Robotics System Toolbox™

User's Guide

MATLAB® & SIMULINK®

R2020**a**

MathWorks®

How to Contact MathWorks

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	Rereleased for Version 3.1 (R2020a)

Contents

1 Robotics System Toolbox Topics

Robotics System Toolbox Topics

Rigid Body Tree Robot Model

In this section...
"Rigid Body Tree Components" on page 1-2
"Robot Configurations" on page 1-4

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`.

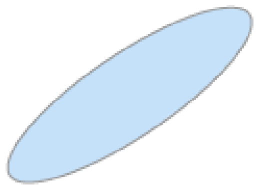
Rigid Body Tree Components

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.



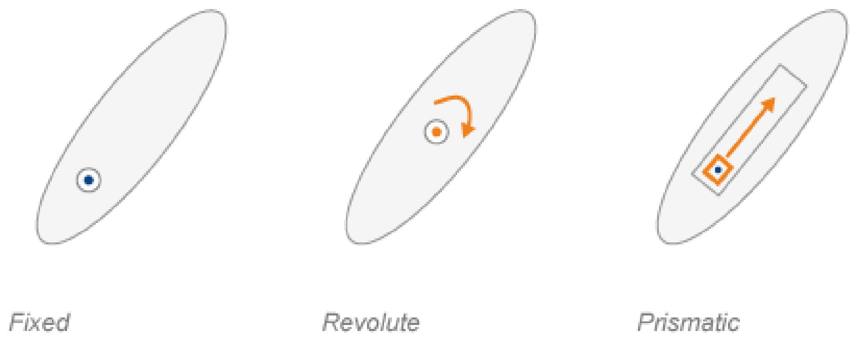
When added to a rigid body tree with multiple bodies, rigid bodies have parent or children bodies associated with them (`Parent` or `Children` properties). The parent is the body that this rigid body is attached to, which can be the robot base. The children are all the bodies attached to this body downstream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

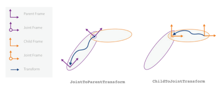


These joints allow the following motion, depending on their type:

- `'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 1-4.

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.

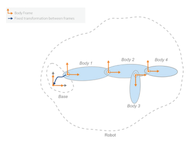
Use `homeConfiguration` to get the `HomePosition` property of each joint and create the home configuration.

Robot configurations are given as an array of structures.

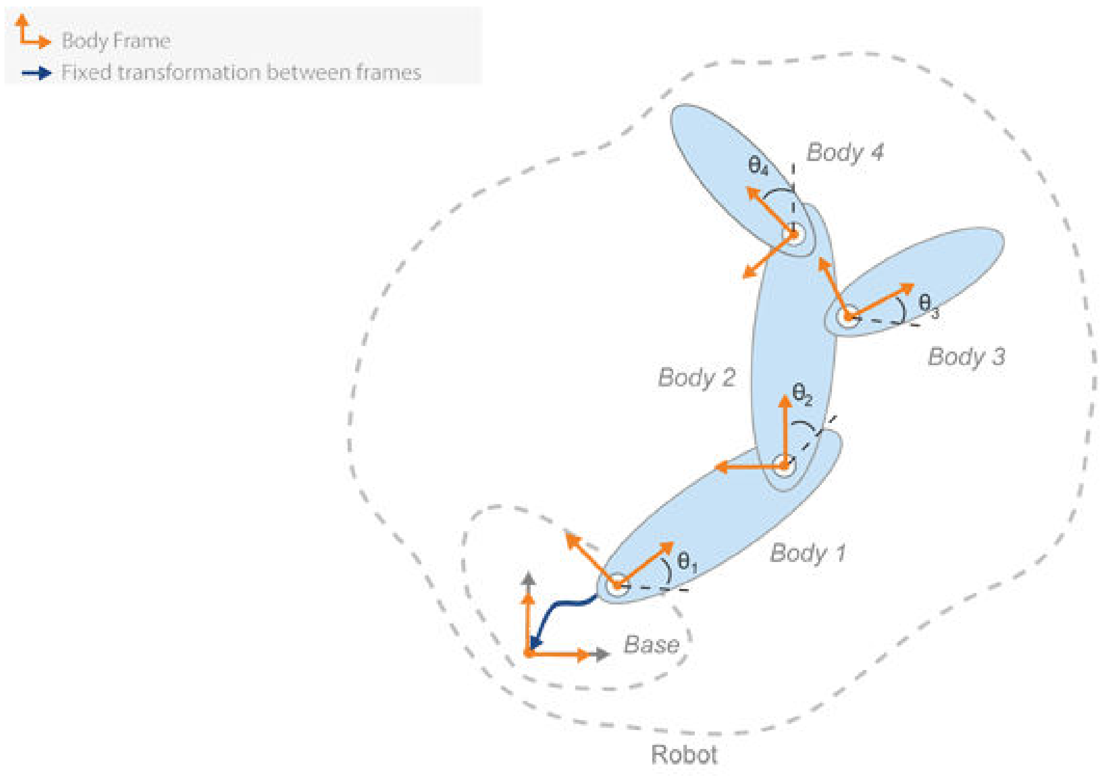
```
config = homeConfiguration(robot)

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.

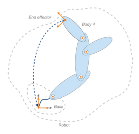
```
config(1)

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

`inverseKinematics` | `rigidBodyTree`

Related Examples

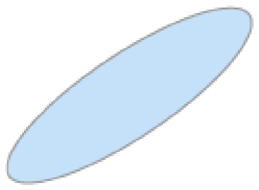
- “Build a Robot Step by Step” on page 1-6
- “Inverse Kinematics Algorithms” on page 1-10

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.

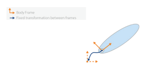
1 Create a rigid body object.

```
body1 = rigidBody('body1');
```



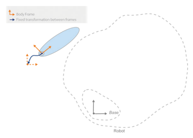
2 Create a joint and assign it to the rigid body. Define the home position property of the joint, `HomePosition`. Set the joint-to-parent transform using a homogeneous transformation, `tform`. Use the `trvec2tform` function to convert from a translation vector to a homogenous transformation.`ChildToJointTransform` 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;
```



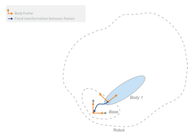
3 Create a rigid body tree. This tree is initialized with a base coordinate frame to attach bodies to.

```
robot = rigidBodyTree;
```



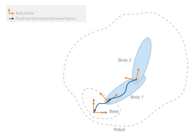
4 Add the first body to the tree. Specify that you are attaching it to the base of the tree. The fixed transform defined previously is from the base (parent) to the first body.

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



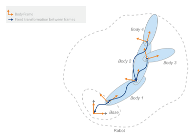
5 Create a second body. Define properties of this body and attach it to the first rigid body. Define the transformation relative to the previous body frame.

```
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
```



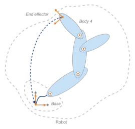
7 If you have a specific end effector that you care about for control, define it as a rigid body with a fixed joint. For this robot, add an end effector to `body4` so that you can get transformations for it.

```
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')
```

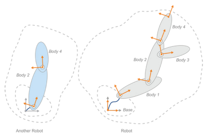
```
config = 

  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.

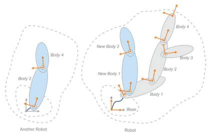
9 You can create a subtree from your existing robot or other robot models by using `subtree`. Specify the body name to use as the base for the new subtree. You can modify this subtree by adding, changing, or removing bodies.

```
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(I
    ---          ---------            ----------            ----------           --------------
      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

`inverseKinematics` | `rigidBodyTree`

Related Examples

- “Rigid Body Tree Robot Model” on page 1-2

Inverse Kinematics Algorithms

In this section...
"Choose an Algorithm" on page 1-10
"Solver Parameters" on page 1-11
"Solution Information" on page 1-12
"References" on page 1-12

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 1-10 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 1-10 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 1-11 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 1-12.
- `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

`generalizedInverseKinematics` | `inverseKinematics` | `rigidBodyTree`

Related Examples

- “2-D Path Tracing With Inverse Kinematics” on page 1-14
- “Control PR2 Arm Movements Using ROS Actions and Inverse Kinematics”
- “Rigid Body Tree Robot Model” on page 1-2

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.

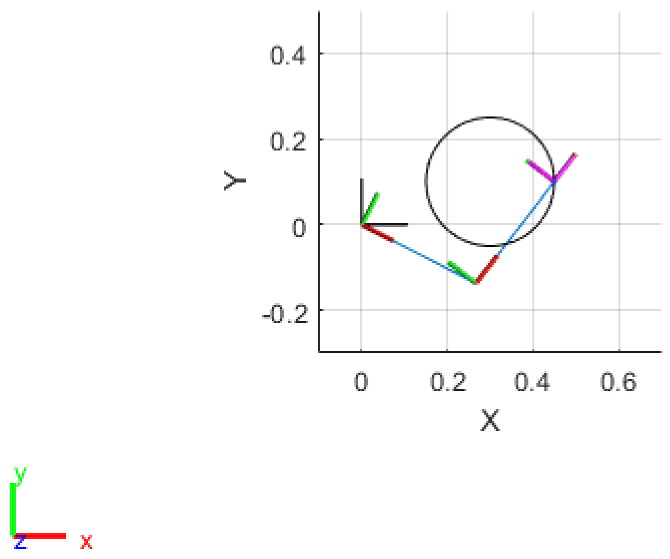
```
qInitial = q0; % Use home configuration as the initial guess
for i = 1:count
    % Solve for the configuration satisfying the desired end effector
    % position
    point = points(i,:);
    qSol = ik(endEffector,trvec2tform(point),weights,qInitial);
    % Store the configuration
    qs(i,:) = qSol;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

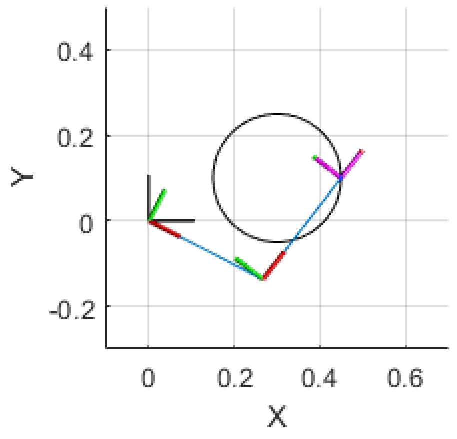
Show the robot in the first configuration of the trajectory. Adjust the plot to show the 2-D plane that circle is drawn on. Plot the desired trajectory.

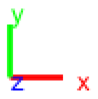
```
figure
show(robot,qs(1,:)');
view(2)
ax = gca;
ax.Projection = 'orthographic';
hold on
plot(points(:,1),points(:,2),'k')
axis([-0.1 0.7 -0.3 0.5])
```



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

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

Related Examples

- “Control PR2 Arm Movements Using ROS Actions and Inverse Kinematics”
- “Inverse Kinematics Algorithms” on page 1-10

Solve Inverse Kinematics for a Four-Bar Linkage

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

Initialize the four-bar linkage rigid body tree model.

```
robot = 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.

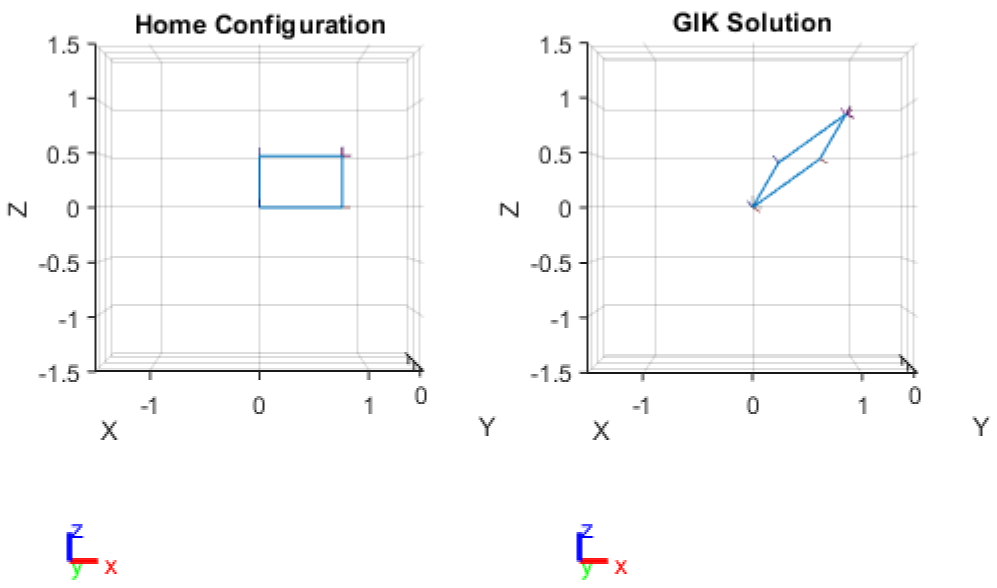
```
gik = generalizedInverseKinematics('RigidBodyTree',robot);
gik.ConstraintInputs = {'position',...  % Position constraint for closed-loop mechanism
                        'position',...  % Position constraint for end-effector
                        'joint'};       % Joint limits
gik.SolverParameters.AllowRandomRestart = 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
`constraintJointBounds` | `constraintPoseTarget` | `generalizedInverseKinematics` | `inverseKinematics` | `rigidBodyTree`

Related Examples

- "Rigid Body Tree Robot Model" on page 1-2
- "Plan a Reaching Trajectory With Multiple Kinematic Constraints"
- "Control LBR Manipulator Motion Through Joint Torque Commands"

Robot Dynamics

In this section...

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

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

Dynamics Properties

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

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

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

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

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

Dynamics Functions

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

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

See Also

`generalizedInverseKinematics` | `inverseKinematics` | `rigidBodyTree`

Related Examples

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

Occupancy Grids

In this section...

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.

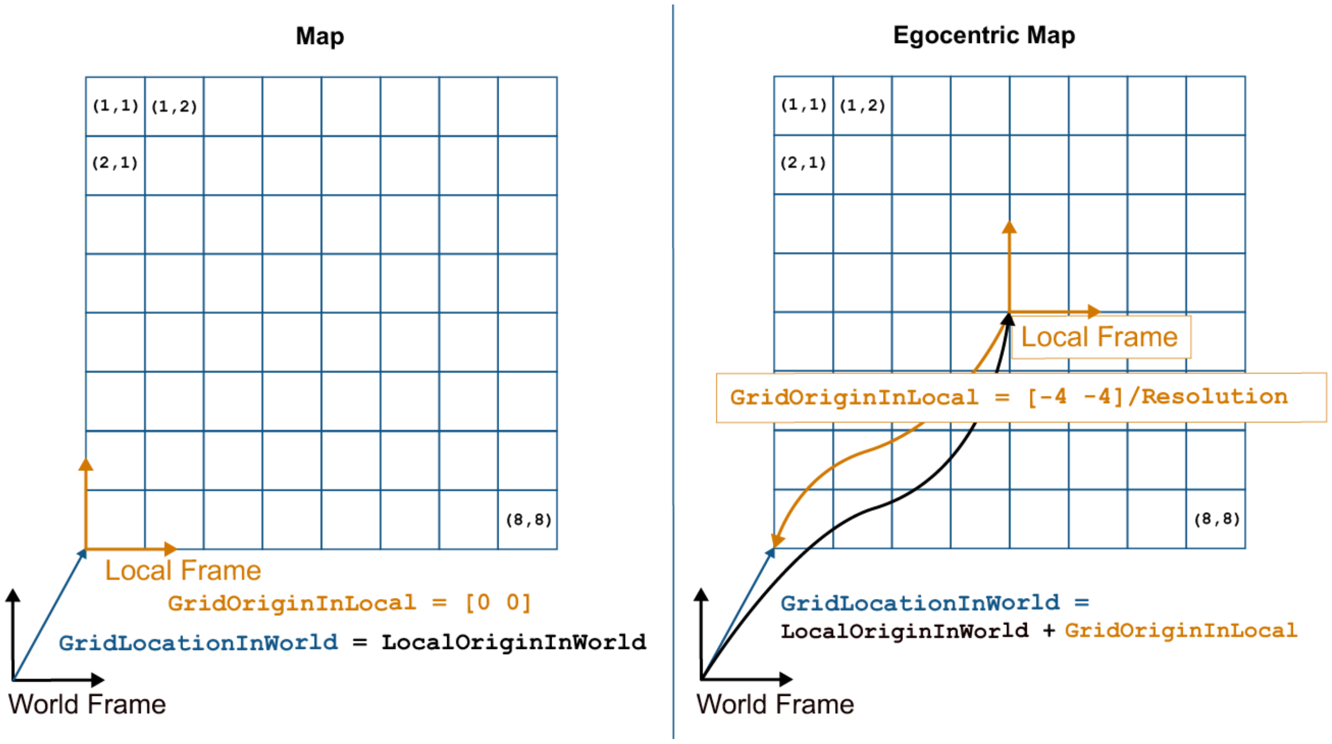
World, Grid, and Local Coordinates

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

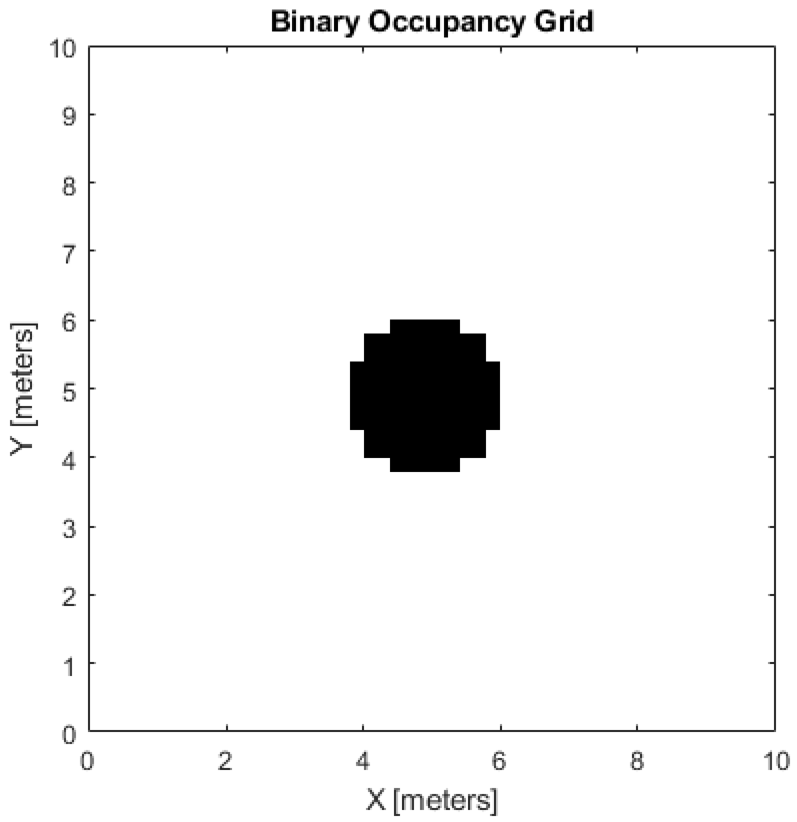
This example shows how to create the map, set the obstacle locations and inflate it by a radius of 1m. Extra plots on the figure help illustrate the inflation and shifting due to conversion to grid locations.

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.

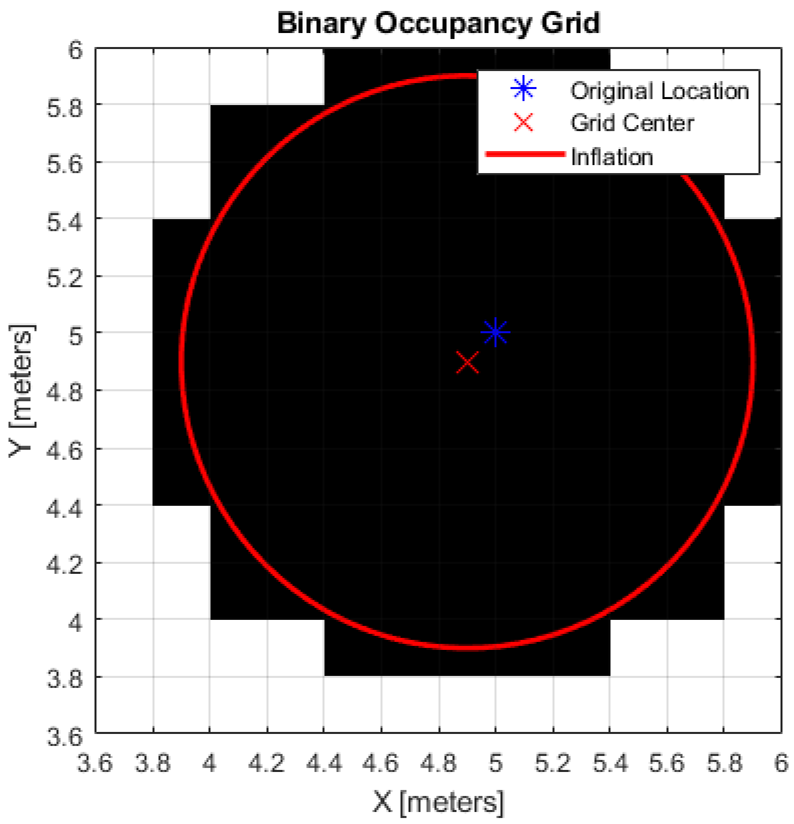
```
inflate(map,1);
show(map)
```



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 1-31 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.

