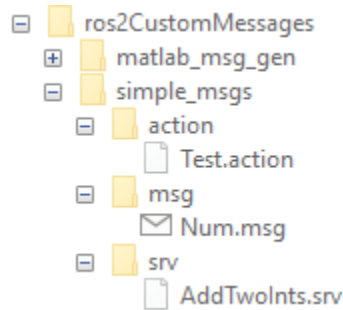
Open a new MATLAB session and create a custom message package folder in a local folder. Specify a short folder path when you generate custom messages on a Windows machine to avoid limitations on the number of characters in the folder path. For example,

```
genDir = fullfile('C:/Work/ros2CustomMessages').
```

```
genDir = fullfile(pwd,'ros2CustomMessages');
packagePath = fullfile(genDir,'simple_msgs');
mkdir(packagePath)
```

Create a folder named `msg` inside the custom message package folder.

```
mkdir(packagePath,'msg')
```

Create a file named `.msg` inside the `msg` folder.

```
messageDefinition = {'int64 num'};

fileID = fopen(fullfile(packagePath,'msg', ...
                'Num.msg'),'w');
fprintf(fileID,'%s\n',messageDefinition{:});
fclose(fileID);
```

Create a folder named `srv` inside the custom message package folder.

```
mkdir(packagePath,'srv')
```

Create a file named `.srv` inside the `srv` folder.

```
serviceDefinition = {'int64 a'
                     'int64 b'
                     '---'
                     'int64 sum'};

fileID = fopen(fullfile(packagePath,'srv', ...
                'AddTwoInts.srv'),'w');
fprintf(fileID,'%s\n',serviceDefinition{:});
fclose(fileID);
```

Create a folder named `action` inside the custom message package folder.

```
mkdir(packagePath,'action')
```

Create a file named `.action` inside the `action` folder.

```
actionDefinition = {'int64 goal'
                    '---'
                    'int64 result'
                    '---'
                    'int64 feedback'};
```

```
fileID = fopen(fullfile(packagePath,'action', ...
               'Test.action'),'w');
fprintf(fileID,'%s\n',actionDefinition{:});
fclose(fileID);
```

Generate the custom messages from the ROS 2 definitions in `.msg`, `.srv` and `.action` files as a shareable ZIP archive.

```
ros2genmsg(genDir,CreateShareableFile=true)
```

```
Identifying message files in folder 'C:/Work/ros2CustomMessages'.Done.
Validating message files in folder 'C:/Work/ros2CustomMessages'.Done.
[1/1] Generating MATLAB interfaces for custom message packages... Done.
Running colcon build in folder 'C:/Work/ros2CustomMessages/matlab_msg_gen/win64'.
Build in progress. This may take several minutes...
Build succeeded.build log
Generating zip file in the folder 'C:/Work/ros2CustomMessages'.Done.
```
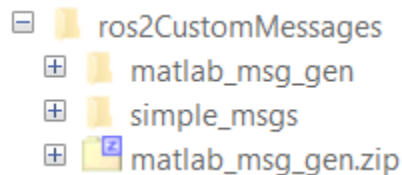
- ros2CustomMessages
  - matlab_msg_gen
  - simple_msgs
  - matlab_msg_gen.zip

Copy the generated custom messages in the ZIP archive to the target computer and register it using the `ros2RegisterMessages` function.

```
ros2RegisterMessages(genDir)
```

- ros2CustomMessages
  - install
  - simple_msgs
  - matlab_msg_gen.zip
  - VersionInfo.mat
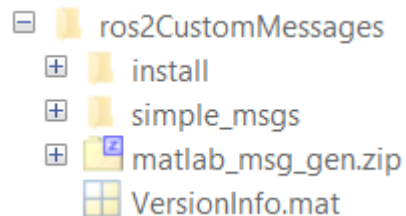
Run `ros2 msg list` on the target computer to view the custom messages for using in the workflow.

```
simple_msgs/AddTwoIntsRequest
simple_msgs/AddTwoIntsResponse
simple_msgs/Num
simple_msgs/TestFeedback
simple_msgs/TestGoal
simple_msgs/TestResult
```

Configure MATLAB Coder for ROS 2 Node Generation

In this section...
"Create MATLAB Coder Configuration Object" on page 4-14
"Configure Hardware Properties" on page 4-14
"Configure Build Options" on page 4-15
"Specify Additional Hardware Properties and ROS Dependencies" on page 4-17

To generate C++ code for ROS 2 Nodes from MATLAB functions, you must configure a MATLAB Coder configuration object. This topic shows you how to configure the properties of the object to customize ROS 2 Node generation.

By default the node uses the `'rmw_fastrtps_cpp'` ROS middleware (RMW) implementation unless otherwise specified by the `RMW_IMPLEMENTATION` environment variable. For more information on RMW implementations see "Switching Between ROS Middleware Implementations" on page 2-172.

```
setenv('RMW_IMPLEMENTATION','rmw_cyclonedds_cpp')
```

Note

- When the `coder.Hardware.BuildAction` property is specified as `'Build and Run'`, the `RMW_IMPLEMENTATION` environment variable must be set before code generation.

- When the `coder.Hardware.BuildAction` property is specified as `'None'` or `'Build and Load'`, the `RMW_IMPLEMENTATION` environment variable must be set after code generation.

Create MATLAB Coder Configuration Object

To create a MATLAB Coder configuration object, use the `coder.config` object. ROS Toolbox only supports the executable build type for the MATLAB Coder configuration object.

```
cfg = coder.config('exe');
```

Configure Hardware Properties

```
cfg.Hardware = coder.hardware('Robot Operating System 2 (ROS 2)');
```

Specify the `Hardware` property of the object as a `'Robot Operating System 2 (ROS 2)'` hardware configuration object, using the `coder.hardware` function.

Then, you can specify these `coder.Hardware` properties specific to ROS 2 node generation. For remote device deployment, the device parameters automatically save to MATLAB preferences, and are used the next time you set the deployment site to `'Remote Device'`.

Property	Values	Description
`cfg.Hardware.DeployTo`	`'Localhost'` (default) `'Remote Device'`	Deployment site for the code, which can be a remote device or the local host device.

Property	Values	Description

`cfg.Hardware.BuildAction`	`'None'` (default)
	
	`'Build and Load'`
	
	`'Build and Run'`
		Build action for code generation.

`cfg.Hardware.RemoteDevic` `eAddress`	`'192.168.128.130'` (default) character vector	Remote IP address or host name of the remote device.
`cfg.Hardware.RemoteDevic` `eUsername`	`'user'` (default) character vector	User name for the remote device.
`cfg.Hardware.RemoteDevic` `ePassword`	`'password'` (default) character vector	Password for the remote device.
`cfg.Hardware.ROS2Workspa` `ce`	`'~/ros2_ws'` (default) character vector	Path to the ROS 2 workspace in the remote device. For Windows devices, the default value replaces `'~'` with the user path.
`cfg.Hardware.ROS2Folder`	`'/opt/ros/foxy'` (default) character vector	Path to the ROS installation folder. Leave it blank to use the MATLAB ROS distribution.
`cfg.Hardware.PackageMain` `tainerName`	`'ROS User'` (default) character vector	ROS package maintainer name, which is used for `package.xml` generation.
`cfg.Hardware.PackageMain` `tainerEmail`	`'rosuser@test.com'` (default) character vector	ROS package maintainer e-mail ID, used for `package.xml` generation.
`cfg.Hardware.PackageLice` `nse`	`'BSD'` (default) character vector	ROS license information, used for `package.xml` generation.
`cfg.Hardware.PackageVers` `ion`	`'1.0.0'` (default) character vector	Version number of the ROS package.

Configure Build Options

You can also specify whether you only need the code to be generated using the `GenCodeOnly` property of the coder configuration object. This table highlights how the code generation and build behavior changes based on `GenCodeOnly` and other hardware properties.

`cfg.GenCodeOnly`	`cfg.Hardware.Deplo` `yTo`	`cfg.Hardware.Build` `Action`	Code Generation and Build Behavior
`true`	`'Localhost'`	`'None'`	Generates ROS Package source code folder `src` on local host device
		`'Build and Load'`	
		`'Build and Run'`	

cfg.GenCodeOnly	cfg.Hardware.DeployTo	cfg.Hardware.BuildAction	Code Generation and Build Behavior
	'Remote Device'	'None'	Generates these artifacts on local host device: • ROS package source code folder `src` • TAR archive of the source code in a `.tgz` file • Shell script to extract and set up the package on remote device
		'Build and Load'	
		'Build and Run'	
false	'Localhost'	'None'	Generates ROS Package source code folder `src` on local host device
		'Build and Load'	Generates ROS Package source code folder `src` and builds the executable on local host device
		'Build and Run'	Generates ROS Package source code folder `src`, builds and runs the executable on local host device
	'Remote Device'	'None'	Generates these artifacts on local host device: • ROS package source code folder `src` • TAR archive of the source code in a `.tgz` file • Shell script to extract and set up the package on remote device

cfg.GenCodeOnly	cfg.Hardware.DeployTo	cfg.Hardware.BuildAction	Code Generation and Build Behavior
		`'Build and Load'`	Generates these artifacts on local host device and builds the executable on remote device: • ROS package source code folder `src` • TAR archive of the source code in a `.tgz` file • Shell script to extract and set up the package on remote device
		`'Build and Run'`	Generates these artifacts on local host device, builds and runs the executable on remote device: • ROS package source code folder `src` • TAR archive of the source code in a `.tgz` file • Shell script to extract and set up the package on remote device

For example, the following code specifies that the generated code must be deployed to the remote device and run after build. It also specifies the remote device parameters.

```
cfg.Hardware.DeployTo = 'Remote Device';
cfg.Hardware.BuildAction = 'Build and run';
cfg.Hardware.RemoteDeviceAddress = '192.168.243.144';
cfg.Hardware.RemoteDeviceUsername = 'user';
cfg.Hardware.RemoteDevicePassword = 'password';
```

Specify Additional Hardware Properties and ROS Dependencies

To specify additional hardware-specific configuration parameters use `coder.HardwareImplementation` (MATLAB Coder) object. For example, this code specifies the manufacturer and type of the hardware as well as `long long` data type support for `int64` and `uint64` values.

```
cfg.HardwareImplementation.ProdHWDeviceType = 'Intel->x86-64 (Linux 64)';
cfg.HardwareImplementation.ProdLongLongMode = true;
```

To specify external ROS packages as dependencies for the generated ROS node, specify appropriate custom toolchain options in the coder configuration object.

```
cfg.BuildConfiguration = 'Specify';
cfg.CustomToolchainOptions{4} = '"ROS2_PKG1","ROS2_PKG2"'; % Add ROS 2 Required Packages
cfg.CustomToolchainOptions{6} = '/usr/include/opencv';   % Add Include Directories
cfg.CustomToolchainOptions{8} = '"-lopencv_core","-lopencv_shape"'; % Add Link Libraries
cfg.CustomToolchainOptions{10} = '/usr/lib/opencv';     % Add Library Paths
cfg.CustomToolchainOptions{12} = '-DMYVAR=1'; % Add Defines
```

See Also
ros2publisher | ros2subscriber | ros2node | ros2message

Generate Standalone ROS 2 Node

This example shows how to generate C++ code for a standalone ROS 2 node from a MATLAB® function. It then shows how to build and run the ROS 2 node on a Windows® machine.

Prerequisites

To run this example, you require access to:

- An Ubuntu Linux system with ROS 2 and an SSH server installed. You can use your own Ubuntu ROS 2 system, or you can use the Linux virtual machine for ROS Toolbox examples. For more information, see "Get Started with Gazebo and Simulated TurtleBot" on page 1-206.
- A C/C++ compiler that is configured properly. You can use the `mex -setup cpp` to view and change the default compiler. For more details, see Change Default Compiler.

Control ROS 2 - Enabled Robot

Open the function `robotROS2FeedbackControl`, which contains a proportional controller introduced in the "Feedback Control of a ROS-Enabled Robot" on page 1-149 example. This function subscribes to the `/odom` topic to get the current odometry status of the robot, and then publishes the output of a proportional controller as a `geometry_msgs/Twist` message to the `/cmd_vel` topic. This provides the control commands for the robot to move towards the desired position.

Copy the `robotROS2FeedbackControl` function to your local directory and change the `defaultDesiredPos` variable to the desired coordinates.

Start a Gazebo Empty World environment for ROS 2 from the Linux virtual machine by running these running commands on the desktop. In the Linux virtual machine for ROS Toolbox, the turtlebot robot is located at the `[0,0]` location by default. This Gazebo world contains a Turtlebot robot, which publishes and subscribes to messages on a ROS 2 network.

```
~$ source /opt/ros/foxy/local_setup.bash
~$ export TURTLEBOT3_MODEL=burger && ros2 launch turtlebot3_gazebo empty_world.launch.py
```

Create a `ros2device` object. Specify the `deviceAddress`, `username,` and `password` values of the Virtual Machine running Gazebo. This establishes an SSH connection between the ROS 2 device and MATLAB.

```
gazeboVMDevice = ros2device('172.21.17.220',"user","password")
```

Create a MATLAB ROS 2 node in the same ROS network as the virtual machine. See the list of available ROS 2 nodes.

```
ros2 node list
```

Run the controller. The robot is moving towards the published destination. At the same time, observe the trajectory of the robot. Keep this figure open to compare the behavior of MATLAB execution and the generated executable node.

```
robotROS2FeedbackControl
```

To change the destination while the robot is moving. Open a new terminal in the virtual machine, source the ROS repository, and publish the new destination coordinates in the form of a `std_msgs/Float64MultiArray` message to the `/dest` topic.

```
~$ source /opt/ros/foxy/local_setup.bash
~$ ros2 topic pub -1 /dest std_msgs/Float64MultiArray "{data:[0,0]}"
```

You can also adjust the `distanceThre`, `linearVelocity`, and `rotationGain` values in the `robotROS2FeedbackControl.m` to change the desired robot behavior. For the proportional controller in this example, you must specify values in these ranges to ensure robust performance. Alternatively, you can replace the proportional controller with a custom controller for performance comparison. To view the behavior, reset the robot on the virtual machine by pressing **Ctrl-R** in Gazebo.

```
distanceThre: 0<x<1
linearVelocity: 0<x<3
rotationGain: 0<x<6
```

The controller loop terminates after the robot reaches the goal or if more than 40 seconds has elapsed since the start time. Alternatively, you can terminate the controller at any time using **Ctrl-C** or entering this command in the terminal from the virtual machine. If you open a new terminal in the virtual machine, you must source the ROS repository.

```
~$ ros2 topic pub -1 /stop std_msgs/Bool "data: 1"
```

Create a Function for Code Generation

To generate a standalone C++ node, modify the function to make it compatible for code generation.

- Remove the lines of code related to plot.
- Save the modified MATLAB function to `robotROS2FeedbackControlCodegen.m`.
- Ensure any other modifications that you made in `robotROS2FeedbackControl` function are reflected in `robot2ROSFeedbackControlCodegen`.

Generate Executable for `robotROS2FeedbackControlCodegen`

Reset the Gazebo scene after simulation using **Ctrl-R** on the virtual machine.

Generate an executable node for the `robotROS2FeedbackControlCodegen` function. Specify the hardware as "`Robot Operating System 2 (ROS 2)`". Set the build action to "`Build and run`" so that the ROS 2 node starts running after you generate it. Specify the coder configuration parameters for remote deployment. This example specifies the remote device parameters of the virtual machine running Gazebo. Note that the actual values might be different for your remote device. Verify them before deployment.

```
cfg = coder.config("exe");
cfg.Hardware = coder.hardware("Robot Operating System 2 (ROS 2)");
cfg.Hardware.BuildAction = "Build and run";
cfg.Hardware.RemoteDeviceAddress = '172.21.17.220';
cfg.Hardware.RemoteDeviceUsername = "user";
cfg.Hardware.RemoteDevicePassword = "password";
cfg.Hardware.DeployTo = "Remote Device";
codegen robotROS2FeedbackControlCodegen -args {} -config cfg
```

Reset the Gazebo scene by using **Ctrl-R** on the virtual machine and call `plotPathFeedbackControl` to plot the robot trajectory.

```
plotPathFeedbackControl
```

The configuration generates and runs the ROS 2 node on your remote device. You can opt to deploy and run the ROS 2 node on the local machine by modifying the coder configuration object, `cfg`.

```
cfg = coder.config('exe');
cfg.Hardware = coder.hardware('Robot Operating System 2 (ROS 2)');
cfg.Hardware.BuildAction = 'Build and run';
cfg.Hardware.DeployTo = 'Localhost';
codegen robotROS2FeedbackControlCodegen -args {} -config cfg
plotPathFeedbackControl
```

Verify Generated ROS Node

After the generated executable starts running for the specified destination coordinates, you verify the trajectory of the robot against the MATLAB execution. You can also observe the robot moving in Gazebo on the virtual machine. You can publish a different destination coordinate while the robot is in motion. The Control a ROS-Enabled Robot with a Function on page 4-19 section, which shows how to publish a new set of destination coordinates through a virtual machine terminal.

Terminate the generated ROS 2 node by using the `stopNode` function of the `ros2Device` object. Alternatively, you can terminate the execution by pressing **Ctrl-C** or sending a message to the `/stop` topic. Reset the Gazebo scene by using **Ctrl-R** on VM Machine.

```
stopNode(gazeboVMDevice,'robotROS2FeedbackControlCodegen')
```

ROS and ROS 2 CUDA MATLAB Topics

Generate CUDA ROS and CUDA ROS 2 Nodes Using MATLAB Coder and GPU Coder

Configure MATLAB® Coder™ software to generate and build CUDA® ROS and CUDA® ROS 2 nodes from a MATLAB function. You can deploy the CUDA nodes to local or remote GPU target devices.

In this example, you will deploy MobileNet-v2 object detection algorithm on a local host computer with a supported NVIDIA graphics card.

Begin by creating a ROS *publisher* to publish the ROS image and run prediction on it using MobileNet-v2 object detection algorithm. You then generate a local host node to deploy the object detection algorithm on a local host computer with a CUDA-enabled NVIDIA GPU. Finally, you create a ROS *subscriber* to plot the top five prediction scores.
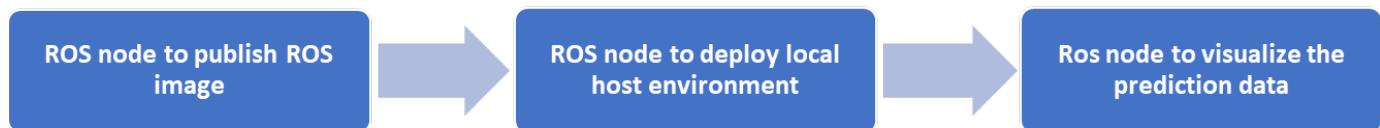
Prerequisites

1. Install and set up these MathWorks® products and support packages: MATLAB® Coder™ (required), Embedded Coder® (recommended), GPU Coder™, and GPU Coder Interface for Deep Learning (required for Deep Learning).

Note: GPU Coder™ is supported only on Windows and Linux.

2. You can also install MATLAB® Coder™ Support Package for NVIDIA® Jetson™ and NVIDIA DRIVE™ Platforms (required for an NVIDIA hardware connection).

3. Your target device can be a local host computer with a CUDA-enabled NVIDIA GPU, or a remote device such as an NVIDIA Jetson board.

4. To use GPU Coder for CUDA code generation, you must install NVIDIA graphics driver, CUDA toolkit, cuDNN library, and TensorRT library. For more information on local host deployment and host simulation, see "Installing Prerequisite Products" (GPU Coder).

5. To set up the environment variables, see "Setting Up the Prerequisite Products" (GPU Coder).

6. To ensure you have the required third-party software, see "ROS Toolbox System Requirements".

Verify GPU Environment for GPU Code Generation

To verify that your development computer has the drivers, tools, libraries, and configuration required for GPU code generation, see "Setting Up the Prerequisite Products" (GPU Coder).

Create a `coder.checkGpuInstall` (GPU Coder) object and set its property values. Then, verify your GPU code generation environment using the `coder.checkGpuInstall` (GPU Coder) function.

```
gpuEnvObj = coder.gpuEnvConfig;
gpuEnvObj.BasicCodegen = 1;
```

```matlab
gpuEnvObj.BasicCodeexec = 1;
gpuEnvObj.DeepLibTarget = "tensorrt";
gpuEnvObj.DeepCodeexec = 1;
gpuEnvObj.DeepCodegen = 1;
results = coder.checkGpuInstall(gpuEnvObj)
```

The output is representative. Results might differ from user to user.

```
Compatible GPU              : PASSED
CUDA Environment            : PASSED
    Runtime   : PASSED
    cuFFT     : PASSED
    cuSOLVER  : PASSED
    cuBLAS    : PASSED
cuDNN Environment           : PASSED
TensorRT Environment        : PASSED
Basic Code Generation       : PASSED
Basic Code Execution        : PASSED
Deep Learning (TensorRT) Code Generation: PASSED
Deep Learning (TensorRT) Code Execution: PASSED

results =

  struct with fields:

                 gpu: 1
                cuda: 1
               cudnn: 1
            tensorrt: 1
         basiccodegen: 1
        basiccodeexec: 1
          deepcodegen: 1
          deepcodeexec: 1
     tensorrtdatatype: 1
            profiling: 0
```

Configure MATLAB Coder for CUDA ROS Node Generation

Initialize ROS Core and Global Node.

```matlab
rosinit
```

```
Launching ROS Core...
.Done in 1.1242 seconds.
Initializing ROS master on http://172.18.226.60:11311.
Initializing global node /matlab_global_node_73382 with NodeURI http://hyd-jkonakal:49931/ and Ma
```

Create MATLAB Function for Deep Learning

Write a MATLAB function to subscribe to the image and apply MobileNet-v2 algorithm to analyze the probability of pepper.

```matlab
type('mobilenet_ros1node_predictor.m')
```

```matlab
function mobilenet_ros1node_predictor
% Copyright 2022 The MathWorks, Inc.
%#codegen

% Create an image subscriber
```

```
imgSub = rossubscriber("/image_topic","sensor_msgs/Image","DataFormat","struct");

% Create a scores results publisher
scoresPub = rospublisher('/score_results','std_msgs/Float32MultiArray',...
    'DataFormat','struct');
scoresMsg = rosmessage("std_msgs/Float32MultiArray", "DataFormat","struct");

% Create a index results publisher
indexPub = rospublisher('/index_results','std_msgs/Float32MultiArray',...
    'DataFormat','struct');
indexMsg = rosmessage("std_msgs/Float32MultiArray", "DataFormat","struct");

upperImageSize = [224,224,3];
in = zeros(upperImageSize,'uint8');

r = rosrate(30);
reset(r);

while (1)
    receivedImage = imgSub.LatestMessage;
    while ~isempty(receivedImage) && ~isempty(receivedImage.Data)
        imageData = rosReadImage(receivedImage,"Encoding","rgb8");
        % Crop the received image if it exceeds the upper size
        imgSize = size(imageData);
        index = min(imgSize,upperImageSize);
        in(1:index(1),1:index(2),1:index(3)) = imageData(1:index(1),1:index(2),1:index(3));

        % pass image to mobilenetv2_predict network
        predict_scores = mobilenetv2_predict(in);
        [scores,indexes] = sort(predict_scores, 'descend');

        scoresMsg.Data = zeros(1000,1,'single');
        indexMsg.Data = zeros(1000,1,'single');

        if ~isempty(scores)
            scoresMsg.Data = single(scores');
            indexMsg.Data = single(indexes');
        end

        send(scoresPub,scoresMsg);
        send(indexPub,indexMsg);
    end
    waitfor(r);
end
end
```

Generate Local Host Node

Configure MATLAB Code Generation `config` object settings and generate local host node.

```
cfg = coder.config("exe");
```

Select the ROS version.

```
cfg.Hardware = coder.hardware("Robot Operating System (ROS)");
```

Select Localhost to deploy the object detection algorithm on a local host computer with a CUDA-enabled NVIDIA GPU.

NOTE: Select Remote Hardware to deploy the object detection algorithm on a remote target device such as NVIDIA Jetson board.

```
cfg.Hardware.DeployTo = "Localhost";
cfg.Hardware.BuildAction = "Build and run";
cfg.HardwareImplementation.ProdHWDeviceType = "Intel->x86-64 (Windows64)";
cfg.HardwareImplementation.ProdLongLongMode = true; % Use 'long long' for Int64 or Uint64 data ty
```

Create the `GpuCodeConfig` object and enable it.

```
cfg.GpuConfig = coder.GpuCodeConfig;
cfg.GpuConfig.Enabled = true;
```

Create a `DeepLearningConfig` object and select the appropriate DNN library.

```
cfg.DeepLearningConfig = coder.DeepLearningConfig('cudnn');
codegen mobilenet_ros1node_predictor -config cfg
```

```
Warning: Generation of C++ classes for MATLAB classes is not supported by GPU Coder. Disabling th

---
Running ROS node.
Warning: Function 'mobilenet_ros1node_predictor' does not terminate due to an infinite loop.

Warning in ==> mobilenet_ros1node_predictor Line: 50 Column: 1
Code generation successful (with warnings): View report
```

When `codegen` is successful, it creates a ROS node `/mobilenet_ros1node_predictor` on your local machine. You can verify the successful creation of the node by using `rosnode list`.

```
rosnode list
```

```
/matlab_global_node_73382
/mobilenet_ros1node_predictor
```

Create a publisher to read the ROS image and publish it to the `/image_topic`.

```
publishpepperimage
```

Visualize Probability Data

Subscribe to the topic and plot the top five prediction scores.

```
visualizepredictiondata
```

Top Five Predictions That Use MobileNet-v2
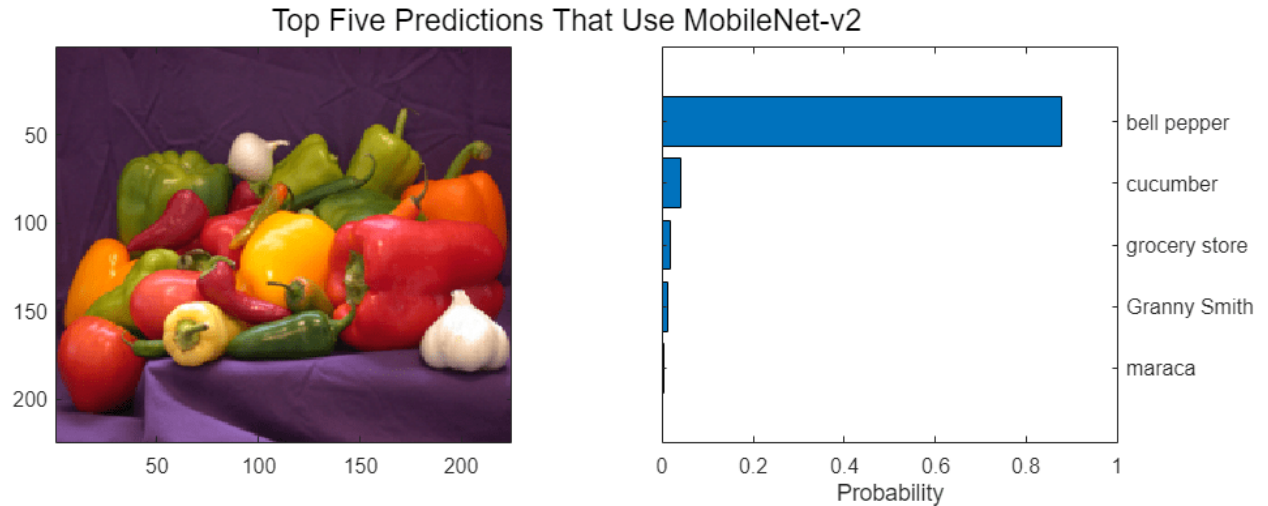
Configure MATLAB Coder for CUDA ROS 2 Node Generation

To generate a CUDA ROS 2 node, follow the same steps that you used to generate a CUDA ROS node. Replace all ROS functions with corresponding ROS 2 functions and configure the `config` object settings for executable code.

```
cfg = coder.config("exe");
```

Select the ROS version.

```
cfg.Hardware = coder.hardware("Robot Operating System 2 (ROS 2)");
```

Select Localhost to deploy the object detection algorithm on a local host computer with a CUDA-enabled NVIDIA GPU.

NOTE: Select Remote Hardware to deploy the object detection algorithm on a remote target device such as NVIDIA Jetson board.

```
cfg.Hardware.DeployTo = "Localhost";
cfg.Hardware.BuildAction = "Build and run";
cfg.HardwareImplementation.ProdHWDeviceType = "Intel->x86-64 (Windows64)";
cfg.HardwareImplementation.ProdLongLongMode = true; % Use 'long long' for Int64 or Uint64 data ty
```

Create the `GpuCodeConfig` object and enable it.

```
cfg.GpuConfig = coder.GpuCodeConfig;
cfg.GpuConfig.Enabled = true;
```

Create a `DeepLearningConfig` object and select the appropriate DNN library.

```
cfg.DeepLearningConfig = coder.DeepLearningConfig('cudnn');
```

Limitations

The build folder path cannot contain any spaces.

Related Topics

1   "Generate a Standalone ROS Node from MATLAB" on page 1-164

2 "Code Generation for Deep Learning Networks" (GPU Coder)

ROS Simulink Topics

ROS Simulink Support and Limitations

In this section...
"ROS Model Reference" on page 6-2
"Remote Desktop" on page 6-2
"ROS 2 Model Build Failure" on page 6-2

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 3.9 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 \Python39\python.exe

See Also

Related Examples

ROS Simulink Interaction

In this section...
"MATLAB ROS Information" on page 6-4
"Simulink ROS Node" on page 6-4
"Differences Between Simulation and Generated Code" on page 6-4
"Publishers and Subscribers in Simulink" on page 6-5
"ROS Model Reference" on page 6-5

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 6-21 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 1-195 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')
```
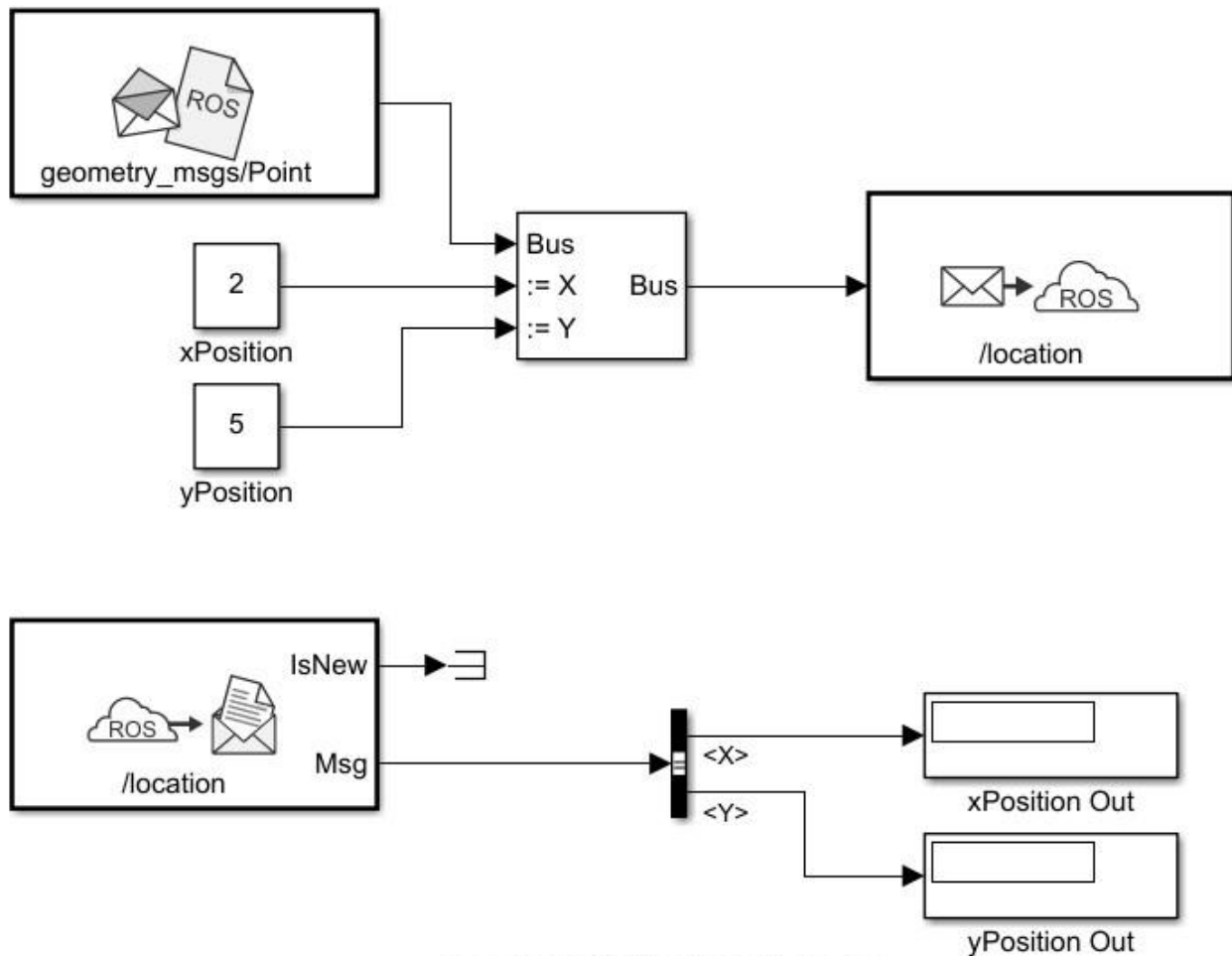

Copyright 2018 The MathWorks, Inc.

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 2.174 seconds.
Initializing ROS master on http://172.30.131.134:52753.
Initializing global node /matlab_global_node_23613 with NodeURI http://bat6234win64:63821/ and M
```

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_23613 with NodeURI http://bat6234win64:63821/ and I
Shutting down ROS master on http://172.30.131.134:52753.
```

Update Header Field of a ROS Message in Simulink®

This example illustrates how to update the header field of a ROS message using Simulink®.

Some ROS messages contain a specific Header field which maps to `std_msgs/Header` message type. The Header field contains the timestamp and coordinate frame information of the ROS message. This example model shows how to use the Header Assignment Block to update that information for a ROS message, in Simulink®.

```
open_system('rosHeaderAssignmentBlockExampleModel.slx')
```

The Blank Message block creates an empty ROS message of type, `sensor_msgs/LaserScan`. Any other message type that contains a Header field of type `std_msgs/Header` can be used here, instead. The output of the Blank Message block is then fed to the Header Assignment block, which updates the Header field of this message. For display, the `frame_id` and `stamp` values of the updated ROS message Header are selected from the list of bus elements using Bus Selector blocks. Additionally, a blank `/rosgraph_msgs/Clock` message is created and a custom time based on the current simulation time is published to the `/clock` topic on the ROS network.
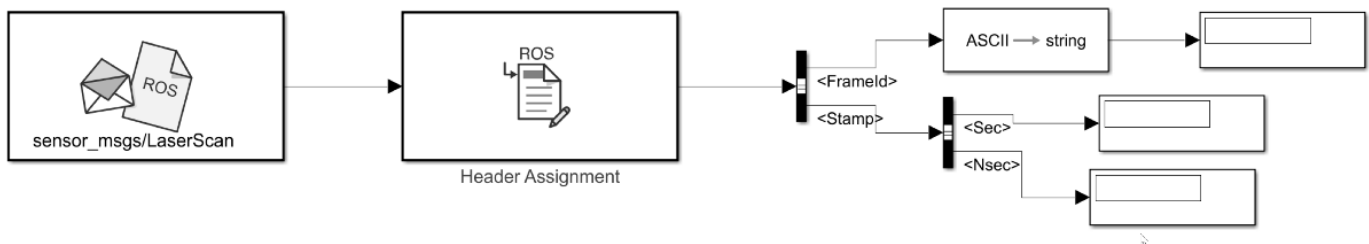
Copyright 2021 The MathWorks, Inc.
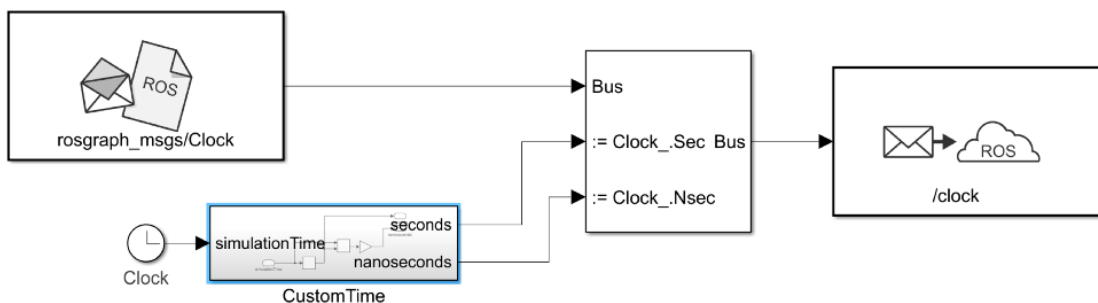
Update coordinate frame id and timestamp values in the Header

Open the Header Assignment block to display its block parameters. The **Set Frame ID** option is selected and the name of the coordinate frame that is associated with the message is specified in the text box as `lidar_link`. This will be set as the `frame_id` value for the Header. The **Set Timestamp** option is also selected which sets the `stamp` value of the Header to the current ROS System time, by default. The **Header field name** option is set to `Use the default Header field name` because, the name of the Header field in a blank message is its default value, `header`. If you are using a ROS message with a custom name for the Header field, you can select the `Specify your own` option from the dropdown and specify the name of the Header field in the text box.

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

```
rosinit

Launching ROS Core...
..Done in 2.7222 seconds.
Initializing ROS master on http://172.30.131.134:52101.
Initializing global node /matlab_global_node_49543 with NodeURI http://bat6234win64:58249/ and Ma
```

Run the model. You can see the updated values for the `frame_id` and `stamp` fields of the ROS Message in their respective displays.

```
sim('rosHeaderAssignmentBlockExampleModel')
```
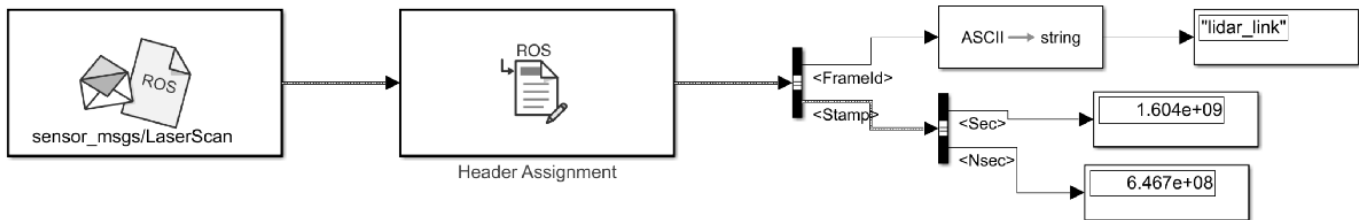

Copyright 2021 The MathWorks, Inc.

Update timestamp value in the Header based on a custom clock

In some cases it is useful to set the timestamp of a ROS message based on the time published by a clock server than the ROS System time. A clock server is a specialized ROS node that publishes timestamp to `/clock` topic in the form of `rosgraph_msgs/Clock` message type. In order to enable this behavior for the Header Assignment block, set the `/use_sim_time` ROS parameter to `true`.

```
rosparam set /use_sim_time true
```

This configures the Header Assignment block to look for `/clock` topic on the ROS Network and update the timestamp of the ROS Message accordingly. If `/clock` topic is not being published, the timestamp will be zeros. Since the Header Assignment block updates during every sample hit, the accuracy of the timestamp always depends on the step-size of the solver. Smaller step-size values result in more accurate timestamp values. Run the model. You should see the time in `stamp` field of the ROS message based on the published `/clock` topic, not the current ROS System Time.

```
sim('rosHeaderAssignmentBlockExampleModel')
```
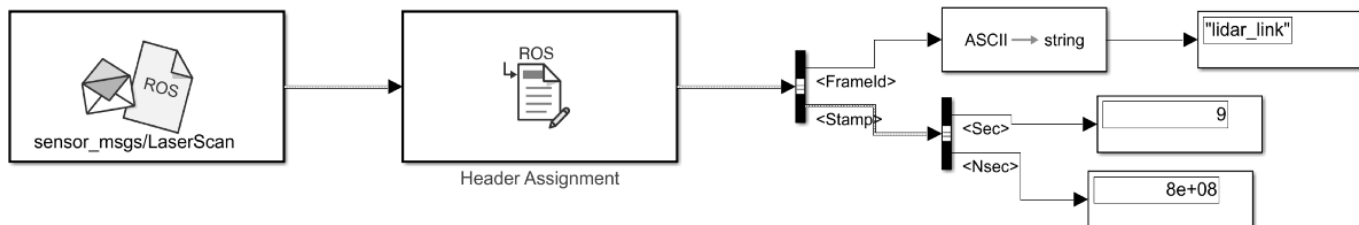

Copyright 2021 The MathWorks, Inc.

Shut down the ROS network.

