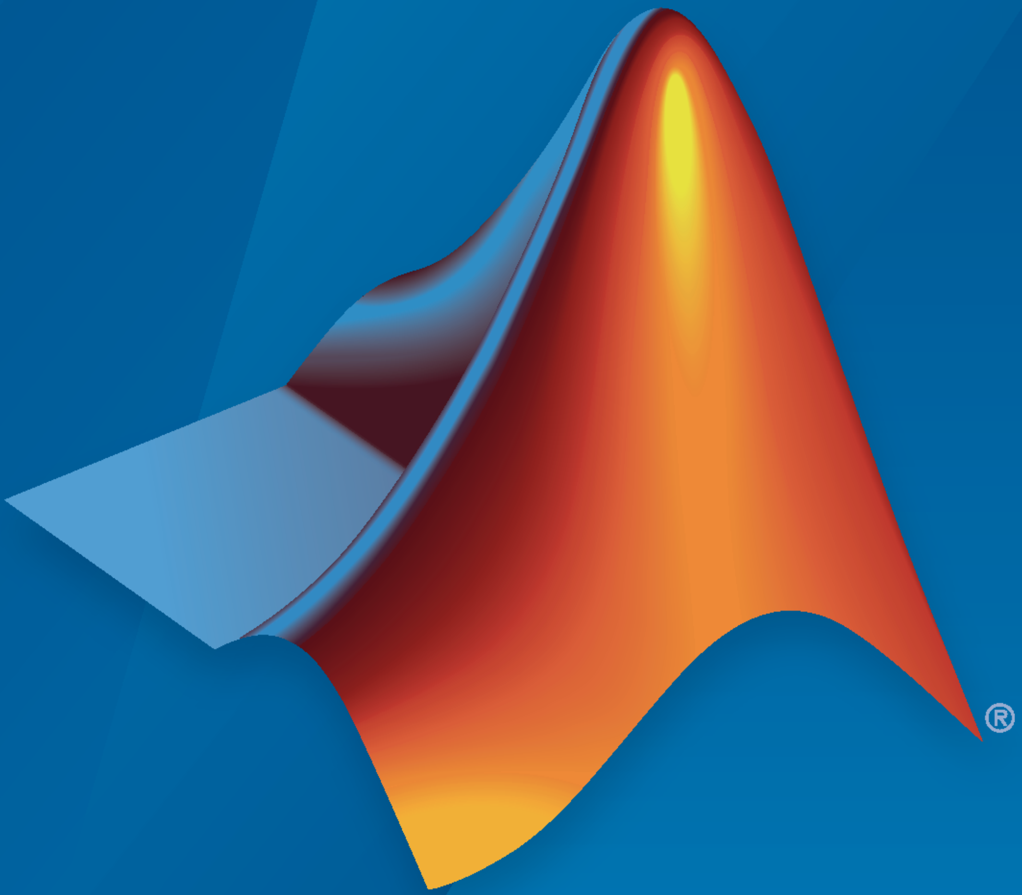
Robotics System Toolbox™

User's Guide



MATLAB®&SIMULINK®

R2022a

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

Robotics System Toolbox™ User's Guide

Trademarks

Patents

Revision History

March 2015	Online only	New for Version 1.0 (R2015a)
September 2015	Online only	Revised for Version 1.1 (R2015b)
October 2015	Online only	Rereleased for Version 1.0.1 (Release 2015aSP1)
March 2016	Online only	Revised for Version 1.2 (R2016a)
September 2016	Online only	Revised for Version 1.3 (R2016b)
March 2017	Online only	Revised for Version 1.4 (R2017a)
September 2017	Online only	Revised for Version 1.5 (R2017b)
March 2018	Online only	Revised for Version 2.0 (R2018a)
September 2018	Online only	Revised for Version 2.1 (R2018b)
March 2019	Online only	Revised for Version 2.2 (R2019a)
September 2019	Online only	Revised for Version 3.0 (R2019b)
March 2020	Online only	Revised for Version 3.1 (R2020a)
September 2020	Online only	Revised for Version 3.2 (R2020b)
March 2021	Online only	Revised for Version 3.3 (R2021a)
September 2021	Online only	Revised for Version 3.4 (R2021b)
March 2022	Online only	Revised for Version 4.0 (R2022a)

Contents

1 Robotics System Toolbox Examples

2 Robotics System Toolbox Topics

3 Examples for Simulink Blocks

1

Robotics System Toolbox Examples

Path Planning in Environments of Different Complexity

This example demonstrates how to compute an obstacle-free path between two locations on a given map using the Probabilistic Roadmap (PRM) path planner. PRM path planner constructs a roadmap in the free space of a given map using randomly sampled nodes in the free space and connecting them with each other. Once the roadmap has been constructed, you can query for a path from a given start location to a given end location on the map.

In this example, the map is represented as an occupancy grid map using imported data. When sampling nodes in the free space of a map, PRM uses this binary occupancy grid representation to deduce free space. Furthermore, PRM does not take into account the robot dimension while computing an obstacle-free path on a map. Hence, you should inflate the map by the dimension of the robot, in order to allow computation of an obstacle-free path that accounts for the robot's size and ensures collision avoidance for the actual robot. Define start and end locations on the map for the PRM path planner to find an obstacle-free path.

Import Example Maps for Planning a Path

```
load exampleMaps.mat
```

The imported maps are : `simpleMap`, `complexMap` and `ternaryMap`.

```
whos *Map*
```

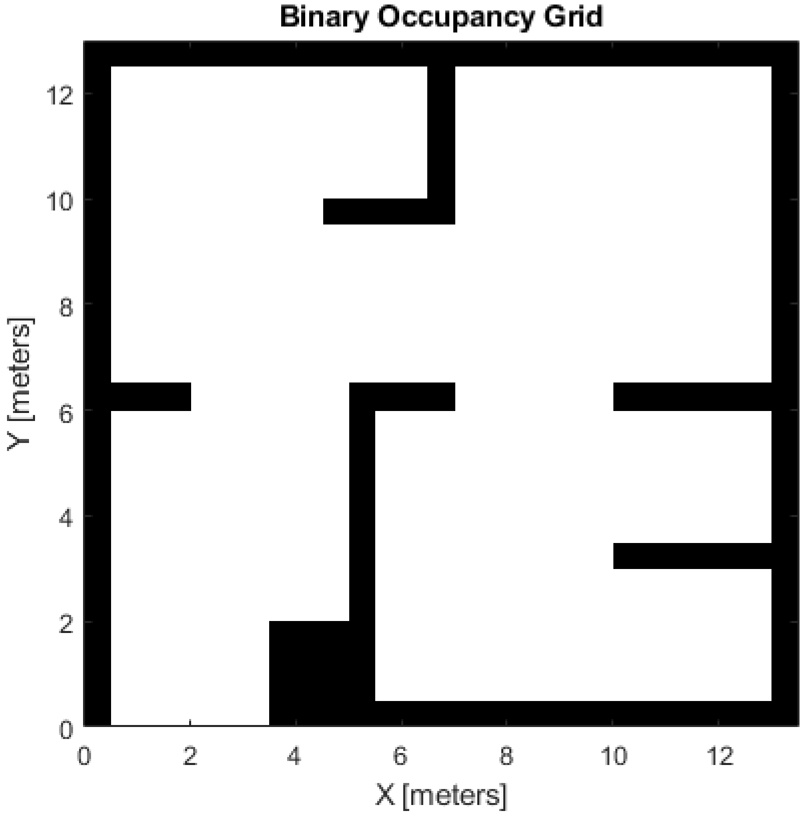
```
  Name                 Size                 Bytes  Class      Attributes

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

Use the imported `simpleMap` data and construct an occupancy grid representation using the `binaryOccupancyMap` object. Set the resolution to 2 cells per meter for this map.

```
map = binaryOccupancyMap(simpleMap,2);
```

Display the map using the `show` function on the `binaryOccupancyMap` object

```
show(map)
```



Define Robot Dimensions and Inflate the Map

To ensure that the robot does not collide with any obstacles, you should inflate the map by the dimension of the robot before supplying it to the PRM path planner.

Here the dimension of the robot can be assumed to be a circle with radius of 0.2 meters. You can then inflate the map by this dimension using the `inflate` function.

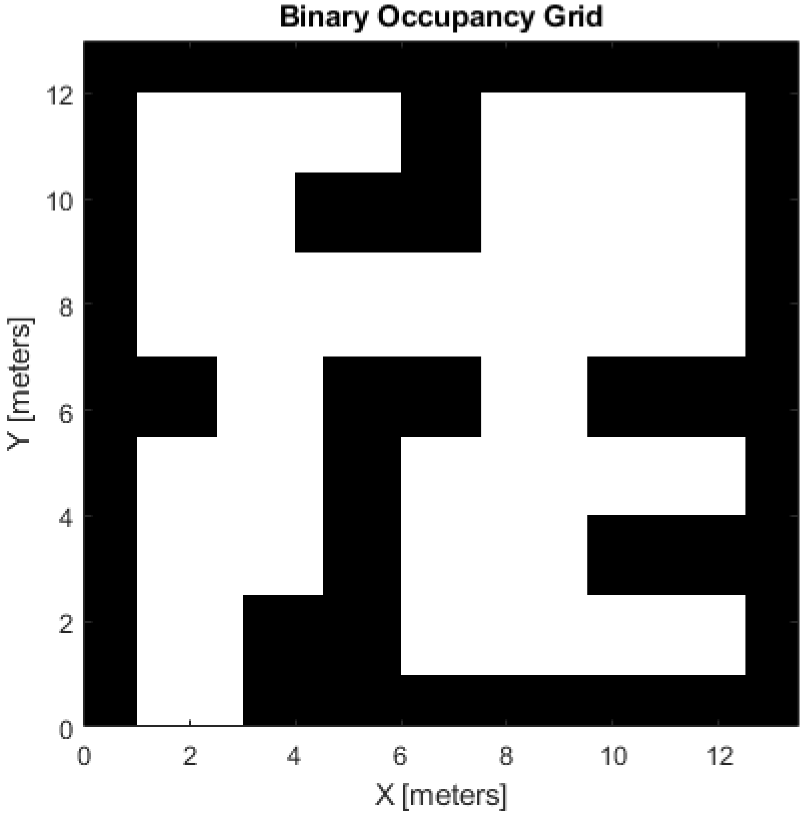
```
robotRadius = 0.2;
```

As mentioned before, PRM does not account for the dimension of the robot, and hence providing an inflated map to the PRM takes into account the robot dimension. Create a copy of the map before using the `inflate` function to preserve the original map.

```
mapInflated = copy(map);
inflate(mapInflated,robotRadius);
```

Display inflated map

```
show(mapInflated)
```



Construct PRM and Set Parameters

Now you need to define a path planner. Create a `mobileRobotPRM` object and define the associated attributes.

```
prm = mobileRobotPRM;
```

Assign the inflated map to the PRM object

```
prm.Map = mapInflated;
```

Define the number of PRM nodes to be used during PRM construction. PRM constructs a roadmap using a given number of nodes on the given map. Based on the dimension and the complexity of the input map, this is one of the primary attributes to tune in order to get a solution between two points on the map. A large number of nodes create a dense roadmap and increases the probability of finding a path. However, having more nodes increases the computation time for both creating the roadmap and finding a solution.

```
prm.NumNodes = 50;
```

Define the maximum allowed distance between two connected nodes on the map. PRM connects all nodes separated by this distance (or less) on the map. This is another attribute to tune in the case of larger and/or complicated input maps. A large connection distance increases the connectivity between nodes to find a path easier, but can increase the computation time of roadmap creation.

```
prm.ConnectionDistance = 5;
```

Find a Feasible Path on the Constructed PRM

Define start and end locations on the map for the path planner to use.

```
startLocation = [2 1];
endLocation = [12 10];
```

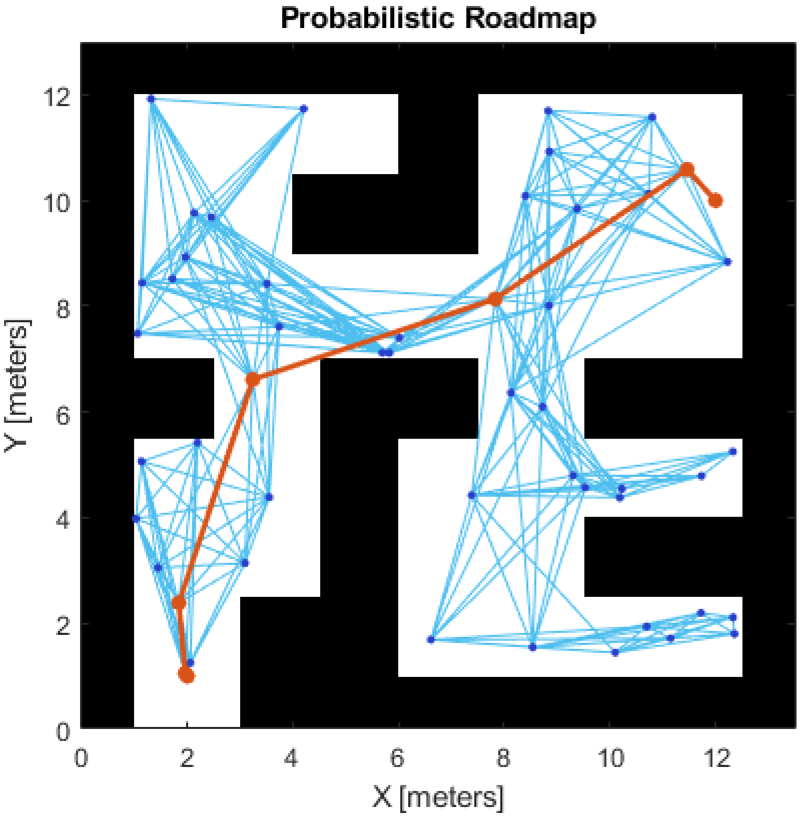
Search for a path between start and end locations using the `findpath` function. The solution is a set of waypoints from start location to the end location. Note that the `path` will be different due to probabilistic nature of the PRM algorithm.

```
path = findpath(prm, startLocation, endLocation)
```

```
path = 7×2

    2.0000    1.0000
    1.9569    1.0546
    1.8369    2.3856
    3.2389    6.6106
    7.8260    8.1330
   11.4632   10.5857
   12.0000   10.0000
```

Display the PRM solution.

```
show(prm)
```



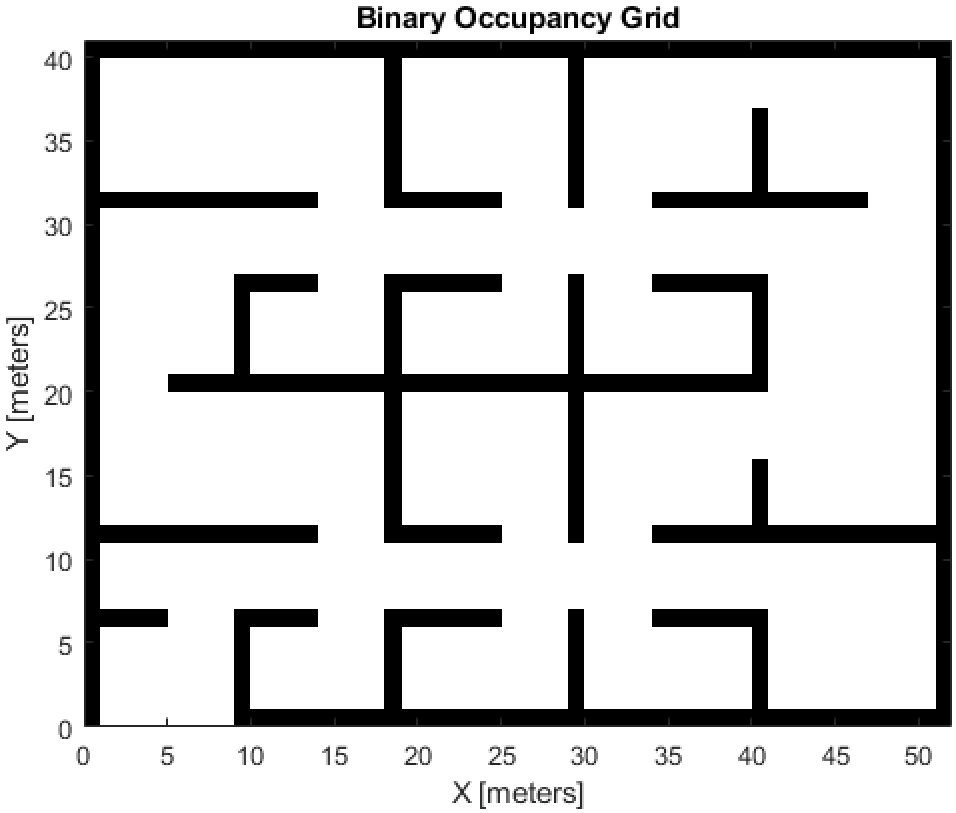
Use PRM for a Large and Complicated Map

Use the imported `complexMap` data, which represents a large and complicated floor plan, and construct a binary occupancy grid representation with a given resolution (1 cell per meter)

```
map = binaryOccupancyMap(complexMap,1);
```

Display the map.

```
show(map)
```



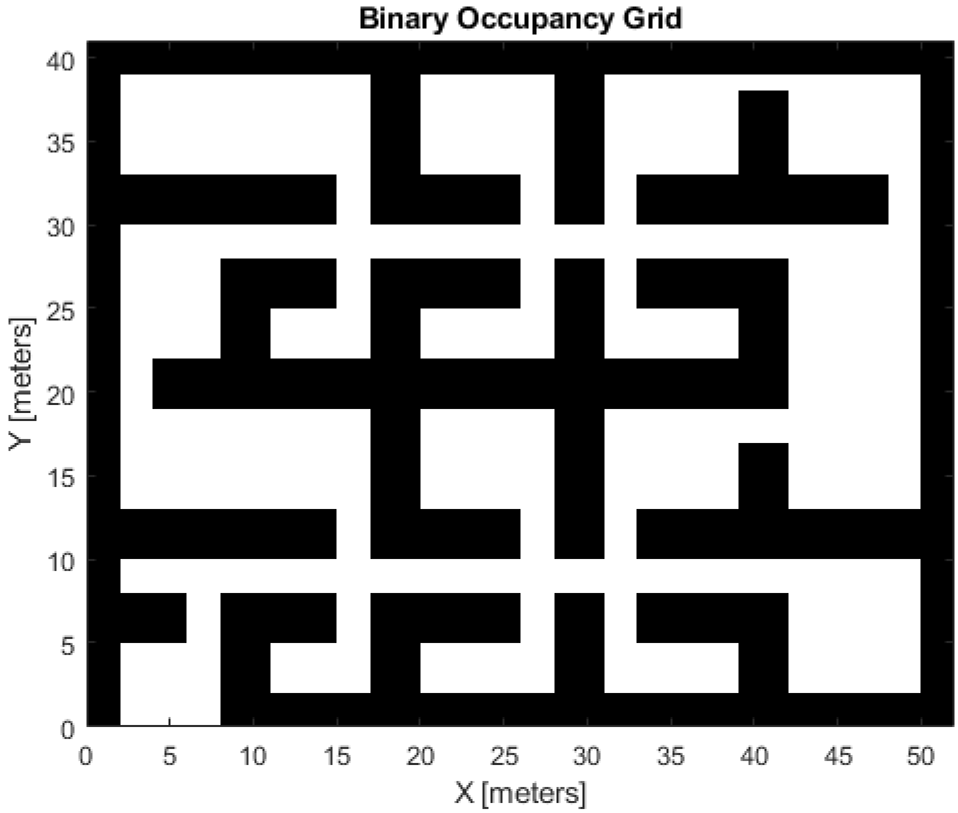
Inflate the Map Based on Robot Dimension

Copy and inflate the map to factor in the robot's size for obstacle avoidance

```
mapInflated = copy(map);
inflate(mapInflated, robotRadius);
```

Display inflated map.

```
show(mapInflated)
```



Associate the Existing PRM Object with the New Map and Set Parameters

Update PRM object with the newly inflated map and define other attributes.

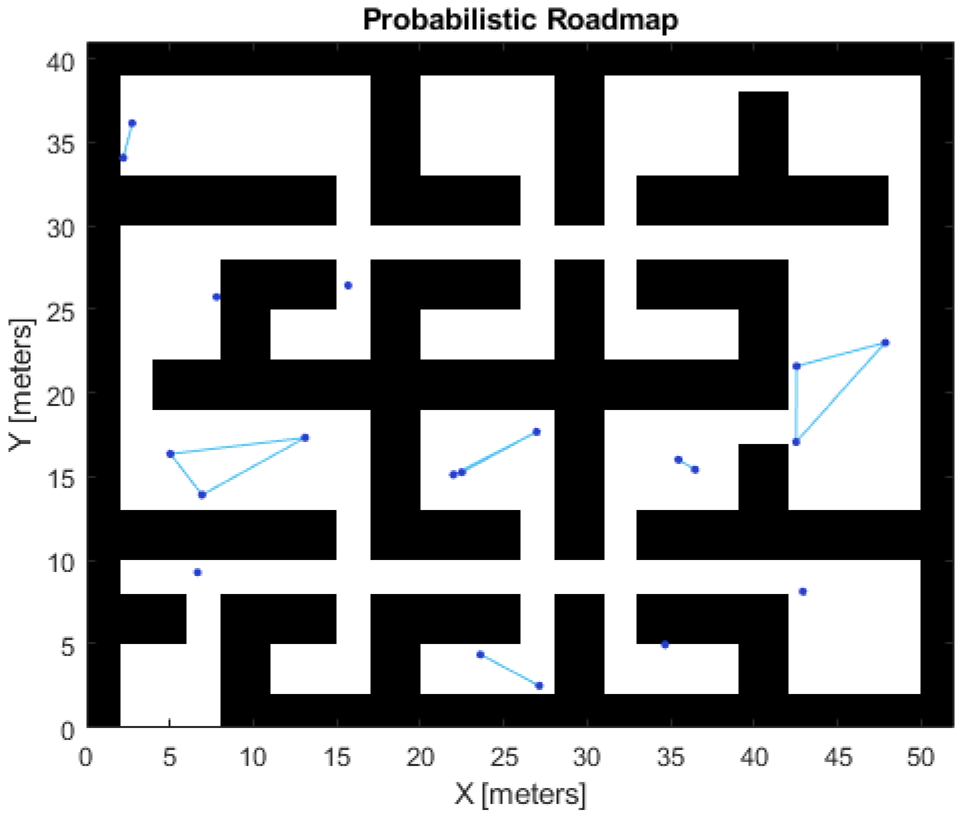
```
prm.Map = mapInflated;
```

Set the `NumNodes` and the `ConnectionDistance` properties.

```
prm.NumNodes = 20;
prm.ConnectionDistance = 15;
```

Display PRM graph.

```
show(prm)
```



Find a Feasible Path on the Constructed PRM

Define start and end location on the map to find an obstacle-free path.

```
startLocation = [3 3];
endLocation = [45 35];
```

Search for a solution between start and end location. For complex maps, there may not be a feasible path for a given number of nodes (returns an empty path).

```
path = findpath(prm, startLocation, endLocation);
```

Since you are planning a path on a large and complicated map, larger number of nodes may be required. However, often it is not clear how many nodes will be sufficient. Tune the number of nodes to make sure there is a feasible path between the start and end location.

```
while isempty(path)
    % No feasible path found yet, increase the number of nodes
    prm.NumNodes = prm.NumNodes + 10;

    % Use the |update| function to re-create the PRM roadmap with the changed
    % attribute
    update(prm);

    % Search for a feasible path with the updated PRM
    path = findpath(prm, startLocation, endLocation);
end
```

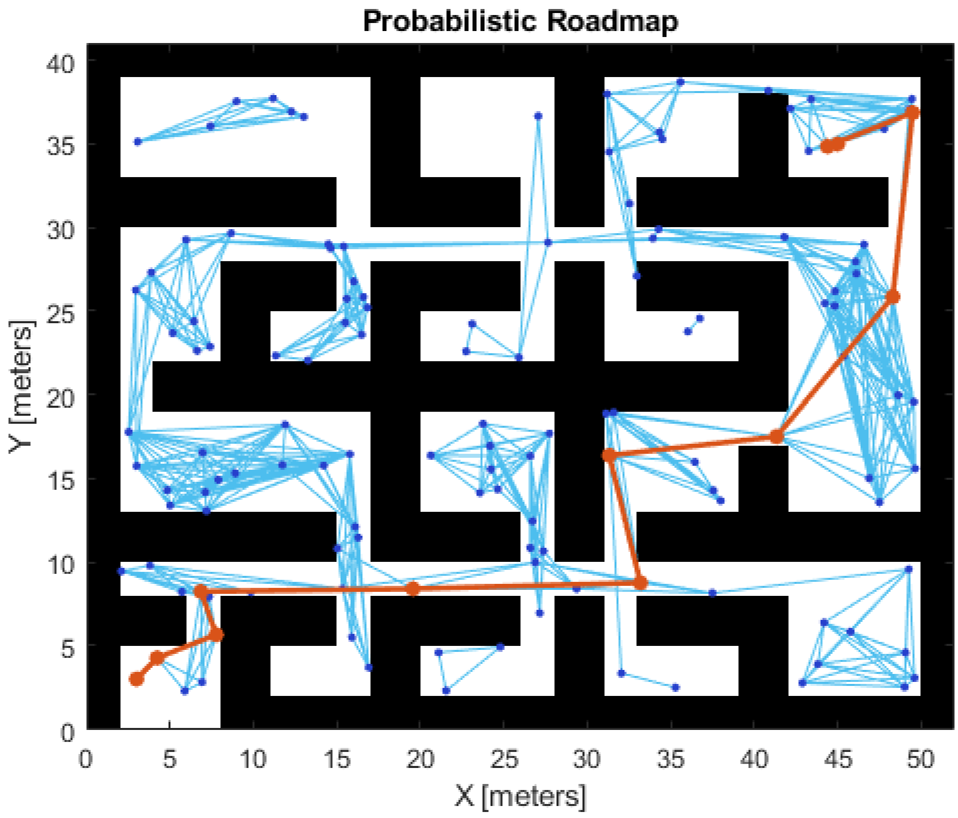
Display path.

```
path
```

```
path = 12×2

    3.0000    3.0000
    4.2287    4.2628
    7.7686    5.6520
    6.8570    8.2389
   19.5613    8.4030
   33.1838    8.7614
   31.3248   16.3874
   41.3317   17.5090
   48.3017   25.8527
   49.4926   36.8804
      ⋮
```

Display PRM solution.

```
show(prm)
```



Path Following for a Differential Drive Robot

This example demonstrates how to control a robot to follow a desired path using a Robot Simulator. The example uses the Pure Pursuit path following controller to drive a simulated robot along a predetermined path. A desired path is a set of waypoints defined explicitly or computed using a path planner (refer to “Path Planning in Environments of Different Complexity” on page 1-2). The Pure Pursuit path following controller for a simulated differential drive robot is created and computes the control commands to follow a given path. The computed control commands are used to drive the simulated robot along the desired trajectory to follow the desired path based on the Pure Pursuit controller.

Note: Starting in R2016b, instead of using the step method to perform the operation defined by the System object™, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Define Waypoints

Define a set of waypoints for the desired path for the robot

```
path = [2.00    1.00;
        1.25    1.75;
        5.25    8.25;
        7.25    8.75;
        11.75   10.75;
        12.00   10.00];
```

Set the current location and the goal location of the robot as defined by the path.

```
robotInitialLocation = path(1,:);
robotGoal = path(end,:);
```

Assume an initial robot orientation (the robot orientation is the angle between the robot heading and the positive X-axis, measured counterclockwise).

```
initialOrientation = 0;
```

Define the current pose for the robot [x y theta]

```
robotCurrentPose = [robotInitialLocation initialOrientation]';
```

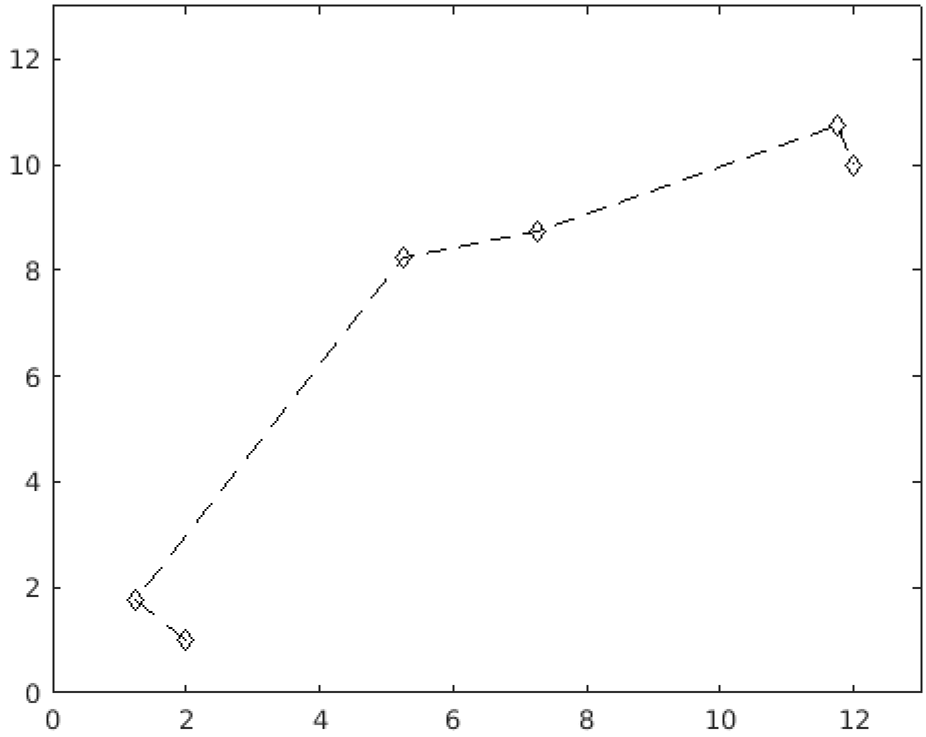
Create a Kinematic Robot Model

Initialize the robot model and assign an initial pose. The simulated robot has kinematic equations for the motion of a two-wheeled differential drive robot. The inputs to this simulated robot are linear and angular velocities.

```
robot = differentialDriveKinematics("TrackWidth", 1, "VehicleInputs", "VehicleSpeedHeadingRate")
```

Visualize the desired path

```
figure
plot(path(:,1), path(:,2),'k--d')
xlim([0 13])
ylim([0 13])
```



Define the Path Following Controller

Based on the path defined above and a robot motion model, you need a path following controller to drive the robot along the path. Create the path following controller using the `controllerPurePursuit` object.

```
controller = controllerPurePursuit;
```

Use the path defined above to set the desired waypoints for the controller

```
controller.Waypoints = path;
```

Set the path following controller parameters. The desired linear velocity is set to 0.6 meters/second for this example.

```
controller.DesiredLinearVelocity = 0.6;
```

The maximum angular velocity acts as a saturation limit for rotational velocity, which is set at 2 radians/second for this example.

```
controller.MaxAngularVelocity = 2;
```

As a general rule, the lookahead distance should be larger than the desired linear velocity for a smooth path. The robot might cut corners when the lookahead distance is large. In contrast, a small lookahead distance can result in an unstable path following behavior. A value of 0.3 m was chosen for this example.

```
controller.LookaheadDistance = 0.3;
```

Using the Path Following Controller, Drive the Robot over the Desired Waypoints

The path following controller provides input control signals for the robot, which the robot uses to drive itself along the desired path.

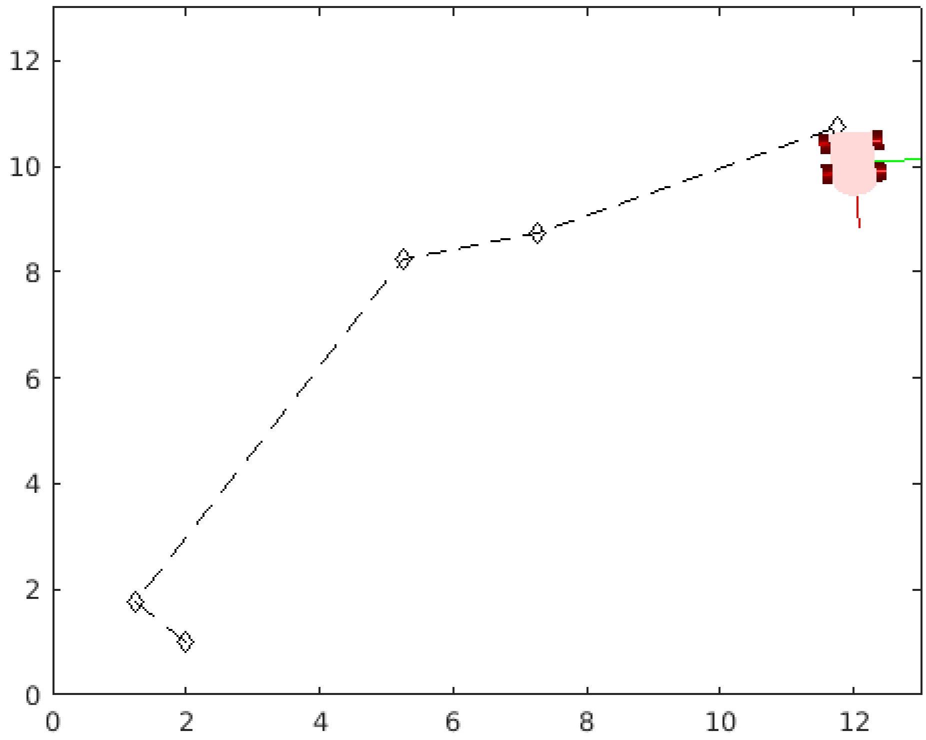
Define a goal radius, which is the desired distance threshold between the robot's final location and the goal location. Once the robot is within this distance from the goal, it will stop. Also, you compute the current distance between the robot location and the goal location. This distance is continuously checked against the goal radius and the robot stops when this distance is less than the goal radius.

Note that too small value of the goal radius may cause the robot to miss the goal, which may result in an unexpected behavior near the goal.

```
goalRadius = 0.1;
distanceToGoal = norm(robotInitialLocation - robotGoal);
```

The `controllerPurePursuit` object computes control commands for the robot. Drive the robot using these control commands until it reaches within the goal radius. If you are using an external simulator or a physical robot, then the controller outputs should be applied to the robot and a localization system may be required to update the pose of the robot. The controller runs at 10 Hz.

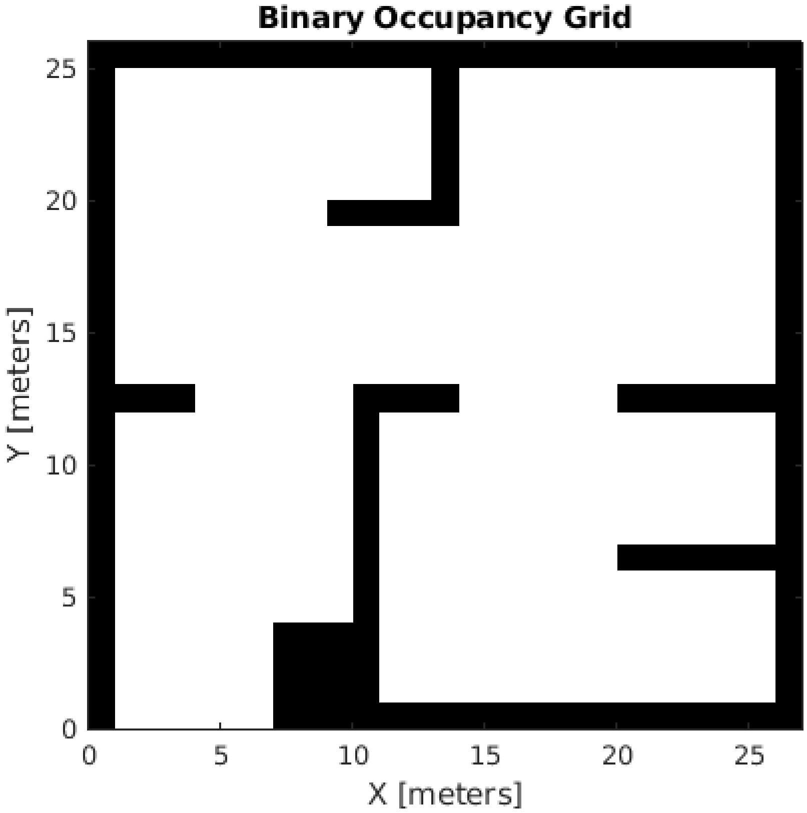
```
% Initialize the simulation loop
sampleTime = 0.1;
vizRate = rateControl(1/sampleTime);

% Initialize the figure
figure

% Determine vehicle frame size to most closely represent vehicle with plotTransforms
frameSize = robot.TrackWidth/0.8;

while( distanceToGoal > goalRadius )

    % Compute the controller outputs, i.e., the inputs to the robot
    [v, omega] = controller(robotCurrentPose);

    % Get the robot's velocity using controller inputs
    vel = derivative(robot, robotCurrentPose, [v omega]);

    % Update the current pose
    robotCurrentPose = robotCurrentPose + vel*sampleTime;

    % Re-compute the distance to the goal
    distanceToGoal = norm(robotCurrentPose(1:2) - robotGoal(:));

    % Update the plot
    hold off

    % Plot path each instance so that it stays persistent while robot mesh
    % moves
    plot(path(:,1), path(:,2),"k--d")
    hold all

    % Plot the path of the robot as a set of transforms
    plotTrVec = [robotCurrentPose(1:2); 0];
    plotRot = axang2quat([0 0 1 robotCurrentPose(3)]);
    plotTransforms(plotTrVec', plotRot, "MeshFilePath", "groundvehicle.stl", "Parent", gca, "Vie
    light;
    xlim([0 13])
    ylim([0 13])

    waitfor(vizRate);
end
```



Using the Path Following Controller Along with PRM

If the desired set of waypoints are computed by a path planner, the path following controller can be used in the same fashion. First, visualize the map

```
load exampleMaps
map = binaryOccupancyMap(simpleMap);
figure
show(map)
```



You can compute the `path` using the PRM path planning algorithm. See "Path Planning in Environments of Different Complexity" on page 1-2 for details.

```
mapInflated = copy(map);
inflate(mapInflated, robot.TrackWidth/2);
prm = robotics.PRM(mapInflated);
prm.NumNodes = 100;
prm.ConnectionDistance = 10;
```

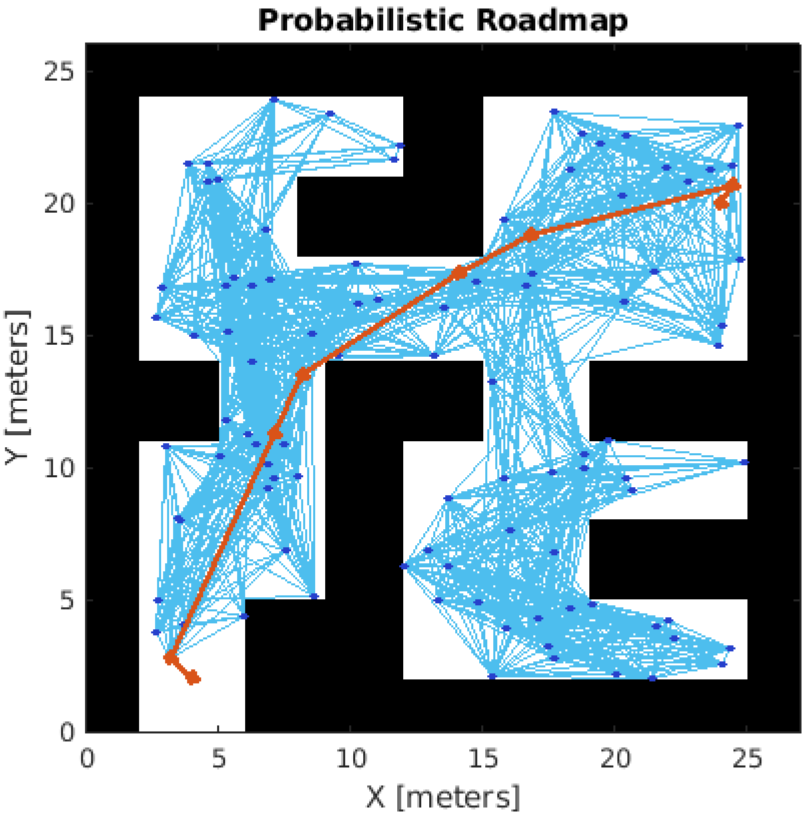
Find a path between the start and end location. Note that the `path` will be different due to the probabilistic nature of the PRM algorithm.

```
startLocation = [4.0 2.0];
endLocation = [24.0 20.0];
path = findpath(prm, startLocation, endLocation)
```

```
path = 8×2

    4.0000    2.0000
    3.1703    2.7616
    7.0797   11.2229
    8.1337   13.4835
   14.0707   17.3248
   16.8068   18.7834
   24.4564   20.6514
   24.0000   20.0000
```

Display the inflated map, the road maps, and the final path.

```
show(prm);
```



You defined a path following controller above which you can re-use for computing the control commands of a robot on this map. To re-use the controller and redefine the waypoints while keeping the other information the same, use the `release` function.

```
release(controller);
controller.Waypoints = path;
```

Set initial location and the goal of the robot as defined by the path

```
robotInitialLocation = path(1,:);
robotGoal = path(end,:);
```

Assume an initial robot orientation

```
initialOrientation = 0;
```

Define the current pose for robot motion [x y theta]

```
robotCurrentPose = [robotInitialLocation initialOrientation]';
```

Compute distance to the goal location

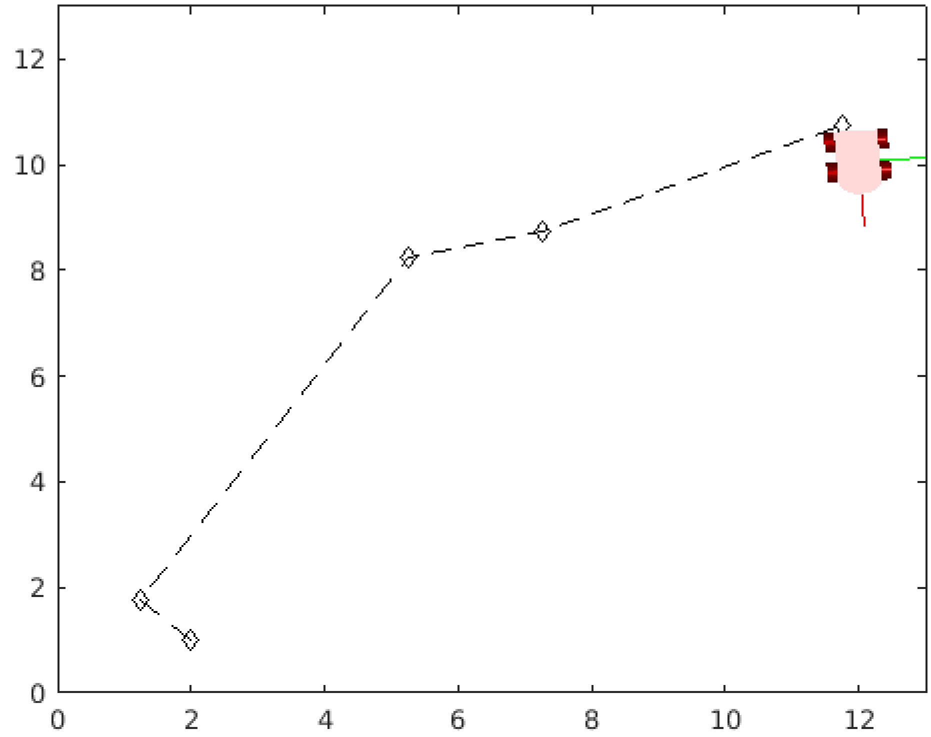
```
distanceToGoal = norm(robotInitialLocation - robotGoal);
```

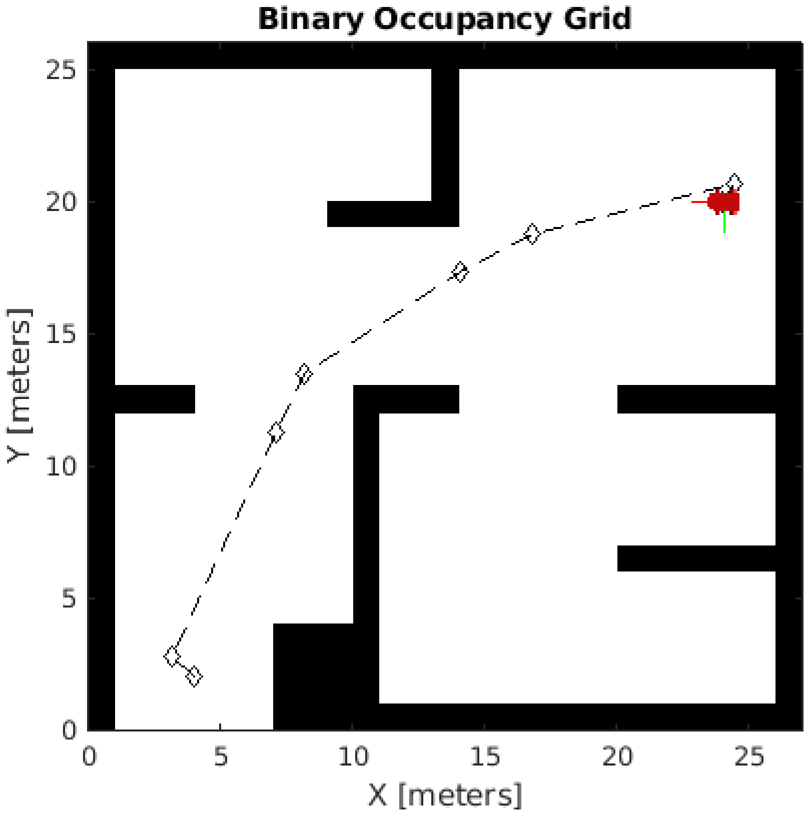
Define a goal radius

```
goalRadius = 0.1;
```

Drive the robot using the controller output on the given map until it reaches the goal. The controller runs at 10 Hz.

```
reset(vizRate);

% Initialize the figure
figure

while( distanceToGoal > goalRadius )

    % Compute the controller outputs, i.e., the inputs to the robot
    [v, omega] = controller(robotCurrentPose);

    % Get the robot's velocity using controller inputs
    vel = derivative(robot, robotCurrentPose, [v omega]);

    % Update the current pose
    robotCurrentPose = robotCurrentPose + vel*sampleTime;

    % Re-compute the distance to the goal
    distanceToGoal = norm(robotCurrentPose(1:2) - robotGoal(:));

    % Update the plot
    hold off
    show(map);
    hold all

    % Plot path each instance so that it stays persistent while robot mesh
    % moves
    plot(path(:,1), path(:,2),"k--d")

    % Plot the path of the robot as a set of transforms
    plotTrVec = [robotCurrentPose(1:2); 0];
    plotRot = axang2quat([0 0 1 robotCurrentPose(3)]);
    plotTransforms(plotTrVec', plotRot, 'MeshFilePath', 'groundvehicle.stl', 'Parent', gca, "View
    light;
    xlim([0 27])
    ylim([0 26])

    waitfor(vizRate);
end
```


0
2
4
6
8
10
12
0
2
4
6
8
10
12



See Also

- “Path Planning in Environments of Different Complexity” on page 1-2
- “Mapping with Known Poses” on page 1-19
- “Simulate Different Kinematic Models for Mobile Robots” on page 1-67

Mapping with Known Poses

This example shows how to create a map of an environment using range sensor readings and robot poses for a differential drive robot. You create a map from range sensor readings that are simulated using the `rangeSensor` object. The `differentialDriveKinematics` motion model simulates driving the robot around the room based on velocity commands. The `rangeSensor` gives range readings based on the pose of the robot as it follows the path.

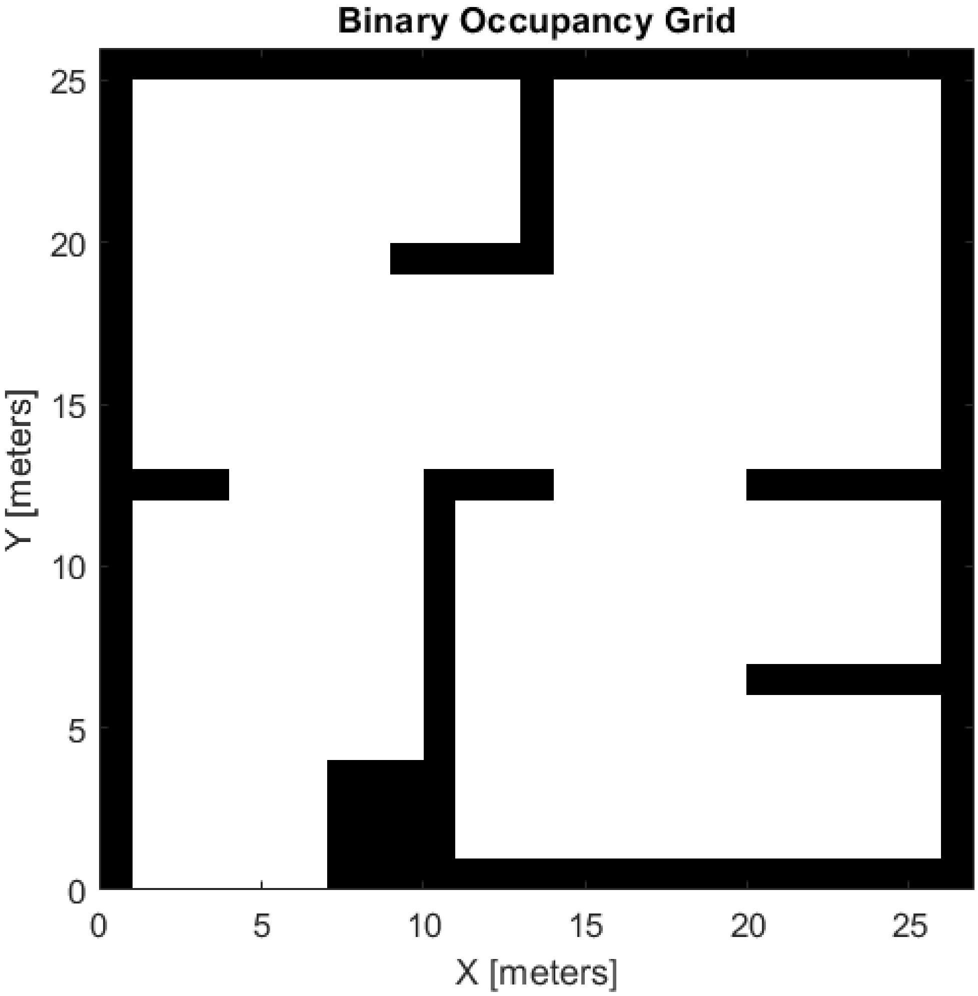
Reference Map and Figures

Load a set of example binary occupancy grids from `exampleMaps`, including `simpleMap`, which this example uses.

```
load exampleMaps.mat
```

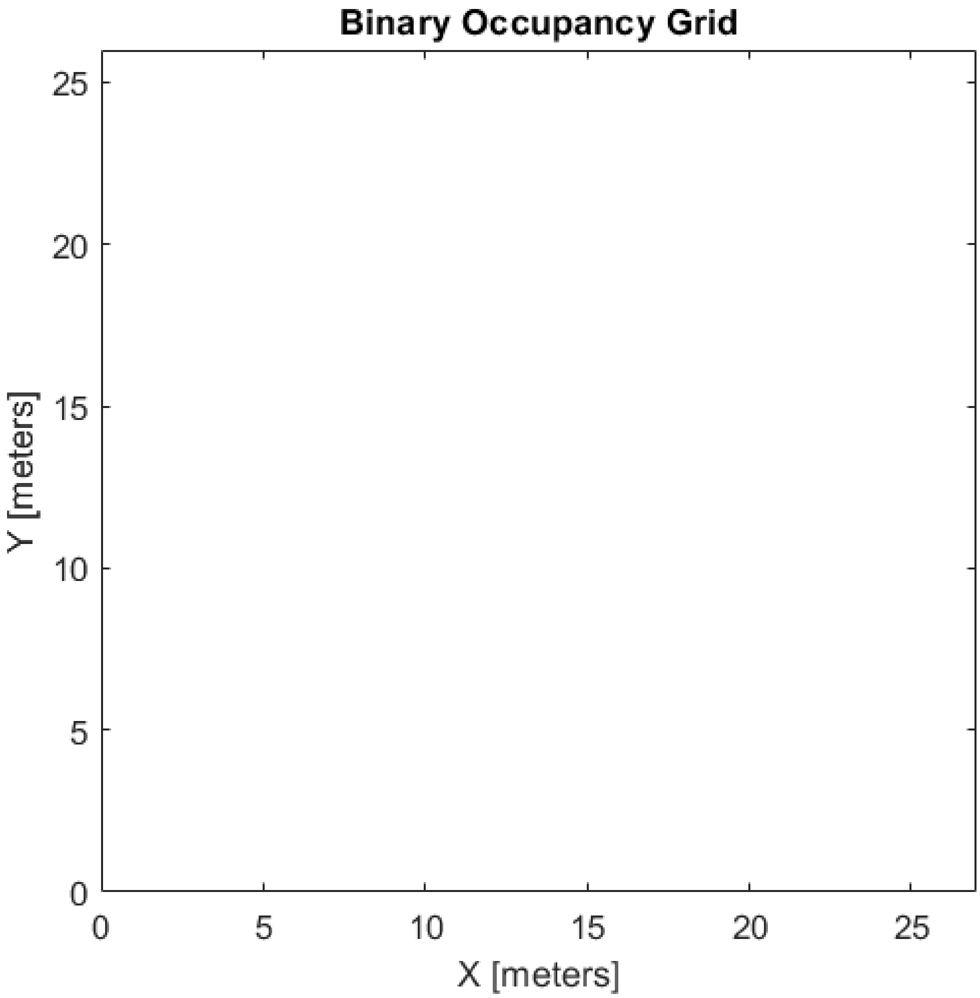
Create the reference binary occupancy map using `simpleMap` with a resolution of 1. Show the figure and save the handle of the figure.

```
refMap = binaryOccupancyMap(simpleMap,1);
refFigure = figure('Name','SimpleMap');
show(refMap);
```



Create an empty map of the same dimensions as the selected map with a resolution of 10. Show the figure and save the handle of the figure. Lock the axes at the size of the map.

```
[mapdimx,mapdimy] = size(simpleMap);
map = binaryOccupancyMap(mapdimy,mapdimx,10);
mapFigure = figure('Name','Unknown Map');
show(map);
```



Initialize Motion Model and Controller

Create a differential-drive kinematic motion model. The motion model represents the motion of the simulated differential-drive robot. This model takes left and right wheels speeds or linear and angular velocities for the robot heading. For this example, use the vehicle speed and heading rate for the `VehicleInputs`.

```
diffDrive = differentialDriveKinematics("VehicleInputs","VehicleSpeedHeadingRate");
```

Create a pure pursuit controller. This controller generates the velocity inputs for the simulated robot to follow a desired path. Set your desired linear velocity and maximum angular velocity, specified in meters per second and radians per second respectively.

```
controller = controllerPurePursuit('DesiredLinearVelocity',2,'MaxAngularVelocity',3);
```

Set Up Range Sensor

Create a sensor with a max range of 10 meters. This sensor simulates range readings based on a given pose and map. The reference map is used with this range sensor to simulate collecting sensor readings in an unknown environment.

```
sensor = rangeSensor;
sensor.Range = [0,10];
```

Create the Planned Path

Create a path to drive through the map for gathering range sensor readings.

```
path = [4 6; 6.5 12.5; 4 22; 12 14; 22 22; 16 12; 20 10; 14 6; 22 3];
```

Plot the path on the reference map figure.

```
figure(refFigure);
hold on
plot(path(:,1),path(:,2), 'o-');
hold off
```

Set the path as the waypoints of the pure pursuit controller.

```
controller.Waypoints = path;
```

Follow Path and Map Environment

Set the initial pose and final goal location based on the path. Create global variables for storing the current pose and an index for tracking the iterations.

```
initPose = [path(1,1) path(1,2), pi/2];
goal = [path(end,1) path(end,2)]';
poses(:,1) = initPose';
```

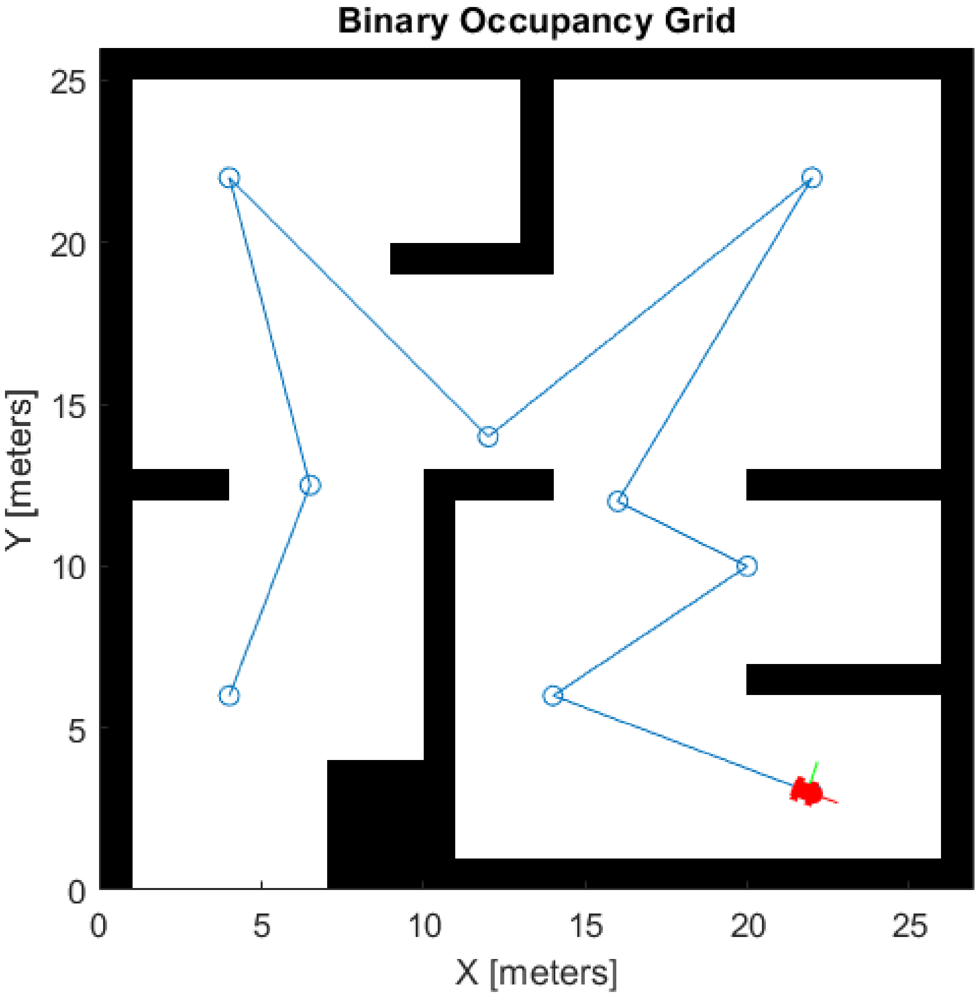
Use the provided helper function `exampleHelperDiffDriveCtrl`. The helper function contains the main loop for navigation the path, getting range readings, and mapping the environment.

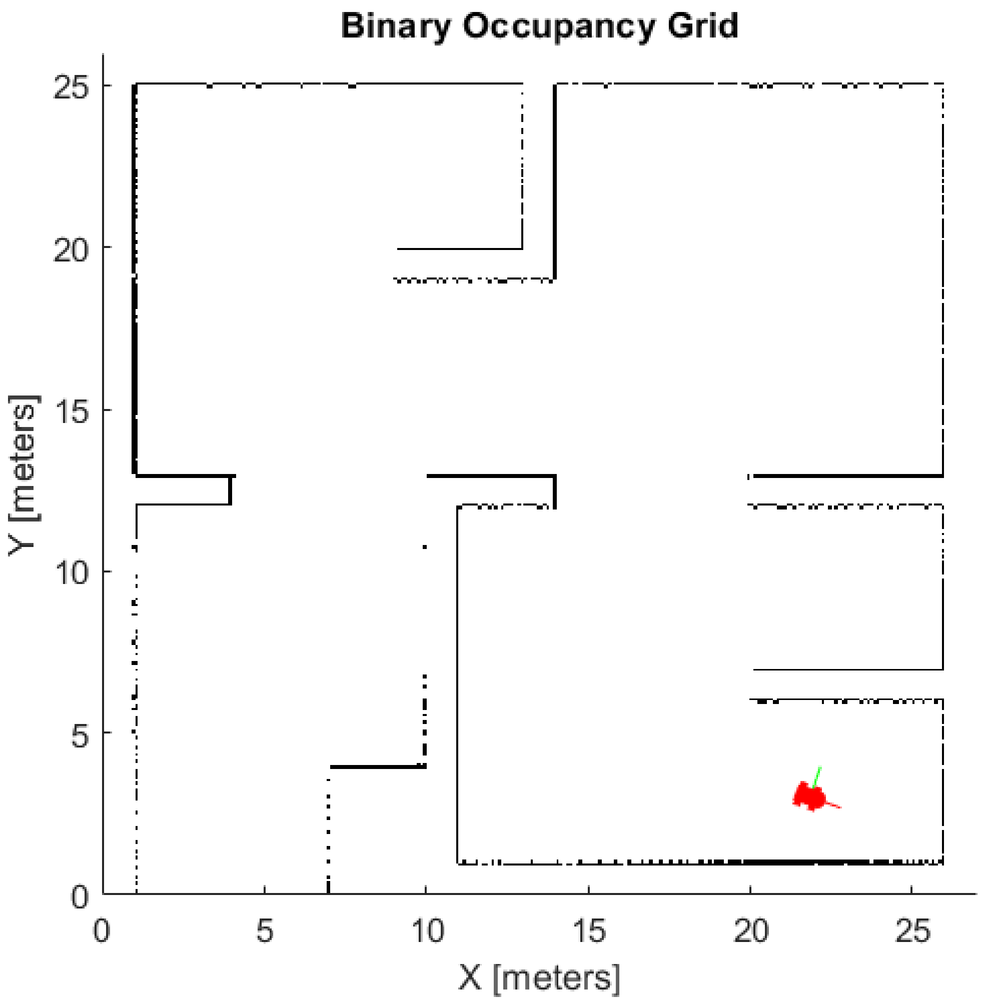
The `exampleHelperDiffDriveControl` function has the following workflow:

- Scan the reference map using the range sensor and the current pose. This simulates normal range readings for driving in an unknown environment.
- Update the map with the range readings.
- Get control commands from pure pursuit controller to drive to next waypoint.
- Calculate derivative of robot motion based on control commands.
- Increment the robot pose based on the derivative.

You should see the robot driving around the empty map and filling in walls as the range sensor detects them.

```
exampleHelperDiffDriveCtrl(diffDrive,controller,initPose,goal,refMap,map,refFigure,mapFigure,sens
```


Binary Occupancy Grid
25
20
15
10
5
0
Y [meters]
0
5
10
15
20
25
X [meters]



```
Goal position reached
```

Differential Drive Control Function

The `exampleHelperDiffDriveControl` function has the following workflow:

- Scan the reference map using the range sensor and the current pose. This simulates normal range readings for driving in an unknown environment.
- Update the map with the range readings.
- Get control commands from pure pursuit controller to drive to next waypoint.
- Calculate derivative of robot motion based on control commands.
- Increment the robot pose based on the derivative.

```
function exampleHelperDiffDriveControl(diffDrive,ppControl,initPose,goal,map1,map2,fig1,fig2,lida
sampleTime = 0.05;             % Sample time [s]
t = 0:sampleTime:100;         % Time array
poses = zeros(3,numel(t));    % Pose matrix
poses(:,1) = initPose';
```

```
% Set iteration rate
r = rateControl(1/sampleTime);

% Get the axes from the figures
ax1 = fig1.CurrentAxes;
ax2 = fig2.CurrentAxes;

    for idx = 1:numel(t)
        position = poses(:,idx)';
        currPose = position(1:2);

        % End if pathfollowing is vehicle has reached goal position within tolerance of 0.2m
        dist = norm(goal'-currPose);
        if (dist < .2)
            disp("Goal position reached")
            break;
        end

        % Update map by taking sensor measurements
        figure(2)
        [ranges, angles] = lidar(position, map1);
        scan = lidarScan(ranges,angles);
        validScan = removeInvalidData(scan,'RangeLimits',[0,lidar.Range(2)]);
        insertRay(map2,position,validScan,lidar.Range(2));
        show(map2);

        % Run the Pure Pursuit controller and convert output to wheel speeds
        [vRef,wRef] = ppControl(poses(:,idx));

        % Perform forward discrete integration step
        vel = derivative(diffDrive, poses(:,idx), [vRef wRef]);
        poses(:,idx+1) = poses(:,idx) + vel*sampleTime;


        % Update visualization
        plotTrvec = [poses(1:2, idx+1); 0];
        plotRot = axang2quat([0 0 1 poses(3, idx+1)]);

        % Delete image of the last robot to prevent displaying multiple robots
        if idx > 1
           items = get(ax1, 'Children');
           delete(items(1));
        end

        %plot robot onto known map
        plotTransforms(plotTrvec', plotRot, 'MeshFilePath', 'groundvehicle.stl', 'View', '2D', '
        %plot robot on new map
        plotTransforms(plotTrvec', plotRot, 'MeshFilePath', 'groundvehicle.stl', 'View', '2D', '

        % waiting to iterate at the proper rate
        waitfor(r);
    end
end
```

Plan Path for a Differential Drive Robot in Simulink

This example demonstrates how to execute an obstacle-free path between two locations on a given map in Simulink®. The path is generated using a probabilistic road map (PRM) planning algorithm (`mobileRobotPRM`). Control commands for navigating this path are generated using the **Pure Pursuit** controller block. A differential drive kinematic motion model simulates the robot motion based on those commands.

Load the Map and Simulink Model

Load the occupancy map, which defines the map limits and obstacles within the map. `exampleMaps.mat` contain multiple maps including `simpleMap`, which this example uses.

```
load exampleMaps.mat
```

Specify a start and end locaiton within the map.

```
startLoc = [5 5];
goalLoc = [20 20];
```

Model Overview

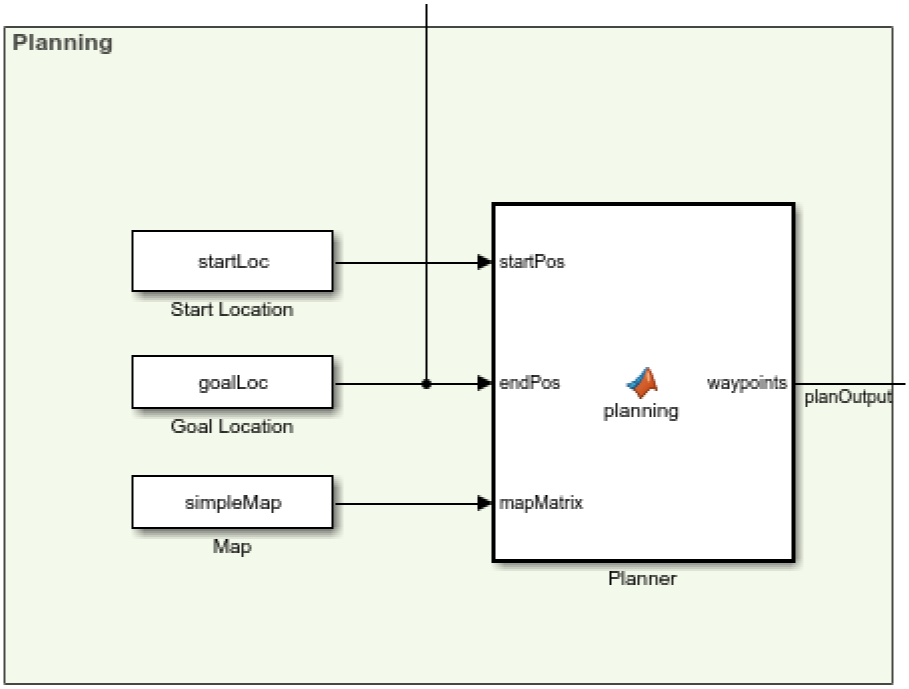
Open the Simulink model.

```
open_system('pathPlanningSimulinkModel.slx')
```

The model is composed of three primary parts:

- **Planning**
- **Control**
- **Plant Model**

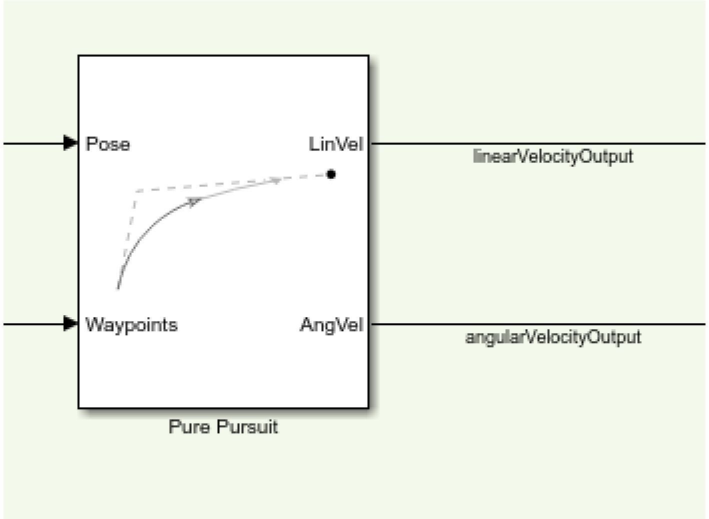
Planning



The **Planner** MATLAB® function block uses the `mobileRobotPRM` path planner and takes a start location, goal location, and map as inputs. The blocks outputs an array of waypoints that the robot follows. The planned waypoints are used downstream by the **Pure Pursuit** controller block.

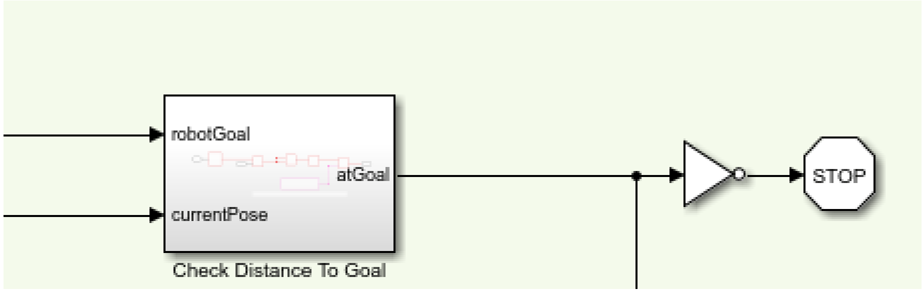
Control

Pure Pursuit



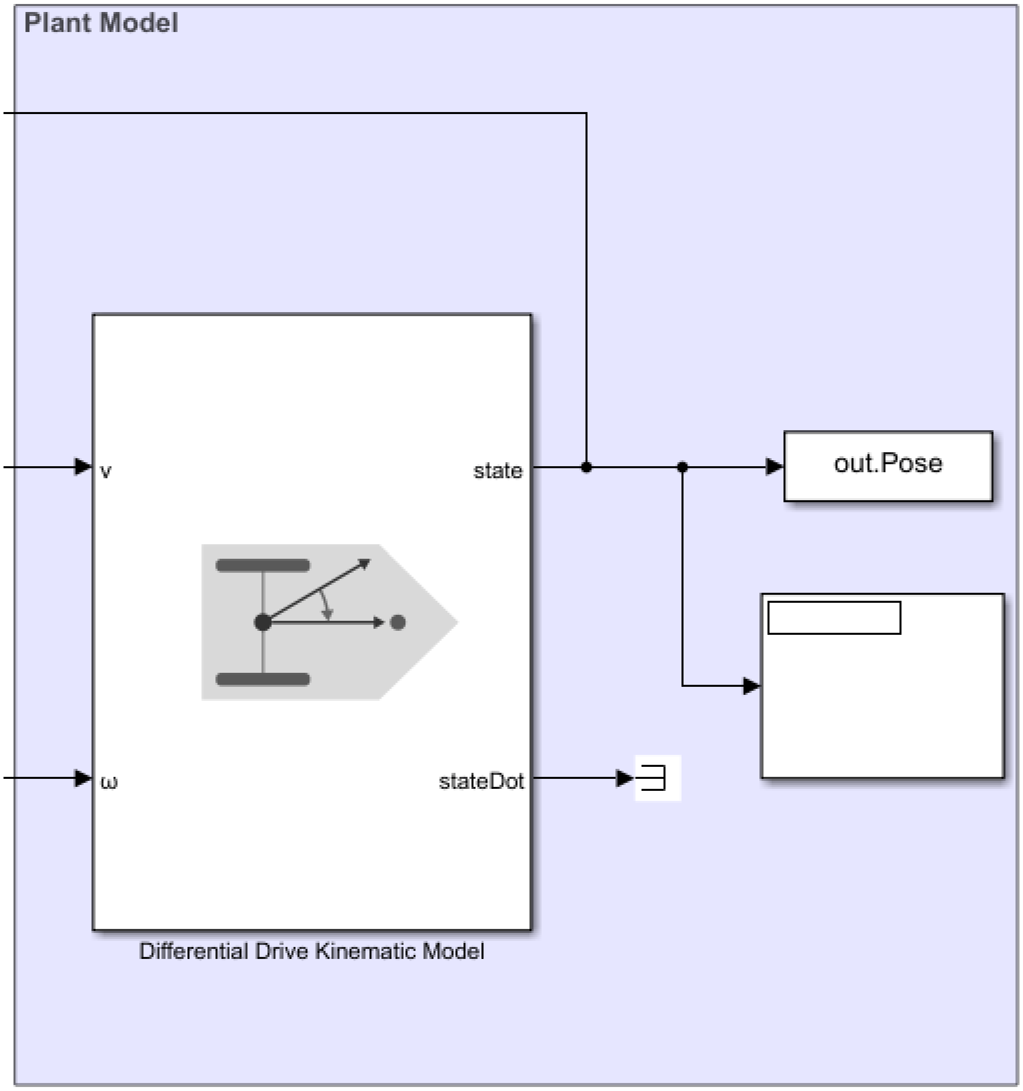
The **Pure Pursuit** controller block generates the linear velocity and angular velocity commands based on the waypoints and the current pose of the robot.

Check if Goal is Reached



The **Check Distance to Goal** subsystem calculates the current distance to the goal and if it is within a threshold, the simulation stops.

Plant Model



The **Differential Drive Kinematic Model** block creates a vehicle model to simulate simplified vehicle kinematics. The block takes linear and angular velocities as command inputs from the **Pure Pursuit** controller block, and outputs the current position and velocity states.

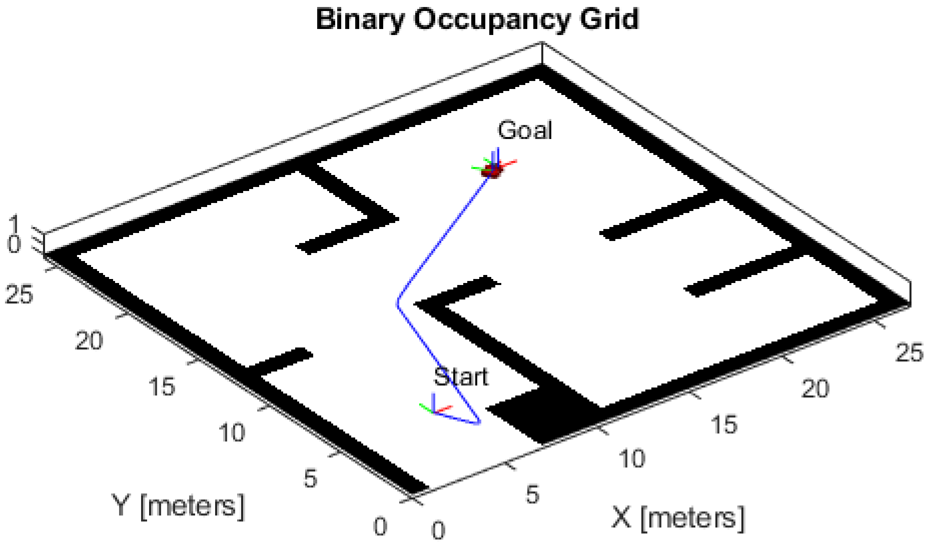
Run the Model

```
simulation = sim('pathPlanningSimulinkModel.slx');
```

Visualize The Motion of Robot

After simulating the model, visualize the robot driving the obstacle-free path in the map.

```
map = binaryOccupancyMap(simpleMap);
robotPose = simulation.Pose;
thetaIdx = 3;

% Translation
xyz = robotPose;
xyz(:, thetaIdx) = 0;

% Rotation in XYZ euler angles
theta = robotPose(:,thetaIdx);
thetaEuler = zeros(size(robotPose, 1), 3 * size(theta, 2));
thetaEuler(:, end) = theta;

% Plot the robot poses at every 10th step.
for k = 1:10:size(xyz, 1)
    show(map)
    hold on;

    % Plot the start location.
    plotTransforms([startLoc, 0], eul2quat([0, 0, 0]))
    text(startLoc(1), startLoc(2), 2, 'Start');

    % Plot the goal location.
    plotTransforms([goalLoc, 0], eul2quat([0, 0, 0]))
    text(goalLoc(1), goalLoc(2), 2, 'Goal');

    % Plot the xy-locations.
    plot(robotPose(:, 1), robotPose(:, 2), '-b')

    % Plot the robot pose as it traverses the path.
    quat = eul2quat(thetaEuler(k, :), 'xyz');
    plotTransforms(xyz(k,:), quat, 'MeshFilePath',...
        'groundvehicle.stl');
    light;
    drawnow;
    hold off;
end
```

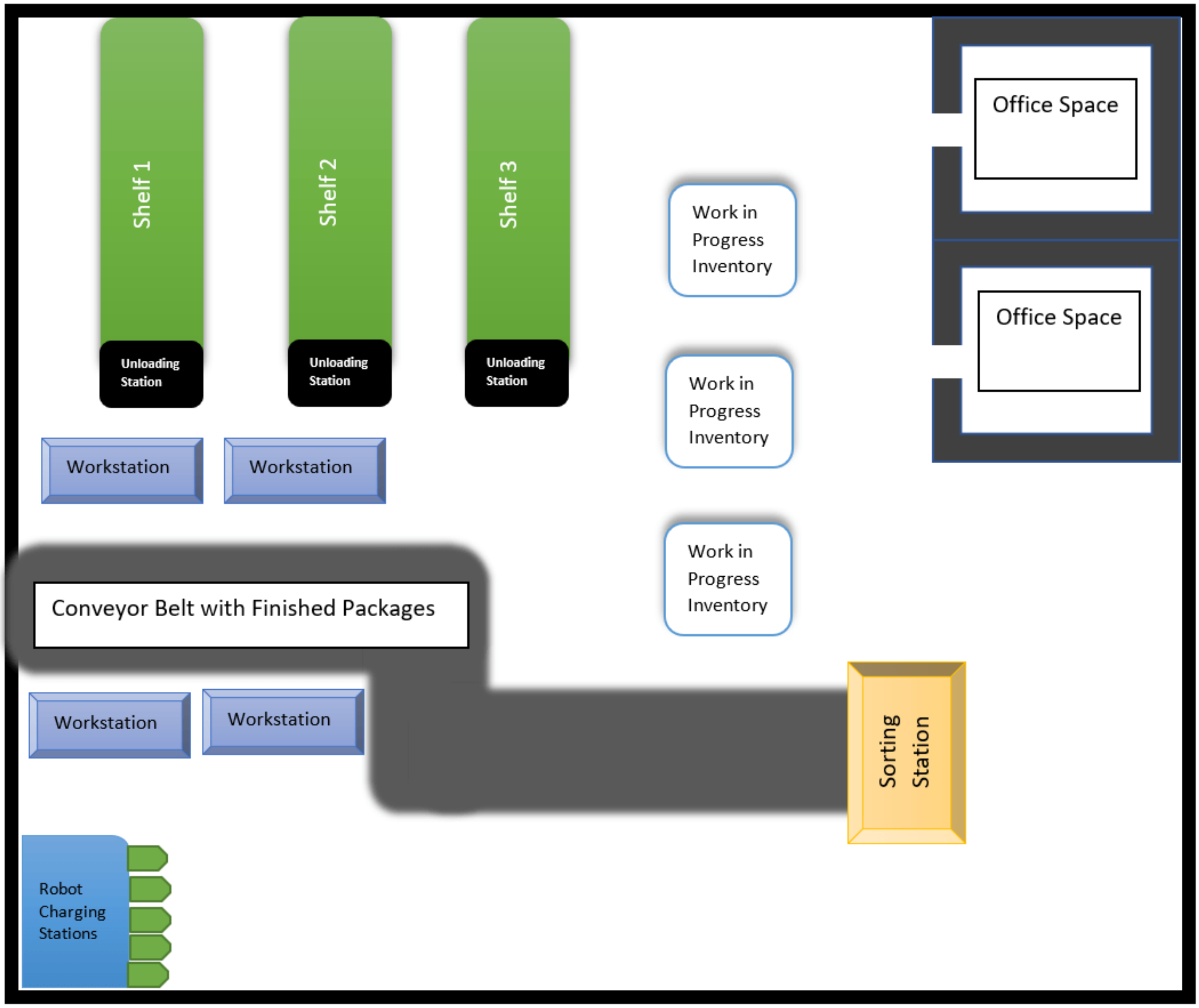

Binary Occupancy Grid
Goal
Start
1
0
25
20
15
10
5
0
Y [meters]
0
5
10
15
20
25
X [meters]

Execute Tasks for a Warehouse Robot

This example demonstrates how to execute an obstacle-free path for a mobile robot between three locations on a given map. The robot is expected to visit the three locations in a warehouse: a charging station, loading station, and unloading location. The sequence in which these locations are visited is dictated by a scheduler. The scheduler gives each robot a goal pose to navigate to. The robot plans a path and uses a Pure Pursuit controller to follow the waypoints based on the current pose of the robot. The **Differential Drive Kinematic Model** block models the simplified kinematics, which takes the linear and angular velocities from the Pure Pursuit Controller. This example builds on top of the “Plan Path for a Differential Drive Robot in Simulink” on page 1-26 example.

Warehouse Map

A typical warehouse of a sorting or distribution facility has packages to be delivered from work stations to storage areas. The warehouse may have off-limit areas like offices and in-progress inventory blocking aisles or walkways. The robots are tasked with picking finished packages as they arrive at the sorting station and are told a location to store them. The warehouse also has a charging station for recharging the robots after a certain time.



This sample warehouse floor-plan can be translated into a binary occupancy map, which indicates all the safe regions in the warehouse facility.

Load the example map file. `logicalMap` is a matrix of logical values indicating free space in the warehouse. Make a `binaryOccupancyMap` from this matrix.

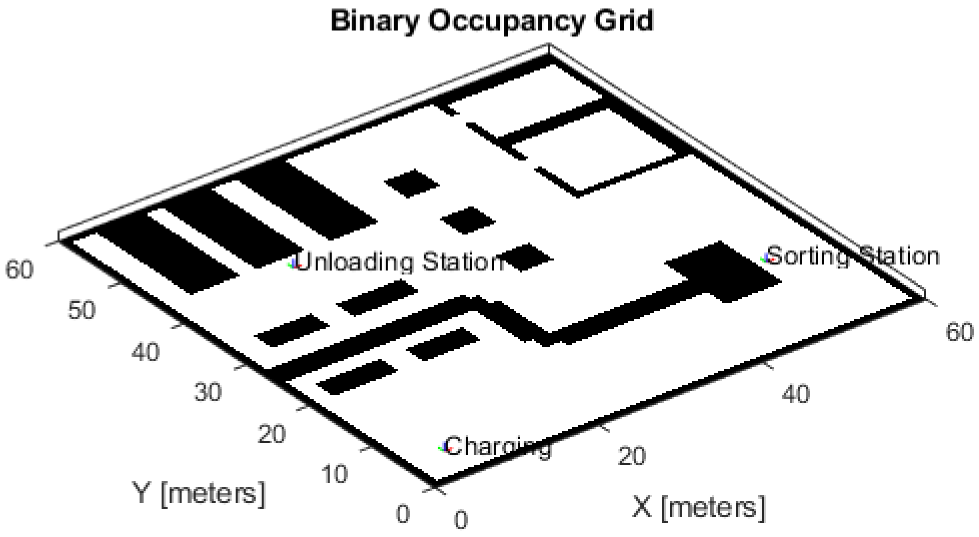
```
load warehouseMaps.mat logicalMap
map = binaryOccupancyMap(logicalMap);
show(map)
```

Assign the *xy*-locations of the charging station, sorting (loading) station, and the unloading location near shelves in the warehouse.

```
chargingStn = [5,5];
loadingStn = [52,15];
unloadingStn = [15,42];
```

Show the various locations on the map

```
hold on;

text(chargingStn(1), chargingStn(2), 1, 'Charging');
plotTransforms([chargingStn, 0], [1 0 0 0])

text(loadingStn(1), loadingStn(2), 1, 'Sorting Station');
plotTransforms([loadingStn, 0], [1 0 0 0])

text(unloadingStn(1), unloadingStn(2), 1, 'Unloading Station');
plotTransforms([unloadingStn, 0], [1 0 0 0])

hold off;
```

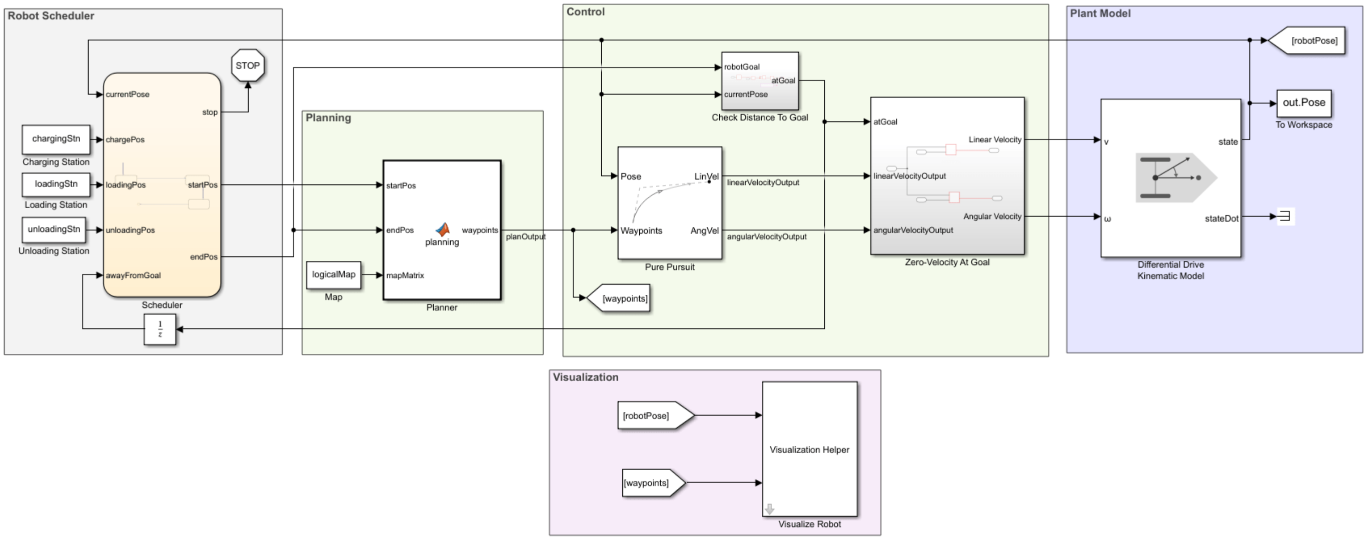


Model Overview

A Simulink® model is provided that models all aspects of the system for scheduling, planning, controlling, and modelling the robot behavior.

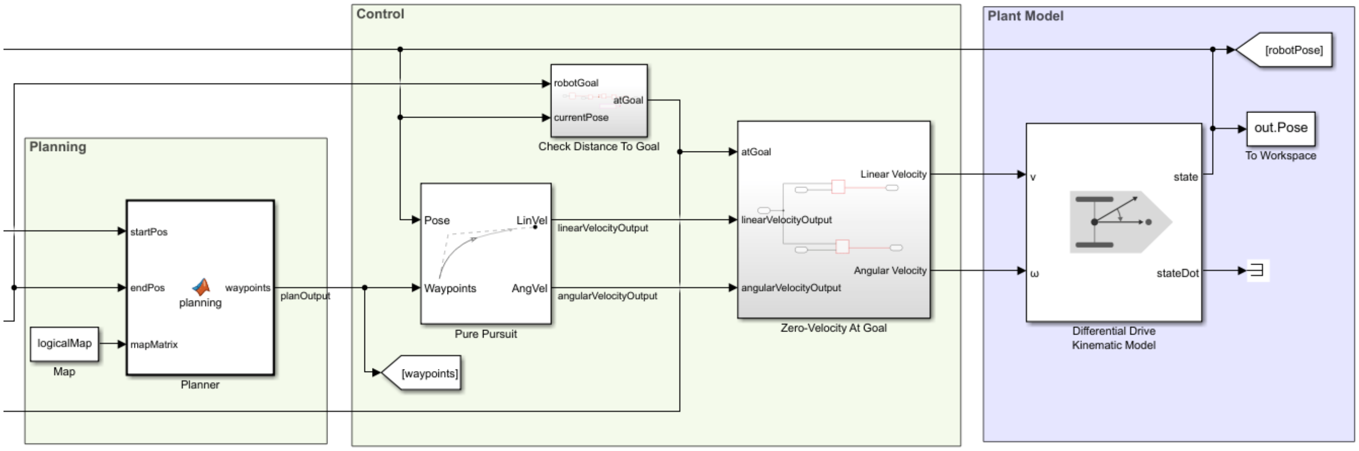
Open the Simulink Model.

```
open_system('warehouseTasksRobotSimulationModel.slx')
```



Planning, Control, and Plant Model

The model uses a planning, control, and plant model similar to the “Plan Path for a Differential Drive Robot in Simulink” on page 1-26 example. The planner takes the start and goal locations from the scheduler and plans an obstacle-free path between them based on the given map. The controller uses a Pure Pursuit controller for generating the linear and angular velocity controls of the robot to navigation the path. These controls are given to the plant model that models the behavior of a differential-drive robot.

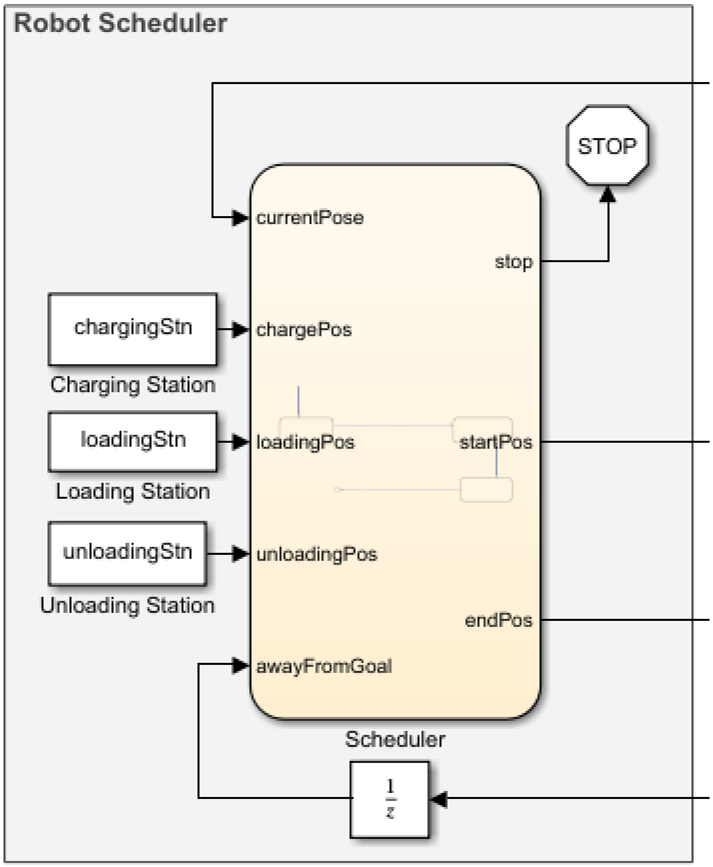


Robot Scheduler

The Scheduler block assigns start and goal locations to the robot. The current pose of the robot is used as a starting location and the end location is determined by a sequence of tasks specified inside the scheduler. The example illustrates the following sequence of tasks for the robot:

1. Starts from the charging location, and goes to the loading location.
2. Pauses as the loading station to load the package and plans a path to the unloading location.

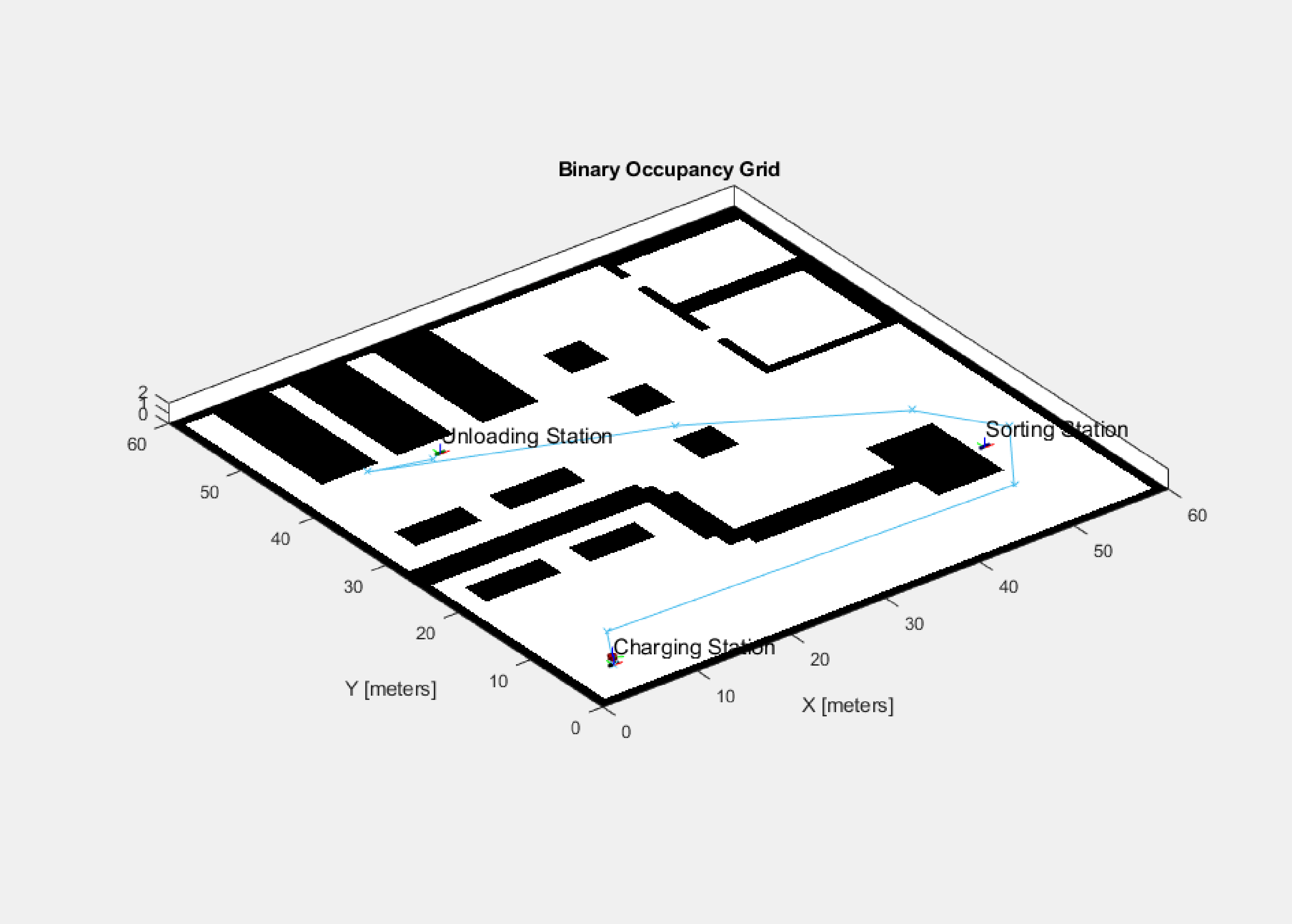
3. Navigates to the unloading station to unload the package. Replans a path to the charging station.
4. Stops at the charging station.



Simulate The Robot

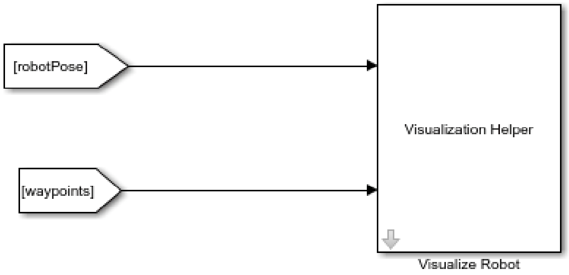
Run the simulation to see the robot execute the tasks.

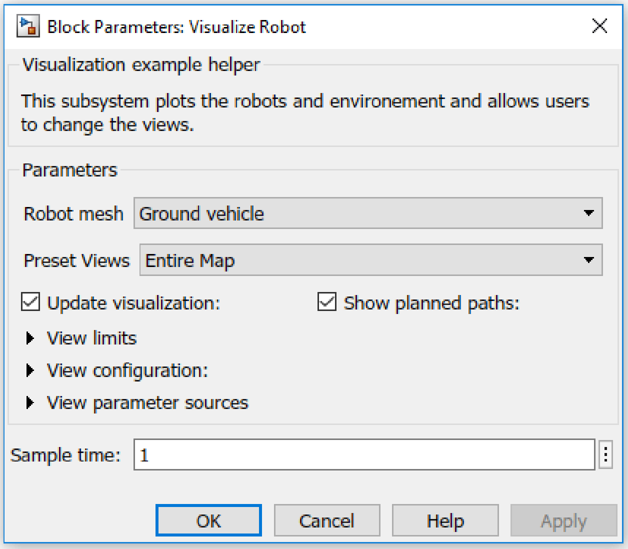
```
simulation = sim('warehouseTasksRobotSimulationModel.slx');
```



Visualize Robot Trajectories

A custom visualization tool is given to mimic a distributed camera system and get more detailed views of the robot trajectory at certain locations in the map. Open the **Visualization Helper** block and use the **Preset Views** drop-down to select different perspectives. The `Sample time` of the visualization has no effect on the simulation of the robot.




Block Parameters: Visualize Robot

Visualization example helper

This subsystem plots the robots and environement and allows users to change the views.

Parameters

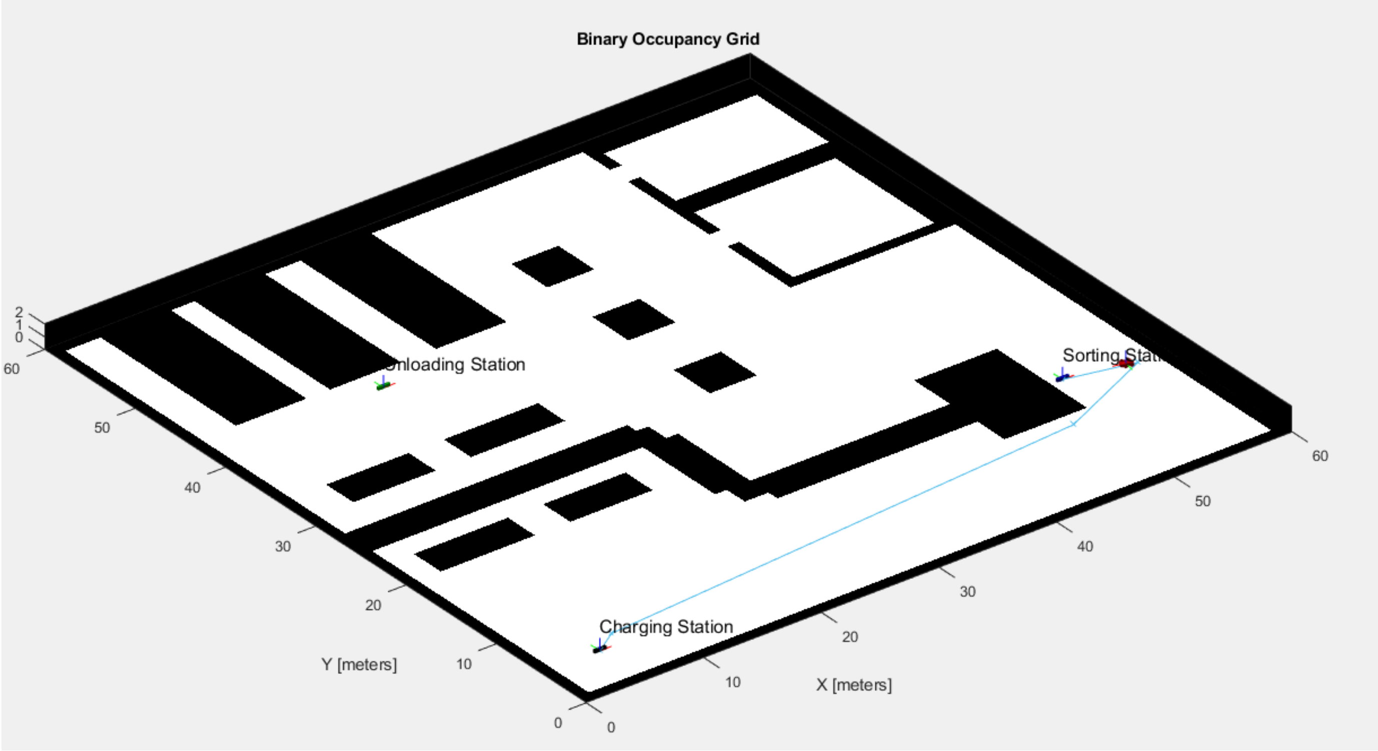
Robot mesh: Ground vehicle

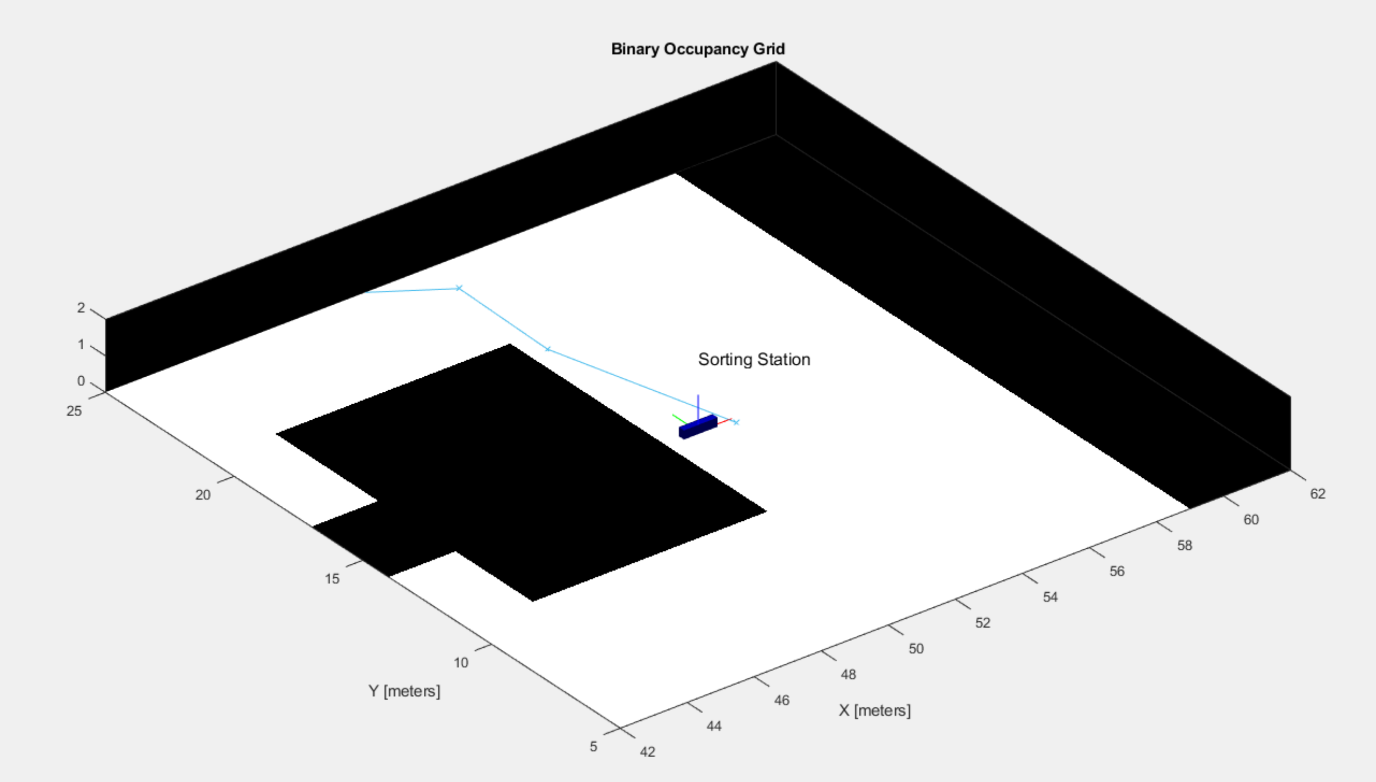
Preset Views: Entire Map

☑ Update visualization: ☑ Show planned paths:

- View limits
- View configuration:
- View parameter sources

Sample time: 1





See Also

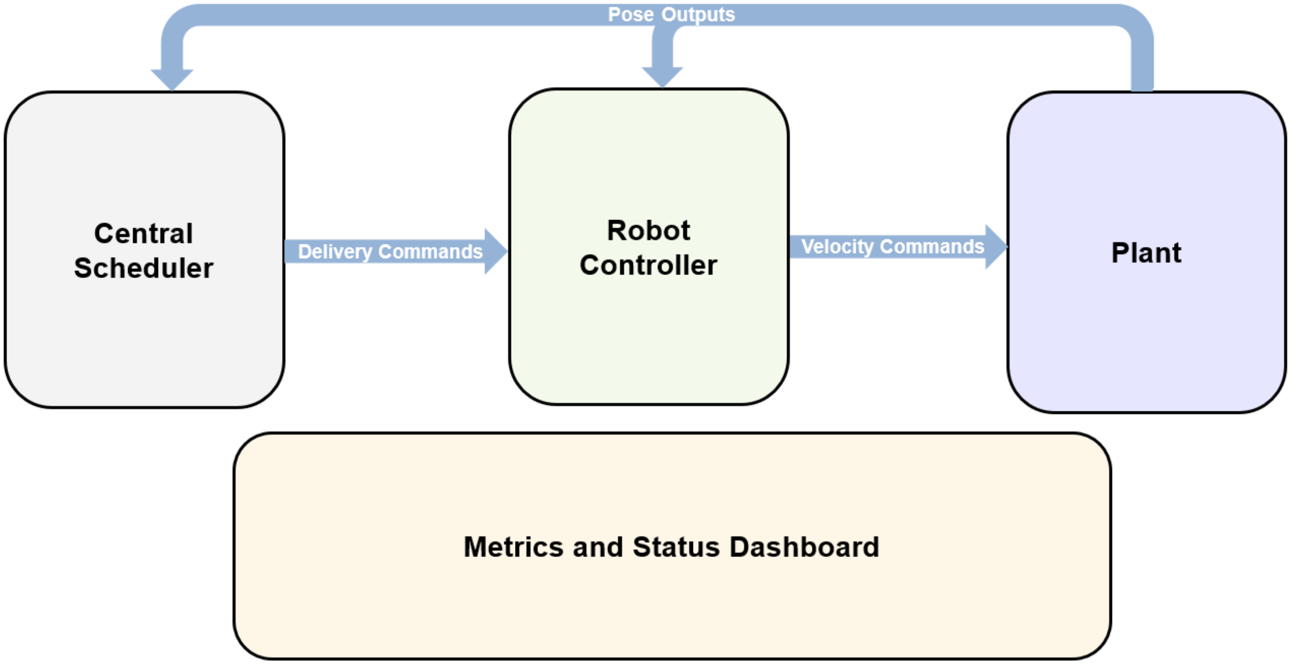
- “Plan Path for a Differential Drive Robot in Simulink” on page 1-26

- “Control and Simulate Multiple Warehouse Robots” on page 1-40

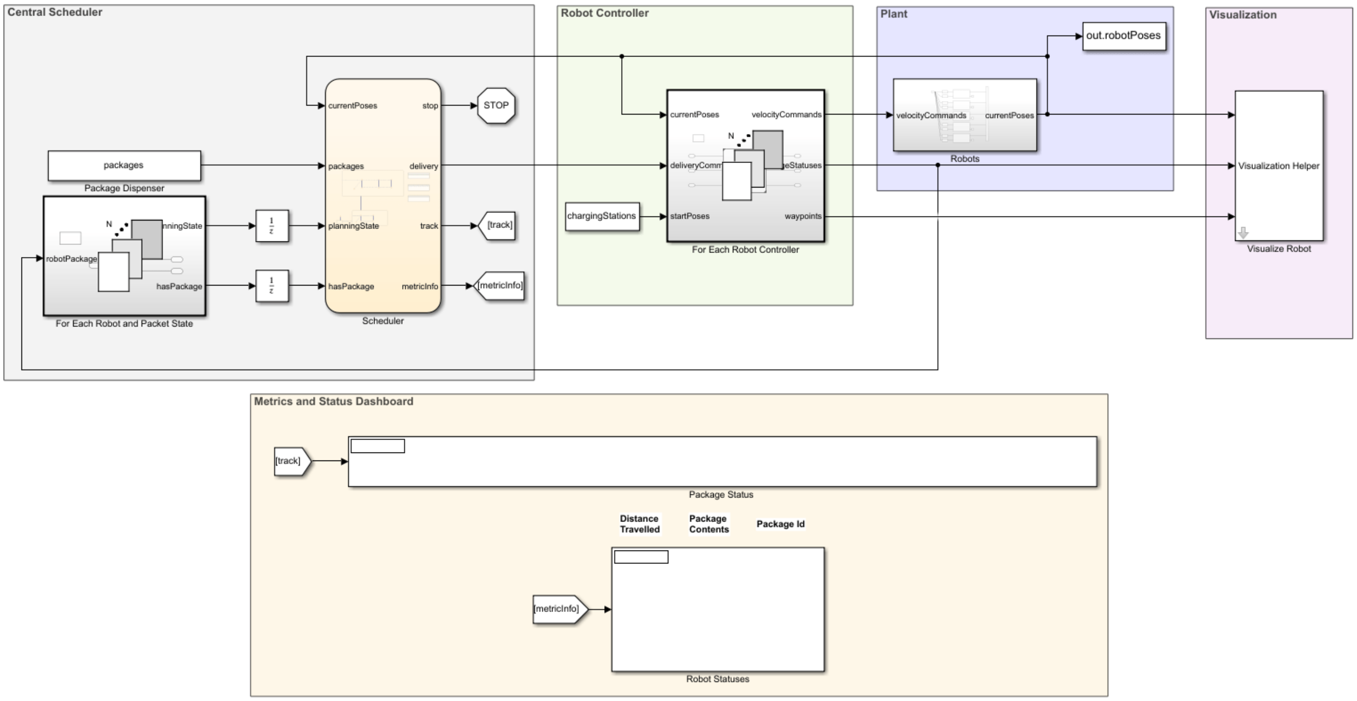
Control and Simulate Multiple Warehouse Robots

This example shows how to control and simulate multiple robots working in a warehouse facility or distribution center. The robots drive around the facility picking up packages and delivering them to stations for storing or processing. This example builds on top of the “Execute Tasks for a Warehouse Robot” on page 1-31 example, which drives a single robot around the same facility.

This package-sorting scenario can be modeled in Simulink® using Stateflow charts and Robotics System Toolbox™ algorithm blocks. A **Central Scheduler** sends commands to robots to pick up packages from the *loading station* and deliver them to a specific *unloading station*. The **Robot Controller** plans the trajectory based on the locations of the loading and unloading stations, and generates velocity commands for the robot. These commands are fed to the **Plant**, which contains a differential-drive robot model for executing the velocity commands and returning ground-truth poses of the robot. The poses are fed back to the scheduler and controller for tracking the robot status. This workflow is done for a group of 5 robots, which are all scheduled, tracked, and modeled simultaneously.



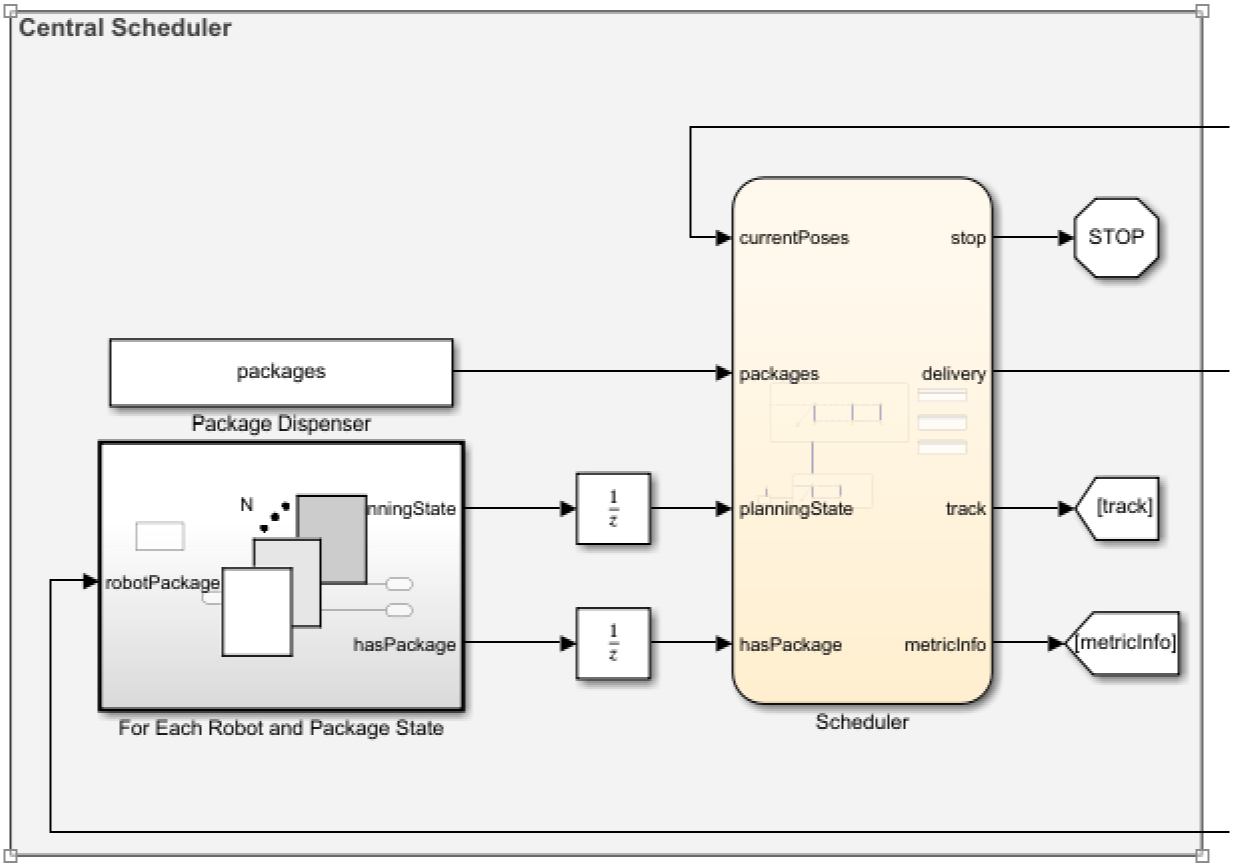
The provided Simulink model, `multiRobotExampleModel`, models the above described scenario.



Central Scheduler

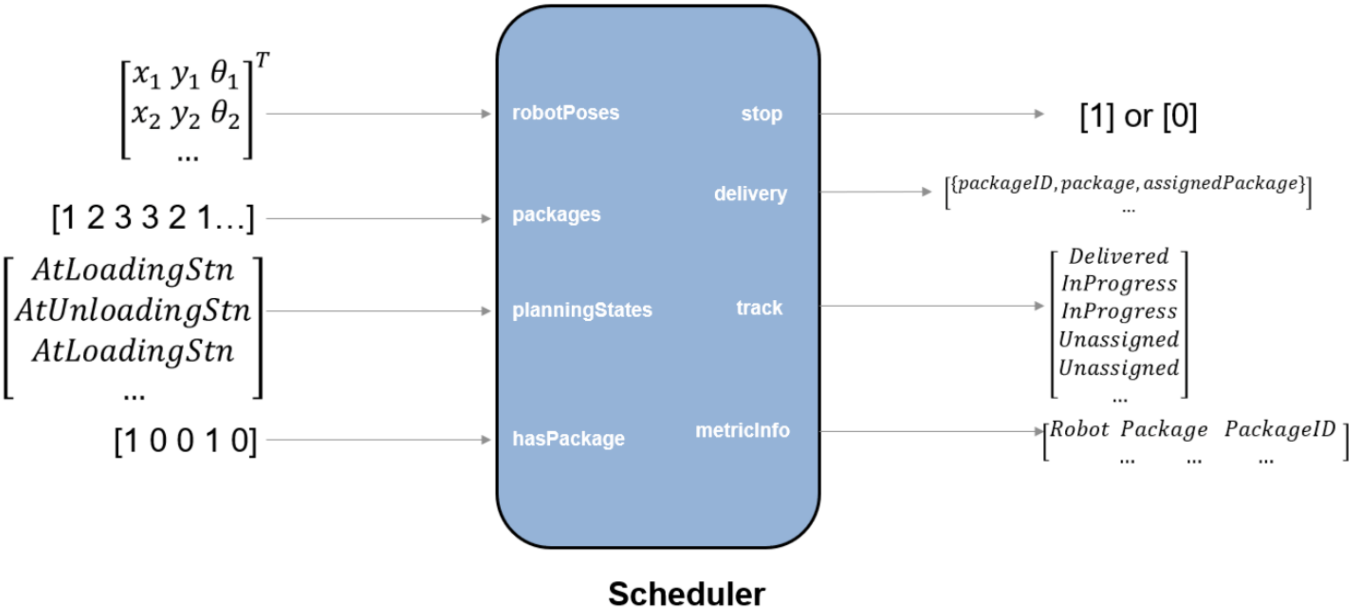
The Central Scheduler uses a Stateflow chart to handle package allocation to the robots from the **Package Dispenser**. Each robot can carry one package at a time and is instructed to go from the loading to an unloading station based on the required location for each package. The scheduler also tracks the status of the packages and robots and updates the **Status Dashboard**. Based on robot poses, the scheduler also sends stop commands to one robot when it detects an imminent collision. This behavior can allow the robots to run local obstacle avoidance if available.

The **For Each Robot and Package State** subsystem is a For Each Subsystem (Simulink) which processes an array of buses for tracking the robot and package states as `RobotPackageStatus` bus object. This makes it easy to update this model for varying number of robots. For more information about processing arrays of buses using a For-Each Subsystem, see “Work with Arrays of Buses” (Simulink).



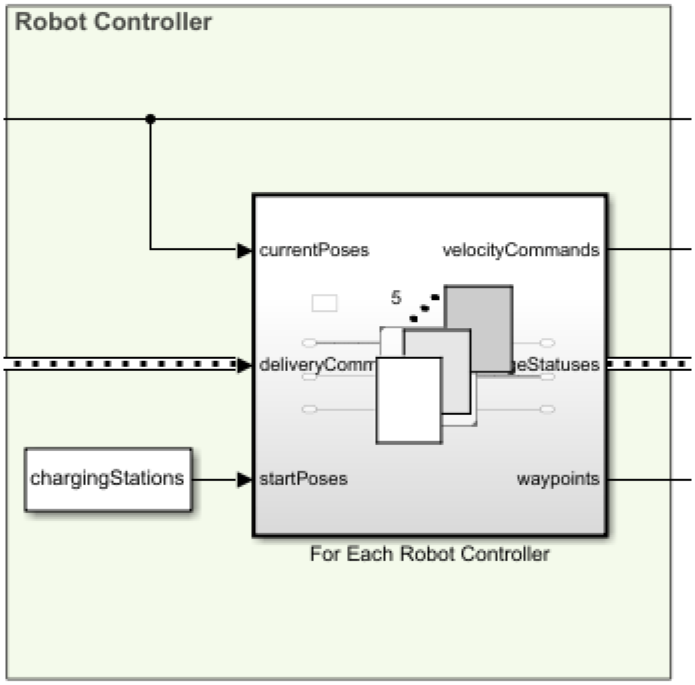
Scheduler

The following schematic details the signal values of the **Scheduler** Stateflow chart.

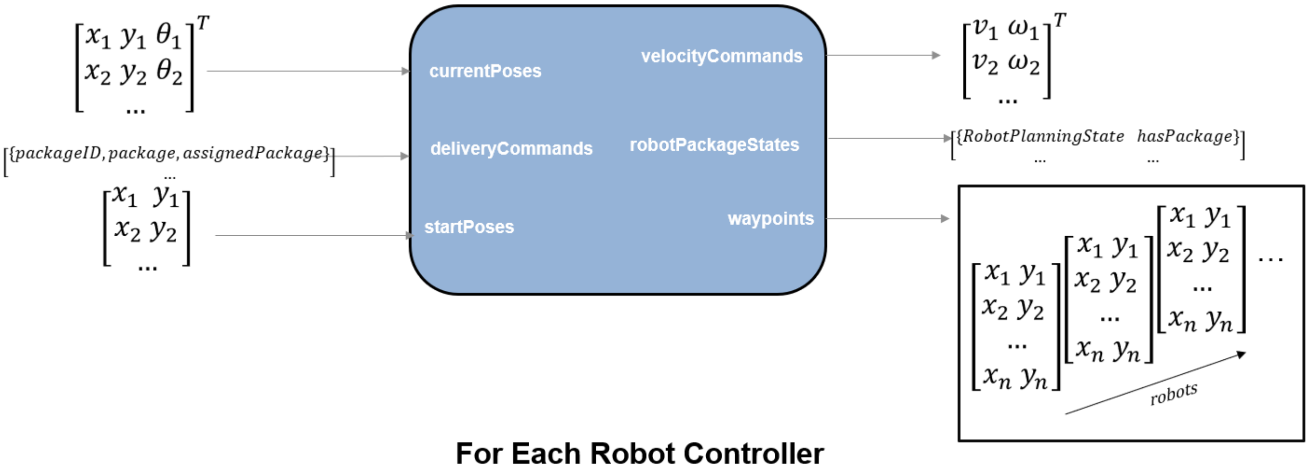

$\begin{bmatrix} x_1\ y_1\ \theta_1 \\ x_2\ y_2\ \theta_2 \\ \dots \end{bmatrix}^T$
[1 2 3 3 2 1…]
$\begin{bmatrix} AtLoadingStn \\ AtUnloadingStn \\ AtLoadingStn \\ \dots \end{bmatrix}$
[1 0 0 1 0]
robotPoses
packages
planningStates
hasPackage
stop
delivery
track
metricInfo
[1] or [0]
$\begin{bmatrix} \{packageID, package, assignedPackage\} \\ \dots \end{bmatrix}$
$\begin{bmatrix} Delivered \\ InProgress \\ InProgress \\ Unassigned \\ Unassigned \\ \dots \end{bmatrix}$
$\begin{bmatrix} Robot & Package & PackageID \\ \dots & \dots & \dots \end{bmatrix}$
Scheduler

Robot Controller

The **Robot Controller** uses a For Each Subsystem (Simulink) to generate an array of robot controllers for your 5 robots.

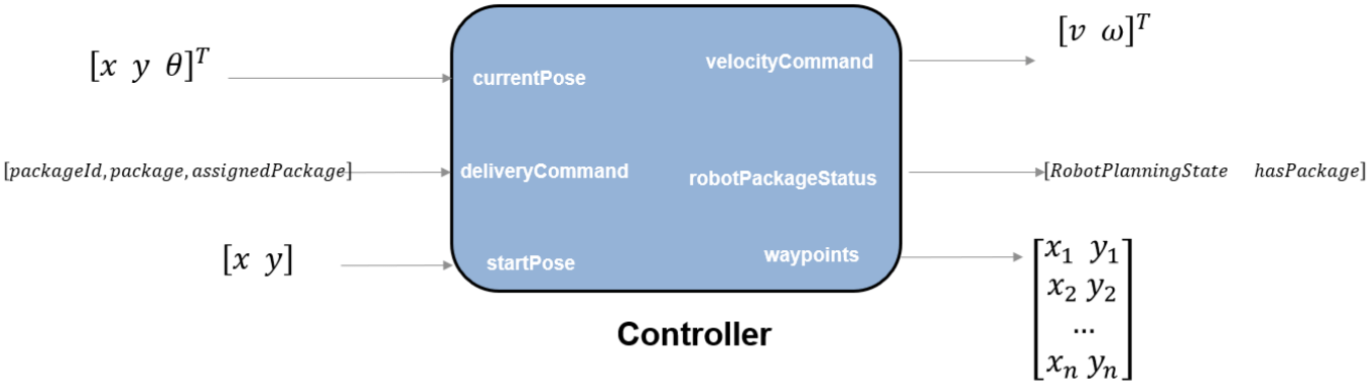


The following schematic details the type of signal values associated with the **For Each Robot Controller**.


$\begin{bmatrix} x_1 & y_1 & \theta_1 \\ x_2 & y_2 & \theta_2 \\ & \dots & \end{bmatrix}^T$
$\begin{bmatrix} \{packageID, package, assignedPackage\} \\ \dots \end{bmatrix}$
$\begin{bmatrix} x_1 & y_1 \\ x_2 & y_2 \\ \dots \end{bmatrix}$
currentPoses
deliveryCommands
startPoses
velocityCommands
robotPackageStates
waypoints
$\begin{bmatrix} v_1 & \omega_1 \\ v_2 & \omega_2 \\ \dots \end{bmatrix}^T$
$\begin{bmatrix} \{RobotPlanningState & hasPackage\} \\ \dots & \dots \end{bmatrix}$
$\begin{bmatrix} x_1 & y_1 \\ x_2 & y_2 \\ \dots \\ x_n & y_n \end{bmatrix} \begin{bmatrix} x_1 & y_1 \\ x_2 & y_2 \\ \dots \\ x_n & y_n \end{bmatrix} \begin{bmatrix} x_1 & y_1 \\ x_2 & y_2 \\ \dots \\ x_n & y_n \end{bmatrix} \dots$
robots

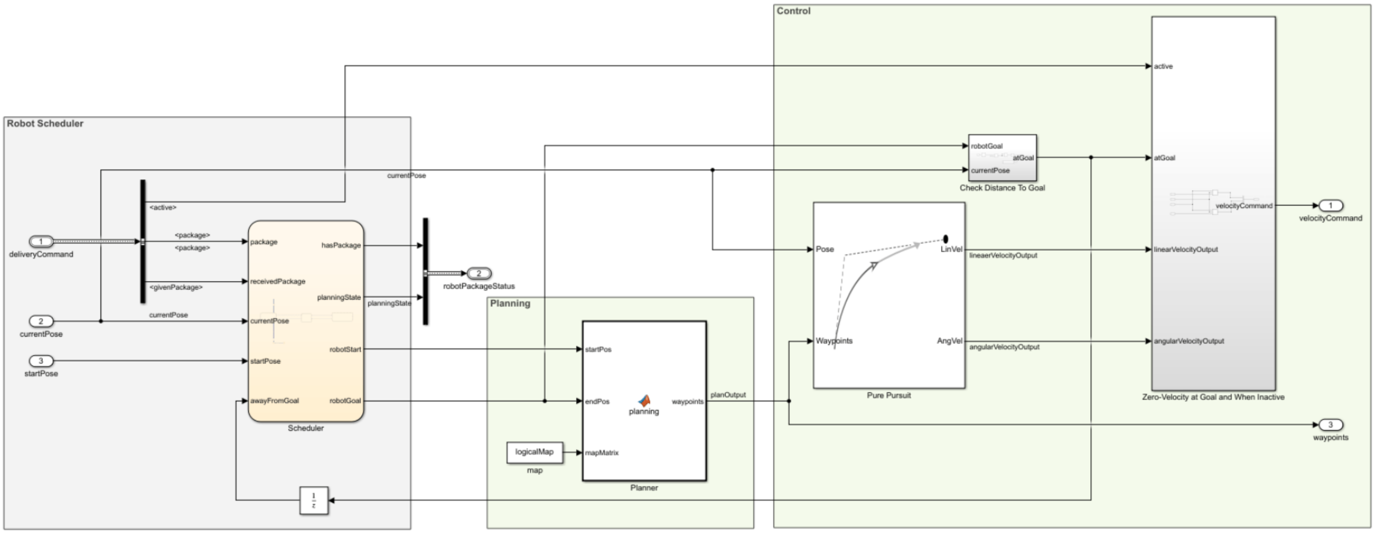
For Each Robot Controller

Each robot controller has the following inputs and outputs.


$[x\ y\ \theta]^T$
currentPose
velocityCommand
$[v\ \omega]^T$
$[packageId, package, assignedPackage]$
deliveryCommand
robotPackageStatus
$[RobotPlanningState\quad hasPackage]$
$[x\ y]$
startPose
waypoints
$\begin{bmatrix} x_1 & y_1 \\ x_2 & y_2 \\ \multicolumn{2}{c}{\dots} \\ x_n & y_n \end{bmatrix}$
Controller

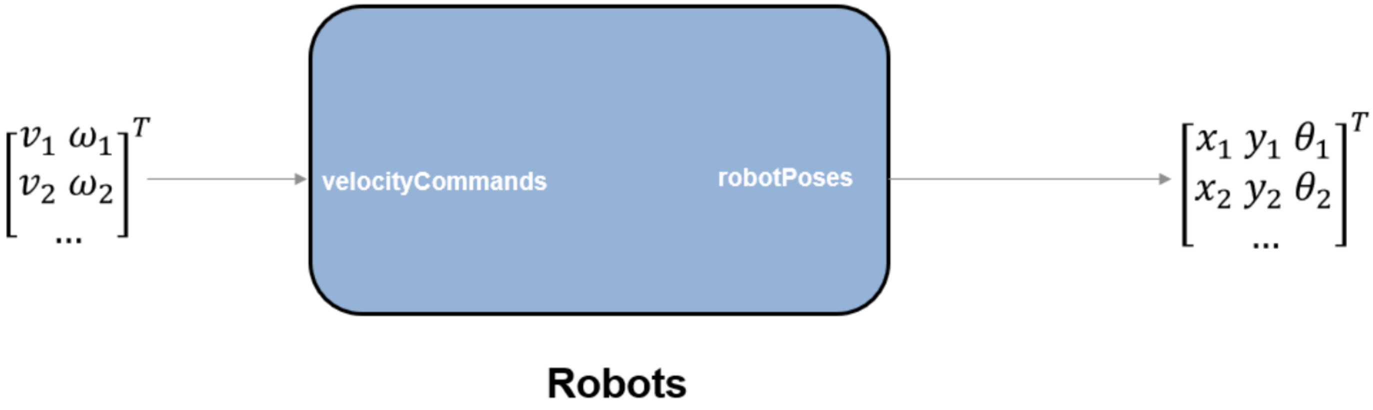
The controller takes delivery commands, which contains the package information, and plans a path for delivering it someone in the warehouse using `mobileRobotPRM`. The **Pure Pursuit** block takes this path and generates velocity commands for visiting each waypoint. Also, the status of the robot and packages get updated when the robot reaches its goal. Each robot also has its own internal scheduler that tells them the location of unloading stations based on the package information, and sends them back to the loading station when they drop off a package.

The robot controller model uses the same model, `warehouseTasksRobotSimulationModel`, shown in “Execute Tasks for a Warehouse Robot” on page 1-31.



Plant

The **Plant** subsystem uses a **Differential Drive Kinematic Model** block to model the motion of the robots.


$\begin{bmatrix} v_1 & \omega_1 \\ v_2 & \omega_2 \\ & \cdots \end{bmatrix}^T$
velocityCommands
robotPoses
$\begin{bmatrix} x_1 & y_1 & \theta_1 \\ x_2 & y_2 & \theta_2 \\ & \cdots \end{bmatrix}^T$
Robots

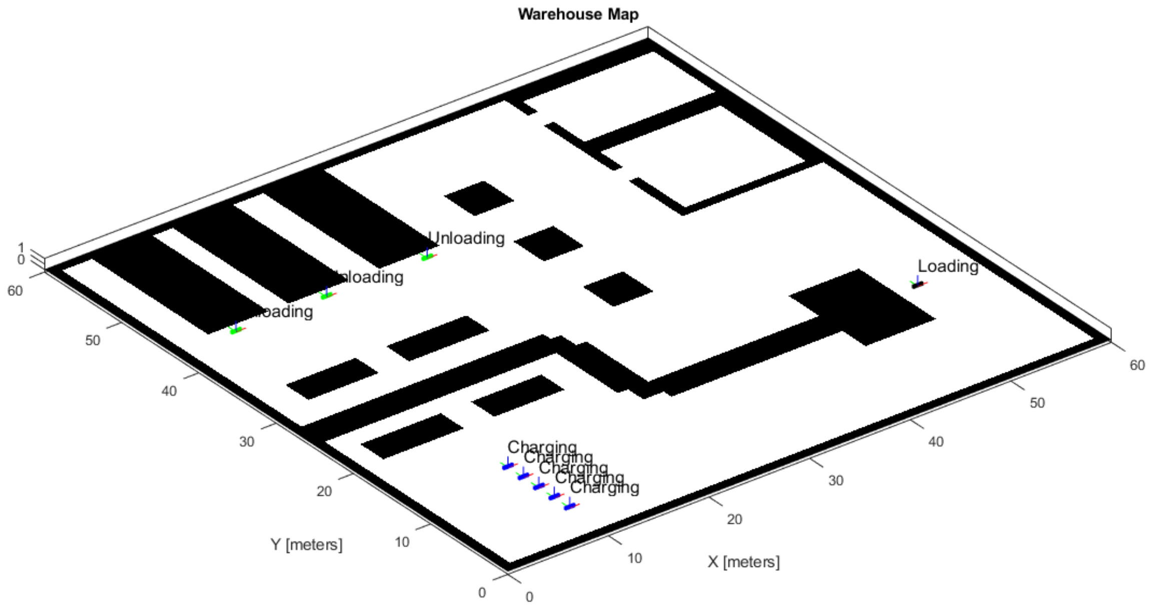
Model Setup

Begin to setup various variables in MATLAB® for the model.

Defining the Warehouse Environment

A logical type matrix, `logicalMap` represents the occupancy map of the warehouse. The warehouse contains obstacles representing walls, shelves, and other processing stations. Loading, unloading, and charging stations are also given in *xy*-coordinates.

```
load multiRobotWarehouseMap.mat logicalMap loadingStation unloadingStations chargingStations
warehouseFig = figure('Name', 'Warehouse Setting', 'Units',"normalized", 'OuterPosition',[0 0 1 
visualizeWarehouse(warehouseFig, logicalMap, chargingStations, unloadingStations, loadingStation
```



Checking occupancy at stations

Ensure that the stations are not occupied in the map.

```
map = binaryOccupancyMap(logicalMap);
if(any(checkOccupancy(map, [chargingStations; loadingStation; unloadingStations])))
    error("At least one of the station locations is occupied in the map.")
end
```

Central Scheduler

The **Central Scheduler** requires the knowledge of the packages that are to be delivered so as to send the delivery commands to the robot controllers.

Defining Packages

Packages are given as an array of index numbers of the various unloading stations that the packages are supposed to be delivered to. Because this example has three unloading stations, a valid package can take a value of 1, 2, or 3.

```
load packages.mat packages
packages
```

```
packages = 1×11

     3     2     1     2     3     1     1     1     2     3     1
```

Number of Robots

The number of robots is used to determine the sizes of the various signals in the initialization of the **Scheduler** Stateflow chart

```
numRobots = size(chargingStations, 1); % Each robot has its own charging station;
```

Collision Detection and Goal-Reached Threshold

The **Central Scheduler** and the **Robot Controller** use certain thresholds for collision detection, `collisionThresh`, and a goal-reached condition, `awayFromGoalThresh`.

Collision detection ensures that for any pair of robots within a certain distance-threshold, the robot with a lower index should be allowed to move while the other robot should stop (zero-velocity command). The still moving robot should be able to avoid local static obstacles in their path. You could achieve this with another low-level controller like the Vector Field Histogram (Navigation Toolbox) block.

The goal-reached condition occurs if the robot is within a distance threshold, `awayFromGoalThresh`, from the goal location.

```
load exampleMultiRobotParams.mat awayFromGoalThresh collisionThresh
```

Bus Objects

The `RobotDeliverCommand` and `RobotPackageStatus` bus objects are used to pass robot-package allocations between the **Central Scheduler** and the **Robot Controller**.

```
load warehouseRobotBusObjects.mat RobotDeliverCommand RobotPackageStatus
```

Simulation

Open the Simulink model.

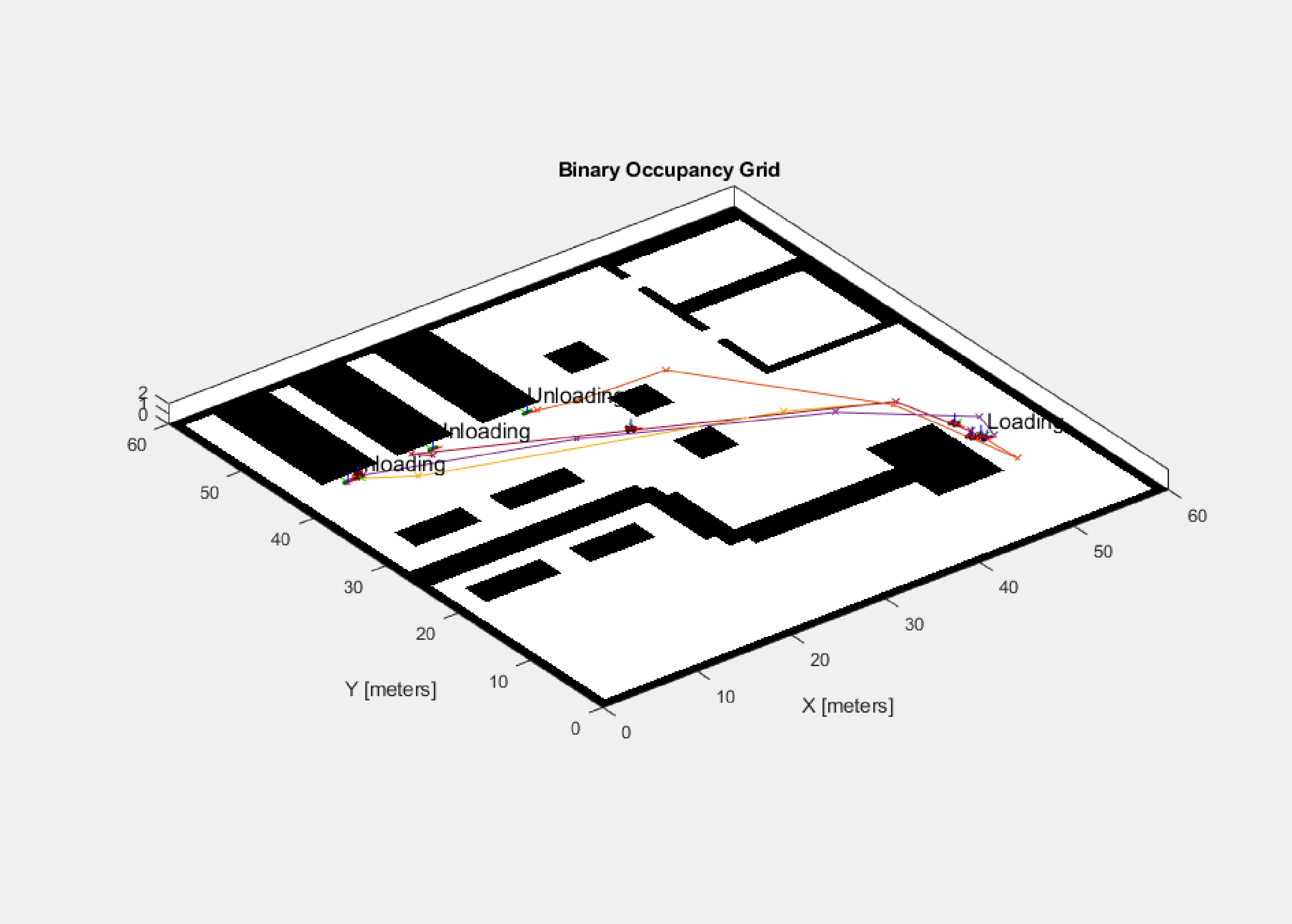
```
open_system("multiRobotExampleModel.slx")
```

Run the simulation. You should see the robots drive plan paths and deliver packages.

```
sim('multiRobotExampleModel');
```

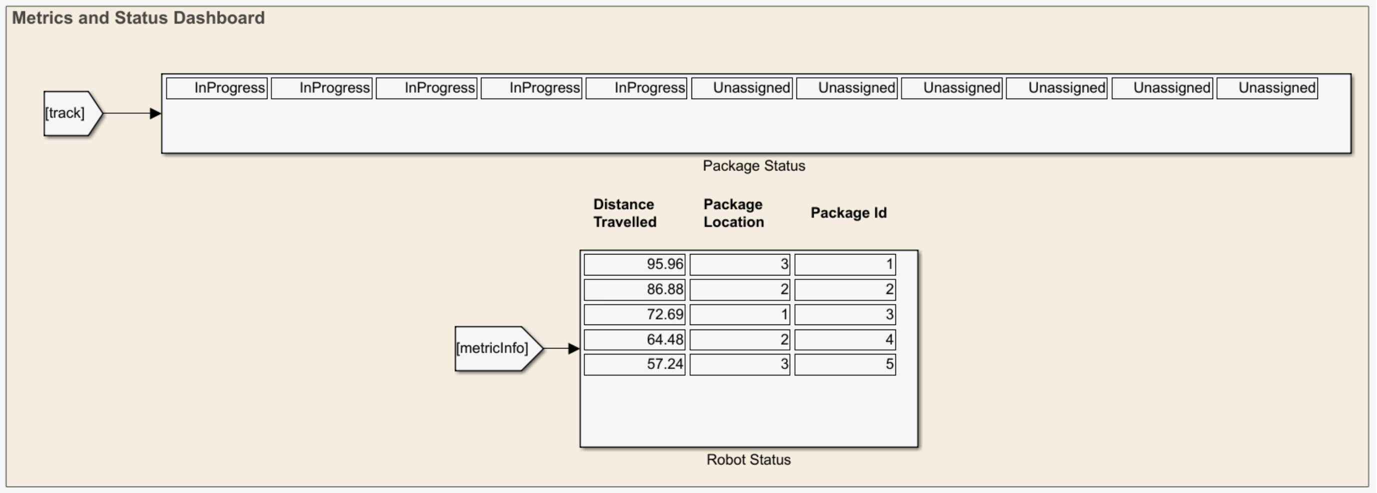
```
### Starting serial model reference simulation build
### Successfully updated the model reference simulation target for: robotController

Build Summary

Simulation targets built:

Model            Action                       Rebuild Reason
========================================================================================
robotController  Code generated and compiled  robotController_msf.mexw64 does not exist.
```

```
1 of 1 models built (0 models already up to date)
Build duration: 0h 2m 30.3s
```



Metrics and Status Dashboard

For each of the packages, the dashboard in the model shows if the package is "InProgress", "Unassigned", or "Delivered". **Robot Status** displays the distance travelled, package location, and a package ID.



InProgress	InProgress	InProgress	InProgress	InProgress	Unassigned	Unassigned	Unassigned	Unassigned	Unassigned	Unassigned

Distance Travelled	Package Location	Package Id
95.96	3	1
86.88	2	2
72.69	1	3
64.48	2	4
57.24	3	5

Extending the Model

This model is setup to handle modifying the number of robots in the warehouse based on availability. Adding more robots requires defining additional charging stations.

```
chargingStations(6, :) = [10, 15]; % Charging Station for the additional 6th robot
chargingStations(7, :) = [10, 17];  % Charging Station for the additional 7th robot
```

You can also add more unloading stations and assign packages to it.

```
unloadingStations(4, :) = [30, 50];
packages = [packages, 4, 4];
```

Additional **Differential Kinematic Model** blocks are also required to match the number of robots. The `exampleHelperReplacePlantSubsystem` adds these by updating `numRobots`.

```
numRobots = size(chargingStations, 1) % As before, each robot has its own charging station
```

```
numRobots = 7
```

```
exampleHelperReplacePlantSubsystem('multiRobotExampleModel/Robots', numRobots);
```

You can also redefine any existing locations. Modify the loading station location.

```
loadingStation = [35, 20];
```

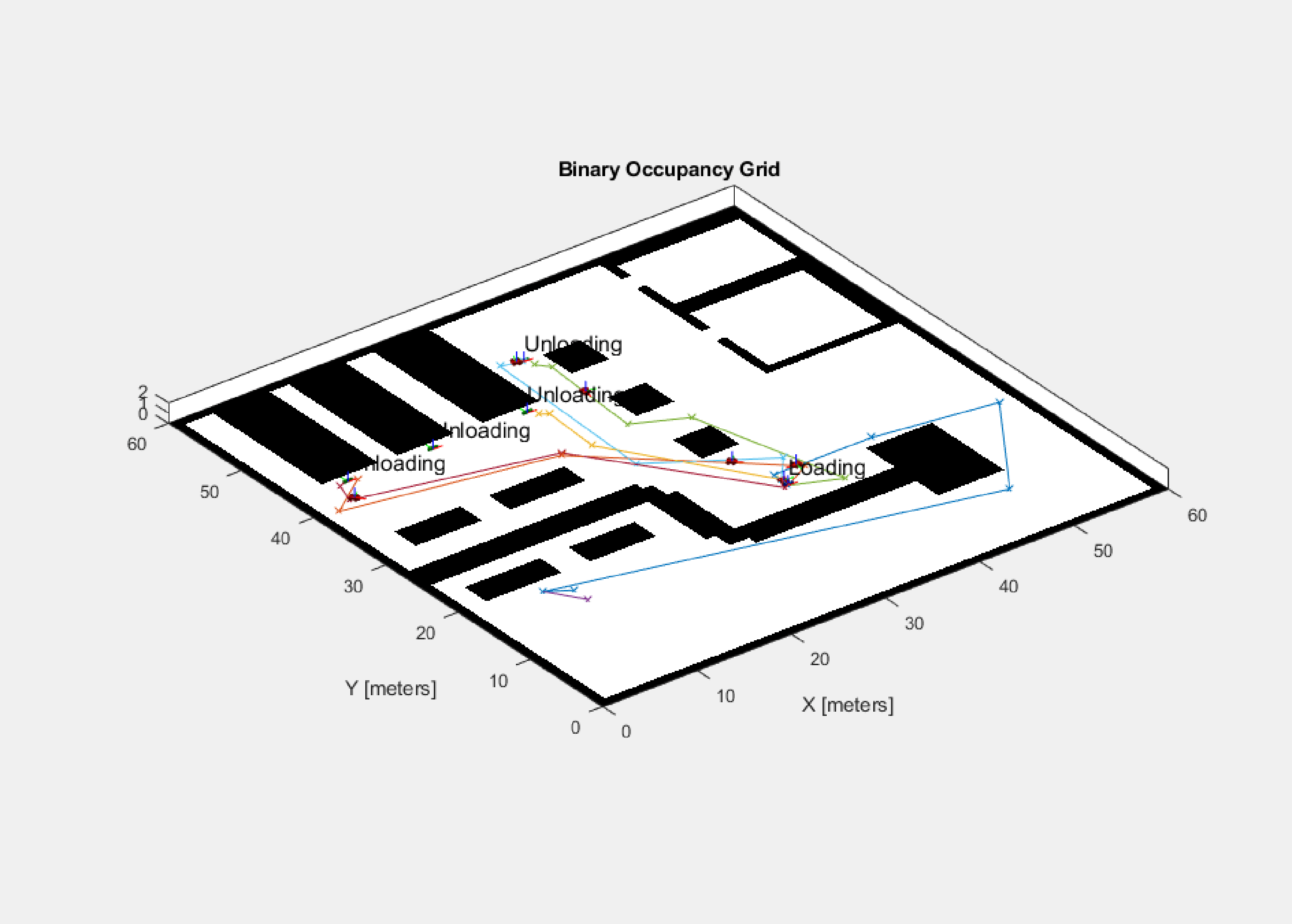
Simulation

After making the modifications, run the simulation again. You should see the updated station locations and an increased number of robots.

```
sim('multiRobotExampleModel');
```

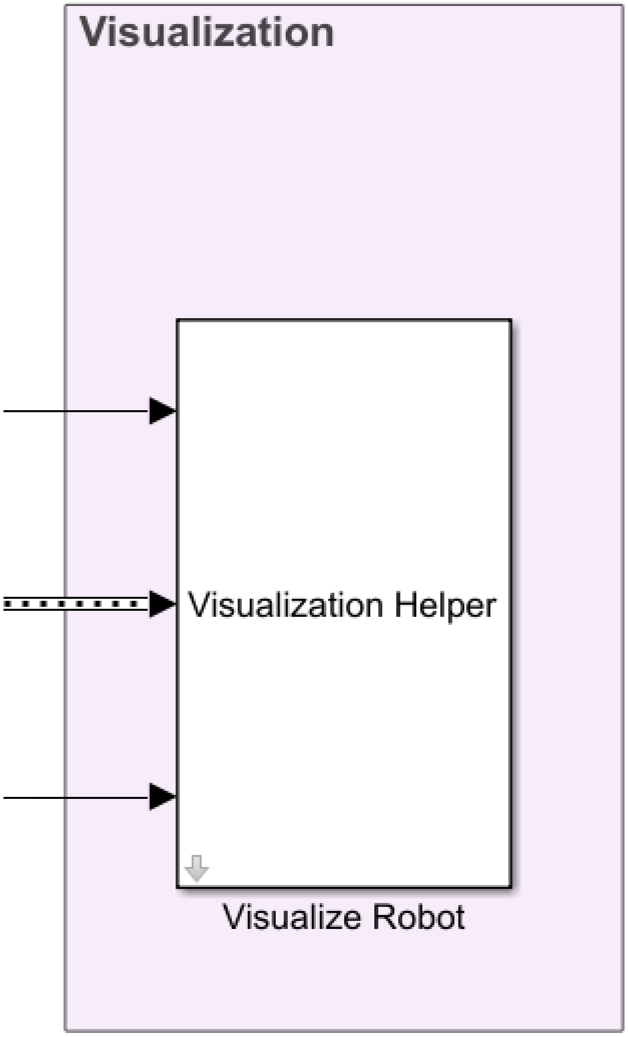
```
### Starting serial model reference simulation build
### Successfully updated the model reference simulation target for: robotController

Build Summary

Simulation targets built:
```

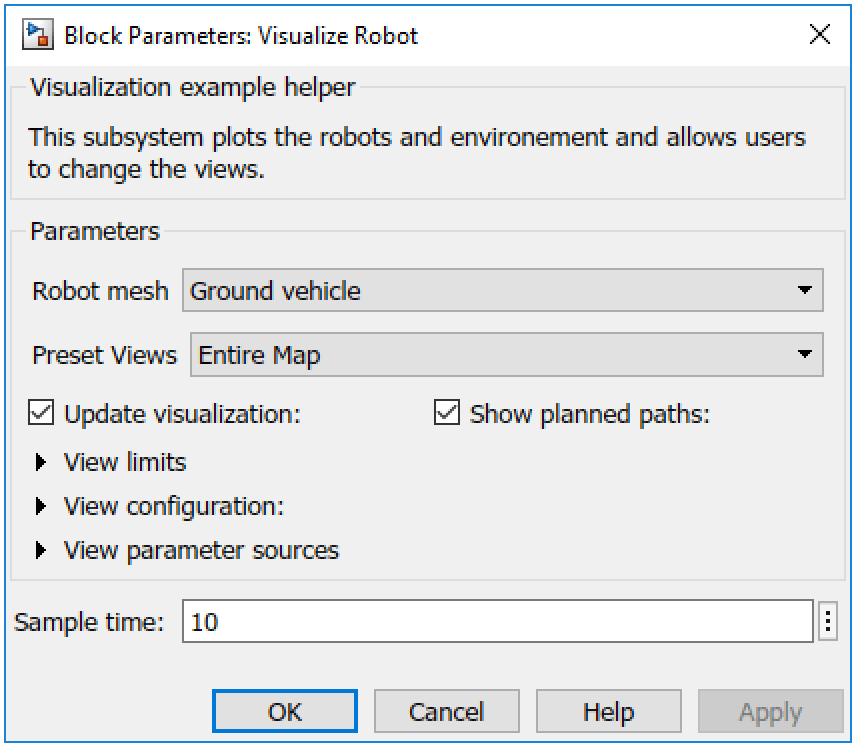
```
Model             Action                       Rebuild Reason
==============================================================================================
robotController   Code generated and compiled  Global variable unloadingStations has changed.

1 of 1 models built (0 models already up to date)
Build duration: 0h 1m 8.942s
```



Visualization

The **Visualization Helper** offers some options for changing the view of the warehouse. Open the block mask to switch between various **Preset Views** of different stations. Toggle path visualization or update robot mesh types. Adjust the **Sample time** to change the rate of the visualization, which does not affect the execution of the actual robot simulation.





Visualization example helper

This subsystem plots the robots and environement and allows users to change the views.

Parameters

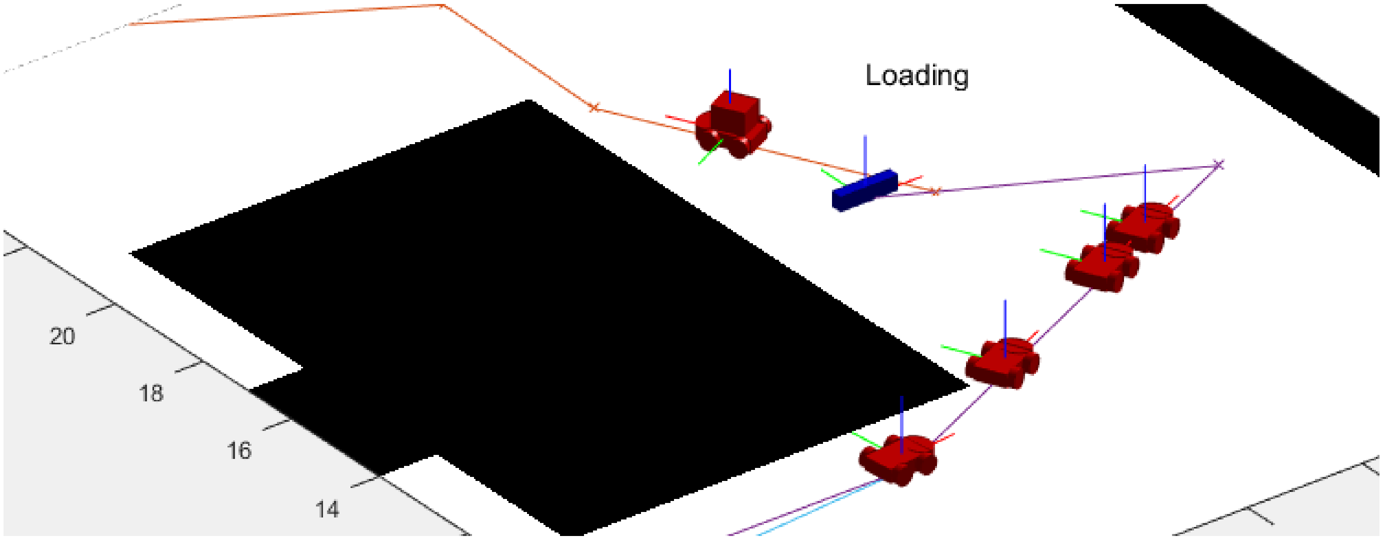
Robot mesh: Ground vehicle

Preset Views: Entire Map

☑ Update visualization: ☑ Show planned paths:

- ▸ View limits
- ▸ View configuration:
- ▸ View parameter sources

Sample time: 10


Loading
20
18
16
14

Simulate a Mobile Robot in a Warehouse Using Gazebo

This example shows how to simulate a warehouse robot in Gazebo. Gazebo enabled you to simulate a mobile robot that uses a range sensor, while executing certain tasks in a simulated environment. This example details how to use a simulator to apply the “Execute Tasks for a Warehouse Robot” on page 1-31 example, where a robot delivers packages in a warehouse scenario. The robot makes use of the simulated range sensor in Gazebo to detect possible collisions with a dynamic environment and avoid them.

Prerequisites

- Download a Virtual Machine with ROS and Gazebo to set up a simulated robot.
- Review the “Execute Tasks for a Warehouse Robot” on page 1-31 example for the workflow of path planning and navigating in a warehouse scenario.
- Review the “Control a Differential Drive Robot in Gazebo with Simulink” on page 1-70 example for basic steps of collecting sensor data and controlling a robot in Gazebo.

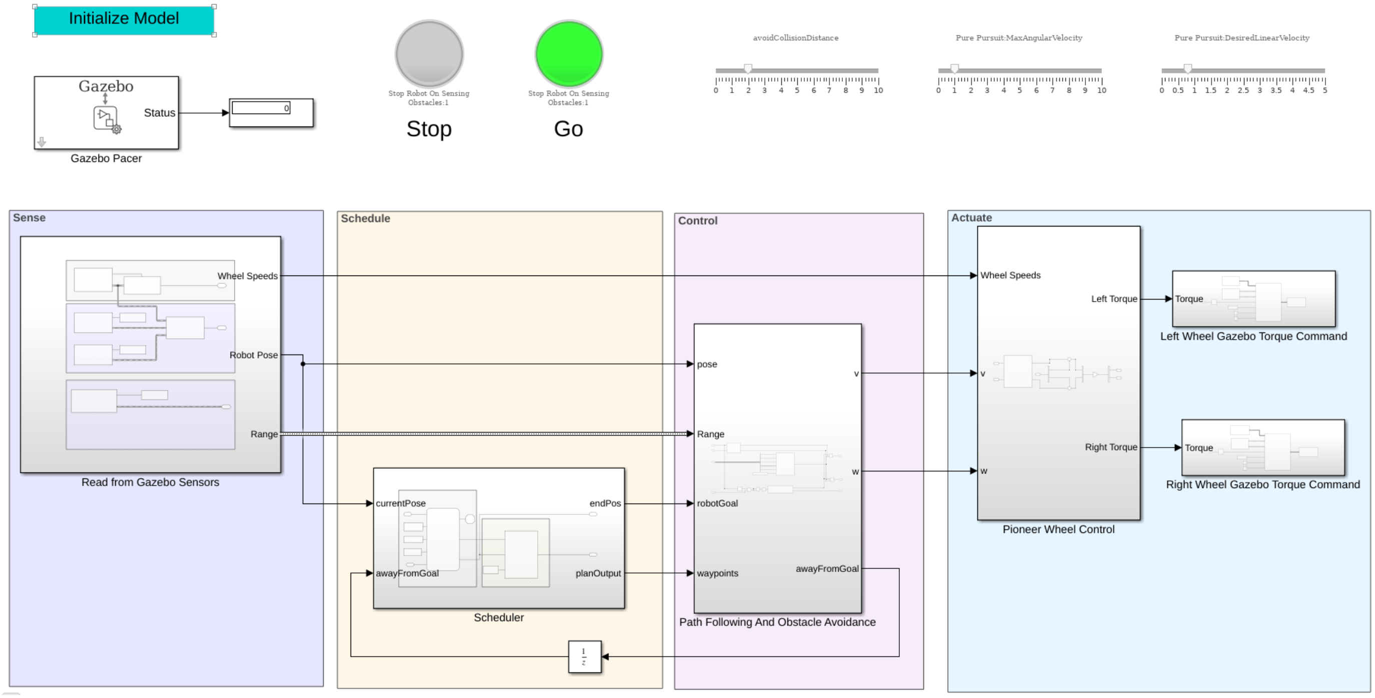
Model Overview

Open the model.

```
open_system('simulateWarehouseRobotInGazebo.slx')
```

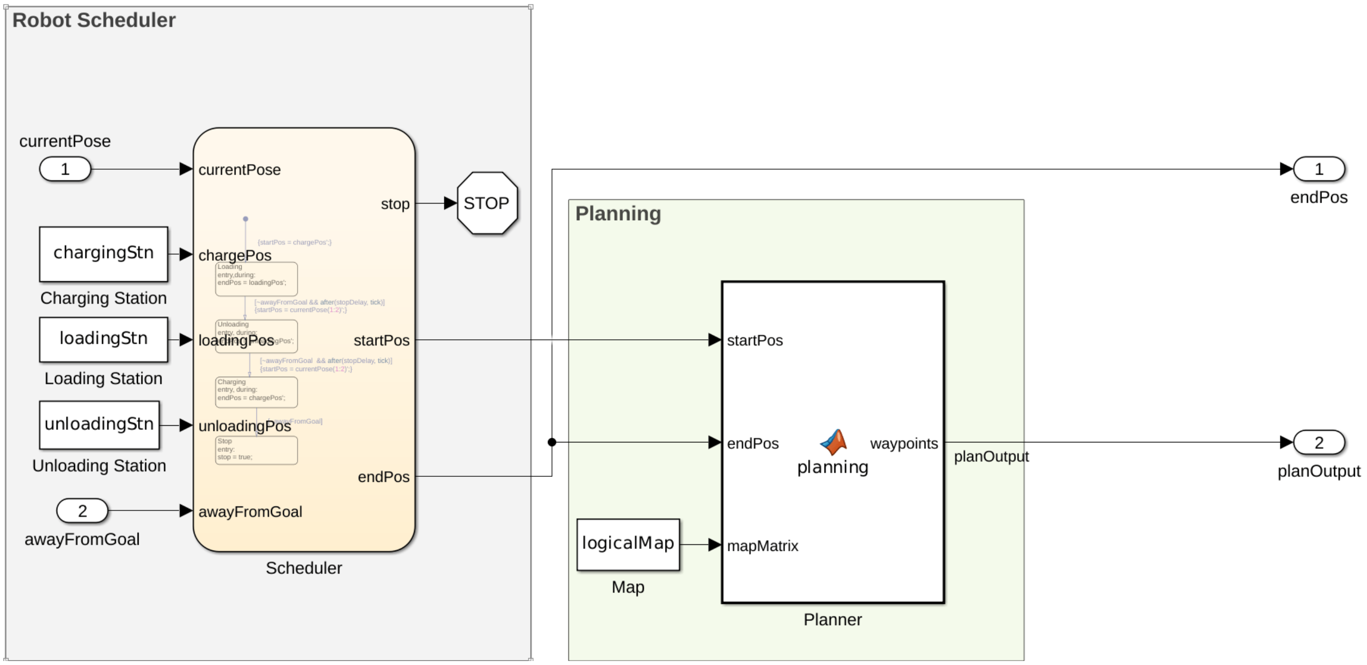
The model can be divided into the following elements:

- **Sense:** Read data from sensors in Gazebo.
- **Schedule:** Assign packages and plan paths for robots to deliver the packages.
- **Control:** Generate commands to follow the predefined path and avoid obstacles.
- **Actuate:** Send commands to Gazebo to actuate the robot in the environment.



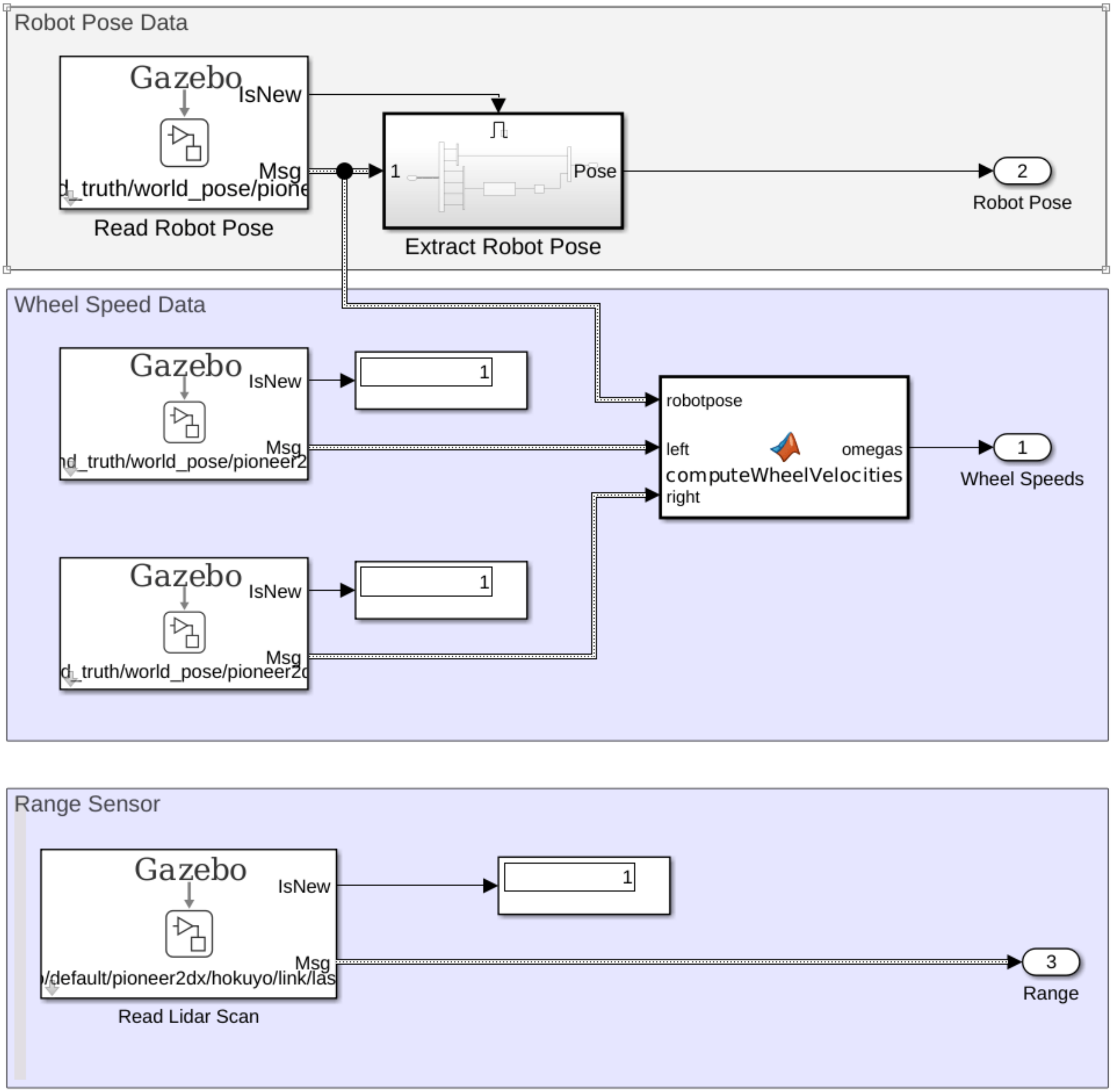
Schedule

The robot performs the task of going between the charging station, the loading station, and the unloading station as guided by the **Scheduler**.



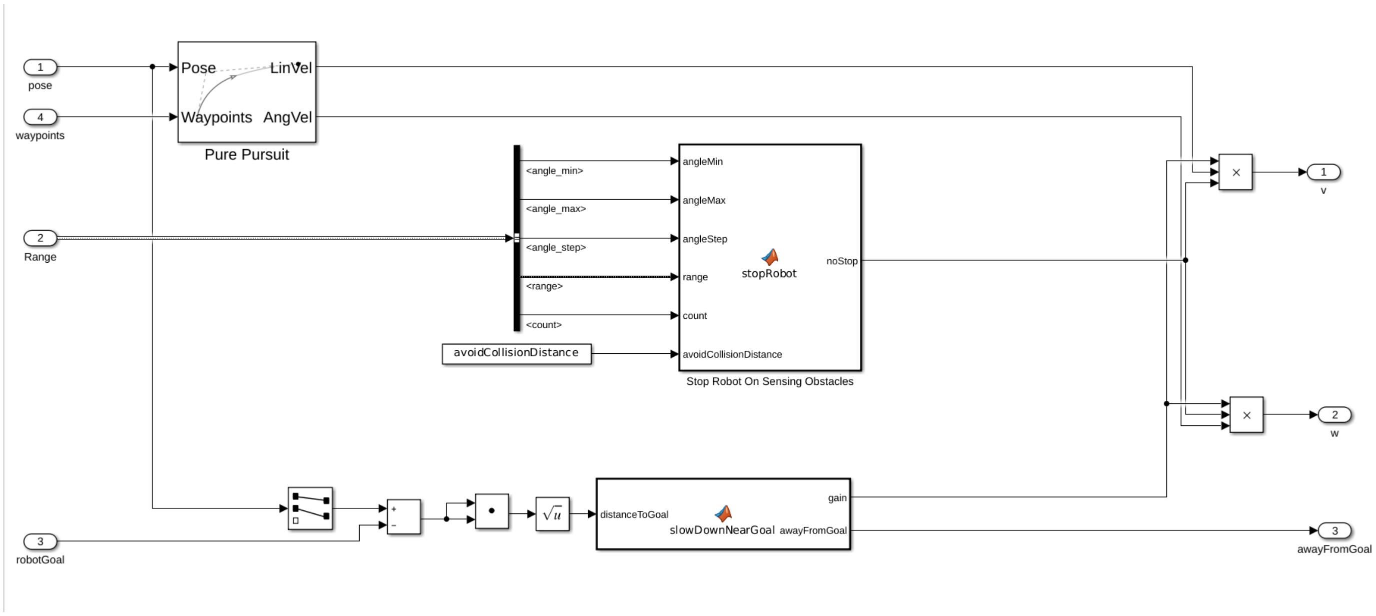
Sense

The current robot pose, the wheel speeds, and the range sensor readings are read from the simulated environment in Gazebo. The figure below is the expanded view of the **Read From Gazebo Sensors** subsystem.



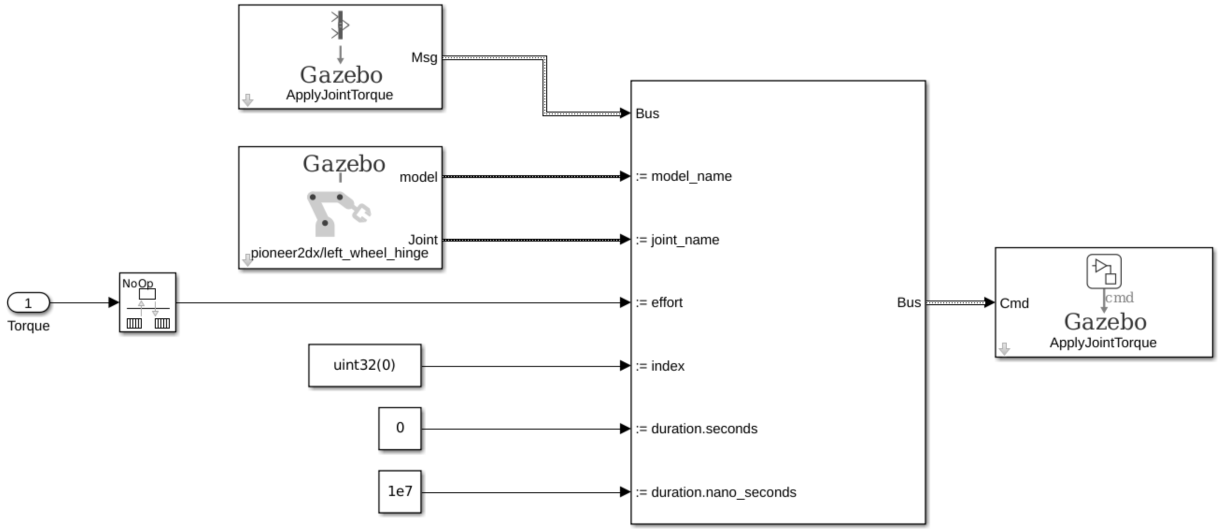
Control

The controller generates control commands for following the waypoints using the **Pure Pursuit** block. If the range sensor on the robot detects an obstacle within the `avoidCollisionDistance` threshold, the robot stops. Also, the robot stops when gets near enough to the goal.



Actuate

Based on the generated control commands, the **Pioneer Wheel Control** subsystem generates a torque value for each wheel.The torque is applied as an `ApplyJointTorque` command.



Setup

Warehouse Facility

Load the example map file, `map`, which is a matrix of logical values indicating occupied space in the warehouse. Invert this matrix to indicate free space, and make a `binaryOccupancyMap` object. Specify a resolution of 100 cells per meter.

The map is based off of the `warehouseExtensions.world` file, which was made using the Building Editor on the same scaling factor as mentioned below. A `.png` file for the map can be made using the `collision_map_creator_plugin` plugin to generate the map matrix. The details on how to install the plugin can be found at Collision Map Creator Plugin.

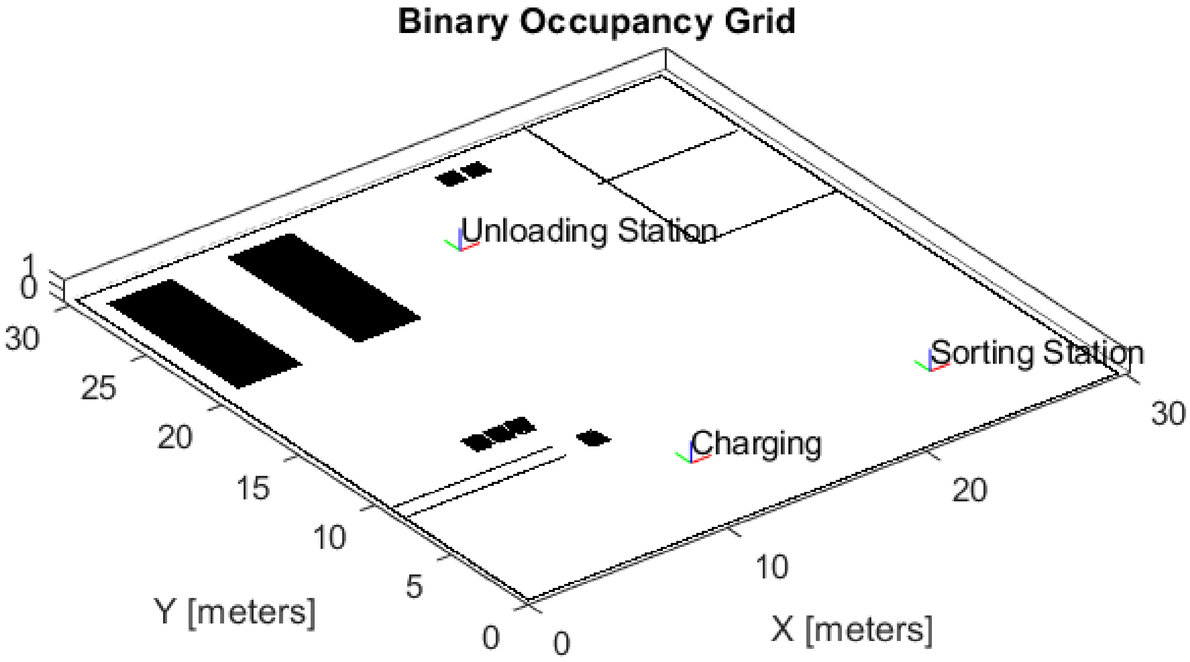
```
mapScalingFactor = 100;
load gazeboWarehouseMap.mat map
logicalMap = ~map;
map = binaryOccupancyMap(logicalMap,mapScalingFactor);
show(map)
```

Assign the *xy*-locations of the charging station, sorting station, and the unloading location near shelves in the warehouse. The values chosen are based on the simlated world in Gazebo.

```
chargingStn = [12,5];
loadingStn = [24,5];
unloadingStn = [15,24];
```

Show the various locations on the map.

```
hold on;

text(chargingStn(1), chargingStn(2), 1, 'Charging');
plotTransforms([chargingStn, 0], [1 0 0 0])

text(loadingStn(1), loadingStn(2), 1, 'Sorting Station');
plotTransforms([loadingStn, 0], [1 0 0 0])

text(unloadingStn(1), unloadingStn(2), 1, 'Unloading Station');
plotTransforms([unloadingStn, 0], [1 0 0 0])

hold off;
```



Range Sensor

The **Read Lidar Scan** block in the Sensing on page 1-0 section is used to read the range values from the simulated range sensor. The `warehouseExtensions.world` file contains the details of the various models and actors (warehouse workers) in the scene. Because `<actor>` tags are static links with only visual meshes, the sensor type of the range sensor is `gpu_ray`.

Additionally, the range sensor uses 640 range, but the default is 128. This requires modification of the buses used in the in the **Read Lidar Scan** block. Load the `exampleHelperWarehouseRobotWithGazeboBuses.mat` file to get a modified bus with `Gazebo_SL_Bus_gazebo_msgs_LaserScan.range` set to 640. The modified buses were saved to a `.mat` file using the Bus Editor.

```
load exampleHelperWarehouseRobotWithGazeboBuses.mat
```

Collision Avoidance

The actors in the world are walking a predefined trajectory. The robot makes use of a range sensor to check for obstacles within a range of 2.0 m (`avoidCollisionDistance)` with range angles from `[-pi/10, pi/10]` Upon a non-zero reading within that range and view, the robot stops and only resumes after the range is clear. The "Stop Robot On Sensing Obstacles" function block incorporates this logic.

While running the simulation, the **Stop** lamp turns green when the robot senses that it is good to proceed. If it has stopped the lamp turns red.

```
avoidCollisionDistance = 2;
```

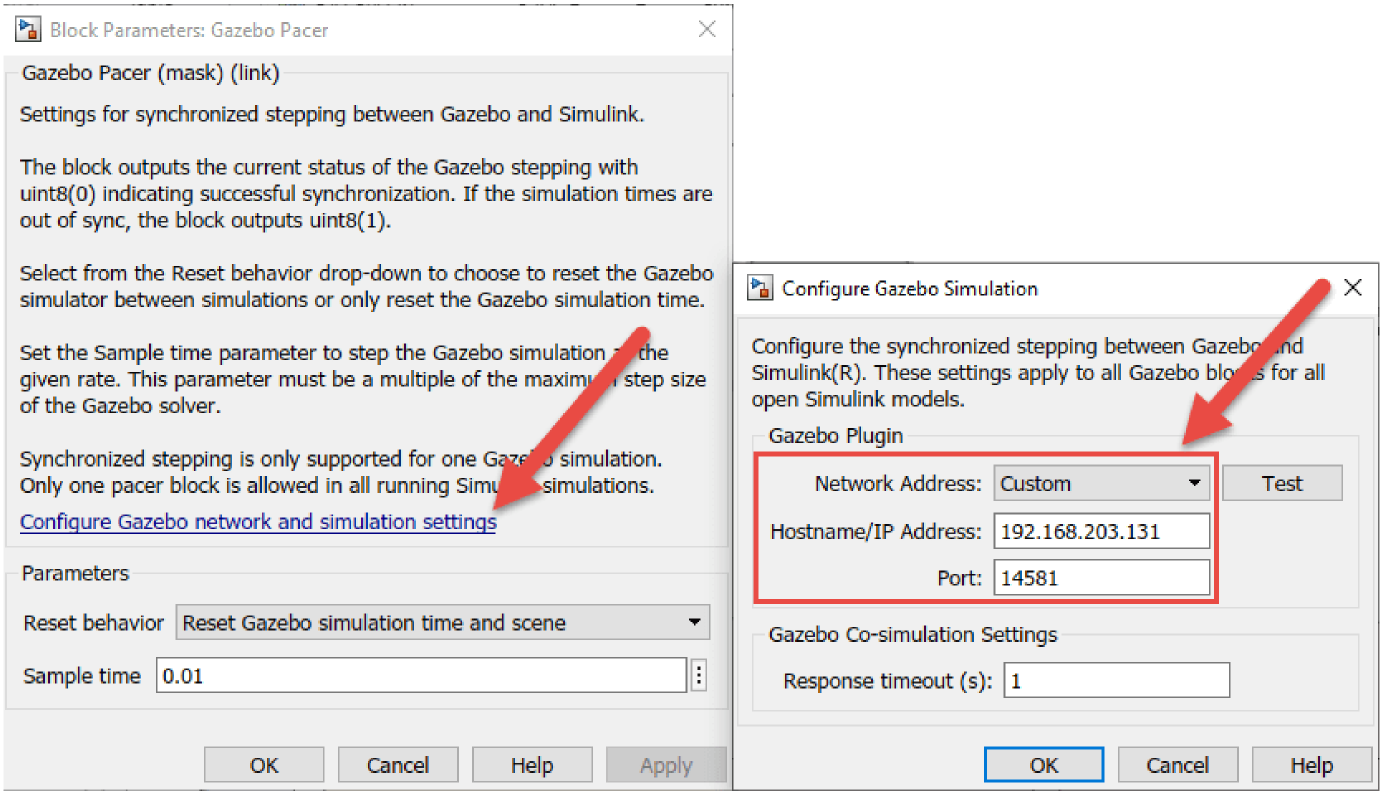
Simulate

To simulate the scenario, set up the connection to Gazebo.

First, run the Gazebo Simulator. In the virtual machine, click the **Gazebo Warehouse Robot** icon. If the Gazebo simulator fails to open, you may need to reinstall the plugin. See **Install Gazebo Plugin Manually** in “Perform Co-Simulation between Simulink and Gazebo” on page 1-431.

In Simulink, open the **Gazebo Pacer** block and click **Configure Gazebo network and simulation settings**. Specify the **Network Address** as **Custom**, the **Hostname/IP Address** for your Gazebo simulation, and a **Port** of 14581, which is the default port for Gazebo. The desktop of the VM displays the IP address.

For more information about connecting to Gazebo to enable co-simulation, see “Perform Co-Simulation between Simulink and Gazebo” on page 1-431.



Block Parameters: Gazebo Pacer

Gazebo Pacer (mask) (link)

Settings for synchronized stepping between Gazebo and Simulink.

The block outputs the current status of the Gazebo stepping with uint8(0) indicating successful synchronization. If the simulation times are out of sync, the block outputs uint8(1).

Select from the Reset behavior drop-down to choose to reset the Gazebo simulator between simulations or only reset the Gazebo simulation time.

Set the Sample time parameter to step the Gazebo simulation at the given rate. This parameter must be a multiple of the maximum step size of the Gazebo solver.

Synchronized stepping is only supported for one Gazebo simulation. Only one pacer block is allowed in all running Simulink simulations.

Configure Gazebo network and simulation settings

Parameters

Reset behavior: Reset Gazebo simulation time and scene

Sample time: 0.01

Configure Gazebo Simulation

Configure the synchronized stepping between Gazebo and Simulink(R). These settings apply to all Gazebo blocks for all open Simulink models.

Gazebo Plugin

Network Address: Custom

Hostname/IP Address: 192.168.203.131

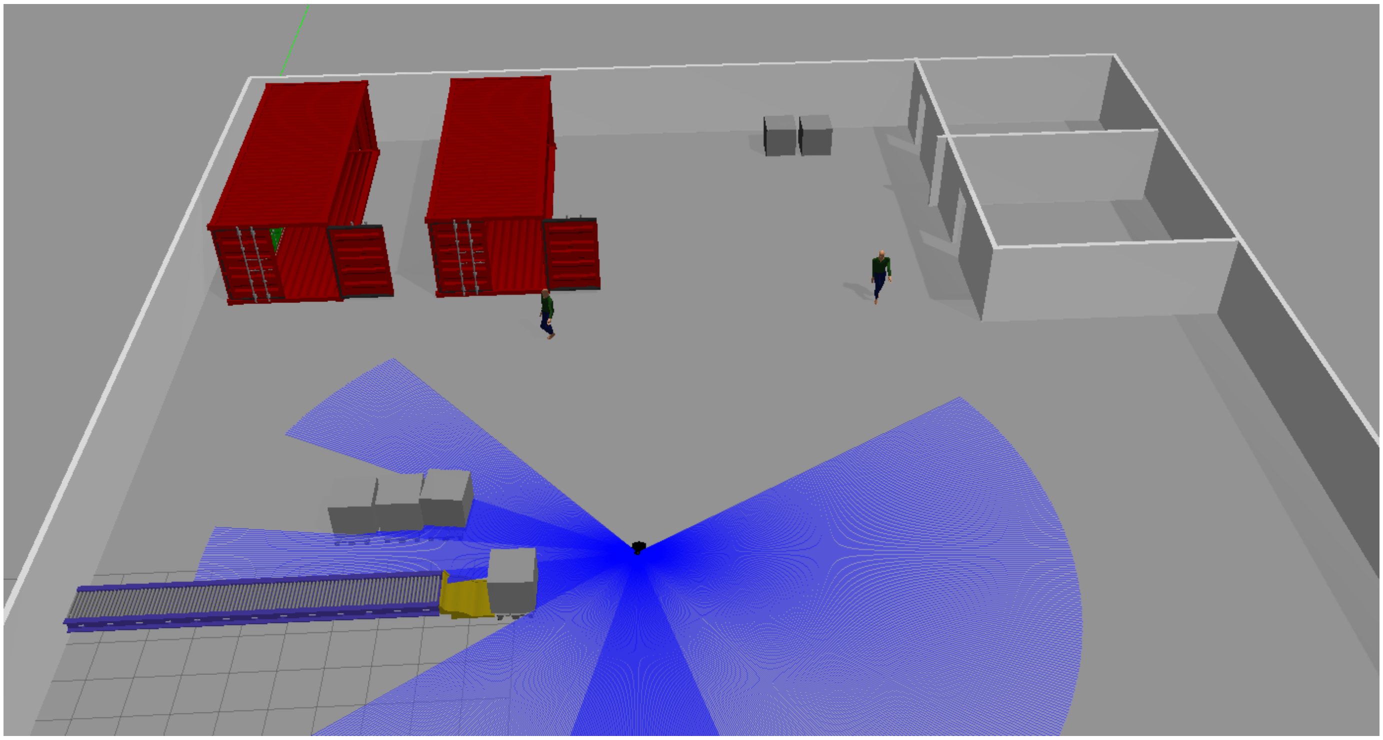
Port: 14581

Gazebo Co-simulation Settings

Response timeout (s): 1

Click the **Initialize Model** button at the top of the model to intialize all the variables declared above.

Run the simulation. The robot drives around the environment and stops whenever a worker gets within the defined threshold.



Track a Car-Like Robot Using Particle Filter

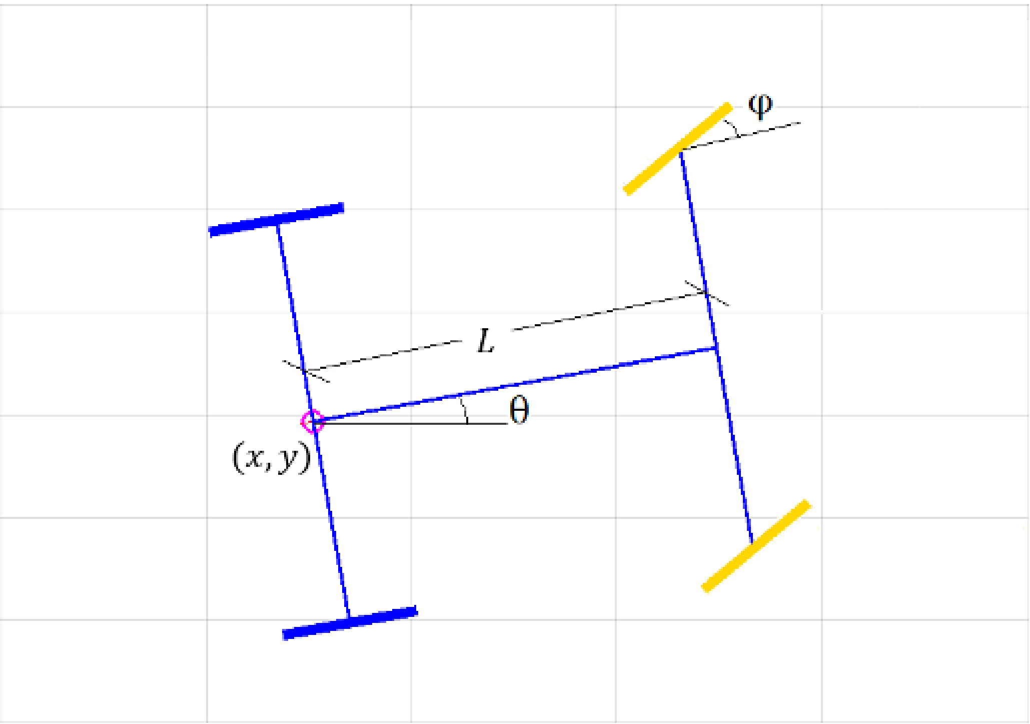
Particle filter is a sampling-based recursive Bayesian estimation algorithm, which is implemented in the `stateEstimatorPF` object. In this example, a remote-controlled car-like robot is being tracked in the outdoor environment. The robot pose measurement is provided by an on-board GPS, which is noisy. There are known motion commands sent to the robot, but the robot will not execute the exact commanded motion due to mechanical slack or model inaccuracy. This example will show how to use `stateEstimatorPF` to reduce the effects of noise in the measurement data and get a more accurate estimation of the pose of the robot. The kinematic model of a car-like robot is described by the following non-linear system. The particle filter is ideally suited for estimating the state of such kind of systems, as it can deal with the inherent non-linearities.

$$\dot{x} = v\cos(\theta)$$

$$\dot{y} = v\sin(\theta)$$

$$\dot{\theta} = \frac{v}{L}\tan\phi$$

$$\dot{\phi} = \omega$$



Scenario: The car-like robot drives and changes its velocity and steering angle continuously. The pose of the robot is measured by some noisy external system, e.g. a GPS or a Vicon system. Along the path it will drive through a roofed area where no measurement can be made.

Input:

- The noisy measurement on robot's partial pose (x, y, θ). **Note** this is not a full state measurement. No measurement is available on the front wheel orientation (ϕ) as well as all the velocities ($\dot{x}$, $\dot{y}$, $\dot{\theta}$, $\dot{\phi}$).

- The linear and angular velocity command sent to the robot (v_c, ω_c). **Note** there will be some difference between the commanded motion and the actual motion of the robot.

Goal: Estimate the partial pose (x, y, θ) of the car-like robot. **Note** again that the wheel orientation (ϕ) is not included in the estimation. **From the observer's perspective**, the full state of the car is only [x, y, θ, $\dot{x}$, $\dot{y}$, $\dot{\theta}$].

Approach: Use `stateEstimatorPF` to process the two noisy inputs (neither of the inputs is reliable by itself) and make best estimation of current (partial) pose.

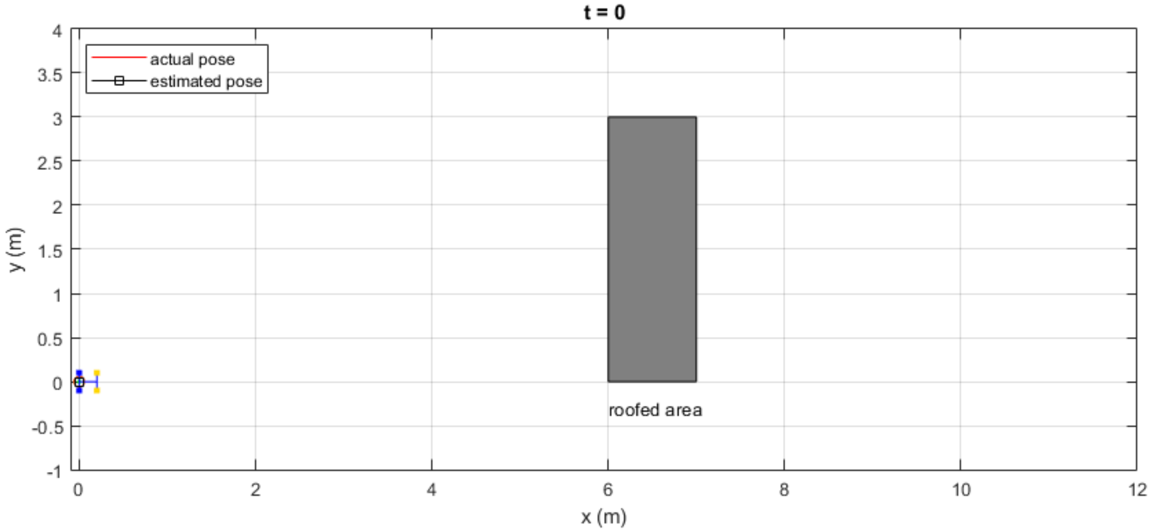
- At the **predict** stage, we update the states of the particles with a simplified, unicycle-like robot model, as shown below. Note that the system model used for state estimation is not an exact representation of the actual system. This is acceptable, as long as the model difference is well-captured in the system noise (as represented by the particle swarm). For more details, see `predict`.

$$\dot{x} = v\cos(\theta)$$
$$\dot{y} = v\sin(\theta)$$
$$\dot{\theta} = \omega$$

- At the **correct** stage, the importance weight (likelihood) of a particle is determined by its error norm from current measurement ($\sqrt{(\Delta x)^2 + (\Delta y)^2 + (\Delta\theta)^2}$), as we only have measurement on these three components. For more details, see `correct`.

Initialize a Car-like Robot

```
rng('default'); % for repeatable result
dt = 0.05; % time step
initialPose = [0  0  0  0]';
carbot = ExampleHelperCarBot(initialPose, dt);
```



Set up the Particle Filter

This section configures the particle filter using 5000 particles. Initially all particles are randomly picked from a normal distribution with mean at initial state and unit covariance. Each particle contains 6 state variables (x, y, θ, $\dot{x}$, $\dot{y}$, $\dot{\theta}$). Note that the third variable is marked as Circular since it is the car orientation. It is also very important to specify two callback functions `StateTransitionFcn` and `MeasurementLikelihoodFcn`. These two functions directly determine the performance of the particle filter. The details of these two functions can be found the in the last two sections of this example.

```
pf = stateEstimatorPF;

initialize(pf, 5000, [initialPose(1:3)', 0, 0, 0], eye(6), 'CircularVariables',[0 0 1 0 0 0]);
pf.StateEstimationMethod = 'mean';
pf.ResamplingMethod = 'systematic';

% StateTransitionFcn defines how particles evolve without measurement
pf.StateTransitionFcn = @exampleHelperCarBotStateTransition;

% MeasurementLikelihoodFcn defines how measurement affect the our estimation
pf.MeasurementLikelihoodFcn = @exampleHelperCarBotMeasurementLikelihood;

% Last best estimation for x, y and theta
lastBestGuess = [0 0 0];
```

Main Loop

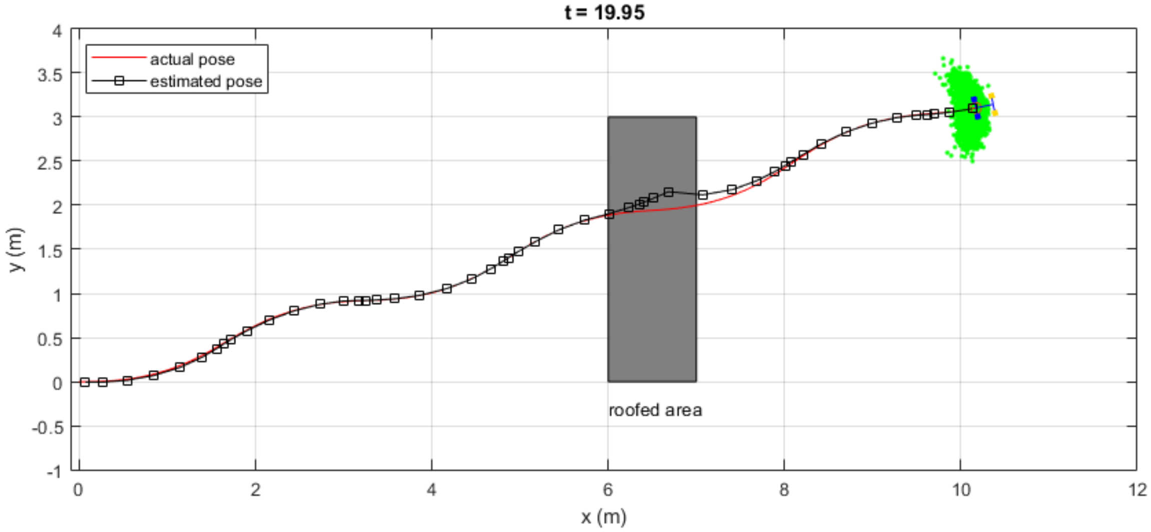
Note in this example, the commanded linear and angular velocities to the robot are arbitrarily-picked time-dependent functions. Also, note the fixed-rate timing of the loop is realized through `rateControl`.

Run loop at 20 Hz for 20 seconds using fixed-rate support.

```
r = rateControl(1/dt);
```

Reset the fixed-rate object to restart the timer. Reset the timer right before running the time-dependent code.

```
reset(r);

simulationTime = 0;

while simulationTime < 20 % if time is not up

    % Generate motion command that is to be sent to the robot
    % NOTE there will be some discrepancy between the commanded motion and the
    % motion actually executed by the robot.
    uCmd(1) = 0.7*abs(sin(simulationTime)) + 0.1;  % linear velocity
    uCmd(2) = 0.08*cos(simulationTime);            % angular velocity

    drive(carbot, uCmd);

    % Predict the carbot pose based on the motion model
    [statePred, covPred] = predict(pf, dt, uCmd);

    % Get GPS reading
    measurement = exampleHelperCarBotGetGPSReading(carbot);
    % If measurement is available, then call correct, otherwise just use
    % predicted result
    if ~isempty(measurement)
        [stateCorrected, covCorrected] = correct(pf, measurement');
    else
        stateCorrected = statePred;
        covCorrected = covPred;
    end

    lastBestGuess = stateCorrected(1:3);

    % Update plot
    if ~isempty(get(groot,'CurrentFigure')) % if figure is not prematurely killed
        updatePlot(carbot, pf, lastBestGuess, simulationTime);
    else
        break
    end

    waitfor(r);

    % Update simulation time
    simulationTime = simulationTime + dt;
end
```



Details of the Result Figures

The three figures show the tracking performance of the particle filter.

- In the first figure, the particle filter is tracking the car well as it drives away from the initial pose.
- In the second figure, the robot drives into the roofed area, where no measurement can be made, and the particles only evolve based on prediction model (marked with orange color). You can see

the particles gradually form a horseshoe-like front, and the estimated pose gradually deviates from the actual one.

- In the third figure, the robot has driven out of the roofed area. With new measurements, the estimated pose gradually converges back to the actual pose.

State Transition Function

The sampling-based state transition function evolves the particles based on a prescribed motion model so that the particles will form a representation of the proposal distribution. Below is an example of a state transition function based on the velocity motion model of a unicycle-like robot. For more details about this motion model, please see Chapter 5 in **[1]**. Decrease `sd1`, `sd2` and `sd3` to see how the tracking performance deteriorates. Here `sd1` represents the uncertainty in the linear velocity, `sd2` represents the uncertainty in the angular velocity. `sd3` is an additional perturbation on the orientation.

```
function predictParticles = exampleHelperCarBotStateTransition(pf, prevParticles, dT, u)

    thetas = prevParticles(:,3);

    w = u(2);
    v = u(1);

    l = length(prevParticles);

    % Generate velocity samples
    sd1 = 0.3;
    sd2 = 1.5;
    sd3 = 0.02;
    vh = v + (sd1)^2*randn(l,1);
    wh = w + (sd2)^2*randn(l,1);
    gamma = (sd3)^2*randn(l,1);

    % Add a small number to prevent div/0 error
    wh(abs(wh)<1e-19) = 1e-19;

    % Convert velocity samples to pose samples
    predictParticles(:,1) = prevParticles(:,1) - (vh./wh).*sin(thetas) + (vh./wh).*sin(thetas
    predictParticles(:,2) = prevParticles(:,2) + (vh./wh).*cos(thetas) - (vh./wh).*cos(thetas
    predictParticles(:,3) = prevParticles(:,3) + wh*dT + gamma*dT;
    predictParticles(:,4) = (- (vh./wh).*sin(thetas) + (vh./wh).*sin(thetas + wh*dT))/dT;
    predictParticles(:,5) = ( (vh./wh).*cos(thetas) - (vh./wh).*cos(thetas + wh*dT))/dT;
    predictParticles(:,6) = wh + gamma;

end
```

Measurement Likelihood Function

The measurement likelihood function computes the likelihood for each predicted particle based on the error norm between particle and the measurement. The importance weight for each particle will be assigned based on the computed likelihood. In this particular example, `predictParticles` is a N x 6 matrix (N is the number of particles), and `measurement` is a 1 x 3 vector.

```
function  likelihood = exampleHelperCarBotMeasurementLikelihood(pf, predictParticles, measure

    % The measurement contains all state variables
    predictMeasurement = predictParticles;

    % Calculate observed error between predicted and actual measurement
    % NOTE in this example, we don't have full state observation, but only
```

```
        % the measurement of current pose, therefore the measurementErrorNorm
        % is only based on the pose error.
        measurementError = bsxfun(@minus, predictMeasurement(:,1:3), measurement);
        measurementErrorNorm = sqrt(sum(measurementError.^2, 2));

        % Normal-distributed noise of measurement
        % Assuming measurements on all three pose components have the same error distribution
        measurementNoise = eye(3);

        % Convert error norms into likelihood measure.
        % Evaluate the PDF of the multivariate normal distribution
        likelihood = 1/sqrt((2*pi).^3 * det(measurementNoise)) * exp(-0.5 * measurementErrorNorm)

    end
```

Reference

[1] S. Thrun, W. Burgard, D. Fox, Probabilistic Robotics, MIT Press, 2006

Simulate Different Kinematic Models for Mobile Robots

This example shows how to model different robot kinematics models in an environment and compare them.

Define Mobile Robots with Kinematic Constraints

There are a number of ways to model the kinematics of mobile robots. All dictate how the wheel velocities are related to the robot state: `[x y theta]`, as *xy*-coordinates and a robot heading, `theta`, in radians.

Unicycle Kinematic Model

The simplest way to represent mobile robot vehicle kinematics is with a unicycle model, which has a wheel speed set by a rotation about a central axle, and can pivot about its z-axis. Both the differential-drive and bicycle kinematic models reduce down to unicycle kinematics when inputs are provided as vehicle speed and vehicle heading rate and other constraints are not considered.

```
unicycle = unicycleKinematics("VehicleInputs","VehicleSpeedHeadingRate");
```

Differential-Drive Kinematic Model

The differential drive model uses a rear driving axle to control both vehicle speed and head rate. The wheels on the driving axle can spin in both directions. Since most mobile robots have some interface to the low-level wheel commands, this model will again use vehicle speed and heading rate as input to simplify the vehicle control.

```
diffDrive = differentialDriveKinematics("VehicleInputs","VehicleSpeedHeadingRate");
```

To differentiate the behavior from the unicycle model, add a wheel speed velocity constraint to the differential-drive kinematic model

```
diffDrive.WheelSpeedRange = [-10 10]*2*pi;
```

Bicycle Kinematic Model

The bicycle model treats the robot as a car-like model with two axles: a rear driving axle, and a front axle that turns about the z-axis. The bicycle model works under the assumption that wheels on each axle can be modeled as a single, centered wheel, and that the front wheel heading can be directly set, like a bicycle.

```
bicycle = bicycleKinematics("VehicleInputs","VehicleSpeedHeadingRate","MaxSteeringAngle",pi/8);
```

Other Models

The Ackermann kinematic model is a modified car-like model that assumes Ackermann steering. In most car-like vehicles, the front wheels do not turn about the same axis, but instead turn on slightly different axes to ensure that they ride on concentric circles about the center of the vehicle's turn. This difference in turning angle is called Ackermann steering, and is typically enforced by a mechanism in actual vehicles. From a vehicle and wheel kinematics standpoint, it can be enforced by treating the steering angle as a rate input.

```
carLike = ackermannKinematics;
```

Set up Simulation Parameters

These mobile robots will follow a set of waypoints that is designed to show some differences caused by differing kinematics.

```
waypoints = [0 0; 0 10; 10 10; 5 10; 11 9; 4 -5];
% Define the total time and the sample rate
sampleTime = 0.05;               % Sample time [s]
tVec = 0:sampleTime:20;          % Time array

initPose = [waypoints(1,:)'; 0]; % Initial pose (x y theta)
```

Create a Vehicle Controller

The vehicles follow a set of waypoints using a Pure Pursuit controller. Given a set of waypoints, the robot current state, and some other parameters, the controller outputs vehicle speed and heading rate.

```
% Define a controller. Each robot requires its own controller
controller1 = controllerPurePursuit("Waypoints",waypoints,"DesiredLinearVelocity",3,"MaxAngularVe
controller2 = controllerPurePursuit("Waypoints",waypoints,"DesiredLinearVelocity",3,"MaxAngularVe
controller3 = controllerPurePursuit("Waypoints",waypoints,"DesiredLinearVelocity",3,"MaxAngularVe
```

Simulate the Models Using an ODE Solver

The models are simulated using the `derivative` function to update the state. This example uses an ordinary differential equation (ODE) solver to generate a solution. Another way would be to update the state using a loop, as shown in “Path Following for a Differential Drive Robot” on page 1-10.

Since the ODE solver requires all outputs to be provided as a single output, the pure pursuit controller must be wrapped in a function that outputs the linear velocity and heading angular velocity as a single output. An example helper, `exampleHelperMobileRobotController`, is used for that purpose. The example helper also ensures that the robot stops when it is within a specified radius of the goal.

```
goalPoints = waypoints(end,:)';
goalRadius = 1;
```

`ode45` is called once for each type of model. The derivative function computes the state outputs with initial state set by `initPose`. Each derivative accepts the corresponding kinematic model object, the current robot pose, and the output of the controller at that pose.

```
% Compute trajectories for each kinematic model under motion control
[tUnicycle,unicyclePose] = ode45(@(t,y)derivative(unicycle,y,exampleHelperMobileRobotController(c
[tBicycle,bicyclePose] = ode45(@(t,y)derivative(bicycle,y,exampleHelperMobileRobotController(cont
[tDiffDrive,diffDrivePose] = ode45(@(t,y)derivative(diffDrive,y,exampleHelperMobileRobotControlle
```

Plot Results

The results of the ODE solver can be easily viewed on a single plot using `plotTransforms` to visualize the results of all trajectories at once.

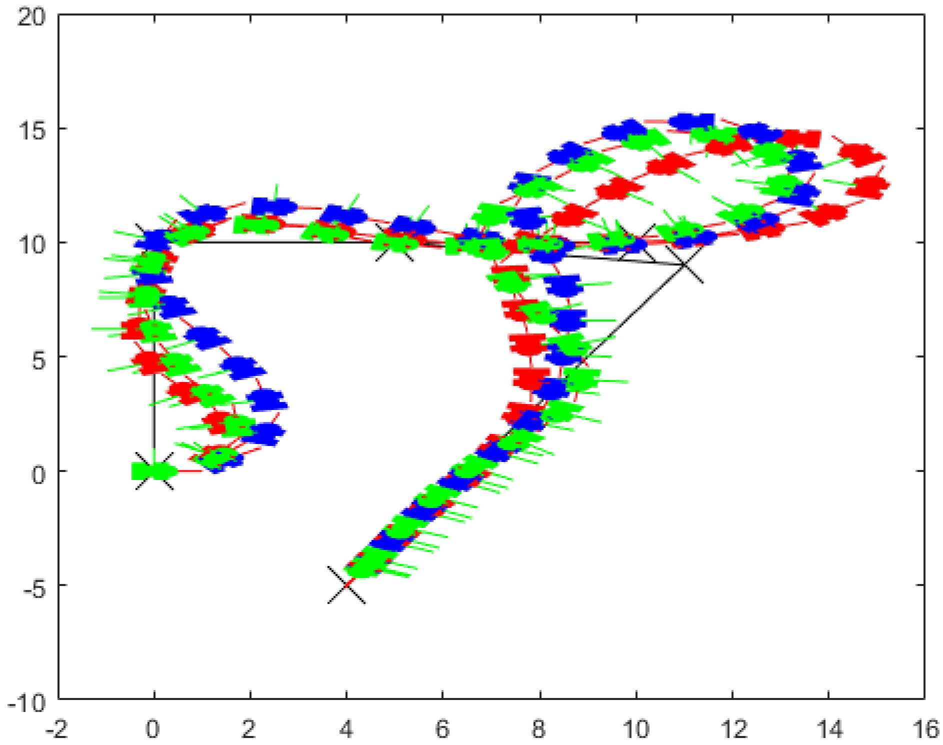
The pose outputs must first be converted to indexed matrices of translations and quaternions.

```
unicycleTranslations = [unicyclePose(:,1:2) zeros(length(unicyclePose),1)];
unicycleRot = axang2quat([repmat([0 0 1],length(unicyclePose),1) unicyclePose(:,3)]);
```

```
bicycleTranslations = [bicyclePose(:,1:2) zeros(length(bicyclePose),1)];
bicycleRot = axang2quat([repmat([0 0 1],length(bicyclePose),1) bicyclePose(:,3)]);

diffDriveTranslations = [diffDrivePose(:,1:2) zeros(length(diffDrivePose),1)];
diffDriveRot = axang2quat([repmat([0 0 1],length(diffDrivePose),1) diffDrivePose(:,3)]);
```

Next, the set of all transforms can be plotted and viewed from the top. The paths of the unicycle, bicycle, and differential-drive robots are red, blue, and green, respectively. To simplify the plot, only show every tenth output.

```
figure
plot(waypoints(:,1),waypoints(:,2),"kx-","MarkerSize",20);
hold all
plotTransforms(unicycleTranslations(1:10:end,:),unicycleRot(1:10:end,:),'MeshFilePath','groundveh
plotTransforms(bicycleTranslations(1:10:end,:),bicycleRot(1:10:end,:),'MeshFilePath','groundvehic
plotTransforms(diffDriveTranslations(1:10:end,:),diffDriveRot(1:10:end,:),'MeshFilePath','groundv
view(0,90)
```



Control a Differential Drive Robot in Gazebo with Simulink

This example shows how to control a differential drive robot in Gazebo co-simulation using Simulink. The robot follows a set of waypoints by reading the pose and wheel encoder positions and generates torque-control commands to drive it.

Run the VM

Follow the instructions in “Perform Co-Simulation between Simulink and Gazebo” on page 1-431 to download the virtual machine (VM) with Gazebo.

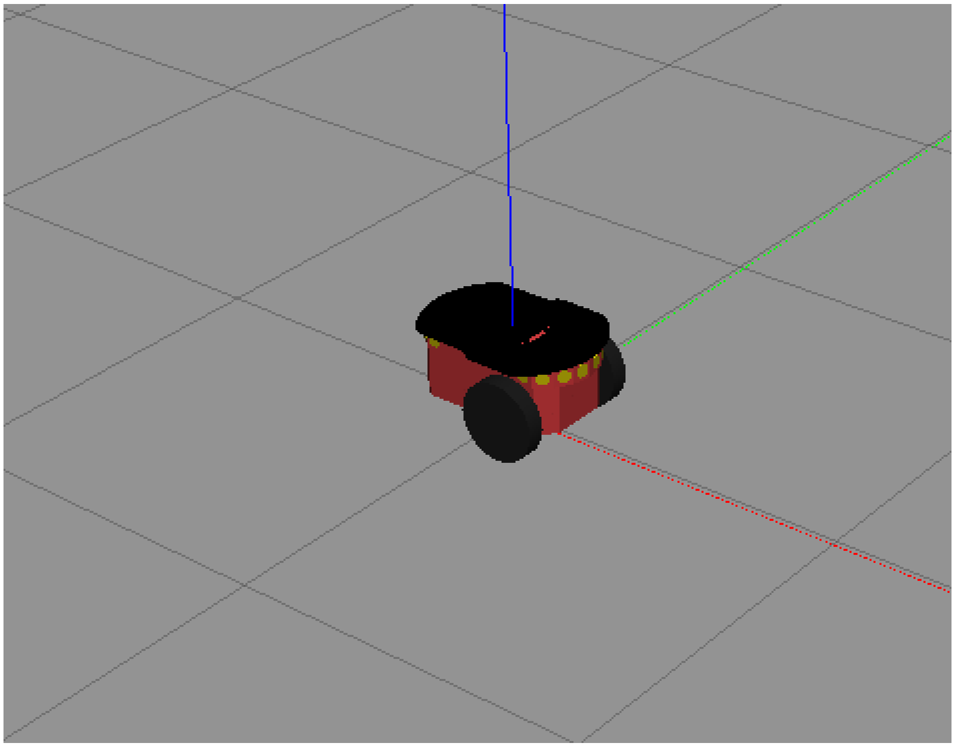
Gazebo World

This example uses a world given in the VM, `differentialDriveRobot.world`, as a simple ground plane with default physics settings. The world uses a Pioneer robot with the default controllers removed, so that the built-in controllers do not compete with torques provided from Simulink. The Pioneer robot is available in default Gazebo installs. The Gazebo plugin references the plugin required for the connection to Simulink, as detailed in “Perform Co-Simulation between Simulink and Gazebo” on page 1-431.

Double-click the **Gazebo Differential Drive Robot** icon.

Alternative, run these commands in the terminal:

```
cd /home/user/src/GazeboPlugin/export
export SVGA_VGPU10=0
gazebo ../world/differentialDriveRobot.world
```



If the Gazebo simulator fails to open, you may need to reinstall the plugin. See **Install Gazebo Plugin Manually** in “Perform Co-Simulation between Simulink and Gazebo” on page 1-431.

Model Overview

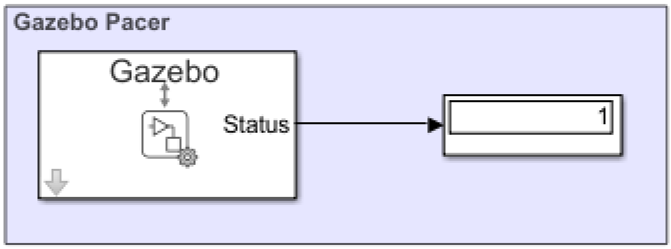
Open the model:

```
open_system('GazeboDifferentialDriveControl')
```

The model has four sections:

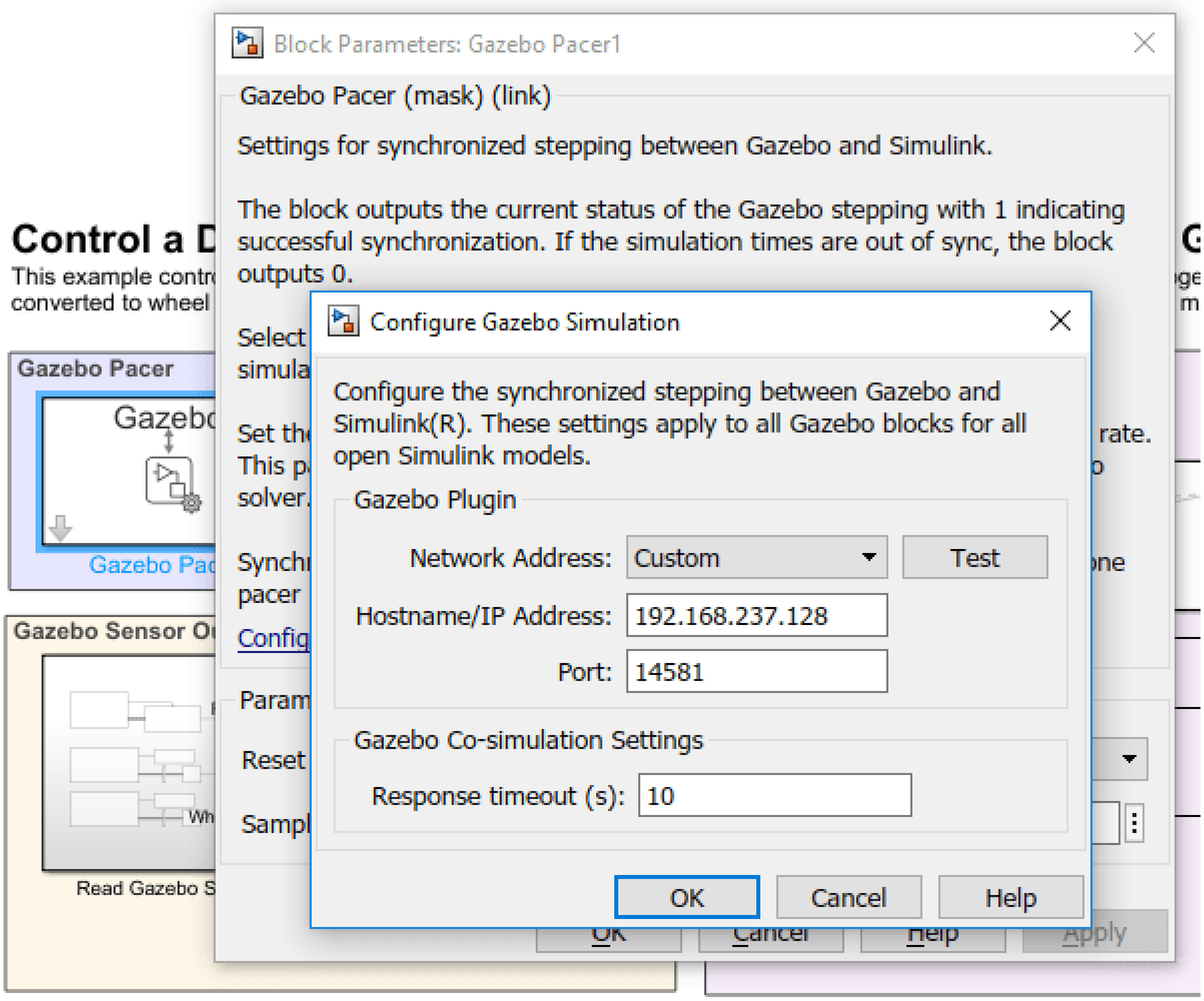
- Gazebo Pacer
- Read Sensor Data
- Control Mobile Robot
- Send Actuation Data to Gazebo

Gazebo Pacer



This section establishes the connection to Gazebo. Double-click the **Gazebo Pacer** block to open its parameters, and then click the **Configure Gazebo network and simulation settings** link. This will open a dialog.

Specify the **IP Address** for your VM. By default, Gazebo connects on the `14581` port. Click the **Test** button to verify the connection to Gazebo.



Block Parameters: Gazebo Pacer1

Gazebo Pacer (mask) (link)

Settings for synchronized stepping between Gazebo and Simulink.

The block outputs the current status of the Gazebo stepping with 1 indicating successful synchronization. If the simulation times are out of sync, the block outputs 0.

Configure Gazebo Simulation

Configure the synchronized stepping between Gazebo and Simulink(R). These settings apply to all Gazebo blocks for all open Simulink models.

Gazebo Plugin

Network Address: Custom

Hostname/IP Address: 192.168.237.128

Port: 14581

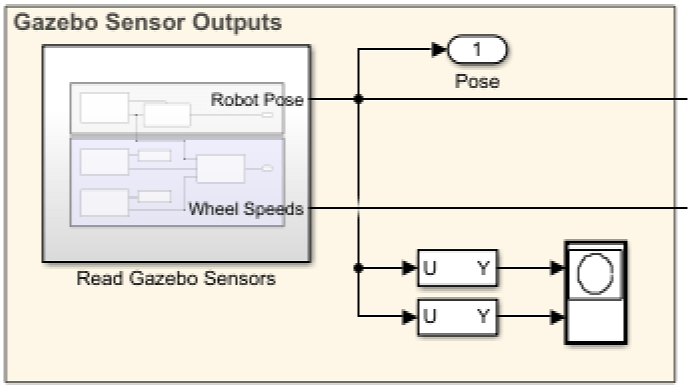
Gazebo Co-simulation Settings

Response timeout (s): 10

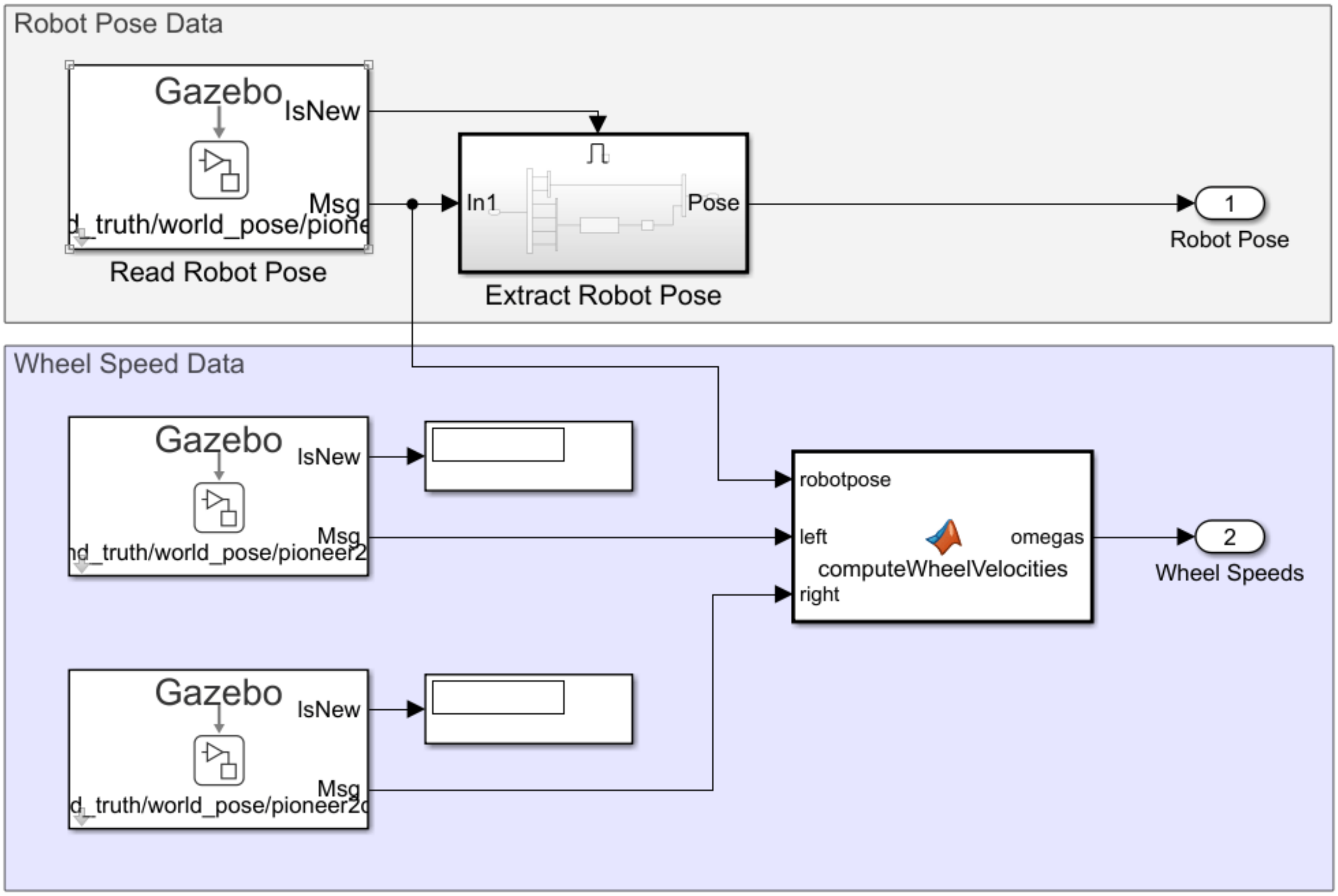
If the test is not successful, make sure to check the instructions in “Perform Co-Simulation between Simulink and Gazebo” on page 1-431 and ensure that Gazebo is properly configured and the associated world is up and running.

Gazebo Sensor Outputs

The sensor outputs read sensor data from Gazebo and passes it to the appropriate Simulink blocks. An XY graph plots the current robot position, and pose data is saved to the simulation output.

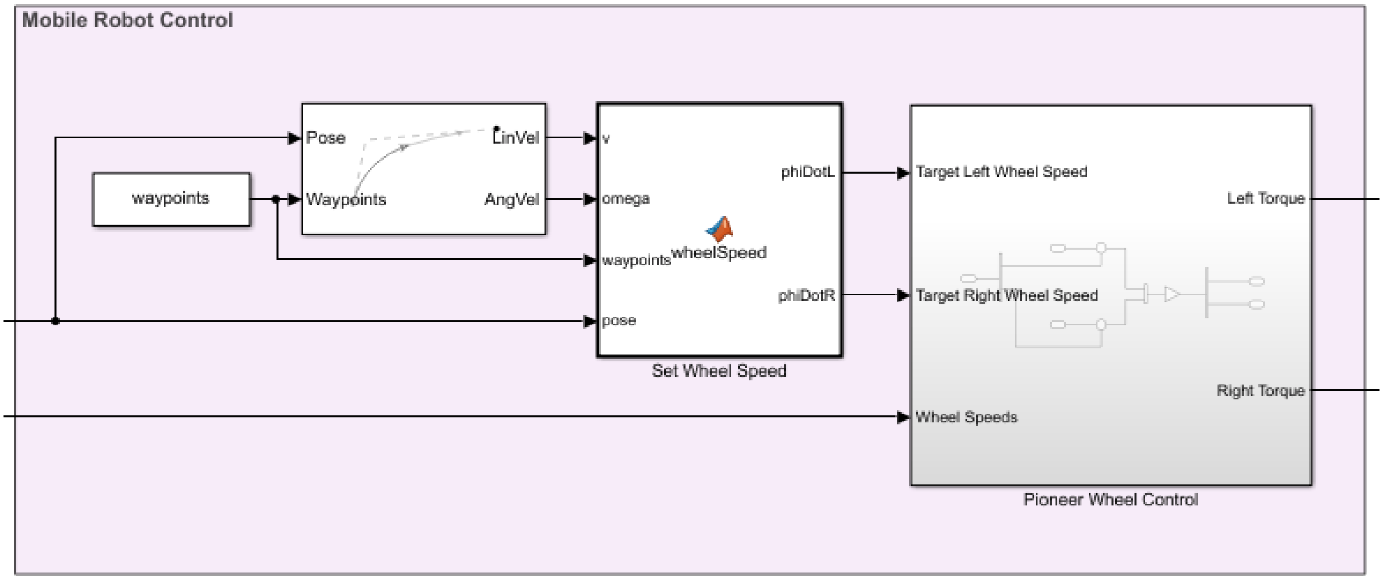


The **Read Gazebo Sensors** subsystem extracts the robot pose and wheel sensor data. The pose data are the *xy*-coordinates and a four-element quaternion for orientation. The wheel speeds are computed based on rate of change of the wheel positions as they rotate.



Mobile Robot Control

The **Mobile Robot Control** section accepts a set of target waypoints, current pose, and the current wheel speeds, and outputs the wheel torques needed to have the robot follow a path that pursues the waypoints.



There are three main components.

The **Pure Pursuit** block is a controller that specifies the vehicle speed and heading angular velocity of the vehicle needed to follow the waypoints at a fixed speed, given the current pose.

The **Set Wheel Speed** MATLAB Function block converts vehicle speed and heading angular velocity to left and right wheel speed, using the kinematics of a differential drive robot:

$$\dot{\phi}_L = \frac{1}{r}\left(v - \frac{\omega \mathrm{d}}{2}\right)$$

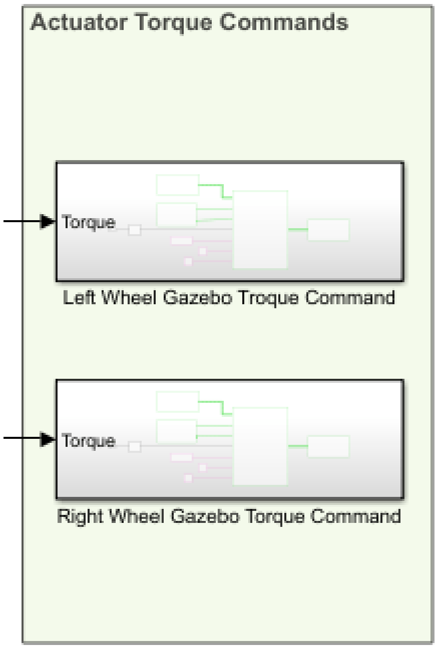
$$\dot{\phi}_R = \frac{1}{r}\left(v + \frac{\omega \mathrm{d}}{2}\right)$$

$\dot{\phi}_L$ and $\dot{\phi}_R$ are the left and right wheel speeds, v is the vehicle speed, ω is the vehicle heading angular velocity, d is the track width, and r is the wheel radius. Additionally, this MATLAB® Function includes code to throttle the wheel speed. Since the **Pure Pursuit** block uses a fixed speed throughout, inside the MATLAB Function block, there are two if-statements. The first slows the velocity at a rate proportional to the distance to the goal when the robot is within a certain distance threshold. The second if-statement stops the robot when it is within a tight threshold. This helps the robot to come to a gentle stop.

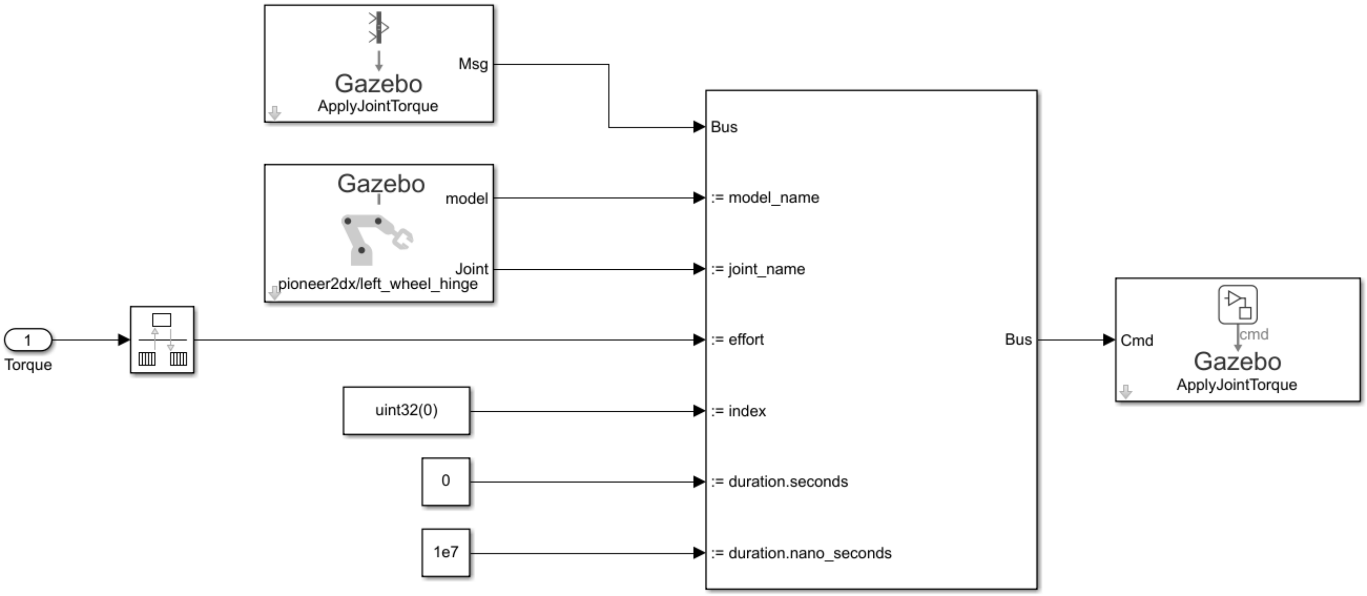
Finally, the **Pioneer Wheel Control** subsystem converts the desired wheel speeds to torques using a proportional controller.

Actuator Torque Commands

The last section of the model takes the torque commands produced by the controller and sends it to Simulink using blocks from the **Gazebo Co-Simulation Library.**


Actuator Torque Commands
Torque
Left Wheel Gazebo Troque Command
Torque
Right Wheel Gazebo Torque Command

Inside each of the subsystems in this block, a **Bus Assignment** block is used to assign the joint torque to the correct joint.



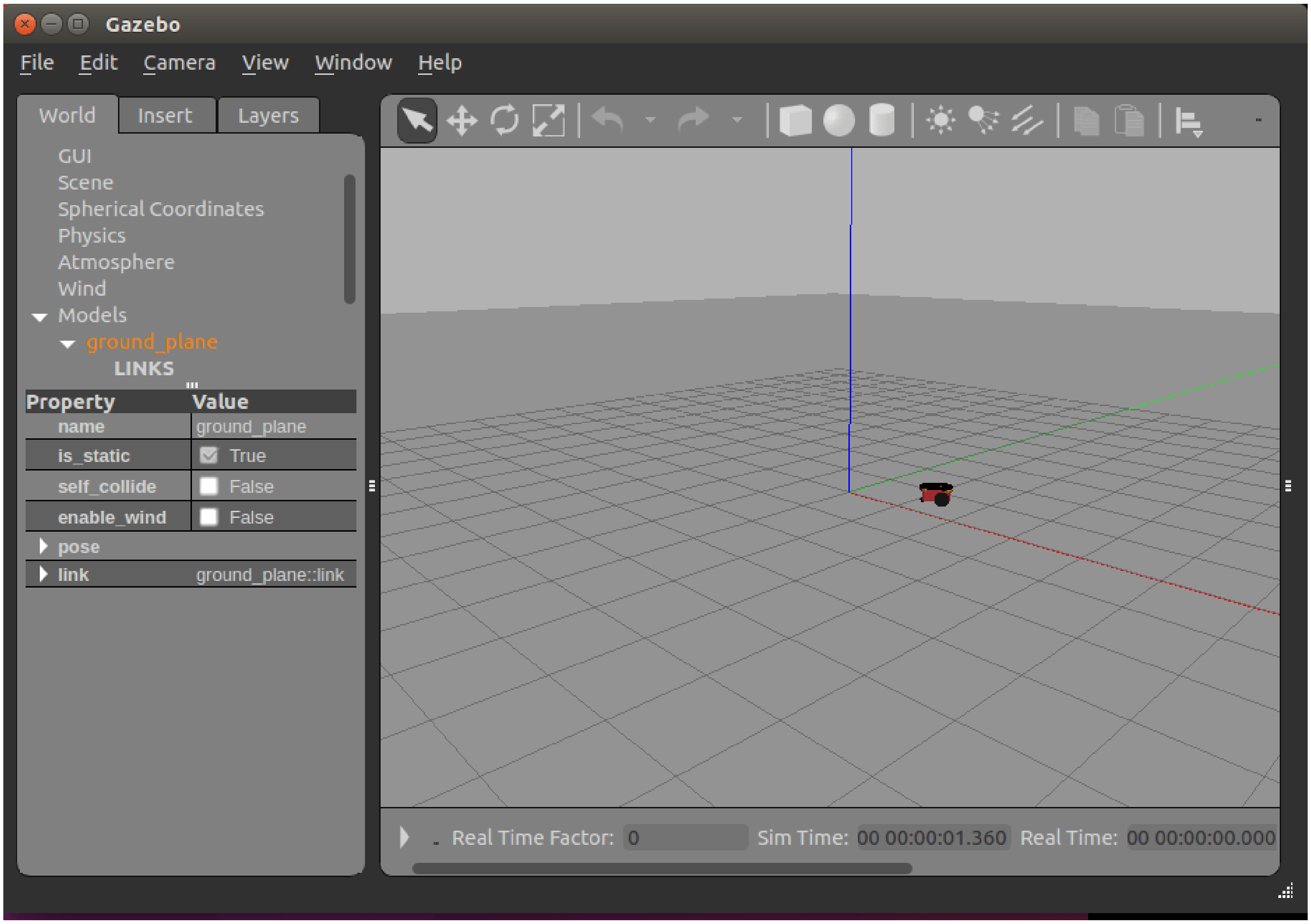
For example, inside the **Left Wheel Gazebo Torque Command** subsystem, shown above, a **Gazebo Blank Message** with the `ApplyJointTorque` command type is used to specify the bus type. The model and joint name are provided by the **Gazebo Select Entity** block, which is linked to the joint associated with the left wheel in the Gazebo world, `left_wheel_hinge`. The torque is applied for the entire step time, 0.01 seconds, specified in nanoseconds since these inputs must be provided as integers. The output of the bus is passed to a **Gazebo Apply Command** block.

Simulate the robot

To run the model, initialize the waypoints and set the sample time:

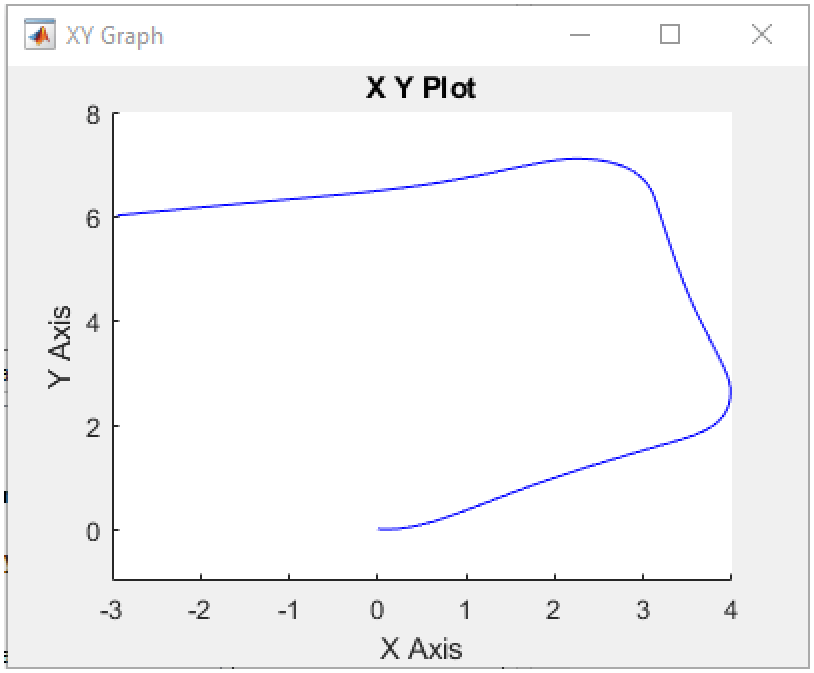
```
waypoints = [0 0; 4 2; 3 7; -3 6];
sampleTime = 0.01;
```

Click **Play** button or use the `sim` command to run the model. During execution, the robot should move in Gazebo, and an **XY Plot** updates the pose observed in Simulink.



Property	Value
name	ground_plane
is_static	☑ True
self_collide	□ False
enable_wind	□ False
pose	
link	ground_plane::link

Real Time Factor: 0 Sim Time: 00 00:00:01.360 Real Time: 00 00:00:00.000


XY Graph
X Y Plot
Y Axis
X Axis
8
6
4
2
0
-3
-2
-1
0
1
2
3
4

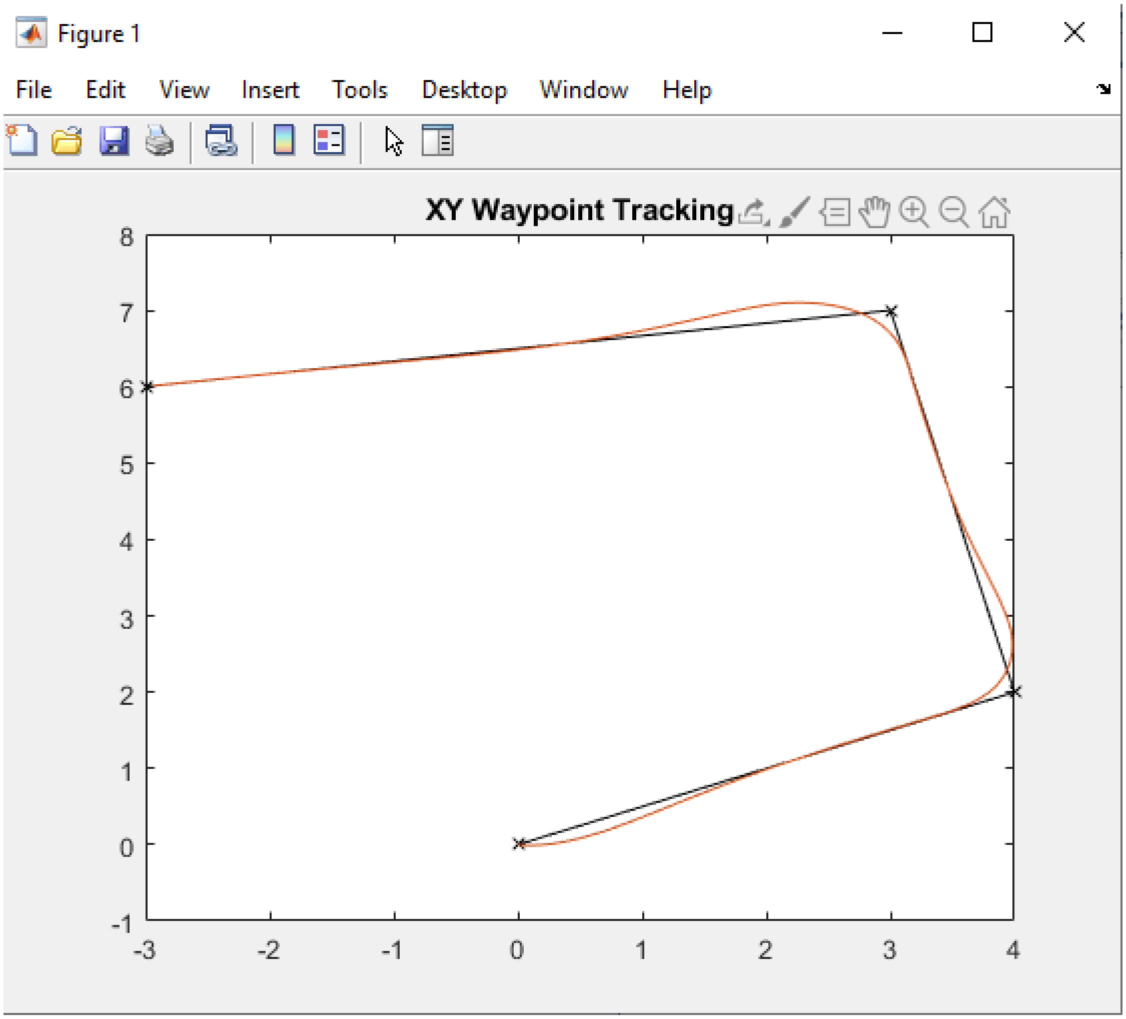


Figure 1
File
Edit
View
Insert
Tools
Desktop
Window
Help
XY Waypoint Tracking
8
7
6
5
4
3
2
1
0
-1
-3
-2
-1
0
1
2
3
4

The figures plot the set of waypoints and the final executed path of the robot.

Avoid Obstacles Using Reinforcement Learning for Mobile Robots

This example uses Deep Deterministic Policy Gradient (DDPG) based reinforcement learning to develop a strategy for a mobile robot to avoid obstacles. For a brief summary of the DDPG algorithm, see “Deep Deterministic Policy Gradient Agents” (Reinforcement Learning Toolbox).

This example scenario trains a mobile robot to avoid obstacles given range sensor readings that detect obstacles in the map. The objective of the reinforcement learning algorithm is to learn what controls (linear and angular velocity), the robot should use to avoid colliding into obstacles. This example uses an occupancy map of a known environment to generate range sensor readings, detect obstacles, and check collisions the robot may make. The range sensor readings are the observations for the DDPG agent, and the linear and angular velocity controls are the action.

Load Map

Load a map matrix, `simpleMap`, that represents the environment for the robot.