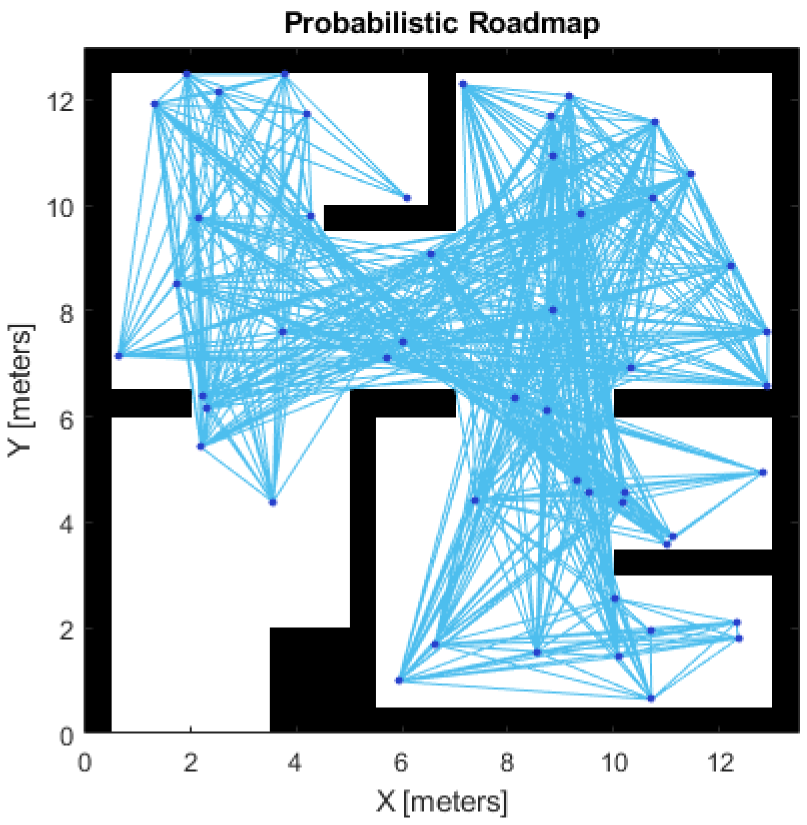
Increasing the number of nodes can increase the efficiency of the path by giving more feasible paths. However, the increased complexity increases computation time. To get good coverage of the map, you might need a large number of nodes. Due to the random placement of nodes, some areas of the map may 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);
```

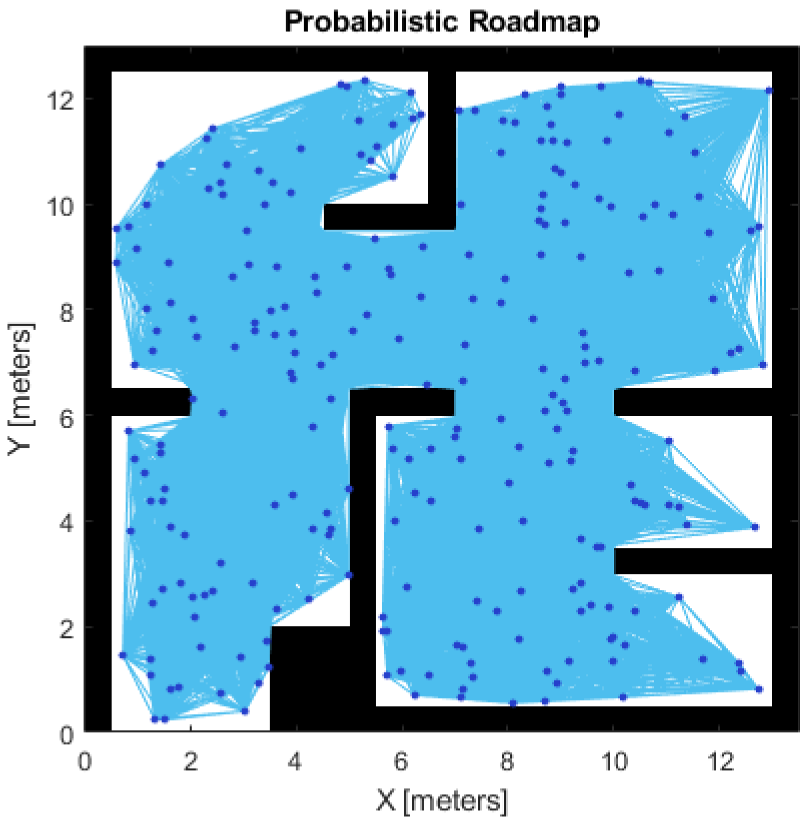
Create a simple roadmap with 50 nodes.

```
prmSimple = mobileRobotPRM(map,50);
show(prmSimple)
```



Create a dense roadmap with 250 nodes.

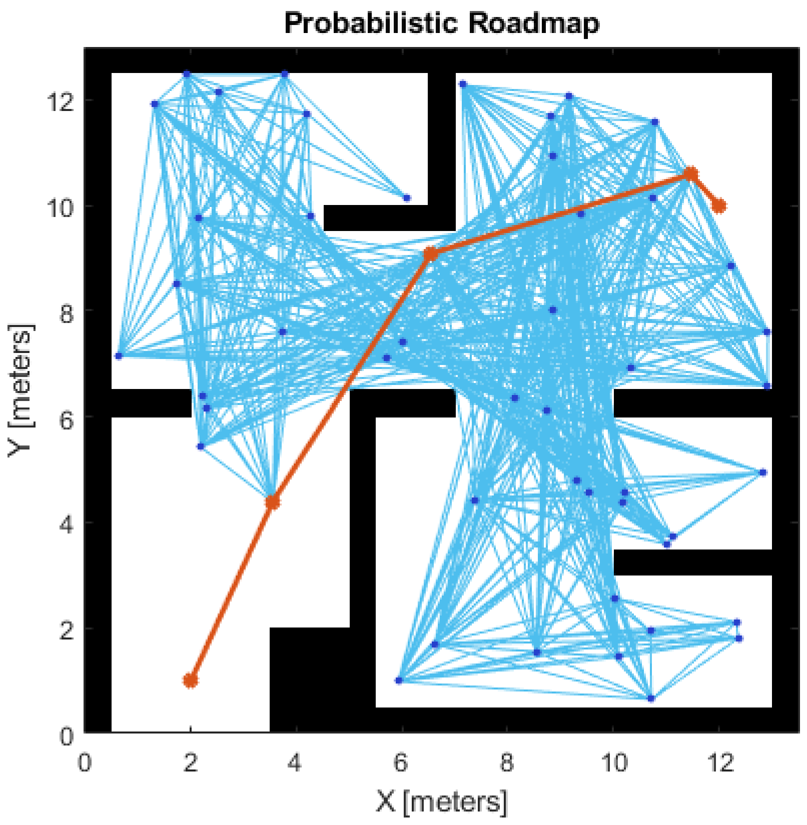
```
prmComplex = mobileRobotPRM(map,250);
show(prmComplex)
```



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.

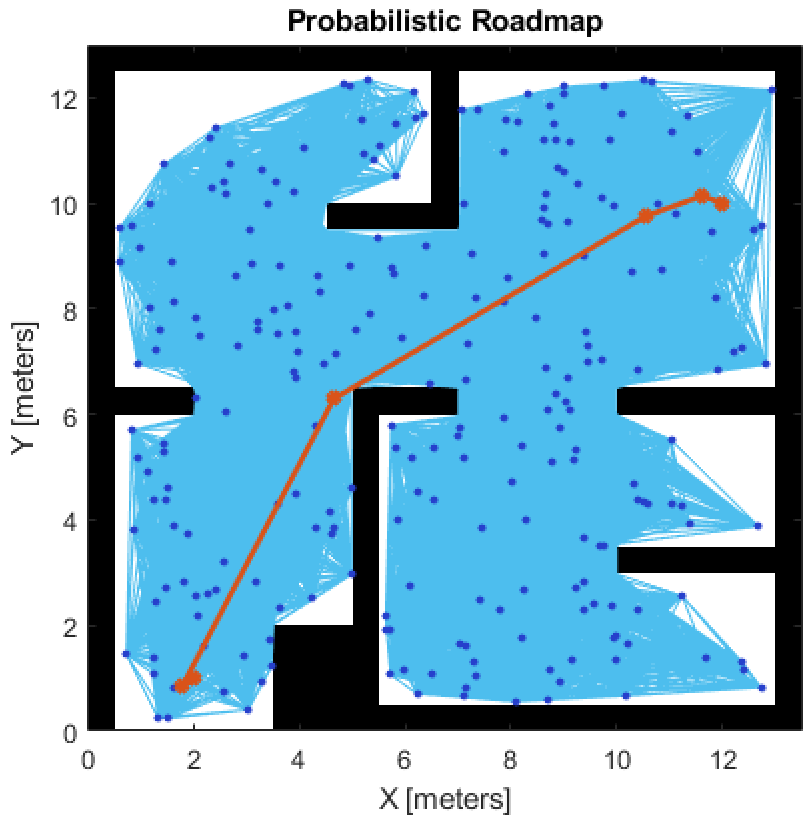
Calculate a simple path.

```
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

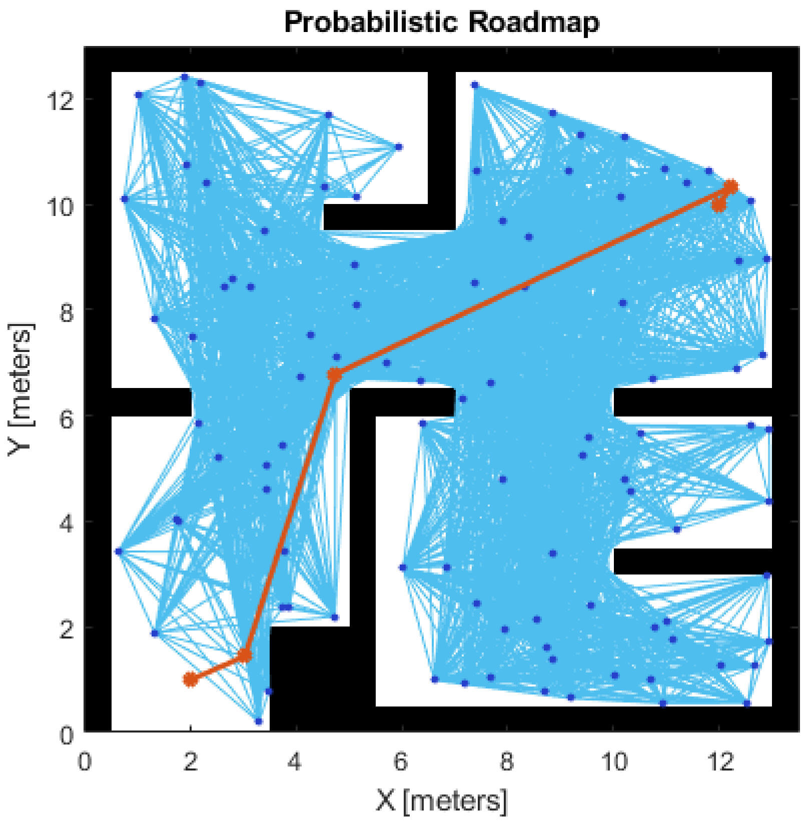
Use the `ConnectionDistance` property on the `PRM` object to tune the algorithm. `ConnectionDistance` is an upper threshold for points that are connected in the roadmap. Each node is connected to all nodes within this connection distance that do not have obstacles between them. By lowering the connection 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);
```

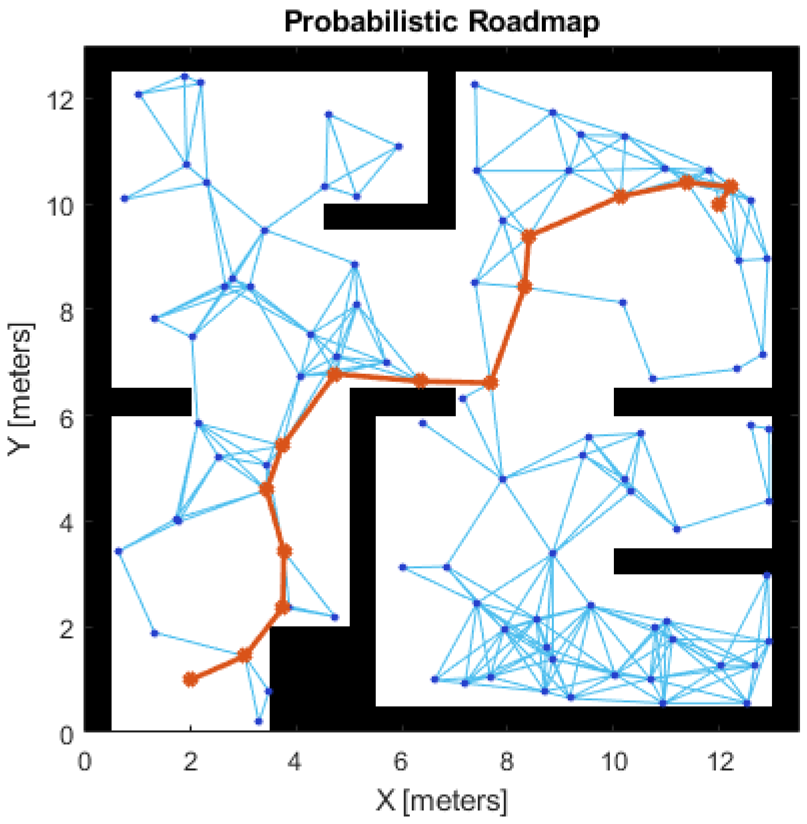
Create a roadmap with 100 nodes and calculate the path. The default `ConnectionDistance` is set to inf. Save the random number generation settings using the rng function. The saved settings enable you to reproduce the same points and see the effect of changing `ConnectionDistance`.

```
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

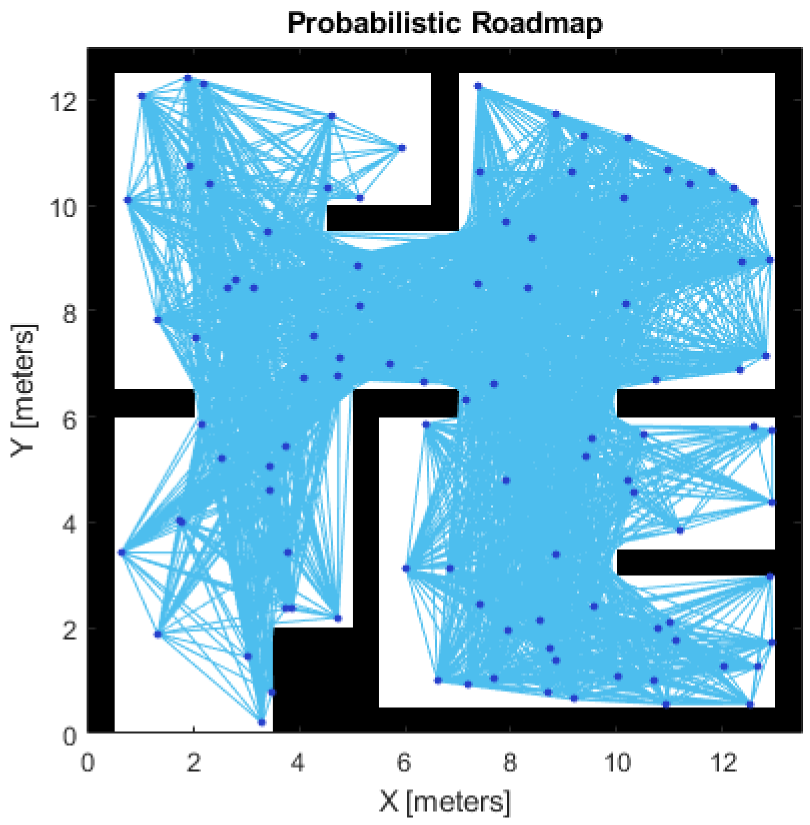
When using the `mobileRobotPRM` object and modifying properties, with each new function call, the object triggers the roadmap points and connections to be recalculated. Because recalculating the map can be computationally intensive, you can reuse the same roadmap by calling `findpath` with different 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);
```

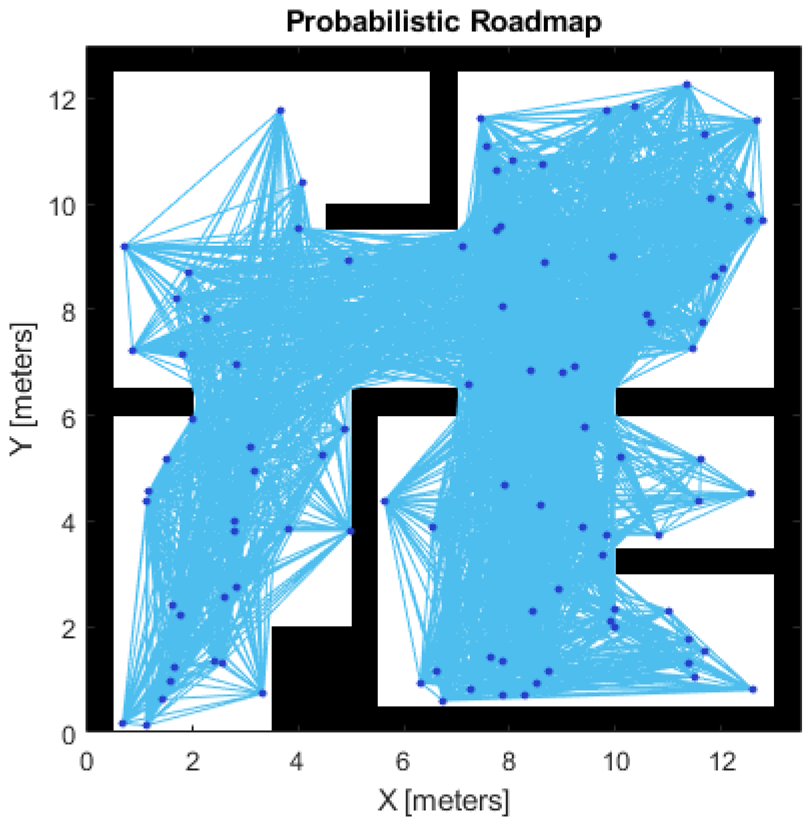
Create a roadmap. Your nodes and connections might look different due to the random placement of nodes.

```
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

`findpath` | `mobileRobotPRM`

Pure Pursuit Controller

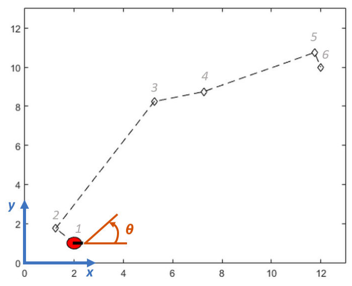
In this section...

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.

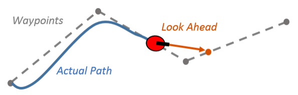
Reference Coordinate System

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).

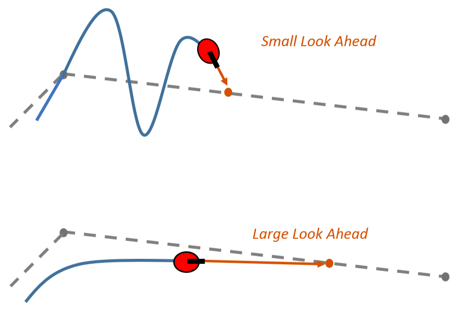


Look Ahead Distance

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

`controllerVFH` | `stateEstimatorPF`

Particle Filter Parameters

In this section...

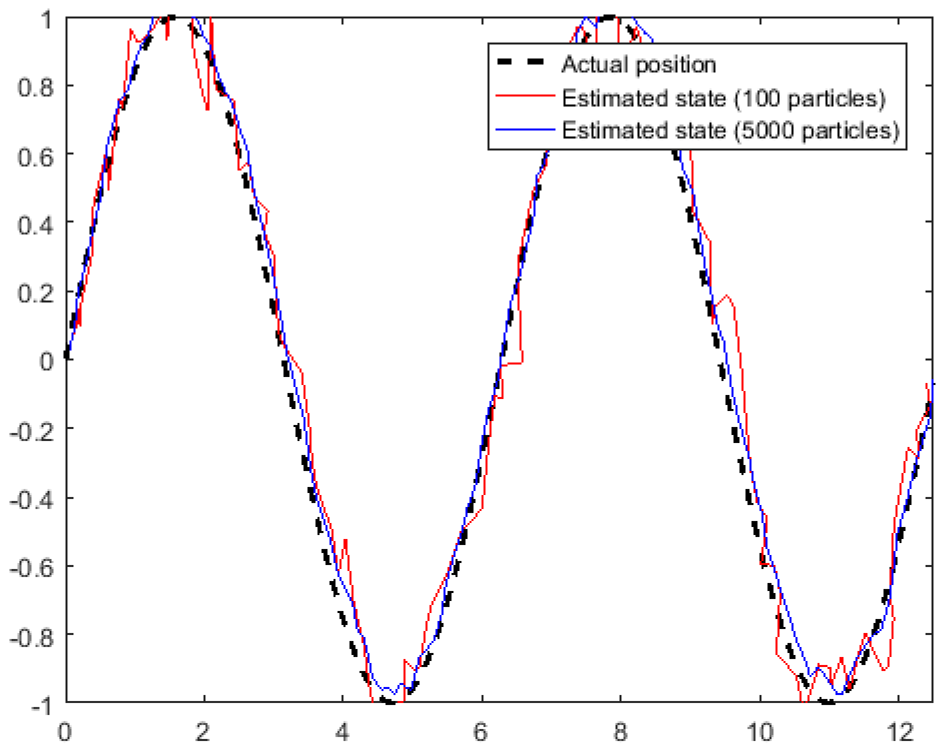
To use the `stateEstimatorPF` particle filter, you must specify parameters such as the number of particles, the initial particle location, and the state estimation method. Also, if you have a specific motion and sensor model, you specify these parameters in the state transition function and measurement 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 1-43.

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

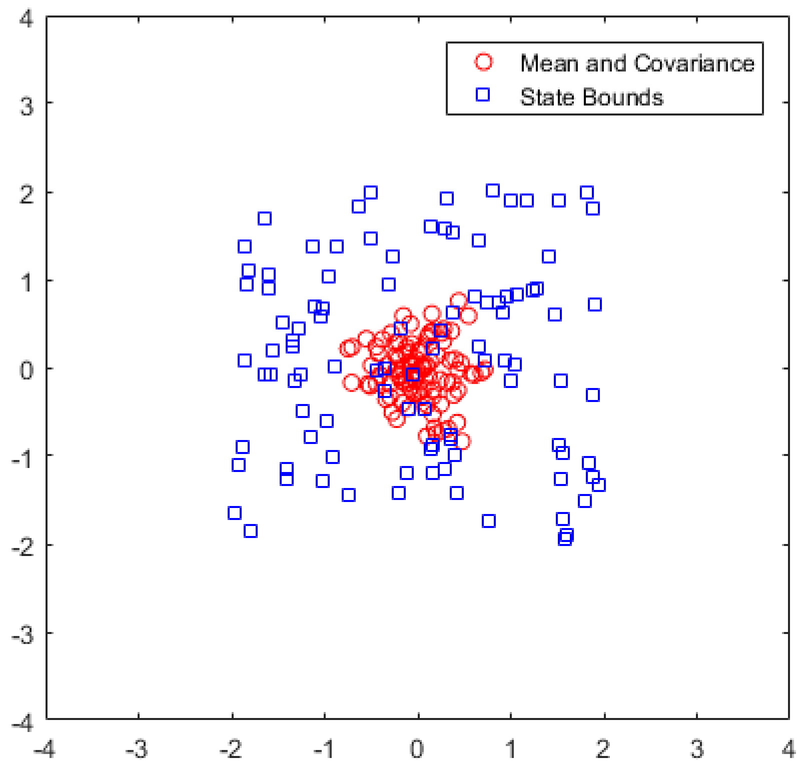
When you initialize your particle filter, you can specify the initial location of the particles using:

- 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 1-43. 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 1-41 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

`resamplingPolicyPF` | `stateEstimatorPF`

Related Examples

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

More About

- “Particle Filter Workflow” on page 1-43

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.