```
rosshutdown
```

```
Shutting down global node /matlab_global_node_49543 with NodeURI http://bat6234win64:58249/ and N
Shutting down ROS master on http://172.30.131.134:52101.
```

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
```

```
Launching ROS Core...
...Done in 3.3416 seconds.
Initializing ROS master on http://172.30.131.134:50487.
Initializing global node /matlab_global_node_31479 with NodeURI http://bat6234win64:63593/ and M
```

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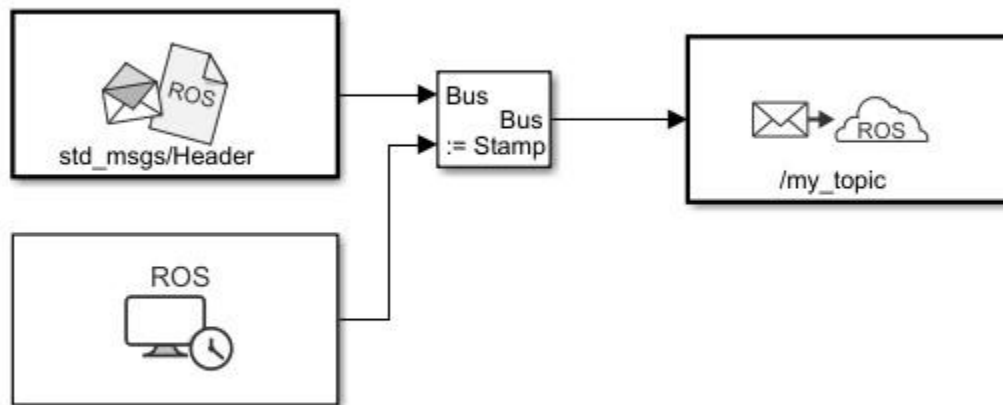
Run the model. The **Publish** block should publish the Header message with the current system time.

```
sim('current_time_stamp_message.slx')
```

Shut down the ROS network.

```
rosshutdown
```

```
Shutting down global node /matlab_global_node_31479 with NodeURI http://bat6234win64:63593/ and N
Shutting down ROS master on http://172.30.131.134:50487.
```

ROS Parameters in Simulink

In this section...
"Get and Set ROS Parameters" on page 6-12
"Set String Parameter on ROS Network" on page 6-13
"Compare ROS String Parameters" on page 6-14
"Check Image Encoding Parameter for ROS Image Message" on page 6-15

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.
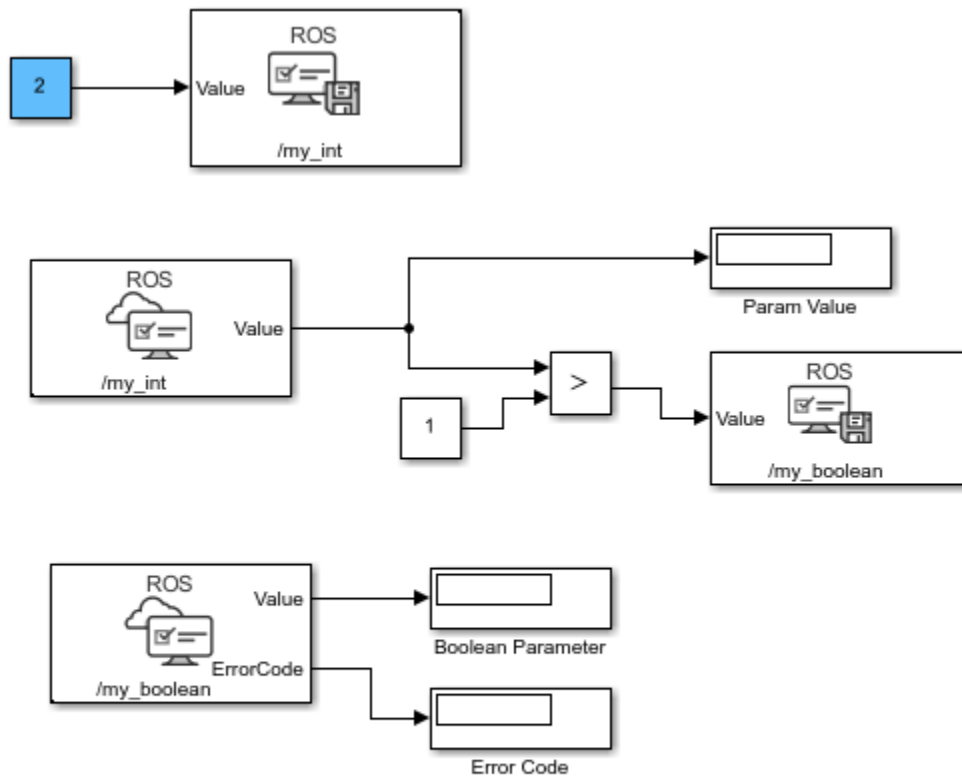
Copyright 2019-2020 The MathWorks, Inc.

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.

Copyright 2019 The MathWorks, Inc.
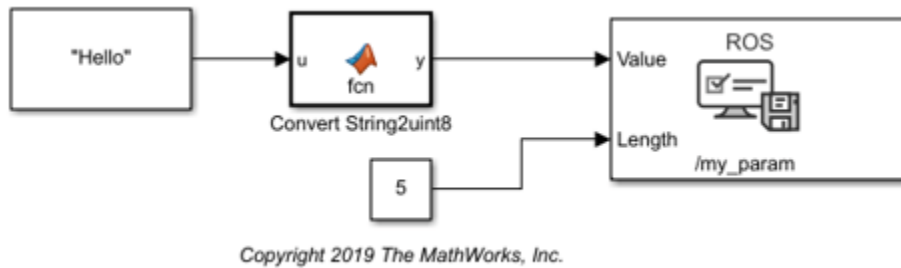
Compare ROS String Parameters

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 2.7038 seconds.
Initializing ROS master on http://172.30.131.134:57626.
Initializing global node /matlab_global_node_45597 with NodeURI http://bat6234win64:64828/ and Ma
```

```
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")
```

```
pvalue =
'jpeg'
```

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");
```
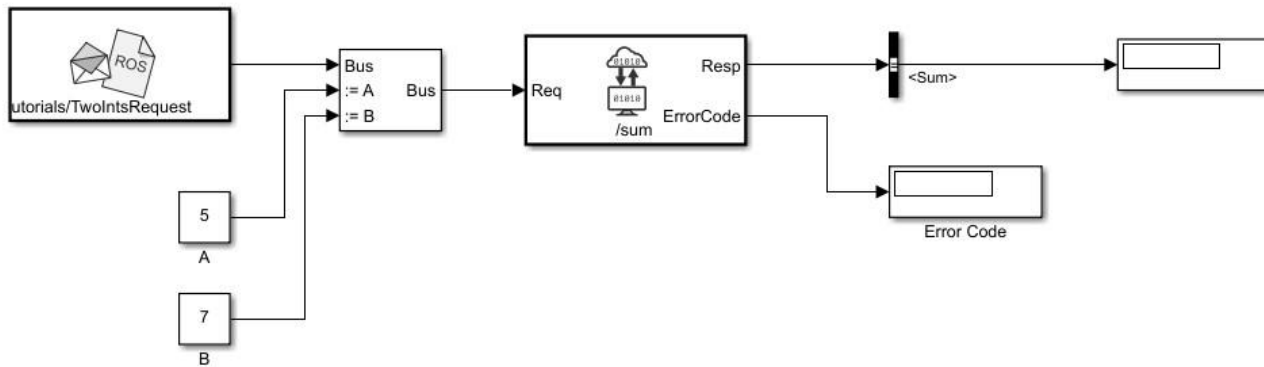
Shutdown ROS network.

```
rosshutdown
```

```
Shutting down global node /matlab_global_node_45597 with NodeURI http://bat6234win64:64828/ and N
Shutting down ROS master on http://172.30.131.134:57626.
```

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 2.6813 seconds.
Initializing ROS master on http://172.30.131.134:50240.
Initializing global node /matlab_global_node_76615 with NodeURI http://bat6234win64:51342/ and Ma
```

```
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")
```
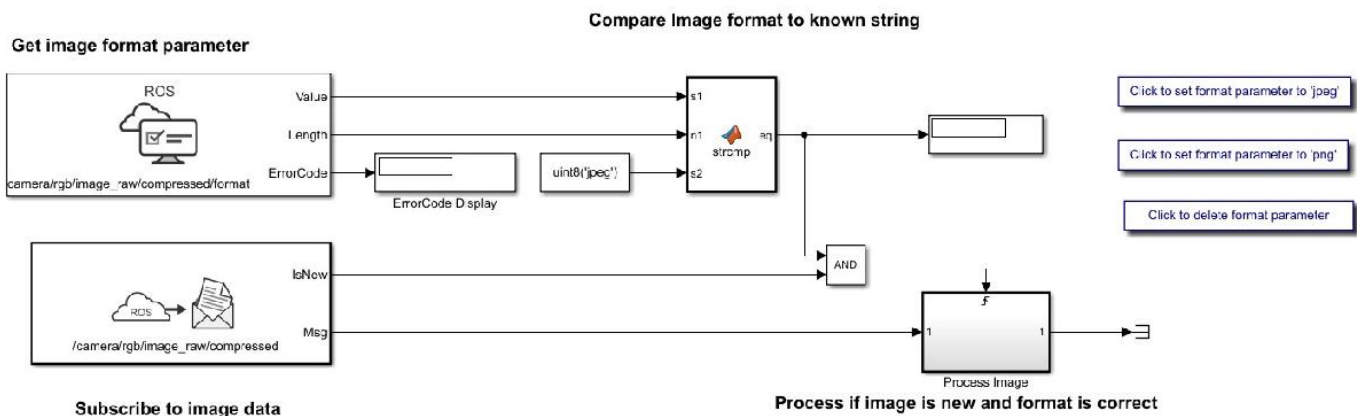

Shut down the ROS network.

```
rosshutdown
```

Shutting down global node /matlab_global_node_76615 with NodeURI http://bat6234win64:51342/ and Shutting down ROS master on http://172.30.131.134:50240.

The `strcmp` function in the MATLAB® Function block is defined as:

```
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.
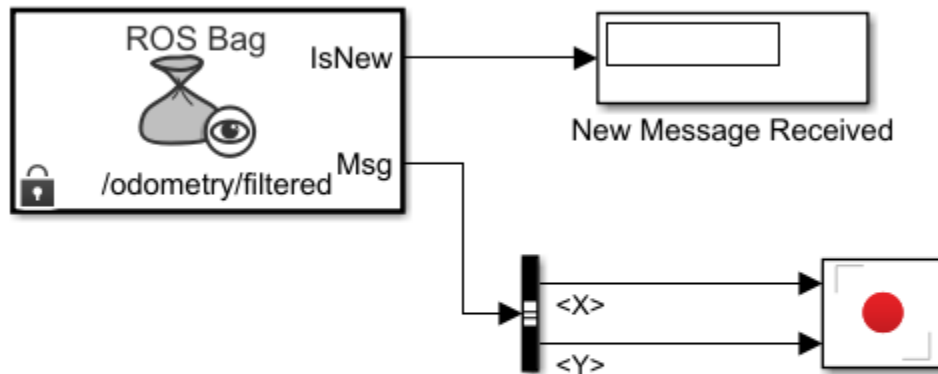
Copyright 2019 The MathWorks, Inc.

```
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 **Record** plot displays the robot position over time.

```
sim(gcs)
```
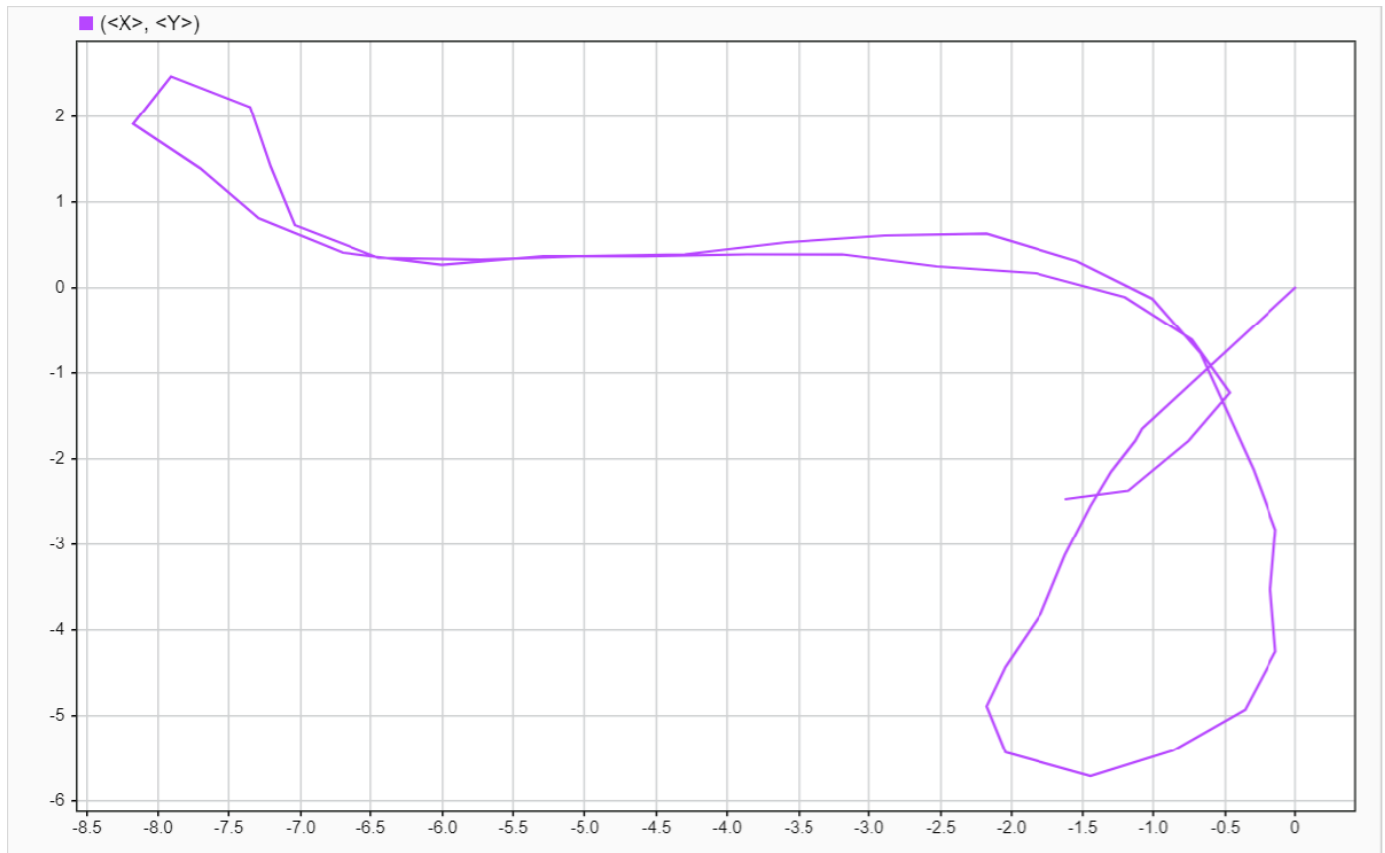
Call ROS Service in Simulink

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 2.1748 seconds.
Initializing ROS master on http://172.30.131.134:56078.
Initializing global node /matlab_global_node_02316 with NodeURI http://bat6234win64:55936/ and Ma
```

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')
```
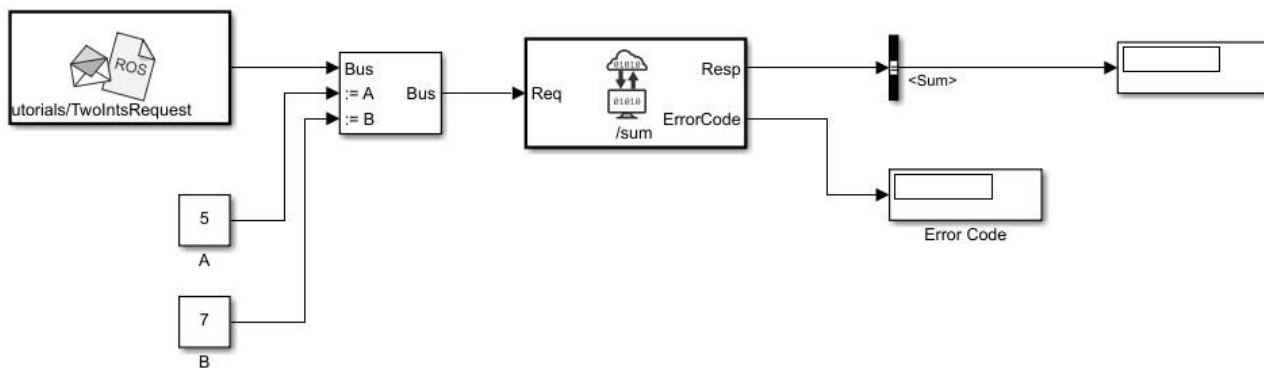

Run the model. The service call should return 0 in the Resp output as part of the response message. An error code of 0 indicates the service call was successful. You can ignore warnings about converting data types.

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

```
Warning: The property "A" in ROS message type "roscpp_tutorials/TwoIntsRequest" with datatype (in
Suggested Actions:
    • Remove conversion of ROS message properties with "int64" datatype to "double" in the Simul:
```

```
Warning: The property "B" in ROS message type "roscpp_tutorials/TwoIntsRequest" with datatype (in
Suggested Actions:
    • Remove conversion of ROS message properties with "int64" datatype to "double" in the Simul:
```

```
Warning: The property "Sum" in ROS message type "roscpp_tutorials/TwoIntsResponse" with datatype
Suggested Actions:
    • Remove conversion of ROS message properties with "int64" datatype to "double" in the Simuli
```

Shut down the ROS network to disconnect.

```
rosshutdown
```

```
Shutting down global node /matlab_global_node_02316 with NodeURI http://bat6234win64:55936/ and N
Shutting down ROS master on http://172.30.131.134:56078.
```

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 1-39

- "Connect to a ROS-enabled Robot from Simulink®" on page 1-141

More About

- "ROS Simulink Interaction" on page 6-4
- "Select ROS Topics, Messages, and Parameters" on page 6-25
- "ROS Simulink Support and Limitations" on page 6-2

Select ROS Topics, Messages, and Parameters

In this section...
"Select ROS Topics" on page 6-25
"Select ROS Message Types" on page 6-26
"Select ROS Parameter Names" on page 6-26

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:
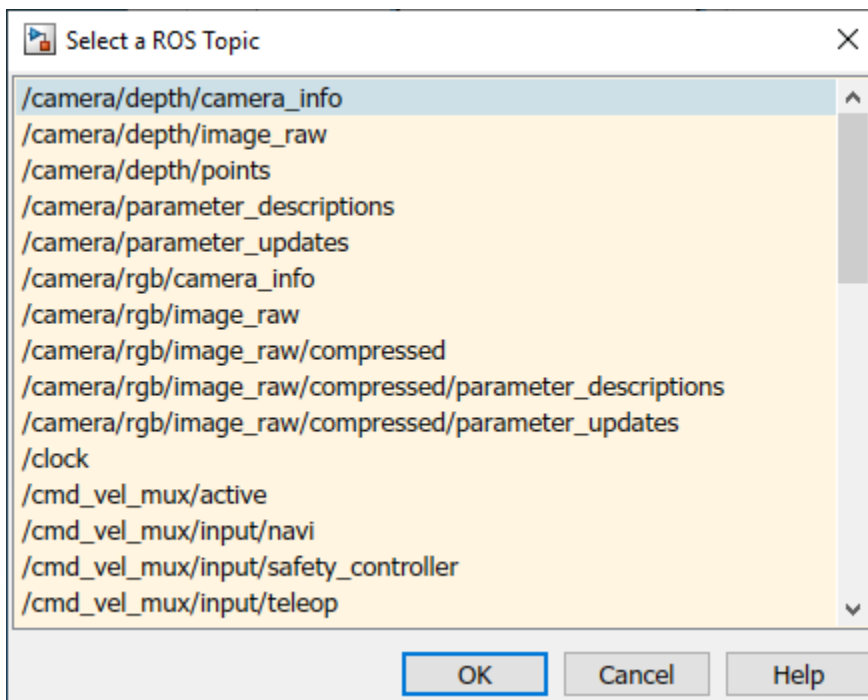
This dialog shows the list of topics available on the ROS master. Selecting a topic from the list automatically populates the **Topic** and **Message type** parameters for the corresponding block mask dialog. If the message type is not supported in MATLAB ROS, Simulink will throw an error. Once the topic 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.
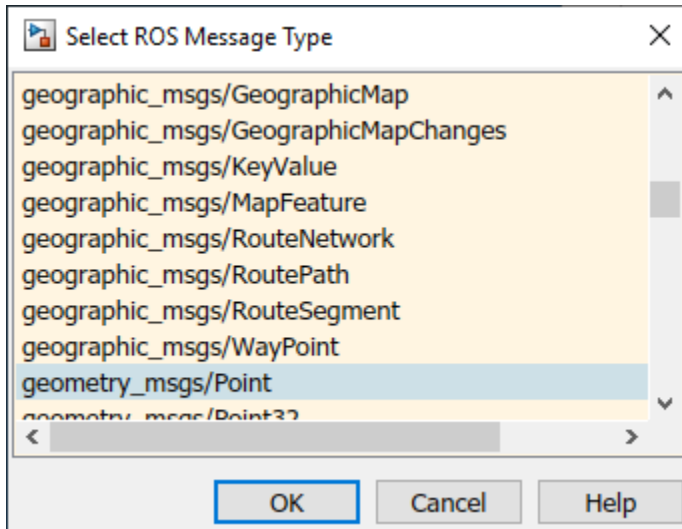
This is the list of all message types supported in MATLAB ROS including any custom message types. You can begin typing in the name of your desired message type or manually search through the list.

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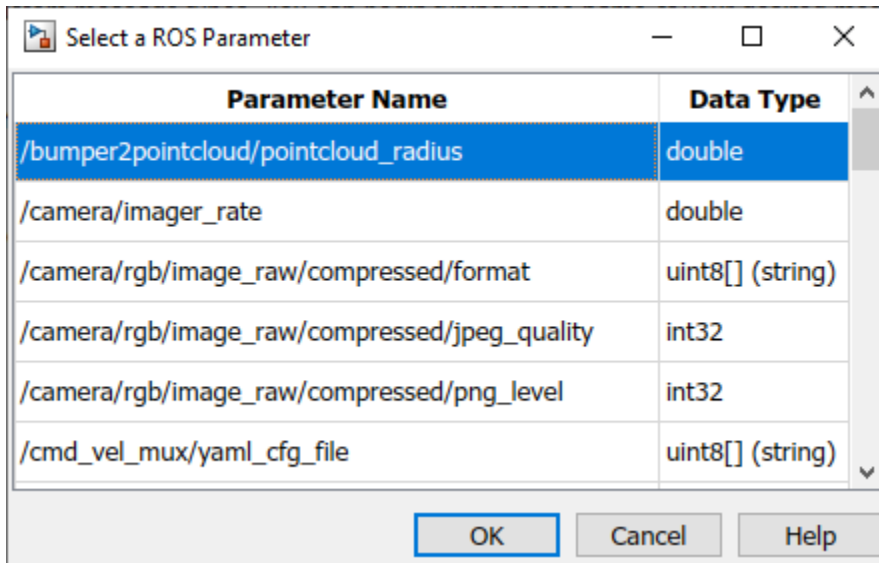
This is the list of parameters you can select from the ROS parameter server. The parameters that are grayed out have unsupported data types. Select a parameter name that is not grayed out and click **OK**. This should auto-fill the **Name** and **Data type** into the block parameters.

See Also

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

Related Examples

- "ROS Parameters in Simulink" on page 6-12
- "Manage Array Sizes for ROS Messages in Simulink" on page 6-28

Manage Array Sizes for ROS Messages in Simulink

A ROS message is represented as a bus. For more information on buses, see "Composite Interface Guidelines" (Simulink).

If you are working with variable-length signals in Simulink, the nonvirtual bus used for messages cannot contain variable-length arrays as properties. All variable-length arrays are converted to fixed-length arrays for nonvirtual buses. Therefore, you must manage the maximum size for these fixed-size arrays.

Ensure that your Simulink model is configured for ROS. If it not, follow these steps:

1   On the **Apps** tab, under **Control Systems**, click **Robot Operating System (ROS)**.
2   In the **Robot Operating System (ROS)** dialog box that opens up, select `Robot Operating System (ROS)` from the **ROS Network** drop-down. This opens up the **ROS** tab in the toolstrip which shows the specified ROS Network in the **Connect** section.

Then, 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.
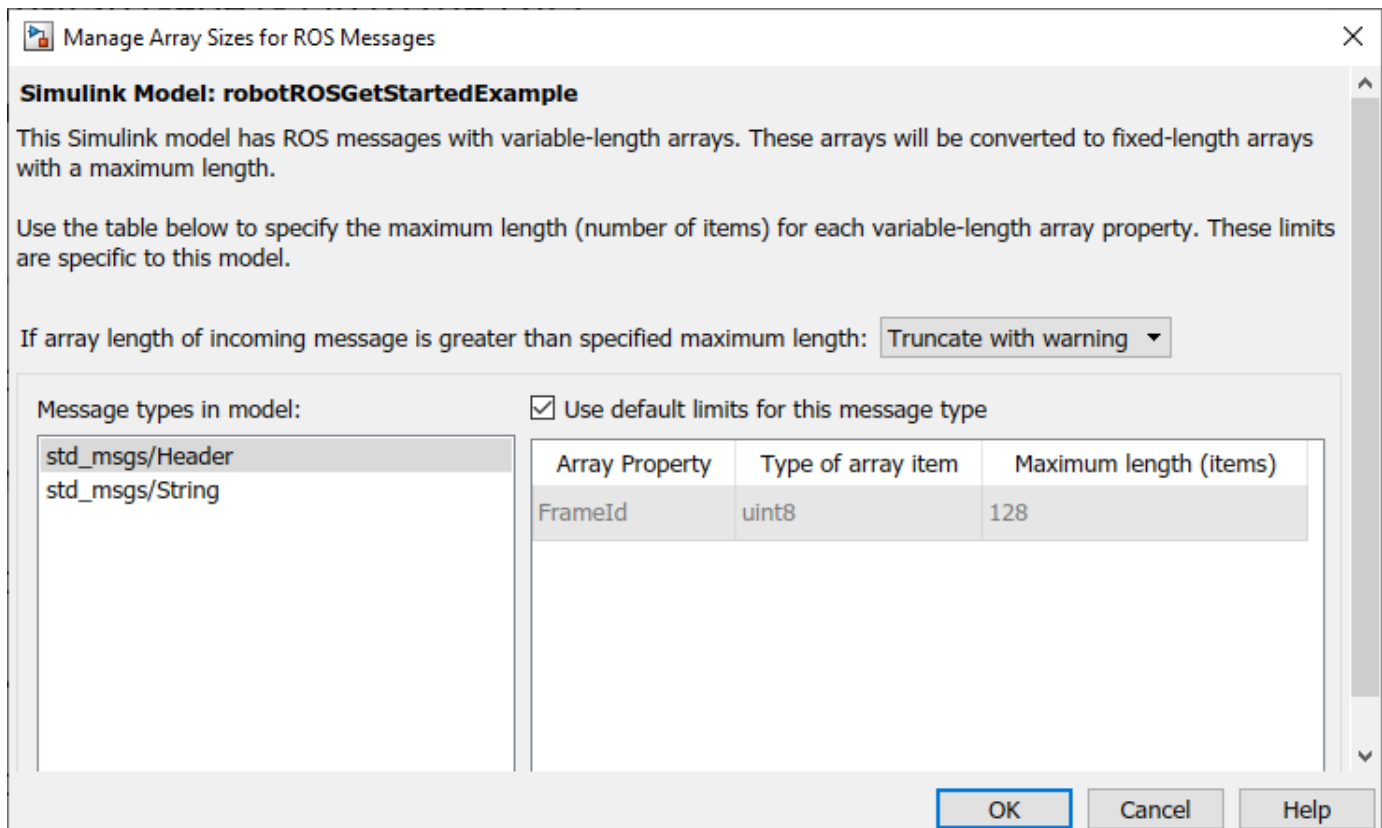
Because the message properties have a variable length, it is possible that they can be truncated if they exceed the maximum size set for that array. You have the option of `Truncate with warning` or `Truncate silently`. Either way, the simulation will run, but `Truncate with warning` displays a warning 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 1-125.

See Also
Publish | Subscribe

Related Examples
- "Get Started with ROS in Simulink" on page 1-125
- "ROS Simulink Support and Limitations" on page 6-2

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 1-195.

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 Simulated TurtleBot" on page 1-206 for instructions on how to install and use the virtual machine.
- Review the "Feedback Control of a ROS-Enabled Robot" on page 1-149 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 1-149 example.

Open the proportional controller model.

`edit robotROSFeedbackControlExample`

Copy the entire model to a new blank Simulink model.

Delete the Simulation Rate Control block.
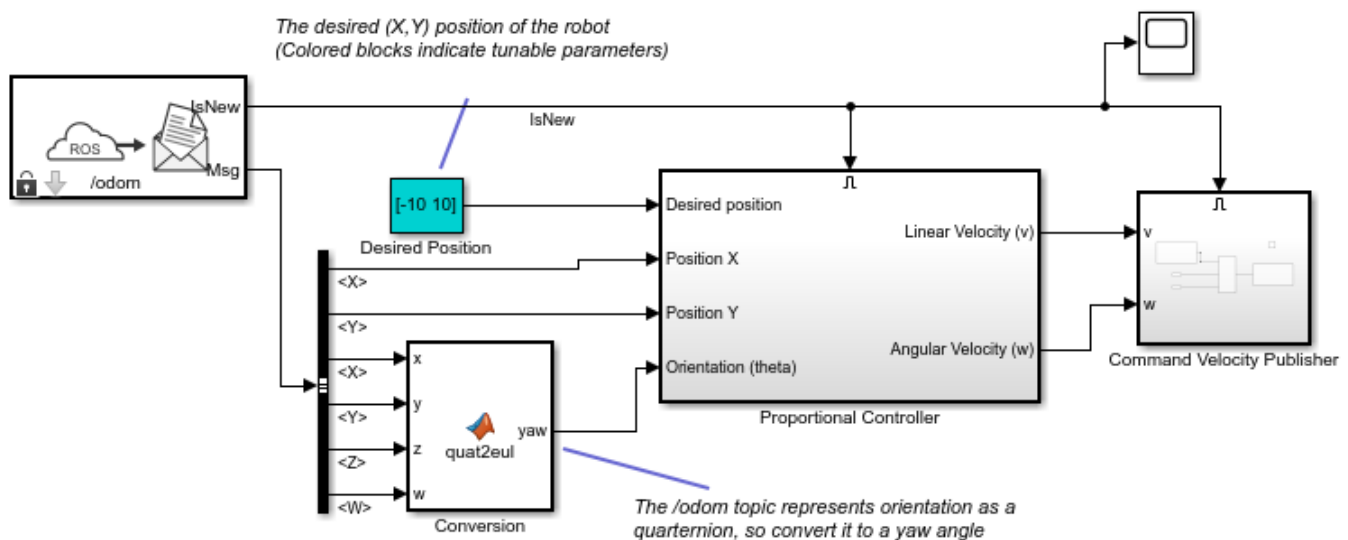
On the Apps tab, under **Control Systems**, click **Robot Operating System (ROS)**.

In the **Robot Operating System (ROS)** dialog box that opens up, select `Robot Operating System (ROS)` from the **ROS Network** drop-down. This opens up the **ROS** tab in the toolstrip which shows the specified ROS Network in the **Connect** section.

In the **Prepare** section under **ROS** tab, click **Hardware Settings** to open the model configuration parameters dialog box.

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.

Click **Deploy** under the **ROS** tab. Then under **Deployment**, click **Build Model**. This setting ensures that code is 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
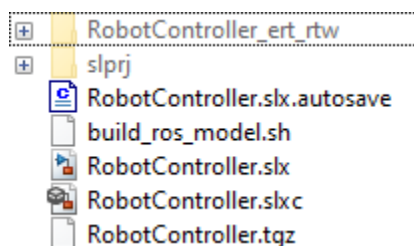
Manually transfer the two files to the target machine. If you connect to a ROS device using `rosdevice`, you can send files using `putFile`. Otherwise, this step assumes you are using the Linux virtual machine used for Robotics System Toolbox™ examples. The virtual machine is configured to accept SSH 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. Build the Catkin workspace.

```
mkdir -p ~/catkin_ws_simulink/src
cd ~/catkin_ws_simulink/src
catkin_init_workspace
cd ..
catkin_make
```

Decompress and build the node there using the following command in Linux. Replace `<path_to_catkin_ws>` with the path to your catkin workspace. If the node uses custom messages, you must manually copy the necessary custom message packages in `<path_to_catkin_ws>/msg` folder before building the node.

```
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 1-149
- "Generate a Standalone ROS Node from Simulink" on page 1-195
- "Tune Parameters and View Signals on Deployed Robot Models Using External Mode" on page 6-34

Tune Parameters and View Signals on Deployed Robot Models Using External Mode

In this section...
"Set Up the Simulink Model" on page 6-34
"Deploy and Run the Model" on page 6-34
"Monitor Signals and Tune Parameters" on page 6-35

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 1-149 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 **Prepare** section under **ROS** tab, click **Hardware Settings** to open the model configuration parameters dialog box.

2. On the **Solver** pane, set **Type** to `Fixed-step` and the **Fixed-step size** to `0.05`.

3. In **Hardware Implementation > Target Hardware Resources**, set the **External mode** parameters. To prioritize model execution speed, enable **Run external mode in a background thread**. Click **OK**.

In the model, add scope blocks to the signals you want to view. For this example, add an Record scope to the X and Y signals that are exiting the Bus Selector from the ROS subscriber that monitors the robot position. Open the Record 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`.

Under **ROS** tab, click **Monitor And Tune**. 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 Record 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 1-149
- "Enable External Mode for ROS Toolbox Models" on page 6-38
- "Generate a Standalone ROS Node from Simulink" on page 1-195

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 1-195

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 1-195.

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. On the Apps tab, under **Control Systems**, click **Robot Operating System (ROS)**.
2. In the **Robot Operating System (ROS)** dialog box that opens up, select `Robot Operating System (ROS)` from the **ROS Network** drop-down. This opens up the **ROS** tab in the toolstrip which shows the specified ROS Network in the **Connect** section.
3. In the **Prepare** section under **ROS** tab, click **Hardware Settings** to open the model configuration parameters dialog box. Under **Target Hardware resources > ROS time**, select **Enable ROS time model stepping**.

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

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

You can also enable overrun detection if the model execution is still processing when the next step is triggered by the ROS time. For more information, see "Overrun Detection with Deployed ROS Nodes" on page 6-39.

See Also
Current Time | Subscribe

Related Examples
- "Generate a Standalone ROS Node from Simulink" on page 1-195
- "Overrun Detection with Deployed ROS Nodes" on page 6-39
- "Get Started with ROS in Simulink" on page 1-125
- "Exchange Data with ROS Publishers and Subscribers" on page 1-59

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   On the Apps tab, under **Control Systems**, click **Robot Operating System (ROS)**.
2   In the **Robot Operating System (ROS)** dialog box that opens up, select `Robot Operating System (ROS)` from the **ROS Network** drop-down. This opens up the **ROS** tab in the toolstrip which shows the specified ROS Network in the **Connect** section.
3   In the **Connect** section, specify **Deploy To** option as `Remote Device`, from the drop-down. To configure the remote device details such as IP address and user details, select `Manage Remote Device` from the drop-down.
4   In the **Prepare** section, click **Hardware Settings** to open the model configuration parameters dialog box. In **Target Hardware Resources**, set the **External mode** parameters. Click **OK**.
5   To build and deploy the model in external mode, under **ROS** tab, click **Monitor And Tune**.

Your model connects to the **Device Address** specified in the "Connect to ROS Device" on page 6-36 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, Signal Monitoring, and Code Execution Profiling" (Simulink Coder).

See Also

Related Examples
- "Generate a Standalone ROS Node from Simulink" on page 1-195
- "Tune Parameters and View Signals on Deployed Robot Models Using External Mode" on page 6-34
- "External Mode Simulations for Parameter Tuning, Signal Monitoring, and Code Execution Profiling" (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 1-195.

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 On the Apps tab, under **Control Systems**, click **Robot Operating System (ROS)**.
2 In the **Robot Operating System (ROS)** dialog box that opens up, select Robot Operating System (ROS) from the **ROS Network** drop-down. This opens up the **ROS** tab in the toolstrip which shows the specified ROS Network in the **Connect** section.
3 In the **Prepare** section under **ROS** tab, click **Hardware Settings** to open the model configuration parameters dialog box. 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
- "Generate a Standalone ROS Node from Simulink" on page 1-195
- "Enable ROS Time Model Stepping for Deployed ROS Nodes" on page 6-37
- "Get Started with ROS in Simulink" on page 1-125
- "Exchange Data with ROS Publishers and Subscribers" on page 1-59

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 transformation. 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. Use struct message format for better efficiency.

```
rosinit
```

```
Launching ROS Core...
..Done in 2.6971 seconds.
Initializing ROS master on http://172.30.131.134:59968.
Initializing global node /matlab_global_node_55551 with NodeURI http://bat6234win64:56786/ and Ma
```

```
[pub,msg] = rospublisher("/pose","geometry_msgs/Pose","DataFormat","struct");
```

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.

-2.22e-16	1	0	4
1	-2.22e-16	0	5
0	0	-1	6
0	0	0	1

Homogeneous Transformation
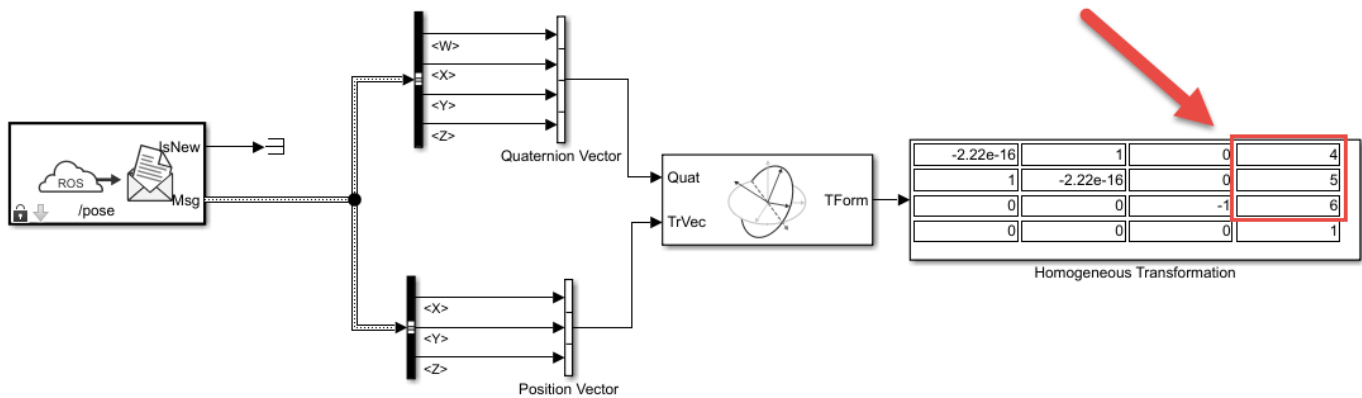
Copyright 2018 The MathWorks, Inc.

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_55551 with NodeURI http://bat6234win64:56786/ and
Shutting down ROS master on http://172.30.131.134:59968.