```
load exampleMaps simpleMap
load exampleHelperOfficeAreaMap office_area_map
mapMatrix = simpleMap;
mapScale = 1;
```

Range Sensor Parameters

Next, set up a `rangeSensor` object which simulates a noisy range sensor. The range sensor readings are considered observations by the agent. Define the angular positions of the range readings, the max range, and the noise parameters.

```
scanAngles = [-3*pi/8:pi/8:3*pi/8];
maxRange = 12;
lidarNoiseVariance = 0.1^2;
lidarNoiseSeeds = randi(intmax,size(scanAngles));
```

Robot Parameters

The action of the agent is a two-dimensional vector $a = [v, \omega]$ where v and ω are the linear and angular velocities of our robot. The DDPG agent uses normalized inputs for both the angular and linear velocities, meaning the actions of the agent are a scalar between -1 and 1, which is multiplied by the `maxLinSpeed` and `maxAngSpeed` parameters to get the actual control. Specify this maximum linear and angular velocity.

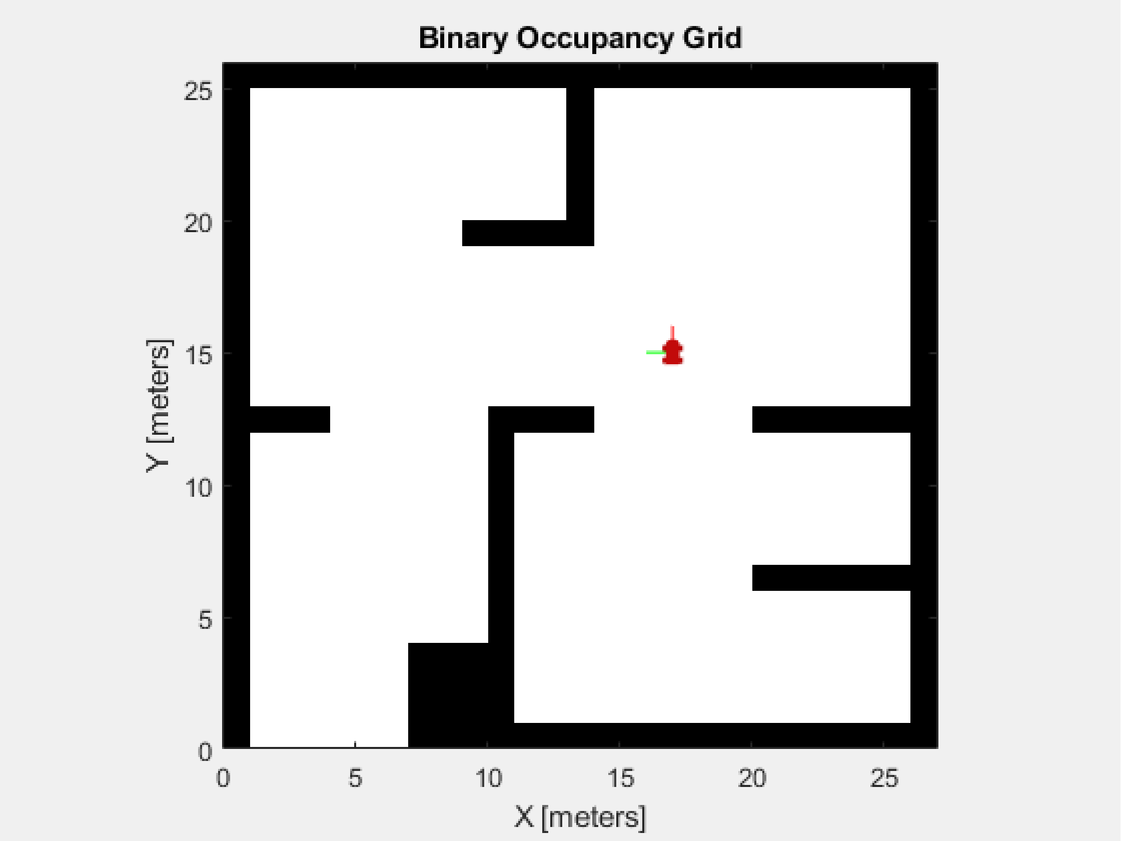
Also, specify the initial position of the robot as `[x y theta]`.

```
% Max speed parameters
maxLinSpeed = 0.3;
maxAngSpeed = 0.3;

% Initial pose of the robot
initX = 17;
initY = 15;
initTheta = pi/2;
```

Show Map and Robot Positions

To visualize the actions of the robot, create a figure. Start by showing the occupancy map and plot the initial position of the robot.

```
fig = figure("Name","simpleMap");
set(fig, "Visible","on");
ax = axes(fig);

show(binaryOccupancyMap(mapMatrix),"Parent",ax);
hold on
plotTransforms([initX,initY,0],eul2quat([initTheta,0,0]),"MeshFilePath","groundvehicle.stl","Vie
light;
hold off
```



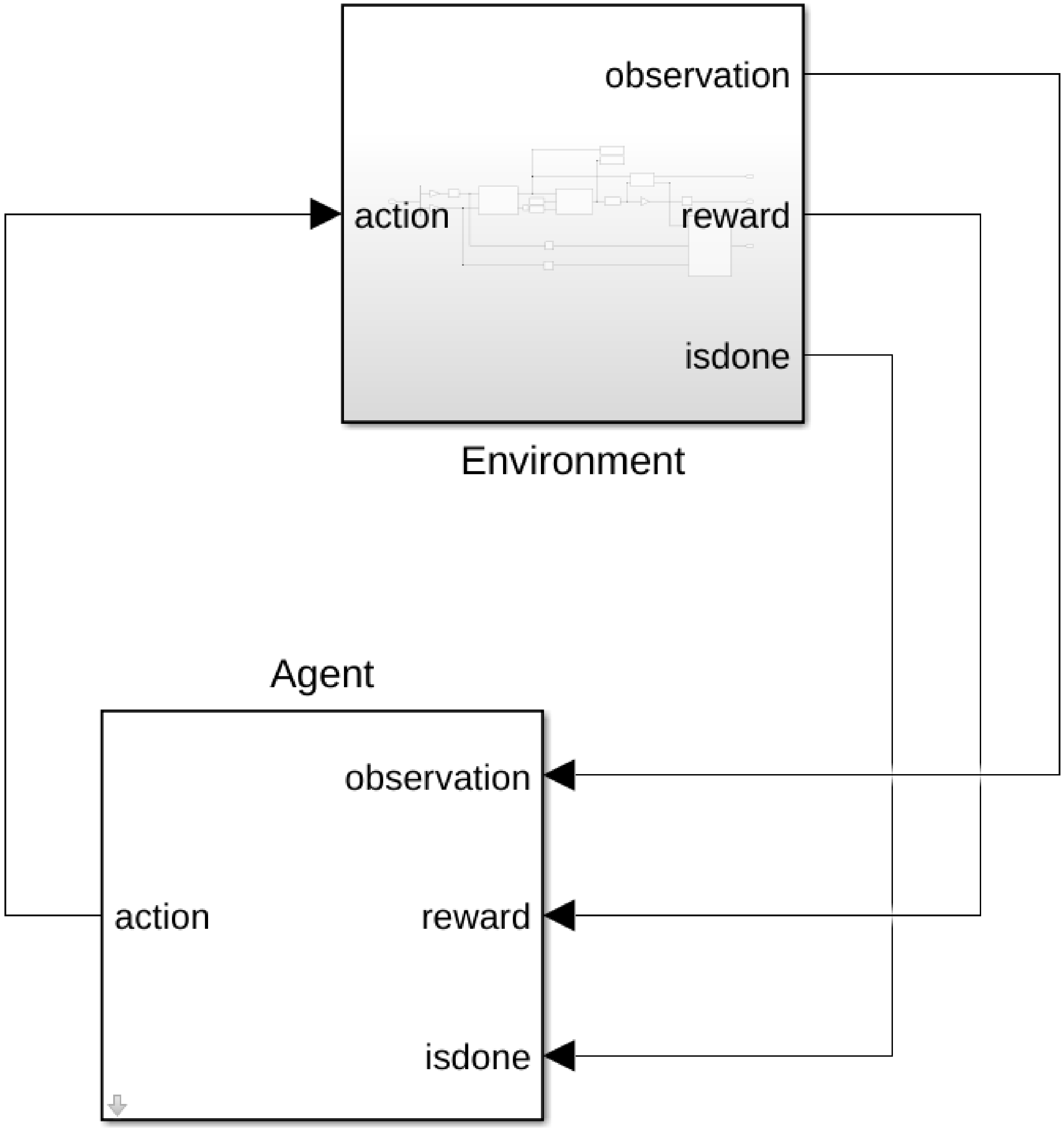
Environment Interface

Create an environment model that takes the action, and gives the observation and reward signals. Specify the provided example model name, `exampleHelperAvoidObstaclesMobileRobot`, the simulation time parameters, and the agent block name.

```
mdl = "exampleHelperAvoidObstaclesMobileRobot";
Tfinal = 100;
sampleTime = 0.1;

agentBlk = mdl + "/Agent";
```

Open the model.

```
open_system(mdl)
```



The model contains the `Environment` and `Agent` blocks. The `Agent` block is not defined yet.

Inside the `Environment` Subsystem block, you should see a model for simulating the robot and sensor data. The subsystem takes in the action, generates the observation signal based on the range sensor readings, and calculates the reward based on the distance from obstacles, and the effort of the action commands.

```
open_system(mdl + "/Environment")
```

Define observation parameters, obsInfo, using the rlNumericSpec object and giving the lower and upper limit for the range readings with enough elements for each angular position in the range sensor.

```
obsInfo = rlNumericSpec([numel(scanAngles) 1],...
    "LowerLimit",zeros(numel(scanAngles),1),...
    "UpperLimit",ones(numel(scanAngles),1)*maxRange);
numObservations = obsInfo.Dimension(1);
```

Define action parameters, actInfo. The action is the control command vector, $a = [v, \omega]$, normalized to $[-1, 1]$.

```
numActions = 2;
actInfo = rlNumericSpec([numActions 1],...
    "LowerLimit",-1,...
    "UpperLimit",1);
```

Build the environment interface object using rlSimulinkEnv (Reinforcement Learning Toolbox). Specify the model, agent block name, observation parameters, and action parameters. Set the reset function for the simulation using exampleHelperRLAvoidObstaclesResetFcn. This function restarts the simulation by placing the robot in a new random location to begin avoiding obstacles.

```
env = rlSimulinkEnv(mdl,agentBlk,obsInfo,actInfo);
env.ResetFcn = @(in)exampleHelperRLAvoidObstaclesResetFcn(in,scanAngles,maxRange,mapMatrix);
env.UseFastRestart = "Off";
```

For another example that sets up a Simulink® environment for training, see “Create Simulink Environment and Train Agent” (Reinforcement Learning Toolbox).

DDPG Agent

A DDPG agent approximates the long-term reward given observations and actions using a critic value function representation. To create the critic, first create a deep neural network with two inputs, the observation and action, and one output. For more information on creating a deep neural network value function representation, see “Create Policies and Value Functions” (Reinforcement Learning Toolbox).

```
statePath = [
    featureInputLayer(numObservations, "Normalization","none","Name","State")
    fullyConnectedLayer(50,"Name","CriticStateFC1")
    reluLayer("Name","CriticRelu1")
    fullyConnectedLayer(25,"Name","CriticStateFC2")];
actionPath = [
    featureInputLayer(numActions,"Normalization","none","Name","Action")
    fullyConnectedLayer(25,"Name","CriticActionFC1")];
commonPath = [
    additionLayer(2,"Name","add")
    reluLayer("Name","CriticCommonRelu")
    fullyConnectedLayer(1,"Name","CriticOutput")];

criticNetwork = layerGraph();
criticNetwork = addLayers(criticNetwork,statePath);
criticNetwork = addLayers(criticNetwork,actionPath);
criticNetwork = addLayers(criticNetwork,commonPath);
criticNetwork = connectLayers(criticNetwork,"CriticStateFC2","add/in1");
criticNetwork = connectLayers(criticNetwork,"CriticActionFC1","add/in2");
criticNetwork = dlnetwork(criticNetwork);
```

Next, specify options for the critic optimizer using `rlOptimizerOptions`.

Finally, create the critic representation using the specified deep neural network and options. You must also specify the action and observation specifications for the critic, which you obtain from the environment interface. For more information, see `rlQValueFunction` (Reinforcement Learning Toolbox).

```
criticOptions = rlOptimizerOptions("LearnRate",1e-3,"L2RegularizationFactor",1e-4,"GradientThresh
critic = rlQValueFunction(criticNetwork,obsInfo,actInfo,"ObservationInputNames","State","ActionIn
```

A DDPG agent decides which action to take given observations using an actor representation. To create the actor, first create a deep neural network with one input, the observation, and one output, the action.

Finally, construct the actor in a similar manner as the critic. For more information, see `rlContinuousDeterministicActor` (Reinforcement Learning Toolbox).

```
actorNetwork = [
    featureInputLayer(numObservations,"Normalization","none","Name","State")
    fullyConnectedLayer(50,"Name","actorFC1")
    reluLayer("Name","actorReLU1")
    fullyConnectedLayer(50, "Name","actorFC2")
    reluLayer("Name","actorReLU2")
    fullyConnectedLayer(2, "Name","actorFC3")
    tanhLayer("Name","Action")];
actorNetwork = dlnetwork(actorNetwork);

actorOptions = rlOptimizerOptions("LearnRate",1e-4,"L2RegularizationFactor",1e-4,"GradientThresho
actor = rlContinuousDeterministicActor(actorNetwork,obsInfo,actInfo);
```

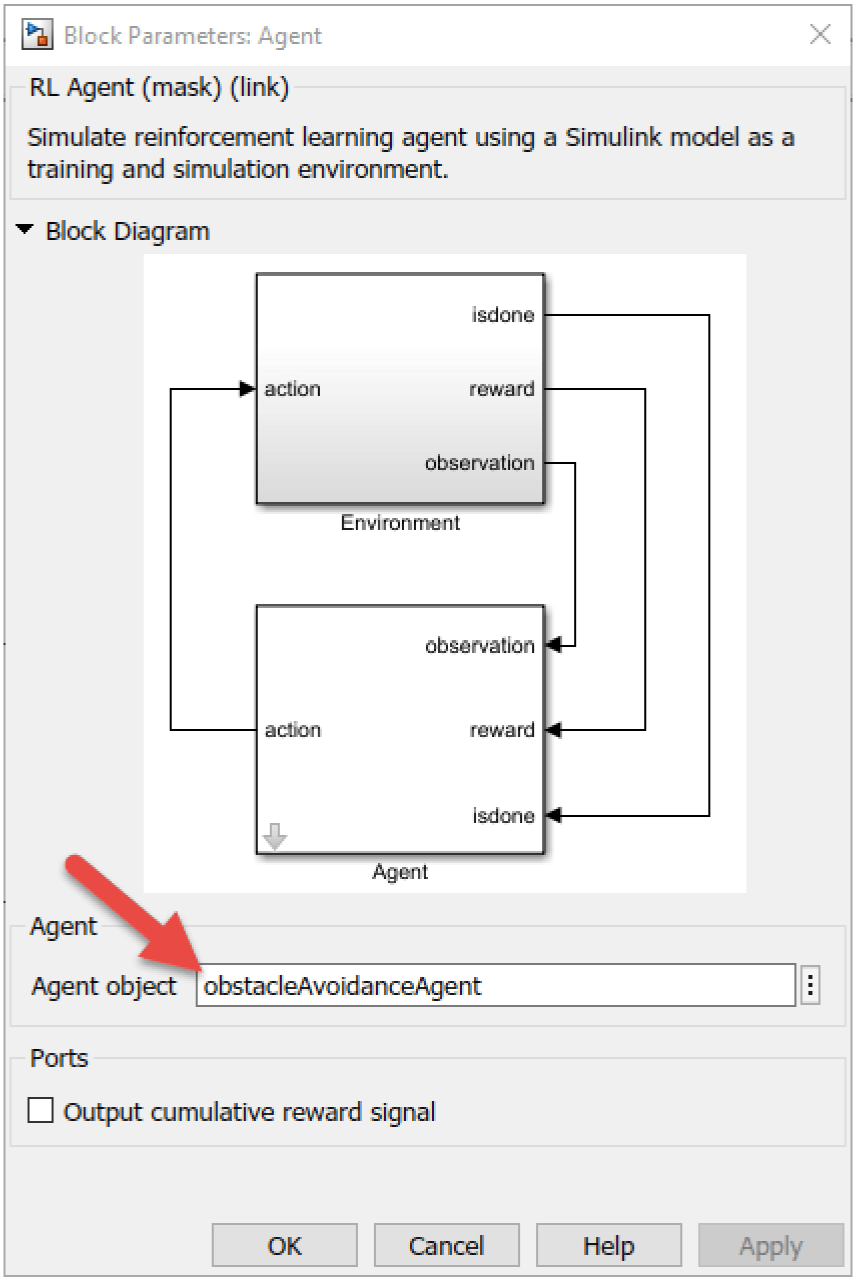
Create DDPG agent object

Specify the agent options.

```
agentOpts = rlDDPGAgentOptions(...
    "SampleTime",sampleTime,...
    "ActorOptimizerOptions",actorOptions,...
    "CriticOptimizerOptions",criticOptions,...
    "DiscountFactor",0.995, ...
    "MiniBatchSize",128, ...
    "ExperienceBufferLength",1e6);

agentOpts.NoiseOptions.Variance = 0.1;
agentOpts.NoiseOptions.VarianceDecayRate = 1e-5;
```

Create the `rlDDPGAgent` object. The `obstacleAvoidanceAgent` variable is used in the model for the `Agent` block.

```
obstacleAvoidanceAgent = rlDDPGAgent(actor,critic,agentOpts);
open_system(mdl + "/Agent")
```



Block Parameters: Agent

RL Agent (mask) (link)

Simulate reinforcement learning agent using a Simulink model as a training and simulation environment.

Block Diagram

Agent

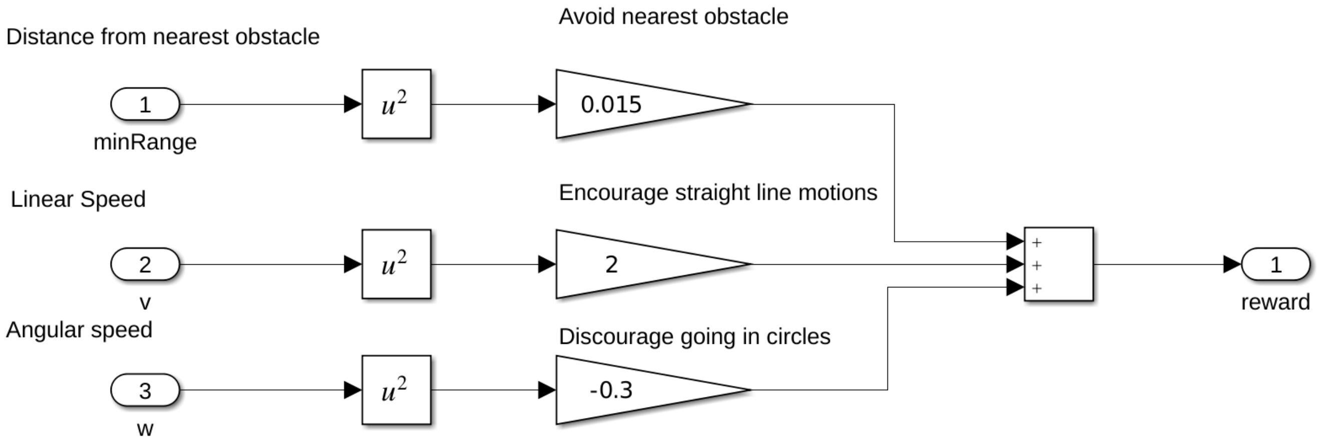
Agent object: obstacleAvoidanceAgent

Ports

□ Output cumulative reward signal

Reward

The reward function for the agent is modeled as shown.



The agent is rewarded to avoid the nearest obstacle, which minimizes the worst-case scenario. Additionally, the agent is given a positive reward for higher linear speeds, and is given a negative reward for higher angular speeds. This rewarding strategy discourages the agent's behavior of going in circles. Tuning your rewards is key to properly training an agent, so your rewards vary depending on your application.

Train Agent

To train the agent, first specify the training options. For this example, use the following options:

- Train for at most `10000` episodes, with each episode lasting at most `maxSteps` time steps.
- Display the training progress in the Episode Manager dialog box (set the `Plots` option) and enable the command line display (set the `Verbose` option to true).
- Stop training when the agent receives an average cumulative reward greater than 400 over fifty consecutive episodes.

For more information, see `rlTrainingOptions` (Reinforcement Learning Toolbox).

```
maxEpisodes = 10000;
maxSteps = ceil(Tfinal/sampleTime);
trainOpts = rlTrainingOptions(...
    "MaxEpisodes",maxEpisodes, ...
    "MaxStepsPerEpisode",maxSteps, ...
    "ScoreAveragingWindowLength",50, ...
    "StopTrainingCriteria","AverageReward", ...
    "StopTrainingValue",400, ...
    "Verbose", true, ...
    "Plots","training-progress");
```

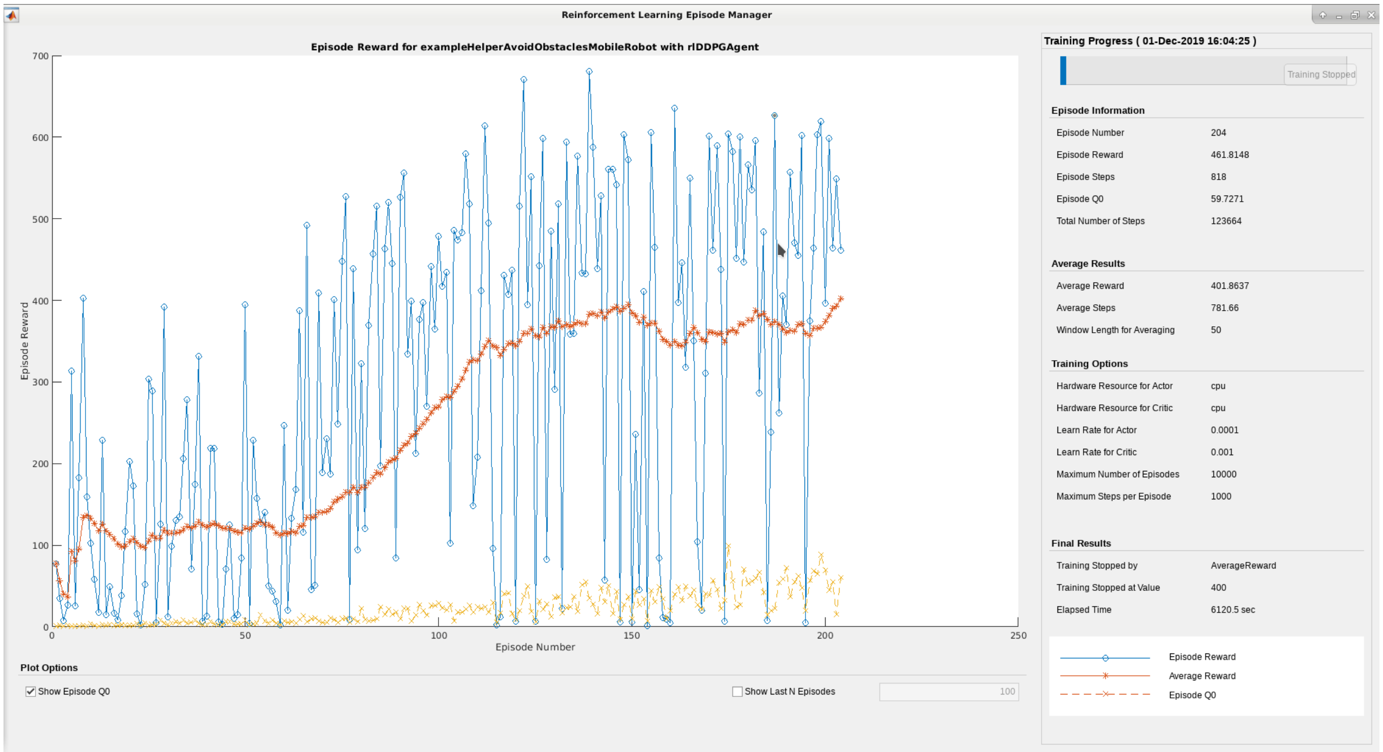
Train the agent using the `train` (Reinforcement Learning Toolbox) function. Training is a computationally-intensive process that takes several minutes to complete. To save time while running this example, load a pretrained agent by setting `doTraining` to `false`. To train the agent yourself, set `doTraining` to `true`.

```
doTraining = false; % Toggle this to true for training.
```

```
if doTraining
```

```
    % Train the agent.
    trainingStats = train(obstacleAvoidanceAgent,env,trainOpts);
else
    % Load pretrained agent for the example.
    load exampleHelperAvoidObstaclesAgent obstacleAvoidanceAgent
end
```

The **Reinforcement Learning Episode Manager** can be used to track episode-wise training progress. As the episode number increases, you want to see an increase in the reward value.



Reinforcement Learning Episode Manager
Episode Reward for exampleHelperAvoidObstaclesMobileRobot with rlDDPGAgent
Episode Reward
700
600
500
400
300
200
100
0
0
50
100
150
200
250
Episode Number
Training Stopped

Plot Options

☑ Show Episode Q0

☐ Show Last N Episodes 100

Training Progress (01-Dec-2019 16:04:25)

Episode Information

Episode Number	204
Episode Reward	461.8148
Episode Steps	818
Episode Q0	59.7271
Total Number of Steps	123664

Average Results

Average Reward	401.8637
Average Steps	781.66
Window Length for Averaging	50

Training Options

Hardware Resource for Actor	cpu
Hardware Resource for Critic	cpu
Learn Rate for Actor	0.0001
Learn Rate for Critic	0.001
Maximum Number of Episodes	10000
Maximum Steps per Episode	1000

Final Results

Training Stopped by	AverageReward
Training Stopped at Value	400
Elapsed Time	6120.5 sec

Simulate

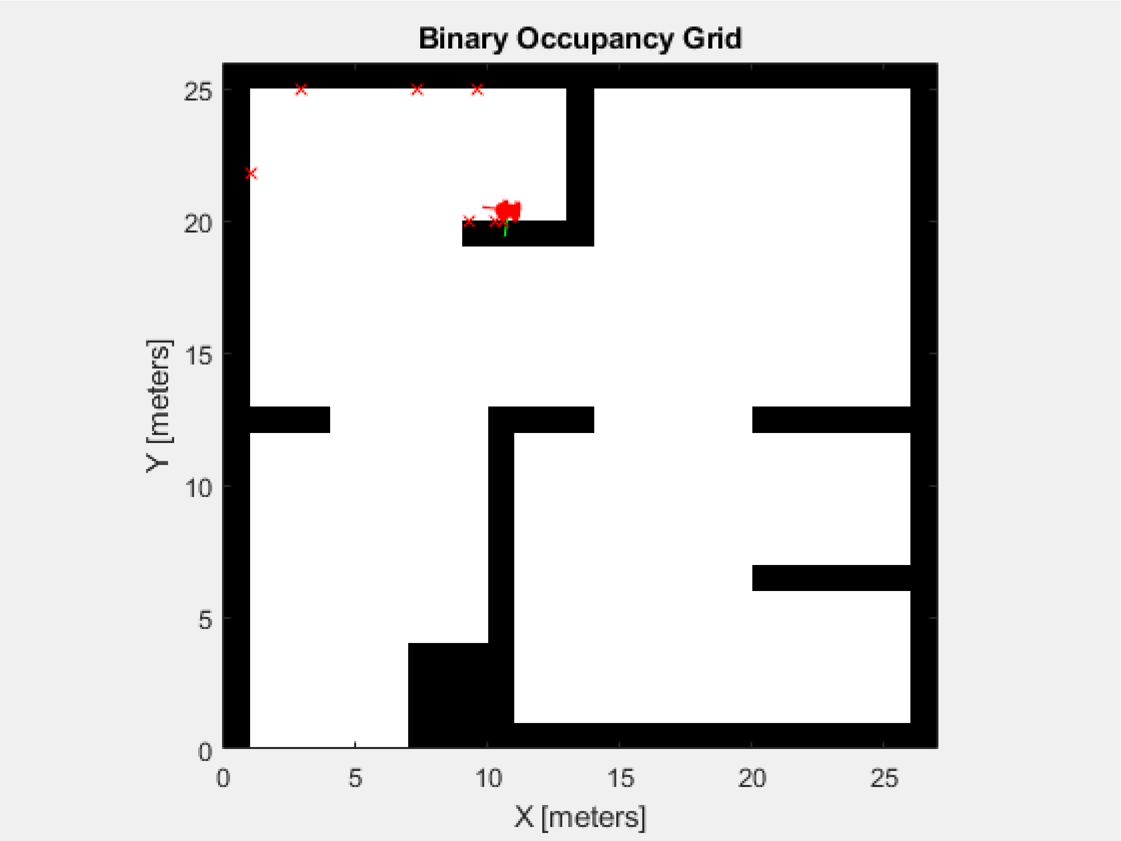
Use the trained agent to simulate the robot driving in the map and avoiding obstacles.

```
out = sim("exampleHelperAvoidObstaclesMobileRobot.slx");
```

Visualize

To visualize the simulation of the robot driving around the environment with range sensor readings, use the helper, `exampleHelperAvoidObstaclesPosePlot`.

```
for i = 1:5:size(out.range,3)
    u = out.pose(i,:);
    r = out.range(:,:,i);
    exampleHelperAvoidObstaclesPosePlot(u,mapMatrix,mapScale,r,scanAngles,ax);
end
```

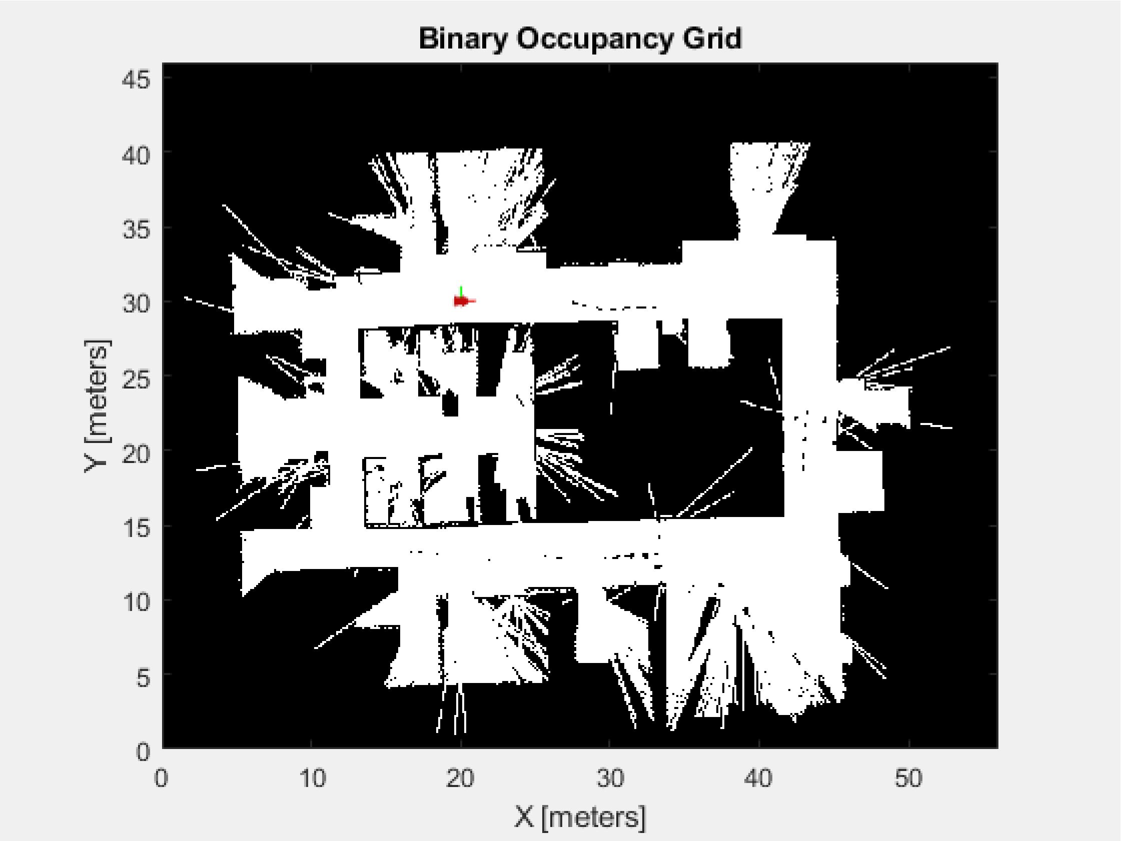


Extensibility

You can now use this agent to simulate driving in a different map. Another map generated from lidar scans of an office environment is used with the same trained model. This map represents a more realistic scenario to apply the trained model after training.

Change the map

```
mapMatrix = office_area_map.occupancyMatrix > 0.5;
mapScale = 10;
initX = 20;
initY = 30;
initTheta = 0;
fig = figure("Name","office_area_map");
set(fig,"Visible","on");
ax = axes(fig);
show(binaryOccupancyMap(mapMatrix,mapScale),"Parent",ax);
hold on
plotTransforms([initX,initY,0],eul2quat([initTheta, 0, 0]),"MeshFilePath","groundvehicle.stl","V
light;
hold off
```

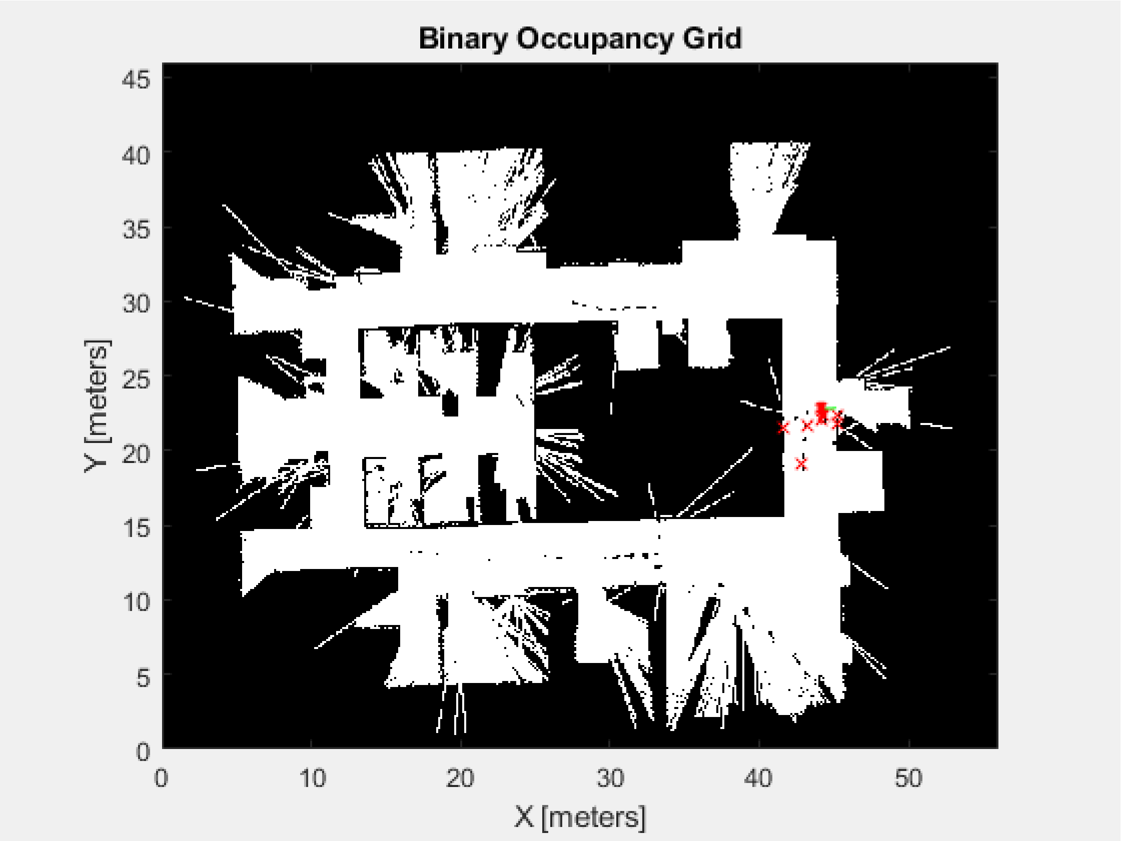


Simulate

```
out = sim("exampleHelperAvoidObstaclesMobileRobot.slx");
```

Visualize

```
for i = 1:5:size(out.range,3)
    u = out.pose(i,:);
    r = out.range(:,:,i);
    exampleHelperAvoidObstaclesPosePlot(u,mapMatrix,mapScale,r,scanAngles,ax);
end
```


Binary Occupancy Grid
45
40
35
30
25
20
15
10
5
0
0
10
20
30
40
50
Y [meters]
X [meters]

A* Path Planning and Obstacle Avoidance in a Warehouse

This example is an extension to the “Simulate a Mobile Robot in a Warehouse Using Gazebo” on page 1-53 example. The example shows to change the PRM path planner with an A* planner, and add a vector field histogram (VFH) algorithm to avoid obstacles.

Prerequisites

- Review the “Simulate a Mobile Robot in a Warehouse Using Gazebo” on page 1-53 example to setup the sensing and actuation elements. This example goes over how to download and use a virtual machine (VM) to setup a simulated robot.
- Review the “Execute Tasks for a Warehouse Robot” on page 1-31 example for the workflow of path planning and navigating in a warehouse scenario.

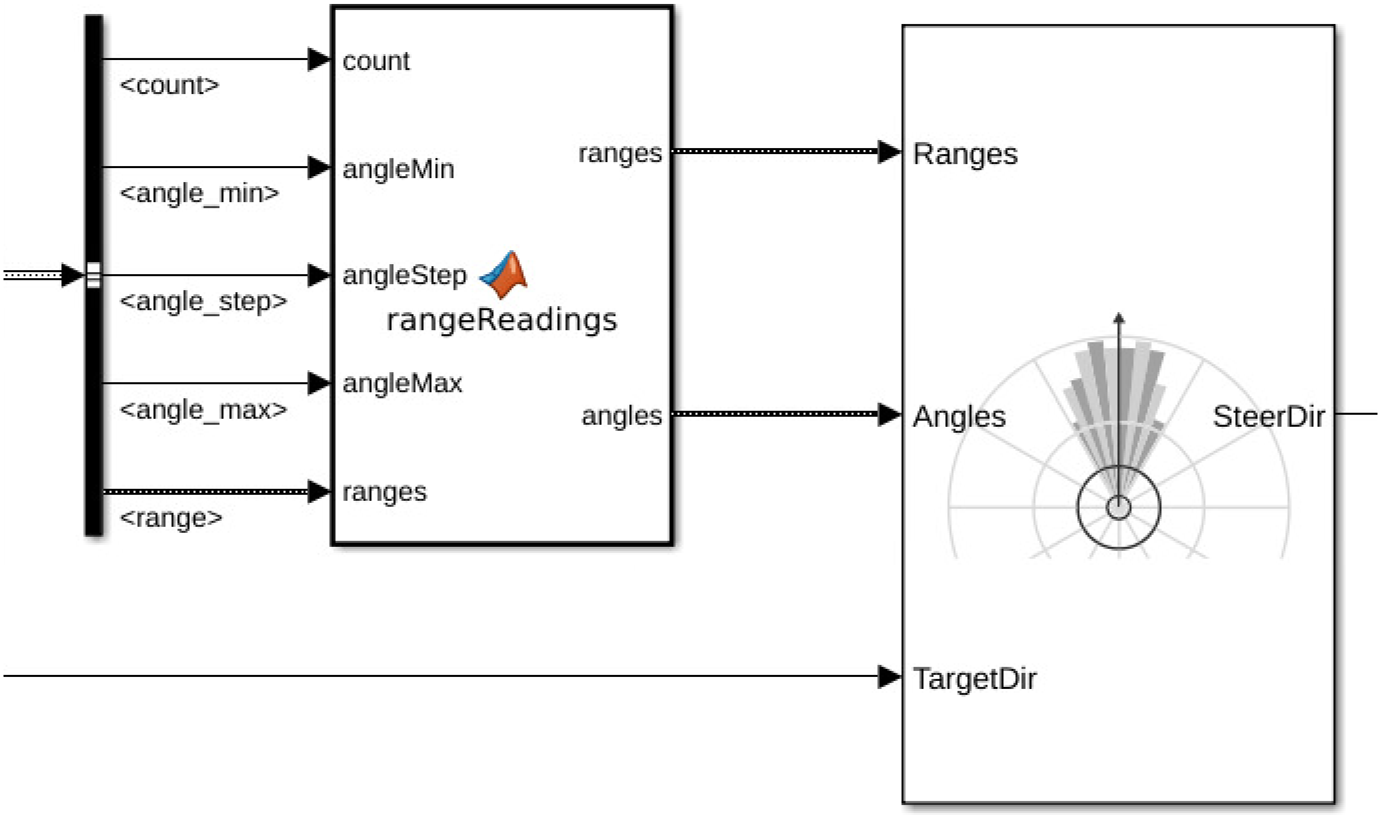
Model Overview

There are two major changes to this model from the “Execute Tasks for a Warehouse Robot” on page 1-31 example. The goal is to replace the path planner algorithm used and add a controller that avoids obstacles in the environment.

The **Planner** MATLAB® Function Block now uses the `plannerAStarGrid` (Navigation Toolbox) object to run the A* path planning algorithm.

The **Obstacle Avoidance** subsystem now uses a **Vector Field Histogram** block as part of the controller. The `rangeReadings` function block outputs the ranges and angles when the data received is not empty. The VFH block then generates a steering direction based on obstacles within the scan range. For close obstacles, the robot should turn to drive around them. Tune the VFH parameters for different obstacle avoidance performance.

```
open_system("aStarPathPlanningAndObstacleAvoidanceInWarehouse.slx");
```



Setup

Warehouse Facility

Load the example map file, `map`, which is a matrix of logical values indicating occupied space in the warehouse. Invert this matrix to indicate free space, and create a `binaryOccupancyMap` object. Specify a resolution of 100 cells per meter.

The map is based off of the `obstacleAvoidanceWorld.world`, which is loaded in the VM. A PNG-file was generated to use as the map matrix with the `collision_map_creator_plugin` plugin. For more information, see Collision Map Creator Plugin.

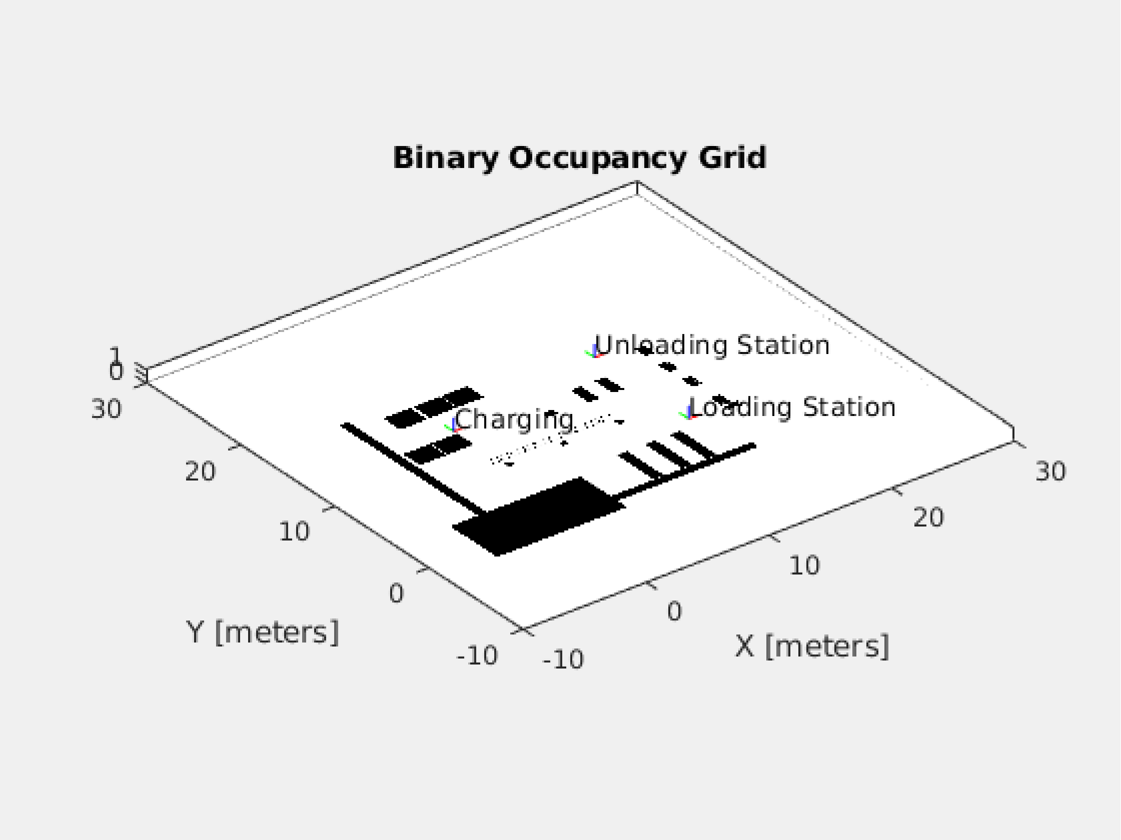
```
close
figure("Name","Warehouse Map","Visible","on")
load exampleHelperWarehouseRobotWithGazeboBuses.mat
load helperPlanningAndObstacleAvoidanceWarehouseMap.mat map
logicalMap = map.getOccupancy;
mapScalingFactor = 100;
show(map)
```

Assign the *xy*-locations of the charging station, sorting station, and the unloading location near shelves in the warehouse. The values chosen are based on the simlated world in Gazebo.

```
chargingStn = [2, 13];
loadingStn = [15, 5];
unloadingStn = [15, 15];
```

Show the various locations on the map.

```
hold on;
localOrigin = map.LocalOriginInWorld;
localTform = trvec2tform([localOrigin 0]);
text(chargingStn(1), chargingStn(2),1,'Charging');
plotTransforms([chargingStn, 0],[1 0 0 0])

text(loadingStn(1), loadingStn(2),1,'Loading Station');
plotTransforms([loadingStn, 0], [1 0 0 0])

text(unloadingStn(1), unloadingStn(2),1,'Unloading Station');
plotTransforms([unloadingStn, 0], [1 0 0 0])

hold off;
```



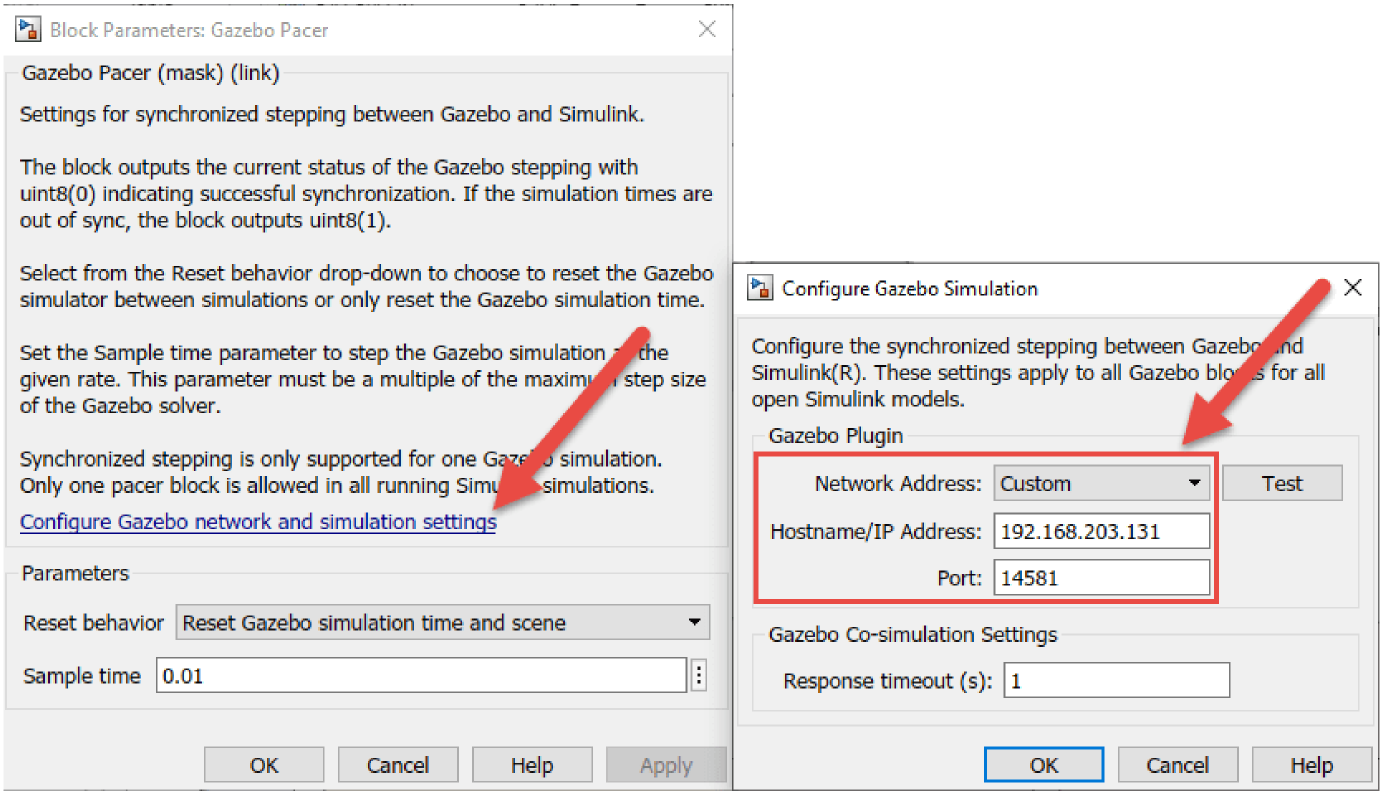
Simulate

To simulate the scenario, set up the connection to Gazebo.

First, run the Gazebo Simulator. In the virtual machine, click the **Gazebo Warehouse Robot with Obstacles** icon. If the Gazebo simulator fails to open, you may need to reinstall the plugin. See **Install Gazebo Plugin Manually** in “Perform Co-Simulation between Simulink and Gazebo” on page 1-431.

In Simulink, open the **Gazebo Pacer** block and click **Configure Gazebo network and simulation settings**. Specify the **Network Address** as **Custom**, the **Hostname/IP Address** for your Gazebo simulation, and a **Port** of 14581, which is the default port for Gazebo. The desktop of the VM displays the IP address.

For more information about connecting to Gazebo to enable co-simulation, see “Perform Co-Simulation between Simulink and Gazebo” on page 1-431.



Block Parameters: Gazebo Pacer

Gazebo Pacer (mask) (link)

Settings for synchronized stepping between Gazebo and Simulink.

The block outputs the current status of the Gazebo stepping with uint8(0) indicating successful synchronization. If the simulation times are out of sync, the block outputs uint8(1).

Select from the Reset behavior drop-down to choose to reset the Gazebo simulator between simulations or only reset the Gazebo simulation time.

Set the Sample time parameter to step the Gazebo simulation at the given rate. This parameter must be a multiple of the maximum step size of the Gazebo solver.

Synchronized stepping is only supported for one Gazebo simulation. Only one pacer block is allowed in all running Simulink simulations.

Configure Gazebo network and simulation settings

Parameters

Reset behavior: Reset Gazebo simulation time and scene

Sample time: 0.01

Configure Gazebo Simulation

Configure the synchronized stepping between Gazebo and Simulink(R). These settings apply to all Gazebo blocks for all open Simulink models.

Gazebo Plugin

Network Address: Custom

Hostname/IP Address: 192.168.203.131

Port: 14581

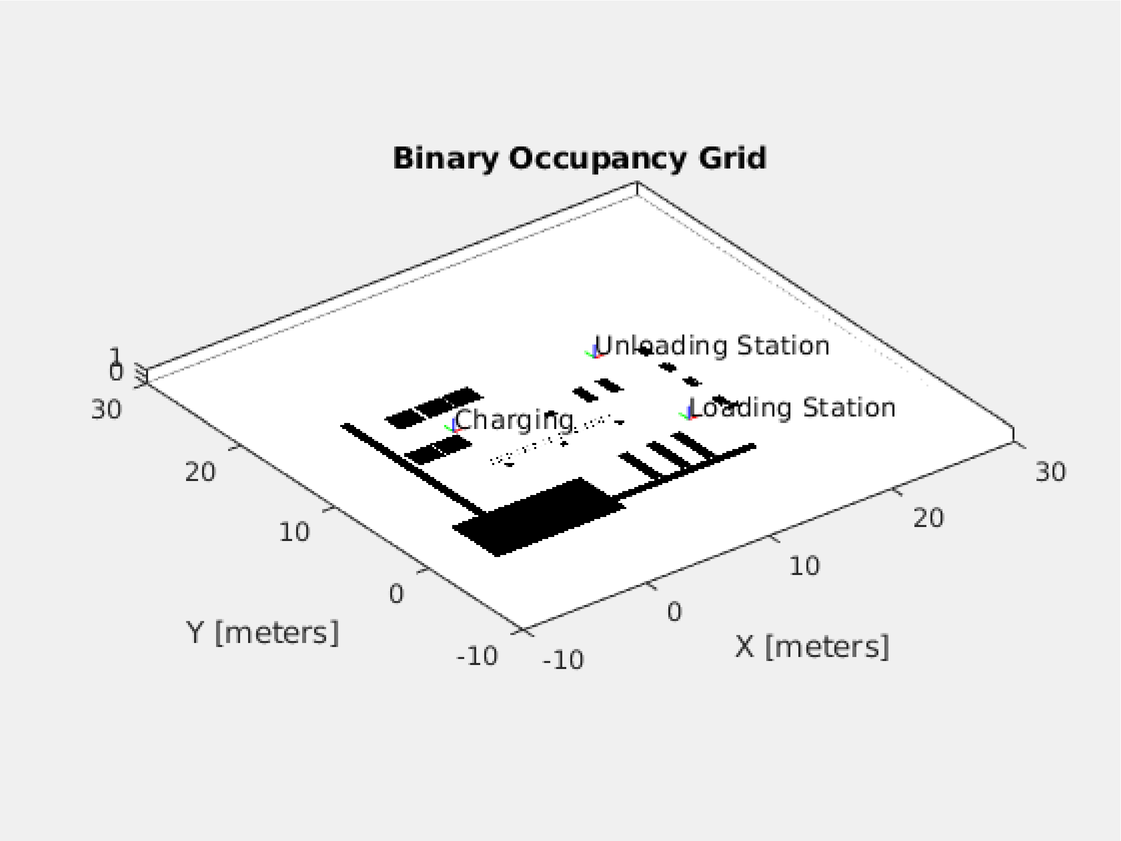
Gazebo Co-simulation Settings

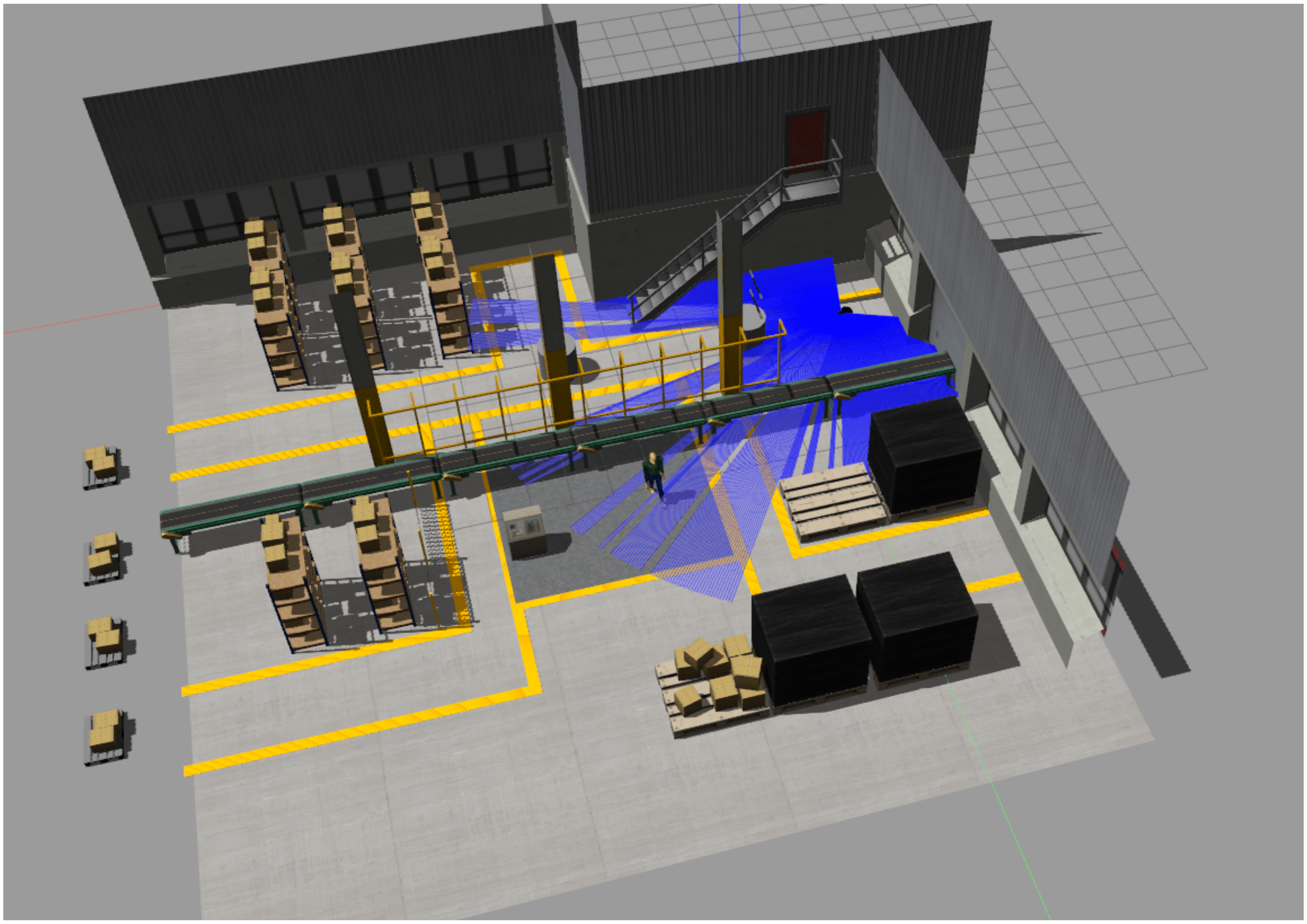
Response timeout (s): 1

Click the **Initialize Model** button at the top of the model to intialize all the variables declared above.

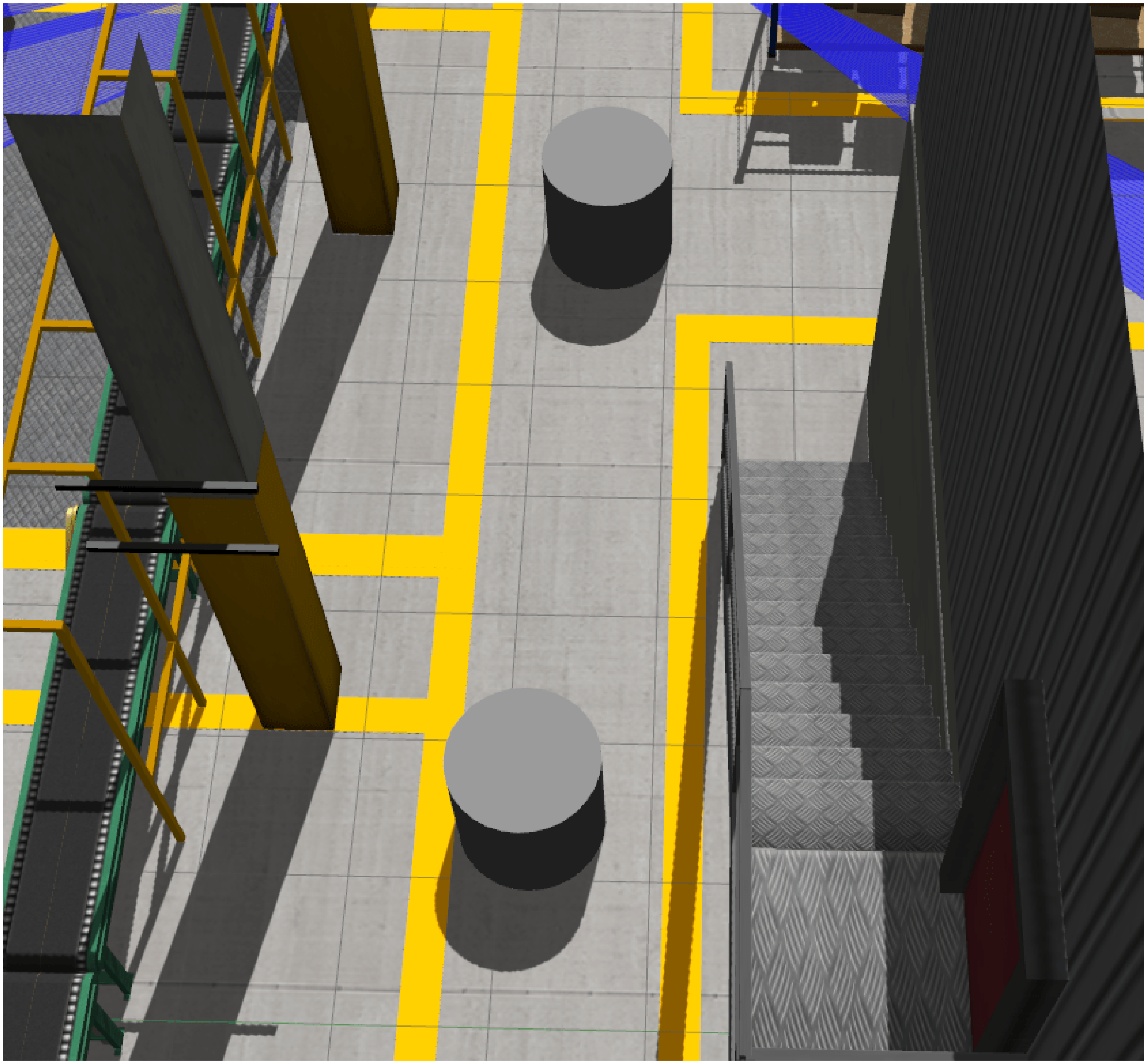
Run the simulation. The robot drives around the environment and avoids unexpected obstacles.

```
sim("aStarPathPlanningAndObstacleAvoidanceInWarehouse.slx");
```

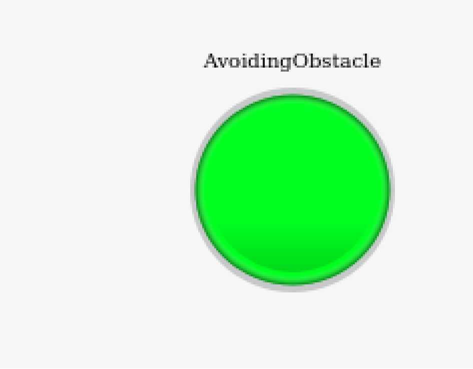

Binary Occupancy Grid
Unloading Station
Charging
Loading Station
Y [meters]
X [meters]
1
0
30
20
10
0
-10
-10
0
10
20
30



Notice that there are a two cylindrical obstacles which are not present on the occupancy map. The robot still avoids them when detected using the VFH algorithm.



A green lamp AvoidingObstacle lights up when the robot is trying to avoid an obstacle.


AvoidingObstacle

Design Position Controlled Manipulator Using Simscape

This example shows you how to use Simulink® with Robotics System Toolbox™ to design a position controller for a manipulator and compute joint position required to drive the Simscape™ Multibody™ model of the manipulator.

Introduction

In this example, you will load an included robot model using `loadrobot` as a `rigidBodyTree` object, then create a Simscape Multibody model of the robot using `smimport`. Configure the model to accept joint torque and return the computed joint position and velocity. Implement a computed torque controller with joint position and velocity feedback using manipulator algorithm blocks. The controller receives joint position and velocity information from the robot model and sends torque commands to drive the robot to the desired joint position computed using Inverse Kinematics (IK).

Load Robot Model in Workspace

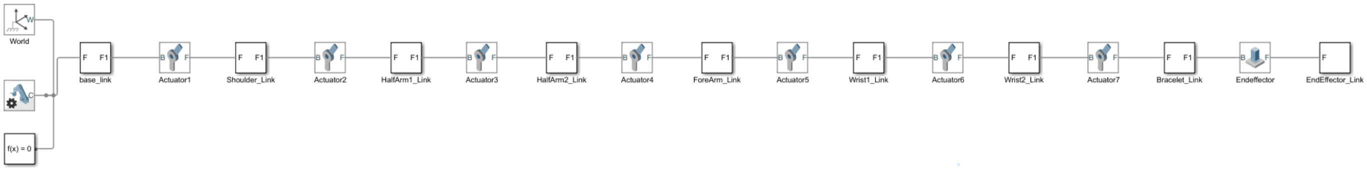
This example uses a model of the KINOVA® Gen3, a 7 degree-of-freedom robot manipulator. Call `loadrobot` to generate a `rigidBodyTree` model of the robot. Set the `DataFormat` properties to be consistent with Simscape.

```
robot = loadrobot("kinovaGen3",DataFormat="column");
```

Generate Simscape Multibody Model from Rigid Body Tree

Import the `robot` object into Simscape Multibody and get the model parameters.

```
robotSM = smimport(robot,ModelName="ManipulatorPositionControl_Subsystem");
sm_mdl = get_param(robotSM,"Name");
```

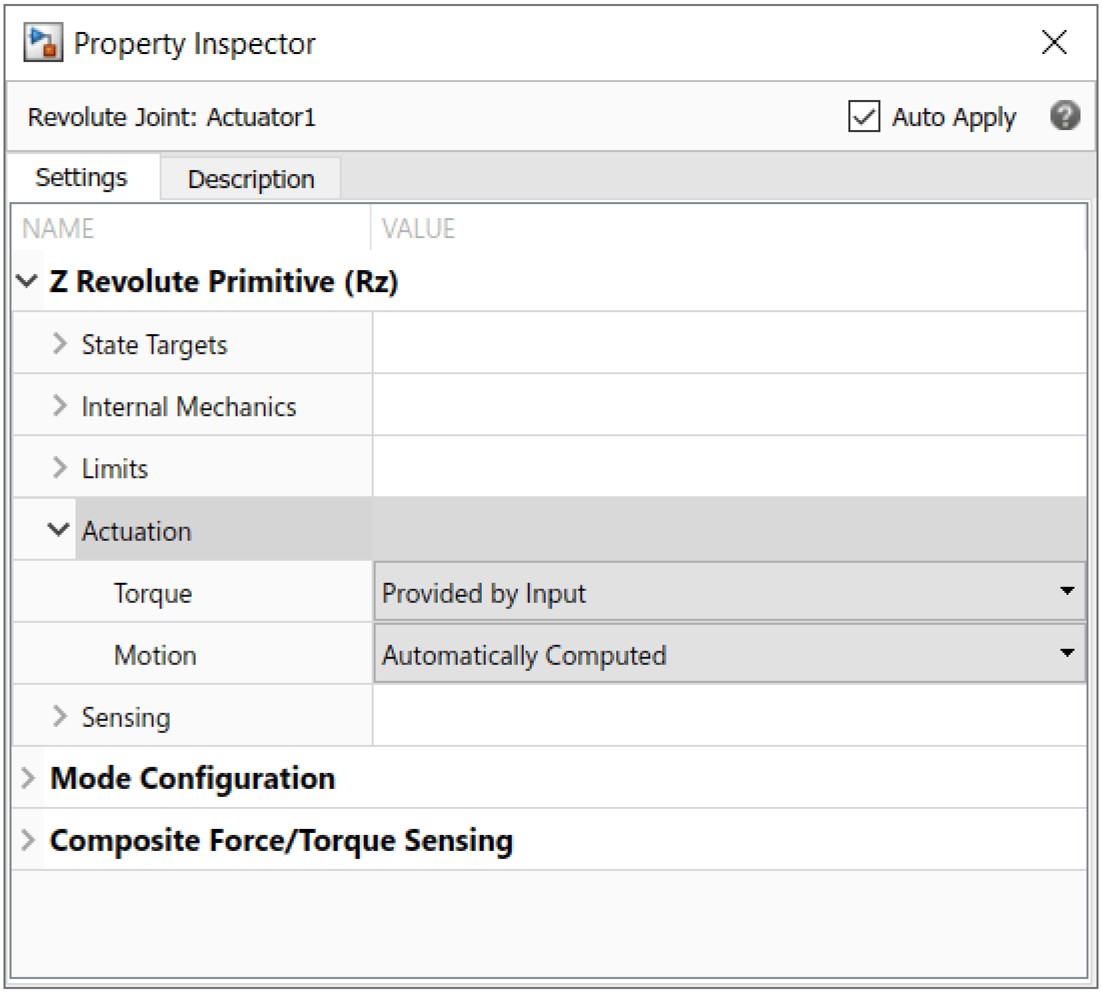


Configure Simscape Multibody Model

Prepare the Simscape Multibody model to accept the joint torque inputs and return the joint positions and velocities. You can follow the steps below to manually configure the model or use the `helperInstrumentSMModels` helper function to automatically configure the model.

Manual Configuration of Simscape Multibody Model

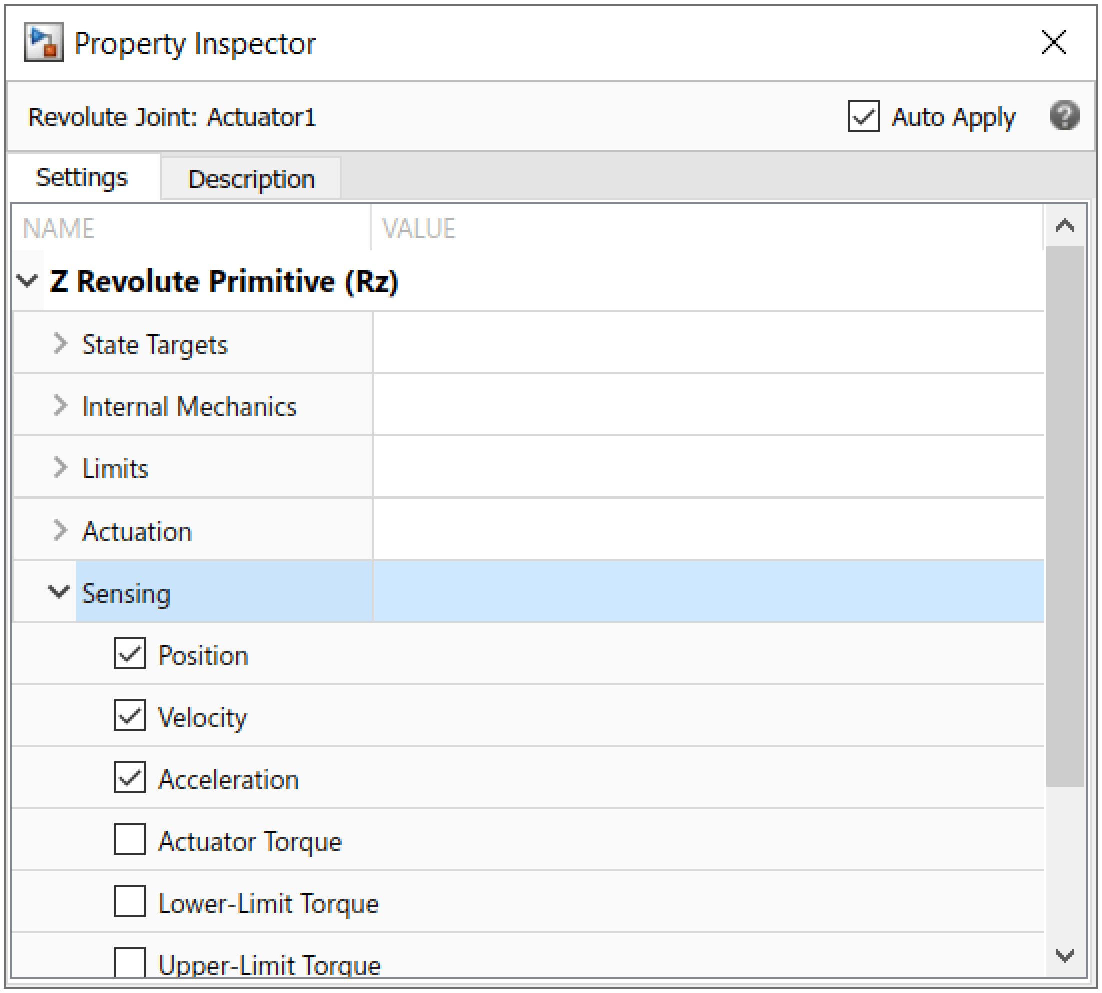
1. In your model, double-click a Joint block. The Property Inspector dialog box opens.
2. In the Property Inspector dialog box, select **Z Revolute Primitive (Rz)** > **Actuation** > **Torque > Provided by Input**, and select **Z Revolute Primitive (Rz)** > **Actuation** > **Motion > Automatically computed**. The block exposes a physical signal input port, labeled `t`.



Torque: Provided by Input

Motion: Automatically Computed

3. Select **Z Revolute Primitive (Rz)** > **Sensing** and enable **Position**, **Velocity**, and **Acceleration**. The block exposes a physical signal output ports, labeled q, w, and b.



Revolute Joint: Actuator1 ☑ Auto Apply

Z Revolute Primitive (Rz)

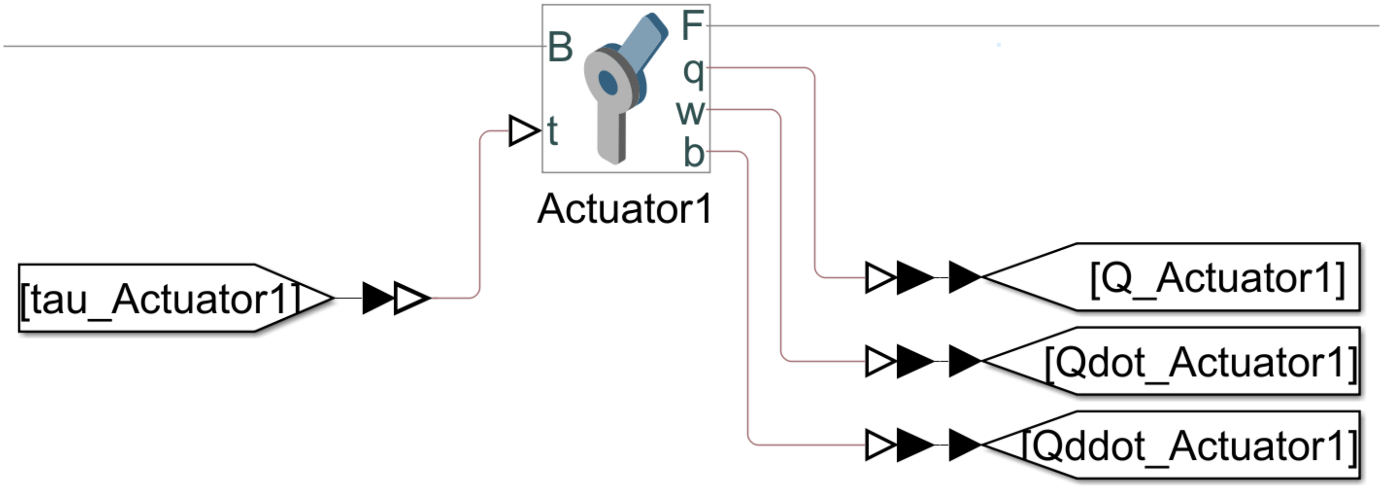
- State Targets
- Internal Mechanics
- Limits
- Actuation
- Sensing
  - ☑ Position
  - ☑ Velocity
  - ☑ Acceleration
  - □ Actuator Torque
  - □ Lower-Limit Torque
  - □ Upper-Limit Torque

4. Add a **Simulink-PS Converter** block from the **Simscape** > **Utilities** library, connect the Simulink-PS Converter block to physical signal input port `t` of the Joint block.

5. Add a **From** block from the **Simulink** > **Signal Routing** library to the input port of the Simulink-PS Converter block.

6. Add three **PS-Simulink Converter** blocks from the **Simscape** > **Utilities** library, connect the PS-Simulink Converter blocks to physical signal output ports `q`, `w`, and `b` of the Joint block.

7. Add three **Goto** blocks from the **Simulink** > **Signal Routing** library to the output port of the PS-Simulink Converter blocks.



8. Repeat these steps for all the Joint blocks.

9. Add a **Demux** block from the **Simulink** > **Signal Routing** library and connect the joint torque **Goto** blocks related to the respective joint torque **From** blocks.

10. Add three **Mux** blocks from the **Simulink** > **Signal Routing** library and connect the joint motions **From** blocks related to the respective joint motions **Goto** blocks.

11. Create a subsystem of the Simscape Multibody model.

Configure Simscape Multibody Model Using Helper Function

Use the `helperInstrumentSMModels` helper function to automatically configure the model.

Call the helper function to automatically configure the Simscape Multibody model to accept torque input.

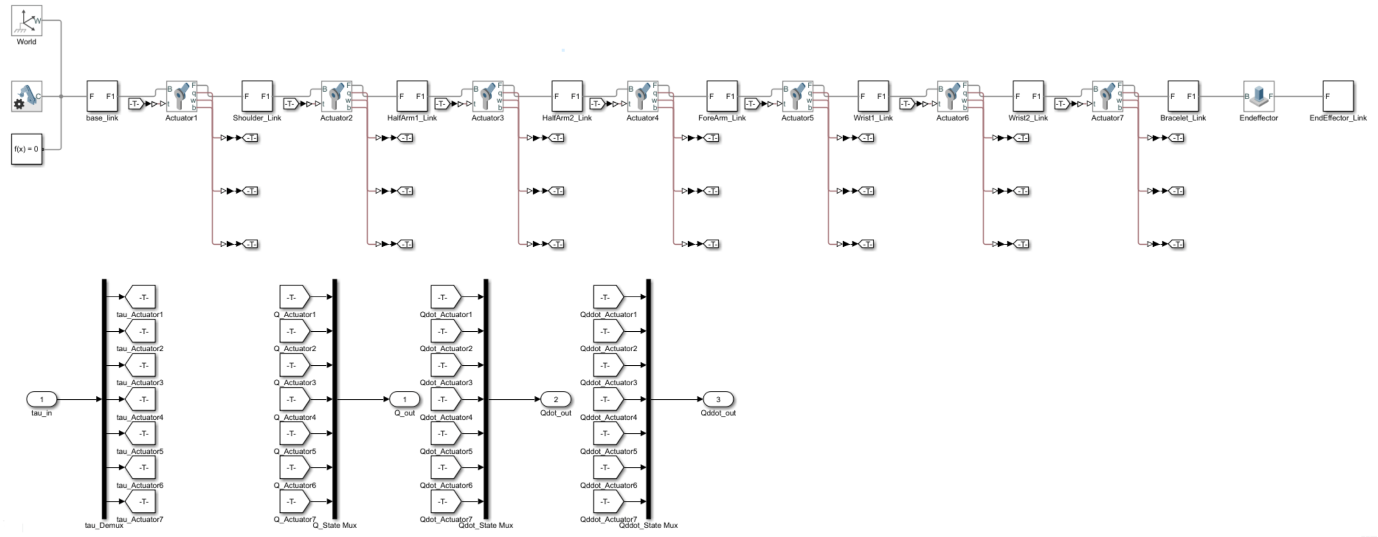
```
helperInstrumentSMModels.instrumentRBTSupportedJointInputs(sm_mdl,robot,"torque")
```

Call the helper function again to configure the Simscape Multibody model to enable position, velocity, and acceleration sensing at each joint.

```
helperInstrumentSMModels.instrumentRBTSupportedJointOutputs(sm_mdl,robot,"motion")
```

Create a subsystem of the Simscape Multibody model.

```
helperInstrumentSMModels.convertToSubsystem(sm_mdl)
```



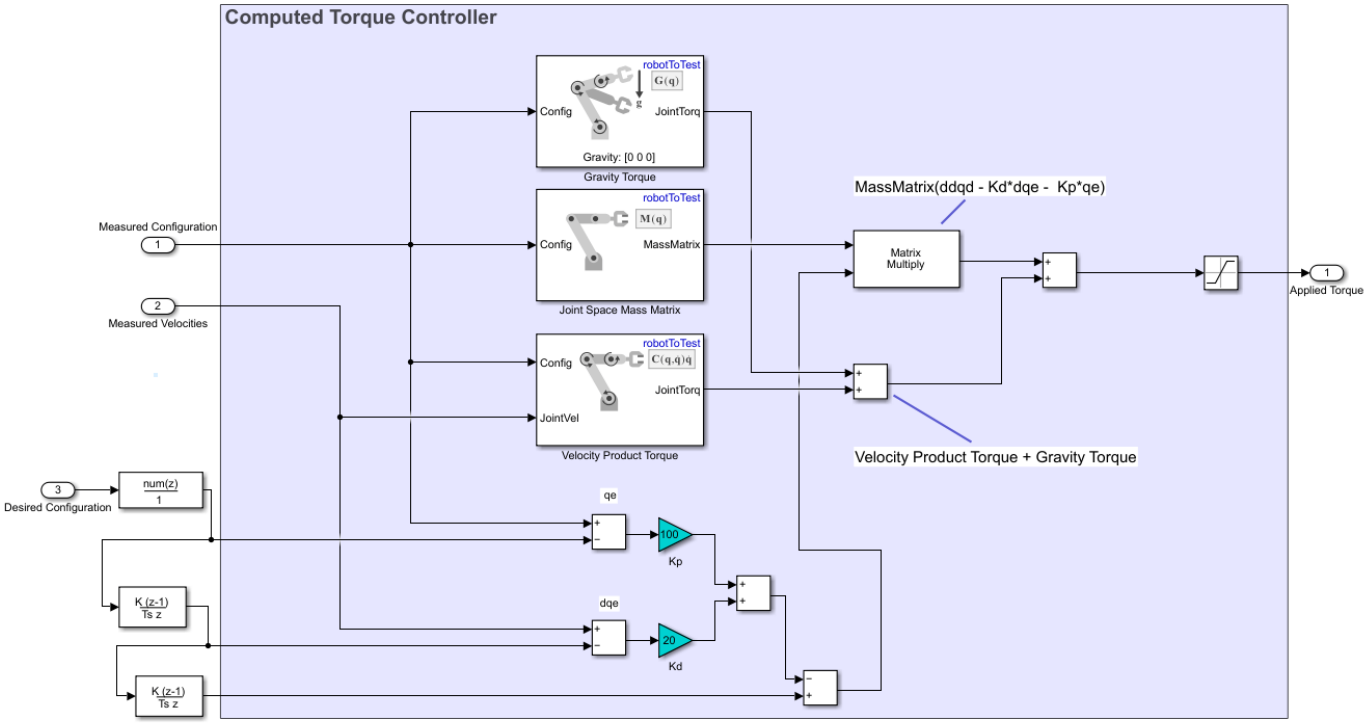
Set Up Variables in Model Workspace

Set up the variables in the model workspace that specify the start and end waypoints, and the joint starting position and velocity.

```
mdlWks = get_param(robotSM,"ModelWorkspace");
assignin(mdlWks,"robotToTest",robot)
assignin(mdlWks,"q0",robot.homeConfiguration)
assignin(mdlWks,"dq0",zeros(size(robot.homeConfiguration)))
```

Computed Torque Controller

The **Computed Torque Controller** subsystem is built using three robotics manipulator blocks: **Joint Space Mass Matrix**, **Velocity Product Torque**, and **Gravity Torque**. The `rigidBodyTree` model, `robotToTest`, is assigned in all those blocks.

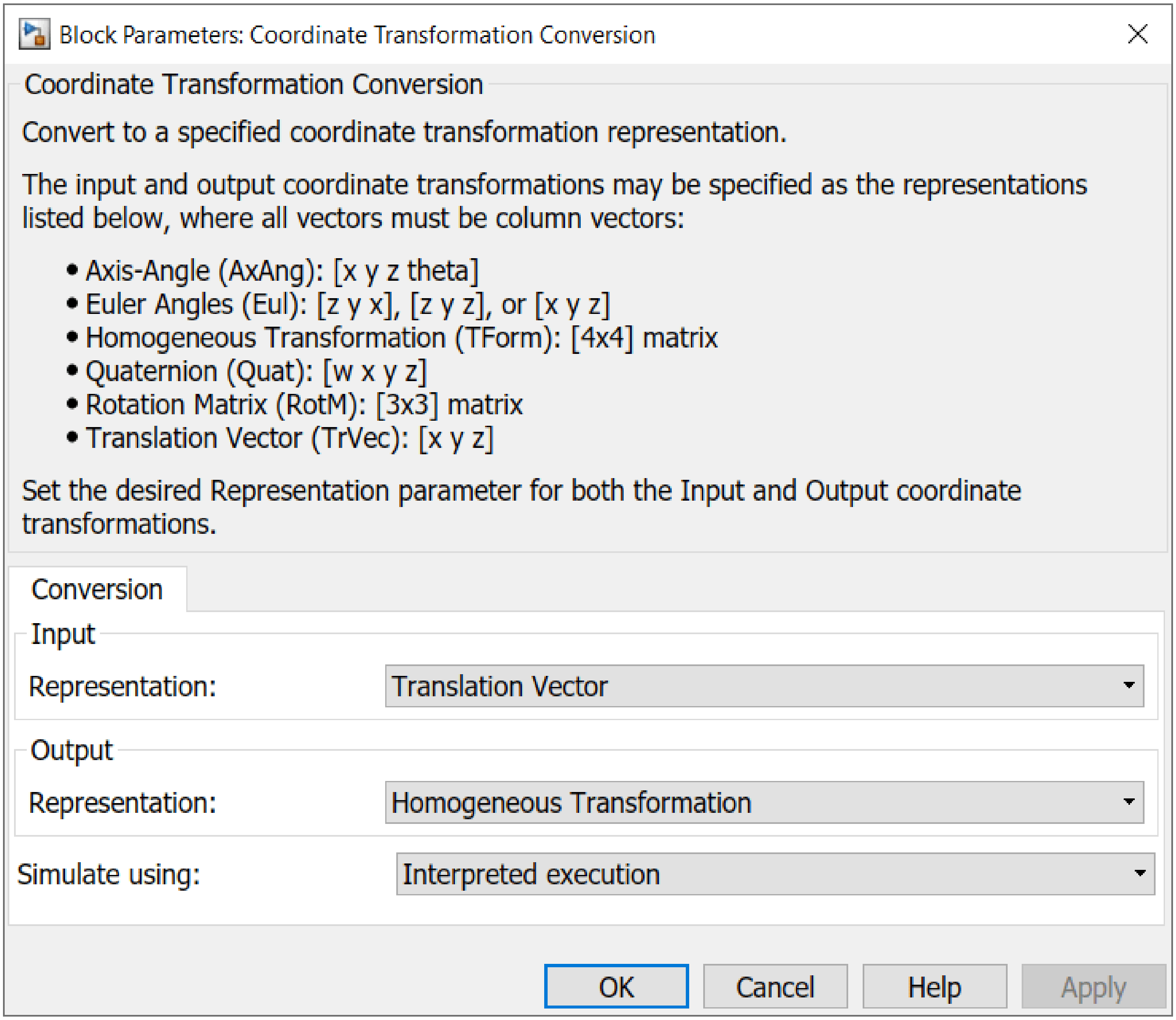


The Computed Torque Controller subsystem accepts **Measured Configuration**, **Measured Velocities** and **Desired Configuration** and returns **Applied Torque** for each joint of the manipulator.

For more details about this controller, see “Perform Safe Trajectory Tracking Control Using Robotics Manipulator Blocks” on page 1-268.

Set Up Controller Input

1. Add a **Coordinate Transformation Conversion** block from the **Robotics System Toolbox > Utilities** library to the model. Set the input representation as **Translation Vector** and the output representation as **Homogeneous Transformation**.



Block Parameters: Coordinate Transformation Conversion

Coordinate Transformation Conversion

Convert to a specified coordinate transformation representation.

The input and output coordinate transformations may be specified as the representations listed below, where all vectors must be column vectors:

- Axis-Angle (AxAng): [x y z theta]
- Euler Angles (Eul): [z y x], [z y z], or [x y z]
- Homogeneous Transformation (TForm): [4x4] matrix
- Quaternion (Quat): [w x y z]
- Rotation Matrix (RotM): [3x3] matrix
- Translation Vector (TrVec): [x y z]

Set the desired Representation parameter for both the Input and Output coordinate transformations.

Conversion

Input

Representation: Translation Vector

Output

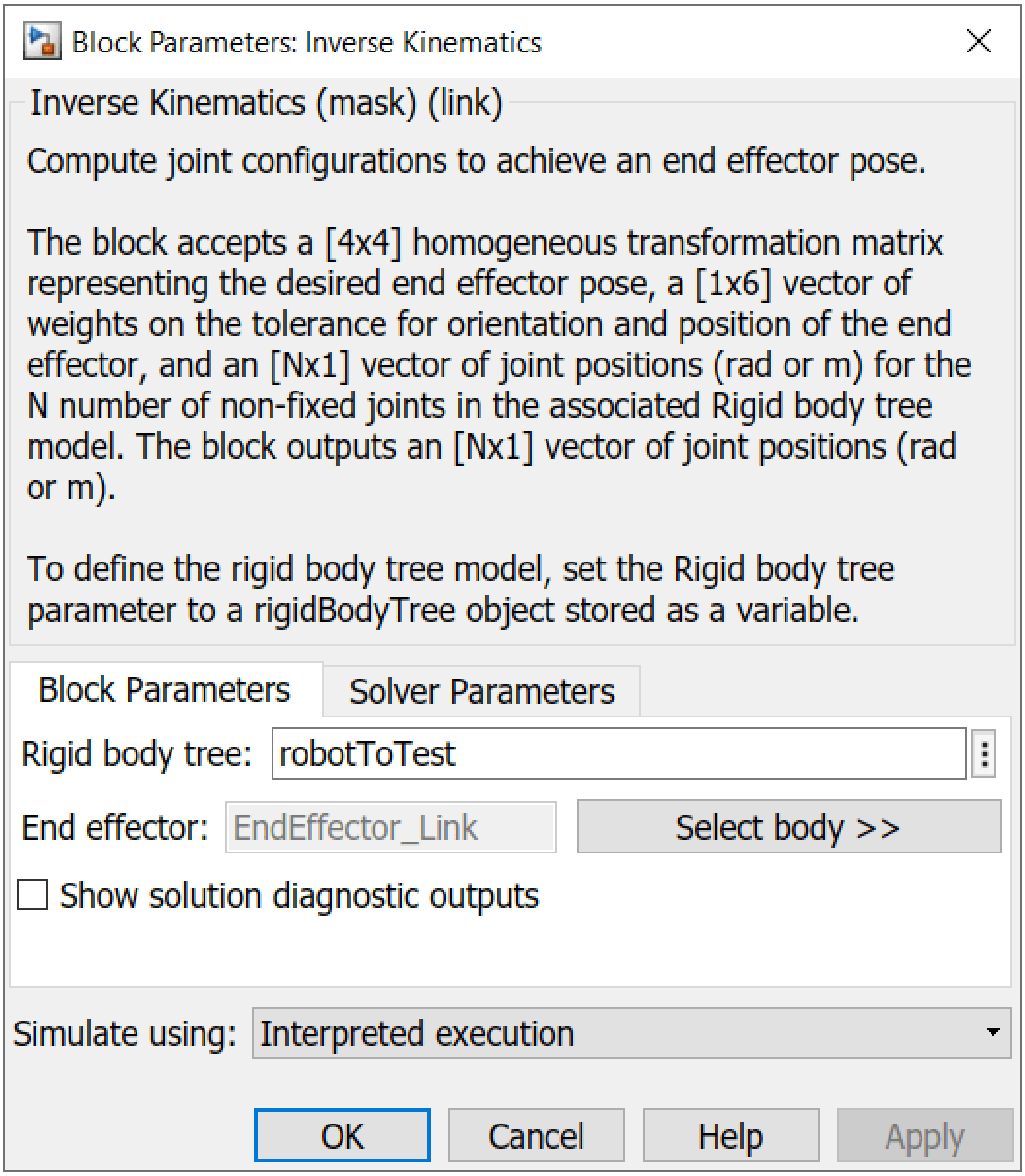
Representation: Homogeneous Transformation

Simulate using: Interpreted execution

2. Add a **Constant** block and set the value as `[0.5 0.5 0.5]`. Connect the Constant block to the input port of the Coordinate Transformation Conversion block.

3. Add a **Inverse Kinematics** block from the **Robotics System Toolbox > Manipulator Algorithms** library to the model.

4. In the **Inverse Kinematics** block, specify the **Rigid body tree** model as `robotToTest`, then click **Select body** next to the **End effector** to select the end effector body.



Block Parameters: Inverse Kinematics

Inverse Kinematics (mask) (link)

Compute joint configurations to achieve an end effector pose.

The block accepts a [4x4] homogeneous transformation matrix representing the desired end effector pose, a [1x6] vector of weights on the tolerance for orientation and position of the end effector, and an [Nx1] vector of joint positions (rad or m) for the N number of non-fixed joints in the associated Rigid body tree model. The block outputs an [Nx1] vector of joint positions (rad or m).

To define the rigid body tree model, set the Rigid body tree parameter to a rigidBodyTree object stored as a variable.

Rigid body tree: robotToTest

End effector: EndEffector_Link

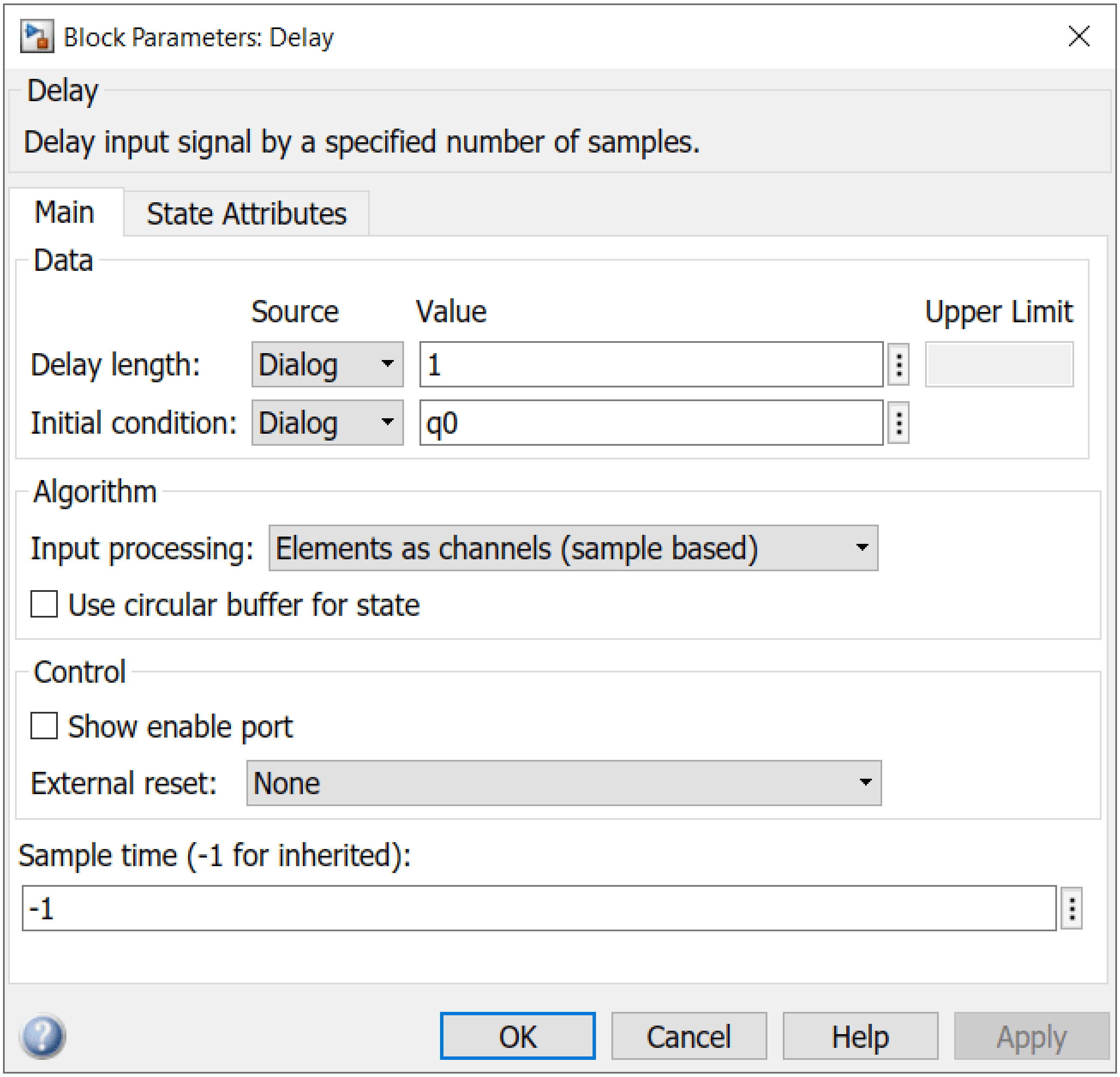
□ Show solution diagnostic outputs

Simulate using: Interpreted execution

5. Connect the output port of Coordinate Transformation Conversion block to the **Pose** port of the Inverse Kinematics block.

6. Add another **Constant** block and set the value as `[0 0 0 1 1 1]`. Connect the Constant block to the **Weights** port of the Inverse Kinematics block.

7. Connect a **Delay** block to the **Config** port of the Inverse Kinematics block and specify the **Initial condition** as `q0`.



Block Parameters: Delay

Delay

Delay input signal by a specified number of samples.

Data

	Source	Value	Upper Limit
Delay length:	Dialog	1	
Initial condition:	Dialog	q0	

Algorithm

Input processing: Elements as channels (sample based)

□ Use circular buffer for state

Control

□ Show enable port

External reset: None

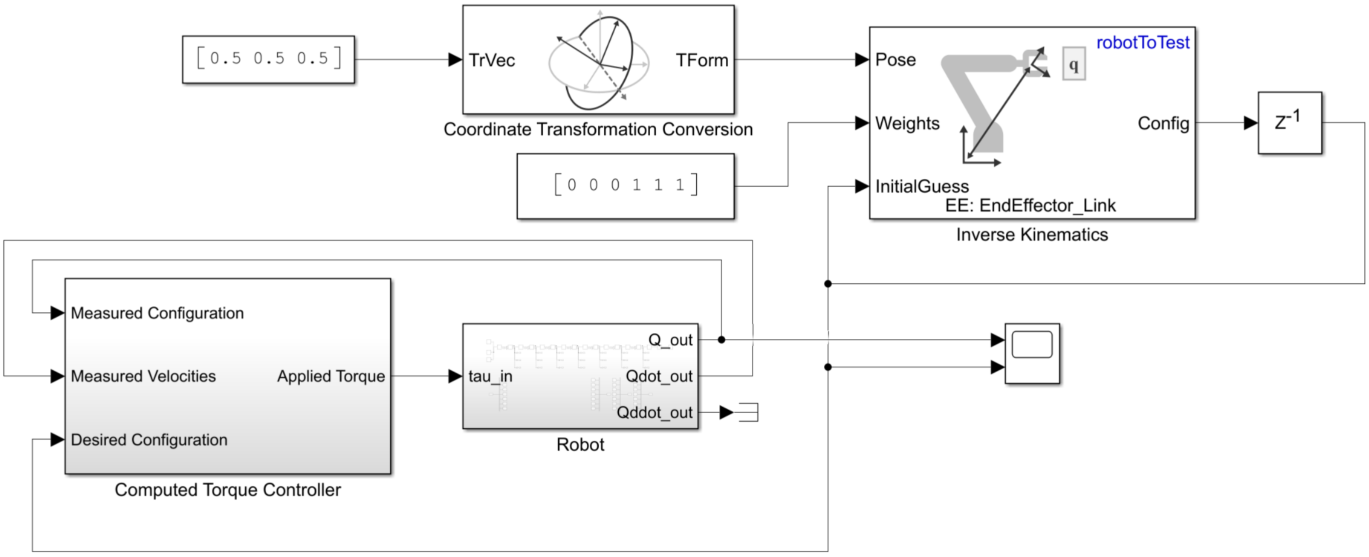
Sample time (-1 for inherited):

-1

8. Connect the output of the Delay block to the **InitialGuess** port of the Inverse Kinematics block.

Final Setup

Connect the Simscape Multibody Model subsystem, Computed Torque Controller subsystem, Controller Input blocks, and a **Scope** block as shown in figure.



Copyright 2021 The MathWorks, Inc.

Simulate Model

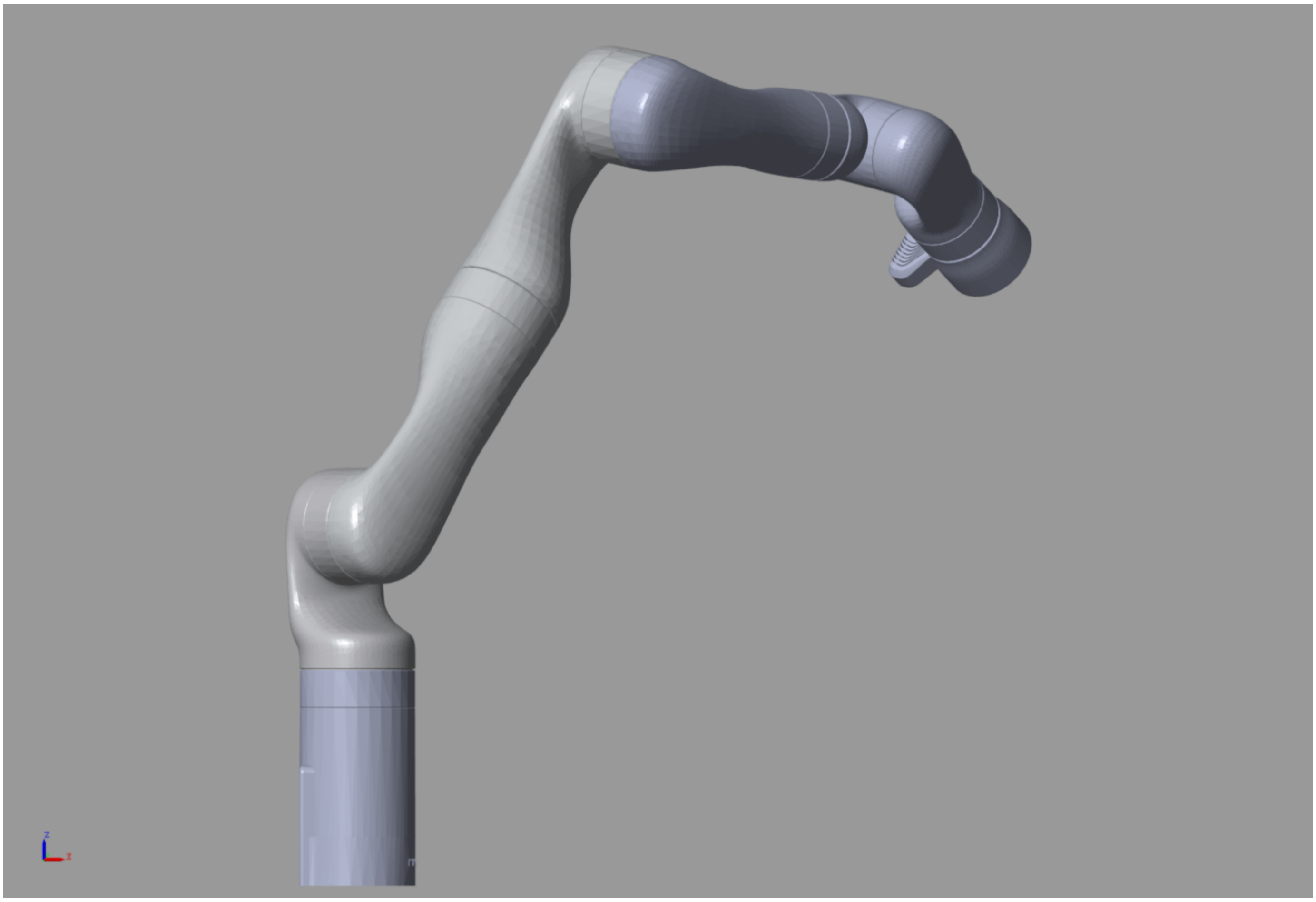
Open the provided **ManipulatorPositionControl.slx** model and replace the **Robot** subsystem with the subsystem created in **ManipulatorPositionControl_Subsystem** model above, for it to be able to fetch the meshes correctly.

```
open_system("ManipulatorPositionControl.slx")
```

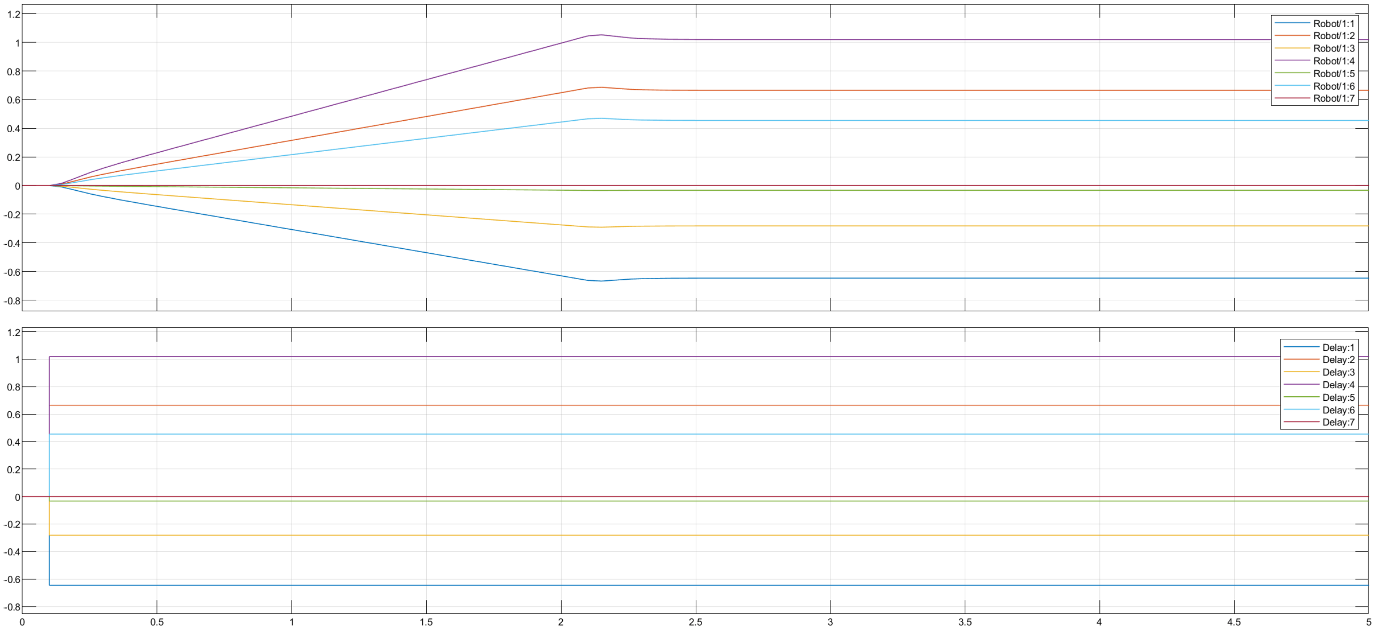
Save the model and simulate it.

```
sim("ManipulatorPositionControl.slx","StopTime","5")
```

Visualize the multibody model in the Mechanics Explorer.



Visualize the joint positions in the Scope.


1.2
1
0.8
0.6
0.4
0.2
0
-0.2
-0.4
-0.6
-0.8
Robot/1:1
Robot/1:2
Robot/1:3
Robot/1:4
Robot/1:5
Robot/1:6
Robot/1:7
Delay:1
Delay:2
Delay:3
Delay:4
Delay:5
Delay:6
Delay:7
0
0.5
1
1.5
2
2.5
3
3.5
4
4.5
5

Perform Trajectory Tracking and Compute Joint Torque for Manipulator Using Simscape

This example shows you how to use Simulink® with Robotics System Toolbox™ to perform trajectory tracking and compute joint torque required to drive the Simscape™ Multibody™ model of the manipulator along the given joint trajectory. This example can be further extended to scenarios with obstacles to observe the changes in torque profile.

Introduction

In this example, you will load an included robot model using `loadrobot` as a `rigidBodyTree` object, then create a Simscape Multibody model of the robot using `smimport`. Configure the model to accept motion inputs such as joint position, velocity, and acceleration generated using the **Trapezoidal Velocity Profile Trajectory** block and return the computed joint torque. Simulate the model to visualize the robot motion and plot the joint torques.

Load Robot Model in Workspace

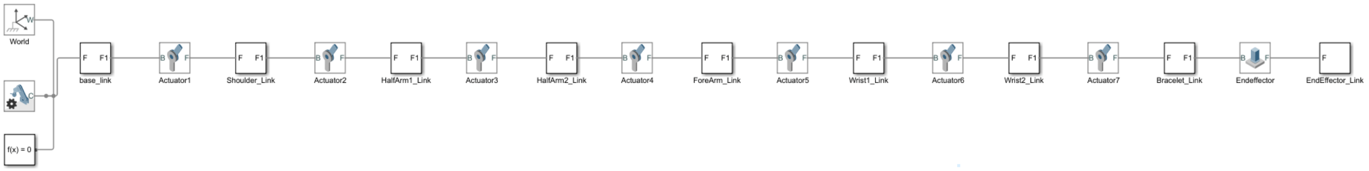
This example uses a model of the KINOVA® Gen3, a 7 degree-of-freedom robot manipulator. Call `loadrobot` to generate a `rigidBodyTree` model of the robot. Set the `DataFormat` properties to be consistent with Simscape.

```
robot = loadrobot("kinovaGen3",DataFormat="column");
```

Generate Simscape Multibody Model from Rigid Body Tree

Import the `robot` object into Simscape Multibody and get the model parameters.

```
robotSM = smimport(robot,ModelName="ManipulatorTrajectoryPlanning_Subsystem");
sm_mdl = get_param(robotSM,"Name");
```

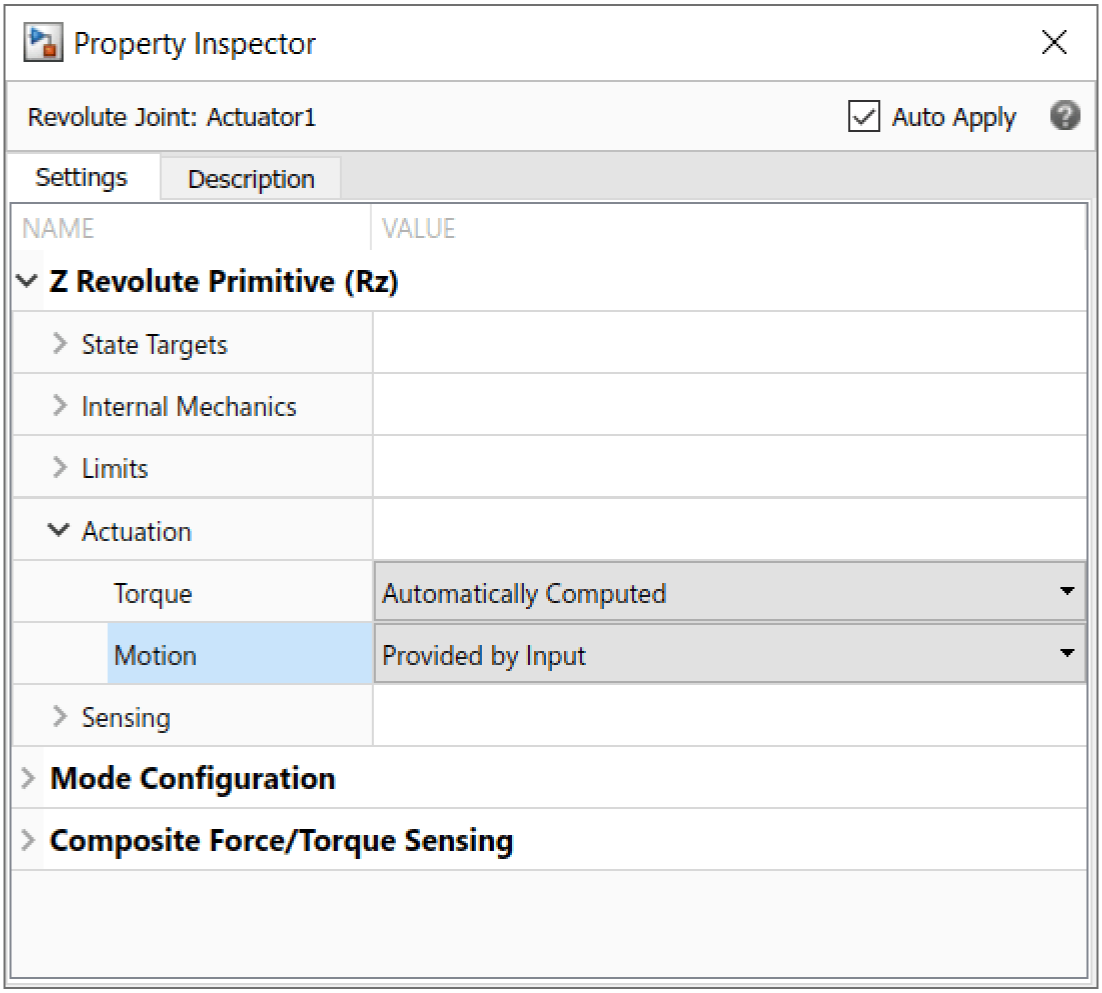


Configure Simscape Multibody Model

Prepare the Simscape Multibody model to accept the joint motion inputs and return the joint torques. You can follow the steps below to manually configure the model or use the `helperInstrumentSMModels` helper function to automatically configure the model.

Manual Configuration of Simscape Multibody Model

1. In your model, double-click a Joint block. The Property Inspector dialog box opens.
2. In the Property Inspector dialog box, select **Z Revolute Primitive (Rz)** > **Actuation** > **Torque > Automatically computed**, and select **Z Revolute Primitive (Rz)** > **Actuation** > **Motion > Provided by Input**. The block exposes a physical signal input port, labeled q.



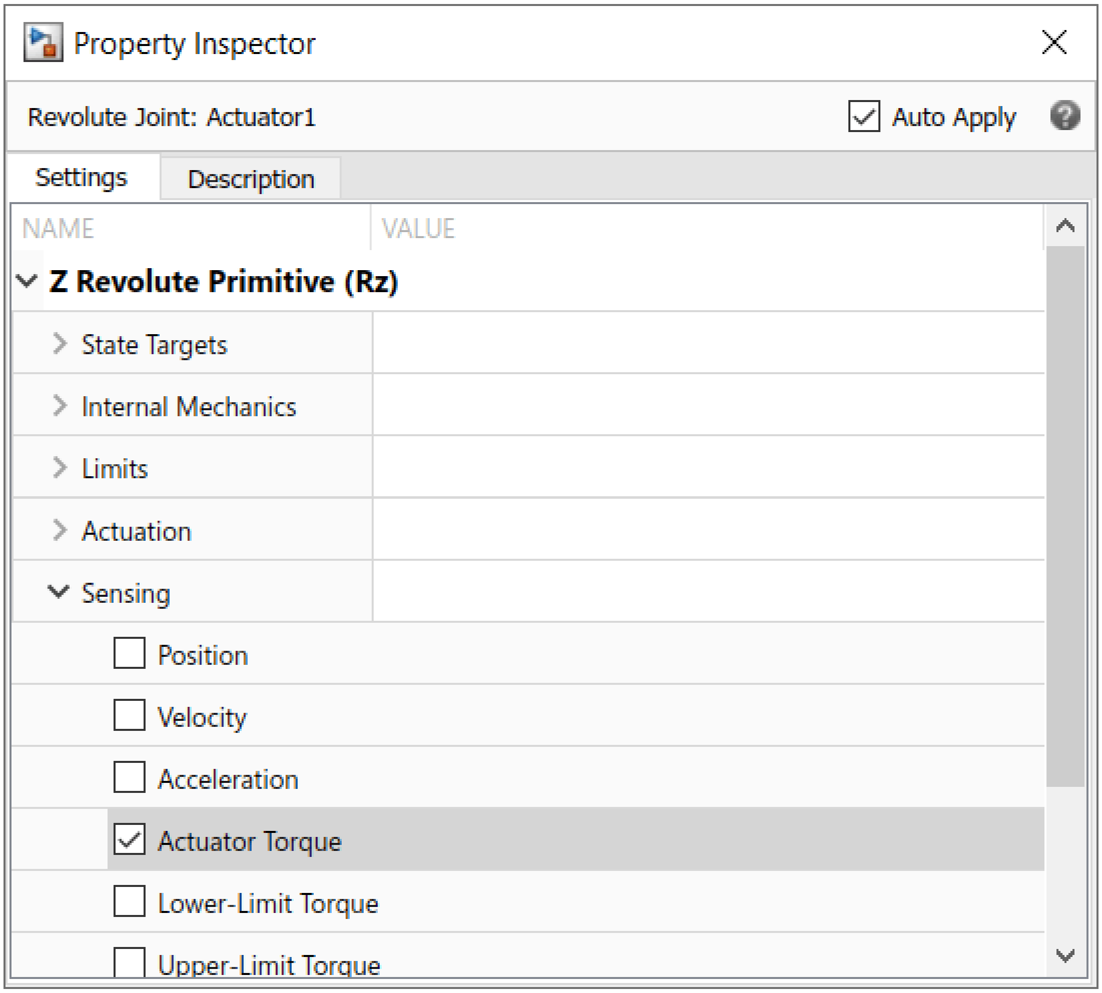
Property Inspector

Revolute Joint: Actuator1 ☑ Auto Apply

Settings | Description

NAME	VALUE
Z Revolute Primitive (Rz)	
State Targets	
Internal Mechanics	
Limits	
Actuation	
Torque	Automatically Computed
Motion	Provided by Input
Sensing	
Mode Configuration	
Composite Force/Torque Sensing	

3. In the Property Inspector block dialog box, select **Z Revolute Primitive (Rz)** > **Sensing** > **Actuator Torque**. The block exposes a physical signal output port, labeled `t`.



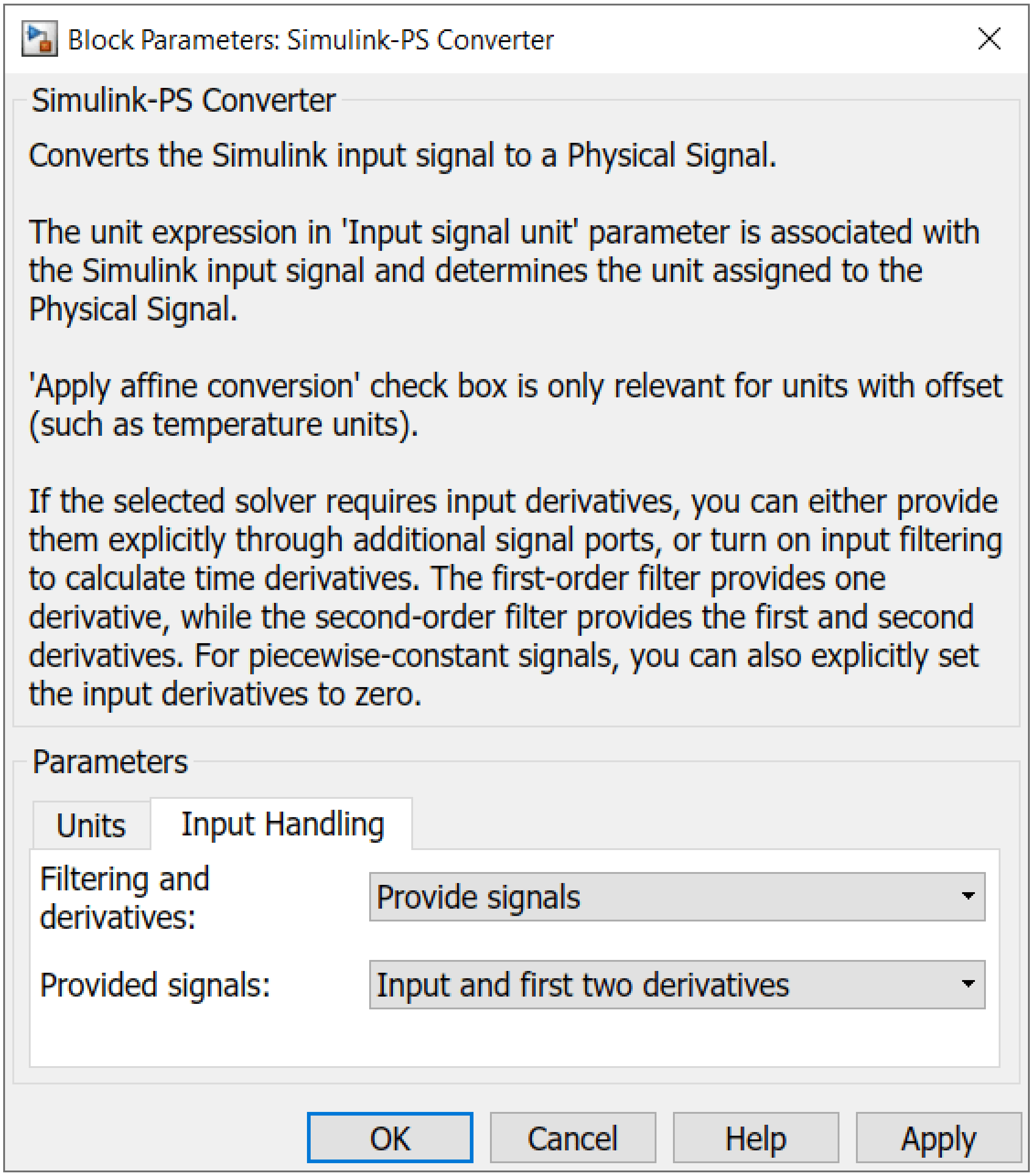
Revolute Joint: Actuator1 ☑ Auto Apply

Z Revolute Primitive (Rz)

- State Targets
- Internal Mechanics
- Limits
- Actuation
- Sensing
  - □ Position
  - □ Velocity
  - □ Acceleration
  - ☑ Actuator Torque
  - □ Lower-Limit Torque
  - □ Upper-Limit Torque

4. Add a **Simulink-PS Converter** block from the **Simscape** > **Utilities** library, connect the Simulink-PS Converter block to physical signal input port q of the Joint block.

5. Double-click the Simulink-PS Converter block to open the block dialog box, click **Input Handling** Tab, select **Provided signals > Input and first two derivatives**.


Block Parameters: Simulink-PS Converter

Simulink-PS Converter

Converts the Simulink input signal to a Physical Signal.

The unit expression in 'Input signal unit' parameter is associated with the Simulink input signal and determines the unit assigned to the Physical Signal.

'Apply affine conversion' check box is only relevant for units with offset (such as temperature units).

If the selected solver requires input derivatives, you can either provide them explicitly through additional signal ports, or turn on input filtering to calculate time derivatives. The first-order filter provides one derivative, while the second-order filter provides the first and second derivatives. For piecewise-constant signals, you can also explicitly set the input derivatives to zero.

Parameters

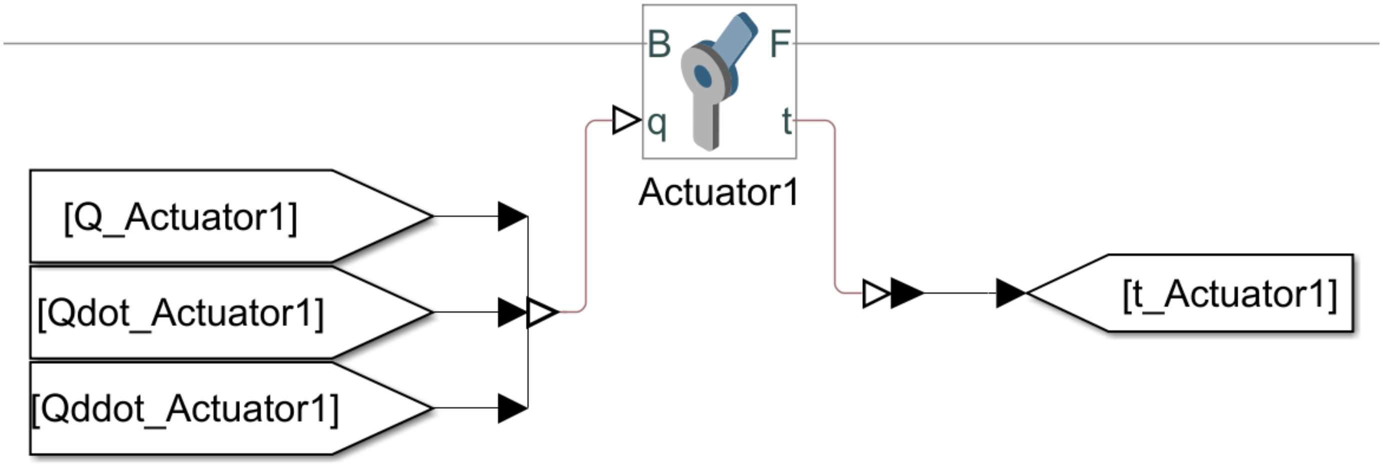
Filtering and derivatives: Provide signals

Provided signals: Input and first two derivatives

6. Add three **From** block from the **Simulink** > **Signal Routing** library to the three input ports of the Simulink-PS Converter block.

7. Add **PS-Simulink Converter** block from the **Simscape** > **Utilities** library, connect the PS-Simulink Converter block to physical signal output port `t` of the Joint block.

8. Add **Goto** block from the **Simulink** > **Signal Routing** library to the output port of the PS-Simulink Converter block.



9. Repeat these steps for all the Joint blocks.

10. Add three **Demux** blocks from the **Simulink** > **Signal Routing** library and connect the joint motions **Goto** blocks related to the respective joint motions **From** blocks.

11. Add a **Mux** block from the **Simulink** > **Signal Routing** library and connect the joint torque **From** blocks related to the respective joint torque **Goto** blocks.

12. Create a subsystem of the Simscape Multibody model.

Configure Simscape Multibody Model Using Helper Function

Use the `helperInstrumentSMModels` helper function to automatically configure the model.

Call the helper function to automatically configure the Simscape Multibody model to accept motion inputs such as joint position, velocity, and acceleration.

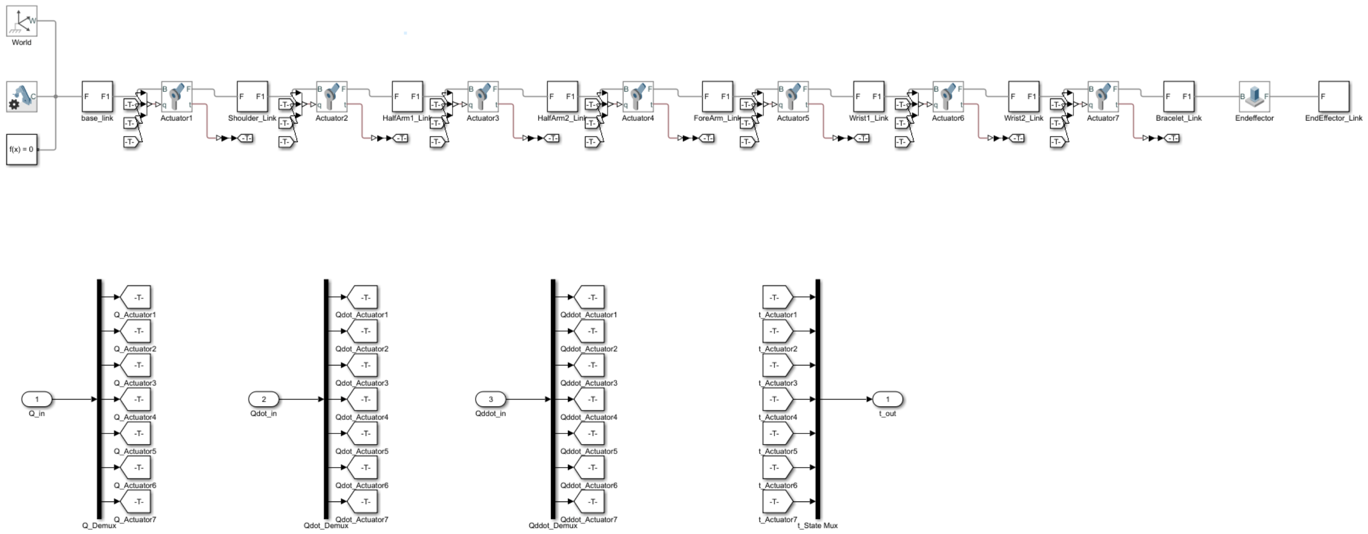
```
helperInstrumentSMModels.instrumentRBTSupportedJointInputs(sm_mdl,robot,"motion")
```

Call the helper function again to configure the Simscape Multibody model to attach torque sensors that measure torque at each joint.

```
helperInstrumentSMModels.instrumentRBTSupportedJointOutputs(sm_mdl,robot,"torque")
```

Create a subsystem of the Simscape Multibody model.

```
helperInstrumentSMModels.convertToSubsystem(sm_mdl)
```



Set Up Variables in Model Workspace

Set up the variables in the model workspace that specify the start and end waypoints, and the joint starting position and velocity.

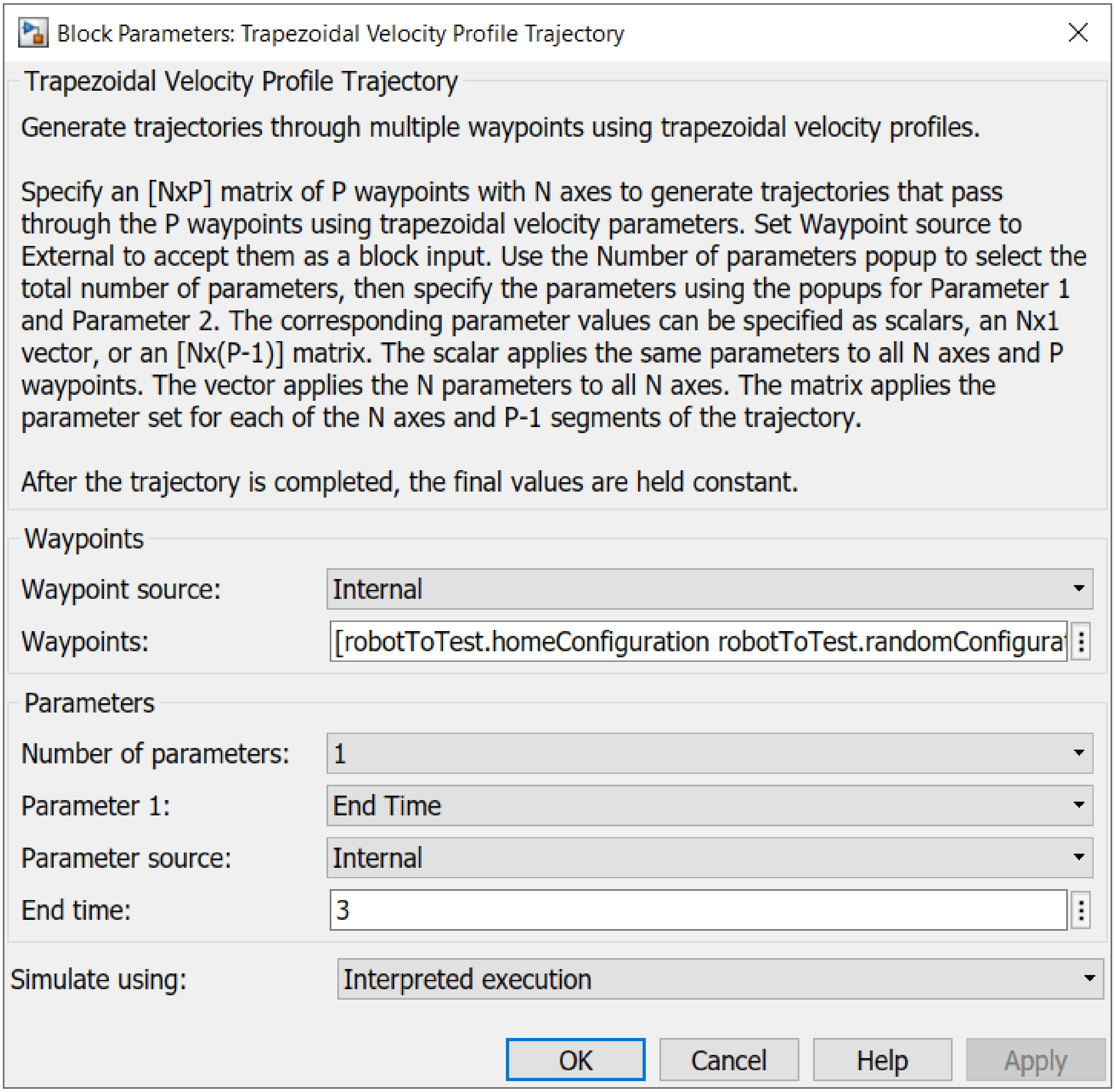
```
mdlWks = get_param(robotSM,'ModelWorkspace');
assignin(mdlWks,"robotToTest",robot)
assignin(mdlWks,"q0",robot.homeConfiguration)
assignin(mdlWks,"dq0",zeros(size(robot.homeConfiguration)))
```

Trajectory Generation

Add a **Trapezoidal Velocity Profile Trajectory** block from the **Robotics System Toolbox > Utilities** library to the model and connect the `q`, `qd`, and `qdd` ports of the block to the `Q_in`, `Qdot_in`, and `Qddot_in` input ports of the Simscape Multibody model subsystem.

In the Trapezoidal Velocity Profile Trajectory block parameters dialog box,

1. Set the **Waypoints** as `[robotToTest.homeConfiguration robotToTest.randomConfiguration robotToTest.randomConfiguration]`.
2. Set the **Number of parameters** to `1` and set the **Parameter 1** as **End Time**.
3. Specify the **End Time** as `3`**.**



Block Parameters: Trapezoidal Velocity Profile Trajectory

Trapezoidal Velocity Profile Trajectory

Generate trajectories through multiple waypoints using trapezoidal velocity profiles.

Specify an [NxP] matrix of P waypoints with N axes to generate trajectories that pass through the P waypoints using trapezoidal velocity parameters. Set Waypoint source to External to accept them as a block input. Use the Number of parameters popup to select the total number of parameters, then specify the parameters using the popups for Parameter 1 and Parameter 2. The corresponding parameter values can be specified as scalars, an Nx1 vector, or an [Nx(P-1)] matrix. The scalar applies the same parameters to all N axes and P waypoints. The vector applies the N parameters to all N axes. The matrix applies the parameter set for each of the N axes and P-1 segments of the trajectory.

After the trajectory is completed, the final values are held constant.

Waypoints

Waypoint source: Internal

Waypoints: [robotToTest.homeConfiguration robotToTest.randomConfigura

Parameters

Number of parameters: 1

Parameter 1: End Time

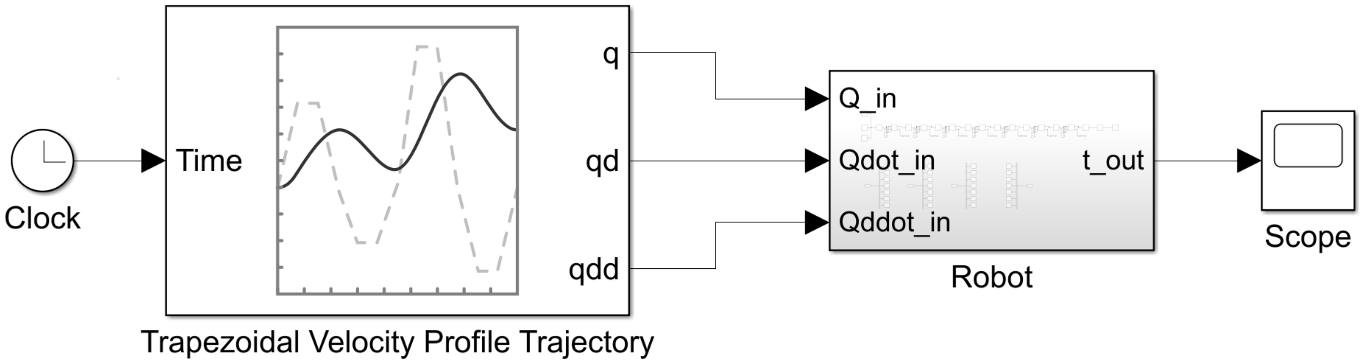
Parameter source: Internal

End time: 3

Simulate using: Interpreted execution

Final Setup

Add a **Clock** block from the **Simulink > Sources** library, connect the Clock block to the `Time` input port of the **Trapezoidal Velocity Profile Trajectory** block. Add a **Scope** block and connect the `t_out` port of the Simscape Multibody model subsystem to the Scope input.



Copyright 2021 The MathWorks, Inc.

Simulate Model

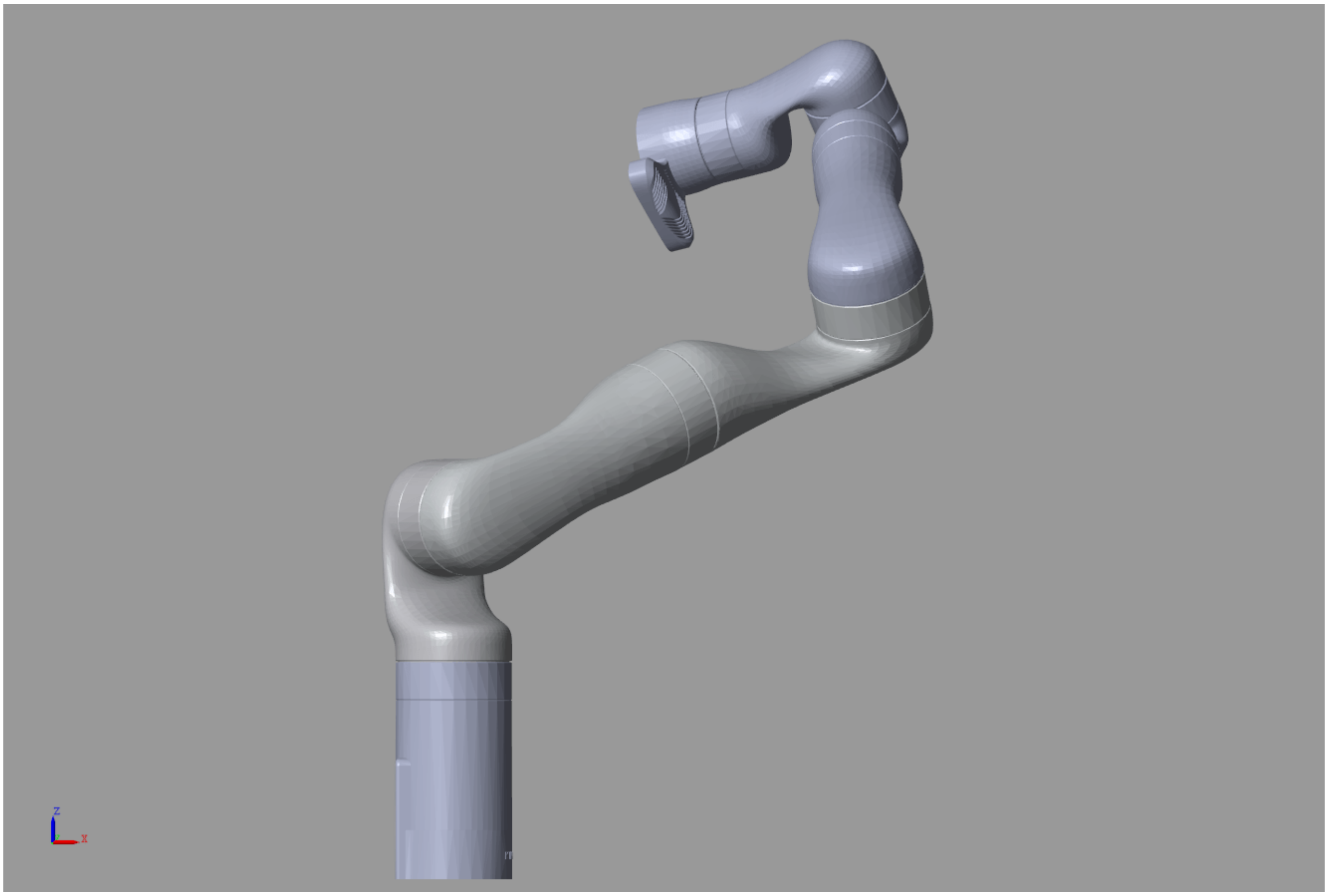
Open the provided **ManipulatorTrajectoryPlanning.slx** model and replace the **Robot** subsystem with the subsystem created in **ManipulatorTrajectoryPlanning_Subsystem** model above, for it to be able to fetch the meshes correctly.

```
open_system("ManipulatorTrajectoryPlanning.slx")
```

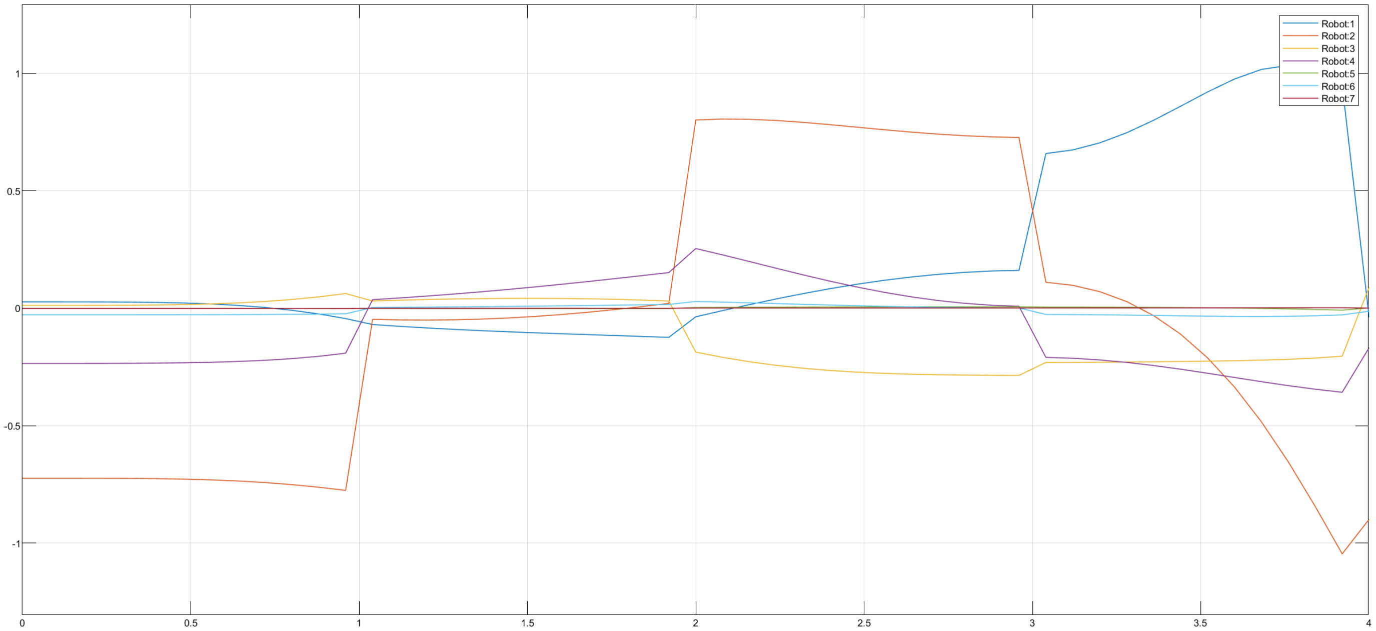
Save the model and simulate it. The **Trapezoidal Velocity Profile Trajectory** block generates random trajectories and drives the manipulator model accordingly. The manipulator model computes the joint torque required to execute the motion.

```
sim("ManipulatorTrajectoryPlanning.slx","StopTime","4")
```

Visualize the multibody model in the Mechanics Explorer.



Visualize the joint torques in the Scope.


Robot:1
Robot:2
Robot:3
Robot:4
Robot:5
Robot:6
Robot:7
1
0.5
0
-0.5
-1
0
0.5
1
1.5
2
2.5
3
3.5
4

Choose Trajectories for Manipulator Paths

This example provides an overview of the types of trajectories available in Robotics System Toolbox™. For manipulator motion, planning, and control applications, you must choose a trajectory for the robot to follow. There are three main sections of this example. The first section shows the types of trajectories that manipulators use, the second section demonstrates functions for generating trajectories, and the final section shows more tools for trajectory planning.

Trajectory Types for Manipulators

When using trajectories with manipulators, the goal is to provide a feasible result subject to certain constraints. For example, you often want a trajectory with smooth and continuous derivatives, such as one that does not require instantaneous velocity or acceleration. The trajectory may also be subject to position, velocity, or acceleration bounds. There are two general ways to use trajectories in the joint space and in the task space**.**

Joint-Space Trajectories

A *joint-space trajectory* typically contains a set of waypoints of multiple robot configurations connected using smooth motion. This example generates a trajectory using a trapezoidal velocity profile, which progressively speeds up each joint to a maximum velocity and slows it down as it approaches the next waypoint. For more information, see Trapezoidal Velocity Profile Trajectory on page 1-0 .

Generate a trapezoidal velocity profile trajectory to connect waypoints for a Franka Emika Panda™ robot. First, define the time vector for the trajectories and load the robot model.

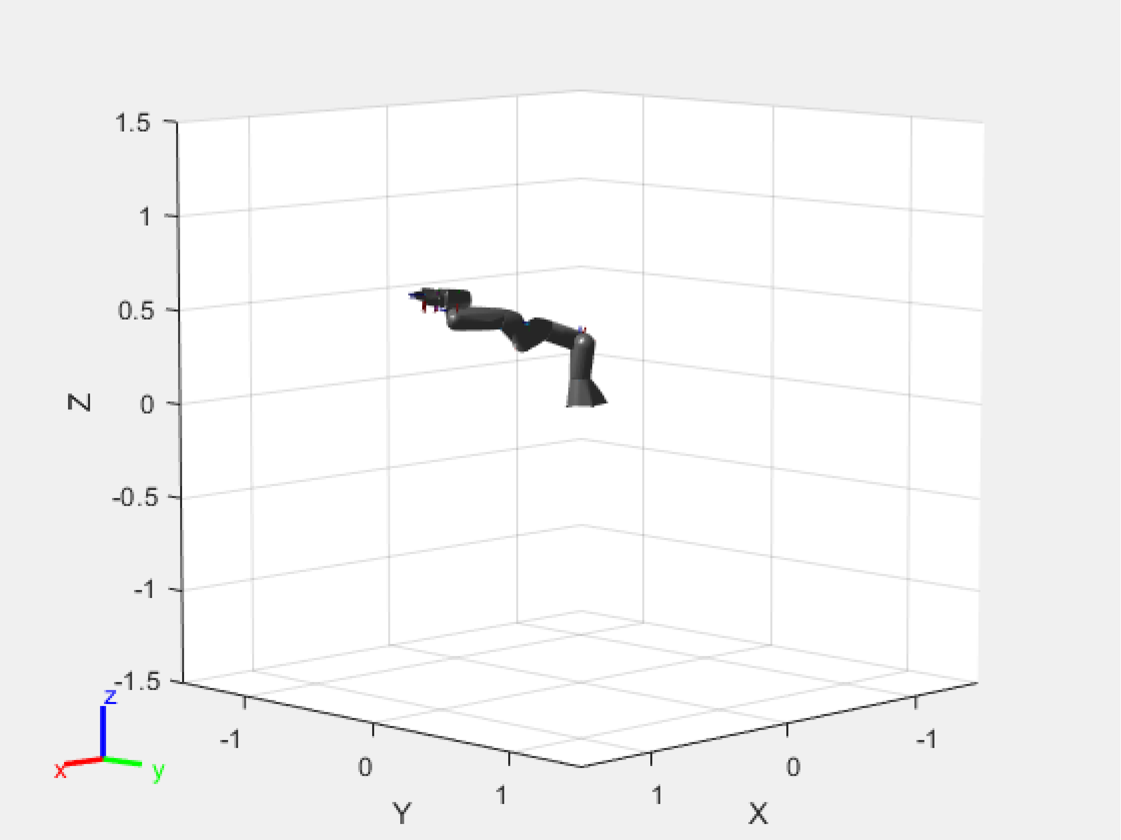
```
tpts = 0:4;
sampleRate = 20;
tvec = tpts(1):1/sampleRate:tpts(end);
numSamples = length(tvec);

robot = loadrobot('frankaEmikaPanda',DataFormat='column');
```

Define the joint-space trajectory. For this trajectory, the waypoints are the home configuration for the model and two random configurations.

```
rng default
frankaWaypoints = [robot.homeConfiguration robot.randomConfiguration robot.randomConfiguration];
frankaTimepoints = linspace(tvec(1),tvec(end),3);
[q,qd] = trapveltraj(frankaWaypoints,numSamples);
```

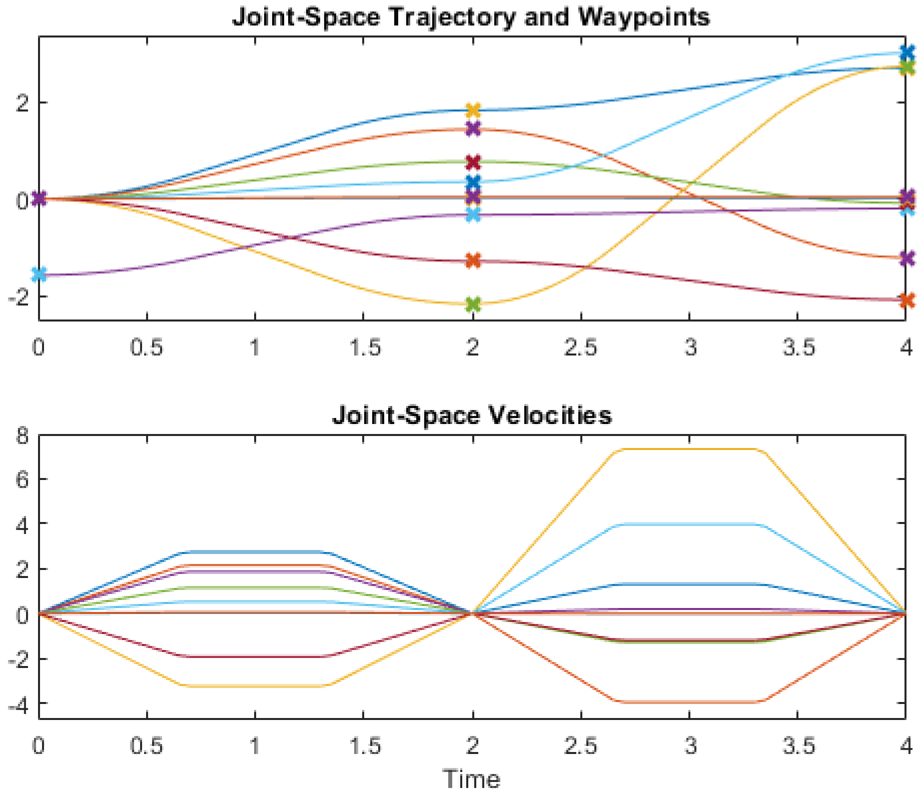
Visualize the robot executing the trajectory by iterating through the generated trajectory points `q`.

```
figure
set(gcf,'Visible','on');
rc = rateControl(sampleRate);
for i = 1:numSamples
    show(robot,q(:,i),FastUpdate=true,PreservePlot=false);
    waitfor(rc);
end
```



To examine the different joint positions and velocities, plot all the dimensions against time. Use the `helperPlotJointSpaceTraj` helper function to plot the joint-space trajectory and its waypoints.

```
helperPlotJointSpaceTraj('Joint-Space Trajectory and Waypoints', ...
    tvec,q,qd,frankaWaypoints,frankaTimepoints);
```



Task-Space Trajectories

A *task-space trajectory* contains waypoints that represent end-effector motion in 3-D space. Generate a minimum-jerk trajectory to connect waypoints in free space. The purpose of this trajectory profile is to create a smooth trajectory with minimal jerky motion. For more information, see Minimum-Jerk Trajectory on page 1-0 . Then generate the joint configurations of the Franka Emika Panda robot using inverse kinematics.

First, create a set of waypoints, and then create the minimum-jerk trajectory using `minjerkpolytraj`.

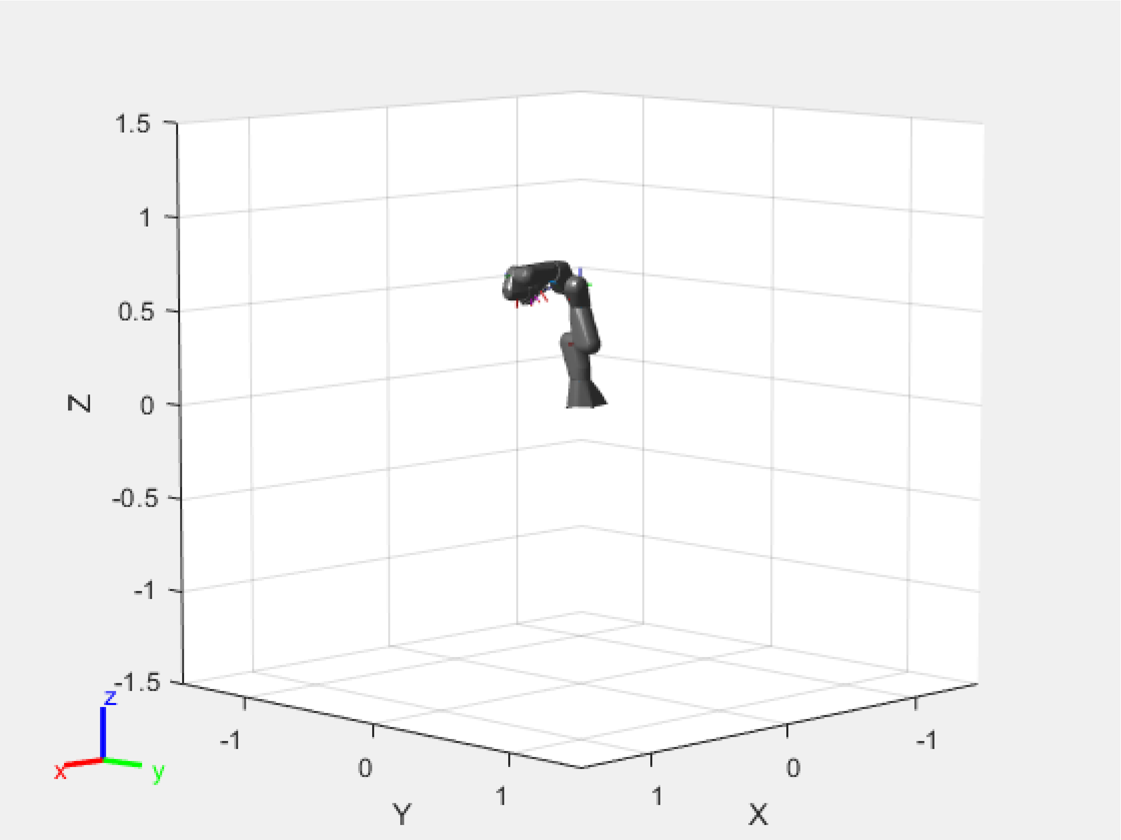
```
frankaSpaceWaypoints = [0.5 0.25 0.25; 0.75 0 0.35; 0.5 -0.25 0.25; 0.5 0.25 0.25]';
frankaTimepoints = linspace(tvec(1),tvec(end),4);
[pos,vel] = minjerkpolytraj(frankaSpaceWaypoints,frankaTimepoints,numSamples);
```

Use the `inverseKinematics` function to create an inverse kinematics solver and solve for configurations that reach the desired end-effector positions over the trajectory.

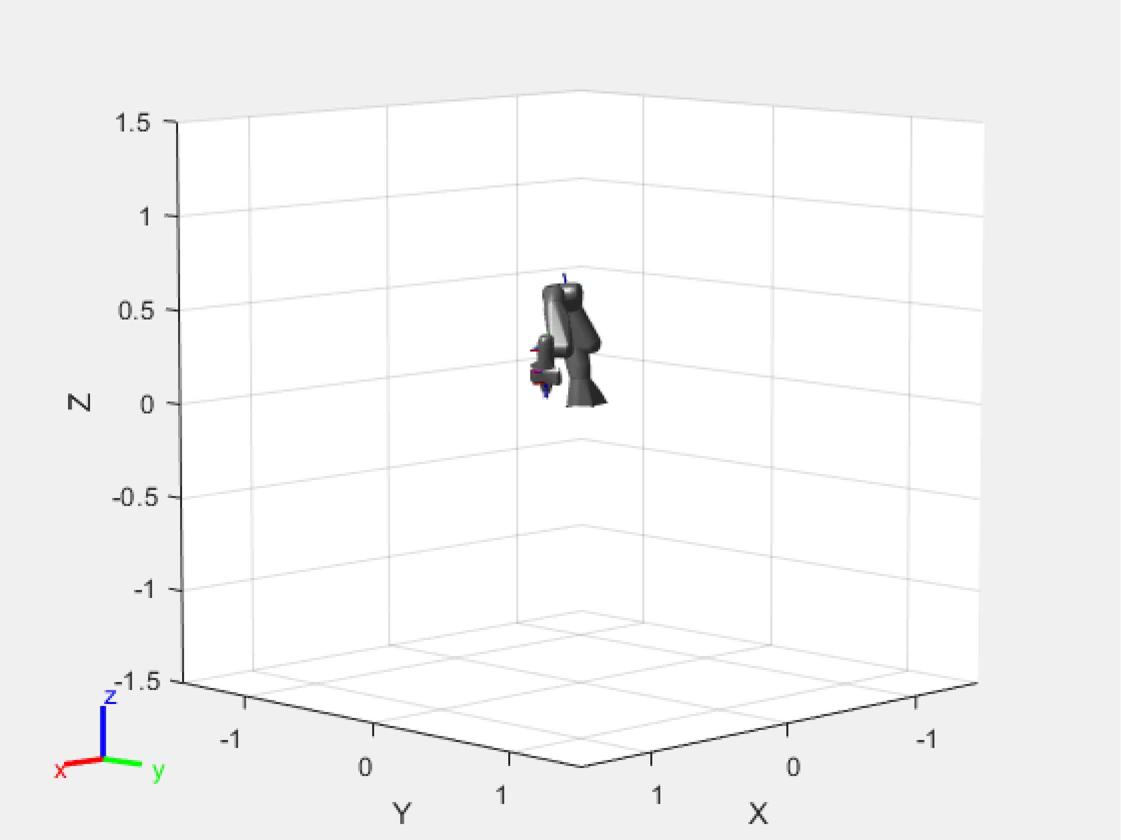
```
rng(0) % Seed the RNG so the inverse kinematics solution is consistent
ik = inverseKinematics(RigidBodyTree=robot);
ik.SolverParameters.AllowRandomRestart = false;
q = zeros(9,numSamples);
weights = [0.2 0.2 0.2 1 1 1]; % Prioritize position over orientation
initialGuess = [0, 0, 0, -pi/2, 0, 0, 0, 0.01, 0.01]'; % Choose an inital guess within the robot
for i = 1:size(pos,2)
    targetPose = trvec2tform(pos(:,i)')*eul2tform([0, 0, pi]);
    q(:,i) = ik('panda_hand',targetPose,weights,initialGuess);
    initialGuess = q(:,i); % Use the last result as the next initial guess
end
```

Show the results using the robot model.

```
figure
set(gcf,'Visible','on')
show(robot);
```

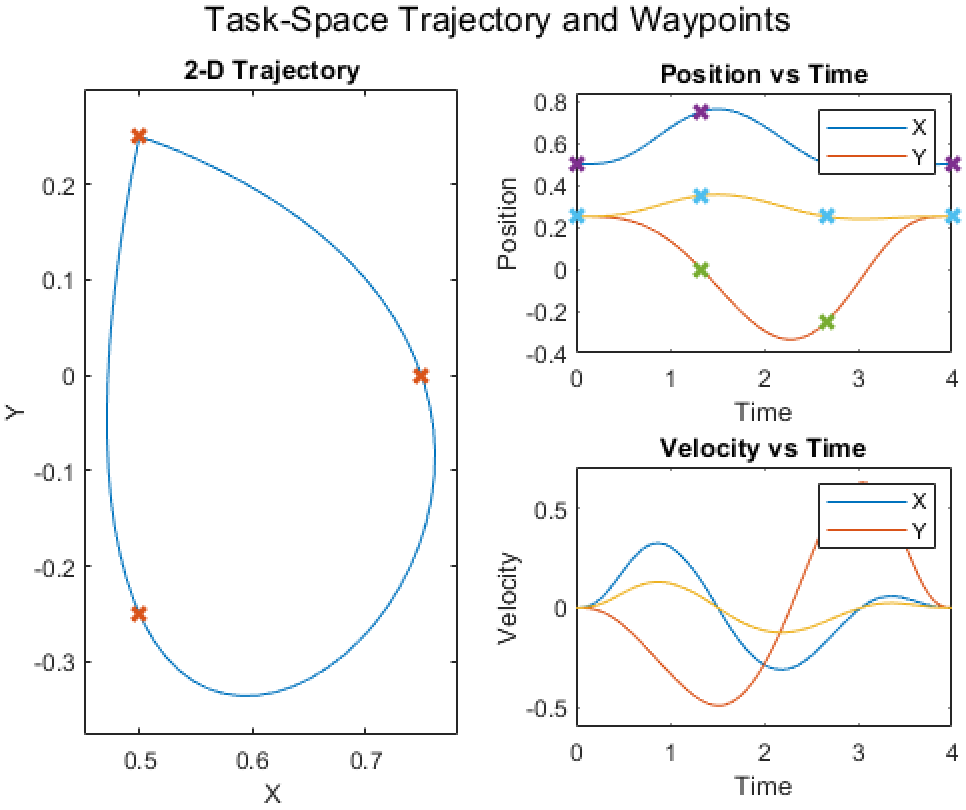


```
rc = rateControl(sampleRate);
for i = 1:numSamples
    show(robot, q(:,i),FastUpdate=true,PreservePlot=false);
    waitfor(rc);
end
```



To examine the different joint positions and velocities, visualize these results by plotting all the dimensions against time. Use the `helperPlotTaskSpaceTraj` helper function to plot the task-space trajectory and its waypoints.

```
helperPlotTaskSpaceTraj('Task-Space Trajectory and Waypoints', ...
    tvec,pos,vel,frankaSpaceWaypoints,frankaTimepoints);
```



Compare Various Trajectory Profiles

Gemerate trajectories using different tools, and then compare them using both task- and joint-space visualization.

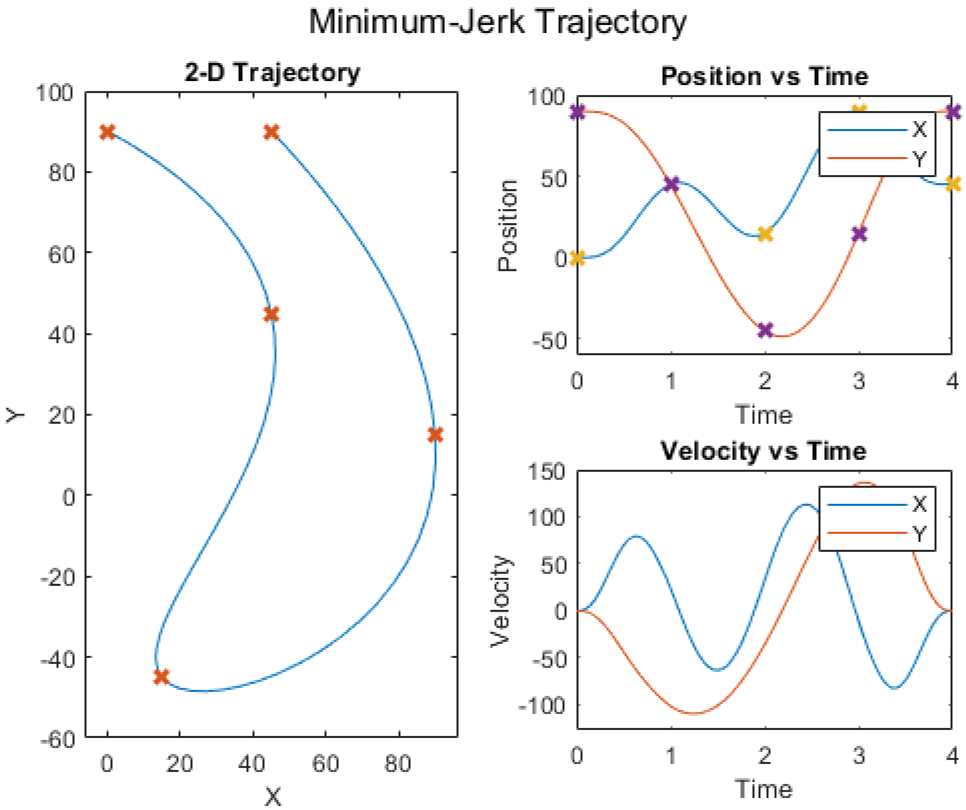
```
wpts = [0 45 15 90 45; 90 45 -45 15 90];
tpts = 0:(size(wpts,2)-1);

% Derived quantities.
sampleRate = 20;
tvec = tpts(1):1/sampleRate:tpts(end);
numSamples = length(tvec);
```

Minimum-Jerk Trajectory

The `minjerkpolytraj` function connects waypoints using a smooth, continuous motion. With the default boundary conditions, the trajectory has zero initial and final velocity, but passes through all intermediate waypoints with a continuous velocity.

Minimum-jerk trajectories are named as such because they minimize jerk, the third time-derivative of the motion, resulting in a smooth profile that is convenient for mechanical systems. The basic minimum-jerk trajectory is an analytical solution that hits the waypoints at the specified time points.

```
[q,qd,~,~,~,~,tvec] = minjerkpolytraj(wpts,tpts,numSamples);
helperPlotTaskSpaceTraj('Minimum-Jerk Trajectory',tvec,q,qd,wpts,tpts);
```



Trapezoidal Velocity Profile Trajectory

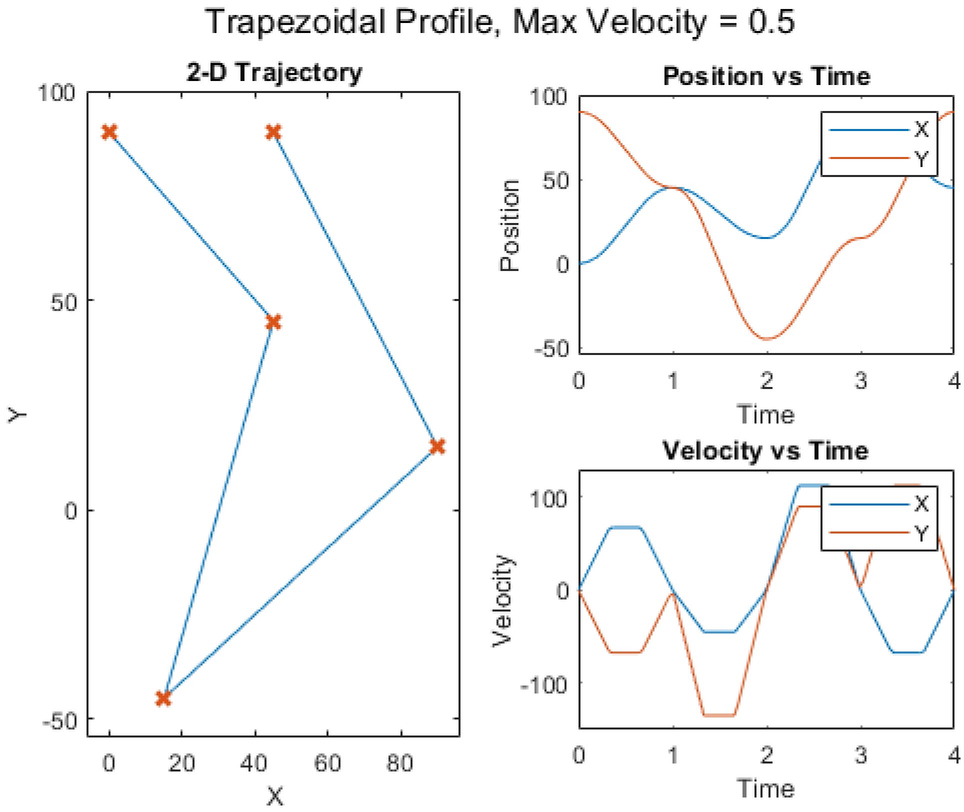
A trapezoidal velocity profile stops at each waypoint, and ensures smooth point-to-point motion. The profile name comes from the three phases of each segment that connects two waypoints:

- Acceleration from zero velocity to peak velocity
- Constant speed at the peak velocity
- Deceleration to zero velocity

This results in a velocity profile that is a trapezoid over each segment. Each segment is characterized by the end time, peak velocity, peak acceleration, and acceleration time parameters, but specifiying any two is sufficient to fully define the motion. For more information, see `trapveltraj`.

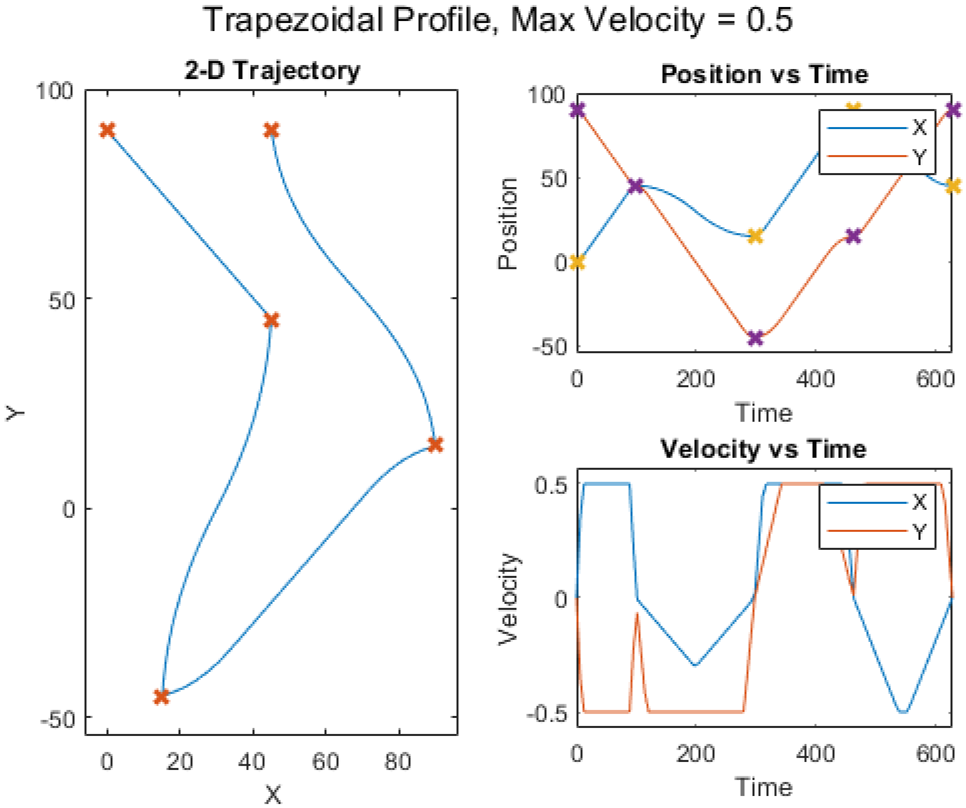
You can use a basic trapezoidal profile when the goal is to connect a set of waypoints, stopping at each along the way. For example, this code connects waypoints using 1-second segments.

```
[q,qd,~,t] = trapveltraj(wpts,100);
helperPlotTaskSpaceTraj('Trapezoidal Profile, Max Velocity = 0.5',t,q,qd,wpts);
```



Trapezoidal profiles often seek to satisfy certain constraints, such as velocity or acceleration bounds. Because the trapezoidal profile is an exact specification, use a helper function to translate constraint bounds to exact profile specifications. The `helperProfileForMaxVel` helper function accepts velocity bounds. For more information on using the `trapveltraj` function to design velocity profiles, see the “Design Trajectory with Velocity Limits Using Trapezoidal Velocity Profile” on page 1-352example.

```
[endTimes,peakVels] = helperProfileForMaxVel(wpts, 0.5);
[q,qd,~,t] = trapveltraj(wpts,100,EndTime=endTimes,PeakVelocity=peakVels);

% The time at which the waypoints are hit is the vector of cumulative sums
% of the end times
trapVelTrajTime = [0 cumsum(endTimes(1,:))];
helperPlotTaskSpaceTraj('Trapezoidal Profile, Max Velocity = 0.5',t,q,qd,wpts,trapVelTrajTime);
```

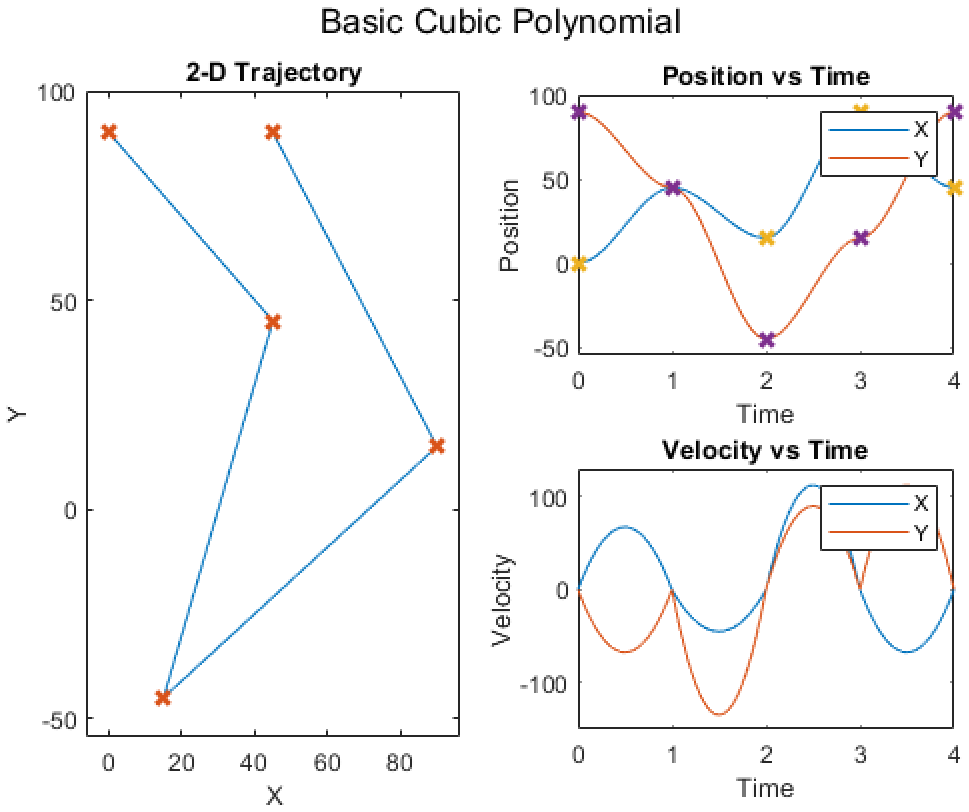


Cubic and Quintic Polynomial Trajectories

Some use cases can require a more general polynomial trajectory. The `cubicpolytraj` and `quinticpolytraj` functions are general tools for creating interpolating piecewise polynomials. Like the previous trajectory tools, they return position, velocity, and acceleration, as well as the piecewise polynomial object.

In the default case, these functions use zero-valued boundary conditions, resulting in trajectories that stop at every waypoint.

```
[q,qd] = cubicpolytraj(wpts,tpts,tvec);
helperPlotTaskSpaceTraj('Basic Cubic Polynomial',tvec,q,qd,wpts,tpts);
```



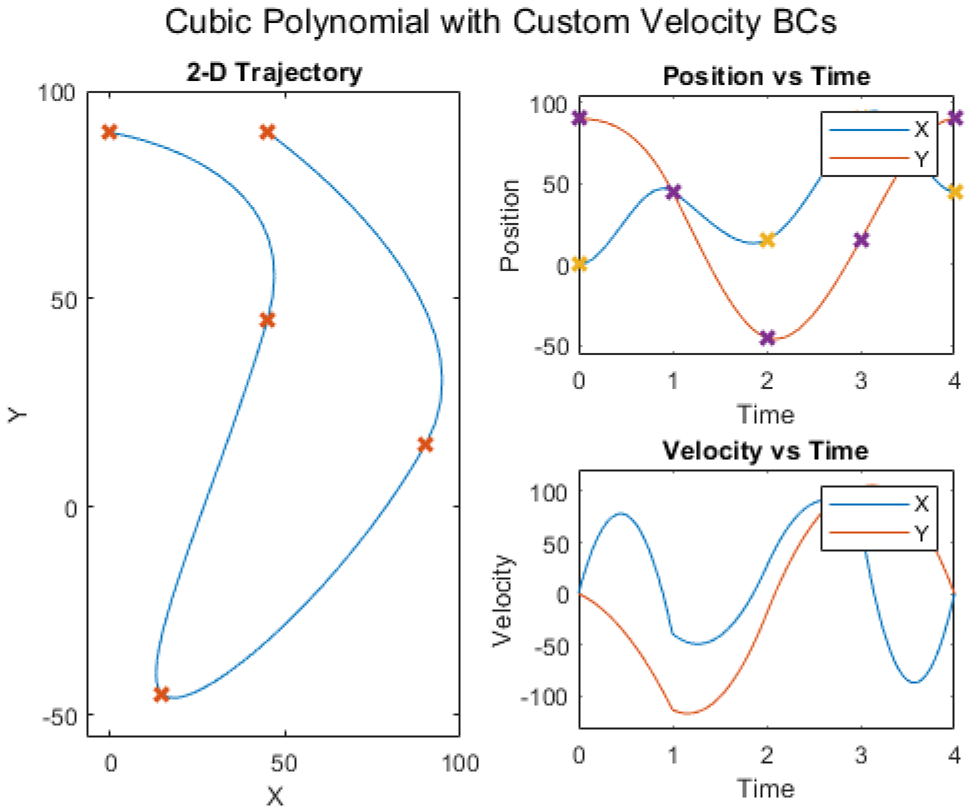
You can use these functions to design an interpolating polynomial with custom boundary behavior. For example, by using these functions together with other piecewise polynomial tools like `spline`, `pchip`, or `makima`, you can create smooth profiles with desirable motion at the waypoints.

This example code derives boundary conditions from one of the built-in piecewise polynomial functions in MATLAB. Select a polynomial function to see how it affects the cubic polynomial velocity profile. The code uses a helper file to compute the velocity.

```
smoothPP = spline ▾(tpts,wpts);
smoothVelPP = mkpp(smoothPP.breaks,robotics.core.internal.polyCoeffsDerivative(smoothPP.coefs),s
smoothVelPoly = ppval(smoothVelPP,tpts);
```

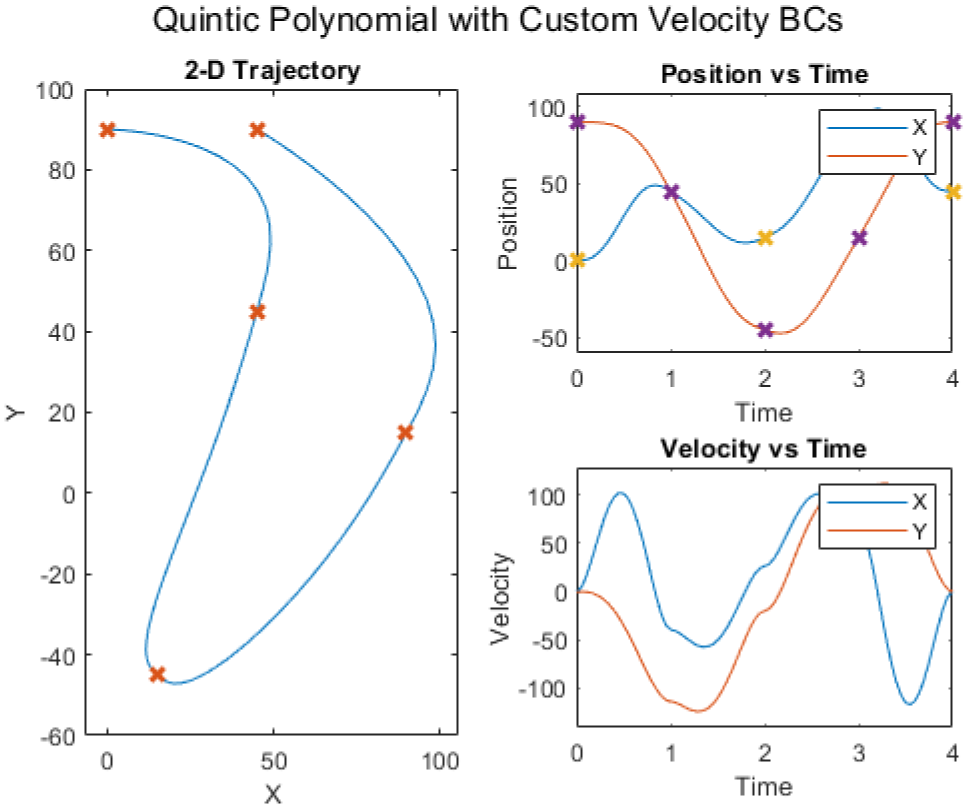
Use the derived velocity as the boundary condition for the inner waypoints. Leave the outside boundary conditions as zero velocity to ensure the trajectory starts and ends at zero velocity

```
boundaryVel = zeros(size(smoothVelPoly));
boundaryVel(:,2:end-1) = smoothVelPoly(:,2:end-1);
[q,qd] = cubicpolytraj(wpts,tpts,tvec,VelocityBoundaryCondition=boundaryVel);
helperPlotTaskSpaceTraj('Cubic Polynomial with Custom Velocity BCs',tvec,q,qd,wpts,tpts);
```



Using a quintic polynomial instead ensures a smooth velocity profile.

```
[q,qd] = quinticpolytraj(wpts,tpts,tvec,VelocityBoundaryCondition=boundaryVel);
helperPlotTaskSpaceTraj('Quintic Polynomial with Custom Velocity BCs',tvec,q,qd,wpts,tpts);
```

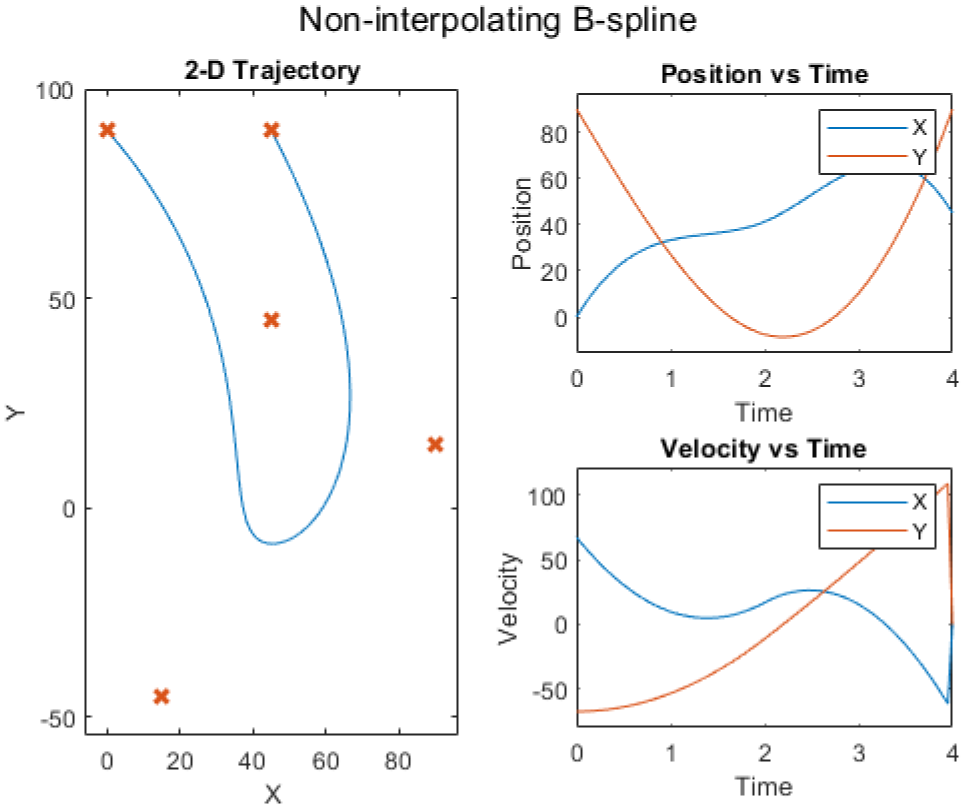


B-Spline Trajectory

A B-spline polynomial results in smooth, continuous motion, and is predominantly used for task-space applications. You can create this trajectory by using the `bsplinepolytraj` function. Unlike the other trajectories, which interpolate the waypoints that define them, a B-spline is defined by a set of control points. The resulting trajectory hits only the initial and final control points, but falls in the convex hull of the complete set of control points.

Apply the previously used waypoints as the control points.

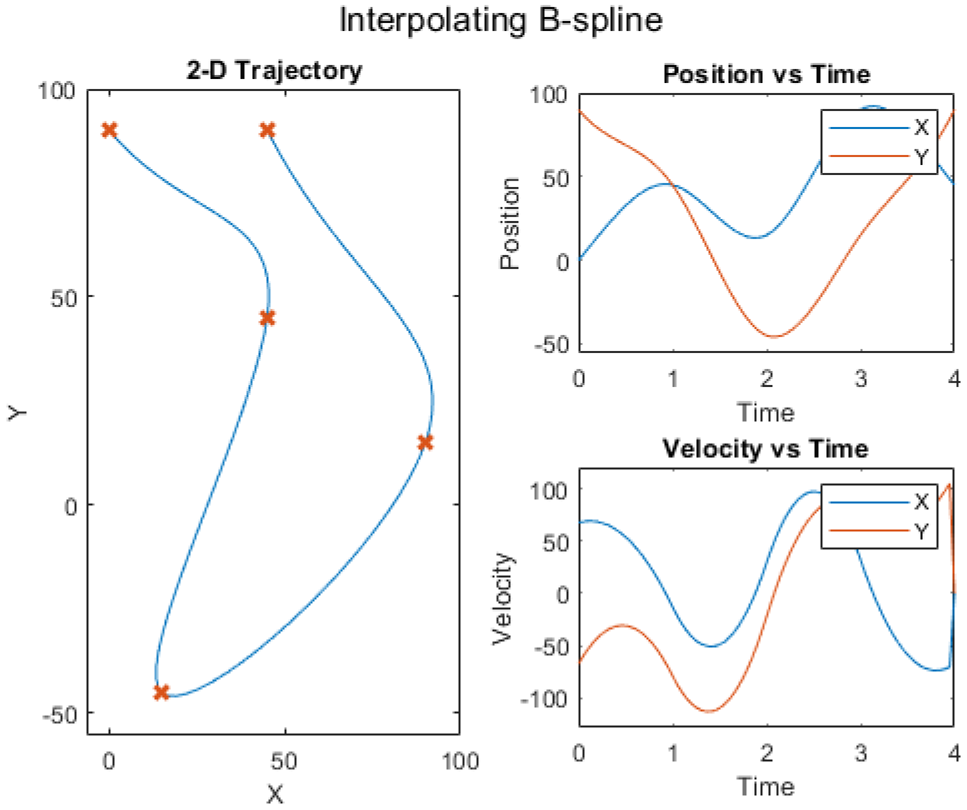
```
[q,qd] = bsplinepolytraj(wpts,tpts([1 end]),tvec);
helperPlotTaskSpaceTraj('Non-interpolating B-spline',tvec,q,qd,wpts);
```



You can create an interpolating B-spline by deriving a new set of control points from the original waypoints, which the B-spline polynomial interpolates with the original waypoints. Like the standard B-spline, use this polynomial primarily for task-space applications.

Use the `helperCreateControlPointsFromWaypoints` helper function to derive the new control points.

```
cpts = helperCreateControlPointsFromWaypoints(wpts);
[q,qd] = bsplinepolytraj(cpts,tpts([1 end]),tvec);
helperPlotTaskSpaceTraj('Interpolating B-spline',tvec,q,qd,wpts);
```



More Tools for Task-Space Trajectories

Some task-space applications benefit from computing not only the trajectories that interpolate positions, but also the full pose of the manipulator. In those cases, consider using the `rottraj` and `transformtraj` functions. The `rottraj` function creates a trajectory between two rotations, while the `transformtraj` function does the same for two 4-by-4 homogeneous transformation matrices. The functions output angular acceleration and velocity in addition to the position derivatives.

This example connects two poses, `T1` and `T2`, which contain position and orientation data.

```
T1 = eul2tform([pi/4 0 pi/3]);
T2 = trvec2tform([5 -2 1]);
tInterval = [0 1];
tvec = 0:0.01:1;
```

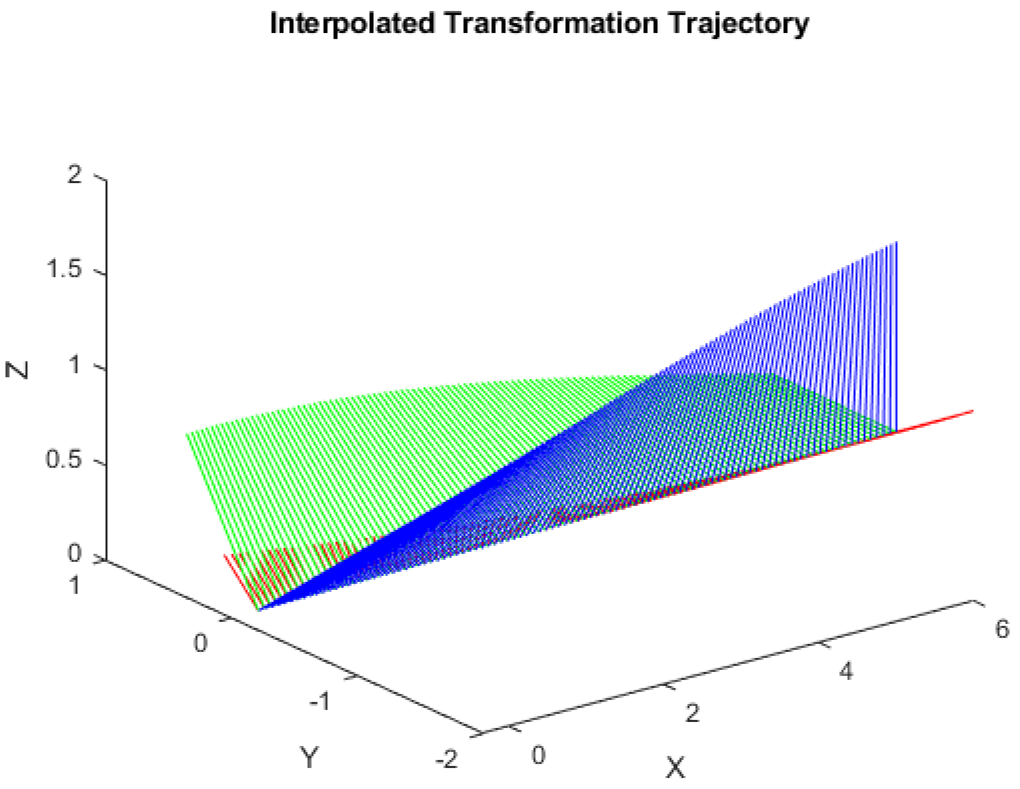
Interpolate the two transformation matrices using `transformtraj` to create a trajectory of full, interpolated poses, represented as transformation matrices.

```
[tfInterp,v1,a1] = transformtraj(T1,T2,tInterval,tvec);
```

Plot the trajectory in 3-D space using `plotTransforms`. This function requires both the rotational data as a quaternion and the translational data from the interpolated transformation matrices trajectory. Use `tform2quat` to find the quaternion and `tform2trvec` to find the translational data from the trajectory.

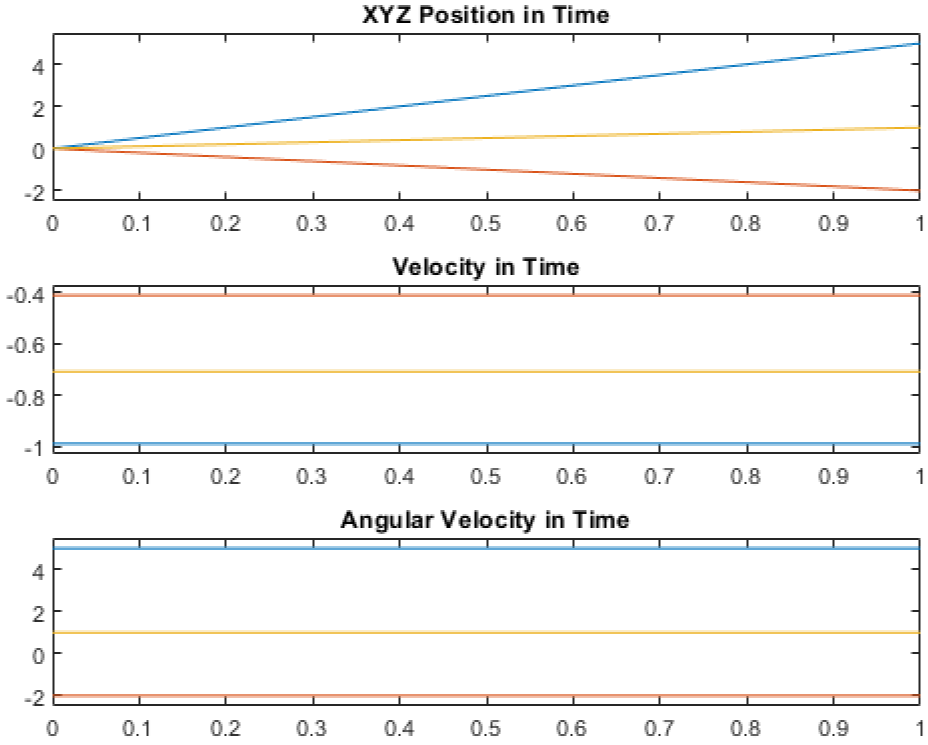
```
figure
rotations = tform2quat(tfInterp);
```

```
translations = tform2trvec(tfInterp);
plotTransforms(translations,rotations)
title('Interpolated Transformation Trajectory')
xlabel('X')
ylabel('Y')
zlabel('Z')
```



The 3-D plot shows that the rotations and motion are linear. Recall that feasible manipulator trajectories should be smooth and continuous. Because the interpolation is linear, the trajectory is not guaranteed to result in a smooth motion. This becomes clearer when you plot the position and velocities separately.

```
figure
positions = reshape(tfInterp(1:3,4,:),3,size(tfInterp,3));
subplot(3,1,1); plot(tvec,positions)
title('XYZ Position in Time')
ylim('padded')
subplot(3,1,2); plot(tvec,v1(1:3,:))
title('Velocity in Time')
ylim('padded')
subplot(3,1,3); plot(tvec,v1(4:6,:));
title('Angular Velocity in Time')
ylim('padded')
```



The velocities start and end at non-zero values, which is not a feasible trajectory for a manipulator, and would cause sudden and jerky motion.

You can use the `TimeScaling` name-value argument of `transformtraj` as a workaround. This argument defines the trajectory time using an intermediate parameterization, *s*, such that `transformtraj` is defined using *s*(*t*) as time. In the default case used in this example, time scaling is uniform, so *s*(*t*) = *t*. The result is a linear motion between each pose. Instead, use time scaling defined by a minimum-jerk trajectory: *s*(*t*) = `minjerkpolytraj`(*t*).

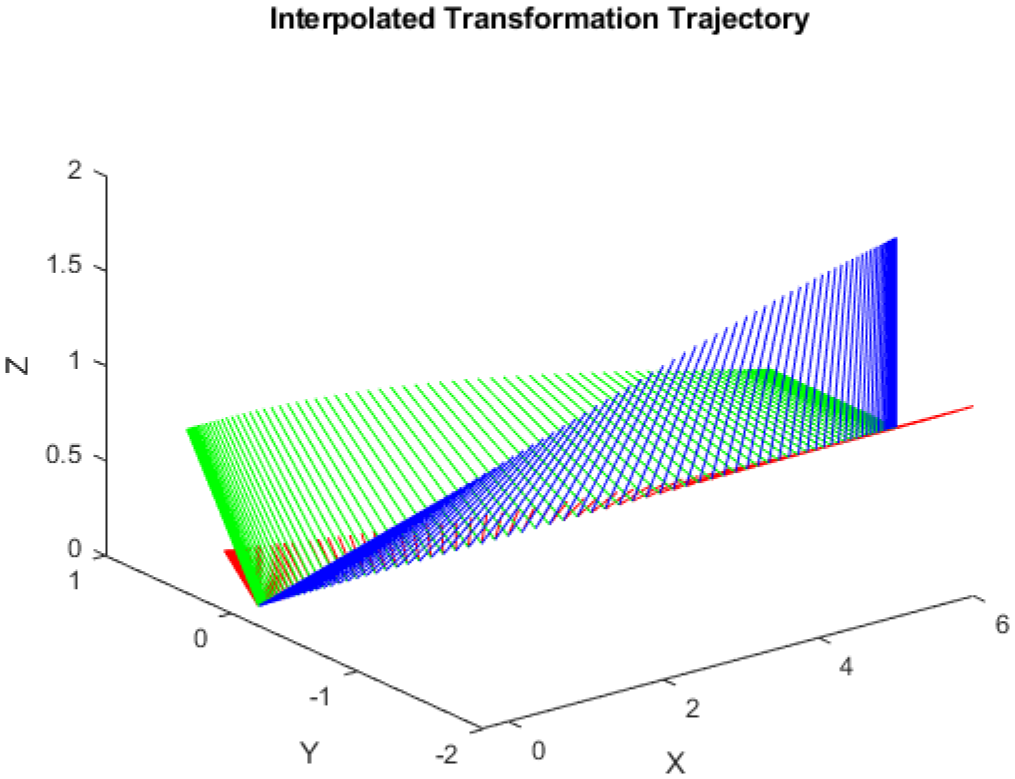
Time scaling is a discrete set of values, $\left[s;\ \frac{d}{\mathrm{d}t}s;\ \frac{d^2}{\mathrm{d}t^2}s\right]$, which sample the function *s*(*t*), defined on the interval $s = [0, 1]$.

```
% The time scaling is a discrete set of values [s; ds/dt; d^2s/dt^2] that
% are sample the function s(t), defined on the interval s = [0,1]
[s,sd,sdd] = minjerkpolytraj([0 1],tInterval,numel(tvec));
[tfInterp,v1,a1] = transformtraj(T1,T2,tInterval,tvec,TimeScaling=[s; sd; sdd]);
```

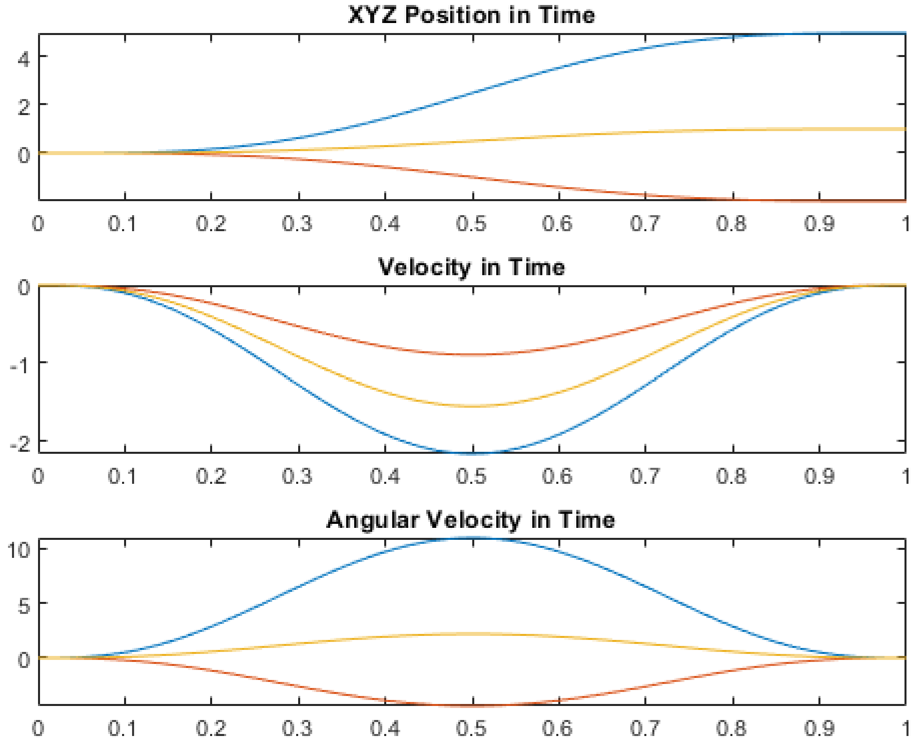
Plot the interpolated transformation trajectory again to compare against the previous plots.

```
figure
rotations = tform2quat(tfInterp);
translations = tform2trvec(tfInterp);
plotTransforms(translations,rotations)
title('Interpolated Transformation Trajectory')
xlabel('X')
```

```
ylabel('Y')
zlabel('Z')
```



```
figure
positions = reshape(tfInterp(1:3,4,:),3,size(tfInterp,3));
subplot(3,1,1); plot(tvec,positions)
title('XYZ Position in Time')
subplot(3,1,2); plot(tvec,v1(1:3,:))
title('Velocity in Time')
subplot(3,1,3); plot(tvec,v1(4:6,:))
title('Angular Velocity in Time')
```



While the motion in space follows the same path, the second set of plots clarifies that the velocities are smooth and followable in time, resulting in a trajectory that is feasible for a manipulator or other mechanical system to follow.

Load Predefined Robot Models

To quickly access common robot models, use the `loadrobot` function, which loads commercially available robot models like the Universal Robots™ UR10 cobot, Boston Dynamics™ Atlas humanoid, and KINOVA™ Gen 3 manipulator. Explore how to generate joint configurations and interact with the robot models.

To import your own universal robot description format (URDF), see the `importrobot` function.

Specify the robot model type as a string to the `loadrobot` function. Utilize tab completion to select from the list of provided models as inputs.

To use column vectors for joint configurations, specify the data format as `"column"`.

```
ur10 = loadrobot("universalUR10");
atlas = loadrobot("atlas");
gen3 = loadrobot("kinovaGen3","DataFormat","column");
```

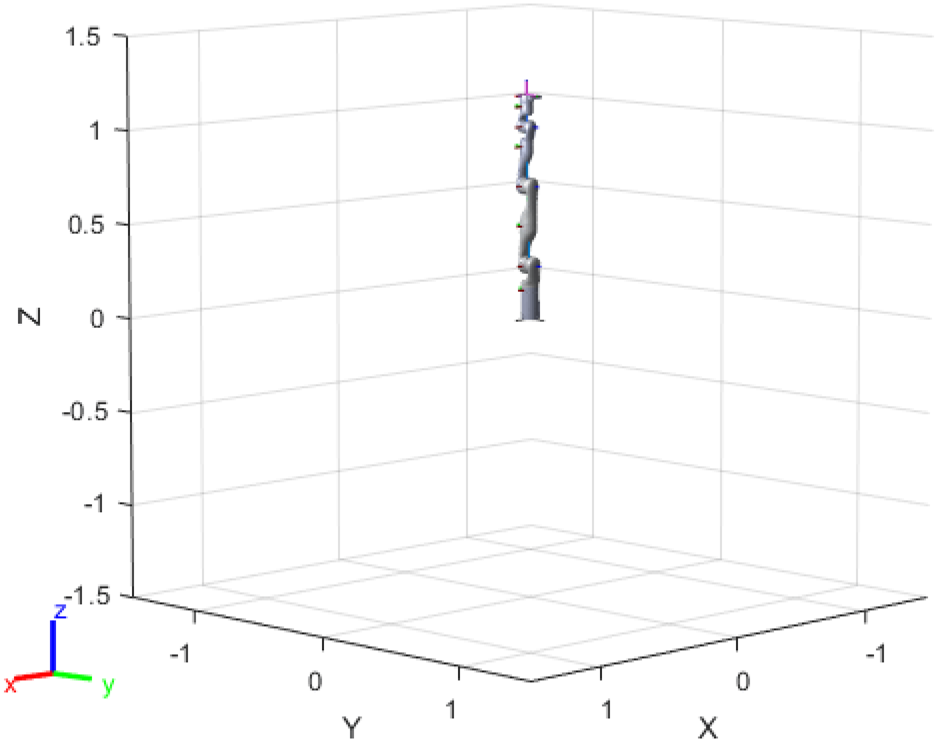
The `loadrobot` function returns a `rigidBodyTree` object thats represents the kinematics and dynamics of each robot model. Some models may not load with dynamics or inertial properties for bodies. Inspect individual rigid bodies using the `Bodies` property or the `getBody` function.

```
disp(gen3);
```

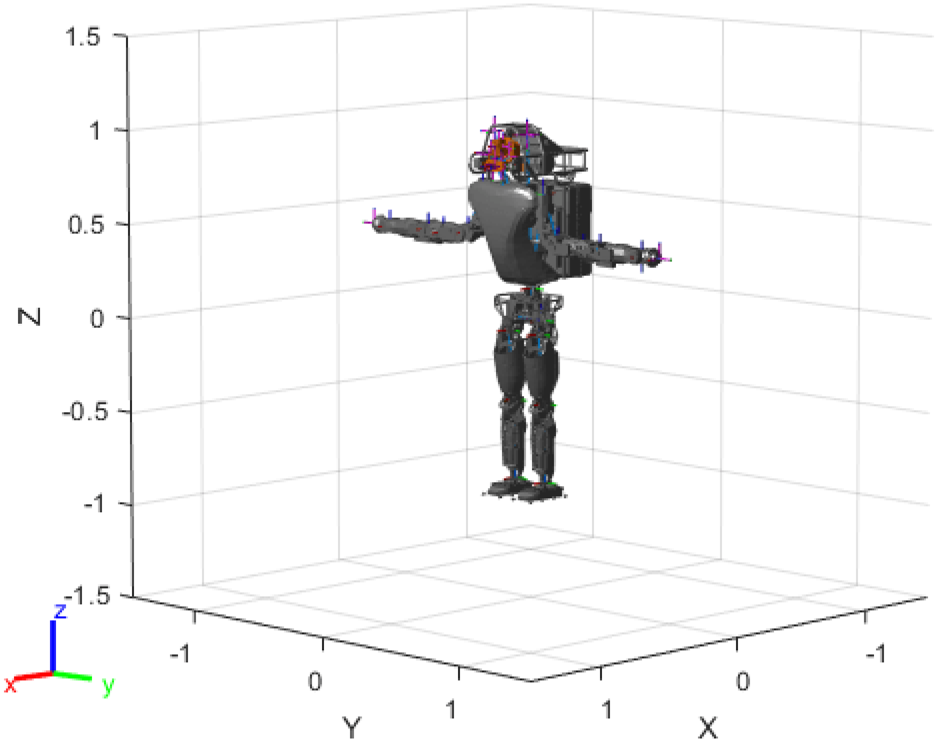
```
  rigidBodyTree with properties:

     NumBodies: 8
        Bodies: {1x8 cell}
          Base: [1x1 rigidBody]
     BodyNames: {1x8 cell}
      BaseName: 'base_link'
       Gravity: [0 0 0]
    DataFormat: 'column'
```

Call `show` to visualize the robot models in the home configuration. Replace the `gen3` object with other models to visualize them.

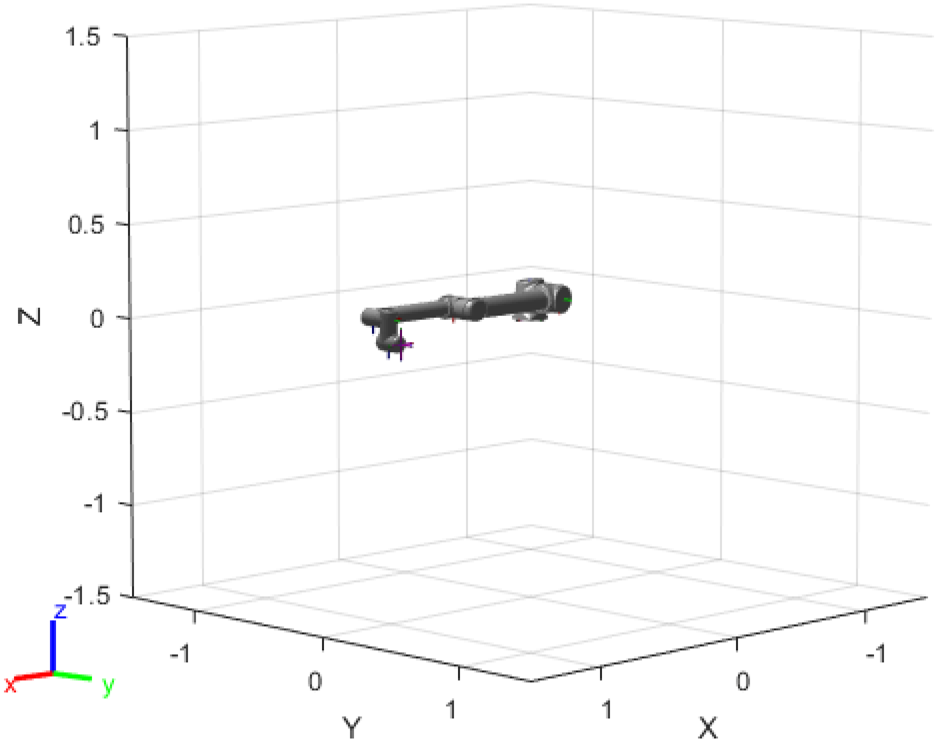
```
show(gen3);
```



```
show(atlas);
```



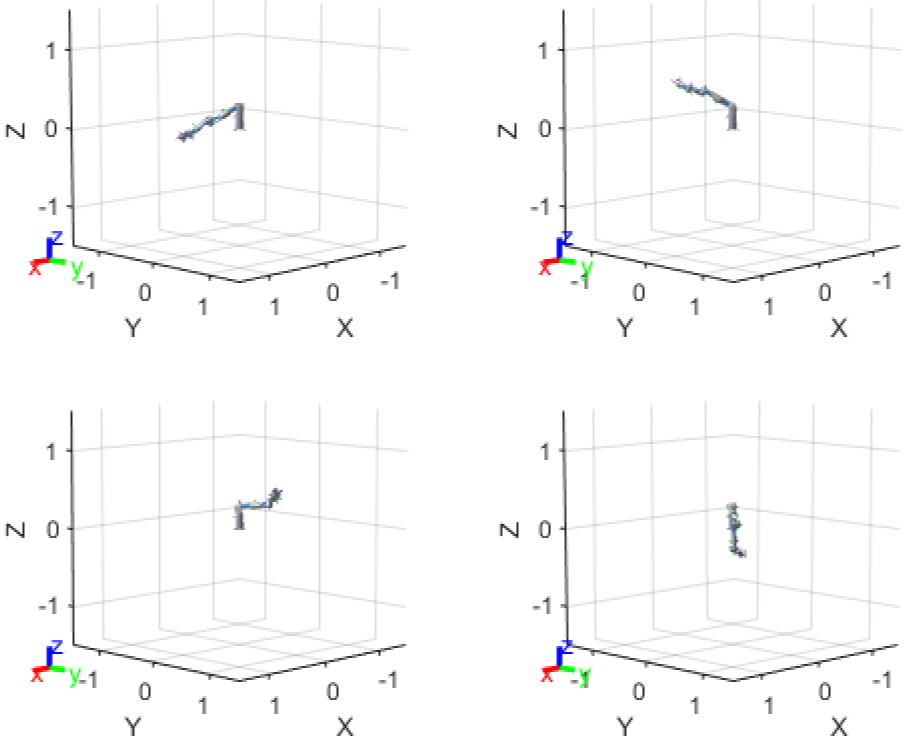
```
show(ur10);
```



Generate Joint Configurations

Generate random configurations for the KINOVA Gen3 robot. The `randomConfiguration` function outputs random joint positions within the limits of the model. To verify the model behaves as expected, visualize a set of four configurations.

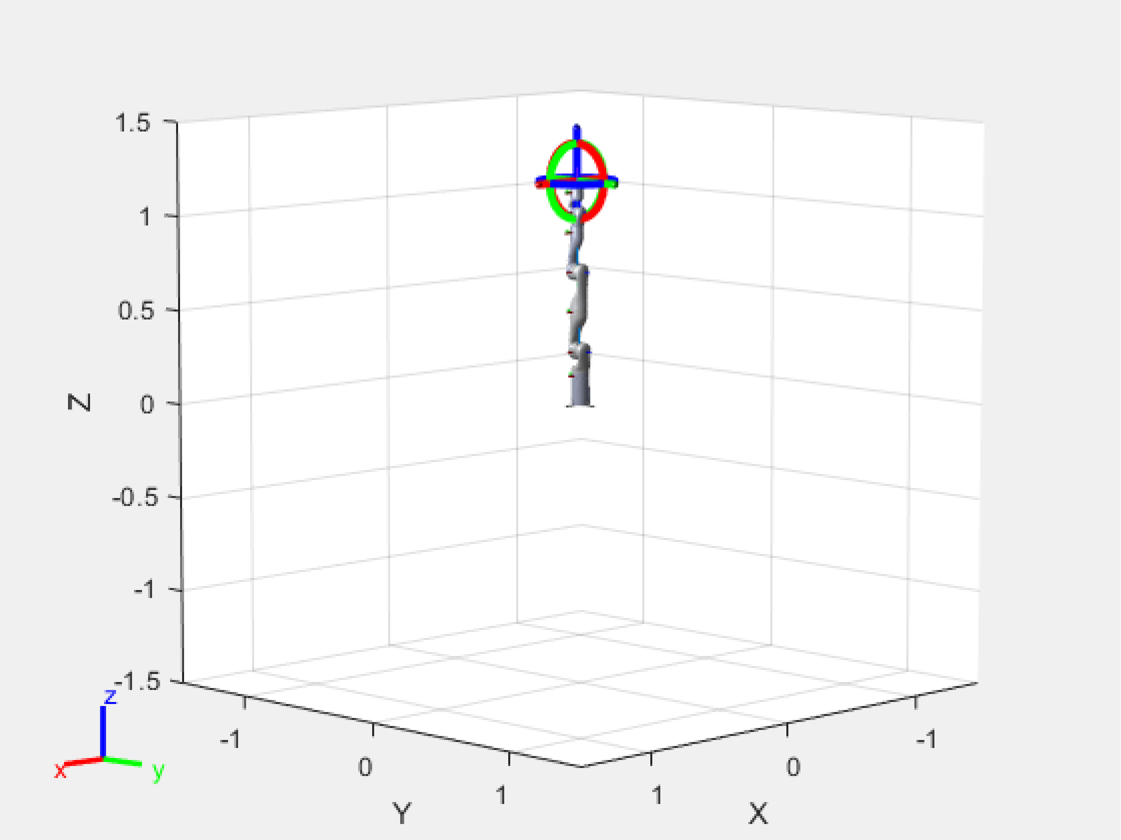
```
for i = 1:4
    subplot(2,2,i)
    config = randomConfiguration(gen3);
    show(gen3,config);
end
```



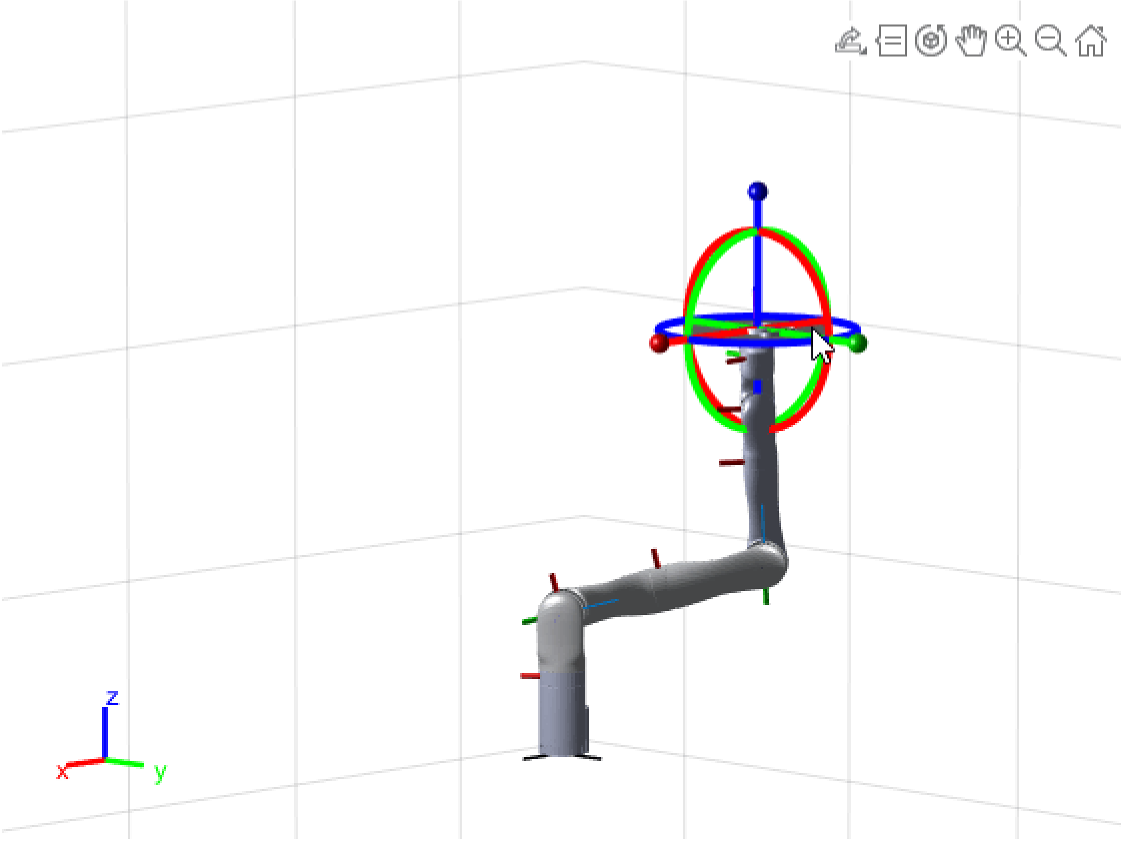
Interact with Robot Model

To move the robot model around and inspect the behavior, load the interactive rigid body tree GUI. You can set target end-effector positions, manually move joints, and select various elements in your model.

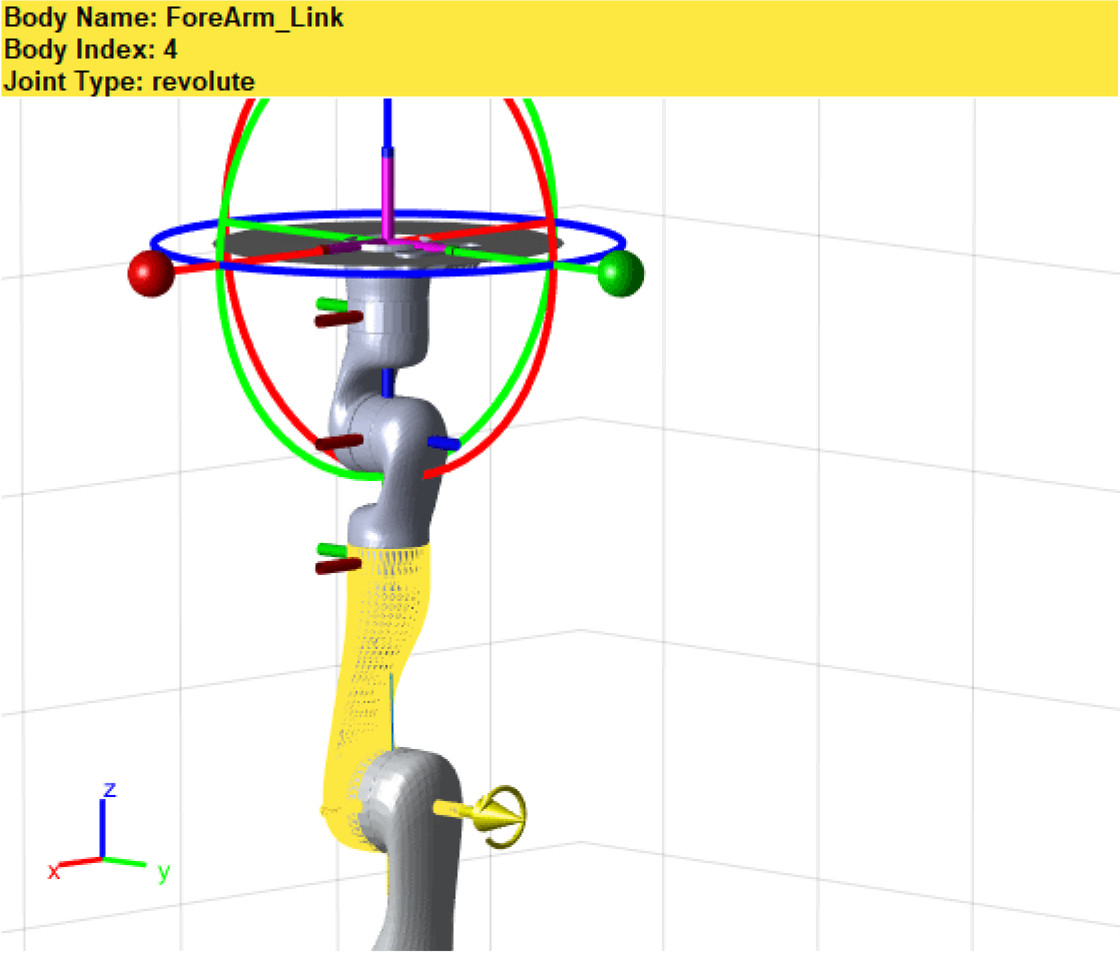
```
interactiveGUI = interactiveRigidBodyTree(gen3);
```



Click and drag the center disk to freely move the target end-effector position. The GUI uses “Inverse Kinematics” to solve for the joint positions of each body. Use the axes to move linearly and the circles to rotate about an axis.

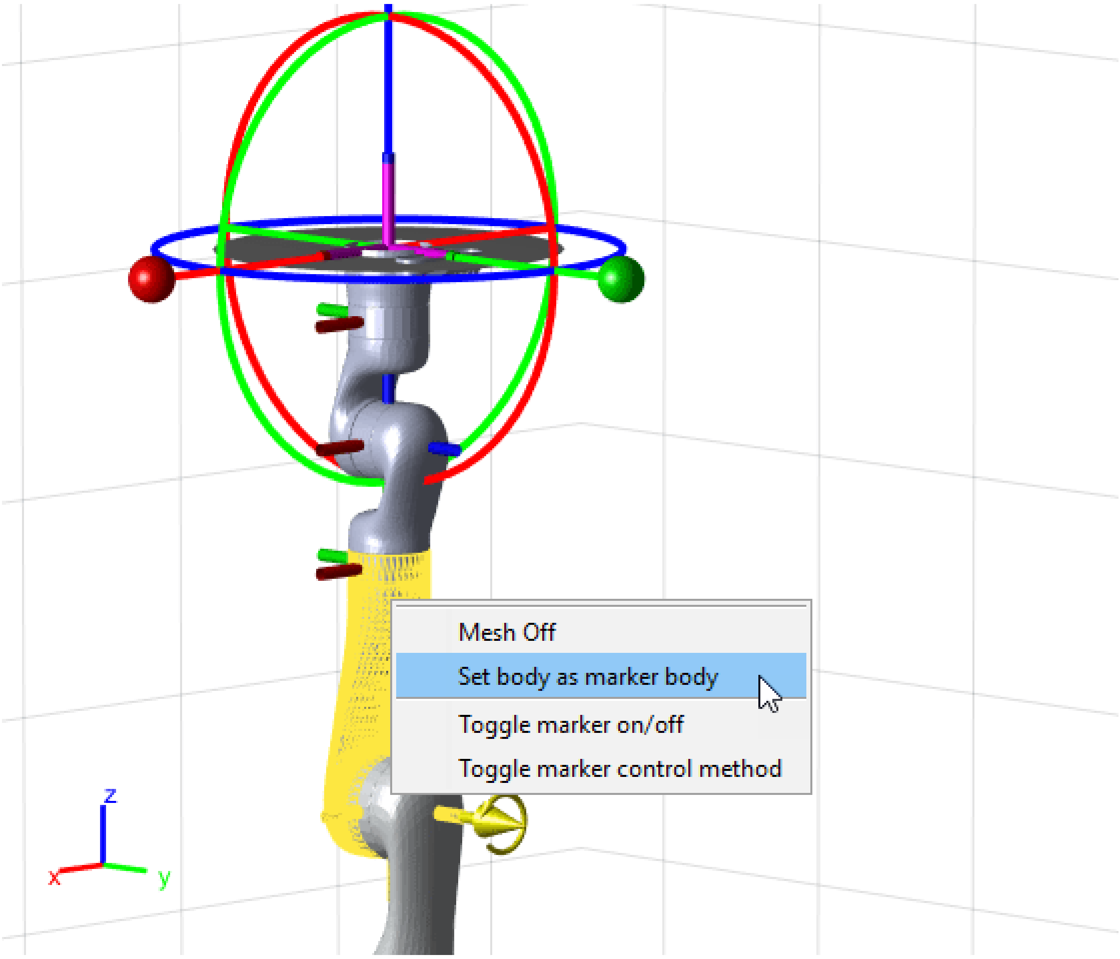


Click a `rigidBody` to view their specific parameters.

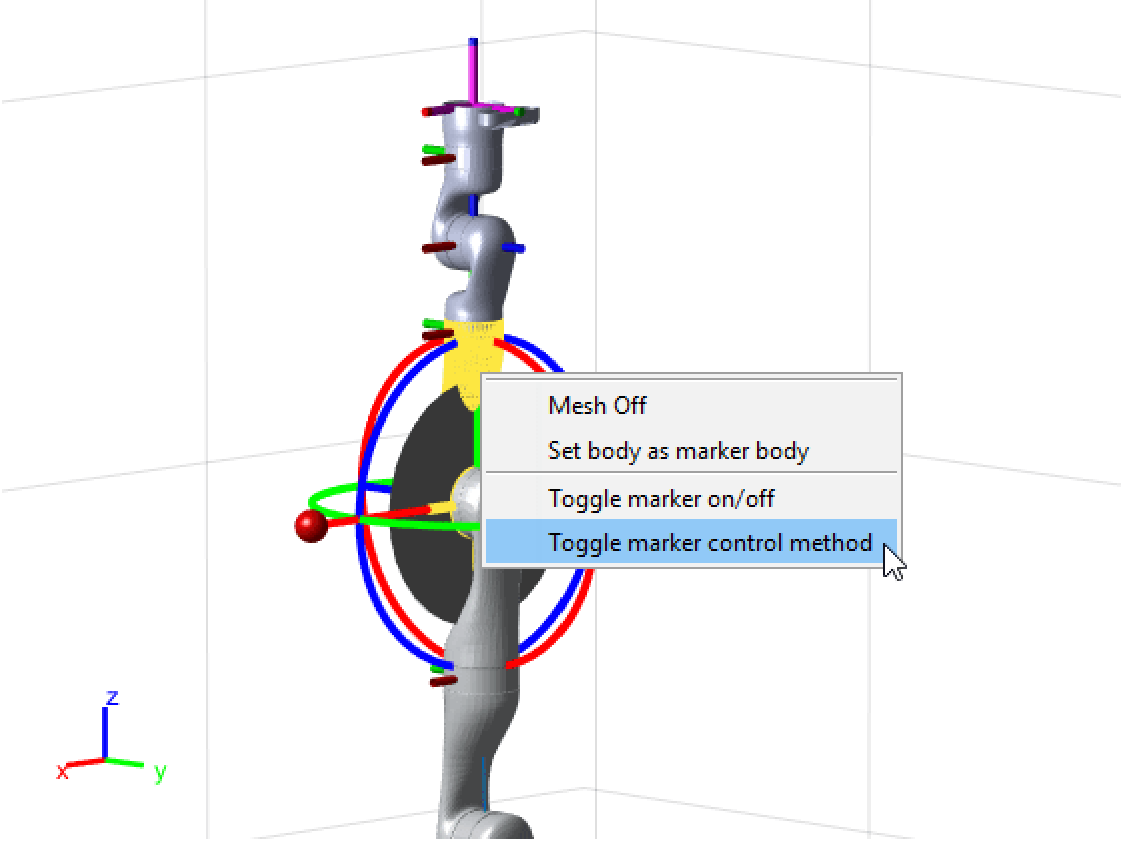


Body Name: ForeArm_Link
Body Index: 4
Joint Type: revolute

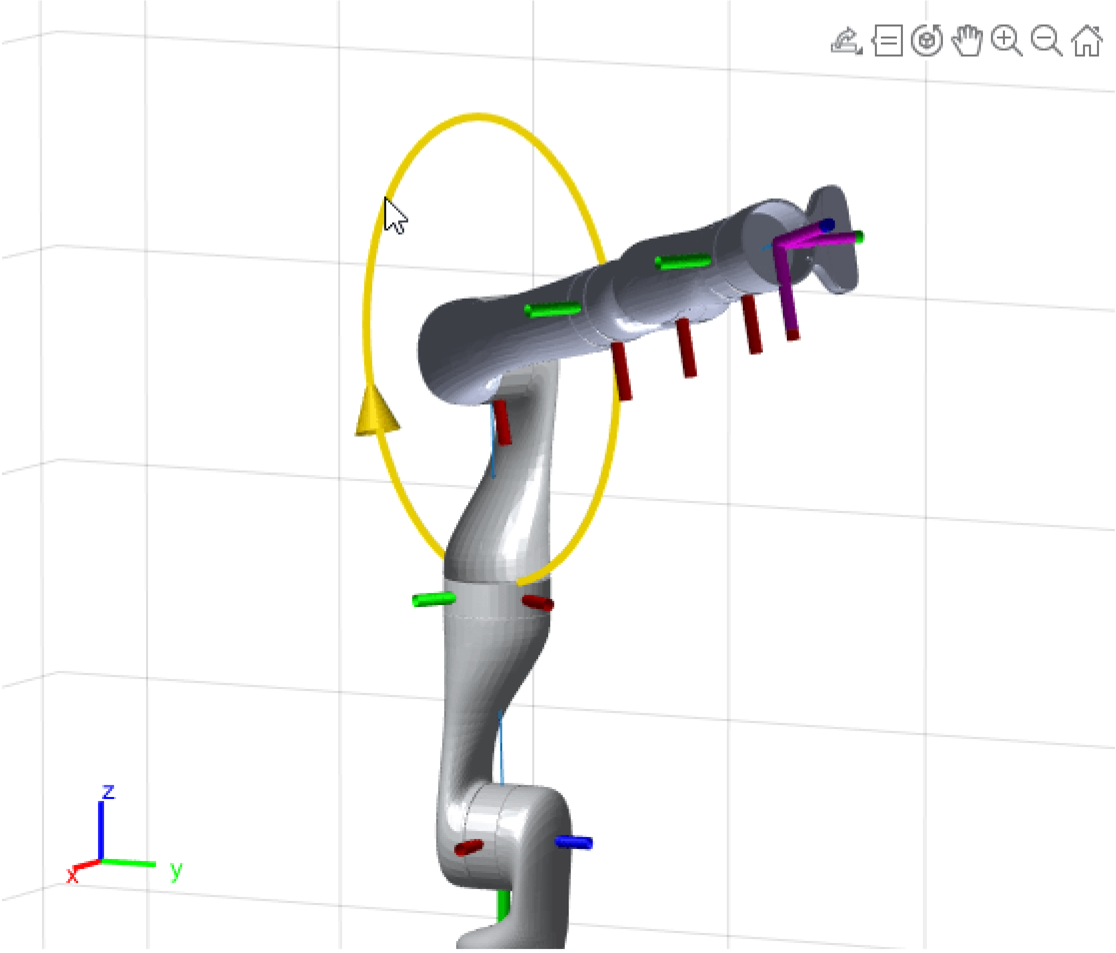
Right-click to set a different target marker body. Changing the target body updates the end-effector of the inverse kinematics solver.



To manually control joints, right-click and toggle the marker control method.



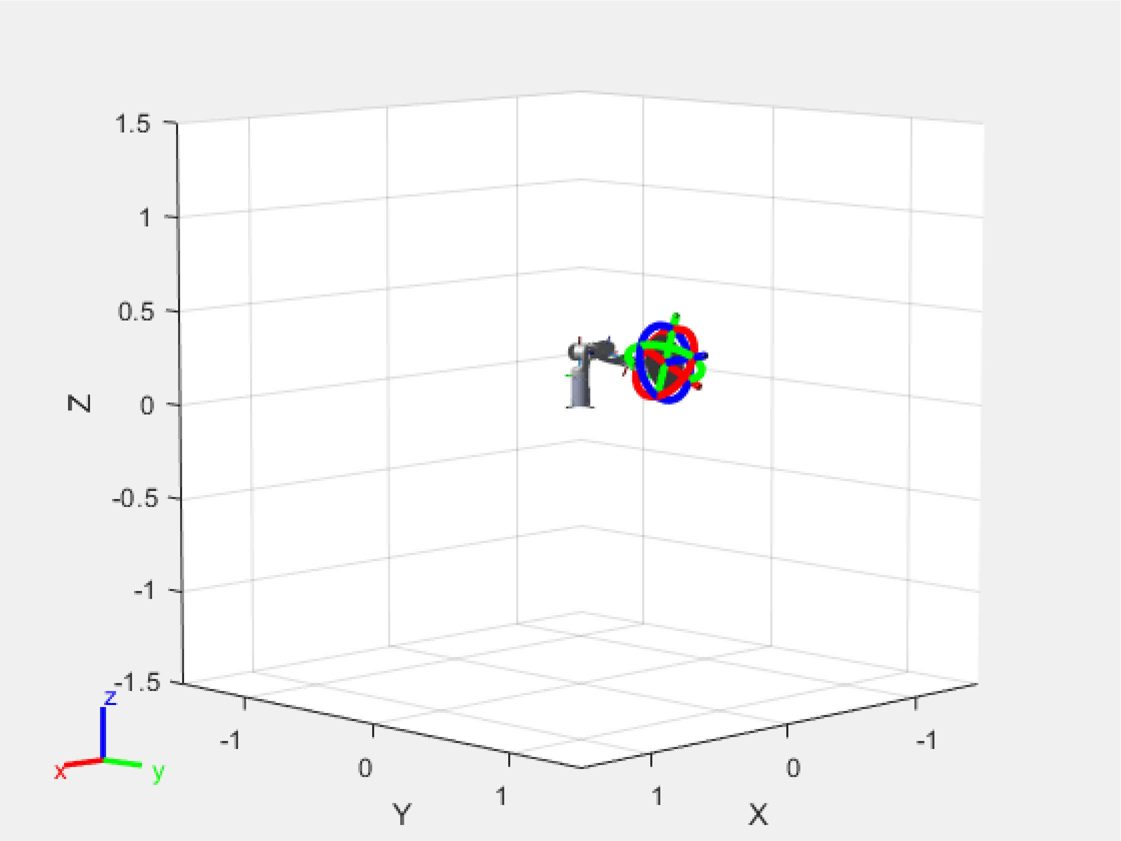
To control the rotation of a revolue joint on the body you selected, click and drag the yellow circle.



Save Joint Configurations

Save specific configurations that you set using the `addConfiguration` function, which stores the current joint positions in the `StoredConfigurations` property. This example sets a random configuration before storing.

```
interactiveGUI.Configuration = randomConfiguration(gen3);
```



```
addConfiguration(interactiveGUI)
disp(interactiveGUI.StoredConfigurations)
```

```
   -0.4218
   -1.6647
    1.3419
   -2.0818
    1.8179
   -0.4140
   -1.4517
```

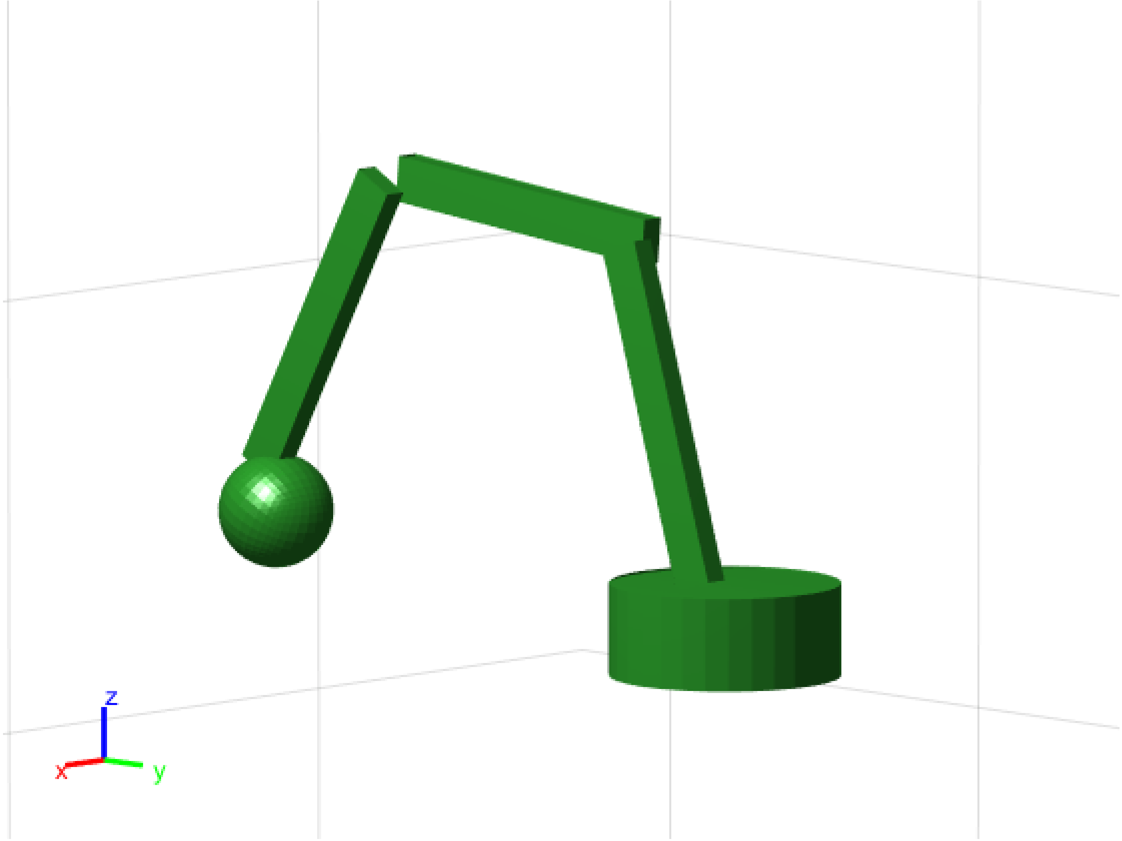
Next Steps

Now that you built your model in MATLAB®, you may want to do many different things.

- Perform “Inverse Kinematics” to get joint configurations based on desired end-effector positions. Specify additional robot constraints other than the model parameters including aiming constraints, Cartesian bounds, or pose targets.
- Generate “Trajectory Generation and Following” based on waypoints and other parameters with trapezoidal velocity profiles, B-splines, or polynomial trajectories.
- Peform “Manipulator Motion Planning” utilizing your robot models and an rapidly-exploring random tree (RRT) path planner.
- Check for “Collision Detection” with obstacles in your environment to ensure safe and effective motion of your robot.

Build Basic Rigid Body Tree Models

This example shows how to use the elements of the rigid body tree robot model to build a basic robot arm with five degrees of freedom. The model built in this example represents a common table top robot arms built with servos and an integrated circuit board.



To load a robot model from a set of common commercially available robots, use the `loadrobot` function. For an example, see **Load Predefined Robot Models**.

If you have a URDF file or Simscape Multibody™ model of your robot, you can import as a rigid body tree using `importrobot`. For an example, see **Import Simscape Multibody Models.**

Create Rigid Body Elements

First, create a `rigidBodyTree` robot model. The rigid body tree robot model represents the entire robot, which is made up of rigid bodies and joints that attach them together. The base of the robot is a fixed frame that defines the world coordinates. Adding your first body, attaches the body to the base.