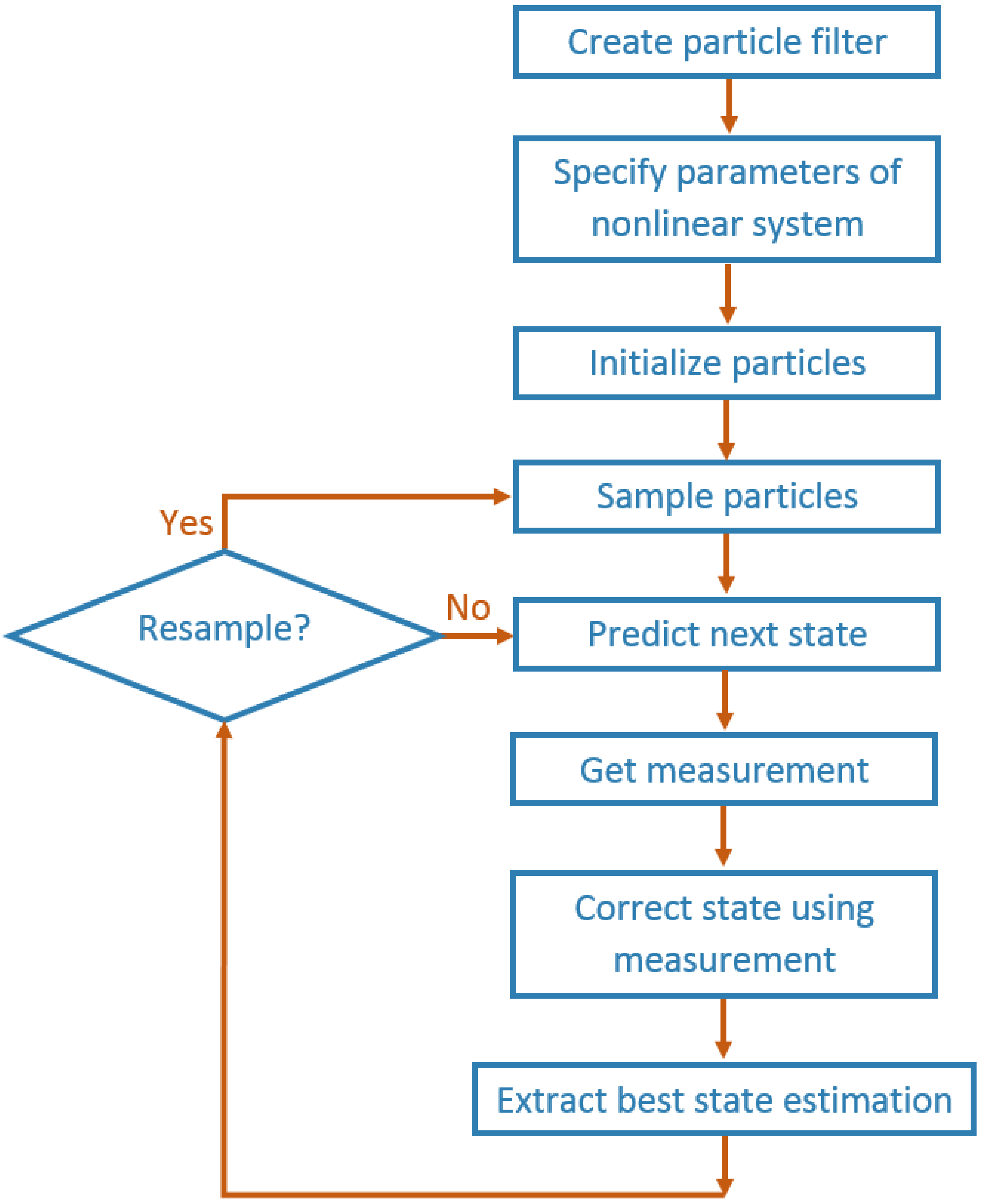
You can apply the particle filter to arbitrary nonlinear system models. Process and measurement noise can follow arbitrary non-Gaussian distributions.

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 1-38.

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 1-38.

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
`correct` | `getStateEstimate` | `initialize` | `predict` | `stateEstimatorPF`

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

More About
- “Particle Filter Parameters” on page 1-38

Standard Units for Robotics System Toolbox

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

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

See Also

More About

- "Coordinate Transformations in Robotics" on page 1-48

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.

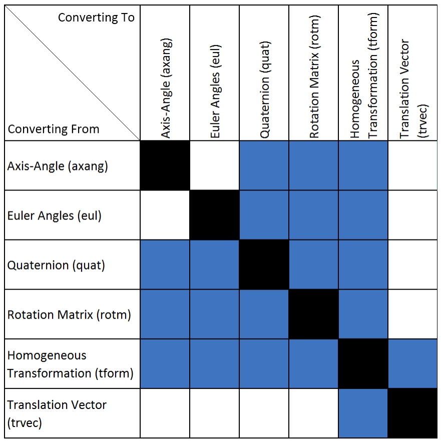
Numeric Representation: 1-by-3 vector

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 1-47

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: 0.005878
Iteration: 2 - Time Elapsed: 1.000761
Iteration: 3 - Time Elapsed: 2.002471
Iteration: 4 - Time Elapsed: 3.000673
Iteration: 5 - Time Elapsed: 4.001496
Iteration: 6 - Time Elapsed: 5.000765
Iteration: 7 - Time Elapsed: 6.000638
Iteration: 8 - Time Elapsed: 7.000956
Iteration: 9 - Time Elapsed: 8.001423
Iteration: 10 - Time Elapsed: 9.000287
```

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: [1x20 double]
           NumPeriods: 20
        AveragePeriod: 1.0203
    StandardDeviation: 0.0422
          NumOverruns: 4
```

Change the `OverrunAction` to `'drop'`. `'drop'` indicates that the `waitfor` function will return at the next time step, even if the `LastPeriod` is greater than the `DesiredRate` property. This effectively drops the iteration that was missed by the slower code execution.

```
rate.OverrunAction = 'drop';
```

Reset `Rate` object and begin loop.

```
reset(rate);

while rate.TotalElapsedTime < loopTime
    if ~isempty(find(slowFrames == floor(rate.TotalElapsedTime)))
        pause(1.1)
    end
    waitfor(rate);
end
stats2 = statistics(rate)
```

```
stats2 = struct with fields:
              Periods: [1x16 double]
           NumPeriods: 16
        AveragePeriod: 1.2501
```

```
    StandardDeviation: 0.4483
          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, `vfhCodeGen`, 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 vfhCodeGen_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 1-14 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` | `inverseKinematics` | `timeit`

Related Examples

- “2-D Path Tracing With Inverse Kinematics” on page 1-14
- 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” (MATLAB)

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 1-61.
- 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

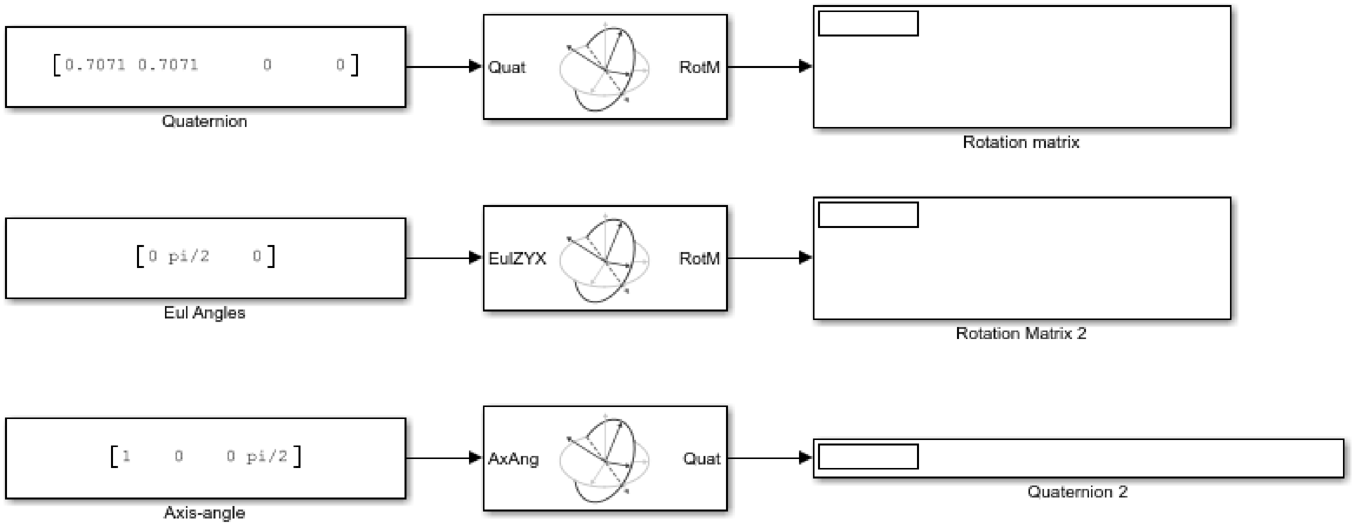
- “Code Generation from MATLAB Code” on page 1-60

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')
```



Compute Geometric Jacobian for Manipulators in Simulink

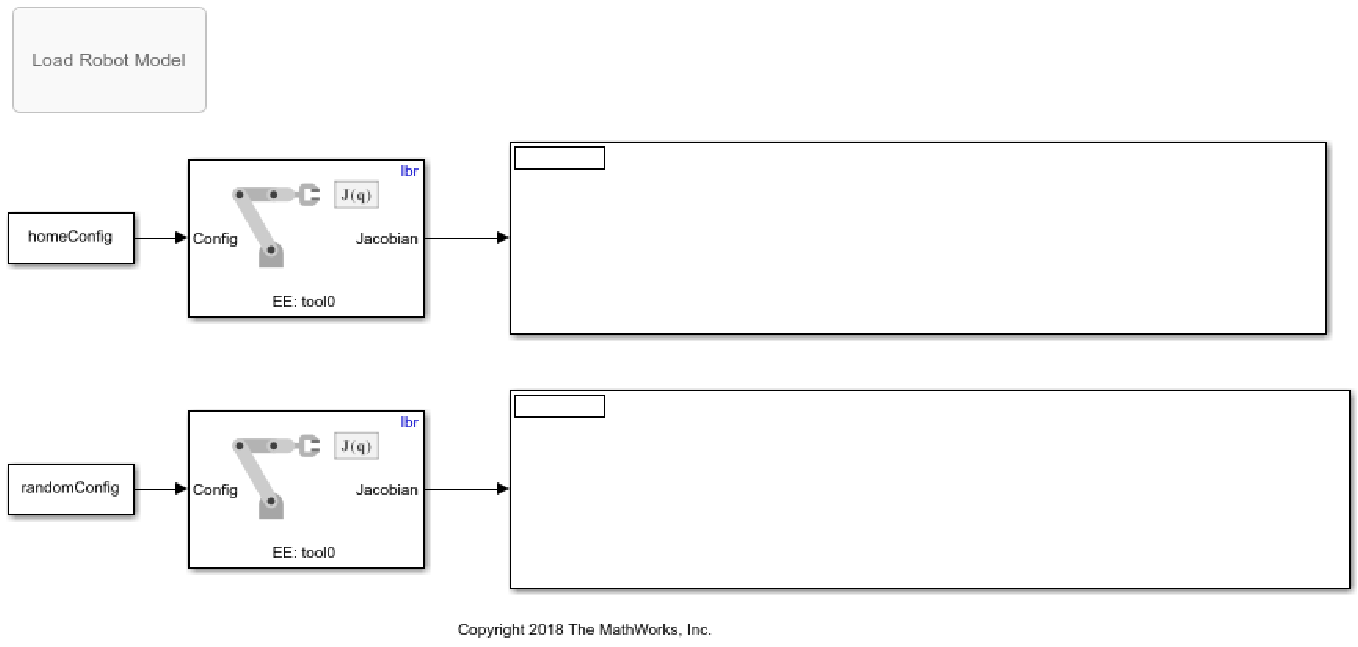
This example shows how to calculate the geometric Jacobian for a robot manipulator by using a `rigidBodyTree` model. The Jacobian maps the joint-space velocity to the end-effector velocity relative to the base coordinate frame. In this example, you define a robot model and robot configurations in MATLAB® 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 | Get Jacobian | Gravity Torque | Inverse Dynamics | Joint Space Mass Matrix

Classes
`RigidBodyTree`

Functions
`externalForce` | `homeConfiguration` | `importrobot` | `randomConfiguration`

Related Examples

- “Perform Safe Trajectory Tracking Control Using Robotics Manipulator Blocks”

Get Transformations for Manipulator Bodies in Simulink

This example shows how to get the transformation between bodies in a `rigidBodyTree` robot model. In this example, you define a robot model and robot configuration in MATLAB® and pass them to Simulink® to be used with the manipulator algorithm block.

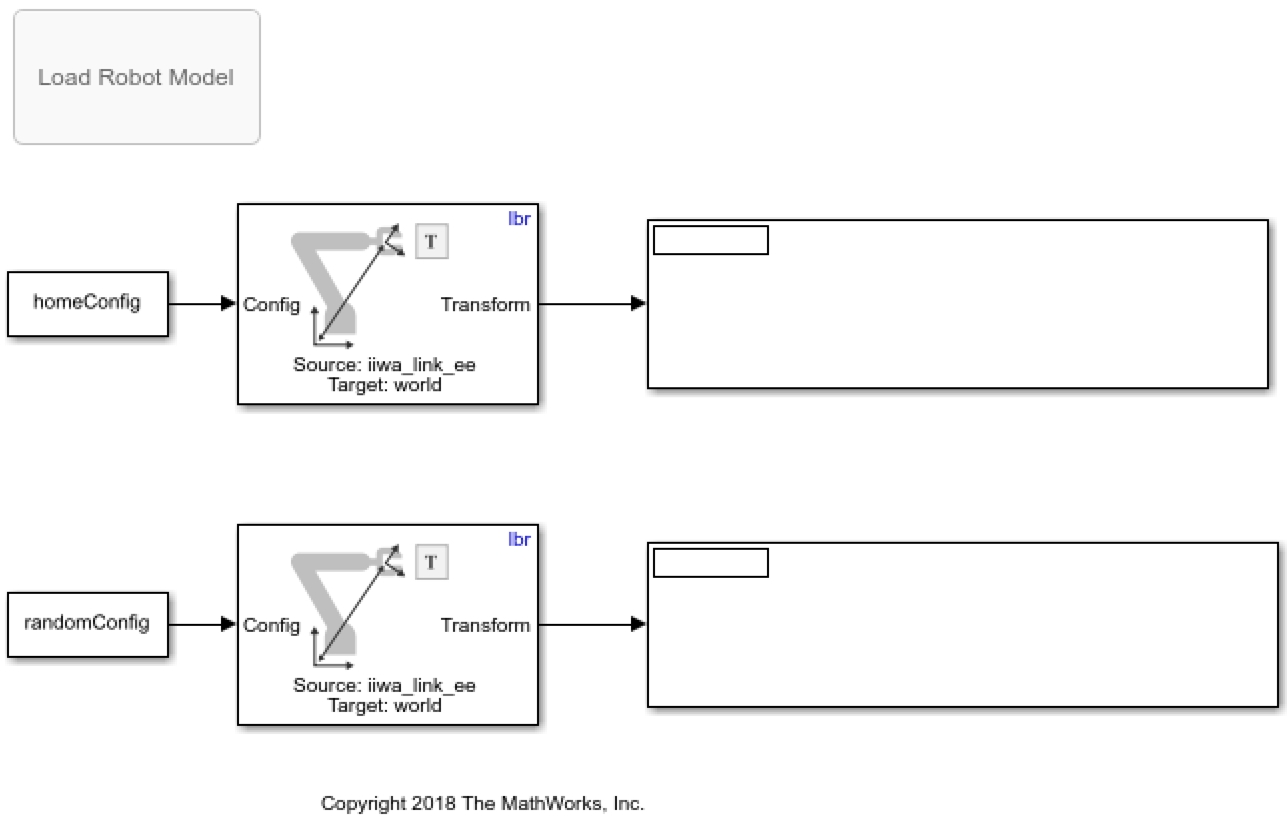
Load the robot model of the KUKA LBR robot as a `RigidBodyTree` object. Use the `homeConfiguration` function to get the home configuration as joint positions of the robot.

```
load('exampleLBR.mat','lbr')
lbr.DataFormat = 'column';

homeConfig = homeConfiguration(lbr);
randomConfig = randomConfiguration(lbr);
```

Open the model. If necessary, use the **Load Robot Model** callback button to reload the robot model and configuration vectors.

The Get Transform block calculates the transformation from the source body to the target body. This transformation converts coordinates from the source body frame to the given target body frame. This example gives you transformations to convert coordinates from the `'iiwa_link_ee'` end effector into the `'world'` base coordinates.

```
open_system('get_transform_example.slx')
```



Run the model to get the transformations.

See Also

Blocks
Forward Dynamics | Get Jacobian | Get Transform | Gravity Torque | Inverse Dynamics | Joint Space Mass Matrix

Classes
`RigidBodyTree`

Functions
`homeConfiguration` | `importrobot` | `randomConfiguration`

Related Examples

- "Perform Safe Trajectory Tracking Control Using Robotics Manipulator Blocks"

Calculate Manipulator Gravity Dynamics in Simulink

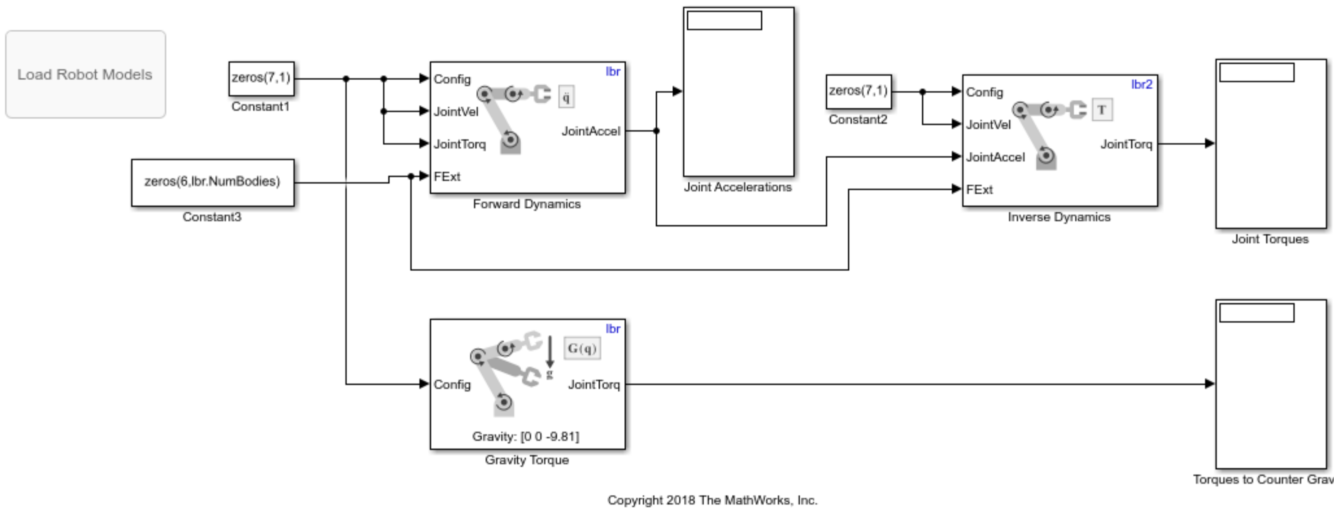
This example shows how to use the manipulator algorithm blocks to compute and compare dynamics due to gravity for a manipulator robot.

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

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

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

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



The Forward Dynamics block calculates the joint accelerations due to gravity for a given `lbr` robot configuration with no initial velocity, torque, or external force. The Inverse Dynamics block then computes the torques needed for the joint to create those same accelerations with no gravity by using the `lbr2` robot. Finally, the Gravity Torque block calculates the torque required to counteract gravity for the `lbr` robot.

Run the model. Besides some small numerical differences, the gravity torque and the torque required for accelerations due to gravity are the same value with opposite directions.

See Also

Blocks
Forward Dynamics | Get Jacobian | Gravity Torque | Inverse Dynamics | Joint Space Mass Matrix | Velocity Product Torque

Classes
`RigidBodyTree`

Functions
`externalForce` | `homeConfiguration` | `importrobot` | `randomConfiguration`

Related Examples

- “Perform Safe Trajectory Tracking Control Using Robotics Manipulator Blocks”

Trace An End-Effector Trajectory with Inverse Kinematics in Simulink

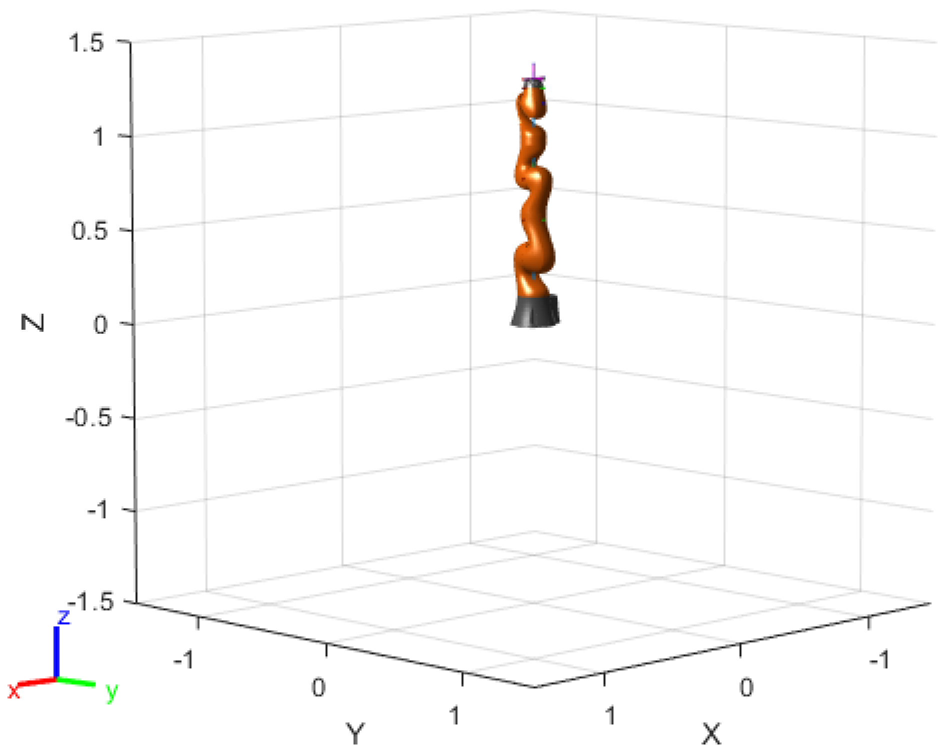
Use a rigid body robot model to compute inverse kinematics using Simulink®. Define a trajectory for the robot end effector and loop through the points to solve robot configurations that trace this trajectory.

