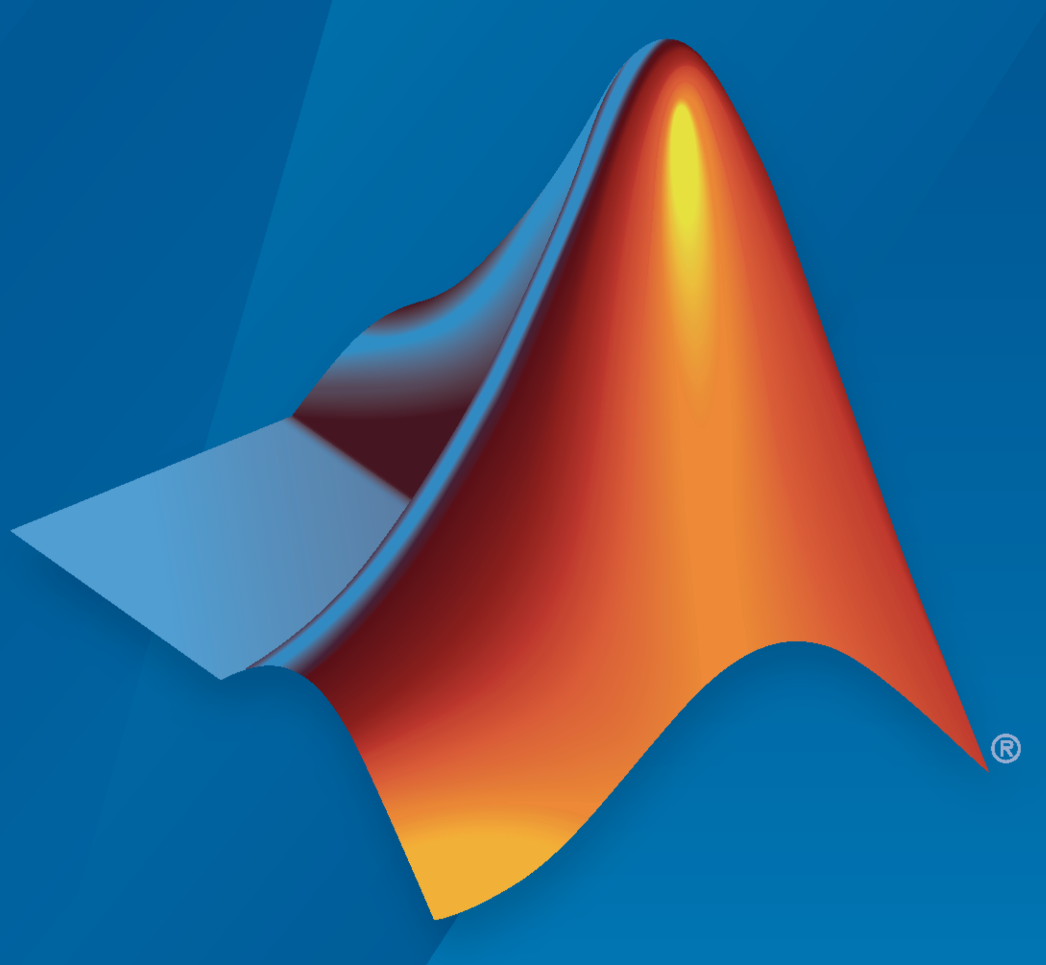
Robotics System Toolbox™

Reference



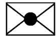
MATLAB® & SIMULINK®

R2023a

How to Contact MathWorks

Latest news:	www.mathworks.com
Sales and services:	www.mathworks.com/sales_and_services
User community:	www.mathworks.com/matlabcentral
Technical support:	www.mathworks.com/support/contact_us

Phone: 508-647-7000



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

Robotics System Toolbox™ Reference

© COPYRIGHT 2015–2023 by The MathWorks, Inc.

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

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

Trademarks

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

Patents

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

Revision History

March 2015	Online only	New for Version 1.0 (R2015a)
September 2015	Online only	Revised for Version 1.1 (R2015b)
October 2015	Online only	Rereleased for Version 1.0.1 (Release 2015aSP1)
March 2016	Online only	Revised for Version 1.2 (R2016a)
September 2016	Online only	Revised for Version 1.3 (R2016b)
March 2017	Online only	Revised for Version 1.4 (R2017a)
September 2017	Online only	Revised for Version 1.5 (R2017b)
March 2018	Online only	Revised for Version 2.0 (R2018a)
September 2018	Online only	Revised for Version 2.1 (R2018b)
March 2019	Online only	Revised for Version 2.2 (R2019a)
September 2019	Online only	Revised for Version 3.0 (R2019b)
March 2020	Online only	Revised for Version 3.1 (R2020a)
September 2020	Online only	Revised for Version 3.2 (R2020b)
March 2021	Online only	Revised for Version 3.3 (R2021a)
September 2021	Online only	Revised for Version 3.4 (R2021b)
March 2022	Online only	Revised for Version 4.0 (R2022a)
September 2022	Online only	Revised for Version 4.1 (R2022b)
March 2023	Online only	Revised for Version 4.2 (R2023a)

Contents

1 Classes

2 Functions

3 Methods

4 Blocks

5 Apps

. . . . 5-2

1

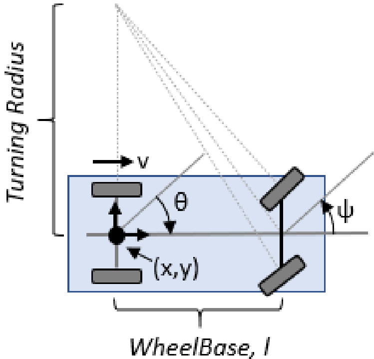
Classes

ackermannKinematics

Car-like steering vehicle model

Description

`ackermannKinematics` creates a car-like vehicle model that uses Ackermann steering. This model represents a vehicle with two axles separated by the distance, Wheelbase. The state of the vehicle is defined as a four-element vector, [*x y theta psi*], with a global *xy*-position, specified in meters. The *xy*-position is located at the middle of the rear axle. The vehicle heading, *theta*, and steering angle, *psi* are specified in radians. The vehicle heading is defined at the center of the rear axle. Angles are given in radians. To compute the time derivative states for the model, use the `derivative` function with input steering commands and the current robot state.



Creation

Syntax

`kinematicModel = ackermannKinematics`

`kinematicModel = ackermannKinematics(Name,Value)`

Description

`kinematicModel = ackermannKinematics` creates an Ackermann kinematic model object with default property values.

`kinematicModel = ackermannKinematics(Name,Value)` sets additional properties to the specified values. You can specify multiple properties in any order.

Properties

`WheelBase` — Distance between front and rear axles
1 (default) | positive numeric scalar

The wheel base refers to the distance between the front and rear axles, specified in meters.

VehicleSpeedRange — Range of vehicle wheel speeds
[-Inf Inf] (default) | two-element vector

The vehicle speed range is a two-element vector that provides the minimum and maximum vehicle speeds, [*MinSpeed MaxSpeed*], specified in meters per second.

Object Functions

derivative   Time derivative of vehicle state

Examples

Simulate Different Kinematic Models for Mobile Robots

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

Define Mobile Robots with Kinematic Constraints

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

Unicycle Kinematic Model

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

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

Differential-Drive Kinematic Model

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

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

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

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

Bicycle Kinematic Model

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

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

Other Models

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

```
carLike = ackermannKinematics;
```

Set up Simulation Parameters

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

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

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

Create a Vehicle Controller

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

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

Simulate the Models Using an ODE Solver

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

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

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

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

```
% Compute trajectories for each kinematic model under motion control
[tUnicycle,unicyclePose] = ode45(@(t,y)derivative(unicycle,y,exampleHelperMobileRobotController(c
```

```
[tBicycle,bicyclePose] = ode45(@(t,y)derivative(bicycle,y,exampleHelperMobileRobotController(con
[tDiffDrive,diffDrivePose] = ode45(@(t,y)derivative(diffDrive,y,exampleHelperMobileRobotControlle
```

Plot Results

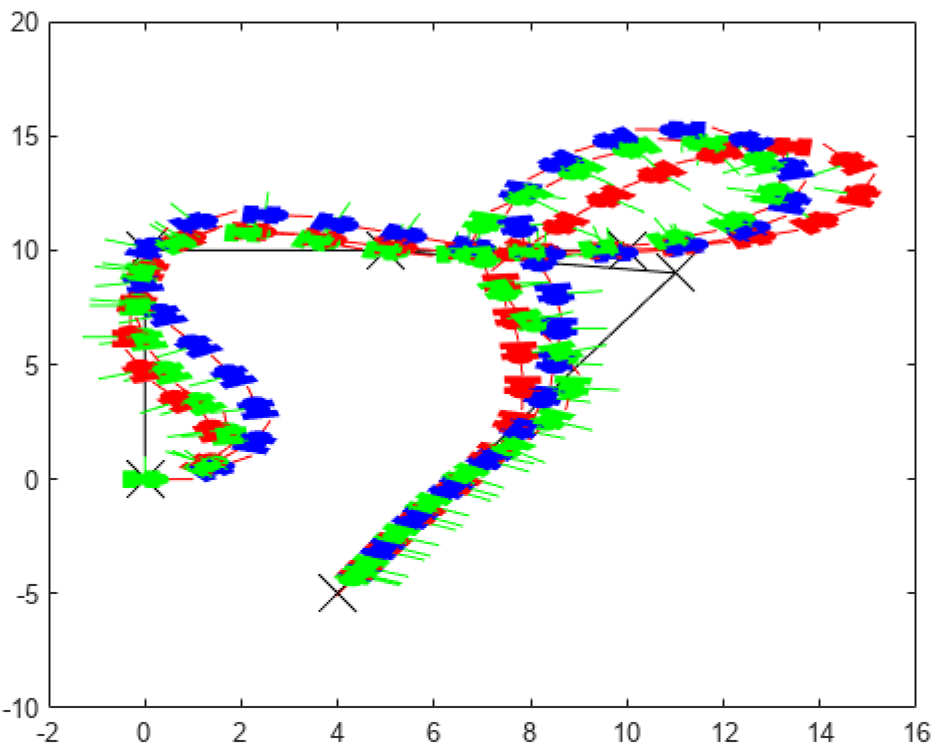
The results of the ODE solver can be easily viewed on a single plot using `plotTransforms` to visualize the results of all trajectories at once.

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

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

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

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

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

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



Simulate Ackermann Kinematic Model with Steering Angle Constraints

Simulate a mobile robot model that uses Ackermann steering with constraints on its steering angle. During simulation, the model maintains maximum steering angle after it reaches the steering limit. To see the effect of steering saturation, you compare the trajectory of two robots, one with the constraints on the steering angle and the other without any steering constraints.

Define the Model

Define the Ackermann kinematic model. In this car-like model, the front wheels are a given distance apart. To ensure that they turn on concentric circles, the wheels have different steering angles. While turning, the front wheels receive the steering input as rate of change of steering angle.

```
carLike = ackermannKinematics;
```

Set Up Simulation Parameters

Set the mobile robot to follow a constant linear velocity and receive a constant steering rate as input. Simulate the constrained robot for a longer period to demonstrate steering saturation.

```
velo = 5;    % Constant linear velocity
psidot = 1;  % Constant left steering rate

% Define the total time and sample rate
sampleTime = 0.05;                  % Sample time [s]
```

```
timeEnd1 = 1.5;                     % Simulation end time for unconstrained robot
timeEnd2 = 10;                      % Simulation end time for constrained robot
tVec1 = 0:sampleTime:timeEnd1;      % Time array for unconstrained robot
tVec2 = 0:sampleTime:timeEnd2;      % Time array for constrained robot

initPose = [0;0;0;0];               % Initial pose (x y theta phi)
```

Create Options Structure for ODE Solver

In this example, you pass an `options` structure as argument to the ODE solver. The `options` structure contains the information about the steering angle limit. To create the `options` structure, use the `Events` option of `odeset` and the created event function, `detectSteeringSaturation`. `detectSteeringSaturation` sets the maximum steering angle to 45 degrees.

For a description of how to define `detectSteeringSaturation`, see **Define Event Function** at the end of this example.

```
options = odeset('Events',@detectSteeringSaturation);
```

Simulate Model Using ODE Solver

Next, you use the `derivative` function and an ODE solver, `ode45`, to solve the model and generate the solution.

```
% Simulate the unconstrained robot
[t1,pose1] = ode45(@(t,y)derivative(carLike,y,[velo psidot]),tVec1,initPose);

% Simulate the constrained robot
[t2,pose2,te,ye,ie] = ode45(@(t,y)derivative(carLike,y,[velo psidot]),tVec2,initPose,options);
```

Detect Steering Saturation

When the model reaches the steering limit, it registers a timestamp of the event. The time it took to reach the limit is stored in `te`.

```
if te < timeEnd2
    str1 = "Steering angle limit was reached at ";
    str2 = " seconds";
    comp = str1 + te + str2;
    disp(comp)
end
```

```
Steering angle limit was reached at 0.785 seconds
```

Simulate Constrained Robot with New Initial Conditions

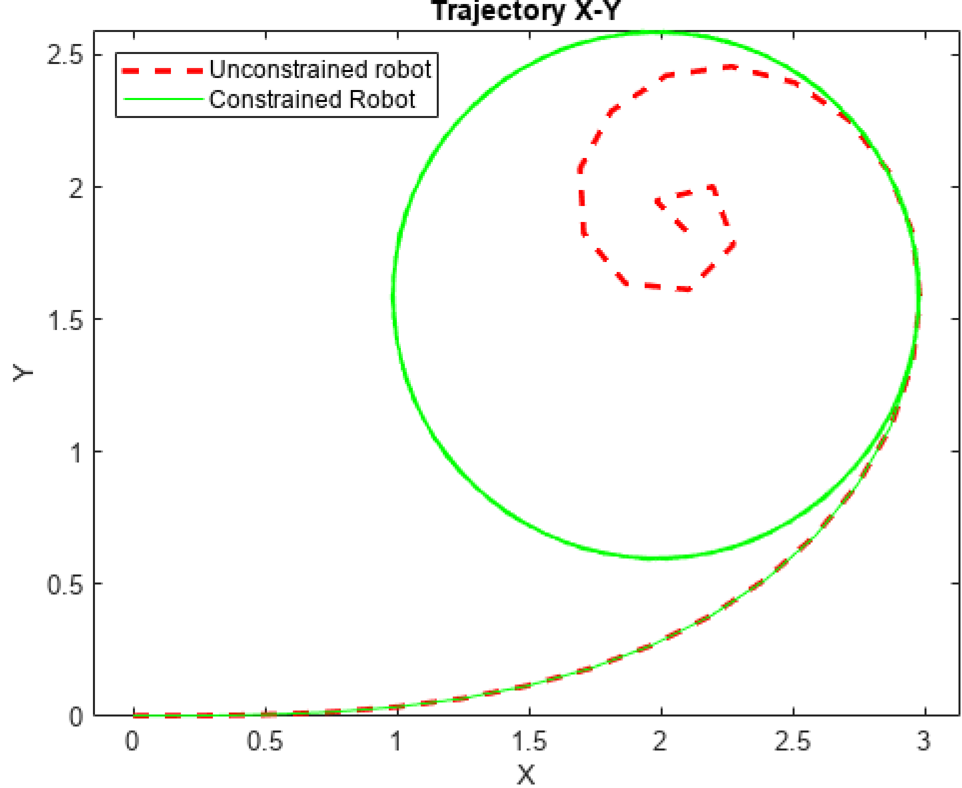
Now use the state of the constrained robot before termination of integration as initial condition for the second simulation. Modify the input vector to represent steering saturation, that is, set the steering rate to zero.

```
saturatedPsiDot = 0;            % Steering rate after saturation
cmds = [velo saturatedPsiDot];  % Command vector
tVec3 = te:sampleTime:timeEnd2; % Time vector
pose3 = pose2(length(pose2),:);
[t3,pose3,te3,ye3,ie3] = ode45(@(t,y)derivative(carLike,y,cmds), tVec3,pose3, options);
```

Plot the Results

Plot the trajectory of the robot using `plot` and the data stored in `pose.`

```
figure(1)
plot(pose1(:,1),pose1(:,2),'--r','LineWidth',2);
hold on;
plot([pose2(:,1); pose3(:,1)],[pose2(:,2);pose3(:,2)],'g');
title('Trajectory X-Y')
xlabel('X')
ylabel('Y')
legend('Unconstrained robot','Constrained Robot','Location','northwest')
axis equal
```



The unconstrained robot follows a spiral trajectory with decreasing radius of curvature while the constrained robot follows a circular trajectory with constant radius of curvature after the steering limit is reached.

Define Event Function

Set the event function such that integration terminates when 4th state, theta, is equal to maximum steering angle.

```
function [state,isterminal,direction] = detectSteeringSaturation(t,y)
  maxSteerAngle = 0.785;                % Maximum steering angle (pi/4 radians)
  state(4) = (y(4) - maxSteerAngle);    % Saturation event occurs when the 4th state, theta, is ec
  isterminal(4) = 1;                    % Integration is terminated when event occurs
  direction(4) = 0;                     % Bidirectional termination
```

```
end
```

Version History

Introduced in R2019b

References

[1] Lynch, Kevin M., and Frank C. Park. *Modern Robotics: Mechanics, Planning, and Control* 1st ed. Cambridge, MA: Cambridge University Press, 2017.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

Classes

`bicycleKinematics` | `unicycleKinematics` | `differentialDriveKinematics`

Blocks

Ackermann Kinematic Model

Functions

`derivative`

Topics

“Mobile Robot Kinematics Equations”

analyticalInverseKinematics

Solve closed-form inverse kinematics

Description

The `analyticalInverseKinematics` object generates functions that computes all closed-form solutions for inverse kinematics (IK) for serial-chain manipulators using an approach based on the Pieper method [1]. The object generates a custom function to find multiple distinct joint configurations that achieve the desired end-effector pose for a kinematic group of a rigid body tree robot model given that the specified kinematic group represents an applicable six-DOF serial manipulator with a wrist and compatible kinematic parameters. A wrist is defined as three consecutive revolute joints with orthogonal axes.

These are the key elements of the solver:

- **Robot model** — Rigid body tree model that defines the kinematics of the robot. Specify this model as a `rigidBodyTree` object when creating the solver.
- **Kinematic group** — Base and end-effector body names for a six-DOF serial chain that is part of the robot model. To set this parameter, use the `showdetails` function.
- **Kinematic group type** — Classification of joints connecting base to end effector.

To see all possible supported kinematic groups for your robot, use the `showdetails` object function. To set a specific group from the list, click the **Use this kinematic group** link for a kinematic group in the returned list.

To calculate inverse kinematics for a specific kinematic group, use the `generateIKFunction` object function. To ensure your robot model and kinematic group are compatible, check the IsValidGroupForIK property after selecting a kinematic group.

To generate numeric solutions, use the `inverseKinematics` and `generalizedInverseKinematics` objects.

Creation

Syntax

```
analyticalIK = analyticalInverseKinematics(robotRBT)
analyticalIK = analyticalInverseKinematics(
robotRBT,'KinematicGroup',kinGroup)
```

Description

`analyticalIK = analyticalInverseKinematics(robotRBT)` creates an analytical inverse kinematics solver for a rigid body tree robot model, specified as a `rigidBodyTree` object. The end effector is the final body listed in the Bodies property of the robot model. The `robotRBT` argument sets the RigidBodyTree property.

`analyticalIK = analyticalInverseKinematics(`
`robotRBT,'KinematicGroup',kinGroup)` sets the KinematicGroup property to the `kinGroup` argument, specified as a structure.

Properties

`RigidBodyTree` — Rigid body tree robot model

`rigidBodyTree` object

Rigid body tree robot model, specified as a `rigidBodyTree` object. To use a provided robot model, see `loadrobot`. To import Unified Robot Description Format (URDF) models, see the `importrobot` function.

`KinematicGroup` — Base and end-effector body names

structure

Base and end-effector body names, specified as a structure. The structure contains these fields:

- `BaseName` — Name of the body in the robot model stored in the `RigidBodyTree` property that represents the base of the kinematic group. The default value is the base of the `RigidBodyTree` property.
- `EndEffectorBodyName` — Name of the body in the robot model stored in the `RigidBodyTree` property that represents the end of the kinematic group. The default value is the last body in the Bodies property of the robot model.

A valid kinematic group must represent a six-DOF serial chain with a wrist and a contain joint types defined by `KinematicGroupType` as `'`*`XXX`*`SSS'`. A wrist is defined as three consecutive revolute joints with orthogonal axes with compatible kinematic parameters and is represented as `SSS`. *XXX* is either three revolute joints RRR or another wrist `SSS`. If the kinematic group type contains a prismatic joint, `P`, the kinematic group is invalid for use with this solver. To check if your kinematic group is valid for this solver, see the IsValidGroupForIK property.

When created, the object automatically selects a kinematic group from the robot model, but other options may be available. To see valid kinematic groups for your model, use the `showdetails` object function.

Example: `struct("BaseName","base","EndEffectoryBodyName","tool0")`

Data Types: `struct`

`KinematicGroupType` — Classification of the kinematic group

character vector

This property is read-only.

Classification of the kinematic group, stored as a character vector. Each character specifies the joint type of each rigid body from the base to the end effector of the kinematic group. These are the options for the characters:

- `R` — Revolute joint that does not form a wrist
- `P` — Prismatic joint
- `S` — Revolute joint of a wrist

Note A wrist or spherical joint is comprised of three consecutive revolute joints with orthogonal axes.

To qualify as a valid kinematic group for the `analyticalInverseKinematics` object, the kinematic group type must be `'`*`XXX`*`SSS'`, where *`XXX`* can be either `RRR` or `SSS`. If the kinematic group type contains a prismatic joint, `P`, the kinematic group is invalid for use with this solver. To check if your kinematic group is valid for this solver, see the IsValidGroupForIK property.

When created, the object automatically selects a kinematic group from the robot model, but other options may be available. To see valid kinematic groups for your model, use the `showdetails` object function.

Example: `'RRRSSS'`

Data Types: `char`

`KinematicGroupConfigIdx` — Mapping of IK solution configuration to rigid body tree configuration

six-element vector

This property is read-only.

Mapping of IK solution configuration to rigid body tree configuration, specified as a six-element vector. This mapping converts the indices of the IK solution that is output from the `generateIKFunction` function to the indices for the robot model stored in the RigidBodyTree property.

Example: `[1 2 3 4 5 6]`

Data Types: `double`

`IsValidGroupForIK` — Indication of whether closed-form solution is possible

`1` (`true`) | `0` (`false`)

This property is read-only.

Indication of whether a closed-form solution is possible, stored as a logical, `1` (`true` or `0` (`false`). When this property is `false`, the `generateIKFunction` function cannot generate an IK solver for the current kinematic group. Use the `showdetails` object function to check if any valid groups exist. To select a valid group, specify a different base or end effector to the `KinematicGroup` property, or change kinematic parameters of your robot model stored in the RigidBodyTree property.

To qualify as a valid kinematic group for the `analyticalInverseKinematics` object, the kinematic group type must be `'`*`XXX`*`SSS'`, where *`XXX`* can be either `RRR` or `SSS`. If the kinematic group type contains a prismatic joint, `P`, the kinematic group is invalid for use with this solver. To check if your kinematic group is valid for this solver, see the IsValidGroupForIK property.

Data Types: `logical`

Object Functions

generateIKFunction	Generate function for closed-form inverse kinematics
showdetails	Display overview of available kinematic groups

Examples

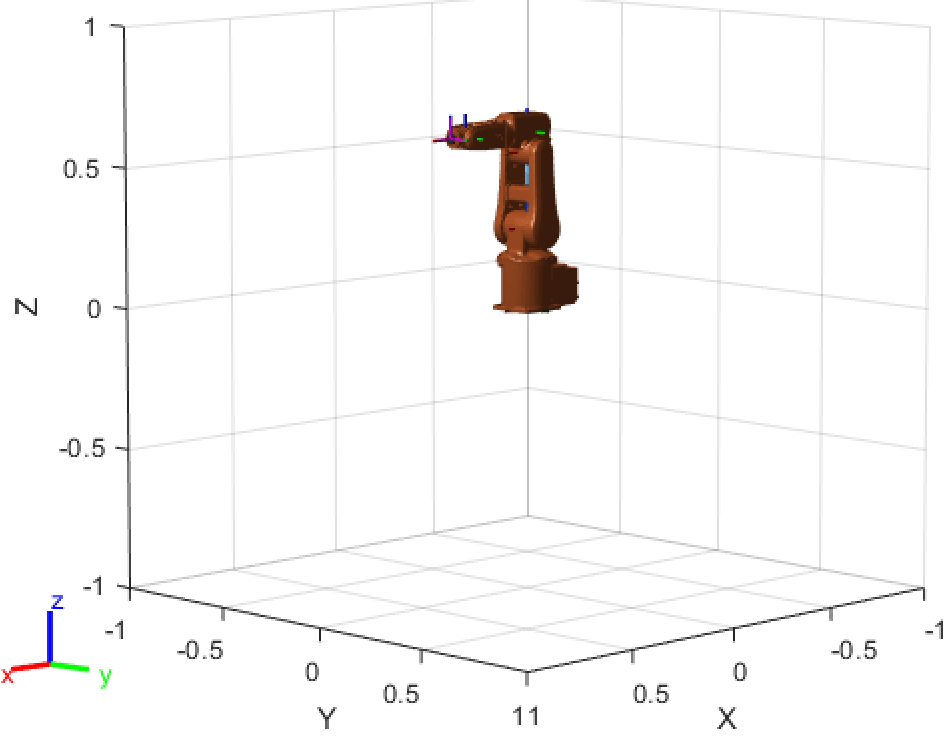
Solve Analytical Inverse Kinematics for Robot Manipulator

Generate closed-form inverse kinematics (IK) solutions for a desired end effector. Load the provided robot model and inspect details about the feasible kinematic groups of base and end-effector bodies. Generate a function for your desired kinematic group. Inspect the various configurations for a specific end-effector pose.

Robot Model

Load the ABB IRB 120 robot model into the workspace. Display the model.

```
robot = loadrobot('abbIrb120','DataFormat','row');
show(robot);
```



Analytical IK

Create the analytical IK solver. Show details for the robot model, which lists the different kinematic groups available for closed-form analytical IK solutions. Select the second kinematic group by clicking the **Use this kinematic group** link in the second row of the table.

```
aik = analyticalInverseKinematics(robot);
showdetails(aik)
```

```
--------------------
Robot: (8 bodies)

Index       Base Name    EE Body Name      Type                         Actions
```

```
-----      ---------   ------------    ----                     -------
    1      base_link         link_6   RRRSSS   Use this kinematic group
    2      base_link          tool0   RRRSSS   Use this kinematic group
```

Inspect the kinematic group, which lists the base and end-effector body names. For this robot, the group uses the `'base_link'` and `'tool0'` bodies, respectively.

```
aik.KinematicGroup
```

```
ans = struct with fields:
               BaseName: 'base_link'
    EndEffectorBodyName: 'tool0'
```

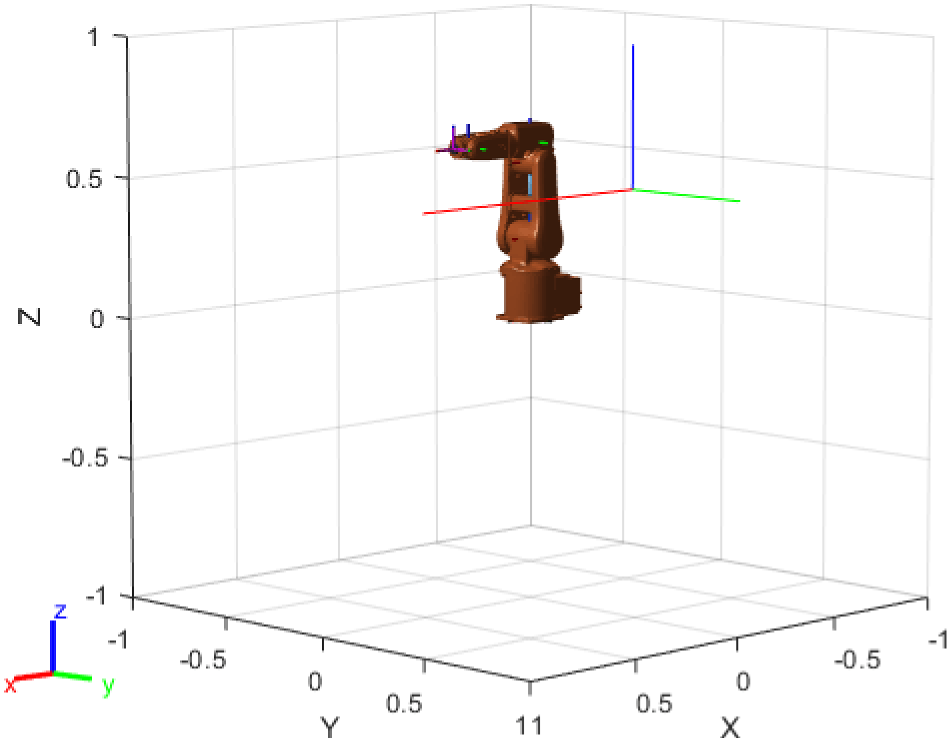
Generate Function

Generate the IK function for the selected kinematic group. Specify a name for the function, which is generated and saved in the current directory.

```
generateIKFunction(aik,'robotIK');
```

Specify a desired end-effector position. Convert the *xyz*-position to a homogeneous transformation.

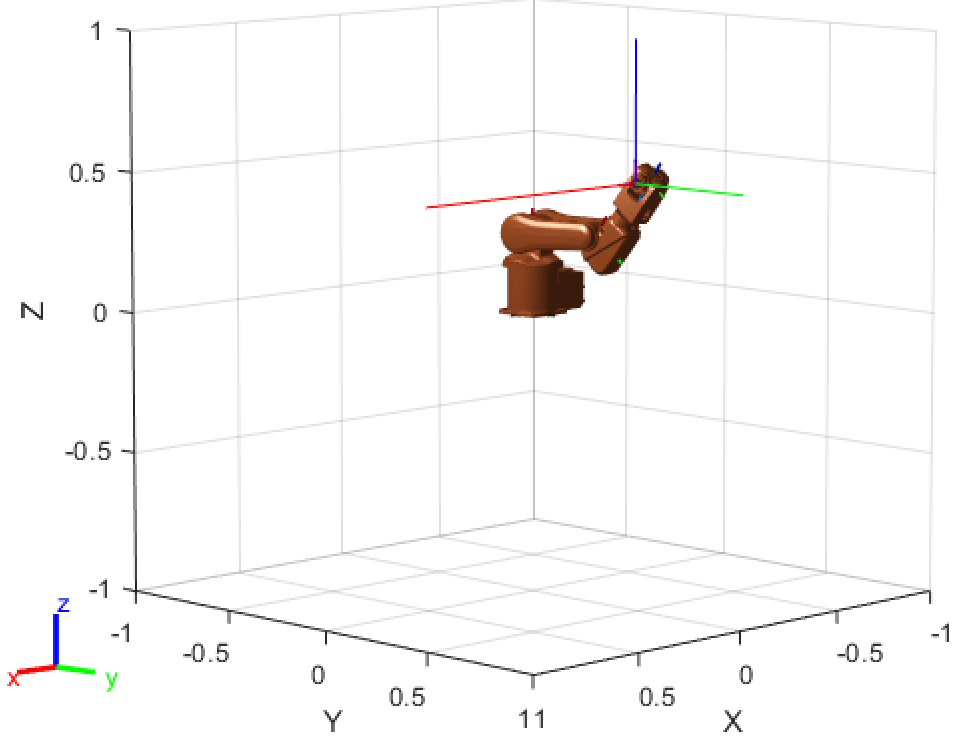
```
eePosition = [0 0.5 0.5];
eePose = trvec2tform(eePosition);
hold on
plotTransforms(eePosition,tform2quat(eePose))
hold off
```



Generate Configuration for IK Solution

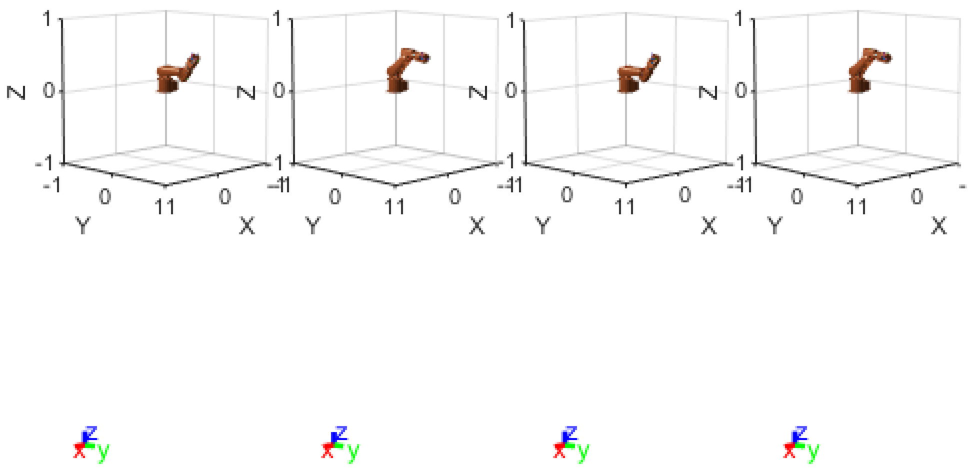
Specify the homogeneous transformation to the generated IK function, which generates all solutions for the desired end-effector pose. Display the first generated configuration to verify that the desired pose has been achieved.

```
ikConfig = robotIK(eePose); % Uses the generated file

show(robot,ikConfig(1,:));
hold on
plotTransforms(eePosition,tform2quat(eePose))
hold off
```



Display all of the closed-form IK solutions sequentially.

```
figure;
numSolutions = size(ikConfig,1);

for i = 1:size(ikConfig,1)
    subplot(1,numSolutions,i)
    show(robot,ikConfig(i,:));
end
```



Solve Analytical IK for Large-DOF Robot

Some manipulator robot models have large degrees-of-freedom (DOFs). To reach certain end-effector poses, however, only six DOFs are required. Use the `analyticalInverseKinematics` object, which supports six-DOF robots, to determine various valid kinematic groups for this large-DOF robot model. Use the `showdetails` object function to get information about the model.

Load Robot Model and Generate IK Solver

Load the robot model into the workspace, and create an `analyicalInverseKinematics` object. Use the `showdetails` object function to see the supported kinematic groups.

```
robot = loadrobot('willowgaragePR2','DataFormat','row');
aik = analyticalInverseKinematics(robot);
opts = showdetails(aik);
```

```
--------------------
Robot: (94 bodies)

Index                                            Base Name                              EE
-----                                            ---------                              --
    1                                   l_shoulder_pan_link                        l_wris
    2                                   r_shoulder_pan_link                        r_wris
    3                                   l_shoulder_pan_link                      l_gripper
    4                                   r_shoulder_pan_link                      r_gripper
```

```
    5                                l_shoulder_pan_link                                l_gripper
    6                                l_shoulder_pan_link                 l_gripper_motor_accelero
    7                                l_shoulder_pan_link                               l_gripper_
    8                                r_shoulder_pan_link                                r_gripper
    9                                r_shoulder_pan_link                 r_gripper_motor_accelero
   10                                r_shoulder_pan_link                               r_gripper_
```

Select a group programmatically using the output of the `showdetails` object function, `opts`. The selected group uses the left shoulder as the base with the left wrist as the end effector.

```
aik.KinematicGroup = opts(1).KinematicGroup;
disp(aik.KinematicGroup)
```

```
               BaseName: 'l_shoulder_pan_link'
    EndEffectorBodyName: 'l_wrist_roll_link'
```

Generate the IK function for the selected group.

```
generateIKFunction(aik,'willowRobotIK');
```

Solve Analytical IK

Define a target end-effector pose using a randomly-generated configuration.

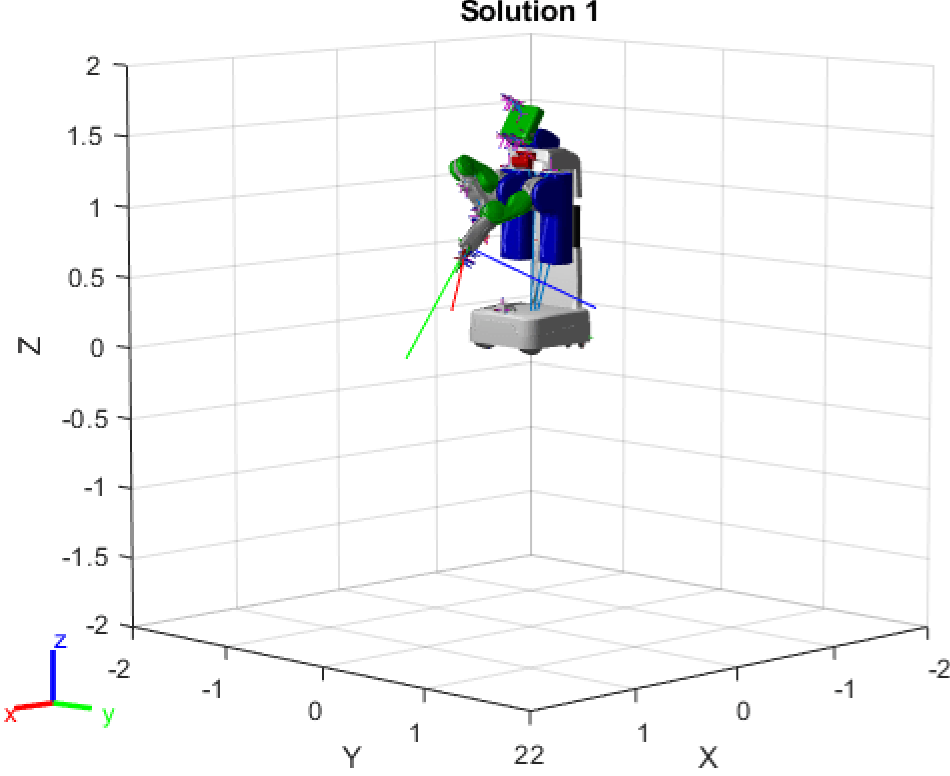
```
rng(0);
expConfig = randomConfiguration(robot);

eeBodyName = aik.KinematicGroup.EndEffectorBodyName;
baseName = aik.KinematicGroup.BaseName;
expEEPose = getTransform(robot,expConfig,eeBodyName,baseName);
```

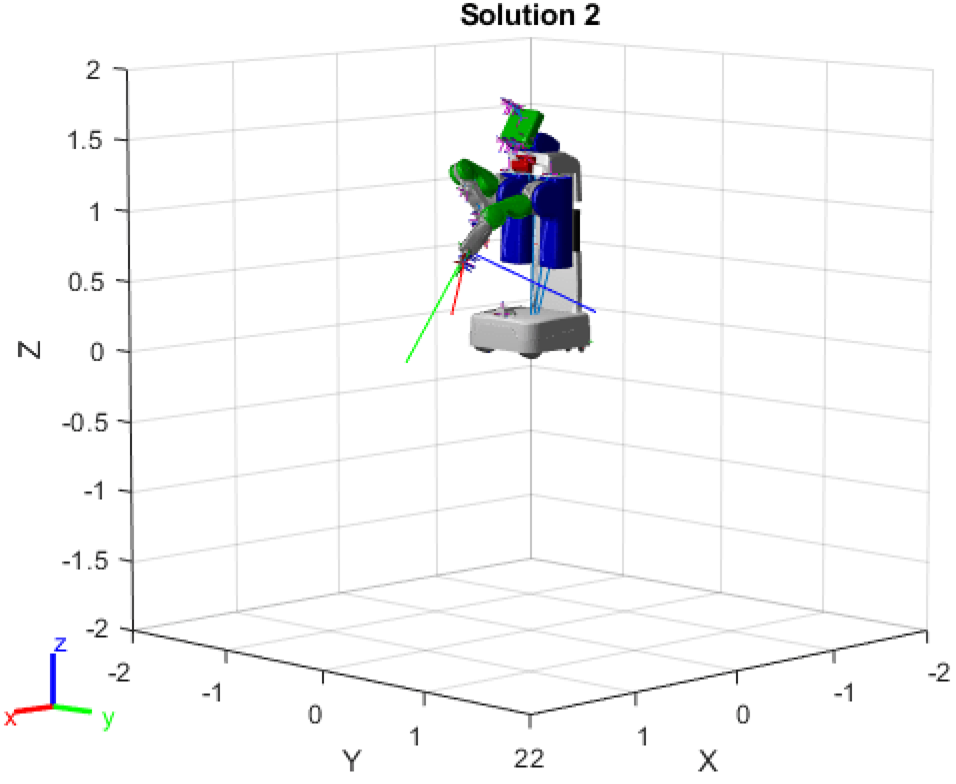
Solve for all robot configurations that achieve the defined end-effector pose using the generated IK function. To ignore joint limits, specify `false` as the second input argument.

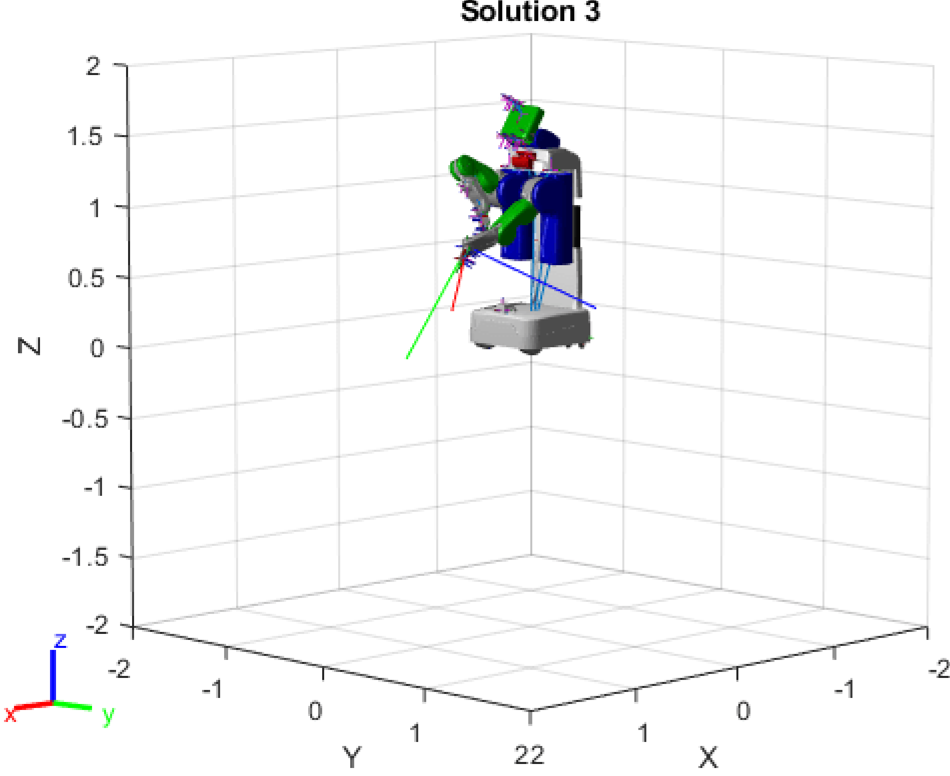
```
ikConfig = willowRobotIK(expEEPose,false);
```

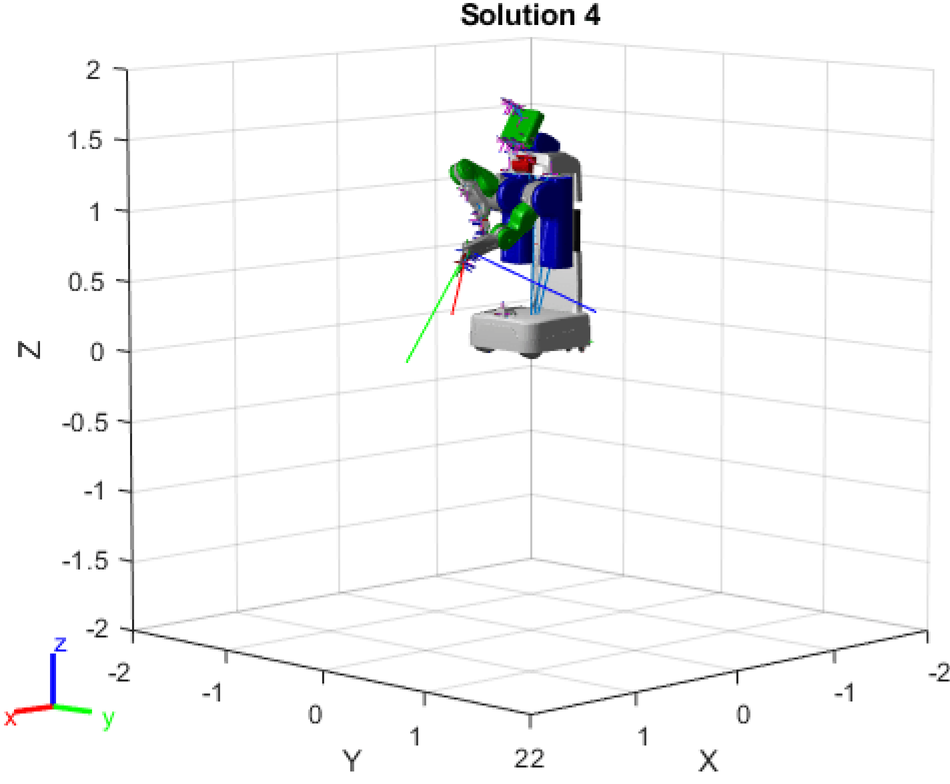
To display the target end-effector pose in the world frame, get the transformation from the base of the robot model, rather than the base of the kinematic group. Display all of the generated IK solutions by specifying the indices for your kinematic group IK solution in the configuration vector used with the `show` function.

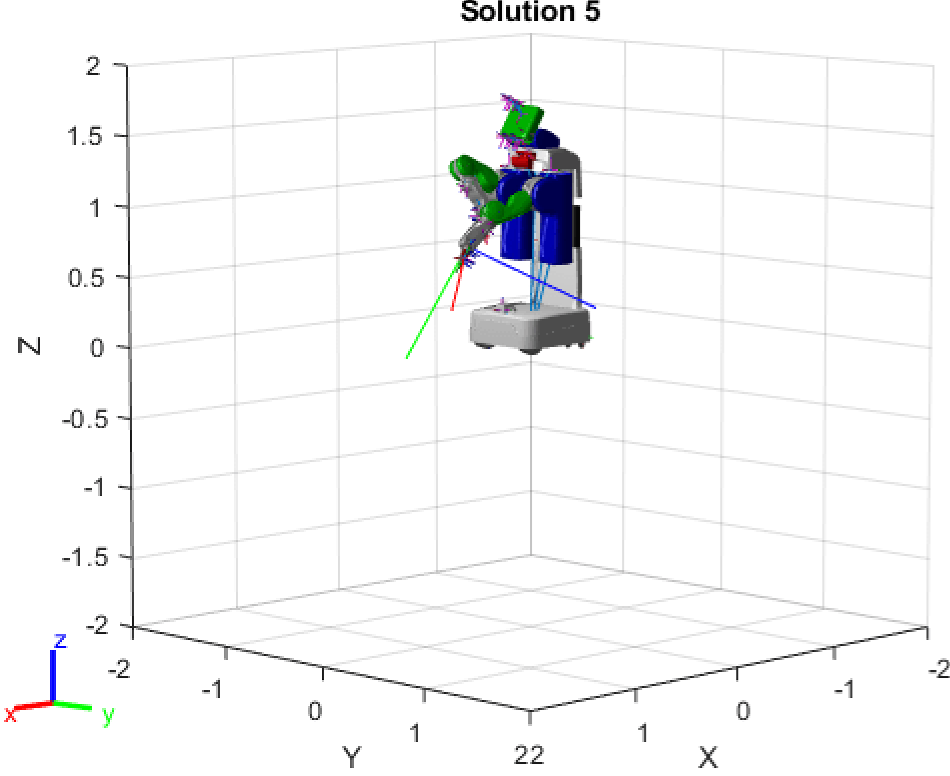
```
eeWorldPose = getTransform(robot,expConfig,eeBodyName);

generatedConfig = repmat(expConfig, size(ikConfig,1), 1);
generatedConfig(:,aik.KinematicGroupConfigIdx) = ikConfig;

for i = 1:size(ikConfig,1)
    figure;
    ax = show(robot,generatedConfig(i,:));
    hold all;
    plotTransforms(tform2trvec(eeWorldPose),tform2quat(eeWorldPose),'Parent',ax);
    title(['Solution ' num2str(i)]);
end
```

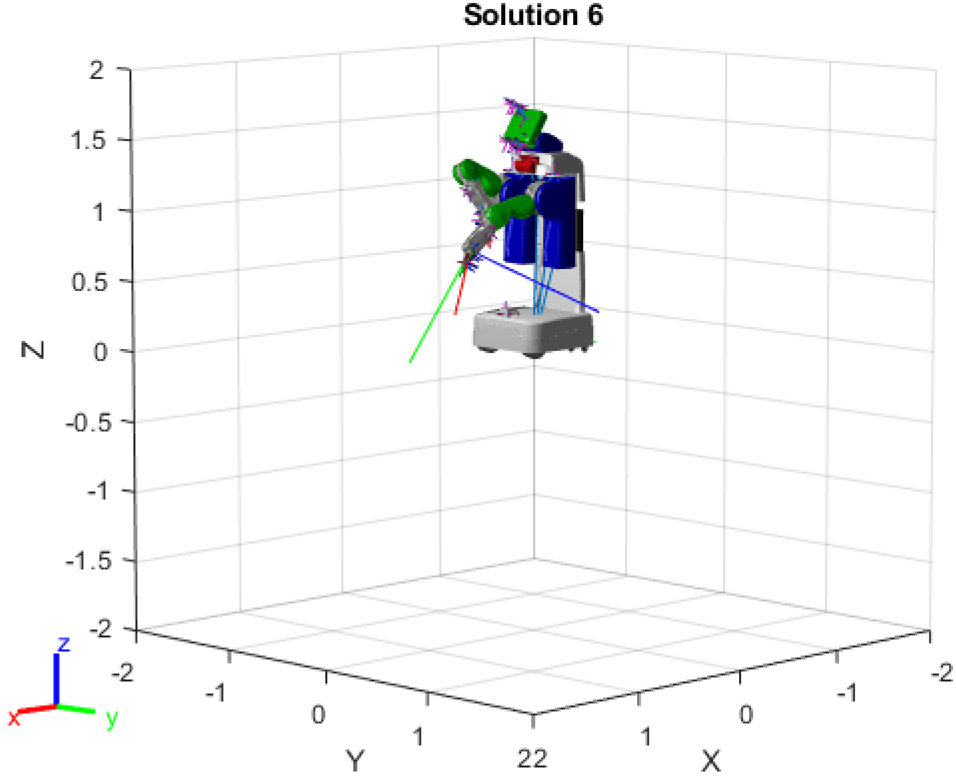

Solution 1
Z
Y
X
2
1.5
1
0.5
0
-0.5
-1
-1.5
-2
-2
-1
0
1
22
1
0
-1
-2
z
x
y

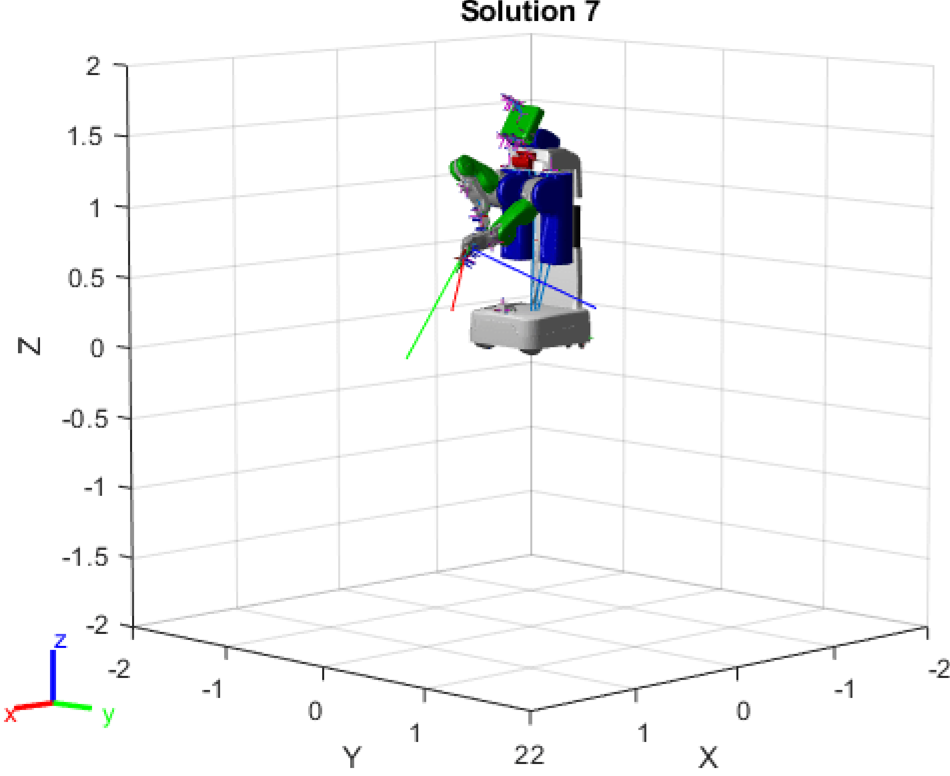

Solution 2
2
1.5
1
0.5
Z
0
-0.5
-1
-1.5
-2
-2
-1
0
1
Y
22
1
0
-1
-2
X
z
x
y

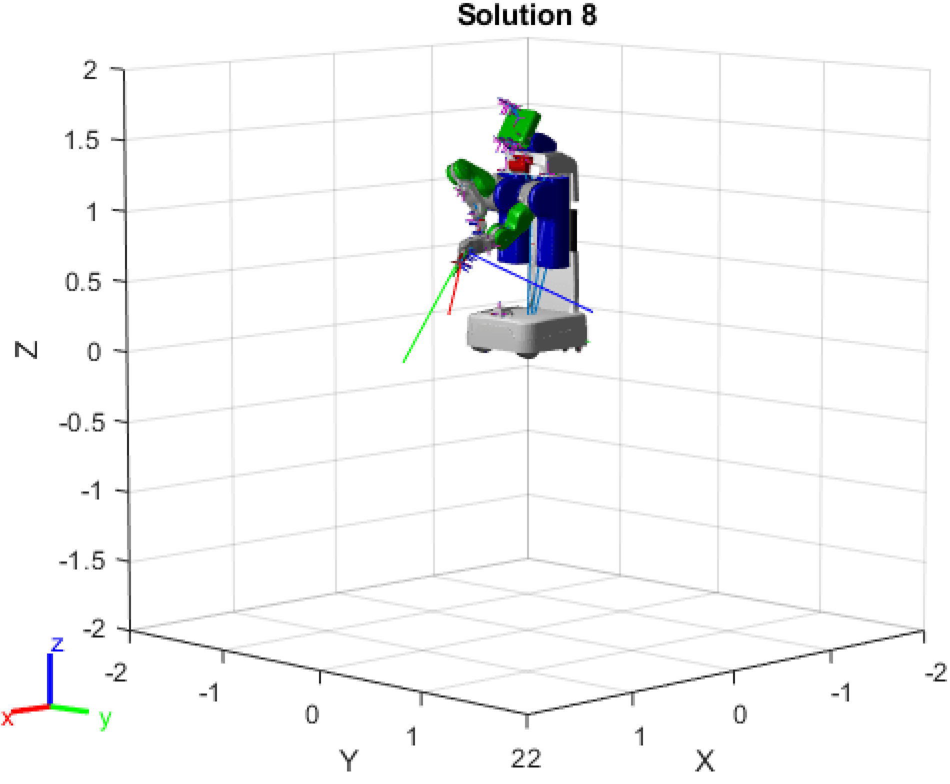

Solution 3
Z
Y
X
2
1.5
1
0.5
0
-0.5
-1
-1.5
-2
-2
-1
0
1
22
1
0
-1
-2
z
x
y


Solution 4
Z
Y
X
2
1.5
1
0.5
0
-0.5
-1
-1.5
-2
-2
-1
0
1
22
1
0
-1
-2
z
x
y


Solution 5
2
1.5
1
0.5
0
-0.5
-1
-1.5
-2
Z
-2
-1
0
1
2
Y
2
1
0
-1
-2
X
z
x
y


Solution 6
2
1.5
1
0.5
0
-0.5
-1
-1.5
-2
Z
-2
-1
0
1
2
Y
2
1
0
-1
-2
X
z
x
y


Solution 7
2
1.5
1
0.5
Z 0
-0.5
-1
-1.5
-2
-2
-1
0
1
Y
22
1
0
X
-1
-2
z
x
y



Version History

Introduced in R2020b

References

[1] Pieper, Donald. *The Kinematics of Manipulators Under Computer Control*. Stanford University, 1968.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

The `analyticalInverseKinematics` object only supports code generation for the function created by calling the `generateIKFunction`. Use the `analyticalInverseKinematics` object to modify parameters and setup the solver. Then, use `generateIKFunction` to create your custom IK function for your robot model. Call `codegen` on the output `ikFunction` that is generated.

See Also

Objects

`inverseKinematics` | `generalizedInverseKinematics` | `rigidBodyTree`

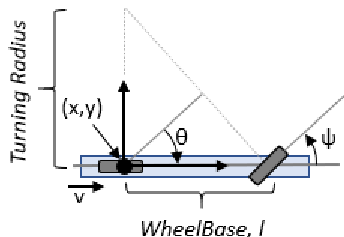
Functions
`loadrobot` | `importrobot` | `generateIKFunction` | `showdetails`

bicycleKinematics

Bicycle vehicle model

Description

`bicycleKinematics` creates a bicycle vehicle model to simulate simplified car-like vehicle dynamics. This model represents a vehicle with two axles separated by a distance, WheelBase. The state of the vehicle is defined as a three-element vector, [*x y theta*], with a global *xy*-position, specified in meters, and a vehicle heading angle, *theta*, specified in radians. The front wheel can be turned with steering angle *psi*. The vehicle heading, *theta*, is defined at the center of the rear axle. To compute the time derivative states of the model, use the `derivative` function with input commands and the current robot state.



Creation

Syntax

`kinematicModel = bicycleKinematics`

`kinematicModel = bicycleKinematics(Name,Value)`

Description

`kinematicModel = bicycleKinematics` creates a bicycle kinematic model object with default property values.

`kinematicModel = bicycleKinematics(Name,Value)` sets additional properties to the specified values. You can specify multiple properties in any order.

Properties

`WheelBase` — Distance between front and rear axles
`1` (default) | positive numeric scalar

The wheel base refers to the distance between the front and rear vehicle axles, specified in meters.

`VehicleSpeedRange` — Range of vehicle speeds
`[-Inf Inf]` (default) | positive numeric scalar

The vehicle speed range is a two-element vector that provides the minimum and maximum vehicle speeds, [*MinSpeed MaxSpeed*], specified in meters per second.

`MaxSteeringAngle` — Maximum steering angle
`pi/4` (default) | numeric scalar

The maximum steering angle, *psi*, refers to the maximum angle the vehicle can be steered to the right or left, specified in radians. This property is used to validate the user-provided state input.

If `MaxSteeringAngle` is set to `pi/2`, it results in a `MinimumTurningRadius` value of `0`.

`MinimumTurningRadius` — Minimum vehicle turning radius
`1.0000` (default) | numeric scalar

This property is read-only.

The minimum vehicle turning radius, specified as a numeric scalar, in meters. The minimum radius is computed using the wheel base and the maximum steering angle.

`VehicleInputs` — Type of motion inputs for vehicle
`"VehicleSpeedSteeringAngle"` (default) | character vector | string scalar

The `VehicleInputs` property specifies the format of the model input commands when using the `derivative` function. The property has two valid options, specified as a string or character vector:

- `"VehicleSpeedSteeringAngle"` — Vehicle speed and steering angle
- `"VehicleSpeedHeadingRate"` — Vehicle speed and heading angular velocity

Object Functions

derivative   Time derivative of vehicle state

Examples

Plot Path of Bicycle Kinematic Robot

Create a Robot

Define a robot and set the initial starting position and orientation.

```
kinematicModel = bicycleKinematics;
initialState = [0 0 0];
```

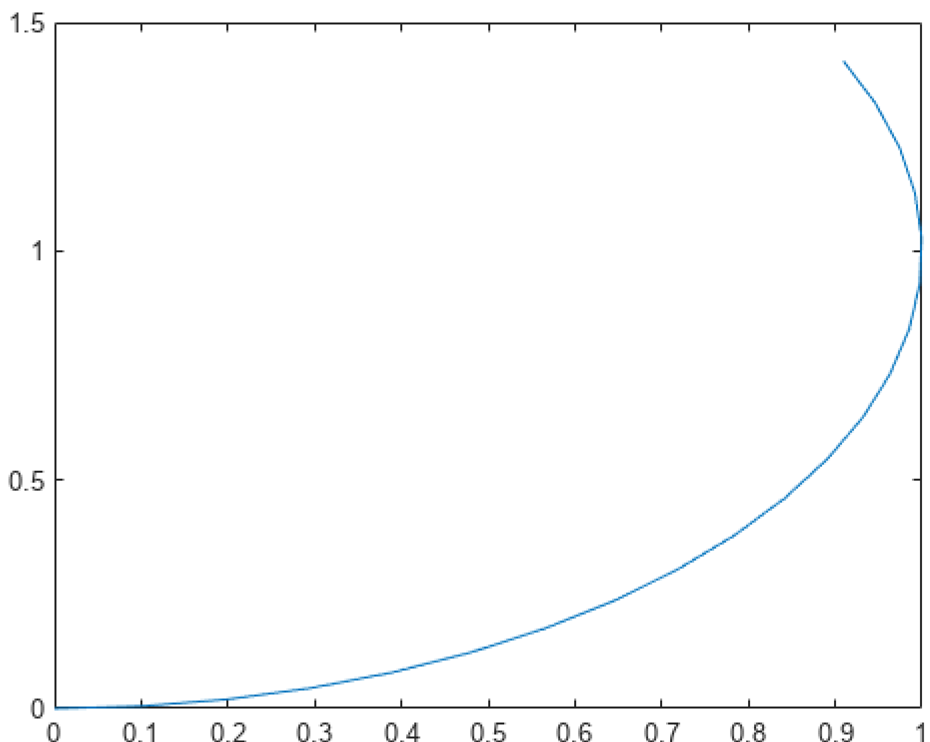
Simulate Robot Motion

Set the timespan of the simulation to 1 s with 0.05 s time steps and the input commands to 2 m/s for the vehicle speed and `pi/4` rad for the steering angle to create a left turn. Simulate the motion of the robot by using the `ode45` solver on the `derivative` function.

```
tspan = 0:0.05:1;
inputs = [2 pi/4]; %Turn left
[t,y] = ode45(@(t,y)derivative(kinematicModel,y,inputs),tspan,initialState);
```

Plot path

```
figure
plot(y(:,1),y(:,2))
```



Version History

Introduced in R2019b

References

[1] Lynch, Kevin M., and Frank C. Park. *Modern Robotics: Mechanics, Planning, and Control* 1st ed. Cambridge, MA: Cambridge University Press, 2017.

[2] Corke, Peter I. *Robotics, Vision and Control: Fundamental Algorithms in MATLAB*. Springer, 2011.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

Classes
ackermannKinematics | unicycleKinematics | differentialDriveKinematics

Blocks
Bicycle Kinematic Model

Functions
derivative

Topics
"Simulate Different Kinematic Models for Mobile Robots"
"Mobile Robot Kinematics Equations"

binaryOccupancyMap

Create occupancy grid with binary values

Description

The `binaryOccupancyMap` creates a 2-D occupancy map object, which you can use to represent and visualize a robot workspace, including obstacles. The integration of sensor data and position estimates create a spatial representation of the approximate locations of the obstacles.

Occupancy grids are used in robotics algorithms such as path planning. They are also used in mapping applications, such as for finding collision-free paths, performing collision avoidance, and calculating localization. You can modify your occupancy grid to fit your specific application.

Each cell in the occupancy grid has a value representing the occupancy status of that cell. An occupied location is represented as `true (1)` and a free location is represented as `false (0)`.

The object keeps track of three reference frames: world, local, and, grid. The world frame origin is defined by `GridLocationInWorld`, which defines the bottom-left corner of the map relative to the world frame. The `LocalOriginInWorld` property specifies the location of the origin of the local frame relative to the world frame. The first grid location with index `(1,1)` begins in the top-left corner of the grid.

Note This object was previously named `robotics.BinaryOccupancyGrid`.

Creation

Syntax

```
map = binaryOccupancyMap
map = binaryOccupancyMap(width,height)
map = binaryOccupancyMap(width,height,resolution)

map = binaryOccupancyMap(rows,cols,resolution,"grid")

map = binaryOccupancyMap(p)
map = binaryOccupancyMap(p,resolution)

map = binaryOccupancyMap(sourcemap)
map = binaryOccupancyMap(sourcemap,resolution)
```

Description

`map = binaryOccupancyMap` creates a 2-D binary occupancy grid with a width and height of 10m. The default grid resolution is one cell per meter.

`map = binaryOccupancyMap(width,height)` creates a 2-D binary occupancy grid representing a work space of `width` and `height` in meters. The default grid resolution is one cell per meter.

map = binaryOccupancyMap(width,height,resolution) creates a grid with the Resolution property specified in cells per meter. The map is in world coordinates by default.

map = binaryOccupancyMap(rows,cols,resolution,"grid") creates a 2-D binary occupancy grid of size (rows,cols).

map = binaryOccupancyMap(p) creates a grid from the values in matrix p. The size of the grid matches the size of the matrix, with each cell value interpreted from its location in the matrix. p contains any numeric or logical type with zeros (0) and ones (1).

map = binaryOccupancyMap(p,resolution) creates a map from a matrix with the Resolution property specified in cells per meter.

map = binaryOccupancyMap(sourcemap) creates an object using values from another binaryOccupancyMap object.

map = binaryOccupancyMap(sourcemap,resolution) creates an object using values from another binaryOccupancyMap object, but resamples the matrix to have the specified resolution.

Input Arguments

width — Map width
positive scalar

Map width, specified as a positive scalar in meters.

height — Map height
positive scalar

Map height, specified as a positive scalar in meters.

p — Map grid values
matrix

Map grid values, specified as a matrix.

sourcemap — Occupancy map object
binaryOccupancyMap object

Occupancy map object, specified as a binaryOccupancyMap object.

Properties

GridSize — Number of rows and columns in grid
two-element vector of form [rows cols]

This property is read-only.

Number of rows and columns in grid, stored as a two-element vector of the form [rows cols].

Resolution — Grid resolution
1 (default) | scalar

This property is read-only.

Grid resolution, stored as a scalar in cells per meter.

XLocalLimits — Minimum and maximum values of *x*-coordinates in local frame
two-element vector of form [min max]

This property is read-only.

Minimum and maximum values of *x*-coordinates in local frame, stored as a two-element vector of the form [min max]. Local frame is defined by LocalOriginInWorld property.

YLocalLimits — Minimum and maximum values of *y*-coordinates in local frame
two-element vector of form [min max]

This property is read-only.

Minimum and maximum values of *y*-coordinates in local frame, stored as a two-element vector of the form [min max]. Local frame is defined by LocalOriginInWorld property.

XWorldLimits — Minimum and maximum values of *x*-coordinates in world frame
two-element vector of form [min max]

This property is read-only.

Minimum and maximum values of *x*-coordinates in world frame, stored as a two-element vector of the form [min max]. These values indicate the world range of the *x*-coordinates in the grid.

YWorldLimits — Minimum and maximum values of *y*-coordinates
two-element vector of form [min max]

This property is read-only.

Minimum and maximum values of *y*-coordinates, stored as a two-element vector of the form [min max]. These values indicate the world range of the *y*-coordinates in the grid.

GridLocationInWorld — Location of the grid in world coordinates
[0 0] (default) | two-element vector | [xGrid yGrid]

Location of the bottom-left corner of the grid in world coordinates, specified as a two-element vector, [xGrid yGrid].

LocalOriginInWorld — Location of the local frame in world coordinates
[0 0] (default) | two-element vector | [xWorld yWorld]

Location of the origin of the local frame in world coordinates, specified as a two-element vector, [xLocal yLocal]. Use the move function to shift the local frame as your vehicle moves.

GridOriginInLocal — Location of the grid in local coordinates
[0 0] (default) | two-element vector | [xLocal yLocal]

Location of the bottom-left corner of the grid in local coordinates, specified as a two-element vector, [xLocal yLocal].

DefaultValue — Default value for unspecified map locations
0 (default) | 1

Default value for unspecified map locations including areas outside the map, specified as 0 or 1.

Object Functions

copy	Create copy of binary occupancy map
checkOccupancy	Check if locations are free or occupied
getOccupancy	Get occupancy value of locations
grid2local	Convert grid indices to local coordinates
grid2world	Convert grid indices to world coordinates
inflate	Inflate each occupied location
insertRay	Insert ray from laser scan observation
local2grid	Convert local coordinates to grid indices
local2world	Convert local coordinates to world coordinates
move	Move map in world frame
occupancyMatrix	Convert occupancy grid to matrix
raycast	Compute cell indices along a ray
rayIntersection	Find intersection points of rays and occupied map cells
setOccupancy	Set occupancy value of locations
show	Display binary occupancy map
syncWith	Sync map with overlapping map
world2grid	Convert world coordinates to grid indices
world2local	Convert world coordinates to local coordinates

Examples

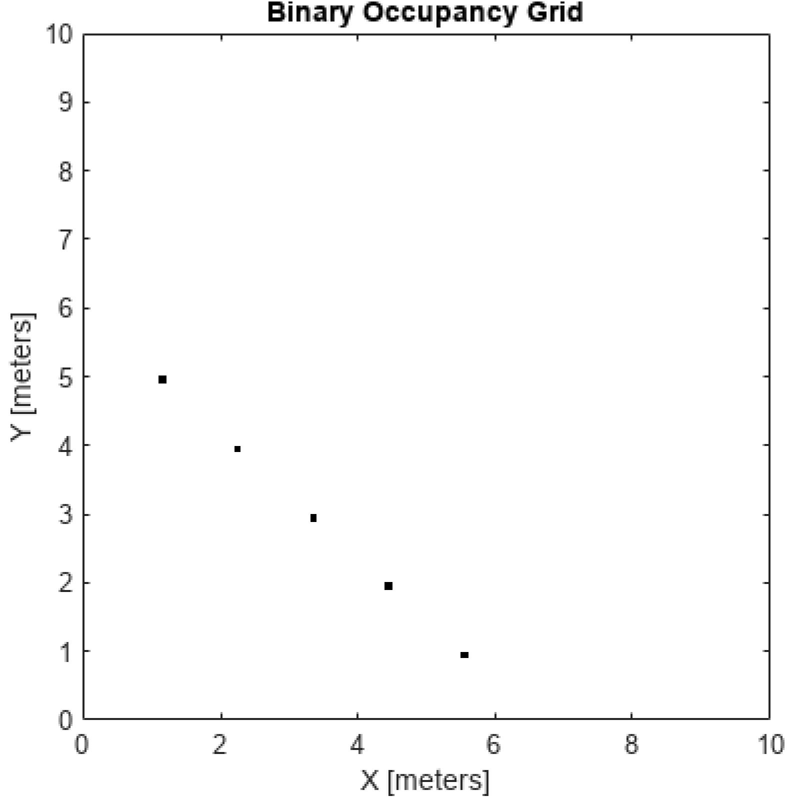
Create and Modify Binary Occupancy Grid

Create a 10m x 10m empty map.

```
map = binaryOccupancyMap(10,10,10);
```

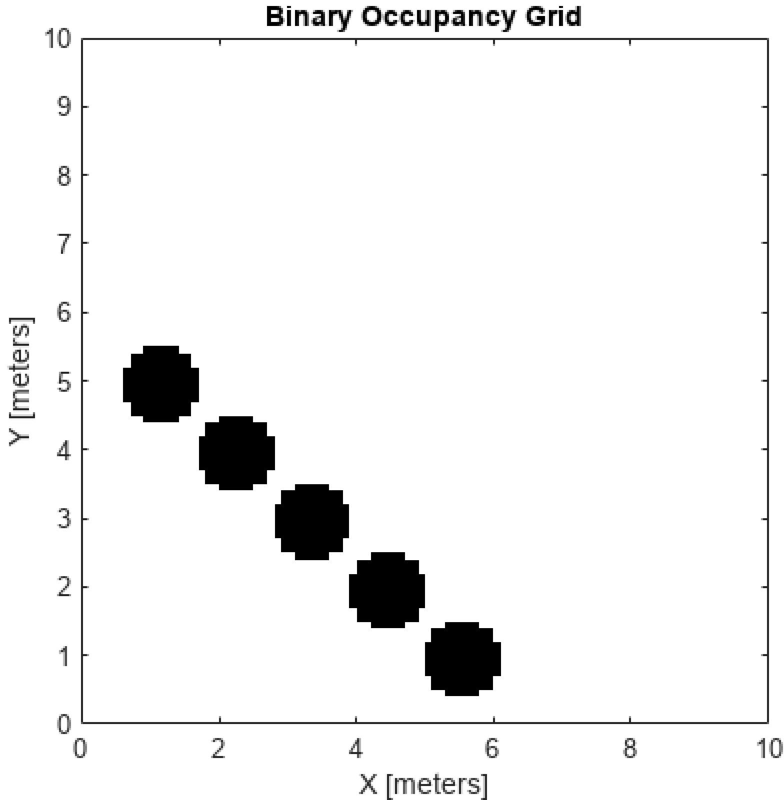
Set occupancy of world locations and show map.

```
x = [1.2; 2.3; 3.4; 4.5; 5.6];
y = [5.0; 4.0; 3.0; 2.0; 1.0];

setOccupancy(map, [x y], ones(5,1))
figure
show(map)
```



Inflate occupied locations by a given radius.

```
inflate(map, 0.5)
figure
show(map)
```



Get grid locations from world locations.

```
ij = world2grid(map, [x y]);
```

Set grid locations to free locations.

```
setOccupancy(map, ij, zeros(5,1), 'grid')
figure
show(map)
```

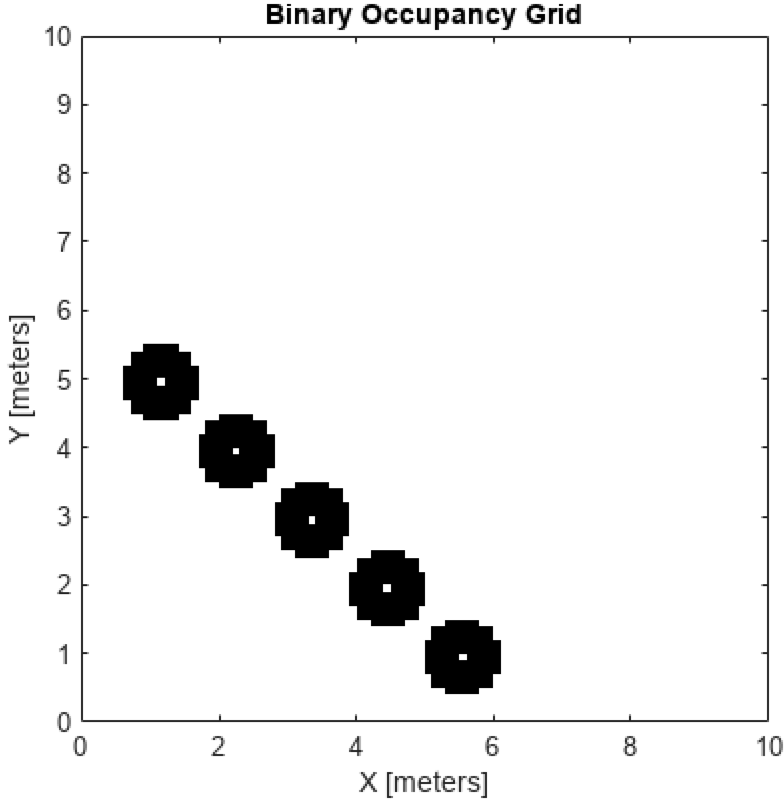


Image to Binary Occupancy Grid Example

This example shows how to convert an image to a binary occupancy grid for using with mapping and path planning.

Import image.

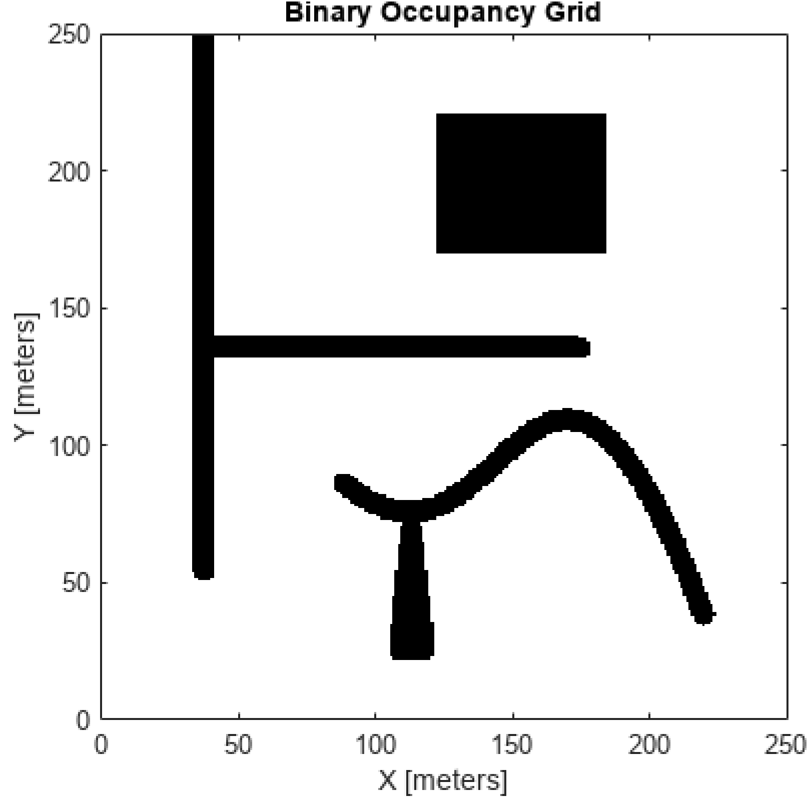
```
image = imread('imageMap.png');
```

Convert to grayscale and then black and white image based on given threshold value.

```
grayimage = rgb2gray(image);
bwimage = grayimage < 0.5;
```

Use black and white image as matrix input for binary occupancy grid.

```
grid = binaryOccupancyMap(bwimage);

show(grid)
```

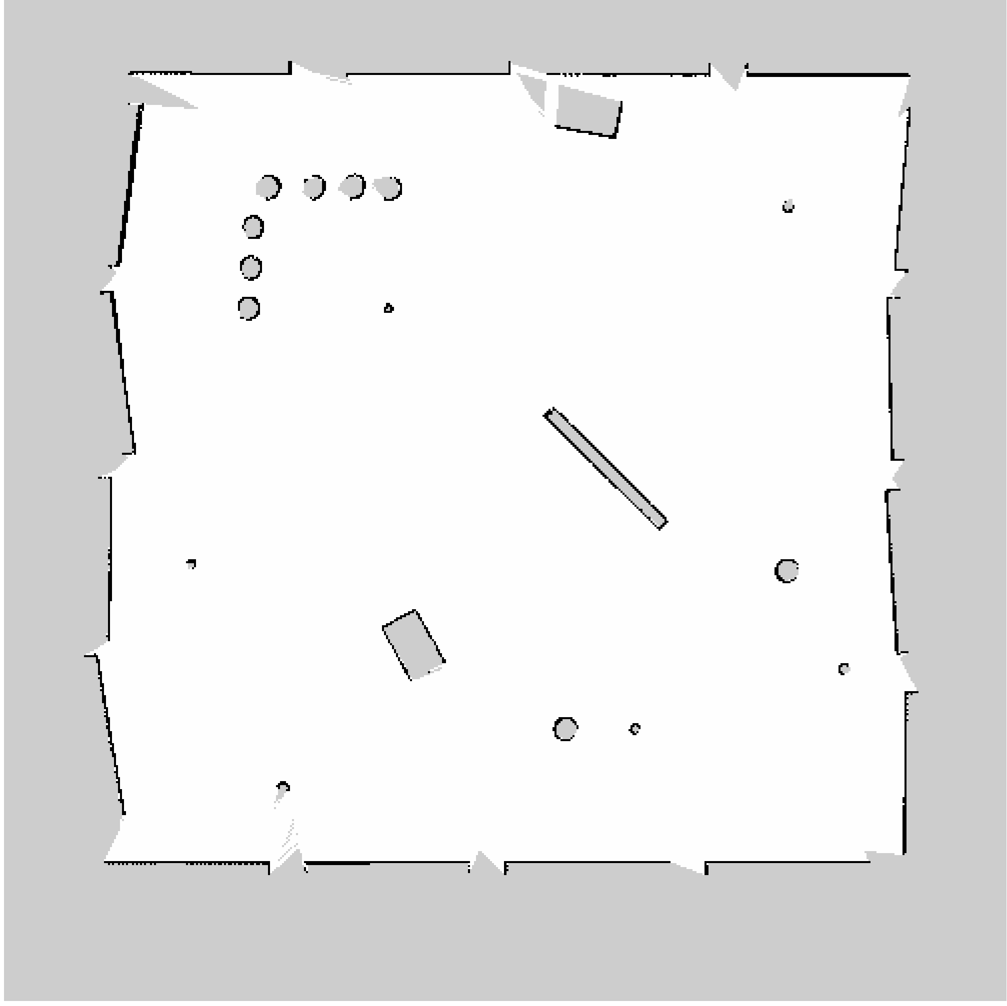


Convert PGM Image to Map

This example shows how to convert a .pgm file into a binaryOccupancyMap object for use in MATLAB.

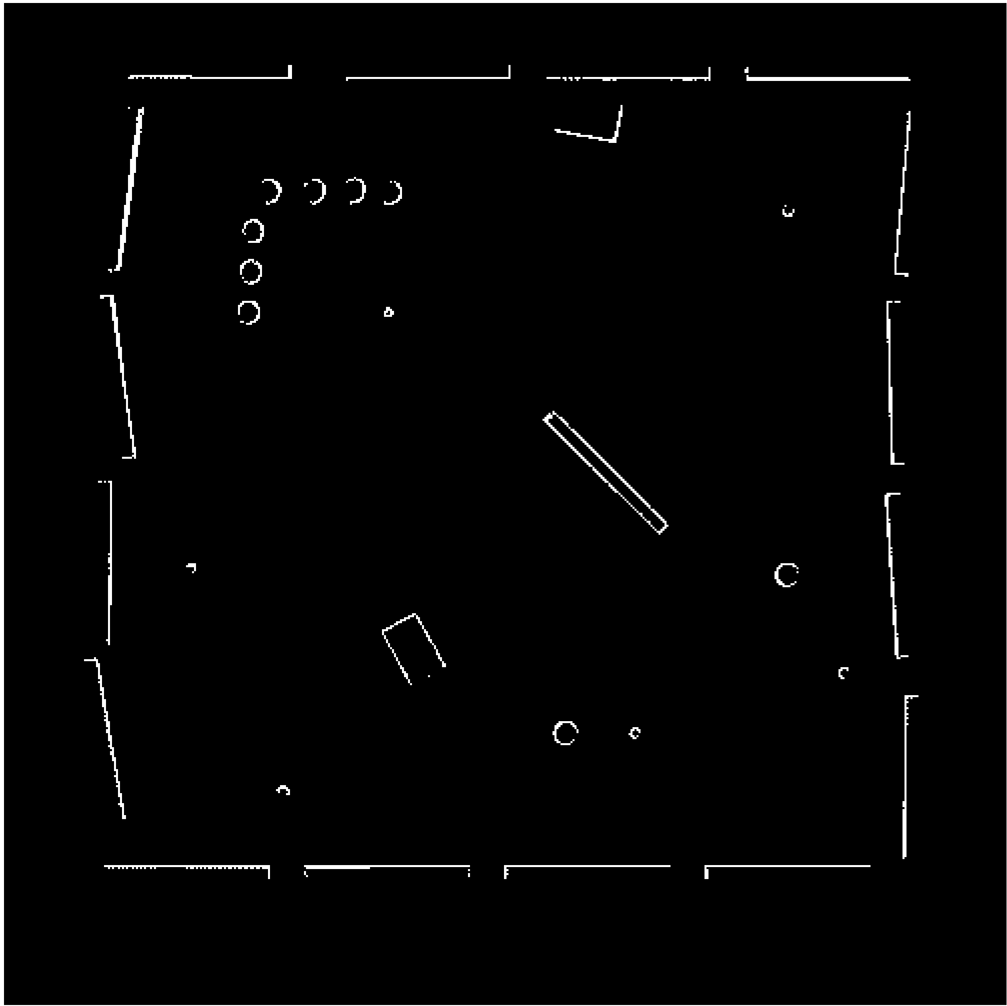
Import image using imread. The image is quite large and should be cropped to the relevant area.

```
image = imread('playpen_map.pgm');
imageCropped = image(750:1250,750:1250);
imshow(imageCropped)
```



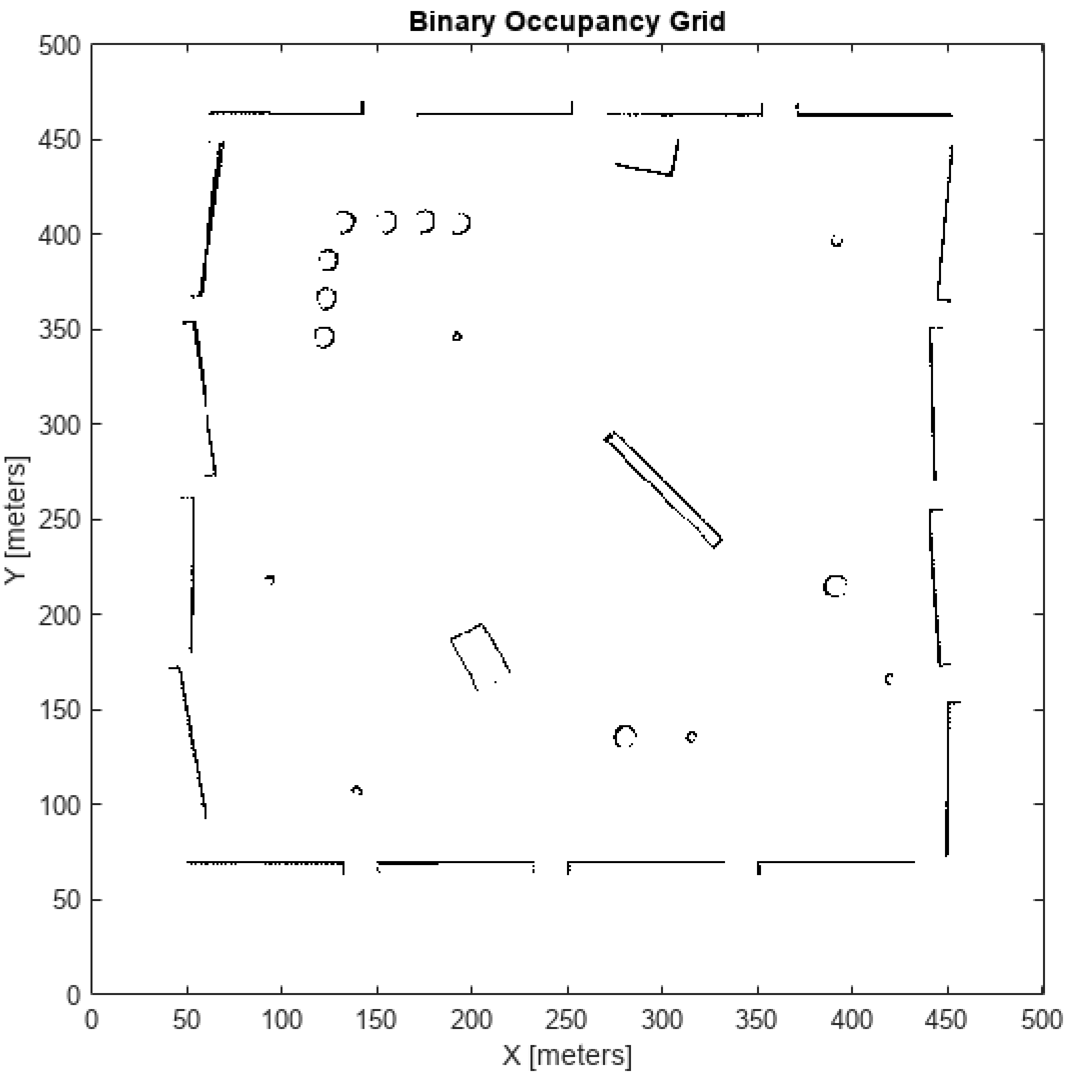
Unknown areas (gray) should be removed and treated as free space. Create a logical matrix based on a threshold. Depending on your image, this value could be different. Occupied space should be set as 1 (white in image).

```
imageBW = imageCropped < 100;
imshow(imageBW)
```



Create `binaryOccupancyMap` object using adjusted map image.

```
map = binaryOccupancyMap(imageBW);
show(map)
```



Version History

Introduced in R2015a

R2019b: `binaryOccupancyMap` was renamed

Behavior change in future release

The `binaryOccupancyMap` object was renamed from `robotics.BinaryOccupancyGrid`. Use `binaryOccupancyMap` for all object creation.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

As of MATLAB® R2022a, default map behavior during code generation has changed, which may result in backwards compatibility issues. Maps such as `binaryOccupancyMap` now support fixed-size code generation (`DynamicMemoryAllocation="off"`).

1. Maps that are either default-constructed or constructed with compile-time constant size information (or matrices that are of compile-time constant size) produce fixed-size maps.
2. To restore the previous behavior, use the `coder.ignoreConst` function when specifying size inputs, or `coder.varsize` matrix variable name specified as a string scalar or character vector, prior to constructing the map.

See Also

`mobileRobotPRM` | `controllerPurePursuit`

Topics

"Occupancy Grids"

capsuleApproximation

Approximate collision geometries of rigid body tree with capsules

Description

The `capsuleApproximation` object approximates the collision geometries associated with every body of a `rigidBodyTree` object by fitting collision capsules on each rigid body. This object enables you to query, modify, and visualize the collision capsules associated with rigid bodies of a rigid body tree.

Creation

Syntax

```
capapprox = capsuleApproximation(robot)
```

Description

`capapprox = capsuleApproximation(robot)` creates a capsule approximation `capapprox` of the input rigid body tree `robot`.

Input Arguments

`robot` — Rigid body tree robot model to approximate
`rigidBodyTree` object

Rigid body tree robot model to approximate with capsules, specified as a `rigidBodyTree` object. To use a provided robot model, see `loadrobot`. To import Unified Robot Description Format (URDF) models, see the `importrobot` function.

Properties

`RigidBodyTree` — Capsule-approximated rigid body tree robot model
`rigidBodyTree` object

Capsule-approximated rigid body tree robot model, stored as a `rigidBodyTree` object.

Object Functions

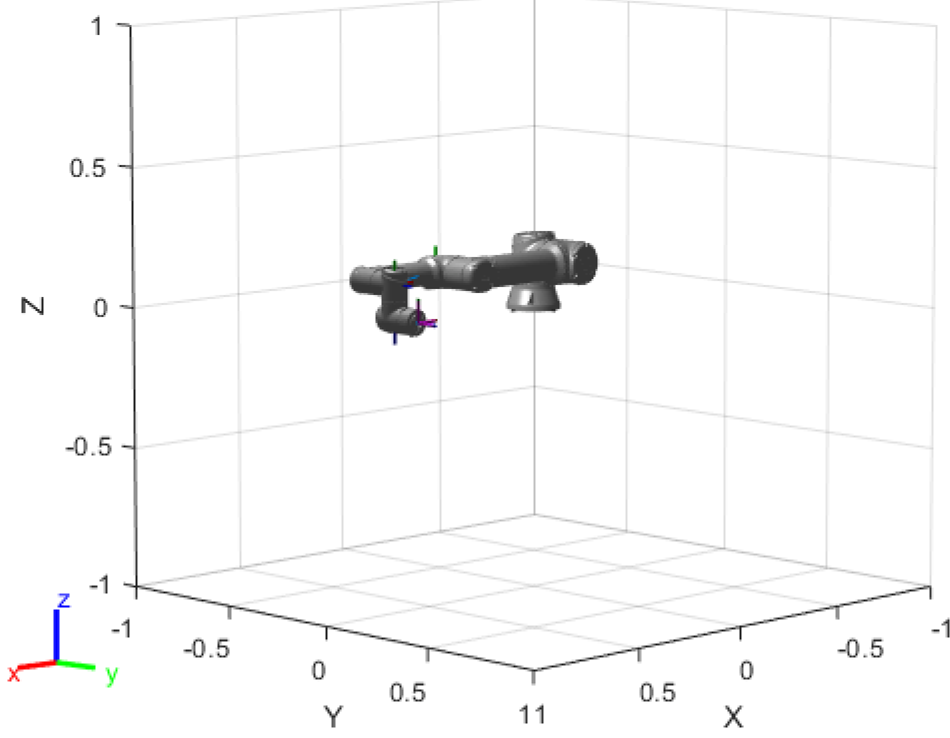
addCapsule	Add collision capsule to rigid body
removeCapsule	Remove collision capsule from rigid body
getCapsules	Get collision capsules of rigid body
show	Visualize capsule approximation of rigid body tree
updateGeometry	Update geometry of collision capsule of rigid body
updatePose	Update pose of collision capsule of rigid body

Examples

Create Capsule Approximation of Robot Model

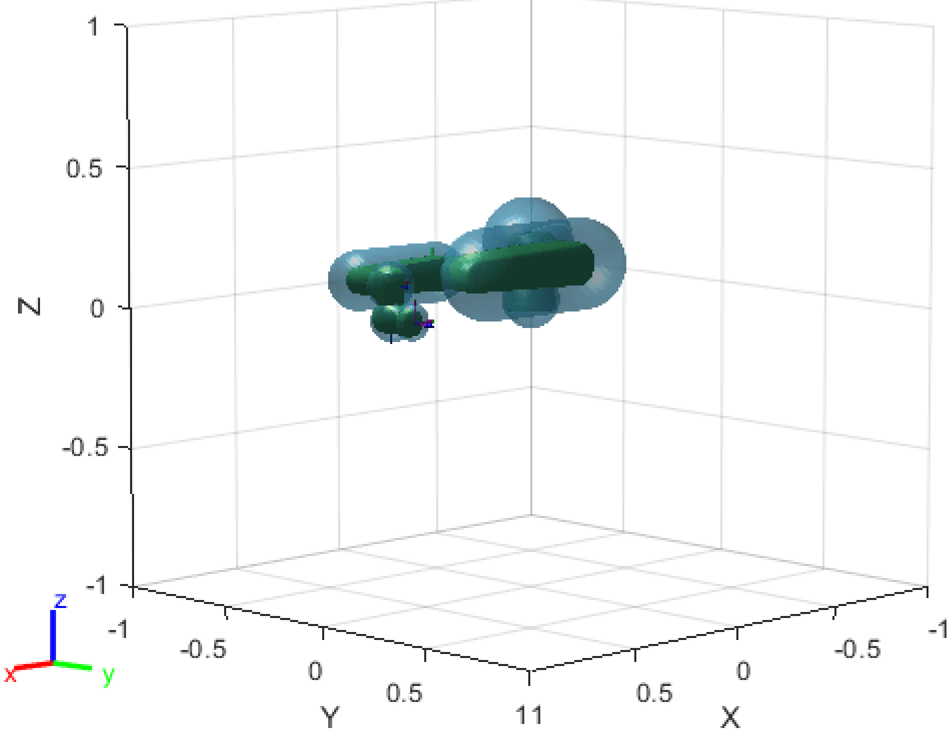
Load a robot into the workspace and visualize it.

```
robot = loadrobot("universalUR16e");
show(robot);
```



Create a capsule approximation of the robot, and visualize the capsule-approximated robot model.

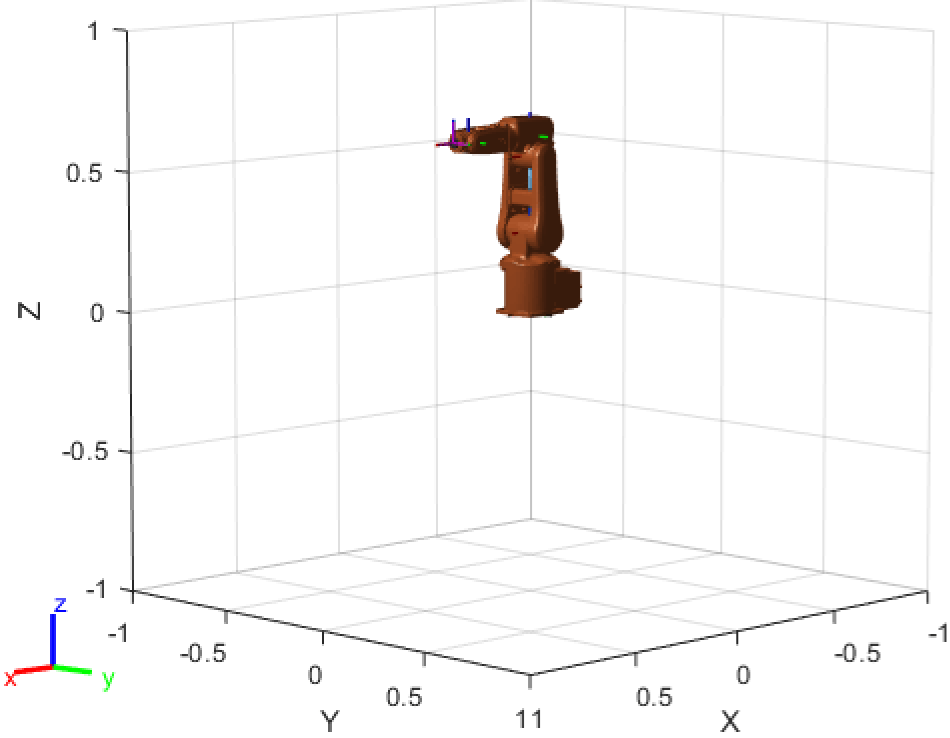
```
capsUR16 = capsuleApproximation(robot);
show(capsUR16,homeConfiguration(robot));
```



Create and Modify Capsule Approximation

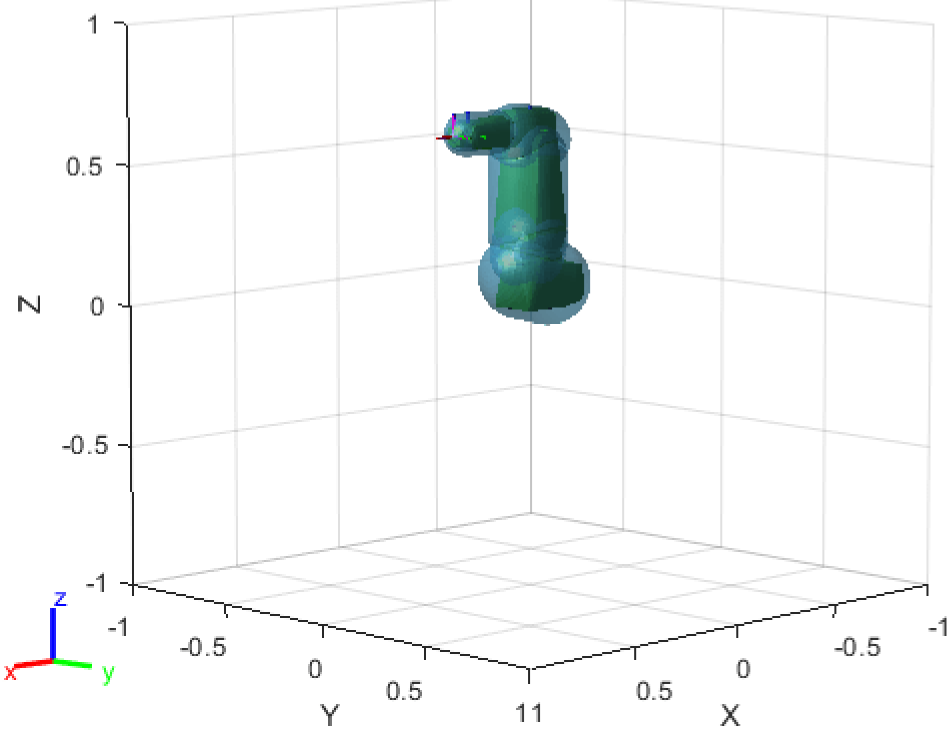
Load a robot into the workspace and visualize it.

```
robotIRB = loadrobot("abbIrb120");
show(robotIRB);
```



Create a capsule approximation of the robot, and visualize the capsule-approximated robot model.

```
capsIRB = capsuleApproximation(robotIRB);
figure
show(capsIRB,homeConfiguration(capsIRB.RigidBodyTree));
```



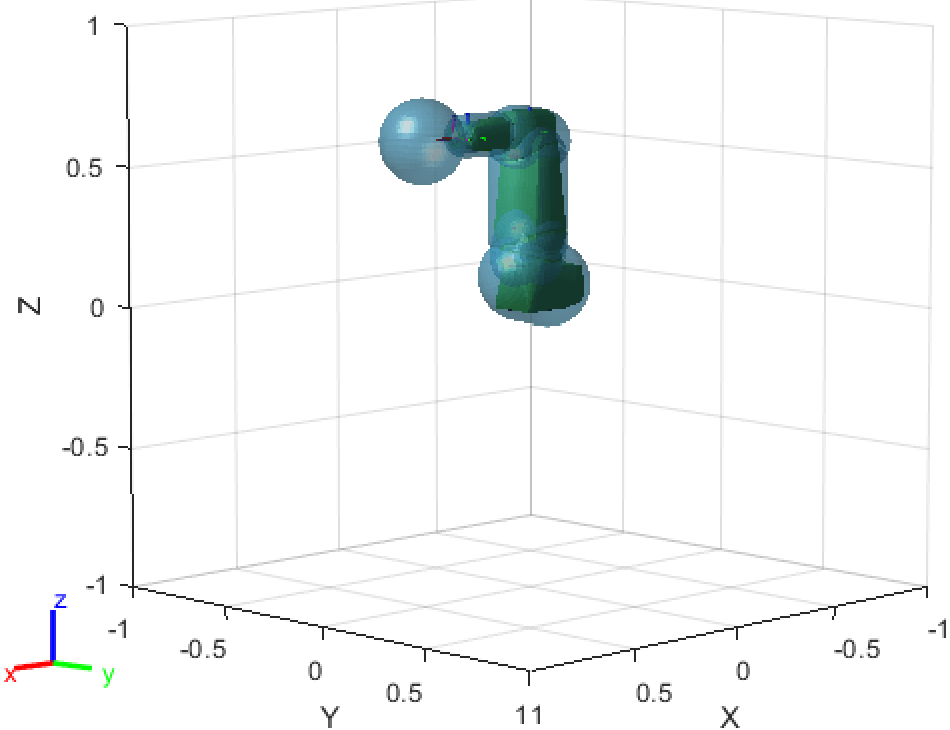
Use the `getCapsules` function to see if the end effector, `"tool0"`, has any collision capsules. Because `tool0` is just a frame, it has no collision mesh to approximate as a collision capsule.

```
capsulesTool = getCapsules(capsIRB,"tool0")
```

```
capsulesTool =

  1x0 empty cell array
```

Add a capsule to `tool0`, at a position `0.15` meters along the *x*-axis, with a radius of `0.15` and a length of `0`.

```
addCapsule(capsIRB,"tool0",[0.15 0],trvec2tform([0.15 0 0]))
show(capsIRB,homeConfiguration(capsIRB.RigidBodyTree));
```



Again check tool0 for a collision capsule, and verify the properties of the detected capsule.

```
capsulesTool = getCapsules(capsIRB,"tool0")
```

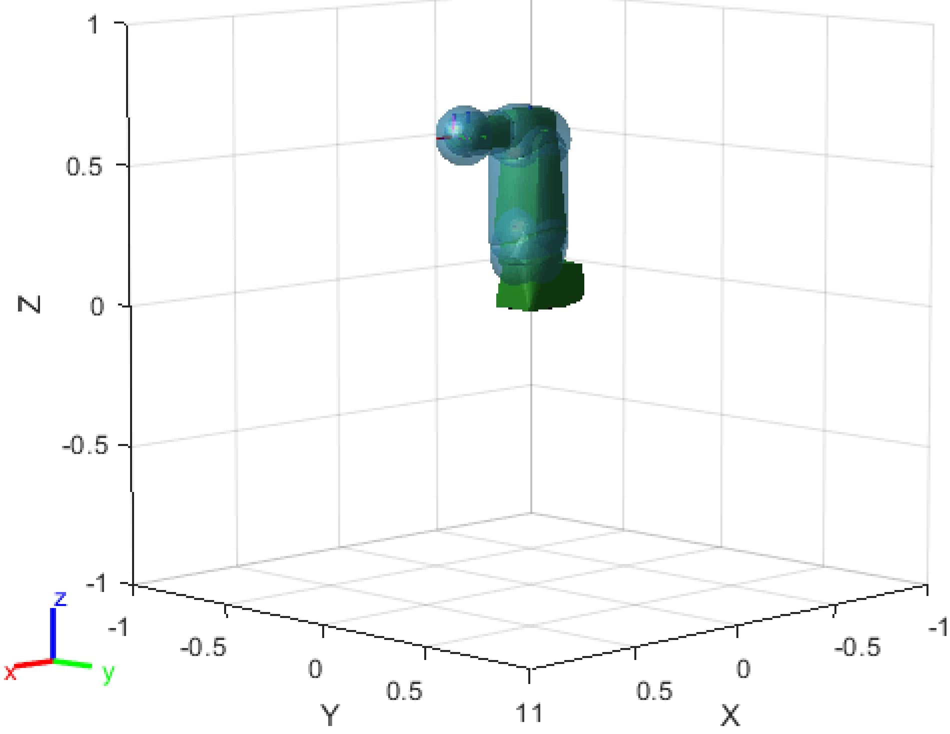
```
capsulesTool = 1x1 cell array
    {1x1 collisionCapsule}
```

```
capsulesTool{1}
```

```
ans = 
  collisionCapsule with properties:

    Radius: 0.1500
    Length: 0
      Pose: [4x4 double]
```

Remove the capsule from the base link. Then, reduce the collision capsule size of `tool0`, and move it `-0.05` meters from the previous position along the *x*-axis.

```
removeCapsule(capsIRB,"base_link",1)
updatePose(capsIRB,"tool0",trvec2tform([-0.05 0 0]),1)
updateGeometry(capsIRB,"tool0",[.1 0.01],1)
show(capsIRB,homeConfiguration(capsIRB.RigidBodyTree));
```



Version History

Introduced in R2022b

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

Objects

`rigidBodyTree` | `collisionCapsule`

Functions

`fitCollisionCapsule` | `checkCollision` | `genspheres`

chompSolverOptions

Solver options for CHOMP motion planner

Description

The `chompSolverOptions` object stores Covariant Hamiltonian Optimization for Motion Planning (CHOMP) solver options that you can use to change the behavior of the solver.

Creation

Syntax

```
OPTS = chompSolverOptions
OPTS = chompSolverOptions(Name=Value)
```

Description

`OPTS = chompSolverOptions` creates a solver options object `OPTS` that you can use to set options for a `manipulatorCHOMP` object to optimize a trajectory.

`OPTS = chompSolverOptions(Name=Value)` specifies properties using one or more name-value arguments.

Properties

`Verbosity` — Verbosity of diagnostic output
`"none"` (default) | `"concise"` | `"detailed"`

Verbosity of the diagnostic output to the command window from the `optimize` function, specified as one of these options:

- `"none"` — Output no information.
- `"concise"` — Output the objective function results.
- `"detailed"` — Output the smoothness costs, collision costs, and the objective function results.

Data Types: `char` | `string`

`FunctionTolerance` — Termination tolerance on objective function
`1e-2` (default) | nonnegative numeric scalar

Termination tolerance on the objective function, specified as a nonnegative numeric scalar. The termination tolerance is the absolute relative change of the objective function between consecutive iterations:

$$\left|\frac{f_{i-1} - f_i}{f_i}\right|,$$

where *f* is the objective function and *i* is an iteration.

MaxTime — Maximum time to optimize trajectory
10 (default) | nonnegative numeric scalar

Maximum time to optimize the trajectory, specified as a nonnegative numeric scalar. Units are in seconds.

MaxIterations — Maximum number of iterations to optimize trajectory
50 (default) | nonnegative integer scalar

Maximum number of iterations to optimize the trajectory, specified as a nonnegative integer scalar.

LearningRate — Learning rate at which to update objective function
5 (default) | positive numeric scalar

Learning rate at which to update the objective function, specified as a positive numeric scalar. Units are in seconds.

Examples

Optimize Collision-Free Trajectory with CHOMP

Load a robot model into the workspace, and create a CHOMP solver.

```
robot = loadrobot("kinovaGen3",DataFormat="row");
chomp = manipulatorCHOMP(robot);
```

Create spheres to represent obstacles, and add them to the CHOMP solver.

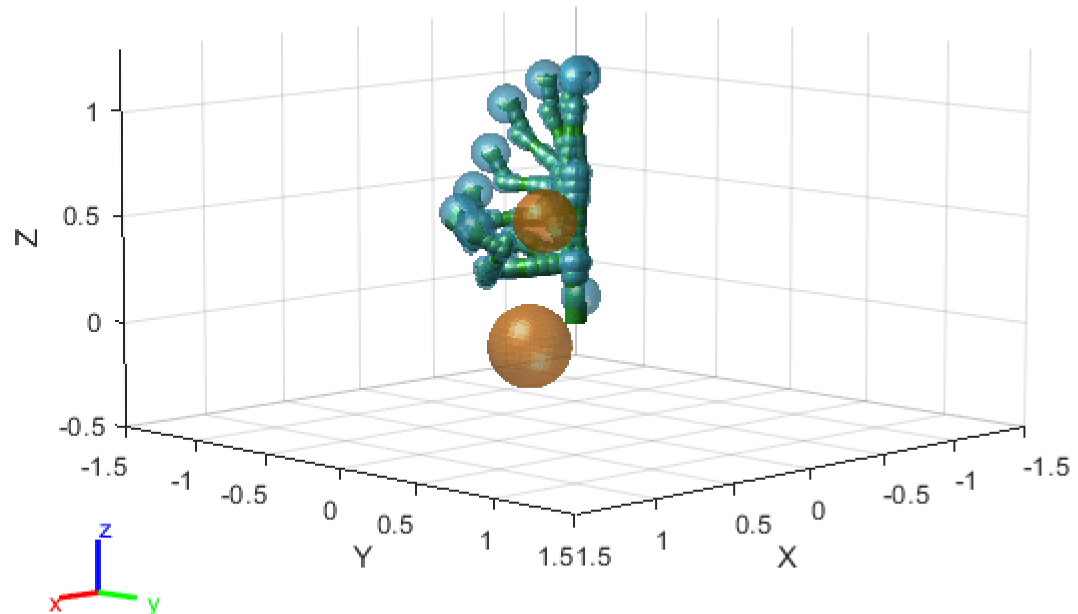
```
env = [0.20 0.2 -0.1 -0.1;   % sphere, radius 0.20 at (0.2,-0.1,-0.1)
       0.15 0.2  0.0  0.5]'; % sphere, radius 0.15 at (0.2,0.0,0.5)
chomp.SphericalObstacles = env;
```

To prioritize a collision-free trajectory, set the smoothness cost weight to a lower value than the collision cost weight. Then add the options to the CHOMP solver.

```
chomp.SmoothnessOptions = chompSmoothnessOptions(SmoothnessCostWeight=1e-3);
chomp.CollisionOptions = chompCollisionOptions(CollisionCostWeight=10);
chomp.SolverOptions = chompSolverOptions(Verbosity="none",LearningRate=7.0);
```

Initialize a trajectory, optimize it using the CHOMP solver, and show the waypoints in a figure.

```
startconfig = homeConfiguration(robot);
goalconfig = [0.5 1.75 -2.25 2.0 0.3 -1.65 -0.4];
timepoints = [0 5];
timestep = 0.1;
trajtype = "minjerkpolytraj";
[wptsamples,tsamples] = optimize(chomp, ...
    [startconfig; goalconfig], ...
    timepoints, ...
    timestep, ...
    InitialTrajectoryFitType=trajtype);
show(chomp,wptsamples,NumSamples=10);
zlim([-0.5 1.3])
```



Version History

Introduced in R2023a

References

[1] Ratliff, Nathan, Siddhartha Srinivasa, Matt Zucker, and Andrew Bagnell. “CHOMP: Gradient Optimization Techniques for Efficient Motion Planning.” In *2009 IEEE International Conference on Robotics and Automation*, 489–94. Kobe, Japan: IEEE, 2009. https://doi.org/10.1109/ROBOT.2009.5152817.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

`optimize` | `chompSmoothnessOptions` | `chompCollisionOptions` | `manipulatorCHOMP`

chompSmoothnessOptions

Smoothness options for CHOMP trajectories

Description

The `chompSmoothnessOptions` object stores smoothness options that determine how to weight smoothness costs for trajectories generated using Covariant Hamiltonian Optimization for Motion Planning (CHOMP). Use this object to optimize the smoothness of a trajectory generated using CHOMP.

Creation

Syntax

```
OPTS = chompSmoothnessOptions
OPTS = chompSmoothnessOptions(Name=Value)
```

Description

`OPTS = chompSmoothnessOptions` creates a smoothness options object `OPTS` that you can use to optimize the smoothness of a trajectory using CHOMP.

`OPTS = chompSmoothnessOptions(Name=Value)` specifies properties using one or more name-value arguments.

Properties

`VelocitySmoothnessWeight` — Weight on velocity smoothness of trajectory
`1` (default) | nonnegative numeric scalar

Weight on the velocity smoothness cost of the trajectory, specified as a nonnegative numeric scalar. This is a weight on the cost obtained by the summation of squared velocity along the trajectory.

`AccelerationSmoothnessWeight` — Weight on acceleration smoothness of trajectory
`1` (default) | nonnegative numeric scalar

Weight on the acceleration smoothness cost of the trajectory, specified as a nonnegative numeric scalar. This is a weight on the cost obtained by the summation of squared acceleration along the trajectory.

`JerkSmoothnessWeight` — Weight on jerk smoothness of trajectory
`1` (default) | nonnegative numeric scalar

Weight on the jerk smoothness cost of the trajectory, specified as a nonnegative numeric scalar. This is a weight on the cost obtained by the summation of squared jerk along the trajectory.

`SmoothnessCostWeight` — Weight on overall smoothness of trajectory
`1e-3` (default) | nonnegative numeric scalar

Weight on the overall smoothness of the trajectory, specified as a nonnegative numeric scalar. This is a weight on the cost obtained by the summation of the velocity, jerk, and acceleration smoothness costs.

Examples

Optimize Collision-Free Trajectory with CHOMP

Load a robot model into the workspace, and create a CHOMP solver.

```
robot = loadrobot("kinovaGen3",DataFormat="row");
chomp = manipulatorCHOMP(robot);
```

Create spheres to represent obstacles, and add them to the CHOMP solver.

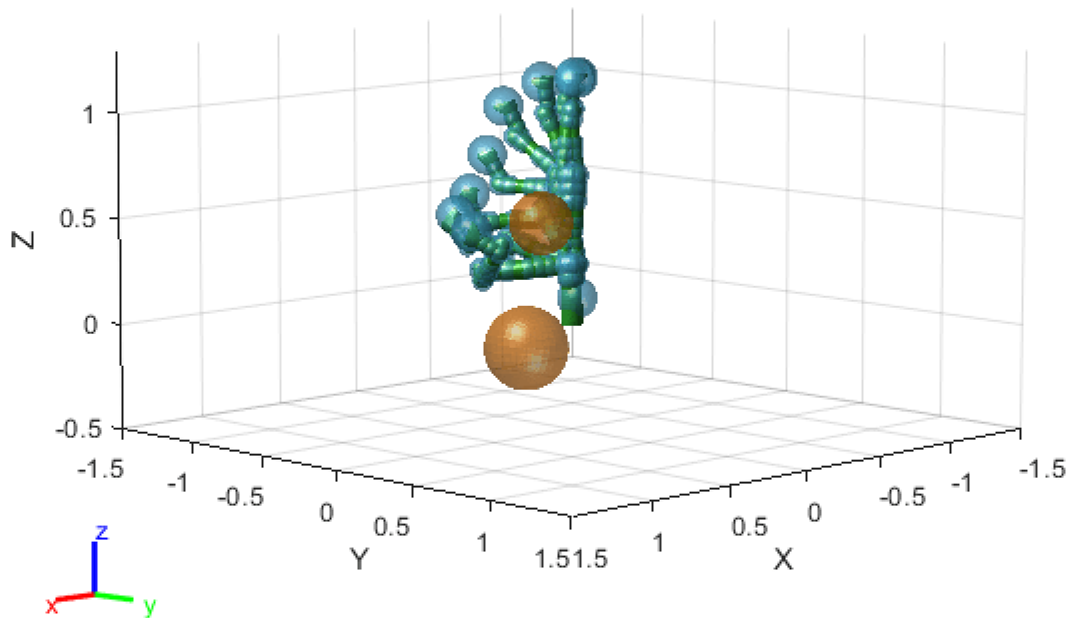
```
env = [0.20 0.2 -0.1 -0.1;   % sphere, radius 0.20 at (0.2,-0.1,-0.1)
       0.15 0.2  0.0  0.5]'; % sphere, radius 0.15 at (0.2,0.0,0.5)
chomp.SphericalObstacles = env;
```

To prioritize a collision-free trajectory, set the smoothness cost weight to a lower value than the collision cost weight. Then add the options to the CHOMP solver.

```
chomp.SmoothnessOptions = chompSmoothnessOptions(SmoothnessCostWeight=1e-3);
chomp.CollisionOptions = chompCollisionOptions(CollisionCostWeight=10);
chomp.SolverOptions = chompSolverOptions(Verbosity="none",LearningRate=7.0);
```

Initialize a trajectory, optimize it using the CHOMP solver, and show the waypoints in a figure.

```
startconfig = homeConfiguration(robot);
goalconfig = [0.5 1.75 -2.25 2.0 0.3 -1.65 -0.4];
timepoints = [0 5];
timestep = 0.1;
trajtype = "minjerkpolytraj";
[wptsamples,tsamples] = optimize(chomp, ...
    [startconfig; goalconfig], ...
    timepoints, ...
    timestep, ...
    InitialTrajectoryFitType=trajtype);
show(chomp,wptsamples,NumSamples=10);
zlim([-0.5 1.3])
```



Version History

Introduced in R2023a

References

[1] Ratliff, Nathan, Siddhartha Srinivasa, Matt Zucker, and Andrew Bagnell. “CHOMP: Gradient Optimization Techniques for Efficient Motion Planning.” In *2009 IEEE International Conference on Robotics and Automation*, 489–94. Kobe, Japan: IEEE, 2009. https://doi.org/10.1109/ROBOT.2009.5152817.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`optimize` | `chompSolverOptions` | `chompCollisionOptions` | `manipulatorCHOMP`

chompCollisionOptions

Collision options for CHOMP trajectories

Description

The `chompCollisionOptions` object stores collision options for Covariant Hamiltonian Optimization for Motion Planning (CHOMP) trajectories. Use this object to optimize a CHOMP trajectory to avoid collisions.

Creation

Syntax

```
OPTS = chompCollisionOptions
OPTS = chompCollisionOptions(Name=Value)
```

Description

`OPTS = chompCollisionOptions` creates a collision options object `OPTS` that you can use to optimize a CHOMP trajectory to avoid collisions.

`OPTS = chompCollisionOptions(Name=Value)` specifies properties using one or more name-value arguments.

Properties

`CollisionClearance` — Minimum distance to maintain from obstacles
`1e-3` (default) | positive numeric scalar

Minimum distance to maintain from obstacles, specified as a positive numeric scalar.

`SkippedSelfCollisions` — Body pairs to omit from self-collision costs
`"parent"` (default) | `"adjacent"`

Body pairs to omit from self-collision costs, specified as either `"parent"` or `"adjacent"`.

- `"parent"` — Skip collision checking between child and parent bodies.
- `"adjacent"` — Skip collision checking between bodies on adjacent indices.

Data Types: `char` | `string`

`IgnoreSelfCollision` — Weight on cost of ignoring self-collision avoidance
`1` (default) | positive numeric scalar

Weight on cost of ignoring self-collision avoidance, specified as a numeric scalar.

`CollisionCostWeight` — Weight on cost of collision
`10` (default) | positive numeric scalar

Weight on the cost of collision, specified as a positive numeric scalar.

Examples

Optimize Collision-Free Trajectory with CHOMP

Load a robot model into the workspace, and create a CHOMP solver.

```
robot = loadrobot("kinovaGen3",DataFormat="row");
chomp = manipulatorCHOMP(robot);
```

Create spheres to represent obstacles, and add them to the CHOMP solver.

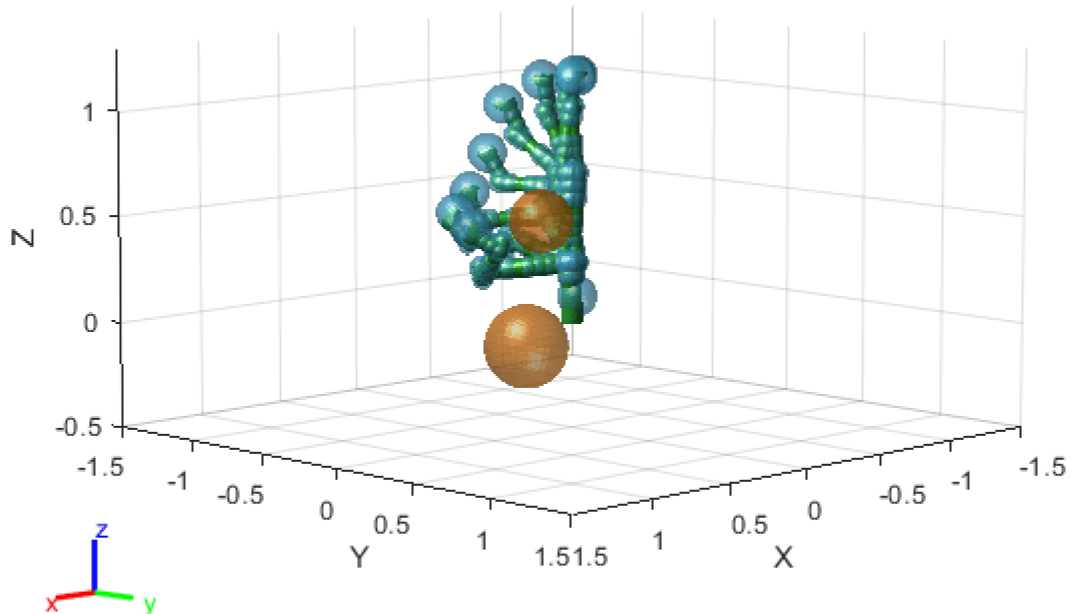
```
env = [0.20 0.2 -0.1 -0.1;   % sphere, radius 0.20 at (0.2,-0.1,-0.1)
       0.15 0.2  0.0  0.5]'; % sphere, radius 0.15 at (0.2,0.0,0.5)
chomp.SphericalObstacles = env;
```

To prioritize a collision-free trajectory, set the smoothness cost weight to a lower value than the collision cost weight. Then add the options to the CHOMP solver.

```
chomp.SmoothnessOptions = chompSmoothnessOptions(SmoothnessCostWeight=1e-3);
chomp.CollisionOptions = chompCollisionOptions(CollisionCostWeight=10);
chomp.SolverOptions = chompSolverOptions(Verbosity="none",LearningRate=7.0);
```

Initialize a trajectory, optimize it using the CHOMP solver, and show the waypoints in a figure.

```
startconfig = homeConfiguration(robot);
goalconfig = [0.5 1.75 -2.25 2.0 0.3 -1.65 -0.4];
timepoints = [0 5];
timestep = 0.1;
trajtype = "minjerkpolytraj";
[wptsamples,tsamples] = optimize(chomp, ...
    [startconfig; goalconfig], ...
    timepoints, ...
    timestep, ...
    InitialTrajectoryFitType=trajtype);
show(chomp,wptsamples,NumSamples=10);
zlim([-0.5 1.3])
```



Version History

Introduced in R2023a

References

[1] Ratliff, Nathan, Siddhartha Srinivasa, Matt Zucker, and Andrew Bagnell. "CHOMP: Gradient Optimization Techniques for Efficient Motion Planning." In *2009 IEEE International Conference on Robotics and Automation*, 489–94. Kobe, Japan: IEEE, 2009. https://doi.org/10.1109/ROBOT.2009.5152817.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

`optimize` | `chompSolverOptions` | `chompSmoothnessOptions` | `manipulatorCHOMP`

collisionBox

Create box collision geometry

Description

Use `collisionBox` to create a box collision geometry centered at the origin.

Creation

Syntax

`BOX = collisionBox(X,Y,Z)`

Description

`BOX = collisionBox(X,Y,Z)` creates an axis-aligned box collision geometry centered at the origin with X, Y, and Z as its side lengths along the corresponding axes in the geometry-fixed frame. By default, the geometry-fixed frame collocates with the world frame.

Properties

X — Side length of box geometry
positive scalar

Side length of box geometry along the *x*-axis, specified as a positive scalar. Units are in meters.

Data Types: `double`

Y — Side length of box geometry
positive scalar

Side length of box geometry along the *y*-axis, specified as a positive scalar. Units are in meters.

Data Types: `double`

Z — Side length of box geometry
positive scalar

Side length of box geometry along the *z*-axis, specified as a positive scalar. Units are in meters.

Data Types: `double`

Pose — Pose
`eye(4)` (default) | real-valued 4-by-4 matrix | `se3` object

Pose of the collision geometry relative to the world frame, specified as a 4-by-4 homogeneous matrix or an `se3` object. You can change the pose after you create the collision geometry.

Note Note that when the pose is specified as an `se3` object, the `Pose` property stores the pose as a numeric 4-by-4 matrix.

Data Types: `single` | `double`

Object Functions

show	Show collision geometry
convertToCollisionMesh	Convert collision primitive geometry into collision mesh geometry
fitCollisionCapsule	Fit collision capsule around collision geometry

Examples

Create and Visualize Box Collision Geometry

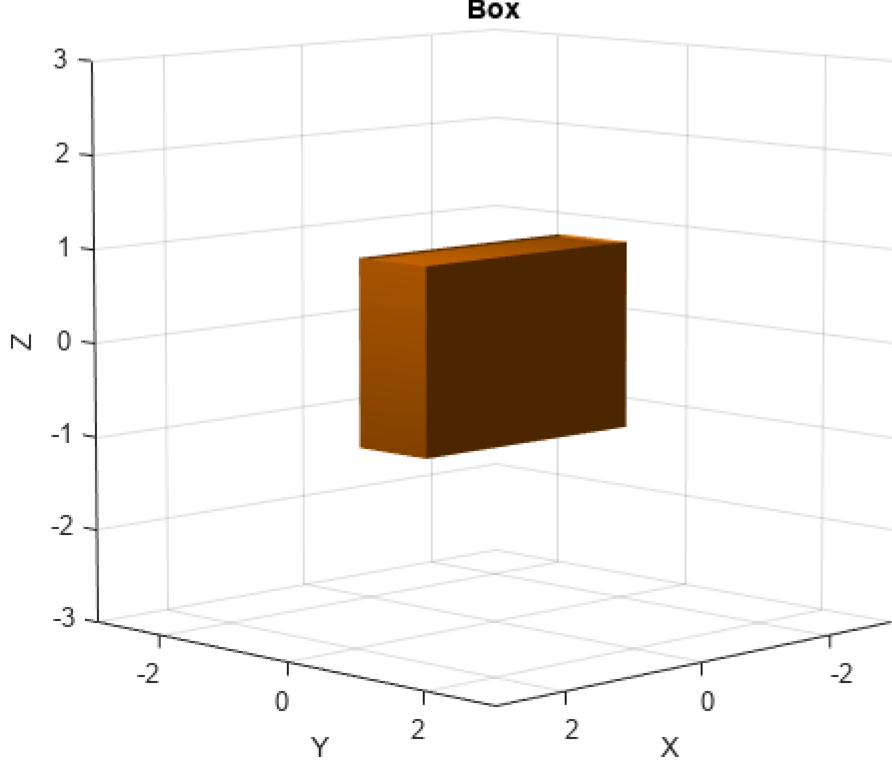
Create a box collision geometry centered at the origin. The side lengths in the *x*-, *y*-, and *z*-directions are 3, 1, and 2 meters, respectively.

```
box = collisionBox(3,1,2)
```

```
box = 
  collisionBox with properties:

       X: 3
       Y: 1
       Z: 2
    Pose: [4x4 double]
```

Visualize the box.

```
show(box)
title('Box')
```

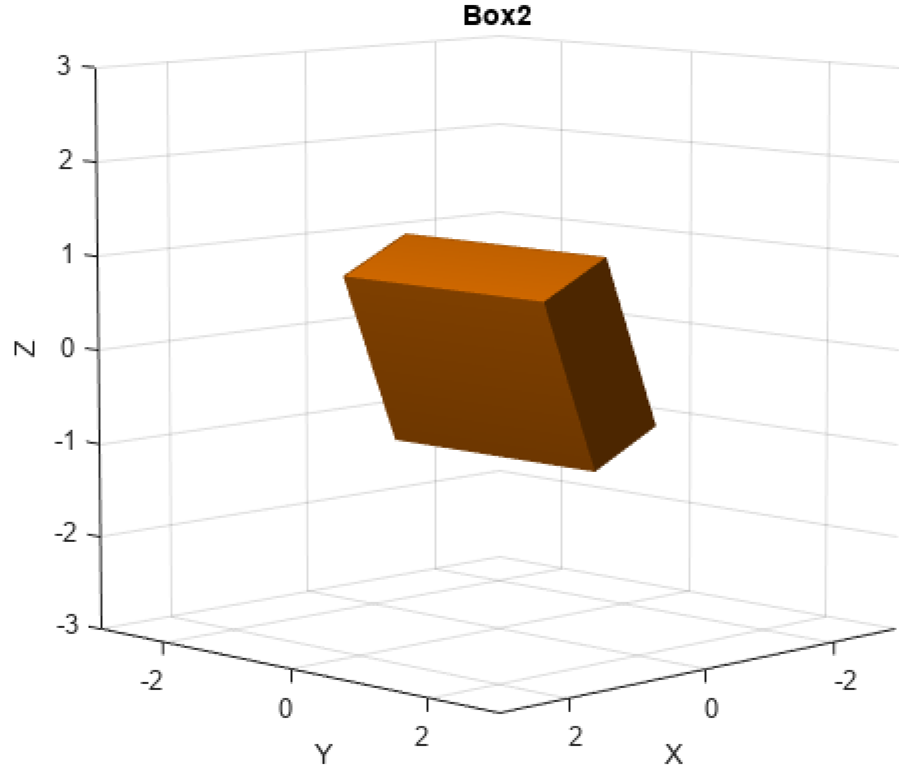


Create two homogeneous transformation matrices. The first matrix is a rotation about the *z*-axis by $\pi/2$ radians, and the second matrix is a rotation about the *x*-axis of $\pi/8$ radians.

```
matZ = axang2tform([0 0 1 pi/2]);
matX = axang2tform([1 0 0 pi/8]);
```

Create a second box collision geometry with the same dimensions as the first. Change its pose to the product of the two matrices. The product corresponds to first rotation about the z-axis followed by rotation about the x-axis. Visualize the result.

```
box2 = collisionBox(3,1,2);
box2.Pose = matZ*matX;
show(box2)
title('Box2')
```



Version History

Introduced in R2019b

R2023a: Pose property supports se3 transformation object

You can now specify the `Pose` property of `collisionBox` as an `se3` transformation object.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

Objects
`collisionCylinder` | `collisionMesh` | `collisionSphere` | `collisionCapsule`

Functions
`checkCollision` | `fitCollisionCapsule`

Topics

“Generate Code for Manipulator Motion Planning in Perceived Environment”

collisionCapsule

Capsule primitive collision geometry

Description

The `collisionCapsule` object is a capsule primitive collision geometry defined by a radius and length. The central line segment of the capsule aligns with its *z*-axis. The origin of the body-fixed frame is at the midpoint of the central line segment of the capsule.

Creation

Syntax

`CAPS = collisionCapsule(radius,length)`

Description

`CAPS = collisionCapsule(radius,length)` creates a capsule primitive with the specified radius `radius` and length `length`. The `radius` and `length` arguments set the `Radius` and `Length` properties, respectively

Properties

`Radius` — Radius of spherical ends of capsule
nonnegative scalar

Radius of the spherical ends of the capsule, specified as a nonnegative scalar. Units are in meters.

Example: `2.5`

`Length` — Length of central line segment of capsule
nonnegative scalar

Length of the central line segment of the capsule, specified as a nonnegative scalar. Units are in meters.

Note This is not the length from end-to-end of the capsule. The total length of the capsule is `Length +2(Radius)`.

Example: `4.5`

Pose — Pose
`eye(4)` (default) | real-valued 4-by-4 matrix | `se3` object

Pose of the collision geometry relative to the world frame, specified as a 4-by-4 homogeneous matrix or an `se3` object. You can change the pose after you create the collision geometry.

Note Note that when the pose is specified as an `se3` object, the `Pose` property stores the pose as a numeric 4-by-4 matrix.

Data Types: `single` | `double`

Object Functions

checkCollision	Check if two geometries are in collision
convertToCollisionMesh	Convert collision primitive geometry into collision mesh geometry
genspheres	Generate spheres along central line segment of capsule
show	Show collision geometry

Examples

Generate Collision Spheres Inside Collision Capsule

Create a collision capsule with a radius of `2` and length of `10`. Visualize the capsule.

```
cCapsule = collisionCapsule(2,10);
[~,p] = show(cCapsule);
```

Generate spheres at ratios `0.0`, `0.5`, and `1.0` of the capsule length.

```
spheres = genspheres(cCapsule,linspace(0,1,3));
```

Display the positions of the spheres.

```
for i = 1:length(spheres)
    disp(tform2trvec(spheres{i}.Pose))
end
```

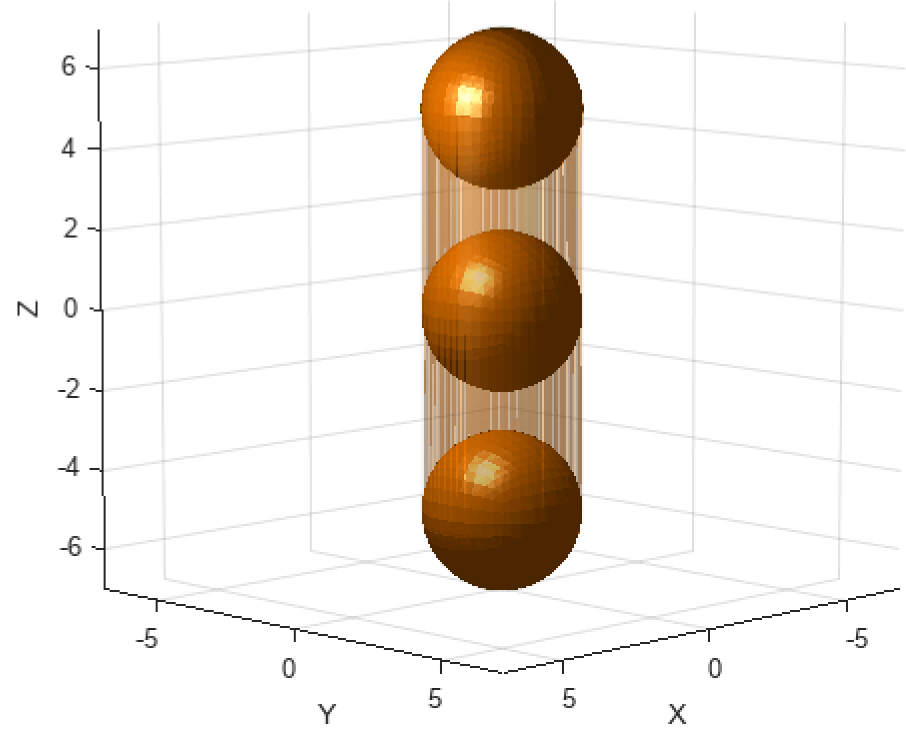
```
     0     0    -5

     0     0     0

     0     0     5
```

Set the face and edge alphas of the capsule to low values. This ensures that both the spheres are visible when you add them to the figure.

```
p.FaceAlpha = 0.4;
p.EdgeAlpha = 0.01;
hold on
```

Display the generated spheres on the capsule.

```
cellfun(@show,spheres);
```



Version History

Introduced in R2022b

R2023a: Pose property supports se3 transformation object

You can now specify the `Pose` property of `collisionCapsule` as an `se3` transformation object.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

Objects
`capsuleApproximation` | `collisionBox` | `collisionCylinder` | `collisionSphere` | `collisionMesh`

Functions
`checkCollision` | `fitCollisionCapsule`

collisionCylinder

Create collision cylinder geometry

Description

Use `collisionCylinder` to create a cylinder collision geometry centered at the origin.

Creation

Syntax

`CYL = collisionCylinder(Radius,Length)`

Description

`CYL = collisionCylinder(Radius,Length)` creates a cylinder collision geometry with a specified Radius and Length. The cylinder is axis-aligned with its own body-fixed frame. The side of the cylinder lies along the *z*-axis. The origin of the body-fixed frame is at the center of the cylinder.

Properties

`Radius` — Radius
positive scalar

Radius of cylinder, specified as a positive scalar. Units are in meters.

Data Types: `double`

`Length` — Length
positive scalar

Length of cylinder, specified as a positive scalar. Units are in meters.

Data Types: `double`

`Pose` — Pose
`eye(4)` (default) | real-valued 4-by-4 matrix | `se3` object

Pose of the collision geometry relative to the world frame, specified as a 4-by-4 homogeneous matrix or an `se3` object. You can change the pose after you create the collision geometry.

Note Note that when the pose is specified as an `se3` object, the `Pose` property stores the pose as a numeric 4-by-4 matrix.

Data Types: `single` | `double`

Object Functions

show	Show collision geometry
convertToCollisionMesh	Convert collision primitive geometry into collision mesh geometry
fitCollisionCapsule	Fit collision capsule around collision geometry

Examples

Create and Visualize Cylinder Collision Geometry

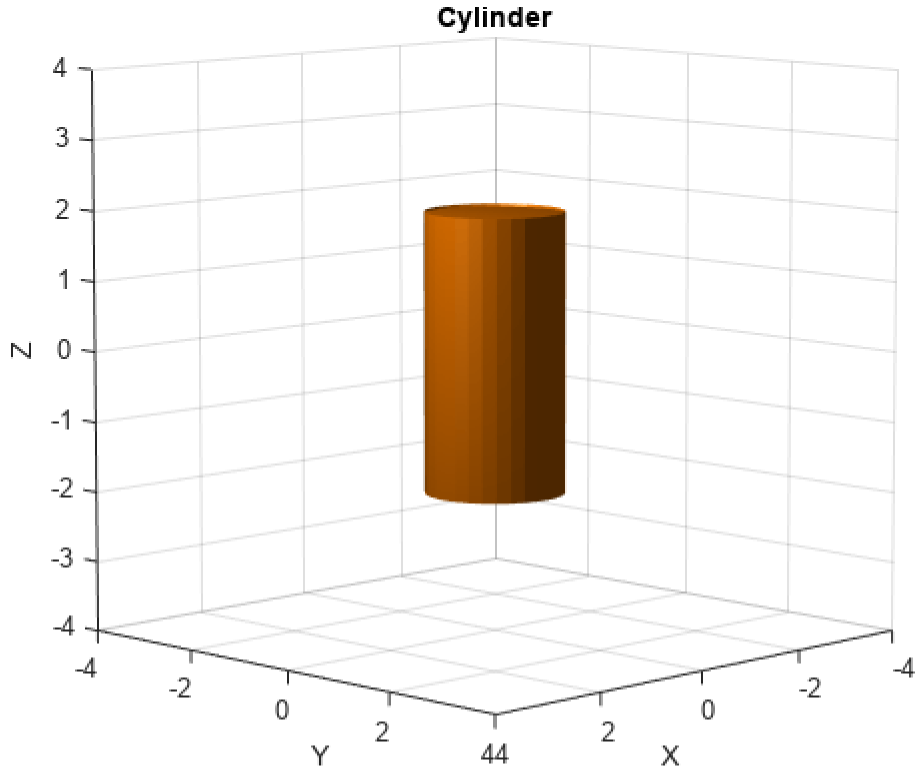
Create a cylinder collision geometry centered at the origin. The cylinder is 4 meters long with a radius of 1 meter.

```
rad = 1;
len = 4;
cyl = collisionCylinder(rad,len)
```

```
cyl = 
  collisionCylinder with properties:

    Radius: 1
    Length: 4
      Pose: [4x4 double]
```

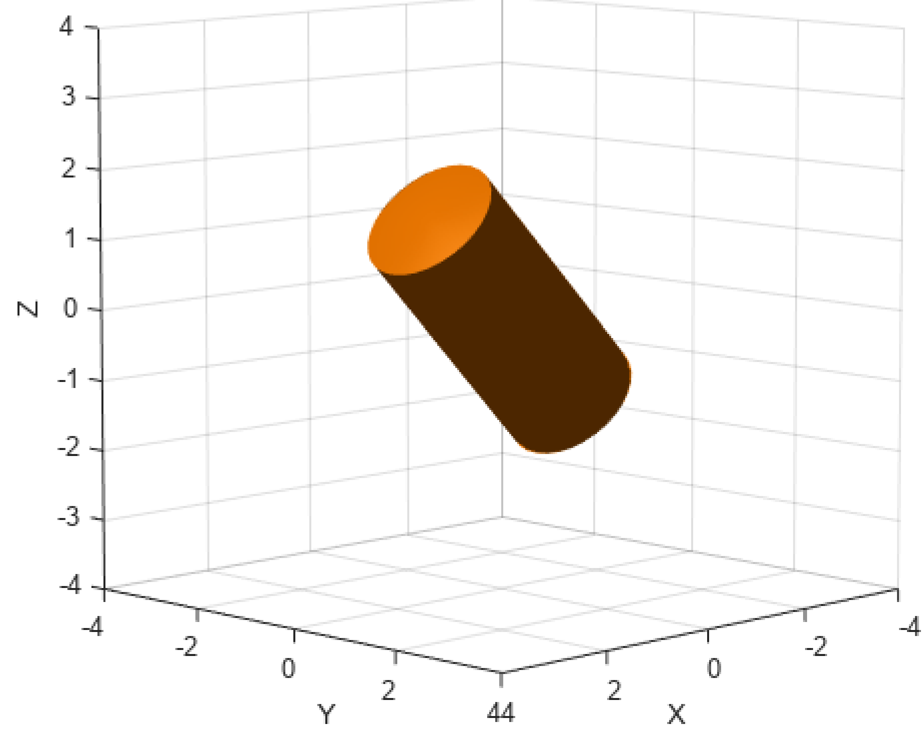
Visualize the cylinder.

```
show(cyl)
title('Cylinder')
```



Create a homogeneous transformation that corresponds to a clockwise rotation of $\pi/4$ radians about the *y*-axis. Set the cylinder pose to the new matrix. Show the cylinder.

```
ang = pi/4;
mat = axang2tform([0 1 0 ang]);
cyl.Pose = mat;
show(cyl)
```



Version History

Introduced in R2019b

R2023a: Pose property supports se3 transformation object

You can now specify the `Pose` property of `collisionCylinder` as an `se3` transformation object.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

Objects
`collisionBox` | `collisionMesh` | `collisionSphere` | `collisionCapsule`

Functions
`checkCollision` | `fitCollisionCapsule`

Topics

“Generate Code for Manipulator Motion Planning in Perceived Environment”

collisionMesh

Create convex mesh collision geometry

Description

Use `collisionMesh` to create a collision geometry as a convex mesh.

Creation

Syntax

`MSH = collisionMesh(Vertices)`

Description

`MSH = collisionMesh(Vertices)` creates a convex mesh collision geometry from the list of 3-D Vertices. The vertices are specified relative to a frame of choice (collision geometry frame). By default, the collision geometry frame collocates with the world frame.

Properties

`Vertices` — Vertices
3-D real-valued array

Vertices of a mesh, specified as an *N*-by-3 array, where *N* is the number of vertices. Each row of `Vertices` represents the coordinates of a point in 3-D space. Note that some of the points can be inside the constructed convex mesh.

Data Types: `double`

Pose — Pose
`eye(4)` (default) | real-valued 4-by-4 matrix | `se3` object

Pose of the collision geometry relative to the world frame, specified as a 4-by-4 homogeneous matrix or an `se3` object. You can change the pose after you create the collision geometry.

Note Note that when the pose is specified as an `se3` object, the `Pose` property stores the pose as a numeric 4-by-4 matrix.