```
robot = rigidBodyTree("DataFormat","column");
base = robot.Base;
```

Then, create a series of linkages as `rigidBody` objects. The robot consists of a rotating base, 3 rectangular arms, and a gripper.

```
rotatingBase = rigidBody("rotating_base");
arm1 = rigidBody("arm1");
```

```
arm2 = rigidBody("arm2");
arm3 = rigidBody("arm3");
gripper = rigidBody("gripper");
```

Create collision objects for each rigid body with different shapes and dimensions. When you create the collision objects, the coordinate frame is centered in the middle of the object by default. Set the `Pose` property to move the frame to the bottom of each body along the *z*-axis. Model the gripper as a sphere for simplicity.

```
collBase = collisionCylinder(0.05,0.04); % cylinder: radius,length
collBase.Pose = trvec2tform([0 0 0.04/2]);
coll1 = collisionBox(0.01,0.02,0.15); % box: length, width, height (x,y,z)
coll1.Pose = trvec2tform([0 0 0.15/2]);
coll2 = collisionBox(0.01,0.02,0.15); % box: length, width, height (x,y,z)
coll2.Pose = trvec2tform([0 0 0.15/2]);
coll3 = collisionBox(0.01,0.02,0.15); % box: length, width, height (x,y,z)
coll3.Pose = trvec2tform([0 0 0.15/2]);
collGripper = collisionSphere(0.025); % sphere: radius
collGripper.Pose = trvec2tform([0 -0.015 0.025/2]);
```

Add the collision bodies to the rigid body objects.

```
addCollision(rotatingBase,collBase)
addCollision(arm1,coll1)
addCollision(arm2,coll2)
addCollision(arm3,coll3)
addCollision(gripper,collGripper)
```

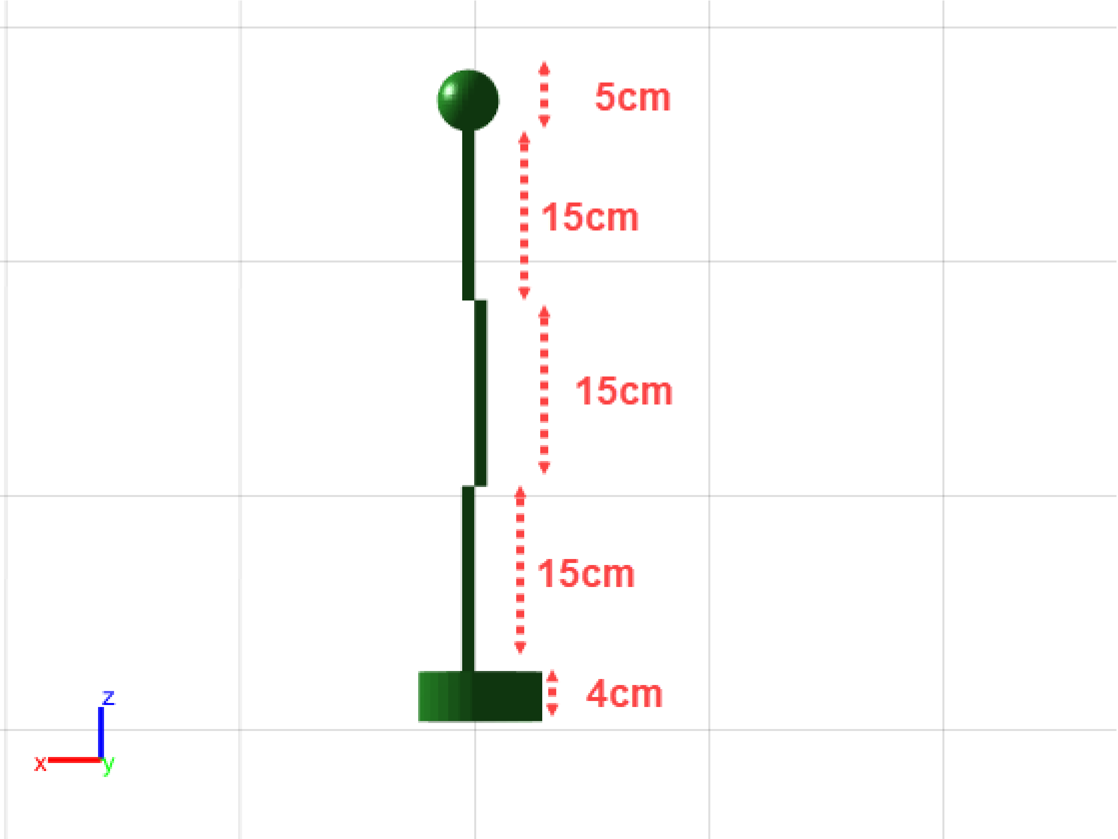
Attach Joints

Each rigid body is attached using a revolute joint. Create the `rigidBodyJoint` objects for each body. Specify the *x*-axis as the axis of rotation for the rectangular arm joints. Specify the *y*-axis for the gripper. The default axis is the *z*-axis.

```
jntBase = rigidBodyJoint("base_joint","revolute");
jnt1 = rigidBodyJoint("jnt1","revolute");
jnt2 = rigidBodyJoint("jnt2","revolute");
jnt3 = rigidBodyJoint("jnt3","revolute");
jntGripper = rigidBodyJoint("gripper_joint","revolute");

jnt1.JointAxis = [1 0 0]; % x-axis
jnt2.JointAxis = [1 0 0];
jnt3.JointAxis = [1 0 0];
jntGripper.JointAxis = [0 1 0] % y-axis
```

```
jntGripper = 
  rigidBodyJoint with properties:

                      Type: 'revolute'
                      Name: 'gripper_joint'
                 JointAxis: [0 1 0]
            PositionLimits: [-3.1416 3.1416]
              HomePosition: 0
    JointToParentTransform: [4x4 double]
     ChildToJointTransform: [4x4 double]
```

Set transformations of the joint attachment between bodies. Each transformation is based on the dimensions of the previous rigid body length (*z*-axis). Offset the arm joints in the *x*-axis to avoid collisions during rotation.

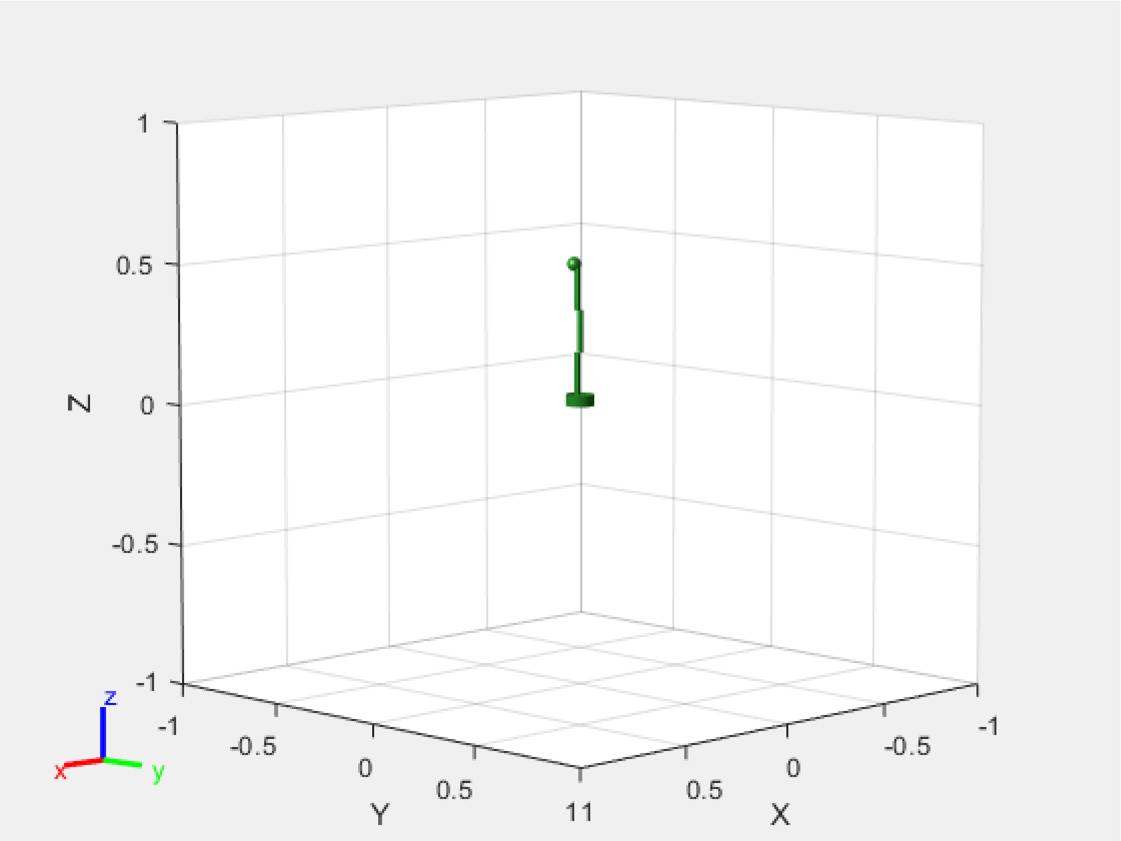


```
setFixedTransform(jnt1,trvec2tform([0.015 0 0.04]))
setFixedTransform(jnt2,trvec2tform([-0.015 0 0.15]))
setFixedTransform(jnt3,trvec2tform([0.015 0 0.15]))
setFixedTransform(jntGripper,trvec2tform([0 0 0.15]))
```

Assemble Robot

Create an object array for both the bodies and joints, including the original base. Add each joint to the body, and then add the body to the tree. Visualize each step.

```
bodies = {base,rotatingBase,arm1,arm2,arm3,gripper};
joints = {[],jntBase,jnt1,jnt2,jnt3,jntGripper};

figure("Name","Assemble Robot","Visible","on")
for i = 2:length(bodies) % Skip base. Iterate through adding bodies and joints.
            bodies{i}.Joint = joints{i};
            addBody(robot,bodies{i},bodies{i-1}.Name)
            show(robot,"Collisions","on","Frames","off");
            drawnow;
end
```



Call the showdetails function to view a list of the final tree information. Ensure the parent-child relationship and joint types are correct.

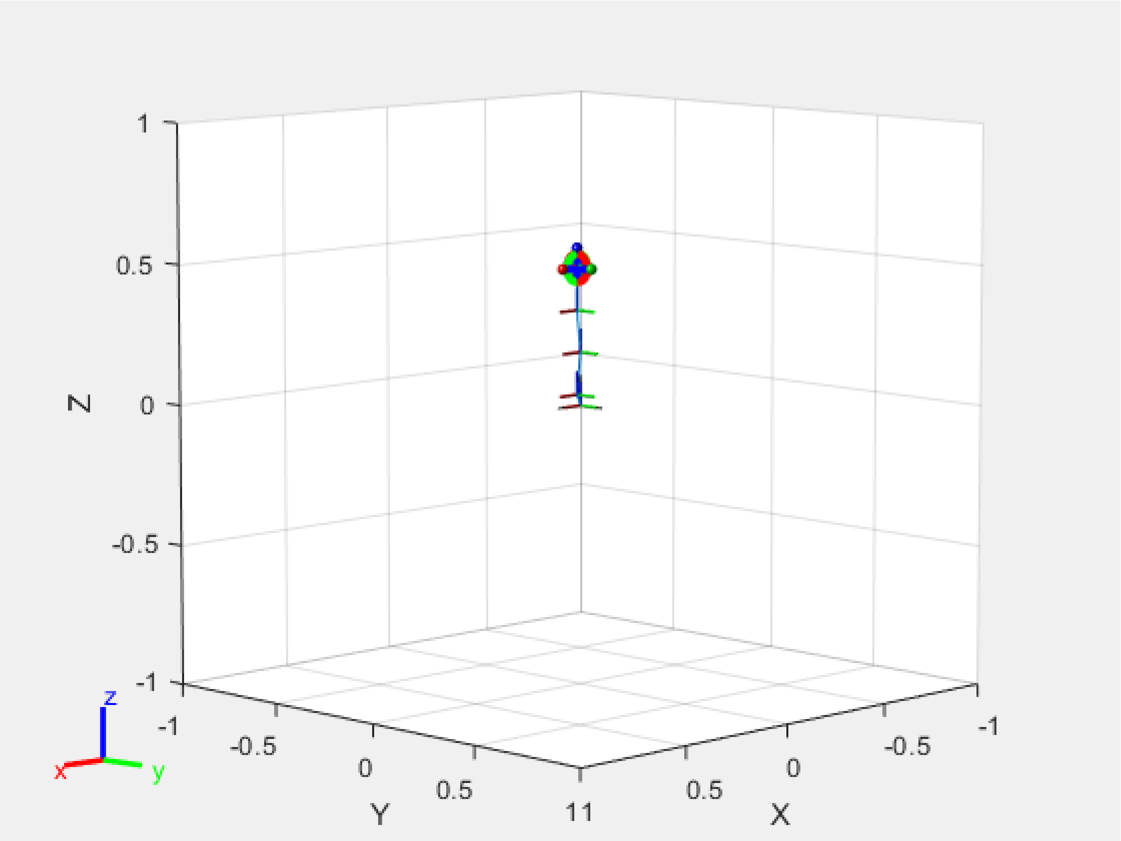
```
showdetails(robot)
```

```
--------------------
Robot: (5 bodies)

 Idx            Body Name           Joint Name           Joint Type            Parent Name(Idx)
 ---            ---------           ----------           ----------            ----------------
   1        rotating_base           base_joint             revolute                     base(0)
   2                 arm1                 jnt1             revolute            rotating_base(1)
   3                 arm2                 jnt2             revolute                     arm1(2)
   4                 arm3                 jnt3             revolute                     arm2(3)
   5              gripper        gripper_joint             revolute                     arm3(4)
--------------------
```

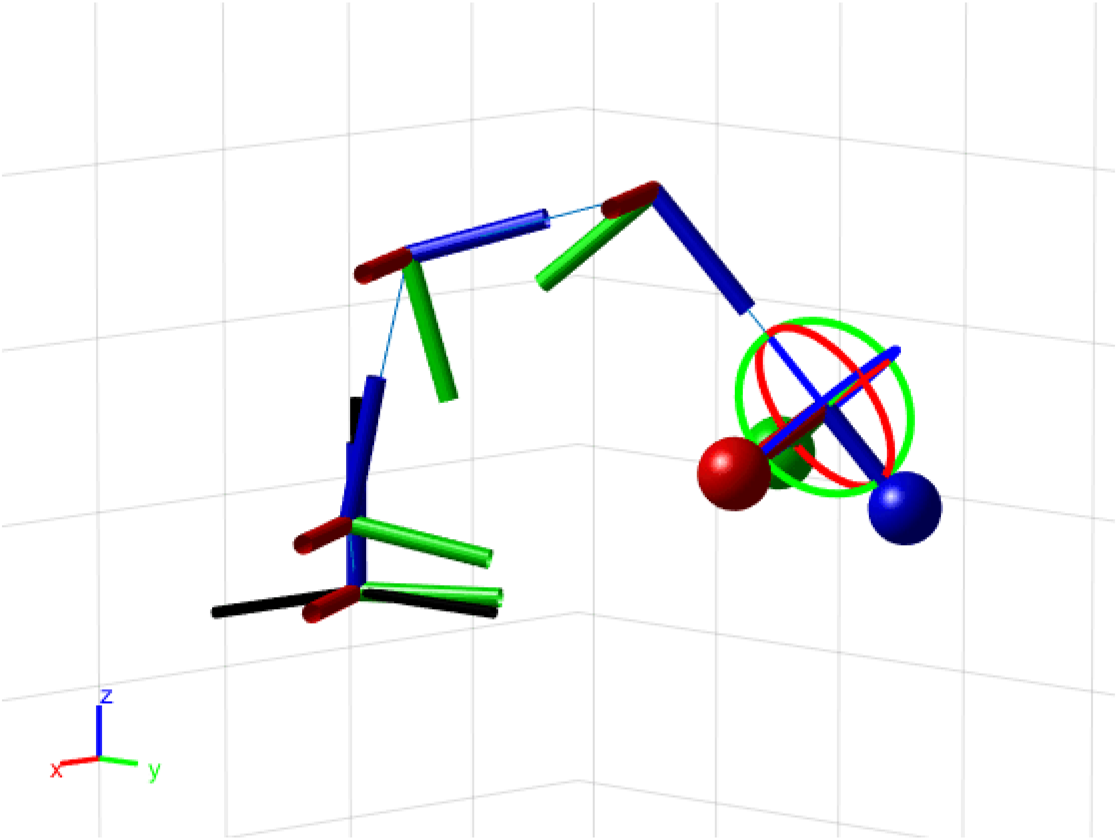
Interact With Robot Model

Visualize the robot model to confirm the dimensions using the interactiveRigidBodyTree object. Use the interactive GUI to move the model around. You may select specific bodies and set their joint position. To interact with more detailed models provided with Robotics System Toolbox™, see **Load Predefined Robot Models** or the loadrobot function.

```
figure("Name","Interactive GUI")
gui = interactiveRigidBodyTree(robot,"MarkerScaleFactor",0.25);
```



Move the interactive marker around to test different desired gripper positions. The GUI uses inverse kinematics to generate the best joint configuration. For more information about the interactive GUI, see `interactiveRigidBodyTree` object.



Next Steps

Now that you built your model in MATLAB®, you may want to do many different things.

- Perform “Inverse Kinematics” to get joint configurations based on desired end-effector positions. Specify additional robot constraints other than the model parameters including aiming constraints, Cartesian bounds, or pose targets.
- Generate “Trajectory Generation and Following” based on waypoints and other parameters with trapezoidal velocity profiles, B-splines, or polynomial trajectories.
- Peform “Manipulator Motion Planning” utilizing your robot models and an rapidly-exploring random tree (RRT) path planner.
- Check for “Collision Detection” with obstacles in your environment to ensure safe and effective motion of your robot.

Interactively Build a Trajectory for an ABB YuMi Robot

This example shows how to use the `interactiveRigidBodyTree` object to move a robot, design a trajectory, and replay it.

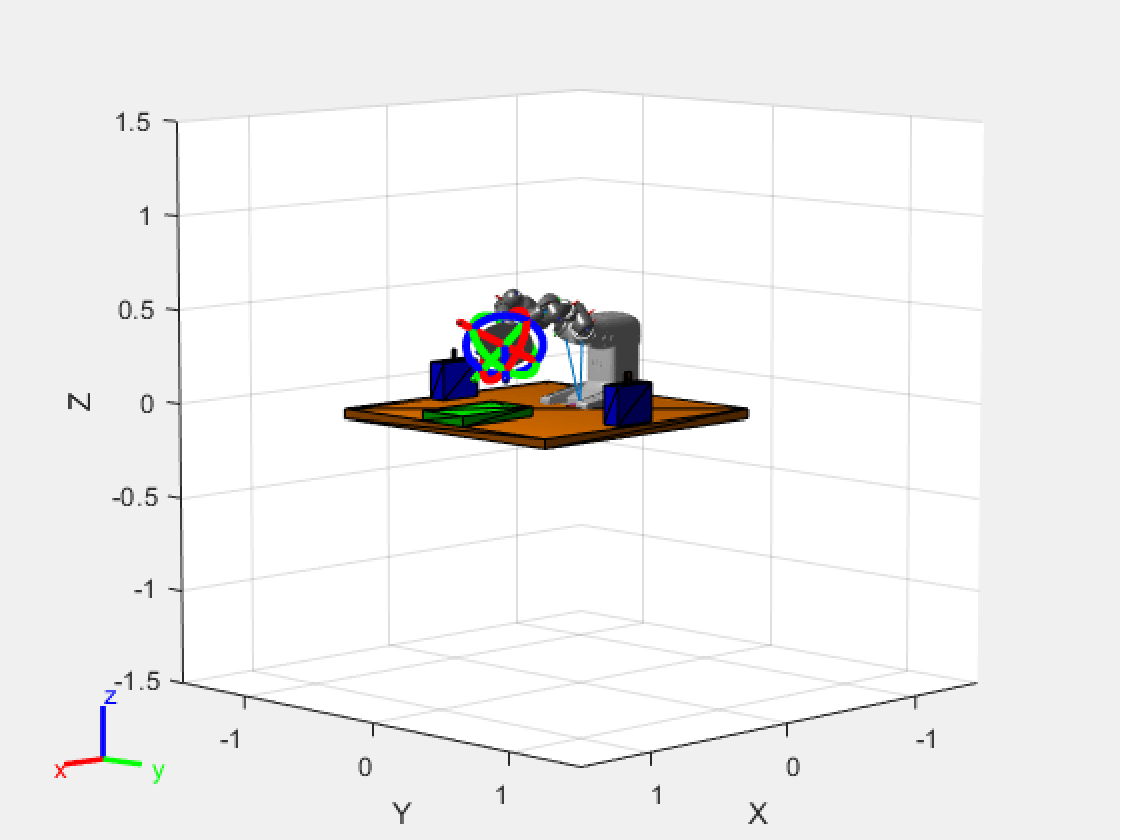
Load Robot Visualization and Build Environment

Load the `'abbYumi'` robot model. Initialize the interactive figure using `interactiveRigidBodyTree`. Save the current axes.

```
robot = loadrobot('abbYumi', 'Gravity', [0 0 -9.81]);
iviz = interactiveRigidBodyTree(robot);
ax = gca;
```

Build an environment consisting of a collision boxes that represent a floor, two shelves with objects, and a center table.

```
plane = collisionBox(1.5,1.5,0.05);
plane.Pose = trvec2tform([0.25 0 -0.025]);
show(plane,'Parent', ax);

leftShelf = collisionBox(0.25,0.1,0.2);
leftShelf.Pose = trvec2tform([0.3 -.65 0.1]);
[~, patchObj] = show(leftShelf,'Parent',ax);
patchObj.FaceColor = [0 0 1];

rightShelf = collisionBox(0.25,0.1,0.2);
rightShelf.Pose = trvec2tform([0.3 .65 0.1]);
[~, patchObj] = show(rightShelf,'Parent',ax);
patchObj.FaceColor = [0 0 1];

leftWidget = collisionCylinder(0.01, 0.07);
leftWidget.Pose = trvec2tform([0.3 -0.65 0.225]);
[~, patchObj] = show(leftWidget,'Parent',ax);
patchObj.FaceColor = [1 0 0];

rightWidget = collisionBox(0.03, 0.02, 0.07);
rightWidget.Pose = trvec2tform([0.3 0.65 0.225]);
[~, patchObj] = show(rightWidget,'Parent',ax);
patchObj.FaceColor = [1 0 0];

centerTable = collisionBox(0.5,0.3,0.05);
centerTable.Pose = trvec2tform([0.75 0 0.025]);
[~, patchObj] = show(centerTable,'Parent',ax);
patchObj.FaceColor = [0 1 0];
```



Interactively Generate Configurations

Use the interactive visualization to move the robot around and set configurations. When the figure is initialized, the robot is in its home configuration with the arms crossed. Zoom in and click on an end effector to get more information.

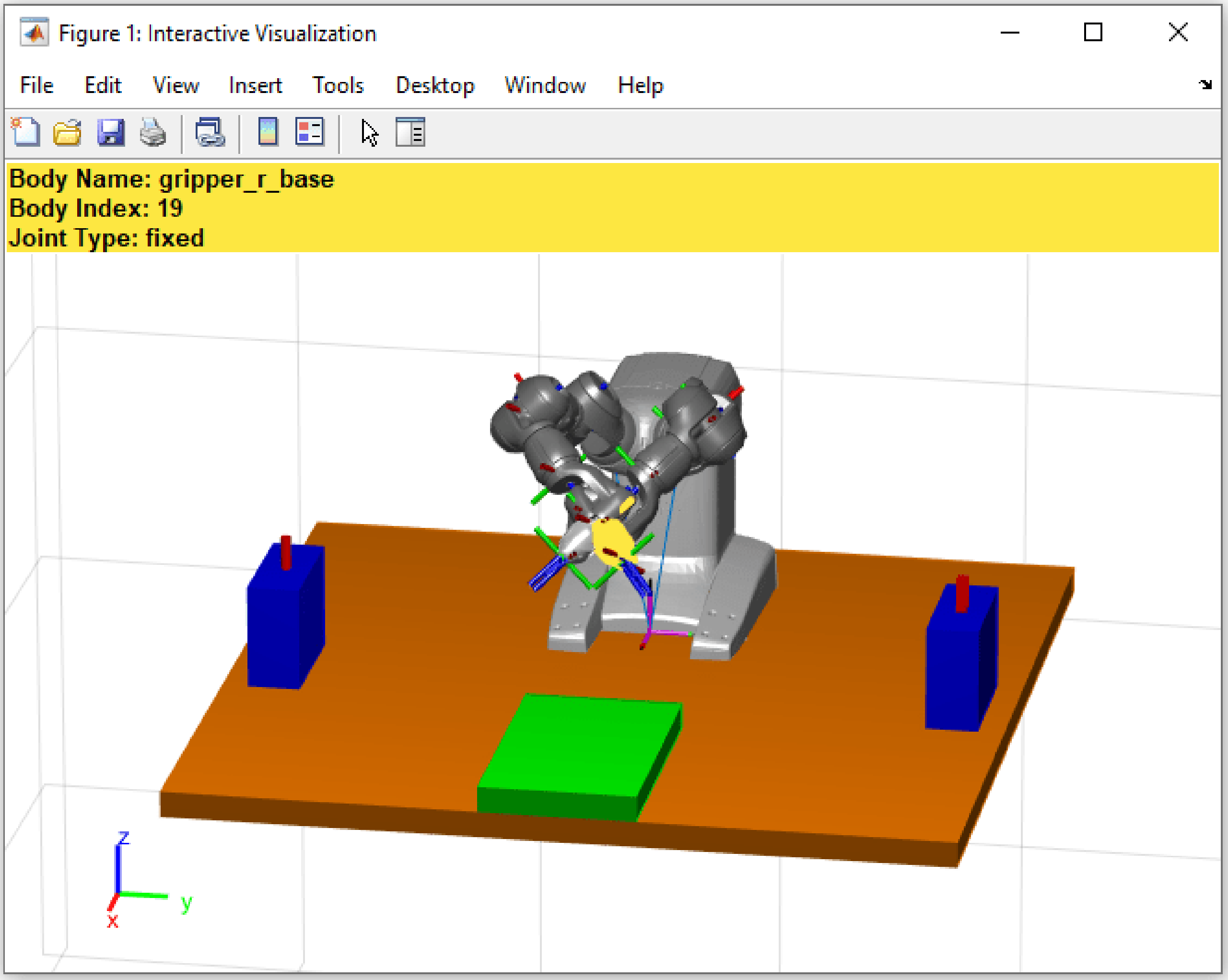
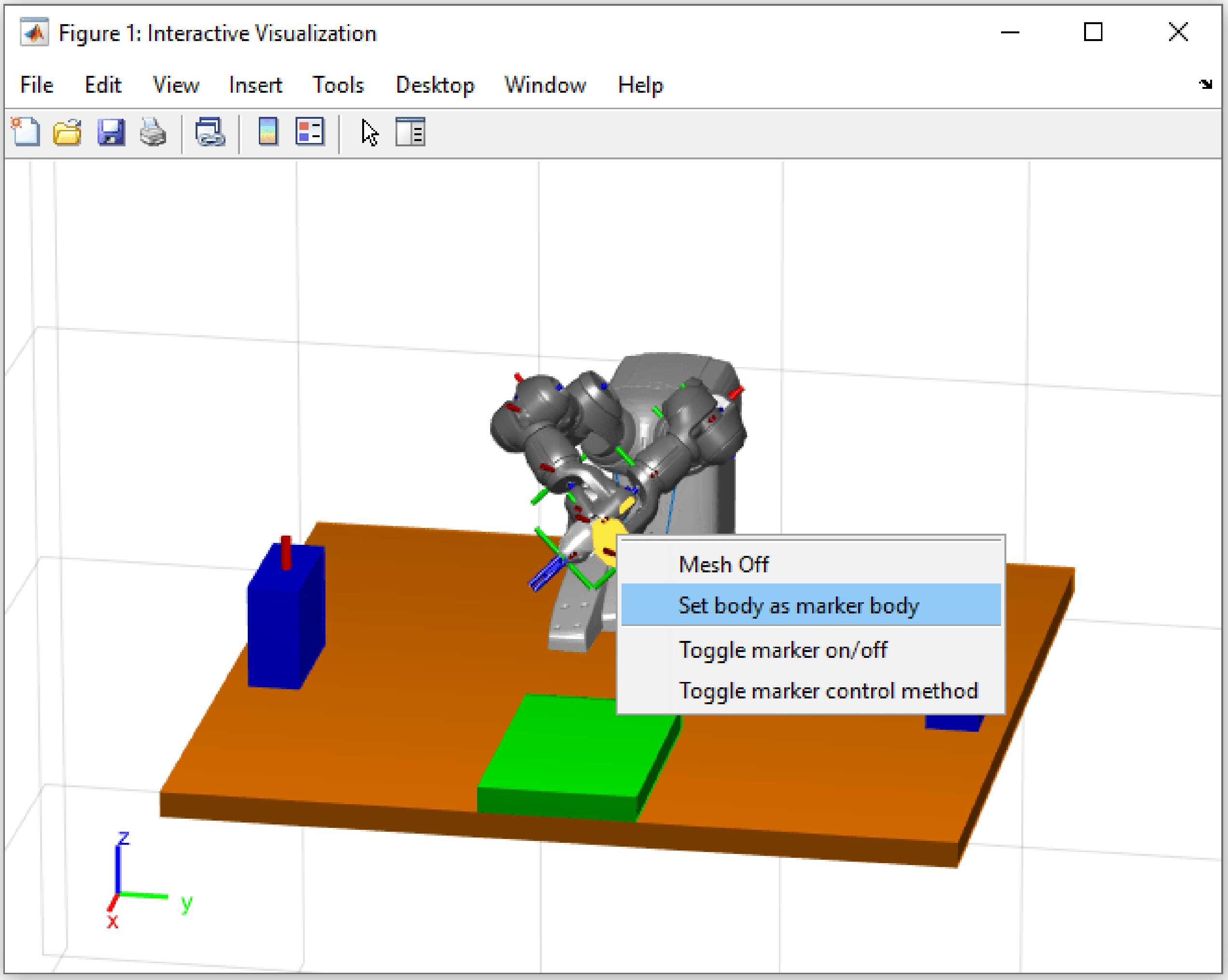


Figure 1: Interactive Visualization
File Edit View Insert Tools Desktop Window Help
z
y
x

Body Name: gripper_r_base
Body Index: 19
Joint Type: fixed

To select the body as the end effector, right-click on the body to select it.



The marker body can also be assigned from the command line:

```
iviz.MarkerBodyName = "gripper_r_base";
```

Once the body has been set, use the provided marker elements to move the marker around, and the selected body follows. Dragging the central gray marker moves the marker in Cartesian space. The red, green, and blue axes move the marker along the *xyz*-axes. The circles rotate the marker about the axes of equivalent color.

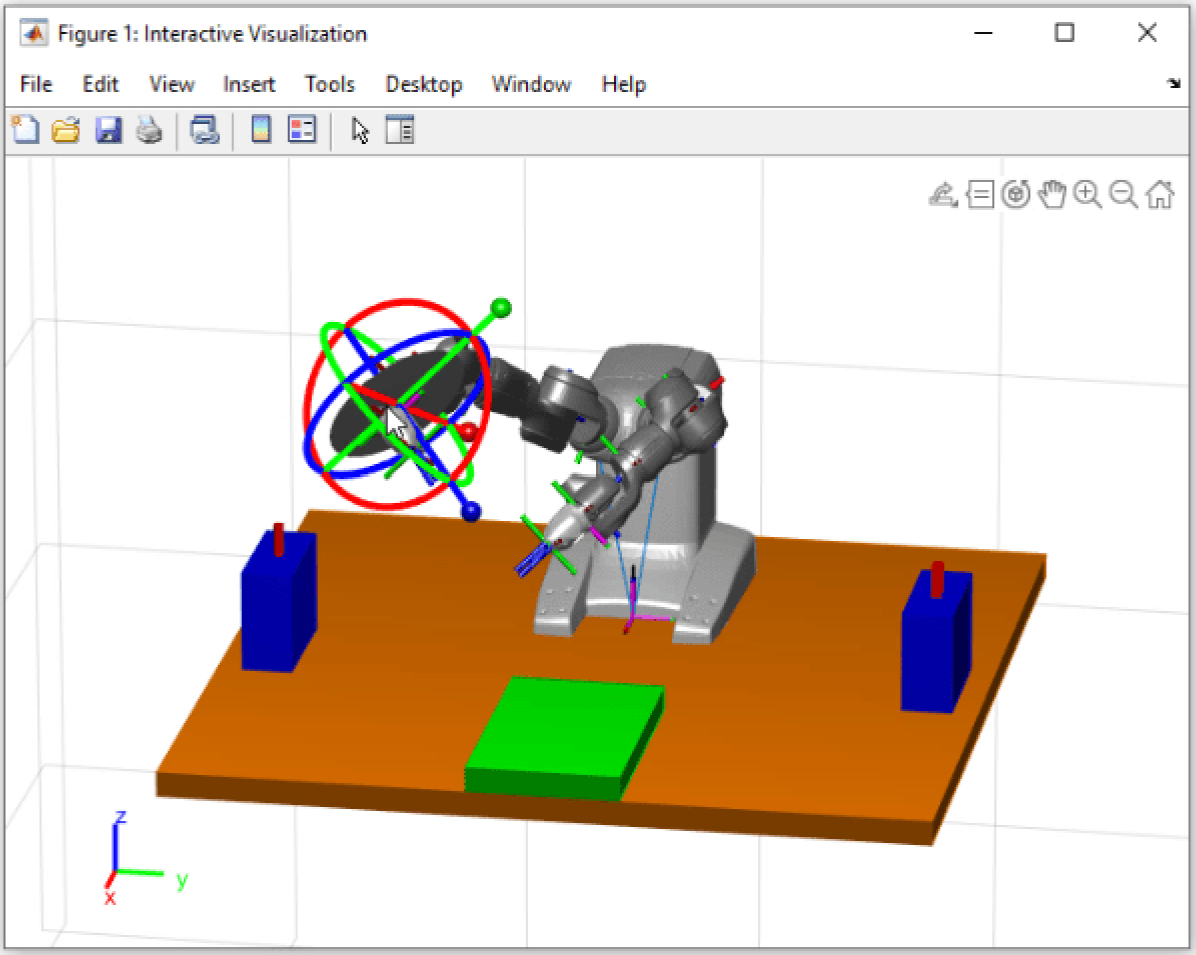


Figure 1: Interactive Visualization
File Edit View Insert Tools Desktop Window Help
z
y
x

You can also move individual joints by right-clicking the joint and click **Toggle marker control method.**

The `MarkerControlMethod` property of the object is set to `"JointControl"`.

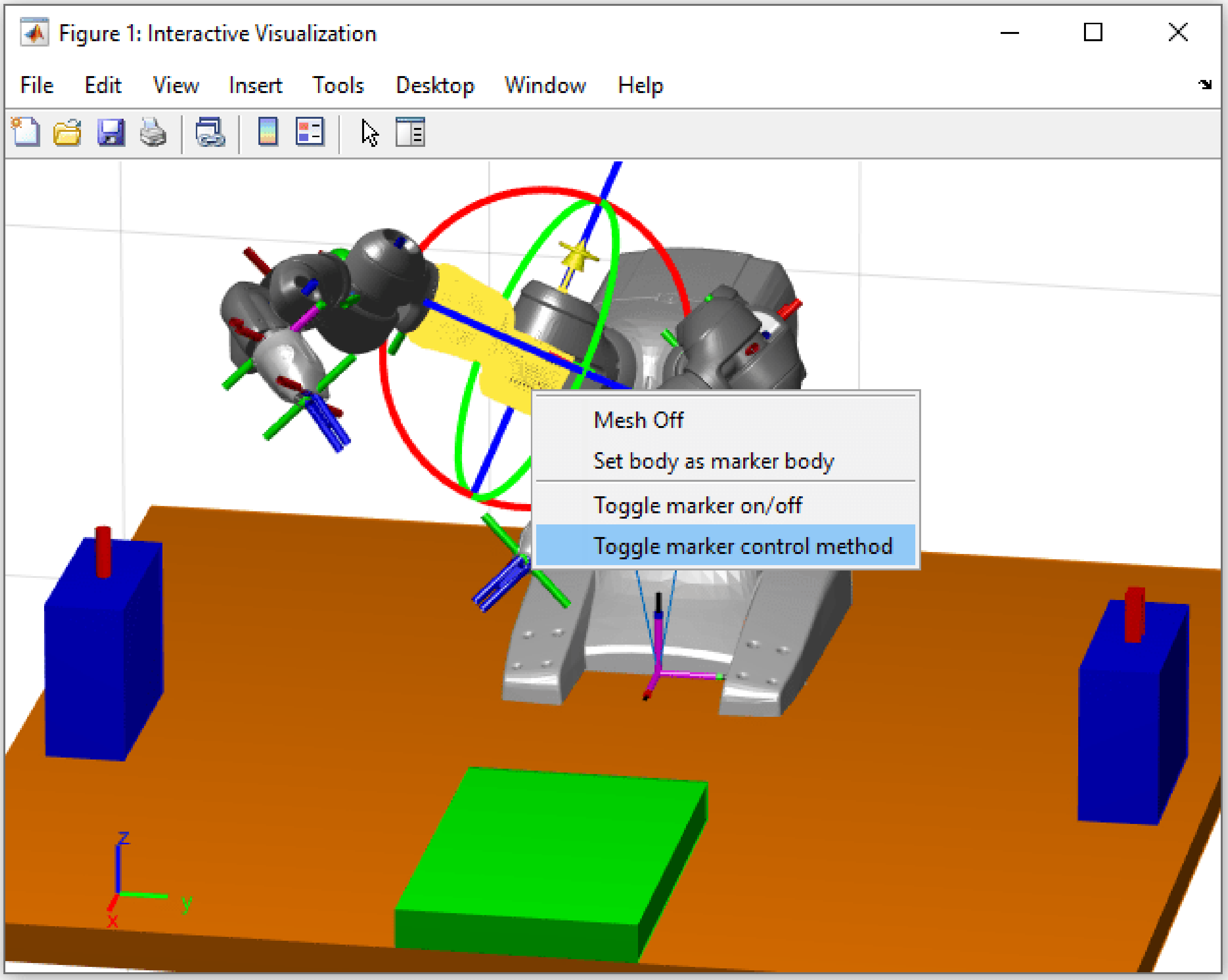
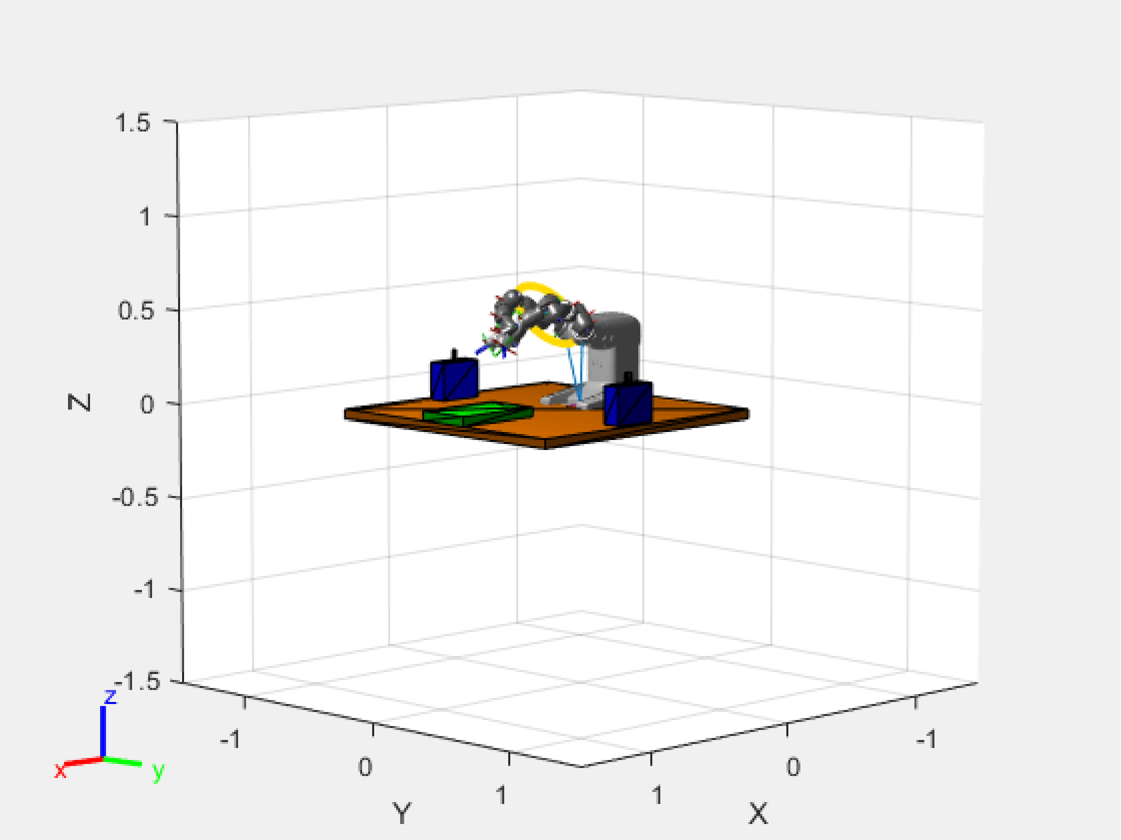


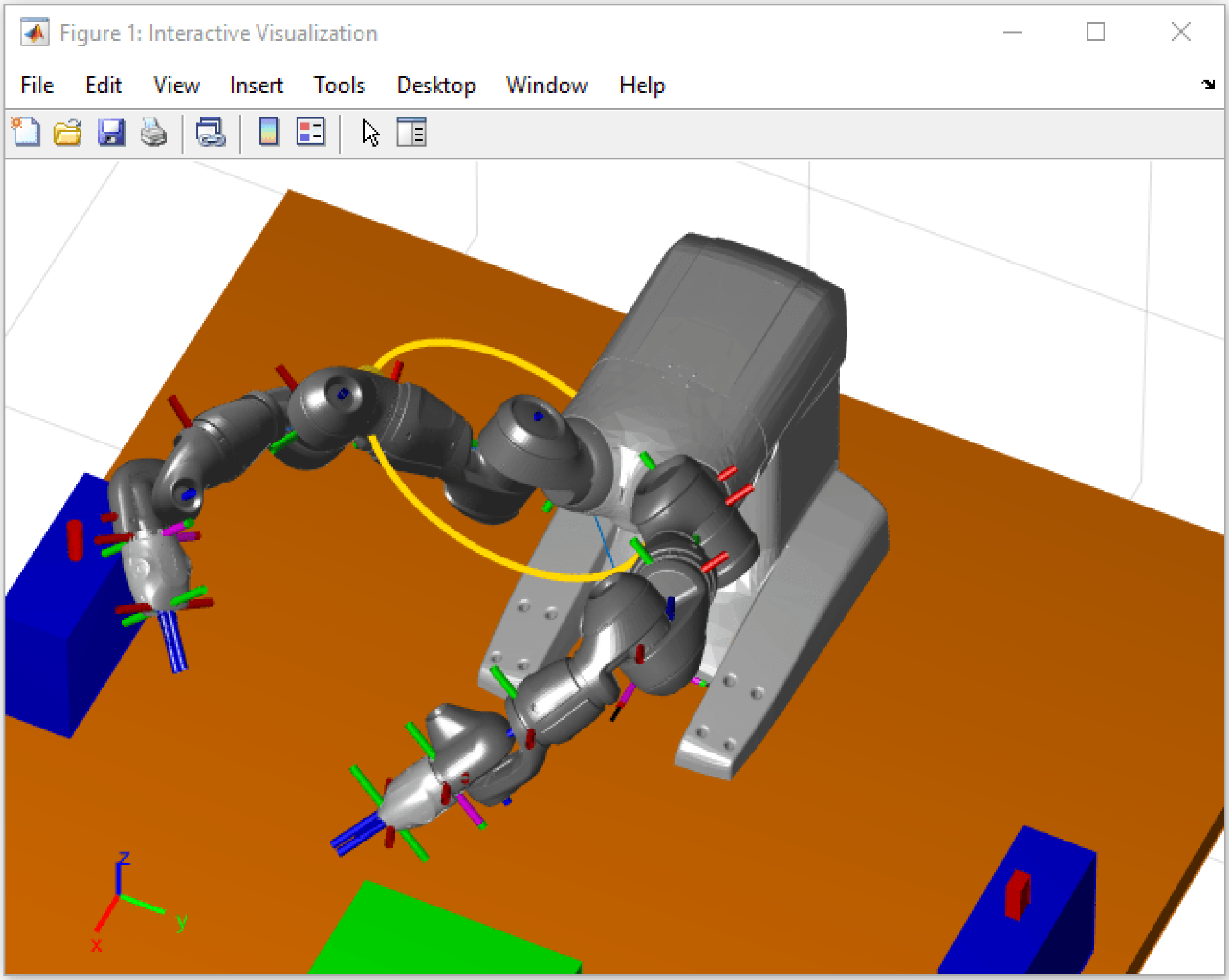
Figure 1: Interactive Visualization
File Edit View Insert Tools Desktop Window Help
Mesh Off
Set body as marker body
Toggle marker on/off
Toggle marker control method
z
y
x

These steps can also be accomplished by changing properties on the object directly.

```
iviz.MarkerBodyName = "yumi_link_2_r";
iviz.MarkerControlMethod = "JointControl";
```



Changing to joint control produces a yellow marker that allows the joint position to be set directly.



Iteractively move the robot around until you have a desired configuration. Save configurations using `addConfiguration`. Each call adds the current configuration to the `StoredConfigurations` property.

```
addConfiguration(iviz)
```

Define Waypoints for a Trajectory

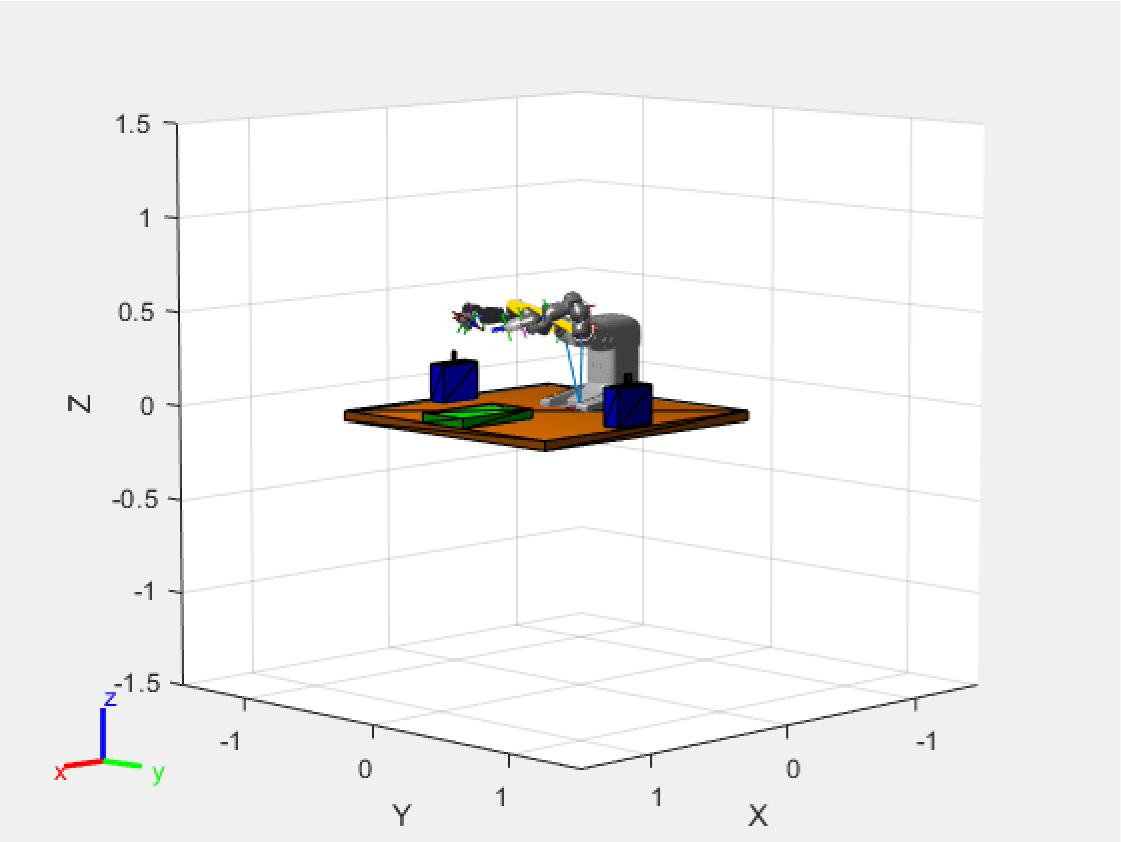
For the purpose of this example, a set of configurations are provided in a `.mat` file.

Load the configurations, and specify them as the set of stored configurations. The first configuration is added by updating the `Configuration` property and calling `addConfiguration`, which you could do interactively, but the rest are simply added by assigning the `StoredConfigurations` property directly.

```
load abbYumiSaveTrajectoryWaypts.mat
```

```
removeConfigurations(iviz) % Clear stored configurations
```

```
% Start at a valid starting configuration
iviz.Configuration = startingConfig;
```



```
addConfiguration(iviz)

% Specify the entire set of waypoints
iviz.StoredConfigurations = [startingConfig, ...
                             graspApproachConfig, ...
                             graspPoseConfig, ...
                             graspDepartConfig, ...
                             placeApproachConfig, ...
                             placeConfig, ...
                             placeDepartConfig, ...
                             startingConfig];
```

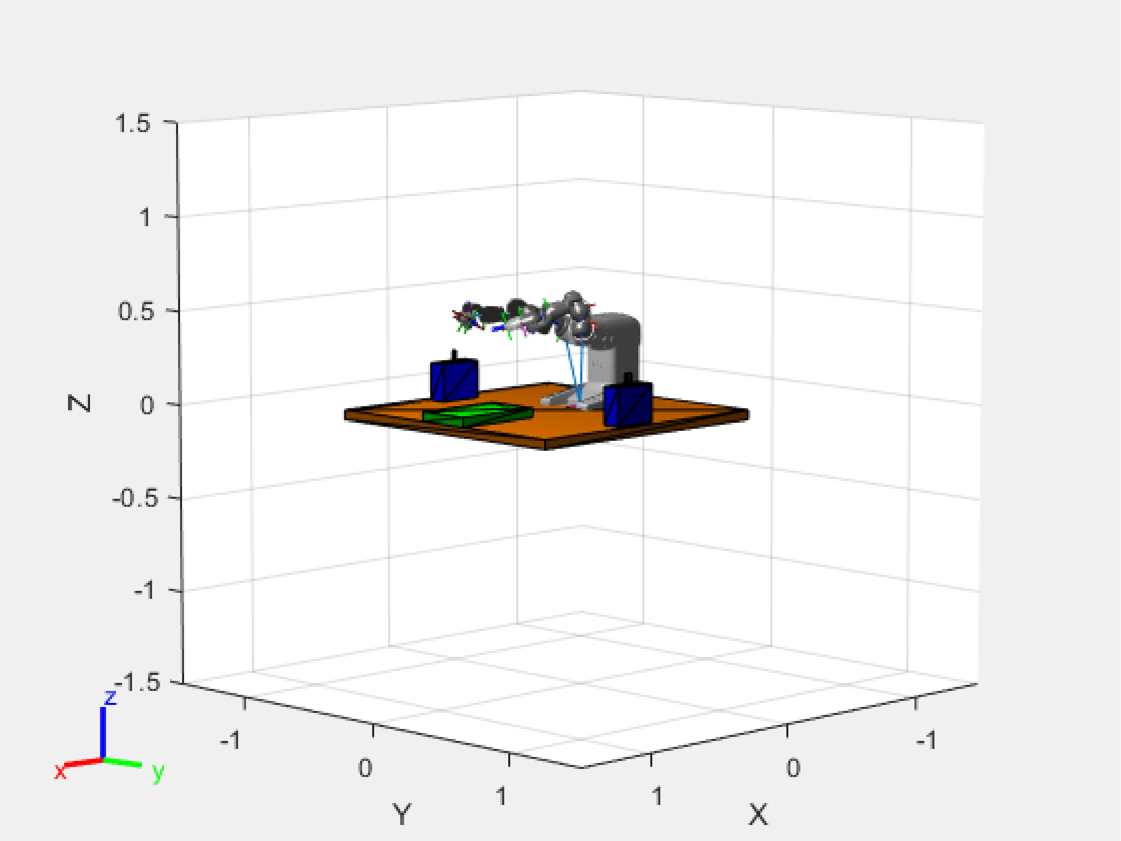
Generate the Trajectory and Play It Back

Once all the waypoints have been stored, construct a trajectory that the robot follows. For this example, a trapezoidal velocity profile is generated using `trapveltraj`. A trapezoidal velocity profile means the robot stops smoothly at each waypoint, but achieves a set max speed while in motion.

```
numSamples = 100*size(iviz.StoredConfigurations, 2) + 1;
[q,~,~, tvec] = trapveltraj(iviz.StoredConfigurations,numSamples,'EndTime',2);
```

Replay the generated trajectory by iterating the generated `q` matrix, which represents a series of joint configurations that move between each waypoint. In this case, a rate control object is used to ensure that the play back speed is reflective of the actual execution speed.

```
iviz.ShowMarker = false;
showFigure(iviz)
rateCtrlObj = rateControl(numSamples/(max(tvec) + tvec(2)));
for i = 1:numSamples
    iviz.Configuration = q(:,i);
    waitfor(rateCtrlObj);
end
```



The figure shows the robot executes a smooth trajectory between all the defined waypoints.

Build Manipulator Robot Using Kinematic DH Parameters

Use the Denavit-Hartenberg (DH) parameters of the Puma560® manipulator robot to incrementally build a rigid body tree robot model. Specify the relative DH parameters for each joint as you attach them. Visualize the robot frames, and interact with the final model.

The DH parameters define the geometry of how each rigid body attaches to its parent via a joint. The parameters follow a four transformation convention:

- A — Length of the common normal line between the two z-axes, which is perpendicular to both axes
- α — Angle of rotation for the common normal
- d — Offset along the z-axis in the normal direction, from parent to child
- θ — Angle of rotation for the x-axis along the previous z-axis

Specify the parameters for the Puma560 robot [1] on page 1-0 as a matrix. Values come from .

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

Create a rigid body tree object.

```
robot = rigidBodyTree;
```

Create a cell array for the rigid body object, and another for the joint objects. Iterate through the DH parameters performing this process:

1. Create a `rigidBody` object with a unique name.
2. Create and name a revolute `rigidBodyJoint` object.
3. Use `setFixedTransform` to specify the body-to-body transformation of the joint using DH parameters. The function ignores the final element of the DH parameters, `theta`, because the angle of the body is dependent on the joint position.
4. Use `addBody` to attach the body to the rigid body tree.

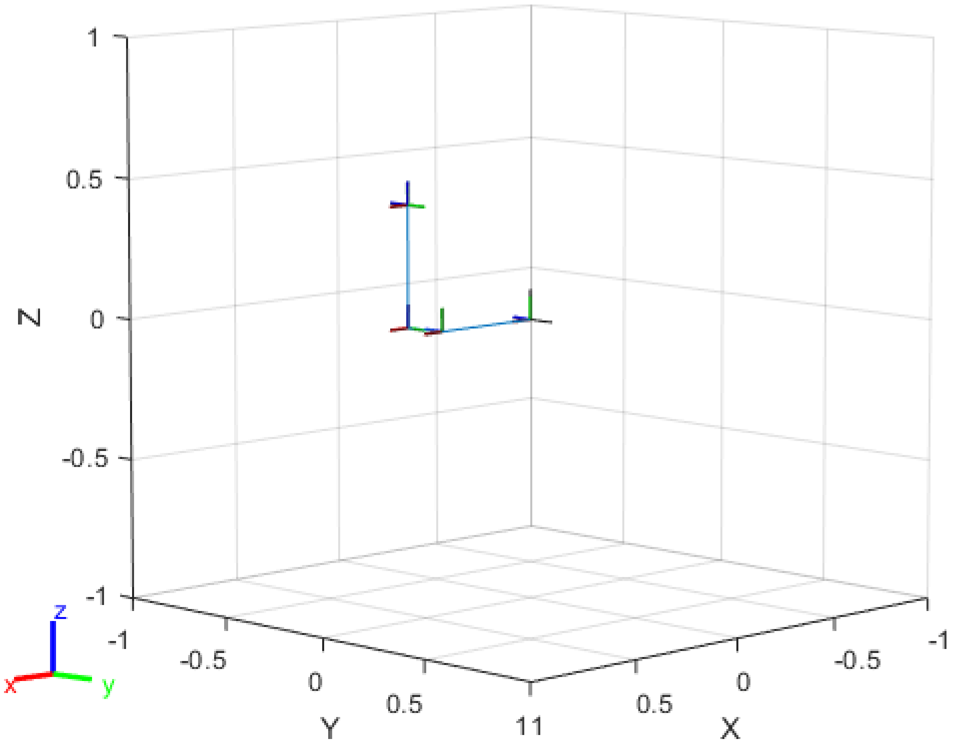
```
bodies = cell(6,1);
joints = cell(6,1);
for i = 1:6
    bodies{i} = rigidBody(['body' num2str(i)]);
    joints{i} = rigidBodyJoint(['jnt' num2str(i)],"revolute");
    setFixedTransform(joints{i},dhparams(i,:),"dh");
    bodies{i}.Joint = joints{i};
    if i == 1 % Add first body to base
        addBody(robot,bodies{i},"base")
    else % Add current body to previous body by name
        addBody(robot,bodies{i},bodies{i-1}.Name)
    end
end
```

Verify that your robot has been built properly by using the `showdetails` or `show` function. The `showdetails` function lists all the bodies of the robot in the MATLAB® command window. The `show` function displays the robot with a specified configuration (home by default).

```
showdetails(robot)
```

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

 Idx     Body Name   Joint Name   Joint Type     Parent Name(Idx)   Children Name(s)
 ---     ---------   ----------   ----------     ----------------   ----------------
   1         body1         jnt1     revolute              base(0)   body2(2)
   2         body2         jnt2     revolute             body1(1)   body3(3)
   3         body3         jnt3     revolute             body2(2)   body4(4)
   4         body4         jnt4     revolute             body3(3)   body5(5)
   5         body5         jnt5     revolute             body4(4)   body6(6)
   6         body6         jnt6     revolute             body5(5)   
--------------------
```

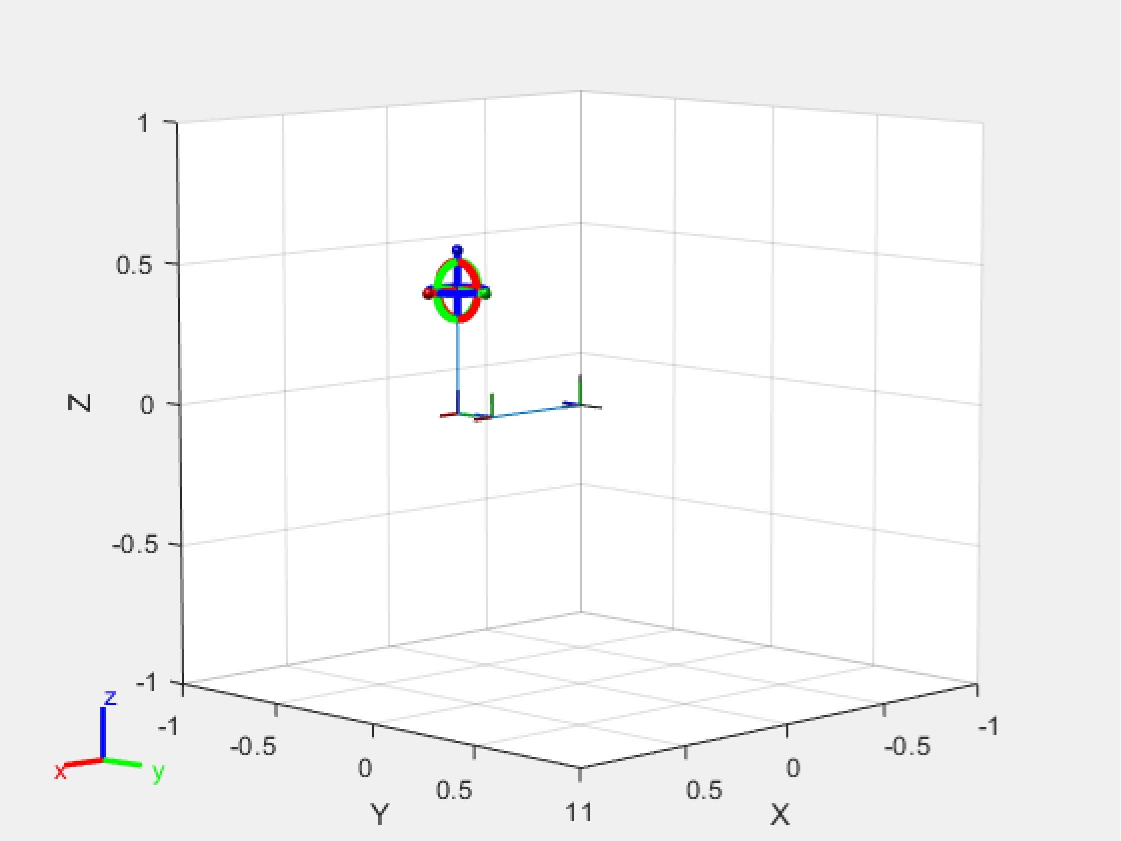
```
figure(Name="PUMA Robot Model")
show(robot);
```



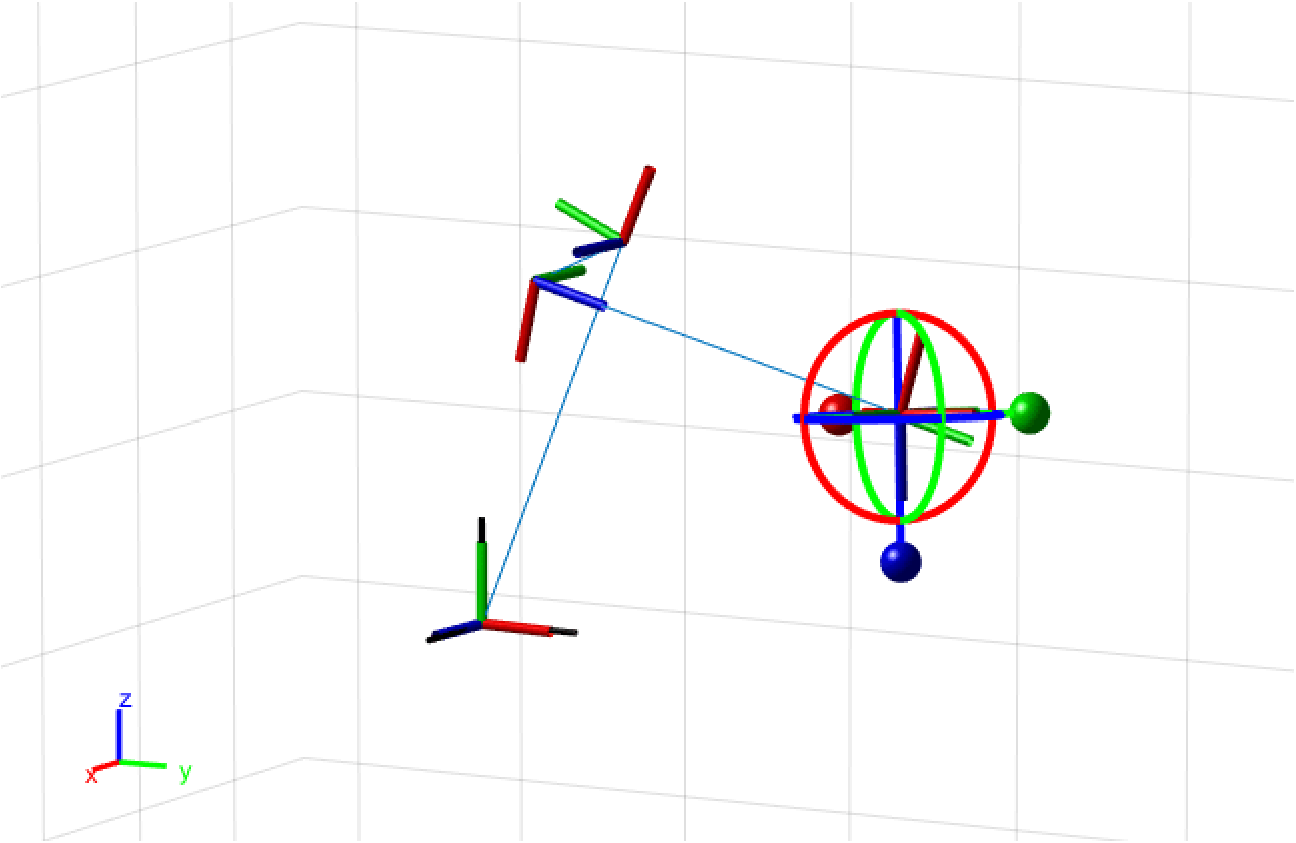
Interact with Robot Model

Visualize the robot model to confirm its dimensions by using the `interactiveRigidBodyTree` object.

```
figure(Name="Interactive GUI")
gui = interactiveRigidBodyTree(robot,MarkerScaleFactor=0.5);
```



Click and drag the marker in the interactive GUI to reposition the end effector. The GUI uses inverse kinematics to solve for the joint positions that achieve the best possible match to the specified end-effector position. Right-click a specific body frame to set it as the target marker body, or to change the control method for setting specific joint positions.



Next Steps

Now that you have built your model in MATLAB®, these are some possible next steps.

- Perform "Inverse Kinematics" to get joint configurations based on desired end-effector positions. Specify robot constraints in addition to those of the model parameters, including aiming constraints, Cartesian bounds, and pose targets.
- "Trajectory Generation and Following", based on waypoints and other parameters, with trapezoidal velocity profiles, B-splines, or polynomial trajectories.
- Peform "Manipulator Motion Planning" utilizing your robot models and a rapidly-exploring random tree (RRT) path planner.
- Use "Collision Detection" with obstacles in your environment to ensure safe and effective motion for your robot.

References

[1] Corke, P. I., and B. Armstrong-Helouvry. "A Search for Consensus Among Model Parameters Reported for the PUMA 560 Robot." *Proceedings of the 1994 IEEE International Conference on Robotics and Automation*, 1608–13. San Diego, CA, USA: IEEE Comput. Soc. Press, 1994. https://doi.org/10.1109/ROBOT.1994.351360.

Interactively Build A Robot Trajectory

Build a trajectory using interactive markers on the ABB YuMi robot model. The `interactiveRigidBodyTree` object displays the rigid body tree robot model in an interactive figure. This example shows how to move different parts of the robot, design a trajectory, and save configurations.

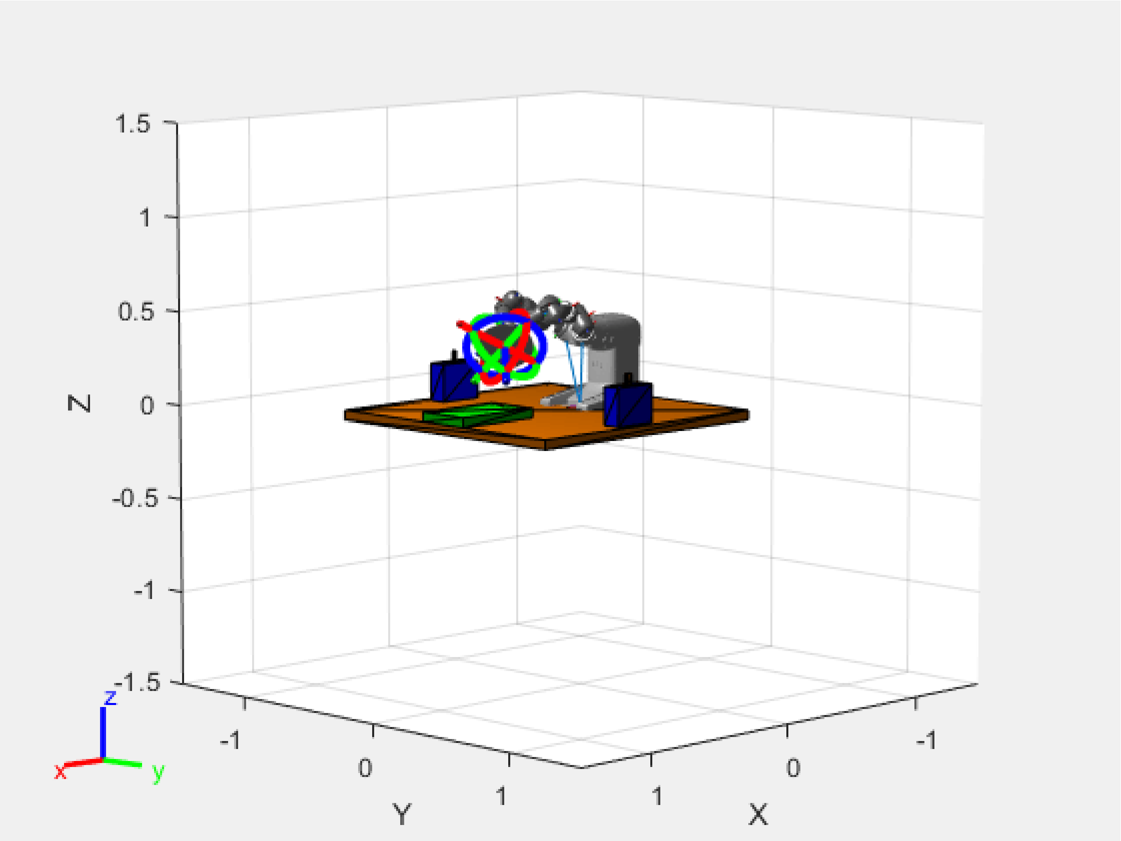
Load Robot Visualization and Build Environment

Load the `'abbYumi'` robot model. Initialize the interactive figure using `interactiveRigidBodyTree`. Save the current axes.

```
robot = loadrobot('abbYumi', 'Gravity', [0 0 -9.81]);
iviz = interactiveRigidBodyTree(robot);
ax = gca;
```

Build an environment consisting of a collision boxes that represent a floor, two shelves with objects, and a center table.

```
plane = collisionBox(1.5,1.5,0.05);
plane.Pose = trvec2tform([0.25 0 -0.025]);
show(plane,'Parent', ax);

leftShelf = collisionBox(0.25,0.1,0.2);
leftShelf.Pose = trvec2tform([0.3 -.65 0.1]);
[~, patchObj] = show(leftShelf,'Parent',ax);
patchObj.FaceColor = [0 0 1];

rightShelf = collisionBox(0.25,0.1,0.2);
rightShelf.Pose = trvec2tform([0.3 .65 0.1]);
[~, patchObj] = show(rightShelf,'Parent',ax);
patchObj.FaceColor = [0 0 1];

leftWidget = collisionCylinder(0.01, 0.07);
leftWidget.Pose = trvec2tform([0.3 -0.65 0.225]);
[~, patchObj] = show(leftWidget,'Parent',ax);
patchObj.FaceColor = [1 0 0];

rightWidget = collisionBox(0.03, 0.02, 0.07);
rightWidget.Pose = trvec2tform([0.3 0.65 0.225]);
[~, patchObj] = show(rightWidget,'Parent',ax);
patchObj.FaceColor = [1 0 0];

centerTable = collisionBox(0.5,0.3,0.05);
centerTable.Pose = trvec2tform([0.75 0 0.025]);
[~, patchObj] = show(centerTable,'Parent',ax);
patchObj.FaceColor = [0 1 0];
```



Interactively Generate Configurations

Use the interactive visualization to move the robot around and set configurations. When the figure is initialized, the robot is in its home configuration with the arms crossed. Zoom in and click on an end effector to get more information.

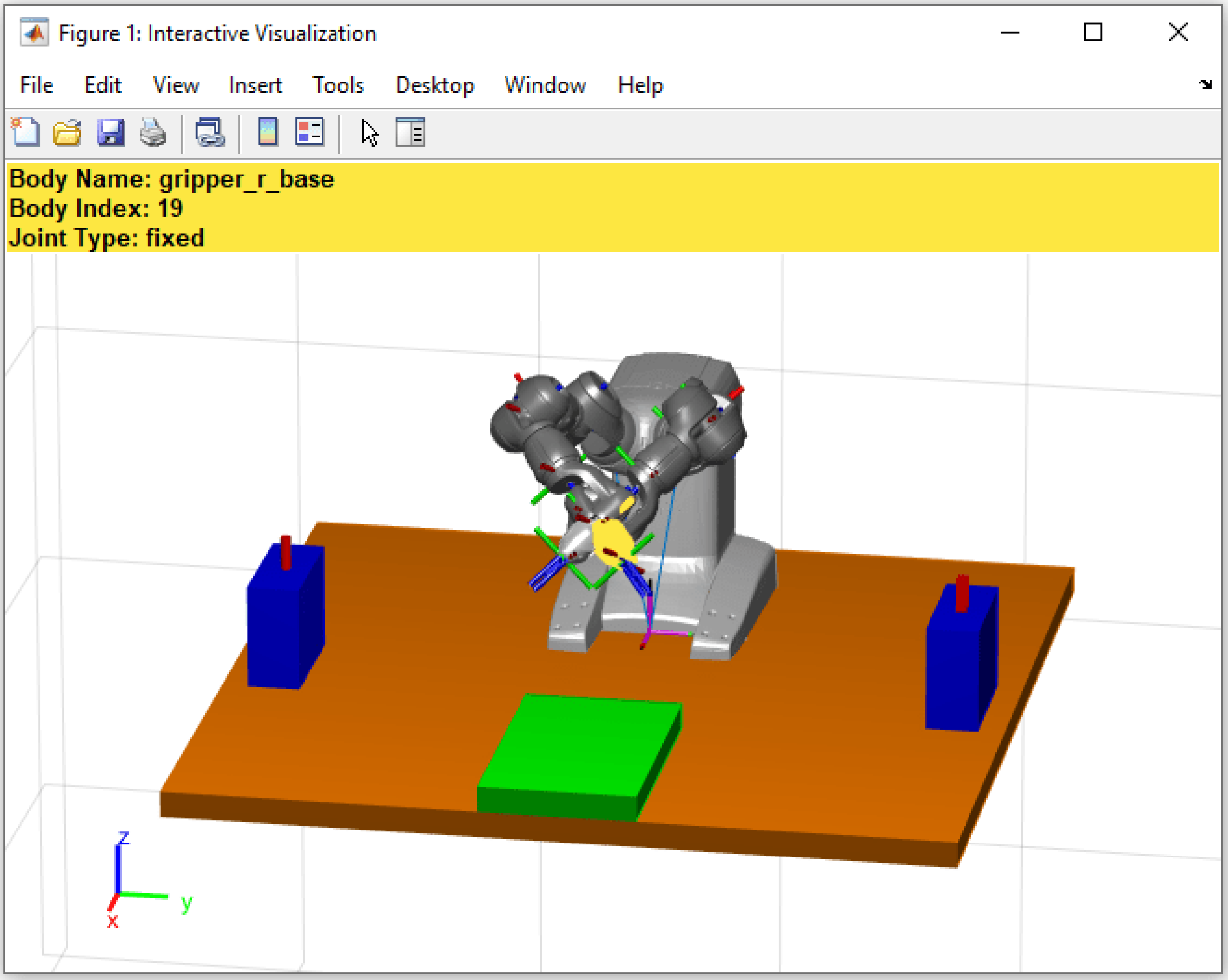


Figure 1: Interactive Visualization
File Edit View Insert Tools Desktop Window Help
z
y
x

Body Name: gripper_r_base
Body Index: 19
Joint Type: fixed

To select the body as the end effector, right-click on the body and choose **Set body as marker body**.

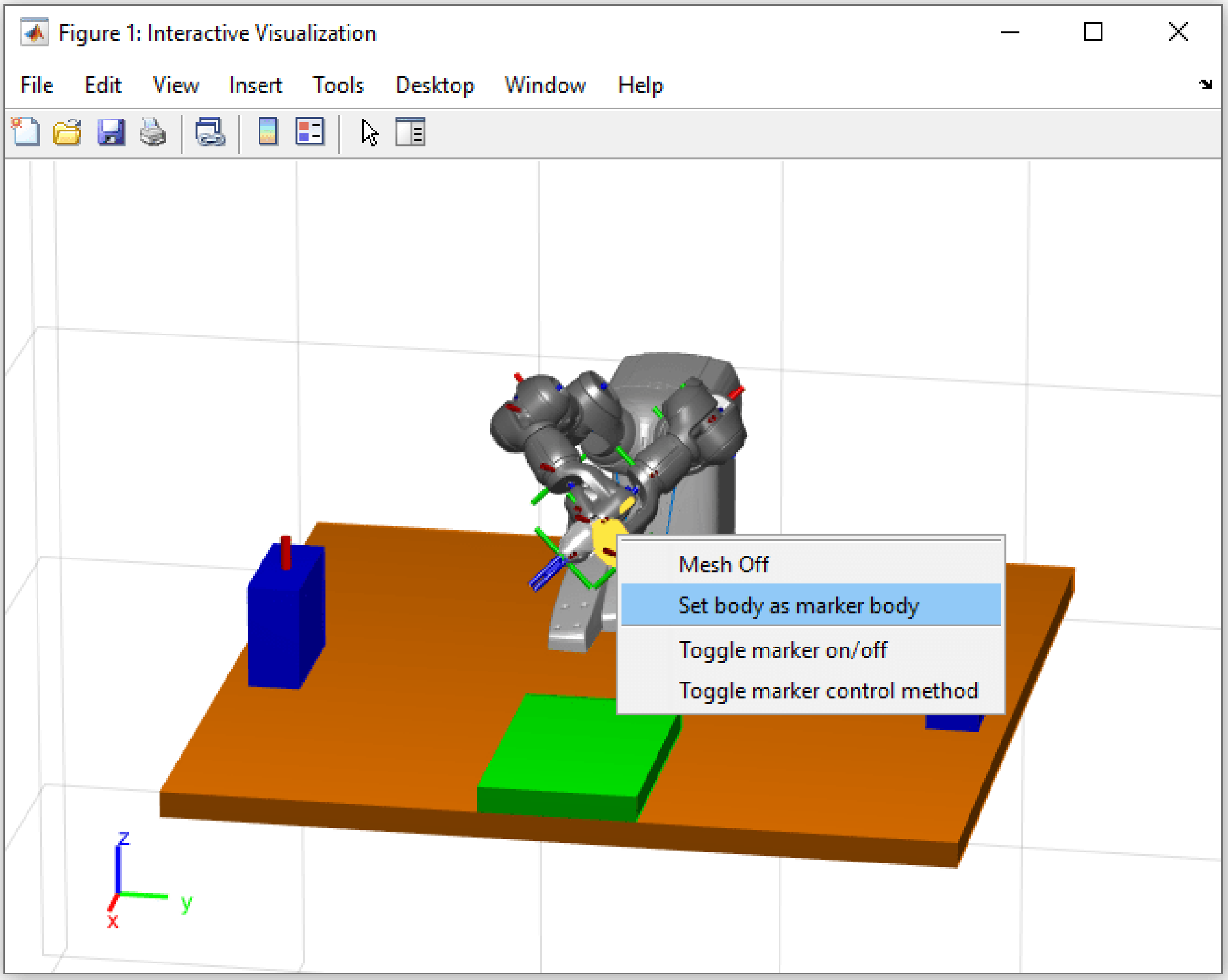
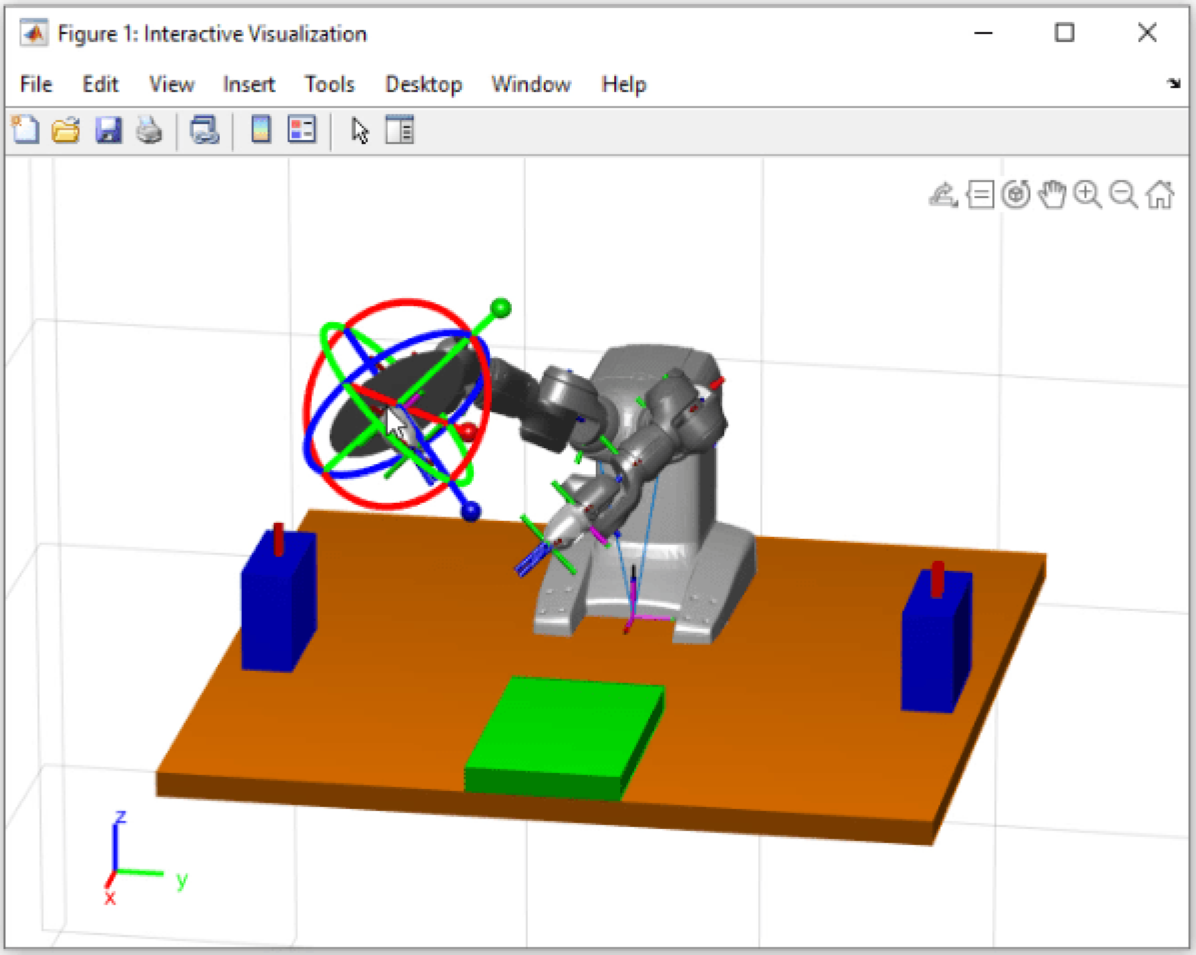


Figure 1: Interactive Visualization
File Edit View Insert Tools Desktop Window Help
Mesh Off
Set body as marker body
Toggle marker on/off
Toggle marker control method
z
y
x

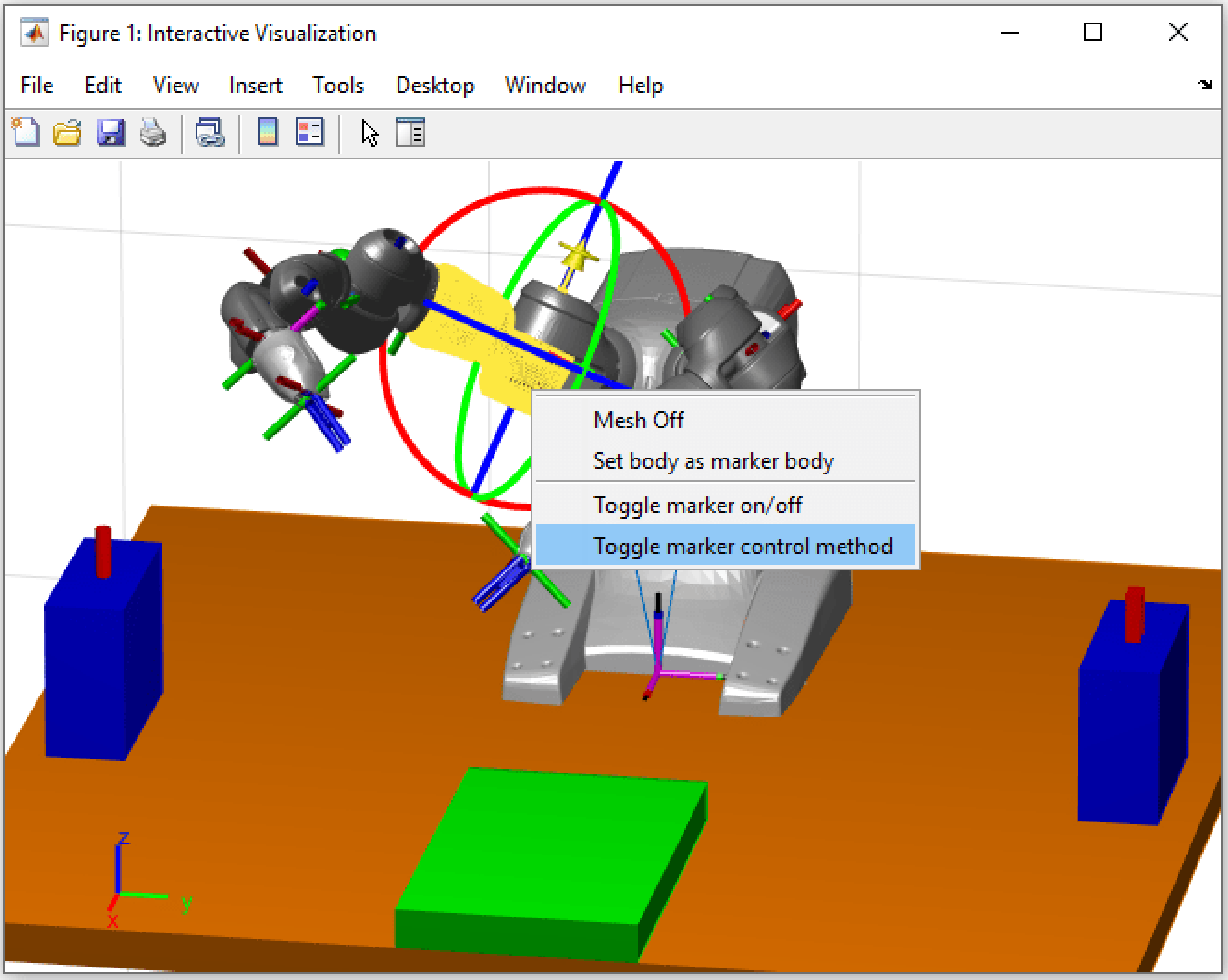
The marker body can also be assigned from the command line:

```
iviz.MarkerBodyName = "gripper_r_base";
```

Once the body has been set, use the provided marker elements to move the marker around, and the selected body follows. Dragging the central gray marker moves the marker in Cartesian space. The red, green, and blue axes move the marker along the *xyz*-axes. The circles rotate the marker about the axes of equivalent color.

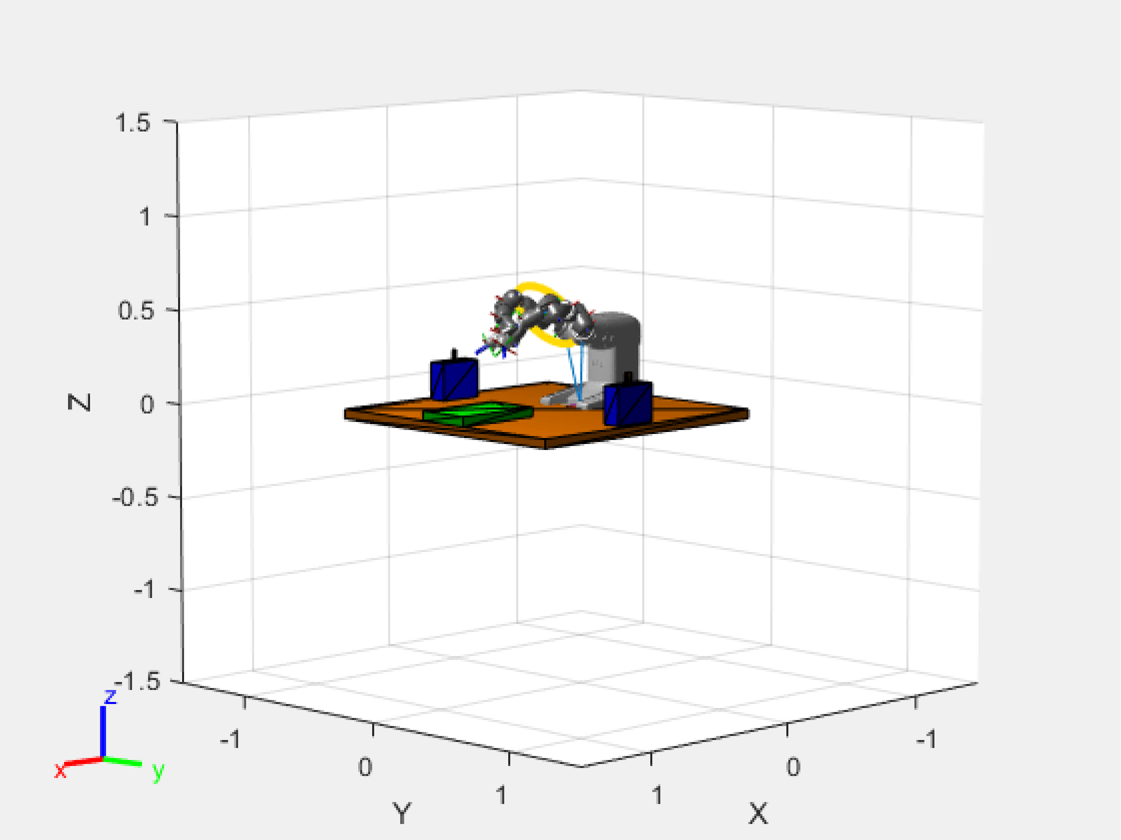


You can also move individual joints by right-clicking the joint and click **Toggle marker control method.**



These steps can also be accomplished by changing properties on the object directly. The `MarkerControlMethod` property of the object is set to `"JointControl"`.

```
iviz.MarkerBodyName = "yumi_link_2_r";
iviz.MarkerControlMethod = "JointControl";
```



Changing to joint control produces a yellow marker that allows the joint position to be set directly.

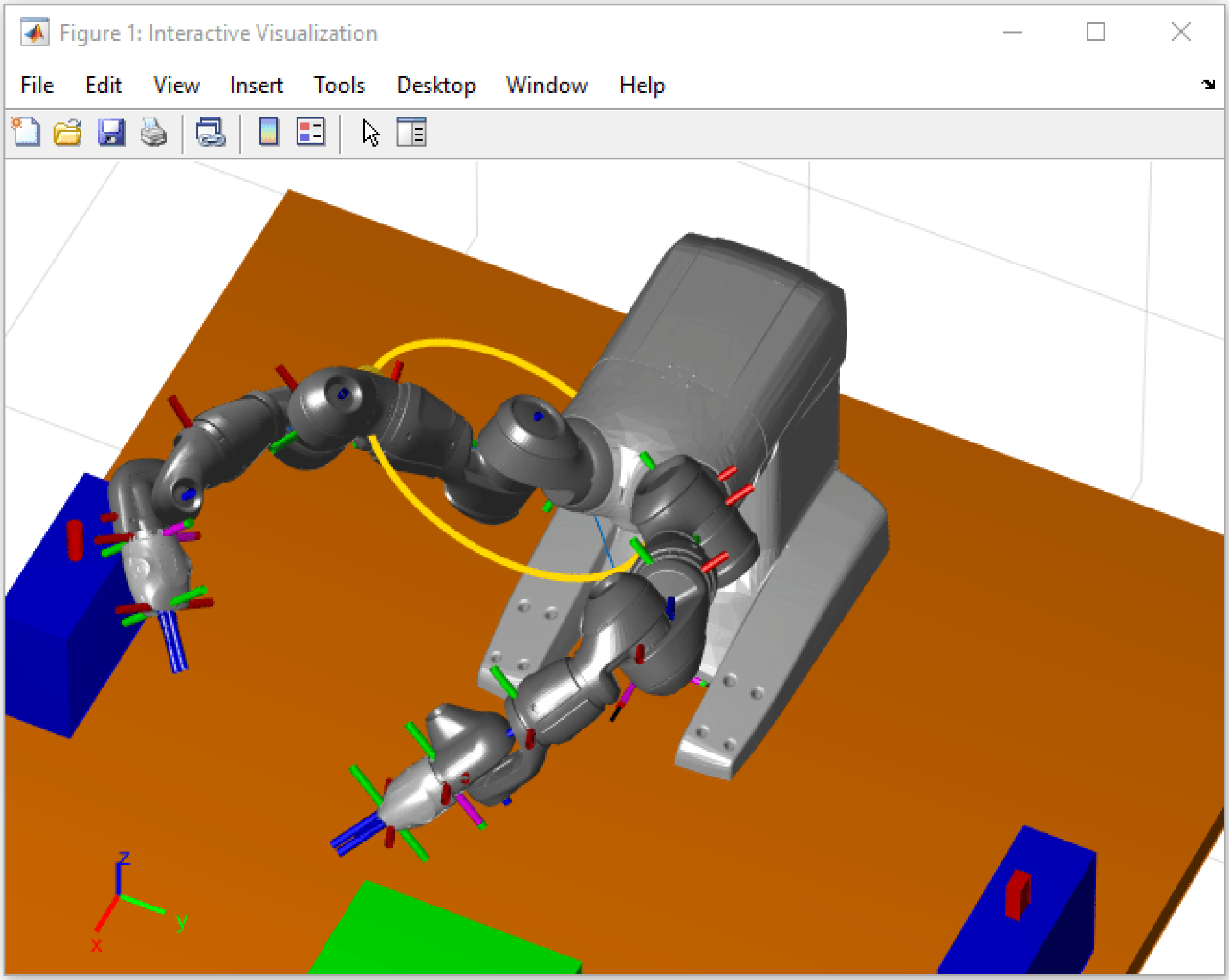


Figure 1: Interactive Visualization
File Edit View Insert Tools Desktop Window Help
Z
y
X

Iteractively move the robot around until you have a desired configuration. Save configurations using `addConfiguration`. Each call adds the current configuration to the `StoredConfigurations` property.

```
addConfiguration(iviz)
```

Define Waypoints for a Trajectory

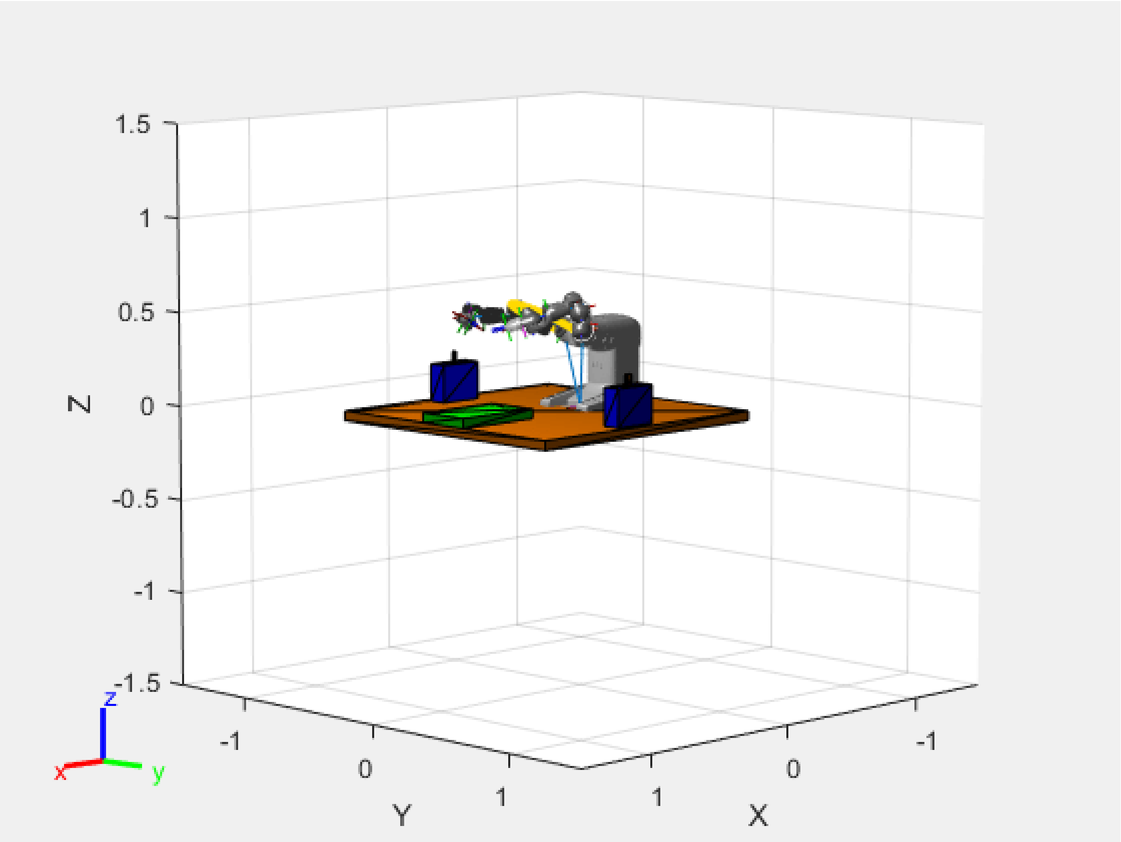
For the purpose of this example, a set of configurations are provided in a `.mat` file.

Load the configurations, and specify them as the set of stored configurations. The first configuration is added by updating the `Configuration` property and calling `addConfiguration`, which you could do interactively, but the rest are simply added by assigning the `StoredConfigurations` property directly.

```
load abbYumiSaveTrajectoryWaypts.mat
```

```
removeConfigurations(iviz) % Clear stored configurations
```

```
% Start at a valid starting configuration
iviz.Configuration = startingConfig;
```



```
addConfiguration(iviz)

% Specify the entire set of waypoints
iviz.StoredConfigurations = [startingConfig, ...
                             graspApproachConfig, ...
                             graspPoseConfig, ...
                             graspDepartConfig, ...
                             placeApproachConfig, ...
                             placeConfig, ...
                             placeDepartConfig, ...
                             startingConfig];
```

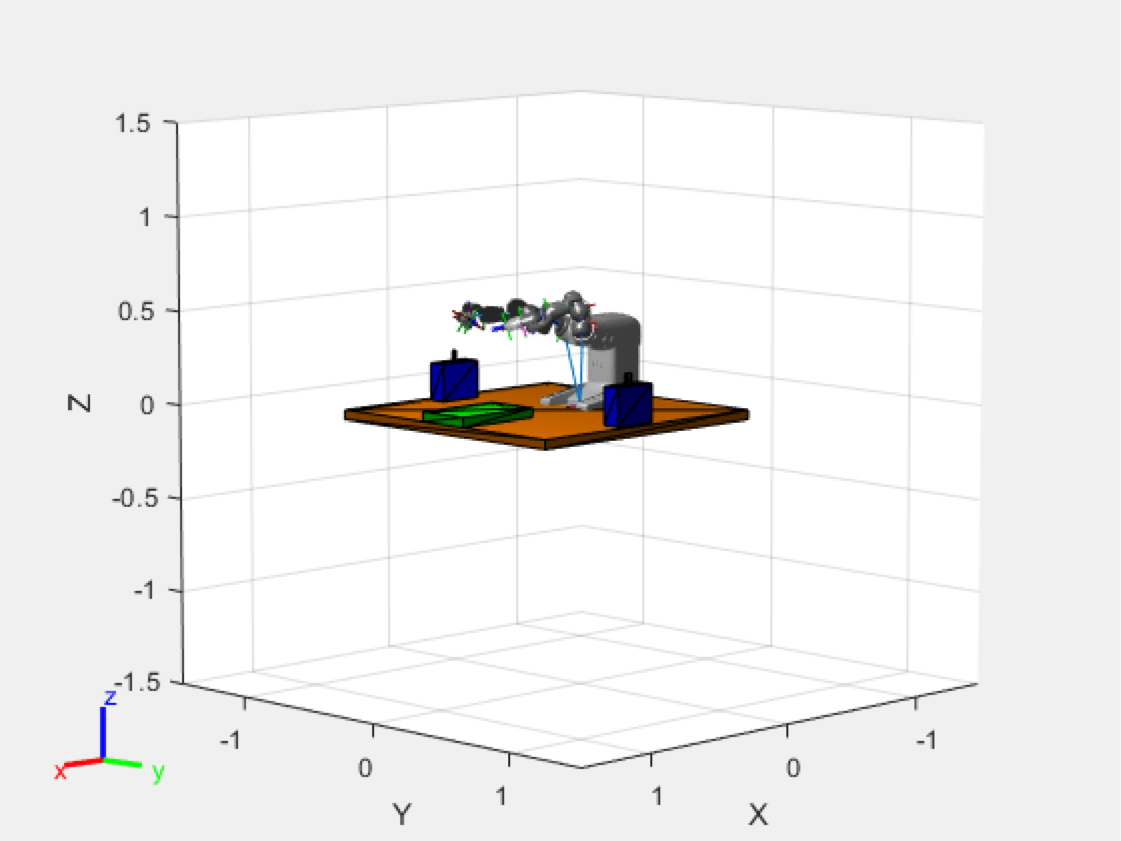
Generate the Trajectory and Play It Back

Once all the waypoints have been stored, construct a trajectory that the robot follows. For this example, a trapezoidal velocity profile is generated using `trapveltraj`. A trapezoidal velocity profile means the robot stops smoothly at each waypoint, but achieves a set max speed while in motion.

```
numSamples = 100*size(iviz.StoredConfigurations, 2) + 1;
[q,~,~,tvec] = trapveltraj(iviz.StoredConfigurations,numSamples,'EndTime',2);
```

Replay the generated trajectory by iterating through the generated `q` matrix, which represents a series of joint configurations that move between each waypoint. In this case, a rate control object is used to ensure that the play back speed is reflective of the actual execution speed.

```
iviz.ShowMarker = false;
showFigure(iviz)
rateCtrlObj = rateControl(numSamples/(max(tvec) + tvec(2)));
for i = 1:numSamples
    iviz.Configuration = q(:,i);
    waitfor(rateCtrlObj);
end
```



The figure shows the robot executes a smooth trajectory between all the defined waypoints.

Position Delta Robot Using Generalized Inverse Kinematics

Model a delta robot using the a `rigidBodyTree` robot model. Specify kinematic constraints for generalized inverse kinematics (GIK) to ensure the proper behavior of the robot. Solve for joint configurations that obey the defined model and constraints.

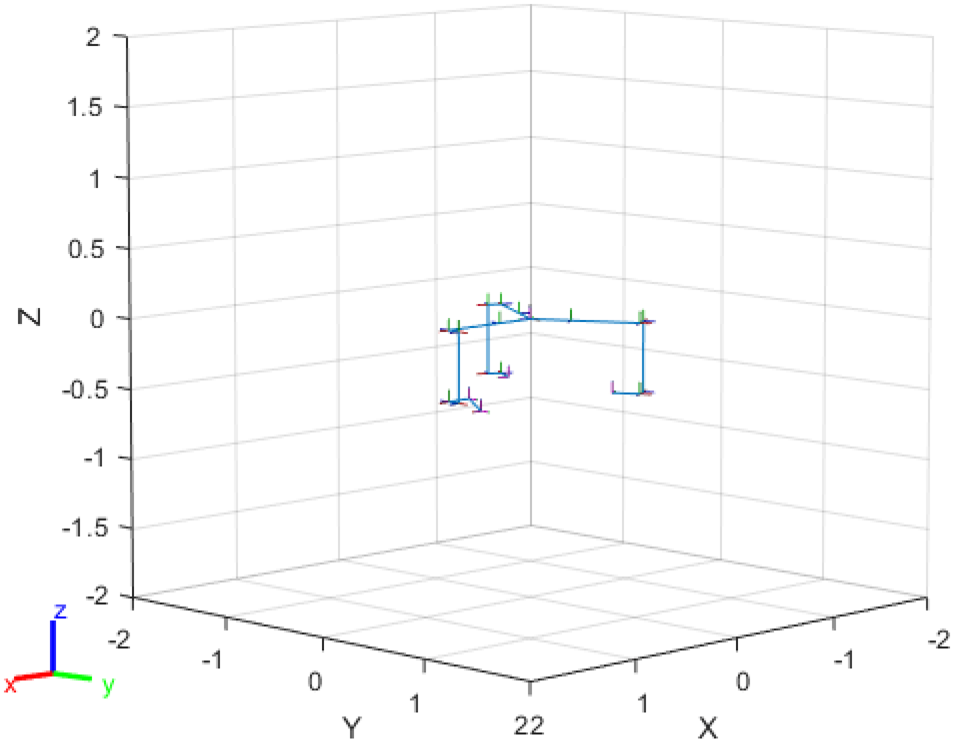
Create Delta Robot

Normally, delta robots contain closed-loop kinematic chains. The `rigidBodyTree` object does not support closed-loop chains. To avoid this, the robot is modeled as a tree, with the arms of the delta robot remaining unconnected. Call the helper function which builds the robot model and outputs the `rigidBodyTree object`.

In a subsequent step, the generalized inverse kinematics solver will apply constraints that force the separate arms of the tree to move together, thereby ensuring that the robot behaves in a kinematically accurate manner.

The robot is fairly complicated, so a helper function is used to create the rigidBodyTree object.

```
robot = exampleHelperDeltaRobot;
show(robot);
```



As shown, the robot consists of three arms, but they still need to be connected to match the classic delta robot configuration.

Create Inverse Kinematic Constraints

Create a `generalizedInverseKinematics` object, and specify the robot model. Limit the maximum number of interations based on performance.

```
gik1 = generalizedInverseKinematics('RigidBodyTree', robot);
gik1.SolverParameters.MaxIterations = 20;
```

Create an `interactiveRigidBodyTree` object to visualize the robot model and provide interactive markers for moving bodies. This interactivity helps verify your kinematic constraints. Specify the `gik1` solver using name-value pairs. Specify a pose weight vector that only focuses on the *xyz*-position and not the orientation.

```
viztree = interactiveRigidBodyTree(robot, 'IKSolver', gik1, 'SolverPoseWeights', [0 1]);
```

Using this interactive object, the end effector can be dragged around to show how the robot moves. Currently, the behavior is not as desired for a normal delta robot.

Store the current axes.

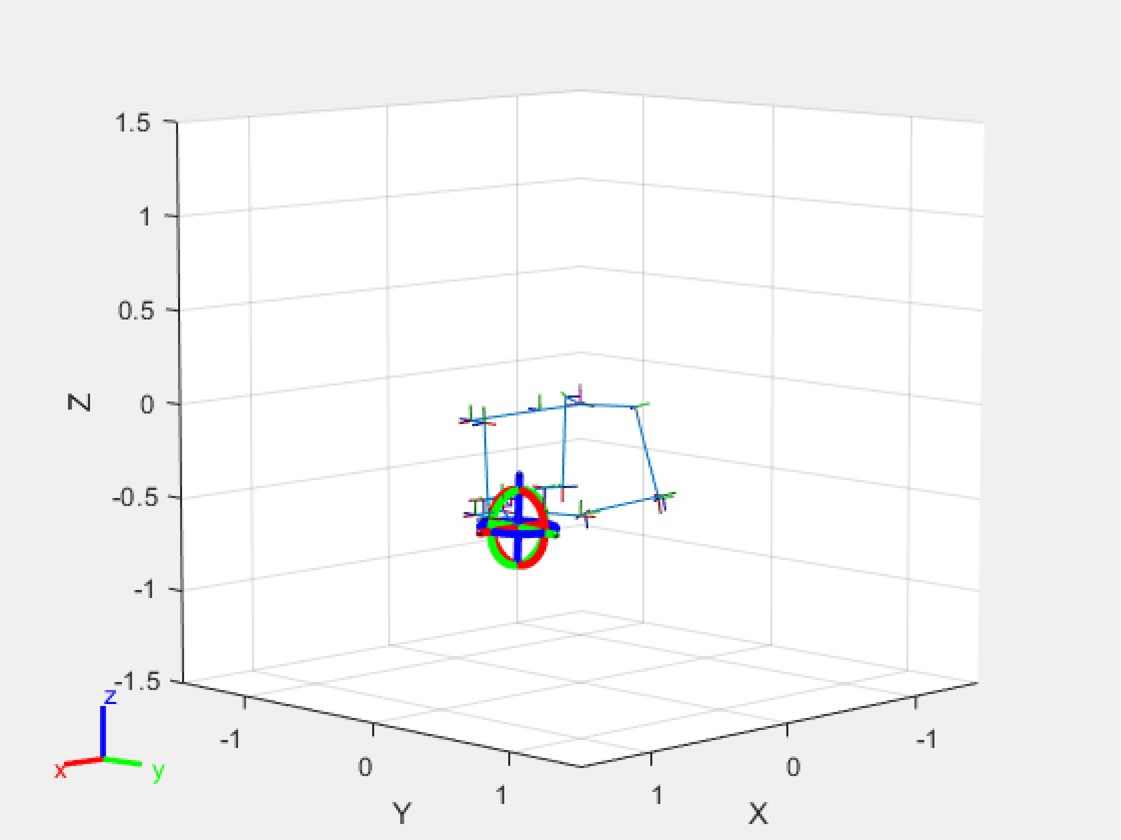
```
ax = gca;
```

Add constraints to the GIK solver to ensure that the arms are connected. Attach the two arms with no end effector to the 6th body of the primary arm which includes the end effector.

```
% Ensure that the body 6 of arm 2 maintains a pose relative to body 6 of arm 1
poseTgt1 = constraintPoseTarget('arm2_body6');
poseTgt1.ReferenceBody = 'arm1_body6';
poseTgt1.TargetTransform = trvec2tform([-sqrt(3)*0.5*0.2, 0.5*0.2, 0]) * eul2tform([2*pi/3, 0, 0

% Ensure that the body 6 of arm 3 maintains a pose relative to body 6 of arm 1
poseTgt2 = constraintPoseTarget('arm3_body6');
poseTgt2.ReferenceBody = 'arm1_body6';
poseTgt2.TargetTransform = trvec2tform([-sqrt(3)*0.5*0.2, -0.5*0.2, 0]) * eul2tform([-2*pi/3, 0,
```

To apply these constraints to the robot, call `addConstraint` to the `vizTree` object.

```
addConstraint(viztree,poseTgt1);
addConstraint(viztree,poseTgt2);
```



Now when the end effector is moved around, the constraints are respected and the arms stay connected.

Solve Generalized Inverse Kinematics Programmatically

The interactive visualization is useful for validating the solver constraints, but for direct programmatic use, create a separate GIK solver that can be called. This solver can be copied from the `IKSolver` property of the `interactiveRigidBodyTree` object, or created independently.

```
gik2 = generalizedInverseKinematics('RigidBodyTree', robot);
gik2.SolverParameters.MaxIterations = 20;
```

For the GIK solver, an additional constraint is required to define the end effector position, which is normally controlled by the interactive marker. Update the `TargetTransform` to solve for different desired end-effector positions.

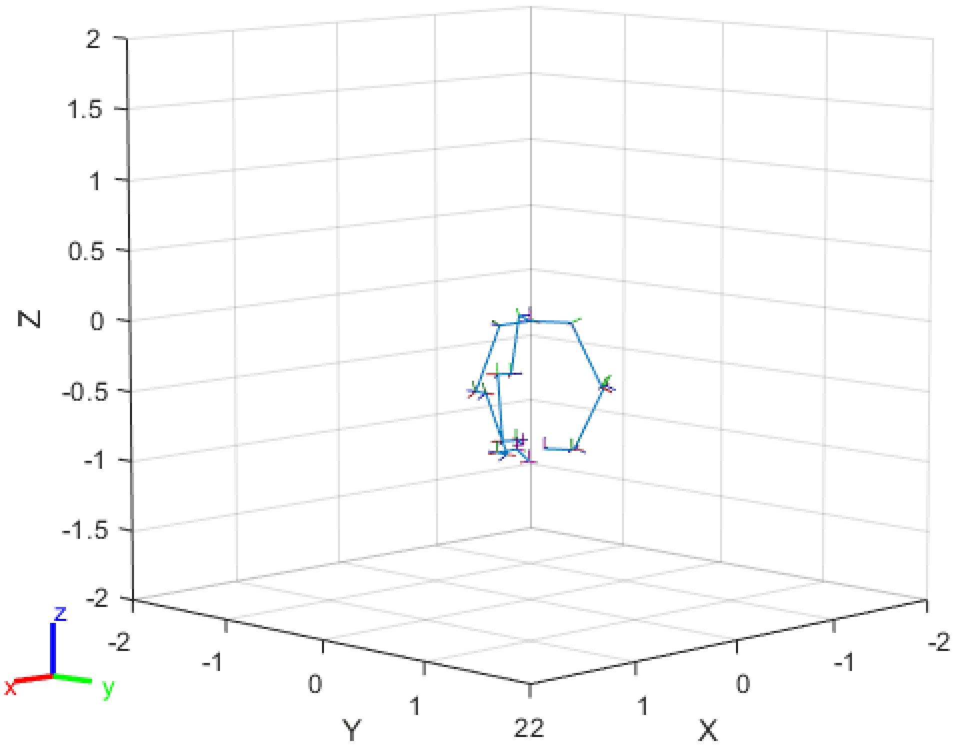
```
poseTgt3 = constraintPoseTarget('endEffector');
poseTgt3.ReferenceBody = 'base';
poseTgt3.TargetTransform = trvec2tform([0, 0, -1]);
```

Specify all the constraint types used by the solver.

```
gik2.ConstraintInputs = {'pose','pose', 'pose'};
```

Call the `gik2` solver with the specified pose target constraint objects. Give an initial guess of the home configuration of the robot. Show the solution.

```
% Provide an initial guess for the solver
q0 = homeConfiguration(robot);

% Solve for a the target pose given to poseTgt3
[q, solutionInfo] = gik2(q0, poseTgt1, poseTgt2, poseTgt3);

% Visualize the results
figure;
show(robot, q);
```



Trajectory Control Modeling with Inverse Kinematics

This Simulink example demonstrates how the Inverse Kinematics block can drive a manipulator along a specified trajectory. The desired trajectory is specified as a series of tightly-spaced poses for the end effector of the manipulator. Trajectory generation and waypoint definition represents many robotics applications like pick and place operation, calculating trajectories from spatial acceleration and velocity profiles, or even mimicking external observations of key frames using cameras and computer vision. Once a trajectory is generated, the Inverse Kinematics block is used to translate this to a joint-space trajectory, which can then be used to simulate the dynamics of the manipulator and controller.

Model Overview

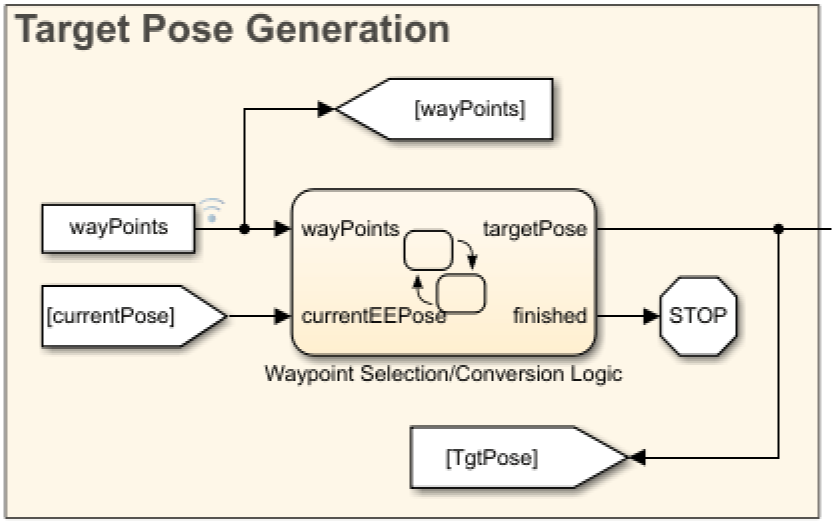
Load the model to see how it is constructed.

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

The model is composed of four primary operations:

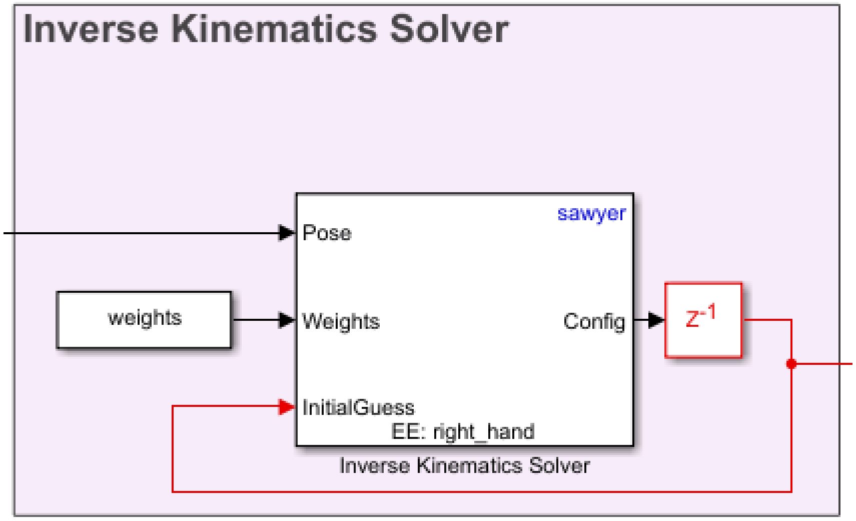
- **Target Pose Generation**
- **Inverse Kinematics**
- **Manipulator Dynamics**
- **Pose Measurement**

Target Pose Generation



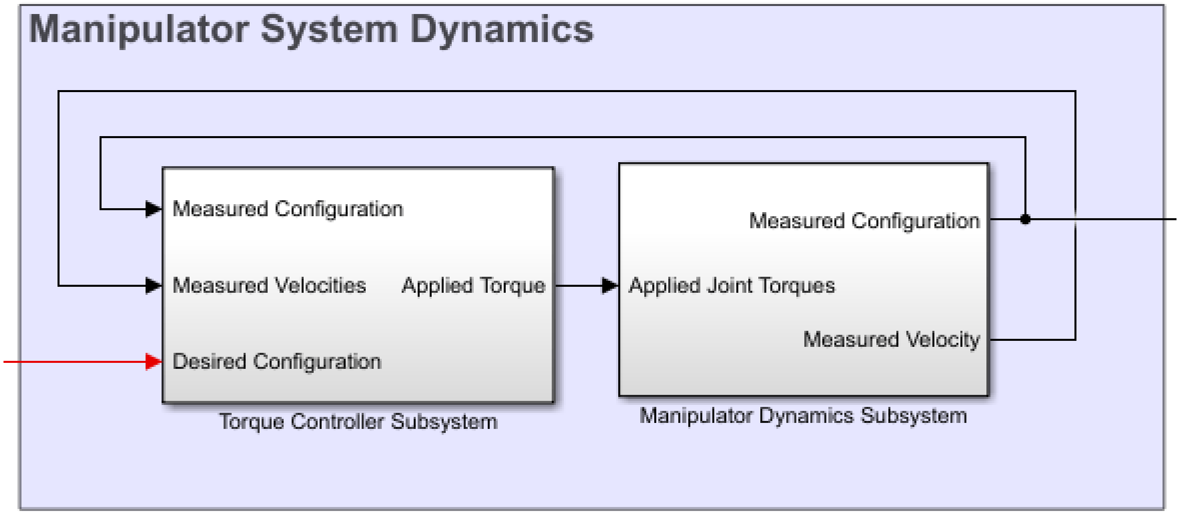
This Stateflow chart selects which waypoint is the current objective for the manipulator. The chart adjusts the target to the next waypoint once the manipulator gets to within a tolerance of the current objective. The chart also converts and assembles the components of the waypoint into a homogenous transformation through the `eul2tform` function. Once there are no more waypoints to select, the chart terminates the simulation.

Inverse Kinematics



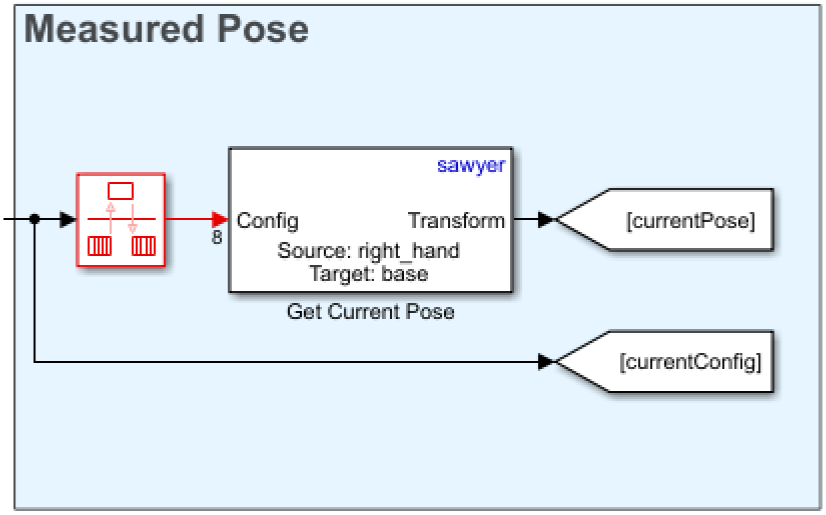
Inverse kinematics calculated a set of joint angles to produce a desired pose for an end effector. Use the Inverse Kinematicswith a `rigidBodyTree` model and specify the target pose of the end effect as a homogenous transformation. Specify a series of weights for the relative tolerance constraints on the position and orientation of the solution, and give an initial estimate of the joint positions. The block outputs a vector of joint positions that produce the desired pose from the `rigidBodyTree` model specified in the block parameters. To ensure smooth continuity of the solutions, the previous configuration solution is used as the starting position for the solver. This also reduces the redundancy of calculations if the target pose has not updated since the last simulation time step.

Manipulator Dynamics



The manipulator dynamics consists of two components, a controller to generate torque signals and a dynamics model to model the dynamics of the manipulator given these torque signals. The controller in the example uses a feed-forward component calculated through the inverse dynamics of the manipulator and a feedback PD controller to correct for error. The model of the manipulator uses the Forward Dynamics block that works with a `rigidBodyTree` object. For more sophisticated dynamics and visualization techniques, consider utilizing tools from the Control Systems Toolbox™ blockset and Simscape Multibody™ to replace the Forward Dynamics block.

Pose Measurement

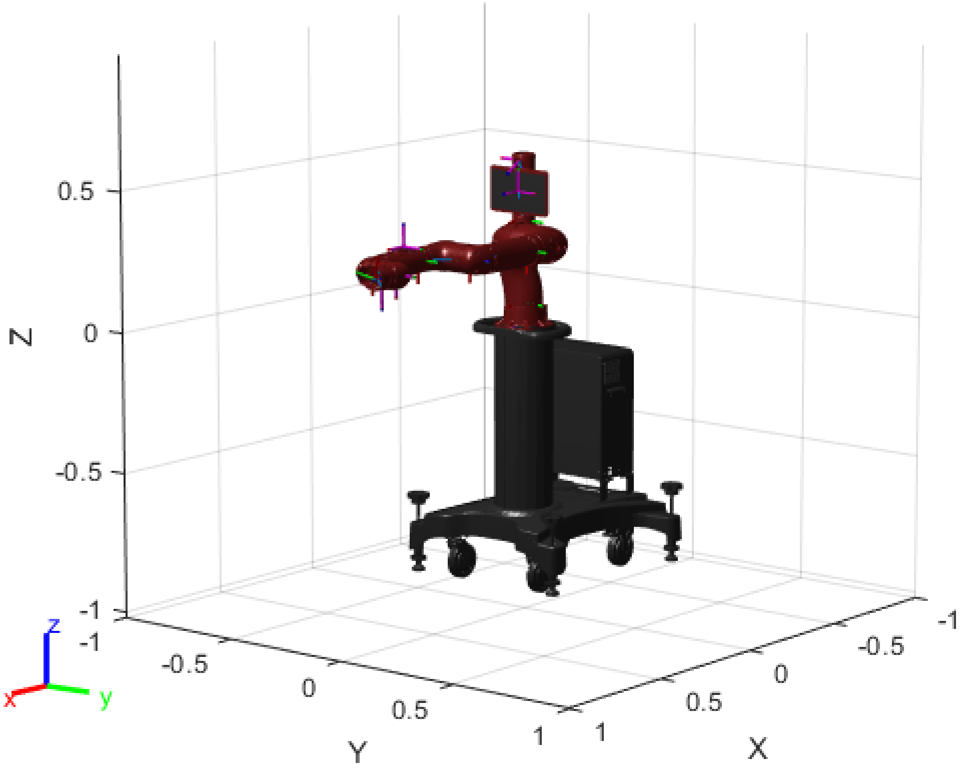


The pose measurement takes the joint angle readings from the manipulator model and converts them into a homogenous transform matrix to be used as feedback in the **Waypoint Selection** section.

Manipulator Definition

The manipulator used for this example is the Rethink Sawyer™ robot manipulator. The `rigidBodyTree` object that describes the manipulator is imported from a URDF (unified robot description format) file using `importrobot`.

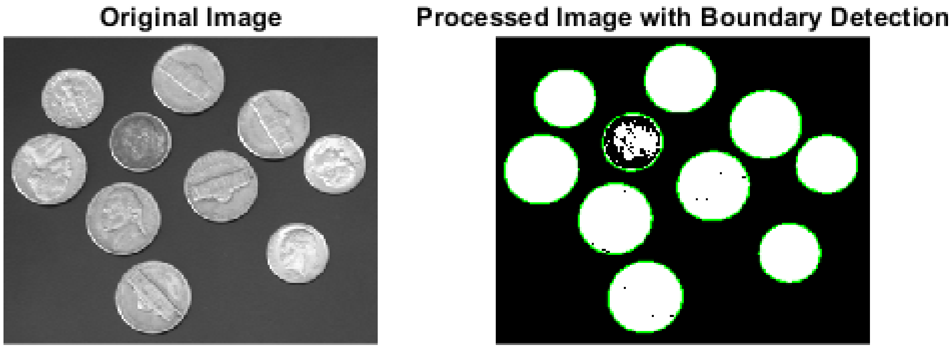
```
% Import the manipulator as a rigidBodyTree Object
sawyer = importrobot('sawyer.urdf');
sawyer.DataFormat = 'column';

% Define end-effector body name
eeName = 'right_hand';

% Define the number of joints in the manipulator
numJoints = 8;

% Visualize the manipulator
show(sawyer);
xlim([-1.00 1.00])
ylim([-1.00 1.00]);
zlim([-1.02 0.98]);
view([128.88 10.45]);
```



Waypoint Generation

In this example, the goal of the manipulator is to be able to trace out the boundaries of the coins detected in the image, `coins.png`. First, the image is processed to find the boundaries of the coins.

```
I = imread('coins.png');
bwBoundaries = imread('coinBoundaries.png');

figure
subplot(1,2,1)
imshow(I,'Border','tight')
title('Original Image')

subplot(1,2,2)
imshow(bwBoundaries,'Border','tight')
title('Processed Image with Boundary Detection')
```



After the image processing, the edges of the coins are extracted as pixel locations. The data is loaded in from the MAT-file, `boundaryData`. `boundaries` is a cell array where each cell contains an array describing the pixel coordinates for a single detected boundary. A more comprehensive view of how to generate this data can be found in the example, "Boundary Tracing in Images" (requires Image Processing Toolbox).

```
load boundaryData.mat boundaries
whos boundaries
```

```
  Name                Size             Bytes  Class     Attributes

  boundaries      10x1             25376  cell
```

To map this data to the world frame, we need to define where the image is located and the scaling between pixel coordinates and spatial coordinates.

```
% Image origin coordinates
imageOrigin = [0.4,0.2,0.08];

% Scale factor to convert from pixels to physical distance
scale = 0.0015;
```

The Euler angles for the desired end effector orientation at each point must also be defined.

```
eeOrientation = [0, pi, 0];
```

In this example the orientation is chosen such that the end effector is always perpendicular to the plane of the image.

Once this information is defined each set of desired coordinates and Euler angles can be compiled into a waypoint. Each waypoint is represented as a six-element vector whose first three elements correspond to the desired *xyz*-positions of the manipulator in the world frame. The last three elements correspond to the ZYX Euler angles of the desired orientation.

$$\text{Waypoint} = \left[X\ Y\ Z\ \phi_z\ \phi_y\ \phi_x\right]$$

The waypoints are concatenated to form an n-by-6 array, where n is the total number of poses in the trajectory. Each row in the array corresponds to a waypoint in the trajectory.

```
% Clear previous waypoints and begin building wayPoint array
clear wayPoints

% Start just above image origin
waypt0 = [imageOrigin + [0 0 .2],eeOrientation];

% Touch the origin of the image
waypt1 = [imageOrigin,eeOrientation];

% Interpolate each element for smooth motion to the origin of the image
for i = 1:6

    interp = linspace(waypt0(i),waypt1(i),100);
    wayPoints(:,i) = interp';

end
```

In total, there are 10 coins. For simiplicity and speed, a smaller subset of coins can be traced by limiting the total number passed to the waypoints. Below, numTraces = 3 coins are traced in the image.

```
% Define the number of coins to trace
numTraces = 3;

% Assemble the waypoints for boundary tracing
for i = 1:min(numTraces, size(boundaries,1))

    %Select a boundary and map to physical size
    segment = boundaries{i}*scale;

    % Pad data for approach waypoint and lift waypoint between boundaries
    segment = [segment(1,:); segment(:,:); segment(end,:)];

    % Z-offset for moving between boundaries
    segment(1,3) = .02;
    segment(end,3) = .02;

    % Translate to origin of image
    cartesianCoord = imageOrigin + segment;

    % Repeat desired orientation to match the number of waypoints being added
    eulerAngles = repmat(eeOrientation,size(segment,1),1);

    % Append data to end of previous wayPoints
    wayPoints = [wayPoints;
                 cartesianCoord, eulerAngles];
end
```

This array is the primary input to the model.

Model Setup

Several parameters must be initialized before the model can be run.

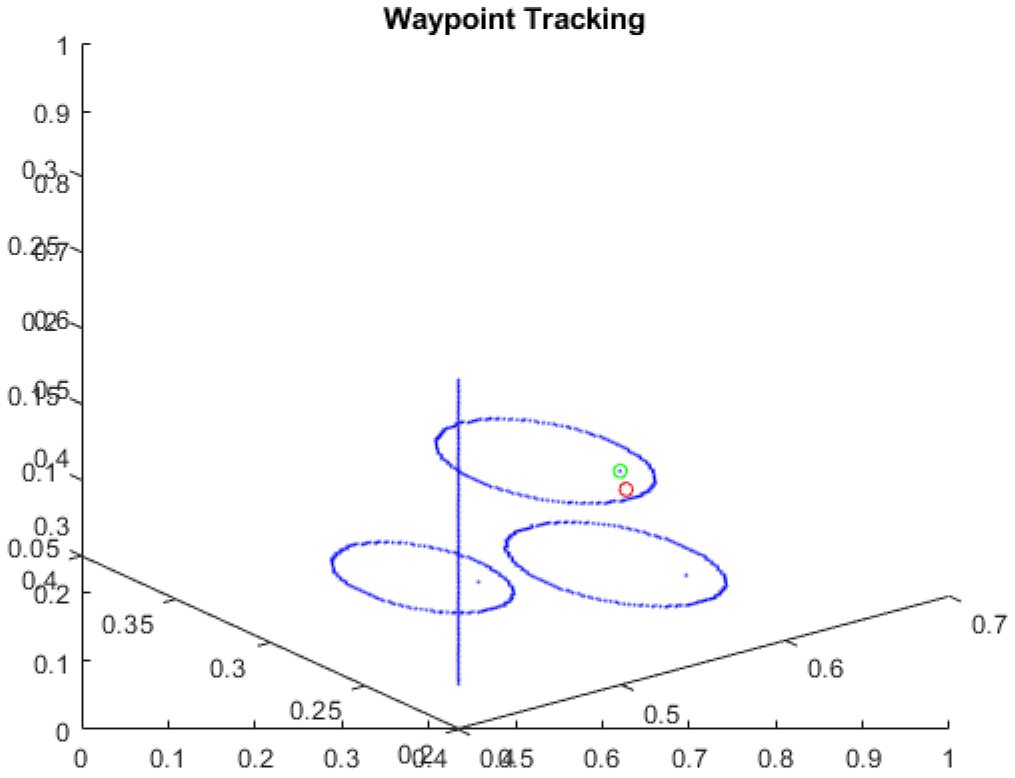
```
% Initialize size of q0, the robot joint configuration at t=0. This will
% later be replaced by the first waypoint.
q0 = zeros(numJoints,1);

% Define a sampling rate for the simulation.
Ts = .01;

% Define a [1x6] vector of relative weights on the orientation and
% position error for the inverse kinematics solver.
weights = ones(1,6);

% Transform the first waypoint to a Homogenous Transform Matrix for initialization
initTargetPose = eul2tform(wayPoints(1,4:6));
initTargetPose(1:3,end) = wayPoints(1,1:3)';

% Solve for q0 such that the manipulator begins at the first waypoint
ik = inverseKinematics('RigidBodyTree',sawyer);
[q0,solInfo] = ik(eeName,initTargetPose,weights,q0);
```

Simulate the Manipulator Motion

To simulate the model, use the `sim` command. The model generates the output dataset, `jointData` and shows the progress in two plots:

- The **X Y Plot** shows a top-down view of the tracing motions of the manipulator. The lines between the circles occur as the manipulator transitions from one coin outline to the next.
- The **Waypoint Tracking** plot visualizes the progress in 3D. The green dot indicates the target position. The red dot indicates the actual end-effector position achieved by the end effector using feedback control.

```
% Close currently open figures
close all

% Open & simulate the model
open_system('IKTrajectoryControlExample.slx');
sim('IKTrajectoryControlExample.slx');
```

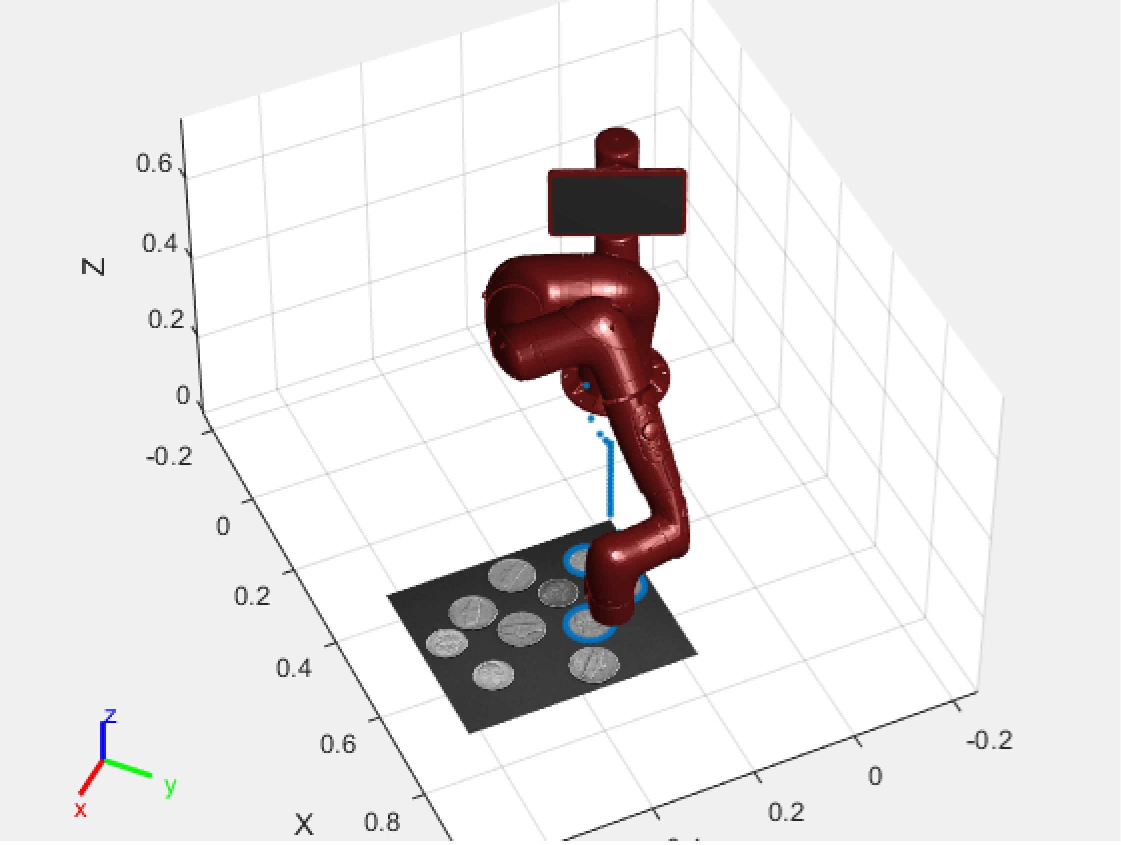


Visualize the Results

The model outputs two datasets that can be used for visualization after simulation. The joint configurations are provided as `jointData`. The robot end-effector poses are output as `poseData`.

```
% Remove unnecessary meshes for faster visualization
clearMeshes(sawyer);

% Data for mapping image
[m,n] = size(I);

[X,Y] = meshgrid(0:m,0:n);
X = imageOrigin(1) + X*scale;
Y = imageOrigin(2) + Y*scale;

Z = zeros(size(X));
Z = Z + imageOrigin(3);

% Close all open figures
close all

% Initialize a new figure window
figure;
set(gcf,'Visible','on');

% Plot the initial robot position
show(sawyer, jointData(1,:)');
```

```
hold on

% Initialize end effector plot position
p = plot3(0,0,0,'.');
warp(X,Y,Z,I');

% Change view angle and axis
view(65,45)
axis([-.25 1 -.25 .75 0 0.75])

% Iterate through the outputs at 10-sample intervals to visualize the results
for j = 1:10:length(jointData)
    % Display manipulator model
    show(sawyer,jointData(j,:)', 'Frames', 'off', 'PreservePlot', false);

    % Get end effector position from homoegenous transform output
    pos = poseData(1:3,4,j);

    % Update end effector position for plot
    p.XData = [p.XData pos(1)];
    p.YData = [p.YData pos(2)];
    p.ZData = [p.ZData pos(3)];

    % Update figure
    drawnow
end
```



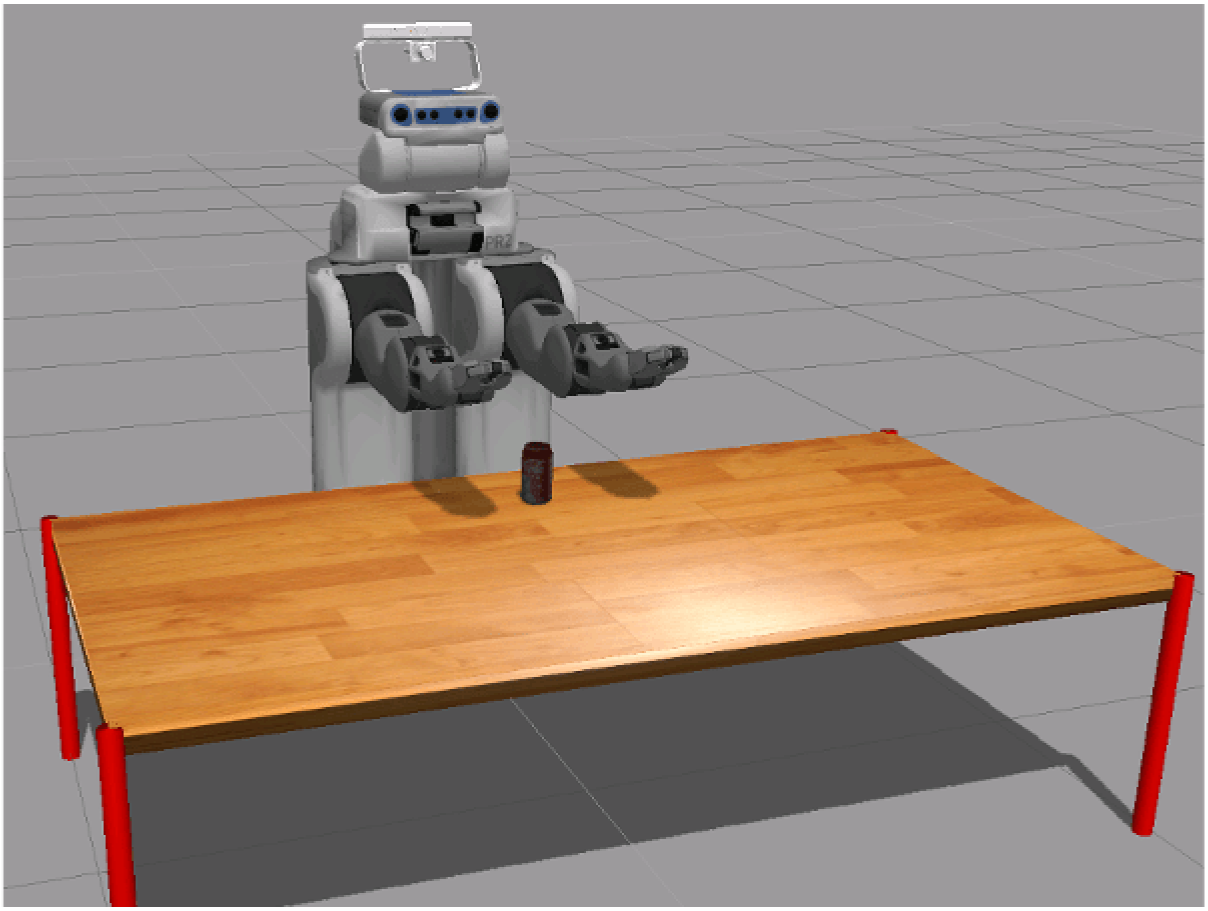
Control PR2 Arm Movements Using ROS Actions and Inverse Kinematics

This example shows how to send commands to robotic manipulators in MATLAB®. Joint position commands are sent via a ROS action client over a ROS network. This example also shows how to calculate joint positions for a desired end-effector position. A rigid body tree defines the robot geometry and joint constraints, which is used with inverse kinematics to get the robot joint positions. You can then send these joint positions as a trajectory to command the robot to move.

Bring up PR2 Gazebo Simulation

This example uses an Ubuntu® virtual machine (VM) containing ROS Melodic available for download here. This example does not support ROS Noetic as it relies on ROS packages which are only supported until ROS Melodic.

Spawn PR2 in a simple environment (only with a table and a coke can) in the Gazebo Simulator by launching the `Gazebo PR2 Simulator` desktop shortcut from the VM desktop. See “Get Started with Gazebo and Simulated TurtleBot” (ROS Toolbox) for more details on this process.



Connect to ROS Network from MATLAB®

In your MATLAB instance on the host computer, run the following commands to initialize ROS global node in MATLAB and connect to the ROS master in the virtual machine through its IP address `ipaddress`. Replace '192.168.233.133' with the IP address of your virtual machine. Specify the port if needed.

```
ipaddress = '192.168.116.161';
rosinit(ipaddress,11311);
```

```
Initializing global node /matlab_global_node_03004 with NodeURI http://192.168.116.1:64988/
```

Create Action Clients for Robot Arms and Send Commands

In this task, you send joint trajectories to the PR2 arms. The arms can be controlled via ROS actions. Joint trajectories are manually specified for each arm and sent as separate goal messages via separate action clients.

Create a ROS simple action client to send goal messages to move the right arm of the robot. `rosactionclient` (ROS Toolbox) object and goal message are returned. Wait for the client to connect to the ROS action server.

```
[rArm, rGoalMsg] = rosactionclient('r_arm_controller/joint_trajectory_action');
waitForServer(rArm);
```

The goal message is a `trajectory_msgs/JointTrajectoryPoint` message. Each trajectory point should specify positions and velocities of the joints.

```
disp(rGoalMsg)
```

```
  ROS JointTrajectoryGoal message with properties:

    MessageType: 'pr2_controllers_msgs/JointTrajectoryGoal'
     Trajectory: [1×1 JointTrajectory]

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

```
disp(rGoalMsg.Trajectory)
```

```
  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
```

Set the joint names to match the PR2 robot joint names. Note that there are 7 joints to control. To find what joints are in PR2 right arm, type this command in the virtual machine terminal:

```
rosparam get /r_arm_controller/joints
```

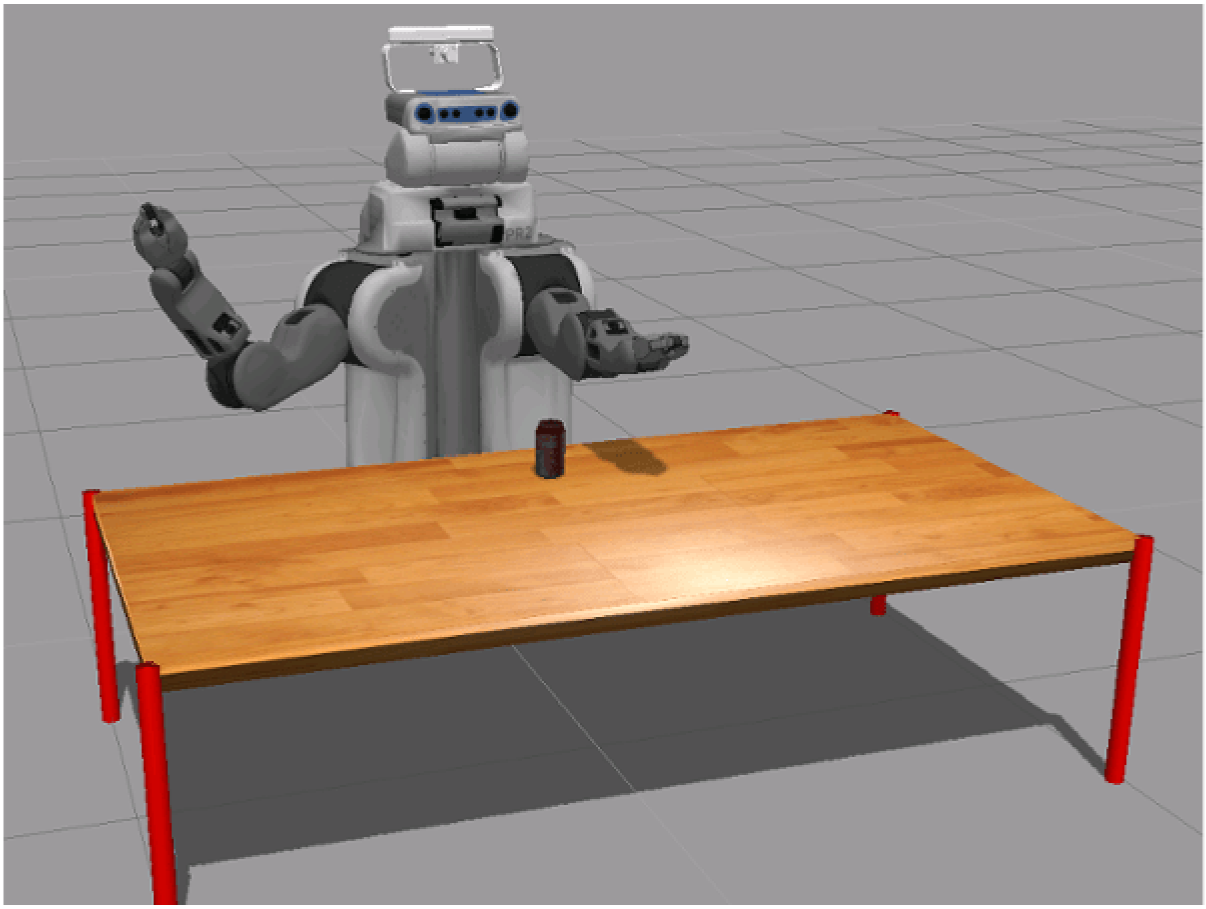
```
rGoalMsg.Trajectory.JointNames = {'r_shoulder_pan_joint', ...
                                   'r_shoulder_lift_joint', ...
                                   'r_upper_arm_roll_joint', ...
                                   'r_elbow_flex_joint',...
                                   'r_forearm_roll_joint',...
                                   'r_wrist_flex_joint',...
                                   'r_wrist_roll_joint'};
```

Create setpoints in the arm joint trajectory by creating ROS `trajectory_msgs/JointTrajectoryPoint` messages and specifying the position and velocity of all 7 joints as a vector. Specify a time from the start as a ROS duration object.

```
% Point 1
tjPoint1 = rosmessage('trajectory_msgs/JointTrajectoryPoint');
tjPoint1.Positions = zeros(1,7);
tjPoint1.Velocities = zeros(1,7);
tjPoint1.TimeFromStart = rosduration(1.0);

% Point 2
tjPoint2 = rosmessage('trajectory_msgs/JointTrajectoryPoint');
tjPoint2.Positions = [-1.0 0.2 0.1 -1.2 -1.5 -0.3 -0.5];
tjPoint2.Velocities = zeros(1,7);
tjPoint2.TimeFromStart = rosduration(2.0);
```

Create an object array with the points in the trajectory and assign it to the goal message. Send the goal using the action client. The `sendGoalAndWait` (ROS Toolbox) function will block execution until the PR2 arm finishes executing the trajectory

```
rGoalMsg.Trajectory.Points = [tjPoint1,tjPoint2];

sendGoalAndWait(rArm,rGoalMsg);
```

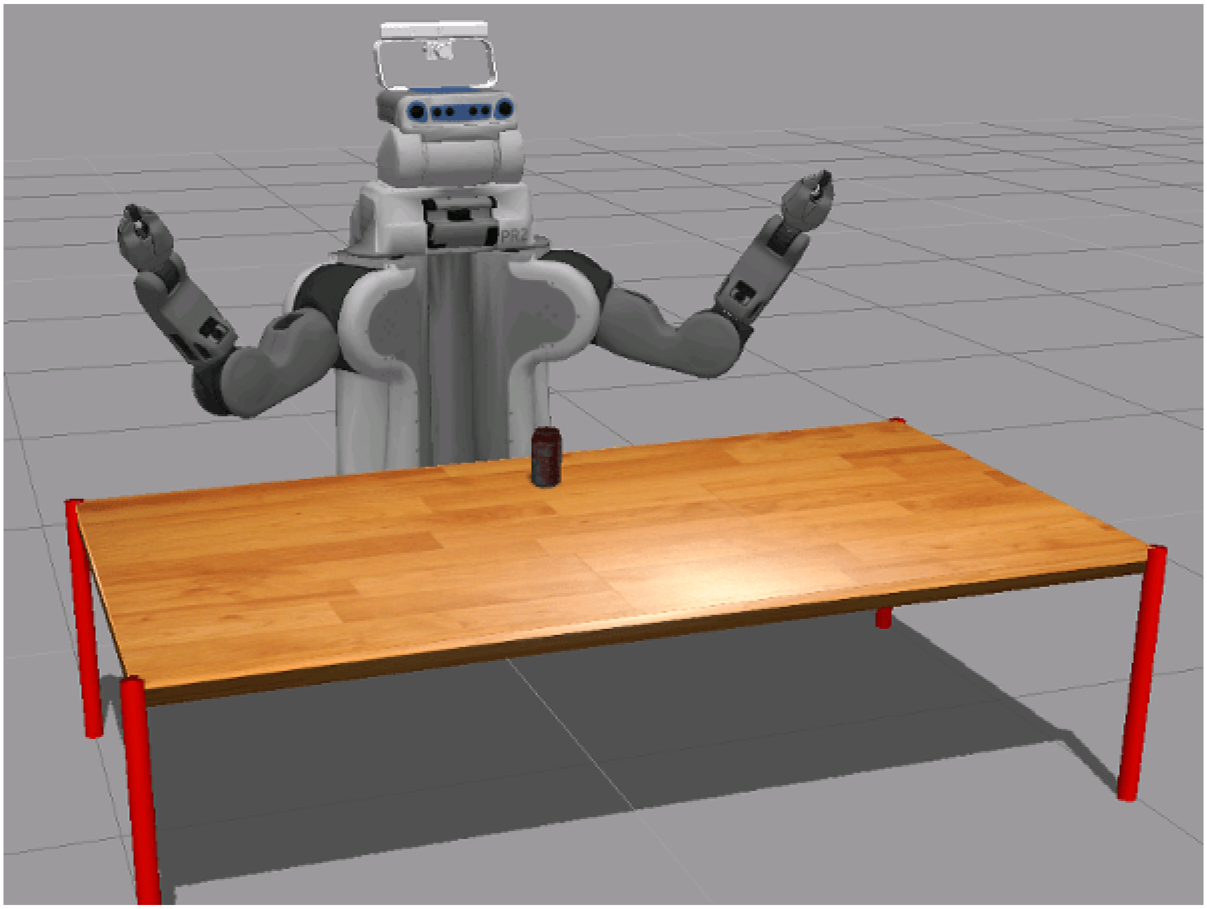


Create another action client to send commands to the left arm. Specify the joint names of the PR2 robot.

```
[lArm, lGoalMsg] = rosactionclient('l_arm_controller/joint_trajectory_action');
waitForServer(lArm);

lGoalMsg.Trajectory.JointNames = {'l_shoulder_pan_joint', ...
                                   'l_shoulder_lift_joint', ...
                                   'l_upper_arm_roll_joint', ...
                                   'l_elbow_flex_joint',...
                                   'l_forearm_roll_joint',...
                                   'l_wrist_flex_joint',...
                                   'l_wrist_roll_joint'};
```

Set a trajectory point for the left arm. Assign it to the goal message and send the goal.

```
tjPoint3 = rosmessage('trajectory_msgs/JointTrajectoryPoint');
tjPoint3.Positions = [1.0 0.2 -0.1 -1.2 1.5 -0.3 0.5];
tjPoint3.Velocities = zeros(1,7);
tjPoint3.TimeFromStart = rosduration(2.0);

lGoalMsg.Trajectory.Points = tjPoint3;

sendGoalAndWait(lArm,lGoalMsg);
```



Calculate Inverse Kinematics for an End-Effector Position

In this section, you calculate a trajectory for joints based on the desired end-effector (PR2 right gripper) positions. The `inverseKinematics` class calculates all the required joint positions, which can be sent as a trajectory goal message via the action client. A `rigidBodyTree` object is used to define the robot parameters, generate configurations, and visualize the robot in MATLAB®.

Perform The following steps:

- Get the current state of the PR2 robot from the ROS network and assign it to a `rigidBodyTree` object to work with the robot in MATLAB®.
- Define an end-effector goal pose.
- Visualize the robot configuration using these joint positions to ensure a proper solution.
- Use inverse kinematics to calculate joint positions from goal end-effector poses.
- Send the trajectory of joint positions to the ROS action server to command the actual PR2 robot.

Create a Rigid Body Tree in MATLAB®

Load a PR2 robot as a `rigidBodyTree` object. This object defines all the kinematic parameters (including joint limits) of the robot.

```
pr2 = exampleHelperWGPR2Kinect;
```

Get the Current Robot State

Create a subscriber to get joint states from the robot.

```
jointSub = rossubscriber('joint_states');
```

Get the current joint state message.

```
jntState = receive(jointSub);
```

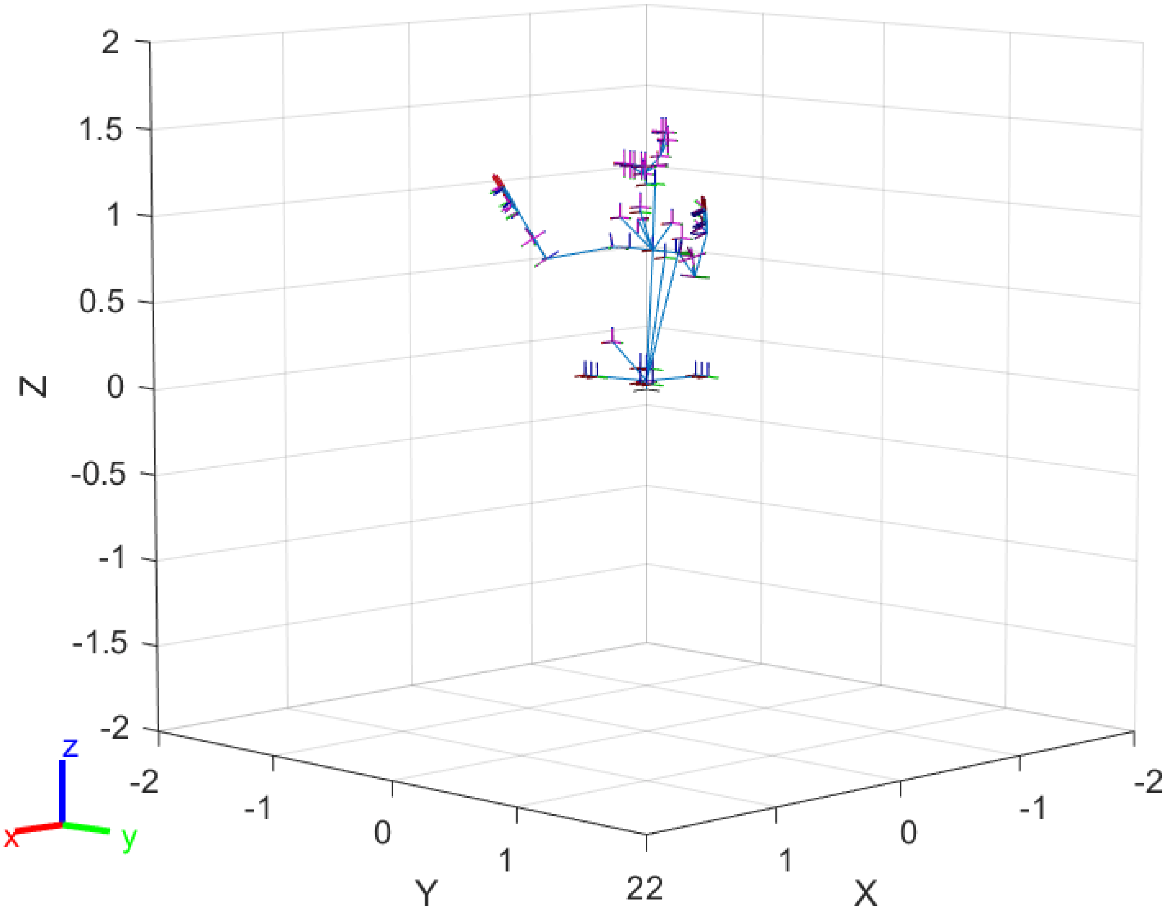
Assign the joint positions from the joint states message to the fields of a configuration struct that the `pr2` object understands.

```
jntPos = exampleHelperJointMsgToStruct(pr2,jntState);
```

Visualize the Current Robot Configuration

Use `show` to visualize the robot with the given joint position vector. This should match the current state of the robot.

```
show(pr2,jntPos)
```



```
ans =
  Axes (Primary) with properties:

             XLim: [-2 2]
             YLim: [-2 2]
           XScale: 'linear'
           YScale: 'linear'
    GridLineStyle: '-'
         Position: [0.1300 0.1100 0.7750 0.8150]
            Units: 'normalized'

  Show all properties
```

Create an `inverseKinematics` object from the `pr2` robot object. The goal of inverse kinematics is to calculate the joint angles for the PR2 arm that places the gripper (i.e. the end-effector) in a desired pose. A sequence of end-effector poses over a period of time is called a trajectory.

In this example, we will only be controlling the robot's arms. Therefore, we place tight limits on the torso-lift joint during planning.

```
torsoJoint = pr2.getBody('torso_lift_link').Joint;
idx = strcmp({jntPos.JointName}, torsoJoint.Name);
torsoJoint.HomePosition = jntPos(idx).JointPosition;
torsoJoint.PositionLimits = jntPos(idx).JointPosition + [-1e-3,1e-3];
```

Create the `inverseKinematics` object.

```
ik = inverseKinematics('RigidBodyTree', pr2);
```

Disable random restart to ensure consistent IK solutions.

```
ik.SolverParameters.AllowRandomRestart = false;
```

Specify weights for the tolerances on each component of the goal pose.

```
weights = [0.25 0.25 0.25 1 1 1];
initialGuess = jntPos; % current jnt pos as initial guess
```

Specify end-effector poses related to the task. In this example, PR2 will reach to the can on the table, grasp the can, move it to a different location and set it down again. We will use the `inverseKinematics` object to plan motions that smoothly transition from one end-effector pose to another.

Specify the name of the end effector.

```
endEffectorName = 'r_gripper_tool_frame';
```

Specify the can's initial (current) pose and the desired final pose.

```
TCanInitial = trvec2tform([0.7, 0.0, 0.55]);
TCanFinal = trvec2tform([0.6, -0.5, 0.55]);
```

Define the desired relative transform between the end-effector and the can when grasping.

```
TGraspToCan = trvec2tform([0,0,0.08])*eul2tform([pi/8,0,-pi]);
```

Define the key waypoints of a desired Cartesian trajectory. The can should move along this trajectory. The trajectory can be broken up as follows:

- Open the gripper
- Move the end-effector from current pose to `T1` so that the robot will feel comfortable to initiate the grasp
- Move the end-effector from `T1` to `TGrasp` (ready to grasp)
- Close the gripper and grab the can
- Move the end-effector from `TGrasp` to `T2` (lift can in the air)
- Move the end-effector from `T2` to `T3` (move can levelly)
- Move the end-effector from `T3` to `TRelease` (lower can to table surface and ready to release)
- Open the gripper and let go of the can
- Move the end-effector from `TRelease` to `T4` (retract arm)

```
TGrasp = TCanInitial*TGraspToCan; % The desired end-effector pose when grasping the can
T1 = TGrasp*trvec2tform([0.,0,-0.1]);
T2 = TGrasp*trvec2tform([0,0,-0.2]);
T3 = TCanFinal*TGraspToCan*trvec2tform([0,0,-0.2]);
```

```
TRelease = TCanFinal*TGraspToCan; % The desired end-effector pose when releasing the can
T4 = TRelease*trvec2tform([-0.1,0.05,0]);
```

The actual motion will be specified by `numWaypoints` joint positions in a sequence and sent to the robot through the ROS simple action client. These configurations will be chosen using the `inverseKinematics` object such that the end effector pose is interpolated from the initial pose to the final pose.

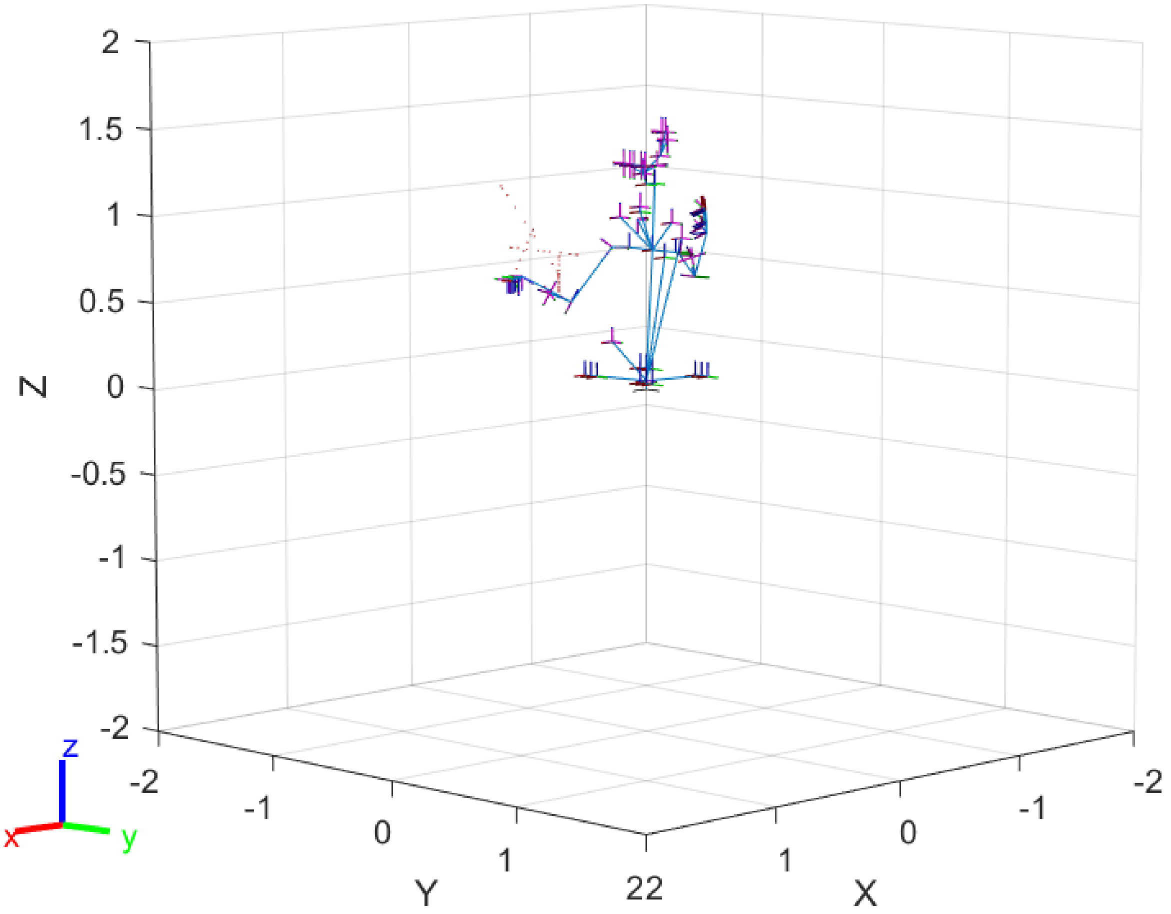
Execute the Motion

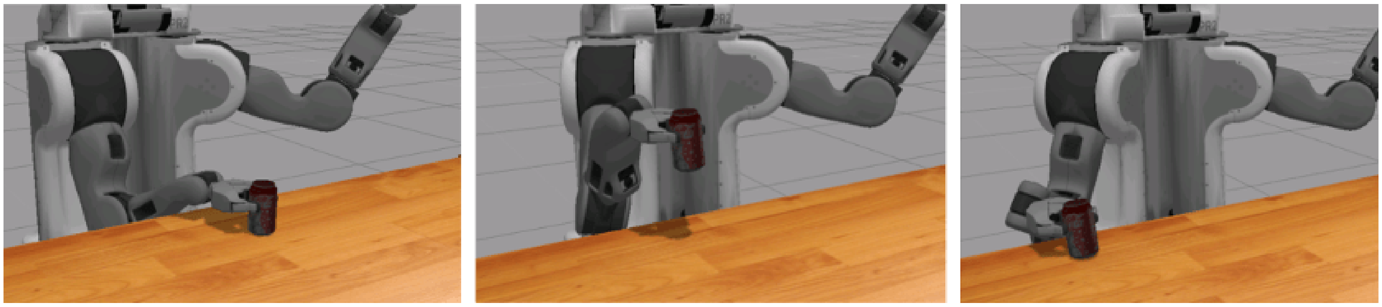
Specify the sequence of motions.

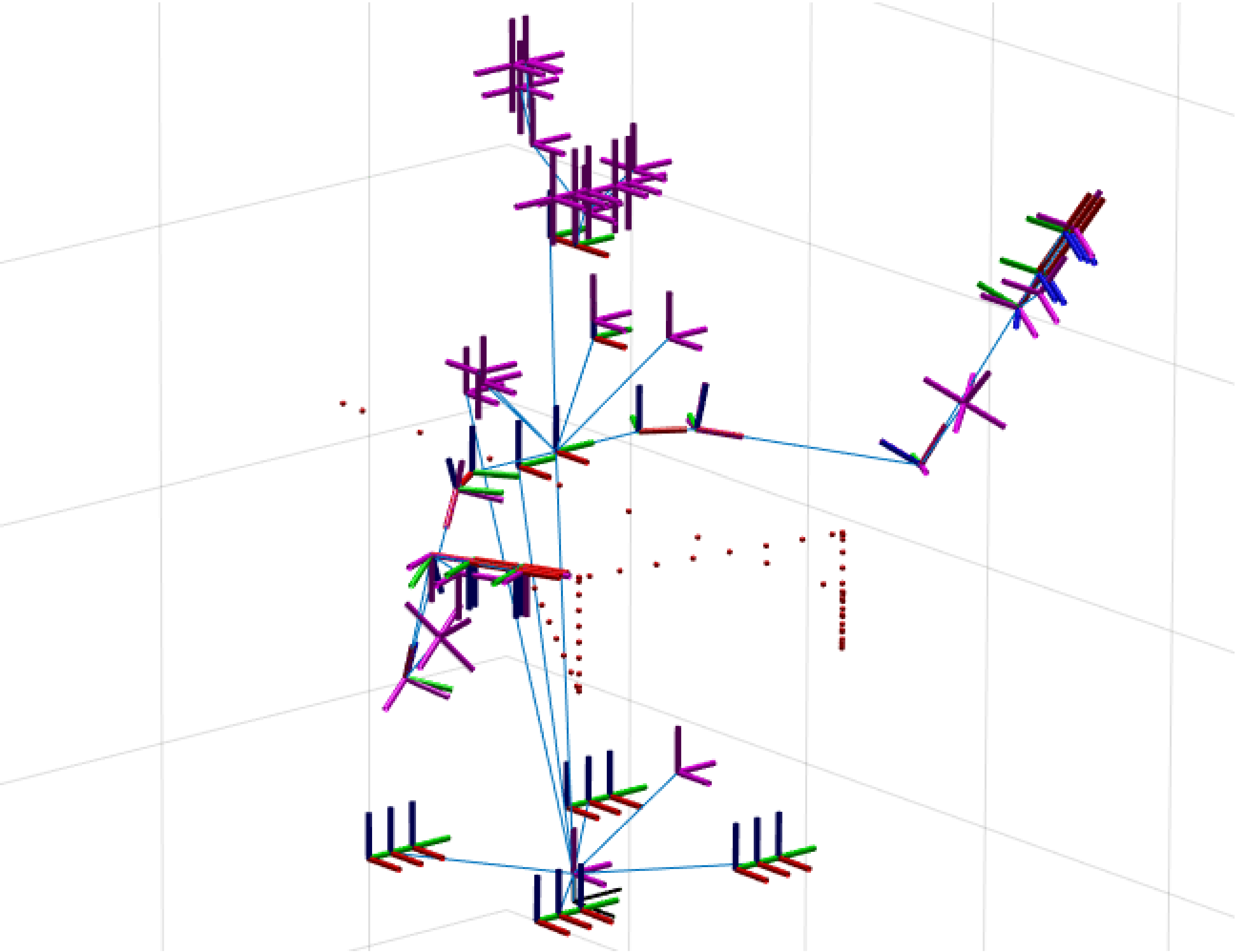
```
motionTask = {'Release', T1, TGrasp, 'Grasp', T2, T3, TRelease, 'Release', T4};
```

Execute each task specified in `motionTask` one by one. Send commands using the specified helper functions.

```
for i = 1: length(motionTask)

    if strcmp(motionTask{i}, 'Grasp')
        exampleHelperSendPR2GripperCommand('right',0.0,1000,false);
        continue
    end

    if strcmp(motionTask{i}, 'Release')
        exampleHelperSendPR2GripperCommand('right',0.1,-1,true);
        continue
    end

    Tf = motionTask{i};
    % Get current joint state
    jntState = receive(jointSub);
    jntPos = exampleHelperJointMsgToStruct(pr2, jntState);

    T0 = getTransform(pr2, jntPos, endEffectorName);

    % Interpolating between key waypoints
    numWaypoints = 10;
    [s, sd, sdd, tvec] = trapveltraj([0 1], numWaypoints, 'AccelTime', 0.4); % Relatively slow ra
    TWaypoints = transformtraj(T0, Tf, [0 1], tvec, 'TimeScaling', [s; sd; sdd]); % end-effector
    jntPosWaypoints = repmat(initialGuess, numWaypoints, 1); % joint position waypoints

    rArmJointNames = rGoalMsg.Trajectory.JointNames;
    rArmJntPosWaypoints = zeros(numWaypoints, numel(rArmJointNames));

    % Calculate joint position for each end-effector pose waypoint using IK
    for k = 1:numWaypoints
        jntPos = ik(endEffectorName, TWaypoints(:,:,k), weights, initialGuess);
        jntPosWaypoints(k, :) = jntPos;
        initialGuess = jntPos;

        % Extract right arm joint positions from jntPos
        rArmJointPos = zeros(size(rArmJointNames));
        for n = 1:length(rArmJointNames)
            rn = rArmJointNames{n};
            idx = strcmp({jntPos.JointName}, rn);
            rArmJointPos(n) = jntPos(idx).JointPosition;
        end
        rArmJntPosWaypoints(k,:) = rArmJointPos';
    end

    % Time points corresponding to each waypoint
    timePoints = linspace(0,3,numWaypoints);

    % Estimate joint velocity trajectory numerically
    h = diff(timePoints); h = h(1);
    jntTrajectoryPoints = arrayfun(@(~) rosmessage('trajectory_msgs/JointTrajectoryPoint'), zeros
    [~, rArmJntVelWaypoints] = gradient(rArmJntPosWaypoints, h);
    for m = 1:numWaypoints
        jntTrajectoryPoints(m).Positions = rArmJntPosWaypoints(m,:);
        jntTrajectoryPoints(m).Velocities = rArmJntVelWaypoints(m,:);
        jntTrajectoryPoints(m).TimeFromStart = rosduration(timePoints(m));
    end

    % Visualize robot motion and end-effector trajectory in MATLAB(R)
    hold on
    for j = 1:numWaypoints
        show(pr2, jntPosWaypoints(j,:),'PreservePlot', false);
        exampleHelperShowEndEffectorPos(TWaypoints(:,:,j));
        drawnow;
        pause(0.1);
    end

    % Send the right arm trajectory to the robot
    rGoalMsg.Trajectory.Points = jntTrajectoryPoints;
    sendGoalAndWait(rArm, rGoalMsg);

end
```


2
1.5
1
0.5
Z
0
-0.5
-1
-1.5
-2
z
x
y
-2
-1
0
1
Y
22
1
X
0
-1
-2





Wrap Up

Move arm back to starting position.

```
exampleHelperSendPR2GripperCommand('r',0.0,-1)
rGoalMsg.Trajectory.Points = tjPoint2;
sendGoal(rArm, rGoalMsg);
```

Call `rosshutdown` to shutdown ROS network and disconnect from the robot.

```
rosshutdown
```

```
Shutting down global node /matlab_global_node_03004 with NodeURI http://192.168.116.1:64988/
```

Plan a Reaching Trajectory With Multiple Kinematic Constraints

This example shows how to use generalized inverse kinematics to plan a joint-space trajectory for a robotic manipulator. It combines multiple constraints to generate a trajectory that guides the gripper to a cup resting on a table. These constraints ensure that the gripper approaches the cup in a straight line and that the gripper remains at a safe distance from the table, without requiring the poses of the gripper to be determined in advance.

Set Up the Robot Model

This example uses a model of the KUKA LBR iiwa, a 7 degree-of-freedom robot manipulator. `importrobot` generates a `rigidBodyTree` model from a description stored in a Unified Robot Description Format (URDF) file.

```
lbr = importrobot('iiwa14.urdf'); % 14 kg payload version
lbr.DataFormat = 'row';
gripper = 'iiwa_link_ee_kuka';
```

Define dimensions for the cup.

```
cupHeight = 0.2;
cupRadius = 0.05;
cupPosition = [-0.5, 0.5, cupHeight/2];
```

Add a fixed body to the robot model representing the center of the cup.

```
body = rigidBody('cupFrame');
setFixedTransform(body.Joint, trvec2tform(cupPosition))
addBody(lbr, body, lbr.BaseName);
```

Define the Planning Problem

The goal of this example is to generate a sequence of robot configurations that satisfy the following criteria:

- Start in the home configuration
- No abrupt changes in robot configuration
- Keep the gripper at least 5 cm above the "table" (z = 0)
- The gripper should be aligned with the cup as it approaches
- Finish with the gripper 5 cm from the center of the cup

This example utilizes constraint objects to generate robot configurations that satisfy these criteria. The generated trajectory consists of five configuration waypoints. The first waypoint, `q0`, is set as the home configuration. Pre-allocate the rest of the configurations in `qWaypoints` using `repmat`.

```
numWaypoints = 5;
q0 = homeConfiguration(lbr);
qWaypoints = repmat(q0, numWaypoints, 1);
```

Create a `generalizedInverseKinematics` solver that accepts the following constraint inputs:

- Cartesian bounds - Limits the height of the gripper
- A position target - Specifies the position of the cup relative to the gripper.

- An aiming constraint - Aligns the gripper with the cup axis
- An orientation target - Maintains a fixed orientation for the gripper while approaching the cup
- Joint position bounds - Limits the change in joint positions between waypoints.

```
gik = generalizedInverseKinematics('RigidBodyTree', lbr, ...
    'ConstraintInputs', {'cartesian','position','aiming','orientation','joint'})
```

```
gik = 
  generalizedInverseKinematics with properties:

      NumConstraints: 5
    ConstraintInputs: {1x5 cell}
       RigidBodyTree: [1x1 rigidBodyTree]
     SolverAlgorithm: 'BFGSGradientProjection'
    SolverParameters: [1x1 struct]
```

Create Constraint Objects

Create the constraint objects that are passed as inputs to the solver. These object contain the parameters needed for each constraint. Modify these parameters between calls to the solver as necessary.

Create a Cartesian bounds constraint that requires the gripper to be at least 5 cm above the table (negative z direction). All other values are given as `inf` or `-inf`.

```
heightAboveTable = constraintCartesianBounds(gripper);
heightAboveTable.Bounds = [-inf, inf; ...
                           -inf, inf; ...
                           0.05, inf]
```

```
heightAboveTable = 
  constraintCartesianBounds with properties:

        EndEffector: 'iiwa_link_ee_kuka'
      ReferenceBody: ''
    TargetTransform: [4x4 double]
             Bounds: [3x2 double]
            Weights: [1 1 1]
```

Create a constraint on the position of the cup relative to the gripper, with a tolerance of 5 mm.

```
distanceFromCup = constraintPositionTarget('cupFrame');
distanceFromCup.ReferenceBody = gripper;
distanceFromCup.PositionTolerance = 0.005
```

```
distanceFromCup = 
  constraintPositionTarget with properties:

          EndEffector: 'cupFrame'
        ReferenceBody: 'iiwa_link_ee_kuka'
       TargetPosition: [0 0 0]
    PositionTolerance: 0.0050
              Weights: 1
```

Create an aiming constraint that requires the z-axis of the `iiwa_link_ee` frame to be approximately vertical, by placing the target far above the robot. The `iiwa_link_ee` frame is oriented such that this constraint aligns the gripper with the axis of the cup.

```
alignWithCup = constraintAiming('iiwa_link_ee');
alignWithCup.TargetPoint = [0, 0, 100]
```

```
alignWithCup = 
  constraintAiming with properties:

         EndEffector: 'iiwa_link_ee'
       ReferenceBody: ''
         TargetPoint: [0 0 100]
    AngularTolerance: 0
             Weights: 1
```

Create a joint position bounds constraint. Set the `Bounds` property of this constraint based on the previous configuration to limit the change in joint positions.

```
limitJointChange = constraintJointBounds(lbr)
```

```
limitJointChange = 
  constraintJointBounds with properties:

     Bounds: [7x2 double]
    Weights: [1 1 1 1 1 1 1]
```

Create an orientation constraint for the gripper with a tolerance of one degree. This constraint requires the orientation of the gripper to match the value specified by the `TargetOrientation` property. Use this constraint to fix the orientation of the gripper during the final approach to the cup.

```
fixOrientation = constraintOrientationTarget(gripper);
fixOrientation.OrientationTolerance = deg2rad(1)
```

```
fixOrientation = 
  constraintOrientationTarget with properties:

             EndEffector: 'iiwa_link_ee_kuka'
           ReferenceBody: ''
       TargetOrientation: [1 0 0 0]
    OrientationTolerance: 0.0175
                 Weights: 1
```

Find a Configuration That Points at the Cup

This configuration should place the gripper at a distance from the cup, so that the final approach can be made with the gripper properly aligned.

```
intermediateDistance = 0.3;
```

Constraint objects have a `Weights` property which determines how the solver treats conflicting constraints. Setting the weights of a constraint to zero disables the constraint. For this configuration, disable the joint position bounds and orientation constraint.

```
limitJointChange.Weights = zeros(size(limitJointChange.Weights));
fixOrientation.Weights = 0;
```

Set the target position for the cup in the gripper frame. The cup should lie on the z-axis of the gripper at the specified distance.

```
distanceFromCup.TargetPosition = [0,0,intermediateDistance];
```

Solve for the robot configuration that satisfies the input constraints using the `gik` solver. You must specify all the input constraints. Set that configuration as the second waypoint.

```
[qWaypoints(2,:),solutionInfo] = gik(q0, heightAboveTable, ...
                       distanceFromCup, alignWithCup, fixOrientation, ...
                       limitJointChange);
```

Find Configurations That Move Gripper to the Cup Along a Straight Line

Re-enable the joint position bound and orientation constraints.

```
limitJointChange.Weights = ones(size(limitJointChange.Weights));
fixOrientation.Weights = 1;
```

Disable the align-with-cup constraint, as the orientation constraint makes it redundant.

```
alignWithCup.Weights = 0;
```

Set the orientation constraint to hold the orientation based on the previous configuration (`qWaypoints(2,:)`). Get the transformation from the gripper to the base of the robot model. Convert the homogeneous transformation to a quaternion.

```
fixOrientation.TargetOrientation = ...
    tform2quat(getTransform(lbr,qWaypoints(2,:),gripper));
```

Define the distance between the cup and gripper for each waypoint

```
finalDistanceFromCup = 0.05;
distanceFromCupValues = linspace(intermediateDistance, finalDistanceFromCup, numWaypoints-1);
```

Define the maximum allowed change in joint positions between each waypoint.

```
maxJointChange = deg2rad(10);
```

Call the solver for each remaining waypoint.

```
for k = 3:numWaypoints
    % Update the target position.
    distanceFromCup.TargetPosition(3) = distanceFromCupValues(k-1);
    % Restrict the joint positions to lie close to their previous values.
    limitJointChange.Bounds = [qWaypoints(k-1,:)' - maxJointChange, ...
                               qWaypoints(k-1,:)' + maxJointChange];
    % Solve for a configuration and add it to the waypoints array.
    [qWaypoints(k,:),solutionInfo] = gik(qWaypoints(k-1,:), ...
                                         heightAboveTable, ...
                                         distanceFromCup, alignWithCup, ...
                                         fixOrientation, limitJointChange);
end
```

Visualize the Generated Trajectory

Interpolate between the waypoints to generate a smooth trajectory. Use `pchip` to avoid overshoots, which might violate the joint limits of the robot.

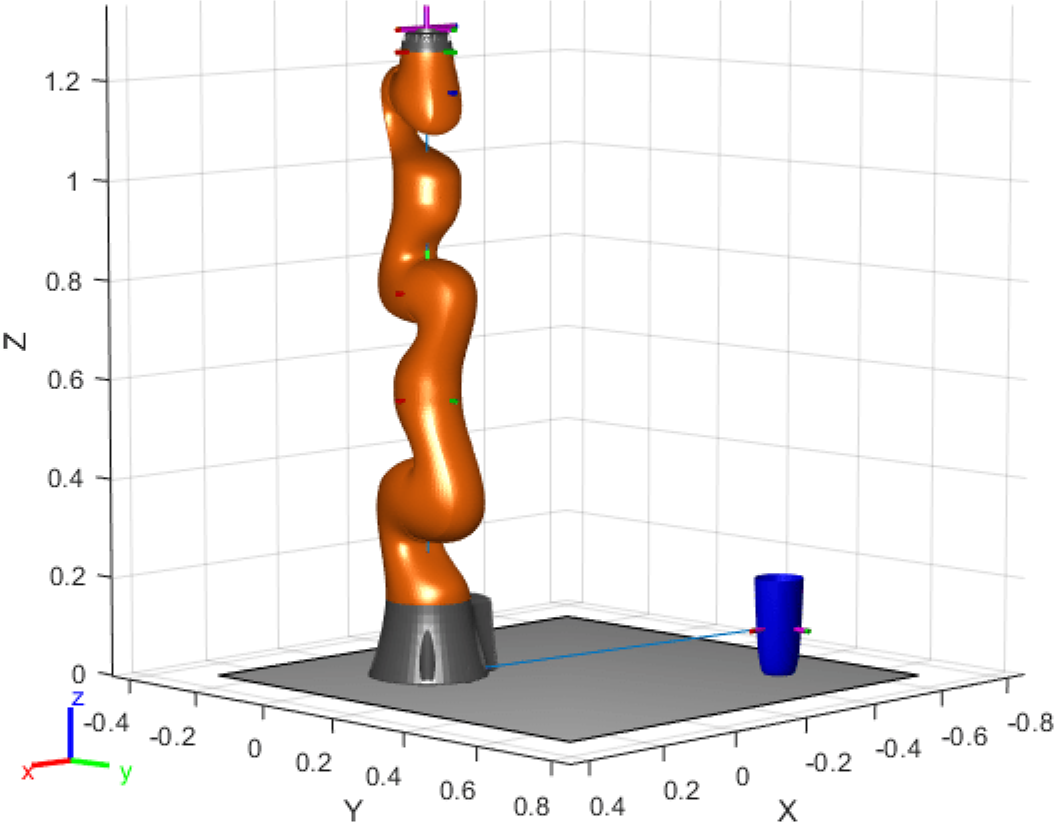
```
framerate = 15;
r = rateControl(framerate);
tFinal = 10;
tWaypoints = [0,linspace(tFinal/2,tFinal,size(qWaypoints,1)-1)];
numFrames = tFinal*framerate;
qInterp = pchip(tWaypoints,qWaypoints',linspace(0,tFinal,numFrames))';
```

Compute the gripper position for each interpolated configuration.

```
gripperPosition = zeros(numFrames,3);
for k = 1:numFrames
    gripperPosition(k,:) = tform2trvec(getTransform(lbr,qInterp(k,:), ...
                                                   gripper));
end
```

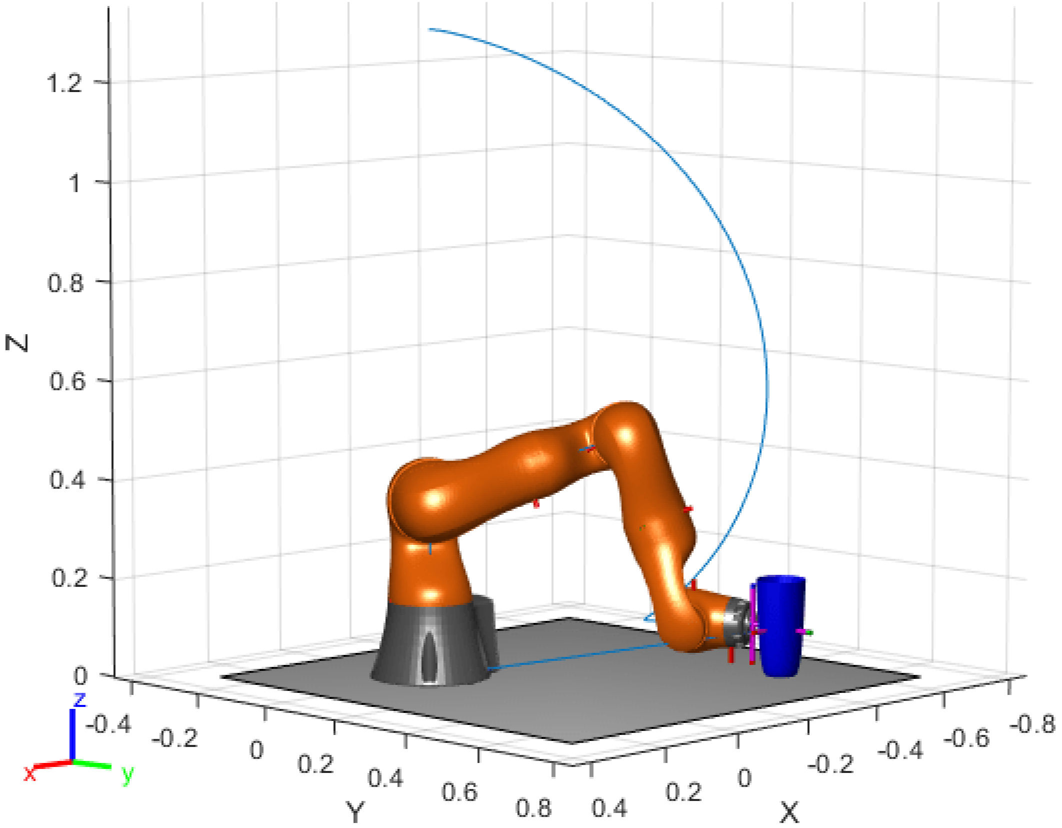
Show the robot in its initial configuration along with the table and cup

```
figure;
show(lbr, qWaypoints(1,:), 'PreservePlot', false);
hold on
exampleHelperPlotCupAndTable(cupHeight, cupRadius, cupPosition);
p = plot3(gripperPosition(1,1), gripperPosition(1,2), gripperPosition(1,3));
```



Animate the manipulator and plot the gripper position.

```
hold on
for k = 1:size(qInterp,1)
    show(lbr, qInterp(k,:), 'PreservePlot', false);
    p.XData(k) = gripperPosition(k,1);
    p.YData(k) = gripperPosition(k,2);
    p.ZData(k) = gripperPosition(k,3);
    waitfor(r);
end
hold off
```



If you want to save the generated configurations to a MAT-file for later use, execute the following:

```
>> save('lbr_trajectory.mat', 'tWaypoints', 'qWaypoints');
```

Obtain Collision Data for Manipulator Collision Checking

This example shows three ways to obtain a `rigidBodyTree` model with collision data. To see more in-depth examples that check for self-collisions or environment collision detection, see these other examples:

- "Check for Manipulator Self Collisions Using Collision Meshes" on page 1-214
- "Check for Environmental Collisions with Manipulators" on page 1-217
- "Plan and Execute Collision-Free Trajectories Using KINOVA Gen3 Manipulator" on page 1-261
- "Pick-and-Place Workflow Using Stateflow for MATLAB" on page 1-286

URDF Import

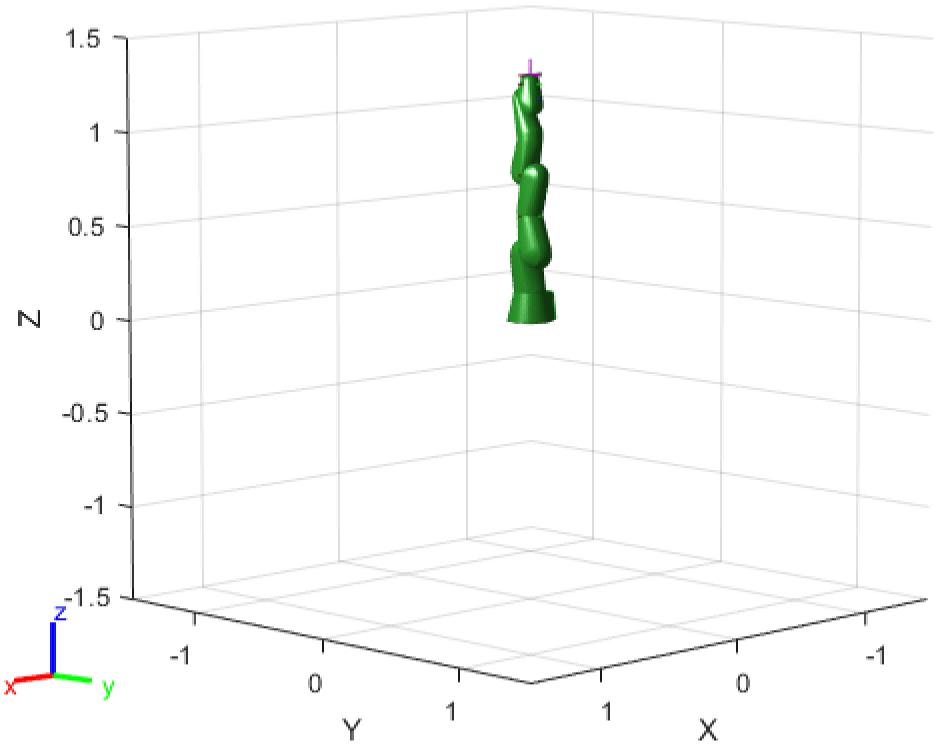
Many robots come with collision meshes or primitives specified in the Unified Robot Definition Format (URDF) file.

The KUKA® IIWA robot comes with a set of collision meshes which are simplified versions of the visual meshes. Call the `importrobot`function to generate a `rigidBodyTree` object from the URDF file. Set the output format for configurations to `"column"`.

```
iiwa = importrobot("iiwa14.urdf");
iiwa.DataFormat = "column";
```

Visually inspect the collision meshes of the robot.

```
show(iiwa,"Visuals","off","Collisions","on");
```



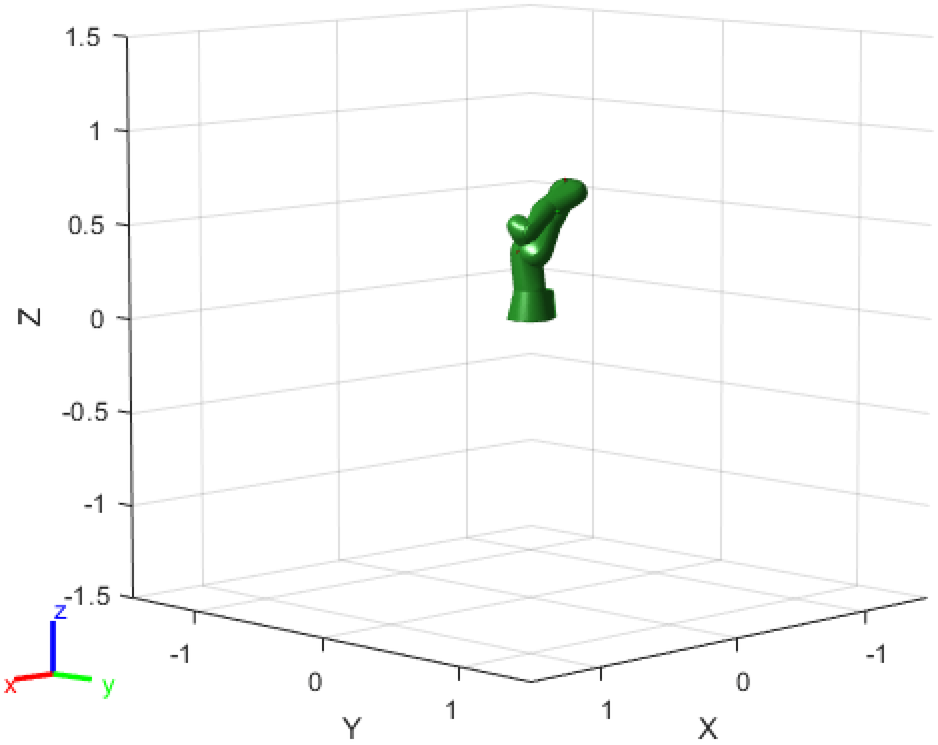
Check for Self-Collisions at a Specified Configuration

Specify a configuration that has a self collision. The `checkCollision` function indicates whether a robot is colliding with itself in a particular configuration. Visualize the configuration.

```
config = [0 -pi/4 pi 0.9*pi 0 -pi/2 0]';
checkCollision(iiwa,config)
```

```
ans = logical
   1
```

```
show(iiwa,config,"Visuals","off","Collisions","on");
```



Load Provided Models

Robotics System Toolbox™ provides common robot models with collision data accessed using the `loadrobot` function.

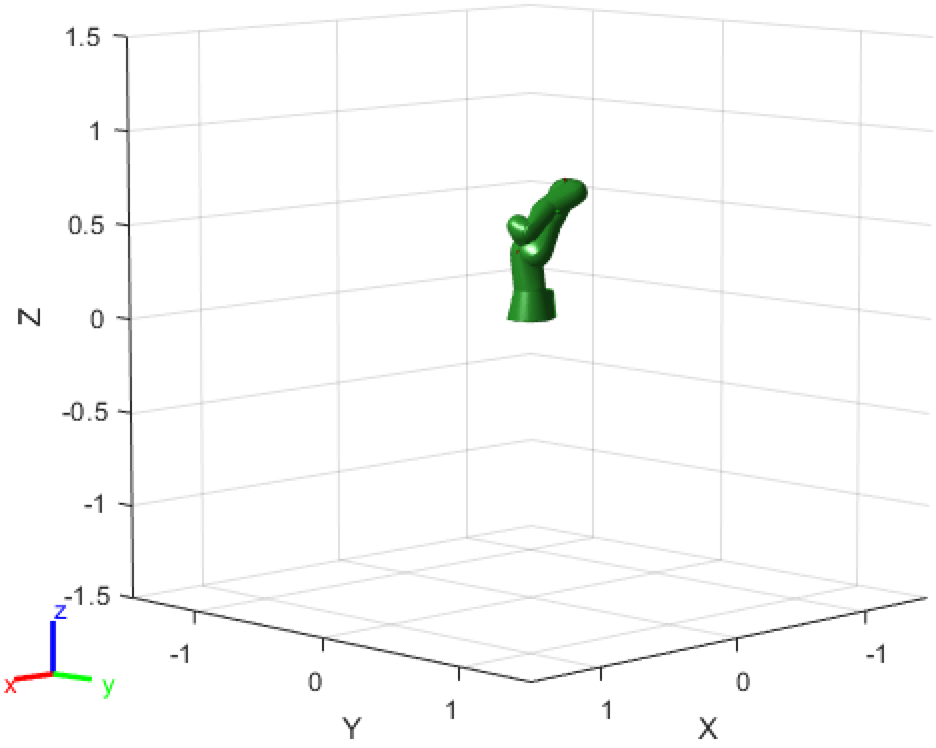
```
kukaIiwa14 = loadrobot("kukaIiwa14","DataFormat","column");
checkCollision(kukaIiwa14,config)
```

```
ans = logical
   1
```

```
config = [0 -pi/4 pi 0.9*pi 0 -pi/2 0]';
```

Visualize the robot with the collision meshes visible.

```
show(kukaIiwa14,config,"Visuals","off","Collisions","on");
```



Adding Individual Collision Obects

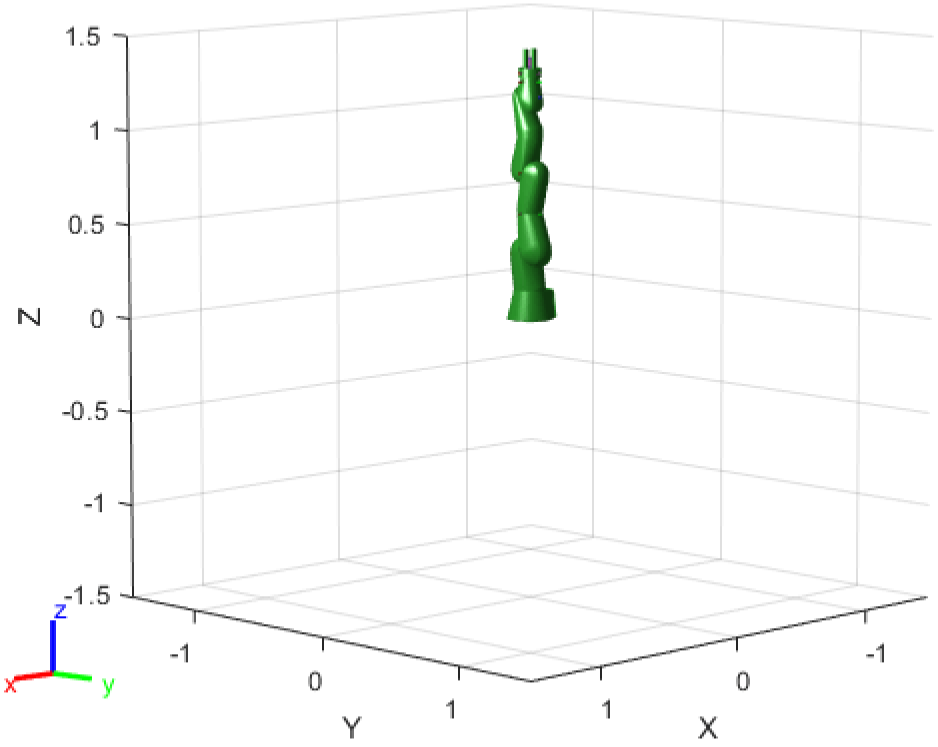
The `addCollision` function enables you to add collision objects to any rigid body on the robot as basic shapes (box, sphere,cylinder). You could build your entire robot with these collision geometries, but this is generally less accurate than higher fidelity mesh definitions.

For this example, notice that the loaded IIWA robot model is missing a gripper. Add a gripper made of collision primitives to the `iiwa_link_ee` body on the robot.

```
addCollision(iiwa.Bodies{end},"cylinder",[0.06,0.05])
addCollision(iiwa.Bodies{end},"box",[0.02,0.02,0.15],trvec2tform([0.03,0,0.05]))
addCollision(iiwa.Bodies{end},"box",[0.02,0.02,0.15],trvec2tform([-0.03,0,0.05]))
```

Visualize the robot. Notice the gripper is attached.

```
show(iiwa,"Visuals","off","Collisions","on");
```


1.5
1
0.5
0
-0.5
-1
-1.5
Z
-1
0
1
Y
1
0
-1
X
z
x
y

Check for Manipulator Self Collisions Using Collision Meshes

This example shows how to check for manipulator self-collisions when executing a trajectory. The collision meshes are loaded via the `<collision>` tag defined in the URDF of a robot model. The following related examples show how to load collision data in other ways, and how to check for environmental collisions:

- "Obtain Collision Data for Manipulator Collision Checking" on page 1-210
- "Check for Environmental Collisions with Manipulators" on page 1-217

Create Robot Representation

Import a URDF file of the KUKA® IIWA-14 serial manipulator as a `rigidBodyTree` model. The URDF captures the collision mesh files for the rigid bodies in the robot. To individually add collision objects to a rigid body, you can use the `addCollision` function. To load a provided robot model with collision objects attached, see the `loadrobot` function.

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

Generate Trajectory and Check for Collisions

Specify a start and end configuration as a set of joint positions. These positions should be collision free.

```
startConfig = [0 -pi/4 pi 3*pi/2 0 -pi/2 pi/8]';
goalConfig = [0 -pi/4 pi 3*pi/4 0 -pi/2 pi/8]';
```

Find a joint space trajectory that connects the two configurations within three seconds.

```
q = trapveltraj([startConfig goalConfig],100,'EndTime',3);
```

To verify this output trajectory does not contain self-collisions, iterate over the output samples and see if any points are in collision using the `checkCollision` function.

While iterating through the trajectory `q`, call the `checkCollision` function on every configuration in the trajectory. Turn on exhaustive checking to continue checking for collisions after the first is detected.

The `isConfigInCollision` variable tracks the collision status of each configuration. The `sepDistForConfig` tracks the separation distance between the bodies of the robot. For each collision, the pair of body indices are stored in the `configCollisionPairs` variable. Note that neighboring bodies are not checked as they are always in contact via the joint that connects them.

```
isConfigInCollision = false(100,1);
configCollisionPairs = cell(100,1);
sepDistForConfig = zeros(iiwa.NumBodies+1,iiwa.NumBodies+1,100);
for i = 1:length(q)
    [isColliding, sepDist] = checkCollision(iiwa,q(:,i),'Exhaustive','on');
    isConfigInCollision(i) = isColliding;
    sepDistForConfig(:,:,i) = sepDist;
end
```

To find out the indices of the bodies in collision, find which entries in the `sepDistForConfig` are `NaN`. `septDist` is a symmetric matrix, so the same value is returned in indexes with flipped indexes. Simplify the list by using `unique`.

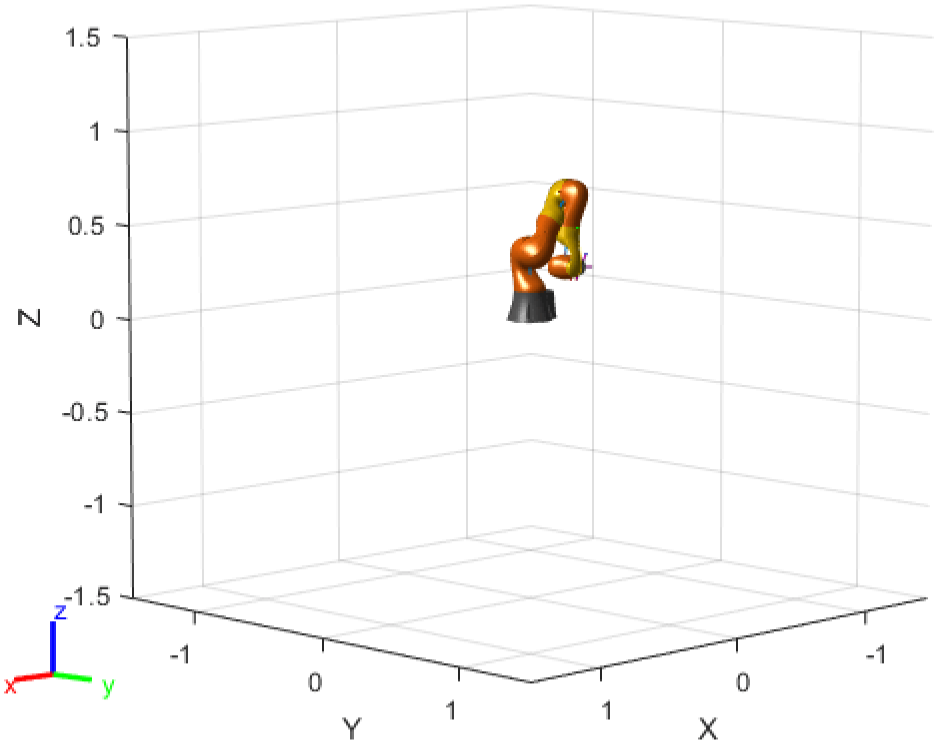
```
for i = 1:length(q)
    sepDist = sepDistForConfig(:,:,i);
    [body1Idx,body2Idx] = find(isnan(sepDist));

    collidingPairs = unique(sort([body1Idx,body2Idx],2));
    configCollisionPairs{i} = collidingPairs;
end
```

By inspecting the output, you can see the planned trajectory goes through a series of collisions. Visualize the configuration where the first collision occurs and highlight the bodies.

```
any(isConfigInCollision)
```

```
ans = logical
   1
```

```
firstCollisionIdx = find(isConfigInCollision,1);

% Visualize the first configuration that is in collision.
figure;
show(iiwa,q(:,firstCollisionIdx));
exampleHelperHighlightCollisionBodies(iiwa,configCollisionPairs{firstCollisionIdx}+1,gca);
```



Generate a Collision-Free Trajectory

This first collision actually occurs at the starting configuration because a joint position is specified past its limits. Call `wrapToPi` to limit the starting positions of the joints.

Generate a new trajectory and check for collisions again.

```
newStartConfig = wrapToPi(startConfig);
q = trapveltraj([newStartConfig goalConfig],100,'EndTime',3);

isConfigInCollision = false(100,1);
configCollisionPairs = cell(100,1);
for i = 1:length(q)
    isColliding = checkCollision(iiwa,q(:,i),'Exhaustive','on');
    isConfigInCollision(i) = isColliding;
end
```

After checking the whole trajectory, no collisions are found.

```
any(isConfigInCollision)
```

```
ans = logical
   0
```

Check for Environmental Collisions with Manipulators

Generate a collision-free trajectory in a constrained workspace.

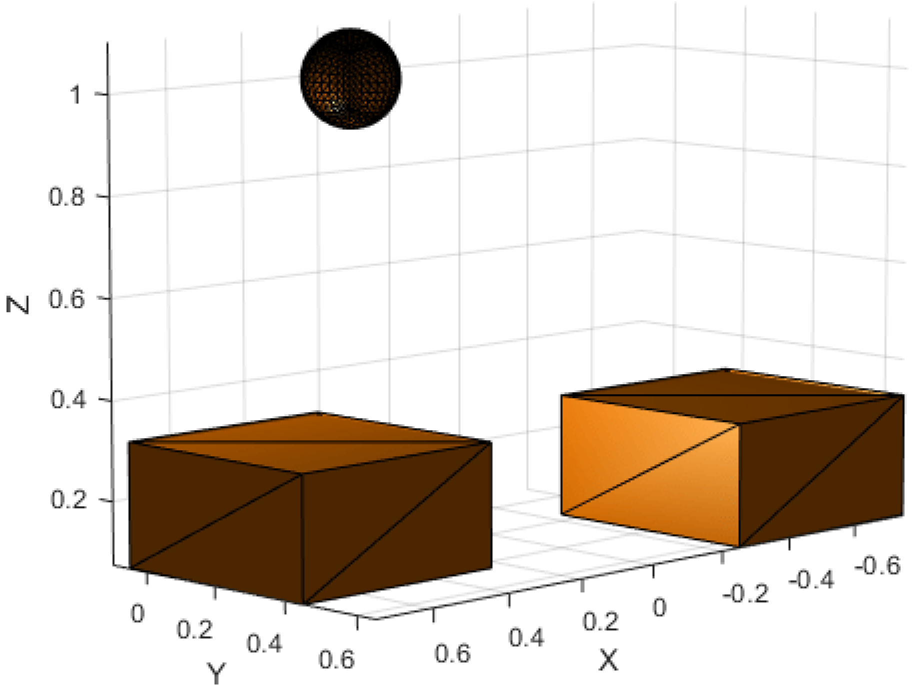
Define Collision Environment

Create a simple environment using collision primitives. This example creates a scene where a robot is in a workspace and has to move objects from one table to another. The robot must also avoid a circular light fixture above the workspace. Model the tables as two boxes and a sphere and specify their pose in the world. More complex environments can be created using `collisionMesh` objects.

```
% Create two platforms
platform1 = collisionBox(0.5,0.5,0.25);
platform1.Pose = trvec2tform([-0.5 0.4 0.2]);

platform2 = collisionBox(0.5,0.5,0.25);
platform2.Pose = trvec2tform([0.5 0.2 0.2]);

% Add a light fixture, modeled as a sphere
lightFixture = collisionSphere(0.1);
lightFixture.Pose = trvec2tform([.2 0 1]);

% Store in a cell array for collision-checking
worldCollisionArray = {platform1 platform2 lightFixture};
```

Visualize the environment using a helper function that iterates through the collision array.

```
ax = exampleHelperVisualizeCollisionEnvironment(worldCollisionArray);
```



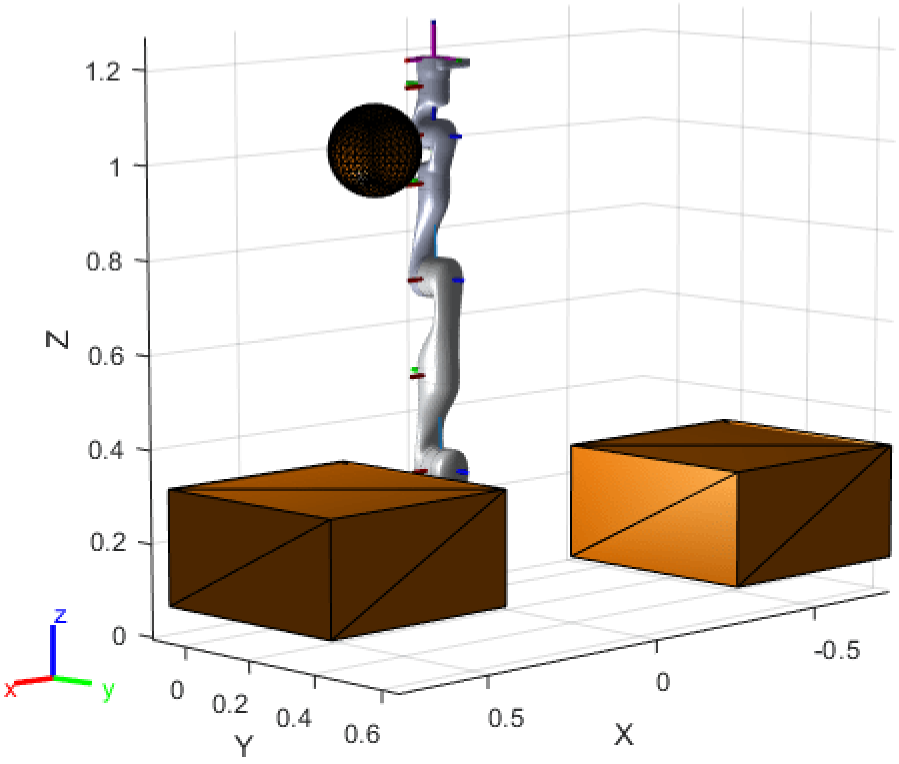
Add Robot To SEnvironment

Load a Kinova manipulator model as a `rigidBodyTree` object using the `loadrobot` function.

```
robot = loadrobot("kinovaGen3","DataFormat","column","Gravity",[0 0 -9.81]);
```

Show the robot in the environment using the same axes as the collision objects. The robot base is fixed to the origin of the world.

```
show(robot,homeConfiguration(robot),"Parent",ax);
```



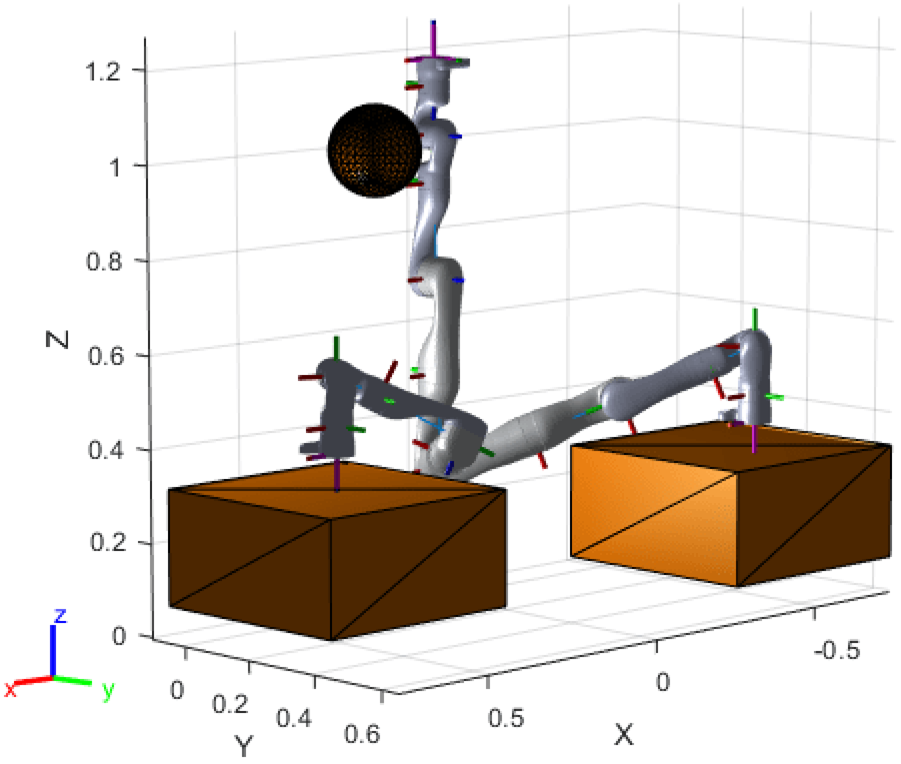
Generate a trajectory and check for collisions

Define a start and end pose using position and orientation vectors that are combined using transformation matrix multiplication.

```
startPose = trvec2tform([-0.5,0.5,0.4])*axang2tform([1 0 0 pi]);
endPose = trvec2tform([0.5,0.2,0.4])*axang2tform([1 0 0 pi]);
```

Use `inverseKinematics` to solve for the joint positions based on the desired poses. Inspect manually to verify that the configurations are valid.

```
% Use a fixed random seed to ensure repeatable results
rng(0);
ik = inverseKinematics("RigidBodyTree",robot);
weights = ones(1,6);
startConfig = ik("EndEffector_Link",startPose,weights,robot.homeConfiguration);
endConfig = ik("EndEffector_Link",endPose,weights,robot.homeConfiguration);

% Show initial and final positions
show(robot,startConfig);
show(robot,endConfig);
```



Use a trapezoidal velocity profile to generate a smooth trajectory from the home position to the start position, and then to the end position.

```
q = trapveltraj([homeConfiguration(robot),startConfig,endConfig],200,"EndTime",2);
```

Check for collisions with the obstacles in the environment using the `checkCollision` function. Enable the `IgnoreSelfCollision` nad `Exhaustive`name-value arguments. Self collisions are ignored because the robot model joint limits prevent most self collisions. Exhausitive checking ensures the function calculates all seperation distances and continues searching for collisions after detecting the first collision.

The `sepDist` output stores the distances between robot bodies and the world collision objects as a matrix. Each row corresponds to a specific world collision object. Each column corresponds to a robot body. Values of `NaN` indicate a collision. Store the indexes of the collision as a cell array.

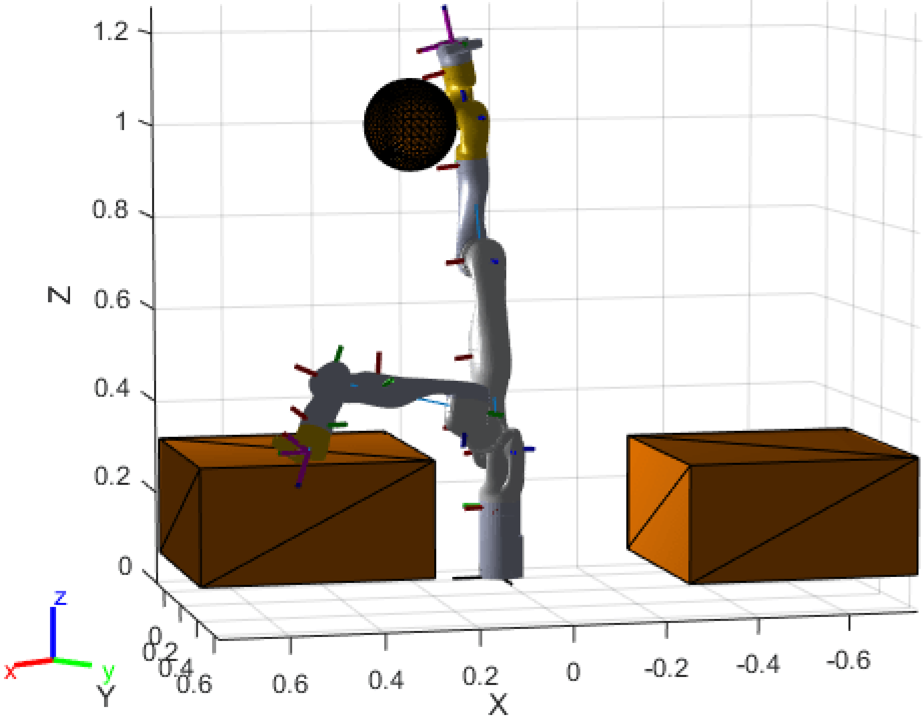
```
% Initialize outputs
inCollision = false(length(q), 1); % Check whether each pose is in collision
worldCollisionPairIdx = cell(length(q),1); % Provide the bodies that are in collision

for i = 1:length(q)

    [inCollision(i),sepDist] = checkCollision(robot,q(:,i),worldCollisionArray,"IgnoreSelfCollis

    [bodyIdx,worldCollisionObjIdx] = find(isnan(sepDist)); % Find collision pairs
    worldCollidingPairs = [bodyIdx,worldCollisionObjIdx];
    worldCollisionPairIdx{i} = worldCollidingPairs;

end
isTrajectoryInCollision = any(inCollision)
```

```
isTrajectoryInCollision = logical
   1
```

Inspect Detected Collisions

From the last step, two collisions are detected. Visualize these configurations to investigate further. Use the `exampleHelperHighlightCollisionBodies` function to highlight bodies based on the indices. You can see a collision occurs at the sphere and the table.

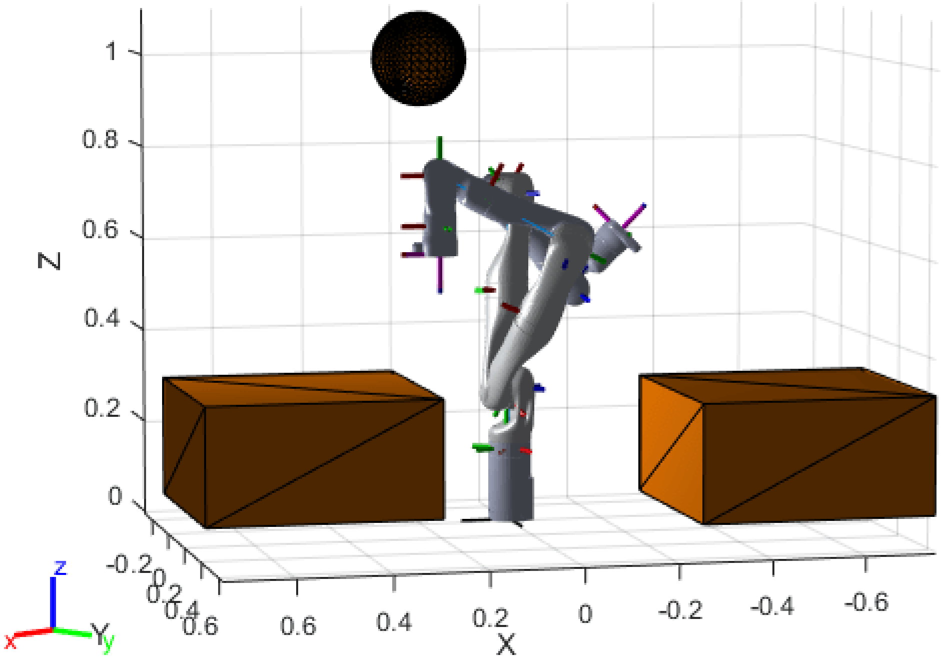
```
collidingIdx1 = find(inCollision,1);
collidingIdx2 = find(inCollision,1,"last");

% Identify the colliding rigid bodies.
collidingBodies1 = worldCollisionPairIdx{collidingIdx1}*[1 0]';
collidingBodies2 = worldCollisionPairIdx{collidingIdx2}*[1 0]';

% Visualize the environment.
ax = exampleHelperVisualizeCollisionEnvironment(worldCollisionArray);

% Add the robotconfigurations & highlight the colliding bodies.
show(robot,q(:,collidingIdx1),"Parent",ax,"PreservePlot",false);
exampleHelperHighlightCollisionBodies(robot,collidingBodies1 + 1,ax);
show(robot,q(:,collidingIdx2),"Parent"',ax);
exampleHelperHighlightCollisionBodies(robot,collidingBodies2 + 1,ax);
```



Generate Collision-Free Trajectory

To avoid these collisions, add intermediate waypoints to ensure the robot navigates around the obstacle.

```
intermediatePose1 = trvec2tform([-.3 -.2 .6])*axang2tform([0 1 0 -pi/4]); % Out and around the sp
intermediatePose2 = trvec2tform([0.2,0.2,0.6])*axang2tform([1 0 0 pi]); % Come in from above

intermediateConfig1 = ik("EndEffector_Link",intermediatePose1,weights,q(:,collidingIdx1));
intermediateConfig2 = ik("EndEffector_Link",intermediatePose2,weights,q(:,collidingIdx2));

% Show the new intermediate poses
ax = exampleHelperVisualizeCollisionEnvironment(worldCollisionArray);
show(robot,intermediateConfig1,"Parent",ax,"PreservePlot",false);
show(robot,intermediateConfig2,"Parent",ax);
```



Generate a new trajectory.

```
[q,qd,qdd,t] = trapveltraj([homeConfiguration(robot),intermediateConfig1,startConfig,intermediate
```

Verify that it is collision-free.

```
%Initialize outputs
inCollision = false(length(q),1); % Check whether each pose is in collision
for i = 1:length(q)
    inCollision(i) = checkCollision(robot,q(:,i),worldCollisionArray,"IgnoreSelfCollision","on")
end
isTrajectoryInCollision = any(inCollision)
```

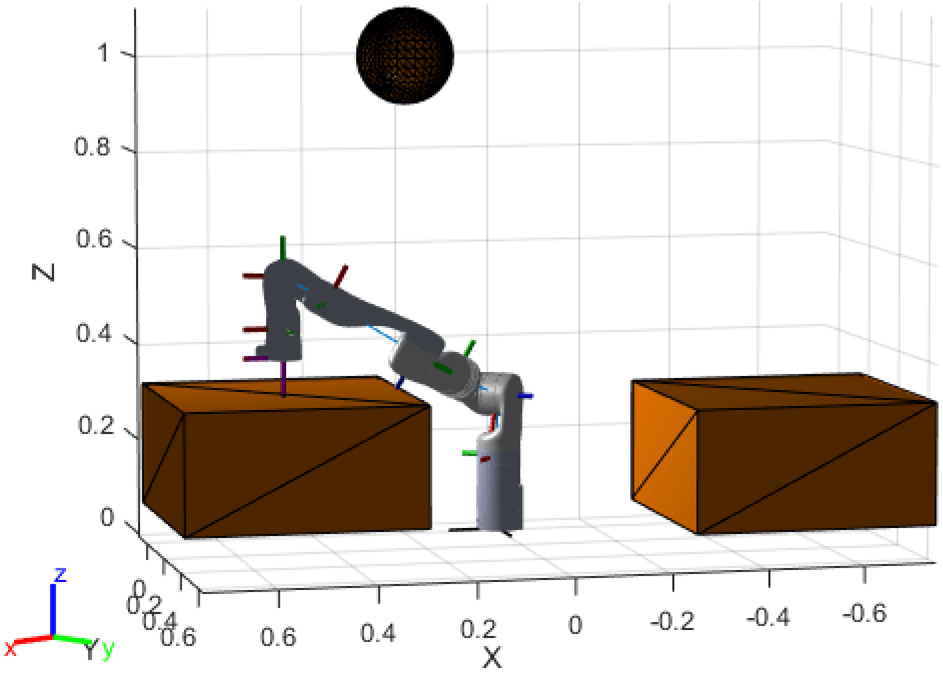
```
isTrajectoryInCollision = logical
   0
```

Visualize the Generated Trajectory

Animate the result.

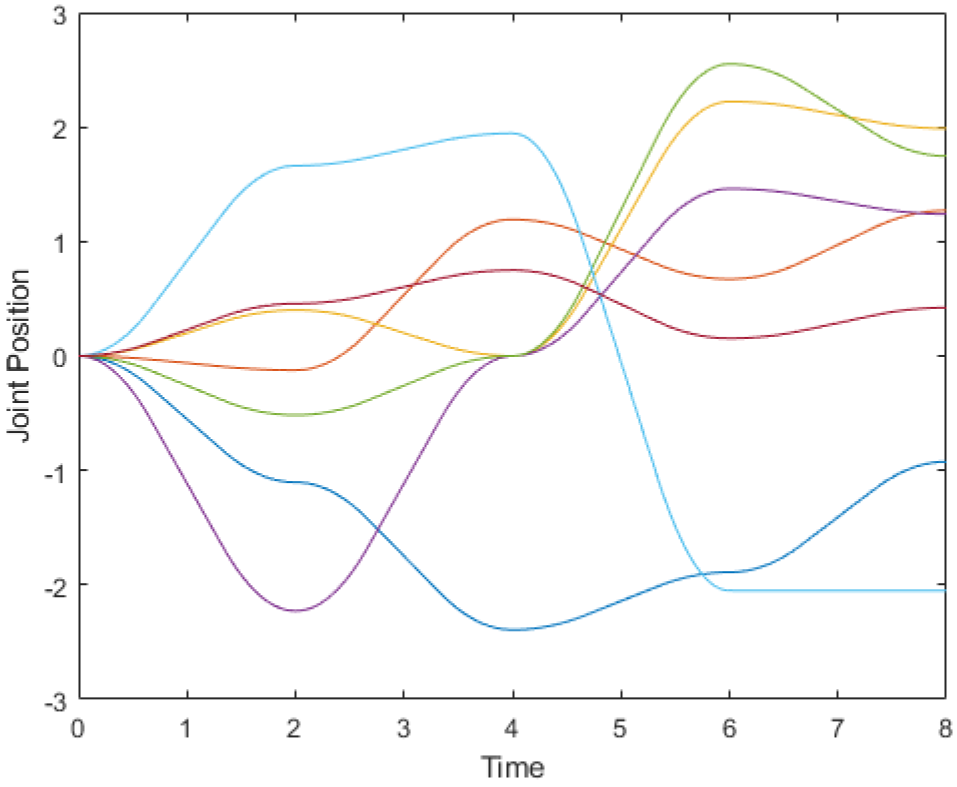
```
% Plot the environment
ax2 = exampleHelperVisualizeCollisionEnvironment(worldCollisionArray);

% Visualize the robot in its home configuration
show(robot,startConfig,"Parent",ax2);

% Update the axis size
```

```
axis equal

% Loop through the other positions
for i = 1:length(q)
    show(robot,q(:,i),"Parent",ax2,"PreservePlot",false);

    % Update the figure
    drawnow
end
```



Plot the joint positions over time.