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

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

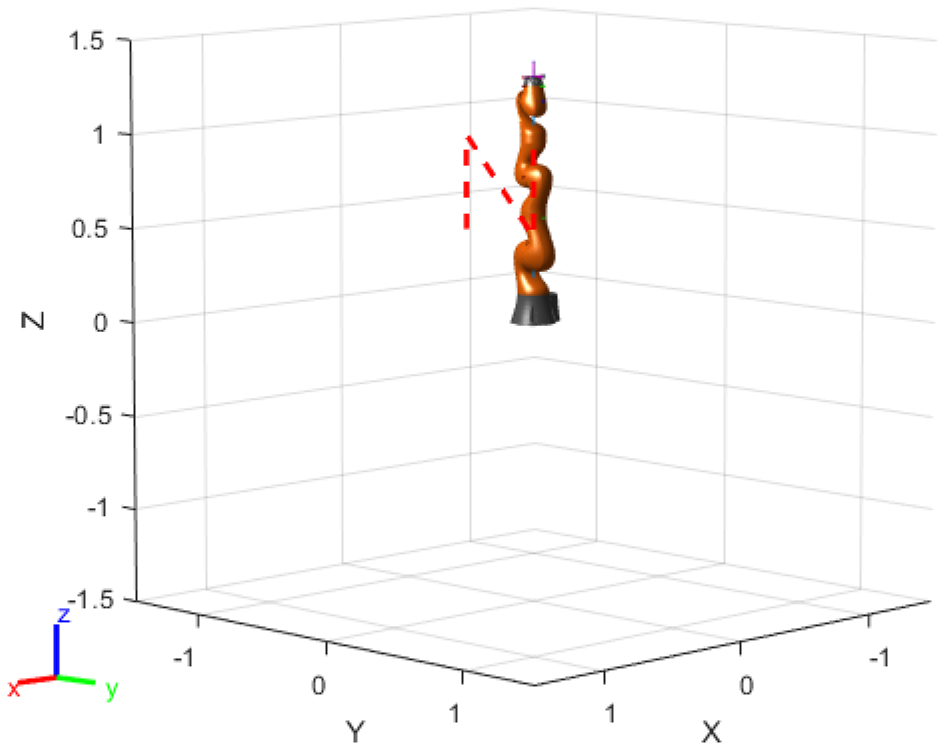
View the robot.

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



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

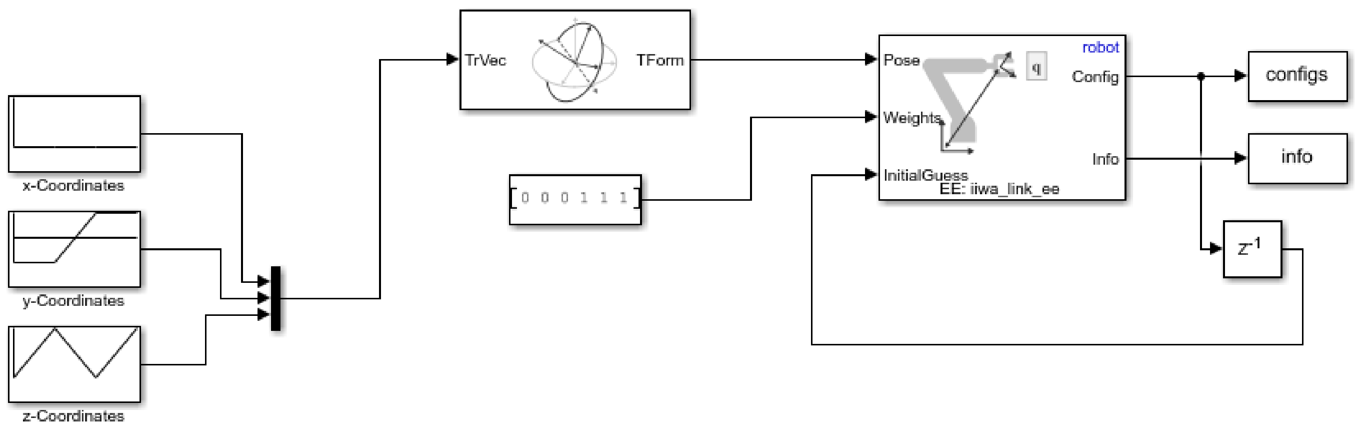
```
x = 0.5*zeros(1,4)+0.25;
y = 0.25*[-1 -1 1 1];
z = 0.25*[-1 1 -1 1] + 0.75;

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')
```

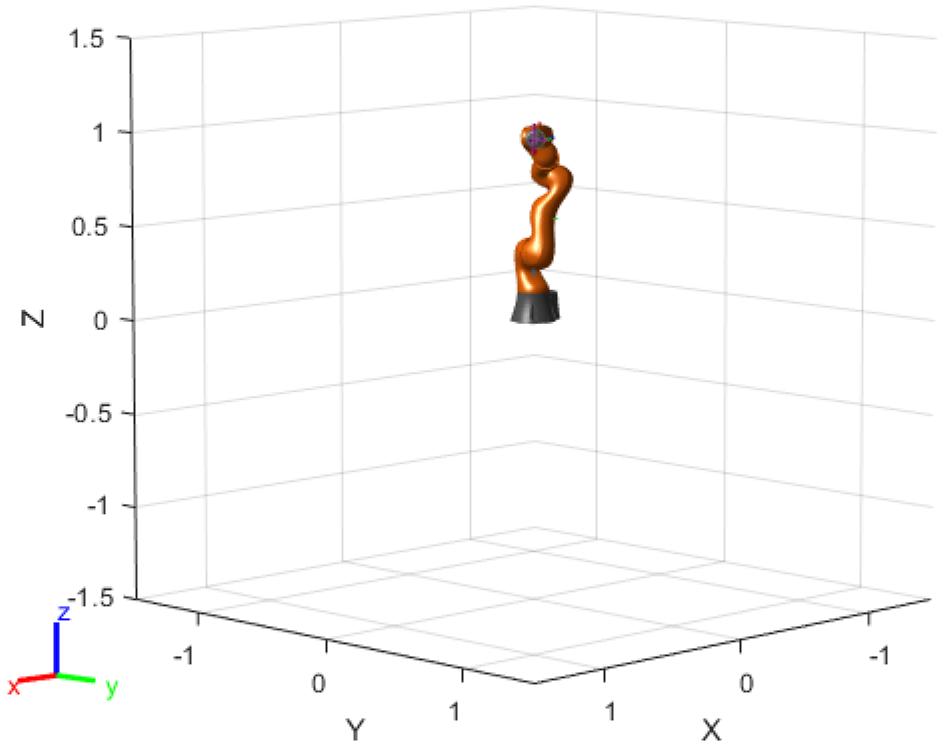


% Run the simulation. The model should generate the robot configurations (`configs`) that follow the specified trajectory for the end effector.

```
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`.

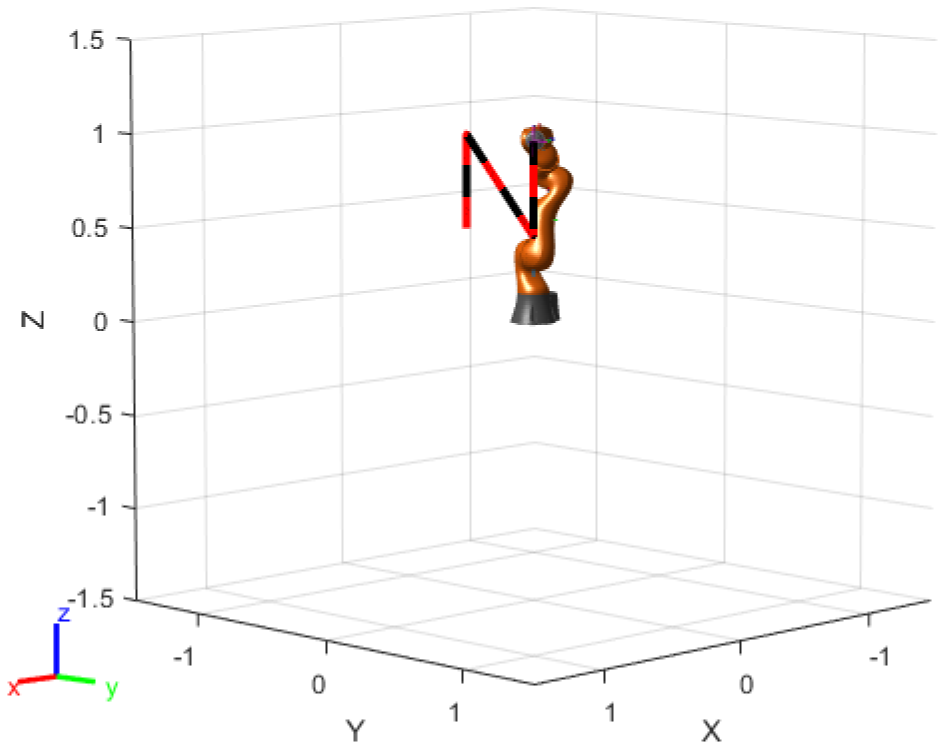
```
figure('Visible','on');
tformIndex = 1;
for i = 1:10:numel(configs.Data)/7
    currConfig = configs.Data(:,1,i);
    show(robot,currConfig);
    drawnow

    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
`generalizedInverseKinematics` | `inverseKinematics` | `rigidBodyTree`

Blocks
Get Transform | Inverse Dynamics | Inverse Kinematics

Related Examples

- “Control PR2 Arm Movements Using ROS Actions and Inverse Kinematics”
- “Inverse Kinematics Algorithms” on page 1-10

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.

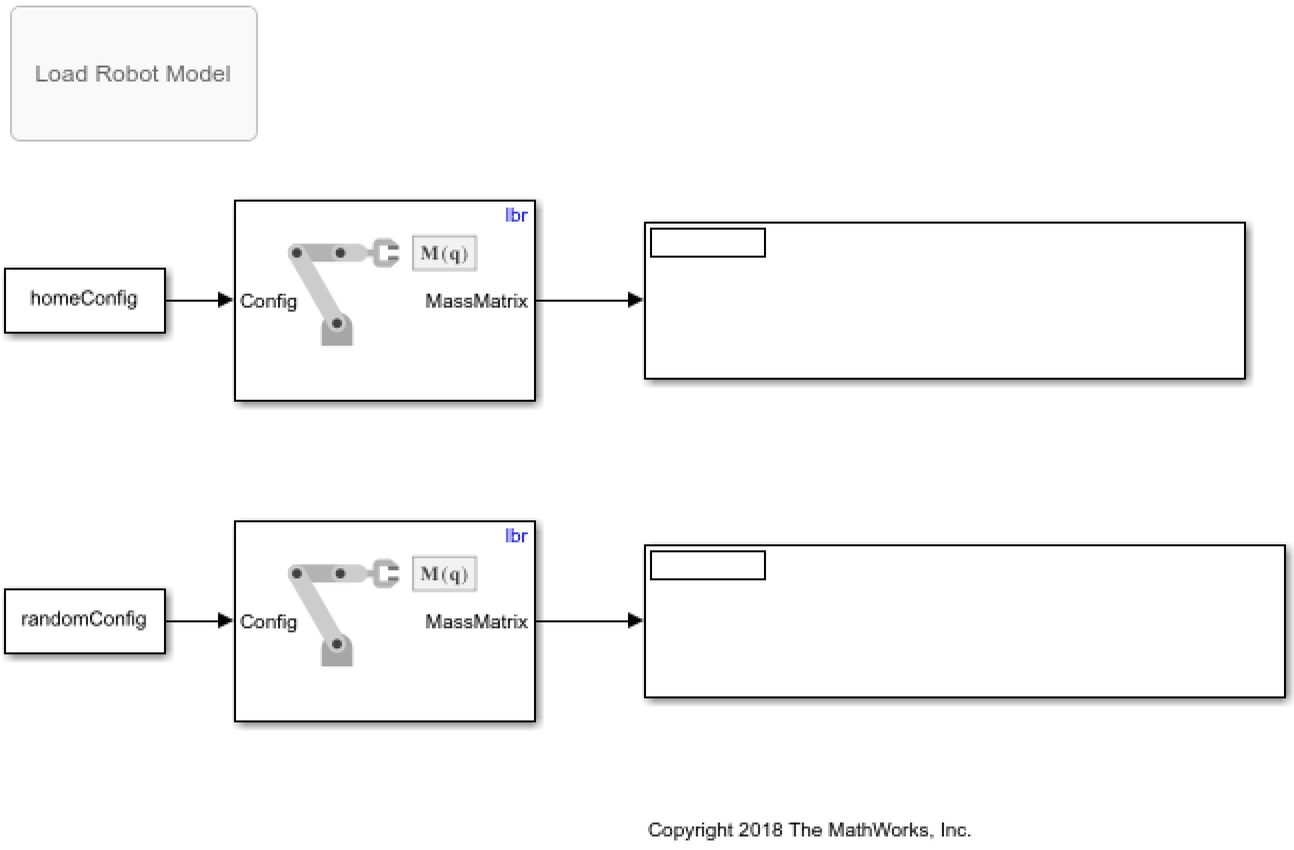
Load a `RigidBodyTree` object that models a KUKA LBR robot. Use the `homeConfiguration` functions to get the home configuration or home joint positions of the robot. Use the `randomConfiguration` function to generate a random configuration within the robot joint limits.

```
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
Forward Dynamics | Get Jacobian | Get Transform | Gravity Torque | Inverse Dynamics

Classes
`RigidBodyTree`

Functions
`homeConfiguration` | `importrobot` | `randomConfiguration`

Related Examples

- “Perform Safe Trajectory Tracking Control Using Robotics Manipulator Blocks”

Generate Heading and Yaw Commands for Orbit Following in Simulink®

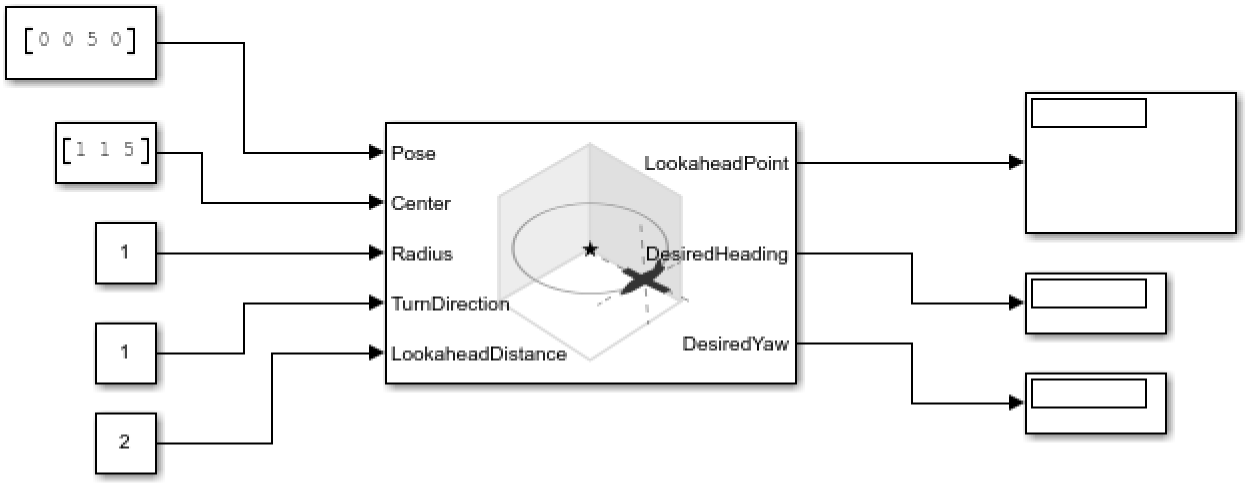
This example shows how to use the **UAV Orbit Follower** block to generate heading and yaw commands for orbiting a location of interest with a UAV.

NOTE: This example requires you to install the UAV Library for Robotics System Toolbox®. Call `roboticsAddons` to open the Add-ons Explorer and install the library.

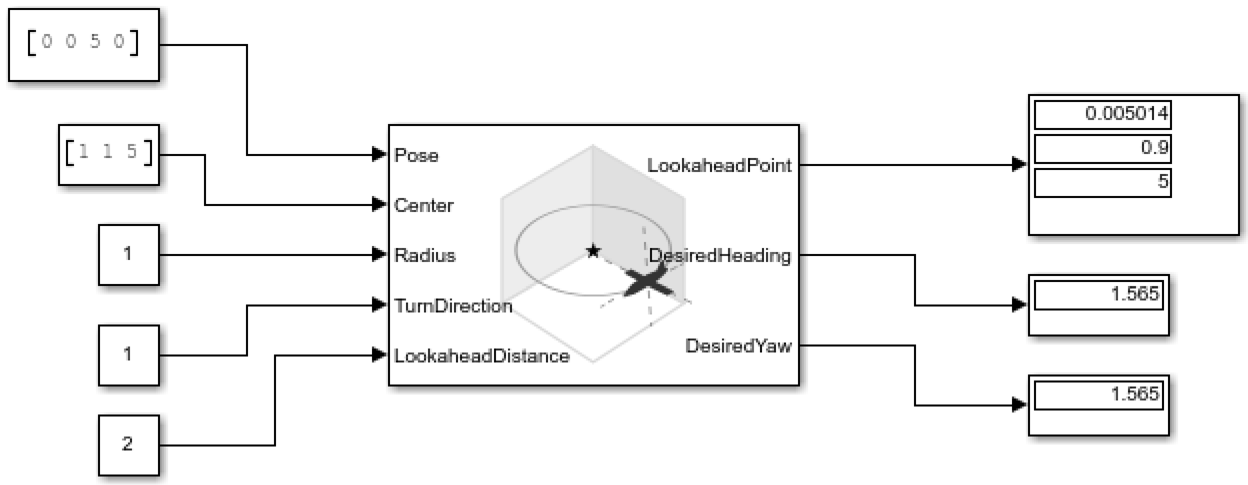
Open the model. Click **Open Live Script** to get a copy of the Simulink® model.

This model illustrates the inputs and the outputs of the block. You must specify the current UAV pose as an `[x;y;z;heading]`. Also, give the orbit center location, orbit radius, turn direction, and lookahead distance on the path. The lookahead distance is important for tuning the path tracking.