Data Types: `single` | `double`

Object Functions

show	Show collision geometry
fitCollisionCapsule	Fit collision capsule around collision geometry

Examples

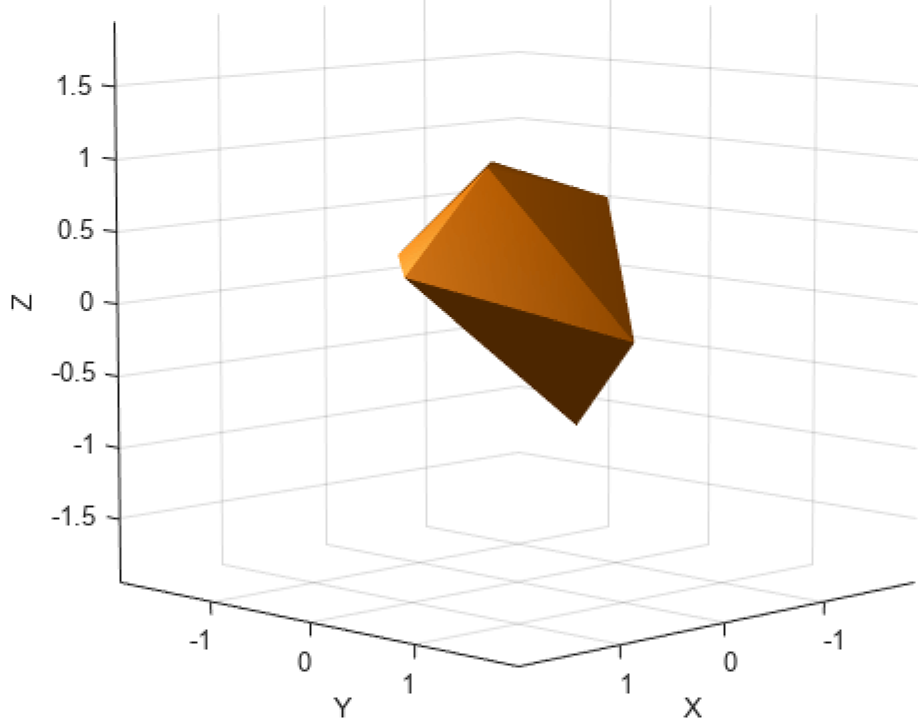
Create and Visualize Mesh Collision Geometry

Create an array consisting of the coordinates of ten points randomly chosen on the unit sphere. For reproducibility, set the random seed to the default value.

```
rng default
n = 10;
pts = zeros(n,3);
for k = 1:n
    ph = 2*pi*rand(1);
    th = pi*rand(1);
    pts(k,:) = [cos(th)*sin(ph) sin(th)*sin(ph) cos(ph)];
end
```

Create a convex mesh collision geometry from the array. Visualize the collision geometry.

```
m = collisionMesh(pts);
show(m)
```



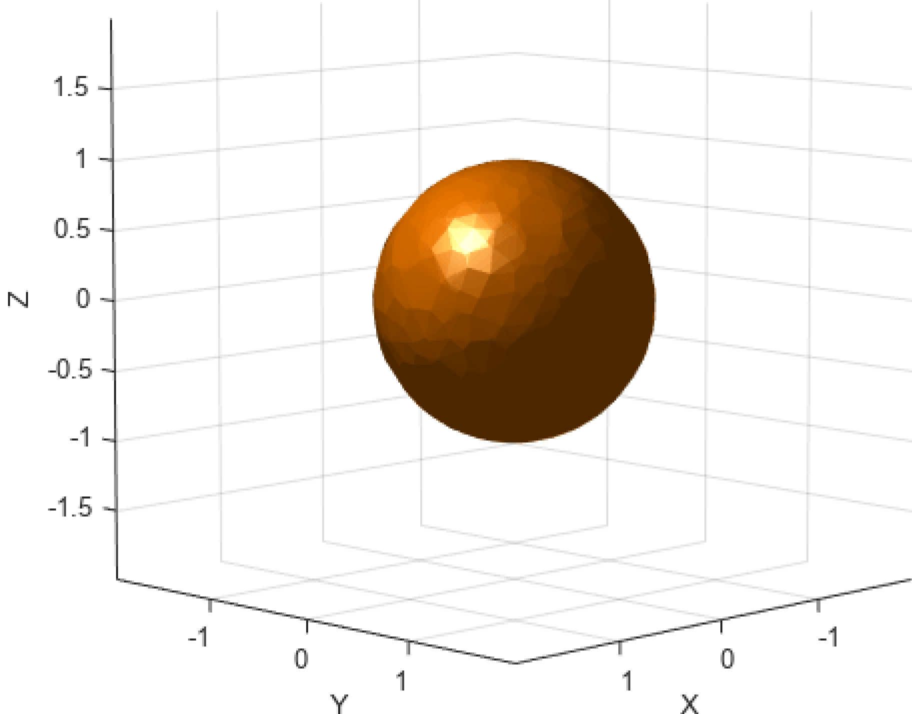
Create a second array similar to the first, but this time consisting of 1000 points randomly chosen on the unit sphere.

```
n = 1000;
pts2 = zeros(n,3);
```

```
for k = 1:n
    ph = 2*pi*rand(1);
    th = pi*rand(1);
    pts2(k,:) = [cos(th)*sin(ph) sin(th)*sin(ph) cos(ph)];
end
```

Create and visualize a mesh collision geometry from the array. Observe that choosing more points on the sphere results in a sphere-like mesh.

```
m2 = collisionMesh(pts2);
show(m2)
```



Create an array consisting of the coordinates of the eight corners of a cube. The cube is centered at the origin and has side length 4.

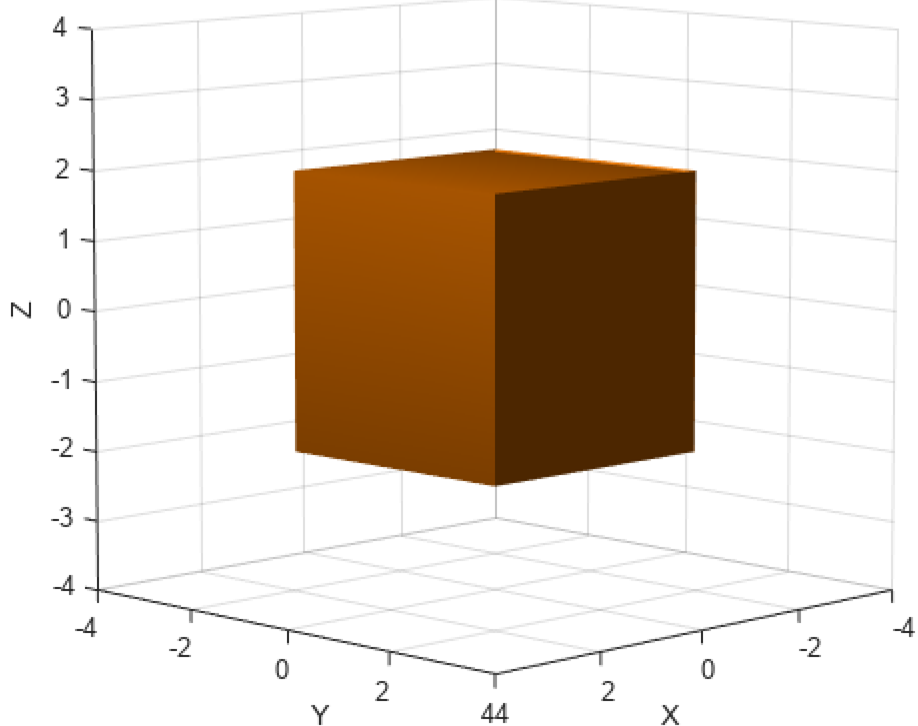
```
cubeCorners = [-2 -2 -2 ; -2 2 -2 ; 2 -2 -2 ; 2 2 -2 ;...
    -2 -2 2 ; -2 2 2 ; 2 -2 2 ; 2 2 2]
```

```
cubeCorners = 8×3

    -2    -2    -2
    -2     2    -2
     2    -2    -2
     2     2    -2
    -2    -2     2
    -2     2     2
     2    -2     2
```

```
     2     2     2
```

Append `cubeCorners` to `pts2`. Create and visualize the mesh collision geometry from the new array. Because the cube contains the sphere, the sphere points that are interior to the cube are disregarded when creating the geometry.

```
pts3 = [pts2;cubeCorners];
m3 = collisionMesh(pts3);
show(m3)
```



Version History

Introduced in R2019b

R2023a: Pose property supports se3 transformation object

You can now specify the `Pose` property of `collisionMesh` as an `se3` transformation object.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

Objects
collisionBox | collisionCylinder | collisionSphere | collisionCapsule

Functions
checkCollision | fitCollisionCapsule

Topics
"Generate Code for Manipulator Motion Planning in Perceived Environment"

collisionSphere

Create sphere collision geometry

Description

Use `collisionSphere` to create a sphere collision geometry centered at the origin.

Creation

Syntax

`sph = collisionSphere(Radius)`

Description

`sph = collisionSphere(Radius)` creates a sphere collision geometry with a specified Radius. The origin of the geometry-fixed frame is at the center of the sphere.

Properties

`Radius` — Radius
positive scalar

Radius of sphere, specified as a positive scalar. Units are in meters.

Data Types: `double`

Pose — Pose
`eye(4)` (default) | real-valued 4-by-4 matrix | `se3` object

Pose of the collision geometry relative to the world frame, specified as a 4-by-4 homogeneous matrix or an `se3` object. You can change the pose after you create the collision geometry.

Note Note that when the pose is specified as an `se3` object, the `Pose` property stores the pose as a numeric 4-by-4 matrix.

Data Types: `single` | `double`

Object Functions

show	Show collision geometry
convertToCollisionMesh	Convert collision primitive geometry into collision mesh geometry
fitCollisionCapsule	Fit collision capsule around collision geometry

Examples

Create and Visualize Sphere Collision Geometry

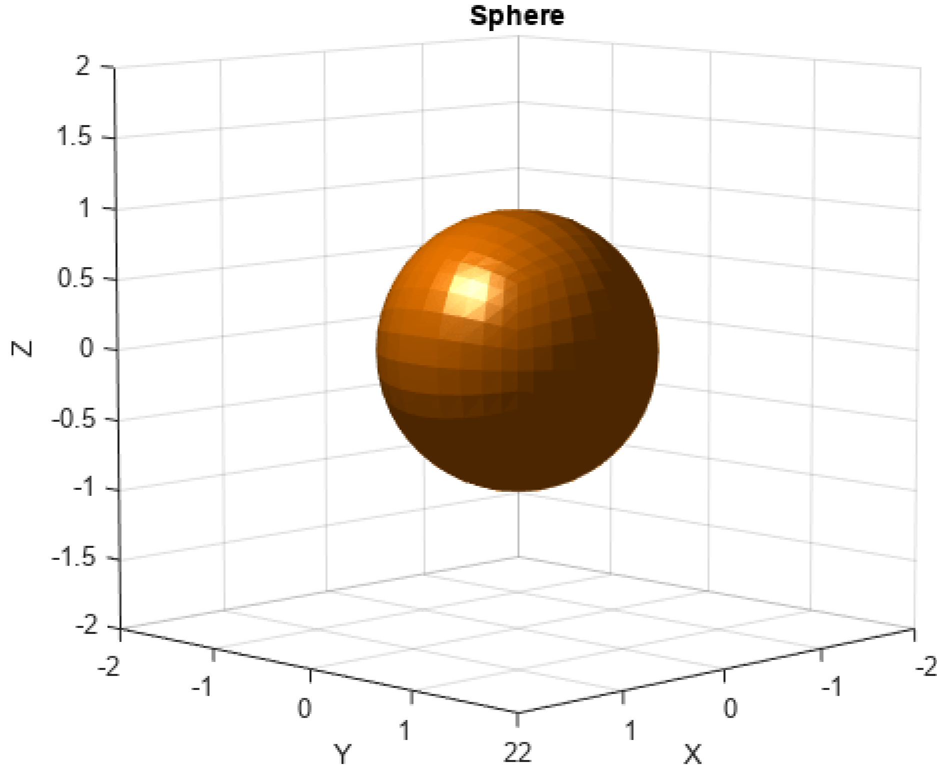
Create a sphere collision geometry centered at the origin. The sphere has a radius of 1 meter.

```
rad = 1;
sph = collisionSphere(rad)
```

```
sph = 
  collisionSphere with properties:

    Radius: 1
      Pose: [4x4 double]
```

Visualize the sphere.

```
show(sph)
title('Sphere')
```



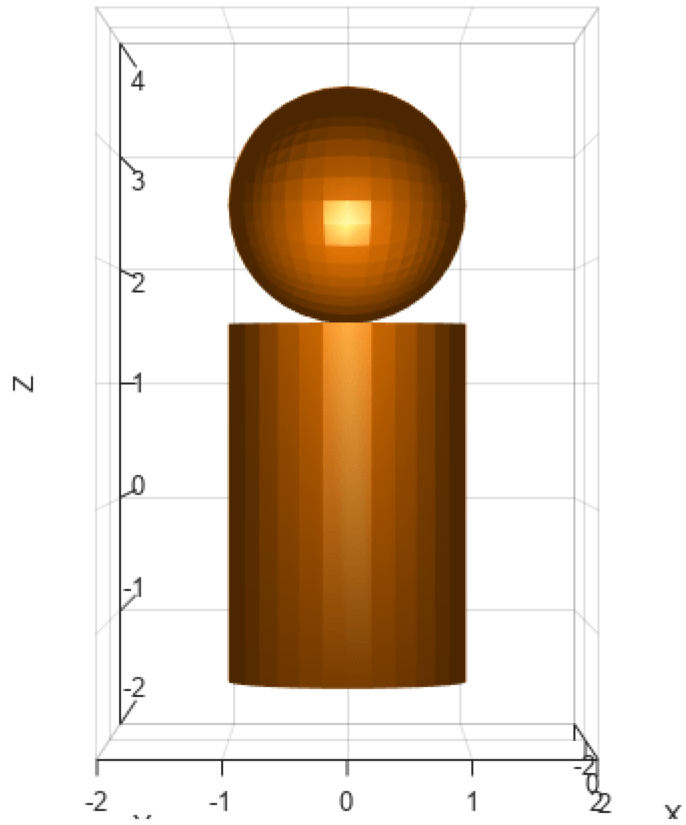
Create a cylinder collision geometry of radius 1 meter and length 3 meters.

```
cyl = collisionCylinder(1,3);
```

Create a homogeneous transformation that corresponds to a translation of 2.5 meters up the *z*-axis. Set the pose of the sphere to the matrix. Show the sphere and the cylinder.

```
mat = trvec2tform([0 0 2.5]);
sph.Pose = mat;
```

```
show(sph)
hold on
show(cyl)
view(90,0)
zlim([-2 4])
```



Version History

Introduced in R2019b

R2023a: Pose property supports se3 transformation object

You can now specify the `Pose` property of `collisionSphere` as an `se3` transformation object.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

Objects
`collisionBox` | `collisionCylinder` | `collisionMesh` | `collisionCapsule`

Functions
`checkCollision` | `fitCollisionCapsule`

Topics
"Generate Code for Manipulator Motion Planning in Perceived Environment"

constraintAiming

Create aiming constraint for pointing at a target location

Description

The `constraintAiming` object describes a constraint that requires the *z*-axis of one body (the end effector) to aim at a target point on another body (the reference body). This constraint is satisfied if the *z*-axis of the end-effector frame is within an angular tolerance in any direction of the line connecting the end-effector origin and the target point. The position of the target point is defined relative to the reference body.

Constraint objects are used in `generalizedInverseKinematics` objects to specify multiple kinematic constraints on a robot.

For an example that uses multiple constraint objects, see “Plan a Reaching Trajectory With Multiple Kinematic Constraints”.

Creation

Syntax

```
aimConst = constraintAiming(endeffector)
aimConst = constraintAiming(endeffector,Name=Value)
```

Description

`aimConst = constraintAiming(endeffector)` returns an aiming constraint object that represents a constraint on a body specified by `endeffector` and sets the `EndEffector` property.

`aimConst = constraintAiming(endeffector,Name=Value)` returns an aiming constraint object with each specified property name set to the specified value by one or more name-value pair arguments.

Properties

`EndEffector` — Name of the end effector
string scalar | character vector

Name of the end effector, specified as a string scalar or character vector. When using this constraint with `generalizedInverseKinematics`, the name must match a body specified in the robot model (`rigidBodyTree`).

Example: `"left_palm"`

Data Types: `char` | `string`

`ReferenceBody` — Name of the reference body frame
`''` (default) | string scalar | character vector

Name of the reference body frame, specified as a string scalar or character vector. The default `''` indicates that the constraint is relative to the base of the robot model. When using this constraint with `generalizedInverseKinematics`, the name must match a body specified in the robot model (`rigidBodyTree`).

Data Types: `char` | `string`

`TargetPoint` — Position of the target relative to the reference body
`[0 0 0]` (default) | `[x y z]` vector

Position of the target relative to the reference body, specified as an `[x y z]` vector. The constraint uses the line between the origin of the `EndEffector` body frame and this target point for maintaining the specified `AngularTolerance`.

`AngularTolerance` — Maximum allowed angle
`0` (default) | numeric scalar

Maximum allowed angle between the *z*-axis of the end-effector frame and the line connecting the end-effector origin to the target point, specified as a numeric scalar in radians.

`Weights` — Weight of the constraint
`1` (default) | numeric scalar

Weight of the constraint, specified as a numeric scalar. This weight is used with the `Weights` property of all the constraints specified in `generalizedInverseKinematics` to properly balance each constraint.

Examples

Plan a Reaching Trajectory With Multiple Kinematic Constraints

This example shows how to use generalized inverse kinematics to plan a joint-space trajectory for a robotic manipulator. It combines multiple constraints to generate a trajectory that guides the gripper to a cup resting on a table. These constraints ensure that the gripper approaches the cup in a straight line and that the gripper remains at a safe distance from the table, without requiring the poses of the gripper to be determined in advance.

Set Up the Robot Model

This example uses a model of the KUKA LBR iiwa, a 7 degree-of-freedom robot manipulator. `importrobot` generates a `rigidBodyTree` model from a description stored in a Unified Robot Description Format (URDF) file.

```
lbr = importrobot('iiwa14.urdf'); % 14 kg payload version
lbr.DataFormat = 'row';
gripper = 'iiwa_link_ee_kuka';
```

Define dimensions for the cup.

```
cupHeight = 0.2;
cupRadius = 0.05;
cupPosition = [-0.5, 0.5, cupHeight/2];
```

Add a fixed body to the robot model representing the center of the cup.

```
body = rigidBody('cupFrame');
setFixedTransform(body.Joint, trvec2tform(cupPosition))
addBody(lbr, body, lbr.BaseName);
```

Define the Planning Problem

The goal of this example is to generate a sequence of robot configurations that satisfy the following criteria:

- Start in the home configuration
- No abrupt changes in robot configuration
- Keep the gripper at least 5 cm above the "table" (z = 0)
- The gripper should be aligned with the cup as it approaches
- Finish with the gripper 5 cm from the center of the cup

This example utilizes constraint objects to generate robot configurations that satisfy these criteria. The generated trajectory consists of five configuration waypoints. The first waypoint, `q0`, is set as the home configuration. Pre-allocate the rest of the configurations in `qWaypoints` using `repmat`.

```
numWaypoints = 5;
q0 = homeConfiguration(lbr);
qWaypoints = repmat(q0, numWaypoints, 1);
```

Create a `generalizedInverseKinematics` solver that accepts the following constraint inputs:

- Cartesian bounds - Limits the height of the gripper
- A position target - Specifies the position of the cup relative to the gripper.
- An aiming constraint - Aligns the gripper with the cup axis
- An orientation target - Maintains a fixed orientation for the gripper while approaching the cup
- Joint position bounds - Limits the change in joint positions between waypoints.

```
gik = generalizedInverseKinematics('RigidBodyTree', lbr, ...
    'ConstraintInputs', {'cartesian','position','aiming','orientation','joint'})
```

```
gik = 
  generalizedInverseKinematics with properties:

      NumConstraints: 5
    ConstraintInputs: {'cartesian'  'position'  'aiming'  'orientation'  'joint'}
       RigidBodyTree: [1x1 rigidBodyTree]
     SolverAlgorithm: 'BFGSGradientProjection'
    SolverParameters: [1x1 struct]
```

Create Constraint Objects

Create the constraint objects that are passed as inputs to the solver. These object contain the parameters needed for each constraint. Modify these parameters between calls to the solver as necessary.

Create a Cartesian bounds constraint that requires the gripper to be at least 5 cm above the table (negative z direction). All other values are given as `inf` or `-inf`.

```
heightAboveTable = constraintCartesianBounds(gripper);
heightAboveTable.Bounds = [-inf, inf; ...
```

```
                           -inf, inf; ...
                           0.05, inf]
```

```
heightAboveTable = 
  constraintCartesianBounds with properties:

        EndEffector: 'iiwa_link_ee_kuka'
      ReferenceBody: ''
    TargetTransform: [4x4 double]
             Bounds: [3x2 double]
            Weights: [1 1 1]
```

Create a constraint on the position of the cup relative to the gripper, with a tolerance of 5 mm.

```
distanceFromCup = constraintPositionTarget('cupFrame');
distanceFromCup.ReferenceBody = gripper;
distanceFromCup.PositionTolerance = 0.005
```

```
distanceFromCup = 
  constraintPositionTarget with properties:

          EndEffector: 'cupFrame'
        ReferenceBody: 'iiwa_link_ee_kuka'
       TargetPosition: [0 0 0]
    PositionTolerance: 0.0050
              Weights: 1
```

Create an aiming constraint that requires the z-axis of the `iiwa_link_ee` frame to be approximately vertical, by placing the target far above the robot. The `iiwa_link_ee` frame is oriented such that this constraint aligns the gripper with the axis of the cup.

```
alignWithCup = constraintAiming('iiwa_link_ee');
alignWithCup.TargetPoint = [0, 0, 100]
```

```
alignWithCup = 
  constraintAiming with properties:

         EndEffector: 'iiwa_link_ee'
       ReferenceBody: ''
         TargetPoint: [0 0 100]
    AngularTolerance: 0
             Weights: 1
```

Create a joint position bounds constraint. Set the `Bounds` property of this constraint based on the previous configuration to limit the change in joint positions.

```
limitJointChange = constraintJointBounds(lbr)
```

```
limitJointChange = 
  constraintJointBounds with properties:

     Bounds: [7x2 double]
    Weights: [1 1 1 1 1 1 1]
```

Create an orientation constraint for the gripper with a tolerance of one degree. This constraint requires the orientation of the gripper to match the value specified by the `TargetOrientation` property. Use this constraint to fix the orientation of the gripper during the final approach to the cup.

```
fixOrientation = constraintOrientationTarget(gripper);
fixOrientation.OrientationTolerance = deg2rad(1)
```

```
fixOrientation = 
  constraintOrientationTarget with properties:

             EndEffector: 'iiwa_link_ee_kuka'
           ReferenceBody: ''
       TargetOrientation: [1 0 0 0]
    OrientationTolerance: 0.0175
                 Weights: 1
```

Find a Configuration That Points at the Cup

This configuration should place the gripper at a distance from the cup, so that the final approach can be made with the gripper properly aligned.

```
intermediateDistance = 0.3;
```

Constraint objects have a `Weights` property which determines how the solver treats conflicting constraints. Setting the weights of a constraint to zero disables the constraint. For this configuration, disable the joint position bounds and orientation constraint.

```
limitJointChange.Weights = zeros(size(limitJointChange.Weights));
fixOrientation.Weights = 0;
```

Set the target position for the cup in the gripper frame. The cup should lie on the z-axis of the gripper at the specified distance.

```
distanceFromCup.TargetPosition = [0,0,intermediateDistance];
```

Solve for the robot configuration that satisfies the input constraints using the `gik` solver. You must specify all the input constraints. Set that configuration as the second waypoint.

```
[qWaypoints(2,:),solutionInfo] = gik(q0, heightAboveTable, ...
                       distanceFromCup, alignWithCup, fixOrientation, ...
                       limitJointChange);
```

Find Configurations That Move Gripper to the Cup Along a Straight Line

Re-enable the joint position bound and orientation constraints.

```
limitJointChange.Weights = ones(size(limitJointChange.Weights));
fixOrientation.Weights = 1;
```

Disable the align-with-cup constraint, as the orientation constraint makes it redundant.

```
alignWithCup.Weights = 0;
```

Set the orientation constraint to hold the orientation based on the previous configuration (`qWaypoints(2,:)`). Get the transformation from the gripper to the base of the robot model. Convert the homogeneous transformation to a quaternion.

```
fixOrientation.TargetOrientation = ...
    tform2quat(getTransform(lbr,qWaypoints(2,:),gripper));
```

Define the distance between the cup and gripper for each waypoint

```
finalDistanceFromCup = 0.05;
distanceFromCupValues = linspace(intermediateDistance, finalDistanceFromCup, numWaypoints-1);
```

Define the maximum allowed change in joint positions between each waypoint.

```
maxJointChange = deg2rad(10);
```

Call the solver for each remaining waypoint.

```
for k = 3:numWaypoints
    % Update the target position.
    distanceFromCup.TargetPosition(3) = distanceFromCupValues(k-1);
    % Restrict the joint positions to lie close to their previous values.
    limitJointChange.Bounds = [qWaypoints(k-1,:)' - maxJointChange, ...
                               qWaypoints(k-1,:)' + maxJointChange];
    % Solve for a configuration and add it to the waypoints array.
    [qWaypoints(k,:),solutionInfo] = gik(qWaypoints(k-1,:), ...
                                         heightAboveTable, ...
                                         distanceFromCup, alignWithCup, ...
                                         fixOrientation, limitJointChange);
end
```

Visualize the Generated Trajectory

Interpolate between the waypoints to generate a smooth trajectory. Use `pchip` to avoid overshoots, which might violate the joint limits of the robot.

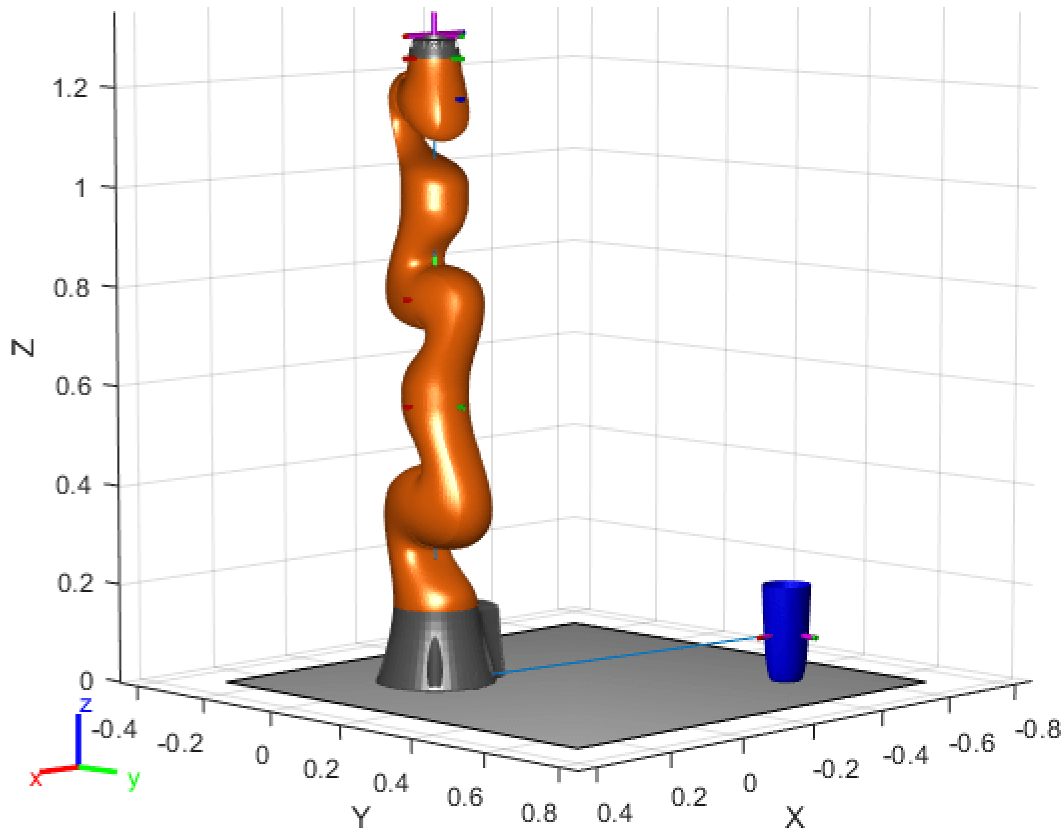
```
framerate = 15;
r = rateControl(framerate);
tFinal = 10;
tWaypoints = [0,linspace(tFinal/2,tFinal,size(qWaypoints,1)-1)];
numFrames = tFinal*framerate;
qInterp = pchip(tWaypoints,qWaypoints',linspace(0,tFinal,numFrames))';
```

Compute the gripper position for each interpolated configuration.

```
gripperPosition = zeros(numFrames,3);
for k = 1:numFrames
    gripperPosition(k,:) = tform2trvec(getTransform(lbr,qInterp(k,:), ...
                                                    gripper));
end
```

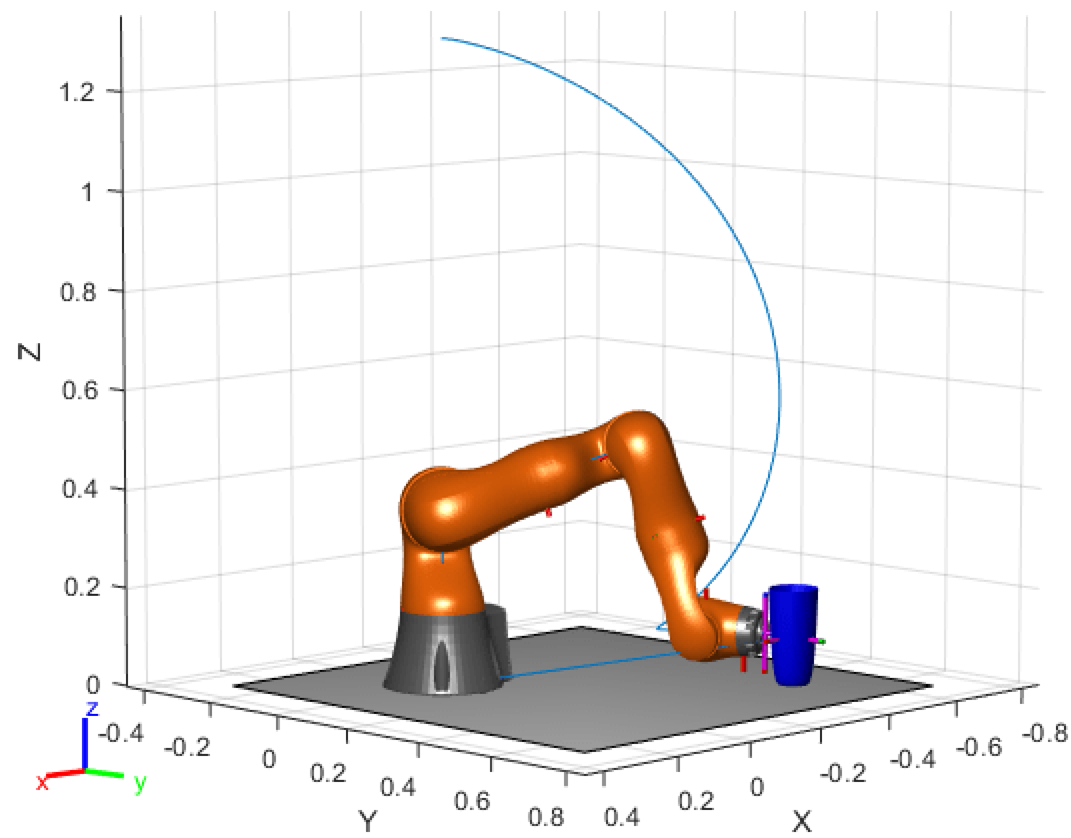
Show the robot in its initial configuration along with the table and cup

```
figure;
show(lbr, qWaypoints(1,:), 'PreservePlot', false);
hold on
exampleHelperPlotCupAndTable(cupHeight, cupRadius, cupPosition);
p = plot3(gripperPosition(1,1), gripperPosition(1,2), gripperPosition(1,3));
```



Animate the manipulator and plot the gripper position.

```
hold on
for k = 1:size(qInterp,1)
    show(lbr, qInterp(k,:), 'PreservePlot', false);
    p.XData(k) = gripperPosition(k,1);
    p.YData(k) = gripperPosition(k,2);
    p.ZData(k) = gripperPosition(k,3);
    waitfor(r);
end
hold off
```



If you want to save the generated configurations to a MAT-file for later use, execute the following:

```
>> save('lbr_trajectory.mat', 'tWaypoints', 'qWaypoints');
```

Version History

Introduced in R2017a

R2019b: constraintAiming was renamed

Behavior change in future release

The `constraintAiming` object was renamed from `robotics.AimingConstraint`. Use `constraintAiming` for all object creation.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

Objects

`generalizedInverseKinematics` | `constraintCartesianBounds` | `constraintJointBounds` | `constraintDistanceBounds` | `constraintOrientationTarget` | `constraintPoseTarget` |

`constraintPositionTarget` | `constraintFixedJoint` | `constraintPrismaticJoint` | `constraintRevoluteJoint`

Topics

"Plan a Reaching Trajectory With Multiple Kinematic Constraints"

constraintCartesianBounds

Create constraint to keep body origin inside Cartesian bounds

Description

The `constraintCartesianBounds` object describes a constraint on the position of one body (the end effector) relative to a target frame fixed on another body (the reference body). This constraint is satisfied if the position of the end-effector origin relative to the target frame remains within the `Bounds` specified. The `TargetTransform` property is the homogeneous transform that converts points in the target frame to points in the `ReferenceBody` frame.

Constraint objects are used in `generalizedInverseKinematics` objects to specify multiple kinematic constraints on a robot.

For an example that uses multiple constraint objects, see “Plan a Reaching Trajectory With Multiple Kinematic Constraints”.

Creation

Syntax

```
cartConst = constraintCartesianBounds(endeffector)
cartConst = constraintCartesianBounds(endeffector,Name=Value)
```

Description

`cartConst = constraintCartesianBounds(endeffector)` returns a Cartesian bounds object that represents a constraint on the body of the robot model specified by `endeffector` and sets the `EndEffector` property.

`cartConst = constraintCartesianBounds(endeffector,Name=Value)` returns a Cartesian bounds object with each specified property name set to the specified value by one or more name-value pair arguments.

Properties

`EndEffector` — Name of the end effector
string scalar | character vector

Name of the end effector, specified as a string scalar or character vector. When using this constraint with `generalizedInverseKinematics`, the name must match a body specified in the robot model (`rigidBodyTree`).

Example: `"left_palm"`

Data Types: `char` | `string`

`ReferenceBody` — Name of the reference body frame
`''` (default) | string scalar | character vector

Name of the reference body frame, specified as a string scalar or character vector. The default `''` indicates that the constraint is relative to the base of the robot model. When using this constraint with `generalizedInverseKinematics`, the name must match a body specified in the robot model (`rigidBodyTree`).

`TargetTransform` — Pose of the target frame relative to the reference body
`eye(4)` (default) | matrix

Pose of the target frame relative to the reference body, specified as a matrix. The matrix is a homogeneous transform that specifies the relative transformation to convert a point in the target frame to the reference body frame.

Example: `[1 0 0 1; 0 1 0 1; 0 0 1 1; 0 0 0 1]`

`Bounds` — Bounds on end-effector position relative to target frame
`zeros(3,2)` (default) | `[xMin xMax; yMin yMax; zMin zMax]` vector

Bounds on end-effector position relative to target frame, specified as a 3-by-2 vector, `[xMin xMax; yMin yMax; zMin zMax]`. Each row defines the minimum and maximum values for the *xyz*-coordinates respectively.

`Weights` — Weights of the constraint
`[1 1 1]` (default) | `[x y z]` vector

Weights of the constraint, specified as an `[x y z]` vector. Each element of the vector corresponds to the weight for the *xyz*-coordinates, respectively. These weights are used with the `Weights` property of all the constraints specified in `generalizedInverseKinematics` to properly balance each constraint.

Examples

Plan a Reaching Trajectory With Multiple Kinematic Constraints

This example shows how to use generalized inverse kinematics to plan a joint-space trajectory for a robotic manipulator. It combines multiple constraints to generate a trajectory that guides the gripper to a cup resting on a table. These constraints ensure that the gripper approaches the cup in a straight line and that the gripper remains at a safe distance from the table, without requiring the poses of the gripper to be determined in advance.

Set Up the Robot Model

This example uses a model of the KUKA LBR iiwa, a 7 degree-of-freedom robot manipulator. `importrobot` generates a `rigidBodyTree` model from a description stored in a Unified Robot Description Format (URDF) file.

```
lbr = importrobot('iiwa14.urdf'); % 14 kg payload version
lbr.DataFormat = 'row';
gripper = 'iiwa_link_ee_kuka';
```

Define dimensions for the cup.

```
cupHeight = 0.2;
cupRadius = 0.05;
cupPosition = [-0.5, 0.5, cupHeight/2];
```

Add a fixed body to the robot model representing the center of the cup.

```
body = rigidBody('cupFrame');
setFixedTransform(body.Joint, trvec2tform(cupPosition))
addBody(lbr, body, lbr.BaseName);
```

Define the Planning Problem

The goal of this example is to generate a sequence of robot configurations that satisfy the following criteria:

- Start in the home configuration
- No abrupt changes in robot configuration
- Keep the gripper at least 5 cm above the "table" (z = 0)
- The gripper should be aligned with the cup as it approaches
- Finish with the gripper 5 cm from the center of the cup

This example utilizes constraint objects to generate robot configurations that satisfy these criteria. The generated trajectory consists of five configuration waypoints. The first waypoint, `q0`, is set as the home configuration. Pre-allocate the rest of the configurations in `qWaypoints` using `repmat`.

```
numWaypoints = 5;
q0 = homeConfiguration(lbr);
qWaypoints = repmat(q0, numWaypoints, 1);
```

Create a `generalizedInverseKinematics` solver that accepts the following constraint inputs:

- Cartesian bounds - Limits the height of the gripper
- A position target - Specifies the position of the cup relative to the gripper.
- An aiming constraint - Aligns the gripper with the cup axis
- An orientation target - Maintains a fixed orientation for the gripper while approaching the cup
- Joint position bounds - Limits the change in joint positions between waypoints.

```
gik = generalizedInverseKinematics('RigidBodyTree', lbr, ...
    'ConstraintInputs', {'cartesian','position','aiming','orientation','joint'})
```

```
gik = 
  generalizedInverseKinematics with properties:

      NumConstraints: 5
    ConstraintInputs: {'cartesian'  'position'  'aiming'  'orientation'  'joint'}
       RigidBodyTree: [1x1 rigidBodyTree]
     SolverAlgorithm: 'BFGSGradientProjection'
    SolverParameters: [1x1 struct]
```

Create Constraint Objects

Create the constraint objects that are passed as inputs to the solver. These object contain the parameters needed for each constraint. Modify these parameters between calls to the solver as necessary.

Create a Cartesian bounds constraint that requires the gripper to be at least 5 cm above the table (negative z direction). All other values are given as `inf` or `-inf`.

```
heightAboveTable = constraintCartesianBounds(gripper);
heightAboveTable.Bounds = [-inf, inf; ...
                           -inf, inf; ...
                           0.05, inf]
```

```
heightAboveTable = 
  constraintCartesianBounds with properties:

        EndEffector: 'iiwa_link_ee_kuka'
      ReferenceBody: ''
    TargetTransform: [4x4 double]
             Bounds: [3x2 double]
            Weights: [1 1 1]
```

Create a constraint on the position of the cup relative to the gripper, with a tolerance of 5 mm.

```
distanceFromCup = constraintPositionTarget('cupFrame');
distanceFromCup.ReferenceBody = gripper;
distanceFromCup.PositionTolerance = 0.005
```

```
distanceFromCup = 
  constraintPositionTarget with properties:

          EndEffector: 'cupFrame'
        ReferenceBody: 'iiwa_link_ee_kuka'
       TargetPosition: [0 0 0]
    PositionTolerance: 0.0050
              Weights: 1
```

Create an aiming constraint that requires the z-axis of the `iiwa_link_ee` frame to be approximately vertical, by placing the target far above the robot. The `iiwa_link_ee` frame is oriented such that this constraint aligns the gripper with the axis of the cup.

```
alignWithCup = constraintAiming('iiwa_link_ee');
alignWithCup.TargetPoint = [0, 0, 100]
```

```
alignWithCup = 
  constraintAiming with properties:

         EndEffector: 'iiwa_link_ee'
       ReferenceBody: ''
         TargetPoint: [0 0 100]
    AngularTolerance: 0
             Weights: 1
```

Create a joint position bounds constraint. Set the `Bounds` property of this constraint based on the previous configuration to limit the change in joint positions.

```
limitJointChange = constraintJointBounds(lbr)
```

```
limitJointChange = 
  constraintJointBounds with properties:

     Bounds: [7x2 double]
    Weights: [1 1 1 1 1 1 1]
```

Create an orientation constraint for the gripper with a tolerance of one degree. This constraint requires the orientation of the gripper to match the value specified by the `TargetOrientation` property. Use this constraint to fix the orientation of the gripper during the final approach to the cup.

```
fixOrientation = constraintOrientationTarget(gripper);
fixOrientation.OrientationTolerance = deg2rad(1)
```

```
fixOrientation = 
  constraintOrientationTarget with properties:

             EndEffector: 'iiwa_link_ee_kuka'
           ReferenceBody: ''
       TargetOrientation: [1 0 0 0]
    OrientationTolerance: 0.0175
                 Weights: 1
```

Find a Configuration That Points at the Cup

This configuration should place the gripper at a distance from the cup, so that the final approach can be made with the gripper properly aligned.

```
intermediateDistance = 0.3;
```

Constraint objects have a `Weights` property which determines how the solver treats conflicting constraints. Setting the weights of a constraint to zero disables the constraint. For this configuration, disable the joint position bounds and orientation constraint.

```
limitJointChange.Weights = zeros(size(limitJointChange.Weights));
fixOrientation.Weights = 0;
```

Set the target position for the cup in the gripper frame. The cup should lie on the z-axis of the gripper at the specified distance.

```
distanceFromCup.TargetPosition = [0,0,intermediateDistance];
```

Solve for the robot configuration that satisfies the input constraints using the `gik` solver. You must specify all the input constraints. Set that configuration as the second waypoint.

```
[qWaypoints(2,:),solutionInfo] = gik(q0, heightAboveTable, ...
                       distanceFromCup, alignWithCup, fixOrientation, ...
                       limitJointChange);
```

Find Configurations That Move Gripper to the Cup Along a Straight Line

Re-enable the joint position bound and orientation constraints.

```
limitJointChange.Weights = ones(size(limitJointChange.Weights));
fixOrientation.Weights = 1;
```

Disable the align-with-cup constraint, as the orientation constraint makes it redundant.

```
alignWithCup.Weights = 0;
```

Set the orientation constraint to hold the orientation based on the previous configuration (`qWaypoints(2,:)`). Get the transformation from the gripper to the base of the robot model. Convert the homogeneous transformation to a quaternion.

```
fixOrientation.TargetOrientation = ...
    tform2quat(getTransform(lbr,qWaypoints(2,:),gripper));
```

Define the distance between the cup and gripper for each waypoint

```
finalDistanceFromCup = 0.05;
distanceFromCupValues = linspace(intermediateDistance, finalDistanceFromCup, numWaypoints-1);
```

Define the maximum allowed change in joint positions between each waypoint.

```
maxJointChange = deg2rad(10);
```

Call the solver for each remaining waypoint.

```
for k = 3:numWaypoints
    % Update the target position.
    distanceFromCup.TargetPosition(3) = distanceFromCupValues(k-1);
    % Restrict the joint positions to lie close to their previous values.
    limitJointChange.Bounds = [qWaypoints(k-1,:)' - maxJointChange, ...
                               qWaypoints(k-1,:)' + maxJointChange];
    % Solve for a configuration and add it to the waypoints array.
    [qWaypoints(k,:),solutionInfo] = gik(qWaypoints(k-1,:), ...
                                         heightAboveTable, ...
                                         distanceFromCup, alignWithCup, ...
                                         fixOrientation, limitJointChange);
end
```

Visualize the Generated Trajectory

Interpolate between the waypoints to generate a smooth trajectory. Use `pchip` to avoid overshoots, which might violate the joint limits of the robot.

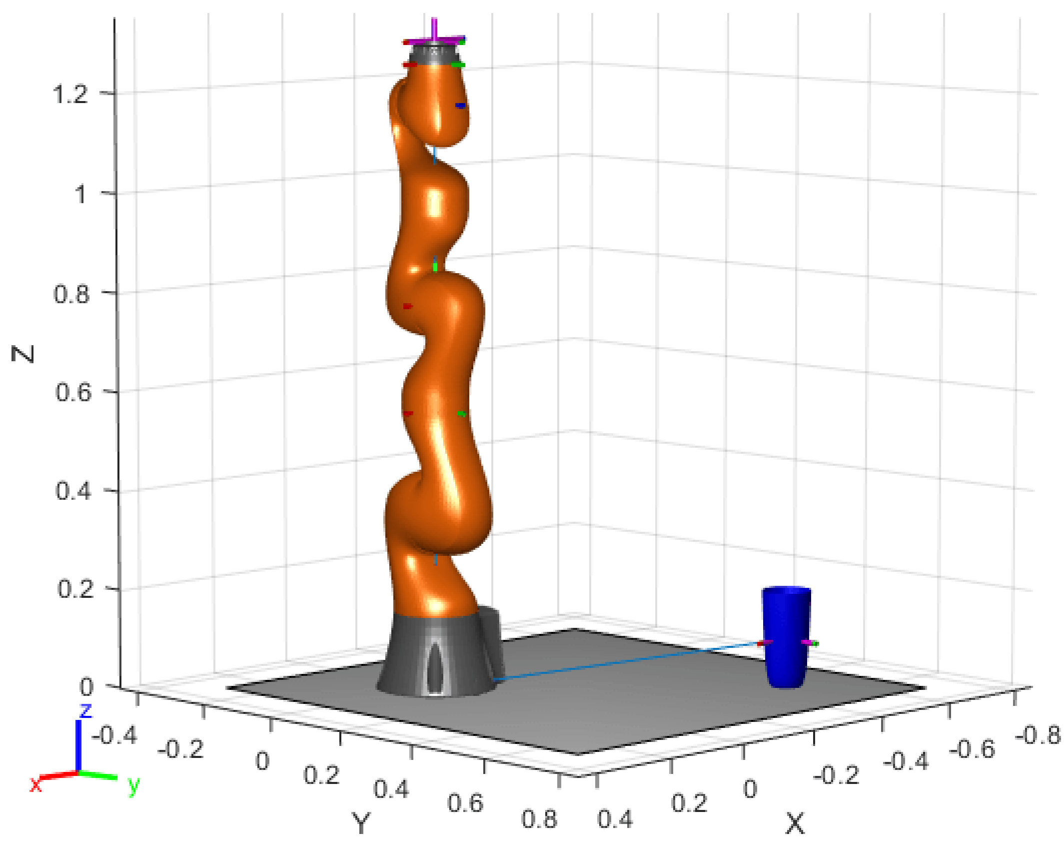
```
framerate = 15;
r = rateControl(framerate);
tFinal = 10;
tWaypoints = [0,linspace(tFinal/2,tFinal,size(qWaypoints,1)-1)];
numFrames = tFinal*framerate;
qInterp = pchip(tWaypoints,qWaypoints',linspace(0,tFinal,numFrames))';
```

Compute the gripper position for each interpolated configuration.

```
gripperPosition = zeros(numFrames,3);
for k = 1:numFrames
    gripperPosition(k,:) = tform2trvec(getTransform(lbr,qInterp(k,:), ...
                                                    gripper));
end
```

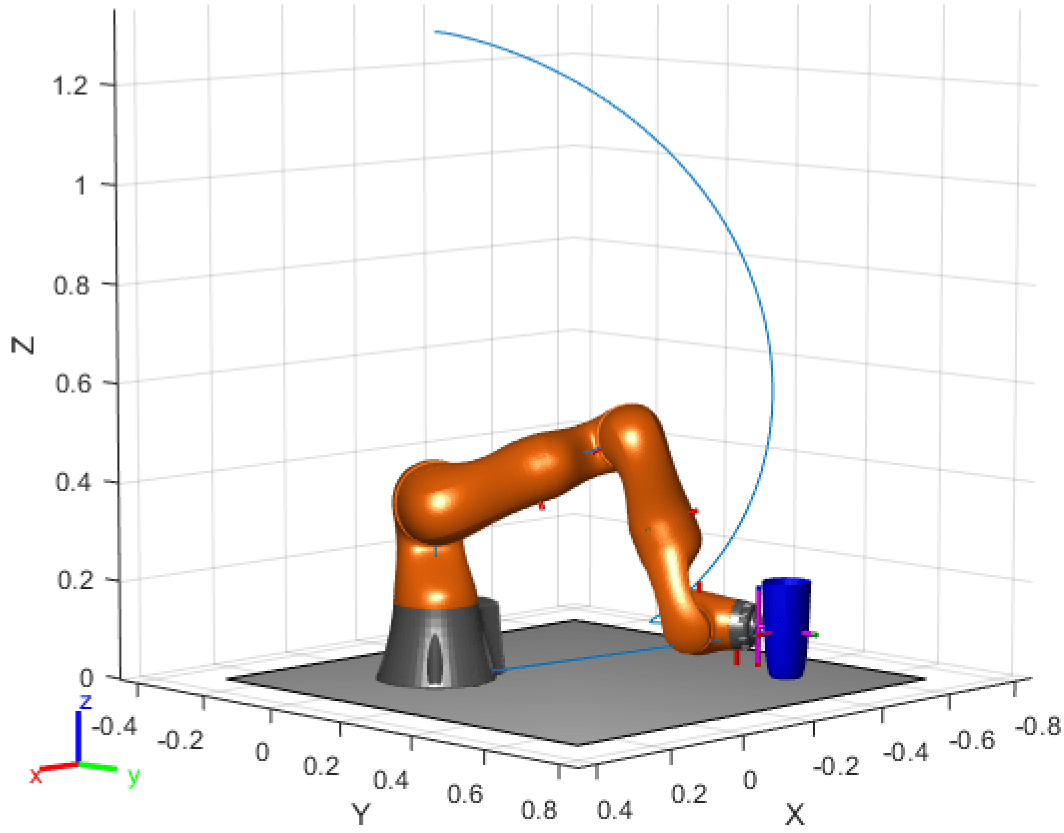
Show the robot in its initial configuration along with the table and cup

```
figure;
show(lbr, qWaypoints(1,:), 'PreservePlot', false);
hold on
exampleHelperPlotCupAndTable(cupHeight, cupRadius, cupPosition);
p = plot3(gripperPosition(1,1), gripperPosition(1,2), gripperPosition(1,3));
```



Animate the manipulator and plot the gripper position.

```
hold on
for k = 1:size(qInterp,1)
    show(lbr, qInterp(k,:), 'PreservePlot', false);
    p.XData(k) = gripperPosition(k,1);
    p.YData(k) = gripperPosition(k,2);
    p.ZData(k) = gripperPosition(k,3);
    waitfor(r);
end
hold off
```



If you want to save the generated configurations to a MAT-file for later use, execute the following:

```
>> save('lbr_trajectory.mat', 'tWaypoints', 'qWaypoints');
```

Version History

Introduced in R2017a

R2019b: `constraintCartesianBounds` was renamed

Behavior change in future release

The `constraintCartesianBounds` object was renamed from `robotics.CartesianBounds`. Use `constraintCartesianBounds` for all object creation.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

Objects

`generalizedInverseKinematics` | `constraintAiming` | `constraintDistanceBounds` | `constraintJointBounds` | `constraintOrientationTarget` | `constraintPoseTarget` |

`constraintPositionTarget` | `constraintFixedJoint` | `constraintPrismaticJoint` | `constraintRevoluteJoint`

Topics
"Plan a Reaching Trajectory With Multiple Kinematic Constraints"

constraintDistanceBounds

Constrain body within distance bounds of reference body

Description

The `constraintDistanceBounds` object describes a constraint on the distance of one body (the end effector) relative to another body (the reference body) within the same `rigidBodyTree`. This constraint is satisfied if the distance, *d*, of the end effector origin relative to the reference body origin frame is within the specified bounds.

Creation

Syntax

```
distConst = constraintDistanceBounds(endeffector)
distConst = constraintDistanceBounds(endeffector,Name=Value)
```

Description

`distConst = constraintDistanceBounds(endeffector)` returns a distance bounds constraint object, `distConst`, that represents a constraint on distance between the specified `endeffector` and the reference body specified by the `ReferenceBody` property.

`distConst = constraintDistanceBounds(endeffector,Name=Value)` specifies properties using one or more name-value arguments.

Properties

`EndEffector` — Name of end effector
string scalar | character vector

Name of the end effector, specified as a string scalar or character vector. When using this constraint with a `generalizedInverseKinematics` solver, the name must match a body specified in the associated `RigidBodyTree` robot model.

Example: `"left_palm"`

Data Types: `char` | `string`

`ReferenceBody` — Name of reference body frame
`''` (default) | string scalar | character vector

Name of the reference body frame, specified as a character vector or string scalar. The default `''` indicates that the constraint is relative to the base of the robot model. When using this constraint with a `generalizedInverseKinematics` solver, the name must match a body specified in the associated `RigidBodyTree` robot model.

Example: `"base"`

Bounds — Distance bounds
[0 0] (default) | two-element row vector

Lower and upper distance bounds imposed on the end effector from the reference body, specified as a two-element row vector of the form [*minimum maximum*].

Example: [1 3]

Weights — Weight of constraint
1 (default) | nonnegative numeric scalar

Weight of the constraint, specified as a numeric scalar. This weight is used with the `Weights` property of all the constraints specified in `generalizedInverseKinematics` solver to properly balance each constraint.

Example: 2

Examples

Create Distance Bounds Constraint

Create a `constraintDistanceBounds` object and observe its effect on an inverse kinematics solution.

Load Robot and Set Up Solver

Load a Universal UR5e robot into the workspace, and create a generalized inverse kinematics solver.

```
rng default;
robot = loadrobot("universalUR5e",DataFormat="column");
gik = generalizedInverseKinematics("RigidBodyTree",robot);
```

Set the constraint inputs `distance` for a distance bounds constraint, and `position` for the target constraint.

```
gik.ConstraintInputs = {'distance','position'};
gik.SolverParameters.MaxIterations = 100;
```

Create Distance Bounds Constraint

Create Distance Bounds constraint to constrain the origin of the end effector body, `tool0`, relative to the origin of the reference frame, `base`.

```
constrDist = constraintDistanceBounds("tool0",ReferenceBody="base");
```

Set the minimum distance between two bodies to `0.25` meters, and the maximum distance to `0.5` meters. This constraint prevents the inverse kinematics solver from solving for a configuration that violates the bounds.

```
minDist = 0.25;
maxDist = 0.5;
constrDist.Bounds = [minDist maxDist];
```

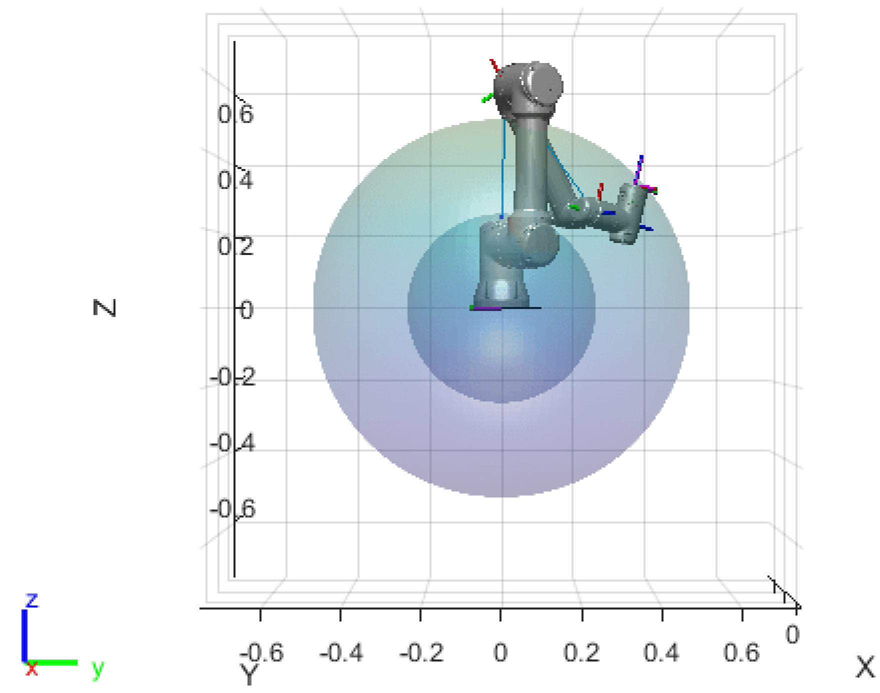
Constrain the first wrist link, `wrist_1_link`, to a target position to add some complexity.

```
forearmTgt = constraintPositionTarget('wrist_1_link');
forearmTgt.TargetPosition = [0.0 0.25 0.25];
```

Visualize Constraint

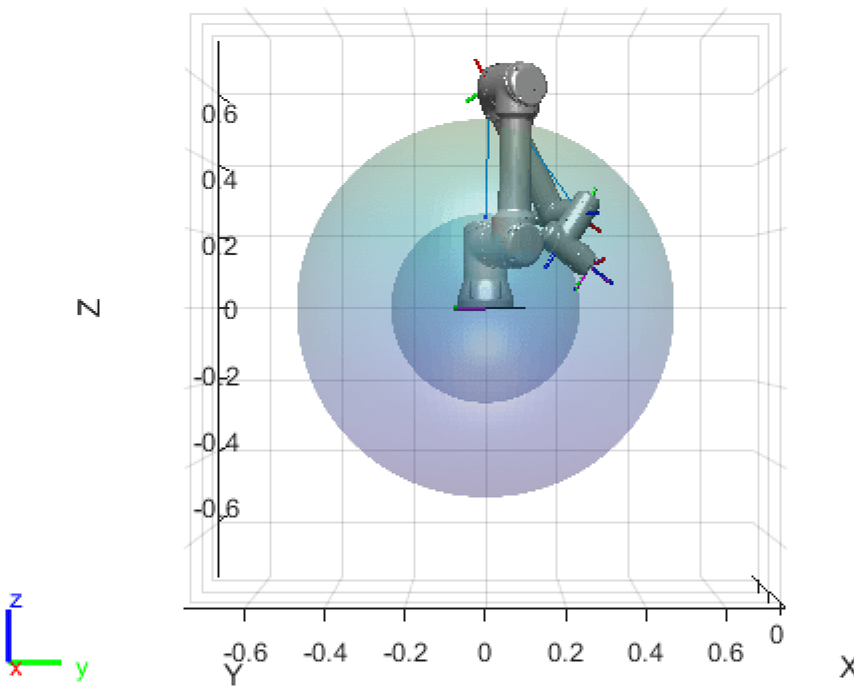
Run the solver through three random configurations, using the constraints, and then display the solver status. Each iteration, the solver finds a solution where the distance of the end effector is either equal to or within the specified bounds. Visualize the bounds by using the `exampleHelperVisualizeBounds` helper function to plot the distance bounds as two transparent spheres.

```
for i = 1:3
    figure
    q0 = randomConfiguration(robot); % Initial guess for solver
    [q,solutionInfo] = gik(q0,constrDist,forearmTgt);
    show(robot,q);
    view(90,0)
    hold on
    exampleHelperVisualizeBounds(minDist,maxDist)
    hold off
    eeDist = norm(tform2trvec(getTransform(robot,q,"tool0")));
    display(["Solver Status: ",solutionInfo.Status])
    display(["End Effector Distance: ",num2str(eeDist)])
end
```

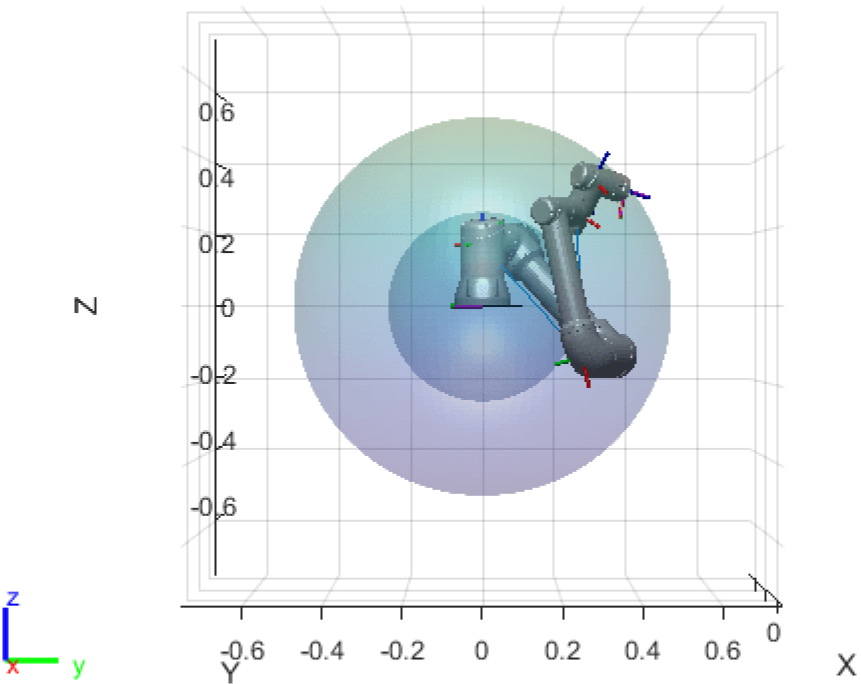



```
1x2 string array

  "Solver Status: "    "success"
```

```
1x2 string array

  "End Effector Distance: "    "0.48425"
```




```
1x2 string array

  "Solver Status: "    "success"

1x2 string array

  "End Effector Distance: "    "0.29671"
```




```
1x2 string array

  "Solver Status: "    "success"

1x2 string array

  "End Effector Distance: "    "0.48713"
```

Version History

Introduced in R2022a

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

Objects

`generalizedInverseKinematics` | `constraintAiming` | `constraintCartesianBounds` | `constraintJointBounds` | `constraintOrientationTarget` | `constraintPoseTarget` | `constraintPositionTarget` | `constraintRevoluteJoint` | `constraintPrismaticJoint` | `constraintFixedJoint`

constraintFixedJoint

Fixed joint constraint between bodies

Description

The `constraintFixedJoint` object describes a closed-loop fixed joint constraint between a successor and predecessor body on the same `rigidBodyTree`. The constraint is satisfied when there is no relative orientation, and the origins of the frames coincide. This constraint allows no relative motion between the intermediate frames when satisfied.

Creation

Syntax

```
fixedConst = constraintFixedJoint(successorbody,predecessorbody)
fixedConst = constraintFixedJoint( ___ ,Name=Value)
```

Description

`fixedConst = constraintFixedJoint(successorbody,predecessorbody)` returns a fixed constraint object, `fixedConst`, that represents a constraint between the specified successor body `successorbody` and predecessor body `predecessorbody` of the joint. The `successorbody` and `predecessor` arguments set the `SuccessorBody` and `PredecessorBody` properties, respectively.

`fixedConst = constraintFixedJoint( ___ ,Name=Value)` specifies properties using one more name-value pair arguments in addition to all input arguments from the previous syntax.

Properties

`SuccessorBody` — Name of successor body of joint
string scalar | character vector

Name of the successor body frame, specified as a string scalar or character vector. When using this constraint with the `generalizedInverseKinematics` inverse kinematics (IK) solver, the name must match a body specified in the `RigidBodyTree` of the `generalizedInverseKinematics` object.

`PredecessorBody` — Name of predecessor body of joint
string scalar | character vector

Name of the predecessor body frame, specified as a string scalar or character vector. When using this constraint with the `generalizedInverseKinematics` inverse kinematics (IK) solver, the name must match a body specified in the `RigidBodyTree` of the `generalizedInverseKinematics` object.

`SuccessorTransform` — Fixed transform of joint constraint with respect to successor body frame
`eye(4)` (default) | 4-by-4 matrix

Fixed transform of the joint constraint with respect to the successor body frame, specified as 4-by-4 matrix.

Example: `[1 0 0 1; 0 1 0 1; 0 0 1 1; 0 0 0 1]`

`PredecessorTransform` — Fixed transform of joint constraint with respect to predecessor body frame
`eye(4)` (default) | 4-by-4 matrix

Fixed transform of the joint constraint with respect to the predecessor body frame, specified as 4-by-4 matrix.

Example: `[1 0 0 1; 0 1 0 1; 0 0 1 1; 0 0 0 1]`

`PositionTolerance` — Position tolerance of joint constraint
`0` (default) | nonnegative scalar

Position tolerance of the joint constraint in meters, specified as a non-negative scalar.

`OrientationTolerance` — Orientation tolerance of joint constraint
`0` (default) | nonnegative scalar

Orientation tolerance of the joint constraint in meters, specified as a nonnegative scalar.

`Weights` — Weights of constraint
`[1 1]` (default) | two-element vector

Weights of the constraint, specified as a two-element vector. The elements of the vector corresponds to the weights for the `PositionTolerance` and `OrientationTolerance` properties, respectively. These weights are used with the weights of all the constraints specified in the `generalizedInverseKinematics` solver to properly balance each constraint.

Example: `[1 4]`

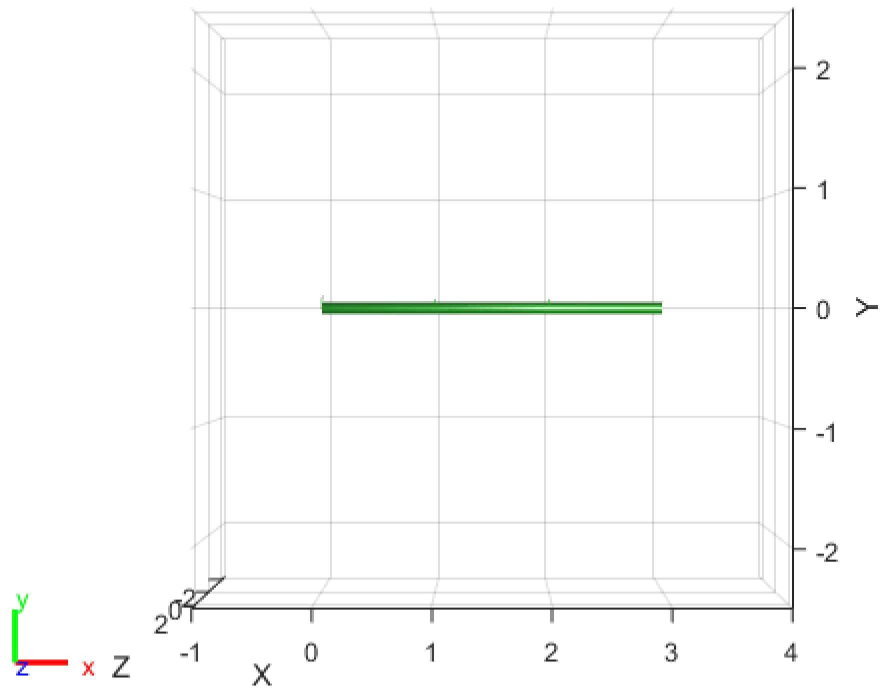
Examples

Create Loop-Closure Joint Constraints

Create a revolute, prismatic, and fixed joint constraints for a simple rigid body tree.

Use the `exampleHelperFourBarLinkageTree` helper function to create a simple robot model to demonstrate the closed-loop constraints.

```
rbt = exampleHelperFourBarLinkageTree;
show(rbt,Collisions="on");
view([0 0 pi])
xlim([-1 4])
```



Revolute Joint Constraint

To demonstrate a revolute joint constraint, create a four-bar linkage by connecting the end of the last link, `link3`, and the first link, `link0`.

Create a generalized inverse kinematics solver with a revolute joint constraint and a joint bounds constraint.

```
gikSolverWithRevoluteJointConstraint = generalizedInverseKinematics(RigidBodyTree=rbt, ...
    ConstraintInputs={'revolute','jointbounds'});
```

To ensure repeatable IK solutions, disable random restarts.

```
gikSolverWithRevoluteJointConstraint.SolverParameters.AllowRandomRestart = false;
theta = pi/2+pi/4;
```

Fix the first joint by setting `theta` as both the minimum and maximum bound.

```
activeJointConstraint = constraintJointBounds(rbt);
activeJointConstraint.Weights = [1 0 0];
activeJointConstraint.Bounds(1,:) = [theta theta];
```

Create a revolute joint constraint with successor and predecessor bodies set to the last link `link3` and the first link `link0,` respectively. Specify predecessor and successor transforms that create intermediate frames 1 meter away, in the *X*-axis, from their respective body. Once defined, these intermediate frames move such that their frame origins coincide when their *Z*-axes align.

```
cRev = constraintRevoluteJoint("link3","link0", ...
    PredecessorTransform=trvec2tform([1 0 0]), ...
    SuccessorTransform=trvec2tform([1 0 0]));
```

Provide `[theta 0 0]` as an initial guess to the solver, along with the constraints.

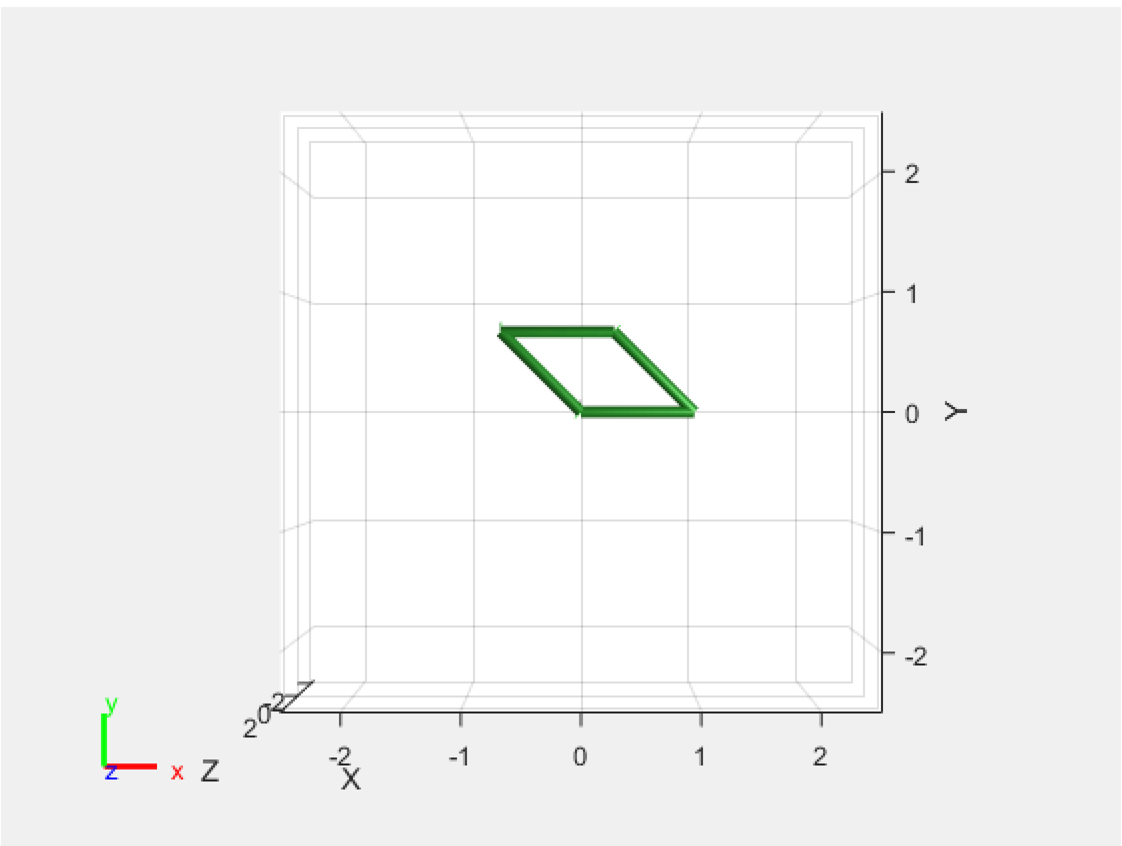
```
qConst = gikSolverWithRevoluteJointConstraint([theta 0 0],cRev,activeJointConstraint);
```

Visualize the robot to see the robot acting as a four-bar linkage. If the first joint rotates, the solver tries to keep the intermediate frames of the revolute joint constraint coincident, acting as a joint and resulting in four-bar motion.

```
figure(Name="Revolute Joint Constraint")
show(rbt,qConst,Collisions="on")
```

```
ans = 
  Axes (Primary) with properties:

             XLim: [-2.5000 2.5000]
             YLim: [-2.5000 2.5000]
           XScale: 'linear'
           YScale: 'linear'
    GridLineStyle: '-'
         Position: [0.1300 0.1100 0.7750 0.8150]
            Units: 'normalized'

  Show all properties
```

```
view([0 0 pi])
```



Prismatic Joint Constraint

Use a prismatic joint constraint to create a slider-crank. Create a new solver with a prismatic joint constraint and a joint bounds constraint.

```
gikSolverWithPrismaticJointConstraint = generalizedInverseKinematics(RigidBodyTree=rbt, ...
    ConstraintInputs={'prismatic','jointbounds'});
gikSolverWithPrismaticJointConstraint.SolverParameters.AllowRandomRestart=false;
```

Create the prismatic joint constraint with `link3` and `link0` as the successor and predecessor bodies, respectively, and set the predecessor transform such that the predecessor intermediate frame is `1` meter away on the *X*-axis and rotated `pi/2` in the *Y*-axis from the predecessor body frame.

```
cPris=constraintPrismaticJoint("link3","link0",PredecessorTransform=trvec2tform([1 0 0])*eul2tfo
```

Provide `[theta 0 0]` as an initial guess to the solver along with the constraints.

```
qConst = gikSolverWithPrismaticJointConstraint([theta 0 0],cPris,activeJointConstraint);
```

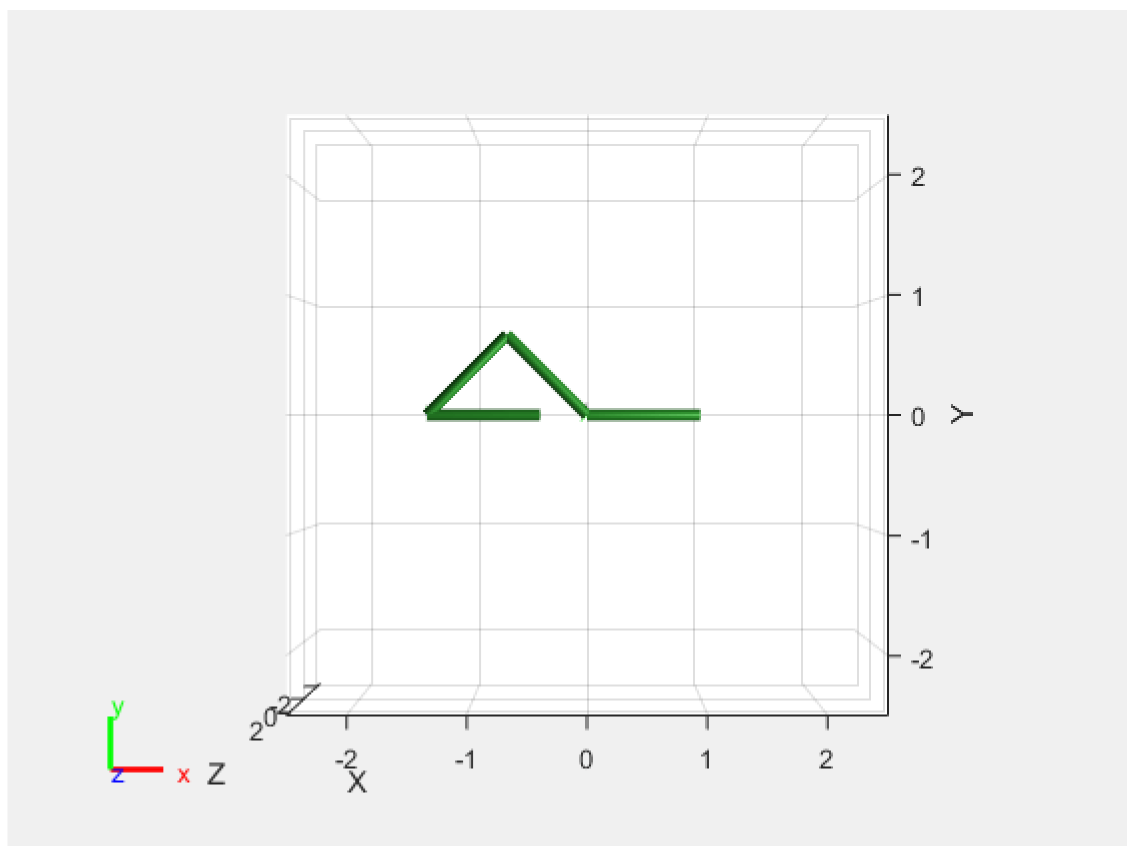
Visualize the robot to see the robot acting as a slider-crank. If the first joint rotates, the solver tries to keep the intermediate frames of the prismatic joint constraint coincident, acting as a joint and resulting in slider-crank motion.

```
figure(Name="Prismatic Joint Constraint")
show(rbt,qConst,Collisions="on")
```

```
ans = 
  Axes (Primary) with properties:
```

```
            XLim: [-2.5000 2.5000]
            YLim: [-2.5000 2.5000]
          XScale: 'linear'
          YScale: 'linear'
   GridLineStyle: '-'
        Position: [0.1300 0.1100 0.7750 0.8150]
           Units: 'normalized'

  Show all properties
```

```
view([0 0 pi])
```



Fixed Joint Constraint

To demonstrate a fixed joint constraint, create a triangle with the links that is preserved when the first joint moves. Create a new solver with a fixed joint constraint.

```
gikSolverWithFixedJointConstraint = generalizedInverseKinematics(RigidBodyTree=rbt, ...
    ConstraintInputs={'fixed'});
```

Create the fixed joint constraint with `link3` and `link0` as the successor and predecessor bodies, respectively, and set the successor transform such that the predecessor intermediate frame is `1` meter away on the *X*-axis from the predecessor body frame.

```
cFix = constraintFixedJoint("link3","link1",SuccessorTransform=trvec2tform([1 0 0]));
```

Set the weight of the orientation constraint of the fixed joint constraint to 0.

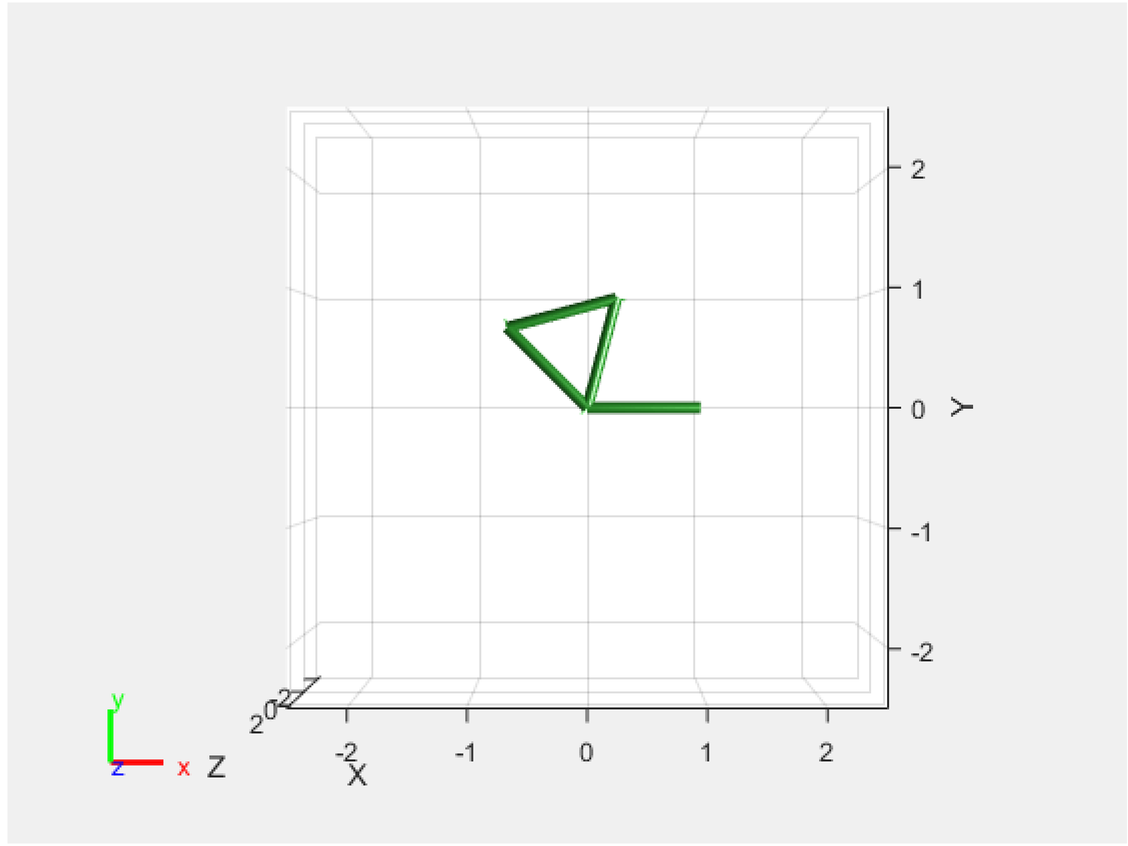
```
cFix.Weights = [1 0];
[qConst,solInfo] = gikSolverWithFixedJointConstraint([theta 0.1 0],cFix);
```

Visualize the robot to see how the fixed constraint joint acts on the robot frame. If the first joint rotates, the solver tries to keep the intermediate frames of the fixed joint constraint coincident, acting as a fixed joint.

```
figure(Name="Fixed Joint Constraint")
show(rbt,qConst,Collisions="on")
```

```
ans = 
  Axes (Primary) with properties:

             XLim: [-2.5000 2.5000]
             YLim: [-2.5000 2.5000]
           XScale: 'linear'
           YScale: 'linear'
    GridLineStyle: '-'
         Position: [0.1300 0.1100 0.7750 0.8150]
            Units: 'normalized'

  Show all properties
```

```
view([0 0 pi])
```



Version History

Introduced in R2022a

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

Objects

`generalizedInverseKinematics` | `constraintAiming` | `constraintCartesianBounds` | `constraintJointBounds` | `constraintOrientationTarget` | `constraintPoseTarget` | `constraintPositionTarget` | `constraintRevoluteJoint` | `constraintPrismaticJoint` | `constraintDistanceBounds`

Topics

"Solve Inverse Kinematics for Closed Loop Linkages"

constraintJointBounds

Create constraint on joint positions of robot model

Description

The `constraintJointBounds` object describes a constraint on the joint positions of a rigid body tree. This constraint is satisfied if the robot configuration vector maintains all joint positions within the `Bounds` specified. The configuration vector contains positions for all nonfixed joints in a `rigidBodyTree` object.

Constraint objects are used in `generalizedInverseKinematics` objects to specify multiple kinematic constraints on a robot.

For an example that uses multiple constraint objects, see “Plan a Reaching Trajectory With Multiple Kinematic Constraints”.

Creation

Syntax

```
jointConst = constraintJointBounds(robot)
jointConst = constraintJointBounds(robot,Name=Value)
```

Description

`jointConst = constraintJointBounds(robot)` returns a joint position bounds object that represents a constraint on the configuration vector of the robot model specified by `robot`.

`jointConst = constraintJointBounds(robot,Name=Value)` returns a joint position bounds object with each specified property name set to the specified value by one or more name-value pair arguments.

Input Arguments

`robot` — Rigid body tree model
`rigidBodyTree` object

Rigid body tree model, specified as a `rigidBodyTree` object.

Properties

`Bounds` — Bounds on the configuration vector
n-by-2 matrix

Bounds on the configuration vector, specified as an *n*-by-2 matrix. Each row of the array corresponds to a nonfixed joint on the robot model and gives the minimum and maximum position for that joint. By default, the bounds are set based on the `PositionLimits` property of each `rigidBodyJoint` object within the input rigid body tree model, `robot`.

`Weights` — Weights of the constraint

`ones(1,n)` (default) | *n*-element vector

Weights of the constraint, specified as an *n*-element vector, where each element corresponds to a row in `Bounds` and gives relative weights for each bound. The default is a vector of ones to give equal weight to all joint positions. These weights are used with the `Weights` property of all the constraints specified in `generalizedInverseKinematics` to properly balance each constraint

Examples

Plan a Reaching Trajectory With Multiple Kinematic Constraints

This example shows how to use generalized inverse kinematics to plan a joint-space trajectory for a robotic manipulator. It combines multiple constraints to generate a trajectory that guides the gripper to a cup resting on a table. These constraints ensure that the gripper approaches the cup in a straight line and that the gripper remains at a safe distance from the table, without requiring the poses of the gripper to be determined in advance.

Set Up the Robot Model

This example uses a model of the KUKA LBR iiwa, a 7 degree-of-freedom robot manipulator. `importrobot` generates a `rigidBodyTree` model from a description stored in a Unified Robot Description Format (URDF) file.

```
lbr = importrobot('iiwa14.urdf'); % 14 kg payload version
lbr.DataFormat = 'row';
gripper = 'iiwa_link_ee_kuka';
```

Define dimensions for the cup.

```
cupHeight = 0.2;
cupRadius = 0.05;
cupPosition = [-0.5, 0.5, cupHeight/2];
```

Add a fixed body to the robot model representing the center of the cup.

```
body = rigidBody('cupFrame');
setFixedTransform(body.Joint, trvec2tform(cupPosition))
addBody(lbr, body, lbr.BaseName);
```

Define the Planning Problem

The goal of this example is to generate a sequence of robot configurations that satisfy the following criteria:

- Start in the home configuration
- No abrupt changes in robot configuration
- Keep the gripper at least 5 cm above the "table" (z = 0)
- The gripper should be aligned with the cup as it approaches
- Finish with the gripper 5 cm from the center of the cup

This example utilizes constraint objects to generate robot configurations that satisfy these criteria. The generated trajectory consists of five configuration waypoints. The first waypoint, `q0`, is set as the home configuration. Pre-allocate the rest of the configurations in `qWaypoints` using `repmat`.

```
numWaypoints = 5;
q0 = homeConfiguration(lbr);
qWaypoints = repmat(q0, numWaypoints, 1);
```

Create a `generalizedInverseKinematics` solver that accepts the following constraint inputs:

- Cartesian bounds - Limits the height of the gripper
- A position target - Specifies the position of the cup relative to the gripper.
- An aiming constraint - Aligns the gripper with the cup axis
- An orientation target - Maintains a fixed orientation for the gripper while approaching the cup
- Joint position bounds - Limits the change in joint positions between waypoints.

```
gik = generalizedInverseKinematics('RigidBodyTree', lbr, ...
    'ConstraintInputs', {'cartesian','position','aiming','orientation','joint'})
```

```
gik = 
  generalizedInverseKinematics with properties:

      NumConstraints: 5
    ConstraintInputs: {'cartesian'  'position'  'aiming'  'orientation'  'joint'}
       RigidBodyTree: [1x1 rigidBodyTree]
     SolverAlgorithm: 'BFGSGradientProjection'
    SolverParameters: [1x1 struct]
```

Create Constraint Objects

Create the constraint objects that are passed as inputs to the solver. These object contain the parameters needed for each constraint. Modify these parameters between calls to the solver as necessary.

Create a Cartesian bounds constraint that requires the gripper to be at least 5 cm above the table (negative z direction). All other values are given as `inf` or `-inf`.

```
heightAboveTable = constraintCartesianBounds(gripper);
heightAboveTable.Bounds = [-inf, inf; ...
                           -inf, inf; ...
                           0.05, inf]
```

```
heightAboveTable = 
  constraintCartesianBounds with properties:

        EndEffector: 'iiwa_link_ee_kuka'
      ReferenceBody: ''
    TargetTransform: [4x4 double]
             Bounds: [3x2 double]
            Weights: [1 1 1]
```

Create a constraint on the position of the cup relative to the gripper, with a tolerance of 5 mm.

```
distanceFromCup = constraintPositionTarget('cupFrame');
distanceFromCup.ReferenceBody = gripper;
distanceFromCup.PositionTolerance = 0.005
```

```
distanceFromCup = 
  constraintPositionTarget with properties:
```

```
        EndEffector: 'cupFrame'
      ReferenceBody: 'iiwa_link_ee_kuka'
     TargetPosition: [0 0 0]
  PositionTolerance: 0.0050
            Weights: 1
```

Create an aiming constraint that requires the z-axis of the `iiwa_link_ee` frame to be approximately vertical, by placing the target far above the robot. The `iiwa_link_ee` frame is oriented such that this constraint aligns the gripper with the axis of the cup.

```
alignWithCup = constraintAiming('iiwa_link_ee');
alignWithCup.TargetPoint = [0, 0, 100]
```

```
alignWithCup = 
  constraintAiming with properties:

         EndEffector: 'iiwa_link_ee'
       ReferenceBody: ''
         TargetPoint: [0 0 100]
    AngularTolerance: 0
             Weights: 1
```

Create a joint position bounds constraint. Set the `Bounds` property of this constraint based on the previous configuration to limit the change in joint positions.

```
limitJointChange = constraintJointBounds(lbr)
```

```
limitJointChange = 
  constraintJointBounds with properties:

     Bounds: [7x2 double]
    Weights: [1 1 1 1 1 1 1]
```

Create an orientation constraint for the gripper with a tolerance of one degree. This constraint requires the orientation of the gripper to match the value specified by the `TargetOrientation` property. Use this constraint to fix the orientation of the gripper during the final approach to the cup.

```
fixOrientation = constraintOrientationTarget(gripper);
fixOrientation.OrientationTolerance = deg2rad(1)
```

```
fixOrientation = 
  constraintOrientationTarget with properties:

             EndEffector: 'iiwa_link_ee_kuka'
           ReferenceBody: ''
       TargetOrientation: [1 0 0 0]
    OrientationTolerance: 0.0175
                 Weights: 1
```

Find a Configuration That Points at the Cup

This configuration should place the gripper at a distance from the cup, so that the final approach can be made with the gripper properly aligned.

```
intermediateDistance = 0.3;
```

Constraint objects have a `Weights` property which determines how the solver treats conflicting constraints. Setting the weights of a constraint to zero disables the constraint. For this configuration, disable the joint position bounds and orientation constraint.

```
limitJointChange.Weights = zeros(size(limitJointChange.Weights));
fixOrientation.Weights = 0;
```

Set the target position for the cup in the gripper frame. The cup should lie on the z-axis of the gripper at the specified distance.

```
distanceFromCup.TargetPosition = [0,0,intermediateDistance];
```

Solve for the robot configuration that satisfies the input constraints using the `gik` solver. You must specify all the input constraints. Set that configuration as the second waypoint.

```
[qWaypoints(2,:),solutionInfo] = gik(q0, heightAboveTable, ...
                       distanceFromCup, alignWithCup, fixOrientation, ...
                       limitJointChange);
```

Find Configurations That Move Gripper to the Cup Along a Straight Line

Re-enable the joint position bound and orientation constraints.

```
limitJointChange.Weights = ones(size(limitJointChange.Weights));
fixOrientation.Weights = 1;
```

Disable the align-with-cup constraint, as the orientation constraint makes it redundant.

```
alignWithCup.Weights = 0;
```

Set the orientation constraint to hold the orientation based on the previous configuration (`qWaypoints(2,:)`). Get the transformation from the gripper to the base of the robot model. Convert the homogeneous transformation to a quaternion.

```
fixOrientation.TargetOrientation = ...
    tform2quat(getTransform(lbr,qWaypoints(2,:),gripper));
```

Define the distance between the cup and gripper for each waypoint

```
finalDistanceFromCup = 0.05;
distanceFromCupValues = linspace(intermediateDistance, finalDistanceFromCup, numWaypoints-1);
```

Define the maximum allowed change in joint positions between each waypoint.

```
maxJointChange = deg2rad(10);
```

Call the solver for each remaining waypoint.

```
for k = 3:numWaypoints
    % Update the target position.
    distanceFromCup.TargetPosition(3) = distanceFromCupValues(k-1);
    % Restrict the joint positions to lie close to their previous values.
    limitJointChange.Bounds = [qWaypoints(k-1,:)' - maxJointChange, ...
                               qWaypoints(k-1,:)' + maxJointChange];
    % Solve for a configuration and add it to the waypoints array.
    [qWaypoints(k,:),solutionInfo] = gik(qWaypoints(k-1,:), ...
                                         heightAboveTable, ...
```

```
                                                distanceFromCup, alignWithCup, ...
                                                fixOrientation, limitJointChange);
end
```

Visualize the Generated Trajectory

Interpolate between the waypoints to generate a smooth trajectory. Use `pchip` to avoid overshoots, which might violate the joint limits of the robot.

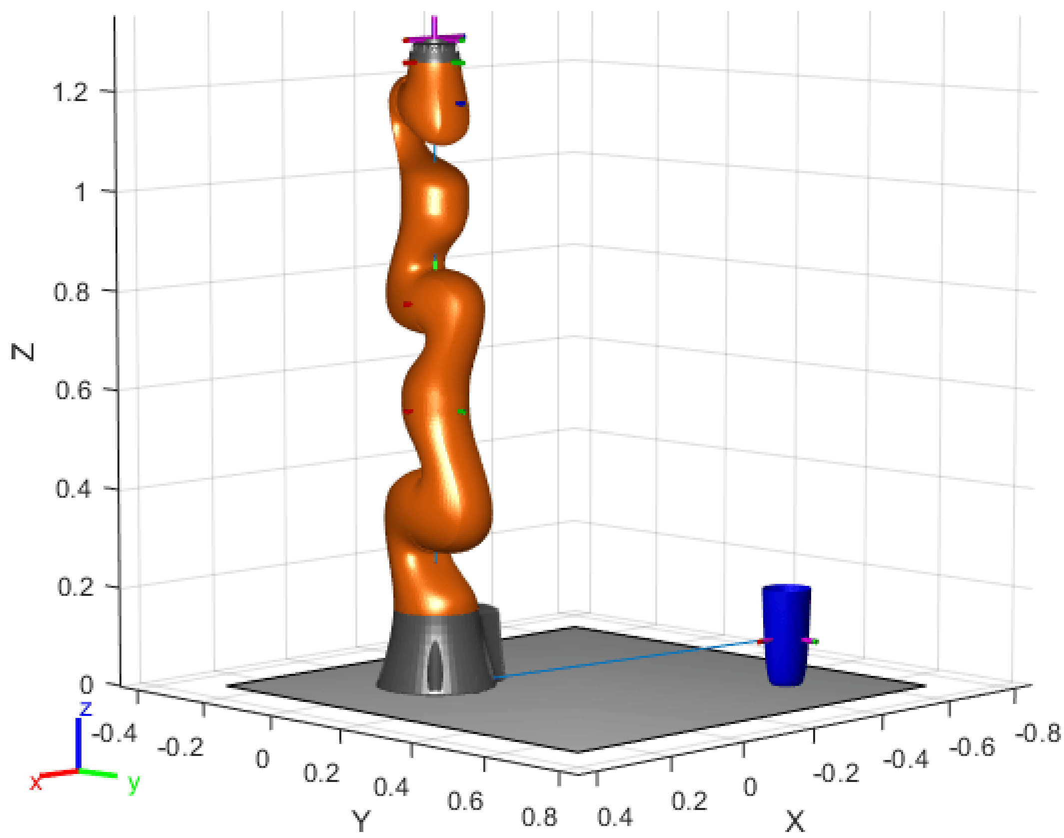
```
framerate = 15;
r = rateControl(framerate);
tFinal = 10;
tWaypoints = [0,linspace(tFinal/2,tFinal,size(qWaypoints,1)-1)];
numFrames = tFinal*framerate;
qInterp = pchip(tWaypoints,qWaypoints',linspace(0,tFinal,numFrames))';
```

Compute the gripper position for each interpolated configuration.

```
gripperPosition = zeros(numFrames,3);
for k = 1:numFrames
    gripperPosition(k,:) = tform2trvec(getTransform(lbr,qInterp(k,:), ...
                                                    gripper));
end
```

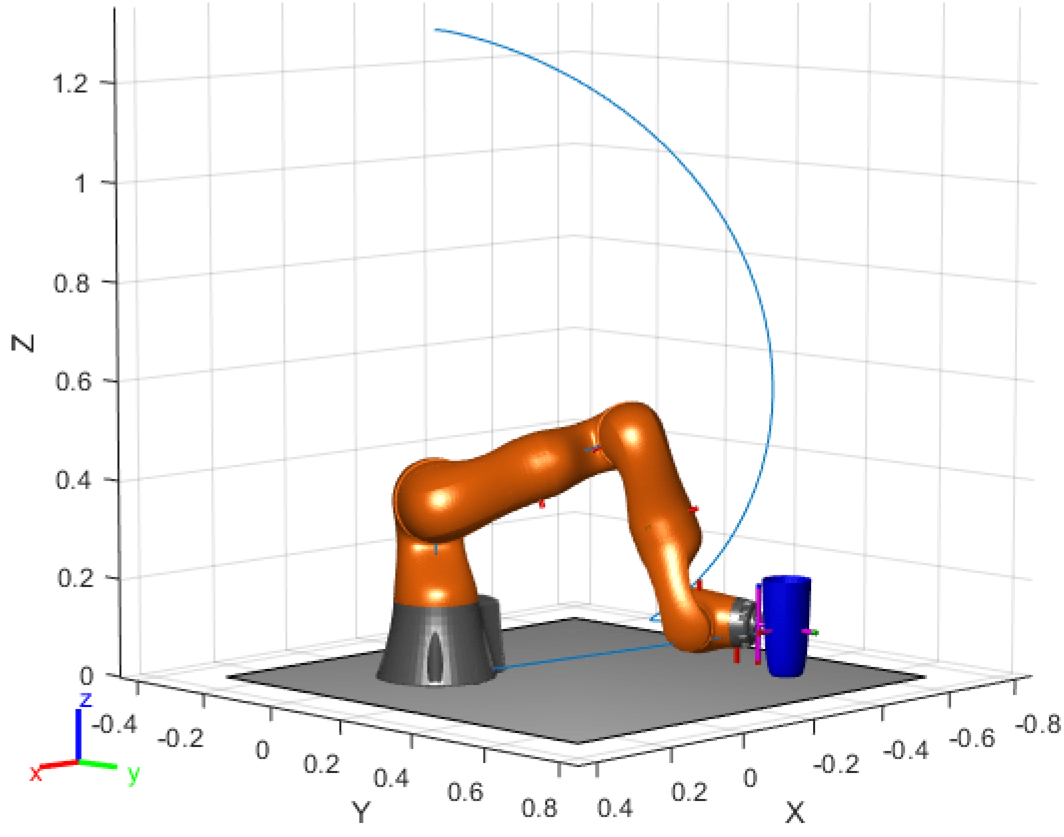
Show the robot in its initial configuration along with the table and cup

```
figure;
show(lbr, qWaypoints(1,:), 'PreservePlot', false);
hold on
exampleHelperPlotCupAndTable(cupHeight, cupRadius, cupPosition);
p = plot3(gripperPosition(1,1), gripperPosition(1,2), gripperPosition(1,3));
```



Animate the manipulator and plot the gripper position.

```
hold on
for k = 1:size(qInterp,1)
    show(lbr, qInterp(k,:), 'PreservePlot', false);
    p.XData(k) = gripperPosition(k,1);
    p.YData(k) = gripperPosition(k,2);
    p.ZData(k) = gripperPosition(k,3);
    waitfor(r);
end
hold off
```



If you want to save the generated configurations to a MAT-file for later use, execute the following:

```
>> save('lbr_trajectory.mat', 'tWaypoints', 'qWaypoints');
```

Version History

Introduced in R2017a

R2019b: constraintJointBounds was renamed

Behavior change in future release

The `constraintJointBounds` object was renamed from `robotics.JointPositionBounds`. Use `constraintJointBounds` for all object creation.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

Objects

`generalizedInverseKinematics` | `constraintAiming` | `constraintCartesianBounds` | `constraintDistanceBounds` | `constraintOrientationTarget` | `constraintPoseTarget` |

`constraintPositionTarget` | `constraintFixedJoint` | `constraintPrismaticJoint` | `constraintRevoluteJoint`

Topics
"Plan a Reaching Trajectory With Multiple Kinematic Constraints"

constraintOrientationTarget

Create constraint on relative orientation of body

Description

The `constraintOrientationTarget` object describes a constraint that requires the orientation of one body (the end effector) to match a target orientation within an angular tolerance in any direction. The target orientation is specified relative to the body frame of the reference body.

Constraint objects are used in `generalizedInverseKinematics` objects to specify multiple kinematic constraints on a robot.

For an example that uses multiple constraint objects, see "Plan a Reaching Trajectory With Multiple Kinematic Constraints".

Creation

Syntax

```
orientationConst = constraintOrientationTarget(endeffector)
orientationConst = constraintOrientationTarget(endeffector,Name=Value)
```

Description

`orientationConst = constraintOrientationTarget(endeffector)` returns an orientation target object that represents a constraint on a body of the robot model specified by `endeffector` and sets the `EndEffector` property.

`orientationConst = constraintOrientationTarget(endeffector,Name=Value)` returns an orientation target object with each specified property name set to the specified value by one or more name-value pair arguments.

Properties

`EndEffector` — Name of the end effector
string scalar | character vector

Name of the end effector, specified as a string scalar or character vector. When using this constraint with `generalizedInverseKinematics`, the name must match a body specified in the robot model (`rigidBodyTree`).

Example: `"left_palm"`

Data Types: `char` | `string`

`ReferenceBody` — Name of the reference body frame
`''` (default) | string scalar | character vector

Name of the reference body frame, specified as a string scalar or character vector. The default `''` indicates that the constraint is relative to the base of the robot model. When using this constraint with `generalizedInverseKinematics`, the name must match a body specified in the robot model (`rigidBodyTree`).

Data Types: `char` | `string`

`TargetOrientation` — Target orientation of the end effector relative to the reference body
`[1 0 0 0]` (default) | four-element vector

Target orientation of the end effector relative to the reference body, specified as four-element vector that represents a unit quaternion. The orientation of the end effector relative to the reference body frame is the orientation that converts a direction specified in the end-effector frame to the same direction specified in the reference body frame.

`OrientationTolerance` — Maximum allowed rotation angle
`0` (default) | numeric scalar

Maximum allowed rotation angle in radians, specified as a numeric scalar. This value is the upper bound on the magnitude of the rotation required to make the end-effector orientation match the target orientation.

`Weights` — Weight of the constraint
`1` (default) | numeric scalar

Weight of the constraint, specified as a numeric scalar. This weight is used with the `Weights` property of all the constraints specified in `generalizedInverseKinematics` to properly balance each constraint.

Examples

Plan a Reaching Trajectory With Multiple Kinematic Constraints

This example shows how to use generalized inverse kinematics to plan a joint-space trajectory for a robotic manipulator. It combines multiple constraints to generate a trajectory that guides the gripper to a cup resting on a table. These constraints ensure that the gripper approaches the cup in a straight line and that the gripper remains at a safe distance from the table, without requiring the poses of the gripper to be determined in advance.

Set Up the Robot Model

This example uses a model of the KUKA LBR iiwa, a 7 degree-of-freedom robot manipulator. `importrobot` generates a `rigidBodyTree` model from a description stored in a Unified Robot Description Format (URDF) file.

```
lbr = importrobot('iiwa14.urdf'); % 14 kg payload version
lbr.DataFormat = 'row';
gripper = 'iiwa_link_ee_kuka';
```

Define dimensions for the cup.

```
cupHeight = 0.2;
cupRadius = 0.05;
cupPosition = [-0.5, 0.5, cupHeight/2];
```

Add a fixed body to the robot model representing the center of the cup.

```
body = rigidBody('cupFrame');
setFixedTransform(body.Joint, trvec2tform(cupPosition))
addBody(lbr, body, lbr.BaseName);
```

Define the Planning Problem

The goal of this example is to generate a sequence of robot configurations that satisfy the following criteria:

- Start in the home configuration
- No abrupt changes in robot configuration
- Keep the gripper at least 5 cm above the "table" (z = 0)
- The gripper should be aligned with the cup as it approaches
- Finish with the gripper 5 cm from the center of the cup

This example utilizes constraint objects to generate robot configurations that satisfy these criteria. The generated trajectory consists of five configuration waypoints. The first waypoint, `q0`, is set as the home configuration. Pre-allocate the rest of the configurations in `qWaypoints` using `repmat`.

```
numWaypoints = 5;
q0 = homeConfiguration(lbr);
qWaypoints = repmat(q0, numWaypoints, 1);
```

Create a `generalizedInverseKinematics` solver that accepts the following constraint inputs:

- Cartesian bounds - Limits the height of the gripper
- A position target - Specifies the position of the cup relative to the gripper.
- An aiming constraint - Aligns the gripper with the cup axis
- An orientation target - Maintains a fixed orientation for the gripper while approaching the cup
- Joint position bounds - Limits the change in joint positions between waypoints.

```
gik = generalizedInverseKinematics('RigidBodyTree', lbr, ...
    'ConstraintInputs', {'cartesian','position','aiming','orientation','joint'})
```

```
gik = 
  generalizedInverseKinematics with properties:

      NumConstraints: 5
    ConstraintInputs: {'cartesian'  'position'  'aiming'  'orientation'  'joint'}
       RigidBodyTree: [1x1 rigidBodyTree]
     SolverAlgorithm: 'BFGSGradientProjection'
    SolverParameters: [1x1 struct]
```

Create Constraint Objects

Create the constraint objects that are passed as inputs to the solver. These object contain the parameters needed for each constraint. Modify these parameters between calls to the solver as necessary.

Create a Cartesian bounds constraint that requires the gripper to be at least 5 cm above the table (negative z direction). All other values are given as `inf` or `-inf`.

```
heightAboveTable = constraintCartesianBounds(gripper);
heightAboveTable.Bounds = [-inf, inf; ...
                           -inf, inf; ...
                           0.05, inf]
```

```
heightAboveTable = 
  constraintCartesianBounds with properties:

        EndEffector: 'iiwa_link_ee_kuka'
      ReferenceBody: ''
    TargetTransform: [4x4 double]
             Bounds: [3x2 double]
            Weights: [1 1 1]
```

Create a constraint on the position of the cup relative to the gripper, with a tolerance of 5 mm.

```
distanceFromCup = constraintPositionTarget('cupFrame');
distanceFromCup.ReferenceBody = gripper;
distanceFromCup.PositionTolerance = 0.005
```

```
distanceFromCup = 
  constraintPositionTarget with properties:

          EndEffector: 'cupFrame'
        ReferenceBody: 'iiwa_link_ee_kuka'
       TargetPosition: [0 0 0]
    PositionTolerance: 0.0050
              Weights: 1
```

Create an aiming constraint that requires the z-axis of the `iiwa_link_ee` frame to be approximately vertical, by placing the target far above the robot. The `iiwa_link_ee` frame is oriented such that this constraint aligns the gripper with the axis of the cup.

```
alignWithCup = constraintAiming('iiwa_link_ee');
alignWithCup.TargetPoint = [0, 0, 100]
```

```
alignWithCup = 
  constraintAiming with properties:

         EndEffector: 'iiwa_link_ee'
       ReferenceBody: ''
         TargetPoint: [0 0 100]
    AngularTolerance: 0
             Weights: 1
```

Create a joint position bounds constraint. Set the `Bounds` property of this constraint based on the previous configuration to limit the change in joint positions.

```
limitJointChange = constraintJointBounds(lbr)
```

```
limitJointChange = 
  constraintJointBounds with properties:

     Bounds: [7x2 double]
    Weights: [1 1 1 1 1 1 1]
```

Create an orientation constraint for the gripper with a tolerance of one degree. This constraint requires the orientation of the gripper to match the value specified by the `TargetOrientation` property. Use this constraint to fix the orientation of the gripper during the final approach to the cup.

```
fixOrientation = constraintOrientationTarget(gripper);
fixOrientation.OrientationTolerance = deg2rad(1)
```

```
fixOrientation = 
  constraintOrientationTarget with properties:

             EndEffector: 'iiwa_link_ee_kuka'
           ReferenceBody: ''
       TargetOrientation: [1 0 0 0]
    OrientationTolerance: 0.0175
                 Weights: 1
```

Find a Configuration That Points at the Cup

This configuration should place the gripper at a distance from the cup, so that the final approach can be made with the gripper properly aligned.

```
intermediateDistance = 0.3;
```

Constraint objects have a `Weights` property which determines how the solver treats conflicting constraints. Setting the weights of a constraint to zero disables the constraint. For this configuration, disable the joint position bounds and orientation constraint.

```
limitJointChange.Weights = zeros(size(limitJointChange.Weights));
fixOrientation.Weights = 0;
```

Set the target position for the cup in the gripper frame. The cup should lie on the z-axis of the gripper at the specified distance.

```
distanceFromCup.TargetPosition = [0,0,intermediateDistance];
```

Solve for the robot configuration that satisfies the input constraints using the `gik` solver. You must specify all the input constraints. Set that configuration as the second waypoint.

```
[qWaypoints(2,:),solutionInfo] = gik(q0, heightAboveTable, ...
                       distanceFromCup, alignWithCup, fixOrientation, ...
                       limitJointChange);
```

Find Configurations That Move Gripper to the Cup Along a Straight Line

Re-enable the joint position bound and orientation constraints.

```
limitJointChange.Weights = ones(size(limitJointChange.Weights));
fixOrientation.Weights = 1;
```

Disable the align-with-cup constraint, as the orientation constraint makes it redundant.

```
alignWithCup.Weights = 0;
```

Set the orientation constraint to hold the orientation based on the previous configuration (`qWaypoints(2,:)`). Get the transformation from the gripper to the base of the robot model. Convert the homogeneous transformation to a quaternion.

```
fixOrientation.TargetOrientation = ...
    tform2quat(getTransform(lbr,qWaypoints(2,:),gripper));
```

Define the distance between the cup and gripper for each waypoint

```
finalDistanceFromCup = 0.05;
distanceFromCupValues = linspace(intermediateDistance, finalDistanceFromCup, numWaypoints-1);
```

Define the maximum allowed change in joint positions between each waypoint.

```
maxJointChange = deg2rad(10);
```

Call the solver for each remaining waypoint.

```
for k = 3:numWaypoints
    % Update the target position.
    distanceFromCup.TargetPosition(3) = distanceFromCupValues(k-1);
    % Restrict the joint positions to lie close to their previous values.
    limitJointChange.Bounds = [qWaypoints(k-1,:)' - maxJointChange, ...
                               qWaypoints(k-1,:)' + maxJointChange];
    % Solve for a configuration and add it to the waypoints array.
    [qWaypoints(k,:),solutionInfo] = gik(qWaypoints(k-1,:), ...
                                         heightAboveTable, ...
                                         distanceFromCup, alignWithCup, ...
                                         fixOrientation, limitJointChange);
end
```

Visualize the Generated Trajectory

Interpolate between the waypoints to generate a smooth trajectory. Use `pchip` to avoid overshoots, which might violate the joint limits of the robot.

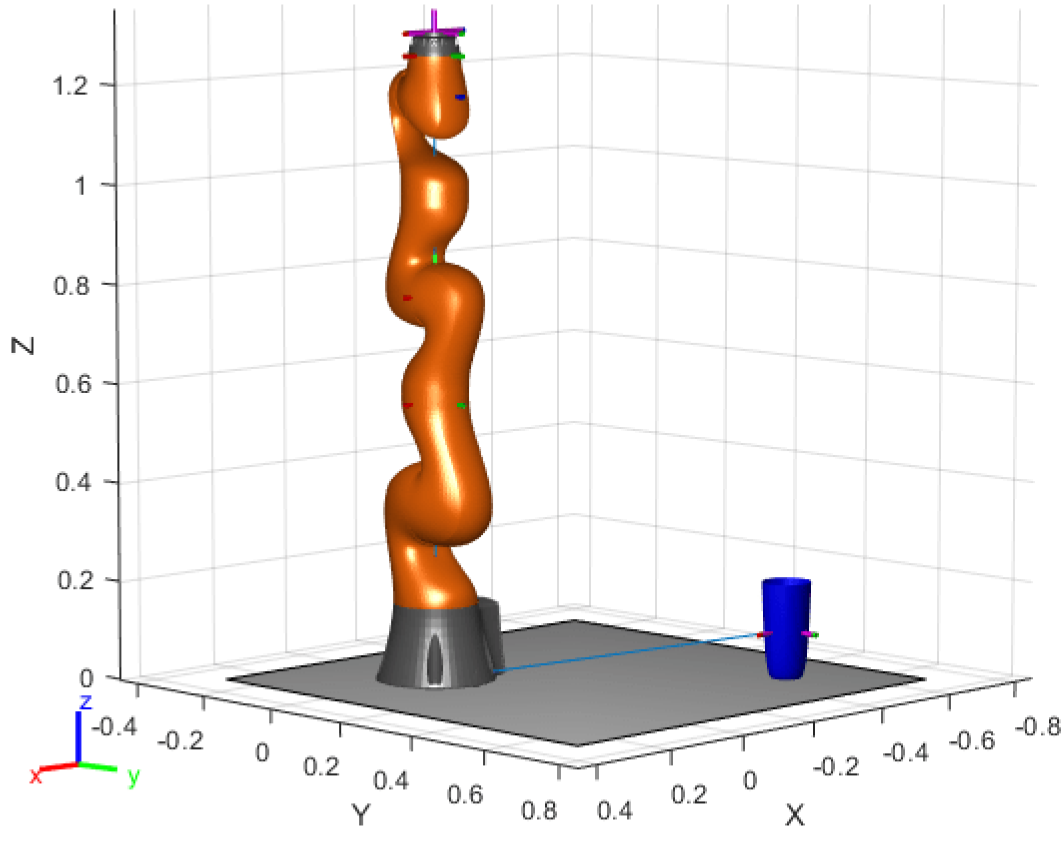
```
framerate = 15;
r = rateControl(framerate);
tFinal = 10;
tWaypoints = [0,linspace(tFinal/2,tFinal,size(qWaypoints,1)-1)];
numFrames = tFinal*framerate;
qInterp = pchip(tWaypoints,qWaypoints',linspace(0,tFinal,numFrames))';
```

Compute the gripper position for each interpolated configuration.

```
gripperPosition = zeros(numFrames,3);
for k = 1:numFrames
    gripperPosition(k,:) = tform2trvec(getTransform(lbr,qInterp(k,:), ...
                                                    gripper));
end
```

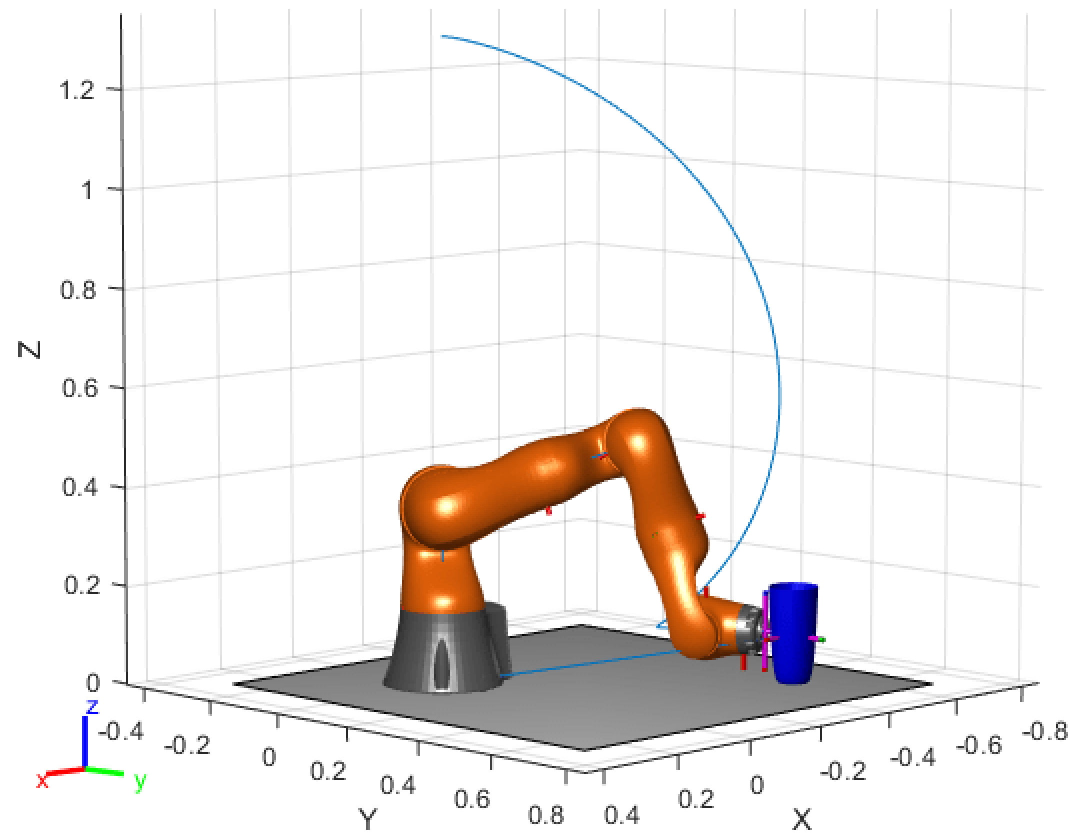
Show the robot in its initial configuration along with the table and cup

```
figure;
show(lbr, qWaypoints(1,:), 'PreservePlot', false);
hold on
exampleHelperPlotCupAndTable(cupHeight, cupRadius, cupPosition);
p = plot3(gripperPosition(1,1), gripperPosition(1,2), gripperPosition(1,3));
```



Animate the manipulator and plot the gripper position.

```
hold on
for k = 1:size(qInterp,1)
    show(lbr, qInterp(k,:), 'PreservePlot', false);
    p.XData(k) = gripperPosition(k,1);
    p.YData(k) = gripperPosition(k,2);
    p.ZData(k) = gripperPosition(k,3);
    waitfor(r);
end
hold off
```



If you want to save the generated configurations to a MAT-file for later use, execute the following:

```
>> save('lbr_trajectory.mat', 'tWaypoints', 'qWaypoints');
```

Version History

Introduced in R2017a

R2019b: constraintOrientationTarget was renamed

Behavior change in future release

The `constraintOrientationTarget` object was renamed from `robotics.OrientationTarget`. Use `constraintOrientationTarget` for all object creation.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

Objects

`generalizedInverseKinematics` | `constraintAiming` | `constraintCartesianBounds` | `constraintDistanceBounds` | `constraintJointBounds` | `constraintPoseTarget` |

`constraintPositionTarget` | `constraintFixedJoint` | `constraintPrismaticJoint` | `constraintRevoluteJoint`

Topics

"Plan a Reaching Trajectory With Multiple Kinematic Constraints"

constraintPoseTarget

Create constraint on relative pose of body

Description

The `constraintPoseTarget` object describes a constraint that requires the pose of one body (the end effector) to match a target pose within a distance and angular tolerance in any direction. The target pose is specified relative to the body frame of the reference body.

Constraint objects are used in `generalizedInverseKinematics` objects to specify multiple kinematic constraints on a robot.

For an example that uses multiple constraint objects, see “Plan a Reaching Trajectory With Multiple Kinematic Constraints”.

Creation

Syntax

```
poseConst = constraintPoseTarget(endeffector)
poseConst = constraintPoseTarget(endeffector,Name=Value)
```

Description

`poseConst = constraintPoseTarget(endeffector)` returns a pose target object that represents a constraint on the body of the robot model specified by `endeffector` and sets the `EndEffector` property.

`poseConst = constraintPoseTarget(endeffector,Name=Value)` returns a pose target object with each specified property name set to the specified value by one or more name-value pair arguments.

Properties

`EndEffector` — Name of the end effector
string scalar | character vector

Name of the end effector, specified as a string scalar or character vector. When using this constraint with `generalizedInverseKinematics`, the name must match a body specified in the robot model (`rigidBodyTree`).

Example: `"left_palm"`

Data Types: `char` | `string`

`ReferenceBody` — Name of the reference body frame
`''` (default) | string scalar | character vector

Name of the reference body frame, specified as a string scalar or character vector. The default `''` indicates that the constraint is relative to the base of the robot model. When using this constraint with `generalizedInverseKinematics`, the name must match a body specified in the robot model (`rigidBodyTree`).

Example:

Data Types: `char` | `string`

`TargetTransform` — Pose of the target frame relative to the reference body
`eye(4)` (default) | matrix

Pose of the target frame relative to the reference body, specified as a matrix. The matrix is a homogeneous transform that specifies the relative transformation to convert a point in the target frame to the reference body frame.

Example: `[1 0 0 1; 0 1 0 1; 0 0 1 1; 0 0 0 1]`

`OrientationTolerance` — Maximum allowed rotation angle
`0` (default) | numeric scalar

Maximum allowed rotation angle in radians, specified as a numeric scalar. This value is the upper bound on the magnitude of the rotation required to make the end-effector orientation match the target orientation.

Example:

`PositionTolerance` — Maximum allowed distance from target
`0` (default) | numeric scalar in meters

Maximum allowed distance from target, specified as a numeric scalar in meters. This value is the upper bound on the distance between the end-effector origin and the target position.

Example:

`Weights` — Weights of the constraint
`[1 1]` (default) | two-element vector

Weights of the constraint, specified as a two-element vector. Each element of the vector corresponds to the weight for the `PositionTolerance` and `OrientationTolerance` respectively. These weights are used with the `Weights` of all the constraints specified in `generalizedInverseKinematics` to properly balance each constraint.

Example:

Examples

Plan a Reaching Trajectory With Multiple Kinematic Constraints

This example shows how to use generalized inverse kinematics to plan a joint-space trajectory for a robotic manipulator. It combines multiple constraints to generate a trajectory that guides the gripper to a cup resting on a table. These constraints ensure that the gripper approaches the cup in a straight line and that the gripper remains at a safe distance from the table, without requiring the poses of the gripper to be determined in advance.

Set Up the Robot Model

This example uses a model of the KUKA LBR iiwa, a 7 degree-of-freedom robot manipulator. `importrobot` generates a `rigidBodyTree` model from a description stored in a Unified Robot Description Format (URDF) file.

```
lbr = importrobot('iiwa14.urdf'); % 14 kg payload version
lbr.DataFormat = 'row';
gripper = 'iiwa_link_ee_kuka';
```

Define dimensions for the cup.

```
cupHeight = 0.2;
cupRadius = 0.05;
cupPosition = [-0.5, 0.5, cupHeight/2];
```

Add a fixed body to the robot model representing the center of the cup.

```
body = rigidBody('cupFrame');
setFixedTransform(body.Joint, trvec2tform(cupPosition))
addBody(lbr, body, lbr.BaseName);
```

Define the Planning Problem

The goal of this example is to generate a sequence of robot configurations that satisfy the following criteria:

- Start in the home configuration
- No abrupt changes in robot configuration
- Keep the gripper at least 5 cm above the "table" (z = 0)
- The gripper should be aligned with the cup as it approaches
- Finish with the gripper 5 cm from the center of the cup

This example utilizes constraint objects to generate robot configurations that satisfy these criteria. The generated trajectory consists of five configuration waypoints. The first waypoint, `q0`, is set as the home configuration. Pre-allocate the rest of the configurations in `qWaypoints` using `repmat`.

```
numWaypoints = 5;
q0 = homeConfiguration(lbr);
qWaypoints = repmat(q0, numWaypoints, 1);
```

Create a `generalizedInverseKinematics` solver that accepts the following constraint inputs:

- Cartesian bounds - Limits the height of the gripper
- A position target - Specifies the position of the cup relative to the gripper.
- An aiming constraint - Aligns the gripper with the cup axis
- An orientation target - Maintains a fixed orientation for the gripper while approaching the cup
- Joint position bounds - Limits the change in joint positions between waypoints.

```
gik = generalizedInverseKinematics('RigidBodyTree', lbr, ...
    'ConstraintInputs', {'cartesian','position','aiming','orientation','joint'})
```

```
gik = 
  generalizedInverseKinematics with properties:
```

```
      NumConstraints: 5
    ConstraintInputs: {'cartesian'  'position'  'aiming'  'orientation'  'joint'}
       RigidBodyTree: [1x1 rigidBodyTree]
     SolverAlgorithm: 'BFGSGradientProjection'
    SolverParameters: [1x1 struct]
```

Create Constraint Objects

Create the constraint objects that are passed as inputs to the solver. These object contain the parameters needed for each constraint. Modify these parameters between calls to the solver as necessary.

Create a Cartesian bounds constraint that requires the gripper to be at least 5 cm above the table (negative z direction). All other values are given as `inf` or `-inf`.

```
heightAboveTable = constraintCartesianBounds(gripper);
heightAboveTable.Bounds = [-inf, inf; ...
                           -inf, inf; ...
                           0.05, inf]
```

```
heightAboveTable =
  constraintCartesianBounds with properties:

        EndEffector: 'iiwa_link_ee_kuka'
      ReferenceBody: ''
    TargetTransform: [4x4 double]
             Bounds: [3x2 double]
            Weights: [1 1 1]
```

Create a constraint on the position of the cup relative to the gripper, with a tolerance of 5 mm.

```
distanceFromCup = constraintPositionTarget('cupFrame');
distanceFromCup.ReferenceBody = gripper;
distanceFromCup.PositionTolerance = 0.005
```

```
distanceFromCup =
  constraintPositionTarget with properties:

          EndEffector: 'cupFrame'
        ReferenceBody: 'iiwa_link_ee_kuka'
       TargetPosition: [0 0 0]
    PositionTolerance: 0.0050
              Weights: 1
```

Create an aiming constraint that requires the z-axis of the `iiwa_link_ee` frame to be approximately vertical, by placing the target far above the robot. The `iiwa_link_ee` frame is oriented such that this constraint aligns the gripper with the axis of the cup.

```
alignWithCup = constraintAiming('iiwa_link_ee');
alignWithCup.TargetPoint = [0, 0, 100]
```

```
alignWithCup =
  constraintAiming with properties:
```

```
         EndEffector: 'iiwa_link_ee'
       ReferenceBody: ''
         TargetPoint: [0 0 100]
    AngularTolerance: 0
             Weights: 1
```

Create a joint position bounds constraint. Set the `Bounds` property of this constraint based on the previous configuration to limit the change in joint positions.

```
limitJointChange = constraintJointBounds(lbr)
```

```
limitJointChange = 
  constraintJointBounds with properties:

     Bounds: [7x2 double]
    Weights: [1 1 1 1 1 1 1]
```

Create an orientation constraint for the gripper with a tolerance of one degree. This constraint requires the orientation of the gripper to match the value specified by the `TargetOrientation` property. Use this constraint to fix the orientation of the gripper during the final approach to the cup.

```
fixOrientation = constraintOrientationTarget(gripper);
fixOrientation.OrientationTolerance = deg2rad(1)
```

```
fixOrientation = 
  constraintOrientationTarget with properties:

             EndEffector: 'iiwa_link_ee_kuka'
           ReferenceBody: ''
       TargetOrientation: [1 0 0 0]
    OrientationTolerance: 0.0175
                 Weights: 1
```

Find a Configuration That Points at the Cup

This configuration should place the gripper at a distance from the cup, so that the final approach can be made with the gripper properly aligned.

```
intermediateDistance = 0.3;
```

Constraint objects have a `Weights` property which determines how the solver treats conflicting constraints. Setting the weights of a constraint to zero disables the constraint. For this configuration, disable the joint position bounds and orientation constraint.

```
limitJointChange.Weights = zeros(size(limitJointChange.Weights));
fixOrientation.Weights = 0;
```

Set the target position for the cup in the gripper frame. The cup should lie on the z-axis of the gripper at the specified distance.

```
distanceFromCup.TargetPosition = [0,0,intermediateDistance];
```

Solve for the robot configuration that satisfies the input constraints using the `gik` solver. You must specify all the input constraints. Set that configuration as the second waypoint.

```
[qWaypoints(2,:),solutionInfo] = gik(q0, heightAboveTable, ...
                       distanceFromCup, alignWithCup, fixOrientation, ...
                       limitJointChange);
```

Find Configurations That Move Gripper to the Cup Along a Straight Line

Re-enable the joint position bound and orientation constraints.

```
limitJointChange.Weights = ones(size(limitJointChange.Weights));
fixOrientation.Weights = 1;
```

Disable the align-with-cup constraint, as the orientation constraint makes it redundant.

```
alignWithCup.Weights = 0;
```

Set the orientation constraint to hold the orientation based on the previous configuration (`qWaypoints(2,:)`). Get the transformation from the gripper to the base of the robot model. Convert the homogeneous transformation to a quaternion.

```
fixOrientation.TargetOrientation = ...
    tform2quat(getTransform(lbr,qWaypoints(2,:),gripper));
```

Define the distance between the cup and gripper for each waypoint

```
finalDistanceFromCup = 0.05;
distanceFromCupValues = linspace(intermediateDistance, finalDistanceFromCup, numWaypoints-1);
```

Define the maximum allowed change in joint positions between each waypoint.

```
maxJointChange = deg2rad(10);
```

Call the solver for each remaining waypoint.

```
for k = 3:numWaypoints
    % Update the target position.
    distanceFromCup.TargetPosition(3) = distanceFromCupValues(k-1);
    % Restrict the joint positions to lie close to their previous values.
    limitJointChange.Bounds = [qWaypoints(k-1,:)' - maxJointChange, ...
                               qWaypoints(k-1,:)' + maxJointChange];
    % Solve for a configuration and add it to the waypoints array.
    [qWaypoints(k,:),solutionInfo] = gik(qWaypoints(k-1,:), ...
                                         heightAboveTable, ...
                                         distanceFromCup, alignWithCup, ...
                                         fixOrientation, limitJointChange);
end
```

Visualize the Generated Trajectory

Interpolate between the waypoints to generate a smooth trajectory. Use `pchip` to avoid overshoots, which might violate the joint limits of the robot.

```
framerate = 15;
r = rateControl(framerate);
tFinal = 10;
tWaypoints = [0,linspace(tFinal/2,tFinal,size(qWaypoints,1)-1)];
numFrames = tFinal*framerate;
qInterp = pchip(tWaypoints,qWaypoints',linspace(0,tFinal,numFrames))';
```

Compute the gripper position for each interpolated configuration.

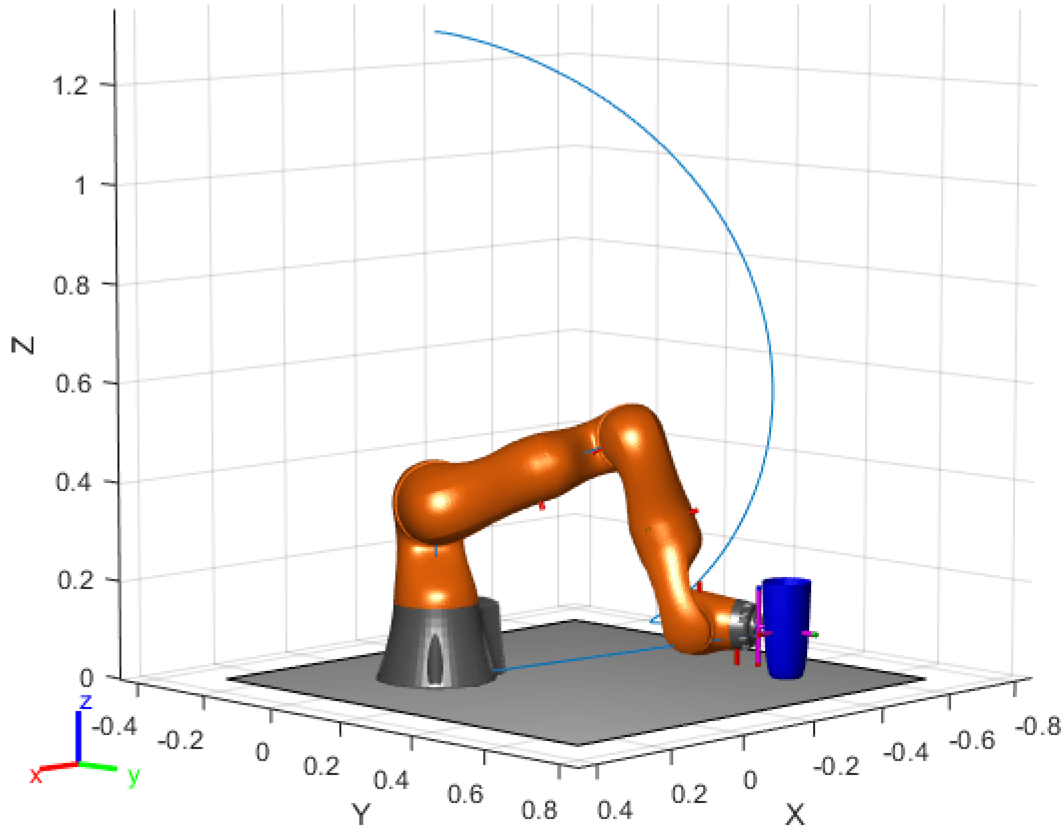
```
gripperPosition = zeros(numFrames,3);
for k = 1:numFrames
    gripperPosition(k,:) = tform2trvec(getTransform(lbr,qInterp(k,:), ...
                                                    gripper));
end
```

Show the robot in its initial configuration along with the table and cup

```
figure;
show(lbr, qWaypoints(1,:), 'PreservePlot', false);
hold on
exampleHelperPlotCupAndTable(cupHeight, cupRadius, cupPosition);
p = plot3(gripperPosition(1,1), gripperPosition(1,2), gripperPosition(1,3));
```

Animate the manipulator and plot the gripper position.

```
hold on
for k = 1:size(qInterp,1)
    show(lbr, qInterp(k,:), 'PreservePlot', false);
    p.XData(k) = gripperPosition(k,1);
    p.YData(k) = gripperPosition(k,2);
    p.ZData(k) = gripperPosition(k,3);
    waitfor(r);
end
hold off
```



If you want to save the generated configurations to a MAT-file for later use, execute the following:

```
>> save('lbr_trajectory.mat', 'tWaypoints', 'qWaypoints');
```

Version History

Introduced in R2017a

R2019b: `constraintPoseTarget` was renamed

Behavior change in future release

The `constraintPoseTarget` object was renamed from `robotics.PoseTarget`. Use `constraintPoseTarget` for all object creation.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

Objects

`generalizedInverseKinematics` | `constraintAiming` | `constraintCartesianBounds` | `constraintDistanceBounds` | `constraintJointBounds` | `constraintOrientationTarget` |

`constraintPositionTarget` | `constraintFixedJoint` | `constraintPrismaticJoint` | `constraintRevoluteJoint`

Topics

"Plan a Reaching Trajectory With Multiple Kinematic Constraints"

constraintPositionTarget

Create constraint on relative position of body

Description

The `constraintPositionTarget` object describes a constraint that requires the position of one body (the end effector) to match a target position within a distance tolerance in any direction. The target position is specified relative to the body frame of the reference body.

Constraint objects are used in `generalizedInverseKinematics` objects to specify multiple kinematic constraints on a robot.

For an example that uses multiple constraint objects, see "Plan a Reaching Trajectory With Multiple Kinematic Constraints".

Creation

Syntax

```
positionConst = constraintPositionTarget(endeffector)
positionConst = constraintPositionTarget(endeffector,Name=Value)
```

Description

`positionConst = constraintPositionTarget(endeffector)` returns a position target object that represents a constraint on the body of the robot model specified by `endeffector` and sets the `EndEffector` property.

`positionConst = constraintPositionTarget(endeffector,Name=Value)` returns a position target object with each specified property name set to the specified value by one or more name-value pair arguments.

Properties

`EndEffector` — Name of the end effector
string scalar | character vector

Name of the end effector, specified as a string scalar or character vector. When using this constraint with `generalizedInverseKinematics`, the name must match a body specified in the robot model (`rigidBodyTree`).

Example: `"left_palm"`

Data Types: `char` | `string`

`ReferenceBody` — Name of the reference body frame
`''` (default) | character vector

Name of the reference body frame, specified as a character vector. The default `''` indicates that the constraint is relative to the base of the robot model. When using this constraint with `generalizedInverseKinematics`, the name must match a body specified in the robot model (`rigidBodyTree`).

Example:

`TargetPosition` — Position of the target relative to the reference body
`[0 0 0]` (default) | `[x y z]` vector

Position of the target relative to the reference body, specified as an `[x y z]` vector. The target position is a point specified in the reference body frame.

Example:

`PositionTolerance` — Maximum allowed distance from target
`0` (default) | numeric scalar

Maximum allowed distance from target in meters, specified as a numeric scalar. This value is the upper bound on the distance between the end-effector origin and the target position.

Example:

`Weights` — Weight of the constraint
`1` (default) | numeric scalar

Weight of the constraint, specified as a numeric scalar. This weight is used with the `Weights` property of all the constraints specified in `generalizedInverseKinematics` to properly balance each constraint.

Example:

Examples

Plan a Reaching Trajectory With Multiple Kinematic Constraints

This example shows how to use generalized inverse kinematics to plan a joint-space trajectory for a robotic manipulator. It combines multiple constraints to generate a trajectory that guides the gripper to a cup resting on a table. These constraints ensure that the gripper approaches the cup in a straight line and that the gripper remains at a safe distance from the table, without requiring the poses of the gripper to be determined in advance.

Set Up the Robot Model

This example uses a model of the KUKA LBR iiwa, a 7 degree-of-freedom robot manipulator. `importrobot` generates a `rigidBodyTree` model from a description stored in a Unified Robot Description Format (URDF) file.

```
lbr = importrobot('iiwa14.urdf'); % 14 kg payload version
lbr.DataFormat = 'row';
gripper = 'iiwa_link_ee_kuka';
```

Define dimensions for the cup.

```
cupHeight = 0.2;
cupRadius = 0.05;
cupPosition = [-0.5, 0.5, cupHeight/2];
```

Add a fixed body to the robot model representing the center of the cup.

```
body = rigidBody('cupFrame');
setFixedTransform(body.Joint, trvec2tform(cupPosition))
addBody(lbr, body, lbr.BaseName);
```

Define the Planning Problem

The goal of this example is to generate a sequence of robot configurations that satisfy the following criteria:

- Start in the home configuration
- No abrupt changes in robot configuration
- Keep the gripper at least 5 cm above the "table" (z = 0)
- The gripper should be aligned with the cup as it approaches
- Finish with the gripper 5 cm from the center of the cup

This example utilizes constraint objects to generate robot configurations that satisfy these criteria. The generated trajectory consists of five configuration waypoints. The first waypoint, `q0`, is set as the home configuration. Pre-allocate the rest of the configurations in `qWaypoints` using `repmat`.

```
numWaypoints = 5;
q0 = homeConfiguration(lbr);
qWaypoints = repmat(q0, numWaypoints, 1);
```

Create a `generalizedInverseKinematics` solver that accepts the following constraint inputs:

- Cartesian bounds - Limits the height of the gripper
- A position target - Specifies the position of the cup relative to the gripper.
- An aiming constraint - Aligns the gripper with the cup axis
- An orientation target - Maintains a fixed orientation for the gripper while approaching the cup
- Joint position bounds - Limits the change in joint positions between waypoints.

```
gik = generalizedInverseKinematics('RigidBodyTree', lbr, ...
    'ConstraintInputs', {'cartesian','position','aiming','orientation','joint'})
```

```
gik = 
  generalizedInverseKinematics with properties:

      NumConstraints: 5
    ConstraintInputs: {'cartesian'  'position'  'aiming'  'orientation'  'joint'}
       RigidBodyTree: [1x1 rigidBodyTree]
     SolverAlgorithm: 'BFGSGradientProjection'
    SolverParameters: [1x1 struct]
```

Create Constraint Objects

Create the constraint objects that are passed as inputs to the solver. These object contain the parameters needed for each constraint. Modify these parameters between calls to the solver as necessary.

Create a Cartesian bounds constraint that requires the gripper to be at least 5 cm above the table (negative z direction). All other values are given as `inf` or `-inf`.

```
heightAboveTable = constraintCartesianBounds(gripper);
heightAboveTable.Bounds = [-inf, inf; ...
                           -inf, inf; ...
                           0.05, inf]
```

```
heightAboveTable = 
  constraintCartesianBounds with properties:

        EndEffector: 'iiwa_link_ee_kuka'
      ReferenceBody: ''
    TargetTransform: [4x4 double]
             Bounds: [3x2 double]
            Weights: [1 1 1]
```

Create a constraint on the position of the cup relative to the gripper, with a tolerance of 5 mm.

```
distanceFromCup = constraintPositionTarget('cupFrame');
distanceFromCup.ReferenceBody = gripper;
distanceFromCup.PositionTolerance = 0.005
```

```
distanceFromCup = 
  constraintPositionTarget with properties:

          EndEffector: 'cupFrame'
        ReferenceBody: 'iiwa_link_ee_kuka'
       TargetPosition: [0 0 0]
    PositionTolerance: 0.0050
              Weights: 1
```

Create an aiming constraint that requires the z-axis of the `iiwa_link_ee` frame to be approximately vertical, by placing the target far above the robot. The `iiwa_link_ee` frame is oriented such that this constraint aligns the gripper with the axis of the cup.

```
alignWithCup = constraintAiming('iiwa_link_ee');
alignWithCup.TargetPoint = [0, 0, 100]
```

```
alignWithCup = 
  constraintAiming with properties:

         EndEffector: 'iiwa_link_ee'
       ReferenceBody: ''
         TargetPoint: [0 0 100]
    AngularTolerance: 0
             Weights: 1
```

Create a joint position bounds constraint. Set the `Bounds` property of this constraint based on the previous configuration to limit the change in joint positions.

```
limitJointChange = constraintJointBounds(lbr)
```

```
limitJointChange = 
  constraintJointBounds with properties:
```

```
     Bounds: [7x2 double]
    Weights: [1 1 1 1 1 1 1]
```

Create an orientation constraint for the gripper with a tolerance of one degree. This constraint requires the orientation of the gripper to match the value specified by the `TargetOrientation` property. Use this constraint to fix the orientation of the gripper during the final approach to the cup.

```
fixOrientation = constraintOrientationTarget(gripper);
fixOrientation.OrientationTolerance = deg2rad(1)
```

```
fixOrientation = 
  constraintOrientationTarget with properties:

              EndEffector: 'iiwa_link_ee_kuka'
            ReferenceBody: ''
        TargetOrientation: [1 0 0 0]
    OrientationTolerance: 0.0175
                  Weights: 1
```

Find a Configuration That Points at the Cup

This configuration should place the gripper at a distance from the cup, so that the final approach can be made with the gripper properly aligned.

```
intermediateDistance = 0.3;
```

Constraint objects have a `Weights` property which determines how the solver treats conflicting constraints. Setting the weights of a constraint to zero disables the constraint. For this configuration, disable the joint position bounds and orientation constraint.

```
limitJointChange.Weights = zeros(size(limitJointChange.Weights));
fixOrientation.Weights = 0;
```

Set the target position for the cup in the gripper frame. The cup should lie on the z-axis of the gripper at the specified distance.

```
distanceFromCup.TargetPosition = [0,0,intermediateDistance];
```

Solve for the robot configuration that satisfies the input constraints using the `gik` solver. You must specify all the input constraints. Set that configuration as the second waypoint.

```
[qWaypoints(2,:),solutionInfo] = gik(q0, heightAboveTable, ...
                       distanceFromCup, alignWithCup, fixOrientation, ...
                       limitJointChange);
```

Find Configurations That Move Gripper to the Cup Along a Straight Line

Re-enable the joint position bound and orientation constraints.

```
limitJointChange.Weights = ones(size(limitJointChange.Weights));
fixOrientation.Weights = 1;
```

Disable the align-with-cup constraint, as the orientation constraint makes it redundant.

```
alignWithCup.Weights = 0;
```

Set the orientation constraint to hold the orientation based on the previous configuration (`qWaypoints(2,:)`). Get the transformation from the gripper to the base of the robot model. Convert the homogeneous transformation to a quaternion.

```
fixOrientation.TargetOrientation = ...
    tform2quat(getTransform(lbr,qWaypoints(2,:),gripper));
```

Define the distance between the cup and gripper for each waypoint

```
finalDistanceFromCup = 0.05;
distanceFromCupValues = linspace(intermediateDistance, finalDistanceFromCup, numWaypoints-1);
```

Define the maximum allowed change in joint positions between each waypoint.

```
maxJointChange = deg2rad(10);
```

Call the solver for each remaining waypoint.

```
for k = 3:numWaypoints
    % Update the target position.
    distanceFromCup.TargetPosition(3) = distanceFromCupValues(k-1);
    % Restrict the joint positions to lie close to their previous values.
    limitJointChange.Bounds = [qWaypoints(k-1,:)' - maxJointChange, ...
                               qWaypoints(k-1,:)' + maxJointChange];
    % Solve for a configuration and add it to the waypoints array.
    [qWaypoints(k,:),solutionInfo] = gik(qWaypoints(k-1,:), ...
                                         heightAboveTable, ...
                                         distanceFromCup, alignWithCup, ...
                                         fixOrientation, limitJointChange);
end
```

Visualize the Generated Trajectory

Interpolate between the waypoints to generate a smooth trajectory. Use `pchip` to avoid overshoots, which might violate the joint limits of the robot.