```
figure
plot(t,q)
xlabel("Time")
ylabel("Joint Position")
```


3
2
1
0
-1
-2
-3
Joint Position
0
1
2
3
4
5
6
7
8
Time

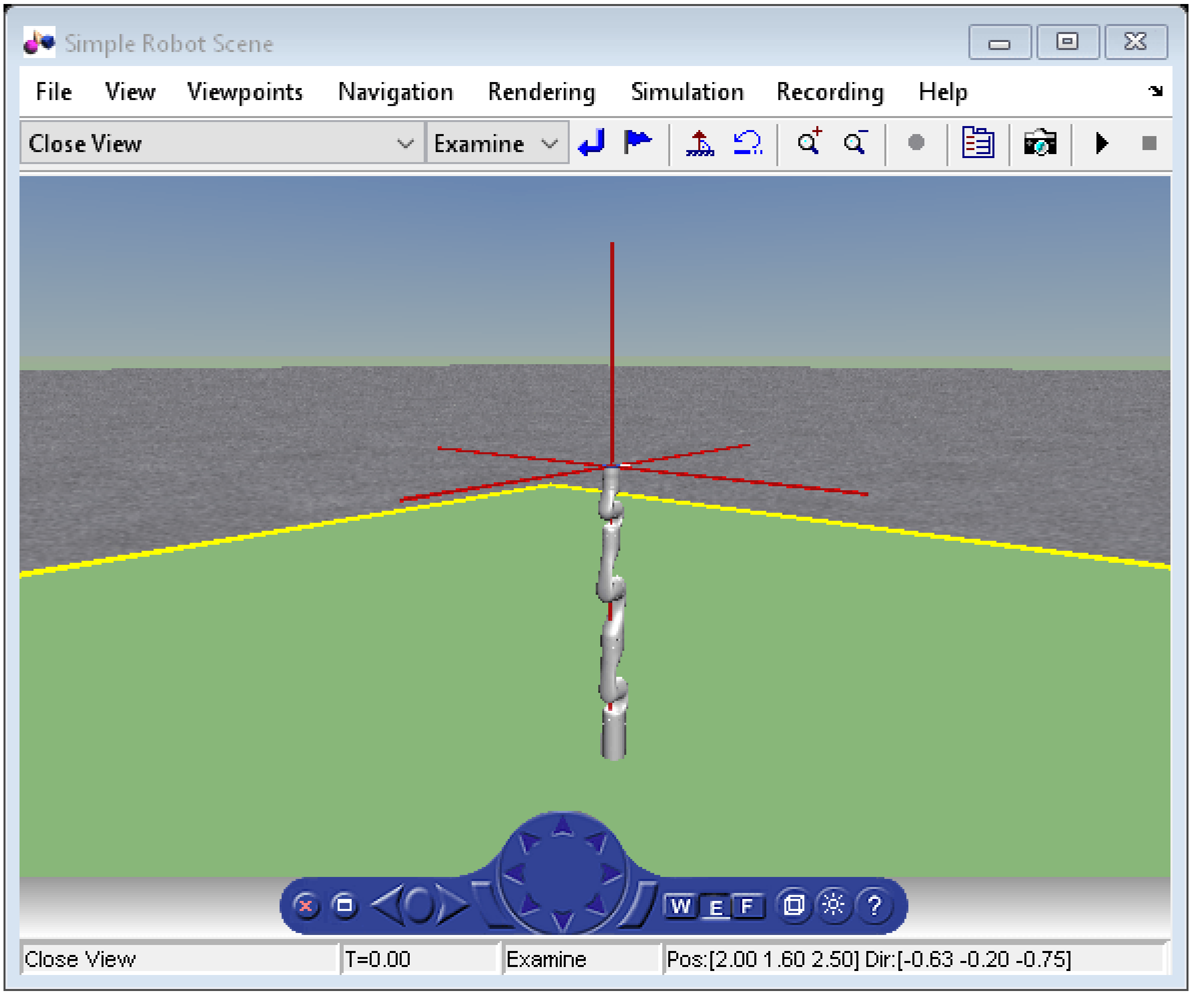
Visualize Manipulator Trajectory Tracking with Simulink 3D Animation

Simulate joint-space trajectories for a rigid body tree robot model and visualize the results with Simulink 3D Animation™.

Model Overview

Load the model with the following command:

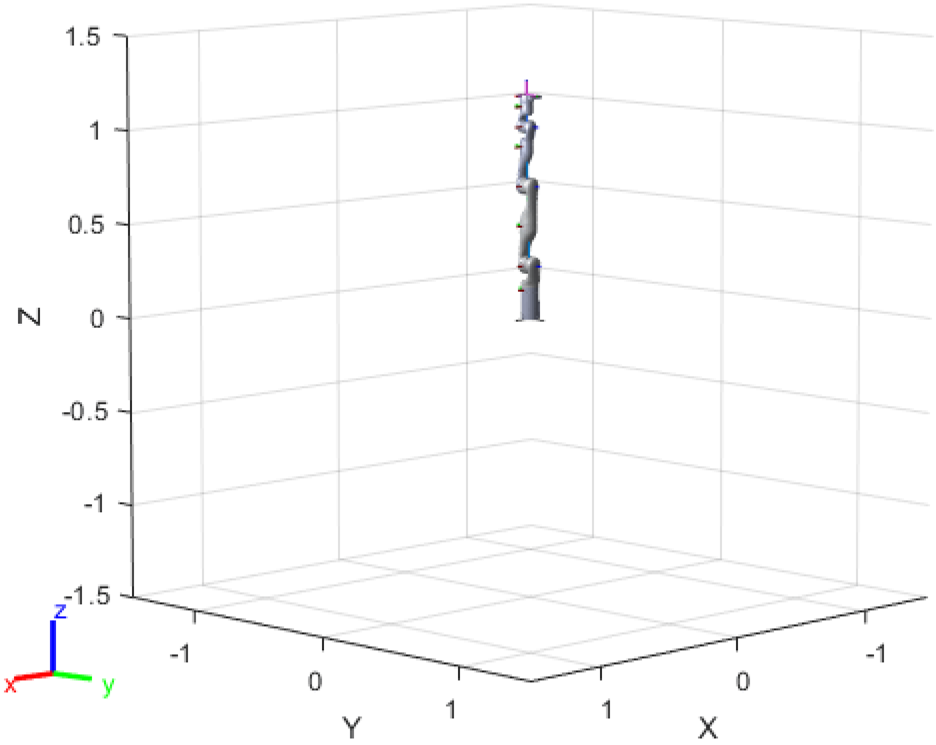
```
open_system("SL3DJointSpaceManipulatorTrajectory")
```



Close View | T=0.00 | Examine | Pos:[2.00 1.60 2.50] Dir:[-0.63 -0.20 -0.75]

This example uses a Kinova Gen3 manipulator, which is stored in the model workspace. However, load and visualize the robot with the following commands:

```
gen3 = loadrobot("kinovaGen3","DataFormat","column");
show(gen3);
```

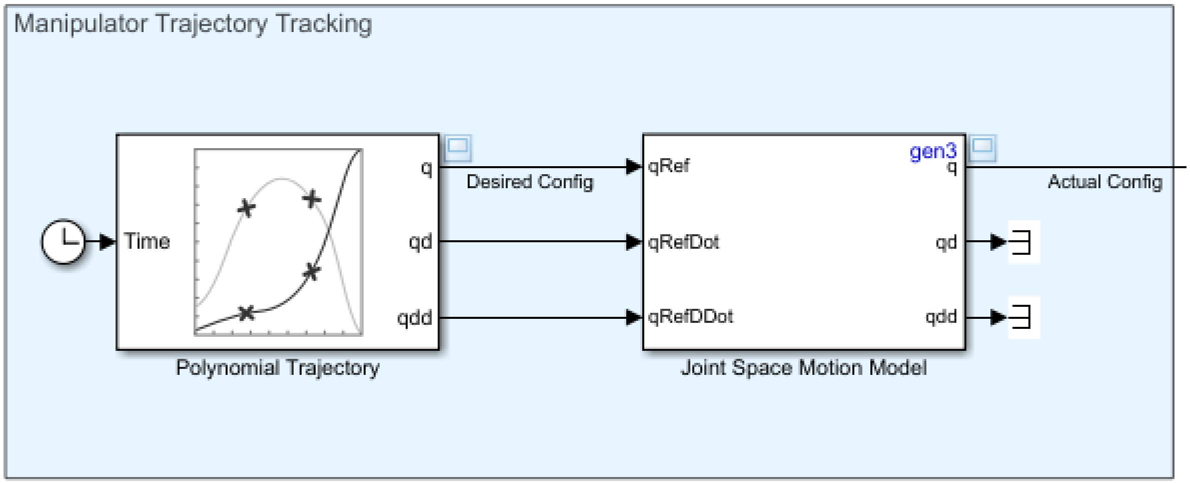


The model is split into two sections:

- Manipulator Trajectory Tracking
- Visualization in Simulink 3D Animation™

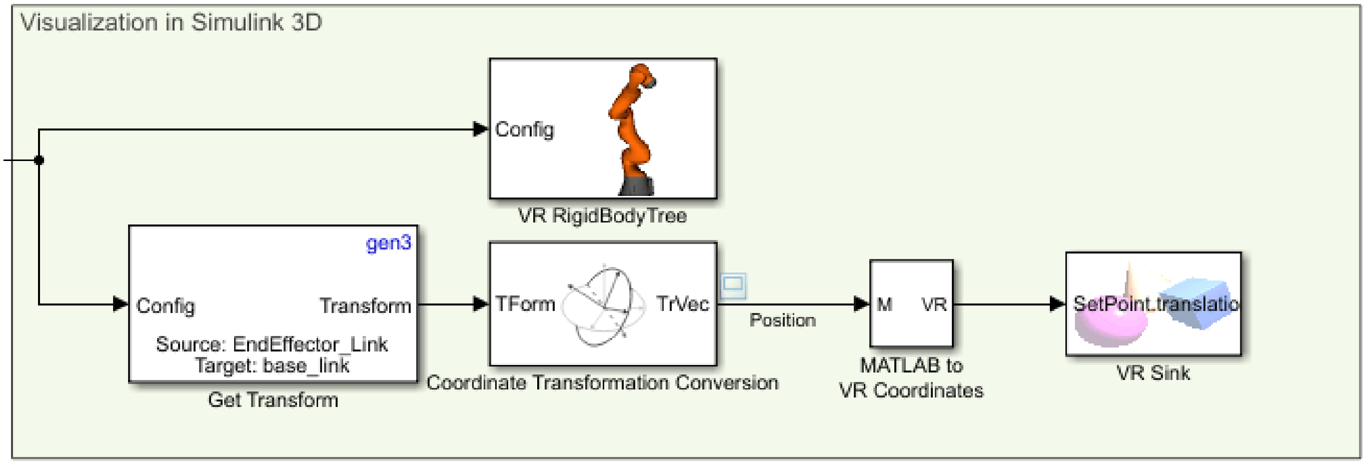
Manipulator Trajectory Tracking

The **Polynomial Trajectory** block generates continuous joint-space trajectories from random sets of waypoints in the range `[-0.375*pi 0.375*pi]`, stopping at each of the waypoints. The **Joint-Space Motion Model** block simulates the closed-loop tracking of these trajectories for a Kinova Gen3 manipulator with computed-torque control.



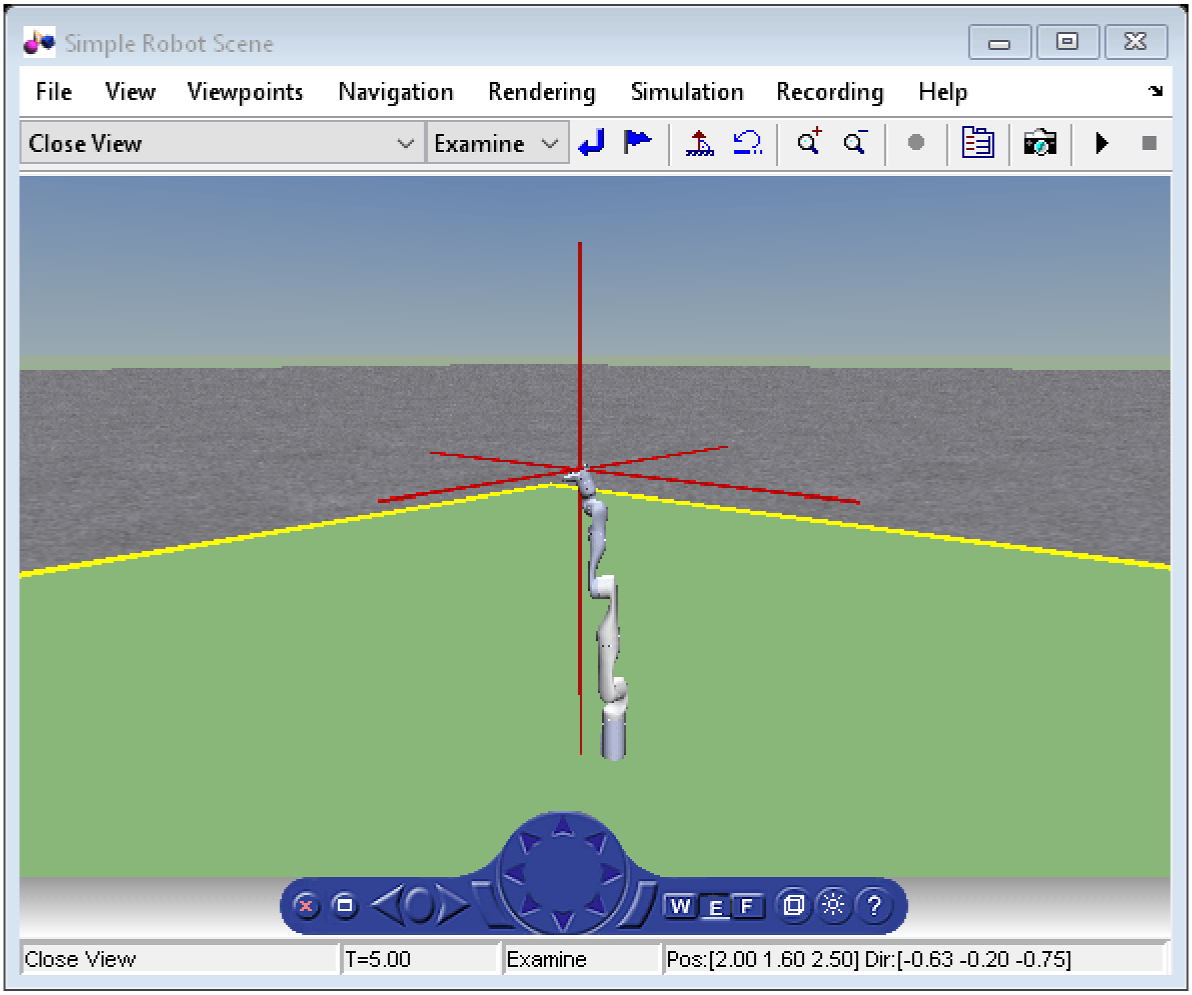
Visualization in Simulink 3D Animation™

The **VR RigidBodyTree** block inserts the manipulator into the scene defined by the associated world file, `robot_scene.wrl`. The **VR Sink** block provides a visualization for the world. In the block parameters, the **VR Sink** block has been modified to treat the setpoint, indicated by the red axes in the output, as an input. The **Get Transform** block is used to get the position of the end effector, which is then converted from a homogeneous transform matrix to a translation vector, and then from MATLAB to VR coordinates.



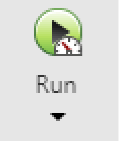
Simulate the Model

```
sim("SL3DJointSpaceManipulatorTrajectory.slx");
```



Close View | T=5.00 | Examine | Pos:[2.00 1.60 2.50] Dir:[-0.63 -0.20 -0.75]

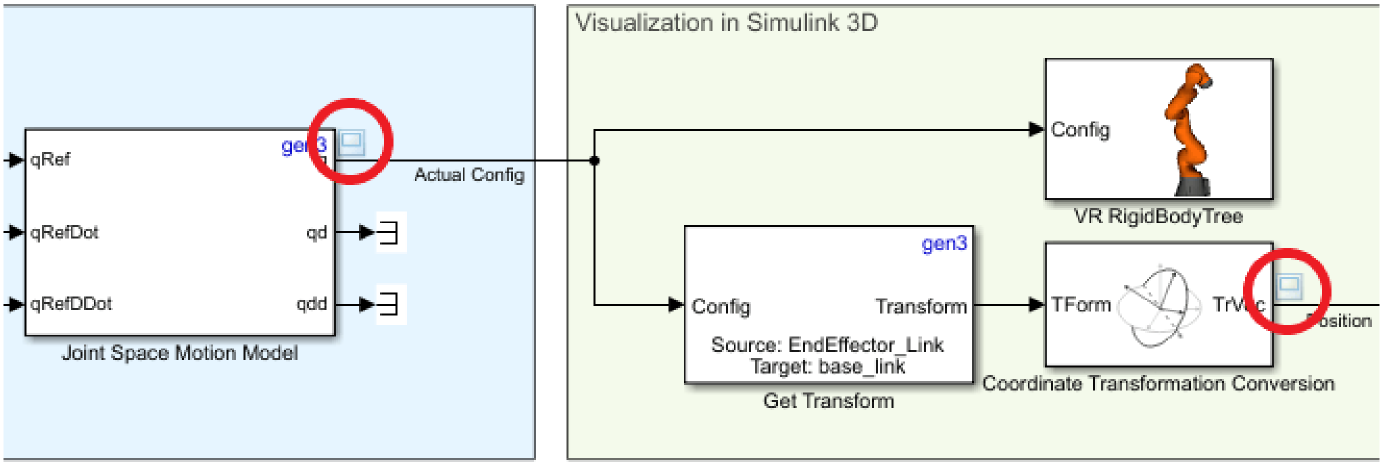
In the model, pacing is active, as indicated by the clock symbol below the run button:



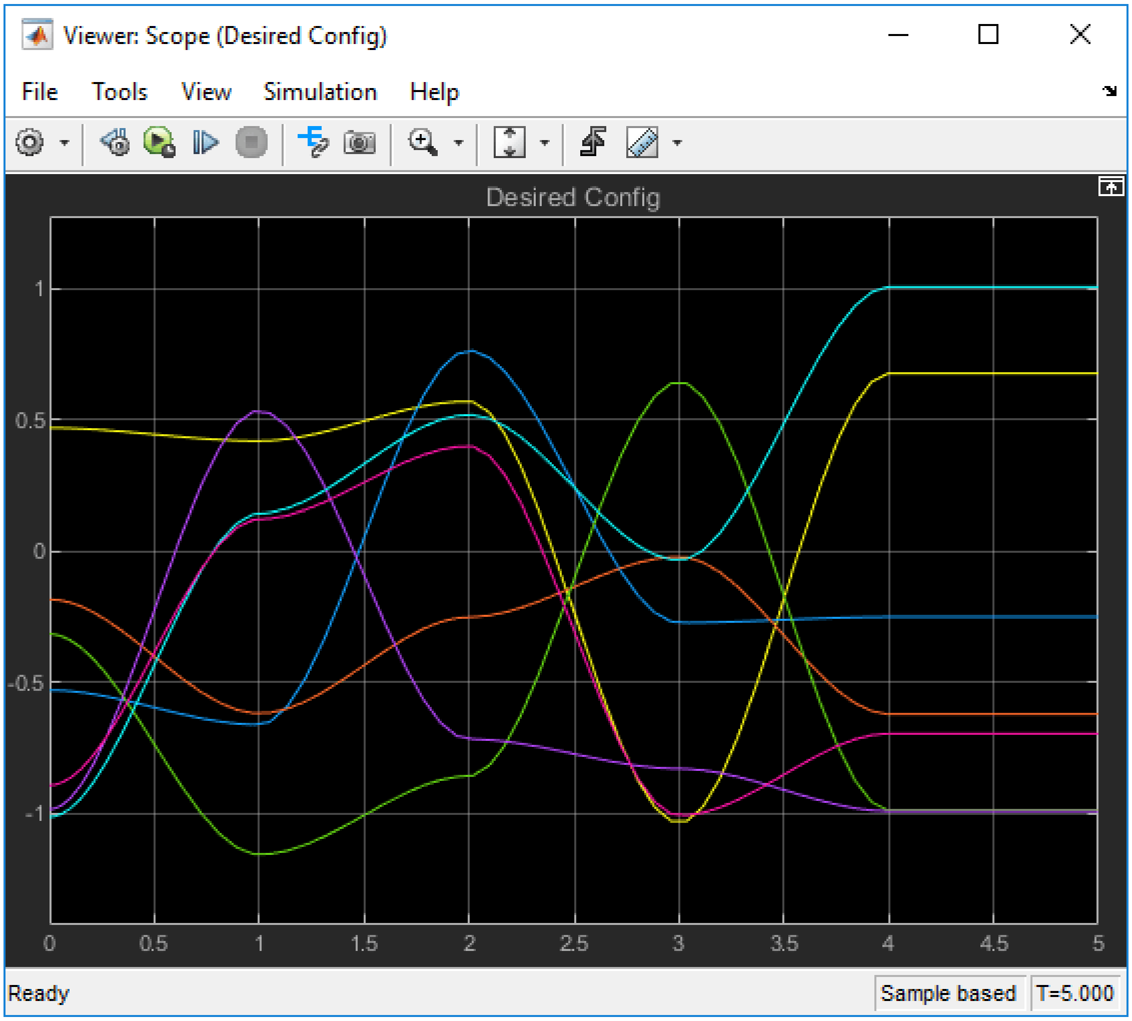
This ensures that the model is slowed down to near real-time speed, so that the visualization can be updated at a realistic pace.

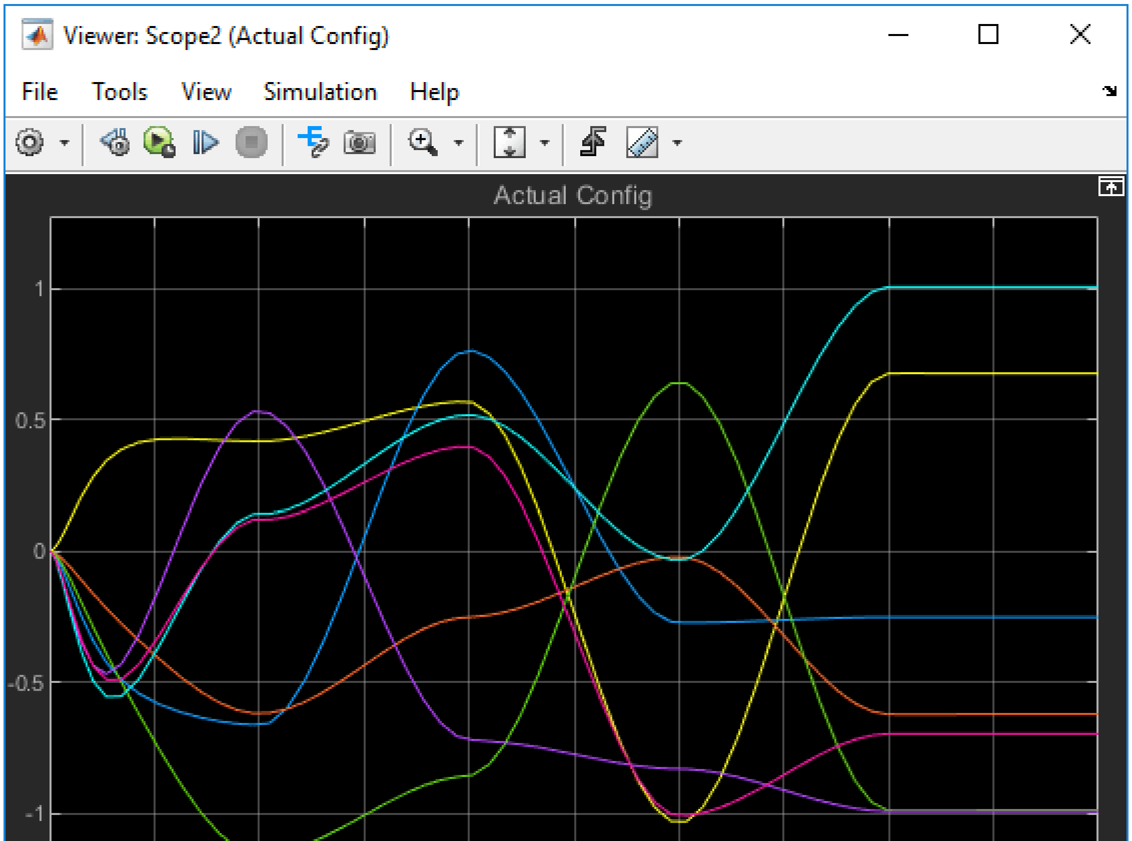
Trajectory Visualization

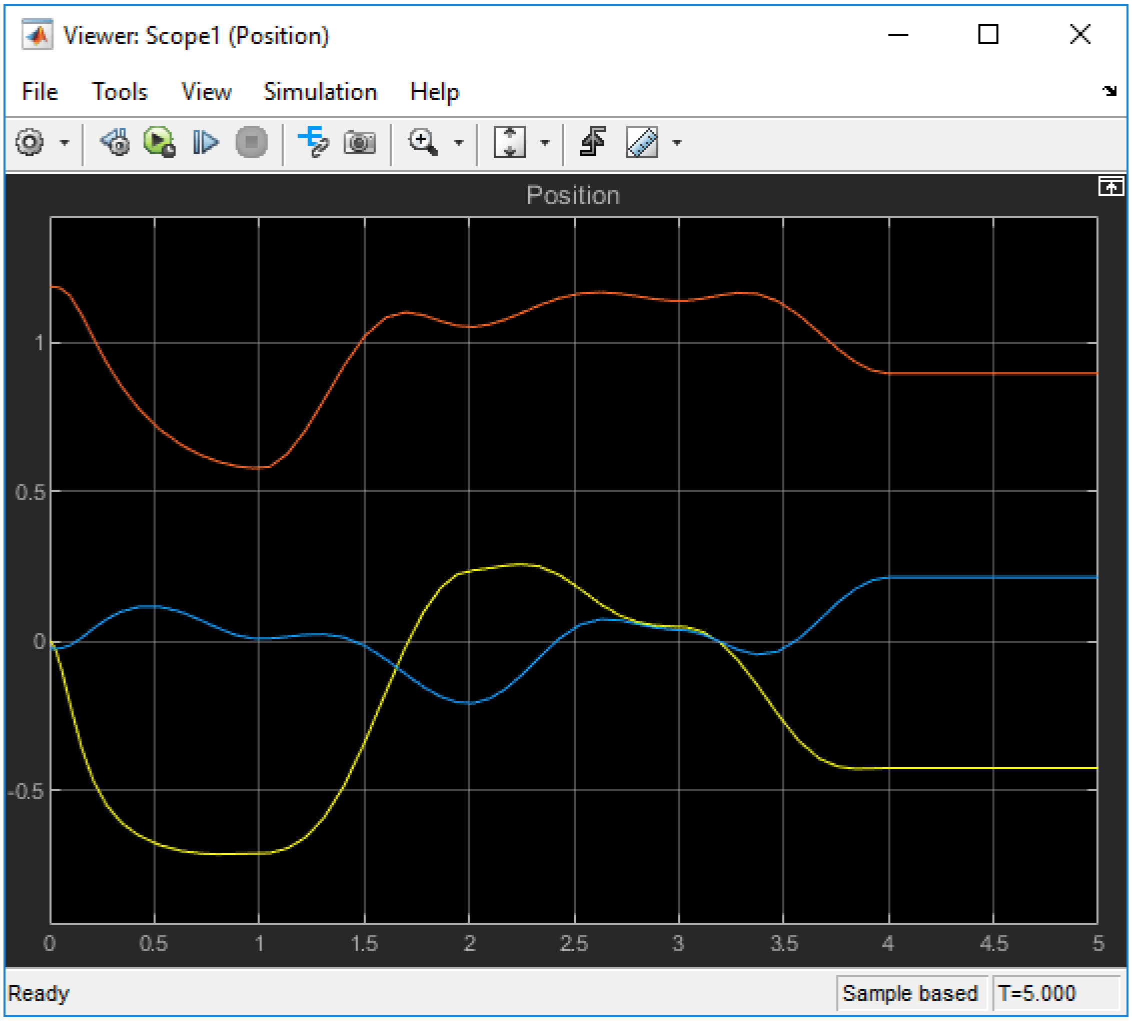
By default, the model opens both the VR visualization and the scopes that display velocity and position information. However, if they are closed, the VR view can be reopened by clicking on the **VR Sink** block, and the scopes can be opened by double-clicking the associated viewer icons:



The scopes show the tracking results of the **Joint Space Motion Model** block. As can be seen on the left in the figures below, the initial configuration of the robot differs from the reference trajectories, but the controlled motion ensures that the trajectory is reached and tracked for the duration of the simulation. The final scope displays the X, Y, and Z position of the end effector in the world frame.


Viewer: Scope (Desired Config)
File
Tools
View
Simulation
Help
Desired Config
1
0.5
0
-0.5
-1
0
0.5
1
1.5
2
2.5
3
3.5
4
4.5
5
Ready
Sample based
T=5.000


Viewer: Scope2 (Actual Config)
File
Tools
View
Simulation
Help
Actual Config
1
0.5
0
-0.5
-1


Viewer: Scope1 (Position)
File
Tools
View
Simulation
Help
Position
1
0.5
0
-0.5
0
0.5
1
1.5
2
2.5
3
3.5
4
4.5
5
Ready
Sample based
T=5.000

Compute Joint Torques To Balance An Endpoint Force and Moment

Generate torques to balance an endpoint force acting on the end-effector body of a planar robot. To calculate the joint torques using various methods, use the `geometricJacobian` and `inverseDynamics` object functions for a `rigidBodyTree` robot model.

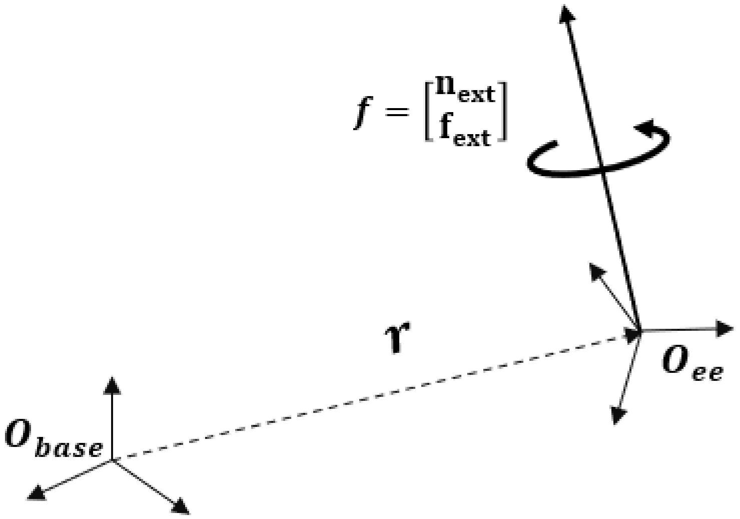
Initialize Robot

The `twoJointRigidBodyTree` robot is a 2-D planar robot. Joint configurations are output as column vectors.

```
twoJointRobot = twoJointRigidBodyTree("column");
```

Problem Setup

The endpoint force `eeForce` is a column vector with a combination of the linear force and moment acting on the end-effector body (`"tool"`). Note that this vector is expressed in the base coordinate frame and is shown below.


$f = \begin{bmatrix} \mathbf{n}_{\mathbf{ext}} \\ \mathbf{f}_{\mathbf{ext}} \end{bmatrix}$
r
O_{ee}
O_{base}

```
fx = 2;
fy = 2;
fz = 0;
```

```
nx = 0;
ny = 0;
nz = 3;
eeForce = [nx;ny;nz;fx;fy;fz];
eeName = "tool";
```

Specify the joint configuration of the robot for the balancing torques.

```
q = [pi/3;pi/4];
Tee = getTransform(twoJointRobot,q,eeName);
```

Geometric Jacobian Method

Using the principle of virtual work [1], find the balancing torque using the `geometricJacobian` object function and multiplying the transpose of the Jacobian by the endpoint force vector.

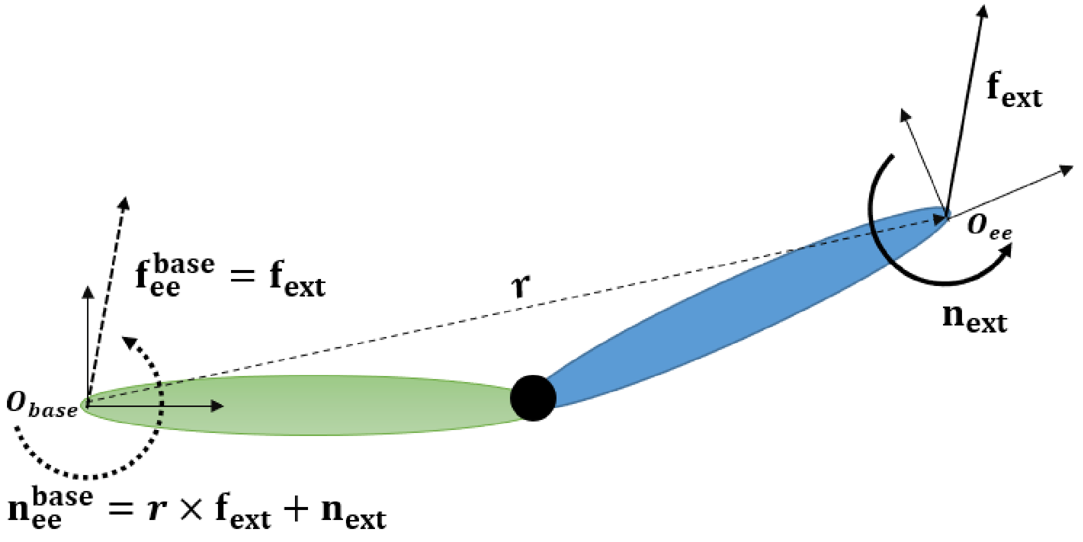
```
J = geometricJacobian(twoJointRobot,q,eeName);
jointTorques = J' * eeForce;
fprintf("Joint torques using geometric Jacobian (Nm): [%.3g, %.3g]",jointTorques);
```

```
Joint torques using geometric Jacobian (Nm): [1.41, 1.78]
```

Inverse Dynamics for Spatially-Transformed Force

Using another method, calculate the balancing torque by computing the inverse dynamics with the endpoint force spatially transformed to the base frame.

Spatially transforming a wrench from the end-effector frame to the base frame means to exert a new wrench in a frame that happens to collocate with the base frame in space, but is still fixed to the end-effector body; this new wrench has the same effect as the original wrench exerted at the **ee** origin. In the figure below, $\mathbf{f_{ext}}$ and $\mathbf{n_{ext}}$ are the endpoint linear force and moment respectively, and the $\mathbf{f_{ee}^{base}}$ and $\mathbf{n_{ee}^{base}}$ are the spatially transformed forces and moments, respectively. In the snippet below, `fbase_ee` is the spatially transformed wrench.


$\mathbf{f}_{\mathrm{ext}}$
O_{ee}
$\mathbf{n}_{\mathrm{ext}}$
$\mathbf{f}_{\mathrm{ee}}^{\mathrm{base}} = \mathbf{f}_{\mathrm{ext}}$
$\boldsymbol{r}$
O_{base}
$\mathbf{n}_{\mathrm{ee}}^{\mathrm{base}} = \boldsymbol{r} \times \mathbf{f}_{\mathrm{ext}} + \mathbf{n}_{\mathrm{ext}}$

$\mathbf{f}_{\mathrm{ext}}$ and $\mathbf{n}_{\mathrm{ext}}$ are defined in base frame's (O_{base}) coordinates

```
r = tform2trvec(Tee);
fbase_ee = [cross(r,[fx fy fz])' + [nx;ny;nz]; fx;fy;fz];
fext = -externalForce(twoJointRobot, eeName, fbase_ee);
jointTorques2 = inverseDynamics(twoJointRobot, q, [], [], fext);
fprintf("Joint torques using inverse dynamics (Nm): [%.3g, %.3g]",jointTorques2)
```

```
Joint torques using inverse dynamics (Nm): [1.41, 1.78]
```

Inverse Dynamics for End-Effector Force

Instead of spatially transforming the endpoint force to the base frame, use a third method by expressing the end-effector force in its own coordinate frame (`fee_ee`). Transform the moment and the linear force vectors into the end-effector coordinate frame. Then, specify that force and the current configuration to the `externalForce` function. Calculate the inverse dynamics from this force vector.

```
eeLinearForce = Tee \ [fx;fy;fz;0];
eeMoment = Tee \ [nx;ny;nz;0];
fee_ee = [eeMoment(1:3); eeLinearForce(1:3)];
fext = -externalForce(twoJointRobot,eeName,fee_ee,q);
jointTorques3 = inverseDynamics(twoJointRobot, q, [], [], fext);
fprintf("Joint torques using inverse dynamics (Nm): [%.3g, %.3g]",jointTorques3);
```

```
Joint torques using inverse dynamics (Nm): [1.41, 1.78]
```

References

[1]Siciliano, B., Sciavicco, L., Villani, L., & Oriolo, G. (2009). Differential kinematics and statics. Robotics: Modelling, Planning and Control, 105-160.

[2]Harry Asada, and John Leonard. 2.12 Introduction to Robotics. Fall 2005. Chapter 6 Massachusetts Institute of Technology: MIT OpenCourseWare, https://ocw.mit.edu. License: Creative Commons BY-NC-SA.

See Also

Objects
`rigidBodyTree`

Functions
`geometricJacobian` | `inverseDynamics` | `externalForce`

Simulate Joint-Space Trajectory Tracking in MATLAB

This example shows how to simulate the joint-space motion of a robotic manipulator under closed-loop control.

Define Robot and Initial State

Load an ABB IRB-120T from the robot library using the `loadrobot` function.

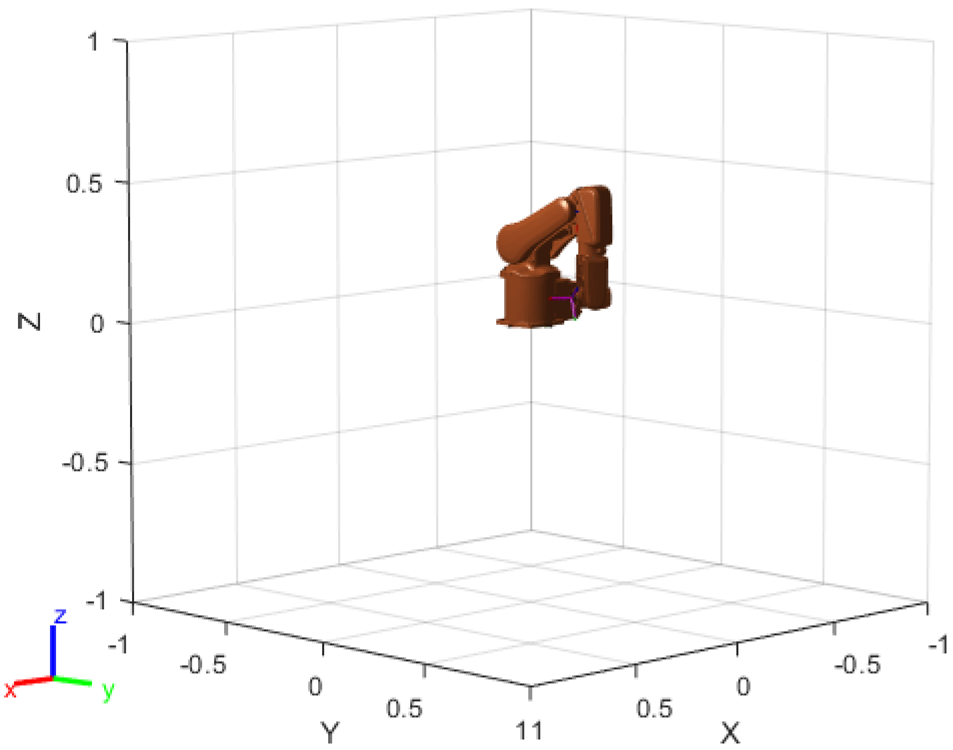
```
robot = loadrobot("abbIrb120T","DataFormat","column","Gravity",[0 0 -9.81]);
numJoints = numel(homeConfiguration(robot));
```

Define simulation parameters, including the time range over which the trajectory is simulated, the initial state as `[joint configuration; jointVelocity]`, and the joint-space set point.

```
% Set up simulation parameters
tSpan = 0:0.01:0.5;
q0 = zeros(numJoints,1);
q0(2) = pi/4; % Something off center
qd0 = zeros(numJoints,1);
initialState = [q0; qd0];

% Set up joint control targets
targetJointPosition = [pi/2 pi/3 pi/6 2*pi/3 -pi/2 -pi/3]';
targetJointVelocity = zeros(numJoints,1);
targetJointAcceleration = zeros(numJoints,1);
```

Visualize the goal position.

```
show(robot,targetJointPosition)
```



```
ans = 
  Axes (Primary) with properties:

             XLim: [-1 1]
             YLim: [-1 1]
           XScale: 'linear'
           YScale: 'linear'
    GridLineStyle: '-'
         Position: [0.1300 0.1100 0.7750 0.8150]
            Units: 'normalized'

  Show all properties
```

Model Behavior with Joint-Space Control

Using a `jointSpaceMotionModel` object, simulate the closed-loop motion of the model under a variety of controllers. This example compares a few of them. Each instance uses the `derivative` function to compute the state derivative. Here, the state is 2*n*-element vector `[joint configuration; joint velocity]`, where *n* is the number of joints in the associated `rigidBodyTree` object.

Computed-Torque Control

Computed-torque control uses an inverse-dynamics computation to compensate for the robot dynamics. The controller drives the closed-loop error dynamics of each joint based on a second-order response.

Create a `jointSpaceMotionModel` and specify the robot model. Set the `"MotionType"` to `"ComputedTorqueControl"`. Update the error dynamics using `updateErrorDynamicsFromStep` and specify the desired settling time and overshoot respectively. Alternatively, you can set the damping ratio and natural frequency directly in the object.

```
computedTorqueMotion = jointSpaceMotionModel("RigidBodyTree",robot,"MotionType","ComputedTorqueCo
updateErrorDynamicsFromStep(computedTorqueMotion,0.2,0.1);
```

This motion model requires position, velocity, and acceleration to be provided.

```
qDesComputedTorque = [targetJointPosition; targetJointVelocity; targetJointAcceleration];
```

To view an example of this controller in practice in Simulink, see the “Perform Safe Trajectory Tracking Control Using Robotics Manipulator Blocks” on page 1-268 example.

Independent Joint Control

With independent joint control, model each joint as a separate system that has a second-order tracking response. This type of model is an idealized behavior, and is best used when the response is slow, or when the dynamics will not have a significant impact on the resultant trajectory. In those cases, it will behave the same as computed-torque control, but with less computational overhead.

Create another `joinSpaceMotionModel` using the `"IndependentJointMotion"` motion type.

```
IndepJointMotion = jointSpaceMotionModel("RigidBodyTree",robot,"MotionType","IndependentJointMot
updateErrorDynamicsFromStep(IndepJointMotion,0.2,0.1);
```

This motion model requires position, velocity, and acceleration to be provided.

```
qDesIndepJoint = [targetJointPosition; targetJointVelocity; targetJointAcceleration];
```

Proportional-Derivative Control

Proportional-Derivative Control, or PD Control, combines gravity compensation with proportional and derivative gains. Despite the simpler nature relative to other closed-form models, the PD Controller can be stable for all positive gain values, which makes it a desirable option. Here, the PD Gains are set as *n*-by-*n* matrices, where *n* is the number of joints in the associated `rigidBodyTree` object. For this robot, *n* = 6. Additionally, PD Control does not require an acceleration profile, so its state vector is just a 2*n*-element vector of joint configurations and joint velocities.

```
pdMotion = jointSpaceMotionModel("RigidBodyTree",robot,"MotionType","PDControl");
pdMotion.Kp = diag(300*ones(1,6));
pdMotion.Kd = diag(10*ones(1,6));
```

This motion model requires position and velocity to be provided.

```
qDesPD = [targetJointPosition; targetJointVelocity];
```

Simulate using an ODE Solver

The `derivative` function outputs the state derivative, which can be integrated using an ordinary differential equation (ODE) solver such as `ode45`. For each motion model, the ODE solver outputs a *m*-element column vector that covers `tspan` and a 2-by-*m* matrix of the 2*n*-element state vector at each instant in time.

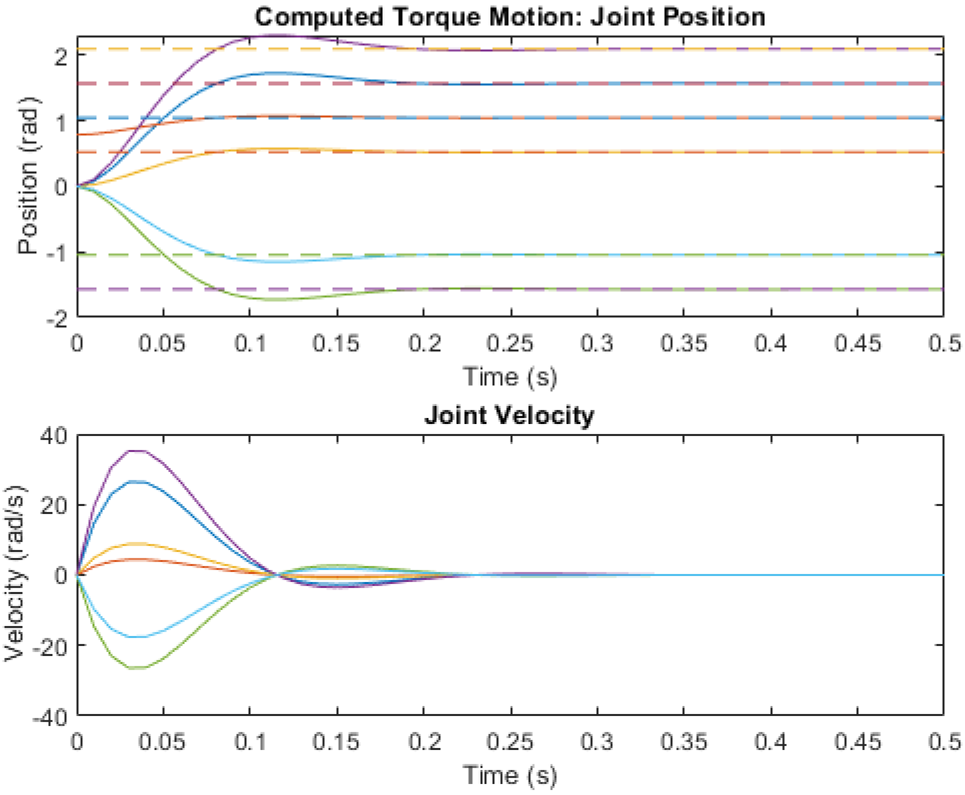
Calculate the trajectory for each motion model, using the most appropriate ODE solver for each system.

```
[tComputedTorque,yComputedTorque] = ode45(@(t,y)derivative(computedTorqueMotion,y,qDesComputedTor
[tIndepJoint,yIndepJoint] = ode45(@(t,y)derivative(IndepJointMotion,y,qDesIndepJoint),tSpan,initi
[tPD,yPD] = ode15s(@(t,y)derivative(pdMotion,y,qDesPD),tSpan,initialState);
```

Plot Results

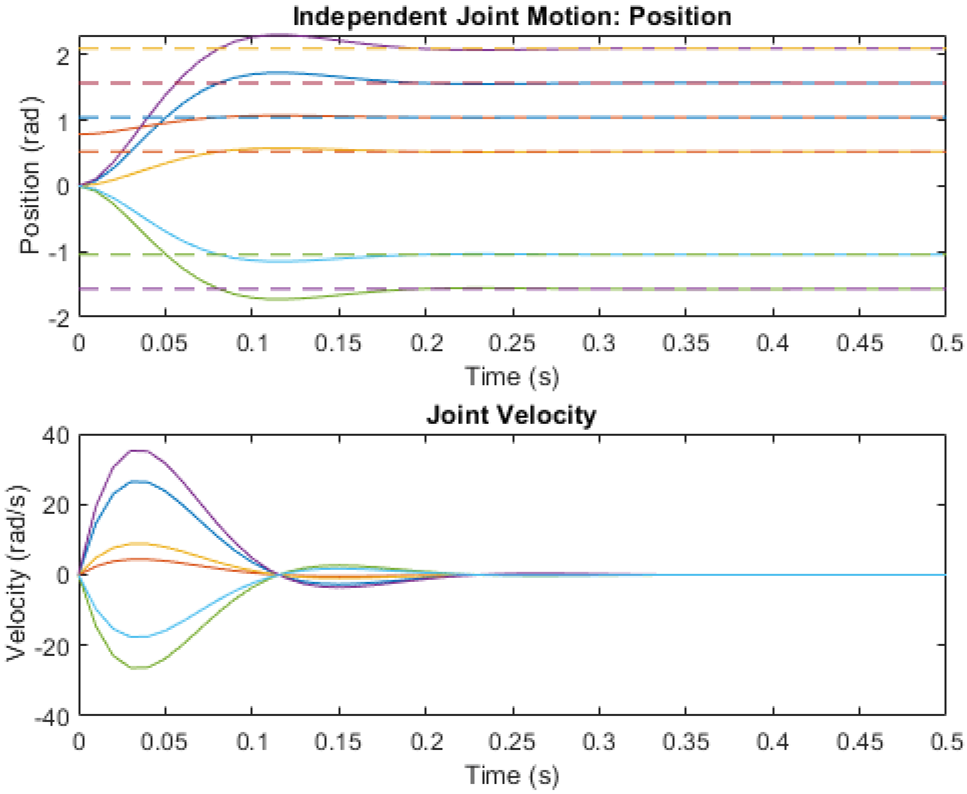
Once the simulation is complete, compare the results side-by-side. Each plot shows the joint position on the top, and velocity on the bottom. The dashed lines indicate the reference trajectories, while the solid lines display the simulated response.

```
% Computed Torque Control
figure
subplot(2,1,1)
plot(tComputedTorque,yComputedTorque(:,1:numJoints)) % Joint position
hold all
plot(tComputedTorque,targetJointPosition*ones(1,length(tComputedTorque)),'--') % Joint setpoint
title('Computed Torque Motion: Joint Position')
xlabel('Time (s)')
ylabel('Position (rad)')
subplot(2,1,2)
plot(tComputedTorque,yComputedTorque(:,numJoints+1:end)) % Joint velocity
title('Joint Velocity')
xlabel('Time (s)')
ylabel('Velocity (rad/s)')
```



In the following plot, use independent joint control to confirm that the computed torque motion behaves equivalently under some simplifying assumptions.

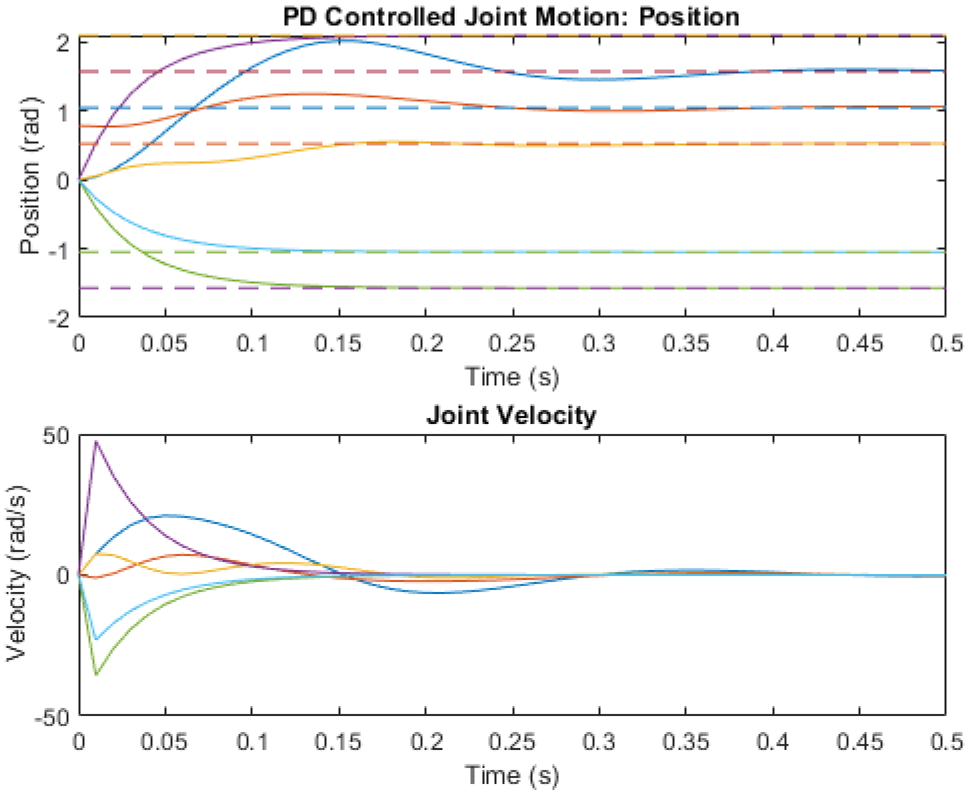
```
% Independent Joint Motion
figure
subplot(2,1,1)
plot(tIndepJoint,yIndepJoint(:,1:numJoints))
hold all
plot(tIndepJoint,targetJointPosition*ones(1,length(tIndepJoint)),'--')
title('Independent Joint Motion: Position')
xlabel('Time (s)')
ylabel('Position (rad)')
subplot(2,1,2);
plot(tIndepJoint,yIndepJoint(:,numJoints+1:end))
title('Joint Velocity')
xlabel('Time (s)')
ylabel('Velocity (rad/s)')
```



Finally, the PD Controller uses fairly aggressive gains to achieve similar rise times, but unlike the other approaches, the individual joints behave differently, since each joint and the associated bodies have slightly different dynamic properties that are not compensated by the controller.

```
% PD with Gravity Compensation
figure
subplot(2,1,1)
plot(tPD,yPD(:,1:numJoints))
hold all
plot(tPD,targetJointPosition*ones(1,length(tPD)),'--')
title('PD Controlled Joint Motion: Position')
xlabel('Time (s)')
```

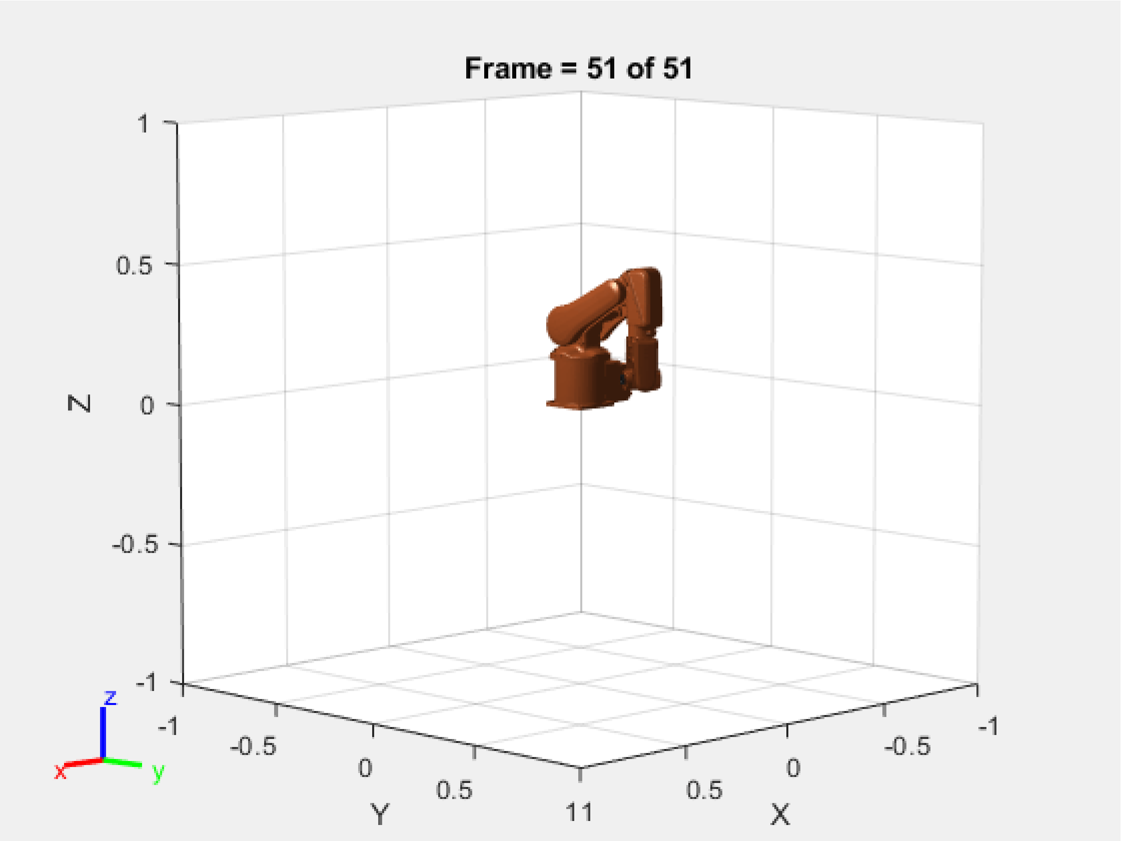
```
ylabel('Position (rad)')
subplot(2,1,2)
plot(tPD,yPD(:,numJoints+1:end))
title('Joint Velocity')
xlabel('Time (s)')
ylabel('Velocity (rad/s)')
```



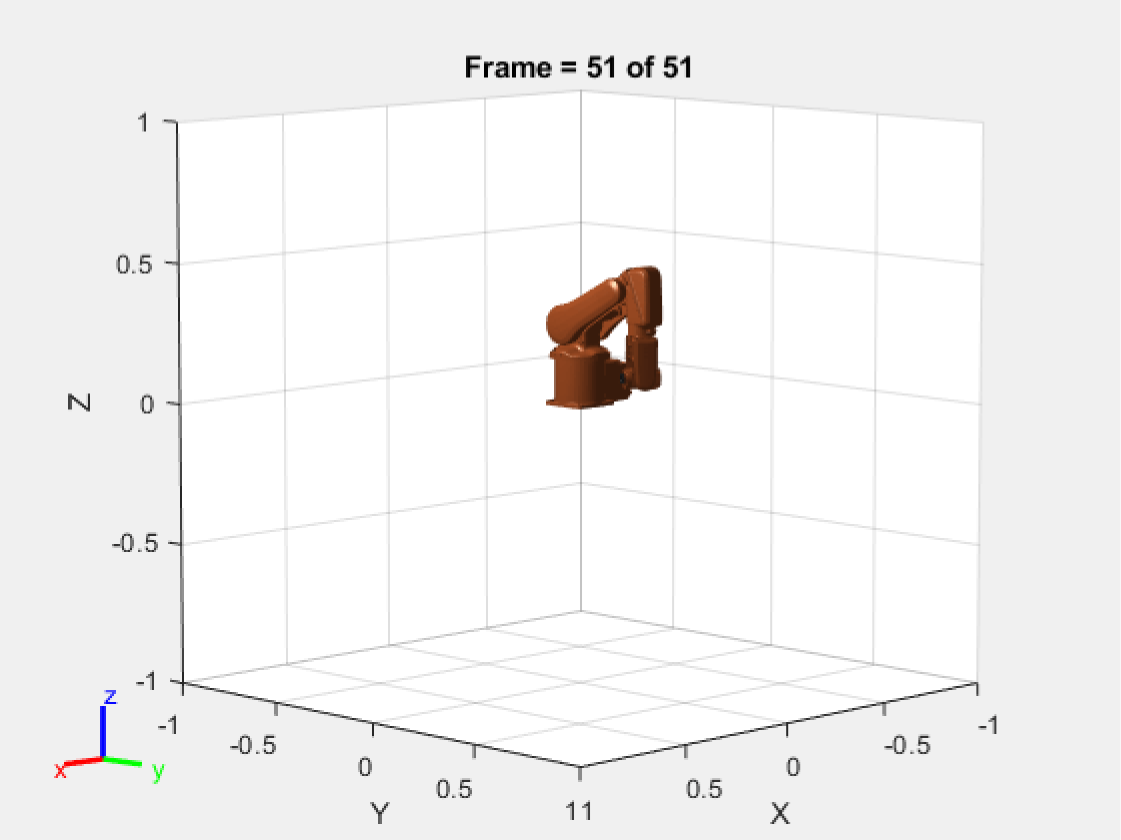
Visualize the Trajectories as an Animation

To see what this behavior looks like in 3-D, the following example helper plots the robot motion in time. The third input is the number of frames between each sample.

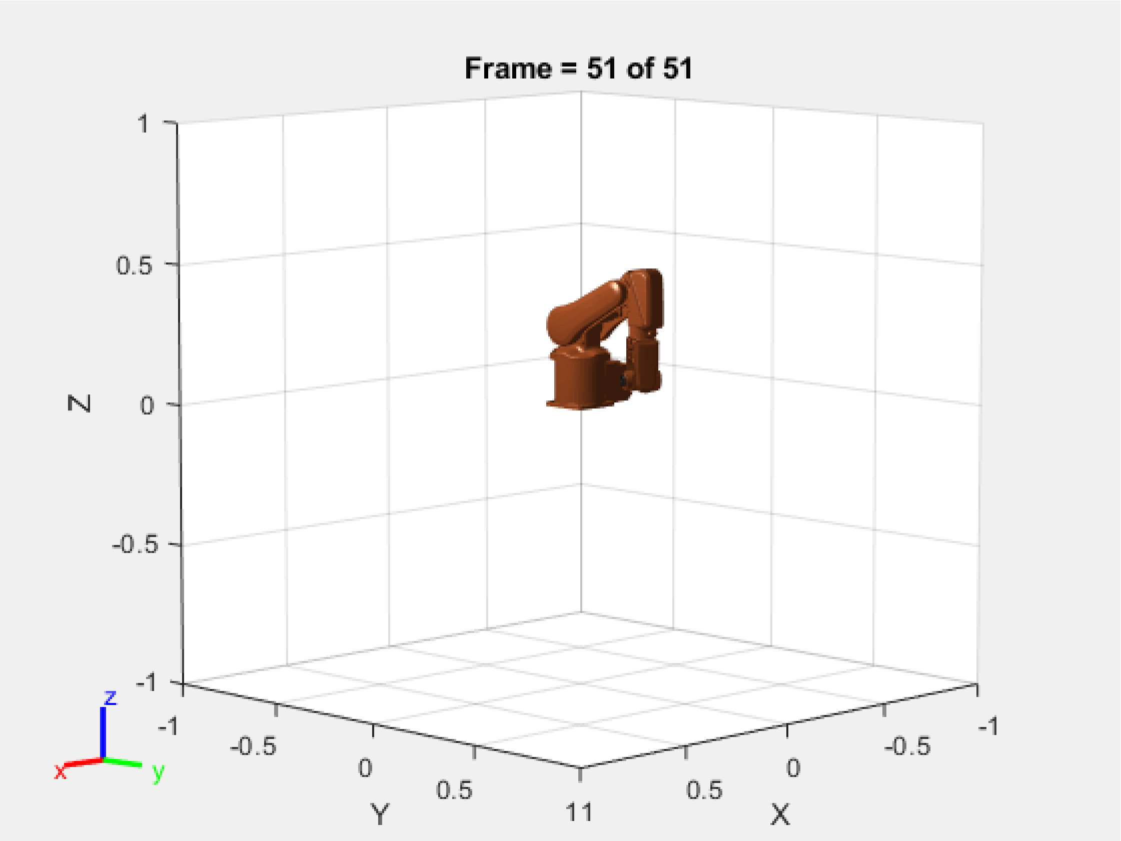
```
exampleHelperRigidBodyTreeAnimation(robot,yComputedTorque,1);
```



```
exampleHelperRigidBodyTreeAnimation(robot,yIndepJoint,1);
```



```
exampleHelperRigidBodyTreeAnimation(robot,yPD,1);
```


Frame = 51 of 51
1
0.5
Z
0
-0.5
-1
-1
-0.5
0
0.5
1
Y
1
0.5
0
-0.5
-1
X
z
x
y

Model and Control a Manipulator Arm with Robotics and Simscape

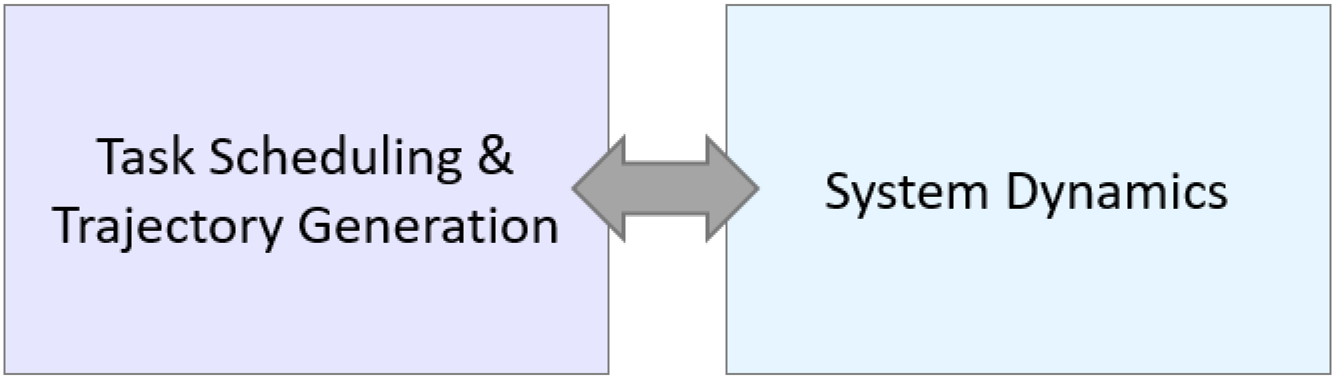
Execute a pick-and-place workflow using an ABB YuMi robot, which demonstrates how to design robot algorithms in Simulink®, and then simulate the action in a test environment using Simscape™. The example also shows how to model a system with different levels of fidelity to focus on better focus on the associated algorithm design.

The design elements of this example are split into three sections to more easily focus on different aspects of the model design:

1. **Create a Task & Trajectory Scheduler for Pick-And-Place using Simplified Manipulator System Dynamics:**
2. **Add Core Manipulator Dynamics and Design a Controller**
3. **Verify Complete Workflow on Simscape Model of the Robot and Environment**

High Level Goals

In the "Interactively Build a Trajectory for an ABB YuMi Robot" on page 1-156 example, a robot waypoint sequence was designed and replayed using a continuous trajectory. In this example, Simulink models convert these waypoints to a complete and repeatable pick-and-place workflow. The model has the two key elements:



The task scheduling and trajectory generation portion defines how the robot traverses through the states. This includes the robot configuration state at any instant, what the goal position is, whether the gripper should be open or closed, and the current trajectory being sent to the robot.

The system dynamics portion models the robot behavior. This defines how the robot moves given a set of reference trajectories and a boolean gripper command (open or closed). The system dynamics can be modeled with different levels of fidelity, depending on the aim of the overall model.

For this example, during the task scheduler design, the aim is to ensure the scheduler behaves correctly under the assumption that the robot is under stable motion control. For this portion, a straightforward model that simulates quickly desirable, so the system dynamics are modeld using the **Joint Space Motion Model** block. This block simulates the manipulator motion given joint-space reference trajectories under a stable controller with predefined response parameters. Once the task scheduling is complete, the focus of the model is shifted to controller design and system verification, which requires more complex system dynamics models.

Define a Robot and Environment

Load an ABB YuMi robot model. The robot is an industrial manipulator with two arms. This example only uses a single arm.

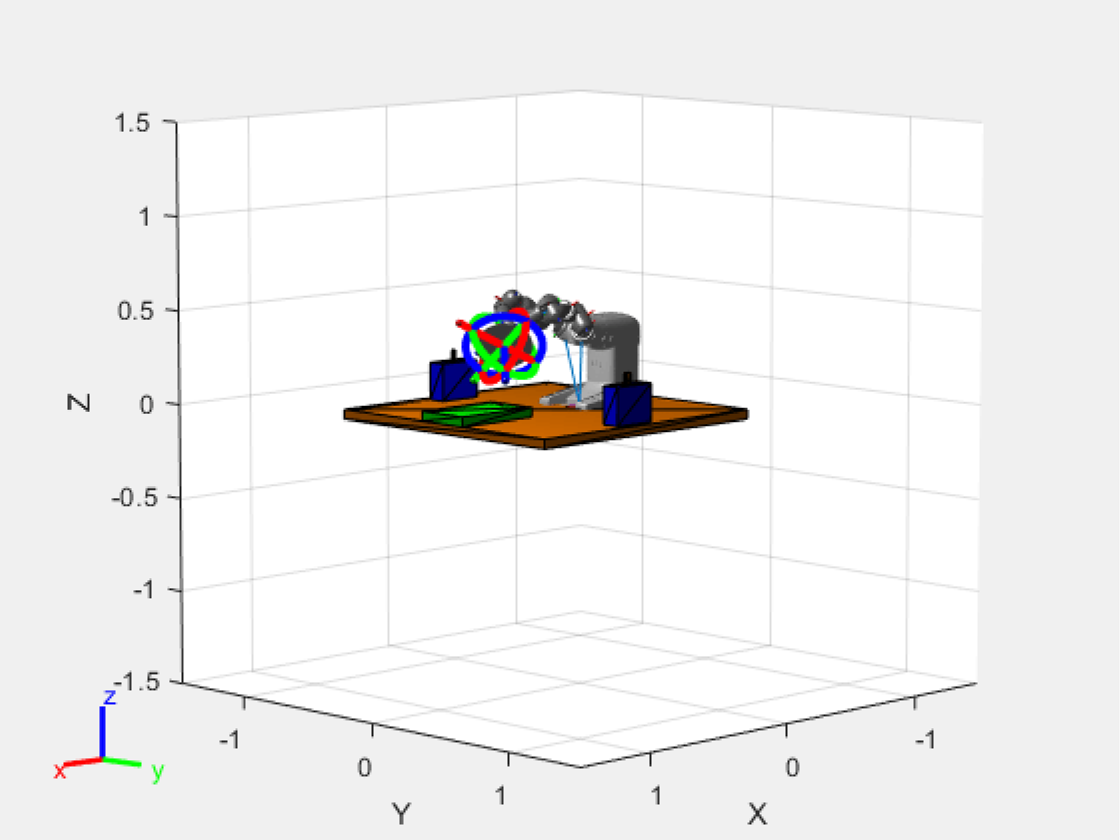
```
robot = loadrobot('abbYumi','Gravity',[0 0 -9.81]);
```

Create a visualization to replay simulated trajectories.

```
iviz = interactiveRigidBodyTree(robot);
ax = gca;
```

Add an environment by creating a set of collision objects using an example helper function.

```
exampleHelperSetupWorkspace(ax);
```



Initialize Shared Simulation Parameters

This example uses a set of predefined configurations, `configSequence`, as robot states. These are stored in an associated MAT file and were initially defined in “Interactively Build a Trajectory for an ABB YuMi Robot” on page 1-156.

```
load abbSavedConfigs.mat configSequence
```

For the simulation, the initial state of the robot must be defined including postion, velocity, and acceleration of each joint.

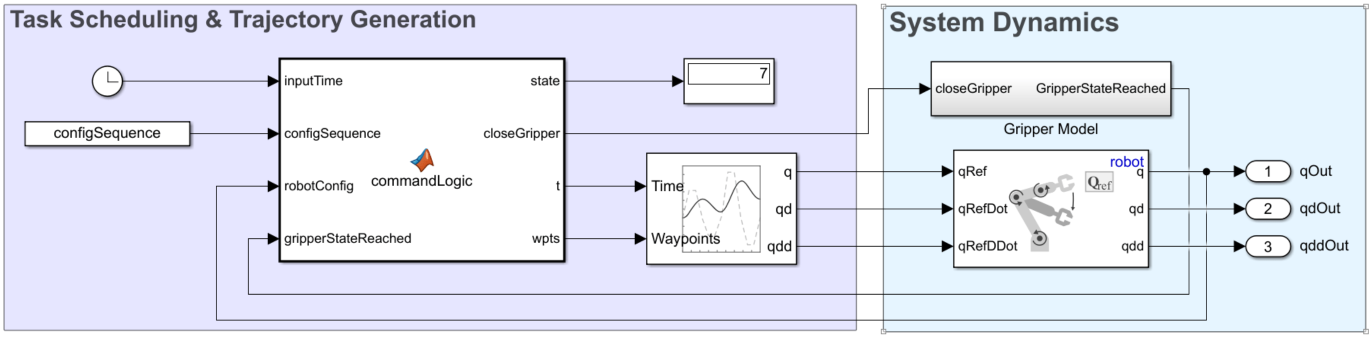
```
% Define initial state
q0 = configSequence(:,1); % Position
```

```
dq0 = zeros(size(q0)); % Velocity
ddq0 = zeros(size(q0)); % Acceleration
```

Create a Task & Trajectory Scheduler

Load the first model, which focuses on the task scheduling and trajectory generation section of the model.

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



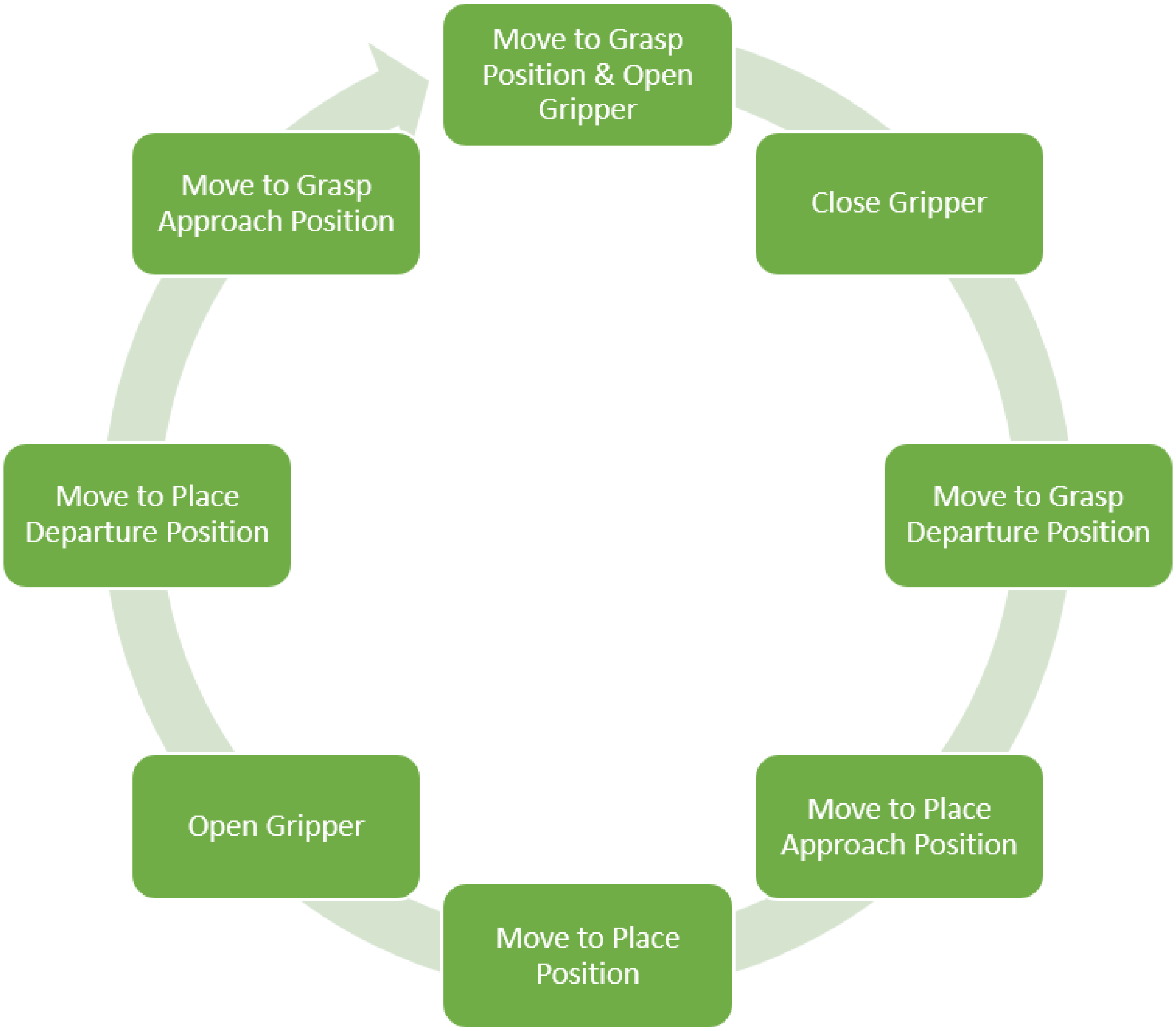
Simplified System Dynamics

To focus on the scheduling portion of the model, the system dynamics are modeled using the **Joint Space Motion Model** block. This motion model assumes the robot can reach specified configurations under stable, accurate control. Later, the example details more accurate modelling of the system dynamics.

The gripper is modeled as a simple Boolean command input as `0` or `1` (open or closed), and an output that indicates whether the gripper achieved the commanded position. Typically, robots treat the gripper command separately from the other configuration inputs.

Task Scheduling

The series of tasks the robot through are eight states:


Move to Grasp Position & Open Gripper
Close Gripper
Move to Grasp Departure Position
Move to Place Approach Position
Move to Place Position
Open Gripper
Move to Place Departure Position
Move to Grasp Approach Position

The scheduler is implemented using a MATLAB Function block, `commandLogic`. The scheduler advances states when the gripper state is reached and all the manipulator joints have reached their target positions within a predefined threshold. Each task is input to the **Trapezoidal Velocity Profile Trajectory** block which generates a smooth trajectory between each waypoint.

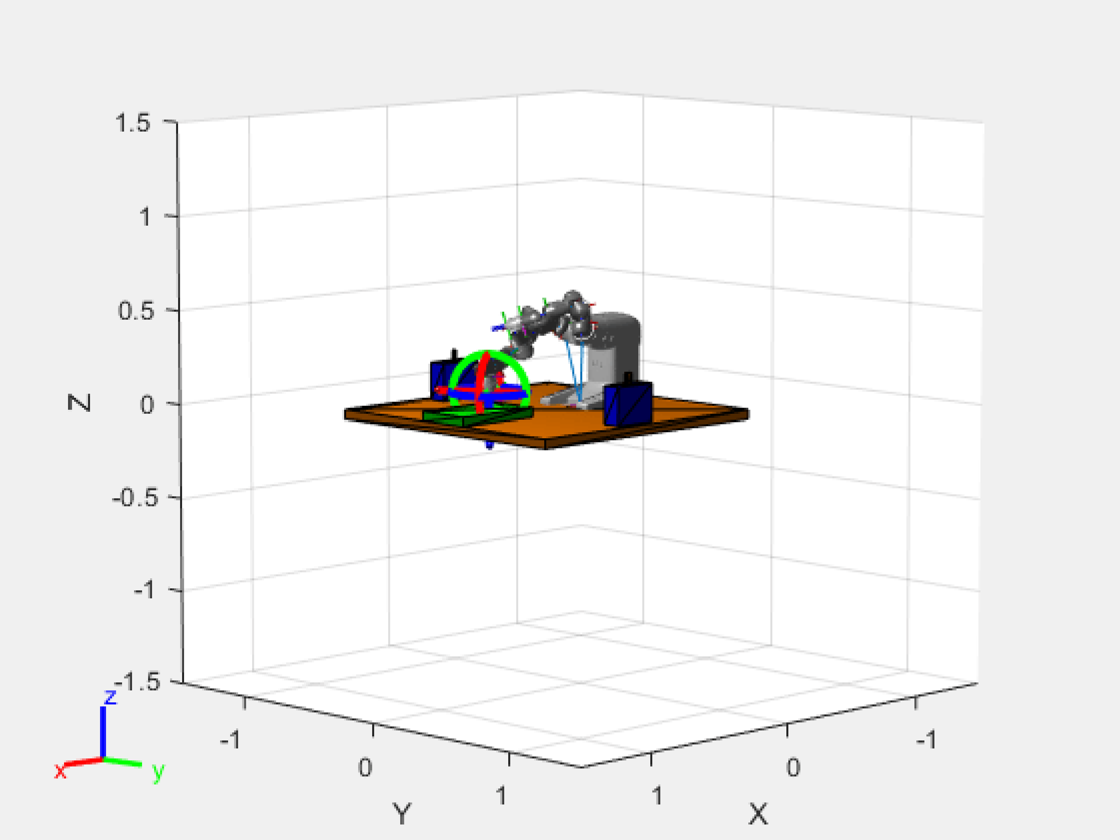
Simulate the Model

The provided Simulink model stores variables relevant to the example in the Model Workspace. Click **Load Default Parameters** to reinitialize the variables if needed. For more information, see “Model Workspaces” (Simulink).

Run the model by calling `sim`.

Use the interactive visualization to play back the motion. The model is simulated for a few extra seconds to ensure that the cycle loops as expected after the first motion. This model does not simulate any environment interaction, so the robot does not actually grab objects in this simulation.

```
simout = sim('modelWithSimplifiedSystemDynamics.slx');

% Visualize the motion using the interactiveRigidBodyTree object.
iviz.ShowMarker = false;
iviz.showFigure;
rateCtrlObj = rateControl(length(simout.tout)/(max(simout.tout)));
for i = 1:length(simout.tout)
    iviz.Configuration = simout.yout{1}.Values.Data(i,:);
    waitfor(rateCtrlObj);
end
```



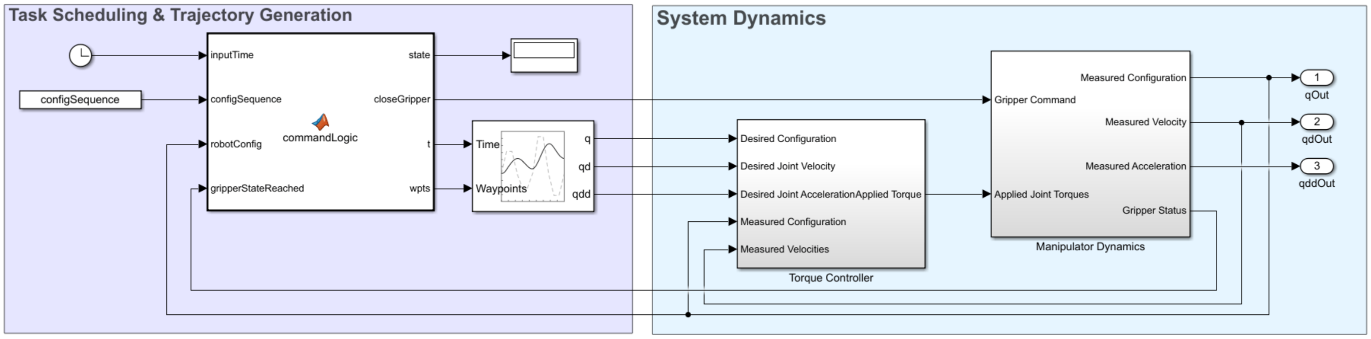
Add Core Manipulator Dynamics and Design a Controller

Now that the scheduler has been designed and verified, add a controller for the robot with two elements

- A more complex manipulator dynamics model that accepts joint torques and gripper commands
- A joint-space controller that returns joint torques given desired and current manipulator states

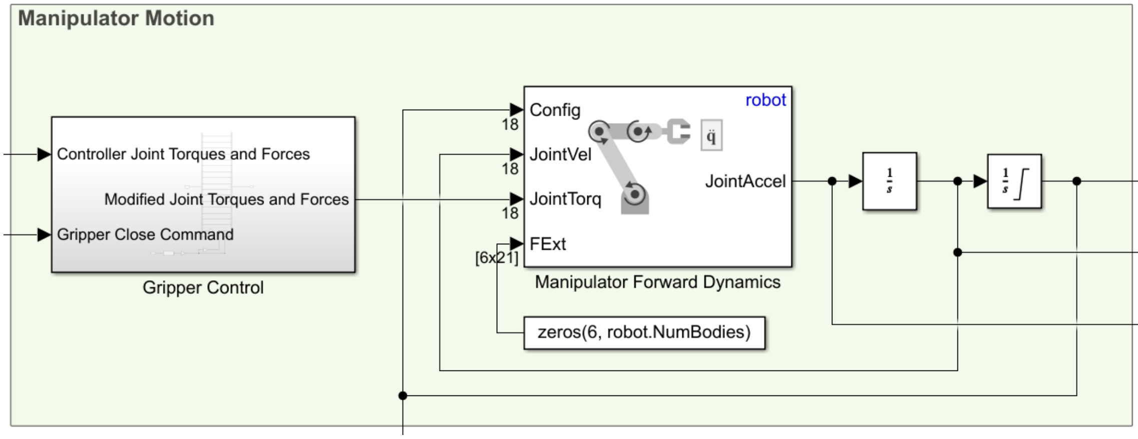
Open the next provided model with the added controller.

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



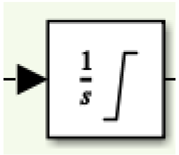
Manipulator Dynamics

For the purpose of designing a controller, the manipulator dynamics have to represent the manipulator joint positions given torque inputs. This is achieved inside the **Manipulator Dynamics** subsystem using a **Forward Dynamics** block to convert joint torques to joint acceleration given the current state, and then integrating twice to get the complete joint configuration. The integrators are initialized to `q0` and `dq0`, the initial joint position and velocity.



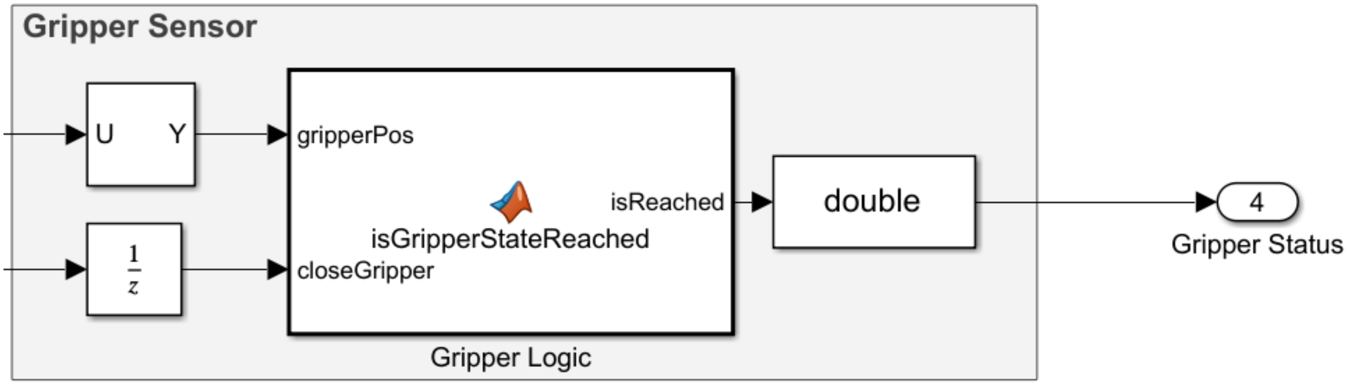
Additionally, the gripper control subsystem overrides the joint control torques to the gripper actuators with 10 N of force that is applied to close or open the gripper.

Note that the second integrator is saturated.



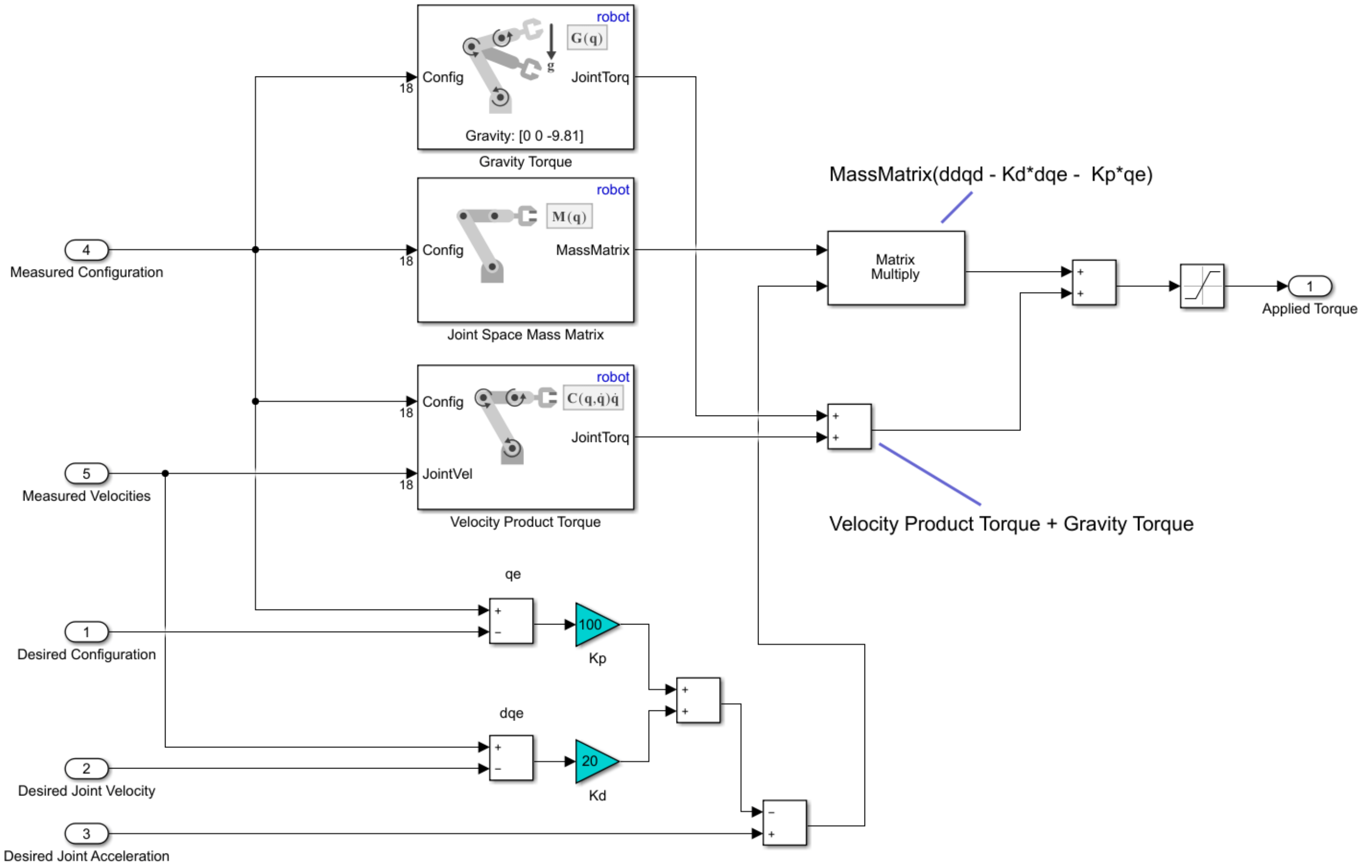
While manipulators under well-designed position controllers typically will not reach joint limits, the addition of open-loop forces from the gripper action mean that joint limits are required to ensure a realistic response. For a more accurate model, the joint saturation could be connected to the velocity to reset integration, but for this model, this level of accuracy is sufficient.

Gripper Sensor



This model also adds a more detailed gripper sensor to check when the gripper has actually been opened or closed. The gripper sensor extracts the last two values of the joint configuration (the values that correspond to the gripper position), and compares them to the intended gripper position, given by the `closeGripper` command, in a MATLAB Function block, **Gripper Logic**. The **Gripper Status** returns 1 when the positions of the gripper joints match the desired state given by the closeGripper command. When the gripper has not yet reached those states, **Gripper Status** returns zero. This matches the behavior of the **Gripper Model** in the earlier, simplified model.

Joint-Space Controller

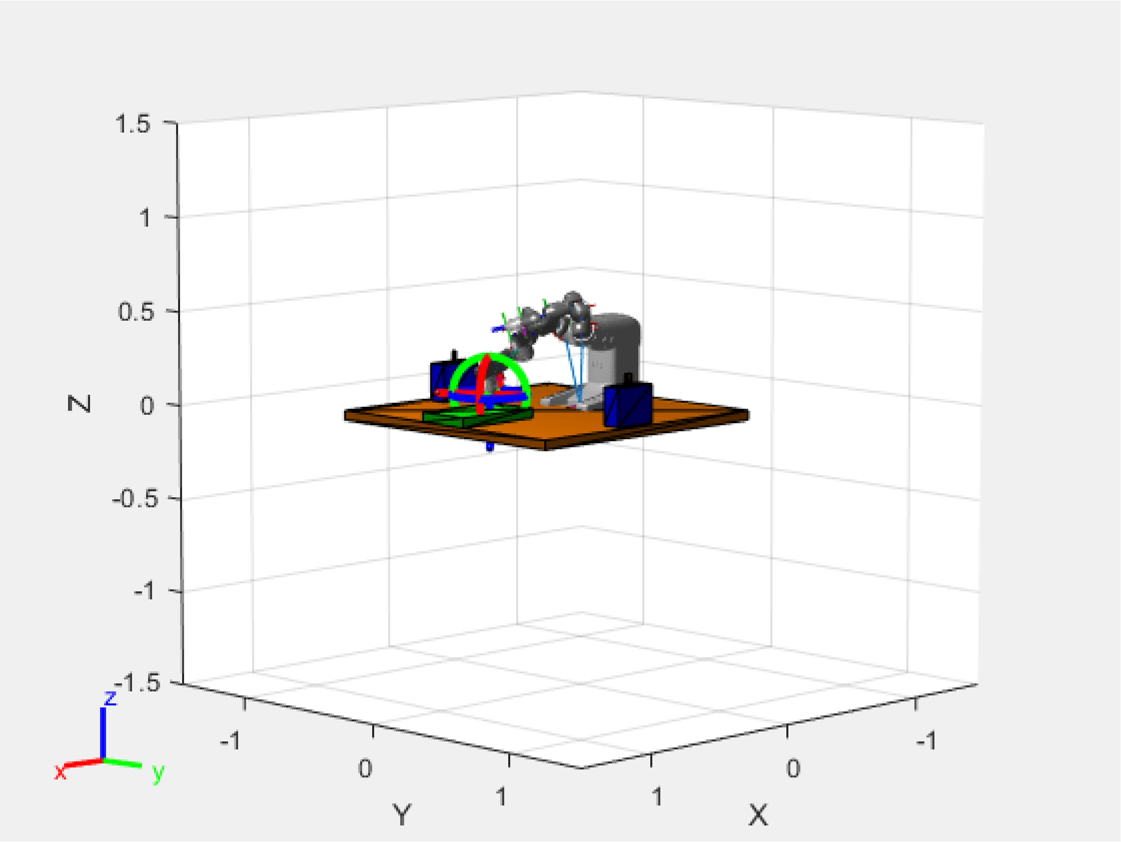


This model also adds a computed torque controller which implements a model-based approach to joint control. For more details, see Build Computed Torque Controller Using Robotics Manipulator Blocks in the “Perform Safe Trajectory Tracking Control Using Robotics Manipulator Blocks” on page 1-268 example. This model uses the same controller, but with the ABB YuMi as the `rigidBodyTree` input rather than the Rethink Sawyer.

Simulate the Model

Simulate and visualize the results using the new model.

```
simout = sim('modelWithControllerAndBasicRobotDynamics.slx');

% Visualize the motion using the interactiveRigidBodyTree
iviz.ShowMarker = false;
iviz.showFigure;
rateCtrlObj = rateControl(length(simout.tout)/(max(simout.tout)));
for i = 1:length(simout.tout)
    iviz.Configuration = simout.yout{1}.Values.Data(i,:);
    waitfor(rateCtrlObj);
end
```



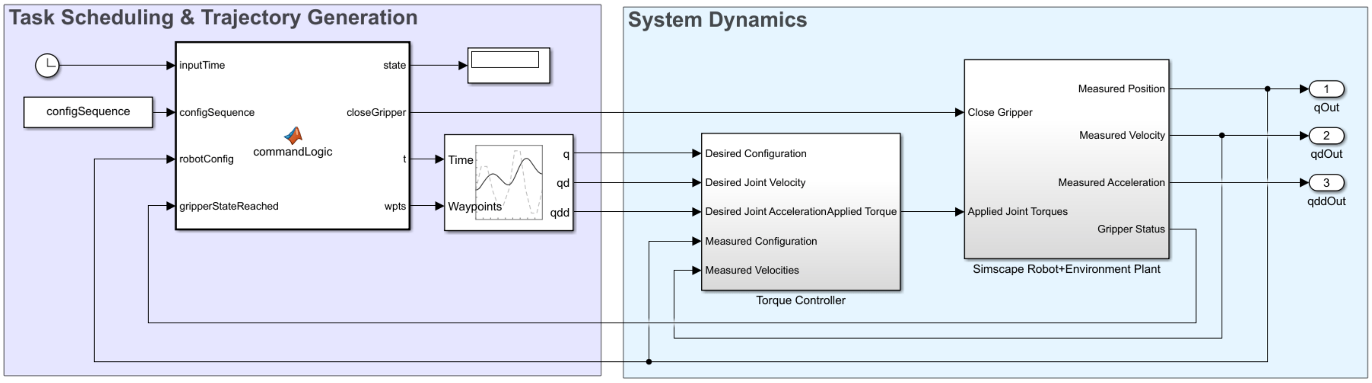
Verify Complete Workflow on Simscape Model of the Robot and Environment

Now that the task scheduler and controller have been designed, add more complex robot and environment models. Use Simscape Multibody™ which can create high-fidelity models of physical systems. In this application, Simscape adds dynamics with built-in joint limits and contact modeling. This last step adds simulation accuracy at the cost of modeling complexity and simulation speed.

Simscape also provides a built-in visualization, **Mechanics Explorer**, that can be viewed during and after simulation.

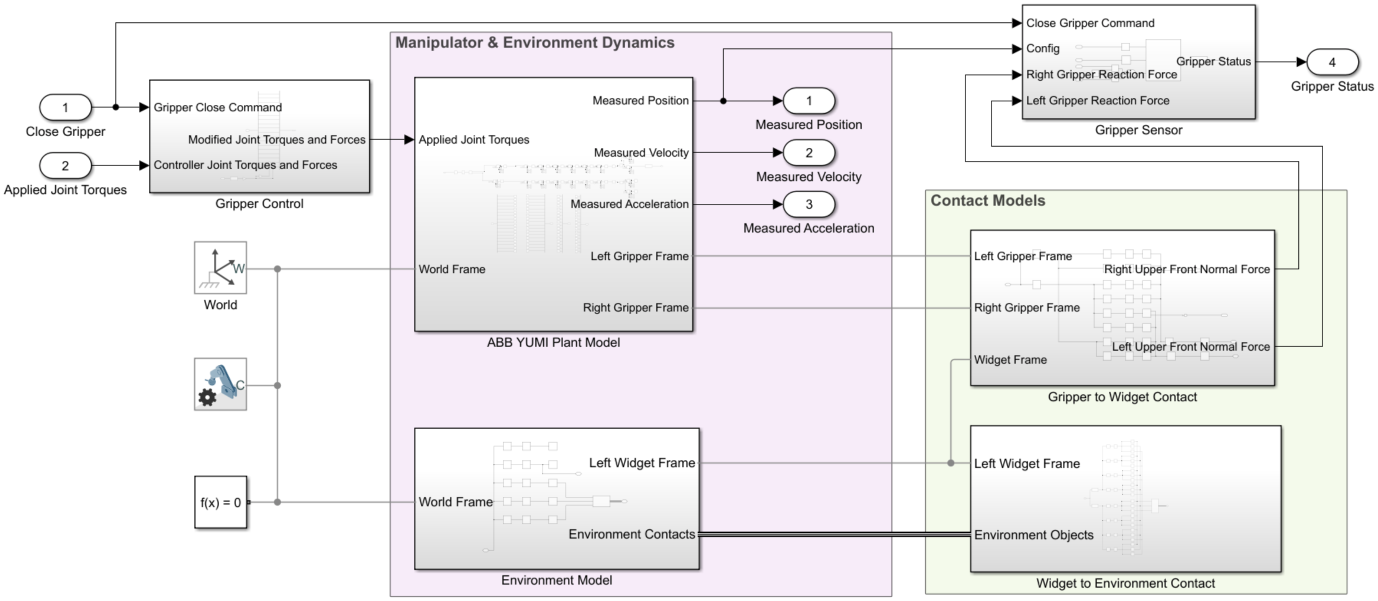
Load the final provided model, which has the same top view.

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



Simscape Robot & Environment Plant

This main different from the previous model is the plant model. The core manipulator dynamics from the previous model have been replaced with a Simscape model for the robot and the environment:



Manipulator & Environment Dynamics

The manipulator and environment are constructed using Simscape Multibody. The robot model was created by calling `smimport` on the robot URDF file with the provided meshes. Then, the joints were actuated with joint torques by linking via **muxes** and **GoTo** tags and outfitted with sensors that return joint position, velocity, and acceleration.

The objects,or widgets, are actually picked up in this simulation, so define the widget size.

```
widgetDimensions = [0.02 0.02 0.07];
```

Contact Models

The contact in this model is split into two categories:

- Contact between the gripper and the widget
- Contact between the widget and the environment

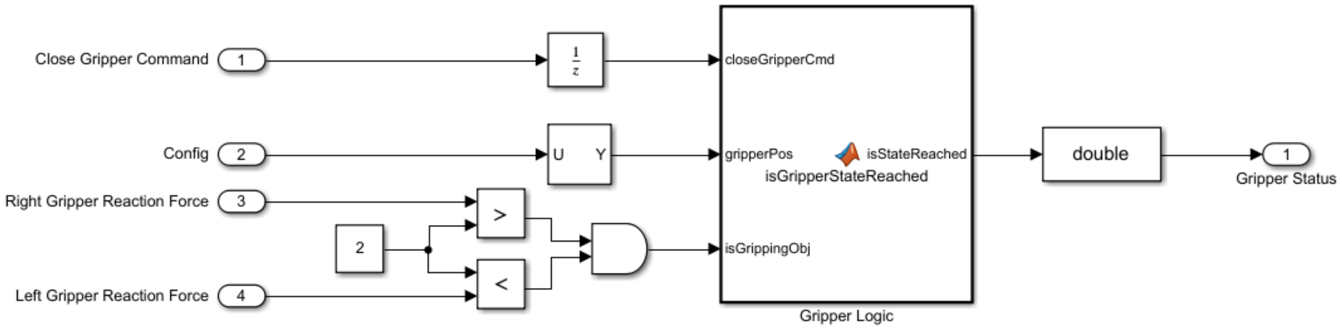
In both cases, *contact proxies* are used in lieu of direct surface-to-surface contact. The use of contact proxies speed up modeling to improve performance. For the gripper-widget contact, the gripper contacts are modeled using two brick solids, while eight spherical contacts are used to model the widget interface. Similarly, the widget-to-environment contact uses spheres at each of the four corners of the widget that contact the brick solids representing the environment.

Define the parameters for the contact models to be close to their default states.

```
% Contact parameters
stiffness = 1e4;
damping = 30;
transition_region_width = 1e-4;
static_friction_coef = 1;
kinetic_friction_coef = 1;
critical_velocity = 1;
```

Gripper Control & Sensing

The **Gripper Control** is the same, but the **Gripper Sensor** is modified. Since this gripper can actually pick up objects, the closed gripper state is reached when the grasp is firm. The actual closed position may never be reached. Therefore, extra logic has been added that returns a value, `isGrippingObj`, that is true when both the left and right gripper reaction forces exceed a threshold value. The **Gripper Logic** MATLAB Function block accepts this variable as an input.

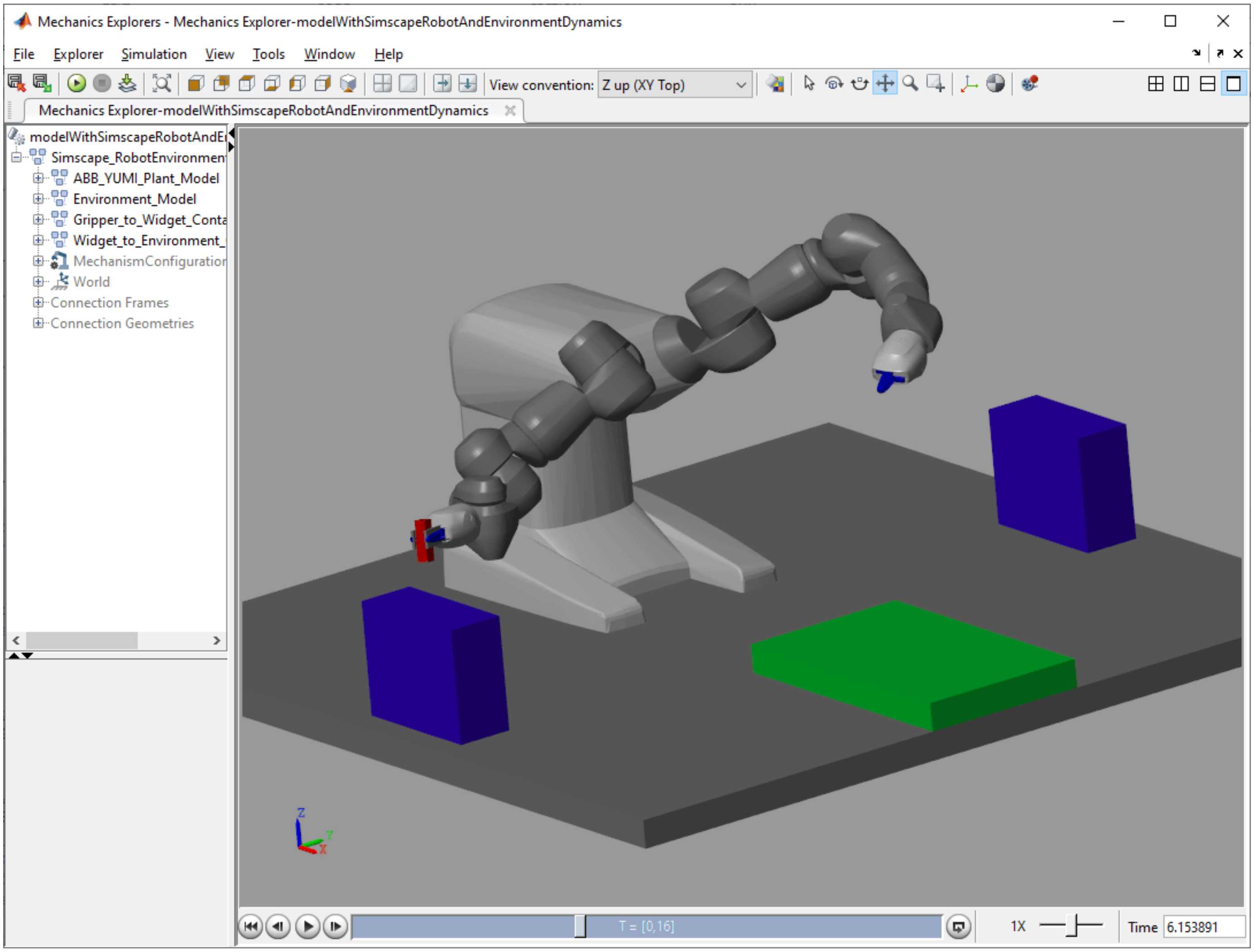


Simulate the Model

Simulate the robot. Due to the high complexity, this may take several minutes.

```
simout = sim('modelWithSimscapeRobotAndEnvironmentDynamics.slx');
```

Use the **Mechanics Explorer** to visualize the performance during and after simulation.



Extensibility

This example has focused on the design of a scheduling and control system for a pick-and-place application. Further investigations might include the effect of sampling on the controller, the impact of unexpected contact using Simscape Multibody, or the extension to a multi-domain model like detailing the behavior of the electrical motors that the robot uses.

Plan and Execute Task- and Joint-Space Trajectories Using KINOVA Gen3 Manipulator

This example shows how to generate and simulate interpolated joint trajectories to move from an initial to a desired end-effector pose. The timing of the trajectories is based on an approximate desired end of arm tool (EOAT) speed.

Load the KINOVA Gen3 rigid body tree (RBT) robot model.

```
robot = loadrobot('kinovaGen3','DataFormat','row','Gravity',[0 0 -9.81]);
```

Set current robot joint configuration.

```
currentRobotJConfig = homeConfiguration(robot);
```

Get number of joints and the end-effector RBT frame.

```
numJoints = numel(currentRobotJConfig);
endEffector = "EndEffector_Link";
```

Specify the trajectory time step and approximate desired tool speed.

```
timeStep = 0.1; % seconds
toolSpeed = 0.1; % m/s
```

Set the initial and final end-effector pose.

```
jointInit = currentRobotJConfig;
taskInit = getTransform(robot,jointInit,endEffector);

taskFinal = trvec2tform([0.4,0,0.6])*axang2tform([0 1 0 pi]);
```

Generate Task-Space Trajectory

Compute task-space trajectory waypoints via interpolation.

First, compute tool traveling distance.

```
distance = norm(tform2trvec(taskInit)-tform2trvec(taskFinal));
```

Next, define trajectory times based on traveling distance and desired tool speed.

```
initTime = 0;
finalTime = (distance/toolSpeed) - initTime;
trajTimes = initTime:timeStep:finalTime;
timeInterval = [trajTimes(1); trajTimes(end)];
```

Interpolate between `taskInit` and `taskFinal` to compute intermediate task-space waypoints.

```
[taskWaypoints,taskVelocities] = transformtraj(taskInit,taskFinal,timeInterval,trajTimes);
```

Control Task-Space Motion

Create a joint space motion model for PD control on the joints. The `taskSpaceMotionModel` object models the motion of a rigid body tree model under task-space proportional-derivative control.

```
tsMotionModel = taskSpaceMotionModel('RigidBodyTree',robot,'EndEffectorName','EndEffector_Link')
```

Set the proportional and derivative gains on orientation to zero, so that controlled behavior just follows the reference positions:

```
tsMotionModel.Kp(1:3,1:3) = 0;
tsMotionModel.Kd(1:3,1:3) = 0;
```

Define the initial states (joint positions and velocities).

```
q0 = currentRobotJConfig;
qd0 = zeros(size(q0));
```

Use `ode15s` to simulate the robot motion. For this problem, the closed-loop system is stiff, meaning that there is a difference in scaling somewhere in the problem. As a result, the integrator is sometimes forced to take exceedingly small steps, and a non-stiff ODE solver such as `ode45` will take much longer to solve the same problem. For more information, refer to “Choose an ODE Solver” and “Solve Stiff ODEs” in the documentation.

Since the reference state changes at each instant, a wrapper function is required to update the interpolated trajectory input to the state derivative at each instant. Therefore, an example helper function is passed as the function handle to the ODE solver. The resultant manipulator states are output in `stateTask`.

```
[tTask,stateTask] = ode15s(@(t,state) exampleHelperTimeBasedTaskInputs(tsMotionModel,timeInterval
```

Generate Joint-Space Trajectory

Create a inverse kinematics object for the robot.

```
ik = inverseKinematics('RigidBodyTree',robot);
ik.SolverParameters.AllowRandomRestart = false;
weights = [1 1 1 1 1 1];
```

Calculate the initial and desired joint configurations using inverse kinematics. Wrap the values to pi to ensure that interpolation is over a minimal domain.

```
initialGuess = jointInit;
jointFinal = ik(endEffector,taskFinal,weights,initialGuess);
```

By default, the IK solution respects joint limits. However for continuous joints (revolute joints with infinite range), the resultant values may be unnecessarily large and can be wrapped to `[-pi, pi]` to ensure that the final trajectory covers a minimal distance. Since non-continuous joints for the Gen3 already have limits within this interval, it is sufficient to simply wrap the joint values to `pi`. The continuous joints will be mapped to the interval `[-pi, pi]`, and the other values will remain the same.

```
wrappedJointFinal = wrapToPi(jointFinal);
```

Interpolate between them using a cubic polynomial function to generate an array of evenly-spaced joint configurations. Use a B-spline to generate a smooth trajectory.

```
ctrlpoints = [jointInit',wrappedJointFinal'];
jointConfigArray = cubicpolytraj(ctrlpoints,timeInterval,trajTimes);
jointWaypoints = bsplinepolytraj(jointConfigArray,timeInterval,1);
```

Control Joint-Space Trajectory

Create a joint space motion model for PD control on the joints. The `jointSpaceMotionModel` object models the motion of a rigid body tree model and uses proportional-derivative control on the specified joint positions.

```
jsMotionModel = jointSpaceMotionModel('RigidBodyTree',robot,'MotionType','PDControl');
```

Set initial states (joint positions and velocities).

```
q0 = currentRobotJConfig;
qd0 = zeros(size(q0));
```

Use `ode15s` to simulate the robot motion. Again, an example helper function is used as the function handle input to the ODE solver in order to update the reference inputs at each instant in time. The joint-space states are output in `stateJoint`.

```
[tJoint,stateJoint] = ode15s(@(t,state) exampleHelperTimeBasedJointInputs(jsMotionModel,timeInte
```

Visualize and Compare Robot Trajectories

Show the initial configuration of the robot.

```
show(robot,currentRobotJConfig,'PreservePlot',false,'Frames','off');
hold on
axis([-1 1 -1 1 -0.1 1.5]);
```

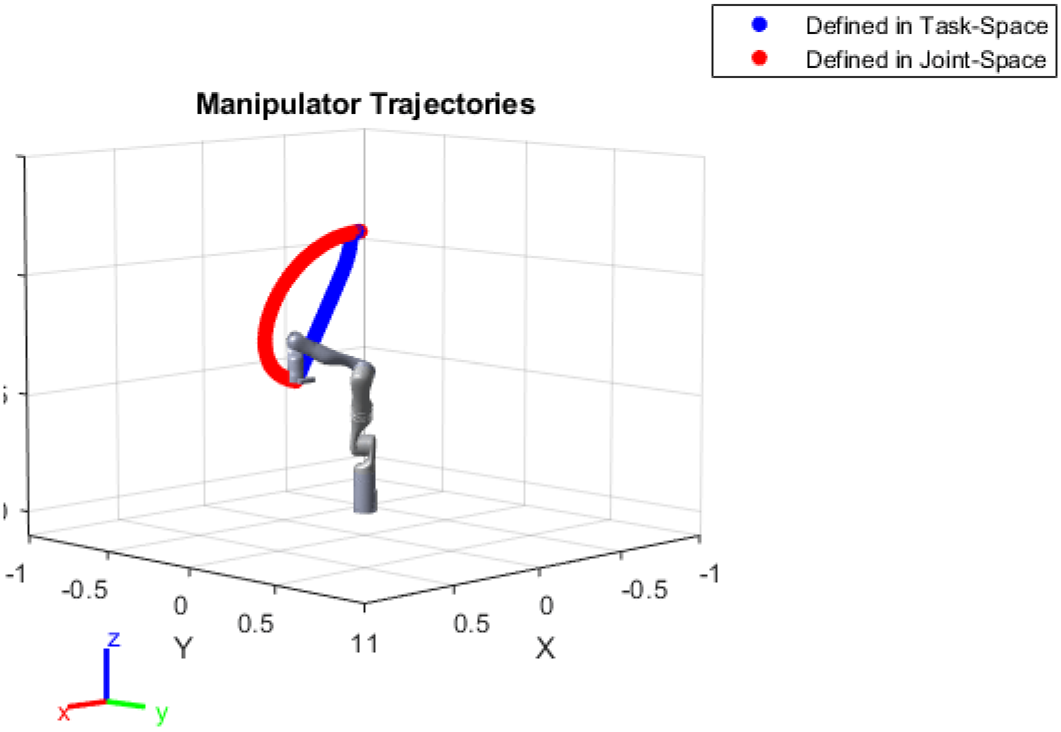
Visualize the task-space trajectory. Iterate through the `stateTask` states and interpolate based on the current time.

```
for i=1:length(trajTimes)
    % Current time 
    tNow= trajTimes(i);
    % Interpolate simulated joint positions to get configuration at current time
    configNow = interp1(tTask,stateTask(:,1:numJoints),tNow);
    poseNow = getTransform(robot,configNow,endEffector);
    show(robot,configNow,'PreservePlot',false,'Frames','off');
    taskSpaceMarker = plot3(poseNow(1,4),poseNow(2,4),poseNow(3,4),'b.','MarkerSize',20);
    drawnow;
end
```

Visualize the joint-space trajectory. Iterate through the `jointTask` states and interpolate based on the current time.

```
% Return to initial configuration
show(robot,currentRobotJConfig,'PreservePlot',false,'Frames','off');

for i=1:length(trajTimes)
    % Current time 
    tNow= trajTimes(i);
    % Interpolate simulated joint positions to get configuration at current time
    configNow = interp1(tJoint,stateJoint(:,1:numJoints),tNow);
    poseNow = getTransform(robot,configNow,endEffector);
    show(robot,configNow,'PreservePlot',false,'Frames','off');
    jointSpaceMarker = plot3(poseNow(1,4),poseNow(2,4),poseNow(3,4),'r.','MarkerSize',20);
    drawnow;
end
```

```
% Add a legend and title
legend([taskSpaceMarker jointSpaceMarker], {'Defined in Task-Space', 'Defined in Joint-Space'});
title('Manipulator Trajectories')
```



Plan and Execute Collision-Free Trajectories Using KINOVA Gen3 Manipulator

This example shows how to plan closed-loop collision-free robot trajectories from an initial to a desired end-effector pose using nonlinear model predictive control. The resulting trajectories are executed using a joint-space motion model with computed torque control. Obstacles can be static or dynamic, and can be either set as primitives (spheres, cylinders, boxes) or as custom meshes.

Robot Description and Poses

Load the KINOVA Gen3 rigid body tree (RBT) model.

```
robot = loadrobot('kinovaGen3', 'DataFormat', 'column');
```

Get the number of joints.

```
numJoints = numel(homeConfiguration(robot));
```

Specify the robot frame where the end-effector is attached.

```
endEffector = "EndEffector_Link";
```

Specify initial and desired end-effector poses. Use inverse kinematics to solve for the initial robot configuration given a desired pose.

```
% Initial end-effector pose
taskInit = trvec2tform([[0.4 0 0.2]])*axang2tform([0 1 0 pi]);

% Compute current robot joint configuration using inverse kinematics
ik = inverseKinematics('RigidBodyTree', robot);
ik.SolverParameters.AllowRandomRestart = false;
weights = [1 1 1 1 1 1];
currentRobotJConfig = ik(endEffector, taskInit, weights, robot.homeConfiguration);

% The IK solver respects joint limits, but for those joints with infinite
% range, they must be wrapped to a finite range on the interval [-pi, pi].
% Since the the other joints are already bounded within this range, it is
% sufficient to simply call wrapToPi on the entire robot configuration
% rather than only on the joints with infinite range.
currentRobotJConfig = wrapToPi(currentRobotJConfig);

% Final (desired) end-effector pose
taskFinal = trvec2tform([0.35 0.55 0.35])*axang2tform([0 1 0 pi]);
anglesFinal = rotm2eul(taskFinal(1:3,1:3),'XYZ');
poseFinal = [taskFinal(1:3,4);anglesFinal']; % 6x1 vector for final pose: [x, y, z, phi, theta, p
```

Collision Meshes and Obstacles

To check for and avoid collisions during control, you must setup a collision `world` as a set of collision objects. This example uses `collisionSphere` objects as obstacles to avoid. Change the following boolean to plan using static instead of moving obstacles.

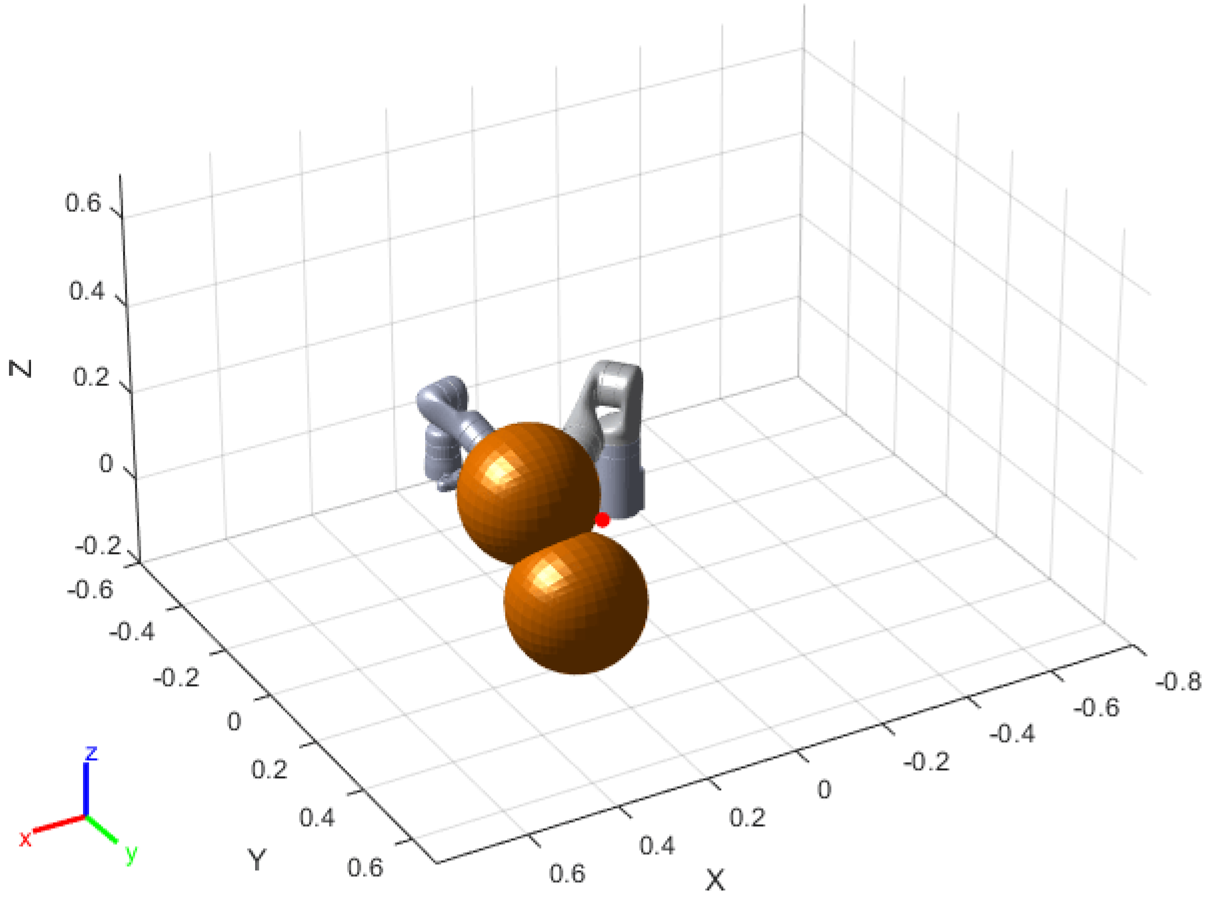
```
isMovingObst = true;
```

The obstacle sizes and locations are initialized in the following helper function. To add more static obstacles, add collision objects in the `world` array.

```
helperCreateObstaclesKINOVA;
```

Visualize the robot at the initial configuration. You should see the obstacles in the environment as well.

```
x0 = [currentRobotJConfig', zeros(1,numJoints)];
helperInitialVisualizerKINOVA;
```



Specify a safety distance away from the obstacles. This value is used in the inequality constraint function of the nonlinear MPC controller.

```
safetyDistance = 0.01;
```

Design Nonlinear Model Predictive Controller

You can design the nonlinear model predictive controller using the following helper file, which creates an `nlmpc` (Model Predictive Control Toolbox) controller object. To views the file, type `edit helperDesignNLMPCobjKINOVA`.

```
helperDesignNLMPCobjKINOVA;
```

The controller is designed based on the following analysis. The maximum number of iterations for the optimization solver is set to 5. The lower and upper bounds for the joint position and velocities (states), and accelerations (control inputs) are set explicitly.

- The robot joints model is described by double integrators. The states of the model are $x = [q, \dot{q}]$, where the 7 joint positions are denoted by q and their velocities are denoted by $\dot{q}$. The inputs of the model are the joint accelerations $u = \ddot{q}$. The dynamics of the model are given by

  $$\dot{x} = \begin{bmatrix} 0 & I_7 \\ 0 & 0 \end{bmatrix} x + \begin{bmatrix} 0 \\ I_7 \end{bmatrix} u$$

where I_7 denotes the 7 × 7 identity matrix. The output of the model is defined as

$$y = [I_7 \ \ 0]x.$$

Therefore, the nonlinear model predictive controller (`nlobj`) has 14 states, 7 outputs, and 7 inputs.

- The cost function for `nlobj` is a custom nonlinear cost function, which is defined in a manner similar to a quadratic tracking cost plus a terminal cost.

  $$J = \int_0^T (p_{\text{ref}} - p(q(t)))' Q_r (p_{\text{ref}} - p(q(t))) + u'(t) Q_u u(t)\, \text{dt} + (p_{\text{ref}} - p(q(T)))' Q_t (p_{\text{ref}} - p(q(T))) \\ + \dot{q}'(T) Q_v \dot{q}(T)$$

Here, $p(q(t))$ transforms the joint positions $q(t)$ to the frame of end effector using forward kinematics and `getTransform`, and p_{ref} denotes the desired end-effector pose.

The matrices Q_r, Q_u, Q_t, and Q_v are constant weight matrices.

- To avoid collisions, the controller has to satisfy the following inequality constraints.

  $$d_{i,\,j} \geq d_{\text{safe}}$$

Here, $d_{i,\,j}$ denotes the distance from the i-th robot body to the j-th obstacle, computed using `checkCollision`.

In this example, i belongs to $\{1, 2, 3, 4, 5, 6\}$ (the base and end-effector robot bodies are excluded), and j belongs to $\{1, 2\}$ (2 obstacles are used).

The Jacobians of the state function, output function, cost function, and inequality constraints are all provided for the prediction model to improve the simulation efficiency. To calculate the inequality constraint Jacobian, use the `geometricJacobian` function and the Jacobian approximation in [1].

Closed-Loop Trajectory Planning

Simulate the robot for a maximum of 50 steps with correct initial conditions.

```
maxIters = 50;
u0 = zeros(1,numJoints);
mv = u0;
time = 0;
goalReached = false;
```

Initialize the data array for control.

```
positions = zeros(numJoints,maxIters);
positions(:,1) = x0(1:numJoints)';

velocities = zeros(numJoints,maxIters);
velocities(:,1) = x0(numJoints+1:end)';
```

```
accelerations = zeros(numJoints,maxIters);
accelerations(:,1) = u0';

timestamp = zeros(1,maxIters);
timestamp(:,1) = time;
```

Use the nlmpcmove (Model Predictive Control Toolbox) function for closed-loop trajectory generation. Specify the trajectory generation options using an nlmpcmoveopt (Model Predictive Control Toolbox) object. Each iteration calculates the position, velocity, and acceleration of the joints to avoid obstacles as they move towards the goal. The helperCheckGoalReachedKINOVA script checks if the robot has reached the goal. The helperUpdateMovingObstacles script moves the obstacle locations based on the time step.

```
options = nlmpcmoveopt;
for timestep=1:maxIters
    disp(['Calculating control at timestep ', num2str(timestep)]);
    % Optimize next trajectory point
    [mv,options,info] = nlmpcmove(nlobj,x0,mv,[],[], options);
    if info.ExitFlag < 0
        disp('Failed to compute a feasible trajectory. Aborting...')
        break;
    end
    % Update states and time for next iteration
    x0 = info.Xopt(2,:);
    time = time + nlobj.Ts;
    % Store trajectory points
    positions(:,timestep+1) = x0(1:numJoints)';
    velocities(:,timestep+1) = x0(numJoints+1:end)';
    accelerations(:,timestep+1) = info.MVopt(2,:)';
    timestamp(timestep+1) = time;
    % Check if goal is achieved
    helperCheckGoalReachedKINOVA;
    if goalReached
        break;
    end
    % Update obstacle pose if it is moving
    if isMovingObst
        helperUpdateMovingObstaclesKINOVA;
    end
end
```

```
Calculating control at timestep 1

Slack variable unused or zero-weighted in your custom cost function. All constraints will be har

Calculating control at timestep 2

Slack variable unused or zero-weighted in your custom cost function. All constraints will be har

Calculating control at timestep 3

Slack variable unused or zero-weighted in your custom cost function. All constraints will be har

Calculating control at timestep 4

Slack variable unused or zero-weighted in your custom cost function. All constraints will be har

Calculating control at timestep 5
```

```
Slack variable unused or zero-weighted in your custom cost function. All constraints will be har

Calculating control at timestep 6

Slack variable unused or zero-weighted in your custom cost function. All constraints will be har

Calculating control at timestep 7

Slack variable unused or zero-weighted in your custom cost function. All constraints will be har

Target configuration reached.
```

Execute Planned Trajectory using Joint-Space Robot Simulation and Control

Trim the trajectory vectors based on the time steps calculated from the plan.

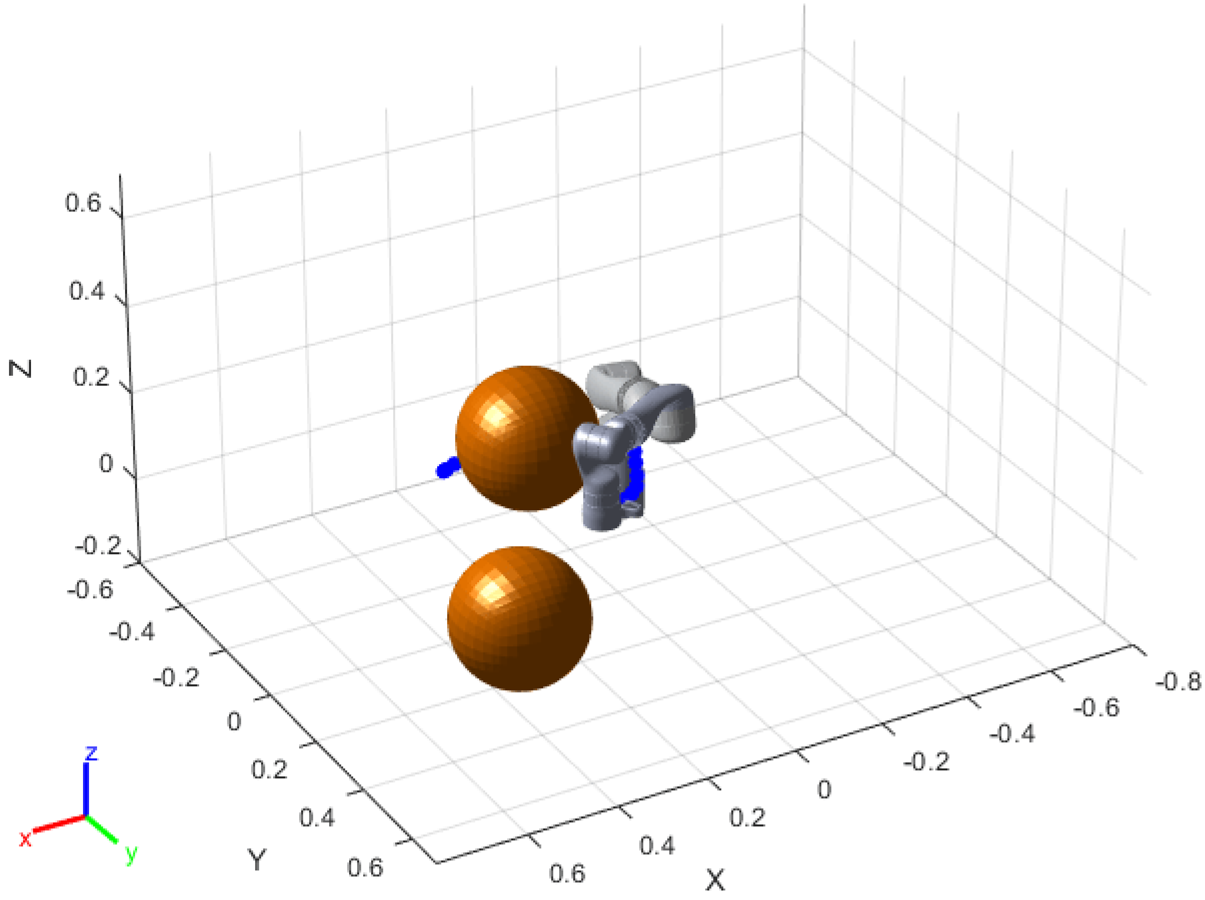
```
tFinal = timestep+1;
positions = positions(:,1:tFinal);
velocities = velocities(:,1:tFinal);
accelerations = accelerations(:,1:tFinal);
timestamp = timestamp(:,1:tFinal);

visTimeStep = 0.2;
```

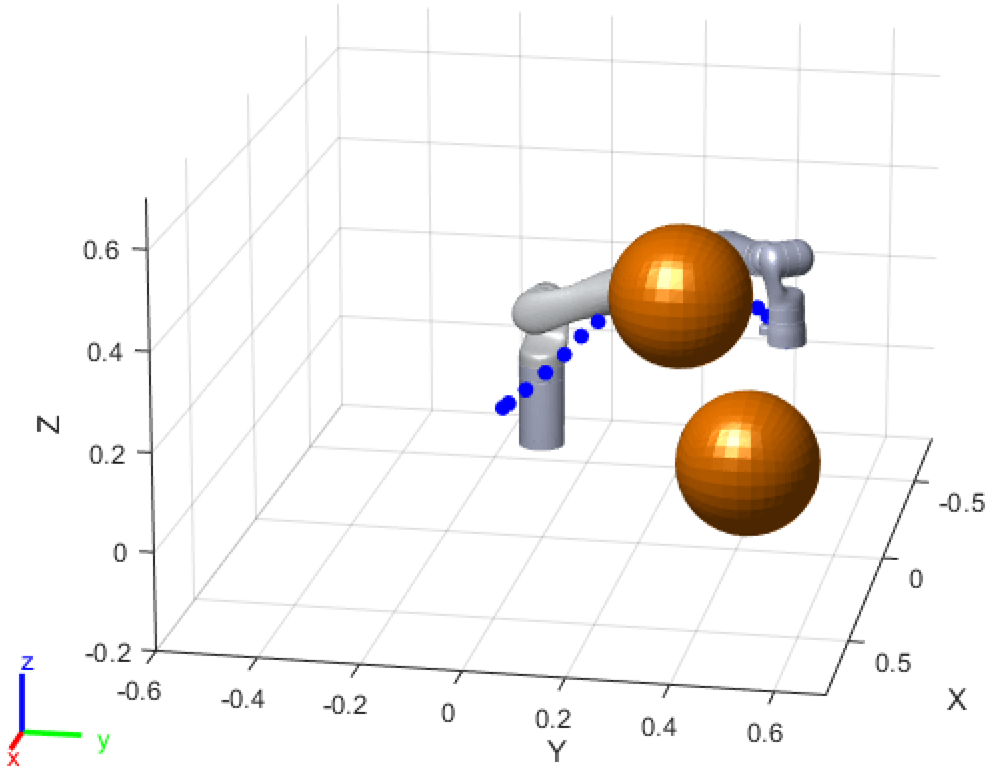
Use a `jointSpaceMotionModel` to track the trajectory with computed-torque control. The `helperTimeBasedStateInputsKINOVA` function generates the derivative inputs for the `ode15s` function for modelling the computed robot trajectory.

```
motionModel = jointSpaceMotionModel('RigidBodyTree',robot);

% Control robot to target trajectory points in simulation using low-fidelity model
initState = [positions(:,1);velocities(:,1)];
targetStates = [positions;velocities;accelerations]';
[t,robotStates] = ode15s(@(t,state) helperTimeBasedStateInputsKINOVA(motionModel,timestamp,targe
                                    [timestamp(1):visTimeStep:timestamp(end)],initState);
```

Visualize the robot motion.

```
helperFinalVisualizerKINOVA;
```


Z
0.6
0.4
0.2
0
-0.2
Y
-0.6
-0.4
-0.2
0
0.2
0.4
0.6
X
0.6
0.4
0.2
0
-0.2
-0.4
-0.6
-0.8
z
x
y



[1] Schulman, J., et al. "Motion planning with sequential convex optimization and convex collision checking." *The International Journal of Robotics Research* 33.9 (2014): 1251-1270.

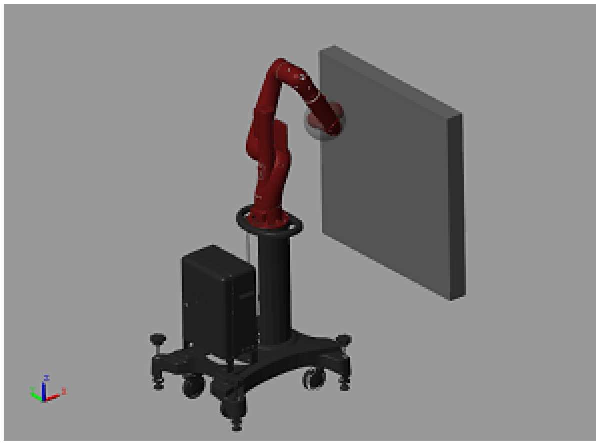
Perform Safe Trajectory Tracking Control Using Robotics Manipulator Blocks

This example shows you how to use Simulink® with manipulator algorithm blocks to achieve safe trajectory tracking control of a simulated robot.

Both Robotics System Toolbox™ and Simscape Multibody™ are required to run this example.

Introduction

This example provides a model that implements a computed-torque controller with joint position and velocity feedback using manipulator algorithm blocks. The controller receives joint position and velocity information from a simulated robot (implemented using Simscape Multibody) and sends torque commands to drive the robot along a given joint trajectory. A planar object is placed in front of the robot so the end effector of the robot arm will collide with it when the trajectory is executed. Without any additional setup, the torque felt from colliding with the object can cause damage on the robot or the object. To achieve safe trajectory tracking, a trajectory scaling block is built to adjust the time stamp when assigning the desired motion to the controller. You may adjust some parameters and interact with the robot while the model is running and observe the effect on the simulated robot.



Set Up Robot Model

This example uses a model of the Rethink Sawyer, a seven degree-of-freedom robot manipulator. Call `importrobot` to generate a `rigidBodyTree` model from a description stored in a Unified Robot Description Format (URDF) file. Set the `DataFormat` and `Gravity` properties to be consistent with Simscape. The Simulink model accesses this robot model from the workspace in the simulation.

```
sawyer = importrobot('sawyer.urdf');
sawyer.removeBody('head');
sawyer.DataFormat = 'column';
sawyer.Gravity = [0, 0, -9.80665];
```

Trajectory Generation

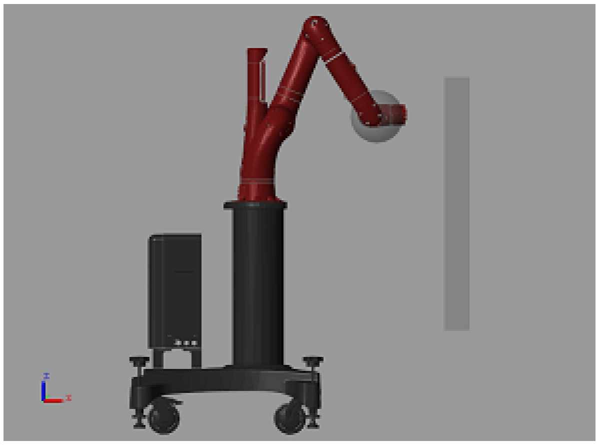
First, assign the start time and duration for the trajectory.

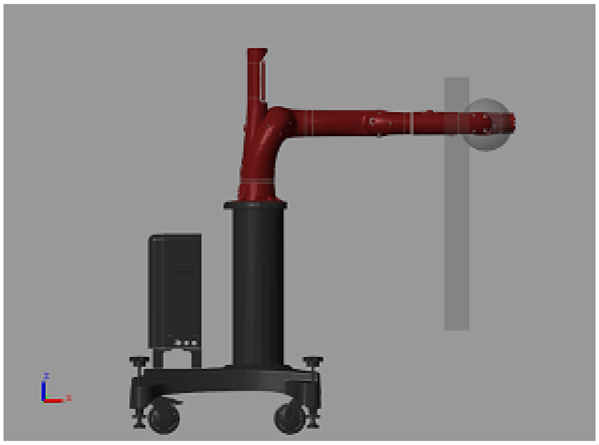
```
tStart = 0.5;
tDuration = 3;
```

Next, assign the initial and target configuration. `q0` is the initial configuration and `q1` is the target configuration.

```
q0 = [0; -1.18; 0; 2.18; 0; -1.0008; 3.3161];
q1 = zeros(7,1);
```

The following figures show the robot visualization of the initial configuration and the target configuration related to the location of the planar object. The planar object is placed so that the robot will collide to it during trajectory tracking.





In the Simulink model, the Polynomial Trajectory block computes the robot's position, velocity, and acceleration at any instant in the trajectory using a fifth-order polynomial.

Simulink Model Overview

Next, open the Simulink model. The variables generated above are already stored in Simulink model workspace:

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

The `robotSafeTrajectoryTracking` model implements a computed torque controller with trajectory scaling for safe trajectory tracking. There are three main subsystems in this model:

- Computed Torque Controller
- Trajectory Scaling and Desired Motion
- Simscape Multibody Model with Simple Contact Mechanics

On each time step, if the trajectory scaling switch is on, the modified time stamp is used for evaluating the desired joint position, velocity and acceleration. Then, the computed torque controller uses the manipulator blocks associated with the `rigidBodyTree` model to track the desired motion. The derived control input is fed into the Sawyer model in Simscape Multibody, where the planar object for interacting with the robot is included.

Build Computed Torque Controller

For a manipulator with n non-fixed joints, the system dynamics can be expressed as:

$$M(q)\ddot{q} + C(q, \dot{q})\dot{q} + G(q) = u$$

where q, $\dot{q}$, $\ddot{q}$, $\in R^n$ are the position, velocity and acceleration of the generalized coordinate, $u \in R^n$ is the control input (torque), $M(q)$ is the joint space mass matrix, $C(q, \dot{q})\dot{q}$ is the velocity product torque, $G(q)$ is the gravity torque. To track along a desired joint trajectory with desired position q_d, velocity $\dot{q}_d$ and acceleration $\ddot{q}_d$, the computed torque controller calculates the torque needed to obtain a given configuration and velocity, provided the robot dynamics variables $M(q)$, $C(q, \dot{q})\dot{q}$, and $G(q)$. In Simulink, these variables can be easily derived using robotics manipulator blocks from Robotics System Toolbox to design the following computed torque controller:

$$u = M(q)(\ddot{q}_d - K_d\dot{q}_e - K_pq_e) + C(q, \dot{q})\dot{q} + G(q)$$

where $q_e = q - q_d$ is the position error and $\dot{q}_e = \dot{q} - \dot{q}_d$ is the velocity error. With this controller input, the system dynamics becomes a second-order ODE:

$$\ddot{q}_e + K_d\dot{q}_e + K_pq_e = 0$$

By choosing K_d and K_p properly, the tracking error q_e will converge to zero when time approaches infinity.

The **Computed Torque Controller** subsystem is built using three robotics manipulator blocks: Joint Space Mass Matrix, Velocity Product Torque, and Gravity Torque. Note that the associated `rigidBodyTree` model, `sawyer`, is assigned in all those blocks, and the configuration and velocity need to be specified as column vectors.

```
open_system('robotSafeTrajectoryTracking/Computed Torque Controller');
```

Inside the **Computed Torque Controller**, there are two tunable parameters (indicated by colored blocks):

- Gain `Kp`: The proportional gain when correcting the robot configuration
- Gain `Kd`: The derivative gain when correcting the robot configuration

A standard way to determine the `Kp` and `Kd` is to calculate them as:

$$K_p = \omega_n^2$$

$$K_d = 2\omega_n\xi$$

where ω_n and ξ are the natural frequency and damping ratio of the second-order ODE. In this example, the default value of `Kp` and `Kd` are derived by setting the natural frequency and damping ratio as $\omega_n = 10$ and $\xi = 1$ to make the second-order ODE a critical damped system.

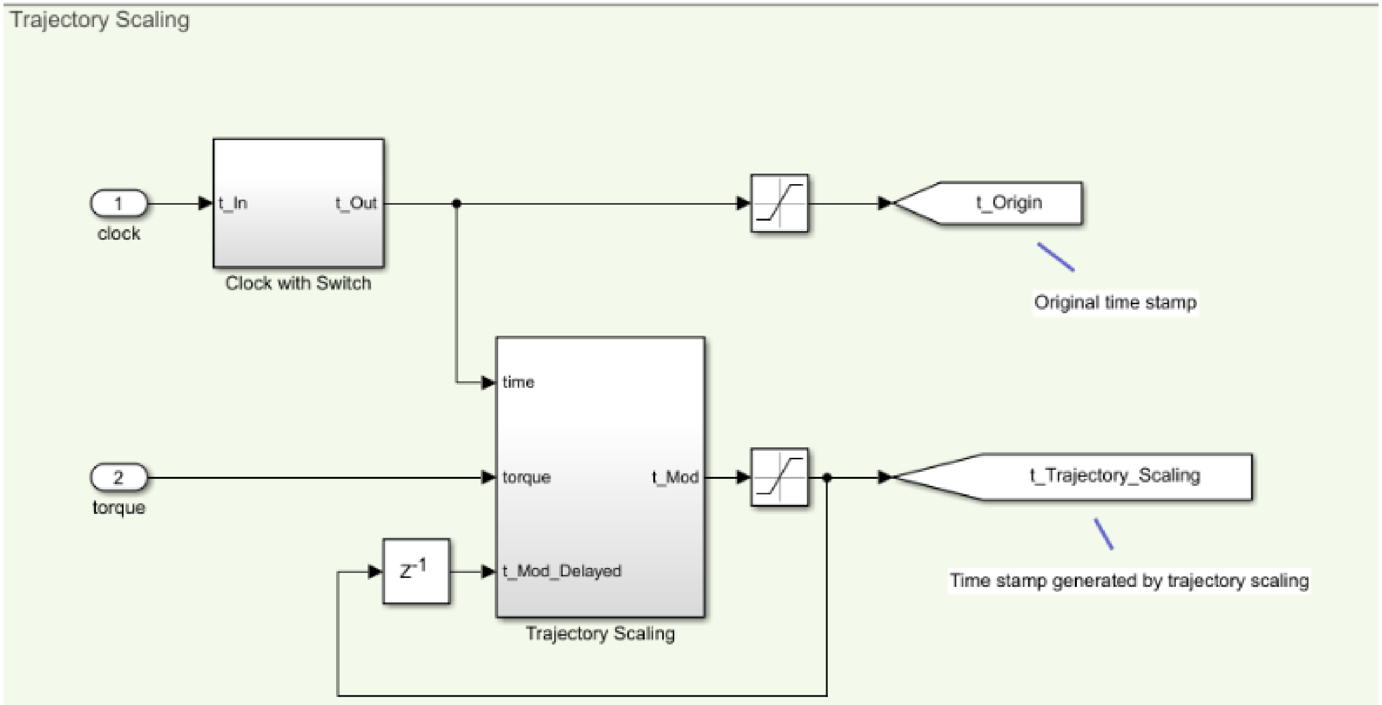
Trajectory Scaling

There are two main blocks in this subsystem:

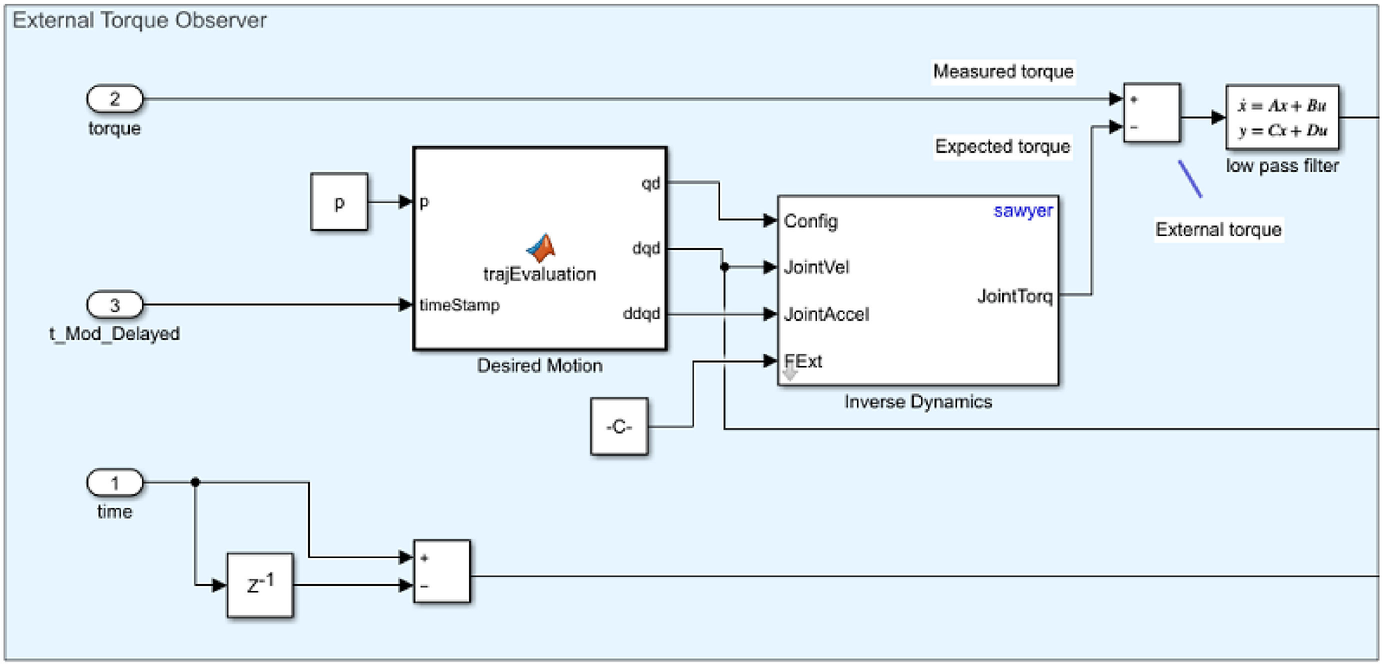
- Trajectory Scaling
- Desired Motion

Trajectory Scaling is the main block deployed for safe trajectory tracking in this example. At each time step t_i, the original time stamp is calculated as $t_i = t_{i-1} + \Delta t$. However, when the robot collides with an unexpected object, the increasing torque and deviance from the planned trajectory can be destructive for both the robot and the object. The main idea of trajectory scaling is to calculate the time stamp as $t_i = t_{i-1} + f_s(q_d, \dot{q}_d, \ddot{q}_d, \tau_{mea})\Delta t$ by introducing $f_s(q_d, \dot{q}_d, \ddot{q}_d, \tau_{mea}) \in [-1, 1]$, which is a function of the desired motion and measured torque τ_{mea}. The function $f_s()$ controls the speed of the robot motion and is determined based on the interference felt by the robot. If the measured torques are greater than expected, $f_s()$ is decreased to make the robot slow down or even move backward until the desired torques are achieved. These values of $f_s()$ have the following effects on the robot's motion:

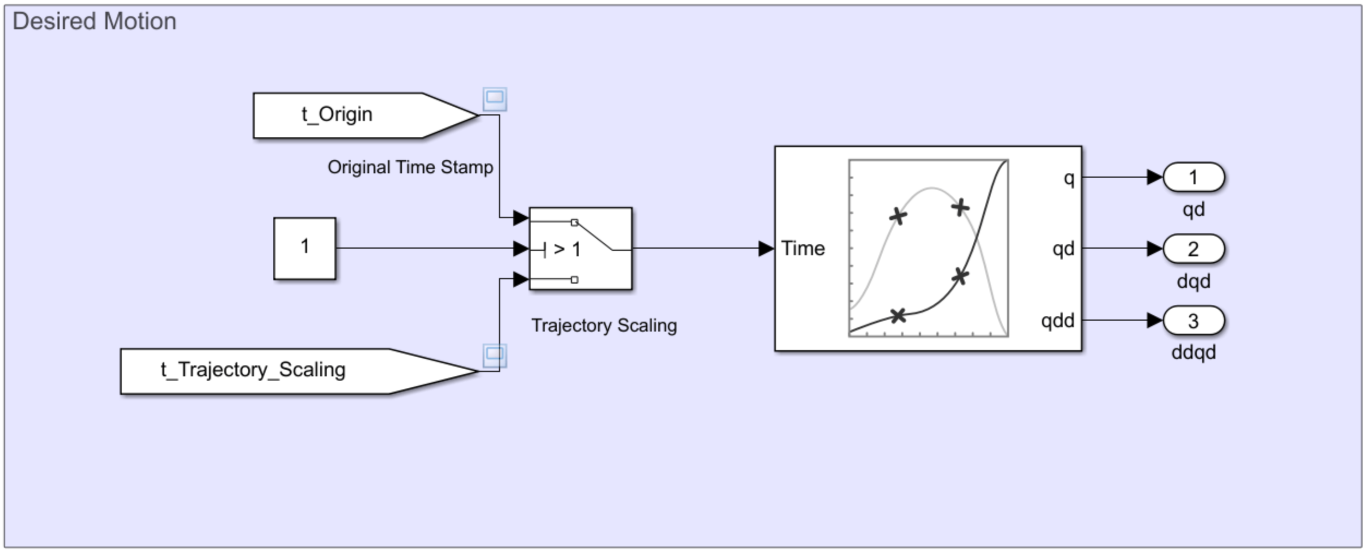
- $f_s() > 0$, the robot moves forward ($f_s() = 1$ is the normal speed).
- $f_s() = 0$, the robot stops.
- $f_s() < 0$, the robot moves backward.



In the **Trajectory Scaling** block, it is required to estimate the external torque $\hat{\tau}_{ext} = \tau_{mea} - \hat{\tau}$ to calculate $f_s()$, where τ_{mea} is the measured torque from the Simscape model, and $\hat{\tau}$ is the expected torque of the desired motion on the previous time stamp. In the External Torque Observer section of the model, the Inverse Dynamics block calculates the expected torque which is subtracted from the measure torque. Expected torque is: $\hat{\tau} = M(q_d)(\ddot{q}_d) + C(q_d, \dot{q}_d)\dot{q}_d + G(q_d)$.

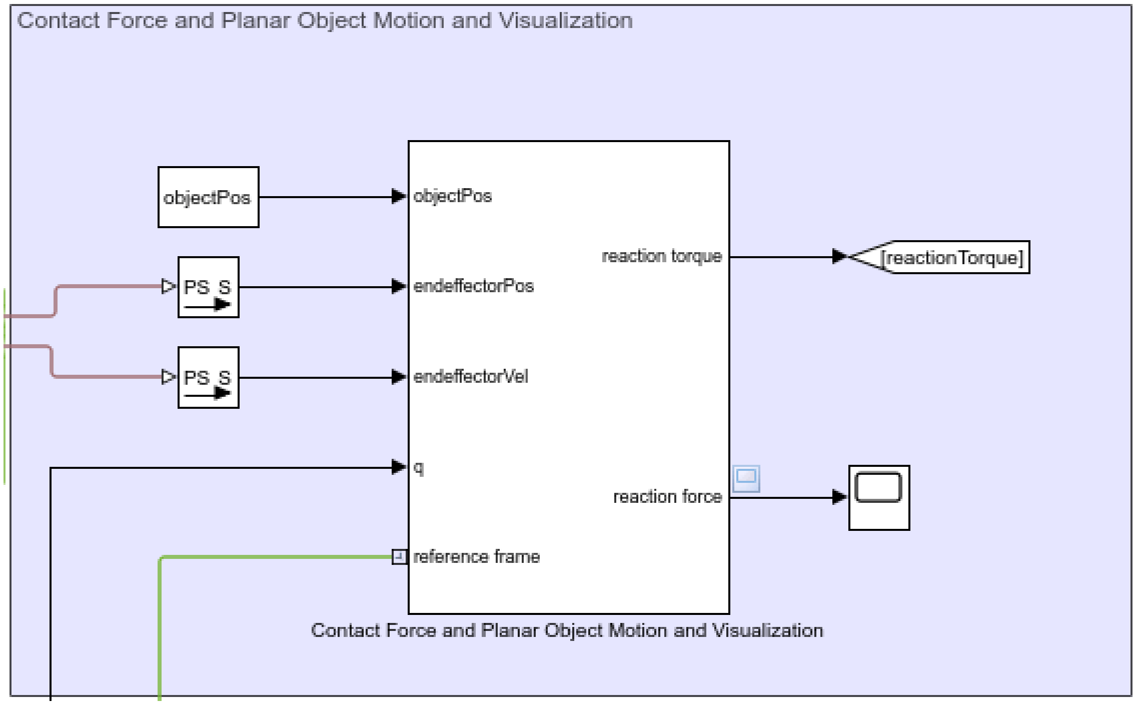


In the **Desired Motion** area, the output of the trajectory scaling algorithm is fed as an input to a **Polynomial Trajectory** block. This block compute a quintic polynomial trajectory given the known values of `q0` and `q1` and their boundary conditions of zero velocity and acceleration. It outputs the position, velocity, and acceleration: $q_d(t)$, $\dot{q}_d(t)$, and $\ddot{q}_d(t)$, which are fed to the **Computed Torque Controller** subsystem.



Simscape Multibody Robot Model and Simple Contact Mechanics

The Simscape Multibody robot model is imported from the same `.urdf` file using `smimport`, where a set of torque actuators and sensors for joint torque, joint position and velocity are added. A contact mechanism block simulates the reaction force between the end effector and the obstacle as a sphere and a plane, where a simple linear spring-damper model is used.

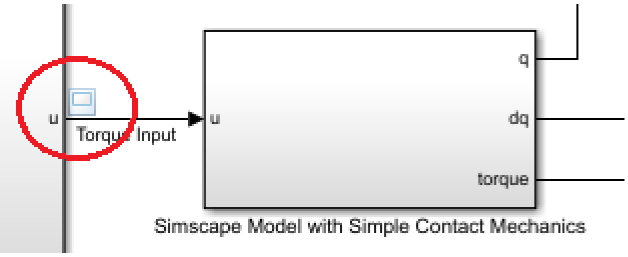


Note: The contact mechanism has only been implemented between the end effector and the planar object. Therefore, other parts of the robot arm may still pass through the obstacle.

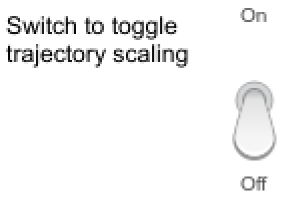
Run the Model

Run the model and observe the behavior of Sawyer in the robot simulator and interact with it.

First, open the viewer by clicking on the scope icon shown below on the left of the Simscape model block. The scope displays signals including the joint torques, reaction contact force between the end effector and the planar object, and the time stamp for calculating desired motion for trajectory tracking.



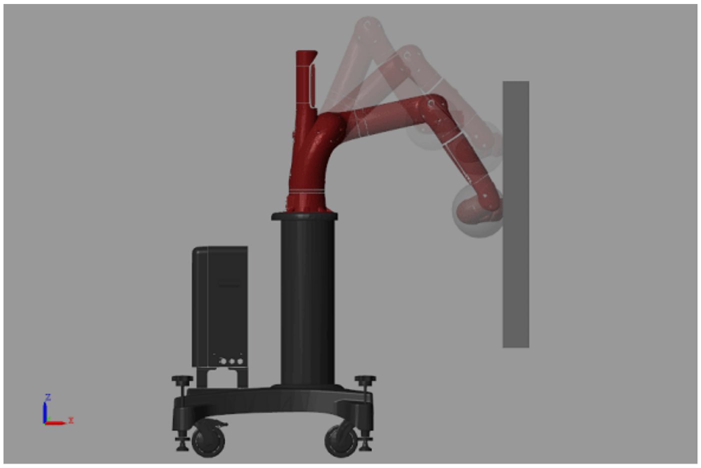
Toggle the trajectory scaling switch to "Off".

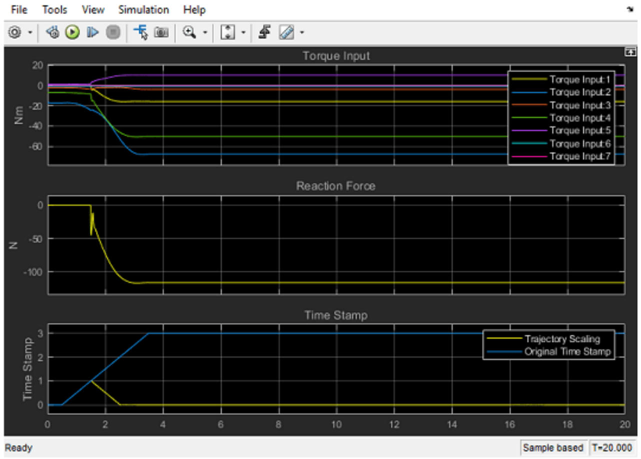


Use the following command or click **Run** to start the simulation.

```
sim('robotSafeTrajectoryTracking.slx','StopTime','5');
```

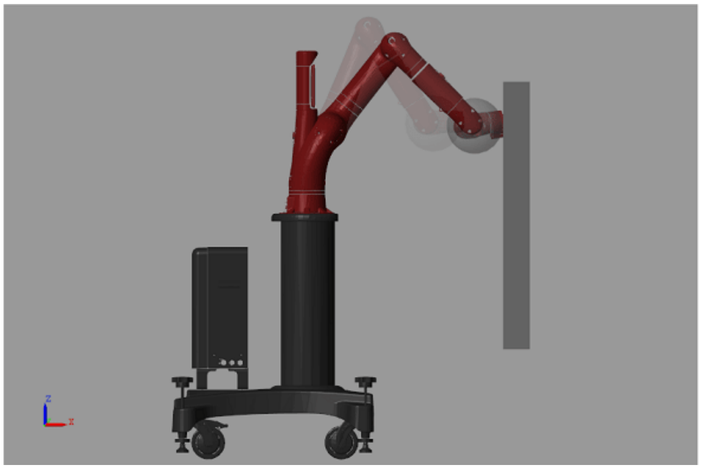
You should see the arm collide with the object yielding high torque inputs and a large reaction force. Note in this case the original time stamp is used. Stop the simulation afterwards.

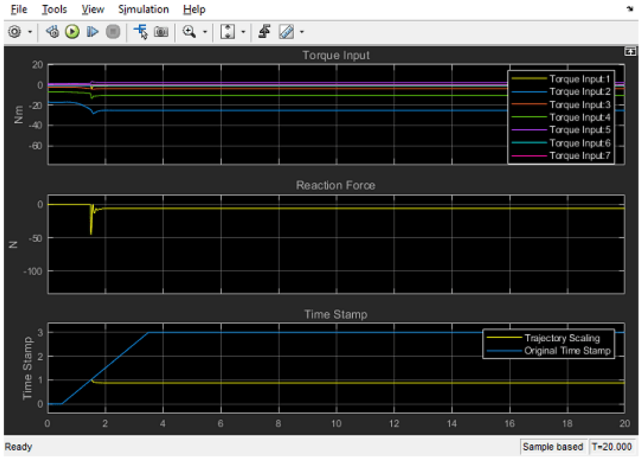




Now, toggle trajectory scaling switch to **On** and rerun the model.

Notice the differences in the computed torques and the reduced reaction force after the collision.





While the simulation is running, adjust the slider to move the object towards or away from the robot. The robot should react to its position while still trying to execute the trajectory safely.

Summary

This example showed how to use robotics manipulator blocks in Simulink to design a computed torque controller, and integrate it with trajectory scaling and dynamic simulation in Simscape Multibody to achieve safe trajectory tracking. The resultant torque, reaction force, and time stamp demonstrated the capability of trajectory scaling for performing safe trajectory tracking.

Pick and Place Using RRT for Manipulators

Using manipulators to pick and place objects in an environment may require path planning algorithms like the rapidly-exploring random tree planner. The planner explores in the joint-configuration space and searches for a collision-free path between different robot configurations. This example shows how to use the `manipulatorRRT` object to tune the planner parameters and plan a path between two joint configurations based on a `rigidBodyTree` robot model of the Franka Emika™ Panda robot. After tuning the planner parameters, the robot manipulator plans a path to move a can from one place to another.

Load Robot Model and Environment

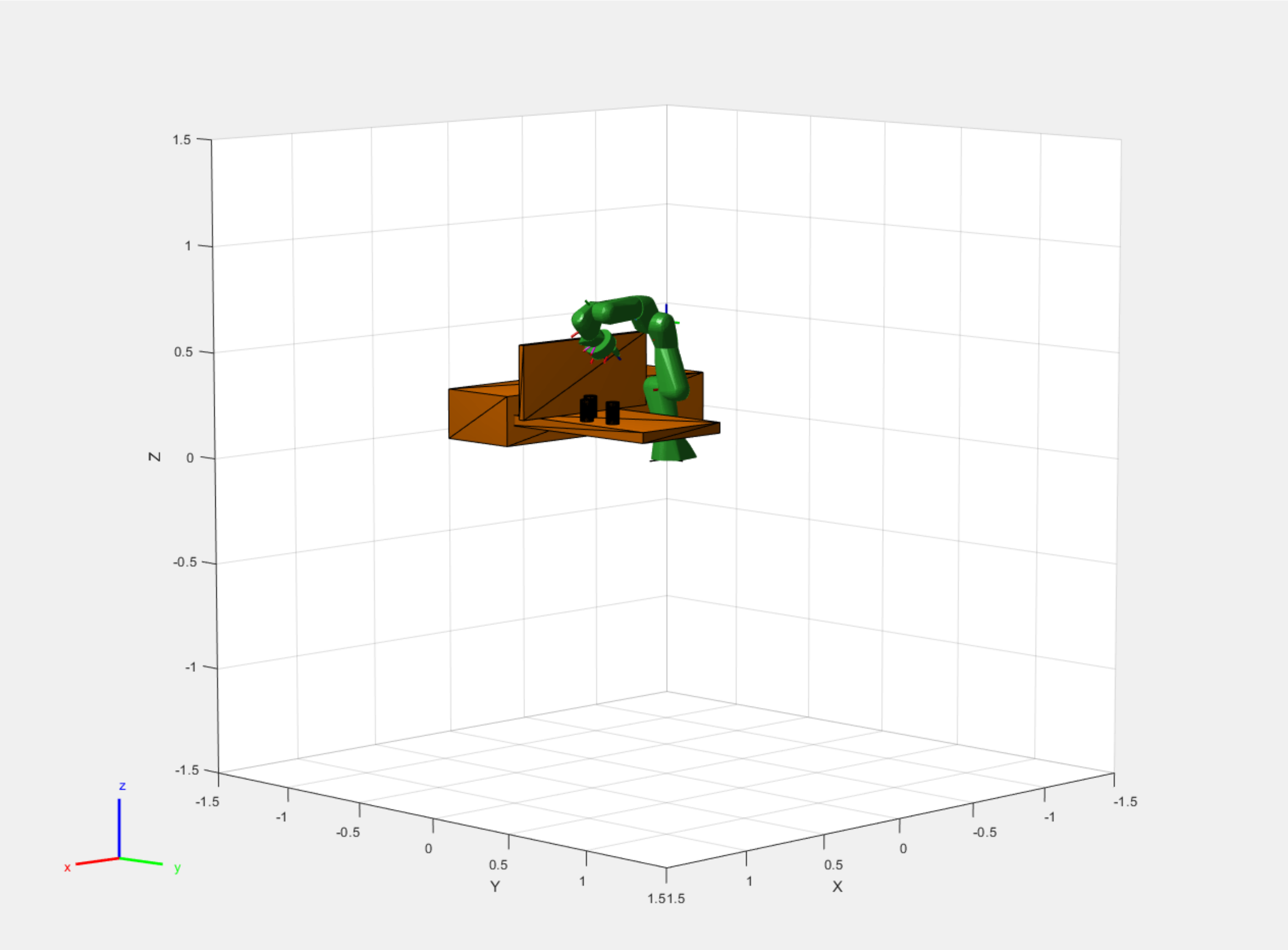
Load the robot and its environment using the `exampleHelperLoadPickAndPlaceRRT` function. The function outputs three variables:

- `franka` — A Franka Emika Panda robot model as a `rigidBodyTree` object. The model has been modified to remove some adjacent collision meshes that are always in collision and adjust position limits based on feasibility.
- `config` — An initial configuration of joint positions for the robot.
- `env` — A set collision objects as a cell array that represent the robot's environment. The path planner checks for self-collisions and collisions with this environment.

```
[franka,config,env] = exampleHelperLoadPickAndPlaceRRT;
```

Visualize the robot model's collision meshes and the environment objects.

```
figure("Name","Pick and Place Using RRT",...
    "Units","normalized",...
    "OuterPosition",[0, 0, 1, 1],...
    "Visible","on");
show(franka,config,"Visuals","off","Collisions","on");
hold on
for i = 1:length(env)
    show(env{i});
end
```



Planner

Create the RRT path planner and specify the robot model and the environment. Define some parameters, which are later tuned, and specify the start and goal configuration for the robot.

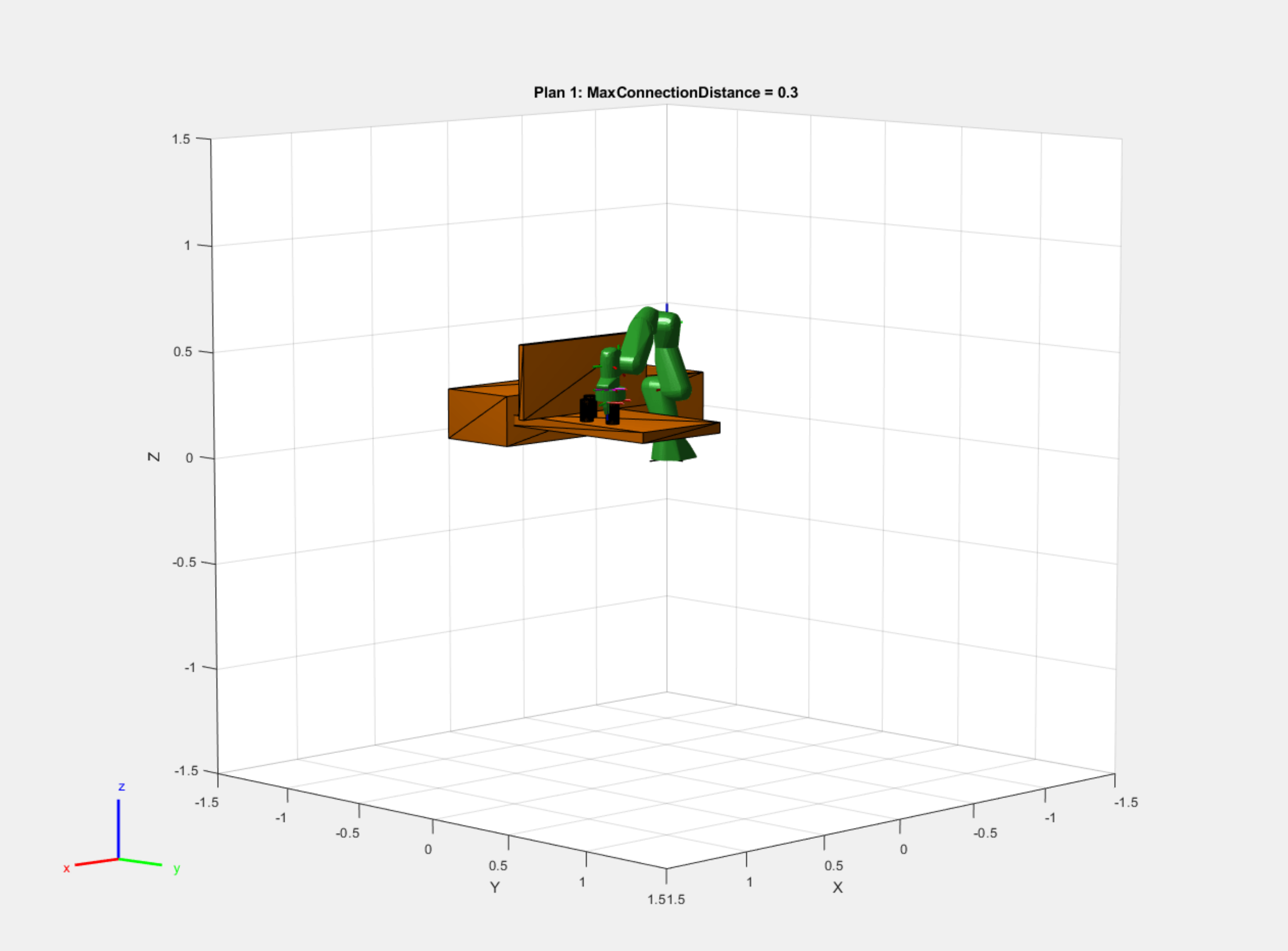
```
planner = manipulatorRRT(franka, env);

planner.MaxConnectionDistance = 0.3;
planner.ValidationDistance = 0.1;

startConfig = config;
goalConfig = [0.2371   -0.0200    0.0542   -2.2272    0.0013    2.2072   -0.9670    0.0400    0.0
```

Plan the path between configurations. The RRT planner should generate a rapidly-exploring tree of random configurations to explore the space and eventually returns a collision-free path through the environment. Before planning, reset the MATLAB's random number generator for repeatabile results.

```
rng('default');
path = plan(planner,startConfig,goalConfig);
```

To visualize the entire path, interpolate the path into small steps. By default, the `interpolate` function generates all of the configurations that were checked for feasibility (collision checking) based on the `ValidationDistance` property.

```
interpStates = interpolate(planner, path);

for i = 1:2:size(interpStates,1)
    show(franka, interpStates(i,:),...
        "PreservePlot", false,...
        "Visuals","off",...
        "Collisions","on");
    title("Plan 1: MaxConnectionDistance = 0.3")
    drawnow;
end
```



Tuning the Planner

Tune the path planner by modifying the `MaxConnectionDistance`, `ValidationDistance`, `EnableConnectHeuristic` properties on the `planner` object.

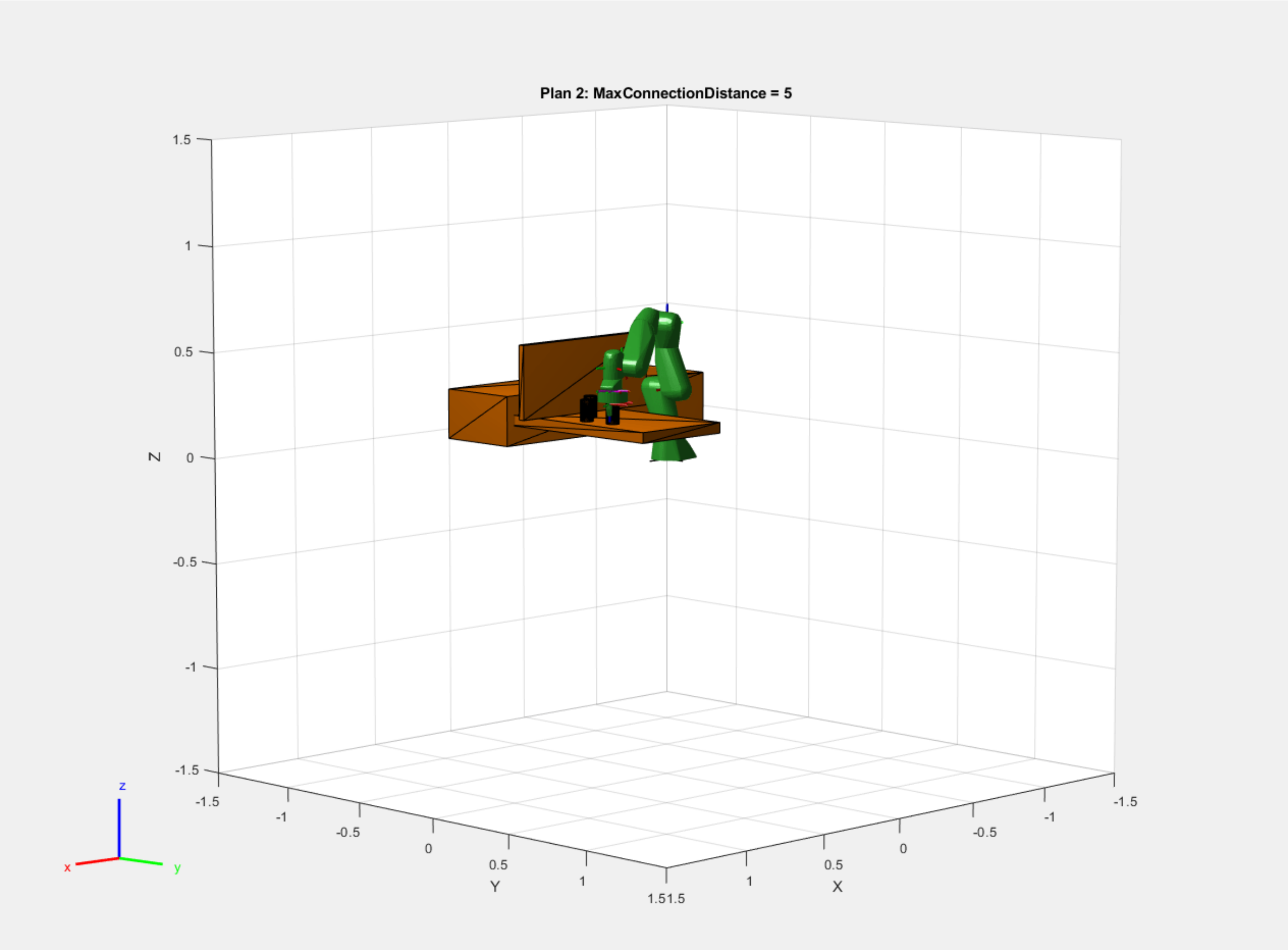
Setting the MaxConnectionDistance property to a larger value causes longer motions in the planned path, but also enables the planner to greedily explore the space. Use tic and toc functions to time the plan function for reference on how these parameters can affect the execution time.

```
planner.MaxConnectionDistance = 5;
tic
path = plan(planner,startConfig,goalConfig);
toc
```

```
Elapsed time is 8.377993 seconds.
```

Notice the change in the path. The robot arm swings much higher due to the larger connection distance.

```
interpStates = interpolate(planner, path);

for i = 1:2:size(interpStates, 1)
    show(franka,interpStates(i,:),...
        "PreservePlot",false,...
        "Visuals","off",...
        "Collisions","on");
    title("Plan 2: MaxConnectionDistance = 5")
    drawnow;
end
```

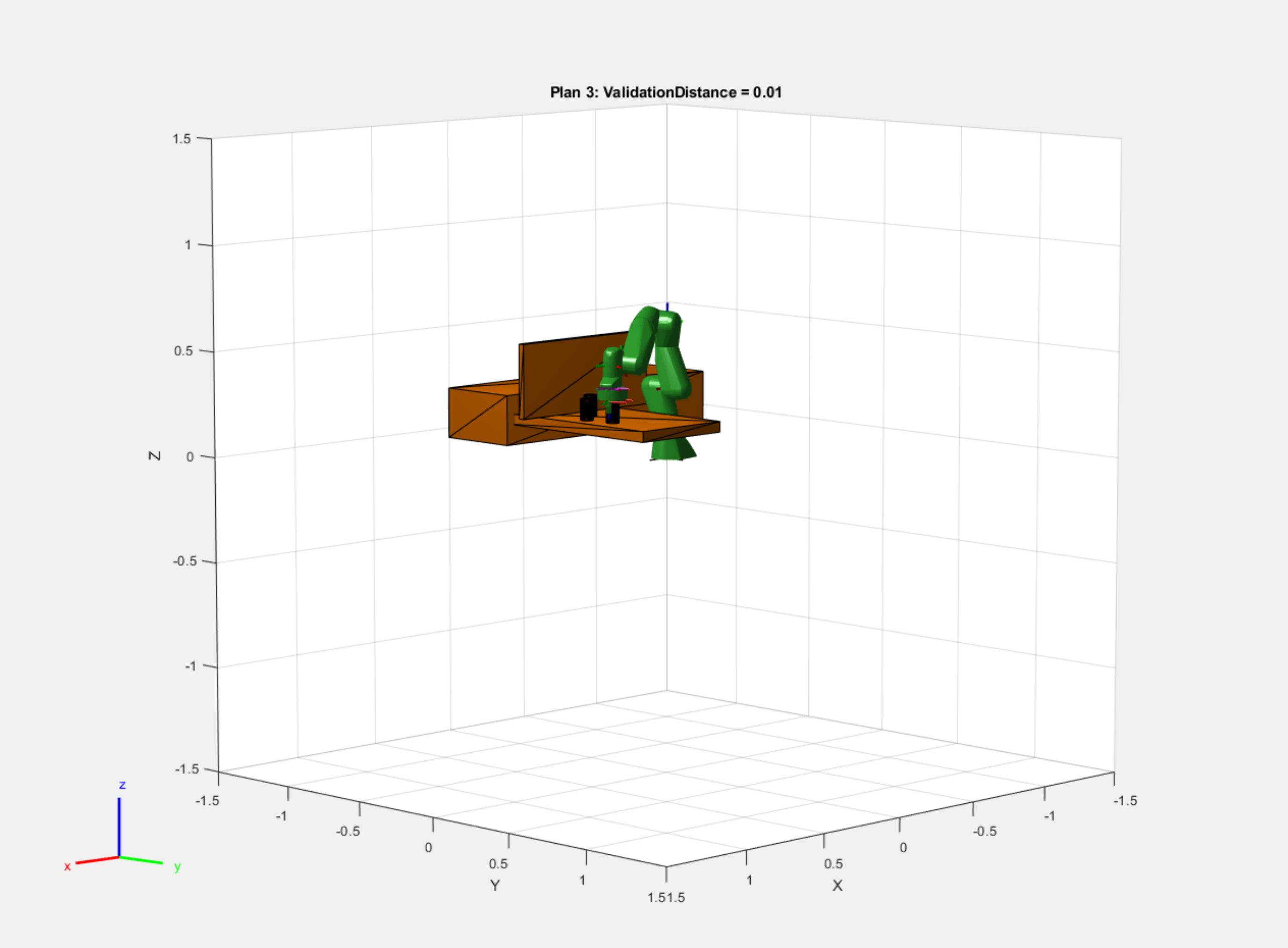


Setting the `ValidationDistance` to a smaller value enables finer validation of the motion along an edge in the planned path. Increasing the number of configurations to be validated along a path leads to longer planning times. A smaller value is useful in case of a cluttered environment with a lot of collision objects. Because of the small validation distance, `interpStates` has a larger number of elements. For faster visualization, the `for` loop skips more states in this step for faster visualization.

```
planner.MaxConnectionDistance = 0.3;
planner.ValidationDistance = 0.01;

tic
path = plan(planner,startConfig,goalConfig);
toc
```

```
Elapsed time is 11.456217 seconds.
```

```
interpStates = interpolate(planner,path);
for i = 1:10:size(interpStates,1)
    show(franka, interpStates(i,:),...
        "PreservePlot",false,...
        "Visuals","off",...
        "Collisions","on");
    title("Plan 3: ValidationDistance = 0.01")
    drawnow;
end
```

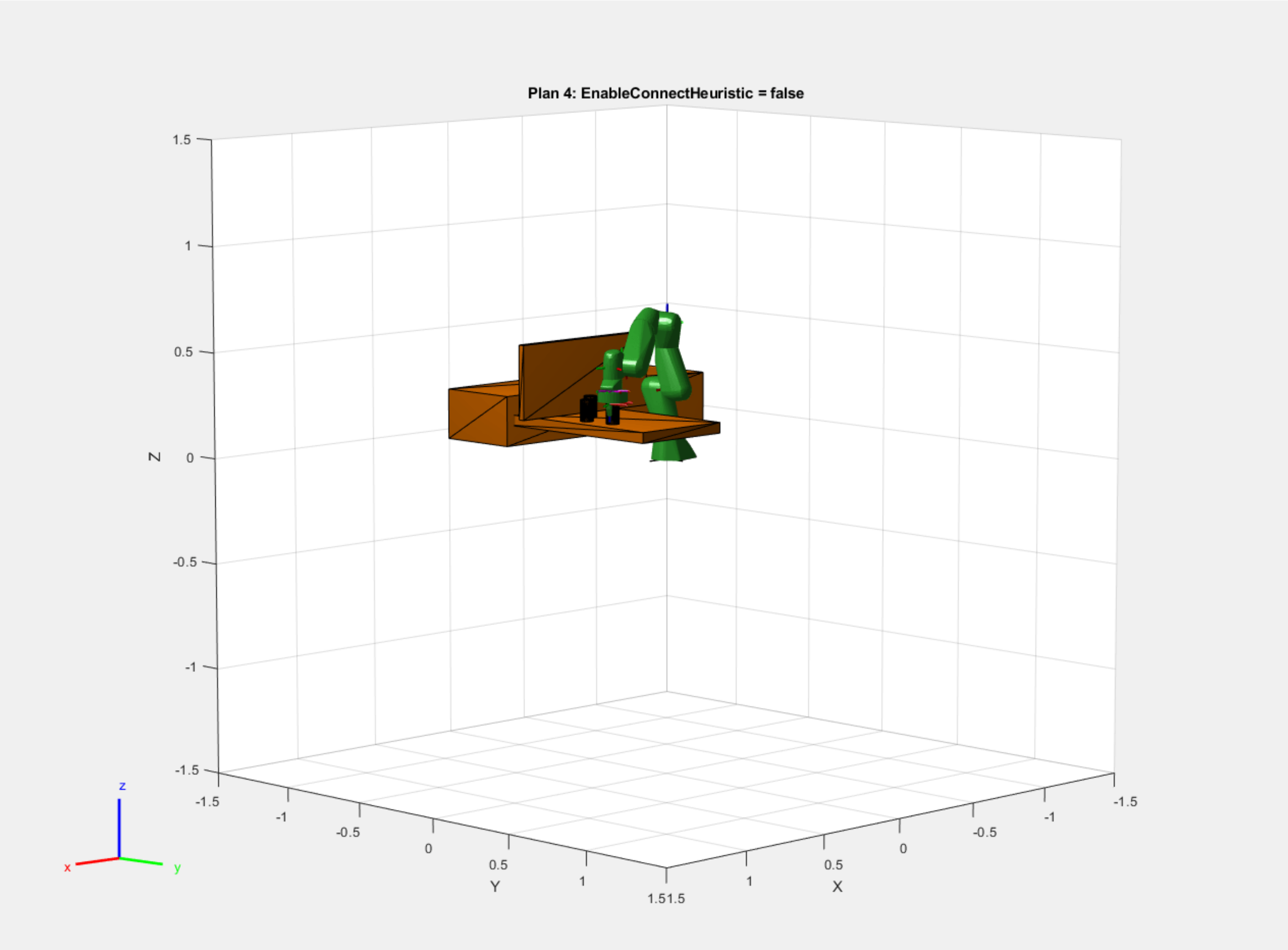


The connect heuristic allows the planner to greedily join the start and goal trees. In places where the environment is less cluttered, this heuristic is useful for shorter planning times. However, a greedy behavior in a cluttered environment leads to wasted connection attempts. Setting the `EnableConnectHeuristic` to `false` may give longer planning times and longer paths, but results in a higher success rate of finding a path given the number of iterations.

```
planner.ValidationDistance = 0.1;
planner.EnableConnectHeuristic = false;

tic
path = plan(planner,startConfig,goalConfig);
toc
```

```
Elapsed time is 2.241681 seconds.
```

```
interpStates = interpolate(planner,path);
for i = 1:2:size(interpStates,1)
    show(franka, interpStates(i,:), ...
        "PreservePlot",false,...
        "Visuals","off",...
        "Collisions","on");
    title("Plan 4: EnableConnectHeuristic = false")
    drawnow;
end
```



Attach the Can to the End-Effector

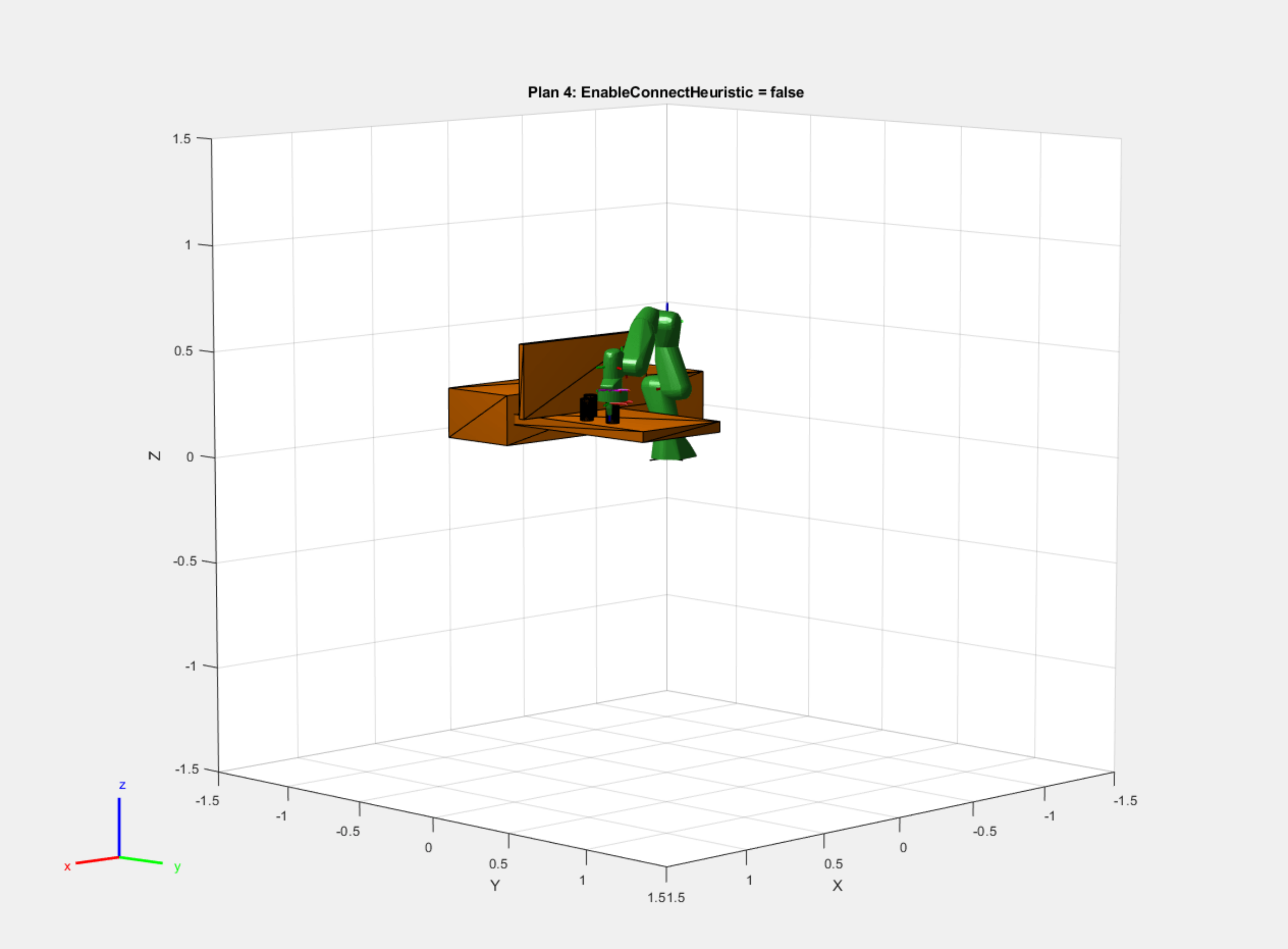
After tuning the planner for the desired behavior, follow the pick-and-place workflow where the robot moves an object through the environment. This example attaches a cylinder, or can, to the end-effector of the robot and moves it to a new location. For each configuration, the planner checks for collisions with the cylinder mesh as well.

```
% Create can as a rigid body
cylinder1 = env{3};
canBody = rigidBody("myCan");
canJoint = rigidBodyJoint("canJoint");

% Get current pose of the robot hand.
startConfig = path(end, :);
```

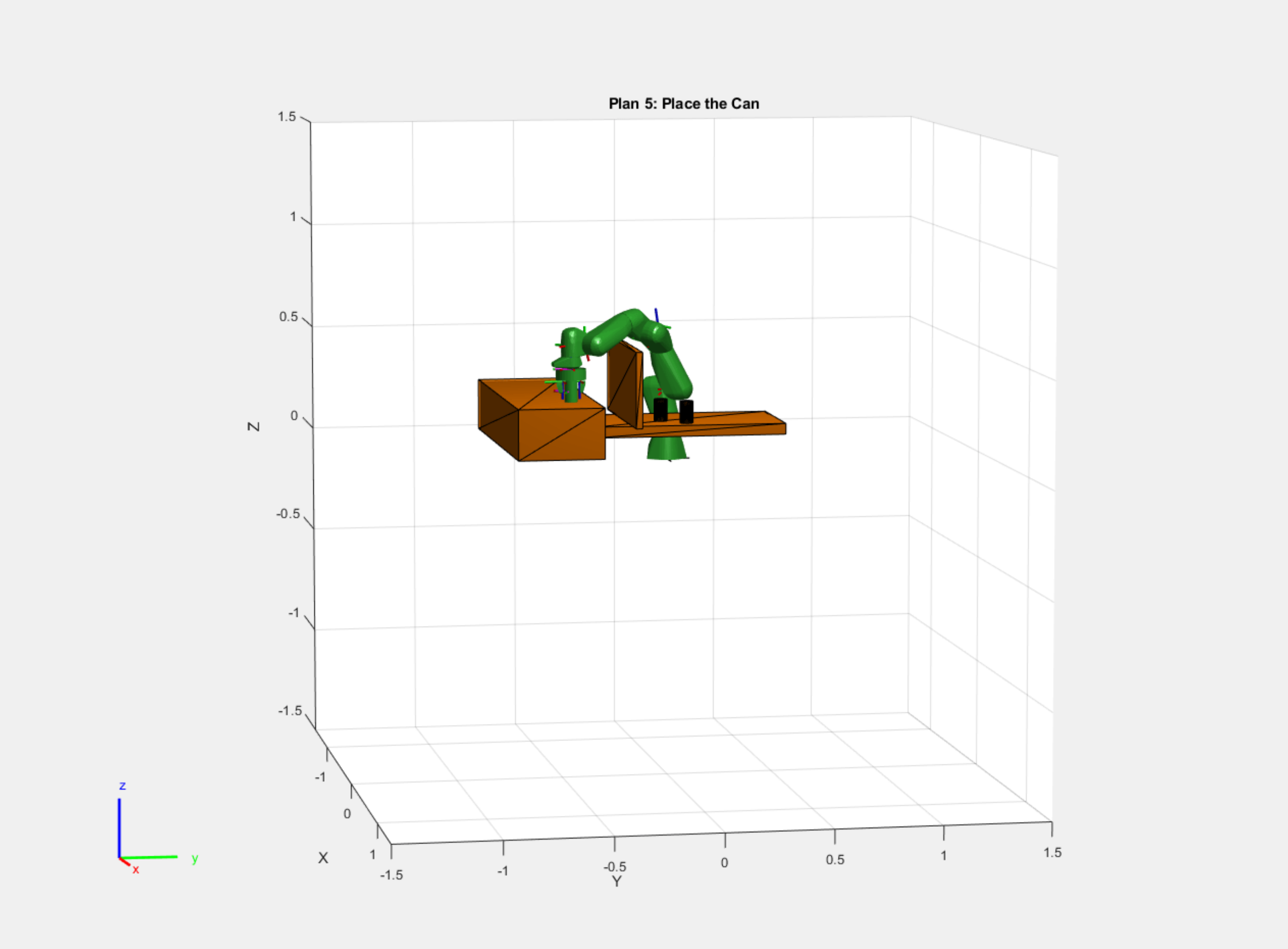
```
endEffectorPose = getTransform(franka,startConfig,"panda_hand");

% Place can into the end effector gripper.
setFixedTransform(canJoint,endEffectorPose\cylinder1.Pose);

% Add collision geometry to rigid body.
addCollision(canBody,cylinder1,inv(cylinder1.Pose));
canBody.Joint = canJoint;

% Add rigid body to robot model.
addBody(franka,canBody,"panda_hand");

% Remove object from environment.
env(3) = [];
```

After the can has been attached to the robot arm, specify a goal configuration for placing the object. Modify the planner parameters. Plan a path from start to goal. Visualize the path. Notice the can clears the wall.

```
goalConfig = [-0.6564 0.2885 -0.3187 -1.5941 0.1103 1.8678 -0.2344 0.04 0.04];

planner.MaxConnectionDistance = 1;
planner.ValidationDistance = 0.2;
planner.EnableConnectHeuristic = false;
path = plan(planner,startConfig,goalConfig);

interpStates = interpolate(planner,path);

hold off
```



```
show(franka,config,"Visuals","off","Collisions","on");
hold on
for i = 1:length(env)
    show(env{i});
end

for i = 1:size(interpStates,1)
    show(franka,interpStates(i,:),...
        "PreservePlot", false,...
        "Visuals","off",...
        "Collisions","on");
    title("Plan 5: Place the Can")
    drawnow;
    if i == (size(interpStates,1))
        view([80,7])
    end
end
```



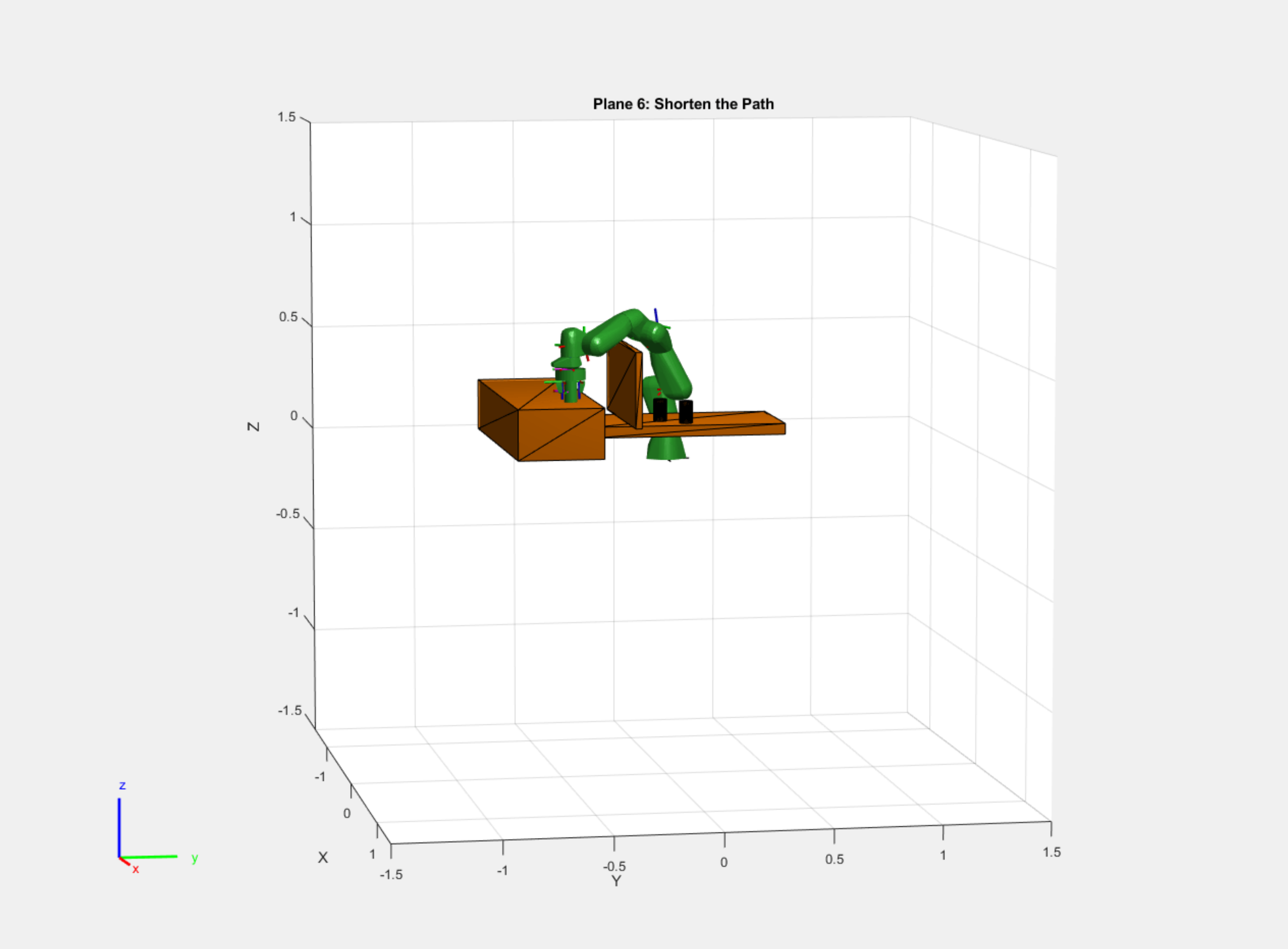
Shorten the Planned Path

To shorten your path, use the `shorten` function and specify a number of iterations. A small value for the `ValidationDistance` property combined with a large number of iterations can result in large computation times.

```
shortenedPath = shorten(planner,path,20);
```

```
interpStates = interpolate(planner,shortenedPath);
for i = 1:size(interpStates,1)
    show(franka, ...
        interpStates(i, :),  ...
        "PreservePlot", false, ...
        "Visuals", "off", ...
        "Collisions", "on");
    drawnow;
    title("Plane 6: Shorten the Path")
    if i > (size(interpStates,1)-2)
        view([80,7])
    end
```

```
end
hold off
```


Plane 6: Shorten the Path
1.5
1
0.5
0
-0.5
-1
-1.5
Z
-1
0
1
X
-1.5
-1
-0.5
0
0.5
1
1.5
Y
z
y
x

Pick-and-Place Workflow Using Stateflow for MATLAB

This example shows how to setup an end-to-end pick and place workflow for a robotic manipulator like the KINOVA® Gen3.

The pick-and-place workflow implemented in this example can be adapted to different scenarios, planners, simulation platforms, and object detection options. The example shown here uses Model Predictive Control for planning and control, and simulates the robot in MATLAB. For other uses, see:

- “Pick-and-Place Workflow Using RRT Planner and Stateflow for MATLAB” on page 1-294
- “Pick-and-Place Workflow in Gazebo Using ROS” on page 1-302
- “Pick-and-Place Workflow in Gazebo Using Point-Cloud Processing and RRT Path Planning” on page 1-311

Overview

This example sorts detected objects and places them on benches using a KINOVA Gen3 manipulator. The example uses tools from four toolboxes:

- **Robotics System Toolbox™** is used to model, simulate, and visualize the manipulator, and for collision-checking.
- **Model Predictive Control Toolbox™** and **Optimization Toolbox™** are used to generated optimized, collision-free trajectories for the manipulator to follow.
- **Stateflow®** is used to schedule the high-level tasks in the example and step from task to task.

This example builds on key concepts from two related examples:

- “Plan and Execute Task- and Joint-Space Trajectories Using KINOVA Gen3 Manipulator” on page 1-257 shows how to generate and simulate interpolated joint trajectories to move from an initial to a desired end-effector pose.
- “Plan and Execute Collision-Free Trajectories Using KINOVA Gen3 Manipulator” on page 1-261 shows how to plan closed-loop collision-free robot trajectories to a desired end-effector pose using nonlinear model predictive control.

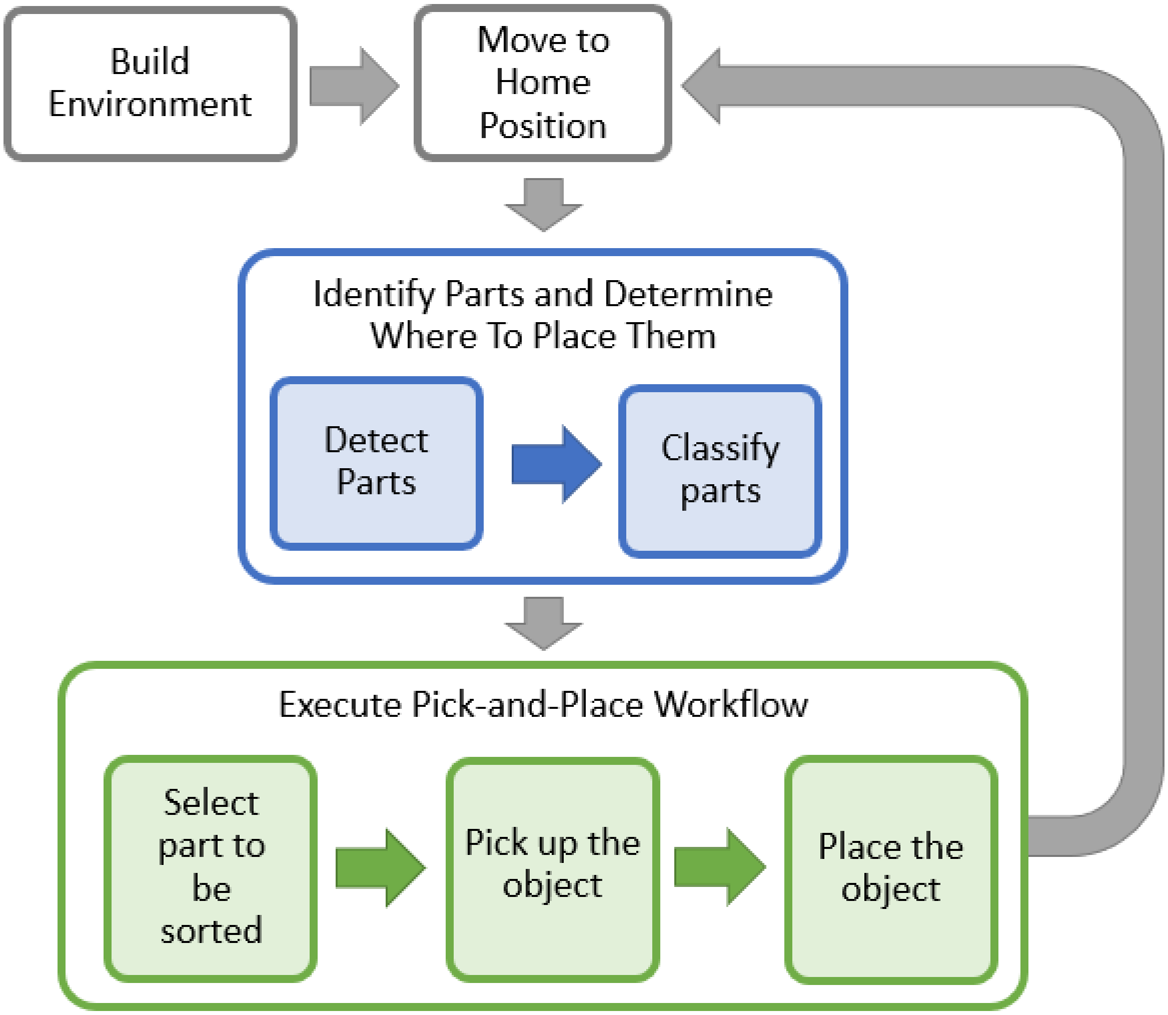
Stateflow Chart

This example uses a Stateflow chart to schedule tasks in the example. Open the chart to examine the contents and follow state transitions during chart execution.

```
edit exampleHelperFlowChartPickPlace.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 Pick-and-Place Workflow



Initialize the Robot and Environment

First, the chart creates an environment consisting of the Kinova Gen3 manipulator, three parts to be sorted, the shelves used for sorting, and a blue obstacle. Next, the robot moves to the home position.

Identify the Parts and Determine Where to Place Them

In the first step of the identification phase, the parts must be detected. The exampleCommand`DetectParts` function directly gives the object poses. Replace this class with your own object detection algorithm based on your sensors or objects.

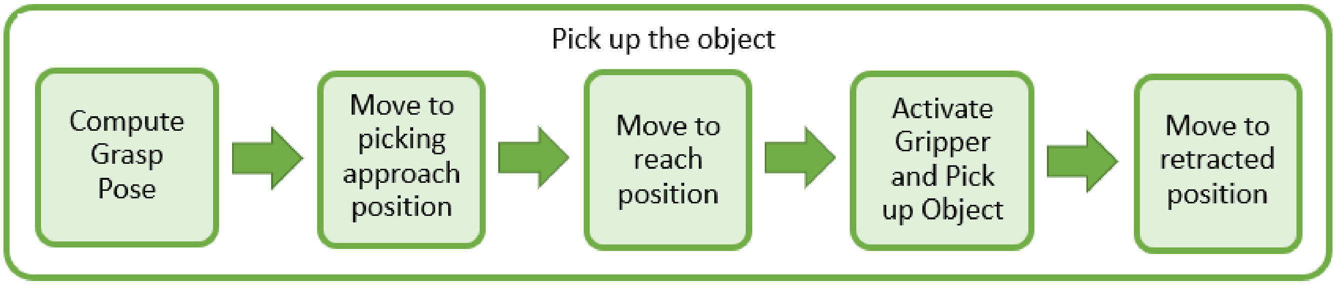
Next, the parts must be classified. The exampleCommand`ClassifyParts` function classifies the parts into two types to determine where to place them (top or bottom shelf). Again, you can replace this function with any method for classifying parts.

Execute Pick-and-Place Workflow

Once parts are identified and their destinations have been assigned, the manipulator must iterate through the parts and move them onto the appropriate tables.

Pick up the Object

The picking phase moves the robot to the object, picks it up, and moves to a safe position, as shown in the following diagram:

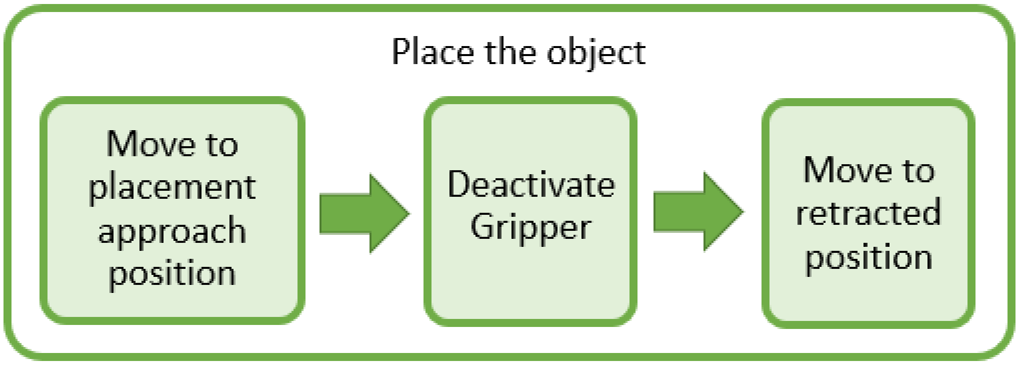

Pick up the object
Compute Grasp Pose
Move to picking approach position
Move to reach position
Activate Gripper and Pick up Object
Move to retracted position

The exampleCommand`ComputeGraspPose` function computes the grasp pose. The class computes a task-space grasping position for each part. Intermediate steps for approaching and reaching towards the part are also defined relative to the object.

This robot picks up objects using a simulated gripper. When the gripper is activated, exampleCommand`ActivateGripper` adds the collision mesh for the part onto the `rigidBodyTree` representation of the robot, which simulates grabbing it. Collision detection includes this object while it is attached. Then, the robot moves to a retracted position away from the other parts.

Place the Object

The robot then places the object on the appropriate shelf.


Place the object
Move to placement approach position
Deactivate Gripper
Move to retracted position

As with the picking workflow, the placement approach and retracted positions are computed relative to the known desired placement position. The gripper is deactivated using exampleCommand`ActivateGripper`, which removes the part from the robot.

Moving the Manipulator to a Specified Pose

Most of the task execution consists of instructing the robot to move between different specified poses. The `exampleHelperPlanExecuteTrajectoryPickPlace` function defines a solver using a nonlinear model predictive controller (see “Nonlinear MPC” (Model Predictive Control Toolbox)) that computes a feasible, collision-free optimized reference trajectory using `nlmpcmove` (Model Predictive Control Toolbox) and `checkCollision`. The obstacles are represented as spheres to ensure the accurate approximation of the constraint Jacobian in the definiton of the nonlinear model predictive control algorithm (see [1]). The helper function then simulates the motion of the manipulator under computed-torque control as it tracks the reference trajectory using the `jointSpaceMotionModel` object, and updates the visualization. The helper function is called from the Stateflow chart via `exampleCommandMoveToTaskConfig`, which defines the correct inputs.

This workflow is examined in detail in “Plan and Execute Collision-Free Trajectories Using KINOVA Gen3 Manipulator” on page 1-261. The controller is used to ensure collision-free motion. For simpler trajectories where the paths are known to be obstacle-free, trajectories could be executed using trajectory generation tools and simulated using the manipulator motion models. See “Plan and Execute Task- and Joint-Space Trajectories Using KINOVA Gen3 Manipulator” on page 1-257.

Task Scheduling in a Stateflow Chart

This example uses a Stateflow chart to direct the workflow in MATLAB®. For more info on creating state flow charts, see “Create Stateflow Charts for Execution as MATLAB Objects” (Stateflow).

The Stateflow chart directs task execution in MATLAB by using command functions. When the command finishes executing, it sends an *input event* to wake up the chart and proceed to the next step of the task execution, see “Execute a Standalone Chart” (Stateflow).

Run and Visualize the Simulation

This simulation uses a KINOVA Gen3 manipulator with a Robotiq gripper. Load the robot model from a `.mat` file as a `rigidBodyTree` object.

```
load('exampleHelperKINOVAGen3GripperColl.mat');
```

Initialize the Pick and Place Coordinator

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

```
currentRobotJConfig = homeConfiguration(robot);
coordinator = exampleHelperCoordinatorPickPlace(robot,currentRobotJConfig, "gripper");
```

Specify the home configuration and two poses for placing objects of different types.

```
coordinator.HomeRobotTaskConfig = trvec2tform([0.4, 0, 0.6])*axang2tform([0 1 0 pi]);
coordinator.PlacingPose{1} = trvec2tform([0.23 0.62 0.33])*axang2tform([0 1 0 pi]);
coordinator.PlacingPose{2} = trvec2tform([0.23 -0.62 0.33])*axang2tform([0 1 0 pi]);
```

Run and Visualize the Simulation

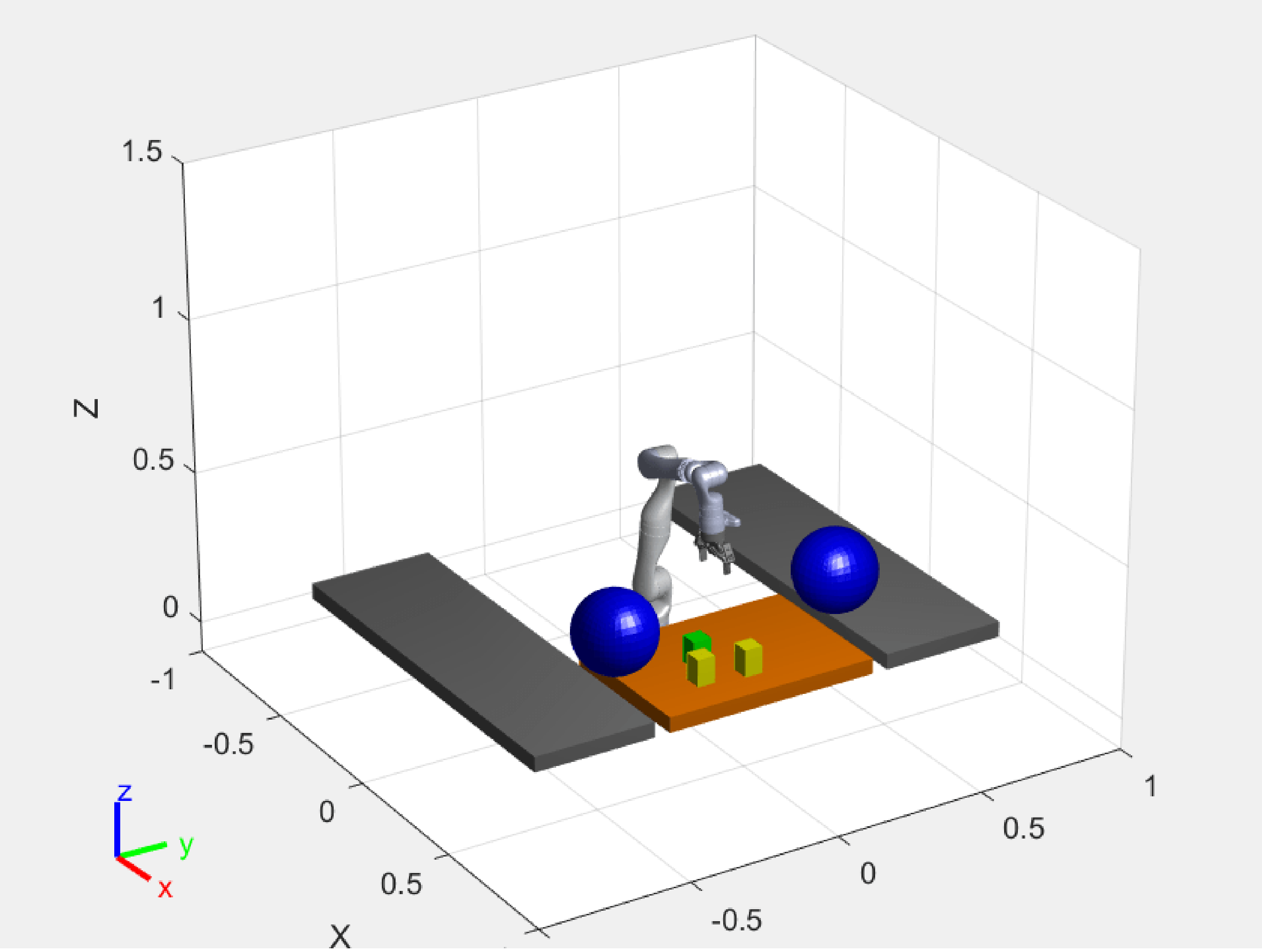
Connect the coordinator to the Stateflow Chart. Once started, the Stateflow chart is responsible for continuously going through the states of detecting objects, picking them up and placing them in the correct staging area.

```
coordinator.FlowChart = exampleHelperFlowChartPickPlace('coordinator', coordinator);
```

Use a dialog to start the pick-and-place task execution. Click **Yes** in the dialog to begin the simulation.

```
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'
            % End Pick and Place
            coordinator.FlowChart.endPickPlace;
            delete(coordinator.FlowChart);
            delete(coordinator);
    end
```



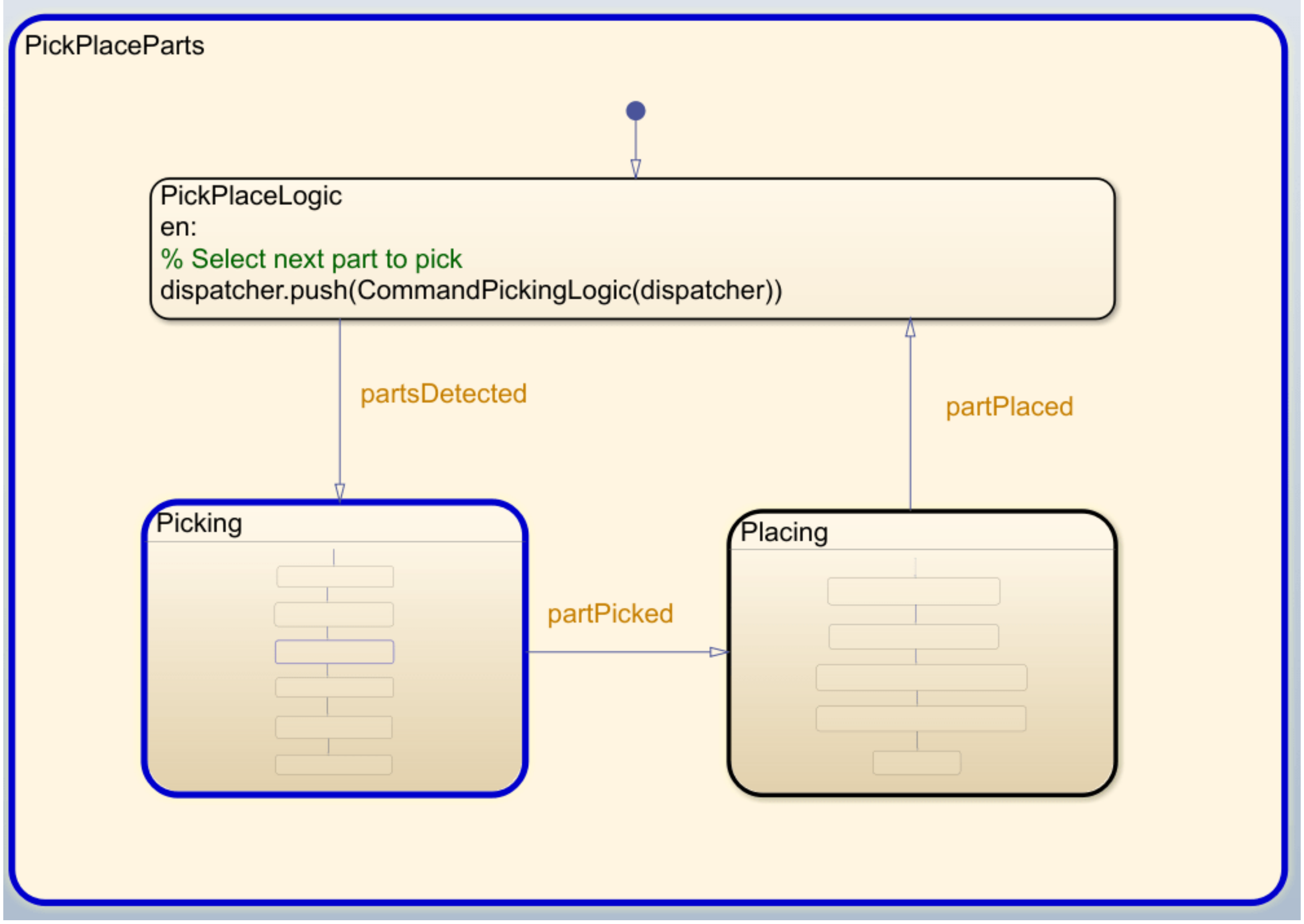
Ending the Pick-and-Place task

The Stateflow chart will finish executing automatically after 3 failed attempts to detect new objects. To end the pick-and-place task prematurely, uncomment and execute the following lines of code or press Ctrl+C in the command window.

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

Observe the Simulation States

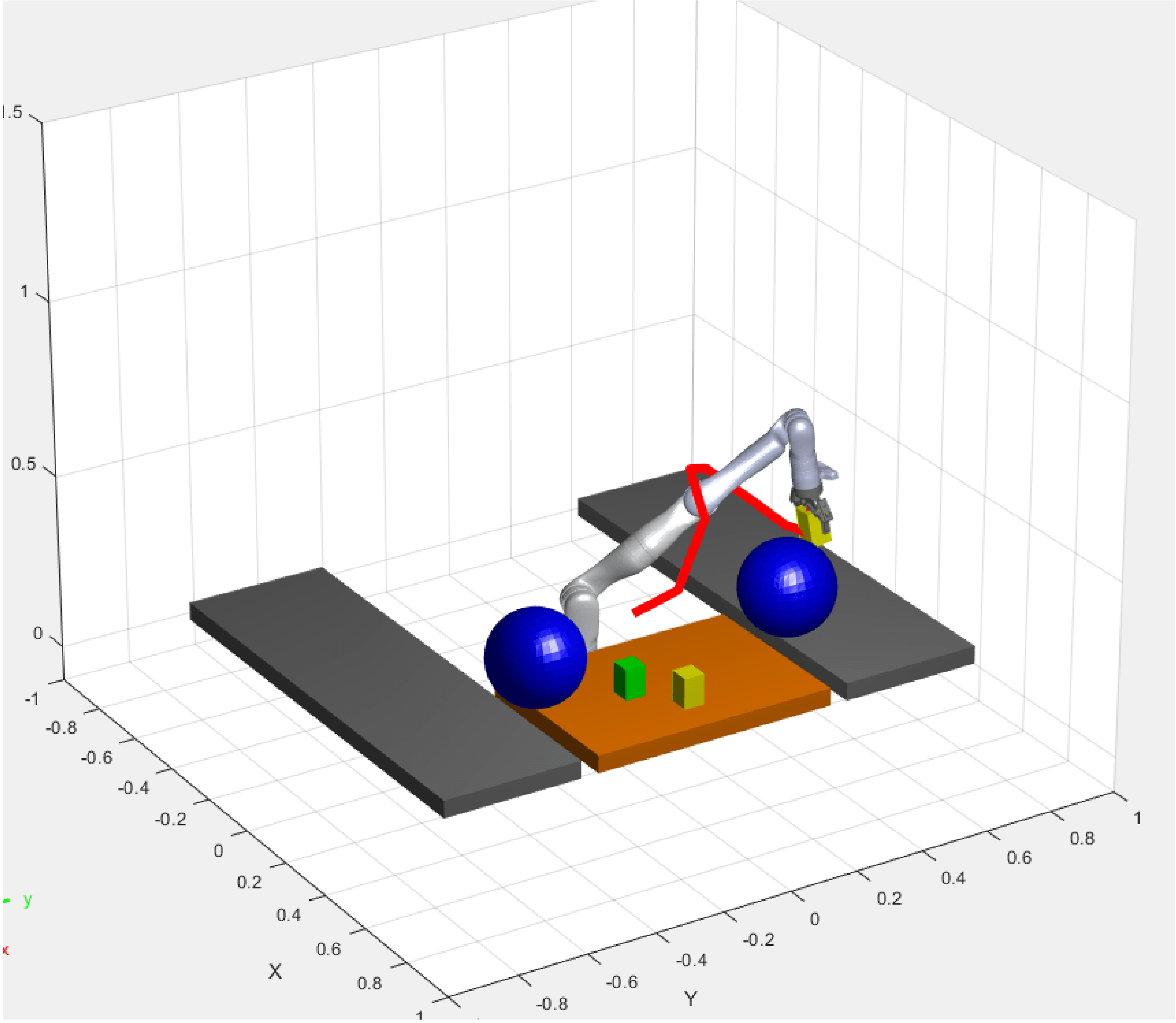
During execution, the active states at each point in time are highlighted in blue in the Stateflow chart. This helps keeping track of what the robot does and when. You can click through the subsystems to see the details of the state in action.

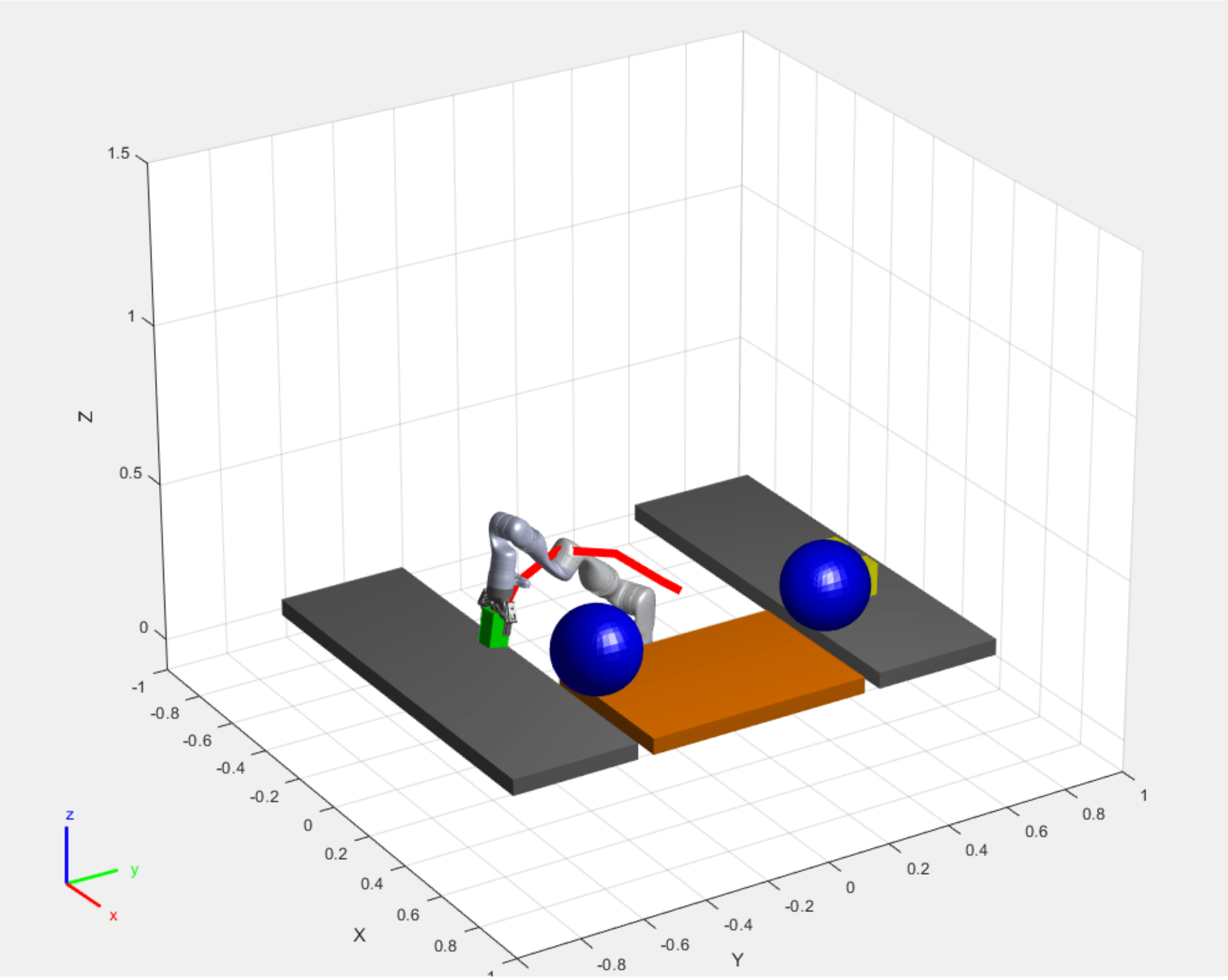


```
PickPlaceLogic
en:
% Select next part to pick
dispatcher.push(CommandPickingLogic(dispatcher))
```

Visualize the Pick-and-Place Action

The example uses `interactiveRigidBodyTree` for robot visualization. The visualization shows the robot in the working area as it moves parts around. The robot avoids obstacles in the environment (blue cylinder) and places objects on top or bottom shelf based on their classification. The robot continues working until all parts have been placed.


1.5
1
0.5
0
-1
-0.8
-0.6
-0.4
-0.2
0
0.2
0.4
0.6
0.8
1
X
-0.8
-0.6
-0.4
-0.2
0
0.2
0.4
0.6
0.8
1
Y
y
x



References

[1] Schulman, J., et al. "Motion planning with sequential convex optimization and convex collision checking." *The International Journal of Robotics Research* 33.9 (2014): 1251-1270.

Pick-and-Place Workflow Using RRT Planner and Stateflow for MATLAB

This example shows how to setup an end-to-end pick-and-place workflow for a robotic manipulator like the KINOVA® Gen3.

The pick-and-place workflow implemented in this example can be adapted to different scenarios, planners, simulation platforms, and object detection options. The example shown here uses the rapidly-exploring random tree (RRT) algorithm for planning, and simulates the robot in MATLAB. For other uses, see:

- “Pick-and-Place Workflow Using Stateflow for MATLAB” on page 1-286
- “Pick-and-Place Workflow in Gazebo Using ROS” on page 1-302
- “Pick-and-Place Workflow in Gazebo Using Point-Cloud Processing and RRT Path Planning” on page 1-311

Overview

This example sorts detected objects onto shelves using a KINOVA Gen3 manipulator. The example uses tools from two toolboxes:

- **Robotics System Toolbox™** is used to model, simulate, and visualize the manipulator, and plan collision-free paths for the manipulator to follow using RRT.
- **Stateflow®** is used to schedule the high-level tasks in the example and step from task to task.

This example uses the RRT algorithm for path planning. For another example that goes into more details about the RRT planner, see “Pick and Place Using RRT for Manipulators” on page 1-275.

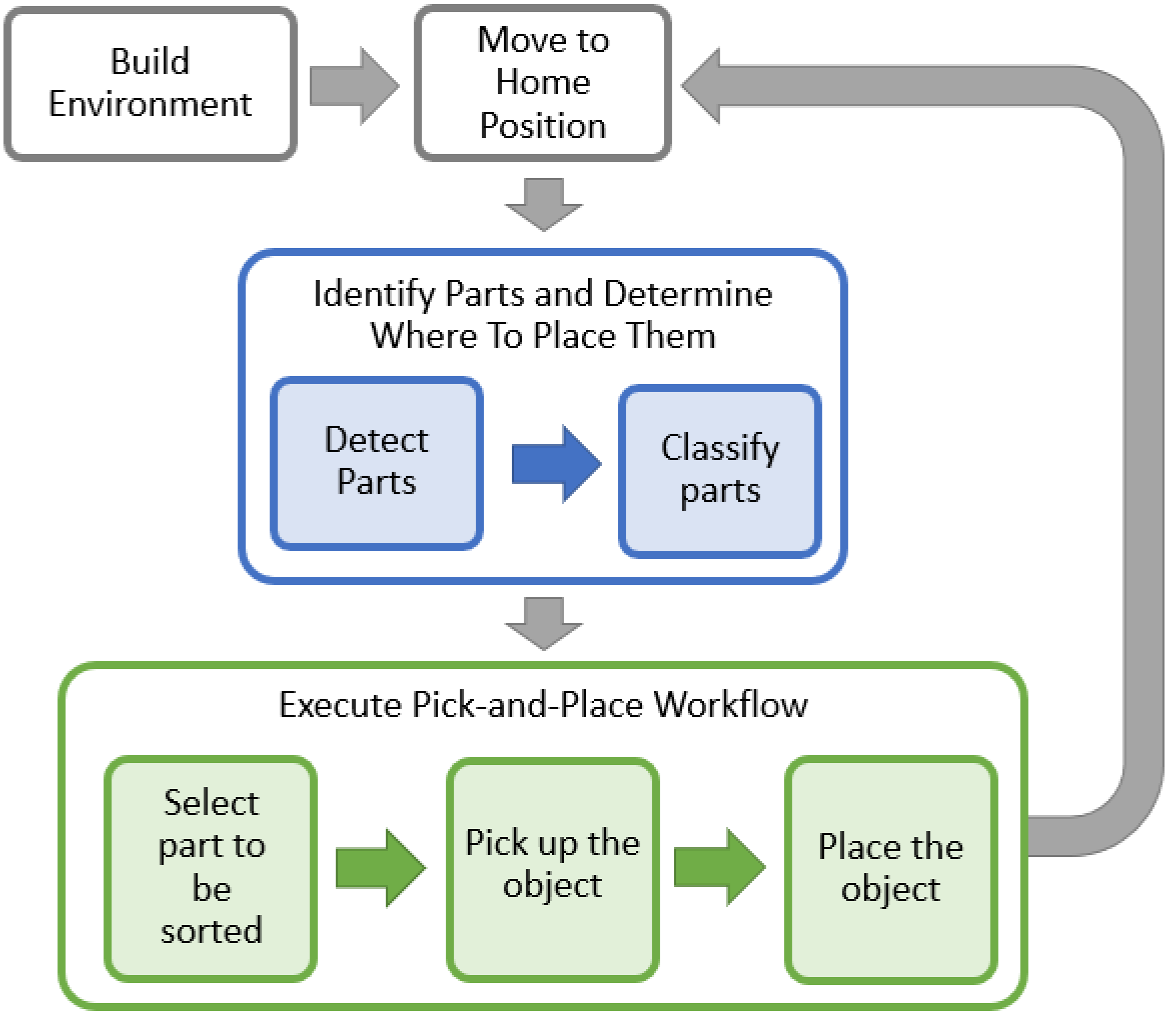
Stateflow Chart

This example uses a Stateflow chart to schedule tasks in the example. Open the chart to examine the contents and follow state transitions during chart execution.

```
edit exampleHelperFlowChartPickPlaceRRT.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 Pick-and-Place Workflow



Initialize the Robot and Environment

First, the chart creates an environment consisting of the Kinova Gen3 manipulator, three parts to be sorted, the shelves used for sorting, and a blue obstacle. Next, the robot moves to the home position.

Identify the Parts and Determine Where to Place Them

In the first step of the identification phase, the parts must be detected. The `exampleCommandDetectParts` function directly gives the object poses. Replace this class with your own object detection algorithm based on your sensors or objects.

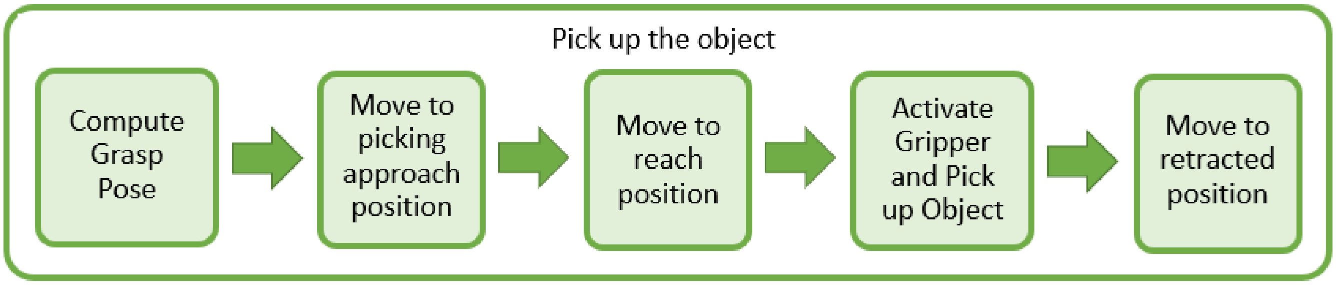
Next, the parts must be classified. The `exampleCommandClassifyParts` function classifies the parts into two types to determine where to place them (top or bottom shelf). Again, you can replace this function with any method for classifying parts.

Execute Pick-and-Place Workflow

Once parts are identified and their destinations have been assigned, the manipulator must iterate through the parts and move them onto the appropriate tables.

Pick up the Object

The picking phase moves the robot to the object, picks it up, and moves to a safe position, as shown in the following diagram:

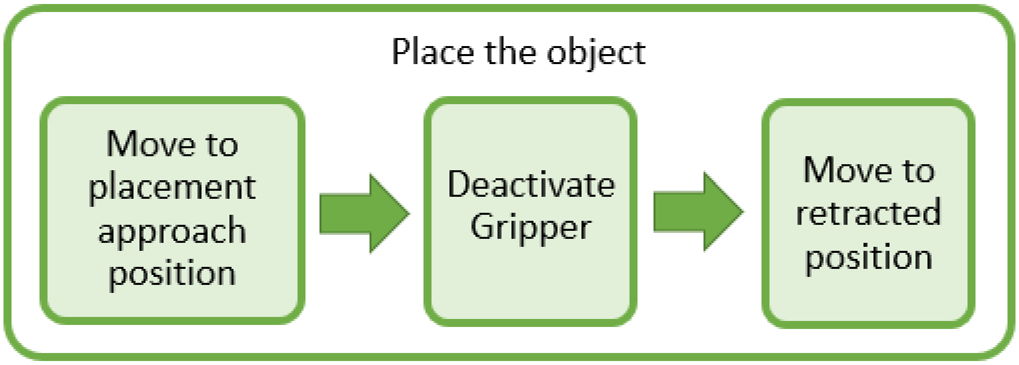

Pick up the object
Compute Grasp Pose
Move to picking approach position
Move to reach position
Activate Gripper and Pick up Object
Move to retracted position

The exampleCommandComputeGraspPose function computes the grasp pose. The class computes a task-space grasping position for each part. Intermediate steps for approaching and reaching towards the part are also defined relative to the object.

This robot picks up objects using a simulated pneumatic gripper. When the gripper is activated, exampleCommandActivateGripper adds the collision mesh for the part onto the rigidBodyTree representation of the robot, by using the addCollision object function. Collision detection includes this object while it is attached. Then, the robot moves to a retracted position away from the other parts.

Place the Object

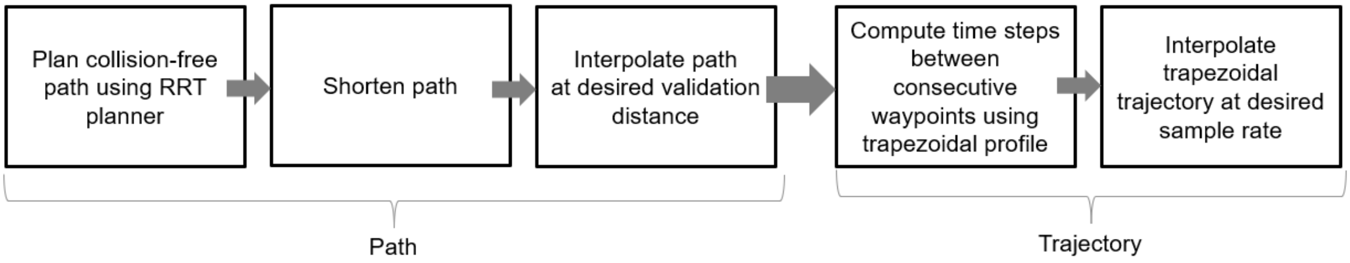
The robot then places the object on the appropriate shelf.


Place the object
Move to placement approach position
Deactivate Gripper
Move to retracted position

As with the picking workflow, the placement approach and retracted positions are computed relative to the known desired placement position. The gripper is deactivated using exampleCommandActivateGripper, which removes the part from the robot, using clearCollision. Every time a part of specific type is placed, the placing pose for this object type is updated so that next part of same type is placed next to the placed part.

Moving the Manipulator to a Specified Pose

Most of the task execution consists of instructing the robot to move between different specified poses. The exampleHelperMoveToTaskConfig function defines an RRT planner using the manipulatorRRT object, which plans paths from an initial to a desired joint configuration by avoiding collisions with specified collision objects in the scene. The resulting path is first shortened and then interpolated at a desired validation distance. To generate a trajectory, the `trapveltraj` function is used to assign time steps to each of the interpolated waypoints following a trapezoidal profile. Finally, the waypoints with their associated times are interpolated to a desired sample rate (every 0.1 seconds). The generated trajectories ensure that the robot moves slowly at the start and the end of the motion when it is approaching or placing an object.


Plan collision-free path using RRT planner
Shorten path
Interpolate path at desired validation distance
Compute time steps between consecutive waypoints using trapezoidal profile
Interpolate trapezoidal trajectory at desired sample rate
Path
Trajectory

For another example that goes into more details about the RRT planner, see “Pick and Place Using RRT for Manipulators” on page 1-275.

For simpler trajectories where the paths are known to be obstacle-free, trajectories could be executed using trajectory generation tools and simulated using the manipulator motion models. See the “Plan and Execute Task- and Joint-Space Trajectories Using KINOVA Gen3 Manipulator” on page 1-257 example.

Task Scheduling in a Stateflow Chart

This example uses a Stateflow chart to direct the workflow in MATLAB®. For more info on creating state flow starts, see “Create Stateflow Charts for Execution as MATLAB Objects” (Stateflow).

The Stateflow chart directs task execution in MATLAB by using command functions. When the command finishes executing, it sends an *input event* to wake up the chart and proceed to the next step of the task execution, see “Execute a Standalone Chart” (Stateflow).

Run and Visualize the Simulation

This simulation uses a KINOVA Gen3 manipulator with a Robotiq gripper. Load the robot model from a `.mat` file as a `rigidBodyTree` object.

```
load('exampleHelperKINOVAGen3GripperCollRRT.mat');
```

Initialize the Pick and Place Coordinator

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

```
currentRobotJConfig = homeConfiguration(robot);
coordinator = exampleHelperCoordinatorPickPlaceRRT(robot,currentRobotJConfig, "gripper");
```

Specify the home configuration and two poses for placing objects of two different types. The first pose corresponds to the middle shelf where all parts of type 1 are placed and the second pose corresponds to the top shelf where parts of type 2 are placed. The placing poses are updated in the Stateflow chart every time a new part is placed successfully. Parts of different type are identified by different colors in the vizualization.

```
coordinator.HomeRobotTaskConfig = trvec2tform([0.4, 0, 0.5])*axang2tform([0 1 0 pi]);
coordinator.PlacingPose{1} = trvec2tform([[-0.15 0.52 0.46]])*axang2tform([1 0 0 -pi/2]);
coordinator.PlacingPose{2} = trvec2tform([[-0.15 0.52 0.63]])*axang2tform([1 0 0 -pi/2]);
```

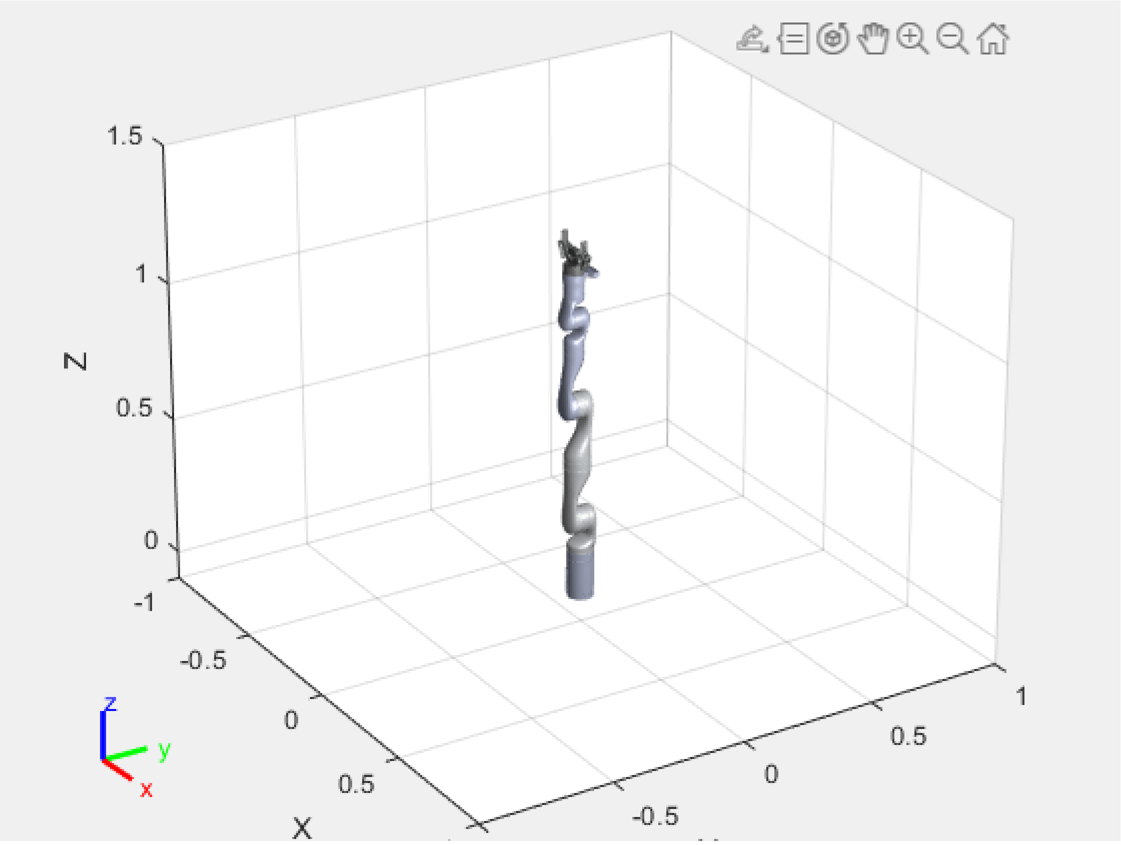
Run and Visualize the Simulation

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 = exampleHelperFlowChartPickPlaceRRT('coordinator', coordinator);
```

Use a dialog to start the pick-and-place task execution. Click **Yes** in the dialog to begin the simulation.

```
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'
        % End Pick and Place
        coordinator.FlowChart.endPickPlace;
        delete(coordinator.FlowChart);
        delete(coordinator);
end
```



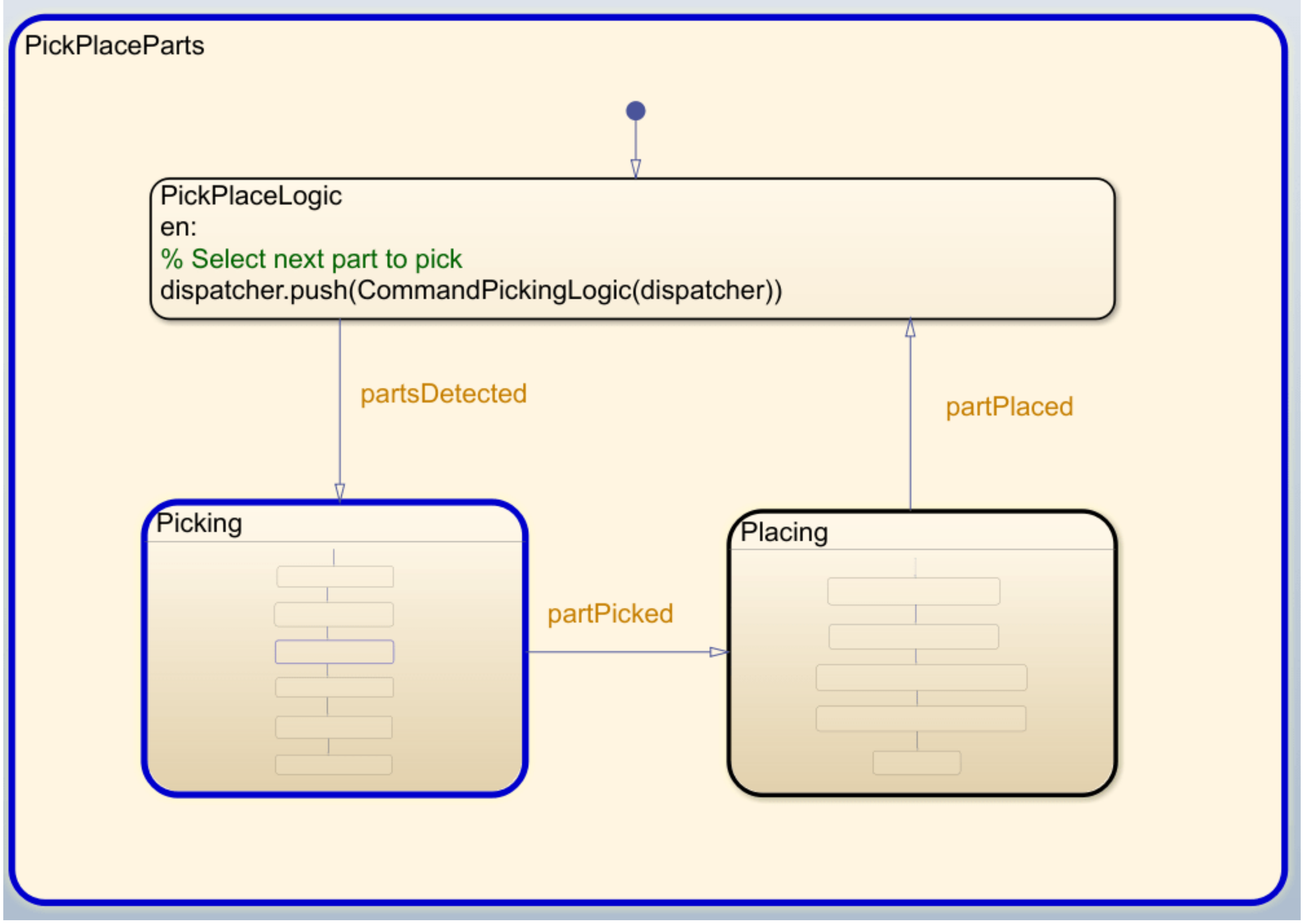
Ending the Pick-and-Place task

The Stateflow chart will finish executing automatically after 3 failed attempts to detect new objects. To end the pick-and-place task prematurely, uncomment and execute the following lines of code or press Ctrl+C in the command window.