```
open_system("uav_orbit_follower_ex1.slx")
```



Run the model to get the desired heading and yaw for following the orbit. These outputs can be used to generate commands for a UAV.

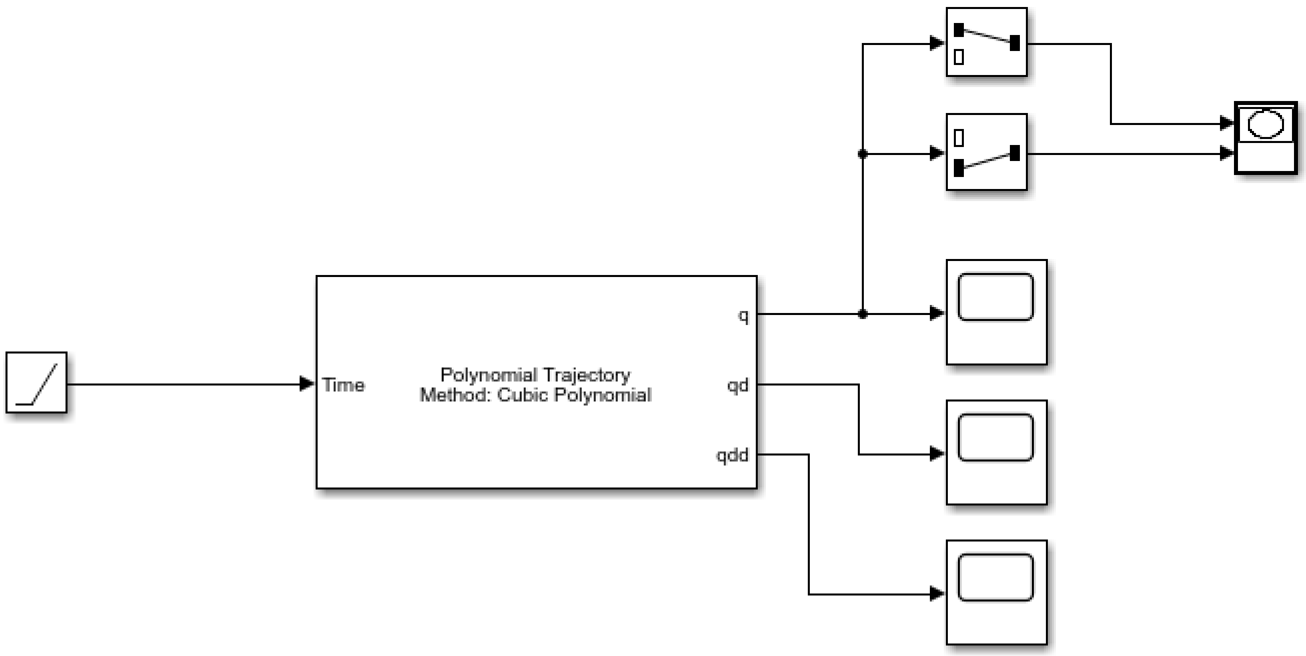

[0 0 5 0]
[1 1 5]
1
1
2
Pose
Center
Radius
TurnDirection
LookaheadDistance
LookaheadPoint
DesiredHeading
DesiredYaw
0.005014
0.9
5
1.565
1.565

Generate Cubic Polynomial Trajectory

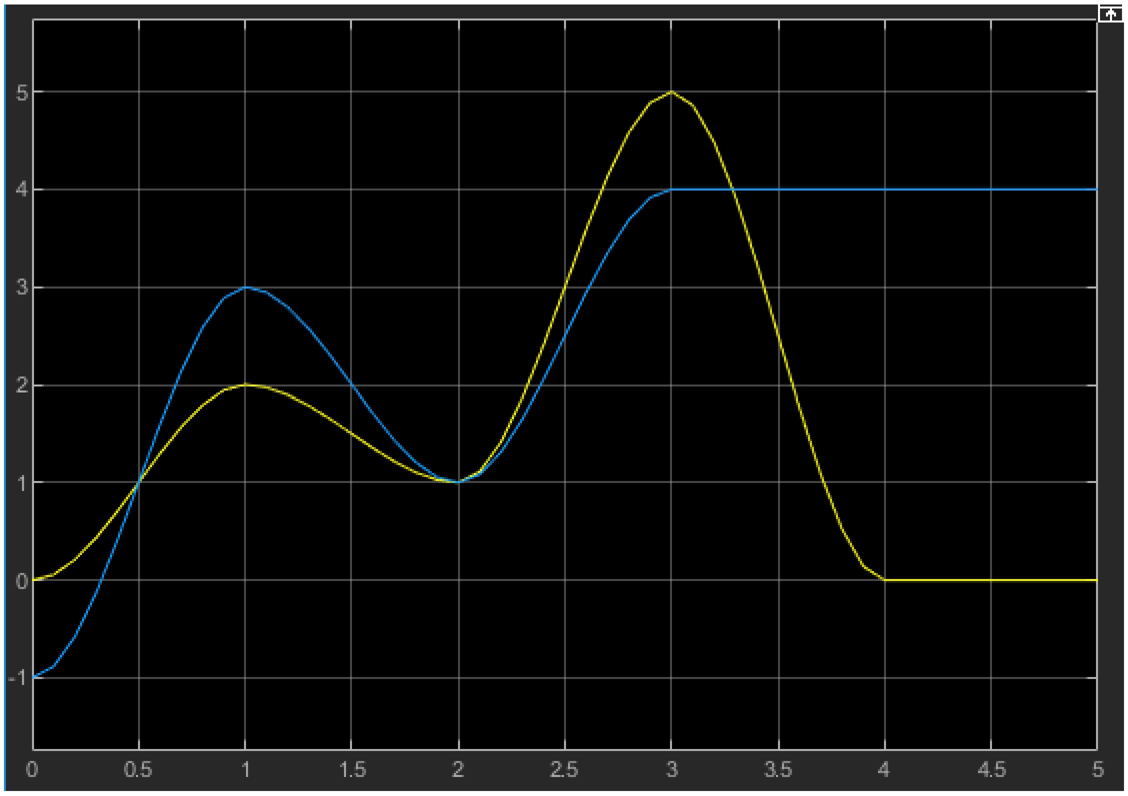
This example shows how to generate a cubic polynomial trajectory using the **Polynomial Trajectory** block.

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.

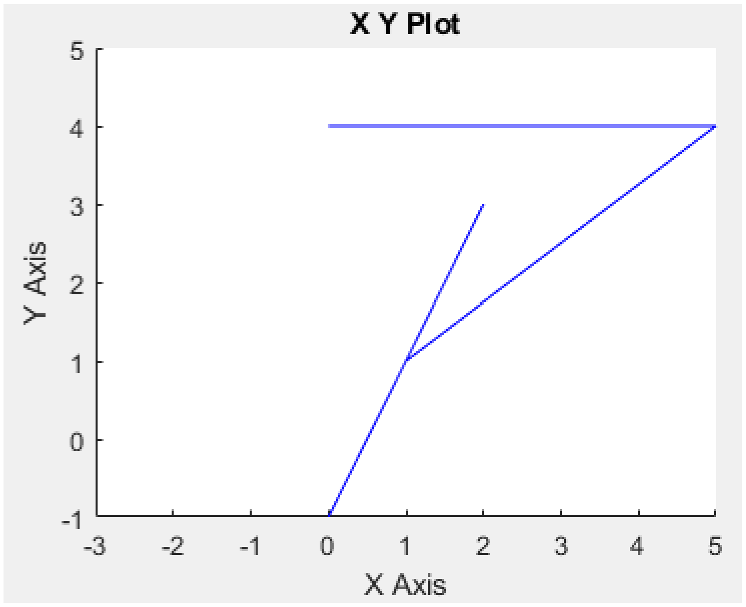
```
open_system('cubic_polytraj_ex1.slx')
```



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.



0
0.5
1
1.5
2
2.5
3
3.5
4
4.5
5
-1
0
1
2
3
4
5



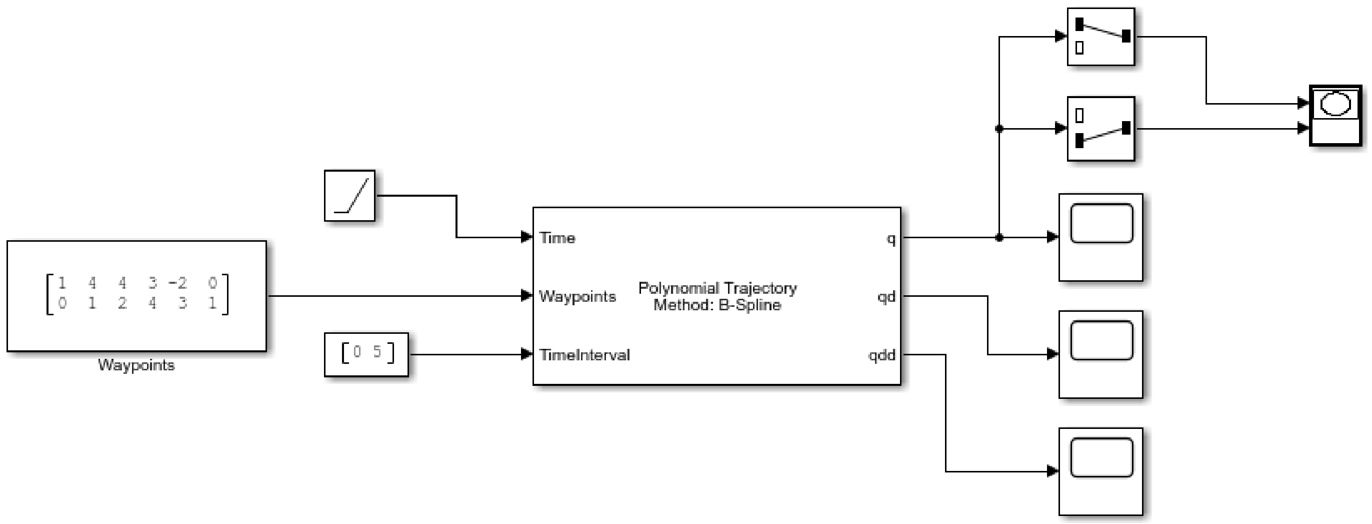
X Y Plot
Y Axis
X Axis
-3
-2
-1
0
1
2
3
4
5
-1
0
1
2
3
4
5

Generate B-Spline Trajectory

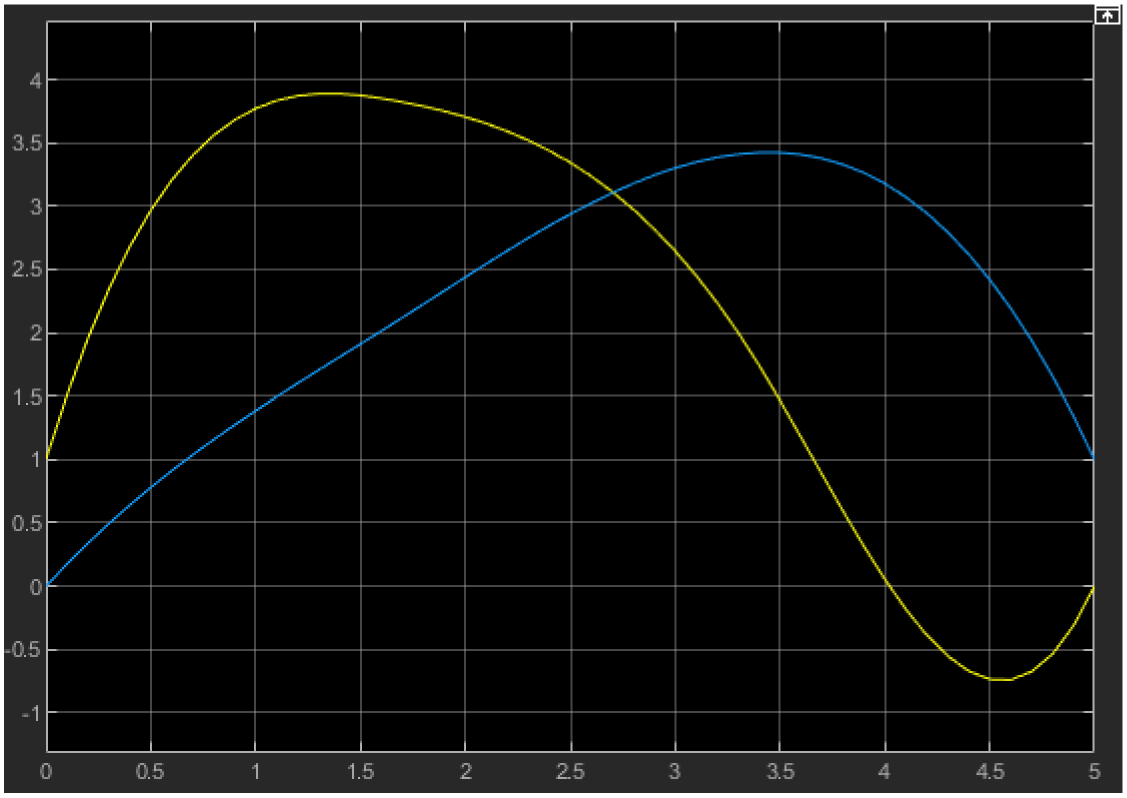
This example shows how to generate a B-spline trajectory using the **Polynomial Trajectory** block.

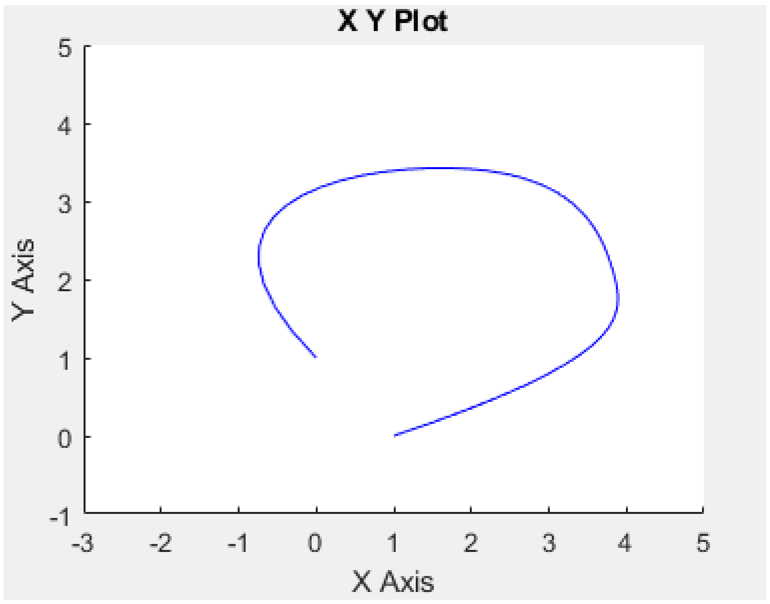
Open the model. The **Waypoints** and **TimeInterval** inputs are toggled in the block mask by setting **Waypoint source** to `External`. For B-splines, the waypoints are actually control points for the convex polygon, but the first and last waypoints are met. The **Time** input is just a ramp signal 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

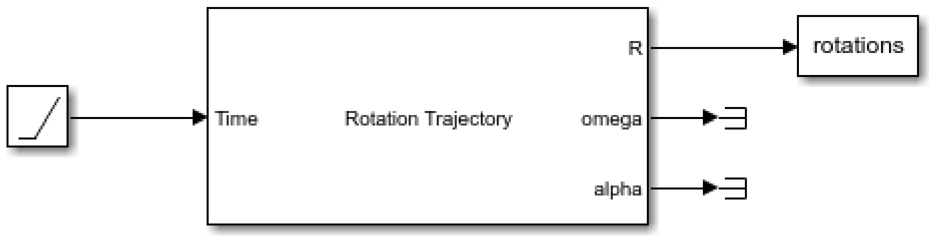

X Y Plot
Y Axis
X Axis
5
4
3
2
1
0
-1
-3
-2
-1
0
1
2
3
4
5

Generate Rotation Trajectory

This example shows how to generate a trajectory that interpolates between rotations using the **Rotation Trajectory** block.

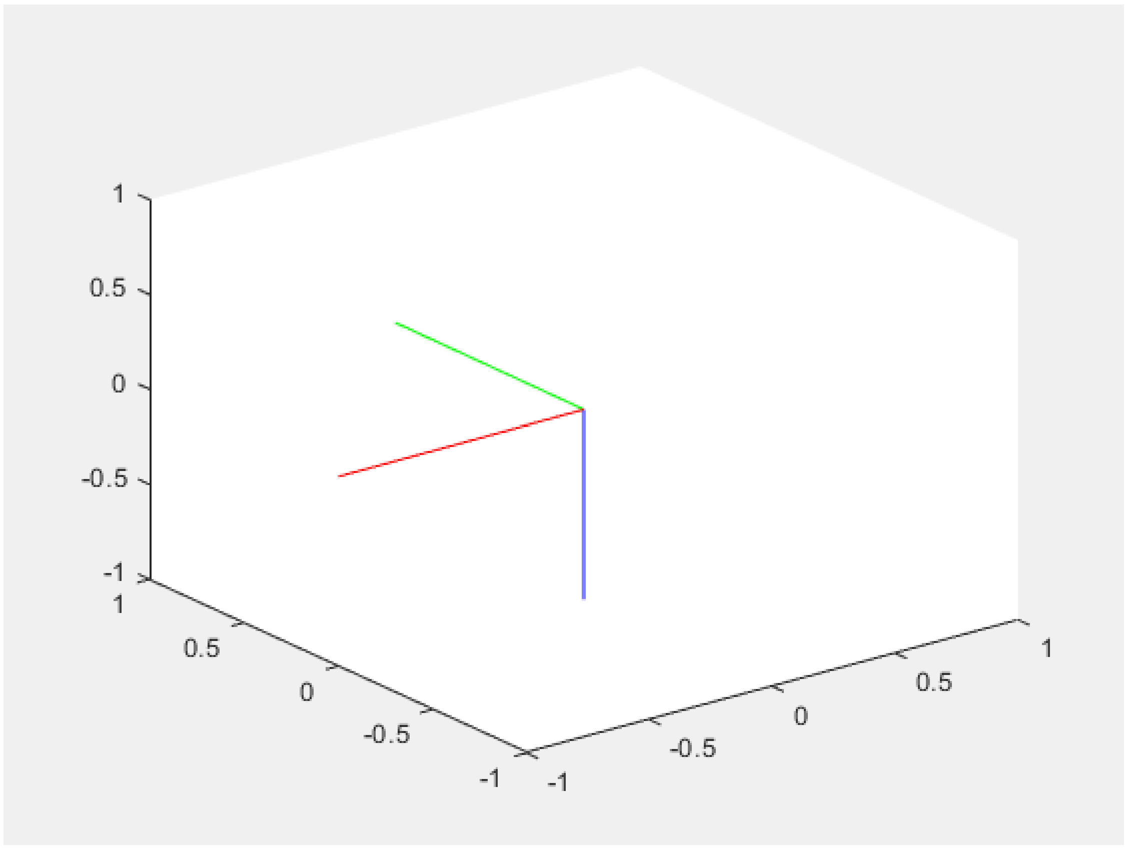
Open and simulate the model. The **Rotation Trajectory** block outputs the trajectory between two rotations and saves the intermediate rotations to the `rotations` variable. This example generates a simple rotation trajectory from the *x*-axis to the *z*-axis.

```
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

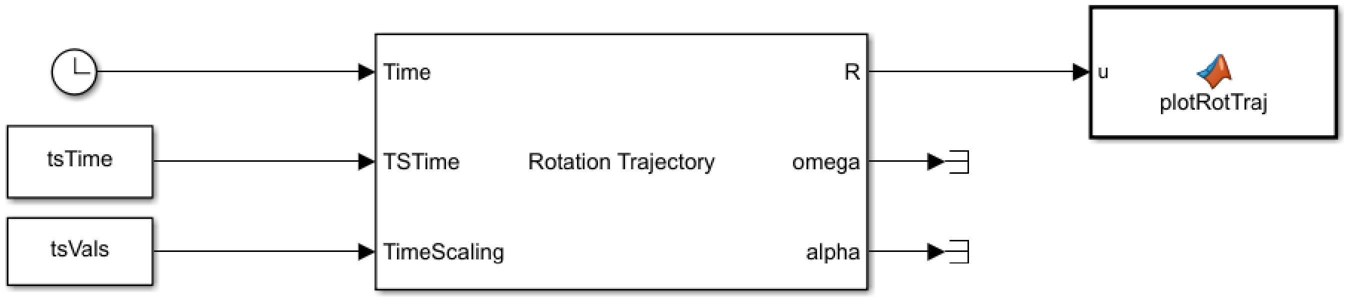
Create vectors for the time scaling time vector and time scaling values. The time scaling time is linear vector from 0 to 5 seconds at 0.1 second intervals. The time scaling values follow a cubic trajectory with the appropriate derivatives specified for velocity and acceleration. These values are used 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.

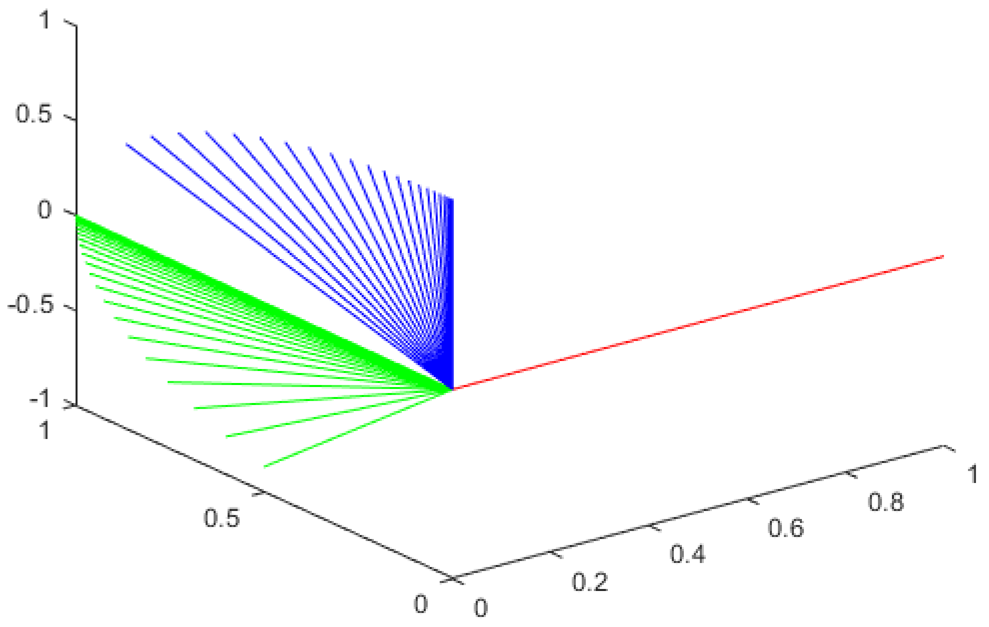
```
open_system("custom_time_scaling_rotation")
```



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])
```

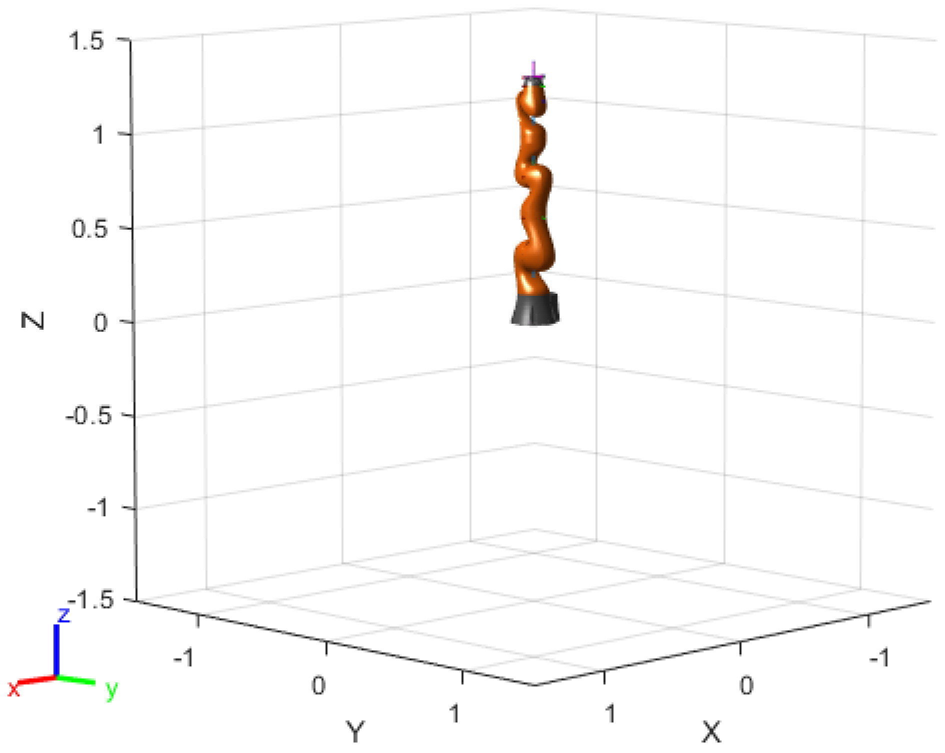
```
tform1 = 4×4

    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
```

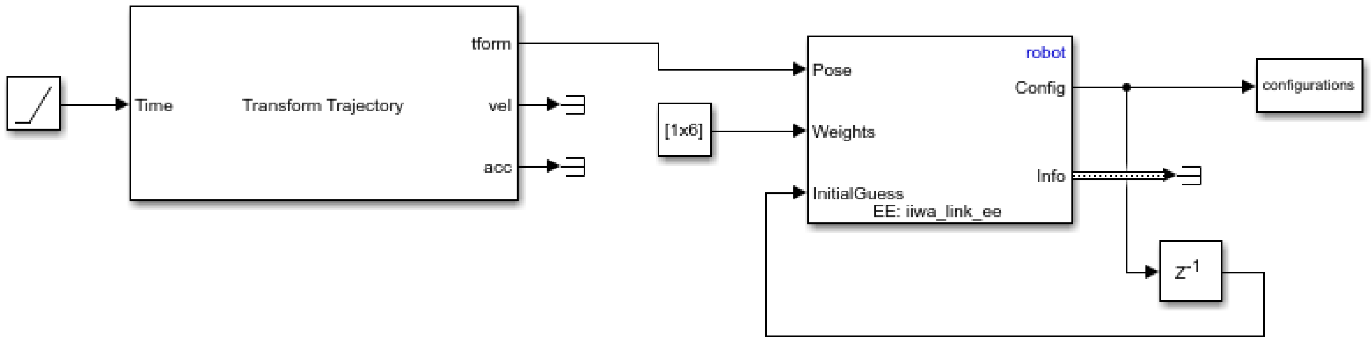
Import the robot model and specify the data format for Simulink®.