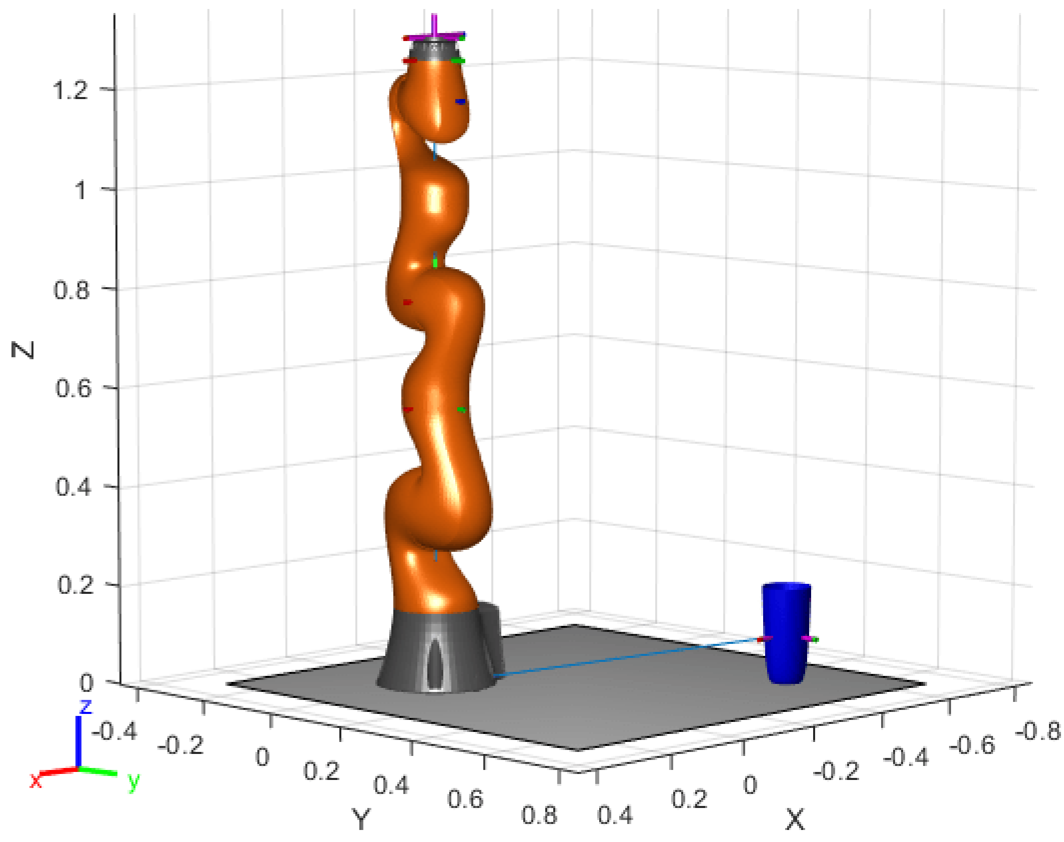
```
framerate = 15;
r = rateControl(framerate);
tFinal = 10;
tWaypoints = [0,linspace(tFinal/2,tFinal,size(qWaypoints,1)-1)];
numFrames = tFinal*framerate;
qInterp = pchip(tWaypoints,qWaypoints',linspace(0,tFinal,numFrames))';
```

Compute the gripper position for each interpolated configuration.

```
gripperPosition = zeros(numFrames,3);
for k = 1:numFrames
    gripperPosition(k,:) = tform2trvec(getTransform(lbr,qInterp(k,:), ...
                                                    gripper));
end
```

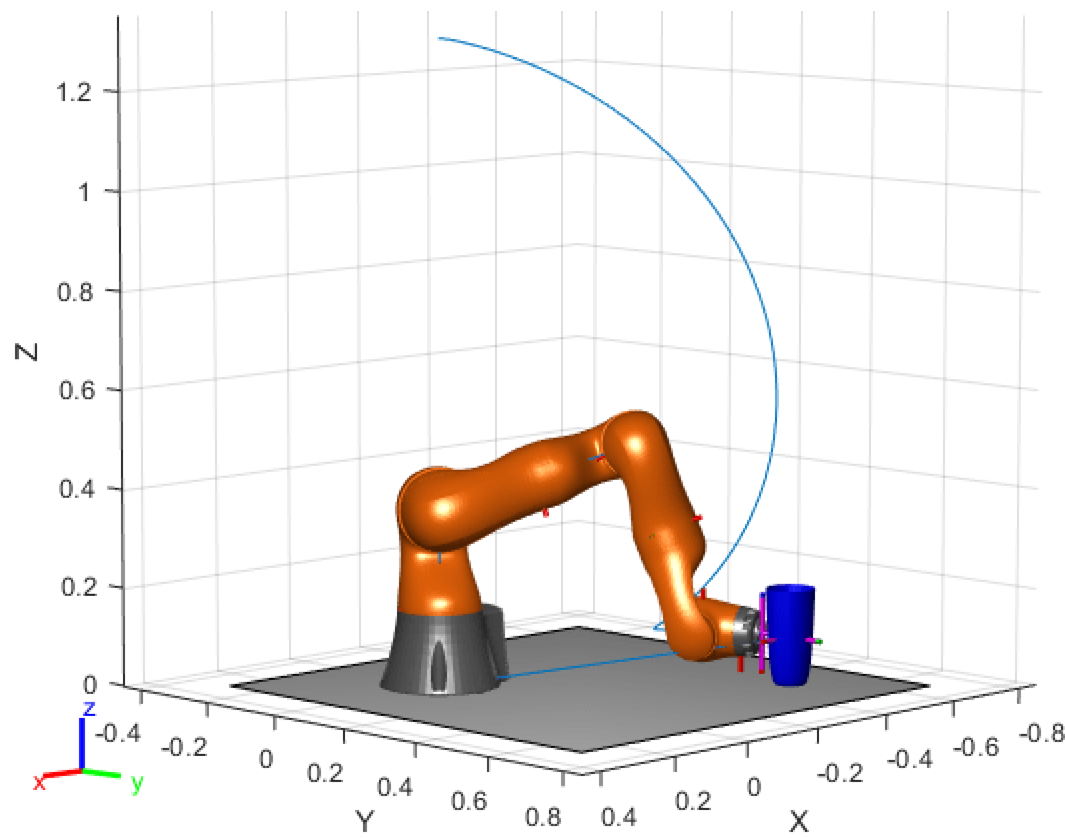
Show the robot in its initial configuration along with the table and cup

```
figure;
show(lbr, qWaypoints(1,:), 'PreservePlot', false);
hold on
exampleHelperPlotCupAndTable(cupHeight, cupRadius, cupPosition);
p = plot3(gripperPosition(1,1), gripperPosition(1,2), gripperPosition(1,3));
```



Animate the manipulator and plot the gripper position.

```
hold on
for k = 1:size(qInterp,1)
    show(lbr, qInterp(k,:), 'PreservePlot', false);
    p.XData(k) = gripperPosition(k,1);
    p.YData(k) = gripperPosition(k,2);
    p.ZData(k) = gripperPosition(k,3);
    waitfor(r);
end
hold off
```



If you want to save the generated configurations to a MAT-file for later use, execute the following:

```
>> save('lbr_trajectory.mat', 'tWaypoints', 'qWaypoints');
```

Version History

Introduced in R2017a

R2019b: `constraintPositionTarget` was renamed

Behavior change in future release

The `constraintPositionTarget` object was renamed from `robotics.PositionTarget`. Use `constraintPositionTarget` for all object creation.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

Objects

`generalizedInverseKinematics` | `constraintAiming` | `constraintCartesianBounds` | `constraintDistanceBounds` | `constraintJointBounds` | `constraintOrientationTarget` |

`constraintPoseTarget` | `constraintFixedJoint` | `constraintPrismaticJoint` | `constraintRevoluteJoint`

Topics

"Plan a Reaching Trajectory With Multiple Kinematic Constraints"

constraintPrismaticJoint

Prismatic joint constraint between bodies

Description

The `constraintPrismaticJoint` object describes a closed-loop prismatic joint constraint between a successor and predecessor body on the same `rigidBodyTree`. This constraint is satisfied when the intermediate frame origin of the successor body lies on the *Z*-axis of the intermediate frame of the predecessor body, and there is no relative orientation between the frames. When satisfied, this constraint allows linear motion along the common *Z*-axes of the intermediate frames.

Creation

Syntax

```
prisConst = constraintPrismaticJoint(successorbody,predecessorbody)
prisConst = constraintPrismaticJoint( ___ ,Name=Value)
```

Description

`prisConst = constraintPrismaticJoint(successorbody,predecessorbody)` returns a prismatic constraint object, `prisConst`, that represents a constraint between the specified successor body `successorbody` and predecessor body `predecessorbody` of the joint. The `successorbody` and `predecessor` arguments set the `SuccessorBody` and `PredecessorBody` properties, respectively..

`prisConst = constraintPrismaticJoint( ___ ,Name=Value)` specifies properties using one more name-value pair arguments in addition to all input arguments from the previous syntax.

Properties

`SuccessorBody` — Name of successor body of joint
string scalar | character vector

Name of the successor body frame, specified as a string scalar or character vector. When using this constraint with the `generalizedInverseKinematics` inverse kinematics (IK) solver, the name must match a body specified in the `RigidBodyTree` of the `generalizedInverseKinematics` object.

`PredecessorBody` — Name of predecessor body of joint
string scalar | character vector

Name of the predecessor body frame, specified as a string scalar or character vector. When using this constraint with the `generalizedInverseKinematics` inverse kinematics (IK) solver, the name must match a body specified in the `RigidBodyTree` of the `generalizedInverseKinematics` object.

`SuccessorTransform` — Fixed transform of joint constraint with respect to successor body frame
`eye(4)` (default) | 4-by-4 matrix

Fixed transform of the joint constraint with respect to the successor body frame, specified as 4-by-4 matrix.

Example: `[1 0 0 1; 0 1 0 1; 0 0 1 1; 0 0 0 1]`

`PredecessorTransform` — Fixed transform of joint constraint with respect to predecessor body frame
`eye(4)` (default) | 4-by-4 matrix

Fixed transform of the joint constraint with respect to the predecessor body frame, specified as 4-by-4 matrix.

Example: `[1 0 0 1; 0 1 0 1; 0 0 1 1; 0 0 0 1]`

`PositionTolerance` — Position tolerance of joint constraint
`0` (default) | nonnegative scalar

Position tolerance of the joint constraint, in meters, specified as a nonnegative scalar.

`JointPositionLimits` — Position limits of joint constraint
`[-100 100]` (default) | two-element row vector

Position limits of the joint constraint, in meters, specified as a two-element row vector of the form `[minimum maximum]`.

Example: `[-25 50]`

`OrientationTolerance` — Orientation tolerance of joint constraint
`0` (default) | nonnegative scalar

Orientation tolerance of the joint constraint, in radians, specified as a nonnegative scalar.

`Weights` — Weights of the constraint
`[1 1 1]` (default) | three-element row vector

Weights of the constraint, specified as a three-element vector. The elements of the vector correspond to the weights for the `PositionTolerance`, `OrientationTolerance`, and `JointPositionLimits` properties, respectively. These weights are used with the weights of all the constraints specified in the `generalizedInverseKinematics` solver, and can be used to specify the relative importance of a constraint violation to the solver.

Example: `[0 1 4]`

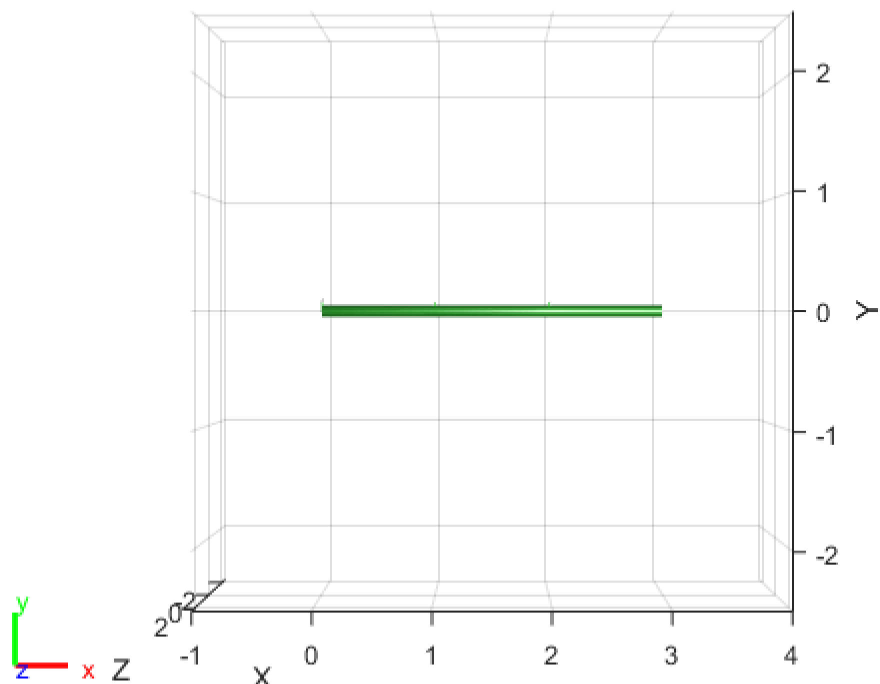
Examples

Create Loop-Closure Joint Constraints

Create a revolute, prismatic, and fixed joint constraints for a simple rigid body tree.

Use the `exampleHelperFourBarLinkageTree` helper function to create a simple robot model to demonstrate the closed-loop constraints.

```
rbt = exampleHelperFourBarLinkageTree;
show(rbt,Collisions="on");
view([0 0 pi])
xlim([-1 4])
```



Revolute Joint Constraint

To demonstrate a revolute joint constraint, create a four-bar linkage by connecting the end of the last link, `link3`, and the first link, `link0`.

Create a generalized inverse kinematics solver with a revolute joint constraint and a joint bounds constraint.

```
gikSolverWithRevoluteJointConstraint = generalizedInverseKinematics(RigidBodyTree=rbt, ...
    ConstraintInputs={'revolute','jointbounds'});
```

To ensure repeatable IK solutions, disable random restarts.

```
gikSolverWithRevoluteJointConstraint.SolverParameters.AllowRandomRestart = false;
theta = pi/2+pi/4;
```

Fix the first joint by setting `theta` as both the minimum and maximum bound.

```
activeJointConstraint = constraintJointBounds(rbt);
activeJointConstraint.Weights = [1 0 0];
activeJointConstraint.Bounds(1,:) = [theta theta];
```

Create a revolute joint constraint with successor and predecessor bodies set to the last link `link3` and the first link `link0`, respectively. Specify predecessor and successor transforms that create

intermediate frames 1 meter away, in the *X*-axis, from their respective body. Once defined, these intermediate frames move such that their frame origins coincide when their *Z*-axes align.

```
cRev = constraintRevoluteJoint("link3","link0", ...
    PredecessorTransform=trvec2tform([1 0 0]), ...
    SuccessorTransform=trvec2tform([1 0 0]));
```

Provide `[theta 0 0]` as an initial guess to the solver, along with the constraints.

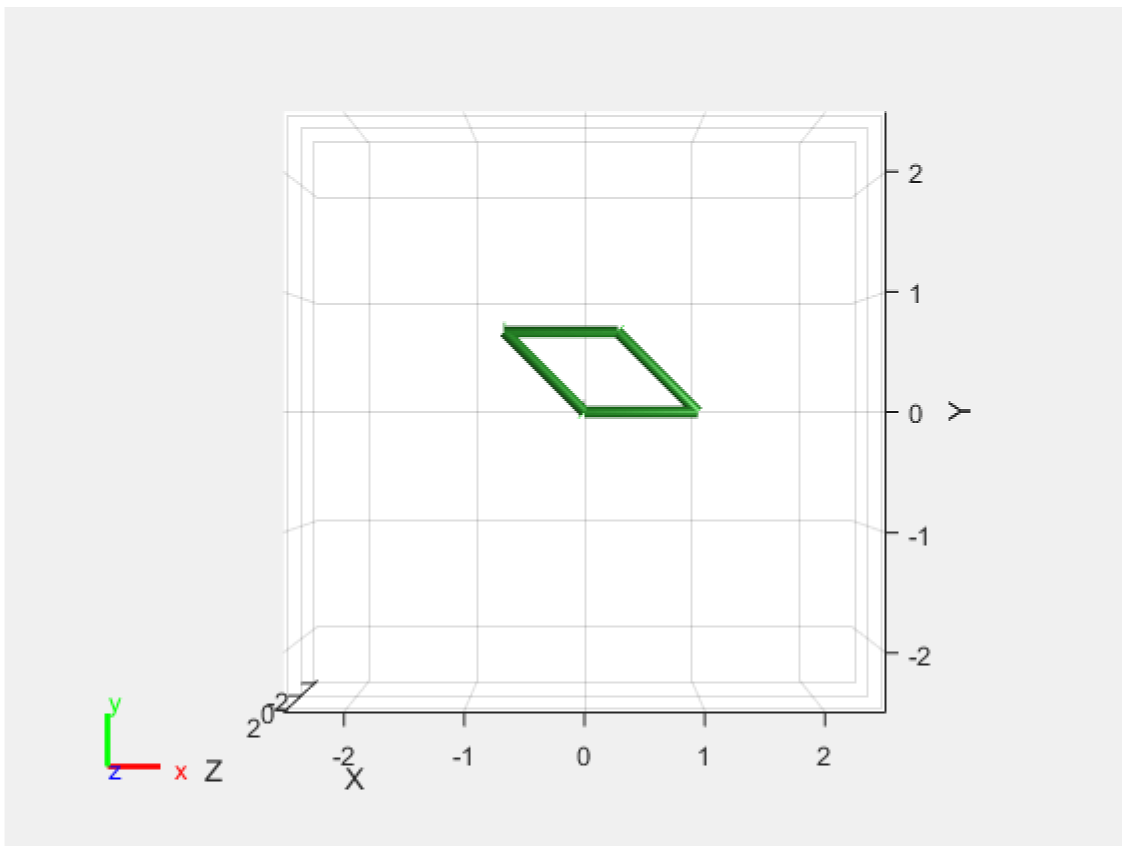
```
qConst = gikSolverWithRevoluteJointConstraint([theta 0 0],cRev,activeJointConstraint);
```

Visualize the robot to see the robot acting as a four-bar linkage. If the first joint rotates, the solver tries to keep the intermediate frames of the revolute joint constraint coincident, acting as a joint and resulting in four-bar motion.

```
figure(Name="Revolute Joint Constraint")
show(rbt,qConst,Collisions="on")
```

```
ans = 
  Axes (Primary) with properties:

             XLim: [-2.5000 2.5000]
             YLim: [-2.5000 2.5000]
           XScale: 'linear'
           YScale: 'linear'
    GridLineStyle: '-'
         Position: [0.1300 0.1100 0.7750 0.8150]
            Units: 'normalized'

  Show all properties
```

```
view([0 0 pi])
```



Prismatic Joint Constraint

Use a prismatic joint constraint to create a slider-crank. Create a new solver with a prismatic joint constraint and a joint bounds constraint.

```
gikSolverWithPrismaticJointConstraint = generalizedInverseKinematics(RigidBodyTree=rbt, ...
    ConstraintInputs={'prismatic','jointbounds'});
gikSolverWithPrismaticJointConstraint.SolverParameters.AllowRandomRestart=false;
```

Create the prismatic joint constraint with `link3` and `link0` as the successor and predecessor bodies, respectively, and set the predecessor transform such that the predecessor intermediate frame is `1` meter away on the *X*-axis and rotated `pi/2` in the *Y*-axis from the predecessor body frame.

```
cPris=constraintPrismaticJoint("link3","link0",PredecessorTransform=trvec2tform([1 0 0])*eul2tfor
```

Provide `[theta 0 0]` as an initial guess to the solver along with the constraints.

```
qConst = gikSolverWithPrismaticJointConstraint([theta 0 0],cPris,activeJointConstraint);
```

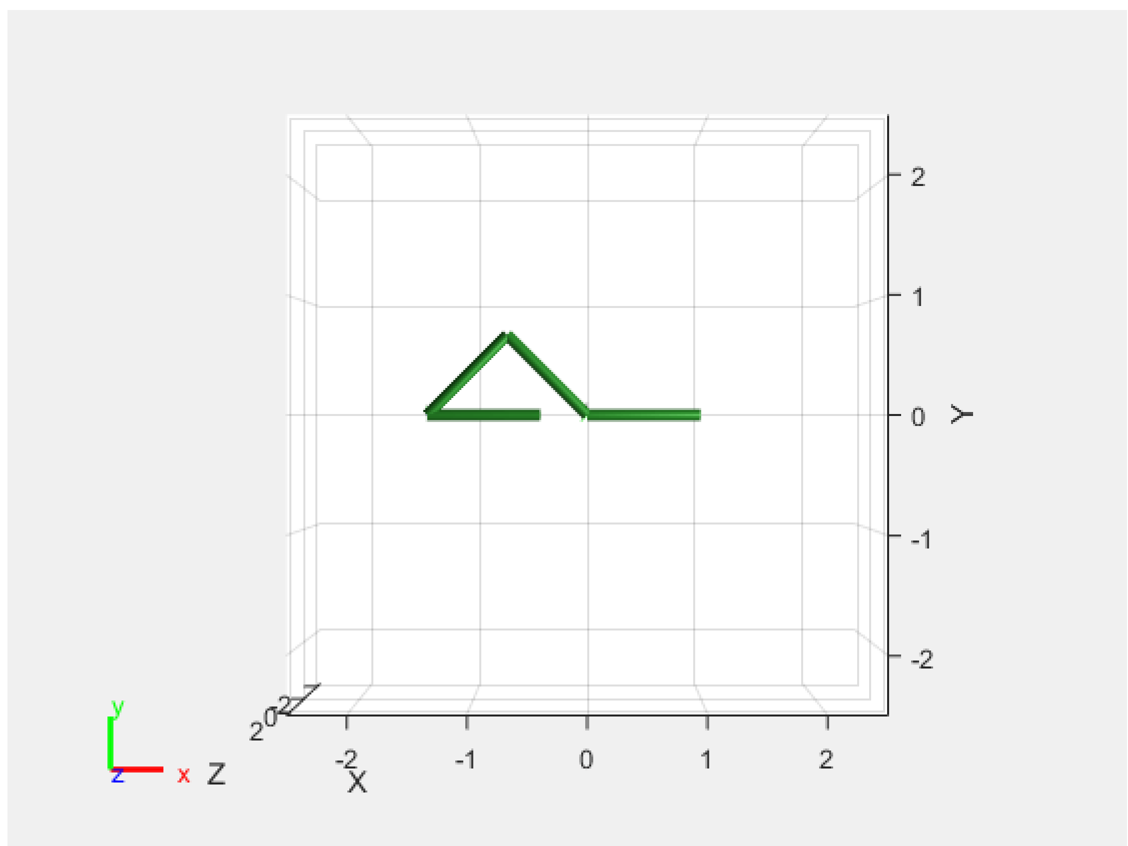
Visualize the robot to see the robot acting as a slider-crank. If the first joint rotates, the solver tries to keep the intermediate frames of the prismatic joint constraint coincident, acting as a joint and resulting in slider-crank motion.

```
figure(Name="Prismatic Joint Constraint")
show(rbt,qConst,Collisions="on")
```

```
ans = 
  Axes (Primary) with properties:
```

```
            XLim: [-2.5000 2.5000]
            YLim: [-2.5000 2.5000]
          XScale: 'linear'
          YScale: 'linear'
   GridLineStyle: '-'
        Position: [0.1300 0.1100 0.7750 0.8150]
           Units: 'normalized'

  Show all properties
```

```
view([0 0 pi])
```



Fixed Joint Constraint

To demonstrate a fixed joint constraint, create a triangle with the links that is preserved when the first joint moves. Create a new solver with a fixed joint constraint.

```
gikSolverWithFixedJointConstraint = generalizedInverseKinematics(RigidBodyTree=rbt, ...
    ConstraintInputs={'fixed'});
```

Create the fixed joint constraint with `link3` and `link0` as the successor and predecessor bodies, respectively, and set the successor transform such that the predecessor intermediate frame is `1` meter away on the *X*-axis from the predecessor body frame.

```
cFix = constraintFixedJoint("link3","link1",SuccessorTransform=trvec2tform([1 0 0]));
```

Set the weight of the orientation constraint of the fixed joint constraint to 0.

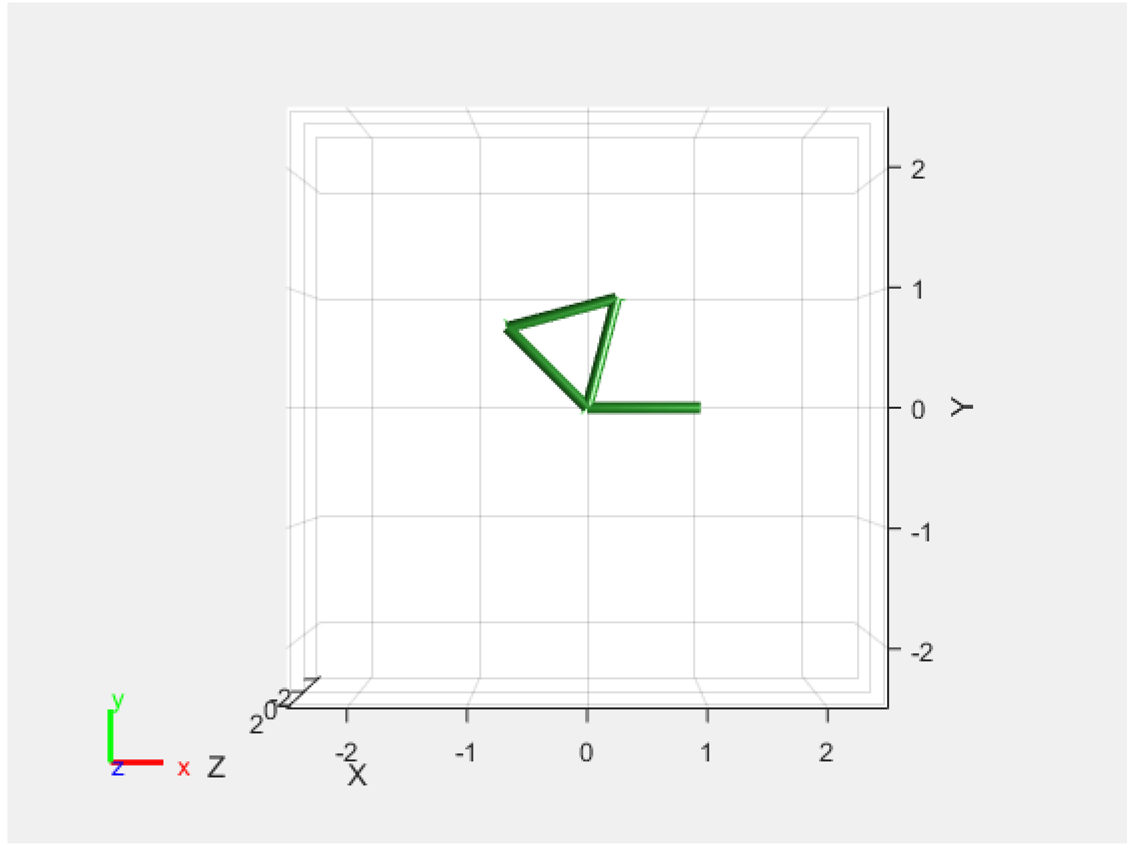
```
cFix.Weights = [1 0];
[qConst,solInfo] = gikSolverWithFixedJointConstraint([theta 0.1 0],cFix);
```

Visualize the robot to see how the fixed constraint joint acts on the robot frame. If the first joint rotates, the solver tries to keep the intermediate frames of the fixed joint constraint coincident, acting as a fixed joint.

```
figure(Name="Fixed Joint Constraint")
show(rbt,qConst,Collisions="on")
```

```
ans = 
  Axes (Primary) with properties:

             XLim: [-2.5000 2.5000]
             YLim: [-2.5000 2.5000]
           XScale: 'linear'
           YScale: 'linear'
    GridLineStyle: '-'
         Position: [0.1300 0.1100 0.7750 0.8150]
            Units: 'normalized'

  Show all properties
```

```
view([0 0 pi])
```



Version History

Introduced in R2022a

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

Objects
`generalizedInverseKinematics` | `constraintAiming` | `constraintCartesianBounds` | `constraintJointBounds` | `constraintOrientationTarget` | `constraintPoseTarget` | `constraintPositionTarget` | `constraintRevoluteJoint` | `constraintFixedJoint` | `constraintDistanceBounds`

Topics
"Solve Inverse Kinematics for Closed Loop Linkages"

constraintRevoluteJoint

Revolute joint constraint between bodies

Description

The constraintRevoluteJoint object describes a closed-loop revolute joint constraint between a successor and predecessor body on the same rigidBodyTree. The constraint is satisfied when the *Z*-axes of the body intermediate frames align and their frame origins coincide. When satisfied, this constraint allows rotation along the *Z*-axes of the intermediate frames.

Creation

Syntax

```
revConst = constraintRevoluteJoint(successorbody,predecessorbody)
revConst = constraintRevoluteJoint( ___ ,Name=Value)
```

Description

revConst = constraintRevoluteJoint(successorbody,predecessorbody) returns a revolute joint constraint object, revConst, that represents a constraint between the specified successor body successorbody and predecessor body predecessorbody of the joint. The successorbody and predecessor arguments set the SuccessorBody and PredecessorBody properties, respectively.

revConst = constraintRevoluteJoint(___ ,Name=Value) specifies properties using one more name-value pair arguments in addition to all input arguments from the previous syntax.

Properties

SuccessorBody — Name of successor body of joint
string scalar | character vector

Name of the successor body frame, specified as a string scalar or character vector. When using this constraint with the generalizedInverseKinematics inverse kinematics (IK) solver, the name must match a body specified in the RigidBodyTree of the generalizedInverseKinematics object.

PredecessorBody — Name of predecessor body of joint
string scalar | character vector

Name of the predecessor body frame, specified as a string scalar or character vector. When using this constraint with the generalizedInverseKinematics inverse kinematics (IK) solver, the name must match a body specified in the RigidBodyTree of the generalizedInverseKinematics object.

`SuccessorTransform` — Fixed transform of joint constraint with respect to successor body frame
`eye(4)` (default) | 4-by-4 matrix

Fixed transform of the joint constraint with respect to the successor body frame, specified as 4-by-4 matrix.

Example: `[1 0 0 1; 0 1 0 1; 0 0 1 1; 0 0 0 1]`

`PredecessorTransform` — Fixed transform of joint constraint with respect to predecessor body frame
`eye(4)` (default) | 4-by-4 matrix

Fixed transform of the joint constraint with respect to the predecessor body frame, specified as 4-by-4 matrix.

Example: `[1 0 0 1; 0 1 0 1; 0 0 1 1; 0 0 0 1]`

`PositionTolerance` — Position tolerance of joint constraint
`0` (default) | nonnegative scalar

Position tolerance of the joint constraint in radians, specified as a non-negative scalar.

`JointPositionLimits` — Joint position limits
`[-3.1416 3.1416]` (default) | two-element row vector

Position limits of the joint constraint, in radians, specified as a two-element row vector in the form `[minimum maximum]`.

Example: `[-1.5708 1.5708]`

`OrientationTolerance` — Orientation tolerance of joint constraint
`0` (default) | nonnegative scalar

Orientation tolerance of the joint constraint in radians, specified as a nonnegative scalar.

`Weights` — Weights of the constraint
`[1 1 1]` (default) | three-element vector

Weights of the constraint, specified as a three-element vector. The elements of the vector correspond to the weights for the `PositionTolerance`, `OrientationTolerance`, and `JointPositionLimits` properties, respectively. These weights are used with the weights of all the constraints specified in the `generalizedInverseKinematics` solver, and can be used to specify the relative importance of a constraint violation to the solver.

Example: `[0 1 4]`

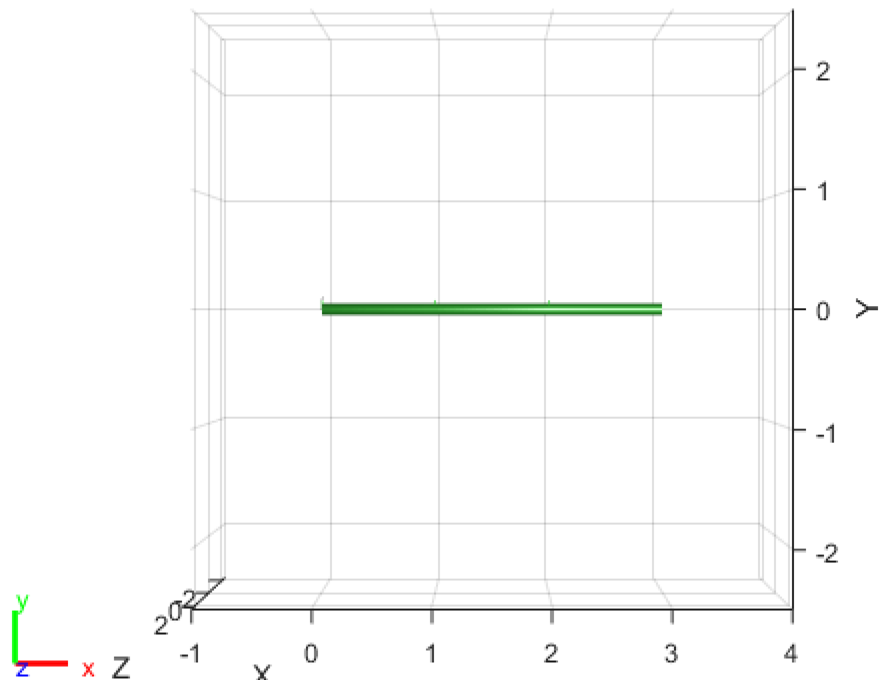
Examples

Create Loop-Closure Joint Constraints

Create a revolute, prismatic, and fixed joint constraints for a simple rigid body tree.

Use the `exampleHelperFourBarLinkageTree` helper function to create a simple robot model to demonstrate the closed-loop constraints.

```
rbt = exampleHelperFourBarLinkageTree;
show(rbt,Collisions="on");
view([0 0 pi])
xlim([-1 4])
```



Revolute Joint Constraint

To demonstrate a revolute joint constraint, create a four-bar linkage by connecting the end of the last link, `link3`, and the first link, `link0`.

Create a generalized inverse kinematics solver with a revolute joint constraint and a joint bounds constraint.

```
gikSolverWithRevoluteJointConstraint = generalizedInverseKinematics(RigidBodyTree=rbt, ...
    ConstraintInputs={'revolute','jointbounds'});
```

To ensure repeatable IK solutions, disable random restarts.

```
gikSolverWithRevoluteJointConstraint.SolverParameters.AllowRandomRestart = false;
theta = pi/2+pi/4;
```

Fix the first joint by setting `theta` as both the minimum and maximum bound.

```
activeJointConstraint = constraintJointBounds(rbt);
activeJointConstraint.Weights = [1 0 0];
activeJointConstraint.Bounds(1,:) = [theta theta];
```

Create a revolute joint constraint with successor and predecessor bodies set to the last link `link3` and the first link `link0`, respectively. Specify predecessor and successor transforms that create

intermediate frames 1 meter away, in the *X*-axis, from their respective body. Once defined, these intermediate frames move such that their frame origins coincide when their *Z*-axes align.

```
cRev = constraintRevoluteJoint("link3","link0", ...
    PredecessorTransform=trvec2tform([1 0 0]), ...
    SuccessorTransform=trvec2tform([1 0 0]));
```

Provide `[theta 0 0]` as an initial guess to the solver, along with the constraints.

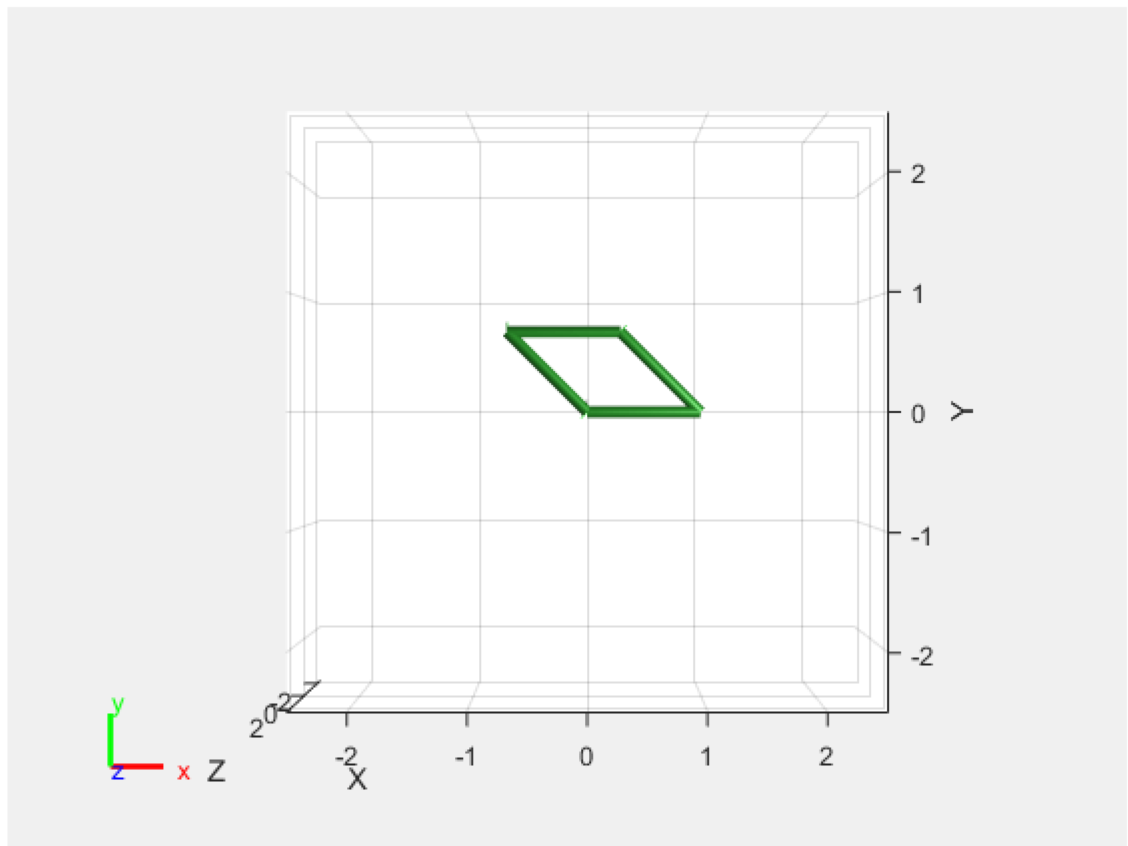
```
qConst = gikSolverWithRevoluteJointConstraint([theta 0 0],cRev,activeJointConstraint);
```

Visualize the robot to see the robot acting as a four-bar linkage. If the first joint rotates, the solver tries to keep the intermediate frames of the revolute joint constraint coincident, acting as a joint and resulting in four-bar motion.

```
figure(Name="Revolute Joint Constraint")
show(rbt,qConst,Collisions="on")
```

```
ans =
  Axes (Primary) with properties:

             XLim: [-2.5000 2.5000]
             YLim: [-2.5000 2.5000]
           XScale: 'linear'
           YScale: 'linear'
    GridLineStyle: '-'
         Position: [0.1300 0.1100 0.7750 0.8150]
            Units: 'normalized'

  Show all properties
```

```
view([0 0 pi])
```



Prismatic Joint Constraint

Use a prismatic joint constraint to create a slider-crank. Create a new solver with a prismatic joint constraint and a joint bounds constraint.

```
gikSolverWithPrismaticJointConstraint = generalizedInverseKinematics(RigidBodyTree=rbt, ...
    ConstraintInputs={'prismatic','jointbounds'});
gikSolverWithPrismaticJointConstraint.SolverParameters.AllowRandomRestart=false;
```

Create the prismatic joint constraint with `link3` and `link0` as the successor and predecessor bodies, respectively, and set the predecessor transform such that the predecessor intermediate frame is `1` meter away on the *X*-axis and rotated `pi/2` in the *Y*-axis from the predecessor body frame.

```
cPris=constraintPrismaticJoint("link3","link0",PredecessorTransform=trvec2tform([1 0 0])*eul2tfo
```

Provide `[theta 0 0]` as an initial guess to the solver along with the constraints.

```
qConst = gikSolverWithPrismaticJointConstraint([theta 0 0],cPris,activeJointConstraint);
```

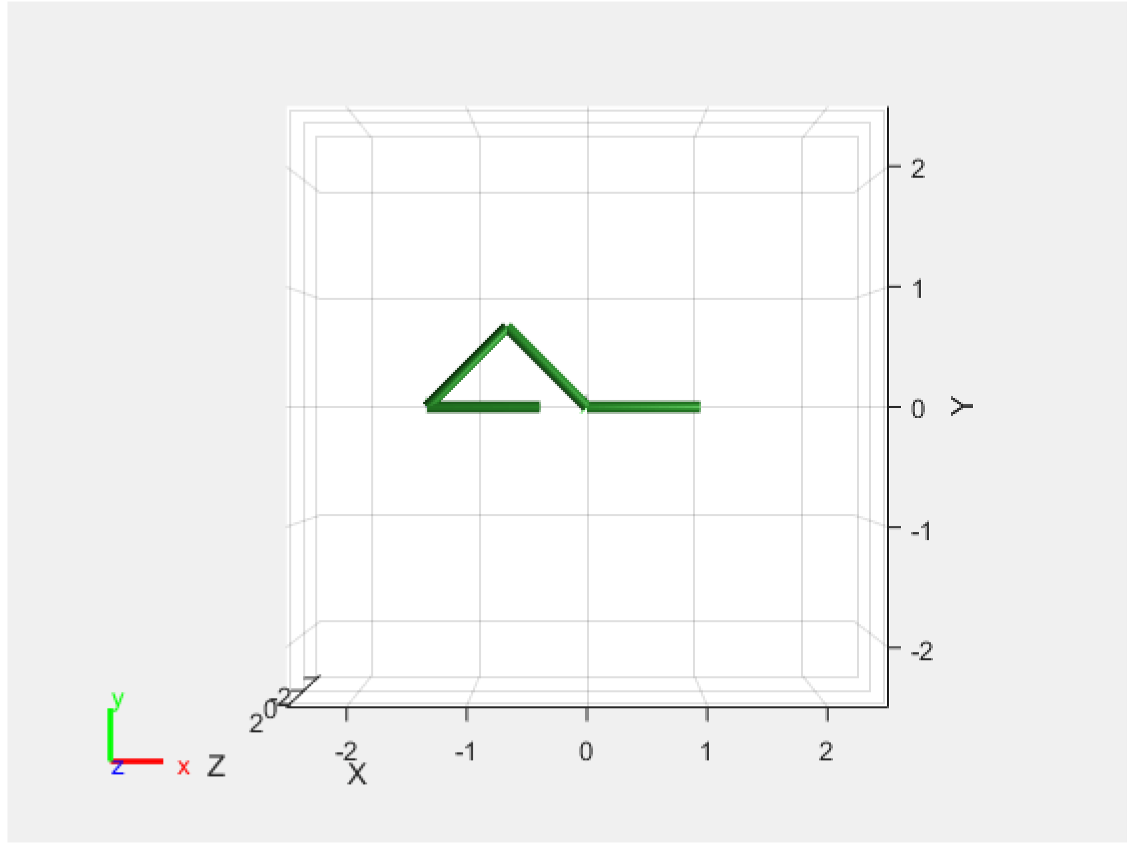
Visualize the robot to see the robot acting as a slider-crank. If the first joint rotates, the solver tries to keep the intermediate frames of the prismatic joint constraint coincident, acting as a joint and resulting in slider-crank motion.

```
figure(Name="Prismatic Joint Constraint")
show(rbt,qConst,Collisions="on")
```

```
ans = 
  Axes (Primary) with properties:
```

```
            XLim: [-2.5000 2.5000]
            YLim: [-2.5000 2.5000]
          XScale: 'linear'
          YScale: 'linear'
   GridLineStyle: '-'
        Position: [0.1300 0.1100 0.7750 0.8150]
           Units: 'normalized'

  Show all properties
```

```
view([0 0 pi])
```



Fixed Joint Constraint

To demonstrate a fixed joint constraint, create a triangle with the links that is preserved when the first joint moves. Create a new solver with a fixed joint constraint.

```
gikSolverWithFixedJointConstraint = generalizedInverseKinematics(RigidBodyTree=rbt, ...
    ConstraintInputs={'fixed'});
```

Create the fixed joint constraint with `link3` and `link0` as the successor and predecessor bodies, respectively, and set the successor transform such that the predecessor intermediate frame is `1` meter away on the *X*-axis from the predecessor body frame.

```
cFix = constraintFixedJoint("link3","link1",SuccessorTransform=trvec2tform([1 0 0]));
```

Set the weight of the orientation constraint of the fixed joint constraint to `0`.

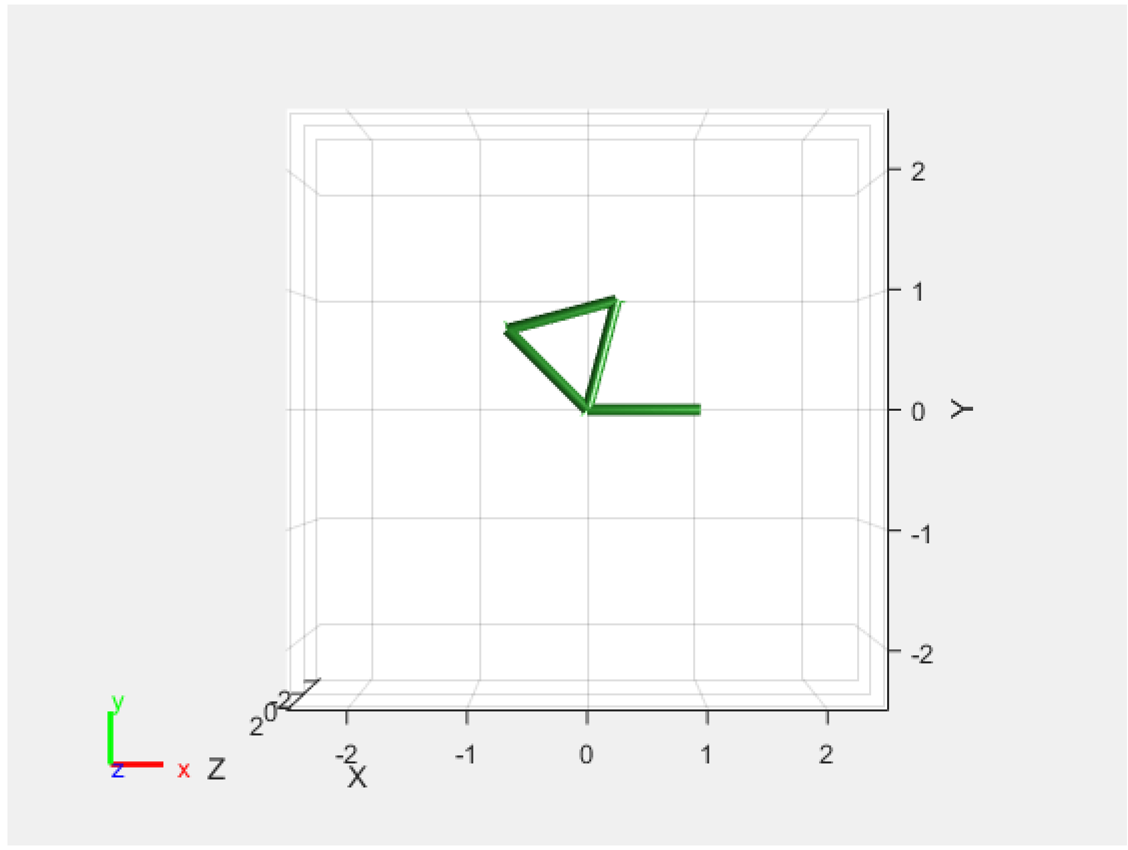
```
cFix.Weights = [1 0];
[qConst,solInfo] = gikSolverWithFixedJointConstraint([theta 0.1 0],cFix);
```

Visualize the robot to see how the fixed constraint joint acts on the robot frame. If the first joint rotates, the solver tries to keep the intermediate frames of the fixed joint constraint coincident, acting as a fixed joint.

```
figure(Name="Fixed Joint Constraint")
show(rbt,qConst,Collisions="on")
```

```
ans = 
  Axes (Primary) with properties:

             XLim: [-2.5000 2.5000]
             YLim: [-2.5000 2.5000]
           XScale: 'linear'
           YScale: 'linear'
    GridLineStyle: '-'
         Position: [0.1300 0.1100 0.7750 0.8150]
            Units: 'normalized'

  Show all properties
```

```
view([0 0 pi])
```



Version History

Introduced in R2022a

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

Objects

`generalizedInverseKinematics` | `constraintAiming` | `constraintCartesianBounds` | `constraintJointBounds` | `constraintOrientationTarget` | `constraintPoseTarget` | `constraintPositionTarget` | `constraintPrismaticJoint` | `constraintFixedJoint` | `constraintDistanceBounds`

Topics

"Solve Inverse Kinematics for Closed Loop Linkages"

controllerPurePursuit

Create controller to follow set of waypoints

Description

The `controllerPurePursuit` System object™ creates a controller object used to make a differential-drive vehicle follow a set of waypoints. The object computes the linear and angular velocities for the vehicle given the current pose. Successive calls to the object with updated poses provide updated velocity commands for the vehicle. Use the `MaxAngularVelocity` and `DesiredLinearVelocity` properties to update the velocities based on the vehicle's performance.

The `LookaheadDistance` property computes a look-ahead point on the path, which is a local goal for the vehicle. The angular velocity command is computed based on this point. Changing `LookaheadDistance` has a significant impact on the performance of the algorithm. A higher look-ahead distance results in a smoother trajectory for the vehicle, but can cause the vehicle to cut corners along the path. A low look-ahead distance can result in oscillations in tracking the path, causing unstable behavior. For more information on the pure pursuit algorithm, see “Pure Pursuit Controller”.

To compute linear and angular velocity control commands:

1. Create the `controllerPurePursuit` object and set its properties.
2. Call the object with arguments, as if it were a function.

To learn more about how System objects work, see What Are System Objects?

Creation

Syntax

`controller = controllerPurePursuit`

`controller = controllerPurePursuit(Name,Value)`

Description

`controller = controllerPurePursuit` creates a pure pursuit object that uses the pure pursuit algorithm to compute the linear and angular velocity inputs for a differential drive vehicle.

`controller = controllerPurePursuit(Name,Value)` creates a pure pursuit object with additional options specified by one or more `Name,Value` pairs. Name is the property name and Value is the corresponding value. Name must appear inside single quotes (`' '`). You can specify several name-value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`. Properties not specified retain their default values.

Example: `controller = controllerPurePursuit('DesiredLinearVelocity', 0.5)`

Properties

`DesiredLinearVelocity` — Desired constant linear velocity

`0.1` (default) | scalar in meters per second

Desired constant linear velocity, specified as a scalar in meters per second. The controller assumes that the vehicle drives at a constant linear velocity and that the computed angular velocity is independent of the linear velocity.

Data Types: `double`

`LookaheadDistance` — Look-ahead distance

`1.0` (default) | scalar in meters

Look-ahead distance, specified as a scalar in meters. The look-ahead distance changes the response of the controller. A vehicle with a higher look-ahead distance produces smooth paths but takes larger turns at corners. A vehicle with a smaller look-ahead distance follows the path closely and takes sharp turns, but potentially creating oscillations in the path.

Data Types: `double`

`MaxAngularVelocity` — Maximum angular velocity

`1.0` (default) | scalar in radians per second

Maximum angular velocity, specified a scalar in radians per second. The controller saturates the absolute angular velocity output at the given value.

Data Types: `double`

`Waypoints` — Waypoints

[ ] (default) | *n*-by-2 array

Waypoints, specified as an *n*-by-2 array of `[x y]` pairs, where *n* is the number of waypoints. You can generate the waypoints from the `mobileRobotPRM` class or from another source.

Data Types: `double`

Usage

Syntax

```
[vel,angvel] = controller(pose)
[vel,angvel,lookaheadpoint] = controller(pose)
```

Description

`[vel,angvel] = controller(pose)` processes the vehicle's position and orientation, `pose`, and outputs the linear velocity, `vel`, and angular velocity, `angvel`.

`[vel,angvel,lookaheadpoint] = controller(pose)` returns the look-ahead point, which is a location on the path used to compute the velocity commands. This location on the path is computed using the `LookaheadDistance` property on the `controller` object.

Input Arguments

pose — Position and orientation of vehicle
3-by-1 vector in the form [x y theta]

Position and orientation of vehicle, specified as a 3-by-1 vector in the form [x y theta]. The vehicle pose is an *x* and *y* position with angular orientation θ (in radians) measured from the *x*-axis.

Output Arguments

vel — Linear velocity
scalar in meters per second

Linear velocity, specified as a scalar in meters per second.

Data Types: double

angvel — Angular velocity
scalar in radians per second

Angular velocity, specified as a scalar in radians per second.

Data Types: double

lookaheadpoint — Look-ahead point on path
[x y] vector

Look-ahead point on the path, returned as an [x y] vector. This value is calculated based on the LookaheadDistance property.

Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named obj, use this syntax:

```
release(obj)
```

Specific to controllerPurePursuit

info	Characteristic information about controllerPurePursuit object

Common to All System Objects

step	Run System object algorithm
release	Release resources and allow changes to System object property values and input characteristics
reset	Reset internal states of System object

Examples

Get Additional Pure Pursuit Object Information

Use the info method to get more information about a controllerPurePursuit object. The info function returns two fields, RobotPose and LookaheadPoint, which correspond to the current

position and orientation of the robot and the point on the path used to compute outputs from the last call of the object.

Create a `controllerPurePursuit` object.

```
pp = controllerPurePursuit;
```

Assign waypoints.

```
pp.Waypoints = [0 0;1 1];
```

Compute control commands using the `pp` object with the initial pose `[x y theta]` given as the input.

```
[v,w] = pp([0 0 0]);
```

Get additional information.

```
s = info(pp)
```

```
s = struct with fields:
         RobotPose: [0 0 0]
    LookaheadPoint: [0.7071 0.7071]
```

Version History

Introduced in R2019b

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

Use in a MATLAB Function block is not supported.

For additional information about code generation for System objects, see “System Objects in MATLAB Code Generation” (MATLAB Coder)

See Also

`binaryOccupancyMap` | `occupancyMap` | `mobileRobotPRM`

Topics

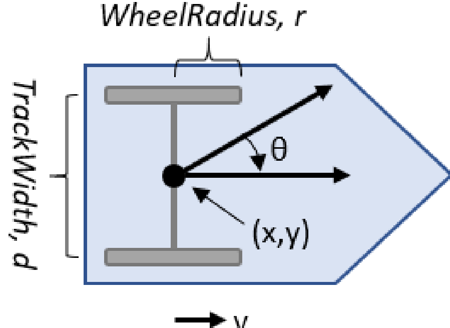
“Path Following for a Differential Drive Robot”
“Pure Pursuit Controller”

differentialDriveKinematics

Differential-drive vehicle model

Description

`differentialDriveKinematics` creates a differential-drive vehicle model to simulate simplified vehicle dynamics. This model approximates a vehicle with a single fixed axle and wheels separated by a specified track width. The wheels can be driven independently. Vehicle speed and heading is defined from the axle center. The state of the vehicle is defined as a three-element vector, *[x y theta]*, with a global *xy*-position, specified in meters, and a vehicle heading, *theta*, specified in radians. To compute the time derivative states for the model, use the `derivative` function with input commands and the current robot state.



Creation

Syntax

`kinematicModel = differentialDriveKinematics`

`kinematicModel = differentialDriveKinematics(Name,Value)`

Description

`kinematicModel = differentialDriveKinematics` creates a differential drive kinematic model object with default property values.

`kinematicModel = differentialDriveKinematics(Name,Value)` sets properties on the object to the specified value. You can specify multiple properties in any order.

Properties

`WheelRadius` — Wheel radius of vehicle
`0.05` (default) | positive numeric scalar

The wheel radius of the vehicle, specified in meters.

`WheelSpeedRange` — Range of vehicle wheel speeds
`[-Inf Inf]` (default) | two-element vector

The vehicle speed range is a two-element vector that provides the minimum and maximum vehicle speeds, [*MinSpeed MaxSpeed*], specified in meters per second.

`TrackWidth` — Distance between wheels on axle
`0.2` (default) | positive numeric scalar

The vehicle track width refers to the distance between the wheels, or the axle length, specified in meters.

`VehicleInputs` — Type of motion inputs for vehicle
`"WheelSpeeds"` (default) | character vector | string scalar

The `VehicleInputs` property specifies the format of the model input commands when using the `derivative` function. Options are specified as one of the following strings:

- `"WheelSpeeds"` — Angular speeds for each of the wheels, specified in radians per second.
- `"VehicleSpeedHeadingRate"` — Vehicle speed and heading angular velocity, specified in meters per second and radians per second respectively.

Object Functions

derivative   Time derivative of vehicle state

Examples

Plot Path of Differential-Drive Kinematic Robot

Create a Robot

Define a robot and set the initial starting position and orientation.

```
kinematicModel = differentialDriveKinematics;
initialState = [0 0 0];
```

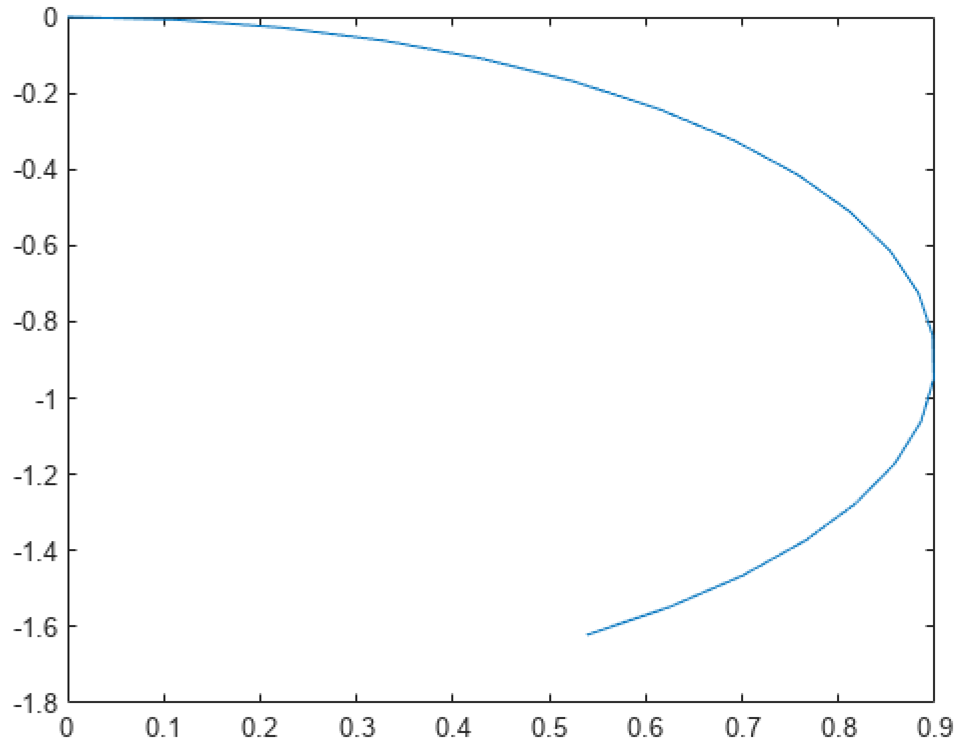
Simulate Robot Motion

Set the timespan of the simulation to `1` s with `0.05` s time steps and the input commands to `50` rad/s for the left wheel and `40` rad/s for the right wheel to result in a right turn. Simulate the motion of the robot by using the `ode45` solver on the `derivative` function.

```
tspan = 0:0.05:1;
inputs = [50 40]; %Left wheel is spinning faster
[t,y] = ode45(@(t,y)derivative(kinematicModel,y,inputs),tspan,initialState);
```

Plot Path

```
figure
plot(y(:,1),y(:,2))
```



Version History

Introduced in R2019b

References

[1] Lynch, Kevin M., and Frank C. Park. *Modern Robotics: Mechanics, Planning, and Control* 1st ed. Cambridge, MA: Cambridge University Press, 2017.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

Classes

`ackermannKinematics` | `bicycleKinematics` | `unicycleKinematics`

Blocks

Differential Drive Kinematic Model

Functions

`derivative`

Topics

“Path Following for a Differential Drive Robot”
“Simulate Different Kinematic Models for Mobile Robots”
“Mobile Robot Kinematics Equations”

extendedObjectMesh

Mesh representation of extended object

Description

The `extendedObjectMesh` represents the 3-D geometry of an object. The 3-D geometry is represented by faces and vertices. Use these object meshes to specify the geometry of an `robotPlatform` for simulating lidar sensor data using `robotLidarPointCloudGenerator`.

Creation

Syntax

```
mesh = extendedObjectMesh('cuboid')
mesh = extendedObjectMesh('cylinder')
mesh = extendedObjectMesh('cylinder',n)
mesh = extendedObjectMesh('sphere')
mesh = extendedObjectMesh('sphere',n)
mesh = extendedObjectMesh(vertices,faces)
```

Description

`mesh = extendedObjectMesh('cuboid')` returns an `extendedObjectMesh` object, that defines a cuboid with unit dimensions. The origin of the cuboid is located at its geometric center.

`mesh = extendedObjectMesh('cylinder')` returns a hollow cylinder mesh with unit dimensions. The cylinder mesh has 20 equally spaced vertices around its circumference. The origin of the cylinder is located at its geometric center. The height is aligned with the *z*-axis.

`mesh = extendedObjectMesh('cylinder',n)` returns a cylinder mesh with `n` equally spaced vertices around its circumference.

`mesh = extendedObjectMesh('sphere')` returns a sphere mesh with unit dimensions. The sphere mesh has 119 vertices and 180 faces. The origin of the sphere is located at its center.

`mesh = extendedObjectMesh('sphere',n)` additionally allows you to specify the resolution, `n`, of the spherical mesh. The sphere mesh has $(n + 1)^2 - 2$ vertices and $2n(n - 1)$ faces.

`mesh = extendedObjectMesh(vertices,faces)` returns a mesh from faces and vertices. `vertices` and `faces` set the `Vertices` and `Faces` properties respectively.

Properties

`Vertices` — Vertices of defined object
N-by-3 matrix of real scalar

Vertices of the defined object, specified as an *N*-by-3 matrix of real scalars. *N* is the number of vertices. The first, second, and third element of each row represents the *x*-, *y*-, and *z*-position of each vertex, respectively.

Faces — Faces of defined object
M-by-3 matrix of positive integer

Faces of the defined object, specified as a *M*-by-3 array of positive integers. *M* is the number of faces. The three elements in each row are the vertex IDs of the three vertices forming the triangle face. The ID of the vertex is its corresponding row number specified in the `Vertices` property.

Object Functions

Use the object functions to develop new meshes.

applyTransform	Apply forward transformation to mesh vertices
join	Join two object meshes
rotate	Rotate mesh about coordinate axes
scale	Scale mesh in each dimension
scaleToFit	Auto-scale object mesh to match specified cuboid dimensions
show	Display the mesh as a patch on the current axes
translate	Translate mesh along coordinate axes

Examples

Create and Translate Cuboid Mesh

Create an `extendedObjectMesh` object and translate the object.

Construct a cuboid mesh.

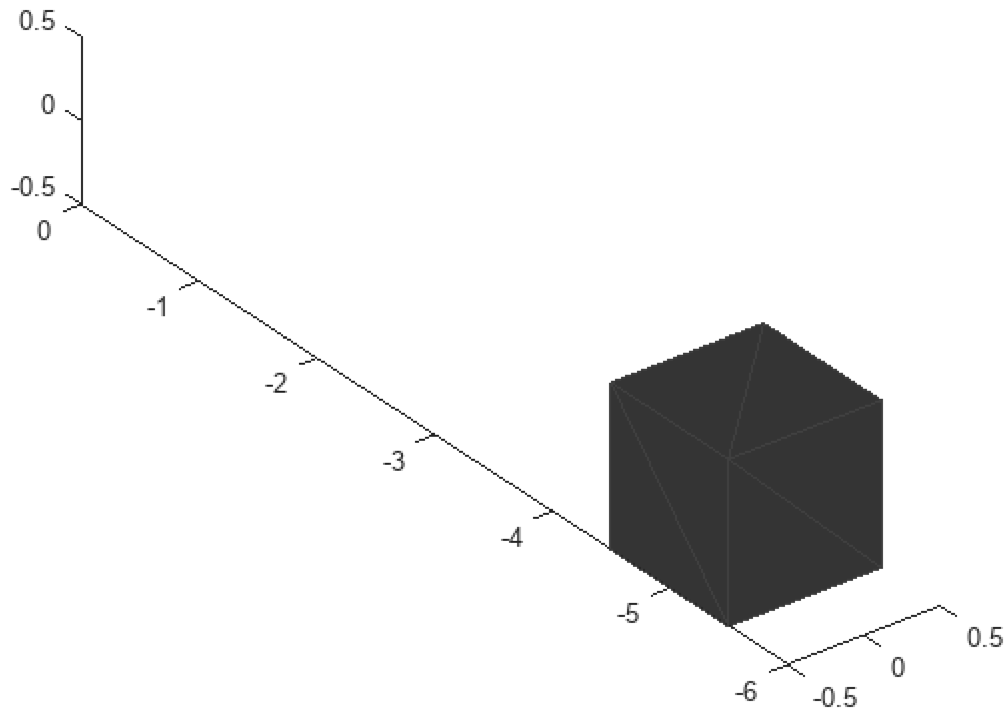
```
mesh = extendedObjectMesh('cuboid');
```

Translate the mesh by 5 units along the negative *y* axis.

```
mesh = translate(mesh,[0 -5 0]);
```

Visualize the mesh.

```
ax = show(mesh);
ax.YLim = [-6 0];
```



Create and Visualize Cylinder Mesh

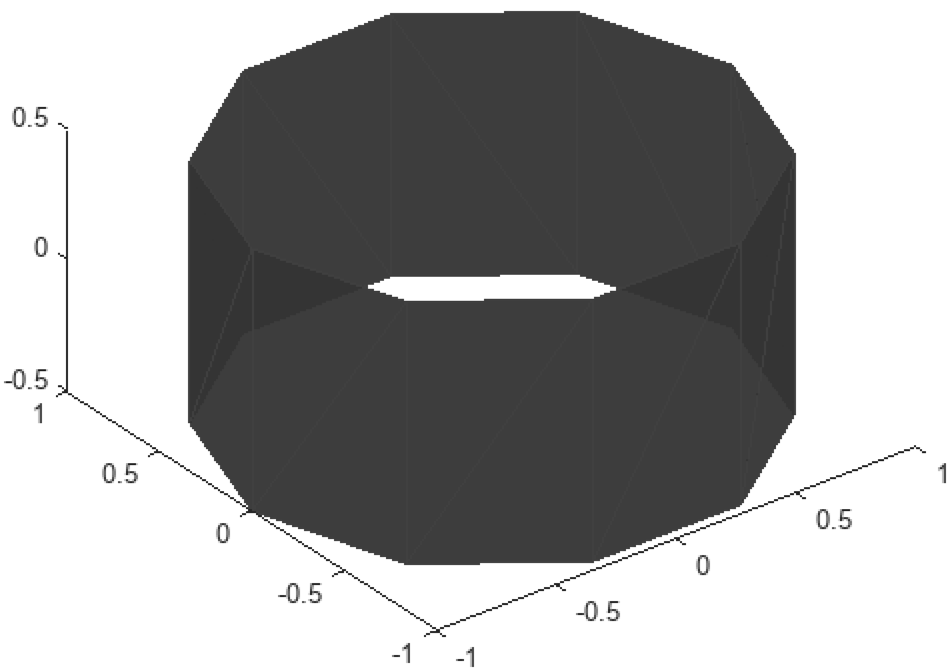
Create an `extendedObjectMesh` object and visualize the object.

Construct a cylinder mesh.

```
mesh = extendedObjectMesh('cylinder');
```

Visualize the mesh.

```
ax = show(mesh);
```



Create and Auto-Scale Sphere Mesh

Create an `extendedObjectMesh` object and auto-scale the object to the required dimensions.

Construct a sphere mesh of unit dimensions.

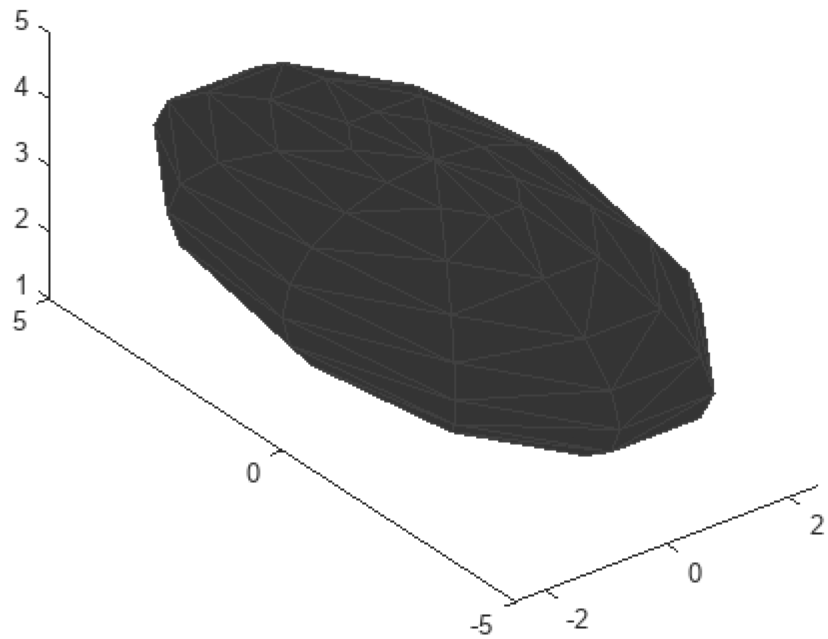
```
sph = extendedObjectMesh('sphere');
```

Auto-scale the mesh to the dimensions in `dims`.

```
dims = struct('Length',5,'Width',10,'Height',3,'OriginOffset',[0 0 -3]);
sph = scaleToFit(sph,dims);
```

Visualize the mesh.

```
show(sph);
```



Version History

Introduced in R2022a

See Also

Objects
robotPlatform | robotLidarPointCloudGenerator

Functions
applyTransform | join | rotate | scale | scaleToFit | show | translate

generalizedInverseKinematics

Create multiconstraint inverse kinematics solver

Description

The `generalizedInverseKinematics` System object uses a set of kinematic constraints to compute a joint configuration for the rigid body tree model specified by a `rigidBodyTree` object. The `generalizedInverseKinematics` object uses a nonlinear solver to satisfy the constraints or reach the best approximation.

Specify the constraint types, `ConstraintInputs`, before calling the object. To change constraint inputs after calling the object, call `release(gik)`.

Specify the constraint inputs as constraint objects and call `generalizedInverseKinematics` with these objects passed into it. To create constraint objects, use the following objects:

- `constraintAiming`
- `constraintCartesianBounds`
- `constraintJointBounds`
- `constraintOrientationTarget`
- `constraintPoseTarget`
- `constraintPositionTarget`
- `constraintDistanceBounds`
- `constraintFixedJoint`
- `constraintPrismaticJoint`
- `constraintRevoluteJoint`

If your only constraint is the end-effector position and orientation, consider using `inverseKinematics` as your solver instead.

For closed-form analytical inverse kinematics solutions, see `analyticalInverseKinematics`.

To solve the generalized inverse kinematics constraints:

1. Create the `generalizedInverseKinematics` object and set its properties.
2. Call the object with arguments, as if it were a function.

To learn more about how System objects work, see What Are System Objects?

Creation

Syntax

```
gik = generalizedInverseKinematics
```

```
gik = generalizedInverseKinematics('RigidBodyTree',rigidbodytree,'
ConstraintInputs',inputTypes)
gik = generalizedInverseKinematics(Name,Value)
```

Description

`gik = generalizedInverseKinematics` returns a generalized inverse kinematics solver with no rigid body tree model specified. Specify a `rigidBodyTree` model and the `ConstraintInputs` property before using this solver.

`gik = generalizedInverseKinematics('RigidBodyTree',rigidbodytree,'ConstraintInputs',inputTypes)` returns a generalized inverse kinematics solver with the rigid body tree model and the expected constraint inputs specified.

`gik = generalizedInverseKinematics(Name,Value)` returns a generalized inverse kinematics solver with each specified property name set to the specified value by one or more `Name,Value` pair arguments. `Name` must appear inside single quotes (`''`). You can specify several name-value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see System Design in MATLAB Using System Objects.

`NumConstraints` — Number of constraint inputs
scalar

This property is read-only.

Number of constraint inputs, specified as a scalar. The value of this property is the number of constraint types specified in the `ConstraintInputs` property.

`ConstraintInputs` — Constraint input types
cell array of character vectors

Constraint input types, specified as a cell array of character vectors. The possible constraint input types with their associated constraint objects are:

- `'orientation'` — `constraintOrientationTarget`
- `'position'` — `constraintPositionTarget`
- `'pose'` — `constraintPoseTarget`
- `'aiming'` — `constraintAiming`
- `'cartesian'` — `constraintCartesianBounds`
- `'jointbounds'` — `constraintJointBounds`
- `'distance'` — `constraintDistanceBounds`

There are also closed-loop joint constraints that constrain two rigid bodies such that the constrained motion is similar to that of an additional joint between the two bodies. Their constraint input types with their associated constraint objects are:

- `'revolutejoint'` — `constraintRevoluteJoint`
- `'prismaticjoint'` — `constraintPrismaticJoint`
- `'fixedjoint'` — `constraintFixedJoint`

Use the constraint objects to specify the required parameters and pass those object types into the object when you call it. For example:

Create the generalized inverse kinematics solver object. Specify the `RigidBodyTree` and `ConstraintInputs` properties.

```
gik = generalizedInverseKinematics(...
                     'RigidBodyTree',rigidbodytree,
                     'ConstraintInputs',{'position','aiming'});
```

Create the corresponding constraint objects.

```
positionTgt = constraintPositionTarget('left_palm');
aimConst = constraintAiming('right_palm');
```

Pass the constraint objects into the solver object with an initial guess.

```
configSol = gik(initialGuess,positionTgt,aimConst);
```

RigidBodyTree — Rigid body tree model

`rigidBodyTree` object

Rigid body tree model, specified as a `rigidBodyTree` object. Define this property before using the solver. If you modify your rigid body tree model, reassign the rigid body tree to this property. For example:

Create IK solver and specify the rigid body tree.

```
gik = generalizedInverseKinematics(...
                     'RigidBodyTree',rigidbodytree,
                     'ConstraintInputs',{'position','aiming'});
```

Modify the rigid body tree model.

```
addBody(rigidbodytree,rigidBody('body1'),'base')
```

Reassign the rigid body tree to the IK solver. If the solver or the `step` function is called before modifying the rigid body tree model, use `release` to allow the property to be changed.

```
gik.RigidBodyTree = rigidbodytree;
```

SolverAlgorithm — Algorithm for solving inverse kinematics

`'BFGSGradientProjection'` (default) | `'LevenbergMarquardt'`

Algorithm for solving inverse kinematics, specified as either `'BFGSGradientProjection'` or `'LevenbergMarquardt'`. For details of each algorithm, see “Inverse Kinematics Algorithms”.

SolverParameters — Parameters associated with algorithm

structure

Parameters associated with the specified algorithm, specified as a structure. The fields in the structure are specific to the algorithm. See “Solver Parameters”.

Usage

Syntax

`[configSol,solInfo] = gik(initialguess,constraintObj,...,constraintObjN)`

Description

`[configSol,solInfo] = gik(initialguess,constraintObj,...,constraintObjN)` finds a joint configuration, `configSol`, based on the initial guess and a comma-separated list of constraint description objects. The number of constraint descriptions depends on the `ConstraintInputs` property.

Input Arguments

`initialguess` — Initial guess of robot configuration
structure array | vector

Initial guess of robot configuration, specified as a structure array or vector. The value of `initialguess` depends on the `DataFormat` property of the object specified in the `RigidBodyTree` property specified in `gik`.

Use this initial guess to guide the solver to the target robot configuration. However, the solution is not guaranteed to be close to this initial guess.

`constraintObj,...,constraintObjN` — Constraint descriptions
constraint objects

Constraint descriptions defined by the `ConstraintInputs` property of `gik`, specified as one or more of these constraint objects:

- `constraintAiming`
- `constraintCartesianBounds`
- `constraintJointBounds`
- `constraintOrientationTarget`
- `constraintPoseTarget`
- `constraintPositionTarget`

Output Arguments

`configSol` — Robot configuration solution
structure array | vector

Robot configuration solution, returned as a structure array or vector, depends on the `DataFormat` property of the object specified in the `RigidBodyTree` property specified in `gik`.

The structure array contains these fields:

- `JointName` — Character vector for the name of the joint specified in the `RigidBodyTree` robot model

- `JointPosition` — Position of the corresponding joint

The vector output is an array of the joint positions that would be given in `JointPosition` for a structure output.

This joint configuration is the computed solution that achieves the target end-effector pose within the solution tolerance.

Note For revolute joints, if the joint limits exceed a range of `2*pi`, where joint position wrapping occurs, then the returned joint position is the one closest to the joint's lower bound.

`solInfo` — Solution information
structure

Solution information, returned as a structure containing these fields:

- `Iterations` — Number of iterations run by the solver.
- `NumRandomRestarts` — Number of random restarts because the solver got stuck in a local minimum.
- `ConstraintViolation` — Information about the constraint, returned as a structure array. Each structure in the array has these fields:
  - `Type`: Type of the corresponding constraint input, as specified in the `ConstraintInputs` property.
  - `Violation`: Vector of constraint violations for the corresponding constraint type. `0` indicates that the constraint is satisfied.
- `ExitFlag` — Code that gives more details on the solver execution and what caused it to return. For the exit flags of each solver type, see “Exit Flags”.
- `Status` — Character vector describing whether the solution is within the tolerances defined by each constraint (`'success'`). If the solution is outside the tolerance, the best possible solution that the solver could find is given (`'best available'`).

Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

Common to All System Objects

step	Run System object algorithm
release	Release resources and allow changes to System object property values and input characteristics
reset	Reset internal states of System object

Examples

Solve Generalized Inverse Kinematics for a Set of Constraints

Create a generalized inverse kinematics solver that holds a robotic arm at a specific location and points toward the robot base. Create the constraint objects to pass the necessary constraint parameters into the solver.

Load predefined KUKA LBR robot model, which is specified as a `rigidBodyTree` object.

```
load exampleRobots.mat lbr
```

Create the System object™ for solving generalized inverse kinematics.

```
gik = generalizedInverseKinematics;
```

Configure the System object to use the KUKA LBR robot.

```
gik.RigidBodyTree = lbr;
```

Tell the solver to expect a `constraintAiming` and `constraintPositionTarget` object as the constraint inputs.

```
gik.ConstraintInputs = {'position','aiming'};
```

Create the two constraint objects.

1. The origin of the body named `tool0` is located at `[0.0 0.5 0.5]` relative to the robot's base frame.
2. The *z*-axis of the body named `tool0` points toward the origin of the robot's base frame.

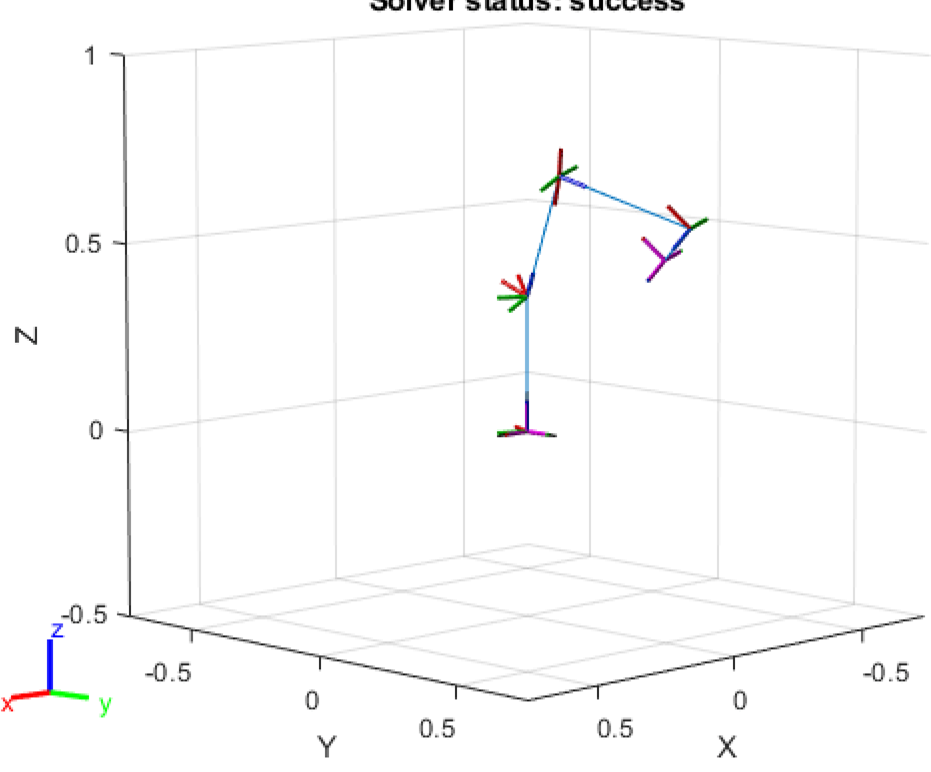
```
posTgt = constraintPositionTarget('tool0');
posTgt.TargetPosition = [0.0 0.5 0.5];

aimCon = constraintAiming('tool0');
aimCon.TargetPoint = [0.0 0.0 0.0];
```

Find a configuration that satisfies the constraints. You must pass the constraint objects into the System object in the order in which they were specified in the `ConstraintInputs` property. Specify an initial guess at the robot configuration.

```
q0 = homeConfiguration(lbr); % Initial guess for solver
[q,solutionInfo] = gik(q0,posTgt,aimCon);
```

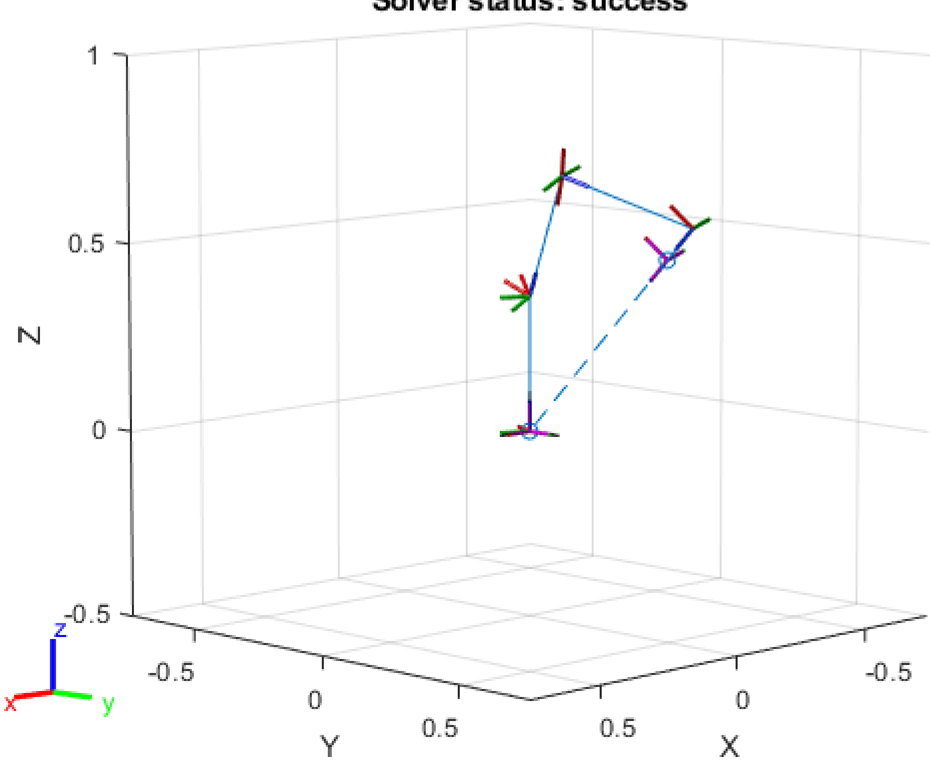
Visualize the configuration returned by the solver.

```
show(lbr,q);
title(['Solver status: ' solutionInfo.Status])
axis([-0.75 0.75 -0.75 0.75 -0.5 1])
```



Plot a line segment from the target position to the origin of the base. The origin of the `tool0` frame coincides with one end of the segment, and its `z`-axis is aligned with the segment.

```
hold on
plot3([0.0 0.0],[0.5 0.0],[0.5 0.0],'--o')
hold off
```



Version History

Introduced in R2017a

R2019b: generalizedInverseKinematics was renamed

Behavior change in future release

The `generalizedInverseKienematics` object was renamed from `robotics.GeneralizedInverseKinematics`. Use `generalizedInverseKienematics` for all object creation.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

When using code generation, you must specify the `ConstraintInputs` and `RigidBodyTree` properties on construction of the object. For example:

```
gik = generalizedInverseKinematics(...
    'ConstraintInputs',{'pose','position'},...
    'RigidBodyTree',rigidbodytree);
```

You also cannot change the `SolverAlgorithm` property after creation. To specify the solver algorithm on creation, use:

```
gik = generalizedInverseKinematics(...
    'ConstraintInputs',{'pose','position'},...
    'RigidBodyTree',rigidbodytree,...
    'SolverAlgorithm','LevenbergMarquardt');
```

See Also

Objects
analyticalInverseKinematics | inverseKinematics | constraintPoseTarget | constraintPositionTarget | constraintAiming | constraintCartesianBounds | constraintJointBounds | constraintOrientationTarget

Topics
“Control PR2 Arm Movements Using ROS Actions and Inverse Kinematics”
“Plan a Reaching Trajectory With Multiple Kinematic Constraints”

gpsSensor

GPS receiver simulation model

Description

The `gpsSensor` System object models data output from a Global Positioning System (GPS) receiver. The object models the position noise as a first order Gauss Markov process, in which the sigma values are specified in the `HorizontalPositionAccuracy` and the `VerticalPositionAccuracy` properties. The object models the velocity noise as Gaussian noise with its sigma value specified in the `VelocityAccuracy` property.

To model a GPS receiver:

1. Create the `gpsSensor` object and set its properties.
2. Call the object with arguments, as if it were a function.

To learn more about how System objects work, see What Are System Objects?

Creation

Syntax

```
GPS = gpsSensor
GPS = gpsSensor('ReferenceFrame',RF)
GPS = gpsSensor( ___ ,Name,Value)
```

Description

`GPS = gpsSensor` returns a `gpsSensor` System object that computes a Global Positioning System receiver reading based on a local position and velocity input signal. The default reference position in geodetic coordinates is

- latitude: 0º N
- longitude: 0º E
- altitude: 0 m

`GPS = gpsSensor('ReferenceFrame',RF)` returns a `gpsSensor` System object that computes a global positioning system receiver reading relative to the reference frame `RF`. Specify `RF` as `'NED'` (North-East-Down) or `'ENU'` (East-North-Up). The default value is `'NED'`.

`GPS = gpsSensor( ___ ,Name,Value)` sets each property `Name` to the specified `Value`. Unspecified properties have default values.

Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see System Design in MATLAB Using System Objects.

SampleRate — Update rate of receiver (Hz)
1 (default) | positive real scalar

Update rate of the receiver in Hz, specified as a positive real scalar.

Data Types: single | double

ReferenceLocation — Origin of local navigation reference frame
[0 0 0] (default) | [latitude longitude altitude]

Reference location, specified as a 3-element row vector in geodetic coordinates (latitude, longitude, and altitude). Altitude is the height above the reference ellipsoid model, WGS84. The reference location is in [degrees degrees meters]. The degree format is decimal degrees (DD).

Data Types: single | double

PositionInputFormat — Position coordinate input format
'Local' (default) | 'Geodetic'

Position coordinate input format, specified as 'Local' or 'Geodetic'.

- If you set the property as 'Local', then you need to specify the truePosition input as Cartesian coordinates with respect to the local navigation frame whose origin is fixed and defined by the ReferenceLcation property. Additionally, when you specify the trueVelocity input, you need to specify it with respect to this local navigation frame.
- If you set the property as 'Geodetic', then you need to specify the truePosition input as geodetic coordinates in latitude, longitude, and altitude. Additionally, when you specify the trueVelocity input, you need to specify it with respect to the navigation frame (NED or ENU) whose origin corresponds to the truePosition input. When setting the property as 'Geodetic', the gpsSensor object neglects the ReferenceLocation property.

Data Types: character vector

HorizontalPositionAccuracy — Horizontal position accuracy (m)
1.6 (default) | nonnegative real scalar

Horizontal position accuracy in meters, specified as a nonnegative real scalar. The horizontal position accuracy specifies the standard deviation of the noise in the horizontal position measurement.

Tunable: Yes

Data Types: single | double

VerticalPositionAccuracy — Vertical position accuracy (m)
3 (default) | nonnegative real scalar

Vertical position accuracy in meters, specified as a nonnegative real scalar. The vertical position accuracy specifies the standard deviation of the noise in the vertical position measurement.

Tunable: Yes

Data Types: single | double

VelocityAccuracy — Velocity accuracy (m/s)
0.1 (default) | nonnegative real scalar

Velocity accuracy in meters per second, specified as a nonnegative real scalar. The velocity accuracy specifies the standard deviation of the noise in the velocity measurement.

Tunable: Yes

Data Types: single | double

DecayFactor — Global position noise decay factor
0.999 (default) | scalar in the range [0,1]

Global position noise decay factor, specified as a scalar in the range [0,1].

A decay factor of 0 models the global position noise as a white noise process. A decay factor of 1 models the global position noise as a random walk process.

Tunable: Yes

Data Types: single | double

RandomStream — Random number source
'Global stream' (default) | 'mt19937ar with seed'

Random number source, specified as a character vector or string:

- 'Global stream' -- Random numbers are generated using the current global random number stream.
- 'mt19937ar with seed' -- Random numbers are generated using the mt19937ar algorithm with the seed specified by the Seed property.

Data Types: char | string

Seed — Initial seed
67 (default) | nonnegative integer scalar

Initial seed of an mt19937ar random number generator algorithm, specified as a nonnegative integer scalar.

Dependencies

To enable this property, set RandomStream to 'mt19937ar with seed'.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

Usage

Syntax

```
[position,velocity,groundspeed,course] = GPS(truePosition,trueVelocity)
```

Description

`[position,velocity,groundspeed,course] = GPS(truePosition,trueVelocity)` computes global navigation satellite system receiver readings from the position and velocity inputs.

Input Arguments

`truePosition` — Position of GPS receiver in navigation coordinate system
N-by-3 matrix

Position of the GPS receiver in the navigation coordinate system, specified as a real finite *N*-by-3 matrix. *N* is the number of samples in the current frame.

- When the `PositionInputFormat` property is specified as `'Local'`, specify `truePosition` as Cartesian coordinates with respect to the local navigation frame whose origin is fixed at `ReferenceLocation`.
- When the `PositionInputFormat` property is specified as `'Geodetic'`, specify `truePosition` as geodetic coordinates in `[latitude longitude altitude]`. `Latitude` and `longitude` are in meters. `altitude` is the height above the WGS84 ellipsoid model in meters.

Data Types: `single` | `double`

`trueVelocity` — Velocity of GPS receiver in navigation coordinate system (m/s)
N-by-3 matrix

Velocity of GPS receiver in the navigation coordinate system in meters per second, specified as a real finite *N*-by-3 matrix. *N* is the number of samples in the current frame.

- When the `PositionInputFormat` property is specified as `'Local'`, specify `trueVelocity` with respect to the local navigation frame (NED or ENU) whose origin is fixed at `ReferenceLocation`.
- When the `PositionInputFormat` property is specified as `'Geodetic'`, specify `trueVelocity` with respect to the navigation frame (NED or ENU) whose origin corresponds to the `truePosition` input.

Data Types: `single` | `double`

Output Arguments

`position` — Position in LLA coordinate system
N-by-3 matrix

Position of the GPS receiver in the geodetic latitude, longitude, and altitude (LLA) coordinate system, returned as a real finite *N*-by-3 array. Latitude and longitude are in degrees with North and East being positive. Altitude is in meters.

N is the number of samples in the current frame.

Data Types: `single` | `double`

`velocity` — Velocity in local navigation coordinate system (m/s)
N-by-3 matrix

Velocity of the GPS receiver in the local navigation coordinate system in meters per second, returned as a real finite *N*-by-3 array. *N* is the number of samples in the current frame.

- When the `PositionInputFormat` property is specified as `'Local'`, the returned velocity is with respect to the local navigation frame whose origin is fixed at `ReferenceLocation`.

- When the `PositionInputFormat` property is specified as `'Geodetic'`, the returned velocity is with respect to the navigation frame (NED or ENU) whose origin corresponds to the `position` output.

Data Types: `single` | `double`

`groundspeed` — Magnitude of horizontal velocity in local navigation coordinate system (m/s)
N-by-1 column vector

Magnitude of the horizontal velocity of the GPS receiver in the local navigation coordinate system in meters per second, returned as a real finite *N*-by-1 column vector.

N is the number of samples in the current frame.

Data Types: `single` | `double`

`course` — Direction of horizontal velocity in local navigation coordinate system (°)
N-by-1 column vector

Direction of the horizontal velocity of the GPS receiver in the local navigation coordinate system in degrees, returned as a real finite *N*-by-1 column of values between 0 and 360. North corresponds to 360 degrees and East corresponds to 90 degrees.

N is the number of samples in the current frame.

Data Types: `single` | `double`

Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

Common to All System Objects

step	Run System object algorithm
release	Release resources and allow changes to System object property values and input characteristics
reset	Reset internal states of System object

Examples

Generate GPS Position Measurements From Stationary Input

Create a `gpsSensor` System object™ to model GPS receiver data. Assume a typical one Hz sample rate and a 1000-second simulation time. Define the reference location in terms of latitude, longitude, and altitude (LLA) of Natick, MA (USA). Define the sensor as stationary by specifying the true position and velocity with zeros.

```
fs = 1;
duration = 1000;
numSamples = duration*fs;
```

```
refLoc = [42.2825 -71.343 53.0352];

truePosition = zeros(numSamples,3);
trueVelocity = zeros(numSamples,3);

gps = gpsSensor('SampleRate',fs,'ReferenceLocation',refLoc);
```

Call `gps` with the specified `truePosition` and `trueVelocity` to simulate receiving GPS data for a stationary platform.

```
position = gps(truePosition,trueVelocity);
```

Plot the true position and the GPS sensor readings for position.

```
t = (0:(numSamples-1))/fs;

subplot(3, 1, 1)
plot(t, position(:,1), ...
     t, ones(numSamples)*refLoc(1))
title('GPS Sensor Readings')
ylabel('Latitude (degrees)')

subplot(3, 1, 2)
plot(t, position(:,2), ...
     t, ones(numSamples)*refLoc(2))
ylabel('Longitude (degrees)')

subplot(3, 1, 3)
plot(t, position(:,3), ...
     t, ones(numSamples)*refLoc(3))
ylabel('Altitude (m)')
xlabel('Time (s)')
```

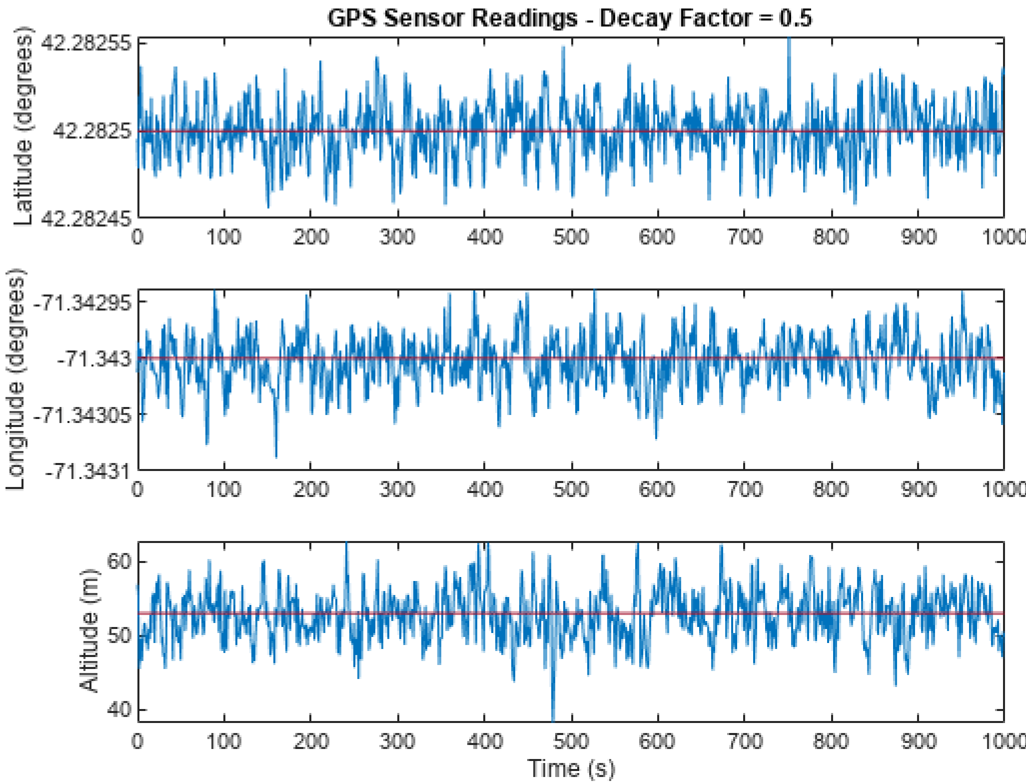
The position readings have noise controlled by `HorizontalPositionAccuracy`, `VerticalPositionAccuracy`, `VelocityAccuracy`, and `DecayFactor`. The `DecayFactor` property controls the drift in the noise model. By default, `DecayFactor` is set to `0.999`, which approaches a random walk process. To observe the effect of the `DecayFactor` property:

1. Reset the `gps` object.
2. Set `DecayFactor` to `0.5`.
3. Call `gps` with variables specifying a stationary position.
4. Plot the results.

The GPS position readings now oscillate around the true position.

```
reset(gps)
gps.DecayFactor = 0.5;
position = gps(truePosition,trueVelocity);

subplot(3, 1, 1)
plot(t, position(:,1), ...
     t, ones(numSamples)*refLoc(1))
title('GPS Sensor Readings - Decay Factor = 0.5')
ylabel('Latitude (degrees)')

subplot(3, 1, 2)
plot(t, position(:,2), ...
     t, ones(numSamples)*refLoc(2))
ylabel('Longitude (degrees)')
```

```
subplot(3, 1, 3)
plot(t, position(:,3), ...
     t, ones(numSamples)*refLoc(3))
ylabel('Altitude (m)')
xlabel('Time (s)')
```



Relationship Between Groundspeed and Course Accuracy

GPS receivers achieve greater course accuracy as groundspeed increases. In this example, you create a GPS receiver simulation object and simulate the data received from a platform that is accelerating from a stationary position.

Create a default `gpsSensor` System object™ to model data returned by a GPS receiver.

```
GPS = gpsSensor
```

```
GPS = 
  gpsSensor with properties:

                  SampleRate: 1                   Hz
         PositionInputFormat: 'Local'
           ReferenceLocation: [0 0 0]             [deg deg m]
   HorizontalPositionAccuracy: 1.6                 m
     VerticalPositionAccuracy: 3                   m
```

```
    VelocityAccuracy: 0.1                m/s
        RandomStream: 'Global stream'
         DecayFactor: 0.999
```

Create matrices to describe the position and velocity of a platform in the NED coordinate system. The platform begins from a stationary position and accelerates to 60 m/s North-East over 60 seconds, then has a vertical acceleration to 2 m/s over 2 seconds, followed by a 2 m/s rate of climb for another 8 seconds. Assume a constant velocity, such that the velocity is the simple derivative of the position.

```
duration = 70;
numSamples = duration*GPS.SampleRate;

course = 45*ones(duration,1);
groundspeed = [(1:60)';60*ones(10,1)];

Nvelocity   = groundspeed.*sind(course);
Evelocity   = groundspeed.*cosd(course);
Dvelocity   = [zeros(60,1);-1;-2*ones(9,1)];
NEDvelocity = [Nvelocity,Evelocity,Dvelocity];

Ndistance   = cumsum(Nvelocity);
Edistance   = cumsum(Evelocity);
Ddistance   = cumsum(Dvelocity);
NEDposition = [Ndistance,Edistance,Ddistance];
```

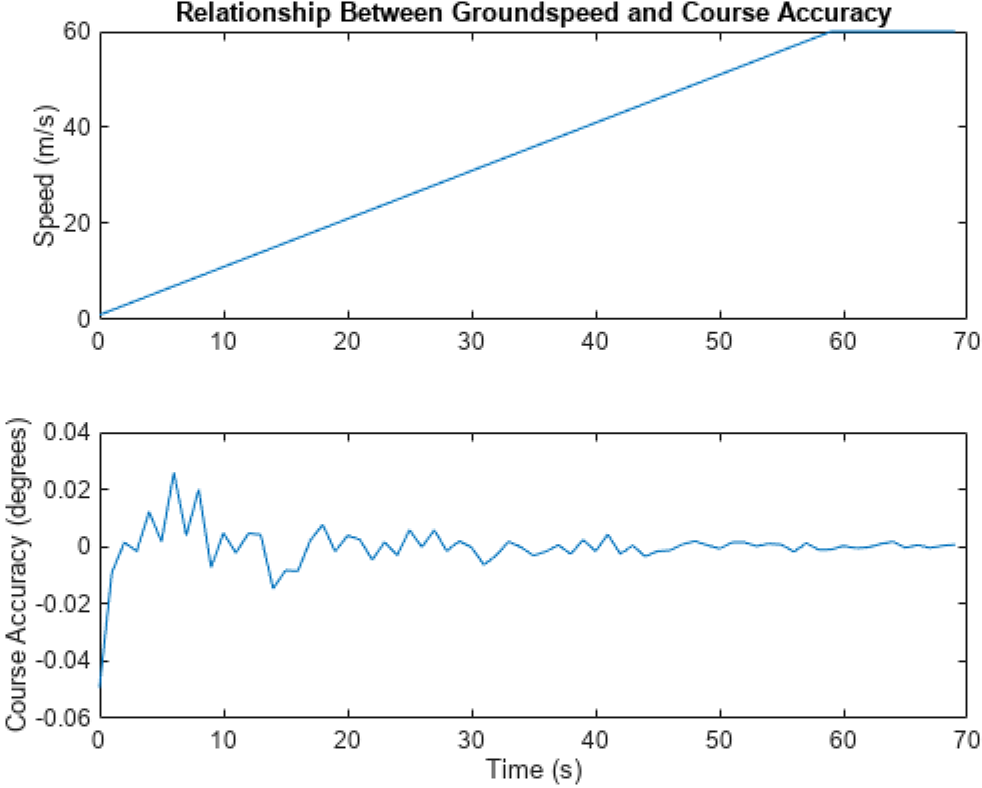
Model GPS measurement data by calling the `GPS` object with your velocity and position matrices.

```
[~,~,groundspeedMeasurement,courseMeasurement] = GPS(NEDposition,NEDvelocity);
```

Plot the groundspeed and the difference between the true course and the course returned by the GPS simulator.

As groundspeed increases, the accuracy of the course increases. Note that the velocity increase during the last ten seconds has no effect, because the additional velocity is not in the ground plane.

```
t = (0:numSamples-1)/GPS.SampleRate;

subplot(2,1,1)
plot(t,groundspeed);
ylabel('Speed (m/s)')
title('Relationship Between Groundspeed and Course Accuracy')

subplot(2,1,2)
courseAccuracy = courseMeasurement - course;
plot(t,courseAccuracy)
xlabel('Time (s)');
ylabel('Course Accuracy (degrees)')
```



Model GPS Receiver Data

Simulate GPS data received during a trajectory from the city of Natick, MA, to Boston, MA.

Define the decimal degree latitude and longitude for the city of Natick, MA USA, and Boston, MA USA. For simplicity, set the altitude for both locations to zero.

```
NatickLLA = [42.27752809999999, -71.34680909999997, 0];
BostonLLA = [42.3600825, -71.05888010000001, 0];
```

Define a motion that can take a platform from Natick to Boston in 20 minutes. Set the origin of the local NED coordinate system as Natick. Create a `waypointTrajectory` object to output the trajectory 10 samples at a time.

```
fs = 1;
duration = 60*20;

bearing = 68; % degrees
distance = 25.39e3; % meters
distanceEast = distance*sind(bearing);
distanceNorth = distance*cosd(bearing);

NatickNED = [0,0,0];
BostonNED = [distanceNorth,distanceEast,0];
```

```
trajectory = waypointTrajectory( ...
    'Waypoints', [NatickNED;BostonNED], ...
    'TimeOfArrival',[0;duration], ...
    'SamplesPerFrame',10, ...
    'SampleRate',fs);
```

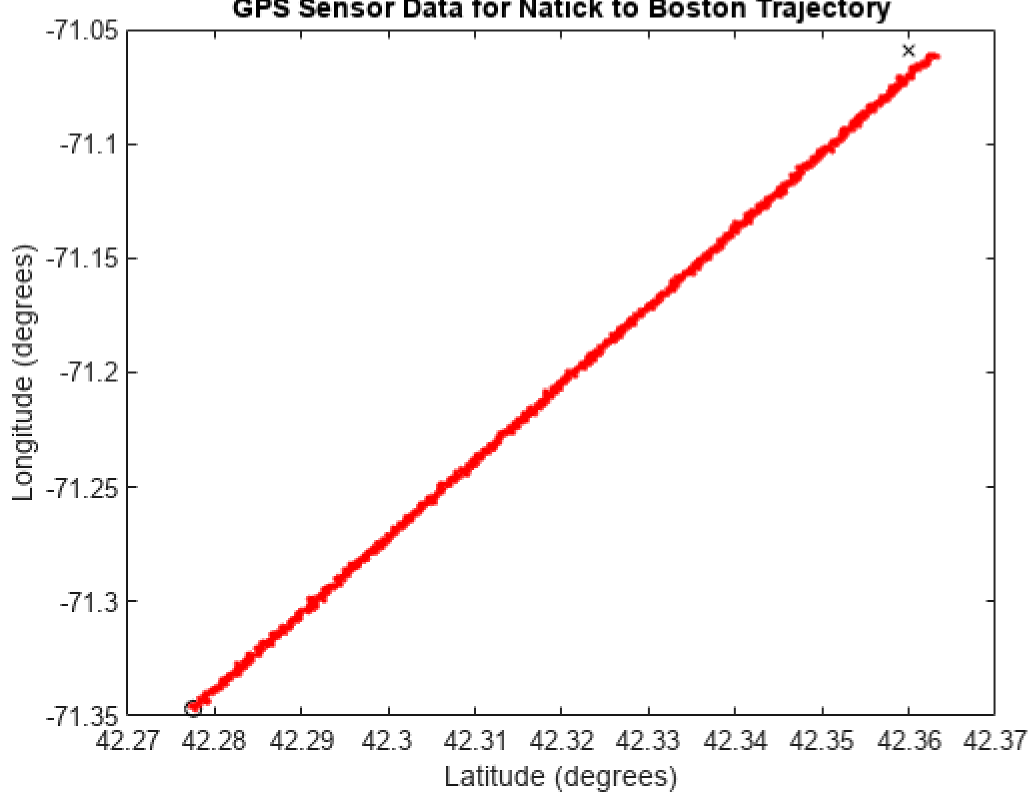
Create a `gpsSensor` object to model receiving GPS data for the platform. Set the `HorizontalPositionalAccuracy` to `25` and the `DecayFactor` to `0.25` to emphasize the noise. Set the `ReferenceLocation` to the Natick coordinates in LLA.

```
GPS = gpsSensor( ...
    'HorizontalPositionAccuracy',25, ...
    'DecayFactor',0.25, ...
    'SampleRate',fs, ...
    'ReferenceLocation',NatickLLA);
```

Open a figure and plot the position of Natick and Boston in LLA. Ignore altitude for simplicity.

In a loop, call the `gpsSensor` object with the ground-truth trajectory to simulate the received GPS data. Plot the ground-truth trajectory and the model of received GPS data.

```
figure(1)
plot(NatickLLA(1),NatickLLA(2),'ko', ...
     BostonLLA(1),BostonLLA(2),'kx')
xlabel('Latitude (degrees)')
ylabel('Longitude (degrees)')
title('GPS Sensor Data for Natick to Boston Trajectory')
hold on

while ~isDone(trajectory)
    [truePositionNED,~,trueVelocityNED] = trajectory();
    reportedPositionLLA = GPS(truePositionNED,trueVelocityNED);

    figure(1)
    plot(reportedPositionLLA(:,1),reportedPositionLLA(:,2),'r.')
end
```



As a best practice, release System objects when complete.

```
release(GPS)
release(trajectory)
```

Version History

Introduced in R2022a

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

Objects

`insSensor` | `robotSensor`

insSensor

Inertial navigation system and GNSS/GPS simulation model

Description

The `insSensor` System object models a device that fuses measurements from an inertial navigation system (INS) and global navigation satellite system (GNSS) such as a GPS, and outputs the fused measurements.

To output fused INS and GNSS measurements:

1. Create the `insSensor` object and set its properties.
2. Call the object with arguments, as if it were a function.

To learn more about how System objects work, see What Are System Objects?

Creation

Syntax

```
INS = insSensor
INS = insSensor(Name,Value)
```

Description

`INS = insSensor` returns a System object, `INS`, that models a device that outputs measurements from an INS and GNSS.

`INS = insSensor(Name,Value)` sets properties on page 1-198 using one or more name-value pairs. Unspecified properties have default values. Enclose each property name in quotes.

Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see System Design in MATLAB Using System Objects.

`MountingLocation` — Location of sensor on platform (m)
`[0 0 0]` (default) | three-element real-valued vector of form [*x y z*]

Location of the sensor on the platform, in meters, specified as a three-element real-valued vector of the form [*x y z*]. The vector defines the offset of the sensor origin from the origin of the platform.

Tunable: Yes

Data Types: `single` | `double`

`RollAccuracy` — Accuracy of roll measurement (deg)
`0.2` (default) | nonnegative real scalar

Accuracy of the roll measurement of the sensor body, in degrees, specified as a nonnegative real scalar.

Roll is the rotation around the *x*-axis of the sensor body. Roll noise is modeled as a white noise process. `RollAccuracy` sets the standard deviation of the roll measurement noise.

Tunable: Yes

Data Types: `single` | `double`

`PitchAccuracy` — Accuracy of pitch measurement (deg)
`0.2` (default) | nonnegative real scalar

Accuracy of the pitch measurement of the sensor body, in degrees, specified as a nonnegative real scalar.

Pitch is the rotation around the *y*-axis of the sensor body. Pitch noise is modeled as a white noise process. `PitchAccuracy` defines the standard deviation of the pitch measurement noise.

Tunable: Yes

Data Types: `single` | `double`

`YawAccuracy` — Accuracy of yaw measurement (deg)
`1` (default) | nonnegative real scalar

Accuracy of the yaw measurement of the sensor body, in degrees, specified as a nonnegative real scalar.

Yaw is the rotation around the *z*-axis of the sensor body. Yaw noise is modeled as a white noise process. `YawAccuracy` defines the standard deviation of the yaw measurement noise.

Tunable: Yes

Data Types: `single` | `double`

`PositionAccuracy` — Accuracy of position measurement (m)
`[1 1 1]` (default) | nonnegative real scalar | three-element real-valued vector

Accuracy of the position measurement of the sensor body, in meters, specified as a nonnegative real scalar or a three-element real-valued vector. The elements of the vector set the accuracy of the *x*-, *y*-, and *z*-position measurements, respectively. If you specify `PositionAccuracy` as a scalar value, then the object sets the accuracy of all three positions to this value.

Position noise is modeled as a white noise process. `PositionAccuracy` defines the standard deviation of the position measurement noise.

Tunable: Yes

Data Types: `single` | `double`

`VelocityAccuracy` — Accuracy of velocity measurement (m/s)
`0.05` (default) | nonnegative real scalar

Accuracy of the velocity measurement of the sensor body, in meters per second, specified as a nonnegative real scalar.

Velocity noise is modeled as a white noise process. `VelocityAccuracy` defines the standard deviation of the velocity measurement noise.

Tunable: Yes

Data Types: `single` | `double`

`AccelerationAccuracy` — Accuracy of acceleration measurement (m/s^2)
`0` (default) | nonnegative real scalar

Accuracy of the acceleration measurement of the sensor body, in meters per second, specified as a nonnegative real scalar.

Acceleration noise is modeled as a white noise process. `AccelerationAccuracy` defines the standard deviation of the acceleration measurement noise.

Tunable: Yes

Data Types: `single` | `double`

`AngularVelocityAccuracy` — Accuracy of angular velocity measurement (deg/s)
`0` (default) | nonnegative real scalar

Accuracy of the angular velocity measurement of the sensor body, in meters per second, specified as a nonnegative real scalar.

Angular velocity is modeled as a white noise process. `AngularVelocityAccuracy` defines the standard deviation of the acceleration measurement noise.

Tunable: Yes

Data Types: `single` | `double`

`TimeInput` — Enable input of simulation time
`false` or `0` (default) | `true` or `1`

Enable input of simulation time, specified as a logical `0` (`false`) or `1` (`true`). Set this property to `true` to input the simulation time by using the `simTime` argument.

Tunable: No

Data Types: `logical`

`HasGNSSFix` — Enable GNSS fix
`true` or `1` (default) | `false` or `0`

Enable GNSS fix, specified as a logical `1` (`true`) or `0` (`false`). Set this property to `false` to simulate the loss of a GNSS receiver fix. When a GNSS receiver fix is lost, position measurements drift at a rate specified by the `PositionErrorFactor` property.

Tunable: Yes

Dependencies

To enable this property, set `TimeInput` to `true`.

Data Types: `logical`

`PositionErrorFactor` — Position error factor without GNSS fix

`[0 0 0]` (default) | nonnegative scalar | 1-by-3 vector of scalars

Position error factor without GNSS fix, specified as a scalar or a 1-by-3 vector of scalars.

When the `HasGNSSFix` property is set to `false`, the position error grows at a quadratic rate due to constant bias in the accelerometer. The position error for a position component $E(t)$ can be expressed as $E(t) = 1/2\alpha t^2$, where α is the position error factor for the corresponding component and t is the time since the GNSS fix is lost. While running, the object computes t based on the `simTime` input. The computed $E(t)$ values for the x, y, and z components are added to the corresponding position components of the `gTruth` input.

Tunable: Yes

Dependencies

To enable this property, set `TimeInput` to `true` and `HasGNSSFix` to `false`.

Data Types: `single` | `double`

`RandomStream` — Random number source

`'Global stream'` (default) | `'mt19937ar with seed'`

Random number source, specified as one of these options:

- `'Global stream'` –– Generate random numbers using the current global random number stream.
- `'mt19937ar with seed'` –– Generate random numbers using the mt19937ar algorithm, with the seed specified by the `Seed` property.

Data Types: `char` | `string`

Seed — Initial seed

`67` (default) | nonnegative integer

Initial seed of the mt19937ar random number generator algorithm, specified as a nonnegative integer.

Dependencies

To enable this property, set `RandomStream` to `'mt19937ar with seed'`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Usage

Syntax

```
measurement = INS(gTruth)
measurement = INS(gTruth,simTime)
```

Description

`measurement = INS(gTruth)` models the data received from an INS sensor reading and GNSS sensor reading. The output measurement is based on the inertial ground-truth state of the sensor body, `gTruth`.

`measurement = INS(gTruth,simTime)` additionally specifies the time of simulation, `simTime`. To enable this syntax, set the `TimeInput` property to `true`.

Input Arguments

`gTruth` — Inertial ground-truth state of sensor body
structure

Inertial ground-truth state of sensor body, in local Cartesian coordinates, specified as a structure containing these fields:

Field	Description
`'Position'`	Position, in meters, specified as a real, finite *N*-by-3 matrix of [*x y z*] vectors. *N* is the number of samples in the current frame.
`'Velocity'`	Velocity (*v*), in meters per second, specified as a real, finite *N*-by-3 matrix of [v_x v_y v_z] vector. *N* is the number of samples in the current frame.
`'Orientation'`	Orientation with respect to the local Cartesian coordinate system, specified as one of these options: • *N*-element column vector of `quaternion` objects • 3-by-3-by-*N* array of rotation matrices • *N*-by-3 matrix of [x_{roll} y_{pitch} z_{yaw}] angles in degrees Each quaternion or rotation matrix is a frame rotation from the local Cartesian coordinate system to the current sensor body coordinate system. *N* is the number of samples in the current frame.
`'Acceleration'`	Acceleration (*a*), in meters per second squared, specified as a real, finite *N*-by-3 matrix of [a_x a_y a_z] vectors. *N* is the number of samples in the current frame.
`'AngularVelocity'`	Angular velocity (*ω*), in degrees per second squared, specified as a real, finite *N*-by-3 matrix of [ω_x ω_y ω_z] vectors. *N* is the number of samples in the current frame.

The field values must be of type `double` or `single`.

The `Position`, `Velocity`, and `Orientation` fields are required. The other fields are optional.

Example: `struct('Position',[0 0 0],'Velocity',[0 0 0],'Orientation',quaternion([1 0 0 0]))`

`simTime` — Simulation time
nonnegative real scalar

Simulation time, in seconds, specified as a nonnegative real scalar.

Data Types: `single` | `double`

Output Arguments

`measurement` — Measurement of sensor body motion
structure

Measurement of the sensor body motion, in local Cartesian coordinates, returned as a structure containing these fields:

Field	Description
`'Position'`	Position, in meters, specified as a real, finite *N*-by-3 matrix of [*x y z*] vectors. *N* is the number of samples in the current frame.
`'Velocity'`	Velocity (*v*), in meters per second, specified as a real, finite *N*-by-3 matrix of [v_x v_y v_z] vector. *N* is the number of samples in the current frame.
`'Orientation'`	Orientation with respect to the local Cartesian coordinate system, specified as one of these options: • *N*-element column vector of `quaternion` objects • 3-by-3-by-*N* array of rotation matrices • *N*-by-3 matrix of [x_{roll} y_{pitch} z_{yaw}] angles in degrees Each quaternion or rotation matrix is a frame rotation from the local Cartesian coordinate system to the current sensor body coordinate system. *N* is the number of samples in the current frame.
`'Acceleration'`	Acceleration (*a*), in meters per second squared, specified as a real, finite *N*-by-3 matrix of [a_x a_y a_z] vectors. *N* is the number of samples in the current frame.
`'AngularVelocity'`	Angular velocity (*ω*), in degrees per second squared, specified as a real, finite *N*-by-3 matrix of [ω_x ω_y ω_z] vectors. *N* is the number of samples in the current frame.

The returned field values are of type `double` or `single` and are of the same type as the corresponding field values in the `gTruth` input.

Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

Specific to `insSensor`

perturbations	Perturbation defined on object
perturb	Apply perturbations to object

Common to All System Objects

step	Run System object algorithm
clone	Create duplicate System object
isLocked	Determine if System object is in use
reset	Reset internal states of System object
release	Release resources and allow changes to System object property values and input characteristics

Examples

Generate INS Measurements from Stationary Input

Create a motion structure that defines a stationary position at the local north-east-down (NED) origin. Because the platform is stationary, you need to define only a single sample. Assume the ground-truth motion is sampled for 10 seconds with a 100 Hz sample rate. Create a default `insSensor` System object™. Preallocate variables to hold output from the `insSensor` object.

```
Fs = 100;
duration = 10;
numSamples = Fs*duration;

motion = struct( ...
    'Position',zeros(1,3), ...
    'Velocity',zeros(1,3), ...
    'Orientation',ones(1,1,'quaternion'));

INS = insSensor;

positionMeasurements = zeros(numSamples,3);
velocityMeasurements = zeros(numSamples,3);
orientationMeasurements = zeros(numSamples,1,'quaternion');
```

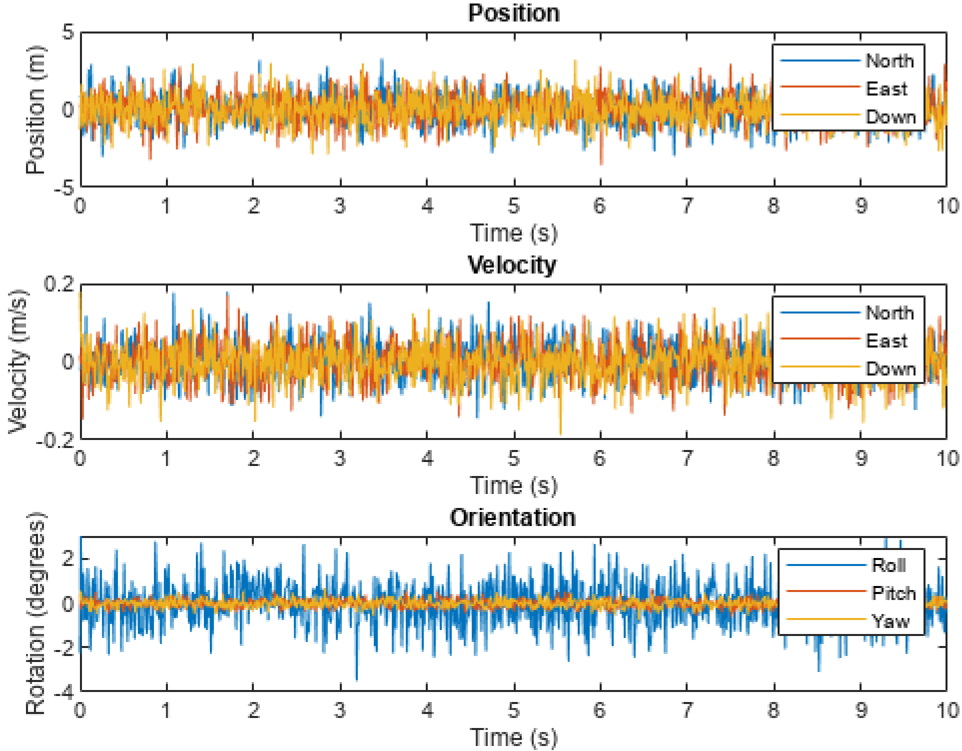
In a loop, call `INS` with the stationary motion structure to return the position, velocity, and orientation measurements in the local NED coordinate system. Log the position, velocity, and orientation measurements.

```
for i = 1:numSamples

    measurements = INS(motion);

    positionMeasurements(i,:) = measurements.Position;
    velocityMeasurements(i,:) = measurements.Velocity;
```

```
    orientationMeasurements(i) = measurements.Orientation;
end
```

Convert the orientation from quaternions to Euler angles for visualization purposes. Plot the position, velocity, and orientation measurements over time.

```
orientationMeasurements = eulerd(orientationMeasurements,'ZYX','frame');

t = (0:(numSamples-1))/Fs;

subplot(3,1,1)
plot(t,positionMeasurements)
title('Position')
xlabel('Time (s)')
ylabel('Position (m)')
legend('North','East','Down')

subplot(3,1,2)
plot(t,velocityMeasurements)
title('Velocity')
xlabel('Time (s)')
ylabel('Velocity (m/s)')
legend('North','East','Down')

subplot(3,1,3)
plot(t,orientationMeasurements)
title('Orientation')
xlabel('Time (s)')
ylabel('Rotation (degrees)')
legend('Roll', 'Pitch', 'Yaw')
```



Generate INS Measurements for a Turning Platform

Generate INS measurements using the `insSensor` System object™. Use `waypointTrajectory` to generate the ground-truth path.

Specify a ground-truth orientation that begins with the sensor body *x*-axis aligned with North and ends with the sensor body *x*-axis aligned with East. Specify waypoints for an arc trajectory and a time-of-arrival vector for the corresponding waypoints. Use a 100 Hz sample rate. Create a `waypointTrajectory` System object with the waypoint constraints, and set `SamplesPerFrame` so that the entire trajectory is output with one call.

```
eulerAngles = [0,0,0; ...
               0,0,0; ...
               90,0,0; ...
               90,0,0];
orientation = quaternion(eulerAngles,'eulerd','ZYX','frame');

r = 20;
waypoints = [0,0,0; ...
             100,0,0; ...
             100+r,r,0; ...
             100+r,100+r,0];

toa = [0,10,10+(2*pi*r/4),20+(2*pi*r/4)];
```

```
Fs = 100;
numSamples = floor(Fs*toa(end));

path = waypointTrajectory('Waypoints',waypoints, ...
    'TimeOfArrival',toa, ...
    'Orientation',orientation, ...
    'SampleRate',Fs, ...
    'SamplesPerFrame',numSamples);
```

Create an `insSensor` System object to model receiving INS data. Set the `PositionAccuracy` to `0.1`.

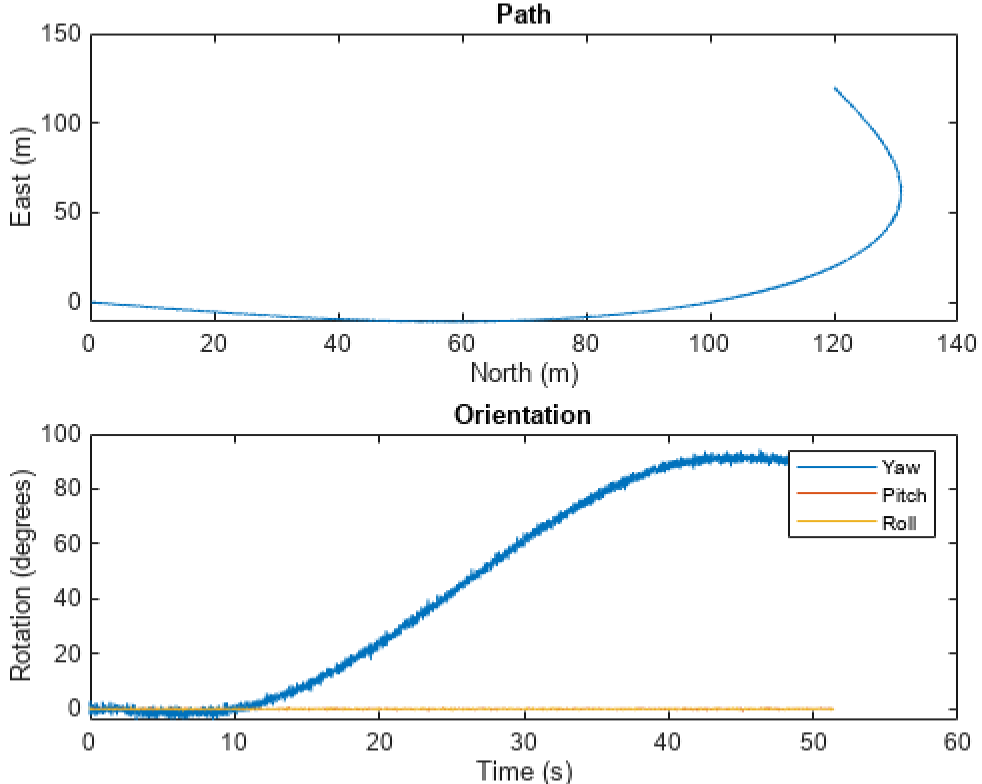
```
ins = insSensor('PositionAccuracy',0.1);
```

Call the waypoint trajectory object, `path`, to generate the ground-truth motion. Call the INS simulator, `ins`, with the ground-truth motion to generate INS measurements.

```
[motion.Position,motion.Orientation,motion.Velocity] = path();
insMeas = ins(motion);
```

Convert the orientation returned by `ins` to Euler angles in degrees for visualization purposes. Plot the full path and orientation over time.

```
orientationMeasurementEuler = eulerd(insMeas.Orientation,'ZYX','frame');

subplot(2,1,1)
plot(insMeas.Position(:,1),insMeas.Position(:,2));
title('Path')
xlabel('North (m)')
ylabel('East (m)')

subplot(2,1,2)
t = (0:(numSamples-1)).'/Fs;
plot(t,orientationMeasurementEuler(:,1), ...
     t,orientationMeasurementEuler(:,2), ...
     t,orientationMeasurementEuler(:,3));
title('Orientation')
legend('Yaw','Pitch','Roll')
xlabel('Time (s)')
ylabel('Rotation (degrees)')
```



Version History

Introduced in R2022a

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

The object functions, `perturbations` and `perturb`, do not support code generation.

Usage notes and limitations:

See "System Objects in MATLAB Code Generation" (MATLAB Coder).

See Also

Objects

`gpsSensor` | `robotSensor`

interactiveRigidBodyTree

Interact with rigid body tree robot models

Description

The `interactiveRigidBodyTree` object creates a figure that displays a robot model using a `rigidBodyTree` object and enables you to directly modify the robot configuration using an interactive marker. The `rigidBodyTree` object defines the geometry of the different connected rigid bodies in the robot model and the joint limits for these bodies.

To compute new configurations using inverse kinematics, click and drag the interactive marker in the figure. The marker supports dragging of the center marker, linear motion along specific axes, and rotation about each axes. You can change the end effector by right-clicking a different body and choosing **Set body as marker body**.

To save the current configuration of the robot model, use the `addConfiguration` object function. The StoredConfigurations property contains the saved configurations.

By default, the joint limits of the robot model are the only constraint on the robot. To add additional constraints, use the `addConstraint` object function. For a list of available constraint objects, see **Robot Constraints** in “Inverse Kinematics”.

Creation

Syntax

```
viztree = interactiveRigidBodyTree(robot)
viztree = interactiveRigidBodyTree(robot,'Frames','off')
viztree = interactiveRigidBodyTree( ___ ,Name,Value)
```

Description

`viztree = interactiveRigidBodyTree(robot)` creates an interactive rigid body tree object and associated figure for the specified robot model. The `robot` argument sets the `RigidBodyTree` property. To interact with the model, click and drag the interactive marker in the figure.

`viztree = interactiveRigidBodyTree(robot,'Frames','off')` turns off the frame axes plotted for each joint in the robot model.

`viztree = interactiveRigidBodyTree( ___ ,Name,Value)` sets properties using one or more name-value pair arguments in addition to any of the input argument combinations in previous syntaxes. Enclose each property name in quotes. For example, `'RigidBodyTree',robot` creates an interactive rigid body tree object with the specified robot model.

Properties

`RigidBodyTree` — Rigid body tree robot model
`rigidBodyTree` object

Rigid body tree robot model, specified as a `rigidBodyTree` object. The robot model defines the geometry of the rigid bodies and the joints connecting them. To access provided robot models, use the `loadrobot` function. To import models from URDF files or Simscape™ Multibody™ models, use the `importrobot` function.

You can set this property when you create the object. After you create the object, this property is read-only.

`IKSolver` — Inverse kinematics solver
`generalizedInverseKinematics` System object with default properties (default) | `generalizedInverseKinematics` object

Inverse kinematics solver, specified as a `generalizedInverseKinematics` System object. By default, the solver uses the Levenberg-Marquardt algorithm with a maximum number of iterations of 2. Increasing the maximum number of iterations can decrease the frame rate in the figure.

You can set this property when you create the object. After you create the object, this property is read-only.

`MarkerBodyName` — Name of rigid body associated with interactive marker
`viztree.RigidBodyTree.BodyNames{end}` (default) | string scalar | character vector

Name of rigid body associated with interactive marker, specified as a string scalar or character vector. By default, this property is set to the body with the highest index in the `RigidBodyTree` property. To change this property using the figure, right-click a rigid body and select **Set body as marker body**.

Example: `"r_foot"`

Data Types: `char` | `string`

`MarkerPose` — Current pose of interactive marker
4-by-4 homogeneous transformation matrix

This property is read-only.

Current pose of interactive marker, specified as a 4-by-4 homogeneous transformation matrix.

Data Types: `double`

`MarkerBodyPose` — Current pose of rigid body associated with interactive marker
4-by-4 homogeneous transformation matrix

This property is read-only.

Current pose of the rigid body associated with the interactive marker, specified as a 4-by-4 homogeneous transformation matrix.

Data Types: `double`

`Constraints` — Constraints on inverse kinematics solver
`{}` (default) | cell array of constraint objects

Constraints on inverse kinematics solver, specified as a cell array of one or more constraint objects:

- `constraintAiming`

- `constraintCartesianBounds`
- `constraintJointBounds`
- `constraintOrientationTarget`
- `constraintPoseTarget`
- `constraintPositionTarget`

By default, the inverse kinematics solver respects only the joint limits of the RigidBodyTree property. To add or remove the constraints on the robot model, use the `addConstraint` and `removeConstraints` object functions respectively. Alternatively, you can assign a new cell array of constraint objects to the property.

Example: `{constraintAiming("head","ReferenceBody","right_hand")}`

Data Types: `cell`

`SolverPoseWeights` — Weights on orientation and position of target pose
`[1 1]` (default) | two-element vector `[`*`orientation position`*`]`

Weights on orientation and position of target pose, respectively, specified as a two-element vector, `[`*`orientation position`*`]`.

To increase priority for matching a desired orientation or position, increase the corresponding weight value.

Example: `[1 4]`

Data Types: `double`

`ShowMarker` — Visibility of interactive marker in figure
`true` or `1` (default) | `false` or `0`

Visibility of interactive marker in figure, specified as logical `1` (`true`) or `0` or (`false`). Set `ShowMarker` to `false` to hide the interactive marker in the figure.

Data Types: `logical`

`MarkerControlMethod` — Type of control for interactive marker
`"InverseKinematics"` (default) | `"JointControl"`

Type of control for interactive marker, specified as `"InverseKinematics"` or `"JointControl"`. By default, the figure computes all the joint configurations of the robot by using inverse kinematics with the end effector set to MarkerBodyName property value. To control the position of a specific joint on the selected rigid body, set this property to `"JointControl"`.

Data Types: `char` | `string`

`MarkerScaleFactor` — Relative scale of interactive marker
`1` (default) | positive real number

Relative scale of interactive marker, specified as a positive real number. To increase or decrease the size of the marker in the figure, adjust this property.

Data Types: `double`

`Configuration` — Current configuration of rigid body tree robot model
`homeConfiguration(viztree.RigidBodyTree)` (default) | *n*-element vector

Current configuration of rigid body tree robot model, specified as an *n*-element vector. `n` is the number of nonfixed joints in the RigidBodyTree property.

Example: `[1 pi 0 0.5 3.156]'`

Data Types: `double`

`StoredConfigurations` — Stored robot configurations

`[]` (default) | *n*-by-*p* matrix

Stored robot configurations, specified as an *n*-by-*p* matrix. Each column of the matrix is a stored robot configuration. *n* is the number of nonfixed joints in the `RigidBodyTree` property. *p* is the number of stored robot configurations. To add or remove stored configurations, use the `addConfiguration` or `removeConfigurations` object functions, respectively.

Data Types: `double`

Object Functions

addConfiguration	Store current configuration
addConstraint	Add inverse kinematics constraint
removeConfigurations	Remove configurations from StoredConfigurations property
removeConstraints	Remove inverse kinematics constraints
showFigure	Show interactive rigid body tree figure

Examples

Interactively Build and Play Back Series of Robot Configurations

Use the `interactiveRigidBodyTree` object to manually move around a robot in a figure. The object uses interactive markers in the figure to track the desired poses of different rigid bodies in the `rigidBodyTree` object.

Load Robot Model

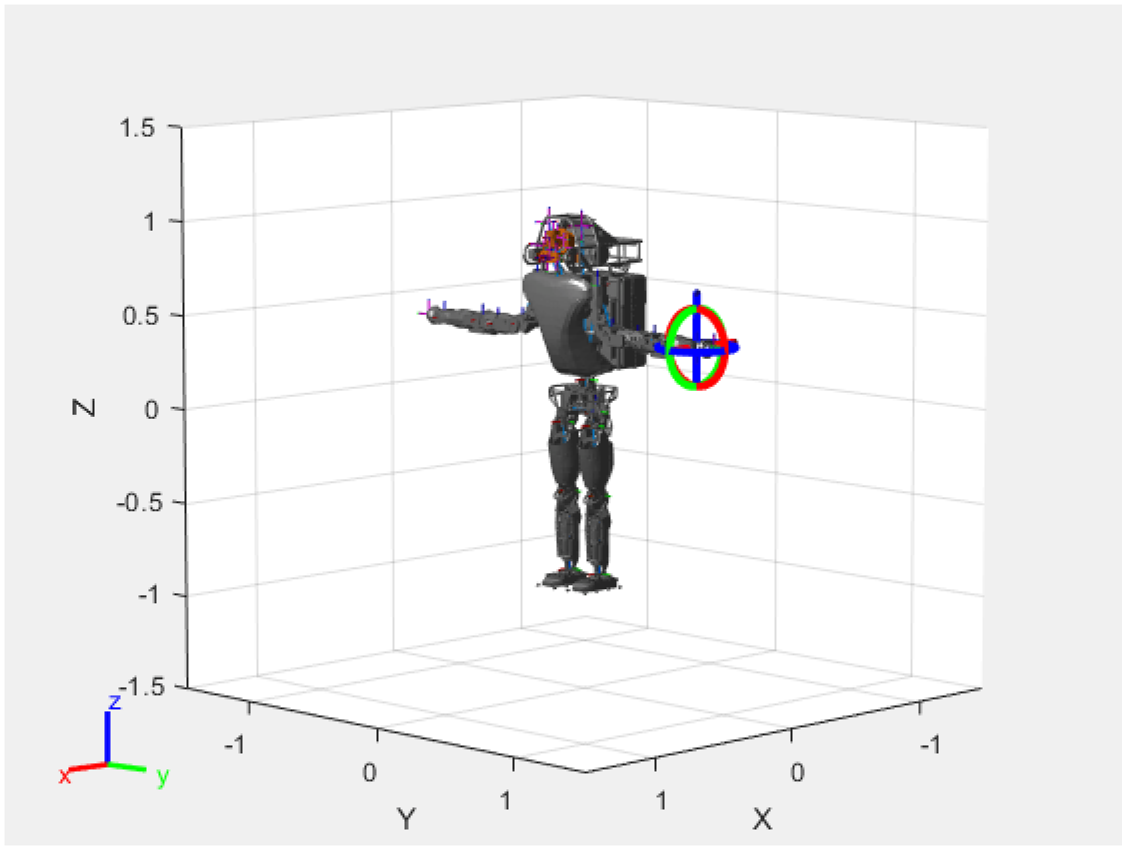
Use the `loadrobot` function to access provided robot models as `rigidBodyTree` objects.

```
robot = loadrobot("atlas");
```

Visualize Robot and Save Configurations

Create an interactive tree object and associated figure, specifying the loaded robot model and its left hand as the end effector.

```
viztree = interactiveRigidBodyTree(robot,"MarkerBodyName","l_hand");
```

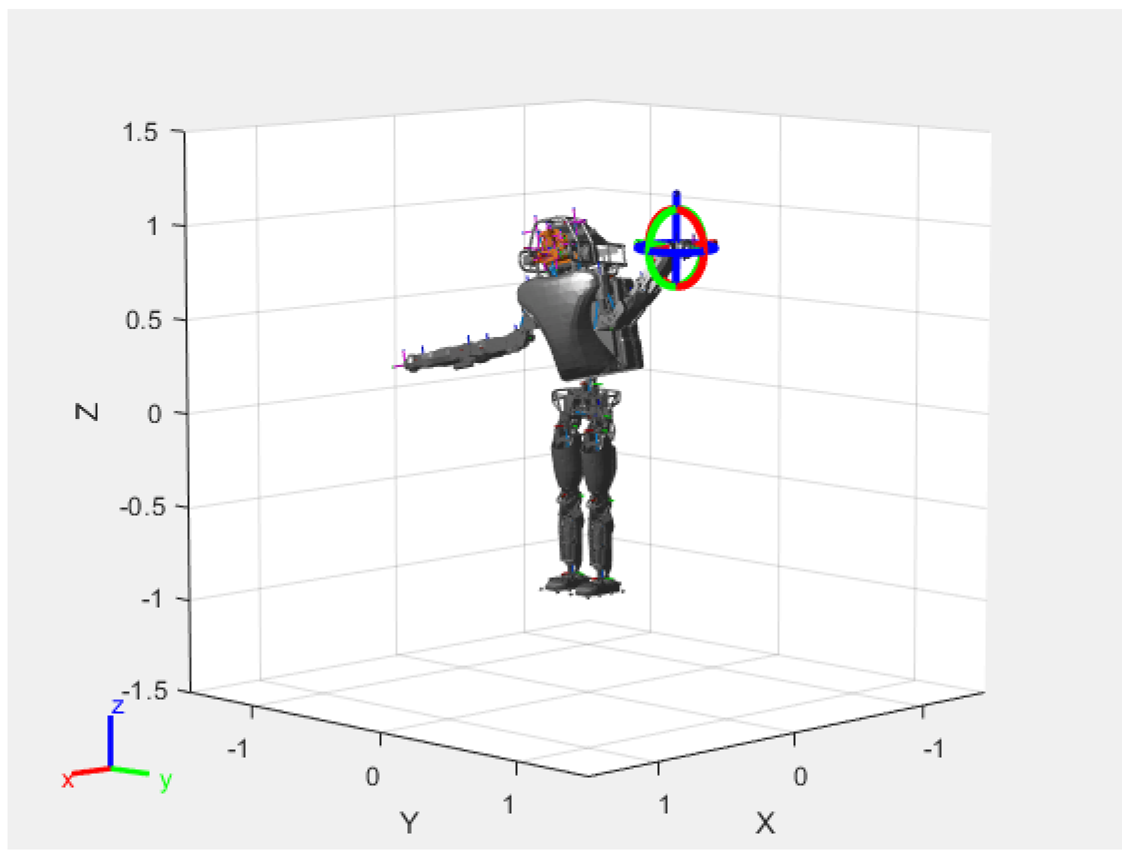


Click and drag the interactive marker to change the robot configuration. You can click and drag any of the axes for linear motion, rotate the body about an axis using the red, green, and blue circles, and drag the center of the interactive marker to position it in 3-D space.

The `interactiveRigidBodyTree` object uses inverse kinematics to determine a configuration that achieves the desired end-effector pose. If the associated rigid body cannot reach the marker, the figure renders the best configuration from the inverse kinematics solver.

Programmatically set the current configuration. Assign a vector of length equal to the number of nonfixed joints in the `RigidBodyTree` to the `Configuration` property.

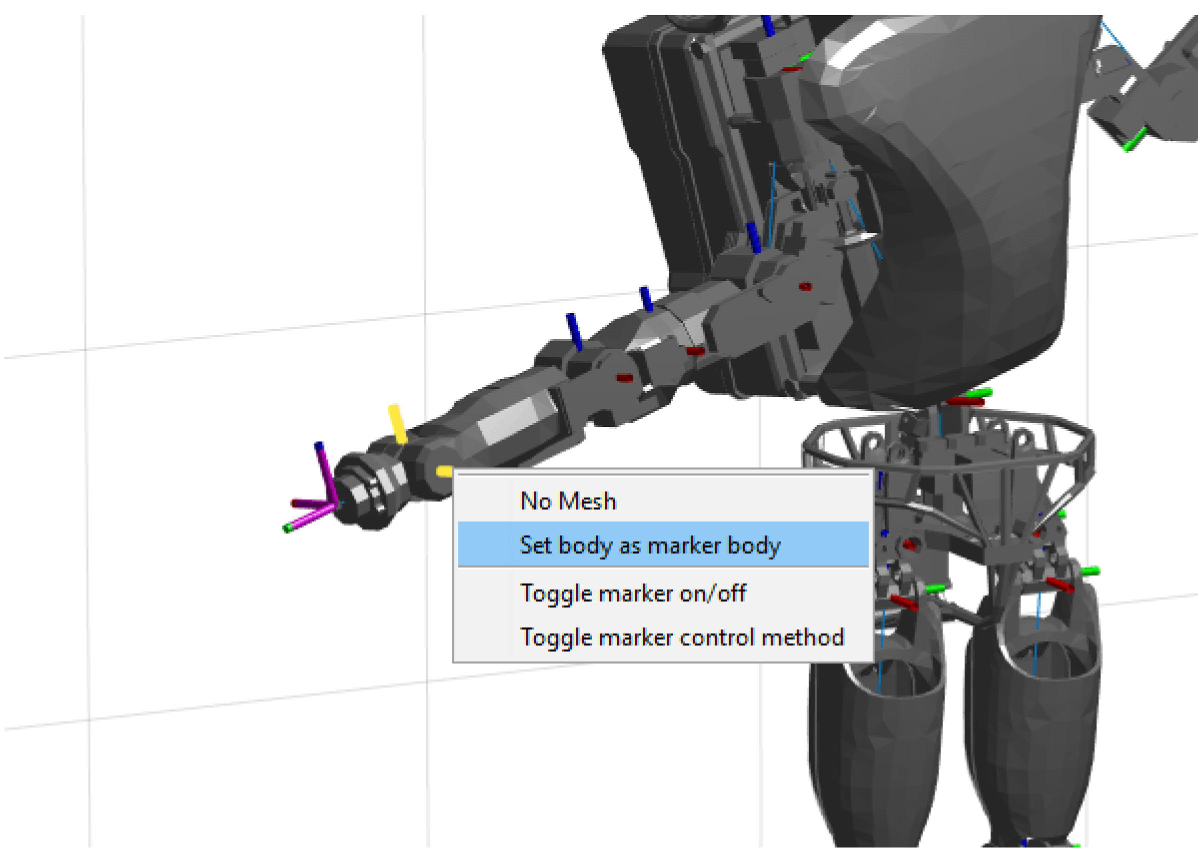
```
currConfig = homeConfiguration(viztree.RigidBodyTree);
currConfig(1:10) = [ 0.2201 -0.1319 0.2278 -0.3415 0.4996 ...
                     0.0747 0.0377 0.0718 -0.8117 -0.0427]';
viztree.Configuration = currConfig;
```



Save the current robot configuration in the `StoredConfigurations` property.

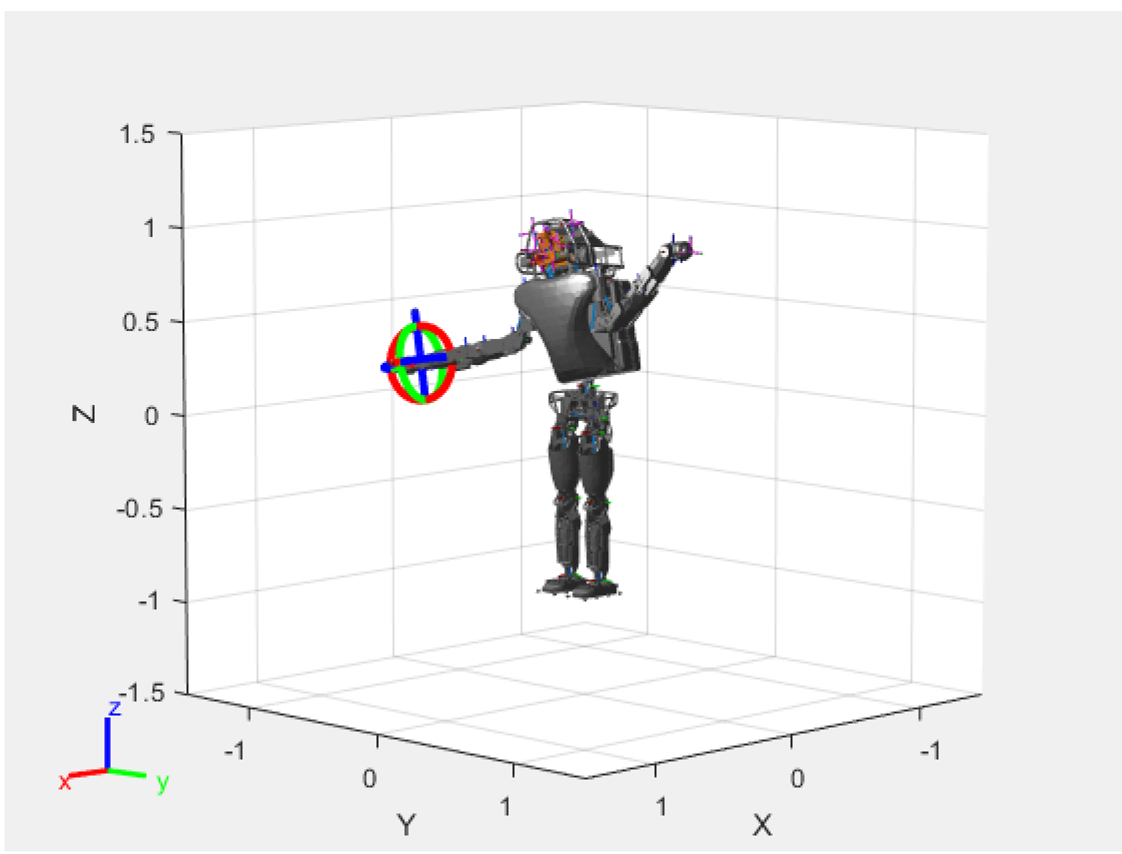
```
addConfiguration(viztree)
```

To switch the end effector to a different rigid body, right-click the desired body in the figure and select **Set body as marker body**. Use this process to select the right hand frame.