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')
```
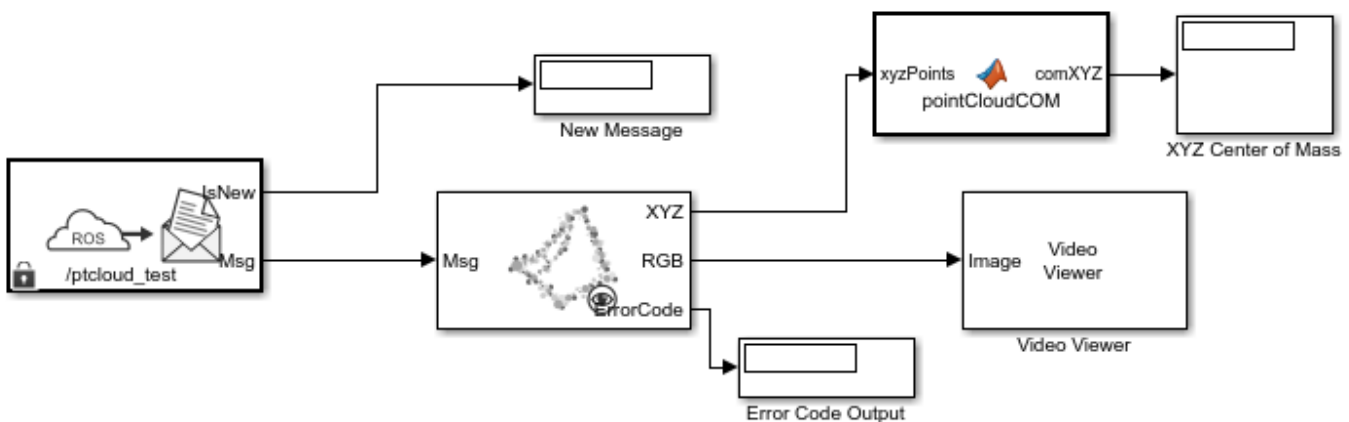

Copyright 2018 The MathWorks, Inc.
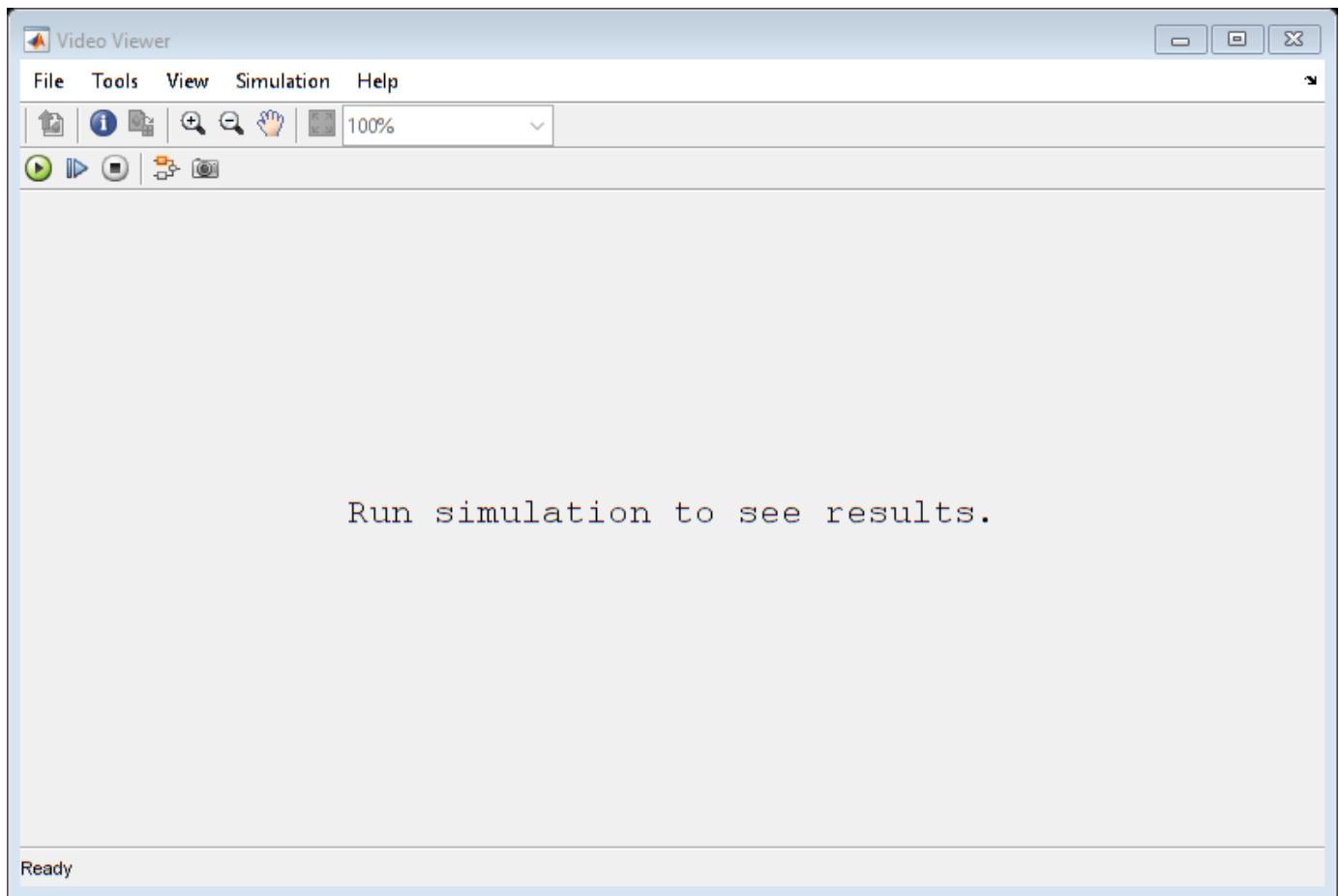
Run the model. The `Video Viewer` shows the sample point cloud as an image. The output center of mass is `[-0.2869 -0.0805 2.232]` for this point cloud.
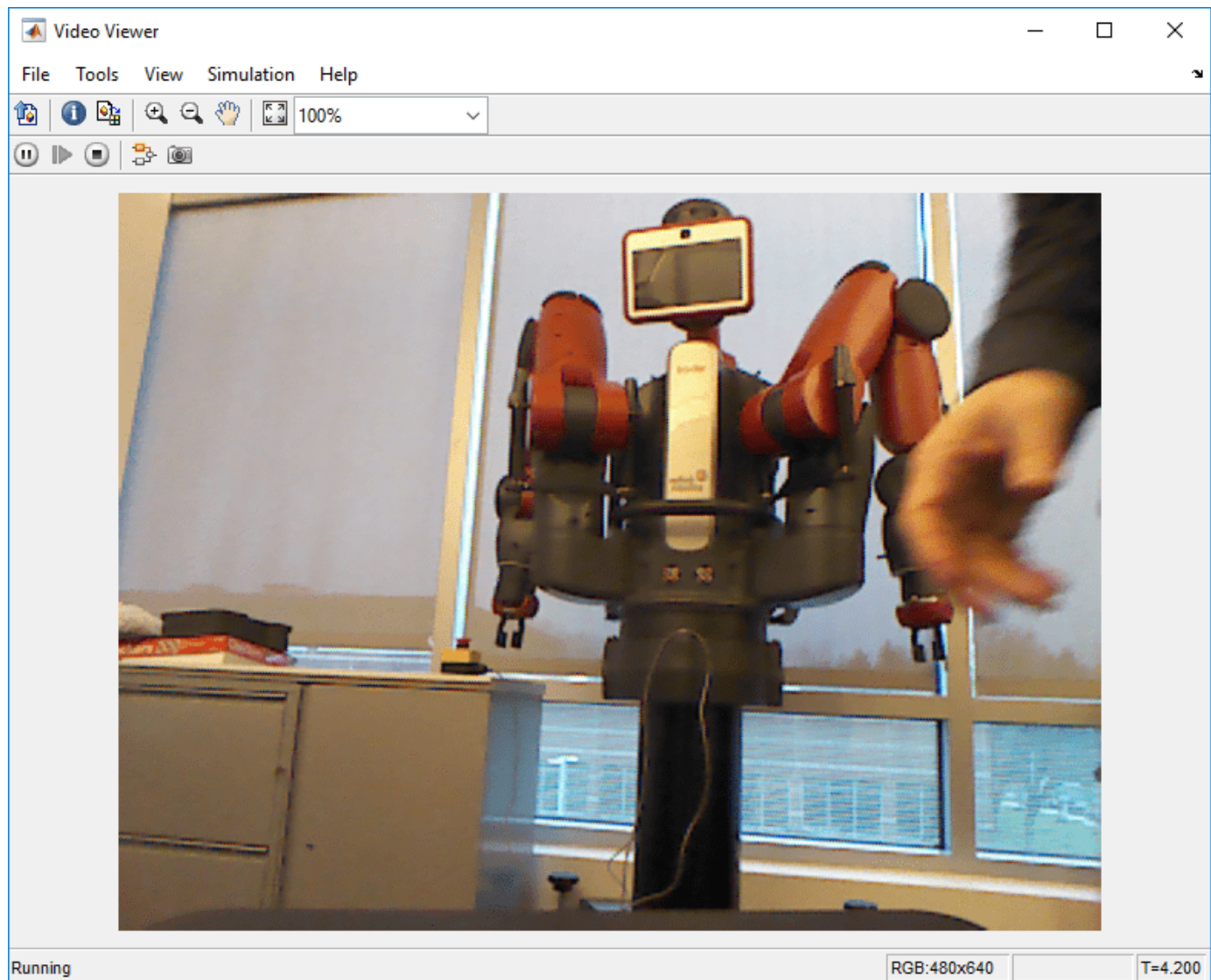
Stop the simulation and shut down the ROS network.

```
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);
```

```
% 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

Start a ROS network.

```
rosinit
```

```
Launching ROS Core...
....Done in 4.236 seconds.
Initializing ROS master on http://172.21.16.85:53224.
Initializing global node /matlab_global_node_92597 with NodeURI http://ah-avijayar:61421/ and Mas
```

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. Under **Simulation** tab, select **ROS Toolbox > Variable Size Messages**. Notice that the **Use default limits for this message type** is clear. Then, in the **Maximum length** column, verify that the `data` property of the `sensor_msgs/Image` message has a value greater than the number of pixels in the image (921600). Run the model.

```
open_system('read_image_example_model.slx')
```
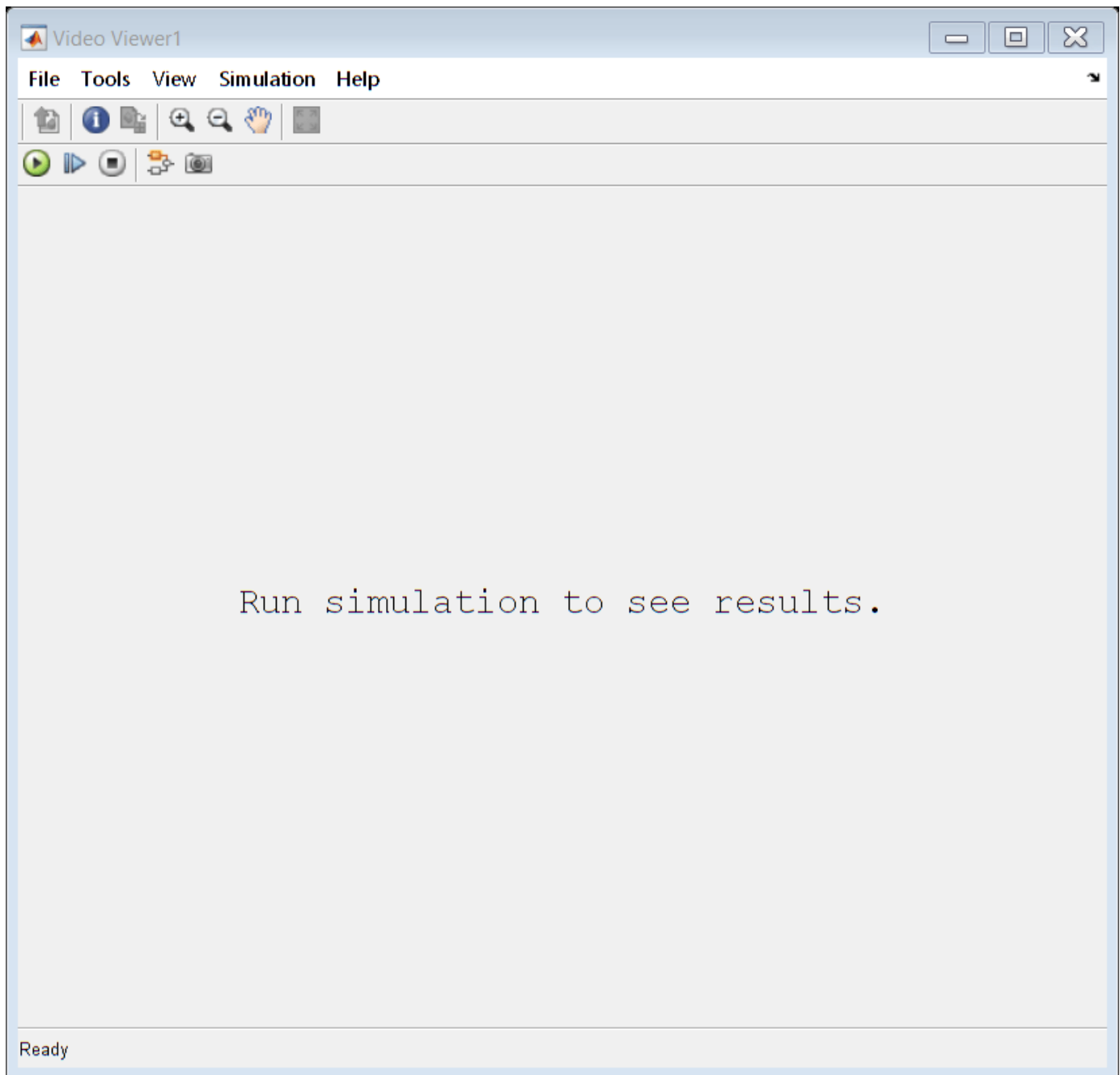
Run the model. The `Video Viewer` shows the sample image.
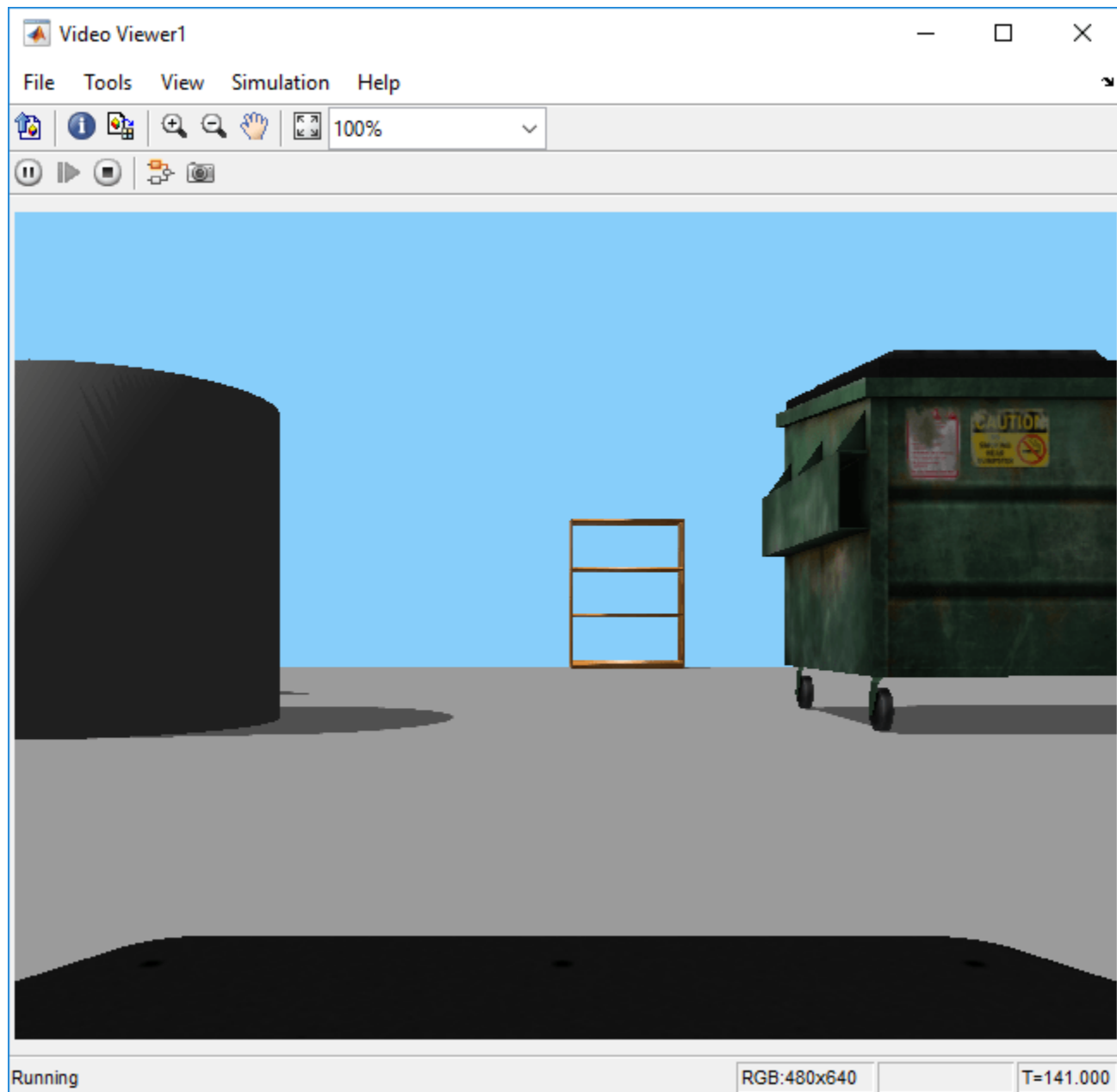
Stop the simulation and shut down the ROS network.

`rosshutdown`

```
Shutting down global node /matlab_global_node_92597 with NodeURI http://ah-avijayar:61421/ and Ma
Shutting down ROS master on http://172.21.16.85:53224.
```

Generate a ROS Control Plugin from Simulink®

This example shows how to generate and build a ros_control plugin from a Simulink model. In this example, you first configure a model to generate C++ code for a ros_control package. You then deploy the plugin on a virtual machine running Gazebo® to control a Pioneer 3DX 2-wheeled differential drive robot.

Prerequisites

*   This example requires Simulink Coder™.
*   Download a virtual machine using instructions in "Get Started with Gazebo and Simulated TurtleBot" on page 1-206
*   Review the "Feedback Control of a ROS-Enabled Robot" on page 1-149 and "Generate a Standalone ROS Node from Simulink" on page 1-195 example.
*   To ensure you have the proper third-party software, see "ROS Toolbox System Requirements".

Overview

The ros_control framework provides a generic set of tools to create controllers for various types of robots.

This example focuses on auto-generating C++ code for a controller designed in Simulink® and packaging it and deploying that controller as a ros_control plugin as shown in the figure below. In this example, you generate code for a controller designed in Simulink, and deploy it as a ros_control plugin. The data flow of the controller is shown in the following diagram.
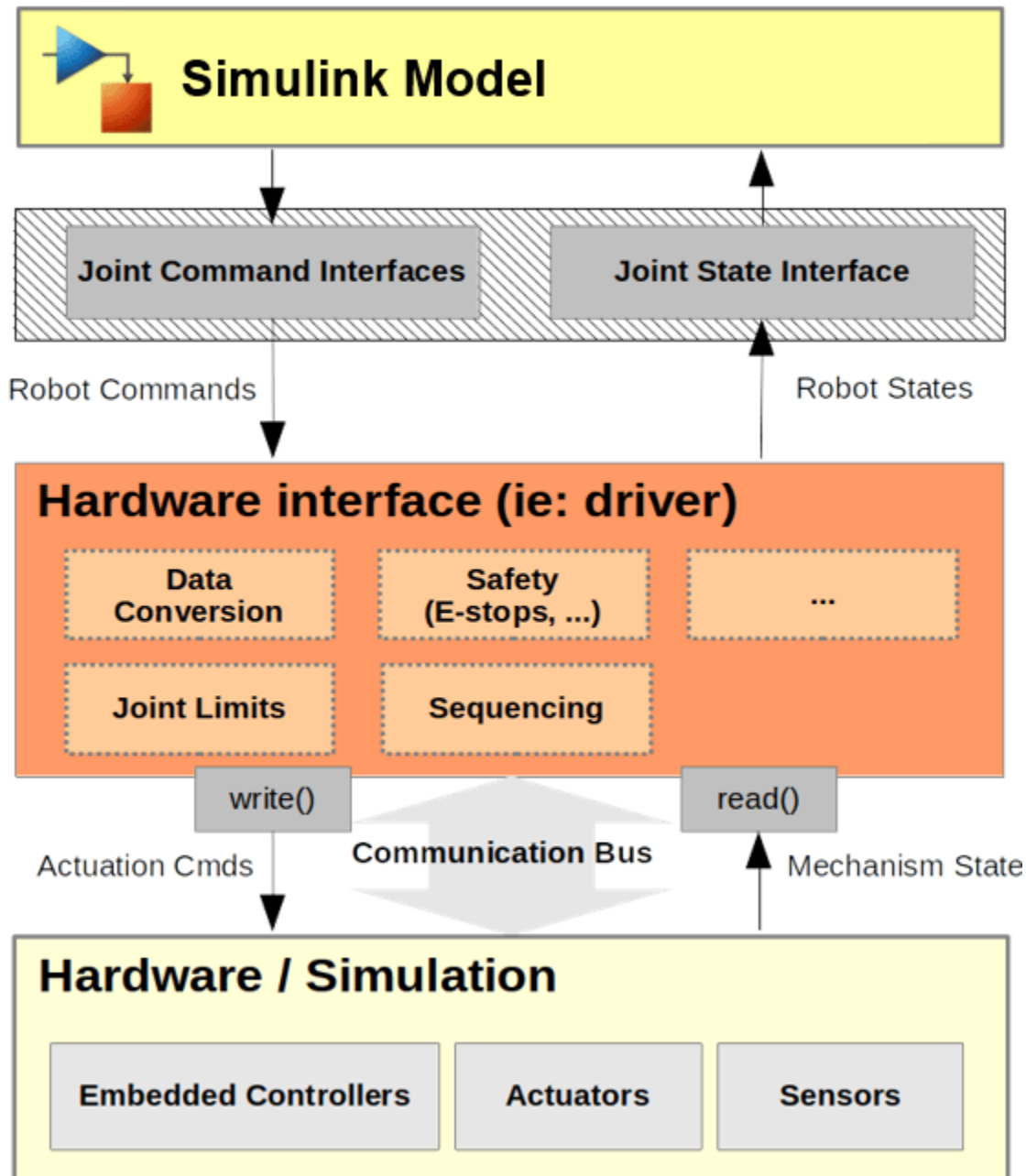
The Simulink model drivecontroller implements a simple closed-loop proportional controller. It receives the goal location from `/command` topic of type, `geometry_msgs/Pose2D`, and the current position of the robot from `/p3dx/ground_truth/state` topic of type, `nav_msgs/Odometry`. The controller sets the left and right wheel speed commands using `VelocityJointInterface`. This topology is shown in the diagram below.
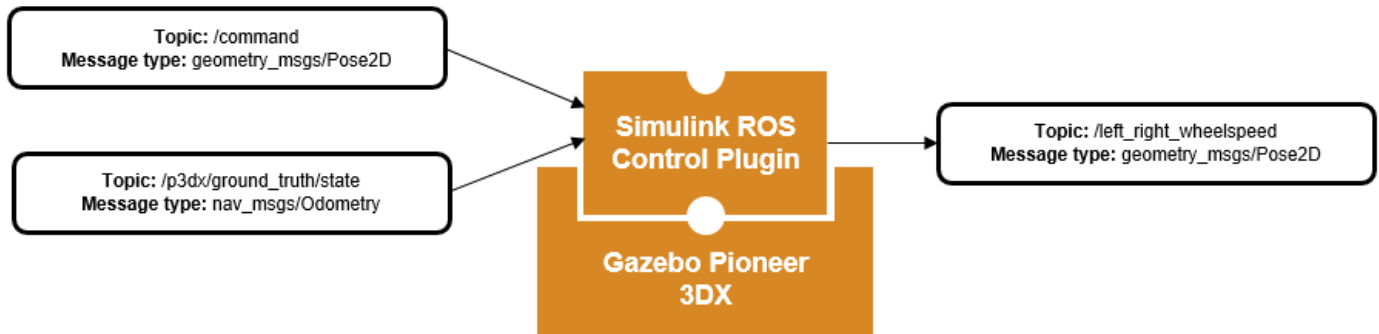
Open Controller Simulink Model

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

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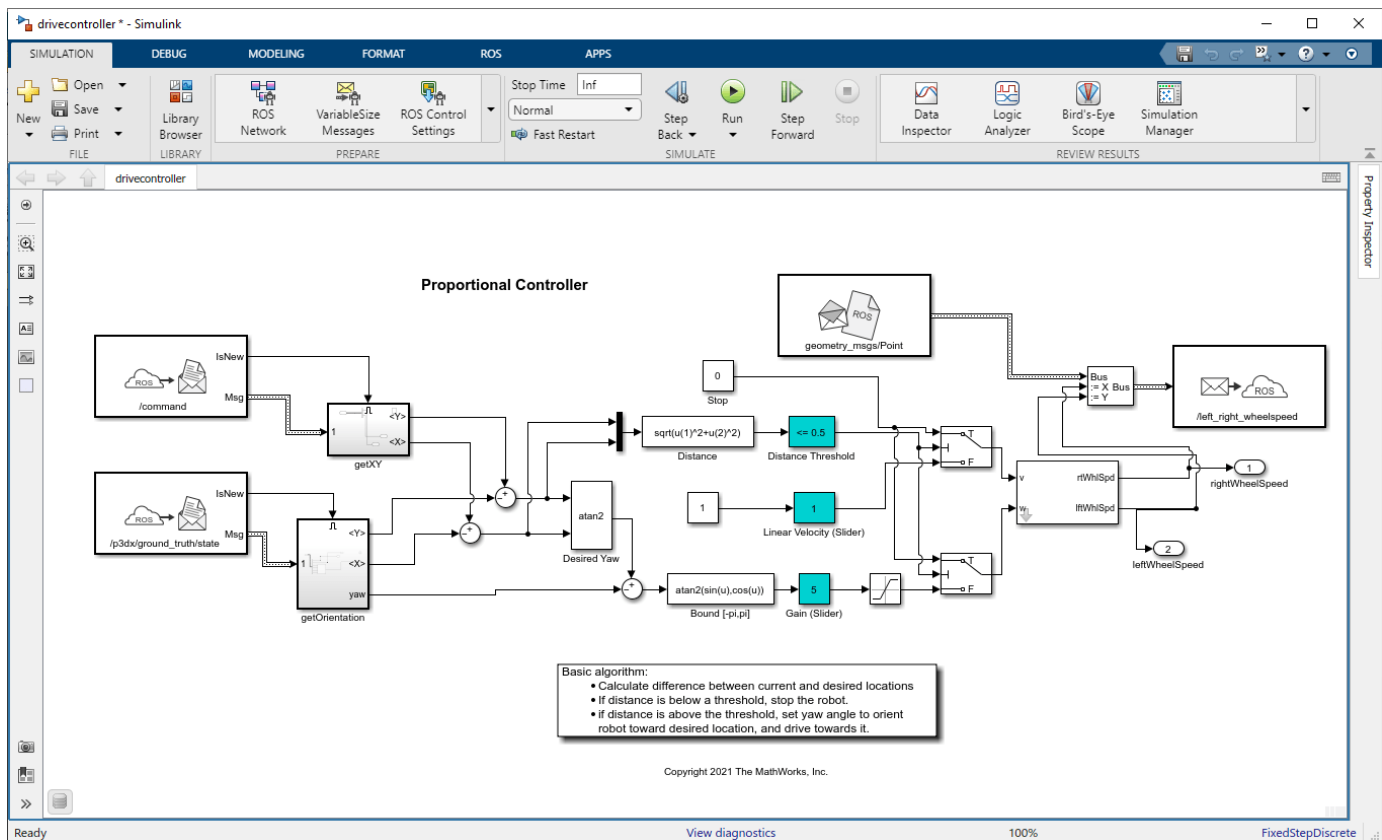

The model implements a proportional controller. At each step, the algorithm orients the robot toward the desired location and drives it forward. Once the desired location is reached, the algorithm stops the robot.

Start ROS Core

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

- ros_control interface setting prepares the model for simulation to ensure that all blocks are properly initialized. This requires a valid connection to a ROS core.
- Start the ROS core on the virtual machine using the `rosdevice` object

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

Another roscore / ROS master is already running on the ROS device. Use the 'stopCore' function to

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

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

Initializing global node /matlab_global_node_69673 with NodeURI http://192.168.93.1:49973/ and Ma

Configure ROS Control Interfaces

The example controller model is configured for Robot Operating System (ROS). Use the ros_control Settings button to map the output ports of the model to correct resource type.

In **ROS** tab, under **PREPARE** section, click on **ROS Control Settings** button from the **ROS Toolbox** category.

In the **Configure ROS Control** dialog:

1. Set the name of the **Controller C++ class** to `ControllerHost`
2. In the **Outports** tab, set the **Resource name** of the output port 1, to `base_right_wheel_joint` and set the name of the output port 2, to `base_left_wheel_joint`.
3. Set the **Resource type** for both output ports to `VelocityJointInterface`.
4. From the **Parameters** tab in the dialog, enable **Gain (Slider)**, **Linear Velocity (Slider)** and **Distance Threshold** for dynamic reconfigure.
5. Select **Generate ros_control controller package** checkbox.
6. Click **OK** to save the settings and close the dialog.

The dialog settings are shown below for reference.

Configure ROS Control — □ ×

To generate ROS Control controller package from the Simulink model, 'drivecontroller', specify the settings below.

Controller C++ class name: ControllerHost

| Inports | Outports | **Parameters** |

Source	Path	Name	Enable Dynamic Reconfigure
Gain	drivecontroller/Subsystem	gain	☐
Gain1	drivecontroller/Subsystem	gain	☐
Gain (Slider)	drivecontroller	gain	☑
Linear Velocity (Slider)	drivecontroller	gain	☑
Constant	drivecontroller	Value	☐
Stop	drivecontroller	Value	☐
TrackWidth	drivecontroller/Subsystem	Value	☐
Distance Threshold	drivecontroller	const	☑
Saturation	drivecontroller	UpperLimit	☐
Saturation	drivecontroller	LowerLimit	☐

☑ Generate ros_control controller package

OK Cancel

Configure Connection to the ROS Device

1 Under the **ROS** tab, from the **Deploy to** drop-down, click **Manage Remote Device**.

2 In the **Connect to a ROS device** dialog that opens, enter all the information that Simulink needs to deploy the ROS node.

3 Specify IP address or host name of your ROS device, login credentials, and the Catkin workspace.

4 Change **Catkin workspace** to ~/Documents/mw_catkin_ws. This workspace has a Pioneer 3DX Gazebo® world and related files to run a simulated differential drive robot.

Deploy the ROS Control Plugin

Generate C++ code for the proportional controller and create a ros_control plugin package, automatically transfer, and build it on the ROS device.

Deploy the ROS Control Plugin

- Configure Simulink to use the same ROS connection settings as MATLAB. Under the **Simulation** tab, in **Prepare** section, select **ROS Network**. Set the ROS Master (ROS 1) and Node Host network addresses to **Default**.
- Under the **ROS** tab, click **Deploy > Build & Load**. 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 package is transferred to the Catkin workspace on the ROS device and built there.

Integrate Simulink ROS Control Plugin with Gazebo

Register Controller and Build Workspace

Modify the Pioneer 3DX controller configuration YAML file to register the generated controller, and build the workspace.

- Replace the controller configuration YAML file using the `rosdevice` object. To use the controller plugin generated from the model, copy the pre-configured `pioneer3dx.yaml` file to the catkin workspace.

```
system(d,['cd ',d.CatkinWorkspace,' && cp src/pioneer_p3dx_model/p3dx_control/config/pioneer3dx.y
```

```
ans =

  0×0 empty char array
```

```
putFile(d,fullfile(pwd,'pioneer3dx.yaml'),[d.CatkinWorkspace,'/src/pioneer_p3dx_model/p3dx_contro
```

- Build the workspace

```
system(d,['cd ',d.CatkinWorkspace,' && source ./devel/setup.bash && catkin_make'])
```

```
ans =
    '[  0%] Built target sensor_msgs_generate_messages_eus
     [  1%] Generating dynamic reconfigure files from cfg/controller_params.cfg: /home/user/Docum
     [  1%] Built target std_msgs_generate_messages_nodejs
     [  1%] Built target std_msgs_generate_messages_py
     [  1%] Built target roscpp_generate_messages_cpp
     [  1%] Built target rosgraph_msgs_generate_messages_nodejs
     [  1%] Built target roscpp_generate_messages_eus
     Generating reconfiguration files for ControllerHost in drivecontroller
     Wrote header file in /home/user/Documents/mw_catkin_ws/devel/include/drivecontroller/Contro
     [  1%] Built target std_msgs_generate_messages_lisp
     [  1%] Built target roscpp_generate_messages_py
     [  1%] Built target std_msgs_generate_messages_eus
     [  1%] Built target roscpp_generate_messages_nodejs
     [  1%] Built target drivecontroller_gencfg
     [  1%] Built target std_msgs_generate_messages_cpp
     [  1%] Built target rosgraph_msgs_generate_messages_eus
     [  1%] Built target rosgraph_msgs_generate_messages_lisp
     [  1%] Built target roscpp_generate_messages_lisp
     [  1%] Built target rosgraph_msgs_generate_messages_py
     [  1%] Built target tf2_msgs_generate_messages_nodejs
     [  1%] Built target tf2_msgs_generate_messages_lisp
     [  1%] Built target tf2_msgs_generate_messages_eus
     [  1%] Built target actionlib_generate_messages_nodejs
     [  1%] Built target actionlib_generate_messages_lisp
     [  1%] Built target sensor_msgs_generate_messages_nodejs
     [  1%] Built target geometry_msgs_generate_messages_nodejs
     [  1%] Built target tf2_msgs_generate_messages_py
     [  1%] Built target geometry_msgs_generate_messages_cpp
     [  1%] Built target dynamic_reconfigure_generate_messages_cpp
     [  1%] Built target tf_generate_messages_lisp
     [  1%] Built target nav_msgs_generate_messages_cpp
     [  1%] Built target sensor_msgs_generate_messages_lisp
     [  1%] Built target nav_msgs_generate_messages_lisp
     [  1%] Built target geometry_msgs_generate_messages_eus
     [  1%] Built target dynamic_reconfigure_generate_messages_eus
     [  1%] Built target dynamic_reconfigure_gencfg
     [  1%] Built target dynamic_reconfigure_generate_messages_lisp
     [  1%] Built target actionlib_msgs_generate_messages_lisp
     [  1%] Built target dynamic_reconfigure_generate_messages_py
     [  1%] Built target nav_msgs_generate_messages_eus
     [  1%] Built target nav_msgs_generate_messages_nodejs
     [  1%] Built target actionlib_msgs_generate_messages_eus
     [  1%] Built target sensor_msgs_generate_messages_py
     [  1%] Built target geometry_msgs_generate_messages_py
     [  1%] Built target nav_msgs_generate_messages_py
     [  1%] Built target actionlib_generate_messages_eus
     [  1%] Built target actionlib_msgs_generate_messages_cpp
     [  1%] Built target rosgraph_msgs_generate_messages_cpp
```

```
[  1%] Built target actionlib_generate_messages_py
[  1%] Built target actionlib_msgs_generate_messages_nodejs
[  1%] Built target geometry_msgs_generate_messages_lisp
[  1%] Built target actionlib_generate_messages_cpp
[  1%] Built target actionlib_msgs_generate_messages_py
[  1%] Built target tf2_msgs_generate_messages_cpp
[  1%] Built target tf_generate_messages_eus
[  1%] Built target tf_generate_messages_py
[  1%] Built target tf_generate_messages_nodejs
[  1%] Built target dynamic_reconfigure_generate_messages_nodejs
[  1%] Built target sensor_msgs_generate_messages_cpp
[  2%] Generating dynamic reconfigure files from cfg/controller_params.cfg: /home/user/Docum
[  3%] Generating dynamic reconfigure files from cfg/controller_params.cfg: /home/user/Docum
Generating reconfiguration files for ControllerHost in drvctrl_pub
Wrote header file in /home/user/Documents/mw_catkin_ws/devel/include/drvctrl_pub/ControllerH
[  3%] Built target drvctrl_pub_gencfg
Generating reconfiguration files for ControllerHost in shape_tracer
Wrote header file in /home/user/Documents/mw_catkin_ws/devel/include/shape_tracer/Controller
[  3%] Built target _turtlebot_actions_generate_messages_check_deps_TurtlebotMoveActionFeedb
[  3%] Built target shape_tracer_gencfg
[  3%] Built target tf_generate_messages_cpp
[  3%] Built target _turtlebot_actions_generate_messages_check_deps_TurtlebotMoveGoal
[  3%] Built target _turtlebot_actions_generate_messages_check_deps_FindFiducialFeedback
[  3%] Built target _turtlebot_actions_generate_messages_check_deps_FindFiducialActionResul
[  3%] Built target _turtlebot_actions_generate_messages_check_deps_FindFiducialGoal
[  3%] Built target _turtlebot_actions_generate_messages_check_deps_TurtlebotMoveAction
[  3%] Built target _turtlebot_actions_generate_messages_check_deps_FindFiducialAction
[  3%] Built target _turtlebot_actions_generate_messages_check_deps_TurtlebotMoveActionResul
[  3%] Built target _turtlebot_actions_generate_messages_check_deps_TurtlebotMoveActionGoal
[  3%] Built target _turtlebot_actions_generate_messages_check_deps_FindFiducialActionGoal
[  3%] Built target _turtlebot_actions_generate_messages_check_deps_TurtlebotMoveResult
[  3%] Built target _turtlebot_actions_generate_messages_check_deps_TurtlebotMoveFeedback
[  3%] Built target _turtlebot_actions_generate_messages_check_deps_FindFiducialActionFeedba
Scanning dependencies of target drvctrl_pub
Scanning dependencies of target drivecontroller
[  4%] Building CXX object drvctrl_pub/CMakeFiles/drvctrl_pub.dir/src/drvctrl_pub_ctrlr_host
Scanning dependencies of target shape_tracer
[  5%] Building CXX object drivecontroller/CMakeFiles/drivecontroller.dir/src/drivecontrolle
[  5%] Built target _turtlebot_actions_generate_messages_check_deps_FindFiducialResult
[  6%] Building CXX object shape_tracer/CMakeFiles/shape_tracer.dir/src/shape_tracer_ctrlr_
[ 19%] Built target turtlebot_actions_generate_messages_lisp
[ 31%] Built target turtlebot_actions_generate_messages_nodejs
[ 45%] Built target turtlebot_actions_generate_messages_py
[ 58%] Built target turtlebot_actions_generate_messages_cpp
[ 71%] Built target turtlebot_actions_generate_messages_eus
[ 71%] Built target turtlebot_actions_gencpp
[ 71%] Built target turtlebot_actions_generate_messages
[ 73%] Built target turtlebot_move_action_server
[ 76%] Built target find_fiducial_pose
[ 77%] Linking CXX shared library /home/user/Documents/mw_catkin_ws/devel/lib/libdrvctrl_pub
/usr/bin/c++ -fPIC -O3 -DNDEBUG  -shared -Wl,-soname,libdrvctrl_pub.so -o /home/user/Documen
[ 78%] Linking CXX shared library /home/user/Documents/mw_catkin_ws/devel/lib/libdrivecontro
/usr/bin/c++ -fPIC -O3 -DNDEBUG  -shared -Wl,-soname,libdrivecontroller.so -o /home/user/Doc
[ 85%] Built target drvctrl_pub
[ 91%] Built target drivecontroller
[ 92%] Linking CXX shared library /home/user/Documents/mw_catkin_ws/devel/lib/libshape_trace
/usr/bin/c++ -fPIC -O3 -DNDEBUG  -shared -Wl,-soname,libshape_tracer.so -o /home/user/Docume
[100%] Built target shape_tracer
```

6-57

```
        Base path: /home/user/Documents/mw_catkin_ws
        Source space: /home/user/Documents/mw_catkin_ws/src
        Build space: /home/user/Documents/mw_catkin_ws/build
        Devel space: /home/user/Documents/mw_catkin_ws/devel
        Install space: /home/user/Documents/mw_catkin_ws/install
        ####
        #### Running command: "make cmake_check_build_system" in "/home/user/Documents/mw_catkin_ws/
        ####
        ####
        #### Running command: "make -j4 -l4" in "/home/user/Documents/mw_catkin_ws/build"
        ####
        '
```

Start Pioneer 3DX Gazebo World

- In the Ubuntu VM, click on **Gazebo Pioneer 3DX** desktop icon to launch the Gazebo world.