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

Observe the Simulation States

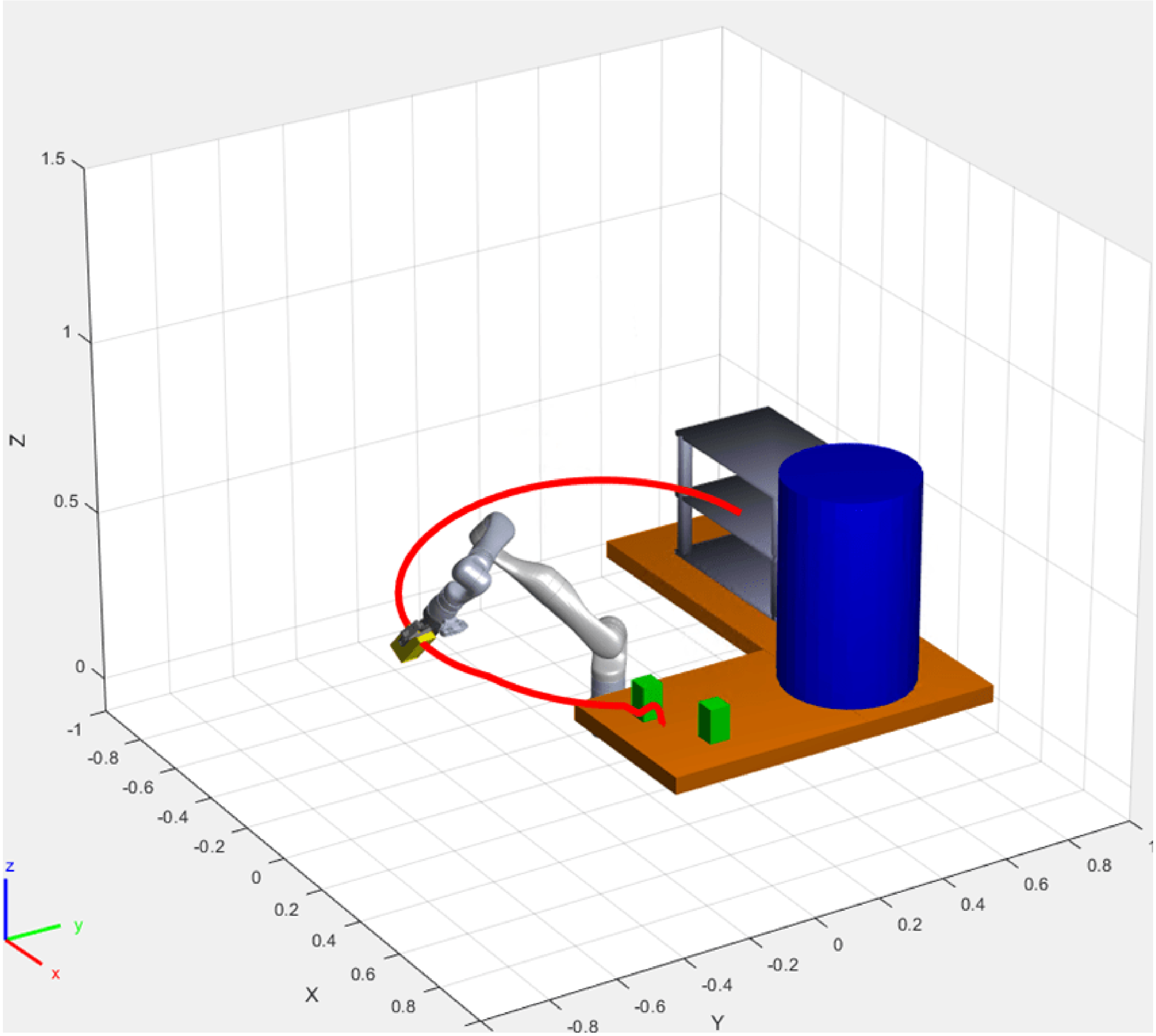
During execution, the active states at each point in time are highlighted in blue in the Stateflow chart. This helps keeping track of what the robot does and when. You can click through the subsystems to see the details of the state in action.

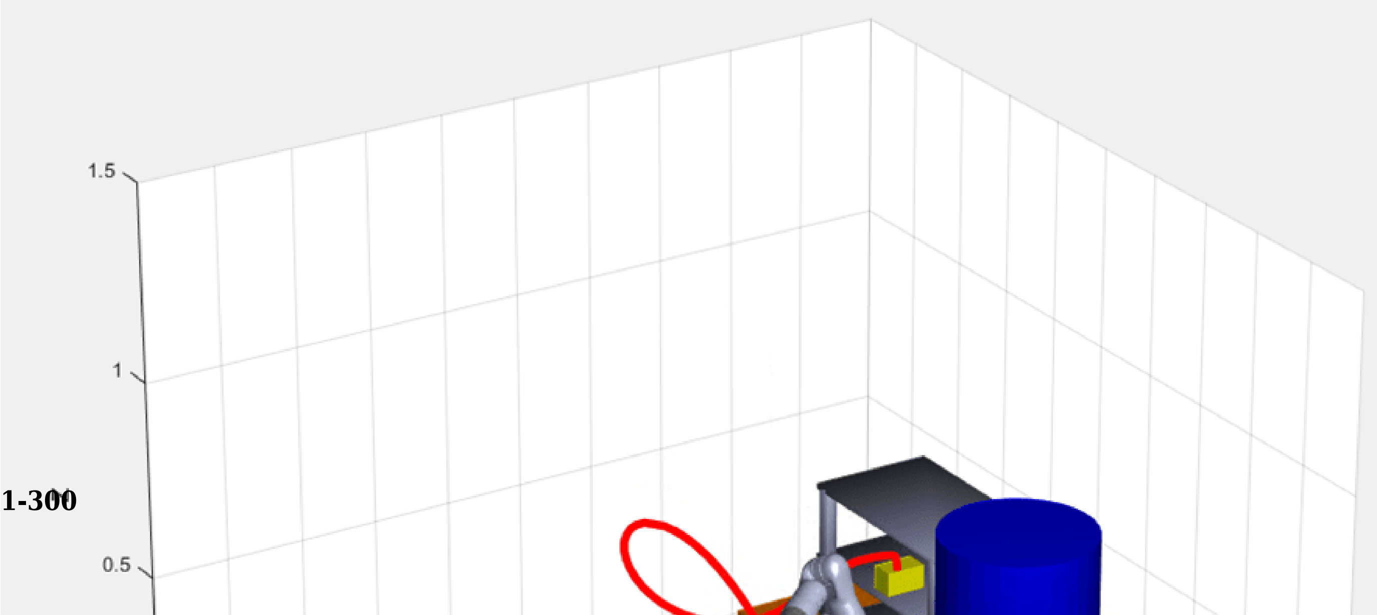


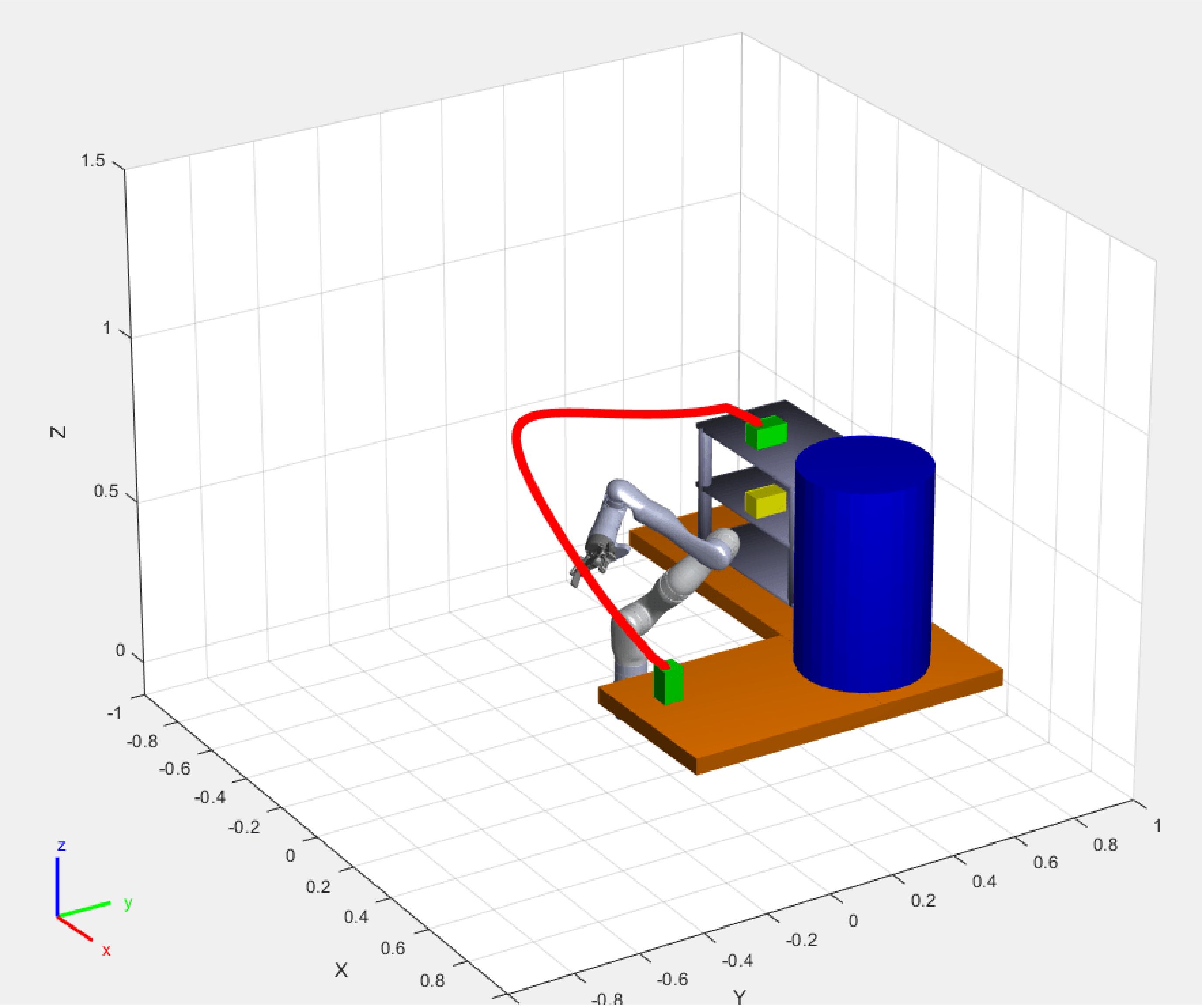
```
PickPlaceLogic
en:
% Select next part to pick
dispatcher.push(CommandPickingLogic(dispatcher))
```

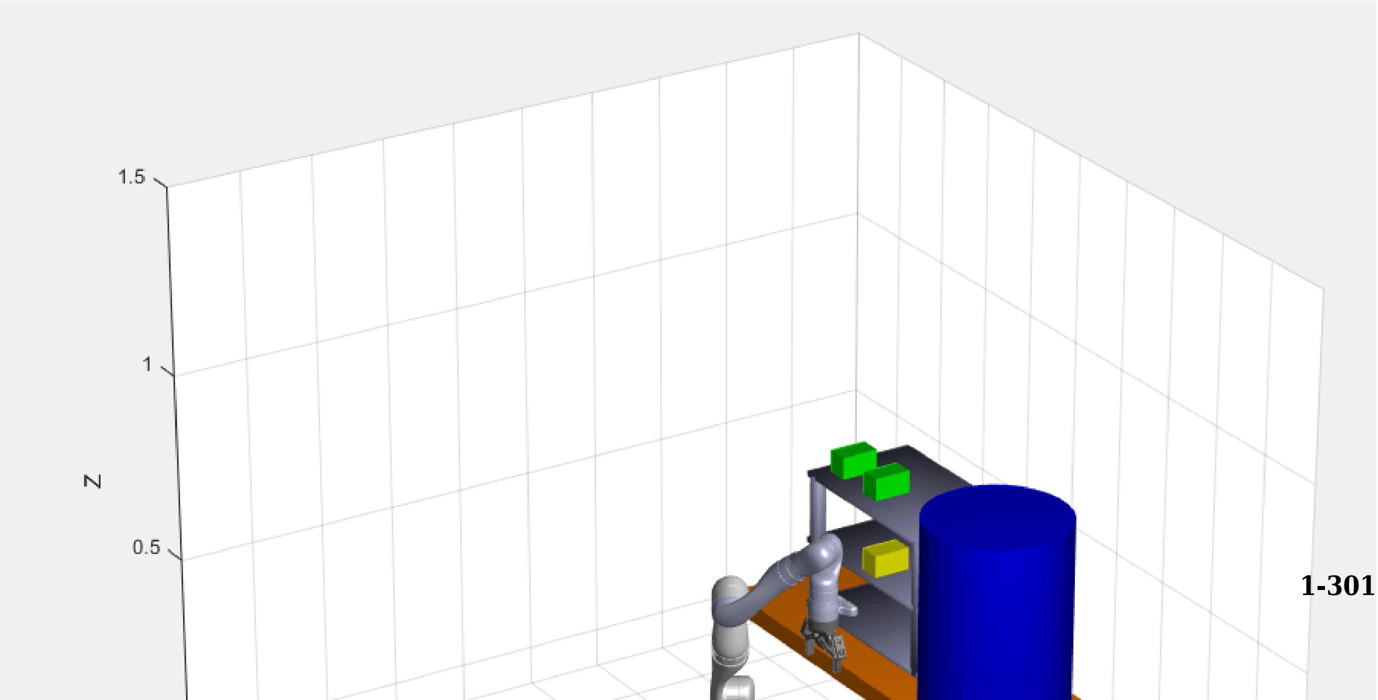
Visualize the Pick-and-Place Action

The example uses the `interactiveRigidBodyTree` object for robot visualization. The visualization shows the robot in the working area as it moves parts around. The robot avoids obstacles in the environment (blue cylinder) and places objects on the top or bottom shelf based on their classification. The robot continues working until all parts have been placed.


1.5
1
Z
0.5
0
-1
-0.8
-0.6
-0.4
-0.2
0
0.2
0.4
0.6
0.8
X
z
y
x
-0.8
-0.6
-0.4
-0.2
0
0.2
0.4
0.6
0.8
1
Y


1.5
1
0.5


1.5
1
Z
0.5
0
-1
-0.8
-0.6
-0.4
-0.2
0
0.2
0.4
0.6
X
0.8
-0.8
-0.6
-0.4
Y
-0.2
0
0.2
0.4
0.6
0.8
1
z
y
x


1.5
1
Z
0.5

Pick-and-Place Workflow in Gazebo Using ROS

This example shows how to setup 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.

Overview

This example identifies and recycles objects into two bins using a KINOVA Gen3 manipulator. The example uses tools from five toolboxes:

- **Robotics System Toolbox™** is used to model and simulate the manipulator.
- **Stateflow®** is used to schedule the high-level tasks in the example and step from task to task.
- **ROS Toolbox™** is used for connecting MATLAB to Gazebo.
- **Computer Vision Toolbox™** and **Deep Learning Toolbox™** are used for object detection using simulated camera in Gazebo.

This example builds on key concepts from the following related examples:

- "Plan and Execute Task- and Joint-Space Trajectories Using KINOVA Gen3 Manipulator" on page 1-257 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" on page 1-286
- Computer Vision Toolbox example: "Train YOLO v2 Network for Vehicle Detection" (Computer Vision Toolbox)
- ROS Toolbox example: "Get Started with Gazebo and Simulated TurtleBot" (ROS Toolbox)

Robot Simulation and Control in Gazebo

Start a ROS-based simulator for a KINOVA Gen3 robot and configure the MATLAB® connection with the robot simulator.

This example uses a virtual machine (VM) containing ROS Melodic available for download here. If you have never used it before, see "Get Started with Gazebo and Simulated TurtleBot" (ROS Toolbox).

- In VM settings, **VM > Settings > Hardware > Display**, disable **Accelerate 3D graphics**.
- Start the Ubuntu® virtual machine desktop.
- In the Ubuntu desktop, click the **Gazebo Recycling World** icon to start the Gazebo world built for this example.
- Specify the IP address and port number of the ROS master in Gazebo so that MATLAB® can communicate with the robot simulator. For this example, the ROS master in Gazebo uses the IP address of `192.168.203.131` displayed on the Desktop. Adjust the `rosIP` variable based on your VM.
- Start the ROS 1 network using `rosinit`.

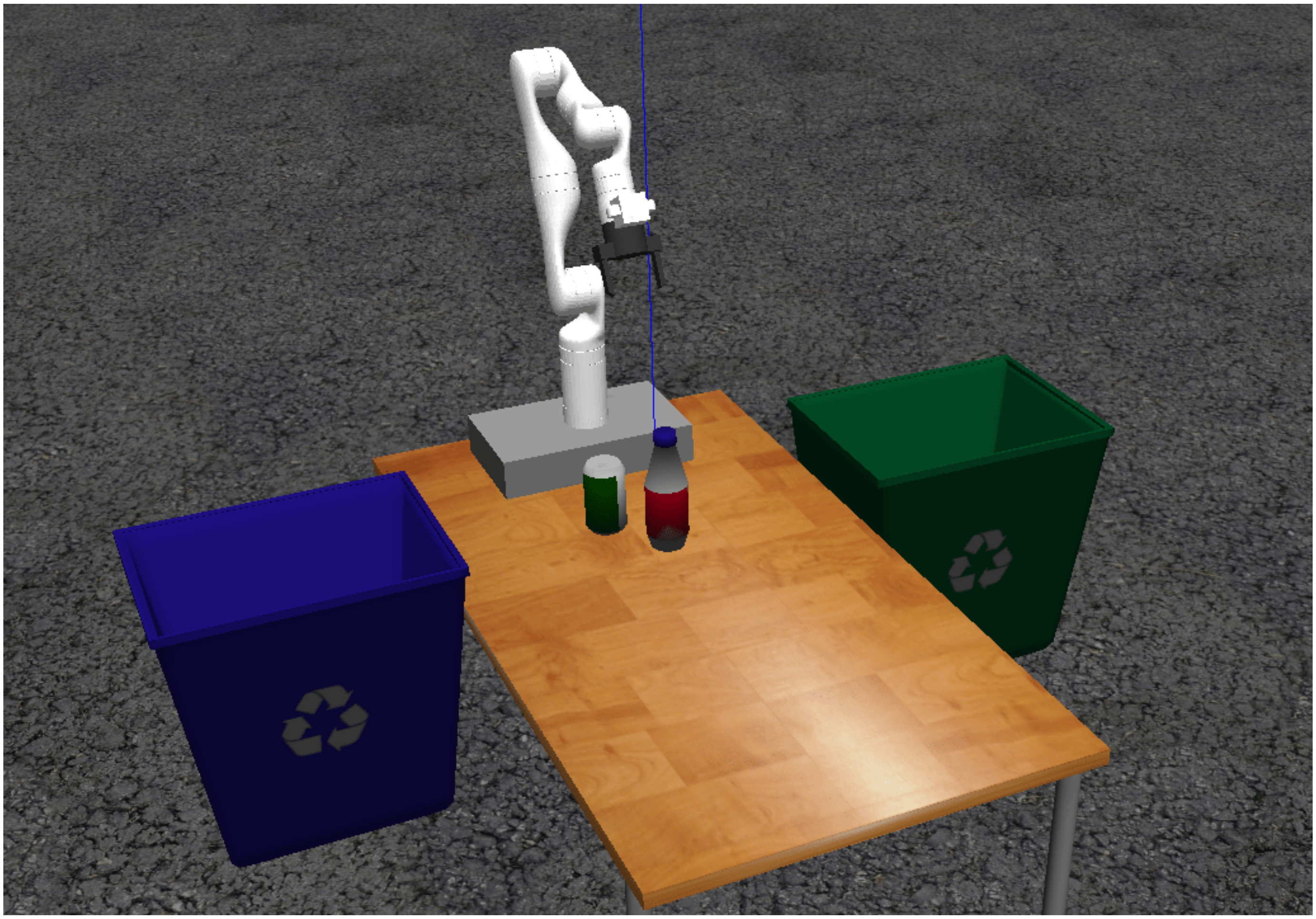
```
rosIP = '192.168.203.131';   % IP address of ROS-enabled machine
```

```
rosinit(rosIP,11311); % Initialize ROS connection
```

```
The value of the ROS_IP environment variable, 192.168.31.1, will be used to set the advertised ad
Initializing global node /matlab_global_node_36570 with NodeURI http://192.168.31.1:51073/
```

After initializing the Gazebo world by click the icon, the VM loads a KINOVA Gen3 Robot arm on a table with one recycling bin on each side. To simulate and control the robot arm in Gazebo, the VM contains the ros_kortex ROS package, which are provided by KINOVA.

The packages use ros_control to control the joints to desired joint positions. For additional details on using the VM, refer to "Get Started with Gazebo and Simulated TurtleBot" (ROS Toolbox)



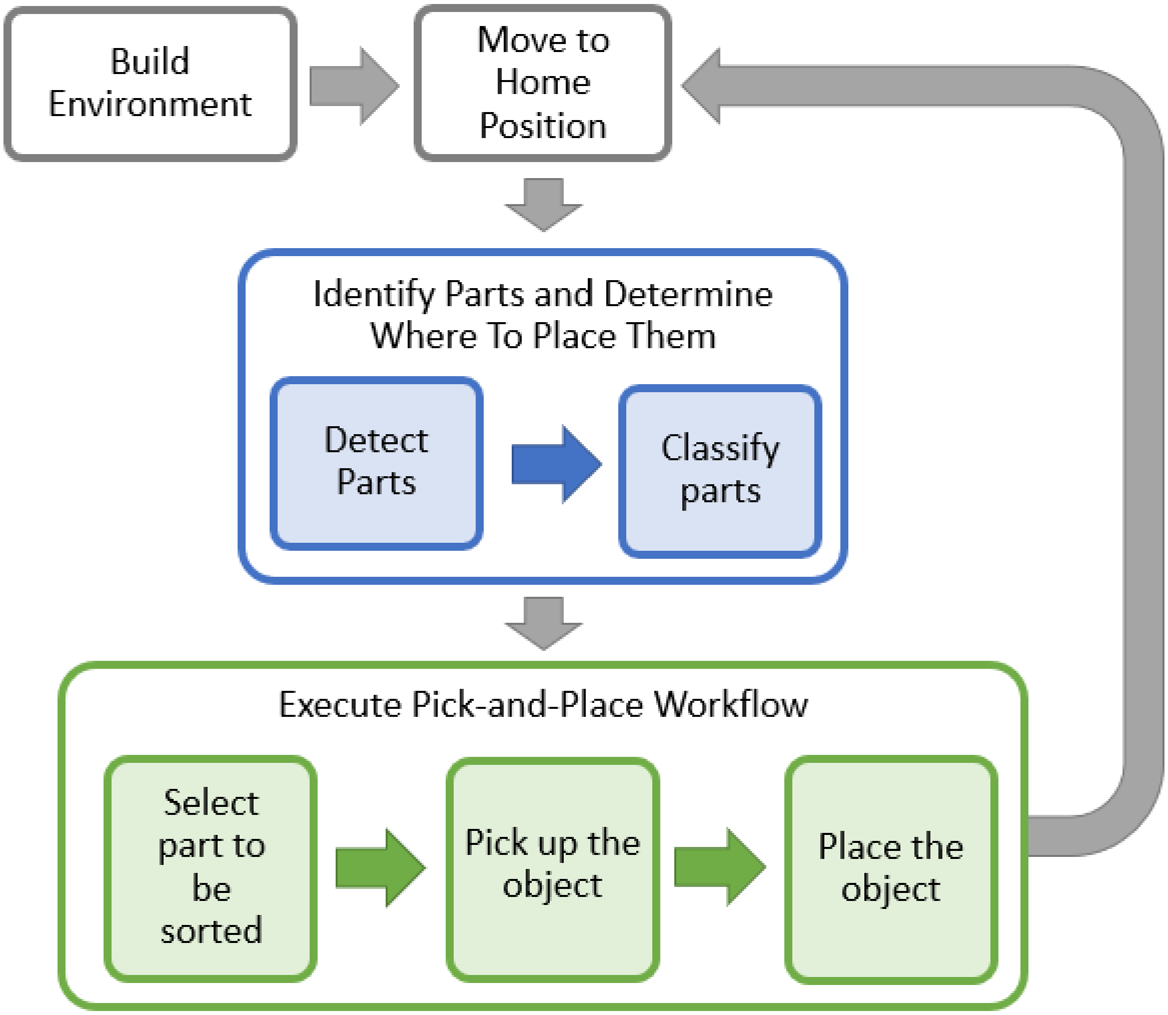
Stateflow Chart

This example uses a Stateflow chart to schedule tasks in the example. Open the chart to examine the contents and follow state transitions during chart execution.

```
edit exampleHelperFlowChartPickPlaceROSGazebo.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 Pick-and-Place Workflow



For a high-level description of the pick-and-place steps, see “Pick-and-Place Workflow Using Stateflow for MATLAB” on page 1-286.

Opening and closing the gripper

The command for activating the gripper, `exampleCommandActivateGripperROSGazebo`, sends an action request to open and close the gripper implemented in Gazebo. To send a request to open the gripper, the following code is used. **Note:** The example code shown just illustrates what the command does. Do not run.

```
[gripAct,gripGoal] = rosactionclient('/my_gen3/custom_gripper_controller/gripper_cmd');
gripperCommand = rosmessage('control_msgs/GripperCommand');
gripperCommand.Position = 0.0;
gripGoal.Command = gripperCommand;
sendGoal(gripAct,gripGoal);
```

Moving the Manipulator to a Specified Pose

The `commandMoveToTaskConfig` command function is used to move the manipulator to specified poses.

Planning

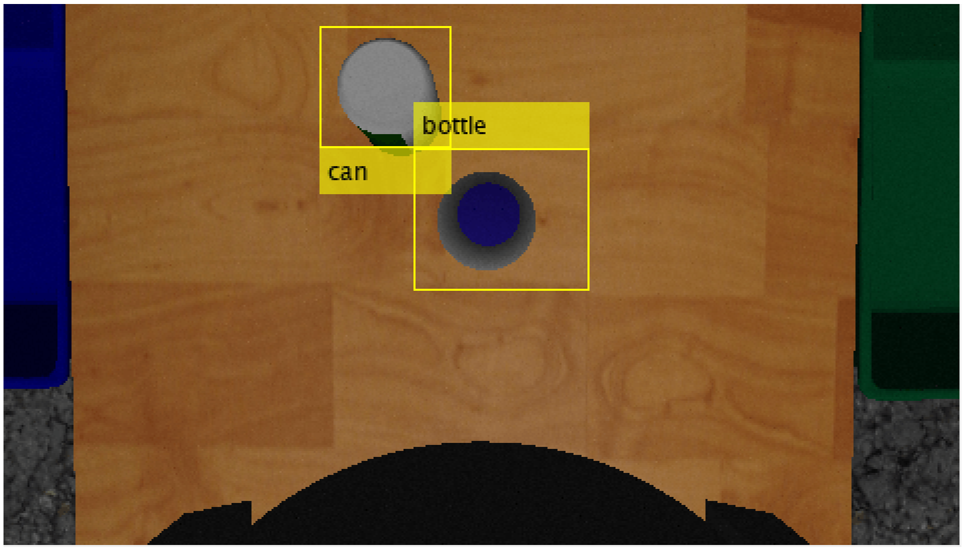
The path planning generates simple task-space trajectories from an initial to a desired task configuration using `trapveltraj` and `transformtraj`. For more details on planning and executing trajectories, see “Plan and Execute Task- and Joint-Space Trajectories Using KINOVA Gen3 Manipulator” on page 1-257.

Joint Trajectory Controller in ROS

After generating a joint trajectory for the robot to follow, `commandMoveToTaskConfig` samples the trajectory at the desired sample rate, packages it into joint-trajectory ROS messages and sends an action request to the joint-trajectory controller implemented in the KINOVA ROS package.

Detecting and classifying objects in the scene

The functions `commandDetectParts` and `commandClassifyParts` use the simulated end-effector camera feed from the robot and apply a pretrained deep-learning model to detect the recyclable parts. The model takes a camera frame as input and outputs the 2D location of the object (pixel position) and the type of recycling it requires (blue or green bin). The 2D location on the image frame is mapped to the robot base frame.



Deep-Learning Model Training: Acquiring and Labeling Gazebo Images

The detection model was trained using a set of images acquired in the simulated environment within the Gazebo world with the two classes of objects (bottle, can) placed on different locations of the table. The images are acquired from the simulated camera on-board the robot, which is moved along the horizontal and vertical planes to get images of the objects from many different camera perspectives.

The images are then labeled using the Image Labeler (Computer Vision Toolbox) app, creating the training dataset for the YOLO v2 detection model. `trainYOLOv2ObjectDetector` (Computer Vision

Toolbox) trains the model. To see how to train a YOLO v2 network in MATLAB, see “Train YOLO v2 Network for Vehicle Detection” (Computer Vision Toolbox).

The trained model is deployed for online inference on the single image acquired by the on-board camera when the robot is in the home position. The `detect` (Computer Vision Toolbox) function returns the image position of the bounding boxes of the detected objects, along with their classes, that is then used to find the position of the center of the top part of the objects. Using a simple camera projection approach, assuming the height of the objects is known, the object position is projected into the world and finally used as reference position for picking the object. The class associated with the bounding boxed decides which bin to place the object.

Start the Pick-and-Place Task

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

```
load('exampleHelperKINOVAGen3GripperROSGazebo.mat');
```

Initialize the Pick and Place Coordinator

Set the initial robot configuration. Create the coordinator, which handles the robot control, by giving 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";
```

Initialize the coordinator by giving the robot model, initial configuration, and end-effector name.

```
coordinator = exampleHelperCoordinatorPickPlaceROSGazebo(robot,initialRobotJConfig, endEffectorF
```

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])*axang2tfor
coordinator.PlacingPose{2} = trvec2tform([[0.2 -0.55 0.26]])*axang2tform([0 0 1 pi/2])*axang2tfo
```

Run and Visualize the 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 = exampleHelperFlowChartPickPlaceROSGazebo('coordinator', coordinator);
```

Use a dialog to start the pick-and-place task execution. Click **Yes** in the dialog to begin the simulation.

```
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
```

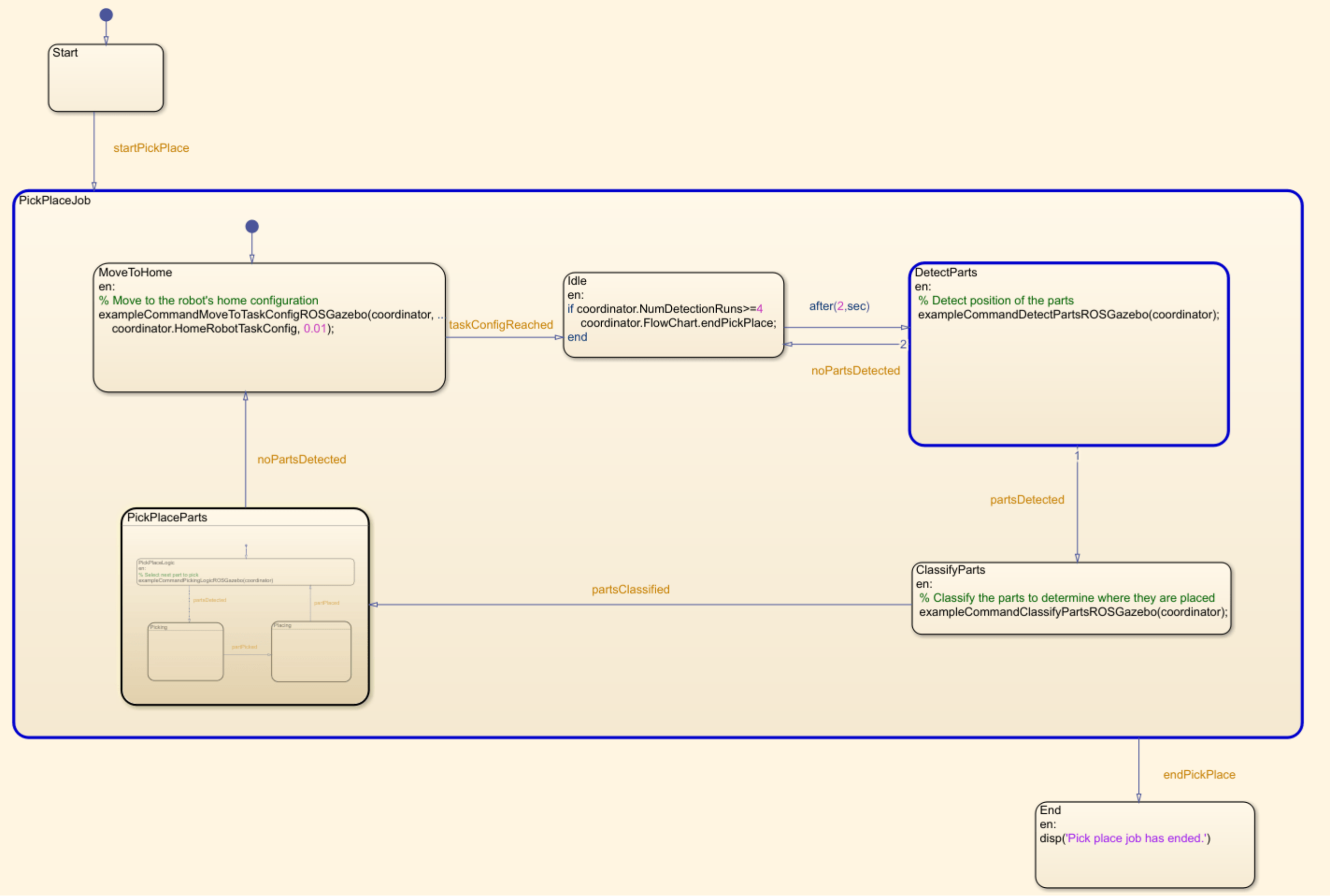
Ending the pick-and-place task

The Stateflow chart will finish executing automatically after 3 failed attempts to detect new objects. To end the pick-and-place task prematurely, uncomment and execute the following lines of code or press Ctrl+C in the command window.

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

Observe the Simulation States

During execution, the active states at each point in time are highlighted in blue in the Stateflow chart. This helps keeping track of what the robot does and when. You can click through the subsystems to see the details of the state in action.



```
MoveToHome
en:
% Move to the robot's home configuration
exampleCommandMoveToTaskConfigROSGazebo(coordinator, ..
    coordinator.HomeRobotTaskConfig, 0.01);
```

```
Idle
en:
if coordinator.NumDetectionRuns>=4
    coordinator.FlowChart.endPickPlace;
end
```

```
DetectParts
en:
 % Detect position of the parts
 exampleCommandDetectPartsROSGazebo(coordinator);
```

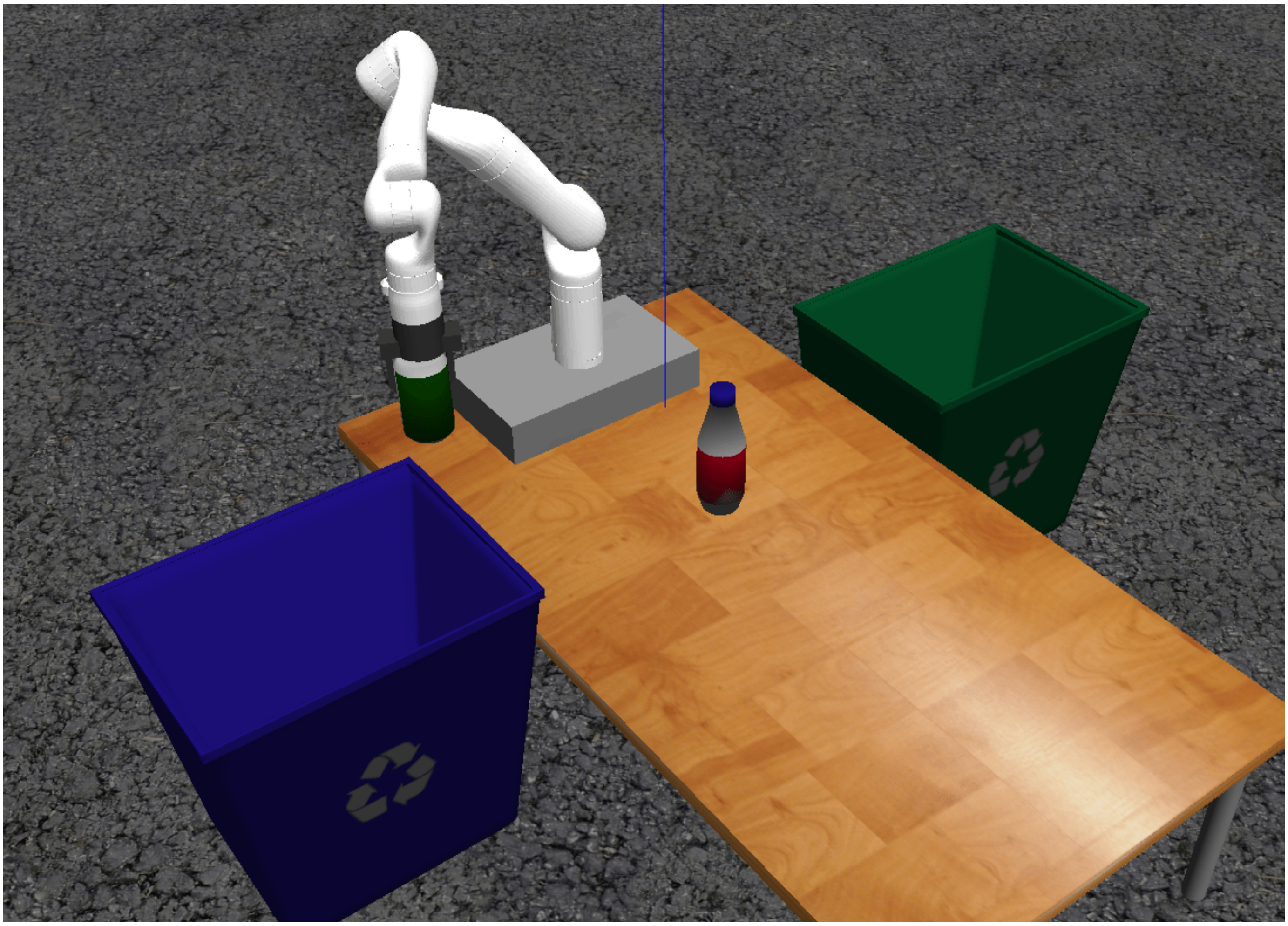
```
PickPlaceLogic
en:
% Select next part to pick
exampleCommandPickingLogicROSGazebo(coordinator)
```

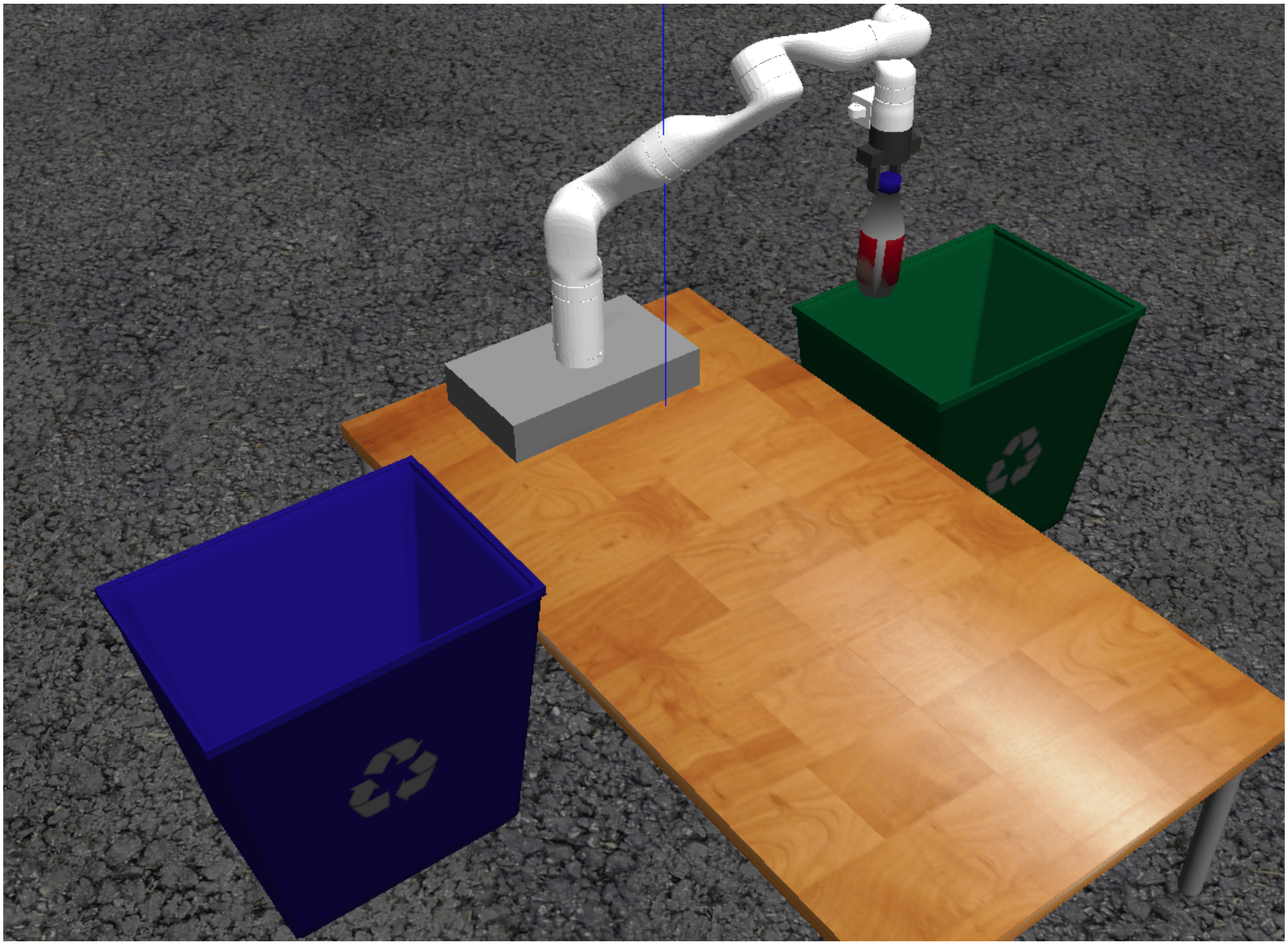
```
ClassifyParts
en:
 % Classify the parts to determine where they are placed
 exampleCommandClassifyPartsROSGazebo(coordinator);
```

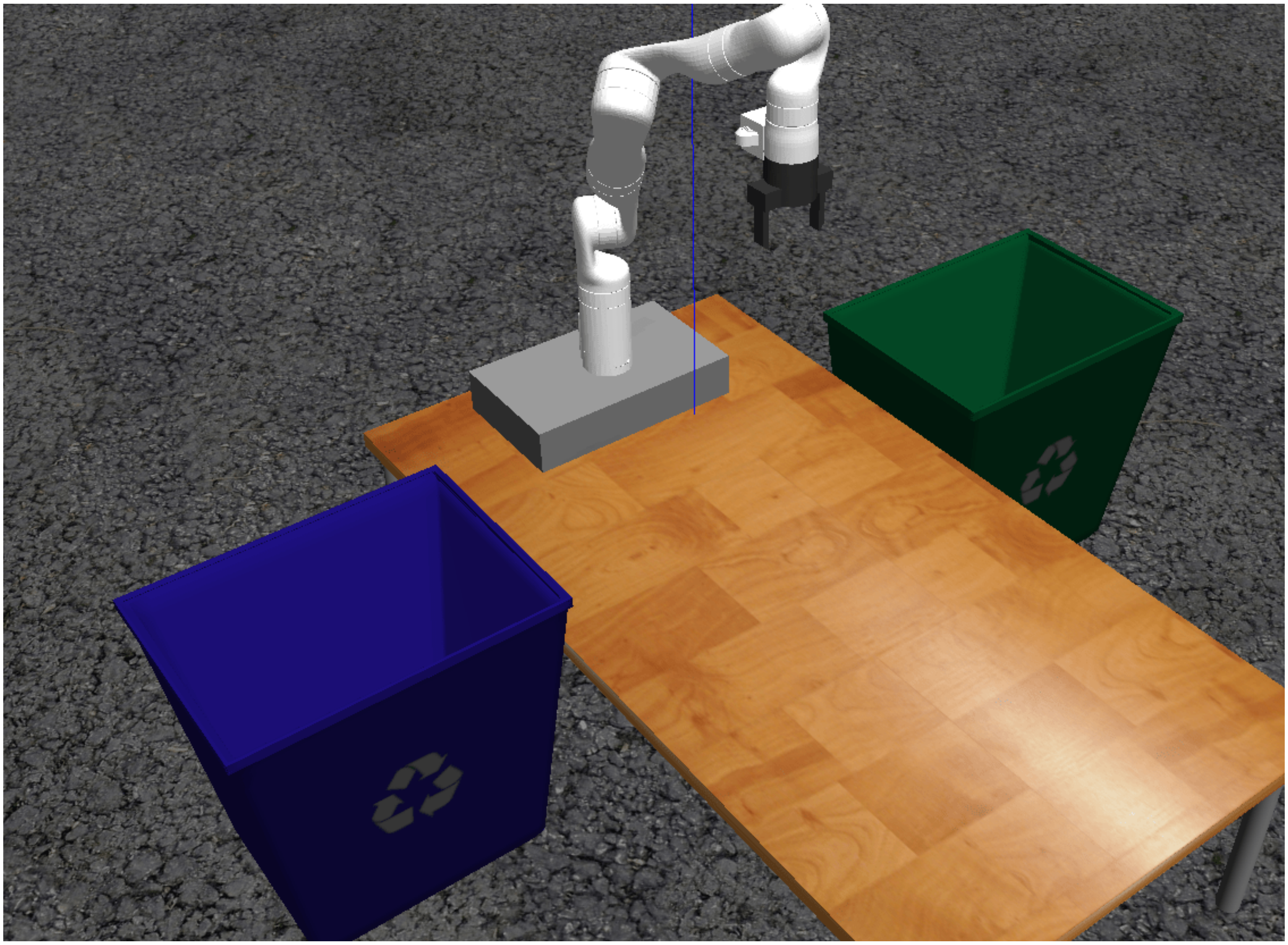
```
End
en:
disp('Pick place job has ended.')
```

Visualize the Pick-and-Place Action in Gazebo

The Gazebo world shows the robot in the working area as it moves parts to the recycling bins. The robot continues working until all parts have been placed. When the detection step doesn't find any more parts four times, the Stateflow chart exits.







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

Shutdown the ROS network after finishing the example.

```
rosshutdown
```

```
Shutting down global node /matlab_global_node_36570 with NodeURI http://192.168.31.1:51073/
```

Copyright 2020 The MathWorks, Inc.

Pick-and-Place Workflow in Gazebo Using Point-Cloud Processing and RRT Path Planning

Setup an end-to-end pick and place workflow for a robotic manipulator like the KINOVA® Gen3.

The pick-and-place workflow implemented in this example can be adapted to different scenarios, planners, simulation platforms, and object detection options. The example shown here uses RRT for planning and simulates the robot in Gazebo using the Robot Operating System (ROS). For other pick-and-place workflows, see:

- “Pick-and-Place Workflow Using Stateflow for MATLAB” on page 1-286
- “Pick-and-Place Workflow Using RRT Planner and Stateflow for MATLAB” on page 1-294
- “Pick-and-Place Workflow in Gazebo Using ROS” on page 1-302

Overview

This example identifies and recycles objects into two bins using a KINOVA Gen3 manipulator. The example uses tools from five toolboxes:

- **Robotics System Toolbox™** is used to model and simulate the manipulator.
- **ROS Toolbox™** is used for connecting MATLAB to Gazebo.
- **Image Processing Toolbox™** and **Computer Vision Toolbox™** are used for object detection using point cloud processing and simulated depth camera in Gazebo.

This example builds on key concepts from the following related examples:

- “Pick and Place Using RRT for Manipulators” on page 1-275
- “Get Started with Gazebo and Simulated TurtleBot” (ROS Toolbox) (ROS Toolbox)
- “3-D Point Cloud Registration and Stitching” (Computer Vision Toolbox) (Computer Vision Toolbox)

Robot Simulation and Control in Gazebo

Start a ROS-based simulator for a KINOVA Gen3 robot and configure the MATLAB® connection with the robot simulator.

This example uses a virtual machine (VM) containing ROS Melodic available for download here.

- Start the Ubuntu® virtual machine desktop.
- In the Ubuntu desktop, click the **Gazebo Recycling World - Depth Sensing** icon to start the Gazebo world built for this example.
- Specify the IP address and port number of the ROS master in Gazebo so that MATLAB® can communicate with the robot simulator. For this example, the ROS master in Gazebo uses the IP address of `172.21.72.160` displayed on the Desktop. Adjust the `rosIP` variable based on your VM.
- Start the ROS 1 network using `rosinit`.

```
rosIP = '172.16.34.129'; % IP address of ROS-enabled machine

rosshutdown;

rosinit(rosIP,11311); % Initialize ROS connection
```

```
Initializing global node /matlab_global_node_63627 with NodeURI http://172.16.34.1:35153/ and Mas
```

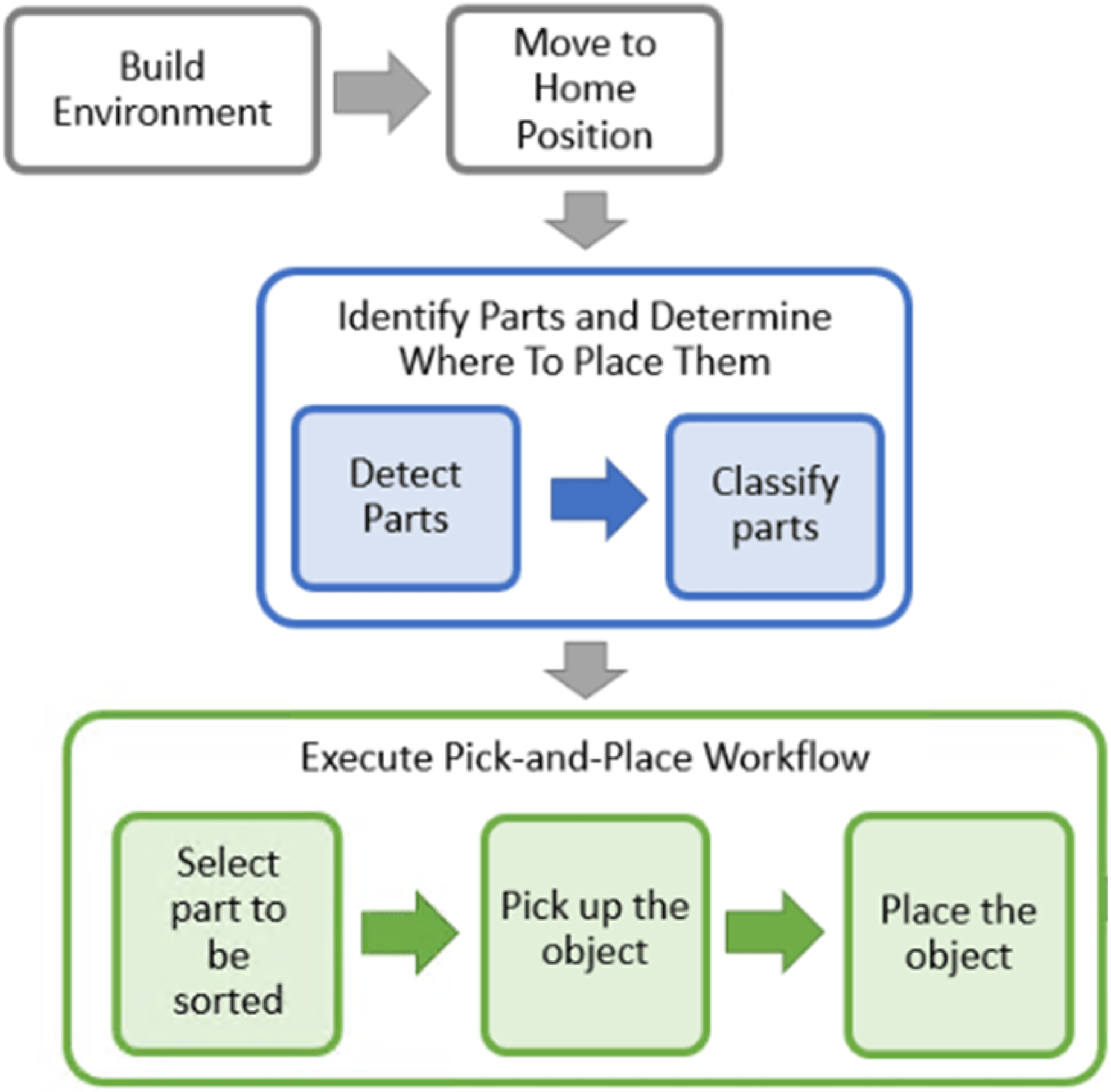
After initializing the Gazebo world by click the icon, the VM loads a KINOVA Gen3 Robot arm on a table with one recycling bin on each side. To simulate and control the robot arm in Gazebo, the VM contains the ros_kortex ROS package, which are provided by KINOVA.

The packages use ros_control to control the joints to desired joint positions. For additional details on using the VM, refer to “Get Started with Gazebo and Simulated TurtleBot” (ROS Toolbox)

Pick-and-Place Tasks

The Pick-and-Place workflow is implemented in MATLAB and consists of basic initialization steps, followed by two main sections:

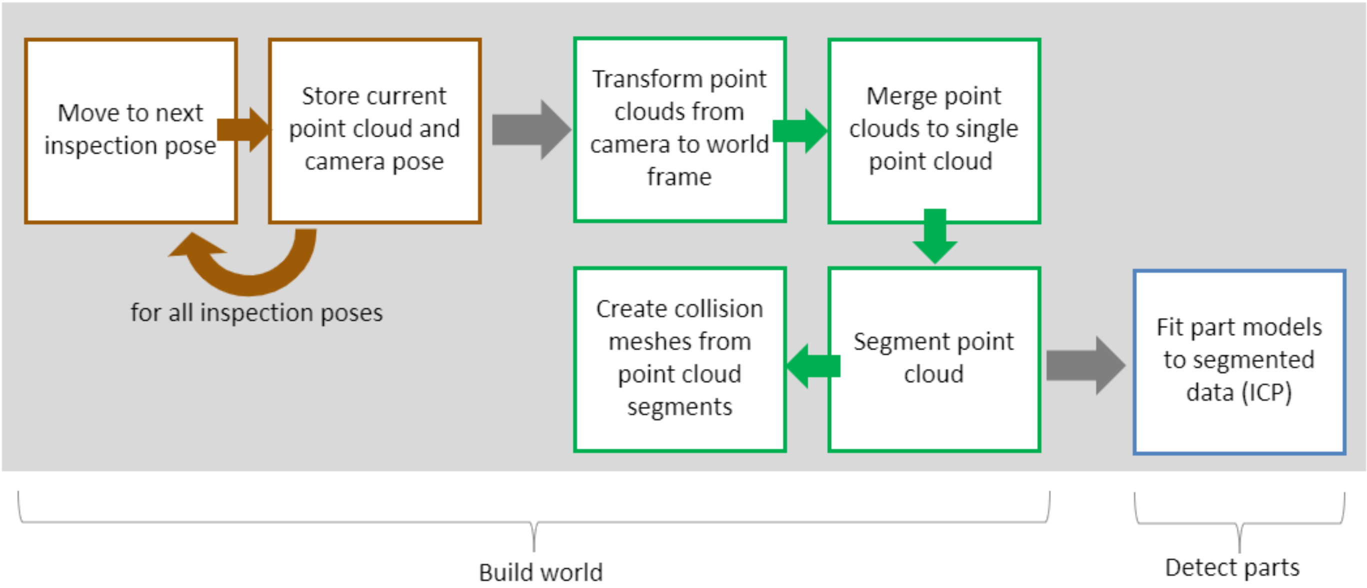
- Identify Parts and Determine Where to Place Them
- Execute Pick-and-Place Workflow



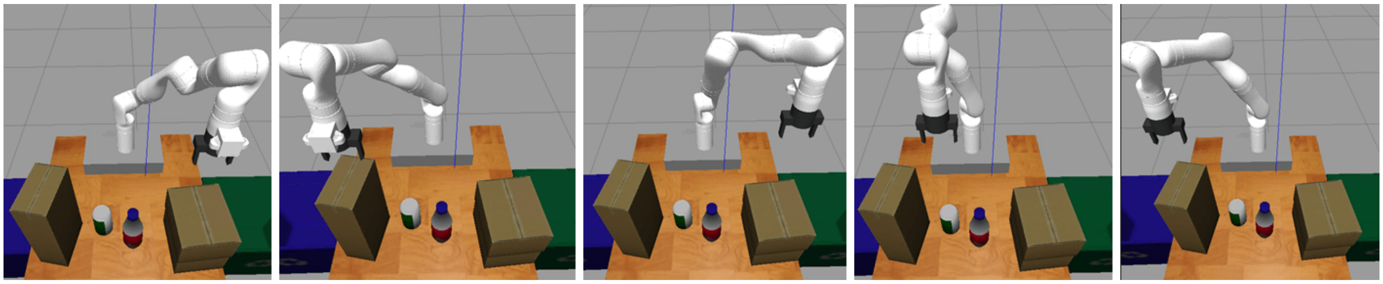
For an implementation that uses Stateflow to schedule the tasks, see “Pick-and-Place Workflow Using Stateflow for MATLAB” on page 1-286.

Scanning the environment to build planning scene for RRT path planner

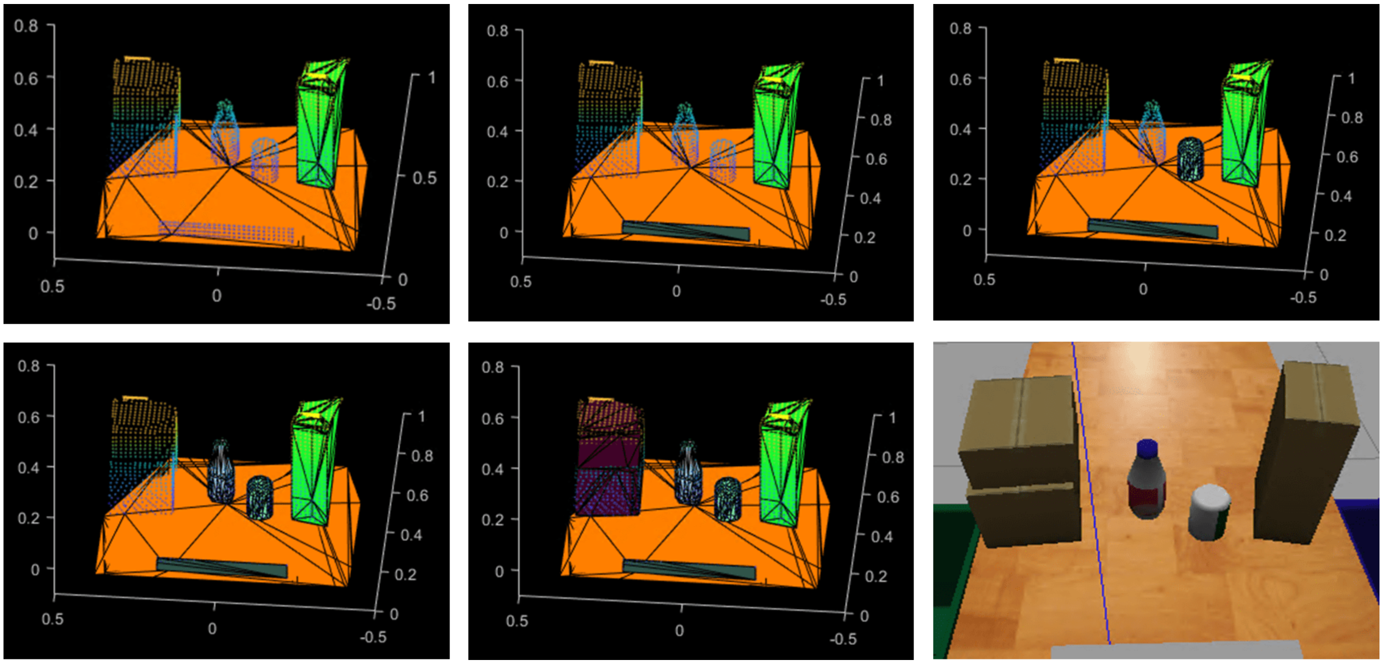
Before starting the pick-and-place job, the robot goes through a set of tasks to identify the planning scene in the `exampleCommandBuildWorld` function and detects the objects to pick using the `exampleCommandDetectParts` function.


Move to next inspection pose
Store current point cloud and camera pose
for all inspection poses
Transform point clouds from camera to world frame
Merge point clouds to single point cloud
Segment point cloud
Create collision meshes from point cloud segments
Fit part models to segmented data (ICP)
Build world
Detect parts

First, the robot moves to predefined scanning poses one by one and captures a set of point clouds of the scene using an onboard depth sensor. At each of the scanning poses, the current camera pose is retrieved by reading the corresponding ROS transformation using `rostf` (ROS Toolbox) and `getTransform` (ROS Toolbox). The scanning poses are visualized below:



Once the robot has visited all the scanning poses, the captured point clouds are transformed from camera to world frame using `pctransform` (Computer Vision Toolbox) and merged to a single point cloud using `pcmerge` (Computer Vision Toolbox). The final point cloud is segmented based on Euclidean distance using `pcsegdist` (Computer Vision Toolbox). The resulting point cloud segments are then encoded as collision meshes (see `collisionMesh`) to be easily identified as obstacles during RRT path planning. The process from point cloud to collision meshes is shown one mesh at at a time below.



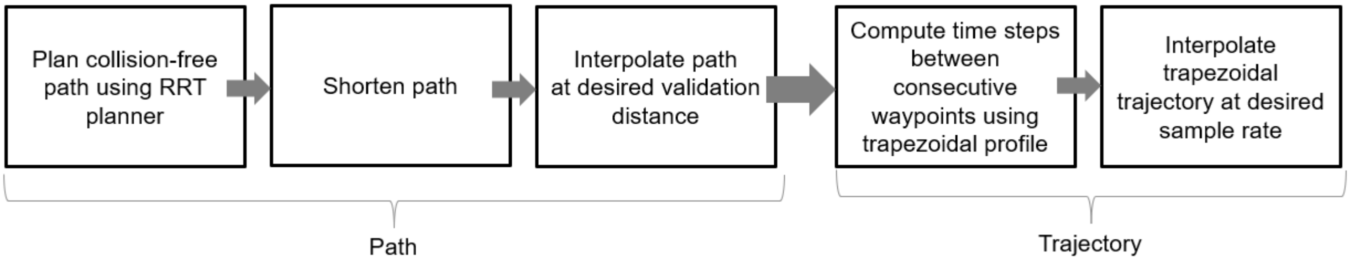
Opening and closing the gripper

The command for activating the gripper, `exampleCommandActivateGripper`, sends an action request to open and close the gripper implemented in Gazebo. For example, to send a request to open the gripper, the following code is used.

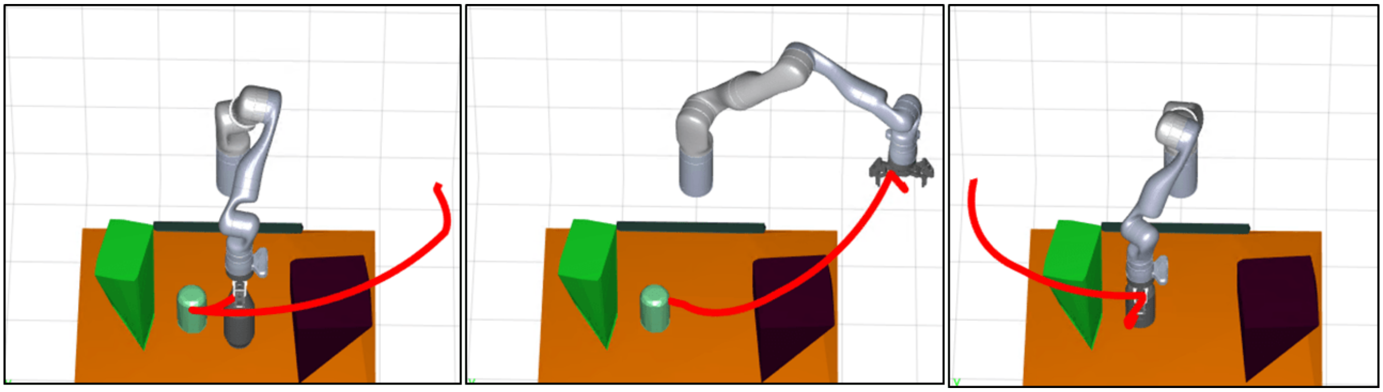
```
[gripAct,gripGoal] = rosactionclient('/my_gen3/custom_gripper_controller/gripper_cmd');
gripperCommand = rosmessage('control_msgs/GripperCommand');
gripperCommand.Position = 0.0;
gripGoal.Command = gripperCommand;
sendGoal(gripAct,gripGoal);
```

Moving the manipulator to a specified pose

Most of the task execution consists of instructing the robot to move between different specified poses. The `exampleHelperMoveToTaskConfig` function defines an RRT planner using the `manipulatorRRT` object, which plans paths from an initial to a desired joint configuration by avoiding collisions with specified collision objects in the scene. The resulting path is first shortened and then interpolated at a desired validation distance. To generate a trajectory, the `trapveltraj` function is used to assign time steps to each of the interpolated waypoints following a trapezoidal profile. Finally, the waypoints with their associated times are interpolated to a desired sample rate (every 0.1 seconds). The generated trajectories ensure that the robot moves slowly at the start and the end of the motion when it is approaching or placing an object.


Plan collision-free path using RRT planner
Shorten path
Interpolate path at desired validation distance
Compute time steps between consecutive waypoints using trapezoidal profile
Interpolate trapezoidal trajectory at desired sample rate
Path
Trajectory

The planned paths are visualized in MATLAB along with the planning scene.



This workflow is examined in detail in the “Pick-and-Place Workflow Using RRT Planner and Stateflow for MATLAB” on page 1-294 example. For more information about the RRT planner, see “Pick and Place Using RRT for Manipulators” on page 1-275. For simpler trajectories where the paths are known to be obstacle-free, trajectories could be executed using trajectory generation tools and simulated using the manipulator motion models. See “Plan and Execute Task- and Joint-Space Trajectories Using KINOVA Gen3 Manipulator” on page 1-257.

Joint Trajectory Controller in ROS

After generating a joint trajectory for the robot to follow, the `exampleCommandMoveToTaskConfig` function samples the trajectory at the desired sample rate, packages it into joint-trajectory ROS messages and sends an action request to the joint-trajectory controller implemented in the KINOVA ROS package.

Detecting and classifying objects in the scene

The functions `exampleCommandDetectParts` and `exampleCommandClassifyParts` use the simulated end-effector depth camera feed from the robot to detect the recyclable parts. Since a complete point cloud of the scene is available from the **Build Environment** step, the iterative closest point (ICP) registration algorithm implemented in `pcregistericp` (Computer Vision Toolbox) identifies which of the segmented point clouds match the geometries of objects that should be picked.

Start the Pick-and-Place Workflow

This simulation uses a KINOVA Gen3 manipulator with a gripper attached.

```
load('exampleHelperKINOVAGen3GripperGazeboRRTScene.mat');
rng(0)
```

Initialize the Pick-and-Place Application

Set the initial robot configuration and name of the end-effector body.

```
initialRobotJConfig =  [3.5797   -0.6562   -1.2507   -0.7008    0.7303   -2.0500   -1.9053];
endEffectorFrame = "gripper";
```

Initialize the coordinator by giving the robot model, initial configuration, and end-effector name.

```
coordinator = exampleHelperCoordinatorPickPlaceROSGazeboScene(robot,initialRobotJConfig, endEffe
```

Specify pick-and-place coordinator properties.

```
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 the Pick-and-Place Application Step by Step

```
% Task 1: Build world
exampleCommandBuildWorldROSGazeboScene(coordinator);
```

```
Moving to scanning pose 1
Searching for other config...
Now planning...
Waiting until robot reaches the desired configuration
Capturing point cloud 1
Getting camera pose 1
Moving to scanning pose 2
Now planning...
Waiting until robot reaches the desired configuration
Capturing point cloud 2
Getting camera pose 2
Moving to scanning pose 3
Searching for other config...
Now planning...
Waiting until robot reaches the desired configuration
Capturing point cloud 3
Getting camera pose 3
Moving to scanning pose 4
Now planning...
Waiting until robot reaches the desired configuration
Capturing point cloud 4
Getting camera pose 4
Moving to scanning pose 5
Now planning...
Waiting until robot reaches the desired configuration
Capturing point cloud 5
Getting camera pose 5
```

```
% Task 2: Move to home position
exampleCommandMoveToTaskConfigROSGazeboScene(coordinator,coordinator.HomeRobotTaskConfig);
```

```
Now planning...
Waiting until robot reaches the desired configuration
```

```
% Task 3: Detect objects in the scene to pick
exampleCommandDetectPartsROSGazeboScene(coordinator);
```

```
Bottle detected...
Can detected...
```

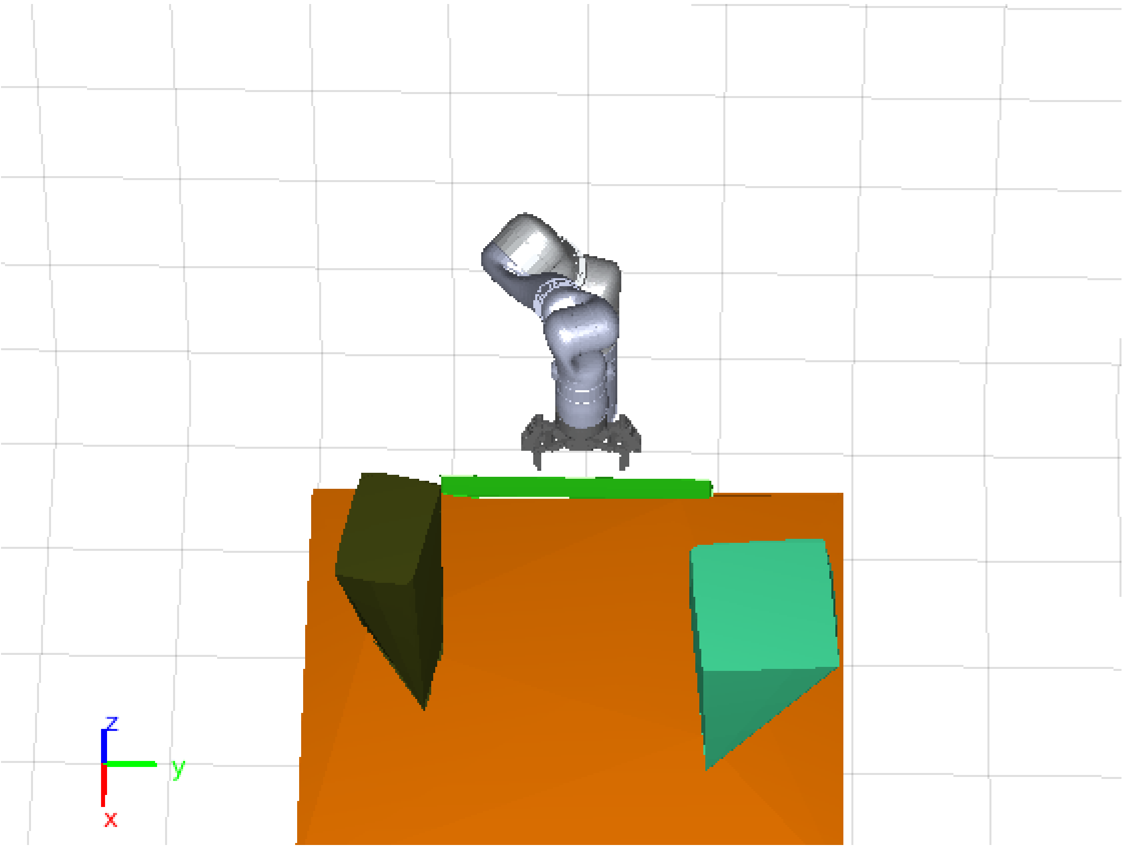
```
% Task 4: Select next part to pick
remainingParts = exampleCommandPickingLogicROSGazeboScene(coordinator);
```

```
     1
```

```
while remainingParts==true
    % Task 5: [PICKING] Compute grasp pose
    exampleCommandComputeGraspPoseROSGazeboScene(coordinator);

    % Task 6: [PICKING] Move to picking pose
    exampleCommandMoveToTaskConfigROSGazeboScene(coordinator, coordinator.GraspPose);

    % Task 7: [PICKING] Activate gripper
    exampleCommandActivateGripperROSGazeboScene(coordinator,'on');

    % Part has been picked

    % Task 8: [PLACING] Move to placing pose
    exampleCommandMoveToTaskConfigROSGazeboScene(coordinator, ...
    coordinator.PlacingPose{coordinator.DetectedParts{coordinator.NextPart}.placingBelt});

    % Task 9: [PLACING] Deactivate gripper
    exampleCommandActivateGripperROSGazeboScene(coordinator,'off');

    % Part has been placed

    % Select next part to pick
    remainingParts = exampleCommandPickingLogicROSGazeboScene(coordinator);

    % Move to home position
    exampleCommandMoveToTaskConfigROSGazeboScene(coordinator,coordinator.HomeRobotTaskConfig);
end
```

```
Now planning...
Waiting until robot reaches the desired configuration

Gripper closed...

Now planning...
Waiting until robot reaches the desired configuration

Gripper open...

     2

Now planning...
Waiting until robot reaches the desired configuration

Now planning...
Waiting until robot reaches the desired configuration

Gripper closed...

Now planning...
Waiting until robot reaches the desired configuration
```

```
Gripper open...

Now planning...
Waiting until robot reaches the desired configuration
```

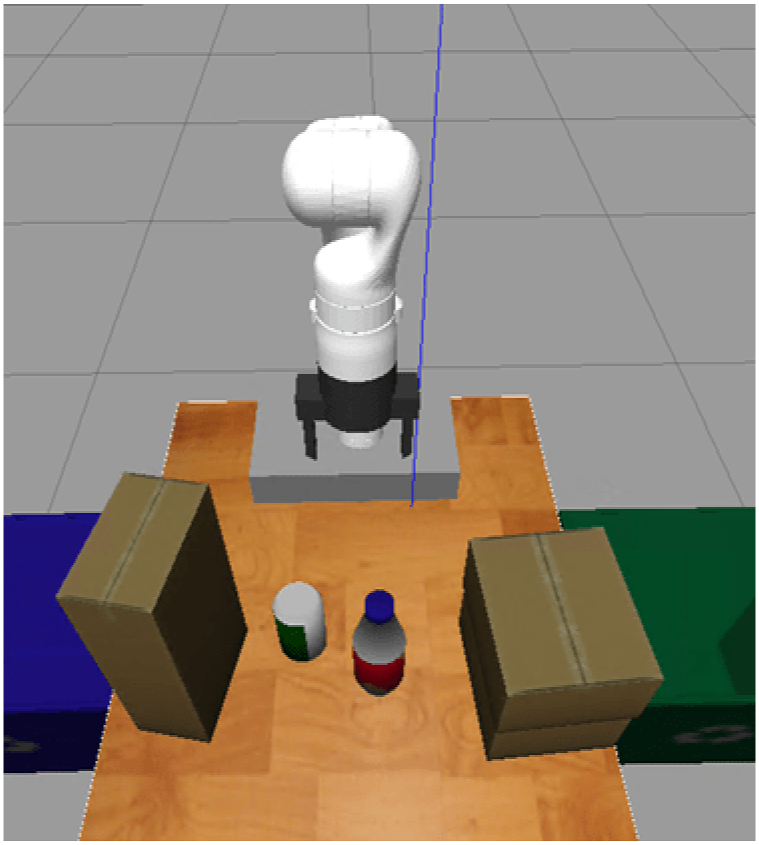


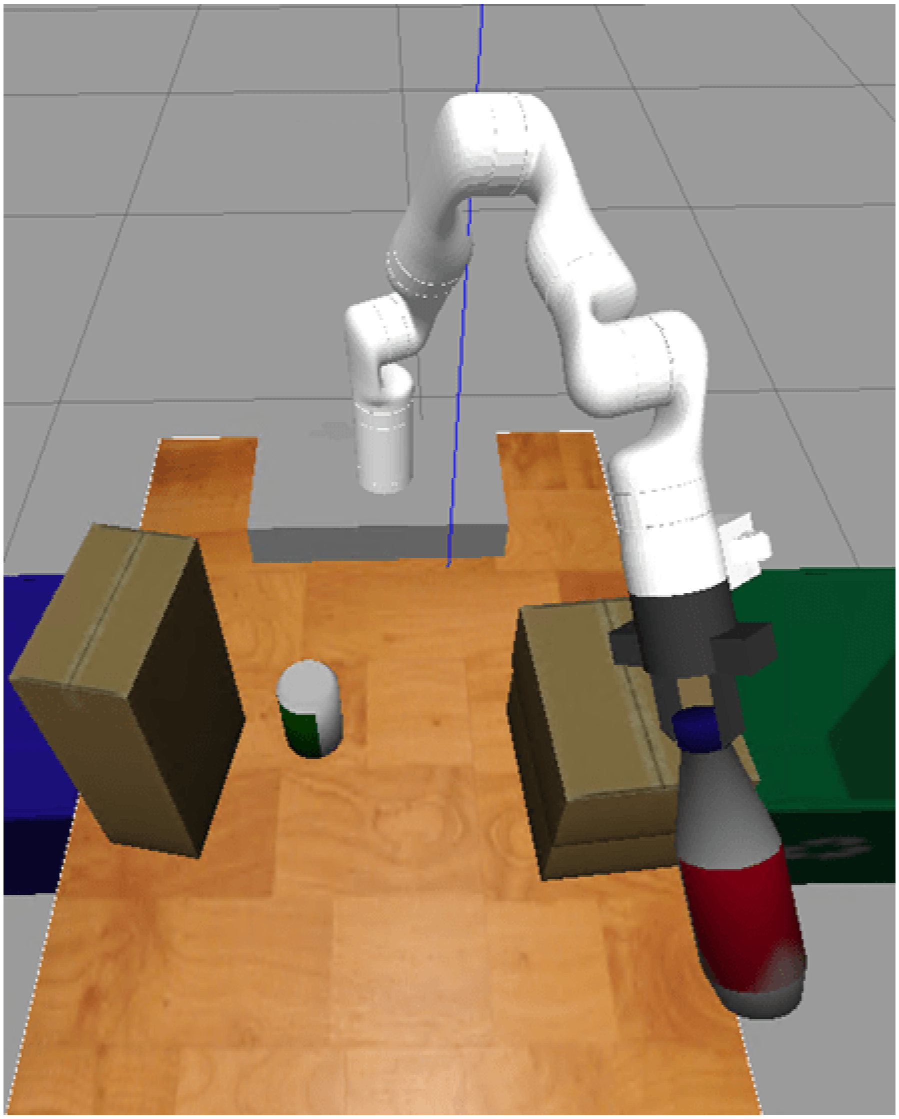
```
% Shut down ros when the pick-and-place application is done
rosshutdown;
```

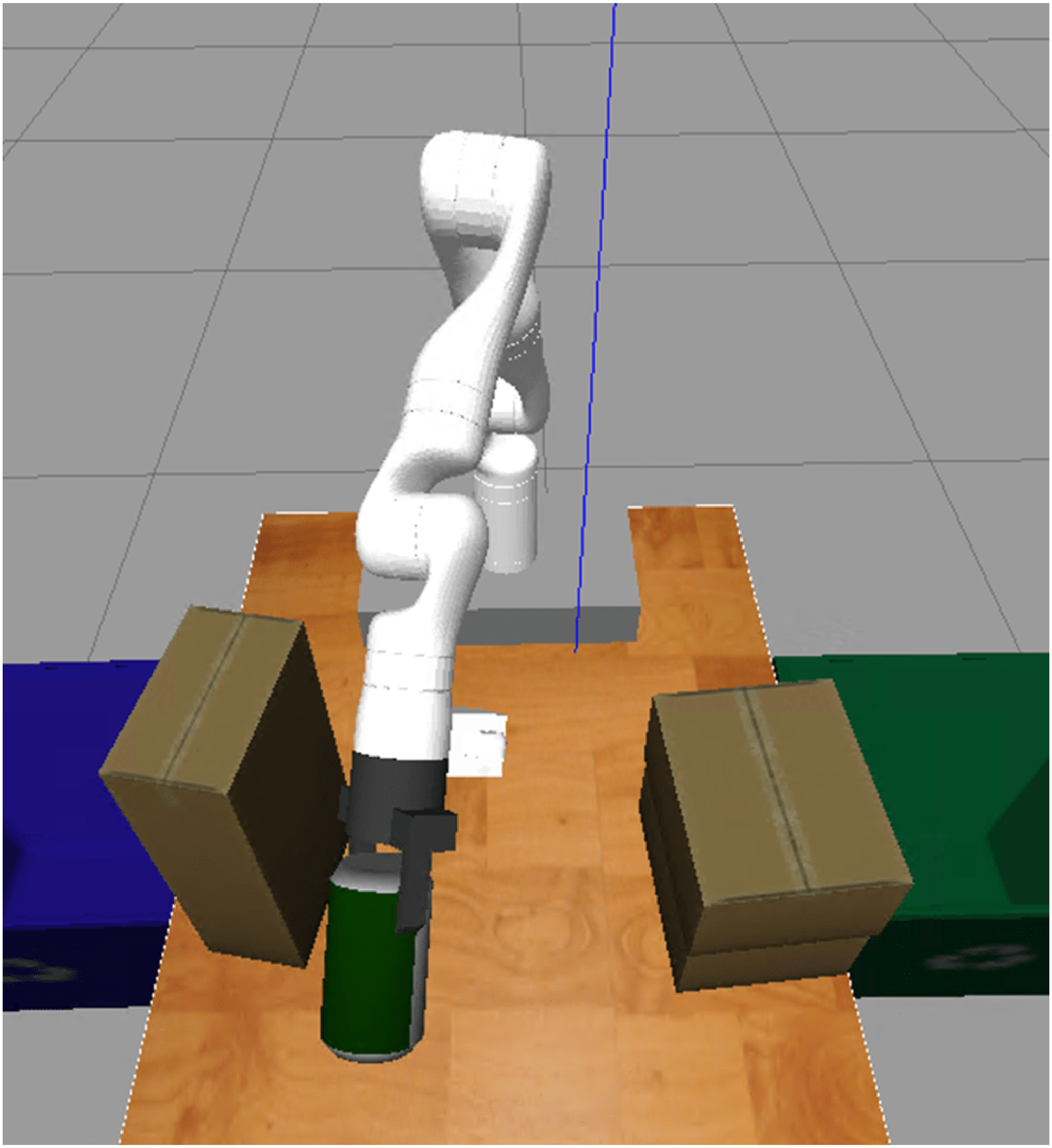
```
Shutting down global node /matlab_global_node_63627 with NodeURI http://172.16.34.1:35153/ and Ma
```

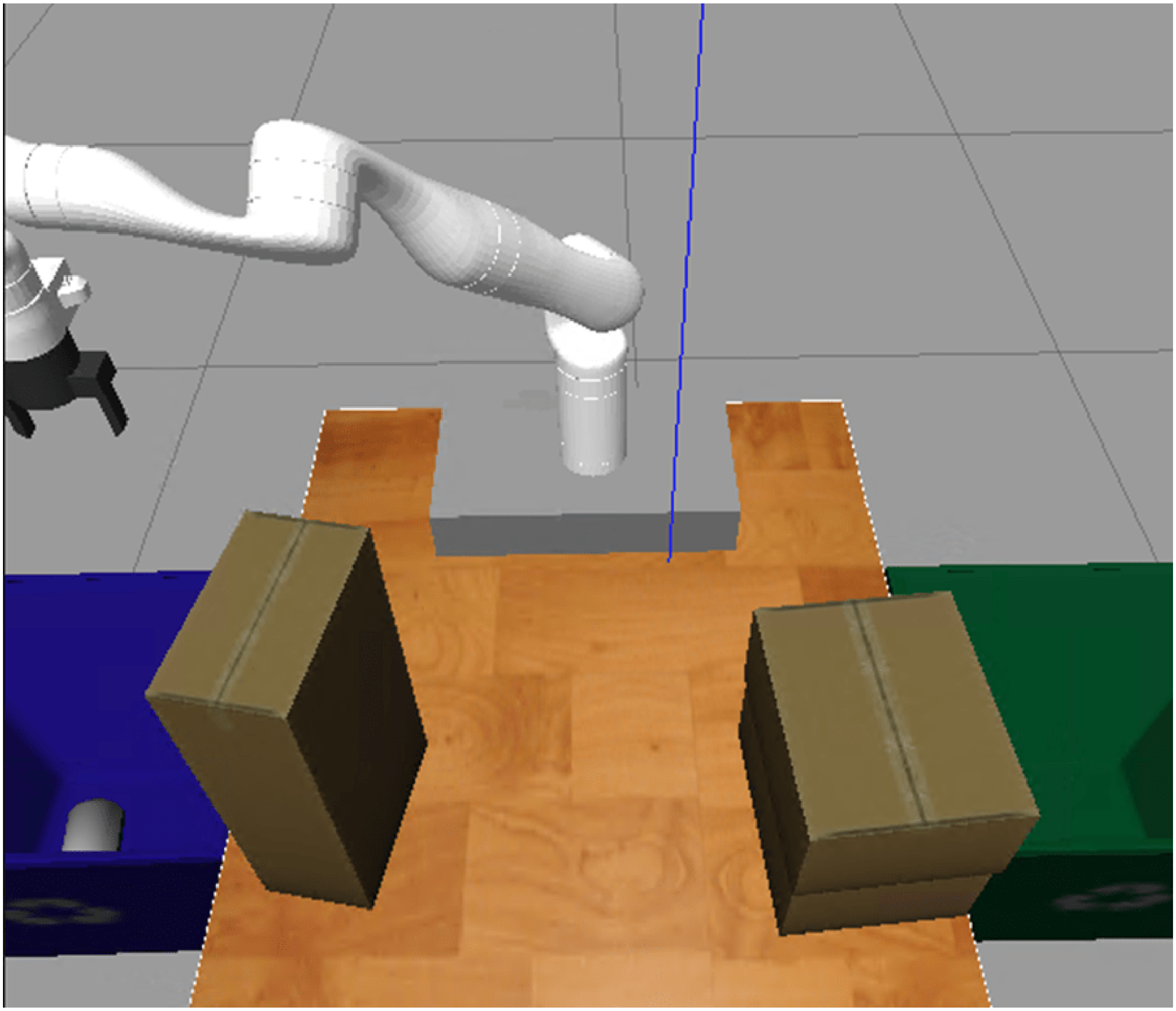
Visualize the Pick-and-Place Action in Gazebo

The Gazebo world shows the robot in the working area as it moves parts to the recycling bins. The robot continues working until all parts have been placed.









Copyright 2021 The MathWorks, Inc.

Plan Paths With End-Effector Constraints Using State Spaces For Manipulators

Plan a manipulator robot path using sampling-based planners like the rapidly-exploring random trees (RRT) algorithm.

This example uses the `manipulatorStateSpace` and `manipulatorCollisionBodyValidator` objects as a state space and state validator that works with sampling-based planners, like the `plannerBiRRT` object available with Navigation Toolbox™. In this example, you define the custom behavior of a manipulator state space to ensure end-effector constraints are met. You plan a path to pick and place a container with a constrained end effector that must remain upright throughout the path. This constraint could be for a container of fluids, a welding tool path, or drilling application where you must keep the end-effector pose in a fixed orientation.

Define State Space

To create a constrained state space, generate a class `exampleHelperConstrainedStateSpace` that derives from `manipulatorStateSpace`. Open this example to get the supporting file.

```
classdef exampleHelperConstrainedStateSpace < manipulatorStateSpace
```

Specify the constructor syntax which requires the robot model, end effector name, and target orientation as inputs. Define a property called `EnableConstraint` for turning the constraint on and off.

The constrained region is defined in the constructor using a `workspaceGoalRegion` object. The bounds on the *xyz*-position are all `[-100,100]` meters. The orientation bounds are constrainted to any rotation `[-pi pi]` about the *z*-axis with zero rotation in *x* and *y*.

Create an inverse kinematics (IK) solver, and specify any solver parameters. This solver generates the joint configurations of the robot for the end-effector poses sampled inside the workspace.

```
        function obj = exampleHelperConstrainedStateSpace(rbt,endEffectorName,targetOrientation)
            %Constructor
            obj@manipulatorStateSpace(rbt);
            obj.EnableConstraint = true;
            obj.Region = workspaceGoalRegion(endEffectorName,'EndEffectorOffsetPose',targetOrien
            obj.Region.Bounds = [-100 100; -100 100; -100 100; -pi pi; 0 0; 0 0];

            % Store a reference of the manipulatorStateSpace/RigidBodyTree
            % in the Robot property. Note that the RigidBodyTree property
            % of the manipulatorStateSpace is read-only. Hence, accessing
            % it will involve creating a copy of the underlying handle,
            % which can be expensive.
            obj.Robot = obj.RigidBodyTree;

            % Configure the IK solver
            obj.IKSolver = inverseKinematics('RigidBodyTree', obj.Robot);
            obj.IKSolver.SolverParameters.AllowRandomRestart = false;
        end
```

Customize Interpolation Method

During the planning phase, the `interpolate` function connects configurations in the search tree. By default, the interpolation is unconstrained. Customize the `interpolate` function to ensure that the end-effector remains upright. Override the `interpolate` method on the `manipulatorStateSpace` object with this code.

```
function constrainedStates = interpolate(obj, state1, state2, ratios)
    constrainedStates = interpolate@manipulatorStateSpace(obj,state1,state2,ratios);
    if(obj.EnableConstraint)
        for i = 1:size(constrainedStates, 1)
            constrainedStates(i,:) = constrainConfig(obj,constrainedStates(i,:));
        end
    end
end
```

The `constraintConfig` function replaces the unconstrained joint configurations with constrained configurations. It finds the end-effector pose closest to the constrained region [1] and returns the corresponding robot joint configuration. Given the end-effector pose T_s^0 corresponding to `config`, the function computes the pose closest to the end-effector pose in the constrained region and returns the corresponding robot joint configuration.

To find the closest pose in the reference frame, first find the displacement of T_s^0 from the region by calculating $T_{s'}^w = T_w^{0\,-1} T_s^0\, T_e^{w-1}$ and converting $T_{s'}^w$ to a pose vector close to the bounds of the region. Given the bounds, there are three possibililties for an element in the pose vector and its corresponding bounds of the region:

1. The value is within bounds. In this case the element remains unchanged as the pose element is within the region.
2. The value is greater than the maximum value of the bound. In this case, clip it to the max value of the bound.
3. The value is lesser than the min value of the bound. In this case, clip it to the min value of the bound.

Once the resulting closest constrained pose is found in the reference frame, convert this to a pose in the world coordinates.

```
function constrainedConfig = constrainConfig(obj, config)
    %constrainConfig Constraint joint configuration to the region
    %   The function finds the joint configuration corresponding to
    %   the end-effector pose closest to the constrained region.
    wgr = obj.Region;
    T0_s = obj.Robot.getTransform(config, wgr.EndEffectorName);
    T0_w = wgr.ReferencePose;
    Tw_e = wgr.EndEffectorOffsetPose;
    Tw_sPrime = T0_w \ T0_s / Tw_e;
    dw = convertTransformToPoseVector(obj, Tw_sPrime);
    bounds = wgr.Bounds;

    % Find the pose vector closeset to the bounds of the region.
    for dofIdx = 1:6
        if(dw(dofIdx) < bounds(dofIdx, 1))
            dw(dofIdx) = bounds(dofIdx, 1);
        elseif(dw(dofIdx) > bounds(dofIdx, 2))
            dw(dofIdx) = bounds(dofIdx, 2);
        end
    end
```

```
                % Convert the pose vector in the region's reference frame to a
                % homogeneous transform.
                constrainedPose = obj.convertPoseVectorToTransform(dw);

                % Convert this pose to world coordinates, and find the
                % corresponding joint configuration.
                constrainedPose = T0_w * constrainedPose * Tw_e;
                constrainedConfig = obj.IKSolver(obj.Region.EndEffectorName, ...
                    constrainedPose,  ...
                    ones(1, 6), ...
                    config);
            end
```

Create Robot State Space

Now that you have setup your constrained state space, load an example robot and environment using the `exampleHelperEndEffectorConstraintedEnvironment` function. The output `kinova` is a rigid body tree robot model of the KINOVA™ Gen 3 and `env` is a cell array of collision body objects in the robot world.

```
[kinova,env] = exampleHelperEndEffectorConstrainedEnvironment;
```

Specify the end-effector name and target orientation. Store the joint position values for opening and closing the gripper.

```
endEffectorName = "EndEffector_Link";
targetOrientation = eul2tform([0 pi 0]);
openPostion = 0.06;
closedPosition = 0.04;
```

Create the customized state space.

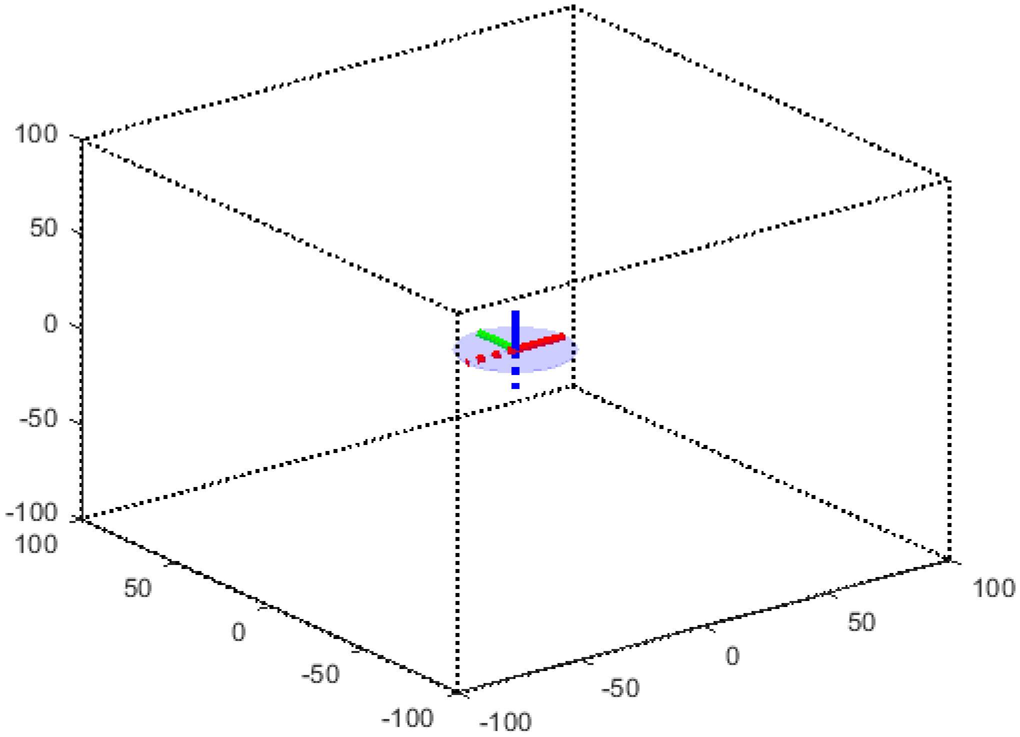
```
ss = exampleHelperConstrainedStateSpace(kinova, endEffectorName, targetOrientation);
```

Specify state bounds on the gripper position so it remains open.

```
ss.StateBounds(end-1:end,:) = repmat([openPostion, openPostion], 2, 1);
```

Visualize the constrained `workspaceGoalRegion` region.

```
figure;
show(ss.Region)
```



Create State Validator

Create a `manipulatorCollisionBodyValidator` object from the state space and add the environment of collision objects. To improve performance, ignore self collisions when valdiating the state space. The robot model uses joint limits to ensure that self collisions will not occur.

```
sv = manipulatorCollisionBodyValidator(ss,"Environment",env);
sv.IgnoreSelfCollision = true;
```

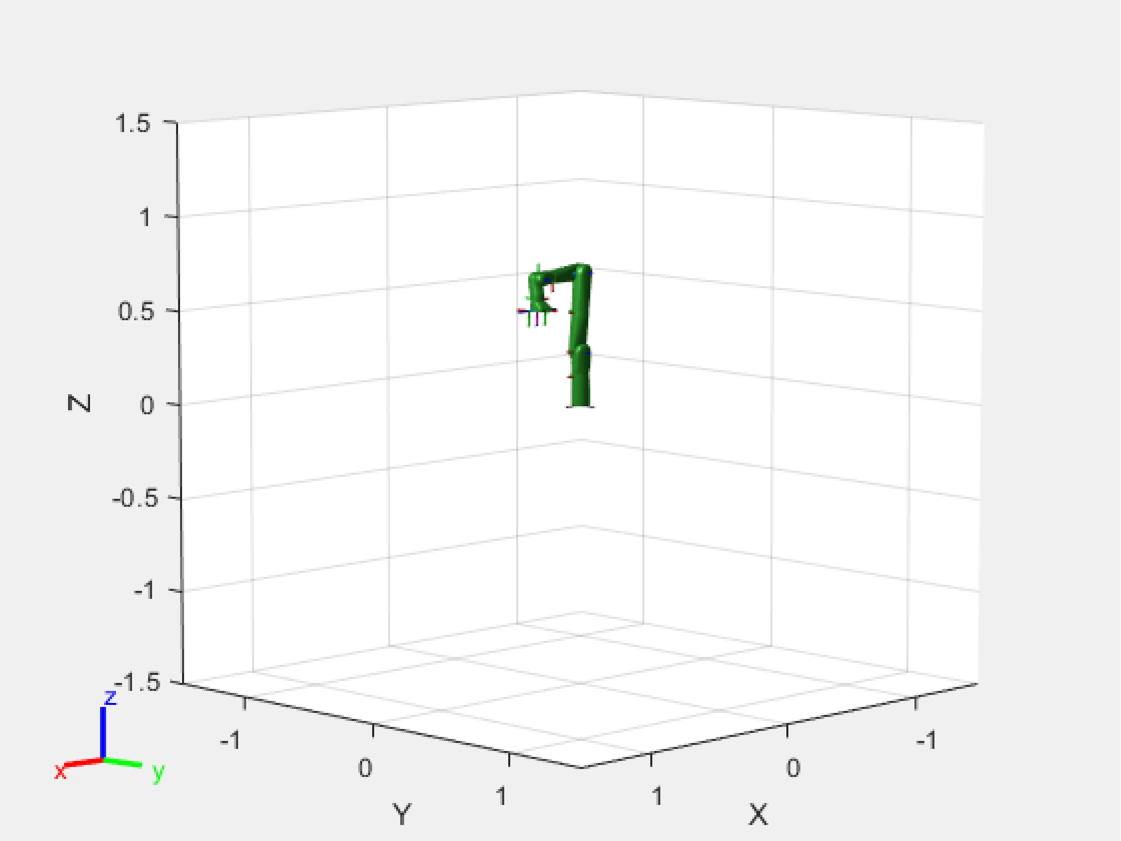
Create Planner

This example uses the `plannerBiRRT` object, which is a bi-directional variant of the RRT algorithm with the "connect" heuristic enabled. For a sparse environment, this planner finds a solution in lesser number of iterations as compared to other RRT-based planners . Alternatively, for shorter paths with trimmed edges, consider the `plannerRRTStar` object.

The robot starts in a joint configuration which satisfies the constraint. Alternatively, the `interpolate` function of the constrained state space can be used to constrain the start configuration. This ensures that all the states in the resulting plan are constrained. Set the random number seed for repeatable results.

```
rng(20);
planner = plannerBiRRT(ss, sv);
planner.MaxConnectionDistance = 0.5;
planner.EnableConnectHeuristic = true;
startConfig = [0.1196 -0.045 -0.1303 1.6528 -0.0019 1.5032 0.0083, openPostion, openPostion];
figure("Visible","on");
```

```
show(kinova, startConfig, ...
    "Visuals", "off", ...
    "Collisions", "on", ...
    "PreservePlot", true);
```



Define Grasp Region

Next, define a grasping region using the `workspaceGoalRegion` object which represents a cylinder to pick in the environment (`env{3}`). Generate a joint configuration for a sampled end-effector pose. You should ensure that the configuration passed to the planner is collision-free.

```
graspingRegion = workspaceGoalRegion(endEffectorName);

% Attach the reference frame to the pose of the cylinder object.
graspingRegion.ReferencePose = env{3}.Pose;

% Define the offset of the end-effector relative to the target orientation.
graspingRegion.EndEffectorOffsetPose = trvec2tform([0, 0, 0.07]) * targetOrientation;

% Generate goal joint configuration of the robot given a sampled end-effector pose in
% the region.
goalConfig = jointConfigurationGiven(ss,sample(graspingRegion));
goalConfig(end-1:end) = openPostion;
```

Plan Path

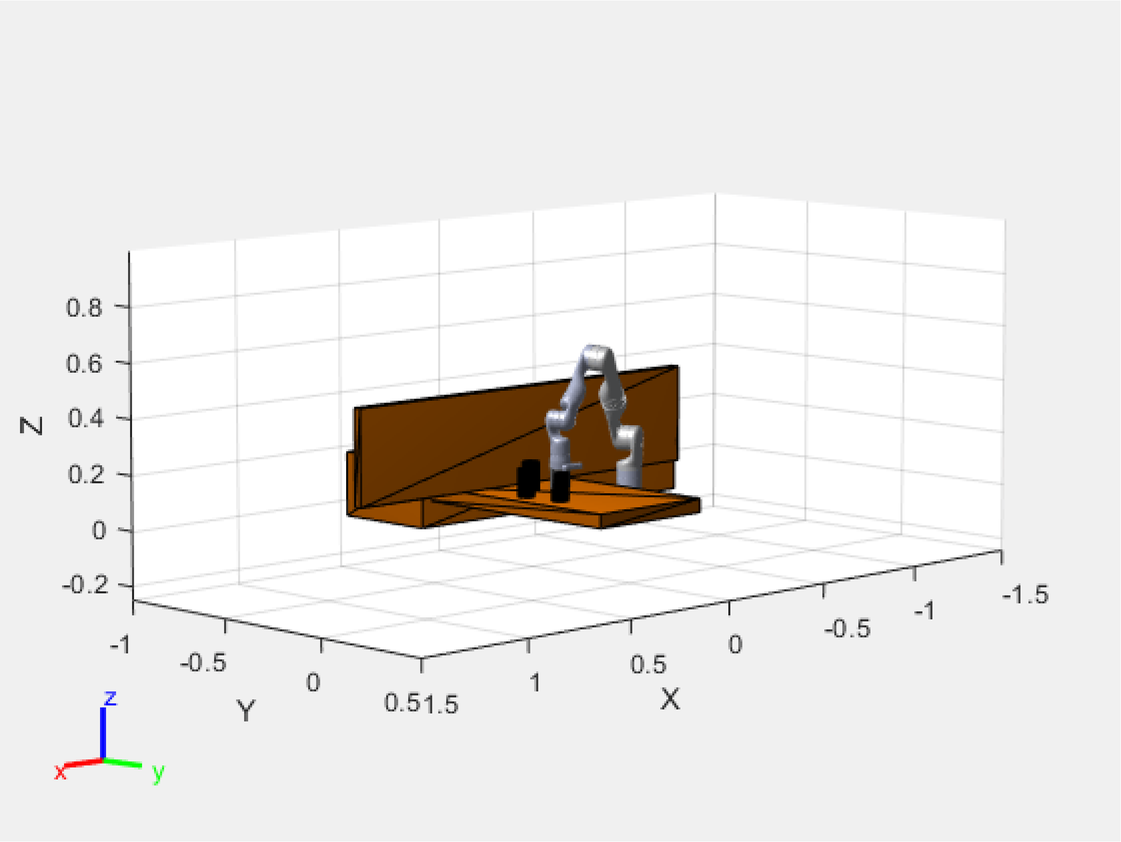
Call the `plan` function to plan a path between the start and goal configurations. Time the planning phase. Interpolate the path to 50 points.

```
tic;
pickPath = plan(planner,startConfig,goalConfig);
tOut = toc;
interpolate(pickPath,50);
fprintf(newline);
fprintf("Planning finished in %d s\n", tOut);
```

```
Planning finished in 1.786575e+00 s
```

Visualize the planned path. The robot moves to the container to grab it.

```
hold on;
ylim([-1 0.5])
zlim([-0.25 1])
for i  = 1:length(env)
    show(env{i})
end
for i = 1:size(pickPath.States, 1)
    show(kinova, pickPath.States(i, :), ...
        "Visuals", "on", ...
        "FastUpdate",true, ...
        "Frames", "off", ...
        "PreservePlot",false);
    drawnow;
end
exampleHelperConstrainedRobotMoveGripper(kinova,...
    pickPath.States(end,:),openPostion,closedPosition);
hold off;
```



Recreate Planner

Attach the collision geometry of the can to the end-effector and remove the can from the environment. Then, with the modified robot, create a planning instance with the new robot model. After planning, you could unceck `ss.EnableContraint` and rerun this script to see a path without a constrainted end effector.

```
zOffset = [0 0 (env{3}.Length)/2];
container = env{3};
addCollision(kinova.getBody(endEffectorName),env{3}, (env{3}.Pose) \ trvec2tform(zOffset));
env(3) = [];
ss = exampleHelperConstrainedStateSpace(kinova,endEffectorName,targetOrientation);

% Keep the end-effector closed during planning
ss.StateBounds(end-1:end,:) = repmat([0.04 0.04],2,1);
sv = manipulatorCollisionBodyValidator(ss, ...
    "Environment",env,"ValidationDistance",0.2);
planner = plannerBiRRT(ss,sv);
planner.MaxConnectionDistance = 1;
planner.EnableConnectHeuristic = true;


ss.EnableConstraint = ☑ ;
```

Plan Path

Next, plan from the last configuration in the previous path to a new goal joint configuration. The goal configuration places the can on the opposite table in the environment.

```
% The last state of the of the previous plan
startConfig = pickPath.States(end,:);
startConfig(end-1:end) = 0.04;
goalConfig = [1.1998 0.6044 0.0091 1.4219 -0.0058 1.1154 0.0076 0.04 0.04];
rng(10);
tic
placePath = plan(planner,startConfig,goalConfig);
tOut = toc;
fprintf("Planning finished in %0.2d s\n",tOut);
```

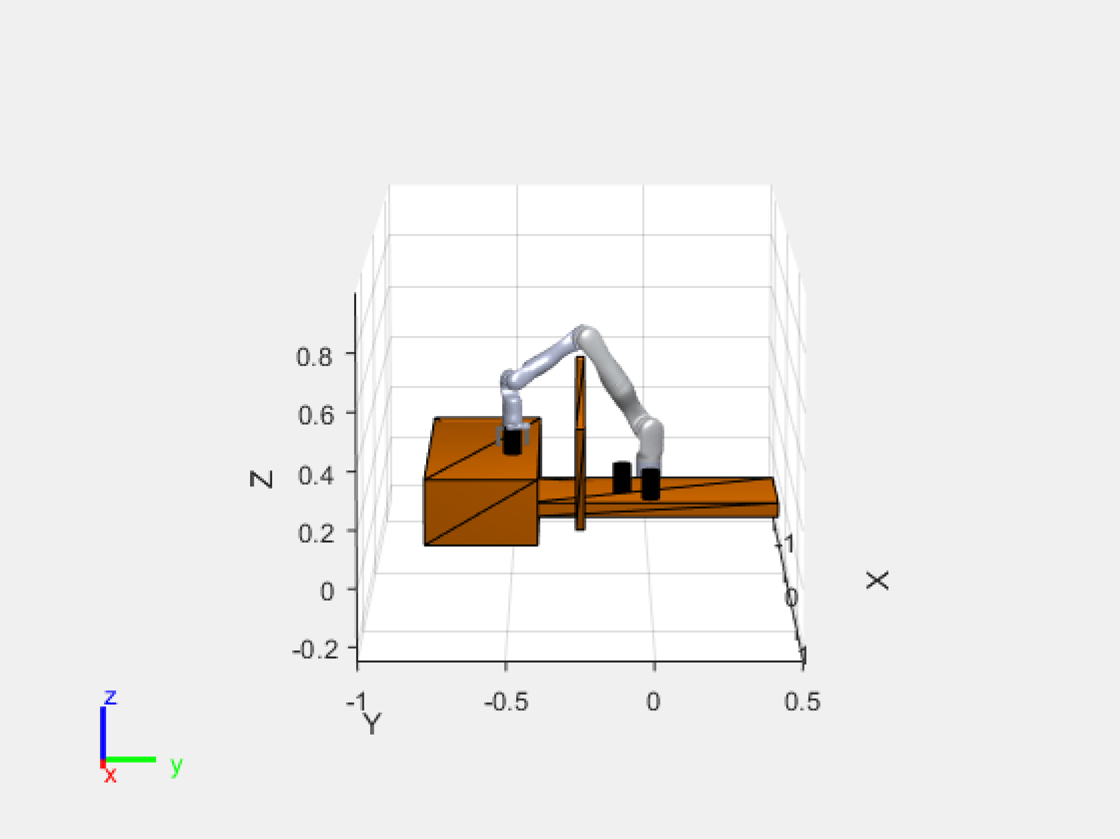
```
Planning finished in 4.07e+00 s
```

```
interpolate(placePath,50);
```

Visualize the path that places the container. Notice that the can remains upright throughout the path.

```
show(kinova, startConfig, ...
    "FastUpdate", false, ...
    "PreservePlot", false);
view([90,0,15])
ylim([-1 0.5])
zlim([-0.25 1])
hold on;
for i  = 1:length(env)
    show(env{i})
end
containerPose = hgtransform();
containerPose.Matrix = container.Pose;
[ax, p] = show(container);
```

```
p.FaceColor = [0 0 1];
p.Parent = containerPose;
for i = 1:size(placePath.States, 1)
    cPose = getTransform(kinova,placePath.States(i, :),endEffectorName);
    containerPose.Matrix = cPose/(container.Pose)*trvec2tform(zOffset);
    show(kinova,placePath.States(i,:), ...
        "FastUpdate",true, ...
        "PreservePlot",false, ...
        "Frames", "off");
    drawnow;
end
exampleHelperConstrainedRobotMoveGripper(kinova,...
    placePath.States(end,:),closedPosition,openPostion);
hold off;
```



Next Steps

To see an unconstrained path, uncheck the `ss.EnableConstraint` checkbox and rerun the script. The cup tilts in a way that could easily spill the contents. You could also consider replacing the `plannerBiRRT` object with another sampling-based planner like `plannerRRTStar` to evaluate their performance.

References

[1] D. Berenson, S. Srinivasa, D. Ferguson, A. Collet, and J. Kuffner, "Manipulation Planning with Workspace Goal Regions", in Proceedings of IEEE International Conference on Robotics and Automation, 2009, pp.1397-1403

Manipulator Shape Tracing in MATLAB and Simulink

This example shows how to trace a predefined 3-D shape in space. Following a smooth, distinct path is useful in many robotics applications such as welding, manufacturing, or inspection. A 3-D trajectory is solved in the task space for tracing the MATLAB® membrane and is executed using the Sawyer robot from Rethink Robotics®. The goal is to generate a smooth path for the end effector of the robot to follow based on a set waypoints.

The "Manipulator Shape Tracing in MATLAB and Simulink" on page 1-333 example shows how to generate a closely discretized set of segments that can then be passed to an inverse kinematics solver to be solved using an iterative solution. However, this example offers an alternate approach to reduce computational complexity. This example splits path segments into just a few discrete points and uses smoothing functions to interpolate between the waypoints. This approach should generate a smoother trajectory and improve run-time efficiency.

Load the Robot

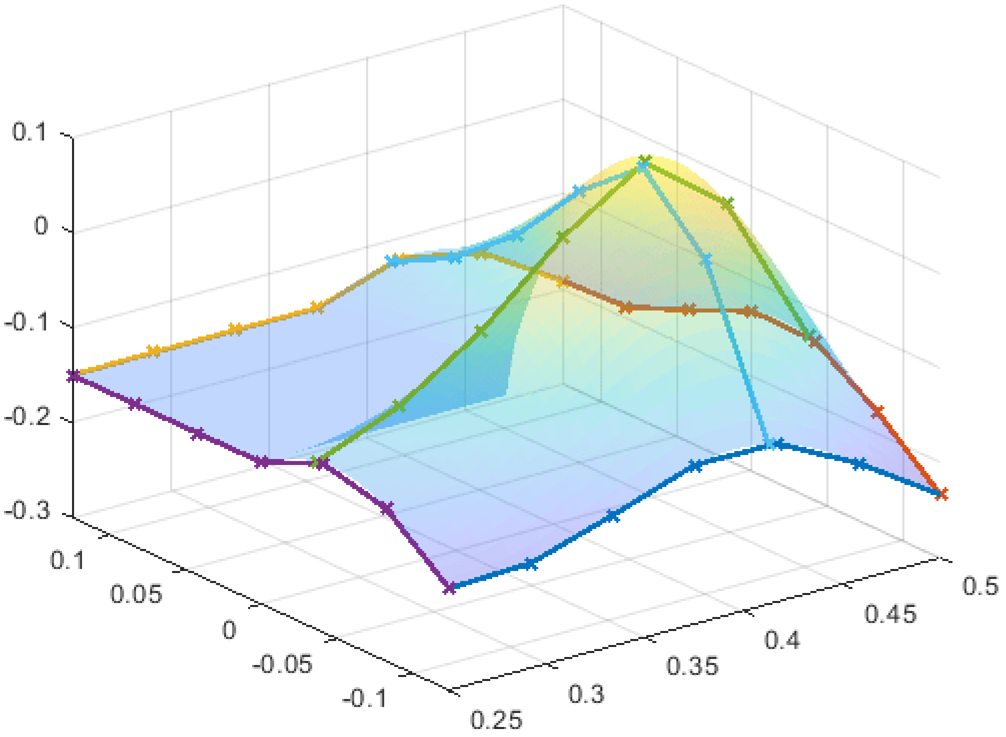
This example uses the Sawyer robot from Rethink Robotics®. Import the URDF file that specifies the rigid body dynamics. Set the `DataFormat` to use column vectors to define robot configurations. Simulink® uses column vectors. The task space limits are defined based on empirical data.

```
sawyer = importrobot('sawyer.urdf');
sawyer.DataFormat = 'column';
taskSpaceLimits = [0.25 0.5; -0.125 0.125; -0.15 0.1];
numJoints = 8; % Number of joints in robot
```

Generate a Set of Task-Space Waypoints

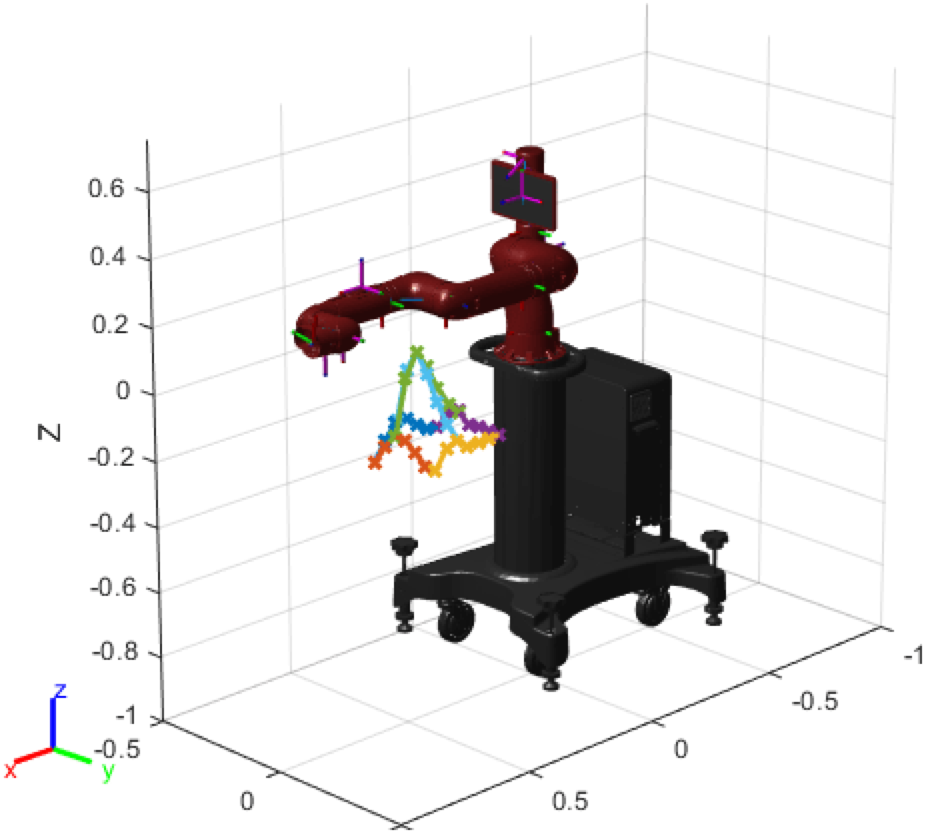
For this example, the goal is to get a set of path segments that trace the MATLAB® membrane logo. The membrane surface and the path segments are generated as cell arrays using the helper function `generateMembranePaths`. To visualize the paths overlaid on the surface, plot the surface using `surf` and the path segments by iterating through the path segment cell array. You can increase `numSamples` to sample more finely across the surace.

```
numSamples = 7;
[pathSegments, surface] = generateMembranePaths(numSamples, taskSpaceLimits);

% Visualize the output
figure
surf(surface{:},'FaceAlpha',0.3,'EdgeColor','none');
hold all
for i=1:numel(pathSegments)
    segment = pathSegments{i};
    plot3(segment(:,1),segment(:,2),segment(:,3),'x-','LineWidth', 2);
end
hold off
```



To ensure that the robot can trace the output, visualize the shape in the robot workspace. Show the `sawyer` robot and plot the line segments in the same figure.

```
figure
show(sawyer);
hold all

for i=1:numel(pathSegments)
    segment = pathSegments{i};
    plot3(segment(:,1),segment(:,2),segment(:,3),'x-','LineWidth',2);
end

view(135,20)
axis([-1 1 -.5 .5 -1 .75])
hold off
```



Create an Inverse Kinematics Solver

Create an inverse kinematics (IK) using the loaded `sawyer` rigid body tree . It is initially configured with a uniform set of weights, using the home configuration as the initial guess. Set the initial guess to the home configuration and the pose tolerances with uniform weights. The end effector for IK solver is the `'right_hand'` body of the robot.

```
ik = inverseKinematics('RigidBodyTree', sawyer);
initialGuess = sawyer.homeConfiguration;
weights = [1 1 1 1 1 1];
eeName = 'right_hand';
```

Convert Task-Space Waypoints to Joint-Space Using Inverse Kinematics

Use the inverse kinematics solver to generate a set of joint space waypoints, which give the joint configurations for the robot at each point of the generated `pathSegments`. Each joint-space segment is filed into a matrix, `jointPathSegmentMatrix`, which is passed to the Simulink model as an input.

```
% Initialize the output matrix
jointPathSegmentMatrix = zeros(length(pathSegments),numJoints,numSamples);

% Define the orientation so that the end effector is oriented down
sawyerOrientation = axang2rotm([0 1 0 pi]);

% Compute IK at each waypoint along each segment
for i = 1:length(pathSegments)
    currentTaskSpaceSegment = pathSegments{i};
    currentJointSegment = zeros(numJoints, length(currentTaskSpaceSegment));
    for j = 1:length(currentTaskSpaceSegment)
        pose = [sawyerOrientation currentTaskSpaceSegment(j,:)'; 0 0 0 1];
        currentJointSegment(:,j) = ik(eeName,pose,weights,initialGuess);
        initialGuess = currentJointSegment(:,j);
    end

    jointPathSegmentMatrix(i, :, :) = (currentJointSegment);
end
```

Load Simulink Model

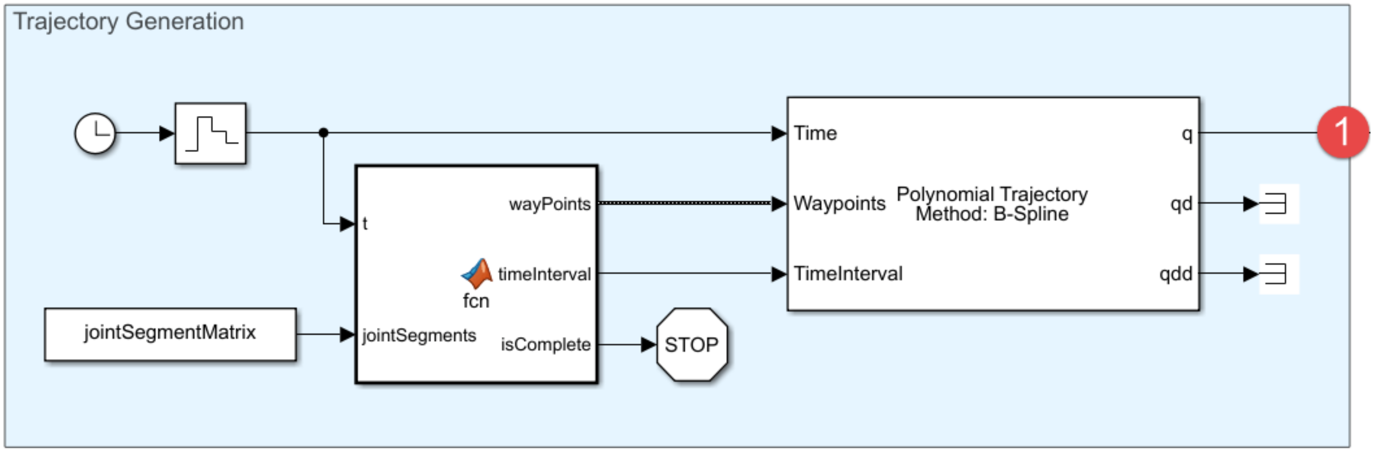
Use the `shapeTracingSawyer` model to execute the trajectories and simulate them on a kinematic model of the robot.

```
open_system("shapeTracingSawyer.slx")
```

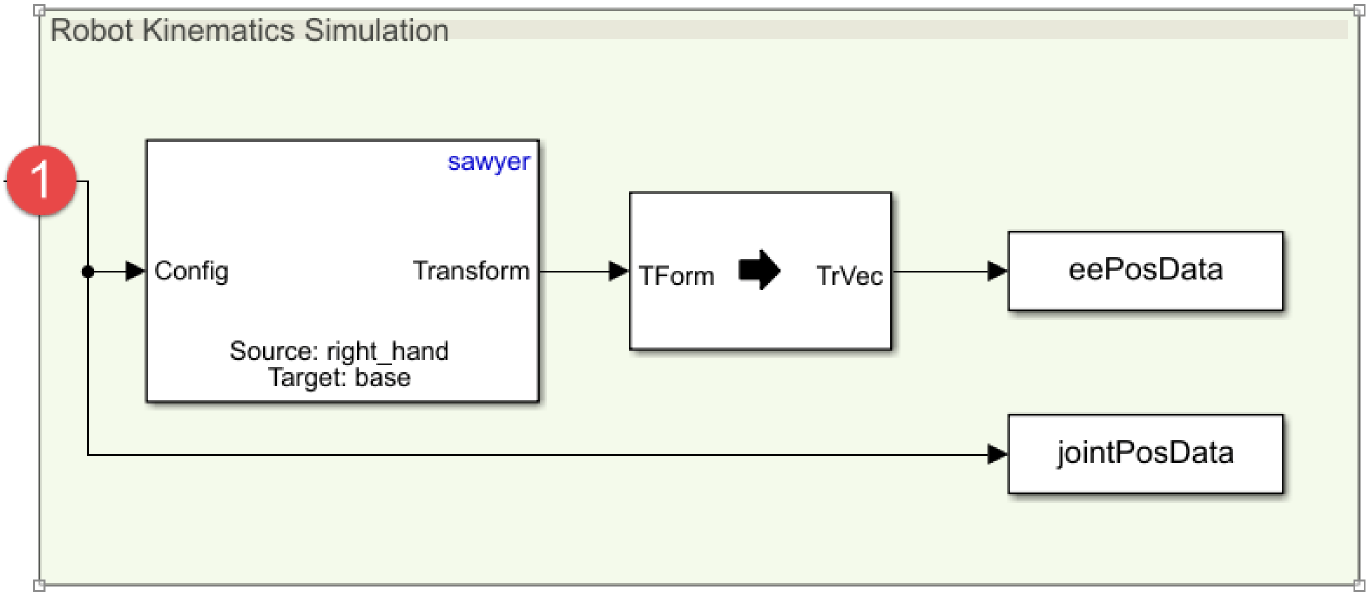
The Simulink model has two main parts:

1. The **Trajectory Generation** section takes the matrix of joint-space path segments, jointPathSegmentMatrix, and converts the segments to a set of discretized joint-space waypoints (joint configurations) at each time step in the simulation using a MATLAB function block. The **Polynomial Trajectory Block** converts the set of joint configurations to a smoothed joint space B-spline trajectory in time.
2. The **Robot Kinematics Simulation** section accepts the joint-space waypoints from the smoothed trajectory and computes the resulting end-effector position for the robot.

Trajectory Generation



Robot Kinematics Simulation



Execute Joint-Space Trajectories in Simulink

Simulate the model to execute the generate trajectories.

```
sim("shapeTracingSawyer.slx")
```

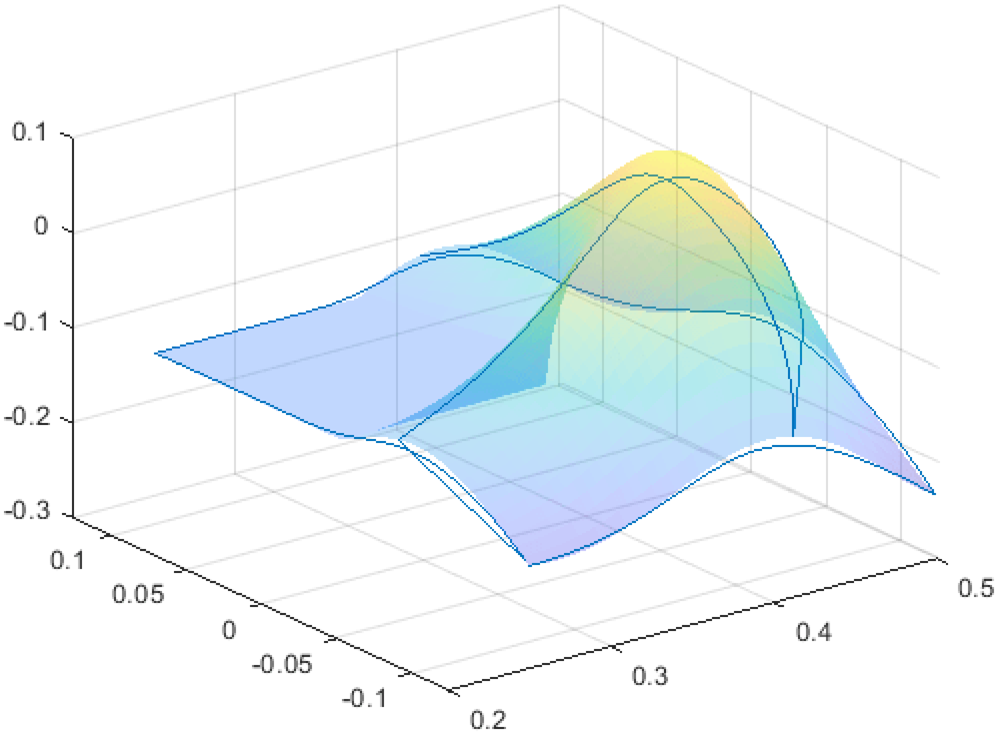
View Trajectory Generation Results

The model outputs the robot joint configurations and the end-effector positions along each smoothed path trajectory. To work easily with MATLAB plotting tools, reshape the data.

```
% End effector positions
xPositionsEE = reshape(eePosData.Data(1,:,:),1,size(eePosData.Data,3));
yPositionsEE = reshape(eePosData.Data(2,:,:),1,size(eePosData.Data,3));
zPositionsEE = reshape(eePosData.Data(3,:,:),1,size(eePosData.Data,3));

% Extract joint-space results
jointConfigurationData = reshape(jointPosData.Data, numJoints, size(eePosData.Data,3));
```

Plot the new end-effector positions on the original membrane surface.

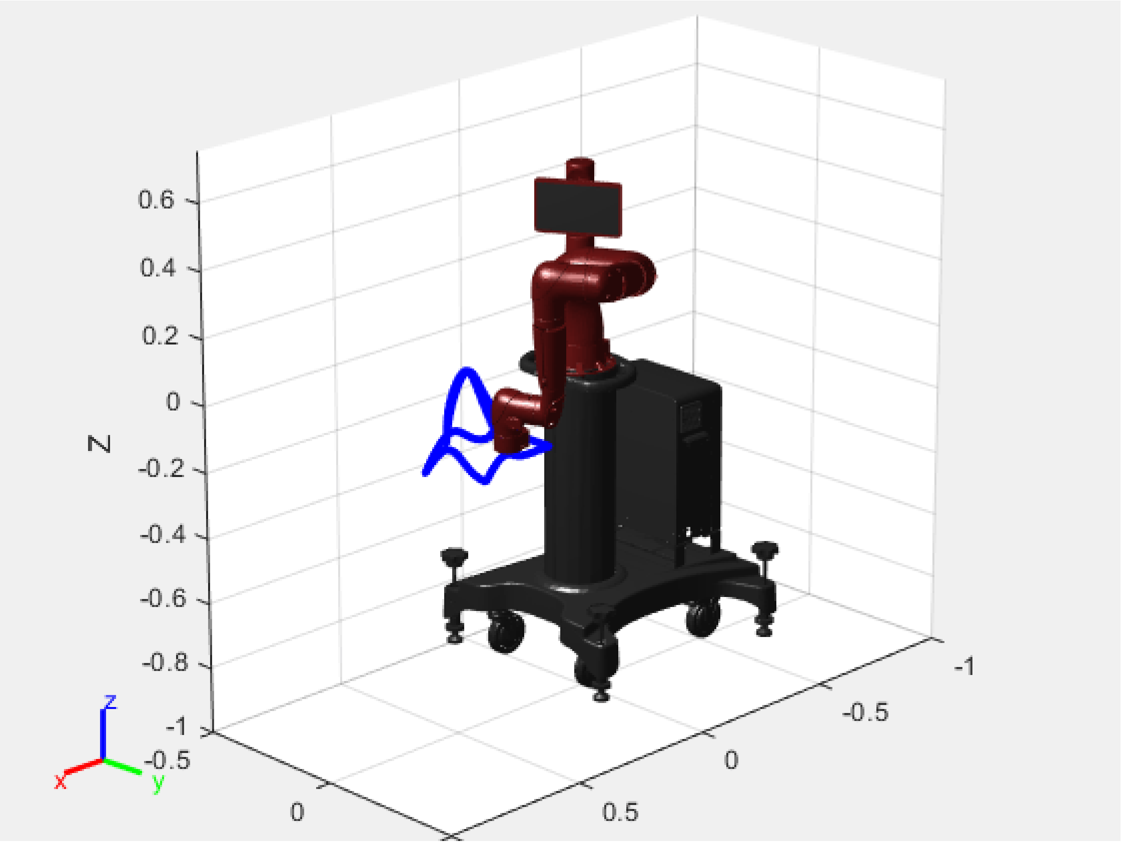
```
figure
surf(surface{:},'FaceAlpha',0.3,'EdgeColor','none');
hold all
plot3(xPositionsEE,yPositionsEE,zPositionsEE)
grid on
hold off
```



Visualize Robot Motion

In addition to the visualization above, the tracing behavior can be recreated using the Sawyer robot model. Iterate through the joint configurations in `jointConfigurationData` to visualize the robot using `show` and continuously plot the end-effector position in 3-D space.

```
% For faster visualization, only display every few steps
vizStep = 5;

% Initialize a new figure window
figure
set(gcf,'Visible','on');

% Iterate through all joint configurations and end-effectort positions
for i = 1:vizStep:length(xPositionsEE)
    show(sawyer, jointConfigurationData(:,i),'Frames','off','PreservePlot',false);
    hold on
    plot3(xPositionsEE(1:i),yPositionsEE(1:i),zPositionsEE(1:i),'b','LineWidth',3)

    view(135,20)
    axis([-1 1 -.5 .5 -1 .75])

    drawnow
end
hold off
```


0.6
0.4
0.2
0
Z
-0.2
-0.4
-0.6
-0.8
-1
-0.5
0
0.5
0
-0.5
-1
z
x
y

Generate Code for Manipulator Motion Planning in Perceived Environment

This example shows how to generate code for planning manipulator motion in a perceived environment. Perceived environments can have a variable number of collision objects that can be a combination of heterogeneous types (spheres, cylinders, meshes, and boxes). This example uses the `convertToCollisionMesh` function to homogenize the cell array of collision objects by converting the primitive type objects to their mesh equivalents.

Set Up Robot and Environment

In this example, you generate a MEX function for a MATLAB function that uses a `manipulatorRRT` object to plan for a Kinova Gen 3 robot.

Load Robot Model

Create a rigid body tree object to model the robot. For this example, load a Kinova Gen3 manipulator.

```
kinova = loadrobot("kinovaGen3",DataFormat="row");
```

Create Environment

Real perception subsystems can output a structure that specifies the type of perceived geometry, its dimensions, and the pose of the object in the base frame of the robot. To simulate this, first create a structure to contain collision objects.

```
geomStructType = struct("Type",exampleHelperCollisionEnum.Box, ...
                  "X",0, ...
                  "Y",0, ...
                  "Z",0, ...
                  "Vertices",coder.typeof(zeros(3),[inf 3],[1 0]), ...
                  "Radius",0, ...
                  "Height",0, ...
                  "Pose",eye(4));
geomStruct = geomStructType;
geomStruct.Vertices=zeros(3);
```

Next create a cylinder, a box, and two spheres, as perceived by a perception subsystem.

Use `geomStruct` as the framework to create a perceived box with `X`, `Y`, and `Z` lengths of `1`, `1`, and `0.1`, respectively.

```
boxGeom= geomStruct;
boxGeom.Type = exampleHelperCollisionEnum.Box;
boxGeom.X = 1;
boxGeom.Y = 1;
boxGeom.Z = 0.1;
boxGeom.Pose = trvec2tform([0 0 -0.051]);
```

Create a perceived cylinder of radius `1` and height `0.1`.

```
cylinderGeom = geomStruct;
cylinderGeom.Type=exampleHelperCollisionEnum.Cylinder;
cylinderGeom.Radius=0.1;
cylinderGeom.Height=1.0;
cylinderGeom.Pose=trvec2tform([0.5 0.3 0.50]);
```

Create two perceived spheres, each with a radius of `0.1`.

```
sphere1Geom = geomStruct;
sphere1Geom.Type=exampleHelperCollisionEnum.Sphere;
sphere1Geom.Radius=0.1;
sphere1Geom.Pose=trvec2tform([0.4 0.4 0.4]);
sphere2Geom = geomStruct;
sphere2Geom.Type=exampleHelperCollisionEnum.Sphere;
sphere2Geom.Radius=0.1;
sphere2Geom.Pose=trvec2tform([-0.4 0.4 0.4]);
```

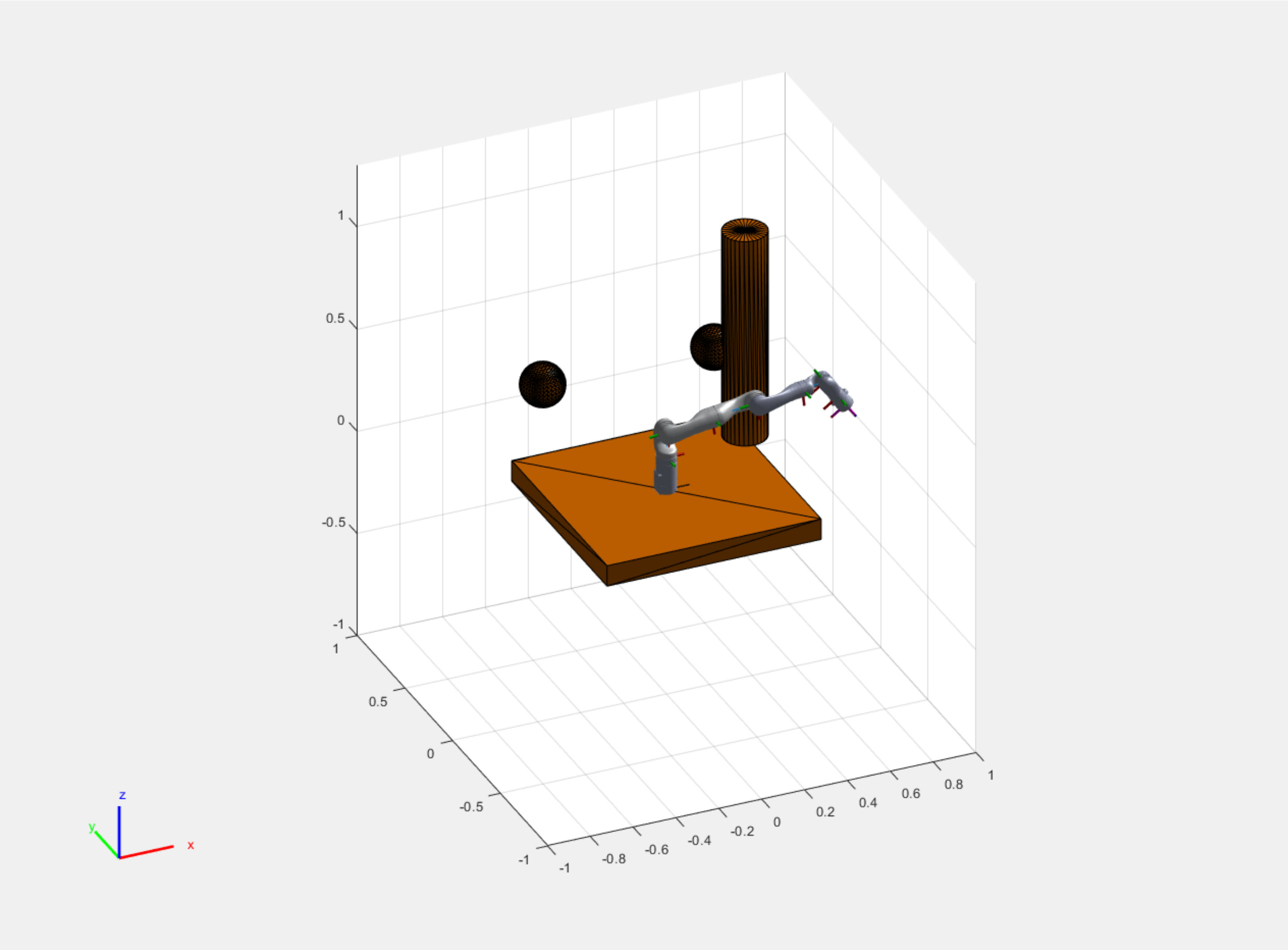
Set Up Planning Variables

The, `exampleHelperVariableHeterogeneousPlanner` helper function accepts the start and goal joint configurations of the robot, as well as an array of structure elements representing the environment. In this example, the start and goal joint configurations are set arbitrarily, but you can set them as necessary.

```
startConfig = [-0.0035 1.4453 0.0334 0.0144 -0.0294 1.0600 -0.0077];
goalConfig = [-1.1519 1.5243 0.3186 0.0009 0.2466 1.2090 -1.1952];
env1 = [boxGeom cylinderGeom sphere1Geom sphere2Geom];
```

Visualize the start configuration of the robot in the perceived environment `env1`.

```
figure(Name="Env1",Visible="on",Units="normalized",OuterPosition=[0 0 1 1])
exampleHelperVisualizeVarSizeEnvironment(env1,kinova,startConfig);
```



Generate Code for Planning

In order to generate a MEX function, `exampleHelperVariableHeterogeneousPlanner_mex`, from the MATLAB function `exampleHelperVariableHeterogeneousPlanner`, run this command:

```
codegen("exampleHelperVariableHeterogeneousPlanner", ...
    "-args",{startConfig,goalConfig,coder.typeof(geomStruct,[1 inf],[0 1])})
```

The `variable_dims` argument of `coder.typeof` (Fixed-Point Designer) specifies that the input row vector `geomStruct` of the entry-point planning function can have an unbounded and variable-sized second dimension.

Plan Collision-Free Path

Next, use the planner entry-point helper function to output a geometric collision-free plan.

```
planInEnv1=exampleHelperVariableHeterogeneousPlanner(startConfig,goalConfig,env1);
```

You can also reuse the `exampleHelperVariableHeterogeneousPlanner` for a different perceived environment. Plan in a new environment, `env2`, created by removing `cylinderGeom` and `sphereGeom` from `env1`.

```
env2=[boxGeom,sphere2Geom];
planInEnv2=exampleHelperVariableHeterogeneousPlanner(startConfig,goalConfig,env2);
```

MEX function for planning in `env2`, and do not need to generate it again. It has the same behavior `exampleHelperVariableHeterogeneousPlanner`, but results in shorter planning times.

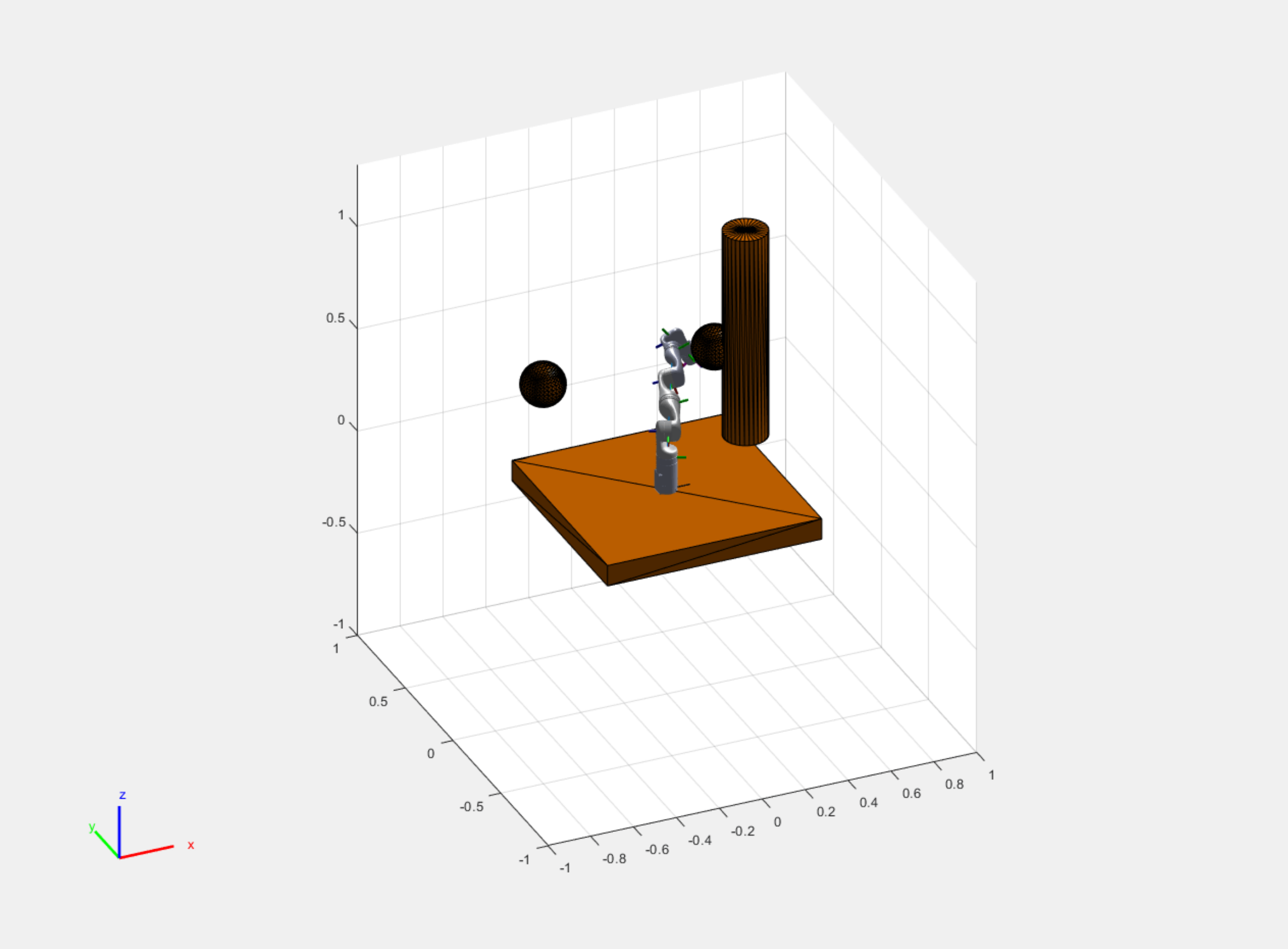
To use the MEX function instead, run this code:

```
planInEnv1 = exampleHelperVariableHeterogeneousPlanner_mex(startConfig,goalConfig,env1);
env2 = [boxGeom sphere2Geom];
planInEnv2 = exampleHelperVariableHeterogeneousPlanner_mex(startConfig,goalConfig,env2);
```

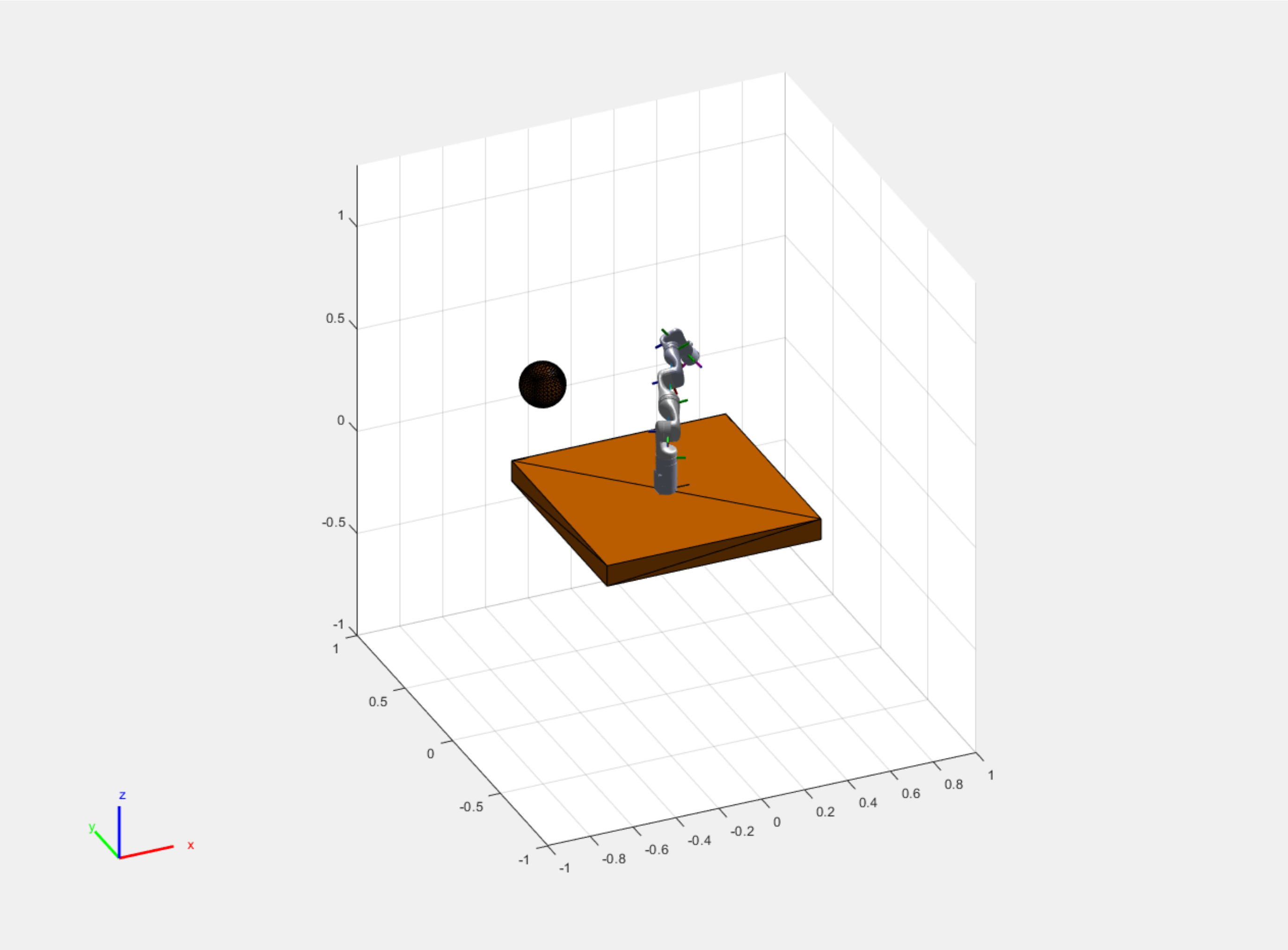
Visualize Planned Path

Visualize the planned output in both `env1` and `env2.`

```
figure(Name="Planned path in env1",Visible="on",Units="normalized",OuterPosition=[0,0,1,1])
exampleHelperVisualizeVarSizeEnvironment(env1,kinova,planInEnv1);
```



```
figure(Name="Planned path in env2",Visible="on",Units="normalized",OuterPosition=[0,0,1,1])
exampleHelperVisualizeVarSizeEnvironment(env2,kinova,planInEnv2);
```



Supporting Functions

Planner Entry Point Function

exampleHelperVariableHeterogeneousPlanner plans motion in the perceived environment of a Kinova Gen 3 robot workspace. The helper function accepts the perceived environment as the geomStructs argument, which is an array of structures where each element contains the information used to represent a collision object.

The input structure array geomStructs is variable in size and consists of structure elements that each represent the collision geometry defined by the Type field of the structure. The Type field stores an enumeration, exampleHelperCollisionEnum, specifiying whether the perceived object is a Box, Sphere, Cylinder or Mesh. For more information on how to construct a collision object structure, see Create Environment on page 1-0 .

The second manipulatorRRT input argument can hold up to 100 objects. Upper bounding is necessary to enable construction of collision objects collisionBox, collisionCylinder, collisionSphere, collisionMesh inside a for loop.

```
function interpolatedPlan = exampleHelperVariableHeterogeneousPlanner(start,goal,geomStructs)
%This function is for internal use only and may be removed in the future.
```

```
%exampleHelperVariableHeterogeneousPlanner Plan in a perceived environment of a Kinova Gen 3 robo
%   INTEPROLATEDPLAN=exampleHelperVariableHeterogeneousPlanner(START,GOAL,GEOMSTRUCTS)
%   Outputs a collision free geometric plan, INTEPROLATEDPLAN, of a Kinova
%   Gen3 robot in an environment defined by GEOMSTRUCTS which is a variable
%   sized array of struct elements that capture the information of a
%   collision geometry in the environment. Each struct element in
%   GEOMSTRUCTS is of the form:
%       geomStruct=struct("Type",exampleHelperCollisionEnum.Box,...
%                 "X",0, ...
%                 "Y",0, ...
%                 "Z",0, ...
%                 "Vertices",zeros(3),...
%                 "Radius",0, ...
%                 "Height",0, ...
%                 "Pose",eye(4));
%   START and GOAL are the start and goal joint configurations of the
%   robot, respectively, and are specified as a row vector.

%Copyright 2021 The MathWorks, Inc.

    % Create a placeholder variable for environment which is a cell-array
    % of collision meshes with vertices that are variably sized. The size
    % of the environment is that of the input array of collision geometry
    % struct elements.
    coder.varsize("vertices",[inf 3],[1 0]);
    vertices = zeros(3);
    env = repmat({collisionMesh(vertices)},1,length(geomStructs));

    % Load the rigid body tree for which the planner will be defined.
    rbt = loadrobot("kinovaGen3",DataFormat="row");

    % Set up the environment. The maximum number of collision objects that
    % the environment can hold is 100.
    for i = coder.unroll(1:100)
        if (i <= length(geomStructs))

            % For each struct element, create the corresponding collision
            % object (collisonBox, collisionCylinder, collisionSphere, or
            % collisionMesh) and convert that to its corresponding mesh
            % equivalent thereby homogenizing the environment.
            s = geomStructs(i);
            if s.Type == exampleHelperCollisionEnum.Box
                env{i} = convertToCollisionMesh( ...
                    collisionBox(s.X,s.Y,s.Z));
            elseif s.Type == exampleHelperCollisionEnum.Sphere
                env{i} = convertToCollisionMesh( ...
                    collisionSphere(s.Radius));
            elseif s.Type == exampleHelperCollisionEnum.Cylinder
                env{i} = convertToCollisionMesh( ...
                    collisionCylinder(s.Radius,s.Height));
            elseif s.Type == exampleHelperCollisionEnum.Mesh
                env{i} = collisionMesh(s.Vertices);
            end

            % Assign the pose of the element.
            env{i}.Pose = s.Pose;
        end
    end
    % Create and set up the planner from the rigid body tree and
    % environment.
    planner = manipulatorRRT(rbt,env);
    planner.MaxConnectionDistance=0.4;
    planner.ValidationDistance=0.05;

    % For repeatable results, seed the random number generator and store
    % the current seed value.
    prevseed = rng(0);

    % Plan and interpolate.
    planOut = planner.plan(start,goal);
    interpolatedPlan = planner.interpolate(planOut);

    % Restore the random number generator to the previously stored seed.
    rng(prevseed);
end
```

Generate Code for Inverse Kinematics Computation Using Robot from Robot Library

This example shows how to perform code generation to compute Inverse Kinematics (IK) using robots from the robot library. For this example, you can use an `inverseKinematics` object with an included `rigidBodyTree` robot model using `loadrobot` to solve for robot configurations that achieve a desired end-effector position.

A circular trajectory is created in a 2-D plane and given as points to the generated MEX inverse kinematics solver. The solver computes the required joint positions to achieve this trajectory. Finally, the robot is animated to show the robot configurations that achieve the circular trajectory.

Write Algorithm to Solve Inverse Kinematics

Create a function, `ikCodegen`, that runs the inverse kinematics algorithm for a KINOVA® Gen3 robot model created using `loadrobot`.

```
function qConfig = ikCodegen(endEffectorName,tform,weights,initialGuess)
    %#codegen
    robot = loadrobot("kinovaGen3","DataFormat","row");
    ik = inverseKinematics('RigidBodyTree',robot);
    [qConfig,~] = ik(endEffectorName,tform,weights,initialGuess);
end
```

The algorithm acts as a wrapper for a standard inverse kinematics call. It accepts standard inputs, and returns a robot configuration solution vector. Since you cannot use a handle object as the input or output to a function that is supported for code generation. Load the robot inside the function. Save the `ikCodegen` function in your current folder.

Verify Inverse Kinematics Algorithm in MATLAB

Verify the IK algorithm in MATLAB before generating code.

Load a predefined KINOVA® Gen3 robot model as `rigidBodyTree` object. Set the data format to `"row"`.

```
robot = loadrobot("kinovaGen3","DataFormat","row");
```

Show details of the robot.

```
showdetails(robot)
```

```
--------------------
Robot: (8 bodies)

 Idx               Body Name         Joint Name         Joint Type               Parent Name(Idx
 ---               ---------         ----------         ----------               ---------------
   1           Shoulder_Link          Actuator1           revolute                   base_link(0
   2           HalfArm1_Link          Actuator2           revolute               Shoulder_Link(1
   3           HalfArm2_Link          Actuator3           revolute               HalfArm1_Link(2
   4            ForeArm_Link          Actuator4           revolute               HalfArm2_Link(3
   5             Wrist1_Link          Actuator5           revolute                ForeArm_Link(4
   6             Wrist2_Link          Actuator6           revolute                 Wrist1_Link(5
   7           Bracelet_Link          Actuator7           revolute                 Wrist2_Link(6
   8        EndEffector_Link        Endeffector              fixed               Bracelet_Link(7
--------------------
```

Specify the end-effector name, the weights for the end-effector transformation, and the initial joint positions.

```
endEffectorName = 'EndEffector_Link';
weights = [0.25 0.25 0.25 1 1 1];
initialGuess = [0 0 0 0 0 0 0];
```

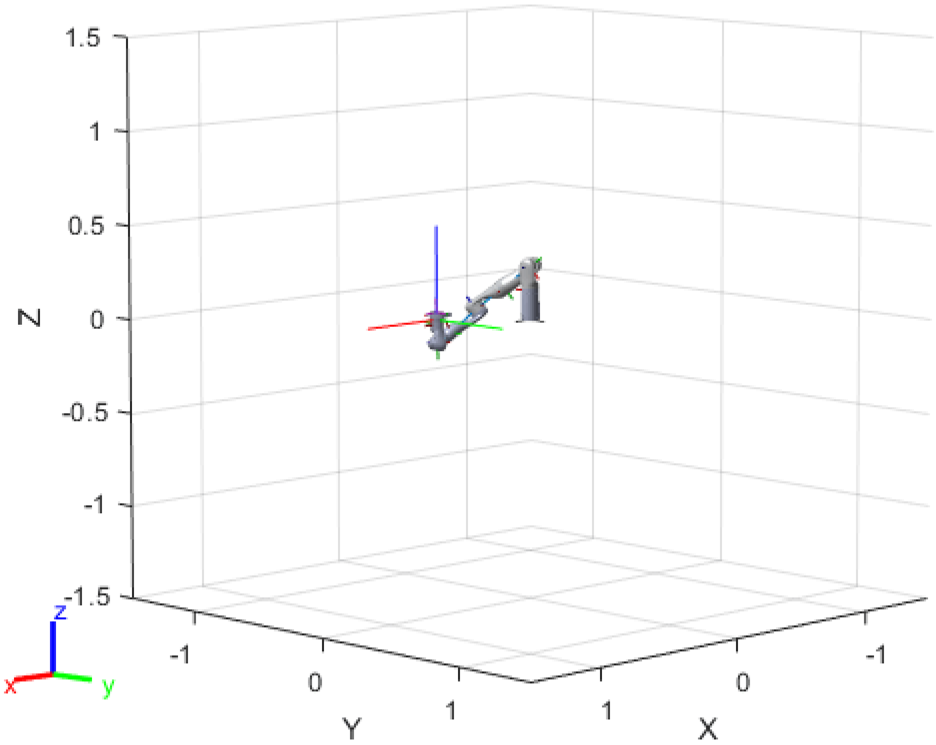
Call the inverse kinematics solver function for the specified end-effector transformation.

```
targetPose = trvec2tform([0.35 -0.35 0]);
qConfig = ikCodegen(endEffectorName,targetPose,weights,initialGuess)
```

```
qConfig = 1×7

    1.3085    2.2000   -1.3011    1.0072   -1.1144    2.0500   -3.2313
```

Visualize the robot with the computed robot configuration solution.

```
figure;
show(robot,qConfig);
hold all
plotTransforms(tform2trvec(targetPose),tform2quat(targetPose),"FrameSize",0.5);
```



Generate Code for Inverse Kinematics Algorithm

You can use either the codegen (MATLAB Coder) function or the MATLAB Coder (MATLAB Coder) app to generate code. For this example, generate a MEX file by calling codegen at the MATLAB command line. Specify sample input arguments for each input to the function using the `-args` input argument.

Call the `codegen` function and specify the input arguments in a cell array. This function creates a separate `ikCodegen_mex` function to use. You can also produce C code by using the options input argument. This step can take some time.

```
codegen ikCodegen -args {endEffectorName,targetPose,weights,initialGuess}
```

```
Code generation successful.
```

Verify Results Using Generated MEX Function

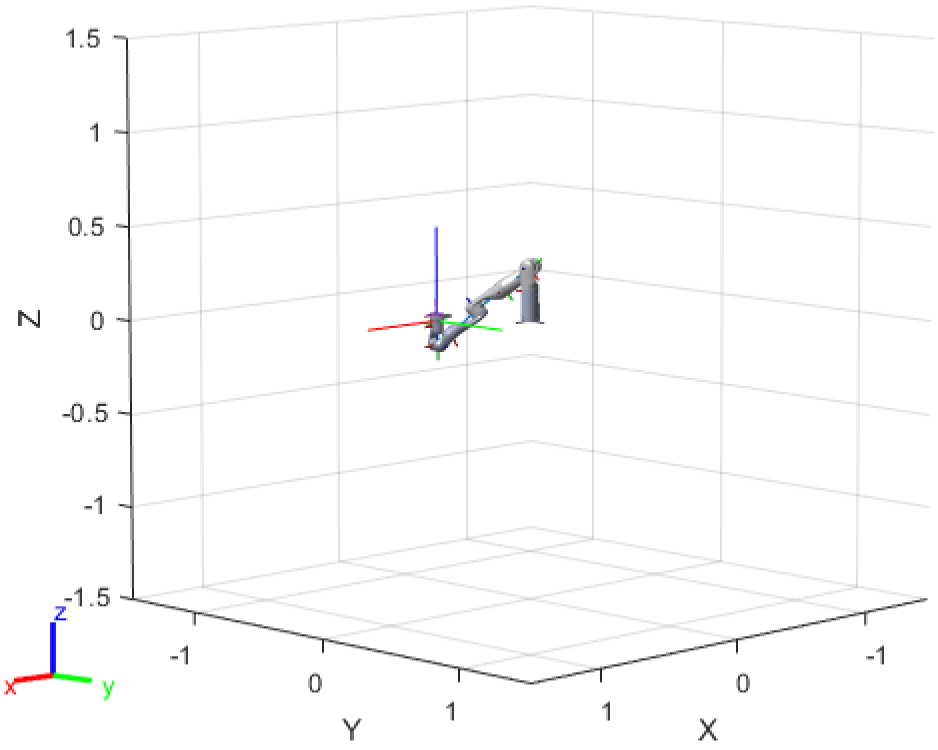
Call the MEX version of the IK solver for the specified transform.

```
targetPose = trvec2tform([0.35 -0.35 0]);
qConfig = ikCodegen_mex(endEffectorName,targetPose,weights,initialGuess)
```

```
qConfig = 1×7

    1.3084    2.2000   -1.2999    1.0092    2.0277   -2.0500   -0.0872
```

Visualize the robot with the robot configuration computed using the MEX version of the IK solver.

```
figure;
show(robot,qConfig);
hold all
plotTransforms(tform2trvec(targetPose),tform2quat(targetPose),"FrameSize",0.5);
```



Compute Inverse Kinematics with MEX function

Use the generated MEX function to compute the Inverse Kinematics solution to achieve a trajectory.

Define Trajectory

Create a circular trajectory.

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

Inverse Kinematics Solution

Preallocate configuration solutions as a matrix `qs`. Specify the weights for the end-effector transformation and the end-effector name.

```
q0 = [0 0 0 0 0 0 0];
ndof = length(q0);
qs = zeros(count,ndof);
weights = [0 0 0 1 1 1];
endEffector = 'EndEffector_Link';
```

Loop through the trajectory of points to trace the circle. Use the `ikCodegen_mex` function to calculate the solution for each point to generate the joint configuration that achieves the end-effector position. Store the configurations for later use.

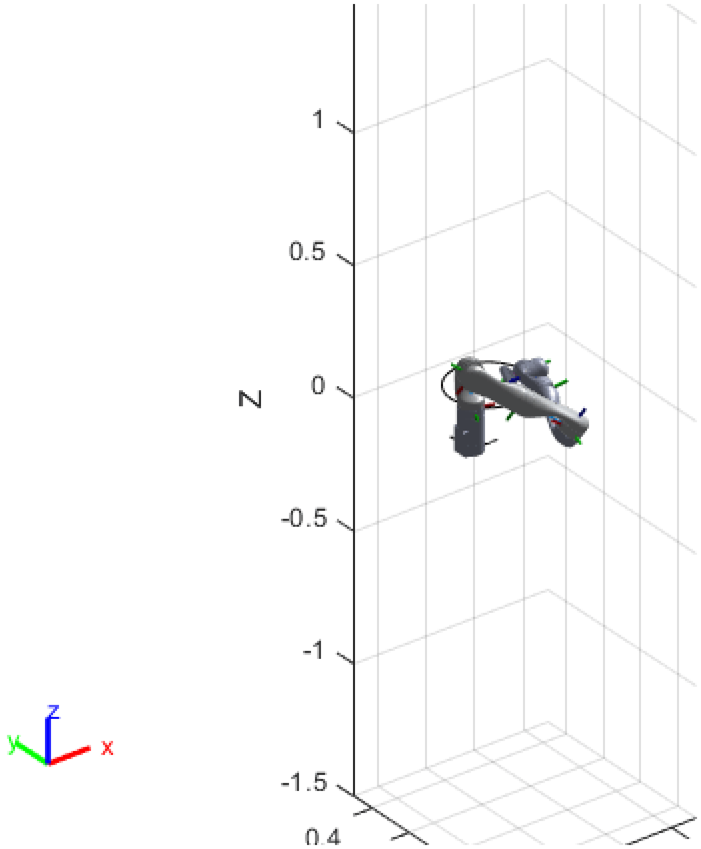
```
qInitial = q0; % Use home configuration as the initial guess
for i = 1:count
    % Solve for the configuration satisfying the desired end-effector
    % position
    point = points(i,:);
    qSol = ikCodegen_mex(endEffector,trvec2tform(point),weights,qInitial);
    % Store the configuration
    qs(i,:)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

Animate Solution

Once you generate all the solutions, animate the results. You must recreate the robot because it was originally defined inside the function. Iterate through all the solutions. Set the `"FastUpdate"` option of the `show` method to `true` to get a smooth animation.

```
robot = loadrobot("kinovaGen3","DataFormat","row");
% Show first solution and set view.
figure
show(robot,qs(1,:));
view(3)
ax = gca;
ax.Projection = 'orthographic';
hold on
plot(points(:,1),points(:,2),'k')
axis([-0.1 0.7 -0.3 0.5])
```

```
% Iterate through the solutions
framesPerSecond = 15;
r = rateControl(framesPerSecond);
for i = 1:count
    show(robot,qs(i,:),'PreservePlot',false,"FastUpdate",true);
    drawnow
    waitfor(r);
end
```



Design Trajectory with Velocity Limits Using Trapezoidal Velocity Profile

This example shows how to use the trapezoidal velocity profile to design a trajectory with input bounds rather than parameters.

The `trapveltraj` function creates trajectories with trapezoidal velocity profiles. These trajectories follow a three-segment path of acceleration, constant velocity, and deceleration between all of their waypoints. You can alter these trajectories by specifying parameters to construct your desired profile, but there are many applications require a set of bounds instead, such as limits on acceleration or velocity. In such applications, you must first translate these bounds into a parameterized trajectory that is both feasible and satisfies the expected bounds. In this example, use the `helperProfileForMaxVel` helper function to create feasible trapezoidal profiles given a velocity bound.

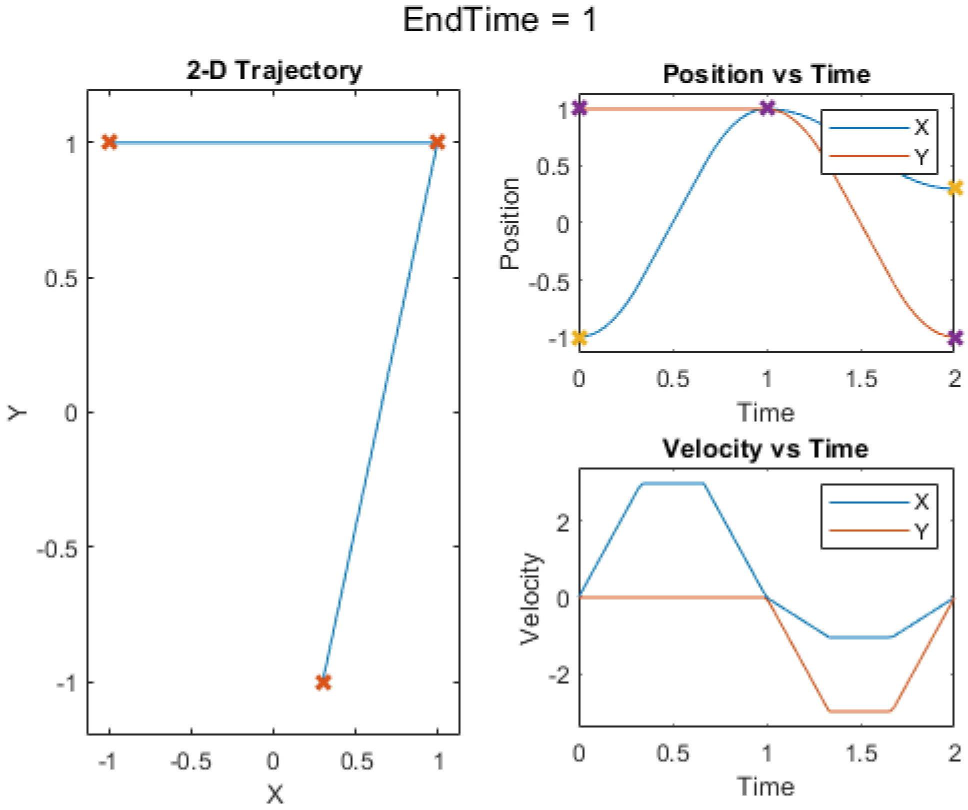
Basic `trapveltraj` Usage

If only the acceleration, constant velocity, and deceleration are known, you can create a trapezoidal velocity profile trajectory by interpolating the waypoints along each dimension using the specified parameters.

```
% Define a set of 2-D waypoints and connect them using a trapezoidal
% profile where each segment has a duration of 1 second
wpts = [-1 1 .3; 1 1 -1];
[q,qd,qdd,t] = trapveltraj(wpts,100,EndTime=1);
```

Visualize the results by using the `helperPlotTaskSpaceTraj` helper function, which creates a figure that plots the 2-D trajectory on the left and the position and velocity with respect to time on the right. Since all of the end times match, this trajectory hits both its vertical and horizontal waypoints simultaneously, making it suitable for numerous applications.

```
tpts = 0:size(wpts,2)-1;
helperPlotTaskSpaceTraj("EndTime = 1",t,q,qd,wpts,tpts);
```

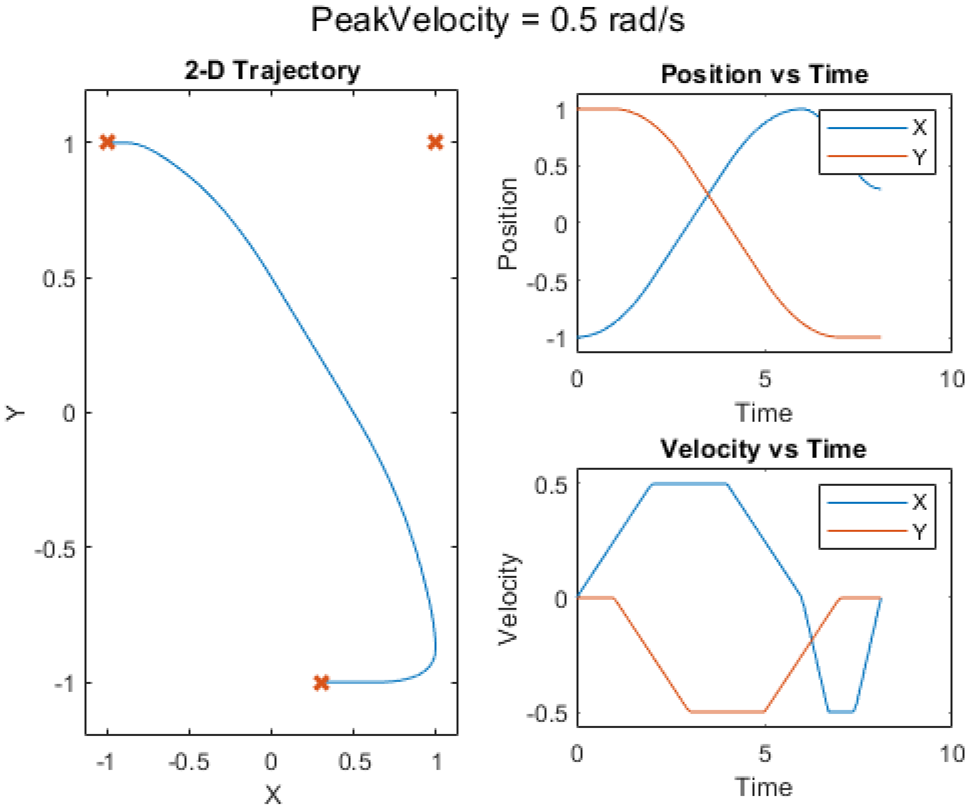


Determine Limitations Using Profile Parameters as Bounds

Try to create a trajectory for a two-degree-of-freedom (2-DoF) robot where the maximum joint velocity is 0.5 rad/s, using the same waypoints. Specify the peak velocity of the trajectory as `0.5`. This sets a specified velocity to achieve in each segment of the trajectory, and because the distances between waypoints can vary across dimensions, each dimension may have a different segment length. This can create unintended results.

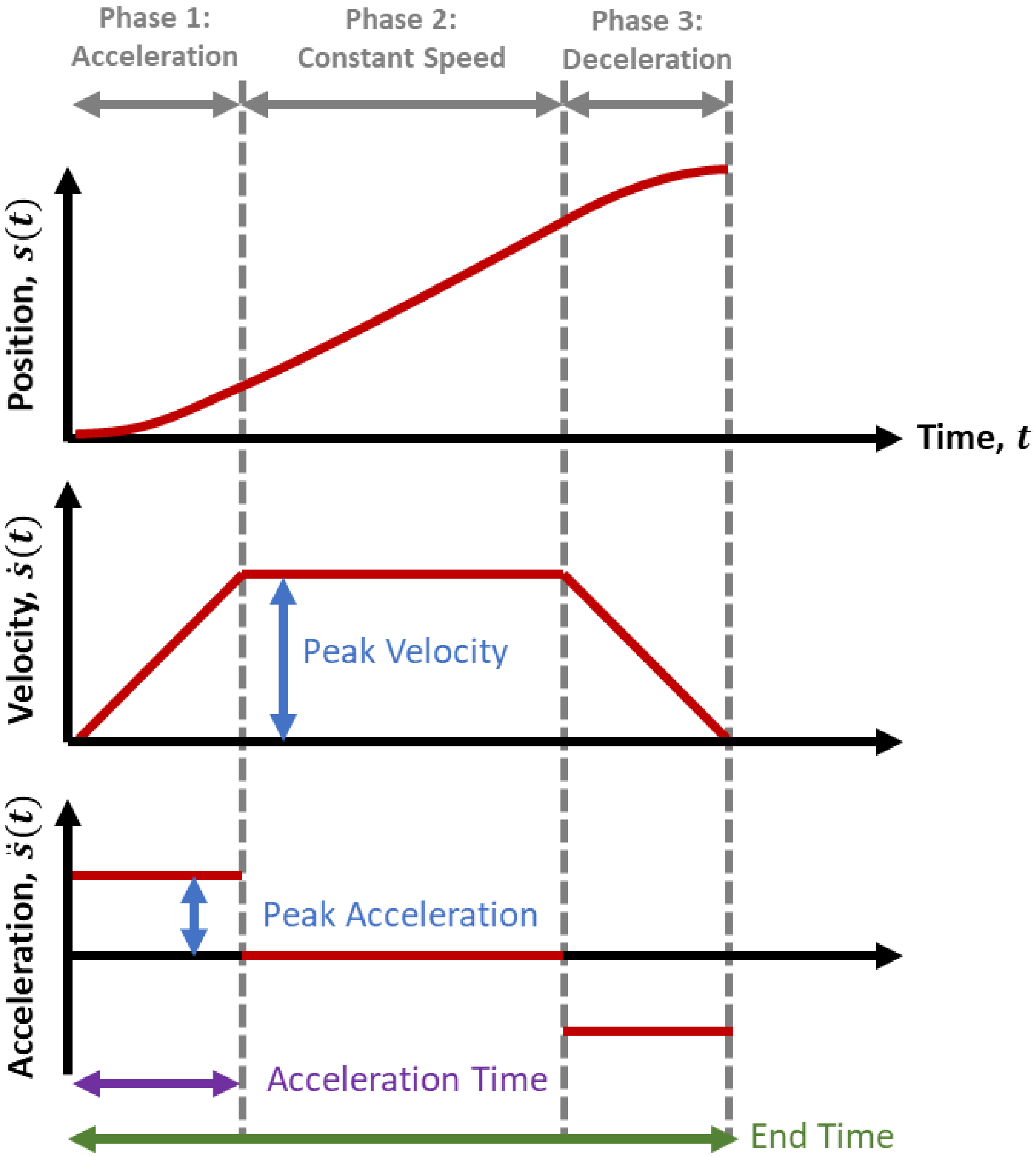
Visualize the results. The generated trajectory is acceptable if the AI must only connect the first and last waypoints in time, but the different segment lengths along dimensions, can make it insufficient for additional tasks, such as collision avoidance, or if using the trajectory to meet points in 3-D space. If the trajectory must meet every waypoint, you can improve it by specifying the end time. This ensures that each segment has the same length.

```
[q,qd,qdd,t] = trapveltraj(wpts,100,PeakVelocity=0.5);
helperPlotTaskSpaceTraj("PeakVelocity = 0.5 rad/s",t,q,qd,wpts);
```



Trapezoidal Velocity Profile Overview

The trapezoidal velocity profile trajectory connects waypoints using a motion profile that stops at each waypoint, and where the waypoint-to-waypoint motion is governed by the following motion profile:



As shown in the image above, the velocity profile has four parameters:

- **End Time** — Duration of each segment between two waypoints
- **Peak Velocity** — Peak velocity for a each segment
- **Acceleration Time** — Time spent in the acceleration and deceleration phases of each segment

- **Peak Acceleration** — Magnitude of the acceleration applied during the acceleration and deceleration phases of each segment

You can fully define a profile by specifying two of these. When you provide only one parameter, `trapveltraj` automatically assigns a second parameter so that the average velocity of each segment is halfway between the two allowable bounds detailed in Constraints for Combining Parameter Specifications on page 1-0 .

Mathematically, this profile defines segments on the interval `[0, endTime]`, where each segment has an acceleration phase, constant speed phase, and a deceleration phase. The lengths of the acceleration and deceleration phases, `accelTime`, are equal.

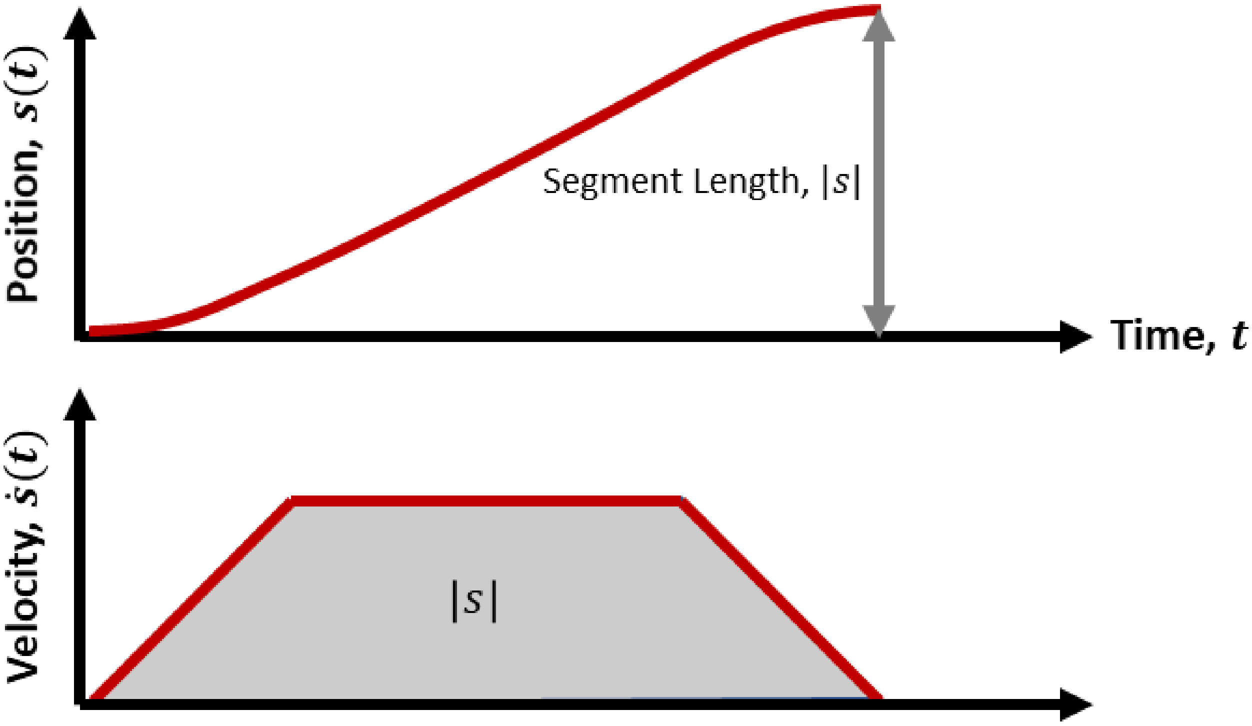
1. **Acceleration Phase** — For t = [0, `accelTime`], $\frac{\mathrm{d}^2}{\mathrm{d}t^2}s(t) = a,\ \frac{d}{\mathrm{d}t}s(t) = at,\ s(t) = a\frac{t^2}{2}$
2. **Constant Speed Phase** — For t = [`accelTime`, `endTime` – `accelTime`], $\frac{\mathrm{d}^2}{\mathrm{d}t^2}s(t) = 0,\ \frac{d}{\mathrm{d}t}s(t) = v,\ s(t) = vt - \frac{v^2}{2a}$
3. **Deceleration Phase** — For t = [`endTime` – `accelTime`, `endTime`], $\frac{\mathrm{d}^2}{\mathrm{d}t^2}s(t) = -a,\ \frac{d}{\mathrm{d}t}s(t) = a(\mathrm{endTime} - t),\ s(t) = \frac{\left(2av * \mathrm{endTime} - 2v^2\right)}{(2a)} - a^2(t - \mathrm{endTime})^2$

Constraints for Combining Parameter Specifications

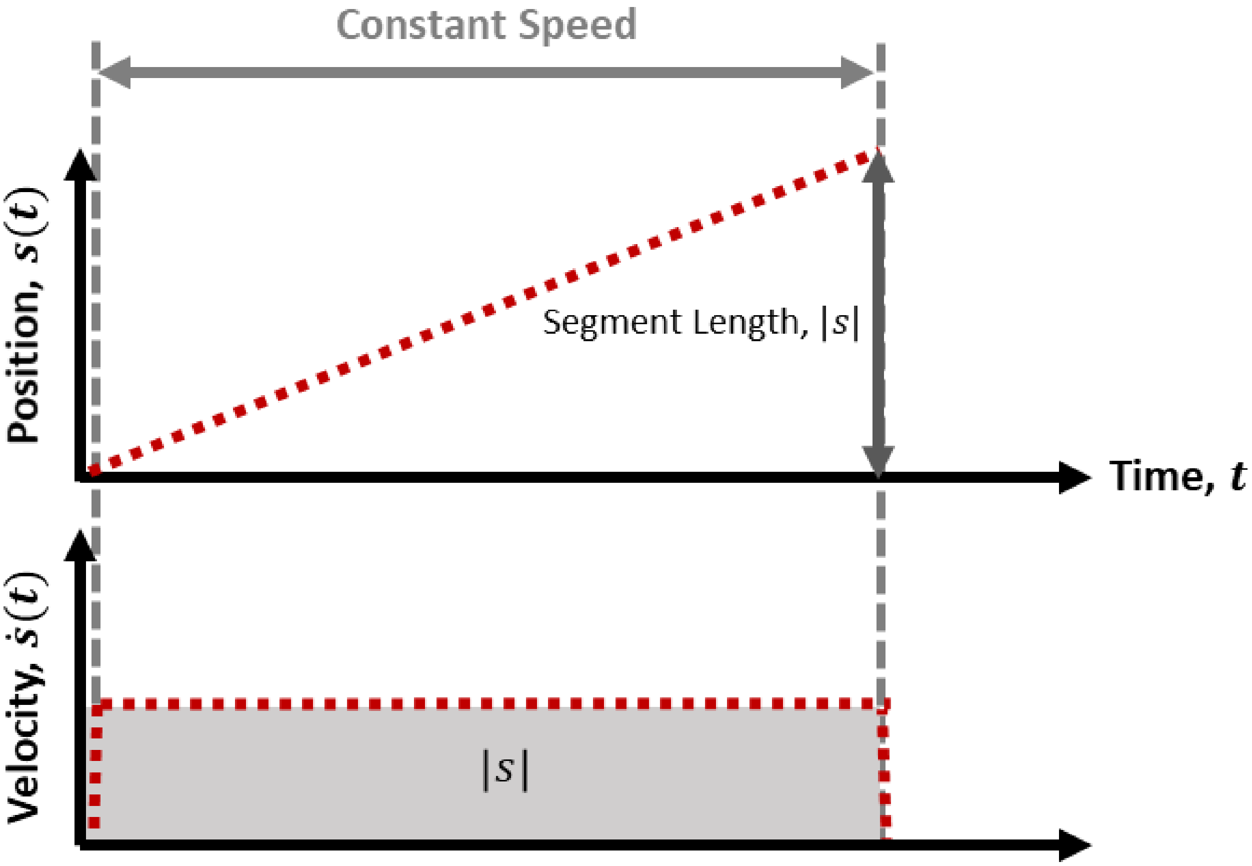
The profile design results in two limiting constraints. The magnitude of the peak velocity must be:

- Peak velocity must be greater than the average speed: $peakVelocity > \frac{s(\mathrm{endTime}) - s(0)}{\mathrm{endTime}}$
- Peak velocity must be less than or equal to twice the average speed: $peakVelocity \leq 2\left(\frac{s(\mathrm{endTime}) - s(0)}{\mathrm{endTime}}\right)$

These constraints are derived from the total segment length $|s| = |s(\mathrm{End\ Time}) - s(0)|$, which is fixed with any two waypoints and results from the integral of the velocity, i.e. the area of the trapezoid:

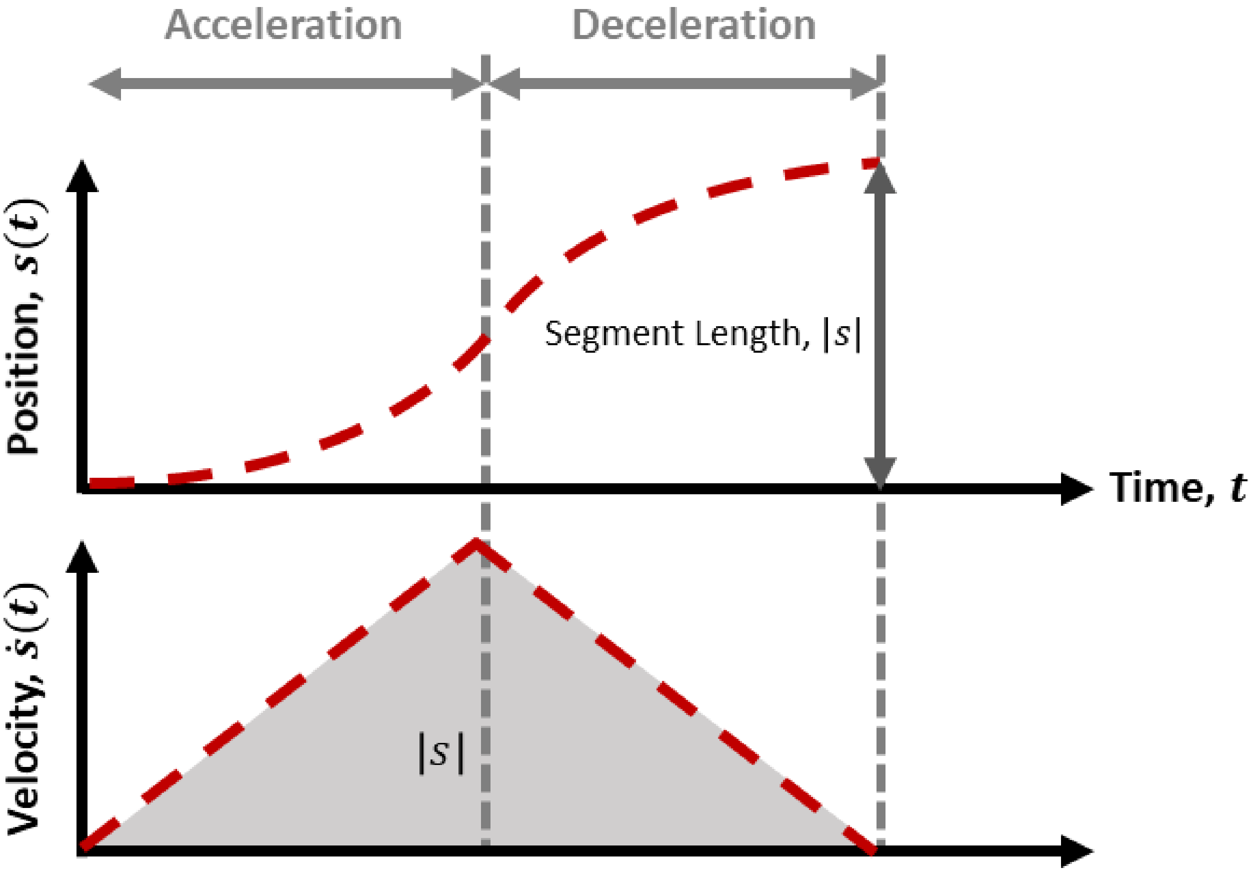


The first constraint sets the lowest allowable peak velocity magnitude. This occurs when the motion spends the most amount of time at constant speed:



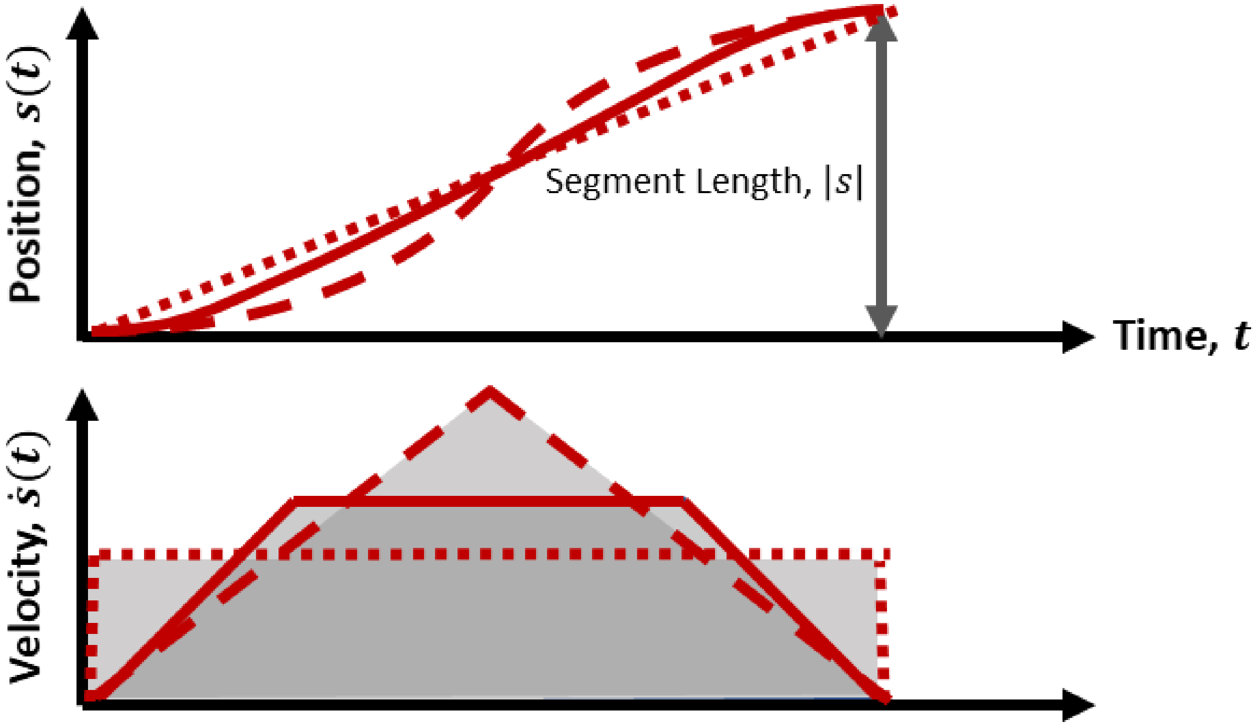
In the second case, the acceleration and deceleration phases are infinitesimal, resulting in a nearly rectangular velocity profile that has a total area A = (Width * Length), or $|s|$ = |peakVelocity| * (endTime). Hence for this case to succeed, the the magnitude of the peak velocity must equal the average speed $\frac{|s|}{\text{endTime}}$. In practice, the acceleration / deceleration phases can never be truly infinitesimal (Acceleration Time > 0), which instead makes this an inequality constraint: peakVelocity > $\frac{|s|}{\text{endTime}}$.

The second constraint defines the highest allowable peak velocity. This occurs when the motion spends as much time as possible accelerating to a peak velocity:



In this situation, there is no constant speed phase, resulting in a triangular velocity profile, which has a total area $A = \frac{\text{Height} * \text{Length}}{2}$ or $|s| = \frac{|\text{Peak Velocity}| * (\text{End Time})}{2}$. Therefore in this extreme, the magnitude of the peak velocity must equal twice the peak speed, resulting in the second constraint, $|\text{Peak Velocity}| > \frac{2|s|}{\text{endTime}}$.

These two constraints define the viable range of trapezoidal velocity profiles for a given segment. The following graphic compares three such trajectories, with the first constraint shown using dotted lines, the second shown with dashed lines, and an example of a trapezoidal profile within these constraints shown as a solid line. All plausible trapezoidal profile trajectories for this segment must fall beteen the dotted and dashed lines, as some variation of the solid line example.



Limitations on Parameter Combinations

In general, any two of the four defining parameters (End Time, Peak Velocity, Peak Acceleration, Acceleration Time) can be used together, but you must ensure that it is possible to generate feasible trajectories within the limits of the profile. For example, suppose there is a segment between point 0 and 1 rad, and the desired segment duration is 1 second.

```
[q,qd]=trapveltraj([0 1],100,'EndTime',1,'PeakVelocity',1);
```

Given the constraints derived above, for the segment from s = 0 to s = 1, the constraints imply that: |Peak Velocity|(End Time) > 1 and |Peak Velocity|(End Time) < 2, whereas with the values used above, the first condition is not met. This could be resolved e.g. by raising the peak velocity to 1.5.

Limitations can also arise because the constraints work for one dimension of a segment, but not for all of them. For example, suppose there is a segment between [0 0] and [1 2], with assigned values for the segment duration and peak velocity:

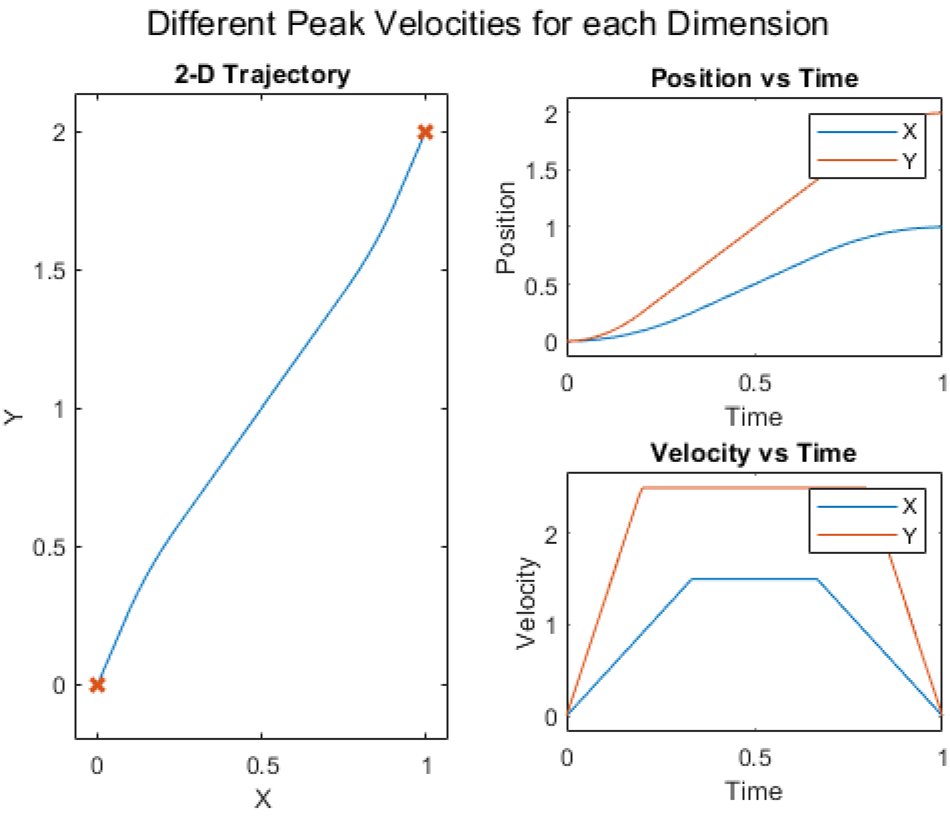
```
[q,qd]=trapveltraj([0 1; 0 2],100,'EndTime',1,'PeakVelocity',1.5);
```

This will fail because the constraints only work for the first dimension:

- For the first dimension, from s = 0 to s = 1, the constraints imply that:
  |Peak Velocity|(End Time) > 1 and |Peak Velocity|(End Time) < 2
- For the second dimension, from s= 0 to s = 2, the constraints imply that:
  |Peak Velocity|(End Time) > 2 and |Peak Velocity|(End Time) < 4

It is evident that these constraints cannot be met if the same values are used for both dimensions. Instead, it is necessary to use different peak velocities for each dimension to ensure that the waypoints can be hit with the same end times. For example, the following solution works:

```
wpts2 = [0 1; 0 2];
[q,qd,qdd,t]=trapveltraj(wpts2,100,'EndTime',1,'PeakVelocity',[1.5; 2.5]);
helperPlotTaskSpaceTraj('Different Peak Velocities for each Dimension',t,q,qd,wpts2);
```



Choosing parameters that satisfy all constraints along each dimension can be challenging. In the next section, a helper function is instead created to help simplify this process.

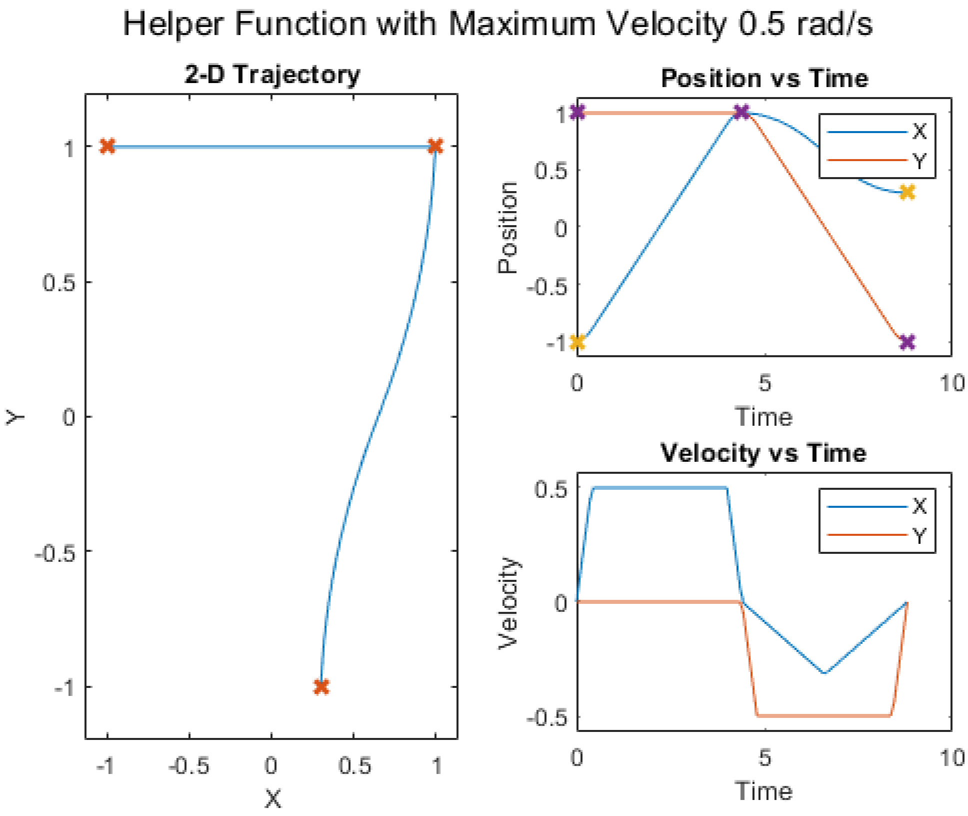
Create a Helper Function to Translate Velocity Bounds to Profile Parameters

The `helperProfileForMaxVel` helper function accepts a limiting segment velocity and outputs a set of end times and velocities that ensure all dimensions have the same segment length, and that the maximum velocity of any segment is less than or equal to the specified velocity maximum.

```
function [endTimes,peakVels] = helperProfileForMaxVel(wpts,maxVelocity)
    %   helperProfileForMaxVel Generate parameters for a trapezoidal velocity profile so it will
    %
    %   This function creates a fast trapezoidal profile with a specified velocity limit as an u
    %   Copyright 2021 The MathWorks, Inc.

    % Check that maxVelocity is a scalar
    validateattributes(maxVelocity,{'numeric'},{'scalar'});

    % Find segment lengths along each dimension of waypoints.
    segLengths = abs(diff(wpts,1,2));

    % Find minimum endTime by assigning maximum velocity to longest segment
    maxSegLengths = max(segLengths,[],1);
    endTimeLowerBound = (maxSegLengths/maxVelocity);

    % Choose acceleration by multiplying endTime to be greater than lower bound
    greaterThanFactor = 1.1;
    endTimes = repmat(greaterThanFactor*endTimeLowerBound,size(wpts,1),1);

    % Determine min and max peak velocity for each segment and dimension
    % Choose largest velocity that doesn't exceed maximum velocity
    minPeakVels = segLengths./endTimes;
    maxPeakVels = 2*segLengths./endTimes;

    peakVels = min(maxPeakVels,maxVelocity);
    peakVels = max(minPeakVels,peakVels);

    % Replace any zero-values with peak velocity
    peakVels(peakVels==0) = maxVelocity;
end
```

Use the helper function to create a trajectory for the waypoints that has a maximum velocity bound of 0.5 rad/s.

```
[endTimes,peakVels] = helperProfileForMaxVel(wpts,0.5);
[q,qd,qdd,t,pp] = trapveltraj(wpts,100,EndTime=endTimes,PeakVelocity=peakVels);
tpts = [0 cumsum(endTimes(1,:))];
helperPlotTaskSpaceTraj("Helper Function with Maximum Velocity 0.5 rad/s",t,q,qd,wpts,tpts);
```



Generate Code for Motion Planning Using Robot Model Imported from URDF

This example shows how to perform code generation to plan motion using robot model imported from URDF file. For this example, you use a `manipulatorRRT` object with a imported `rigidBodyTree` robot model to find a obstacle-free path between two configurations of the robot. After you verify the algorithm in MATLAB®, use the generated MEX file in the algorithm to visualize the robot movement.

Write Algorithm to Plan Path

Create a function, `iiwaPathPlanner`, that uses a `manipulatorRRT` object to plan a path between two configurations for the KUKA LBR iiwa 14 robot model in an obstacle filled environment.

```
function path = iiwaPathPlanner(startConfig, goalConfig, collisionDims, collisionPoses)
    %#codegen
    robot = iiwaForCodegen('row');
    collisionObjects = cell(1,length(collisionDims));
    for i=1:length(collisionDims)
        switch length(collisionDims{i})
            case 1
                sphereRadius = collisionDims{i};
                collisionObjects{i} = collisionSphere(sphereRadius{1});
                collisionObjects{i}.Pose = collisionPoses{i};
            case 2
                cylinderDims = collisionDims{i};
                collisionObjects{i} = collisionCylinder(cylinderDims{1}, cylinderDims{2});
                collisionObjects{i}.Pose = collisionPoses{i};
            case 3
                boxDims = collisionDims{i};
                collisionObjects{i} = collisionBox(boxDims{1}, boxDims{2}, boxDims{3});
                collisionObjects{i}.Pose = collisionPoses{i};
        end
    end

    planner = manipulatorRRT(robot, collisionObjects);
    path = plan(planner, startConfig, goalConfig);
end
```

The algorithm acts as a wrapper for a standard RRT motion planning call. It accepts standard inputs, and returns a set of robot configuration vectors as the path. Since you cannot use handle objects as the input or output for functions that are supported for code generation. Load the robot inside the function. Save the `iiwaPathPlanner` function in your current folder.

Verify Motion Planning Algorithm in MATLAB

Verify the motion planning algorithm in MATLAB before generating code.

Import a KUKA LBR iiwa 14 robot model as a `rigidBodyTree` object. Set the data format to `"row"`.

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

Create `rigidBodyTree` code generating function for the robot model.

```
writeAsFunction(robot,'iiwaForCodegen');
```

Create a simple environment with obstacles using collision primitives.

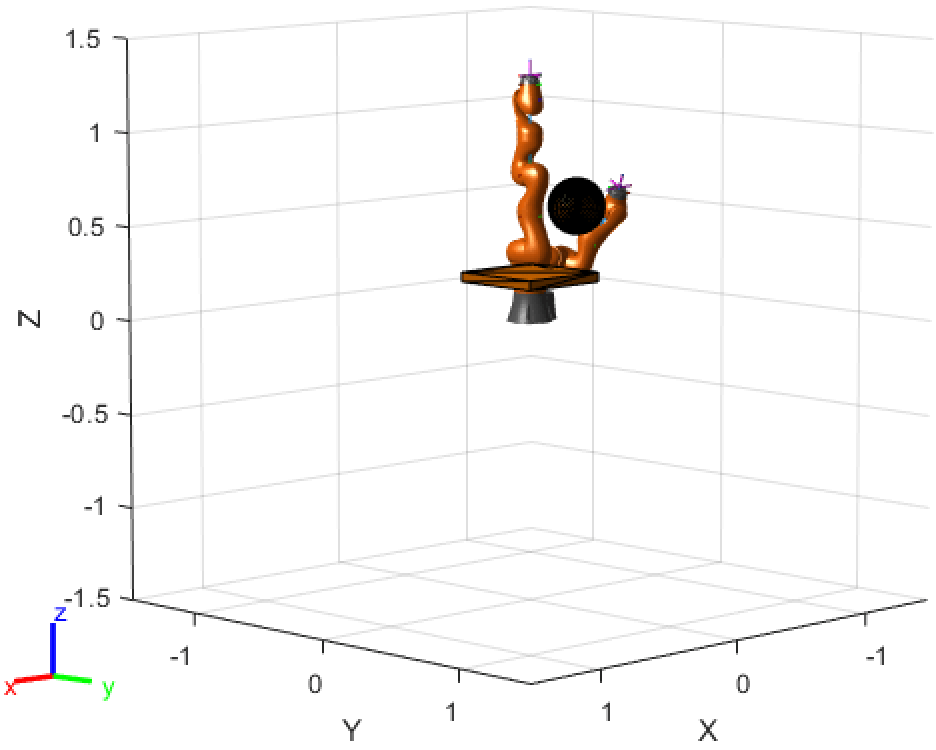
```
env = {collisionBox(0.5, 0.5, 0.05),collisionSphere(0.15)};
env{1}.Pose = trvec2tform([0.35 0.35 0.3]);
env{2}.Pose = trvec2tform([-0.25 0.1 0.6]);
```

Define a start and goal configuration. You must specify a start and goal that do not overlap with the obstacles. Otherwise, the planner throws an error.

```
startConfig = robot.homeConfiguration;
goalConfig = [-2.9236 1.8125 0.6484 1.6414 -0.4698 -0.4181 0.3295];
```

Visualize initial and final positions of the robot.

```
figure;
show(robot,startConfig);
hold all;
show(robot,goalConfig);
show(env{1});
show(env{2});
```



Extract the collision data from the environment.

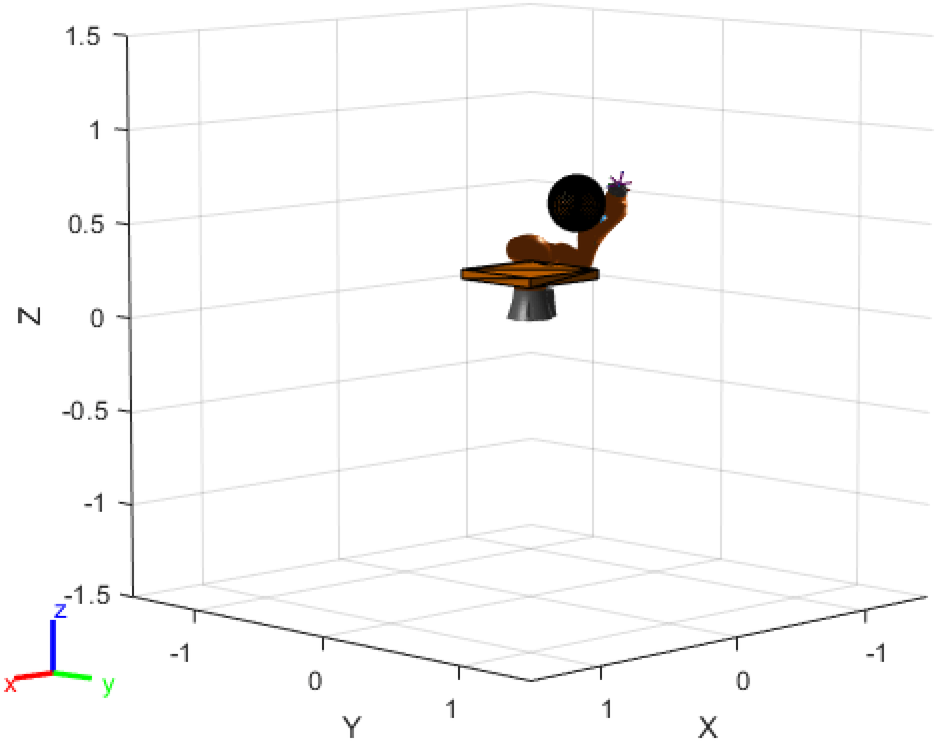
```
collisionDims = {{env{1}.X env{1}.Y env{1}.Z},{env{2}.Radius}};
collisionPoses = cellfun(@(x)(x.Pose),env,'UniformOutput',false);
```

Plan the path.

```
path = iiwaPathPlanner(startConfig,goalConfig,collisionDims,collisionPoses);
```

Visualize the robot. Set the 'FastUpdate' option of the show method to true to get a smooth animation.

```
figure;
ax = show(robot,startConfig);
hold all
show(env{1},'Parent',ax);
show(env{2},'Parent',ax);
rrt = manipulatorRRT(robot,env);
interpPath = interpolate(rrt,path);
for i = 1:size(interpPath,1)
    show(robot,interpPath(i,:),'PreservePlot',false,'FastUpdate',true);
    drawnow;
end
```



Generate Code for Motion Planning Algorithm

You can use either the codegen (MATLAB Coder) function or the MATLAB Coder (MATLAB Coder) app to generate code. For this example, generate a MEX file by calling codegen at the MATLAB command line. Specify sample input arguments for each input to the function using the -args input argument.

Call the codegen function and specify the input arguments in a cell array. This function creates a separate iiwaPathPlanner_mex function to use. You can also produce C code by using the options input argument. This step can take some time.

```
codegen iiwaPathPlanner -args {startConfig,goalConfig,collisionDims,collisionPoses}
```

```
Code generation successful.
```

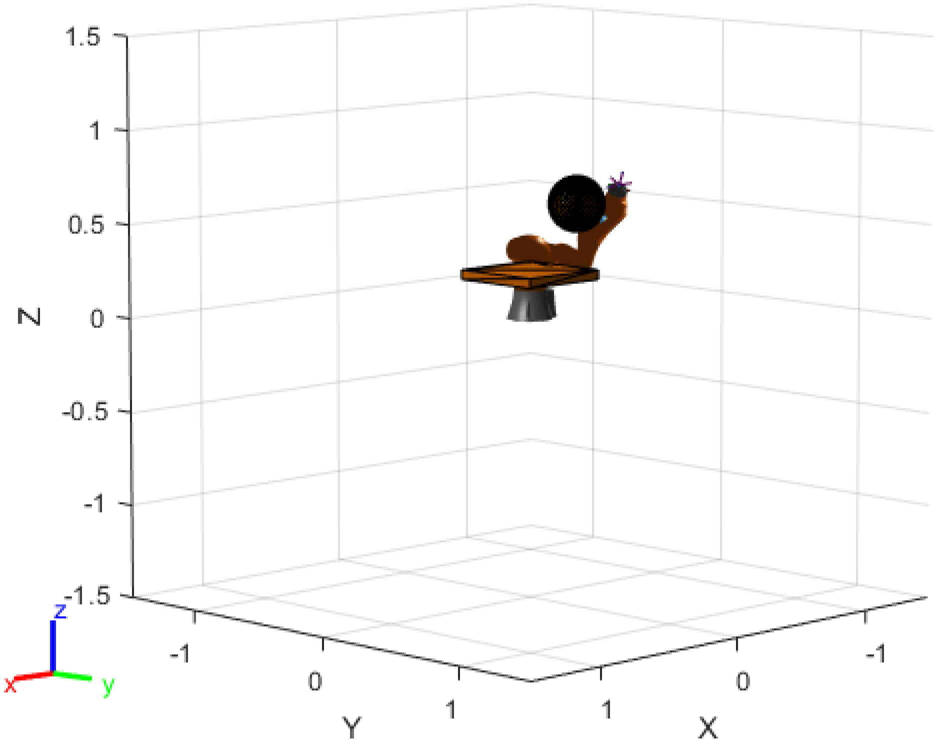
Verify Results Using Generated MEX Function

Plan the path by calling the MEX version of the motion planning algorithm for the specified start and goal configurations, and collision data from the environment.

```
path = iiwaPathPlanner_mex(startConfig,goalConfig,collisionDims,collisionPoses);
```

Visualize the robot with the robot configurations computed using the MEX version of the motion planning algorithm. Set the `'FastUpdate'` option of the `show` method to `true` to get a smooth animation.

```
figure;
ax = show(robot, startConfig);
hold all
show(env{1},'Parent',ax);
show(env{2},'Parent',ax);

interpPath = interpolate(rrt,path);
for i = 1:size(interpPath,1)
    show(robot,interpPath(i,:),'PreservePlot',false,'FastUpdate',true);
    drawnow;
end
```



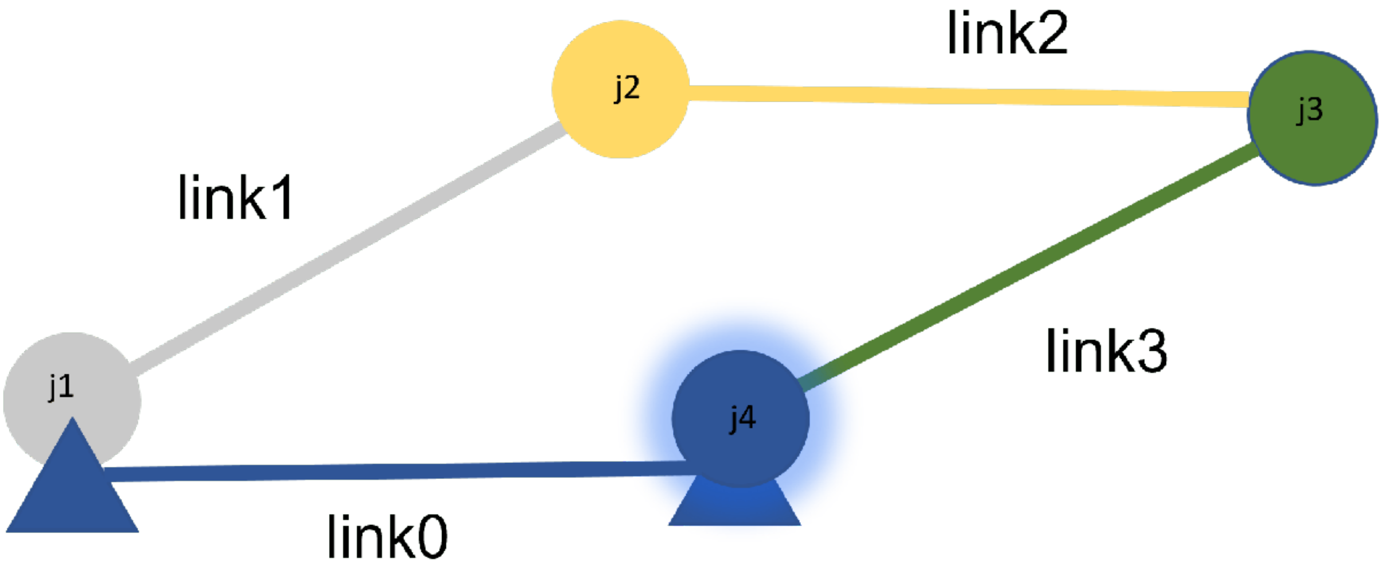
Solve Inverse Kinematics for Closed Loop Linkages

Closed loop linkages are widely used in automobiles, construction and manufacturing machines, and in robot manipulation. Although you cannot directly model closed-loop linkages with the `rigidBodyTreeImportInfo` object in Robotics System Toolbox™, you can still study the kinematics of closed-loop systems by combining a rigid body tree with constraints that mimic loop-closing joints. The `constraintRevoluteJoint`, `constraintPrismaticJoint`, and `constraintFixedJoint` constraint objects enable you to model loop-closing revolute, prismatic, and fixed joints, respectively. To kinematically model closed-loop linkages, use the constraints with a `generalizedInverseKinematics` solver to constrain the solutions to act as desired.

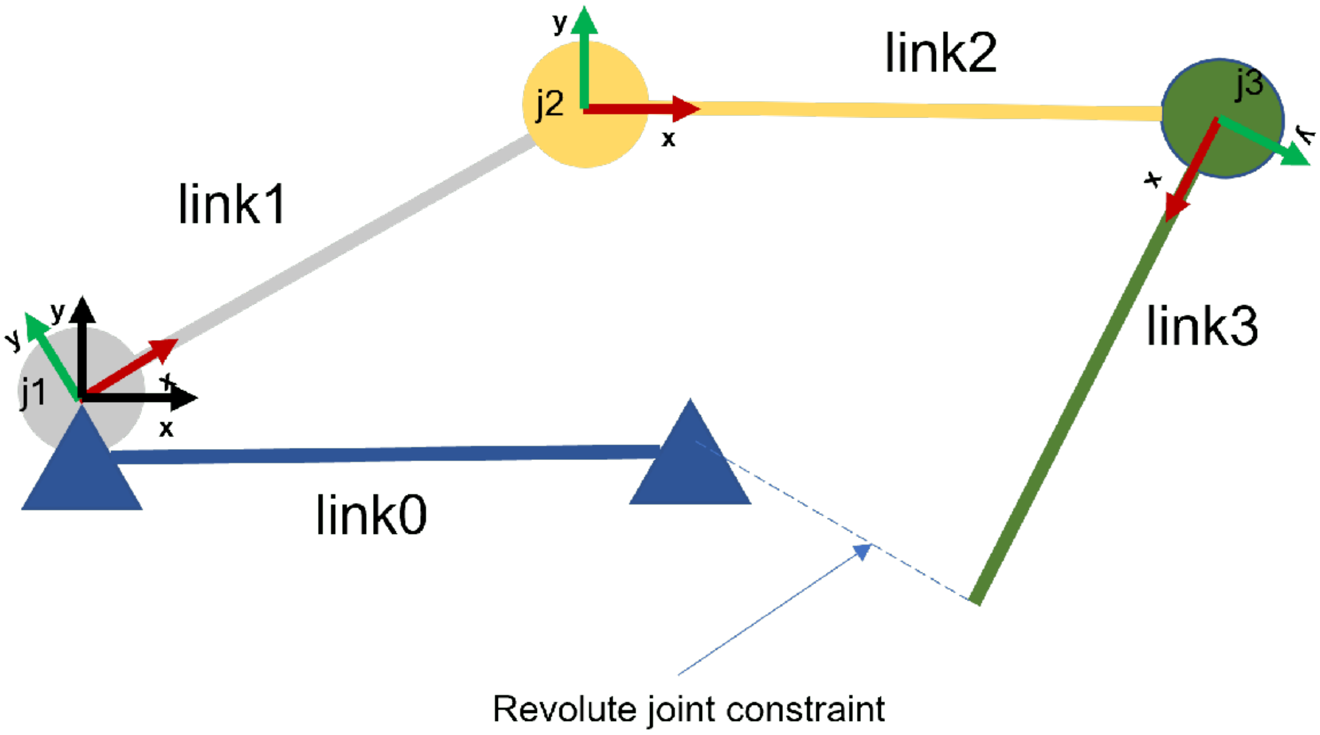
This example shows how to model a four-bar linkage, a widely used closed-loop linkage, using the `rigidBodyTree` and `constraintRevoluteJoint` objects, and the `generalizedInverseKinematics` System object™.

Four-Bar Linkage as Rigid Body Tree with Revolute Joint Constraint

This figure shows a four-bar linkage. The highlighted joint j4 is a loop-closing revolute joint.

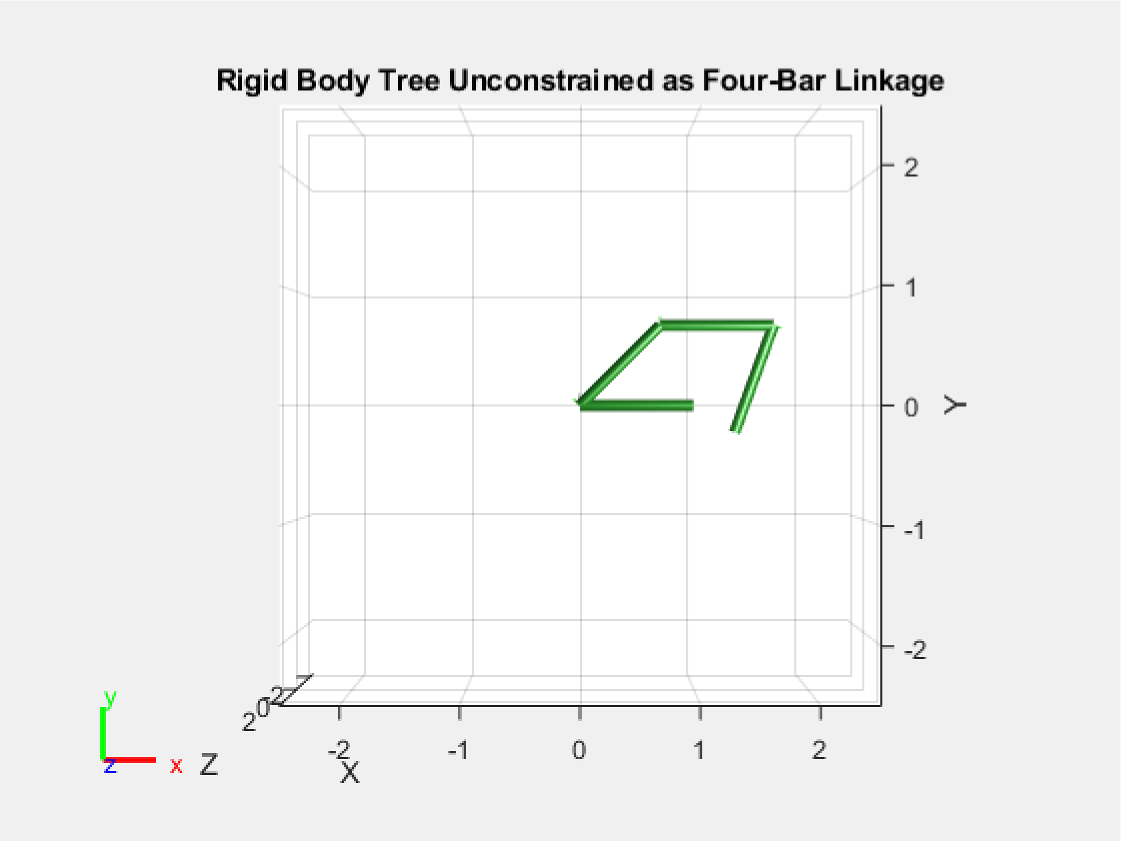


You can create a rigid body tree corresponding to the four-bar linkage, which consists of joints j1, j2, and j3, and is constrained with a revolute joint constraint that acts as a fourth joint, j4. This figure shows the reference frames of the links of the rigid body tree in an unconstrained joint configuration. The frames of the links are located at their respective joints. The frame corresponding to the base body of the rigid body tree, link0, is shown is represented by the black arrows.



Visualize the rigid body tree in an unconstrained configuration.

```
fourBarLinkageTree =  exampleHelperFourBarLinkageTree();
figure(Name="Rigid Body Tree Unconstrained As Four-Bar Linkage",Visible="on")
unconstrainedConfig = [pi/4 -pi/4 -110*pi/180];
ax = show(fourBarLinkageTree, ...
    unconstrainedConfig, ...
    Collisions="on", ...
    Frames="on", ...
    PreservePlot=true, ...
    FastUpdate=false);
title("Rigid Body Tree Unconstrained as Four-Bar Linkage")
view([0 0 pi/2])
```

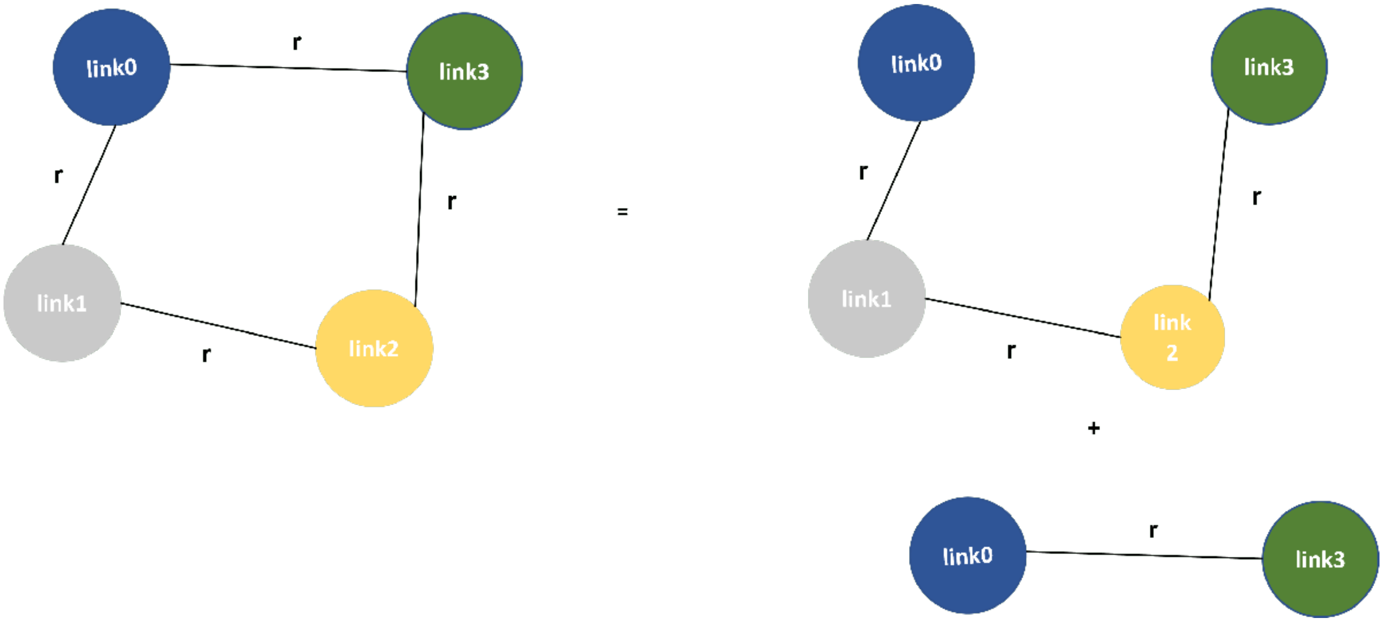


Connectivity Graph of Four-Bar Linkage

One way to approach the modeling of closed-loop linkages is by using connectivity graphs. This approach models the kinematic linkage using a graph whose vertices are the links of the linkage, and an edge between links represents the joint between them.

Using this representation, the goal of modeling the closed-loop linkage as a rigid body tree with joint constraints as removing edges from the graph so that the graph becomes a tree. A tree is a graph with no loops, which instead uses constraints to enforce the removed edges.

In the connectivity graph, the difference between the four-bar linkage and its equivalent rigid body tree representation is that the rigid body tree needs a loop-closing joint constraint from link0 to link3.



Set Up Inverse Kinematics Solver

The four-bar linkage is a one degree-of-freedom linkage, and is driven by the crank link, link1. Given a crank position specified by the value of joint j1, the generalized inverse kinematics solver outputs the joint positions corresponding to joints j2 and j3.

Create an instance of the generalized inverse kinematics solver. For this task, the solver requires two constraint objects, `constraintDistanceBounds` and `constraintRevoluteJoint`.

```
solverFourBarLinkage = generalizedInverseKinematics(...
    RigidBodyTree=fourBarLinkageTree, ...
    ConstraintInputs={'jointbounds','revolutejoint'});

% Disable random restarts for repeatable IK solutions
solverFourBarLinkage.SolverParameters.AllowRandomRestart = false;
solverFourBarLinkage.SolverParameters.StepTolerance = 1e-14;
```

Define the revolute joint constraint between link3 and link0.

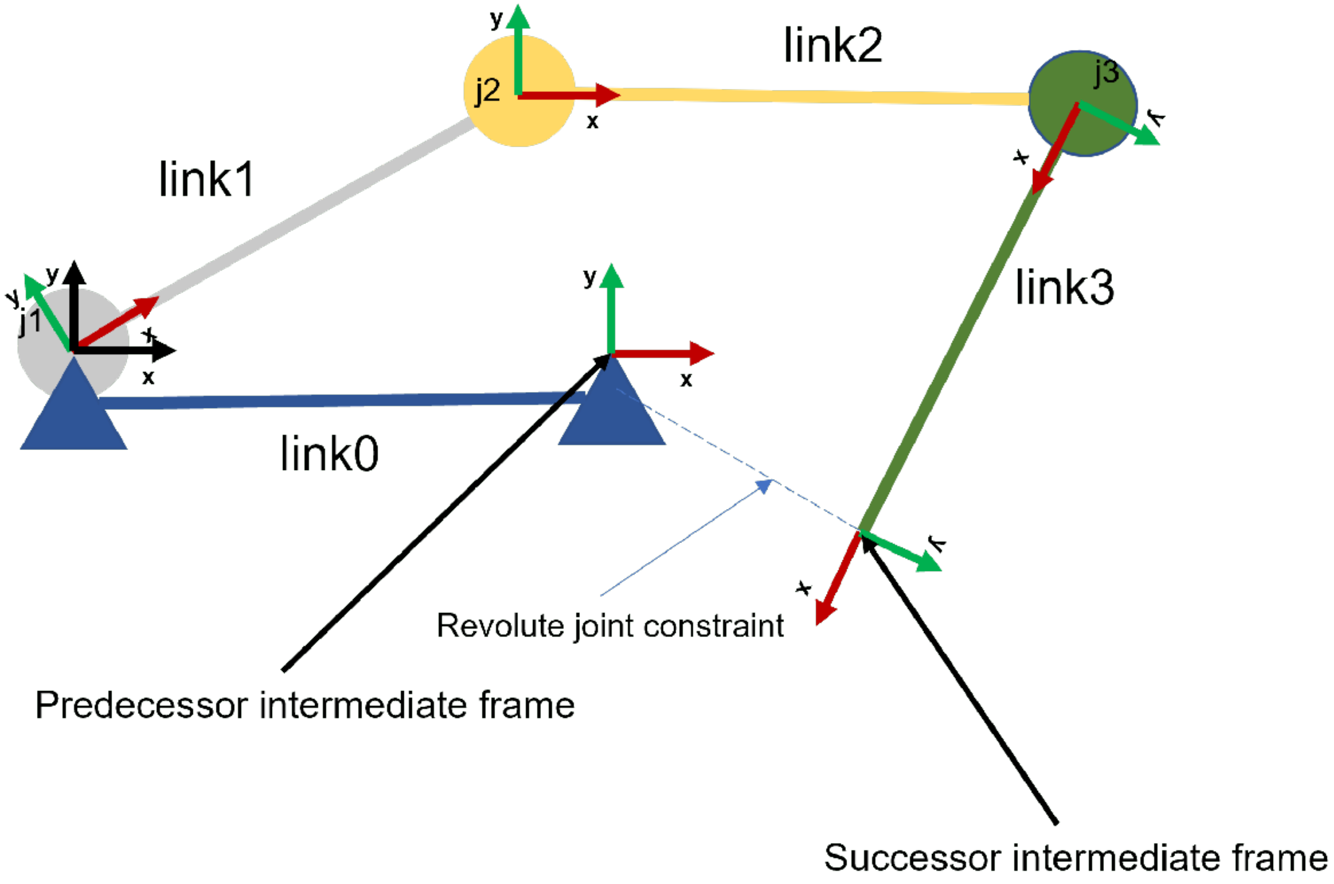
```
% Revolute joint constraint between "link3" and "link0"
cRevolute = constraintRevoluteJoint("link3","link0");
```

Toensure that the crank position is fixed with respect to the solution, you must implement a joint bounds constraint. The elements of the `Weights` property of the joint bounds constraint corresponds to a joint position in the joint configuration of the rigid body tree. In this case, only joint j1 has a non-zero weight, as it is the joint that we want to constrain.

```
% Joint bounds constraint to fix the crank position.
activeJointConstraint = constraintJointBounds(fourBarLinkageTree);
activeJointConstraint.Weights = [1 0 0];
```

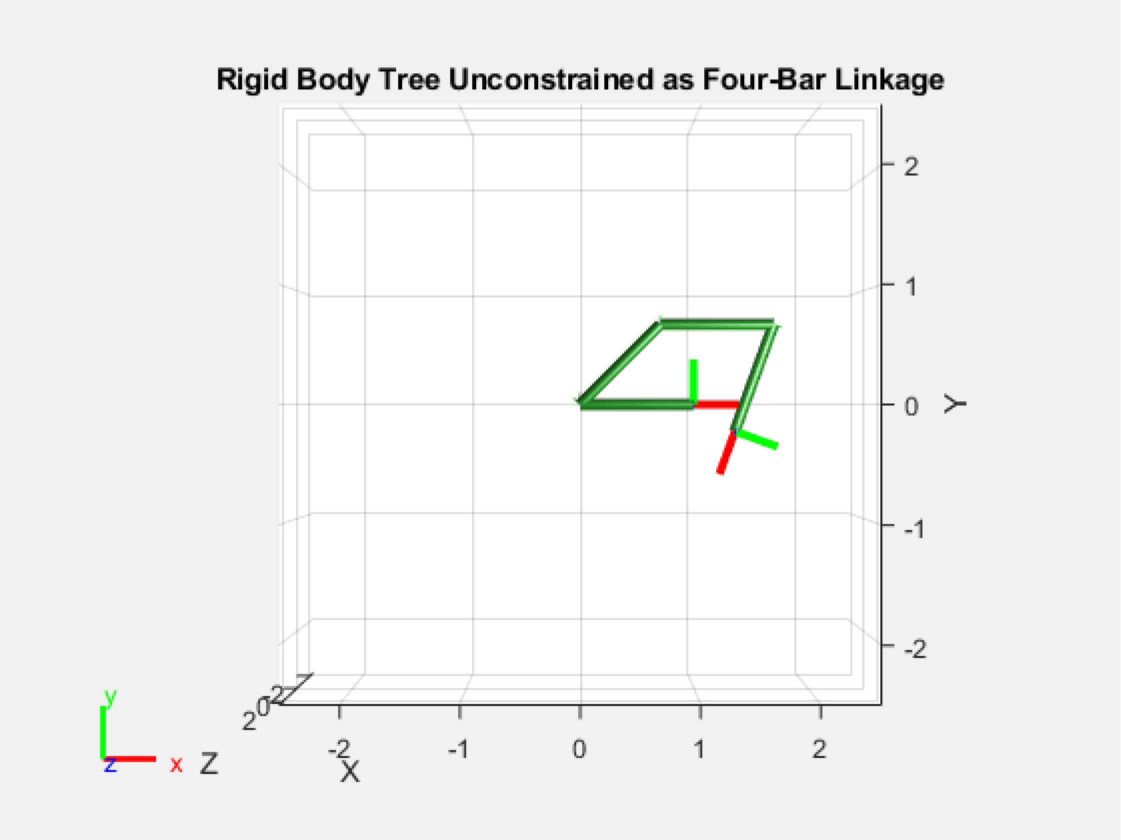
Intermediate Frames of Constraints

Revolute joint constraints constrain two intermediate frames. Each frame corresponds to the predecessor and the successor body. This figure below shows which intermediate frames to constrain to mimic the loop-closing joint.

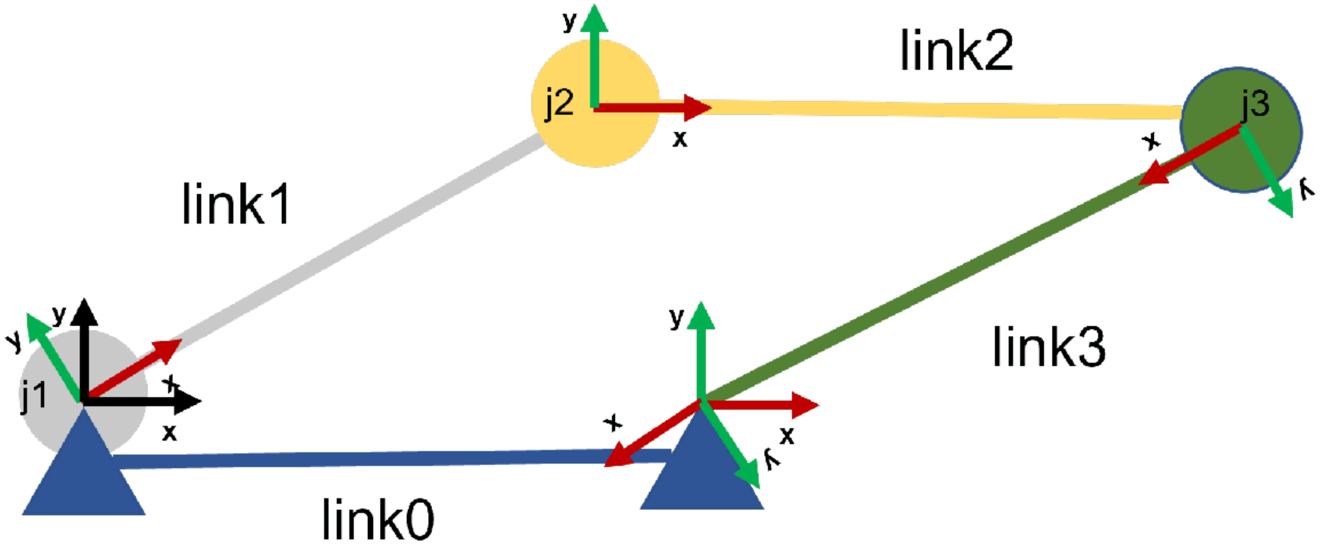

link2
j2
y
x
j3
y
x
link1
link3
j1
y
y
x
x
y
x
link0
y
x
Revolute joint constraint
Predecessor intermediate frame
Successor intermediate frame

In the unconstrained configuration, the predecessor intermediate frame is located at along the *x*-axis of the base (link0) frame at `[1 0 0]`, shown as the black arrows. Similarly, the successor intermediate frame is located along the *x*-axis of the link3 frame at `[1 0 0]`.

```
cRevolute.PredecessorTransform = trvec2tform([1 0 0]);
cRevolute.SuccessorTransform = trvec2tform([1 0 0]);
vis = exampleHelperFrameVisual(fourBarLinkageTree,cRevolute,ax);
vis.update(unconstrainedConfig);
view([0 0 pi/2])
```



If you constrain the intermediate frames, they co-locate. This figure shows a constrained configuration that corresponds to the unconstrained configuration.



Visualize Constrained Motion

The joint position, `theta`, of the crank drives the linkage. For a discrete set of crank positions in the interval $\left[\frac{\pi}{2}, 2\pi + \frac{\pi}{2}\right]$, the solver solves for joint positions in the rigid body tree such that it behaves like a four-bar linkage. This code visualizes the constrained rigid body tree for every crank position, `theta`.

For the initial crank position, $\frac{\pi}{2}$, guess an initial constrained configuration for the solver. The visualization loop overwrites this value.

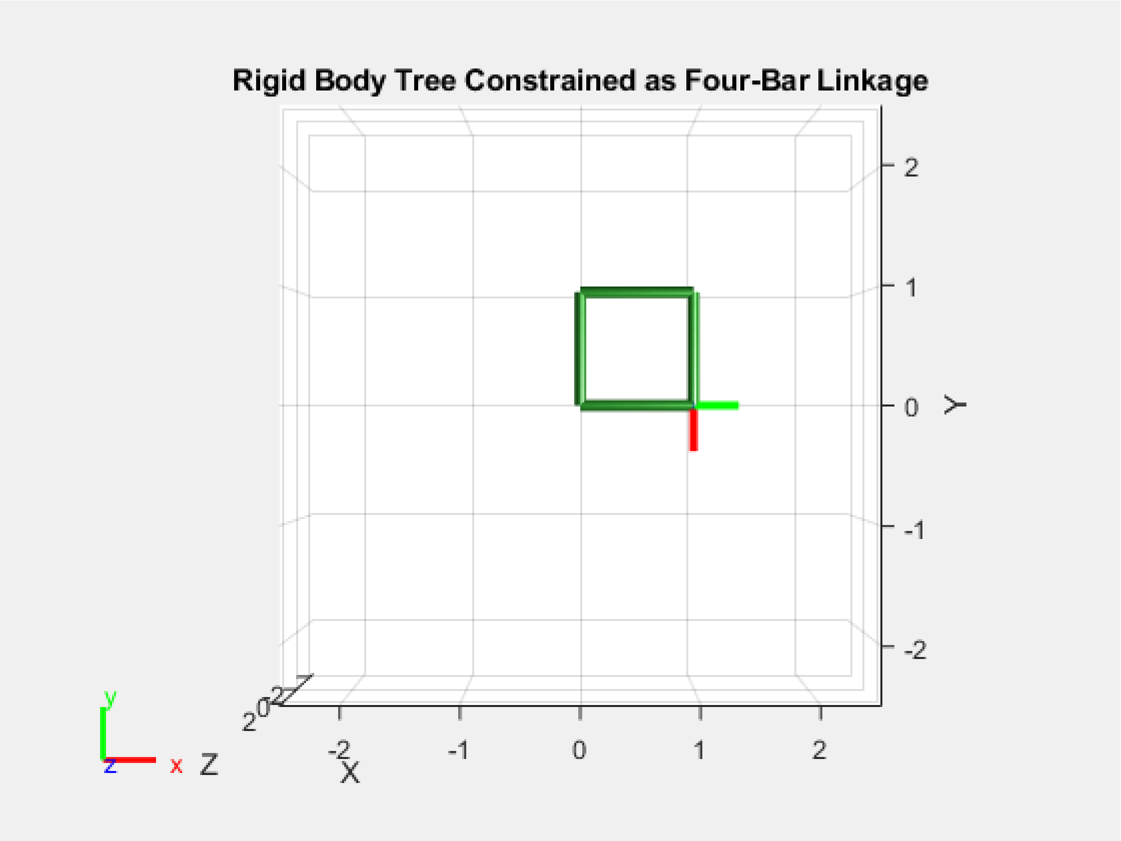
```
constrainedConfig = [pi/2 0 pi/3];
```

Initialize the figure for visualization.

```
figure(Name="Rigid Body Tree Constrained As Four-Bar Linkage",Visible="on")
ax = show(fourBarLinkageTree,constrainedConfig, ...
    PreservePlot=false, ...
    FastUpdate=true, ...
    Collisions="on");
title("Rigid Body Tree Constrained as Four-Bar Linkage")
vis = exampleHelperFrameVisual(fourBarLinkageTree,cRevolute,ax);
```

Visualize the four-bar linkage for a specified crank position.

```
for theta = linspace(pi/2,5*pi/2,101)
    % Fix the crank position to "theta" in the solver-generated constrained
    % config. This is done by setting identical upper and lower values for
    % the joint bounds constraint.
    config = [theta constrainedConfig(2) constrainedConfig(3)];
    activeJointConstraint.Bounds(1,:) = [theta theta];
    constrainedConfig = solverFourBarLinkage(config,activeJointConstraint,cRevolute);

    % Visualize the constrained rigid body tree and the
    % predecessor and successor intermediate frames of the constraint.
    show(fourBarLinkageTree,constrainedConfig, ...
        PreservePlot=false, ...
        FastUpdate=true, ...
        Collisions="on");
    vis.update(constrainedConfig);
    view([0 0 pi/2]);
    drawnow;
end
```


Rigid Body Tree Constrained as Four-Bar Linkage
2
1
0
-1
-2
Y
-2
-1
0
1
2
X
-2
0
2
Z
y
x
z

Plan Manipulator Path for Dispensing Task Using Inverse Kinematics Designer

This example shows how to design a robotic manipulator path for a dispensing task. A successful path consists of a sequence of collision-free waypoints that are designed and verified in the **Inverse Kinematics Designer** app. Create waypoints for an adhesive dispensing task, in which the robot picks up two adhesive strips, applies glue, and then applies the strips to a box.