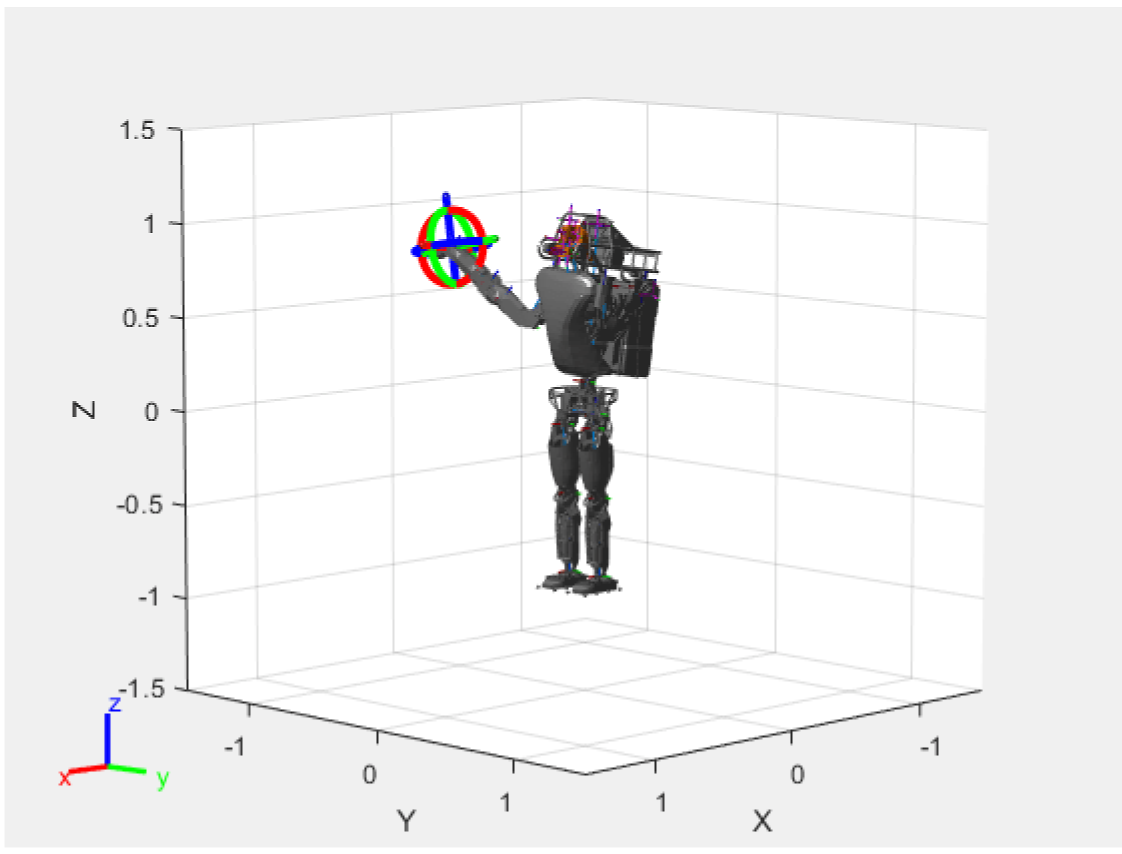
You can also set the `MarkerBodyName` property to the specific body name.

```
viztree.MarkerBodyName = "r_hand";
```



Move the right hand to a new position. Set the configuration programmatically. The marker moves to the new position of the end effector.

```
currConfig(1:18) = [-0.1350 -0.1498 -0.0167 -0.3415 0.4996 0.0747
                     0.0377 0.0718 -0.8117 -0.0427 0 0.4349
                    -0.5738 0.0563 -0.0095 0.0518 0.8762 -0.0895]';

viztree.Configuration = currConfig;
```



Save the current configuration.

```
addConfiguration(viztree)
```

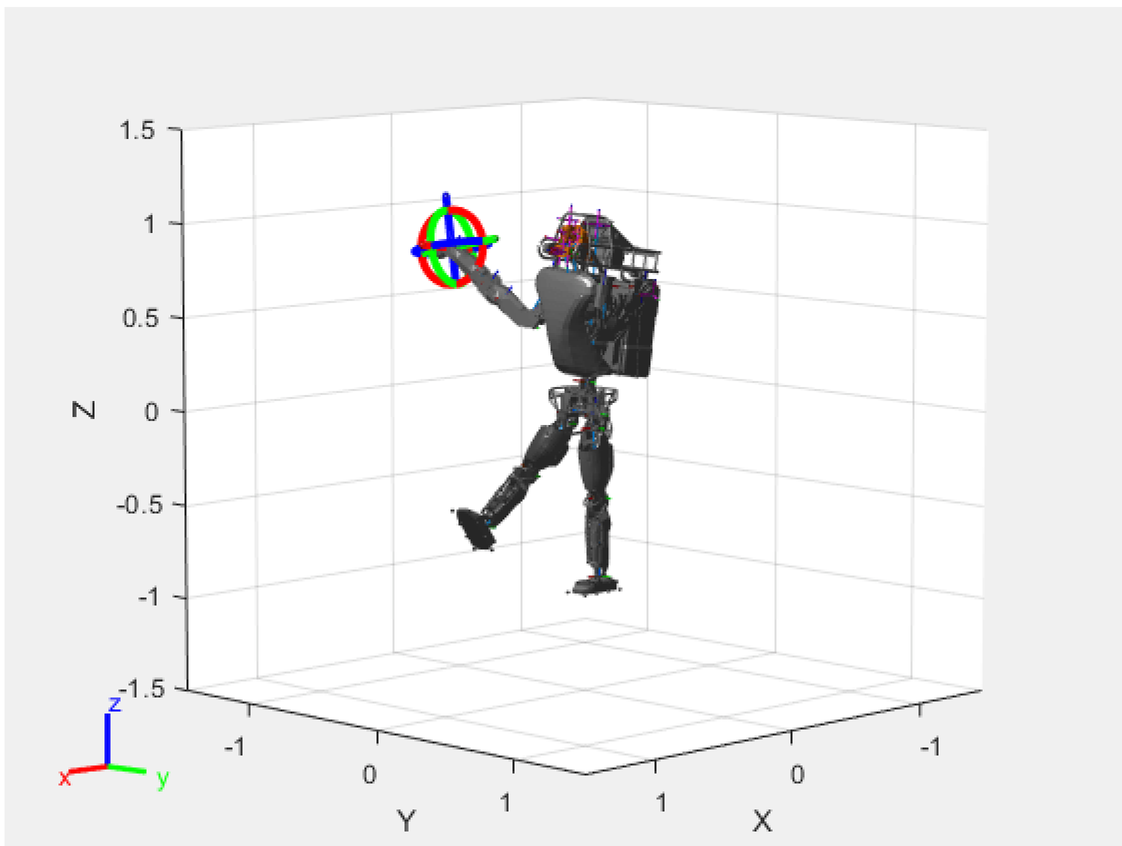
Add Constraints

By default, the robot model respects only the joint limits of the `rigidBodyJoint` objects associated with the `RigidBodyTree` property. To add constraints, generate **Robot Constraint** objects and specify them as a cell array in the `Constraints` property. To see a list of robotic constraints, see "Inverse Kinematics". Specify a pose target for the pelvis to keep it fixed to the home position. Specify a position target for the right foot to be raised in front front and above its current position.

```
fixedWaist = constraintPoseTarget("pelvis");
raiseRightLeg = constraintPositionTarget("r_foot","TargetPosition",[1 0 0.5]);
```

Apply these constraints to the interactive rigid body tree object as a cell array. The right leg in the resulting figure changes position.

```
viztree.Constraints = {fixedWaist raiseRightLeg};
```



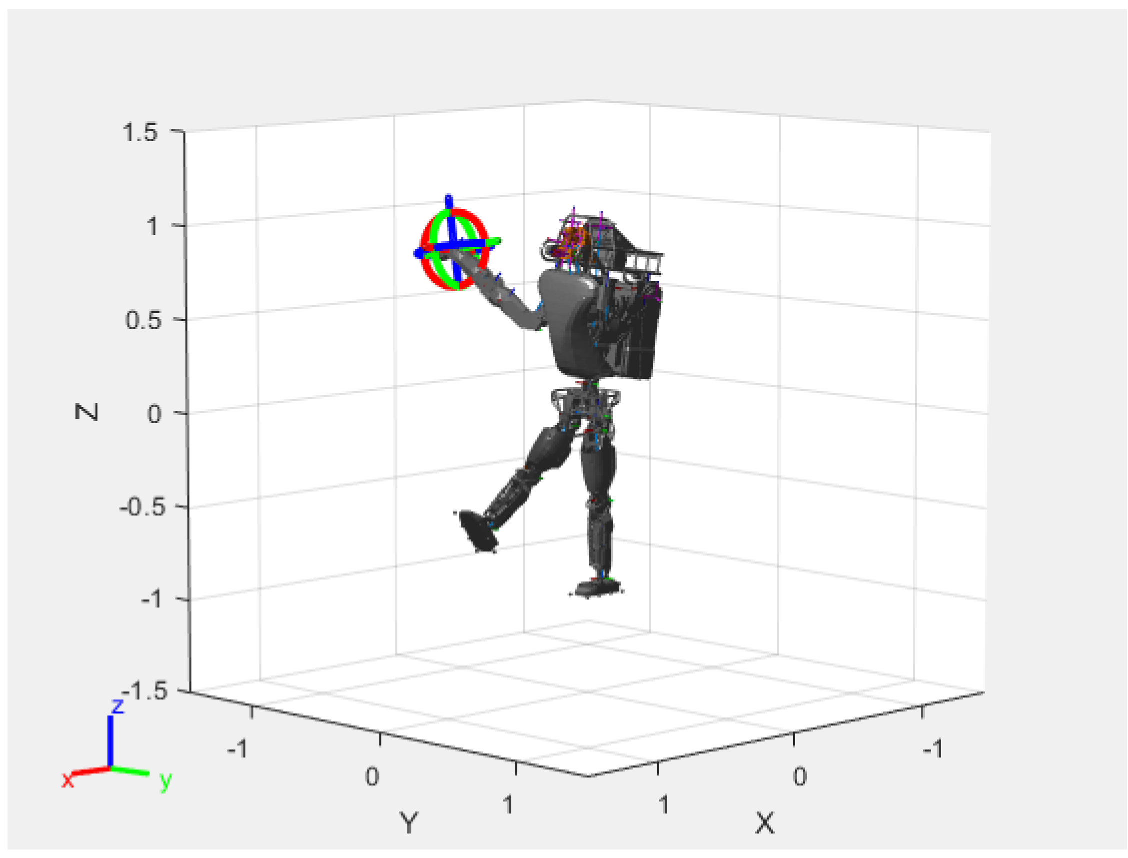
Notice the change in position of the right leg. Save this configuration as well.

```
addConfiguration(viztree)
```

Play Back Configurations

To play back configurations, iterate through the stored configurations index and set the current configuration equal to the stored configuration column vector at each iteration. Because configurations are stored as column vectors, use the second dimension of the matrix.

```
for i = 1:size(viztree.StoredConfigurations,2)
    viztree.Configuration = viztree.StoredConfigurations(:,i);
    pause(0.5)
end
```



Generate Robot Trajectory Using Interactive Rigid Body Tree Model

Use the `interactiveRigidBodyTree` object to visualize a robot model and interactively create waypoints and use them to generate a smooth trajectory using `cubicpolytraj`. For more information, see the `interactiveRigidBodyTree` object and `cubicpolytraj function.`

Load the Robot Model

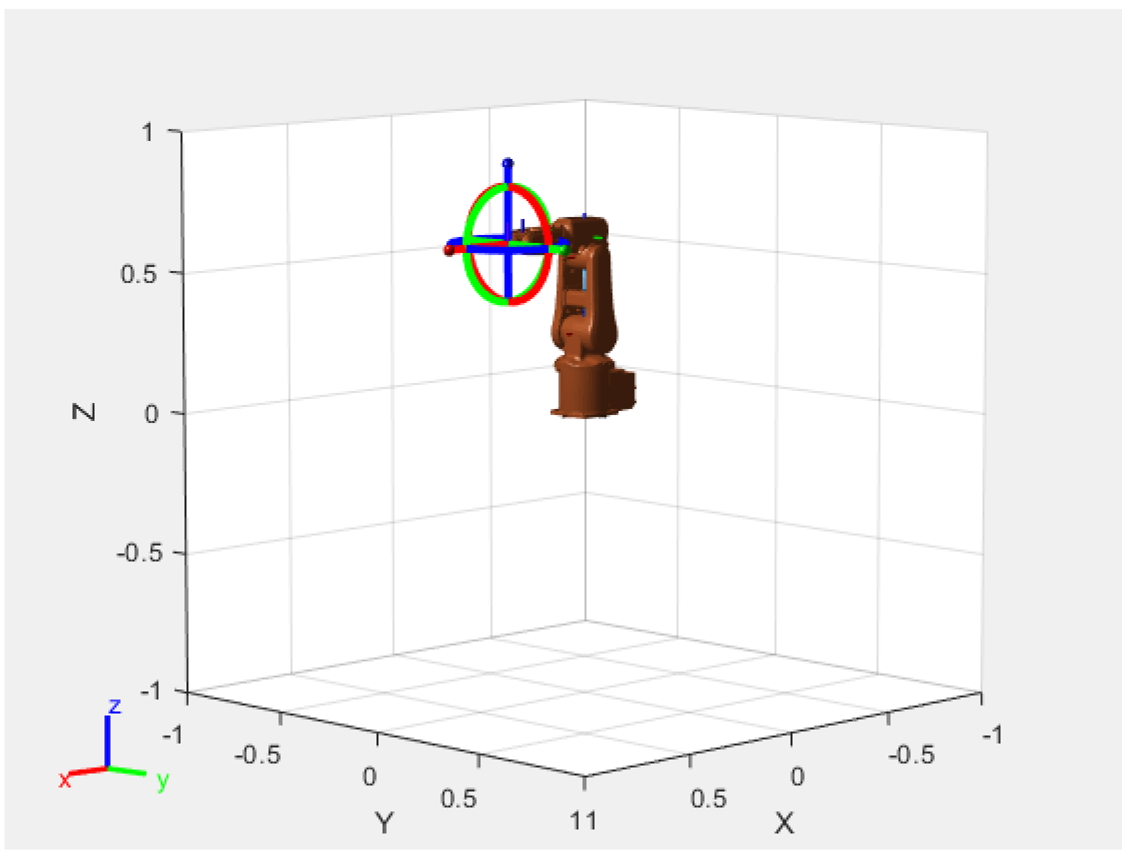
Use the `loadrobot` function to access provided robot models as `rigidBodyTree` objects.

```
robot = loadrobot('abbIrb120');
```

Visualize Robot and Save Configurations

Create an interactive tree object using the `interactiveRigidBodyTree` function. By default, the interactive marker is set to the body with the highest index in the `RigidBodyTree` property. To change this property using the figure, right-click a rigid body and select **Set body as marker body**. Alternatively, `MarkerBodyName` property for the `interactiveRigidBodyTree` can be set using name-value pairs.

```
iRBT = interactiveRigidBodyTree(robot);
```



Interactively Add Configurations

Click and drag the interactive marker to change the robot configuration. You can click and drag any of the axes for linear motion, rotate the body about an axis using the red, green, and blue circles, and drag the center of the interactive marker to position it in 3-D space.

The `interactiveRigidBodyTree` object uses inverse kinematics to determine a configuration that achieves the desired end-effector pose. If the associated rigid body cannot reach the marker, the figure renders the best configuration from the inverse kinematics solver.

When the robot is in a desired configuration use the `addConfiguration` object function to add the configuration to the `StoredConfiguration` property of the object.

In this example, 6 waypoints are created using the interactive marker and `addConfiguration` object function. They are saved in `wayPoints.mat`. Stored configurations can be accessed using `iRBT.StoredConfigurations`.

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

Generate Smooth Trajectory Using the Waypoints

Use the `cubicpolytraj` function to generate smooth trajectory between the waypoints. Define time points that correspond to each waypoint. Define the time vector for generating the trajectory. The `cubicpolyTraj` function generates a configuration for each timestep in the time vector `tvec.`

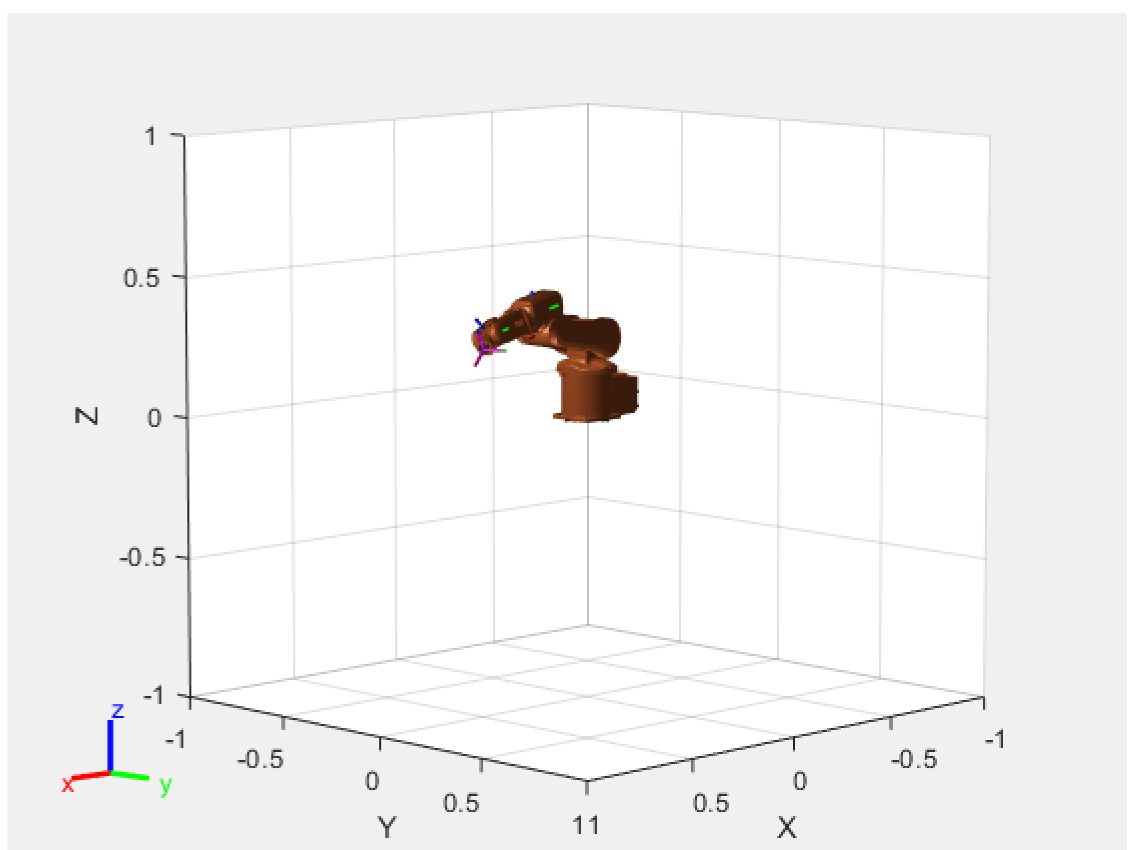
```
iRBT.StoredConfigurations = wayPts ;          % Waypoints
tpts = [0 2 4 6 8 10];                        % Time Points
```

```
tvec = 0:0.1:10;                                          % Time Vector
[q,qd,qdd,pp] = cubicpolytraj(iRBT.StoredConfigurations,tpts,tvec);
```

Visualize Robot Motion on the Trajectory

Define the simulation frequency using a `rateControl` object. Use the `showFigure` function to visualize the robot model and use a `for` loop to play all the configurations of the robot.

```
r = rateControl(10);
iRBT.ShowMarker = false;  % Hide the marker


showFigure(iRBT)

for i = 1:size(q',1)
    iRBT.Configuration = q(:,i);
    waitfor(r);
end
```



Limitations

- If the `interactiveRigidBodyTree` object is deleted while the figure is still open, the interactivity of the figure is disabled and the title of the figure is updated.

Tips

- To maximize performance when visualizing complex robot models with complex meshes, ensure you enable hardware-accelerated OpenGL. By default, MATLAB uses hardware-accelerated OpenGL if your graphics hardware supports it. For more information, see the `opengl` function.

Version History

Introduced in R2020a

See Also

Functions
`loadrobot` | `importrobot` | `homeConfiguration`

Objects
`rigidBodyTree` | `rigidBody` | `rigidBodyJoint` | `generalizedInverseKinematics`

Topics
"Rigid Body Tree Robot Model"
"Plan a Reaching Trajectory With Multiple Kinematic Constraints"
"Trajectory Control Modeling with Inverse Kinematics"

inverseKinematics

Create inverse kinematic solver

Description

The `inverseKinematics` System object creates an inverse kinematic (IK) solver to calculate joint configurations for a desired end-effector pose based on a specified rigid body tree model. Create a rigid body tree model for your robot using the `rigidBodyTree` class. This model defines all the joint constraints that the solver enforces. If a solution is possible, the joint limits specified in the robot model are obeyed.

To specify more constraints besides the end-effector pose, including aiming constraints, position bounds, or orientation targets, consider using `generalizedInverseKinematics`. This object allows you to compute multiconstraint IK solutions.

For closed-form analytical IK solutions, see `analyticalInverseKinematics`.

To compute joint configurations for a desired end-effector pose:

1. Create the `inverseKinematics` object and set its properties.
2. Call the object with arguments, as if it were a function.

To learn more about how System objects work, see What Are System Objects?

Creation

Syntax

```
ik = inverseKinematics
ik = inverseKinematics(Name,Value)
```

Description

`ik = inverseKinematics` creates an inverse kinematic solver. To use the solver, specify a rigid body tree model in the `RigidBodyTree` property.

`ik = inverseKinematics(Name,Value)` creates an inverse kinematic solver with additional options specified by one or more `Name,Value` pair arguments. `Name` is a property name and `Value` is the corresponding value. `Name` must appear inside single quotes (`''`). You can specify several name-value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see System Design in MATLAB Using System Objects.

RigidBodyTree — Rigid body tree model
rigidBodyTree object

Rigid body tree model, specified as a `rigidBodyTree` object. If you modify your rigid body tree model, reassign the rigid body tree to this property. For example:

Create IK solver and specify the rigid body tree.

```
ik = inverseKinematics('RigidBodyTree',rigidbodytree)
```

Modify the rigid body tree model.

```
addBody(rigidbodytree,rigidBody('body1'),'base')
```

Reassign the rigid body tree to the IK solver. If the solver or the `step` function is called before modifying the rigid body tree model, use `release` to allow the property to be changed.

```
ik.RigidBodyTree = rigidbodytree;
```

SolverAlgorithm — Algorithm for solving inverse kinematics
'BFGSGradientProjection' (default) | 'LevenbergMarquardt'

Algorithm for solving inverse kinematics, specified as either `'BFGSGradientProjection'` or `'LevenbergMarquardt'`. For details of each algorithm, see “Inverse Kinematics Algorithms”.

SolverParameters — Parameters associated with algorithm
structure

Parameters associated with the specified algorithm, specified as a structure. The fields in the structure are specific to the algorithm. See “Solver Parameters”.

Usage

Syntax

```
[configSol,solInfo] = ik(endeffector,pose,weights,initialguess)
```

Description

`[configSol,solInfo] = ik(endeffector,pose,weights,initialguess)` finds a joint configuration that achieves the specified end-effector pose. Specify an initial guess for the configuration and your desired weights on the tolerances for the six components of `pose`. Solution information related to execution of the algorithm, `solInfo`, is returned with the joint configuration solution, `configSol`.

Input Arguments

endeffector — End-effector name
character vector

End-effector name, specified as a character vector. The end effector must be a body on the `rigidBodyTree` object specified in the `inverseKinematics` System object.

`pose` — End-effector pose
4-by-4 homogeneous transform

End-effector pose, specified as a 4-by-4 homogeneous transform. This transform defines the desired position and orientation of the rigid body specified in the `endeffector` property.

`weights` — Weight for pose tolerances
six-element vector

Weight for pose tolerances, specified as a six-element vector. The first three elements correspond to the weights on the error in orientation for the desired pose. The last three elements correspond to the weights on the error in *xyz* position for the desired pose.

`initialguess` — Initial guess of robot configuration
structure array | vector

Initial guess of robot configuration, specified as a structure array or vector. Use this initial guess to help guide the solver to a desired robot configuration. The solution is not guaranteed to be close to this initial guess.

To use the vector form, set the `DataFormat` property of the object assigned in the `RigidBodyTree` property to either `'row'` or `'column'` .

Output Arguments

`configSol` — Robot configuration solution
structure array | vector

Robot configuration, returned as a structure array. The structure array contains these fields:

- `JointName` — Character vector for the name of the joint specified in the `RigidBodyTree` robot model
- `JointPosition` — Position of the corresponding joint

This joint configuration is the computed solution that achieves the desired end-effector pose within the solution tolerance.

Note For revolute joints, if the joint limits exceed a range of `2*pi`, where joint position wrapping occurs, then the returned joint position is the one closest to the joint's lower bound.

To use the vector form, set the `DataFormat` property of the object assigned in the `RigidBodyTree` property to either `'row'` or `'column'` .

`solInfo` — Solution information
structure

Solution information, returned as a structure. The solution information structure contains these 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".
- `Status` — Character vector describing whether the solution is within the tolerance (`'success'`) or the best possible solution the algorithm could find (`'best available'`).

Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

Common to All System Objects

step	Run System object algorithm
release	Release resources and allow changes to System object property values and input characteristics
reset	Reset internal states of System object

Examples

Generate Joint Positions to Achieve End-Effector Position

Generate joint positions for a robot model to achieve a desired end-effector position. The `inverseKinematics` system object uses inverse kinematic algorithms to solve for valid joint positions.

Load example robots. The `puma1` robot is a `rigidBodyTree` model of a six-axis robot arm with six revolute joints.

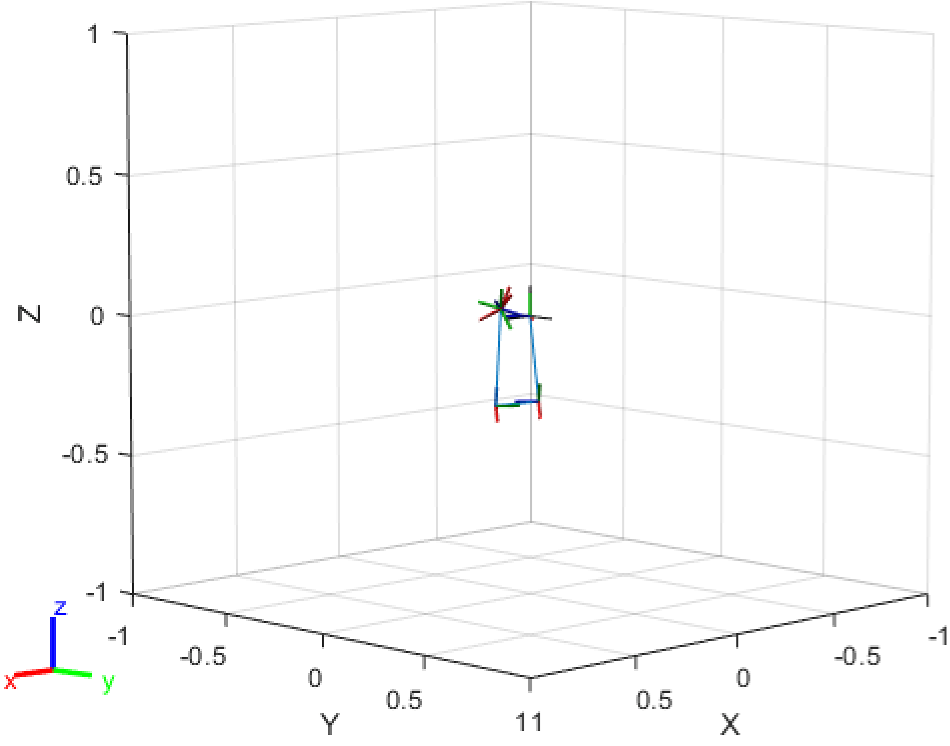
```
load exampleRobots.mat
showdetails(puma1)
```

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

 Idx    Body Name   Joint Name   Joint Type    Parent Name(Idx)   Children Name(s)
 ---    ---------   ----------   ----------    ----------------   ----------------
   1           L1         jnt1     revolute             base(0)   L2(2)
   2           L2         jnt2     revolute               L1(1)   L3(3)
   3           L3         jnt3     revolute               L2(2)   L4(4)
   4           L4         jnt4     revolute               L3(3)   L5(5)
   5           L5         jnt5     revolute               L4(4)   L6(6)
   6           L6         jnt6     revolute               L5(5)
--------------------
```

Generate a random configuration. Get the transformation from the end effector (L6) to the base for that random configuration. Use this transform as a goal pose of the end effector. Show this configuration.

```
randConfig = puma1.randomConfiguration;
tform = getTransform(puma1,randConfig,'L6','base');
```

```
show(puma1,randConfig);
```



Create an `inverseKinematics` object for the `puma1` model. Specify weights for the different components of the pose. Use a lower magnitude weight for the orientation angles than the position components. Use the home configuration of the robot as an initial guess.

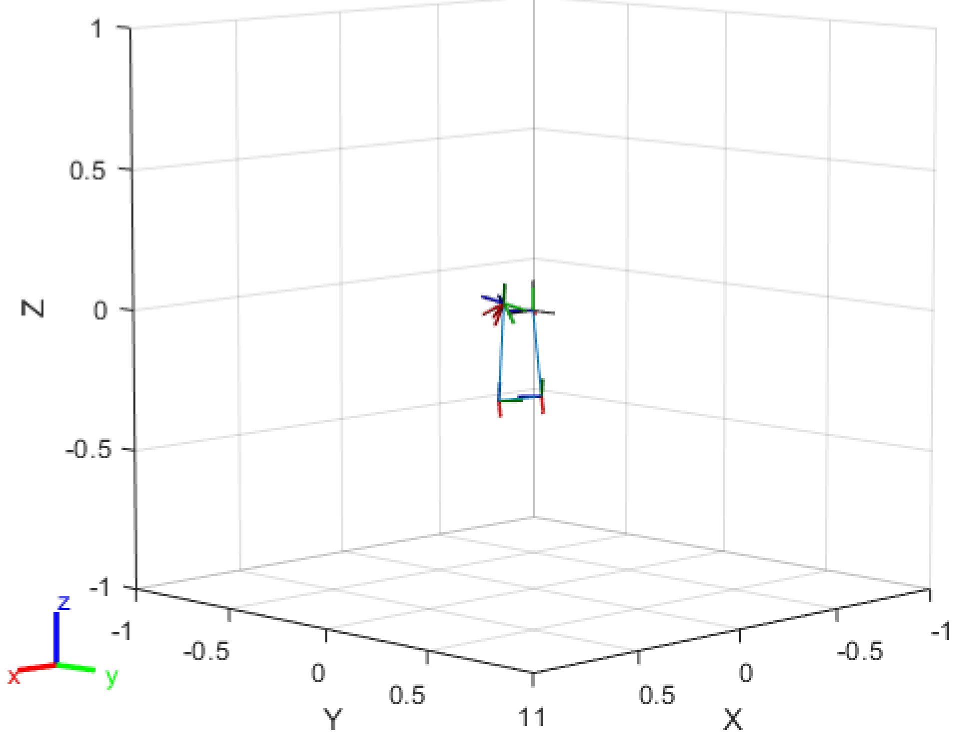
```
ik = inverseKinematics('RigidBodyTree',puma1);
weights = [0.25 0.25 0.25 1 1 1];
initialguess = puma1.homeConfiguration;
```

Calculate the joint positions using the `ik` object.

```
[configSoln,solnInfo] = ik('L6',tform,weights,initialguess);
```

Show the newly generated solution configuration. The solution is a slightly different joint configuration that achieves the same end-effector position. Multiple calls to the `ik` object can give similar or very different joint configurations.

```
show(puma1,configSoln);
```



Version History

Introduced in R2016b

R2019b: `inverseKinematics` was renamed

Behavior change in future release

The `inverseKinematics` object was renamed from `robotics.InverseKinematics`. Use `inverseKinematics` for all object creation.

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.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

When using code generation, you must specify the `RigidBodyTree` property to define the robot on construction of the object. For example:

```
ik = inverseKinematics('RigidBodyTree',robotModel);
```

You also cannot change the `SolverAlgorithm` property after creation. To specify the solver algorithm on creation, use:

```
ik = inverseKinematics('RigidBodyTree',robotModel,...
        'SolverAlgorithm','LevenbergMarquardt');
```

See Also

`analyticalInverseKinematics` | `rigidBodyJoint` | `rigidBody` | `rigidBodyTree` | `generalizedInverseKinematics`

Topics

“Control PR2 Arm Movements Using ROS Actions and Inverse Kinematics”
“Inverse Kinematics Algorithms”

jointSpaceMotionModel

Model rigid body tree motion given joint-space inputs

Description

The `jointSpaceMotionModel` object models the closed-loop joint-space motion of a manipulator robot, specified as a `rigidBodyTree` object. The motion model behavior is defined by the MotionType property.

For more details about the equations of motion, see "Joint-Space Motion Model".

Creation

Syntax

```
motionModel = jointSpaceMotionModel
motionModel = jointSpaceMotionModel("RigidBodyTree",tree)
motionModel = jointSpaceMotionModel(Name,Value)
```

Description

`motionModel = jointSpaceMotionModel` creates a motion model for a default two-joint manipulator.

`motionModel = jointSpaceMotionModel("RigidBodyTree",tree)` creates a motion model for the specified `rigidBodyTree` object.

`motionModel = jointSpaceMotionModel(Name,Value)` sets additional properties specified as name-value pairs. You can specify multiple properties in any order.

Properties

`RigidBodyTree` — Rigid body tree robot model
`rigidBodyTree` object

Rigid body tree robot model, specified as a `rigidBodyTree` object that defines the inertial and kinematic properties of the manipulator.

`NaturalFrequency` — Natural frequency of error dynamics
`[10 10]` (default) | *n*-element vector | scalar

Natural frequency of error dynamics, specified as a scalar or *n*-element vector in Hz, where *n* is the number of nonfixed joints in the associated `rigidBodyTree` object in the RigidBodyTree property.

Dependencies

To use this property, set the MotionType property to `"ComputedTorqueControl"` or `"IndependentJointMotion"`.

`DampingRatio` — Damping ratio of error dynamics

`[1 1]` (default) | *n*-element vector | scalar

Damping ratio of the second-order error dynamics, specified as a scalar or *n*-element vector of real values, where *n* is the number of nonfixed joints in the associated `rigidBodyTree` object in the RigidBodyTree property. If a scalar is specified, then `DampingRatio` becomes an *n*-element vector of value `s`, where `s` is the specified scalar.

Dependencies

To use this property, set the MotionType property to `"ComputedTorqueControl"` or `"IndependentJointMotion"`.

Kp — Proportional gain for PD control

`100*eye(2)` (default) | *n*-by-*n* | scalar

Proportional gain for proportional-derivative (PD) control, specified as a scalar or *n*-by-*n* matrix, where *n* is the number of nonfixed joints in the associated `rigidBodyTree` object in the RigidBodyTree property. You must set the MotionType property to `"PDControl"`. If a scalar is specified, then `Kp` becomes `s*eye(n)`, where `s` is the specified scalar.

Dependencies

To use this property, set the MotionType property to `"PDControl"`.

Kd — Derivative gain for PD control

`10*eye(2)` (default) | *n*-by-*n* | scalar

Derivative gain for PD control, specified as a scalar or *n*-by-*n* matrix, where *n* in the number of nonfixed joints in the `rigidBodyTree` object in the RigidBodyTree property. If a scalar is specified, then `Kp` becomes `s*eye(n)`, where `s` is the specified scalar.

Dependencies

To use this property, set the MotionType property to `"PDControl"`.

`MotionType` — Type of motion computed by the motion model

`"ComputedTorqueControl"` (default) | `"IndependentJointMotion"` | `"PDControl"`

Type of motion, specified as a string scalar or character vector that defines the closed-loop joint-space behavior that the object models. Options are:

- `"ComputedTorqueControl"` — Compensates for full-body dynamics and assigns the error dynamics specified in the NaturalFrequency and DampingRatio properties.
- `"IndependentJointMotion"` — Models each joint as an independent second-order system using the error dynamics specified by the `NaturalFrequency` and `DampingRatio` properties.
- `"PDControl"` — Uses proportional-derivative control on the joints based on the specified Kp and Kd properties.

Object Functions

derivative	Time derivative of manipulator model states
updateErrorDynamicsFromStep	Update values of NaturalFrequency and DampingRatio properties given desired step response

Examples

Create Joint-Space Motion Model

This example shows how to create and use a `jointSpaceMotionModel` object for a manipulator robot in joint-space.

Create the Robot

```
robot = loadrobot("kinovaGen3","DataFormat","column","Gravity",[0 0 -9.81]);
```

Set Up the Simulation

Set the timespan to be 1 s with a timestep size of 0.01 s. Set the initial state to be the robots, home configuration with a velocity of zero.

```
tspan = 0:0.01:1;
initialState = [homeConfiguration(robot); zeros(7,1)];
```

Define the a reference state with a target position, zero velocity, and zero acceleration.

```
targetState = [pi/4; pi/3; pi/2; -pi/3; pi/4; -pi/4; 3*pi/4; zeros(7,1); zeros(7,1)];
```

Create the Motion Model

Model the system with computed torque control and error dynamics defined by a moderately fast step response with 5% overshoot.

```
motionModel = jointSpaceMotionModel("RigidBodyTree",robot);
updateErrorDynamicsFromStep(motionModel,.3,.05);
```

Simulate the Robot

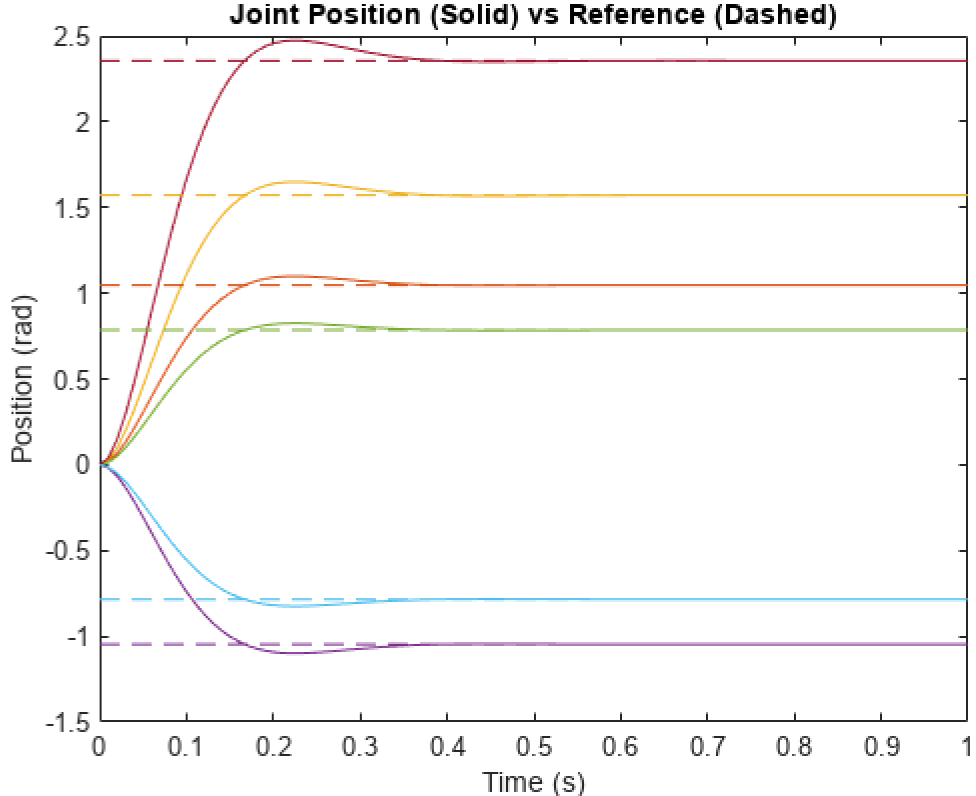
Use the derivative function of the model as the input to the `ode45` solver to simulate the behavior over 1 second.

```
[t,robotState] = ode45(@(t,state)derivative(motionModel,state,targetState),tspan,initialState);
```

Plot the Response

Plot the positions of all the joints actuating to their target state. Joints with a higher displacement between the starting position and the target position actuate to the target at a faster rate than those with a lower displacement. This leads to an overshoot, but all of the joints have the same settling time.

```
figure
plot(t,robotState(:,1:motionModel.NumJoints));
hold all;
plot(t,targetState(1:motionModel.NumJoints)*ones(1,length(t)),"--");
title("Joint Position (Solid) vs Reference (Dashed)");
xlabel("Time (s)")
ylabel("Position (rad)");
```



Version History

Introduced in R2019b

References

[1] Craig, John J. *Introduction to Robotics: Mechanics and Control*. Upper Saddle River, NJ: Pearson Education, 2005.

[2] Spong, Mark W., Seth Hutchinson, and Mathukumalli Vidyasagar. *Robot Modeling and Control*. Hoboken, NJ: Wiley, 2006.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

Classes

`taskSpaceMotionModel`

Blocks

Joint Space Motion Model

Functions
`derivative` | `updateErrorDynamicsFromStep`

Topics
"Simulate Joint-Space Trajectory Tracking in MATLAB"
"Plan and Execute Task- and Joint-Space Trajectories Using KINOVA Gen3 Manipulator"

lidarScan

Create object for storing 2-D lidar scan

Description

A `lidarScan` object contains data for a single 2-D lidar (light detection and ranging) scan. The lidar scan is a laser scan for a 2-D plane with distances (`Ranges`) measured from the sensor to obstacles in the environment at specific angles (`Angles`). Use this laser scan object as an input to other robotics algorithms such as `matchScans`, `controllerVFH`, or `monteCarloLocalization`.

Creation

Syntax

```
scan = lidarScan(ranges,angles)
scan = lidarScan(cart)
```

Description

`scan = lidarScan(ranges,angles)` creates a `lidarScan` object from the `ranges` and `angles`, that represent the data collected from a lidar sensor. The `ranges` and `angles` inputs are vectors of the same length and are set directly to the `Ranges` and `Angles` properties.

`scan = lidarScan(cart)` creates a `lidarScan` object using the input Cartesian coordinates as an *n*-by-2 matrix. The `Cartesian` property is set directly from this input.

`scan = lidarScan(scanMsg)` creates a `lidarScan` object from a `LaserScan` ROS message object.

Properties

`Ranges` — Range readings from lidar in meters
vector

Range readings from lidar, specified as a vector in meters. This vector is the same length as `Angles`, and the vector elements are measured in meters.

Data Types: `single` | `double`

`Angles` — Angle of readings from lidar in radians
vector

Angle of range readings from lidar, specified as a vector. This vector is the same length as `Ranges`, and the vector elements are measured in radians. Angles are measured counter-clockwise around the positive *z*-axis.

Data Types: `single` | `double`

`Cartesian` — Cartesian coordinates of lidar readings in meters
`[x y]` matrix

Cartesian coordinates of lidar readings, returned as an `[x y]` matrix. In the lidar coordinate frame, positive *x* is forward and positive *y* is to the left.

Data Types: `single` | `double`

`Count` — Number of lidar readings
scalar

Number of lidar readings, returned as a scalar. This scalar is also equal to the length of the `Ranges` and `Angles` vectors or the number of rows in `Cartesian`.

Data Types: `double`

Object Functions

plot	Display laser or lidar scan readings
removeInvalidData	Remove invalid range and angle data
transformScan	Transform laser scan based on relative pose

Examples

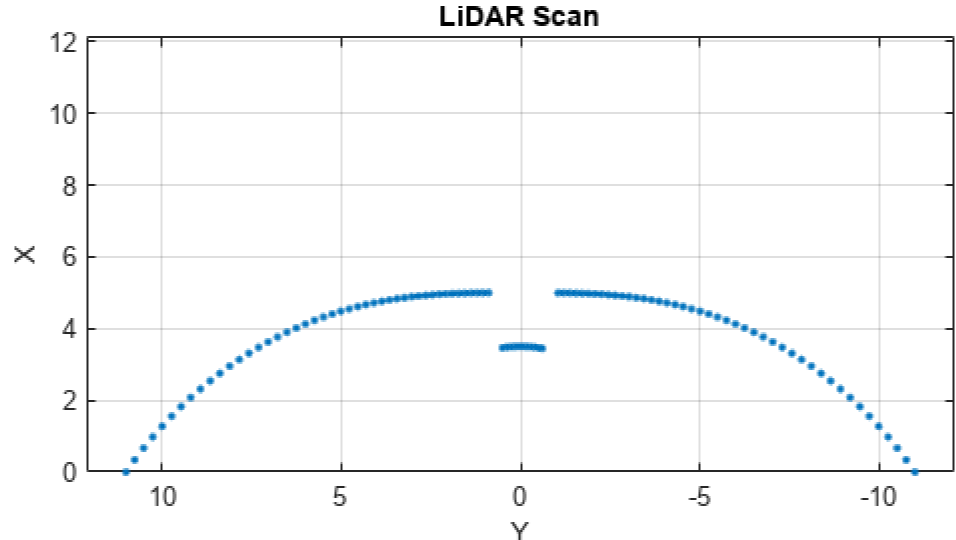
Plot Lidar Scan and Remove Invalid Points

Specify lidar data as vectors of ranges and angles. These values include readings outside of the sensors range.

```
x = linspace(-2,2);
ranges = abs((1.5).*x.^2 + 5);
ranges(45:55) = 3.5;
angles = linspace(-pi/2,pi/2,numel(ranges));
```

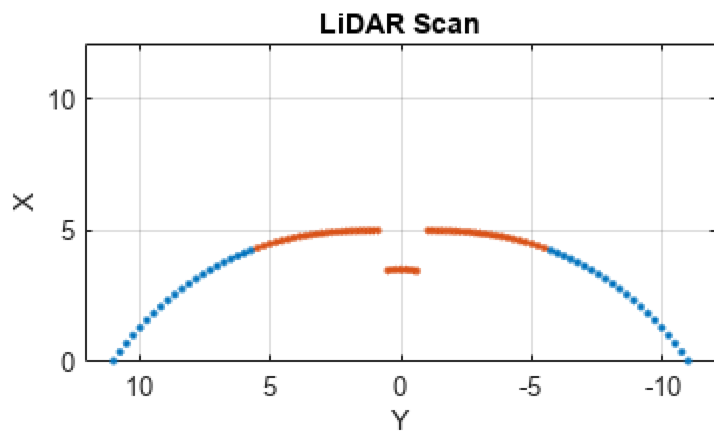
Create a lidar scan by specifying the ranges and angles. Plot all points of the lidar scan.

```
scan = lidarScan(ranges,angles);
plot(scan)
```



Remove invalid points based on a specified minimum and maximum range.

```
minRange = 0.1;
maxRange = 7;
scan2 = removeInvalidData(scan,'RangeLimits',[minRange maxRange]);
hold on
plot(scan2)
legend('All Points','Valid Points')
```



Transform Laser Scans

Create a `lidarScan` object. Specify the ranges and angles as vectors.

```
refRanges = 5*ones(1,300);
refAngles = linspace(-pi/2,pi/2,300);
refScan = lidarScan(refRanges,refAngles);
```

Translate the laser scan by an `[x y]` offset of `(0.5,0.2)`.

```
transformedScan = transformScan(refScan,[0.5 0.2 0]);
```

Rotate the laser scan by 20 degrees.

```
rotateScan = transformScan(refScan,[0,0,deg2rad(20)]);
```

Version History

Introduced in R2019b

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

Lidar scans require a limited size in code generation. The lidar scans are limited to 4000 points (range and angles) as a maximum.

See Also
`transformScan`

manipulatorCHOMP

Covariant Hamiltonian optimizer for rigid body tree motion planning

Description

The `manipulatorCHOMP` object optimizes rigid body tree motions using the Covariant Hamiltonian Optimization for Motion Planning (CHOMP) algorithm. CHOMP optimizes trajectories for smoothness and collision avoidance by minimizing a cost function comprised of a smoothness cost and collision cost.

To specify options for smoothness cost, collision cost, and the solver, use the `chompSmoothnessOptions`, `chompCollisionOptions`, and `chompSolverOptions` objects, respectively.

Creation

Syntax

```
manipCHOMP = manipulatorCHOMP(robotRBT)
manipCHOMP = manipulatorCHOMP(robotRBT,Name=Value)
```

Description

`manipCHOMP = manipulatorCHOMP(robotRBT)` creates a CHOMP-based optimizer for a rigid body tree `robotRBT`. The `robotRBT` argument sets the `RigidBodyTree` property.

`manipCHOMP = manipulatorCHOMP(robotRBT,Name=Value)` specifies properties using one or more name-value arguments.

Properties

`SmoothnessOptions` — Smoothness cost options
`chompSmoothnessOptions` object

Smoothness cost options, specified as a `chompSmoothnessOptions` object.

By default, this property contains a `chompSmoothnessOptions` object with default property values.

`CollisionOptions` — Collision cost options
`chompCollisionOptions` object

Collision cost options, specified as a `chompCollisionOptions` object.

By default, this property contains a `chompCollisionOptions` object with default property values.

`SolverOptions` — Motion-planning solver options
`chompSolverOptions` object

Motion-planning solver options, specified as a `chompSolverOptions` object.

By default, this property contains a `chompSolverOptions` object with default property values.

RigidBodyTree — Robot model used for motion planning
`rigidBodyTree` object

This property is read-only.

Robot model used for motion planning, specified as a `rigidBodyTree` object at object construction.

You can access the spherical approximation in `RigidBodyTreeSpheres` as collision geometries on the tree by using the `RigidBodyTree` property.

RigidBodyTreeSpheres — Spheres of bodies in rigid body tree
table

Spheres of the bodies in the rigid body tree, stored as a table. The table contains two columns. The first column contains the name of a rigid body in `RigidBodyTree` and the second column contains a corresponding cell array. Each cell array contains a 4-by-*N* matrix as the only element in the cell array, where *N* is the number of spheres for the corresponding rigid body. Each column of the matrix of spheres contains the information for a sphere in the form `[r; x; y; z]`, defined with respect to the frame of the rigid body. `r` is the radius of the sphere, and `x`, `y`, and `z` are the `x`-, `y`-, and `z`-coordinates of the center of the sphere, respectively.

SphericalObstacles — Spherical obstacles in environment
[ ] (default) | 4-by-*N* matrix

Spherical obstacles in the environment, specified as a 4-by-*N* matrix. Each column represents the information about each sphere in the form `[r; x; y; z]`, defined with respect to the base frame of the robot. `r` is the radius of the sphere, and `x`, `y`, and `z` are the `x`-, `y`-, and `z`-coordinates of the center of the sphere, respectively.

Object Functions

optimize — Optimize trajectory using CHOMP
show — Visualize CHOMP trajectory of rigid body tree

Examples

Optimize Collision-Free Trajectory with CHOMP

Load a robot model into the workspace, and create a CHOMP solver.

```
robot = loadrobot("kinovaGen3",DataFormat="row");
chomp = manipulatorCHOMP(robot);
```

Create spheres to represent obstacles, and add them to the CHOMP solver.

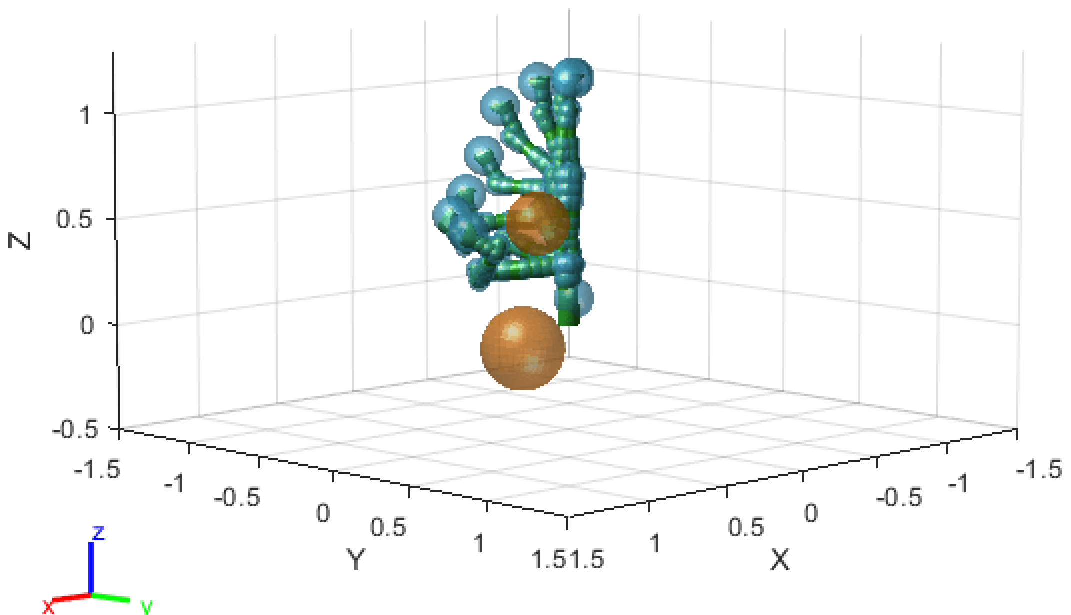
```
env = [0.20 0.2 -0.1 -0.1;   % sphere, radius 0.20 at (0.2,-0.1,-0.1)
       0.15 0.2  0.0  0.5]'; % sphere, radius 0.15 at (0.2,0.0,0.5)
chomp.SphericalObstacles = env;
```

To prioritize a collision-free trajectory, set the smoothness cost weight to a lower value than the collision cost weight. Then add the options to the CHOMP solver.

```
chomp.SmoothnessOptions = chompSmoothnessOptions(SmoothnessCostWeight=1e-3);
chomp.CollisionOptions = chompCollisionOptions(CollisionCostWeight=10);
chomp.SolverOptions = chompSolverOptions(Verbosity="none",LearningRate=7.0);
```

Initialize a trajectory, optimize it using the CHOMP solver, and show the waypoints in a figure.

```
startconfig = homeConfiguration(robot);
goalconfig = [0.5 1.75 -2.25 2.0 0.3 -1.65 -0.4];
timepoints = [0 5];
timestep = 0.1;
trajtype = "minjerkpolytraj";
[wptsamples,tsamples] = optimize(chomp, ...
    [startconfig; goalconfig], ...
    timepoints, ...
    timestep, ...
    InitialTrajectoryFitType=trajtype);
show(chomp,wptsamples,NumSamples=10);
zlim([-0.5 1.3])
```



Version History

Introduced in R2023a

References

[1] Ratliff, Nathan, Siddhartha Srinivasa, Matt Zucker, and Andrew Bagnell. “CHOMP: Gradient Optimization Techniques for Efficient Motion Planning.” In *2009 IEEE International Conference on Robotics and Automation*, 489–94. Kobe, Japan: IEEE, 2009. https://doi.org/10.1109/ROBOT.2009.5152817.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also
`chompSolverOptions` | `chompSmoothnessOptions` | `chompCollisionOptions`

Topics
“Pick-And-Place Workflow Using CHOMP for Manipulators”

manipulatorCollisionBodyValidator

Validate states for collision bodies of rigid body tree

Description

The `manipulatorCollisionBodyValidator` object performs state and motion validity checks for a rigid body tree robot model. To check if the collision bodies collide either with other bodies (self-collisions) or the environment, use the `isStateValid` object function. To check if a motion between two states is valid, use the `isMotionValid` object function.

Creation

Syntax

```
manipSV = manipulatorCollisionBodyValidator
manipSV = manipulatorCollisionBodyValidator(ss)
manipSV = manipulatorCollisionBodyValidator(ss,Name=Value)
```

Description

`manipSV = manipulatorCollisionBodyValidator` creates a state validator with default values for a `manipulatorStateSpace` object.

`manipSV = manipulatorCollisionBodyValidator(ss)` creates a state validator for a `manipulatorStateSpace` object that represents a robot model state space and contains collision bodies for rigid body elements. Specify `ss` as a `manipulatorStateSpace` object.

`manipSV = manipulatorCollisionBodyValidator(ss,Name=Value)` specifies “Properties” on page 1-244 as name-value arguments

Properties

`ValidationDistance` — Distance resolution for motion validation
`0.1` (default) | positive scalar in meters

Distance resolution for motion validation, specified as a positive scalar. The validation distance determines the number of interpolated states between states specified to the `isMotionValid` object function. The object function validates each interpolated state by checking for collisions with the robot and the environment.

Data Types: `double`

`IgnoreSelfCollision` — Ignore self collisions toggle
`0` or `false` (default) | `1` or `true`

Ignore self collisions toggle, specified as a logical. If this property is set to `true`, the `isMotionValid` object function skips checking between bodies for collisions and only compares the

bodies to the environment. Not checking for self-collisions can improve the speed of the planning phase, but your state space should contain joint limits that ensure self-collisions are not possible.

Data Types: `logical`

`Environment` — Collision objects in robot environment
`{}` (default) | cell array of collision body objects

Collision objects in the robot environment, specified as a cell array of collision objects of these types:

- `collisionBox`
- `collisionCylinder`
- `collisionMesh`
- `collisionSphere`

```
box = collisionBox(0.1,0.1,0.5); % XYZ Lengths
box.Pose = trvec2tform([0.2 0.2 0.5]); % XYZ Position
sphere = collisionSphere(0.25); % Radius
sphere.Pose = trvec2tform([-0.2 -0.2 0.5]); % XYZ Position
env = {box sphere};
manipSS = manipulatorCollisionBodyValidator(ss,Environment=env);
```

Data Types: `logical`

`StateSpace` — Manipulator state space
`manipulatorStateSpace` object

Manipulator state space, specified as a `manipulatorStateSpace` object, which is a subclass of `nav.StateSpace`.

`SkippedSelfCollisions` — Body pairs skipped for checking self-collisions
`"parent"` (default) | `"adjacent"`

Body pairs skipped for checking self-collisions, specified as either `"parent"` or `"adjacent"`:

- `"parent"` — Skip collision checking between child and parent bodies.
- `"adjacent"` — Skip collision checking between bodies on adjacent indices.

See “Change Self-Collision Checking Behavior” on page 3-365 for more information.

Data Types: `char` | `string`

Object Functions

isStateValid	Check if state is valid
isMotionValid	Check if path between states is valid

Examples

Validate State and Motion Manipulator State Space

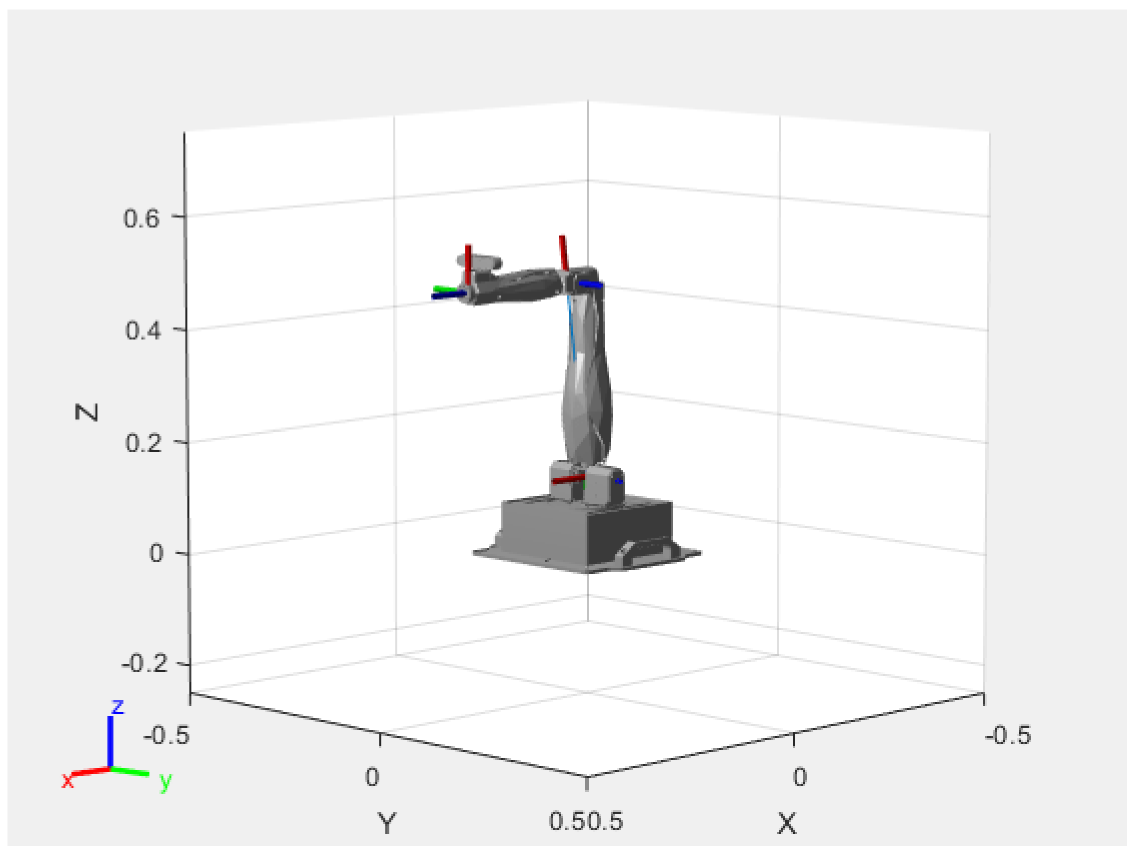
Generate states to form a path, validate motion between states, and check for self-collisions and environmental collisions with objects in your world. The `manipulatorStateSpace` object represents the joint configuration space of your rigid body tree robot model, and can sample states,

calculate distances, and enforce state bounds. The `manipulatorCollisionBodyValidator` object validates the state and motion based on the collision bodies in your robot model and any obstacles in your environment.

Load Robot Model

Use the `loadrobot` function to access predefined robot models. This example uses the Quanser QArm™ robot and joint configurations are specified as row vectors.

```
robot = loadrobot("quanserQArm",DataFormat="row");
figure(Visible="on")
show(robot);
xlim([-0.5 0.5])
ylim([-0.5 0.5])
zlim([-0.25 0.75])
hold on
```



Configure State Space and State Validation

Create the state space and state validator from the robot model.

```
ss = manipulatorStateSpace(robot);
sv = manipulatorCollisionBodyValidator(ss,SkippedSelfCollisions="parent");
```

Set the validation distance to `0.05`, which is based on the distance normal between two states. You can configure the validator to ignore self collisions to improve the speed of validation, but must consider whether your robot model has the proper joint limit settings set to ensure it does not collide with itself.

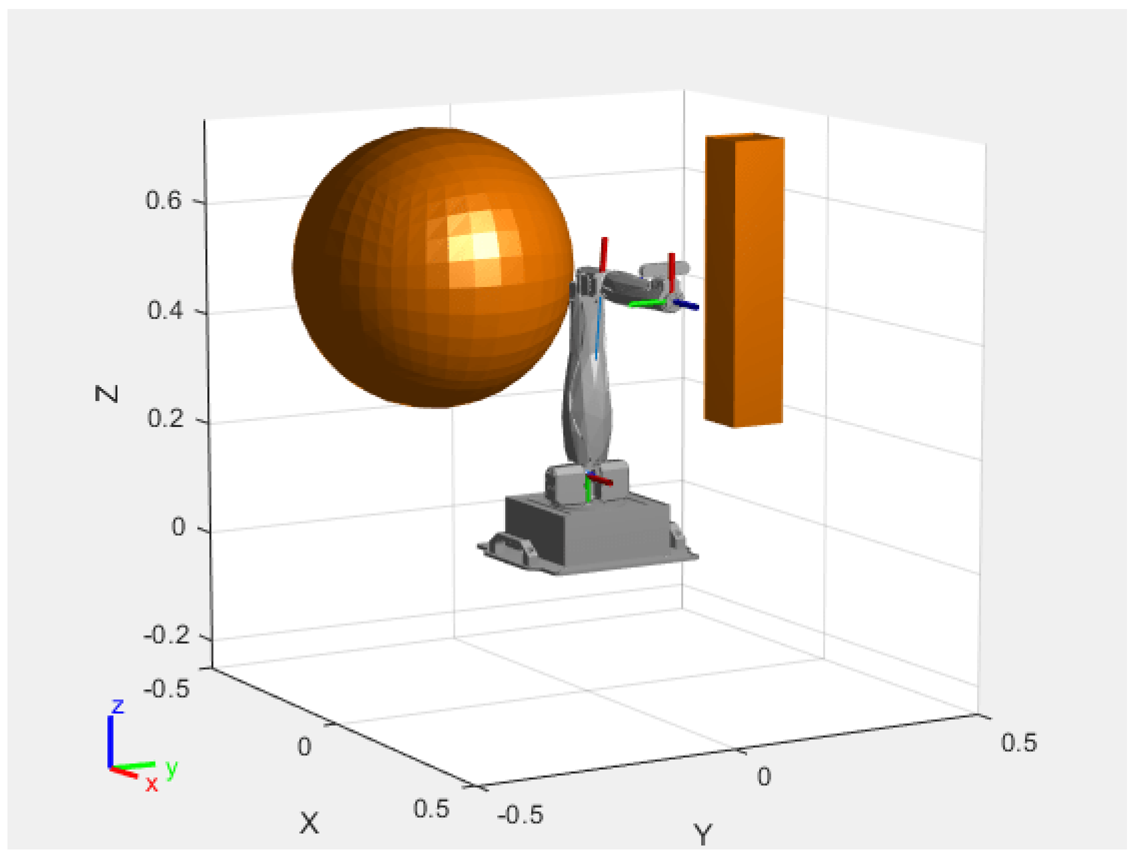
```
sv.ValidationDistance = 0.05;
sv.IgnoreSelfCollision = true;
```

Place collision objects in the robot environment. Set the `Environment` property of the collision validator object using a cell array of objects.

```
box = collisionBox(0.1,0.1,0.5); % XYZ Lengths
box.Pose = trvec2tform([0.2 0.2 0.5]); % XYZ Position
sphere = collisionSphere(0.25); % Radius
sphere.Pose = trvec2tform([-0.2 -0.2 0.5]); % XYZ Position
env = {box sphere};
sv.Environment = env;
```

Visualize the environment.

```
for i = 1:length(env)
    show(env{i})
end
view(60,10)
```



Plan Path

Start with the home configuration as the first point on the path.

```
rng(0); % Repeatable results
start = homeConfiguration(robot);
path = start;
idx = 1;
```

Plan a path using these steps, in a loop:

- Sample a nearby goal configuration, using the Gaussian distribution, by specifying the standard deviation for each joint angle.
- Check if the sampled goal state is valid.
- If the sampled goal state is valid, check if the motion between states is valid and, if so, add it to the path.

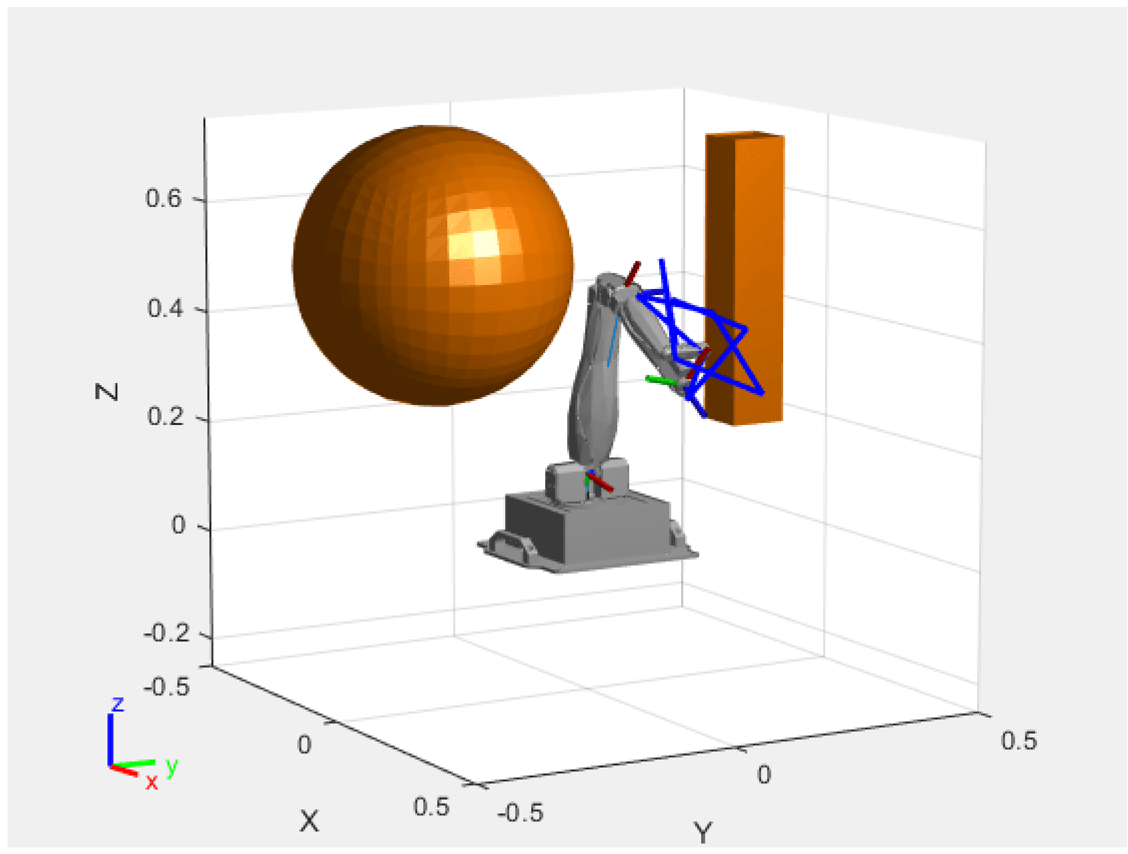
```
for i = 2:25
    goal = sampleGaussian(ss,start,0.25*ones(4,1));
    validState = isStateValid(sv,goal);

    if validState % If state is valid, check motion between states.
        [validMotion,~] = isMotionValid(sv,path(idx,:),goal);

        if validMotion % If motion is valid, add to path.
            path = [path; goal];
            idx = idx + 1;
        end
    end
end
```

Visualize Path

After generating the path of valid motions, visualize the robot motion. Because you sampled random states near the home configuration, you should see the arm move around that initial configuration.

To visualize the path of the end effector in 3-D, get the transformation, relative to the base world frame at each point. Store the points as an *xyz* translation vector. Plot the path of the end effector.

```
eePose = nan(3,size(path,1));

for i = 1:size(path,1)
    show(robot,path(i,:),PreservePlot=false);
    eePos(i,:) = tform2trvec(getTransform(robot,path(i,:),"END-EFFECTOR")); % XYZ translation vec
    plot3(eePos(:,1),eePos(:,2),eePos(:,3),"-b",LineWidth=2)
    drawnow
end
```



Version History

Introduced in R2021b

R2022b: Alter rigid body tree self-collision checking behavior change and new default self-collision checking behavior

Behavior change in future release

You can now specify self-collision checking behavior for a rigid body tree robot model by using the `SkippedSelfCollisions` property. Specify `SkippedSelfCollisions` as `"parent"` or `"adjacent"`:

- `"parent"` — Collision checking ignores self-collisions between parent and child rigid bodies.
- `"adjacent"` — Collision checking ignores self -collisions between rigid bodies of adjacent indices.

As of R2022b, the default behavior of collision checking is to ignore self-collisions between parent and child rigid bodies. In previous releases, the default behavior of self-collision checking was to ignore self-collisions between adjacent rigid bodies. To instead ignore self-collisions between rigid bodies of adjacent indices, specify `SkippedSelfCollisions` as `"adjacent"`.

See “Change Self-Collision Checking Behavior” on page 3-365 for more information.

See Also

Objects
`rigidBodyTree` | `manipulatorStateSpace` | `workspaceGoalRegion` | `manipulatorRRT`

Functions
`isStateValid` | `isMotionValid` | `sampleUniform` | `sampleGaussian` | `interpolate` | `distance` | `enforceStateBounds`

manipulatorRRT

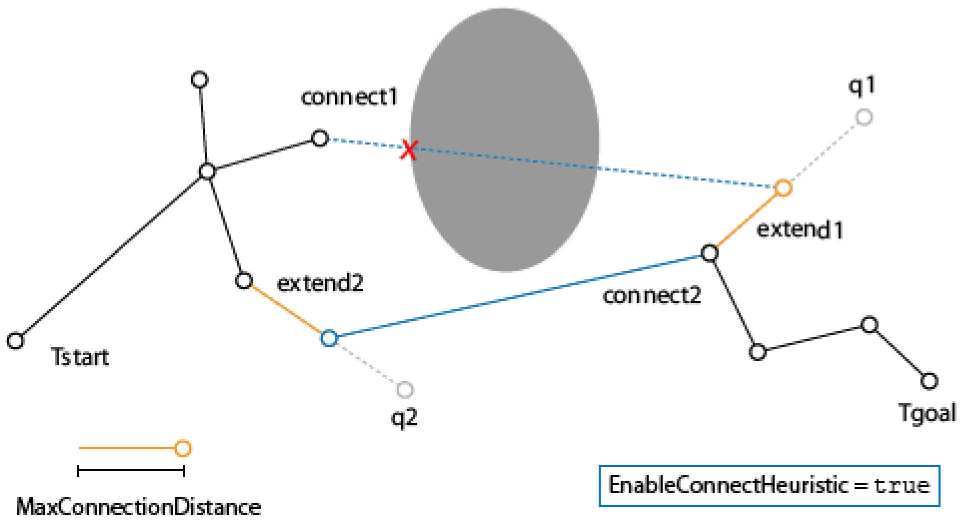
Plan motion for rigid body tree using bidirectional RRT

Description

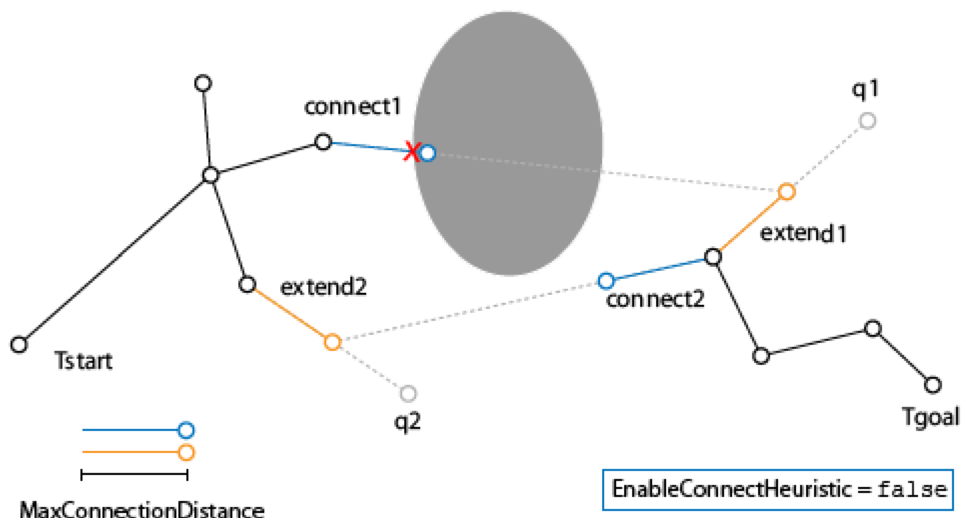
The `manipulatorRRT` object is a single-query planner for manipulator arms that uses the bidirectional rapidly exploring random trees (RRT) algorithm with an optional connect heuristic to potentially increase speed.

The bidirectional RRT planner creates two trees with root nodes at the specified start and goal configurations. To extend each tree, the planner generates a random configuration and, if valid, takes a step from the nearest node based on the MaxConnectionDistance property. After each extension, the planner attempts to connect between the two trees using the new extension and the closest node on the opposite tree. Invalid configurations or connections that collide with the environment are not added to the tree.

For a greedier search, enabling the EnableConnectHeuristic property disables the limit on the `MaxConnectionDistance` property when connecting between the two trees.



Setting the `EnableConnectHueristic` property to `false` limits the extension distance when connecting between the two trees to the value of the `MaxConnectionDistance` property.



The object uses a `rigidBodyTree` robot model to generate the random configurations and intermediate states between nodes. Collision objects are specified in the robot model to validate the configurations and check for collisions with the environment or the robot itself.

To plan a path between a start and a goal configuration, use the `plan` object function. After planning, you can interpolate states along the path using the `interpolate` object function. To attempt to shorten the path by trimming edges, use the `shorten` object function.

To specify a region to sample end-effector poses near the goal configuration, create a `workspaceGoalRegion` object and specify it as the `goalRegion` input to the `plan` object function. To change the probability of sampling additional goal configurations, specify the WorkspaceGoalRegionBias property.

For more information about the computational complexity, see Planning Complexity on page 1-262.

Creation

Syntax

```
rrt = manipulatorRRT(robotRBT,{})
rrt = manipulatorRRT(robotRBT,collisionObjects)
rrt = manipulatorRRT( ___ ,Map=map)
```

Description

`rrt = manipulatorRRT(robotRBT,{})` creates a bidirectional RRT planner for the specified `rigidBodyTree` robot model. The empty cell array indicates that there are no obstacles in the environment.

`rrt = manipulatorRRT(robotRBT,collisionObjects)` creates a planner for a robot model with collision objects placed in the environment. The planner checks for collisions with these objects.

rrt = manipulatorRRT(___ ,Map=map) specifies a 3-D occupancy map map to represent the environment. This requires Navigation Toolbox™.

Input Arguments

map — 3-D occupancy map representing environment
occupancyMap3D object

3-D occupancy map representing the environment, specified as a occupancyMap3D object. Note that the manipulatorRRT object does not deep copy the specified map and instead holds the handle to the specified map.

Specifying this input argument requires Navigation Toolbox.

Properties

MaxConnectionDistance — Maximum length between planned configurations
0.1 (default) | positive scalar

Maximum length between planned configurations, specified as a positive scalar. The object computes the length of the motion as the Euclidean distance between the two joint configurations. During the extension process, this is the maximum distance a configuration can change.

When revolute joints have infinite limits, differences between two joint positions are calculated using the angdiff function.

If the EnableConnectheuristic property is set to true, the object ignores this distance when connecting the two trees during the connect stage.

Data Types: double

ValidationDistance — Distance resolution for validating motion between configurations
0.01 (default) | positive scalar

Distance resolution for validating motion between configurations, specified as a positive scalar. The validation distance determines the number of interpolated nodes between two adjacent nodes in the tree. The object validates each interpolated node by checking for collisions with the robot and the environment.

Data Types: double

MaxIterations — Maximum number of random configurations generated
10000 (default) | positive integer

Maximum number of random configurations generated, specified as a positive integer.

Data Types: double

EnableConnectHeuristic — Directly join trees during connect phase
true or 1 (default) | false or 0

Directly join trees during the connect phase of the planner, specified as a logical 1 (true) or 0 (false). Setting this property to true causes the object to ignore the MaxConnectionDistance property when attempting to connect the two trees together.

Data Types: logical

`WorkspaceGoalRegionBias` — Probability to sample additional goal state from workspace goal region
0.50 (default) | positive value in the range [0,1)

Probability to sample a goal state from the workspace goal region, specified as a positive value in the range [0,1). The bias defines the probability to add additional goal states to the tree from the `workspaceGoalRegion` object. When this value is set to zero, the `workspaceGoalRegion` object still samples a single goal for the planner to plan to.

Increasing this value increases the likelihood of reaching a goal state in the goal region, but may lead to longer planning times because each new goal state adds additional complexity for planning.

Dependency

You must use the `goalRegion` input when calling the `plan` object function.

Data Types: `double`

`IgnoreSelfCollision` — Ignore self collisions during planning
`0` or `false` (default) | `1` or `true`

Ignore self collisions during planning, specified as a logical. If this property is set to `true`, the `plan` function skips checking between bodies for collisions and only compares the bodies to the environment. By not checking for self-collisions, you may improve the speed of the planning phase.

Data Types: `logical`

`SkippedSelfCollisions` — Body pairs skipped for checking self-collisions
`"parent"` (default) | `"adjacent"`

Body pairs skipped for checking self-collisions, specified as either `"parent"` or `"adjacent"`:

- `"parent"` — Skip collision checking between child and parent bodies.
- `"adjacent"` — Skip collision checking between bodies on adjacent indices.

See “Change Self-Collision Checking Behavior” on page 3-365 for more information.

Data Types: `char` | `string`

Object Functions

plan	Plan path using RRT for manipulators
interpolate	Interpolate states along path from RRT
shorten	Trim edges to shorten path from RRT

Examples

Plan Path for Manipulator Robot Using RRT

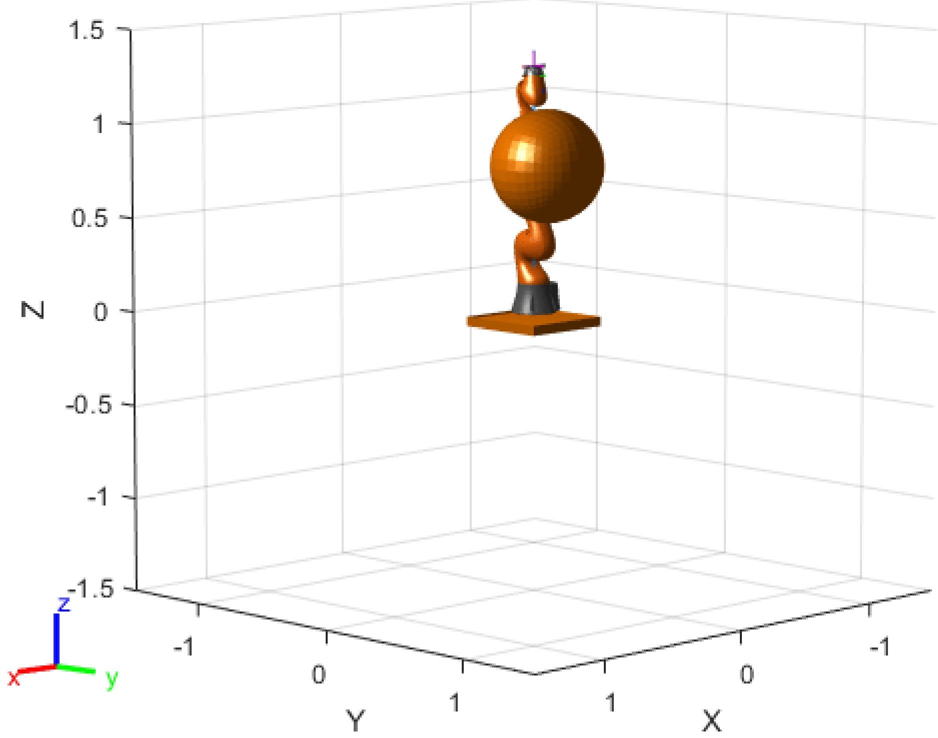
Use the `manipulatorRRT` object to plan a path for a rigid body tree robot model in an environment with obstacles. Visualize the planned path with interpolated states.

Load a robot model into the workspace. Use the KUKA LBR iiwa© manipulator arm.

```
robot = loadrobot("kukaIiwa14","DataFormat","row");
```

Generate the environment for the robot. Create collision objects and specify their poses relative to the robot base. Visualize the environment.

```
env = {collisionBox(0.5, 0.5, 0.05) collisionSphere(0.3)};
env{1}.Pose(3, end) = -0.05;
env{2}.Pose(1:3, end) = [0.1 0.2 0.8];

show(robot);
hold on
show(env{1})
show(env{2})
```



Create the RRT planner for the robot model.

```
rrt = manipulatorRRT(robot,env);
rrt.SkippedSelfCollisions = "parent";
```

Specify a start and a goal configuration.

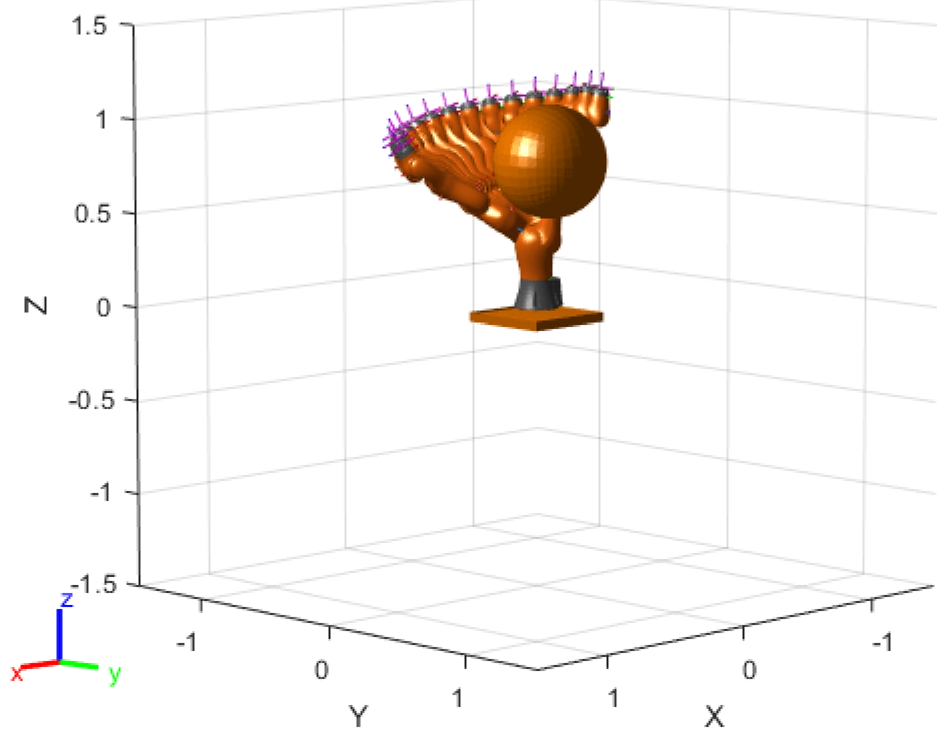
```
startConfig = [0.08 -0.65 0.05 0.02 0.04 0.49 0.04];
goalConfig =  [2.97 -1.05 0.05 0.02 0.04 0.49 0.04];
```

Plan the path. Due to the randomness of the RRT algorithm, set the `rng` seed for repeatability.

```
rng(0)
path = plan(rrt,startConfig,goalConfig);
```

Visualize the path. To add more intermediate states, interpolate the path. By default, the `interpolate` object function uses the value of `ValidationDistance` property to determine the number of intermediate states. The `for` loop shows every 20th element of the interpolated path.

```
interpPath = interpolate(rrt,path);
clf
for i = 1:20:size(interpPath,1)
    show(robot,interpPath(i,:));
    hold on
end
show(env{1})
show(env{2})
hold off
```

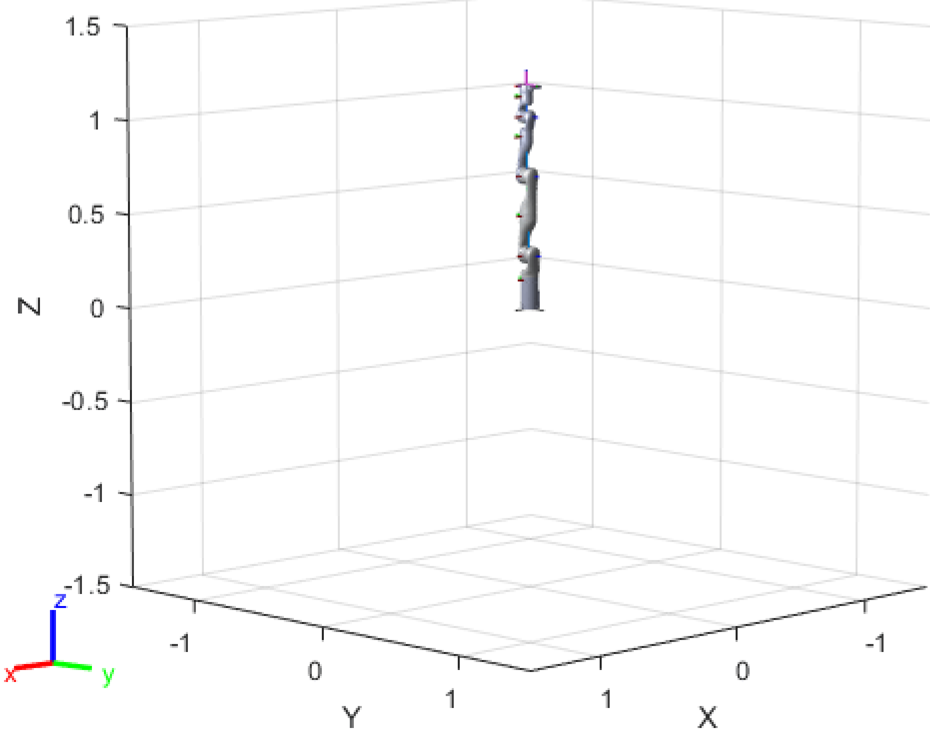


Plan Path To Workspace Goal Region

Specify a goal region in your workspace and plan a path within those bounds. The `workspaceGoalRegion` object defines the bounds on the *xyz*-position and *zyx* Euler orientation of the robot end effector. The `manipulatorRRT` object plans a path based on that goal region and samples random poses within the bounds.

Load an existing robot model as a `rigidBodyTree` object.

```
robot = loadrobot("kinovaGen3", "DataFormat", "row");
ax = show(robot);
```



Create Path Planner

Create a rapidly-exploring random tree (RRT) path planner for the robot. This example uses an empty environment, but this workflow also works well with cluttered environments. You can add collision objects to the environment like the `collisionBox` or `collisionMesh` object.

```
planner = manipulatorRRT(robot,{});
planner.SkippedSelfCollisions="parent";
```

Define Goal Region

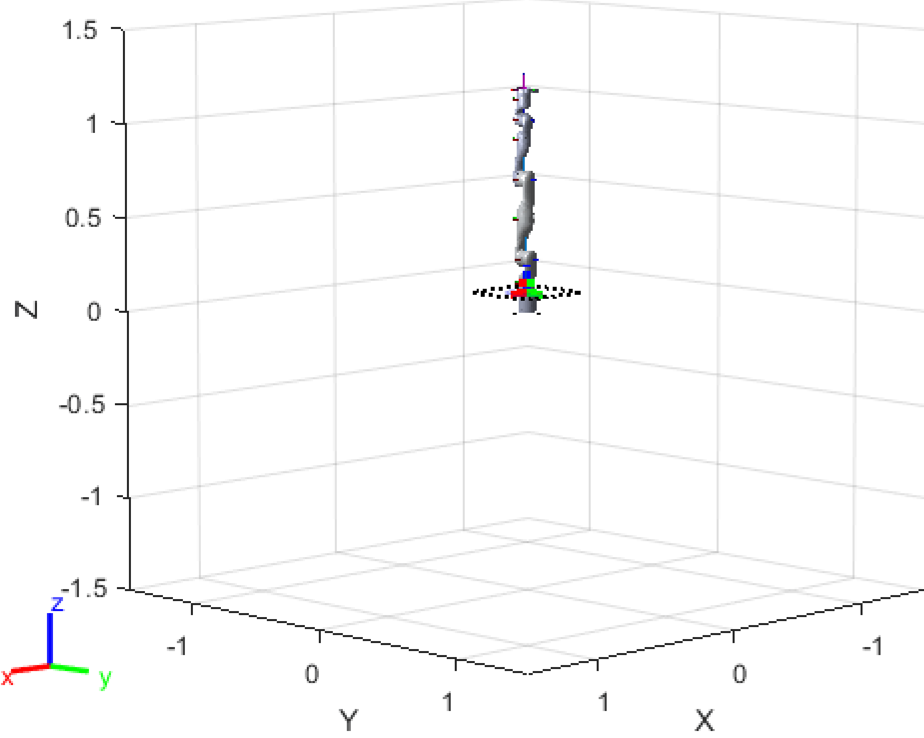
Create a workspace goal region using the end-effector body name of the robot.

Define the goal region parameters for your workspace. The goal region includes a reference pose, *xyz*-position bounds, and orientation limits on the *zyx* Euler angles. This example specifies bounds on the *xy*-plane in meters and allows rotation about the *z*-axis in radians.

```
goalRegion = workspaceGoalRegion(robot.BodyNames{end});
goalRegion.ReferencePose = trvec2tform([0.5 0.5 0.2]);
goalRegion.Bounds(1, :) = [-0.2 0.2];    % X Bounds
goalRegion.Bounds(2, :) = [-0.2 0.2];    % Y Bounds
goalRegion.Bounds(4, :) = [-pi/2 pi/2];  % Rotation about the Z-axis
```

You can also apply a fixed offset to all poses sampled within the region. This offset can account for grasping tools or variations in dimensions within your workspace. For this example, apply a fixed transformation that places the end effector 5 cm above the workspace.

```
goalRegion.EndEffectorOffsetPose = trvec2tform([0 0 0.05]);
hold on
show(goalRegion);
```



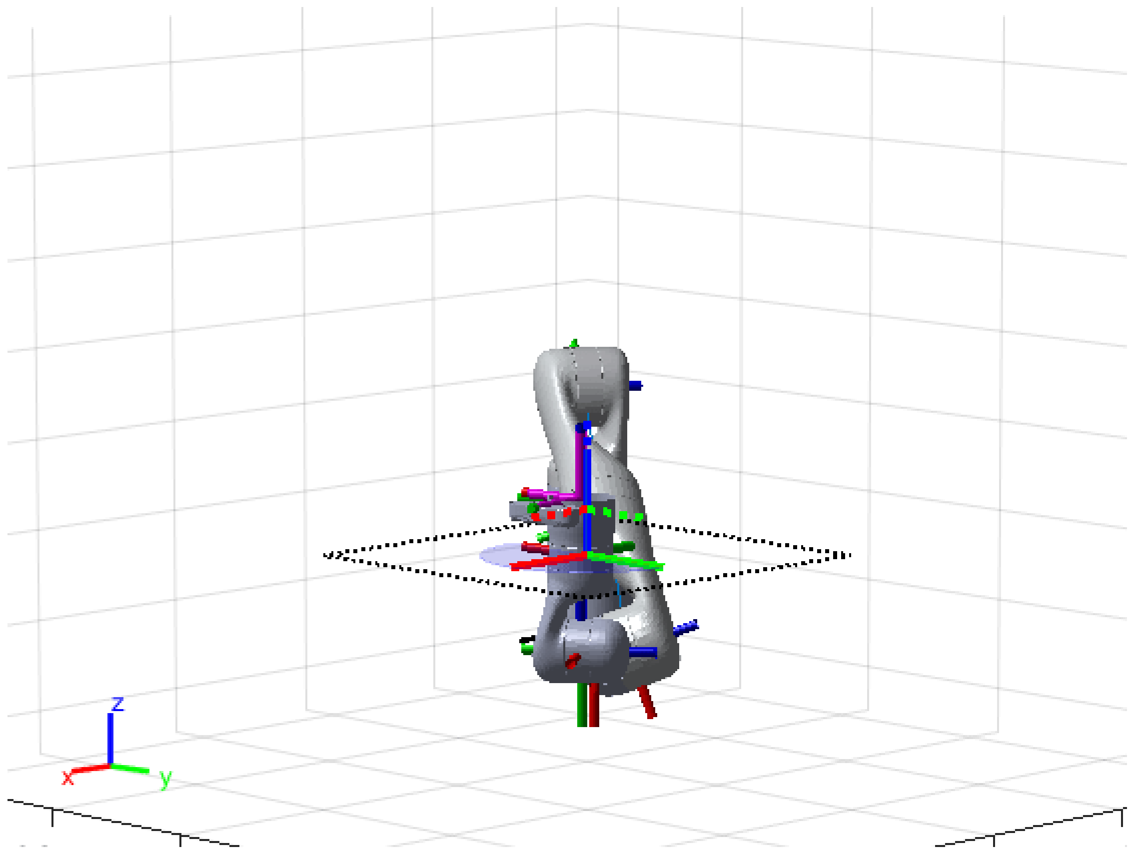
Plan Path To Goal Region

Plan a path to the goal region from the robot's home configuration. Due to the randomness in the RRT algorithm, this example sets the `rng` seed to ensure repeatable results.

```
rng(0)
path = plan(planner,homeConfiguration(robot),goalRegion);
```

Show the robot executing the path. To visualize a more realistic path, interpolate points between path configurations.

```
interpConfigurations = interpolate(planner,path,5);

for i = 1 : size(interpConfigurations)
    show(robot,interpConfigurations(i,:),"PreservePlot",false);
    set(ax,'ZLim',[-0.05 0.75],'YLim',[-0.05 1],'XLim',[-0.05 1],...
        'CameraViewAngle',5)

    drawnow
end
hold off
```



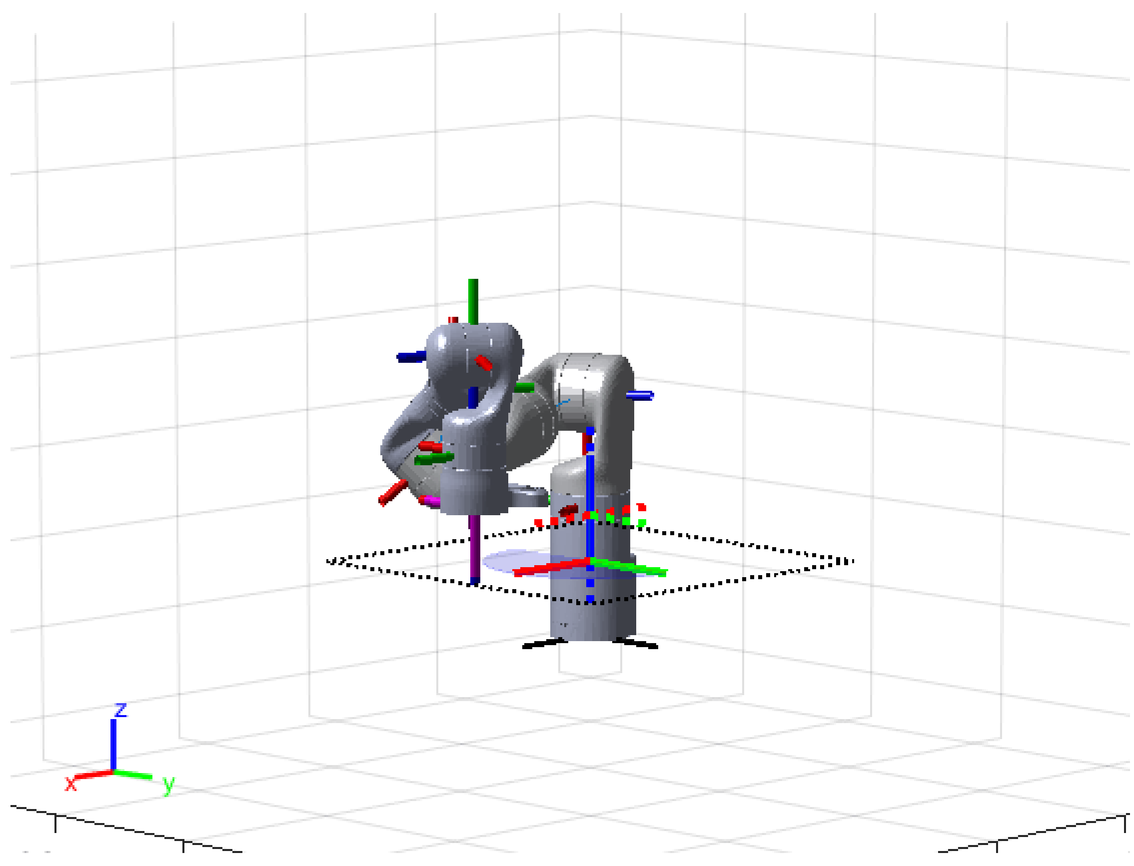
Adjust End-Effector Pose

Notice that the robot arm approaches the workspace from the bottom. To flip the orientation of the final position, add a `pi` rotation to the Y-axis for the reference pose.

```
goalRegion.EndEffectorOffsetPose = ...
    goalRegion.EndEffectorOffsetPose*eul2tform([0 pi 0],"ZYX");
```

Replan the path and visualize the robot motion again. The robot now approaches from the top.

```
hold on
show(goalRegion);
path = plan(planner,homeConfiguration(robot),goalRegion);

interpConfigurations = interpolate(planner,path,5);

for i = 1 : size(interpConfigurations)
    show(robot, interpConfigurations(i, :),"PreservePlot",false);
    set(ax,'ZLim',[-0.05 0.75],'YLim',[-0.05 1],'XLim',[-0.05 1])
    drawnow;
end
hold off
```



Plan Path Through 3-D Occupancy Map

Load the Kinova Gen 3 robot.

```
rbt = loadrobot("kinovagen3",DataFormat="row");
```

Create a 3-D occupancy map and set the coordinate at `[0.4 0.0 0.4]` to occupied.

```
map = occupancyMap3D(10);
map.setOccupancy([0.4 0.0 0.4],1);
```

Display the robot in the map.

```
show(map);
hold on
show(rbt);
axis("equal")
xlim([-1,1.0])
ylim([-1,1.0])
zlim([-0.5,1.2])
```

Define a start configuration.

```
startconfig = [2.2131,-1.3950,0.1618,0.2053,-0.1624,1.1684,-2.1886];
```

Define a goal configuration that is the same as the start configuration except for the first joint.

```
goalconfig = startconfig;
goalconfig(1) = 3.4;
```

Create the manipulator RRT planner for the robot and specify the map as the environment using the `Map` argument.

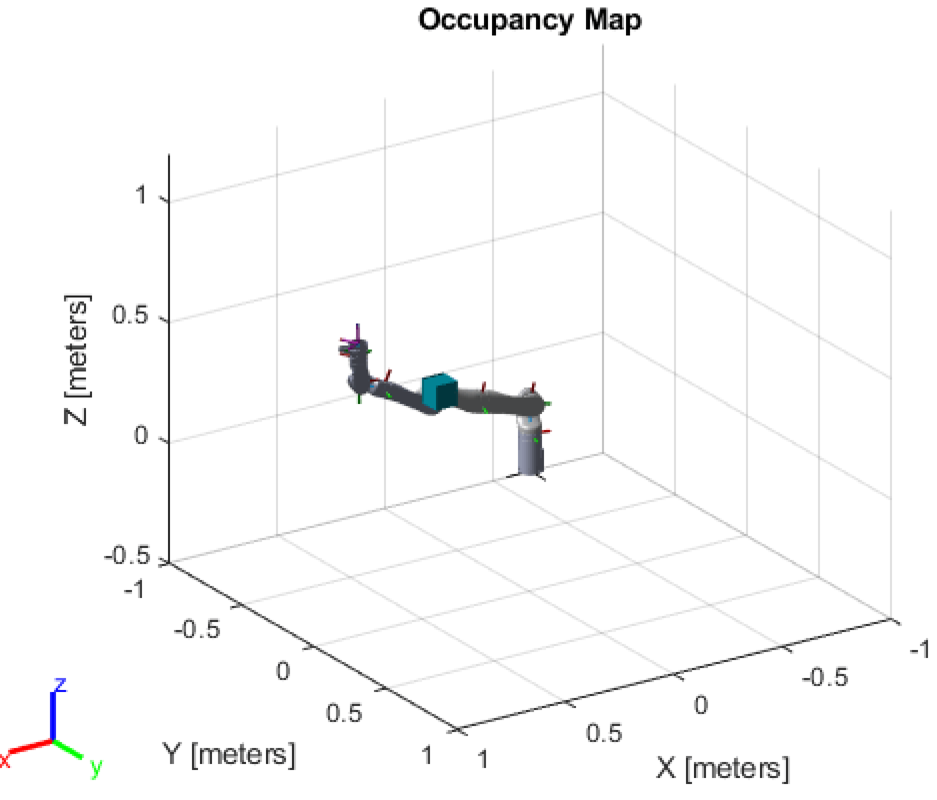
```
planner = manipulatorRRT(rbt,{},Map=map);
planner.ValidationDistance=0.1;
planner.MaxConnectionDistance=0.2;
planner.SkippedSelfCollisions="parent";
```

Plan a path between the start and goal configuration. Then interpolate between the paths.

```
plannedpath = plan(planner,startconfig,goalconfig);
interpoalatedpath = interpolate(planner,plannedpath);
```

Animate the robot following the path.

```
rc=rateControl(10);
view([pi/3,pi/2,pi/4]);
for i = 1:size(interpoalatedpath,1)
    show(rbt,interpoalatedpath(i,:),FastUpdate=true,PreservePlot=false);
    waitfor(rc);
end
```



Tips

Planning Complexity

- When planning the motion between nodes in the tree, a set of configurations are generated and validated. This computation time of the planner is proportional to the number of configurations generated. The number of configurations between nodes is controlled by the ratio of the MaxConnectionDistance and ValidationDistance properties. To improve planning time, consider increasing the validation distance or decreasing the max connection distance.
- Validating each configuration has a complexity of $O(mn+m^2)$, where m is the number of collision bodies in the `rigidBodyTree` object and n is the number of collision objects in `worldObjects`. Using large numbers of meshes to represent your robot or environment increases the time for validating each configuration.

Infinite Joint Limits

- If your `rigidBodyTree` robot model has joint limits that have infinite range (e.g. revolute joint with limits of `[-Inf Inf]`), the `manipulatorRRT` object uses limits of `[-1e10 1e10]` to perform uniform random sampling in the joint limits.

Version History

Introduced in R2020b

R2022b: Alter rigid body tree self-collision checking behavior change and new default self-collision checking behavior

Behavior change in future release

You can now specify self-collision checking behavior for a rigid body tree robot model by using the `SkippedSelfCollisions` property. Specify `SkippedSelfCollisions` as `"parent"` or `"adjacent"`:

- `"parent"` — Collision checking ignores self-collisions between parent and child rigid bodies.
- `"adjacent"` — Collision checking ignores self -collisions between rigid bodies of adjacent indices.

As of R2022b, the default behavior of collision checking is to ignore self-collisions between parent and child rigid bodies. In previous releases, the default behavior of self-collision checking was to ignore self-collisions between adjacent rigid bodies. To instead ignore self-collisions between rigid bodies of adjacent indices, specify `SkippedSelfCollisions` as `"adjacent"`.

See “Change Self-Collision Checking Behavior” on page 3-365 for more information.

R2023a: Specify environments using 3-D occupancy maps

If you have the Navigation Toolbox, you can specify the planning environment as a 3-D occupancy map using the `occupancyMap3D` object.

References

[1] Kuffner, J. J., and S. M. LaValle. “RRT-Connect: An Efficient Approach to Single-Query Path Planning.” In *Proceedings 2000 ICRA. Millennium Conference. IEEE International Conference*

on Robotics and Automation. Symposia Proceedings (Cat. No.00CH37065), 2:995–1001. San Francisco, CA, USA: IEEE, 2000. https://doi:10.1109/ROBOT.2000.844730.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

Objects
`rigidBodyTree` | `interactiveRigidBodyTree` | `analyticalInverseKinematics` | `occupancyMap3D`

Functions
`plan` | `interpolate` | `shorten`

Topics
"Pick and Place Using RRT for Manipulators"
"Pick-and-Place Workflow Using RRT Planner and Stateflow for MATLAB"

manipulatorStateSpace

State space for rigid body tree robot models

Description

The `manipulatorStateSpace` object represents the joint configuration state space of a rigid body tree robot model. For a given `rigidBodyTree` object, the nonfixed joints in the rigid body tree model form the state space. When sampling the state or specifying bounds, the values of the state vector correspond to joint positions that define a joint configuration with dimension equal to the `NumStateVariables` property.

Typically, the manipulator state space works with sampling-based path planners like the `plannerRRT` and `plannerBiRRT` objects. To sample and validate paths for manipulators, combine the state space with a state validator `manipulatorCollisionBodyValidator` object. Because the `manipulatorStateSpace` object derives from the `nav.StateSpace` class, and is specified in the `StateSpace` property of the path planners.

To plan paths for manipulators using only Robotics System Toolbox, see the `manipulatorRRT` object.

Creation

Syntax

```
manipSS = manipulatorStateSpace
manipSS = manipulatorStateSpace(robot)
manipSS = manipulatorStateSpace(robot,numStateVariables)
```

Description

`manipSS = manipulatorStateSpace` creates a default state space for a rigid body tree with two revolute joints.

`manipSS = manipulatorStateSpace(robot)` creates a state space for the specified `rigidBodyTree` object, `robot`.

`manipSS = manipulatorStateSpace(robot,numStateVariables)` specifies the number of state variables, which is the number of nonfixed joints in the robot model. You must use this syntax for code generation.

Properties

`RigidBodyTree` — Rigid body tree robot model
`rigidBodyTree` object

Rigid body tree robot model, specified as a `rigidBodyTree` object. After you create the `manipulatorStateSpace` object, this property is read-only.

Name — Name of state space object
"manipulatorStateSpace" (default) | string scalar | character vector

This property is read-only.

Name of the state space object, specified as a string scalar or character vector.

Example: "customManipulatorState"

NumStateVariables — Dimension of state space
2 (default) | positive numeric integer

Dimension of the state space, specified as a positive numeric integer. This property is the dimension of the state space and should match the size of the robot model joint configuration. To get a joint configuration, see the homeConfiguration or randomConfiguration function.

After you create the object, this property is read-only.

StateBounds — Minimum and maximum bounds of joint positions
n-by-2 matrix

Min and max bounds of the joint positions, specified as an *n*-by-2 matrix with rows of form [min max]. *n* is the number of state variables in the joint configuration space, specified in the NumStateVariables property. You must specify the [min max] joint positions in meters for prismatic joints and in radians for revolute joints.

Example: [-10 10; -10 10; -pi pi]

Data Types: double

Object Functions

distance	Distance between states
enforceStateBounds	Limit state to state space bounds
sampleUniform	Sample state using uniform distribution
sampleGaussian	Sample state using Gaussian distribution
interpolate	Interpolate between states

Examples

Validate State and Motion Manipulator State Space

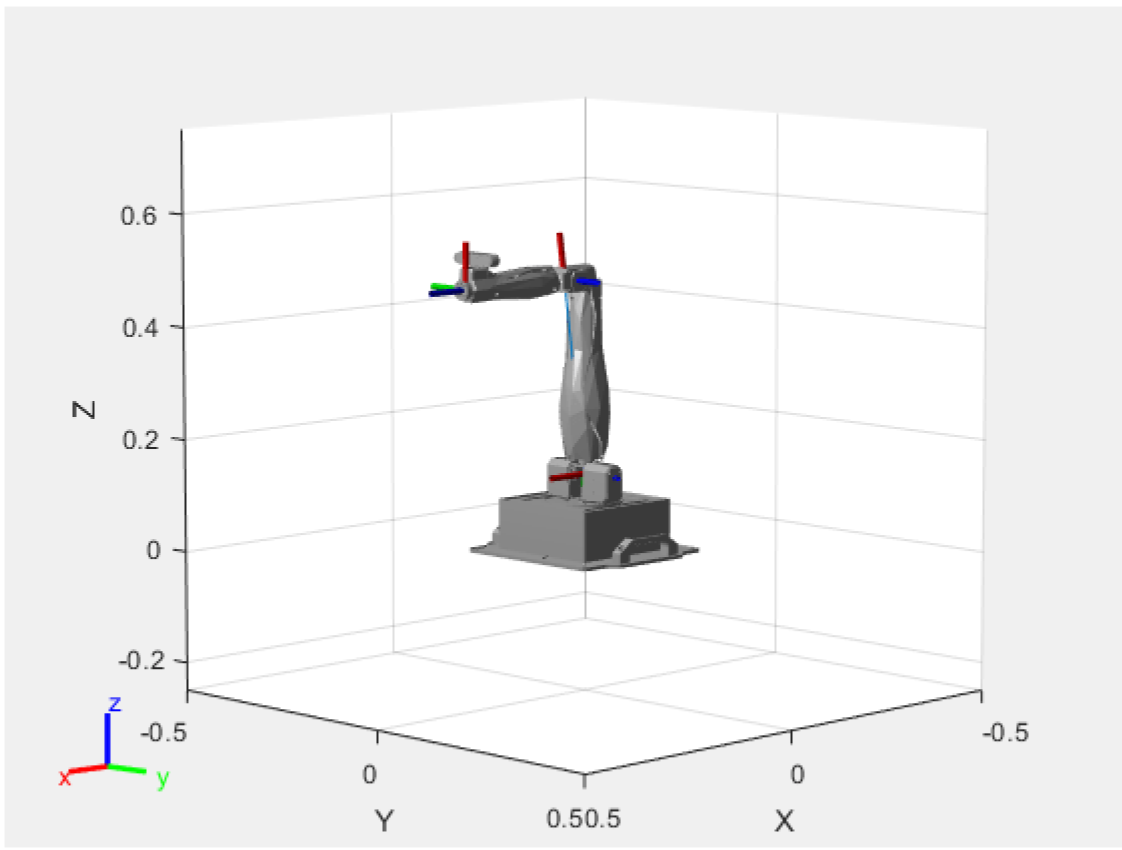
Generate states to form a path, validate motion between states, and check for self-collisions and environmental collisions with objects in your world. The manipulatorStateSpace object represents the joint configuration space of your rigid body tree robot model, and can sample states, calculate distances, and enforce state bounds. The manipulatorCollisionBodyValidator object validates the state and motion based on the collision bodies in your robot model and any obstacles in your environment.

Load Robot Model

Use the loadrobot function to access predefined robot models. This example uses the Quanser QArm™ robot and joint configurations are specified as row vectors.

```
robot = loadrobot("quanserQArm",DataFormat="row");
figure(Visible="on")
```

```
show(robot);
xlim([-0.5 0.5])
ylim([-0.5 0.5])
zlim([-0.25 0.75])
hold on
```



Configure State Space and State Validation

Create the state space and state validator from the robot model.

```
ss = manipulatorStateSpace(robot);
sv = manipulatorCollisionBodyValidator(ss,SkippedSelfCollisions="parent");
```

Set the validation distance to `0.05`, which is based on the distance normal between two states. You can configure the validator to ignore self collisions to improve the speed of validation, but must consider whether your robot model has the proper joint limit settings set to ensure it does not collide with itself.

```
sv.ValidationDistance = 0.05;
sv.IgnoreSelfCollision = true;
```

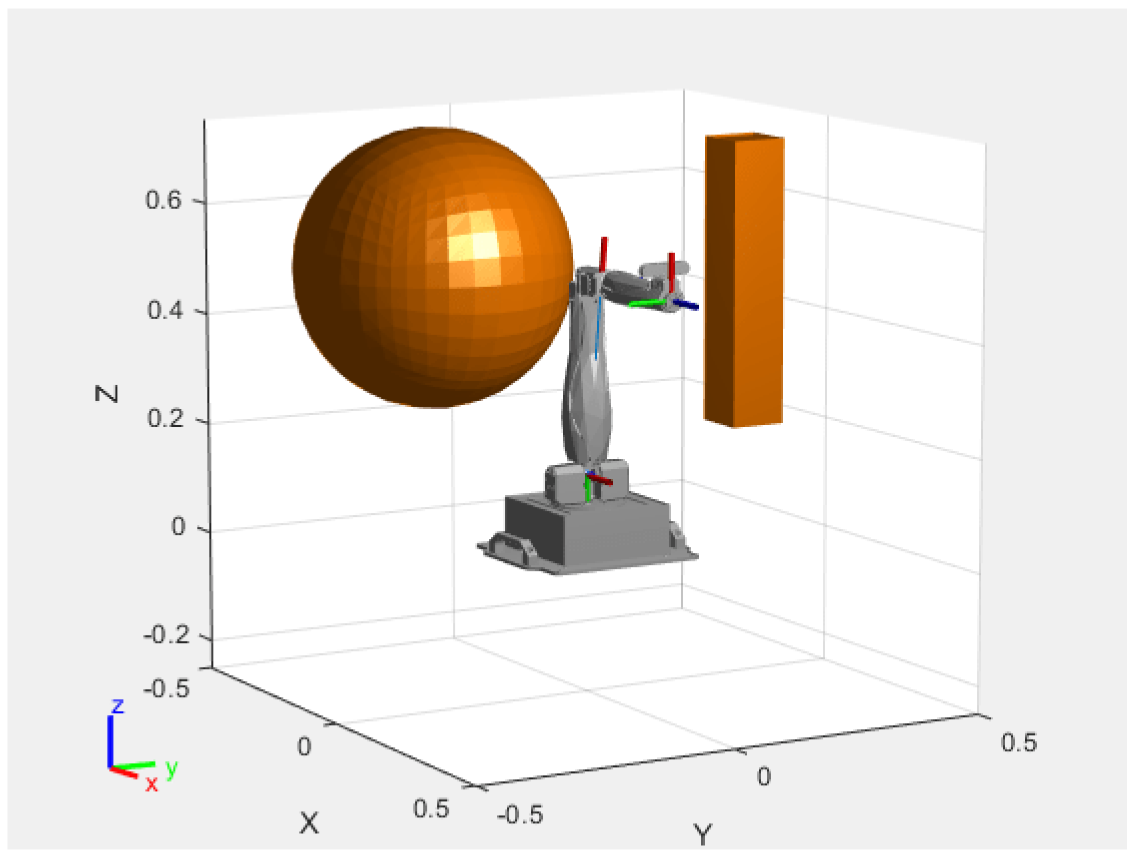
Place collision objects in the robot environment. Set the `Environment` property of the collision validator object using a cell array of objects.

```
box = collisionBox(0.1,0.1,0.5); % XYZ Lengths
box.Pose = trvec2tform([0.2 0.2 0.5]); % XYZ Position
sphere = collisionSphere(0.25); % Radius
sphere.Pose = trvec2tform([-0.2 -0.2 0.5]); % XYZ Position
```

```
env = {box sphere};
sv.Environment = env;
```

Visualize the environment.

```
for i = 1:length(env)
    show(env{i})
end
view(60,10)
```



Plan Path

Start with the home configuration as the first point on the path.

```
rng(0); % Repeatable results
start = homeConfiguration(robot);
path = start;
idx = 1;
```

Plan a path using these steps, in a loop:

- Sample a nearby goal configuration, using the Gaussian distribution, by specifying the standard deviation for each joint angle.
- Check if the sampled goal state is valid.
- If the sampled goal state is valid, check if the motion between states is valid and, if so, add it to the path.

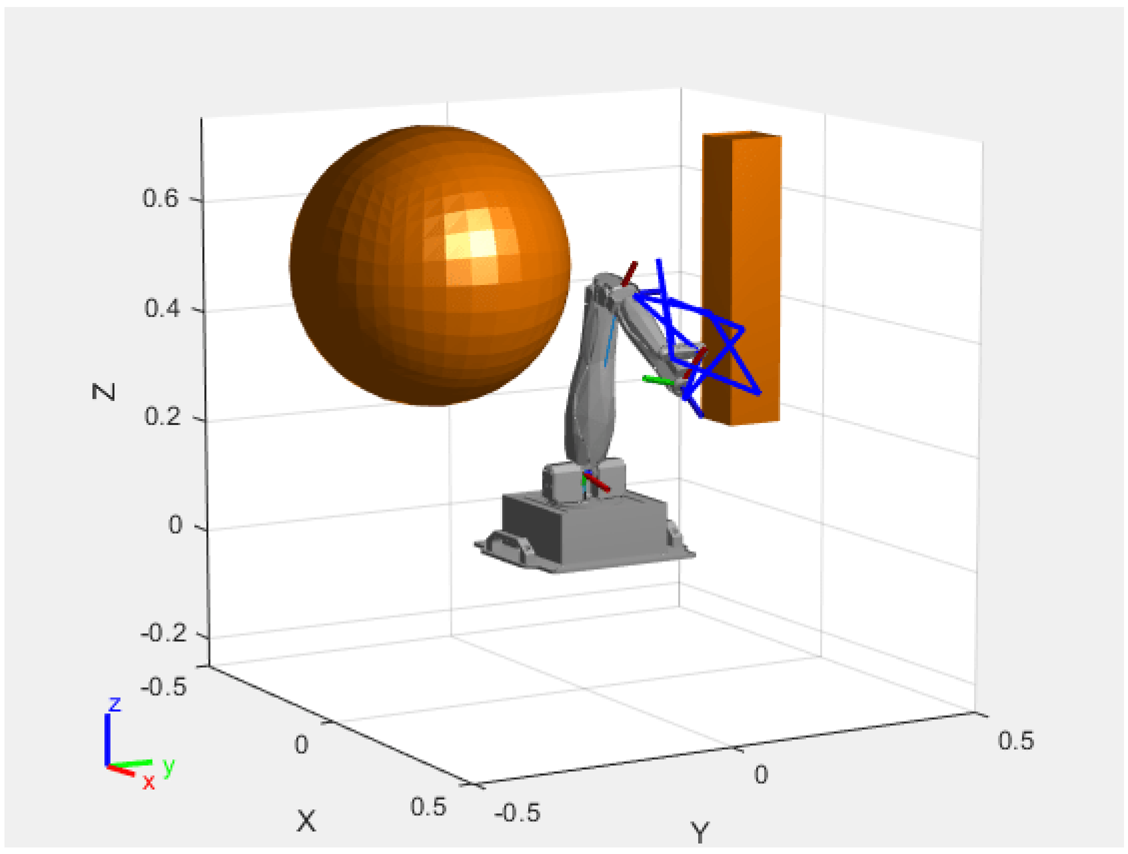
```
for i = 2:25
    goal = sampleGaussian(ss,start,0.25*ones(4,1));
    validState = isStateValid(sv,goal);

    if validState % If state is valid, check motion between states.
        [validMotion,~] = isMotionValid(sv,path(idx,:),goal);

        if validMotion % If motion is valid, add to path.
            path = [path; goal];
            idx = idx + 1;
        end
    end
end
```

Visualize Path

After generating the path of valid motions, visualize the robot motion. Because you sampled random states near the home configuration, you should see the arm move around that initial configuration.

To visualize the path of the end effector in 3-D, get the transformation, relative to the base world frame at each point. Store the points as an *xyz* translation vector. Plot the path of the end effector.

```
eePose = nan(3,size(path,1));

for i = 1:size(path,1)
    show(robot,path(i,:),PreservePlot=false);
    eePos(i,:) = tform2trvec(getTransform(robot,path(i,:),"END-EFFECTOR")); % XYZ translation vec
    plot3(eePos(:,1),eePos(:,2),eePos(:,3),"-b",LineWidth=2)
    drawnow
end
```



Version History

Introduced in R2021b

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

Objects

`rigidBodyTree` | `manipulatorCollisionBodyValidator` | `manipulatorRRT` | `workspaceGoalRegion`

Functions

`isStateValid` | `isMotionValid` | `sampleUniform` | `sampleGaussian` | `interpolate` | `distance` | `enforceStateBounds`

mobileRobotPRM

Create probabilistic roadmap path planner

Description

The mobileRobotPRM object is a roadmap path planner object for the environment map specified in the Map property. The object uses the map to generate a roadmap, which is a network graph of possible paths in the map based on free and occupied spaces. You can customize the number of nodes, NumNodes, and the connection distance, ConnectionDistance, to fit the complexity of the map and find an obstacle-free path from a start to an end location.

After the map is defined, the mobileRobotPRM path planner generates the specified number of nodes throughout the free spaces in the map. A connection between nodes is made when a line between two nodes contains no obstacles and is within the specified connection distance.

After defining a start and end location, to find an obstacle-free path using this network of connections, use the findpath method. If findpath does not find a connected path, it returns an empty array. By increasing the number of nodes or the connection distance, you can improve the likelihood of finding a connected path, but tuning these properties is necessary. To see the roadmap and the generated path, use the visualization options in show. If you change any of the mobileRobotPRM properties, call update, show, or findpath to recreate the roadmap.

Creation

Syntax

```
planner = mobileRobotPRM

planner = mobileRobotPRM(map)
planner = mobileRobotPRM(map,numnodes)
```

Description

planner = mobileRobotPRM creates an empty roadmap with default properties. Before you can use the roadmap, you must specify a binaryOccupancyMap object in the Map property.

planner = mobileRobotPRM(map) creates a roadmap with map set as the Map property, where map is a binaryOccupancyMap object.

planner = mobileRobotPRM(map,numnodes) sets the maximum number of nodes, numnodes, to the NumNodes property.

Input Arguments

map — Map representation
binaryOccupancyMap object

Map representation, specified as a `binaryOccupancyMap` object. This object represents the environment of the robot. The object is a matrix grid with binary values indicating obstacles as `true` (`1`) and free locations as `false` (`0`).

numnodes — Maximum number of nodes in roadmap
50 (default) | scalar

Maximum number of nodes in roadmap, specified as a scalar. By increasing this value, the complexity and computation time for the path planner increases.

Properties

`ConnectionDistance` — Maximum distance between two connected nodes
`inf` (default) | scalar in meters

Maximum distance between two connected nodes, specified as the comma-separated pair consisting of `"ConnectionDistance"` and a scalar in meters. This property controls whether nodes are connected based on their distance apart. Nodes are connected only if no obstacles are directly in the path. By decreasing this value, the number of connections is lowered, but the complexity and computation time decreases as well.

`Map` — Map representation
`binaryOccupancyMap` object | `occupancyMap` object

Map representation, specified as the comma-separated pair consisting of `"Map"` and a `binaryOccupancyMap` or `occupancyMap` object. This object represents the environment of the robot. The object is a matrix grid with values indicating the occupancy of locations in the map.

`NumNodes` — Number of nodes in the map
50 (default) | scalar

Number of nodes in the map, specified as the comma-separated pair consisting of `"NumNodes"` and a scalar. By increasing this value, the complexity and computation time for the path planner increases.

Object Functions

findpath	Find path between start and goal points on roadmap
show	Show map, roadmap, and path
update	Create or update roadmap

Version History

Introduced in R2019b

R2019b: `mobileRobotPRM` was renamed
Behavior change in future release

The `mobileRobotPRM` object was renamed from `robotics.PRM`. Use `mobileRobotPRM` for all object creation.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

The `map` input must be specified on creation of the `mobileRobotPRM` object.

See Also

`binaryOccupancyMap` | `occupancyMap` | `controllerPurePursuit`

Topics

"Path Planning in Environments of Different Complexity"
"Probabilistic Roadmaps (PRM)"

polynomialTrajectory

Piecewise-polynomial trajectory generator

Description

The `polynomialTrajectory` System object generates trajectories using a specified piecewise polynomial.

You can create a piecewise-polynomial structure using trajectory generators like `minjerkpolytraj`, `minsnappolytraj`, and `cubicpolytraj`, as well as any custom trajectory generator. You can then pass the structure to the `polynomialTrajectory` System object to create a trajectory interface for scenario simulation using the `robotScenario` object.

To generate a trajectory from a piecewise polynomial:

1. Create the `polynomialTrajectory` object and set its properties.
2. Call the object with arguments, as if it were a function.

To learn more about how System objects work, see What Are System Objects?

Creation

Syntax

```
trajectory = polynomialTrajectory(pp,Name=Value)
```

Description

`trajectory = polynomialTrajectory(pp,Name=Value)` returns a System object, `trajectory`, that generates a trajectory using the piecewise polynomial `pp`. Specify properties using one or more name-value arguments. Properties that you do not specify retain their default values.

Input Arguments

pp — Piecewise polynomial
structure

Piecewise polynomial, specified as a structure that defines the polynomial for each section of the piecewise trajectory.

Data Types: `struct`

Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see System Design in MATLAB Using System Objects.

`SampleRate` — Sample rate of trajectory (Hz)

`100` (default) | positive scalar

Sample rate of the trajectory in Hz, specified as a positive scalar.

Tunable: Yes

Data Types: `double`

`SamplesPerFrame` — Number of samples per output frame

`1` (default) | positive integer

Number of samples per output frame, specified as a positive integer.

Data Types: `double`

`Orientation` — Orientation at each waypoint

N-element `quaternion` column vector | 3-by-3-by-*N* array of real numbers

Orientation at each waypoint, specified as an *N*-element `quaternion` column vector or 3-by-3-by-*N* array of real numbers. *N* is the number of waypoints.

Each `quaternion` must have a norm of `1`. Each 3-by-3 rotation matrix must be an orthonormal matrix. Each quaternion or rotation matrix is a frame rotation from the local navigation coordinate system to the current body coordinate system at the corresponding waypoint.

If you do not specify this property, then the object sets yaw to the direction of travel at each waypoint, and pitch and roll are subject to the values of the AutoPitch and AutoBank properties, respectively.

Data Types: `double`

`Waypoints` — Positions in navigation coordinate system (m)

N-by-3 matrix

This property is read-only.

Positions in the navigation coordinate system, in meters, specified as an *N*-by-3 matrix. The columns of the matrix correspond to the first, second, and third axes, respectively. The rows of the matrix, *N*, correspond to individual waypoints.

The object infers this value from the piecewise polynomial `pp`.

Data Types: `double`

`TimeOfArrival` — Timestamp at each waypoint (s)

N-element column vector of nonnegative increasing numbers

This property is read-only.

Timestamp at each waypoint, in seconds, specified as an *N*-element column vector. The number of samples, *N*, is the same as the number of samples (rows) in Waypoints property. The value of each element of the vector must be greater than the value of the previous element.

The object infers this value from the piecewise polynomial `pp`.

Data Types: `double`

`AutoPitch` — Align pitch angle with direction of motion
`false`or `0` (default) | `true`or `1`

Align the pitch angle with the direction of motion, specified as a logical `0` (`false`) or `1` (`true`). When specified as `true`, the pitch angle automatically aligns with the direction of motion. If specified as `false`, the object sets the pitch angle to `0` (level orientation).

Dependencies

To set this property, you must not specify the Orientation property.

Data Types: `logical`

`AutoBank` — Align roll angle to counteract centripetal force
`false`or `0` (default) | `true`or `1`

Align the roll angle to counteract the centripetal force, specified as a logical `0` (`false`) or `1` (`true`). When specified as `true`, the roll angle automatically counteracts the centripetal force. If specified as `false`, the object sets the roll angle to `0` (flat orientation).

Dependencies

To set this property, you must not specify the Orientation property.

Data Types: `logical`

`ReferenceFrame` — Reference frame of trajectory
`"NED"` (default) | `"ENU"`

Reference frame of the trajectory, specified as `"NED"` (North-East-Down) or `"ENU"` (East-North-Up).

Data Types: `char` | `string`

`Velocities` — Velocity in navigation coordinate system at each waypoint (m/s)
N-by-3 matrix

This property is read-only.

Velocity in the navigation coordinate system at each waypoint, in meters per second, specified as an *N*-by-3 matrix. The columns of the matrix correspond to the first, second, and third axes, respectively. The number of samples, *N*, is the same as the number of samples (rows) in Waypoints property.

The object infers this value from the derivative of the piecewise polynomial `pp`.

Data Types: `double`

`Course` — Horizontal direction of travel (degrees)
N-element real vector

This property is read-only.

Horizontal direction of travel, in degrees, specified as an *N*-element real vector. The number of samples, *N*, is the same as the number of samples (rows) in Waypoints property.

The object infers this value from the derivative of the piecewise polynomial `pp`.

Data Types: `double`

GroundSpeed — Groundspeed at each waypoint (m/s)
N-element real vector

This property is read-only.

Groundspeed at each waypoint, in meters per second, specified as an *N*-element real vector. The number of samples, *N*, is the same as the number of samples (rows) in Waypoints property.

The object infers this value from the derivative of the piecewise polynomial `pp`.

Data Types: `double`

ClimbRate — Climb rate at each waypoint (m/s)
N-element real vector

This property is read-only.

Climb Rate at each waypoint, in meters per second, specified as an *N*-element real vector. The number of samples, *N*, is the same as the number of samples (rows) in Waypoints property.

The object infers this value from the derivative of the piecewise polynomial `pp`.

Data Types: `double`

Usage

Syntax

`[position,orientation,velocity,acceleration,angularVelocity] = trajectory()`

Description

`[position,orientation,velocity,acceleration,angularVelocity] = trajectory()` outputs a frame of trajectory data based on specified creation arguments and properties. The trajectory returns `NaN` for positions and orientations outside the range of the time of arrival.

Output Arguments

position — Position in local navigation coordinate system
M-by-3 matrix

Position in the local navigation coordinate system, returned as an *M*-by-3 matrix in meters.

M is specified by the SamplesPerFrame property.

Data Types: `double`

orientation — Orientation in local navigation coordinate system
M-element `quaternion` column vector | 3-by-3-by-*M* real array

Orientation in the local navigation coordinate system, returned as an *M*-element `quaternion` column vector or a 3-by-3-by-*M* real array.

Each quaternion or 3-by-3 rotation matrix is a frame rotation from the local navigation coordinate system to the current body coordinate system at the corresponding sample.

M is specified by the SamplesPerFrame property.

Data Types: `double`

`velocity` — Velocity in local navigation coordinate system
M-by-3 matrix

Velocity in the local navigation coordinate system, returned as an *M*-by-3 matrix, in meters per second.

M is specified by the SamplesPerFrame property.

Data Types: `double`

`acceleration` — Acceleration in local navigation coordinate system
M-by-3 matrix

Acceleration in the local navigation coordinate system, returned as an *M*-by-3 matrix, in meters per second squared.

M is specified by the SamplesPerFrame property.

Data Types: `double`

`angularVelocity` — Angular velocity in local navigation coordinate system
M-by-3 matrix

Angular velocity in the local navigation coordinate system, returned as an *M*-by-3 matrix, in radians per second.

M is specified by the SamplesPerFrame property.

Data Types: `double`

Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

Specific to `polynomialTrajectory`

lookupPose	Obtain pose information for certain time
waypointInfo	Get waypoint information table

Common to All System Objects

step	Run System object algorithm
release	Release resources and allow changes to System object property values and input characteristics
clone	Create duplicate System object
isLocked	Determine if System object is in use
reset	Reset internal states of System object
isDone	End-of-data status

Examples

Generate Trajectory from Piecewise Polynomial Using polynomialTrajectory

Use the `minjerkpolytraj` function to generate the piecewise polynomial and the time samples for the specified waypoints of a trajectory.

```
waypoints = [0 20 20 0 0; 0 0 5 5 0; 0 5 10 5 0];
timePoints = cumsum([0 10 1.25*pi 10 1.25*pi]);
numSamples = 100;
[~,~,~,~,pp,~,tsamples] = minjerkpolytraj(waypoints,timePoints,numSamples);
```

Use the `polynomialTrajectory` System object to generate a trajectory from the piecewise polynomial that a multirotor must follow. Specify the sample rate of the trajectory and the orientation at each waypoint.

```
eulerAngles = [0 0 0; 0 0 0; 180 0 0; 180 0 0; 0 0 0];
q = quaternion(eulerAngles,"eulerd","ZYX","frame");
traj = polynomialTrajectory(pp,SampleRate=100,Orientation=q);
```

Inspect the waypoints, times of arrival, and orientation by using `waypointInfo`.

```
waypointInfo(traj)
```

```
ans=5×3 table
    TimeOfArrival                    Waypoints                          Orientation
    _____________    _______________________________________    ________________

           0                 0              0              0    {1x1 quaternion}
          10                20              0              5    {1x1 quaternion}
      13.927                20              5             10    {1x1 quaternion}
      23.927                 0              5              5    {1x1 quaternion}
      27.854         6.409e-14    -1.1102e-13    -1.1902e-13    {1x1 quaternion}
```

Obtain pose information one buffer frame at a time.

```
[pos,orient,vel,acc,angvel] = traj();
i = 1;
spf = traj.SamplesPerFrame;
while ~isDone(traj)
    idx = (i+1):(i+spf);
    [pos(idx,:),orient(idx,:), ...
     vel(idx,:),acc(idx,:),angvel(idx,:)] = traj();
    i = i + spf;
end
```

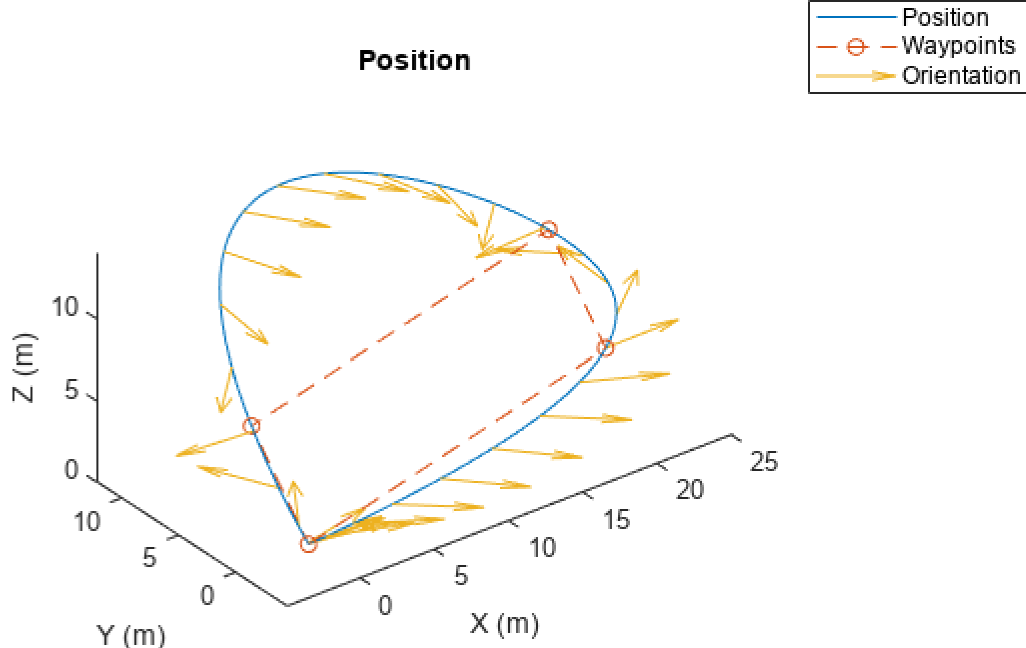
Get the yaw angle from the orientation.

```
eulOrientation = quat2eul(orient);
yawAngle = eulOrientation(:,1);
```

Plot the generated positions and orientations, as well as the specified waypoints.

```
plot3(pos(:,1),pos(:,2),pos(:,3), ...
      waypoints(1,:),waypoints(2,:),waypoints(3,:),"--o")
hold on
```

```
% Plot the yaw using quiver.
quiverIdx = 1:100:length(pos);
quiver3(pos(quiverIdx,1),pos(quiverIdx,2),pos(quiverIdx,3), ...
        cos(yawAngle(quiverIdx)),sin(yawAngle(quiverIdx)), ...
        zeros(numel(quiverIdx),1))
title("Position")
xlabel("X (m)")
ylabel("Y (m)")
zlabel("Z (m)")
legend({"Position","Waypoints","Orientation"})
axis equal
hold off
```



Obtain Pose Information of Polynomial Trajectory at Certain Time

Use the `minsnappolytraj` function to generate the piecewise polynomial and the time samples for the specified waypoints of a trajectory.

```
waypoints = [0 20 20 0 0; 0 0 5 5 0; 0 5 10 5 0];
timePoints = linspace(0,30,5);
numSamples = 100;
[~,~,~,~,~,pp,~,~] = minsnappolytraj(waypoints,timePoints,numSamples);
```

Use the `polynomialTrajectory` System object to generate a trajectory from the piecewise polynomial. Specify the sample rate of the trajectory.

```
traj = polynomialTrajectory(pp,SampleRate=100);
```

Inspect the waypoints and times of arrival by using `waypointInfo`.

```
waypointInfo(traj)
```

```
ans=5×2 table
    TimeOfArrival                     Waypoints
    _____________    ________________________________________

          0                   0              0              0
        7.5                  20              0              5
         15                  20              5             10
       22.5                   0              5              5
         30          3.3973e-13    -2.7018e-12    -2.6041e-12
```

Obtain the time of arrival between the second and fourth waypoint. Create timestamps to sample the trajectory.

```
t0 = traj.TimeOfArrival(2);
tf = traj.TimeOfArrival(4);
sampleTimes = linspace(t0,tf,1000);
```

Obtain the position, orientation, velocity, and acceleration information at the sampled timestamps using the `lookupPose` object function.

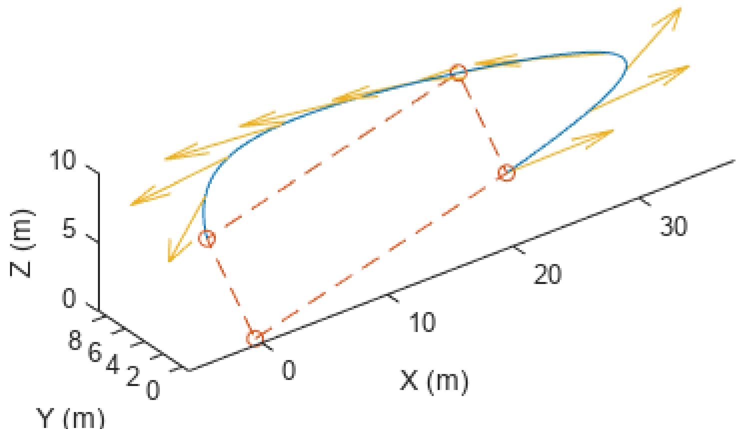
```
[pos,orient,vel,accel,~] = lookupPose(traj,sampleTimes);
```

Get the yaw angle from the orientation.

```
eulOrientation = quat2eul(orient);
yawAngle = eulOrientation(:,1);
```

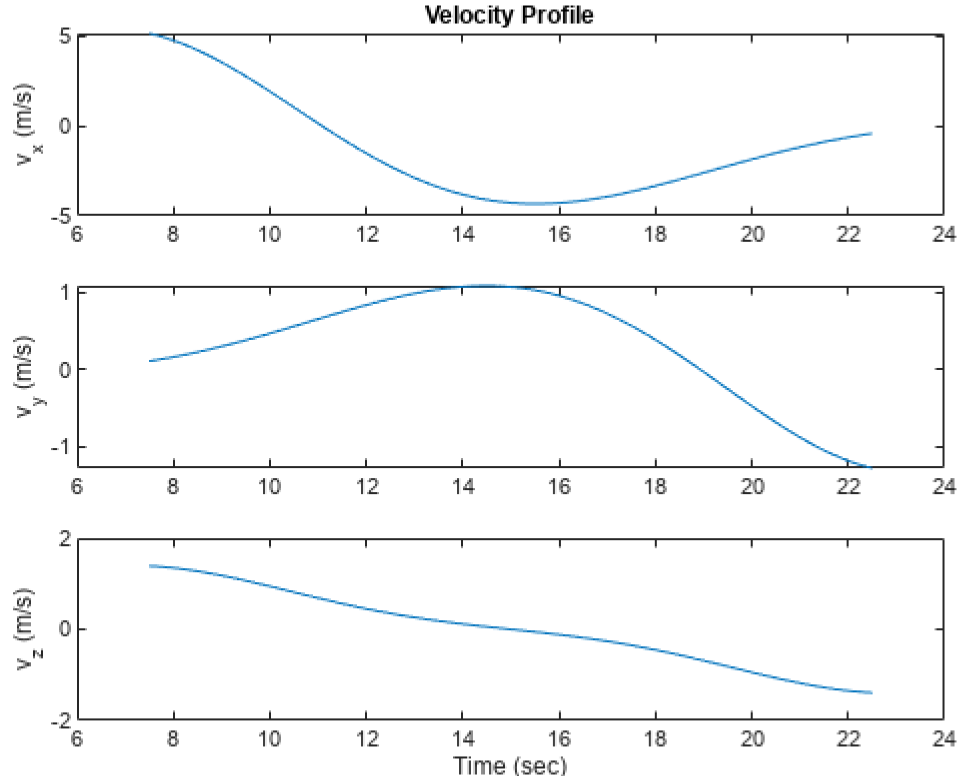
Plot the generated positions and orientations, as well as the specified waypoints.

```
plot3(pos(:,1),pos(:,2),pos(:,3), ...
      waypoints(1,:),waypoints(2,:),waypoints(3,:),"--o")
hold on
% Plot the yaw using quiver.
quiverIdx = 1:100:length(pos);
quiver3(pos(quiverIdx,1),pos(quiverIdx,2),pos(quiverIdx,3), ...
        cos(yawAngle(quiverIdx)),sin(yawAngle(quiverIdx)), ...
        zeros(numel(quiverIdx),1))
title("Position and Orientation")
xlabel("X (m)")
ylabel("Y (m)")
zlabel("Z (m)")
legend({"Position","Waypoints","Orientation"})
axis equal
hold off
```



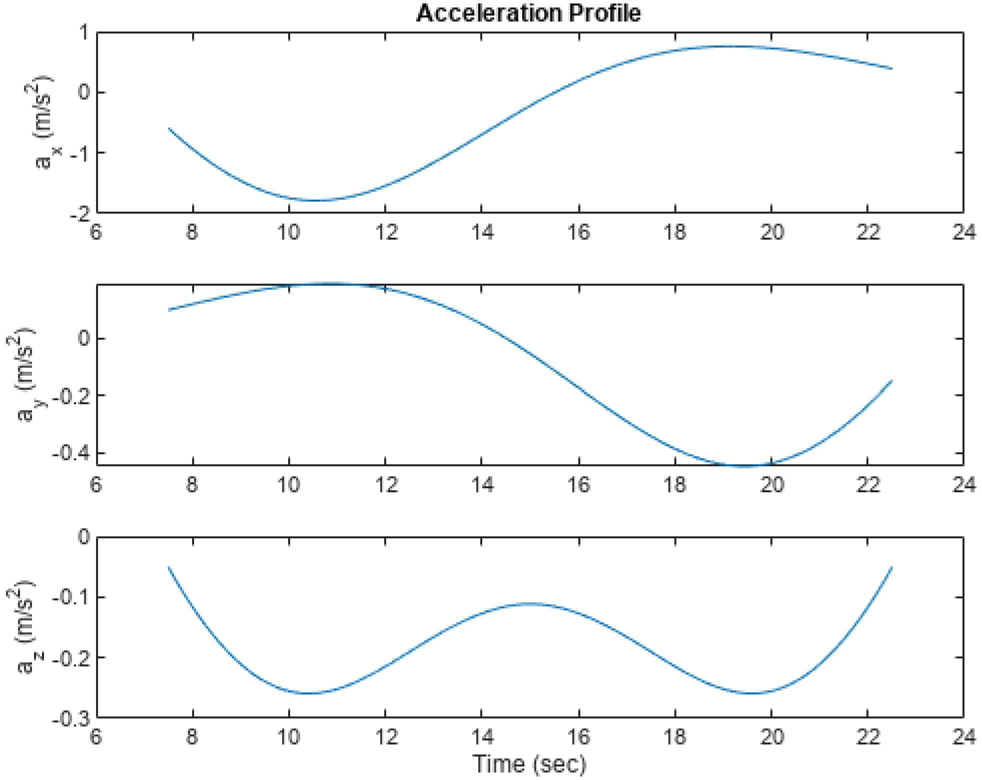
Plot the velocity profiles.

```
figure
subplot(3,1,1)
plot(sampleTimes,vel(:,1))
title("Velocity Profile")
ylabel("v_x (m/s)")
subplot(3,1,2)
plot(sampleTimes,vel(:,2))
ylabel("v_y (m/s)")
subplot(3,1,3)
plot(sampleTimes,vel(:,3))
ylabel("v_z (m/s)")
xlabel("Time (sec)")
```



Plot the acceleration profiles.