- Use `rossvcclient` to load the controller, `sl_drivecontroller`

```matlab
allServices = rosservice('list');
controllerName = 'sl_drivecontroller';
loadSvcName = allServices(endsWith(allServices,'/load_controller'));
switchCtrlSvcName = allServices(endsWith(allServices,'/switch_controller'));
% Load controller
loadCtrlSvc = rossvcclient(loadSvcName{1},'DataFormat','struct','Timeout',10);
loadCtrlMsg = rosmessage(loadCtrlSvc);
loadCtrlMsg.Name = controllerName;
call(loadCtrlSvc,loadCtrlMsg)

ans = struct with fields:
    MessageType: 'controller_manager_msgs/LoadControllerResponse'
             Ok: 1


% Start controller
switchCtrlSvc = rossvcclient(switchCtrlSvcName{1},'DataFormat','struct','Timeout',10);
swtCtrlMsg = rosmessage(switchCtrlSvc);
swtCtrlMsg.StartControllers = {controllerName};
swtCtrlMsg.Strictness = swtCtrlMsg.BESTEFFORT;
call(switchCtrlSvc,swtCtrlMsg)

ans = struct with fields:
    MessageType: 'controller_manager_msgs/SwitchControllerResponse'
             Ok: 1
```

- Alternatively, you can also use the `rqt_controller_manager` application as shown below:
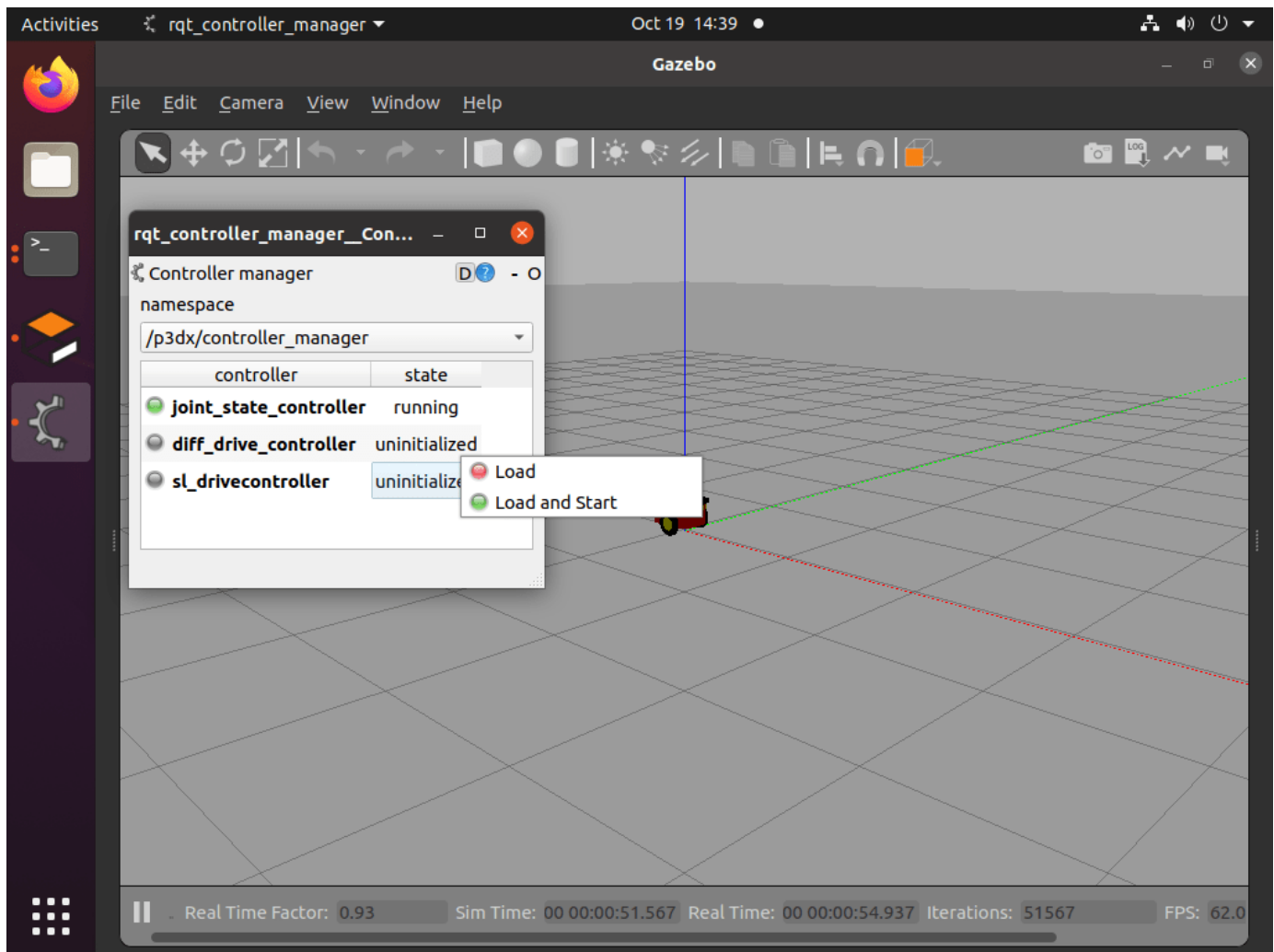
To change the display name of the controller, `sl_drivecontroller`, in `rqt_controller_manager` UI,

- Open `pioneer3dx.yaml` file and modify the structure name.
- Follow the steps in Register Controller and Build Workspace on page 6-55 to re-build the catkin workspace.
- Restart Gazebo and `rqt_controller_manager` UI to load new configuration.

Send Commands to Move the Robot

To move the Pioneer 3DX robot, open the set_command model and click **RUN** from **SIMULATE** section in the **SIMULATION** tab. Observe the robot move in the Gazebo world.
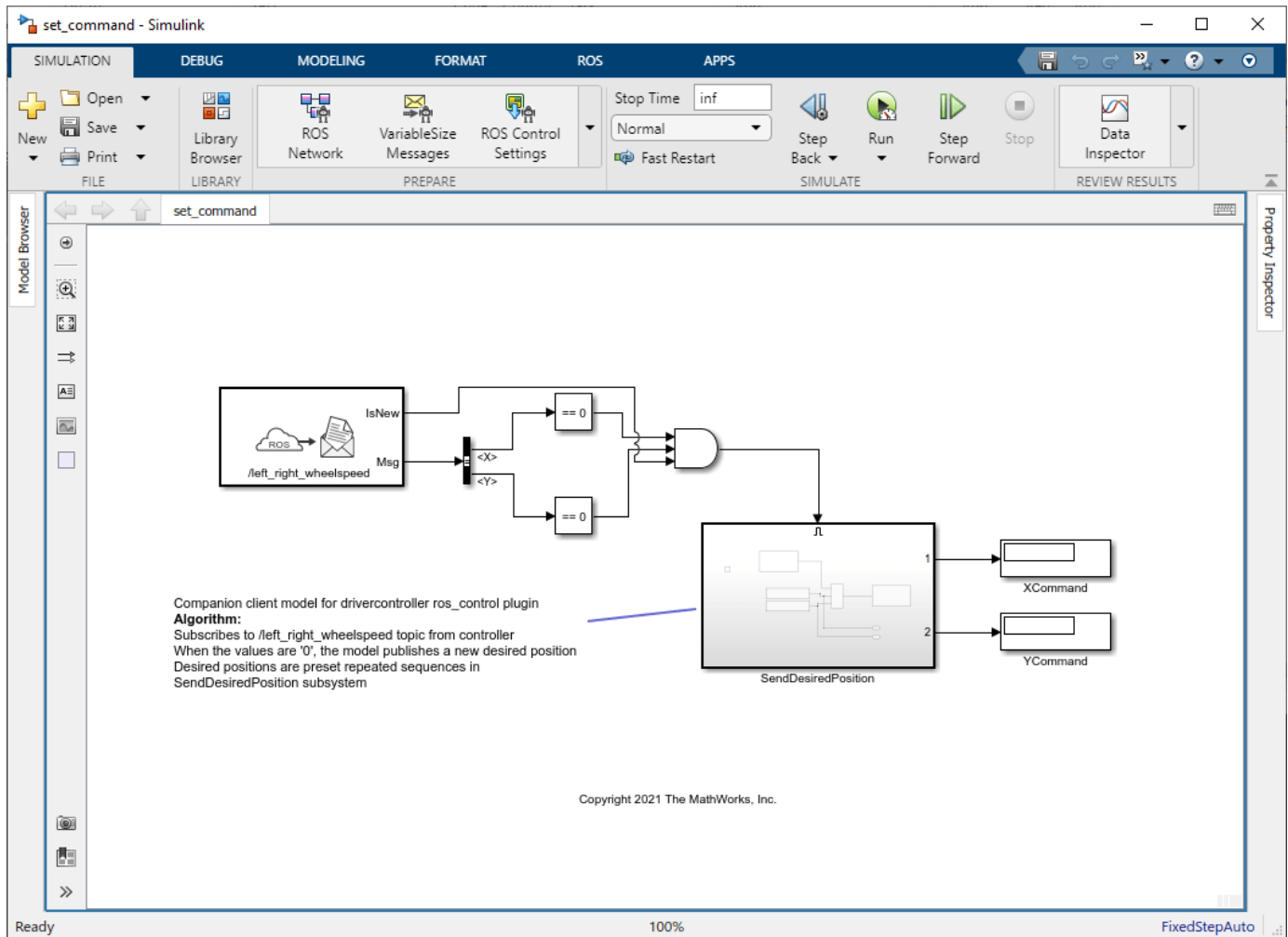
Tune Controller Parameters with Dynamic Reconfigure

While the above model sends different commands, you can tune the controller parameters `Gain (Slider)`, `Linear Velocity (Slider)` and `Distance Threshold`, using dynamic reconfigure framework. Use the following MATLAB code to send updated values for these parameters:

Display Parameter

```
dynParamSvc = rossvcclient('/p3dx/sl_drivecontroller/set_parameters',DataFormat='struct',Timeout=
msg = rosmessage(dynParamSvc);
resp = call(dynParamSvc,msg);
% List the Names and Values
for k=1:numel(resp.Config.Doubles)
    disp(resp.Config.Doubles(k));
end

    MessageType: 'dynamic_reconfigure/DoubleParameter'
           Name: 'GainSlider_gain'
          Value: 5

    MessageType: 'dynamic_reconfigure/DoubleParameter'
           Name: 'LinearVelocitySlider_gain'
```

```
        Value: 1

    MessageType: 'dynamic_reconfigure/DoubleParameter'
           Name: 'DistanceThreshold_const'
          Value: 0.5000

for k=1:numel(resp.Config.Ints)
    disp(resp.Config.Doubles(k));
end
for k=1:numel(resp.Config.Strs)
    disp(resp.Config.Doubles(k));
end
for k=1:numel(resp.Config.Bools)
    disp(resp.Config.Doubles(k));
end
```

Change Linear Velocity

```
% Set Linear Velocity (Slider) gain to 2
msg.Config.Doubles(1).Name = 'LinearVelocitySlider_gain';
msg.Config.Doubles(1).Value = 2;
call(dynParamSvc,msg)

ans = struct with fields:
    MessageType: 'dynamic_reconfigure/ReconfigureResponse'
         Config: [1×1 struct]
```

Observe that the linear velocity of the mobile robot has increased.

Change Distance Threshold

```
% Set Distance Threshold to 0.1 and Linear Velocity to 0.5
msg.Config.Doubles(1).Name = 'LinearVelocitySlider_gain';
msg.Config.Doubles(1).Value = 0.5;
msg.Config.Doubles(2).Name = 'DistanceThreshold_const';
msg.Config.Doubles(2).Value = 0.1;
call(dynParamSvc,msg)

ans = struct with fields:
    MessageType: 'dynamic_reconfigure/ReconfigureResponse'
         Config: [1×1 struct]
```

With smaller velocity and distance threshold, the robot should be able to go very close to the desired position.

Use rqt_reconfigure Application

Alternatively, you can also change these values from the `rqt_reconfigure` application on the Virtual Machine as shown in the snapshot below:
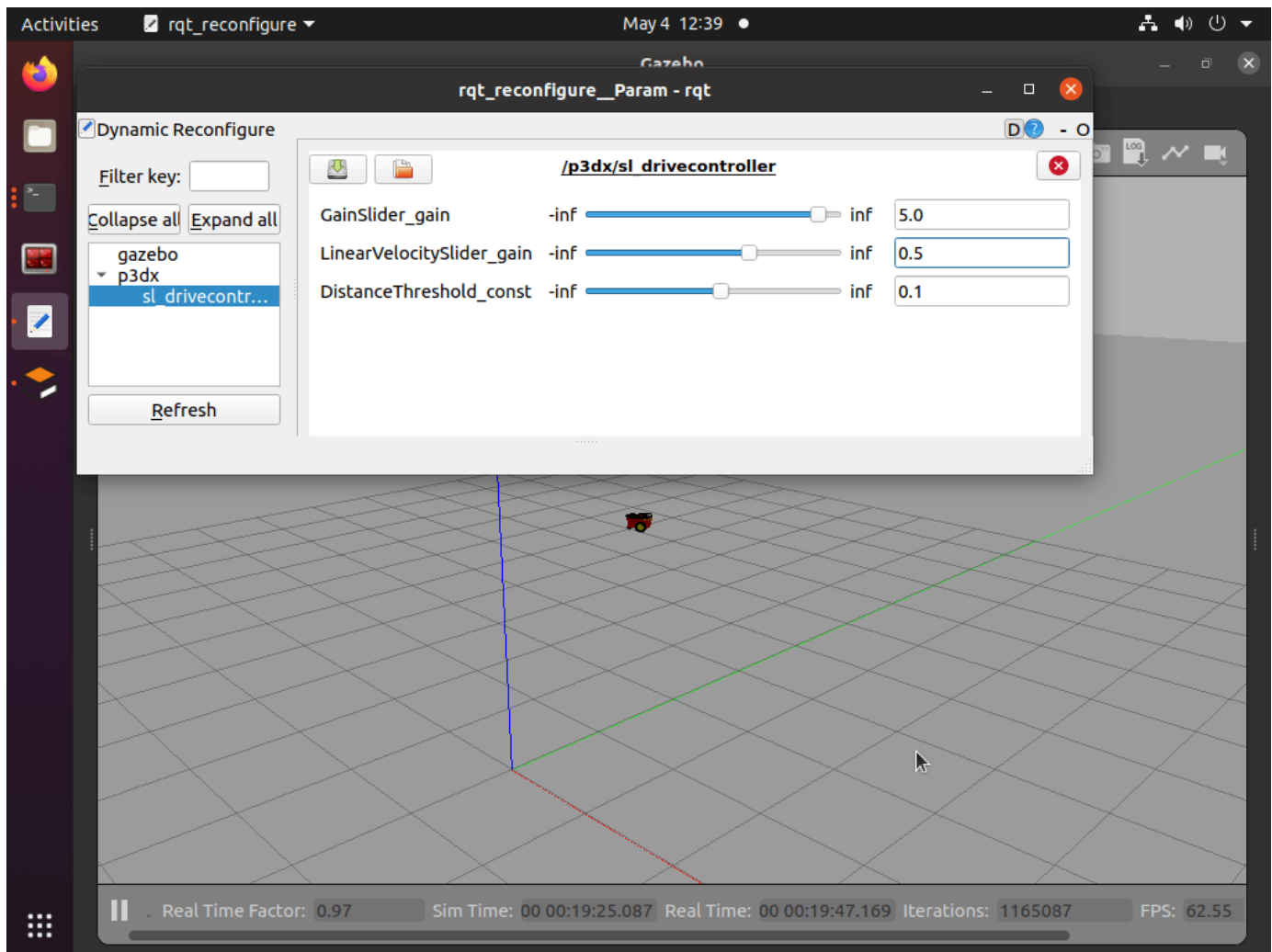
Shutdown the global MATLAB node

`rosshutdown`

Shutting down global node /matlab_global_node_69673 with NodeURI http://192.168.93.1:49973/ and M

References

S. Chitta, E. Marder-Eppstein, W. Meeussen, V. Pradeep, A. Rodríguez Tsouroukdissian, J. Bohren, D. Coleman, B. Magyar, G. Raiola, M. Lüdtke and E. Fernandez Perdomo

- "ros_control: A generic and simple control framework for ROS"
- The Journal of Open Source Software, 2017. PDF

Log ROS Messages from Simulink to a Rosbag Logfile

This example enables you to save ROS message data from Simulink® to a rosbag file.

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. Using the ROS Logger app, you can log ROS messages from Simulink® models to rosbag files.

Open the Simulink model. This model is configured for ROS Simulation. To configure a new model for ROS simulation, under **Apps**, select **Robot Operating System (ROS)** and specify the appropriate ROS network configuration parameters.

```
open_system("logROSMessagesFromSimulinkExampleModel")
```
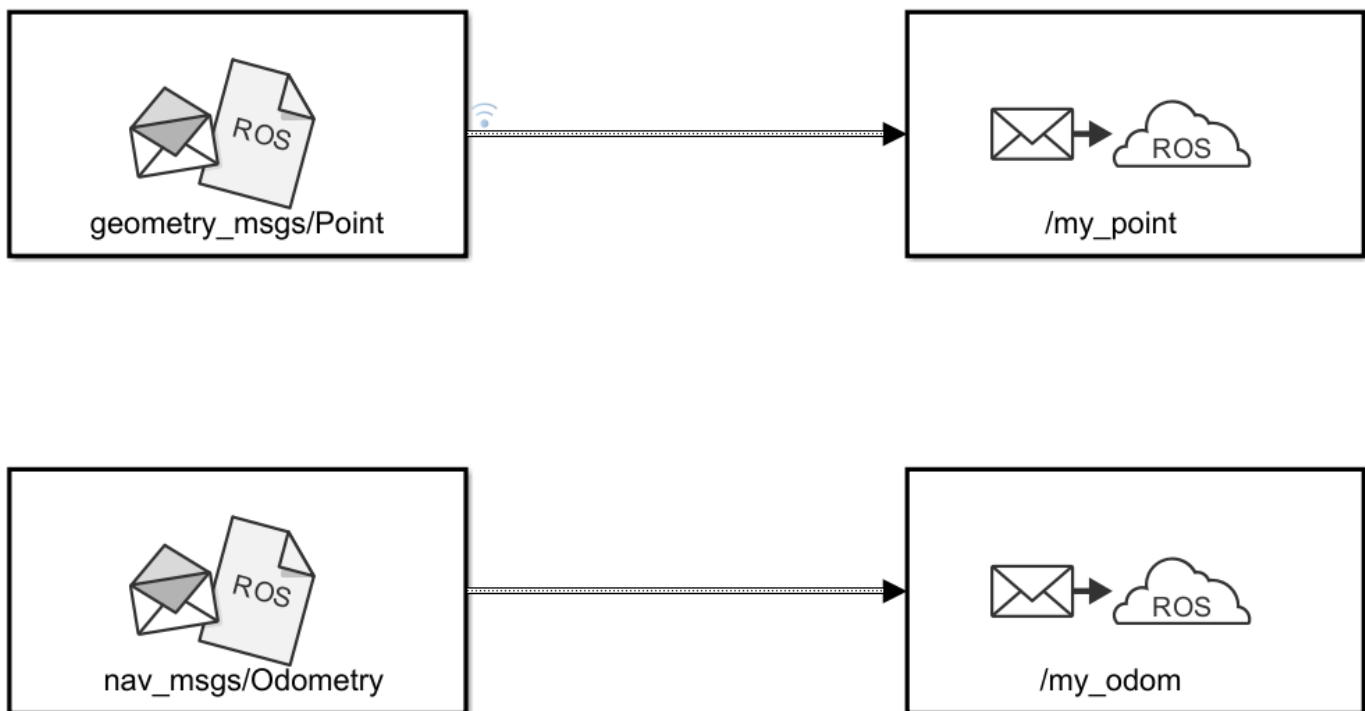

Copyright 2022 The MathWorks, Inc.

To open the ROS Logger app dialog, under **Simulation** tab, from **Prepare** section, select **ROS Logger**.

Follow these steps to configure the signals to be logged and the rosbag file parameters:

- Ensure that **Enable ROS signal logging to ROS Bag file** is checked.
- Select the signals to be logged in the rosbag file by checking their respective checkboxes under the **Enable** column.
- The default name of the generated rosbag file will be *modelName_mmddyy_hh_mm_ss.bag.* To provide a custom name, from the **Logfile name** dropdown, select `Custom`. Then specify the desired **File name**.
- Specify the **Compression format** to be `uncompressed`, `bz2` or `lz4`.
- Specify the desired chunk size for writing, in bytes.

Initialize the ROS network.

```
rosinit
```

```
Launching ROS Core...
......Done in 7.0793 seconds.
Initializing ROS master on http://172.21.16.85:51228.
Initializing global node /matlab_global_node_26159 with NodeURI http://ah-avijayar:49405/ and Mas
```

Simulate the model.

```
sim("logROSMessagesFromSimulinkExampleModel");
```

Simulation Complete. Start logging ROS bag file...
Successfully logged ROS bag file to logROSMessagesFromSimulinkExampleModel_061622_15_56_22.bag.

After simulation, logging messages to rosbag file begins and completes with the success message on
the **Diagnostic Viewer.**

Use ROS Logger App to Save ROS Messages from Simulink

Use ROS Logger app to record ROS messages during Simulink® simulation, and obtain a rosbag file with fully synchronized ROS messages saved during simulation.

In this example, you will:

- Load pre-defined 3D simulation environment provided by Automated Driving Toolbox(TM).
- Configure ROS message logging in ROS Logger app.
- Visualize logged ROS messages in ROS Bag Viewer app.

Load 3D Simulation Environment

Use the prebuilt Large Parking Lot scene created using the Unreal Engine 3D simulation environment in Simulink. To interactively select a sequence of waypoints from a scene and generate a custom vehicle trajectory, refer to "Select Waypoints for Unreal Engine Simulation" (Automated Driving Toolbox) example.

```
% Extract scene for visualization
sceneName = 'LargeParkingLot';
[sceneImage, sceneRef] = helperGetSceneImage(sceneName);
hScene = figure;
helperShowSceneImage(sceneImage, sceneRef)
title(sceneName)

% Interactively Select Waypoints
hFig = helperSelectSceneWaypoints(sceneImage, sceneRef);

% Prepare smooth poses for simulation
if exist('refPoses','var')==0 || exist('wayPoints','var')==0
    % Load MAT-file containing preselected waypoints
    data = load('waypointsForROSLoggerAppParking');

    % Assign to caller workspace
    assignin('caller','wayPoints',data.wayPoints);
    assignin('caller','refPoses',data.refPoses);
end
numPoses = size(refPoses{1}, 1);

refDirections  = ones(numPoses,1);   % Forward-only motion
numSmoothPoses = 10 * numPoses;      % Increase this to increase the number of returned poses

[smoothRefPoses,~,cumLengths] = smoothPathSpline(refPoses{1}, refDirections, numSmoothPoses);
```

Configure ROS Message Logging

To configure saving options, open the ROS Logger app under SIMULATION> PREPARE> ROS Logger. You can enable/disable ROS messages for saving, define custom file name, and rename messages saved to rosbag file based on your preference.
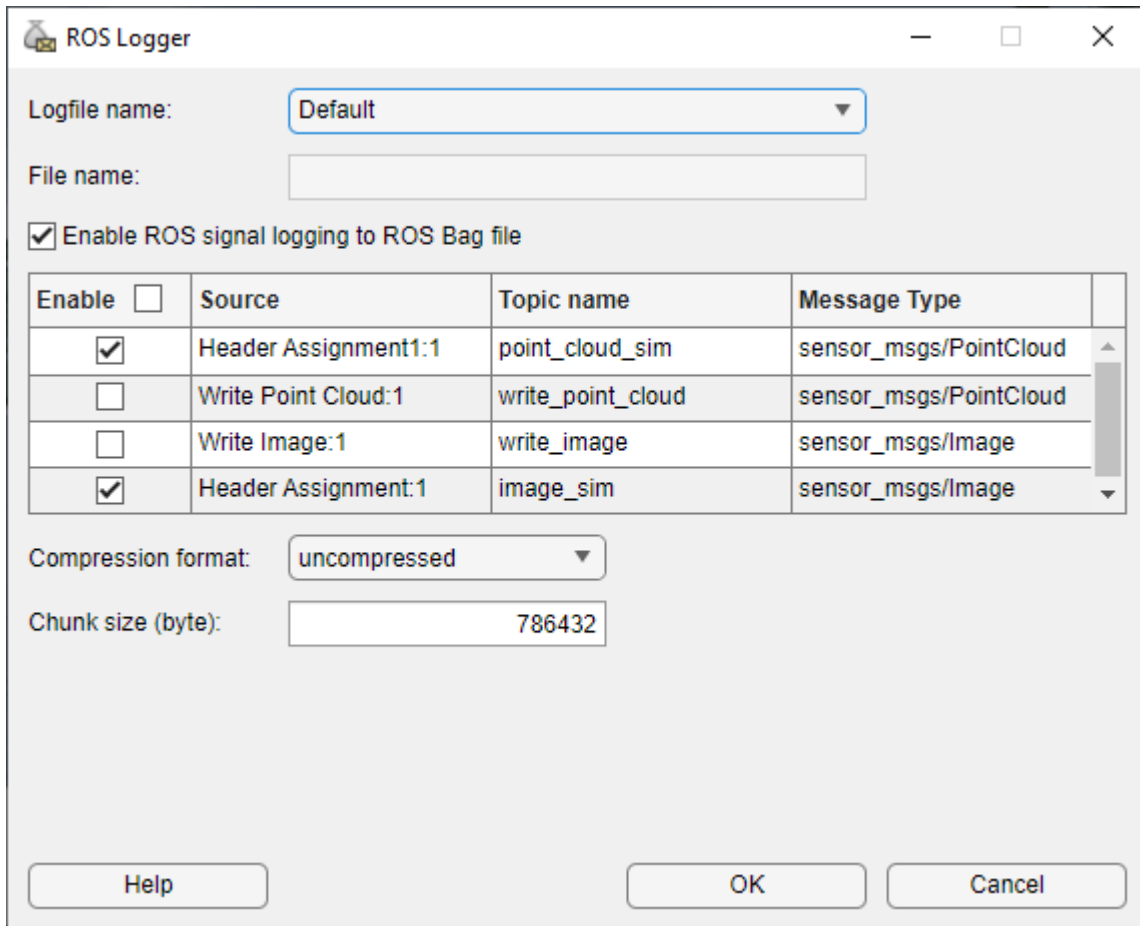
After configuring with ROS Logger app, run these commands to setup model parameters and run the simulation.

```
modelName = 'LogROSMessageFrom3DSimulation';
open_system(modelName);
% Configure the model to stop simulation at 5 seconds.
simStopTime = 5;
set_param(gcs, 'StopTime', num2str(simStopTime));

% Create a constant velocity profile by generating a time vector
% proportional to the cumulative path length.
timeVector = normalize(cumLengths, 'range', [0, simStopTime]);

% Create variables required by the Simulink model.
refPosesX = [timeVector, smoothRefPoses(:,1)];
refPosesY = [timeVector, smoothRefPoses(:,2)];
refPosesT = [timeVector, smoothRefPoses(:,3)];
% Run the simulation
rosinit

Launching ROS Core...
.......Done in 7.9891 seconds.
Initializing ROS master on http://172.21.17.121:53417.
Initializing global node /matlab_global_node_65044 with NodeURI http://ah-sransing-old:61320/ and
```

```
sim(modelName);
```

```
Simulation Complete. Start logging ROS bag file...
Successfully logged ROS bag file to LogROSMessageFrom3DSimulation_110722_09_35_40.bag.
```

After running the simulation, you can see a rosbag file generated in your current working directory. Long running simulations take some time to generate rosbag files. You can proceed to inspect the bag with "`rosbag info`" after you see the message "Successfully logged ROS bag file to…".

Visualize Logged ROS Messages

Open the ROS Bag Viewer app

```
rosbagViewer
```

Click **Open** and browse for the generated bag file and load it. Inspect the **Topic List** and **Source Details** panels on the left side of the app to check the bag file info. Create **Image** and **Point Cloud** visualizers from the toolstrip. Play the bag file to visualize the logged messages.
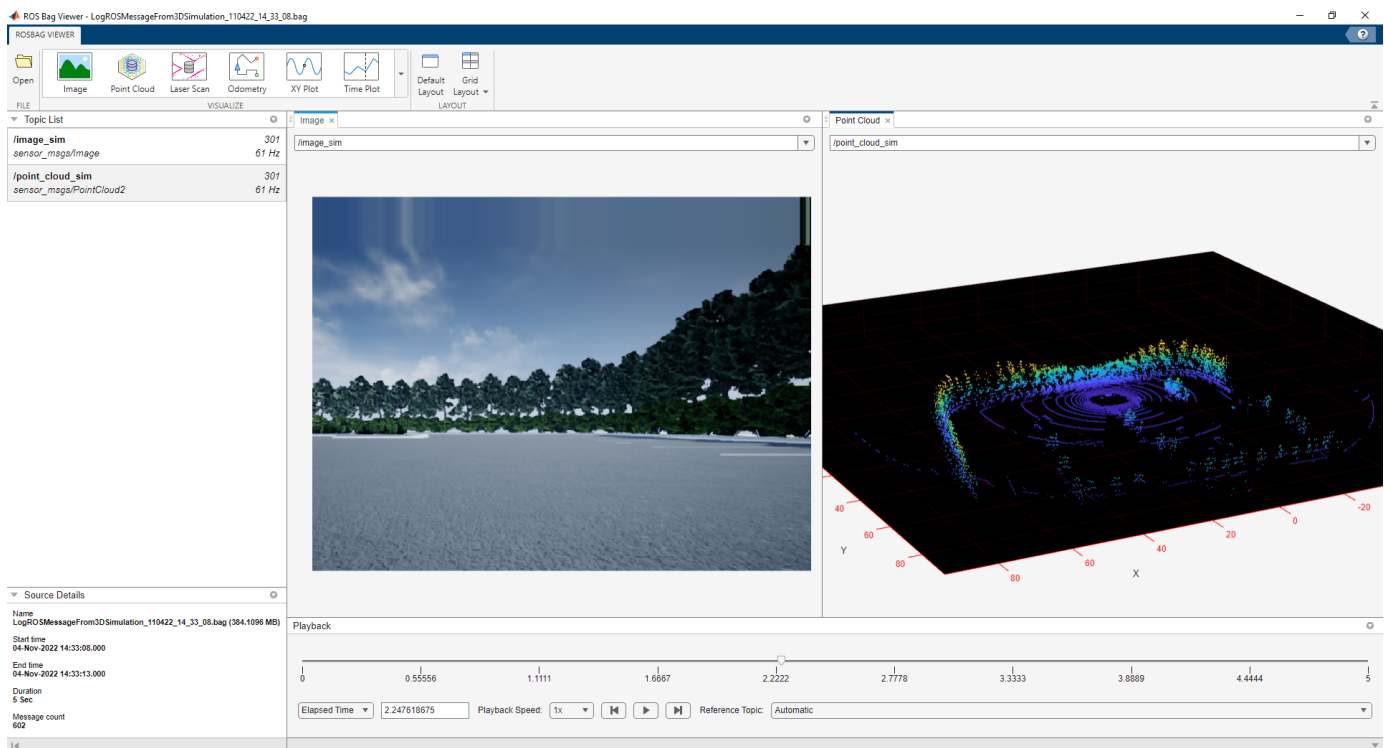

```
% Shutdown ROS network and close all windows
rosshutdown
```

```
Shutting down global node /matlab_global_node_65044 with NodeURI http://ah-sransing-old:61320/ a
Shutting down ROS master on http://172.21.17.121:53417.
```

```
close(hFig)
close_system(modelName, 0)
close(hScene)
```

Emergency Braking of Ego Vehicle in CARLA Simulator Using Simulink and CARLA ROS Bridge

This example shows how to set up, connect to, and control an ego vehicle in the CARLA simulator using ROS Toolbox™.

Overview

You control the CARLA ego vehicle by sending steering, braking, and throttle control signals to a vehicle controls publisher using Simulink® software. The model reads point cloud data from the front camera, odometer, and lidar sensors of the ego vehicle. The point cloud processor removes ground reflection and calculates the average distance of the objects in front of the ego vehicle. The model sends the calculated distance to the emergency braking unit which issues a braking command if the distance drops below 10 meters.

Setup

Install the latest CARLA simulator on your host machine, available here https://github.com/carla-simulator/carla/releases/tag/0.9.13. This example uses the binary installation of version 0.9.13 of the CARLA Simulator.

Download and install the ROS virtual machine by following the instructions on this page: https://www.mathworks.com/ros_vm_install/v7. This virtual machine is based on the Ubuntu® Linux® operating system and is pre-configured to support the examples in ROS Toolbox™.

Here is the diagram depicts the setup of this example.
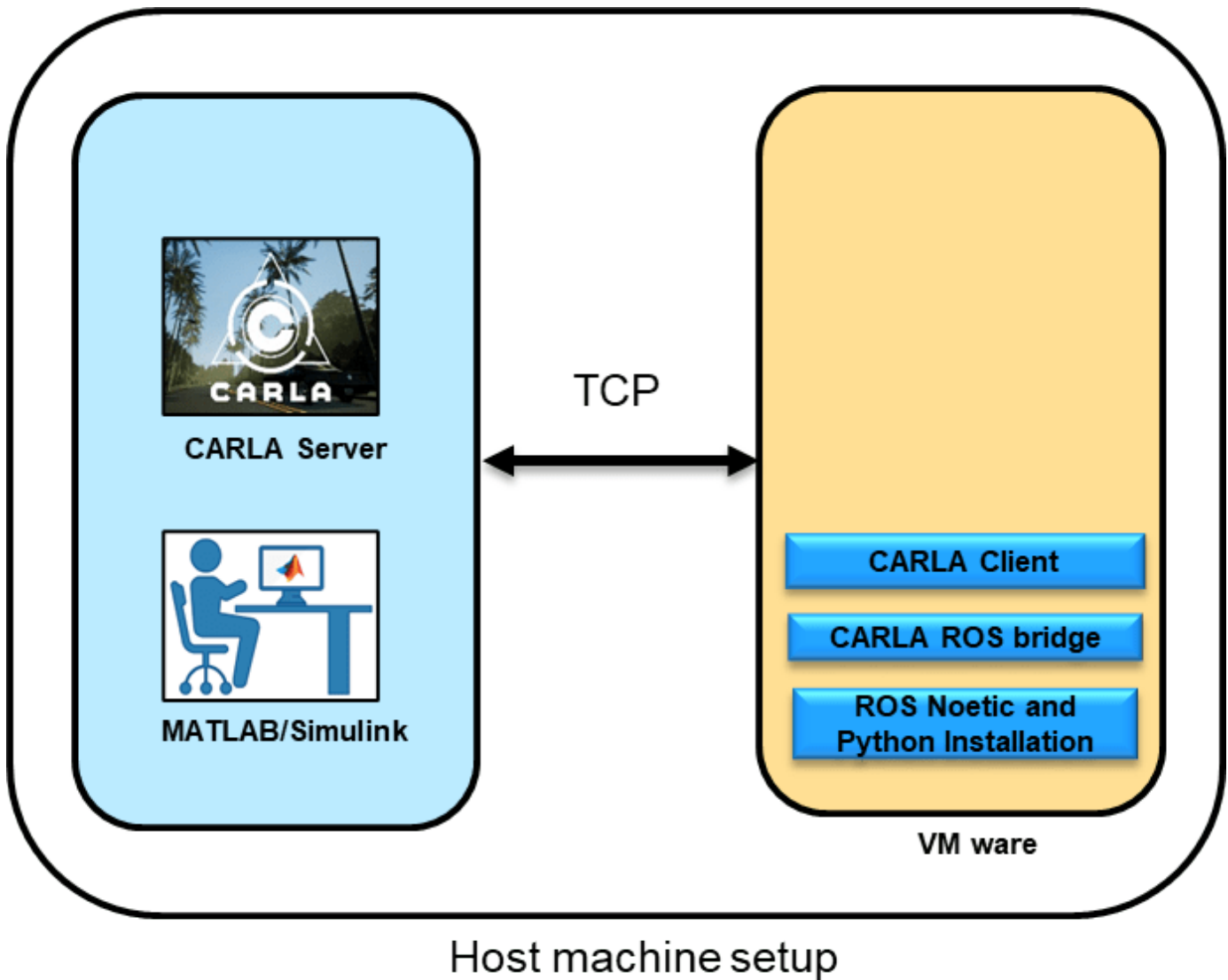
Host machine setup

Step 1: Start the CARLA Server

- On your windows host machine, navigate to the installed location of CARLA. Click on the **CarlaUE4.exe** to start the CARLA server.

Step 2: Setup the CARLA ROS bridge

- Launch the virtual machine.
- On the Ubuntu desktop, click the **ROS Noetic Core Terminal** shortcut to start the ROS master. Once the master is launched, please note down the **ROS_MASTER_URI**.
- On the Ubuntu desktop, click the **ROS Noetic Terminal** shortcut to start the ROS terminal. Run this command to setup the CARLA ROS bridge environment:

```
. ~/carla-ros-bridge/catkin_ws/devel/setup.bash
```
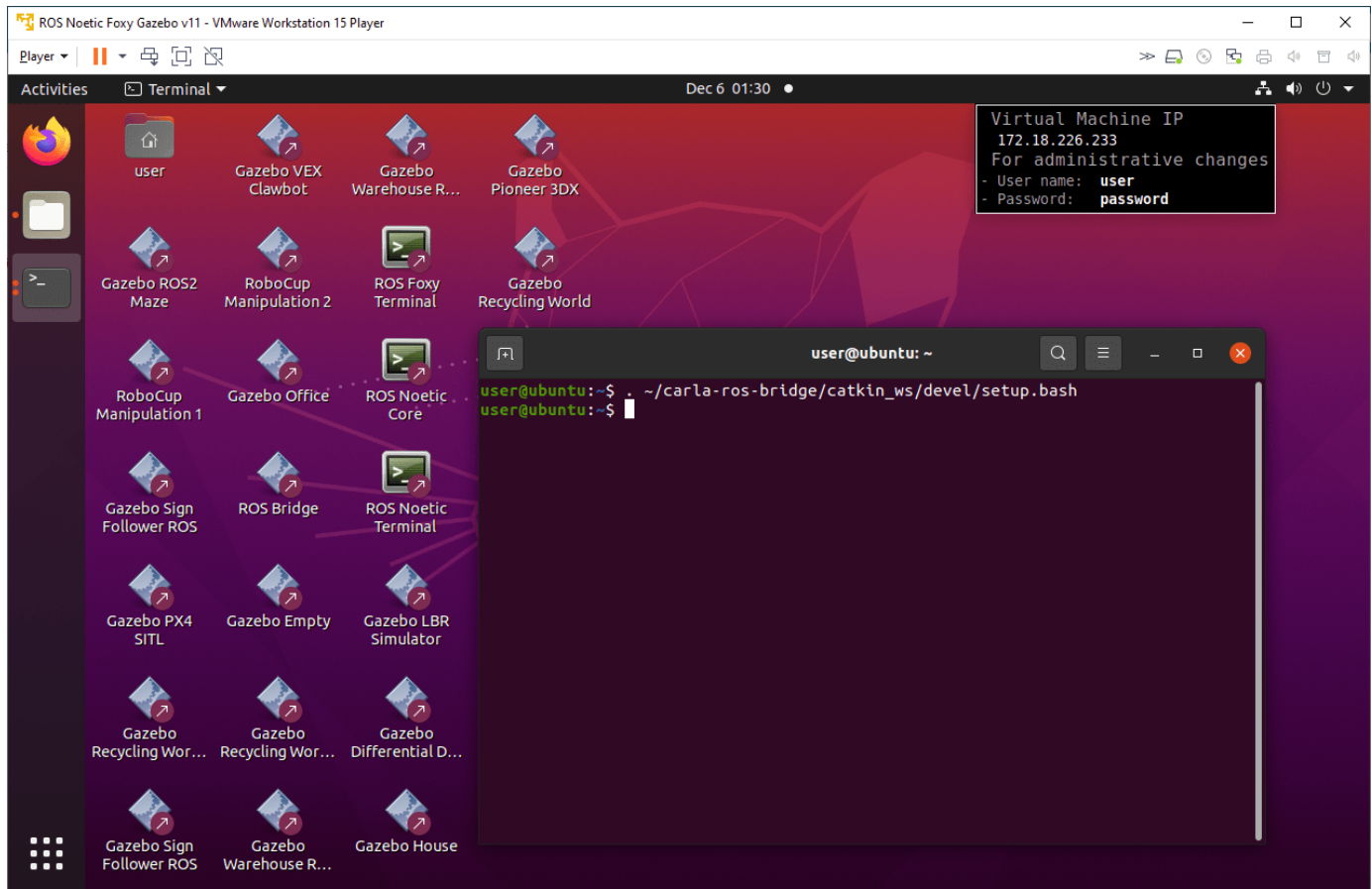
Step 3: Launch Ego Vehicle in Town03 using CARLA client

- In the same terminal, run the `follow` command to launch the ego vehicle in the Town03 environment of CARLA simulator. The `follow` command launches the ego vehicle near to the gas station.