Create Robot and Environment Elements for the Scene

This example has three main steps:

1. The robot picks up adhesive strips from a loading station using a custom tool.
2. The robot places the tool under a dispensing stand, where glue is applied to both strips.
3. The robot attaches the two strips to the object in the desired location.

The example uses a Universal Robots UR5e manipulator, equipped with a custom end effector. The result of this application is a series of waypoints that can be connected via a planner, trajectory tooling, or both. The path planner takes the waypoints to generate a single collision-free path that reaches all the waypoints.

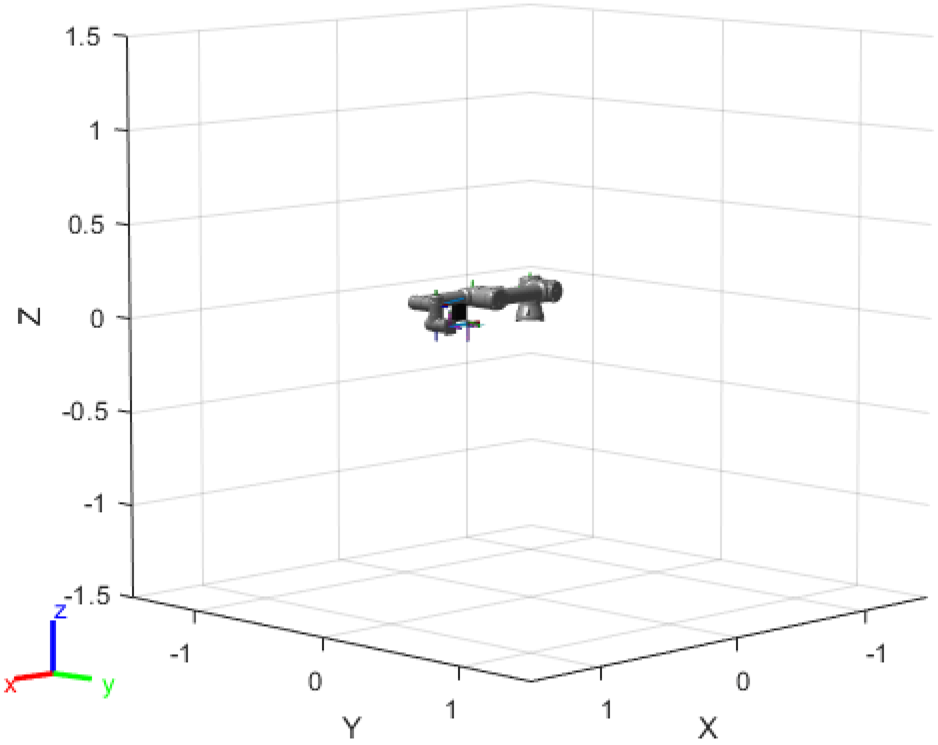
Create UR5e with Custom End Effector

This example will make use of the UR5 e-series, which is available in the robot library.

```
ur5e_robot = loadrobot("universalUR5e");
```

Create the custom gripper with the `customEEBuilder` helper function and then attach the custom gripper to the robot. Show the robot to verify that the gripper is on the robot.

```
customEE = customEEBuilder.build(true);
addSubtree(ur5e_robot,'tool0',customEE);
show(ur5e_robot);
```



Create Environment Representation

The complete scene has several collision objects to represent scene objects. Create the primitive collision objects using the `constructToolStation` helper function.

Add the feeder station at the position `[0 0.4 0]`.

```
stationPose = trvec2tform([0 0.4 0]);
[toolStationBase,partFeeder] = constructToolStation(stationPose);
```

Next add the dispensing station, modeled by a small cylinder next to the feeder at `[.25 .45 .65]`.

```
dispensingStation = collisionCylinder(.01,.1);
dispensingStation.Pose = trvec2tform([.25 .45 .65]);
```

Create a platform underneath the robot located at `[0 0 -.011]`.

```
platform = collisionBox(1,1,0.02);
platform.Pose = trvec2tform([0 0 -.011]);
```

Add the box that the robot places adhesive strips on in front of the robot at `[0.3 -.3 .05]`.

```
box = collisionBox(0.2,0.3,0.1);
box.Pose = trvec2tform([0.3 -.3 .05]);
```

Use Inverse Kinematics Designer to Create Waypoints

Use **Inverse Kinematics Designer** to find the waypoint configurations that satisfy specific goals. Once the waypoints are created, these configurations are exported from the app to the base workspace.

To skip to the final state, load the included save session `dispensingSessionData` and that associated exported configurations by executing:

```
inverseKinematicsDesigner("dispensingSessionData.mat")
load pathWaypointData.mat
```

Otherwise, follow along with the example.

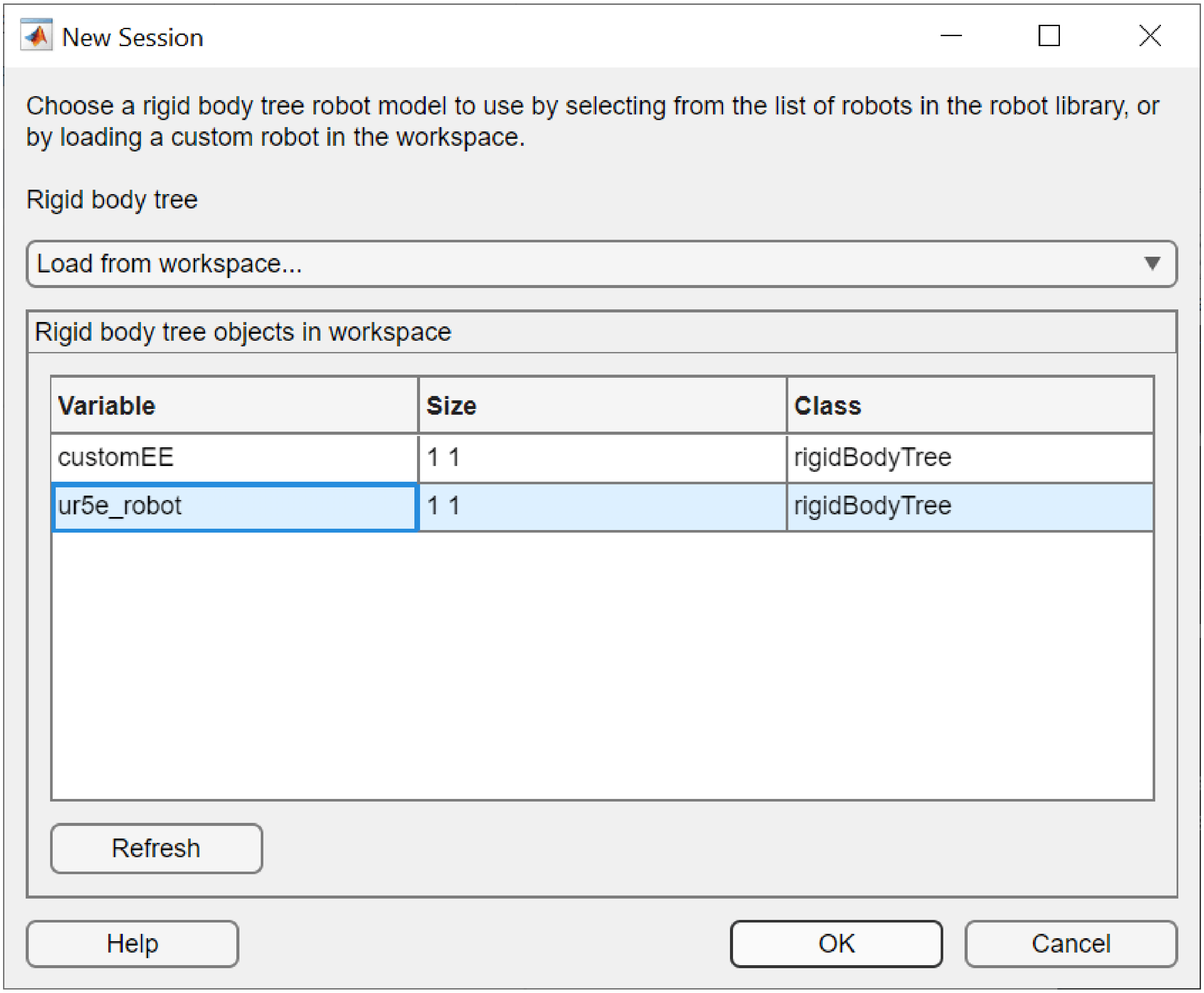
Start New Session

Now that the scene elements have been defined, start a new app session and populate it with the robot and environment pieces.

To start the app, open it from the Apps ribbon, or call it using the following command:

```
inverseKinematicsDesigner
```

Next, start a new session. On the toolstrip, click **New Session** to bring up the **New Session** dialog box. Make sure the dropdown is set to **Load robot from workspace** and select `ur5e_robot`, the robot that was defined in the previous section. Click **OK** to import it and start the session.



New Session

Choose a rigid body tree robot model to use by selecting from the list of robots in the robot library, or by loading a custom robot in the workspace.

Rigid body tree

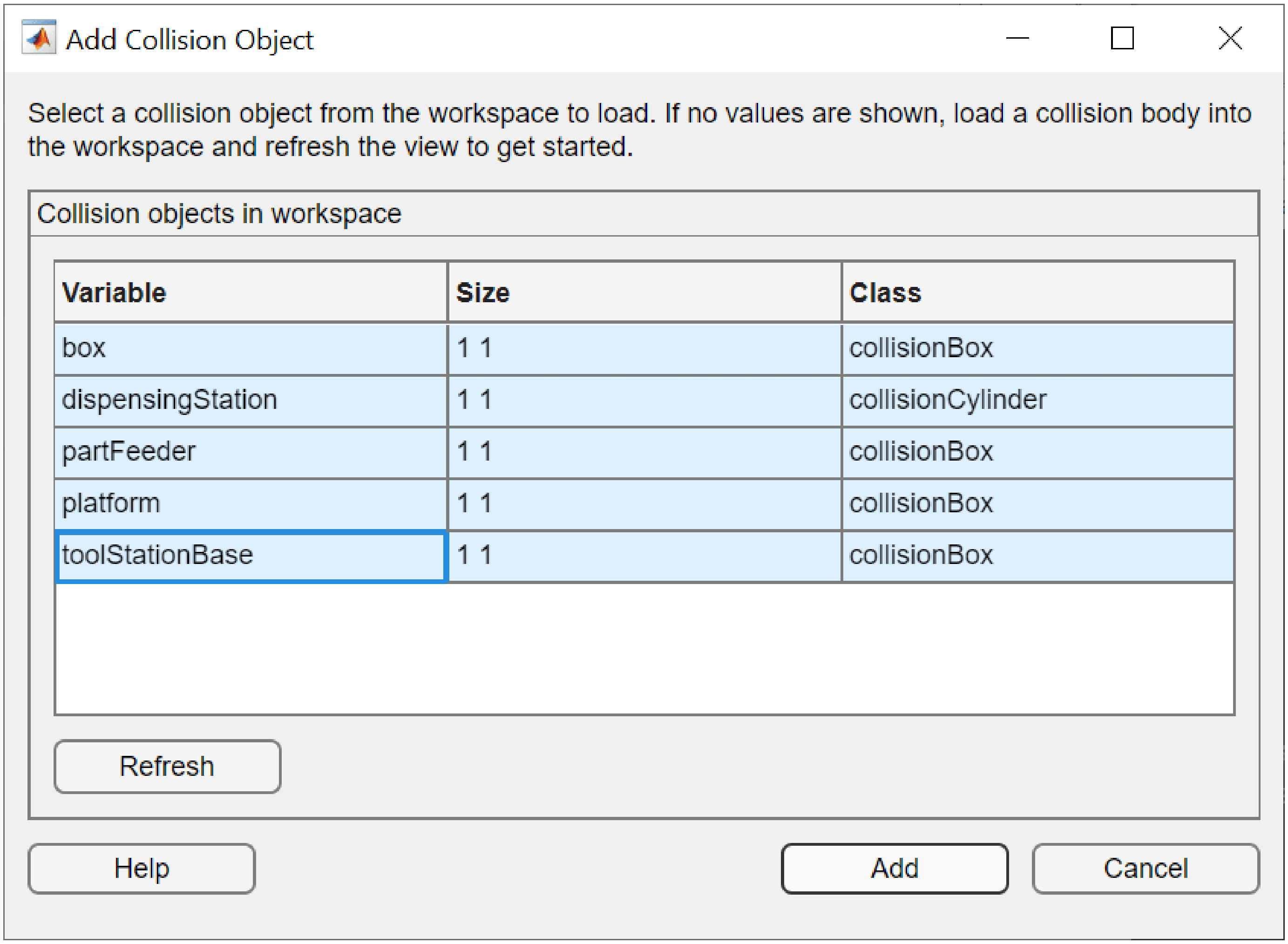
Load from workspace...

Rigid body tree objects in workspace

Variable	Size	Class
customEE	1 1	rigidBodyTree
ur5e_robot	1 1	rigidBodyTree

Add Environment to Scene

To load the environment model, click **Scene > Add Collision Object** in the **Inverse Kinematics** tab. This will bring up the **Add Collision Object** dialog, which shows all the available collision objects in the MATLAB® workspace. Shift-click to select the box, dispensing station, tool station base, part feeder. and platform.



Add Collision Object

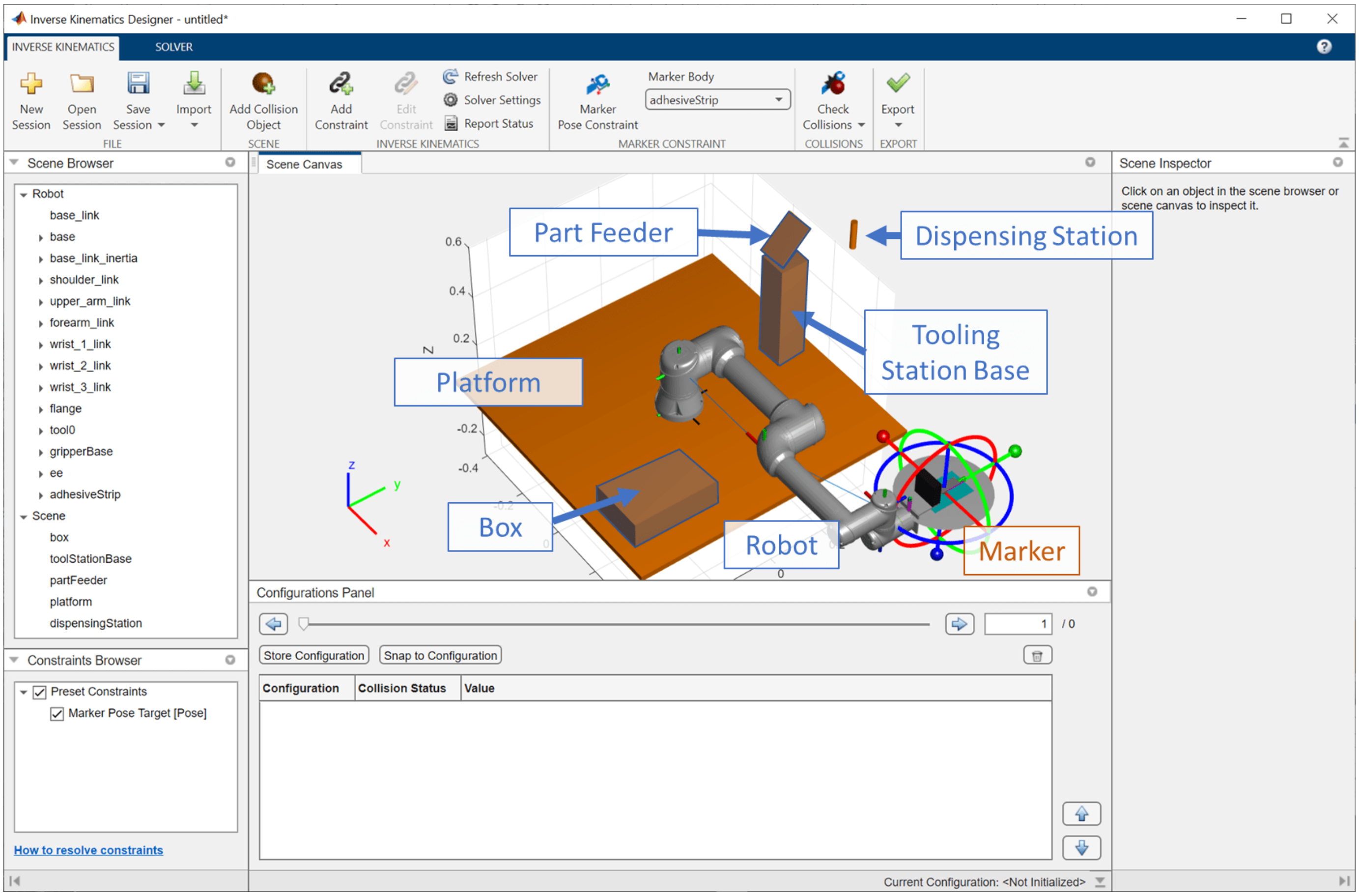
Select a collision object from the workspace to load. If no values are shown, load a collision body into the workspace and refresh the view to get started.

Collision objects in workspace

Variable	Size	Class
box	1 1	collisionBox
dispensingStation	1 1	collisionCylinder
partFeeder	1 1	collisionBox
platform	1 1	collisionBox
toolStationBase	1 1	collisionBox

Click **Add** to add these objects to the scene. Then use the **Axes Toolbar** on the figure in the **Scene Canvas** to reposition to scene so it fits. See “Use Scene Canvas and Move Robot” for more information.

The **Inverse Kinematics Designer** session is now set up, containing the scene elements and robot, which can be interactively guided by the marker.



Click on an object in the scene browser or scene canvas to inspect it.

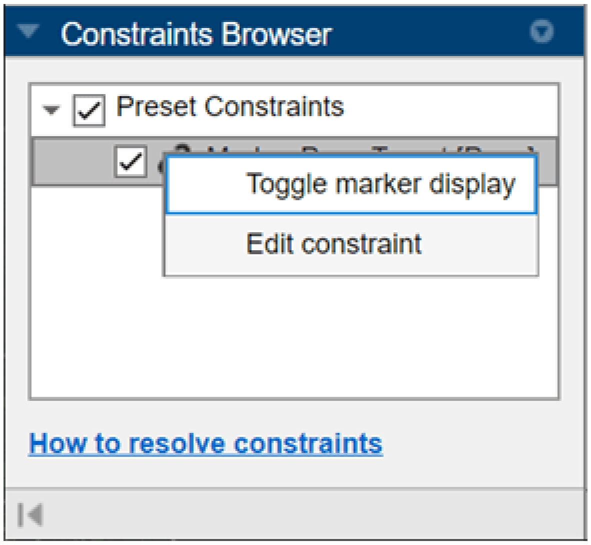
Current Configuration: <Not Initialized>

Add Waypoints for each Task

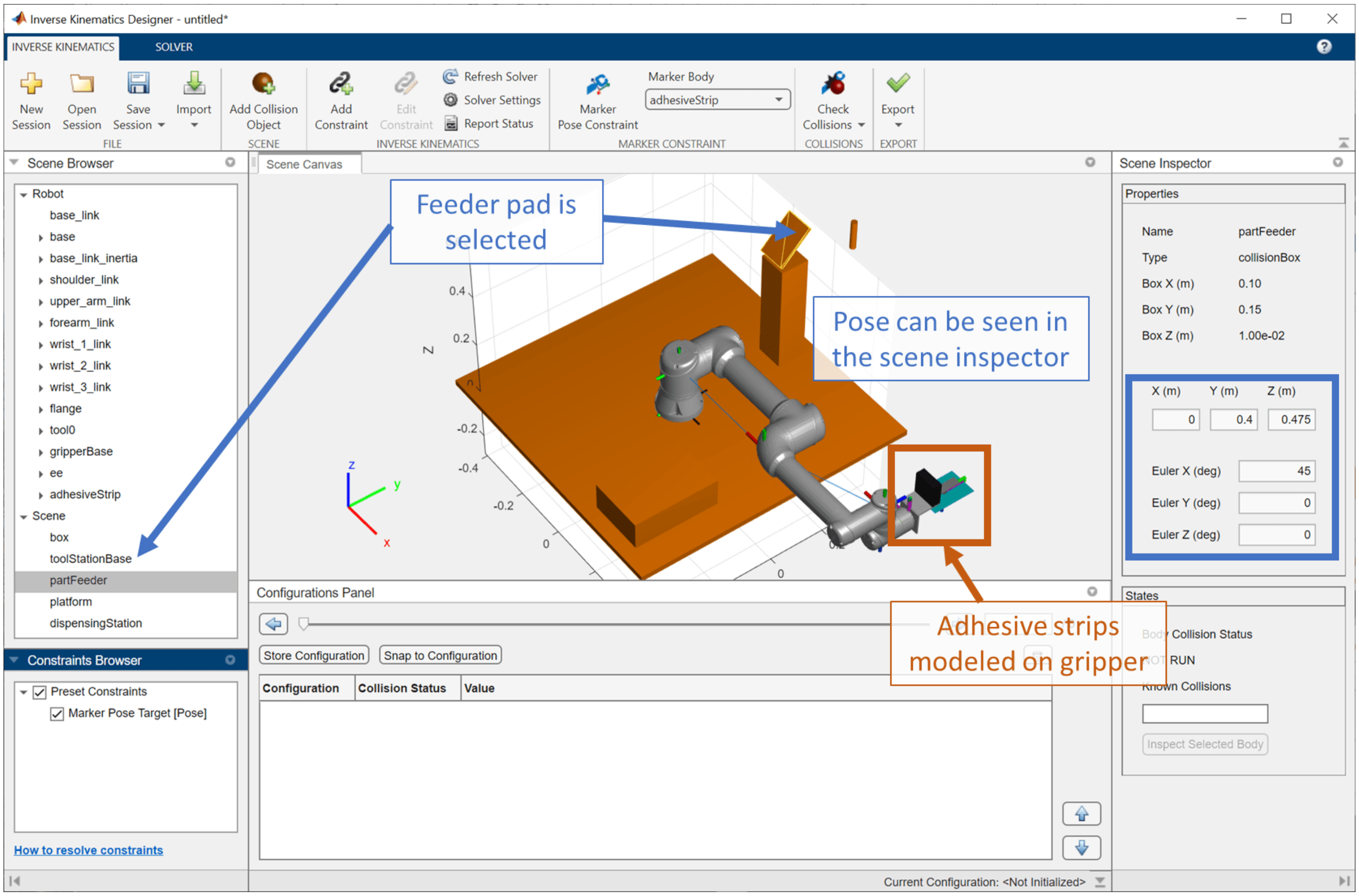
Find the configurations that correspond to the waypoints for each task. Constraints will be used along the way to ensure that each configuration satsifies the target pose.

Pick up the Adhesive Strips from the Part Feeders

The two flat panels simulation adhesive grip attachments on the gripper must align with the pads at the feeding station. To see these parts, it is helpful to temporarily disable the marker visuals. Right-click on the **Marker Pose Constraint** in the **Constraints Browser** and select **Toggle marker display** to disable the marker visuals.



Now click on `partFeeder` in the **Scene Browser**. To ensure the parts are properly picked, the goal will be to place the modeled attached strip on the gripper so that it aligns with the part feeder. Read the pose of the object from the scene inspector. The object is located at `[-.05 .4 .525]` and rotated 45 degrees about the world X axis.



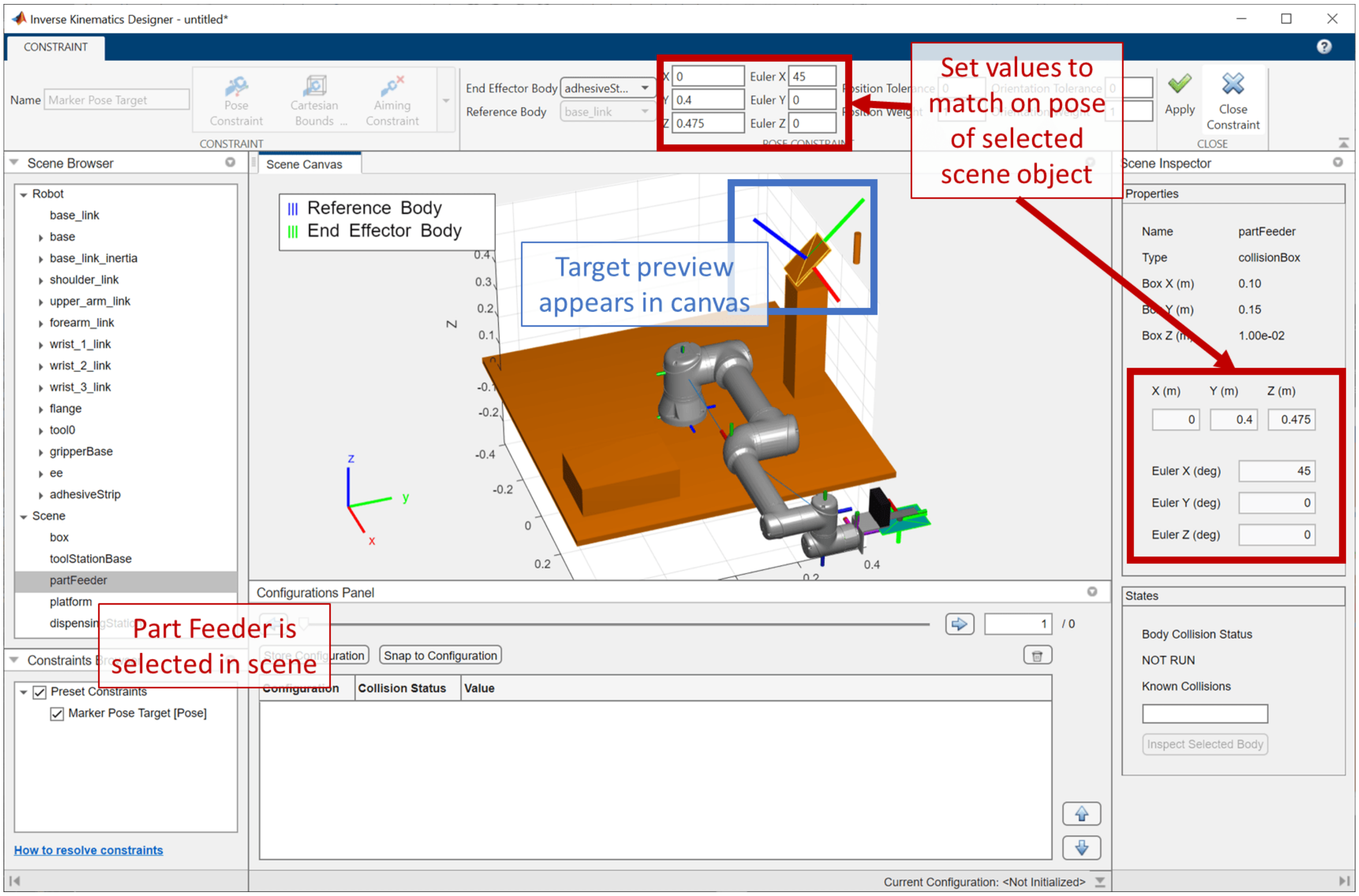
Properties

Name	partFeeder
Type	collisionBox
Box X (m)	0.10
Box Y (m)	0.15
Box Z (m)	1.00e-02

X (m)	Y (m)	Z (m)
0	0.4	0.475

Euler X (deg)	45
Euler Y (deg)	0
Euler Z (deg)	0

Move marker pose target to this pose so that a configuration is found in which the robot is aligned in the picking task pose. While it could be possible to achieve this via hand tuning, the most direct way is to directly set the end effector pose. Click on **Marker Pose Constraint** in the toolstrip, or alternatively select the **Marker Pose Constraint** in the **Constraints Browser** and selecting **Edit Constraint**. Once the constraint appears in the toolstrip, enter the pose of the object, `[-.05 .4 .525]` and rotated `45` degrees about the world X. As each value is added, the pose axis visual preview in the scene updates to indicate the new target pose.


Inverse Kinematics Designer - untitled*

Name: Marker Pose Target

End Effector Body: adhesiveSt...

Reference Body: base_link

X: 0 Euler X: 45
Y: 0.4 Euler Y: 0
Z: 0.475 Euler Z: 0

Set values to match on pose of selected scene object

Target preview appears in canvas

Part Feeder is selected in scene

Scene Inspector — Properties

Name: partFeeder
Type: collisionBox
Box X (m): 0.10
Box Y (m): 0.15
Box Z (m): 1.00e-02

X (m): 0 Y (m): 0.4 Z (m): 0.475

Euler X (deg): 45
Euler Y (deg): 0
Euler Z (deg): 0

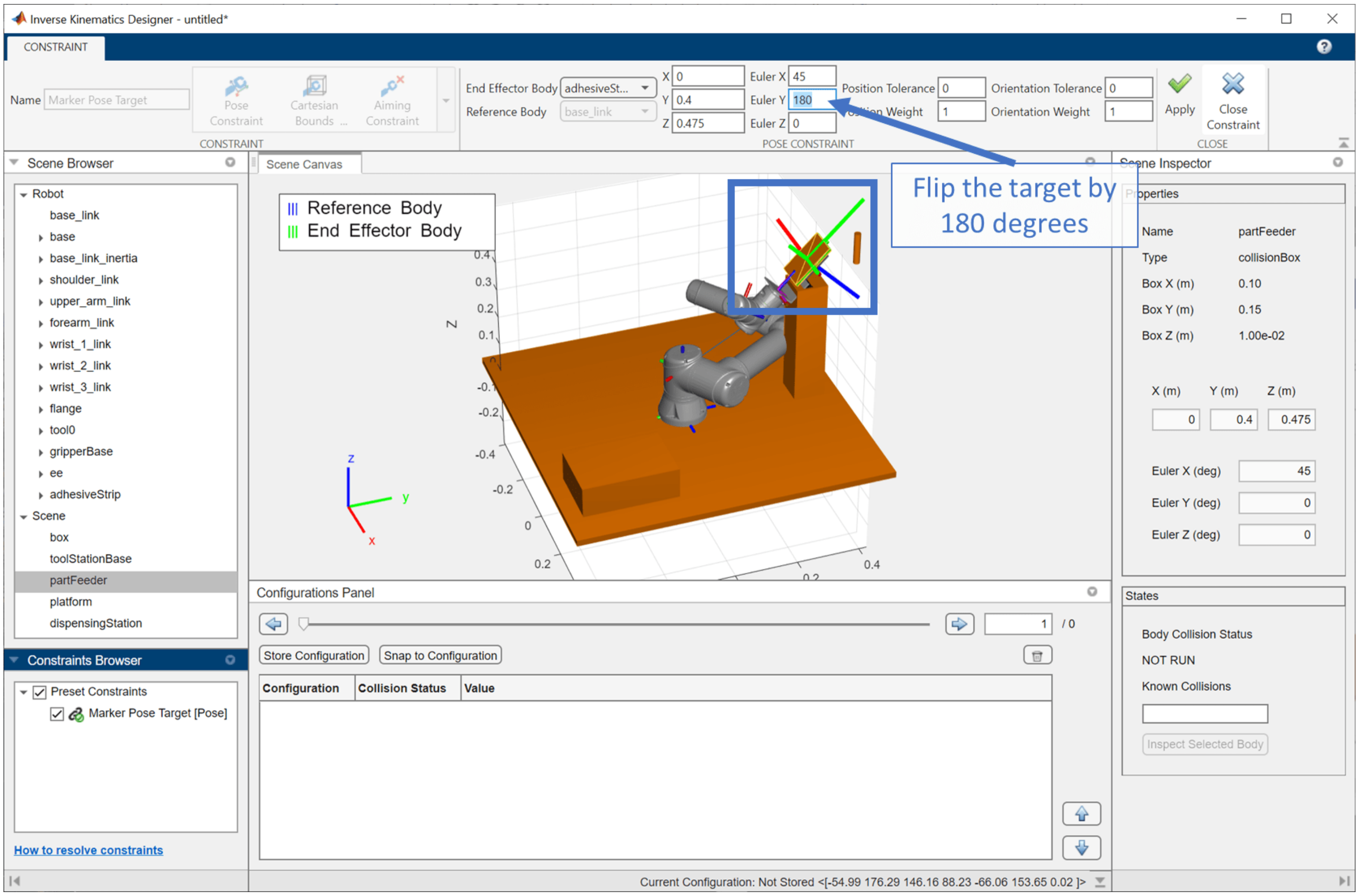
States

Body Collision Status: NOT RUN

Known Collisions

Current Configuration: <Not Initialized>

Click **Apply** in the **Constraint** tab to set this value as the target. This updates the pose, but the robot is backwards from the intended direction. Add a rotation of `180` degrees about the Y axis to flip the target pose.



Name Marker Pose Target

End Effector Body adhesiveSt...
Reference Body base_link

X 0
Y 0.4
Z 0.475

Euler X 45
Euler Y 180
Euler Z 0

Position Tolerance 0
Position Weight 1
Orientation Tolerance 0
Orientation Weight 1

Flip the target by 180 degrees

- Robot
  - base_link
  - base
  - base_link_inertia
  - shoulder_link
  - upper_arm_link
  - forearm_link
  - wrist_1_link
  - wrist_2_link
  - wrist_3_link
  - flange
  - tool0
  - gripperBase
  - ee
  - adhesiveStrip
- Scene
  - box
  - toolStationBase
  - partFeeder
  - platform
  - dispensingStation

- ☑ Preset Constraints
  - ☑ Marker Pose Target [Pose]

Properties

Name	partFeeder
Type	collisionBox
Box X (m)	0.10
Box Y (m)	0.15
Box Z (m)	1.00e-02

X (m)	Y (m)	Z (m)
0	0.4	0.475

Euler X (deg) 45
Euler Y (deg) 0
Euler Z (deg) 0

States

Body Collision Status
NOT RUN
Known Collisions

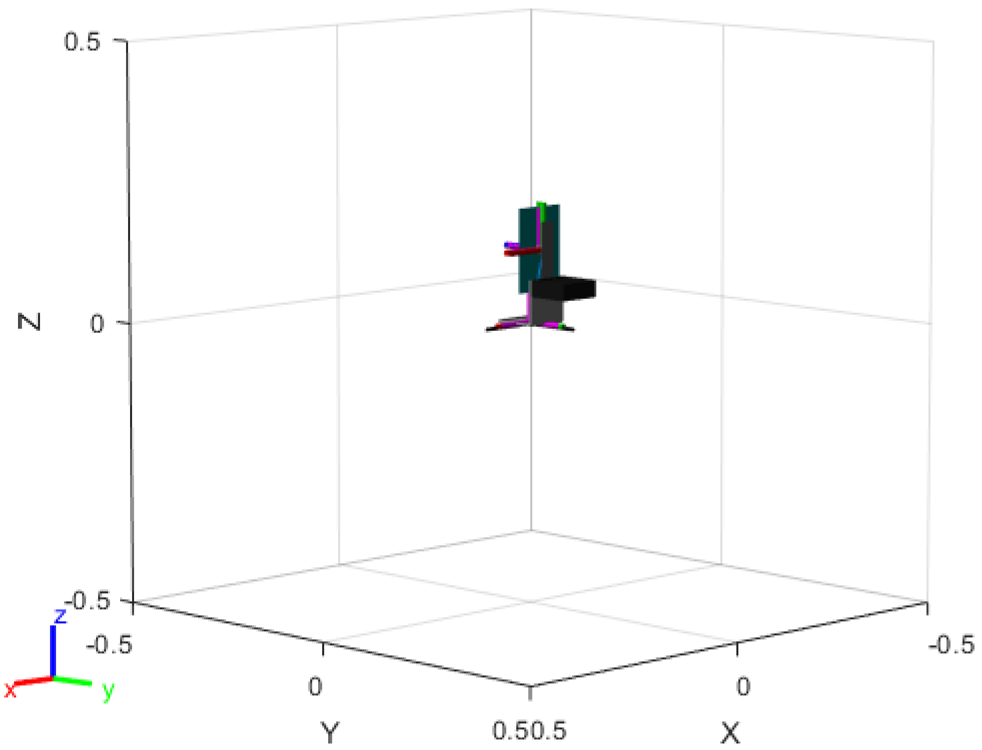
1 / 0

Configuration	Collision Status	Value

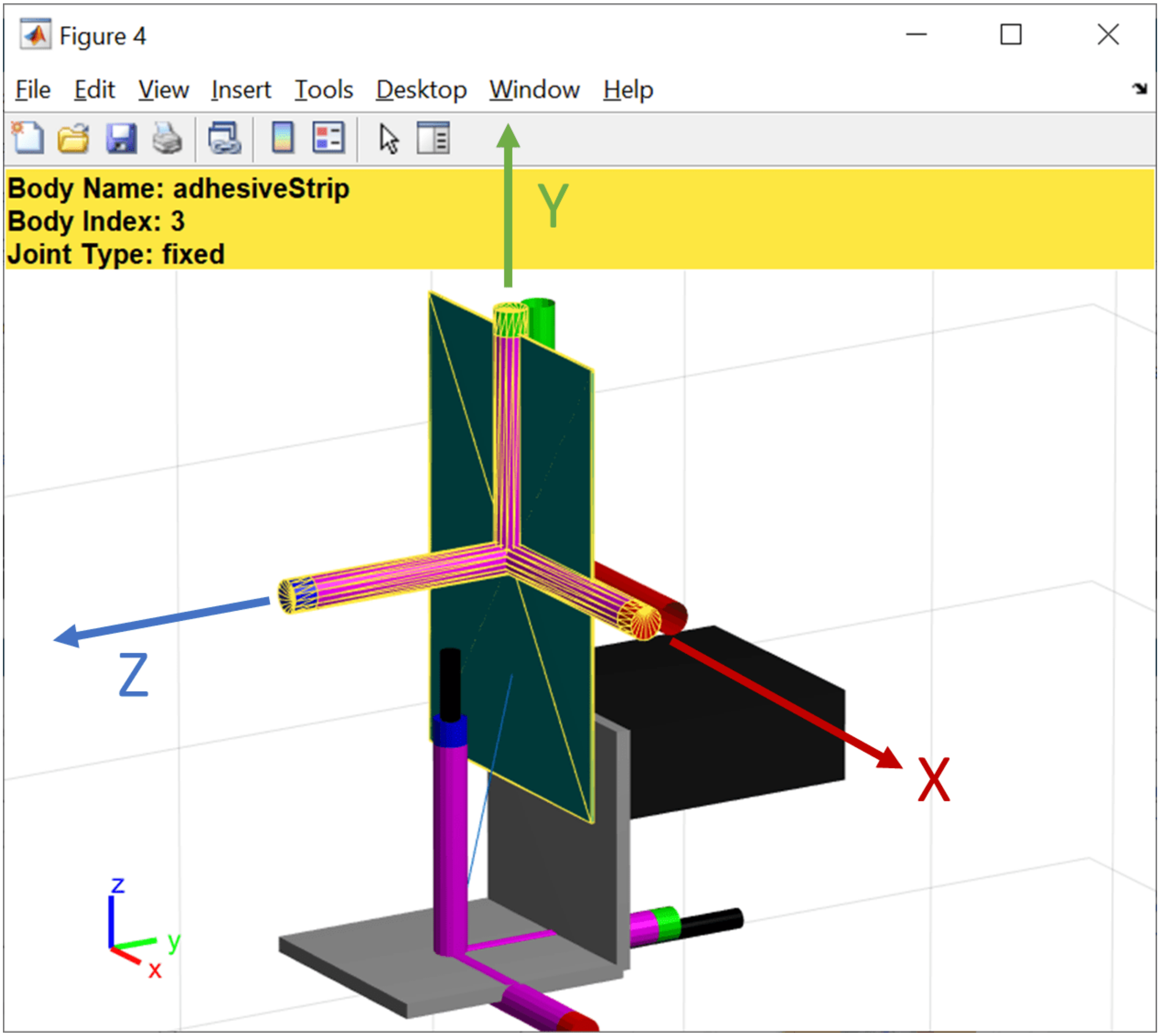
Current Configuration: Not Stored <[-54.99 176.29 146.16 88.23 -66.06 153.65 0.02]>

The necessary orientation of the end effector can be determined ahead of time by examining the gripper.

```
show(customEE);
```



Select the adhesive strip to highlight the orientation of its origin and reference frame.



Body Name: adhesiveStrip
Body Index: 3
Joint Type: fixed

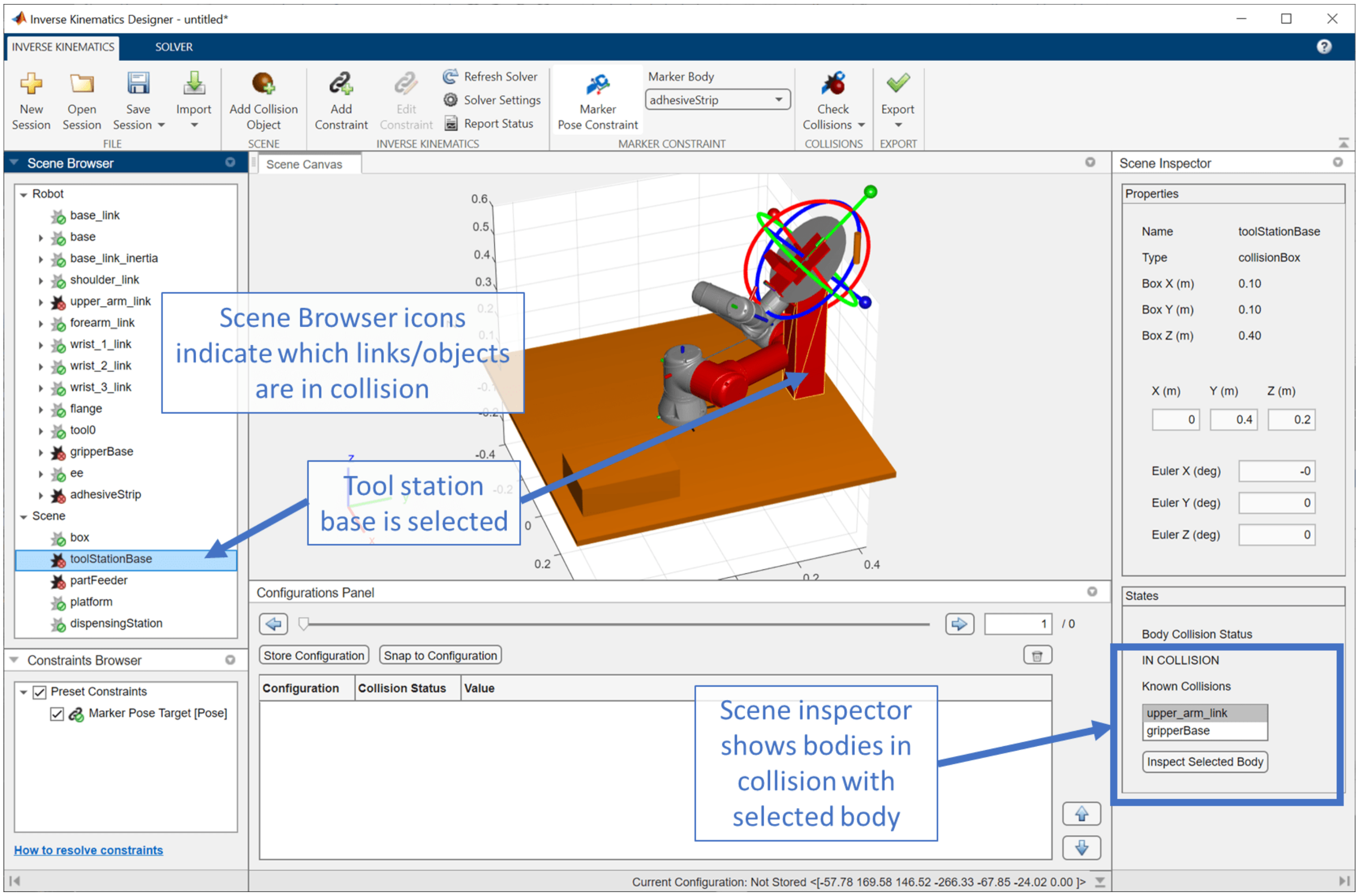
As indicated by the red, green, and blue tip highlighting on the violet reference joint marker, the Z axis is orthogonal to the part, while the Y axis points away from the L bracket that attaches to the gripper.

Therefore, the following rules must be satisfied regarding part orientation:

- When when picking up the part, the Z axis must point at the feeder, and the Y axis should be in-line with the global Y axis.
- When applying glue, the Z axis must point at the dispenser.
- When applying a part, the Z axis should be pointed at the part, with the Y axis pointed down, so that the L bracket is above the part.

Adjust Configuration to Avoid Collisions

Determine which bodies are in collision by selecting **Check Collisions** in the toolstrip. The red highlighting in the scene and the icons on the left show the objects in collision. Selecting an object shows the objects it is colliding with in the scene inspector.



Scene Browser icons indicate which links/objects are in collision

Tool station base is selected

Scene inspector shows bodies in collision with selected body

Name	toolStationBase
Type	collisionBox
Box X (m)	0.10
Box Y (m)	0.10
Box Z (m)	0.40

X (m)	Y (m)	Z (m)
0	0.4	0.2

Euler X (deg)	-0
Euler Y (deg)	0
Euler Z (deg)	0

Body Collision Status

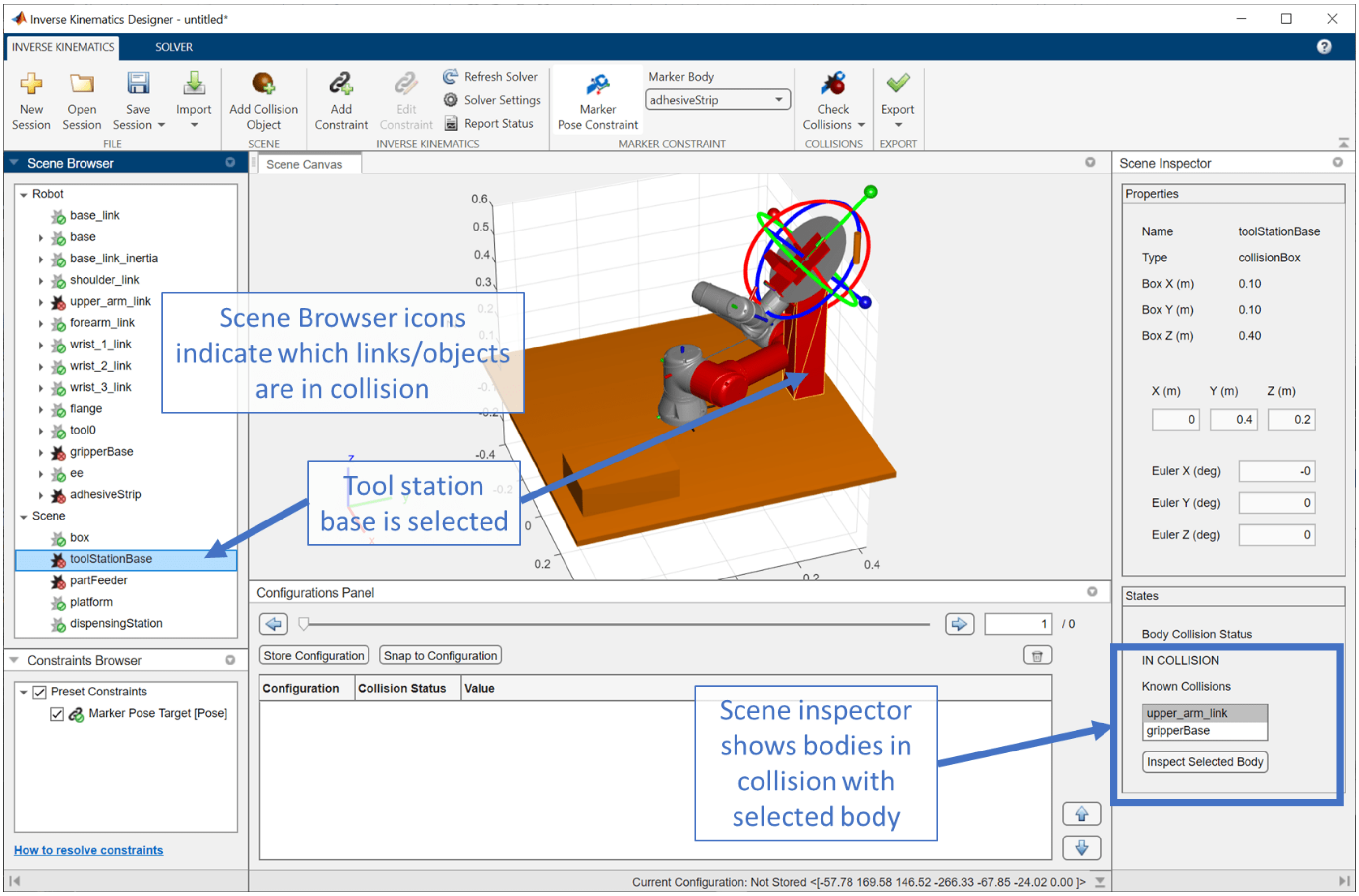
IN COLLISION

Known Collisions

upper_arm_link
gripperBase

Current Configuration: Not Stored <[-57.78 169.58 146.52 -266.33 -67.85 -24.02 0.00]>

From this view, see that the upper arm and gripper base links are in collision with the tool station base, and the adhesive strip is in collision with the part feeder.



Scene Browser icons indicate which links/objects are in collision

Tool station base is selected

Scene inspector shows bodies in collision with selected body

Name	toolStationBase
Type	collisionBox
Box X (m)	0.10
Box Y (m)	0.10
Box Z (m)	0.40

X (m)	Y (m)	Z (m)
0	0.4	0.2

Euler X (deg)	-0
Euler Y (deg)	0
Euler Z (deg)	0

Body Collision Status

IN COLLISION

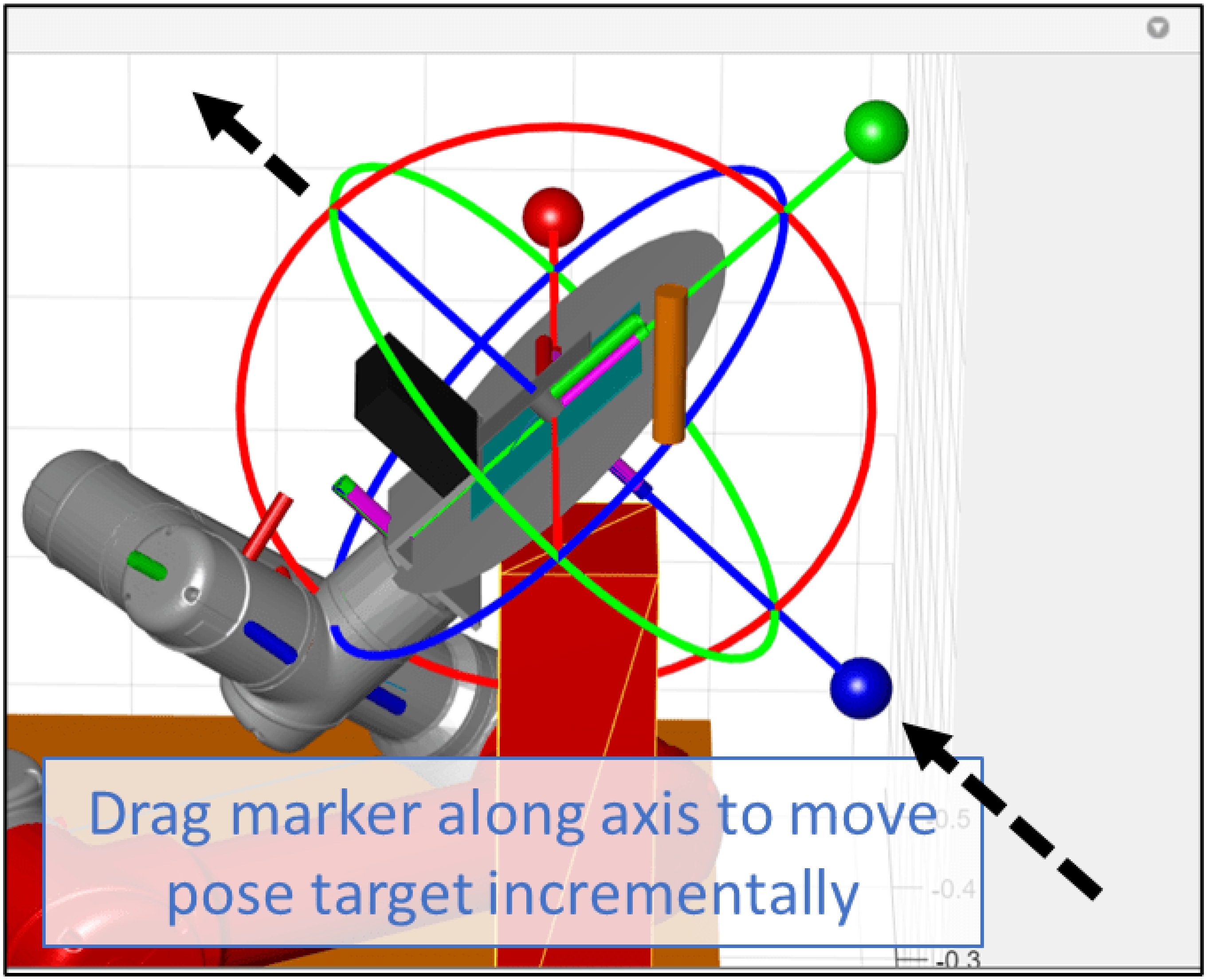
Known Collisions

upper_arm_link
gripperBase

Current Configuration: Not Stored <[-57.78 169.58 146.52 -266.33 -67.85 -24.02 0.00]>

From this view, it can be seen that the upper arm and gripper base links are in collision with the tool station base, and the adhesive strip is in collision with the part feeder. The latter is expected as the original marker pose target coinsides with the same position as the part feeder. However, assuming the part will lay flat on the feeder, the end effector can be retracted such that the end effector has enough clearance to pick up the part.

To make this change, click on the lateral Z axis grip of the marker, and drag the marker along its local Z axis.

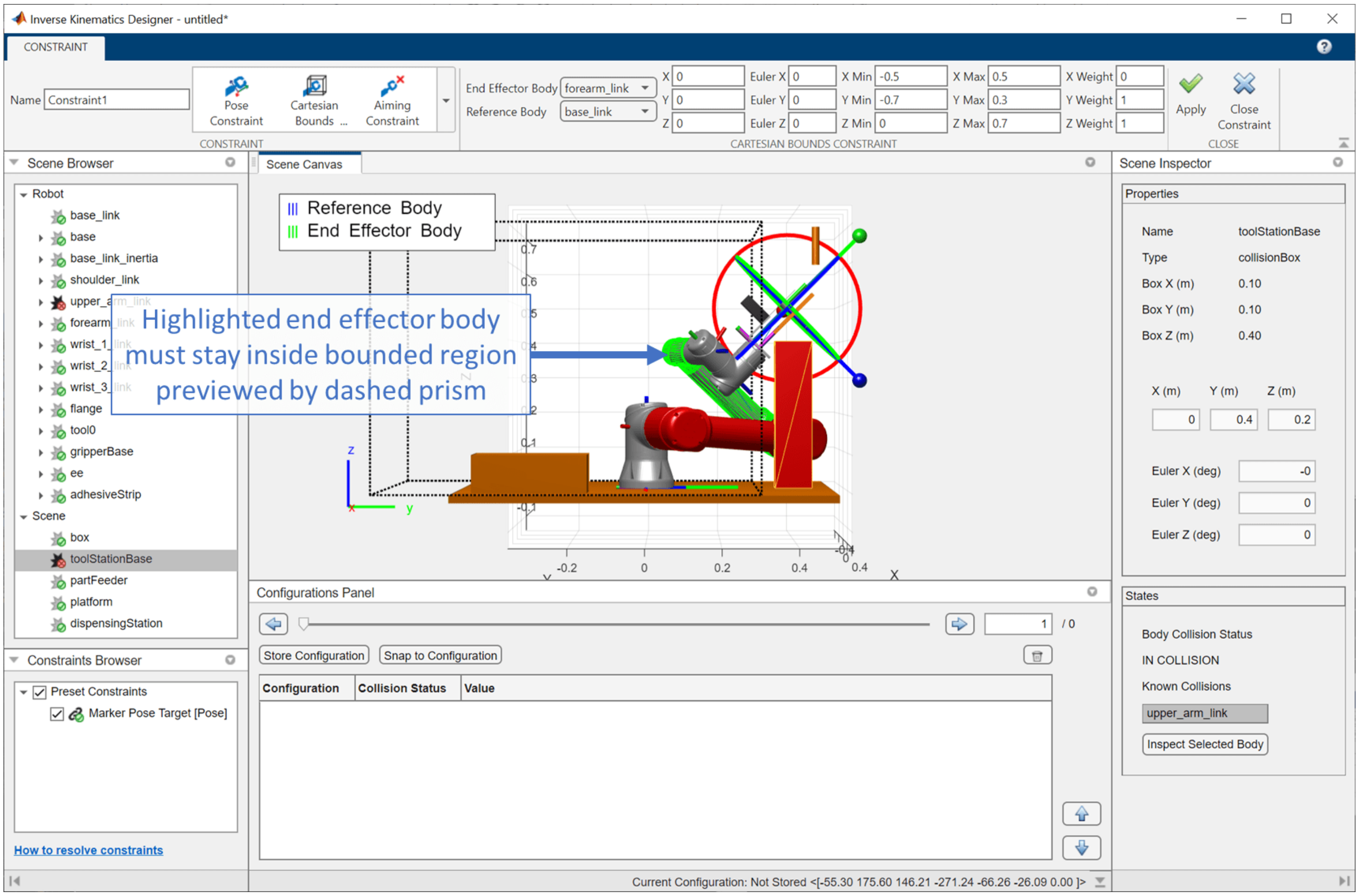


Click **Check Collision** again to verify that the adhesive and gripper bodies are no longer in collision.

The second collision is between the tool station base and two links in the robot. To avoid these in-collision configurations, use constraints such as the Cartesian bounds constraint to adjust the solver so that it no longer finds configurations in which the forearm and upper arm can collide with the base.

A bounds constraint can be used to keep bodies inside a particular bounded range. In this case, the aim would be to keep the distal end of the upper arm from contacting the base. Since that end of the link is near coincident with the original (and reference point) of the forearm link, a Cartesian Bounds constraint that prevents the forearm from reaching the base along Y should prevent this collision. To add this, click **Add Constraint > Cartesian Bounds Constraint**.

Create a target bounded region for the manipulator that avoids the tooling station base. Set the default upper Y bound to `0.3` meters and the default upper Z bound to `0.7` meters creates a region for this body to be in that avoids the base. Additionally, set the weight on the X direction to `0` since the range in that direction is unlimited. Observe the the preview in the image below.



Name: Constraint1

End Effector Body: forearm_link

Reference Body: base_link

X 0	Euler X 0	X Min -0.5	X Max 0.5	X Weight 0
Y 0	Euler Y 0	Y Min -0.7	Y Max 0.3	Y Weight 1
Z 0	Euler Z 0	Z Min 0	Z Max 0.7	Z Weight 1

Highlighted end effector body must stay inside bounded region previewed by dashed prism

1 / 0

Name	toolStationBase
Type	collisionBox
Box X (m)	0.10
Box Y (m)	0.10
Box Z (m)	0.40

X (m)	Y (m)	Z (m)
0	0.4	0.2

Euler X (deg)	-0
Euler Y (deg)	0
Euler Z (deg)	0

Body Collision Status

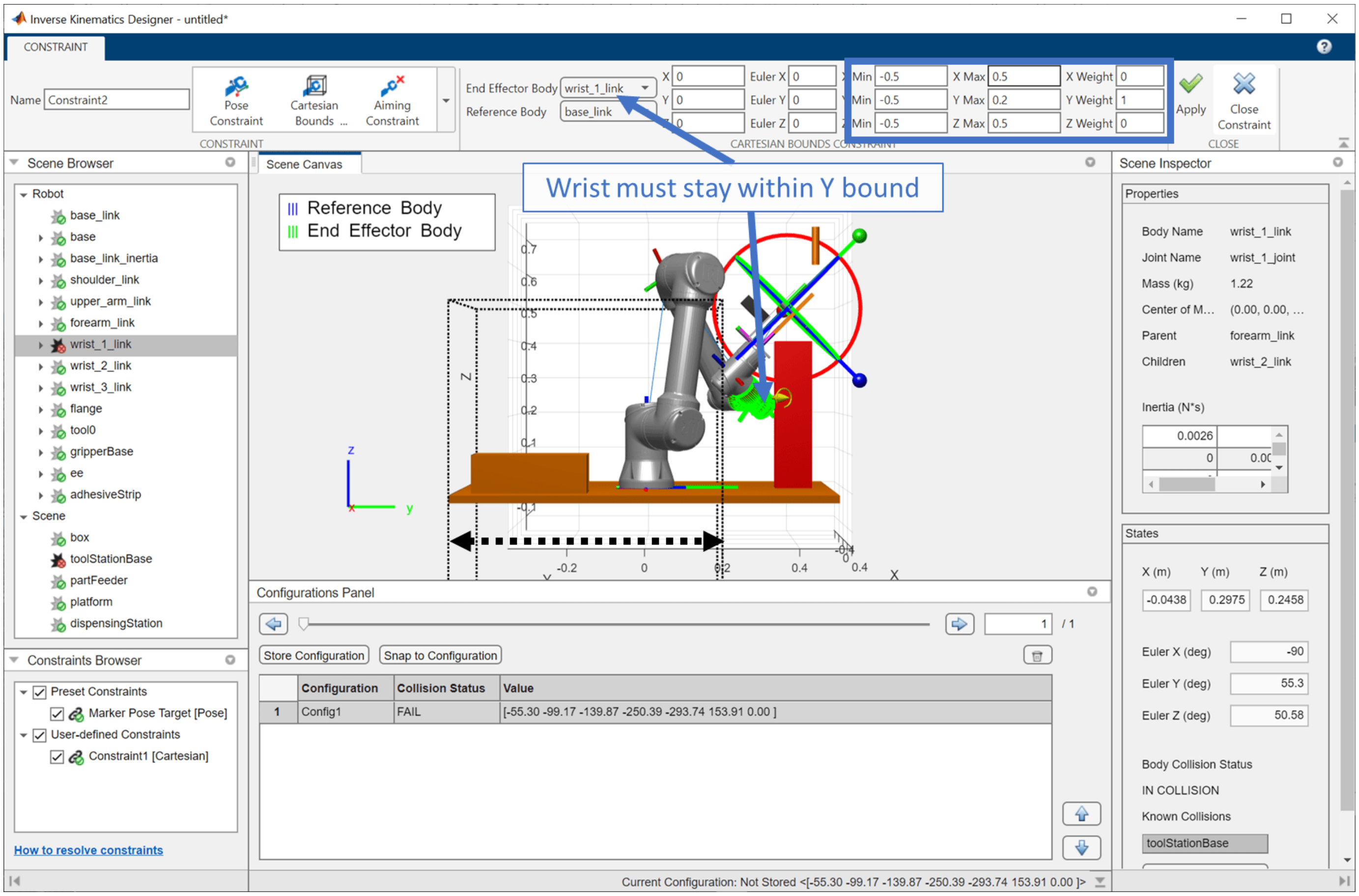
IN COLLISION

Known Collisions

upper_arm_link

Current Configuration: Not Stored <[-55.30 175.60 146.21 -271.24 -66.26 -26.09 0.00]>

Click **Apply** to apply the constraint, then close the constraints browser. The robot is now repositioned and collision-check confirms that while the arm links are no longer in collision with the base, the wrist is still in collision. Create another Cartesian bounds constraint with a Y range of `-0.5` meters to `0.2` meters to prevent this collision. Set the weights on the other bounds to `0`, so that only Y is considered.



Name: Constraint2

End Effector Body: wrist_1_link

Reference Body: base_link

X: 0 Euler X: 0 X Min: -0.5 X Max: 0.5 X Weight: 0

Y: 0 Euler Y: 0 Y Min: -0.5 Y Max: 0.2 Y Weight: 1

Z: 0 Euler Z: 0 Z Min: -0.5 Z Max: 0.5 Z Weight: 0

Wrist must stay within Y bound

1 / 1

	Configuration	Collision Status	Value
1	Config1	FAIL	[-55.30 -99.17 -139.87 -250.39 -293.74 153.91 0.00]

Current Configuration: Not Stored <[-55.30 -99.17 -139.87 -250.39 -293.74 153.91 0.00]>

Body Name: wrist_1_link

Joint Name: wrist_1_joint

Mass (kg): 1.22

Center of M...: (0.00, 0.00, ...

Parent: forearm_link

Children: wrist_2_link

Inertia (N*s)

0.0026	
0	0.0(

X (m): -0.0438 Y (m): 0.2975 Z (m): 0.2458

Euler X (deg): -90

Euler Y (deg): 55.3

Euler Z (deg): 50.58

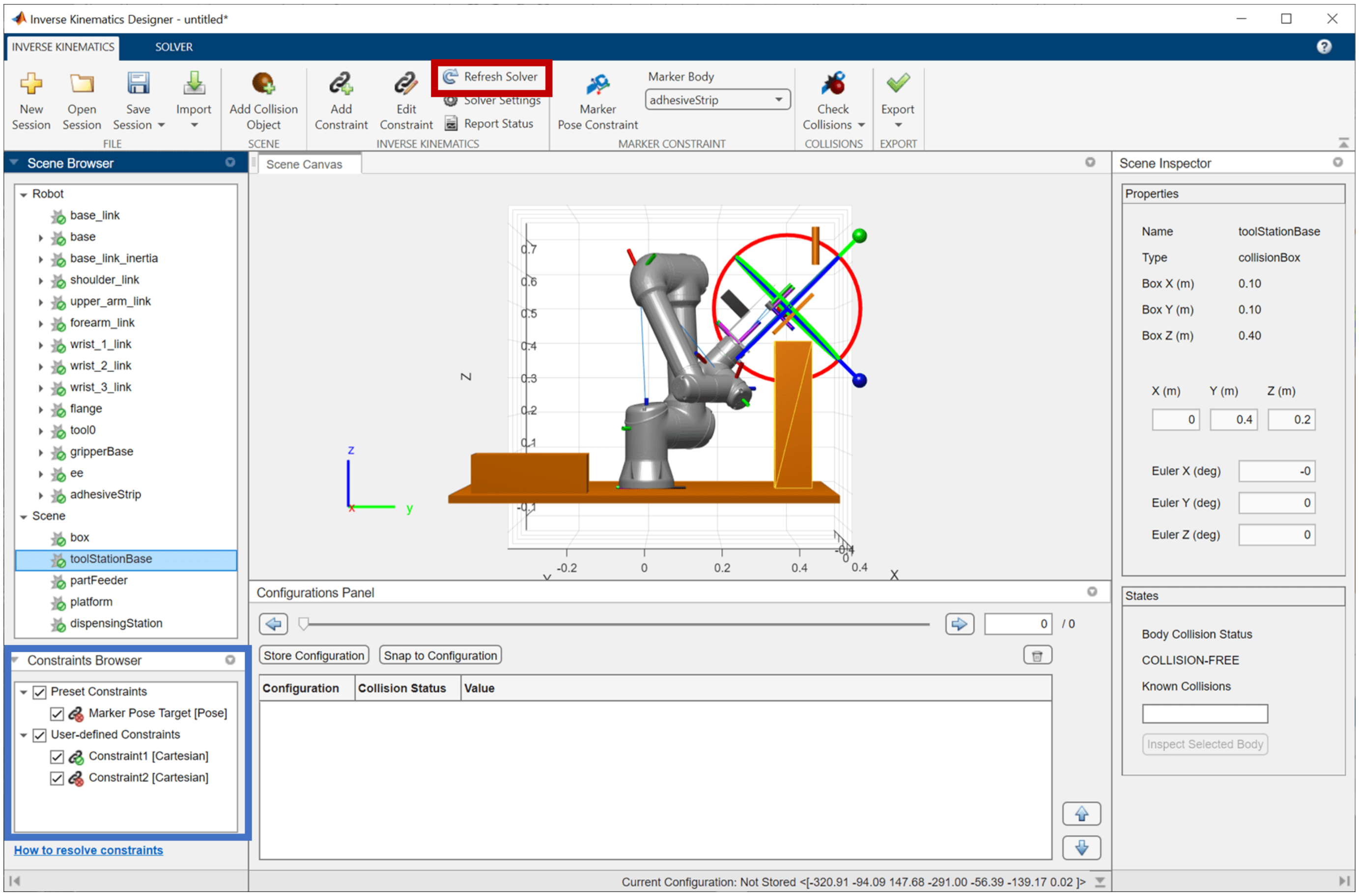
Body Collision Status

IN COLLISION

Known Collisions

toolStationBase

Apply the changes and exit the **Constraints** tab. Then click **Check Collisions** again verify that the configuration is collision-free.



Marker Body: adhesiveStrip

Name	toolStationBase
Type	collisionBox
Box X (m)	0.10
Box Y (m)	0.10
Box Z (m)	0.40

X (m)	Y (m)	Z (m)
0	0.4	0.2

Euler X (deg)	-0
Euler Y (deg)	0
Euler Z (deg)	0

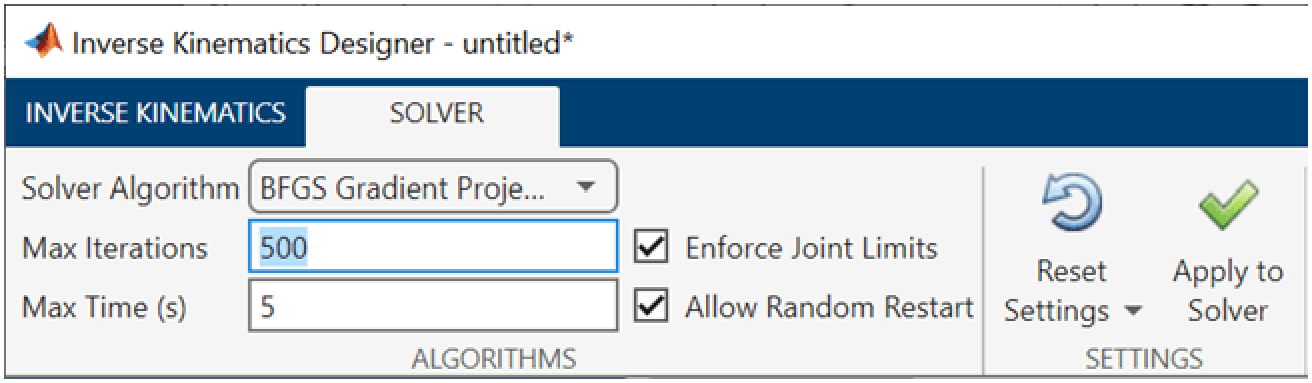
Body Collision Status
COLLISION-FREE
Known Collisions

0 / 0

Current Configuration: Not Stored <[-320.91 -94.09 147.68 -291.00 -56.39 -139.17 0.02]>

In the case above, while the solution is now collision-free and visually close to the target, the marker pose target constraint is not met. Click **Refresh Solver** to attempt to see if the solver can find another solution. Try this a few times. If the solver cannot find a solution where the marker pose target constraint is not met, modify the solver parameters so it can find a solution that satisfies all constraints during its execution time.

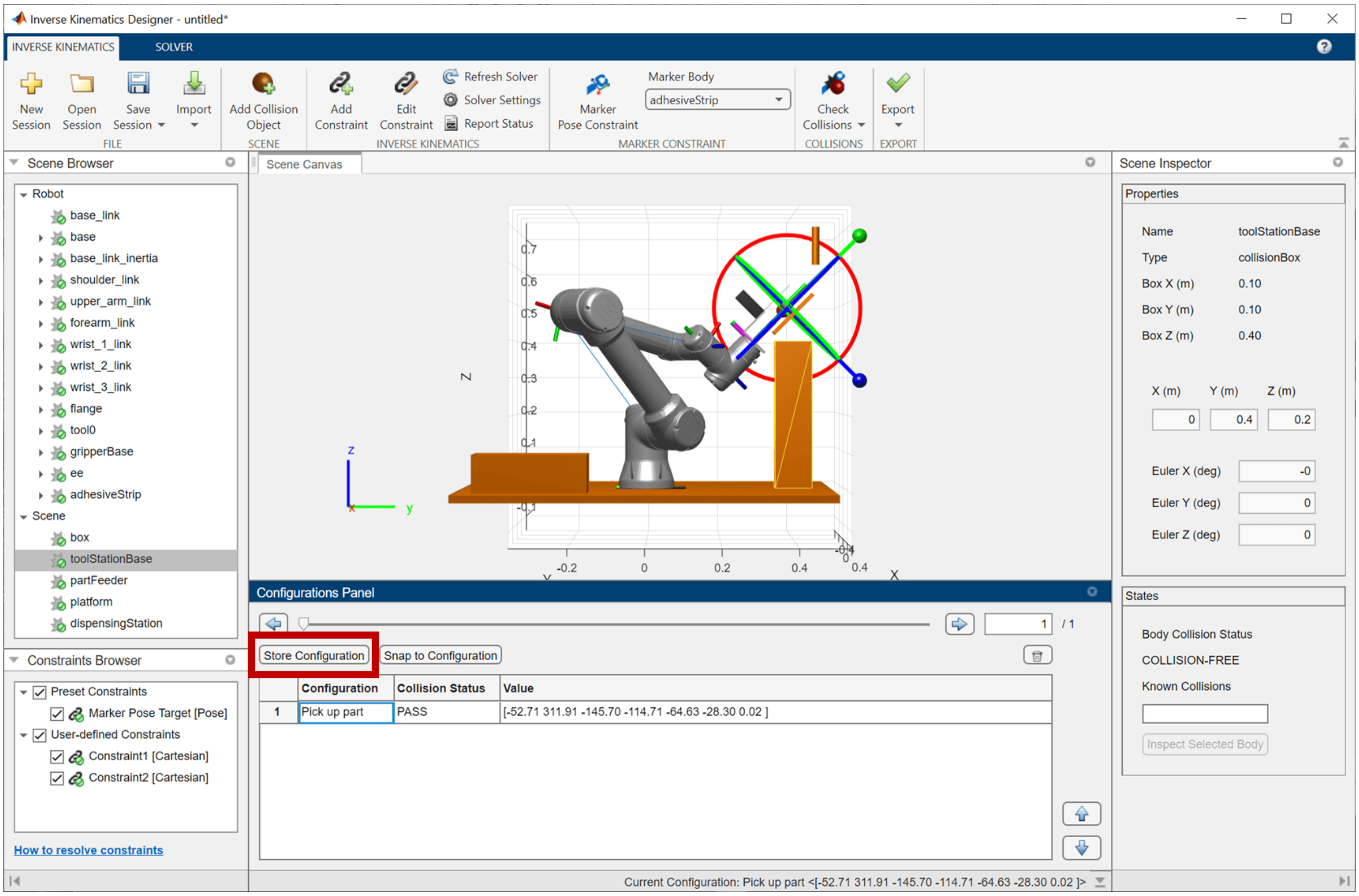
The default solver parameters are set to be fairly fast with just `50` iterations, which can struggle with a higher number of constraints. In the toolstrip, click **Solver Settings** or select the **Solver** tab. Then set the value of the number of **Maximum Iterations** to `500`. Click **Apply** to run the solver and return to the **Inverse Kinematics** tab.



Solver Algorithm: BFGS Gradient Proje...
Max Iterations: 500
Max Time (s): 5
☑ Enforce Joint Limits
☑ Allow Random Restart

The robot should now be in a configuration that is collision-free and meets all constraints.

Once satisfied with the configuration, store it by clicking **Store Configuration** in the **Configurations Panel**. Rename the configuration to `Pick up part`.



Name	toolStationBase
Type	collisionBox
Box X (m)	0.10
Box Y (m)	0.10
Box Z (m)	0.40

X (m)	Y (m)	Z (m)
0	0.4	0.2

Euler X (deg)	-0
Euler Y (deg)	0
Euler Z (deg)	0

Body Collision Status
COLLISION-FREE
Known Collisions

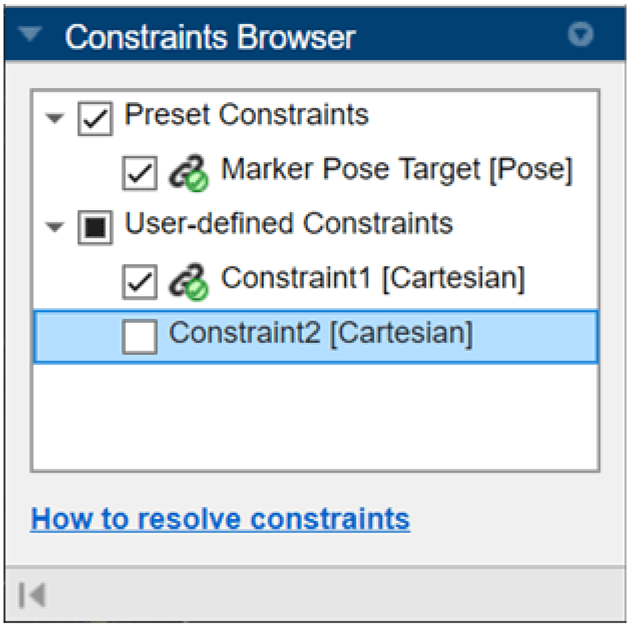
	Configuration	Collision Status	Value
1	Pick up part	PASS	[-52.71 311.91 -145.70 -114.71 -64.63 -28.30 0.02]

Current Configuration: Pick up part <[-52.71 311.91 -145.70 -114.71 -64.63 -28.30 0.02]>

The method used above to find a solution, i.e. adding constraints and extending solver run-time, is one way to ensure a result can be reached. Another method is to give the solver different starting configurations. Since each call to inverse kinematics uses the current configuration as the initial guess, you can "help" the solver by putting the robot in a pose that appears near satisfaction. This may be done by setting selecting links and modifying their joint values directly in the scene inspector, or by using the marker pose target to move the robot around. You can also continue to click **Refresh Solver** or modify the parameters further to search for different configurations if unsatisfied with the current one.

Apply Glue to Part at Dispensing Station

The next pose is below the dispensing station. For this constraint to be reachable, the second cartesian constraint must be removed. De-select the constraint by clicking the checkbox next to `Constraint2 [Cartesian]`. Alternatively, right-click on it and delete it since it is no longer needed.

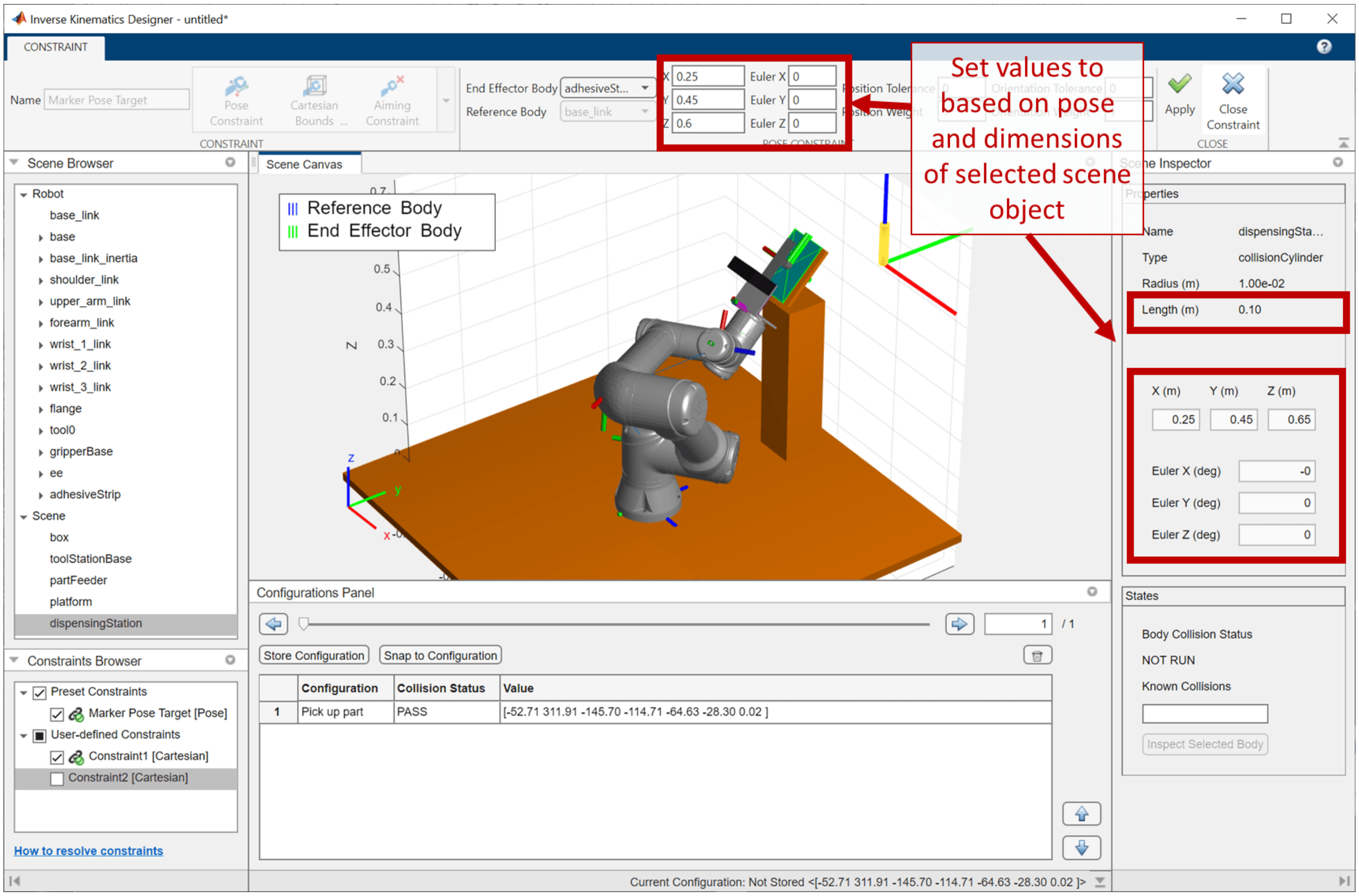


Constraints Browser

- ☑ Preset Constraints
  - ☑ Marker Pose Target [Pose]
- ■ User-defined Constraints
  - ☑ Constraint1 [Cartesian]
  - □ Constraint2 [Cartesian]

How to resolve constraints

Next, click on the dispensing station to determine its global pose and dimensions, then modify the marker target pose so it lies at the base of the dispensing station. Since the dispensing station is located at `[0.25 0.45 0.65]` meters and has length `0.1` meters, set the target pose to `0.05` meters below the origin of the dispensing station at `[0.25 0.45 0.6]` meters. With the orientation set to `[0 0 0]` degrees so the Z axis of the adhesive strip is pointing at the dispensing station.



Name Marker Pose Target

End Effector Body adhesiveSt...
Reference Body base_link

X 0.25 Euler X 0
Y 0.45 Euler Y 0
Z 0.6 Euler Z 0

Position Tolerance 0 Orientation Tolerance 0
Position Weight

Set values to based on pose and dimensions of selected scene object

1 / 1

	Configuration	Collision Status	Value
1	Pick up part	PASS	[-52.71 311.91 -145.70 -114.71 -64.63 -28.30 0.02]

Current Configuration: Not Stored <[-52.71 311.91 -145.70 -114.71 -64.63 -28.30 0.02]>

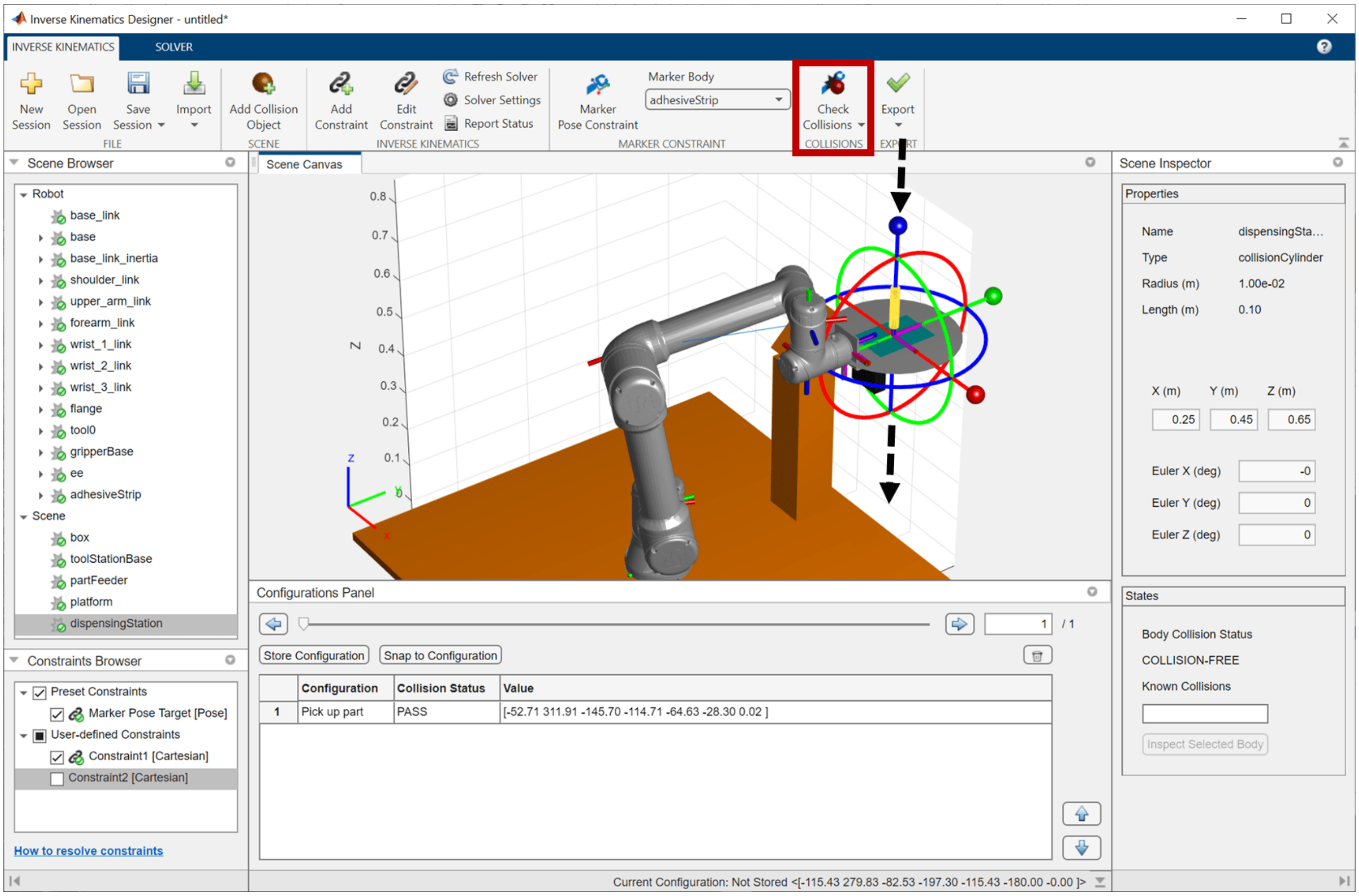
Name	dispensingSta...
Type	collisionCylinder
Radius (m)	1.00e-02
Length (m)	0.10

X (m)	Y (m)	Z (m)
0.25	0.45	0.65

Euler X (deg) -0
Euler Y (deg) 0
Euler Z (deg) 0

Body Collision Status
NOT RUN
Known Collisions

Apply the changes and exit the **Constraints** tab. Then drag the Z axis of the marker to provide some clearance between the part and the dispensing station. Check collisions and verify that the pose is collision-free.



☑ Preset Constraints
- ☑ Marker Pose Target [Pose]

■ User-defined Constraints
- ☑ Constraint1 [Cartesian]
- □ Constraint2 [Cartesian]

Configurations Panel: 1 / 1

	Configuration	Collision Status	Value
1	Pick up part	PASS	[-52.71 311.91 -145.70 -114.71 -64.63 -28.30 0.02]

Current Configuration: Not Stored <[-115.43 279.83 -82.53 -197.30 -115.43 -180.00 -0.00]>

Properties

Name	dispensingSta...
Type	collisionCylinder
Radius (m)	1.00e-02
Length (m)	0.10

X (m)	Y (m)	Z (m)
0.25	0.45	0.65

Euler X (deg)	-0
Euler Y (deg)	0
Euler Z (deg)	0

States

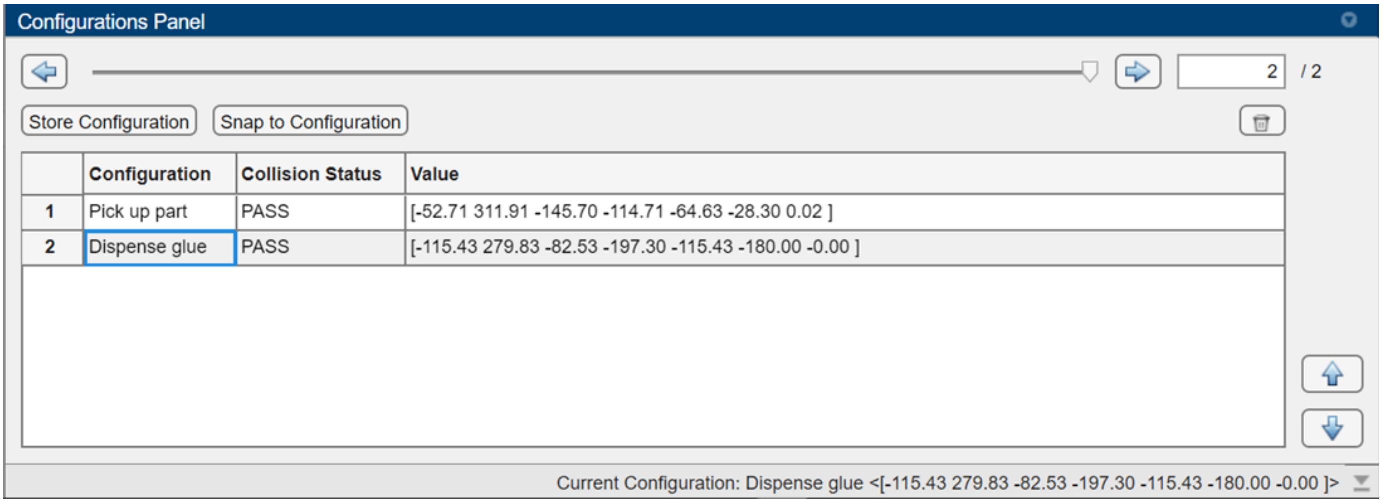
Body Collision Status

COLLISION-FREE

Known Collisions

Inspect Selected Body

Lastly, store the configuration and rename it to `Dispense glue`.



2 / 2

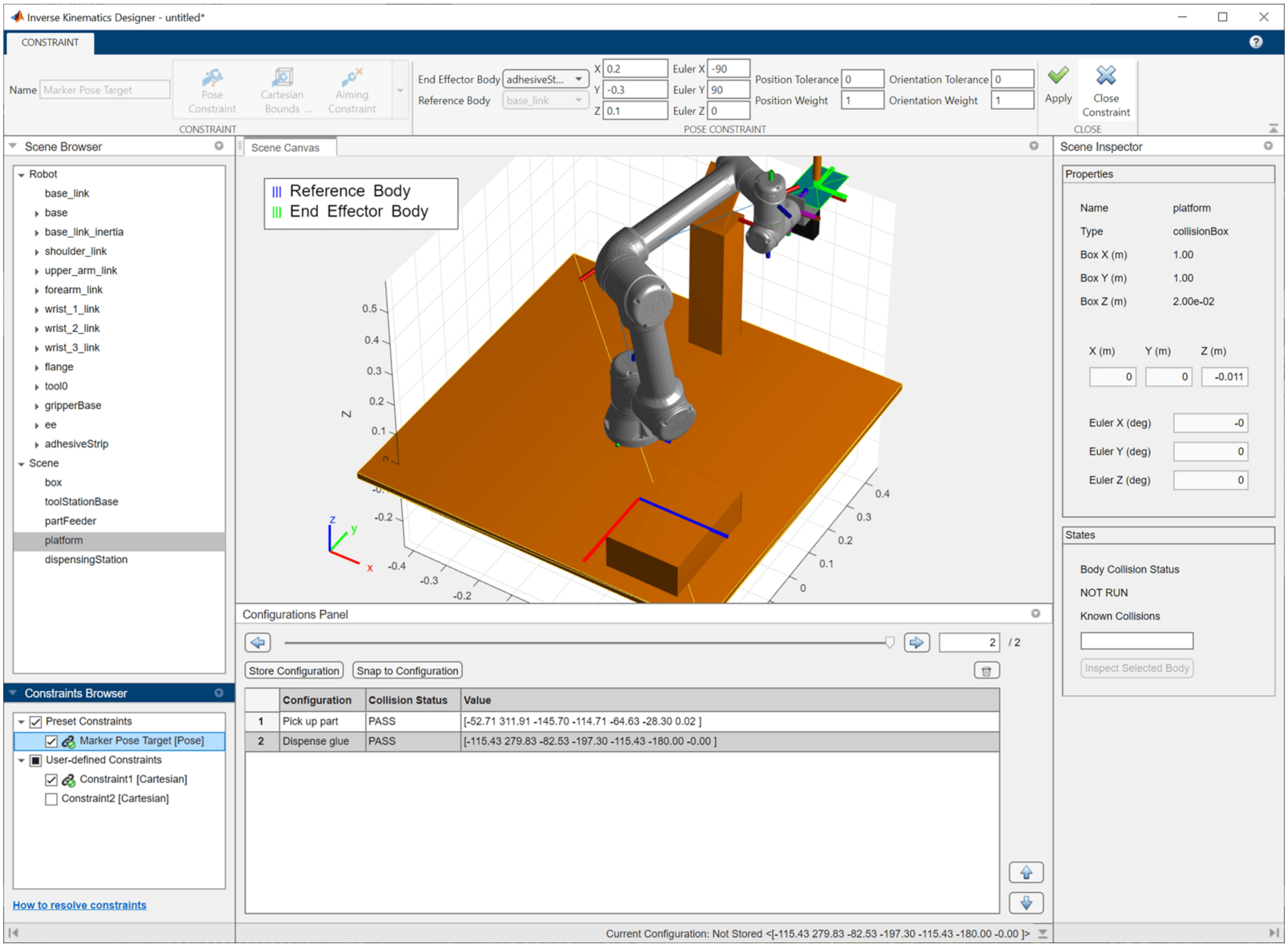
	Configuration	Collision Status	Value
1	Pick up part	PASS	[-52.71 311.91 -145.70 -114.71 -64.63 -28.30 0.02]
2	Dispense glue	PASS	[-115.43 279.83 -82.53 -197.30 -115.43 -180.00 -0.00]

Current Configuration: Dispense glue <[-115.43 279.83 -82.53 -197.30 -115.43 -180.00 -0.00]>

Apply Strips to Product

For the final configuration, the part will be applied to the box. As with the previous two configurations, the pose will be based off of the part location.

Start by determining the target pose. Click on the box, and choose a target that will orient the gripper orthogonally to the box with the L bracket above the box. Based on the position of the box, `[0.3 -0.3 -.05]`, the gripper should be placed in the center of the box pointing outward, but with clearance for the `0.15` meter long strip to be placed without colliding with the platform. An example of such a position is `[0.2 -0.3 0]` meters. Using the orientation rules described, the Z axis of the gripper must point towards the box, along the global X axis, and the Y axis of the gripper must point down, towards the platform.



Name: Marker Pose Target
End Effector Body: adhesiveSt...
Reference Body: base_link
X: 0.2 Y: -0.3 Z: 0.1
Euler X: -90 Euler Y: 90 Euler Z: 0
Position Tolerance: 0 Position Weight: 1
Orientation Tolerance: 0 Orientation Weight: 1

Scene Browser:
- Robot
  - base_link
  - base
  - base_link_inertia
  - shoulder_link
  - upper_arm_link
  - forearm_link
  - wrist_1_link
  - wrist_2_link
  - wrist_3_link
  - flange
  - tool0
  - gripperBase
  - ee
  - adhesiveStrip
- Scene
  - box
  - toolStationBase
  - partFeeder
  - platform
  - dispensingStation

Constraints Browser:
- ☑ Preset Constraints
  - ☑ Marker Pose Target [Pose]
- ■ User-defined Constraints
  - ☑ Constraint1 [Cartesian]
  - □ Constraint2 [Cartesian]

Scene Inspector — Properties

Name	platform
Type	collisionBox
Box X (m)	1.00
Box Y (m)	1.00
Box Z (m)	2.00e-02

X (m): 0 Y (m): 0 Z (m): -0.011
Euler X (deg): -0
Euler Y (deg): 0
Euler Z (deg): 0

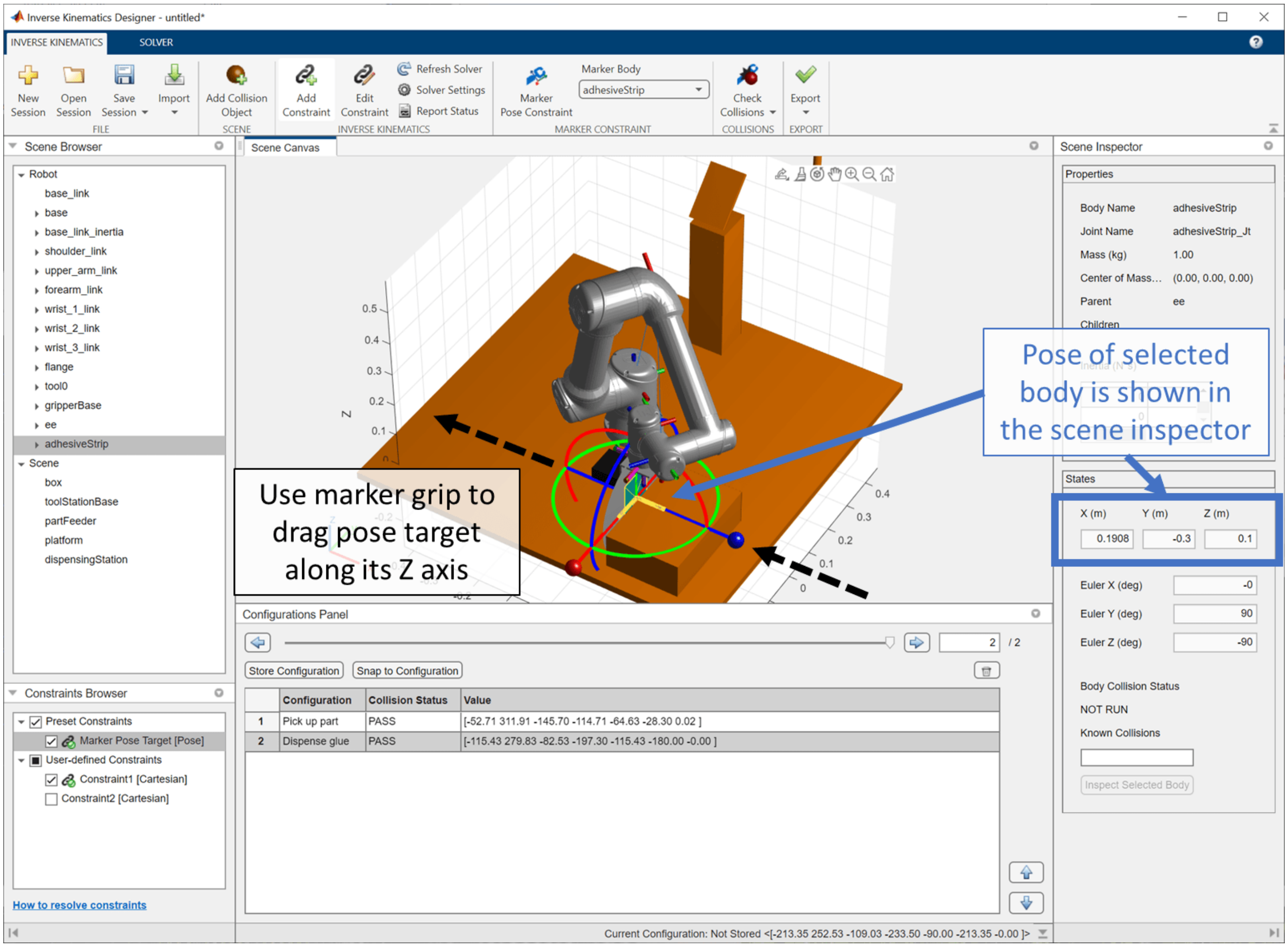
States
Body Collision Status
NOT RUN
Known Collisions

Configuration 2 / 2

	Configuration	Collision Status	Value
1	Pick up part	PASS	[-52.71 311.91 -145.70 -114.71 -64.63 -28.30 0.02]
2	Dispense glue	PASS	[-115.43 279.83 -82.53 -197.30 -115.43 -180.00 -0.00]

Current Configuration: Not Stored <[-115.43 279.83 -82.53 -197.30 -115.43 -180.00 -0.00]>

Apply the constraint, then exit the tab. Check collisions and observe that the robot has reached the configuration, but is still in collision with the box. Again, this is expected due to the positions coinciding. Provide a bit of clearance by dragging the Z axis of the marker to move the gripper. You can see the pose of the selected link in the Scene Inspector whenever the marker is released.



Use marker grip to drag pose target along its Z axis

Pose of selected body is shown in the scene inspector

Body Name	adhesiveStrip
Joint Name	adhesiveStrip_Jt
Mass (kg)	1.00
Center of Mass...	(0.00, 0.00, 0.00)
Parent	ee
Children	

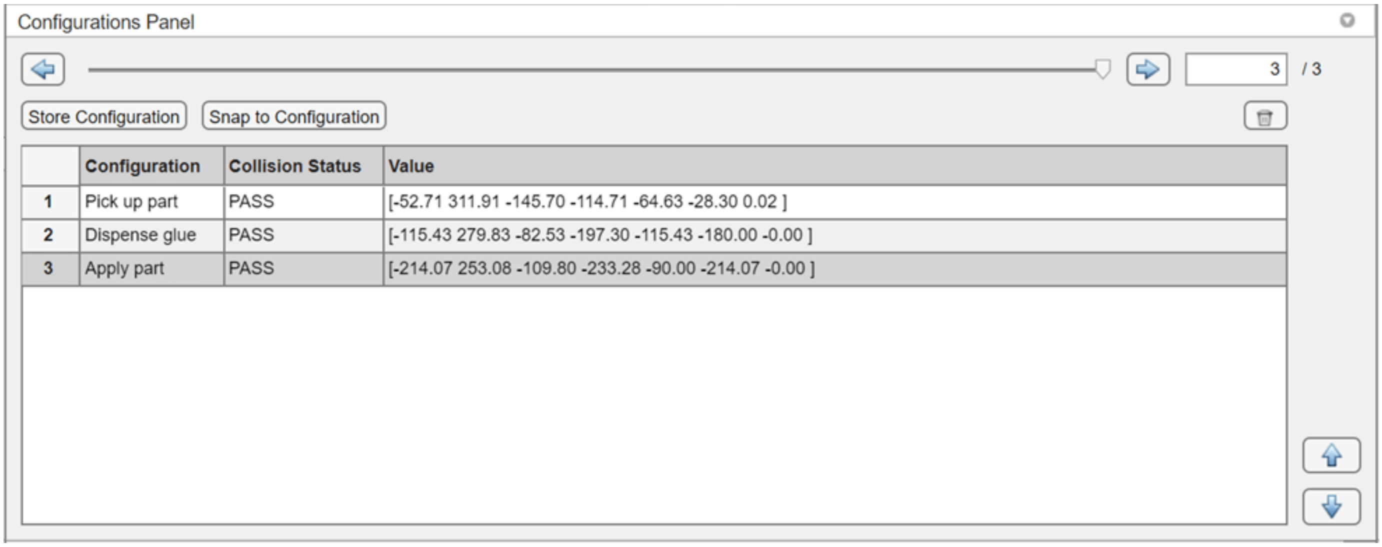
X (m)	Y (m)	Z (m)
0.1908	-0.3	0.1

Euler X (deg)	-0
Euler Y (deg)	90
Euler Z (deg)	-90

	Configuration	Collision Status	Value
1	Pick up part	PASS	[-52.71 311.91 -145.70 -114.71 -64.63 -28.30 0.02]
2	Dispense glue	PASS	[-115.43 279.83 -82.53 -197.30 -115.43 -180.00 -0.00]

Current Configuration: Not Stored <[-213.35 252.53 -109.03 -233.50 -90.00 -213.35 -0.00]>

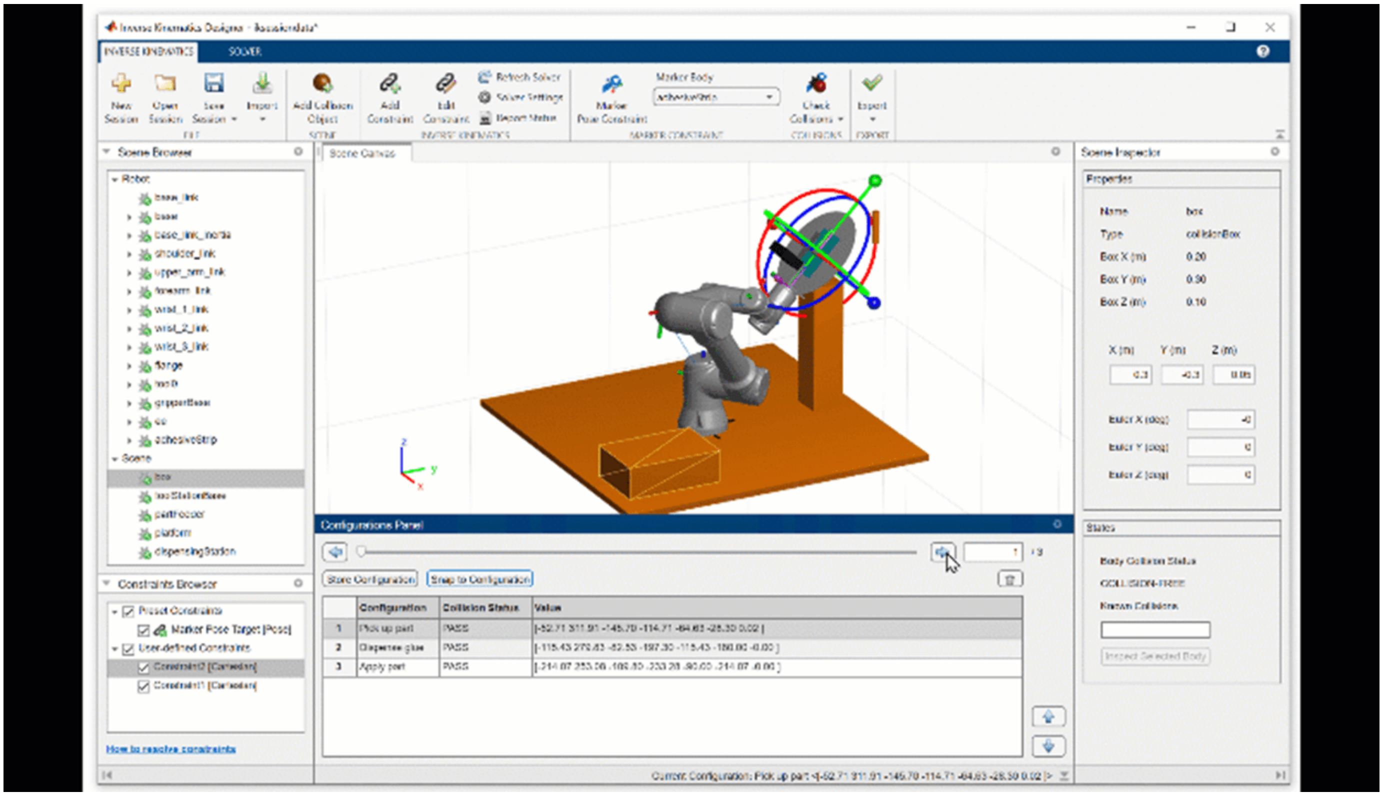
After verifying that the last configuration is collision-free, save it to the **Configurations Browser**.



	Configuration	Collision Status	Value
1	Pick up part	PASS	[-52.71 311.91 -145.70 -114.71 -64.63 -28.30 0.02]
2	Dispense glue	PASS	[-115.43 279.83 -82.53 -197.30 -115.43 -180.00 -0.00]
3	Apply part	PASS	[-214.07 253.08 -109.80 -233.28 -90.00 -214.07 -0.00]

Verify Configurations Work as Waypoints

Check that there are no collisions in any of the configurations and iterate through the waypoints.



	Configuration	Collision Status	Value
1	Pick up part	PASS	[-52.71 311.91 -145.70 -114.71 -84.83 -28.30 0.02]
2	Dispense glue	PASS	[-115.43 272.83 -82.53 -197.30 -115.43 -180.00 -0.00]
3	Apply part	PASS	[-214.07 253.06 -109.80 -233.26 -90.00 -214.07 -0.00]

Name	box
Type	collisionBox
Box X (m)	0.20
Box Y (m)	0.30
Box Z (m)	0.10

X (m)	Y (m)	Z (m)
0.3	-0.3	0.05

Euler X (deg)	-0
Euler Y (deg)	0
Euler Z (deg)	0

Body Collision Status

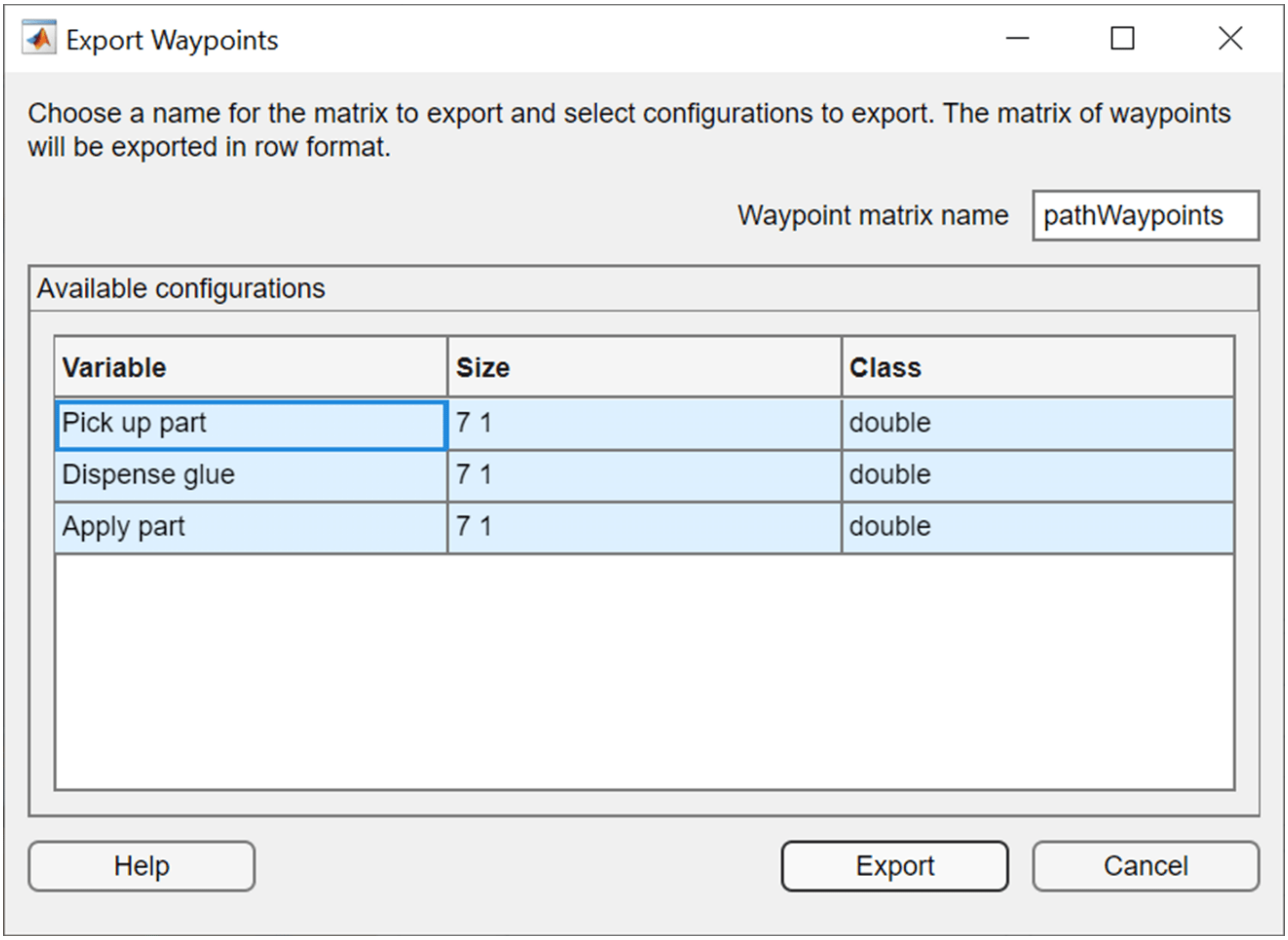
COLLISION-FREE

Known Collisions

Current Configuration: Pick up part <[-52.71 311.91 -145.70 -114.71 -84.83 -28.30 0.02]>

Export Waypoints and Plan Path

Now that all waypoint configurations have been designed, click **Export Path** to export the waypoints to the workspace. Specify the name of the waypoint matrix, shift-click to select all waypoints, and click **Export.**



Choose a name for the matrix to export and select configurations to export. The matrix of waypoints will be exported in row format.

Waypoint matrix name: pathWaypoints

Available configurations

Variable	Size	Class
Pick up part	7 1	double
Dispense glue	7 1	double
Apply part	7 1	double

The set of ordered waypoints now exist as a matrix of configurations in the workspace.

Plan Path Between Waypoints

Since the path between these waypoints is direct and clear of obstacles, a trapezoidal velocity profile trajectory could be used to connect these waypoints with smooth trajectories that stop at each waypoint.

However, because the robot is very close to the collision objects, it is more prudent to use a path planner. Use the `manipulatorRRT` function to plan a path between the three waypoints created in **Inverse Kinematics Designer**.

```
load pathWaypointData.mat
env = {dispensingStation partFeeder toolStationBase platform box};
planner = manipulatorRRT(ur5e_robot,env);
planner.MaxConnectionDistance = 0.45;
planner.ValidationDistance = 0.1;

rng(10);
numPts = 25;
numWaypoints = size(pathWaypoints,1);
paths = cell(1,numWaypoints);
```

```
for segIdx = 1:numWaypoints
    tic;
    plannedPath = plan(planner, pathWaypoints(segIdx,:), pathWaypoints(mod(segIdx,numWaypoints)+1
    shortenedPath = shorten(planner, plannedPath, 10);
    paths{segIdx} = interpolate(planner, shortenedPath, 10);

    segTime = toc;
    disp(['Done planning for segment ',num2str(segIdx),' in ',num2str(segTime), ' seconds']) %i i
end
```

```
Done planning for segment 1 in 29.4579 seconds
Done planning for segment 2 in 76.0054 seconds
Done planning for segment 3 in 19.5043 seconds
```

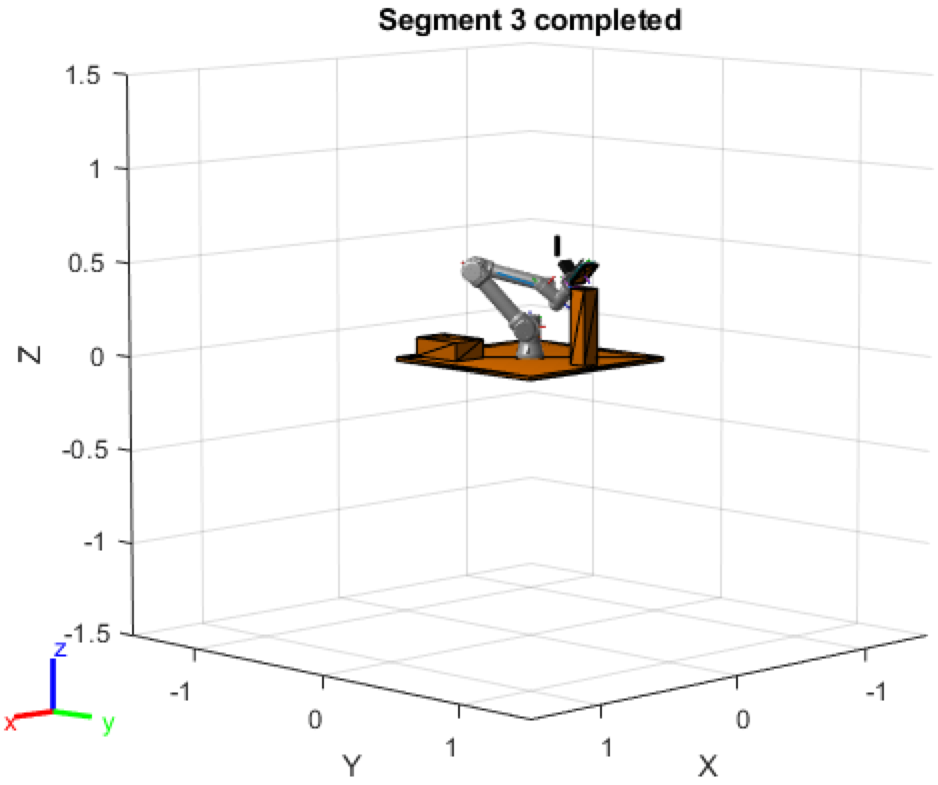
A complete path can be made by combining the segments.

```
totalSegs = vertcat(paths{:});
```

Visualize Completed Trajectory

Now that the waypoints have been planned, play back the path to make sure it works.

```
ax = show(ur5e_robot);
hold all
for i = 1:numel(env)
    env{i}.show("Parent", ax);
end

% Display the figure window the animation
pathFig = ancestor(ax, 'figure');
set(pathFig, "Visible", "on")

% Set up timing and configure robot
r = rateControl(10);
tvec = linspace(1,numWaypoints,numWaypoints*numPts);
ur5e_robot.DataFormat = "row";

% Animate all path segments
for pathSegIdx = 1:numel(paths)
    path = paths{pathSegIdx};

    % For each path segment, step through all the configurations
    for configIdx = 1:size(path,1)
        show(ur5e_robot, path(configIdx,:), "FastUpdate",true, "PreservePlot",false,"Parent",ax)
        waitfor(r);
    end

    % Hold the pose and update the title each time a waypoint is reached
    title(sprintf('Segment %i completed', pathSegIdx), "Parent",ax);
    pause(1);
end
```



Next Steps

Once satisfied with the path, this workflow can be completed by smoothing the paths using trajectories and incorporating the them into a higher-level workflow. For example, these paths may be deployed to a robot using hardware tools, integrating with hardware tools to actuate the gripper. See “Get Started” (Robotics System Toolbox Support Package for Manipulators) for more information.

Create Constrained Inverse Kinematics Solver Using Inverse Kinematics Designer

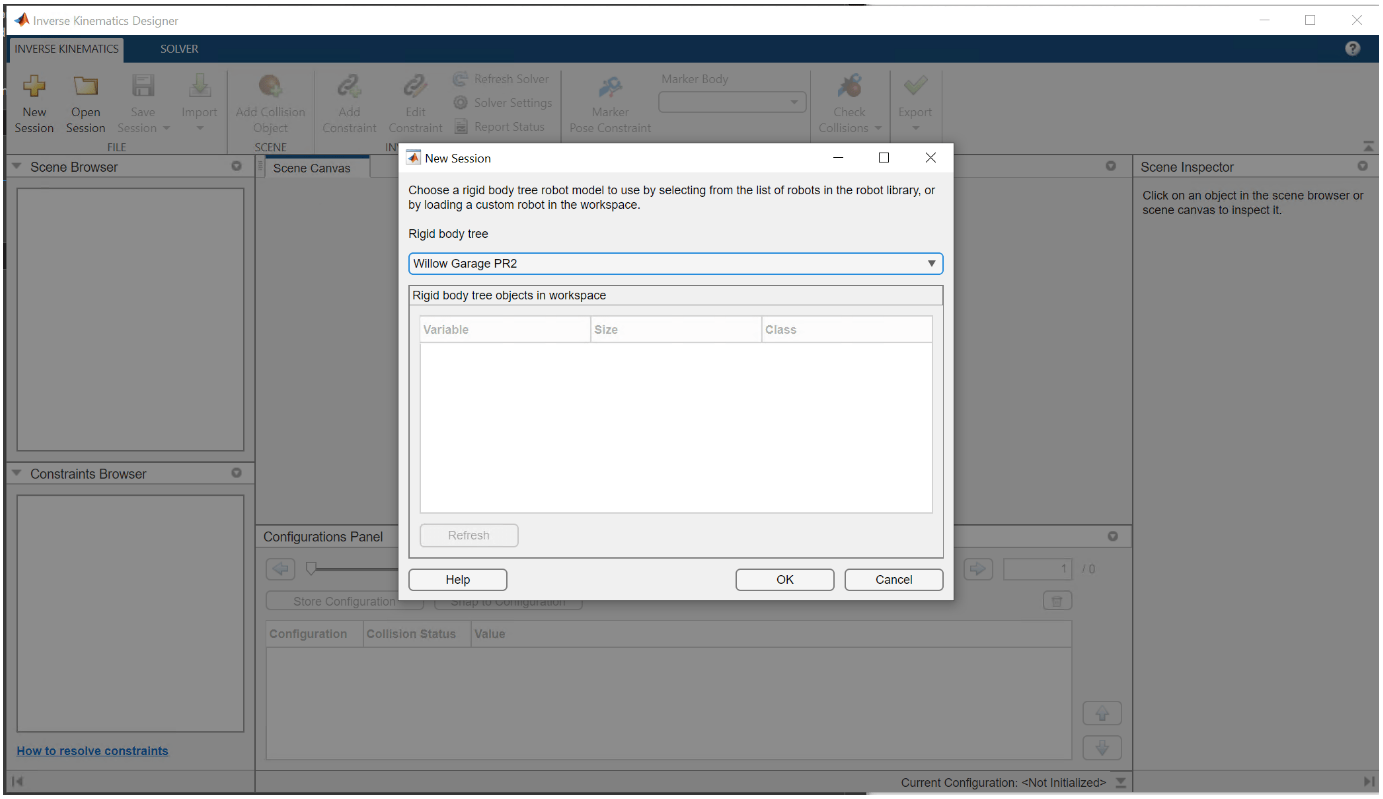
This example shows how to create an inverse kinematics (IK) solver and constraints using the **Inverse Kinematics Designer** app. Inverse kinematics (IK) solvers are used in robotics to determine the joint configurations for a robot that satisfies certain constraints. A constrained IK solver can be used to enforce a variety of behaviors, such as end effector pose targets, joint position limits, and other kinematic constraints. In this example, a solver & constraints are designed to enforce a camera aiming behavior for the Willow Garage PR2. The PR2 includes a body that represents a camera sensor. By adding a constraint that forces the camera to point at the left gripper, the camera follows the gripper as it moves, ensuring that the camera will track any object that the gripper is acting on.

Start New Session

Click **Apps > Inverse Kinematics Designer** to start the app, or use the `inverseKinematicsDesigner` function:

```
inverseKinematicsDesigner
```

Click **New Session** to bring up the **New Session** dialog box to start a new app session. The dialog allows you to choose from a list of robots available in the robot library or provide custom rigid body trees. Custom rigid body trees must be available in the MATLAB workspace before loading them into the app.



New Session

Choose a rigid body tree robot model to use by selecting from the list of robots in the robot library, or by loading a custom robot in the workspace.

Rigid body tree: Willow Garage PR2

Rigid body tree objects in workspace

Variable	Size	Class

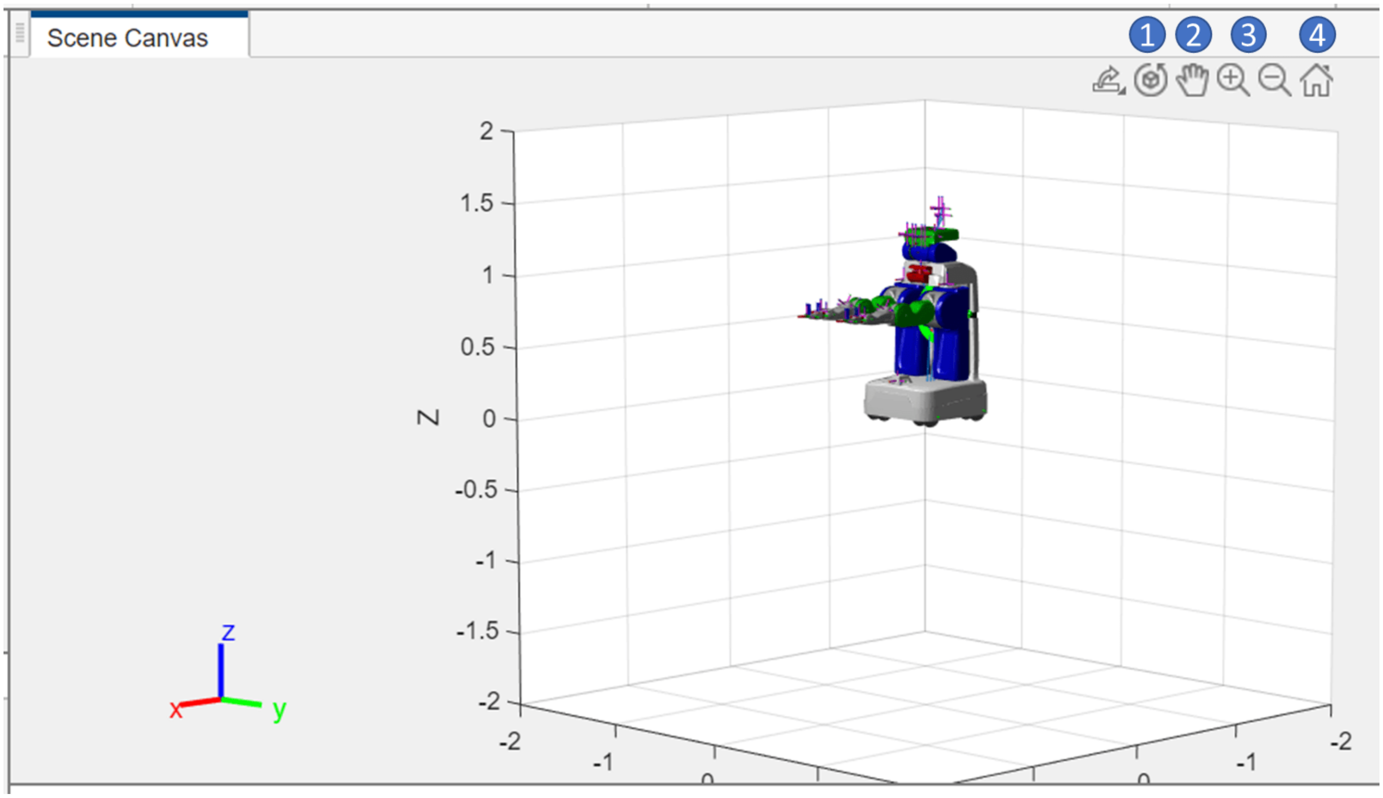
Click on an object in the scene browser or scene canvas to inspect it.

Current Configuration: <Not Initialized>

Select `Willow Garage PR2` from the **Rigid body tree** list, and click **OK**.

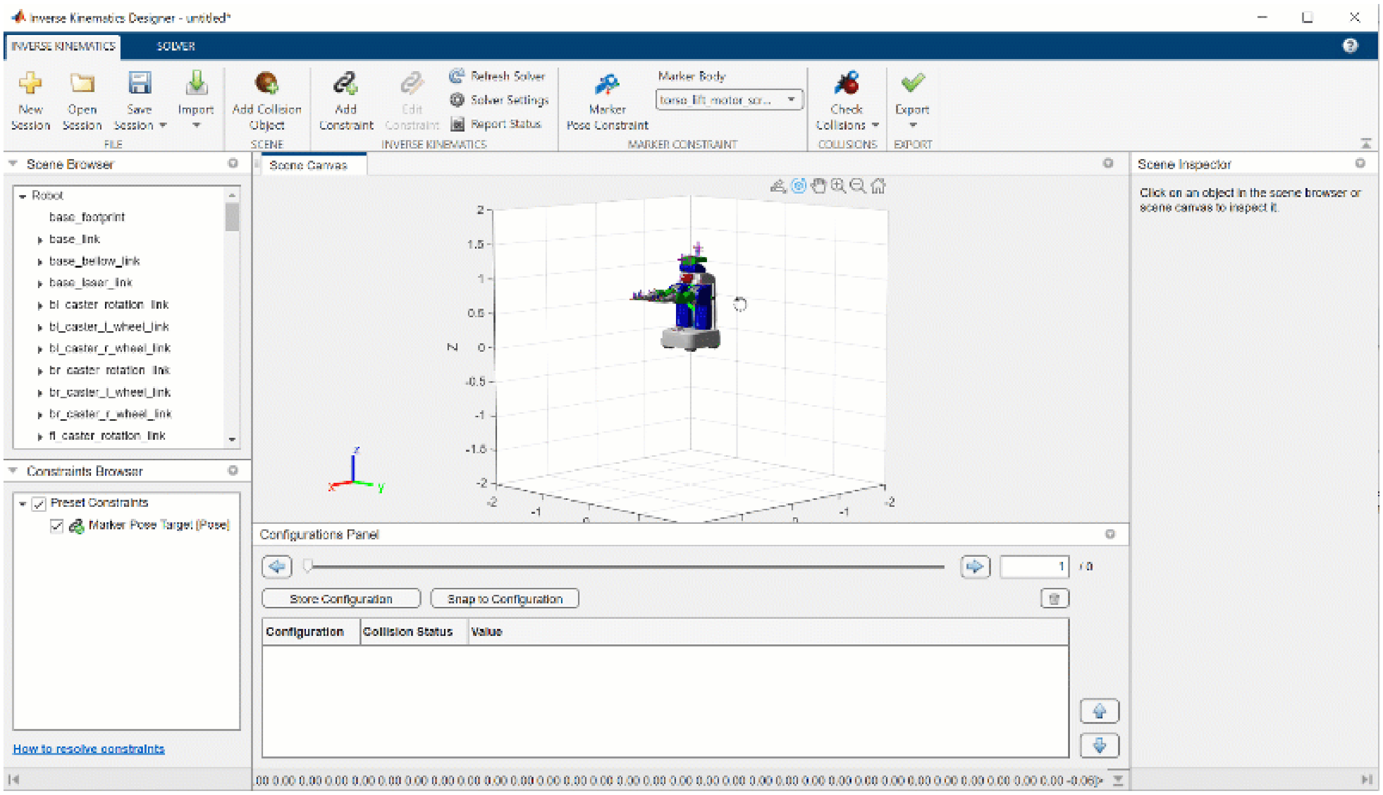
Position Robot in Scene Canvas

Once the app is loaded, use the controls in the **Scene Canvas** to reposition the robot to more comfortably use the space. Hover over the axes shown in the Scene Canvas to bring up the Axes Toolbar, shown with numbered components in the figure below.



Use these buttons to control the figure mode. The **Rotate 3D** control (labeled with a 1 in the figure above) to rotate the robot. In this mode, a scroll wheel or similar function will zoom the entire Scene Canvas, and can be useful to fill the screen. The **Pan** button (labeled with a 2), allows the user to pan the robot within the axes bounds, and has the same default zoom behavior as the Rotate 3D option. The **Zoom In** and **Zoom Out** modes (labled with a 3 above) enable the user to zoom in *within the axes bounds* by clicking or using the scroll wheel zoom. Lastly, the **Home** control resets the view to the default one.

In the following GIF, all four modes are used to reposition the robot into a more usable position given screen space.



Click on an object in the scene browser or scene canvas to inspect it.

Zoom Mode

It is necessary to enter one of the given figure control modes to zoom in using the scroll wheel or a similar feature. As noted above, there are two such modes. When the **Rotate 3D** or **Pan** controls are active, the entire canvas is zoomed (i.e. the axes limits don't change). When the **Zoom In** or **Zoom Out** controls are active, the canvas is zoomed inside the figure bounds. By using these together, it's possible to use the canvas space as efficiently as possible for a task.

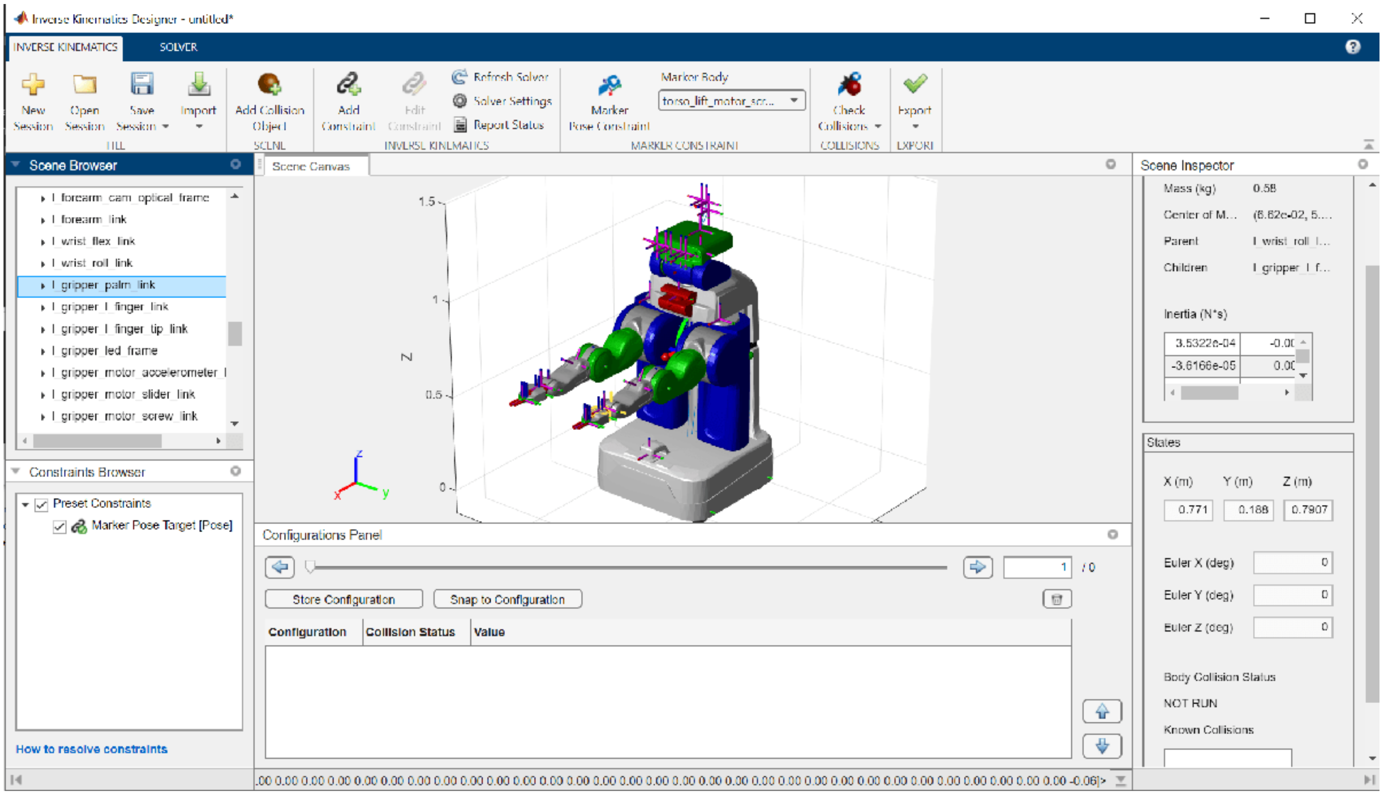
Choose Body for Robot Gripper

Choose the body to use as the robot gripper. There are many possible bodies in the gripper, so it is useful to click on them and select one that moves intuitively. There are several ways to inspect the robot.

Explore Robot Using Scene Panes

The **Scene Browser** is the panel to the left of the scene canvas. It displays all the bodies and joints in the robot, as well as any collision objects that have been loaded into the scene. By clicking on objects in the scene inspector, they will be highlighted in the **Scene Canvas**, and the details of the selected object are brought up in the **Scene Inspector**. The Scene Inspector is the rightmost panel, and shows object properties and states, such as the current pose.

To get started, select the body, `l_gripper_palm_link`, in the **Scene Browser** to select the palm of the left gripper.



Mass (kg) 0.58
Center of M... (8.82e-02, 5....
Parent l_wrist_roll_l...
Children l_gripper_l_f...

Inertia (N*s)

3.5322e-04	-0.00
-3.6166e-05	0.00

X (m)	Y (m)	Z (m)
0.771	0.188	0.7907

Euler X (deg) 0
Euler Y (deg) 0
Euler Z (deg) 0

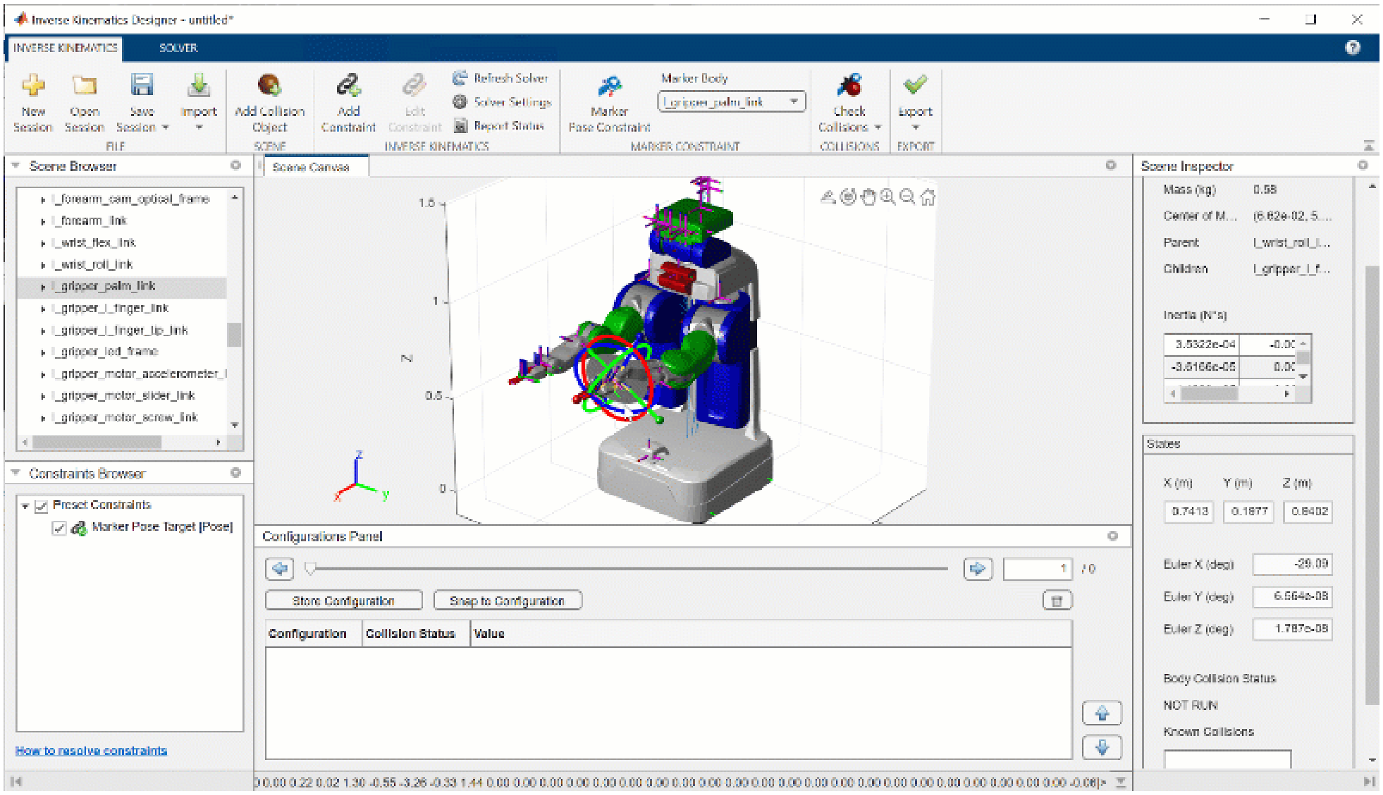
Body Collision Status
NOT RUN

.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 -0.06]>

To further verify its motion, right-click on the body and select **Assign Pose To Marker Body**. This assigns the marker to this body so that it may be moved around using an interactive marker pose target.

Move End Effector Using Marker Pose Constraint

The Inverse Kinematics Designer always creates a solver with one default constraint: the marker pose constraint. This constraint assigns the target pose for a specified end effector body. The body can be set via the right-click menus or in the toolstrip, or by editing the constraint and modifying the constraint details directly. Unlike other constraints, however, the marker pose constraint can be directly set using an interactive marker.



Mass (kg) 0.58

Center of M... (6.62e-02, 5....

Parent l_wrist_roll_l...

Children l_gripper_l_f...

X (m)	Y (m)	Z (m)
0.7413	0.1877	0.8402

Euler X (deg) -29.09

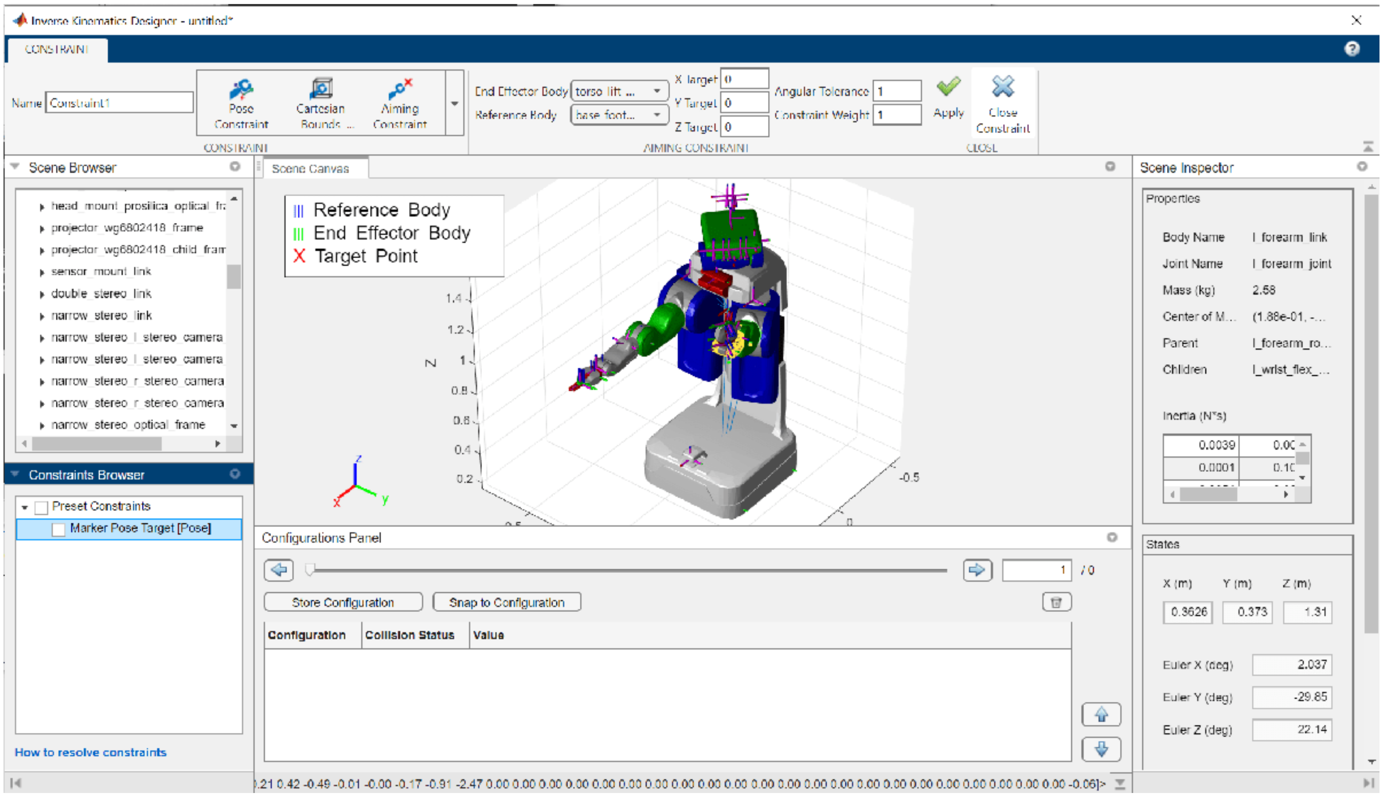
Euler Y (deg) 6.564e-08

Euler Z (deg) 1.787e-08

This tool allows the robot to be easily manipulated, which helps with solver verification. The constraint can also be disabled by unchecking the check box next to it in the **Constraints Browser.**

Point Camera at Gripper Body Using Aiming Constraint

Now that the app has been configured, add an aiming constraint to ensure that the camera is always pointing at the left gripper. An aiming constraint ensures that the Z axis of a designated body points at a given point in space, specified in reference to some other body. To add a constraint, click **Add Constraint** in the toolstrip to open the **Constraint** tab, and then select **Aiming Constraint.** This opens the aiming constraint parameters section of the **Constraint** tab.



Name: Constraint1

End Effector Body: torso_lift ...

Reference Body: base_foot...

X Target: 0

Y Target: 0

Z Target: 0

Angular Tolerance: 1

Constraint Weight: 1

1 / 0

Configuration	Collision Status	Value

Properties

Body Name	l_forearm_link
Joint Name	l_forearm_joint
Mass (kg)	2.58
Center of M...	(1.88e-01, -...
Parent	l_forearm_ro...
Children	l_wrist_flex_...

Inertia (N*s)

0.0039	0.00
0.0001	0.10

States

X (m)	Y (m)	Z (m)
0.3626	0.373	1.31

Euler X (deg): 2.037

Euler Y (deg): -29.85

Euler Z (deg): 22.14

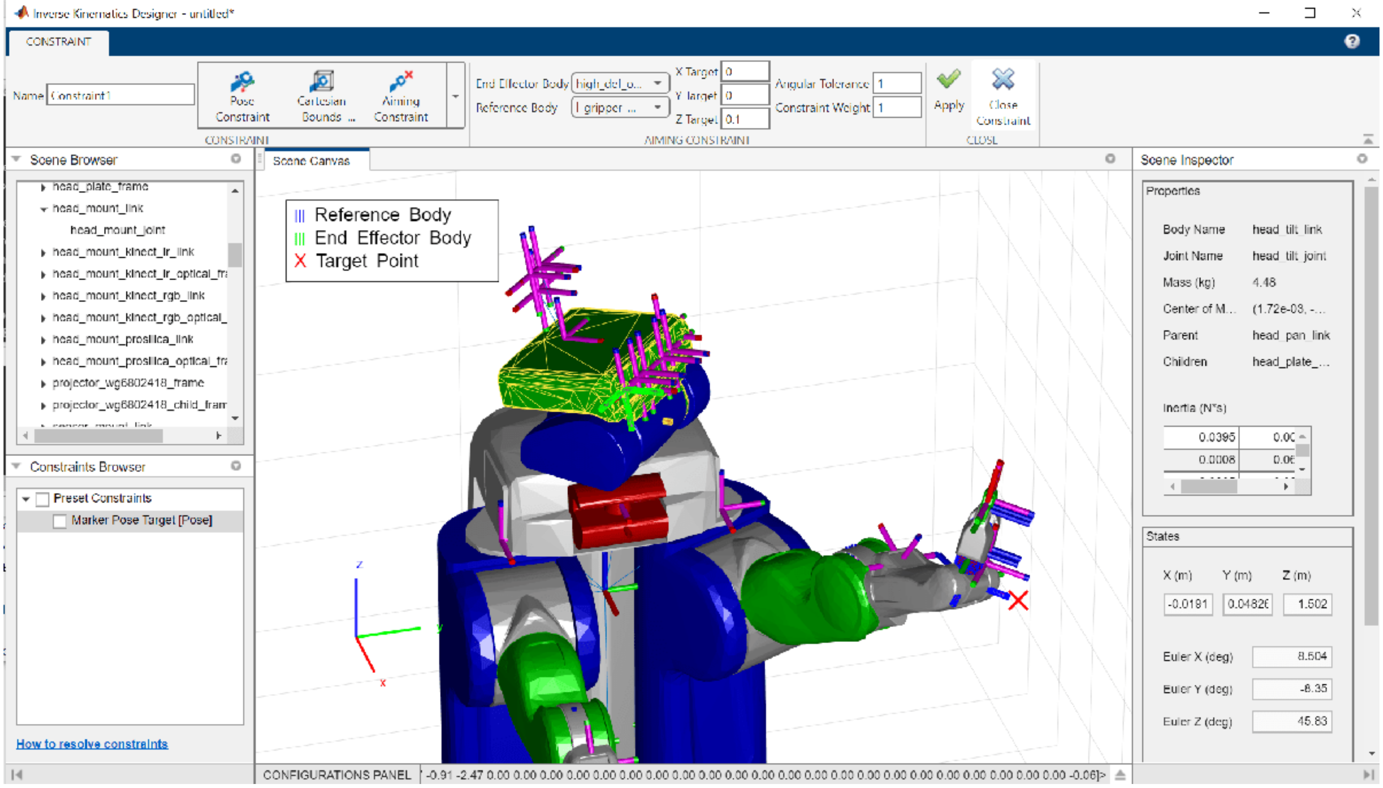
```
.21 0.42 -0.49 -0.01 -0.00 -0.17 -0.91 -2.47 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 -0.06]>
```

Assign Constraint Parameters

To configure the aiming constraint, three main parameters must be set:

- Assign an **End Effector** body. The end effector is the body that aims at the reference point along its Z axis. Since the goal is to have the camera aim, it is necessary to choose a body on the head (the green body) with a forward-facing Z axis is chosen. Select `high_def_optical_frame` from the **End Effector Body** list, and observe that the corresponding body is now highlighted in green.
- Assign a **Reference Body**. The target point is defined with respect to a reference body. Since we have been using the palm to move the robot arm, select `l_gripper_palm_link` from the **Reference Body** list. Observe that the palm body is now highlighted in blue.
- Modify the **Target Point.** By default, the target point is at the origin of the reference body. However, suppose we want to instead aim at a point that is slightly offset from the palm, as though it were holding something. Set the **Z Target** to `0.1` and observe that the red X that indicates the target point is now offset along the reference body's Z-axis.

During constraint modification, the marker can be hidden to make the constraint changes easier to inspect. Disable the marker either by unchecking **Marker Pose Target** in the **Constraints Browser**, or by right-clicking on the marker pose constraint in the Constraints Browser and selecting **Toggle Marker Display**.



Name: Constraint1

End Effector Body: high_def_o...

Reference Body: l_gripper ...

X Target: 0

Y Target: 0

Z Target: 0.1

Angular Tolerance: 1

Constraint Weight: 1

Body Name	head_tilt_link
Joint Name	head_tilt_joint
Mass (kg)	4.48
Center of M...	(1.72e-08, -...
Parent	head_pan_link
Children	head_plate_...

Inertia (N*s)

0.0395	0.00
0.0008	0.06

X (m)	Y (m)	Z (m)
-0.0181	0.04826	1.502

Euler X (deg): 8.604

Euler Y (deg): -8.35

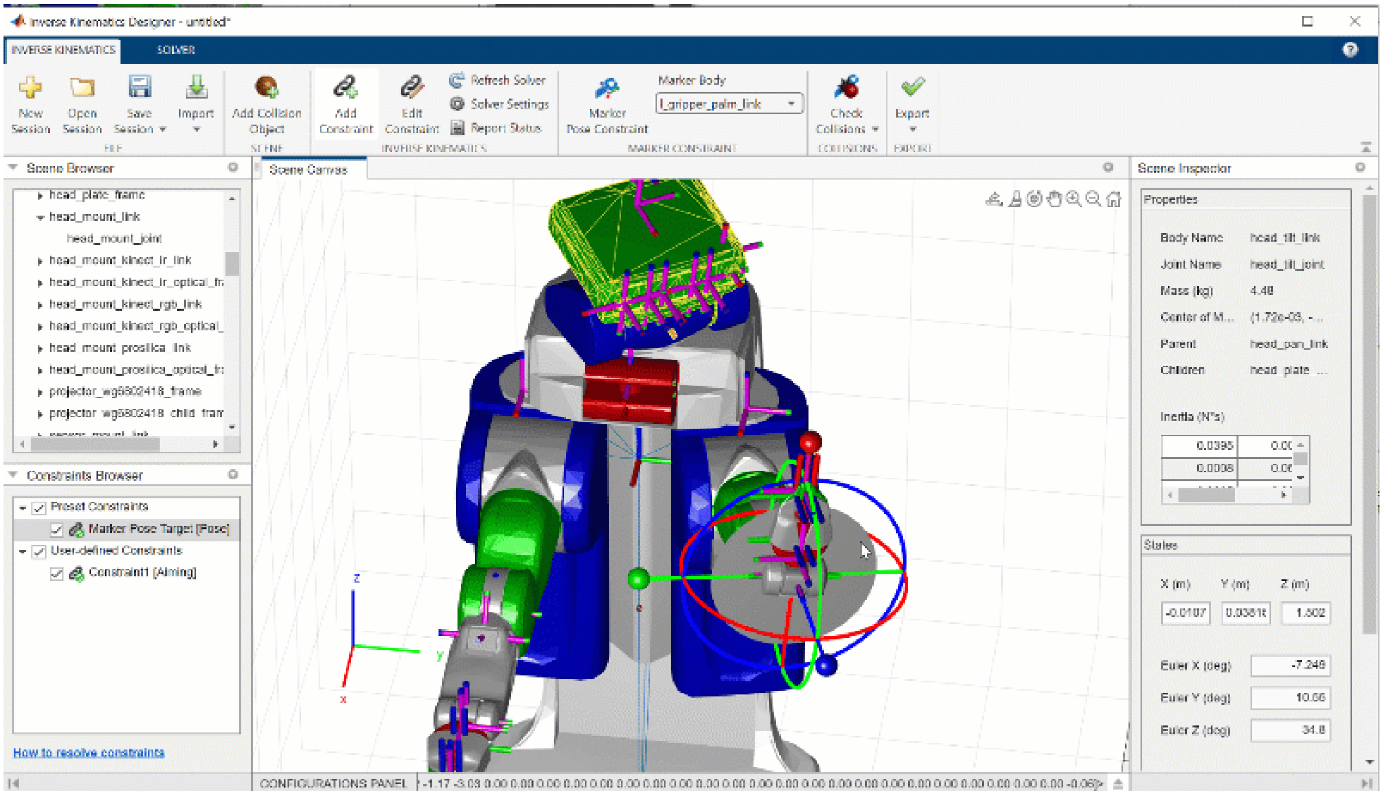
Euler Z (deg): 45.83

CONFIGURATIONS PANEL [-0.91 -2.47 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 -0.06]>

Once you are satisfied with the constraint, click **Apply** to add the constraint to the solver, and **Close Constraint** to exit the constraint tab.

Verify Constrained Behavior

Once the constraint has been applied, verify that it works as expected by moving the palm around and seeing if the camera follows it. To do this, first make sure that both constraints are enabled in the **Constraints Browser** (they should be checked). Next, use the marker to move the palm around, and see if the head follows.



Scene Browser:

- head_plate_frame
- head_mount_link
  - head_mount_joint
- head_mount_kinect_ir_link
- head_mount_kinect_ir_optical_fr
- head_mount_kinect_rgb_link
- head_mount_kinect_rgb_optical_
- head_mount_prosilica_link
- head_mount_prosilica_optical_fr
- projector_wg6802418_frame
- projector_wg6802418_child_fram

Constraints Browser:

- ☑ Preset Constraints
  - ☑ Marker Pose Target [Pose]
- ☑ User-defined Constraints
  - ☑ Constraint1 [Aiming]

Properties

Body Name	head_tilt_link
Joint Name	head_tilt_joint
Mass (kg)	4.48
Center of M...	(1.72e-03, -...
Parent	head_pan_link
Children	head_plate_...

Inertia (N*s)

0.0395	0.0(
0.0008	0.0(

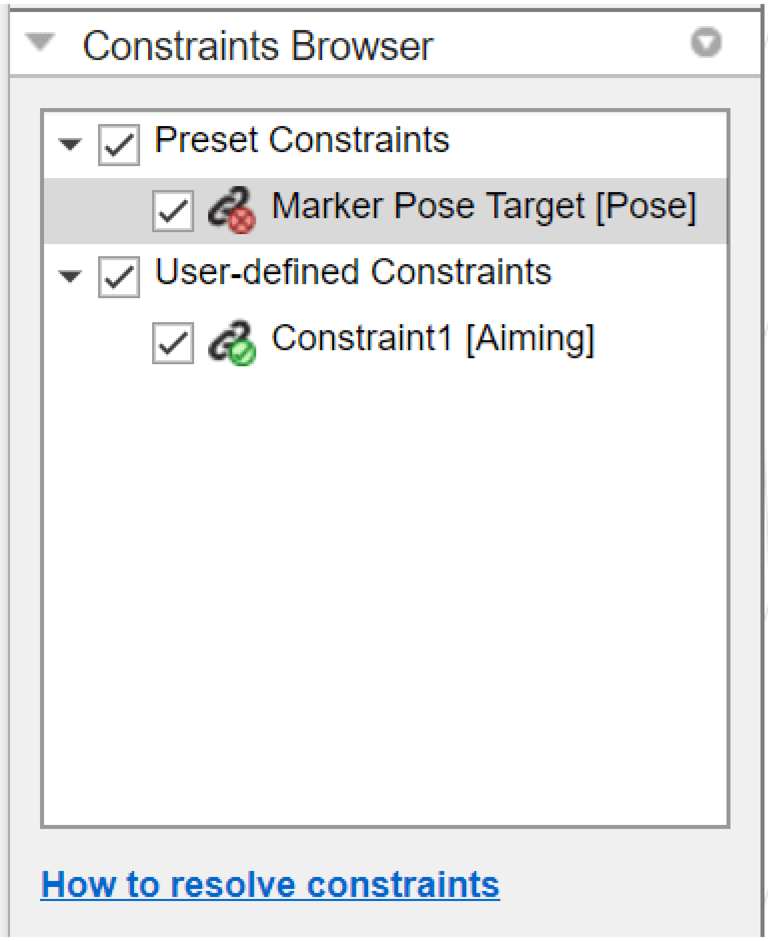
States

X (m)	Y (m)	Z (m)
-0.0107	0.0361t	1.502

Euler X (deg)	-7.249
Euler Y (deg)	10.66
Euler Z (deg)	34.8

CONFIGURATIONS PANEL -1.17 -3.03 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 -0.06

As you move the marker, notice that the robot may not always hit the marker exactly. You can see whether a constraint has been met by looking at the **Constraints Browser**. When the constraint icon with a green check is shown, the constraint has been fully satisfied. However, when a constraint icon with a red X is shown, the constraint has not been met. For example, in the following snapshot, the **Marker Pose Target** is not met, while the aiming constraint has been met.



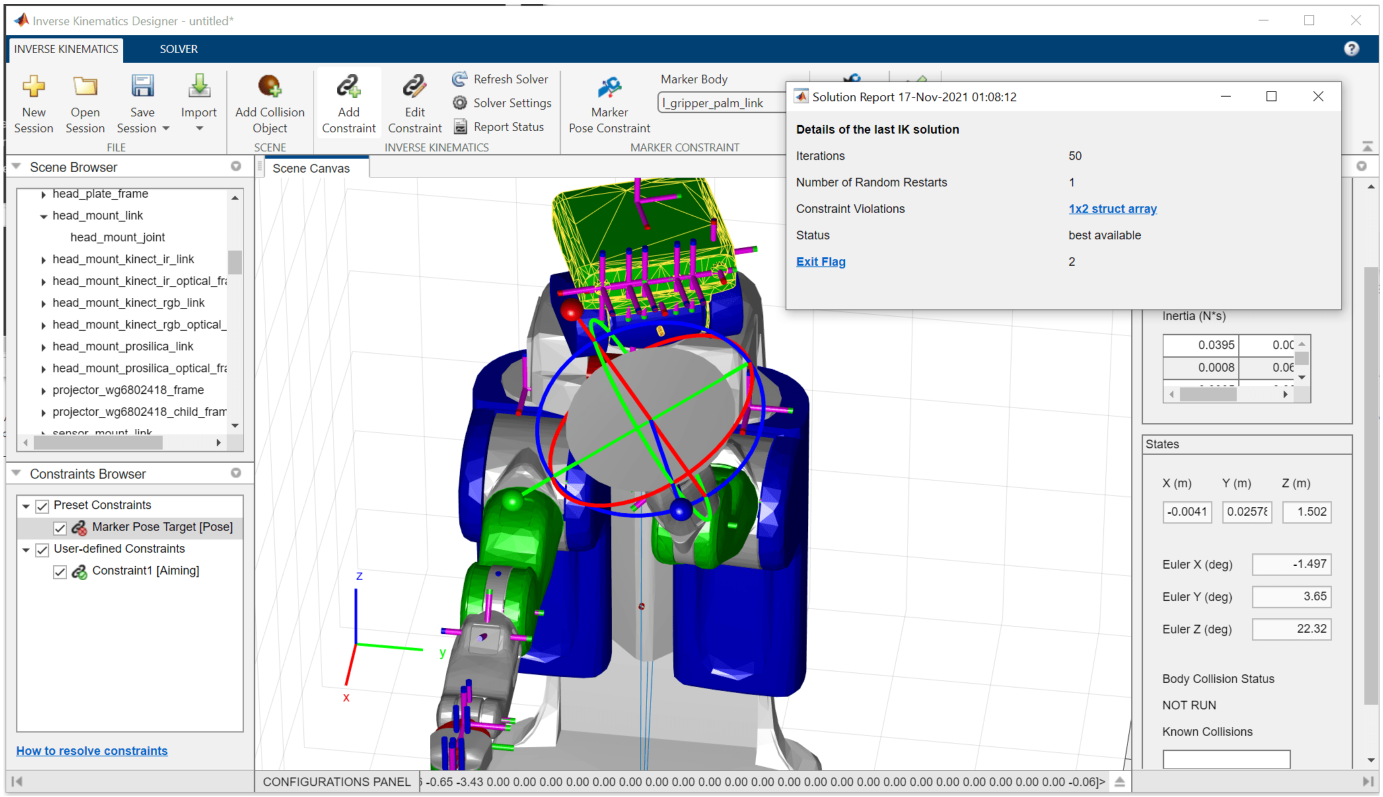
Constraints Browser

- ☑ Preset Constraints
  - ☑ Marker Pose Target [Pose]
- ☑ User-defined Constraints
  - ☑ Constraint1 [Aiming]

How to resolve constraints

There are many reasons why a constraint might not be met, ranging from conflicts with other constraints to not enough solver iterations. Click the **Report Status** button to see why the solver exited. This opens a window that indicates details of the last solver call.

For example, in the situation above, clicking on the report indicates that the solver ran to the maximum number of iterations and then returned the best available solution.



Solution Report 17-Nov-2021 01:08:12

Details of the last IK solution

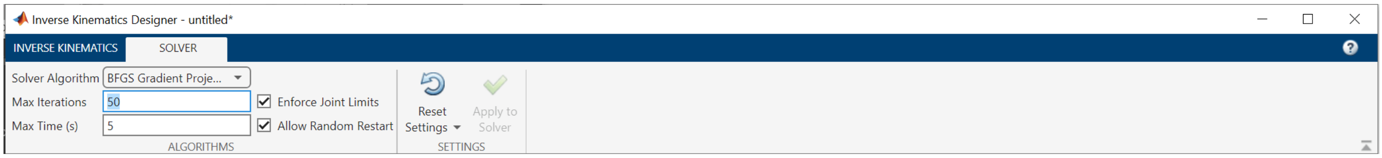
Iterations	50
Number of Random Restarts	1
Constraint Violations	1x2 struct array
Status	best available
Exit Flag	2

Therefore, it is recommended to call the solver again using the current conditions as the initial conditions. If that does not succeed, it is also possible that the pose is unachievable. For more information, refer to “Resolving Constraint Conflict”**.**

Tune Solver Settings

Based on the exit conditions, it may be helpful to modify the solver settings. For example, as more constraints are added, the maximum iterations should be increased to ensure that the solver has time to converge.

To update the settings, click **Solver Settings**, or select the **Solver** tab.



Solver Algorithm: BFGS Gradient Proje...

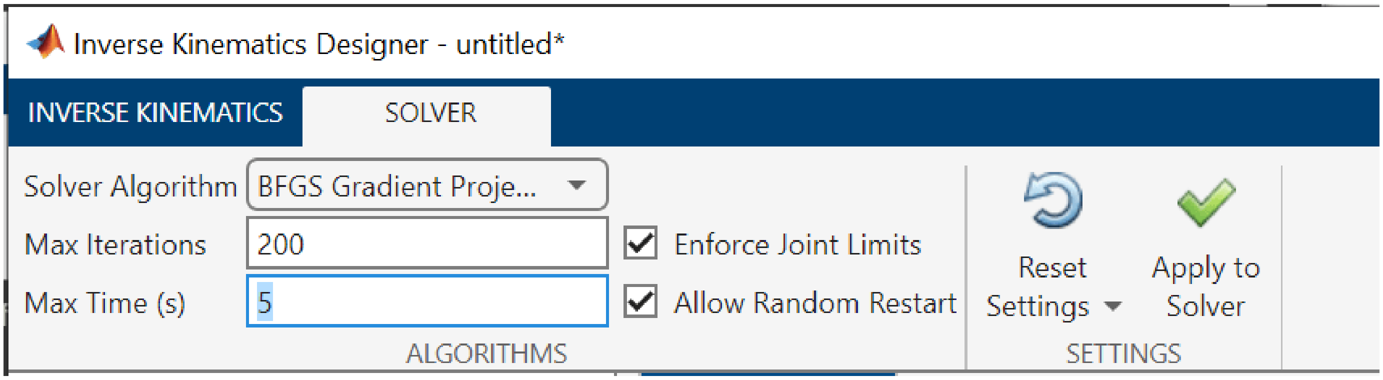
Max Iterations: 50

Max Time (s): 5

☑ Enforce Joint Limits

☑ Allow Random Restart

Select the **Maximum Iterations**, and change the value to `200`. Click outside the box to confirm the change, then click **Apply to Solver** to apply changes.



Solver Algorithm: BFGS Gradient Proje...

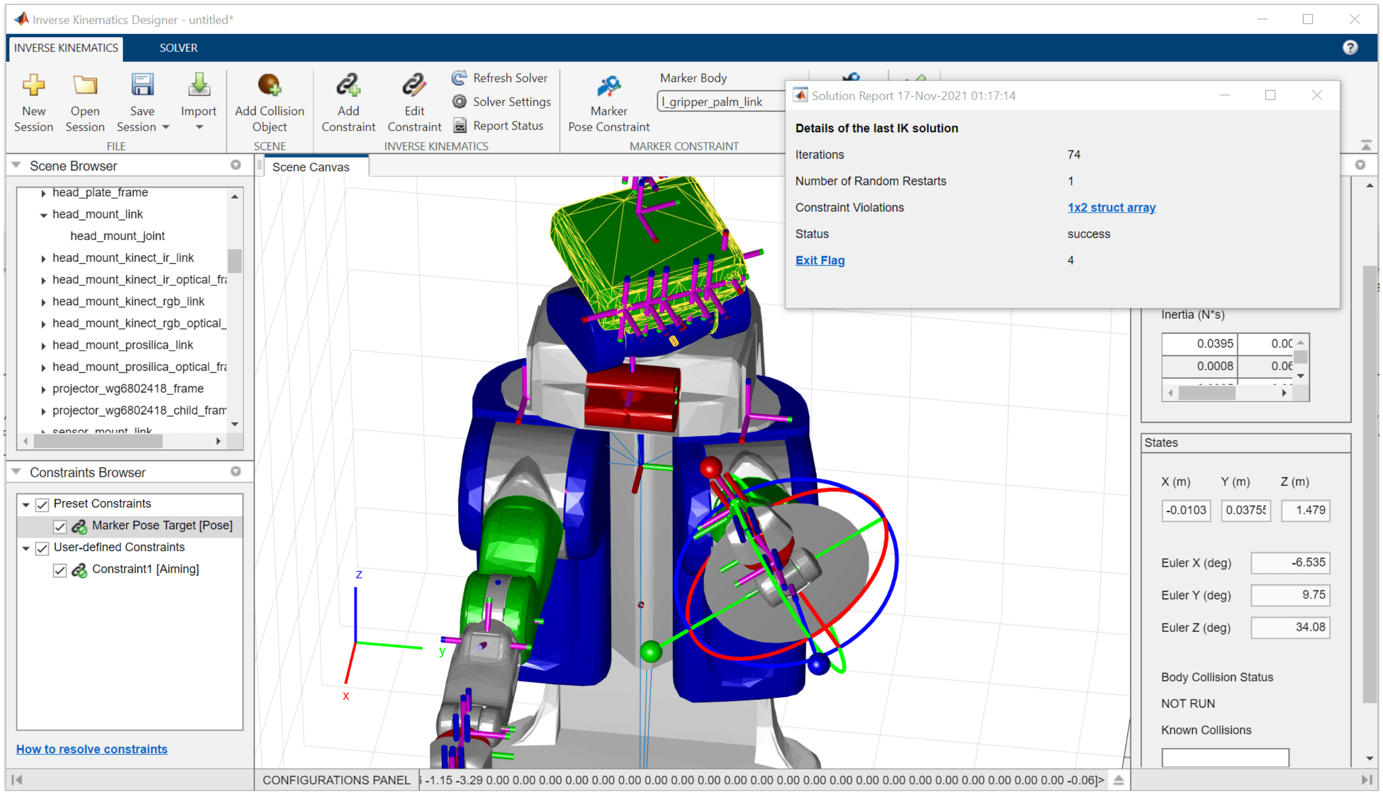
Max Iterations: 200

Max Time (s): 5

☑ Enforce Joint Limits

☑ Allow Random Restart

Once the new solver settings have been applied, return to the **Inverse Kinematics** tab and click **Refresh Solver** to run the solver again. Click **Report Status** to verify that when a solution is achievable, the solution converges before the maximum iterations are hit.



Solution Report 17-Nov-2021 01:17:14

Details of the last IK solution

Iterations	74
Number of Random Restarts	1
Constraint Violations	1x2 struct array
Status	success
Exit Flag	4

X (m)	Y (m)	Z (m)
-0.0103	0.03755	1.479

Euler X (deg) -6.535

Euler Y (deg) 9.75

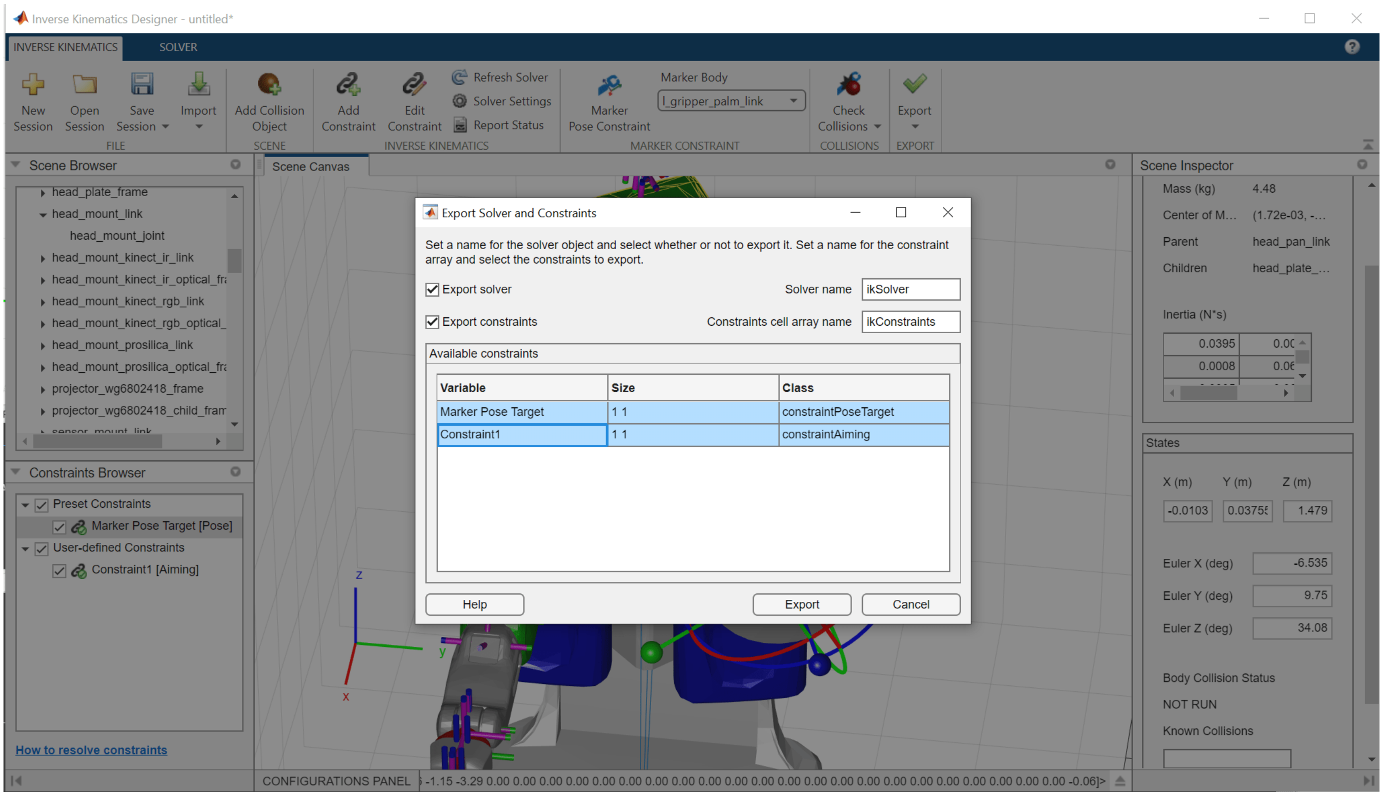
Euler Z (deg) 34.08

[-1.15 -3.29 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 -0.06]

Export Solver and Constraints

To use the solver object outside of the app, export it to the MATLAB® workspace. Click **Export Solver and Constraints**, which will bring up the **Export Solver and Constraints** dialog.

Set **Solver name** and **Constraints cell array name** or use the default values. Shift-click to select multiple constraints and click **Export**.



Export Solver and Constraints

Set a name for the solver object and select whether or not to export it. Set a name for the constraint array and select the constraints to export.

☑ Export solver Solver name ikSolver

☑ Export constraints Constraints cell array name ikConstraints

Available constraints

Variable	Size	Class
Marker Pose Target	1 1	constraintPoseTarget
Constraint1	1 1	constraintAiming

Help Export Cancel

Mass (kg) 4.48
Center of M... (1.72e-03, -...
Parent head_pan_link
Children head_plate_...
Inertia (N*s)

0.0395	0.00
0.0008	0.06

X (m) -0.0103 Y (m) 0.03755 Z (m) 1.479
Euler X (deg) -6.535
Euler Y (deg) 9.75
Euler Z (deg) 34.08
Body Collision Status NOT RUN

<-1.15 -3.29 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 -0.06]>

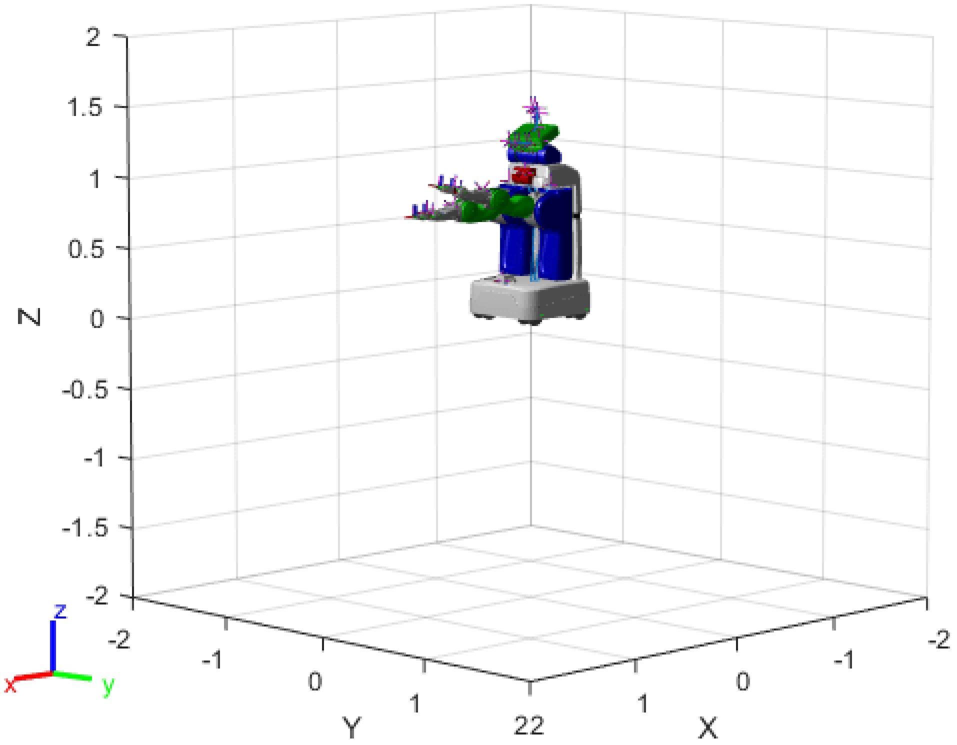
Once everything is selected, click **Export** to export these to the workspace.

Use Exported Solver

Use the exported solver object to constrain another configuration directly from the command line.

Note that the exported solver uses a robot with row format so the exported configurations are in row format. A handle to the rigid body tree is stored on the exported solver object.

```
load("ikSolverDesignExampleData.mat")

robot = ikSolver.RigidBodyTree;

% Assign a new target pose to the gripper
ikConstraints{1}.TargetTransform = trvec2tform([-.01 .04 1.45]);

% Call the solver with these two constraints
[qSol, info] = ikSolver(robot.homeConfiguration, ikConstraints{:});

% Display results
show(robot, qSol);
```


2
1.5
1
0.5
0
-0.5
-1
-1.5
-2
Z
-2
-1
0
1
22
Y
1
0
-1
-2
X
z
x
y

Perform Path Planning Simulation with Mobile Robot

Create a scenario to simulate a mobile robot navigating a room. The example demonstrates how to create a scenario, model a robot platform from a rigid body tree object, obtain a binary occupancy grid map from the scenario, and plan a path for the mobile robot to follow using the `mobileRobotPRM` path planning algorithm.

Create Scenario with Ground Plane and Static Meshes

A `robotScenario` object consists of a set of static obstacles and movable objects called platforms. Use `robotPlatform` object to model the mobile robot within the scenario. This example builds a scenario consisting of a ground plane and box meshes to create a room.

```
scenario = robotScenario(UpdateRate=5);
```

Add a plane mesh as ground plane in the scenario.

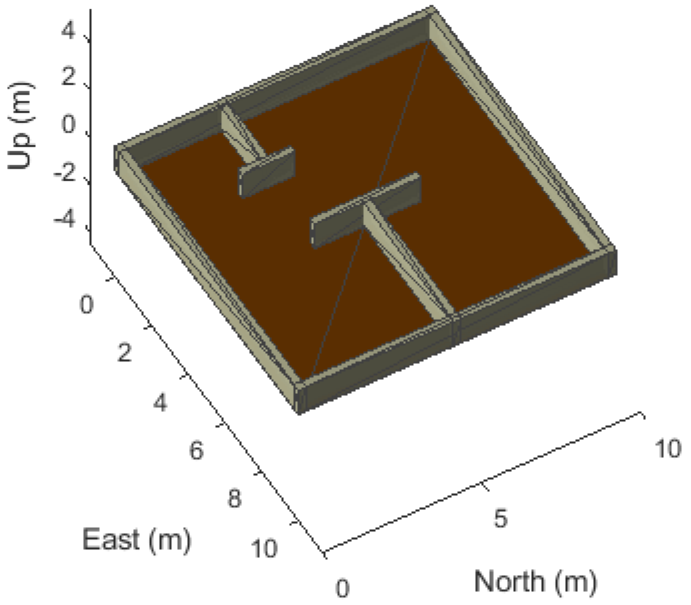
```
floorColor = [0.5882 0.2941 0];
addMesh(scenario,"Plane",Position=[5 5 0],Size=[10 10],Color=floorColor);
```

The walls of the room are modeled as box meshes. The static meshes are added with the `IsBinaryOccupied` value set to `true`, so these obstacles are incorporated into the binary occupancy map used for path planning.

```
wallHeight = 1;
wallWidth = 0.25;
wallLength = 10;
wallColor = [1 1 0.8157];

% Add outer walls.
addMesh(scenario,"Box",Position=[wallWidth/2, wallLength/2, wallHeight/2],...
    Size=[wallWidth, wallLength, wallHeight],Color=wallColor,IsBinaryOccupied=true);
addMesh(scenario,"Box",Position=[wallLength-wallWidth/2, wallLength/2, wallHeight/2],...
    Size=[wallWidth, wallLength, wallHeight],Color=wallColor,IsBinaryOccupied=true);
addMesh(scenario,"Box",Position=[wallLength/2, wallLength-wallWidth/2, wallHeight/2],...
    Size=[wallLength, wallWidth, wallHeight],Color=wallColor,IsBinaryOccupied=true);
addMesh(scenario,"Box",Position=[wallLength/2, wallWidth/2, wallHeight/2],...
    Size=[wallLength, wallWidth, wallHeight],Color=wallColor,IsBinaryOccupied=true);

% Add inner walls.
addMesh(scenario,"Box",Position=[wallLength/8, wallLength/3, wallHeight/2],...
    Size=[wallLength/4, wallWidth, wallHeight],Color=wallColor,IsBinaryOccupied=true);
addMesh(scenario,"Box",Position=[wallLength/4, wallLength/3, wallHeight/2],...
    Size=[wallWidth, wallLength/6,  wallHeight],Color=wallColor,IsBinaryOccupied=true);
addMesh(scenario,"Box",Position=[(wallLength-wallLength/4), wallLength/2, wallHeight/2],...
   Size=[wallLength/2, wallWidth, wallHeight],Color=wallColor,IsBinaryOccupied=true);
addMesh(scenario,"Box",Position=[wallLength/2, wallLength/2, wallHeight/2],...
    Size=[wallWidth, wallLength/3, wallHeight],Color=wallColor,IsBinaryOccupied=true);
```

Visualize the scenario.

```
show3D(scenario);
lightangle(-45,30);
view(60,50);
```



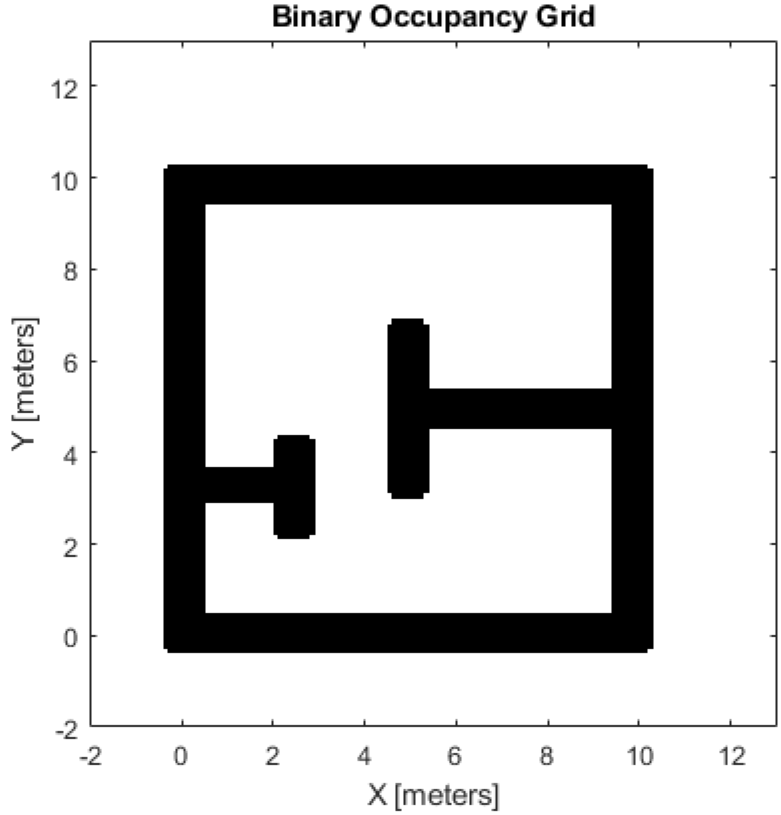
Obtain Binary Occupancy Map from Scenario

Obtain an occupancy map as a binaryOccupancyMap object from the scenario for path planning. Inflate the occupied spaces on the map by 0.3m.

```
map = binaryOccupancyMap(scenario,GridOriginInLocal=[-2 -2],...
                                          MapSize=[15 15],...
                                          MapHeightLimits=[0 3]);
inflate(map,0.3);
```

Visualize the 2-D occupancy map.

```
show(map)
```



Path Planning

Use the `mobileRobotPRM` path planner to find an obstacle-free path between the start and goal positions on the obtained map.

Specify the start and goal positions of the mobile robot.

```
startPosition = [1 1];
goalPosition = [8 8];
```

Set the `rng` seed for repeatability.

```
rng(100)
```

Create a `mobileRobotPRM` object with the binary occupancy map and specify the maximum number of nodes. Specify the maximum distance between the two connected nodes.

```
numnodes = 2000;
planner = mobileRobotPRM(map,numnodes);
planner.ConnectionDistance = 1;
```

Find a path between the start and goal positions.

```
waypoints = findpath(planner,startPosition,goalPosition);
```

Trajectory Generation

Generate trajectory for the mobile robot to follow with the waypoints from the planned path using the `waypointTrajectory` System object.

```
% Robot height from base.
robotheight = 0.12;
% Number of waypoints.
numWaypoints = size(waypoints,1);
% Robot arrival time at first waypoint.
firstInTime = 0;
% Robot arrival time at last waypoint.
lastInTime = firstInTime + (numWaypoints-1);
% Generate waypoint trajectory with waypoints from planned path.
traj = waypointTrajectory(SampleRate=10,...
                          TimeOfArrival=firstInTime:lastInTime, ...
                          Waypoints=[waypoints, robotheight*ones(numWaypoints,1)], ...
                          ReferenceFrame="ENU");
```

Add Robot Platform to Scenario

Load the Clearpath Husky mobile robot from the robot library as a `rigidBodyTree` object.

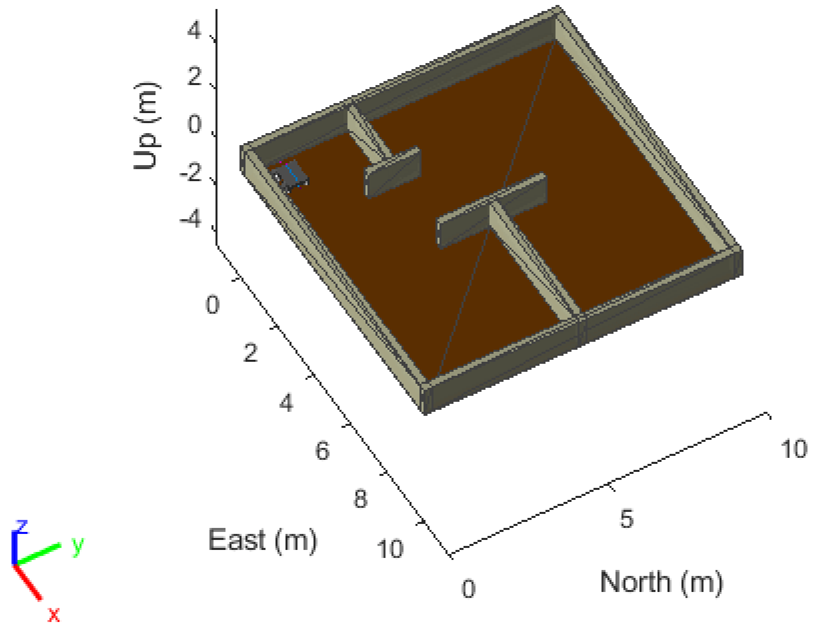
```
huskyRobot = loadrobot("clearpathHusky");
```

Create a `robotPlatform` object with the robot model specified as a `rigidBodyTree` object and its trajectory specified as a `waypointTrajectory` System object.

```
platform = robotPlatform("husky",scenario, RigidBodyTree=huskyRobot,...
                         BaseTrajectory=traj);
```

Visualize the scenario with the robot.

```
[ax,plotFrames] = show3D(scenario);
lightangle(-45,30)
view(60,50)
```



Simulate Mobile Robot

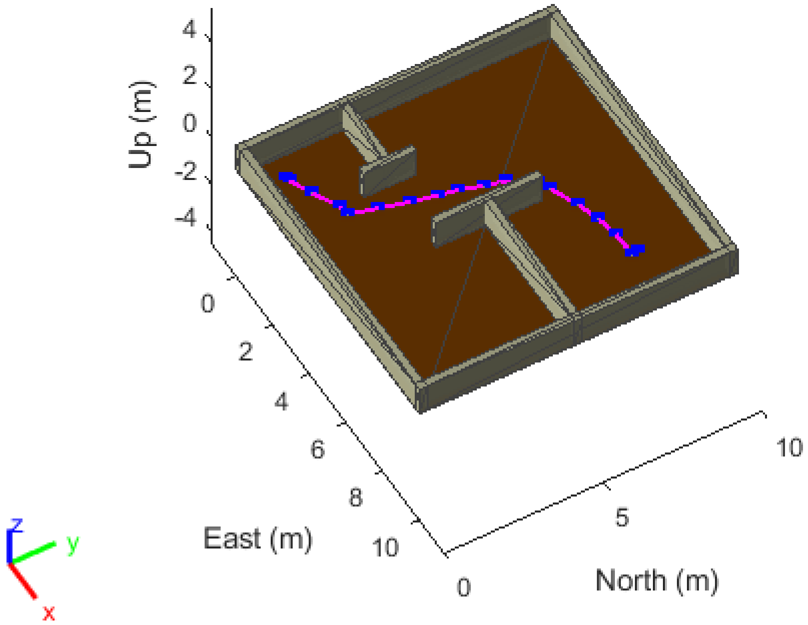
Visualize the planned path.

```
hold(ax,"on")
plot(ax,waypoints(:,1),waypoints(:,2),"-ms",...
               LineWidth=2,...
               MarkerSize=4,...
               MarkerEdgeColor="b",...
               MarkerFaceColor=[0.5 0.5 0.5]);
hold(ax,"off")
```

Set up the simulation. Since all the poses of the robot are known in advance, simply step through the simulation and update the visualization at each step.

```
setup(scenario)

% Control simulation rate at 20 Hz.
r = rateControl(20);

% Status of robot in simulation.
robotStartMoving = false;

while advance(scenario)
    show3D(scenario,Parent=ax,FastUpdate=true);
    waitfor(r);

    currentPose = read(platform);
```

```
        if ~any(isnan(currentPose))
            % implies that robot is in the scene and performing simulation.
            robotStartMoving = true;
        end
        if any(isnan(currentPose)) && robotStartMoving
            % break, once robot reaches goal position.
            break;
        end
    end
```



To re-run the simulation and visualize the results again, reset the simulation in the scenario.

```
restart(scenario)
```

Perform Obstacle Avoidance in Warehouse Scenario with Mobile Robots

Create a scenario to simulate two mobile robots performing obstacle avoidance in a warehouse. This example demonstrates how to create a warehouse scenario, add mobile robots using the rigid body tree representation, model the kinematics of the robots, and simulate the behavior of the control algorithms using the resultant scenario.

Create Warehouse Scenario

A `robotScenario` object consists of static meshes and `robotPlatform` objects. The `robotPlatform` objects can be static or movable. The `robotPlatform` object supports robot model specified as `rigidBodyTree` object, which enables SDF and URDF model support. In this example, the warehouse scenario can be created with static box meshes or with SDF models.

```
scenario = robotScenario(UpdateRate=10);
```

Add a plane mesh as ground plane in the scenario.

```
addMesh(scenario,"Plane",Position=[5 0 0],Size=[20 12],Color=[0.7 0.7 0.7]);
```

Create Warehouse Scenario Using Static Meshes

By default, `scenarioOptions` is set to `Cuboid`, here the warehouse scenario is constructed using static cuboid meshes. The cuboid meshes provides low fidelity simulation environment, which helps in testing algorithms with basic scenario elements.

```
% Select warehouse scenario options.
scenarioOptions = Cuboid ▾;
```

In the warehouse scenario, the left and right side box meshes are considered as stationary shelves. So these areas are treated as restricted region for the robots and considered in the occupancy map, with `IsBinaryOccupied` parameter set to `true`.

```
if strcmp(scenarioOptions,"Cuboid")

    addMesh(scenario,"Box",Position=[3  5 2],Size=[4 2 4],Color=[1 0.5 0.25],IsBinaryOccupied=tru
    addMesh(scenario,"Box",Position=[3 -5 2],Size=[4 2 4],Color=[1 0.5 0.25],IsBinaryOccupied=tru
    addMesh(scenario,"Box",Position=[7  5 2],Size=[4 2 4],Color=[1 0.5 0.25],IsBinaryOccupied=tru
    addMesh(scenario,"Box",Position=[7 -5 2],Size=[4 2 4],Color=[1 0.5 0.25],IsBinaryOccupied=tru
```

The robot loads and unloads objects from the loading and unloading positions. These objects are non-stationary and represented with static cuboid meshes. So these meshes are not considered in the occupancy map.

```
    addMesh(scenario,"Box",Position=[-3  0 0.5],Size=[1 1 1],Color=[0.1 0.1 0.1]);
    addMesh(scenario,"Box",Position=[-5  1 0.5],Size=[1 1 1],Color=[0.1 0.1 0.1]);
    addMesh(scenario,"Box",Position=[-5 -1 0.5],Size=[1 1 1],Color=[0.1 0.1 0.1]);
    addMesh(scenario,"Box",Position=[13  0 0.5],Size=[1 1 1],Color=[0.1 0.1 0.1]);
    addMesh(scenario,"Box",Position=[15  1 0.5],Size=[1 1 1],Color=[0.1 0.1 0.1]);
    addMesh(scenario,"Box",Position=[15 -1 0.5],Size=[1 1 1],Color=[0.1 0.1 0.1]);
```

The warehouse scenario also contains an unattended non-stationary object which is represented with static cuboid mesh. The robot treats this object as obstacle and avoids if found between planned paths. This scenario element is not considered in the occupancy map.

```
    addMesh(scenario,"Box",Position=[5 0 1.5],Size=[2 2 3],Color=[1 0 0]);
end
```

Create Warehouse Scenario Using SDF Models

To get more realistic warehouse scenario, download robomaker warehouse models and set `scenarioOptions` to `SDF`. These SDF models are added as static `robotPlatform` objects to create the warehouse scenario. The SDF models provides high fidelity simulation environment.

Load the shelves, cluttering, and bucket SDF models as `rigidBodyTree` object.

```
if strcmp(scenarioOptions,"SDF")

    shelfARBT = importrobot(fullfile(...
                    "roboMaker","models","aws_robomaker_warehouse_ShelfD_01","model.sdf"));
    shelfBRBT = importrobot(fullfile(...
                    "roboMaker","models","aws_robomaker_warehouse_ShelfE_01","model.sdf"));
    clutteringRBT = importrobot(fullfile(...
                    "roboMaker","models","aws_robomaker_warehouse_ClutteringA_01","model.sdf"));
    bucketWareRBT = importrobot(fullfile(...
                    "roboMaker","models","aws_robomaker_warehouse_Bucket_01","model.sdf"));
```

The SDF models which represents shelves on left and right side of the warehouse, are considered as stationary. So these areas are treated as restricted region for the robots and considered in the occupancy map, with `IsBinaryOccupied` parameter set to `true`.

```
    shelfAModel = robotPlatform("ShelfA",scenario,RigidBodyTree=shelfARBT,...
                        InitialBasePosition=[3  4 0],IsBinaryOccupied=true);
    shelfCModel = robotPlatform("ShelfC",scenario,RigidBodyTree=shelfARBT,...
                        InitialBasePosition=[3 -4 0],IsBinaryOccupied=true);
    shelfBModel = robotPlatform("ShelfB",scenario,RigidBodyTree=shelfBRBT,...
                        InitialBasePosition=[7  4 0],IsBinaryOccupied=true);
    shelfDModel = robotPlatform("ShelfD",scenario,RigidBodyTree=shelfBRBT,...
                        InitialBasePosition=[7 -4 0],IsBinaryOccupied=true);
```

The cluttering SDF models are considered as loading and unloading boxes, which are accessed by robots. As these objects are non-stationary, these SDF models are not considered in the occupancy map.

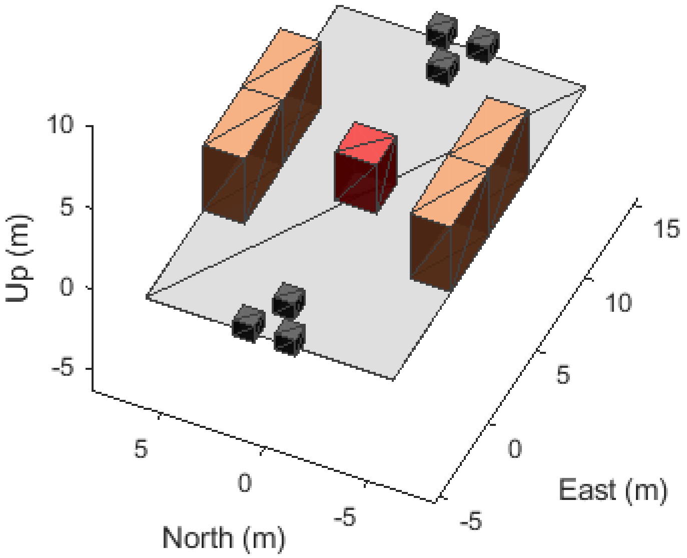
```
    loadingBoxModel = robotPlatform("LoadingBox",scenario,...
                        RigidBodyTree=clutteringRBT,...
                        InitialBasePosition=[-3 0 0],...
                        InitialBaseOrientation=eul2quat([pi/2 0 0],"ZYX"));
    unloadingBoxModel = robotPlatform("UnloadingBox",scenario,...
                        RigidBodyTree=clutteringRBT,...
                        InitialBasePosition=[13 0 0],...
                        InitialBaseOrientation=eul2quat([pi/2 0 0],"ZYX"));
```

The bucket SDF model is considered as unattended non-stationary object by the robots. The robot treats this object as obstacle and avoids if found between planned paths. This scenario element is not considered in the occupancy map.

```
    unattendedBoxModel = robotPlatform("UnattendedBox",scenario, ...
                        RigidBodyTree=bucketWareRBT,...
                        InitialBasePosition=[5 0 0]);
end
```

Visualize the scenario.

```
show3D(scenario);
view(-65,45)
light
```



Add Mobile Robots to Scenario

This example models two mobile robots that are running obstacle avoidance algorithms. The robots will be moving between two specified positions in the warehouse.

```
loadingPosition = [0 0];
unloadingPosition = [10 0];
```

Load two AMR Pioneer 3DX mobile robot from the robot library as a `rigidBodyTree` object.

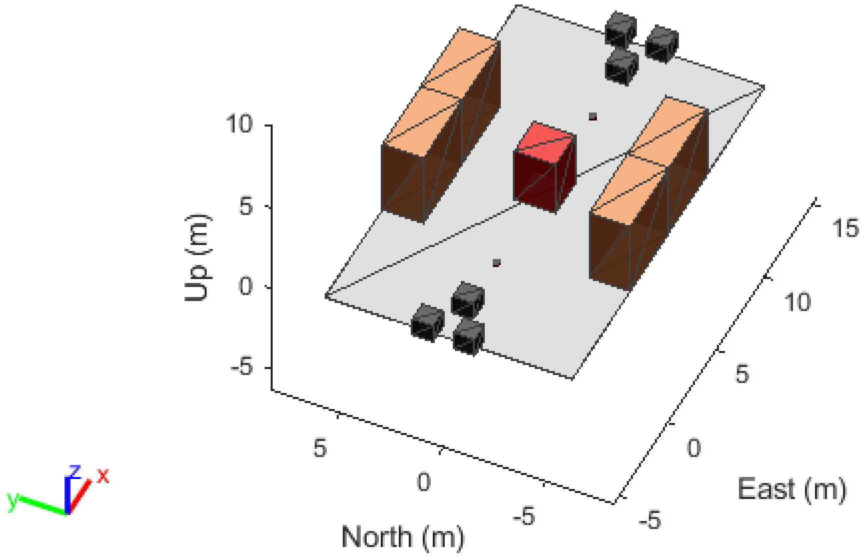
```
[pioneerRBT,pioneerRBTInfo] = loadrobot("amrPioneer3DX");
```

Add the two robot models specified as a `rigidBodyTree` object to the scenario using `robotPlatform` objects.

```
robotA = robotPlatform("MobileRobotA",scenario,...
            RigidBodyTree=pioneerRBT,...
            InitialBasePosition=[loadingPosition,0]);
robotB = robotPlatform("MobileRobotB",scenario,...
            RigidBodyTree=pioneerRBT,...
            InitialBasePosition=[unloadingPosition,0]);
```

Visualize the scenario with the robots.

```
show3D(scenario);
view(-65,45)
light
```



Mount Lidar Sensor on Robots

A lidar sensor is mounted on each robot for obstacle detection.

Specify lidar sensor configurations.

```
lidarConfig = struct(angleLower=-120,...
                           angleUpper=120,...
                           angleStep=0.2,...
                           maxRange=3,...
                           updateRate=2);
```

Create lidar sensor model.

```
lidarmodel = robotLidarPointCloudGenerator(...
                AzimuthResolution=lidarConfig.angleStep,...
                AzimuthLimits=[lidarConfig.angleLower,lidarConfig.angleUpper],...
                ElevationLimits=[0 1],...
                ElevationResolution=1,...
                MaxRange=lidarConfig.maxRange,...
                UpdateRate=lidarConfig.updateRate,...
                HasOrganizedOutput=true);
```

Mount a sensor on each of the mobile robots.

```
lidarA = robotSensor("LidarA",robotA,lidarmodel,MountingLocation=[0 0 0.2]);
lidarB = robotSensor("LidarB",robotB,lidarmodel,MountingLocation=[0 0 0.2]);
```

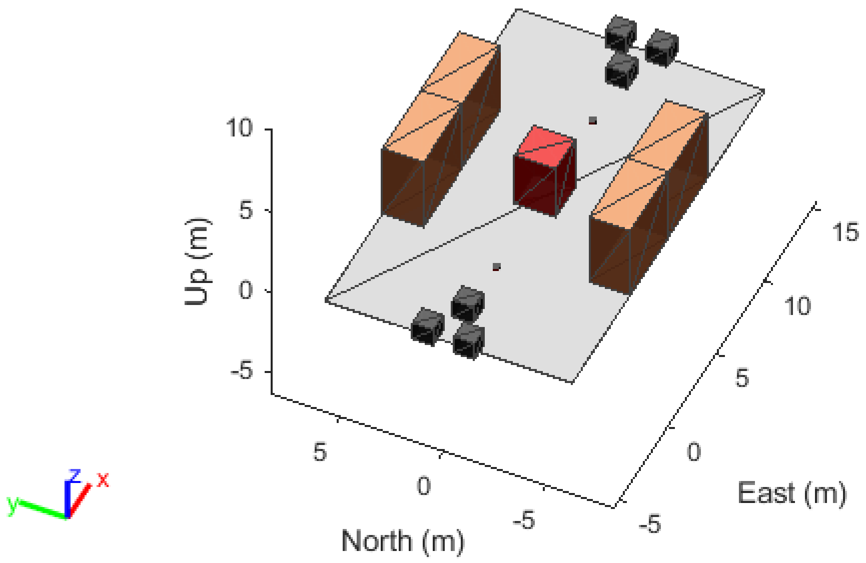
Sensor Data Visualization

During the simulation of the `robotScenario`, use the provided `plotFrames` output from the scene as the parent axes to visualize your sensor data in the correct coordinate frames.

```
[ax,plotFrames] = show3D(scenario);
view(-65,45)
light
```

Visualize the lidar sensor point cloud with scatter plot.

```
[pointCloudA,pointCloudB] = exampleHelperInitializeSensorVisualization(ax,plotFrames);
```



Plan Initial Paths for Robots

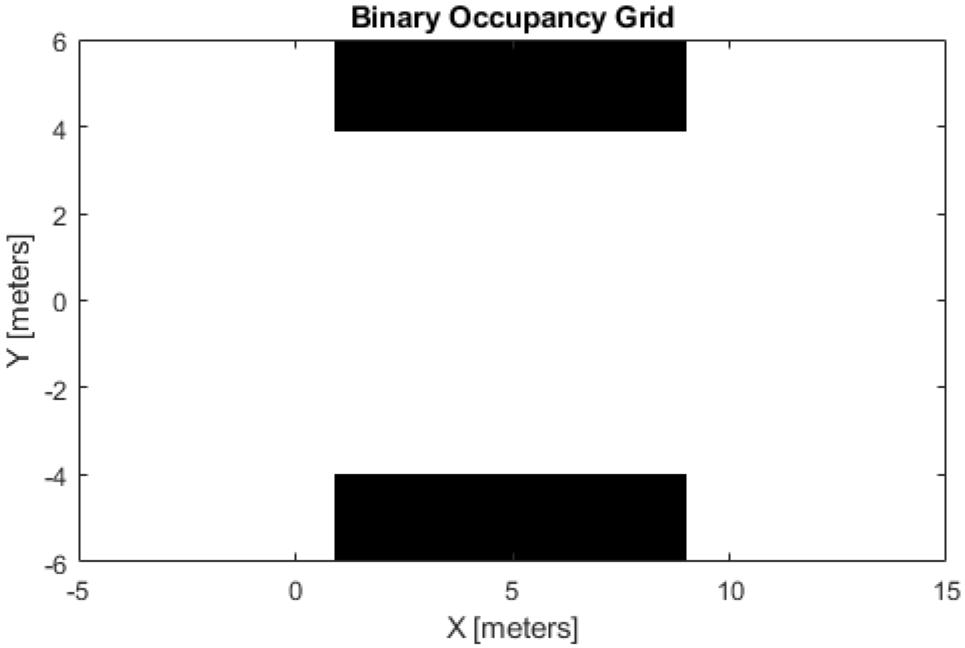
To get an initial plan, use the binary occupancy map extracted from the scenario. In `Cuboid` based scenario, left and right-side box meshes and in `SDF` based scenario, left and right-side shelf models are considered as static elements. Therefore, these areas are considered in the occupancy map, by enabling the `IsBinaryOccupied` parameter.

Get Binary Occupancy Map from Scenario

```
map = binaryOccupancyMap(scenario,MapHeightLimits=[0 5],...
                              GridOriginInLocal=[-5 -6],...
                              MapSize=[20 12]);
```

Visualize the 2-D binary occupancy map.

```
figure;
show(map)
```



Plan Initial Paths for Mobile Robots

Use the `plannerAStarGrid` planner to plan the paths for the robots with the knowledge of the obstacles known in prior. During the scenario simulation, the robots can react to other obstacles detected by the lidar sensors.

Store the grid locations of the loading and unloading positions on the map grid.

```
loadingGridLocation = world2grid(map,loadingPosition);
unloadingGridLocation = world2grid(map,unloadingPosition);
```

Plan the path for mobile robot A from loading to unloading position.

```
plannerA = plannerAStarGrid(map);
plannedGridPathA = plan(plannerA,loadingGridLocation,unloadingGridLocation);
plannedWorldPathA = grid2world(map,plannedGridPathA);
```

Plan the path for mobile robot B from unloading to loading position.

```
plannerB = plannerAStarGrid(map);
plannedGridPathB = plan(plannerB,unloadingGridLocation,loadingGridLocation);
plannedWorldPathB = grid2world(map,plannedGridPathB);
```

Create Kinematic Motion Model for Robots

The AMR Pioneer 3DX is a small differential-drive robot whose kinematic motion model is modeled using the `differentialDriveKinematics` object. Since this robot model is from the robot library, the kinematic models parameters are pre-defined. Modify the motion inputs and wheel speed range of the vehicle.

```
% Specify kinematic motion model for robot A.
robotDiffA = copy(pioneerRBTInfo.MobileBaseMotionModel);
robotDiffA.VehicleInputs = "VehicleSpeedHeadingRate";
robotDiffA.WheelSpeedRange = [-Inf Inf];

% Specify kinematic motion model for robot B.
robotDiffB = copy(robotDiffA);
```

Get Start and Goal Poses for Simulation

The current pose of each mobile robot is initialized with first waypoint from the planned path. The goal pose of each mobile robot is the last waypoint from the planned path.

```
% Get Start and Goal Poses for robot A.
robotAStartPosition = plannedWorldPathA(1,:);
robotAGoalPosition = plannedWorldPathA(end,:);
robotAInitialOrientation = 0;
robotACurrentPose = [robotAStartPosition,robotAInitialOrientation]';

% Get Start and Goal Poses for robot B.
robotBStartPosition = plannedWorldPathB(1,:);
robotBGoalPosition = plannedWorldPathB(end,:);
robotBInitialOrientation = 0;
robotBCurrentPose = [robotBStartPosition,robotBInitialOrientation]';
```

Create Controllers for Path Following and Obstacle Avoidance

Use `controllerPurePursuit` motion controller to make the robot follow the planned path.

```
% Setup controllerPurePursuit for robot A.
controllerA = controllerPurePursuit;
controllerA.Waypoints = plannedWorldPathA;
controllerA.DesiredLinearVelocity = 0.6;
controllerA.MaxAngularVelocity = 2;
controllerA.LookaheadDistance = 0.3;

% Setup controllerPurePursuit for robot B.
controllerB = controllerPurePursuit;
controllerB.Waypoints = plannedWorldPathB;
controllerB.DesiredLinearVelocity = 0.6;
controllerB.MaxAngularVelocity = 2;
controllerB.LookaheadDistance = 0.3;
```

To react to obstacles in the path, the robots need to be equipped with an obstacle avoidance algorithm. The `controllerVFH` computes steering angle based on the inputs from a lidar sensor.

```
% Setup controllerVFH for robot A.
vfhA = controllerVFH;
vfhA.UseLidarScan = true;
vfhA.SafetyDistance = 1;
```

```
% Setup controllerVFH for robot B.
vfhB = controllerVFH;
vfhB.UseLidarScan = true;
vfhB.SafetyDistance = 1;
```

Simulate Robots in Scenario

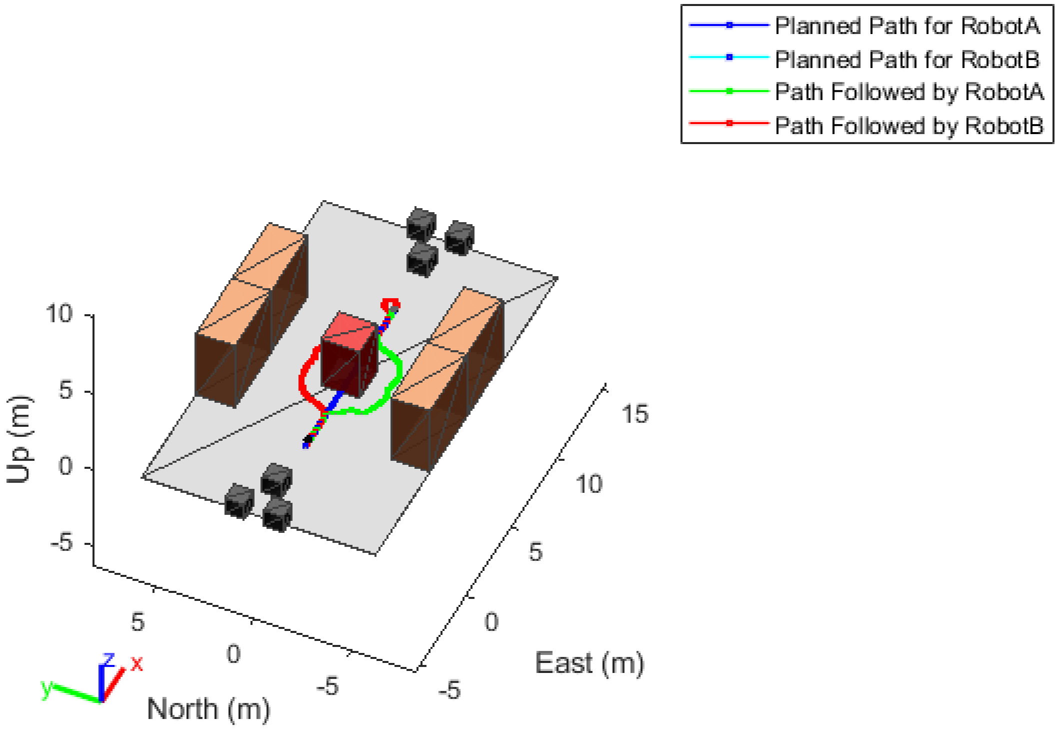
Now that the warehouse scenario has been configured, visualize the original planned paths and then simulate the actual behavior of the mobile robots in the scenario.

Start by overlaying the original planned paths on the scenario. Observe how the robot moves with regards to the unexpected obstacle.

```
hold(ax,"on")
% Visualize planned path for robot A.
plannedPathA = plot(ax,plannedWorldPathA(:,1),plannedWorldPathA(:,2),"-bs",...
                    LineWidth=1,...
                    MarkerSize=1.5,...
                    MarkerEdgeColor="b",...
                    MarkerFaceColor=[0.5 0.5 0.5]);

% Visualize planned path for robot B.
plannedPathB = plot(ax,plannedWorldPathB(:,1),plannedWorldPathB(:,2),"-cs",...
                   LineWidth=1,...
                   MarkerSize=1.5,...
                   MarkerEdgeColor="b",...
                   MarkerFaceColor=[0.5 0.5 0.5]);
hold(ax,"off")
```

Set up and run the simulation.

```
setup(scenario)

robotAReached = false;
robotBReached = false;
stopSimulation = false;

while ~stopSimulation

    if ~robotAReached
        % Get current pose of the robot A, along with lidar sensor point
        % cloud till robot reaches to destination.
        [robotACurrentPose, isUpdatedA, robotAReached, pointCloudA] = ...
            exampleHelperGetCurrentPose(controllerA, robotDiffA, vfhA, lidarA, ...
                    lidarConfig, robotACurrentPose, robotAGoalPosition, robotAReached);
    end

    if ~robotBReached
        % Get current pose of the robot B, along with lidar sensor point
        % cloud till robot reaches to destination.
        [robotBCurrentPose, isUpdatedB, robotBReached, pointCloudB] = ...
            exampleHelperGetCurrentPose(controllerB, robotDiffB, vfhB, lidarB, ...
                    lidarConfig, robotBCurrentPose, robotBGoalPosition, robotBReached);
    end

    % Stop simulation if both robots reached to the destination.
    if robotAReached && robotBReached
        stopSimulation = true;
    end

    % Update plot when lidar sensor readings are updated.
    if isUpdatedA || isUpdatedB
        % Use fast update to move platform visualization frames.
        show3D(scenario,FastUpdate=true,Parent=ax);
        % Refresh all plot data and visualize.
        refreshdata
        drawnow limitrate
    end

    if ~robotAReached
        % Move robot A with current robot pose.
        move(robotA,"base",[robotACurrentPose(1), robotACurrentPose(2), 0, ...
                            zeros(1,6), eul2quat([robotACurrentPose(3), 0,0]),...
                            zeros(1,3)]);

        hold(ax,"on")
        % Visualize the path followed by Robot A.
        followedPathA = plot(ax,robotACurrentPose(1),robotACurrentPose(2),"-gs",...
                                LineWidth=1,...
                                MarkerSize=1.5,...
                                MarkerEdgeColor="g",...
                                MarkerFaceColor=[0.5 0.5 0.5]);
        hold(ax,"off")

    end

    if ~robotBReached
        % Move robot B with current robot pose.
        move(robotB,"base",[robotBCurrentPose(1), robotBCurrentPose(2), 0, ...
                            zeros(1,6), eul2quat([robotBCurrentPose(3), 0,0]),...
                            zeros(1,3)]);

        hold(ax,"on")
        % Visualize the path followed by Robot B.
        followedPathB = plot(ax,robotBCurrentPose(1),robotBCurrentPose(2),"-rs",...
                                LineWidth=1,...
                                MarkerSize=1.5,...
                                MarkerEdgeColor="r",...
                                MarkerFaceColor=[0.5 0.5 0.5]);
        hold(ax,"off")
    end

    hold(ax,"on")
    legend(ax,[plannedPathA, plannedPathB, followedPathA, followedPathB],...
               ["Planned Path for RobotA","Planned Path for RobotB",...
                "Path Followed by RobotA","Path Followed by RobotB"])
    hold(ax,"off")
    % Advance scenario simulation time.
    advance(scenario);
    % Update all sensors in the scenario.
    updateSensors(scenario);
end
```


Planned Path for RobotA
Planned Path for RobotB
Path Followed by RobotA
Path Followed by RobotB
Up (m)
10
5
0
-5
North (m)
5
0
-5
East (m)
-5
0
5
10
15
z
x
y

Perform Co-Simulation between Simulink and Gazebo

This example shows how to set up a synchronized simulation between Simulink™ and Gazebo to send commands and receive data from Gazebo.

Setup Gazebo Simulation Environment

For this example, use your own Linux environment with Gazebo or download the provided Virtual Machine with ROS and Gazebo. In the virtual machine (VM), the required Gazebo plugin is located in /home/user/src/GazeboPlugin.

For more information on the Linux VM and the requirements for setting up your own Linux environment with Gazebo, see “Gazebo Simulation Environment Requirements and Limitations” on page 2-65.

If using your own Linux environment, follow the steps in Install Gazebo Plugin Manually on page 1-0 . Othwerise, go to Launch Gazebo Simulation Environment on page 1-0 .

Install Gazebo Plugin Manually

Obtain the plugin source code as a zip package. This function creates a folder called GazeboPlugin in your current working directory and compresses it as GazeboPlugin.zip.

```
packageGazeboPlugin
```

Copy the GazeboPlugin.zip to your Linux machine that meets the following requirement:

Unzip the package on your Linux platform, for this example we unpack to /home/user/src/GazeboPlugin.

Run these commands in the terminal to compile the plugin.

```
cd /home/user/src/GazeboPlugin
```

If the build folder already exists, remove it.

```
rm -r build
```

Install the plugin.

```
mkdir build
cd build
cmake ..
make
```

The plugin location is /home/user/src/GazeboPlugin/export/lib/libGazeboCoSimPlugin.so.

Remove the generated plugin from host computer.

```
if exist('GazeboPlugin', 'dir')
    rmdir('GazeboPlugin', 's');
end

if exist('GazeboPlugin.zip', 'file')
    delete('GazeboPlugin.zip');
end
```

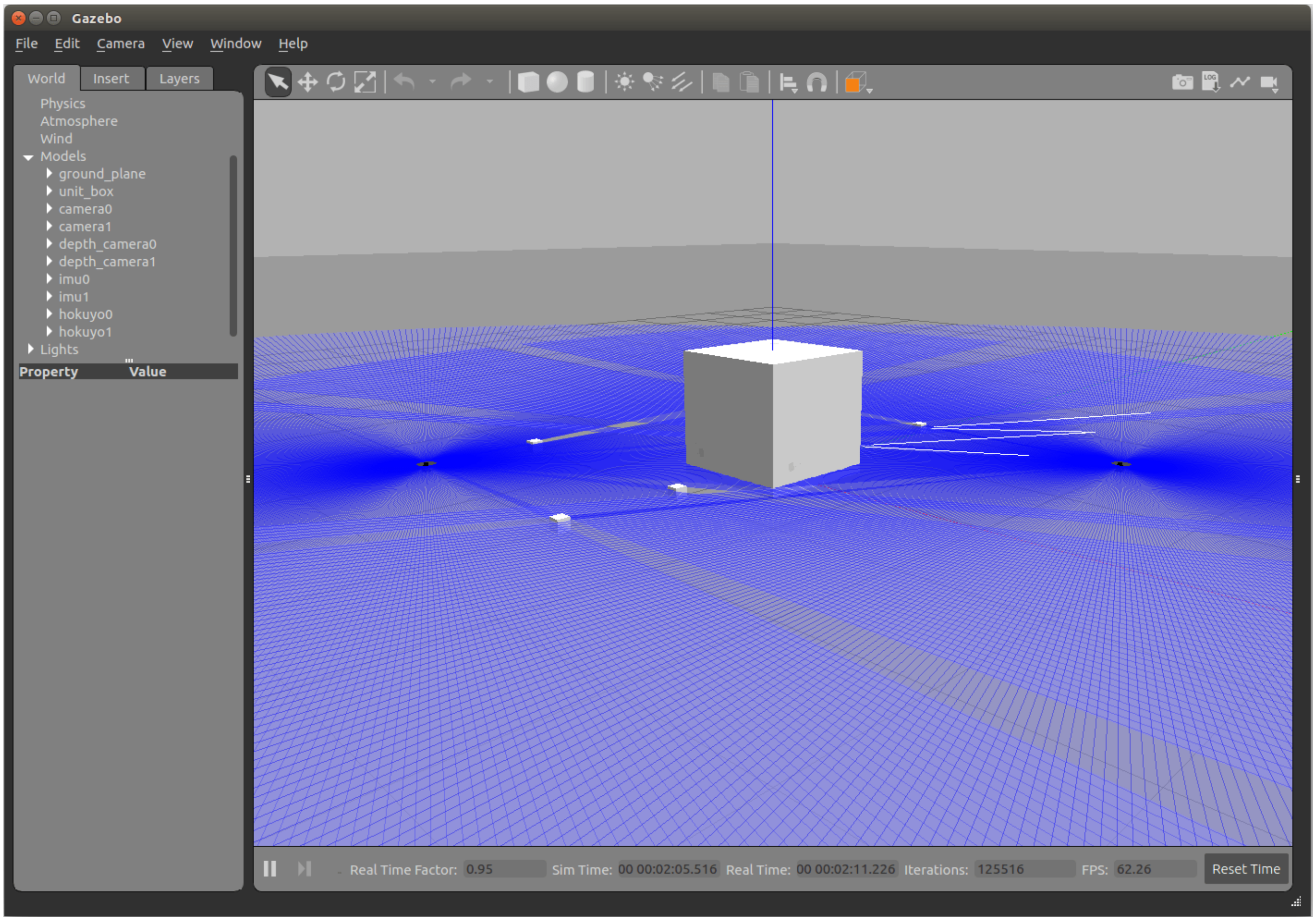
Launch Gazebo Simulation Environment

Open a terminal in the VM or your own Linux operating system, run the following commands to launch the Gazebo simulator.

```
cd /home/user/src/GazeboPlugin/export
export SVGA_VGPU10=0
gazebo ../world/multiSensorPluginTest.world --verbose
```

These commands launch a Gazebo simulator with:

- Two laser range finders: `hokuyo0` and `hokuyo1`
- Two RGB cameras: `camera0` and `camera1`
- Two depth cameras: `depth_camera0` and `depth_camera1`
- Two IMU sensors: `imu0` and `imu1`
- Unit box model: `unit_box`



Real Time Factor: 0.95 Sim Time: 00 00:02:05.516 Real Time: 00 00:02:11.226 Iterations: 125516 FPS: 62.26

The `multiSensorPluginTest.world` is located in `/home/user/src/GazeboPlugin/world` folder. This world file includes the Gazebo plugin for co-simulation with Simulink using the following lines in its `.xml` body:

```
<plugin name="GazeboPlugin" filename="lib/libGazeboCoSimPlugin.so"><portNumber>14581</portNumber
```

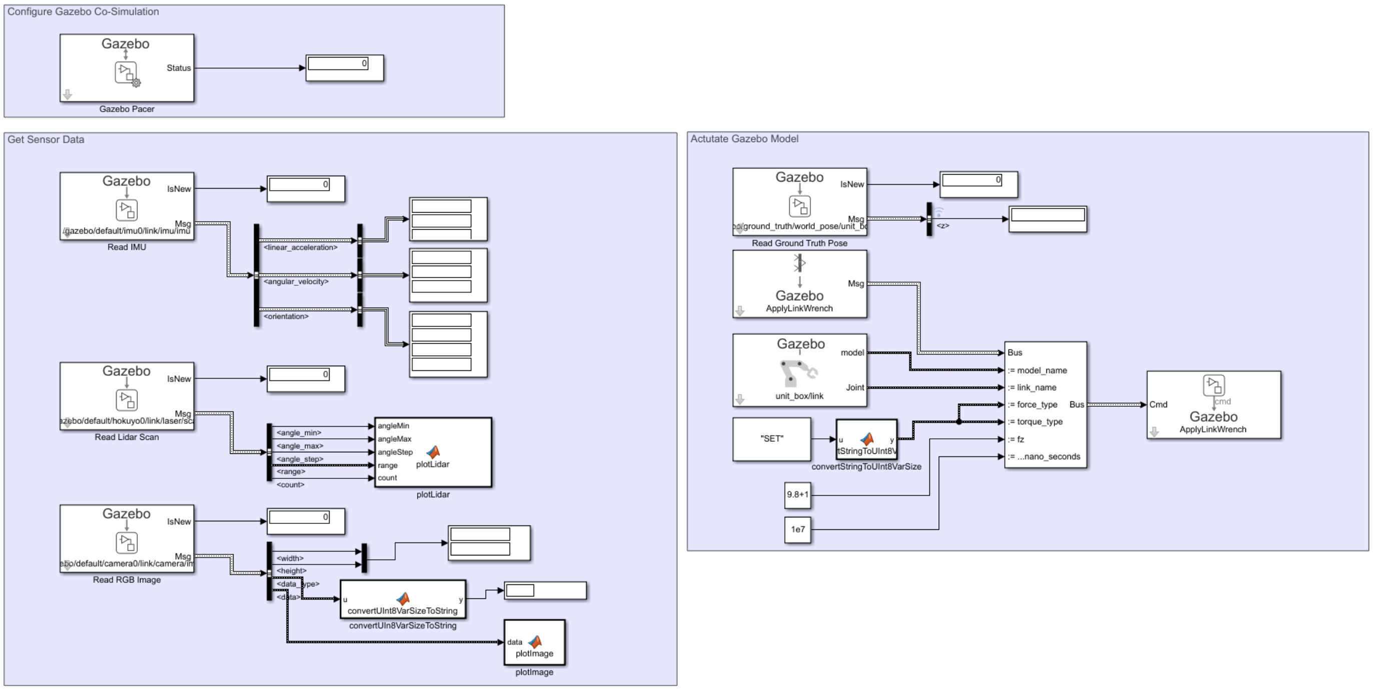
The filename field must be pointing to the location of the compiled Gazebo plugin. This path can be relative to the location Gazebo itself is launched, or you could add it to the Gazebo plugin search path by running:

```
export GAZEBO_PLUGIN_PATH=${GAZEBO_PLUGIN_PATH}:/home/user/src/GazeboPlugin/export
```

Configure Gazebo Co-Simulation

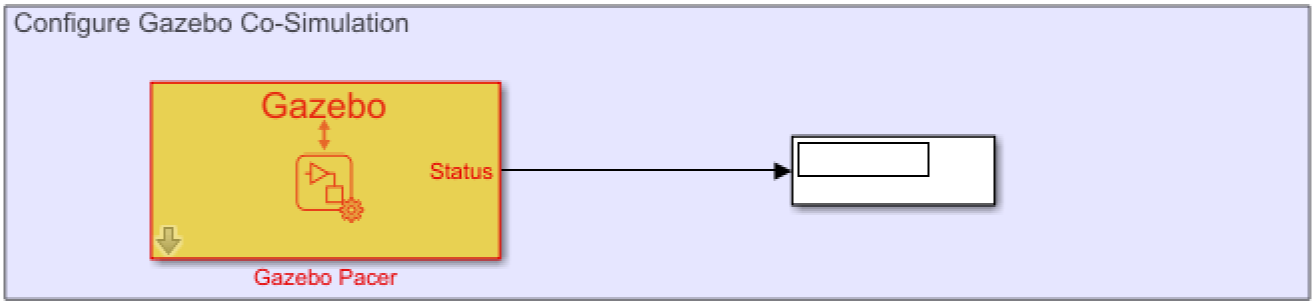
Open the `performCoSimulationWithGazebo` model, which demonstrates how to receive sensor data from these simulated sensors and how to actuate the unit box model from Simulink.

```
open_system("performCoSimulationWithGazebo")
```



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

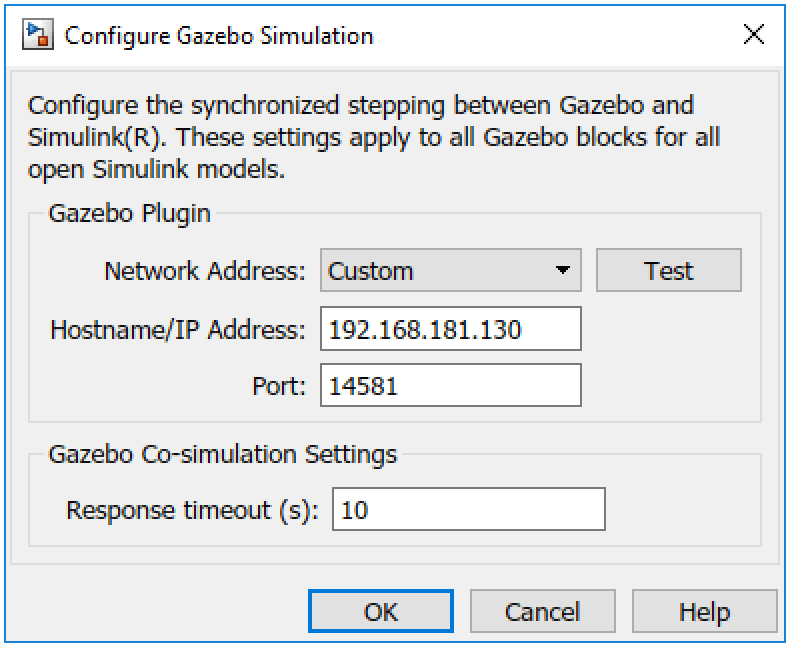
```
hilite_system('performCoSimulationWithGazebo/Gazebo Pacer')
```



Open the block and click the **Configure Gazebo network and simulation settings** link.

```
open_system('performCoSimulationWithGazebo/Gazebo Pacer')
```

In the **Network Address** drop down, select `Custom`. Enter the IP address of your Linux machine. The default **Port** for Gazebo is `14581`. Set **Response timeout** to 10 seconds.



Configure the synchronized stepping between Gazebo and Simulink(R). These settings apply to all Gazebo blocks for all open Simulink models.

Network Address: Custom

Hostname/IP Address: 192.168.181.130

Port: 14581

Response timeout (s): 10

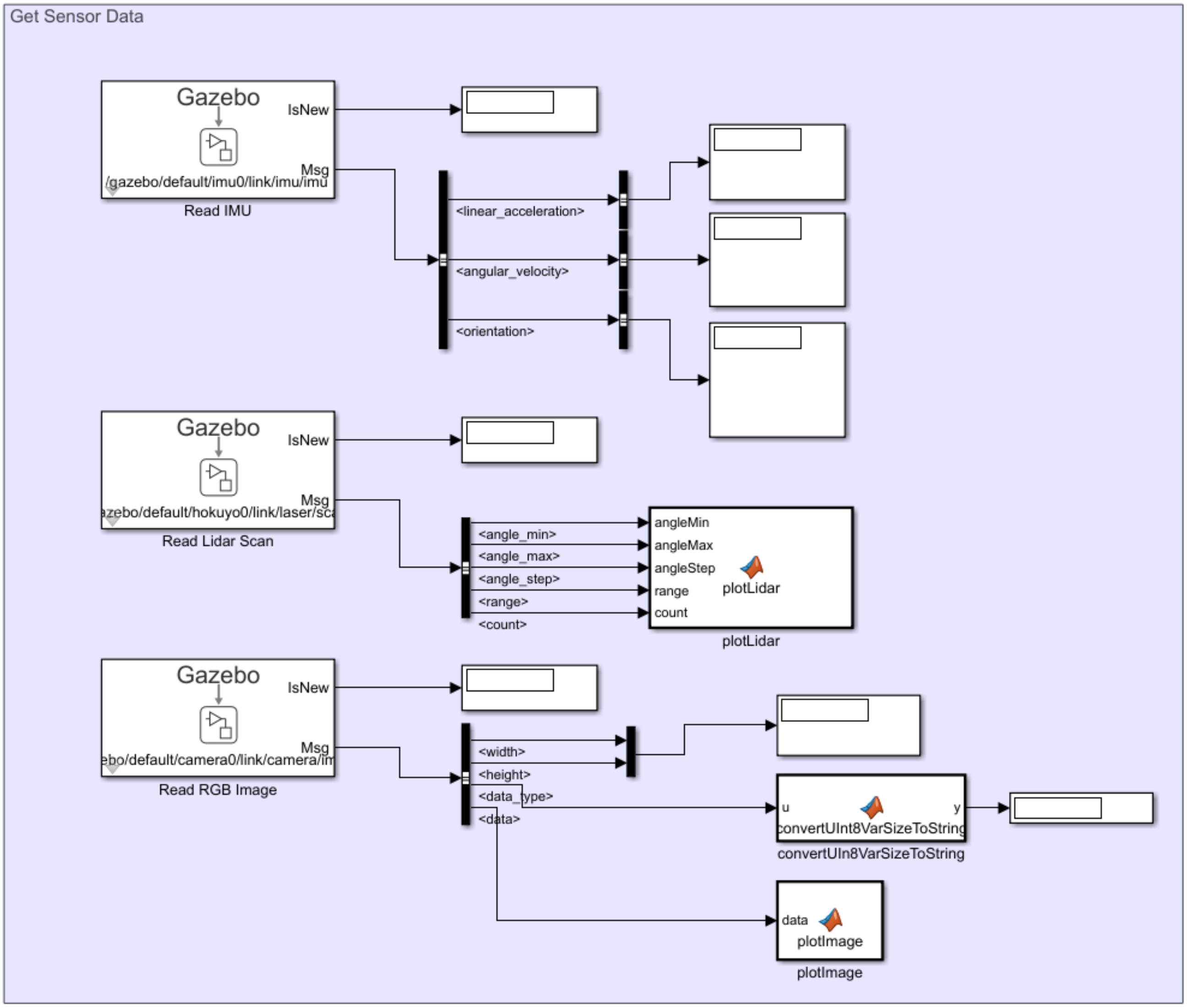
Click the **Test** button to test the connection to the running Gazebo simulator.

Get Sensor Data

Use the **Gazebo Read** block to obtain data on specific topics from three sensors:

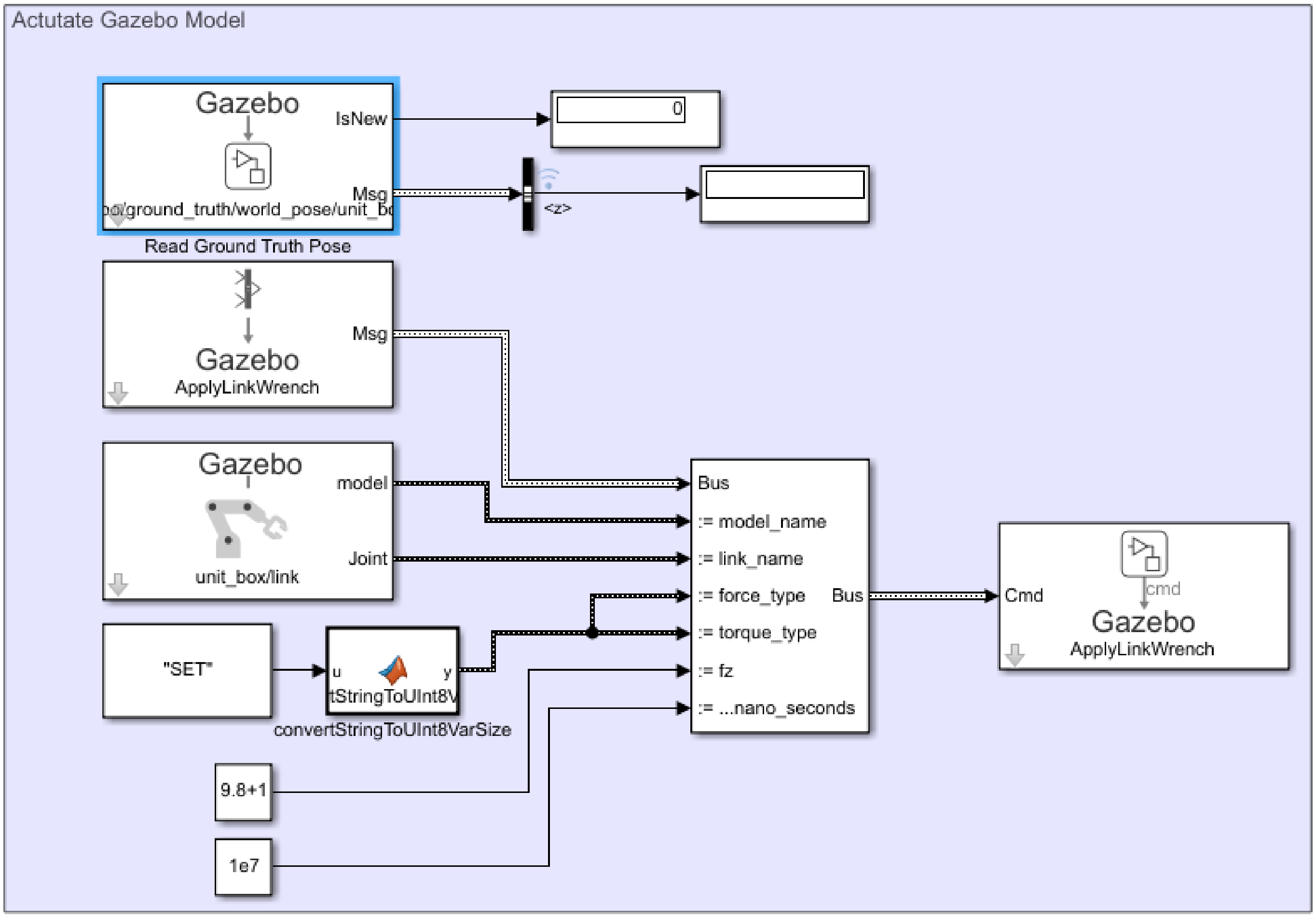
- IMU, `/gazebo/default/imu0/link/imu/imu`
- Lidar Scan, `/gazebo/default/hokuyo0/link/laser/scan`
- RGB camera, `/gazebo/default/camera0/link/camera/image`

Display the IMU readings and visualize the Lidar Scan and RGB Image using MATLAB® function blocks.



Actuate Gazebo Model

Use the **Gazebo Apply Command** block to apply a constant force in the *z*-direction to the unit box that results in an acceleration of 1 m/s^2. Create a blank `ApplyLinkWrench` message using **Gazebo Blank Message**. Specify elements of the message to apply the force to the `unit_box/link` entity using the **Bus Assignment** block. Use **Gazebo Read** to output the ground truth pose of the box. The displacement of the box over a 1 second period should be close to 0.5 meters.