```
figure
subplot(3,1,1)
plot(sampleTimes,accel(:,1))
title("Acceleration Profile")
ylabel("a_x (m/s^2)")
subplot(3,1,2)
plot(sampleTimes,accel(:,2))
ylabel("a_y (m/s^2)")
subplot(3,1,3)
plot(sampleTimes,accel(:,3))
ylabel("a_z (m/s^2)")
xlabel("Time (sec)")
```



Version History

Introduced in R2023a

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

Objects

`waypointTrajectory` | `robotPlatform`

Functions

`waypointInfo` | `lookupPose`

pointCloud

Object for storing 3-D point cloud

Description

The `pointCloud` object creates point cloud data from a set of points in 3-D coordinate system. The point cloud data is stored as an object with the properties listed in “Properties” on page 1-285. Use “Object Functions” on page 1-286 to retrieve, select, and remove desired points from the point cloud data.

Creation

Syntax

```
ptCloud = pointCloud(xyzPoints)
ptCloud = pointCloud(xyzPoints,Name,Value)
```

Description

`ptCloud = pointCloud(xyzPoints)` returns a point cloud object with coordinates specified by `xyzPoints`.

`ptCloud = pointCloud(xyzPoints,Name,Value)` creates a `pointCloud` object with properties specified as one or more `Name,Value` pair arguments. For example, `pointCloud(xyzPoints,'Color',[0 0 0])` sets the `Color` property of the point `xyzPoints` as `[0 0 0]`. Enclose each property name in quotes. Any unspecified properties have default values.

Input Arguments

`xyzPoints` — 3-D coordinate points
M-by-3 list of points | *M*-by-*N*-by-3 array for organized point cloud

3-D coordinate points, specified as an *M*-by-3 list of points or an *M*-by-*N*-by-3 array for an organized point cloud. The 3-D coordinate points specify the *x*, *y*, and *z* positions of a point in the 3-D coordinate space. The first two dimensions of an organized point cloud correspond to the scanning order from sensors such as RGBD or lidar. This argument sets the `Location` property.

Data Types: `single` | `double`

Output Arguments

`ptCloud` — Point cloud
`pointCloud` object

Point cloud, returned as a `pointCloud` object with the properties listed in “Properties” on page 1-285.

Properties

`Location` — Position of the points in 3-D coordinate space

M-by-3 array | *M*-by-*N*-by-3 array

This property is read-only.

Position of the points in 3-D coordinate space, specified as an *M*-by-3 or *M*-by-*N*-by-3 array. Each entry specifies the *x*, *y*, and *z* coordinates of a point in the 3-D coordinate space. You cannot set this property as a name-value pair. Use the `xyzPoints` input argument.

Data Types: `single` | `double`

`Color` — Point cloud color

`[]` (default) | *M*-by-3 array | *M*-by-*N*-by-3 array

Point cloud color, specified as an *M*-by-3 or *M*-by-*N*-by-3 array. Use this property to set the color of points in point cloud. Each entry specifies the RGB color of a point in the point cloud data. Therefore, you can specify the same color for all points or a different color for each point.

- The specified RGB values must lie within the range [0, 1], when you specify the data type for `Color` as `single` or `double`.
- The specified RGB values must lie within the range [0, 255], when you specify the data type for `Color` as `uint8`.

Coordinates	Valid assignment of `Color`
M-by-3 array	*M*-by-3 array containing RGB values for each point
M-by-*N*-by-3 array	*M*-by-*N*-by-3 array containing RGB values for each point

Data Types: `uint8`

`Normal` — Surface normals

`[]` (default) | *M*-by-3 array | *M*-by-*N*-by-3 array

Surface normals, specified as a *M*-by-3 or *M*-by-*N*-by-3 array. Use this property to specify the normal vector with respect to each point in the point cloud. Each entry in the surface normals specifies the *x*, *y*, and *z* component of a normal vector.

Coordinates	Surface Normals
M-by-3 array	*M*-by-3 array, where each row contains a corresponding normal vector.
M-by-*N*-by-3 array	*M*-by-*N*-by-3 array containing a 1-by-1-by-3 normal vector for each point.

Data Types: `single` | `double`

`Intensity` — Grayscale intensities

`[]` (default) | *M*-by-1 vector | *M*-by-*N* matrix

Grayscale intensities at each point, specified as a *M*-by-1 vector or *M*-by-*N* matrix. The function maps each intensity value to a color value in the current colormap.

Coordinates	Intensity
M-by-3 array	*M*-by-1 vector, where each row contains a corresponding intensity value.

Coordinates	Intensity
M-by-*N*-by-3 array	*M*-by-*N* matrix containing intensity value for each point.

Data Types: `single` | `double` | `uint8`

`Count` — Number of points
positive integer

This property is read-only.

Number of points in the point cloud, stored as a positive integer.

`XLimits` — Range of *x* coordinates
1-by-2 vector

This property is read-only.

Range of coordinates along *x*-axis, stored as a 1-by-2 vector.

`YLimits` — Range of *y* coordinates
1-by-2 vector

This property is read-only.

Range of coordinates along *y*-axis, stored as a 1-by-2 vector.

`ZLimits` — Range of *z* coordinates
1-by-2 vector

This property is read-only.

Range of coordinates along *z*-axis, stored as a 1-by-2 vector.

Object Functions

findNearestNeighbors	Find nearest neighbors of a point in point cloud
findNeighborsInRadius	Find neighbors within a radius of a point in the point cloud
findPointsInROI	Find points within a region of interest in the point cloud
removeInvalidPoints	Remove invalid points from point cloud
select	Select points in point cloud
copy	Copy array of handle objects

Tips

The `pointCloud` object is a `handle` object. If you want to create a separate copy of a point cloud, you can use the MATLAB `copy` method.
`ptCloudB = copy(ptCloudA)`

If you want to preserve a single copy of a point cloud, which can be modified by point cloud functions, use the same point cloud variable name for the input and output.
`ptCloud =` *pcFunction*`(ptCloud)`

Version History

Introduced in R2022a

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

Usage notes and limitations:

- GPU code generation for variable input sizes is not optimized. Consider using constant size inputs for an optimized code generation.
- GPU code generation supports the `'Color'`, `'Normal'`, and `'Intensity'` name-value pairs.
- GPU code generation supports the `findNearestNeighbors`, `findNeighborsInRadius`, `findPointsInROI`, `removeInvalidPoints`, and `select` methods.
- For very large inputs, the memory requirements of the algorithm may exceed the GPU device limits. In such cases, consider reducing the input size to proceed with code generation.

See Also

Functions

`findNearestNeighbors` | `findNeighborsInRadius` | `findPointsInROI` | `removeInvalidPoints` | `select`

quaternion

Create a quaternion array

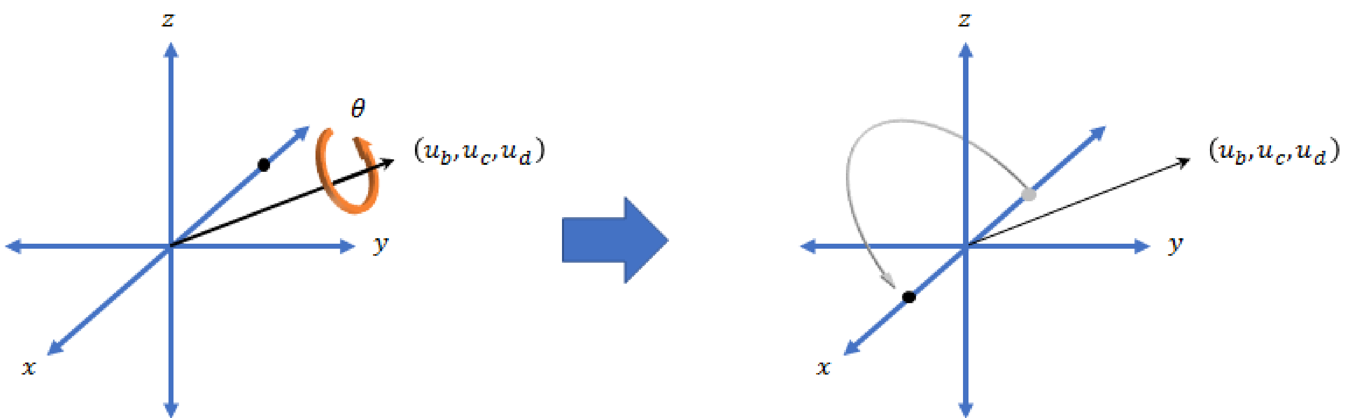
Description

A quaternion is a four-part hyper-complex number used in three-dimensional rotations and orientations.

A quaternion number is represented in the form $a + b\text{i} + c\text{j} + d\text{k}$, where a, b, c, and d parts are real numbers, and i, j, and k are the basis elements, satisfying the equation: $\text{i}^2 = \text{j}^2 = \text{k}^2 = \text{ijk} = -1$.

The set of quaternions, denoted by **H**, is defined within a four-dimensional vector space over the real numbers, $\mathbf{R}^4$. Every element of **H** has a unique representation based on a linear combination of the basis elements, i, j, and k.

All rotations in 3-D can be described by an axis of rotation and angle about that axis. An advantage of quaternions over rotation matrices is that the axis and angle of rotation is easy to interpret. For example, consider a point in $\mathbf{R}^3$. To rotate the point, you define an axis of rotation and an angle of rotation.



The quaternion representation of the rotation may be expressed as $q = \cos\left(\theta/2\right) + \sin\left(\theta/2\right)(u_b\text{i} + u_c\text{j} + u_d\text{k})$, where θ is the angle of rotation and $[u_b$, u_c, and $u_d]$ is the axis of rotation.

Creation

Syntax

```
quat = quaternion()
quat = quaternion(A,B,C,D)
quat = quaternion(matrix)
quat = quaternion(RV,'rotvec')
```

```
quat = quaternion(RV,'rotvecd')
quat = quaternion(RM,'rotmat',PF)
quat = quaternion(E,'euler',RS,PF)
quat = quaternion(E,'eulerd',RS,PF)
quat = quaternion(transformation)
quat = quaternion(rotation)
```

Description

`quat = quaternion()` creates an empty quaternion.

`quat = quaternion(A,B,C,D)` creates a quaternion array where the four quaternion parts are taken from the arrays `A`, `B`, `C`, and `D`. All the inputs must have the same size and be of the same data type.

`quat = quaternion(matrix)` creates an *N*-by-1 quaternion array from an *N*-by-4 matrix, where each column becomes one part of the quaternion.

`quat = quaternion(RV,'rotvec')` creates an *N*-by-1 quaternion array from an *N*-by-3 matrix of rotation vectors, `RV`. Each row of `RV` represents a rotation vector in radians.

`quat = quaternion(RV,'rotvecd')` creates an *N*-by-1 quaternion array from an *N*-by-3 matrix of rotation vectors, `RV`. Each row of `RV` represents a rotation vector in degrees.

`quat = quaternion(RM,'rotmat',PF)` creates an *N*-by-1 quaternion array from the 3-by-3-by-*N* array of rotation matrices, `RM`. `PF` can be either `'point'` if the Euler angles represent point rotations or `'frame'` for frame rotations.

`quat = quaternion(E,'euler',RS,PF)` creates an *N*-by-1 quaternion array from the *N*-by-3 matrix, `E`. Each row of `E` represents a set of Euler angles in radians. The angles in `E` are rotations about the axes in sequence `RS`.

`quat = quaternion(E,'eulerd',RS,PF)` creates an *N*-by-1 quaternion array from the *N*-by-3 matrix, `E`. Each row of `E` represents a set of Euler angles in degrees. The angles in `E` are rotations about the axes in sequence `RS`.

`quat = quaternion(transformation)` creates a quaternion array from the SE(3) transformation `transformation`.

`quat = quaternion(rotation)` creates an quaternion array from the SO(3) rotation `rotation`.

Input Arguments

`A,B,C,D` — Quaternion parts

comma-separated arrays of the same size

Parts of a quaternion, specified as four comma-separated scalars, matrices, or multi-dimensional arrays of the same size.

Example: `quat = quaternion(1,2,3,4)` creates a quaternion of the form 1 + 2i + 3j + 4k.

Example: `quat = quaternion([1,5],[2,6],[3,7],[4,8])` creates a 1-by-2 quaternion array where `quat(1,1) = 1 + 2i + 3j + 4k` and `quat(1,2) = 5 + 6i + 7j + 8k`

Data Types: `single` | `double`

`matrix` — Matrix of quaternion parts
N-by-4 matrix

Matrix of quaternion parts, specified as an *N*-by-4 matrix. Each row represents a separate quaternion. Each column represents a separate quaternion part.

Example: `quat = quaternion(rand(10,4))` creates a 10-by-1 quaternion array.

Data Types: `single` | `double`

RV — Matrix of rotation vectors
N-by-3 matrix

Matrix of rotation vectors, specified as an *N*-by-3 matrix. Each row of RV represents the [X Y Z] elements of a rotation vector. A rotation vector is a unit vector representing the axis of rotation scaled by the angle of rotation in radians or degrees.

To use this syntax, specify the first argument as a matrix of rotation vectors and the second argument as the `'rotvec'` or `'rotvecd'`.

Example: `quat = quaternion(rand(10,3),'rotvec')` creates a 10-by-1 quaternion array.

Data Types: `single` | `double`

RM — Rotation matrices
3-by-3 matrix | 3-by-3-by-*N* array

Array of rotation matrices, specified by a 3-by-3 matrix or 3-by-3-by-*N* array. Each page of the array represents a separate rotation matrix.

Example: `quat = quaternion(rand(3),'rotmat','point')`

Example: `quat = quaternion(rand(3),'rotmat','frame')`

Data Types: `single` | `double`

PF — Type of rotation matrix
`'point'` | `'frame'`

Type of rotation matrix, specified by `'point'` or `'frame'`.

Example: `quat = quaternion(rand(3),'rotmat','point')`

Example: `quat = quaternion(rand(3),'rotmat','frame')`

Data Types: `char` | `string`

E — Matrix of Euler angles
N-by-3 matrix

Matrix of Euler angles, specified by an *N*-by-3 matrix. If using the `'euler'` syntax, specify `E` in radians. If using the `'eulerd'` syntax, specify `E` in degrees.

Example: `quat = quaternion(E,'euler','YZY','point')`

Example: `quat = quaternion(E,'euler','XYZ','frame')`

Data Types: `single` | `double`

RS — Rotation sequence
character vector | scalar string

Rotation sequence, specified as a three-element character vector:

- 'YZY'
- 'YXY'
- 'ZYZ'
- 'ZXZ'
- 'XYX'
- 'XZX'
- 'XYZ'
- 'YZX'
- 'ZXY'
- 'XZY'
- 'ZYX'
- 'YXZ'

Assume you want to determine the new coordinates of a point when its coordinate system is rotated using frame rotation. The point is defined in the original coordinate system as:

```
point = [sqrt(2)/2,sqrt(2)/2,0];
```

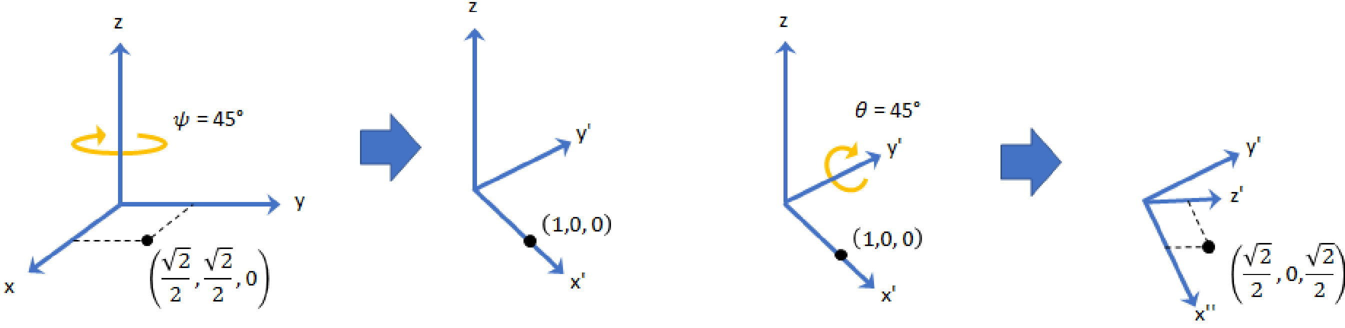
In this representation, the first column represents the *x*-axis, the second column represents the *y*-axis, and the third column represents the *z*-axis.

You want to rotate the point using the Euler angle representation [45,45,0]. Rotate the point using two different rotation sequences:

- If you create a quaternion rotator and specify the 'ZYX' sequence, the frame is first rotated 45° around the *z*-axis, then 45° around the new *y*-axis.

  ```
  quatRotator = quaternion([45,45,0],'eulerd','ZYX','frame');
  newPointCoordinate = rotateframe(quatRotator,point)
  ```

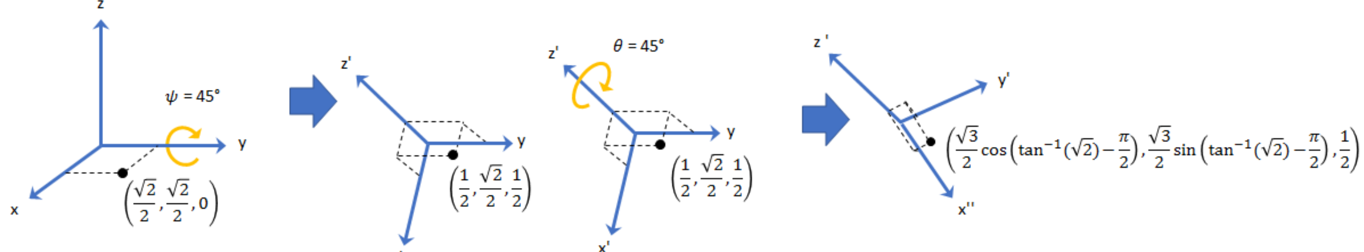
  ```
  newPointCoordinate =

      0.7071   -0.0000    0.7071
  ```



- If you create a quaternion rotator and specify the 'YZX' sequence, the frame is first rotated 45° around the *y*-axis, then 45° around the new *z*-axis.

```
quatRotator = quaternion([45,45,0],'eulerd','YZX','frame');
newPointCoordinate = rotateframe(quatRotator,point)
```

```
newPointCoordinate =

    0.8536    0.1464    0.5000
```



Data Types: `char` | `string`

`transformation` — Homogeneous transformation
`se3` object | *N*-element array of `se3` objects

Transformation, specified as an `se3` object, or as an *N*-element array of `se3` objects. *N* is the total number of transformations.

The `quaternion` object ignores the translational component of the transformation and converts the rotational 3-by-3 submatrix of the transformation to a quaternion.

`rotation` — Orthonormal rotation
`so3` object | *N*-element array of `so3` objects

Orthonormal rotation, specified as an `so3` object, or as an *N*-element array of `so3` objects. *N* is the total number of rotations.

Object Functions

angvel	Angular velocity from quaternion array
classUnderlying	Class of parts within quaternion
compact	Convert quaternion array to N-by-4 matrix
conj	Complex conjugate of quaternion
'	Complex conjugate transpose of quaternion array
dist	Angular distance in radians
euler	Convert quaternion to Euler angles (radians)
eulerd	Convert quaternion to Euler angles (degrees)
exp	Exponential of quaternion array
.\,ldivide	Element-wise quaternion left division
log	Natural logarithm of quaternion array
meanrot	Quaternion mean rotation
-	Quaternion subtraction
*	Quaternion multiplication
norm	Quaternion norm
normalize	Quaternion normalization
ones	Create quaternion array with real parts set to one and imaginary parts set to zero
parts	Extract quaternion parts

.^,power	Element-wise quaternion power
prod	Product of a quaternion array
randrot	Uniformly distributed random rotations
./,rdivide	Element-wise quaternion right division
rotateframe	Quaternion frame rotation
rotatepoint	Quaternion point rotation
rotmat	Convert quaternion to rotation matrix
rotvec	Convert quaternion to rotation vector (radians)
rotvecd	Convert quaternion to rotation vector (degrees)
slerp	Spherical linear interpolation
.*,times	Element-wise quaternion multiplication
'	Transpose a quaternion array
-	Quaternion unary minus
zeros	Create quaternion array with all parts set to zero

Examples

Create Empty Quaternion

```
quat = quaternion()
```

```
quat =

  0x0 empty quaternion array
```

By default, the underlying class of the quaternion is a double.

```
classUnderlying(quat)
```

```
ans =
'double'
```

Create Quaternion by Specifying Individual Quaternion Parts

You can create a quaternion array by specifying the four parts as comma-separated scalars, matrices, or multidimensional arrays of the same size.

Define quaternion parts as scalars.

```
A = 1.1;
B = 2.1;
C = 3.1;
D = 4.1;
quatScalar = quaternion(A,B,C,D)
```

```
quatScalar = quaternion
     1.1 + 2.1i + 3.1j + 4.1k
```

Define quaternion parts as column vectors.

```
A = [1.1;1.2];
B = [2.1;2.2];
```

```
C = [3.1;3.2];
D = [4.1;4.2];
quatVector = quaternion(A,B,C,D)
```

```
quatVector = 2x1 quaternion array
     1.1 + 2.1i + 3.1j + 4.1k
     1.2 + 2.2i + 3.2j + 4.2k
```

Define quaternion parts as matrices.

```
A = [1.1,1.3; ...
     1.2,1.4];
B = [2.1,2.3; ...
     2.2,2.4];
C = [3.1,3.3; ...
     3.2,3.4];
D = [4.1,4.3; ...
     4.2,4.4];
quatMatrix = quaternion(A,B,C,D)
```

```
quatMatrix = 2x2 quaternion array
     1.1 + 2.1i + 3.1j + 4.1k     1.3 + 2.3i + 3.3j + 4.3k
     1.2 + 2.2i + 3.2j + 4.2k     1.4 + 2.4i + 3.4j + 4.4k
```

Define quaternion parts as three dimensional arrays.

```
A = randn(2,2,2);
B = zeros(2,2,2);
C = zeros(2,2,2);
D = zeros(2,2,2);
quatMultiDimArray = quaternion(A,B,C,D)
```

```
quatMultiDimArray = 2x2x2 quaternion array
quatMultiDimArray(:,:,1) =

     0.53767 +       0i +       0j +       0k     -2.2588 +       0i +       0j +       0k
      1.8339 +       0i +       0j +       0k     0.86217 +       0i +       0j +       0k


quatMultiDimArray(:,:,2) =

     0.31877 +       0i +       0j +       0k    -0.43359 +       0i +       0j +       0k
     -1.3077 +       0i +       0j +       0k     0.34262 +       0i +       0j +       0k
```

Create Quaternion by Specifying Quaternion Parts Matrix

You can create a scalar or column vector of quaternions by specify an *N*-by-4 matrix of quaternion parts, where columns correspond to the quaternion parts A, B, C, and D.

Create a column vector of random quaternions.

```
quatParts = rand(3,4)
```

```
quatParts = 3×4
```

```
    0.8147    0.9134    0.2785    0.9649
    0.9058    0.6324    0.5469    0.1576
    0.1270    0.0975    0.9575    0.9706
```

```
quat = quaternion(quatParts)
```

```
quat = 3x1 quaternion array
     0.81472 + 0.91338i +  0.2785j + 0.96489k
     0.90579 + 0.63236i + 0.54688j + 0.15761k
     0.12699 + 0.09754i + 0.95751j + 0.97059k
```

To retrieve the `quatParts` matrix from quaternion representation, use `compact`.

```
retrievedquatParts = compact(quat)
```

```
retrievedquatParts = 3×4

    0.8147    0.9134    0.2785    0.9649
    0.9058    0.6324    0.5469    0.1576
    0.1270    0.0975    0.9575    0.9706
```

Create Quaternion by Specifying Rotation Vectors

You can create an *N*-by-1 quaternion array by specifying an *N*-by-3 matrix of rotation vectors in radians or degrees. Rotation vectors are compact spatial representations that have a one-to-one relationship with normalized quaternions.

Rotation Vectors in Radians

Create a scalar quaternion using a rotation vector and verify the resulting quaternion is normalized.

```
rotationVector = [0.3491,0.6283,0.3491];
quat = quaternion(rotationVector,'rotvec')
```

```
quat = quaternion
     0.92124 + 0.16994i + 0.30586j + 0.16994k
```

```
norm(quat)
```

```
ans = 1.0000
```

You can convert from quaternions to rotation vectors in radians using the `rotvec` function. Recover the `rotationVector` from the quaternion, `quat`.

```
rotvec(quat)
```

```
ans = 1×3

    0.3491    0.6283    0.3491
```

Rotation Vectors in Degrees

Create a scalar quaternion using a rotation vector and verify the resulting quaternion is normalized.

```
rotationVector = [20,36,20];
quat = quaternion(rotationVector,'rotvecd')
```

```
quat = quaternion
     0.92125 + 0.16993i + 0.30587j + 0.16993k
```

```
norm(quat)
```

```
ans = 1
```

You can convert from quaternions to rotation vectors in degrees using the `rotvecd` function. Recover the `rotationVector` from the quaternion, `quat`.

```
rotvecd(quat)
```

```
ans = 1×3

   20.0000   36.0000   20.0000
```

Create Quaternion by Specifying Rotation Matrices

You can create an N-by-1 quaternion array by specifying a 3-by-3-by-N array of rotation matrices. Each page of the rotation matrix array corresponds to one element of the quaternion array.

Create a scalar quaternion using a 3-by-3 rotation matrix. Specify whether the rotation matrix should be interpreted as a frame or point rotation.

```
rotationMatrix = [1 0         0; ...
                  0 sqrt(3)/2 0.5; ...
                  0 -0.5      sqrt(3)/2];
quat = quaternion(rotationMatrix,'rotmat','frame')
```

```
quat = quaternion
     0.96593 + 0.25882i +       0j +       0k
```

You can convert from quaternions to rotation matrices using the `rotmat` function. Recover the `rotationMatrix` from the quaternion, `quat`.

```
rotmat(quat,'frame')
```

```
ans = 3×3

    1.0000         0         0
         0    0.8660    0.5000
         0   -0.5000    0.8660
```

Create Quaternion by Specifying Euler Angles

You can create an *N*-by-1 quaternion array by specifying an *N*-by-3 array of Euler angles in radians or degrees.

Euler Angles in Radians

Use the `euler` syntax to create a scalar quaternion using a 1-by-3 vector of Euler angles in radians. Specify the rotation sequence of the Euler angles and whether the angles represent a frame or point rotation.

```
E = [pi/2,0,pi/4];
quat = quaternion(E,'euler','ZYX','frame')
```

```
quat = quaternion
     0.65328 +  0.2706i +  0.2706j + 0.65328k
```

You can convert from quaternions to Euler angles using the `euler` function. Recover the Euler angles, `E`, from the quaternion, `quat`.

```
euler(quat,'ZYX','frame')
```

```
ans = 1×3

    1.5708         0    0.7854
```

Euler Angles in Degrees

Use the `eulerd` syntax to create a scalar quaternion using a 1-by-3 vector of Euler angles in degrees. Specify the rotation sequence of the Euler angles and whether the angles represent a frame or point rotation.

```
E = [90,0,45];
quat = quaternion(E,'eulerd','ZYX','frame')
```

```
quat = quaternion
     0.65328 +  0.2706i +  0.2706j + 0.65328k
```

You can convert from quaternions to Euler angles in degrees using the `eulerd` function. Recover the Euler angles, `E`, from the quaternion, `quat`.

```
eulerd(quat,'ZYX','frame')
```

```
ans = 1×3

   90.0000         0   45.0000
```

Quaternion Algebra

Quaternions form a noncommutative associative algebra over the real numbers. This example illustrates the rules of quaternion algebra.

Addition and Subtraction

Quaternion addition and subtraction occur part-by-part, and are commutative:

```
Q1 = quaternion(1,2,3,4)

Q1 = quaternion
     1 + 2i + 3j + 4k

Q2 = quaternion(9,8,7,6)

Q2 = quaternion
     9 + 8i + 7j + 6k

Q1plusQ2 = Q1 + Q2

Q1plusQ2 = quaternion
     10 + 10i + 10j + 10k

Q2plusQ1 = Q2 + Q1

Q2plusQ1 = quaternion
     10 + 10i + 10j + 10k

Q1minusQ2 = Q1 - Q2

Q1minusQ2 = quaternion
    -8 - 6i - 4j - 2k

Q2minusQ1 = Q2 - Q1

Q2minusQ1 = quaternion
     8 + 6i + 4j + 2k
```

You can also perform addition and subtraction of real numbers and quaternions. The first part of a quaternion is referred to as the *real* part, while the second, third, and fourth parts are referred to as the *vector*. Addition and subtraction with real numbers affect only the real part of the quaternion.

```
Q1plusRealNumber = Q1 + 5

Q1plusRealNumber = quaternion
     6 + 2i + 3j + 4k

Q1minusRealNumber = Q1 - 5

Q1minusRealNumber = quaternion
    -4 + 2i + 3j + 4k
```

Multiplication

Quaternion multiplication is determined by the products of the basis elements and the distributive law. Recall that multiplication of the basis elements, *i*, *j*, and *k*, are not commutative, and therefore quaternion multiplication is not commutative.

```
Q1timesQ2 = Q1 * Q2
```

```
Q1timesQ2 = quaternion
    -52 + 16i + 54j + 32k
```

```
Q2timesQ1 = Q2 * Q1
```

```
Q2timesQ1 = quaternion
    -52 + 36i + 14j + 52k
```

```
isequal(Q1timesQ2,Q2timesQ1)
```

```
ans = logical
   0
```

You can also multiply a quaternion by a real number. If you multiply a quaternion by a real number, each part of the quaternion is multiplied by the real number individually:

```
Q1times5 = Q1*5
```

```
Q1times5 = quaternion
      5 + 10i + 15j + 20k
```

Multiplying a quaternion by a real number is commutative.

```
isequal(Q1*5,5*Q1)
```

```
ans = logical
   1
```

Conjugation

The complex conjugate of a quaternion is defined such that each element of the vector portion of the quaternion is negated.

```
Q1
```

```
Q1 = quaternion
     1 + 2i + 3j + 4k
```

```
conj(Q1)
```

```
ans = quaternion
     1 - 2i - 3j - 4k
```

Multiplication between a quaternion and its conjugate is commutative:

```
isequal(Q1*conj(Q1),conj(Q1)*Q1)
```

```
ans = logical
   1
```

Quaternion Array Manipulation

You can organize quaternions into vectors, matrices, and multidimensional arrays. Built-in MATLAB® functions have been enhanced to work with quaternions.

Concatenate

Quaternions are treated as individual objects during concatenation and follow MATLAB rules for array manipulation.

```
Q1 = quaternion(1,2,3,4);
Q2 = quaternion(9,8,7,6);

qVector = [Q1,Q2]
```

```
qVector = 1x2 quaternion array
     1 + 2i + 3j + 4k     9 + 8i + 7j + 6k
```

```
Q3 = quaternion(-1,-2,-3,-4);
Q4 = quaternion(-9,-8,-7,-6);

qMatrix = [qVector;Q3,Q4]
```

```
qMatrix = 2x2 quaternion array
     1 + 2i + 3j + 4k     9 + 8i + 7j + 6k
    -1 - 2i - 3j - 4k    -9 - 8i - 7j - 6k
```

```
qMultiDimensionalArray(:,:,1) = qMatrix;
qMultiDimensionalArray(:,:,2) = qMatrix
```

```
qMultiDimensionalArray = 2x2x2 quaternion array
qMultiDimensionalArray(:,:,1) =

     1 + 2i + 3j + 4k     9 + 8i + 7j + 6k
    -1 - 2i - 3j - 4k    -9 - 8i - 7j - 6k


qMultiDimensionalArray(:,:,2) =

     1 + 2i + 3j + 4k     9 + 8i + 7j + 6k
    -1 - 2i - 3j - 4k    -9 - 8i - 7j - 6k
```

Indexing

To access or assign elements in a quaternion array, use indexing.

```
qLoc2 = qMultiDimensionalArray(2)
```

```
qLoc2 = quaternion
    -1 - 2i - 3j - 4k
```

Replace the quaternion at index two with a quaternion one.

```
qMultiDimensionalArray(2) = ones('quaternion')
```

```
qMultiDimensionalArray = 2x2x2 quaternion array
qMultiDimensionalArray(:,:,1) =

     1 + 2i + 3j + 4k     9 + 8i + 7j + 6k
     1 + 0i + 0j + 0k    -9 - 8i - 7j - 6k


qMultiDimensionalArray(:,:,2) =

     1 + 2i + 3j + 4k     9 + 8i + 7j + 6k
    -1 - 2i - 3j - 4k    -9 - 8i - 7j - 6k
```

Reshape

To reshape quaternion arrays, use the `reshape` function.

```
qMatReshaped = reshape(qMatrix,4,1)
```

```
qMatReshaped = 4x1 quaternion array
     1 + 2i + 3j + 4k
    -1 - 2i - 3j - 4k
     9 + 8i + 7j + 6k
    -9 - 8i - 7j - 6k
```

Transpose

To transpose quaternion vectors and matrices, use the `transpose` function.

```
qMatTransposed = transpose(qMatrix)
```

```
qMatTransposed = 2x2 quaternion array
     1 + 2i + 3j + 4k    -1 - 2i - 3j - 4k
     9 + 8i + 7j + 6k    -9 - 8i - 7j - 6k
```

Permute

To permute quaternion vectors, matrices, and multidimensional arrays, use the `permute` function.

```
qMultiDimensionalArray
```

```
qMultiDimensionalArray = 2x2x2 quaternion array
qMultiDimensionalArray(:,:,1) =

     1 + 2i + 3j + 4k     9 + 8i + 7j + 6k
     1 + 0i + 0j + 0k    -9 - 8i - 7j - 6k


qMultiDimensionalArray(:,:,2) =
```

```
     1 + 2i + 3j + 4k     9 + 8i + 7j + 6k
    -1 - 2i - 3j - 4k    -9 - 8i - 7j - 6k
```

```
qMatPermute = permute(qMultiDimensionalArray,[3,1,2])
```

```
qMatPermute = 2x2x2 quaternion array
qMatPermute(:,:,1) =

     1 + 2i + 3j + 4k     1 + 0i + 0j + 0k
     1 + 2i + 3j + 4k    -1 - 2i - 3j - 4k


qMatPermute(:,:,2) =

     9 + 8i + 7j + 6k    -9 - 8i - 7j - 6k
     9 + 8i + 7j + 6k    -9 - 8i - 7j - 6k
```

Version History

Introduced in R2018a

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

se3 | so3

rateControl

Execute loop at fixed frequency

Description

The rateControl object enables you to run a loop at a fixed frequency. It also collects statistics about the timing of the loop iterations. Use waitfor in the loop to pause code execution until the next time step. The loop operates every DesiredPeriod seconds, unless the enclosed code takes longer to operate. The object uses the OverrunAction property to determine how it handles longer loop operation times. The default setting, 'slip', immediately executes the loop if LastPeriod is greater than DesiredPeriod. Using 'drop' causes the waitfor method to wait until the next multiple of DesiredPeriod is reached to execute the next loop.

Tip The scheduling resolution of your operating system and the level of other system activity can affect rate execution accuracy. As a result, accurate rate timing is limited to 100 Hz for execution of MATLAB code. To improve performance and execution speeds, use code generation.

Creation

Syntax

```
rateObj = rateControl(desiredRate)
```

Description

rateObj = rateControl(desiredRate) creates an object that operates loops at a fixed-rate based on your system time and directly sets the DesireRate property.

Properties

DesiredRate — Desired execution rate
scalar

Desired execution rate of loop, specified as a scalar in Hz. When using waitfor, the loop operates every DesiredRate seconds, unless the loop takes longer. It then begins the next loop based on the specified OverrunAction.

DesiredPeriod — Desired time period between executions
scalar

Desired time period between executions, specified as a scalar in seconds. This property is equal to the inverse of DesiredRate.

TotalElapsedTime — Elapsed time since construction or reset
scalar

Elapsed time since construction or reset, specified as a scalar in seconds.

LastPeriod — Elapsed time between last two calls to waitfor

NaN (default) | scalar

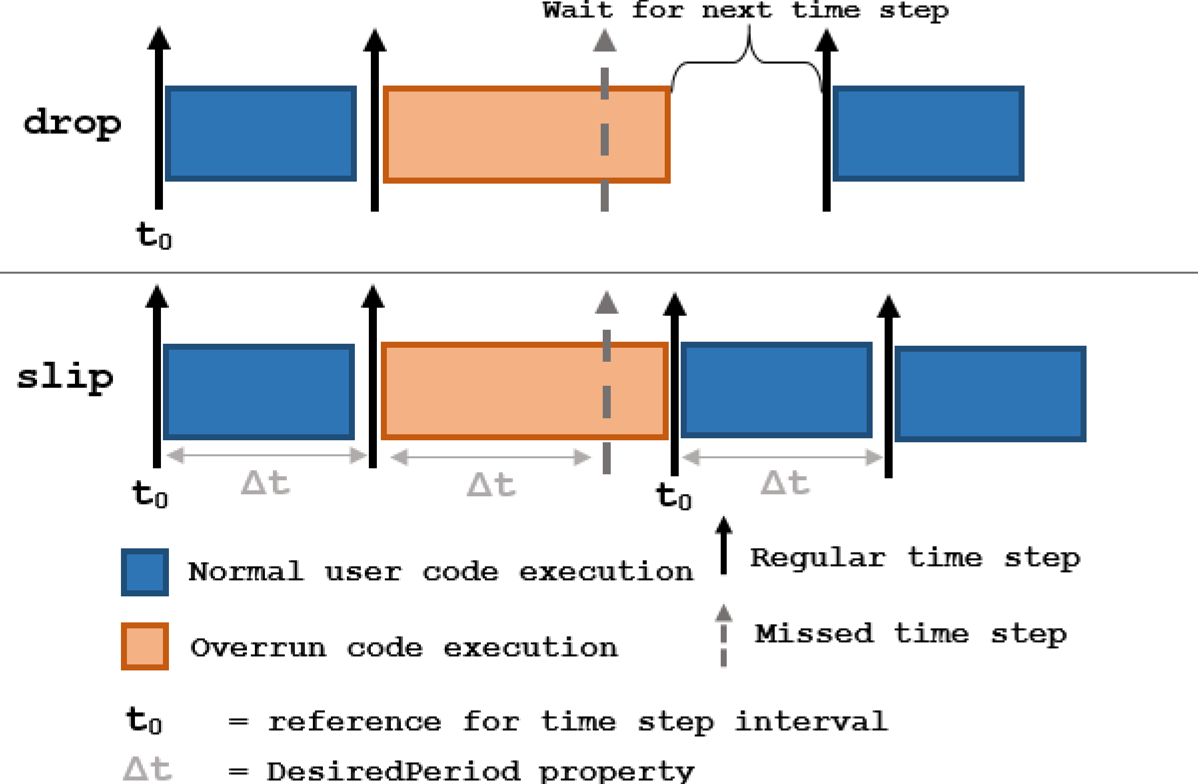
Elapsed time between last two calls to waitfor, specified as a scalar. By default, LastPeriod is set to NaN until waitfor is called for the first time. After the first call, LastPeriod equals TotalElapsedTime.

OverrunAction — Method for handling overruns

'slip' (default) | 'drop'

Method for handling overruns, specified as one of these character vectors:

- 'drop' — waits until the next time interval equal to a multiple of DesiredPeriod
- 'slip' — immediately executes the loop again



Each code section calls waitfor at the end of execution.

Object Functions

waitfor	Pause code execution to achieve desired execution rate
statistics	Statistics of past execution periods
reset	Reset Rate object

Examples

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: 0.003114
Iteration: 2 - Time Elapsed: 1.003618
Iteration: 3 - Time Elapsed: 2.000382
Iteration: 4 - Time Elapsed: 3.000483
Iteration: 5 - Time Elapsed: 4.001145
Iteration: 6 - Time Elapsed: 5.011602
Iteration: 7 - Time Elapsed: 6.005804
Iteration: 8 - Time Elapsed: 7.000292
Iteration: 9 - Time Elapsed: 8.000162
Iteration: 10 - Time Elapsed: 9.006431
```

Each iteration executes at a 1-second interval.

Get Statistics From Rate Object Execution

Create a `rateControl` object for running at 20 Hz.

```
r = rateControl(20);
```

Start a loop and control operation using the `rateControl` object.

```
for i = 1:30
    % Your code goes here
    waitfor(r);
end
```

Get `Rate` object statistics after loop operation.

```
stats = statistics(r)
```

```
stats = struct with fields:
              Periods: [0.0599 0.0418 0.0629 0.0449 0.0456 0.0591 0.0449 0.0461 0.0599 0.0449 0.0
           NumPeriods: 30
        AveragePeriod: 0.0501
    StandardDeviation: 0.0079
          NumOverruns: 0
```

Run Loop At Fixed Rate and Reset Rate Object

Create a `rateControl` object for running at 20 Hz.

```
r = rateControl(2);
```

Start a loop and control operation using the `Rate` object.

```
for i = 1:30
    % Your code goes here
    waitfor(r);
end
```

Display the `rateControl` object properties after loop operation.

```
disp(r)
```

```
  rateControl with properties:

         DesiredRate: 2
       DesiredPeriod: 0.5000
       OverrunAction: 'slip'
    TotalElapsedTime: 15.0287
          LastPeriod: 0.5118
```

Reset the object to restart the time statistics.

```
reset(r);
disp(r)
```

```
  rateControl with properties:

         DesiredRate: 2
       DesiredPeriod: 0.5000
       OverrunAction: 'slip'
    TotalElapsedTime: 0.0026
          LastPeriod: NaN
```

Version History

Introduced in R2016a

R2019b: `rateControl` was renamed

Behavior change in future release

The `rateControl` object was renamed from `robotics.Rate`. Use `rateControl` for all object creation.

See Also

`rosrate` | `waitfor` | `statistics` | `reset`

Topics

"Execute Code at a Fixed-Rate"

resamplingPolicyPF

Create resampling policy object with resampling settings

Description

The `resamplingPolicyPF` object stores settings for when resampling should occur when using a particle filter for state estimation. The object contains the method that triggers resampling and the relevant threshold for this resampling. Use this object as the `ResamplingPolicy` property of the `stateEstimatorPF` object.

Creation

Syntax

```
policy = resamplingPolicyPF
```

Description

`policy = resamplingPolicyPF` creates a `resamplingPolicyPF` object `policy`, which contains properties to be modified to control when resampling should be triggered. Use this object as the `ResamplingPolicy` property of the `stateEstimatorPF` object.

Properties

`TriggerMethod` — Method for determining if resampling should occur
`'ratio'` (default) | character vector

Method for determining if resampling should occur, specified as a character vector. Possible choices are `'ratio'` and `'interval'`. The `'interval'` method triggers resampling at regular intervals of operating the particle filter. The `'ratio'` method triggers resampling based on the ratio of effective total particles.

`SamplingInterval` — Fixed interval between resampling
1 (default) | scalar

Fixed interval between resampling, specified as a scalar. This interval determines during which correction steps the resampling is executed. For example, a value of 2 means the resampling is executed every second correction step. A value of `inf` means that resampling is never executed.

This property only applies with the `TriggerMethod` is set to `'interval'`.

`MinEffectiveParticleRatio` — Minimum desired ratio of effective to total particles
0.5 (default) | scalar

Minimum desired ratio of effective to total particles, specified as a scalar. The effective number of particles is a measure of how well the current set of particles approximates the posterior distribution. A lower effective particle ratio means less particles are contributing to the estimation and resampling

might be required. If the ratio of effective particles to total particles falls below the `MinEffectiveParticleRatio`, a resampling step is triggered.

Version History

Introduced in R2019b

See Also

`stateEstimatorPF` | `correct`

Topics

"Track a Car-Like Robot Using Particle Filter"

rigidBody

Create a rigid body

Description

The `rigidBody` object represents a rigid body. A rigid body is the building block for any tree-structured robot manipulator. Each `rigidBody` has a `rigidBodyJoint` object attached to it that defines how the rigid body can move. Rigid bodies are assembled into a tree-structured robot model using `rigidBodyTree`.

Set a joint object to the `Joint` property before calling `addBody` to add the rigid body to the robot model. When a rigid body is in a rigid body tree, you cannot directly modify its properties because it corrupts the relationships between bodies. Use `replaceJoint` to modify the entire tree structure.

Creation

Syntax

```
body = rigidBody(name)
```

Description

`body = rigidBody(name)` creates a rigid body with the specified name. By default, the body comes with a fixed joint.

Input Arguments

name — Name of rigid body
string scalar | character vector

Name of the rigid body, specified as a string scalar or character vector. This name must be unique to the body so that it can be accessed in a `rigidBodyTree` object.

Properties

Name — Name of rigid body
string scalar | character vector

Name of the rigid body, specified as a string scalar or character vector. This name must be unique to the body so that it can be found in a `rigidBodyTree` object.

Data Types: `char` | `string`

Joint — `rigidBodyJoint` object
handle

`rigidBodyJoint` object, specified as a handle. By default, the joint is `'fixed'` type.

Mass — Mass of rigid body
1 kg (default) | numeric scalar

Mass of rigid body, specified as a numeric scalar in kilograms.

CenterOfMass — Center of mass position of rigid body
[0 0 0] m (default) | [x y z] vector

Center of mass position of 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
[1 1 1 0 0 0] kg•m^2 (default) | [Ixx Iyy Izz Iyz Ixz Ixy] vector

Inertia of rigid body, specified as a [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. The last three elements are the off-diagonal elements of the inertia tensor. The inertia tensor is a positive semi-definite symmetric matrix:

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

Parent — Rigid body parent
rigidBody object handle

Rigid body parent, specified as a rigidBody object handle. The rigid body joint defines how this body can move relative to the parent. This property is empty until the rigid body is added to a rigidBodyTree robot model.

Children — Rigid body children
cell array of rigidBody object handles

Rigid body children, specified as a cell array of rigidBody object handles. These rigid body children are all attached to this rigid body object. This property is empty until the rigid body is added to a rigidBodyTree robot model, and at least one other body is added to the tree with this body as its parent.

Visuals — Visual geometries
cell array of string scalars | cell array of character vectors

Visual geometries, specified as a cell array of string scalars or character vectors. Each character vector describes a type and source of a visual geometry. For example, if a mesh file, link_0.stl, is attached to the rigid body, the visual would be Mesh:link_0.stl. Visual geometries are added to the rigid body using addVisual.

Collisions — Collision geometries
cell array of character vectors

Collision geometries, specified as a cell array of character vectors. Each character vector describes the type of collision object and relevant parameters of the collision geometry. To modify the collision geometries for a rigid body, use the addCollision and clearCollision functions.

Object Functions

copy	Create a deep copy of rigid body
addCollision	Add collision geometry to rigid body
addVisual	Add visual geometry data to rigid body
clearCollision	Clear all attached collision geometries
clearVisual	Clear all visual geometries

Examples

Attach Rigid Body and Joint to Rigid Body Tree

Add a rigid body and corresponding joint to a rigid body tree. Each `rigidBody` object contains a `rigidBodyJoint` object and must be added to the `rigidBodyTree` using `addBody`.

Create a rigid body tree.

```
rbtree = rigidBodyTree;
```

Create a rigid body with a unique name.

```
body1 = rigidBody('b1');
```

Create a revolute joint. By default, the `rigidBody` object comes with a fixed joint. Replace the joint by assigning a new `rigidBodyJoint` object to the `body1.Joint` property.

```
jnt1 = rigidBodyJoint('jnt1','revolute');
body1.Joint = jnt1;
```

Add the rigid body to the tree. Specify the body name that you are attaching the rigid body to. Because this is the first body, use the base name of the tree.

```
basename = rbtree.BaseName;
addBody(rbtree,body1,basename)
```

Use `showdetails` on the tree to confirm the rigid body and joint were added properly.

```
showdetails(rbtree)
```

```
--------------------
Robot: (1 bodies)

 Idx    Body Name   Joint Name   Joint Type    Parent Name(Idx)   Children Name(s)
 ---    ---------   ----------   ----------    ----------------   ----------------
   1           b1         jnt1     revolute             base(0)   
--------------------
```

Build Manipulator Robot Using Denavit-Hartenberg Parameters

Use the Denavit-Hartenberg (DH) parameters of the Puma560® robot to build a robot. Each rigid body is added one at a time, with the child-to-parent transform specified by the joint object.

The DH parameters define the geometry of the robot with relation to how each rigid body is attached to its parent. For convenience, setup the parameters for the Puma560 robot in a matrix[1] on page 1-

314. The Puma robot is a serial chain manipulator. The DH parameters are relative to the previous row in the matrix, corresponding to the previous joint attachment.

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

Create a rigid body tree object to build the robot.

```
robot = rigidBodyTree;
```

Create the first rigid body and add it to the robot. To add a rigid body:

1. Create a `rigidBody` object and give it a unique name.
2. Create a `rigidBodyJoint` object and give it a unique name.
3. Use `setFixedTransform` to specify the body-to-body transformation using DH parameters. The last element of the DH parameters, `theta`, is ignored because the angle is dependent on the joint position.
4. Call `addBody` to attach the first body joint to the base frame of the robot.

```
body1 = rigidBody('body1');
jnt1 = rigidBodyJoint('jnt1','revolute');

setFixedTransform(jnt1,dhparams(1,:),'dh');
body1.Joint = jnt1;

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

Create and add other rigid bodies to the robot. Specify the previous body name when calling `addBody` to attach it. Each fixed transform is relative to the previous joint coordinate frame.

```
body2 = rigidBody('body2');
jnt2 = rigidBodyJoint('jnt2','revolute');
body3 = rigidBody('body3');
jnt3 = rigidBodyJoint('jnt3','revolute');
body4 = rigidBody('body4');
jnt4 = rigidBodyJoint('jnt4','revolute');
body5 = rigidBody('body5');
jnt5 = rigidBodyJoint('jnt5','revolute');
body6 = rigidBody('body6');
jnt6 = rigidBodyJoint('jnt6','revolute');

setFixedTransform(jnt2,dhparams(2,:),'dh');
setFixedTransform(jnt3,dhparams(3,:),'dh');
setFixedTransform(jnt4,dhparams(4,:),'dh');
setFixedTransform(jnt5,dhparams(5,:),'dh');
setFixedTransform(jnt6,dhparams(6,:),'dh');

body2.Joint = jnt2;
body3.Joint = jnt3;
body4.Joint = jnt4;
body5.Joint = jnt5;
body6.Joint = jnt6;
```

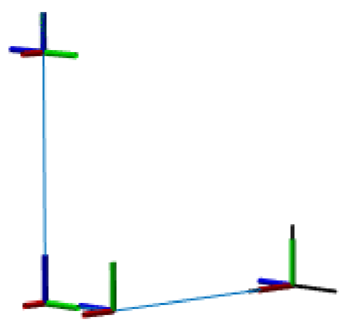
```
addBody(robot,body2,'body1')
addBody(robot,body3,'body2')
addBody(robot,body4,'body3')
addBody(robot,body5,'body4')
addBody(robot,body6,'body5')
```

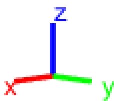
Verify that your robot was built properly by using the `showdetails` or `show` function. `showdetails` lists all the bodies in the MATLAB® command window. `show` displays the robot with a given configuration (home by default). Calls to `axis` modify the axis limits and hide the axis labels.

```
showdetails(robot)
```

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

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

```
show(robot);
axis([-0.5,0.5,-0.5,0.5,-0.5,0.5])
axis off
```





References

[1] Corke, P. I., and B. Armstrong-Helouvry. "A Search for Consensus among Model Parameters Reported for the PUMA 560 Robot." *Proceedings of the 1994 IEEE International Conference on Robotics and Automation*, IEEE Computer. Soc. Press, 1994, pp. 1608–13. *DOI.org (Crossref)*, doi:10.1109/ROBOT.1994.351360.

Version History

Introduced in R2016b

R2019b: `rigidBody` was renamed

Behavior change in future release

The `rigidBody` object was renamed from `robotics.RigidBody`. Use `rigidBody` for all object creation.

References

[1] Craig, John J. *Introduction to Robotics: Mechanics and Control*. Reading, MA: Addison-Wesley, 1989.

[2] Siciliano, Bruno. Robotics: Modelling, Planning and Control. London: Springer, 2009.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also
`rigidBodyJoint` | `rigidBodyTree` | `addBody` | `replaceJoint` | `addCollision` | `addVisual` | `clearCollision` | `clearVisual`

Topics
"Build a Robot Step by Step"
"Rigid Body Tree Robot Model"

rigidBodyJoint

Create a joint

Description

The `rigidBodyJoint` objects defines how a rigid body moves relative to an attachment point. In a tree-structured robot, a joint always belongs to a specific rigid body, and each rigid body has one joint.

The `rigidBodyJoint` object can describe joints of various types. When building a rigid body tree structure with `rigidBodyTree`, you must assign the `Joint` object to a rigid body using the `rigidBody` class.

The different joint types supported are:

- `fixed` — Fixed joint that prevents relative motion between two bodies.
- `revolute` — Single degree of freedom (DOF) joint that rotates around a given axis. Also called a pin or hinge joint.
- `prismatic` — Single DOF joint that slides along a given axis. Also called a sliding joint.

Each joint type has different properties with different dimensions, depending on its defined geometry.

Creation

Syntax

```
jointObj = rigidBodyJoint(jname)
jointObj = rigidBodyJoint(jname,jtype)
```

Description

`jointObj = rigidBodyJoint(jname)` creates a fixed joint with the specified name.

`jointObj = rigidBodyJoint(jname,jtype)` creates a joint of the specified type with the specified name.

Input Arguments

`jname` — Joint name
string scalar | character vector

Joint name, specified as a string scalar or character vector. The joint name must be unique to access it off the rigid body tree.

Example: `"elbow_right"`

Data Types: `char` | `string`

`jtype` — Joint type
`'fixed'` (default) | string scalar | character vector

Joint type, specified as a string scalar or character vector. The joint type predefines certain properties when creating the joint.

The different joint types supported are:

- `fixed` — Fixed joint that prevents relative motion between two bodies.
- `revolute` — Single degree of freedom (DOF) joint that rotates around a given axis. Also called a pin or hinge joint.
- `prismatic` — Single DOF joint that slides along a given axis. Also called a sliding joint.

Example: `"prismatic"`

Data Types: `char` | `string`

Properties

`Type` — Joint type
`'fixed'` (default) | string scalar | character vector

This property is read-only.

Joint type, returned as a string scalar or character vector. The joint type predefines certain properties when creating the joint.

The different joint types supported are:

- `fixed` — Fixed joint that prevents relative motion between two bodies.
- `revolute` — Single degree of freedom (DOF) joint that rotates around a given axis. Also called a pin or hinge joint.
- `prismatic` — Single DOF joint that slides along a given axis. Also called a sliding joint.

If the rigid body that contains this joint is added to a robot model, the joint type must be changed by replacing the joint using `replaceJoint`.

Example: `"prismatic"`

Data Types: `char` | `string`

`Name` — Joint name
string scalar | character vector

Joint name, returned as a string scalar or character vector. The joint name must be unique to access it off the rigid body tree. If the rigid body that contains this joint is added to a robot model, the joint name must be changed by replacing the joint using `replaceJoint`.

Example: `"elbow_right"`

Data Types: `char` | `string`

`PositionLimits` — Position limits of joint
vector

Position limits of the joint, specified as a vector of `[min max]` values. Depending on the type of joint, these values have different definitions.

- `fixed` — `[NaN NaN]` (default). A fixed joint has no joint limits. Bodies remain fixed between each other.
- `revolute` — `[-pi pi]` (default). The limits define the angle of rotation around the axis in radians.
- `prismatic` — `[-0.5 0.5]` (default). The limits define the linear motion along the axis in meters.

Example: `[-pi/2, pi/2]`

`HomePosition` — Home position of joint
scalar

Home position of joint, specified as a scalar that depends on your joint type. The home position must fall in the range set by `PositionLimits`. This property is used by `homeConfiguration` to generate the predefined home configuration for an entire rigid body tree.

Depending on the joint type, the home position has a different definition.

- `fixed` — `0` (default). A fixed joint has no relevant home position.
- `revolute` — `0` (default). A revolute joint has a home position defined by the angle of rotation around the joint axis in radians.
- `prismatic` — `0` (default). A prismatic joint has a home position defined by the linear motion along the joint axis in meters.

Example: `pi/2` radians for a `revolute` joint

`JointAxis` — Axis of motion for joint
[NaN NaN NaN] (default) | three-element unit vector

Axis of motion for joint, specified as a three-element unit vector. The vector can be any direction in 3-D space in local coordinates.

Depending on the joint type, the joint axis has a different definition.

- `fixed` — A fixed joint has no relevant axis of motion.
- `revolute` — A revolute joint rotates the body in the plane perpendicular to the joint axis.
- `prismatic` — A prismatic joint moves the body in a linear motion along the joint axis direction.

Example: `[1 0 0]` for motion around the *x*-axis for a `revolute` joint

`JointToParentTransform` — Fixed transform from joint to parent frame
`eye(4)` (default) | 4-by-4 homogeneous transform matrix

This property is read-only.

Fixed transform from joint to parent frame, returned as a 4-by-4 homogeneous transform matrix. The transform converts the coordinates of points in the joint predecessor frame to the parent body frame.

Example: `eye(4)`

`ChildToJointTransform` — Fixed transform from child body to joint frame
`eye(4)` (default) | 4-by-4 homogeneous transform matrix

This property is read-only.

Fixed transform from child body to joint frame, returned as a 4-by-4 homogeneous transform matrix. The transform converts the coordinates of points in the child body frame to the joint successor frame.

Example: `eye(4)`

Object Functions

copy	Create copy of joint
setFixedTransform	Set fixed transform properties of joint

Examples

Attach Rigid Body and Joint to Rigid Body Tree

Add a rigid body and corresponding joint to a rigid body tree. Each `rigidBody` object contains a `rigidBodyJoint` object and must be added to the `rigidBodyTree` using `addBody`.

Create a rigid body tree.

```
rbtree = rigidBodyTree;
```

Create a rigid body with a unique name.

```
body1 = rigidBody('b1');
```

Create a revolute joint. By default, the `rigidBody` object comes with a fixed joint. Replace the joint by assigning a new `rigidBodyJoint` object to the `body1.Joint` property.

```
jnt1 = rigidBodyJoint('jnt1','revolute');
body1.Joint = jnt1;
```

Add the rigid body to the tree. Specify the body name that you are attaching the rigid body to. Because this is the first body, use the base name of the tree.

```
basename = rbtree.BaseName;
addBody(rbtree,body1,basename)
```

Use `showdetails` on the tree to confirm the rigid body and joint were added properly.

```
showdetails(rbtree)
```

```
--------------------
Robot: (1 bodies)

 Idx    Body Name   Joint Name   Joint Type    Parent Name(Idx)   Children Name(s)
 ---    ---------   ----------   ----------    ----------------   ----------------
   1           b1         jnt1     revolute             base(0)
--------------------
```

Build Manipulator Robot Using Denavit-Hartenberg Parameters

Use the Denavit-Hartenberg (DH) parameters of the Puma560® robot to build a robot. Each rigid body is added one at a time, with the child-to-parent transform specified by the joint object.

The DH parameters define the geometry of the robot with relation to how each rigid body is attached to its parent. For convenience, setup the parameters for the Puma560 robot in a matrix[1] on page 1-322. The Puma robot is a serial chain manipulator. The DH parameters are relative to the previous row in the matrix, corresponding to the previous joint attachment.

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

Create a rigid body tree object to build the robot.

```
robot = rigidBodyTree;
```

Create the first rigid body and add it to the robot. To add a rigid body:

1. Create a `rigidBody` object and give it a unique name.
2. Create a `rigidBodyJoint` object and give it a unique name.
3. Use `setFixedTransform` to specify the body-to-body transformation using DH parameters. The last element of the DH parameters, `theta`, is ignored because the angle is dependent on the joint position.
4. Call `addBody` to attach the first body joint to the base frame of the robot.

```
body1 = rigidBody('body1');
jnt1 = rigidBodyJoint('jnt1','revolute');

setFixedTransform(jnt1,dhparams(1,:),'dh');
body1.Joint = jnt1;

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

Create and add other rigid bodies to the robot. Specify the previous body name when calling `addBody` to attach it. Each fixed transform is relative to the previous joint coordinate frame.

```
body2 = rigidBody('body2');
jnt2 = rigidBodyJoint('jnt2','revolute');
body3 = rigidBody('body3');
jnt3 = rigidBodyJoint('jnt3','revolute');
body4 = rigidBody('body4');
jnt4 = rigidBodyJoint('jnt4','revolute');
body5 = rigidBody('body5');
jnt5 = rigidBodyJoint('jnt5','revolute');
body6 = rigidBody('body6');
jnt6 = rigidBodyJoint('jnt6','revolute');

setFixedTransform(jnt2,dhparams(2,:),'dh');
setFixedTransform(jnt3,dhparams(3,:),'dh');
setFixedTransform(jnt4,dhparams(4,:),'dh');
setFixedTransform(jnt5,dhparams(5,:),'dh');
```

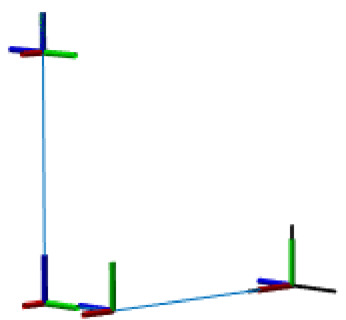
```
setFixedTransform(jnt6,dhparams(6,:),'dh');

body2.Joint = jnt2;
body3.Joint = jnt3;
body4.Joint = jnt4;
body5.Joint = jnt5;
body6.Joint = jnt6;

addBody(robot,body2,'body1')
addBody(robot,body3,'body2')
addBody(robot,body4,'body3')
addBody(robot,body5,'body4')
addBody(robot,body6,'body5')
```

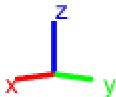
Verify that your robot was built properly by using the `showdetails` or `show` function. `showdetails` lists all the bodies in the MATLAB® command window. `show` displays the robot with a given configuration (home by default). Calls to `axis` modify the axis limits and hide the axis labels.

```
showdetails(robot)
```

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

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

```
show(robot);
axis([-0.5,0.5,-0.5,0.5,-0.5,0.5])
axis off
```





References

[1] Corke, P. I., and B. Armstrong-Helouvry. “A Search for Consensus among Model Parameters Reported for the PUMA 560 Robot.” *Proceedings of the 1994 IEEE International Conference on Robotics and Automation*, IEEE Computer. Soc. Press, 1994, pp. 1608–13. *DOI.org (Crossref)*, doi:10.1109/ROBOT.1994.351360.

Modify a Robot Rigid Body Tree Model

Make changes to an existing `rigidBodyTree` object. You can get replace joints, bodies and subtrees in the rigid body tree.

Load example robots as `rigidBodyTree` objects.

```
load exampleRobots.mat
```

View the details of the Puma robot using `showdetails`.

```
showdetails(puma1)
```

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

```
 Idx     Body Name    Joint Name    Joint Type     Parent Name(Idx)   Children Name(s)
 ---     ---------    ----------    ----------     ----------------   ----------------
   1            L1          jnt1      revolute              base(0)   L2(2)
   2            L2          jnt2      revolute                L1(1)   L3(3)
   3            L3          jnt3      revolute                L2(2)   L4(4)
   4            L4          jnt4      revolute                L3(3)   L5(5)
   5            L5          jnt5      revolute                L4(4)   L6(6)
   6            L6          jnt6      revolute                L5(5)
--------------------
```

Get a specific body to inspect the properties. The only child of the `L3` body is the `L4` body. You can copy a specific body as well.

```
body3 = getBody(puma1,'L3');
childBody = body3.Children{1}
```

```
childBody =
  rigidBody with properties:

            Name: 'L4'
           Joint: [1x1 rigidBodyJoint]
            Mass: 1
    CenterOfMass: [0 0 0]
         Inertia: [1 1 1 0 0 0]
          Parent: [1x1 rigidBody]
        Children: {[1x1 rigidBody]}
         Visuals: {}
      Collisions: {}
```

```
body3Copy = copy(body3);
```

Replace the joint on the `L3` body. You must create a new `Joint` object and use `replaceJoint` to ensure the downstream body geometry is unaffected. Call `setFixedTransform` if necessary to define a transform between the bodies instead of with the default identity matrices.

```
newJoint = rigidBodyJoint('prismatic');
replaceJoint(puma1,'L3',newJoint);

showdetails(puma1)
```

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

 Idx     Body Name        Joint Name        Joint Type     Parent Name(Idx)   Children Name(s)
 ---     ---------        ----------        ----------     ----------------   ----------------
   1            L1              jnt1          revolute              base(0)   L2(2)
   2            L2              jnt2          revolute                L1(1)   L3(3)
   3            L3         prismatic             fixed                L2(2)   L4(4)
   4            L4              jnt4          revolute                L3(3)   L5(5)
   5            L5              jnt5          revolute                L4(4)   L6(6)
   6            L6              jnt6          revolute                L5(5)
--------------------
```

Remove an entire body and get the resulting subtree using `removeBody`. The removed body is included in the subtree.

```
subtree = removeBody(puma1,'L4')
```

```
subtree = 
  rigidBodyTree with properties:

     NumBodies: 3
        Bodies: {[1x1 rigidBody]  [1x1 rigidBody]  [1x1 rigidBody]}
          Base: [1x1 rigidBody]
     BodyNames: {'L4'  'L5'  'L6'}
      BaseName: 'L3'
       Gravity: [0 0 0]
    DataFormat: 'struct'
```

Remove the modified `L3` body. Add the original copied `L3` body to the `L2` body, followed by the returned subtree. The robot model remains the same. See a detailed comparison through `showdetails`.

```
removeBody(puma1,'L3');
addBody(puma1,body3Copy,'L2')
addSubtree(puma1,'L3',subtree)

showdetails(puma1)
```

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

 Idx    Body Name   Joint Name   Joint Type    Parent Name(Idx)   Children Name(s)
 ---    ---------   ----------   ----------    ----------------   ----------------
   1           L1         jnt1     revolute             base(0)   L2(2)  
   2           L2         jnt2     revolute               L1(1)   L3(3)  
   3           L3         jnt3     revolute               L2(2)   L4(4)  
   4           L4         jnt4     revolute               L3(3)   L5(5)  
   5           L5         jnt5     revolute               L4(4)   L6(6)  
   6           L6         jnt6     revolute               L5(5)   
--------------------
```

Version History

Introduced in R2016b

R2019b: `rigidBodyJoint` was renamed

Behavior change in future release

The `rigidBodyJoint` object was renamed from `robotics.Joint`. Use `rigidBodyJoint` for all object creation.

References

[1] Craig, John J. *Introduction to Robotics: Mechanics and Control*. Reading, MA: Addison-Wesley, 1989.

[2] Siciliano, Bruno. *Robotics: Modelling, Planning and Control*. London: Springer, 2009.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also
`rigidBody` | `rigidBodyTree`

Topics
"Build a Robot Step by Step"
"Rigid Body Tree Robot Model"

rigidBodyTree

Create tree-structured robot

Description

The `rigidBodyTree` is a representation of the connectivity of rigid bodies with joints. Use this class to build robot manipulator models in MATLAB. If you have a robot model specified using the Unified Robot Description Format (URDF), use `importrobot` to import your robot model.

A rigid body tree model is made up of rigid bodies as `rigidBody` objects. Each rigid body has a `rigidBodyJoint` object associated with it that defines how it can move relative to its parent body. Use `setFixedTransform` to define the fixed transformation between the frame of a joint and the frame of one of the adjacent bodies. You can add, replace, or remove rigid bodies from the model using the methods of the `RigidBodyTree` class.

Robot dynamics calculations are also possible. Specify the `Mass`, `CenterOfMass`, and `Inertia` properties for each `rigidBody` in the robot model. You can calculate forward and inverse dynamics with or without external forces and compute dynamics quantities given robot joint motions and joint inputs. To use the dynamics-related functions, set the `DataFormat` property to `"row"` or `"column"`.

For a given rigid body tree model, you can also use the robot model to calculate joint angles for desired end-effector positions using the robotics inverse kinematics algorithms. Specify your rigid body tree model when using `inverseKinematics` or `generalizedInverseKinematics`.

The `show` method supports visualization of body meshes. Meshes are specified as `.stl` files and can be added to individual rigid bodies using `addVisual`. Also, by default, the `importrobot` function loads all the accessible `.stl` files specified in your URDF robot model.

Creation

Syntax

```
robot = rigidBodyTree
robot = rigidBodyTree("MaxNumBodies",N,"DataFormat",dataFormat)
```

Description

`robot = rigidBodyTree` creates a tree-structured robot object. Add rigid bodies to it using `addBody`.

`robot = rigidBodyTree("MaxNumBodies",N,"DataFormat",dataFormat)` specifies an upper bound on the number of bodies allowed in the robot when generating code. You must also specify the `DataFormat` property as a name-value pair.

Properties

`NumBodies` — Number of bodies
integer

This property is read-only.

Number of bodies in the robot model (not including the base), returned as an integer.

Bodies — List of rigid bodies
cell array of handles

This property is read-only.

List of rigid bodies in the robot model, returned as a cell array of handles. Use this list to access specific `RigidBody` objects in the model. You can also call `getBody` to get a body by its name.

BodyNames — Names of rigid bodies
cell array of string scalars | cell array of character vectors

This property is read-only.

Names of rigid bodies, returned as a cell array of character vectors.

BaseName — Name of robot base
`'base'` (default) | string scalar | character vector

Name of robot base, returned as a string scalar or character vector.

Gravity — Gravitational acceleration experienced by robot
`[0 0 0]` m/s^2 (default) | `[x y z]` vector

Gravitational acceleration experienced by robot, specified as an `[x y z]` vector in meters per second squared. Each element corresponds to the acceleration of the base robot frame in that direction.

DataFormat — Input/output data format for kinematics and dynamics functions
`"struct"` (default) | `"row"` | `"column"`

Input/output data format for kinematics and dynamics functions, specified as `"struct"`, `"row"`, or `"column"`. To use dynamics functions, you must use either `"row"` or `"column"`.

Object Functions

addBody	Add body to robot
addSubtree	Add subtree to robot
centerOfMass	Center of mass position and Jacobian
checkCollision	Check if robot is in collision
copy	Copy robot model
externalForce	Compose external force matrix relative to base
forwardDynamics	Joint accelerations given joint torques and states
geometricJacobian	Geometric Jacobian for robot configuration
gravityTorque	Joint torques that compensate gravity
getBody	Get robot body handle by name
getTransform	Get transform between body frames
homeConfiguration	Get home configuration of robot
inverseDynamics	Required joint torques for given motion
massMatrix	Joint-space mass matrix
randomConfiguration	Generate random configuration of robot
removeBody	Remove body from robot

replaceBody	Replace body on robot
replaceJoint	Replace joint on body
show	Show robot model in figure
showdetails	Show details of robot model
subtree	Create subtree from robot model
velocityProduct	Joint torques that cancel velocity-induced forces
writeAsFunction	Create rigidBodyTree code generating function

Examples

Attach Rigid Body and Joint to Rigid Body Tree

Add a rigid body and corresponding joint to a rigid body tree. Each `rigidBody` object contains a `rigidBodyJoint` object and must be added to the `rigidBodyTree` using `addBody`.

Create a rigid body tree.

```
rbtree = rigidBodyTree;
```

Create a rigid body with a unique name.

```
body1 = rigidBody('b1');
```

Create a revolute joint. By default, the `rigidBody` object comes with a fixed joint. Replace the joint by assigning a new `rigidBodyJoint` object to the `body1.Joint` property.

```
jnt1 = rigidBodyJoint('jnt1','revolute');
body1.Joint = jnt1;
```

Add the rigid body to the tree. Specify the body name that you are attaching the rigid body to. Because this is the first body, use the base name of the tree.

```
basename = rbtree.BaseName;
addBody(rbtree,body1,basename)
```

Use `showdetails` on the tree to confirm the rigid body and joint were added properly.

```
showdetails(rbtree)
```

```
--------------------
Robot: (1 bodies)

 Idx    Body Name   Joint Name   Joint Type    Parent Name(Idx)   Children Name(s)
 ---    ---------   ----------   ----------    ----------------   ----------------
   1           b1         jnt1     revolute             base(0)   
--------------------
```

Build Manipulator Robot Using Denavit-Hartenberg Parameters

Use the Denavit-Hartenberg (DH) parameters of the Puma560® robot to build a robot. Each rigid body is added one at a time, with the child-to-parent transform specified by the joint object.

The DH parameters define the geometry of the robot with relation to how each rigid body is attached to its parent. For convenience, setup the parameters for the Puma560 robot in a matrix[1] on page 1-

331. The Puma robot is a serial chain manipulator. The DH parameters are relative to the previous row in the matrix, corresponding to the previous joint attachment.

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

Create a rigid body tree object to build the robot.

```
robot = rigidBodyTree;
```

Create the first rigid body and add it to the robot. To add a rigid body:

1. Create a `rigidBody` object and give it a unique name.
2. Create a `rigidBodyJoint` object and give it a unique name.
3. Use `setFixedTransform` to specify the body-to-body transformation using DH parameters. The last element of the DH parameters, `theta`, is ignored because the angle is dependent on the joint position.
4. Call `addBody` to attach the first body joint to the base frame of the robot.

```
body1 = rigidBody('body1');
jnt1 = rigidBodyJoint('jnt1','revolute');

setFixedTransform(jnt1,dhparams(1,:),'dh');
body1.Joint = jnt1;

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

Create and add other rigid bodies to the robot. Specify the previous body name when calling `addBody` to attach it. Each fixed transform is relative to the previous joint coordinate frame.

```
body2 = rigidBody('body2');
jnt2 = rigidBodyJoint('jnt2','revolute');
body3 = rigidBody('body3');
jnt3 = rigidBodyJoint('jnt3','revolute');
body4 = rigidBody('body4');
jnt4 = rigidBodyJoint('jnt4','revolute');
body5 = rigidBody('body5');
jnt5 = rigidBodyJoint('jnt5','revolute');
body6 = rigidBody('body6');
jnt6 = rigidBodyJoint('jnt6','revolute');

setFixedTransform(jnt2,dhparams(2,:),'dh');
setFixedTransform(jnt3,dhparams(3,:),'dh');
setFixedTransform(jnt4,dhparams(4,:),'dh');
setFixedTransform(jnt5,dhparams(5,:),'dh');
setFixedTransform(jnt6,dhparams(6,:),'dh');

body2.Joint = jnt2;
body3.Joint = jnt3;
body4.Joint = jnt4;
body5.Joint = jnt5;
body6.Joint = jnt6;
```

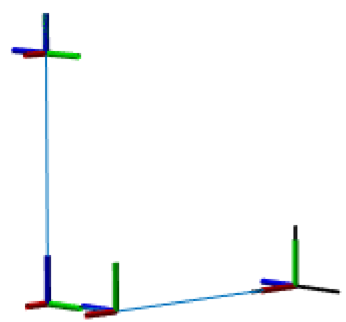
```
addBody(robot,body2,'body1')
addBody(robot,body3,'body2')
addBody(robot,body4,'body3')
addBody(robot,body5,'body4')
addBody(robot,body6,'body5')
```

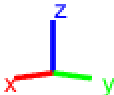
Verify that your robot was built properly by using the `showdetails` or `show` function. `showdetails` lists all the bodies in the MATLAB® command window. `show` displays the robot with a given configuration (home by default). Calls to `axis` modify the axis limits and hide the axis labels.

```
showdetails(robot)
```

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

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

```
show(robot);
axis([-0.5,0.5,-0.5,0.5,-0.5,0.5])
axis off
```





References

[1] Corke, P. I., and B. Armstrong-Helouvry. “A Search for Consensus among Model Parameters Reported for the PUMA 560 Robot.” *Proceedings of the 1994 IEEE International Conference on Robotics and Automation*, IEEE Computer. Soc. Press, 1994, pp. 1608–13. *DOI.org (Crossref)*, doi:10.1109/ROBOT.1994.351360.

Modify a Robot Rigid Body Tree Model

Make changes to an existing `rigidBodyTree` object. You can get replace joints, bodies and subtrees in the rigid body tree.

Load example robots as `rigidBodyTree` objects.

```
load exampleRobots.mat
```

View the details of the Puma robot using `showdetails`.

```
showdetails(puma1)
```

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

```
 Idx    Body Name   Joint Name   Joint Type    Parent Name(Idx)   Children Name(s)
 ---    ---------   ----------   ----------    ----------------   ----------------
   1           L1         jnt1     revolute             base(0)   L2(2)
   2           L2         jnt2     revolute               L1(1)   L3(3)
   3           L3         jnt3     revolute               L2(2)   L4(4)
   4           L4         jnt4     revolute               L3(3)   L5(5)
   5           L5         jnt5     revolute               L4(4)   L6(6)
   6           L6         jnt6     revolute               L5(5)
--------------------
```

Get a specific body to inspect the properties. The only child of the `L3` body is the `L4` body. You can copy a specific body as well.

```
body3 = getBody(puma1,'L3');
childBody = body3.Children{1}
```

```
childBody =
  rigidBody with properties:

            Name: 'L4'
           Joint: [1x1 rigidBodyJoint]
            Mass: 1
    CenterOfMass: [0 0 0]
         Inertia: [1 1 1 0 0 0]
          Parent: [1x1 rigidBody]
        Children: {[1x1 rigidBody]}
         Visuals: {}
      Collisions: {}
```

```
body3Copy = copy(body3);
```

Replace the joint on the `L3` body. You must create a new `Joint` object and use `replaceJoint` to ensure the downstream body geometry is unaffected. Call `setFixedTransform` if necessary to define a transform between the bodies instead of with the default identity matrices.

```
newJoint = rigidBodyJoint('prismatic');
replaceJoint(puma1,'L3',newJoint);

showdetails(puma1)
```

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

 Idx    Body Name       Joint Name       Joint Type    Parent Name(Idx)   Children Name(s)
 ---    ---------       ----------       ----------    ----------------   ----------------
   1           L1             jnt1         revolute             base(0)   L2(2)
   2           L2             jnt2         revolute               L1(1)   L3(3)
   3           L3        prismatic            fixed               L2(2)   L4(4)
   4           L4             jnt4         revolute               L3(3)   L5(5)
   5           L5             jnt5         revolute               L4(4)   L6(6)
   6           L6             jnt6         revolute               L5(5)
--------------------
```

Remove an entire body and get the resulting subtree using `removeBody`. The removed body is included in the subtree.

```
subtree = removeBody(puma1,'L4')
```

```
subtree = 
  rigidBodyTree with properties:

     NumBodies: 3
        Bodies: {[1x1 rigidBody]  [1x1 rigidBody]  [1x1 rigidBody]}
          Base: [1x1 rigidBody]
     BodyNames: {'L4'  'L5'  'L6'}
      BaseName: 'L3'
       Gravity: [0 0 0]
    DataFormat: 'struct'
```

Remove the modified `L3` body. Add the original copied `L3` body to the `L2` body, followed by the returned subtree. The robot model remains the same. See a detailed comparison through `showdetails`.

```
removeBody(puma1,'L3');
addBody(puma1,body3Copy,'L2')
addSubtree(puma1,'L3',subtree)

showdetails(puma1)
```

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

 Idx    Body Name   Joint Name   Joint Type    Parent Name(Idx)   Children Name(s)
 ---    ---------   ----------   ----------    ----------------   ----------------
   1           L1         jnt1     revolute             base(0)   L2(2)
   2           L2         jnt2     revolute               L1(1)   L3(3)
   3           L3         jnt3     revolute               L2(2)   L4(4)
   4           L4         jnt4     revolute               L3(3)   L5(5)
   5           L5         jnt5     revolute               L4(4)   L6(6)
   6           L6         jnt6     revolute               L5(5)
--------------------
```

Specify Dynamics Properties to Rigid Body Tree

To use dynamics functions to calculate joint torques and accelerations, specify the dynamics properties for the `rigidBodyTree` object and `rigidBody`.

Create a rigid body tree model. Create two rigid bodies to attach to it.

```
robot = rigidBodyTree('DataFormat','row');
body1 = rigidBody('body1');
body2 = rigidBody('body2');
```

Specify joints to attach to the bodies. Set the fixed transformation of `body2` to `body1`. This transform is 1m in the *x*-direction.

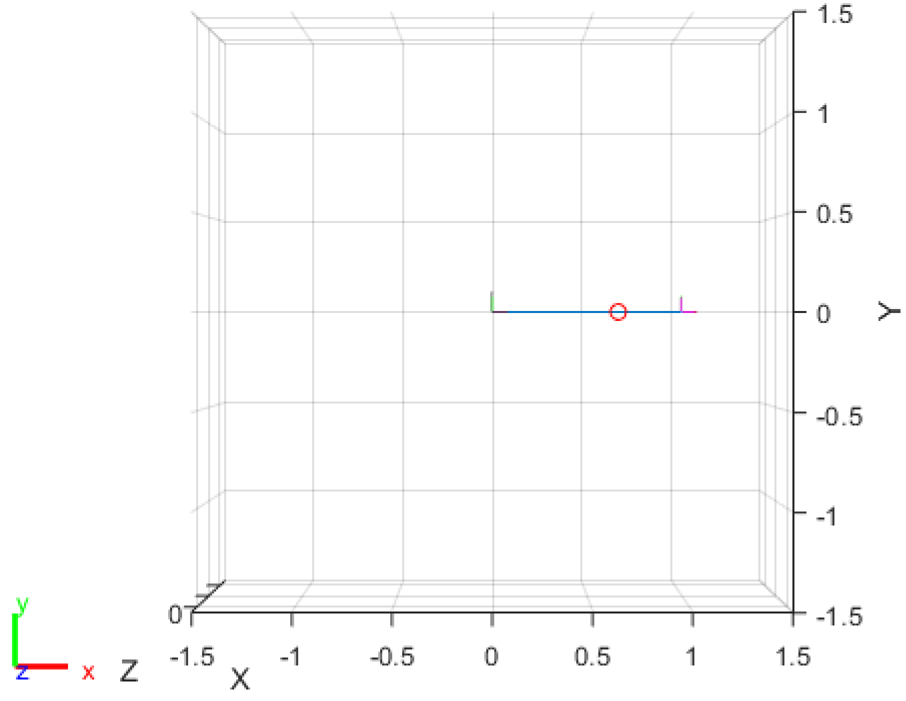
```
joint1 = rigidBodyJoint('joint1','revolute');
joint2 = rigidBodyJoint('joint2');
setFixedTransform(joint2,trvec2tform([1 0 0]))
body1.Joint = joint1;
body2.Joint = joint2;
```

Specify dynamics properties for the two bodies. Add the bodies to the robot model. For this example, basic values for a rod (`body1`) with an attached spherical mass (`body2`) are given.

```
body1.Mass = 2;
body1.CenterOfMass = [0.5 0 0];
body1.Inertia = [0.001 0.67 0.67 0 0 0];

body2.Mass = 1;
body2.CenterOfMass = [0 0 0];
body2.Inertia = 0.0001*[4 4 4 0 0 0];

addBody(robot,body1,'base');
addBody(robot,body2,'body1');
```

Compute the center of mass position of the whole robot. Plot the position on the robot. Move the view to the *xy* plane.

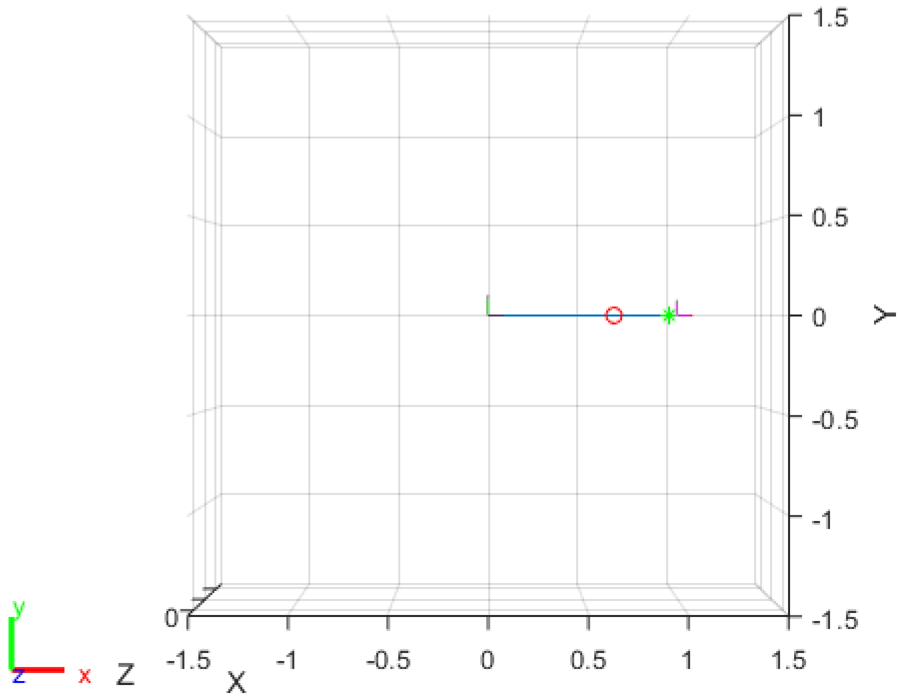
```
comPos = centerOfMass(robot);

show(robot);
hold on
plot(comPos(1),comPos(2),'or')
view(2)
```



Change the mass of the second body. Notice the change in center of mass.

```
body2.Mass = 20;
replaceBody(robot,'body2',body2)
```

```
comPos2 = centerOfMass(robot);
plot(comPos2(1),comPos2(2),'*g')
hold off
```



Compute Forward Dynamics Due to External Forces on Rigid Body Tree Model

Calculate the resultant joint accelerations for a given robot configuration with applied external forces and forces due to gravity. A wrench is applied to a specific body with the gravity being specified for the whole robot.

Load a predefined KUKA LBR robot model, which is specified as a `RigidBodyTree` object.

```
load exampleRobots.mat lbr
```

Set the data format to `'row'`. For all dynamics calculations, the data format must be either `'row'` or `'column'`.

```
lbr.DataFormat = 'row';
```

Set the gravity. By default, gravity is assumed to be zero.

```
lbr.Gravity = [0 0 -9.81];
```

Get the home configuration for the `lbr` robot.

```
q = homeConfiguration(lbr);
```

Specify the wrench vector that represents the external forces experienced by the robot. Use the `externalForce` function to generate the external force matrix. Specify the robot model, the end effector that experiences the wrench, the wrench vector, and the current robot configuration. `wrench` is given relative to the `'tool0'` body frame, which requires you to specify the robot configuration, `q`.

```
wrench = [0 0 0.5 0 0 0.3];
fext = externalForce(lbr,'tool0',wrench,q);
```

Compute the resultant joint accelerations due to gravity, with the external force applied to the end-effector `'tool0'` when `lbr` is at its home configuration. The joint velocities and joint torques are assumed to be zero (input as an empty vector `[]`).

```
qddot = forwardDynamics(lbr,q,[],[],fext);
```

Compute Inverse Dynamics from Static Joint Configuration

Use the `inverseDynamics` function to calculate the required joint torques to statically hold a specific robot configuration. You can also specify the joint velocities, joint accelerations, and external forces using other syntaxes.

Load a predefined KUKA LBR robot model, which is specified as a `RigidBodyTree` object.

```
load exampleRobots.mat lbr
```

Set the data format to `'row'`. For all dynamics calculations, the data format must be either `'row'` or `'column'`.

```
lbr.DataFormat = 'row';
```

Set the `Gravity` property to give a specific gravitational acceleration.

```
lbr.Gravity = [0 0 -9.81];
```

Generate a random configuration for `lbr`.

```
q = randomConfiguration(lbr);
```

Compute the required joint torques for `lbr` to statically hold that configuration.

```
tau = inverseDynamics(lbr,q);
```

Compute Joint Torque to Counter External Forces

Use the `externalForce` function to generate force matrices to apply to a rigid body tree model. The force matrix is an *m*-by-6 vector that has a row for each joint on the robot to apply a six-element wrench. Use the `externalForce` function and specify the end effector to properly assign the wrench to the correct row of the matrix. You can add multiple force matrices together to apply multiple forces to one robot.

To calculate the joint torques that counter these external forces, use the `inverseDynamics` function.

Load a predefined KUKA LBR robot model, which is specified as a `RigidBodyTree` object.

```
load exampleRobots.mat lbr
```

Set the data format to `'row'`. For all dynamics calculations, the data format must be either `'row'` or `'column'`.

```
lbr.DataFormat = 'row';
```

Set the `Gravity` property to give a specific gravitational acceleration.

```
lbr.Gravity = [0 0 -9.81];
```

Get the home configuration for `lbr`.

```
q = homeConfiguration(lbr);
```

Set external force on `link1`. The input wrench vector is expressed in the base frame.

```
fext1 = externalForce(lbr,'link_1',[0 0 0.0 0.1 0 0]);
```

Set external force on the end effector, `tool0`. The input wrench vector is expressed in the `tool0` frame.

```
fext2 = externalForce(lbr,'tool0',[0 0 0.0 0.1 0 0],q);
```

Compute the joint torques required to balance the external forces. To combine the forces, add the force matrices together. Joint velocities and accelerations are assumed to be zero (input as `[]`).

```
tau = inverseDynamics(lbr,q,[],[],fext1+fext2);
```

Display Robot Model with Visual Geometries

You can import robots that have `.stl` files associated with the Unified Robot Description format (URDF) file to describe the visual geometries of the robot. Each rigid body has an individual visual geometry specified. The `importrobot` function parses the URDF file to get the robot model and visual geometries. The function assumes that visual geometry and collision geometry of the robot are the same and assigns the visual geometries as collision geometries of corresponding bodies.

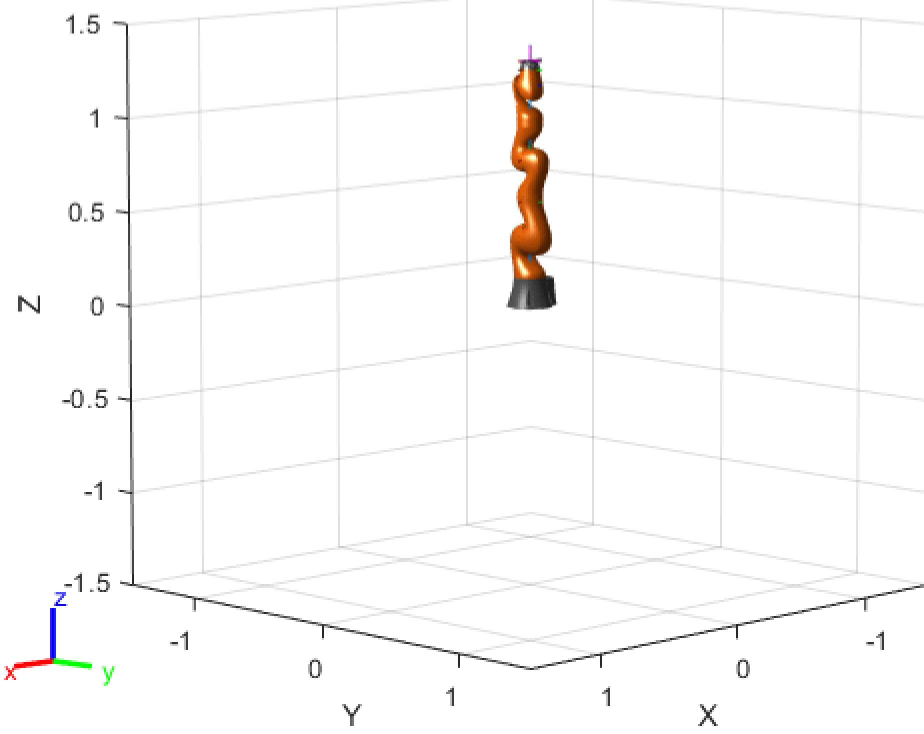
Use the `show` function to display the visual and collision geometries of the robot model in a figure. You can then interact with the model by clicking components to inspect them and right-clicking to toggle visibility.

Import a robot model as a URDF file. The `.stl` file locations must be properly specified in this URDF. To add other `.stl` files to individual rigid bodies, see `addVisual`.

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

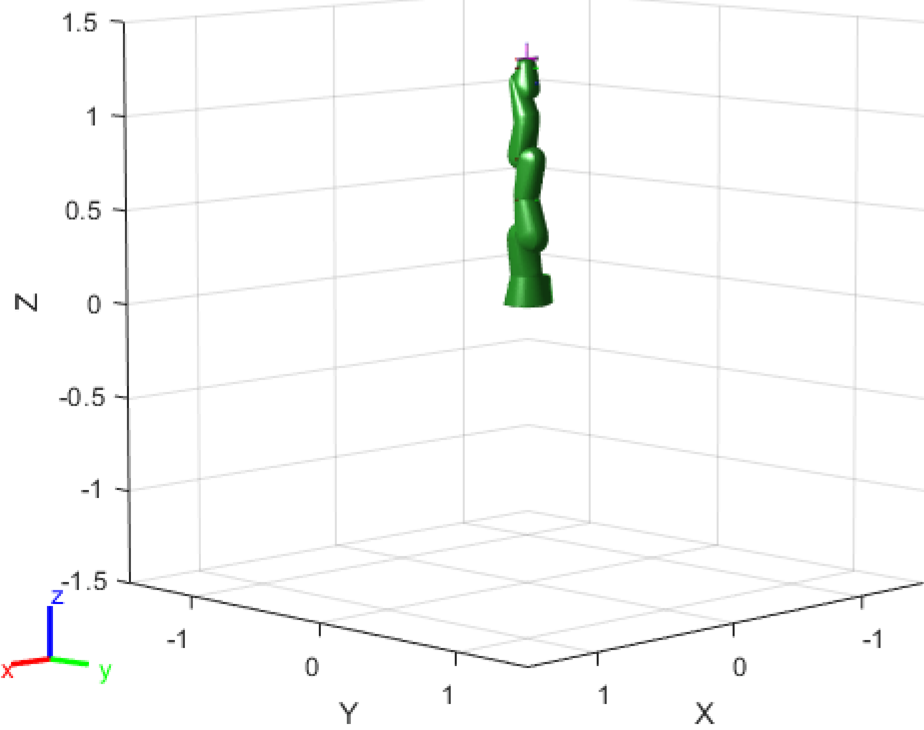
Visualize the robot with the associated visual model. Click bodies or frames to inspect them. Right-click bodies to toggle visibility for each visual geometry.

```
show(robot,'visuals','on','collision','off');
```



Visualize the robot with the associated collision geometries. Click bodies or frames to inspect them. Right-click bodies to toggle visibility for each collision geometry.

```
show(robot,'visuals','off','collision','on');
```



More About

Dynamics Properties

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

- `Mass` — Mass of the rigid body in kilograms.
- `CenterOfMass` — Center of mass position of the rigid body, specified as a vector of the form `[x y z]`. The vector describes the location of the center of mass of the rigid body, relative to the body frame, in meters. The `centerOfMass` object function uses these rigid body property values when computing the center of mass of a robot.
- `Inertia` — Inertia of the rigid body, specified as a vector of the form `[Ixx Iyy Izz Iyz Ixz Ixy]`. The vector is relative to the body frame in kilogram square meters. The inertia tensor is a positive definite matrix of the form:

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

The first three elements of the `Inertia` vector are the moment of inertia, which are the diagonal elements of the inertia tensor. The last three elements are the product of inertia, which are the off-diagonal elements of the inertia tensor.

For information related to the entire manipulator robot model, specify these `rigidBodyTree` object properties:

- `Gravity` — Gravitational acceleration experienced by the robot, specified as an `[x y z]` vector in m/s^2. By default, there is no gravitational acceleration.
- `DataFormat` — The input and output data format for the kinematics and dynamics functions, specified as `"struct"`, `"row"`, or `"column"`.

Dynamics Equations

Manipulator rigid body dynamics are governed by this equation:

$$\frac{d}{dt}\begin{bmatrix} q \\ \dot{q} \end{bmatrix} = \begin{bmatrix} \dot{q} \\ M(q)^{-1}\left(-C(q,\dot{q})\dot{q} - G(q) - J(q)^T F_{Ext} + \tau\right) \end{bmatrix}$$

also written as:

$$M(q)\ddot{q} = -C(q,\dot{q})\dot{q} - G(q) - J(q)^T F_{Ext} + \tau$$

where:

- $M(q)$ — is a joint-space mass matrix based on the current robot configuration. Calculate this matrix by using the `massMatrix` object function.
- $C(q, \dot{q})$ — are the Coriolis terms, which are multiplied by $\dot{q}$ to calculate the velocity product. Calculate the velocity product by using by the `velocityProduct` object function.
- $G(q)$ — is the gravity torques and forces required for all joints to maintain their positions in the specified gravity `Gravity`. Calculate the gravity torque by using the `gravityTorque` object function.
- $J(q)$ — is the geometric Jacobian for the specified joint configuration. Calculate the geometric Jacobian by using the `geometricJacobian` object function.
- F_{Ext} — is a matrix of the external forces applied to the rigid body. Generate external forces by using the `externalForce` object function.
- τ — are the joint torques and forces applied directly as a vector to each joint.
- $q, \dot{q}, \ddot{q}$ — are the joint configuration, joint velocities, and joint accelerations, respectively, as individual vectors. For revolute joints, specify values in radians, rad/s, and rad/s^2, respectively. For prismatic joints, specify in meters, m/s, and m/s^2.

To compute the dynamics directly, use the `forwardDynamics` object function. The function calculates the joint accelerations for the specified combinations of the above inputs.

To achieve a certain set of motions, use the `inverseDynamics` object function. The function calculates the joint torques required to achieve the specified configuration, velocities, accelerations, and external forces.

Version History

Introduced in R2016b

R2019b: `rigidBodyTree` was renamed

Behavior change in future release

The `rigidBodyTree` object was renamed from `robotics.RigidBodyTree`. Use `rigidBodyTree` for all object creation.

References

[1] Craig, John J. *Introduction to Robotics: Mechanics and Control*. Reading, MA: Addison-Wesley, 1989.

[2] Siciliano, Bruno, Lorenzo Sciavicco, Luigi Villani, and Giuseppe Oriolo. *Robotics: Modelling, Planning and Control*. London: Springer, 2009.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

When creating the `rigidBodyTree` object, use the syntax that specifies the `MaxNumBodies` as an upper bound for adding bodies to the robot model. You must also specify the `DataFormat` property as a name-value pair. For example:

```
robot = rigidBodyTree("MaxNumBodies",15,"DataFormat","row")
```

To minimize data usage, limit the upper bound to a number close to the expected number of bodies in the model. All data formats are supported for code generation. To use the dynamics functions, the data format must be set to `"row"` or `"column"`.

The `show` and `showdetails` functions do not support code generation.

See Also

`importrobot` | `inverseKinematics` | `generalizedInverseKinematics` | `rigidBodyJoint` | `rigidBody`

Topics

"Build a Robot Step by Step"
"Solve Inverse Kinematics for Closed Loop Linkages"
"Compute Joint Torques To Balance An Endpoint Force and Moment"
"Rigid Body Tree Robot Model"

rigidBodyTreeImportInfo

Object for storing `rigidBodyTree` import information

Description

The `rigidBodyTreeImportInfo` object is created by the `importrobot` function when converting a Simulink® model using Simscape Multibody components. Get import information for specific bodies, joints, or blocks using the object functions. Changes to the Simulink model are not reflected in this object after initially calling `importrobot`.

Creation

`[robot,importInfo] = importrobot(model)` imports a Simscape Multibody model and returns an equivalent `rigidBodyTree` object, `robot`, and info about the import in `importInfo`. Only fixed, prismatic, and revolute joints are supported in the output `rigidBodyTree` object.

If you are importing a model that uses other joint types, constraint blocks, or variable inertias, use the “Simscape Multibody Model Import” on page 2-0 name-value pairs to disable errors.

Properties

`SourceModelName` — Name of source model from Simscape Multibody
character vector

This property is read-only.

Name of the source model from Simscape Multibody, specified as a character vector. This property matches the name of the input `model` when calling `importrobot`.

Example: `'sm_import_humanoid_urdf'`

Data Types: `char`

`RigidBodyTree` — Robot model
`rigidBodyTree` object

This property is read-only.

Robot model, returned as a `rigidBodyTree` object.

`BlockConversionInfo` — List of blocks that were converted
structure

This property is read-only.

List of blocks that were converted from Simscape Multibody blocks to preserve compatibility, specified as a structure with the nested fields:

- `AddedBlocks`

  - `ImplicitJoints` — Cell array of implicit joints added during the conversion process.
- `ConvertedBlocks`
  - `Joints` — Cell array of joint blocks that were converted to fixed joints.
  - `JointSourceType` — `containers.Map` object that associates converted joint blocks to their original joint type.
- `RemovedBlocks`
  - `ChainClosureJoints`— Cell array of joint blocks removed to open closed chains.
  - `SMConstraints` — Cell array of constraint blocks that were removed.
  - `VariableInertias` — Cell array of variable inertia blocks that were removed.

Object Functions

bodyInfo	Import information for body
bodyInfoFromBlock	Import information for block name
bodyInfoFromJoint	Import information for given joint name
showdetails	Display details of imported robot

Version History

Introduced in R2018b

R2019b: `rigidBodyTreeImportInfo` was renamed

Behavior change in future release

The `rigidBodyTreeImportInfo` object was renamed from `robotics.RigidBodyTreeImportInfo`. Use `rigidBodyTreeImportInfo` for all object creation.

See Also

`importrobot` | `rigidBodyTree`

Topics

"Rigid Body Tree Robot Model"

robotLidarPointCloudGenerator

Generate point cloud from meshes

Description

The `robotLidarPointCloudGenerator` System object generates detections from a statistical simulated lidar sensor. The system object uses a statistical sensor model to simulate lidar detections with added random noise. All detections are with respect to the coordinate frame of the vehicle-mounted sensor. You can use the `robotLidarPointCloudGenerator` object in a scenario, created using a `robotSensor`, containing static meshes, robot platforms, and sensors.

To generate lidar point clouds:

1. Create the `robotLidarPointCloudGenerator` object and set its properties.
2. Call the object with arguments, as if it were a function.

To learn more about how System objects work, see What Are System Objects?

Creation

Syntax

```
lidar = robotLidarPointCloudGenerator
lidar = robotLidarPointCloudGenerator(Name=Value)
```

Description

`lidar = robotLidarPointCloudGenerator` creates a statistical sensor model to generate point cloud for a lidar. This sensor model will have default properties.

`lidar = robotLidarPointCloudGenerator(Name=Value)` sets properties using one or more name-value pair arguments. For example, `robotLidarPointCloudGenerator(UpdateRate=100,HasNoise=false)` creates a lidar point cloud generator that reports detections at an update rate of 100 Hz without noise.

Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see System Design in MATLAB Using System Objects.

`UpdateRate` — Update rate of lidar sensor
`10` (default) | positive real scalar

Update rate of the lidar sensor, specified as a positive real scalar in Hz. This property sets the frequency at which new detections happen.

Example: `UpdateRate=100`

Data Types: `single` | `double`

`MaxRange` — Maximum detection range of lidar sensor
`120` (default) | positive real scalar

Maximum detection range of the lidar sensor, specified as a positive real scalar in meters. The sensor does not detect objects beyond this range.

Example: `MaxRange=100`

Data Types: `single` | `double`

`RangeAccuracy` — Accuracy of range measurements
`0.0020` (default) | positive real scalar

Accuracy of the range measurements, specified as a positive real scalar in meters. This property sets the one-standard-deviation accuracy of the sensor range measurements.

Example: `RangeAccuracy=0.001`

Data Types: `single` | `double`

`AzimuthResolution` — Azimuthal resolution of lidar sensor
`0.1600` (default) | positive real scalar

Azimuthal resolution of the lidar sensor, specified as a positive real scalar in degrees. The azimuthal resolution defines the minimum separation in azimuth angle at which the lidar sensor can distinguish two targets.

Example: `AzimuthResolution=0.6000`

Data Types: `single` | `double`

`ElevationResolution` — Elevation resolution of lidar sensor
`1.2500` (default) | positive real scalar

Elevation resolution of the lidar sensor, specified as a positive real scalar in degrees. The elevation resolution defines the minimum separation in elevation angle at which the lidar can distinguish two targets.

Example: `ElevationResolution=0.6000`

Data Types: `single` | `double`

`AzimuthLimits` — Azimuthal limits of lidar sensor
`[-180 180]` (default) | two-element vector

Azimuth limits of the lidar sensor, specified as a two-element vector of the form [*min max*]. Units are in degrees.

Example: `AzimuthLimits=[-60 100]`

Data Types: `single` | `double`

`ElevationLimits` — Elevation limits of lidar sensor
`[-20 20]` (default) | two-element vector

Elevation limits of the lidar sensor, specified as a two-element vector of the form [*min max*]. Units are in degrees.

Example: `ElevationLimits=[-60 100]`

Data Types: `single` | `double`

`HasNoise` — Add noise to lidar sensor measurements
`true` or `1` (default) | `false` or `0`

Add noise to lidar sensor measurements, specified as `true` or `false`. Set this property to `true` to add noise to the sensor measurements. Otherwise, the measurements have no noise. The sensor adds random Gaussian noise to each point with mean equal to zero and standard deviation specified by the RangeAccuracy property.

Example: `HasNoise=false`

Data Types: `logical`

`HasOrganizedOutput` — Output generated data as organized point cloud locations
`true` or `1` (default) | `false` or `0`

Output generated data as organized point cloud locations, specified as `true` or `false`.

When this property is set as `true`, the Location property of the `pointCloud` object is an *M*-by-*N*-by-3 matrix of organized point cloud. *M* is the number of elevation channels, and *N* is the number of azimuth channels.

When this property is set as `false`, the Location property of the `pointCloud` object is an *M*-by-3 matrix of unorganized list of points. *M* is the product of the numbers of elevation and azimuth channels.

Example: `HasOrganizedOutput=false`

Data Types: `logical`

Usage

Syntax

```
ptCloud = lidar(tgts,simTime)
[ptCloud,isValidTime] = lidar(tgts,simTime)
```

Description

`ptCloud = lidar(tgts,simTime)` generates a lidar point cloud object `ptCloud` from the specified target object, `tgts`, at the specified simulation time `simTime`.

`[ptCloud,isValidTime] = lidar(tgts,simTime)` additionally returns `isValidTime` which specifies if the specified `simTime` is a multiple of the sensor's update interval `(1/UpdateRate)`.

Input Arguments

`tgts` — Target object data
structure | structure array

Target object data, specified as a structure or structure array. Each structure corresponds to a mesh. The table shows the properties that the object uses to generate detections.

Target Object Data

Field	Description
`Mesh`	An `extendedObjectMesh` object representing the geometry of the target object in its own coordinate frame.
`Position`	A three-element vector defining the coordinate position of the target with respect to the sensor frame.
`Orientation`	A `quaternion` object or a 3-by-3 matrix, containing Euler angles, defining the orientation of the target with respect to the sensor frame.

Example: `struct("Mesh",scale(extendedObjectMesh('cuboid'),[100 100 2]),"Position",[0 0 -10],"Orientation",quaternion([1 0 0 0]))`

Data Types: `struct`

`simTime` — Current simulation time
positive real scalar

Current simulation time, specified as a positive real scalar in seconds. The `lidar` object calls the lidar point cloud generator at regular intervals to generate new point clouds at a frequency defined by the `updateRate` property. The value of the `UpdateRate` property must be an integer multiple of the simulation time interval. Updates requested from the sensor between update intervals do not generate a point cloud.

Example: `0`

Data Types: `single` | `double`

Output Arguments

`ptCloud` — Point cloud data
`pointCloud` object

Point cloud data, returned as a `pointCloud` object.

`isValidTime` — Valid time to generate point cloud
`false` or `0` | `true` or `1`

Valid time to generate point cloud, returned as logical `0` (`false`) or `1` (`true`). `isValidTime` is `0` when the requested update time is not a multiple of the `updateRate` property value.

Data Types: `logical`

Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

Common to All System Objects

step	Run System object algorithm
release	Release resources and allow changes to System object property values and input characteristics
reset	Reset internal states of System object

Examples

Generate Point Clouds from Mesh

This example shows how to use a statistical lidar sensor model to generate point clouds from a mesh.

Create Sensor Model

Create a statistical sensor model, `lidar`, using the `robotLidarPointCloudGenerator` System object.

```
lidar = robotLidarPointCloudGenerator(HasOrganizedOutput=false);
```

Create Floor

Use the `extendedObjectMesh` object to create mesh for the target object. Define the position and orientation of the target object with respect to the sensor frame.

```
target = struct("Mesh",scale(extendedObjectMesh("cuboid"),[100 100 2]), ...
                "Position",[0 0 -10], ...
                "Orientation",quaternion([1 0 0 0]));
```

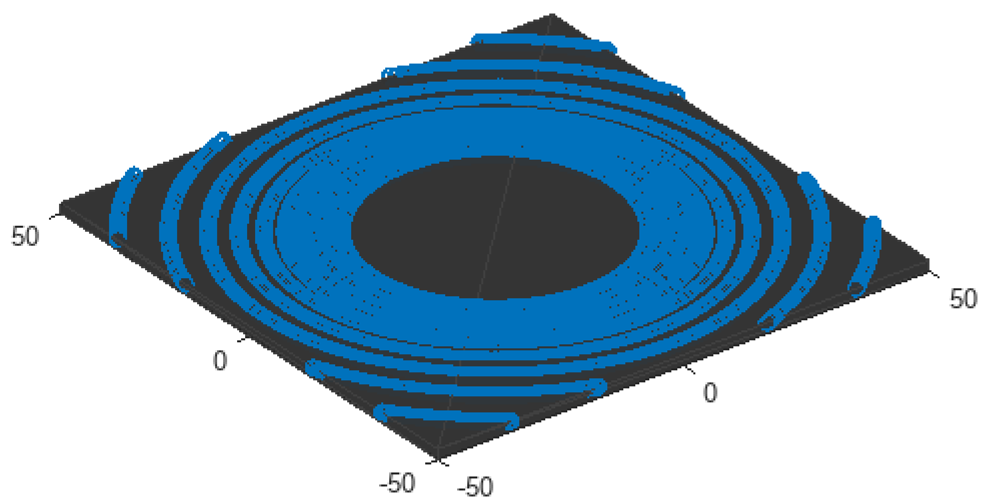
Generate Point Clouds from Floor

```
ptCloud = lidar(target,0);
```

Visualize

Use the `translate` function to translate the object mesh to its specified location and use the `show` function to visualize it. Use the `scatter3` function to plot the point clouds stored in `ptCloud`.

```
show(translate(target.Mesh,target.Position));
hold on
scatter3(ptCloud.Location(:,1),ptCloud.Location(:,2),ptCloud.Location(:,3))
```



Version History

Introduced in R2022a

See Also

`robotPlatform` | `robotScenario` | `robotSensor` | `extendedObjectMesh` | `pointCloud`

robotPlatform

Create robot platform in scenario

Description

The `robotPlatform` object represents a robot platform in a given robot scenario. Use the platform to define and track the trajectory of an object in the scenario. To simulate sensor readings for the platform, mount sensors such as the `gpsSensor`, `insSensor`, and `robotLidarPointCloudGenerator` System object to the platform as `robotSensor` objects. Add a body mesh to the platform for visualization using the `updateMesh` object function.

Creation

Syntax

```
platform = robotPlatform(name,scenario)
platform = robotPlatform( ___ ,Name=Value)
```

Description

`platform = robotPlatform(name,scenario)` creates a platform with a specified name `name` and adds it to the scenario, specified as a `robotScenario` object. Specify the `name` argument as a string scalar. The `name` argument sets the Name property.

`platform = robotPlatform( ___ ,Name=Value)` specifies options using one or more name-value arguments. You can specify properties as name-value arguments as well.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Example: `StartTime=10` sets the initial time of the platform trajectory to 10 seconds.

`BaseTrajectory` — Trajectory for robot platform base motion
`[]` (default) | `waypointTrajectory` object | `polynomialTrajectory` object

Trajectory for robot platform base motion, specified as a `waypointTrajectory` or `polynomialTrajectory` object. By default, the platform is assumed to be stationary and at the scenario origin. To move the platform at each simulation step of the scenario, use the `move` object function.

Note The `robotPlatform` object must specify the same `ReferenceFrame` property as specified in the trajectory object.

`IsBinaryOccupied` — Occupied state of binary occupancy map
`false` (default) | `true`

Occupied state of binary occupancy map, specified as `true` or `false`. Set the value as `true` if robot platform is incorporated in the binary occupancy map.

Data Types: `logical`

`InitialBasePosition` — Initial robot platform base position
`[0 0 0]` (default) | vector of form `[x y z]`

Initial robot platform base position, specified as a vector of the form `[x y z]`. Only specify this name-value argument if not specifying the BaseTrajectory property.

Data Types: `single` | `double`

`InitialBaseVelocity` — Initial velocity of robot platform base
`[0 0 0]` (default) | vector of form `[vx vy vz]`

Initial velocity of robot platform base, specified as a vector of the form `[vx vy vz]`. Only specify this name-value argument if not specifying the BaseTrajectory property.

Data Types: `single` | `double`

`InitialBaseAcceleration` — Initial acceleration of robot platform base
`[0 0 0]` (default) | vector of form `[ax ay az]`

Initial acceleration of robot platform base, specified as a vector of the form `[ax ay az]`. Only specify this name-value argument if not specifying the BaseTrajectory property.

Data Types: `single` | `double`

`InitialBaseOrientation` — Initial robot platform base orientation
`[1 0 0 0]` (default) | vector of form `[w x y z]`

Initial robot platform base orientation, specified as a vector of the form `[w x y z]`, representing a quaternion. Only specify this name-value argument if not specifying the BaseTrajectory property.

Data Types: `single` | `double`

`InitialBaseAngularVelocity` — Initial angular velocity of robot platform base
`[0 0 0]` (default) | vector of form `[wx wy wz]`

Initial angular velocity of robot platform base, specified as a vector of the form `[wx wy wz]`. The magnitude of the vector defines the angular speed in radians per second. The *xyz*-coordinates define the axis of clockwise rotation. Only specify this name-value argument if not specifying the BaseTrajectory property.

Data Types: `single` | `double`

`InitialJointConfiguration` — Initial joint configuration of `rigidBodyTree`-based robot platform
`homeConfiguration` (default) | *N*-element vector

Initial joint configuration of the `rigidBodyTree`-based robot platform, specified as an *N*-element vector. *N* is the total number of joints associated with the `rigidBodyTree` object.

Data Types: `single` | `double`

`ReferenceFrame` — Reference frame for computing robot platform motion
`"ENU"` (default) | `"NED"`

Reference frame for computing robot platform motion, specified as `"ENU"` or `"NED"`, which matches any reference frame in the `robotScenario`. All platform motion is computed relative to this inertial frame.

Data Types: `string` | `char`

`RigidBodyTree` — Rigid body tree robot platform
`[]` (default) | `rigidBodyTree` object

Rigid body tree robot platform, specified as a `rigidBodyTree` object.

`StartTime` — Initial time of robot platform trajectory
`0` (default) | scalar in seconds

Initial time of the robot platform trajectory, specified as a scalar in seconds.

Data Types: `single` | `double`

`Collision` — Add collision object to platform mesh
`"default"` (default) | `false` | `"mesh"` | `"capsule"` | `collisionBox` object | `collisionCapsule` object | `collisionCylinder` object | `collisionMesh` object | `collisionSphere` object

Add collision object to the platform mesh, specified as one of these values:

- `false` — Keeps the platform mesh collision object free. Clear existing collision for `rigidBodyTree`-based platform.
- `"default"` — Keeps existing collision mesh for `rigidBodyTree`-based platform. For other platforms, keeps the platform mesh collision free.
- `"mesh"` — Fits collision object such as `collisionBox`, `collisionCylinder`, `collisionMesh`, and `collisionSphere` based on the platform mesh type.
- `"capsule"` — Fits collision capsule object with the platform mesh.
- One of these externally created collision objects:
  - `collisionBox`
  - `collisionCapsule`
  - `collisionCylinder`
  - `collisionMesh`
  - `collisionSphere`

The `rigidBodyTree`-based platform accepts externally created collision mesh only for base body.

`CollisionOffset` — Transformation of collision mesh relative to platform mesh
`eye(4)` (default) | 4-by-4 homogeneous transformation matrix

Transformation of collision mesh relative to the platform mesh, specified as a 4-by-4 homogeneous transformation matrix. Use the `CollisionOffset` input for `rigidBodyTree`-based platforms only when the `Collision` input is specified as an externally created collision object.

Data Types: `single` | `double`

Properties

Name — Identifier for robot platform
string scalar | character vector

Identifier for the robot platform, specified as a string scalar or character vector. The name must be unique within the scenario.

Data Types: char | string

BaseTrajectory — Trajectory for robot platform base motion
[] (default) | waypointTrajectory object | polynomialTrajectory object

Trajectory for the robot platform base motion, specified as a waypointTrajectory or polynomialTrajectory object. By default, the object assumes the base of the platform is stationary and at the scenario origin. When specified as a waypointTrajectory or polynomialTrajectory object, base of the platform is moved along the trajectory during the scenario simulation. To move the platform at each simulation step of the scenario, use the move object function.

Note The robotPlatform object must specify the same ReferenceFrame property as specified in the trajectory object.

ReferenceFrame — Reference frame for computing robot platform motion
"ENU" (default) | "NED"

Reference frame for computing robot platform motion, specified as "ENU" or "NED", which matches any reference frame in the robotScenario. The object computes all platform motion relative to this inertial frame.

Data Types: char | string

RigidBodyTree — Rigid body tree robot platform
[] (default) | rigidBodyTree object

Rigid body tree robot platform, specified as a rigidBodyTree object.

BaseMesh — Robot platform base body mesh
[1 0.5 0.3] (default) | extendedObjectMesh object

Robot platform base body mesh, specified as an extendedObjectMesh object. The body mesh describes the 3-D model of the platform for visualization purposes. The body mesh is used to generate 3-D point cloud. The default mesh is a cuboid of the form [*xlength ylength zlength*] in meters.

BaseMeshColor — Robot platform base body mesh color
[1 0 0] (default) | RGB triplet

Robot platform base body mesh color when displayed in the scenario, specified as an RGB triplet.

Data Types: single | double

BaseMeshTransform — Transform between robot platform base body and mesh frames
eye(4) (default) | 4-by-4 homogeneous transformation matrix

Transform between robot platform base body and mesh frames, specified as a 4-by-4 homogeneous transformation matrix that maps points in the platform mesh frame to points in the body frame.

Data Types: `single` | `double`

`IsBinaryOccupied` — Status of robot platform in binary occupancy map
`false` (default) | `true`

Status of robot platform in binary occupancy map, specified as `true` or `false`.

Data Types: `logical`

`CollisionMesh` — Robot platform base collision mesh
`[]` (default) | `collisionBox` object | `collisionCapsule` object | `collisionCylinder` object | `collisionMesh` object | `collisionSphere` object

Collision mesh associated with the robot platform base body mesh, specified as a collision object. The supported collision object types are: `collisionBox`, `collisionCapsule`, `collisionCylinder`, `collisionMesh`, and `collisionSphere`.

`CollisionMeshOffset` — Transform between robot platform base body and collision mesh
`eye(4)` (default) | 4-by-4 homogeneous transformation matrix

Transform between the robot platform base body and collision mesh, specified as a 4-by-4 homogeneous transformation matrix.

Data Types: `single` | `double`

`Sensors` — Sensors mounted on robot platform
`[]` (default) | array of `robotSensor` objects

Sensors mount on robot platform, specified as an array of `robotSensor` objects.

Object Functions

attach	Attach target robot platform to source robot platform
checkCollision	Check collision between robot platform and target bodies
detach	Detach target robot platform from source robot platform
move	Move robot platform in scenario
read	Read robot platform in scenario
updateMesh	Update robot platform body mesh

Examples

Create and Simulate Robot Scenario

Create a robot scenario.

```
scenario = robotScenario(UpdateRate=100,StopTime=1);
```

Add the ground plane and a box as meshes.

```
addMesh(scenario,"Plane",Size=[3 3],Color=[0.7 0.7 0.7]);
addMesh(scenario,"Box",Size=[0.5 0.5 0.5],Position=[0 0 0.25], ...
        Color=[0 1 0])
```

Create a waypoint trajectory for the robot platform using an ENU reference frame.

```
waypoint = [0 -1 0; 1 0 0; -1 1 0; 0 -1 0];
toa = linspace(0,1,length(waypoint));
traj = waypointTrajectory("Waypoints",waypoint, ...
                          "TimeOfArrival",toa, ...
                          "ReferenceFrame","ENU");
```

Create a `rigidBodyTree` object of the TurtleBot 3 Waffle Pi robot with `loadrobot`.

```
robotRBT = loadrobot("robotisTurtleBot3WafflePi");
```

Create a robot platform with trajectory.

```
platform = robotPlatform("TurtleBot",scenario, ...
                         BaseTrajectory=traj);
```

Set up platform mesh with the `rigidBodyTree` object.

```
updateMesh(platform,"RigidBodyTree",Object=robotRBT)
```

Create an INS sensor object and attach the sensor to the platform.

```
ins = robotSensor("INS",platform,insSensor("RollAccuracy",0), ...
                  UpdateRate=scenario.UpdateRate);
```

Visualize the scenario.

```
[ax,plotFrames] = show3D(scenario);
axis equal
hold on
```

In a loop, step through the trajectory to output the position, orientation, velocity, acceleration, and angular velocity.

```
count = 1;
while ~isDone(traj)
    [Position(count,:),Orientation(count,:),Velocity(count,:), ...
     Acceleration(count,:),AngularVelocity(count,:)] = traj();
    count = count+1;
end
```

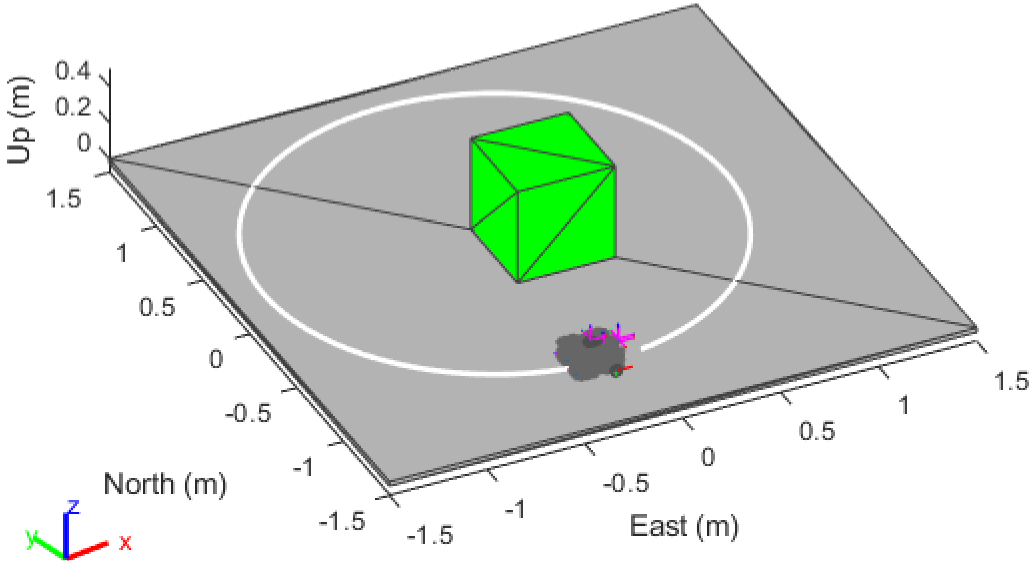
Create a line plot for the trajectory. First create the plot with `plot3`, then manually modify the data source properties of the plot. This improves the performance of the plotting.

```
trajPlot = plot3(nan,nan,nan,"Color",[1 1 1],"LineWidth",2);
trajPlot.XDataSource = "Position(:,1)";
trajPlot.YDataSource = "Position(:,2)";
trajPlot.ZDataSource = "Position(:,3)";
```

Set up the simulation. Then, iterate through the positions and show the scene each time the INS sensor updates. Advance the scene, move the robot platform, and update the sensors.

```
setup(scenario)
for idx = 1:count-1
    % Read sensor readings.
    [isUpdated,insTimestamp(idx,1),sensorReadings(idx)] = read(ins);
    if isUpdated
        % Use fast update to move platform visualization frames.
```

```
        show3D(scenario,FastUpdate=true,Parent=ax);
        % Refresh all plot data and visualize.
        refreshdata
        drawnow limitrate
    end
    % Advance scenario simulation time.
    advance(scenario);
    % Update all sensors in the scene.
    updateSensors(scenario)
end
hold off
```



Perform Pick and Place in Robot Scenario

Create a `robotScenario` object.

```
scenario = robotScenario(UpdateRate=1,StopTime=10);
```

Create a `rigidBodyTree` object of the Franka Emika Panda manipulator using `loadrobot`.

```
robotRBT = loadrobot("frankaEmikaPanda");
```

Create a `rigidBodyTree`-based `robotPlatform` object using the manipulator model.

```
robot = robotPlatform("Manipulator",scenario, ...
                      RigidBodyTree=robotRBT);
```

Create a non-`rigidBodyTree`-based `robotPlatform` object of a box to manipulate. Specify the mesh type and size.

```
box = robotPlatform("Box",scenario,Collision="mesh", ...
                    InitialBasePosition=[0.5 0.15 0.278]);
updateMesh(box,"Cuboid",Collision="mesh",Size=[0.06 0.06 0.1])
```

Visualize the scenario.

```
ax = show3D(scenario,Collisions="on");
view(79,36)
light
```

Specify the initial and the pick-up joint configuration of the manipulator, to move the manipulator from its initial pose to close to the box.

```
initialConfig = homeConfiguration(robot.RigidBodyTree);
pickUpConfig = [0.2371 -0.0200 0.0542 -2.2272 0.0013 ...
                2.2072 -0.9670 0.0400 0.0400];
```

Create an RRT path planner using the `manipulatorRRT` object, and specify the manipulator model.

```
planner = manipulatorRRT(robot.RigidBodyTree,scenario.CollisionMeshes);
planner.IgnoreSelfCollision = true;
```

Plan the path between the initial and the pick-up joint configurations. Then, to visualize the entire path, interpolate the path into small steps.

```
rng("default")
path = plan(planner,initialConfig,pickUpConfig);
path = interpolate(planner,path,25);
```

Set up the simulation.

```
setup(scenario)
```

Check the collision before manipulator picks up the box.

```
checkCollision(robot,"Box", ...
               IgnoreSelfCollision="on")
```

```
ans = logical
   0
```

Move the joints of the manipulator along the path and visualize the scenario.

```
helperRobotMove(path,robot,scenario,ax)
```

Check the collision after manipulator picks up the box.

```
checkCollision(robot,"Box", ...
               IgnoreSelfCollision="on")
```

```
ans = logical
   1
```

Use the `attach` function to attach the box to the gripper of the manipulator.

```
attach(robot,"Box","panda_hand", ...
       ChildToParentTransform=trvec2tform([0 0 0.1]))
```

Specify the drop-off joint configuration of the manipulator to move the manipulator from its pick-up pose to the box drop-off pose.

```
dropOffConfig = [-0.6564 0.2885 -0.3187 -1.5941 0.1103 ...
                 1.8678 -0.2344 0.04 0.04];
```

Plan the path between the pick-up and drop-off joint configurations.

```
path = plan(planner,pickUpConfig,dropOffConfig);
path = interpolate(planner,path,25);
```

Move the joints of the manipulator along the path and visualize the scenario.

```
helperRobotMove(path,robot,scenario,ax)
```

Use the `detach` function to detach the box from the manipulator gripper.

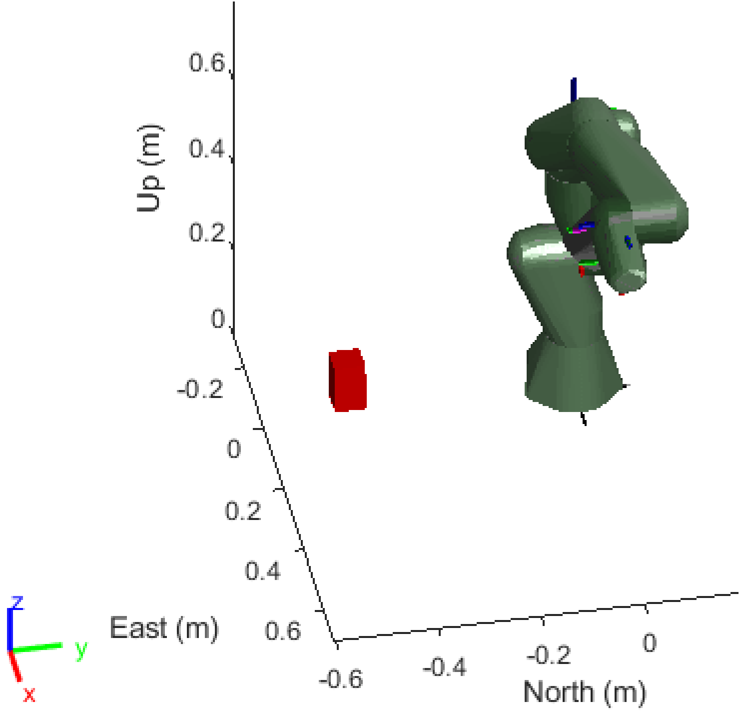
```
detach(robot)
```

Plan the path between the drop-off and initial joint configurations to move the manipulator from its box drop-off pose to its initial pose.

```
path = plan(planner,dropOffConfig,initialConfig);
path = interpolate(planner,path,25);
```

Move the joints of the manipulator along the path and visualize the scenario.

```
helperRobotMove(path,robot,scenario,ax)
```



Helper function to move the joints of the manipulator.

```
function helperRobotMove(path,robot,scenario,ax)
    for idx = 1:size(path,1)
        jointConfig = path(idx,:);
        move(robot,"joint",jointConfig)
        show3D(scenario,fastUpdate=true,Parent=ax,Collisions="on");
        drawnow
        advance(scenario);
    end
end
```

Version History

Introduced in R2022a

See Also

Objects
robotScenario | robotSensor | extendedObjectMesh | waypointTrajectory

Functions
attach | checkCollision | detach | move | read | updateMesh

robotScenario

Generate robot simulation scenario

Description

The `robotScenario` object generates a simulation scenario consisting of static meshes, robot platforms, and sensors in a 3-D environment.

Creation

Syntax

```
scenario = robotScenario
scenario = robotScenario(Name=Value)
```

Description

`scenario = robotScenario` creates an empty robot scenario with default property values. The default inertial frames are the east-north-up (ENU) and the north-east-down (NED) frames.

`scenario = robotScenario(Name=Value)` configures a `robotScenario` object with properties using one or more name-value arguments.

Properties

`UpdateRate` — Simulation update rate
`10` (default) | positive scalar

Simulation update rate, specified as a positive scalar in Hz. The step size of the scenario when using an `advance` object function is equal to the inverse of the update rate.

Example: `2`

Data Types: `single` | `double`

`StopTime` — Stop time of simulation
`inf` (default) | nonnegative scalar

Stop time of the simulation, specified as a nonnegative scalar in seconds. A scenario stops advancing when it reaches the stop time.

Example: `60`

Data Types: `single` | `double`

`HistoryBufferSize` — Maximum number of steps stored in scenario
`100` (default) | positive integer greater than `1`

Maximum number of steps stored in scenario, specified as a positive integer greater than `1`. This property determines the maximum number of frames of platform poses stored in the scenario. If the

number of simulated steps exceeds the value of this property, then the scenario stores only latest steps.

Example: `60`

Data Types: `single` | `double`

`ReferenceLocation` — Scenario origin in geodetic coordinates
`[0 0 0]` (default) | vector of form `[`*`latitude longitude altitude`*`]`

Scenario origin in geodetic coordinates, specified as a three-element vector of the form `[`*`latitude longitude altitude`*`]`. *latitude* and *longitude* are geodetic coordinates in degrees. *altitude* is the height above the WGS84 reference ellipsoid in meters.

Example: `[46.017 7.750 1673]`

Data Types: `single` | `double`

`MaxNumFrames` — Maximum number of frames in scenario
`50` (default) | positive integer

Maximum number of frames in the scenario, specified as a positive integer. The combined number of inertial frames, platforms, and sensors added to the scenario must be less than or equal to the value of this property.

Example: `15`

Data Types: `single` | `double`

`CurrentTime` — Current simulation time
nonnegative scalar

This property is read-only.

Current simulation time, specified as a nonnegative scalar.

Data Types: `single` | `double`

`IsRunning` — Indicate whether scenario is running
`true` | `false`

This property is read-only.

Indicate whether the scenario is running, specified as `true` or `false`. After a scenario simulation starts, it runs until it reaches the stop time.

Data Types: `logical`

`TransformTree` — Transformation information between frames
`transformTree` object

This property is read-only.

Transformation information between all the frames in the scenario, specified as a `transformTree` object. This property contains the transformation information between the inertial, platform, and sensor frames associated with the scenario.

`InertialFrames` — Names of inertial frames in scenario
vector of strings

This property is read-only.

Names of the inertial frames in the scenario, specified as a vector of strings.

Data Types: `string`

`Meshes` — Static meshes in scenario
1-by-*N* cell array of `extendedObjectMesh` objects

This property is read-only.

Static meshes in the scenario, specified as a 1-by-*N* cell array of `extendedObjectMesh` objects.

`CollisionMeshes` — Collision meshes in scenario
1-by-*N* cell array of collision objects

This property is read-only.

Collision meshes in the scenario, specified as a 1-by-*N* cell array of collision objects. The supported collision object types are: `collisionBox`, `collisionCapsule`, `collisionCylinder`, `collisionMesh`, and `collisionSphere`.

`Platforms` — Robot platforms in scenario
array of `robotPlatform` objects

This property is read-only.

Robot platforms in the scenario, specified as an array of `robotPlatform` objects.

Object Functions

addInertialFrame	Define new inertial frame in robot scenario
addMesh	Add new static mesh to robot scenario
advance	Advance robot scenario simulation by one time step
binaryOccupancyMap	Create 2-D binary occupancy map from robot scenario
restart	Reset simulation of robot scenario
setup	Prepare robot scenario for simulation
show3D	Visualize robot scenario in 3-D
updateSensors	Update sensor readings in robot scenario

Examples

Create and Simulate Robot Scenario

Create a robot scenario.

```
scenario = robotScenario(UpdateRate=100,StopTime=1);
```

Add the ground plane and a box as meshes.

```
addMesh(scenario,"Plane",Size=[3 3],Color=[0.7 0.7 0.7]);
addMesh(scenario,"Box",Size=[0.5 0.5 0.5],Position=[0 0 0.25], ...
        Color=[0 1 0])
```

Create a waypoint trajectory for the robot platform using an ENU reference frame.

```
waypoint = [0 -1 0; 1 0 0; -1 1 0; 0 -1 0];
toa = linspace(0,1,length(waypoint));
traj = waypointTrajectory("Waypoints",waypoint, ...
                          "TimeOfArrival",toa, ...
                          "ReferenceFrame","ENU");
```

Create a `rigidBodyTree` object of the TurtleBot 3 Waffle Pi robot with `loadrobot`.

```
robotRBT = loadrobot("robotisTurtleBot3WafflePi");
```

Create a robot platform with trajectory.

```
platform = robotPlatform("TurtleBot",scenario, ...
                         BaseTrajectory=traj);
```

Set up platform mesh with the `rigidBodyTree` object.

```
updateMesh(platform,"RigidBodyTree",Object=robotRBT)
```

Create an INS sensor object and attach the sensor to the platform.

```
ins = robotSensor("INS",platform,insSensor("RollAccuracy",0), ...
                  UpdateRate=scenario.UpdateRate);
```

Visualize the scenario.

```
[ax,plotFrames] = show3D(scenario);
axis equal
hold on
```

In a loop, step through the trajectory to output the position, orientation, velocity, acceleration, and angular velocity.

```
count = 1;
while ~isDone(traj)
    [Position(count,:),Orientation(count,:),Velocity(count,:), ...
     Acceleration(count,:),AngularVelocity(count,:)] = traj();
    count = count+1;
end
```

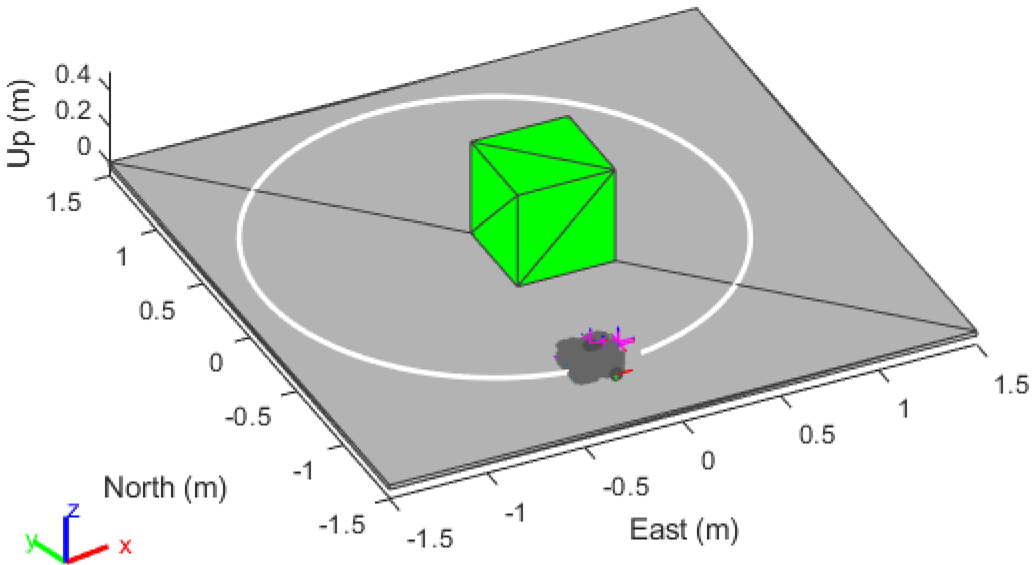
Create a line plot for the trajectory. First create the plot with `plot3`, then manually modify the data source properties of the plot. This improves the performance of the plotting.

```
trajPlot = plot3(nan,nan,nan,"Color",[1 1 1],"LineWidth",2);
trajPlot.XDataSource = "Position(:,1)";
trajPlot.YDataSource = "Position(:,2)";
trajPlot.ZDataSource = "Position(:,3)";
```

Set up the simulation. Then, iterate through the positions and show the scene each time the INS sensor updates. Advance the scene, move the robot platform, and update the sensors.

```
setup(scenario)
for idx = 1:count-1
    % Read sensor readings.
    [isUpdated,insTimestamp(idx,1),sensorReadings(idx)] = read(ins);
    if isUpdated
        % Use fast update to move platform visualization frames.
        show3D(scenario,FastUpdate=true,Parent=ax);
        % Refresh all plot data and visualize.
```

```
        refreshdata
        drawnow limitrate
    end
    % Advance scenario simulation time.
    advance(scenario);
    % Update all sensors in the scene.
    updateSensors(scenario)
end
hold off
```



Version History

Introduced in R2022a

See Also

Objects
robotPlatform | robotSensor | extendedObjectMesh | transformTree

Functions
addInertialFrame | addMesh | advance | binaryOccupancyMap | restart | setup | show3D | updateSensors

robotSensor

Sensor for robot scenario

Description

The `robotSensor` object creates a sensor that is rigidly attached to a robot platform, specified as a `robotPlatform` object. You can specify different mounting positions and orientations. Configure this object to automatically generate readings at fixed rate from a sensor specified as an `gpsSensor`, `insSensor`, or `robotLidarPointCloudGenerator` System object or `robotics.SensorAdaptor` class.

Creation

Syntax

```
sensor = robotSensor(name,platform,sensormodel)
sensor = robotSensor( ___ ,Name=Value)
```

Description

`sensor = robotSensor(name,platform,sensormodel)` creates a sensor with the specified name `name` and sensor model `sensormodel`, which set the Name and SensorModel properties respectively. The sensor is added to the platform `platform` specified as a `robotPlatform` object. The `platform` argument sets the MountingBodyName property.

`sensor = robotSensor( ___ ,Name=Value)` sets properties using one or more name-value pair arguments in addition to the input arguments in the previous syntax. You can specify the MountingLocation, MountingAngles, or UpdateRate properties as name-value pairs.

Properties

Name — Sensor name
string scalar

Sensor name, specified as a string scalar. Choose a name to identify this specific sensor.

Data Types: `char` | `string`

`SensorModel` — Sensor model for generating readings
`gpsSensor` System object | `insSensor` System object | `robotLidarPointCloudGenerator` System object

Sensor model for generating readings, specified as an `gpsSensor`, `insSensor`, or `robotLidarPointCloudGenerator` System object.

`MountingBodyName` — Name of sensor mounted platform body
`robotPlatform.Name` (default) | string scalar

Name of the sensor mounted platform body, specified as a string scalar. The `rigidBodyTree` based robot platform can have multiple bodies, any valid body can be selected to mount sensor.

Data Types: `char` | `string`

`MountingLocation` — Sensor position on platform
`[0 0 0]` (default) | vector of form `[x y z]`

Sensor position on platform, specified as a vector of the form `[x y z]` in the platform frame. Units are in meters.

Data Types: `single` | `double`

`MountingAngles` — Sensor orientation on platform
`[0 0 0]` (default) | vector of form `[z y x]`

Sensor orientation on platform, specified as a vector of the form `[z y x]` where *z*, *y*, and *x* are rotations about the *z*-axis, *y*-axis, and *x*-axis, sequentially, in degrees. The orientation is relative to the platform body frame.

Data Types: `single` | `double`

`UpdateRate` — Update rate of sensor
positive scalar

Update rate of the sensor, specified as a positive scalar in Hz. By default, the object uses the `UpdateRate` property of the specified sensor model object.

The sensor update interval `(1/UpdateRate)` must be a multiple of the update interval for the associated `robotScenario` object.

Data Types: `single` | `double`

Object Functions

read   Gather latest reading from robot sensor

Examples

Create and Simulate Robot Scenario

Create a robot scenario.

```
scenario = robotScenario(UpdateRate=100,StopTime=1);
```

Add the ground plane and a box as meshes.

```
addMesh(scenario,"Plane",Size=[3 3],Color=[0.7 0.7 0.7]);
addMesh(scenario,"Box",Size=[0.5 0.5 0.5],Position=[0 0 0.25], ...
        Color=[0 1 0])
```

Create a waypoint trajectory for the robot platform using an ENU reference frame.

```
waypoint = [0 -1 0; 1 0 0; -1 1 0; 0 -1 0];
toa = linspace(0,1,length(waypoint));
traj = waypointTrajectory("Waypoints",waypoint, ...
```

```
                                  "TimeOfArrival",toa, ...
                                  "ReferenceFrame","ENU");
```

Create a `rigidBodyTree` object of the TurtleBot 3 Waffle Pi robot with `loadrobot`.

```
robotRBT = loadrobot("robotisTurtleBot3WafflePi");
```

Create a robot platform with trajectory.

```
platform = robotPlatform("TurtleBot",scenario, ...
                         BaseTrajectory=traj);
```

Set up platform mesh with the `rigidBodyTree` object.

```
updateMesh(platform,"RigidBodyTree",Object=robotRBT)
```

Create an INS sensor object and attach the sensor to the platform.

```
ins = robotSensor("INS",platform,insSensor("RollAccuracy",0), ...
                  UpdateRate=scenario.UpdateRate);
```

Visualize the scenario.

```
[ax,plotFrames] = show3D(scenario);
axis equal
hold on
```

In a loop, step through the trajectory to output the position, orientation, velocity, acceleration, and angular velocity.

```
count = 1;
while ~isDone(traj)
    [Position(count,:),Orientation(count,:),Velocity(count,:), ...
     Acceleration(count,:),AngularVelocity(count,:)] = traj();
    count = count+1;
end
```

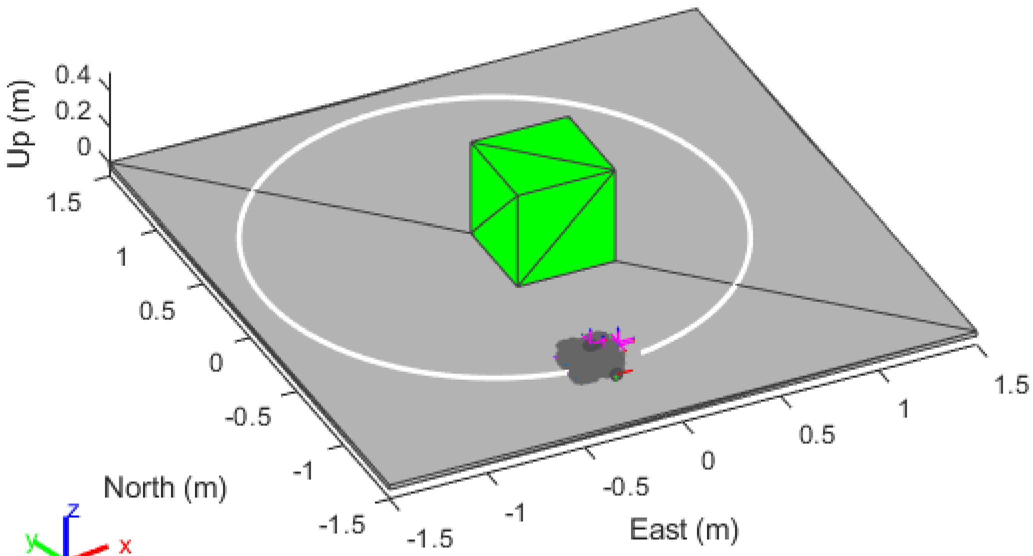
Create a line plot for the trajectory. First create the plot with `plot3`, then manually modify the data source properties of the plot. This improves the performance of the plotting.

```
trajPlot = plot3(nan,nan,nan,"Color",[1 1 1],"LineWidth",2);
trajPlot.XDataSource = "Position(:,1)";
trajPlot.YDataSource = "Position(:,2)";
trajPlot.ZDataSource = "Position(:,3)";
```

Set up the simulation. Then, iterate through the positions and show the scene each time the INS sensor updates. Advance the scene, move the robot platform, and update the sensors.

```
setup(scenario)
for idx = 1:count-1
    % Read sensor readings.
    [isUpdated,insTimestamp(idx,1),sensorReadings(idx)] = read(ins);
    if isUpdated
        % Use fast update to move platform visualization frames.
        show3D(scenario,FastUpdate=true,Parent=ax);
        % Refresh all plot data and visualize.
        refreshdata
        drawnow limitrate
    end
```

```
    % Advance scenario simulation time.
    advance(scenario);
    % Update all sensors in the scene.
    updateSensors(scenario)
end
hold off
```



Version History

Introduced in R2022a

See Also

Objects
robotPlatform | robotScenario | robotLidarPointCloudGenerator | gpsSensor | insSensor | robotics.SensorAdaptor

Functions
read

robotics.SensorAdaptor class

Package: `robotics`

Custom robot sensor interface

Description

The `robotics.SensorAdaptor` class is an interface for adapting custom sensor models to for use with the `robotScenario` object for robot scenario simulation.

The `robotics.SensorAdaptor` class is a `handle` class.

Class Attributes

`Abstract`	`true`

For information on class attributes, see "Class Attributes".

Creation

Syntax

`sensorObj = robotics.SensorAdaptor(sensorModel)`

Description

`sensorObj = robotics.SensorAdaptor(sensorModel)` creates a sensor object compatible with the `robotScenario` object. `sensorModel` is an object handle for a custom implementation of the `SensorAdaptor` class.

To get a template for a custom sensor implementation, use the `createCustomRobotSensorTemplate` function.

Properties

`UpdateRate` — Sensor update rate (Hz)
positive scalar

Sensor update rate, specified as a positive scalar in Hz.

Example: `10`

Data Types: `double`

`SensorModel` — Custom sensor model implementation
object handle

Custom sensor model implementation, specified as an object handle. To get a template for a custom sensor implementation, use the `createCustomRobotSensorTemplate` function.

Attributes:

SetAccess	private

Methods

Public Methods

setup	Set up custom sensor model
read	Read from custom sensor model
reset	Reset custom sensor model
getEmptyOutputs	Return empty sensor outputs without sensor inputs

Static Methods

robotics.SensorAdaptor.getMotion	Get sensor motion in platform reference frame

Examples

Simulate Ultrasonic Sensors Mounted on Mobile Robots

This example focuses on creating and mounting an ultrasonic sensor on a mobile robot in a `robotScenario`. The `ultrasonicDetectionGenerator` from the Automated Driving Toolbox cannot be used directly with `robotScenario`. We will be implementing a custom sensor adaptor for the `ultrasonicDetectionGenerator` that makes it compatible with `robotScenario`. The sensor will be used to position a mobile robot correctly at a charging station.

Create Custom Sensor Adaptor

Use the `createCustomRobotSensorTemplate` function to generate a template sensor and update it to adapt an `ultrasonicDetectionGenerator` object for usage in Robot scenario.

```
createCustomRobotSensorTemplate
```

This example provides the adaptor class `CustomUltrasonicSensor`, which can be viewed using the following command.

```
edit CustomUltrasonicSensor.m
```

Use the Sensor Adaptor in Robot Scenario Simulation

Create a `robotScenario` object with a sample rate of 10.

```
sampleRate = 10;
scenario = robotScenario(UpdateRate=sampleRate);
```

Add a plane mesh to show the warehouse floor.

```
addMesh(scenario,"Plane",Position=[5 0 0],Size=[20 12],Color=[0.7 0.7 0.7]);
```

Create a `waypointTrajectory` that traverses a set of waypoints to the charging station and use the `lookupPose` method of the `waypointTrajectory` object to fetch the pose of the robot along the trajectory.