- Run this command to change the host address to the host address of your machine:

```
roslaunch carla_ros_bridge carla_ros_bridge_with_example_ego_vehicle.launch
host:=172.18.226.60 timeout:=60000 town:='Town03'
spawn_point:=-25,-134,0.5,0,0,-90
```
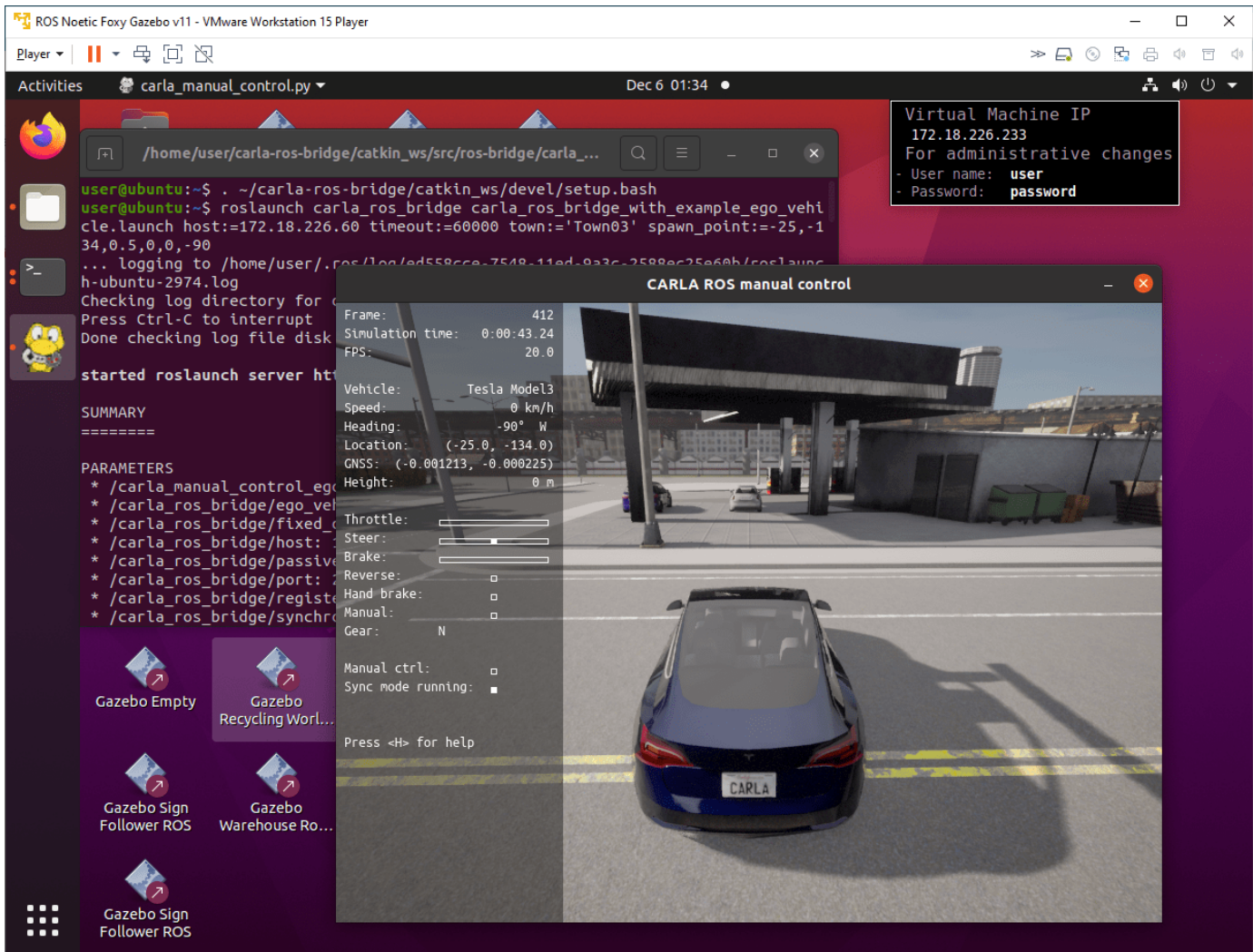
Control Ego Vehicle in CARLA Using Simulink

Connect Simulink to the ROS master running in the VM Ware.

```
rosinit('http://172.18.226.233:11311')
```

Initializing global node /matlab_global_node_37271 with NodeURI http://172.18.226.60:60619/ and I

Ensure that you can access ROS topics related to the CARLA simulator by running the following introspection command.

```
rostopic list
```

```
/carla/ego_vehicle/collision
/carla/ego_vehicle/control/set_transform
/carla/ego_vehicle/enable_autopilot
/carla/ego_vehicle/gnss
/carla/ego_vehicle/lane_invasion
/carla/ego_vehicle/odometry
/carla/ego_vehicle/rgb_view/image
/carla/ego_vehicle/vehicle_control_cmd_manual
```

```
/carla/ego_vehicle/vehicle_control_manual_override
/carla/ego_vehicle/vehicle_info
/carla/ego_vehicle/vehicle_status
/carla/status
/clock
/initialpose
/rosout
/rosout_agg
/tf
```

Open the Simulink model.

```
open_system('EmergencybrakingofCarlaEgoVehicle');
```

The model shows the point cloud data from the lidar sensor that the vehicle uses to measure the distance to the obstacle. When the distance is below 10 meters, the emergency braking system sends a braking command to the vehicle.
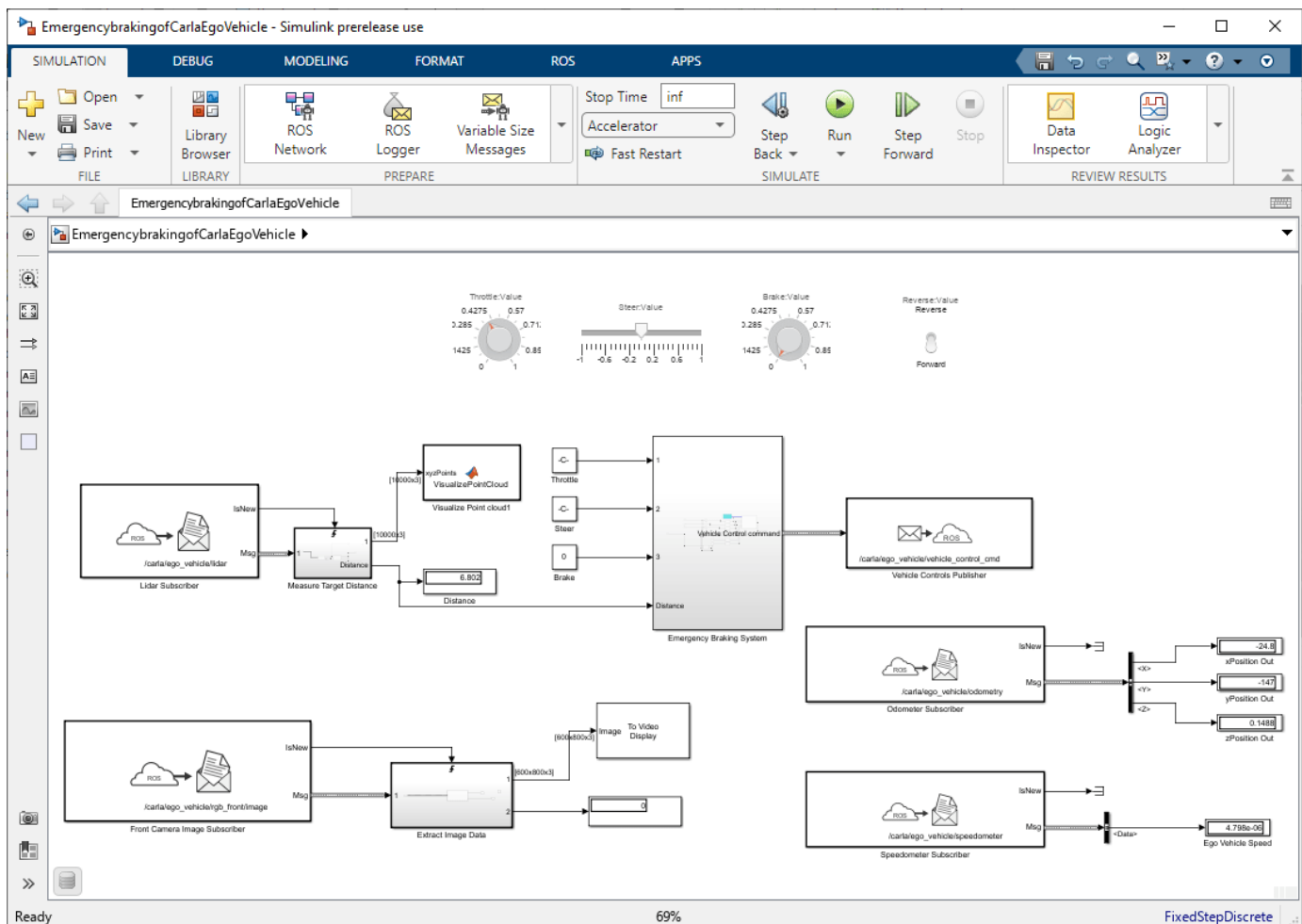
Run the Simulink model. The ego vehicle automatically stops 10 meters ahead of the vehicle in front of it at the gas station. The CARLA ROS manual control terminal shows the stationary vehicle with the brake applied.
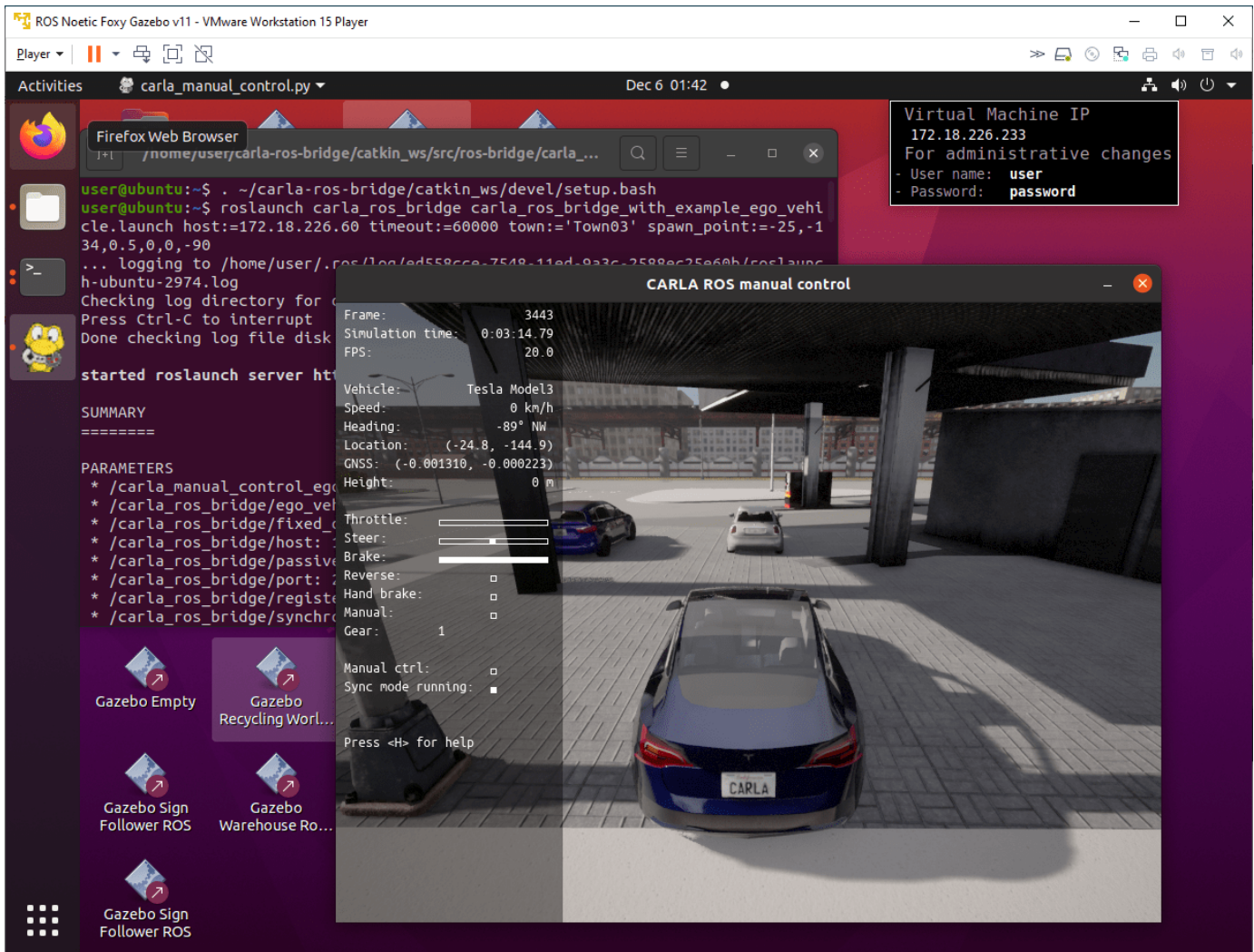
Shut down the ROS network.

```
rosshutdown
```

Shutting down global node /matlab_global_node_37271 with NodeURI http://172.18.226.60:60619/ and

ROS 2 Simulink Topics

Log ROS 2 Messages from Simulink to a ROS 2 Bag File

This example shows how to save ROS 2 message data from Simulink® to a ROS 2 Bag File.

A `ros2bag` or ROS 2 Bag is a file format in ROS 2 for storing message data. These bags are often created by subscribing to one or more ROS 2 topics, and storing the received message data in an efficient file structure. Using the ROS 2 Logger app, you can log ROS 2 messages from Simulink® models to `ros2bag` files.

Open the Simulink model. This model is configured for ROS 2 Simulation. To configure a new model for ROS 2 simulation, under **Apps**, select **Robot Operating System (ROS)** and specify the appropriate ROS 2 network configuration parameters.

```
open_system("logROS2MessagesFromSimulinkExampleModel")
```
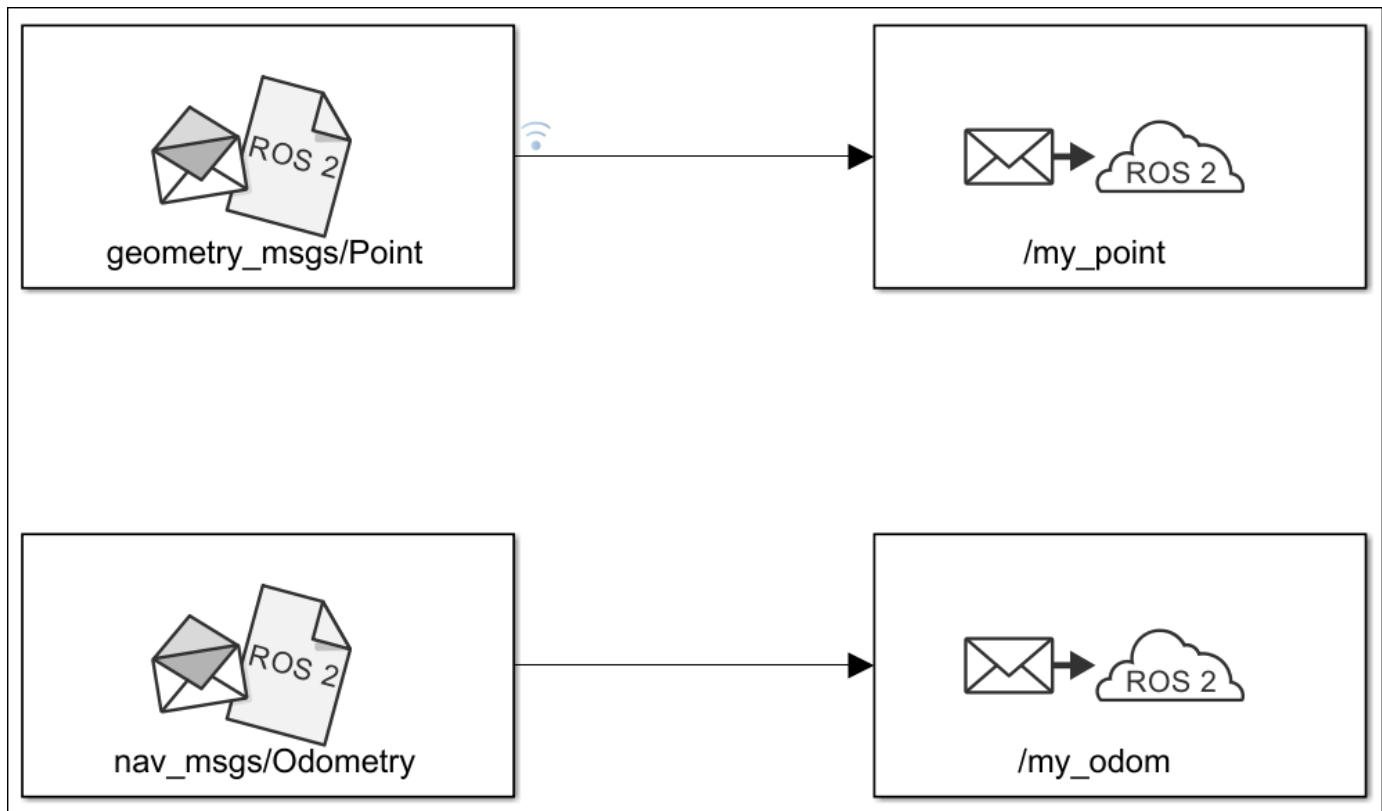

To open the ROS 2 Logger app dialog, select **ROS Logger** under **Simulation** tab from **Prepare** section.

Follow these steps to configure the signals to be logged and the `ros2bag` file parameters.

1   Ensure that **Enable ROS signal logging to ROS 2 Bag file** is checked.

2   Select the signals to be logged in the `ros2bag` file by checking their respective checkboxes under the **Enable** column.

3   If **Simulation Data Inspector** is set to **Off**, the corresponding signal will not be logged anyway. If **Simulation Data Inspector** is set to **On**, the signal will be logged only when it is set to **Enabled** in the app.

4   The default name of the generated `ros2bag` file will be *modelName_mmddyy_hh_mm_ss.bag.* To provide a custom name, from the **Logfolder name** dropdown, select `Custom`. Then specify the desired **Folder name.**

5   Specify the desired **Cache size**.

Simulate the model.

```
sim("logROS2MessagesFromSimulinkExampleModel");
```

```
Simulation Complete. Start logging ROS 2 bag file...
Successfully logged ROS 2 bag file to logROS2MessagesFromSimulinkExampleModel_030323_16_49_03.
```

After simulation, logging messages to `ros2bag` file begins and completes with the success message on the **Diagnostic Viewer.**

Use ROS 2 Logger App to Save ROS 2 Messages from Simulink

Use ROS 2 Logger app to record ROS 2 messages during Simulink® simulation, and obtain a `ros2bag` file with fully synchronized ROS 2 messages saved during simulation.

In this example, you will

- Load pre-defined 3D simulation environment provided by Automated Driving Toolbox(TM).
- Configure ROS 2 message logging in ROS 2 Logger app.
- Visualize logged ROS 2 messages.

Load 3D Simulation Environment

Use the prebuilt Large Parking Lot scene created using the Unreal Engine 3D simulation environment in Simulink. To interactively select a sequence of waypoints from a scene and generate a custom vehicle trajectory, refer to "Select Waypoints for Unreal Engine Simulation" (Automated Driving Toolbox) example.

```matlab
% Extract scene for visualization
sceneName = 'LargeParkingLot';
[sceneImage, sceneRef] = helperGetSceneImage(sceneName);
hScene = figure;
helperShowSceneImage(sceneImage, sceneRef)
title(sceneName)

% Interactively Select Waypoints
hFig = helperSelectSceneWaypoints(sceneImage, sceneRef);

% Prepare smooth poses for simulation
if exist('refPoses','var')==0 || exist('wayPoints','var')==0
    % Load MAT-file containing preselected waypoints
    data = load('waypointsForROSLoggerAppParking');

    % Assign to caller workspace
    assignin('caller','wayPoints',data.wayPoints);
    assignin('caller','refPoses',data.refPoses);
end
numPoses = size(refPoses{1}, 1);

refDirections  = ones(numPoses,1);   % Forward-only motion
numSmoothPoses = 10 * numPoses;      % Increase this to increase the number of returned poses

[smoothRefPoses,~,cumLengths] = smoothPathSpline(refPoses{1}, refDirections, numSmoothPoses);
```

Configure ROS 2 Message Logging

To launch ROS 2 Logger app and configure saving options, click **ROS Logger** in **Prepare** section under **Simulation**. You can enable or disable ROS 2 messages for saving, define custom file name, and rename messages saved to `ros2bag` file based on your preference.
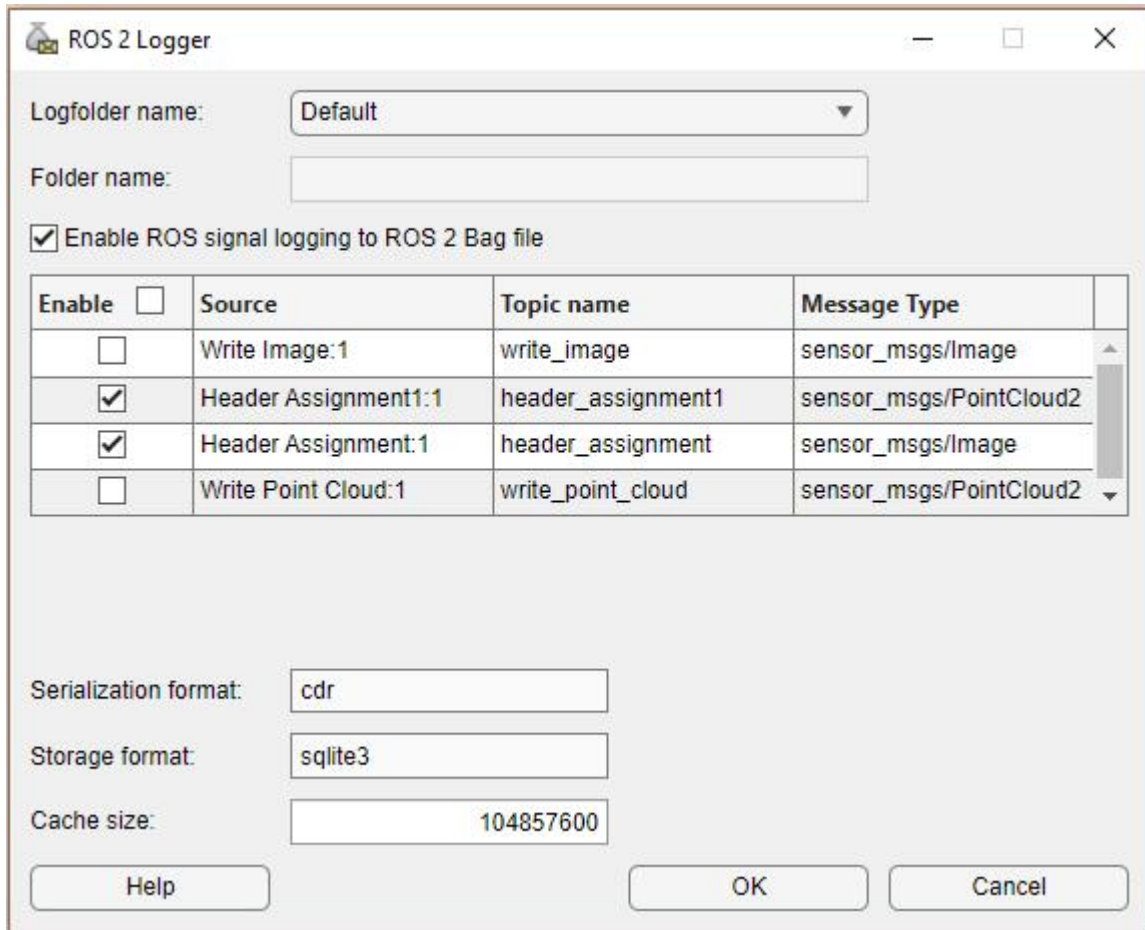
After configuring with ROS 2 Logger app, run these commands to setup model parameters and run the simulation.

```
modelName = 'LogROS2MessageFrom3DSimulation';
open_system(modelName);
% Configure the model to stop simulation at 5 seconds.
simStopTime = 5;
set_param(gcs, 'StopTime', num2str(simStopTime));

% Create a constant velocity profile by generating a time vector
% proportional to the cumulative path length.
timeVector = normalize(cumLengths, 'range', [0, simStopTime]);

% Create variables required by the Simulink model.
refPosesX = [timeVector, smoothRefPoses(:,1)];
refPosesY = [timeVector, smoothRefPoses(:,2)];
refPosesT = [timeVector, smoothRefPoses(:,3)];

% Run the simulation
sim(modelName);
```

After running the simulation, you can see a `ros2bag` folder generated in your current working directory. Long running simulations take some time to generate `ros2bag` files.

Visualize Logged ROS 2 Messages

Based on how you setup the "Folder name" in ROS 2 Logger app, the name of the `ros2bag` folder would be different. In this section, we use `folderpath` to represent the path to the generated bag folder.

You can proceed to inspect the bag with "`ros2("bag","info",folderpath)`" after you see the message "Successfully logged ROS 2 bag file to...".

You can inspect the saved PointCloud2 messages with this message.

```
bagReader = ros2bagreader(folderpath);
bagSel = select(bagReader,"Topic","/header_assignment1");
pointCloudMsgs = readMessages(bagSel);
ptCloudXYZ = rosReadXYZ(pointCloudMsgs{1,1});
player = pcplayer([-50 100],[-50 100],[-50 100]);
ptCloud = pointCloud(ptCloudXYZ);
view(player,ptCloud);
```
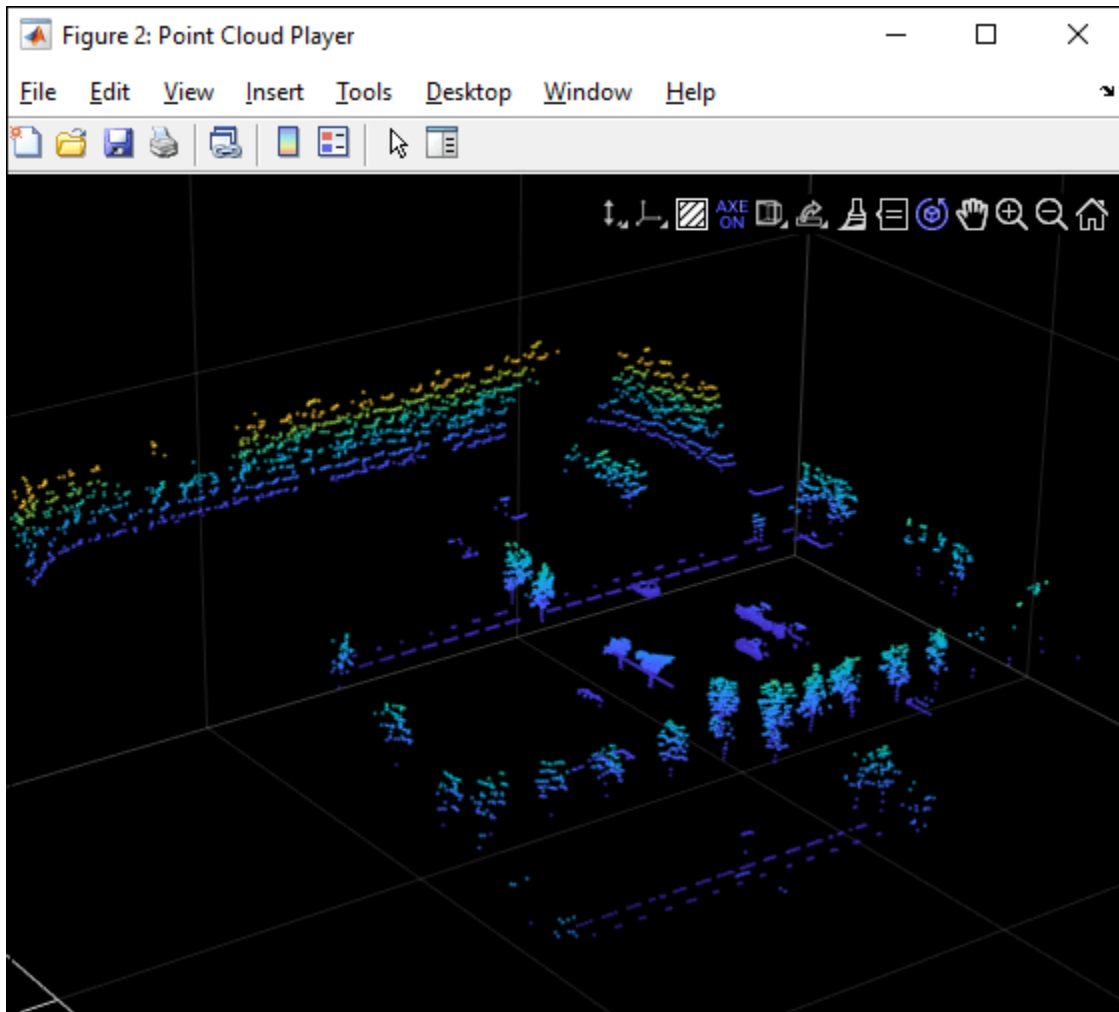

You can also inspect the saved Image messages with this message.

```
bagReader = ros2bagreader(folderpath);
bagSel = select(bagReader,"Topic","/header_assignment");
imgMsgs = readMessages(bagSel);
img = rosReadImage(imgMsgs{201,1});
imshow(img)
```
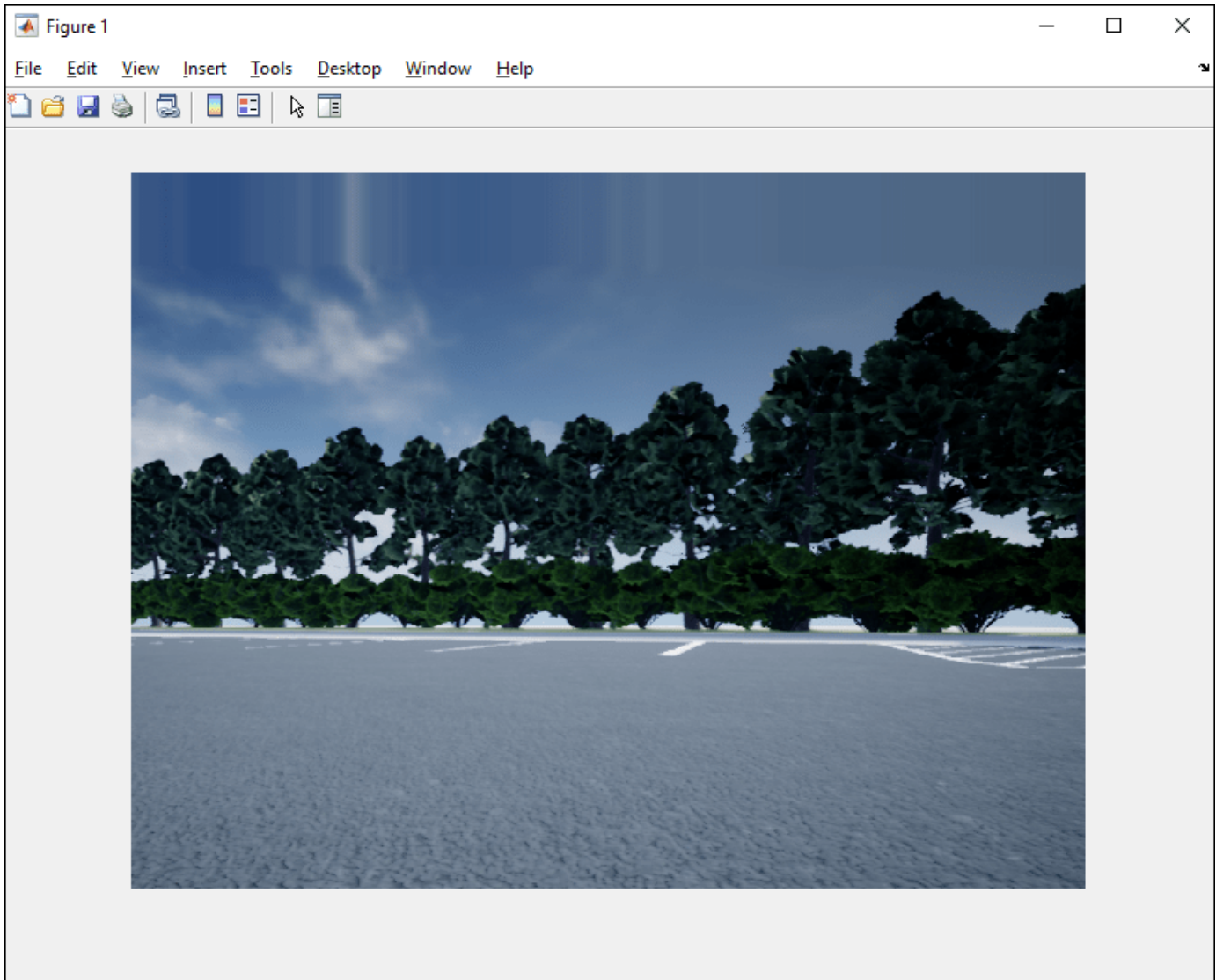

Close the Simulink model and all the windows.

```
close(hFig)
close_system(modelName, 0)
close(hScene)
```

ROS CUDA Simulink Topics

Lane and Vehicle Detection in ROS Using YOLO v2 Deep Learning Algorithm

This example shows how to use deep convolutional neural networks inside a ROS enabled Simulink® model to perform lane and vehicle detection. In this example, you first read traffic video as the input and publish the frames as sensor_msgs/Image messages to a topic on the ROS network. Then you detect vehicles, and the left and right lane boundaries corresponding to the ego vehicle in every frame, annotate the input image with the detections, and publish them to a topic in the ROS network. Finally, you generate CUDA® optimized code for the ROS node from the Simulink model for lane and vehicle detection.

This example uses the pretrained lane detection network from the *Lane Detection Optimized with GPU Coder* example of GPU Coder Toolbox™. For more information, see "Lane Detection Optimized with GPU Coder" (GPU Coder).

This example also uses the pretrained vehicle detection network from the *Object Detection Using YOLO v2 Deep Learning* example of Computer Vision Toolbox™. For more information, see Object Detection Using YOLO v2 Deep Learning.

This example illustrates the following concepts:

- Model the conversion of video frames to ROS image in Simulink.
- Model the lane detection application in Simulink. First the traffic video is preprocessed by resizing to 227-by-227-by-3 and multiplying by a constant factor of 255. It is then fed to the pretrained network loaded in the Predict block from Deep Learning Toolbox™. Finally, if the left and right lane boundaries are detected, the parabolic coefficients to model the trajectories of the lane boundaries are obtained.
- Model the vehicle detection application in Simulink. The traffic video is processed by a pretrained YOLO v2 detector. This network detects vehicles in the video and outputs the coordinates of the bounding boxes for these vehicles and their confidence score.
- Configure the Simulink model for CUDA ROS node generation on host platform.
- Generate a CUDA executable for the Simulink model configured for ROS.

Third-Party Prerequisites

- CUDA enabled NVIDIA® GPU
- NVIDIA CUDA toolkit and driver
- NVIDIA cuDNN library
- Environment variables for the compilers and libraries. For more information, see "Installing Prerequisite Products" (GPU Coder) and "Setting Up the Prerequisite Products" (GPU Coder).

Overview

This example uses three Simulink models, which are each represented as ROS nodes.. The following diagram illustrates their interaction. The rosImagePublisher node reads the raw video file and converts the frame data into a ROS image. The laneAndVechicleDetection node performs lane and vehicle detection and publishes the annotated image. The annotateImageDisplay node displays the annotated images as a video stream.
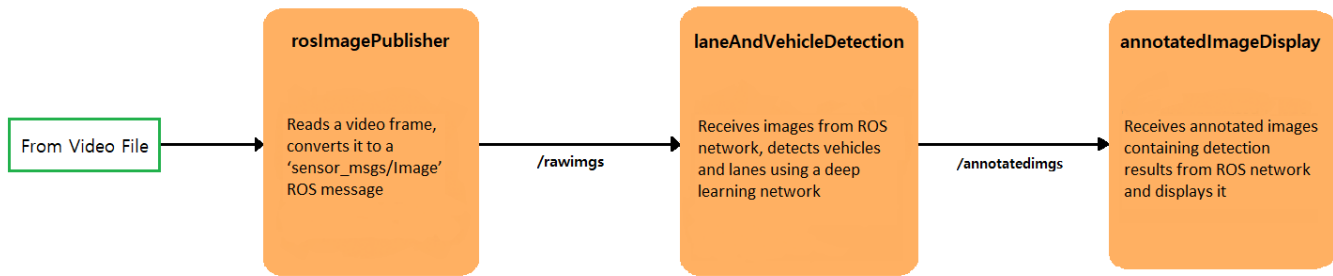
Download Example Video

```matlab
if ~exist('./caltech_washington1.avi','file')
    url = 'https://www.mathworks.com/supportfiles/gpucoder/media/caltech_washington1.avi';
    websave('caltech_washington1.avi',url);
end
```

Get Pretrained Lane and Vehicle Detection Networks

The function downloads the `trainedLaneNet.mat` and `yolov2ResNet50VehicleExample.mat` files if they are not already present.

```matlab
getVehicleDetectionAndLaneDetectionNetworks
```

Lane and Vehicle Detection Simulink Model

The `laneAndVehicleDetection` model performs lane and vehicle detection as shown. When lane images are being published to the topic `/rawlaneimgs`, the model runs the detection algorithm on them and publishes the annotated images to a different topic. For a more detailed explanation of Lane Detection, Vehicle Detection or Annotation using Vehicle Bounding Boxes and Lane Trajectory subsystems, see Lane and Vehicle Detection in Simulink Using Deep Learning.
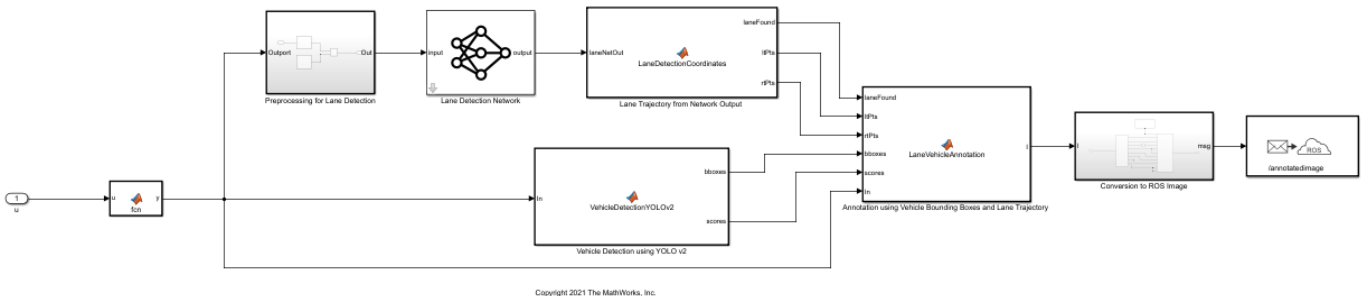
Run Simulation

Initialize the ROS master.

```matlab
rosinit;
```

Open the Simulink models.

```matlab
open_system('rosImagePublisher');
open_system('annotatedImageDisplay');
open_system('laneAndVehicleDetection');
```

Open the Configuration Parameters dialog box of the `laneAndVehicleDetection` model. In the **Simulation Target** pane, select **GPU acceleration**. In the **Deep Learning** section, select the target library as **cuDNN**. Alternatively, you can set the parameters using the following commands.

```
set_param('laneAndVehicleDetection','GPUAcceleration','on');
set_param('laneAndVehicleDetection','SimDLTargetLibrary','cudnn');
set_param('laneAndVehicleDetection','DLTargetLibrary','cudnn');
```
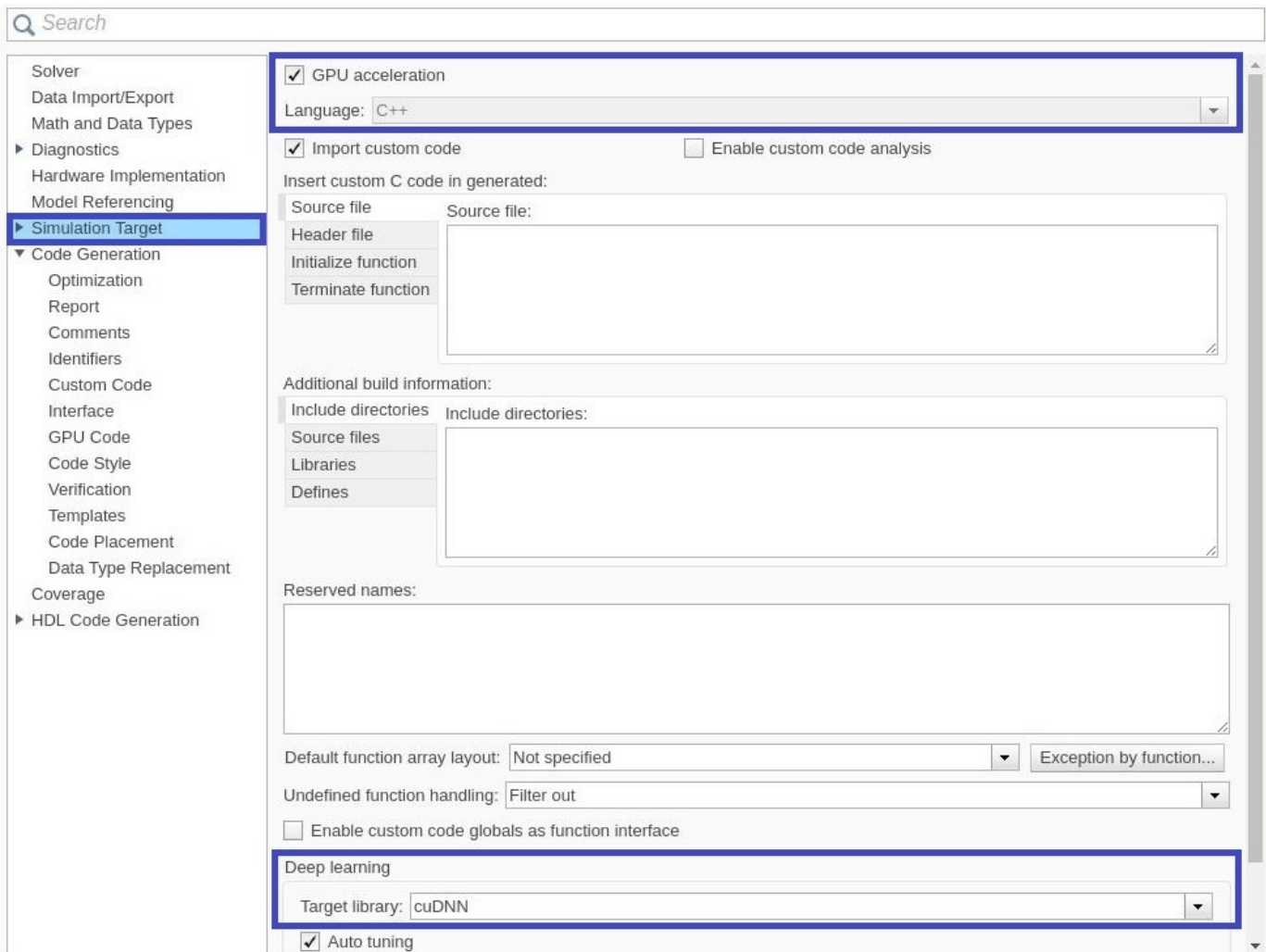