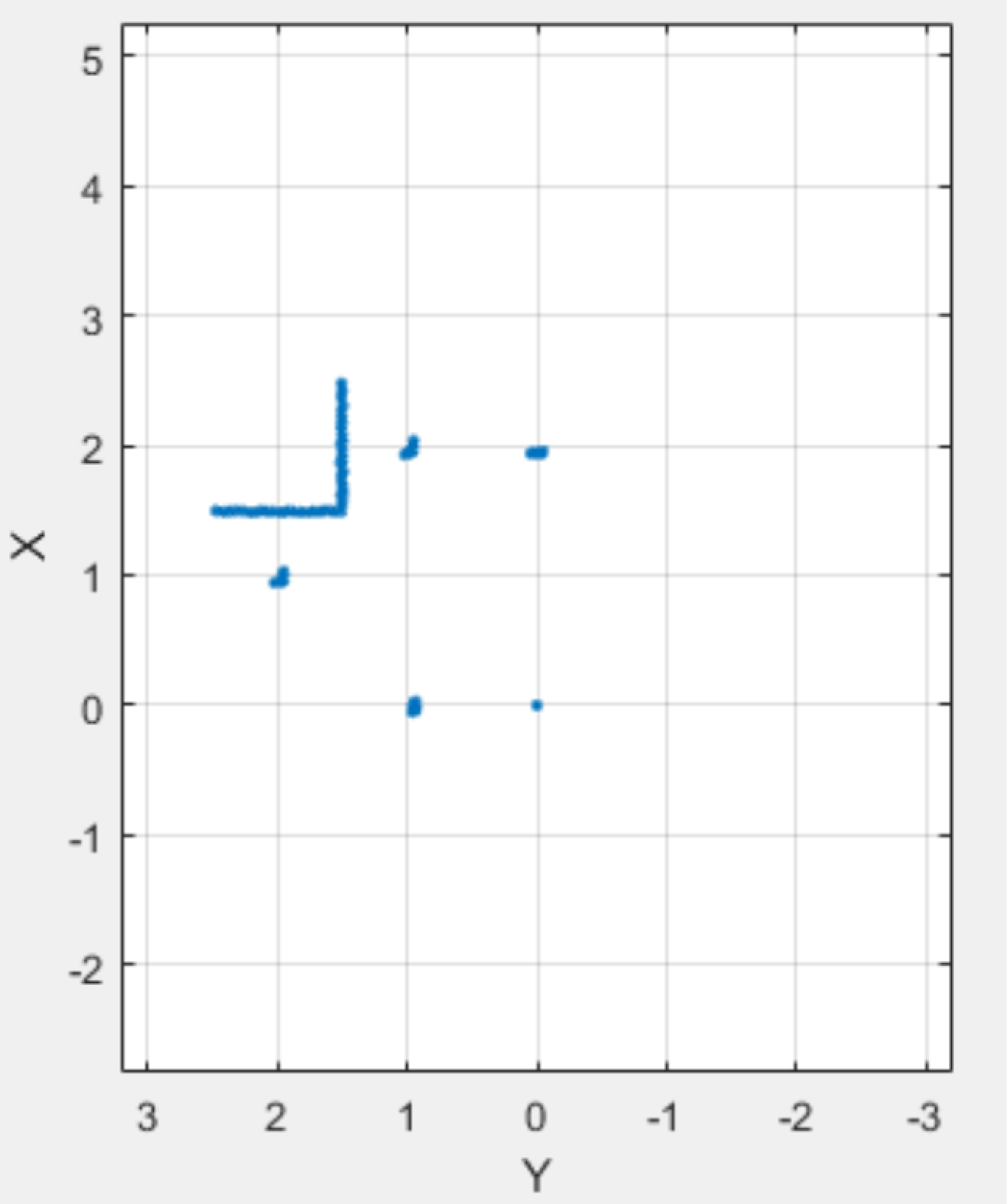
Perform Co-Simulation

To start co-simulation,click **Run**. You can also step the simulation using **Step Forward**. **Step Back** is not supported during co-simulation.

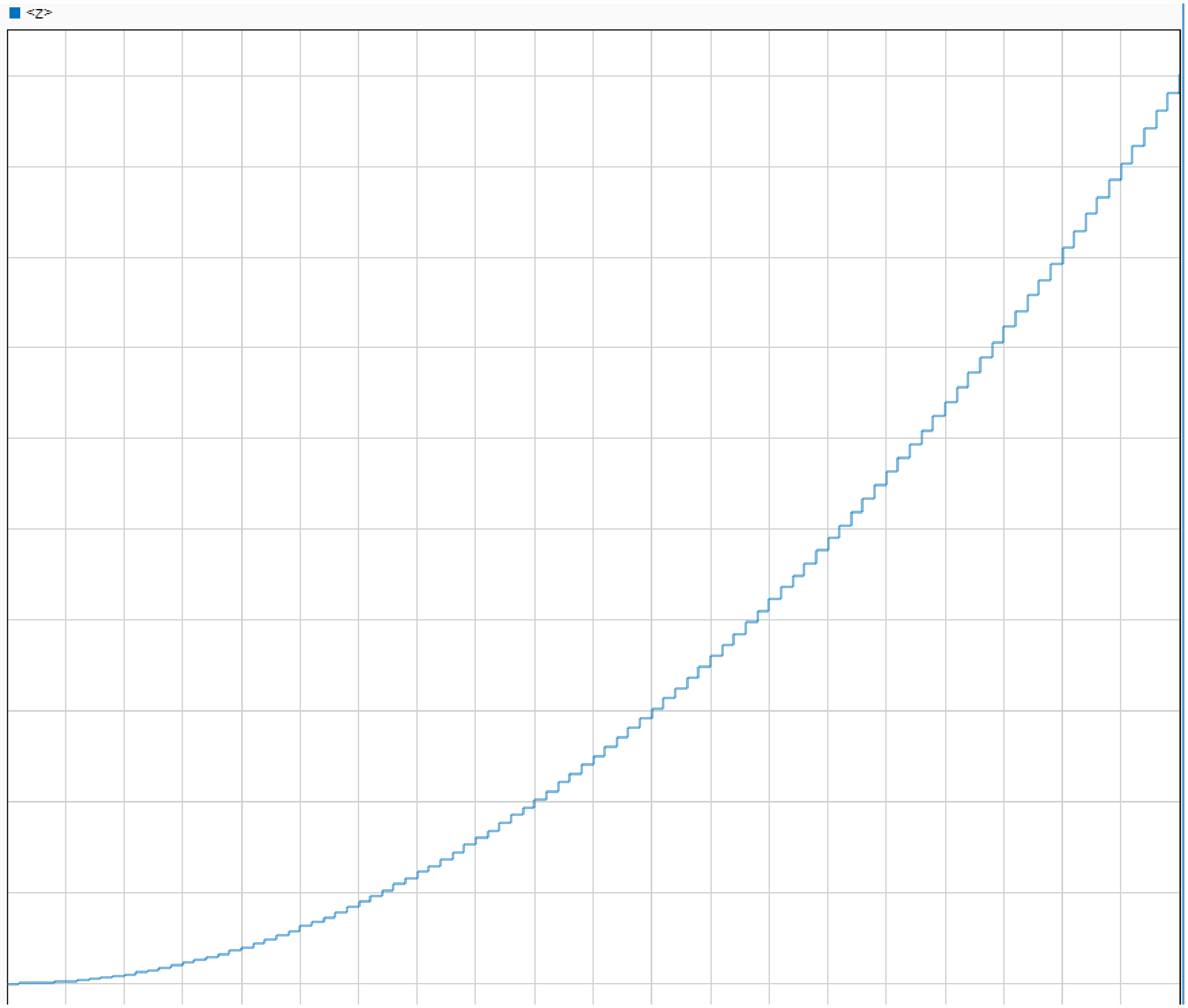
While the simulation is running, notice that Gazebo simulator and Simulink time are synchronized.

This model visualizes the Gazebo sensor data using MATLAB function block and MATLAB plotting functionalities. Here is a snapshot of the image data obtained from Gazebo camera:

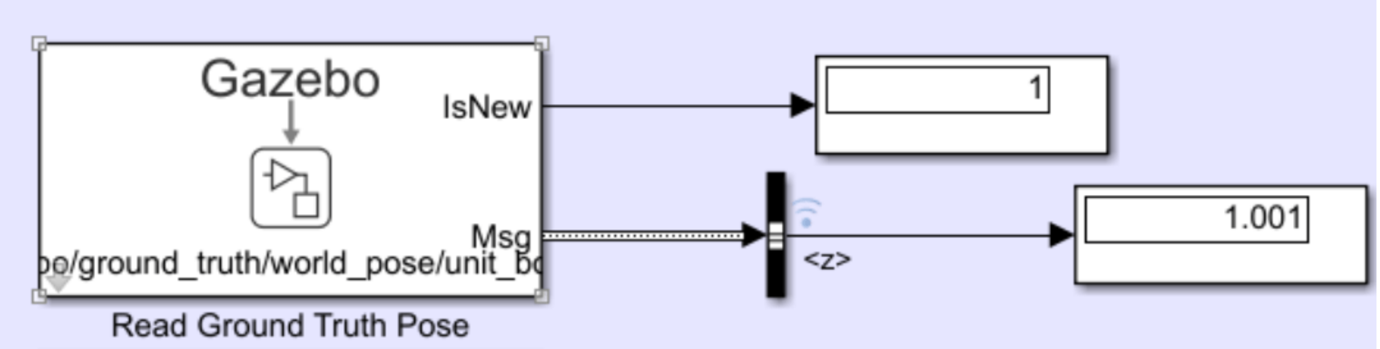
Here is a snapshot of the lidar scan image:



The time-plot of the position of the unit box block in *z*-direction can be viewed using **Data Inspector**. The block tracks a parabolic shape due to the constant acceleration over time.



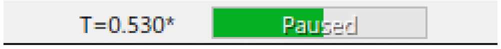
The position of the unit box at the end of simulation is 1.001, leading to a 0.5001 displacement, which is slightly different from the expected value of 0.5. This is due to the error of the Gazebo physics engine. Make the max step size in the Gazebo physics engine smaller to reduce this error.



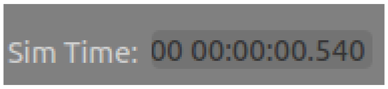
Time Synchronization

During co-simulation, you can pause Simulink and the Gazebo Simulator at any time using **Pause**

Note: Gazebo pauses one time step ahead of the simulation.

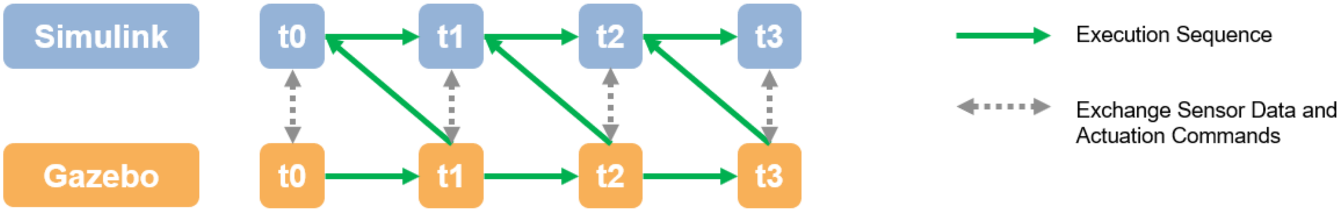


T=0.530* Paused



Sim Time: 00 00:00:00.540

This is due to the following co-simulation time sequence:



Sensor data and actuation commands are exchanged at the correct time step. The execution chooses to step Gazebo first, then Simulink. The simulation execution is still on the *t+1*, Simulink just stays on the previous step time until you resume the model.

Next Steps

- "Control a Differential Drive Robot in Gazebo with Simulink" on page 1-70

Configure Gazebo and Simulink for Co-simulation of a Manipulator Robot

Set up a UR10 robot model to perform co-simulation between Gazebo and Simulink™. Co-simulation with Gazebo enables you to connect directly from Simulink to Gazebo and control simulation pacing using the Simulink model.

For an introduction to Gazebo co-simulation and getting connected for the first time, see "Perform Co-Simulation between Simulink and Gazebo" on page 1-431. This example uses the virtual machine and plug-in provided in that example, but also tells you how to configure your own model and system. The robot arm used in this example is a 6-DoF robot, the Universal Robots UR10.

Create Robot Representations for MATLAB and Gazebo

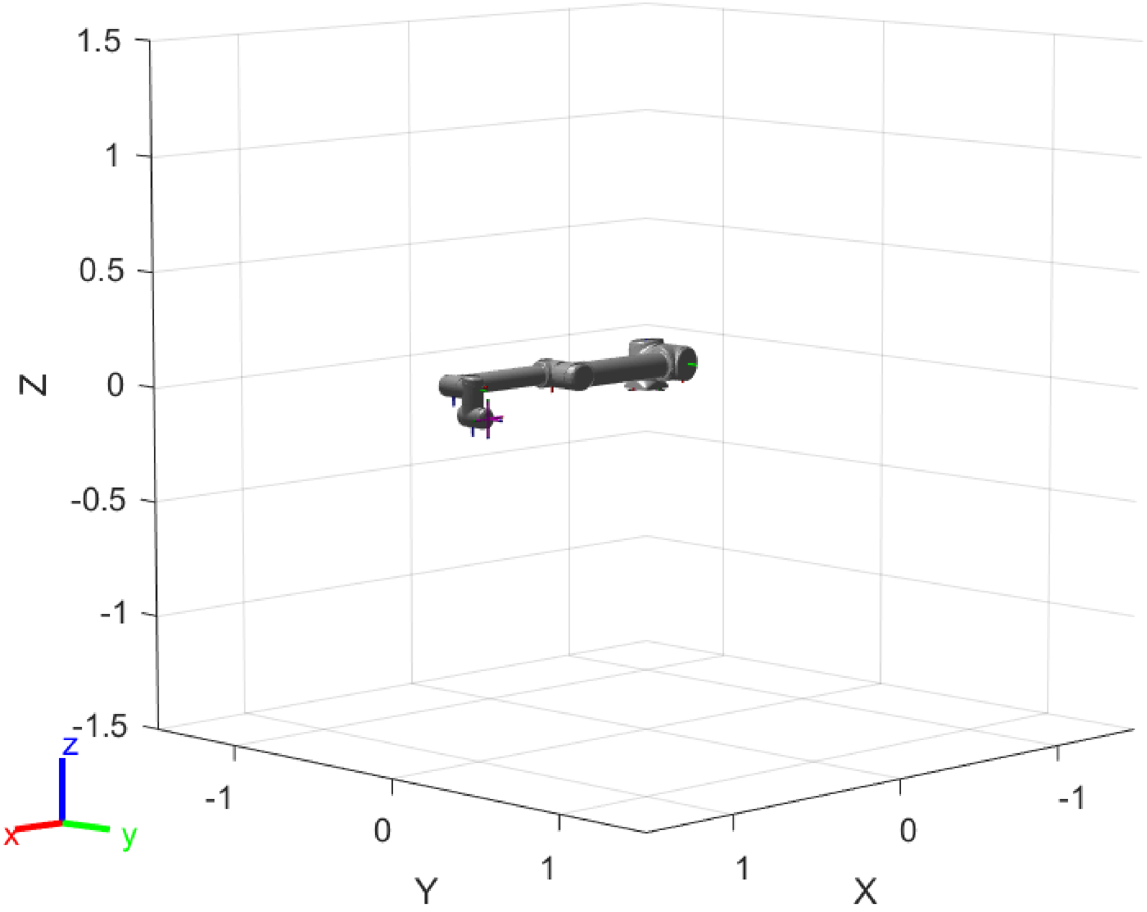
First, add the manipulator to a Gazebo *.world* file. In this example, a world file is provided (`Ur10BasicWithPlugin.world`) that was created from the provided robot models in the `loadrobot` function. If your goal is just to interface with the Gazebo simulation, a world file is sufficient. To see how to control a model using just the world file, see "Control a Differential Drive Robot in Gazebo with Simulink" on page 1-70. However, the goal of this example is to control the robot using model-based tools in MATLAB® and Simulink. Therefore, you should also define a rigid body tree robot model for your manipulator in MATLAB. There are several ways to obtain these representations together:

- **Provided Robot Models:** Obtain the rigid body tree from the `loadrobot` function, then create a model for Gazebo using the source repository.
- **Custom URDF Models:** Import a URDF file as a rigid body tree object via the `importrobot` function. Manually modify the URDF file to make it compatible with the Gazebo SDF model format. Save the model to a new or existing *.world* file. If you only have a Gazebo *.world* file, consider modifying a similar manipulator or building the robot from scratch, using queries of the Gazebo world.

Load Robot Model

To get started, load and show the robot in MATLAB.

```
[robot,info] = loadrobot('universalUR10','Gravity',[0 0 -9.81]);
show(robot);
```



Create Gazebo World from Robot Model

The `Ur10BasicWithPlugin.world` world file attached to this example is also provided in the virtual machine from the “Perform Co-Simulation between Simulink and Gazebo” on page 1-431 example. If you are using the virtual machine provided in that example, skip directly to Open World in Gazebo on page 1-0 .

The world file contains a UR10 robot with the base joint fixed to the ground and a red box to manipulator. The world was created using the source repository provided in the `info` output:

```
info.Source
```

```
ans = 
"https://github.com/ros-industrial/universal_robot/tree/1.2.1"
```

The following steps were used to create the `Ur10BasicWithPlugin.world` file:

- Open the repository linked in `info.Source` and install per the instructions in the `readme` file. Make sure the tags match so that the robot's numerical data matches that used in the equivalent rigid body tree object. If using the provided virtual machine, make sure to allot at least 4 GB RAM and 4 cores to avoid compilation errors. On the provided Virtual Machine, some of the tools in the associated ROS package have been disabled to avoid dependency issues during installation:

```
%# Ignore the ur_kinematics package
touch universal_robot/ur_kinematics/CATKIN_IGNORE
```

- The repository installs a package to open a world using the `roslaunch` command:

```
roslaunch ur_gazebo ur10.launch
```

- Save the newly created world to a *.world* file. You may also save the model to an SDF file and add it to an existing world, manually or via the Gazebo GUI.
- In the world file source code, fix the robot to the ground plane by manually adding a fixed joint to the robot model.

```
<!-- Add a custom fixed joint that fixes the robot to the world-->
<joint name="world_to_robot" type="fixed">
   <parent>world</parent>
   <child>ur10::base_link</child>
</joint>
```

- In the world file source code, add a red box object that can be moved around.

```
<!-- Add a red box -->
    <model name='redBox'>
      <link name='link'>
% See file for more
```

- In the world file source code, attach the Gazebo plug-in by adding the following lines:

```
<!-- Include the Gazebo plugin to ensure connection to MATLAB/Simulink -->
<plugin name="GazeboPlugin" filename="lib/libGazeboCoSimPlugin.so">
    <portNumber>14581</portNumber>
</plugin>
```

The `Ur10BasicWithPlugin.world` world file attached to this example has additional comments to clarify the various elements. To learn more about robot elements in a Gazebo world, search the Gazebo documentation for the "Make A Simple Gripper" tutorial.

Set Up Gazebo with Robot Model and Plugin

To run this example, you must have access to a machine with Gazebo with the plugin for co-simulation installed and the provided world file. These steps are covered in “Perform Co-Simulation between Simulink and Gazebo” on page 1-431. If you are using the virtual machine provided in that example, skip directly to Open World in Gazebo on page 1-0 . Virtual machines downloaded prior to the R2021a release may need to be updated.

To execute this example, you must either be able to connect to a machine with Linux installed with an environment that has been configured accordingly:

- Set up the Gazebo environment and add the plug-in. Follow the **Install Gazebo Plugin Manually** instructions in “Perform Co-Simulation between Simulink and Gazebo” on page 1-431. This example assumes that the plug-in is located in the `home/user/src/GazeboPlugIn` directory.
- Add the provided world file to the directory `/home/user/worlds`
- Access the robot mesh files from the robot model. The mesh files are Collada (*.DAE files) that are required to visualize the robot. This robot was created using `loadrobot`, so the repository linked in `info.Source` contains all the associated information, including meshes.

```
disp(info.Source)
```

```
https://github.com/ros-industrial/universal_robot/tree/1.2.1
```

- Execute the following commands on the Gazebo machine to clone the linked repository. The URL in the second command should match `info.Source`.

```
cd /home/user/catkin_ws/src
git clone -b 1.2.1 https://github.com/ros-industrial/universal_robot/

%# (Optional) Ignore UR_Kinematic package to avoid dependency issue during build
touch universal_robot/ur_kinematics/CATKIN_IGNORE
```

Please note that is is not necessary to build the package for this example to work (this example just requires the associated mesh files), though doing so will not cause any issues either.

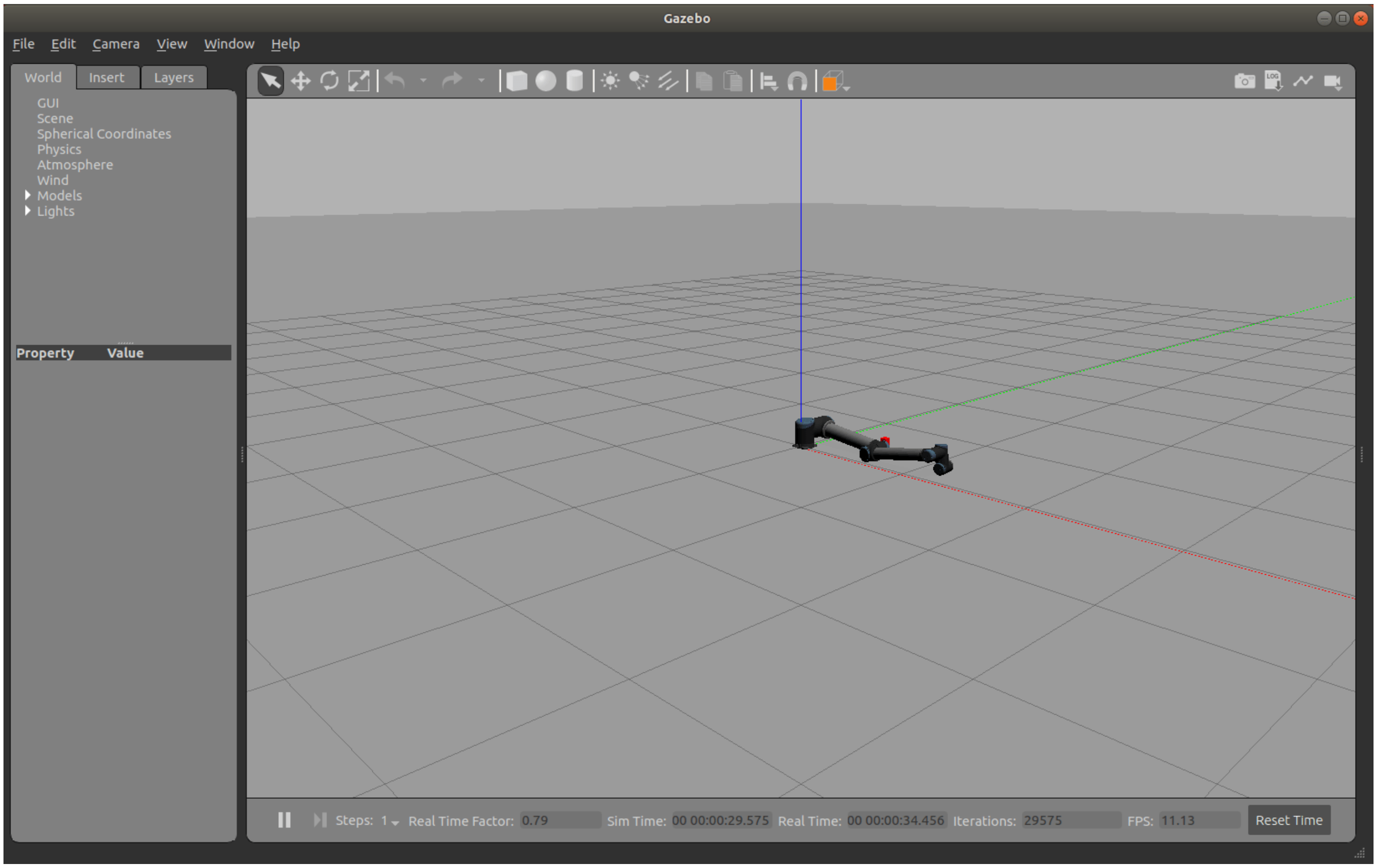
Check to ensure the robot mesh files are located at `/home/user/catkin_ws/src/universal_robot/ur_description/`.

Open World in Gazebo

Open the world by running these commands in the terminal of the Gazebo machine:

```
cd /home/user/src/GazeboPlugin/export
export SVGA_VGPU10=0
gazebo /home/user/worlds/Ur10BasicWithPlugin.world --verbose
```

Gazebo shows the robot and any other objects in the world.



Steps: 1 Real Time Factor: 0.79 Sim Time: 00 00:00:29.575 Real Time: 00 00:00:34.456 Iterations: 29575 FPS: 11.13

If the Gazebo simulator fails to open, you may need to reinstall the plugin. See **Install Gazebo Plugin Manually** in "Perform Co-Simulation between Simulink and Gazebo" on page 1-431.

Connect to Gazebo

Next, initialize the Gazebo connection to MATLAB and Simulink. Specify the IP address and a port number of `14581`, which is the default port for the Gazebo plugin.

```
ipGazebo = '192.168.116.162'; % Replace this with the IP of the Gazebo machine
gzinit(ipGazebo,14581);
```

Inspect the Model

Verify that the model has the desired elements using the Gazebo MATLAB interface commands. Call the `gzmodel` function to return all the models in the current Gazebo world.

```
gzmodel('list')
```

```
MODEL LIST:
ground_plane
ur10
redBox
```

List the links in the UR10 robot model.

```
gzlink("list","ur10")
```

```
MODEL:  ur10


LINKS:  base_link forearm_link shoulder_link upper_arm_link wrist_1_link wrist_2_link wrist_3_lin
```

Control Robot Position in Simulink

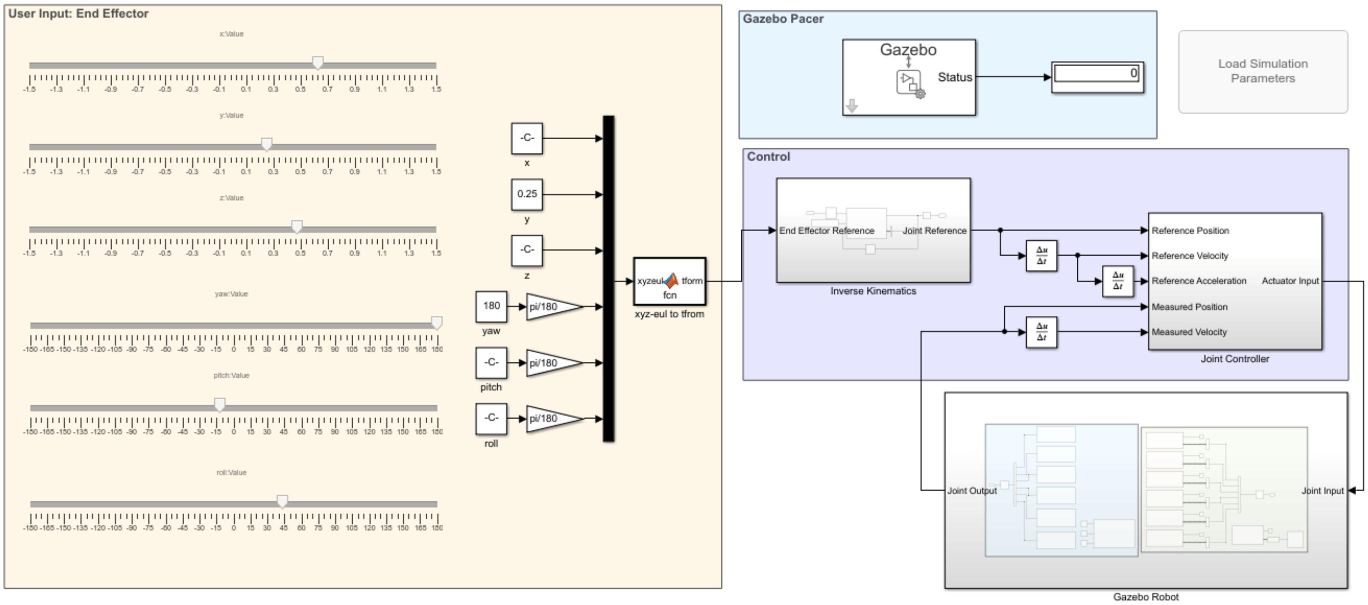
Now that the Gazebo world is connected to MATLAB, you can perform co-simulation from Simulink to advance the simulation state in Gazebo.

The `gazeboCosimControl` model controls the end-effector position of the manipulator using the sliders in the **User Input: End Effector** section. The **Inverse Kinematics** subsystem generates a joint configuration that achieves the desired postion. Then, the **Joint Controller** subsystems generates torque forces for each joint to reach this position. Details about this model can be found in the "Control Manipulator Robot with Co-Simulation in Simulink and Gazebo" on page 1-447 example.

The **Gazebo Pacer** block controls the stepping of Gazebo based on Simulink steps.

Open the model and initialize the model parameters for the trajectory, starting position, and bus objects for sending commands.

```
% Open the model
open_system('gazeboCosimControl');

% Initialize parameters
Ts = 0.01;
Ts_trajectory = 0.05;
q0 = [0 -70 140 0 0 0]' * pi/180;
load('custom_busobjects_basic');
```



Simulate the model for a few seconds.

```
sim('gazeboCosimControl','StopTime','5');
```

Use the sliders in the **User Input: End Effector** area on the left to control the position of the robot.

Now run the model directly using the green "run" arrow.

While the simulation is running, you may send MATLAB commands to update the world state. For example, move the red box to a new location.

```
% Move the box to a new position on the opposite side of the robot and 0.3 m off the ground
gzlink("set","redBox","link","Position",[0.5 -0.4 .3])
```

```
STATUS:  Succeed


MESSAGE:  Parameter set successfully.
```

Change the robot position by setting a joint position and see how the controller responds.

```
% Move the robot shoulder lift joint to pi/4
[status,message] = gzjoint('set','ur10','shoulder_lift_joint','Axis','0','Angle',-pi/4);
```

Since Gazebo is now being stepped by Simulink, pausing the Simulink model also pauses the Gazebo simulation.

Next Steps

To learn more about the `gazeboCosimControl` model for controlling the robot in simulation, see the “Control Manipulator Robot with Co-Simulation in Simulink and Gazebo” on page 1-447 example.

Control Manipulator Robot with Co-Simulation in Simulink and Gazebo

Simulate control of a robotic manipulator using co-simulation between Simulink and Gazebo. The example uses Simulink™ to model the robot behavior, generate control commands, send these commands to Gazebo, and control the pace of the Gazebo simulation.

- The Gazebo Pacer block steps the Gazebo simulation at the same rate as Simulink, which enables commands to be executed accurately while simulating the physical dynamics in Gazebo.
- The Gazebo Read and Gazebo Apply Command blocks are used for communication between MATLAB® and Gazebo.
- The `gzlink`, `gzjoint`, and `gzworld` functions provide easy access to model parameters and queries.

This example reviews each of these components and their configurations in detail using a Simulink model that controls the positions of a Universal Robotics UR10 manipulator. The model uses inverse kinematics to relate desired end effector pose to joint positions, then applies a joint-space PD controller with dynamic compensation feedforward terms to govern the motion.

For more information about setting up the environment for this example, see “Configure Gazebo and Simulink for Co-simulation of a Manipulator Robot” on page 1-441.

Set Up Gazebo with Robot Model and Plugin

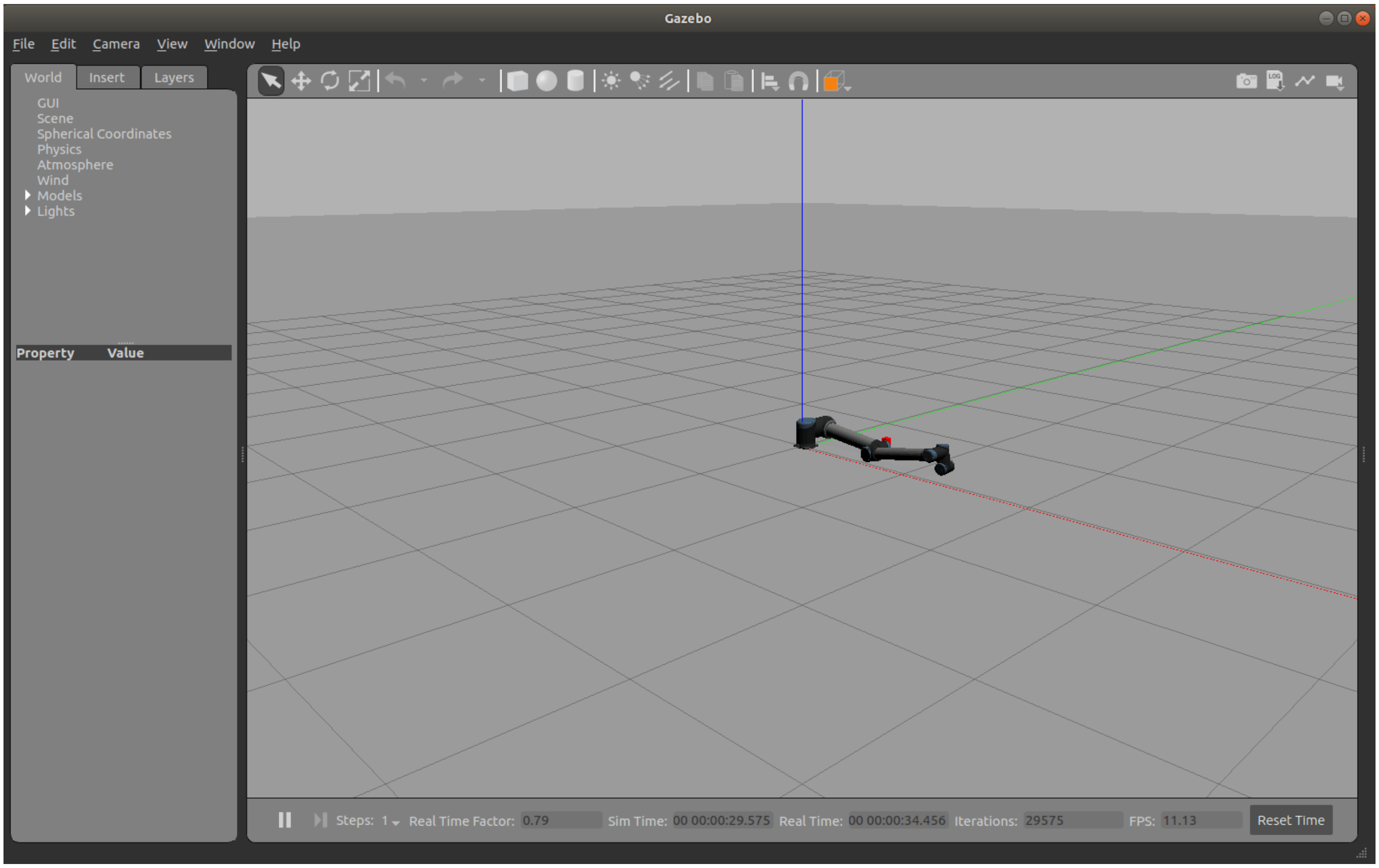
To setup the Gazebo world, install the necessary plugins, and test the connection with MATLAB and simulink, see the **Set Up Gazebo with Robot Model and Plugin** section of the “Configure Gazebo and Simulink for Co-simulation of a Manipulator Robot” on page 1-441 example.

Open World in Gazebo

Open the world by running these commands in the terminal of the Gazebo machine:

```
cd /home/user/src/GazeboPlugin/export
export SVGA_VGPU10=0
gazebo /home/user/worlds/Ur10BasicWithPlugin.world --verbose
```

Gazebo shows the robot and any other objects in the world. If the Gazebo simulator fails to open, you may need to reinstall the plugin. See **Install Gazebo Plugin Manually** in “Perform Co-Simulation between Simulink and Gazebo” on page 1-431.



Steps: 1 Real Time Factor: 0.79 Sim Time: 00 00:00:29.575 Real Time: 00 00:00:34.456 Iterations: 29575 FPS: 11.13

Connect to Gazebo

Next, initialize the Gazebo connection to MATLAB and Simulink. Specify the IP address and a port number of 14581, which is the default port for the Gazebo plugin.

```
ipGazebo = '192.168.116.162'; % Replace this with the IP of the Gazebo machine
gzinit(ipGazebo,14581);
```

Load the Robot Model

This model uses a universal UR10 robot created using loadrobot. The Gazebo model and robot models match since they are from the same source repository. For more information, see “Configure Gazebo and Simulink for Co-simulation of a Manipulator Robot” on page 1-441.

```
robot = loadrobot('universalUR10','Gravity',[0 0 -9.81],'DataFormat','column');
showdetails(robot)
```

```
--------------------
Robot: (10 bodies)

 Idx            Body Name                       Joint Name                      Join
 ---            ---------                       ----------                      ----
   1            base_link                      world_joint
   2                 base     base_link-base_fixed_joint
   3        shoulder_link              shoulder_pan_joint                         re
   4       upper_arm_link             shoulder_lift_joint                         re
   5         forearm_link                     elbow_joint                         re
```

```
    6        wrist_1_link                    wrist_1_joint                      rev
    7        wrist_2_link                    wrist_2_joint                      rev
    8        wrist_3_link                    wrist_3_joint                      rev
    9             ee_link                   ee_fixed_joint
   10               tool0       wrist_3_link-tool0_fixed_joint
--------------------
```

Set the initial configuration of the robot.

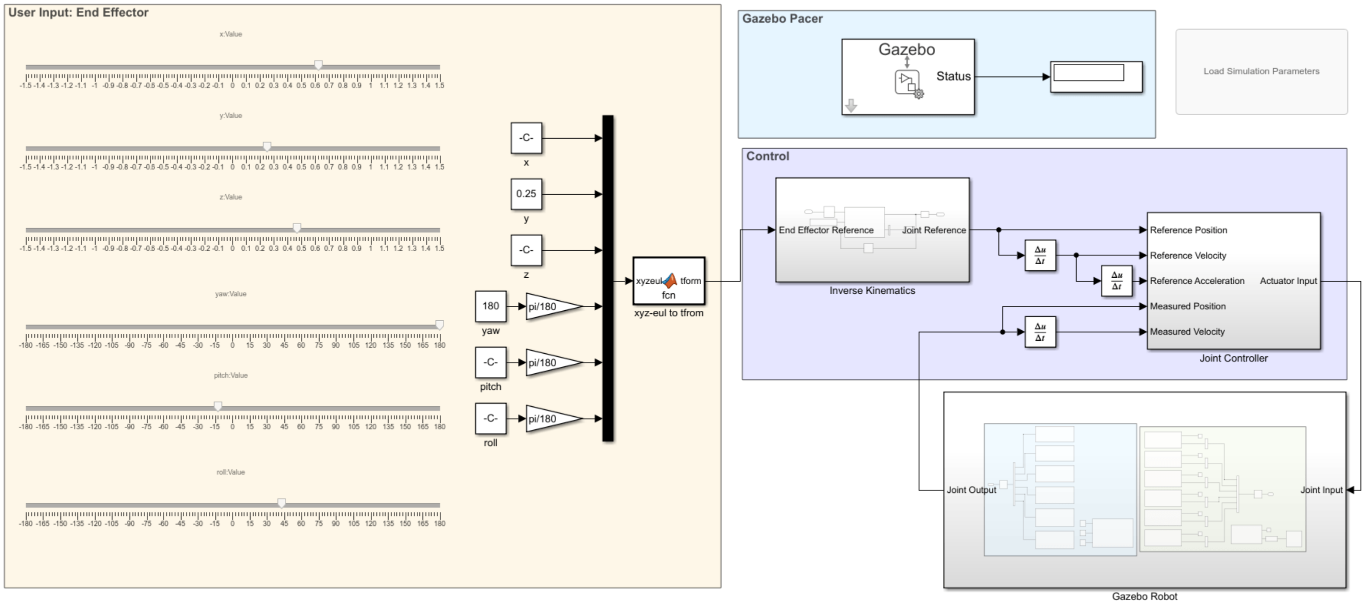
```
q0 = [0 -70 140 0 0 0]' * pi/180;
```

Load Simulink Model

The `gazeboCosimControl` model controls the end-effector position of the manipulator using the sliders in the **User Input: End Effector** section. The **Inverse Kinematics** subsystem generates a joint configuration that achieves the desired postion. Then, the **Joint Controller** subsystems generates torque forces for each joint to reach this position

Open the model.

```
open_system('gazeboCosimControl');
```



Set Controller and Trajectory Sample Times

```
Ts = 0.01;
Ts_trajectory = 0.05;
```

The sample time of the controller is short to ensure good performance. The sample time of trajectory is longer to ensure simulation speed, which is typical for high-level tasks like camera sensor, inverse kinematics, and trajectory generation.

Understand the model construction

The Simulink model consists of four areas:

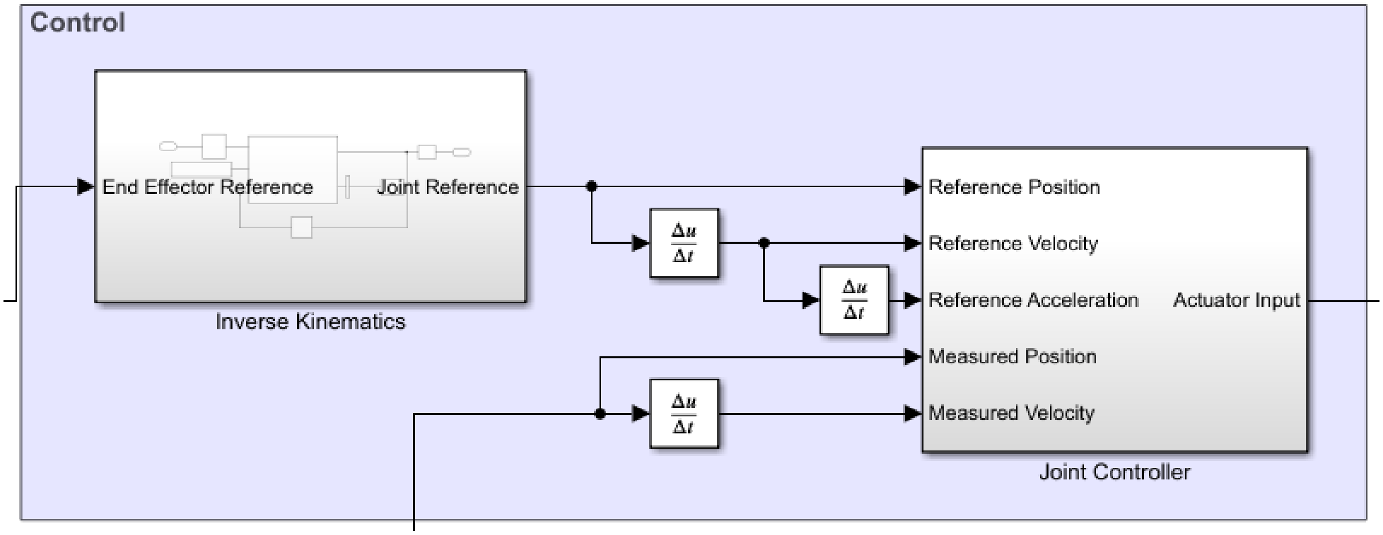
- **User input:** Provides inputs for the desired robot end effector position using sliders.
- **Control:** Maps the end effector position to joint position using inverse kinematics and controls joint position using a computed torque controller
- **Gazebo Pacer:** Maintains the connection to Gazebo to ensure that the Simulink model steps both simulations.
- **Gazebo Robot:** Sends commands to and receives commands from the Gazebo world.

These sections are outlined in greater detail below.

User Input

The user input has 6 sliders that control the end effector position: three to control the X, Y, and Z position in space, and three to control the orientation.

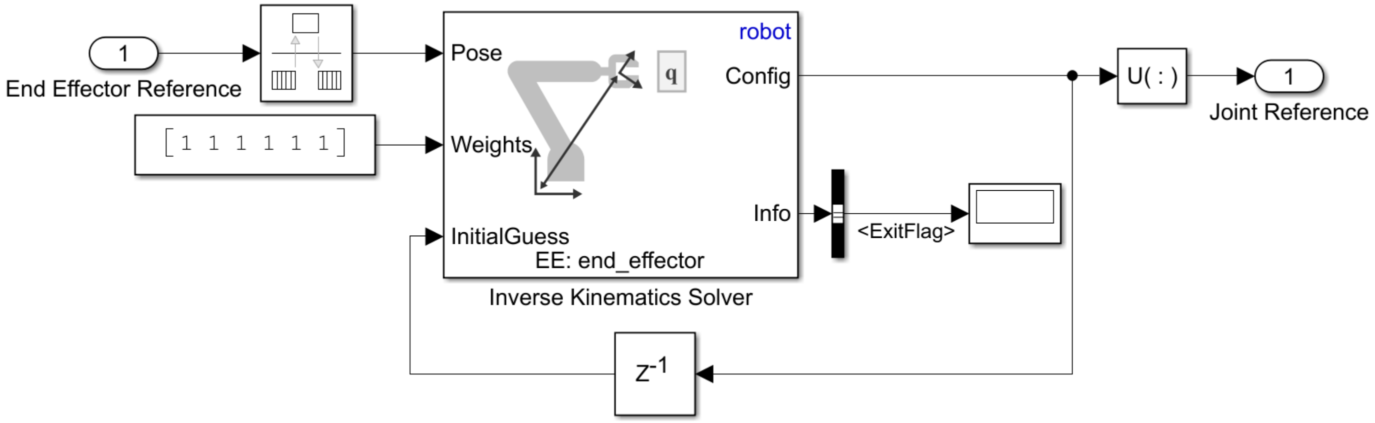
Control



The control section translates the desired end-effector pose to joint torques. First, the **Inverse Kinematics** subsystem computes joint positions that satisfy a target end-effector pose. Next, the **Joint Controller** subsystem produces actuator input torques given the joint reference position, velocity, and acceleration, and the current robot state. The state contains the measured position and velocity output from Gazebo.

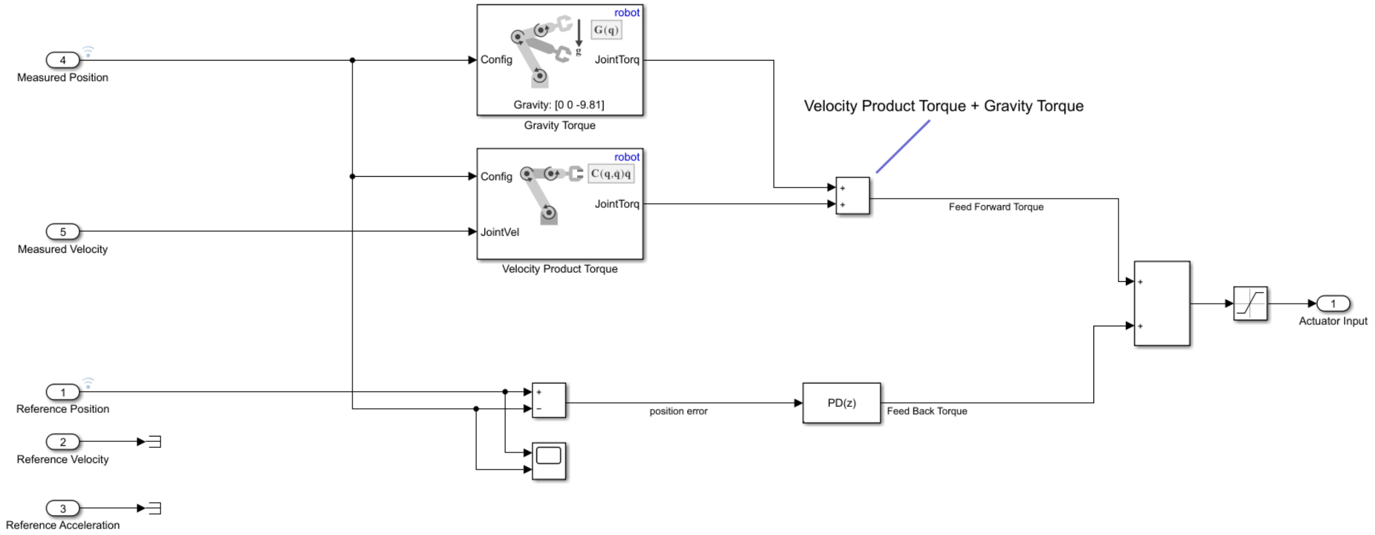
Inverse Kinematics Subsystem

Inverse kinematics calculates the joint position for the corresponding end effector pose (position and orientation) given an initial guess. For fast convergence, set the initial condition of the unit delay. In this subsystem, the weights are uniform, implying that the solution should place equal importance on achieving all positions and orientations in the desired pose.



Joint Controller subsystem

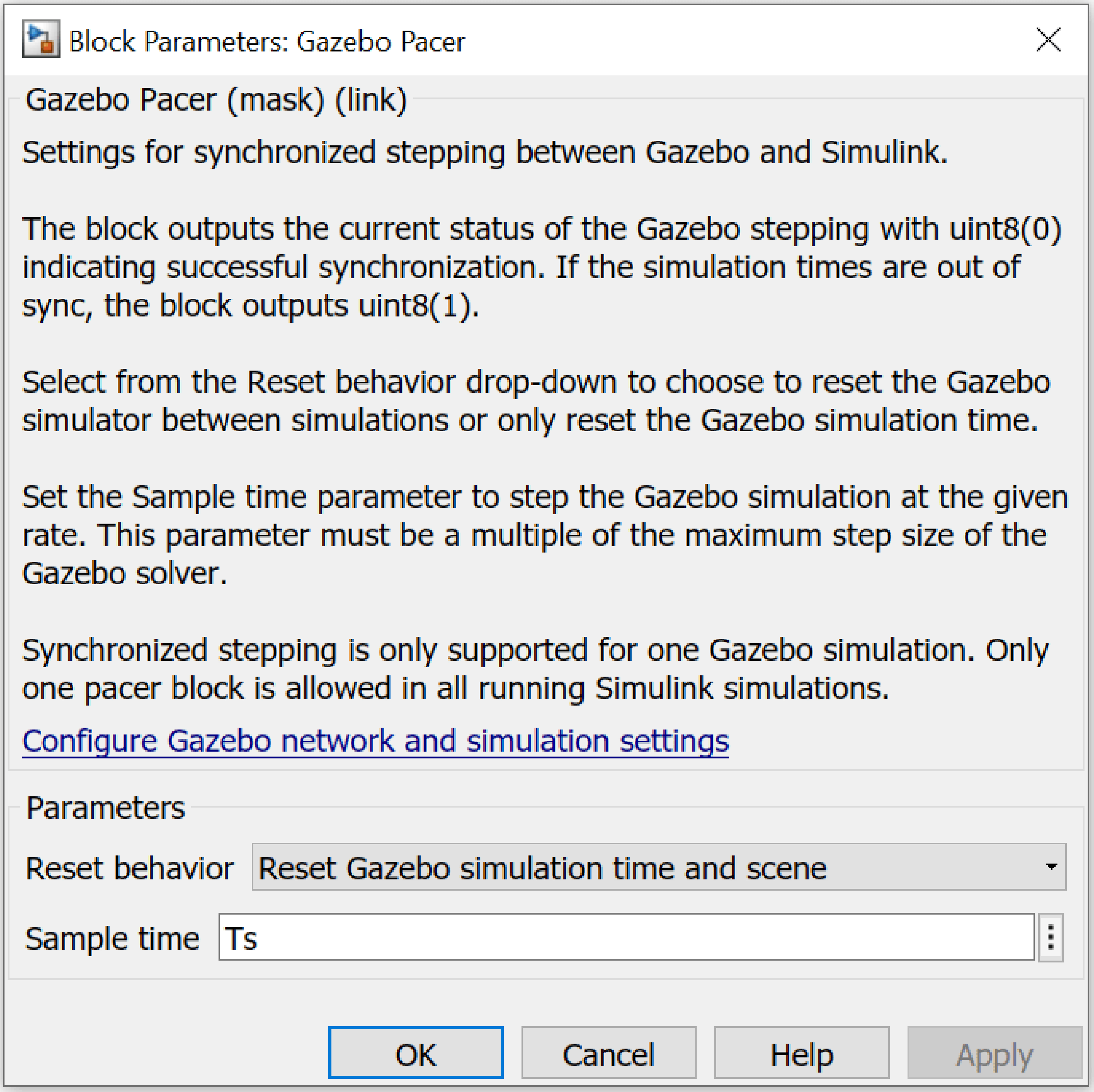
In this example, the joint controller uses feedback and a feedforward terms. Gravity torque and velocity product values combine to form the feedforward term. A proportional-derivative (PD) controller generates the feedback term. For more details about joint controllers, refer to “Perform Safe Trajectory Tracking Control Using Robotics Manipulator Blocks” on page 1-268.



Gazebo Pacer

The Gazebo Pacer block ensures co-simulation with Gazebo by stepping the Gazebo simulation in sync with Simulink steps. This is important for applications like this model, where a controller is used to directly govern the position of a robot in the Gazebo world. The joint measurements and torque commands must be synced to ensure the simulation reflects of the actual behavior.

In the block mask, specify the sample time as `Ts`. You can test your Gazebo connection using the **Configure Gazebo network and simulation settings** link. This connection was already set up using `gzinit`.



Block Parameters: Gazebo Pacer

Gazebo Pacer (mask) (link)

Settings for synchronized stepping between Gazebo and Simulink.

The block outputs the current status of the Gazebo stepping with uint8(0) indicating successful synchronization. If the simulation times are out of sync, the block outputs uint8(1).

Select from the Reset behavior drop-down to choose to reset the Gazebo simulator between simulations or only reset the Gazebo simulation time.

Set the Sample time parameter to step the Gazebo simulation at the given rate. This parameter must be a multiple of the maximum step size of the Gazebo solver.

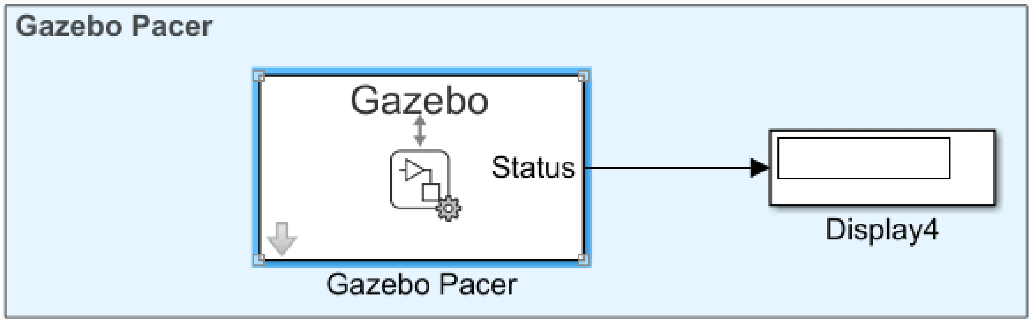
Synchronized stepping is only supported for one Gazebo simulation. Only one pacer block is allowed in all running Simulink simulations.

Configure Gazebo network and simulation settings

Parameters

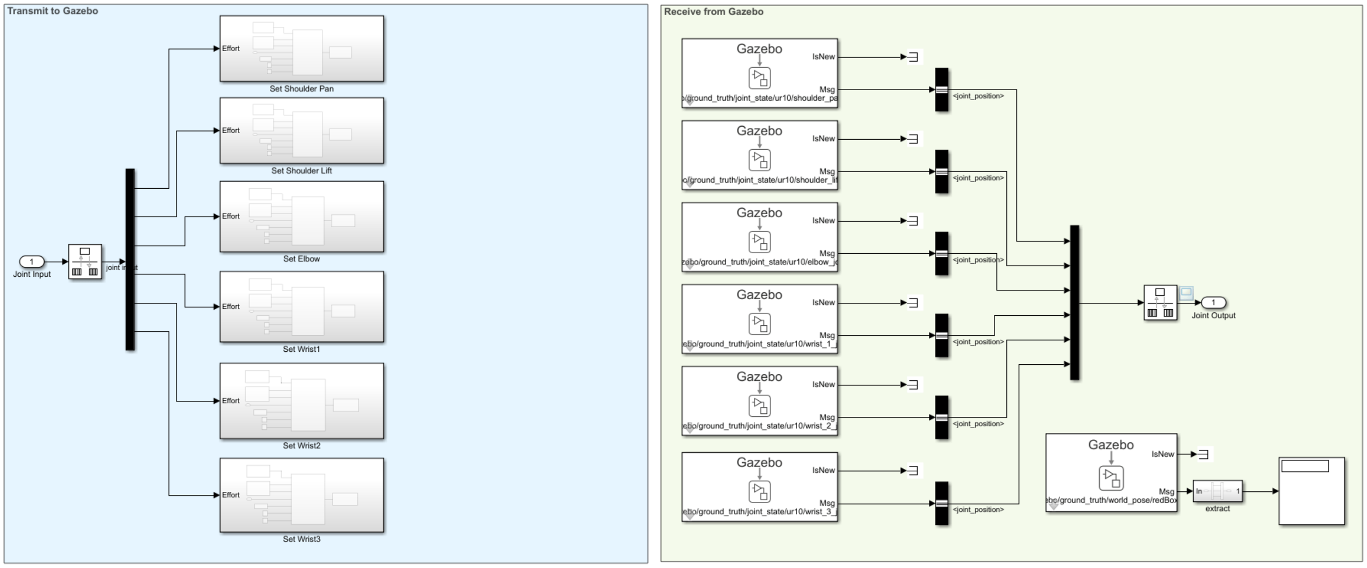
Reset behavior: Reset Gazebo simulation time and scene

Sample time: Ts



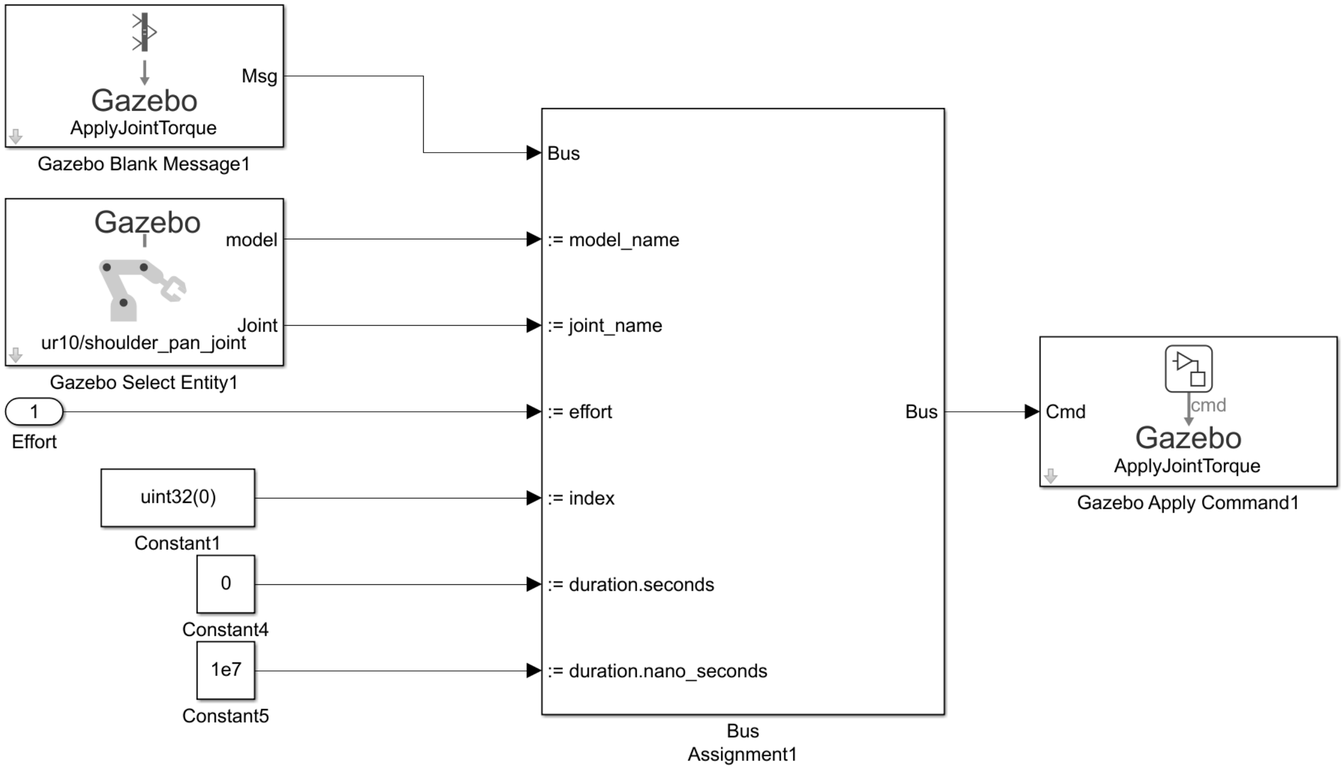
Gazebo Model

The Gazebo Model subsystem contains the blocks used to communicate with the Gazebo world. In this case, the controller applies a torque to each joint and reads back the joint position and velocity.



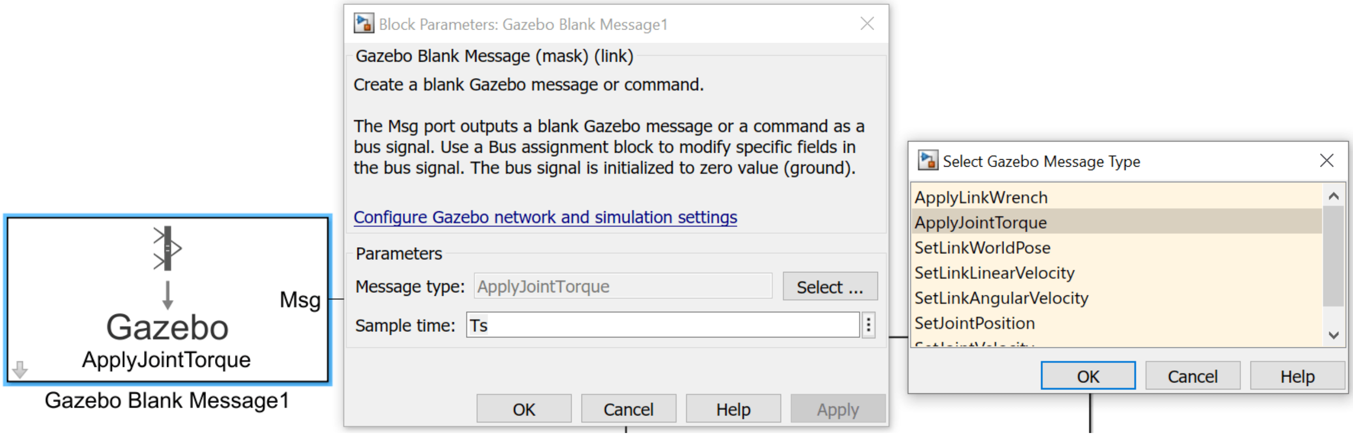
Apply Joint Torque

For each joint, the torque must be applied in the **Set <joint name>** subsystems of the Transmit to Gazebo area of the Gazebo model subsystem. For example, for the shoulder pan joint, the tools to send the shoulder plan subsystem are contained in the **Set Shoulder Pan** subsystem.



Each of these subsystems uses a Bus Assignment block whose inputs define the type of bus, the details of the model and duration, and the resultant command.

The **Gazebo Blank Message** block defines the bus type to be a Gazebo message format of the appropriate type. In the block parameters dialog, select "ApplyJointTorque" from the list of command types. The block is required to ensure that the correct bus template is used, i.e. that the port is the **Bus Assignment** block have the correct values.


Block Parameters: Gazebo Blank Message1

Gazebo Blank Message (mask) (link)

Create a blank Gazebo message or command.

The Msg port outputs a blank Gazebo message or a command as a bus signal. Use a Bus assignment block to modify specific fields in the bus signal. The bus signal is initialized to zero value (ground).

Configure Gazebo network and simulation settings

Parameters

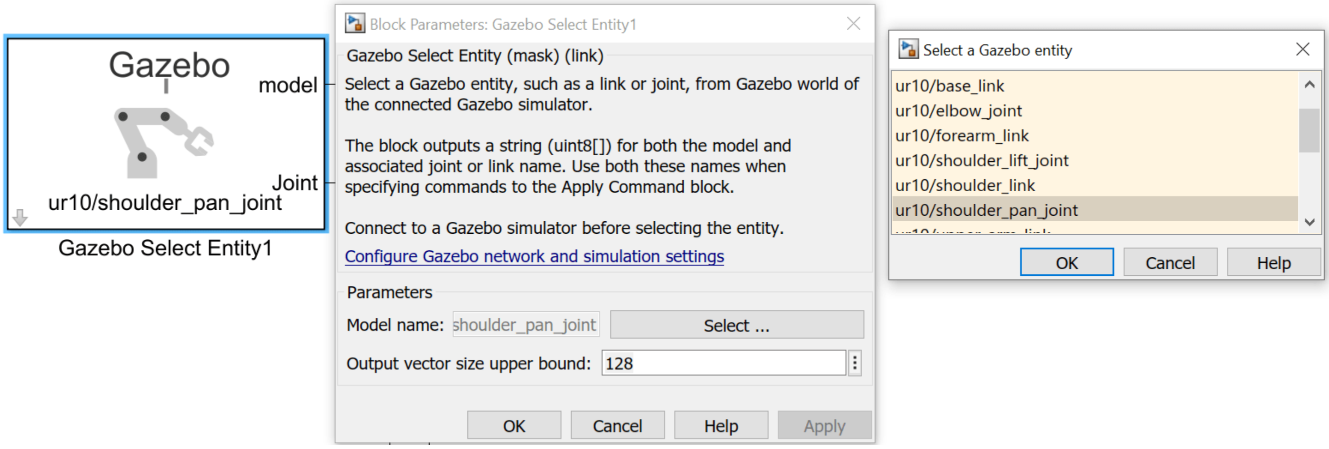
Message type: ApplyJointTorque

Sample time: Ts

Select Gazebo Message Type

ApplyLinkWrench
ApplyJointTorque
SetLinkWorldPose
SetLinkLinearVelocity
SetLinkAngularVelocity
SetJointPosition

The **Gazebo Select Entity** block selects the object to which this message will be applied. In this case, since a torque is being applied, the "entity" is a *joint*. Since this subsystem will apply torque to the shoulder joint, select the corresponding joint in the Gazebo world from the list of entity types. As long as a connection to the Gazebo world has been established, this list will be automatically populated. If the connection is lost, click the "Configure Gazebo network and simulation settings" link to reestablish, or use the `gzinit` command line interface. The Gazebo world must be open for this operation to succeed.



Gazebo Select Entity (mask) (link)

Select a Gazebo entity, such as a link or joint, from Gazebo world of the connected Gazebo simulator.

The block outputs a string (uint8[]) for both the model and associated joint or link name. Use both these names when specifying commands to the Apply Command block.

Connect to a Gazebo simulator before selecting the entity.

Configure Gazebo network and simulation settings

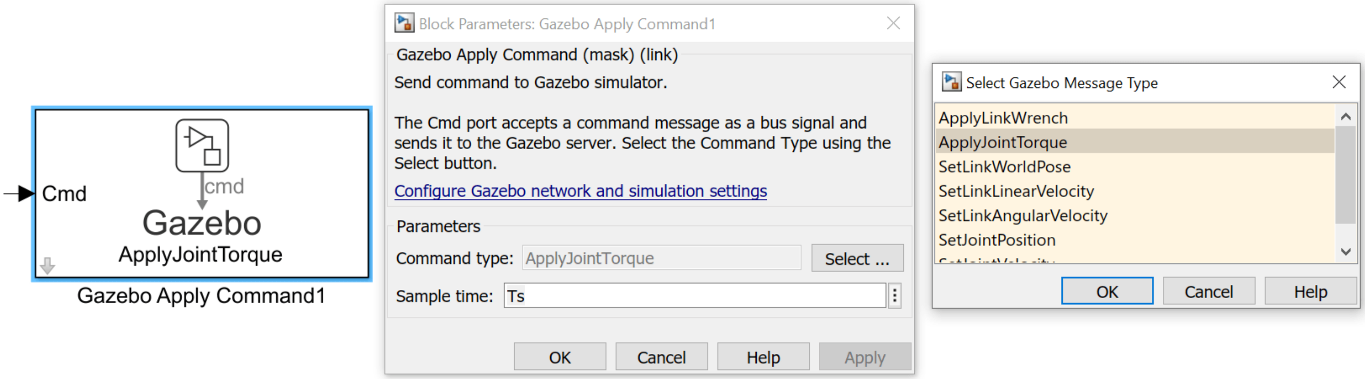
Parameters

Model name: shoulder_pan_joint

Output vector size upper bound: 128

ur10/base_link
ur10/elbow_joint
ur10/forearm_link
ur10/shoulder_lift_joint
ur10/shoulder_link
ur10/shoulder_pan_joint

The Gazebo Apply Command block takes the contents of the message sends it to the Gazebo server. Open the block dialog parameters, select the Command type parameter, and select ApplyJointTorque to send the corresponding command to Gazebo.



Gazebo Apply Command (mask) (link)

Send command to Gazebo simulator.

The Cmd port accepts a command message as a bus signal and sends it to the Gazebo server. Select the Command Type using the Select button.

Configure Gazebo network and simulation settings

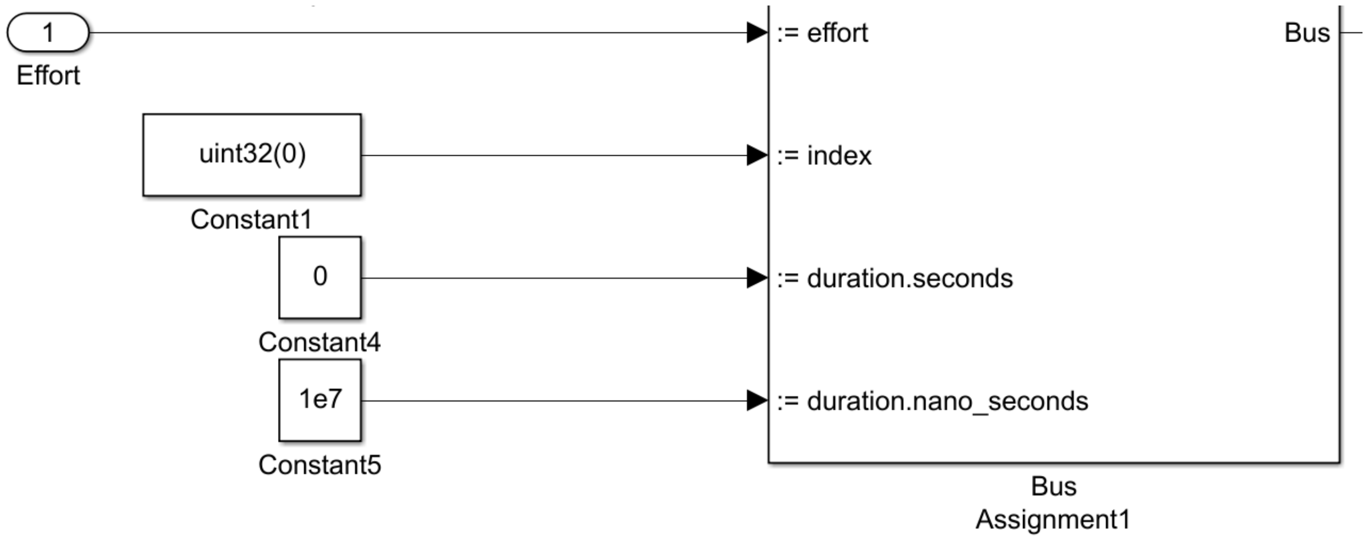
Parameters

Command type: ApplyJointTorque

Sample time: Ts

ApplyLinkWrench
ApplyJointTorque
SetLinkWorldPose
SetLinkLinearVelocity
SetLinkAngularVelocity
SetJointPosition

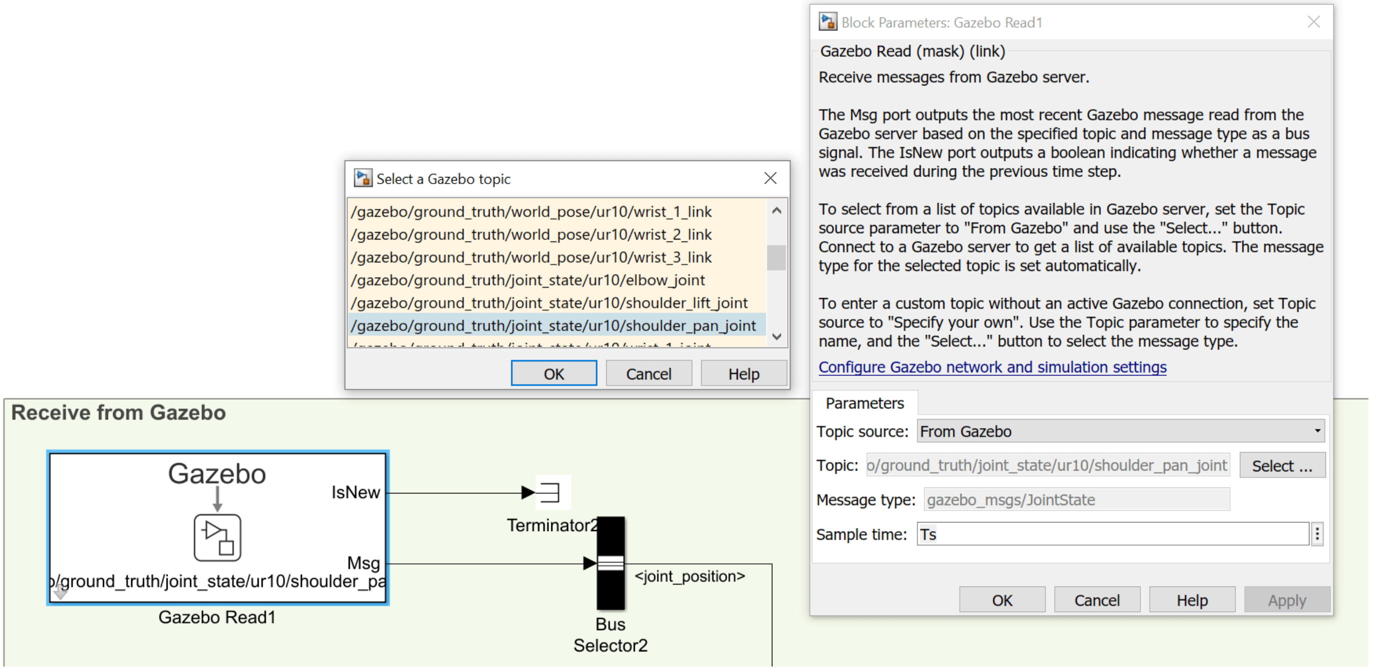
The remaining inputs govern the amount and duration of the applied force. The **effort** inport specifies the joint torque quantity. The **index** input dictates the axis to which the torque should be applied. Since each of the revolute joints is just 1 degree of freedom, this value should be set to `uint32(0)` to indicate the first axis. Finally, the duration inputs specify the duration of the applied torque, divided into seconds and nanoseconds. For example, if a duration is 1.005 seconds, it would be 1 second and 5,000,000 nanoseconds as a bus. They are separated out to increase precision. In this case, the controller applies the torque for the length of the sample time, previously specified to 0.01 seconds, or 1e7 nanoseconds. Therefore the first input is zero (0 seconds) and the second input is 1e7.



The **Gazebo Model** subsystem includes 5 other subsystems corresponding to the other five joints, which have been configured in the same fashion.

Measure Joint State

The second part of the Gazebo Model interface requires reading values from Gazebo using the **Gazebo Read** block for each joint. For example, the block for the shoulder pan joint is shown below.



Select a Gazebo topic

/gazebo/ground_truth/world_pose/ur10/wrist_1_link
/gazebo/ground_truth/world_pose/ur10/wrist_2_link
/gazebo/ground_truth/world_pose/ur10/wrist_3_link
/gazebo/ground_truth/joint_state/ur10/elbow_joint
/gazebo/ground_truth/joint_state/ur10/shoulder_lift_joint
/gazebo/ground_truth/joint_state/ur10/shoulder_pan_joint

Gazebo Read (mask) (link)

Receive messages from Gazebo server.

The Msg port outputs the most recent Gazebo message read from the Gazebo server based on the specified topic and message type as a bus signal. The IsNew port outputs a boolean indicating whether a message was received during the previous time step.

To select from a list of topics available in Gazebo server, set the Topic source parameter to "From Gazebo" and use the "Select..." button. Connect to a Gazebo server to get a list of available topics. The message type for the selected topic is set automatically.

To enter a custom topic without an active Gazebo connection, set Topic source to "Specify your own". Use the Topic parameter to specify the name, and the "Select..." button to select the message type.

Configure Gazebo network and simulation settings

Topic source: From Gazebo
Topic: o/ground_truth/joint_state/ur10/shoulder_pan_joint
Message type: gazebo_msgs/JointState
Sample time: Ts

The **Gazebo Read** block reads messages from the Gazebo server. In the block dialog, click "Select.." next to the topic to choose the right topic to read from, and select the corresponding measured joint value, `/gazebo/ground_truth/joint_state/ur10/shoulder_pan_joint`. This message selects the precise ground truth value, though it is also possible to place sensors in Gazebo and read from those instead.

Use the **Bus Selector** to only choose the relevant values from the message. In this case, only the position is measured from the Gazebo model, which is ultimately fed back to the controller.

This section also reads back the value of the red box in Gazebo. While it is possible to also set the position of this box using a format similar to that used above, it is simpler to use the gzlink command line interface to update the block at discrete instants during the model execution.

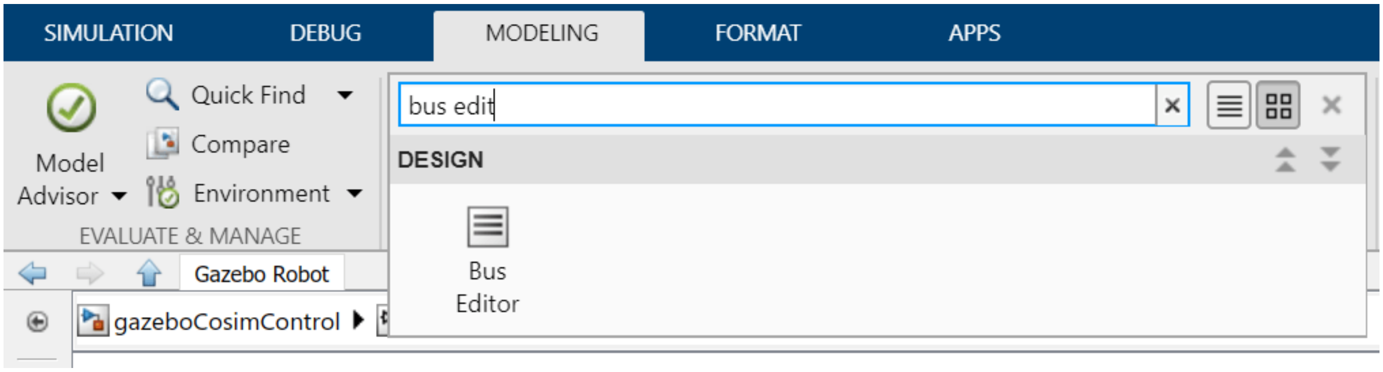
Customize Bus Objects for Simulation

Gazebo bus objects are generated when the cosimulation blocks are introduced in a model. Signals from Gazebo are read through these bus objects. By default, dimensions are set to variable size since joints can have multiple degrees of freedom. Since all joints in this example are revolute joints, they are 1-dimensional and fixed in size, which can be specified as a property of the bus signals**.** This step is optional, but it is useful to overcome limitations of *variable-size signals*.

Load the custom bus objects.

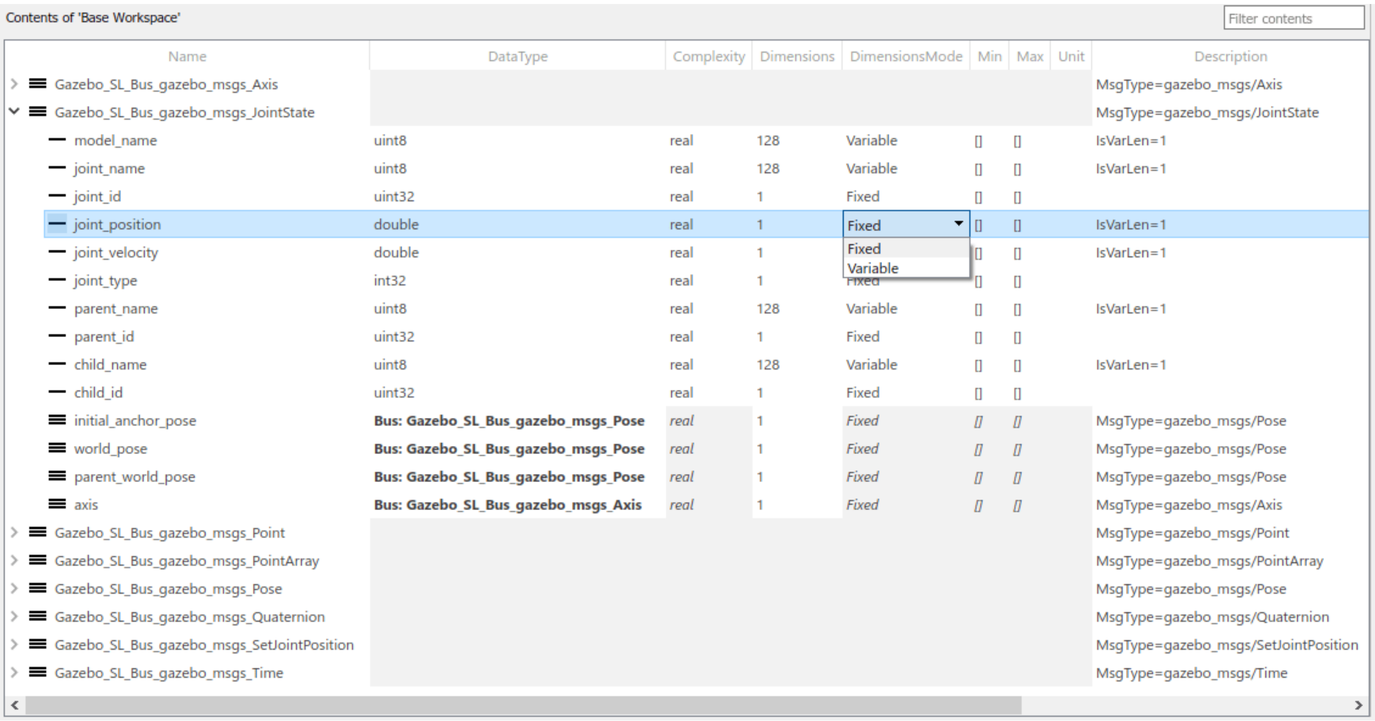
```
load('custom_busobjects_basic');
```

To interactively load the bus objects, open the **Bus Editor** in Simulink. In the Simulink Toolstrip, on the **Modeling** tab, in the **Design** gallery, click **Bus Editor**.



In the Bus Editor toolstrip, select **Import > MAT File**. Then, select the MAT file to load.

In the Bus Editor table, select the bus object used to read joint state, then select the corresponding element name. For the bus object named `Gazebo_SL_Bus_gazebo_msgs_JointState`, select the elements named `joint_position` and `joint_velocity` and change **Dimension** to `1` and **DimensionMode** to `Fixed`.



Contents of 'Base Workspace'

Name	DataType	Complexity	Dimensions	DimensionsMode	Min	Max	Unit	Description
Gazebo_SL_Bus_gazebo_msgs_Axis								MsgType=gazebo_msgs/Axis
Gazebo_SL_Bus_gazebo_msgs_JointState								MsgType=gazebo_msgs/JointState
model_name	uint8	real	128	Variable	[]	[]		IsVarLen=1
joint_name	uint8	real	128	Variable	[]	[]		IsVarLen=1
joint_id	uint32	real	1	Fixed	[]	[]		
joint_position	double	real	1	Fixed	[]	[]		IsVarLen=1
joint_velocity	double	real	1		[]	[]		IsVarLen=1
joint_type	int32	real	1	Fixed	[]	[]		
parent_name	uint8	real	128	Variable	[]	[]		IsVarLen=1
parent_id	uint32	real	1	Fixed	[]	[]		
child_name	uint8	real	128	Variable	[]	[]		IsVarLen=1
child_id	uint32	real	1	Fixed	[]	[]		
initial_anchor_pose	**Bus: Gazebo_SL_Bus_gazebo_msgs_Pose**	*real*	1	*Fixed*	*[]*	*[]*		MsgType=gazebo_msgs/Pose
world_pose	**Bus: Gazebo_SL_Bus_gazebo_msgs_Pose**	*real*	1	*Fixed*	*[]*	*[]*		MsgType=gazebo_msgs/Pose
parent_world_pose	**Bus: Gazebo_SL_Bus_gazebo_msgs_Pose**	*real*	1	*Fixed*	*[]*	*[]*		MsgType=gazebo_msgs/Pose
axis	**Bus: Gazebo_SL_Bus_gazebo_msgs_Axis**	*real*	1	*Fixed*	*[]*	*[]*		MsgType=gazebo_msgs/Axis
Gazebo_SL_Bus_gazebo_msgs_Point								MsgType=gazebo_msgs/Point
Gazebo_SL_Bus_gazebo_msgs_PointArray								MsgType=gazebo_msgs/PointArray
Gazebo_SL_Bus_gazebo_msgs_Pose								MsgType=gazebo_msgs/Pose
Gazebo_SL_Bus_gazebo_msgs_Quaternion								MsgType=gazebo_msgs/Quaternion
Gazebo_SL_Bus_gazebo_msgs_SetJointPosition								MsgType=gazebo_msgs/SetJointPosition
Gazebo_SL_Bus_gazebo_msgs_Time								MsgType=gazebo_msgs/Time

Run the Simulation

Reset the Gazebo world and box position before simulating using MATLAB commands. Commands like these can be more directly incorporated into the Simulink model as a `StartFcn` callback to ensure that they are executed at each Simulation run. In the **Gazebo Pacer** block parameters, only the simulation time is reset when you run the model unless you change the **Reset Behavior** drop down.

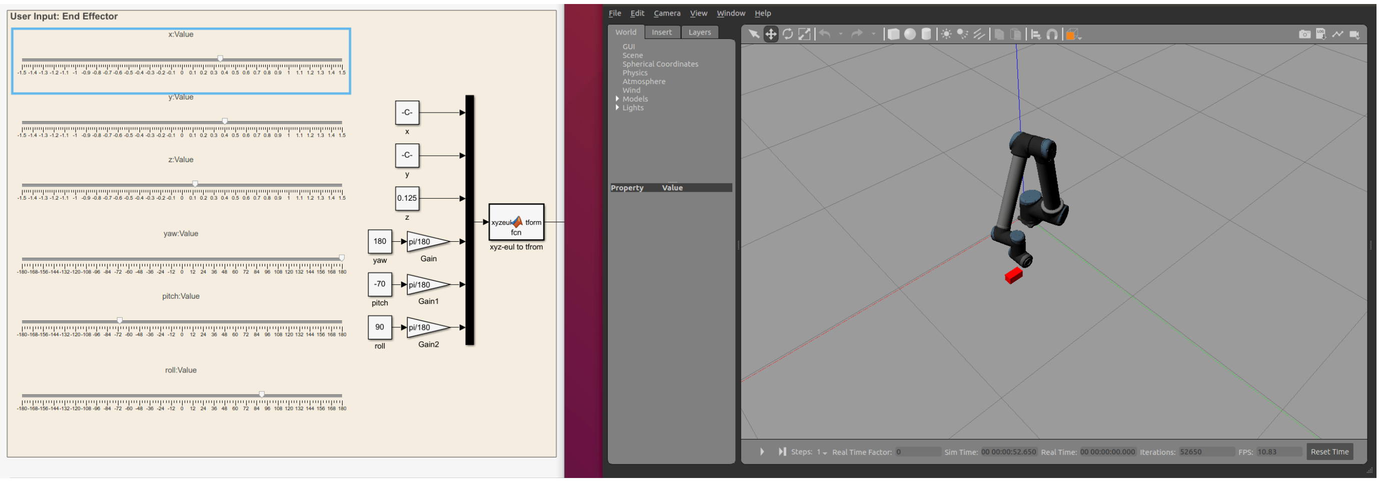
```
gzworld("reset"); % Reset the world to its initial state
gzlink("set","redBox","link","Position",[0.5 -0.4 .3]); % Move the box to a new location
```

```
STATUS:  Succeed


MESSAGE:  Parameter set successfully.
```

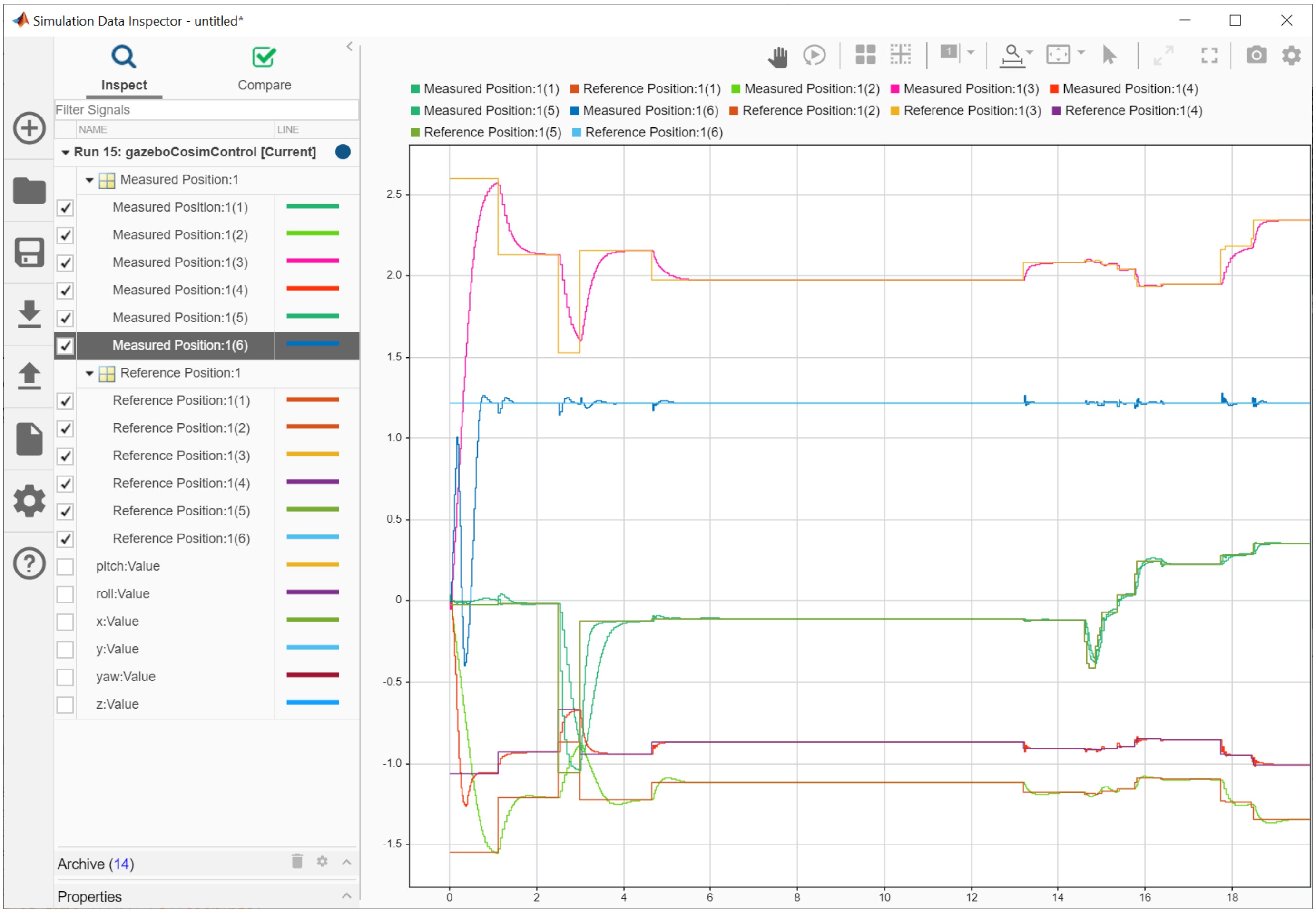
You can also run these types of commands during simulation. To do so, use the green "run" button to simulate the Simulink model rather than the **sim** command to ensure that the command line can be executed while the simulation is running.

Run the simulation for 20 seconds and test different poses using the sliders in **User Input: End Effector**. Verify the inverse kinematics and joint controller are behaving properly in Gazebo.

```
simoutput = sim('gazeboCosimControl','StopTime','20');
```



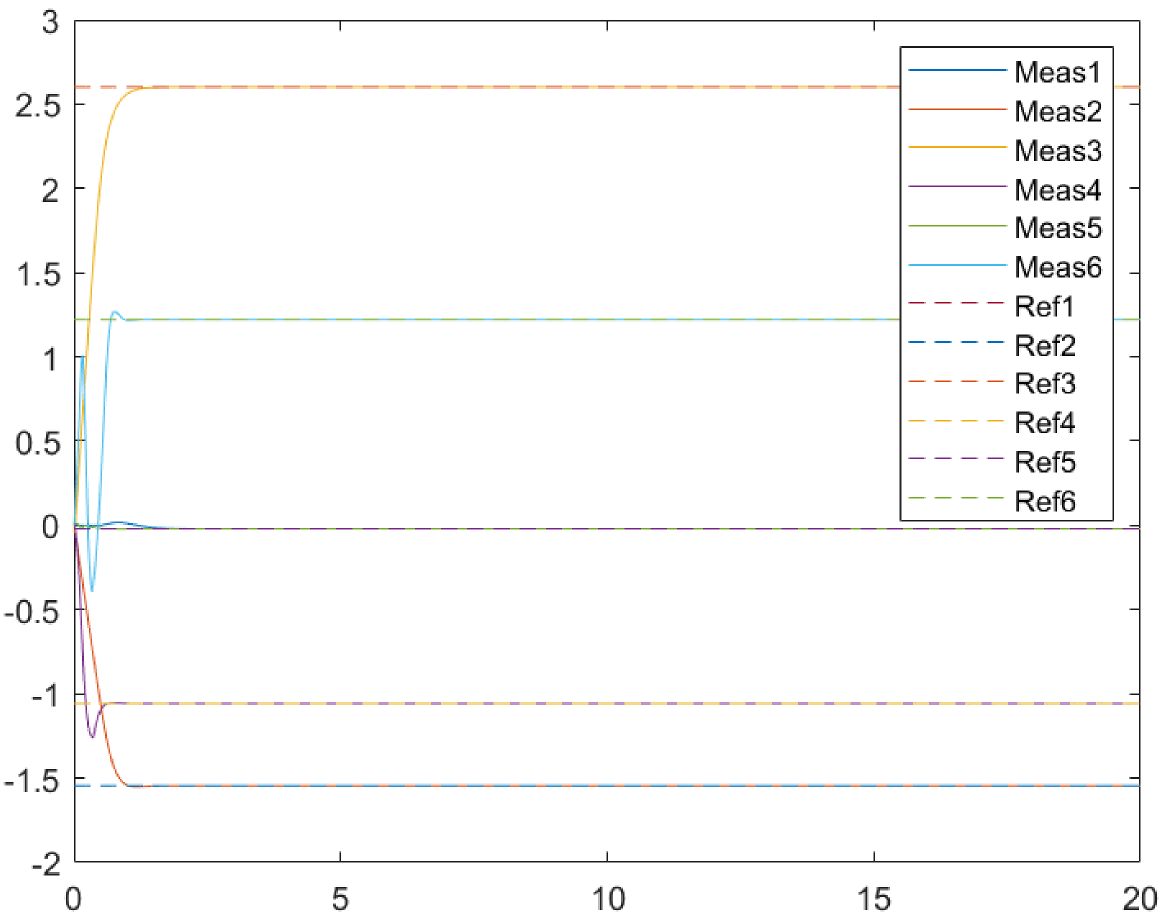
Use the **Simulation Data Inspector** to see behavior of the joint postions. This image shows data when the manipulator command positions have been changed several times over the course of the simulation.



View Performance

Joint measurements and references are logged. Once the simulation is complete, plot the logged outputs.

```
measuredPosition = simoutput.logsout{1}.Values;
referencePosition = simoutput.logsout{2}.Values;
figure
plot(measuredPosition.Time, measuredPosition.Data, '-', referencePosition.Time, referencePositio
legend({'Meas1','Meas2','Meas3','Meas4','Meas5','Meas6','Ref1','Ref2','Ref3','Ref4','Ref5','Ref6
```



Plan Minimum Jerk Trajectory for Robot Arm

This example shows how to plan a minimum jerk polynomial trajectory for a robotic manipulator. The example shows how to load an included robot model, plan a path for the robot model in an environment with obstacles, generate a minimum jerk trajectory from the path, and visualize the generated trajectories and the robot motion.

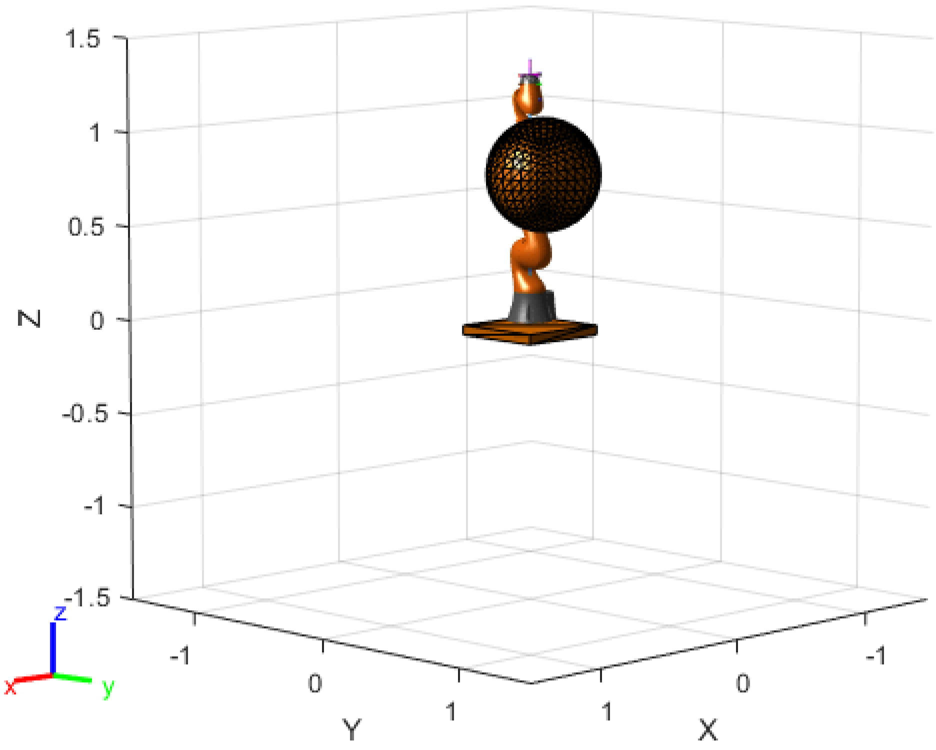
Set Up Robot Model and Environment

This example uses a model of the KUKA LBR iiwa, a 7 degree-of-freedom robot manipulator. Use `loadrobot` to load the robot model into the workspace as a `rigidBodyTree` object. Set the output format for configurations to `"row"`.

```
robot = loadrobot("kukaIiwa14","DataFormat","row");
```

Generate the environment for the robot. Create collision objects and specify their poses relative to the robot base. Visualize the environment.

```
env = {collisionBox(0.5,0.5,0.05) collisionSphere(0.3)};
env{1}.Pose(3,end) = -0.05;
env{2}.Pose(1:3,end) = [0.1 0.2 0.8];

show(robot);
hold on
show(env{1})
show(env{2})
```



Plan Path Using manipulatorRRT

Create the RRT planner for the robot model using manipulatorRRT. Set the ValidationDistance property to increase the number of intermediate states while interpolating the path.

```
rrt = manipulatorRRT(robot,env);
rrt.ValidationDistance = 0.2;
```

Specify a start and a goal configuration.

```
startConfig = [0.08 -0.65 0.05 0.02 0.04 0.49 0.04];
goalConfig =  [2.97 -1.05 0.05 0.02 0.04 0.49 0.04];
```

Plan the path. Due to the randomness of the RRT algorithm, set the rng seed for repeatability.

```
rng(0)
path = plan(rrt,startConfig,goalConfig);
```

Interpolate the path and retrieve the waypoints.

```
interpPath = interpolate(rrt,path);
wpts = interpPath';
```

Generate Minimum Jerk Polynomial Trajectory

The planner returns a path as an ordered set of waypoints. To pass these to a robot, you must first determine a trajectory through them. The minjerkpolytraj function creates a smooth trajectory with minimum jerk that hits all the specified waypoints.

Provide an initial guess for the times at which the robot arm arrives at the waypoints.

```
initialGuess = linspace(0,size(wpts,2)*0.2,size(wpts,2));
```

Specify the number of samples to take while estimating the trajectory.

```
numSamples = 100;
```

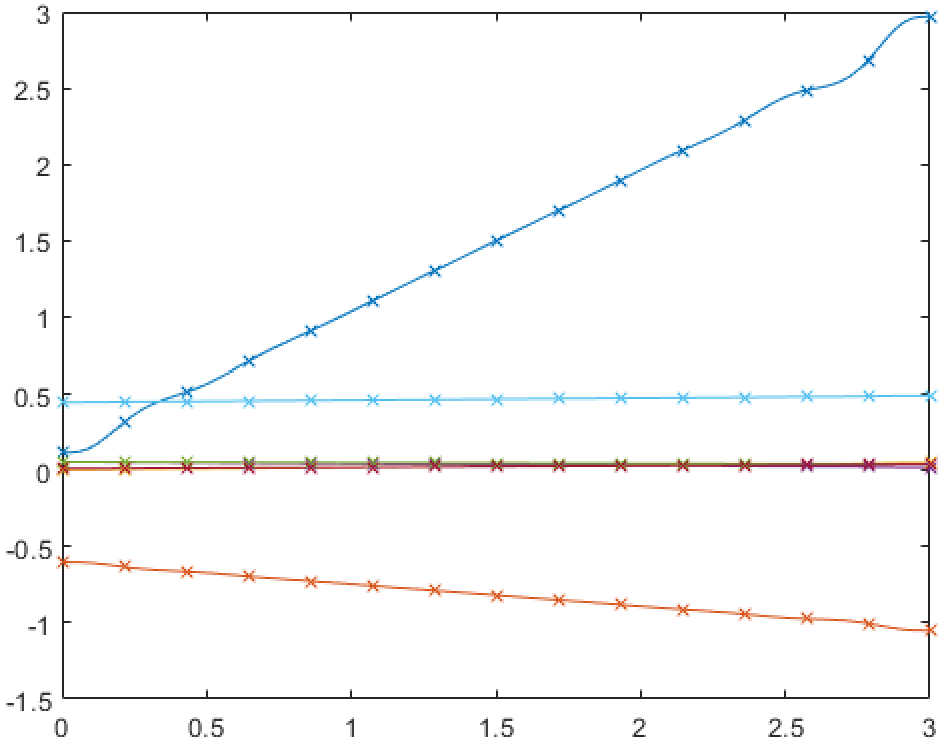
Compute the minimum jerk polynomial trajectory.

```
[q,qd,qdd,qddd,pp,tpts,tSamples] = minjerkpolytraj(wpts,initialGuess,numSamples);
```

Visualize Trajectories and Waypoints

Plot the trajectories and the waypoints over time.

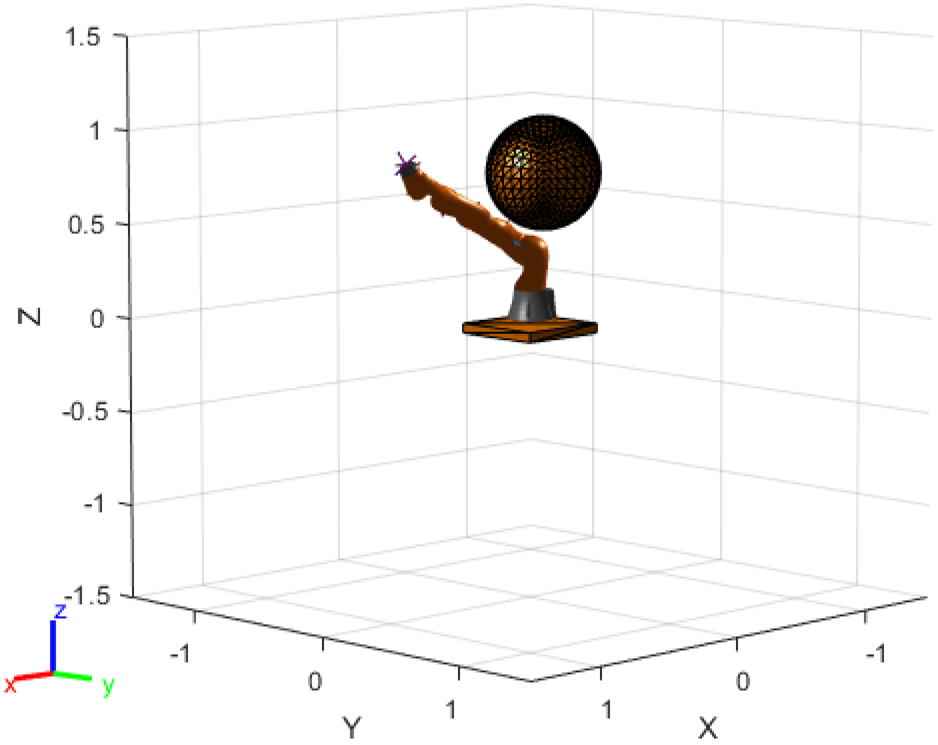
```
minJerkPath = q';
figure
plot(tSamples,q)
hold all
plot(tpts,wpts,"x")
```



Visualize Robot Motion

Use the `show` object function to animate the resulting motion. This visualization enables fast updates to ensure a smooth animation.

```
figure;
ax = show(robot,startConfig);
hold all

% Ensure the figure pops out of the Live Editor so animations are visible
set(gcf,"Visible","on");
for i = 1:length(env)
    show(env{i},"Parent",ax);
end

for i = 1:size(minJerkPath,1)
    show(robot,minJerkPath(i,:),"PreservePlot",false,"FastUpdate",true);
    drawnow;
end

hold off
```


1.5
1
0.5
0
-0.5
-1
-1.5
Z
-1
0
1
Y
1
0
-1
X
z
x
y

Design and Simulate Warehouse Pick-and-Place Application Using Mobile Manipulator in Simulink and Gazebo

This example shows how to set up an end-to-end, pick-and-place workflow for a mobile manipulator like the KINOVA® Gen3 on a Husky® mobile robot, in Simulink®.

Overview

This example simulates a mobile manipulator identifying and recycling objects into two bins using tools from these toolboxes:

- **Robotics System Toolbox™** — Models and simulates the manipulator.
- **Stateflow®** — Schedules the high-level tasks in the example and step from task to task.
- **ROS Toolbox™** — Connects MATLAB® and Simulink to Gazebo®.
- **Computer Vision Toolbox™** and **Deep Learning Toolbox™** — Perform object detection using a simulated camera in Gazebo.

This example builds on key concepts from the following related examples:

- “Plan and Execute Task- and Joint-Space Trajectories Using KINOVA Gen3 Manipulator” on page 1-257 — Shows how to generate and simulate interpolated joint trajectories to move from an initial to a desired end-effector pose.
- “Pick-and-Place Workflow in Gazebo Using ROS” on page 1-302 — Shows how to setup 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.
- “Train YOLO v2 Network for Vehicle Detection” (Computer Vision Toolbox) — Shows how to use a YOLO v2 object detector.
- “Get Started with Gazebo and Simulated TurtleBot” (ROS Toolbox) — Shows how to set up the connection between MATLAB and Gazebo.

Robot Simulation and Control in Gazebo

This example uses Simulink to control two robots in Gazebo. Using ROS as the primary communication mechanism, enables you to use the allows the official Husky and KINOVA ROS packages for low-level motion control and sensing. This facilitates a straightforward transition from simulation to hardware, as the ROS commands stay the same.

The supplied Gazebo world also uses the Gazebo co-simulation plug-in, which enables direct MATLAB communication for tools such as querying the Gazebo world or directly setting the states of links in Gazebo.

Simulated sensors

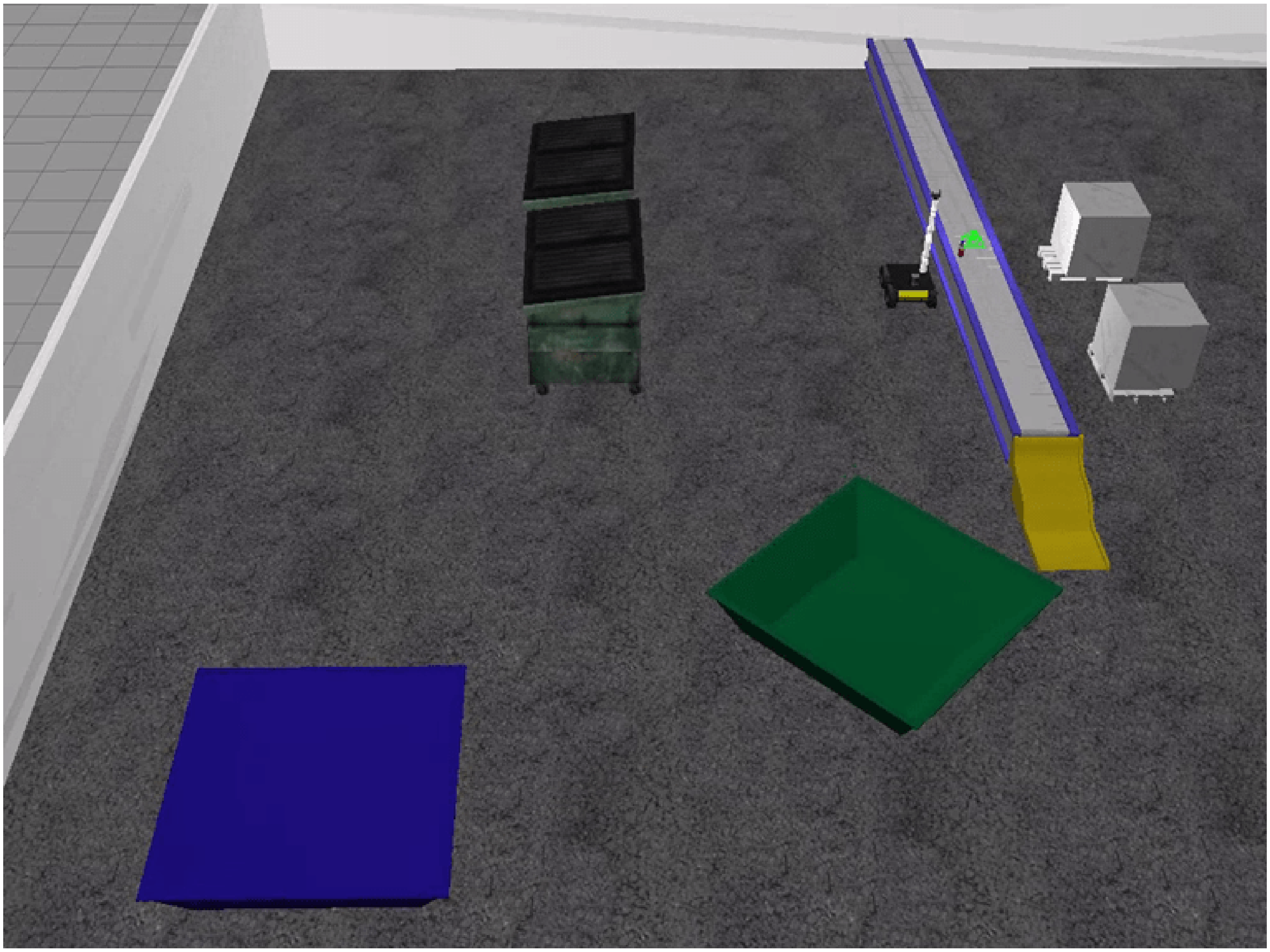
Gazebo simulates these sensors:

- RGB-D camera attached to the end-effector of the robot manipulator. The model uses this camera feed to detect objects to pick up.
- 2-D lidar sensor at the front of the mobile robot base. The model uses this laser feed to localize the robot in a precomputed map.

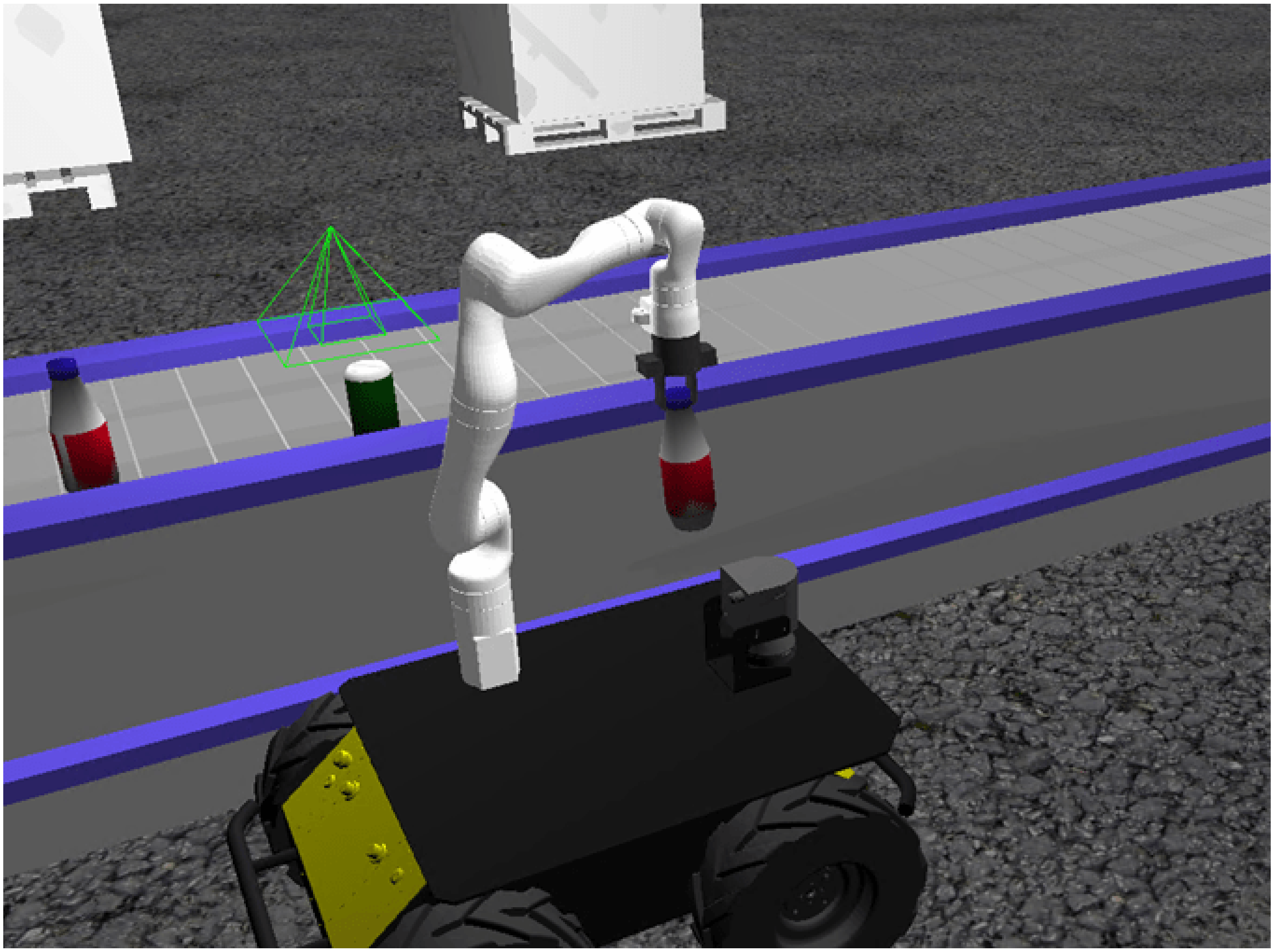
Pick-and-Place Workflow Using Simulink and Stateflow

In this example, the mobile manipulator operates in a simulated warehouse recycling facility, picking up recyclable objects from a central conveyer belt and transferring them to the corresponding recycling stations one by one.

- The simulated recycling facility



- The simulated mobile manipulator that is a Kinova Gen3 manipulator on a Husky mobile robot

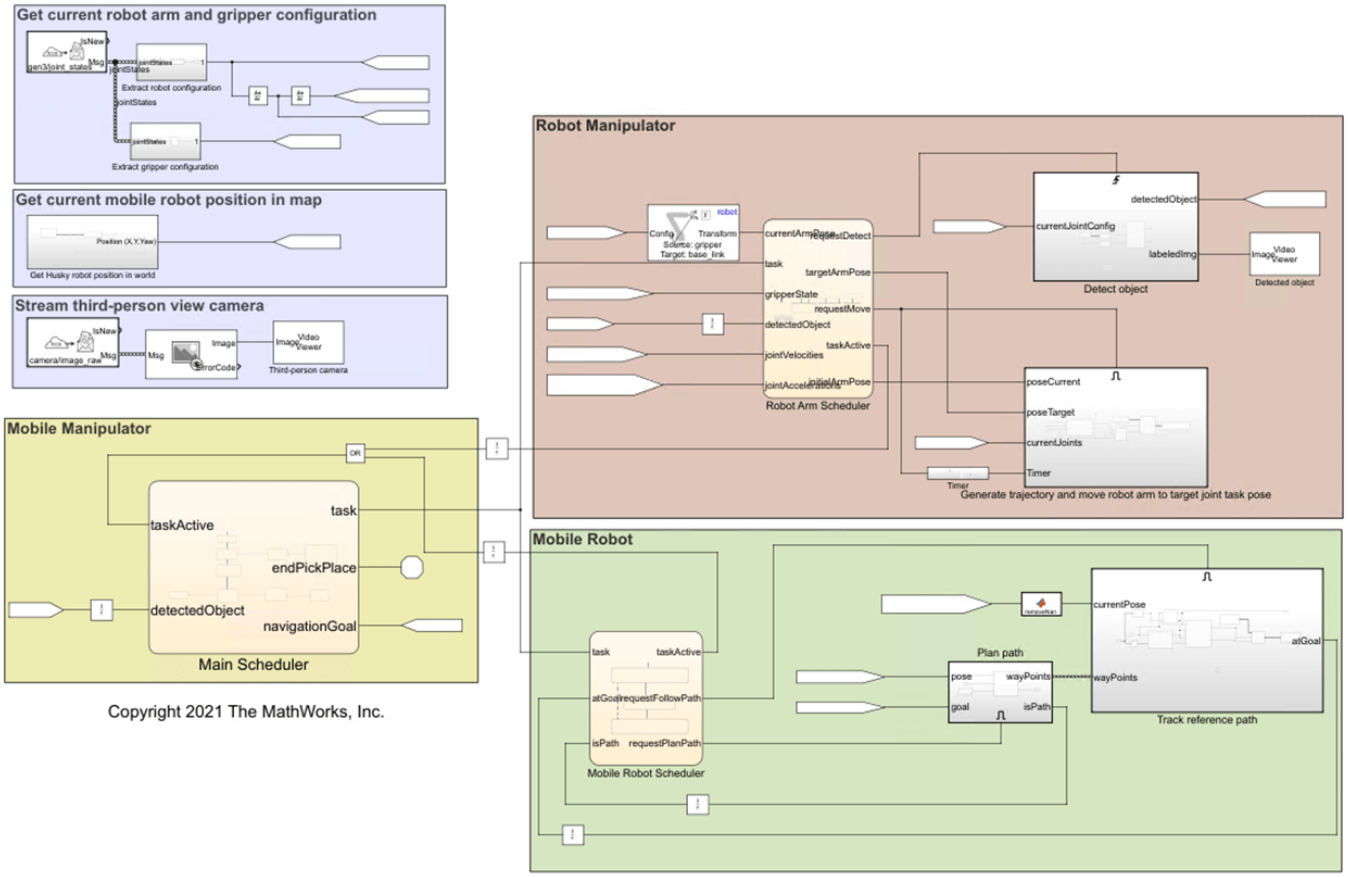


Open the Simulink model to explore the pick-and-place components.

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

The Simulink model consists of these components:

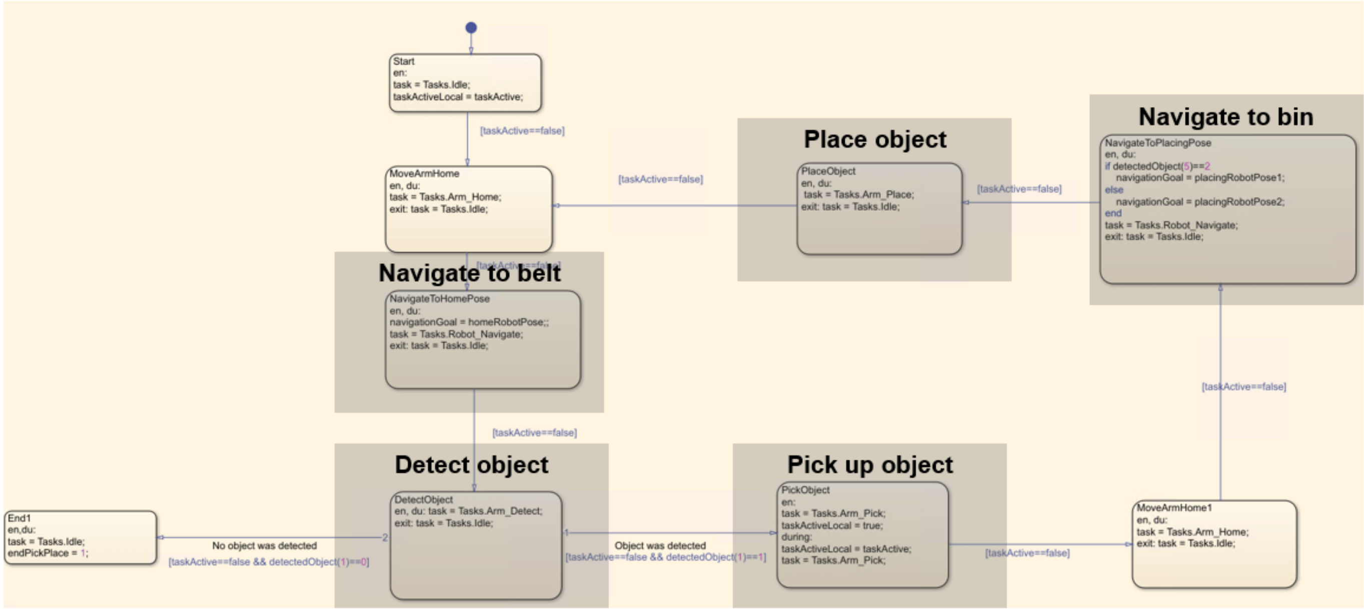
1. `Main Task Scheduler` — This Stateflow chart schedules the tasks for the mobile manipulator to complete the pick-and-place job. It activates tasks for the robot manipulator or the mobile robot according to the required workflow.
2. `Robot Manipulator` — Implements tasks assigned to the robot manipulator by the `Main Task Scheduler`. This component consists of these main modules: the Robot Arm Scheduler, the Motion Planning subsystem and the Perception subsystem.
3. `Mobile Robot` — Implements tasks assigned to the mobile robot by the main scheduler. This component consists of these modules: the Mobile Robot Scheduler, the Path Planning subsystem and the Path Following subsystem.



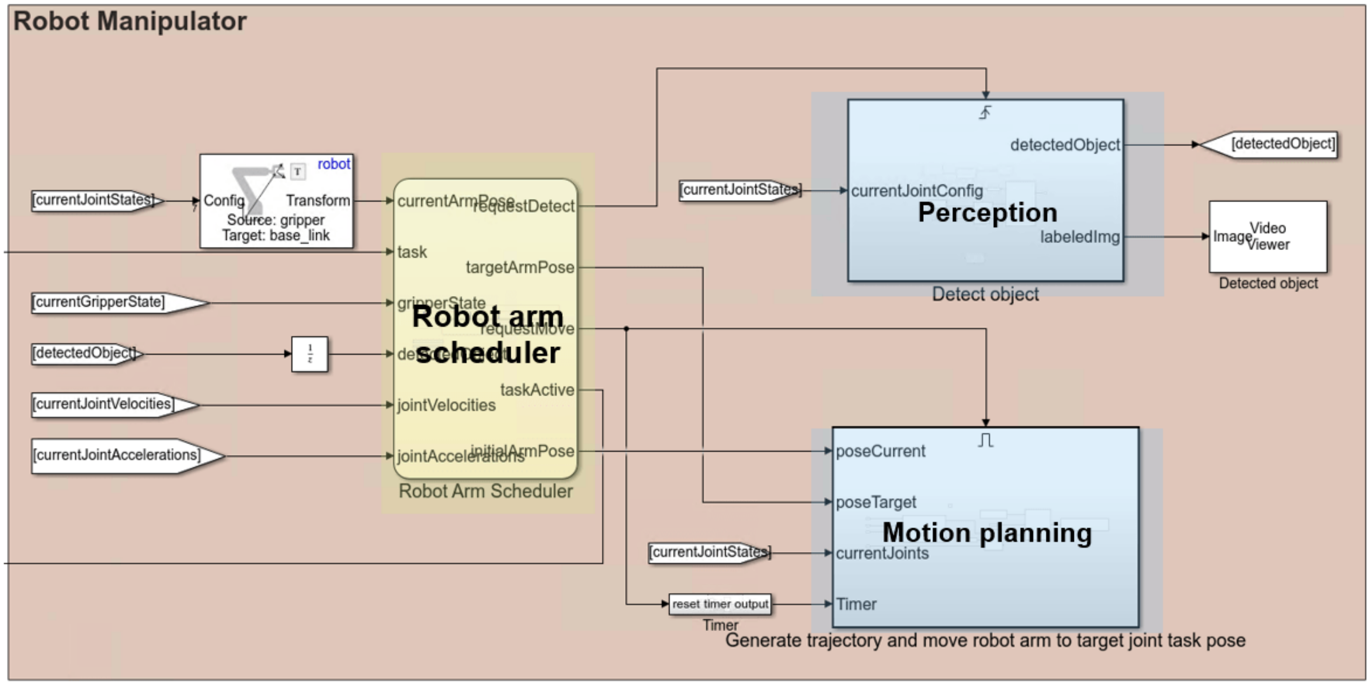
1. Main Task Scheduler

The model implements the main workflow for the mobile manipulator by using a central Stateflow chart, which follows these steps:

1. The mobile manipulator navigates to the conveyer belt to pick up an object to recycle.
2. The RGB-D camera on the robot arm detects the poses and types of objects using deep learning.
3. The robot arm picks up the detected object.
4. The mobile manipulator navigates to the appropriate recycling bin for the detected object type (bottle or can) and places it.
5. The robot returns to the conveyer belt and repeats this workflow until no more objects are detected.

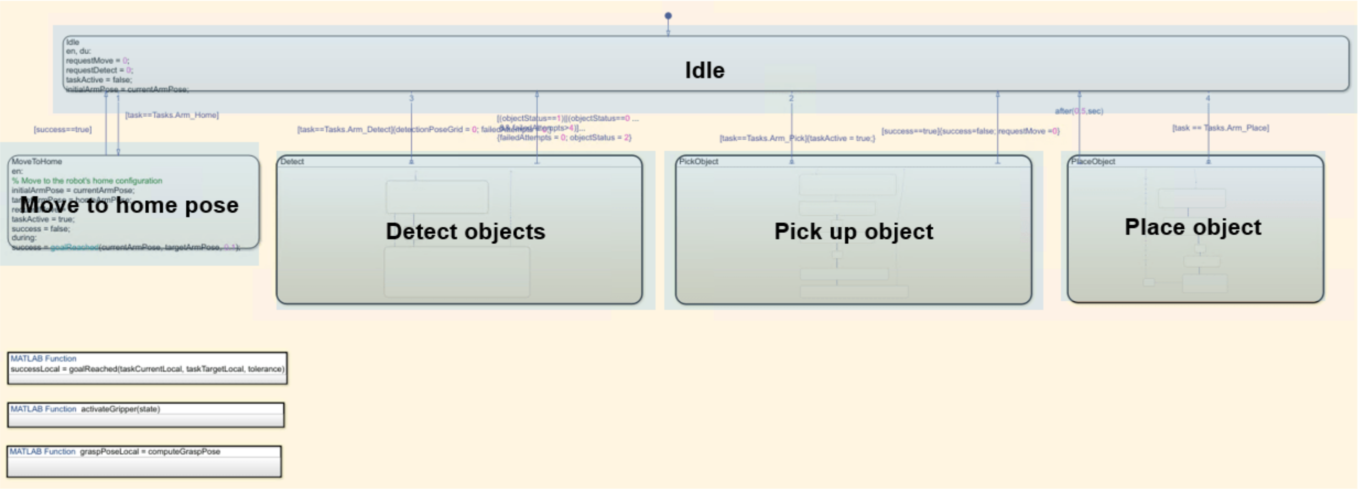


2. Robot Manipulator



A. Robot Arm Scheduler

The robot arm scheduler contains the `Idle`, `MoveToHome`, `Detect`, `PickObject`, and `PlaceObject` states. At any point in time, the robot arm is in one of these states, according to the task ordered by the `Main Task Scheduler`. To avoid conflict, if the robot arm or mobile robot is in a non-idle state, then the other must be in their idle state.


Idle
Move to home pose
Detect objects
Pick up object
Place object
MoveToHome
Detect
PickObject
PlaceObject
1
2
3
4
[success==true]
[task==Tasks.Arm_Home]
[task==Tasks.Arm_Detect]{detectionPoseGrid = 0; failedAttempts = 0;}
[(objectStatus==1)||(objectStatus==0 ...
&& failedAttempts>4)]...
{failedAttempts = 0; objectStatus = 2}
[task==Tasks.Arm_Pick]{taskActive = true;}
[success==true]{success=false; requestMove =0}
after(0.5,sec)
[task == Tasks.Arm_Place]

```
Idle
en, du:
requestMove = 0;
requestDetect = 0;
taskActive = false;
initialArmPose = currentArmPose;
```

```
MoveToHome
en:
% Move to the robot's home configuration
initialArmPose = currentArmPose;
targetArmPose = homeArmPose;
re[illegible]
taskActive = true;
success = false;
during:
success = goalReached(currentArmPose, targetArmPose, 0.1);
```

```
MATLAB Function
successLocal = goalReached(taskCurrentLocal, taskTargetLocal, tolerance)
```

```
MATLAB Function  activateGripper(state)
```

```
MATLAB Function  graspPoseLocal = computeGraspPose
```

B. Motion Planning

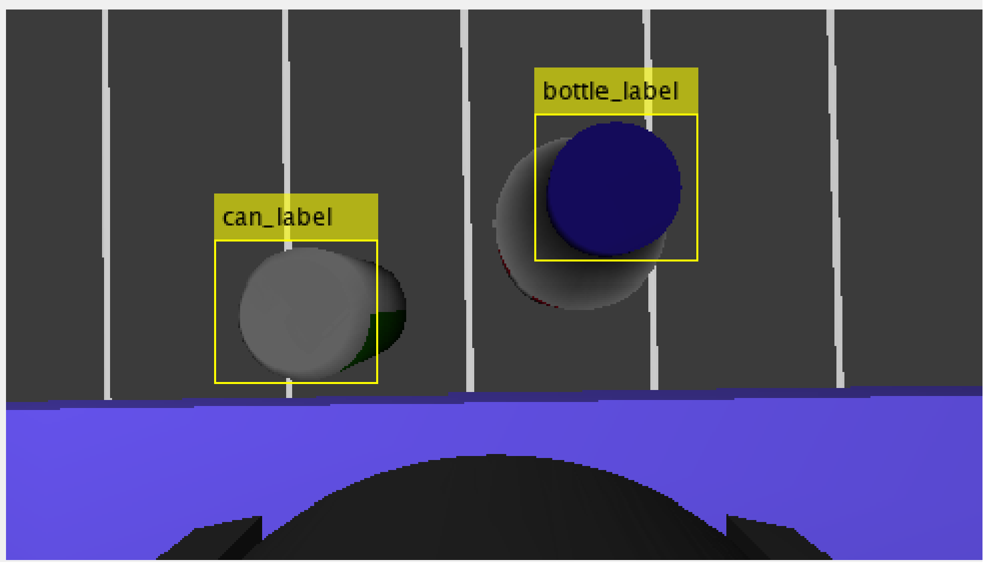
The motion planning subsystem is an enabled subsystem that, once enabled by a signal from the robot arm scheduler, plans a trajectory and uses the ros-control package to send the command for tracking that trajectory to the joint trajectory controller running in ROS.

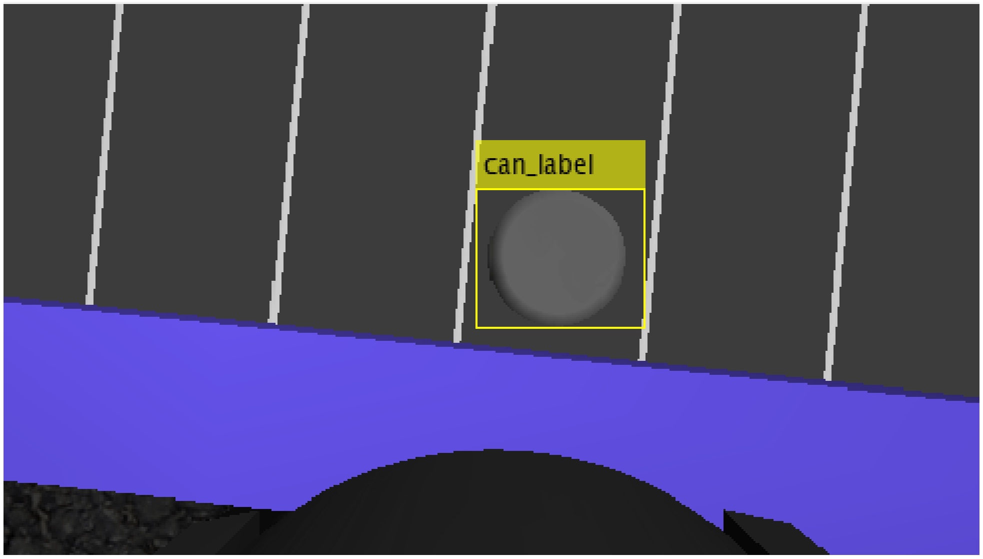
The robot arm scheduler sends a signal to *enable* this subsystem when the robot manipulator must move during the pick-and-place workflow. The subsystem remains enabled until the robot reaches the desired target pose. For more information in enabled subsystems, see “Using Enabled Subsystems” (Simulink).

C. Perception

The perception subsystem is a triggered subsystem that applies a pretrained deep learning model to the simulated end-effector camera feed from the robot to detect recyclable parts. The deep learning model takes a camera frame as input and outputs the 2-D location of the object (pixel position) and the type of recycling it requires (blue or green bin). The 2-D location in the image is mapped to the robot base frame using information about the camera properties (focal length and field of view), the input from the depth sensor, and the robot forward kinematics.

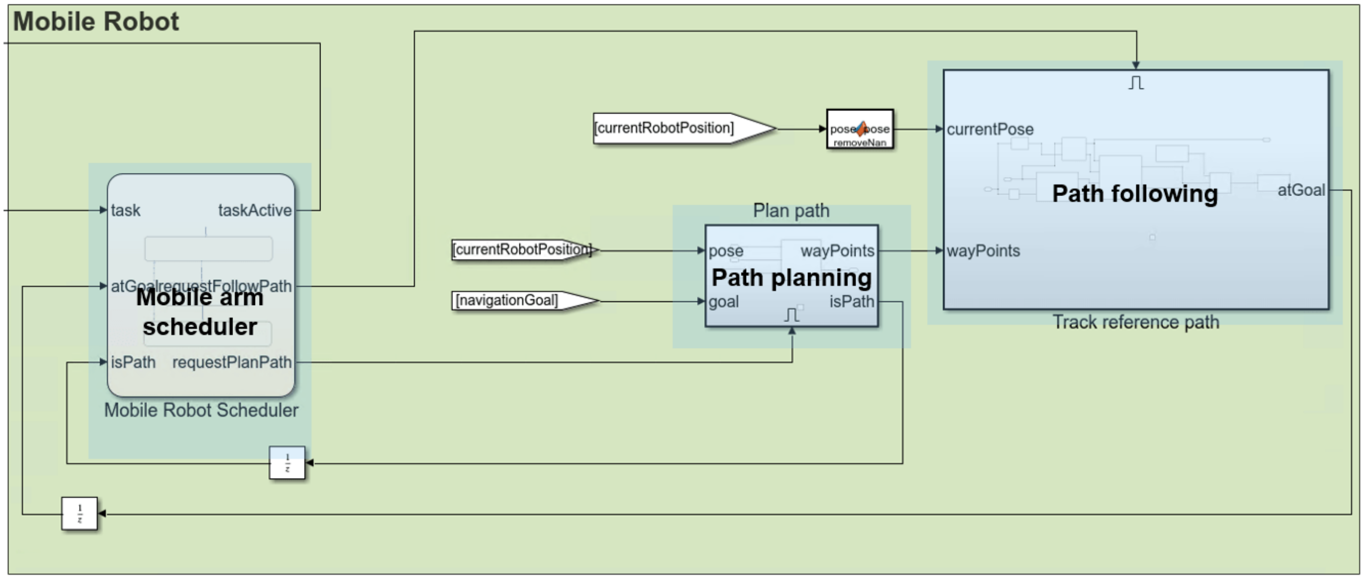
The robot manipulator task scheduler sends a signal to *trigger* this subsystem when the robot must detect the next object during the pick-and-place workflow.For more information on triggered subsystems, see “Using Triggered Subsystems” (Simulink).





3. Mobile robot

The `Mobile Robot` workflow consists of a Stateflow chart that governs the overall behavior of the mobile robot, a triggered subsystem that plans the path using a MATLAB function, and a the path following subsystem that uses control logic to follow the reference path.

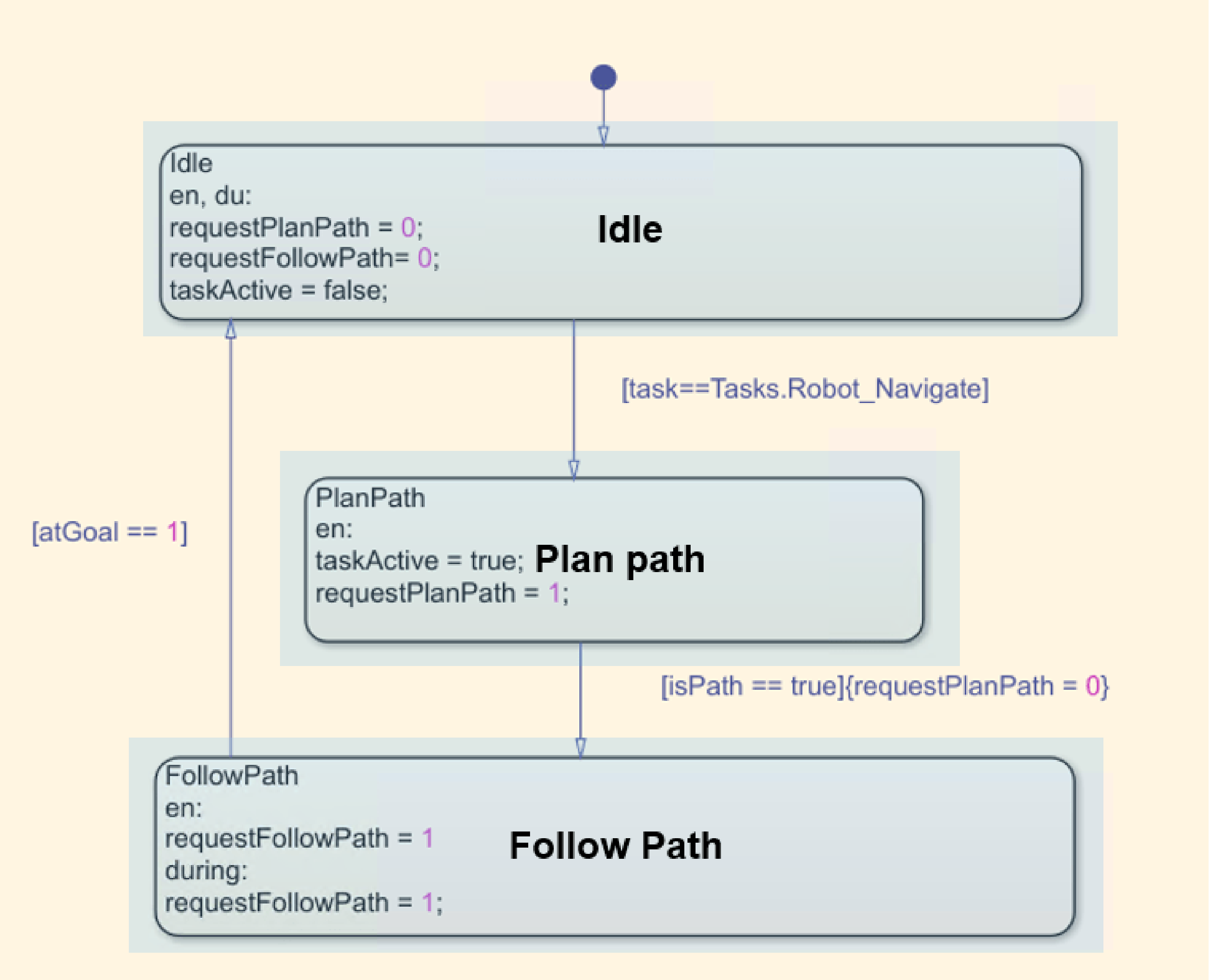


Building Warehouse Space Map

In order to obtain a map of the workspace, you must first scan the environment. Mount a lidar sensor on the Husky to navigate around the environment and use the `buildmap` function to create an occupancy grid map. For more information, see “Implement Simultaneous Localization And Mapping (SLAM) with Lidar Scans” (Navigation Toolbox).

A. Mobile Robot Scheduler

The mobile robot scheduler contains the `Idle`, `PlanPath`, and `FollowPath` states. By default, the mobile robot is in the `Idle` state, meaning that the path planning and path following systems are inactive. The mobile robot scheduler receives task commands from the `Main Task Scheduler`. Whenthe mobile robot scheduler receives a `Tasks.Robot_Navigate` task, it switches from the `Idle` state to the `PlanPath` state. The scheduler than continues to step, using feedback from the path planning and path following subystems to advance through the stages, ultimately returning to the Idle state, which renders the task inactive and indicates to the `Main Task Scheduler` that it can move on with high-level tasks. The process then repeats given new inputs from the `Main Task Scheduler`.



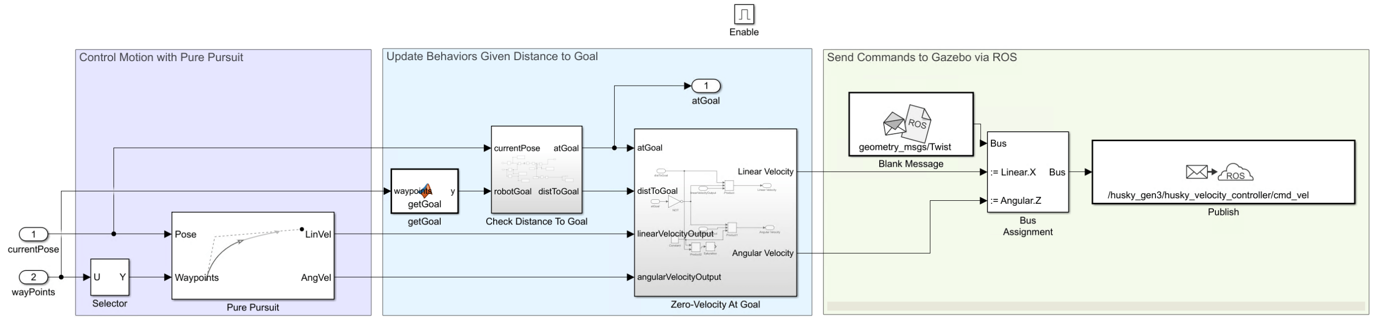
B. Path Planning

During the `PlanPath` state, the mobile robot scheduler sets `taskActive` to `true`, which triggers the path planning subsystem. This subsystem contains a MATLAB function block that plans a path on the provided binary occupancy grid map using a `plannerRRTStar` object. The subsystem returns a set of path waypoints in SE(2), as well as a logical flag, `isPath` that indicates whether a path was successfully found.

Once the mobile robot scheduler receives a value of `true` for `isPath`, it advances to the `FollowPath` state.

C. Path following

In the `FollowPath` state, the mobile robot scheduler sets the `requestFollowPath` flag to `true`, which triggers the path following subsystem.



The subsystem has these main parts, listed in the order of execution:

1. **Control Motion with Pure Pursuit** — The Pure Pursuit block is the primary motion controller. Given the current pose of the robot and the upcoming waypoint, it assigns a velocity to move the robot to the waypoint.
2. **Update Behaviors Given Distance to Goal** — The `Check Distance to Goal` and `Zero-Velocity at Goal` subsystems ensure that the robot stops when it has reached the goal. The model also uses the output from the `Check Distance to Goal` subsystem to verify task completion.
3. **Send Commands to Gazebo via ROS** — The `Blank Message` and `Publish` blocks send the command linear and angular velocities from the `Zero-Velocity at Goal` subsystem to the appropriate velocity controller on the robot in Gazebo.

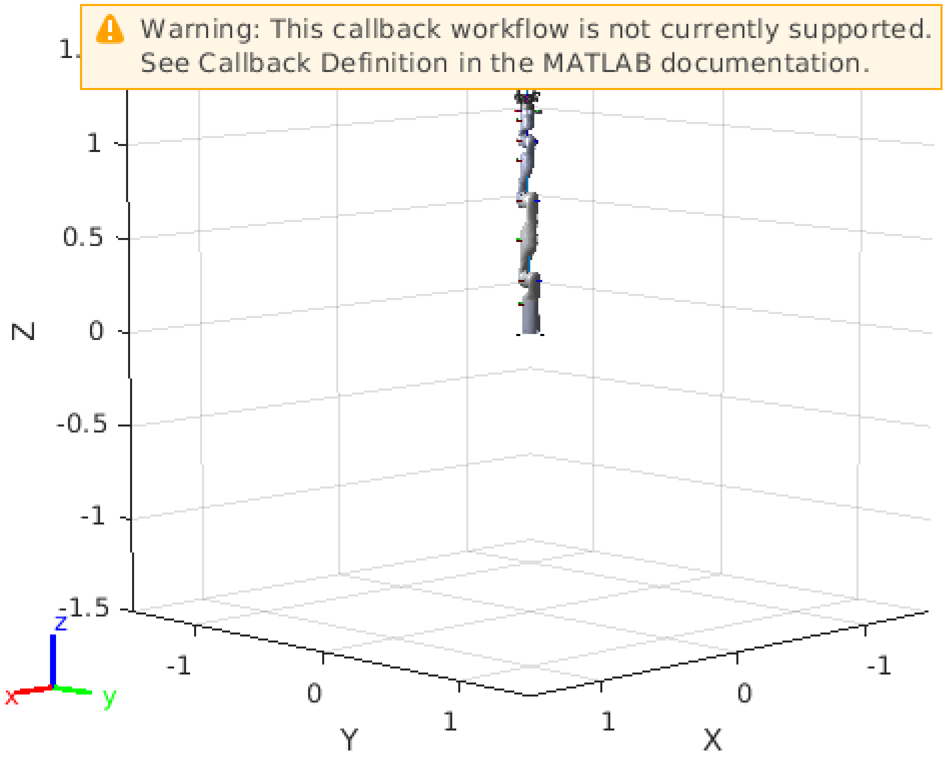
Together, these components drive the robot through the Gazebo world. Once the robot reaches the goal, the subsystem returns a value of `true` for `atGoal`, which indicates to the mobile robot scheduler that this state is complete.

Assign Variables for Use In Model

To run the Simulink model, you must assign some parameters related to the robot, default configurations, and target poses.

This simulation uses a KINOVA Gen3 manipulator attached to a Husky mobile robot. For the manipulator, a model of a KINOVA Gen3 with a gripper affixed is stored in a MAT file. Load the robot manipulator rigid body tree.

```
load("helperKINOVAGen3MobileArmPickPlace.mat");
show(robot)
```



Warning: This callback workflow is not currently supported.
See Callback Definition in the MATLAB documentation.

```
ans = 
  Axes (Primary) with properties:

             XLim: [-1.5000 1.5000]
             YLim: [-1.5000 1.5000]
           XScale: 'linear'
           YScale: 'linear'
    GridLineStyle: '-'
         Position: [0.1300 0.1100 0.7750 0.8150]
            Units: 'normalized'

  Show all properties
```

Initialize Pick and Place Application

Set the initial robot arm configuration and name of the end-effector body.

```
initialRobotJConfig = [1.583 0.086 -0.092 1.528 0.008 1.528 -0.08];
endEffectorFrame = "gripper";
```

Specify the home robot arm configuration and two poses for dropping objects of two different types. The first pose corresponds to the blue bin, for objects of type 1, and the second pose corresponds to the green bin, for objects of type 2.

```
homeArmPose = trvec2tform([0.0 -0.35 0.4])*axang2tform([0 0 1 -pi/2])*axang2tform([0 1 0 pi]);
detectionArmPose = trvec2tform([0.4 0.0 0.56])*axang2tform([0 0 1 -pi/2])*axang2tform([0 1 0 pi]
placingArmPose = trvec2tform([0.6 0.0 0.57])*axang2tform([0 0 1 -pi/2])*axang2tform([0 1 0 pi/2]
```

Specify the home mobile robot pose and two poses for dropping objects in the blue and green bins, depending on object type.

```
homeRobotPose = [7.95, 10.6, 0]; % x, y, theta
placingRobotPose2 =[13.16, 7.94, -0.92];
placingRobotPose1 = [15.15, 3.8, -1.752];
```

Set the step size for the simulation.

```
Ts = 0.1;
```

Start the Pick-and-Place Workflow

Start a ROS-based simulator for a KINOVA Gen3 robot and the configure MATLAB connection with the robot simulator. This example

Start Gazebo Simulation

This example uses a virtual machine (VM) containing ROS Melodic available for download here. This example does not support ROS Noetic as it relies on ROS packages which are only supported until ROS Melodic.

- Start the Ubuntu® virtual machine desktop.
- In the Ubuntu desktop, click the **Mobile Manipulator World** icon to start the Gazebo world built for this example, or run these commands:

```
source /opt/ros/melodic/setup.bash; source ~/catkin_ws/devel/setup.bash
export SVGA_VGPU10=0
export GAZEBO_PLUGIN_PATH=/home/user/src/GazeboPlugin/export:$GAZEBO_PLUGIN_PATH
roslaunch kortex_gazebo_depth mobilemanipulator.launch
```

Start ROS Master and Gazebo Interface Connection

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 on 192.168.203.128:11311 and your host computer address is `192.168.31.1`. Replace these with the appropriate values corresponding to your ROS device setup. Start the ROS 1 network using `rosinit`.

```
rosIP = '172.16.34.129'; % IP address of ROS enabled machine
rosshutdown % shut down existing connection to ROS
```

```
Shutting down global node /matlab_global_node_01327 with NodeURI http://172.16.34.1:44315/ and Ma
```

```
rosinit(rosIP,11311);
```

```
Initializing global node /matlab_global_node_63154 with NodeURI http://172.16.34.1:34719/ and Mas
```

Initialize the Gazebo Interface using `gzinit`.

```
gzinit(rosIP);
```

Initialize Model

Set the initial arm pose in Gazebo using ROS.

```
configResp = helperSetCurrentRobotJConfig(initialRobotJConfig);
```

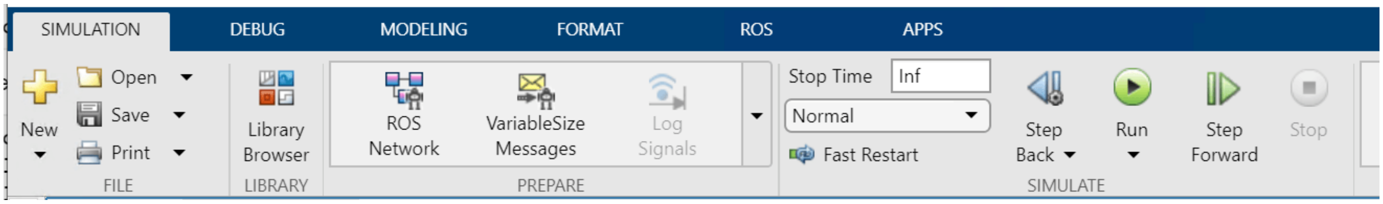
Unpause Gazebo physics.

```
physicsClient = rossvcclient("gazebo/unpause_physics");
physicsResp = call(physicsClient,"Timeout",3);
```

Load the precomputed map of the warehouse recycling facility.

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

Start the workflow. Simulate the model by selecting **Run** on the **Simulation** tab.



Stop Time Inf

Normal

The model simulates until stopped.

Once both items have been moved, with the model still running, use the Gazebo MATLAB Interface to reset the positions of the red bottle and green can:

```
gzlink('set','Green Can','link','Position',[7.80007 11.2462 0.736121],'Orientation',[1 0 0 0]);
```

```
STATUS:  Succeed


MESSAGE:  Parameter set successfully.
```

```
gzlink('set','Red Bottle','link','Position',[7.932210479374374  11.287119941912177 0.78380376701
```

```
STATUS:  Succeed


MESSAGE:  Parameter set successfully.
```

You can also change the position values to try out the controller on different positions. The MATLAB interface enables you to control the Gazebo world directly from MATLAB, via commands by using functions such as `gzlink` and `gzmodel`.

To restart the model:

- Close and re-run the script on the Ubuntu VM. Or if using the terminal and list of commands, simply close (**Ctrl+C**) and call the `roslaunch` command again:

```
roslaunch kortex_gazebo_depth mobilemanipulator.launch
```

- Run the setup commands starting from the Initialize Model on page 1-0   section in MATLAB side and in Simulink, select **Run**.

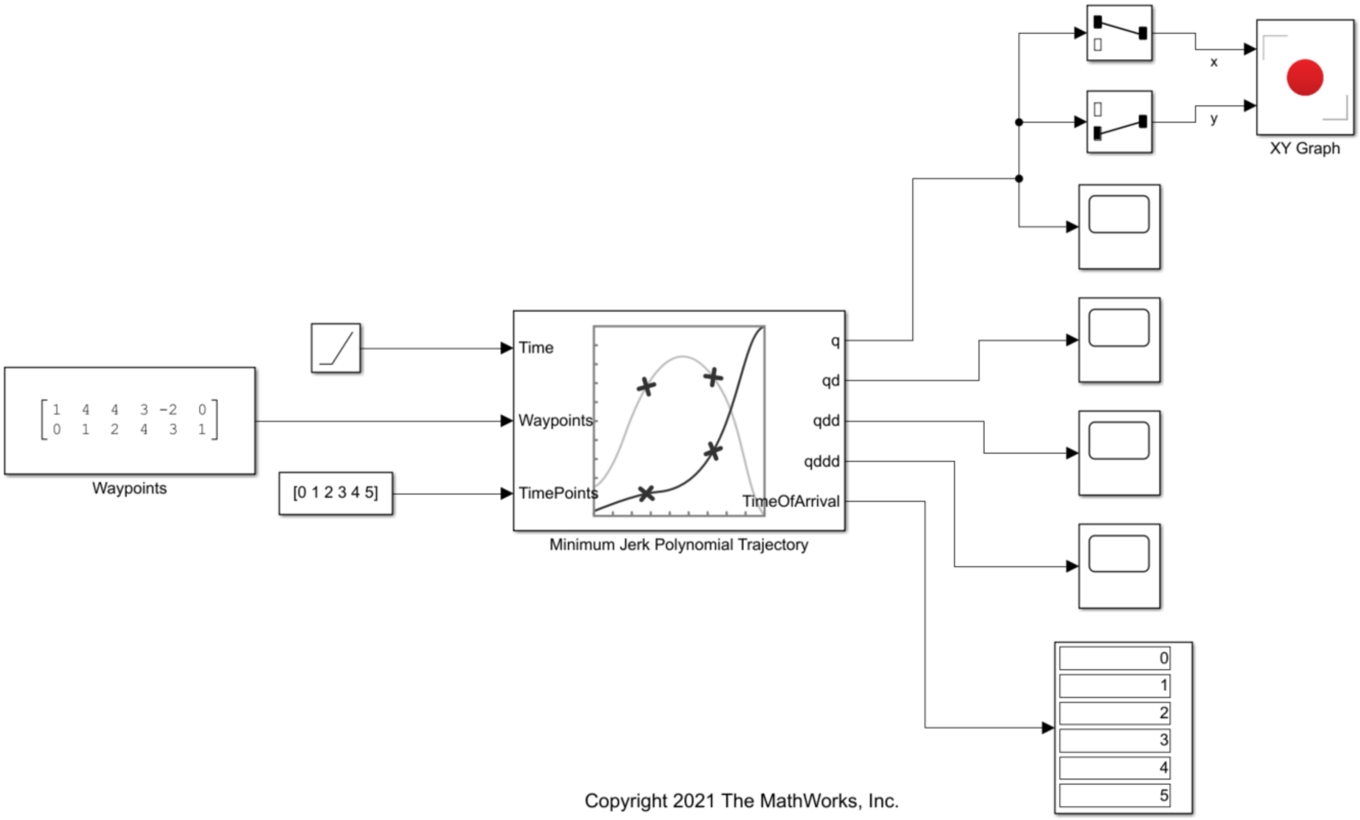
Generate Minimum Jerk Trajectory

This example shows how to generate a minimum jerk trajectory using the Minimum Jerk Polynomial Trajectory block.

Example Model

Open the model.

```
open_system("minjerk_traj_ex1.slx")
```



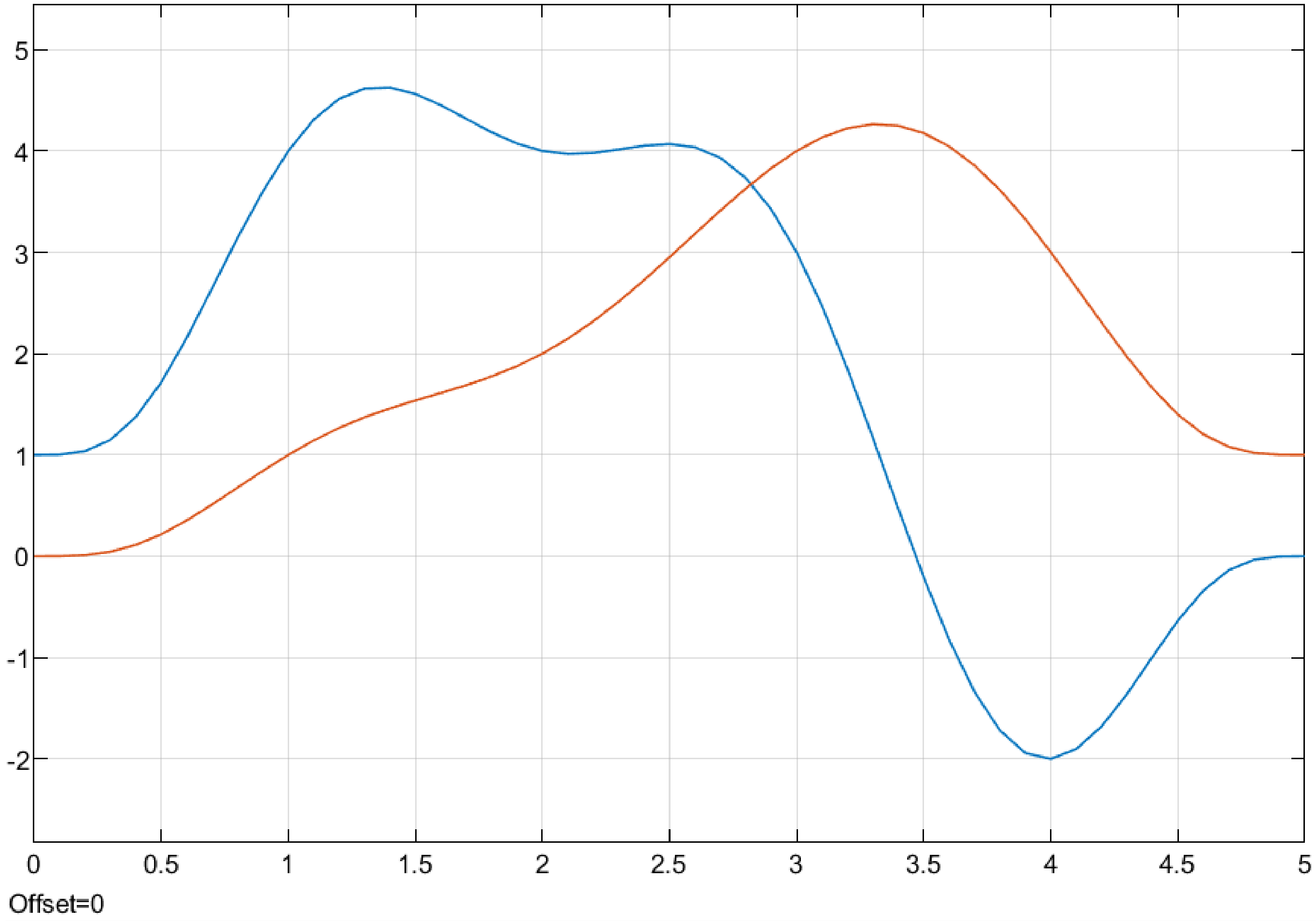
The model contains a Constant block, `Waypoints`, that specifies six two-dimensional waypoints to the **Waypoints** port of the Minimum Jerk Polynomial Trajectory block, and another Constant block specifies time points for each of those waypoints to the **TimePoints** port. The input to the **Time** port is a ramp signal, to simulate time progressing.

Simulate and Display Results

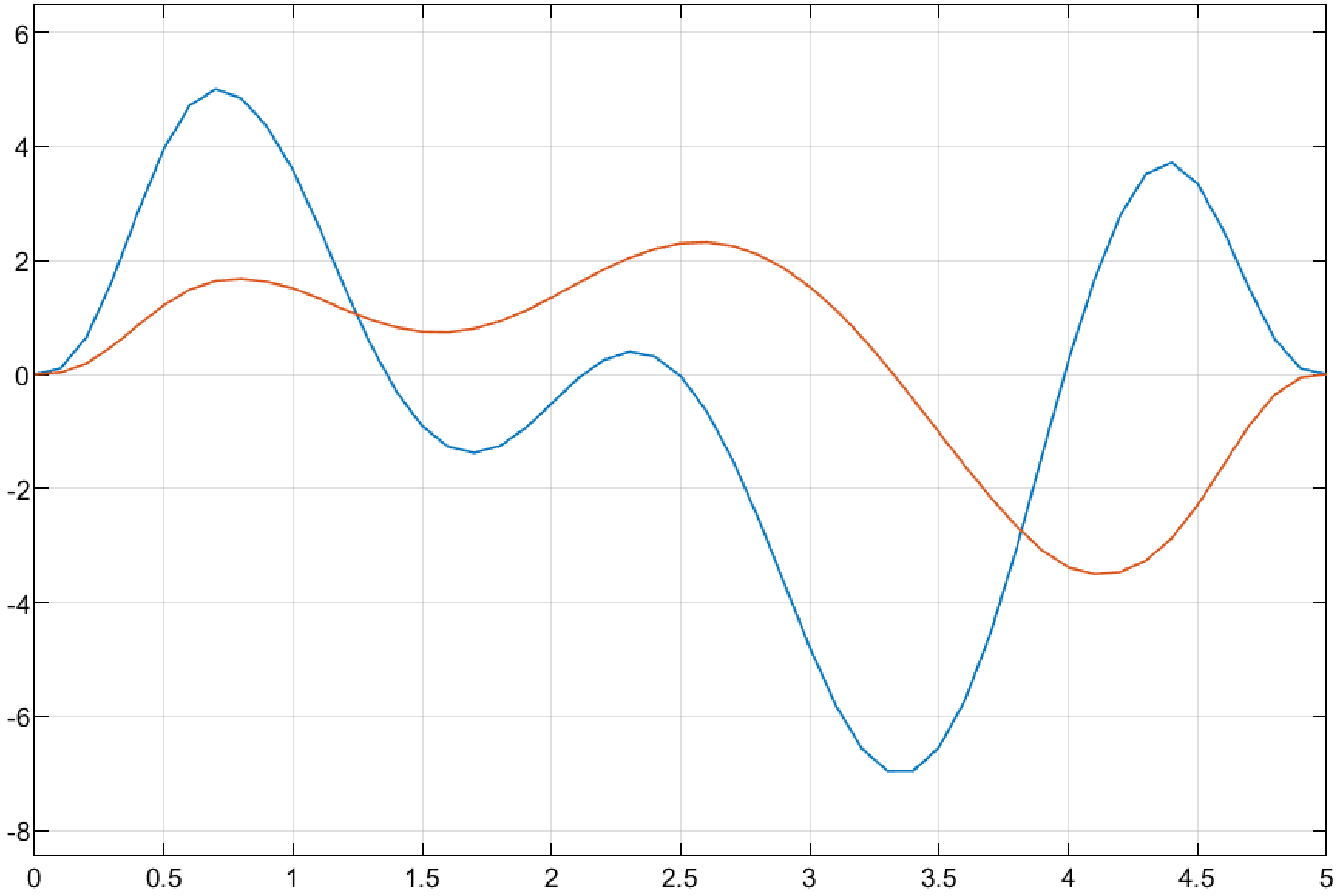
Run the simulation. Scope blocks visualize the **q** port output of positions, the **qd** port output of velocities, the **qdd** port output of accelerations, and the **qddd** port output of jerks of the trajectory.

The XY Graph shows the actual 2-D trajectory, which stays inside the defined control points, and reaches the first and last waypoints.

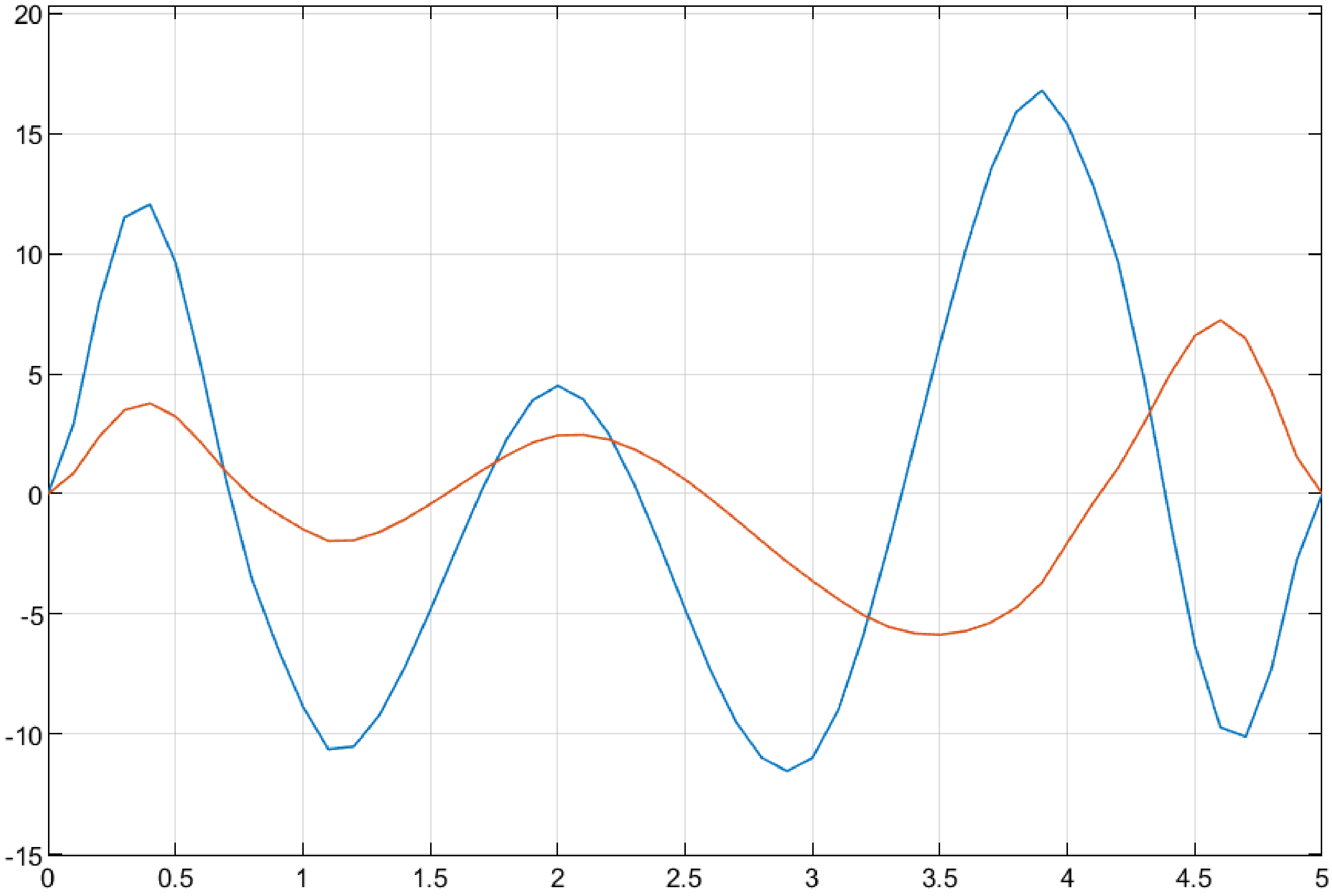
Positions



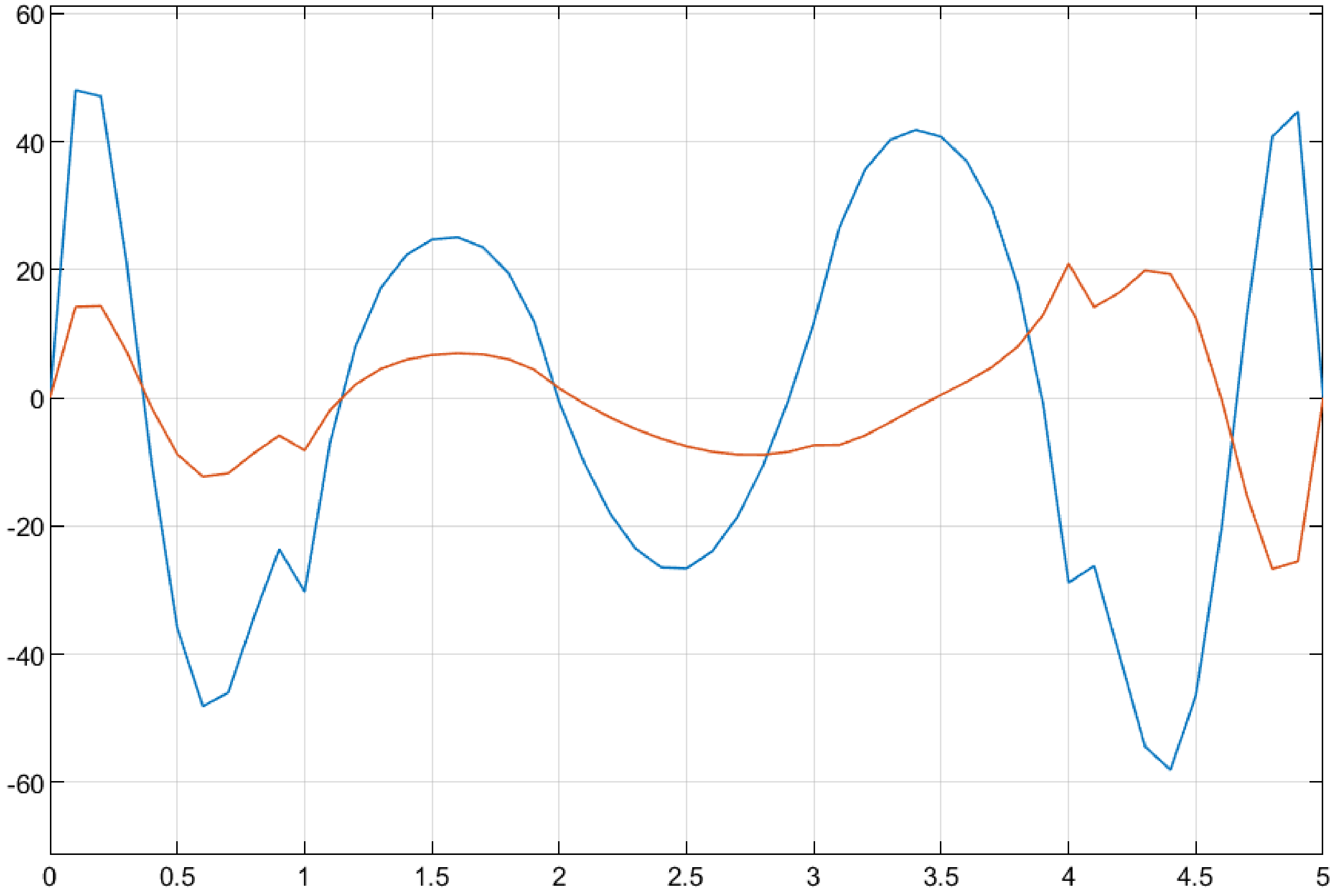
Velocities



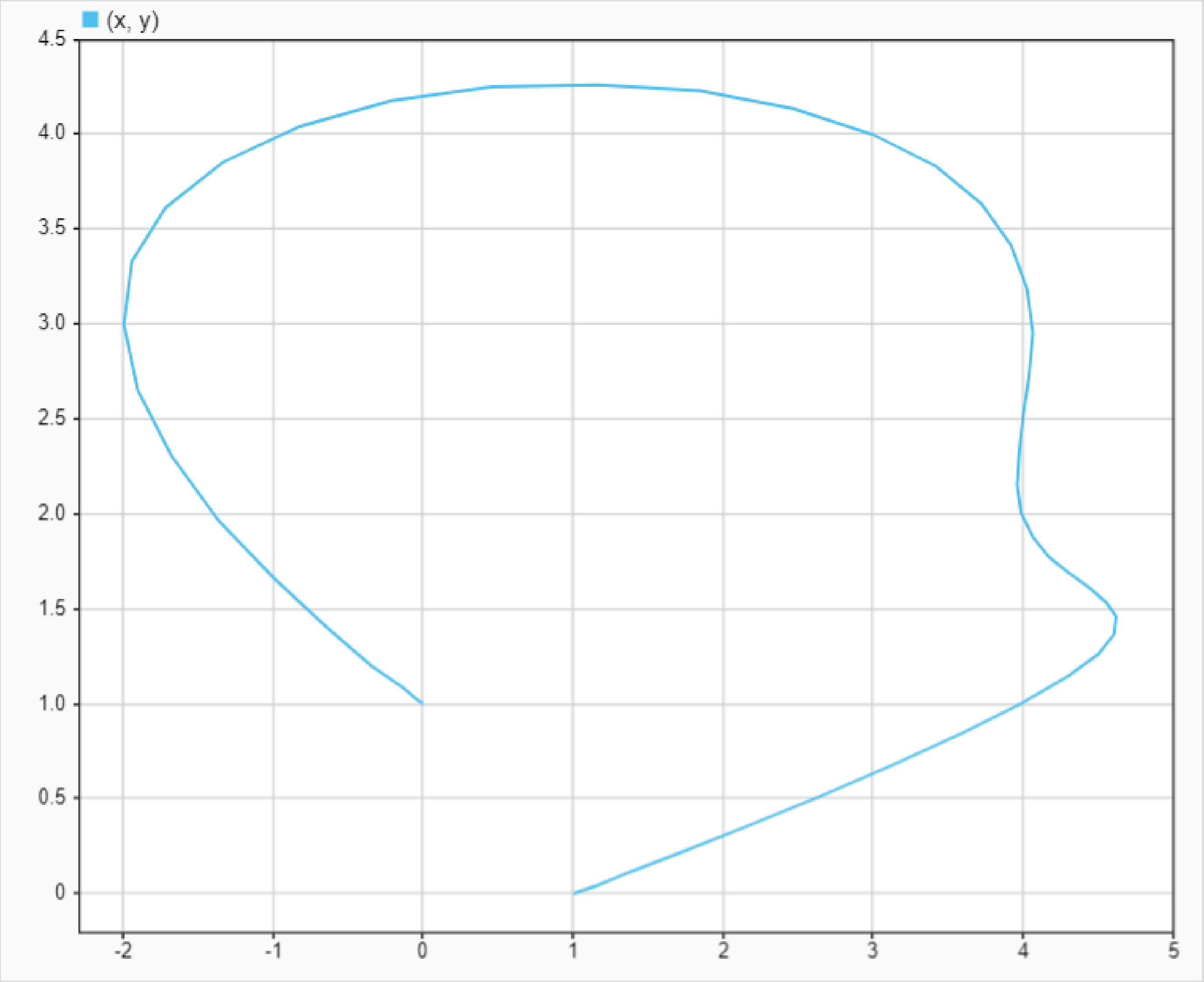
Accelerations



Jerks



x- and y-positions



🟦 (x, y)
4.5
4.0
3.5
3.0
2.5
2.0
1.5
1.0
0.5
0
-2
-1
0
1
2
3
4
5

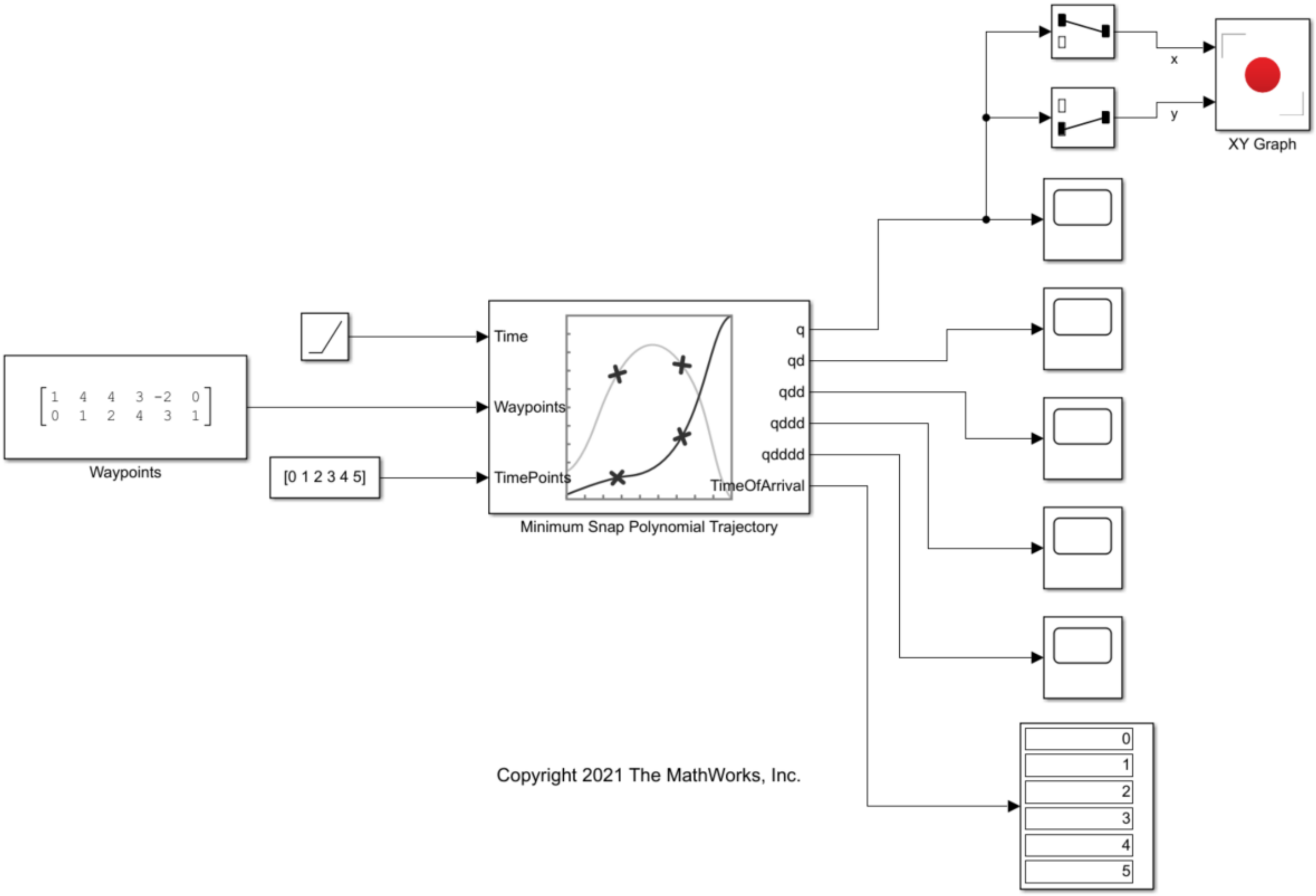
Generate Minimum Snap Trajectory

This example shows how to generate a minimum snap trajectory using the Minimum Snap Polynomial Trajectory block.

Example Model

Open the model.

```
open_system("minsnap_traj_ex1.slx")
```



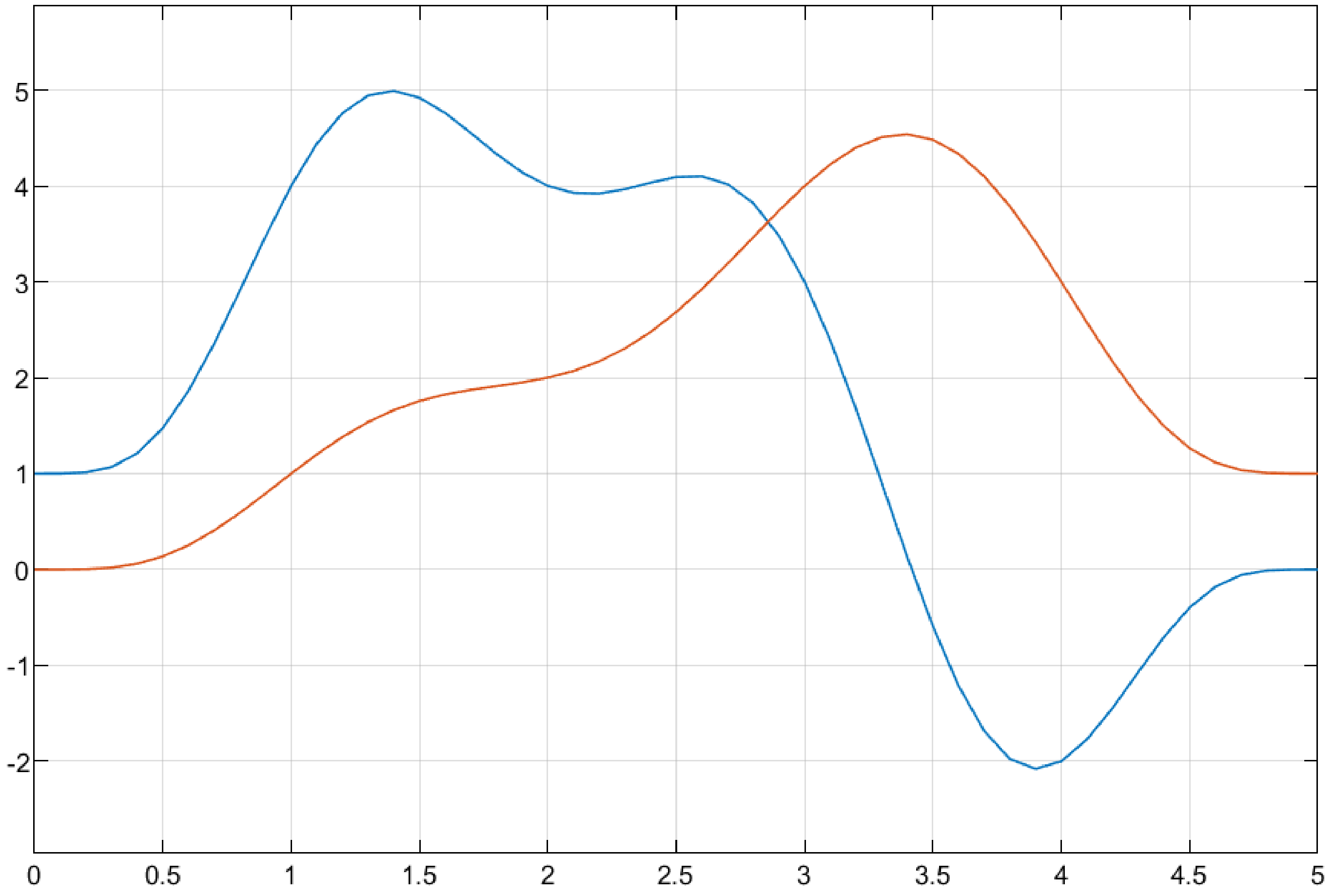
The model contains a Constant block, `Waypoints`, that specifies six two-dimensional waypoints to the **Waypoints** port of the Minimum Jerk Polynomial Trajectory block, and another Constant block specifies time points for each of those waypoints to the **TimePoints** port. The input to the **Time** port is a ramp signal, to simulate time progressing.

Simulate and Display Results

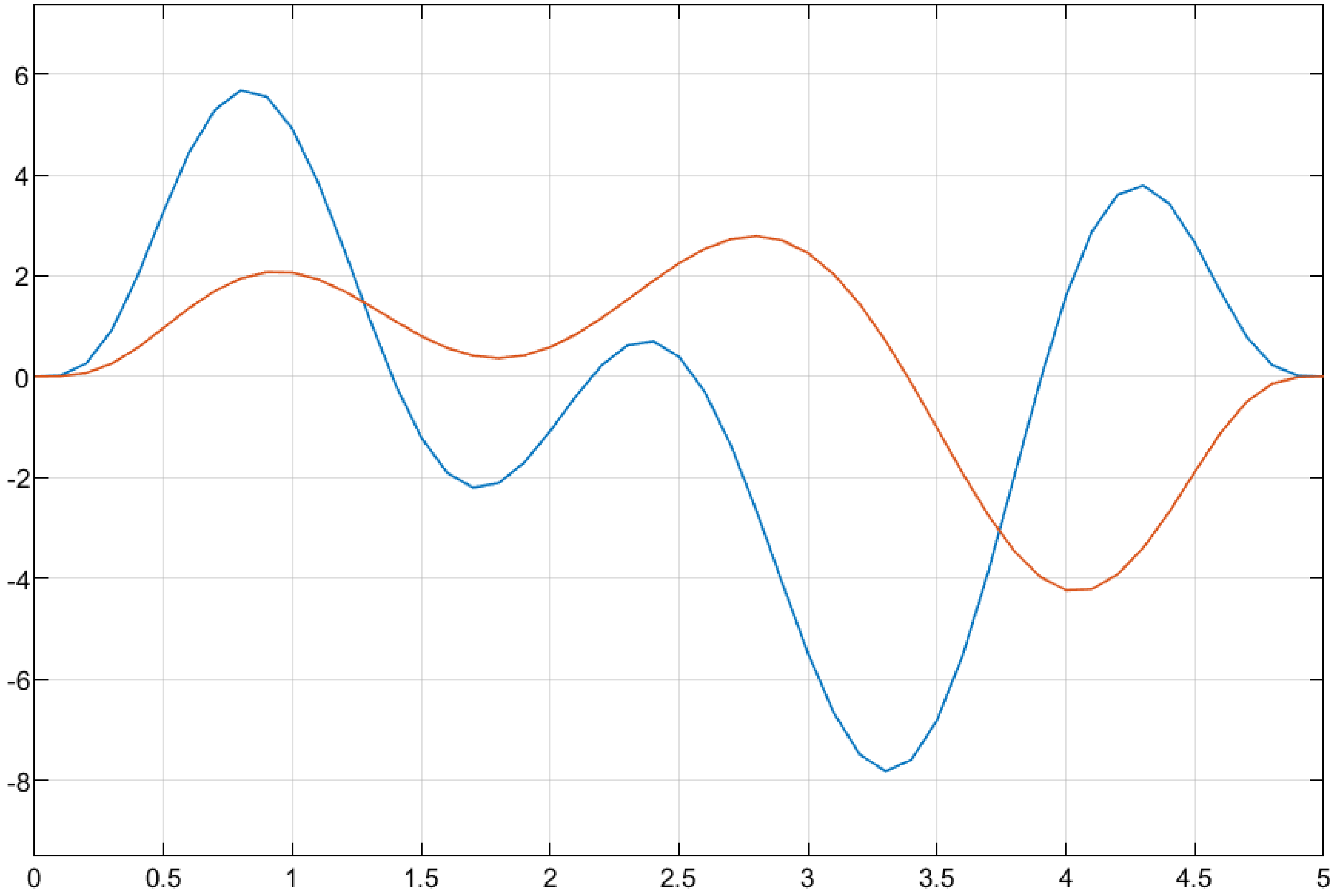
Run the simulation. Scope blocks visualize the **q** port output of positions, the **qd** port output of velocities, the **qdd** port output of accelerations, the **qddd** port output of jerks, and **qdddd** port output of snaps of the trajectory.

The XY Graph shows the actual 2-D trajectory, which stays inside the defined control points, and reaches the first and last waypoints.

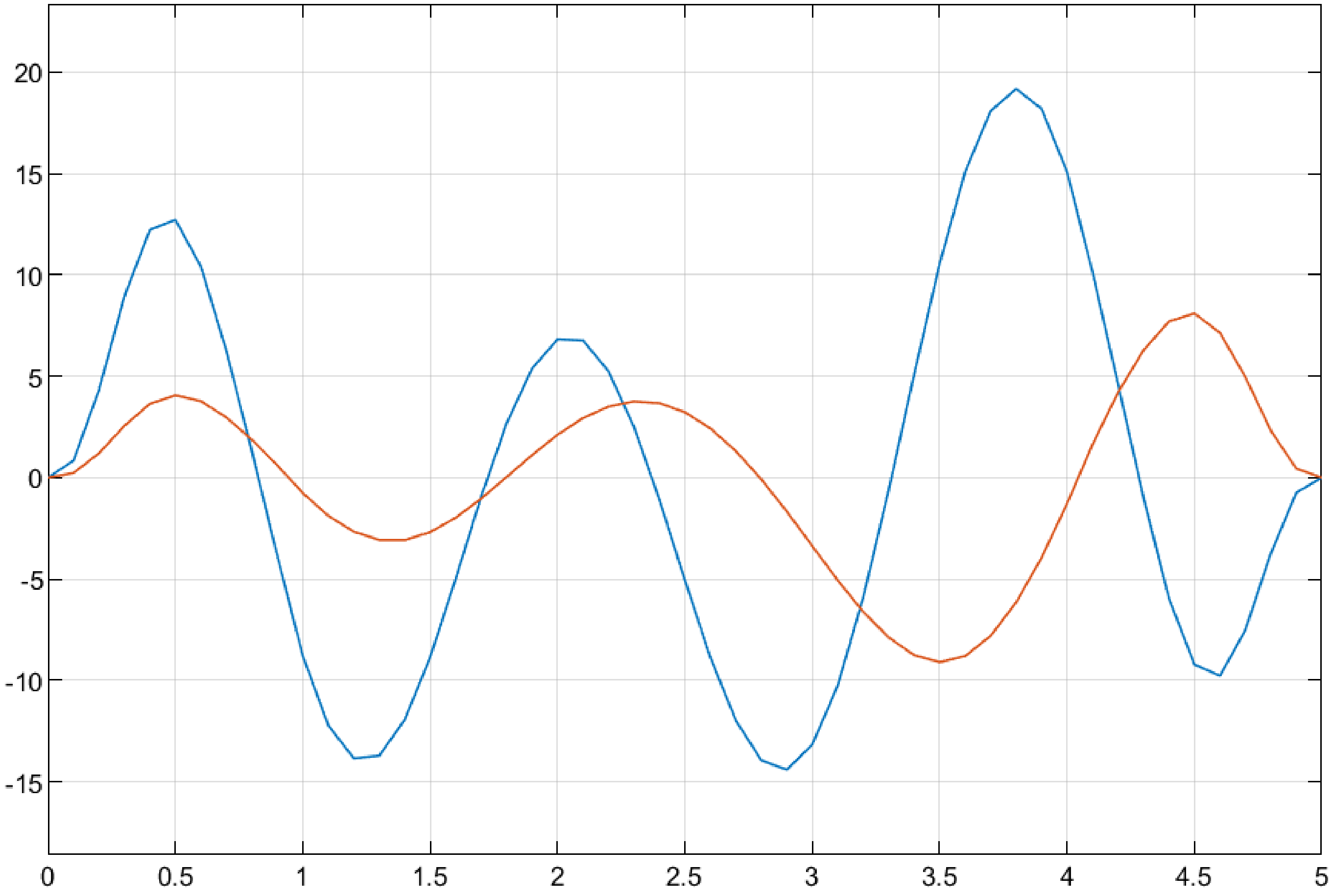
Positions



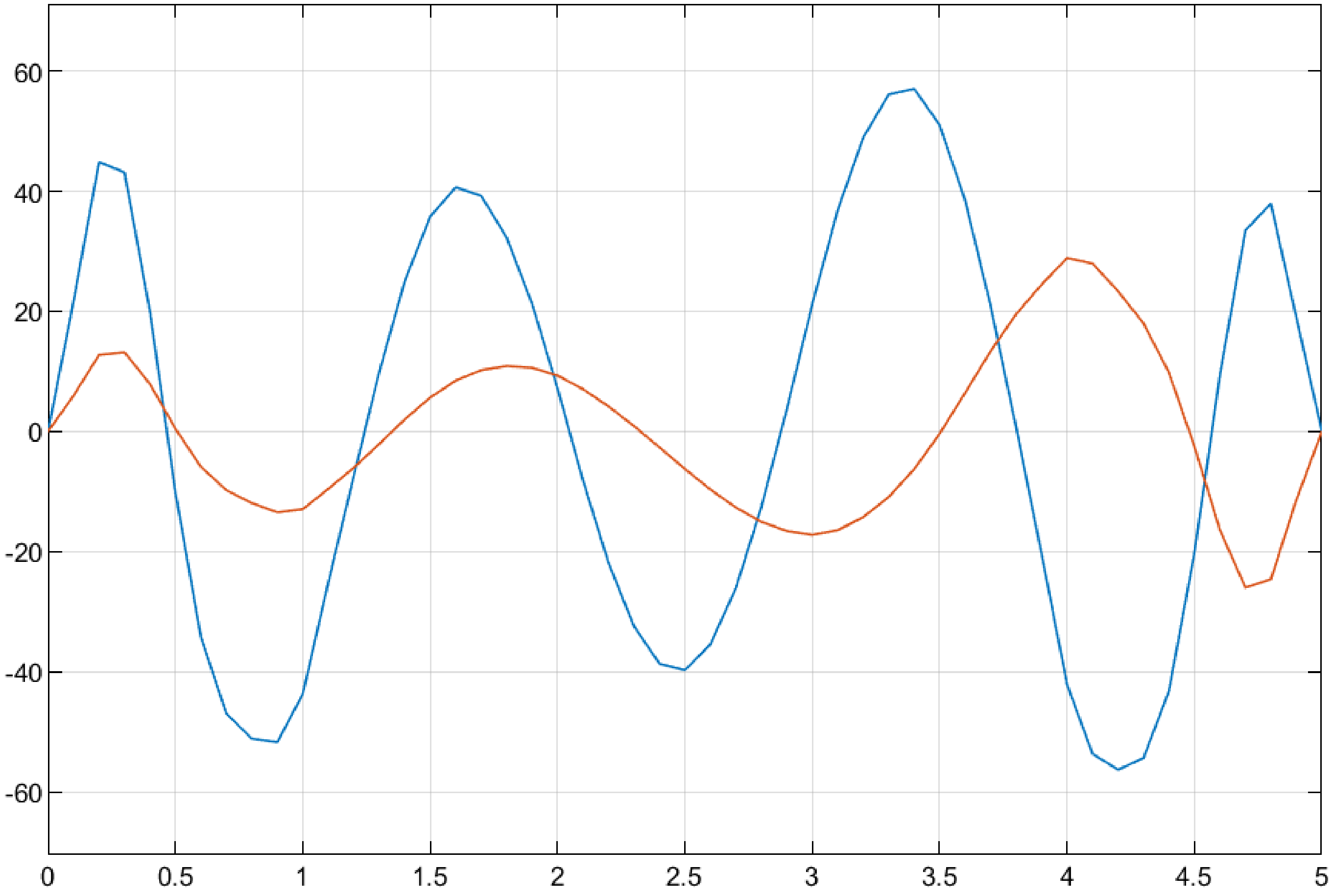
Velocities



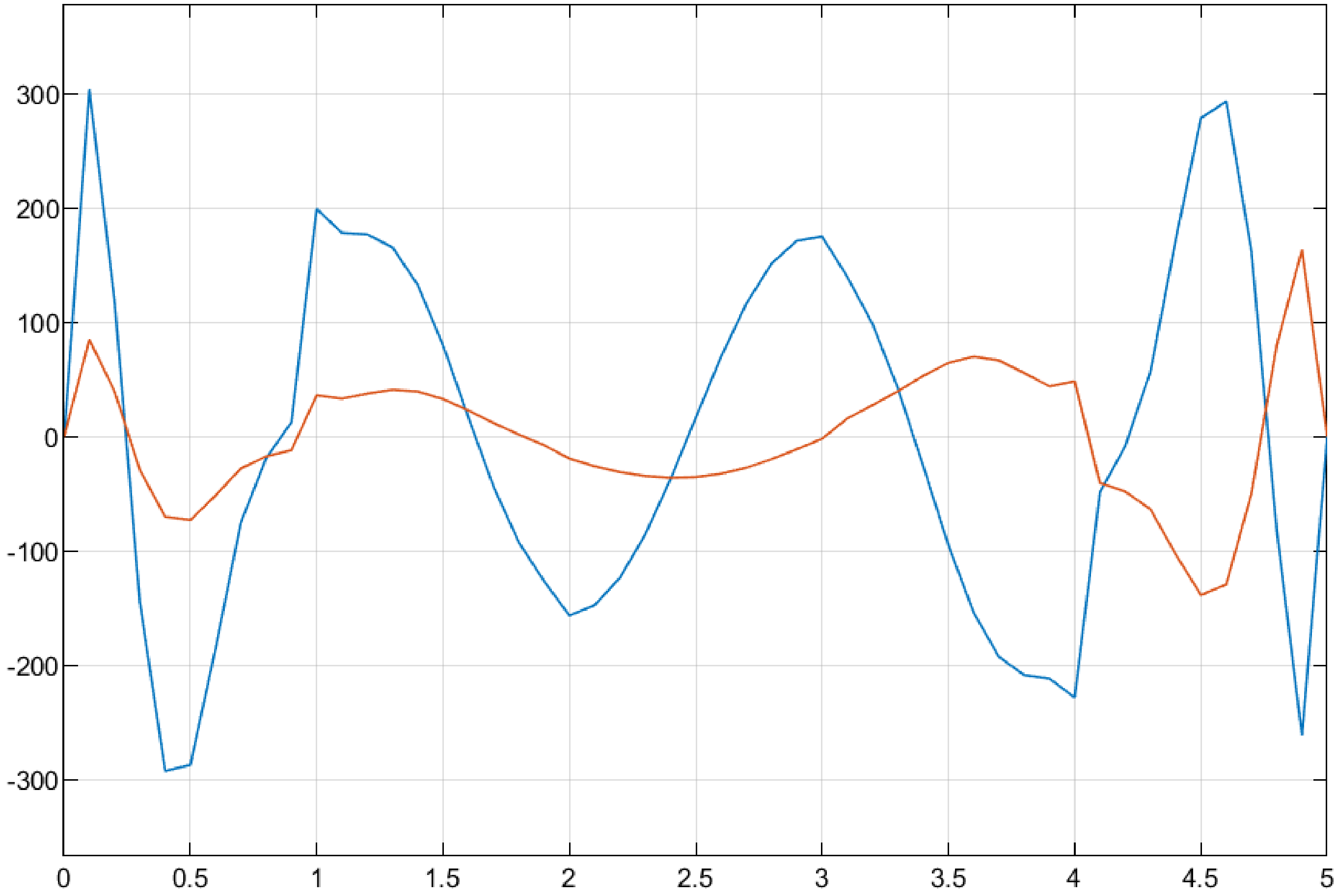
Accelerations



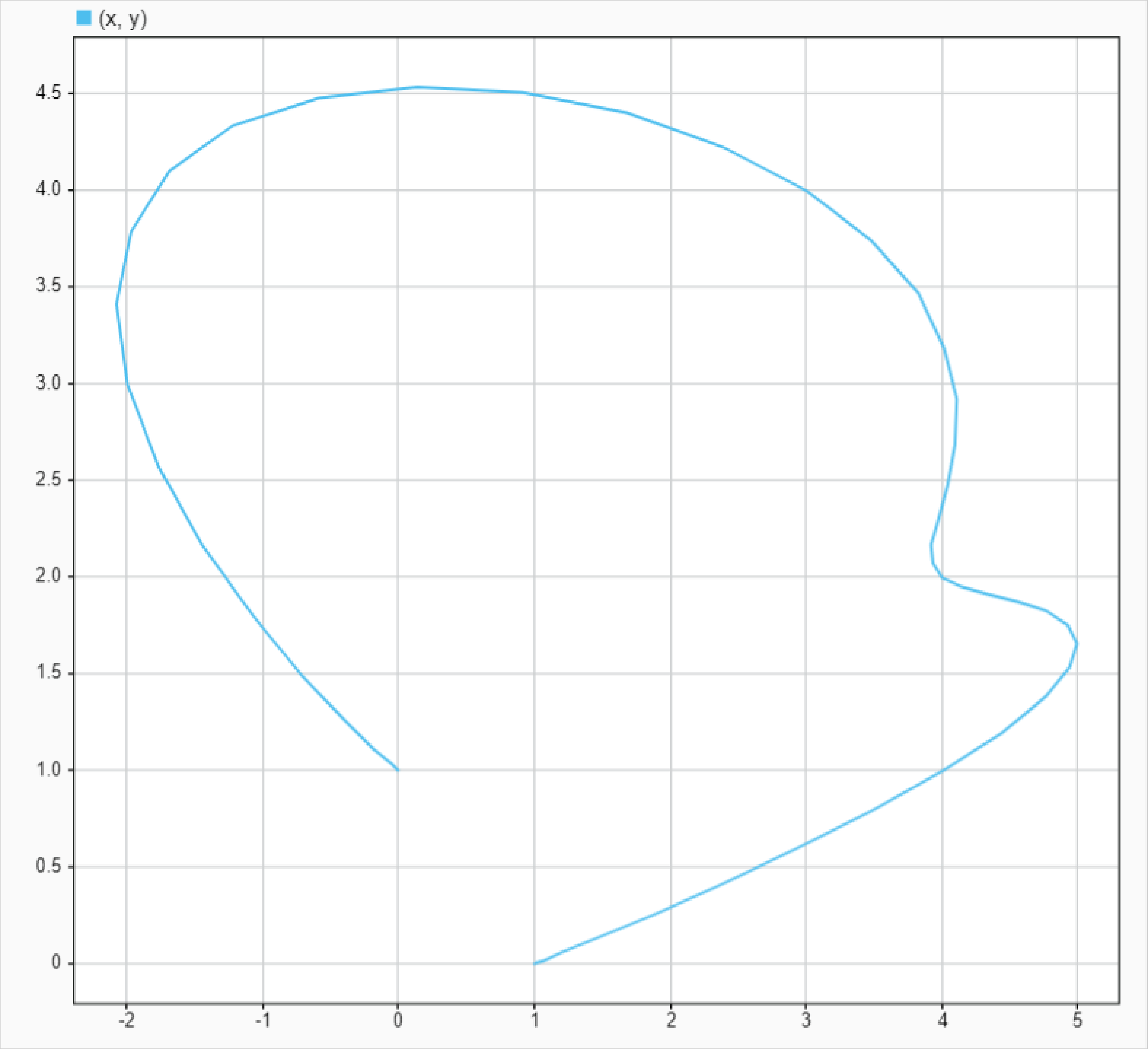
Jerks



Snaps



x- and y-positions


(x, y)
4.5
4.0
3.5
3.0
2.5
2.0
1.5
1.0
0.5
0
-2
-1
0
1
2
3
4
5

Robotics System Toolbox Topics

Rigid Body Tree Robot Model

In this section...

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

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

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

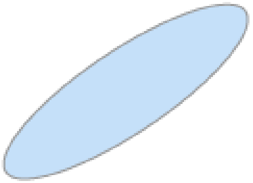
Rigid Body Tree Components

Base

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

Rigid Body

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



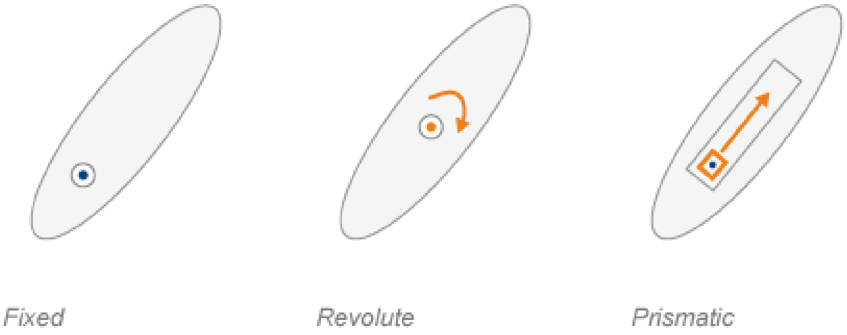
When added to a rigid body tree with multiple bodies, rigid bodies have parent or children bodies associated with them (`Parent` or `Children` properties). The parent is the body that this rigid body is attached to, which can be the robot base. The children are all the bodies attached to this body downstream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

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

Joint

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

The `rigidBodyJoint` object supports fixed, revolute, and prismatic joints.

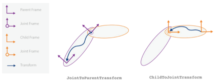


These joints allow the following motion, depending on their type:

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

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

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



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

Note The actual joint positions are not part of this Joint object. The robot model is stateless. There is an intermediate transformation between the parent and child joint frames that defines the position of the joint along the axis of motion. This transformation is defined in the robot configuration. See “Robot Configurations” on page 2-4.

Robot Configurations

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

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

Robot configurations are given as an array of structures.

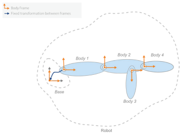
```
config = homeConfiguration(robot)
```

```
config = 

  1×6 struct array with fields:

    JointName
    JointPosition
```

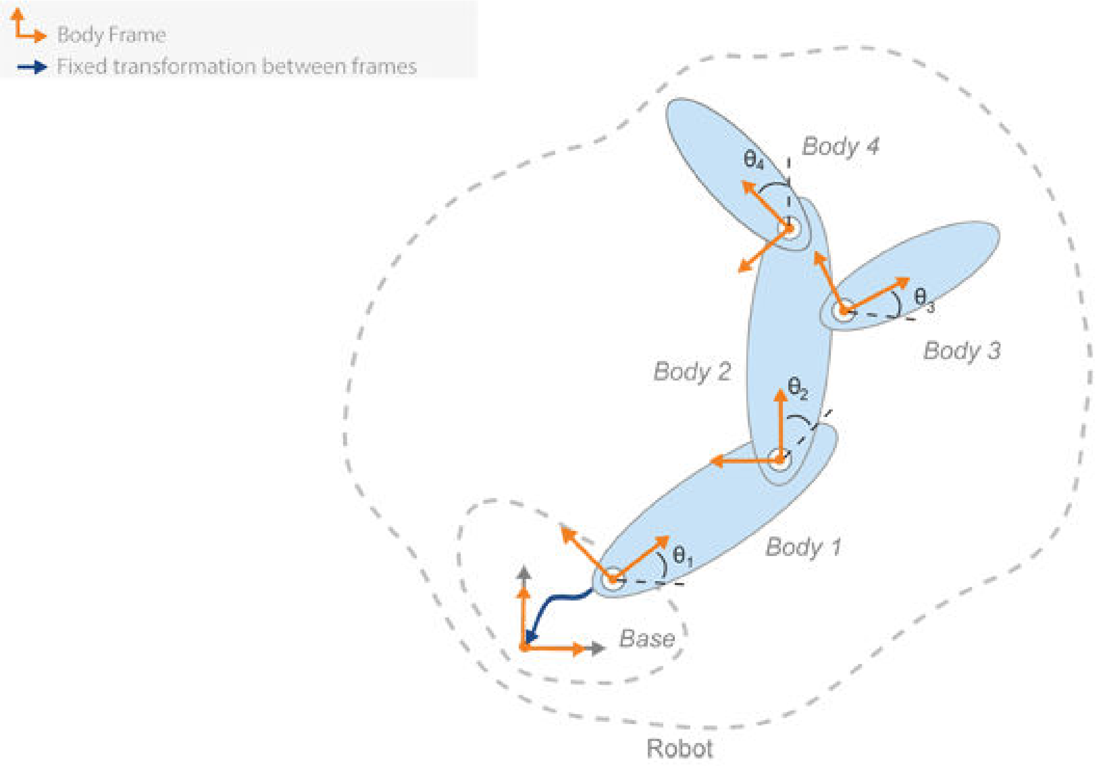
Each element in the array is a structure that contains the name and position of one of the robot joints.

```
config(1)
```

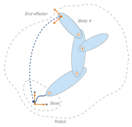
```
ans = 

  struct with fields:

        JointName: 'jnt1'
    JointPosition: 0
```



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



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



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

See Also

`rigidBodyTree` | `inverseKinematics`

Related Examples

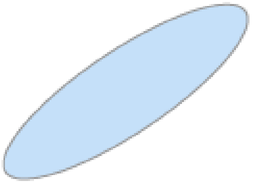
- “Build a Robot Step by Step” on page 2-6
- “Inverse Kinematics Algorithms” on page 2-10

Build a Robot Step by Step

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

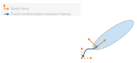
1 Create a rigid body object.

```
body1 = rigidBody('body1');
```



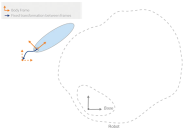
2 Create a joint and assign it to the rigid body. Define the home position property of the joint, `HomePosition`. Set the joint-to-parent transform using a homogeneous transformation, `tform`. Use the `trvec2tform` function to convert from a translation vector to a homogenous transformation.`ChildToJointTransform` is set to an identity matrix.

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



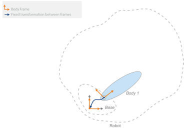
3 Create a rigid body tree. This tree is initialized with a base coordinate frame to attach bodies to.

```
robot = rigidBodyTree;
```



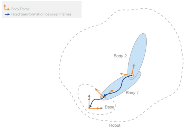
4 Add the first body to the tree. Specify that you are attaching it to the base of the tree. The fixed transform defined previously is from the base (parent) to the first body.

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



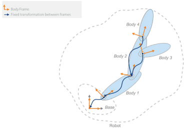
5 Create a second body. Define properties of this body and attach it to the first rigid body. Define the transformation relative to the previous body frame.

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



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

```
body3 = rigidBody('body3');
body4 = rigidBody('body4');
jnt3 = rigidBodyJoint('jnt3','revolute');
jnt4 = rigidBodyJoint('jnt4','revolute');
tform3 = trvec2tform([0.6, -0.1, 0])*eul2tform([-pi/2, 0, 0]); % User defined
tform4 = trvec2tform([1, 0, 0]); % User defined
setFixedTransform(jnt3,tform3);
setFixedTransform(jnt4,tform4);
jnt3.HomePosition = pi/4; % User defined
body3.Joint = jnt3
body4.Joint = jnt4
addBody(robot,body3,'body2'); % Add body3 to body2
addBody(robot,body4,'body2'); % Add body4 to body2
```



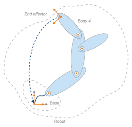
7 If you have a specific end effector that you care about for control, define it as a rigid body with a fixed joint. For this robot, add an end effector to `body4` so that you can get transformations for it.

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

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

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

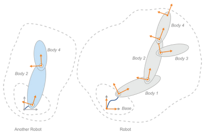
```
config = 

  1×2 struct array with fields:

    JointName
    JointPosition


tform =
   -0.5484    0.8362         0         0
   -0.8362   -0.5484         0         0
         0         0    1.0000         0
         0         0         0    1.0000
```



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

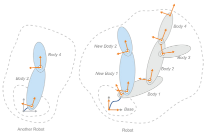
9 You can create a subtree from your existing robot or other robot models by using `subtree`. Specify the body name to use as the base for the new subtree. You can modify this subtree by adding, changing, or removing bodies.

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



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

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

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



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

```
showdetails(robot)
```

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

See Also

`rigidBodyTree` | `inverseKinematics`

Related Examples

- “Rigid Body Tree Robot Model” on page 2-2

Inverse Kinematics Algorithms

In this section...

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

Choose an Algorithm

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

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

BFGS Gradient Projection

The Broyden-Fletcher-Goldfarb-Shanno (BFGS) gradient projection algorithm is a quasi-Newton method that uses the gradients of the cost function from past iterations to generate approximate second-derivative information. The algorithm uses this second-derivative information in determining the step to take in the current iteration. A gradient projection method is used to deal with boundary limits on the cost function that the joint limits of the robot model create. The direction calculated is modified so that the search direction is always valid.

This method is the default algorithm and is more robust at finding solutions than the Levenberg-Marquardt method. It is more effective for configurations near joint limits or when the initial guess is not close to the solution. If your initial guess is close to the solution and a quicker solution is needed, consider the "Levenberg-Marquardt" on page 2-10 method.

Levenberg-Marquardt

The Levenberg-Marquardt (LM) algorithm variant used in the `InverseKinematics` class is an error-damped least-squares method. The error-damped factor helps to prevent the algorithm from escaping a local minimum. The LM algorithm is optimized to converge much faster if the initial guess is close to the solution. However the algorithm does not handle arbitrary initial guesses well. Consider using this algorithm for finding IK solutions for a series of poses along a desired trajectory of the end effector. Once a robot configuration is found for one pose, that configuration is often a good initial guess at an IK solution for the next pose in the trajectory. In this situation, the LM algorithm may yield faster results. Otherwise, use the “BFGS Gradient Projection” on page 2-10 instead.

Solver Parameters

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

BFGS Gradient Projection

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

- `MaxIterations` — Maximum number of iterations allowed. The default is 1500.
- `MaxTime` — Maximum number of seconds that the algorithm runs before timing out. The default is 10.
- `GradientTolerance` — Threshold on the gradient of the cost function. The algorithm stops if the magnitude of the gradient falls below this threshold. Must be a positive scalar.
- `SolutionTolerance` — Threshold on the magnitude of the error between the end-effector pose generated from the solution and the desired pose. The weights specified for each component of the pose in the object are included in this calculation. Must be a positive scalar.
- `EnforceJointLimits` — Indicator if joint limits are considered in calculating the solution. `JointLimits` is a property of the robot model in `rigidBodyTree`. By default, joint limits are enforced.
- `AllowRandomRestarts` — Indicator if random restarts are allowed. Random restarts are triggered when the algorithm approaches a solution that does not satisfy the constraints. A randomly generated initial guess is used. `MaxIteration` and `MaxTime` are still obeyed. By default, random restarts are enabled.
- `StepTolerance` — Minimum step size allowed by the solver. Smaller step sizes usually mean that the solution is close to convergence. The default is 10^{-14}.

Levenberg-Marquardt

The solver parameters for the LM algorithm have the following extra fields in addition to what the “BFGS Gradient Projection” on page 2-11 method requires:

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

Solution Information

While using the inverse kinematics algorithms, each call on the object returns solution information about how the algorithm performed. The solution information is provided as a structure with the following fields:

- `Iterations` — Number of iterations run by the algorithm.
- `NumRandomRestarts` — Number of random restarts because algorithm got stuck in a local minimum.
- `PoseErrorNorm` — The magnitude of the pose error for the solution compared to the desired end effector pose.
- `ExitFlag` — Code that gives more details on the algorithm execution and what caused it to return. For the exit flags of each algorithm type, see “Exit Flags” on page 2-12.
- `Status` — Character vector describing whether the solution is within the tolerance (`'success'`) or the best possible solution the algorithm could find (`'best available'`).

Exit Flags

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

`'BFGSGradientProjection'` algorithm exit flags:

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

`'LevenbergMarquardt'` algorithm exit flags:

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

References

[1] Badreddine, Hassan, Stefan Vandewalle, and Johan Meyers. "Sequential Quadratic Programming (SQP) for Optimal Control in Direct Numerical Simulation of Turbulent Flow." *Journal of Computational Physics*. 256 (2014): 1–16. doi:10.1016/j.jcp.2013.08.044.

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

[3] Goldfarb, Donald. "Extension of Davidon’s Variable Metric Method to Maximization Under Linear Inequality and Equality Constraints." *SIAM Journal on Applied Mathematics*. Vol. 17, No. 4 (1969): 739–64. doi:10.1137/0117067.

[4] Nocedal, Jorge, and Stephen Wright. *Numerical Optimization*. New York, NY: Springer, 2006.

[5] Sugihara, Tomomichi. "Solvability-Unconcerned Inverse Kinematics by the Levenberg–Marquardt Method." *IEEE Transactions on Robotics* Vol. 27, No. 5 (2011): 984–91. doi:10.1109/tro.2011.2148230.

[6] Zhao, Jianmin, and Norman I. Badler. "Inverse Kinematics Positioning Using Nonlinear Programming for Highly Articulated Figures." *ACM Transactions on Graphics* Vol. 13, No. 4 (1994): 313–36. doi:10.1145/195826.195827.

See Also

`rigidBodyTree` | `generalizedInverseKinematics` | `inverseKinematics`

Related Examples

- “2-D Path Tracing with Inverse Kinematics” on page 2-14
- “Control PR2 Arm Movements Using ROS Actions and Inverse Kinematics” on page 1-193
- “Rigid Body Tree Robot Model” on page 2-2

2-D Path Tracing with Inverse Kinematics

Introduction

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

Construct The Robot

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

Start with a blank rigid body tree model.

```
robot = rigidBodyTree('DataFormat','column','MaxNumBodies',3);
```

Specify arm lengths for the robot arm.

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

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

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

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

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

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

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

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

```
showdetails(robot)
```

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

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

Define The Trajectory

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

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

Inverse Kinematics Solution

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

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

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

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

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

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

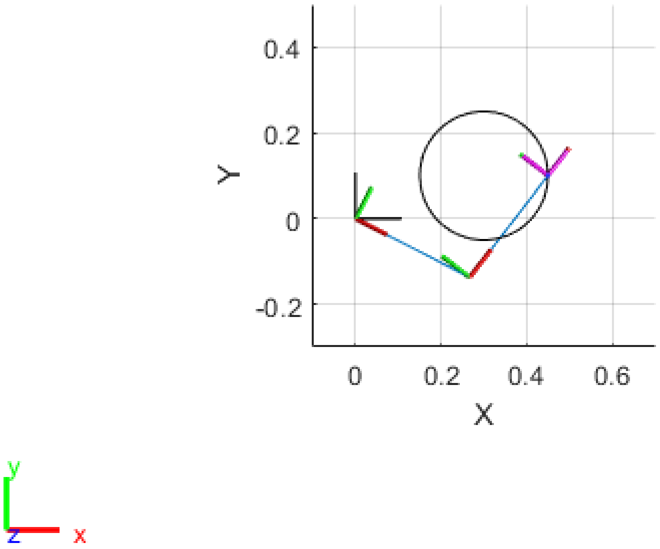
```
qInitial = q0; % Use home configuration as the initial guess
for i = 1:count
    % Solve for the configuration satisfying the desired end effector
    % position
    point = points(i,:);
    qSol = ik(endEffector,trvec2tform(point),weights,qInitial);
    % Store the configuration
    qs(i,:) = qSol;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

Animate The Solution

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

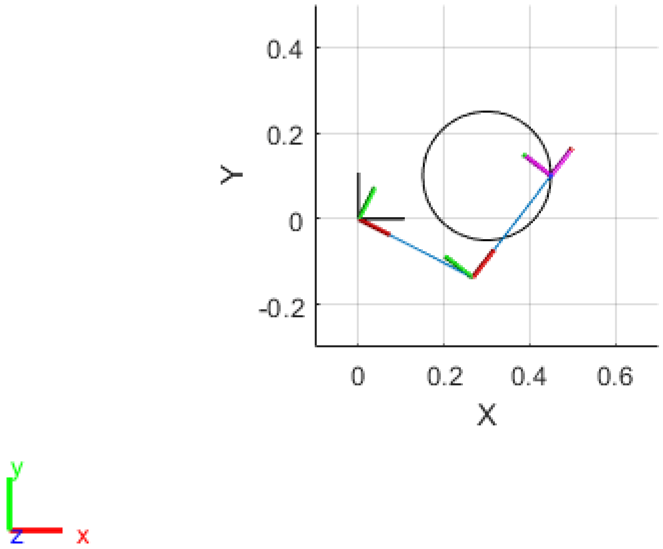
Show the robot in the first configuration of the trajectory. Adjust the plot to show the 2-D plane that circle is drawn on. Plot the desired trajectory.

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



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

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



See Also

`rigidBodyTree` | `rigidBody` | `rigidBodyJoint` | `inverseKinematics`

Related Examples

- “Control PR2 Arm Movements Using ROS Actions and Inverse Kinematics” on page 1-193
- “Inverse Kinematics Algorithms” on page 2-10

Solve Inverse Kinematics for a Four-Bar Linkage

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

Initialize the four-bar linkage rigid body tree model.