```
startPosition = [-3 -3];
chargingPosition = [13 0];
```

```
wPts = [[startPosition 0.1]; ...
    5 0 0.1; ...
    10 0 0.1; ...
    13.75 0 0.1]; %Charging station

toa = [0 4 7 10];
traj = waypointTrajectory(Waypoints=wPts,...
    TimeOfArrival=toa, ReferenceFrame='ENU', ...
    SampleRate=sampleRate);
[pos, orient, vel, acc, angvel] = traj.lookupPose(0:1/sampleRate:10);
```

Add a `robotPlatform` to the scene for our mobile robot. Load the Clearpath Husky model for the `rigidBodyTree` of the `robotPlatform`. Also add cuboid meshes to denote obstacles in the scene. Add a 1-by-1 plane to denote where the charging station is.

```
robot = robotPlatform("rst", scenario,...
    RigidBodyTree=loadrobot("clearpathHusky"), ...
    InitialBasePosition=pos(1,:), InitialBaseOrientation=compact(orient(1)));

addMesh(scenario,"Box",Position=[3  5 2],Size=[4 2 4],Color=[1 0.5 0.25],IsBinaryOccupied=true);
addMesh(scenario,"Box",Position=[3 -5 2],Size=[4 2 4],Color=[1 0.5 0.25],IsBinaryOccupied=true);
addMesh(scenario,"Box",Position=[7  5 2],Size=[4 2 4],Color=[1 0.5 0.25],IsBinaryOccupied=true);
addMesh(scenario,"Box",Position=[7 -5 2],Size=[4 2 4],Color=[1 0.5 0.25],IsBinaryOccupied=true);
addMesh(scenario,"Box",Position=[-3  -5 0.5],Size=[1 1 1],Color=[0.1 0.1 0.1]);

% Plane to denote Charging station location
addMesh(scenario,"Plane",Position=[13 0 .05],Size=[1 1],Color=[0 1 0]);
```

Create the charging station using a `robotPlatform` object. The `robotPlatform` allows us to fetch the transform between the object and the sensor for use in the custom sensor read method. Here, the charging station can be modeled using a cuboid. The robot has to reach within 5cm of the surface of the charging station to start charging.

```
chargeStation = robotPlatform("chargeStation", scenario,InitialBasePosition=[13.75 0 0]);
chargeStation.updateMesh("Cuboid",Size=[0.5 1 1], Color=[0 0.8 0]);
```

The ultrasonic sensor model requires inputs of the profile of the obstacles to be detected. The profile structure includes information about the dimensions of the obstacle.

```
chargingStationProfile = struct("Length", 0.5, "Width", 1, "Height", 1, 'OriginOffset', [0 0 0]);
```

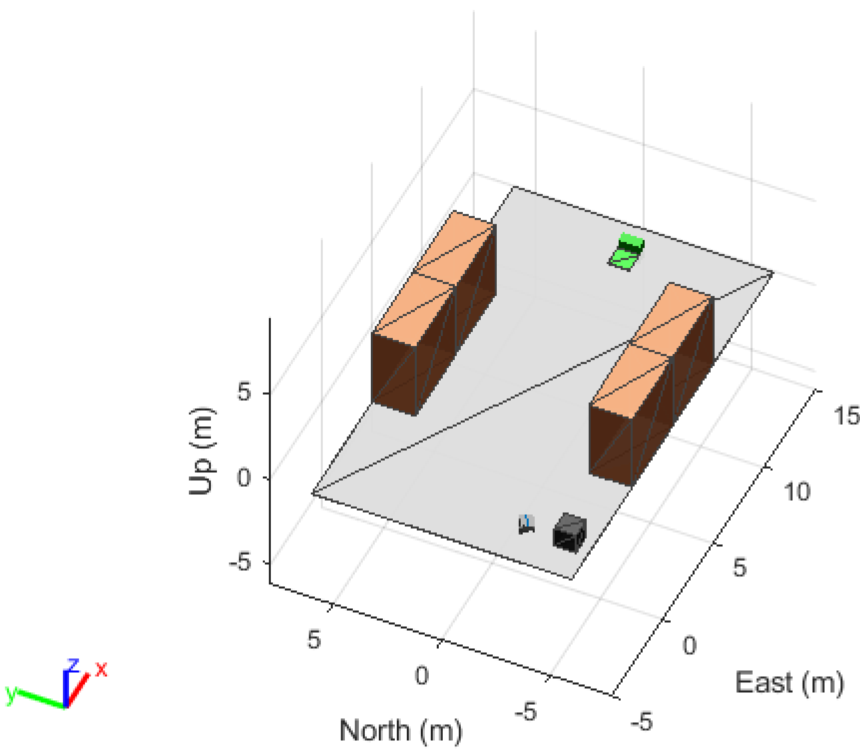
Create the ultrasonic sensor using the `ultrasonicDetectionGenerator` object and set its mounting location to `[0 0 0]`, detection range to `[0.03 0.04 5]` and field of view to `[70, 35]`. Also pass in the profile of the charging station that was created earlier.

```
ultraSonicSensorModel = ultrasonicDetectionGenerator(MountingLocation=[0 0 0], ...
    DetectionRange=[0.03 0.04 5], ...
    FieldOfView=[70, 35], ...
    Profiles=chargingStationProfile);
```

Create a `robotSensor` object that uses the custom sensor adaptor `CustomUltrasonicSensor`. The adaptor uses the ultrasonic sensor model created above. The mounting location will be at the front of the robot.

```
ult = robotSensor("UltraSonic", robot, ...
    CustomUltrasonicSensor(ultraSonicSensorModel), ...
    MountingLocation=[0.5 0 0.05]);
```

```
figure(1);
ax = show3D(scenario);
view(-65,45)
light
grid on
```

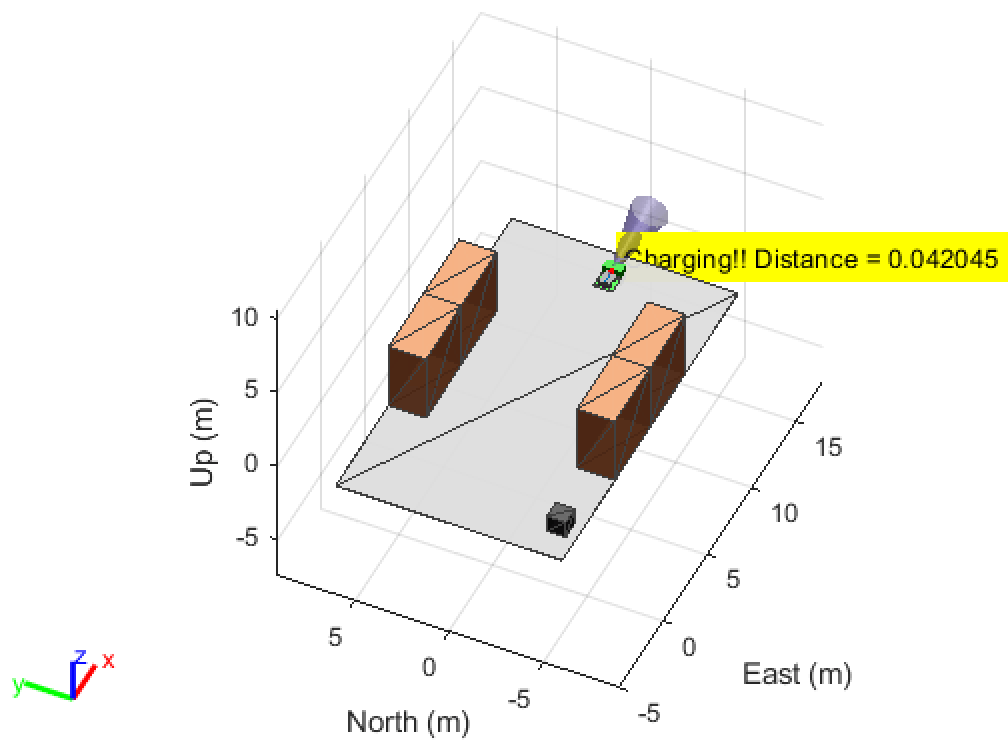


In this scene, the mobile robot will follow the trajectory to the charging station. When the ultrasonic sensor comes within a range of 20cm of the charging station, then mobile robot advance at a slower rate of 1cm per frame towards the charging station. When the robot is within 5cm of the surface of the charging station, it stops and the charging starts. The simulation ends when the charging starts.

```
isCharging = false;
i = 1;

setup(scenario);

while ~isCharging
    [isUpdated, t, det, isValid] = read(ult);

    figure(1);
    show3D(scenario);
    view(-65,45)
    light
    grid on

    % Read the motion vector of the robot from the platform ground truth
```

```
% This motion vector will be used only for plotting graphic elements
pose = robot.read();
rotAngle = quat2eul(pose(10:13));
hold on

if ~isempty(det)

    % Distance to object
    distance = det{1}.Measurement;

    % Plot a red shpere where the ultrasonic sensor detects an object
    exampleHelperPlotDetectionPoint(scenario, ...
        det{1}.ObjectAttributes{1}.PointOnTarget, ...
        ult.Name, ...
        pose);

    displayText = ['Distance = ',num2str(distance)];
else
    distance = inf;
    displayText = 'No object detected!';
end

% Plot a cone to represent the field of view and range of the ultrasonic sensor
exampleHelperPlotFOVCylinder(pose, ultraSonicSensorModel.DetectionRange(3));
hold off

if distance <= 0.2
    % Advance in steps of 1cm when the robot is within 20cm of the charging station
    currentMotion = lastMotion;
    currentMotion(1) = currentMotion(1) + 0.01;

    move(robot,"base",currentMotion);
    lastMotion = currentMotion;
    displayText = ['Detected Charger! Distance = ',num2str(distance)];
    if distance <= 0.05
        % The robot is charging when it is within 5cm of the charging station
        displayText = ['Charging!! Distance = ',num2str(distance)];
        isCharging = true;
    end
else
    % Follow the waypointTrajectory to the vicinity of the charging station
    if i<=length(pos)
        motion = [pos(i,:), vel(i,:), acc(i,:), ...
            compact(orient(i)), angvel(i,:)];
        move(robot,"base",motion);
        lastMotion = motion;
        i=i+1;
    end
end

% Display the distance to the charging station detected by the ultrasonic sensor
t = text(15, 0, displayText, "BackgroundColor",'yellow');
t(1).Color = 'black';
t(1).FontSize = 10;

advance(scenario);
```

```
    updateSensors(scenario);
end
```



Version History

Introduced in R2022b

See Also

Functions
setup | read | reset | getEmptyOutputs | robotics.SensorAdaptor.getMotion | createCustomRobotSensorTemplate

Objects
robotScenario | robotPlatform | robotSensor

se3

SE(3) homogeneous transformation

Description

The `se3` object represents an SE(3) transformation as a 3-D homogeneous transformation matrix consisting of a translation and rotation:

For more information, see the “3-D Homogeneous Transformation Matrix” on page 1-382 section.

This object acts like a numerical matrix enabling you to compose poses using multiplication and division.

Creation

Syntax

```
transformation = se3
transformation = se3(rotation)
transformation = se3(rotation,translation)
transformation = se3(transformation)

transformation = se3(euler,"eul")
transformation = se3(euler,"eul",sequence)
transformation = se3(quat,"quat")
transformation = se3(quaternion)
transformation = se3(axang,"axang")
transformation = se3(angle,axis)
transformation = se3( ___ ,translation,"trvec")

transformation = se3(translation,"trvec")
transformation = se3(pose,"xyzquat")
```

Description

Rotation Matrices, Translation Vectors, and Transformation Matrices

`transformation = se3` creates an SE(3) transformation representing an identity rotation with no translation.

$$transformation = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

`transformation = se3(rotation)` creates an SE(3) transformation representing a pure rotation defined by the orthonormal rotation `rotation` with no translation. The rotation matrix is represented by the elements in the top left of the `transformation` matrix.

$$rotation = \begin{bmatrix} r_{11} & r_{12} & r_{13} \\ r_{11} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{bmatrix}$$

$$transformation = \begin{bmatrix} r_{11} & r_{12} & r_{13} & 0 \\ r_{21} & r_{22} & r_{23} & 0 \\ r_{31} & r_{32} & r_{33} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

`transformation = se3(rotation,translation)` creates an SE(3) transformation representing a rotation defined by the orthonormal rotation `rotation` and the translation `translation`. The function applies the rotation matrix first, then translation vector to create the transformation.

$$rotation = \begin{bmatrix} r_{11} & r_{12} & r_{13} \\ r_{11} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{bmatrix}, translation = \begin{bmatrix} t_1 \\ t_2 \\ t_3 \end{bmatrix}$$

$$transformation = \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_1 \\ r_{21} & r_{22} & r_{23} & t_2 \\ r_{31} & r_{32} & r_{33} & t_3 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & t_1 \\ 0 & 1 & 0 & t_2 \\ 0 & 0 & 1 & t_3 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} r_{11} & r_{12} & r_{13} & 0 \\ r_{21} & r_{22} & r_{23} & 0 \\ r_{31} & r_{32} & r_{33} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

`transformation = se3(transformation)` creates an SE(3) transformation representing a translation and rotation as defined by the homogeneous transformation `transformation`.

Other 3-D Rotation Representations

`transformation = se3(euler,"eul")` creates an SE(3) transformation from the rotations defined by the Euler angles `euler`.

`transformation = se3(euler,"eul",sequence)` specifies the sequence of the Euler angle rotations `sequence`. For example, the sequence `"ZYX"` rotates the *z*-axis, then the *y*-axis and *x*-axis.

`transformation = se3(quat,"quat")` creates an SE(3) transformation from the rotations defined by the numeric quaternions `quat`.

`transformation = se3(quaternion)` creates an SE(3) transformation from the rotations defined by the quaternion `quaternion`.

`transformation = se3(axang,"axang")` creates an SE(3) transformation from the rotations defined by the axis-angle rotation `axang`.

`transformation = se3(angle,axis)` creates an SE(3) transformation from the rotations `angles` about the rotation axis `axis`.

`transformation = se3( ___ ,translation,"trvec")` creates an SE(3) transformation from the translation vector `translation` along with any other type of rotation input arguments.

Other Translations and Transformation Representations

`transformation = se3(translation,"trvec")` creates an SE(3) transformation from the translation vector `translation`.

`transformation = se3(pose,"xyzquat")` creates an SE(3) transformation from the 3-D compact pose `pose`.

Note If any inputs contain more than one rotation, translation, or transformation, then the output `transformation` is an *N*-element array of `se3` objects corresponding to each of the *N* input rotations, translations, or transformations.

Input Arguments

`rotation` — Orthonormal rotation

3-by-3 matrix | 3-by-3-by-*N* matrix | `so3` object | *N*-element array of `so3` objects

Orthonormal rotation, specified as a 3-by-3 matrix, a 3-by-3-by-*N* array, a scalar `so3` object, or an *N*-element array of `so3` objects. *N* is the total number of rotations.

If `rotation` contains more than one rotation and you also specify `translation` at construction, the number of translations in `translation` must be one or equal to the number of rotations in `rotation`. The resulting number of transformation objects is equal to the value of the `translation` or `rotation` argument, whichever is larger.

Example: `eye(3)`

Data Types: `single` | `double`

`translation` — Translation

three-element row vector | *N*-by-3 matrix

Translation, specified as a three-element row vector or an *N*-by-3 array. *N* is the total number of translations and each translation is of the form [*x y z*].

If `translation` contains more than one translation, the number of rotations in `rotation` must be one or equal to the number of translations in `translation`. The resulting number of created transformation objects is equal to the value of the `translation` or `rotation` argument, whichever is larger.

Example: `[1 4 3]`

Data Types: `single` | `double`

`transformation` — Homogeneous transformation

4-by-4 matrix | 4-by-4-*N* array | `se3` object | *N*-element array of `se3` objects

Homogeneous transformation, specified as a 4-by-4 matrix, a 4-by-4-*N* array, a scalar `se3` object, or an *N*-element array of `se3` objects. *N* is the total number of transformations specified.

If `transformation` is an array, the resulting number of created `se3` objects is equal to *N*.

Example: `eye(4)`

Data Types: `single` | `double`

`quaternion` — Quaternion

`quaternion` object | *N*-element array of `quaternion` objects

Quaternion, specified as a scalar `quaternion` object or as an *N*-element array of `quaternion` objects. *N* is the total number of specified quaternions.

If `quaternion` is an *N*-element array, the resulting number of created `se3` objects is equal to *N*.

Example: `quaternion(1,0.2,0.4,0.2)`

`euler` — Euler angles
N-by-3 matrix

Euler angles, specified as an *N*-by-3 matrix, in radians. Each row represents one set of Euler angles with the axis-rotation sequence defined by the `sequence` argument. The default axis-rotation sequence is *ZYX*.

If `euler` is an *N*-by-3 matrix, the resulting number of created `se3` objects is equal to *N*.

Example: `[pi/2 pi pi/4]`

Data Types: `single` | `double`

`sequence` — Axis-rotation sequence
`"ZYX"` (default) | `"ZYZ"` | `"ZXY"` | `"ZXZ"` | `"YXY"` | `"YZX"` | `"YXZ"` | `"YZY"` | `"XYX"` | `"XYZ"` | `"XZX"` | `"XZY"`

Axis-rotation sequence for the Euler angles, specified as one of these string scalars:

- `"ZYX"` (default)
- `"ZYZ"`
- `"ZXY"`
- `"ZXZ"`
- `"YXY"`
- `"YZX"`
- `"YXZ"`
- `"YZY"`
- `"XYX"`
- `"XYZ"`
- `"XZX"`
- `"XZY"`

These are orthonormal rotation matrices for rotations of ϕ, ψ, and θ about the *x*-, *y*-, and *z*-axis, respectively:

$$R_x(\phi) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\phi & \sin\phi \\ 0 & -\sin\phi & \cos\phi \end{bmatrix}, R_y(\psi) = \begin{bmatrix} \cos\psi & 0 & \sin\psi \\ 0 & 1 & 0 \\ -\sin\psi & 0 & \cos\psi \end{bmatrix}, R_z(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

When constructing the rotation matrix from this sequence, each character indicates the corresponding axis. For example, if the sequence is `"XYZ"`, then the `se3` object constructs the rotation matrix *R* by multiplying the rotation about *x*-axis with the rotation about the *y*-axis, and then multiplying that product with the rotation about the *z*-axis:

$$R = R_x(\phi) \cdot R_y(\psi) \cdot R_z(\theta)$$

Example: `se3([pi/2 pi/3 pi/4],"eul","ZYZ")` rotates a point by `pi/4` radians about the *z*-axis, then rotates the point by `pi/3` radians about the *y*-axis, and then rotates the point by `pi/2`

radians about the *z*-axis. This is equivalent to `se3(pi/2,"rotz") * se3(pi/3,"roty") * se3(pi/4,"rotz")`

Data Types: `string` | `char`

quat — Quaternion
N-by-4 matrix

Quaternion, specified as an *N*-by-4 matrix. *N* is the number of specified quaternions. Each row represents one quaternion of the form [*qw qx qy qz*], where *qw* is a scalar number.

If `quat` is an *N*-by-4 matrix, the resulting number of created `se3` objects is equal to *N*.

Note The `se3` object normalizes the input quaternions before converting the quaternions to a rotation matrix.

Example: `[0.7071 0.7071 0 0]`

Data Types: `single` | `double`

axang — Axis-angle rotation
N-by-4 matrix

Axis-angle rotation, specified as an *N*-by-4 matrix in the form [*x y z theta*]. *N* is the total number of axis-angle rotations. *x*, *y*, and *z* are vector components from the *x*-, *y*-, and *z*-axis, respectively. The vector defines the axis to rotate by the angle *theta*, in radians.

If `axang` is an *N*-by-4 matrix, the resulting number of created `se3` objects is equal to *N*.

Example: `[.2 .15 .25 pi/4]` rotates the axis, defined as `0.2` in the *x*-axis, `0.15` along the *y*-axis, and `0.25` along the *z*-axis, by `pi/4` radians.

Data Types: `single` | `double`

angle — Single-axis-angle rotation
N-by-*M* matrix

Single-axis-angle rotation, specified as an *N*-by-*M* matrix. Each element of the matrix is an angle, in radians, about the axis specified using the `axis` argument, and the `se3` object creates an `se3` object for each angle.

If `angle` is an *N*-by-*M* matrix, the resulting number of created `se3` objects is equal to *N*.

The rotation angle is counterclockwise positive when you look along the specified axis toward the origin.

Data Types: `single` | `double`

axis — Axis to rotate
`"rotx"` | `"roty"` | `"rotz"`

Axis to rotate, specified as one of these options:

- `"rotx"` — Rotate about the *x*-axis:

$$R_x(\phi) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\phi & \sin\phi \\ 0 & -\sin\phi & \cos\phi \end{bmatrix}$$

- `"roty"` — Rotate about the *y*-axis:

$$R_y(\psi) = \begin{bmatrix} \cos\psi & 0 & \sin\psi \\ 0 & 1 & 0 \\ -\sin\psi & 0 & \cos\psi \end{bmatrix}$$

- `"rotz"` — Rotate about the *z*-axis:

$$R_z(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Use the `angle` argument to specify how much to rotate about the specified axis.

Example: `Rx = se3(phi,"rotx");`

Example: `Ry = se3(psi,"roty");`

Example: `Rz = se3(theta,"rotz");`

Data Types: `string` | `char`

pose — 3-D compact pose
N-by-7 matrix

3-D compact pose, specified as an *N*-by-7 matrix, where *N* is the total number of compact poses. Each row is a pose, comprised of a xyz position and quaternion, in the form [*x y z qw qx qy qz*]. *x*, *y*, and *z* are the positions in the *x*-, *y*-, and *z*-axes, respectively. *qw*, *qx*, *qy*, and *qz* together are the quaternion rotation in *w*, *x*, *y*, and *z*, respectively.

If `pose` is an *N*-by-7 matrix, the resulting number of created `se3` objects is equal to *N*.

Data Types: `single` | `double`

Object Functions

Mathematical Operations

mtimes, *	Transformation or rotation multiplication
mrdivide, /	Transformation or rotation right division
rdivide, ./	Element-wise transformation or rotation right division
times, .*	Element-wise transformation or rotation multiplication

Utilities

interp	Interpolate between transformations
dist	Calculate distance between transformations
normalize	Normalize transformation or rotation matrix
transform	Apply rigid body transformation to points

Numerical Conversions

axang	Convert transformation or rotation into axis-angle rotations
eul	Convert transformation or rotation into Euler angles
rotm	Extract rotation matrix
quat	Convert transformation or rotation to numeric quaternion
quaternion	Create a quaternion array
trvec	Extract translation vector
tform	Extract homogeneous transformation
xyzquat	Convert transformation or rotation to compact 3-D pose representation

Object Conversions

so3	SO(3) rotation

Examples

Create SE(3) Transformation Using Euler Angles and Translation

Define an Euler-angle rotation of `[pi/2 0 pi/7]` with a `"XYZ"` rotation sequence, and a *xyz* translation of `[6 4 1]`.

```
angles = [pi/2 0 pi/7];
trvec = [6 4 1];
```

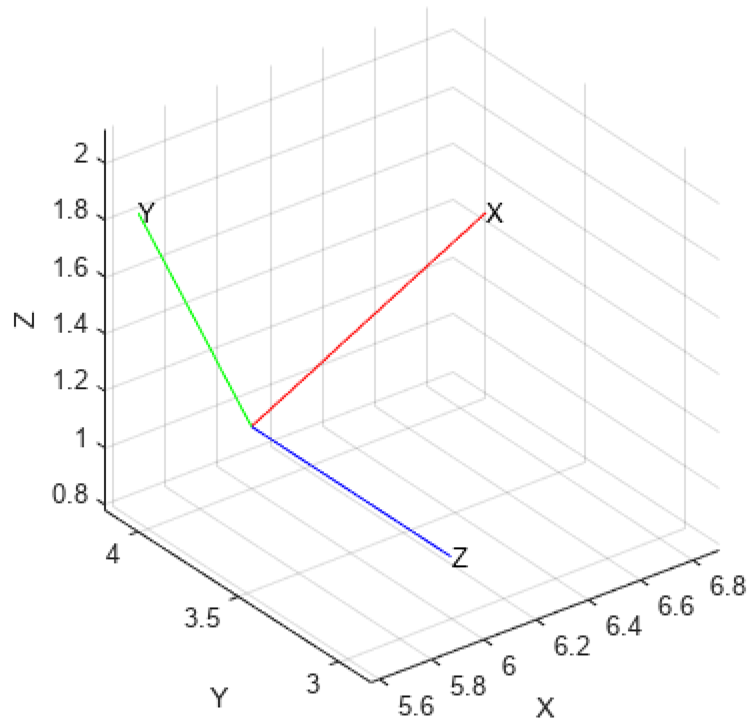
Create an SE(3) transformation using the Euler angles and the translation.

```
TF = se3(angles,"eul","XYZ",trvec)
```

```
TF = se3
    0.9010   -0.4339         0    6.0000
    0.0000    0.0000   -1.0000    4.0000
    0.4339    0.9010    0.0000    1.0000
         0         0         0    1.0000
```

Plot the transformation.

```
plotTransforms(TF,AxisLabels="on",FrameAxisLabels="on");
```



Algorithms

3-D Homogeneous Transformation Matrix

3-D homogeneous transformation matrices consist of both an SO(3) rotation and an *xyz*-translation.

To read more about SO(3) rotations, see the “3-D Orthonormal Rotation Matrix” on page 1-395 section of the `so3` object.

The translation is along the *x*-, *y*-, and *z*-axes as a three-element column vector:

$$t = \begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

The SO(3) rotation matrix R is applied to the translation vector t to create the homogeneous translation matrix T. The rotation matrix is present in the upper-left of the transformation matrix as 3-by-3 submatrix, and the translation vector is present as a three-element vector in the last column.

$$T = \begin{bmatrix} R & t \\ 0_{1x3} & 1 \end{bmatrix} = \begin{bmatrix} I_3 & t \\ 0_{1x3} & 1 \end{bmatrix} \cdot \begin{bmatrix} R & 0 \\ 0_{1x3} & 1 \end{bmatrix}$$

Version History

Introduced in R2022b

R2023a: New methods and syntaxes

se3 supports new methods and syntaxes for converting to and from other transformations and rotations.

The new methods are:

- axang
- eul
- so3
- quat
- quaternion
- xyzquat

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

Functions
axang2tform | eul2tform | quat2tform | rotm2tform | trvec2tform | plotTransforms

Objects
se2 | so2 | so3 | quaternion

se2

SE(2) homogeneous transformation

Description

The `se2` object represents an SE(2) transformation as a 2-D homogeneous transformation matrix consisting of a translation and rotation.

For more information, see the "2-D Homogeneous Transformation Matrix" on page 1-388 section.

This object acts like a numerical matrix, enabling you to compose poses using multiplication and division.

Creation

Syntax

```
transformation = se2
transformation = se2(rotation)
transformation = se2(rotation,translation)
transformation = se2(transformation)
```

```
transformation = se2(angle,"theta")
transformation = se2(angle,"theta",translation)
transformation = se2(translation,"trvec")
transformation = se2(pose,"xytheta")
```

Description

Rotation Matrices, Translation Vectors, and Transformation Matrices

`transformation = se2` creates an SE(2) transformation representing an identity rotation with no translation.

$$transformation = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

`transformation = se2(rotation)` creates an SE(2) transformation representing a pure rotation defined by the orthonormal rotation `rotation` with no translation. The rotation matrix is represented by the elements in the top left of the `transformation` matrix.

$$rotation = \begin{bmatrix} r_{11} & r_{12} \\ r_{21} & r_{22} \end{bmatrix}$$

$$transformation = \begin{bmatrix} r_{11} & r_{12} & 0 \\ r_{21} & r_{22} & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

`transformation = se2(rotation,translation)` creates an SE(2) transformation representing a rotation defined by the orthonormal rotation `rotation` and the translation `translation`. The function applies the rotation matrix first, then the translation vector, to create the transformation.

$$rotation = \begin{bmatrix} r_{11} & r_{12} \\ r_{21} & r_{22} \end{bmatrix}, translation = \begin{bmatrix} t_1 \\ t_2 \end{bmatrix}$$

$$transformation = \begin{bmatrix} r_{11} & r_{12} & t_1 \\ r_{21} & r_{22} & t_2 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & t_1 \\ 0 & 1 & t_2 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} r_{11} & r_{12} & 0 \\ r_{21} & r_{22} & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

`transformation = se2(transformation)` creates an SE(2) transformation representing a translation and rotation as defined by the homogeneous transformation `transformation`.

Other 2-D Rotations and Transformation Representations

`transformation = se2(angle,"theta")` creates SE(2) transformations `transformation` from rotations around the *z*-axis, in radians. The transformation contains zero translation.

`transformation = se2(angle,"theta",translation)` creates SE(2) transformations from rotations around the *z*-axis, in radians, with translations `translation`.

`transformation = se2(translation,"trvec")` creates an SE(2) transformation from the translation vector `translation`.

`transformation = se2(pose,"xytheta")` creates an SE(2) transformation from the 2-D compact pose `pose`.

Input Arguments

`rotation` — Orthonormal rotation

2-by-2 matrix | 2-by-2-by-*N* matrix | `so2` object | *N*-element array of `so2` objects

Orthonormal rotation, specified as a 2-by-2 matrix, a 2-by-2-by-*N* array, a scalar `so2` object, or an *N*-element array of `so2` objects. *N* is the total number of rotations.

If `rotation` contains more than one rotation and you also specify `translation` at construction, the number of translations in `translation` must be one or equal to the number of rotations in `rotation`. The resulting number of transformation objects is equal to the value of the `translation` or `rotation` argument, whichever is larger.

If `rotation` contains one rotation and you also specify `translation` as an *N*-by-2 matrix, then the resulting transformations contain the same rotation specified by `rotation` and the corresponding translation vector in `translation`. The resulting number of transformation objects is equal to the number of translations in `translation`.

Example: `eye(2)`

Data Types: `single` | `double`

`translation` — Translation

two-element row vector | *N*-by-2 matrix

Translation, specified as an *N*-by-2 matrix. *N* is the total number of translations and each translation is of the form [*x y*].

If `translation` contains more than one translation, the number of rotations in `rotation` must be one or equal to the number of translations in `translation`. The resulting number of created transformation objects is equal to the value of the `translation` or `rotation` argument, whichever is larger.

If you specify more than one translation but only one rotation, the resulting transformations contain the same rotation specified in `rotation` and the corresponding translation in `translation`. The resulting number of created `se2` objects is equal to the value of the `translation`.

Example: `[1 4]`

Data Types: `single` | `double`

`transformation` — Homogeneous transformation
3-by-3 matrix | 3-by-3-*N* array | `se2` object | *N*-element array of `se2` objects

Homogeneous transformation, specified as a 3-by-3 matrix, a 3-by-3-*N* array, a scalar `se3` object, or an *N*-element array of `se2` objects. *N* is the total number of transformations specified.

If `transformation` is an array, the resulting number of created `se2` objects is equal to *N*.

Example: `eye(3)`

Data Types: `single` | `double`

`angle` — *z*-axis rotation angle
N-by-*M* matrix

z-axis rotation angle, specified as an *N*-by-*M* matrix. Each element of the matrix is an angle, in radians, about the *z*-axis. The `se2` object creates an `se2` object for each angle.

If `angle` is an *N*-by-*M* matrix, the resulting number of created `se2` objects is equal to *N*.

The rotation angle is counterclockwise positive when you look along the specified axis toward the origin.

Data Types: `single` | `double`

`pose` — 2-D compact pose
N-by-3 matrix

3-D compact pose, specified as an *N*-by-3 matrix, where *N* is the total number of compact poses. Each row is a pose, comprised of an xy position and a rotation about the *z*-axis, in the form [*x y theta*]. *x*, *y* are the xy-positions and *theta* is the rotation about the *z*-axis.

If `pose` is an *N*-by-3 matrix, the resulting number of created `se2` objects is equal to *N*.

Data Types: `single` | `double`

Object Functions

Mathematical Operations

mtimes, *	Transformation or rotation multiplication
mrdivide, /	Transformation or rotation right division
rdivide, ./	Element-wise transformation or rotation right division

times, .*	Element-wise transformation or rotation multiplication

Utilities

interp	Interpolate between transformations
dist	Calculate distance between transformations
normalize	Normalize transformation or rotation matrix
transform	Apply rigid body transformation to points

Numerical Conversions

rotm	Extract rotation matrix
trvec	Extract translation vector
tform	Extract homogeneous transformation
theta	Convert transformation or rotation to 2-D rotation angle
xytheta	Convert transformation or rotation to compact 2-D pose representation

Object Conversions

so2	SO(2) rotation

Examples

Create SE(2) Transformation Using Angle and Translation

Define an angle rotation of `pi/4` and a *xyz* translation of `[6 4]`.

```
angle = pi/6;
trvec = [2 1];
```

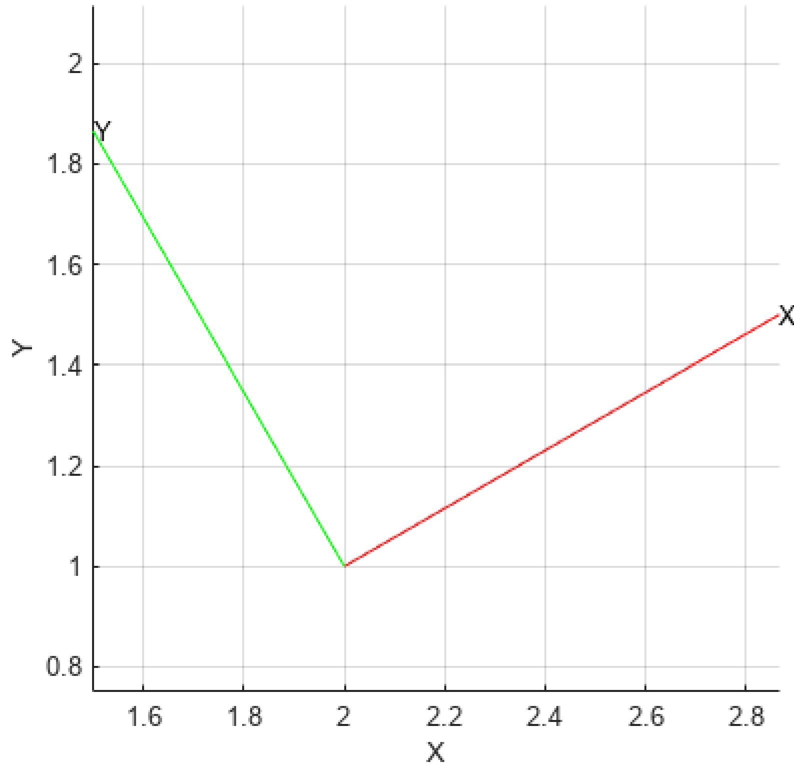
Create an SE(2) transformation using the angle and translation.

```
TF = se2(angle,"theta",trvec)
```

```
TF = se2
    0.8660   -0.5000    2.0000
    0.5000    0.8660    1.0000
         0         0    1.0000
```

Plot the transformation.

```
plotTransforms(TF,AxisLabels="on",FrameAxisLabels="on");
```



Algorithms

2-D Homogeneous Transformation Matrix

2-D homogeneous transformation matrices consist of both an SO(2) rotation and an *xy*-translation.

To read more about SO(2) rotations, see the “2-D Orthonormal Rotation Matrix” on page 1-399 section of the `so2` object.

The translation is along the *x*-, *y*-, and *z*-axes as a three-element column vector:

$$t = \begin{bmatrix} x \\ y \end{bmatrix}$$

The SO(2) rotation matrix R is applied to the translation vector t to create the homogeneous translation matrix T. The rotation matrix is present in the upper-left of the transformation matrix as 2-by-2 submatrix, and the translation vector is present as a two-element vector in the last column.

$$T = \begin{bmatrix} R & t \\ 0_{1 \times 2} & 1 \end{bmatrix} = \begin{bmatrix} I_2 & t \\ 0_{1 \times 2} & 1 \end{bmatrix} \cdot \begin{bmatrix} R & 0 \\ 0_{1 \times 2} & 1 \end{bmatrix}$$

Version History

Introduced in R2022b

R2023a: New methods and syntaxes

se2 supports new methods and syntaxes for converting to and from other transformations and rotations.

The new methods are:

- se3
- so2
- theta
- xytheta

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

Functions
axang2tform | eul2tform | quat2tform | rotm2tform | trvec2tform | plotTransforms

Objects
se3 | so2 | so3 | quaternion

so3

SO(3) rotation

Description

The `so3` object represents an SO(3) rotation in 3-D in a right-handed Cartesian coordinate system.

The SO(3) rotation is a 3-by-3 orthonormal rotation matrix. For example, these are orthonormal rotation matrices for rotations of ϕ, ψ, and θ about the *x*-, *y*-, and *z*-axis, respectively:

$$R_x(\phi) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\phi & \sin\phi \\ 0 & -\sin\phi & \cos\phi \end{bmatrix}, R_y(\psi) = \begin{bmatrix} \cos\psi & 0 & \sin\psi \\ 0 & 1 & 0 \\ -\sin\psi & 0 & \cos\psi \end{bmatrix}, R_z(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

For more information, see the 3-D Orthonormal Rotation Matrix section.

This object acts like a numerical matrix, enabling you to compose rotations using multiplication and division.

Creation

Syntax

```
rotation = so3
rotation = so3(rotation)
rotation = so3(quaternion)
rotation = so3(transformation)

rotation = so3(euler,"eul")
rotation = so3(euler,"eul",sequence)
rotation = so3(quat,"quat")
rotation = so3(axang,"axang")
rotation = so3(angle,axis)
```

Description

3-D Rotation Representations

`rotation = so3` creates an SO(3) rotation representing an identity rotation with no translation.

$$rotation = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

`rotation = so3(rotation)` creates an SO(3) rotation representing a pure rotation defined by the orthonormal rotation `rotation`.

$$rotation = \begin{bmatrix} r_{11} & r_{12} & r_{13} \\ r_{11} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{bmatrix}$$

`rotation = so3(quaternion)` creates an SO(3) rotation from the rotations defined by the quaternion `quaternion`.

`rotation = so3(transformation)` creates an SO(3) rotation from the SE(3) transformation `transformation`.

Other Numeric 3-D Rotation Representations

`rotation = so3(euler,"eul")` creates an SO(3) rotation from the rotations defined by the Euler angles `euler`.

`rotation = so3(euler,"eul",sequence)` specifies the sequence of the Euler angle rotations `sequence`. For example, the sequence `"ZYX"` rotates the *z*-axis, then the *y*-axis and *x*-axis.

`rotation = so3(quat,"quat")` creates an SO(3) rotation from the rotations defined by the numeric quaternions `quat`.

`rotation = so3(axang,"axang")` creates an SO(3) rotation from the rotations defined by the axis-angle rotation `axang`.

`rotation = so3(angle,axis)` creates an SO(3) rotation from the rotations `angles` about the rotation axis `axis`.

Note If any inputs contain more than one rotation, then the output `rotation` is an *N*-element array of `so3` objects corresponding to each of the *N* input rotations.

Input Arguments

`rotation` — Orthonormal rotation

3-by-3 matrix | 3-by-3-by-*N* matrix | `so3` object | *N*-element array of `so3` objects

Orthonormal rotation, specified as a 3-by-3 matrix, a 3-by-3-by*N* array, a scalar `so3` object, or an *N*-element array of `so3` objects. *N* is the total number of rotations.

If `rotation` is an array, the resulting number of created `so3` objects in the output array is equal to *N*.

Example: `eye(3)`

`transformation` — Homogeneous transformation

`se3` object | *N*-element array of `se3` objects

Homogeneous transformation, specified as an `se3` object or a *N*-element array of `se3` objects. *N* is the total number of transformations specified.

The output `so3` object contains only the rotational submatrix of the `se3` object.

If `transformation` is an array, the resulting number of created `so3` objects in the output array is equal to *N*.

Example: `se3(pi/4,"rotx")`

`quaternion` — Quaternion
`quaternion` object | *N*-element array of `quaternion` objects

Quaternion, specified as a scalar `quaternion` object or as an *N*-element array of `quaternion` objects. *N* is the total number of specified quaternions.

If `quaternion` is an *N*-element array, the resulting number of created `so3` objects is equal to *N*.

Example: `quaternion(1,0.2,0.4,0.2)`

`euler` — Euler angles
N-by-3 matrix

Euler angles, specified as an *N*-by-3 matrix, in radians. Each row represents one set of Euler angles with the axis-rotation sequence defined by the `sequence` argument. The default axis-rotation sequence is *ZYX*.

If `euler` is an *N*-by-3 matrix, the resulting number of created `so3` objects is equal to *N*.

Example: `[pi/2 pi pi/4]`

Data Types: `single` | `double`

`sequence` — Axis-rotation sequence
`"ZYX"` (default) | `"ZYZ"` | `"ZXY"` | `"ZXZ"` | `"YXY"` | `"YZX"` | `"YXZ"` | `"YZY"` | `"XYX"` | `"XYZ"` | `"XZX"` | `"XZY"`

Axis-rotation sequence for the Euler angles, specified as one of these string scalars:

- `"ZYX"` (default)
- `"ZYZ"`
- `"ZXY"`
- `"ZXZ"`
- `"YXY"`
- `"YZX"`
- `"YXZ"`
- `"YZY"`
- `"XYX"`
- `"XYZ"`
- `"XZX"`
- `"XZY"`

These are orthonormal rotation matrices for rotations of ϕ, ψ, and θ about the *x*-, *y*-, and *z*-axis, respectively:

$$R_x(\phi) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\phi & \sin\phi \\ 0 & -\sin\phi & \cos\phi \end{bmatrix}, R_y(\psi) = \begin{bmatrix} \cos\psi & 0 & \sin\psi \\ 0 & 1 & 0 \\ -\sin\psi & 0 & \cos\psi \end{bmatrix}, R_z(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

When constructing the rotation matrix from this sequence, each character indicates the corresponding axis. For example, if the sequence is `"XYZ"`, then the `so3` object constructs the

rotation matrix *R* by multiplying the rotation about *x*-axis with the rotation about the *y*-axis, and then multiplying that product with the rotation about the *z*-axis:

$$R = R_x(\phi) \cdot R_y(\psi) \cdot R_z(\theta)$$

Example: `so3([pi/2 pi/3 pi/4],"eul","ZYZ")` rotates a point by `pi/4` radians about the *z*-axis, then rotates the point by `pi/3` radians about the *y*-axis, and then rotates the point by `pi/2` radians about the *z*-axis. This is equivalent to `so3(pi/2,"rotz") * so3(pi/3,"roty") * so3(pi/4,"rotz")`

Data Types: `string` | `char`

`quat` — Quaternion
N-by-4

Quaternion, specified as an *N*-by-4 matrix. *N* is the number of specified quaternions. Each row represents one quaternion of the form [*qw qx qy qz*], where *qw* is a scalar number.

If `quat` is an *N*-by-4 matrix, the resulting number of created `so3` objects is equal to *N*.

Note The `so3` object normalizes the input quaternions before converting the quaternions to a rotation matrix.

Example: `[0.7071 0.7071 0 0]`

Data Types: `single` | `double`

`axang` — Axis-angle rotation
N-by-4 matrix

Axis-angle rotation, specified as an *N*-by-4 matrix in the form [*x y z theta*]. *N* is the total number of axis-angle rotations. *x*, *y*, and *z* are vector components from the *x*-, *y*-, and *z*-axis, respectively. The vector defines the axis to rotate by the angle *theta*, in radians.

If `axang` is an *N*-by-4 matrix, the resulting number of created `so3` objects is equal to *N*.

Example: `[.2 .15 .25 pi/4]` rotates the axis, defined as `0.2` in the *x*-axis, `0.15` along the *y*-axis, and `0.25` along the *z*-axis, by `pi/4` radians.

Data Types: `single` | `double`

`angle` — Single-axis-angle rotation
N-by-*M* matrix

Single-axis-angle rotation, specified as an *N*-by-*M* matrix. Each element of the matrix is an angle, in radians, about the axis specified using the `axis` argument, and the `so3` object creates an `so3` object for each angle.

If `angle` is an *N*-by-*M* matrix, the resulting number of created `so3` objects is equal to *N*.

The rotation angle is counterclockwise positive when you look along the specified axis toward the origin.

Data Types: `single` | `double`

axis — Axis to rotate
"rotx" | "roty" | "rotz"

Axis to rotate, specified as one of these options:

- "rotx" — Rotate about the *x*-axis:

$$R_x(\phi) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\phi & \sin\phi \\ 0 & -\sin\phi & \cos\phi \end{bmatrix}$$

- "roty" — Rotate about the *y*-axis:

$$R_y(\psi) = \begin{bmatrix} \cos\psi & 0 & \sin\psi \\ 0 & 1 & 0 \\ -\sin\psi & 0 & \cos\psi \end{bmatrix}$$

- "rotz" — Rotate about the *z*-axis:

$$R_z(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Use the `angle` argument to specify how much to rotate about the specified axis.

Example: `Rx = so3(phi,"rotx");`

Example: `Ry = so3(psi,"roty");`

Example: `Rz = so3(theta,"rotz");`

Data Types: `string` | `char`

Object Functions

Mathematical Operations

mtimes, *	Transformation or rotation multiplication
mrdivide, /	Transformation or rotation right division
rdivide, ./	Element-wise transformation or rotation right division
times, .*	Element-wise transformation or rotation multiplication

Utilities

interp	Interpolate between transformations
dist	Calculate distance between transformations
normalize	Normalize transformation or rotation matrix
transform	Apply rigid body transformation to points

Numerical Conversions

axang	Convert transformation or rotation into axis-angle rotations
eul	Convert transformation or rotation into Euler angles
rotm	Extract rotation matrix
quat	Convert transformation or rotation to numeric quaternion
trvec	Extract translation vector

tform	Extract homogeneous transformation
xyzquat	Convert transformation or rotation to compact 3-D pose representation

Object Conversions

se3	SE(3) homogeneous transformation
quaternion	Create a quaternion array

Examples

Convert SO(3) Rotation to Euler Angles

Create SO(3) rotation defined by a Euler angles.

```
eul1 = [pi/4 pi/3 pi/8]
```

```
eul1 = 1×3

    0.7854    1.0472    0.3927
```

```
R = so3(eul1,"eul")
```

```
R = so3
    0.3536   -0.4189    0.8364
    0.3536    0.8876    0.2952
   -0.8660    0.1913    0.4619
```

Get the Euler angles from the transformation.

```
eul2 = eul(R)
```

```
eul2 = 1×3

    0.7854    1.0472    0.3927
```

Algorithms

3-D Orthonormal Rotation Matrix

SO(3) rotation matrices are 3-by-3 orthonormal matrices that represent any rotation in 3-D Euclidean space. SO(3) rotations have many special properties. For example, SO(3) rotation matrices are in the 3-D special orthogonal group, so the product of two SO(3) rotation matrices is an SO(3) rotation matrix. This enables you to compose rotations from multiple rotations. For example, these are orthonormal rotation matrices for rotations of ϕ, ψ, and θ about the *x*-, *y*-, and *z*-axis, respectively:

$$R_x(\phi) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\phi & \sin\phi \\ 0 & -\sin\phi & \cos\phi \end{bmatrix}, R_y(\psi) = \begin{bmatrix} \cos\psi & 0 & \sin\psi \\ 0 & 1 & 0 \\ -\sin\psi & 0 & \cos\psi \end{bmatrix}, R_z(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Depending on the order in which you multiply these *x*-, *y*-, and *z*-axis rotations, you can construct compound matrices that represent any rotation in 3-D Euclidean space.

These are other properties of SO(3) rotations:

- Each column is both orthogonal and a unit vector, meaning that none of the columns are multiples of each other.
- The determinant of the matrix is positive 1.
- The inverse of the matrix is the same as the transpose of the matrix: $R^{-1} = R^T$.

Version History

Introduced in R2022b

R2023a: New methods and syntaxes

`so3` supports new methods and syntaxes for converting to and from other transformations and rotations.

The new methods are:

- `axang`
- `eul`
- `quat`
- `quaternion`
- `tform`
- `trvec`
- `xyzquat`

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

Functions

`axang2rotm` | `eul2rotm` | `quat2rotm` | `tform2rotm`

Objects

`se2` | `se3` | `so2` | `quaternion`

so2

SO(2) rotation

Description

The `so2` object represents an SO(2) rotation in 2-D.

For more information, see the “2-D Orthonormal Rotation Matrix” on page 1-399 section.

This object acts like a numerical matrix, enabling you to compose rotations using multiplication and division.

Creation

Syntax

```
rotation = so2
rotation = so2(rotation)
rotation = so2(transformation)
rotation = so2(angle,"theta")
```

Description

`rotation = so2` creates an SO(2) rotation representing an identity rotation with no translation.

$$rotation = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

`rotation = so2(rotation)` creates an SO(2) rotation `rotation` representing a pure rotation defined by the orthonormal rotation `rotation`.

$$rotation = \begin{bmatrix} r_{11} & r_{12} \\ r_{21} & r_{22} \end{bmatrix}$$

`rotation = so2(transformation)` creates an SO(2) rotation from the SE(2) transformation `transformation`.

`rotation = so2(angle,"theta")` creates an SO(2) rotation `rotation` from a rotation angle about the *z*-axis `angle`.

Note If any inputs contain more than one rotation, the output `rotation` is an *N*-element array of `so2` objects corresponding to each of the *N* input rotations.

Input Arguments

`rotation` — Orthonormal rotation
2-by-2 matrix | 2-by-2-by-*N* matrix | `so2` object | *N*-element array of `so2` objects

Orthonormal rotation, specified as a 2-by-2 matrix, a 3-by-3-by-*N* array, a scalar `so2` object, or an *N*-element array of `so2` objects. *N* is the total number of rotations.

If `rotation` is an array, the resulting number of created `so2` objects in the output array is equal to *N*.

Example: `eye(3)`

Data Types: `single` | `double`

`transformation` — Homogeneous transformation
`se2` object | *N*-element array of `se2` objects

Homogeneous transformation, specified as an `se2` object or an *N*-element array of `se2` objects. *N* is the total number of transformations specified.

The output `so2` object contains only the rotational submatrix of the `se2` object.

If `transformation` is an array, the resulting number of created `so2` objects in the output array is equal to *N*.

Example: `se2([1 2],"trvec")`

`angle` — *z*-axis rotation angle
N-by-*M* matrix

z-axis rotation angle, specified as an *N*-by-*M* matrix. Each element of the matrix is an angle, in radians, about the *z*-axis. The `so2` object creates an `so2` object for each angle.

If `angle` is an *N*-by-*M* matrix, the resulting number of created `so2` objects is equal to *N*.

The rotation angle is counterclockwise positive when you look along the specified axis toward the origin.

Data Types: `single` | `double`

Object Functions

Mathematical Operations

mtimes, *	Transformation or rotation multiplication
mrdivide, /	Transformation or rotation right division
rdivide, ./	Element-wise transformation or rotation right division
times, .*	Element-wise transformation or rotation multiplication

Utilities

interp	Interpolate between transformations
dist	Calculate distance between transformations
normalize	Normalize transformation or rotation matrix
transform	Apply rigid body transformation to points

Numerical Conversions

rotm	Extract rotation matrix
trvec	Extract translation vector
tform	Extract homogeneous transformation
theta	Convert transformation or rotation to 2-D rotation angle
xytheta	Convert transformation or rotation to compact 2-D pose representation

Object Conversions

so3	SO(3) rotation

Examples

Create SO(2) Rotation Using Angle

Define an angle rotation of `pi/4` and a *xy* translation of `[6 4]`.

```
angle = pi/4;
```

Create an SO(2) rotation using the angle.

```
R = so2(angle,"theta")
```

```
R = so2
    0.7071   -0.7071
    0.7071    0.7071
```

Algorithms

2-D Orthonormal Rotation Matrix

SO(2) rotation matrices are 2-by-2 orthonormal matrices that represent a rotation about a single axis 2-D Euclidean space. SO(2) rotations have many special properties. For example, SO(2) rotation matrices are in the 2-D special orthogonal group, so the product of two SO(2) rotation matrices is an SO(2) rotation matrix. This enables you to compose rotations from multiple rotations.

This is a 2-D orthonormal rotation matrix that describes describe a rotation θ about the *z*-axis:

$$R_z(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}$$

These are other properties of SO(2) rotations:

- Each column is both orthogonal and a unit vector, meaning that none of the columns are multiples of each other.
- The determinant of the matrix is positive 1.
- The inverse of the matrix is the same as the transpose of the matrix: $R^{-1} = R^T$.

Version History

Introduced in R2022b

R2023a: New methods and syntaxes

`so2` supports new methods and syntaxes for converting to and from other transformations and rotations.

The new methods are:

- `so3`
- `theta`
- `tform`
- `trvec`
- `xytheta`

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

Functions
`axang2rotm` | `eul2rotm` | `quat2rotm` | `tform2rotm`

Objects
`se2` | `se3` | `so3` | `quaternion`

stateEstimatorPF

Create particle filter state estimator

Description

The `stateEstimatorPF` object 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 and correction. The prediction step uses the previous state to predict the current state based on a given system model. The correction step uses the current sensor measurement to correct the state estimate. The algorithm periodically redistributes, or resamples, the particles in the state space to match the posterior distribution of the estimated state.

The estimated state consists of state variables. Each particle represents a discrete state hypothesis of these state variables. 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.

For more information on the particle filter workflow and setting specific parameters, see:

- “Particle Filter Workflow”
- “Particle Filter Parameters”

Creation

Syntax

`pf = stateEstimatorPF`

Description

`pf = stateEstimatorPF` creates an object that enables the state estimation for a simple system with three state variables. Use the `initialize` method to initialize the particles with a known mean and covariance or uniformly distributed particles within defined bounds. To customize the particle filter’s system and measurement models, modify the `StateTransitionFcn` and `MeasurementLikelihoodFcn` properties.

After you create the object, use `initialize` to initialize the `NumStateVariables` and `NumParticles` properties. The `initialize` function sets these two properties based on your inputs.

Properties

`NumStateVariables` — Number of state variables
3 (default) | scalar

This property is read-only.

Number of state variables, specified as a scalar. This property is set based on the inputs to the `initialize` method. The number of states is implicit based on the specified matrices for initial state and covariance.

`NumParticles` — Number of particles used in the filter
1000 (default) | scalar

This property is read-only.

Number of particles using in the filter, specified as a scalar. You can specify this property only by calling the `initialize` method.

`StateTransitionFcn` — Callback function for determining the state transition between particle filter steps
function handle

Callback function for determining the state transition between particle filter steps, specified as a function handle. The state transition function evolves the system state for each particle. The function signature is:

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

The callback function accepts at least two input arguments: the `stateEstimatorPF` object, `pf`, and the particles at the previous time step, `prevParticles`. These specified particles are the `predictParticles` returned from the previous call of the object. `predictParticles` and `prevParticles` are the same size: `NumParticles`-by-`NumStateVariables`.

You can also use `varargin` to pass in a variable number of arguments from the `predict` function. When you call:

```
predict(pf,arg1,arg2)
```

MATLAB essentially calls `stateTranstionFcn` as:

```
stateTransitionFcn(pf,prevParticles,arg1,arg2)
```

`MeasurementLikelihoodFcn` — Callback function calculating the likelihood of sensor measurements
function handle

Callback function calculating the likelihood of sensor measurements, specified as a function handle. Once a sensor measurement is available, this callback function calculates the likelihood that the measurement is consistent with the state hypothesis of each particle. You must implement this function based on your measurement model. The function signature is:

```
function likelihood = measurementLikelihoodFcn(PF,predictParticles,measurement,varargin)
```

The callback function accepts at least three input arguments:

1. `pf` – The associated `stateEstimatorPF` object
2. `predictParticles` – The particles that represent the predicted system state at the current time step as an array of size `NumParticles`-by-`NumStateVariables`
3. `measurement` – The state measurement at the current time step

You can also use `varargin` to pass in a variable number of arguments. These arguments are passed by the `correct` function. When you call:

```
correct(pf,measurement,arg1,arg2)
```

MATLAB essentially calls `measurementLikelihoodFcn` as:

```
measurementLikelihoodFcn(pf,predictParticles,measurement,arg1,arg2)
```

The callback needs to return exactly one output, `likelihood`, which is the likelihood of the given `measurement` for each particle state hypothesis.

`IsStateVariableCircular` — Indicator if state variables have a circular distribution
[0 0 0] (default) | logical array

Indicator if state variables have a circular distribution, specified as a logical array. Circular (or angular) distributions use a probability density function with a range of `[-pi,pi]`. If the object has multiple state variables, then `IsStateVariableCircular` is a row vector. Each vector element indicates if the associated state variable is circular. If the object has only one state variable, then `IsStateVariableCircular` is a scalar.

`ResamplingPolicy` — Policy settings that determine when to trigger resampling
object

Policy settings that determine when to trigger resampling, specified as an object. You can trigger resampling either at fixed intervals, or you can trigger it dynamically, based on the number of effective particles. See `resamplingPolicyPF` for more information.

`ResamplingMethod` — Method used for particle resampling
`'multinomial'` (default) | `'residual'` | `'stratified'` | `'systematic'`

Method used for particle resampling, specified as `'multinomial'`, `'residual'`, `'stratified'`, and `'systematic'`.

`StateEstimationMethod` — Method used for state estimation
`'mean'` (default) | `'maxweight'`

Method used for state estimation, specified as `'mean'` and `'maxweight'`.

`Particles` — Array of particle values
`NumParticles`-by-`NumStateVariables` matrix

Array of particle values, specified as a `NumParticles`-by-`NumStateVariables` matrix. Each row corresponds to the state hypothesis of a single particle.

`Weights` — Particle weights
`NumParticles`-by-1 vector

Particle weights, specified as a `NumParticles`-by-1 vector. Each weight is associated with the particle in the same row in the `Particles` property.

`State` — Best state estimate
vector

This property is read-only.

Best state estimate, returned as a vector with length `NumStateVariables`. The estimate is extracted based on the `StateEstimationMethod` property.

`State Covariance` — Corrected system covariance
N-by-*N* matrix | `[]`

This property is read-only.

Corrected system variance, returned as an *N*-by-*N* matrix, where *N* is equal to the `NumStateVariables` property. The corrected state is calculated based on the `StateEstimationMethod` property and the `MeasurementLikelihoodFcn`. If you specify a state estimate method that does not support covariance, then the property is set to `[]`.

Object Functions

initialize	Initialize the state of the particle filter
getStateEstimate	Extract best state estimate and covariance from particles
predict	Predict state of robot in next time step
correct	Adjust state estimate based on sensor measurement

Examples

Particle Filter Prediction and Correction

Create a `stateEstimatorPF` object, and execute a prediction and correction step for state estimation. The particle filter gives a predicted state estimate based on the return value of `StateTransitionFcn`. It then corrects the state based on a given measurement and the return value of `MeasurementLikelihoodFcn`.

Create a particle filter with the default three states.

```
pf = stateEstimatorPF
```

```
pf = 
  stateEstimatorPF with properties:

           NumStateVariables: 3
                NumParticles: 1000
          StateTransitionFcn: @nav.algs.gaussianMotion
    MeasurementLikelihoodFcn: @nav.algs.fullStateMeasurement
     IsStateVariableCircular: [0 0 0]
            ResamplingPolicy: [1x1 resamplingPolicyPF]
            ResamplingMethod: 'multinomial'
       StateEstimationMethod: 'mean'
            StateOrientation: 'row'
                   Particles: [1000x3 double]
                     Weights: [1000x1 double]
                       State: 'Use the getStateEstimate function to see the value.'
             StateCovariance: 'Use the getStateEstimate function to see the value.'
```

Specify the mean state estimation method and systematic resampling method.

```
pf.StateEstimationMethod = 'mean';
pf.ResamplingMethod = 'systematic';
```

Initialize the particle filter at state [4 1 9] with unit covariance (`eye(3)`). Use 5000 particles.

```
initialize(pf,5000,[4 1 9],eye(3));
```

Assuming a measurement [4.2 0.9 9], run one predict and one correct step.

```
[statePredicted,stateCov] = predict(pf);
[stateCorrected,stateCov] = correct(pf,[4.2 0.9 9]);
```

Get the best state estimate based on the `StateEstimationMethod` algorithm.

```
stateEst = getStateEstimate(pf)
```

```
stateEst = 1×3

    4.1562    0.9185    9.0202
```

Estimate Robot Position in a Loop Using Particle Filter

Use the `stateEstimatorPF` object to track a robot as it moves in a 2-D space. The measured position has random noise added. Using `predict` and `correct`, track the robot based on the measurement and on an assumed motion model.

Initialize the particle filter and specify the default state transition function, the measurement likelihood function, and the resampling policy.

```
pf = stateEstimatorPF;
pf.StateEstimationMethod = 'mean';
pf.ResamplingMethod = 'systematic';
```

Sample 1000 particles with an initial position of [0 0] and unit covariance.

```
initialize(pf,1000,[0 0],eye(2));
```

Prior to estimation, define a sine wave path for the dot to follow. Create an array to store the predicted and estimated position. Define the amplitude of noise.

```
t = 0:0.1:4*pi;
dot = [t; sin(t)]';
robotPred = zeros(length(t),2);
robotCorrected = zeros(length(t),2);
noise = 0.1;
```

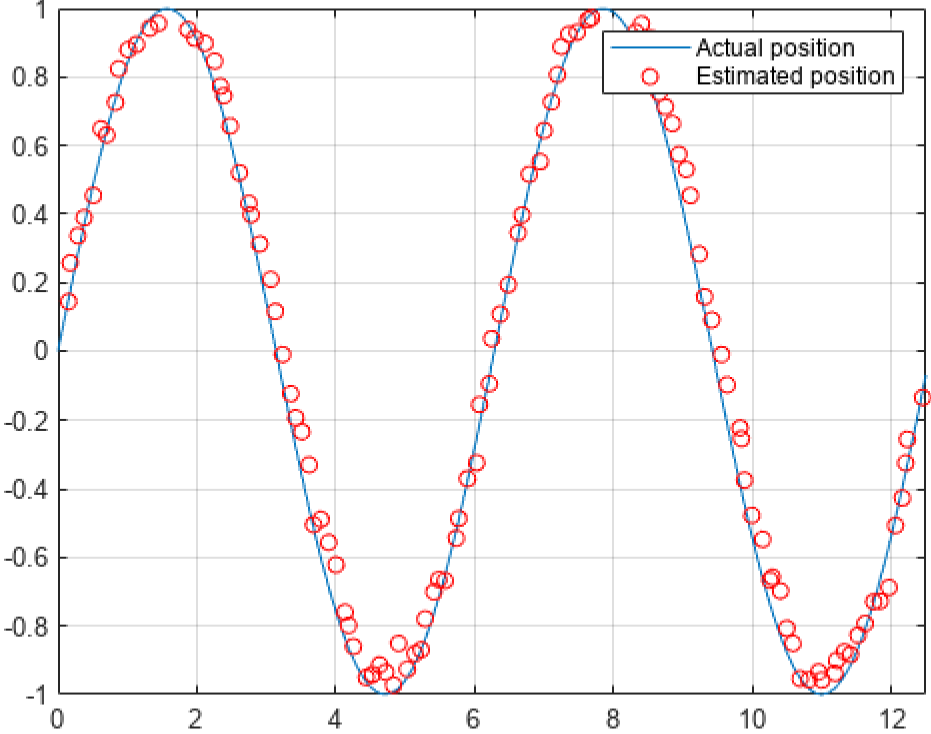
Begin the loop for predicting and correcting the estimated position based on measurements. The resampling of particles occurs based on the`ResamplingPolicy` property. The robot moves based on a sine wave function with random noise added to the measurement.

```
for i = 1:length(t)
    % Predict next position. Resample particles if necessary.
    [robotPred(i,:),robotCov] = predict(pf);
    % Generate dot measurement with random noise. This is
    % equivalent to the observation step.
    measurement(i,:) = dot(i,:) + noise*(rand([1 2])-noise/2);
    % Correct position based on the given measurement to get best estimation.
```

```
    % Actual dot position is not used. Store corrected position in data array.
    [robotCorrected(i,:),robotCov] = correct(pf,measurement(i,:));
end
```

Plot the actual path versus the estimated position. Actual results may vary due to the randomness of particle distributions.

```
plot(dot(:,1),dot(:,2),robotCorrected(:,1),robotCorrected(:,2),'or')
xlim([0 t(end)])
ylim([-1 1])
legend('Actual position','Estimated position')
grid on
```



The figure shows how close the estimate state matches the actual position of the robot. Try tuning the number of particles or specifying a different initial position and covariance to see how it affects tracking over time.

Version History

Introduced in R2016a

R2019b: `stateEstimatorPF` was renamed

Behavior change in future release

The `stateEstimatorPF` object was renamed from `robotics.ParticleFilter`. Use `stateEstimatorPF` for all object creation.

References

[1] Arulampalam, M.S., S. Maskell, N. Gordon, and T. Clapp. "A Tutorial on Particle Filters for Online Nonlinear/Non-Gaussian Bayesian Tracking." *IEEE Transactions on Signal Processing.* Vol. 50, No. 2, Feb 2002, pp. 174-188.

[2] Chen, Z. "Bayesian Filtering: From Kalman Filters to Particle Filters, and Beyond." *Statistics*. Vol. 182, No. 1, 2003, pp. 1-69.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

`resamplingPolicyPF` | `initialize` | `getStateEstimate` | `predict` | `correct`

Topics
"Track a Car-Like Robot Using Particle Filter"
"Particle Filter Parameters"
"Particle Filter Workflow"

taskSpaceMotionModel

Model rigid body tree motion given task-space reference inputs

Description

The `taskSpaceMotionModel` object models the closed-loop task-space motion of a manipulator, specified as a rigid body tree object. The motion model behavior is defined by the MotionType property.

For more details about the equations of motion, see "Task-Space Motion Model".

Creation

Syntax

```
motionModel = taskSpaceMotionModel
```

```
motionModel = taskSpaceMotionModel("RigidBodyTree",tree)
```

```
motionModel = taskSpaceMotionControlModel(Name,Value)
```

Description

`motionModel = taskSpaceMotionModel` creates a motion model for a default two-joint manipulator.

`motionModel = taskSpaceMotionModel("RigidBodyTree",tree)` creates a motion model for the specified `rigidBodyTree` object.

`motionModel = taskSpaceMotionControlModel(Name,Value)` sets additional properties specified as name-value pairs. You can specify multiple properties in any order.

Properties

`RigidBodyTree` — Rigid body tree robot model
`rigidBodyTree` object

Rigid body tree robot model, specified as a `rigidBodyTree` object that defines the inertial and kinematic properties of the manipulator.

`EndEffectorName` — End effector body
`'tool'` (default) | string scalar | character vector

This property defines the body that will be used as the end effector, and for which the task space motion is defined. The property must correspond to a body name in the `rigidBodyTree` object of the RigidBodyTree property. If the rigid body tree is updated without also updating the end effector, the body with the highest index becomes the end-effector body by default.

Kp — Proportional gain for PD Control
`500*eye(6)` (default) | 6-by-6 matrix

Proportional gain for PD control, specified as a 6-by-6 matrix.

Kd — Derivative gain for PD control
`100*eye(6)` (default) | 6-by-6 matrix

Derivative gain for proportional-derivative (PD) control, specified as a 6-by-6 matrix.

`JointDamping` — Joint damping constant
`ones(1,n)` (default) | *n*-element vector

Joint damping constant, specified as an *n*-element vector, where *n* is the number of non-fixed joints in the robot model specified by the Rigid Body Tree property. Joint damping units are N/(m/s) or N/(rad/s) for prismatic and revolute joints, respectively.

`MotionType` — Type of motion computed by the motion model
`"PDControl"` (default)

Type of motion, specified as `"PDControl"`, which uses proportional-derivative (PD) control mapped to the joints via a Jacobian-Transpose controller. The control is based on the specified Kp and Kd properties.

Object Functions

derivative	Time derivative of manipulator model states
updateErrorDynamicsFromStep	Update values of NaturalFrequency and DampingRatio properties given desired step response

Examples

Create Task-Space Motion Model

This example shows how to create and use a `taskSpaceMotionModel` object for a manipulator robot arm in task-space.

Create the Robot

```
robot = loadrobot("kinovaGen3","DataFormat","column","Gravity",[0 0 -9.81]);
```

Set Up the Simulation

Set the time span to be 1 second with a timestep size of 0.02 seconds. Set the initial state to the home configuration of the robot, with a velocity of zero.

```
tspan = 0:0.02:1;
initialState = [homeConfiguration(robot);zeros(7,1)];
```

Define a reference state with a target position and zero velocity.

```
refPose = trvec2tform([0.6 -.1 0.5]);
refVel = zeros(6,1);
```

Create the Motion Model

Model the behavior as a system under proportional-derivative (PD) control.

```
motionModel = taskSpaceMotionModel("RigidBodyTree",robot,"EndEffectorName","EndEffector_Link");
```

Simulate the Robot

Simulate the behavior over 1 second using a stiff solver to more efficiently capture the robot dynamics. Using `ode15s` enables higher precision around the areas with a high rate of change.

```
[t,robotState] = ode15s(@(t,state)derivative(motionModel,state,refPose,refVel),tspan,initialState
```

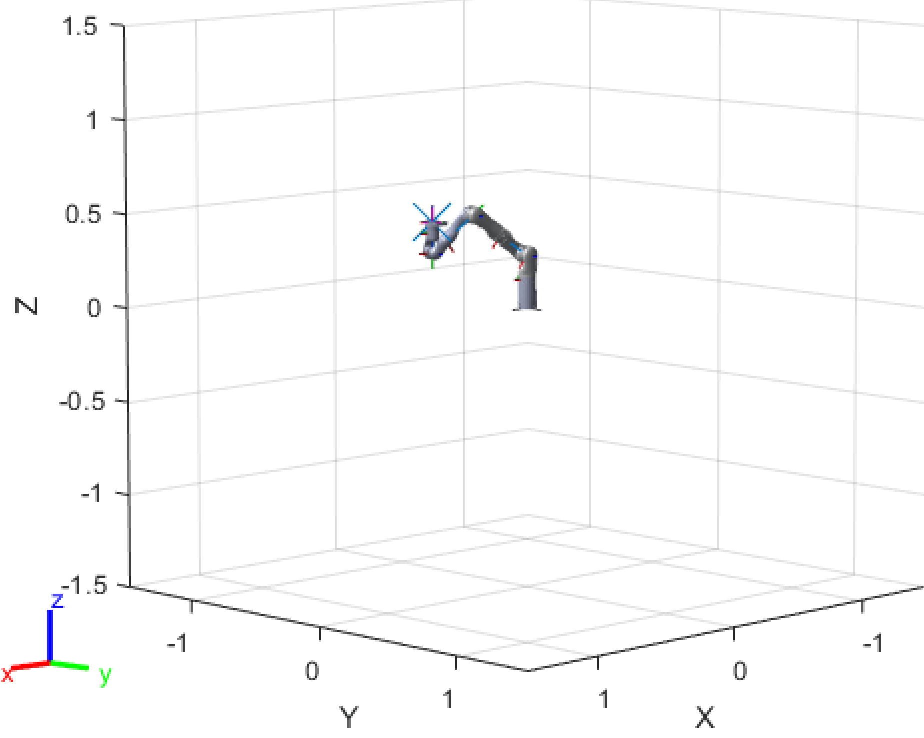
Plot the Response

Plot the robot's initial position and mark the target with an X.

```
figure
show(robot,initialState(1:7));
hold all
plot3(refPose(1,4),refPose(2,4),refPose(3,4),"x","MarkerSize",20)
```

Observe the response by plotting the robot in a 5 Hz loop.

```
r = rateControl(5);
for i = 1:size(robotState,1)
    show(robot,robotState(i,1:7)',"PreservePlot",false);
    waitfor(r);
end
```



Version History

Introduced in R2019b

References

[1] Craig, John J. *Introduction to Robotics: Mechanics and Control*. Upper Saddle River, NJ: Pearson Education, 2005.

[2] Spong, Mark W., Seth Hutchinson, and Mathukumalli Vidyasagar. *Robot Modeling and Control*. Hoboken, NJ: Wiley, 2006.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

Classes
`jointSpaceMotionModel`

Blocks
Task Space Motion Model

Functions
`derivative`

Topics
“Plan and Execute Task- and Joint-Space Trajectories Using KINOVA Gen3 Manipulator”

transformTree

Define coordinate frames and relative transformations

Description

The `transformTree` object contains an organized tree structure for coordinate frames and their relative transformations over time. The object stores the relative transformations between children frames and their parents. You can specify a timestamped transform for frames and query the relative transformations between different frames in the tree. The object interpolates intermediate timestamps using a constant velocity assumption for linear motion, and spherical linear interpolation (SLERP) for angular motion. Otherwise, the relative transformations are kept constant past the range of the timestamps specified. Times prior to the first timestamp return `NaN`.

Use the `updateTransform` function to add timestamps to the tree by defining the parent-to-child relationships. Query specific transformations at given timestamps using `getTransform` and display the frame relationships using `show`.

Creation

Syntax

```
frames = transformTree
frames = transformTree(baseName)
frames = transformTree(baseName,numFrames)
frames = transformTree(baseName,numFrames,numTransforms)
frames = transformTree(baseName,numFrames,numTransforms,rootTime)
```

Description

`frames = transformTree` creates a transformation tree data structure with a single frame, `"root"`, with the maximum number of frames and timestamped transforms per frame, set to 10.

`frames = transformTree(baseName)` specifies the name of the root frame as a string or character vector.

`frames = transformTree(baseName,numFrames)` additionally sets the MaxNumFrames property, which defines the max number of named frames in the object.

`frames = transformTree(baseName,numFrames,numTransforms)` additionally sets the MaxNumTransforms property, which defines the max number of timestamped transforms per frame name.

`frames = transformTree(baseName,numFrames,numTransforms,rootTime)` additionally specifies the timestamp of the initial `baseName` frame as a scalar time in seconds.

Properties

`MaxNumFrames` — Maximum number of frames in tree
10 (default) | positive integer

Maximum number of frames in the tree, specified as a positive integer. Each frame has associated timestamped transforms that define the state of the frame at those specific times.

Data Types: `double`

`MaxNumTransforms` — Maximum number of timestamped transforms per frame
10 (default) | positive integer

Maximum number of timestamped transforms per frame, specified as a positive integer. This property sets an upper limit on the number of timestamped transforms the object can store for each frame named in the structure. A `transformTree` object with MaxNumFrames and `MaxNumTransforms` set to 10 can store a maximum of 100 transformations with 10 for each frame.

Data Types: `double`

`NumFrames` — Current number of coordinate frames stored
1 (default) | positive integer

Current number of coordinate frames stored, specified as a positive integer. The object starts with a single root frame, and new frames and specific timestamps are added using `updateTransform` function.

Data Types: `double`

Object Functions

getGraph	Graph object representing tree structure
getTransform	Get relative transform between frames
info	List all frame names and stored timestamps
removeTransform	Remove frame transform relative to its parent
show	Show transform tree
updateTransform	Update frame transform relative to its parent

Version History

Introduced in R2022a

See Also

Objects
`robotScenario`

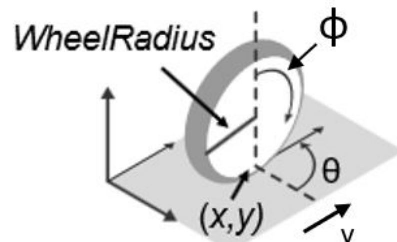
Functions
`getGraph` | `getTransform` | `info` | `removeTransform` | `show` | `updateTransform`

unicycleKinematics

Unicycle vehicle model

Description

`unicycleKinematics` creates a unicycle vehicle model to simulate simplified car-like vehicle dynamics. The state of the vehicle is defined as a three-element vector, *[x y theta]*, with a global *xy*-position, specified in meters, and a vehicle heading angle, *theta*, specified in radians. This model approximates a unicycle vehicle with a given wheel radius, WheelRadius, that can spin in place according to a heading angle, *theta*. To compute the time derivative states for the model, use the `derivative` function with input commands and the current robot state.



Creation

Syntax

`kinematicModel = unicycleKinematics`

`kinematicModel = unicycleKinematics(Name,Value)`

Description

`kinematicModel = unicycleKinematics` creates a unicycle kinematic model object with default property values.

`kinematicModel = unicycleKinematics(Name,Value)` sets additional properties to the specified values. You can specify multiple properties in any order.

Properties

`WheelRadius` — Wheel radius of vehicle
`0.1` (default) | positive numeric scalar

The wheel radius of the vehicle, specified in meters.

`WheelSpeedRange` — Range of vehicle wheel speeds
`[-Inf Inf]` (default) | two-element vector

The vehicle speed range is a two-element vector that provides the minimum and maximum vehicle speeds, [*MinSpeed MaxSpeed*], specified in meters per second.

`VehicleInputs` — Type of motion inputs for vehicle
`"WheelSpeedHeadingRate"` (default) | character vector | string scalar

The `VehicleInputs` property specifies the format of the model input commands when using the `derivative` function. Options are specified as one of the following strings:

- `"WheelSpeedHeadingRate"` — Wheel speed and heading angular velocity, specified in radians per second.
- `"VehicleSpeedHeadingRate"` — Vehicle speed and heading angular velocity, specified in radians per second.

Object Functions

derivative   Time derivative of vehicle state

Examples

Plot Path of Unicycle Kinematic Robot

Create a Robot

Define a robot and set the initial starting position and orientation.

```
kinematicModel = unicycleKinematics;
initialState = [0 0 0];
```

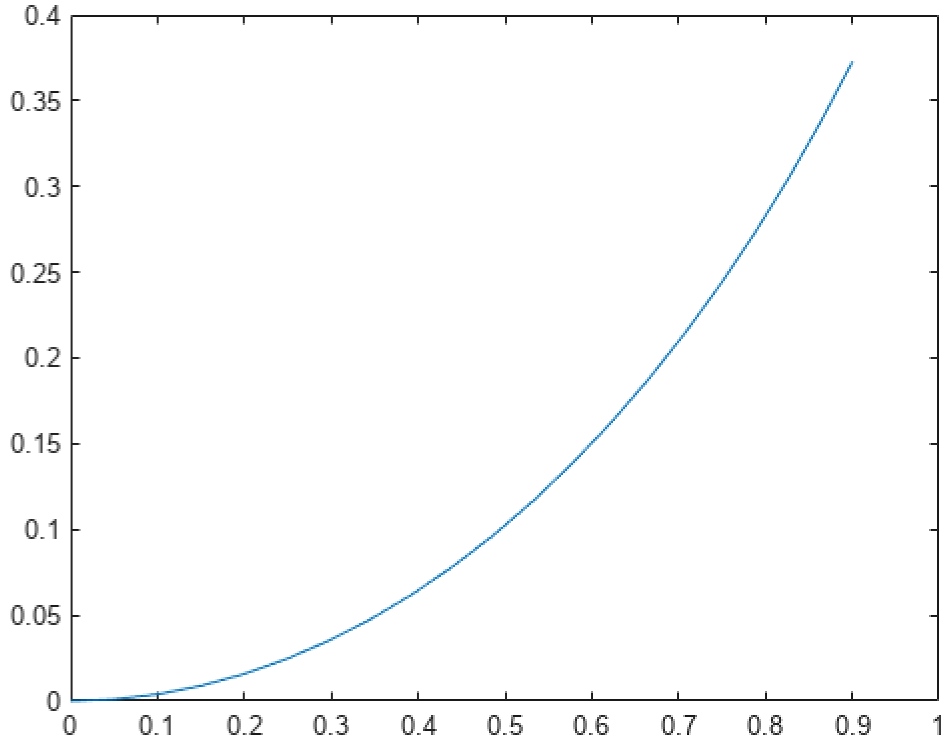
Simulate Robot Motion

Set the timespan of the simulation to `1` s with `0.05` s time steps and the input commands to `10` m/s and a heading angular velocity of `pi/4` rad/s to do a left turn. Simulate the motion of the robot by using the `ode45` solver on the `derivative` function.

```
tspan = 0:0.05:1;
inputs = [10 pi/4]; %Constant speed and turning left
[t,y] = ode45(@(t,y)derivative(kinematicModel,y,inputs),tspan,initialState);
```

Plot path

```
figure
plot(y(:,1),y(:,2))
```



Version History

Introduced in R2019b

References

[1] Lynch, Kevin M., and Frank C. Park. *Modern Robotics: Mechanics, Planning, and Control* 1st ed. Cambridge, MA: Cambridge University Press, 2017.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

Classes
`bicycleKinematics` | `ackermannKinematics` | `differentialDriveKinematics`

Blocks
Unicycle Kinematic Model

Functions
`derivative`

Topics
“Simulate Different Kinematic Models for Mobile Robots”
“Mobile Robot Kinematics Equations”

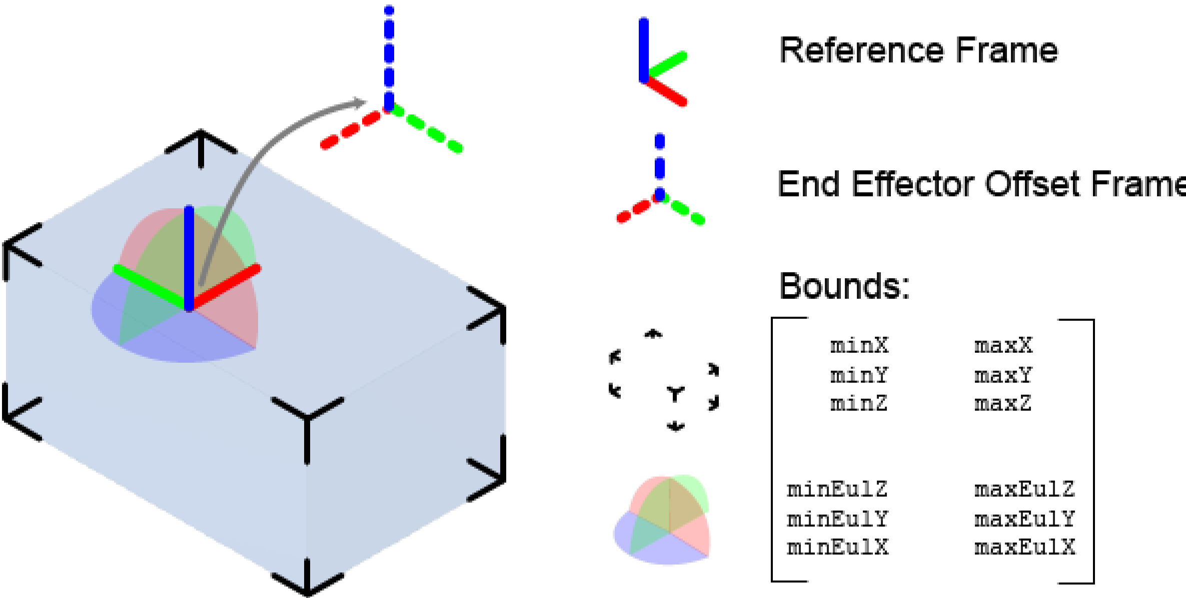
workspaceGoalRegion

Define workspace region of end-effector goal poses

Description

The `workspaceGoalRegion` object defines a region for valid end-effector goal poses. To sample poses within the bounds of the goal region, use the `sample` object function. You can also visualize the bounds you define using the `show` object function.

The object is typically used with rapidly exploring random tree (RRT) planners like the `manipulatorRRT` object. The `sample` generates alternative goal states to increase the likelihood of finding valid paths.



The key elements of the goal region are defined as object properties:

- ReferencePose — Pose of the reference frame in the world frame. The bounds and offset pose are relative to this frame.
- EndEffectorOffsetPose — Offset pose applied to any pose sampled in the reference frame. Use this offset if the end effector needs to be positioned differently based on grasping or other geometric restrictions.
- Bounds — Bounds of the region as a 6-by-2 matrix with the minimum and maximum values for the *XYZ*-position and *ZYX* Euler angle orientation, in respective column vectors.

Creation

Syntax

```
goalRegion = workspaceGoalRegion(EndEffectorName)
goalRegion = workspaceGoalRegion(EndEffectorName,Name,Value)
```

Description

`goalRegion = workspaceGoalRegion(EndEffectorName)` creates a default workspace goal region object for the specified end-effector name. Sets the `EndEffectorName` property.

`goalRegion = workspaceGoalRegion(EndEffectorName,Name,Value)` sets additional properties on page 1-420 on the object using name-value pairs. For example, `workSpaceGoalRegion("endEffector","Bounds",limits)` creates a workspace goal region with the Bounds property specified as a matrix.

Properties

`EndEffectorName` — Name of end effector
string scalar

Name of the end effector, specified as a string scalar.

Example: `"eeTool"`

Data Types: `string`

`ReferencePose` — Pose of reference frame
`eye(4)` (default) | 4-by-4 homogeneous transform

Pose of the reference frame, specified as a 4-by-4 homogeneous transformation matrix. The `Bounds` property defines the limits of the goal region relative to this reference frame.

Example: `trvect2tform([1 2 3])`

Data Types: `double`

`EndEffectorOffsetPose` — End-effector offset pose applied to poses sampled in reference frame
`eye(4)` (default) | 4-by-4 homogeneous transform

End-effector offset pose applied to poses sampled in the reference frame, specified as a 4-by-4 homogeneous transformation matrix. This offset is applied to all poses sampled. Use this offset if the end effector needs to be positioned differently based on grasping or other geometric restrictions.

Example: `trvect2tform([0.5 1 0])`

Example: `eul2tform([pi/2 0 -pi/4])`

Data Types: `double`

`Bounds` — Position and orientation bounds
`zeros(6,2)` (default) | 6-by-2 matrix

Position and orientation bounds on pose samples, specified as a 6-by-2 matrix with the minimum and maximum values in column vectors.

```
wgr.Bounds = [ minX  maxX;
               minY  maxY;
               minZ  maxZ;
               minEulZ  maxEulZ;
               minEulY  maxEulY;
               minEulX  maxEulX ];
```

The first three rows are the *XYZ*-position bounds. The last three rows are the orientation bounds, which are specified as intrinsic *ZYX* Euler angles. Orientation is based on the right-hand rule, with counterclockwise rotations about the respective axes being positive and measured in radians. During sampling, a pose is uniformly sampled within each of these bounds to obtain a sample pose in the reference frame.

Data Types: `double`

Object Functions

sample	Sample end-effector poses in world frame
show	Visualize workspace bounds, reference frame, and offset frame

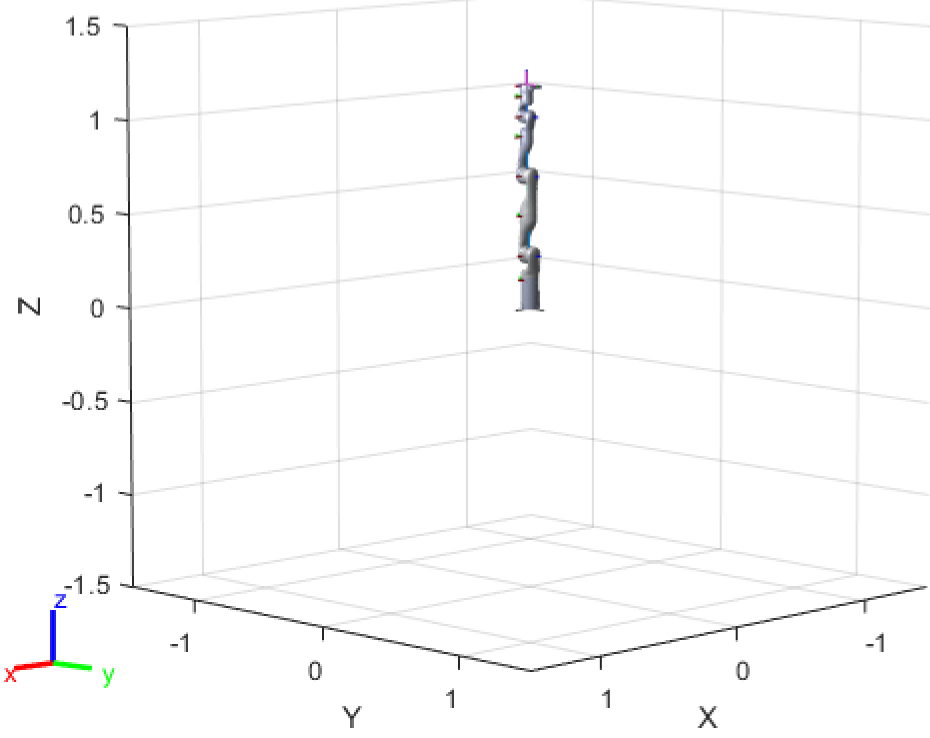
Examples

Plan Path To Workspace Goal Region

Specify a goal region in your workspace and plan a path within those bounds. The `workspaceGoalRegion` object defines the bounds on the *xyz*-position and *zyx* Euler orientation of the robot end effector. The `manipulatorRRT` object plans a path based on that goal region and samples random poses within the bounds.

Load an existing robot model as a `rigidBodyTree` object.

```
robot = loadrobot("kinovaGen3", "DataFormat", "row");
ax = show(robot);
```



Create Path Planner

Create a rapidly-exploring random tree (RRT) path planner for the robot. This example uses an empty environment, but this workflow also works well with cluttered environments. You can add collision objects to the environment like the `collisionBox` or `collisionMesh` object.

```
planner = manipulatorRRT(robot,{});
planner.SkippedSelfCollisions="parent";
```

Define Goal Region

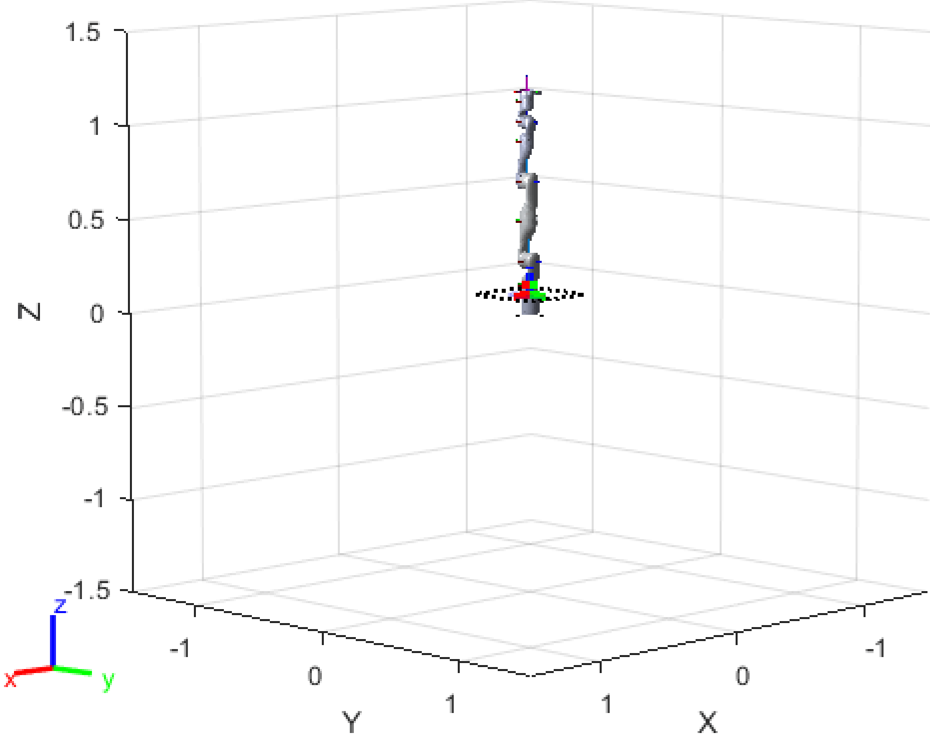
Create a workspace goal region using the end-effector body name of the robot.

Define the goal region parameters for your workspace. The goal region includes a reference pose, *xyz*-position bounds, and orientation limits on the *zyx* Euler angles. This example specifies bounds on the *xy*-plane in meters and allows rotation about the *z*-axis in radians.

```
goalRegion = workspaceGoalRegion(robot.BodyNames{end});
goalRegion.ReferencePose = trvec2tform([0.5 0.5 0.2]);
goalRegion.Bounds(1, :) = [-0.2 0.2];    % X Bounds
goalRegion.Bounds(2, :) = [-0.2 0.2];    % Y Bounds
goalRegion.Bounds(4, :) = [-pi/2 pi/2]; % Rotation about the Z-axis
```

You can also apply a fixed offset to all poses sampled within the region. This offset can account for grasping tools or variations in dimensions within your workspace. For this example, apply a fixed transformation that places the end effector 5 cm above the workspace.

```
goalRegion.EndEffectorOffsetPose = trvec2tform([0 0 0.05]);
hold on
show(goalRegion);
```



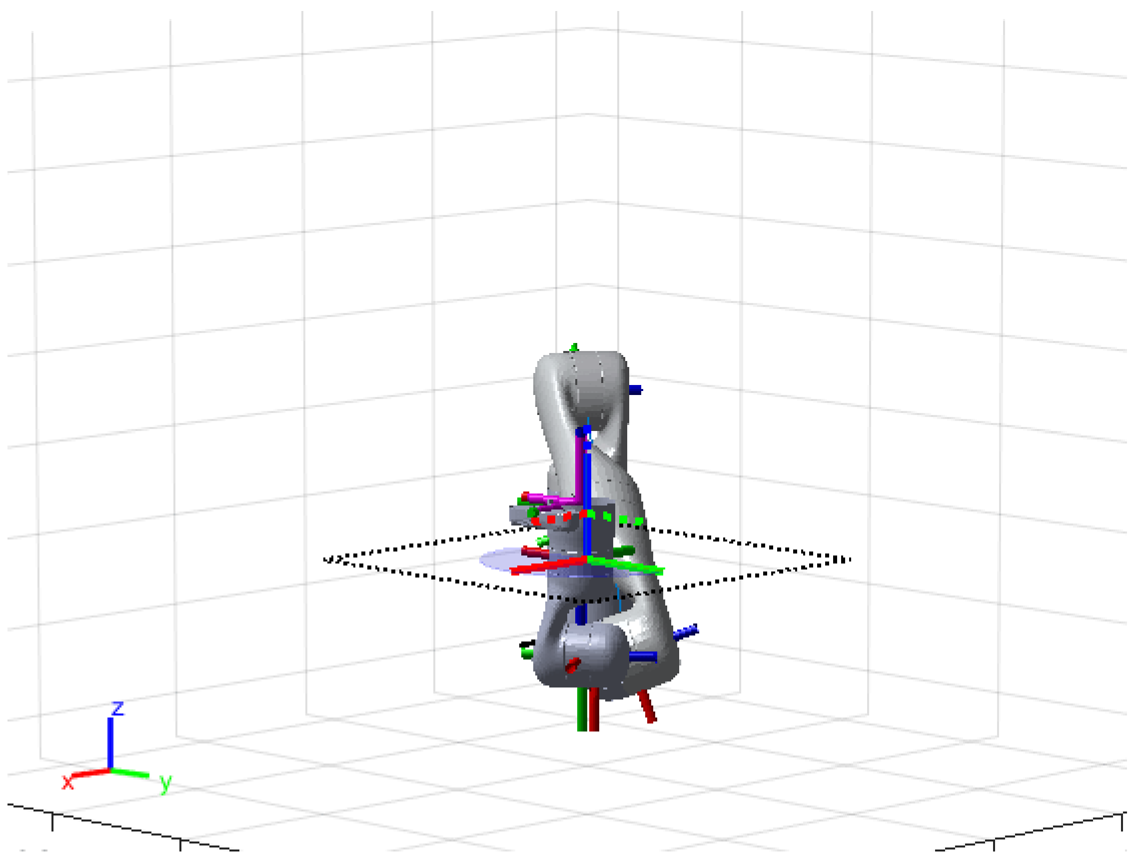
Plan Path To Goal Region

Plan a path to the goal region from the robot's home configuration. Due to the randomness in the RRT algorithm, this example sets the `rng` seed to ensure repeatable results.

```
rng(0)
path = plan(planner,homeConfiguration(robot),goalRegion);
```

Show the robot executing the path. To visualize a more realistic path, interpolate points between path configurations.

```
interpConfigurations = interpolate(planner,path,5);

for i = 1 : size(interpConfigurations)
    show(robot,interpConfigurations(i,:),"PreservePlot",false);
    set(ax,'ZLim',[-0.05 0.75],'YLim',[-0.05 1],'XLim',[-0.05 1],...
        'CameraViewAngle',5)

    drawnow
end
hold off
```



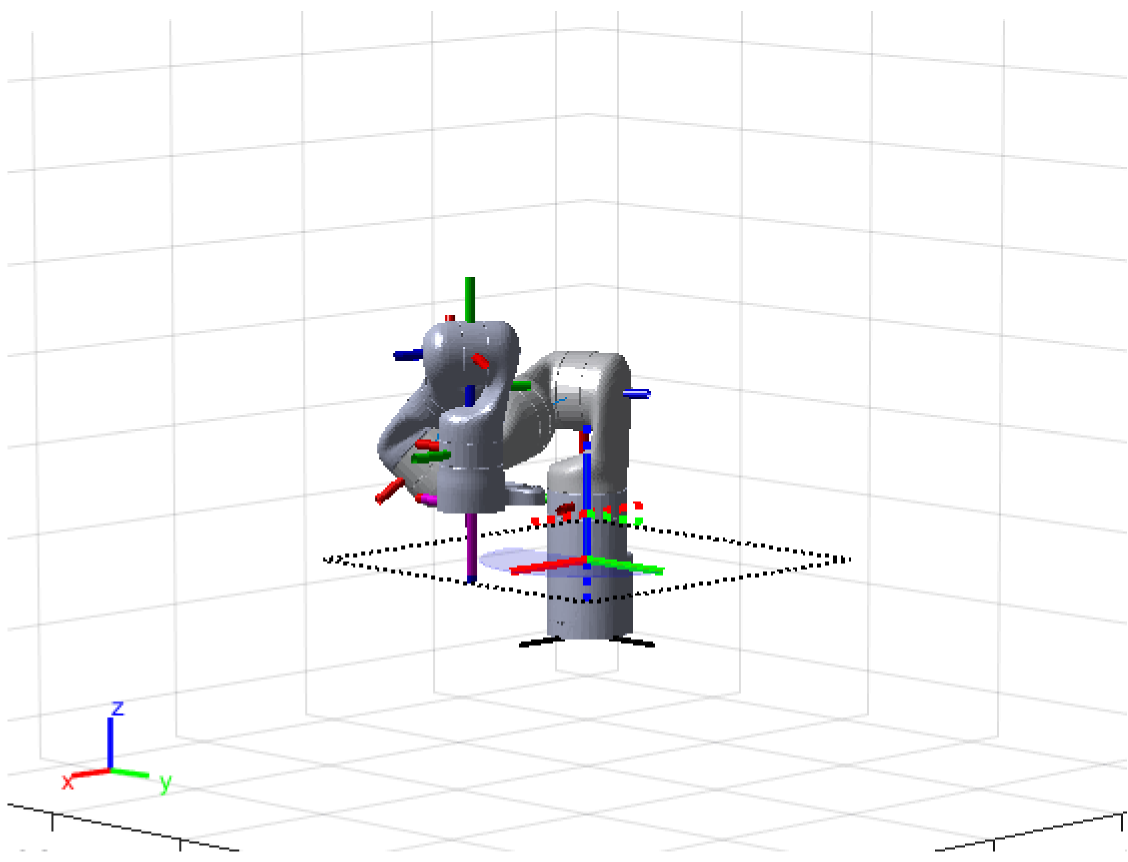
Adjust End-Effector Pose

Notice that the robot arm approaches the workspace from the bottom. To flip the orientation of the final position, add a `pi` rotation to the Y-axis for the reference pose.

```
goalRegion.EndEffectorOffsetPose = ...
    goalRegion.EndEffectorOffsetPose*eul2tform([0 pi 0],"ZYX");
```

Replan the path and visualize the robot motion again. The robot now approaches from the top.

```
hold on
show(goalRegion);
path = plan(planner,homeConfiguration(robot),goalRegion);

interpConfigurations = interpolate(planner,path,5);

for i = 1 : size(interpConfigurations)
    show(robot, interpConfigurations(i, :),"PreservePlot",false);
    set(ax,'ZLim',[-0.05 0.75],'YLim',[-0.05 1],'XLim',[-0.05 1])
    drawnow;
end
hold off
```



Version History

Introduced in R2021a

References

[1] Berenson, Dmitry, Siddhartha S. Srinivasa, Dave Ferguson, Alvaro Collet, and James J. Kuffner. "Manipulation Planning with Workspace Goal Regions." In *2009 IEEE International Conference on Robotics and Automation (ICRA)*, 618–24. Kobe, Japan: Institute of Electrical and Electronics Engineers, 2009. https://doi.org/10.1109/ROBOT.2009.5152401.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`manipulatorRRT` | `sample` | `show`

waypointTrajectory

Waypoint trajectory generator

Description

The waypointTrajectory System object generates trajectories based on specified waypoints. When you create the System object, you can choose to specify the time of arrival, velocity, or ground speed at each waypoint. You can optionally specify other properties such as orientation at each waypoint. See “Algorithms” on page 1-460 for more details.

To generate a trajectory from waypoints:

1. Create the waypointTrajectory object and set its properties.
2. Call the object as if it were a function.

To learn more about how System objects work, see What Are System Objects?.

Creation

Syntax

```
trajectory = waypointTrajectory
trajectory = waypointTrajectory(Waypoints,TimeOfArrival)
trajectory = waypointTrajectory(Waypoints,GroundSpeed=groundSpeed)
trajectory = waypointTrajectory(Waypoints,Velocities=velocities)
trajectory = waypointTrajectory( ___ ,Name=Value)
```

Description

trajectory = waypointTrajectory returns a System object, trajectory, that generates a trajectory based on default stationary waypoints.

trajectory = waypointTrajectory(Waypoints,TimeOfArrival) specifies the time of arrival at which the generated trajectory passes through each waypoint. See the TimeOfArrival property for more details.

Tip When you specify the TimeOfArrival argument, you must not specify these properties:

- JerkLimit
- InitialTime
- WaitTime

trajectory = waypointTrajectory(Waypoints,GroundSpeed=groundSpeed) specifies the ground speed at which the generated trajectory passes through at each waypoint. See the GroundSpeed property for more details.

`trajectory = waypointTrajectory(Waypoints,Velocities=velocities)` specifies the velocity at which the generated trajectory passes through at each waypoint. See the `Velocities` property for more details.

`trajectory = waypointTrajectory( ___ ,Name=Value)` sets each property by using name-value arguments. Unspecified properties have default or inferred values. You can use this syntax with any of the previous syntaxes.

Example: `trajectory = waypointTrajectory([10,10,0;20,20,0;20,20,10],[0,0.5,10])` creates a waypoint trajectory System object, `trajectory`, that starts at waypoint `[10,10,0]`, and then passes through `[20,20,0]` after 0.5 seconds and `[20,20,10]` after 10 seconds.

Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see System Design in MATLAB Using System Objects.

`SampleRate` — Sample rate of trajectory (Hz)
`100` (default) | positive scalar

Sample rate of trajectory in Hz, specified as a positive scalar.

Tunable: Yes

Data Types: `double`

`SamplesPerFrame` — Number of samples per output frame
`1` (default) | positive scalar integer

Number of samples per output frame, specified as a positive scalar integer.

Data Types: `double`

`Waypoints` — Positions in the navigation coordinate system (m)
N-by-3 matrix

Positions in the navigation coordinate system in meters, specified as an *N*-by-3 matrix. The columns of the matrix correspond to the first, second, and third axes, respectively. The rows of the matrix, *N*, correspond to individual waypoints.

Tip To let the trajectory wait at a specific waypoint, use one of the two options:

- If you specified the `TimeOfArrival` input argument, repeat the waypoint coordinate in two consecutive rows.
- If you did not specify the `TimeOfArrival` input argument, specify the wait time using the `WaitTime` property.

Data Types: `double`

`TimeOfArrival` — Time at each waypoint (s)
N-element column vector of nonnegative increasing numbers

Time corresponding to arrival at each waypoint in seconds, specified as an *N*-element column vector. The first element of `TimeOfArrival` must be `0`. The number of samples, *N*, must be the same as the number of samples (rows) defined by `Waypoints`.

Dependencies

To set this property, you must not specify these properties:

- `JerkLimit`
- `InitialTime`
- `WaitTime`

Data Types: `double`

`Velocities` — Velocity in navigation coordinate system at each waypoint (m/s)
N-by-3 matrix

Velocity in the navigation coordinate system at each waypoint in meters per second, specified as an *N*-by-3 matrix. The columns of the matrix correspond to the first, second, and third axes, respectively. The number of samples, *N*, must be the same as the number of samples (rows) defined by `Waypoints`.

If the velocity is specified as a non-zero value, the object automatically calculates the course of the trajectory based on the velocity. If the velocity is specified as zero, the object infers the course of the trajectory from adjacent waypoints.

Data Types: `double`

`Course` — Horizontal direction of travel (degree)
N-element real vector

Horizontal direction of travel, specified as an *N*-element real vector in degrees. The number of samples, *N*, must be the same as the number of samples (rows) defined by `Waypoints`. If neither `Velocities` nor `Course` is specified, course is inferred from the waypoints.

Dependencies

To set this property, the Velocities property must not be specified.

Data Types: `double`

`GroundSpeed` — Groundspeed at each waypoint (m/s)
N-element real vector

Groundspeed at each waypoint, specified as an *N*-element real vector in m/s. If the property is not specified, it is inferred from the waypoints. The number of samples, *N*, must be the same as the number of samples (rows) defined by `Waypoints`.

- To render forward motion, specify positive ground speed values.
- To render backward motion, specify negative ground speed values.
- To render reverse motion, separate positive and negative groundspeed values by a zero groundspeed value.

Dependencies

To set this property, the Velocities property must not be specified.

Data Types: `double`

`ClimbRate` — Climb rate at each waypoint (m/s)

N-element real vector

Climb Rate at each waypoint in meters per second, specified as an *N*-element real vector. The number of samples, *N*, must be the same as the number of samples (rows) defined by `Waypoints`. If neither `Velocities` nor `Course` is specified, climb rate is inferred from the waypoints.

Dependencies

To set this property, the Velocities property must not be specified.

Data Types: `double`

`JerkLimit` — Longitudinal jerk limit (m/s^3)

`Inf` (default) | positive scalar

Longitudinal jerk limit, specified as a positive scalar in m/s^3. Jerk is the time derivative of the acceleration. When you specify this property, the object produces a horizontal trapezoidal acceleration profile based on the jerk limit. If the `waypointTrajectory` object cannot achieve the specified `JerkLimit`, the object issues an error. You can set this property only during object creation.

Dependencies

To set this property, the TimeOfArrival property must not be specified.

Data Types: `double`

`InitialTime` — Time before trajectory starts (s)

`0` (default) | nonnegative scalar

Time before the trajectory starts, specified as a nonnegative scalar in seconds. The object reports quantities, such as position and velocity, as `NaN` before the trajectory starts. You can set this property only during object creation.

Dependencies

To set this property, the TimeOfArrival property must not be specified. Instead, you must specify either the `GroundSpeed` or `Velocities` property when creating the object.

Data Types: `double`

`WaitTime` — Wait time at each waypoint (s)

N-element vector of `0` (default) | *N*-element vector of nonnegative scalars

Wait time at each waypoint, specified as an *N*-element vector of nonnegative scalars. *N* must be the same as the number of samples (rows) defined by `Waypoints`. You can set this property only during object creation.

Dependencies

To set this property, the TimeOfArrival property must not be specified.

If you specified the TimeOfArrival property, then you cannot specify wait time through this property. Instead, specify wait time by repeating the waypoint coordinate in two consecutive rows in the Waypoints property.

Data Types: `double`

`Orientation` — Orientation at each waypoint

N-element quaternion column vector | 3-by-3-by-*N* array of real numbers

Orientation at each waypoint, specified as an *N*-element `quaternion` column vector or 3-by-3-by-*N* array of real numbers. Each `quaternion` must have a norm of `1`. Each 3-by-3 rotation matrix must be an orthonormal matrix. The number of quaternions or rotation matrices, *N*, must be the same as the number of samples (rows) defined by `Waypoints`.

If `Orientation` is specified by quaternions, the underlying class must be `double`.

Data Types: `double`

`AutoPitch` — Align pitch angle with direction of motion

`false` (default) | `true`

Align pitch angle with the direction of motion, specified as `true` or `false`. When specified as `true`, the pitch angle automatically aligns with the direction of motion. If specified as `false`, the pitch angle is set to zero (level orientation).

Dependencies

To set this property, the Orientation property must not be specified.

`AutoBank` — Align roll angle to counteract centripetal force

`false` (default) | `true`

Align roll angle to counteract the centripetal force, specified as `true` or `false`. When specified as `true`, the roll angle automatically counteracts the centripetal force. If specified as `false`, the roll angle is set to zero (flat orientation).

Dependencies

To set this property, the Orientation property must not be specified.

`ReferenceFrame` — Reference frame of trajectory

`'NED'` (default) | `'ENU'`

Reference frame of the trajectory, specified as `'NED'` (North-East-Down) or `'ENU'` (East-North-Up).

Usage

Syntax

```
[position,orientation,velocity,acceleration,angularVelocity] = trajectory()
```

Description

`[position,orientation,velocity,acceleration,angularVelocity] = trajectory()` outputs a frame of trajectory data based on specified creation arguments and properties.

Output Arguments

`position` — Position in local navigation coordinate system (m)

M-by-3 matrix

Position in the local navigation coordinate system in meters, returned as an *M*-by-3 matrix.

M is specified by the SamplesPerFrame property.

Data Types: `double`

`orientation` — Orientation in local navigation coordinate system

M-element quaternion column vector | 3-by-3-by-*M* real array

Orientation in the local navigation coordinate system, returned as an *M*-by-1 `quaternion` column vector or a 3-by-3-by-*M* real array.

Each quaternion or 3-by-3 rotation matrix is a frame rotation from the local navigation coordinate system to the current body coordinate system.

M is specified by the SamplesPerFrame property.

Data Types: `double`

`velocity` — Velocity in local navigation coordinate system (m/s)

M-by-3 matrix

Velocity in the local navigation coordinate system in meters per second, returned as an *M*-by-3 matrix.

M is specified by the SamplesPerFrame property.

Data Types: `double`

`acceleration` — Acceleration in local navigation coordinate system (m/s^2)

M-by-3 matrix

Acceleration in the local navigation coordinate system in meters per second squared, returned as an *M*-by-3 matrix.

M is specified by the SamplesPerFrame property.

Data Types: `double`

`angularVelocity` — Angular velocity in local navigation coordinate system (rad/s)

M-by-3 matrix

Angular velocity in the local navigation coordinate system in radians per second, returned as an *M*-by-3 matrix.

M is specified by the SamplesPerFrame property.

Data Types: `double`

Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

Specific to `waypointTrajectory`

waypointInfo	Get waypoint information table
lookupPose	Obtain pose information for certain time
perturbations	Perturbation defined on object
perturb	Apply perturbations to object

Common to All System Objects

clone	Create duplicate System object
step	Run System object algorithm
release	Release resources and allow changes to System object property values and input characteristics
reset	Reset internal states of System object
isDone	End-of-data status

Examples

Create Default `waypointTrajectory`

```
trajectory = waypointTrajectory
```

```
trajectory = 
  waypointTrajectory with properties:

         SampleRate: 100
     SamplesPerFrame: 1
          Waypoints: [2x3 double]
      TimeOfArrival: [2x1 double]
         Velocities: [2x3 double]
             Course: [2x1 double]
        GroundSpeed: [2x1 double]
          ClimbRate: [2x1 double]
        Orientation: [2x1 quaternion]
          AutoPitch: 0
           AutoBank: 0
     ReferenceFrame: 'NED'
```

Inspect the default waypoints and times of arrival by calling `waypointInfo`. By default, the waypoints indicate a stationary position for one second.

```
waypointInfo(trajectory)
```

```
ans=2×2 table
    TimeOfArrival     Waypoints 
    _____________    ___________
```

```
        0          0    0    0
        1          0    0    0
```

Create Square Trajectory

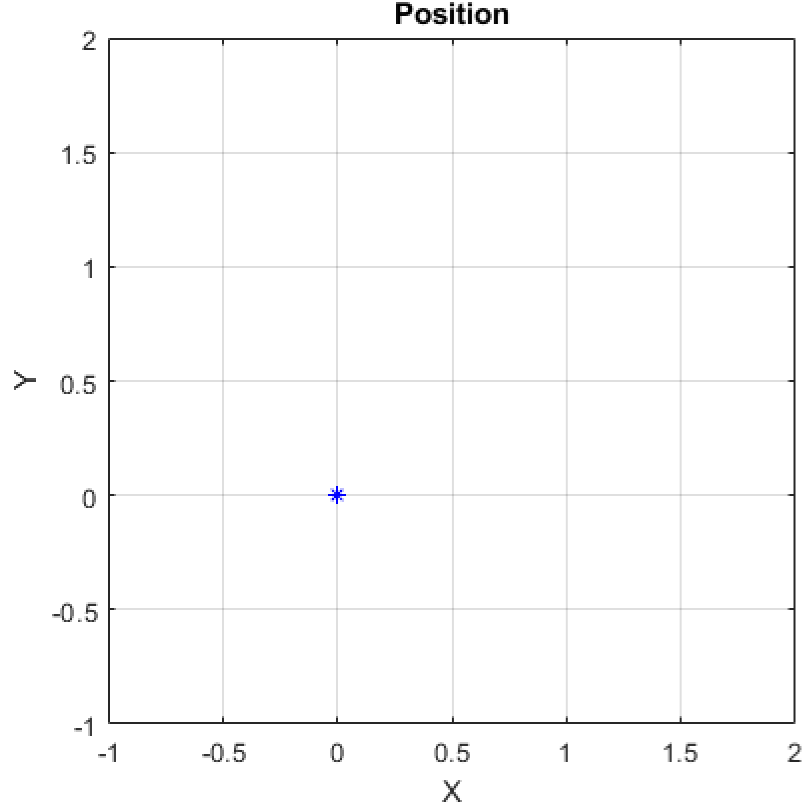
Create a square trajectory and examine the relationship between waypoint constraints, sample rate, and the generated trajectory.

Create a square trajectory by defining the vertices of the square. Define the orientation at each waypoint as pointing in the direction of motion. Specify a 1 Hz sample rate and use the default `SamplesPerFrame` of 1.

```
waypoints = [0,0,0; ... % Initial position
             0,1,0; ...
             1,1,0; ...
             1,0,0; ...
             0,0,0];    % Final position

toa = 0:4; % time of arrival

orientation = quaternion([0,0,0; ...
                          45,0,0; ...
                          135,0,0; ...
                          225,0,0; ...
                          0,0,0], ...
                          "eulerd","ZYX","frame");

trajectory = waypointTrajectory(waypoints, ...
    TimeOfArrival=toa, ...
    Orientation=orientation, ...
    SampleRate=1);
```

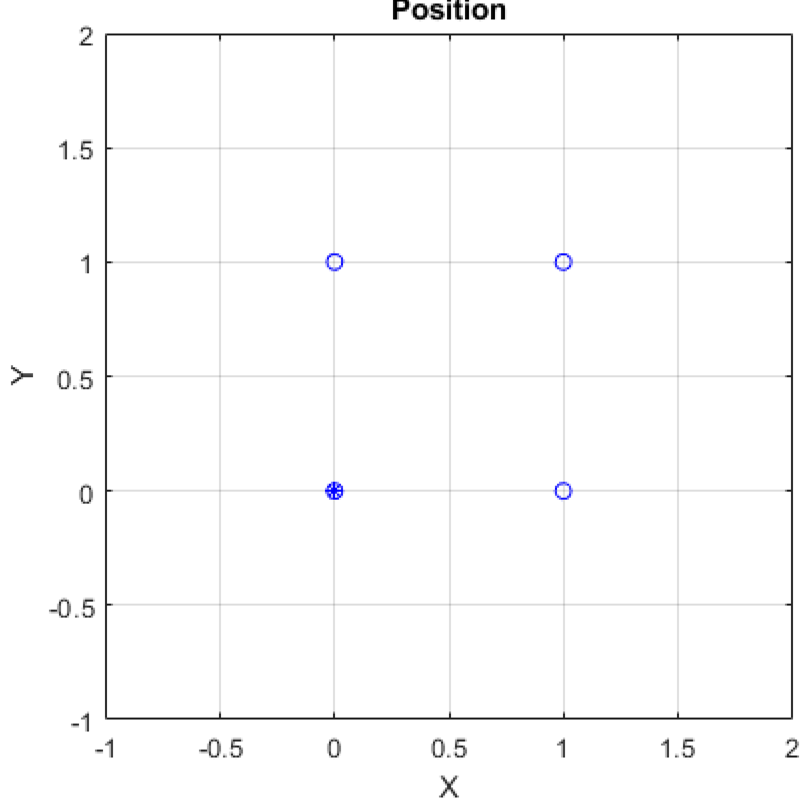
Create a figure and plot the initial position of the platform.

```
figure(1)
plot(waypoints(1,1),waypoints(1,2),"b*")
title("Position")
axis([-1,2,-1,2])
axis square
xlabel("X")
ylabel("Y")
grid on
hold on
```



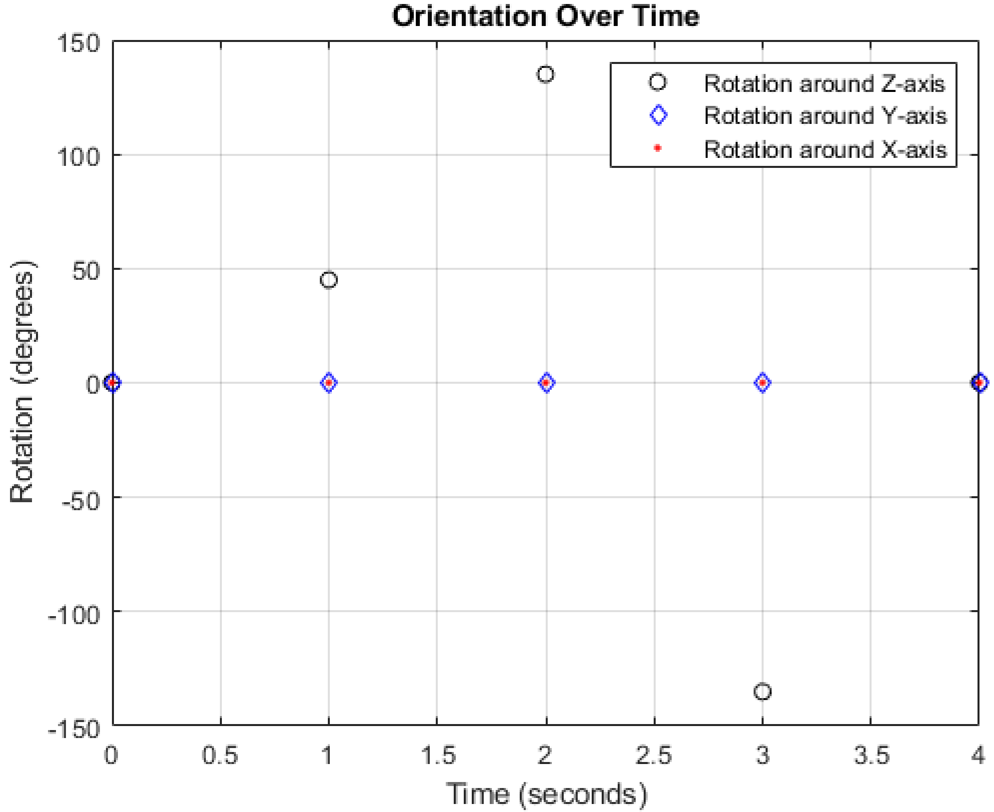
In a loop, step through the trajectory to output the current position and current orientation. Plot the current position and log the orientation. Use `pause` to mimic real-time processing.

```
orientationLog = zeros(toa(end)*trajectory.SampleRate,1,"quaternion");
count = 1;
while ~isDone(trajectory)
   [currentPosition,orientationLog(count)] = trajectory();

   plot(currentPosition(1),currentPosition(2),"bo")

   pause(trajectory.SamplesPerFrame/trajectory.SampleRate)
   count = count + 1;
end
hold off
```



Convert the orientation quaternions to Euler angles for easy interpretation, and then plot orientation over time.

```
figure(2)
eulerAngles = eulerd([orientation(1);orientationLog],"ZYX","frame");
plot(toa,eulerAngles(:,1),"ko", ...
     toa,eulerAngles(:,2),"bd", ...
     toa,eulerAngles(:,3),"r.");
title("Orientation Over Time")
legend("Rotation around Z-axis","Rotation around Y-axis","Rotation around X-axis")
xlabel("Time (seconds)")
ylabel("Rotation (degrees)")
grid on
```



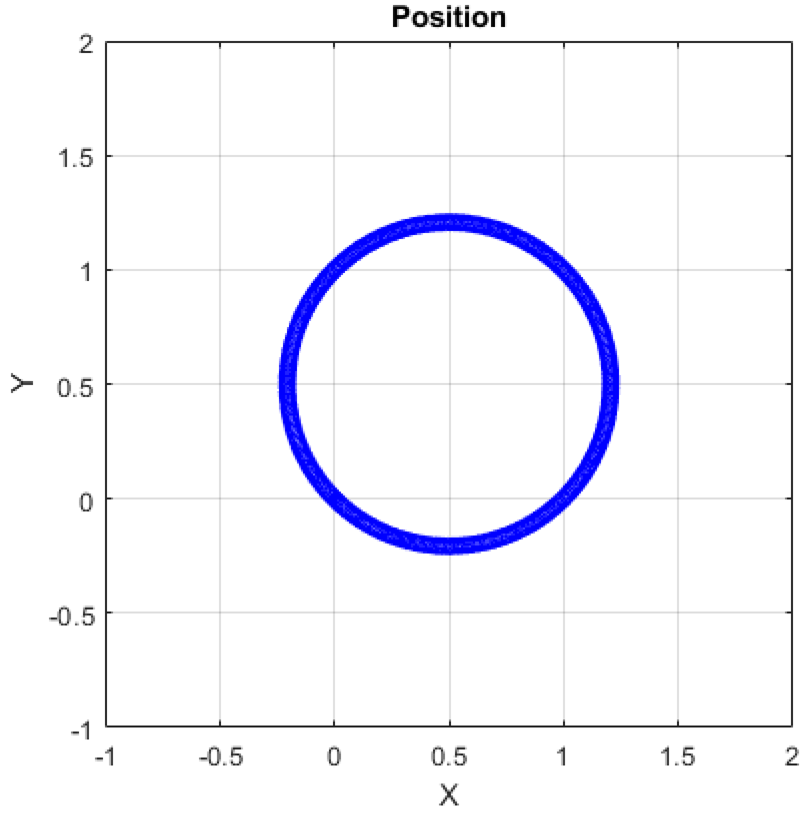
So far, the trajectory object has only output the waypoints that were specified during construction. To interpolate between waypoints, increase the sample rate to a rate faster than the time of arrivals of the waypoints. Set the `trajectory` sample rate to 100 Hz and call `reset`.

```
trajectory.SampleRate = 100;
reset(trajectory)
```

Create a figure and plot the initial position of the platform. In a loop, step through the trajectory to output the current position and current orientation. Plot the current position and log the orientation. Use `pause` to mimic real-time processing.

```
figure(1)
plot(waypoints(1,1),waypoints(1,2),"b*")
title("Position")
axis([-1,2,-1,2])
axis square
xlabel("X")
ylabel("Y")
grid on
hold on

orientationLog = zeros(toa(end)*trajectory.SampleRate,1,"quaternion");
count = 1;
while ~isDone(trajectory)
   [currentPosition,orientationLog(count)] = trajectory();

   plot(currentPosition(1),currentPosition(2),"bo")
```

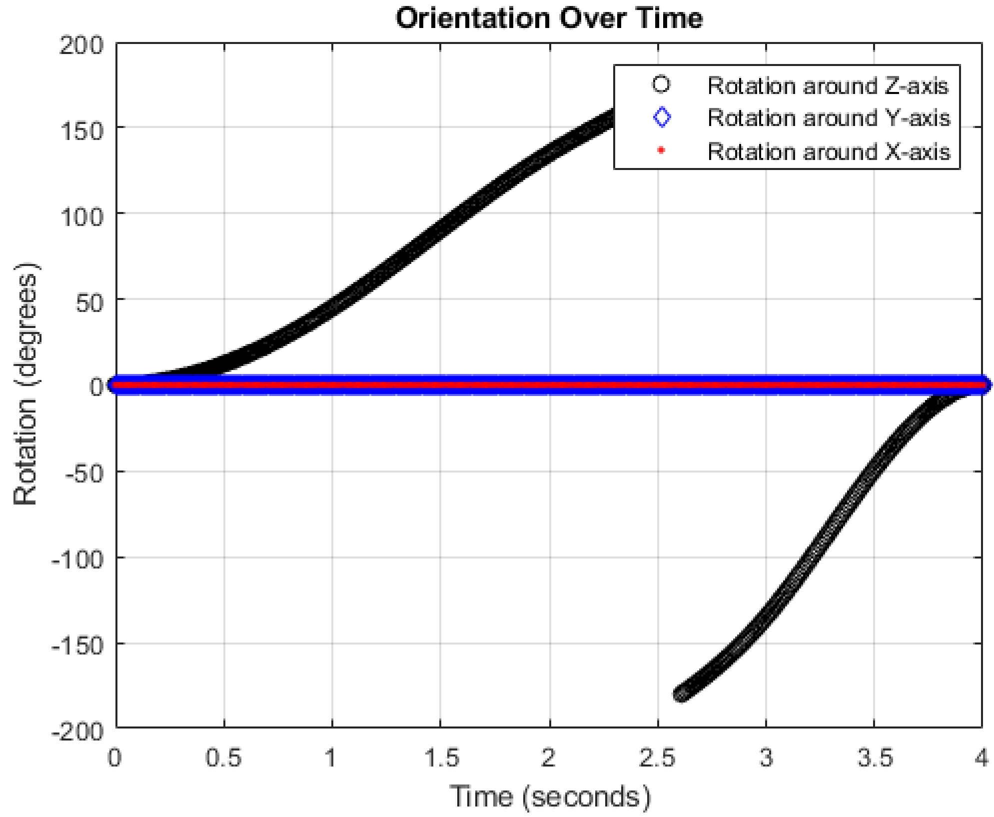
```
    pause(trajectory.SamplesPerFrame/trajectory.SampleRate)
    count = count + 1;
end
hold off
```



The trajectory output now appears circular. This is because the `waypointTrajectory` System object™ minimizes the acceleration and angular velocity when interpolating, which results in smoother, more realistic motions in most scenarios.

Convert the orientation quaternions to Euler angles for easy interpretation, and then plot orientation over time. The orientation is also interpolated.

```
figure(2)
eulerAngles = eulerd([orientation(1);orientationLog],"ZYX","frame");
t = 0:1/trajectory.SampleRate:4;
plot(t,eulerAngles(:,1),"ko", ...
     t,eulerAngles(:,2),"bd", ...
     t,eulerAngles(:,3),"r.");
title("Orientation Over Time")
legend("Rotation around Z-axis","Rotation around Y-axis","Rotation around X-axis")
xlabel("Time (seconds)")
ylabel("Rotation (degrees)")
grid on
```

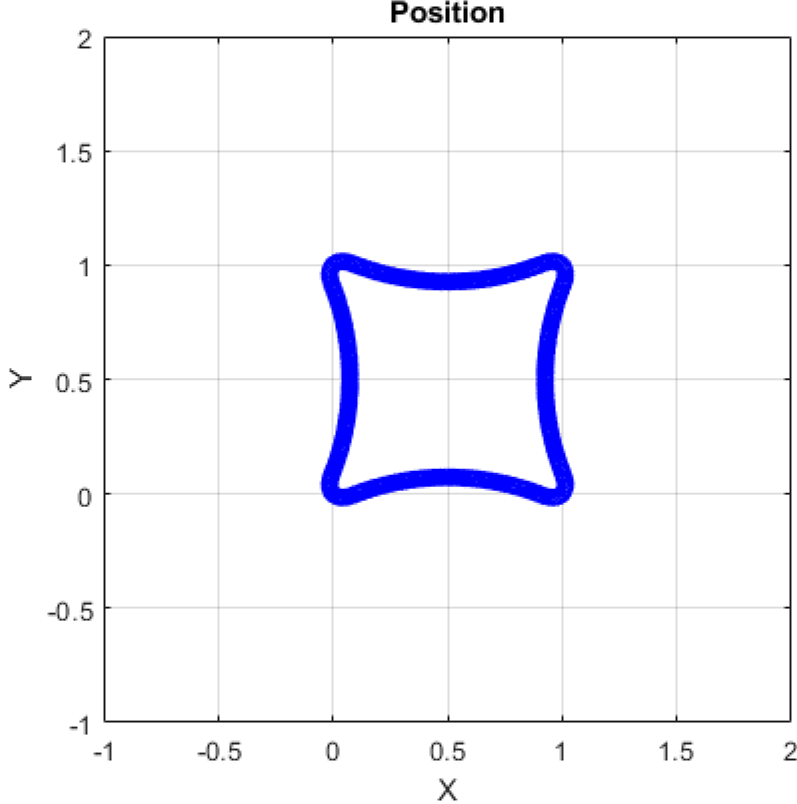


The `waypointTrajectory` algorithm interpolates the waypoints to create a smooth trajectory. To return to the square trajectory, provide more waypoints, especially around sharp changes. To track corresponding times, waypoints, and orientation, specify all the trajectory info in a single matrix.

```
                % Time, Waypoint, Orientation
trajectoryInfo = [0,   0,0,0,    0,0,0; ... % Initial position
                  0.1, 0,0.1,0,  0,0,0; ...

                  0.9, 0,0.9,0,  0,0,0; ...
                  1,   0,1,0,    45,0,0; ...
                  1.1, 0.1,1,0,  90,0,0; ...

                  1.9, 0.9,1,0,  90,0,0; ...
                  2,   1,1,0,    135,0,0; ...
                  2.1, 1,0.9,0,  180,0,0; ...

                  2.9, 1,0.1,0,  180,0,0; ...
                  3,   1,0,0,    225,0,0; ...
                  3.1, 0.9,0,0,  270,0,0; ...

                  3.9, 0.1,0,0,  270,0,0; ...
                  4,   0,0,0,    270,0,0];    % Final position

trajectory = waypointTrajectory(trajectoryInfo(:,2:4), ...
    TimeOfArrival=trajectoryInfo(:,1), ...
    Orientation=quaternion(trajectoryInfo(:,5:end),"eulerd","ZYX","frame"), ...
    SampleRate=100);
```

Create a figure and plot the initial position of the platform. In a loop, step through the trajectory to output the current position and current orientation. Plot the current position and log the orientation. Use `pause` to mimic real-time processing.

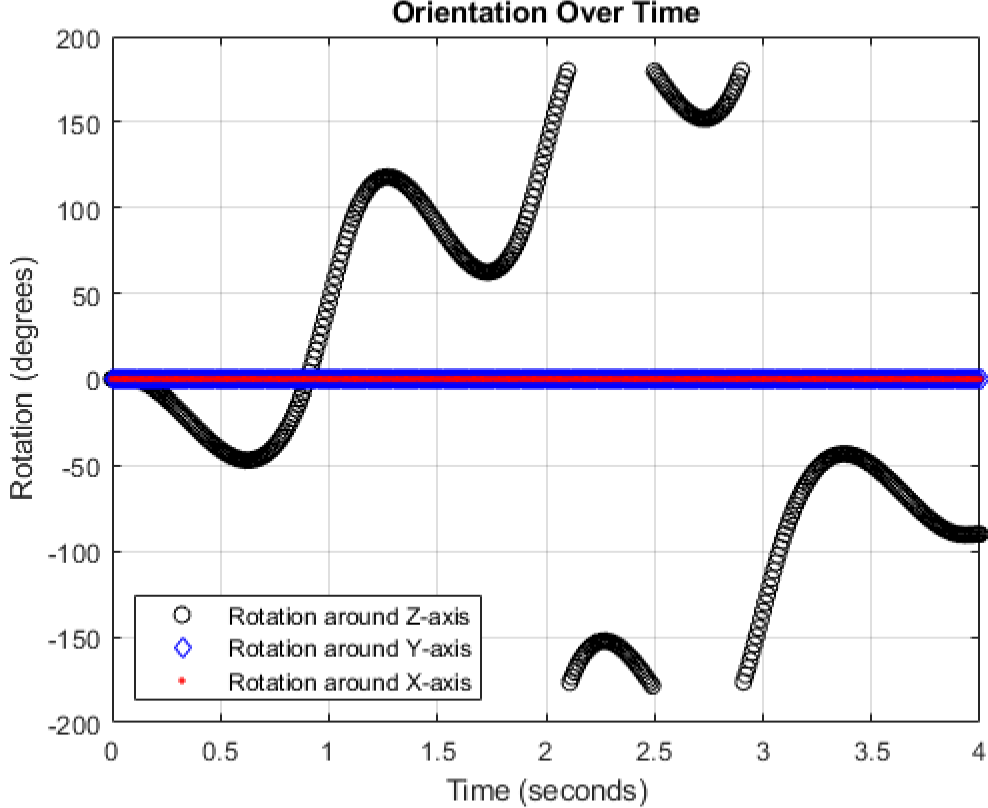
```
figure(1)
plot(waypoints(1,1),waypoints(1,2),"b*")
title("Position")
axis([-1,2,-1,2])
axis square
xlabel("X")
ylabel("Y")
grid on
hold on

orientationLog = zeros(toa(end)*trajectory.SampleRate,1,"quaternion");
count = 1;
while ~isDone(trajectory)
   [currentPosition,orientationLog(count)] = trajectory();

   plot(currentPosition(1),currentPosition(2),"bo")

   pause(trajectory.SamplesPerFrame/trajectory.SampleRate)
   count = count+1;
end
hold off
```



The trajectory output now appears more square-like, especially around the vertices with waypoints.

Convert the orientation quaternions to Euler angles for easy interpretation, and then plot orientation over time.

```
figure(2)
eulerAngles = eulerd([orientation(1);orientationLog],"ZYX","frame");
t = 0:1/trajectory.SampleRate:4;
eulerAngles = plot(t,eulerAngles(:,1),"ko", ...
                   t,eulerAngles(:,2),"bd", ...
                   t,eulerAngles(:,3),"r.");
title("Orientation Over Time")
legend("Rotation around Z-axis", ...
       "Rotation around Y-axis", ...
       "Rotation around X-axis", ...
       "Location", "SouthWest")
xlabel("Time (seconds)")
ylabel("Rotation (degrees)")
grid on
```



Create Arc Trajectory

This example shows how to create an arc trajectory using the `waypointTrajectory` System object™. `waypointTrajectory` creates a path through specified waypoints that minimizes acceleration and angular velocity. After creating an arc trajectory, you restrict the trajectory to be within preset bounds.

Create an Arc Trajectory

Define a constraints matrix consisting of waypoints, times of arrival, and orientation for an arc trajectory. The generated trajectory passes through the waypoints at the specified times with the specified orientation. The `waypointTrajectory` System object requires orientation to be specified using quaternions or rotation matrices. Convert the Euler angles saved in the constraints matrix to quaternions when specifying the `Orientation` property.

```
          % Arrival, Waypoints, Orientation
constraints = [0,    20,20,0,    90,0,0;
               3,    50,20,0,    90,0,0;
               4,    58,15.5,0,  162,0,0;
               5.5,  59.5,0,0    180,0,0];

trajectory = waypointTrajectory(constraints(:,2:4), ...
    TimeOfArrival=constraints(:,1), ...
    Orientation=quaternion(constraints(:,5:7),"eulerd","ZYX","frame"));
```

Call `waypointInfo` on `trajectory` to return a table of your specified constraints. The creation properties `Waypoints`, `TimeOfArrival`, and `Orientation` are variables of the table. The table is convenient for indexing while plotting.

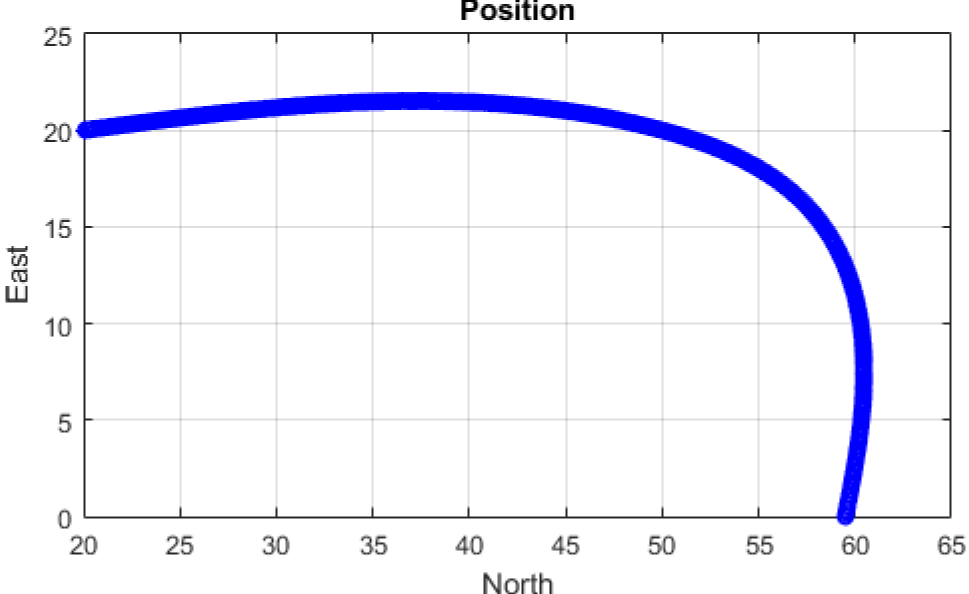
```
tInfo = waypointInfo(trajectory)
```

```
tInfo =

  4x3 table

    TimeOfArrival         Waypoints             Orientation
    _____________    ____________________    ________________

          0            20      20       0    {1x1 quaternion}
          3            50      20       0    {1x1 quaternion}
          4            58    15.5       0    {1x1 quaternion}
        5.5          59.5       0       0    {1x1 quaternion}
```

The trajectory object outputs the current position, velocity, acceleration, and angular velocity at each call. Call `trajectory` in a loop and plot the position over time. Cache the other outputs.

```
figure(1)
plot(tInfo.Waypoints(1,1),tInfo.Waypoints(1,2),"b*")
title("Position")
axis([20,65,0,25])
xlabel("North")
ylabel("East")
grid on
daspect([1 1 1])
hold on

orient = zeros(tInfo.TimeOfArrival(end)*trajectory.SampleRate,1,"quaternion");
vel = zeros(tInfo.TimeOfArrival(end)*trajectory.SampleRate,3);
acc = vel;
angVel = vel;

count = 1;
while ~isDone(trajectory)
```

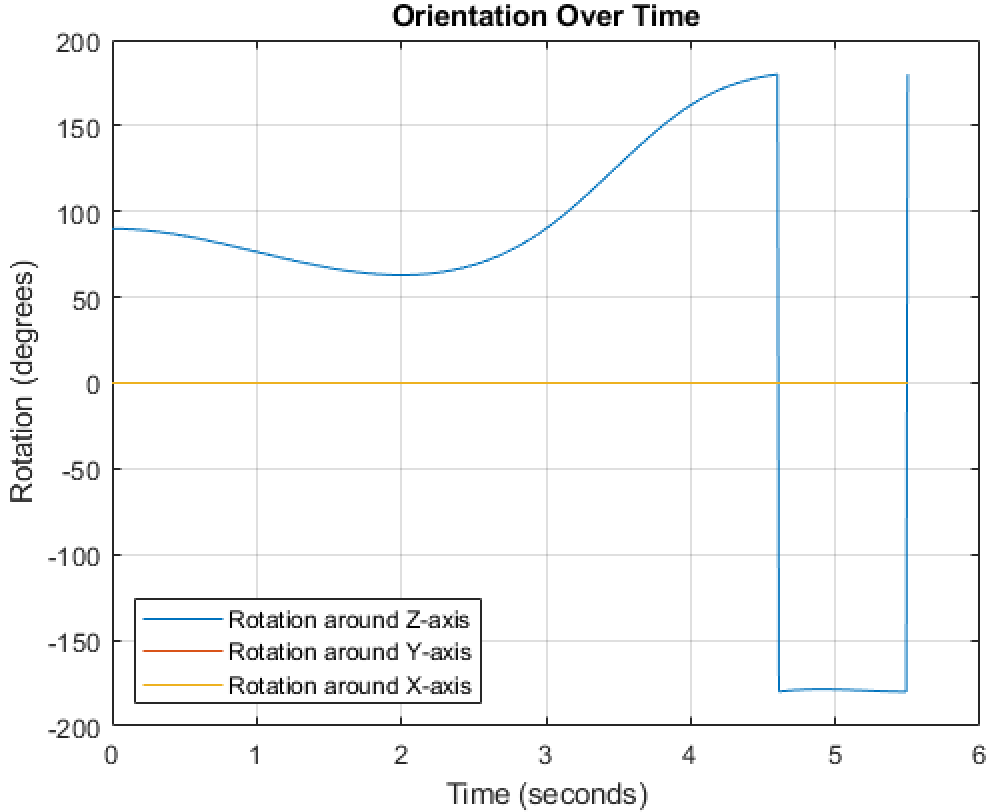
```
    [pos,orient(count),vel(count,:),acc(count,:),angVel(count,:)] = trajectory();

    plot(pos(1),pos(2),"bo")

    pause(trajectory.SamplesPerFrame/trajectory.SampleRate)
    count = count + 1;
end
```

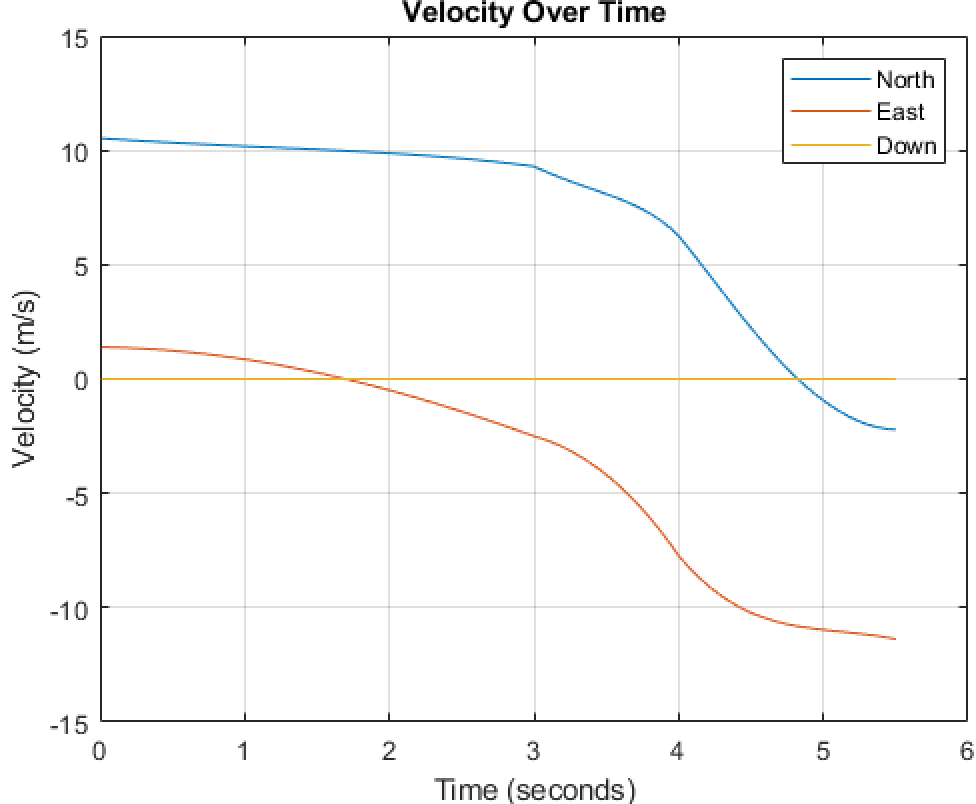


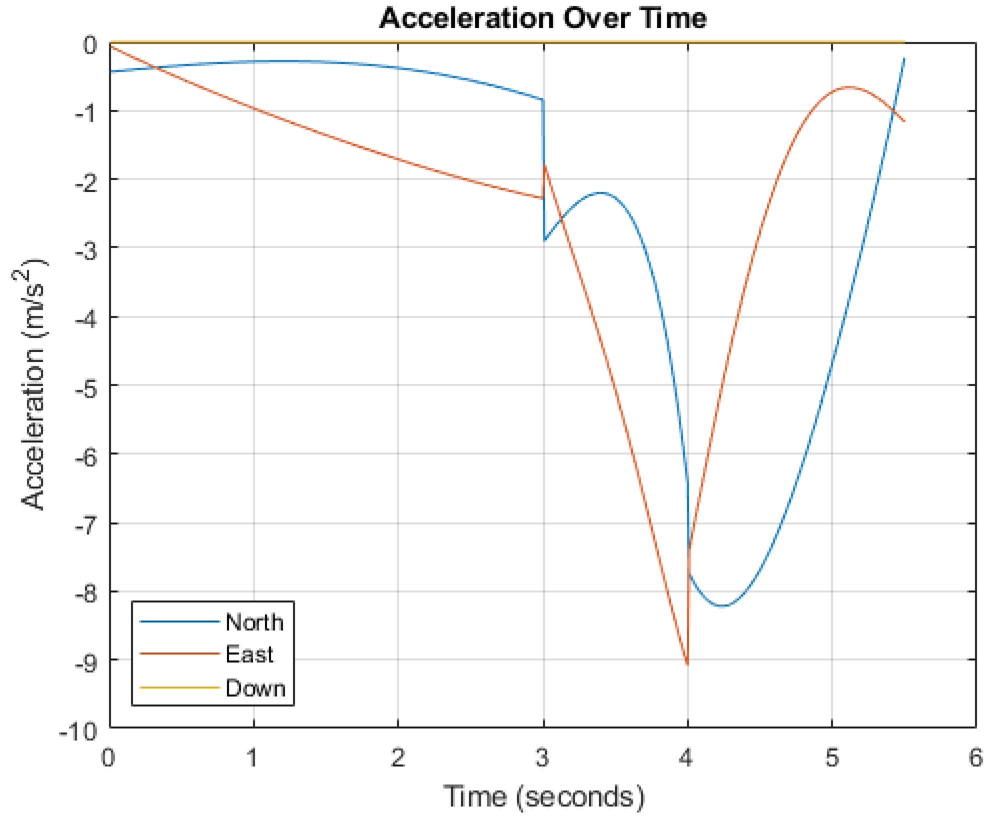
Inspect the orientation, velocity, acceleration, and angular velocity over time. The `waypointTrajectory` System object™ creates a path through the specified constraints that minimized acceleration and angular velocity.