First run the models, `laneAndVehicleDetection` and `annotatedImageDisplay`, by selecting **Run** from the from **Simulation** tab. Then, run `rosImagePublisher`, which starts publishing video frames to ROS network. Every new frame received by `annotatedImageDisplay` is updated in the **Video Viewer** window.
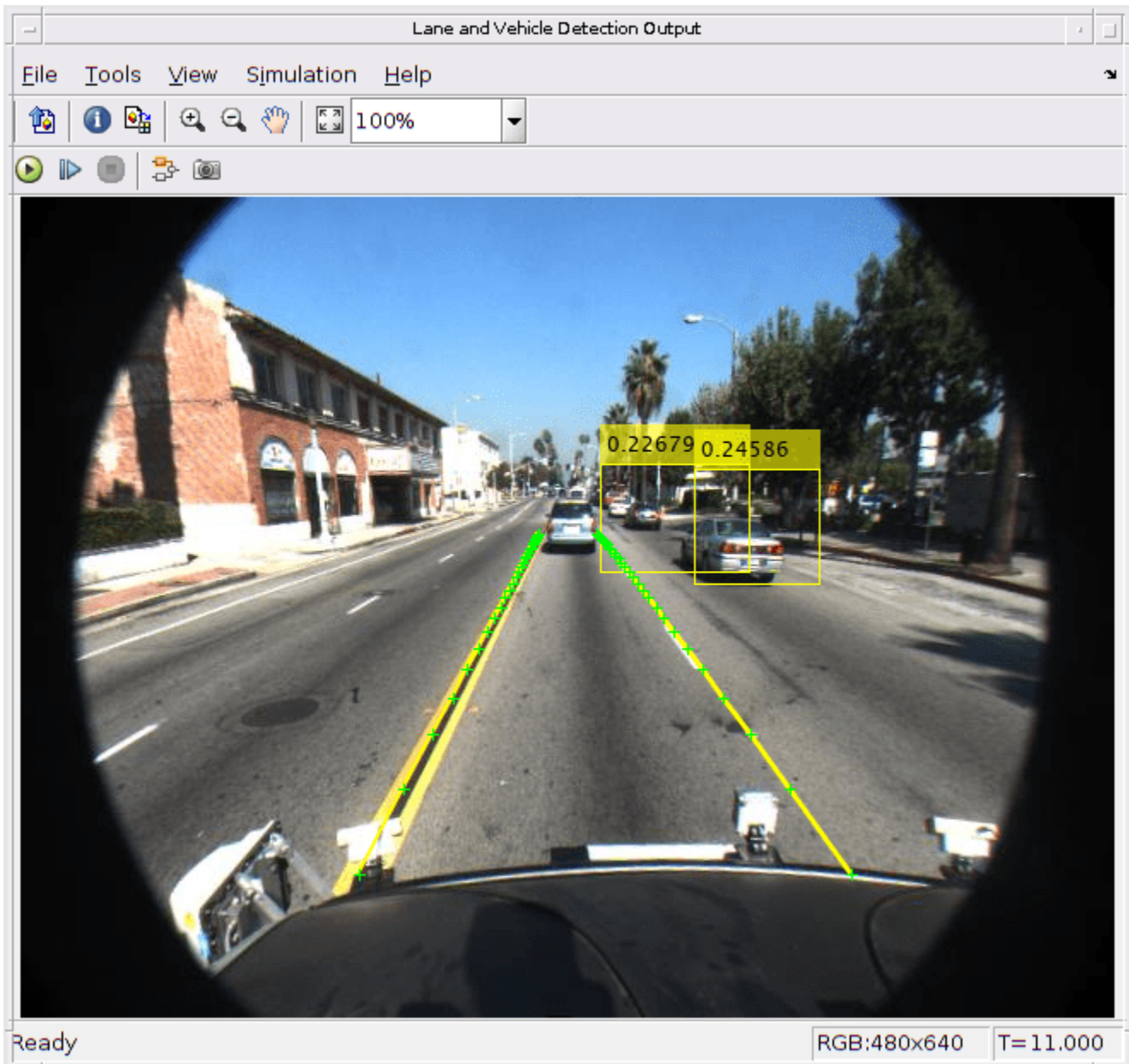
Generate and Build Simulink Model

In the **Hardware Implementation** pane, select **Robot Operating System (ROS)** for **Hardware Board** and specify **Device Vendor** and **Device Type**.
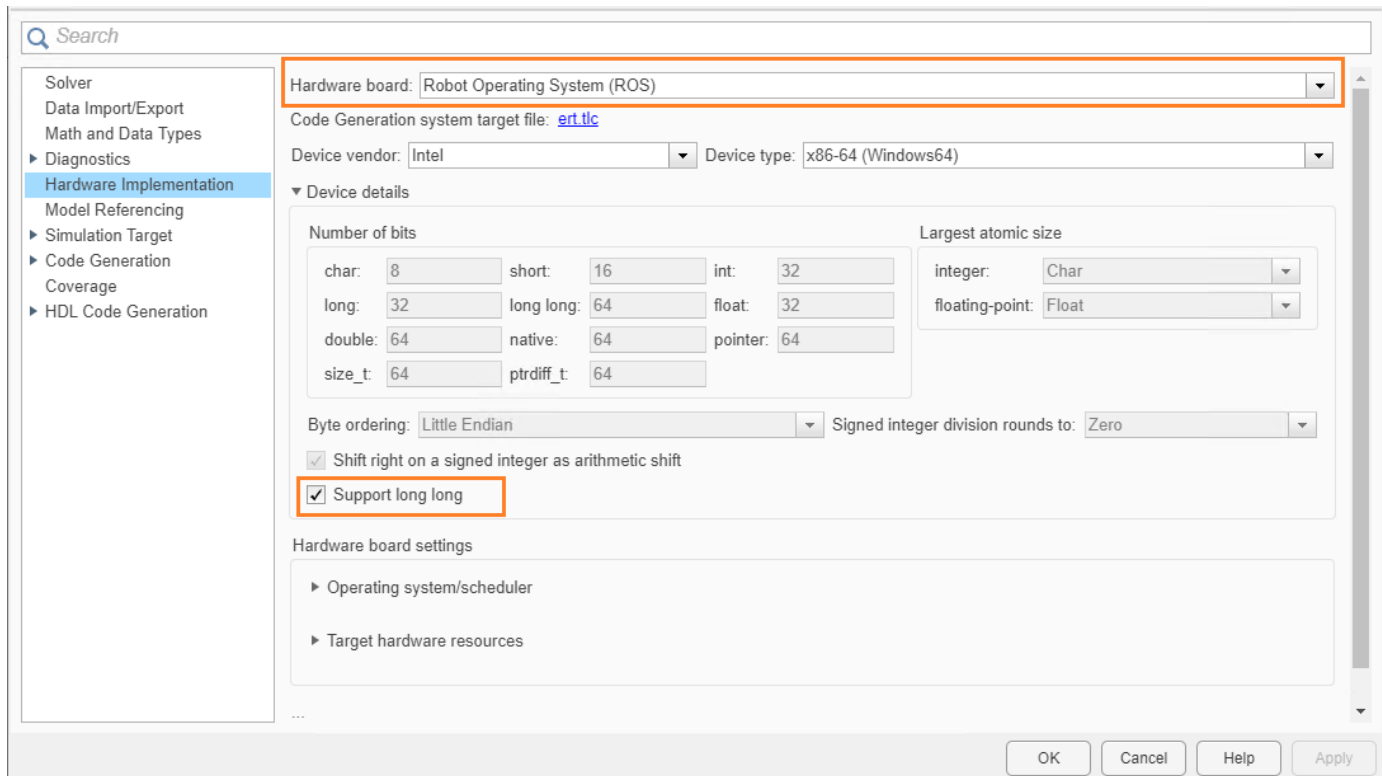
In the **Code Generation** pane, select the **Language** as **C++** and enable **Generate GPU code**.

```
set_param('laneAndVehicleDetection','TargetLang','C++');
set_param('laneAndVehicleDetection','GenerateGPUCode','CUDA');
```

In the **Code Generation > Libraries > GPU Code** pane, enable **cuBLAS**, **cuSOLVER** and **cuFFT**.

```
set_param('laneAndVehicleDetection','GPUcuBLAS','on');
set_param('laneAndVehicleDetection','GPUcuSOLVER','on');
set_param('laneAndVehicleDetection','GPUcuFFT','on');
```

To configure ROS node deployment on your local host machine, in the **Connect** section of the **ROS** tab, set **Deploy to** `Localhost`. Click the **Build and Run** button from ROS tab to deploy the node. At the end of the build process, you will see the ROS node running on local host machine. Then, run `annotatedImageDisplay` model, you should be able to view the annotated image video in the **Video Viewer** output.

Related Topics

- "Lane Detection Optimized with GPU Coder" (GPU Coder)
- Object Detection Using YOLO v2 Deep Learning
- Lane and Vehicle Detection in Simulink Using Deep Learning

Generate CUDA ROS Node from Simulink

Configure Simulink® Coder™ to generate and build a CUDA® ROS node from a Simulink model. You configure the model to simulate and generate CUDA code for ROS node. You then deploy the CUDA ROS node to local or remote device targets.

Prerequisites

- Install and set up these MathWorks® products and support packages: Simulink Coder (required), Embedded Coder® (recommended), GPU Coder™, and GPU Coder Interface for Deep Learning (required for deep learning).
- You can also install MATLAB® Coder™ Support Package for NVIDIA® Jetson™ and NVIDIA DRIVE™ Platforms (required for NVIDIA hardware connection).
- Development computer with CUDA-enabled NVIDIA GPU.
- Your target device can be a local host computer with a CUDA-enabled NVIDIA GPU, or a remote device such as an NVIDIA Jetson board.
- To use GPU Coder for CUDA code generation, you must install NVIDIA graphics driver, CUDA toolkit, cuDNN library, and TensorRT library. For more information, see "Installing Prerequisite Products" (GPU Coder).
- To set up the environment variables, see "Setting Up the Prerequisite Products" (GPU Coder).
- To ensure you have the required third-party software, see "ROS Toolbox System Requirements".

Verify GPU Environment for GPU Code Generation

To verify that your development computer has the drivers, tools, libraries, and configuration required for GPU code generation, enter these commands in the MATLAB Command Window:

```
gpuEnvObj = coder.gpuEnvConfig;
gpuEnvObj.BasicCodegen = 1;
gpuEnvObj.BasicCodeexec = 1;
gpuEnvObj.DeepLibTarget = "tensorrt";
gpuEnvObj.DeepCodeexec = 1;
gpuEnvObj.DeepCodegen = 1;
results = coder.checkGpuInstall(gpuEnvObj)
```

The output is representative. Your results might differ.

```
Compatible GPU            : PASSED
CUDA Environment          : PASSED
    Runtime   : PASSED
    cuFFT     : PASSED
    cuSOLVER  : PASSED
    cuBLAS    : PASSED
cuDNN Environment         : PASSED
TensorRT Environment      : PASSED
Basic Code Generation     : PASSED
Basic Code Execution      : PASSED
Deep Learning (TensorRT) Code Generation: PASSED
Deep Learning (TensorRT) Code Execution: PASSED

results =

  struct with fields:
```

```
          gpu: 1
         cuda: 1
        cudnn: 1
     tensorrt: 1
  basiccodegen: 1
 basiccodeexec: 1
   deepcodegen: 1
  deepcodeexec: 1
tensorrtdatatype: 1
     profiling: 0
```

Configure Simulink Model for Simulation and GPU Code Generation

Configure a model to generate CUDA ROS Node.

- Open the Simulink model you want to configure for GPU code generation.
- From the **Simulation** tab, in the **Prepare** section, expand the gallery and, under **Configuration & Simulation,** select **Model Settings**.
- From the left pane of the Configuration Parameters dialog box, select the **Hardware Implementation** node. Set **Hardware Board** to `Robot Operating System (ROS)` and specify the **Device Vendor** and **Device Type** for your hardware. Expand **Device details** and verify that **Support long long** is selected.

- In the **Simulation Target** node, select **GPU acceleration**. If your model has deep learning blocks, then, under **Deep Learning**, select the appropriate **Target library**.

- Select the **Code Generation** node. Under **Target selection**, set **Language** to C++, and select **Generate GPU code**.

- If your model has deep learning blocks, expand the **Code Generation** node and select the **Interface** node. Then, under in **Deep Learning** select the appropriate value for **Target library**.

- Select the **GPU Code** node and, under **Libraries**, enable **cuBLAS**, **cuSOLVER**, and **cuFFT**.

Code Generation and Deployment

To generate and deploy a CUDA ROS node to a ROS device, follow the steps in the "Generate a Standalone ROS Node from Simulink" on page 1-195 example.

Limitations

- Model reference is not supported.
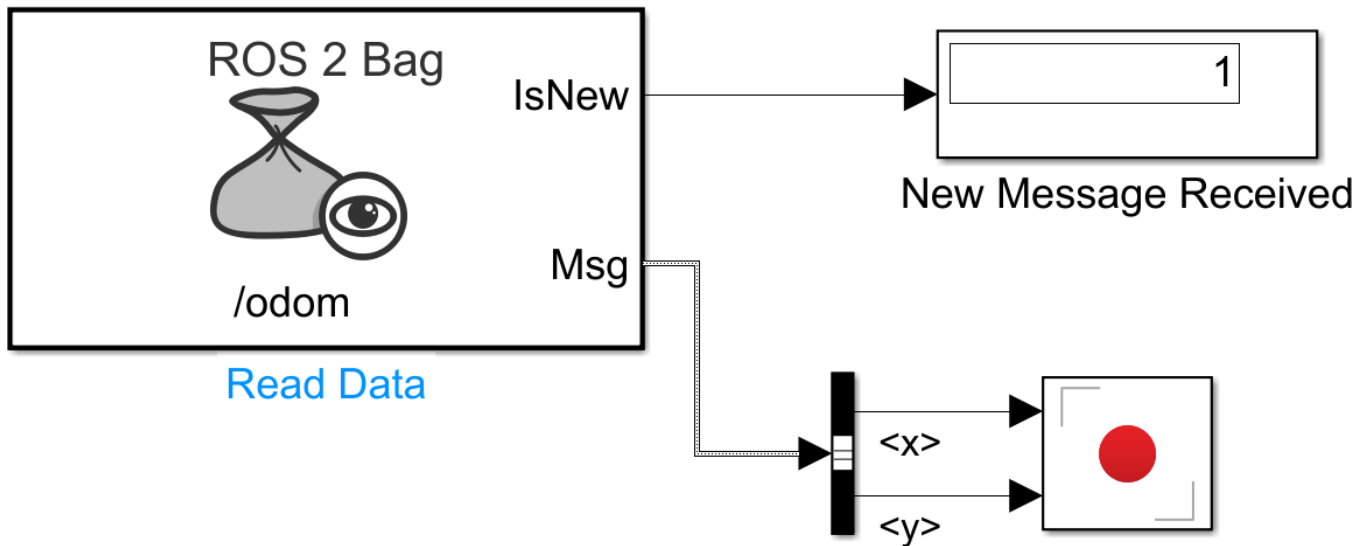- The build folder path cannot contain any spaces.

Related Topics

- "Lane and Vehicle Detection in ROS Using YOLO v2 Deep Learning Algorithm" on page 8-2
- "Sign Following Robot Using YOLOv2 Detection Algorithm with ROS in Simulink" on page 2-117

Play Back Data from ROS 2 Bag Logfile in Simulink®

Use the Read Data block to play back data from a ROS 2 bag logfile.

Load Model

Copyright 2021 The MathWorks, Inc.

```
open_system("read_pose_log.slx")
```

Set up Model

1 Open the **Read Data** block mask to load a ROS 2 bag logfile (`.db3`).
2 Click the **Load logfile data** link.
3 Browse for the logfile and specify a time offset or limited duration if needed. The `ros2_bag.db3` file and the associated `metadata.yaml` file are attached to this example.
4 Select the desired topic, `/odom` which contains `nav_msgs/Odometry` messages.
5 The **Read Data** block outputs the messages from the ROS 2 bag logfile.
6 A bus selector extracts the *xy*-position from the `nav_msgs/Odometry` messages.

Run Model

The block plays back data in sync with the simulation time. The **Record** plot displays the robot position over time.

```
sim(gcs)
```
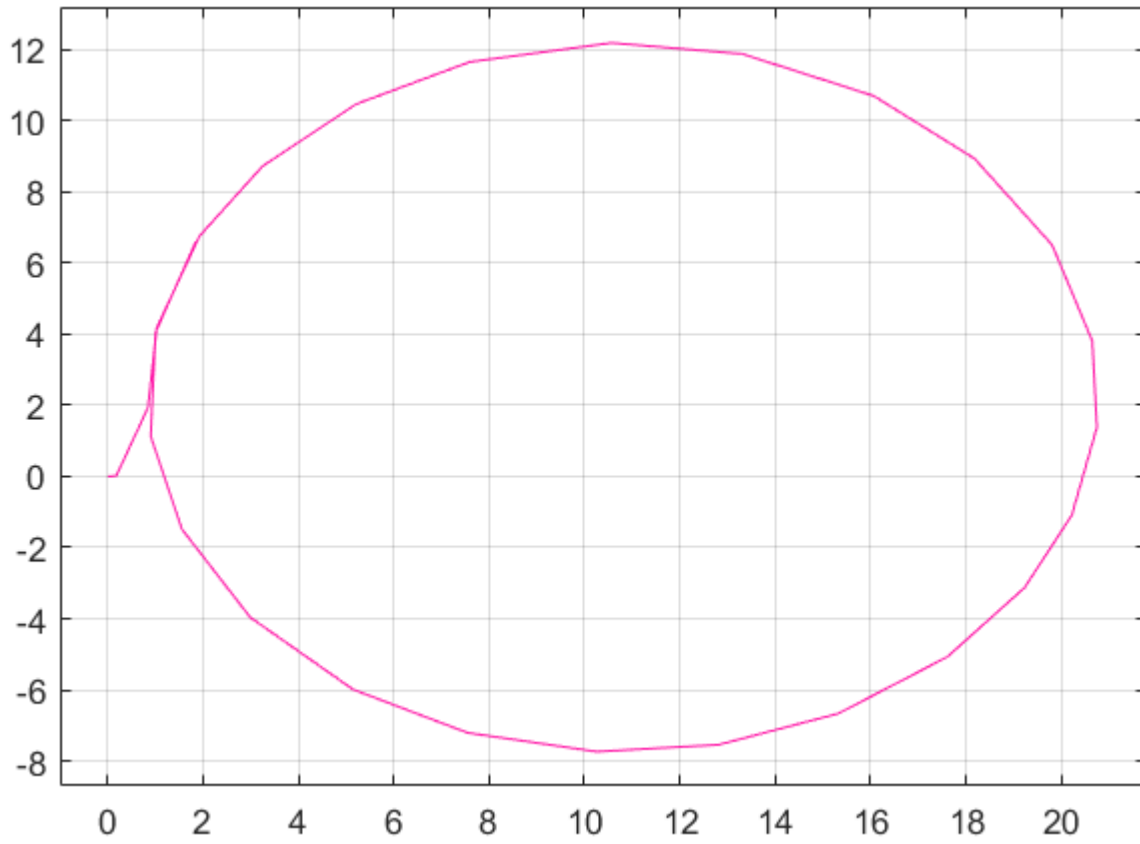
ROS 2 CUDA Simulink Topics

Generate CUDA ROS 2 Node from Simulink

Configure Simulink® Coder™ to generate and build a CUDA® ROS 2 node from a Simulink model. You configure the model to simulate and generate CUDA code for ROS 2 node. You then deploy the CUDA ROS 2 node to local or remote device targets.

Prerequisites

- Install and set up these MathWorks® products and support packages: Simulink Coder (required), Embedded Coder® (recommended), GPU Coder™, and GPU Coder Interface for Deep Learning (required for deep learning).
- You can also install MATLAB® Coder™ Support Package for NVIDIA® Jetson™ and NVIDIA DRIVE™ Platforms (required for NVIDIA hardware connection).
- Development computer with CUDA-enabled NVIDIA GPU.
- Your target device can be a local host computer with a CUDA-enabled NVIDIA GPU, or a remote device such as an NVIDIA Jetson board.
- To use GPU Coder for CUDA code generation, you must install NVIDIA graphics driver, CUDA toolkit, cuDNN library, and TensorRT library. For more information, see "Installing Prerequisite Products" (GPU Coder).
- To set up the environment variables, see "Setting Up the Prerequisite Products" (GPU Coder).
- To ensure you have the required third-party software, see "ROS Toolbox System Requirements".

Verify GPU Environment for GPU Code Generation

To verify that your development computer has the drivers, tools, libraries, and configuration required for GPU code generation, see "Setting Up the Prerequisite Products" (GPU Coder).

In the MATLAB Command Window, enter:

```
gpuEnvObj = coder.gpuEnvConfig;
gpuEnvObj.BasicCodegen = 1;
gpuEnvObj.BasicCodeexec = 1;
gpuEnvObj.DeepLibTarget = "tensorrt";
gpuEnvObj.DeepCodeexec = 1;
gpuEnvObj.DeepCodegen = 1;
results = coder.checkGpuInstall(gpuEnvObj)
```

The output is representative. Your results might differ.

```
Compatible GPU            : PASSED
CUDA Environment          : PASSED
    Runtime   : PASSED
    cuFFT     : PASSED
    cuSOLVER  : PASSED
    cuBLAS    : PASSED
cuDNN Environment         : PASSED
TensorRT Environment      : PASSED
Basic Code Generation     : PASSED
Basic Code Execution      : PASSED
Deep Learning (TensorRT) Code Generation: PASSED
Deep Learning (TensorRT) Code Execution: PASSED

results =
```
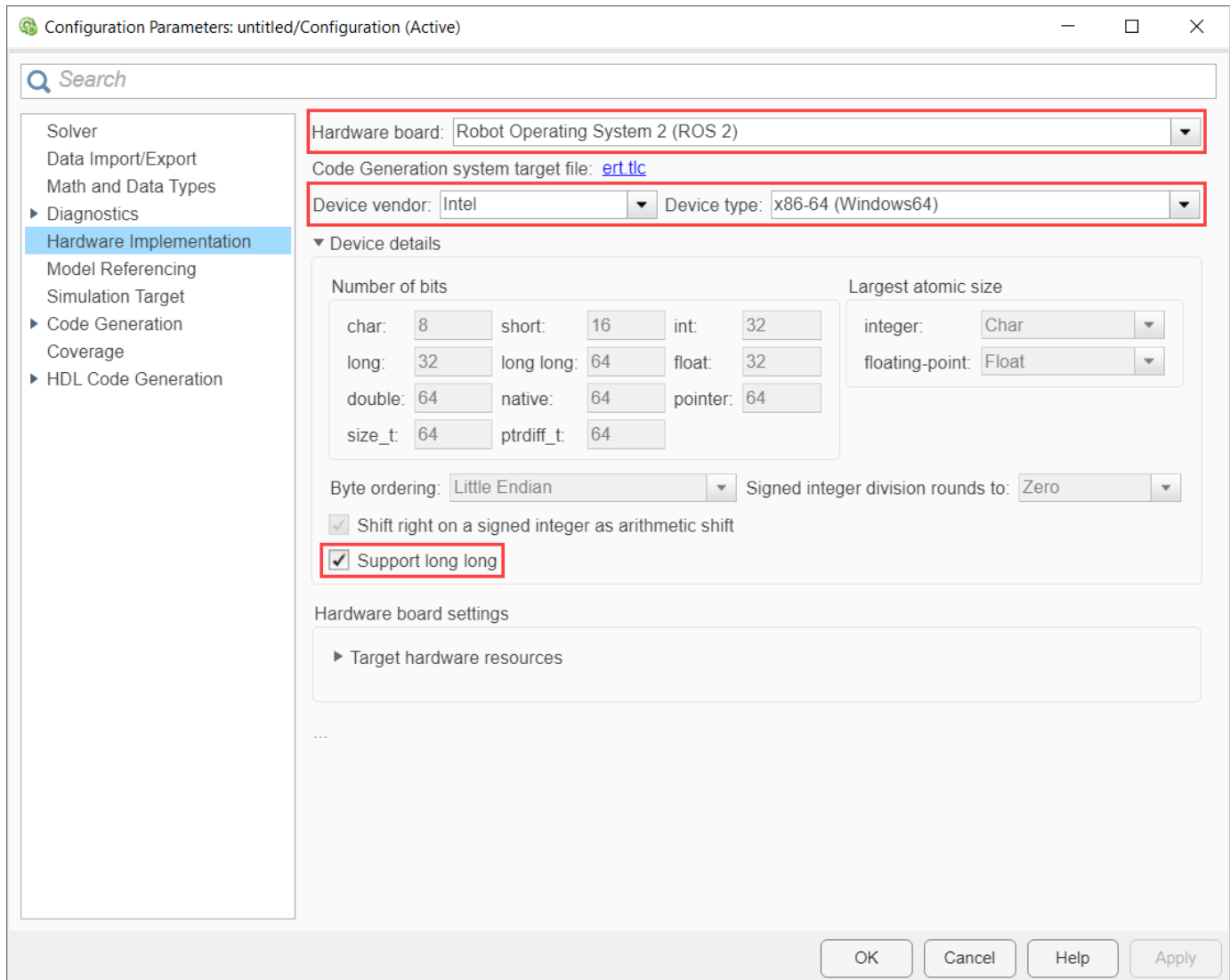
```
struct with fields:

                gpu: 1
               cuda: 1
              cudnn: 1
           tensorrt: 1
       basiccodegen: 1
      basiccodeexec: 1
        deepcodegen: 1
       deepcodeexec: 1
  tensorrtdatatype: 1
          profiling: 0
```

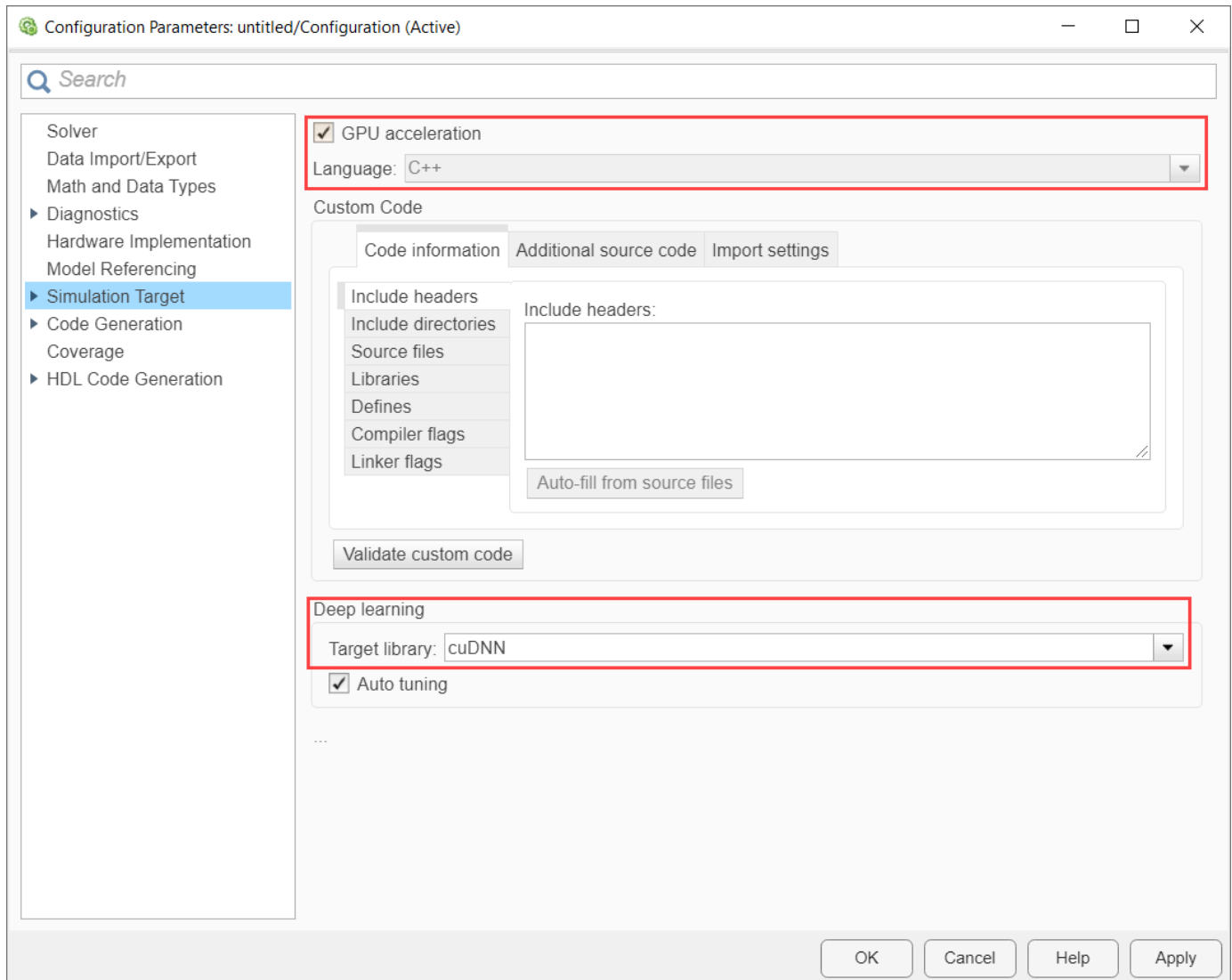**Configure Simulink Model for Simulation and GPU Code Generation**

Configure a model to generate CUDA ROS 2 Node.

- Open the Simulink model you want to configure for GPU code generation.
- From the **Simulation** tab, in the **Prepare** section, expand the gallery and, under **Configuration & Simulation,** select **Model Settings**.
- From the left pane of the Configuration Parameters dialog box, select the **Hardware Implementation** node. Set **Hardware Board** to Robot Operating System 2 (ROS 2) and specify the **Device Vendor** and **Device Type** for your hardware. Expand **Device details** and verify that **Support long long** is selected.
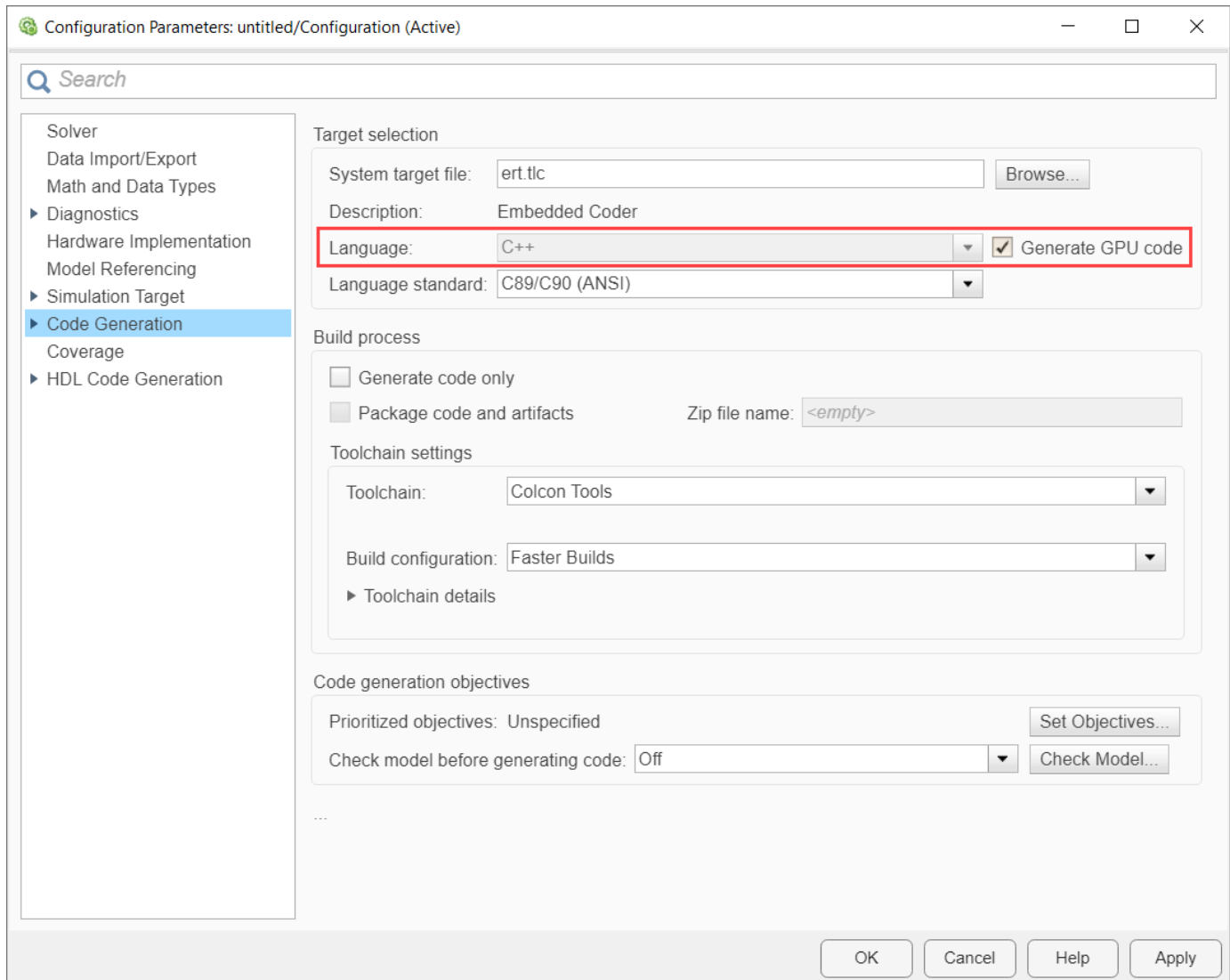
- In the **Simulation Target** node, select **GPU acceleration**. If your model has deep learning blocks, then, under **Deep Learning**, select the appropriate **Target library**.
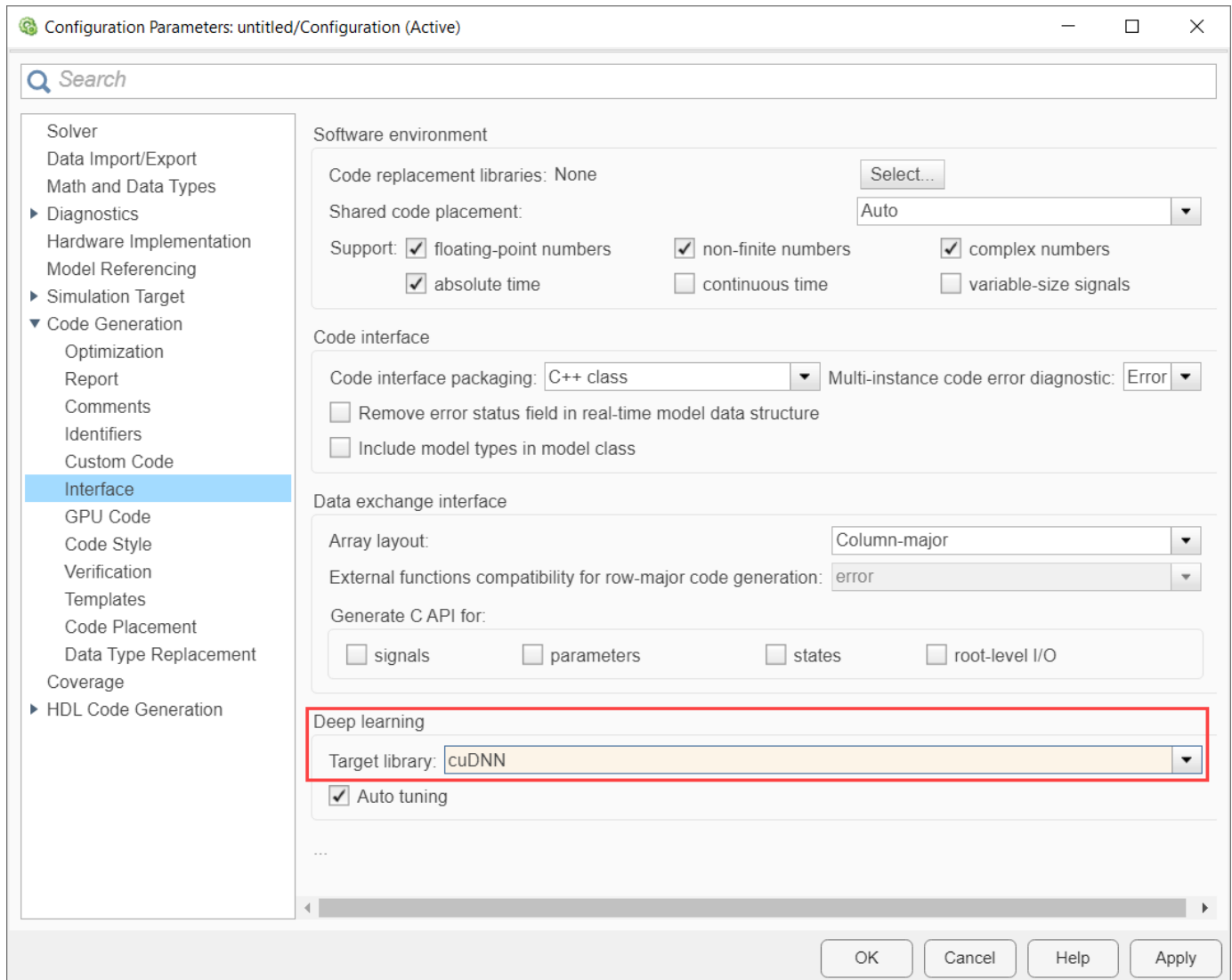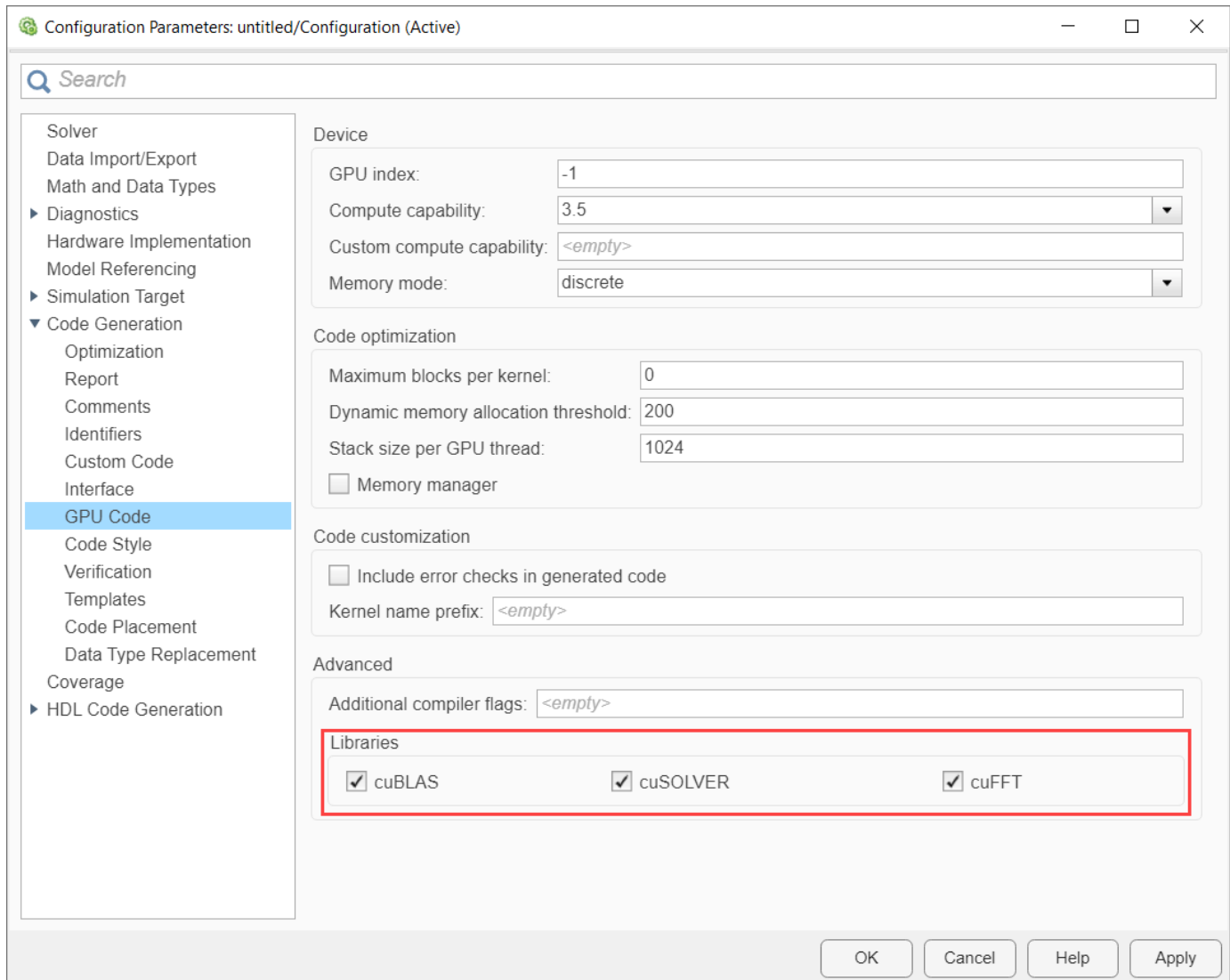
- Select the **Code Generation** node. Under **Target selection**, set **Language** to C++, and select **Generate GPU code**.