```
robot = rigidBodyTree('Dataformat','column','MaxNumBodies',7);
```

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

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

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

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

```
for k = 1:6

    b = rigidBody(bodyNames{k});
    b.Joint = rigidBodyJoint(jointNames{k},jointTypes{k});

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

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

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

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

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

Specify kinematic constraints for the `GeneralizedInverseKinematics` object:

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

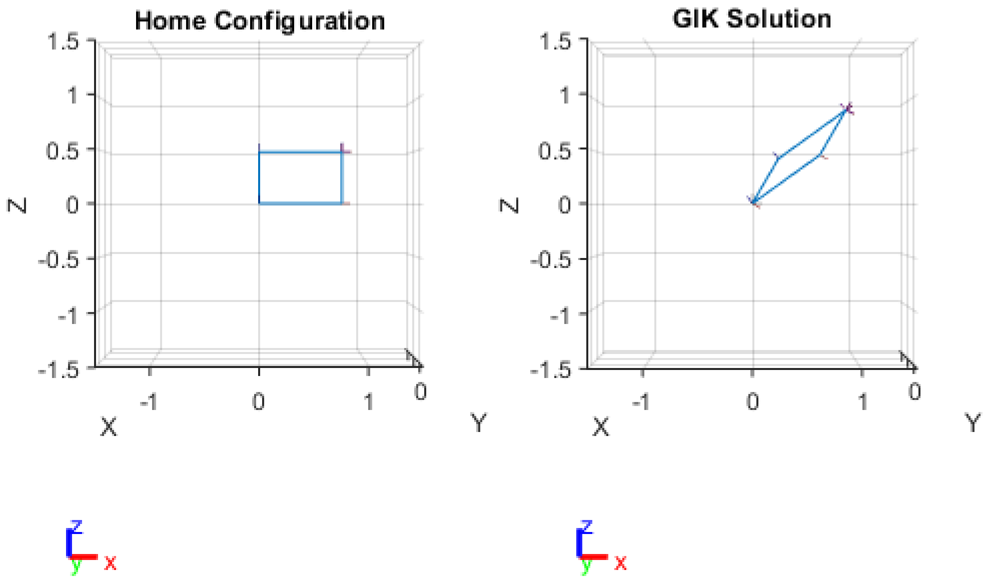
```
gik = generalizedInverseKinematics('RigidBodyTree',robot);
gik.ConstraintInputs = {'position',...  % Position constraint for closed-loop mechanism
                        'position',...  % Position constraint for end-effector
                        'joint'};       % Joint limits
gik.SolverParameters.AllowRandomRestart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

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

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

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

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

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

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

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

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



See Also

Classes

`generalizedInverseKinematics` | `rigidBodyTree` | `inverseKinematics` | `constraintPoseTarget` | `constraintJointBounds`

Related Examples

- “Rigid Body Tree Robot Model” on page 2-2
- “Plan a Reaching Trajectory With Multiple Kinematic Constraints” on page 1-204

Robot Dynamics

In this section...

This topic details the different elements, properties, and equations of rigid body robot dynamics. Robot dynamics are the relationship between the forces acting on a robot and the resulting motion of the robot. In Robotics System Toolbox, manipulator dynamics information is contained within a `rigidBodyTree` object, which specifies the rigid bodies, attachment points, and inertial parameters for both kinematics and dynamics calculations.

Note To use dynamics object functions, you must set the `DataFormat` property of the `rigidBodyTree` object to `"row"` or `"column"`. These setting accept inputs and return outputs as row or column vectors, respectively, for relevant robotics calculations, such as robot configurations or joint torques.

Dynamics Properties

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

- `Mass` — Mass of the rigid body in kilograms.
- `CenterOfMass` — Center of mass position of the rigid body, specified as a vector of the form `[x y z]`. The vector describes the location of the center of mass of the rigid body, relative to the body frame, in meters. The `centerOfMass` object function uses these rigid body property values when computing the center of mass of a robot.
- `Inertia` — Inertia of the rigid body, specified as a vector of the form `[Ixx Iyy Izz Iyz Ixz Ixy]`. The vector is relative to the body frame in kilogram square meters. The inertia tensor is a positive definite matrix of the form:

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

  The first three elements of the `Inertia` vector are the moment of inertia, which are the diagonal elements of the inertia tensor. The last three elements are the product of inertia, which are the off-diagonal elements of the inertia tensor.

For information related to the entire manipulator robot model, specify these `rigidBodyTree` object properties:

- `Gravity` — Gravitational acceleration experienced by the robot, specified as an `[x y z]` vector in m/s^2. By default, there is no gravitational acceleration.
- `DataFormat` — The input and output data format for the kinematics and dynamics functions, specified as `"struct"`, `"row"`, or `"column"`.

Dynamics Equations

Manipulator rigid body dynamics are governed by this equation:

$$\frac{d}{dt}\begin{bmatrix} q \\ \dot{q} \end{bmatrix} = \begin{bmatrix} \dot{q} \\ M(q)^{-1}\left(-C(q,\dot{q})\dot{q} - G(q) - J(q)^T F_{Ext} + \tau\right) \end{bmatrix}$$

also written as:

$$M(q)\ddot{q} = -C(q,\dot{q})\dot{q} - G(q) - J(q)^T F_{Ext} + \tau$$

where:

- $M(q)$ — is a joint-space mass matrix based on the current robot configuration. Calculate this matrix by using the `massMatrix` object function.
- $C(q, \dot{q})$ — is the coriolis terms, which are multiplied by $\dot{q}$ to calculate the velocity product. Calculate the velocity product by using by the `velocityProduct` object function.
- $G(q)$ — is the gravity torques and forces required for all joints to maintain their positions in the specified gravity `Gravity`. Calculate the gravity torque by using the `gravityTorque` object function.
- $J(q)$ — is the geometric Jacobian for the specified joint configuration. Calculate the geometric Jacobian by using the `geometricJacobian` object function.
- F_{Ext} — is a matrix of the external forces applied to the rigid body. Generate external forces by using the `externalForce` object function.
- τ — are the joint torques and forces applied directly as a vector to each joint.
- $q, \dot{q}, \ddot{q}$ — are the joint configuration, joint velocities, and joint accelerations, respectively, as individual vectors. For revolute joints, specify values in radians, rad/s, and rad/s^2, respectively. For prismatic joints, specify in meters, m/s, and m/s^2.

To compute the dynamics directly, use the `forwardDynamics` object function. The function calculates the joint accelerations for the specified combinations of the above inputs.

To achieve a certain set of motions, use the `inverseDynamics` object function. The function calculates the joint torques required to achieve the specified configuration, velocities, accelerations, and external forces.

See Also

Functions
`forwardDynamics` | `inverseDynamics` | `externalForce` | `geometricJacobian` | `gravityTorque` | `centerOfMass` | `massMatrix` | `velocityProduct`

Objects
`rigidBodyTree` | `jointSpaceMotionModel` | `taskSpaceMotionModel` | `inverseKinematics`

Related Examples

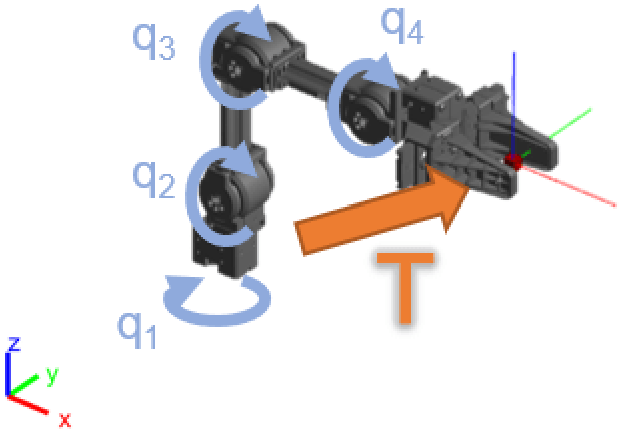
- “Control PR2 Arm Movements Using ROS Actions and Inverse Kinematics” on page 1-193

Task-Space Motion Model

The *task-space motion model* characterizes the closed-loop motion of a manipulator under task-space control, where the control action is defined in the SE(3) space with respect to the pose of a specified end effector. This topic covers the variables and equations for computing the behavior of joint position, velocity, and acceleration given motion modeled in the “Plan and Execute Task- and Joint-Space Trajectories Using KINOVA Gen3 Manipulator” on page 1-257 object. For joint-space motion models, see the `jointSpaceMotionModel` object.

For an example that covers the difference between task-space and joint-space control, see “Plan and Execute Task- and Joint-Space Trajectories Using KINOVA Gen3 Manipulator” on page 1-257.

- Key Variables on page 2-0
- Equations of Motion on page 2-0



Key Variables

The model state consists of these values:

- q — Robot joint configuration, as a vector of joint positions. Specified in *rad* for revolute joints and *m* for prismatic joints.
- $\dot{q}$ — Vector of joint velocities in $rad \cdot s^{-1}$ for revolute joints and $m \cdot s^{-1}$ for prismatic joints
- $\ddot{q}$ — Vector of joint accelerations in $rad \cdot s^{-2}$ for revolute joints or $m \cdot s^{-2}$ for prismatic joints

The end-effector pose $T(q)$ of the robot is a 4-by-4 homogeneous matrix defined relative to the origin at the robot base. Positions are in meters. Two forms of T are used for calculating errors in control:

- T_{ref} — Reference end-effector pose, specified as a desired end-effector pose
- T_{act} — Actual end-effector pose achieved by the motion

The end-effector transform decomposes as:

$$T = \begin{bmatrix} R & X \\ 0 & 1 \end{bmatrix}$$

where R is the orientation as a 3-by-3 rotation matrix, and X is a 3-by-1 vector of *xyz*-positions in meters.

The task-space velocity v and acceleration a consist of two 6-by-1 vectors:

$$v = \begin{bmatrix} \omega \\ \dot{X} \end{bmatrix}, a = \begin{bmatrix} \alpha \\ \ddot{X} \end{bmatrix}$$

where ω and α are 3-by-1 vectors of angular velocities and accelerations of the frame, respectively.

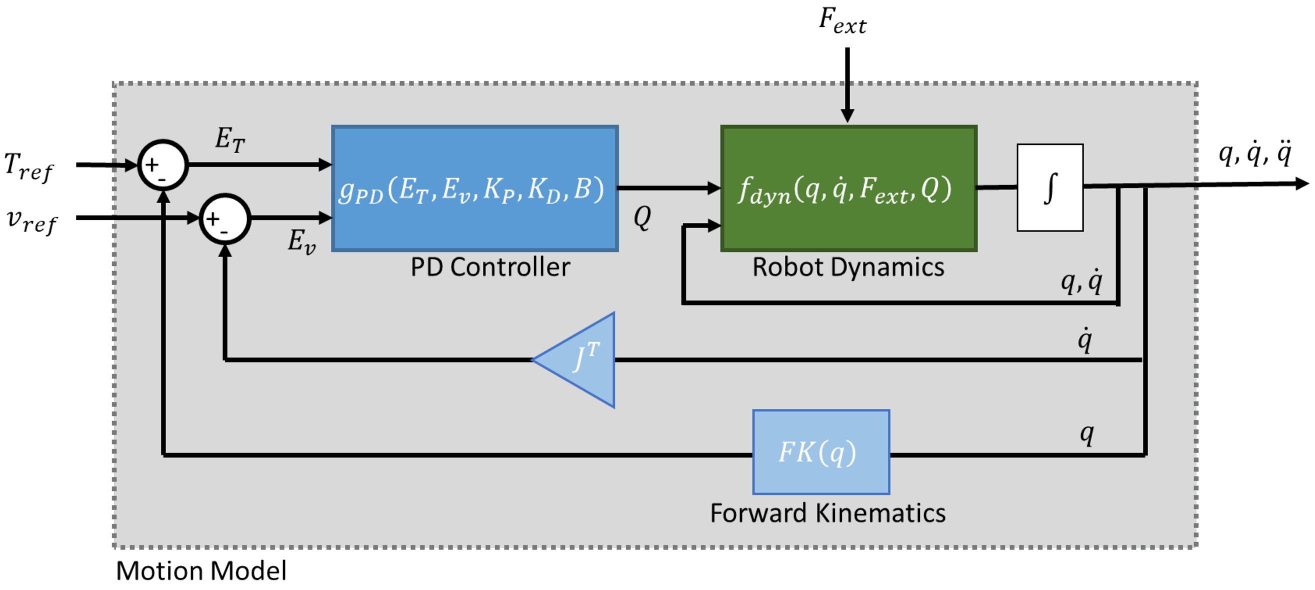
Equations of Motion

Use the task-space motion model to represent robots that are subject to a control law that acts on the task-space error. For example, when the input to the control law specifies end-effector motion. While there are many ways to implement such a system, in this model, the closed-loop response is approximated in the “Plan and Execute Task- and Joint-Space Trajectories Using KINOVA Gen3 Manipulator” on page 1-257 object by providing a system under proportional-derivative Jacobian-Transpose style control. Regardless of how the system you are using works, this model can be used as a low-fidelity approximation of a system under closed-loop task-space control.

Proportional-Derivative Control

When the motion model uses proportional-derivative (PD) control, as determined by the “MotionType” property of the `taskSpaceMotionModel` object, the model computes forward dynamics using standard rigid body dynamics, but with subject to a PD control law that acts on the error between desired and actual end-effector pose.

- Inputs — This model accepts reference pose T_{ref} and reference end-effector velocities v_{ref}
- Outputs — The model returns the $q, \dot{q}, \ddot{q}$ as the joint configuration, velocities, and accelerations as vectors
- Complexity — This is a medium complexity motion model. It uses complete rigid body dynamics, but the control law used in the model is relatively simple.



In this system, the joint positions, velocities, and accelerations are computed using standard rigid-body “Robot Dynamics” on page 2-21. The generalized force input Q is given by the PD Control law on the task-space error, scaled to the joint-space via a Jacobian-Transpose style control:

$$\frac{d}{dt}\begin{bmatrix} q \\ \dot{q} \end{bmatrix} = f_{dyn}(q, \dot{q}, Q, F_{ext})$$

$$Q = J(q)^T(K_P E_T + K_D E_v) - B\dot{q} + G(q)$$

$$E_T = \begin{bmatrix} e_{rot} \\ e_{trans} \end{bmatrix}$$

$$E_v = \left(v_{ref} - J(q)\dot{q}\right)$$

where:

- e_{rot} — is the rotational error converted to Euler angles using `rotm2eul(`$R_{ref}R_{act}^T$`)`
- e_{pos} — is the error in *xyz*-coordinates, calculated as $X_{ref} - X_{act}$.
- $G(q)$ — are the gravity torques and forces for all joints to maintain their positions in the specified gravity. For more information, see the `gravityTorque` function.
- $J(q)$ — is the geometric Jacobian for the given joint configuration for more information, see the `geometricJacobian` function.

The control input relies on these user-defined parameters:

- K_P — Proportional gain, specified as a 6-by-6 matrix
- K_D — Derivative gain, specified as a 6-by-6 matrix
- B — Joint damping vector, specified as a two-element vector of damping constants in $N \cdot s \cdot rad^{-1}$ for revolute joints and $N \cdot s \cdot m^{-1}$ for prismatic joints

You can specify these parameters as properties on the “Plan and Execute Task- and Joint-Space Trajectories Using KINOVA Gen3 Manipulator” on page 1-257 object.

This model accepts the following inputs:

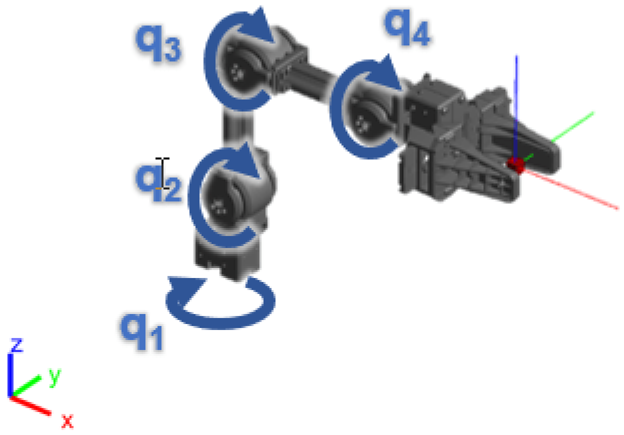
- T_{ref} — Reference end-effector pose, specified as the desired end-effector pose
- v_{ref} — Reference end-effector velocities, specified as a vector $v = \begin{bmatrix} \omega \\ \dot{X} \end{bmatrix}$, with angular velocities ω and translational velocities $\dot{X}$

Joint-Space Motion Model

The joint-space motion model characterizes the closed-loop motion of a manipulator under joint-space control, where the control action is defined in the joint configuration space. Motion models are used as low-fidelity plant models of robots under closed-loop position control. This topic covers the variables and equations for computing the behavior of the joint-space position, velocity, and accelerations relative to reference inputs, as used in the `jointSpaceMotionModel` object. For task-space motion models, see the "Plan and Execute Task- and Joint-Space Trajectories Using KINOVA Gen3 Manipulator" on page 1-257 object.

This topic covers these types of joint-space control:

- Computed Torque Control on page 2-0
- Proportional-Derivative (PD) Control on page 2-0
- Independent Joint Motion on page 2-0



For an example that covers the difference between task-space and joint-space control, see "Plan and Execute Task- and Joint-Space Trajectories Using KINOVA Gen3 Manipulator" on page 1-257.

State and Controls

The joint-space motion model state consists of these values:

- q — Robot joint configuration, as a vector of joint positions. Specified in rad for revolute joints and m for prismatic joints.
- $\dot{q}$ — Vector of joint velocities in $rad \cdot s^{-1}$ for revolute joints and $m \cdot s^{-1}$ for prismatic joints
- $\ddot{q}$ — Vector of joint accelerations in $rad \cdot s^{-2}$ for revolute joints or $m \cdot s^{-2}$ for prismatic joints

The joint-space motion model is used when you need a low-fidelity model of your system under closed-loop control and the inputs are specified as joint configuration, velocity, and acceleration. The motion model includes three ways to model its overall behavior:

- Computed Torque Control — The rigid-body dynamics are modeled using the standard equations of motion, but compensating for the full-body dynamics and assigning error dynamics. This is a higher-fidelity version of independent joint motion control.

- PD Control — The rigid-body dynamics are modeled using the standard equations of motion with a joint torque input given by proportional-derivative (PD) control. This model represents a controller that does not compensate tightly for the overall effects of rigid-body motion.
- Independent Joint Motion — Each joint is modeled independently as a closed-loop second-order system. This model is a lower fidelity version of computed torque control motion model, and may be considered a best-case scenario for how closed-loop motion may behave since the dynamics are simplified and directly prescribed.

To set these different motion types, use the “MotionType” property of the `jointSpaceMotionModel` object. These motion types are not exhaustive, but they do provide a set of options to use when approximating the closed-loop behavior of your system. For details and suggestions on when to use which model, see the sections below.

Equations of Motion

In this section, the equations of motion for each model are introduced, in order of decreasing complexity.

Computed Torque Control

With computed torque control, the motion model uses standard rigid body dynamics, but the generalized force input is given by a control law that compensates for the rigid body dynamics and instead assigns a second-order error dynamics response.

- Inputs — This model accepts q_{ref}, $\dot{q}_{ref}$, $\ddot{q}_{ref}$ as the desired reference joint configuration, velocities, and accelerations as vectors. The user may also optional provide the external force F_{ext}, specified in Newtons.
- Outputs — The model outputs q, $\dot{q}$, $\ddot{q}$ as the joint configuration, velocities, and accelerations as vectors. In the MATLAB version of the model, only accelerations are returned, and the user must choose an integrator or ODE solver to return the other states.
- Complexity — This is high complexity. The motion model uses full rigid body dynamics with optional external forces, the controller is modeled as part of the closed loop system, and the controller includes dynamic compensation terms.
- When to apply — Use when the closed-loop system being simulated has approximable error dynamics, or when it uses a controller that treats the robot as a multi-body system, and external forces may be present

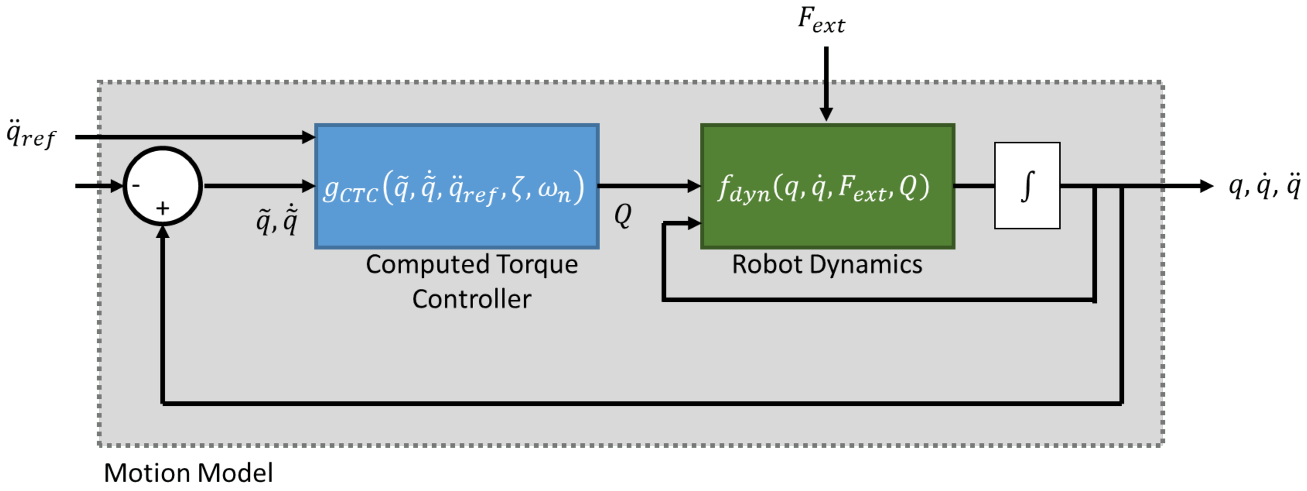
The resultant closed-loop system aims to achieve the following second error behavior for the *i*-th joint:

$$\ddot{\tilde{q}}_i = -\omega_n^2\tilde{q}_i - 2\zeta\omega_n\dot{\tilde{q}}_i$$

$$\tilde{q}_i = q_i - q_{i\,.\,ref}$$

These parameters characterize the desired response defined for each joint:

- ω_n — Natural frequency, specified in Hz (s^{-1})
- ζ — The damping ratio, which is unitless


F_{ext}
$\ddot{q}_{ref}$
-
+
$\tilde{q}, \dot{\tilde{q}}$
$g_{CTC}(\tilde{q}, \dot{\tilde{q}}, \ddot{q}_{ref}, \zeta, \omega_n)$
Computed Torque Controller
Q
$f_{dyn}(q, \dot{q}, F_{ext}, Q)$
Robot Dynamics
$\int$
$q, \dot{q}, \ddot{q}$
Motion Model

As seen in the diagram, the complete system consists of the standard rigid-body “Robot Dynamics” on page 2-21 with a control law that enforces closed error dynamics via the generalized force input Q:

$$\frac{d}{dt}\begin{bmatrix} q \\ \dot{q} \end{bmatrix} = f_{dyn}(q, \dot{q}, Q, F_{ext})$$

$$Q = g_{CTC}(\tilde{q}, \dot{\tilde{q}}, \ddot{q}_{ref}, \omega_n, \zeta) = M(q)a_q + C(q, \dot{q})\dot{q} + G(q)$$

$$a_q = \ddot{q}_{ref} - \left[\omega_n^2\right]_{diag}\tilde{q} - [2\zeta\omega_n]_{diag}\dot{\tilde{q}}$$

$$\tilde{q} = q - q_{ref}$$

where,

- $M(q)$ — is a joint-space mass matrix based on the current robot configuration Calculate this matrix by using the `massMatrix` object function.
- $C(q, \dot{q})$ — is the coriolis terms, which are multiplied by $\dot{q}$ to calculate the velocity product. Calculate the velocity product by using by the `velocityProduct` object function.
- $G(q)$ — is the gravity torques and forces required for all joints to maintain their positions in the specified gravity “Gravity”. Calculate the gravity torque by using the `gravityTorque` object function.

The control input relies on these user-defined parameters:

- $\left[-\omega_n^2\right]_{diag}$ — Diagonal matrix, where the (i,i)-th element corresponds to the i-th element of the n-element vector of natural frequencies in the “NaturalFrequency” property of the `jointSpaceMotionModel` object are in Hz (s^{-1}).
- $\left[-2\zeta\omega_n^2\right]_{diag}$ — Diagonal matrix, where the (i,i)-th element corresponds to the i-th element of the product of the squared natural frequencies vector ω_n and the i-th element of the damping ratios vector ζ, specified in the “DampingRatio” property of the `jointSpaceMotionModel` object.

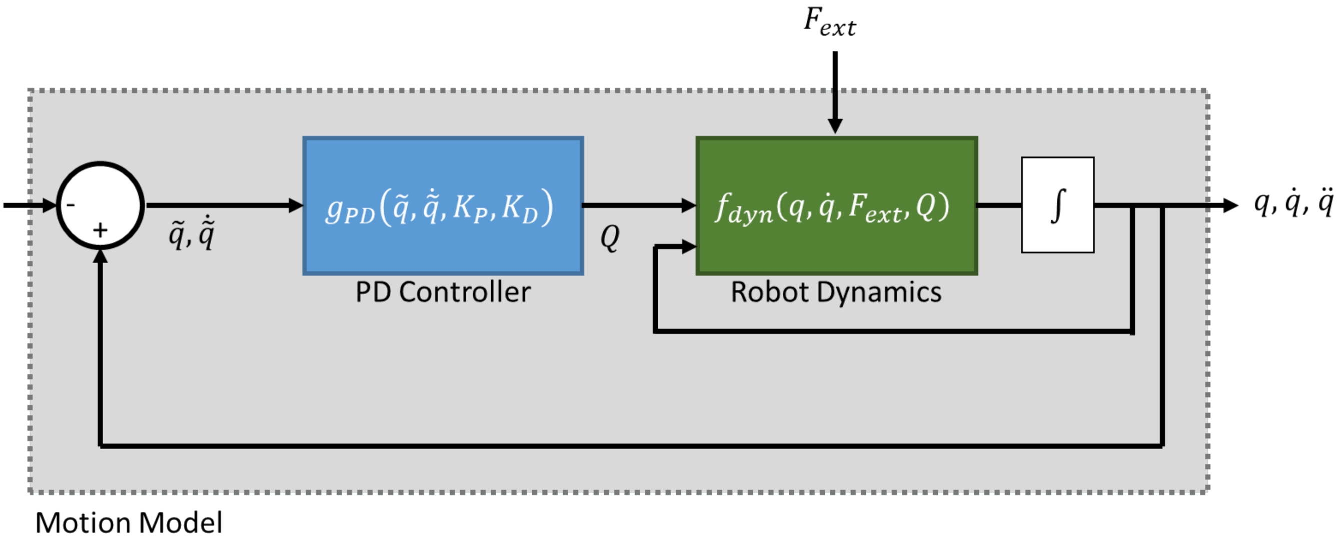
Because the dynamics are compensated, in the absence of external force inputs large acceleration/deceleration, the error dynamics should be achieved. In the absence of external forces, the independent joint motion type provides a simpler way of achieving this result.

The values of ω_n and ζ may be set directly, or they may be provided using the the `updateErrorDynamicsFromStep` method, which computes values for ω_n and ζ based on desired unit step response (defined using it's transient behavior characteristics).

Proportional-Derivative (PD) Control

With PD control, the robot models behavior according to standard rigid body dynamics, but with the generalized force input Q given by a control law that applies PD control based on the joint error, as well as gravity compensation.

- Inputs — This model accepts $q_{ref}, \dot{q}_{ref}$ as the desired reference joint configuration, velocities, and accelerations as vectors. The user may also optional provide the external force F_{ext}, specified in Newtons.
- Outputs — The model outputs $q, \dot{q}, \ddot{q}$ as the joint configuration, velocities, and accelerations. In the MATLAB version of the model, only accelerations are returned, and the user must choose an integrator or ODE solver to return the other states.
- Complexity — This is medium complexity. The motion model uses full rigid body dynamics with optional external forces and the controller is modeled as part of the closed loop system, but the controller is relatively simple.
- When to apply — Use when the closed-loop system being simulated uses a controller that treats joints as independent systems, or when a PD style controller is used, and external forces may be present.


F_{ext}
-
+
$\tilde{q}, \dot{\tilde{q}}$
$g_{PD}(\tilde{q}, \dot{\tilde{q}}, K_P, K_D)$
PD Controller
Q
$f_{dyn}(q, \dot{q}, F_{ext}, Q)$
Robot Dynamics
∫
$q, \dot{q}, \ddot{q}$
Motion Model

As with computed torque control, this system behavior uses the standard rigid-body “Robot Dynamics” on page 2-21, but uses the PD control law define the generalized force input Q:

$$\frac{d}{dt}\begin{bmatrix} q \\ \dot{q} \end{bmatrix} = f_{dyn}(q, \dot{q}, \tau, F_{ext})$$

$$Q = g_{PD}(\widetilde{q}, \dot{\widetilde{q}}, K_P, K_D) = -K_P(\widetilde{q}) - K_D(\dot{\widetilde{q}}) + G(q)$$

$$\widetilde{q} = q - q_{ref}$$

where

- $G(q)$ — is the gravity torques and forces required for all joints to maintain their positions in the specified gravity “Gravity”. Calculate the gravity torque by using the `gravityTorque` object function.

The control input relies on these user-defined parameters:

- K_P — Proportional gain, specified as an N-by-N matrix, where N is the number of movable joints of the manipulator
- K_D — Derivative gain, specified as an N-by-N matrix

Independent Joint Motion

When this system is modeled with independent joint motion, instead of modeling the closed loop system as standard rigid body dynamics plus a control input, each joint is instead modeled as a second-order system that already has the desired error behavior:

- Inputs — This model accepts $q_{ref}, \dot{q}_{ref}$ as the desired reference joint configuration, velocities, and accelerations as vectors. There is no external force input.
- Outputs — The model outputs $q, \dot{q}, \ddot{q}$ as the joint configuration, velocities, and accelerations. In the MATLAB version of the model, only accelerations are returned, and the user must choose an integrator or ODE solver to return the other states.
- Complexity — This is low complexity. The motion model simply prescribes the error behavior that a position controller could aim to achieve.
- When to apply — Use when the system has approximable error dynamics and there are no external force inputs required.

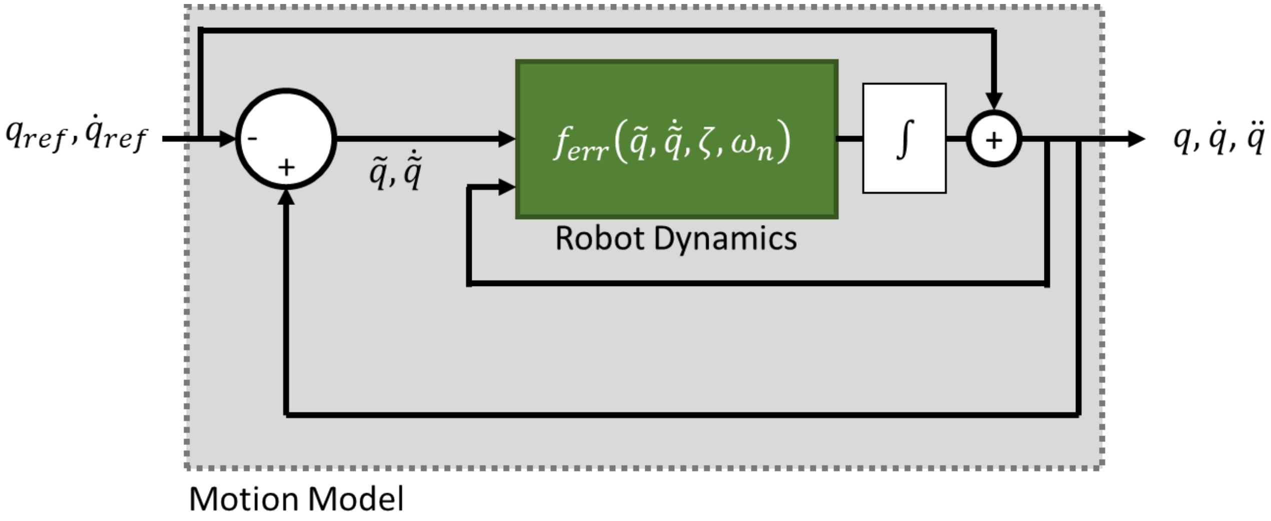
The system models the following closed-loop second order behavior for the ith joint:

$$\frac{d}{dt}\begin{bmatrix} \tilde{q} \\ \dot{\tilde{q}} \end{bmatrix} = f_{err}(\tilde{q}, \dot{\tilde{q}}, \zeta, \omega_n) = \begin{bmatrix} \dot{\tilde{q}} \\ -\omega_n^2 \tilde{q}_i - 2\zeta\omega_n \dot{\tilde{q}}_i \end{bmatrix}$$

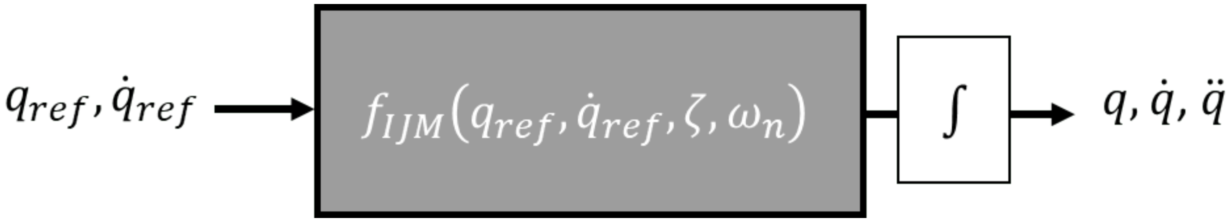
$$\tilde{q}_i = q_i - q_{i\,.\,ref}$$

These parameters characterize the desired response defined for each joint:

- ω_n — the natural frequency specified in units of s^{-1}
- ζ — t the damping ratio, which is unitless


$q_{ref}, \dot{q}_{ref}$
-
+
$\tilde{q}, \dot{\tilde{q}}$
$f_{err}(\tilde{q}, \dot{\tilde{q}}, \zeta, \omega_n)$
$\int$
+
$q, \dot{q}, \ddot{q}$
Robot Dynamics
Motion Model

Or as:



The complete system is therefore modeled as:

$$\frac{d}{dt}\begin{bmatrix} q \\ \dot{q} \end{bmatrix} = f_{IJM}(q_{ref}, \dot{q}_{ref}, \zeta, \omega_n) = \begin{bmatrix} 0 & I \\ [-\omega_n^2]_{diag} & [-2\zeta\omega_n]_{diag} \end{bmatrix}\begin{bmatrix} q \\ \dot{q} \end{bmatrix} + \begin{bmatrix} 0 & I \\ [\omega_n^2]_{diag} & [2\zeta\omega_n]_{diag} \end{bmatrix}\begin{bmatrix} q_{ref} \\ \dot{q}_{ref} \end{bmatrix}$$

The model relies on these user-defined parameters:

- $[-\omega_n^2]_{diag}$ — Diagonal matrix, where the (i, i)-th element corresponds to the i-th element of the n-element vector of natural frequencies in the “NaturalFrequency” property of the `jointSpaceMotionModel` object are in Hz (s^{-1}).
- $[-2\zeta\omega_n^2]_{diag}$ — Diagonal matrix, where the (i, i)-th element corresponds to the i-th element of the product of the squared natural frequencies vector ω_n and the i-th element of the damping ratios vector ζ, specified in the “DampingRatio” property of the `jointSpaceMotionModel` object.

The values of ω_n and ζ may be set directly, or they may be provided using the the `updateErrorDynamicsFromStep` method, which computes values for ω_n and ζ based on desired unit step response (defined using it's transient behavior characteristics).

The Independent Joint Motion model represents a closed loop system under idealized behavior. In the absence of external forces, the motion model using computed torque control should produce an equivalent output.

Install Robotics System Toolbox Robot Library Data Support Package

The Robotics System Toolbox Robot Library Data Support Package provides source mesh files to visualize and simulate robots in the robot library on platforms besides MATLAB and Simulink.

To install the support package.

1. In a MATLAB Command Window, type:

   ```
   roboticsAddons
   ```

   The Add-On Explorer starts.
2. Select **Robotics System Toolbox Robot Library Data**.
3. On the support package page, click the **Install** button.
4. Alternatively, you can download and install the Robotics System Toolbox Robot Library Data support package from File Exchange.

To check for updates, repeat this process when a new version of MATLAB is released. You can also check for updates between releases using this process.

See Also

`importrobot`

Related Examples

- “Design Position Controlled Manipulator Using Simscape” on page 1-98
- “Perform Trajectory Tracking and Compute Joint Torque for Manipulator Using Simscape” on page 1-110

Mobile Robot Kinematics Equations

Learn details about mobile robot kinematics equations including unicycle, bicycle, differential drive, and Ackermann models. This topic covers the variables and specific equations for each motion model [1]. For an example that simulates the different mobile robots using these models, see “Simulate Different Kinematic Models for Mobile Robots” on page 1-67.

Variable Overview

The robot state is represented as a three-element vector: [x y θ].

For a given robot state:

- x: Global vehicle x-position in meters
- y: Global vehicle y-position in meters
- θ: Global vehicle heading in radians

For Ackermann kinematics, the state also includes steering angle:

- ψ: Vehicle steering angle in radians

The unicycle, bicycle, and differential drive models share a genrealized control input, which accepts the following:

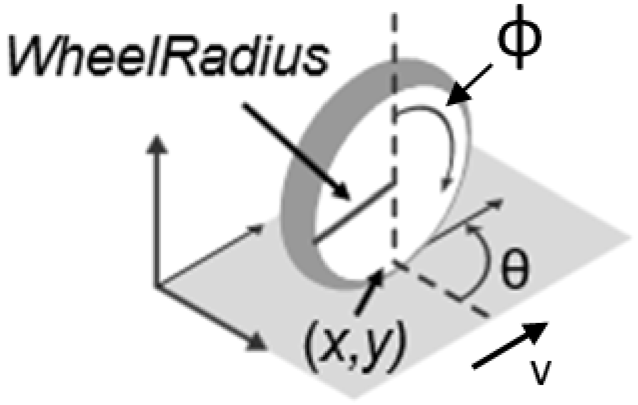
- v: Vehicle speed in meters/s
- ω: Vehicle angular velocity in radians/s

Other variables represented in the kinematics equations are:

- r: Wheel radius in meters
- $\dot{\phi}$: Wheel speed in radians/s
- d: Track width in meters
- l: Wheel base in meters
- ψ: Vehicle steering angle in radians

Unicycle Kinematics

The unicycle kinematics equations model a single rolling wheel that pivots about a central axis using the `unicycleKinematics` object.



The unicycle model state is [x y θ].

Variables

- x: Global vehicle x-position in meters
- y: Global vehicle y-position in meters
- θ: Global vehicle heading in radians
- $\dot{\phi}$: Wheel speed in meters/s
- r: Wheel radius in meters
- v: Vehicle speed in meters/s
- ω: Vehicle heading angular velocity in radians/s

Kinematic Equations

Depending on the `VehicleInputs` name-value argument, you can input only wheel speeds or the vehicle speed and heading rate. This change in input affects the equations.

Wheel Speed Equation

$$\begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{bmatrix} = \begin{bmatrix} r\cos(\theta) & 0 \\ r\sin(\theta) & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} \dot{\phi} \\ \omega \end{bmatrix}$$

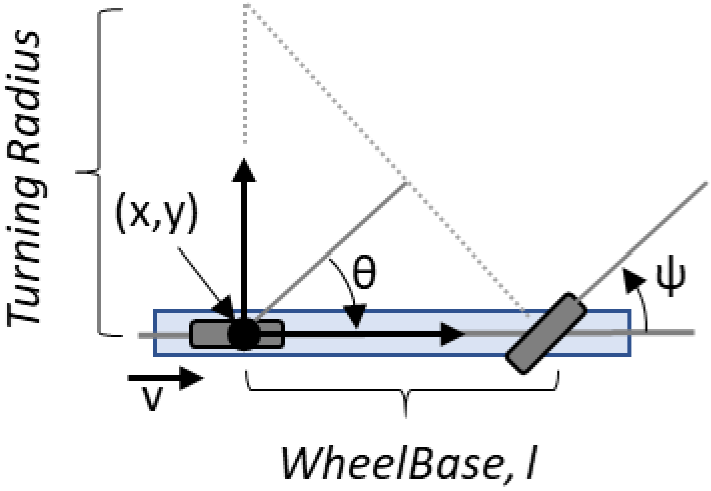
Vehicle Speed and Heading Rate Equation (Generalized)

When the generalized inputs are given as the speed $v = r\dot{\phi}$ and vehicle heading angular velocity ω, the equation simplifies to:

$$\begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{bmatrix} = \begin{bmatrix} \cos(\theta) & 0 \\ \sin(\theta) & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} v \\ \omega \end{bmatrix}$$

Bicycle Kinematics

The bicycle kinematics equations model a car-like vehicle that accepts the front steering angle as a control input using the `bicycleKinematics` object.



The bicycle model state is [x y θ].

Variables

- x: Global vehicle x-position in meters
- y: Global vehicle y-position in meters
- θ: Global vehicle heading in radians
- l: Wheel base, in meters
- ψ: Vehicle steering angle in radians
- v: Vehicle speed in meters/s
- ω: Vehicle heading angular velocity in radians/s

Kinematic Equations

Depending on the `VehicleInputs` name-value argument, you can input the vehicle speed as either the steering angle or heading rate. This change in input affects the equations.

Steering Angle Equation

$$\begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{bmatrix} = \begin{bmatrix} v\cos(\theta) \\ v\sin(\theta) \\ \frac{v}{l}\tan(\psi) \end{bmatrix} \begin{bmatrix} v \\ \omega \end{bmatrix}$$

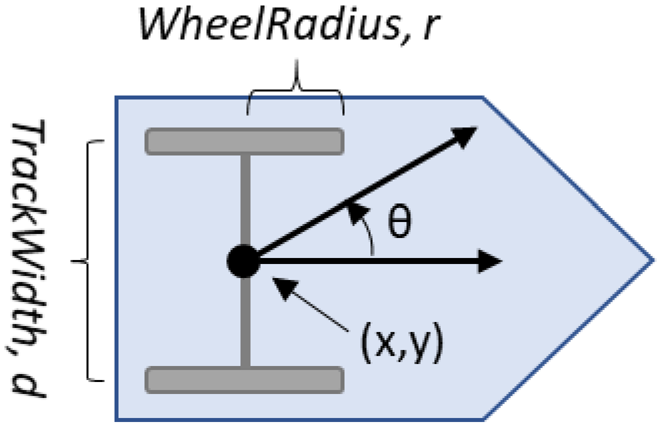
Vehicle Speed and Heading Rate Equation (Generalized)

In this generalized format, the heading rate ω can be related to the steering angle ψ with the relation $\omega = \frac{v}{l}\tan\psi$. Then, the ODE simplifies to:

$$\begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{bmatrix} = \begin{bmatrix} \cos(\theta) & 0 \\ \sin(\theta) & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} v \\ \omega \end{bmatrix}$$

Differential Drive Kinematics

The differential drive kinematics equations model a vehicle where the wheels on the left and right may spin independently using the `differentialDriveKinematics` object.



The differential drive model state is [x y θ].

Variables

- x: Global vehicle x-position, in meters
- y: Global vehicle y-position, in meters
- θ: Global vehicle heading, in radians
- $\dot{\phi}_L$: Left wheel speed in meters/s
- $\dot{\phi}_R$: Right wheel speed in meters/s
- r: Wheel radius in meters
- d: Track width in meters
- v: Vehicle speed in meters/s
- ω: Vehicle heading angular velocity in radians/s

Kinematic Equations

Depending on the `VehicleInputs` name-value argument, you can input the wheel speed as either the steering angle or heading rate. This change in input affects the equations.

Wheel Speed Equation

$$\begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{bmatrix} = \begin{bmatrix} \frac{r}{2}\cos(\theta) & \frac{r}{2}\cos(\theta) \\ \frac{r}{2}\sin(\theta) & \frac{r}{2}\sin(\theta) \\ -r/2d & r/2d \end{bmatrix} \begin{bmatrix} \dot{\phi}_L \\ \dot{\phi}_R \end{bmatrix}$$

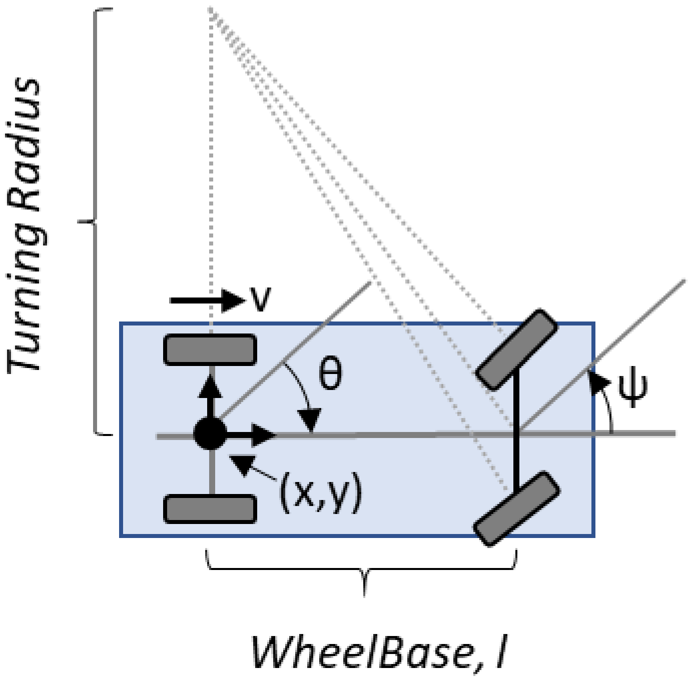
Vehicle Speed and Heading Rate Equation (Generalized)

In the generalized format, the inputs are given as the speed $v = \frac{r}{2}\left(\dot{\phi}_R + \dot{\phi}_L\right)$ and vehicle heading angular velocity $\omega = \frac{r}{2d}\left(\dot{\phi}_R - \dot{\phi}_L\right)$. The ODE simplifies to:

$$\begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{bmatrix} = \begin{bmatrix} \cos(\theta) & 0 \\ \sin(\theta) & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} v \\ \omega \end{bmatrix}$$

Ackermann Kinematics

The Ackermann kinematic equations model a car-like vehicle model with an Ackermann-steering mechanism using the `ackermannKinematics` object. The equation adjusts the position of the axle tires based on the track width so that the tires follow concentric circles. Mathematically, this means that the input has to be the steering heading angular velocity $\dot{\psi}$, and there is no generalized format.



The differential drive model state is [x y θ ψ].

Variables

- x: Global vehicle x-position in meters
- y: Global vehicle y-position in meters
- θ: Global vehicle heading in radians
- ψ: Vehicle steering angle in radians
- l: Wheel base in meters

- v: Vehicle speed in meters/s

Kinematic Equations

For the Ackermann kinematics model, the ODE is:

$$\begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \\ \dot{\psi} \end{bmatrix} = \begin{bmatrix} \cos(\theta) & 0 \\ \sin(\theta) & 0 \\ \tan(\psi)/l & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} v \\ \dot{\psi} \end{bmatrix}$$

References

[1] Lynch, Kevin M., and Frank C. Park. *Modern Robotics: Mechanics, Planning, and Control*. Cambridge University Press, 2017.

For an example the simulates the different mobile robot using these models, see “Simulate Different Kinematic Models for Mobile Robots” on page 1-67.

Occupancy Grids

Overview

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

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

For 3-D occupancy maps, see `occupancyMap3D`.

For 2-D occupancy grids, there are two representations:

- Binary occupancy grid (see `binaryOccupancyMap`)
- Probability occupancy grid (see `occupancyMap`)

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

A probability occupancy grid uses probability values to create a more detailed map representation. This representation is the preferred method for using occupancy grids. This grid is commonly referred to as simply an occupancy grid. Each cell in the occupancy grid has a value representing the probability of the occupancy of that cell. Values close to 1 represent a high certainty that the cell contains an obstacle. Values close to 0 represent certainty that the cell is not occupied and obstacle free. The probabilistic values can give better fidelity of objects and improve performance of certain algorithm applications.

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

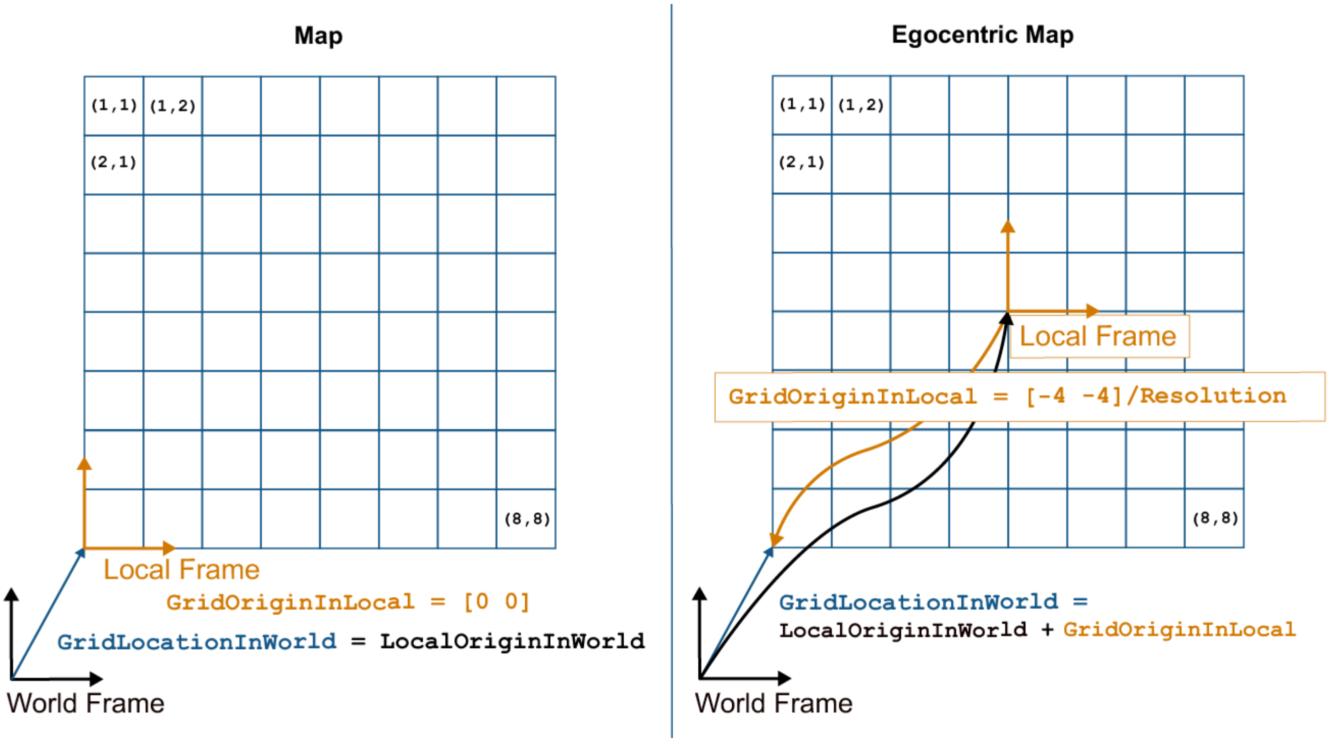
World, Grid, and Local Coordinates

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

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

The local frame refers to the egocentric frame for a vehicle navigating the map. The `GridOriginInLocal` and `LocalOriginInWorld` properties define the origin of the grid in local coordinates and the relative location of the local frame in the world coordinates. You can adjust this local frame using the `move` function. For an example using the local frame as an egocentric map to emulate a vehicle moving around and sending local obstacles, see "Create Egocentric Occupancy Maps Using Range Sensors" (Navigation Toolbox).

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


Map
(1,1)
(1,2)
(2,1)
(8,8)
Local Frame
GridOriginInLocal = [0 0]
GridLocationInWorld = LocalOriginInWorld
World Frame
Egocentric Map
(1,1)
(1,2)
(2,1)
(8,8)
Local Frame
GridOriginInLocal = [-4 -4]/Resolution
GridLocationInWorld =
LocalOriginInWorld + GridOriginInLocal
World Frame

Inflation of Coordinates

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

Binary Occupancy Grid

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

Inflate Obstacles in a Binary Occupancy Grid

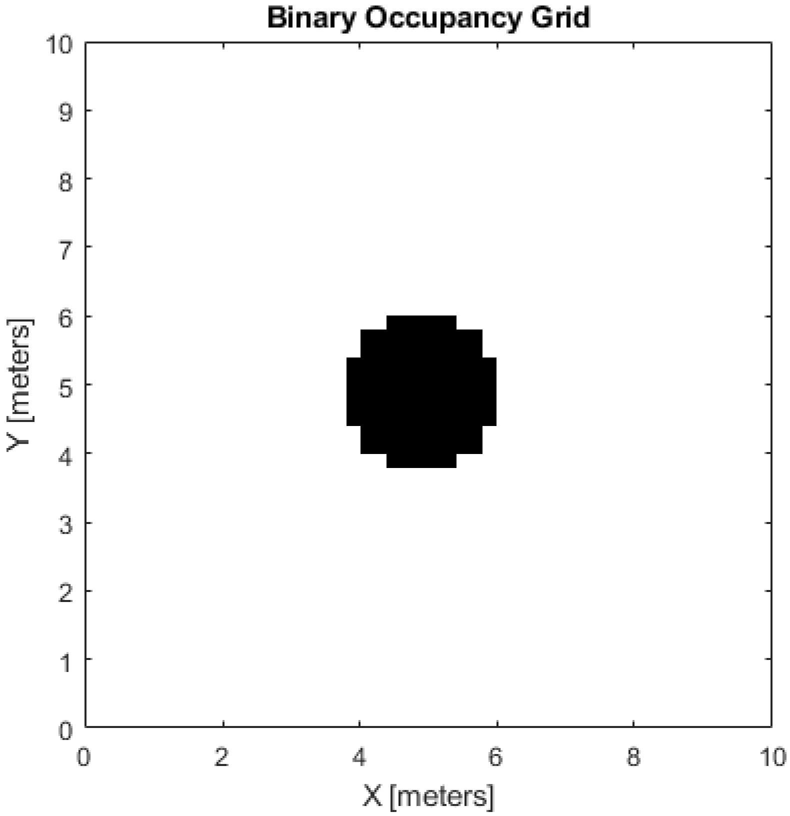
This example shows how to create the map, set the obstacle locations and inflate it by a radius of 1m. Extra plots on the figure help illustrate the inflation and shifting due to conversion to grid locations.

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

```
map = binaryOccupancyMap(10,10,5);
setOccupancy(map,[5 5], 1);
```

Inflate occupied spaces on map by 1m.

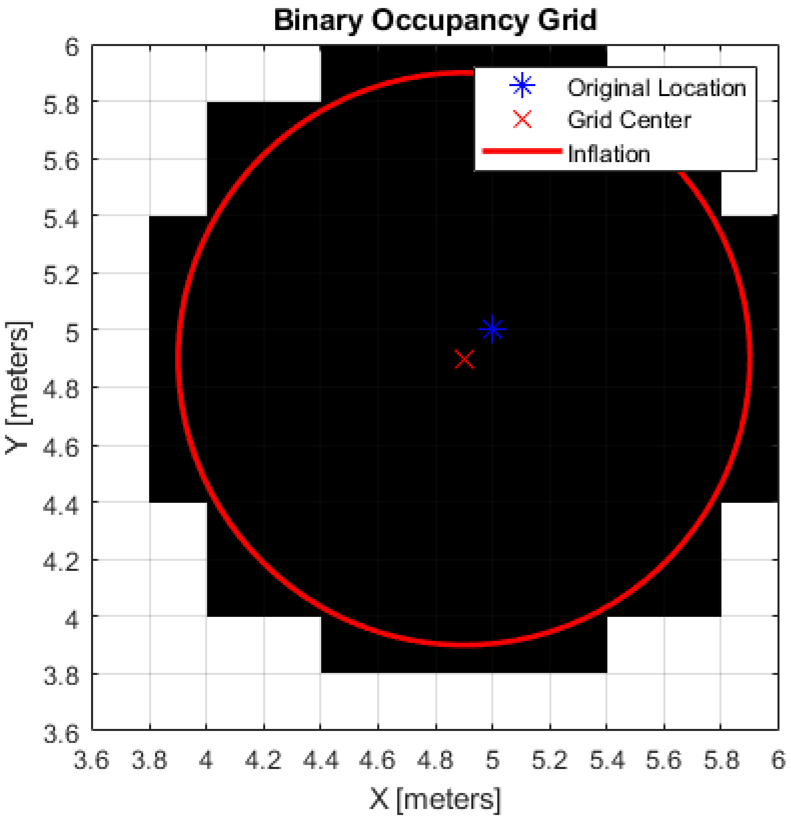
```
inflate(map,1);
show(map)
```



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

```
hold on
theta = linspace(0,2*pi);
```

```
x = 4.9+cos(theta); % x circle coordinates
y = 4.9+sin(theta); % y circle coordinates
plot(5,5,'*b','MarkerSize',10) % Original location
plot(4.9,4.9,'xr','MarkerSize',10) % Grid location center
plot(x,y,'-r','LineWidth',2); % Circle of radius 1m.
axis([3.6 6 3.6 6])
ax = gca;
ax.XTick = [3.6:0.2:6];
ax.YTick = [3.6:0.2:6];
grid on
legend('Original Location','Grid Center','Inflation')
```



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

See Also

`binaryOccupancyMap` | `occupancyMap` | `occupancyMap3D`

Related Examples

- “Create Egocentric Occupancy Maps Using Range Sensors” (Navigation Toolbox)
- “Build Occupancy Map from Lidar Scans and Poses” (Navigation Toolbox)

Probabilistic Roadmaps (PRM)

In this section...

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

When creating or updating the mobileRobotPRM class, the node locations are randomly generated, which can affect your final path between multiple iterations. This selection of nodes occurs when you specify Map initially, change the parameters, or update is called. To get consistent results with the same node placement, use rng to save the state of the random number generation. See “Tune the Connection Distance” on page 2-47 for an example using rng.

Tune the Number of Nodes

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

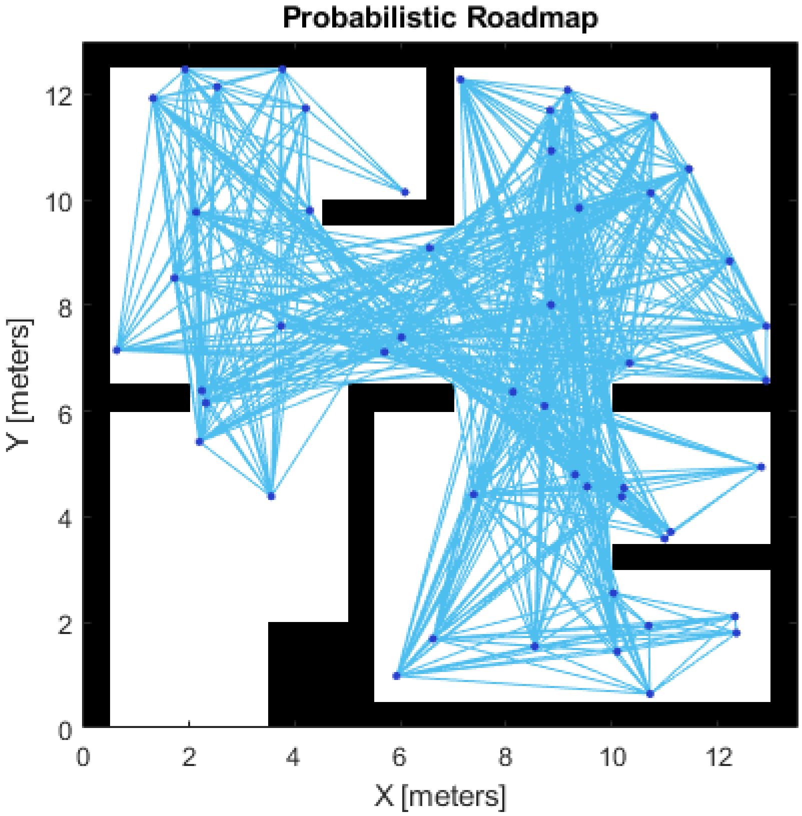
Increasing the number of nodes can increase the efficiency of the path by giving more feasible paths. However, the increased complexity increases computation time. To get good coverage of the map, you might need a large number of nodes. Due to the random placement of nodes, some areas of the map may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

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

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

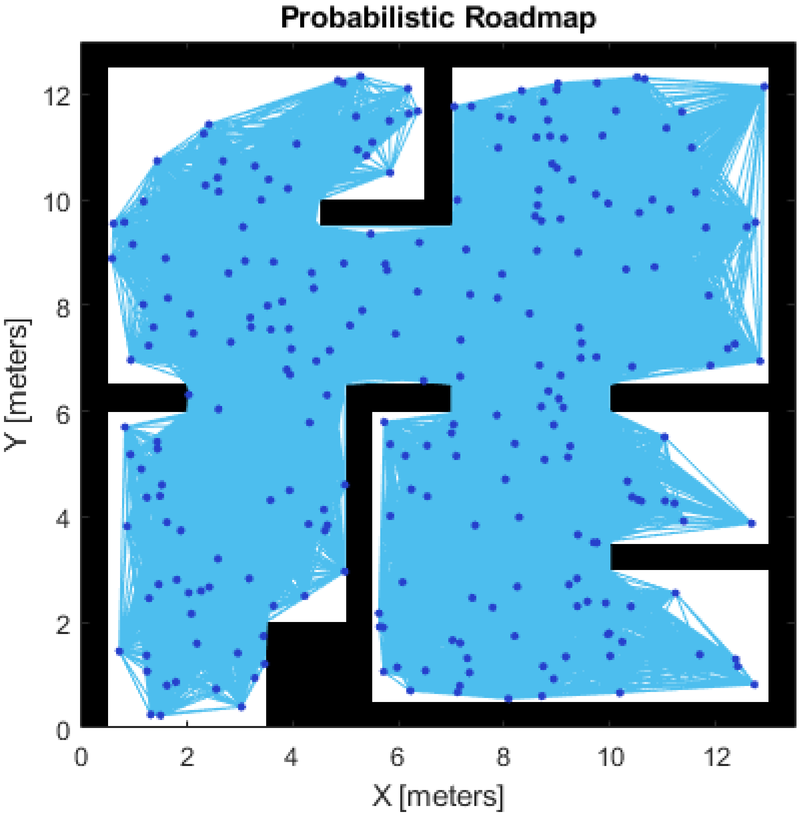
Create a simple roadmap with 50 nodes.

```
prmSimple = mobileRobotPRM(map,50);
show(prmSimple)
```



Create a dense roadmap with 250 nodes.

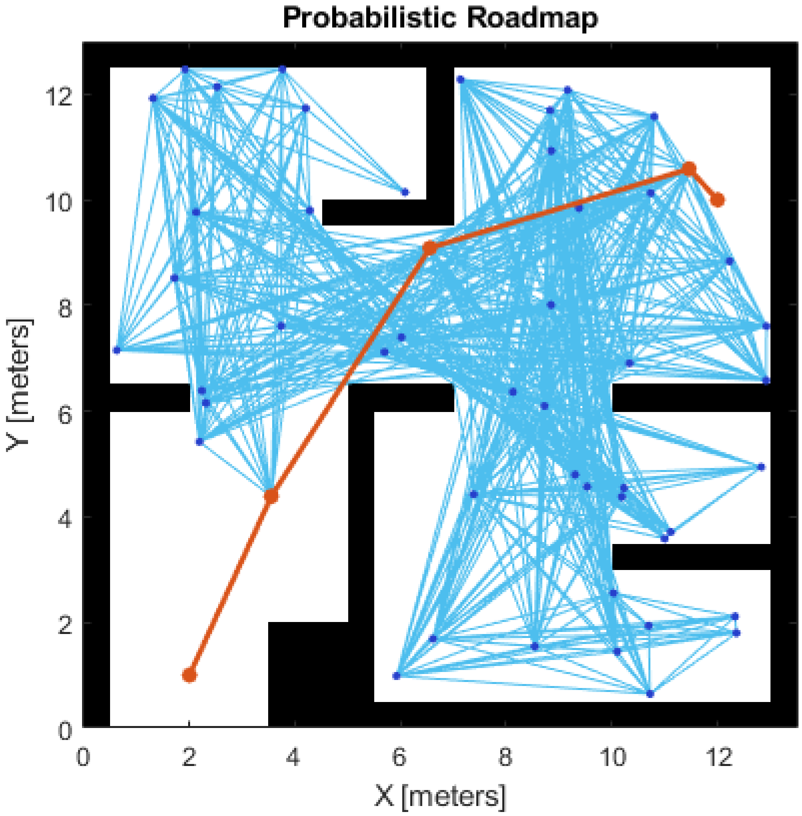
```
prmComplex = mobileRobotPRM(map,250);
show(prmComplex)
```



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

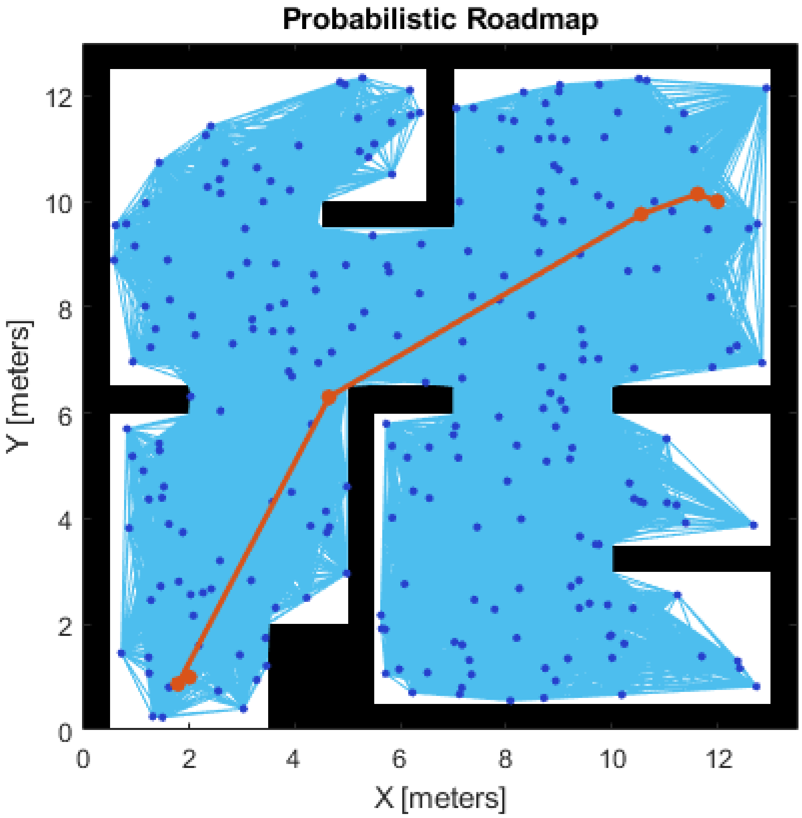
Calculate a simple path.

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



Calculate a complex path.

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



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

Tune the Connection Distance

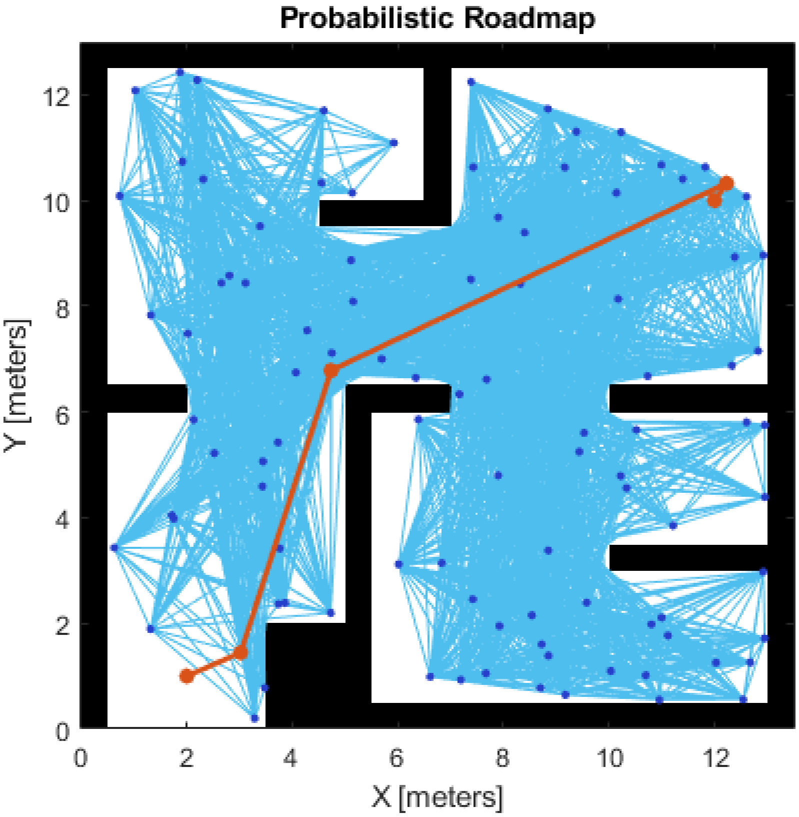
Use the `ConnectionDistance` property on the `PRM` object to tune the algorithm. `ConnectionDistance` is an upper threshold for points that are connected in the roadmap. Each node is connected to all nodes within this connection distance that do not have obstacles between them. By lowering the connection distance, you can limit the number of connections to reduce the computation time and simplify the map. However, a lowered distance limits the number of available paths from which to find a complete obstacle-free path. When working with simple maps, you can use a higher connection distance with a small number of nodes to increase efficiency. For complex maps with lots of obstacles, a higher number of nodes with a lowered connection distance increases the chance of finding a solution.

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

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

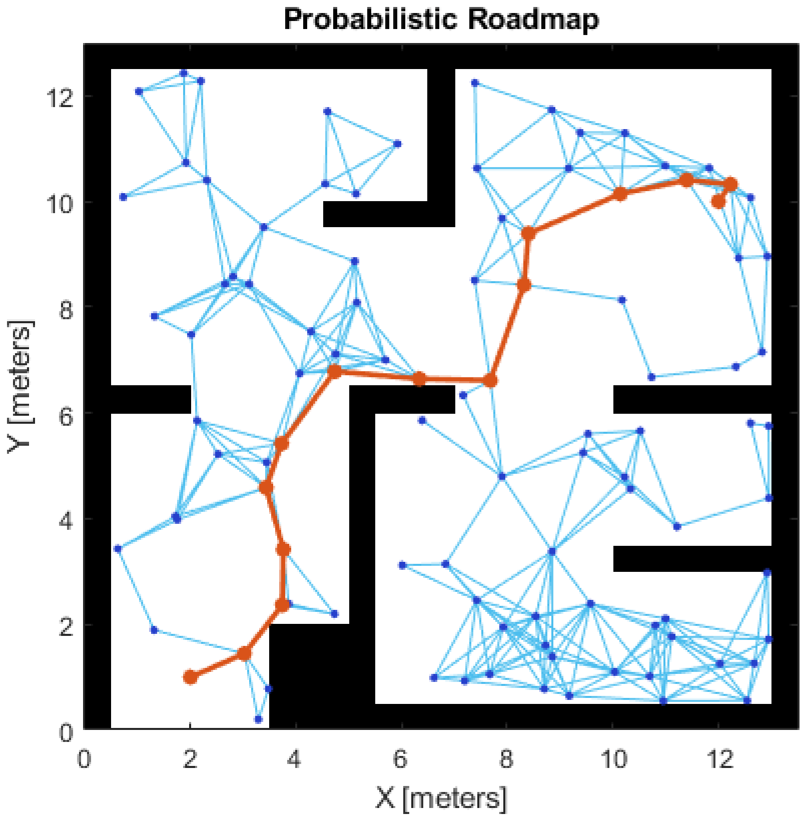
Create a roadmap with 100 nodes and calculate the path. The default `ConnectionDistance` is set to inf. Save the random number generation settings using the rng function. The saved settings enable you to reproduce the same points and see the effect of changing `ConnectionDistance`.

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



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

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



Create or Update PRM

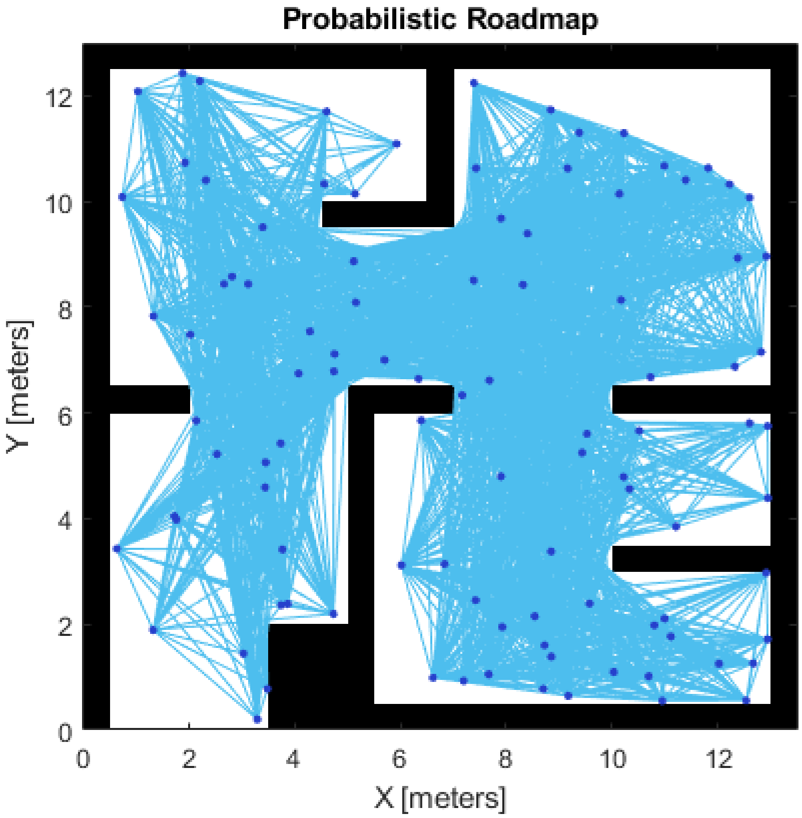
When using the mobileRobotPRM object and modifying properties, with each new function call, the object triggers the roadmap points and connections to be recalculated. Because recalculating the map can be computationally intensive, you can reuse the same roadmap by calling findpath with different starting and ending locations.

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

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

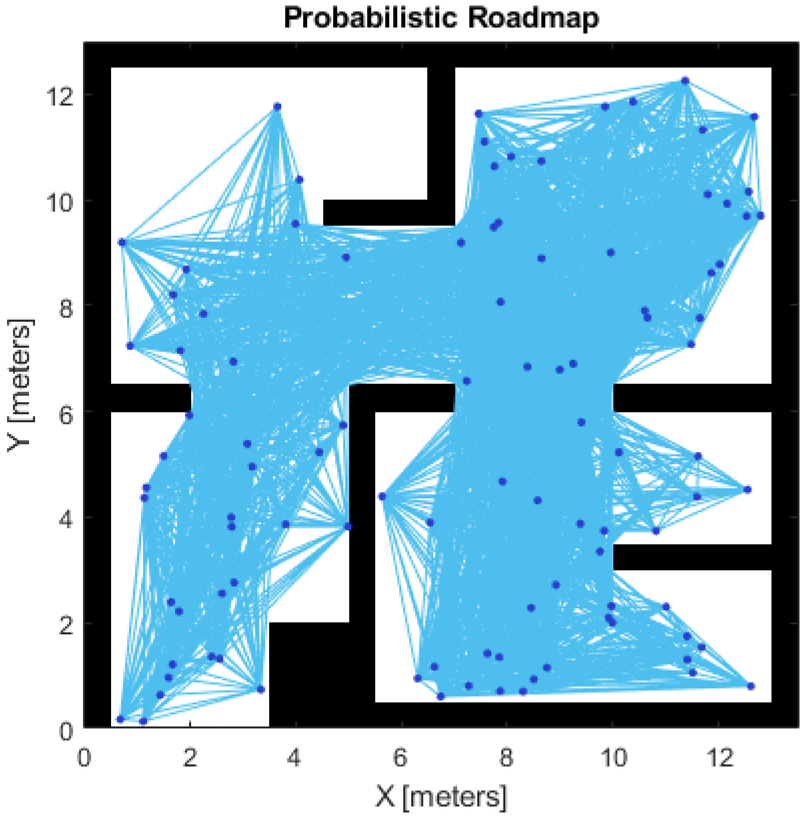
Create a roadmap. Your nodes and connections might look different due to the random placement of nodes.

```
prm = mobileRobotPRM(map,100);
show(prm)
```



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

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



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

References

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

See Also

`mobileRobotPRM` | `findpath`

Pure Pursuit Controller

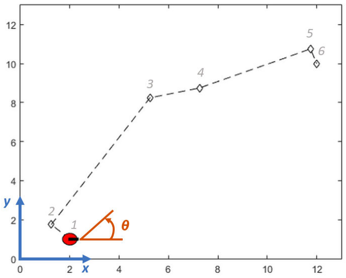
In this section...

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

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

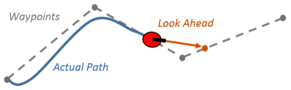
Reference Coordinate System

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

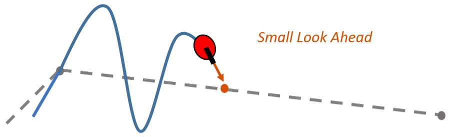


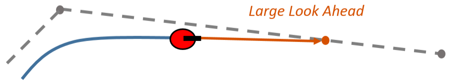
Look Ahead Distance

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



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





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

Limitations

There are a few limitations to note about this pure pursuit algorithm:

- As shown above, the controller cannot exactly follow direct paths between waypoints. Parameters must be tuned to optimize the performance and to converge to the path over time.
- This pure pursuit algorithm does not stabilize the robot at a point. In your application, a distance threshold for a goal location should be applied to stop the robot near the desired goal.

References

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

See Also

`stateEstimatorPF` | `controllerVFH`

Particle Filter Parameters

In this section...

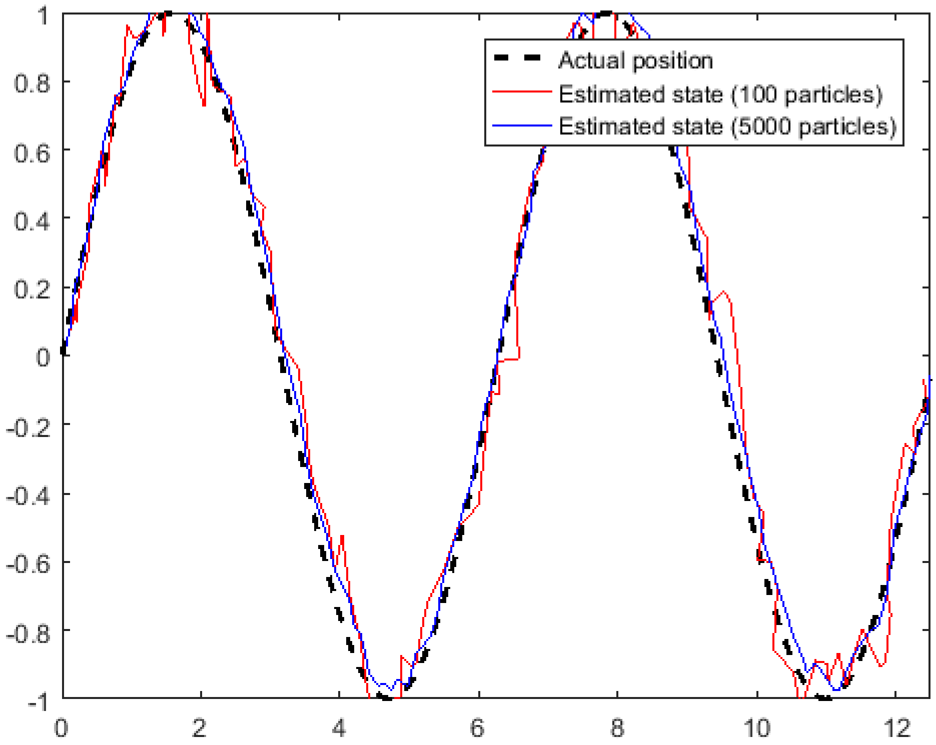
To use the `stateEstimatorPF` particle filter, you must specify parameters such as the number of particles, the initial particle location, and the state estimation method. Also, if you have a specific motion and sensor model, you specify these parameters in the state transition function and measurement likelihood function, respectively. The details of these parameters are detailed on this page. For more information on the particle filter workflow, see “Particle Filter Workflow” on page 2-59.

Number of Particles

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

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

These results, which are based on the `stateEstimatorPF` example, show the difference in tracking accuracy when using 100 particles and 5000 particles.



Initial Particle Location

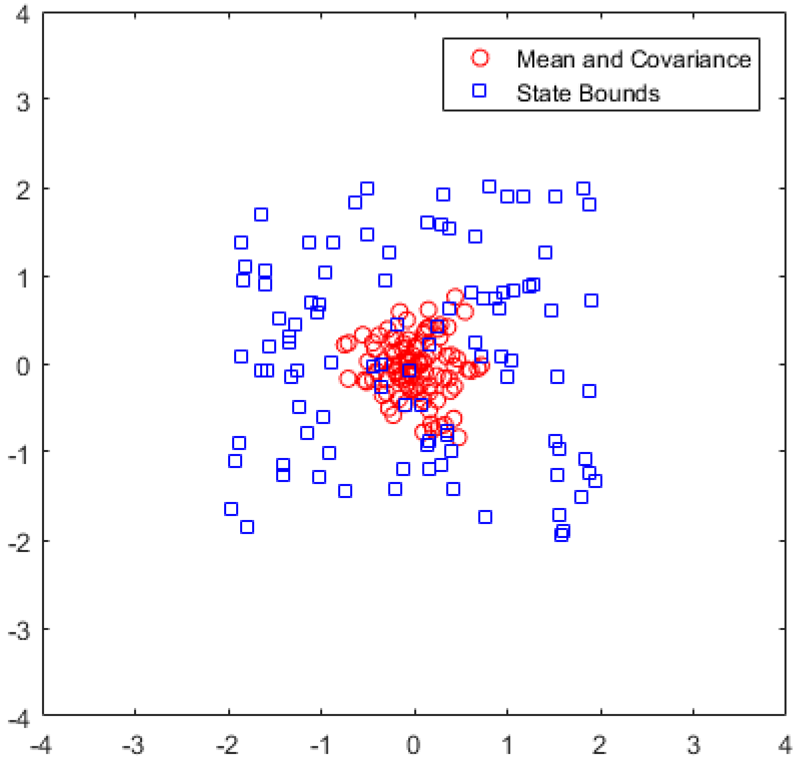
When you initialize your particle filter, you can specify the initial location of the particles using:

- Mean and covariance
- State bounds

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

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

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



State Transition Function

The state transition function, `StateTransitionFcn`, of a particle filter helps to evolve the particles to the next state. It is used during the prediction step of the "Particle Filter Workflow" on page 2-59. In the `stateEstimatorPF` object, the state transition function is specified as a callback function that takes the previous particles, and any other necessary parameters, and outputs the predicted location. The function header syntax is:

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

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

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

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

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

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

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

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

The output particles, `predictParticles`, are then either used by the “Measurement Likelihood Function” on page 2-57 to correct the particles, or used in the next prediction step if correction is not required.

Measurement Likelihood Function

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

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

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

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

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

These parameters match the measurement likelihood function you defined:

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

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

Resampling Policy

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

When you call `correct`, the particles used for state estimation can be resampled depending on the `ResamplingPolicy` property specified in the `stateEstimatorPF` object. This property is specified as a `resamplingPolicyPFresamplingPolicyPF` object. The `TriggerMethod` property on that object tells the particle filter which method to use for resampling.

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

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

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

State Estimation Method

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

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

See Also

`stateEstimatorPF` | `resamplingPolicyPF`

Related Examples

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

More About

- “Particle Filter Workflow” on page 2-59

Particle Filter Workflow

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

The particle filter algorithm computes the state estimate recursively and involves two steps:

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

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

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

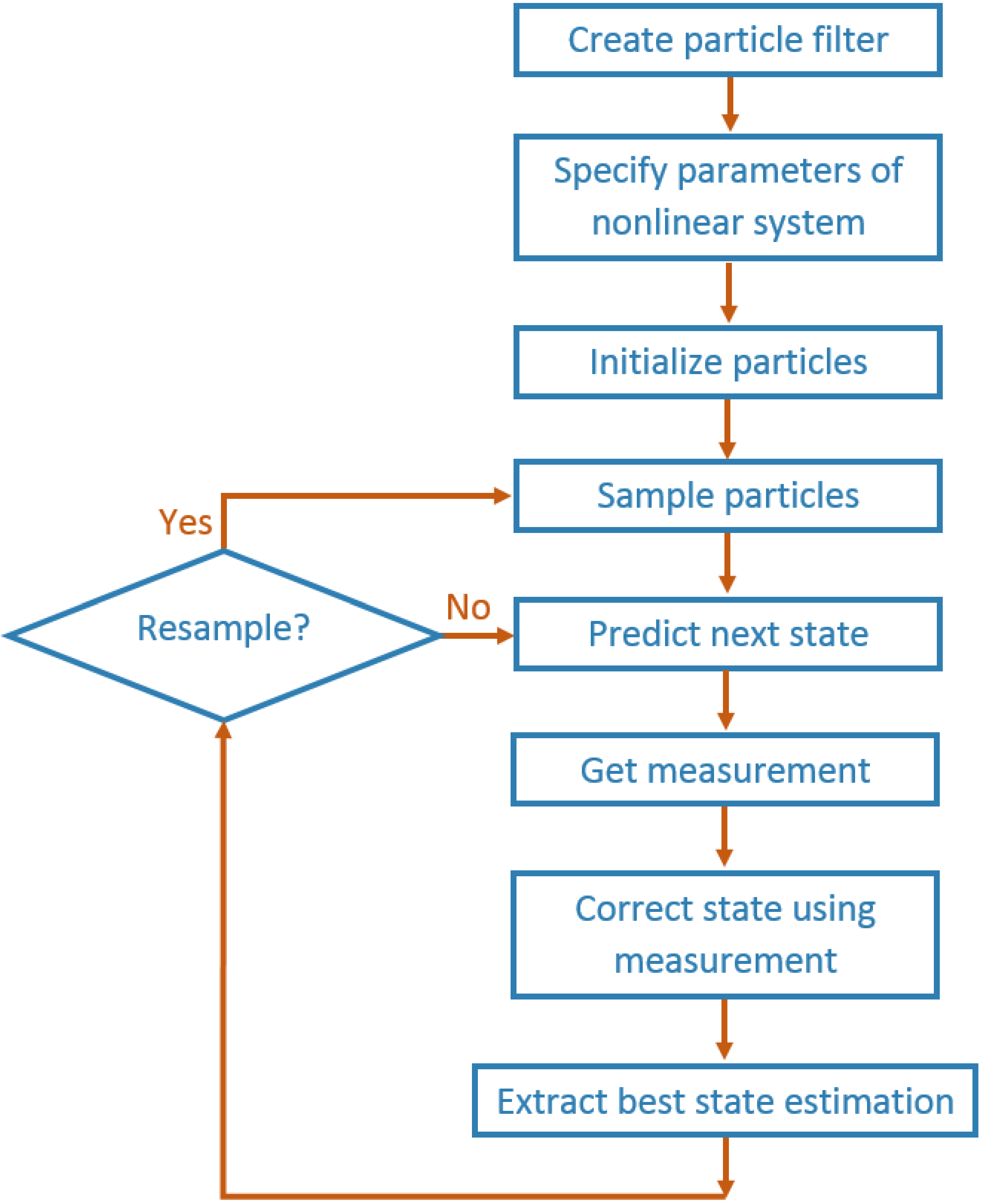
You can apply the particle filter to arbitrary nonlinear system models. Process and measurement noise can follow arbitrary non-Gaussian distributions.

To use the particle filter properly, you must specify parameters such as the number of particles, the initial particle location, and the state estimation method. Also, if you have a specific motion and sensor model, you specify these parameters in the state transition function and measurement likelihood function, respectively. For more information, see “Particle Filter Parameters” on page 2-54.

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

Estimation Workflow

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


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

Create Particle Filter

Create a `stateEstimatorPF` object.

Set Parameters of Nonlinear System

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

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

Default values for these parameters are given for basic operation.

The `StateTransitionFcn` and `MeasurementLikelihoodFcn` functions define the system behavior and measurement integration. They are vital for the particle filter to track accurately. For more information, see "Particle Filter Parameters" on page 2-54.

Initialize Particles

Use the `initialize` function to set the number of particles and the initial state.

Sample Particles from a Distribution

You can sample the initial particle locations in two ways:

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

Predict

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

Get Measurement

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

Correct

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

Extract Best State Estimation

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

Resample Particles

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

Continuously Predict and Correct

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

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

See Also

`stateEstimatorPF` | `initialize` | `getStateEstimate` | `predict` | `correct`

Related Examples

- “Track a Car-Like Robot Using Particle Filter” on page 1-61
- “Estimate Robot Position in a Loop Using Particle Filter”

More About

- “Particle Filter Parameters” on page 2-54

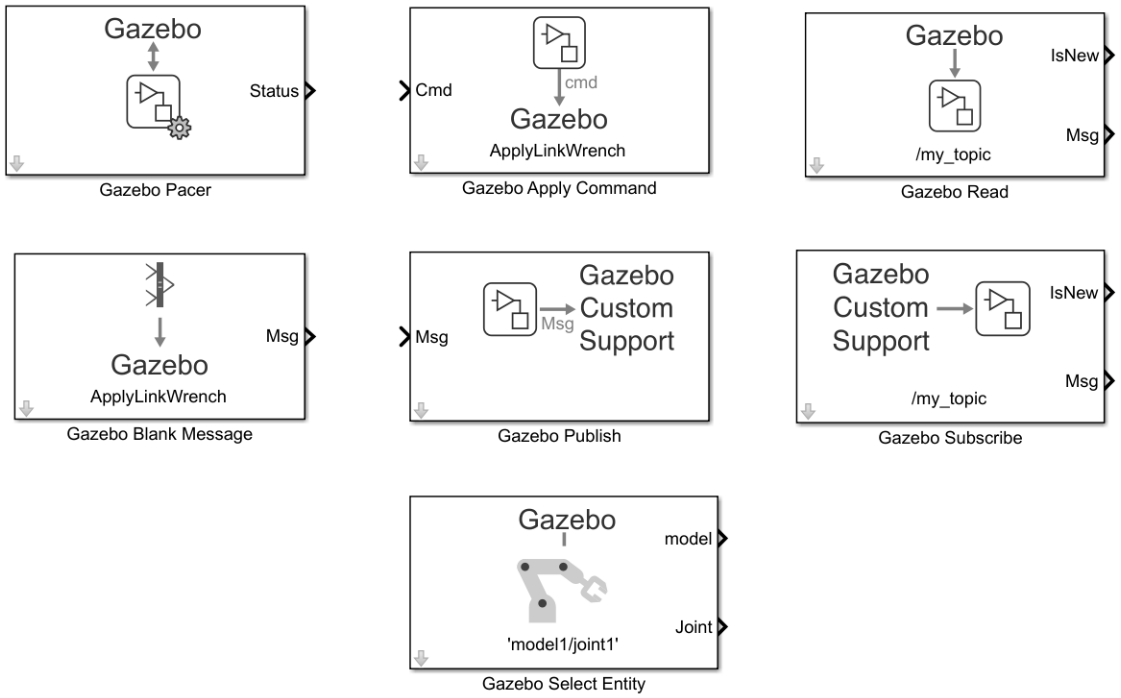
Gazebo Simulation for Robotics System Toolbox

Robotics System Toolbox provides a co-simulation framework that enables you to use robotics algorithms in MATLAB and Simulink and visualize their performance in a virtual simulation environment. This environment uses the Gazebo Simulator. For more details on the simulation environment, see “How Gazebo Simulation for Robotics System Toolbox Works” on page 2-68.

This simulation tool is commonly used to supplement real data when developing, testing, and verifying the performance of robotics algorithms.

Gazebo Co-Simulation Blocks

The **Robotics System Toolbox > Gazebo Co-Simulation** block library contains Simulink blocks related to the simulation environment. To view the library, at the MATLAB command prompt, enter `robotgazebolib`.



Block	Description
Gazebo Apply Command	Send command to Gazebo simulator
Gazebo Blank Message	Create blank Gazebo command
Gazebo Pacer	Settings for synchronized stepping between Gazebo and Simulink
Gazebo Read	Receive messages from Gazebo server
Gazebo Publish	Send custom messages to Gazebo server
Gazebo Subscribe	Receive custom messages from Gazebo server

Block	Description
Gazebo Select Entity	Select a Gazebo entity

Gazebo Co-Simulation Functions

These are the MATLAB functions related to the simulation environment.

Function	Description
`gzinit`	Initialize connection settings for Gazebo Co-Simulation MATLAB interface
`gzjoint`	Assign and retrieve Gazebo model joint information
`gzlink`	Assign and retrieve Gazebo model link information
`gzmodel`	Assign and retrieve Gazebo model information
`gzworld`	Interact with Gazebo world
`gazebogenmsg`	Generate dependencies for Gazebo custom message support
`packageGazeboPlugin`	Create Gazebo plugin package for Simulink

See Also

Related Examples

- “How Gazebo Simulation for Robotics System Toolbox Works” on page 2-68
- “Gazebo Simulation Environment Requirements and Limitations” on page 2-65

Gazebo Simulation Environment Requirements and Limitations

Robotics System Toolbox provides an interface for a simulation environment visualized using the Gazebo Simulator. Gazebo enables you to test and experiment using robots in realistically simulated physical scenarios with high quality graphics.

Gazebo runs on Linux® machines or Linux virtual machines, and uses a plugin package to communicate with MATLAB and Simulink. When simulating in Gazebo, the requirements and limitations in mind.

Run Gazebo Simulator on Linux Virtual Machine

Download and install the virtual machine (VM) from Virtual Machine with ROS and Gazebo. In the VM, the required Gazebo plugin is located in the `/home/user/src/GazeboPlugin` folder. The VM contains these software and has these hardware requirements.

Software Included in VM

- **Operating System** — Ubuntu® Ubuntu 20.04.3 LTS (Focal Fossa)
- **Software packages** — CMake 3.16.3, Gazebo 11, and the Gazebo plugin

Minimum Hardware Requirements

- **Processor (CPU)** — Quad core Intel® i5, or equivalent
- **Memory (RAM)** — 4 GB or more
- **Graphics card (GPU)** — Dedicated GPU with 1 GB or more graphics memory
- **Disk space** — At least 20 GB free disk space

Install and Run Gazebo Simulator on Linux Machine

You can also install and run the Gazebo Simulator on a Linux machine.

Software Requirements

- **Operating System** — Ubuntu 16.04 LTS (Xenial Xerus) or Ubuntu 18.04 LTS (Bionic Beaver) or Ubuntu 20.04 LTS (Focal Fossa)
- **Software packages** — CMake 2.8 or later, Gazebo 9 or Gazebo 10 or Gazebo 11, and the Gazebo plugin

Minimum Hardware Requirements

- **Processor (CPU)** — Quad core Intel i5, or equivalent
- **Memory (RAM)** — 4 GB or more
- **Graphics card (GPU)** — Dedicated GPU with 1 GB or more graphics memory
- **Disk space** — At least 500 MB free disk space

Gazebo Simulator Installation

Install the CMake and Gazebo packages on Ubuntu by running these commands at the Linux terminal. For more information on installing Gazebo on Ubuntu machine, see Install Gazebo using Ubuntu packages.

```
sudo sh -c 'echo "deb http://packages.osrfoundation.org/gazebo/ubuntu-stable `lsb_release -cs` ma
wget https://packages.osrfoundation.org/gazebo.key -O - | sudo apt-key add -
sudo apt-get update
sudo apt-get install cmake gazebo11 libgazebo11-dev
```

Note Gazebo co-simulation framework supports Gazebo 9, Gazebo 10, and Gazebo 11.

Gazebo Plugin Installation

1. Create a Gazebo plugin package on the host computer in MATLAB by using the `packageGazeboPlugin` function. This function creates a folder called `GazeboPlugin` in your current working directory and compresses it into a `GazeboPlugin.zip` file.

   ```
   packageGazeboPlugin
   ```
2. Copy `GazeboPlugin.zip` to the home directory of your Linux machine.
3. Create a directory, `src`, and unzip the plugin package to that directory.

   ```
   mkdir src
   unzip GazeboPlugin.zip -d ~/src/
   ```
4. Change the directory to the uncompressed `GazeboPlugin` folder.

   ```
   cd ~/src/GazeboPlugin/
   ```
5. Run these commands in the Linux terminal to compile and install the plugin.

   ```
   mkdir build
   cd build
   cmake ..
   make
   ```
6. Optionally, you can remove the generated plugin from the host computer using MATLAB.

   ```
   if exist("GazeboPlugin","dir")
       rmdir("GazeboPlugin","s");
   end

   if exist("GazeboPlugin.zip","file")
       delete("GazeboPlugin.zip");
   end
   ```

Limitations

MATLAB

- Code generation is not supported.
- Communication between MATLAB and the Gazebo Simulator is asynchronous.

Simulink

- Code generation is not supported.
- Rapid accelerator mode is not supported.

See Also

Related Examples

- “Gazebo Simulation for Robotics System Toolbox” on page 2-63
- “How Gazebo Simulation for Robotics System Toolbox Works” on page 2-68

External Websites

- Gazebo

How Gazebo Simulation for Robotics System Toolbox Works

Robotics System Toolbox provides a co-simulation framework that enables you to use robotics algorithms in MATLAB and Simulink and visualize their performance in a virtual simulation environment. This environment uses the Gazebo Simulator.

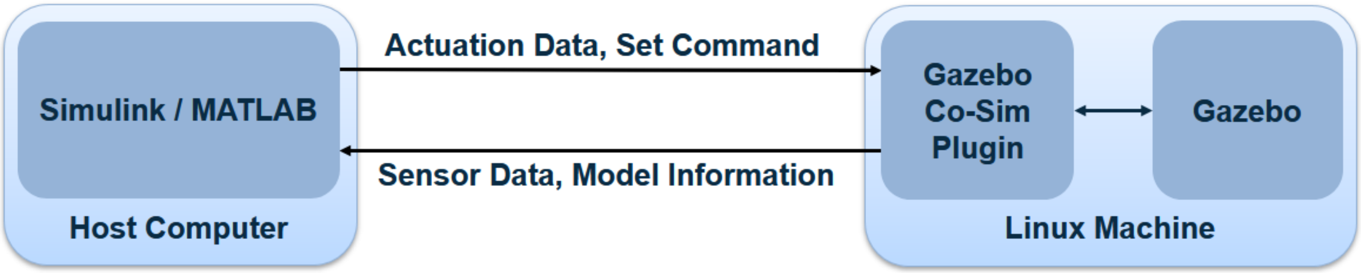
Understanding how this simulation environment works can help you troubleshoot issues and customize your models.

Communication with 3-D Simulation Environment

When you use Robotics System Toolbox to run your algorithms, MATLAB or Simulink co-simulates the algorithms in the simulation environment.

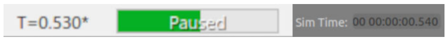
On the host computer, Simulink or MATLAB sends actuation data and set commands to the Gazebo Co-Simulation Plugin on the target Linux machine. The plugin passes this information to the Gazebo Simulator, which returns sensor data and model information through the plugin to Simulink or MATLAB on the host computer.

The diagram summarizes the communication between MATLAB or Simulink and the simulation environment.

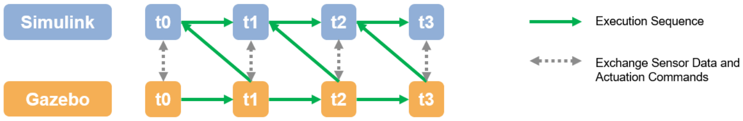


Time Synchronization

During co-simulation, you can pause Simulink and the Gazebo Simulator at any time using Pause. Gazebo pauses one time step ahead of the simulation.



The gap is due to the co-simulation time sequence:



Sensor data and actuation commands are exchanged at the correct time step. The execution steps Gazebo first, then Simulink. When paused, simulation execution is still at the time step executed by Gazebo, but Simulink remains on the previous step time until you resume the model.

See Also

Related Examples

- “Gazebo Simulation for Robotics System Toolbox” on page 2-63
- “Gazebo Simulation Environment Requirements and Limitations” on page 2-65

Standard Units for Robotics System Toolbox

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

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

See Also

More About

- “Coordinate Transformations in Robotics” on page 2-71

Coordinate Transformations in Robotics

In this section...

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

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

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

Axis-Angle

Abbreviation: `axang`

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

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

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

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

Euler Angles

Abbreviation: `eul`

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

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

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

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

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

Homogeneous Transformation Matrix

Abbreviation: `tform`

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

Numeric Representation: 4-by-4 matrix

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

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

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

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

Quaternion

Abbreviation: `quat`

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

Numeric Representation: 1-by-4 vector

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

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

Rotation Matrix

Abbreviation: `rotm`

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

Numeric Representation: 3-by-3 matrix

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

```
rotm =

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

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

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

Translation Vector

Abbreviation: `trvec`

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

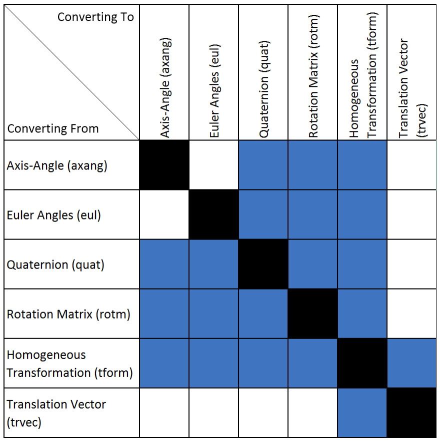
Numeric Representation: 1-by-3 vector

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

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

Conversion Functions and Transformations

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



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

The names of all the conversion functions follow a standard format. They follow the form `alpha2beta` where `alpha` is the abbreviation for what you are converting from and `beta` is what you are converting to as an abbreviation. For example, converting from Euler angles to quaternion would be `eul2quat`.

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

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

See Also

More About

- “Standard Units for Robotics System Toolbox” on page 2-70

Execute Code at a Fixed-Rate

In this section...

Introduction

By executing code at constant intervals, you can accurately time and schedule tasks. Using a `rateControl` object allows you to control the rate of your code execution. These examples show different applications for the `rateControl` object including its uses with ROS and sending commands for robot control.

Run Loop at Fixed Rate

Create a rate object that runs at 1 Hz.

```
r = rateControl(1);
```

Start a loop using the `rateControl` object inside to control the loop execution. Reset the object prior to the loop execution to reset timer. Print the iteration and time elapsed.

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

```
Iteration: 1 - Time Elapsed: 0.002257
Iteration: 2 - Time Elapsed: 1.011465
Iteration: 3 - Time Elapsed: 2.011186
Iteration: 4 - Time Elapsed: 3.014202
Iteration: 5 - Time Elapsed: 4.001980
Iteration: 6 - Time Elapsed: 5.001219
Iteration: 7 - Time Elapsed: 6.000373
Iteration: 8 - Time Elapsed: 7.000181
Iteration: 9 - Time Elapsed: 8.000670
Iteration: 10 - Time Elapsed: 9.000240
```

Each iteration executes at a 1-second interval.

Overrun Actions for Fixed Rate Execution

The `rateControl` object uses the `OverrunAction` property to decide how to handle code that takes longer than the desired period to operate. The options are `'slip'` (default) or `'drop'`. This example shows how the `OverrunAction` affects code execution.

Setup desired rate and loop time. `slowFrames` is an array of times when the loop should be stalled longer than the desired rate.

```
desiredRate = 1;
loopTime = 20;
slowFrames = [3 7 12 18];
```

Create the Rate object and specify the OverrunAction property. 'slip' indicates that the waitfor function will return immediately if the time for LastPeriod is greater than the DesiredRate property.

```
rate = rateControl(desiredRate);
rate.OverrunAction = 'slip';
```

Reset Rate object and begin loop. This loop will execute at the desired rate until the loop time is reached. When the TotalElapsedTime reaches a slow frame time, it will stall for longer than the desired period.

```
reset(rate);

while rate.TotalElapsedTime < loopTime
    if ~isempty(find(slowFrames == floor(rate.TotalElapsedTime)))
        pause(desiredRate + 0.1)
    end
    waitfor(rate);
end
```

View statistics on the Rate object. Notice the number of periods.

```
stats = statistics(rate)
```

```
stats = struct with fields:
              Periods: [1.0143 0.9965 1.0014 1.1026 1.0077 0.9991 0.9961 ... ]
           NumPeriods: 20
        AveragePeriod: 1.0225
    StandardDeviation: 0.0421
          NumOverruns: 4
```

Change the OverrunAction to 'drop'. 'drop' indicates that the waitfor function will return at the next time step, even if the LastPeriod is greater than the DesiredRate property. This effectively drops the iteration that was missed by the slower code execution.

```
rate.OverrunAction = 'drop';
```

Reset Rate object and begin loop.

```
reset(rate);

while rate.TotalElapsedTime < loopTime
    if ~isempty(find(slowFrames == floor(rate.TotalElapsedTime)))
        pause(1.1)
    end
    waitfor(rate);
end
stats2 = statistics(rate)
```

```
stats2 = struct with fields:
              Periods: [1.0143 0.9872 1.0096 1.9941 1.0006 0.9991 2.0073 ... ]
           NumPeriods: 16
        AveragePeriod: 1.2501
```

```
    StandardDeviation: 0.4481
          NumOverruns: 4
```

Using the 'drop' over run action resulted in 16 periods when the 'slip' resulted in 20 periods. This difference is because the 'slip' did not wait until the next interval based on the desired rate. Essentially, using 'slip' tries to keep the AveragePeriod property as close to the desired rate. Using 'drop' ensures the code will execute at an even interval relative to DesiredRate with some iterations being skipped.

See Also

rateControl | rosrate | waitfor

Accelerate Robotics Algorithms with Code Generation

In this section...

You can generate code for select Robotics System Toolbox algorithms to speed up their execution. Set up the algorithm that supports code generation as a separate function that you can insert into your workflow. To use code generation, you must have a MATLAB Coder™ license. For a list of code generation support in Robotics System Toolbox, see Functions Supporting Code Generation.

For this example, use a `inverseKinematics` object with a `rigidBodyTree` robot model to solve for robot configurations that achieve a desired end-effector position.

Create Separate Function for Algorithm

Create a separate function, `ikCodegen`, that runs the inverse kinematics algorithm. Create `inverseKinematics` object and build the `rigidBodyTree` model inside the function. Specify `%#codegen` inside the function to identify it as a function for code generation.

```
function qConfig = ikCodegen(endEffectorName,tform,weights,initialGuess)
    %#codegen

    robot = rigidBodyTree('MaxNumBodies',3,'DataFormat','row');
    body1 = rigidBody('body1');
    body1.Joint = rigidBodyJoint('jnt1','revolute');

    body2 = rigidBody('body2');
    jnt2 = rigidBodyJoint('jnt2','revolute');
    setFixedTransform(jnt2,trvec2tform([1 0 0]))
    body2.Joint = jnt2;

    body3 = rigidBody('tool');
    jnt3 = rigidBodyJoint('jnt3','revolute');
    setFixedTransform(jnt3,trvec2tform([1 0 0]))
    body3.Joint = jnt3;

    addBody(robot,body1,'base')
    addBody(robot,body2,'body1')
    addBody(robot,body3,'body2')


    ik = inverseKinematics('RigidBodyTree',robot);

    [qConfig,~] = ik(endEffectorName,tform,weights,initialGuess);
end
```

Save the function in your current folder.

Perform Code Generation for Algorithm

You can use either the codegen function or the **MATLAB Coder** app to generate code. In this example, generate a MEX file by calling codegen on the MATLAB command line. Specify sample input arguments for each input to the function using the -args input argument

Specify sample values for the input arguments.

```
endEffectorName = 'tool';
tform = trvec2tform([0.7 -0.7 0]);
weights = [0.25 0.25 0.25 1 1 1];
initialGuess = [0 0 0];
```

Call the codegen function and specify the input arguments in a cell array. This function creates a separate ikCodegen_mex function to use. You can also produce C code by using the options input argument.

```
codegen ikCodegen -args {endEffectorName,tform,weights,initialGuess}
```

If your input can come from variable-size lengths, specify the canonical type of the inputs by using coder.typeof with the codegen function.

Check Performance of Generated Code

Compare the timing of the generated MEX function to the timing of your original function by using timeit.

```
time = timeit(@() ikCodegen(endEffectorName,tform,weights,initialGuess))
mexTime = timeit(@() ikCodegen_mex(endEffectorName,tform,weights,initialGuess))
```

```
time =

    0.0425


mexTime =

    0.0011
```

The MEX function runs over 30 times faster in this example. Results might vary in your system.

Replace Algorithm Function with MEX Function

Open the main function for running your robotics workflow. Replace the ik object call with the MEX function that you created using code generation. For this example, use the simple 2-D path tracing example.

Open the “2-D Path Tracing with Inverse Kinematics” on page 2-14 example.

```
openExample('robotics/TwoDInverseKinematicsExampleExample')
```

Modify the example code to use the new ikCodegen_mex function. The code that follows is a copy of the example with modifications to use of the new MEX function. Defining the robot model is done inside the function, so skip the **Construct the Robot** section.

Define The Trajectory

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

Inverse Kinematics Solution

Pre-allocate configuration solutions as a matrix, `qs`. Specify the weights for the end-effector transformation and the end-effector name.

```
q0 = [0 0 0];
ndof = length(q0);
qs = zeros(count, ndof);
weights = [0, 0, 0, 1, 1, 0];
endEffector = 'tool';
```

Loop through the trajectory of points to trace the circle. Replace the `ik` object call with the `ikCodegen_mex` function. Calculate the solution for each point to generate the joint configuration that achieves the end-effector position. Store the configurations to use later.

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

Animate Solution

Now that all the solutions have been generated. Animate the results. You must recreate the robot because it was originally defined inside the function. Iterate through all the solutions.

```
robot = rigidBodyTree('MaxNumBodies',15,'DataFormat','row');
body1 = rigidBody('body1');
body1.Joint = rigidBodyJoint('jnt1','revolute');

body2 = rigidBody('body2');
jnt2 = rigidBodyJoint('jnt2','revolute');
setFixedTransform(jnt2,trvec2tform([0.3 0 0]))
body2.Joint = jnt2;

body3 = rigidBody('tool');
jnt3 = rigidBodyJoint('jnt3','revolute');
setFixedTransform(jnt3,trvec2tform([0.3 0 0]))
body3.Joint = jnt3;

addBody(robot,body1,'base')
addBody(robot,body2,'body1')
addBody(robot,body3,'body2')
```

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

% Iterate through the solutions
framesPerSecond = 15;
r = rateControl(framesPerSecond);
for i = 1:count
    show(robot,qs(i,:),'PreservePlot',false);
    drawnow
    waitfor(r);
end
```

This example showed you how can you generate code for specific algorithms or functions to improve their speed and simply replace them with the generated MEX function in your workflow.

See Also

`codegen` | `timeit` | `inverseKinematics`

Related Examples

- “2-D Path Tracing with Inverse Kinematics” on page 2-14
- Functions Supporting Code Generation
- “Generate C Code at the Command Line” (MATLAB Coder)
- “Generate C Code by Using the MATLAB Coder App” (MATLAB Coder)

Install Robotics System Toolbox Add-ons

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

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

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

   - **Robotics System Toolbox UAV Library**
3. Click **Install**, and select either:

   - **Install**
   - **Download Only...** — Downloads an install file to use offline.
4. Continue to follow the setup instructions on the **Add-Ons Explorer** to install your add-ons.

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

See Also

Related Examples

- “Add-Ons”

Code Generation from MATLAB Code

Several Robotics System Toolbox functions are enabled to generate C/C++ code. Code generation from MATLAB code requires the MATLAB Coder product. To generate code from robotics functions, follow these steps:

- Write your function or application that uses Robotics System Toolbox functions that are enabled for code generation. For code generation, some of these functions have requirements that you must follow. See “Code Generation Support” on page 2-84.
- Add the `%#codegen` directive to your MATLAB code.
- Follow the workflow for code generation from MATLAB code using either the MATLAB Coder app or the command-line interface.

Using the app, the basic workflow is:

1. Set up a project. Specify your top-level functions and define input types.

   The app screens your code for code generation readiness. It reports issues such as a function that is not supported for code generation.
2. Check for run-time issues.

   The app generates and runs a MEX version of your function. This step detects issues that can be hard to detect in the generated C/C++ code.
3. Configure the code generation settings for your application.
4. Generate C/C++ code.
5. Verify the generated C/C++ code. If you have an Embedded Coder® license, you can use software-in-the-loop execution (SIL) or processor-in-the-loop (PIL) execution.

For a tutorial, see “Generate C Code by Using the MATLAB Coder App” (MATLAB Coder).

Using the command-line interface, the basic workflow is:

- To detect issues and verify the behavior of the generated code, generate a MEX version of your function.
- Use `coder.config` to create a code configuration object for a library or executable.
- Modify the code configuration object properties as required for your application.
- Generate code using the `codegen` command.
- Verify the generated code. If you have an Embedded Coder license, you can use software-in-the-loop execution (SIL) or processor-in-the-loop (PIL) execution.

For a tutorial, see “Generate C Code at the Command Line” (MATLAB Coder).

To view a full list of code generation support, see Functions Supporting Code Generation. You can also view the **Extended Capabilities** section on any reference page.

See Also

More About

- Functions Supporting Code Generation

Code Generation Support

To generate code from MATLAB code that contains Robotics System Toolbox functions, classes, or System objects, you must have the MATLAB Coder software.

To view a full list of code generation support, see Functions Supporting Code Generation. You can also view the **Extended Capabilities** section on any reference page.

See Also

More About

- “Code Generation from MATLAB Code” on page 2-83

3

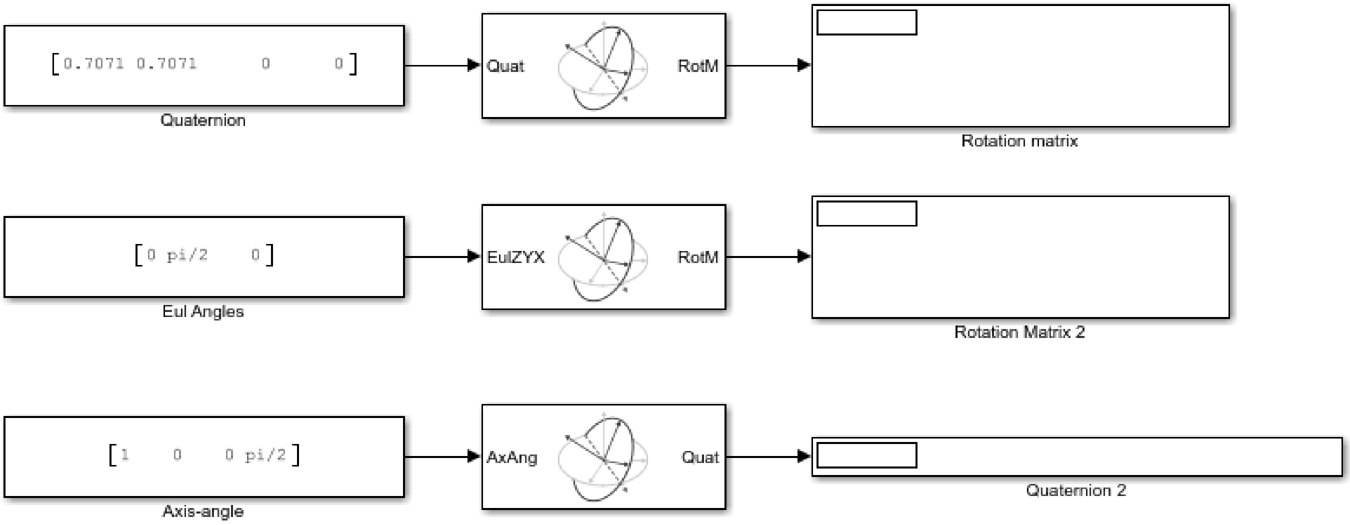
Examples for Simulink Blocks

Convert Coordinate System Transformations

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

```
open_system('coord_trans_block_example_model.slx')
```

```
Warning: Unrecognized function or variable 'registerTICCS'.
Warning: Unrecognized function or variable 'customizationticcs'.
```



Compute Geometric Jacobian for Manipulators in Simulink

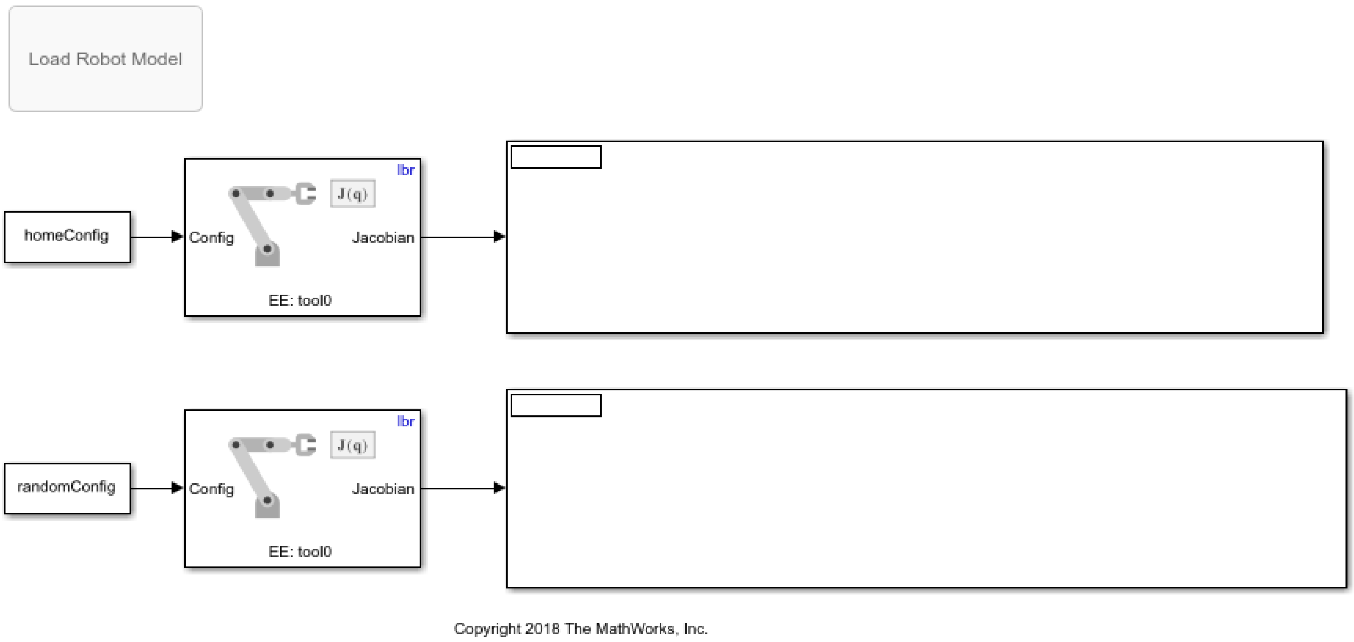
This example shows how to calculate the geometric Jacobian for a robot manipulator by using a `rigidBodyTree` model. The Jacobian maps the joint-space velocity to the end-effector velocity relative to the base coordinate frame. In this example, you define a robot model and robot configurations in MATLAB® and pass them to Simulink® to be used with the manipulator algorithm blocks.

Load a `RigidBodyTree` object that models a KUKA LBR robot. Use the `homeConfiguration` function to get the home configuration or home joint positions of the robot. Use the `randomConfiguration` function to generate a random configuration within the specified joint limits.

```
load('exampleRobots.mat','lbr')
lbr.DataFormat = 'column';
homeConfig = homeConfiguration(lbr);
randomConfig = randomConfiguration(lbr);
```

Open the model. If necessary, use the **Load Robot Model** callback button to reload the robot model and the configuration vectors. The `'tool0'` body is selected as the end-effector in both blocks.

```
open_system('get_jacobian_example.slx')
```



Run the model to display the Jacobian for each configuration.

See Also

Blocks

Forward Dynamics | Inverse Dynamics | Get Jacobian | Gravity Torque | Joint Space Mass Matrix

Classes

`rigidBodyTree`

Functions
`externalForce` | `importrobot` | `homeConfiguration` | `randomConfiguration`

Related Examples

- "Perform Safe Trajectory Tracking Control Using Robotics Manipulator Blocks" on page 1-268

Get Transformations for Manipulator Bodies in Simulink

This example shows how to get the transformation between bodies in a `rigidBodyTree` robot model. In this example, you define a robot model and robot configuration in MATLAB® and pass them to Simulink® to be used with the manipulator algorithm block.

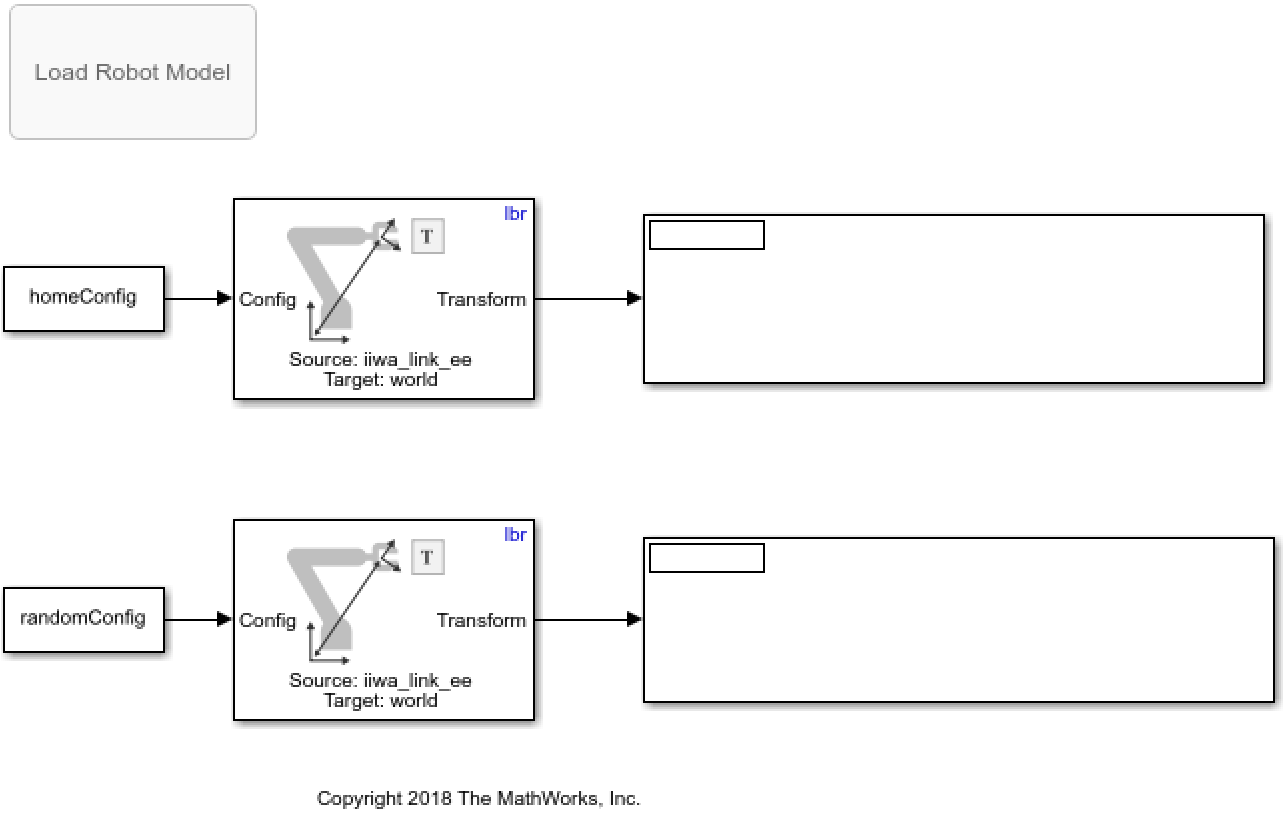
Load the robot model of the KUKA LBR robot as a `RigidBodyTree` object. Use the `homeConfiguration` function to get the home configuration as joint positions of the robot.

```
load('exampleLBR.mat','lbr')
lbr.DataFormat = 'column';

homeConfig = homeConfiguration(lbr);
randomConfig = randomConfiguration(lbr);
```

Open the model. If necessary, use the **Load Robot Model** callback button to reload the robot model and configuration vectors.

The Get Transform block calculates the transformation from the source body to the target body. This transformation converts coordinates from the source body frame to the given target body frame. This example gives you transformations to convert coordinates from the `'iiwa_link_ee'` end effector into the `'world'` base coordinates.

```
open_system('get_transform_example.slx')
```



Run the model to get the transformations.

See Also

Blocks
Get Transform | Forward Dynamics | Inverse Dynamics | Get Jacobian | Gravity Torque | Joint Space Mass Matrix

Classes
`rigidBodyTree`

Functions
`importrobot` | `homeConfiguration` | `randomConfiguration`

Related Examples

- “Perform Safe Trajectory Tracking Control Using Robotics Manipulator Blocks” on page 1-268

Calculate Manipulator Gravity Dynamics in Simulink

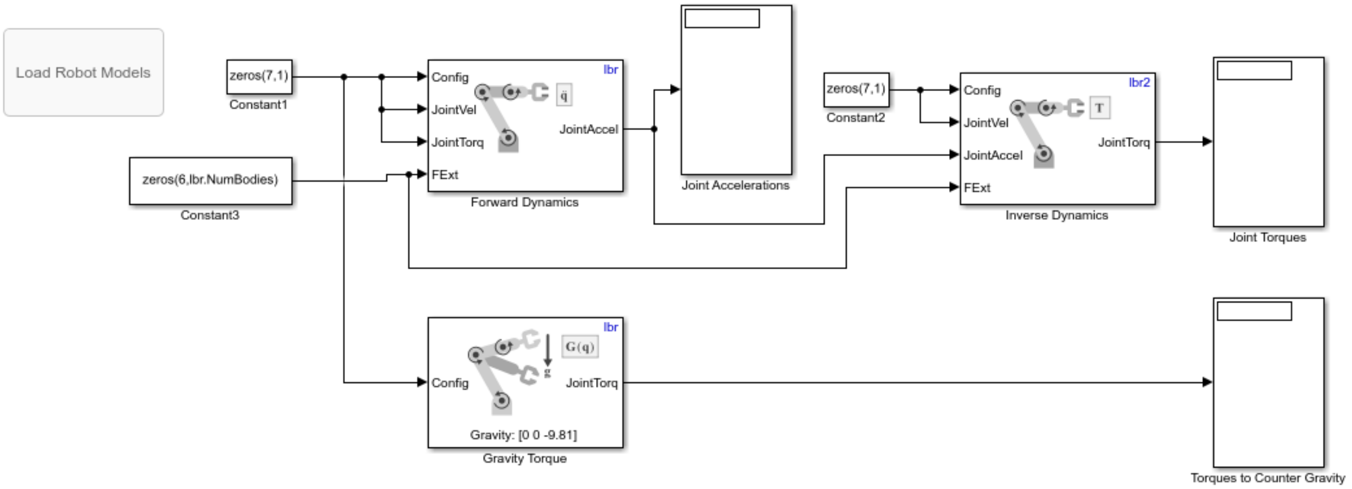
This example shows how to use the manipulator algorithm blocks to compute and compare dynamics due to gravity for a manipulator robot.

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

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

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

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



Copyright 2018 The MathWorks, Inc.

The Forward Dynamics block calculates the joint accelerations due to gravity for a given `lbr` robot configuration with no initial velocity, torque, or external force. The Inverse Dynamics block then computes the torques needed for the joint to create those same accelerations with no gravity by using the `lbr2` robot. Finally, the Gravity Torque block calculates the torque required to counteract gravity for the `lbr` robot.

Run the model. Besides some small numerical differences, the gravity torque and the torque required for accelerations due to gravity are the same value with opposite directions.

See Also

Blocks

Forward Dynamics | Inverse Dynamics | Get Jacobian | Gravity Torque | Joint Space Mass Matrix | Velocity Product Torque

Classes
`rigidBodyTree`

Functions
`externalForce` | `importrobot` | `homeConfiguration` | `randomConfiguration`

Related Examples

- “Perform Safe Trajectory Tracking Control Using Robotics Manipulator Blocks” on page 1-268

Trace An End-Effector Trajectory with Inverse Kinematics in Simulink

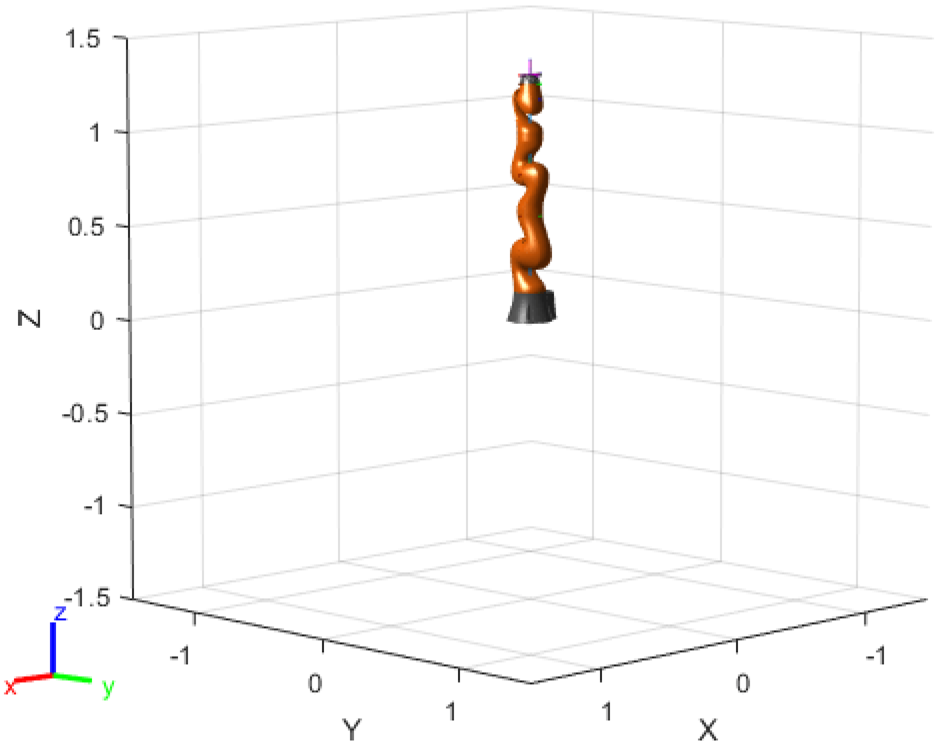
Use a rigid body robot model to compute inverse kinematics using Simulink®. Define a trajectory for the robot end effector and loop through the points to solve robot configurations that trace this trajectory.

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

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

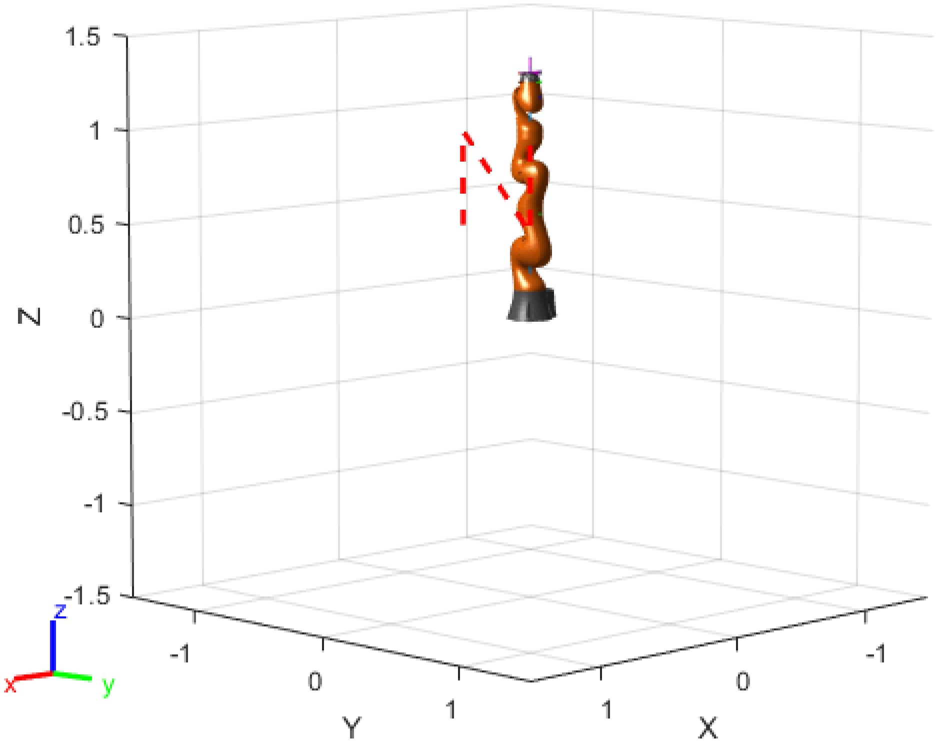
View the robot.

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



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

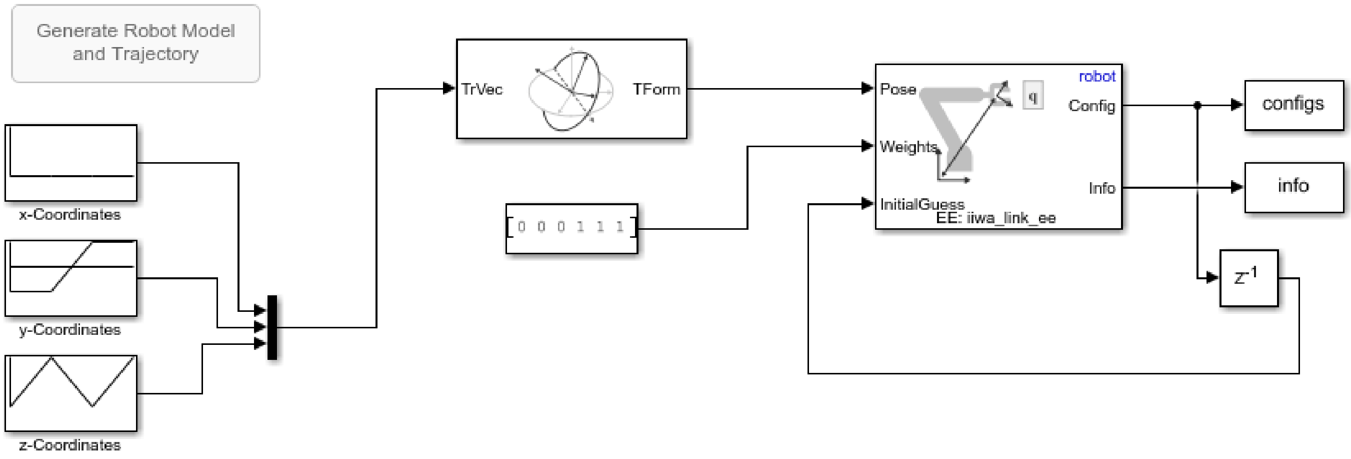
```
x = 0.5*zeros(1,4)+0.25;
y = 0.25*[-1 -1 1 1];
z = 0.25*[-1 1 -1 1] + 0.75;

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



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

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

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

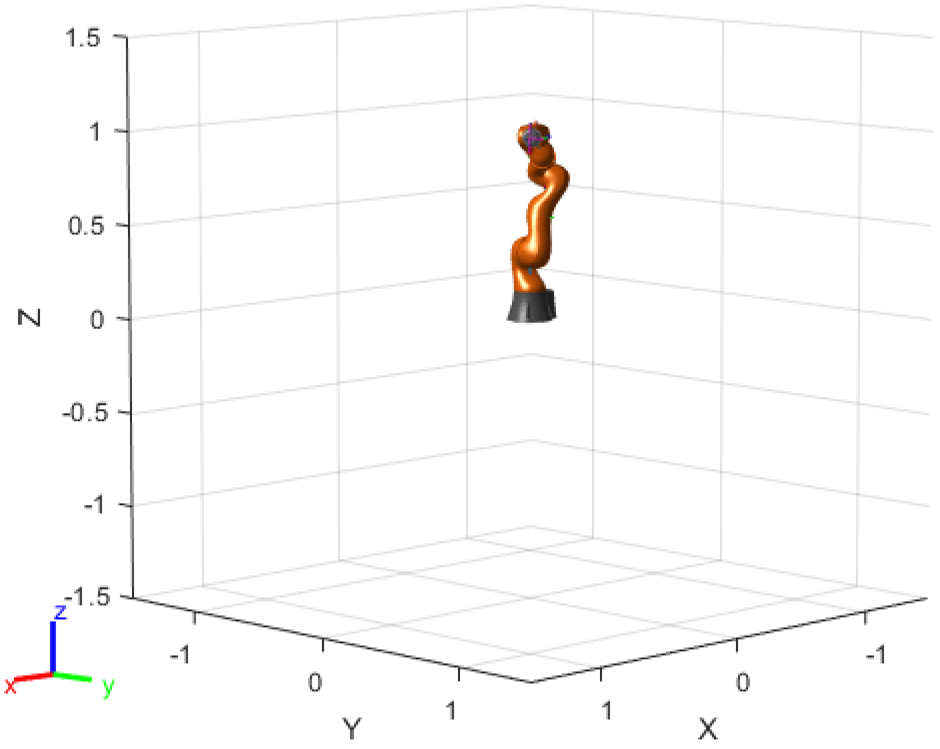


Run the simulation. The model should generate the robot configurations (`configs`) that follow the specified trajectory for the end effector.

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

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

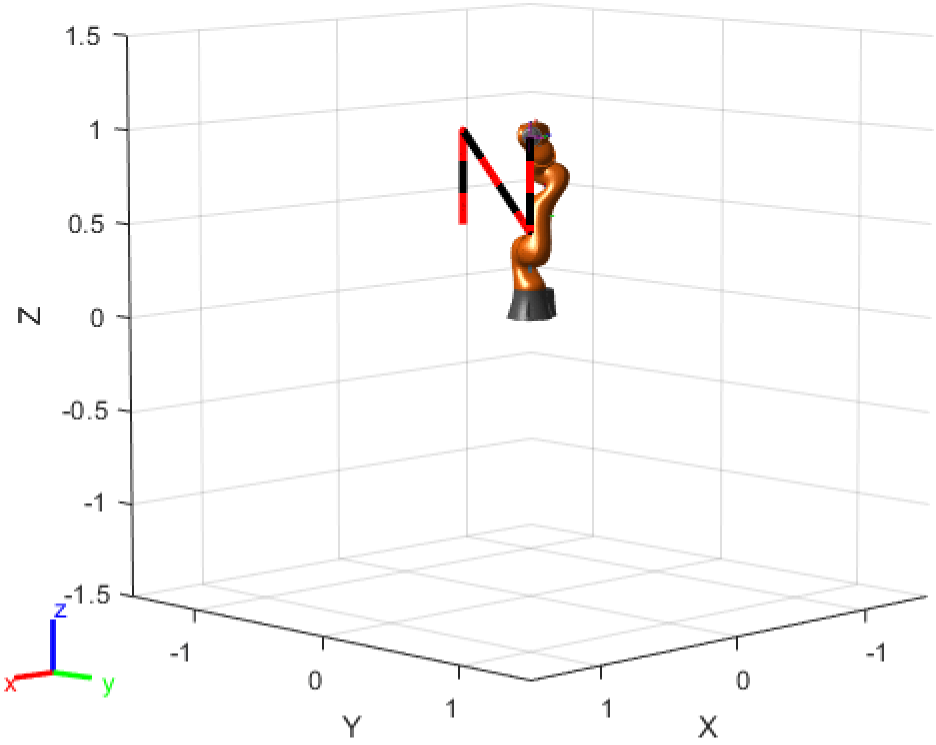
```
figure('Visible','on');
tformIndex = 1;
for i = 1:10:numel(configs.Data)/7
    currConfig = configs.Data(:,1,i);
    show(robot,currConfig);
    drawnow

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



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

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

hold on
plot3(xyz(:,1),xyz(:,2),xyz(:,3),'-k','LineWidth',3);
```

```
plot3(x,y,z,'--r','LineWidth',3)
hold off
```



See Also

Objects
inverseKinematics | rigidBodyTree | generalizedInverseKinematics

Blocks
Inverse Kinematics | Get Transform | Inverse Dynamics

Related Examples

- “Control PR2 Arm Movements Using ROS Actions and Inverse Kinematics” on page 1-193
- “Inverse Kinematics Algorithms” on page 2-10

Get Mass Matrix for Manipulators in Simulink

This example shows how to calculate the mass matrix for a robot manipulator using a `rigidBodyTree` model. In this example, you define a robot model and robot configurations in MATLAB® and pass them to Simulink® to be used with the manipulator algorithm blocks.

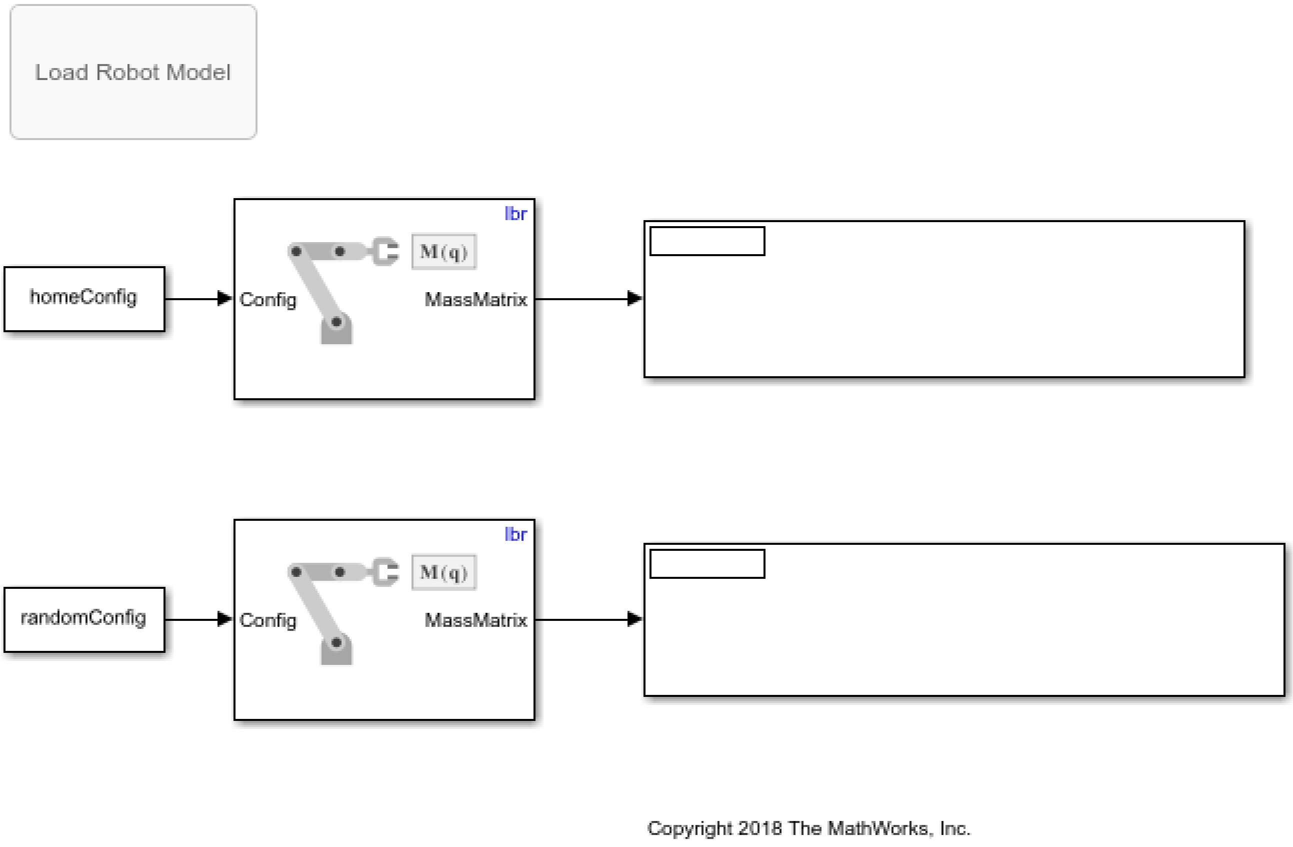
Load a `RigidBodyTree` object that models a KUKA LBR robot. Use the `homeConfiguration` functions to get the home configuration or home joint positions of the robot. Use the `randomConfiguration` function to generate a random configuration within the robot joint limits.

```
load('exampleRobots.mat','lbr')
lbr.DataFormat = 'column';

homeConfig = homeConfiguration(lbr);
randomConfig = randomConfiguration(lbr);
```

Open the model. If necessary, use the **Load Robot Model** callback button to reload the robot model and configuration vectors.

The Joint Space Mass Matrix block calculates the mass matrix for the given configuration.

```
open_system('mass_matrix_example.slx')
```



Run the model to display the mass matrices for each configuration.

See Also

Blocks
Get Transform | Forward Dynamics | Inverse Dynamics | Get Jacobian | Gravity Torque

Classes
`rigidBodyTree`

Functions
`importrobot` | `homeConfiguration` | `randomConfiguration`

Related Examples

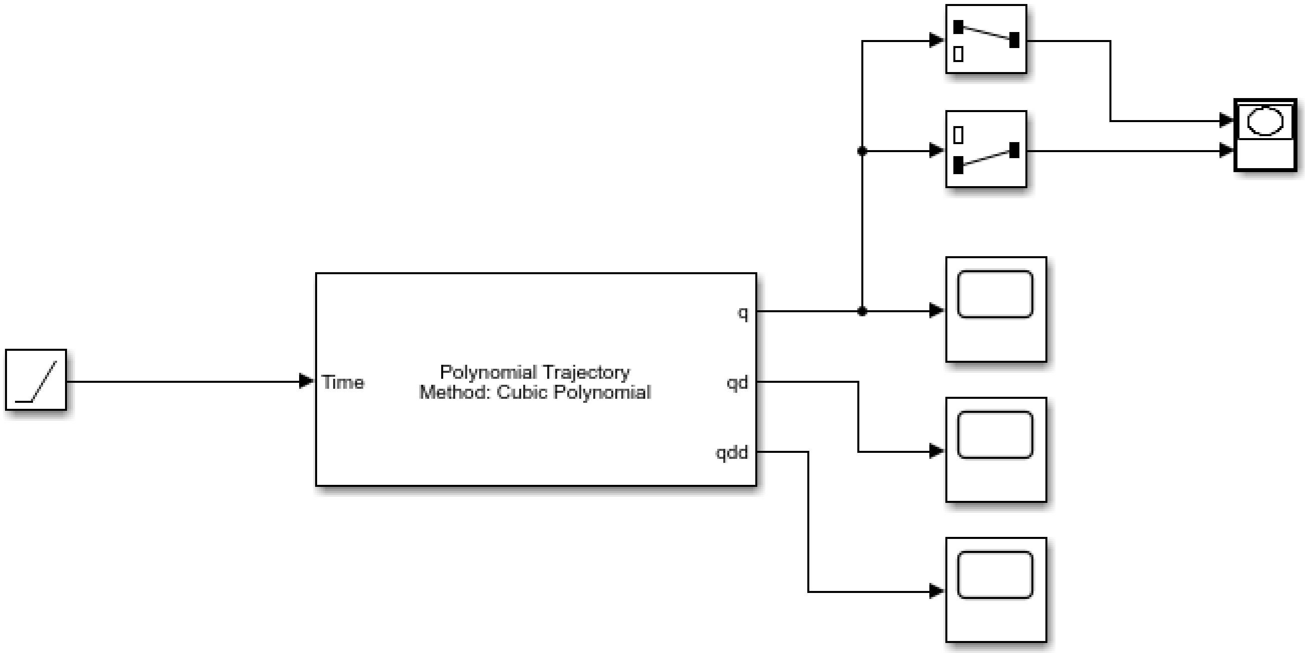
- “Perform Safe Trajectory Tracking Control Using Robotics Manipulator Blocks” on page 1-268

Generate Cubic Polynomial Trajectory

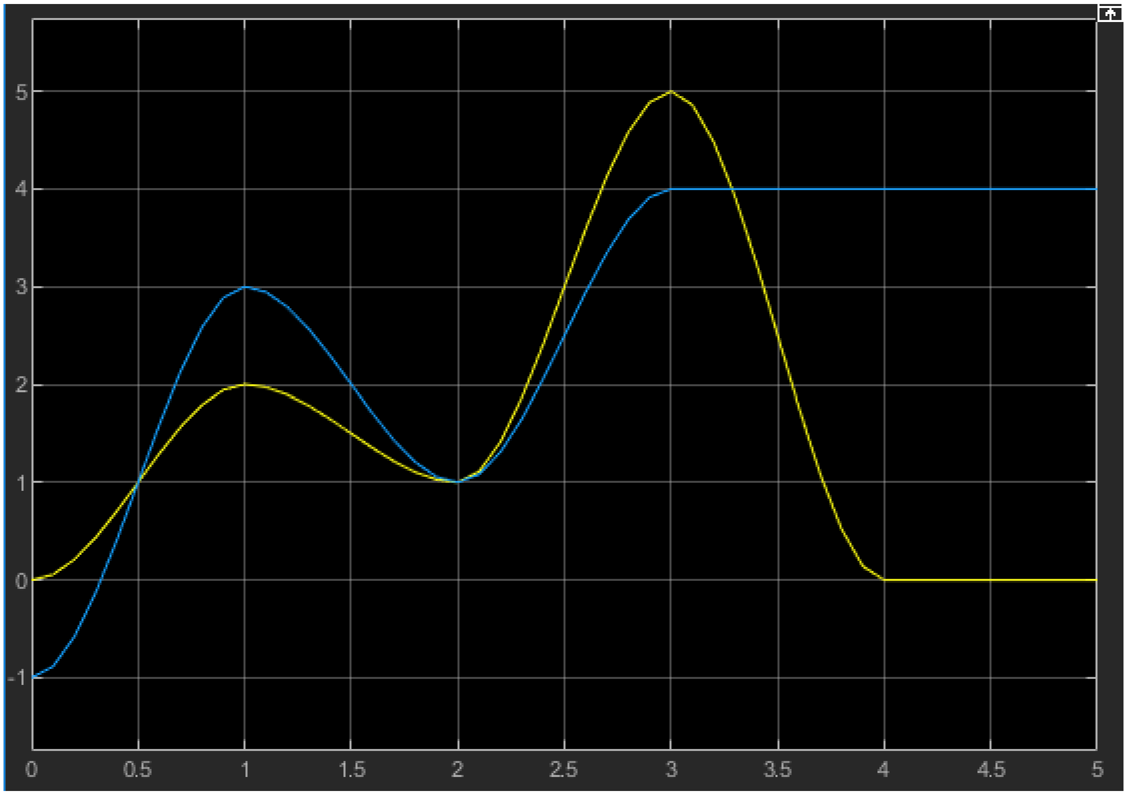
This example shows how to generate a cubic polynomial trajectory using the **Polynomial Trajectory** block.

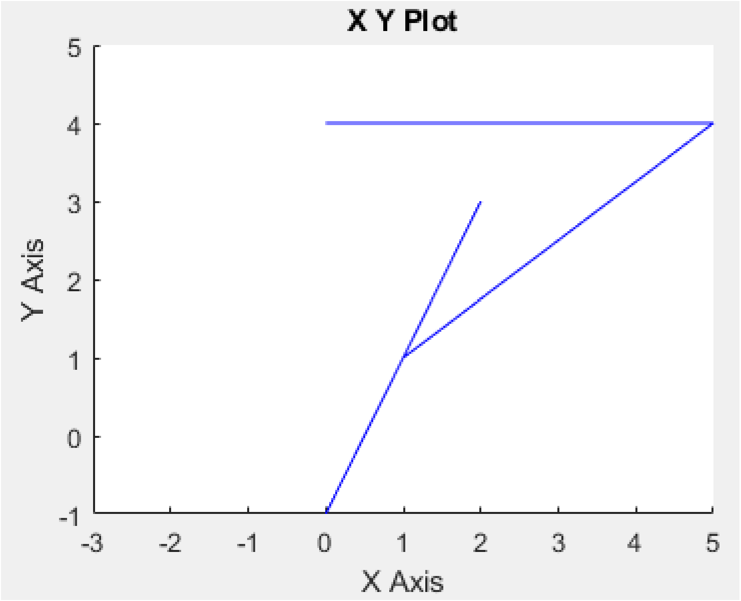
Open the model. The block has a set of 2-D waypoints defined in the block mask. The **Time** input is just a ramp signal to simulate time progressing.

```
open_system('cubic_polytraj_ex1.slx')
```



Run the simulation. The first figure shows the output of the q vector for the positions of the trajectory. Notice the cubic polynomial shape to the trajectory between waypoints. The **XY Plot** shows the actual 2-D trajectory, which hits the waypoints specified in the block mask.


5
4
3
2
1
0
-1
0
0.5
1
1.5
2
2.5
3
3.5
4
4.5
5

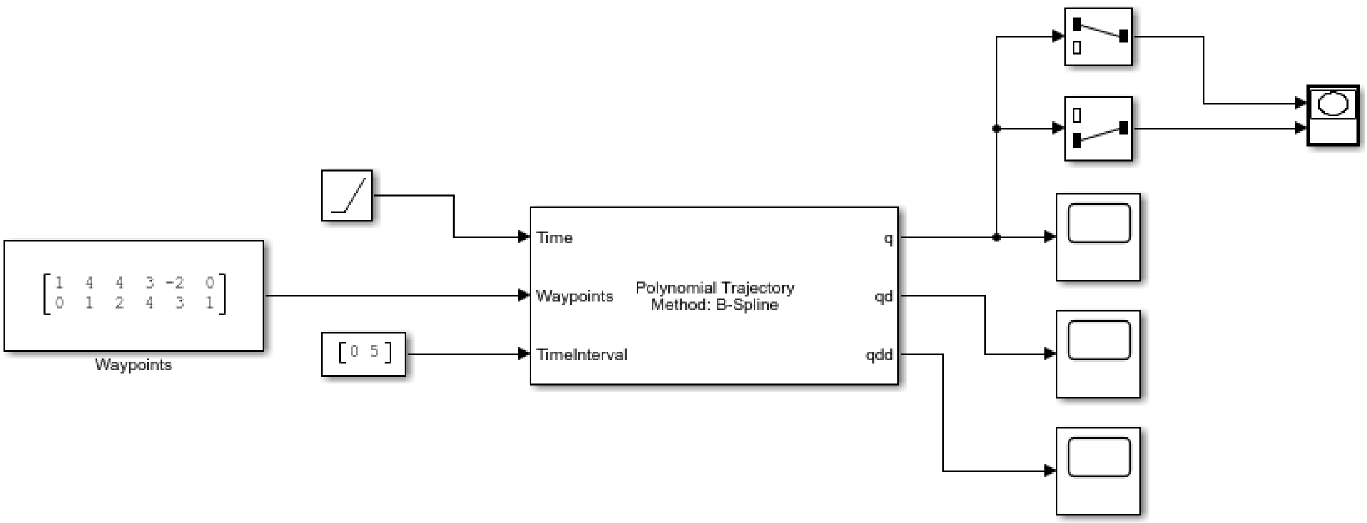

X Y Plot
Y Axis
X Axis
5
4
3
2
1
0
-1
-3
-2
-1
0
1
2
3
4
5

Generate B-Spline Trajectory

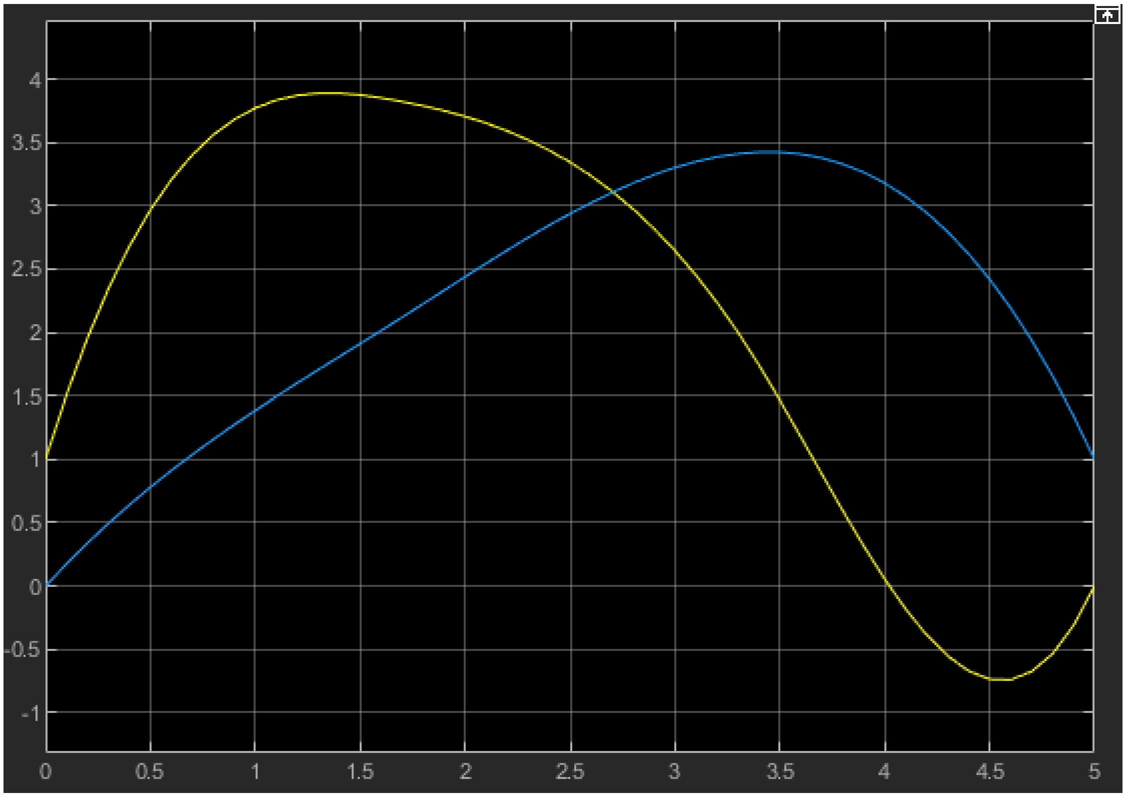
This example shows how to generate a B-spline trajectory using the **Polynomial Trajectory** block.

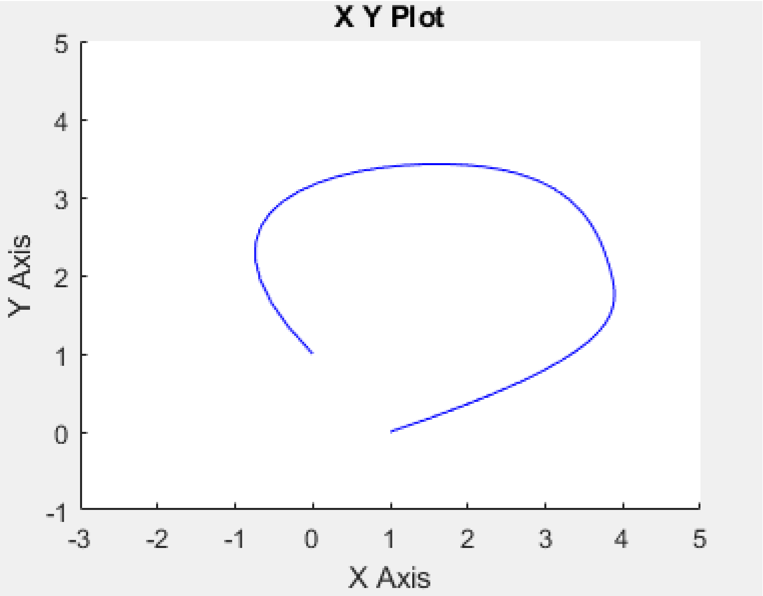
Open the model. The **Waypoints** and **TimeInterval** inputs are toggled in the block mask by setting **Waypoint source** to `External`. For B-splines, the waypoints are actually control points for the convex polygon, but the first and last waypoints are met. The **Time** input is just a ramp signal to simulate time progressing.

```
open_system('bspline_polytraj_ex1.slx')
```



Run the simulation. The first figure shows the output of the `q` vector for the positions of the trajectory. The **X Y Plot** shows the actual 2-D trajectory, which stays inside the defined control points and hits the first and last waypoints.


4
3.5
3
2.5
2
1.5
1
0.5
0
-0.5
-1
0
0.5
1
1.5
2
2.5
3
3.5
4
4.5
5

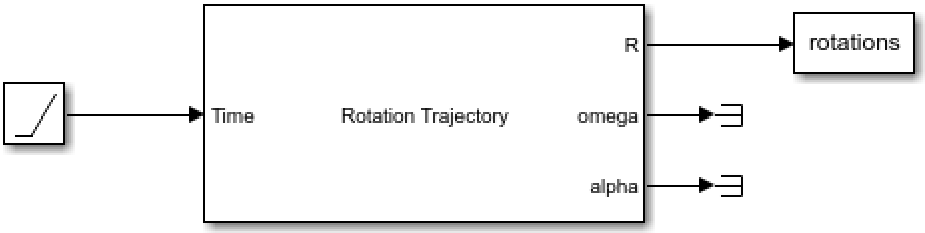

X Y Plot
Y Axis
5
4
3
2
1
0
-1
-3
-2
-1
0
1
2
3
4
5
X Axis

Generate Rotation Trajectory

This example shows how to generate a trajectory that interpolates between rotations using the **Rotation Trajectory** block.

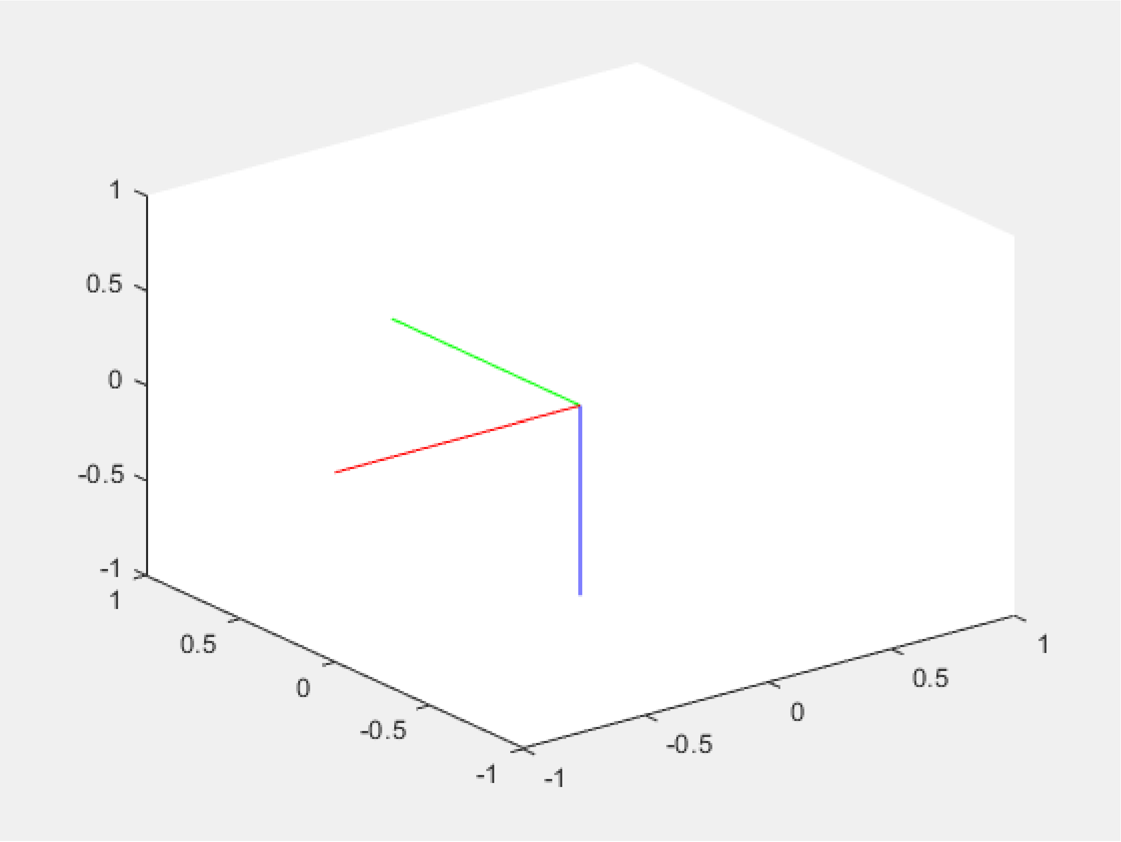
Open and simulate the model. The **Rotation Trajectory** block outputs the trajectory between two rotations and saves the intermediate rotations to the `rotations` variable. This example generates a simple rotation trajectory from the *x*-axis to the *z*-axis.

```
open_system('rot_traj_ex1.slx')
simOut = sim('rot_traj_ex1.slx');
```



Use `plotTransforms` to plot the rotation trajectory.

```
numRotations = size(simOut.rotations,3);
translations = zeros(3,numRotations);
figure("Visible","on")

for i = 1:numRotations
    plotTransforms(translations(:,i)',simOut.rotations(:,i)')
    xlim([-1 1])
    ylim([-1 1])
    zlim([-1 1])
    drawnow
    pause(0.1)
end
```


1
0.5
0
-0.5
-1
1
0.5
0
-0.5
-1
-1
-0.5
0
0.5
1

Use Custom Time Scaling for a Rotation Trajectory

This example shows how to specify custom time-scaling in the **Rotation Trajectory** block to execute an interpolated trajectory. Two rotations are specified in the block to generate a trajectory between them. The goal is to move between rotations using a nonlinear time scaling with more time samples closer to the final rotation.

Specify the Time Scaling

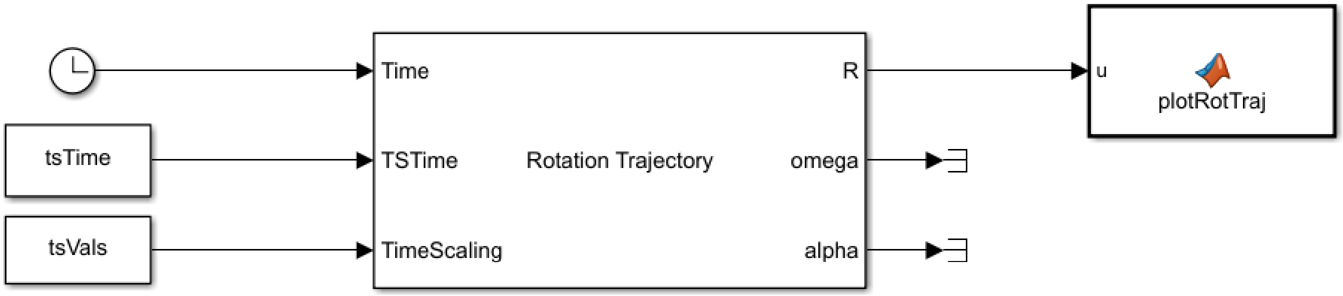
Create vectors for the time scaling time vector and time scaling values. The time scaling time is linear vector from 0 to 5 seconds at 0.1 second intervals. The time scaling values follow a cubic trajectory with the appropriate derivatives specified for velocity and acceleration. These values are used in the model.

```
tsTime = 0:0.1:5;
tsVals(1,:) = (tsTime/5).^3;         % Position
tsVals(2,:) = ((3/125).*tsTime).^2;  % Velocity
tsVals(3,:) = (18/125^2).*tsTime;    % Acceleration
```

Open the Model

The **Clock** block outputs simulation time and is used for querying the rotation trajectory at those specify time points. The full set of time scaling time and values are input to the **Rotation Trajectory** block, but the **Time** input defined when to sample from this trajectory. The MATLAB® function block uses `plotTransforms` to plot a coordinate frame that moves along the generated rotation trajectory.

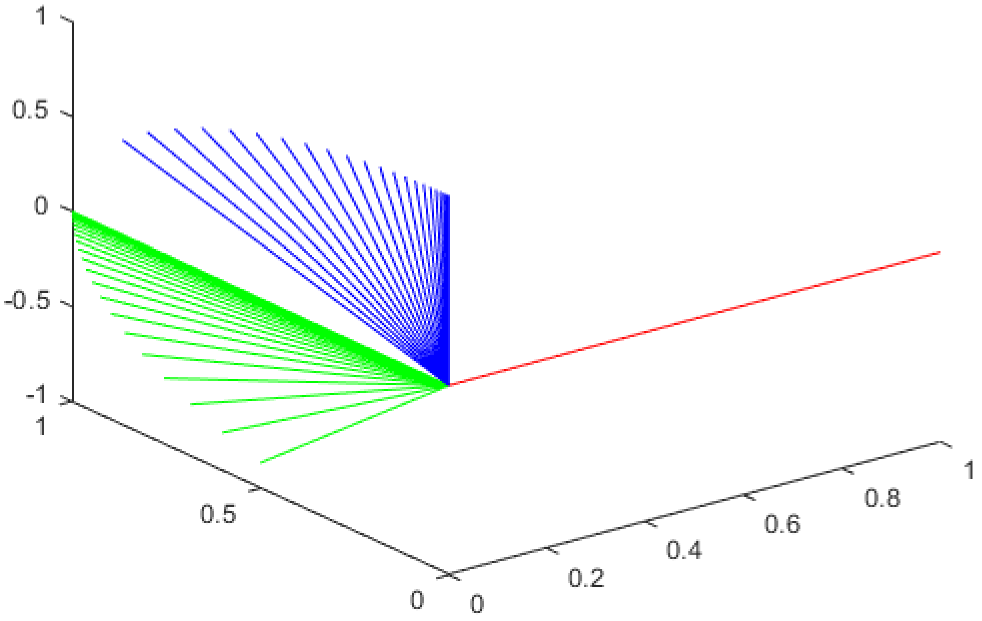
```
open_system("custom_time_scaling_rotation")
```



Simulate the Model

Simulate the model. The plot shows how the rotation follows a nonlinear interpolated trajectory parameterized in time. The model runs with a fixed-step solver at an interval of 0.1 seconds, so each frame is 0.1 seconds apart. Notice that the transformations are sampled more closely near the final rotation.

```
sim("custom_time_scaling_rotation")
hold off
```


1
0.5
0
-0.5
-1
1
0.5
0
0
0.2
0.4
0.6
0.8
1

Execute Transformation Trajectory Using Manipulator and Inverse Kinematics

This example shows how to generate a transformation trajectory using the **Transform Trajectory** block and execute it for a manipulator robot using inverse kinematics.

Generate two homogenous transformations for the start and end points of the trajectory.

```
tform1 = trvec2tform([0.25 -0.25 1])
```

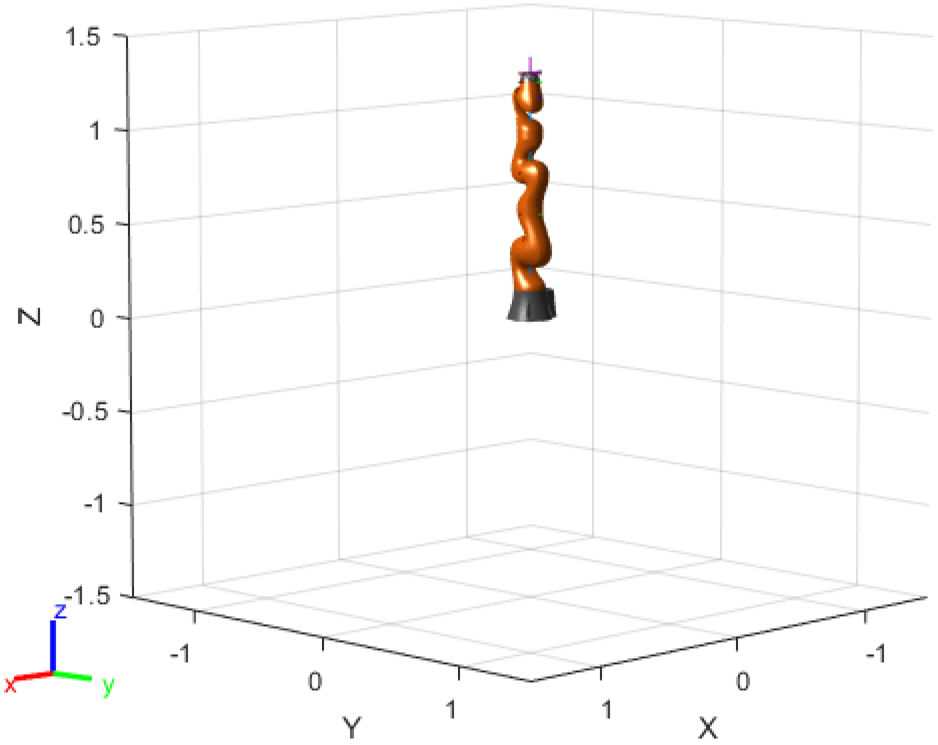
```
tform1 = 4×4

    1.0000         0         0    0.2500
         0    1.0000         0   -0.2500
         0         0    1.0000    1.0000
         0         0         0    1.0000
```

```
tform2 = trvec2tform([0.25 0.25 0.5])
```

```
tform2 = 4×4

    1.0000         0         0    0.2500
         0    1.0000         0    0.2500
         0         0    1.0000    0.5000
         0         0         0    1.0000
```

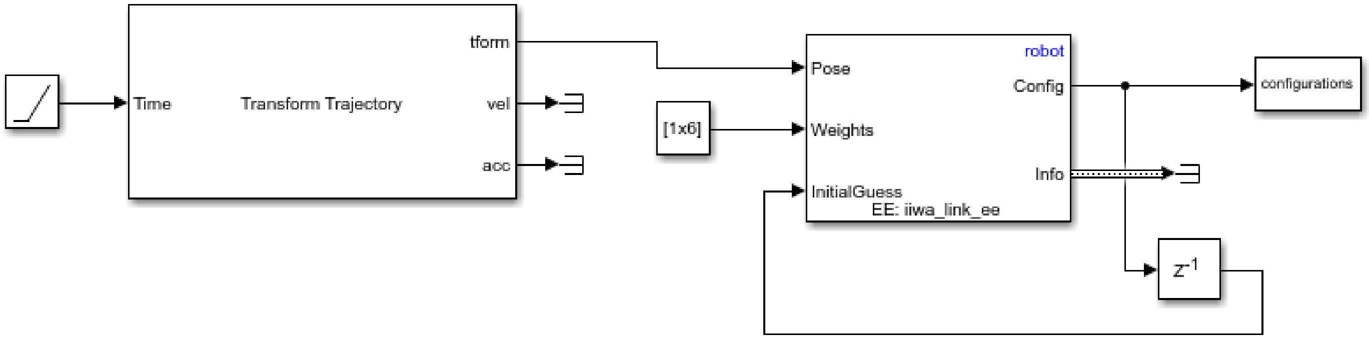
Import the robot model and specify the data format for Simulink®.

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



Open the model. The **Transform Trajectory** block interpolates between the initial and final transformation specified in the block mask. These transformations are fed to the **Inverse Kinematics** block to solve for the robot configuration that makes the end effector reach the desired transformation. The configurations are output to the workspace as `configurations`.

```
open_system('transform_traj_ex1.slx')
```



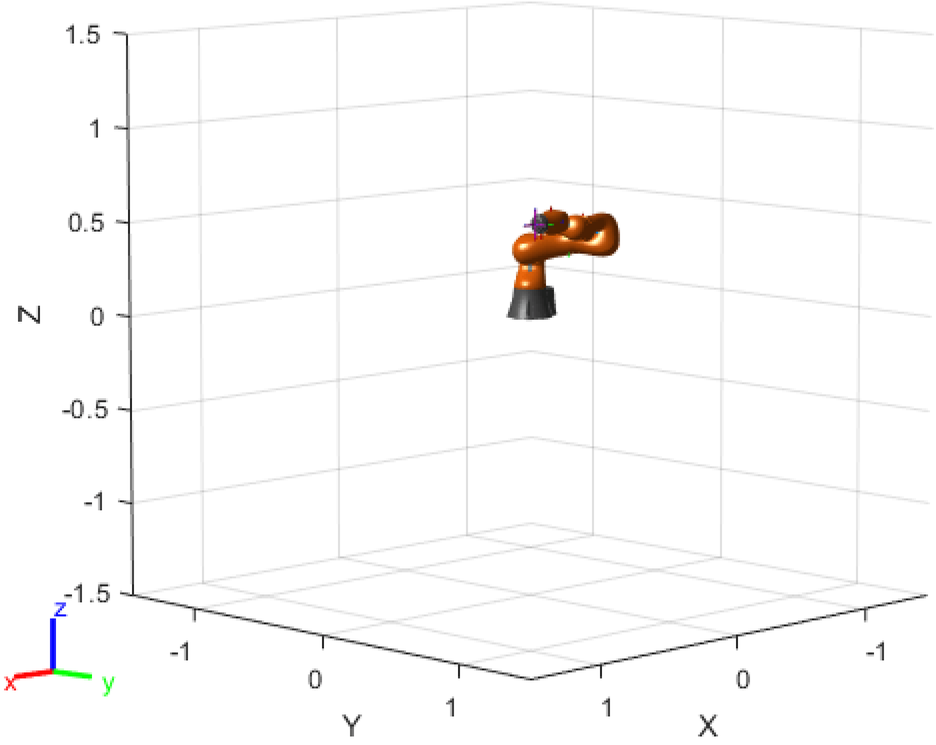
Run the simulation and get the robot configurations.

```
simOut = sim('transform_traj_ex1.slx')
```

```
simOut = 
  Simulink.SimulationOutput:
         configurations: [7x1x52 double]
                   tout: [52x1 double]

     SimulationMetadata: [1x1 Simulink.SimulationMetadata]
           ErrorMessage: [0x0 char]
```

Show the robot configurations to animate the robot going through the trajectory.

```
for i = 1:numel(simOut.configurations)/7
    currConfig = simOut.configurations(:,:,i);
    show(robot,currConfig);
    drawnow
end
```



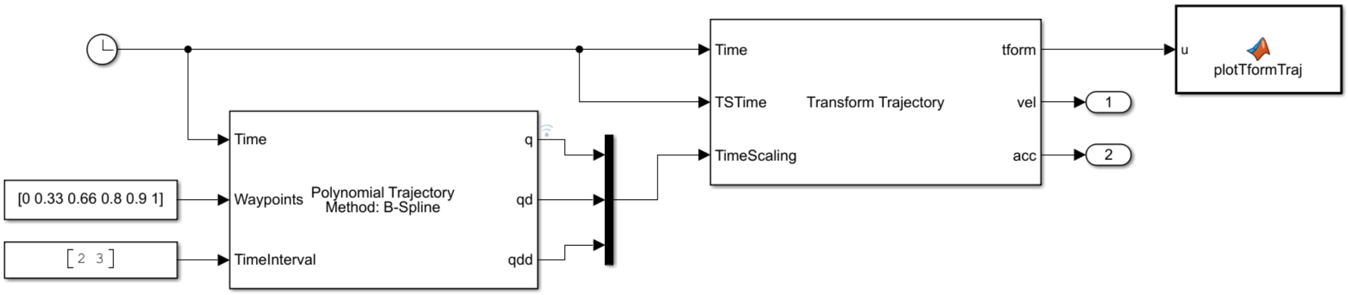
Use Custom Time Scaling for a Transform Trajectory

This example shows how to specify custom time-scaling in the **Transform Trajectory** block to execute an interpolated trajectory. Two transformations are specified in the block to generate a trajectory between the two. The goal is to move between transforms using a nonlinear time scaling where the trajectory moves quickly at the start and slowly at the end.

Open the Model

A custom time scaling trajectory is generated using the **Polynomial Trajectory** block, which gives the position, velocity, and acceleration defined by the custom time scaling at the instant in time, as given by the **Clock** block. The **Clock** block outputs simulation time and is used for querying the transformation trajectory at those specify time points. The input **Waypoints** define the waypoints of the nonlinear time scaling to use and includes a shorter time interval between points near the final time. The 3x1 time scaling, output from the **Polynomial Trajectory** block as **q**, **qd**, and **qdd**, is input to the **Transform Trajectory** block with the current clock time as the **TSTime**, which indicates this is the time scaling at that instance. The MATLAB® function block uses `plotTransforms` to plot a coordinate frame that moves along the generated transformation trajectory.

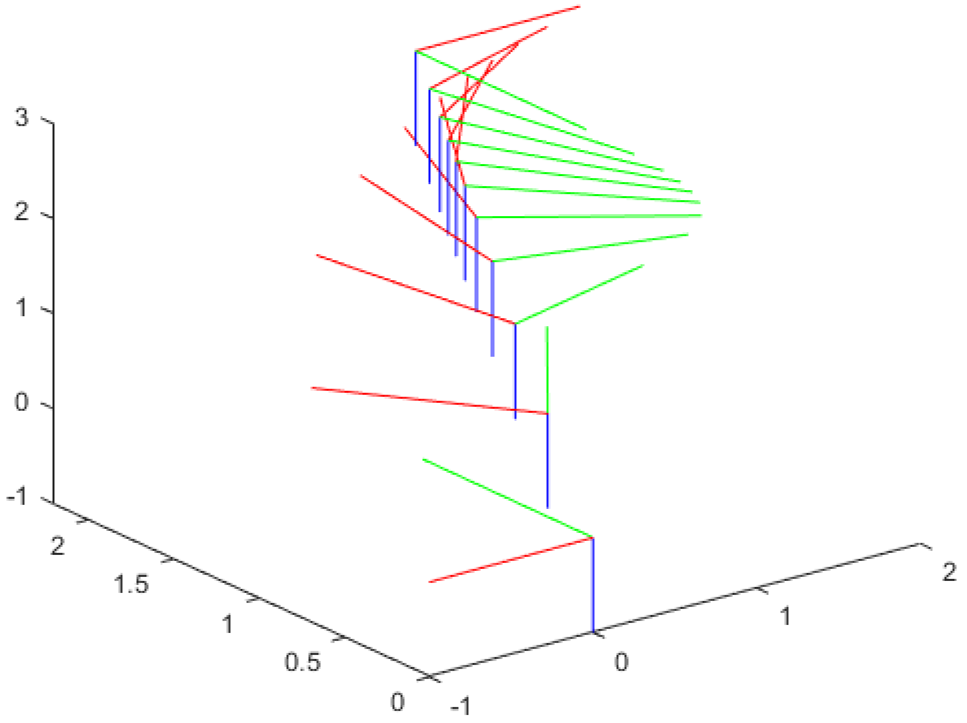
```
open_system("custom_time_scaling_transform")
```



Simulate the Model

Simulate the model. The plot shows how the transformation follows a nonlinear interpolated trajectory parameterized in time. The model runs with a fixed-step solver at an interval of 0.1 seconds, so each frame is 0.1 seconds apart. Notice that the transformations are sampled more closely near the final transformation.

```
sim("custom_time_scaling_transform")
```

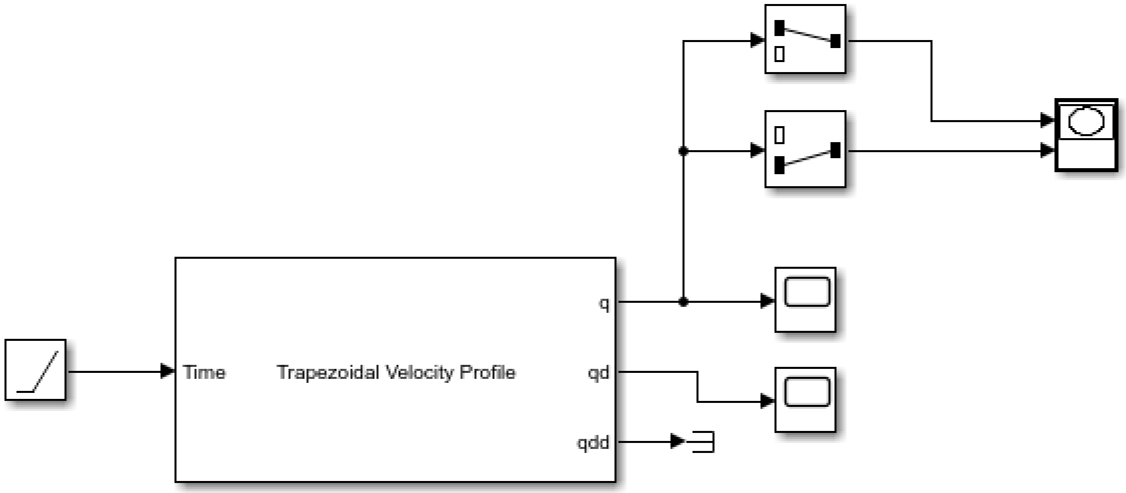

3
2
1
0
-1
2
1.5
1
0.5
0
-1
0
1
2

Generate Trapezoidal Velocity Trajectory

This example shows how to generate a trapezoidal velocity trajectory using the **Trapezoidal Velocity** block.

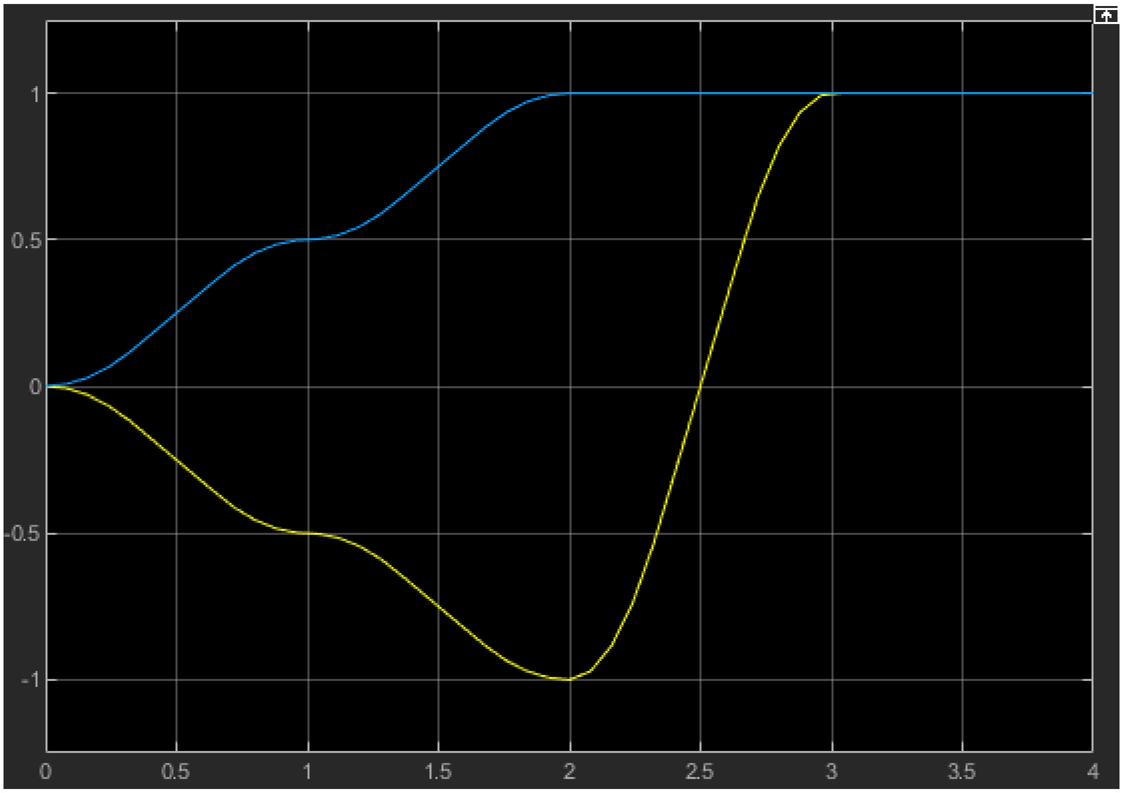
Open the model. The waypoints are specified in the block mask. The position and velocity outputs are connect to scopes and the position is plotted to an **XY Plot**. The **Time** input is just a ramp signal to simulate time progressing.

```
open_system('trapvel_traj_ex1.slx')
```

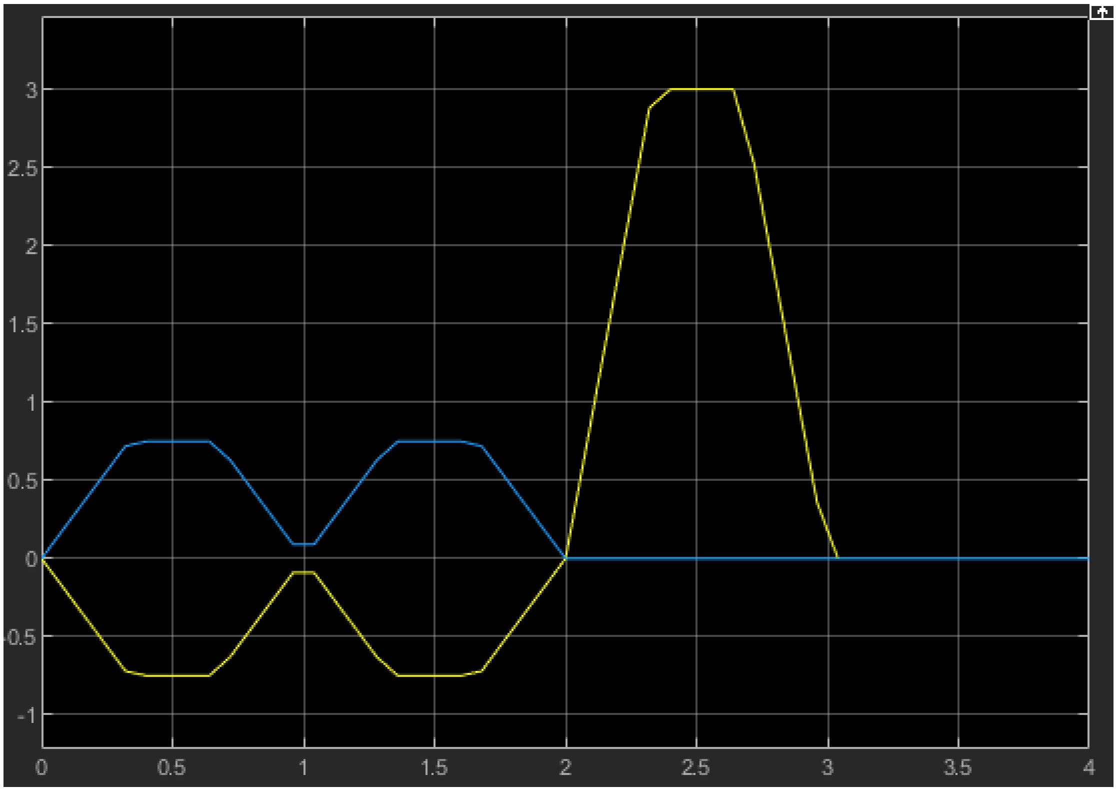


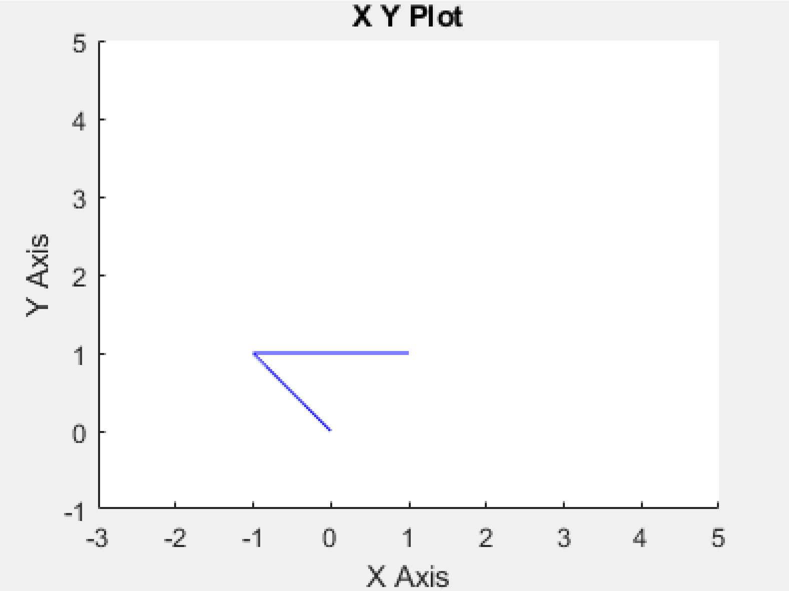
Run the Simulation. The first figure shows the output of the q vector for the positions of the trajectory. The second figure shows the qdd vector for the velocity. Notice the trapezoidal profile for each waypoint transition. The **XY Plot** shows the actual 2-D trajectory, which hits the specified waypoints.

Positions



Velocities




X Y Plot
Y Axis
5
4
3
2
1
0
-1
-3
-2
-1
0
1
2
3
4
5
X Axis

Compute Velocity Product for Manipulators in Simulink

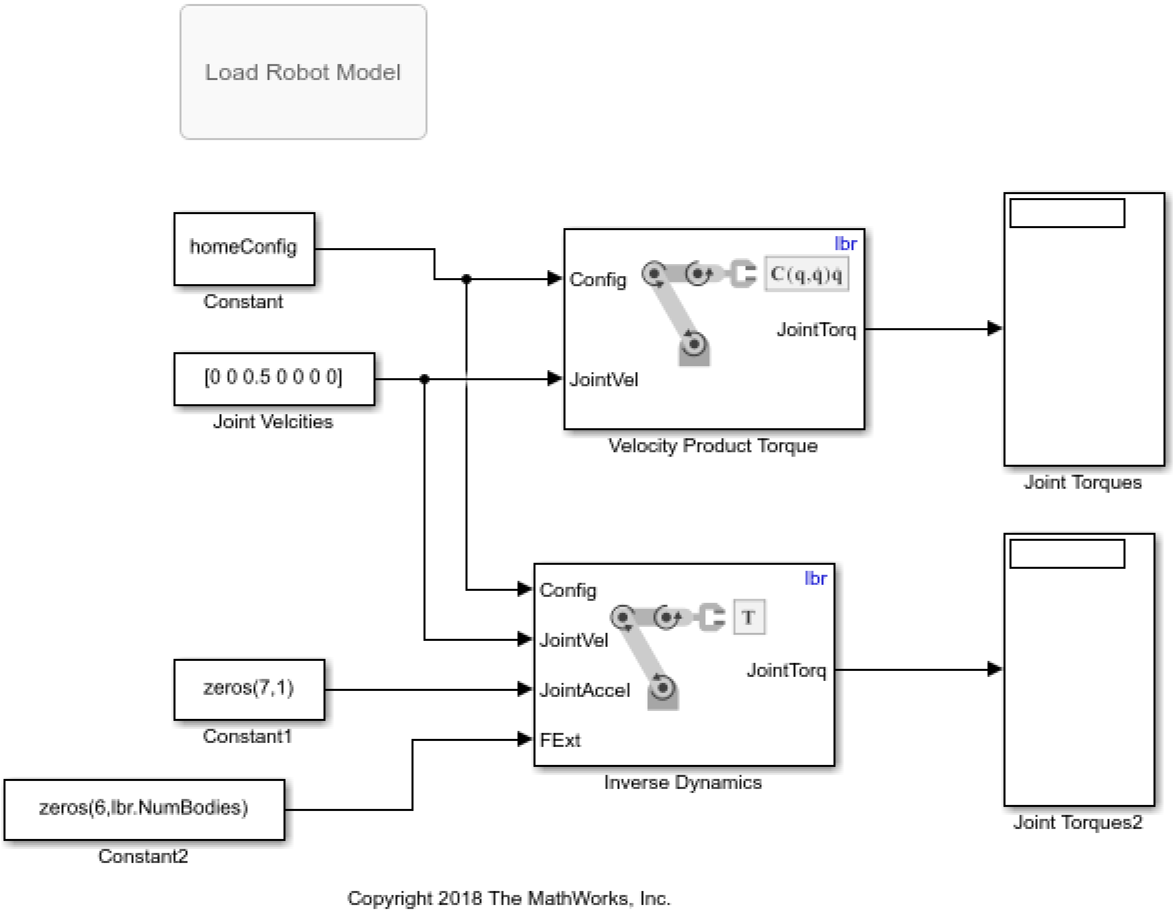
This example shows how to calculate the velocity-induced torques for a robot manipulator by using a `rigidBodyTree` model. In this example, you define a robot model and robot configuration in MATLAB® and pass them to Simulink® to be used with the manipulator algorithm blocks.

Load a `RigidBodyTree` object that models a KUKA LBR robot. Use the `homeConfiguration` function to get the home configuration or home joint positions of the robot.

```
load('exampleLBR.mat','lbr')
lbr.DataFormat = 'column';

homeConfig = homeConfiguration(lbr);
```

Open the model. If necessary, use the **Load Robot Model** callback button to reload the robot model and configuration vector.

```
open_system('velocity_product_example.slx')
```



Run the model. The Velocity Product block calculates the torques induced by the given velocities. Verify these values by passing the same velocities to the Inverse Dynamics block with no acceleration or external forces.

See Also

Blocks
Forward Dynamics | Inverse Dynamics | Get Jacobian | Gravity Torque | Joint Space Mass Matrix

Classes
`rigidBodyTree`

Functions
`externalForce` | `importrobot` | `homeConfiguration` | `randomConfiguration`

Related Examples

- “Perform Safe Trajectory Tracking Control Using Robotics Manipulator Blocks” on page 1-268

Plan Path for a Unicycle Robot in Simulink

This example demonstrates how to execute an obstacle-free path between two locations on a given map in Simulink®. The path is generated using a probabilistic road map (PRM) planning algorithm (`mobileRobotPRM`). Control commands for navigating this path are generated using the **Pure Pursuit** controller block. A unicycle kinematic motion model simulates the robot motion based on those commands.

Load the Map and Simulink Model

Load the occupancy map, which defines the map limits and obstacles within the map. `exampleMaps.mat` contain multiple maps including `simpleMap`, which this example uses.

```
load exampleMaps.mat
```

Specify a start and end locaiton within the map.

```
startLoc = [5 5];
goalLoc = [12 3];
```

Model Overview

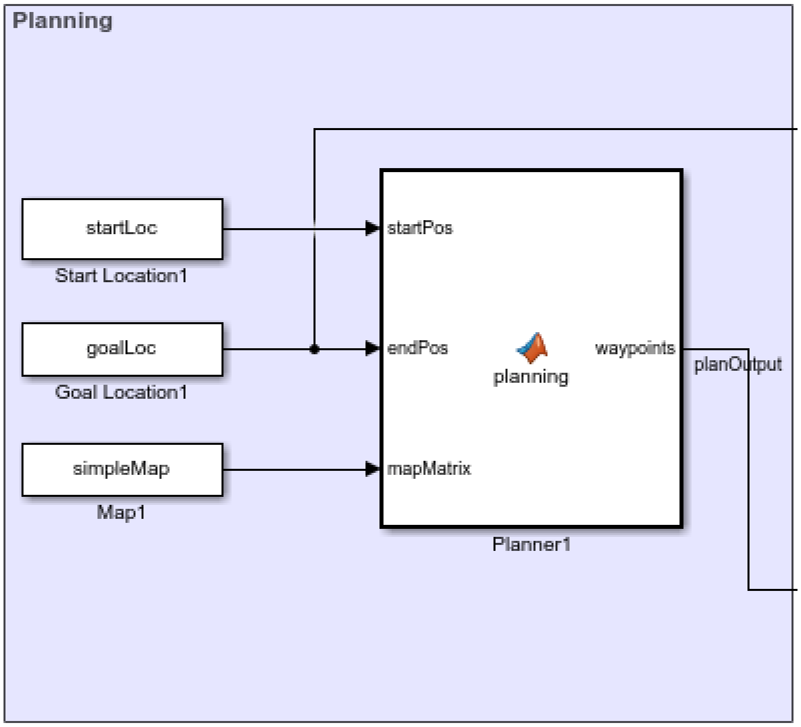
Open the Simulink Model

```
open_system('pathPlanningUnicycleSimulinkModel.slx')
```

The model is composed of three primary parts:

- **Planning**
- **Control**
- **Plant Model**

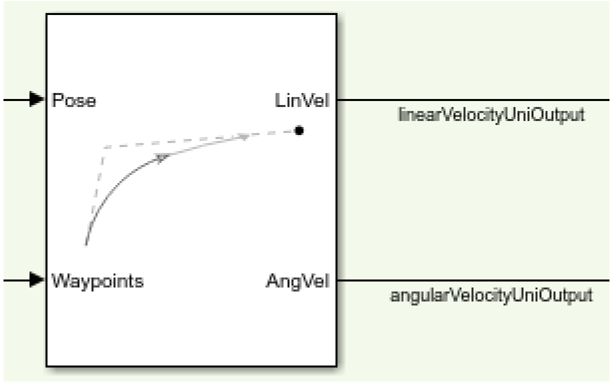
Planning



The **Planner** MATLAB® function block uses the `mobileRobotPRM` path planner and takes a start location, goal location, and map as inputs. The blocks outputs an array of waypoints that the robot follows. The planned waypoints are used downstream by the **Pure Pursuit** controller block.

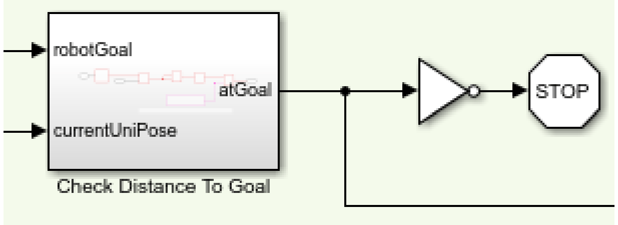
Control

Pure Pursuit



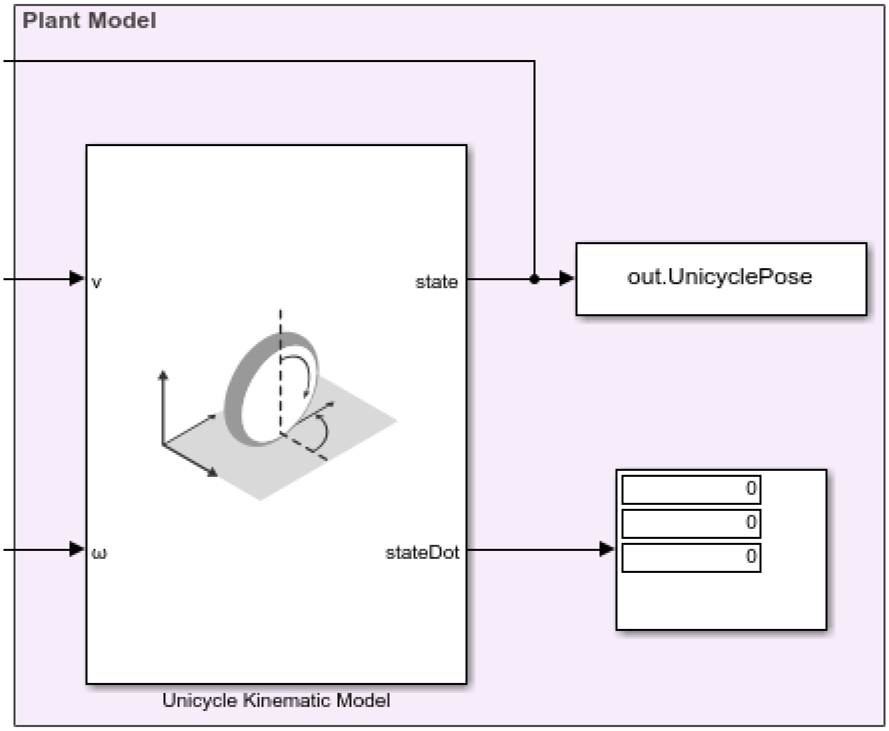
The **Pure Pursuit** controller block generates the linear velocity and angular velocity commands based on the waypoints and the current pose of the robot.

Check if Goal is Reached



The **Check Distance to Goal** subsystem calculates the current distance to the goal and if it is within a threshold, the simulation stops.

Plant Model



The **Unicycle Kinematic Model** block creates a vehicle model to simulate simplified vehicle kinematics. The block takes linear and angular velocities as command inputs from the **Pure Pursuit** controller block, and outputs the current position and velocity states.

Run the Model

To simulate the model

```
simulation = sim('pathPlanningUnicycleSimulinkModel.slx');
```

Visualize The Motion of Robot

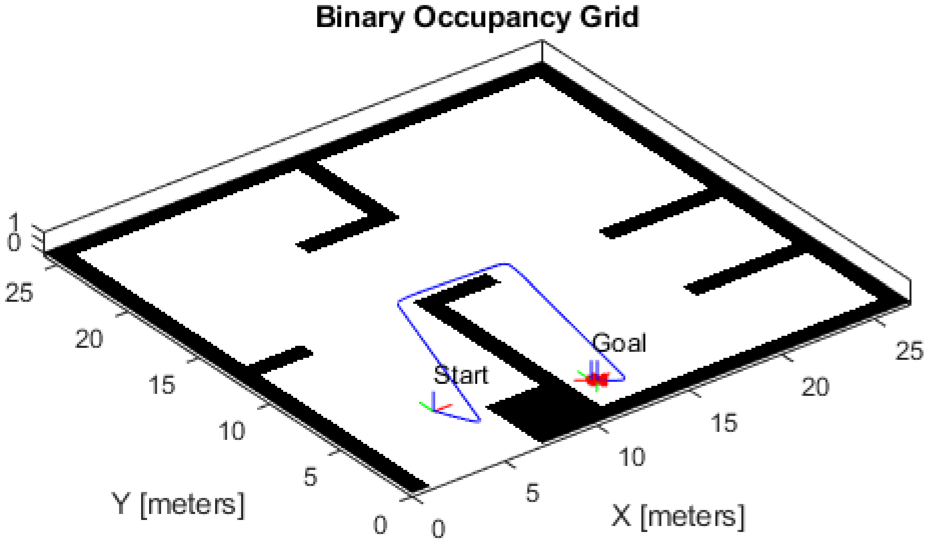
After simulating the model, visualize the robot driving the obstacle-free path in the map.

```
map = binaryOccupancyMap(simpleMap)
map = 
  binaryOccupancyMap with properties:

   mapLayer Properties
              LayerName: 'binaryLayer'
               DataType: 'logical'
           DefaultValue: 0
    GridLocationInWorld: [0 0]
      GridOriginInLocal: [0 0]
     LocalOriginInWorld: [0 0]
             Resolution: 1
               GridSize: [26 27]
           XLocalLimits: [0 27]
           YLocalLimits: [0 26]
           XWorldLimits: [0 27]
           YWorldLimits: [0 26]
```

```
robotPose = simulation.UnicyclePose
```

```
robotPose = 428×3

    5.0000    5.0000         0
    5.0000    5.0000   -0.0002
    5.0001    5.0000   -0.0012
    5.0006    5.0000   -0.0062
    5.0031    5.0000   -0.0313
    5.0156    4.9988   -0.1569
    5.0707    4.9707   -0.7849
    5.0945    4.9354   -1.1140
    5.1075    4.9059   -1.1828
    5.1193    4.8759   -1.2030
       ⋮
```

```
numRobots = size(robotPose, 2) / 3;
thetaIdx = 3;

% Translation
xyz = robotPose;
xyz(:, thetaIdx) = 0;

% Rotation in XYZ euler angles
theta = robotPose(:,thetaIdx);
thetaEuler = zeros(size(robotPose, 1), 3 * size(theta, 2));
thetaEuler(:, end) = theta;

for k = 1:size(xyz, 1)
    show(map)
    hold on;

    % Plot Start Location
    plotTransforms([startLoc, 0], eul2quat([0, 0, 0]))
    text(startLoc(1), startLoc(2), 2, 'Start');

    % Plot Goal Location
    plotTransforms([goalLoc, 0], eul2quat([0, 0, 0]))
    text(goalLoc(1), goalLoc(2), 2, 'Goal');
    % Plot Robot's XY locations
    plot(robotPose(:, 1), robotPose(:, 2), '-b')

    % Plot Robot's pose as it traverses the path
    quat = eul2quat(thetaEuler(k, :), 'xyz');
    plotTransforms(xyz(k,:), quat, 'MeshFilePath',...
        'groundvehicle.stl');

    pause(0.01)
    hold off;
end
```



Plan Path for a Differential Drive Robot in Simulink

This example demonstrates how to execute an obstacle-free path between two locations on a given map in Simulink®. The path is generated using a probabilistic road map (PRM) planning algorithm (`mobileRobotPRM`). Control commands for navigating this path are generated using the **Pure Pursuit** controller block. A differential drive kinematic motion model simulates the robot motion based on those commands.

Load the Map and Simulink Model

Load the occupancy map, which defines the map limits and obstacles within the map. `exampleMaps.mat` contain multiple maps including `simpleMap`, which this example uses.

```
load exampleMaps.mat
```

Specify a start and end locaiton within the map.

```
startLoc = [5 5];
goalLoc = [20 20];
```

Model Overview

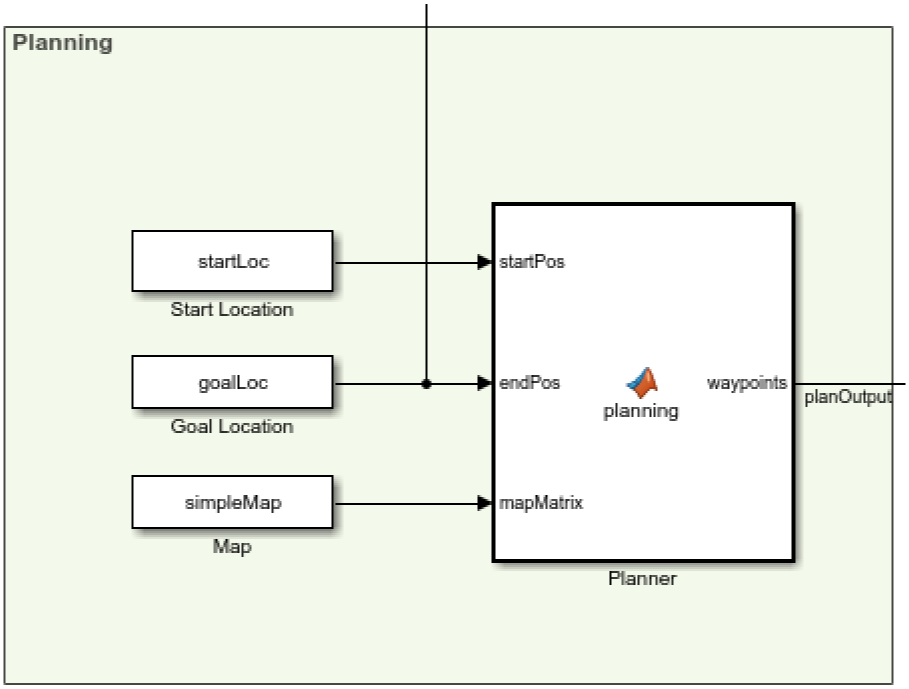
Open the Simulink model.

```
open_system('pathPlanningSimulinkModel.slx')
```

The model is composed of three primary parts:

- **Planning**
- **Control**
- **Plant Model**

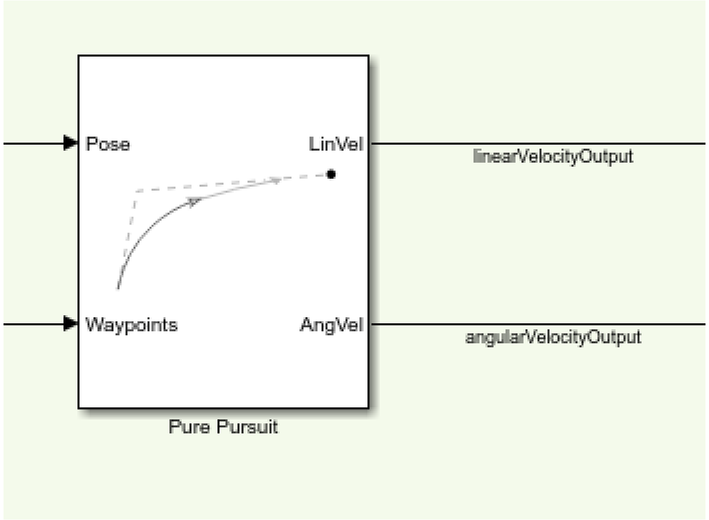
Planning



The **Planner** MATLAB® function block uses the `mobileRobotPRM` path planner and takes a start location, goal location, and map as inputs. The blocks outputs an array of waypoints that the robot follows. The planned waypoints are used downstream by the **Pure Pursuit** controller block.

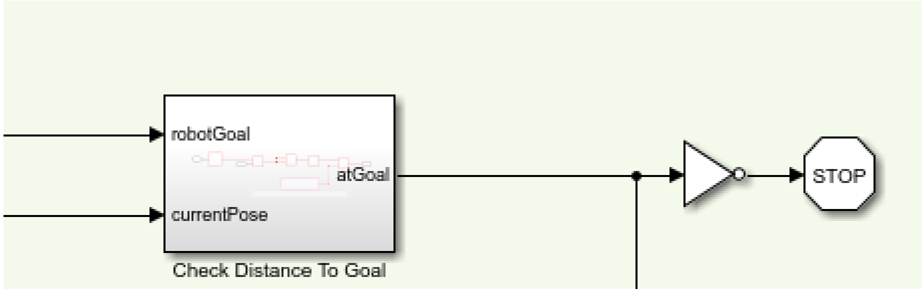
Control

Pure Pursuit



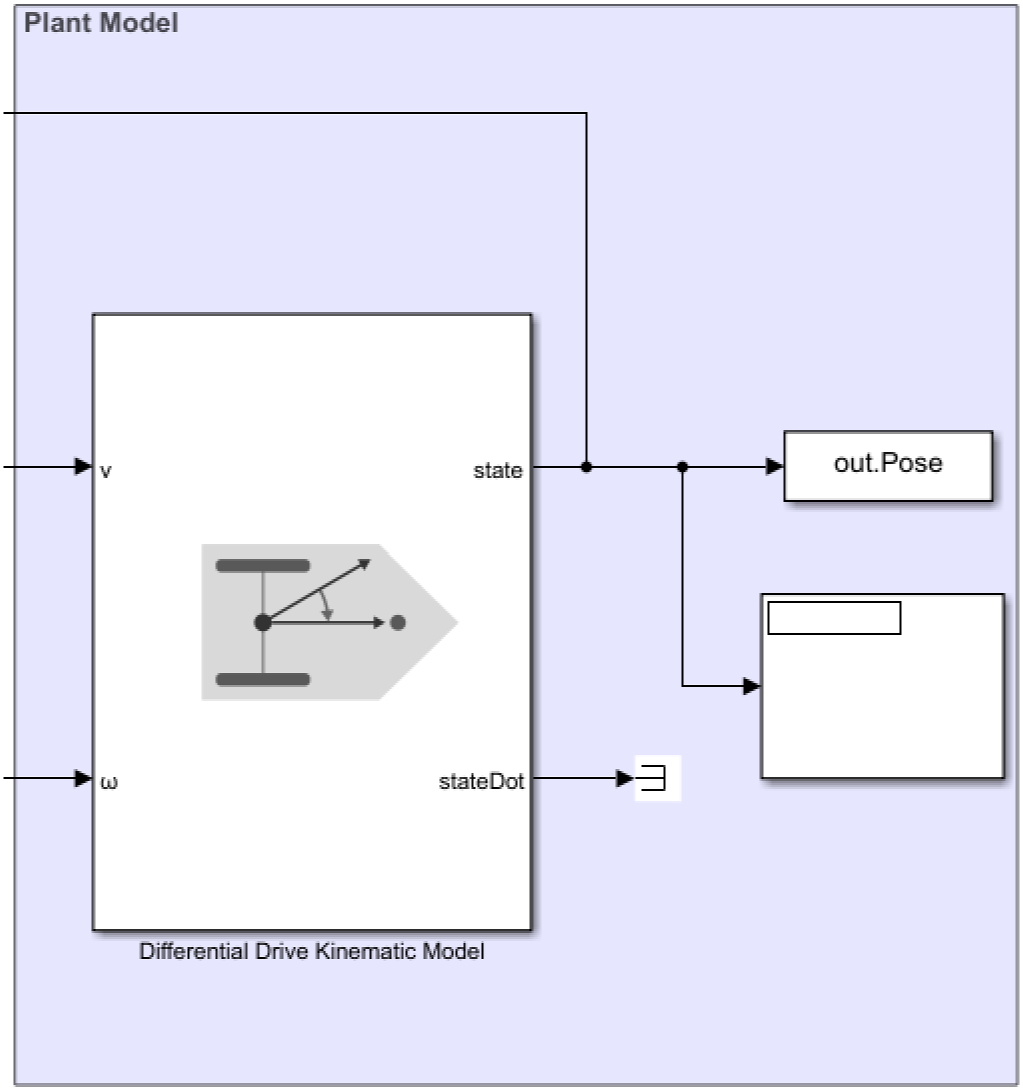
The **Pure Pursuit** controller block generates the linear velocity and angular velocity commands based on the waypoints and the current pose of the robot.

Check if Goal is Reached



The **Check Distance to Goal** subsystem calculates the current distance to the goal and if it is within a threshold, the simulation stops.

Plant Model



The **Differential Drive Kinematic Model** block creates a vehicle model to simulate simplified vehicle kinematics. The block takes linear and angular velocities as command inputs from the **Pure Pursuit** controller block, and outputs the current position and velocity states.

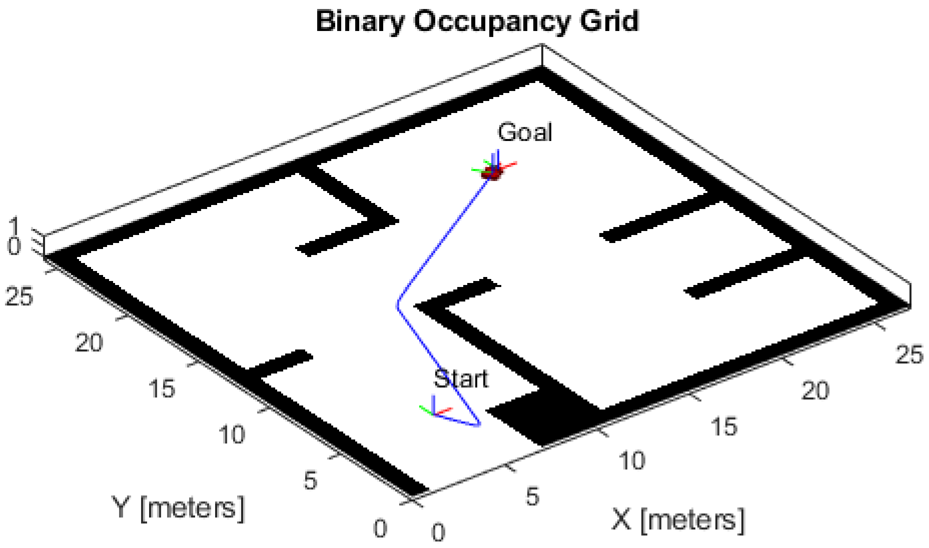
Run the Model

```
simulation = sim('pathPlanningSimulinkModel.slx');
```

Visualize The Motion of Robot

After simulating the model, visualize the robot driving the obstacle-free path in the map.

```
map = binaryOccupancyMap(simpleMap);
robotPose = simulation.Pose;
thetaIdx = 3;

% Translation
xyz = robotPose;
xyz(:, thetaIdx) = 0;

% Rotation in XYZ euler angles
theta = robotPose(:,thetaIdx);
thetaEuler = zeros(size(robotPose, 1), 3 * size(theta, 2));
thetaEuler(:, end) = theta;

% Plot the robot poses at every 10th step.
for k = 1:10:size(xyz, 1)
    show(map)
    hold on;

    % Plot the start location.
    plotTransforms([startLoc, 0], eul2quat([0, 0, 0]))
    text(startLoc(1), startLoc(2), 2, 'Start');

    % Plot the goal location.
    plotTransforms([goalLoc, 0], eul2quat([0, 0, 0]))
    text(goalLoc(1), goalLoc(2), 2, 'Goal');

    % Plot the xy-locations.
    plot(robotPose(:, 1), robotPose(:, 2), '-b')

    % Plot the robot pose as it traverses the path.
    quat = eul2quat(thetaEuler(k, :), 'xyz');
    plotTransforms(xyz(k,:), quat, 'MeshFilePath',...
        'groundvehicle.stl');
    light;
    drawnow;
    hold off;
end
```


Binary Occupancy Grid
Goal
Start
1
0
25
20
15
10
5
0
Y [meters]
0
5
10
15
20
25
X [meters]

Plan Path for a Bicycle Robot in Simulink

This example demonstrates how to execute an obstacle-free path between two locations on a given map in Simulink®. The path is generated using a probabilistic road map (PRM) planning algorithm (`mobileRobotPRM`). Control commands for navigating this path are generated using the **Pure Pursuit** controller block. A bicycle kinematic motion model simulates the robot motion based on those commands.

Load the Map and Simulink Model

Load map in MATLAB workspace

```
load exampleMaps.mat
```

Enter start and goal locations

```
startLoc = [5 5];
goalLoc = [12 3];
```

The imported maps are : `simpleMap`, `complexMap` and `ternaryMap`.

Open the Simulink Model

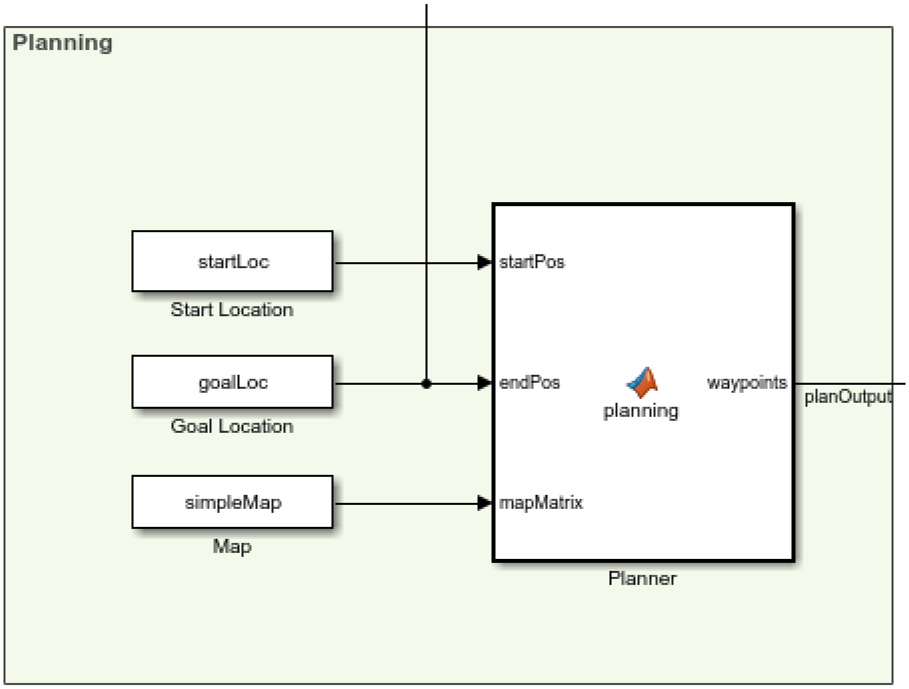
```
open_system('pathPlanningBicycleSimulinkModel.slx')
```

Model Overview

The model is composed of four primary operations :

- **Planning**
- **Control**
- **Plant Model**

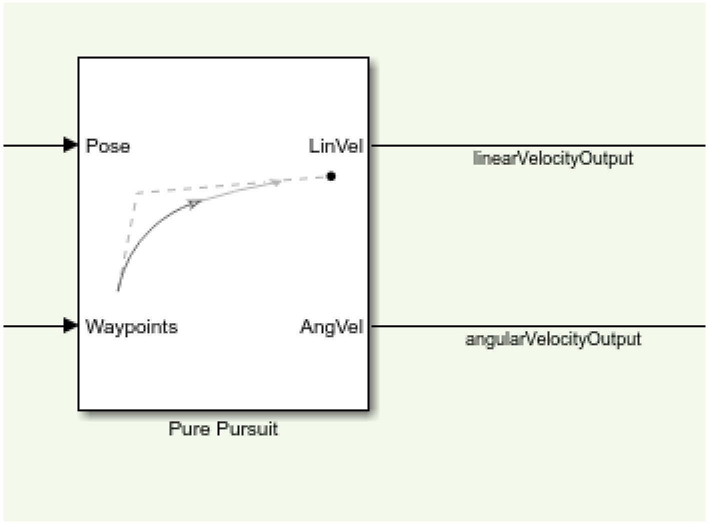
Planning



The **Planner** MATLAB® function block uses the `mobileRobotPRM` path planner and takes a start location, goal location, and map as inputs. The blocks outputs an array of waypoints that the robot follows. The planned waypoints are used downstream by the **Pure Pursuit** controller block.

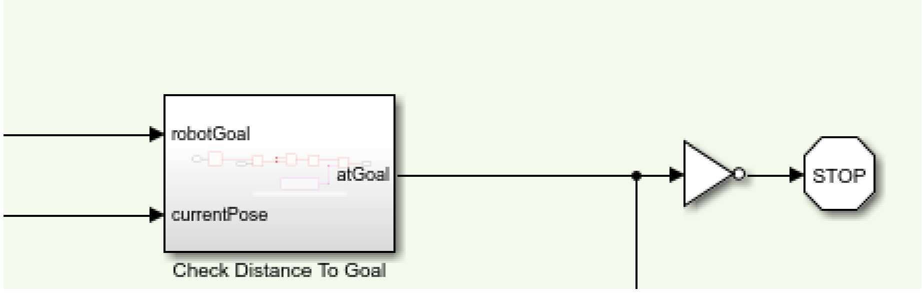
Control

Pure Pursuit



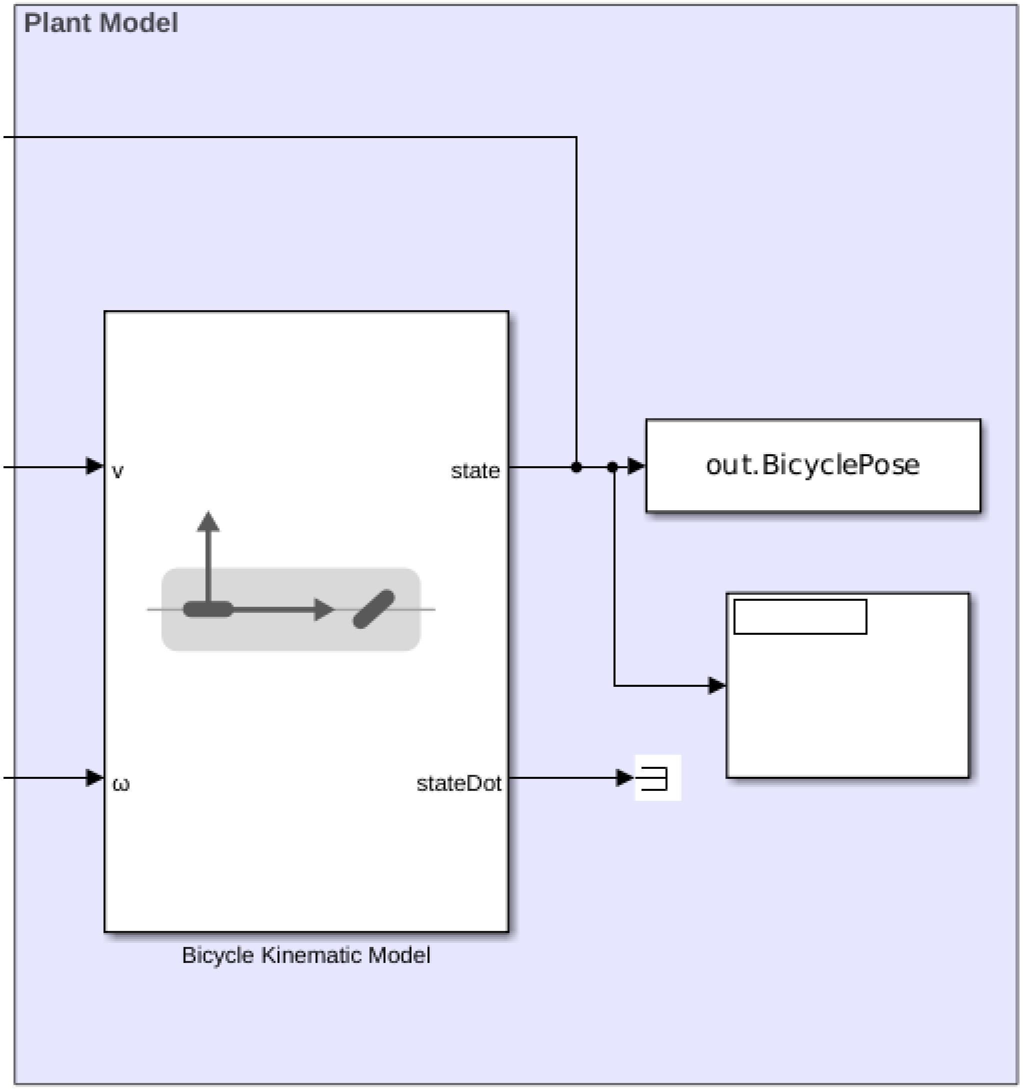
The **Pure Pursuit** controller block generates the linear velocity and angular velocity commands based on the waypoints and the current pose of the robot.

Check if Goal is Reached



The **Check Distance to Goal** subsystem calculates the current distance to the goal and if it is within a threshold, the simulation stops.

Plant Model



The **Bicycle Kinematic Model** block creates a vehicle model to simulate simplified vehicle kinematics. The block takes linear and angular velocities as command inputs from the **Pure Pursuit** controller block, and outputs the current position and velocity states.

Run the Model

To simulate the model

```
simulation = sim('pathPlanningBicycleSimulinkModel.slx');
```

Visualize The Motion of Robot

To see the poses :

```
map = binaryOccupancyMap(simpleMap)
```

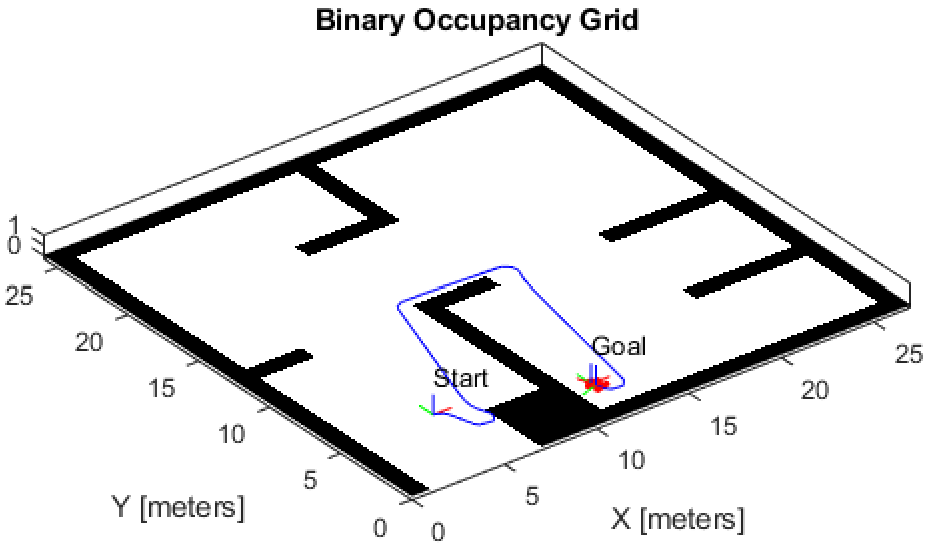
```
map = 
  binaryOccupancyMap with properties:

   mapLayer Properties
              LayerName: 'binaryLayer'
               DataType: 'logical'
           DefaultValue: 0
    GridLocationInWorld: [0 0]
      GridOriginInLocal: [0 0]
     LocalOriginInWorld: [0 0]
             Resolution: 1
               GridSize: [26 27]
           XLocalLimits: [0 27]
           YLocalLimits: [0 26]
           XWorldLimits: [0 27]
           YWorldLimits: [0 26]
```

```
robotPose = simulation.BicyclePose
```

```
robotPose = 362×3

    5.0000    5.0000         0
    5.0001    5.0000   -0.0002
    5.0007    5.0000   -0.0012
    5.0036    5.0000   -0.0062
    5.0181    4.9997   -0.0313
    5.0902    4.9929   -0.1569
    5.4081    4.8311   -0.7849
    5.5189    4.6758   -1.1170
    5.5366    4.6356   -1.1930
    5.5512    4.5942   -1.2684
      ⋮
```

```
numRobots = size(robotPose, 2) / 3;
thetaIdx = 3;

% Translation
xyz = robotPose;
xyz(:, thetaIdx) = 0;

% Rotation in XYZ euler angles
```

```
theta = robotPose(:,thetaIdx);
thetaEuler = zeros(size(robotPose, 1), 3 * size(theta, 2));
thetaEuler(:, end) = theta;

for k = 1:size(xyz, 1)
    show(map)
    hold on;

    % Plot Start Location
    plotTransforms([startLoc, 0], eul2quat([0, 0, 0]))
    text(startLoc(1), startLoc(2), 2, 'Start');

    % Plot Goal Location
    plotTransforms([goalLoc, 0], eul2quat([0, 0, 0]))
    text(goalLoc(1), goalLoc(2), 2, 'Goal');

    % Plot Robot's XY locations
    plot(robotPose(:, 1), robotPose(:, 2), '-b')

    % Plot Robot's pose as it traverses the path
    quat = eul2quat(thetaEuler(k, :), 'xyz');
    plotTransforms(xyz(k,:), quat, 'MeshFilePath',...
        'groundvehicle.stl');

    pause(0.01)
    hold off;
end
```



Plot Ackermann Drive Vehicle in Simulink

This example shows how to plot the position of an Ackermann Kinematic Model block and change it's vehicle velocity and steering angular velocity in real-time.

Open the Simulink model.

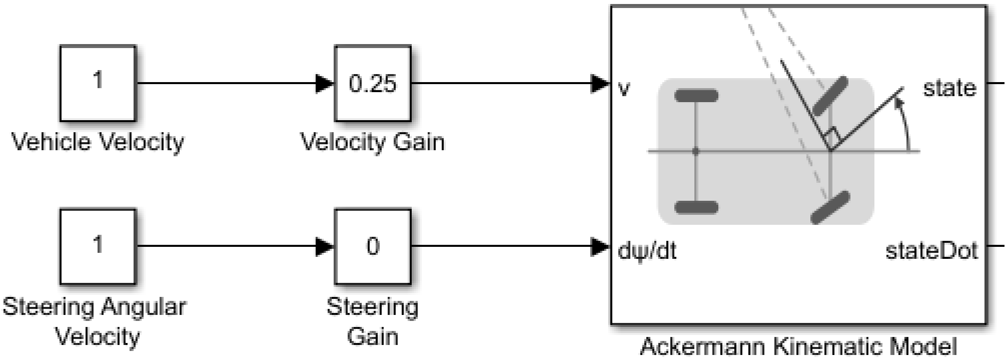
```
open_system("plotAckermannDriveSimulinkModel.slx");
```

Ackermann Kinematic Block

The Ackermann Kinematic Model block parameters are the default values, but it is important to note two parameters for this example, the **Vehicle speed range** and **Maximum steering angle**. Both parameters limit the motion of the vehicle. The lower bound of the **Vehicle speed range** parameter is set to `-inf` and the upper bound is set to `inf`, so the vehicle velocity can be any real value you set. The **Maximum steering angle** is set to `pi/4`, so there's a max turning radius that the vehicle can achieve.

Vehicle and Steering Velocity

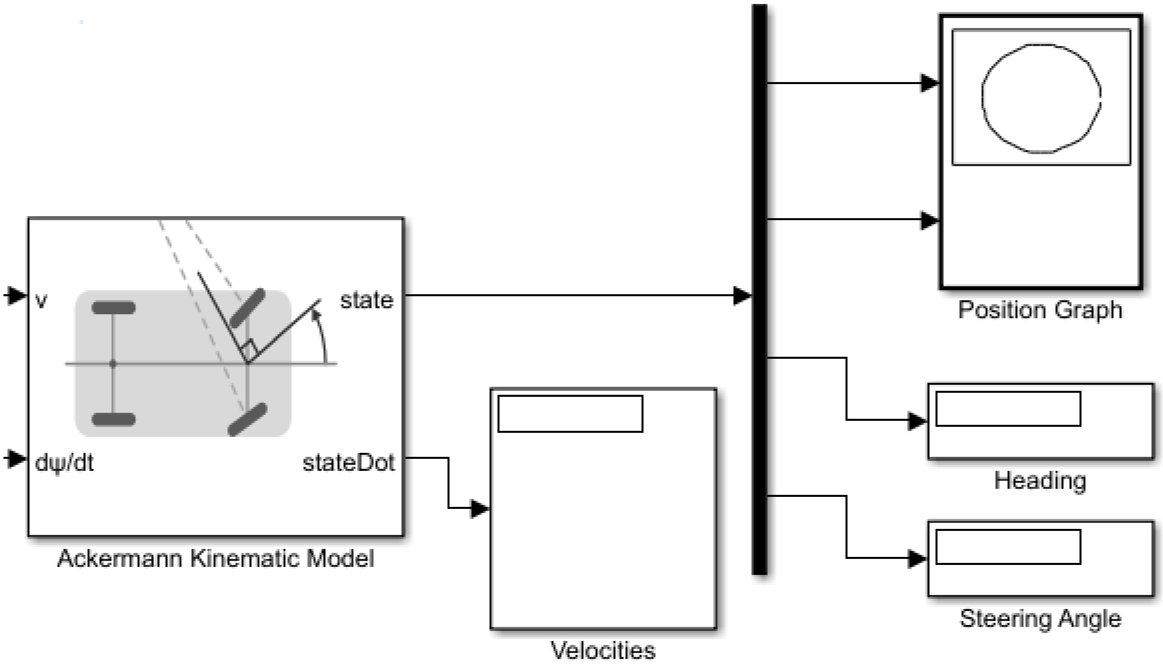
The Ackermann Kinematic Model block takes two inputs, vehicle velocity and steering angular velocity. This model uses **Slider Gain** blocks to change the inputs.



These values can be any real values within the parameter constraints set in the Ackermann Kinematic Model block.

Graphing the Output

Using a demux block, the x and y signals of the `state` output connect to a **XY Graph** block. The signals of `stateDot` and the other two signals of `state` connect to **Display** blocks.



Run the Model

- Set the model run time to `inf`.
- Click **Play** to run the model. The graph will appear and you can see the path of the vehicle.
- Open the **Slider Gain** blocks and adjust the values of the blocks to see their affects on the path of the vehicle.
- Adjust the graph limits as needed.
- Observe the **Steering Angle** display as you adjust the value of the **Steering Gain**.

Follow Joint Space Trajectory in Simulink

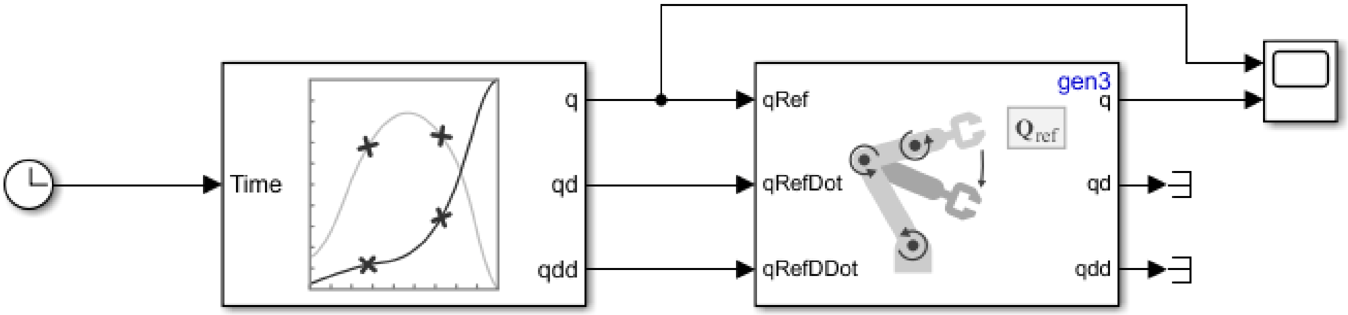
This example shows how to use a **Joint Space Motion Model** block to follow a trajectory in Simulink.

This example uses the Kinova Gen3 manipulator robot to follow the trajectories. Load the Gen3 manipulator using `loadrobot` and save the `RigidBodyTree` output as `gen3`. Open the Simulink model.

```
[gen3,metadata] = loadrobot("kinovaGen3");
```

Open the simulink model.

```
open_system("followJointSpaceTrajectoryModel.slx");
```

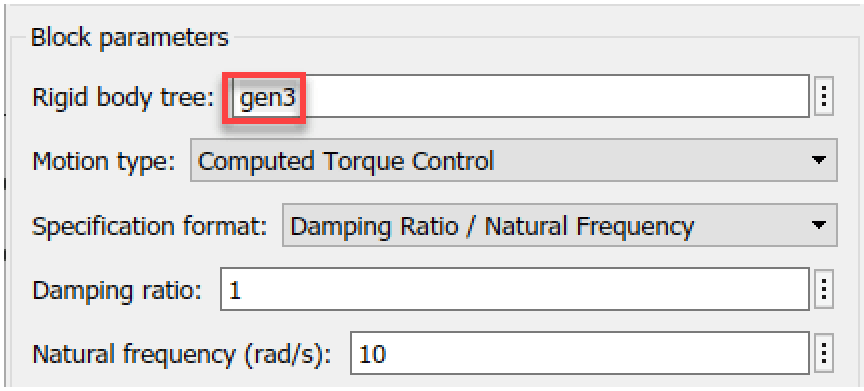


Plan Trajectory

The **Polynomial Trajectory** block generates a trajectory from a set of waypoints specified in the **Waypoints** parameter in joint space. This example uses five time points, specified row vector and also the Kinova Gen3 has seven degrees of freedom, so the waypoints matrix must be a 7-by-5 size matrix. The block is set up to generate a new set of waypoints every simulation.

Motion Model

The Joint Space Motion Model uses a RigidBodyTree, `gen3`, to calculate the joint positions to reach the random trajectory generated by the **Polynomial Trajectory** block. Leave the other block parameters as default.



Block parameters

Rigid body tree: gen3

Motion type: Computed Torque Control

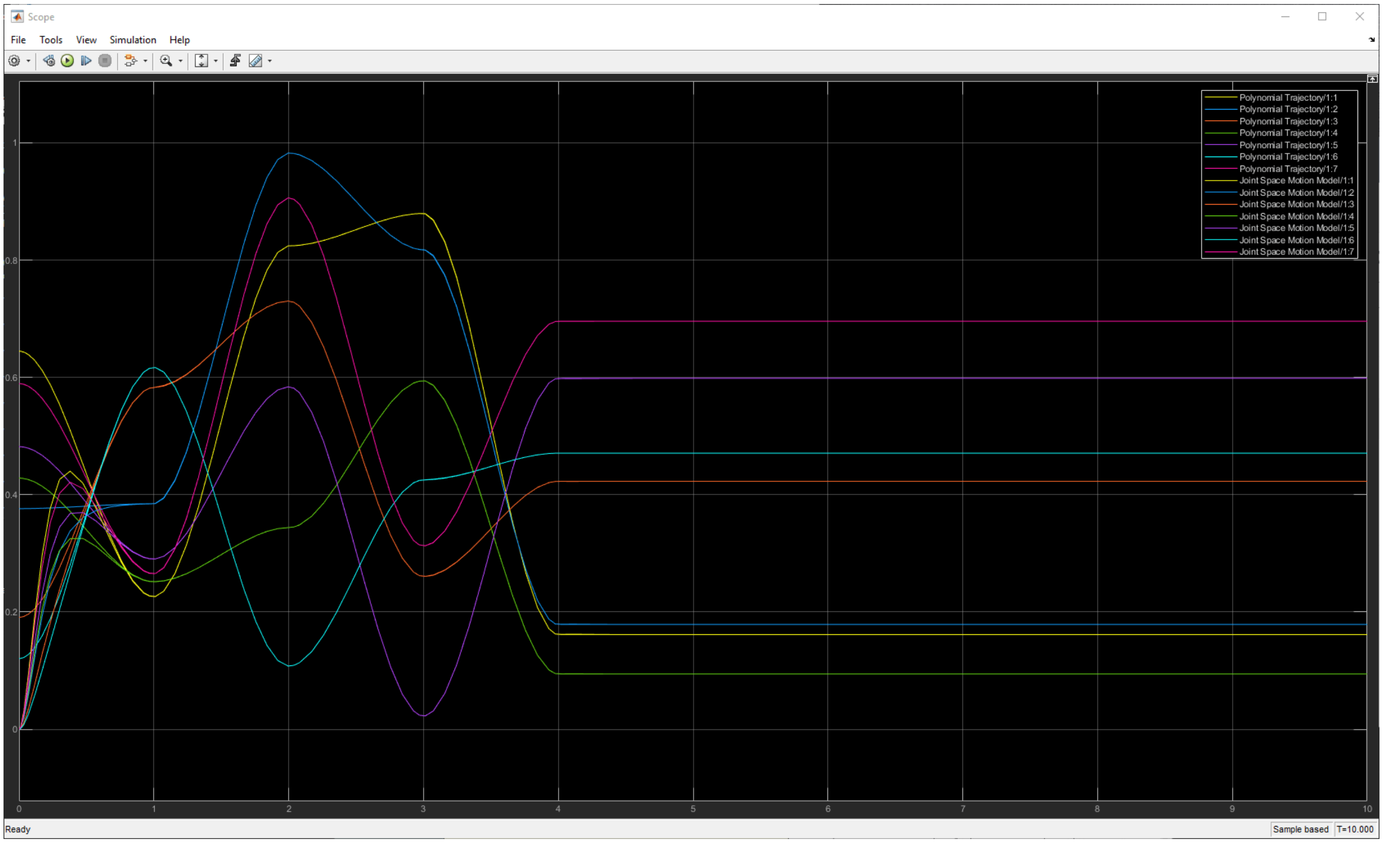
Specification format: Damping Ratio / Natural Frequency

Damping ratio: 1

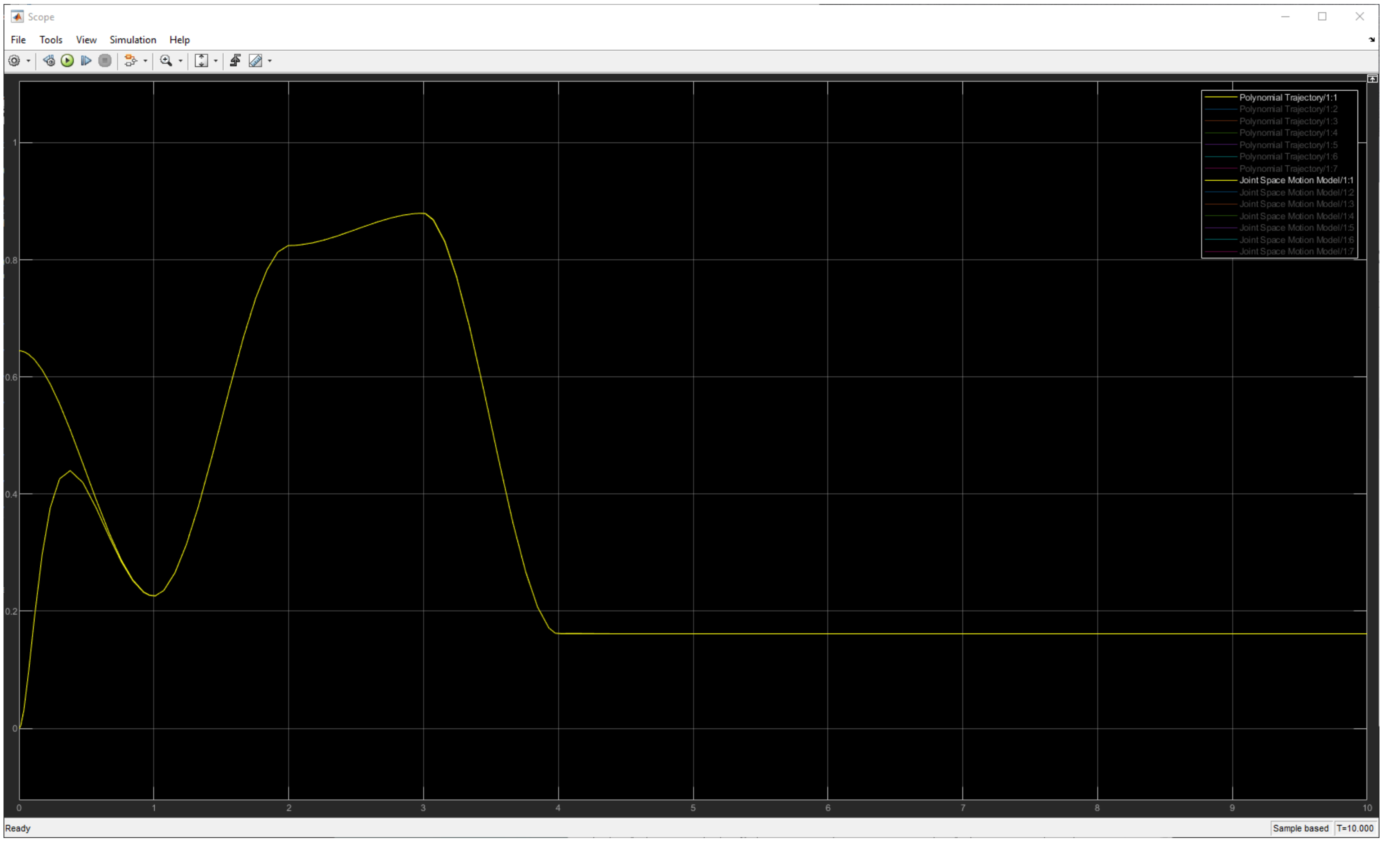
Natural frequency (rad/s): 10

Visualize Results

The joint target positions and the calculated joint values from the **Joint Space Motion Model** connect to a **Scope** block. Using the legend, you can select a smaller set of signals to compare with better clarity.


Scope
File
Tools
View
Simulation
Help
Polynomial Trajectory/1:1
Polynomial Trajectory/1:2
Polynomial Trajectory/1:3
Polynomial Trajectory/1:4
Polynomial Trajectory/1:5
Polynomial Trajectory/1:6
Polynomial Trajectory/1:7
Joint Space Motion Model/1:1
Joint Space Motion Model/1:2
Joint Space Motion Model/1:3
Joint Space Motion Model/1:4
Joint Space Motion Model/1:5
Joint Space Motion Model/1:6
Joint Space Motion Model/1:7
1
0.8
0.6
0.4
0.2
0
0
1
2
3
4
5
6
7
8
9
10
Ready
Sample based
T=10.000

Observe that the signals for the first joint start separated, and overlap when time is equal to 1s. So from the initial configuration, the first joint was able to follow the trajectory.


Scope
File
Tools
View
Simulation
Help
Polynomial Trajectory/1:1
Polynomial Trajectory/1:2
Polynomial Trajectory/1:3
Polynomial Trajectory/1:4
Polynomial Trajectory/1:5
Polynomial Trajectory/1:6
Polynomial Trajectory/1:7
Joint Space Motion Model/1:1
Joint Space Motion Model/1:2
Joint Space Motion Model/1:3
Joint Space Motion Model/1:4
Joint Space Motion Model/1:5
Joint Space Motion Model/1:6
Joint Space Motion Model/1:7
1
0.8
0.6
0.4
0.2
0
0
1
2
3
4
5
6
7
8
9
10
Ready
Sample based
T=10.000

Follow Task Space Trajectory in Simulink

This example shows how to use a Task Space Motion Model to follow a task space trajectory.

Load Robot and Simulink Model

This example uses a Kinova Gen3 manipulator robot. Load the model using `loadrobot`.

```
[gen3,metadata] = loadrobot("kinovaGen3",'DataFormat','column');
initialConfig = homeConfiguration(gen3);
targetPosition = trvec2tform([0.6 -.1 0.5])
```

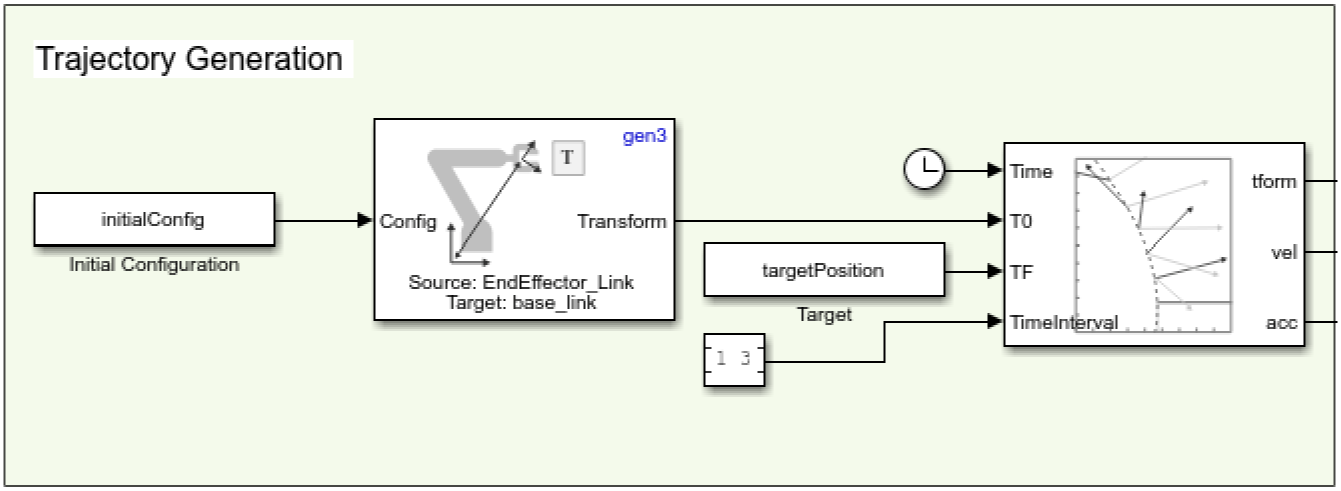
```
targetPosition = 4×4

    1.0000         0         0    0.6000
         0    1.0000         0   -0.1000
         0         0    1.0000    0.5000
         0         0         0    1.0000
```

Open the Simulink model.

```
open_system("followTaskSpaceTrajectoryModel.slx")
```

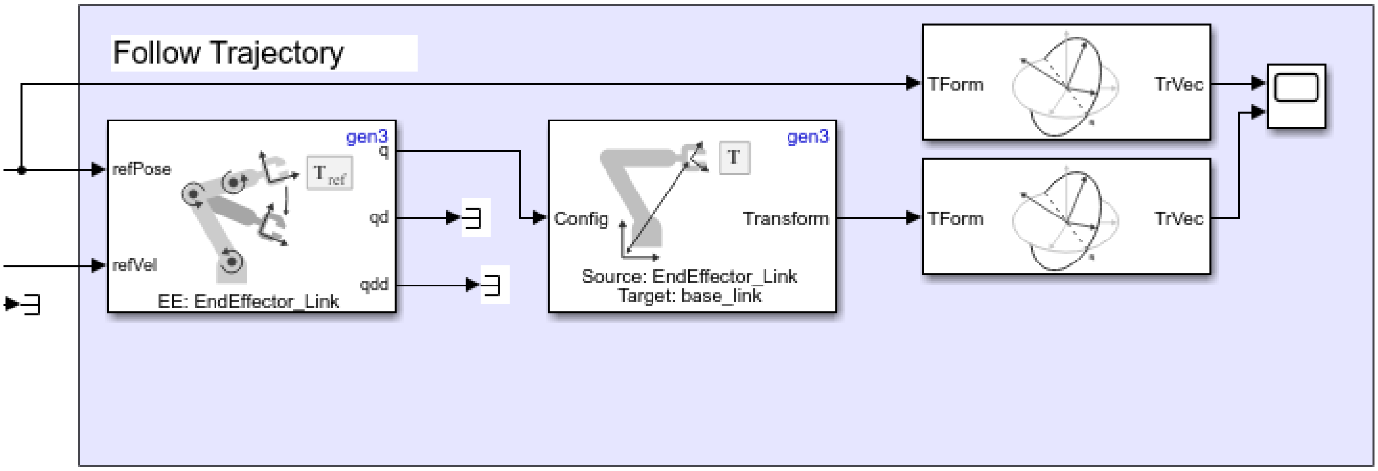
Trajectory Generation

The **Transform Trajectory** block creates a trajectory between the initial homogeneous transform matrix of the end effector of the Gen3, and the target position over a 3 second time interval.



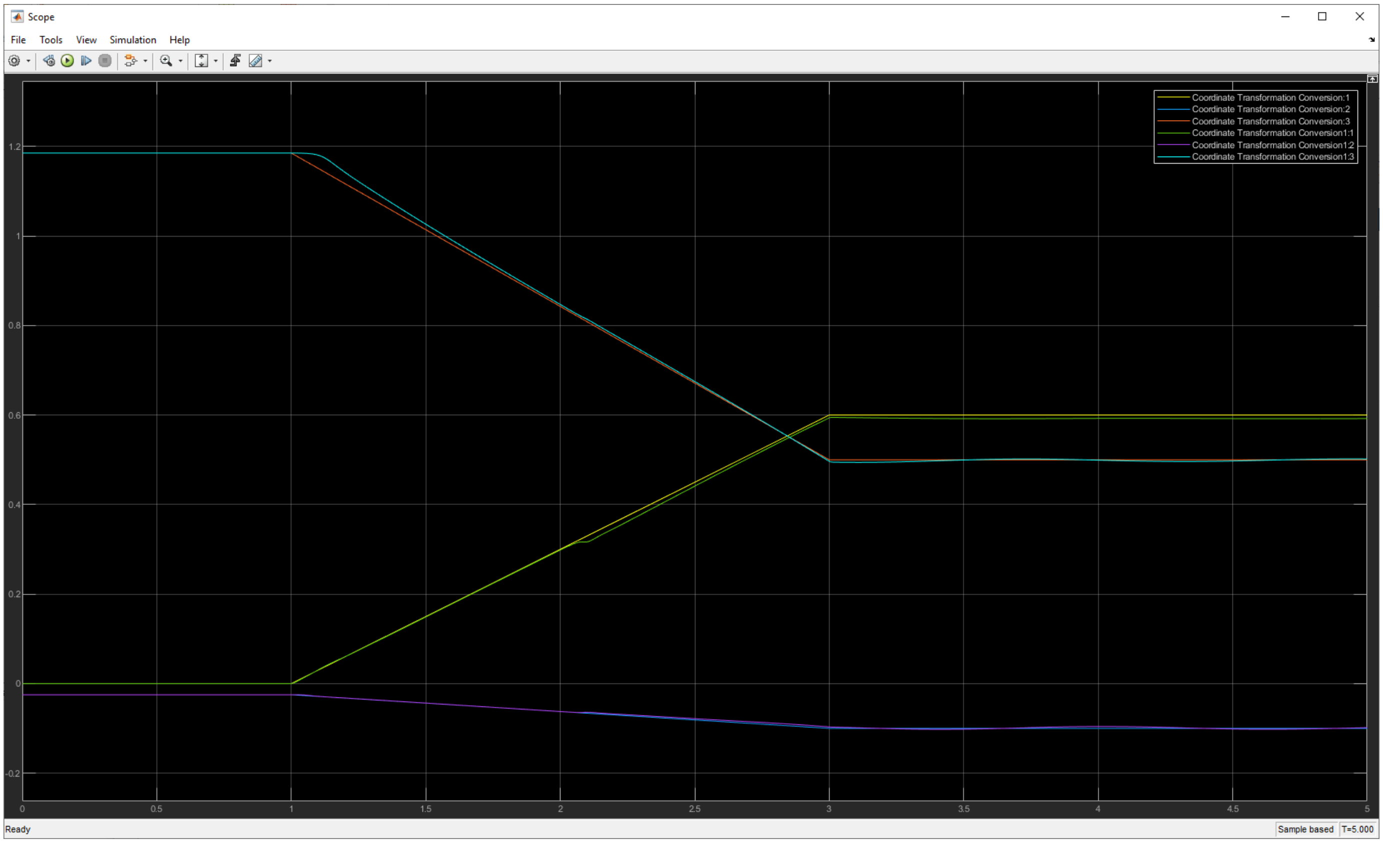
Follow Trajectory

The Joint Space Motion Model uses a RigidBodyTree, `gen3`, to calculate the joint positions to follow the trajectory. The joint positions are converted to homogeneous transform matrices and then the converted to a translation vector so that it is easier to visualize.



Visualize Results

The joint target positions and the calculated joint values from the **Task Space Motion Model** connect to a **Scope** block. Using the legend, you can select a smaller set of signals to compare with better clarity. Observe that the x, y, and z positions of the end effector match closely with the x, y, and z positions of the trajectory to the target position.


Scope
File
Tools
View
Simulation
Help
Coordinate Transformation Conversion:1
Coordinate Transformation Conversion:2
Coordinate Transformation Conversion:3
Coordinate Transformation Conversion1:1
Coordinate Transformation Conversion1:2
Coordinate Transformation Conversion1:3
1.2
1
0.8
0.6
0.4
0.2
0
-0.2
0
0.5
1
1.5
2
2.5
3
3.5
4
4.5
5
Ready
Sample based
T=5.000