```
figure(2)
timeVector = 0:(1/trajectory.SampleRate):tInfo.TimeOfArrival(end);
eulerAngles = eulerd([tInfo.Orientation{1};orient],"ZYX","frame");
plot(timeVector,eulerAngles(:,1), ...
     timeVector,eulerAngles(:,2), ...
     timeVector,eulerAngles(:,3));
title("Orientation Over Time")
legend("Rotation around Z-axis", ...
       "Rotation around Y-axis", ...
       "Rotation around X-axis", ...
       "Location","southwest")
xlabel("Time (seconds)")
ylabel("Rotation (degrees)")
grid on
```

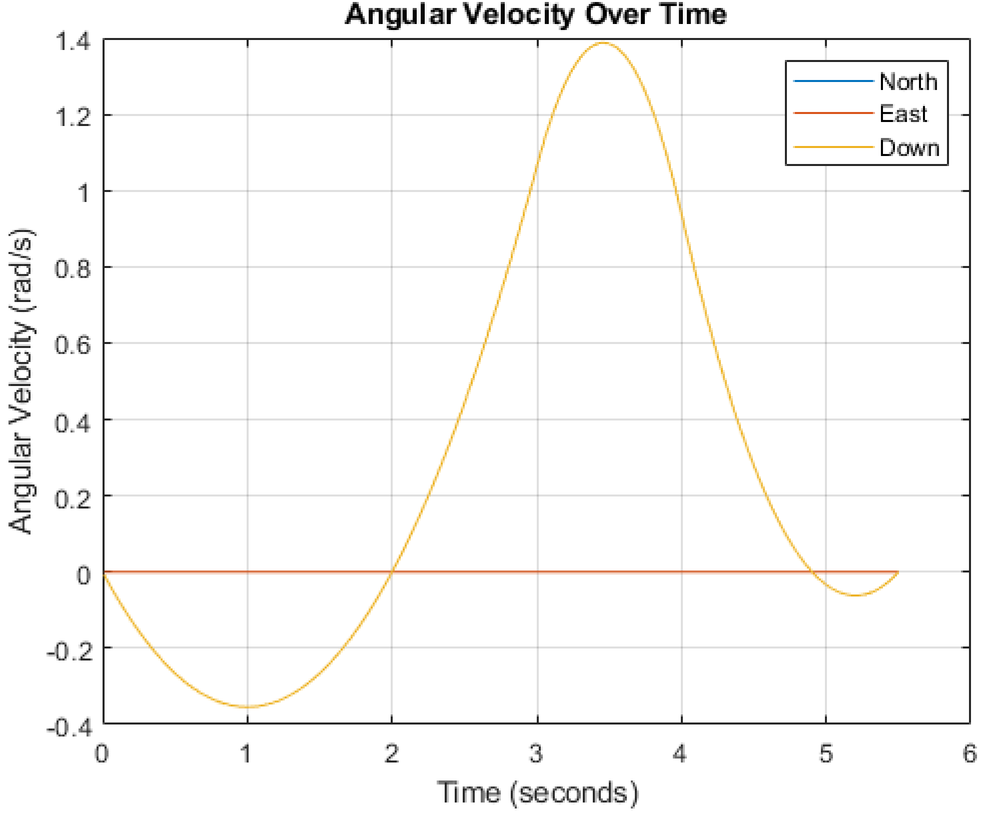
```
figure(3)
plot(timeVector(2:end),vel(:,1), ...
     timeVector(2:end),vel(:,2), ...
     timeVector(2:end),vel(:,3));
title("Velocity Over Time")
legend("North","East","Down")
xlabel("Time (seconds)")
ylabel("Velocity (m/s)")
grid on

figure(4)
plot(timeVector(2:end),acc(:,1), ...
     timeVector(2:end),acc(:,2), ...
     timeVector(2:end),acc(:,3));
title("Acceleration Over Time")
legend("North","East","Down","Location","southwest")
xlabel("Time (seconds)")
ylabel("Acceleration (m/s^2)")
grid on

figure(5)
plot(timeVector(2:end),angVel(:,1), ...
     timeVector(2:end),angVel(:,2), ...
     timeVector(2:end),angVel(:,3));
title("Angular Velocity Over Time")
legend("North","East","Down")
xlabel("Time (seconds)")
ylabel("Angular Velocity (rad/s)")
grid on
```


Orientation Over Time
Rotation (degrees)
Time (seconds)
200
150
100
50
0
-50
-100
-150
-200
0
1
2
3
4
5
6
Rotation around Z-axis
Rotation around Y-axis
Rotation around X-axis


Velocity Over Time
North
East
Down
Velocity (m/s)
Time (seconds)
15
10
5
0
-5
-10
-15
0
1
2
3
4
5
6



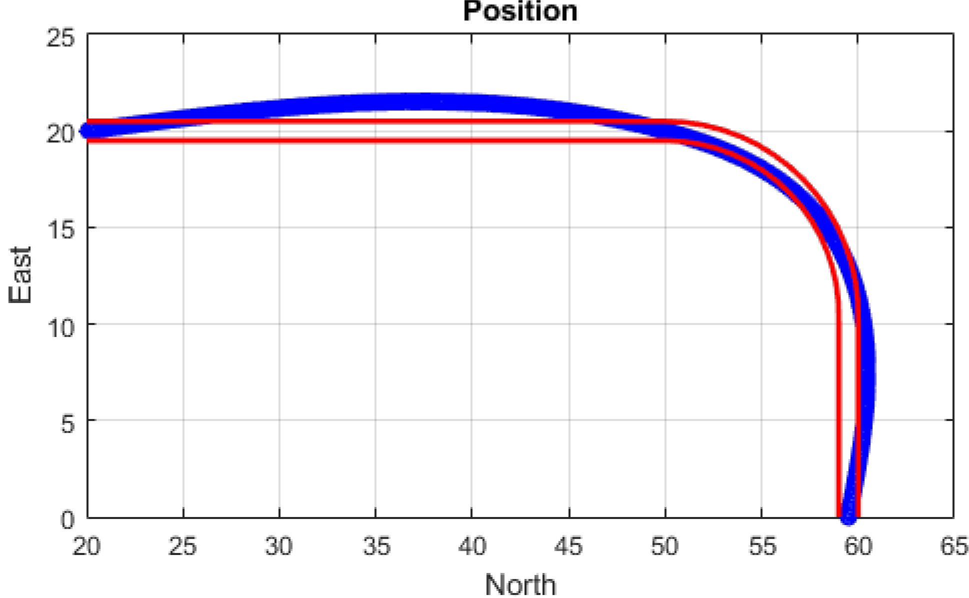
Acceleration Over Time
Acceleration (m/s^2)
Time (seconds)
North
East
Down
0
-1
-2
-3
-4
-5
-6
-7
-8
-9
-10
0
1
2
3
4
5
6



Restrict Arc Trajectory Within Preset Bounds

You can specify additional waypoints to create trajectories within given bounds. Create upper and lower bounds for the arc trajectory.

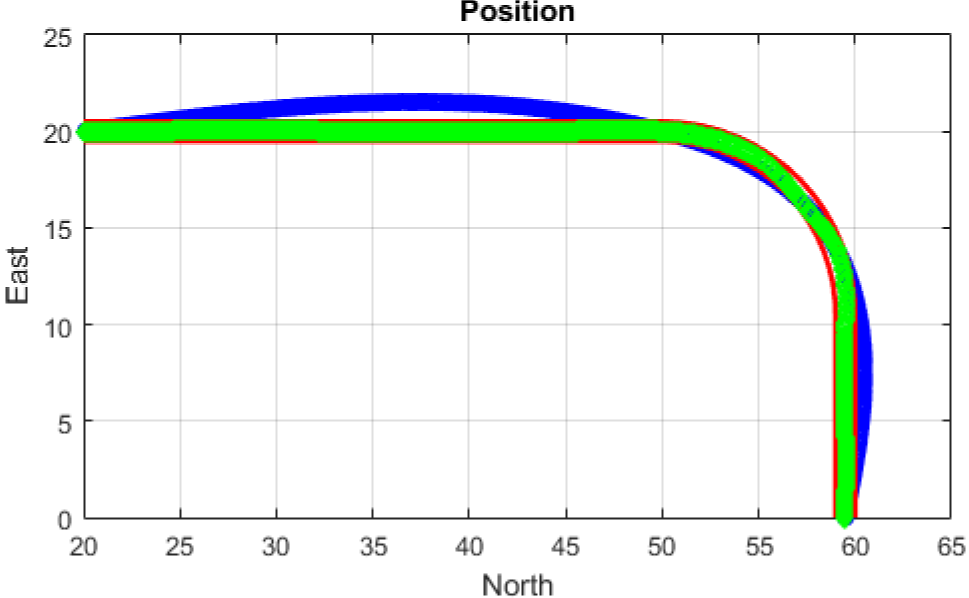
```
figure(1)
xUpperBound = [(20:50)';50+10*sin(0:0.1:pi/2)';60*ones(11,1)];
yUpperBound = [20.5.*ones(31,1);10.5+10*cos(0:0.1:pi/2)';(10:-1:0)'];

xLowerBound = [(20:49)';50+9*sin(0:0.1:pi/2)';59*ones(11,1)];
yLowerBound = [19.5.*ones(30,1);10.5+9*cos(0:0.1:pi/2)';(10:-1:0)'];

plot(xUpperBound,yUpperBound,"r","LineWidth",2);
plot(xLowerBound,yLowerBound,"r","LineWidth",2)
```



To create a trajectory within the bounds, add additional waypoints. Create a new `waypointTrajectory` System object™, and then call it in a loop to plot the generated trajectory. Cache the orientation, velocity, acceleration, and angular velocity output from the `trajectory` object.

```
              % Time,  Waypoint,     Orientation
constraints = [0,     20,20,0,       90,0,0;
               1.5,   35,20,0,       90,0,0;
               2.5    45,20,0,       90,0,0;
               3,     50,20,0,       90,0,0;
               3.3,   53,19.5,0,     108,0,0;
               3.6,   55.5,18.25,0,  126,0,0;
               3.9,   57.5,16,0,     144,0,0;
               4.2,   59,14,0,       162,0,0;
               4.5,   59.5,10,0      180,0,0;
               5,     59.5,5,0       180,0,0;
               5.5,   59.5,0,0       180,0,0];

trajectory = waypointTrajectory(constraints(:,2:4), ...
    TimeOfArrival=constraints(:,1), ...
    Orientation=quaternion(constraints(:,5:7),"eulerd","ZYX","frame"));
tInfo = waypointInfo(trajectory);

figure(1)
plot(tInfo.Waypoints(1,1),tInfo.Waypoints(1,2),"b*")

count = 1;
```

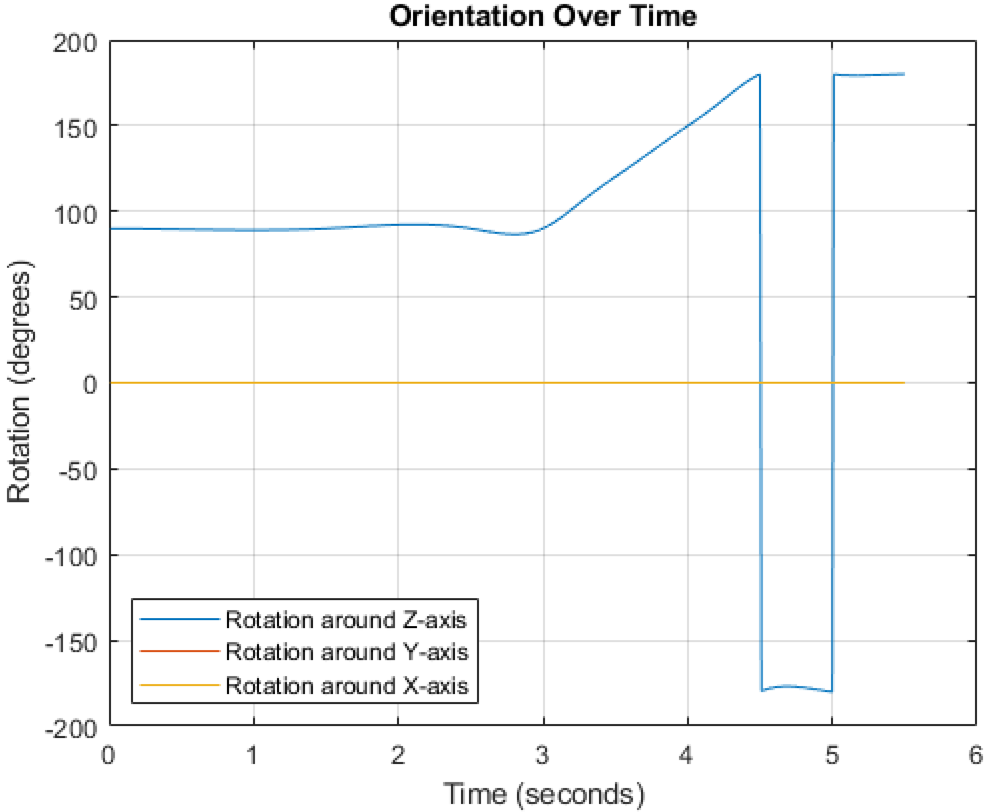
```
while ~isDone(trajectory)
   [pos,orient(count),vel(count,:),acc(count,:),angVel(count,:)] = trajectory();

   plot(pos(1),pos(2),"gd")

   pause(trajectory.SamplesPerFrame/trajectory.SampleRate)
   count = count + 1;
end
```

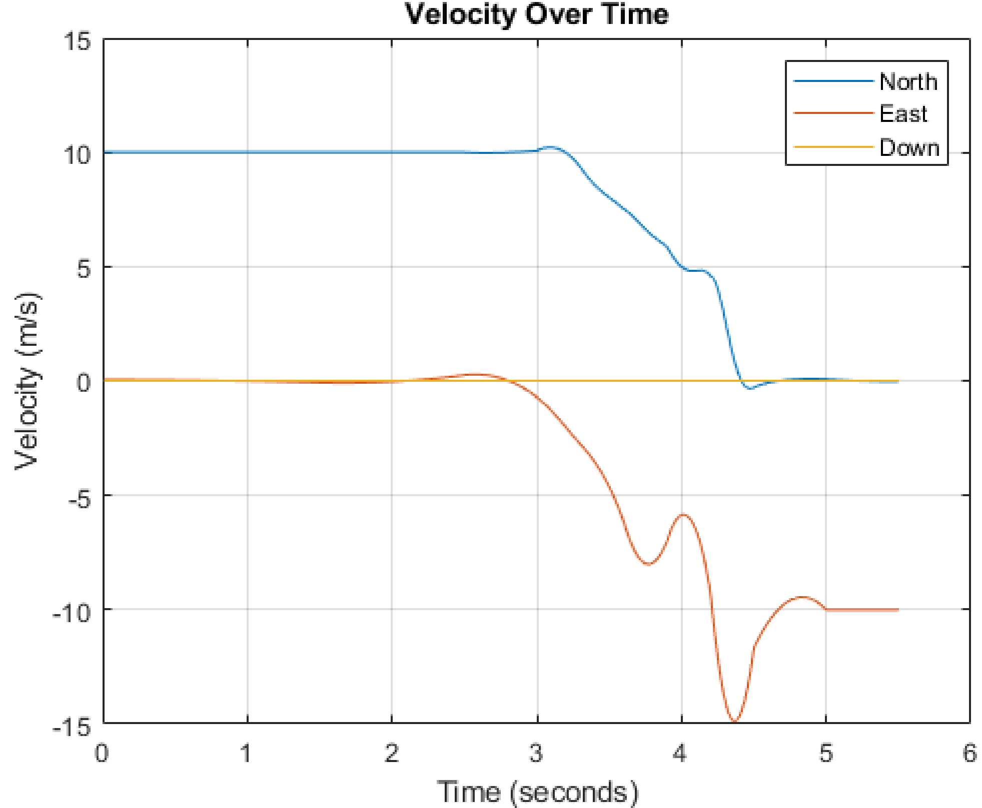


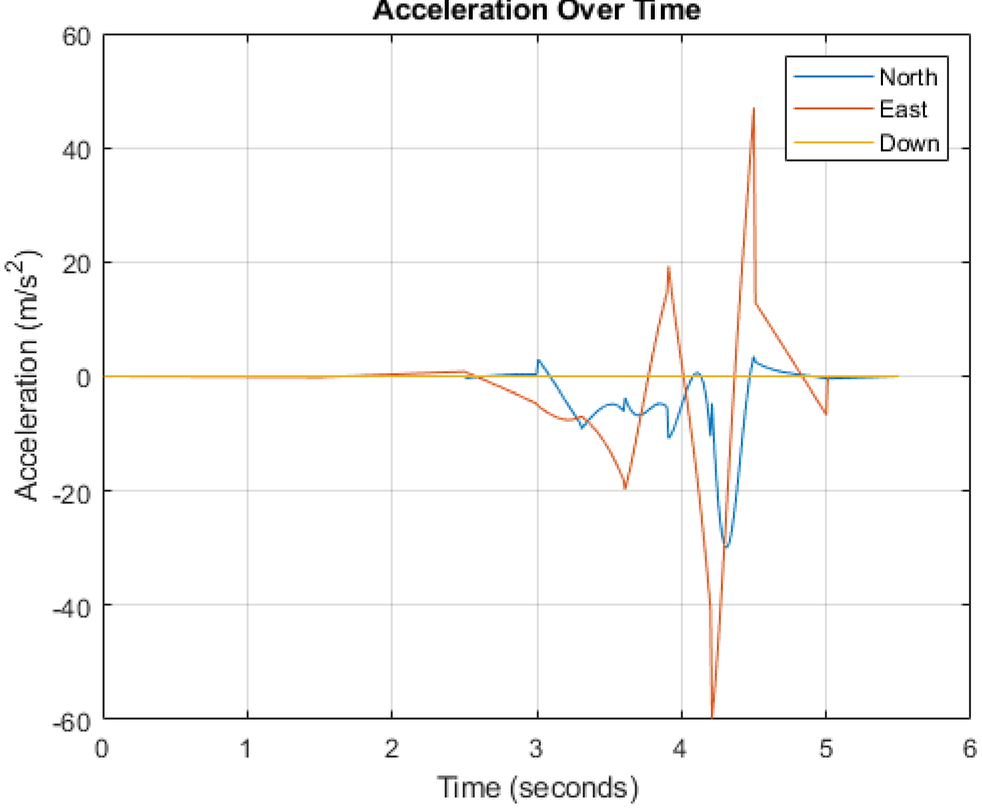
The generated trajectory now fits within the specified boundaries. Visualize the orientation, velocity, acceleration, and angular velocity of the generated trajectory.

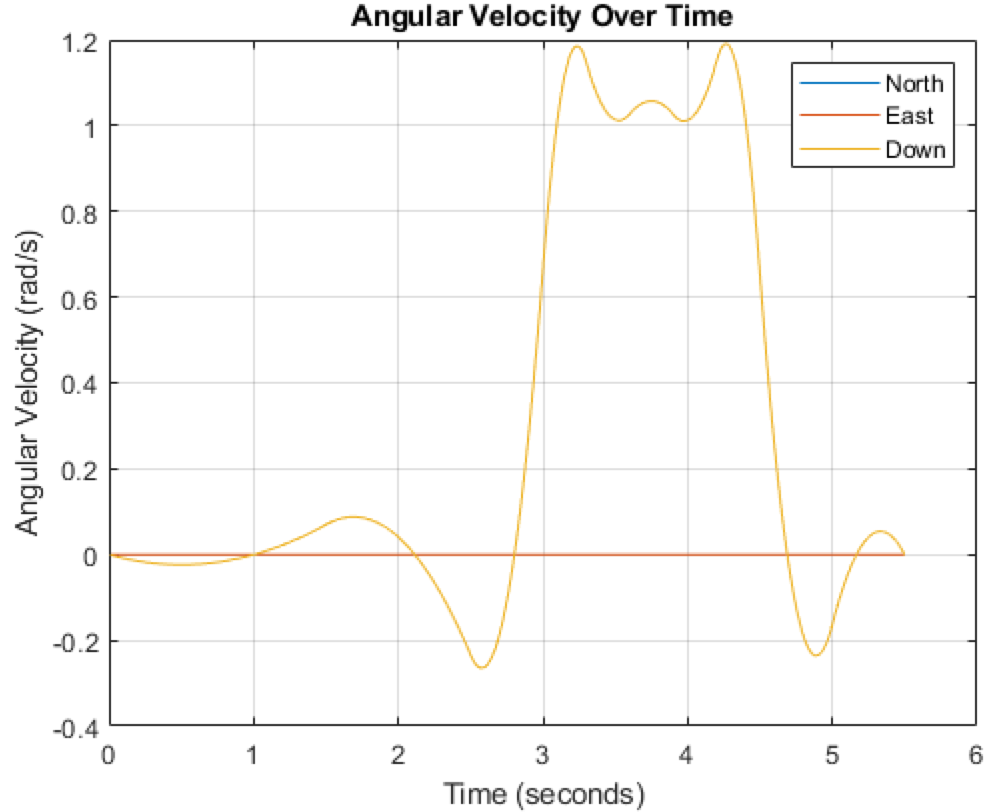
```
figure(2)
timeVector = 0:(1/trajectory.SampleRate):tInfo.TimeOfArrival(end);
eulerAngles = eulerd(orient,"ZYX","frame");
plot(timeVector(2:end),eulerAngles(:,1), ...
     timeVector(2:end),eulerAngles(:,2), ...
     timeVector(2:end),eulerAngles(:,3));
title("Orientation Over Time")
legend("Rotation around Z-axis", ...
       "Rotation around Y-axis", ...
       "Rotation around X-axis", ...
       "Location","southwest")
xlabel("Time (seconds)")
ylabel("Rotation (degrees)")
grid on

figure(3)
```

```
plot(timeVector(2:end),vel(:,1), ...
     timeVector(2:end),vel(:,2), ...
     timeVector(2:end),vel(:,3));
title("Velocity Over Time")
legend("North","East","Down")
xlabel("Time (seconds)")
ylabel("Velocity (m/s)")
grid on

figure(4)
plot(timeVector(2:end),acc(:,1), ...
     timeVector(2:end),acc(:,2), ...
     timeVector(2:end),acc(:,3));
title("Acceleration Over Time")
legend("North","East","Down")
xlabel("Time (seconds)")
ylabel("Acceleration (m/s^2)")
grid on

figure(5)
plot(timeVector(2:end),angVel(:,1), ...
     timeVector(2:end),angVel(:,2), ...
     timeVector(2:end),angVel(:,3));
title("Angular Velocity Over Time")
legend("North","East","Down")
xlabel("Time (seconds)")
ylabel("Angular Velocity (rad/s)")
grid on
```


Orientation Over Time
Rotation (degrees)
Time (seconds)
200
150
100
50
0
-50
-100
-150
-200
0
1
2
3
4
5
6
Rotation around Z-axis
Rotation around Y-axis
Rotation around X-axis


Velocity Over Time
North
East
Down
Velocity (m/s)
Time (seconds)
15
10
5
0
-5
-10
-15
0
1
2
3
4
5
6

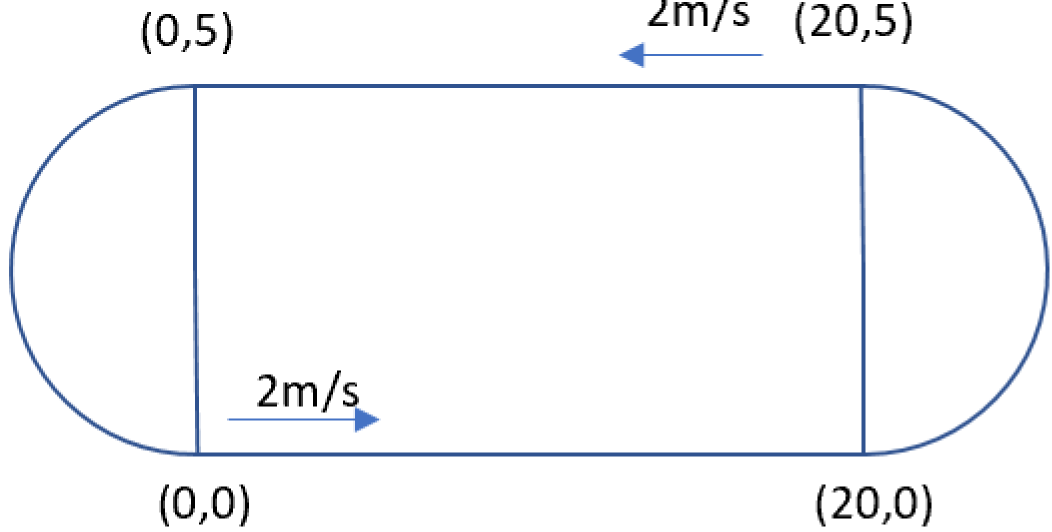

Acceleration Over Time
North
East
Down
Acceleration (m/s2)
Time (seconds)
60
40
20
0
-20
-40
-60
0
1
2
3
4
5
6



Note that while the generated trajectory now fits within the spatial boundaries, the acceleration and angular velocity of the trajectory are somewhat erratic. This is due to over-specifying waypoints.

Generate Racetrack Trajectory Using `waypointTrajectory`

Consider a racetrack trajectory as the following.



The four corner points of the trajectory are (0,0,0), (20,0,0), (20,5,0) and (0,5,0) in meters, respectively. Therefore, specify the waypoints of a loop as:

```
wps = [0 0 0;
      20 0 0;
      20 5 0;
      0  5 0;
      0  0 0];
```

Assume the trajectory has a constant speed of 2 m/s, and thus the velocities at the five waypoints are:

```
vels = [2 0 0;
        2 0 0;
       -2 0 0;
       -2 0 0;
        2 0 0];
```

The time of arrival for the five waypoints is:

```
t = cumsum([0 20/2 5*pi/2/2 20/2 5*pi/2/2]');
```

The orientation of the trajectory at the five waypoints are:

```
eulerAngs = [0 0 0;
             0 0 0;
           180 0 0;
           180 0 0;
             0 0 0]; % Angles in degrees.
% Convert Euler angles to quaternions.
quats = quaternion(eulerAngs,"eulerd","ZYX","frame");
```

Specify the sample rate as 100 for smoothing trajectory lines.

```
fs = 100;
```

Construct the `waypointTrajectory`.

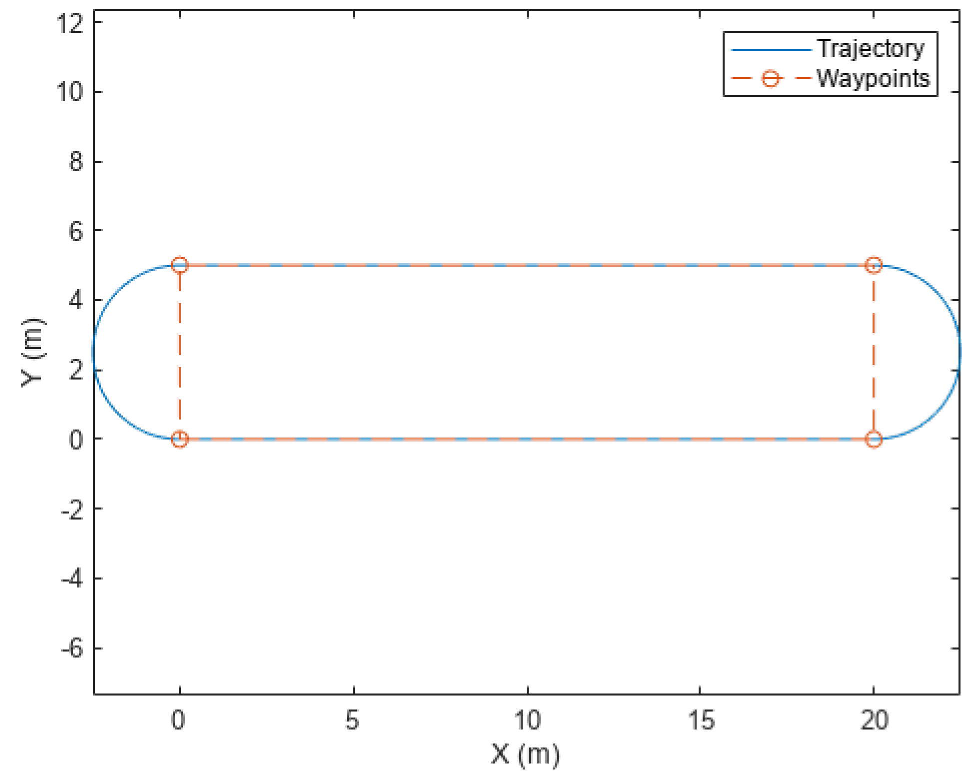
```
traj = waypointTrajectory(wps,SampleRate=fs, ...
        Velocities=vels,...
        TimeOfArrival=t,...
        Orientation=quats);
```

Sample and plot the trajectory.

```
[pos, orient, vel, acc, angvel] = traj();
i = 1;

spf = traj.SamplesPerFrame;
while ~isDone(traj)
    idx = (i+1):(i+spf);
    [pos(idx,:), orient(idx,:), ...
        vel(idx,:), acc(idx,:), angvel(idx,:)] = traj();
    i = i+spf;
end
```

Plot the trajectory and the specified waypoints.

```
plot(pos(:,1),pos(:,2), wps(:,1),wps(:,2), "--o")
xlabel("X (m)")
ylabel("Y (m)")
zlabel("Z (m)")
legend({"Trajectory", "Waypoints"})
axis equal
```



Create Trajectory Using Waypoints and Ground Speed

Create a `waypointTrajectory` object that connects two waypoints. The velocity of the trajectory at the two waypoints is 0 m/s and 10 m/s, respectively. Restrict the jerk limit to 0.5 m/s^3 to enable the trapezoidal acceleration profile.

```
waypoints = [0  0  0;
             10 50 10];
speeds = [0 10];
jerkLimit = 0.5;
trajectory = waypointTrajectory(waypoints,GroundSpeed=speeds,JerkLimit=jerkLimit);
```

Obtain the initial time and final time of the trajectory by querying the `TimeOfArrival` property. Create time stamps to sample the trajectory.

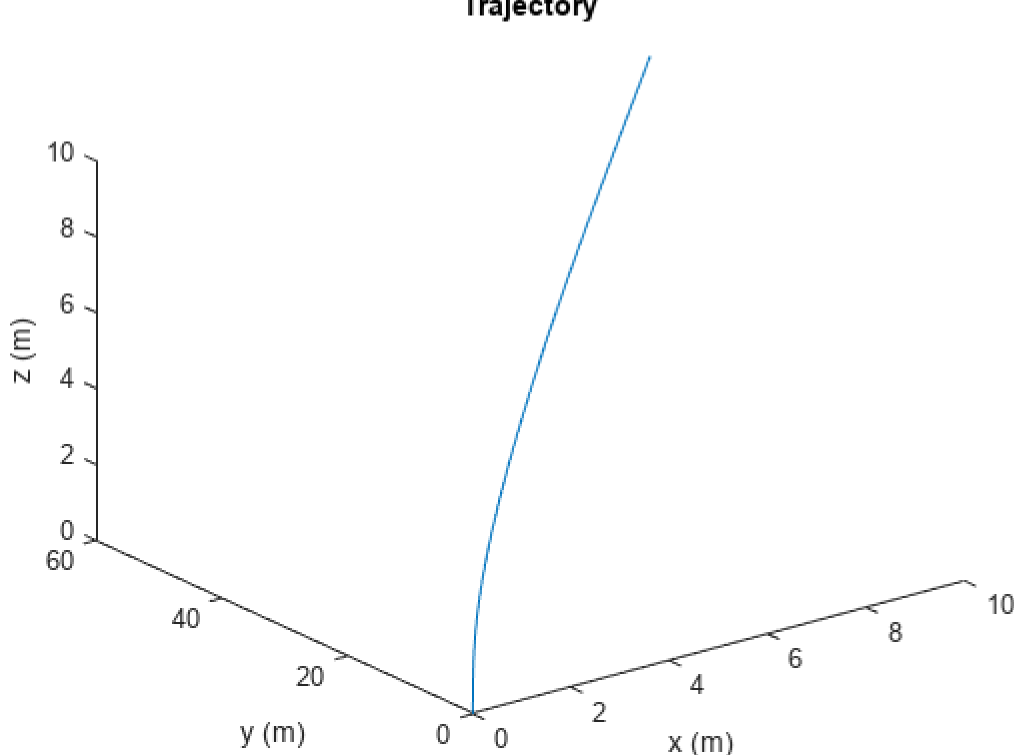
```
t0 = trajectory.TimeOfArrival(1);
tf = trajectory.TimeOfArrival(end);
sampleTimes = linspace(t0,tf,100);
```

Obtain the position, velocity, and acceleration information at these sampled time stamps using the `lookupPose` object function.

```
[position,~,velocity,acceleration,~] = lookupPose(trajectory,sampleTimes);
```

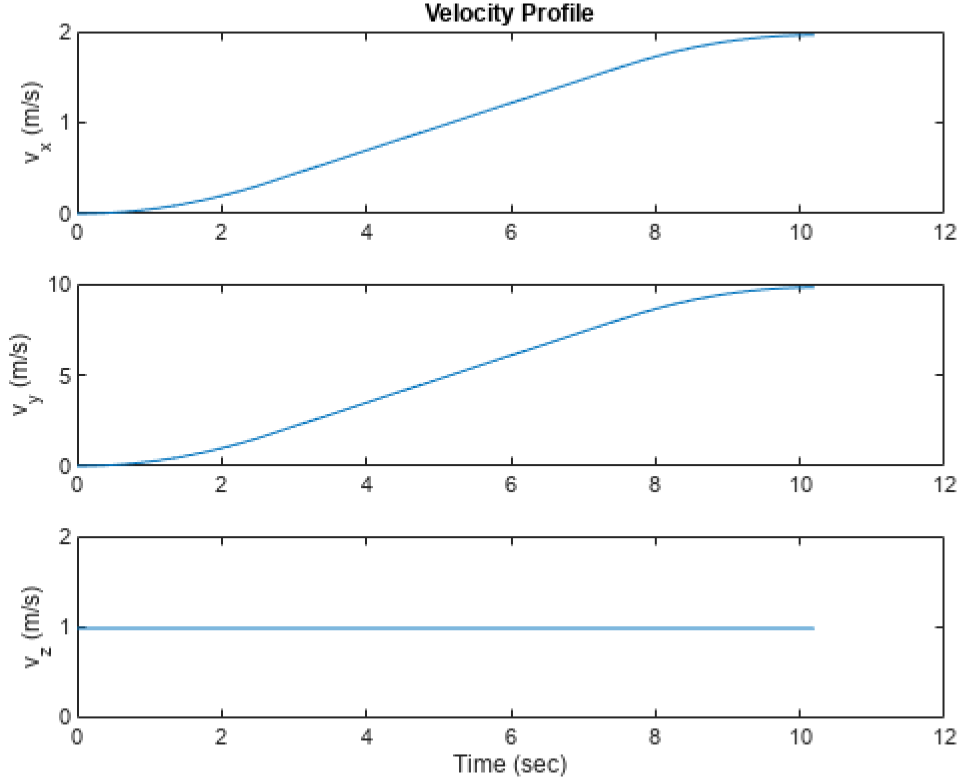
Plot the trajectory.

```
figure()
plot3(position(:,1),position(:,2),position(:,3))
xlabel("x (m)")
ylabel("y (m)")
zlabel("z (m)")
title("Trajectory")
```



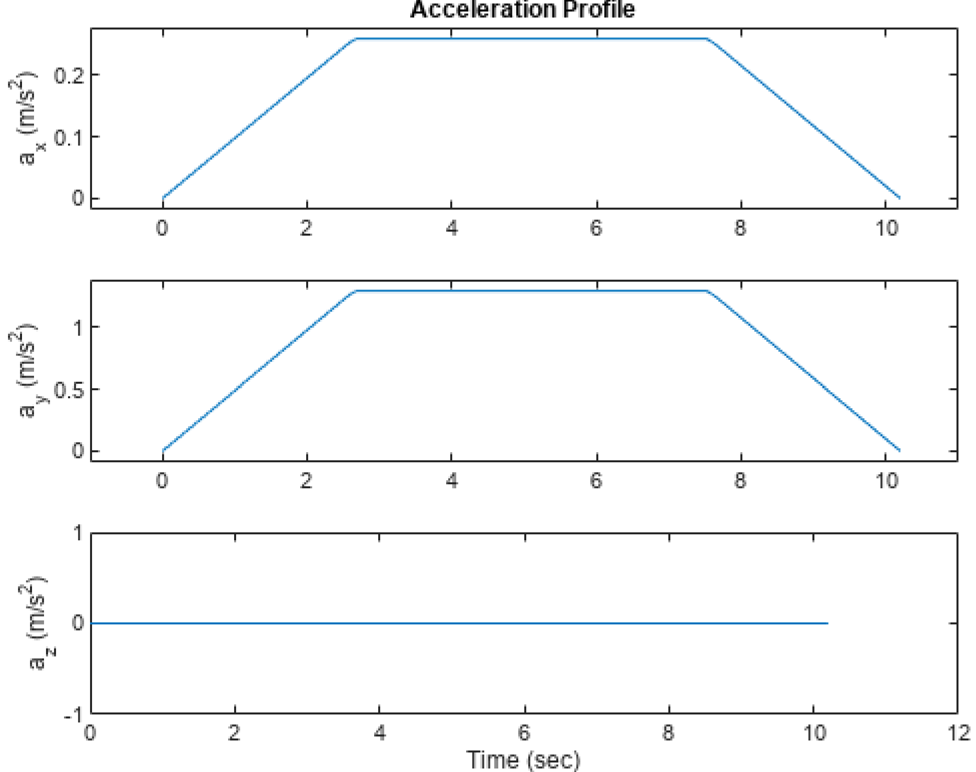
Plot the velocity profile.

```
figure()
subplot(3,1,1)
plot(sampleTimes,velocity(:,1));
ylabel("v_x (m/s)")
title("Velocity Profile")
subplot(3,1,2)
plot(sampleTimes,velocity(:,2));
ylabel("v_y (m/s)")
subplot(3,1,3)
plot(sampleTimes,velocity(:,3));
ylabel("v_z (m/s)")
xlabel("Time (sec)")
```



Plot the acceleration profile. From the results, the acceleration profile of the planar motion is trapezoidal.

```
figure()
subplot(3,1,1)
plot(sampleTimes,acceleration(:,1));
axis padded
ylabel("a_x (m/s^2)")
title("Acceleration Profile")
subplot(3,1,2)
plot(sampleTimes,acceleration(:,2));
ylabel("a_y (m/s^2)")
axis padded
subplot(3,1,3)
plot(sampleTimes,acceleration(:,3));
ylabel("a_z (m/s^2)")
xlabel("Time (sec)")
```



Algorithms

The `waypointTrajectory` System object defines a trajectory that smoothly passes through waypoints. The trajectory connects the waypoints through an interpolation that assumes the gravity direction expressed in the trajectory reference frame is constant. Generally, you can use `waypointTrajectory` to model platform or vehicle trajectories within a hundreds of kilometers distance span.

The planar path of the trajectory (the *x*-*y* plane projection) consists of piecewise, clothoid curves. The curvature of the curve between two consecutive waypoints varies linearly with the curve length between them. The tangent direction of the path at each waypoint is chosen to minimize discontinuities in the curvature, unless the course is specified explicitly via the `Course` property or implicitly via the `Velocities` property. Once the path is established, the object uses cubic Hermite interpolation to compute the location of the vehicle throughout the path as a function of time and the planar distance traveled. If the `JerkLimit` property is specified, the objects produces a horizontal trapezoidal acceleration profile for any segment that is between two waypoints. The trapezoidal acceleration profile consists of three subsegments:

- A constant-magnitude jerk subsegment
- A constant-magnitude acceleration subsegment
- A constant-magnitude jerk subsegment

The normal component (*z*-component) of the trajectory is subsequently chosen to satisfy a shape-preserving piecewise spline (PCHIP) unless the climb rate is specified explicitly via the `ClimbRate`

property or the third column of the `Velocities` property. Choose the sign of the climb rate based on the selected `ReferenceFrame`:

- When an 'ENU' reference frame is selected, specifying a positive climb rate results in an increasing value of z.
- When an 'NED' reference frame is selected, specifying a positive climb rate results in a decreasing value of z.

You can define the orientation of the vehicle through the path in two primary ways:

- If the `Orientation` property is specified, then the object uses a piecewise-cubic, quaternion spline to compute the orientation along the path as a function of time.
- If the `Orientation` property is not specified, then the yaw of the vehicle is always aligned with the path. The roll and pitch are then governed by the `AutoBank` and `AutoPitch` property values, respectively.

AutoBank	AutoPitch	Description
`false`	`false`	The vehicle is always level (zero pitch and roll). This is typically used for large marine vessels.
`false`	`true`	The vehicle pitch is aligned with the path, and its roll is always zero. This is typically used for ground vehicles.
`true`	`false`	The vehicle pitch and roll are chosen so that its local z-axis is aligned with the net acceleration (including gravity). This is typically used for rotary-wing craft.
`true`	`true`	The vehicle roll is chosen so that its local transverse plane aligns with the net acceleration (including gravity). The vehicle pitch is aligned with the path. This is typically used for two-wheeled vehicles and fixed-wing aircraft.

Version History

Introduced in R2022a

R2023a: Specify `waypointTrajectory` using ground speed or velocity input and new properties

When creating a waypointTrajectory object, if you specify the velocity or ground speed input, the time-of-arrival input is no longer required. When you do not specify the time-of-arrival input, you can use these new properties:

- JerkLimit — Longitudinal limit of trajectory jerk. Jerk is the derivative of the translational acceleration. If you specify a finite value for the jerk limit, waypointTrajectory produces a horizontal trapezoidal acceleration profile based on JerkLimit.
- InitialTime — Time before trajectory starts. If specified as nonzero, waypointTrajectory delays the start of the trajectory by the initial time.
- WaitTime— Wait time at each waypoint. If specified as nonzero for a waypoint, waypointTrajectory waits at the waypoint.

R2022b: Specify wait and reverse motion for waypoint trajectory

You can now specify wait and reverse motion using the waypointTrajectory System object.

- To let the trajectory wait at a specific waypoint, simply repeat the waypoint coordinate in two consecutive rows when specifying the Waypoints property.
- To render reverse motion, separate positive (forward) and negative (backward) groundspeed values by a zero value in the GroundSpeed property.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

The object function, waypointInfo, does not support code generation.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

Objects
robotPlatform

2

Functions

angdiff

Difference between two angles

Syntax

```
delta = angdiff(alpha,beta)
```

```
delta = angdiff(alpha)
```

Description

`delta = angdiff(alpha,beta)` calculates the difference between the angles `alpha` and `beta`. This function subtracts `alpha` from `beta` with the result wrapped on the interval `[-pi,pi]`. You can specify the input angles as single values or as arrays of angles that have the same number of values.

`delta = angdiff(alpha)` returns the angular difference between adjacent elements of `alpha` along the first dimension whose size does not equal 1. If `alpha` is a vector of length *n*, the first entry is subtracted from the second, the second from the third, etc. The output, `delta`, is a vector of length *n-1*. If `alpha` is an *m*-by-*n* matrix with *m* greater than 1, the output, `delta`, will be a matrix of size *m-1*-by-*n*. If `alpha` is a scalar, `delta` returns as an empty vector.

Examples

Calculate Difference Between Two Angles

```
d = angdiff(pi,2*pi)
```

```
d = 3.1416
```

Calculate Difference Between Two Angle Arrays

```
d = angdiff([pi/2 3*pi/4 0],[pi pi/2 -pi])
```

```
d = 1×3

    1.5708   -0.7854   -3.1416
```

Calculate Angle Differences of Adjacent Elements

```
angles = [pi pi/2 pi/4 pi/2];
d = angdiff(angles)
```

```
d = 1×3
```

```
   -1.5708   -0.7854    0.7854
```

Input Arguments

`alpha` — Angle in radians
scalar | vector | matrix | multidimensional array

Angle in radians, specified as a scalar, vector, matrix, or multidimensional array. This is the angle that is subtracted from `beta` when specified. If `alpha` is a scalar, `delta` returns as an empty vector.

Example: `pi/2`

`beta` — Angle in radians
scalar | vector | matrix | multidimensional array

Angle in radians, specified as a scalar, vector, matrix, or multidimensional array of the same size as `alpha`. This is the angle that `alpha` is subtracted from when specified.

Example: `pi/2`

Output Arguments

`delta` — Difference between two angles
scalar | vector | matrix | multidimensional array

Angular difference between two angles, returned as a scalar, vector, or array. `delta` is wrapped to the interval `[-pi,pi]`. If `alpha` is a scalar, `delta` returns as an empty vector.

Version History

Introduced in R2015a

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

angvel

Angular velocity from quaternion array

Syntax

```
AV = angvel(Q,dt,'frame')
AV = angvel(Q,dt,'point')
[AV,qf] = angvel(Q,dt,fp,qi)
```

Description

`AV = angvel(Q,dt,'frame')` returns the angular velocity array from an array of quaternions, `Q`. The quaternions in `Q` correspond to frame rotation. The initial quaternion is assumed to represent zero rotation.

`AV = angvel(Q,dt,'point')` returns the angular velocity array from an array of quaternions, `Q`. The quaternions in `Q` correspond to point rotation. The initial quaternion is assumed to represent zero rotation.

`[AV,qf] = angvel(Q,dt,fp,qi)` allows you to specify the initial quaternion, `qi`, and the type of rotation, `fp`. It also returns the final quaternion, `qf`.

Examples

Generate Angular Velocity From Quaternion Array

Create an array of quaternions.

```
eulerAngles = [(0:10:90).',zeros(numel(0:10:90),2)];
q = quaternion(eulerAngles,'eulerd','ZYX','frame');
```

Specify the time step and generate the angular velocity array.

```
dt = 1;
av = angvel(q,dt,'frame') % units in rad/s
```

```
av = 10×3

         0         0         0
         0         0    0.1743
         0         0    0.1743
         0         0    0.1743
         0         0    0.1743
         0         0    0.1743
         0         0    0.1743
         0         0    0.1743
         0         0    0.1743
         0         0    0.1743
```

Input Arguments

`Q` — Quaternions
N-by-1 vector of quaternions

Quaternions, specified as an *N*-by-1 vector of quaternions.

Data Types: `quaternion`

`dt` — Time step
nonnegative scalar

Time step, specified as a nonnegative scalar.

Data Types: `single` | `double`

`fp` — Type of rotation
`'frame'` | `'point'`

Type of rotation, specified as `'frame'` or `'point'`.

`qi` — Initial quaternion
quaternion

Initial quaternion, specified as a quaternion.

Data Types: `quaternion`

Output Arguments

`AV` — Angular velocity
N-by-3 real matrix

Angular velocity, returned as an *N*-by-3 real matrix. *N* is the number of quaternions given in the input *Q*. Each row of the matrix corresponds to an angular velocity vector.

`qf` — Final quaternion
quaternion

Final quaternion, returned as a quaternion. `qf` is the same as the last quaternion in the *Q* input.

Data Types: `quaternion`

Version History

Introduced in R2020a

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

quaternion

axang2quat

Convert axis-angle rotation to quaternion

Syntax

`quat = axang2quat(axang)`

Description

`quat = axang2quat(axang)` converts a rotation given in axis-angle form, `axang`, to quaternion, `quat`.

Examples

Convert Axis-Angle Rotation to Quaternion

```
axang = [1 0 0 pi/2];
quat = axang2quat(axang)
```

```
quat = 1×4

    0.7071    0.7071         0         0
```

Input Arguments

axang — Rotation given in axis-angle form
n-by-4 matrix

Rotation given in axis-angle form, specified as an *n*-by-4 matrix of *n* axis-angle rotations. The first three elements of every row specify the rotation axis, and the last element defines the rotation angle (in radians).

Example: `[1 0 0 pi/2]`

Output Arguments

quat — Unit quaternion
n-by-4 matrix

Unit quaternion, returned as an *n*-by-4 matrix containing *n* quaternions. Each quaternion, one per row, is of the form $q = [w\ x\ y\ z]$, with *w* as the scalar number.

Example: `[0.7071 0.7071 0 0]`

Version History

Introduced in R2015a

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

`quat2axang` | `quaternion`

Topics
"Coordinate Transformations in Robotics"

axang2rotm

Convert axis-angle rotation to rotation matrix

Syntax

```
rotm = axang2rotm(axang)
```

Description

`rotm = axang2rotm(axang)` converts a rotation given in axis-angle form, `axang`, to an orthonormal rotation matrix, `rotm`. When using the rotation matrix, premultiply it with the coordinates to be rotated (as opposed to postmultiplying).

Examples

Convert Axis-Angle Rotation to Rotation Matrix

```
axang = [0 1 0 pi/2];
rotm = axang2rotm(axang)
```

```
rotm = 3×3

    0.0000         0    1.0000
         0    1.0000         0
   -1.0000         0    0.0000
```

Input Arguments

axang — Rotation given in axis-angle form
n-by-4 matrix

Rotation given in axis-angle form, specified as an *n*-by-4 matrix of *n* axis-angle rotations. The first three elements of every row specify the rotation axis, and the last element defines the rotation angle (in radians).

Example: `[1 0 0 pi/2]`

Output Arguments

`rotm` — Rotation matrix
3-by-3-by-*n* matrix

Rotation matrix, returned as a 3-by-3-by-*n* matrix containing *n* rotation matrices. Each rotation matrix has a size of 3-by-3 and is orthonormal. When using the rotation matrix, premultiply it with the coordinates to be rotated (as opposed to postmultiplying).

Example: `[0 0 1; 0 1 0; -1 0 0]`

Version History

Introduced in R2015a

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`rotm2axang` | `so2` | `so3`

Topics

"Coordinate Transformations in Robotics"

axang2tform

Convert axis-angle rotation to homogeneous transformation

Syntax

`tform = axang2tform(axang)`

Description

`tform = axang2tform(axang)` converts a rotation given in axis-angle form, `axang`, to a homogeneous transformation matrix, `tform`. When using the transformation matrix, premultiply it with the coordinates to be transformed (as opposed to postmultiplying).

Examples

Convert Axis-Angle Rotation to Homogeneous Transformation

```
axang = [1 0 0 pi/2];
tform = axang2tform(axang)
```

```
tform = 4×4

    1.0000         0         0         0
         0    0.0000   -1.0000         0
         0    1.0000    0.0000         0
         0         0         0    1.0000
```

Input Arguments

axang — Rotation given in axis-angle form
n-by-4 matrix

Rotation given in axis-angle form, specified as an *n*-by-4 matrix of *n* axis-angle rotations. The first three elements of every row specify the rotation axis, and the last element defines the rotation angle (in radians).

Example: `[1 0 0 pi/2]`

Output Arguments

`tform` — Homogeneous transformation
4-by-4-by-*n* matrix

Homogeneous transformation matrix, specified by a 4-by-4-by-*n* matrix of *n* homogeneous transformations. When using the transformation matrix, premultiply it with the coordinates to be formed (as opposed to postmultiplying).

Example: `[0 0 1 0; 0 1 0 0; -1 0 0 0; 0 0 0 1]`

Version History

Introduced in R2015a

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`tform2axang` | `se2` | `se3`

Topics

"Coordinate Transformations in Robotics"

bsplinepolytraj

Generate polynomial trajectories using B-splines

Syntax

```
[q,qd,qdd,pp] = bsplinepolytraj(controlPoints,tInterval,tSamples)
```

Description

`[q,qd,qdd,pp] = bsplinepolytraj(controlPoints,tInterval,tSamples)` generates a piecewise cubic B-spline trajectory that falls in the control polygon defined by `controlPoints`. The trajectory is uniformly sampled between the start and end times given in `tInterval`. The function returns the positions, velocities, and accelerations at the input time samples, `tSamples`. The function also returns the piecewise polynomial `pp` form of the polynomial trajectory with respect to time.

Examples

Compute B-Spline Trajectory for 2-D Planar Motion

Use the `bsplinepolytraj` function with a given set of 2-D *xy* control points. The B-spline uses these control points to create a trajectory inside the polygon. The start and end time for the trajectory are also given.

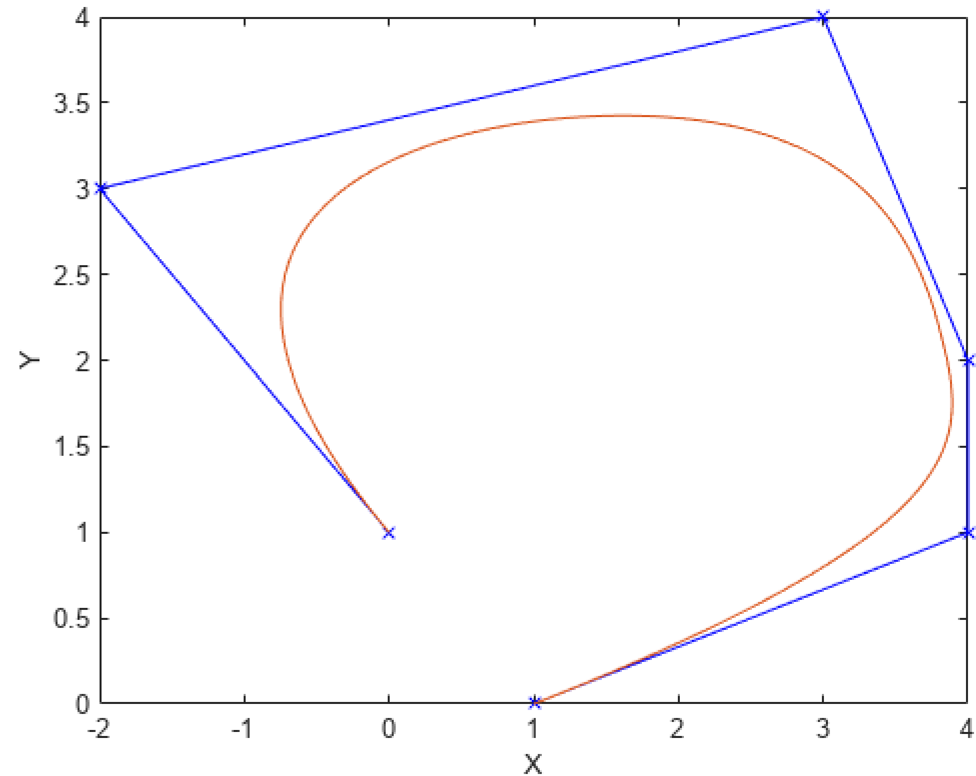
```
cpts = [1 4 4 3 -2 0; 0 1 2 4 3 1];
tpts = [0 5];
```

Compute the B-spline trajectory. The function outputs the trajectory positions (`q`), velocity (`qd`), acceleration (`qdd`), time vector (`tvec`), and polynomial coefficients (`pp`) of the polynomial that achieves the waypoints using the control points.

```
tvec = 0:0.01:5;
[q, qd, qdd, pp] = bsplinepolytraj(cpts,tpts,tvec);
```

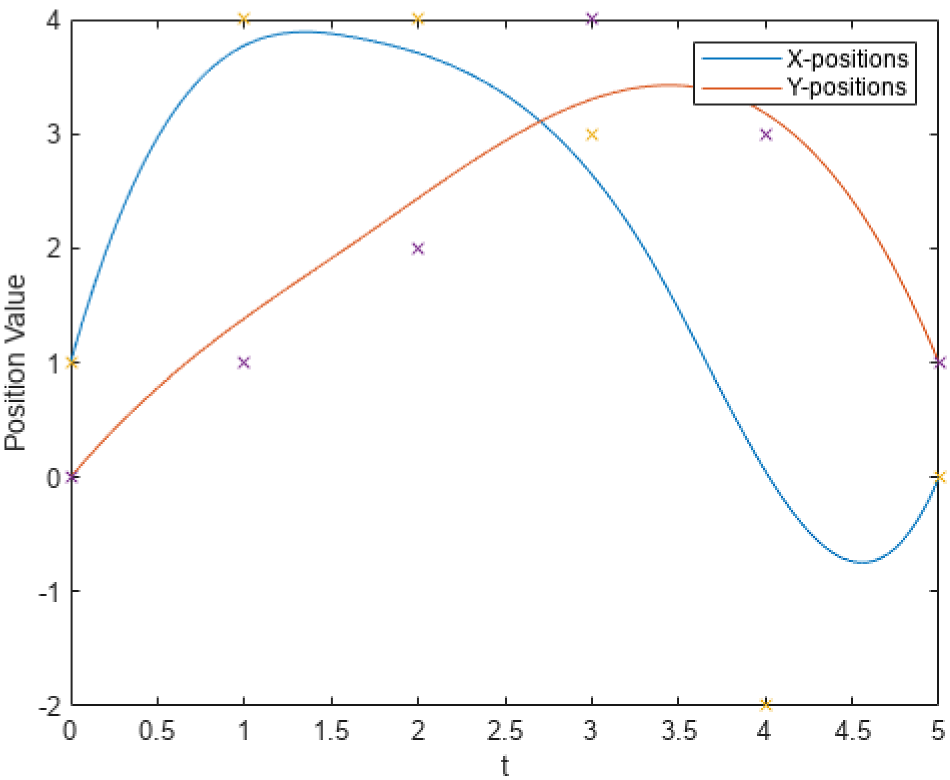
Plot the results. Show the control points and the resulting trajectory inside them.

```
figure
plot(cpts(1,:),cpts(2,:),'xb-')
hold all
plot(q(1,:), q(2,:))
xlabel('X')
ylabel('Y')
hold off
```



Plot the position of each element of the B-spline trajectory. These trajectories are cubic piecewise polynomials parameterized in time.

```
figure
plot(tvec,q)
hold all
plot([0:length(cpts)-1],cpts,'x')
xlabel('t')
ylabel('Position Value')
legend('X-positions','Y-positions')
hold off
```



Interpolate with B-Spline

Create waypoints to interpolate with a B-Spline.

```
wpts1 = [0 1 2.1 8 4 3];
wpts2 = [0 1 1.3 .8 .3 .3];
wpts = [wpts1; wpts2];
L = length(wpts) - 1;
```

Form matrices used to compute interior points of control polygon

```
r = zeros(L+1, size(wpts,1));
A = eye(L+1);
for i= 1:(L-1)
    A(i+1,(i):(i+2)) = [1 4 1];
    r(i+1,:) = 6*wpts(:,i+1)';
end
```

Override end points and choose `r0` and `rL`.

```
A(2,1:3) = [3/2 7/2 1];
A(L,(L-1):(L+1)) = [1 7/2 3/2];

r(1,:) = (wpts(:,1) + (wpts(:,2) - wpts(:,1))/2)';
r(end,:) = (wpts(:,end-1) + (wpts(:,end) - wpts(:,end-1))/2)';
```

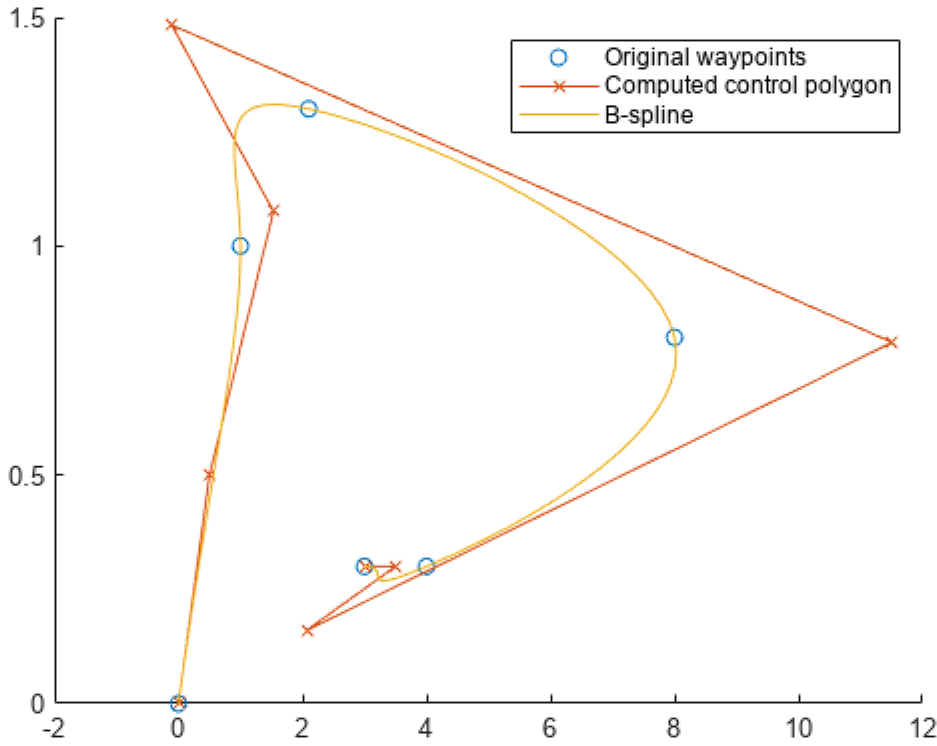
```
dInterior = (A\r)';
```

Construct a complete control polygon and use `bsplinepolytraj` to compute a polynomial with the new control points

```
cpts = [wpts(:,1) dInterior wpts(:,end)];
t = 0:0.01:1;
q = bsplinepolytraj(cpts, [0 1], t);
```

Plot the results. Show the original waypoints, the computed polygon, and the interpolated B-spline.

```
figure;
hold all
plot(wpts1, wpts2, 'o');
plot(cpts(1,:), cpts(2,:), 'x-');
plot(q(1,:), q(2,:));
legend('Original waypoints', 'Computed control polygon', 'B-spline');
```



[1] Farin, Section 9.1

Input Arguments

`controlPoints` — Points for control polygon

n-by-p matrix

Points for control polygon of B-spline trajectory, specified as an *n*-by-*p* matrix, where *n* is the dimension of the trajectory and *p* is the number of control points.

Example: `[1 4 4 3 -2 0; 0 1 2 4 3 1]`

Data Types: `single` | `double`

`tInterval` — Start and end times for trajectory
two-element vector

Start and end times for the trajectory, specified as a two-element vector.

Example: `[0 10]`

Data Types: `single` | `double`

`tSamples` — Time samples for trajectory
vector

Time samples for the trajectory, specified as a vector. The output position, `q`, velocity, `qd`, and accelerations, `qdd`, are sampled at these time intervals.

Example: `0:0.01:10`

Data Types: `single` | `double`

Output Arguments

q — Positions of trajectory
vector

Positions of the trajectory at the given time samples in `tSamples`, returned as a vector.

Data Types: `single` | `double`

qd — Velocities of trajectory
vector

Velocities of the trajectory at the given time samples in `tSamples`, returned as a vector.

Data Types: `single` | `double`

qdd — Accelerations of trajectory
vector

Accelerations of the trajectory at the given time samples in `tSamples`, returned as a vector.

Data Types: `single` | `double`

pp — Piecewise-polynomial
structure

Piecewise-polynomial, returned as a structure that defines the polynomial for each section of the piecewise trajectory. You can build your own piecewise polynomials using `mkpp`, or evaluate the polynomial at specified times using `ppval`. The structure contains the fields:

- `form`: `'pp'`.
- `breaks`: *p*-element vector of times when the piecewise trajectory changes forms. *p* is the number of waypoints.

- `coefs`: $n(p–1)$-by-`order` matrix for the coefficients for the polynomials. $n(p–1)$ is the dimension of the trajectory times the number of `pieces`. Each set of n rows defines the coefficients for the polynomial that described each variable trajectory.
- `pieces`: p–1. The number of breaks minus 1.
- `order`: Degree of the polynomial + 1. For example, cubic polynomials have an order of 4.
- `dim`: n. The dimension of the control point positions.

Version History

Introduced in R2019a

References

[1] Farin, Gerald E. *Curves and Surfaces for Computer Aided Geometric Design: A Practical Guide*. San Diego, CA: Academic Press, 1993.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`contopptraj` | `cubicpolytraj` | `quinticpolytraj` | `rottraj` | `transformtraj` | `trapveltraj`

cart2hom

Convert Cartesian coordinates to homogeneous coordinates

Syntax

```
hom = cart2hom(cart)
```

Description

`hom = cart2hom(cart)` converts a set of points in Cartesian coordinates to homogeneous coordinates.

Examples

Convert 3-D Cartesian Points to Homogeneous Coordinates

```
c = [0.8147 0.1270 0.6324; 0.9058 0.9134 0.0975];
h = cart2hom(c)
```

```
h = 2×4

    0.8147    0.1270    0.6324    1.0000
    0.9058    0.9134    0.0975    1.0000
```

Input Arguments

`cart` — Cartesian coordinates
n-by-*k* matrix

Cartesian coordinates, specified as an *n*-by-*k* matrix, containing *n* points. Each row of `cart` represents a point in *k*-dimensional space. *k* must be greater than or equal to 1.

Example: `[0.8147 0.1270 0.6324; 0.9058 0.9134 0.0975]`

Output Arguments

`hom` — Homogeneous points
n-by-(*k*+1) matrix

Homogeneous points, returned as an *n*-by-(*k*+1) matrix, containing *n* points. *k* must be greater than or equal to 1.

Example: `[0.2785 0.9575 0.1576 0.5; 0.5469 0.9649 0.9706 0.5]`

Version History

Introduced in R2015a

R2023a: `cart2hom` Supports 2-D Cartesian Coordinates

The `cart` argument now accepts 2-D Cartesian coordinates and `cart2hom` outputs 2-D homogeneous points.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also
`hom2cart`

Topics
"Coordinate Transformations in Robotics"

checkCollision

Check if two geometries are in collision

Syntax

```
collisionStatus = checkCollision(geom1,geom2)
[collisionStatus,sepdist,witnesspts] = checkCollision(geom1,geom2)
```

Description

`collisionStatus = checkCollision(geom1,geom2)` returns the collision status between the two convex geometries `geom1` and `geom2`. If the two geometries are in collision at their specified poses, `collisionStatus` is `1`. If the function does not find a collision, `collisionStatus` is `0`.

`[collisionStatus,sepdist,witnesspts] = checkCollision(geom1,geom2)` returns the minimal distance `sepdist` and witness points `witnesspts` of each geometry when the function does not find a collision between the two geometries.

Examples

Check Geometry Collision Status

This example shows how to check the collision status of two collision geometries.

Create a box collision geometry.

```
bx = collisionBox(1,2,3);
```

Create a cylinder collision geometry.

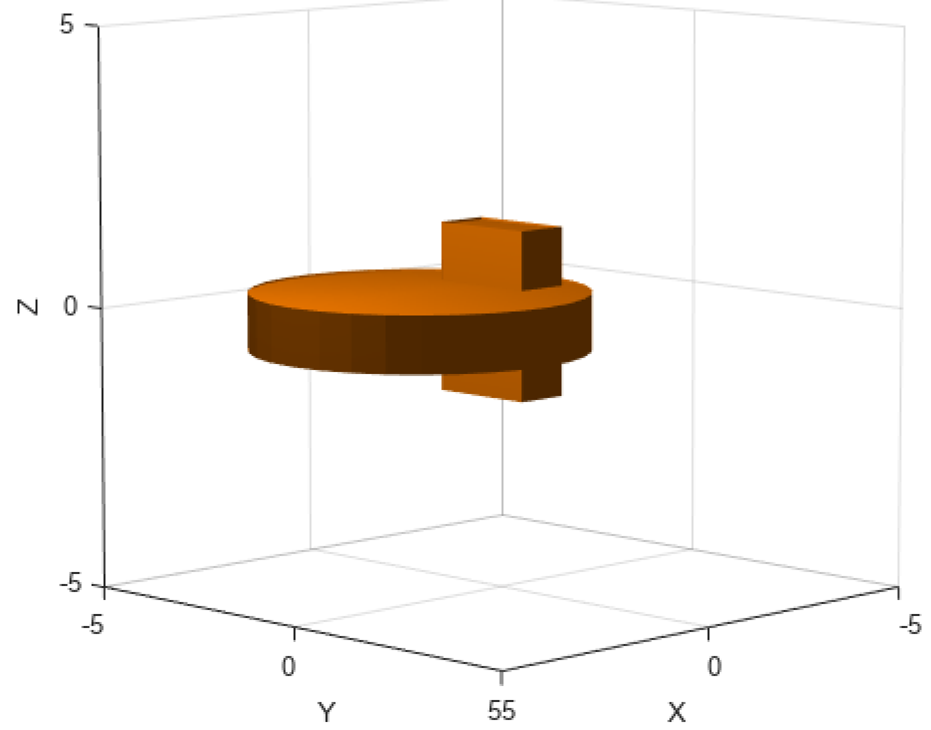
```
cy = collisionCylinder(3,1);
```

Translate the cylinder along the *x*-axis by 2.

```
T = trvec2tform([2 0 0]);
cy.Pose = T;
```

Plot the two geometries.

```
show(cy)
hold on
show(bx)
xlim([-5 5])
ylim([-5 5])
zlim([-5 5])
hold off
```



Check the collision status. Confirm the status is consistent with the plot.

```
[areIntersecting,dist,witnessPoints] = checkCollision(bx,cy)
```

```
areIntersecting = 1

dist = NaN

witnessPoints = 3×2

   NaN   NaN
   NaN   NaN
   NaN   NaN
```

Translate the box along the *x*-axis by 3 and down the *z*-axis by 4. Confirm the box and cylinder are not colliding.

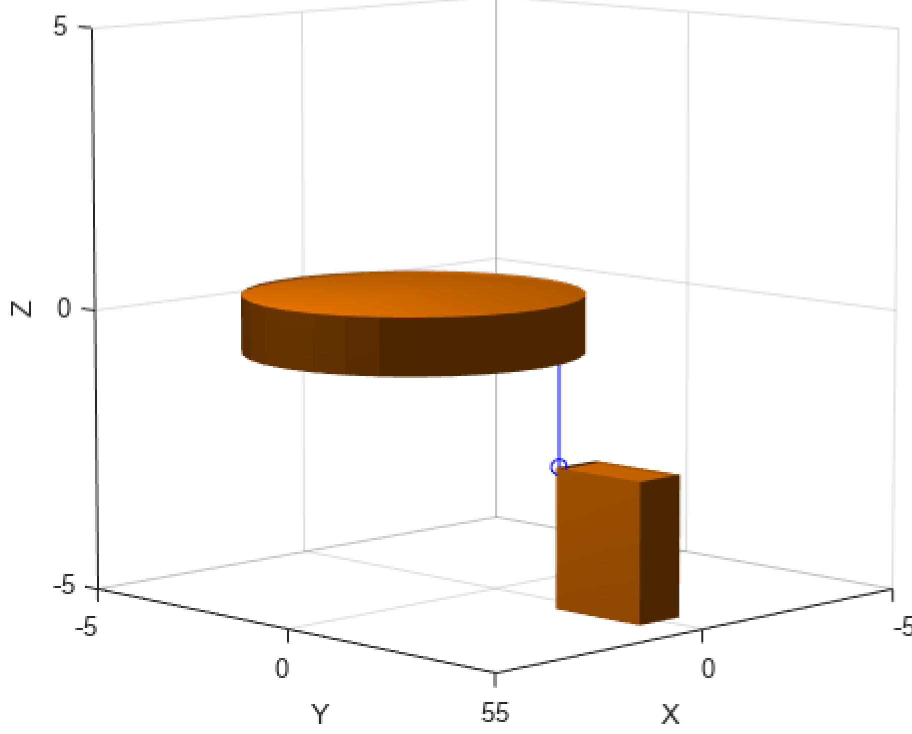
```
T = trvec2tform([0 3 -4]);
bx.Pose = T;
[areIntersecting,dist,witnessPoints] = checkCollision(bx,cy)
```

```
areIntersecting = 0

dist = 2

witnessPoints = 3×2

    0.4286    0.4286
```

```
    2.0000    2.0000
   -2.5000   -0.5000
```

Plot the box, cylinder, and the line segment representing the minimum distance between the two geometries.

```
show(cy)
hold on
show(bx)
wp = witnessPoints;
plot3([wp(1,1) wp(1,2)], [wp(2,1) wp(2,2)], [wp(3,1) wp(3,2)], 'bo-')
xlim([-5 5])
ylim([-5 5])
zlim([-5 5])
hold off
```

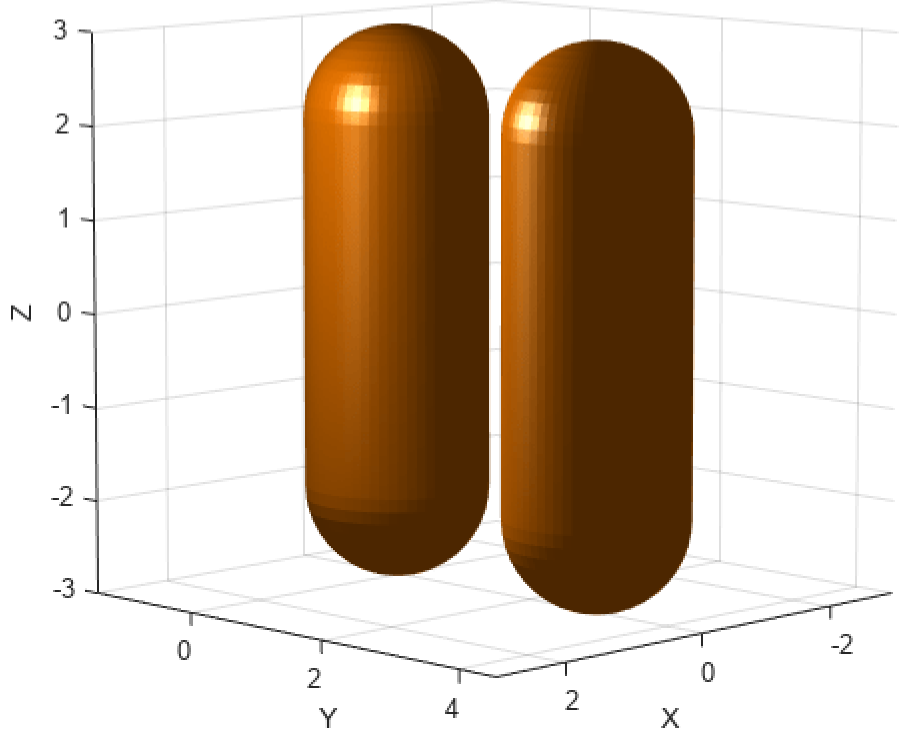


Check Collision Between Collision Capsules

Create two collision capsules. Center one at the origin, and set the pose of the other capsule to 3 meters away from the origin on the *y*-axis. Display the capsules.

```
cc1 = collisionCapsule(1,4);
cc2 = collisionCapsule(1,4);
cc2.Pose = trvec2tform([0 3 0]);
```

```
show(cc1);
hold on
show(cc2);
axis auto
hold off
```



Check for collision between the two collision capsules. Because they are not visually colliding, the function should return real-valued separation distances and witness points. Display the separation distances and witness points.

```
[isColliding,separationDist,witnessPts] = checkCollision(cc1,cc2);
disp("Separation Distance: " + num2str(separationDist))
```

```
Separation Distance: 1
```

```
disp("Capsule 1 Witness Point (X Y Z): " + num2str(witnessPts(:,1)'))
```

```
Capsule 1 Witness Point (X Y Z): 0  1 -2
```

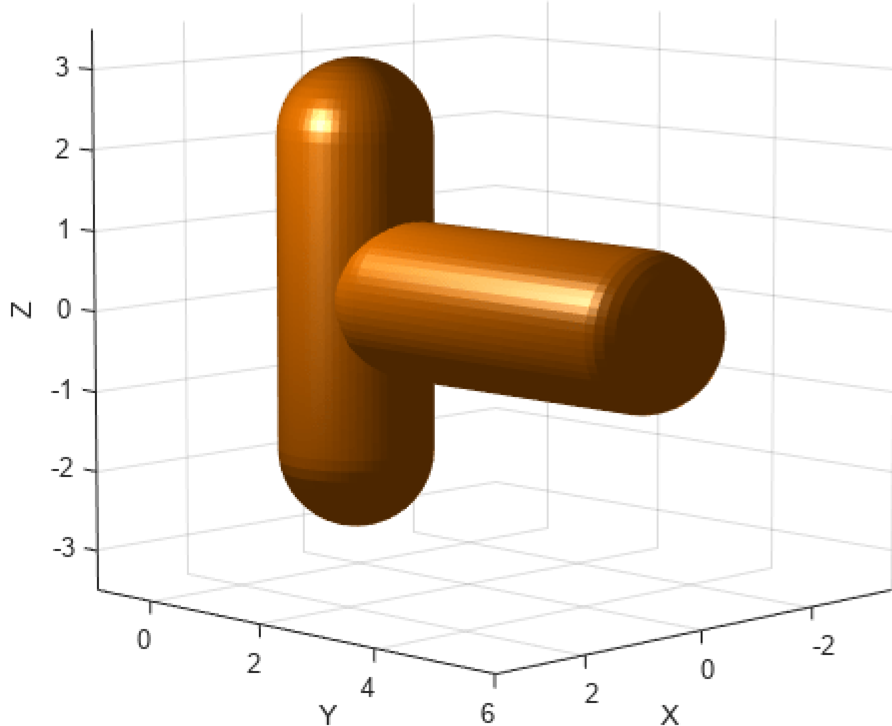
```
disp("Capsule 2 Witness Point (X Y Z): " + num2str(witnessPts(:,2)'))
```

```
Capsule 2 Witness Point (X Y Z): 0  2 -2
```

Rotate the second capsule `90` degrees on the *z*-axis.

```
cc2.Pose(1:3,1:3) = eul2rotm([0 0 pi/2]);
show(cc1);
hold on
```

```
show(cc2);
axis auto
```



Check again for collision between the capsules. Because they are in collision, the function returns the separation distance and witness points as NaN.

```
[isColliding,separationDist,witnessPts] = checkCollision(cc1,cc2);
disp("Separation Distance: " + num2str(separationDist))
```

```
Separation Distance: NaN
```

```
disp("Capsule 1 Witness Point (X Y Z): " + num2str(witnessPts(:,1)'))
```

```
Capsule 1 Witness Point (X Y Z): NaN  NaN  NaN
```

```
disp("Capsule 2 Witness Point (X Y Z): " + num2str(witnessPts(:,2)'))
```

```
Capsule 2 Witness Point (X Y Z): NaN  NaN  NaN
```

Input Arguments

geom1 — First collision geometry
collision geometry object

First collision geometry, specified as one of these collision geometry objects:

- `collisionBox`
- `collisionCapsule`
- `collisionCylinder`
- `collisionMesh`
- `collisionSphere`

`geom2` — Second collision geometry
collision geometry object

Collision geometry, specified as one of these collision geometry objects:

- `collisionBox`
- `collisionCapsule`
- `collisionCylinder`
- `collisionMesh`
- `collisionSphere`

Output Arguments

`collisionStatus` — Collision status
`0` | `1`

Collision status, returned as `0` or `1`. If the two geometries are in collision, `collisionStatus` is `1`. Otherwise, the value is `0`.

Data Types: `double`

`sepdist` — Minimal distance between collision geometries
real number

Minimal distance between the two collision geometries, returned as a real number. The line segment that connects the witness points `witnesspts` determines the minimal distance between the two geometries. When the two geometries are in collision, `sepdist` is `NaN`.

Data Types: `double`

`witnesspts` — Witness points on each geometry
3-by-2 matrix

Witness points on each geometry, returned as a 3-by-2 matrix. Each column is the location of the witness point on the corresponding geometry, `geom1` or `geom2`. The line segment that connects the two witness points has length `septdist`. When the two geometries are in collision, every element of `witnesspts` is `NaN`.

Data Types: `double`

Limitations

- Collision checking results are unreliable when the minimal distance is below 10^{-5} m.

Version History

Introduced in R2019b

References

[1] Gilbert, E.G., D.W. Johnson, and S.S. Keerthi. "*A fast procedure for computing the distance between complex objects in three-dimensional space.*" *IEEE Journal on Robotics and Automation* 4, no. 2 (April 1988): 193–203. https://doi.org/10.1109/56.2083.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`collisionBox` | `collisionCylinder` | `collisionMesh` | `collisionSphere`

classUnderlying

Class of parts within quaternion

Syntax

```
underlyingClass = classUnderlying(quat)
```

Description

`underlyingClass = classUnderlying(quat)` returns the name of the class of the parts of the quaternion `quat`.

Examples

Get Underlying Class of Quaternion

A quaternion is a four-part hyper-complex number used in three-dimensional representations. The four parts of the quaternion are of data type `single` or `double`.

Create two quaternions, one with an underlying data type of `single`, and one with an underlying data type of `double`. Verify the underlying data types by calling `classUnderlying` on the quaternions.

```
qSingle = quaternion(single([1,2,3,4]))
```

```
qSingle = quaternion
     1 + 2i + 3j + 4k
```

```
classUnderlying(qSingle)
```

```
ans =
'single'
```

```
qDouble = quaternion([1,2,3,4])
```

```
qDouble = quaternion
     1 + 2i + 3j + 4k
```

```
classUnderlying(qDouble)
```

```
ans =
'double'
```

You can separate quaternions into their parts using the `parts` function. Verify the parts of each quaternion are the correct data type. Recall that `double` is the default MATLAB® type.

```
[aS,bS,cS,dS] = parts(qSingle)
```

```
aS = single
    1
```

```
bS = single
    2

cS = single
    3

dS = single
    4
```

```
[aD,bD,cD,dD] = parts(qDouble)
```

```
aD = 1

bD = 2

cD = 3

dD = 4
```

Quaternions follow the same implicit casting rules as other data types in MATLAB. That is, a quaternion with underlying data type `single` that is combined with a quaternion with underlying data type `double` results in a quaternion with underlying data type `single`. Multiply `qDouble` and `qSingle` and verify the resulting underlying data type is `single`.

```
q = qDouble*qSingle;
classUnderlying(q)
```

```
ans = 
'single'
```

Input Arguments

`quat` — Quaternion to investigate
scalar | vector | matrix | multi-dimensional array

Quaternion to investigate, specified as a quaternion or array of quaternions.

Data Types: `quaternion`

Output Arguments

`underlyingClass` — Underlying class of quaternion object
`'single'` | `'double'`

Underlying class of quaternion, returned as the character vector `'single'` or `'double'`.

Data Types: `char`

Version History

Introduced in R2018a

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

Functions
`compact` | `parts`

Objects
`quaternion`

compact

Convert quaternion array to *N*-by-4 matrix

Syntax

```
matrix = compact(quat)
```

Description

`matrix = compact(quat)` converts the quaternion array, `quat`, to an *N*-by-4 matrix. The columns are made from the four quaternion parts. The ith row of the matrix corresponds to `quat(i)`.

Examples

Convert Quaternion Array to Compact Representation of Parts

Create a scalar quaternion with random parts. Convert the parts to a 1-by-4 vector using `compact`.

```
randomParts = randn(1,4)
```

```
randomParts = 1×4

    0.5377    1.8339   -2.2588    0.8622
```

```
quat = quaternion(randomParts)
```

```
quat = quaternion
     0.53767 +  1.8339i -  2.2588j + 0.86217k
```

```
quatParts = compact(quat)
```

```
quatParts = 1×4

    0.5377    1.8339   -2.2588    0.8622
```

Create a 2-by-2 array of quaternions, then convert the representation to a matrix of quaternion parts. The output rows correspond to the linear indices of the quaternion array.

```
quatArray = [quaternion([1:4;5:8]),quaternion([9:12;13:16])]
```

```
quatArray = 2x2 quaternion array
      1 +  2i +  3j +  4k      9 + 10i + 11j + 12k
      5 +  6i +  7j +  8k     13 + 14i + 15j + 16k
```

```
quatArrayParts = compact(quatArray)
```

```
quatArrayParts = 4×4
```

```
     1     2     3     4
     5     6     7     8
     9    10    11    12
    13    14    15    16
```

Input Arguments

quat — Quaternion to convert
scalar | vector | matrix | multidimensional array

Quaternion to convert, specified as scalar, vector, matrix, or multidimensional array of quaternions.

Data Types: `quaternion`

Output Arguments

matrix — Quaternion in matrix form
N-by-4 matrix

Quaternion in matrix form, returned as an *N*-by-4 matrix, where *N* = `numel(quat)`.

Data Types: `single` | `double`

Version History

Introduced in R2018a

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

Functions
`parts` | `classUnderlying`

Objects
`quaternion`

conj

Complex conjugate of quaternion

Syntax

```
quatConjugate = conj(quat)
```

Description

`quatConjugate = conj(quat)` returns the complex conjugate of the quaternion, `quat`.

If $q = a + b\text{i} + c\text{j} + d\text{k}$, the complex conjugate of q is $q^* = a - b\text{i} - c\text{j} - d\text{k}$. Considered as a rotation operator, the conjugate performs the opposite rotation. For example,

```
q = quaternion(deg2rad([16 45 30]),'rotvec');
a = q*conj(q);
rotatepoint(a,[0,1,0])
```

```
ans =

     0     1     0
```

Examples

Complex Conjugate of Quaternion

Create a quaternion scalar and get the complex conjugate.

```
q = normalize(quaternion([0.9 0.3 0.3 0.25]))
```

```
q = quaternion
     0.87727 + 0.29242i + 0.29242j + 0.24369k
```

```
qConj = conj(q)
```

```
qConj = quaternion
     0.87727 - 0.29242i - 0.29242j - 0.24369k
```

Verify that a quaternion multiplied by its conjugate returns a quaternion one.

```
q*qConj
```

```
ans = quaternion
     1 + 0i + 0j + 0k
```

Input Arguments

quat — Quaternion
scalar | vector | matrix | multidimensional array

Quaternion to conjugate, specified as a scalar, vector, matrix, or array of quaternions.

Data Types: quaternion

Output Arguments

quatConjugate — Quaternion conjugate
scalar | vector | matrix | multidimensional array

Quaternion conjugate, returned as a quaternion or array of quaternions the same size as quat.

Data Types: quaternion

Version History

Introduced in R2018a

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

Functions
norm | .*,times

Objects
quaternion

contopptraj

Generate trajectory subject to kinematic constraints

Syntax

```
[q,qd,qdd,t] = contopptraj(waypoints,vellim,accellim)
[___] = contopptraj(polypath,vellim,accellim)
[___] = contopptraj(___,NumSamples=N)
[___,solninfo] = contopptraj(___)
```

Description

`[q,qd,qdd,t] = contopptraj(waypoints,vellim,accellim)` generates a trajectory by fitting a path to a set of waypoints `waypoints`. The function returns a time-optimal trajectory along the path for position `q`, velocity `qd`, and acceleration `qdd` at sample times `t`, while constrained by the velocity `vellim` and acceleration limits `accellim`. This function requires Optimization Toolbox™.

`[___] = contopptraj(polypath,vellim,accellim)` generates a trajectory along the specified polynomial path `polypath` using all output arguments from the previous syntax.

`[___] = contopptraj(___,NumSamples=N)` specifies the number of samples to use when generating the trajectory, in addition to any combination of arguments from previous syntaxes.

`[___,solninfo] = contopptraj(___)` outputs solution information `solninfo` with diagnostic information associated with the output trajectory, in addition to any combination of arguments from previous syntaxes.

Examples

Create Kinematically Constrained Trajectory

Create waypoints, velocity limits, and acceleration limits.

```
waypoints = [0 5 10 -10; 0 8 -10 5; 0 -10 15 5];
velLimits = [-1 1; -2 2; -3 3];
accelLimits = [-1 1; -2 2; -3 3];
```

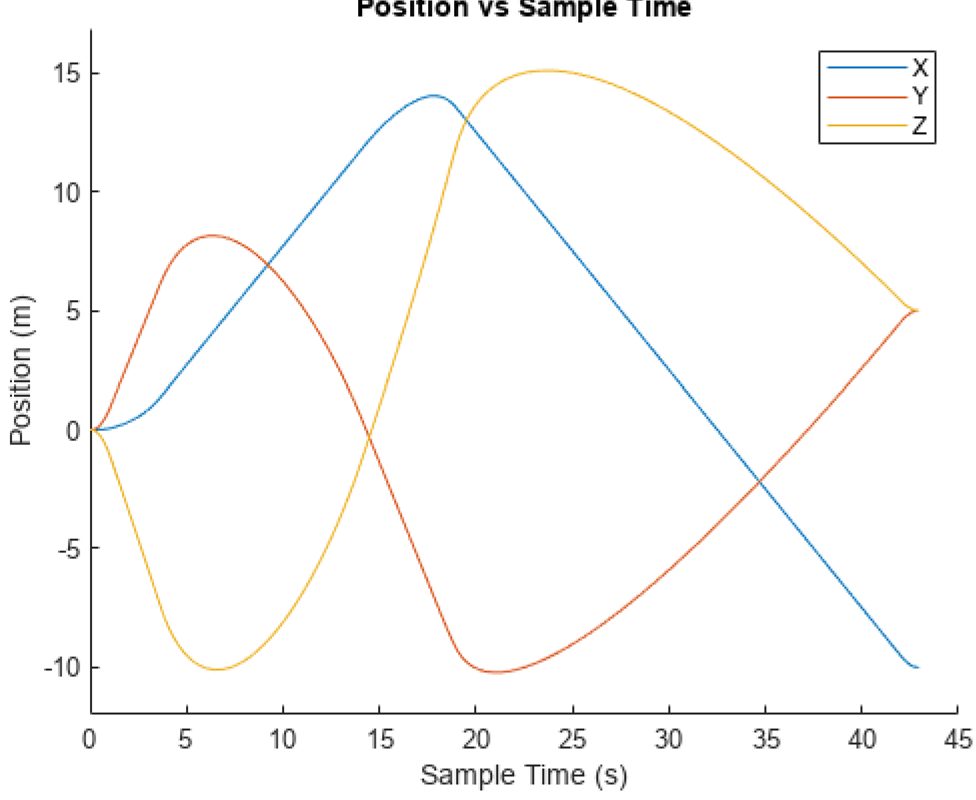
Generate the position, velocity, acceleration, and time vector of the trajectory with 200 samples.

```
[q,qd,qdd,t] = contopptraj(waypoints,velLimits,accelLimits,NumSamples=200);
```

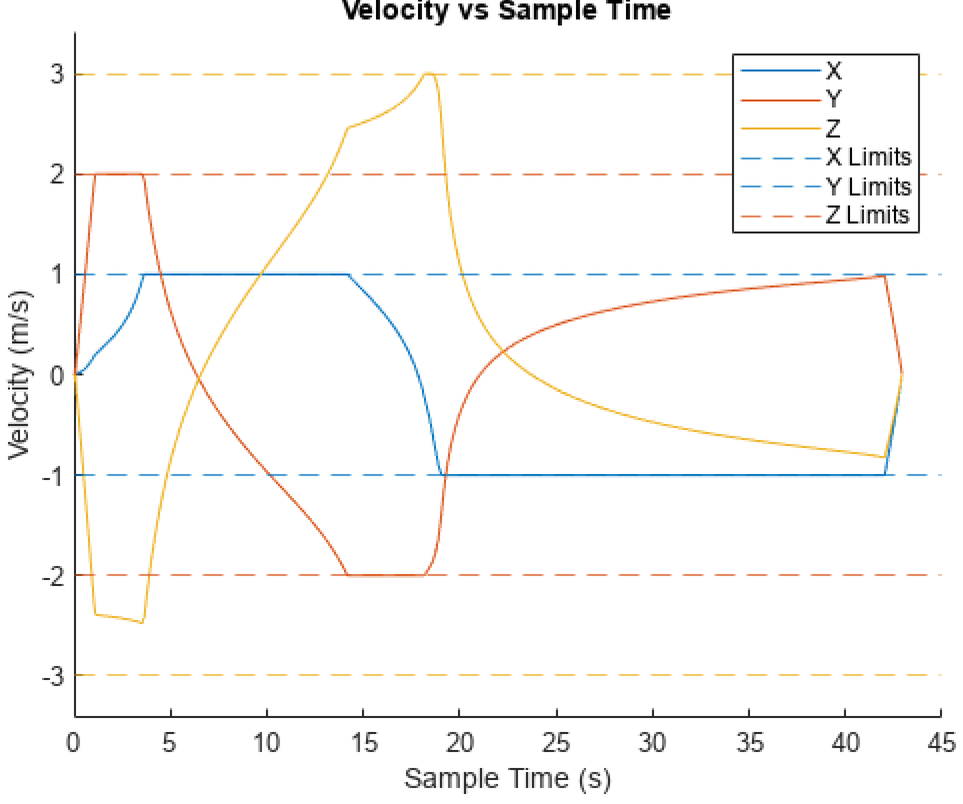
Plot the positions, velocities, and accelerations against the sample times `t`.

```
figure
title("Position vs Sample Time")
ylim("padded")
hold on
q = real(q);
qd = real(qd);
qdd = real(qdd);
```

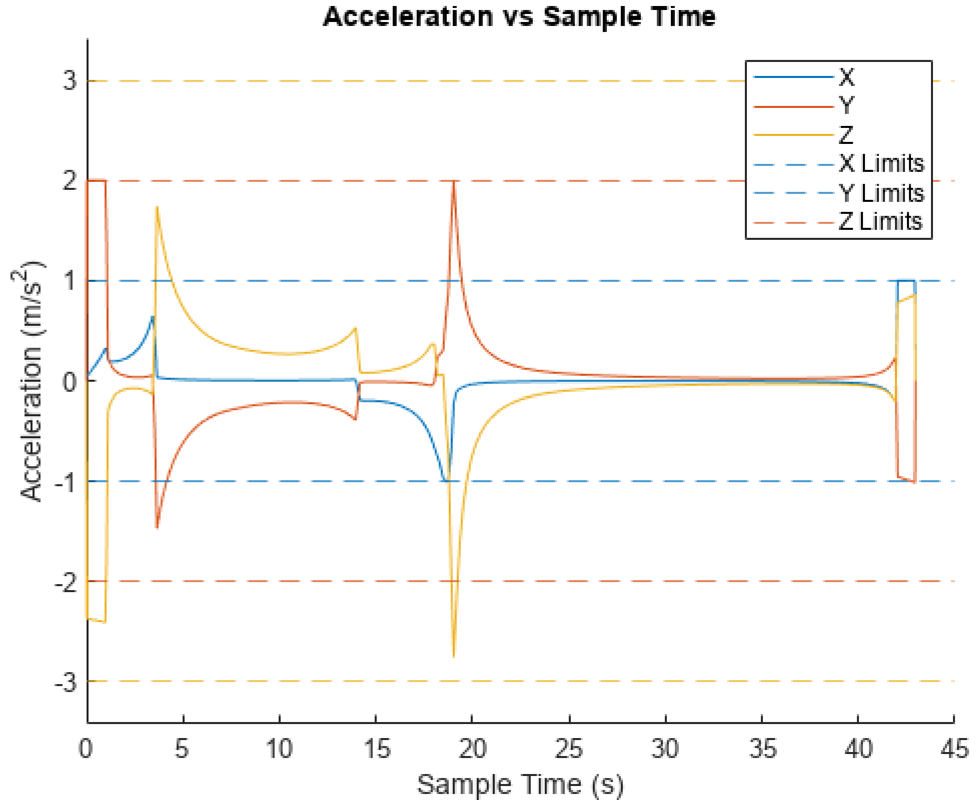
```
t = real(t);
plot(t,q(1,:)) % X Position
plot(t,q(2,:)) % Y Position
plot(t,q(3,:)) % Z Position
xlabel("Sample Time (s)")
ylabel("Position (m)")
legend(["X","Y","Z"])
hold off
```




```
helperPlotConstrainedTrajectory(qd,t,velLimits,"Velocity")
ylabel("Velocity (m/s)")
```




```
helperPlotConstrainedTrajectory(qdd,t,accelLimits,"Acceleration")
ylabel("Acceleration (m/s^2)")
```



Input Arguments

`waypoints` — Trajectory waypoints for path fitting
n-by-*p* matrix

Trajectory waypoints for path fitting, specified as an *m*-by-*n* matrix. *n* is the number of elements in the state space, and *p* is the number of distinct waypoints.

Example: `[1 1 2 3; 2 5 4 5; 5 7 6 8]` is a set of four waypoints with three state space elements each.

`vellim` — Minimum and maximum velocity limits of trajectory
n-by-2 matrix

Minimum and maximum velocity limits of the trajectory, specified as an *n*-by-2 matrix, in units per second. *n* is the number of elements in the state space, and each row is of the form [*minimumLimit maximumLimit*], specifying the minimum and maximum velocity for each state space element. To generate a trajectory without velocity limits, specify this argument as an empty array.

Example: `[-1 1; -2 5; -5 8]` is a set of velocity limits for three state space elements.

`accellim` — Minimum and maximum acceleration limits of trajectory
n-by-2 matrix

Minimum and maximum acceleration limits of the trajectory, specified as an *n*-by-2 matrix, in units per seconds squared. *n* is the number of elements in the state space, and each row is of the form

[*minimumLimit maximumLimit*], specifying the minimum and maximum velocity for each state space element. To generate a trajectory without acceleration limits, specify this argument as an empty array.

Example: `[-1 1; -2 5; -5 8]` is a set of acceleration limits for three state space elements.

`polypath` — Piecewise polynomial path for trajectory generation
structure

Piecewise polynomial path for trajectory generation, specified as a structure such as the output returned from the `mkpp` or `spline` functions. For more information on the fields of this structure, see the `pp` argument of the `mkpp` the function.

Data Types: `struct`

`N` — Number of samples for trajectory generation
`100` (default) | positive integer

Number of samples for trajectory generation, specified as a positive integer.

Example: `NumSamples=200`

Output Arguments

q — Trajectory positions
n-by-*m* matrix

Trajectory positions, returned as a *n*-by-*m* matrix. *n* is the number of elements in the state space, and *m* is the number of samples in the trajectory.

qd — Trajectory velocities
n-by-*m* matrix

Trajectory velocities, returned as a *n*-by-*m* matrix. *n* is the number of elements in the state space, and *m* is the number of samples in the trajectory.

qdd — Trajectory accelerations
n-by-*m* matrix

Trajectory accelerations, returned as a *n*-by-*m* matrix. *n* is the number of elements in the state space, and *m* is the number of samples in the trajectory.

`t` — Sample times
m-element vector

Sample times, returned as a *m*-element vector, in seconds. *m* is the number of samples in the trajectory.

`solninfo` — Solution information
structure

Solution information, specified as a structure containing these fields:

- `ExitFlag` — Number indicating whether `contopptraj` could find a solution. A value of `1` indicates the function successfully found a solution, and a value of `0` indicates that the function was not able to find a feasible solution.

Version History

Introduced in R2022b

References

[1] Pham, Hung, and Quang-Cuong Pham. "A New Approach to Time-Optimal Path Parameterization Based on Reachability Analysis." *IEEE Transactions on Robotics*, 34, no. 3 (June 2018): 645–59. https://doi.org/10.1109/TRO.2018.2819195.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

Functions

`bsplinepolytraj` | `cubicpolytraj` | `minjerkpolytraj` | `minsnappolytraj` | `quinticpolytraj` | `rottraj` | `transformtraj` | `trapveltraj`

createCustomRobotSensorTemplate

Create sample implementation for robot custom sensor interface

Syntax

`createCustomRobotSensorTemplate`

Description

`createCustomRobotSensorTemplate` creates a sample template implementation for robot custom sensor that inherits from the `robotics.SensorAdaptor` class. This function opens a new file in MATLAB editor.

Examples

Simulate Ultrasonic Sensors Mounted on Mobile Robots

This example focuses on creating and mounting an ultrasonic sensor on a mobile robot in a `robotScenario`. The `ultrasonicDetectionGenerator` from the Automated Driving Toolbox cannot be used directly with `robotScenario`. We will be implementing a custom sensor adaptor for the `ultrasonicDetectionGenerator` that makes it compatible with `robotScenario`. The sensor will be used to position a mobile robot correctly at a charging station.

Create Custom Sensor Adaptor

Use the `createCustomRobotSensorTemplate` function to generate a template sensor and update it to adapt an `ultrasonicDetectionGenerator` object for usage in Robot scenario.

```
createCustomRobotSensorTemplate
```

This example provides the adaptor class `CustomUltrasonicSensor`, which can be viewed using the following command.

```
edit CustomUltrasonicSensor.m
```

Use the Sensor Adaptor in Robot Scenario Simulation

Create a `robotScenario` object with a sample rate of 10.

```
sampleRate = 10;
scenario = robotScenario(UpdateRate=sampleRate);
```

Add a plane mesh to show the warehouse floor.

```
addMesh(scenario,"Plane",Position=[5 0 0],Size=[20 12],Color=[0.7 0.7 0.7]);
```

Create a `waypointTrajectory` that traverses a set of waypoints to the charging station and use the `lookupPose` method of the `waypointTrajectory` object to fetch the pose of the robot along the trajectory.

```
startPosition = [-3 -3];
chargingPosition = [13 0];
```

```
wPts = [[startPosition 0.1]; ...
    5 0 0.1; ...
    10 0 0.1; ...
    13.75 0 0.1]; %Charging station

toa = [0 4 7 10];
traj = waypointTrajectory(Waypoints=wPts,...
    TimeOfArrival=toa, ReferenceFrame='ENU', ...
    SampleRate=sampleRate);
[pos, orient, vel, acc, angvel] = traj.lookupPose(0:1/sampleRate:10);
```

Add a `robotPlatform` to the scene for our mobile robot. Load the Clearpath Husky model for the `rigidBodyTree` of the `robotPlatform`. Also add cuboid meshes to denote obstacles in the scene. Add a 1-by-1 plane to denote where the charging station is.

```
robot = robotPlatform("rst", scenario,...
    RigidBodyTree=loadrobot("clearpathHusky"), ...
    InitialBasePosition=pos(1,:), InitialBaseOrientation=compact(orient(1)));

addMesh(scenario,"Box",Position=[3  5 2],Size=[4 2 4],Color=[1 0.5 0.25],IsBinaryOccupied=true);
addMesh(scenario,"Box",Position=[3 -5 2],Size=[4 2 4],Color=[1 0.5 0.25],IsBinaryOccupied=true);
addMesh(scenario,"Box",Position=[7  5 2],Size=[4 2 4],Color=[1 0.5 0.25],IsBinaryOccupied=true);
addMesh(scenario,"Box",Position=[7 -5 2],Size=[4 2 4],Color=[1 0.5 0.25],IsBinaryOccupied=true);
addMesh(scenario,"Box",Position=[-3  -5 0.5],Size=[1 1 1],Color=[0.1 0.1 0.1]);

% Plane to denote Charging station location
addMesh(scenario,"Plane",Position=[13 0 .05],Size=[1 1],Color=[0 1 0]);
```

Create the charging station using a `robotPlatform` object. The `robotPlatform` allows us to fetch the transform between the object and the sensor for use in the custom sensor read method. Here, the charging station can be modeled using a cuboid. The robot has to reach within 5cm of the surface of the charging station to start charging.

```
chargeStation = robotPlatform("chargeStation", scenario,InitialBasePosition=[13.75 0 0]);
chargeStation.updateMesh("Cuboid",Size=[0.5 1 1], Color=[0 0.8 0]);
```

The ultrasonic sensor model requires inputs of the profile of the obstacles to be detected. The profile structure includes information about the dimensions of the obstacle.

```
chargingStationProfile = struct("Length", 0.5, "Width", 1, "Height", 1, 'OriginOffset', [0 0 0])
```

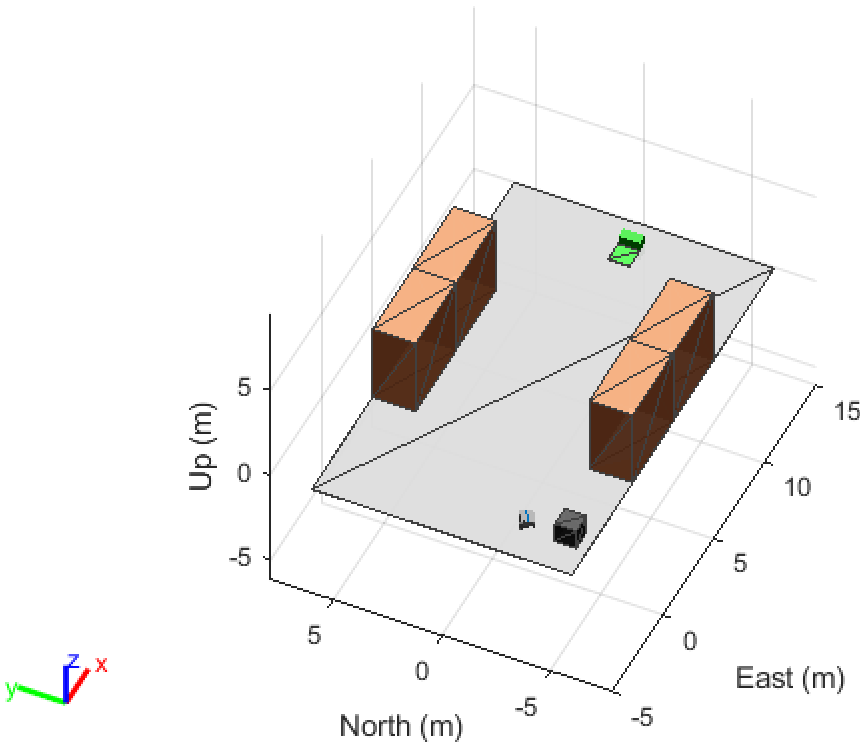
Create the ultrasonic sensor using the `ultrasonicDetectionGenerator` object and set its mounting location to `[0 0 0]`, detection range to `[0.03 0.04 5]` and field of view to `[70, 35]`. Also pass in the profile of the charging station that was created earlier.

```
ultraSonicSensorModel = ultrasonicDetectionGenerator(MountingLocation=[0 0 0], ...
    DetectionRange=[0.03 0.04 5], ...
    FieldOfView=[70, 35], ...
    Profiles=chargingStationProfile);
```

Create a `robotSensor` object that uses the custom sensor adaptor `CustomUltrasonicSensor`. The adaptor uses the ultrasonic sensor model created above. The mounting location will be at the front of the robot.

```
ult = robotSensor("UltraSonic", robot, ...
    CustomUltrasonicSensor(ultraSonicSensorModel), ...
    MountingLocation=[0.5 0 0.05]);
```

```
figure(1);
ax = show3D(scenario);
view(-65,45)
light
grid on
```

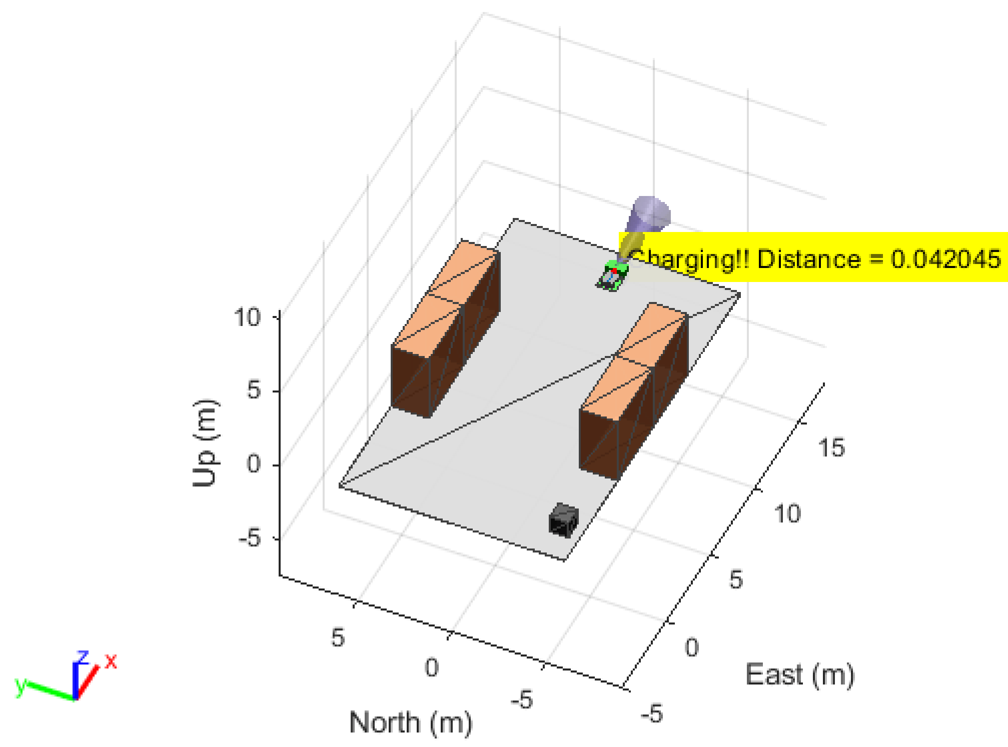


In this scene, the mobile robot will follow the trajectory to the charging station. When the ultrasonic sensor comes within a range of 20cm of the charging station, then mobile robot advance at a slower rate of 1cm per frame towards the charging station. When the robot is within 5cm of the surface of the charging station, it stops and the charging starts. The simulation ends when the charging starts.

```
isCharging = false;
i = 1;

setup(scenario);

while ~isCharging
    [isUpdated, t, det, isValid] = read(ult);

    figure(1);
    show3D(scenario);
    view(-65,45)
    light
    grid on

    % Read the motion vector of the robot from the platform ground truth
```

```
% This motion vector will be used only for plotting graphic elements
pose = robot.read();
rotAngle = quat2eul(pose(10:13));
hold on

if ~isempty(det)

    % Distance to object
    distance = det{1}.Measurement;

    % Plot a red shpere where the ultrasonic sensor detects an object
    exampleHelperPlotDetectionPoint(scenario, ...
        det{1}.ObjectAttributes{1}.PointOnTarget, ...
        ult.Name, ...
        pose);

    displayText = ['Distance = ',num2str(distance)];
else
    distance = inf;
    displayText = 'No object detected!';
end

% Plot a cone to represent the field of view and range of the ultrasonic sensor
exampleHelperPlotFOVCylinder(pose, ultraSonicSensorModel.DetectionRange(3));
hold off

if distance <= 0.2
    % Advance in steps of 1cm when the robot is within 20cm of the charging station
    currentMotion = lastMotion;
    currentMotion(1) = currentMotion(1) + 0.01;

    move(robot,"base",currentMotion);
    lastMotion = currentMotion;
    displayText = ['Detected Charger! Distance = ',num2str(distance)];
    if distance <= 0.05
        % The robot is charging when it is within 5cm of the charging station
        displayText = ['Charging!! Distance = ',num2str(distance)];
        isCharging = true;
    end
else
    % Follow the waypointTrajectory to the vicinity of the charging station
    if i<=length(pos)
        motion = [pos(i,:), vel(i,:), acc(i,:), ...
            compact(orient(i)), angvel(i,:)];
        move(robot,"base",motion);
        lastMotion = motion;
        i=i+1;
    end
end

% Display the distance to the charging station detected by the ultrasonic sensor
t = text(15, 0, displayText, "BackgroundColor",'yellow');
t(1).Color = 'black';
t(1).FontSize = 10;

advance(scenario);
```

```
    updateSensors(scenario);
end
```



Version History

Introduced in R2022b

See Also

`robotics.SensorAdaptor`

ctranspose, '

Complex conjugate transpose of quaternion array

Syntax

```
quatTransposed = quat'
```

Description

`quatTransposed = quat'` returns the complex conjugate transpose of the quaternion, `quat`.

Examples

Vector Complex Conjugate Transpose

Create a vector of quaternions and compute its complex conjugate transpose.

```
quat = quaternion(randn(4,4))
```

```
quat = 4x1 quaternion array
      0.53767 +  0.31877i +   3.5784j +   0.7254k
       1.8339 -   1.3077i +   2.7694j - 0.063055k
      -2.2588 -  0.43359i -   1.3499j +  0.71474k
      0.86217 +  0.34262i +   3.0349j -  0.20497k
```

```
quatTransposed = quat'
```

```
quatTransposed = 1x4 quaternion array
      0.53767 -  0.31877i -   3.5784j -   0.7254k       1.8339 +   1.3077i -   2.7694j + 0.06305
```

Matrix Complex Conjugate Transpose

Create a matrix of quaternions and compute its complex conjugate transpose.

```
quat = [quaternion(randn(2,4)),quaternion(randn(2,4))]
```

```
quat = 2x2 quaternion array
      0.53767 -   2.2588i +  0.31877j -  0.43359k       3.5784 -   1.3499i +   0.7254j +  0.7147
       1.8339 +  0.86217i -   1.3077j +  0.34262k       2.7694 +   3.0349i - 0.063055j -  0.2049
```

```
quatTransposed = quat'
```

```
quatTransposed = 2x2 quaternion array
      0.53767 +   2.2588i -  0.31877j +  0.43359k       1.8339 -  0.86217i +   1.3077j -  0.3426
       3.5784 +   1.3499i -   0.7254j -  0.71474k       2.7694 -   3.0349i + 0.063055j +  0.2049
```

Input Arguments

`quat` — Quaternion to transpose
scalar | vector | matrix

Quaternion to transpose, specified as a vector or matrix or quaternions. The complex conjugate transpose is defined for 1-D and 2-D arrays.

Data Types: `quaternion`

Output Arguments

`quatTransposed` — Conjugate transposed quaternion
scalar | vector | matrix

Conjugate transposed quaternion, returned as an *N*-by-*M* array, where `quat` was specified as an *M*-by-*N* array.

Data Types: `quaternion`

Version History

Introduced in R2018a

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

Functions
`transpose, '`

Objects
`quaternion`

cubicpolytraj

Generate third-order polynomial trajectories

Syntax

```
[q,qd,qdd,pp] = cubicpolytraj(wayPoints,timePoints,tSamples)
[q,qd,qdd,pp] = cubicpolytraj( ___ ,Name,Value)
```

Description

`[q,qd,qdd,pp] = cubicpolytraj(wayPoints,timePoints,tSamples)` generates a third-order polynomial that achieves a given set of input waypoints with corresponding time points. The function outputs positions, velocities, and accelerations at the given time samples, `tSamples`. The function also returns the piecewise polynomial `pp` form of the polynomial trajectory with respect to time.

`[q,qd,qdd,pp] = cubicpolytraj( ___ ,Name,Value)` specifies additional parameters as `Name,Value` pair arguments using any combination of the previous syntaxes.

Examples

Compute Cubic Trajectory for 2-D Planar Motion

Use the `cubicpolytraj` function with a given set of 2-D *xy* waypoints. Time points for the waypoints are also given.

```
wpts = [1 4 4 3 -2 0; 0 1 2 4 3 1];
tpts = 0:5;
```

Specify a time vector for sampling the trajectory. Sample at a smaller interval than the specified time points.

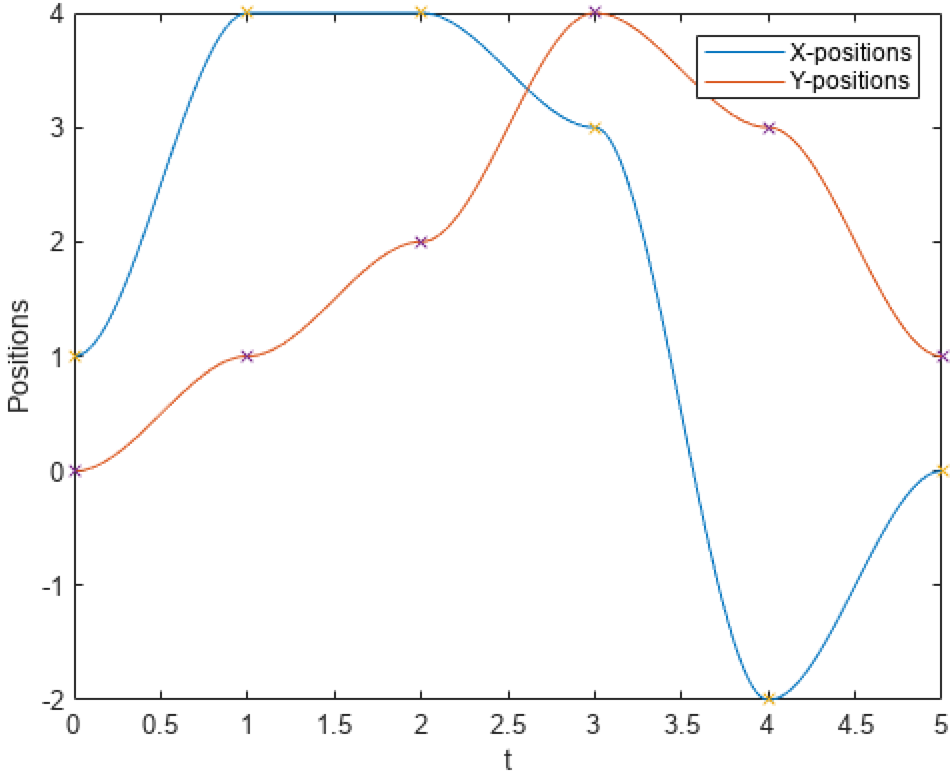
```
tvec = 0:0.01:5;
```

Compute the cubic trajectory. The function outputs the trajectory positions (`q`), velocity (`qd`), acceleration (`qdd`), and polynomial coefficients (`pp`) of the cubic polynomial.

```
[q, qd, qdd, pp] = cubicpolytraj(wpts, tpts, tvec);
```

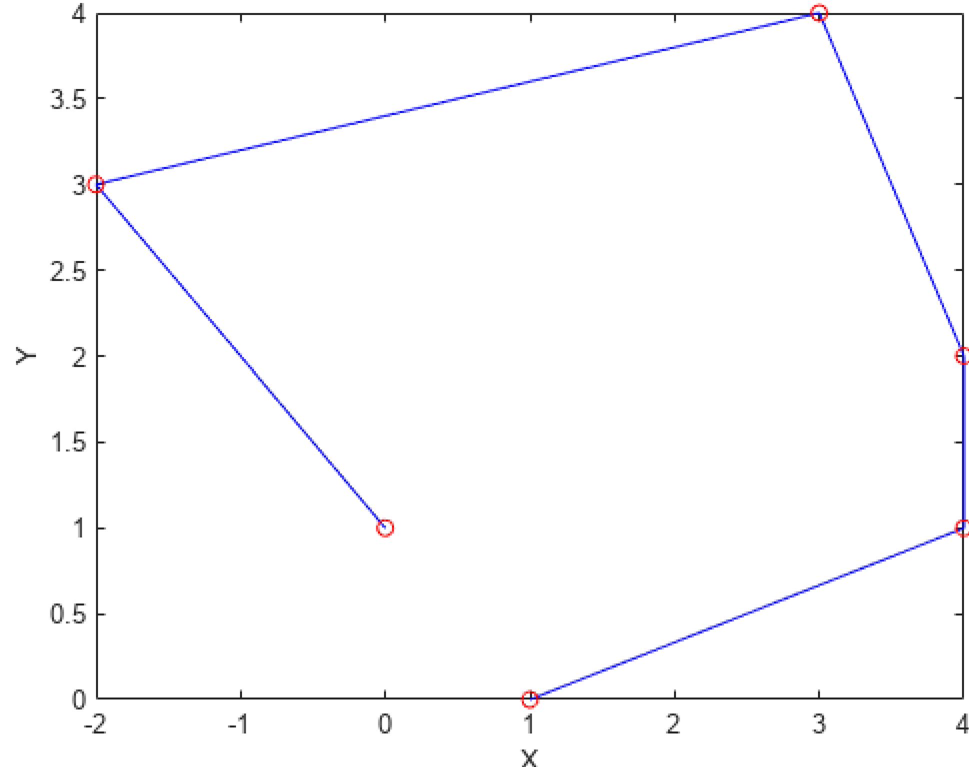
Plot the cubic trajectories for the *x*- and *y*-positions. Compare the trajectory with each waypoint.

```
plot(tvec, q)
hold all
plot(tpts, wpts, 'x')
xlabel('t')
ylabel('Positions')
legend('X-positions','Y-positions')
hold off
```



You can also verify the actual positions in the 2-D plane. Plot the separate rows of the q vector and the waypoints as *x*- and *y* -positions.

```
figure
plot(q(1,:),q(2,:),'-b',wpts(1,:),wpts(2,:),'or')
xlabel('X')
ylabel('Y')
```



Input Arguments

`wayPoints` — Waypoints for trajectory

n-by-*p* matrix

Points for waypoints of trajectory, specified as an *n*-by-*p* matrix, where *n* is the dimension of the trajectory and *p* is the number of waypoints.

Example: `[1 4 4 3 -2 0; 0 1 2 4 3 1]`

Data Types: `single` | `double`

`timePoints` — Time points for waypoints of trajectory

p-element vector

Time points for waypoints of trajectory, specified as a *p*-element vector.

Example: `[0 2 4 5 8 10]`

Data Types: `single` | `double`

`tSamples` — Time samples for trajectory

m-element vector

Time samples for the trajectory, specified as an *m*-element vector. The output position, `q`, velocity, `qd`, and accelerations, `qdd`, are sampled at these time intervals.

Example: `0:0.01:10`

Data Types: `single` | `double`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `'VelocityBoundaryCondition',[1 0 -1 -1 0 0; 1 1 1 -1 -1 -1]`

`VelocityBoundaryCondition` — Velocity boundary conditions for each waypoint
`zeroes(n,p)` (default) | *n*-by-*p* matrix

Velocity boundary conditions for each waypoint, specified as the comma-separated pair consisting of `'VelocityBoundaryCondition'` and an *n*-by-*p* matrix. Each row corresponds to the velocity at all *p* waypoints for the respective variable in the trajectory.

Example: `[1 0 -1 -1 0 0; 1 1 1 -1 -1 -1]`

Data Types: `single` | `double`

Output Arguments

q — Positions of trajectory
m-element vector

Positions of the trajectory at the given time samples in `tSamples`, returned as an *m*-element vector, where *m* is the length of `tSamples`.

Data Types: `single` | `double`

qd — Velocities of trajectory
vector

Velocities of the trajectory at the given time samples in `tSamples`, returned as a vector.

Data Types: `single` | `double`

qdd — Accelerations of trajectory
vector

Accelerations of the trajectory at the given time samples in `tSamples`, returned as a vector.

Data Types: `single` | `double`

pp — Piecewise-polynomial
structure

Piecewise-polynomial, returned as a structure that defines the polynomial for each section of the piecewise trajectory. You can build your own piecewise polynomials using `mkpp`, or evaluate the polynomial at specified times using `ppval`. The structure contains the fields:

- `form`: `'pp'`.

- `breaks`: *p*-element vector of times when the piecewise trajectory changes forms. *p* is the number of waypoints.
- `coefs`: *n*(*p*–1)-by-`order` matrix for the coefficients for the polynomials. *n*(*p*–1) is the dimension of the trajectory times the number of `pieces`. Each set of *n* rows defines the coefficients for the polynomial that described each variable trajectory.
- `pieces`: *p*–1. The number of breaks minus 1.
- `order`: Degree of the polynomial + 1. For example, cubic polynomials have an order of 4.
- `dim`: *n*. The dimension of the control point positions.

Version History

Introduced in R2019a

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`bsplinepolytraj` | `contopptraj` | `quinticpolytraj` | `rottraj` | `transformtraj` | `trapveltraj`

dist

Angular distance in radians

Syntax

```
distance = dist(quatA,quatB)
```

Description

`distance = dist(quatA,quatB)` returns the angular distance in radians between two quaternions, `quatA` and `quatB`.

Examples

Calculate Quaternion Distance

Calculate the quaternion distance between a single quaternion and each element of a vector of quaternions. Define the quaternions using Euler angles.

```
q = quaternion([0,0,0],'eulerd','zyx','frame')
```

```
q = quaternion
     1 + 0i + 0j + 0k
```

```
qArray = quaternion([0,45,0;0,90,0;0,180,0;0,-90,0;0,-45,0],'eulerd','zyx','frame')
```

```
qArray = 5x1 quaternion array
       0.92388 +         0i +   0.38268j +         0k
       0.70711 +         0i +   0.70711j +         0k
    6.1232e-17 +         0i +         1j +         0k
       0.70711 +         0i -   0.70711j +         0k
       0.92388 +         0i -   0.38268j +         0k
```

```
quaternionDistance = rad2deg(dist(q,qArray))
```

```
quaternionDistance = 5×1

   45.0000
   90.0000
  180.0000
   90.0000
   45.0000
```

If both arguments to `dist` are vectors, the quaternion distance is calculated between corresponding elements. Calculate the quaternion distance between two quaternion vectors.

```
angles1 = [30,0,15; ...
           30,5,15; ...
```

```
            30,10,15; ...
            30,15,15];
angles2 = [30,6,15; ...
            31,11,15; ...
            30,16,14; ...
            30.5,21,15.5];

qVector1 = quaternion(angles1,'eulerd','zyx','frame');
qVector2 = quaternion(angles2,'eulerd','zyx','frame');

rad2deg(dist(qVector1,qVector2))
```

```
ans = 4×1

    6.0000
    6.0827
    6.0827
    6.0287
```

Note that a quaternion represents the same rotation as its negative. Calculate a quaternion and its negative.

```
qPositive = quaternion([30,45,-60],'eulerd','zyx','frame')
```

```
qPositive = quaternion
     0.72332 - 0.53198i + 0.20056j +  0.3919k
```

```
qNegative = -qPositive
```

```
qNegative = quaternion
    -0.72332 + 0.53198i - 0.20056j -  0.3919k
```

Find the distance between the quaternion and its negative.

```
dist(qPositive,qNegative)
```

```
ans = 0
```

The components of a quaternion may look different from the components of its negative, but both expressions represent the same rotation.

Input Arguments

quatA,quatB — Quaternions to calculate distance between

scalar | vector | matrix | multidimensional array

Quaternions to calculate distance between, specified as comma-separated quaternions or arrays of quaternions. `quatA` and `quatB` must have compatible sizes:

- `size(quatA) == size(quatB)`, or
- `numel(quatA) == 1`, or
- `numel(quatB) == 1`, or

- if `[Adim1,…,AdimN] = size(quatA)` and `[Bdim1,…,BdimN] = size(quatB)`, then for `i = 1:N`, either `Adimi==Bdimi` or `Adim==1` or `Bdim==1`.

  If one of the quaternion arguments contains only one quaternion, then this function returns the distances between that quaternion and every quaternion in the other argument.

Data Types: `quaternion`

Output Arguments

`distance` — Angular distance (radians)
scalar | vector | matrix | multidimensional array

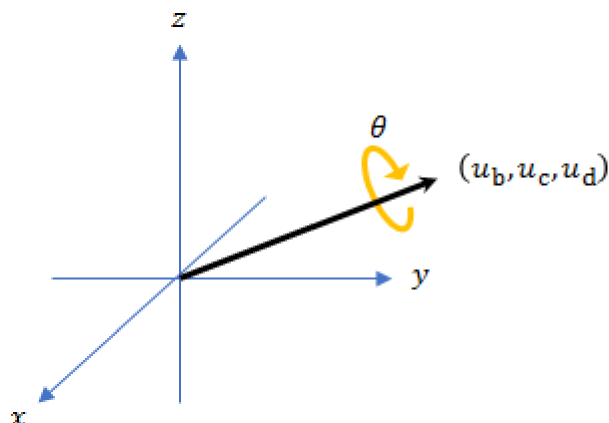
Angular distance in radians, returned as an array. The dimensions are the maximum of the union of `size(quatA)` and `size(quatB)`.

Data Types: `single` | `double`

Algorithms

The `dist` function returns the angular distance between two quaternions.

A quaternion may be defined by an axis (u_b,u_c,u_d) and angle of rotation θ_q:
$q = \cos\left(\theta_q/2\right) + \sin\left(\theta_q/2\right)(u_b\mathrm{i} + u_c\mathrm{j} + u_d\mathrm{k})$.



Given a quaternion in the form, $q = a + b\mathrm{i} + c\mathrm{j} + d\mathrm{k}$, where a is the real part, you can solve for the angle of q as $\theta_q = 2\cos^{-1}(a)$.

Consider two quaternions, p and q, and the product $z = p*\mathrm{conjugate}(q)$. As p approaches q, the angle of z goes to 0, and z approaches the unit quaternion.

The angular distance between two quaternions can be expressed as $\theta_z = 2\cos^{-1}(\mathrm{real}(z))$.

Using the `quaternion` data type syntax, the angular distance is calculated as:

```
angularDistance = 2*acos(abs(parts(p*conj(q))));
```

Version History

Introduced in R2018a

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

Functions

`parts` | `conj`

Objects

`quaternion`

eul2quat

Convert Euler angles to quaternion

Syntax

```
quat = eul2quat(eul)
quat = eul2quat(eul,sequence)
```

Description

`quat = eul2quat(eul)` converts a given set of Euler angles, `eul`, to the corresponding quaternion, `quat`. The default order for Euler angle rotations is `"ZYX"`.

`quat = eul2quat(eul,sequence)` converts a set of Euler angles into a quaternion. The Euler angles are specified in the axis rotation sequence, `sequence`. The default order for Euler angle rotations is `"ZYX"`.

Examples

Convert Euler Angles to Quaternion

```
eul = [0 pi/2 0];
qZYX = eul2quat(eul)
```

```
qZYX = 1×4

    0.7071         0    0.7071         0
```

Convert Euler Angles to Quaternion Using Default ZYZ Axis Order

```
eul = [pi/2 0 0];
qZYZ = eul2quat(eul,"ZYZ")
```

```
qZYZ = 1×4

    0.7071         0         0    0.7071
```

Input Arguments

`eul` — Euler rotation angles
n-by-3 matrix

Euler rotation angles in radians, specified as an *n*-by-3 array of intrinsic Euler rotation angles. Each row represents one Euler angle set in the sequence defined by the `sequence` argument. For example, with the default sequence `"ZYX"`, each row of `eul` is of the form `[zAngle yAngle xAngle]`.

Example: `[0 0 1.5708]`

sequence — Axis-rotation sequence
"ZYX" (default) | "ZYZ" | "ZXY" | "ZXZ" | "YXY" | "YZX" | "YXZ" | "YZY" | "XYX" | "XYZ" | "XZX" | "XZY"

Axis-rotation sequence for the Euler angles, specified as one of these string scalars:

- "ZYX" (default)
- "ZYZ"
- "ZXY"
- "ZXZ"
- "YXY"
- "YZX"
- "YXZ"
- "YZY"
- "XYX"
- "XYZ"
- "XZX"
- "XZY"

Each character indicates the corresponding axis. For example, if the sequence is "ZYX", then the three specified Euler angles are interpreted in order as a rotation around the *z*-axis, a rotation around the *y*-axis, and a rotation around the *x*-axis. When applying this rotation to a point, it will apply the axis rotations in the order *x*, then *y*, then *z*.

Data Types: `string` | `char`

Output Arguments

quat — Unit quaternion
n-by-4 matrix

Unit quaternion, returned as an *n*-by-4 matrix containing *n* quaternions. Each quaternion, one per row, is of the form $q = [w\ x\ y\ z]$, with *w* as the scalar number.

Example: `[0.7071 0.7071 0 0]`

Version History

Introduced in R2015a

R2023a: Additional Euler sequence support

`eul2quat` supports additional Euler sequences for the `sequences` argument. These are all the supported Euler sequences:

- "ZYX"

- `"ZYZ"`
- `"ZXY"`
- `"ZXZ"`
- `"YXY"`
- `"YZX"`
- `"YXZ"`
- `"YZY"`
- `"XYX"`
- `"XYZ"`
- `"XZX"`
- `"XZY"`

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

`quat2eul` | `quaternion`

Topics
"Coordinate Transformations in Robotics"

eul2rotm

Convert Euler angles to rotation matrix

Syntax

```
rotm = eul2rotm(eul)
rotm = eul2rotm(eul,sequence)
```

Description

`rotm = eul2rotm(eul)` converts a set of Euler angles, `eul`, to the corresponding rotation matrix, `rotm`. When using the rotation matrix, premultiply it with the coordinates to be rotated (as opposed to postmultiplying). The default order for Euler angle rotations is `"ZYX"`.

`rotm = eul2rotm(eul,sequence)` converts Euler angles to a rotation matrix, `rotm`. The Euler angles are specified in the axis rotation sequence, `sequence`. The default order for Euler angle rotations is `"ZYX"`.

Examples

Convert Euler Angles to Rotation Matrix

```
eul = [0 pi/2 0];
rotmZYX = eul2rotm(eul)
```

```
rotmZYX = 3×3

    0.0000         0    1.0000
         0    1.0000         0
   -1.0000         0    0.0000
```

Convert Euler Angles to Rotation Matrix Using ZYZ Axis Order

```
eul = [0 pi/2 pi/2];
rotmZYZ = eul2rotm(eul,'ZYZ')
```

```
rotmZYZ = 3×3

    0.0000   -0.0000    1.0000
    1.0000    0.0000         0
   -0.0000    1.0000    0.0000
```

Input Arguments

`eul` — Euler rotation angles
n-by-3 matrix

Euler rotation angles in radians, specified as an *n*-by-3 array of intrinsic Euler rotation angles. Each row represents one Euler angle set in the sequence defined by the `sequence` argument. For example, with the default sequence `"ZYX"`, each row of `eul` is of the form `[zAngle yAngle xAngle]`.

Example: `[0 0 1.5708]`

`sequence` — Axis-rotation sequence
`"ZYX"` (default) | `"ZYZ"` | `"ZXY"` | `"ZXZ"` | `"YXY"` | `"YZX"` | `"YXZ"` | `"YZY"` | `"XYX"` | `"XYZ"` | `"XZX"` | `"XZY"`

Axis-rotation sequence for the Euler angles, specified as one of these string scalars:

- `"ZYX"` (default)
- `"ZYZ"`
- `"ZXY"`
- `"ZXZ"`
- `"YXY"`
- `"YZX"`
- `"YXZ"`
- `"YZY"`
- `"XYX"`
- `"XYZ"`
- `"XZX"`
- `"XZY"`

Each character indicates the corresponding axis. For example, if the sequence is `"ZYX"`, then the three specified Euler angles are interpreted in order as a rotation around the *z*-axis, a rotation around the *y*-axis, and a rotation around the *x*-axis. When applying this rotation to a point, it will apply the axis rotations in the order *x*, then *y*, then *z*.

Data Types: `string` | `char`

Output Arguments

`rotm` — Rotation matrix
3-by-3-by-*n* matrix

Rotation matrix, returned as a 3-by-3-by-*n* matrix containing *n* rotation matrices. Each rotation matrix has a size of 3-by-3 and is orthonormal. When using the rotation matrix, premultiply it with the coordinates to be rotated (as opposed to postmultiplying).

Example: `[0 0 1; 0 1 0; -1 0 0]`

Version History

Introduced in R2015a

R2023a: Additional Euler sequence support

`eul2rotm` supports additional Euler sequences for the `sequences` argument. These are all the supported Euler sequences:

- "ZYX"
- "ZYZ"
- "ZXY"
- "ZXZ"
- "YXY"
- "YZX"
- "YXZ"
- "YZY"
- "XYX"
- "XYZ"
- "XZX"
- "XZY"

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also
`rotm2eul` | `so2` | `so3`

Topics
"Coordinate Transformations in Robotics"

eul2tform

Convert Euler angles to homogeneous transformation

Syntax

```
tform = eul2tform(eul)
tform = eul2tform(eul,sequence)
```

Description

`tform = eul2tform(eul)` converts a set of Euler angles, `eul`, into a homogeneous transformation matrix, `tform`. When using the transformation matrix, premultiply it with the coordinates to be transformed (as opposed to postmultiplying). The default order for Euler angle rotations is `"ZYX"`.

`tform = eul2tform(eul,sequence)` converts Euler angles to a homogeneous transformation. The Euler angles are specified in the axis rotation sequence, `sequence`. The default order for Euler angle rotations is `"ZYX"`.

Examples

Convert Euler Angles to Homogeneous Transformation Matrix

```
eul = [0 pi/2 0];
tformZYX = eul2tform(eul)
```

```
tformZYX = 4×4

    0.0000         0    1.0000         0
         0    1.0000         0         0
   -1.0000         0    0.0000         0
         0         0         0    1.0000
```

Convert Euler Angles to Homogeneous Transformation Matrix Using ZYZ Axis Order

```
eul = [0 pi/2 pi/2];
tformZYZ = eul2tform(eul,'ZYZ')
```

```
tformZYZ = 4×4

    0.0000   -0.0000    1.0000         0
    1.0000    0.0000         0         0
   -0.0000    1.0000    0.0000         0
         0         0         0    1.0000
```

Input Arguments

`eul` — Euler rotation angles
n-by-3 matrix

Euler rotation angles in radians, specified as an n-by-3 array of intrinsic Euler rotation angles. Each row represents one Euler angle set in the sequence defined by the `sequence` argument. For example, with the default sequence `"ZYX"`, each row of `eul` is of the form `[zAngle yAngle xAngle]`.

Example: `[0 0 1.5708]`

`sequence` — Axis-rotation sequence
`"ZYX"` (default) | `"ZYZ"` | `"ZXY"` | `"ZXZ"` | `"YXY"` | `"YZX"` | `"YXZ"` | `"YZY"` | `"XYX"` | `"XYZ"` | `"XZX"` | `"XZY"`

Axis-rotation sequence for the Euler angles, specified as one of these string scalars:

- `"ZYX"` (default)
- `"ZYZ"`
- `"ZXY"`
- `"ZXZ"`
- `"YXY"`
- `"YZX"`
- `"YXZ"`
- `"YZY"`
- `"XYX"`
- `"XYZ"`
- `"XZX"`
- `"XZY"`

Each character indicates the corresponding axis. For example, if the sequence is `"ZYX"`, then the three specified Euler angles are interpreted in order as a rotation around the z-axis, a rotation around the y-axis, and a rotation around the x-axis. When applying this rotation to a point, it will apply the axis rotations in the order x, then y, then z.

Data Types: `string` | `char`

Output Arguments

`tform` — Homogeneous transformation
4-by-4-by-n matrix

Homogeneous transformation matrix, specified by a 4-by-4-by-n array of n homogeneous transformation matrices. When using the transformation matrix, premultiply it with the coordinates to be rotated (as opposed to postmultiplying).

Example: `[0 0 1 0; 0 1 0 0; -1 0 0 0; 0 0 0 1]`

Version History

Introduced in R2015a

R2023a: Additional Euler sequence support

`eul2tform` supports additional Euler sequences for the `sequences` argument. These are all the supported Euler sequences:

- `"ZYX"`
- `"ZYZ"`
- `"ZXY"`
- `"ZXZ"`
- `"YXY"`
- `"YZX"`
- `"YXZ"`
- `"YZY"`
- `"XYX"`
- `"XYZ"`
- `"XZX"`
- `"XZY"`

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also
`tform2eul` | `se2` | `se3`

Topics
"Coordinate Transformations in Robotics"

euler

Convert quaternion to Euler angles (radians)

Syntax

`eulerAngles = euler(quat,rotationSequence,rotationType)`

Description

`eulerAngles = euler(quat,rotationSequence,rotationType)` converts the quaternion, `quat`, to an *N*-by-3 matrix of Euler angles.

Examples

Convert Quaternion to Euler Angles in Radians

Convert a quaternion frame rotation to Euler angles in radians using the 'ZYX' rotation sequence.

```
quat = quaternion([0.7071 0.7071 0 0]);
eulerAnglesRandians = euler(quat,'ZYX','frame')
```

```
eulerAnglesRandians = 1×3

         0         0    1.5708
```

Input Arguments

`quat` — Quaternion to convert to Euler angles
scalar | vector | matrix | multidimensional array

Quaternion to convert to Euler angles, specified as a scalar, vector, matrix, or multidimensional array of quaternions.

Data Types: `quaternion`

`rotationSequence` — Rotation sequence
`'ZYX'` | `'ZYZ'` | `'ZXY'` | `'ZXZ'` | `'YXZ'` | `'YXY'` | `'YZX'` | `'YZY'` | `'XYZ'` | `'XYX'` | `'XZY'` | `'XZX'`

Rotation sequence of Euler representation, specified as a character vector or string.

The rotation sequence defines the order of rotations about the axes. For example, if you specify a rotation sequence of `'YZX'`:

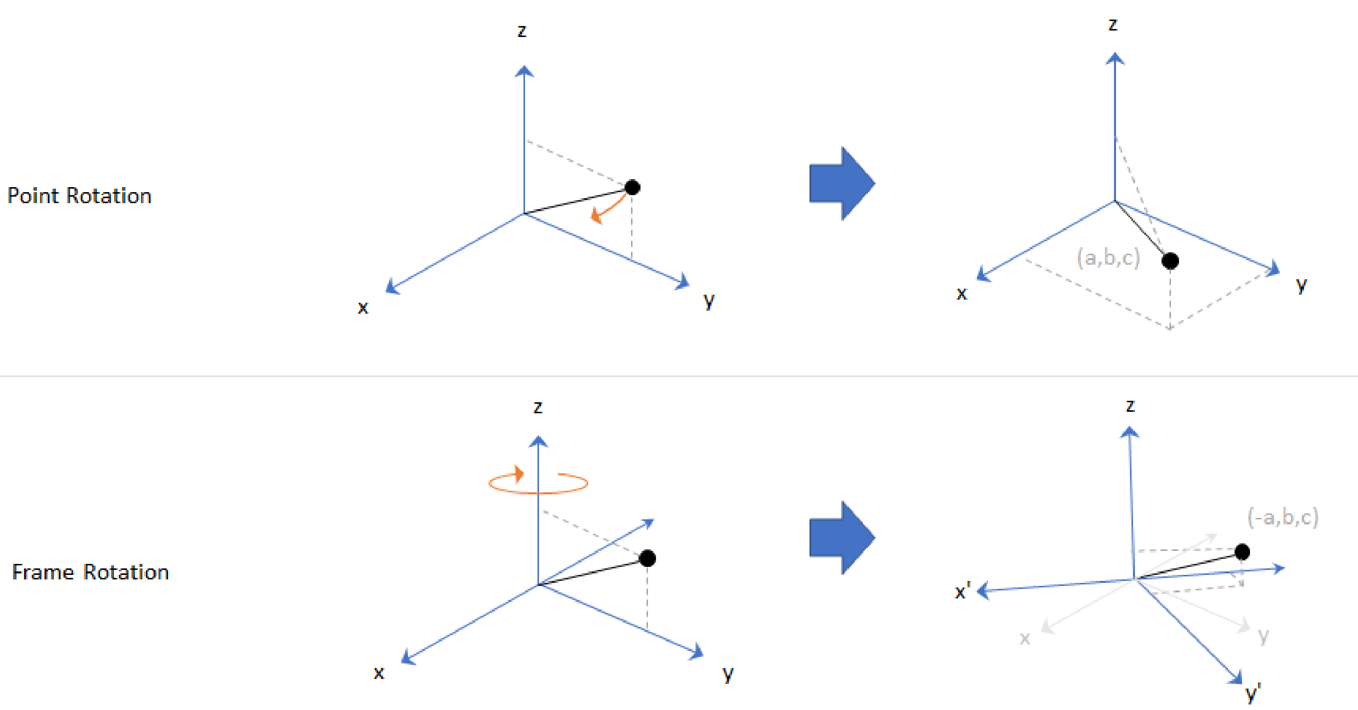
1. The first rotation is about the y-axis.
2. The second rotation is about the new z-axis.
3. The third rotation is about the new x-axis.

Data Types: `char` | `string`

`rotationType` — Type of rotation
`'point'` | `'frame'`

Type of rotation, specified as `'point'` or `'frame'`.

In a point rotation, the frame is static and the point moves. In a frame rotation, the point is static and the frame moves. Point rotation and frame rotation define equivalent angular displacements but in opposite directions.



Data Types: `char` | `string`

Output Arguments

`eulerAngles` — Euler angle representation (radians)
N-by-3 matrix

Euler angle representation in radians, returned as a *N*-by-3 matrix. *N* is the number of quaternions in the `quat` argument.

For each row of `eulerAngles`, the first element corresponds to the first axis in the rotation sequence, the second element corresponds to the second axis in the rotation sequence, and the third element corresponds to the third axis in the rotation sequence.

The data type of the Euler angles representation is the same as the underlying data type of `quat`.

Data Types: `single` | `double`

Version History

Introduced in R2018a

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

Functions
`eulerd` | `rotateframe` | `rotatepoint`

Objects
`quaternion`

eulerd

Convert quaternion to Euler angles (degrees)

Syntax

```
eulerAngles = eulerd(quat,rotationSequence,rotationType)
```

Description

`eulerAngles = eulerd(quat,rotationSequence,rotationType)` converts the quaternion, `quat`, to an *N*-by-3 matrix of Euler angles in degrees.

Examples

Convert Quaternion to Euler Angles in Degrees

Convert a quaternion frame rotation to Euler angles in degrees using the 'ZYX' rotation sequence.

```
quat = quaternion([0.7071 0.7071 0 0]);
eulerAnglesDegrees = eulerd(quat,'ZYX','frame')
```

```
eulerAnglesDegrees = 1×3

         0         0   90.0000
```

Input Arguments

`quat` — Quaternion to convert to Euler angles
scalar | vector | matrix | multidimensional array

Quaternion to convert to Euler angles, specified as a scalar, vector, matrix, or multidimensional array of quaternions.

Data Types: `quaternion`

`rotationSequence` — Rotation sequence
`'ZYX'` | `'ZYZ'` | `'ZXY'` | `'ZXZ'` | `'YXZ'` | `'YXY'` | `'YZX'` | `'YZY'` | `'XYZ'` | `'XYX'` | `'XZY'` | `'XZX'`

Rotation sequence of Euler angle representation, specified as a character vector or string.

The rotation sequence defines the order of rotations about the axes. For example, if you specify a rotation sequence of `'YZX'`:

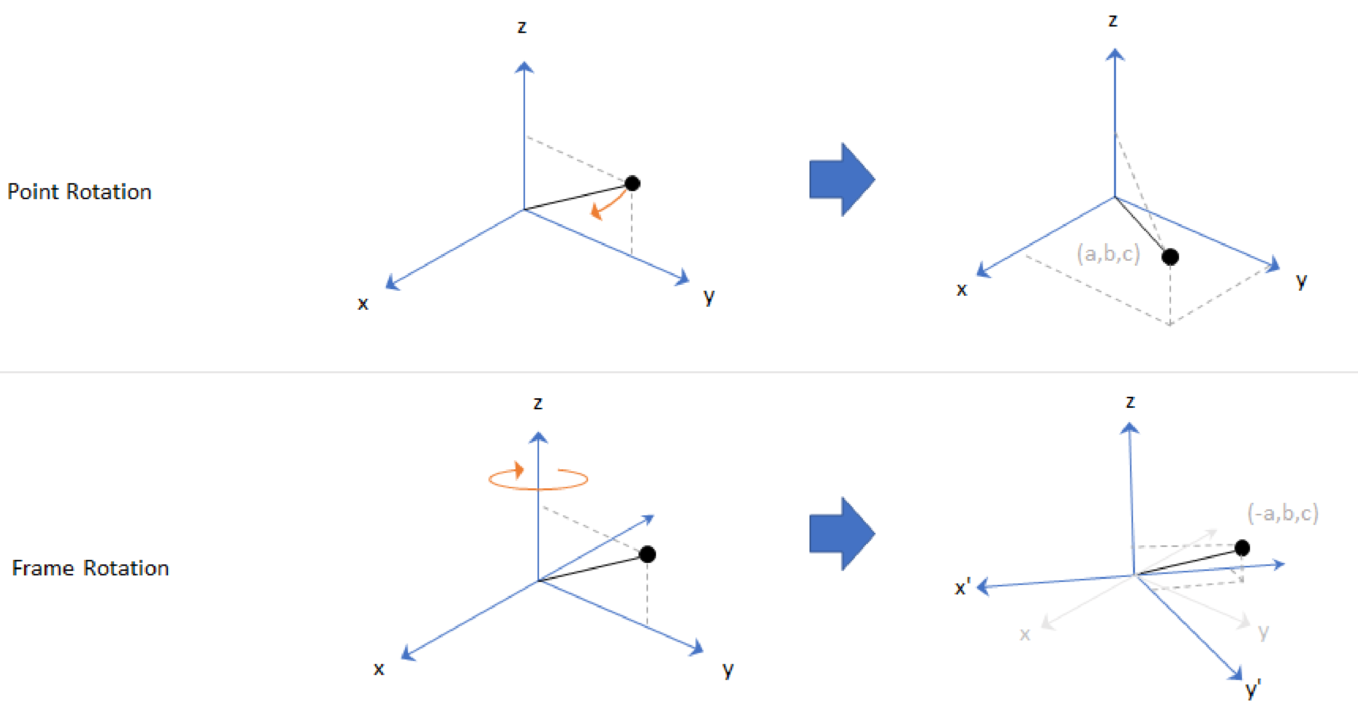
1. The first rotation is about the *y*-axis.
2. The second rotation is about the new *z*-axis.
3. The third rotation is about the new *x*-axis.

Data Types: `char` | `string`

`rotationType` — Type of rotation
`'point'` | `'frame'`

Type of rotation, specified as `'point'` or `'frame'`.

In a point rotation, the frame is static and the point moves. In a frame rotation, the point is static and the frame moves. Point rotation and frame rotation define equivalent angular displacements but in opposite directions.



Data Types: `char` | `string`

Output Arguments

`eulerAngles` — Euler angle representation (degrees)
N-by-3 matrix

Euler angle representation in degrees, returned as a *N*-by-3 matrix. *N* is the number of quaternions in the `quat` argument.

For each row of `eulerAngles`, the first column corresponds to the first axis in the rotation sequence, the second column corresponds to the second axis in the rotation sequence, and the third column corresponds to the third axis in the rotation sequence.

The data type of the Euler angles representation is the same as the underlying data type of `quat`.

Data Types: `single` | `double`

Version History

Introduced in R2018b

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

Functions
`euler` | `rotateframe` | `rotatepoint`

Objects
`quaternion`

exp

Exponential of quaternion array

Syntax

```
B = exp(A)
```

Description

`B = exp(A)` computes the exponential of the elements of the quaternion array `A`.

Examples

Exponential of Quaternion Array

Create a 4-by-1 quaternion array A.