- If your model has deep learning blocks, expand the **Code Generation** node and select the **Interface** node. Then, under in **Deep Learning** select the appropriate value for **Target library**.

- Select the **GPU Code** node and, under **Libraries**, enable **cuBLAS**, **cuSOLVER**, and **cuFFT**.

Code Generation and Deployment

To generate and deploy a CUDA ROS 2 node to a ROS 2 device, follow the steps in the "Generate a Standalone ROS 2 Node from Simulink" on page 2-99 example.

Limitations

- Model reference is not supported.
- The build folder path cannot contain any spaces.

ROS 2 Simulink Topics

Work with ROS 2 Messages in Simulink

This example illustrates how to work with complex ROS 2 messages in Simulink®, such as messages with nested sub-messages and variable-length arrays.

Introduction

In ROS 2 Simulink Models, bus signals represent ROS 2 Messages. Each field of a ROS 2 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.
- **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 2 message, the message fields restore to their native ROS 2 types. For example, the ROS message type `std_msgs/Header` has a field, `frame_id`, which is a string. In the corresponding Simulink bus, the `frame_id` field is a `uint8` array. When the bus converts to a ROS message, `frame_id` converts back to a string.
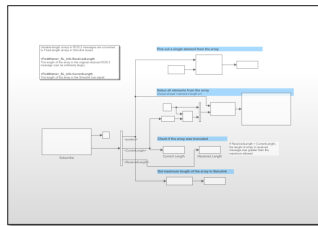
Model

The following model has several examples of working with complex ROS 2 messages in Simulink. The rest of the tasks in this example focus on specific scenarios.

```
open_system('robotROS2MessageUsageExample');
```
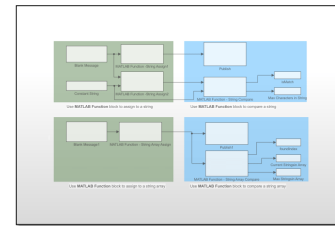
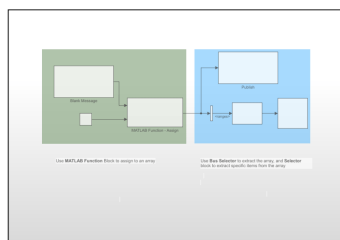**Working with ROS 2 messages in Simulink**
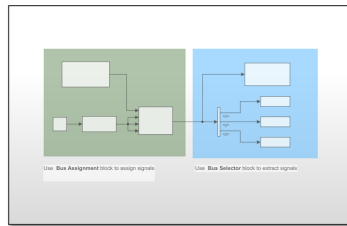
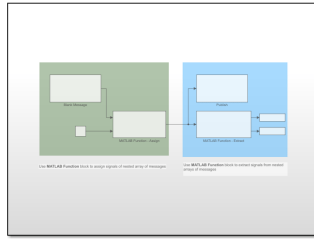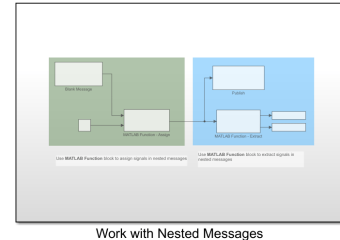Work with Nested Messages

Work with Variable-length Arrays

Work with Strings and String Arrays

Work with Numeric Arrays

Work with
Nested Arrays of Messages

Work with Nested Messages
using MATLAB Function Block

MATLAB Function blocks require their
output buses to be created explicitly. Use
ros.ros2.createSimulinkBus to do this.

Click here to run
ros.ros2.createSimulinkBus

Access Data in a Variable-length Array

A ROS 2 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('robotROS2MessageUsageExample/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 2 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 2 Network

- Under the **Simulation** tab, select **ROS Network** from the **Prepare** section. If you do not see ROS Toolbox, select **Robot Operating System (ROS)** on the Apps tab, under **Control Systems**. In the dialog box that opens up, select **Robot Operating System 2 (ROS 2)** from the **ROS Network** drop-down.

- Set the **Domain ID** and **RMW Implementation** to desired values.

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.

```
node = ros2node("/my_node");
[pub, msg] = ros2publisher(node,"/my_joint_state", "sensor_msgs/JointState",Durability="transient
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 using the name `SL_Bus_<messageType>` and stored in the Simulink data dictionary `ros2lib`. You can find this data dictionary under **External Data > From Libraries** in **Model Explorer**.

```
ros.ros2.createSimulinkBus(gcs)
```

- Double-click the **MATLAB Function - Assign** block. Double-click the **MATLAB Function - Assign** block. In the MATLAB Editor toolstrip, under **Modeling** tab, click **Symbols Pane**. Then right click on `msg`, select **Inspect** and set its type to `SL_Bus_geometry_msgs_PoseArray`. Click **Apply** and close *Ports and Data Manager*.
- If you do not see `SL_Bus_geometry_msgs_PoseArray` listed as an option in the **Type** dropdown, select **Refresh data types**.
- If you are using variable-size messages within MATLAB Function block, you must enable **Dynamic memory allocation in MATLAB functions** setting under **Model Settings > Simulation Target > Advanced parameters**.

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.ros2.createSimulinkBus(gcs)` statement has to be re-run each time the model is loaded. 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.

# Control a Simulated UAV Using ROS 2 and PX4 Bridge

This example demonstrates how to receive sensor readings and autopilot status from a simulated UAV with PX4 autopilot, and send control commands to navigate the simulated UAV.

**Set Up Simulation Environment**

In MATLAB®, generate custom messages for the `px4_msgs` package. You can add this to an existing custom message folder and regenerate it, or generate them in this example folder. This can take some time, as there are a large number of messages to generate.

```
unzip("px4_msgs.zip")
ros2genmsg("custom")
```

Download and connect to the vistual machine (VM) using instructions in "Get Started with Gazebo and Simulated TurtleBot" on page 1-206. On the VM, start the simulator and PX4 Bridge by clicking the "Gazebo PX4 SITL RTPS" desktop shortcut. This brings up a simple simulator of a quadrotor in Gazebo.
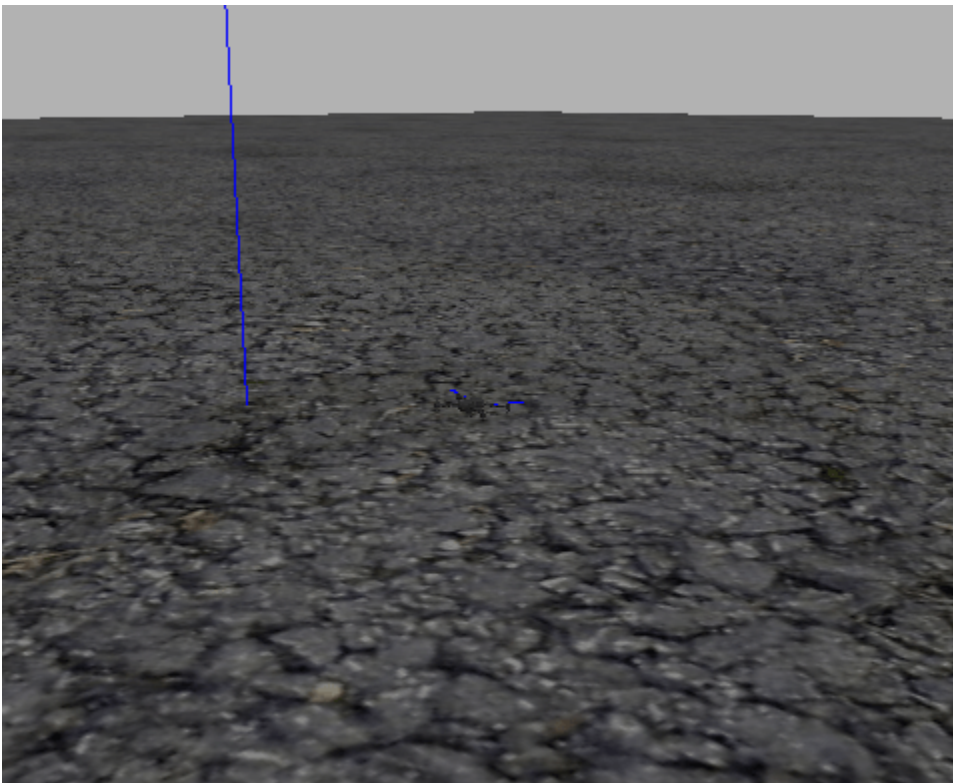
This also brings up the *Agent* half of the `microRTPS` bridge. The *Client* half of the communication is included in the PX4 Autopilot packages, and launches with the simulator.
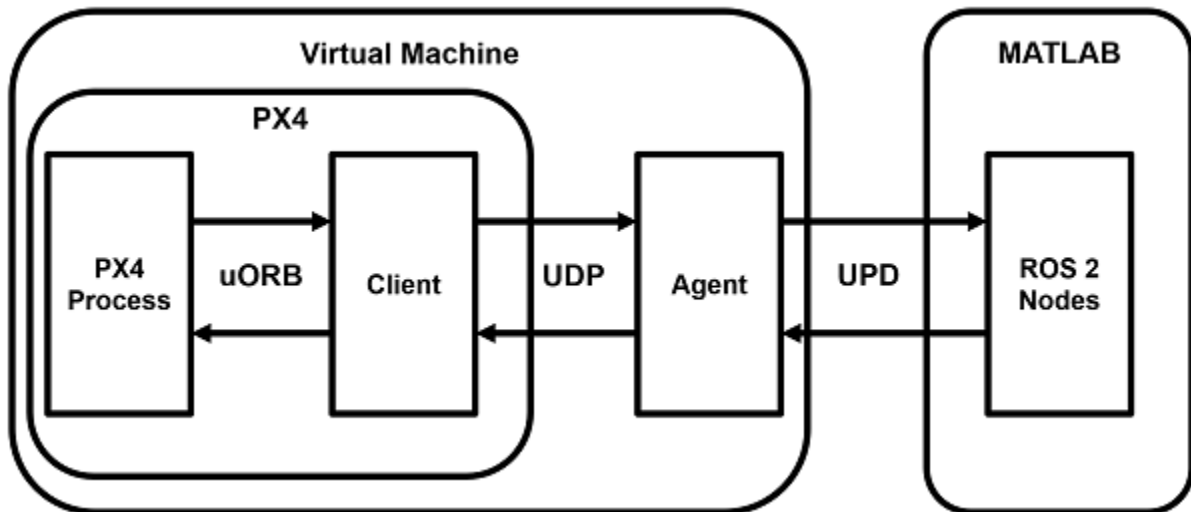
You can enter PX4 commands on the terminal corresponding to the Gazebo simulation, but that is not required for this example.

Set Up ROS 2 Network

View the available topics that are broadcast by the bridge Agent. Notice the convention it uses to indicate the topics it is publishes to (`out`) and those it is subscribing to (`in`).

```
ros2 topic list
```

```
/LaserScan/out
/clock
/fmu/collision_constraints/out
/fmu/debug_array/in
/fmu/debug_key_value/in
/fmu/debug_value/in
/fmu/debug_vect/in
/fmu/offboard_control_mode/in
/fmu/onboard_computer_status/in
/fmu/optical_flow/in
/fmu/position_setpoint/in
/fmu/position_setpoint_triplet/in
/fmu/sensor_combined/out
/fmu/telemetry_status/in
/fmu/timesync/in
/fmu/timesync/out
/fmu/trajectory_bezier/in
/fmu/trajectory_setpoint/in
/fmu/trajectory_waypoint/out
/fmu/vehicle_command/in
/fmu/vehicle_control_mode/out
/fmu/vehicle_local_position_setpoint/in
/fmu/vehicle_mocap_odometry/in
/fmu/vehicle_odometry/out
/fmu/vehicle_status/out
/fmu/vehicle_trajectory_bezier/in
/fmu/vehicle_trajectory_waypoint/in
/fmu/vehicle_trajectory_waypoint_desired/out
/fmu/vehicle_visual_odometry/in
```

```
/parameter_events
/rosout
/timesync_status
```

Create a node to handle the sensor input and control feedback for the UAV. Create subscribers for sensors and information of interest, and publishers to direct the UAV in offboard control mode.

```
node = ros2node("/control_node");

% General information about the UAV system
controlModePub = ros2publisher(node,"fmu/offboard_control_mode/in","px4_msgs/OffboardControlMode"
statusSub = ros2subscriber(node,"/fmu/vehicle_status/out","px4_msgs/VehicleStatus");
timeSub = ros2subscriber(node,"fmu/timesync/out","px4_msgs/Timesync");

% Sensor and control communication
odomSub = ros2subscriber(node,"/fmu/vehicle_odometry/out","px4_msgs/VehicleOdometry");
setpointPub = ros2publisher(node,"fmu/trajectory_setpoint/in","px4_msgs/TrajectorySetpoint");
cmdPub = ros2publisher(node,"/fmu/vehicle_command/in","px4_msgs/VehicleCommand");
```

Get the system and component IDs from the UAV status. These help direct the commands to the UAV. This also ensures the UAV is up and running before moving into the control phase.

```
timeLimit = 5;
statusMsg = receive(statusSub,timeLimit);
receive(odomSub,timeLimit);
receive(timeSub,timeLimit);
```

Test MATLAB Communication with Gazebo instance

Use the previously created publisher and subscriber to instruct the UAV to take off, fly to a new point, and then land. To perform offboard control, the UAV requires control messages to be flowing regularly (at least 2 Hz). Starting the control messages before engaging offboard mode is the easiest way to achieve this.

```
% Start flow of control messages
tStart = tic;
xyz = [0 0 -1]; % NED coordinates - negative Z is higher altitude
while toc(tStart) < 1
    publishOffboardControlMode(timeSub,controlModePub,"position")
    publishTrajectorySetpoint(timeSub,setpointPub,xyz)
    pause(0.1)
end

% Instruct the UAV to accept offboard commands
% Arm the UAV so it allows flying
engageOffboardMode(timeSub,cmdPub)
arm(timeSub,cmdPub)

% Navigate to a nearby location and hover there
while toc(tStart) < 21
    publishOffboardControlMode(timeSub,controlModePub,"position")
    publishTrajectorySetpoint(timeSub,setpointPub,xyz)
    pause(0.1)
end

% Land once complete
land(timeSub,cmdPub)
```
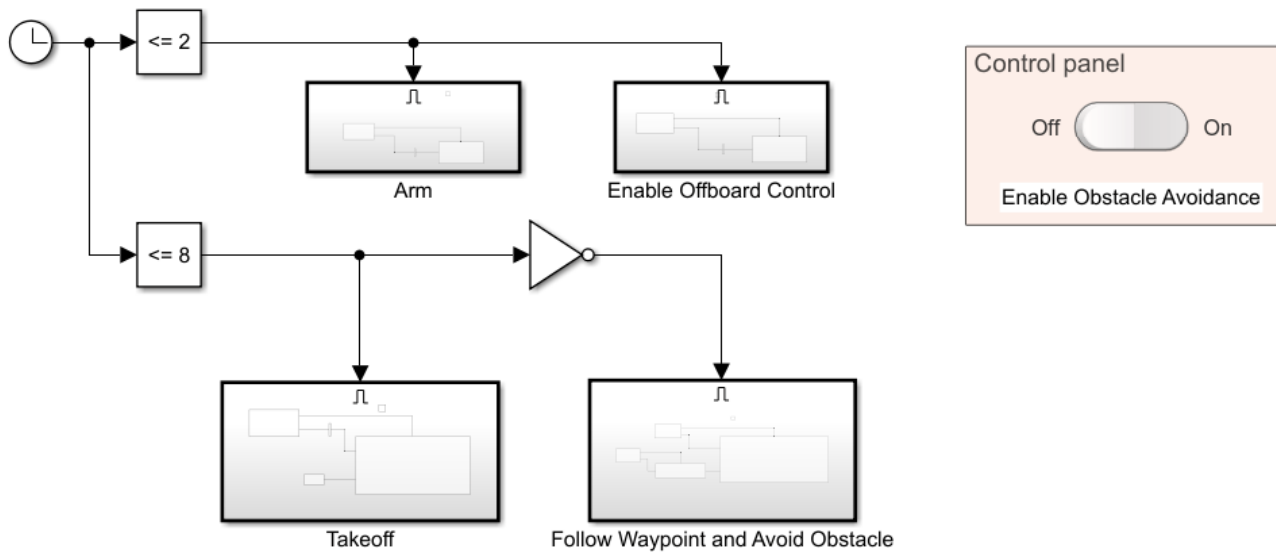
Control UAV from Simulink

You can also control the UAV using the Simulink® model `ControlUAVUsingROS2`. The Simulink model includes four main subsystems: Arm, Offboard, Takeoff, and Waypoint following with obstacle avoidance.

```
open_system("ControlUAVUsingROS2.slx");
```


Control a UAV Using PX4-ROS2 Bridge

Copyright 2021 The MathWorks, Inc.

A Clock block triggers different UAV activity according to the simulation time. The model arms and enables the offboard control mode on the PX4 autopilot for the first 2 seconds. It sends out a takeoff command to bring the UAV to 1 meter above the ground for the first 8 seconds, then it engages in waypoint-following behavior.
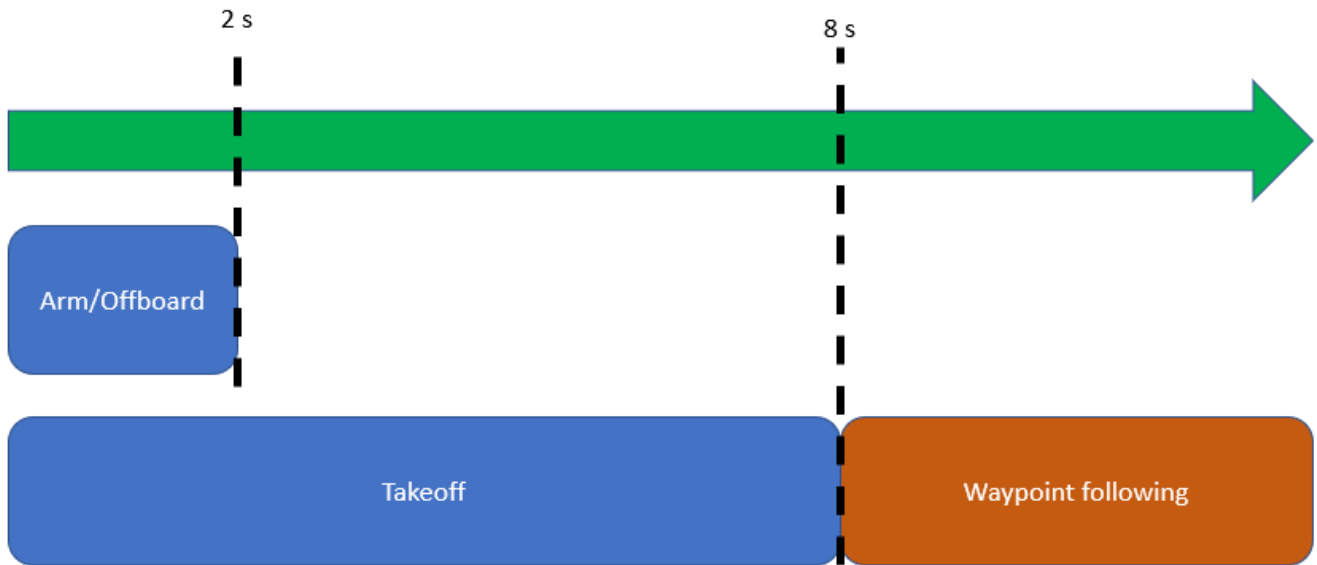
You can use the control panel to enable or disable the obstacle avoidance behavior.
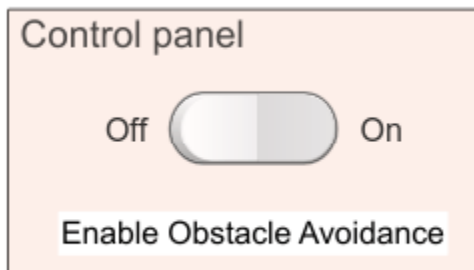
If obstacle avoidance is turned off, the UAV flies towards the goal point behind the obstacle box on a straight path, resulting in a collision. If obstacle avoidance is turned on, the UAV tries to fly above the obstacle based on the lidar sensor readings.
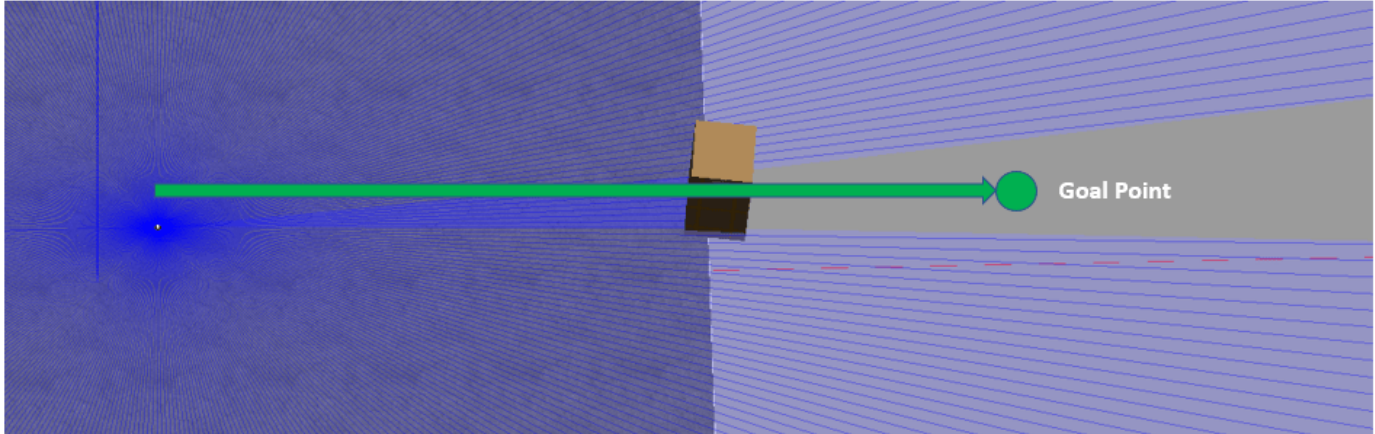
Once you start a fresh PX4-SITL and Gazebo instance in the VM, you can run the model and observe the flight behavior of the UAV.

Helper Functions

```matlab
function arm(timeSub,cmdPub)
% Allow UAV flight
    cmdMsg = ros2message(cmdPub);
    cmdMsg.command = uint32(cmdMsg.VEHICLE_CMD_COMPONENT_ARM_DISARM);
    cmdMsg.param1 = single(1);
    publishVehicleCommand(timeSub,cmdPub,cmdMsg)
end

function land(timeSub,cmdPub)
% Land the UAV
    cmdMsg = ros2message(cmdPub);
    cmdMsg.command = uint32(cmdMsg.VEHICLE_CMD_NAV_LAND);
    publishVehicleCommand(timeSub,cmdPub,cmdMsg)
end

function engageOffboardMode(timeSub,cmdPub)
% Allow offboard control messages to be utilized
    cmdMsg = ros2message(cmdPub);
    cmdMsg.command = uint32(cmdMsg.VEHICLE_CMD_DO_SET_MODE);
    cmdMsg.param1 = single(1);
    cmdMsg.param2 = single(6);
    publishVehicleCommand(timeSub,cmdPub,cmdMsg)
end

function publishOffboardControlMode(timeSub,controlModePub,controlType)
% Set the type of offboard control to be used
% controlType must match the field name in the OffboardControlMode message
    modeMsg = ros2message(controlModePub);
    modeMsg.timestamp = timeSub.LatestMessage.timestamp;
    % Initially set all to false
    modeMsg.position = false;
    modeMsg.velocity = false;
    modeMsg.acceleration = false;
    modeMsg.attitude = false;
    modeMsg.body_rate = false;
```

```matlab
    % Override desired control mode to true
    modeMsg.(controlType) = true;
    send(controlModePub,modeMsg)
end

function publishTrajectorySetpoint(timeSub,setpointPub, xyz)

    setpointMsg = ros2message(setpointPub);
    setpointMsg.timestamp = timeSub.LatestMessage.timestamp;
    setpointMsg.x = single(xyz(1));
    setpointMsg.y = single(xyz(2));
    setpointMsg.z = single(xyz(3));
    send(setpointPub,setpointMsg)
end

function publishVehicleCommand(timeSub,cmdPub,cmdMsg)
    cmdMsg.timestamp = timeSub.LatestMessage.timestamp;
    cmdMsg.target_system = uint8(1);
    cmdMsg.target_component = uint8(1);
    cmdMsg.source_system = uint8(1);
    cmdMsg.source_component = uint8(1);
    cmdMsg.from_external = true;
    send(cmdPub,cmdMsg)
end
```

Enable ROS 2 Time Model Stepping for Deployed Nodes

You can enable a deployed ROS 2 node to execute based on the time published on the `/clock` topic on a ROS 2 network. To deploy a ROS 2 node from Simulink, see "Generate a Standalone ROS 2 Node from Simulink" on page 2-99.

When you enable ROS 2 time model stepping, the deployed node executes when the published ROS 2 time is a multiple of the base rate of the model. To enable model stepping based on ROS 2 time:

1 On the Apps tab, under **Control Systems**, click **Robot Operating System 2 (ROS 2)**.
2 In the **Robot Operating System 2 (ROS 2)** dialog box that opens up, select `Robot Operating System 2 (ROS 2)` from the **ROS Network** drop-down. This opens up the **ROS** tab in the toolstrip which shows the specified ROS Network in the **Connect** section.
3 In the **Prepare** section under **ROS** tab, click **Hardware Settings** to open the model configuration parameters dialog box. Under **Target Hardware resources > ROS 2 time**, select **Enable ROS 2 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 the ROS 2 time is published. The ROS 2 node publishes a `std_msgs/String` message type with a string containing a `'+'` or `'-'` and the model name (`+ros2time_test`, for example). A `'+'` indicates the model was stepped. A `'-'` indicates the published ROS 2 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 2 node, the model waits for the ROS 2 time to be published.

See Also
Current Time | Subscribe

Related Examples
- "Generate a Standalone ROS 2 Node from Simulink" on page 2-99
- "Get Started with ROS 2 in Simulink®" on page 2-71
- "Exchange Data with ROS 2 Publishers and Subscribers" on page 2-19

Read ROS 2 Image Messages in Simulink® and Perform Registration using Feature Matching

This example shows how to read ROS 2 `sensor_msgs/Image` messages in Simulink® and perform image registration using feature matching to estimate the relative rotation of one image with respect to the other. This example requires Computer Vision Toolbox®.

Set Up ROS 2 Network

Load two sample `sensor_msgs/Image` messages, imageMsg1 and imageMsg2. Create a ROS 2 node with two publishers to publish the messages on the topics /image_1 and /image2. For the publishers, set the quality of service (QoS) property `Durability` to `transientlocal`. This ensures that the publishers maintain the messages for any subscribers that join after the messages have already been sent. Publish the messages using the `send` function.

```
load('ros2ImageMsgs.mat');

node = ros2node('/image_test');
imgPub1 = ros2publisher(node,'/image1','sensor_msgs/Image',Durability='transientlocal',Depth=2);
imgPub2 = ros2publisher(node,'/image2','sensor_msgs/Image',Durability='transientlocal',Depth=2);

send(imgPub1,imageMsg1);
send(imgPub2,imageMsg2);
```

Read messages from ROS 2 Network in Simulink and Perform Image Registration

Open the model.

```
modelName = 'readImageAndEstimateTransformationExampleModel.slx';
open_system(modelName);
```
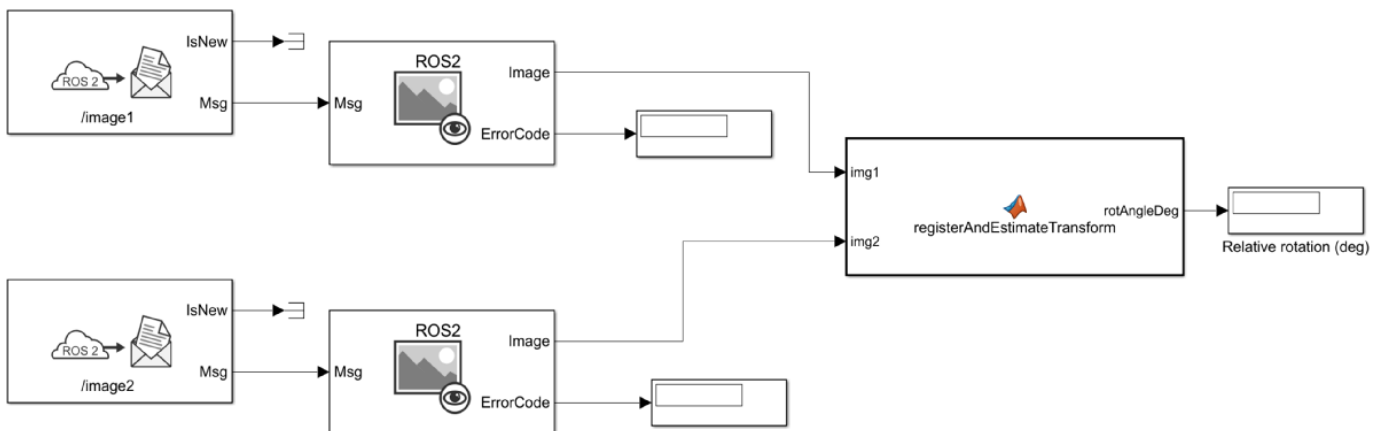

Copyright 2021 The MathWorks, Inc.

Ensure that the two Subscribe blocks are connected to the topics, /image1 and /image2. The value of the **Durability** QoS parameter of the Subscribe blocks is `Transient local`. This ensures compatibility with the QoS settings of the publishers that you configured in the previous section. The messages from the subscribers are fed to the Read Image blocks to extract the image data, which is

then fed to the `registerAndEstimateTransform` MATLAB Function block to perform registration using feature matching.

Under **Simulation** tab, select **ROS Toolbox > Variable Size Messages**. Notice that the **Use default limits for this message type** is clear. Then, in the **Maximum length** column, verify that the `data` property of the `sensor_msgs/Image` message has a value greater than the number of pixels in the image (921600). Run the model.

`sim(modelName);`

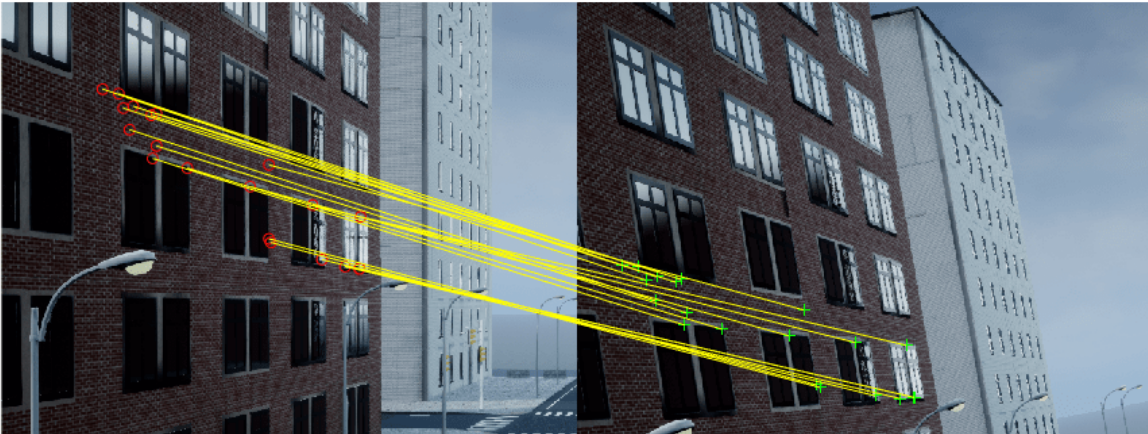Matching points in Image 1 (left) and Image 2 (inliers only)

Image 2 (right) transformed to the view of Image 1

The model performs image registration, estimates the 2D transformation and displays the relative rotation (deg) of the image published on `/image2` topic, with respect to that on `/image1`. It also visualizes the two images along with the matched inlier points and the `/image2` image transformed into the view of the `/image1` image.

MATLAB function for Image Registration using Feature Matching

This model uses the algorithm in the `ExampleHelperRegisterAndEstimateTransform` MATLAB Function to perform image registration and the relative transform estimation, consisting of these steps:

1 Detect and extract ORB features and corresponding valid points in the two images using `detectORBFeatures` (Computer Vision Toolbox) and `extractFeatures` (Computer Vision Toolbox).

2 Find candidate matches and collect the actual corresponding point locations from both the images using `matchFeatures` (Computer Vision Toolbox).

3 Estimate the relative geometric transformation between the two images along with inliers using `estimateGeometricTransform2D` (Computer Vision Toolbox), which implements the M-estimator sample consensus (MSAC), a variant of the random sample consensus (RANSAC) algorithm.

4 Compute and output the relative rotation from the estimated geometric transformation.

For more information about local feature detection and extraction such as different kinds of feature detectors, see "Local Feature Detection and Extraction" (Computer Vision Toolbox). Feature matching is also a key initial step in Visual SLAM applications. For an overview of the Visual SLAM workflow using MATLAB™, see "Implement Visual SLAM in MATLAB" (Computer Vision Toolbox).

```matlab
function rotAngleDeg  = ExampleHelperRegisterAndEstimateTransform(img1,img2)
    %%
    %#codegen
    % Declare functions not supported for code generation as extrinsic
    coder.extrinsic('extractFeatures');
    coder.extrinsic('showMatchedFeatures')
    coder.extrinsic('imshowpair');

    % Initialize the transform
    tform = affine2d(single(eye(3)));

    %% Feature Detection and Extraction
    % Convert images to grayscale
    Igray1 = rgb2gray(img1);
    Igray2 = rgb2gray(img2);

    % Detect ORB features in two images
    pointsImage1 = detectORBFeatures(Igray1,'ScaleFactor',1.2,'NumLevels',8);
    pointsImage2 = detectORBFeatures(Igray2,'ScaleFactor',1.2,'NumLevels',8);

    % Select 1500 uniformly distributed points
    pointsImage1 = selectUniform(pointsImage1,1500,size(Igray1,1:2));
    pointsImage2 = selectUniform(pointsImage2,1500,size(Igray2,1:2));

    % Extract features from the two images
    [featuresImage1, validPointsImage1] = extractFeatures(Igray1,pointsImage1);
    [featuresImage2, validPointsImage2] = extractFeatures(Igray2,pointsImage2);

    %% Feature Matching
    % Find candidate matches
    indexPairs = matchFeatures(featuresImage1,featuresImage2,'Unique',true,'MaxRatio',0.6,'Match
    sz = size(indexPairs);
```

```matlab
        % Find the matched feature point locations in both images
        matchedPointsImage1 = validPointsImage1(indexPairs(1:sz(1),1));
        matchedPointsImage2 = validPointsImage2(indexPairs(1:sz(1),2));

        %% Transform estimation
        % Estimate the transformation matrixbetween the two images based on matched points
        [tform,inlierIdx] = estimateGeometricTransform2D(matchedPointsImage2,matchedPointsImage1,'sir

        % Compute the rotation angle from the transformation matrix
        cosVal = tform.T(1,1);
        sinVal = tform.T(2,1);
        rotAngleDeg = atan2d(sinVal,cosVal);

        % Remove all the outliers
        inliersImage1 = matchedPointsImage1(inlierIdx);
        inliersImage2 = matchedPointsImage2(inlierIdx);

        %% Visualize matched inliers and transformed image
        figure
        % subplot(2,1,1)
        showMatchedFeatures(img1,img2,inliersImage1,inliersImage2,'montage');
        title('Matching points in Image 1 (left) and Image 2 (inliers only)')

        figure
        % subplot(2,1,2)
        outputView = imref2d(size(img1));
        img3 = imwarp(img2,tform,'OutputView',outputView);
        imshowpair(img1,img3,'montage');
        title('Image 2 (right) transformed to the view of Image 1')
end
```

Read ROS 2 Point Cloud Messages in Simulink and Perform Stitching Using Registration

This example shows how to read ROS 2 `PointCloud2` messages into Simulink® and reconstruct a 3-D scene by combining multiple point clouds using the normal distributions transform (NDT) algorithm. This example requires Computer Vision Toolbox®.

Set Up ROS 2 Network

Load two sample `sensor_msgs/PointCloud2` messages, `ptcloudMsg1` and `ptcloudMsg2`. Create a ROS 2 node with two publishers to publish the messages on the topics `/ptcloud1` and `/ptcloud2`. For the publishers, set the quality of service (QoS) property `Durability` to `transientlocal`. This ensures that the publishers maintain the messages for any subscribers that join after the messages have already been sent. Publish the messages using the `send` function.

```
load("ros2PtcloudMsgs.mat");

node = ros2node("/point_cloud_publisher");
ptcloudPub1 = ros2publisher(node,"/ptcloud1","sensor_msgs/PointCloud2",Durability="transientlocal
ptcloudPub2 = ros2publisher(node,"/ptcloud2","sensor_msgs/PointCloud2",Durability="transientlocal

send(ptcloudPub1,ptcloudMsg1)
send(ptcloudPub2,ptcloudMsg2)
```
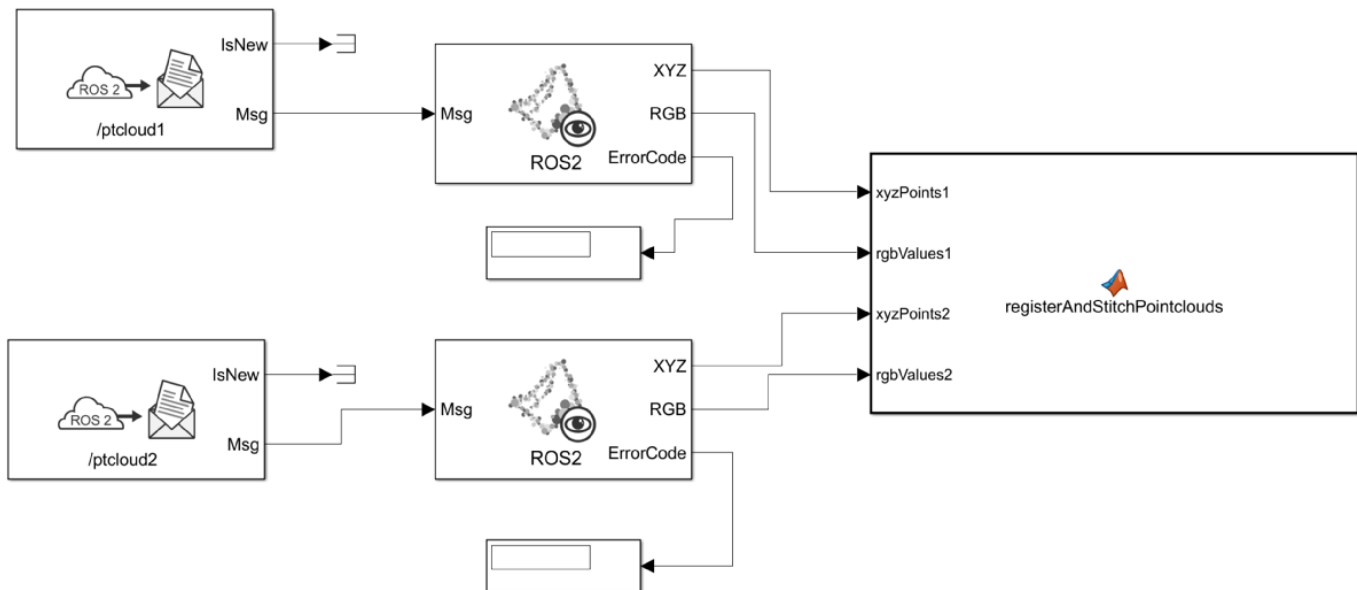
Read Messages from ROS 2 Network in Simulink and Perform Point Cloud Stitching

Open the model.