```
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')
```



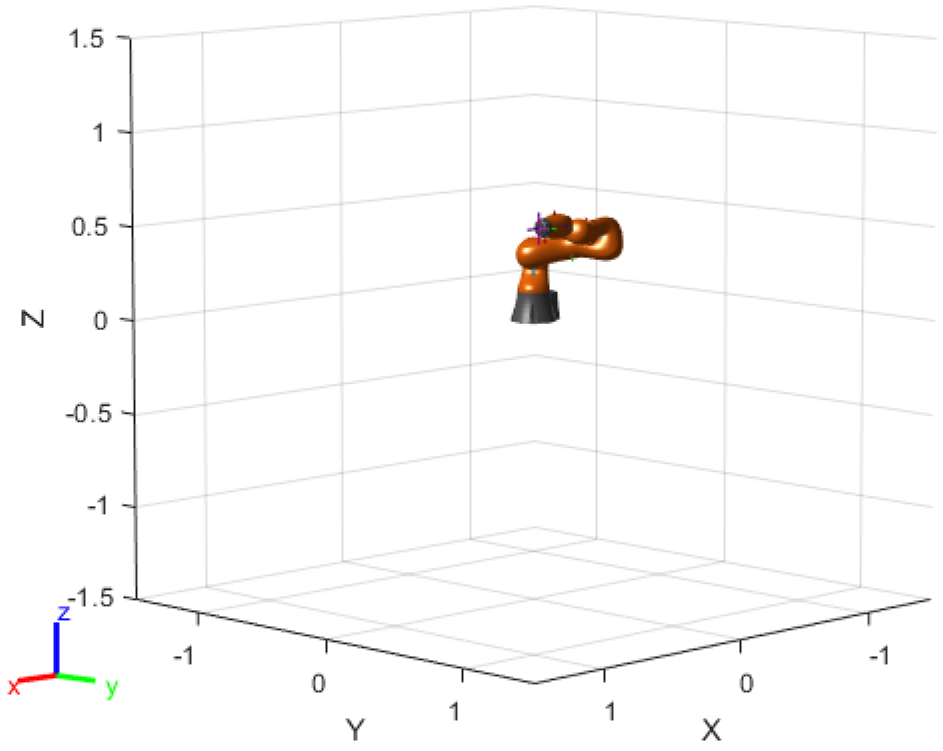
Run the simulation and get the robot configurations.

```
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
```



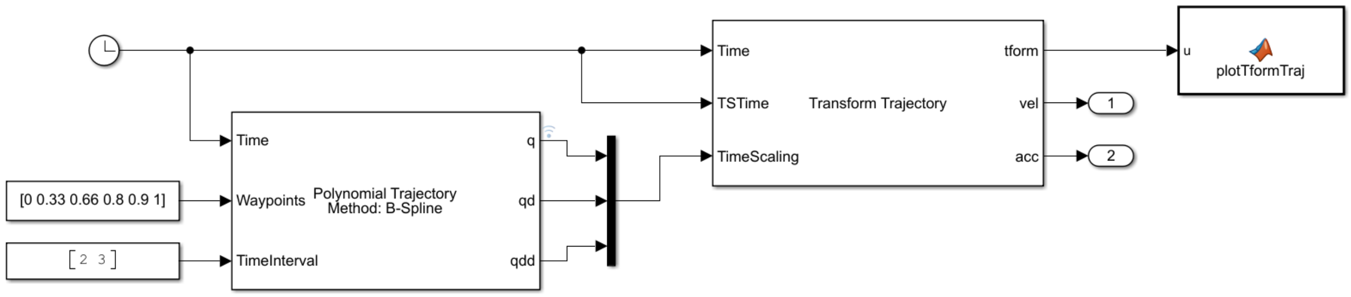
Use Custom Time Scaling for a Transform Trajectory

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.

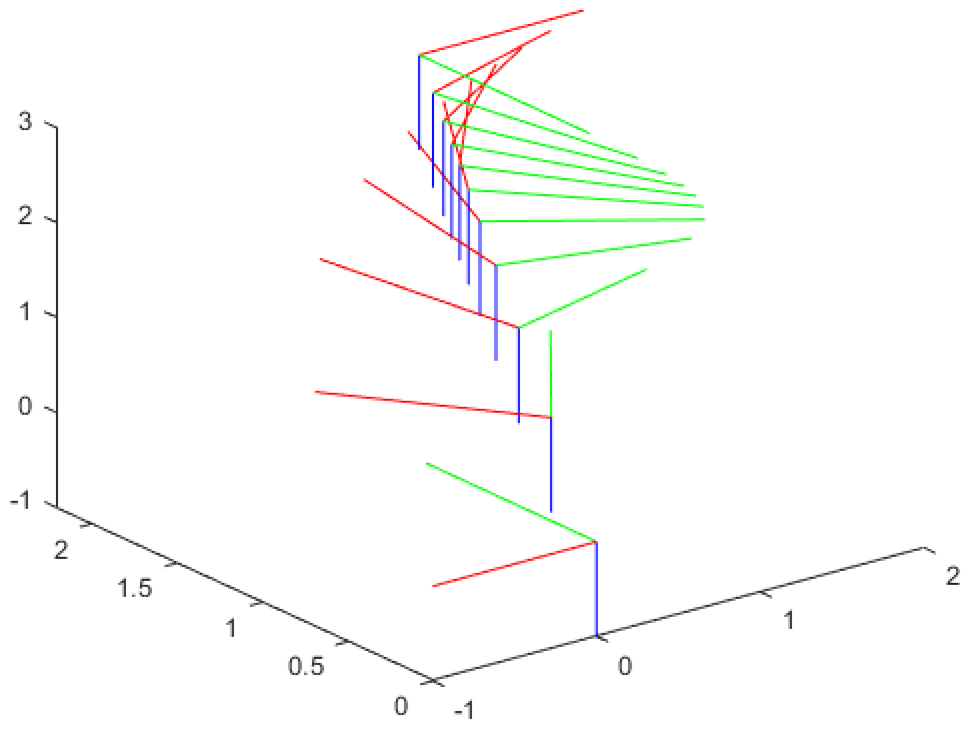
```
open_system("custom_time_scaling_transform")
```



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")
hold off
```

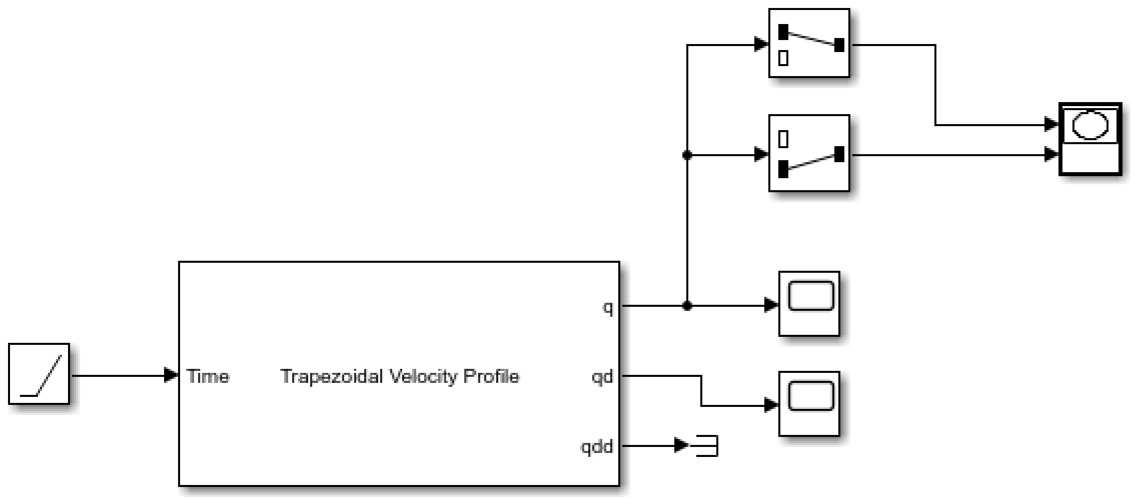

3
2
1
0
-1
2
1.5
1
0.5
0
-1
0
1
2

Generate Trapezoidal Velocity Trajectory

This example shows how to generate a trapezoidal velocity trajectory using the **Trapezoidal Velocity** block.

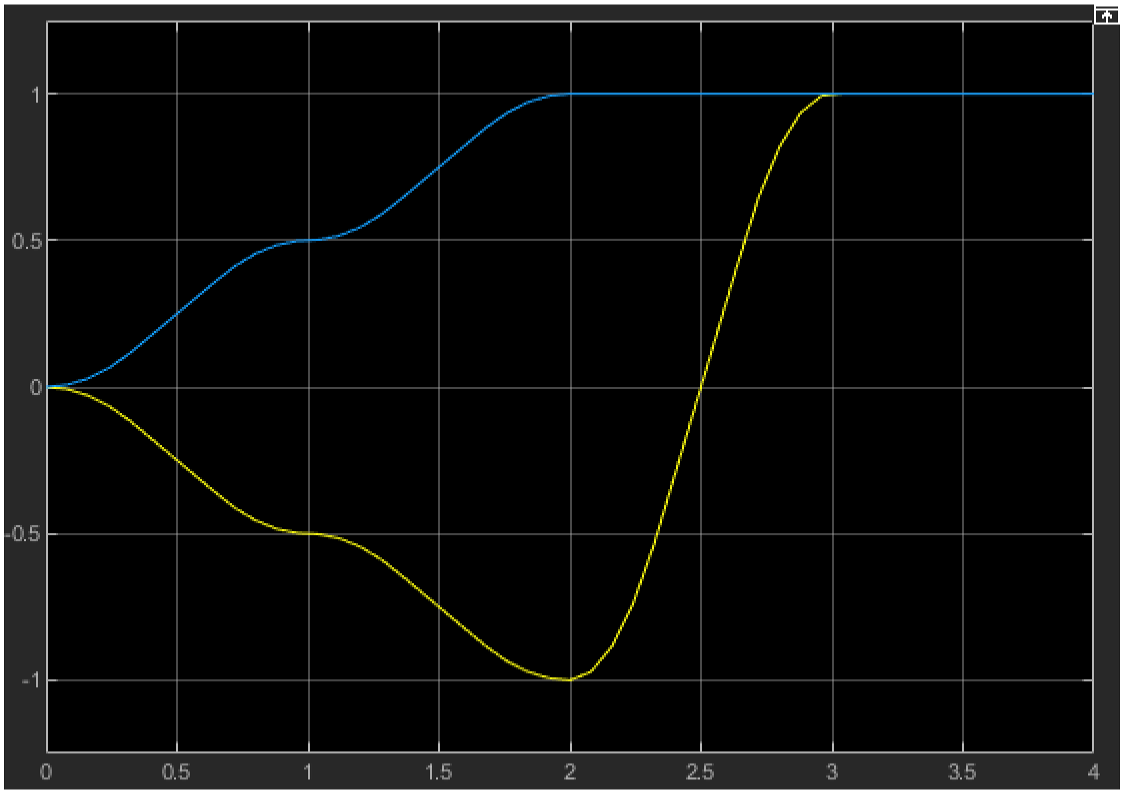
Open the model. The waypoints are specified in the block mask. The position and velocity outputs are connect to scopes and the position is plotted to an **XY Plot**. The **Time** input is just a ramp signal to simulate time progressing.

```
open_system('trapvel_traj_ex1.slx')
```

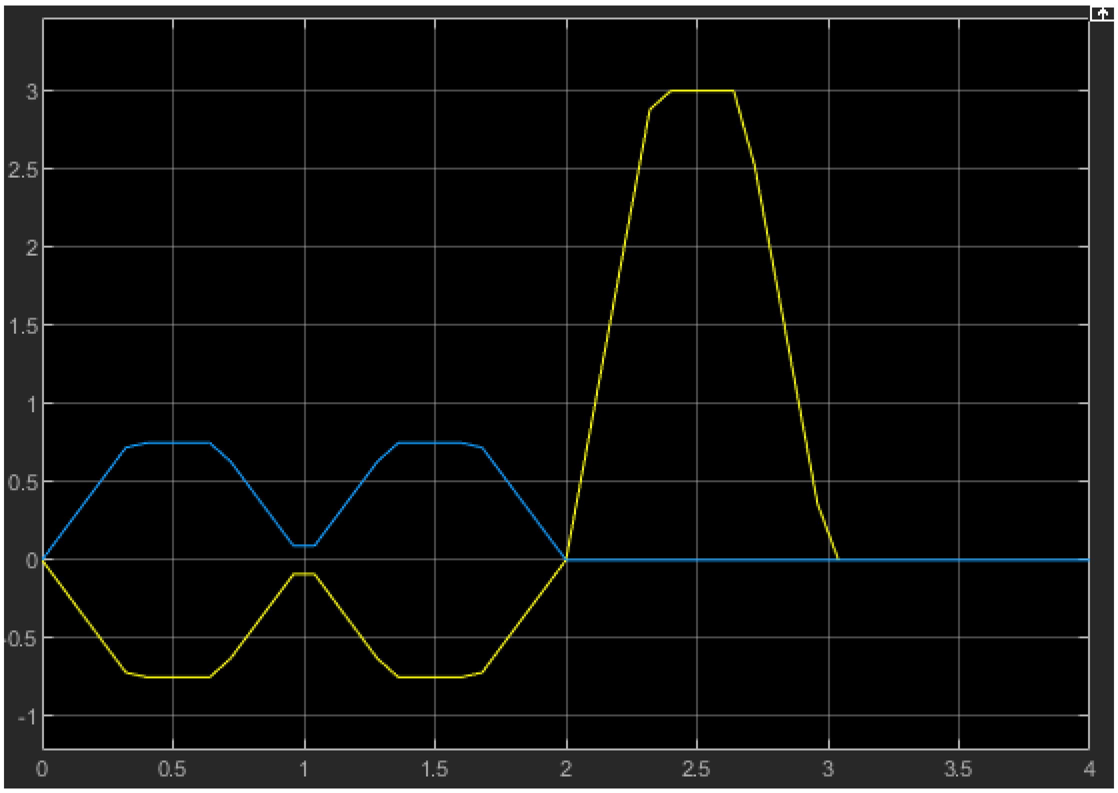


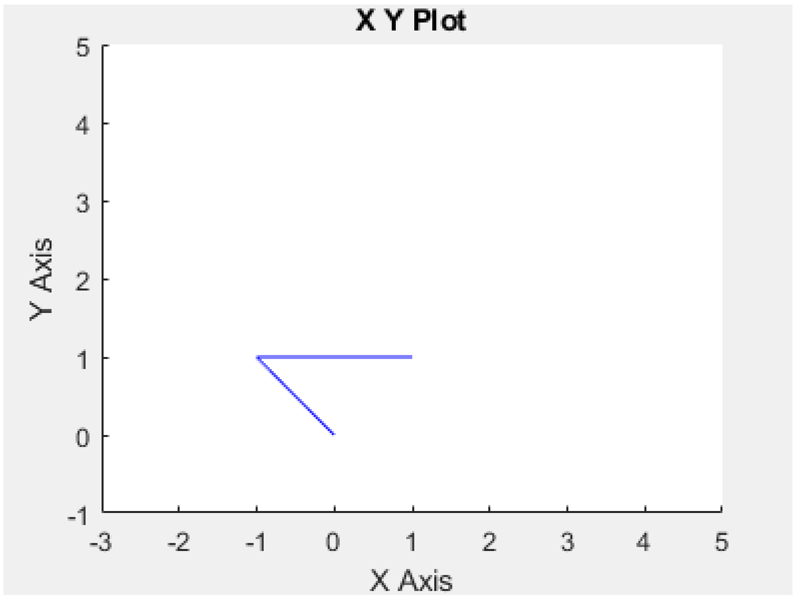
Run the Simulation. The first figure shows the output of the `q` vector for the positions of the trajectory. The second figure shows the `qdd` vector for the velocity. Notice the trapezoidal profile for each waypoint transition. The **XY Plot** shows the actual 2-D trajectory, which hits the specified waypoints.

Positions



Velocities




X Y Plot
Y Axis
X Axis
5
4
3
2
1
0
-1
-3
-2
-1
0
1
2
3
4
5

Compute Velocity Product for Manipulators in Simulink

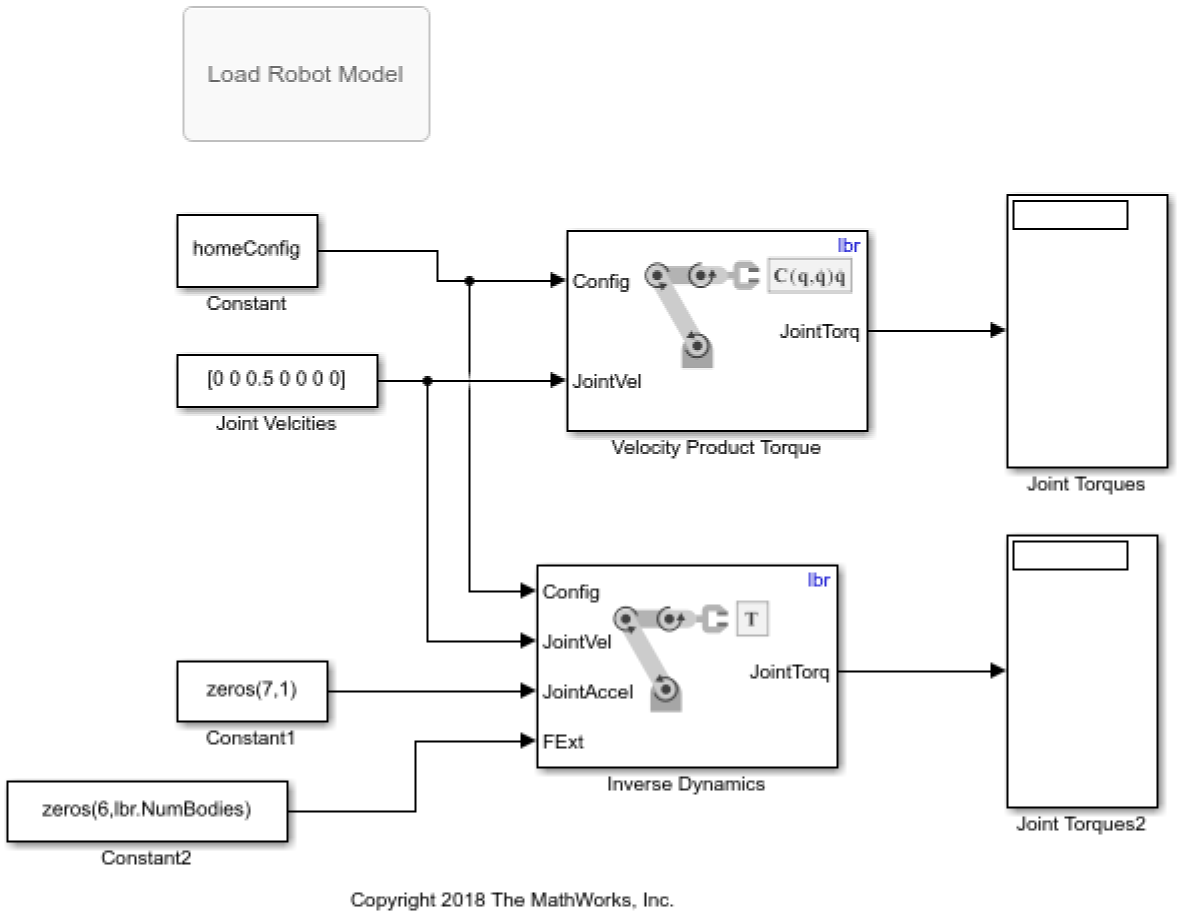
This example shows how to calculate the velocity-induced torques for a robot manipulator by using a `rigidBodyTree` model. In this example, you define a robot model and robot configuration in MATLAB® and pass them to Simulink® to be used with the manipulator algorithm blocks.

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 | Get Jacobian | Gravity Torque | Inverse Dynamics | Joint Space Mass Matrix

Classes
`RigidBodyTree`

Functions
`externalForce` | `homeConfiguration` | `importrobot` | `randomConfiguration`

Related Examples

- “Perform Safe Trajectory Tracking Control Using Robotics Manipulator Blocks”

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

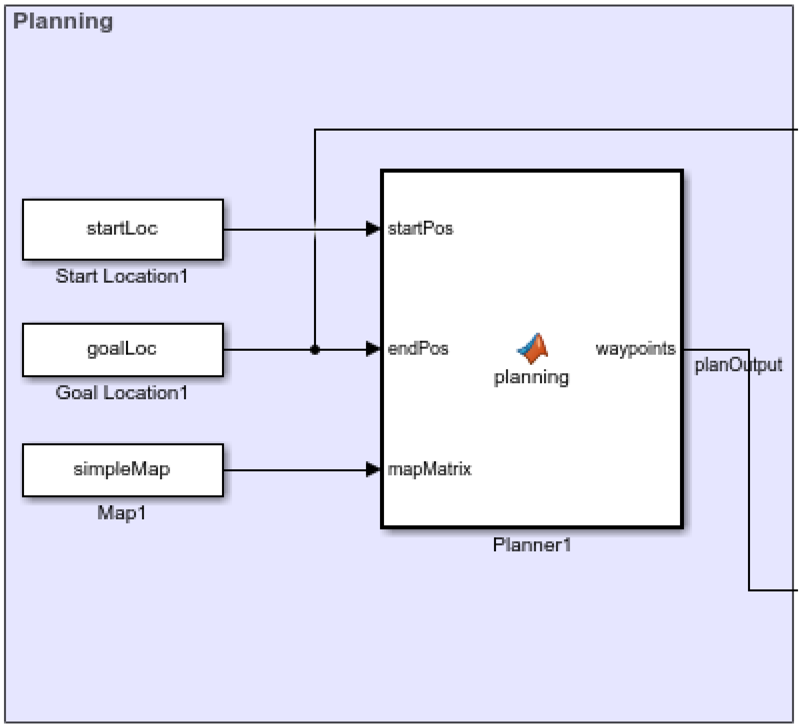
Open the Simulink Model

```
open_system('pathPlanningUnicycleSimulinkModel.slx')
```

The model is composed of three primary parts:

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

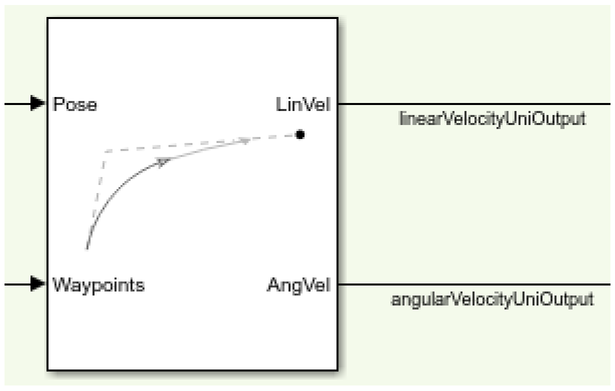
Planning



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.