```
modelName = "read_pointcloud_and_perform_stitching_example_model.slx";
open_system(modelName)
```


Copyright 2021 The MathWorks, Inc.

Ensure that the two Subscribe blocks are connected to the topics, `/ptcloud1` and `/ptcloud2`. The value of the **Durability** QoS parameter of the Subscribe blocks is `Transient local`. This ensures compatibility with the QoS settings of the publishers that you configured in the previous section. The messages from the subscribers are fed to the Read Point Cloud blocks to extract the point cloud data, which is then fed to the `registerAndStitchPointclouds` MATLAB Function block to perform registration and stitching.

Under **Simulation** tab, select **ROS Toolbox > Variable Size Messages**. Notice that the **Use default limits for this message type** is clear. Then, in the **Maximum length** column, verify that the `data` property of the `sensor_msgs/PointCloud2` message has a value greater than the number of points in the pointcloud (4915200). Run the model.

```
sim(modelName);
```
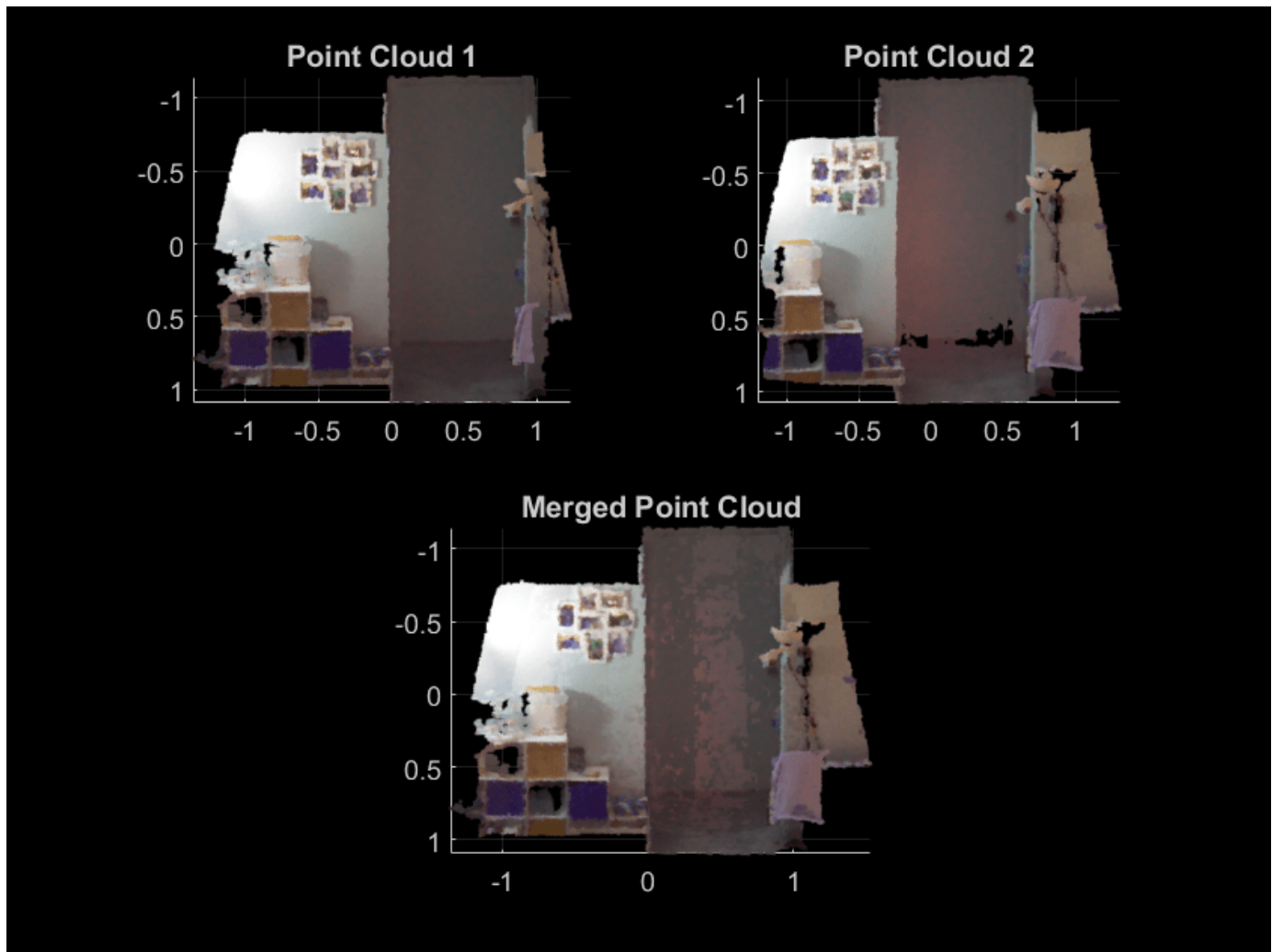

The model performs point cloud registration using the NDT algorithm and estimates the 3-D transformation of the point cloud published on the `/ptcloud2` topic, with respect to the point cloud published on the `/ptcloud1` topic. Next, it transforms the `/ptcloud2` point cloud to the reference coordinate system of the `/ptcloud1` point cloud. It then stitches the transformed point cloud with the `/pcloud1` point cloud to create the 3-D world scene. It also visualizes the two point clouds along with the world scene.

MATLAB Function for Point Cloud Stitching Using Registration

This model uses the algorithm in the `ExampleHelperRegisterAndStitchPointclouds` MATLAB Function to perform point cloud stitching using registration, consisting of these steps:

1   Downsample the point clouds with a `10` cm grid filter to speed up registration and improve accuracy by using the `pcdownsample` (Computer Vision Toolbox) function.

2   Align the two downsampled point clouds using `pcregisterndt` (Computer Vision Toolbox), which uses the NDT algorithm to estimate the 3-D rigid transformation of second pointcloud with respect to the first point cloud.

3   Transform the second point cloud to the reference frame of the first point cloud using the `pctransform` (Computer Vision Toolbox) function.

4   Merge the transformed point cloud and the first point cloud with a 1.5 cm box grid filter by using the `pcmerge` (Computer Vision Toolbox) function.

5   Visualize the two point clouds and the reconstructed 3-D world scene.

For more information about registering and stitching point clouds, see "3-D Point Cloud Registration and Stitching" (Computer Vision Toolbox). Registration is also a key initial step in point cloud SLAM applications. For an overview of the point cloud SLAM workflow using MATLAB, see "Implement Point Cloud SLAM in MATLAB" (Computer Vision Toolbox).

```matlab
function ExampleHelperRegisterAndStitchPointclouds(xyzPoints1, rgbValues1, xyzPoints2, rgbValues2
    %%
    %#codegen
    % Declare functions not supported for code generation as extrinsic
    coder.extrinsic("pcshow");
    %%
    %Create pointcloud objects from extracted xyz points and color data
    pcloud1 = pointCloud(xyzPoints1);
    pcloud1.Color = uint8(rgbValues1*255);
    pcloud2 = pointCloud(xyzPoints2);
    pcloud2.Color = uint8(rgbValues2*255);

    %Downsample the point clouds
    pcloud1Downsampled = pcdownsample(pcloud1,gridAverage=0.1);
    pcloud2Downsampled = pcdownsample(pcloud2,gridAverage=0.1);

    % Register the two pointclouds using the NDT algorithm
    gridStep = 0.5;
    tform = pcregisterndt(pcloud2Downsampled,pcloud1Downsampled,gridStep);

    % Transform and align the frame of point cloud 2 to the frame of point cloud 1
    pCloudAligned = pctransform(pcloud2,tform);

    % Stitch the transformed point cloud 2 with point cloud 1
    mergeSize = 0.015;
    ptCloudScene = pcmerge(pcloud1, pCloudAligned, mergeSize);

    % Visualize the two point clouds and the merged point clouds
    subplot(2,2,1)
    pcshow(pcloud1.Location,pcloud1.Color,VerticalAxis="Y",VerticalAxisDir="Down");
    title("Point Cloud 1")
    view(0,-90);

    subplot(2,2,2)
```

```
pcshow(pcloud2.Location,pcloud2.Color,VerticalAxis="Y",VerticalAxisDir="Down");
title("Point Cloud 2")
view(0,-90);

subplot(2,2,[3,4])
pcshow(ptCloudScene.Location,ptCloudScene.Color,VerticalAxis="Y",VerticalAxisDir="Down");
title("Merged Point Cloud")
view(0,-90);
end
```

Call ROS 2 Service in Simulink

This example shows how to call a service on the ROS 2 network in Simulink® using the Call Service block and receive a response.

Set Up ROS 2 Network and Service Server

Create a sample ROS 2 network with one node.

```
node_1 = ros2node(exampleHelperCreateRandomNodeName);
```

Create a service that adds two integers using the exiisting service type `example_interfaces/AddTwoInts`. Specify the callback function to be `exampleHelperROS2SumCallback` which performs the addition of numbers in the **a** and **b** fields of the service request message.

```
sumserver = ros2svcserver(node_1,"/sum","example_interfaces/AddTwoInts",@exampleHelperROS2SumCal
```

Call Service Server from Simulink

Open the Simulink model with the Call Service block. Use the Blank Message block to output a request message with the `example_interfaces/AddTwoIntsRequest` message type. Populate the bus with two values to sum together. You can ignore warnings about converting data types.
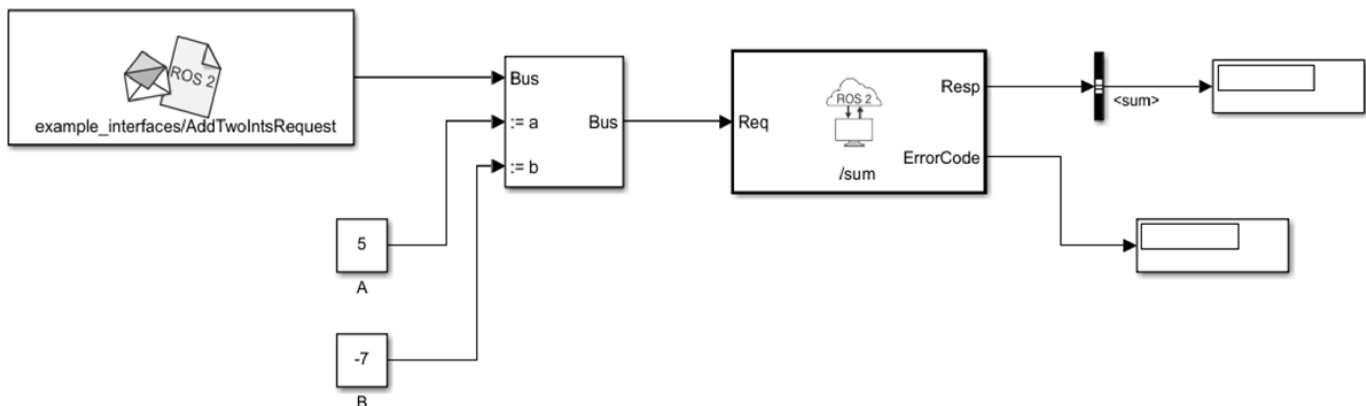
© Copyright 2021 The MathWorks, Inc.

```
open_system("AddTwoIntsROS2ServiceExampleModel.slx")
```

Run the model. The service call should return -2 from the **Resp** output port, as part of the `sum` field in the response message. An error code of 0 indicates that the service call was successful.

```
sim("AddTwoIntsROS2ServiceExampleModel.slx");
```

Read A ROS Scan Message In Simulink®

This example shows how to read in a laser scan message from a ROS network. Calculate the points in Cartesian coordinates and visualize with a 2D plot.

Start a ROS network.

```
rosinit
```

```
Launching ROS Core...
..Done in 2.3118 seconds.
Initializing ROS master on http://172.30.131.134:53152.
Initializing global node /matlab_global_node_97449 with NodeURI http://bat6234win64:62021/ and Ma
```

Load a sample ROS `LaserScan` message. Create a publisher for the topic `/scan` and specify the message type as `sensor_msgs/LaserScan`. Publish the sample `LaserScan` message on this topic.

```
exampleHelperROSLoadScan
pub = rospublisher("/scan","sensor_msgs/LaserScan",DataFormat="struct");
send(pub,scan)
```

Open the Simulink® model for subscribing to a published message and reading the scan message from ROS.

```
open_system("read_scan_example_model.slx")
```
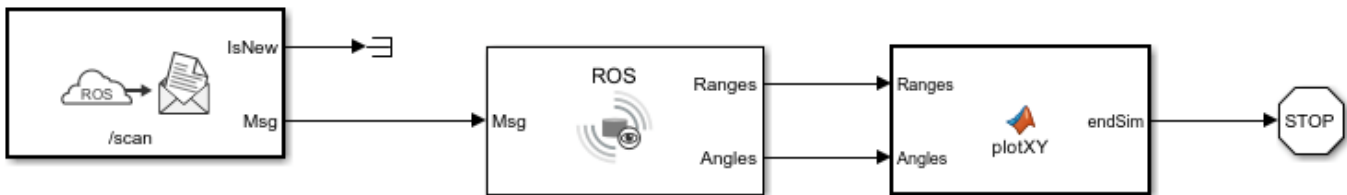

Ensure that the **Subscribe** block is subscribing to the `/scan` topic.

The model reads the `LaserScan` message from the ROS network. The **Read Scan** block extracts range and angle data from the message. These are then fed to the `plotXY` **MATLAB Function** block which converts the ranges and angles to points in Cartesian coordinates and visualizes them on a plot. All NaN values are ignored.

Run the model. The plot shows the points in the laser scan.
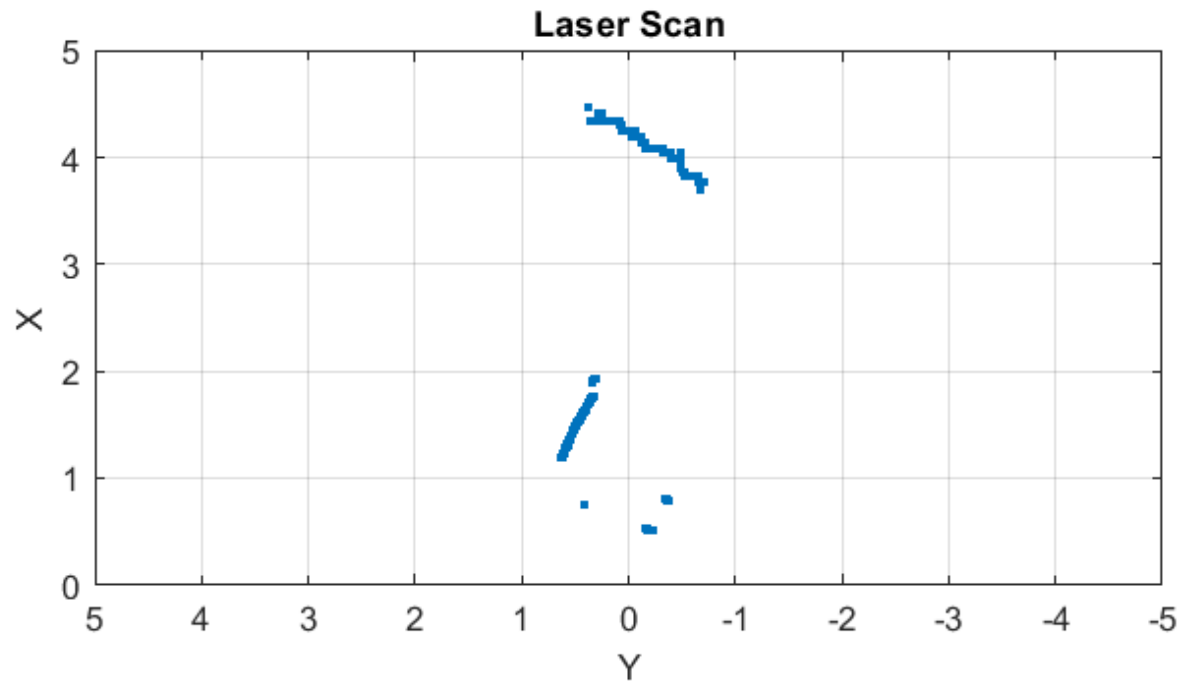
Shut down the ROS network.

```
rosshutdown
```

Shutting down global node /matlab_global_node_97449 with NodeURI http://bat6234win64:62021/ and I
Shutting down ROS master on http://172.30.131.134:53152.

Write A ROS Image Message In Simulink®

This example shows how to write a image message and publish it to a ROS network in Simulink®.

Start a ROS network.

```
rosinit
```

```
Launching ROS Core...
..Done in 2.6694 seconds.
Initializing ROS master on http://172.30.131.134:50750.
Initializing global node /matlab_global_node_34491 with NodeURI http://bat6234win64:52511/ and Ma
```

Load a sample image. Create a subscriber to receive a ROS `Image` message on the `/image` topic. Specify the message type as `sensor_msgs/Image` and the data format as `struct`.

```
exampleHelperROSLoadImageData
sub = rossubscriber("/image","sensor_msgs/Image",DataFormat="struct");
```

Open the Simulink® model for writing and publishing ROS `Image` message.

```
open_system("write_image_example_model.slx");
```
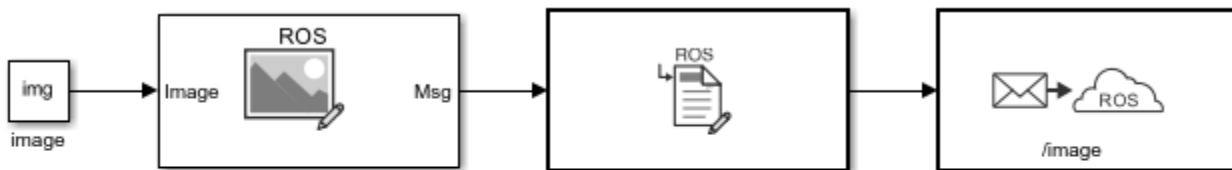

Copyright 2021 The MathWorks, Inc.

Ensure that the **Publish** block is publishing to the `/image` topic. Under the **Simulation** tab, select **ROS Toolbox** > **Variable Size Arrays** and verify the `Data` array has a maximum length greater than the length of the `Data` field in `img` (921,600).

The model writes the input `img` to a ROS `Image` message. Using a **Header Assignment** block, the frame ID of the message header is set to `/camera_depth_optical_frame`.

Run the model to publish the message.

```
sim("write_image_example_model.slx");
```

The image message is published to the ROS network and received by the subscriber created earlier. You can use the `rosReadImage` function to retrieve the image in a format that is compatible with MATLAB.

```
imageFormatted = rosReadImage(sub.LatestMessage);
```

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

```
figure
imshow(imageFormatted)
```
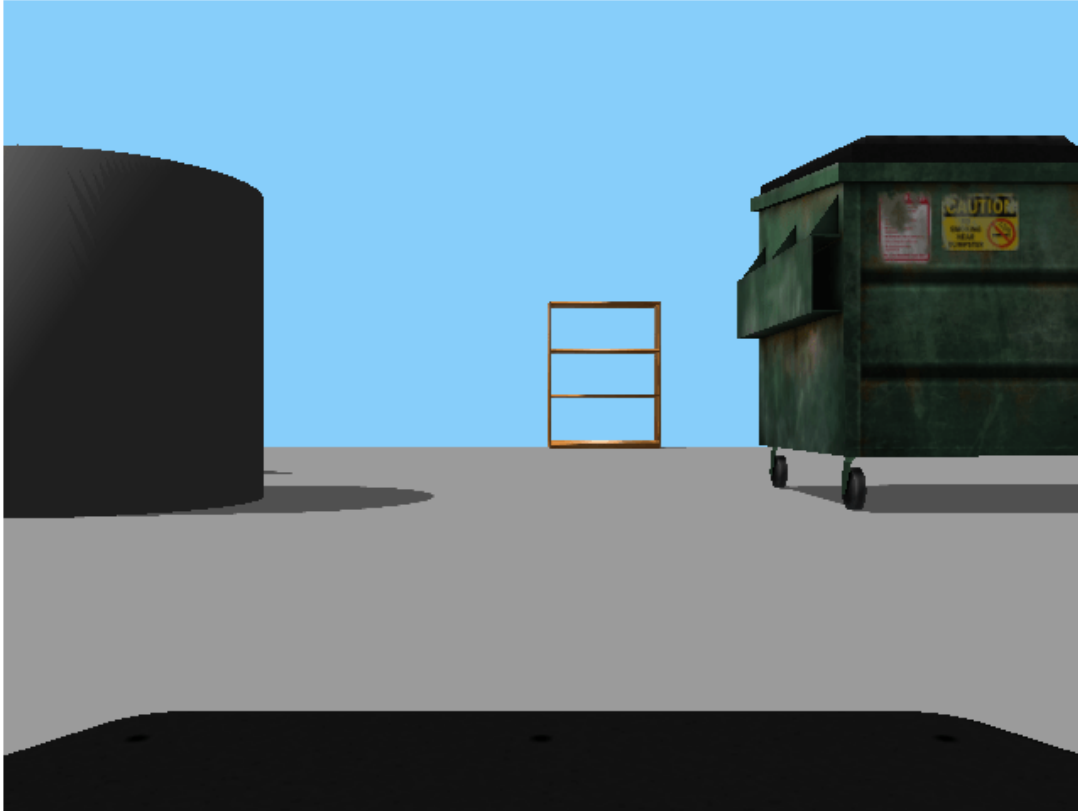

Close the model and shut down the ROS network.

```
close_system("write_image_example_model.slx",0);
rosshutdown
```

Shutting down global node /matlab_global_node_34491 with NodeURI http://bat6234win64:52511/ and M
Shutting down ROS master on http://172.30.131.134:50750.

Write A ROS Point Cloud Message In Simulink®

This example shows how to write a point cloud message and publish it to a ROS network in Simulink®.

Start a ROS network.

```
rosinit
```

```
Launching ROS Core...
..Done in 2.1384 seconds.
Initializing ROS master on http://172.30.131.134:53026.
Initializing global node /matlab_global_node_91440 with NodeURI http://bat6234win64:58104/ and Ma
```

Load the 3D coordinate locations and color data for the sample point cloud. Create a subscriber to receive a ROS `PointCloud2` message from the `/point_cloud` topic. Specify the message type as `sensor_msgs/PointCloud2` and the data format as a `struct`.

```
exampleHelperROSLoadPointCloudData
sub = rossubscriber("/point_cloud","sensor_msgs/PointCloud2",DataFormat="struct");
```

Open the Simulink® model for writing a ROS point cloud message and publishing it to a ROS network.

```
open_system("write_point_cloud_example_model.slx");
```
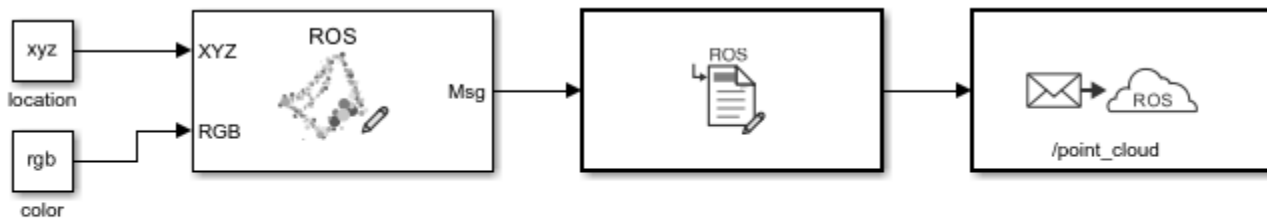

Copyright 2021 The MathWorks, Inc.

Ensure that the **Publish** block is publishing to the `/point_cloud` topic. Under the **Simulation** tab, select **ROS Toolbox > Variable Size Arrays** and verify the `Data` array has a maximum length greater than the length of the `Data` field in `ptcloud` (9,830,400).

The model writes the color and location data to a ROS point cloud message. Using a **Header Assignment** block, the frame ID of the message header is set to `/camera_rgb_optical_frame`.

Run the model to publish the message.

```
sim("write_point_cloud_example_model.slx");
```

The point cloud message is published to the ROS network and received by the subscriber created earlier. Visualize the point cloud with the `rosPlot` function. `rosPlot` automatically extracts the x-, y-, and z- coordinates and shows them in a 3-D scatter plot. The `rosPlot` function ignores points with NaN values for coordinates regardless of the presence of color values at those points.

```
rosPlot(sub.LatestMessage);
```
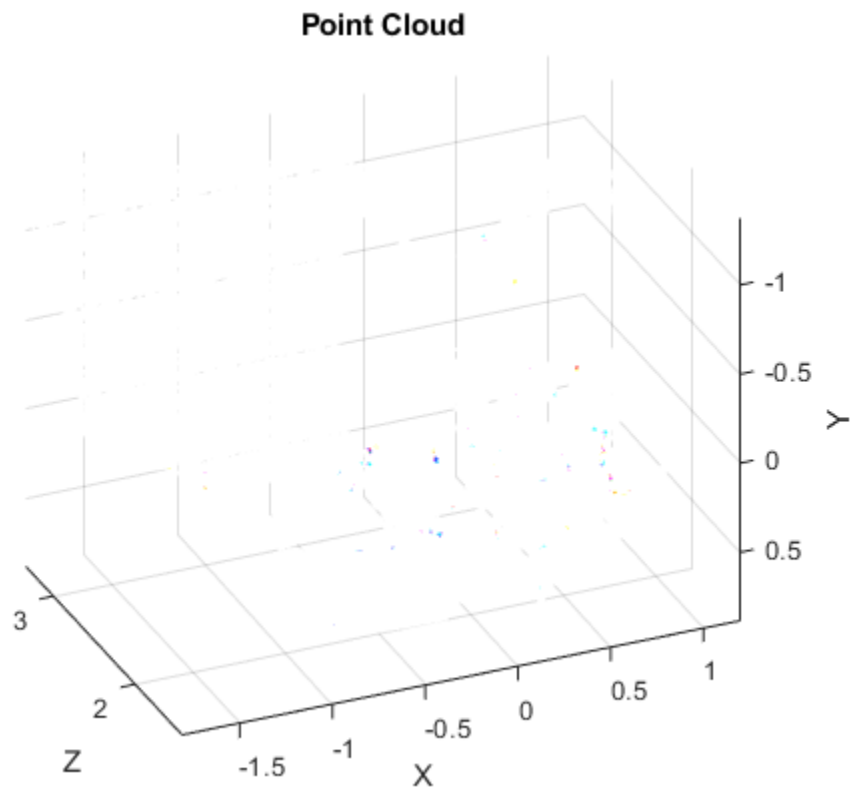

Close the model and shut down the ROS network.

```
close_system("write_point_cloud_example_model.slx",0);
rosshutdown
```

Shutting down global node /matlab_global_node_91440 with NodeURI http://bat6234win64:58104/ and I
Shutting down ROS master on http://172.30.131.134:53026.

Get and Set ROS 2 Parameters in Simulink

This example shows how to use ROS 2 parameters in Simulink® and share data over the ROS 2 network.

Create and Get ROS 2 Parameter

Open the model.

```
modelName = "GetAndSetROS2Parameters";
open_system(modelName)
```

This model uses the Get Parameter block to create a parameter `myparam`. The parameter is associated with the ROS 2 node that the Simulink model creates during simulation. You can specify the initial value of `myparam` in the block parameters for the Get Parameter block.
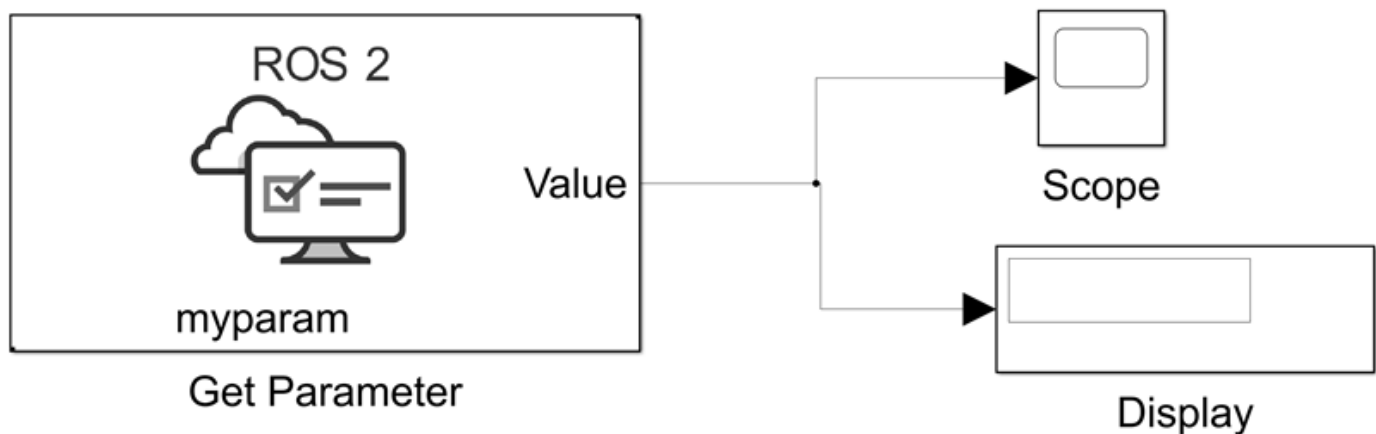
Copyright 2022 The MathWorks Inc

Get and Set ROS 2 Parameter

To set the parameter associated with the Simulink model during the simulation, you must use programmatic commands to run simulation. Start the simulation.

```
set_param(modelName,"SimulationCommand","start")
```

Pause the simulation to obtain the name of the ROS 2 node created by the Simulink model. Create a `ros2param` object for the node, to access its parameters.

```
set_param(modelName,"SimulationCommand","pause")

if get_param(modelName,"SimulationStatus") == "paused"
    nodelist = ros2("node","list");
    simNodeArr = strfind(nodelist,modelName);
    simNodeIdx = find(~cellfun(@isempty, simNodeArr));
    simNodeName = nodelist{simNodeIdx};
    paramObj = ros2param(simNodeName);
end
```

Continue the simulation. At each timestep, get the current value of the parameter by using the `get` object function on `ros2param` object. Then, set the new value of the parameter using the `set` object function. Observe the Scope showing the parameter value increase by 1 every simulation time step.

```
set_param(modelName,"SimulationCommand","step")
while get_param(modelName,"SimulationStatus") == "paused"
    % Get and set parameter
    [currParam,status] = get(paramObj,"myparam");
    if status
        set(paramObj,"myparam",currParam+1);
    end
    set_param(modelName,"SimulationCommand","step")
end
```
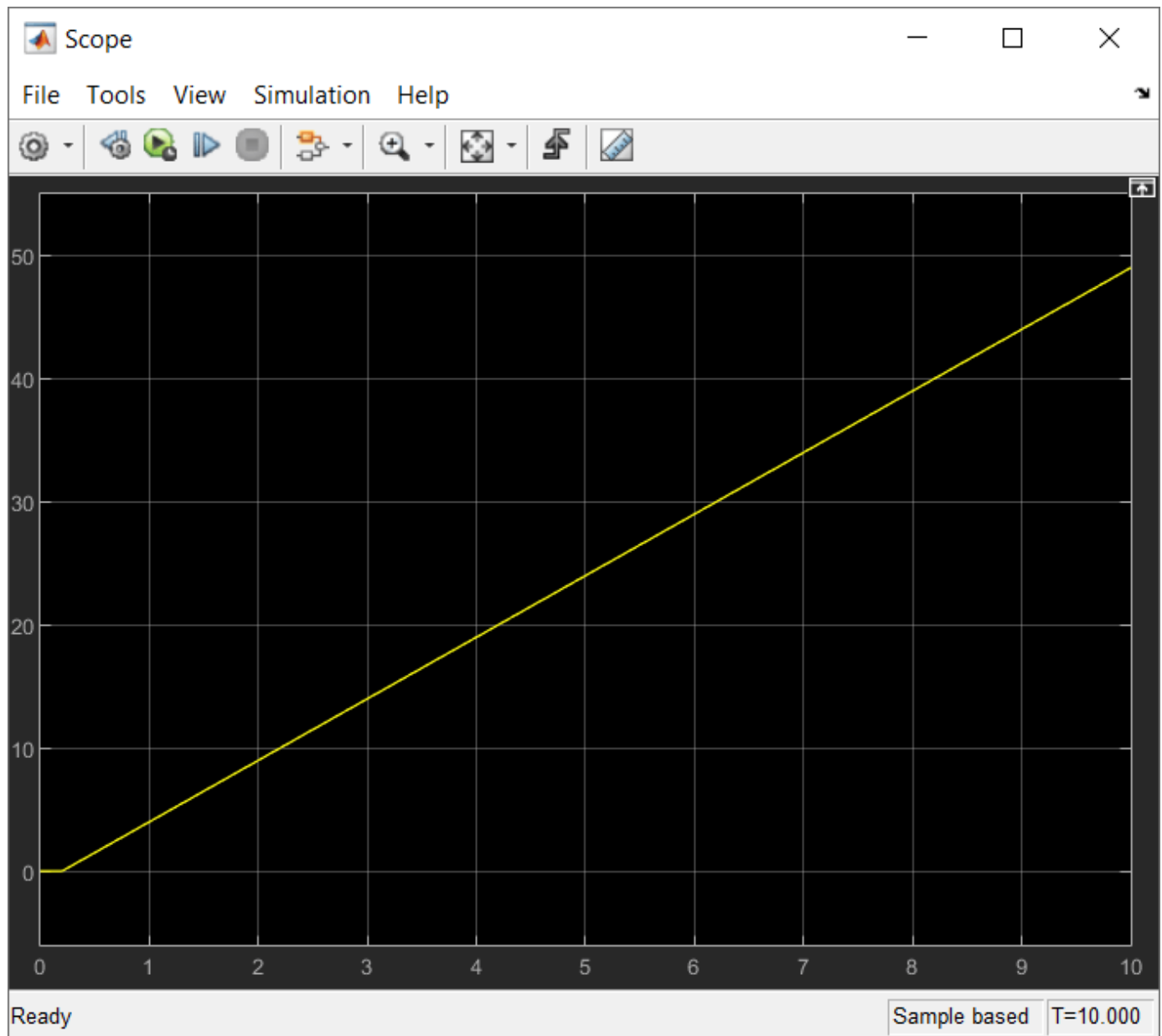
Time Stamp a ROS 2 Message Using Current Time in Simulink

This example shows how to time stamp a ROS 2 message with the current system time of your computer. Use the **Current Time** block and assign its output to the `stamp` field of a `std_msgs/Header` message. Publish the message on a desired topic.

Open the model. The model uses a **Bus Assignment** block to add the **Current Time** output to the stamp field of the ROS 2 message.

```
open_system("current_time_ros2_example_model.slx")
```
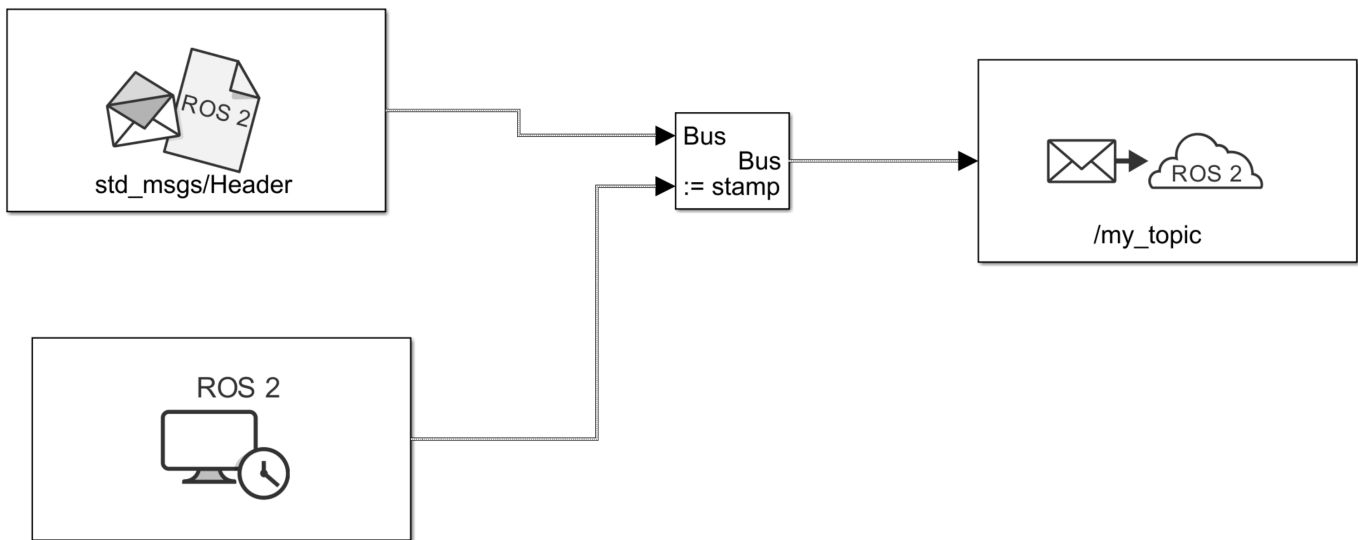

Copyright 2022 The MathWorks Inc

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

```
sim("current_time_ros2_example_model.slx");
```

Update Header Field of a ROS 2 Message in Simulink

This example illustrates how to update the header field of a ROS 2 message using Simulink®.

Some ROS 2 messages contain a specific header field which maps to `std_msgs/Header` message type. The header field contains timestamp and coordinate frame information of the ROS 2 message. This example model shows how to use the ROS 2 Header Assignment block to update that information for a ROS 2 message, in Simulink.

```
open_system('ros2HeaderAssignmentExampleModel.slx')
```

The Blank Message block creates an empty ROS 2 message of type, `sensor_msgs/LaserScan`. Any other message type that contains a header field of `std_msgs/Header` can be used here, instead. The output of the Blank Message block is then fed to the ROS 2 Header Assignment block, which updates the header field of this message. For display, the `frame_id` and `stamp` values of the updated ROS 2 message header are selected from the list of bus elements uisng Bus Selector blocks.
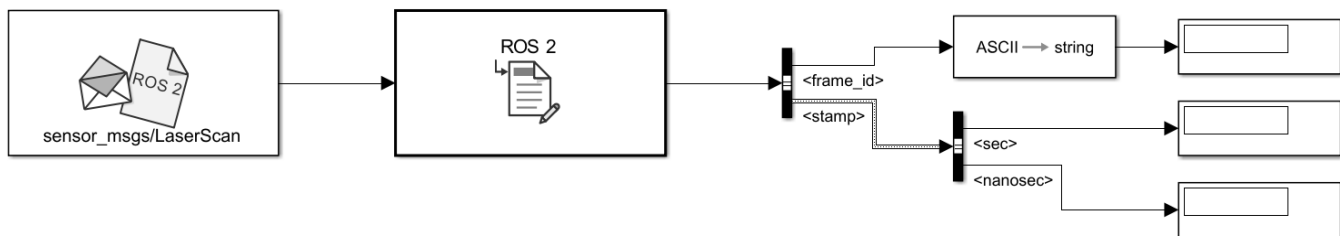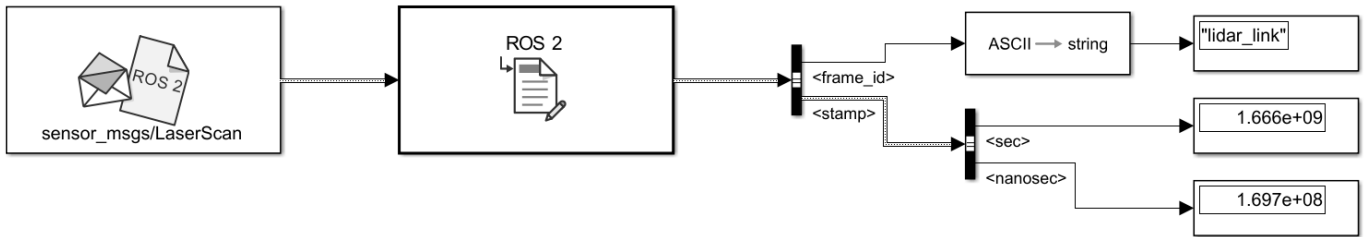

Copyright 2022 The MathWorks, Inc.

Update Coordinate Frame ID and Timestamp Values in Header

Open the ROS 2 Header Assignment block to display its block parameters. The **Set Frame ID** option is selected and the name of the coordinate frame that is associated with the message is specified in the text box as `lidar_link`. This will be set as the `frame_id` value for the Header. The **Set Timestamp** option is also selected which sets the `stamp` value of the Header to the current ROS 2 System time, by default. The **Header field name** option is set to `Use the default Header field name` because, the name of the Header field in a blank message is its default value, `header`. If you are using a ROS 2 message with a custom name for the Header field, you can select the `Specify your own` option from the dropdown and specify the name of the Header field in the text box.

Run the model. Observe the updated values for the frame_id and stamp fields of ROS 2 message in their repective displays.

```
sim('ros2HeaderAssignmentExampleModel');
```

Copyright 2022 The MathWorks, Inc.