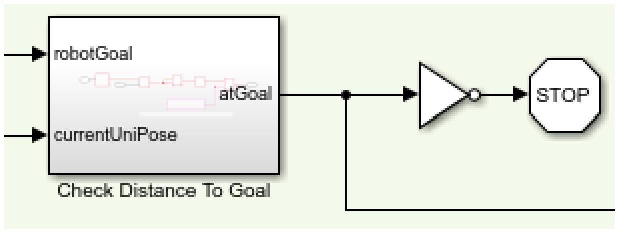
Control

Pure Pursuit



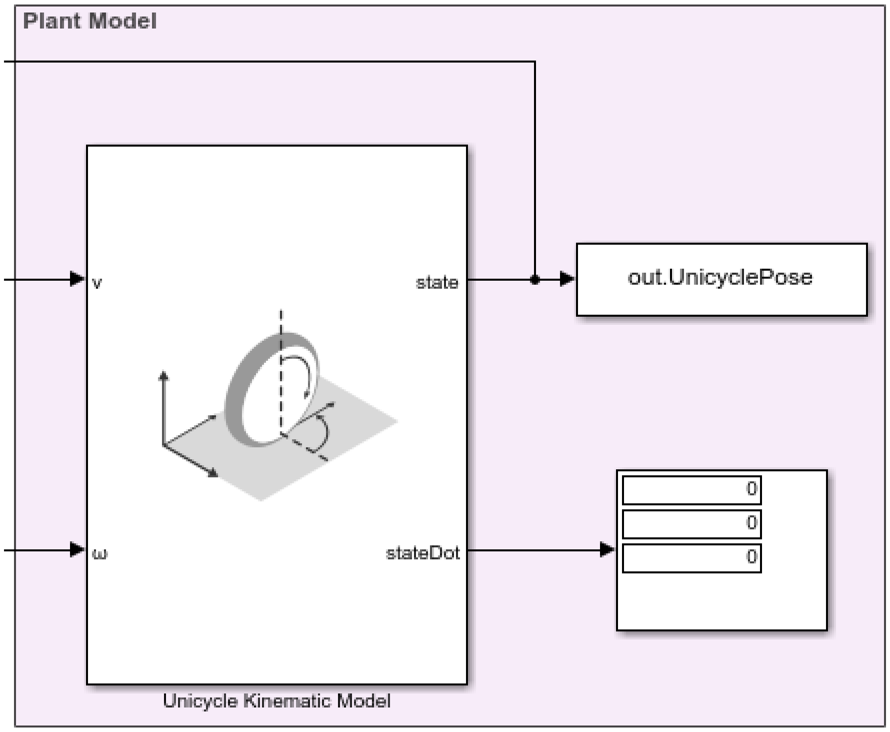
The **Pure Pursuit** controller block generates the linear velocity and angular velocity commands based on the waypoints and the current pose of the robot.

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

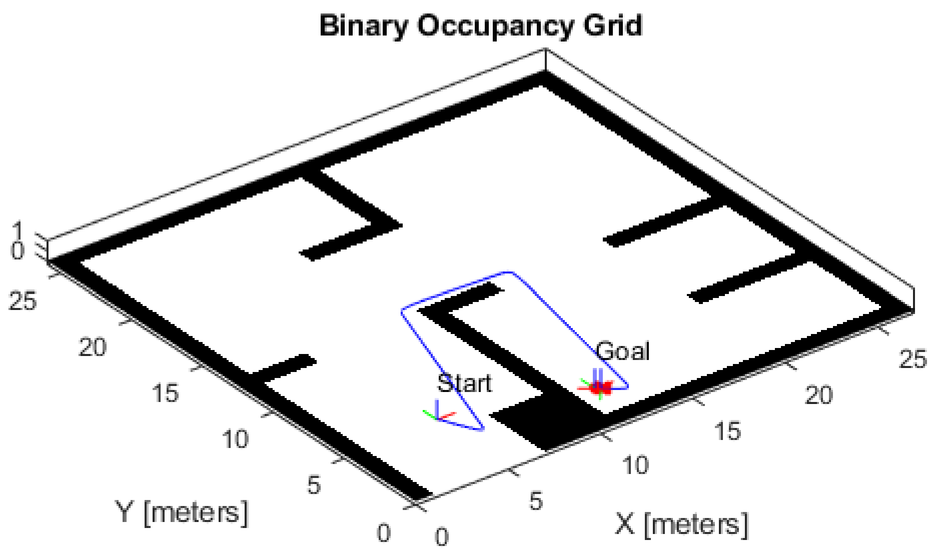
After simulating the model, visualize the robot driving the obstacle-free path in the map.

```
map = binaryOccupancyMap(simpleMap)
map = 
  binaryOccupancyMap with properties:

    GridLocationInWorld: [0 0]
           XWorldLimits: [0 27]
           YWorldLimits: [0 26]
               DataType: 'logical'
           DefaultValue: 0
             Resolution: 1
               GridSize: [26 27]
           XLocalLimits: [0 27]
           YLocalLimits: [0 26]
      GridOriginInLocal: [0 0]
     LocalOriginInWorld: [0 0]
```

```
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

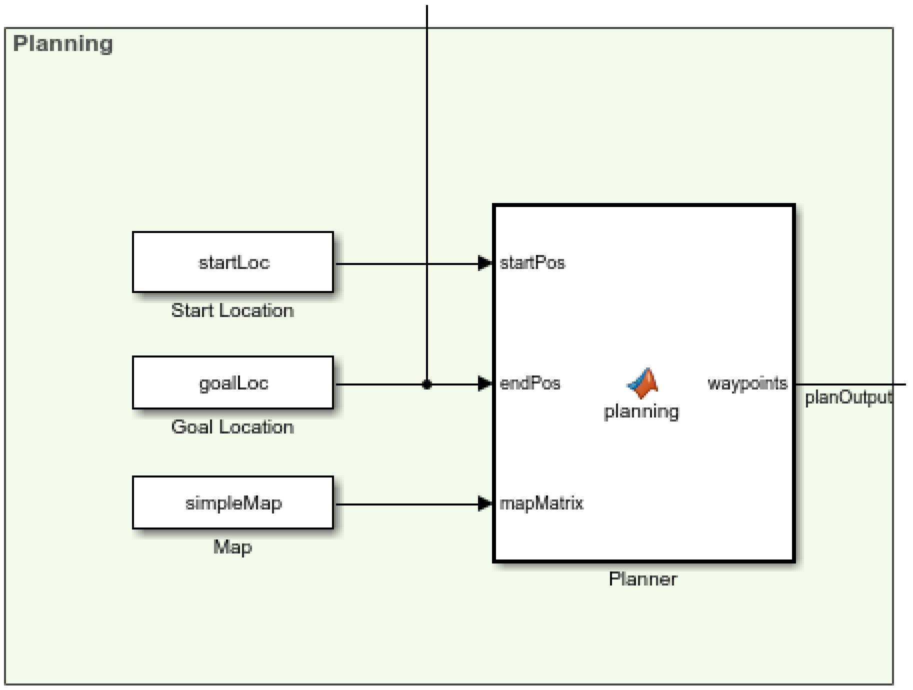
Open the Simulink model.

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

The model is composed of three primary parts:

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

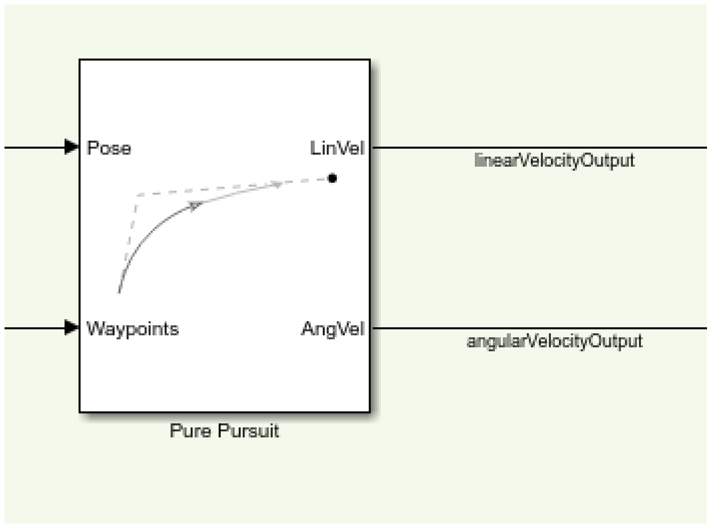
Planning



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 wapoints that the robot follows. The planned waypoints are used downstream by the **Pure Pursuit** controller block.

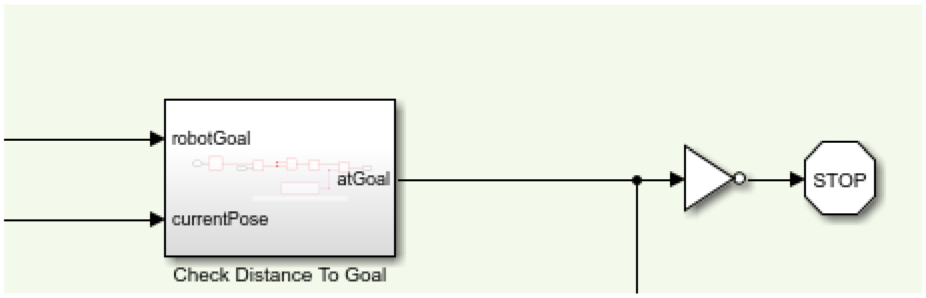
Control

Pure Pursuit



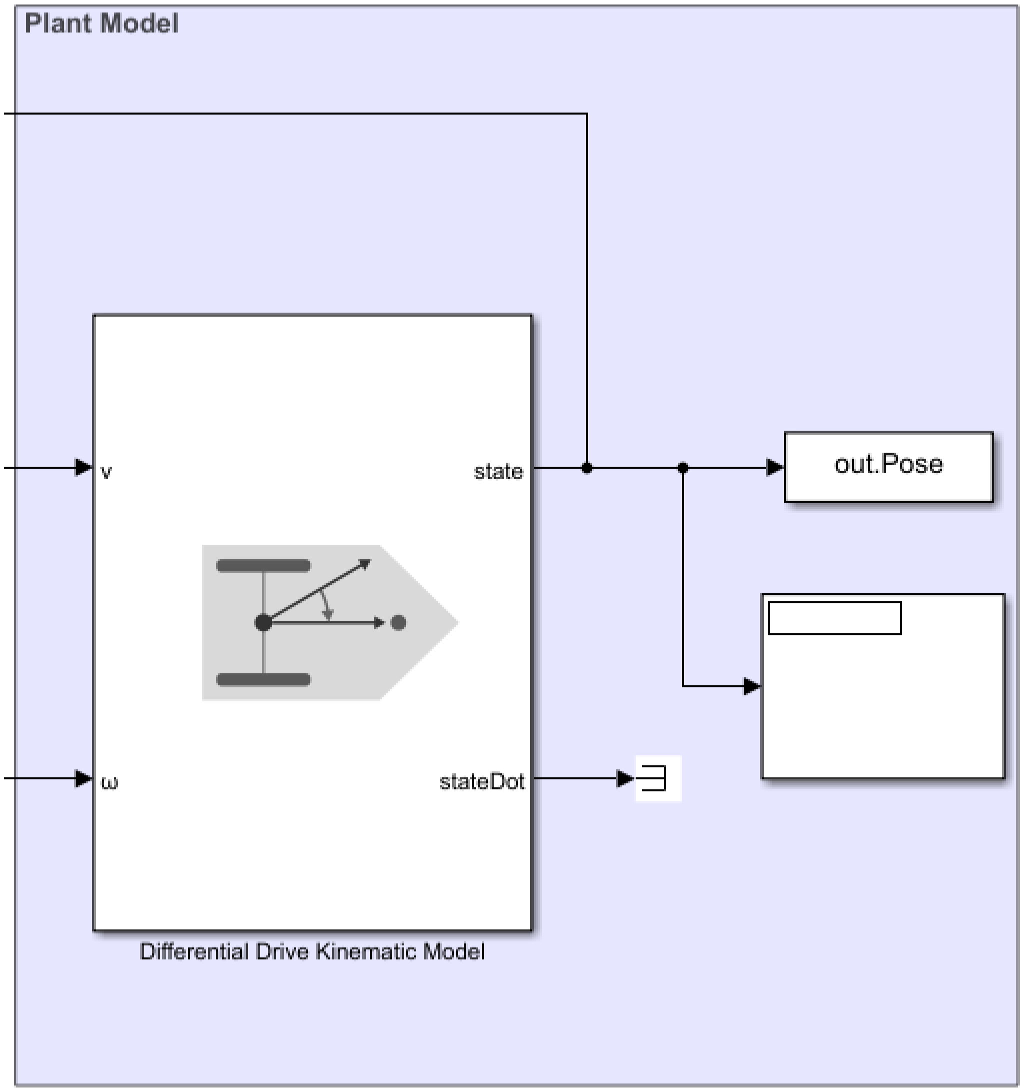
The **Pure Pursuit** controller block generates the linear velocity and angular velocity commands based on the waypoints and the current pose of the robot.

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.

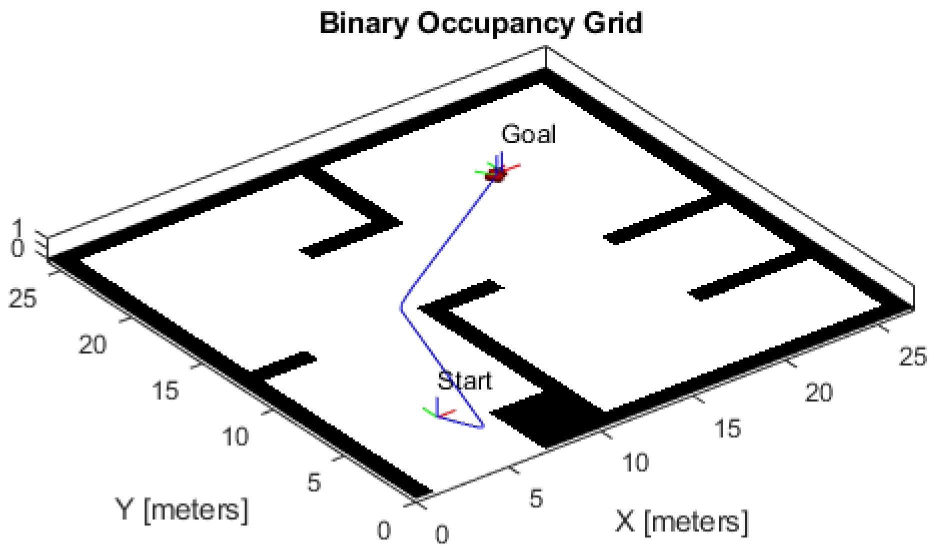
Run the Model

```
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 motion on an obstacle free path between two random locations on a given offline map.

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

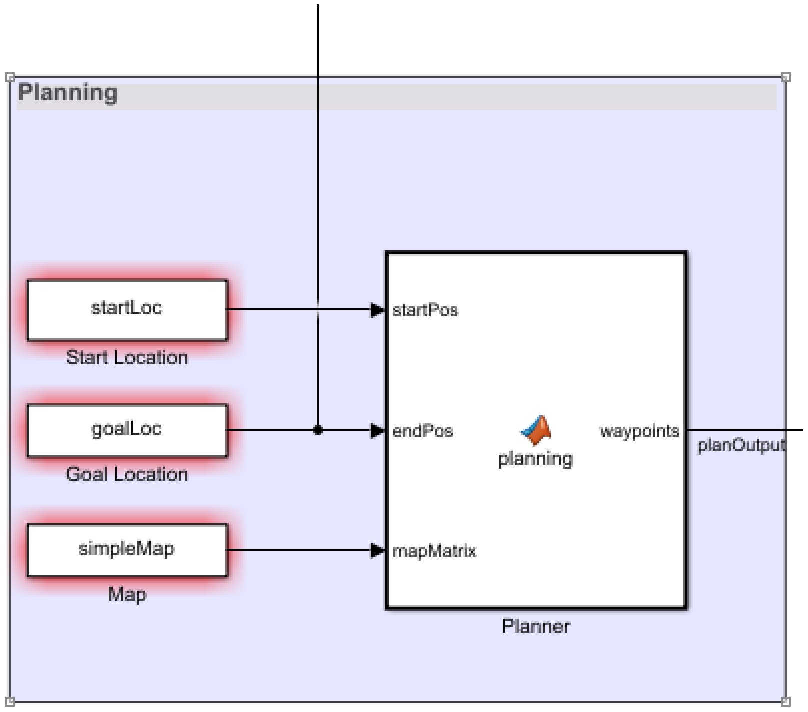
```
open_system('pathPlanningBicycleSimulinkModel.slx')
```

Model Overview

The model is composed of four primary operations :

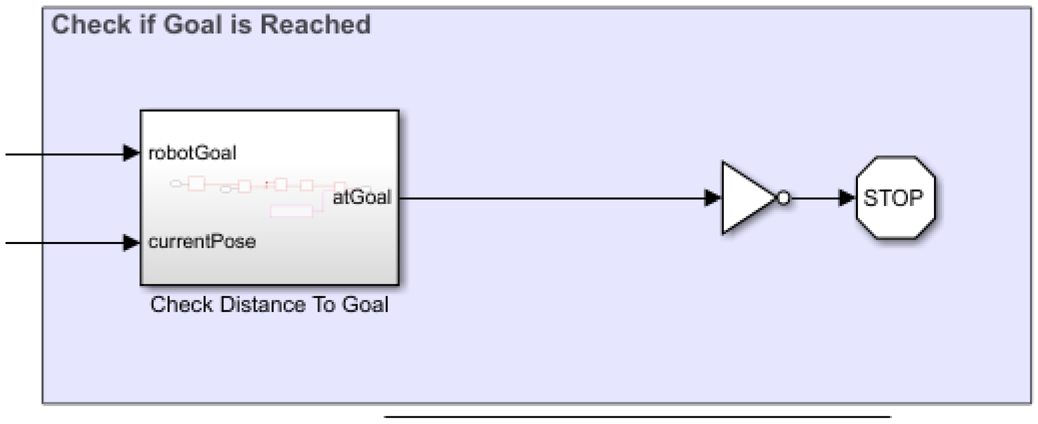
- **Planning**
- **Check if Goal is Reached**
- **Controller**
- **Bicycle Kinematic Model**

Planning



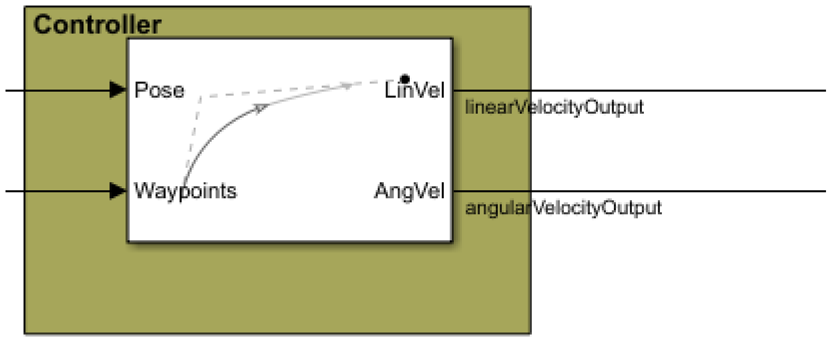
This blocks takes a start location, a goal location and map as inputs and outputs an array of wapoints which robot will follow. The planned waypoints are used downstream by the Pure pursuit controller which outputs the angular and linear velocities given the current pose of the robot and the planned waypoints as inputs.

Check if Goal is Reached



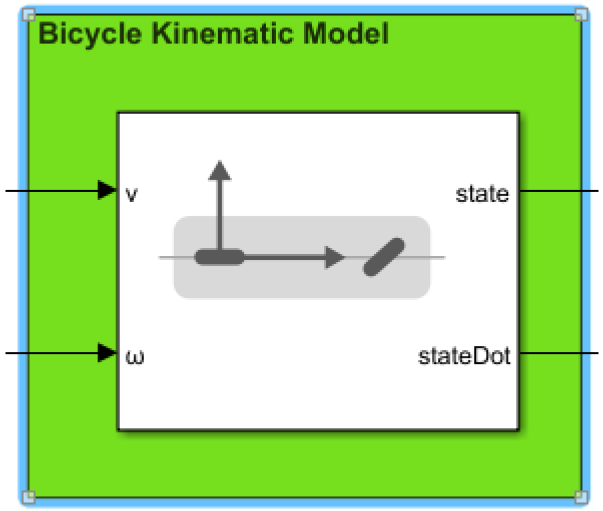
If robot has reached the goal location, the simulaion is stopped.

Controller



Controller outputs the linear velocity and angular velocity based on the waypoints and the robot's current pose. The Pure Pursuit Controller block is used for the same.

Bicycle Kinematic Model



Bicycle Kinematic Model creates a vehicle model to simulate simplified vehicle dynamics. It takes linear and angular velocities as input from the Pure Pursuit Controller block, and outputs i.e. state and stateDot which are the robot's state and the robot's state's time derivative, respectively. The robot' state is also used to calculate the the distance to goal and check if the robot has reached the goal location.

Run the Model

To simulate the model

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

Visualize The Motion of Robot

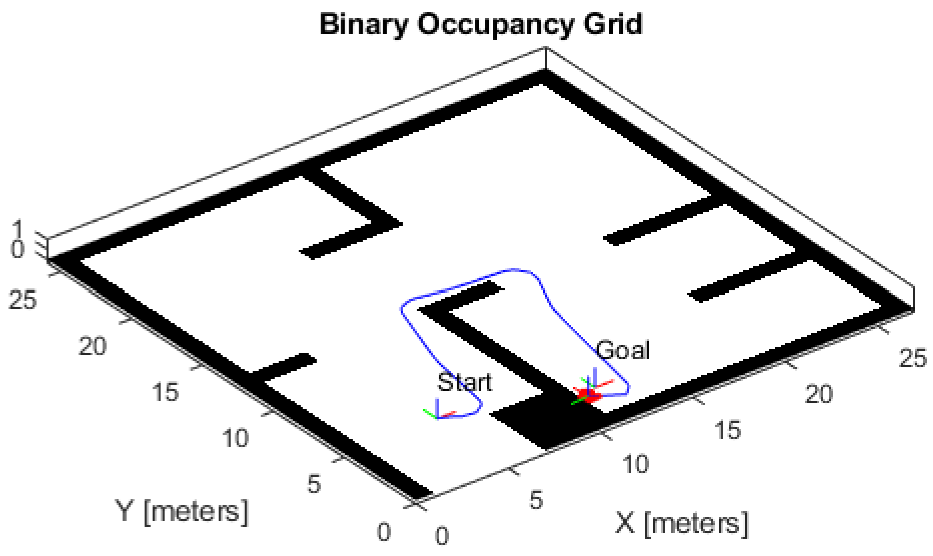
To see the poses :

```
map = binaryOccupancyMap(simpleMap)
map = 
  binaryOccupancyMap with properties:

    GridLocationInWorld: [0 0]
           XWorldLimits: [0 27]
           YWorldLimits: [0 26]
               DataType: 'logical'
           DefaultValue: 0
             Resolution: 1
               GridSize: [26 27]
           XLocalLimits: [0 27]
           YLocalLimits: [0 26]
      GridOriginInLocal: [0 0]
     LocalOriginInWorld: [0 0]
```

```
robotPose = simulation.BicyclePose
```

```
robotPose = 304×3

    5.0000    5.0000         0
    5.0002    5.0000   -0.0002
    5.0012    5.0000   -0.0012
    5.0062    5.0000   -0.0062
    5.0313    4.9995   -0.0313
    5.1563    4.9877   -0.1569
    5.7068    4.7074   -0.7849
    5.8197    4.6015   -0.6638
    5.9427    4.5193   -0.5157
    6.6589    4.4144    0.2249
      ⋮
```

```
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.

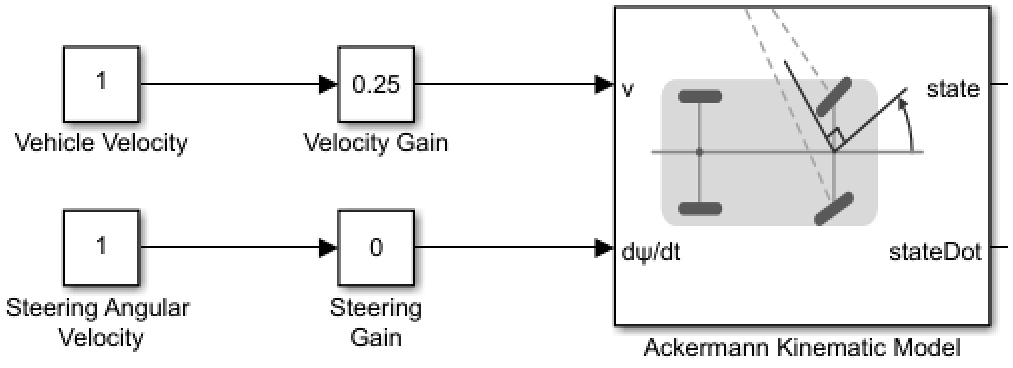
```
open_system("plotAckermannDriveSimulinkModel.slx");
```

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

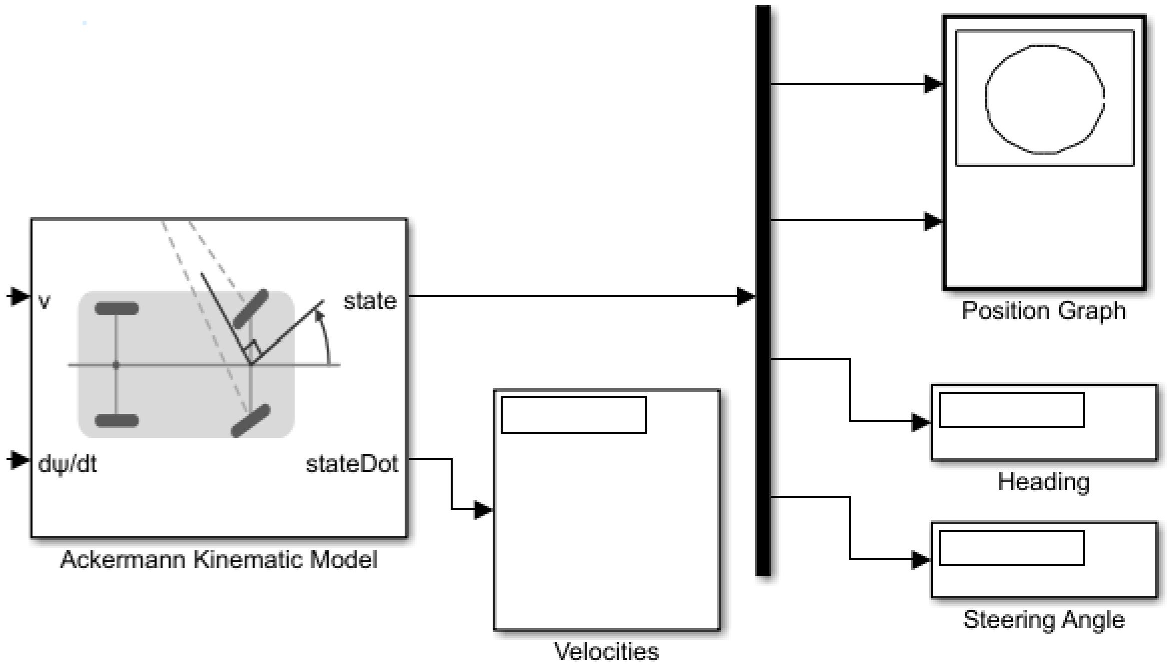
The Ackermann Kinematic Model block takes two inputs, vehicle velocity and steering angular velocity. This model uses **Slider Gain** blocks to change the inputs.



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

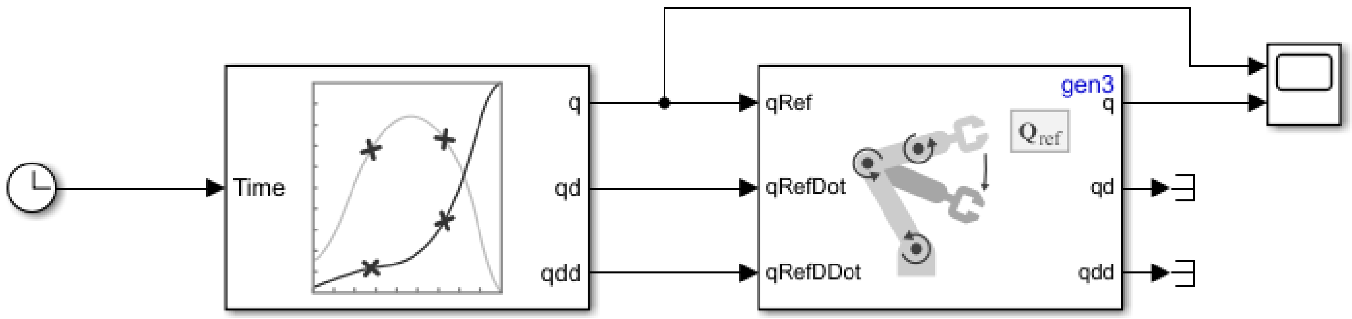
This example shows how to use a **Joint Space Motion Model** block to follow a trajectory in Simulink.

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");
```

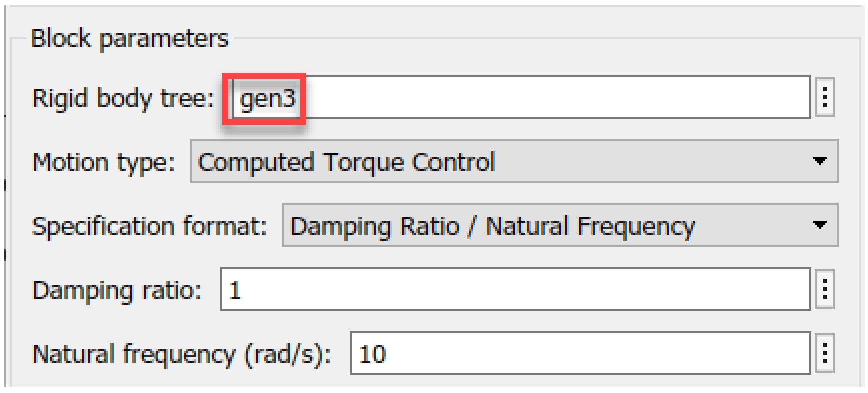


Plan Trajectory

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.

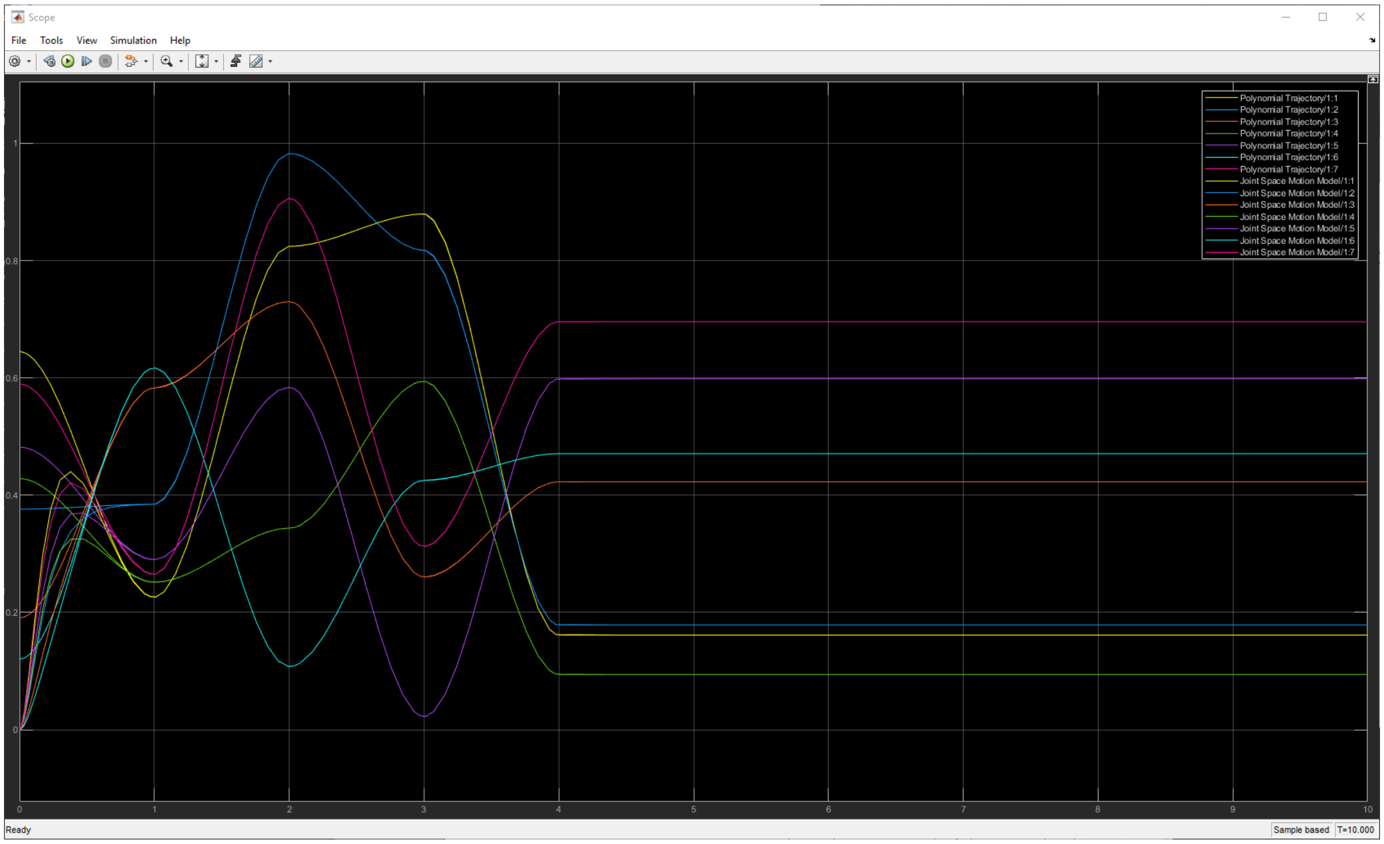


Block parameters

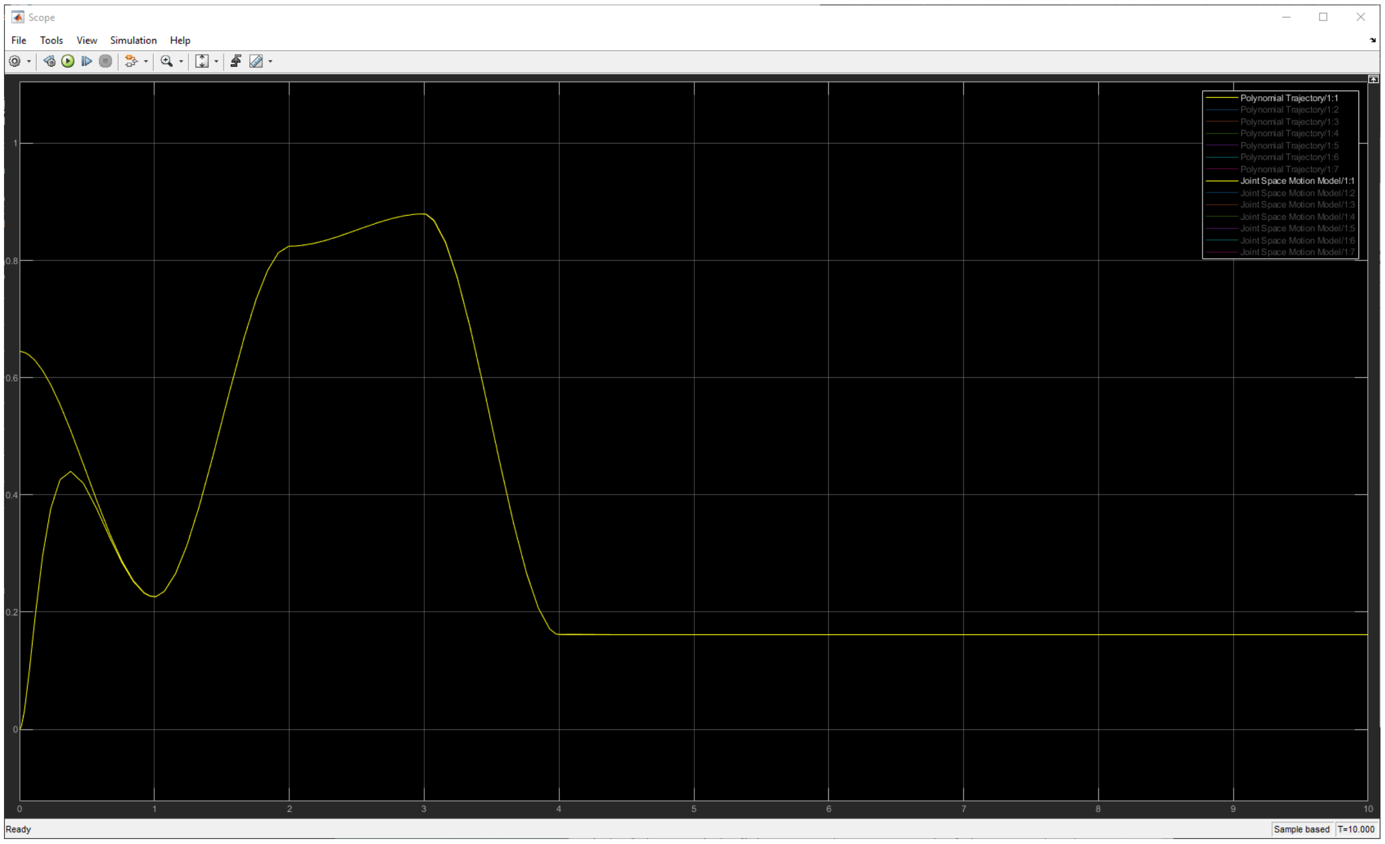
Rigid body tree:	gen3
Motion type:	Computed Torque Control
Specification format:	Damping Ratio / Natural Frequency
Damping ratio:	1
Natural frequency (rad/s):	10

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.



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])
```

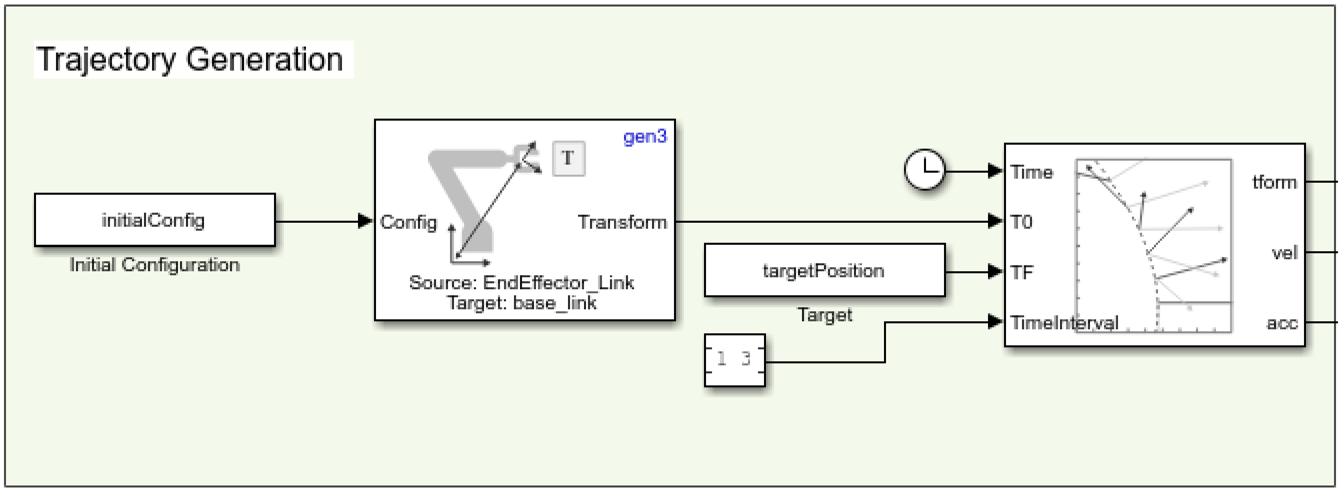
```
targetPosition = 4×4

    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")
```

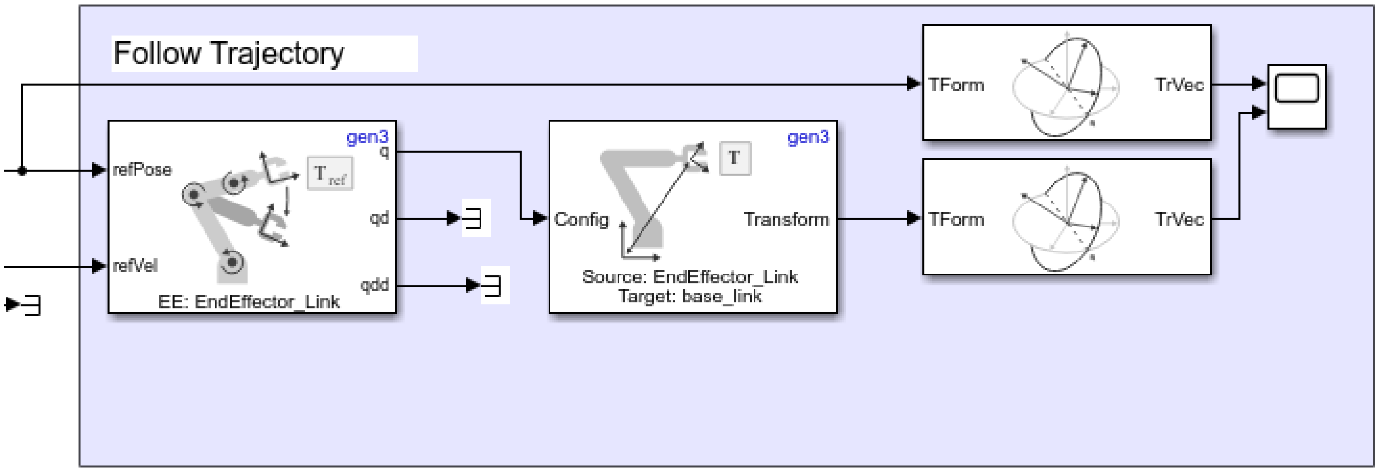
Trajectory Generation

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.



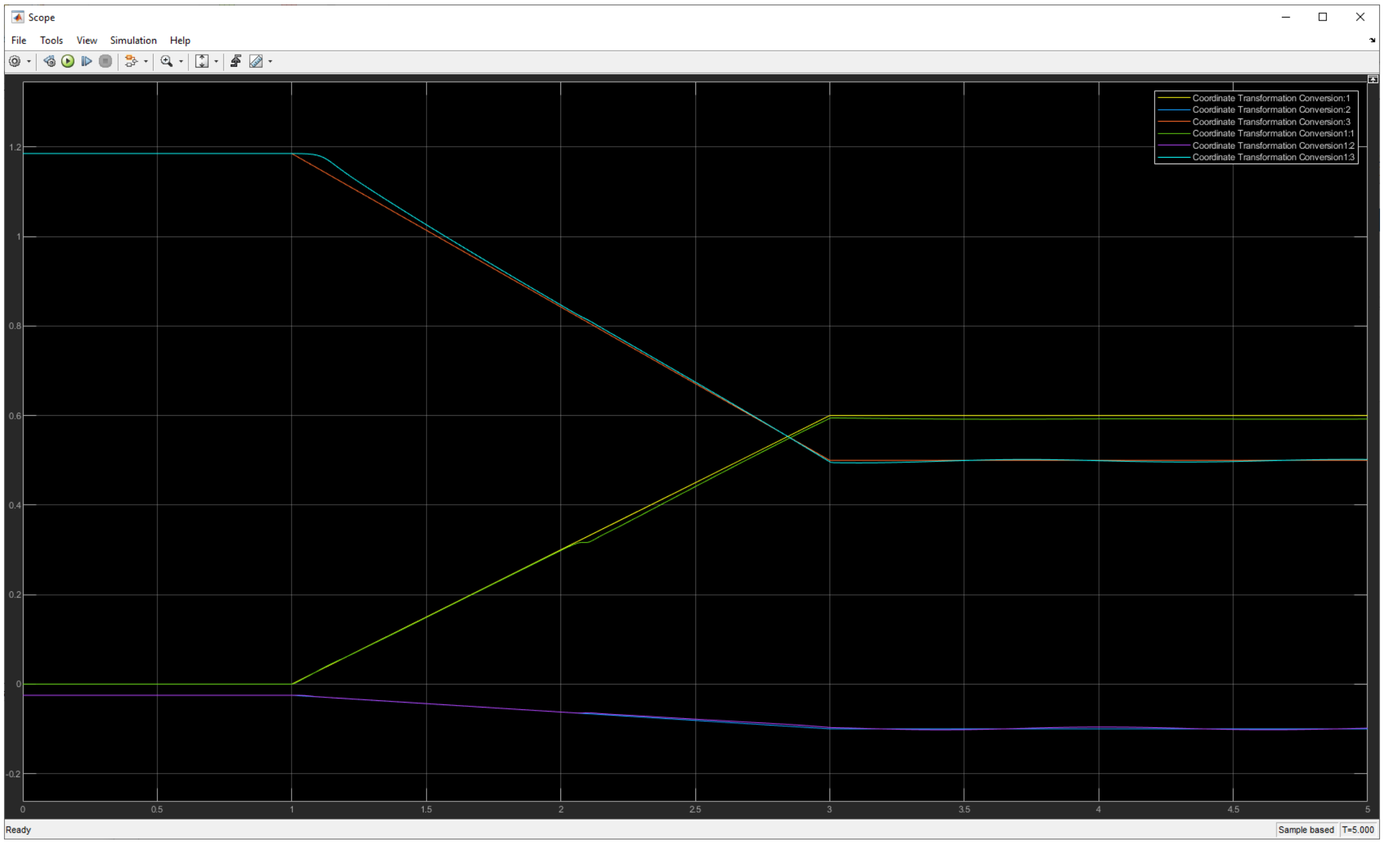
Follow Trajectory

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
Ready
Sample based
T=5.000