```
A = quaternion(magic(4))
```

```
A = 4x1 quaternion array
     16 +  2i +  3j + 13k
      5 + 11i + 10j +  8k
      9 +  7i +  6j + 12k
      4 + 14i + 15j +  1k
```

Compute the exponential of A.

```
B = exp(A)
```

```
B = 4x1 quaternion array
     5.3525e+06 + 1.0516e+06i + 1.5774e+06j + 6.8352e+06k
        -57.359 -     89.189i -     81.081j -     64.865k
        -6799.1 +     2039.1i +     1747.8j +     3495.6k
          -6.66 +     36.931i +     39.569j +     2.6379k
```

Input Arguments

A — Input quaternion
scalar | vector | matrix | multidimensional array

Input quaternion, specified as a scalar, vector, matrix, or multidimensional array.

Data Types: `quaternion`

Output Arguments

B — Result
scalar | vector | matrix | multidimensional array

Result of quaternion exponential, returned as a scalar, vector, matrix, or multidimensional array.

Data Types: `quaternion`

Algorithms

Given a quaternion $A = a + b\text{i} + c\text{j} + d\text{k} = a + \bar{v}$, the exponential is computed by

$$\exp(A) = e^a\left(\cos\|\bar{v}\| + \frac{\bar{v}}{\|\bar{v}\|}\sin\|\bar{v}\|\right)$$

Version History

Introduced in R2018b

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

Functions

`.^,power` | `log`

Objects

`quaternion`

gazebogenmsg

Generate dependencies for Gazebo custom message support

Syntax

```
gazebogenmsg(folderpath)
gazebogenmsg(folderpath,Name,Value)
```

Description

`gazebogenmsg(folderpath)` generates dependencies for Gazebo custom message support using the protocol buffer (protobuf) files (`.proto`) in the specified folder `folderpath`. It then outputs the generated dependency files to the same folder. The function expects one or more `.proto` files in the same folder. See "Algorithms" on page 2-78 for more information about using Simulink to communicate with Gazebo, as well as sending and receiving custom messages.

`gazebogenmsg(folderpath,Name,Value)` specifies options using one or more name-value pair arguments.

For example, `'GazeboVersion','Gazebo 10'` sets the Gazebo message version to Gazebo 10.

Examples

Generate Dependencies for User-Defined Gazebo Custom Message

Create a folder in a local directory.

```
folderPath = fullfile(pwd,'customMessage')
```

```
folderPath = 
'C:\TEMP\Bdoc23a_2213998_3568\ib570499\13\tp73e6043f\robotics-ex62907275\customMessage'
```

```
mkdir(folderPath)
```

Create a `.proto` file inside the folder and define protobuf custom message fields.

```
messageDefinition = {'message MyPose'
                     '{'
                     '    required double x = 1;'
                     '    required double y = 2;'
                     '    required double z = 3;'
                     '}'};
fileID = fopen(fullfile(folderPath,'MyPose.proto'),'w');
fprintf(fileID,'%s\n',messageDefinition{:});
fclose(fileID);
```

Use the `gazebogenmsg` function to generate dependences in the created folder.

```
gazebogenmsg(folderPath)
```

```
Validating ...
Selected compiler details: "Microsoft Visual C++ 2019 16.0"
```

```
[libprotobuf WARNING] No syntax specified for the proto file: MyPose.proto. Please use 'syntax = 
Building shared library ...
Microsoft (R) C/C++ Optimizing Compiler Version 19.15.26726 for x64
Copyright (C) Microsoft Corporation.  All rights reserved.

MyPose.pb.cc
Microsoft (R) Incremental Linker Version 14.15.26726.0
Copyright (C) Microsoft Corporation.  All rights reserved.

/out:MyPose.pb.dll
/dll
/implib:MyPose.pb.lib
/LIBPATH:B:\matlab\toolbox\shared\robotics\externalDependency\libprotobuf\lib
libprotobuf3.lib
/OUT:C:\TEMP\Bdoc23a_2213998_3568\ib570499\13\tp73e6043f\robotics-ex62907275\customMessage\insta
/IMPLIB:C:\TEMP\Bdoc23a_2213998_3568\ib570499\13\tp73e6043f\robotics-ex62907275\customMessage\ins
C:\TEMP\Bdoc23a_2213998_3568\ib570499\13\tp73e6043f\robotics-ex62907275\customMessage\install\MyP
   Creating library C:\TEMP\Bdoc23a_2213998_3568\ib570499\13\tp73e6043f\robotics-ex62907275\custo
Building MEX for "MyPose.proto" file ...
Building with 'Microsoft Visual C++ 2019'.
MEX completed successfully.
Building with 'Microsoft Visual C++ 2019'.
MEX completed successfully.
Building custom message utilities  ...
DONE.

To use the gazebo custom messages, execute following commands:

addpath('C:\TEMP\Bdoc23a_2213998_3568\ib570499\13\tp73e6043f\robotics-ex62907275\customMessage\in
savepath
```

Use the following commands to add and save the install path.

```
addpath(fullfile(folderPath,'install'))
```

```
savepath
```

Create a Gazebo plugin package `'MyPlugin'` inside the custom message folder using the `packageGazeboPlugin` function.

```
packageGazeboPlugin(fullfile(folderPath,'MyPlugin'),folderPath)
```

Generate Dependencies for Built-in Gazebo Message

Create a folder in a local directory.

```
folderPath = fullfile(pwd,'customMessage');
mkdir(folderPath)
cd(folderPath)
```

Use the `gazebogenmsg` function to generate dependencies for a built-in gazebo message in the specified folder.

```
gazebogenmsg(folderPath,"GazeboMessageList","gazebo.msgs.Image");
```

```
Validating ...
Selected compiler details: "Microsoft Visual C++ 2019 16.0"
Building shared library ...
Microsoft (R) C/C++ Optimizing Compiler Version 19.15.26726 for x64
Copyright (C) Microsoft Corporation.  All rights reserved.

image.pb.cc
Microsoft (R) Incremental Linker Version 14.15.26726.0
Copyright (C) Microsoft Corporation.  All rights reserved.

/out:image.pb.dll
/dll
/implib:image.pb.lib
/LIBPATH:B:\matlab\toolbox\shared\robotics\externalDependency\libprotobuf\lib
libprotobuf3.lib
/OUT:C:\TEMP\Bdoc23a_2213998_3568\ib570499\13\tp73e6043f\robotics-ex40128733\customMessage\insta
/IMPLIB:C:\TEMP\Bdoc23a_2213998_3568\ib570499\13\tp73e6043f\robotics-ex40128733\customMessage\ins
C:\TEMP\Bdoc23a_2213998_3568\ib570499\13\tp73e6043f\robotics-ex40128733\customMessage\install\ima
   Creating library C:\TEMP\Bdoc23a_2213998_3568\ib570499\13\tp73e6043f\robotics-ex40128733\custo
Building MEX for "image.proto" file ...
Building with 'Microsoft Visual C++ 2019'.
MEX completed successfully.
Building with 'Microsoft Visual C++ 2019'.
MEX completed successfully.
Building custom message utilities  ...
DONE.

To use the gazebo custom messages, execute following commands:

addpath('C:\TEMP\Bdoc23a_2213998_3568\ib570499\13\tp73e6043f\robotics-ex40128733\customMessage\in
savepath
```

Use the following commands to add and save the install path.

```
addpath(fullfile(folderPath,'install'))
```

```
savepath
```

Create a Gazebo plugin package using the `packageGazeboPlugin` function.

```
packageGazeboPlugin
```

Input Arguments

`folderpath` — Path of custom message folder
string scalar | character vector

Path of the custom message folder, specified as a string scalar or character vector. The folder must contain one or more `.proto` files. The path also specifies the location in which to output the generated dependency files.

Example: `gazebogenmsg('C:\GazeboCustomMsg')`

Data Types: `char` | `string`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `'GazeboMessageList','gazebo.msgs.IMU'` generates dependences for the built-in Gazebo message `gazebo.msgs.IMU`.

`GazeboVersion` — Gazebo message version
`'Gazebo 9'` (default) | `'Gazebo 10'` | `'Gazebo 11'`

Gazebo message version, specified as the comma-separated pair consisting of `'GazeboVersion'` and either `'Gazebo 9'`, `'Gazebo 10'` or `'Gazebo 11'`.

Example: `'GazeboVersion','Gazebo 10'`

Data Types: `char` | `string`

`GazeboMessageList` — Gazebo built-in messages
string scalar | character vector

Gazebo built-in messages, specified as the comma-separated pair consisting of `'GazeboMessageList'` and one or more built-in messages from the list of valid Gazebo messages.

To get a list of valid Gazebo messages, press **Tab** after entering the `'GazeboMessageList'` argument name. You can select a valid Gazebo message value from the list.

Example: `'GazeboMessageList','gazebo.msgs.Altimeter'`

Data Types: `char` | `string`

Limitations

- The `gazebogenmsg` function supports the **proto2** version of the protobuf language. The function does not support the proto2 fields `map`, `group`, `extend`, `extensions`, and `reserved`.
- You can run the Simulink model multiple times but you need to restart MATLAB to run `gazebogenmsg` function again.
- `gazebogenmsg` function not supported with MATLAB Compiler™.

Tips

Supported Compilers

Windows: Microsoft Visual C++ 14.0 and later

Linux: g++ 6.0.0 and later

Mac: Xcode Clang++ 10.0.0 and later

Algorithms

1. Add and save the install path by running the command presented at the end of `gazebogenmsg` function output.
2. Use the `packageGazeboPlugin` function to package the plugin.
3. Copy, install and run the plugin on the Gazebo machine.
4. Use the Gazebo Publish Simulink block to send the custom messages to the Gazebo machine.
5. Use the Gazebo Subscribe Simulink block to receive the custom messages from the Gazebo machine.

Version History

Introduced in R2020b

References

[1] Google Developers. “Language Guide | Protocol Buffers.” Accessed July 17, 2020. https://developers.google.com/protocol-buffers/docs/proto.

See Also

`packageGazeboPlugin` | Gazebo Publish | Gazebo Subscribe

Topics

“Perform Co-Simulation between Simulink and Gazebo”

gzinit

Initialize connection settings for Gazebo Co-Simulation MATLAB interface

Syntax

```
gzinit
gzinit(HostIP)
gzinit(HostIP,HostPort)
gzinit(HostIP,HostPort,Timeout)
```

Description

`gzinit` initializes connection settings and checks connectivity with the Gazebo plugin running on `localhost` and port `14581`. This syntax sets response timeout to 1 second.

`gzinit(HostIP)` specifies the host name or IP address of the Gazebo plugin `HostIP`.

`gzinit(HostIP,HostPort)` specifies the port number `HostPort`.

`gzinit(HostIP,HostPort,Timeout)` specifies the response timeout `Timeout` in seconds.

Examples

Perform Co-Simulation Between MATLAB and Gazebo

Set up a simulation between MATLAB and Gazebo, receive data from Gazebo, and send commands to Gazebo.

Prerequisite

Follow the instructions in “Perform Co-Simulation between Simulink and Gazebo” to download the Linux virtual machine (VM) with Gazebo and set up `multiSensorPluginTest.world`.

Configure and Perform Gazebo Co-Simulation

Initialize connection settings and check connectivity with the Gazebo plugin running on `192.168.198.129` and port `14581.`

```
gzinit("192.168.198.129",14581)
```

Assign and Retrieve Gazebo Model Information

List the models available in the Gazebo world.

```
modelList = gzmodel("list")
```

```
modelList = 1×11 string
    "ground_plane"    "unit_box"    "camera0"    "camera1"    "depth_camera0"    "depth_camera1"
```

Assign values to the `Position` and `SelfCollide` parameters of the `unit_box` model.

```
[status,message] = gzmodel("set","unit_box","Position",[2 2 0.5],"SelfCollide","on")
```

```
status = 1×2 logical array

   1   1


message = 1×2 string
    "Position parameter set successfully."    "SelfCollide parameter set successfully."
```

Retrieve the values of the `Position` and `SelfCollide` parameters of the `unit_box` model.

```
[position,selfcollide] = gzmodel("get","unit_box","Position","SelfCollide")
```

```
position = 1×3

                         2                         2          0.4999999999951


selfcollide = logical
   1
```

Assign and Retrieve Gazebo Model Link Information

List the links available in the `unit_box` model.

```
linkList = gzlink("list","unit_box")
```

```
linkList = 
"link"
```

Assign values to the link parameters `Mass` and `Gravity` of the link `link` in the `unit_box` model.

```
[status,message] = gzlink("set","unit_box","link","Mass",2,"Gravity","off")
```

```
status = 1×2 logical array

   1   1


message = 1×2 string
    "Mass parameter set successfully."    "Gravity parameter set successfully."
```

Retrieve the values of the link parameters `Mass` and `Gravity` of the link `link` in the `unit_box` model.

```
[mass,gravity] = gzlink("get","unit_box","link","Mass","Gravity")
```

```
mass = 
     2


gravity = logical
   0
```

Assign and Retrieve Gazebo Model Joint Information

List the joints available in the `unit_box` model.

```
jointList = gzjoint("list","unit_box")
```

```
jointList = 
"joint"
```

Assign a value to the joint parameter `Damping` of the axis `Axis0` for the joint `joint` in the `unit_box` model.

```
[status,message] = gzjoint("set","unit_box","joint","Axis","0","Damping",0.25)
```

```
status = logical
   1


message = 
"Damping parameter set successfully."
```

Retrieve the value of the joint parameter `Damping` of the axis `Axis0` for the joint `joint` in the `unit_box` model.

```
damping = gzjoint("get","unit_box","joint","Axis0","Damping")
```

```
damping = 
                      0.25
```

Reset all Gazebo model configurations.

```
gzworld("reset")
```

Input Arguments

`HostIP` — Host name or IP address of machine with Gazebo plugin
`localhost` (default) | string scalar | character vector

The host name or IP address of the machine with the Gazebo plugin, specified as a string scalar or character vector.

Example: `gzinit("172.18.250.191")`

`HostPort` — Port number of machine with Gazebo plugin
`14581` (default) | positive integer

Port number of the machine with the Gazebo plugin, specified as a positive integer. The port number must be the same as the value of `'portNumber'` in the Gazebo `'.world'` file.

Example: `gzinit("172.18.250.191",14581)`

`Timeout` — Response timeout
`1` (default) | positive numeric scalar

Response timeout, specified as a positive numeric scalar. This value determines how long the client will wait for a response from the server, in seconds. Set a higher `Timeout` value for a network with poor connectivity.

Example: `gzinit("172.18.250.191",14581,10)`

Limitations

- `gzinit` function not supported with MATLAB Compiler.

Version History

Introduced in R2021a

See Also

`gzlink` | `gzjoint` | `gzmodel` | `gzworld`

gzjoint

Assign and retrieve Gazebo model joint information

Syntax

```
List = gzjoint("list",modelname)
[Status,Message] = gzjoint("set",modelname,jointname,Name,Value)
[Output1,...,OutputN] = gzjoint("get",modelname,jointname,params)
```

Description

`List = gzjoint("list",modelname)` returns and displays a list of joint names `List` of the specified Gazebo model `modelname`.

`[Status,Message] = gzjoint("set",modelname,jointname,Name,Value)` assigns values to the joint parameters using one or more name-value pair arguments for the specified Gazebo model `modelname` and the joint `jointname`. The function returns the status of the value assignments `Status` and the message of their success and failure `Message`. For example, `gzjoint("set","unit_box","joint","Position",[2 2 0.5])` sets the position of the joint in the model `unit_box`.

`[Output1,...,OutputN] = gzjoint("get",modelname,jointname,params)` retrieves values of the joint parameters using one or more parameter names, `params`, for the specified Gazebo model `modelname` and the joint `jointname`. The function returns one or more outputs, `Output1,...,OutputN`, corresponding to the specified parameter names.

Examples

Perform Co-Simulation Between MATLAB and Gazebo

Set up a simulation between MATLAB and Gazebo, receive data from Gazebo, and send commands to Gazebo.

Prerequisite

Follow the instructions in “Perform Co-Simulation between Simulink and Gazebo” to download the Linux virtual machine (VM) with Gazebo and set up `multiSensorPluginTest.world`.

Configure and Perform Gazebo Co-Simulation

Initialize connection settings and check connectivity with the Gazebo plugin running on `192.168.198.129` and port `14581.`

```
gzinit("192.168.198.129",14581)
```

Assign and Retrieve Gazebo Model Information

List the models available in the Gazebo world.

```
modelList = gzmodel("list")
```

```
modelList = 1×11 string
    "ground_plane"    "unit_box"    "camera0"    "camera1"    "depth_camera0"    "depth_camera1"
```

Assign values to the `Position` and `SelfCollide` parameters of the `unit_box` model.

```
[status,message] = gzmodel("set","unit_box","Position",[2 2 0.5],"SelfCollide","on")
```

```
status = 1×2 logical array

   1   1

message = 1×2 string
    "Position parameter set successfully."    "SelfCollide parameter set successfully."
```

Retrieve the values of the `Position` and `SelfCollide` parameters of the `unit_box` model.

```
[position,selfcollide] = gzmodel("get","unit_box","Position","SelfCollide")
```

```
position = 1×3

                        2                         2           0.4999999999951

selfcollide = logical
   1
```

Assign and Retrieve Gazebo Model Link Information

List the links available in the `unit_box` model.

```
linkList = gzlink("list","unit_box")
```

```
linkList = 
"link"
```

Assign values to the link parameters `Mass` and `Gravity` of the link `link` in the `unit_box` model.

```
[status,message] = gzlink("set","unit_box","link","Mass",2,"Gravity","off")
```

```
status = 1×2 logical array

   1   1

message = 1×2 string
    "Mass parameter set successfully."    "Gravity parameter set successfully."
```

Retrieve the values of the link parameters `Mass` and `Gravity` of the link `link` in the `unit_box` model.

```
[mass,gravity] = gzlink("get","unit_box","link","Mass","Gravity")
```

```
mass = 
     2
```

```
gravity = logical
   0
```

Assign and Retrieve Gazebo Model Joint Information

List the joints available in the `unit_box` model.

```
jointList = gzjoint("list","unit_box")
```

```
jointList = 
"joint"
```

Assign a value to the joint parameter `Damping` of the axis `Axis0` for the joint `joint` in the `unit_box` model.

```
[status,message] = gzjoint("set","unit_box","joint","Axis","0","Damping",0.25)
```

```
status = logical
   1


message = 
"Damping parameter set successfully."
```

Retrieve the value of the joint parameter `Damping` of the axis `Axis0` for the joint `joint` in the `unit_box` model.

```
damping = gzjoint("get","unit_box","joint","Axis0","Damping")
```

```
damping = 
                      0.25
```

Reset all Gazebo model configurations.

```
gzworld("reset")
```

Input Arguments

`modelname` — Gazebo model name
string scalar | character vector

Gazebo model name, specified as a string scalar or character vector.

Data Types: `char` | `string`

`jointname` — Associated joint name
string scalar | character vector

Associated joint name, specified as a string scalar or character vector.

Data Types: `char` | `string`

`params` — Gazebo model joint parameters
string scalars | character vectors

Gazebo model joint parameters, specified as a comma-separated list of string scalars or character vectors. Specify the list of parameters you want to retrieve the values, from these tables.

Gazebo model joint axis parameters for `Axis0` or `Axis1`, specified as the comma-separated arguments consisting of `"Axis0"` or `"Axis1"`, respectively, and one or more of the options in this table.

Option Name	Description
`"Angle"`	Get the angle parameter of the Gazebo model joint axis for `Axis0` or `Axis1`.
`"Damping"`	Get the damping parameter of the Gazebo model joint axis for `Axis0` or `Axis1`.
`"Friction"`	Get the friction parameter of the Gazebo model joint axis for `Axis0` or `Axis1`.
`"XYZ"`	Get the position of the Gazebo model joint axis for `Axis0` or `Axis1`.

Gazebo model joint parameters:

Parameters	Description
`"CFM"`	Get the CFM parameter of the Gazebo model joint.
`"FudgeFactor"`	Get the fudge factor parameter of the Gazebo model joint.
`"Orientation"`	Get the orientation parameter of the Gazebo model joint.
`"Position"`	Get the position of the Gazebo model joint.
`"SuspensionCFM"`	Get the suspension CFM parameter of the Gazebo model joint.
`"SuspensionERP"`	Get the suspension ERP parameter of the Gazebo model joint.

Example: `[ang,damp,cfm,pos] = gzjoint("get","unit_box","joint","Axis0","Angle","Damping","CFM","Position")`

Data Types: `char` | `string`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: For example, `gzjoint("set","unit_box","joint","Position",[2 2 0.5])` sets the position of the joint in the model `unit_box`.

`Axis` — Gazebo model joint axis parameters

`"0"` | `"1"`

Gazebo model joint axis parameters, specified as the comma-separated pair consisting of `'Axis'` and either `"0"` or `"1"`. Then, specify one or more of the options in this table as the comma-separated pair consisting of an option name and its value.

Option Name	Description
"Angle"	Set the angle parameter of the Gazebo model joint axis as a numeric scalar in radians.
"Damping"	Set the damping parameter of the Gazebo model joint axis as a numeric scalar in newton meter second per radians.
"Friction"	Set the friction parameter of the Gazebo model joint axis as a numeric scalar in newton.
"XYZ"	Set the position of the Gazebo model joint axis as a three-element vector of the form `[X Y Z]` in meters.

Example: `[status,message] = gzjoint("set","unit_box","joint","Axis","0","Damping",0.25)`

Data Types: `single` | `double`

`CFM` — Gazebo model joint constraint force mixing (CFM) parameter
numeric scalar

Gazebo model joint CFM parameter, specified as the comma-separated pair consisting of `'CFM'` and a numeric scalar.

Example: `[status,message] = gzjoint("set","unit_box","joint","CFM",1);`

Data Types: `single` | `double`

`FudgeFactor` — Gazebo model joint fudge factor parameter
numeric scalar

Gazebo model joint fudge factor parameter, specified as the comma-separated pair consisting of `'FudgeFactor'` and a numeric scalar.

Example: `[status,message] = gzjoint("set","unit_box","joint","FudgeFactor",1);`

Data Types: `single` | `double`

`Orientation` — Gazebo model joint orientation parameter
four-element vector

Gazebo model joint orientation parameter, specified as the comma-separated pair consisting of `'Orientation'` and a four-element quaternion vector of the form [*w x y z*].

Example: `[status,message] = gzjoint("set","unit_box","joint","Orientation",[1 0 0 0]);`

Data Types: `single` | `double`

`Position` — Gazebo model joint position
three-element vector

Gazebo model joint position parameter, specified as the comma-separated pair consisting of `'Position'` and a three-element vector of the form [*x y z*] in meters.

Example: `[status,message] = gzjoint("set","unit_box","joint","Position",[0 0 0]);`

Data Types: `single` | `double`

`SuspensionCFM` — Gazebo model joint suspension CFM parameter
numeric scalar

Gazebo model joint suspension CFM parameter, specified as the comma-separated pair consisting of `'SuspensionCFM'` and a numeric scalar.

Example: `[status,message] = gzjoint("set","unit_box","joint","SuspensionCFM",1);`

Data Types: `single` | `double`

`SuspensionERP` — Gazebo model joint suspension error reduction parameter (ERP) parameter
numeric scalar

Gazebo model joint suspension ERP parameter, specified as the comma-separated pair consisting of `'SuspensionERP'` and a numeric scalar.

Example: `[status,message] = gzjoint("set","unit_box","joint","SuspensionERP",1);`

Data Types: `single` | `double`

Output Arguments

`List` — List of joints in model
cell array of character vectors

List of joints in the model, returned as a cell array of character vectors.

`Status` — Status of values assigned to parameters
logical array

Status of the values assigned to the parameters, returned as a logical array.

`Message` — Success or failure message
string array

Success or failure message, returned as a string array.

`Output1,...,OutputN` — Values of specified parameters
numeric scalar | numeric vector

Values of specified parameters, returned as a numeric scalar or numeric vector based on the specified parameters. The following tables shows the returned data type of parameter values.

Gazebo model joint axis parameters for `Axis0` or `Axis1`:

Option Name	Description
`"Angle"`	Gazebo model joint axis angle parameter for `Axis0` or `Axis1`, returns a numeric scalar in radians.

Option Name	Description
`"Damping"`	Gazebo model joint axis damping parameter for `Axis0` or `Axis1`, returns a numeric scalar in newton meter second per radians.
`"Friction"`	Gazebo model joint axis friction parameter for `Axis0` or `Axis1`, returns a numeric scalar in newton.
`"XYZ"`	Gazebo model joint axis position parameter for `Axis0` or `Axis1`, returns a three-element vector of the form `[X Y Z]` in meters.

Gazebo model joint parameters:

Parameters	Description
`"CFM"`	Gazebo model joint CFM parameter, returns a numeric scalar.
`"FudgeFactor"`	Gazebo model joint fudge factor parameter, returns a numeric scalar.
`"Orientation"`	Gazebo model joint orientation parameter, returns a four-element quaternion vector of the form [*w x y z*].
`"Position"`	Gazebo model joint position parameter, returns a three-element vector of the form `[x y z]` in meters.
`"SuspensionCFM"`	Gazebo model joint suspension CFM parameter, returns a numeric scalar.
`"SuspensionERP"`	Gazebo model joint suspension ERP parameter, returns a numeric scalar.

Limitations

- `gzjoint` function not supported with MATLAB Compiler.

Version History

Introduced in R2021a

See Also

`gzinit` | `gzlink` | `gzmodel` | `gzworld`

gzlink

Assign and retrieve Gazebo model link information

Syntax

```
List = gzlink("list",modelname)
[Status,Message] = gzlink("set",modelname,linkname,Name,Value)
[Output1,...,OutputN] = gzlink("get",modelname,linkname,params)
```

Description

`List = gzlink("list",modelname)` returns and displays a list of link names `List` in the specified Gazebo model `modelname`.

`[Status,Message] = gzlink("set",modelname,linkname,Name,Value)` assigns values to the link parameters using one or more name-value pair arguments for the specified Gazebo model `modelname` and the link `linkname`. The function returns the status of the value assignments `Status` and the message of their success and failure `Message`. For example, `gzlink("set","unit_box","link","Position",[2 2 0.5])` sets the position of the link in the model `unit_box`.

`[Output1,...,OutputN] = gzlink("get",modelname,linkname,params)` retrieves values of the link parameters using one or more parameter name, `params`, for the specified Gazebo model `modelname` and the link `linkname`. The function returns one or more outputs, `Output1,...,OutputN`, corresponding to the specified parameter names.

Examples

Perform Co-Simulation Between MATLAB and Gazebo

Set up a simulation between MATLAB and Gazebo, receive data from Gazebo, and send commands to Gazebo.

Prerequisite

Follow the instructions in “Perform Co-Simulation between Simulink and Gazebo” to download the Linux virtual machine (VM) with Gazebo and set up `multiSensorPluginTest.world`.

Configure and Perform Gazebo Co-Simulation

Initialize connection settings and check connectivity with the Gazebo plugin running on `192.168.198.129` and port `14581`.

```
gzinit("192.168.198.129",14581)
```

Assign and Retrieve Gazebo Model Information

List the models available in the Gazebo world.

```
modelList = gzmodel("list")
```

```
modelList = 1×11 string
    "ground_plane"    "unit_box"    "camera0"    "camera1"    "depth_camera0"    "depth_camera1"
```

Assign values to the `Position` and `SelfCollide` parameters of the `unit_box` model.

```
[status,message] = gzmodel("set","unit_box","Position",[2 2 0.5],"SelfCollide","on")
```

```
status = 1×2 logical array

   1   1


message = 1×2 string
    "Position parameter set successfully."    "SelfCollide parameter set successfully."
```

Retrieve the values of the `Position` and `SelfCollide` parameters of the `unit_box` model.

```
[position,selfcollide] = gzmodel("get","unit_box","Position","SelfCollide")
```

```
position = 1×3

                        2                         2           0.4999999999951


selfcollide = logical
   1
```

Assign and Retrieve Gazebo Model Link Information

List the links available in the `unit_box` model.

```
linkList = gzlink("list","unit_box")
```

```
linkList = 
"link"
```

Assign values to the link parameters `Mass` and `Gravity` of the link `link` in the `unit_box` model.

```
[status,message] = gzlink("set","unit_box","link","Mass",2,"Gravity","off")
```

```
status = 1×2 logical array

   1   1


message = 1×2 string
    "Mass parameter set successfully."    "Gravity parameter set successfully."
```

Retrieve the values of the link parameters `Mass` and `Gravity` of the link `link` in the `unit_box` model.

```
[mass,gravity] = gzlink("get","unit_box","link","Mass","Gravity")
```

```
mass = 
     2
```

```
gravity = logical
   0
```

Assign and Retrieve Gazebo Model Joint Information

List the joints available in the `unit_box` model.

```
jointList = gzjoint("list","unit_box")
```

```
jointList = 
"joint"
```

Assign a value to the joint parameter `Damping` of the axis `Axis0` for the joint `joint` in the `unit_box` model.

```
[status,message] = gzjoint("set","unit_box","joint","Axis","0","Damping",0.25)
```

```
status = logical
   1


message = 
"Damping parameter set successfully."
```

Retrieve the value of the joint parameter `Damping` of the axis `Axis0` for the joint `joint` in the `unit_box` model.

```
damping = gzjoint("get","unit_box","joint","Axis0","Damping")
```

```
damping = 
                      0.25
```

Reset all Gazebo model configurations.

```
gzworld("reset")
```

Input Arguments

`modelname` — Gazebo model name
string scalar | character vector

Gazebo model name, specified as a string scalar or character vector.

Data Types: `char` | `string`

`linkname` — Associated link name
string scalar | character vector

Associated link name, specified as a string scalar or character vector.

Data Types: `char` | `string`

`params` — Gazebo model link parameters
string scalars | character vectors

Gazebo model link parameters, specified as a comma-separated list of string scalars or character vectors. Specify the list of parameters you want to retrieve the values, from this table.

Parameters	Description
"Canonical"	Get the canonical parameter of the Gazebo model link.
"EnableWind"	Get the wind parameter of the Gazebo model link.
"Gravity"	Get the gravity parameter of the Gazebo model link.
"IsStatic"	Get the IsStatic parameter of the Gazebo model link.
"Kinematic"	Get the kinematic parameter of the Gazebo model link.
"Mass"	Get the mass parameter of the Gazebo model link.
"Orientation"	Get the orientation parameter of the Gazebo model link.
"Position"	Get the position parameter of the Gazebo model link.
"PrincipalMoments"	Get the principal moments parameter of the Gazebo model link.
"ProductOfInertia"	Get the product of inertia parameter of the Gazebo model link.
"SelfCollide"	Get the SelfCollide parameter of the Gazebo model link.

Example: `[mass,gravity] = gzlink("get","unit_box","link","Mass","Gravity")`

Data Types: `char` | `string`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: For example, `gzlink("set","unit_box","link","Position",[2 2 0.5])` sets the position of the link in the model `unit_box`.

`Canonical` — Gazebo model link canonical parameter

`'on'` | `'off'`

Gazebo model link canonical parameter, specified as the comma-separated pair consisting of `'Canonical'` and either `'on'` or `'off'`.

Example: `[status,message] = gzlink("set","unit_box","link","Canonical","off");`

Data Types: `char` | `string`

`EnableWind` — Gazebo model link wind parameter

`'on'` | `'off'`

Gazebo model link wind parameter, specified as the comma-separated pair consisting of `'EnableWind'` and either `'on'` or `'off'`.

Example: `[status,message] = gzlink("set","unit_box","link","EnableWind","off");`

Data Types: `char` | `string`

`Gravity` — Gazebo model link gravity parameter
`'on'` | `'off'`

Gazebo model link gravity parameter, specified as the comma-separated pair consisting of `'Gravity'` and either `'on'` or `'off'`.

Example: `[status,message] = gzlink("set","unit_box","link","Gravity","off");`

Data Types: `char` | `string`

`IsStatic` — Gazebo model link IsStatic parameter
`'on'` | `'off'`

Gazebo model link IsStatic parameter, specified as the comma-separated pair consisting of `'IsStatic'` and either `'on'` or `'off'`.

Example: `[status,message] = gzlink("set","unit_box","link","IsStatic","off");`

Data Types: `char` | `string`

`Kinematic` — Gazebo model link kinematic parameter
`'on'` | `'off'`

Gazebo model link kinematic parameter, specified as the comma-separated pair consisting of `'Kinematic'` and either `'on'` or `'off'`.

Example: `[status,message] = gzlink("set","unit_box","link","Kinematic","off");`

Data Types: `char` | `string`

`Mass` — Gazebo model link mass parameter
numeric scalar

Gazebo model link mass parameter, specified as the comma-separated pair consisting of `'Mass'` and a numeric scalar in kilograms.

Example: `[status,message] = gzlink("set","unit_box","link","Mass",1);`

Data Types: `single` | `double`

`Orientation` — Gazebo model link orientation parameter
four-element vector

Gazebo model link orientation parameter, specified as the comma-separated pair consisting of `'Orientation'` and a four-element quaternion vector of the form [*w x y z*].

Example: `[status,message] = gzlink("set","unit_box","link","Orientation",[1 0 0 0]);`

Data Types: `single` | `double`

`Position` — Gazebo model link position
three-element vector

Gazebo model link position parameter, specified as the comma-separated pair consisting of `'Position'` and a three-element vector of the form [*x y z*] in meters.

Example: `[status,message] = gzlink("set","unit_box","link","Position",[0 0 0]);`

Data Types: `single` | `double`

`PrincipalMoments` — Gazebo model link principal moments
three-element vector

Gazebo model link principal moments parameter, specified as the comma-separated pair consisting of `'PrincipalMoments'` and a three-element vector of the form [*ixx iyy izz*] in kilogram square meters.

Example: `[status,message] = gzlink("set","unit_box","link","PrincipalMoments",[0 0 0]);`

Data Types: `single` | `double`

`ProductOfInertia` — Gazebo model link product of inertia
three-element vector

Gazebo model link product of inertia parameter, specified as the comma-separated pair consisting of `'ProductOfInertia'` and a three-element vector of the form [*ixy ixz iyz*] in kilogram square meters.

Example: `[status,message] = gzlink("set","unit_box","link","ProductOfInertia",[0 0 0]);`

Data Types: `single` | `double`

`SelfCollide` — Gazebo model link SelfCollide parameter
`'on'` | `'off'`

Gazebo model link SelfCollide parameter, specified as the comma-separated pair consisting of `'SelfCollide'` and either `'on'` or `'off'`.

Example: `[status,message] = gzlink("set","unit_box","link","SelfCollide","off");`

Data Types: `char` | `string`

Output Arguments

`List` — List of links in model
cell array of character vectors

List of links in the model, returned as a cell array of character vectors.

`Status` — Status of values assigned to parameters
logical array

Status of the values assigned to the parameters, returned as a logical array.

`Message` — Success or failure message
string array

Success or failure message, returned as a string array.

`Output1,...,OutputN` — Values of specified parameters
logical scalar | numeric scalar | numeric vector

Values of specified parameters, returned as a logical or numeric vector based on the specified parameters. The following table shows the returned data type of parameter values.

Parameters	Description
"Canonical"	Gazebo model link canonical parameter, returns a logical scalar.
"EnableWind"	Gazebo model link wind parameter, returns a logical scalar.
"Gravity"	Gazebo model link gravity parameter, returns a logical scalar.
"IsStatic"	Gazebo model link IsStatic parameter, returns a logical scalar.
"Kinematic"	Gazebo model link kinematic parameter, returns a logical scalar.
"Mass"	Gazebo model link mass parameter, returns a numeric scalar in kilograms.
"Orientation"	Gazebo model link orientation parameter, returns a four-element quaternion vector of the form [*w x y z*].
"Position"	Gazebo model link position parameter, returns a three-element vector of the form [*x y z*] in meters.
"PrincipalMoments"	Gazebo model link principal moments parameter, returns a three-element vector of the form [*ixx iyy izz*] in kilogram square meters.
"ProductOfInertia"	Gazebo model link product of inertia parameter, returns a three-element vector of the form [*ixy ixz iyz*] in kilogram square meters.
"SelfCollide"	Gazebo model link SelfCollide parameter, returns a logical scalar.

Limitations

- `gzlink` function not supported with MATLAB Compiler.

Version History

Introduced in R2021a

See Also

`gzinit` | `gzjoint` | `gzmodel` | `gzworld`

gzmodel

Assign and retrieve Gazebo model information

Syntax

```
List = gzmodel("list")
[Links,Joints] = gzmodel("info",modelname)
[Status,Message] = gzmodel("set",modelname,Name,Value)
[Output1,...,OutputN] = gzmodel("get",modelname,params)
sdfString = gzmodel("importSDF",modelname)
```

Description

`List = gzmodel("list")` returns and displays a list of model names `List` available in the Gazebo world.

If you do not define the output argument, the model names are returned in the MATLAB Command Window.

`[Links,Joints] = gzmodel("info",modelname)` returns and displays a list of link names `Links` and joint names `Joints` of the specified Gazebo model `modelname`.

If you do not define the output argument, the model info is returned in the MATLAB Command Window.

`[Status,Message] = gzmodel("set",modelname,Name,Value)` assigns values to the model parameters using one or more name-value pair arguments for the specified Gazebo model `modelname`. The function returns the status of the value assignments `Status` and the message of their success and failure `Message`. For example, `gzmodel("set","unit_box","Position",[2 2 0.5])` sets the position of the model `unit_box`.

If you do not define the output argument, the status and message are returned in the MATLAB Command Window.

`[Output1,...,OutputN] = gzmodel("get",modelname,params)` retrieves values of the model parameters using one or more parameter name, `params`, for the specified Gazebo model `modelname`. The function returns one or more outputs `Output1,...,OutputN`, corresponding to the specified parameter names.

If you do not define the output argument, the model parameters are returned in the MATLAB Command Window.

`sdfString = gzmodel("importSDF",modelname)` returns the Simulation Description Format (SDF) of the specified Gazebo model as a string.

If you do not define the output argument, the SDF model description are returned in the MATLAB Command Window.

Examples

Perform Co-Simulation Between MATLAB and Gazebo

Set up a simulation between MATLAB and Gazebo, receive data from Gazebo, and send commands to Gazebo.

Prerequisite

Follow the instructions in “Perform Co-Simulation between Simulink and Gazebo” to download the Linux virtual machine (VM) with Gazebo and set up `multiSensorPluginTest.world`.

Configure and Perform Gazebo Co-Simulation

Initialize connection settings and check connectivity with the Gazebo plugin running on `192.168.198.129` and port `14581`.

```
gzinit("192.168.198.129",14581)
```

Assign and Retrieve Gazebo Model Information

List the models available in the Gazebo world.

```
modelList = gzmodel("list")
```

```
modelList = 1×11 string
    "ground_plane"    "unit_box"    "camera0"    "camera1"    "depth_camera0"    "depth_camera1"
```

Assign values to the `Position` and `SelfCollide` parameters of the `unit_box` model.

```
[status,message] = gzmodel("set","unit_box","Position",[2 2 0.5],"SelfCollide","on")
```

```
status = 1×2 logical array

   1   1

message = 1×2 string
    "Position parameter set successfully."    "SelfCollide parameter set successfully."
```

Retrieve the values of the `Position` and `SelfCollide` parameters of the `unit_box` model.

```
[position,selfcollide] = gzmodel("get","unit_box","Position","SelfCollide")
```

```
position = 1×3

                         2                         2           0.4999999999951

selfcollide = logical
   1
```

Assign and Retrieve Gazebo Model Link Information

List the links available in the `unit_box` model.

```
linkList = gzlink("list","unit_box")
```

```
linkList = 
"link"
```

Assign values to the link parameters `Mass` and `Gravity` of the link `link` in the `unit_box` model.

```
[status,message] = gzlink("set","unit_box","link","Mass",2,"Gravity","off")
```

```
status = 1×2 logical array

   1   1


message = 1×2 string
    "Mass parameter set successfully."    "Gravity parameter set successfully."
```

Retrieve the values of the link parameters `Mass` and `Gravity` of the link `link` in the `unit_box` model.

```
[mass,gravity] = gzlink("get","unit_box","link","Mass","Gravity")
```

```
mass = 
     2


gravity = logical
   0
```

Assign and Retrieve Gazebo Model Joint Information

List the joints available in the `unit_box` model.

```
jointList = gzjoint("list","unit_box")
```

```
jointList = 
"joint"
```

Assign a value to the joint parameter `Damping` of the axis `Axis0` for the joint `joint` in the `unit_box` model.

```
[status,message] = gzjoint("set","unit_box","joint","Axis","0","Damping",0.25)
```

```
status = logical
   1


message = 
"Damping parameter set successfully."
```

Retrieve the value of the joint parameter `Damping` of the axis `Axis0` for the joint `joint` in the `unit_box` model.

```
damping = gzjoint("get","unit_box","joint","Axis0","Damping")
```

```
damping = 
                      0.25
```

Reset all Gazebo model configurations.

```
gzworld("reset")
```

Input Arguments

modelname — Gazebo model name

string scalar | character vector

Gazebo model name, specified as a string scalar or character vector.

Data Types: char | string

params — Gazebo model parameters

string scalars | character vectors

Gazebo model parameters, specified as a comma-separated list of string scalars or character vectors. Specify the list of parameters you want to retrieve the values, from this table.

Parameters	Description
"EnableWind"	Get the wind parameter of the Gazebo model.
"IsStatic"	Get the IsStatic parameter of the Gazebo model.
"Orientation"	Get the orientation parameter of the Gazebo model.
"Position"	Get the position parameter of the Gazebo model.
"SelfCollide"	Get the SelfCollide parameter of the Gazebo model.

Example: [position,selfcollide] = gzmodel("get","unit_box","Position","SelfCollide")

Data Types: char | string

Name-Value Pair Arguments

Specify optional pairs of arguments as Name1=Value1,...,NameN=ValueN, where Name is the argument name and Value is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose Name *in quotes.*

Example: For example, gzmodel("set","unit_box","Position",[2 2 0.5]) sets the position of the model unit_box.

EnableWind — Gazebo model wind parameter

'on' | 'off'

Gazebo model wind parameter, specified as the comma-separated pair consisting of 'EnableWind' and either 'on' or 'off'.

Example: [status,message] = gzmodel("set","unit_box","EnableWind","off");

Data Types: char | string

IsStatic — Gazebo model IsStatic parameter

'on' | 'off'

Gazebo model IsStatic parameter, specified as the comma-separated pair consisting of `'IsStatic'` and either `'on'` or `'off'`.

Example: `[status,message] = gzmodel("set","unit_box","IsStatic","off");`

Data Types: `char` | `string`

`Orientation` — Gazebo model orientation parameter
four-element vector

Gazebo model orientation parameter, specified as the comma-separated pair consisting of `'Orientation'` and a four-element quaternion vector of the form [*w x y z*].

Example: `[status,message] = gzmodel("set","unit_box","Orientation",[1 0 0 0]);`

Data Types: `single` | `double`

`Position` — Gazebo model position
three-element vector

Gazebo model position parameter, specified as the comma-separated pair consisting of `'Position'` and a three-element vector of the form [*x y z*] in meters.

Example: `[status,message] = gzmodel("set","unit_box","Position",[0 0 0]);`

Data Types: `single` | `double`

`SelfCollide` — Gazebo model SelfCollide parameter
`'on'` | `'off'`

Gazebo model SelfCollide parameter, specified as the comma-separated pair consisting of `'SelfCollide'` and either `'on'` or `'off'`.

Example: `[status,message] = gzmodel("set","unit_box","SelfCollide","off");`

Data Types: `char` | `string`

Output Arguments

`List` — List of models
cell array of character vectors

List of models, returned as a cell array of character vectors.

`Links` — List of links in model
cell array of character vectors

List of links in the model, returned as a cell array of character vectors.

`Joints` — List of joints in model
cell array of character vectors

List of joints in the model, returned as a cell array of character vectors.

`Status` — Status of values assigned to parameters
logical array

Status of the values assigned to the parameters, returned as a logical array.

`Message` — Success or failure message
string array

Success or failure message, returned as a string array.

`Output1,...,OutputN` — Values of specified parameters
logical scalar | numeric vector

Values of specified parameters, returned as a logical or numeric vector based on the specified parameters. The following table shows the returned data type of parameter values.

Parameters	Description
`"EnableWind"`	Gazebo model wind parameter, returns a logical scalar.
`"IsStatic"`	Gazebo model IsStatic parameter, returns a logical scalar.
`"Orientation"`	Gazebo model orientation parameter, returns a four-element quaternion vector of the form [w x y z].
`"Position"`	Gazebo model position parameter, returns a three-element vector of the form [x y z] in meters.
`"SelfCollide"`	Gazebo model SelfCollide parameter, returns a logical scalar.

Limitations

- `gzmodel` function not supported with MATLAB Compiler.

Version History

Introduced in R2021a

See Also

`gzinit` | `gzlink` | `gzjoint` | `gzworld`

gzworld

Interact with Gazebo world

Syntax

```
gzworld("reset")
```

Description

`gzworld("reset")` resets all Gazebo model configurations and Gazebo simulation time.

Examples

Perform Co-Simulation Between MATLAB and Gazebo

Set up a simulation between MATLAB and Gazebo, receive data from Gazebo, and send commands to Gazebo.

Prerequisite

Follow the instructions in "Perform Co-Simulation between Simulink and Gazebo" to download the Linux virtual machine (VM) with Gazebo and set up `multiSensorPluginTest.world`.

Configure and Perform Gazebo Co-Simulation

Initialize connection settings and check connectivity with the Gazebo plugin running on `192.168.198.129` and port `14581.`

```
gzinit("192.168.198.129",14581)
```

Assign and Retrieve Gazebo Model Information

List the models available in the Gazebo world.

```
modelList = gzmodel("list")
```

```
modelList = 1×11 string
    "ground_plane"    "unit_box"    "camera0"    "camera1"    "depth_camera0"    "depth_camera1"
```

Assign values to the `Position` and `SelfCollide` parameters of the `unit_box` model.

```
[status,message] = gzmodel("set","unit_box","Position",[2 2 0.5],"SelfCollide","on")
```

```
status = 1×2 logical array

   1   1

message = 1×2 string
    "Position parameter set successfully."    "SelfCollide parameter set successfully."
```

Retrieve the values of the Position and SelfCollide parameters of the unit_box model.

```
[position,selfcollide] = gzmodel("get","unit_box","Position","SelfCollide")
```

```
position = 1×3

                         2                          2           0.4999999999951

selfcollide = logical
   1
```

Assign and Retrieve Gazebo Model Link Information

List the links available in the unit_box model.

```
linkList = gzlink("list","unit_box")
```

```
linkList = 
"link"
```

Assign values to the link parameters Mass and Gravity of the link link in the unit_box model.

```
[status,message] = gzlink("set","unit_box","link","Mass",2,"Gravity","off")
```

```
status = 1×2 logical array

   1   1

message = 1×2 string
    "Mass parameter set successfully."    "Gravity parameter set successfully."
```

Retrieve the values of the link parameters Mass and Gravity of the link link in the unit_box model.

```
[mass,gravity] = gzlink("get","unit_box","link","Mass","Gravity")
```

```
mass = 
     2

gravity = logical
   0
```

Assign and Retrieve Gazebo Model Joint Information

List the joints available in the unit_box model.

```
jointList = gzjoint("list","unit_box")
```

```
jointList = 
"joint"
```

Assign a value to the joint parameter Damping of the axis Axis0 for the joint joint in the unit_box model.

```
[status,message] = gzjoint("set","unit_box","joint","Axis","0","Damping",0.25)
```

```
status = logical
   1


message = 
"Damping parameter set successfully."
```

Retrieve the value of the joint parameter `Damping` of the axis `Axis0` for the joint `joint` in the `unit_box` model.

```
damping = gzjoint("get","unit_box","joint","Axis0","Damping")
```

```
damping = 
                      0.25
```

Reset all Gazebo model configurations.

```
gzworld("reset")
```

Limitations

- `gzworld` function not supported with MATLAB Compiler.

Version History

Introduced in R2021a

See Also

`gzinit` | `gzlink` | `gzjoint` | `gzmodel`

hom2cart

Convert homogeneous coordinates to Cartesian coordinates

Syntax

```
cart = hom2cart(hom)
```

Description

`cart = hom2cart(hom)` converts a set of homogeneous points to Cartesian coordinates.

Examples

Convert Homogeneous Points to 3-D Cartesian Points

```
h = [0.2785 0.9575 0.1576 0.5; 0.5469 0.9649 0.9706 0.5];
c = hom2cart(h)
```

```
c = 2×3

    0.5570    1.9150    0.3152
    1.0938    1.9298    1.9412
```

Input Arguments

`hom` — Homogeneous points
n-by-*k* matrix

Homogeneous points, returned as an *n*-by-*k* matrix, containing *n* points. *k* must be greater than or equal to 2.

Example: `[0.2785 0.9575 0.1576 0.5; 0.5469 0.9649 0.9706 0.5]`

Output Arguments

`cart` — Cartesian coordinates
n-by-(*k*-1) matrix

Cartesian coordinates, specified as an *n*-by-(*k*-1) matrix, containing *n* points. Each row of `cart` represents a point in *k*-dimensional space. *k* must be greater than or equal to 1.

Example: `[0.8147 0.1270 0.6324; 0.9058 0.9134 0.0975]`

Version History

Introduced in R2015a

R2023a: `hom2cart` Supports 2-D Homogeneous Points

The `hom` argument now accepts 2-D homogeneous points and `hom2cart` outputs 2-D Cartesian coordinates.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

`cart2hom`

Topics
“Coordinate Transformations in Robotics”

importrobot

Import rigid body tree model from URDF, Xacro, SDF file, text, or Simscape Multibody model

Syntax

```
robot = importrobot(filename)
robot = importrobot(text)
robot = importrobot( ___ ,format)
robot = importrobot( ___ ,Name,Value)

[robot,importInfo] = importrobot(model)
[robot,importInfo] = importrobot( ___ ,Name,Value)
```

Description

URDF, Xacro, or SDF Import

`robot = importrobot(filename)` returns a `rigidBodyTree` object by parsing the Unified Robot Description Format (URDF), XML Macros (Xacro), or Simulation Description Format (SDF) file specified by `filename`.

`robot = importrobot(text)` parses robot description from URDF, Xacro, or SDF text.

`robot = importrobot( ___ ,format)` explicitly specifies the type of the robot description in addition to any combination of input arguments from previous syntaxes. If the format of the text file does not match the format specified in the `format` argument, the function returns an error.

`robot = importrobot( ___ ,Name,Value)` specifies options using one or more name-value pair arguments in addition to any combination of input arguments from previous syntaxes. Use the "URDF, Xacro, or SDF Import" on page 2-0 name-value pairs to import a model from URDF, Xacro, or SDF file, or text.

Simscape Multibody Model Import

`[robot,importInfo] = importrobot(model)` imports a Simscape Multibody model and returns an equivalent `rigidBodyTree` object and information about the import in `importInfo`. Only fixed, prismatic, and revolute joints are supported in the output `rigidBodyTree` object.

`[robot,importInfo] = importrobot( ___ ,Name,Value)` specifies options using one or more name-value pair arguments in addition to the Simscape Multibody model from the previous syntax. Use the "Simscape Multibody Model Import" on page 2-0 name-value pairs to import a model that uses other joint types, constraint blocks, or variable inertias.

Examples

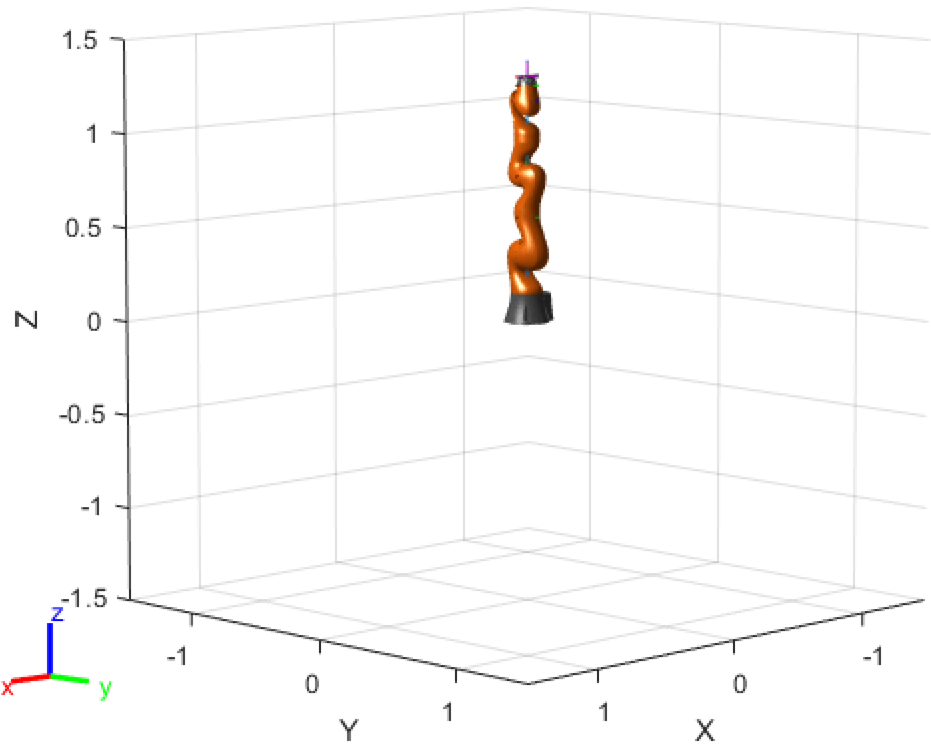
Import Robot from URDF File

Import a URDF file as a `rigidBodyTree` object.

```
robot = importrobot('iiwa14.urdf')
```

```
robot = 
  rigidBodyTree with properties:

     NumBodies: 10
        Bodies: {[1x1 rigidBody]  [1x1 rigidBody]  [1x1 rigidBody]  [1x1 rigidBody]  [1x1 rigidBo
          Base: [1x1 rigidBody]
     BodyNames: {'iiwa_link_0'  'iiwa_link_1'  'iiwa_link_2'  'iiwa_link_3'  'iiwa_link_4'  'iiwa
      BaseName: 'world'
       Gravity: [0 0 0]
    DataFormat: 'struct'
```

```
show(robot)
```




```
ans = 
  Axes (Primary) with properties:

             XLim: [-1.5000 1.5000]
             YLim: [-1.5000 1.5000]
           XScale: 'linear'
           YScale: 'linear'
    GridLineStyle: '-'
         Position: [0.1300 0.1100 0.7750 0.8150]
            Units: 'normalized'

  Show all properties
```

Import Robot from URDF Character Vector

Specify the URDF character vector. This character vector is a minimalist description for creating a valid robot model.

```
URDFtext = '<?xml version="1.0" ?><robot name="min"><link name="L0"/></robot>';
```

Import the robot model. The description creates a `rigidBodyTree` object that has only a robot base link named `'L0'`.

```
robot = importrobot(URDFtext)
```

```
robot = 
  rigidBodyTree with properties:

     NumBodies: 0
        Bodies: {1x0 cell}
          Base: [1x1 rigidBody]
     BodyNames: {1x0 cell}
      BaseName: 'L0'
       Gravity: [0 0 0]
    DataFormat: 'struct'
```

Display Robot Model with Visual Geometries

You can import robots that have `.stl` files associated with the Unified Robot Description format (URDF) file to describe the visual geometries of the robot. Each rigid body has an individual visual geometry specified. The `importrobot` function parses the URDF file to get the robot model and visual geometries. The function assumes that visual geometry and collision geometry of the robot are the same and assigns the visual geometries as collision geometries of corresponding bodies.

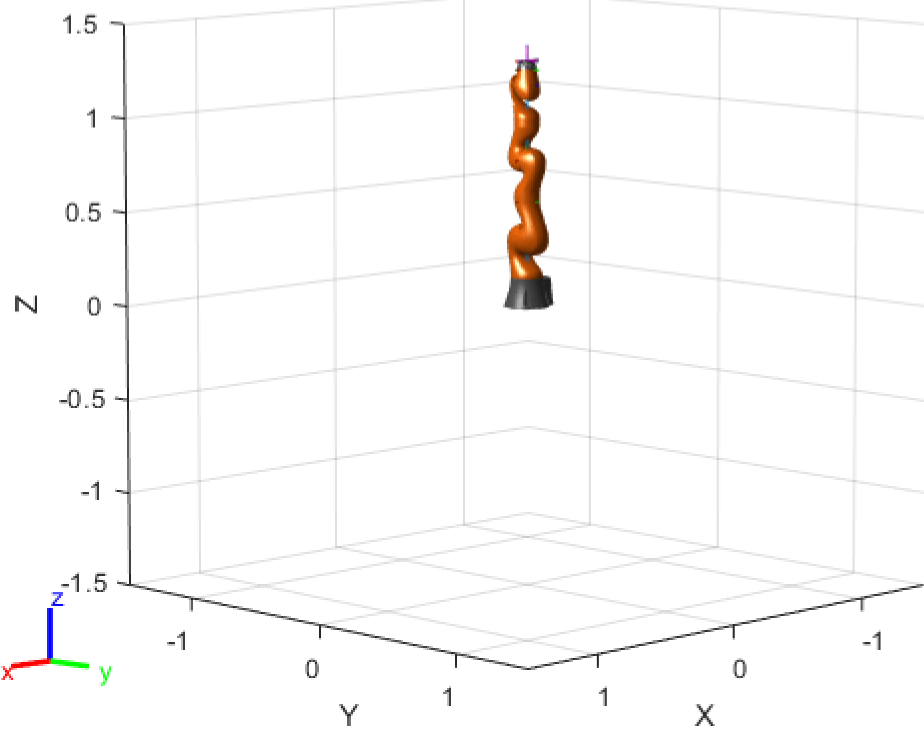
Use the `show` function to display the visual and collision geometries of the robot model in a figure. You can then interact with the model by clicking components to inspect them and right-clicking to toggle visibility.

Import a robot model as a URDF file. The `.stl` file locations must be properly specified in this URDF. To add other `.stl` files to individual rigid bodies, see `addVisual`.

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

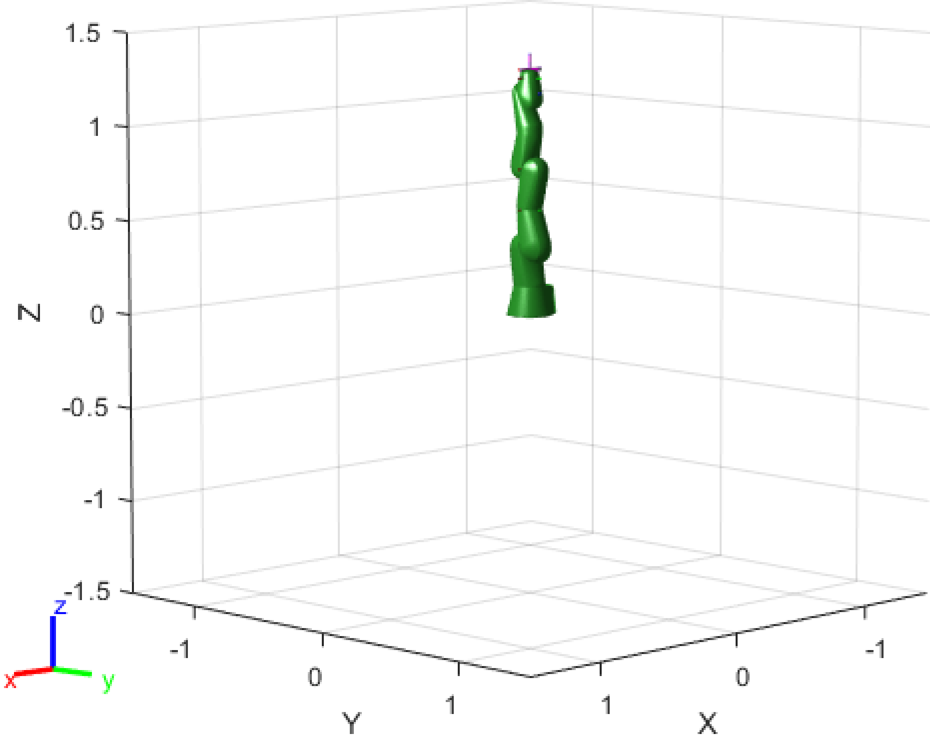
Visualize the robot with the associated visual model. Click bodies or frames to inspect them. Right-click bodies to toggle visibility for each visual geometry.

```
show(robot,'visuals','on','collision','off');
```



Visualize the robot with the associated collision geometries. Click bodies or frames to inspect them. Right-click bodies to toggle visibility for each collision geometry.

```
show(robot,'visuals','off','collision','on');
```



Import Simscape™ Multibody™ model to `RigidBodyTree` Object

Import an existing Simscape™ Multibody™ robot model into the Robotics System Toolbox™ as a `rigidBodyTree` object.

Open the Simscape™ Multibody™ model. This is a model for a humanoid robot.

```
open_system('example_smhumanoidrobot.slx')
```

Import the model.

```
[robot,importInfo] = importrobot(gcs)
```

```
robot = 
  rigidBodyTree with properties:

     NumBodies: 21
        Bodies: {1x21 cell}
          Base: [1x1 rigidBody]
     BodyNames: {'Body01'  'Body02'  'Body03'  'Body04'  'Body05'  'Body06'  'Body07'  'Body08'
      BaseName: 'Base'
       Gravity: [0 0 -9.8066]
    DataFormat: 'struct'
```

```
importInfo =
  rigidBodyTreeImportInfo with properties:

        SourceModelName: 'example_smhumanoidrobot'
          RigidBodyTree: [1x1 rigidBodyTree]
    BlockConversionInfo: [1x1 struct]
```

Display details about the created `rigidBodyTree` object.

```
showdetails(importInfo)
```

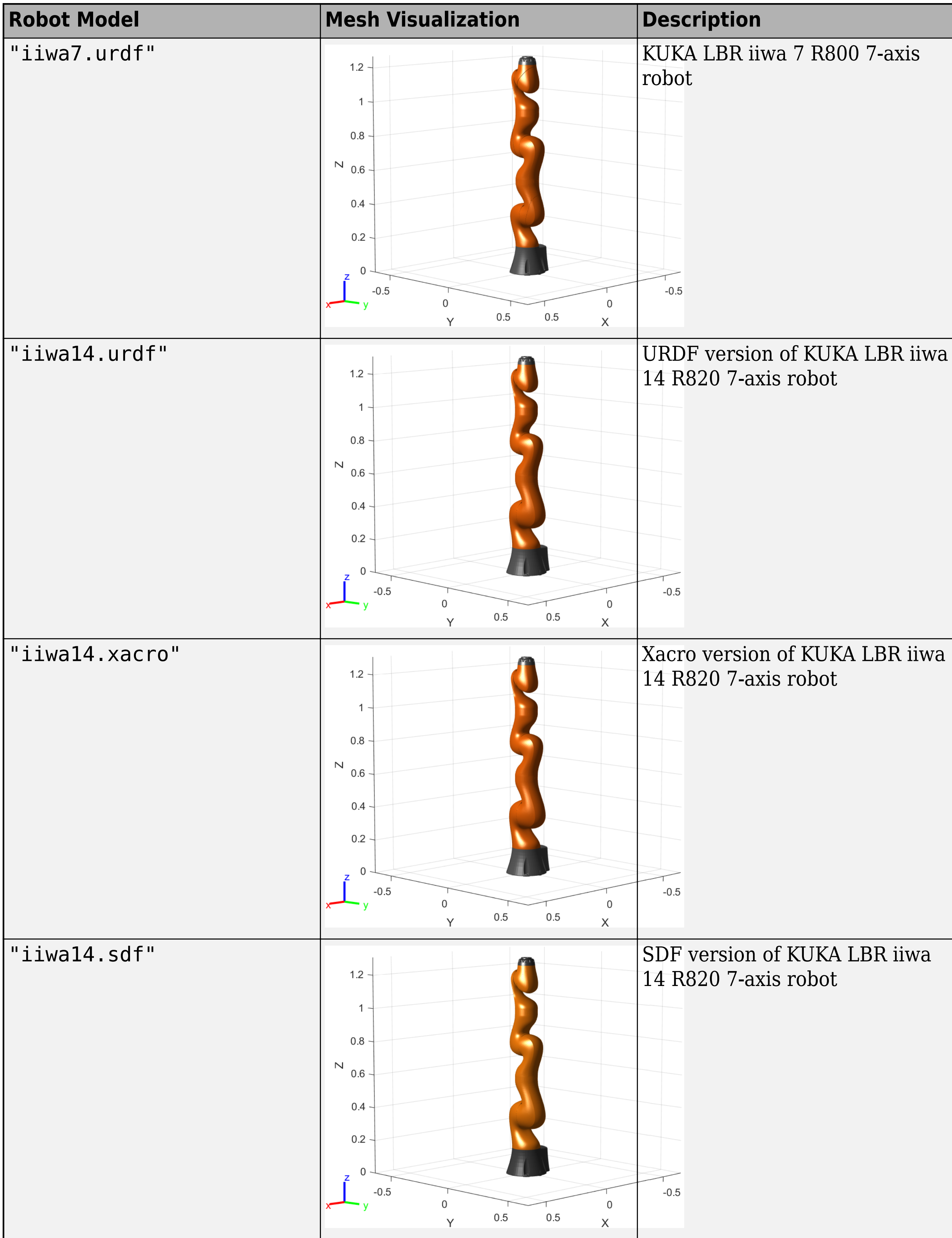
```
--------------------
Robot: (21 bodies)

 Idx     Body Name     Simulink Source Blocks     Joint Name     Simulink Source Blocks   Joint T
 ---     ---------     ----------------------     ----------     ----------------------   -------
   1        Body01    Info | List | Highlight        Joint01    Info | List | Highlight     revol
   2        Body02    Info | List | Highlight        Joint02    Info | List | Highlight     revol
   3        Body03    Info | List | Highlight        Joint03    Info | List | Highlight     revol
   4        Body04    Info | List | Highlight        Joint04    Info | List | Highlight     revol
   5        Body05    Info | List | Highlight        Joint05    Info | List | Highlight     revol
   6        Body06    Info | List | Highlight        Joint06    Info | List | Highlight     revol
   7        Body07    Info | List | Highlight        Joint07    Info | List | Highlight     revol
   8        Body08    Info | List | Highlight        Joint08    Info | List | Highlight     revol
   9        Body09    Info | List | Highlight        Joint09    Info | List | Highlight     revol
  10        Body10    Info | List | Highlight        Joint10    Info | List | Highlight     revol
  11        Body11    Info | List | Highlight        Joint11    Info | List | Highlight     revol
  12        Body12    Info | List | Highlight        Joint12    Info | List | Highlight     revol
  13        Body13    Info | List | Highlight        Joint13    Info | List | Highlight     revol
  14        Body14    Info | List | Highlight        Joint14    Info | List | Highlight     revol
  15        Body15    Info | List | Highlight        Joint15    Info | List | Highlight     revol
  16        Body16    Info | List | Highlight        Joint16    Info | List | Highlight     revol
  17        Body17    Info | List | Highlight        Joint17    Info | List | Highlight     revol
  18        Body18    Info | List | Highlight        Joint18    Info | List | Highlight     revol
  19        Body19    Info | List | Highlight        Joint19    Info | List | Highlight        fi
  20        Body20    Info | List | Highlight        Joint20    Info | List | Highlight        fi
  21        Body21    Info | List | Highlight        Joint21    Info | List | Highlight        fi
--------------------
```

Input Arguments

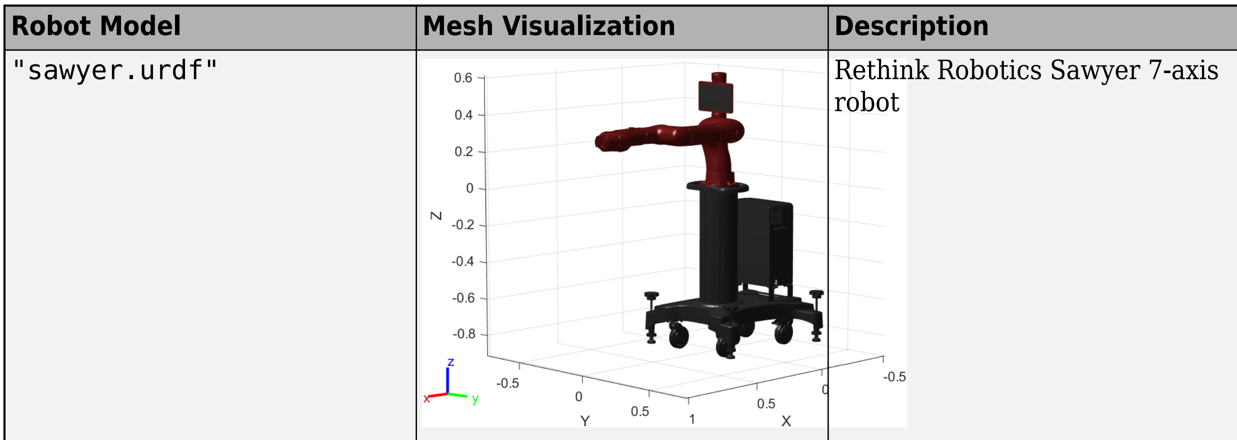
`filename` — Name of robot description file
string scalar | character vector

Name of the robot description file, specified as a string scalar or character vector. This file must be a valid URDF robot description, Xacro robot description, or SDF model description.

Included Robot Models with Mesh Data



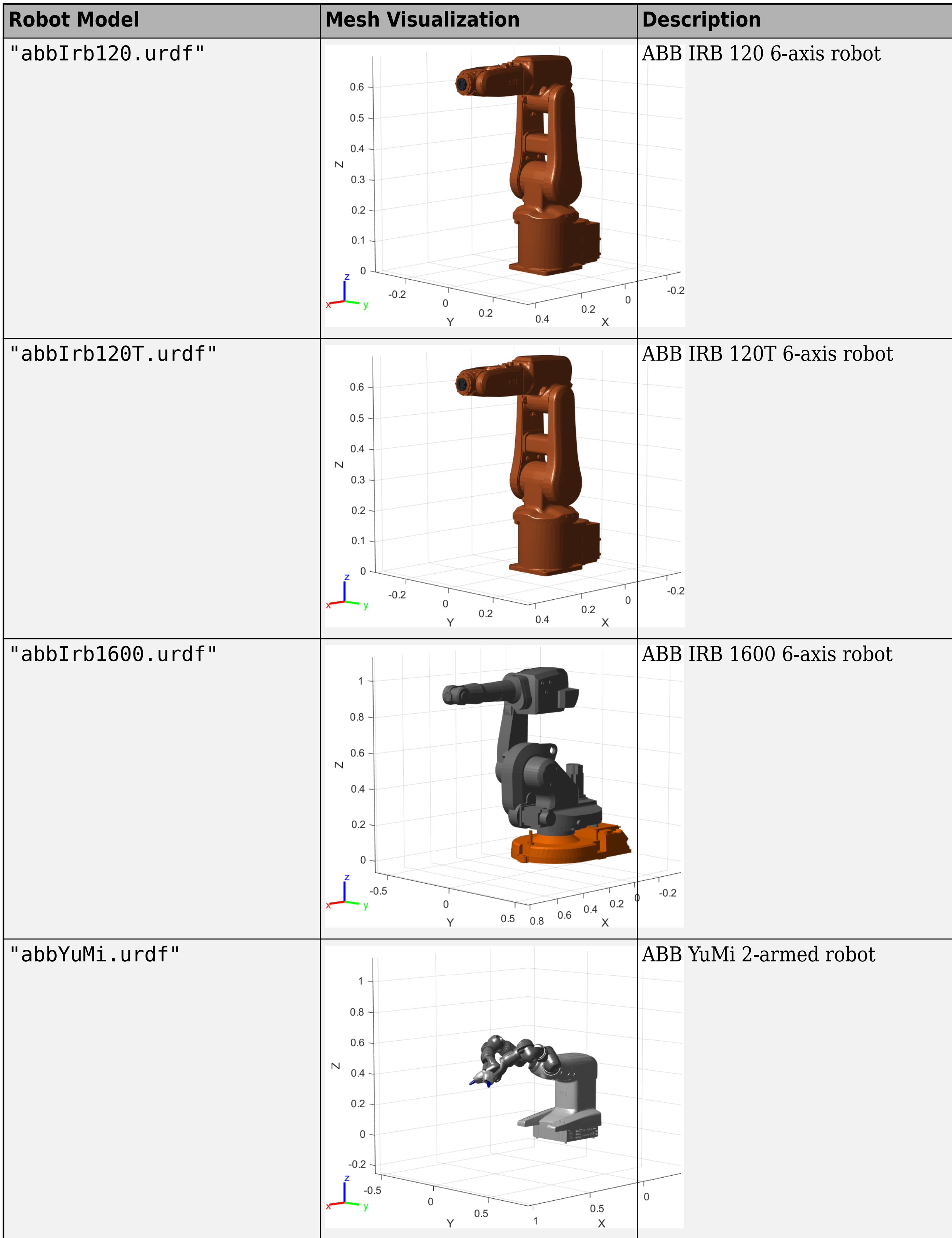
Robot Model	Mesh Visualization	Description
`"iiwa7.urdf"`		KUKA LBR iiwa 7 R800 7-axis robot
`"iiwa14.urdf"`		URDF version of KUKA LBR iiwa 14 R820 7-axis robot
`"iiwa14.xacro"`		Xacro version of KUKA LBR iiwa 14 R820 7-axis robot
`"iiwa14.sdf"`		SDF version of KUKA LBR iiwa 14 R820 7-axis robot

Robot Model	Mesh Visualization	Description
`"sawyer.urdf"`		Rethink Robotics Sawyer 7-axis robot

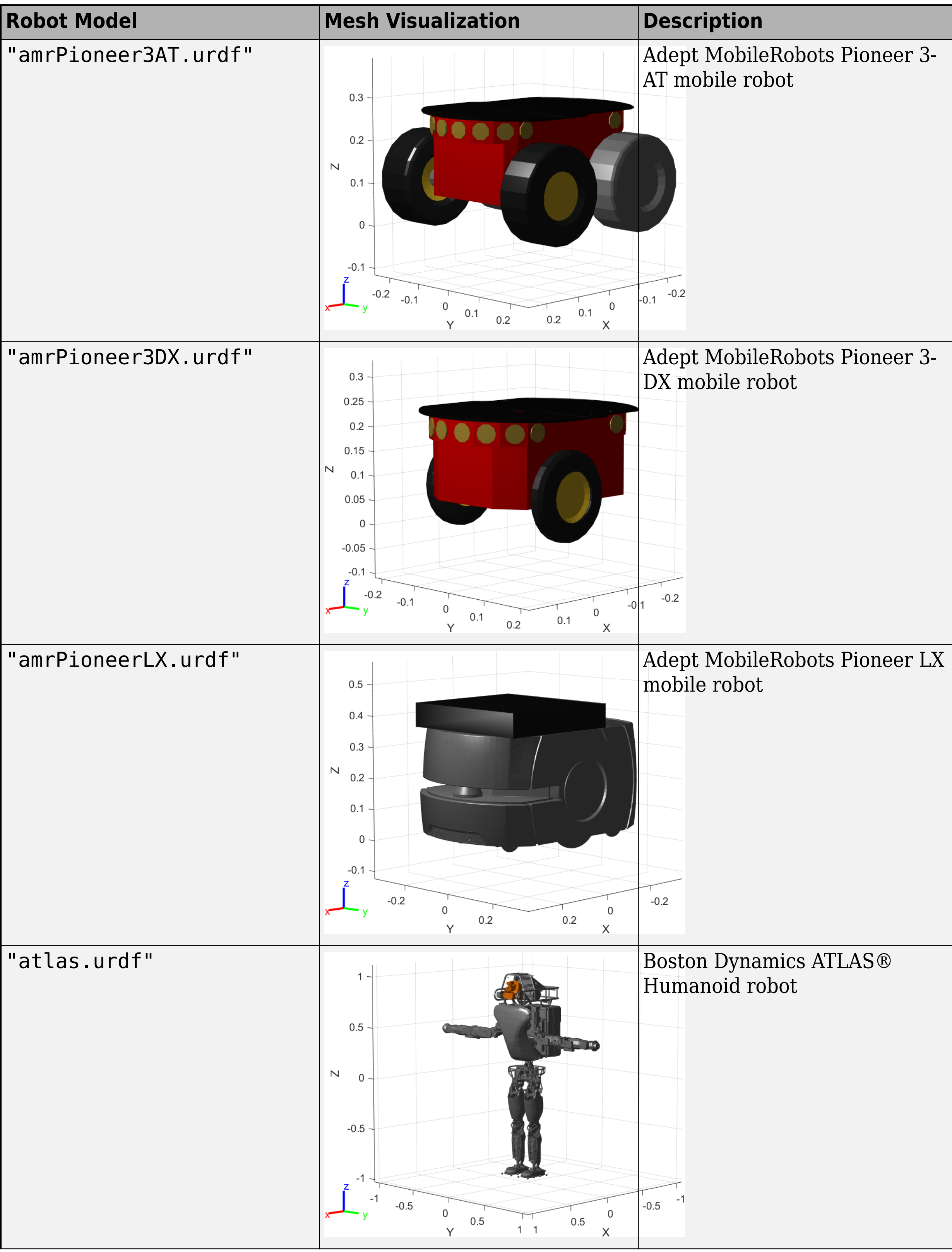
---

Note To download the mesh data for the included robot models without the mesh data, see “Install Robotics System Toolbox Robot Library Data Support Package”.

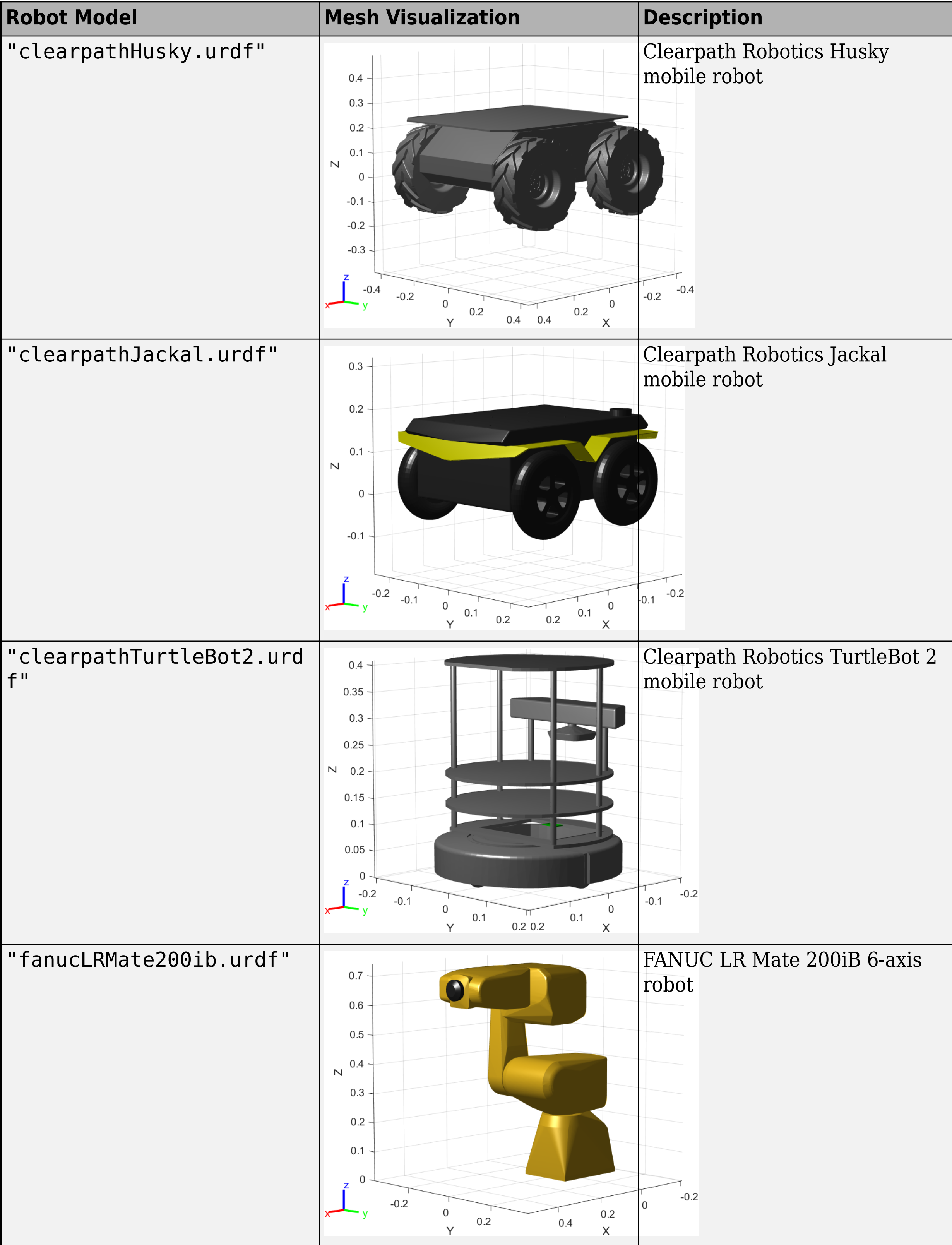
Included Robot Models without Mesh Data



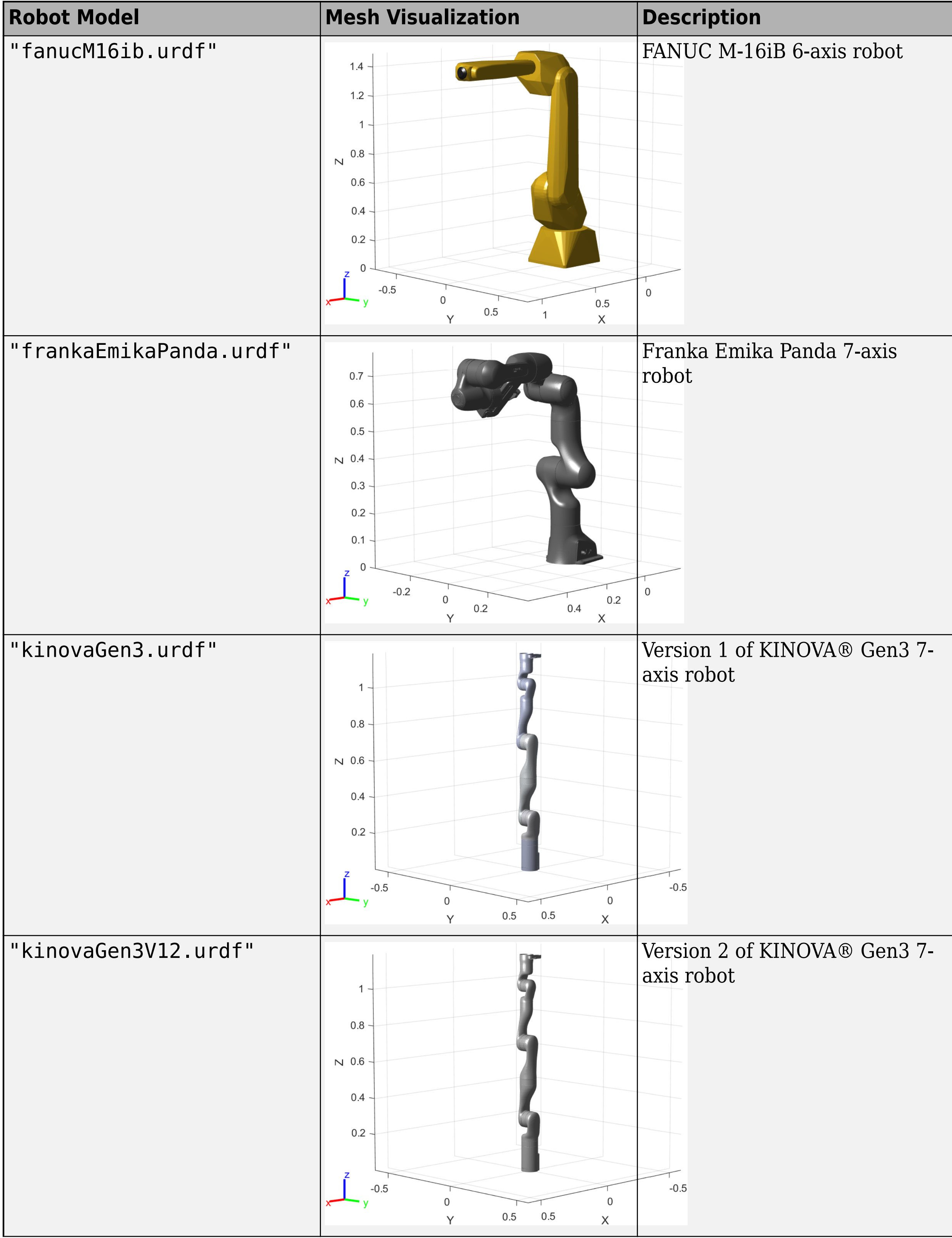
Robot Model	Mesh Visualization	Description
"abbIrb120.urdf"		ABB IRB 120 6-axis robot
"abbIrb120T.urdf"		ABB IRB 120T 6-axis robot
"abbIrb1600.urdf"		ABB IRB 1600 6-axis robot
"abbYuMi.urdf"		ABB YuMi 2-armed robot



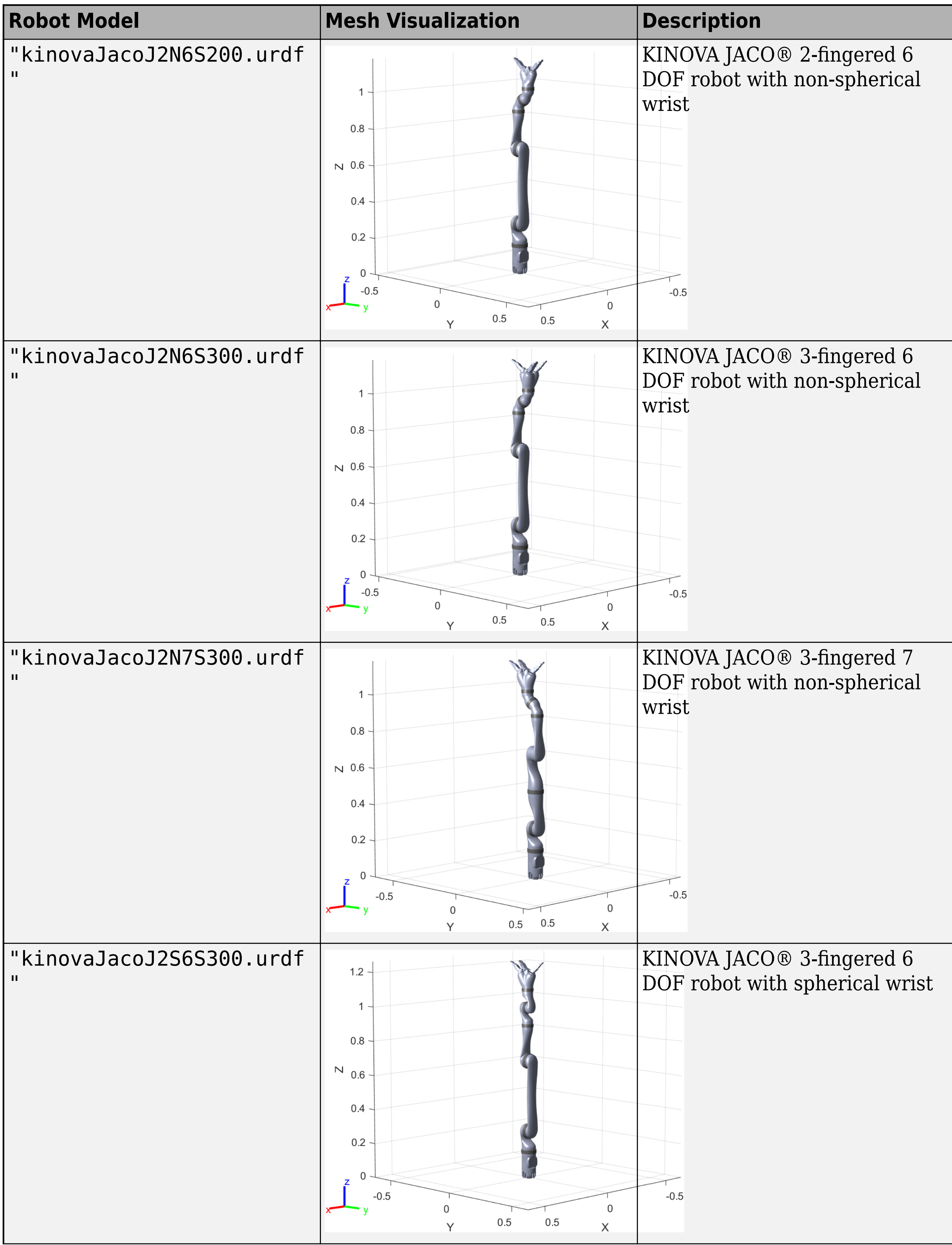
Robot Model	Mesh Visualization	Description
"amrPioneer3AT.urdf"		Adept MobileRobots Pioneer 3-AT mobile robot
"amrPioneer3DX.urdf"		Adept MobileRobots Pioneer 3-DX mobile robot
"amrPioneerLX.urdf"		Adept MobileRobots Pioneer LX mobile robot
"atlas.urdf"		Boston Dynamics ATLAS® Humanoid robot



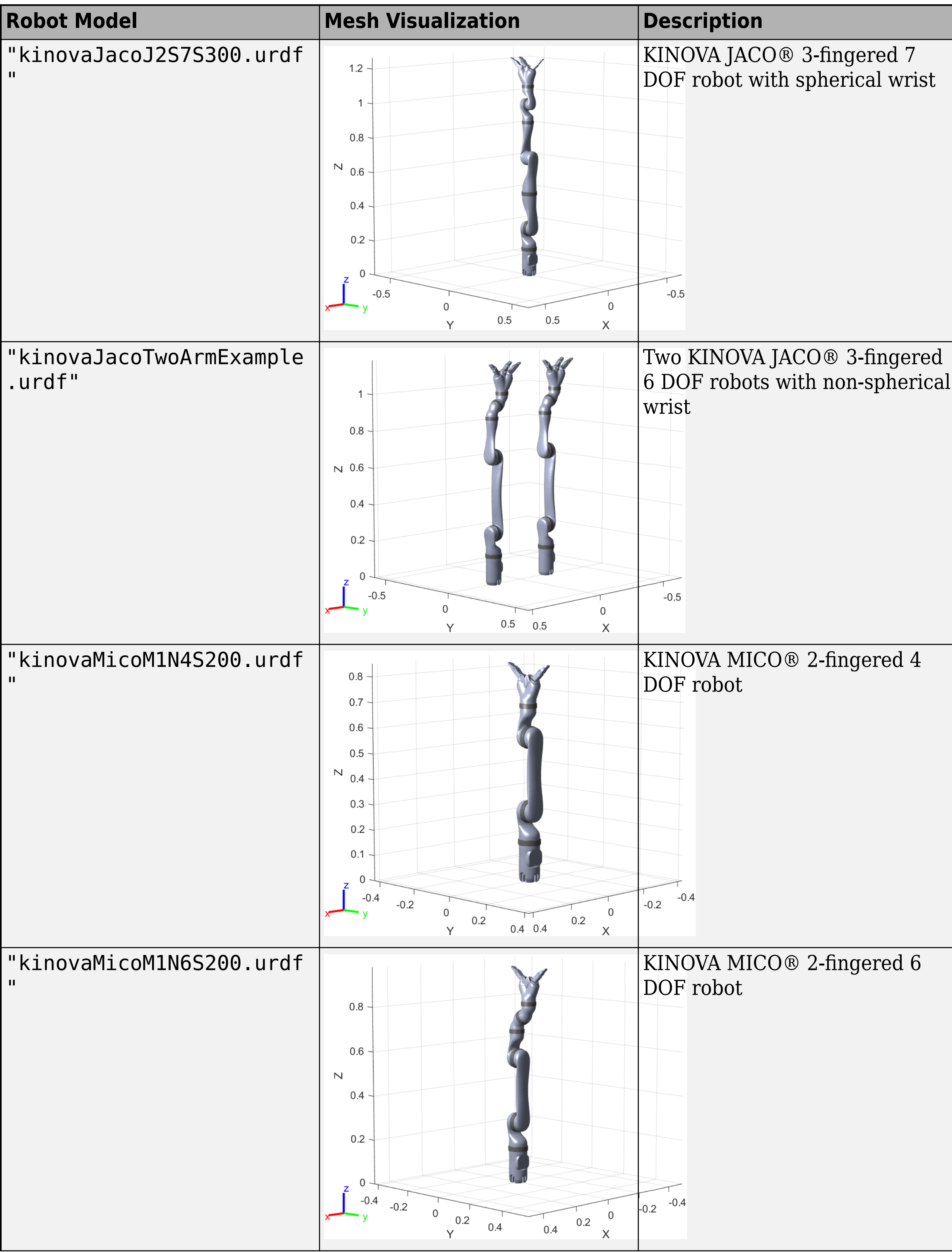
Robot Model	Mesh Visualization	Description
"clearpathHusky.urdf"		Clearpath Robotics Husky mobile robot
"clearpathJackal.urdf"		Clearpath Robotics Jackal mobile robot
"clearpathTurtleBot2.urdf"		Clearpath Robotics TurtleBot 2 mobile robot
"fanucLRMate200ib.urdf"		FANUC LR Mate 200iB 6-axis robot



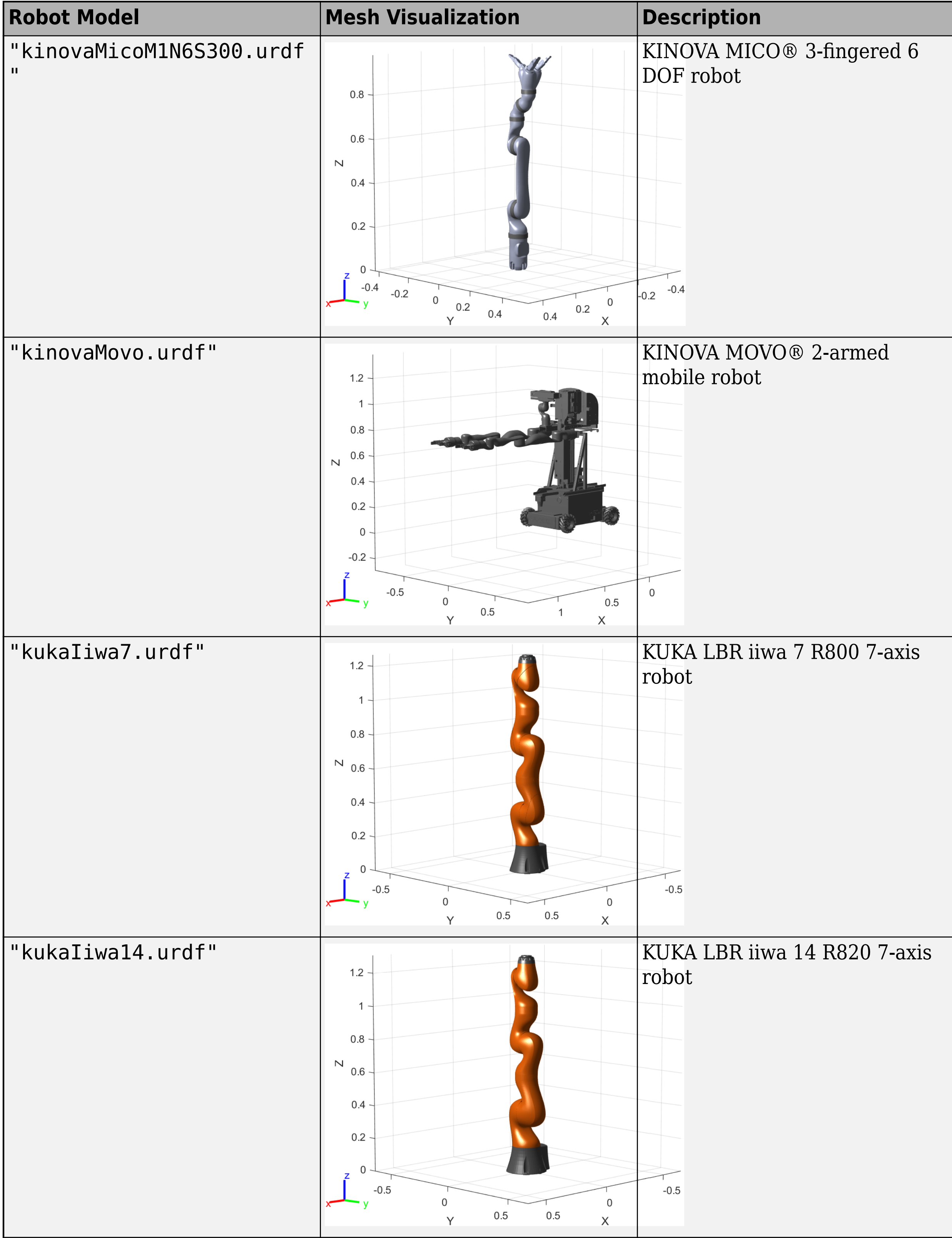
Robot Model	Mesh Visualization	Description
`"fanucM16ib.urdf"`		FANUC M-16iB 6-axis robot
`"frankaEmikaPanda.urdf"`		Franka Emika Panda 7-axis robot
`"kinovaGen3.urdf"`		Version 1 of KINOVA® Gen3 7-axis robot
`"kinovaGen3V12.urdf"`		Version 2 of KINOVA® Gen3 7-axis robot



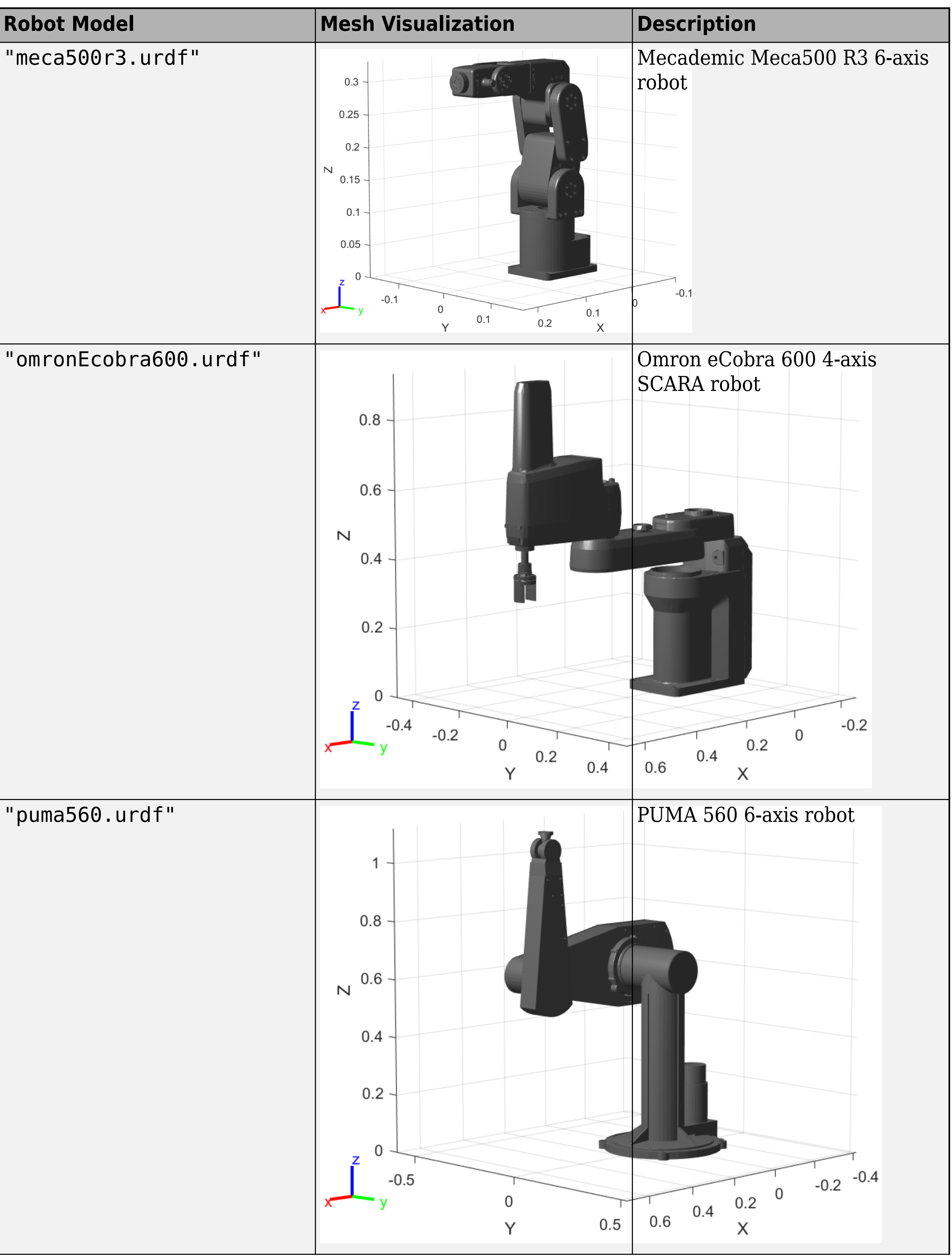
Robot Model	Mesh Visualization	Description
"kinovaJacoJ2N6S200.urdf"		KINOVA JACO® 2-fingered 6 DOF robot with non-spherical wrist
"kinovaJacoJ2N6S300.urdf"		KINOVA JACO® 3-fingered 6 DOF robot with non-spherical wrist
"kinovaJacoJ2N7S300.urdf"		KINOVA JACO® 3-fingered 7 DOF robot with non-spherical wrist
"kinovaJacoJ2S6S300.urdf"		KINOVA JACO® 3-fingered 6 DOF robot with spherical wrist

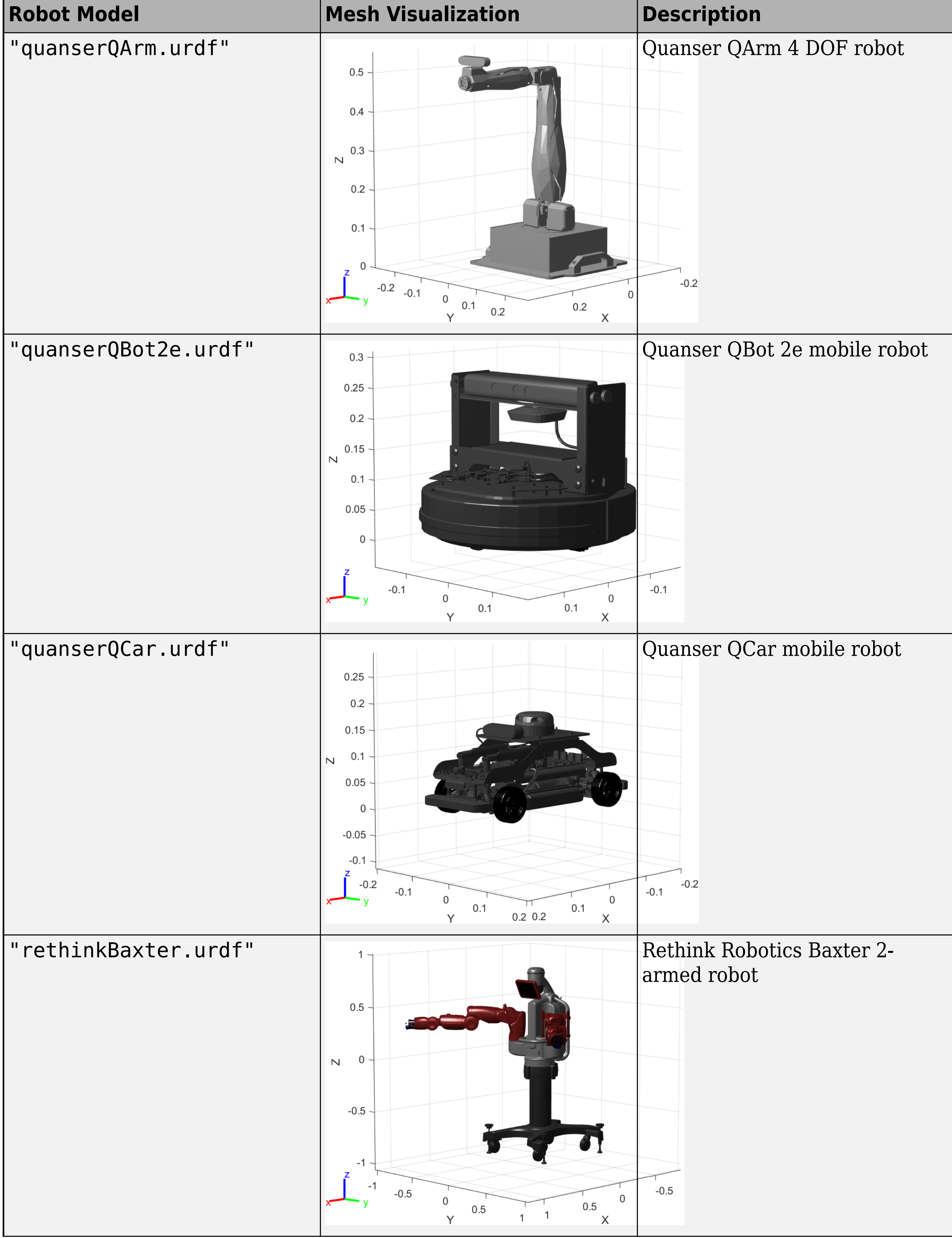


Robot Model	Mesh Visualization	Description
"kinovaJacoJ2S7S300.urdf"		KINOVA JACO® 3-fingered 7 DOF robot with spherical wrist
"kinovaJacoTwoArmExample.urdf"		Two KINOVA JACO® 3-fingered 6 DOF robots with non-spherical wrist
"kinovaMicoM1N4S200.urdf"		KINOVA MICO® 2-fingered 4 DOF robot
"kinovaMicoM1N6S200.urdf"		KINOVA MICO® 2-fingered 6 DOF robot

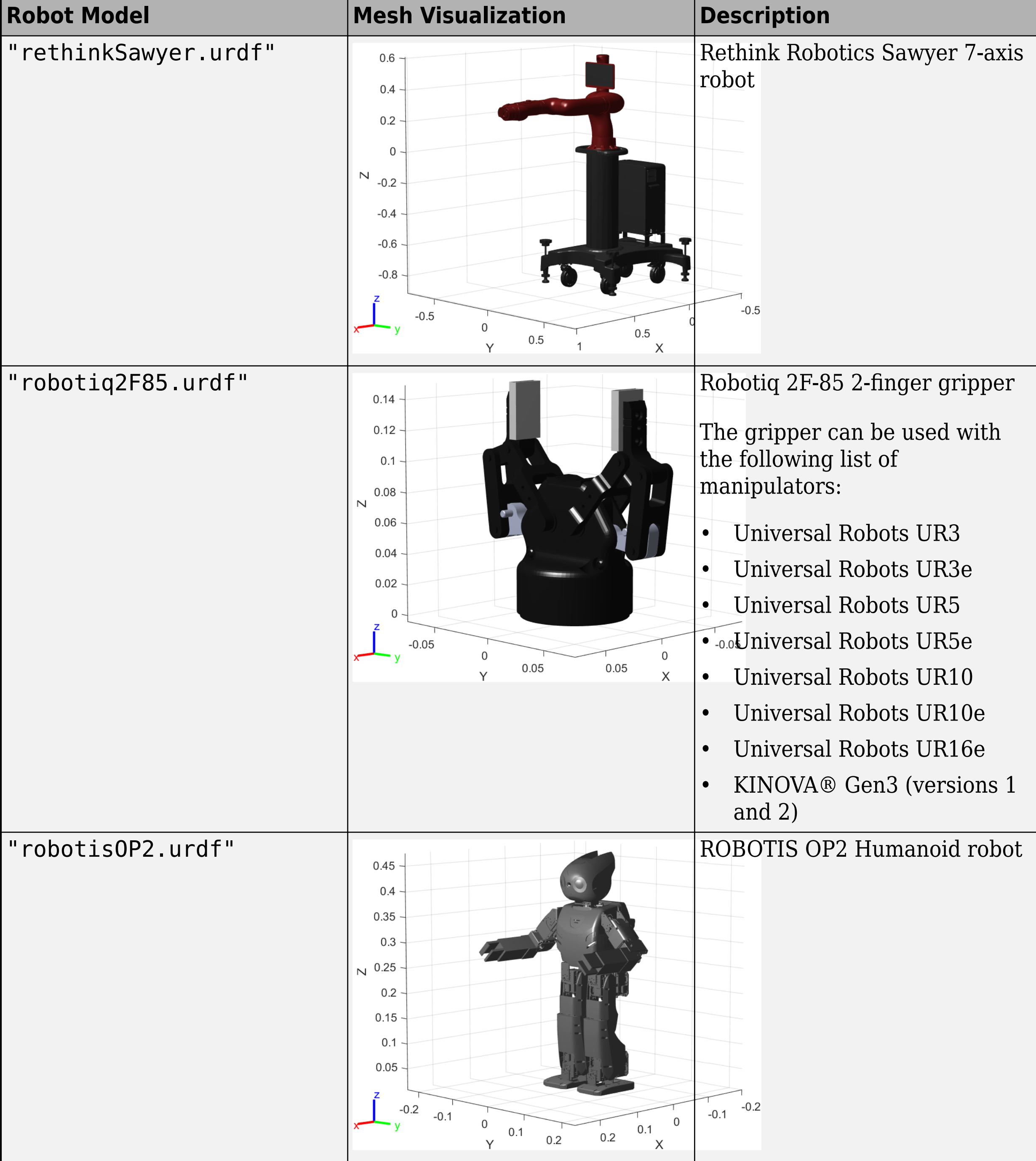


Robot Model	Mesh Visualization	Description
"kinovaMicoM1N6S300.urdf"		KINOVA MICO® 3-fingered 6 DOF robot
"kinovaMovo.urdf"		KINOVA MOVO® 2-armed mobile robot
"kukaIiwa7.urdf"		KUKA LBR iiwa 7 R800 7-axis robot
"kukaIiwa14.urdf"		KUKA LBR iiwa 14 R820 7-axis robot

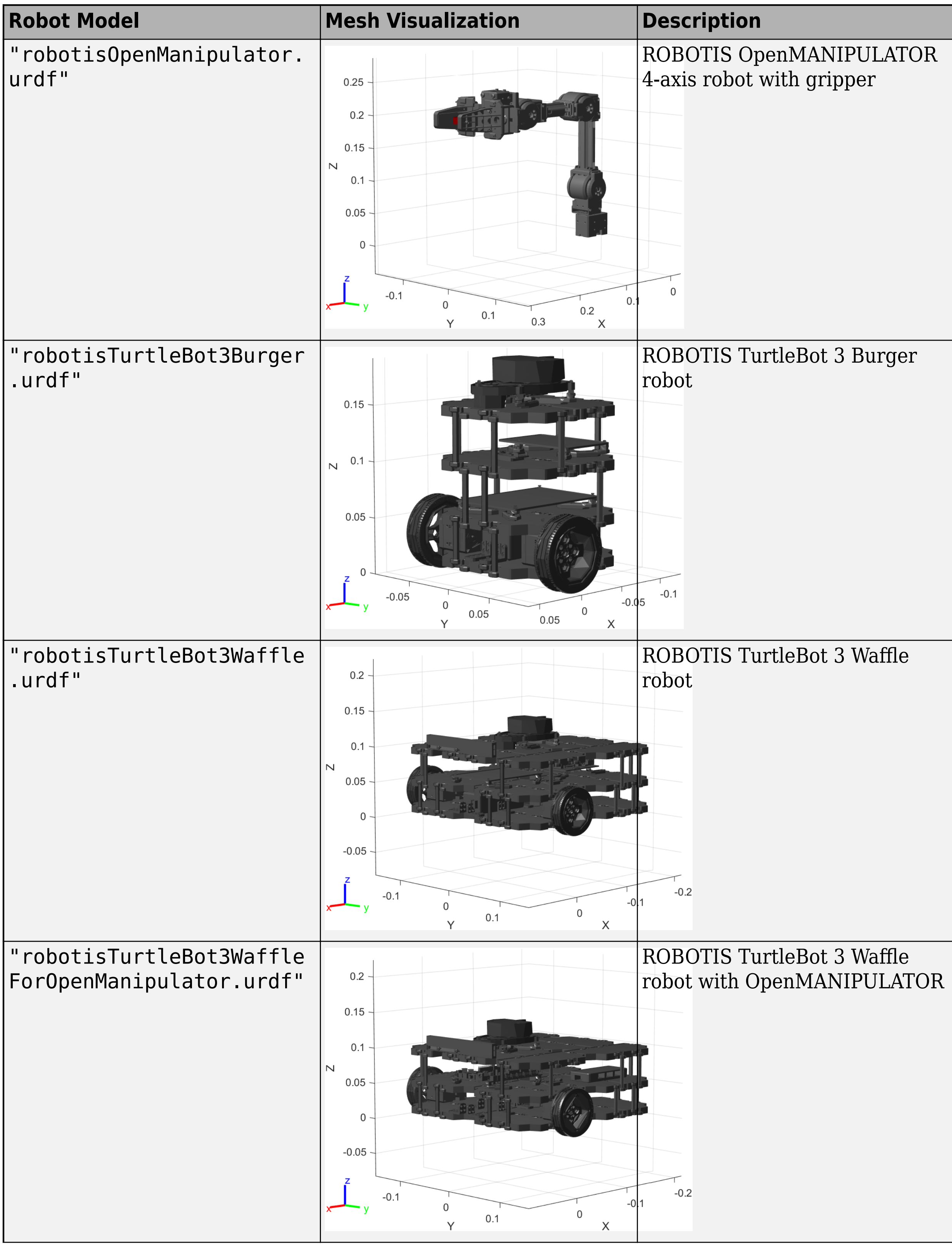
Robot Model	Mesh Visualization	Description
"meca500r3.urdf"		Mecademic Meca500 R3 6-axis robot
"omronEcobra600.urdf"		Omron eCobra 600 4-axis SCARA robot
"puma560.urdf"		PUMA 560 6-axis robot



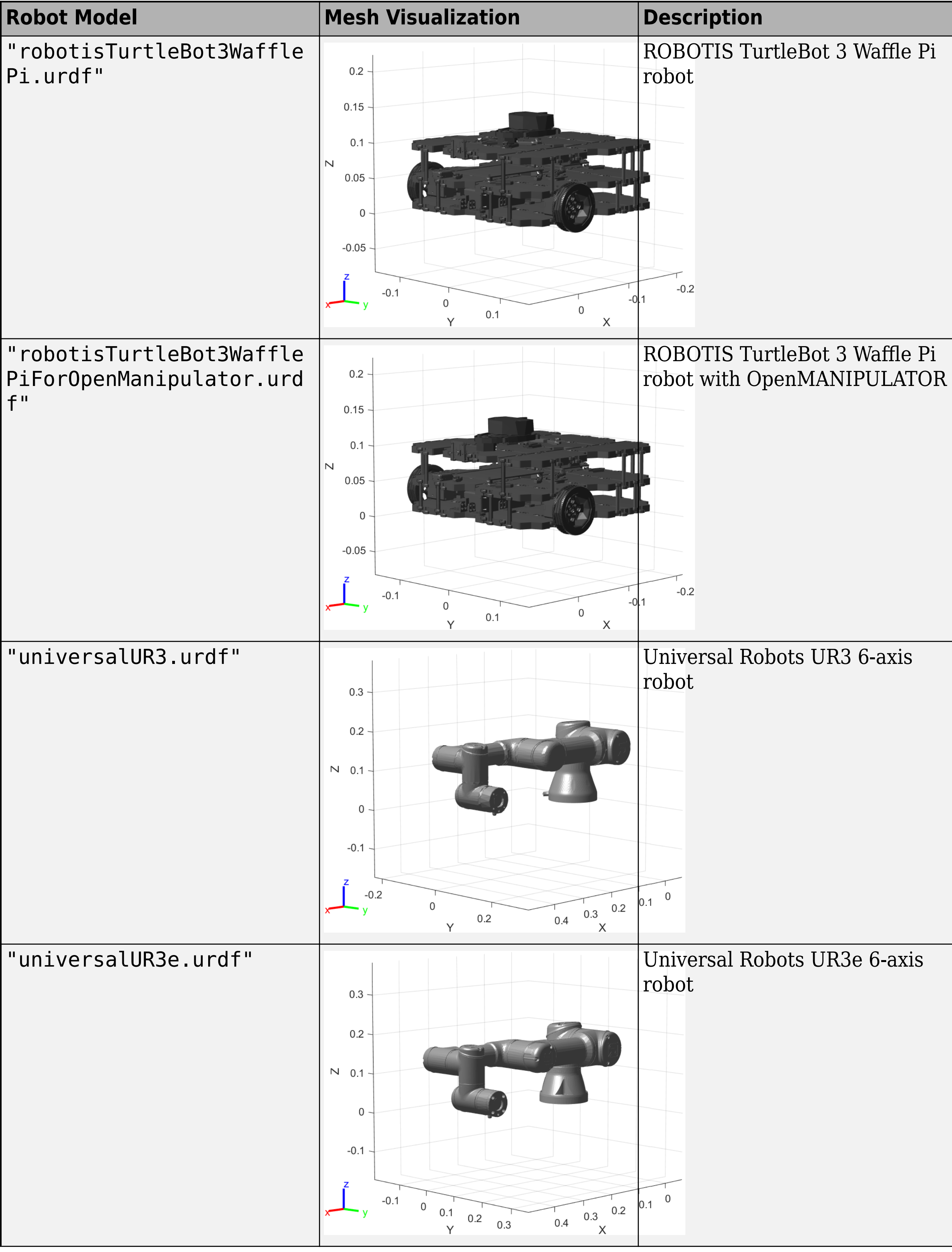
Robot Model	Mesh Visualization	Description
"quanserQArm.urdf"		Quanser QArm 4 DOF robot
"quanserQBot2e.urdf"		Quanser QBot 2e mobile robot
"quanserQCar.urdf"		Quanser QCar mobile robot
"rethinkBaxter.urdf"		Rethink Robotics Baxter 2-armed robot



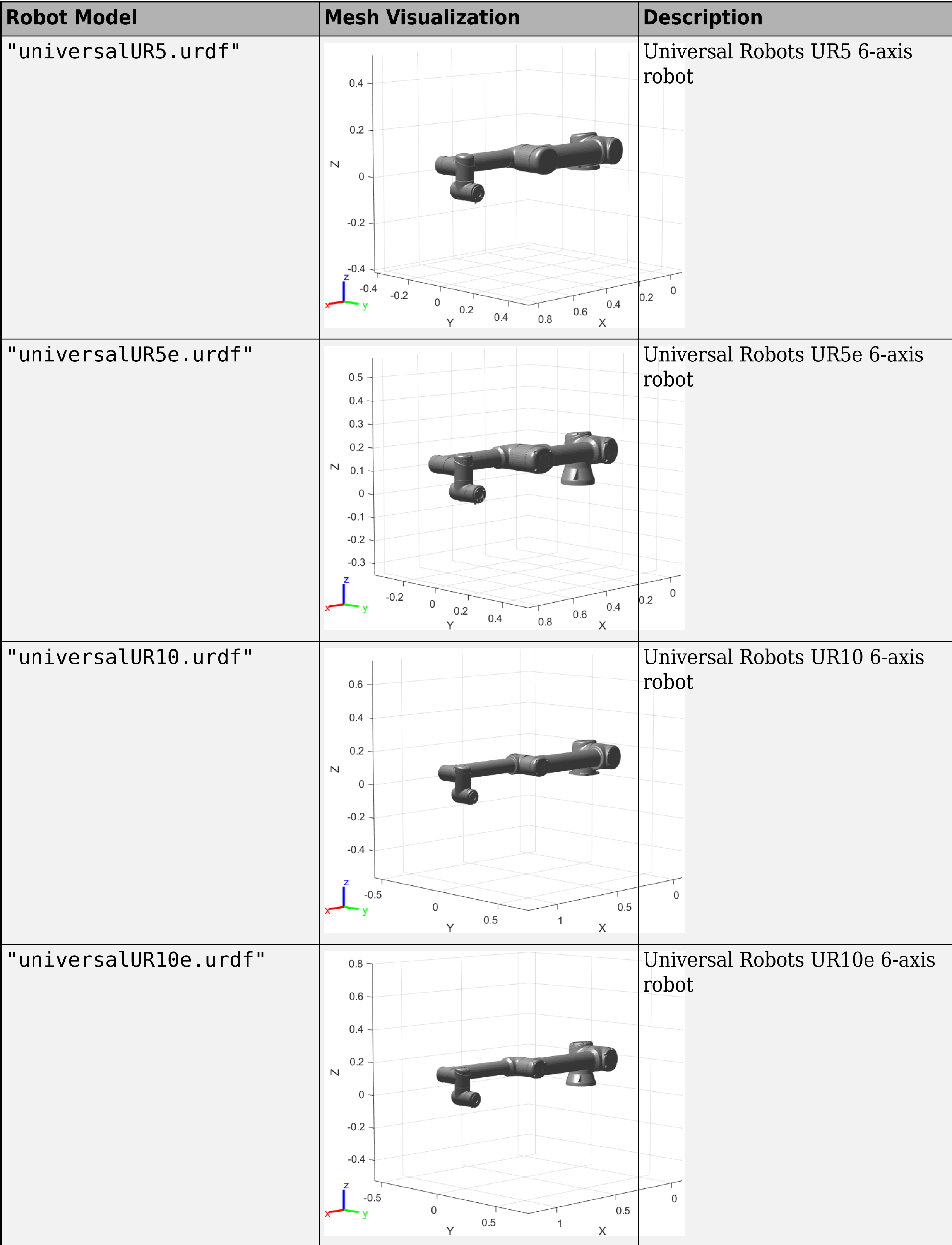
Robot Model	Mesh Visualization	Description
"rethinkSawyer.urdf"		Rethink Robotics Sawyer 7-axis robot
"robotiq2F85.urdf"		Robotiq 2F-85 2-finger gripper The gripper can be used with the following list of manipulators: • Universal Robots UR3 • Universal Robots UR3e • Universal Robots UR5 • Universal Robots UR5e • Universal Robots UR10 • Universal Robots UR10e • Universal Robots UR16e • KINOVA® Gen3 (versions 1 and 2)
"robotisOP2.urdf"		ROBOTIS OP2 Humanoid robot



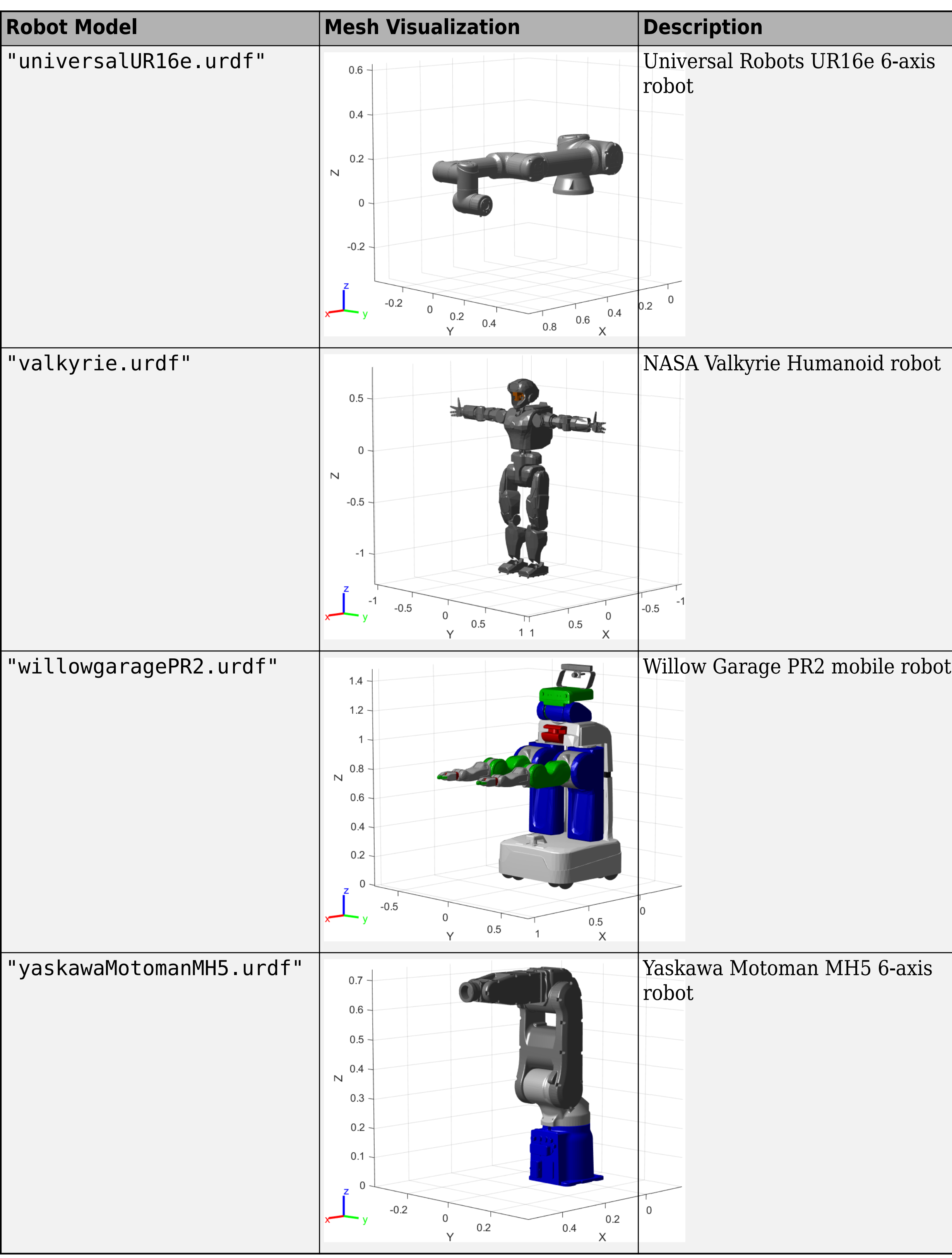
Robot Model	Mesh Visualization	Description
`"robotisOpenManipulator.urdf"`		ROBOTIS OpenMANIPULATOR 4-axis robot with gripper
`"robotisTurtleBot3Burger.urdf"`		ROBOTIS TurtleBot 3 Burger robot
`"robotisTurtleBot3Waffle.urdf"`		ROBOTIS TurtleBot 3 Waffle robot
`"robotisTurtleBot3WaffleForOpenManipulator.urdf"`		ROBOTIS TurtleBot 3 Waffle robot with OpenMANIPULATOR



Robot Model	Mesh Visualization	Description
"robotisTurtleBot3WafflePi.urdf"		ROBOTIS TurtleBot 3 Waffle Pi robot
"robotisTurtleBot3WafflePiForOpenManipulator.urdf"		ROBOTIS TurtleBot 3 Waffle Pi robot with OpenMANIPULATOR
"universalUR3.urdf"		Universal Robots UR3 6-axis robot
"universalUR3e.urdf"		Universal Robots UR3e 6-axis robot



Robot Model	Mesh Visualization	Description
"universalUR5.urdf"		Universal Robots UR5 6-axis robot
"universalUR5e.urdf"		Universal Robots UR5e 6-axis robot
"universalUR10.urdf"		Universal Robots UR10 6-axis robot
"universalUR10e.urdf"		Universal Robots UR10e 6-axis robot



Robot Model	Mesh Visualization	Description
`"universalUR16e.urdf"`		Universal Robots UR16e 6-axis robot
`"valkyrie.urdf"`		NASA Valkyrie Humanoid robot
`"willowgaragePR2.urdf"`		Willow Garage PR2 mobile robot
`"yaskawaMotomanMH5.urdf"`		Yaskawa Motoman MH5 6-axis robot

Example: `"robot_file.urdf"`

Example: "robot_file.xacro"

Example: "robot_file.sdf"

Data Types: char | string

text — Robot description text
string scalar | character vector

Robot description text, specified as a string scalar or character vector. The text must be a valid URDF robot description, Xacro robot description, or SDF model description.

Parse URDF Robot Description from text

Import robot model from URDF text.

```
% Specify URDF text as a character vector.
text = ['<?xml version="1.0" ?>', ...
        '<robot name="min">', ...
        '<link name="L0"/>', ...
        '</robot>'];
% Import the robot model from the URDF text.
robot = importrobot(text);
```

Import robot model from a URDF text file.

```
% Specify URDF text as a character vector.
text = ['<?xml version="1.0" ?>', ...
        '<robot name="min">', ...
        '<link name="L0"/>', ...
        '</robot>'];
% Write the text to file.
writelines(text,"URDF_robot.txt")
% Import the robot model from the URDF text file. Specify the format of
% the robot description text file.
robot = importrobot("URDF_robot.txt","urdf");
```

Parse Xacro Robot Description from text

Import robot model from Xacro text.

```
% Specify Xacro text as a character vector.
text = ['<?xml version="1.0" ?>', ...
        '<robot name="min" ', ...
        'xmlns:xacro="http://www.ros.org/wiki/xacro">', ...
        '<link name="L0"/>', ...
        '</robot>'];
% Import the robot model from the Xacro text.
robot = importrobot(text);
```

Import robot model from a Xacro text file.

```
% Specify Xacro text as a character vector.
text = ['<?xml version="1.0" ?>', ...
        '<robot name="min" ', ...
        'xmlns:xacro="http://www.ros.org/wiki/xacro">', ...
        '<link name="L0"/>', ...
        '</robot>'];
% Write the text to file.
```

```
writelines(text,"Xacro_robot.txt")
% Import the robot model from the Xacro text file. Specify the format of
% the robot description text file.
robot = importrobot("Xacro_robot.txt","xacro");
```

Parse SDF Model Description from text

Import robot model from SDF text.

```
% Specify SDF text as a character vector.
text = ['<?xml version="1.0" ?>', ...
        '<sdf version="1.6">', ...
        '<model name="min">', ...
        '<link name="L0"/>', ...
        '</model>', ...
        '</sdf>'];
% Import the robot model from the SDF text.
robot = importrobot(text);
```

Import robot model from a SDF text file.

```
% Specify SDF text as a character vector.
text = ['<?xml version="1.0" ?>', ...
        '<sdf version="1.6">', ...
        '<model name="min">', ...
        '<link name="L0"/>', ...
        '</model>', ...
        '</sdf>'];
% Write the text to file.
writelines(text,"SDF_robot.txt")
% Import the robot model from the SDF text file. Specify the format of
% the robot description text file.
robot = importrobot("SDF_robot.txt","sdf");
```

Data Types: `char` | `string`

`format` — File format of robot description text file
`"urdf"` | `"xacro"` | `"sdf"`

File format of robot description text file, specified as a string scalar or character vector. Use this argument to specify explicitly the required format for the robot description file.

Example: `"robot_file.txt","urdf"`

Example: `"robot_file.txt","xacro"`

Example: `"robot_file.txt","sdf"`

Data Types: `char` | `string`

`model` — Simscape Multibody model
model handle | string scalar | character vector

Simscape Multibody model, specified as a model handle, string scalar, or character vector.

Data Types: `char` | `string`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `"MeshPath",{"../arm_meshes","../body_meshes"}`

URDF, Xacro, or SDF Import

`MeshPath` — Relative search paths for mesh files

string scalar | character vector | cell array of string scalars or character vectors

Relative search paths for mesh files, specified as a string scalar, character vector, or cell array of string scalars or character vectors. Mesh files must still be specified inside the URDF, Xacro, or SDF file, but `MeshPath` defines the relative paths for these specified files.

Data Types: `char` | `string` | `cell`

`DataFormat` — Input/output data format for kinematics and dynamics functions

`"struct"` (default) | `"row"` | `"column"`

Input/output data format for the kinematics and dynamics functions of the robot model, specified as the comma-separated pair consisting of `'DataFormat'` and `"struct"`, `"row"`, or `"column"`. To use dynamics functions, you must specify either `"row"` or `"column"`. This name-value pair sets the DataFormat property of the `rigidBodyTree` robot model.

Data Types: `char` | `string`

`SDFModel` — Select model from SDF that contain multiple models

string scalar | character vector

Select a model from the SDF file or text that contain multiple models, specified as a string scalar or character vector.

Note This name-value pair only applies to the SDF model and text.

Data Types: `char` | `string`

`MaxNumBodies` — Maximum number of bodies allowed in imported robot during code generation

integer

Maximum number of bodies allowed in imported robot during code generation, specified as an integer. Use `MaxNumBodies` to add rigid bodies to the imported tree inside a function that supports code generation. The number of additional bodies that can be added is the difference between `MaxNumBodies` and the number of bodies in the imported tree, `rigidBodyTree.NumBodies`.

Note This name-value pair is only necessary for code generation workflows.

Simscape Multibody Model Import

`BreakChains` — Indicates whether to break closed chains
`"error"` (default) | `"remove-joints"`

Indicates whether to break closed chains in the given `model` input, specified as `"error"` or `"remove-joints"`. If you specify `"remove-joints"`, the resulting `robot` output has chain closure joints removed. Otherwise, the function throws an error.

Data Types: `char` | `string`

`ConvertJoints` — Indicates whether to convert unsupported joints to fixed
`"error"` (default) | `"convert-to-fixed"`

Indicates whether to convert unsupported joints to fixed joints in the given `model` input, specified as `"error"` or `"convert-to-fixed"`. If you specify `"convert-to-fixed"`, the resulting `robot` output has any unsupported joints converted to fixed joints. Only fixed, prismatic, and revolute joints are supported in the output `rigidBodyTree` object. Otherwise, if the `model` contains unsupported joints, the function throws an error.

Data Types: `char` | `string`

`SMConstraints` — Indicates whether to remove constraint blocks
`"error"` (default) | `"remove"`

Indicates whether to remove constraint blocks in the given `model` input, specified as `"error"` or `"remove"`. If you specify `"remove"`, the resulting `robot` output has the constraints removed. Otherwise, if the `model` contains constraint blocks, the function throws an error.

Data Types: `char` | `string`

`VariableInertias` — Indicates whether to remove variable inertia blocks
`"error"` (default) | `"remove"`

Indicates whether to remove variable inertia blocks in the given `model` input, specified as `"error"` or `"remove"`. If you specify `"remove"`, the resulting `robot` output has the variable inertias removed. Otherwise, if the `model` contains variable inertia blocks, the function throws an error.

Data Types: `char` | `string`

`DataFormat` — Input/output data format for kinematics and dynamics functions
`"struct"` (default) | `"row"` | `"column"`

Input/output data format for the kinematics and dynamics functions of the robot model, specified as the comma-separated pair consisting of `'DataFormat'` and `"struct"`, `"row"`, or `"column"`. To use dynamics functions, you must specify either `"row"` or `"column"`. This name-value pair sets the DataFormat property of the `rigidBodyTree` robot model.

Data Types: `char` | `string`

Output Arguments

`robot` — Robot model
`rigidBodyTree` object

Robot model, returned as a `rigidBodyTree` object.

Note If the gravity is not specified in the URDF file, the default `Gravity` property is set to `[0 0 0]`. Simscape Multibody uses a default of `[0 0 -9.80665]`m/s^2 when using `smimport` to import a URDF.

`importInfo` — Object for storing import information
`rigidBodyTreeImportInfo` object

Object for storing import information, returned as a `rigidBodyTreeImportInfo` object. This object contains the relationship between the input `model` and the resulting `robot` output.

Use `showdetails` to list all the import info for each body in the `robot`. Links to display the rigid body info, their corresponding blocks in the model, and highlighting specific blocks in the model are output to the command window.

Use `bodyInfo`, `bodyInfoFromBlock`, or `bodyInfoFromJoint` to get information about specific components in either the `robot` output or the `model` input.

Tips

When importing a robot model with visual meshes, the `importrobot` function searches for the `.stl` or `.dae` files to assign to each rigid body using these rules:

- The function searches the raw mesh path for a specified rigid body from the URDF, Xacro, or SDF file. References to ROS packages have the `package:\\<pkg_name>` removed.
- Absolute paths are checked directly with no modification.
- Relative paths are checked using the following directories in order:
  - User-specified `MeshPath`
  - Current folder
  - MATLAB path
  - The folder containing the URDF, Xacro, or SDF file
  - One level above the folder containing the URDF, Xacro, or SDF file
- The file name from the mesh path in the URDF, Xacro, or SDF file is appended to the `MeshPath` input argument.

If the mesh file is still not found, the parser ignores the mesh file and returns a `rigidBodyTree` object without visual.

Version History

Introduced in R2017a

See Also

`rigidBodyTree` | `rigidBodyTreeImportInfo` | `loadrobot`

Topics

"Rigid Body Tree Robot Model"

loadrobot

Load rigid body tree robot model

Syntax

```
robotRBT = loadrobot(robotname)
[robotRBT,robotData] = loadrobot(robotname)
[robotRBT,robotData] = loadrobot(robotname,Name,Value)
```

Description

`robotRBT = loadrobot(robotname)` loads a robot model as a `rigidBodyTree` object specified by robot model name `robotname`.

To import your own robot model from a Unified Robot Description Format (URDF), XML Macros (Xacro), or Simulation Description Format (SDF) file or Simscape Multibody model, see the `importrobot` function.

`[robotRBT,robotData] = loadrobot(robotname)` returns additional information about the robot model as a structure, `robotData`.

`[robotRBT,robotData] = loadrobot(robotname,Name,Value)` specifies additional options using one or more name-value pair arguments. For example, `'Gravity',[0 0 –9.81]` sets the gravity property to –9.81 m/s^2 in the *z*-direction for the robot model.

Examples

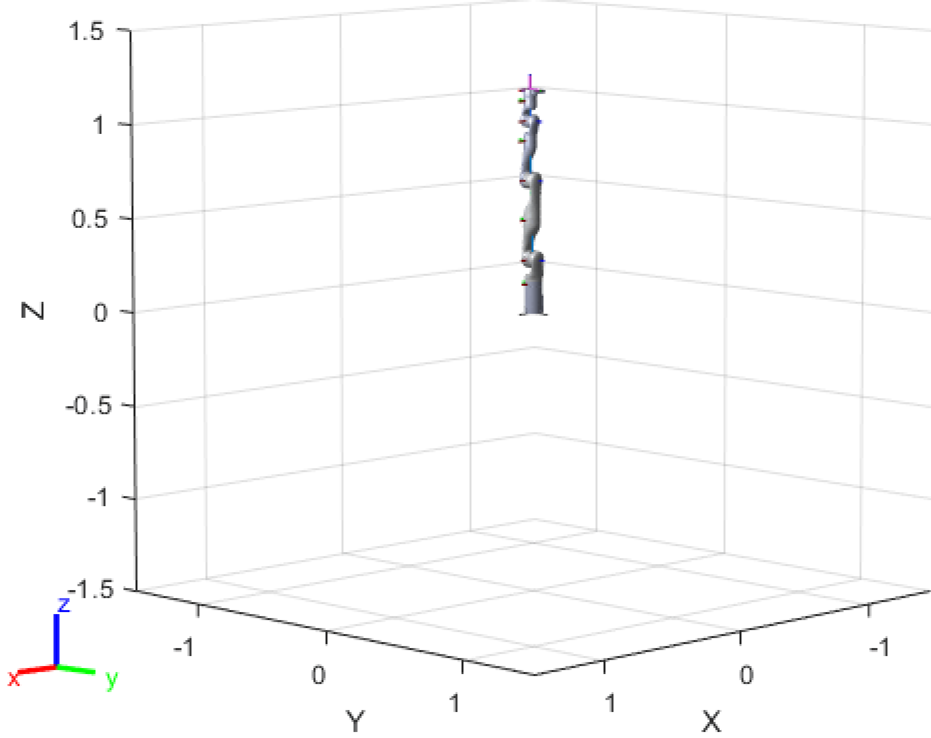
Load Provided Robot Model

This example shows how to load an included robot model using `loadrobot`. Specify one of the select robot names to get a `rigidBodyTree` robot model that contains kinematic and dynamic constraints and visual meshes for the specified robot geometry.

```
gen3 = loadrobot("kinovaGen3");
```

Show the robot model in a figure.

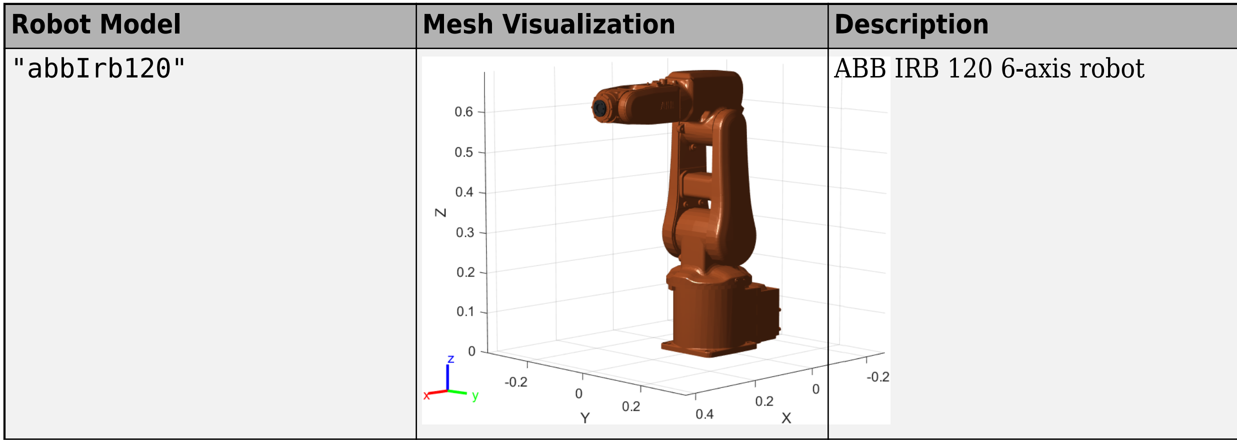
```
show(gen3);
```



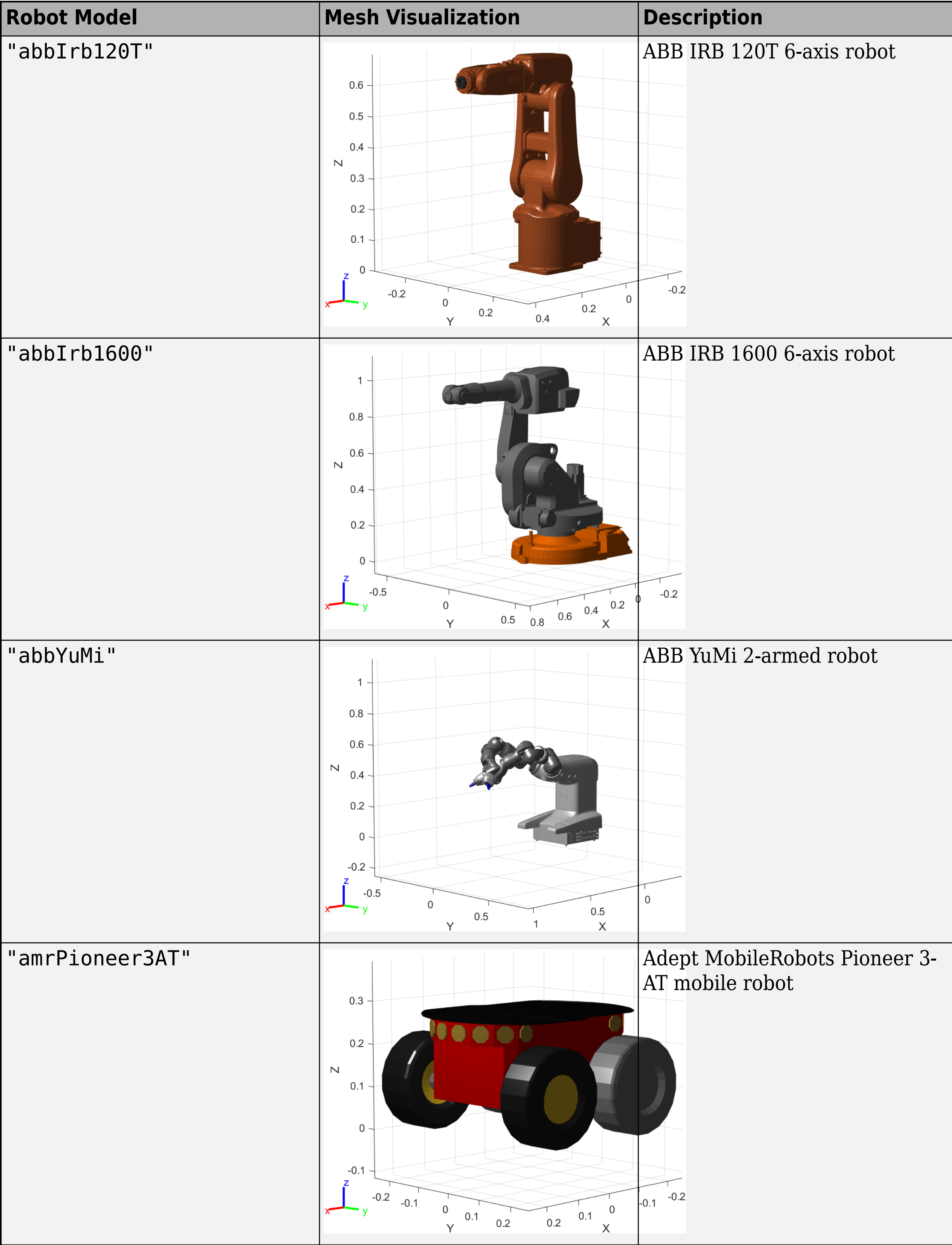
Input Arguments

`robotname` — Name of robot model
`"abbIrb120"` | `"abbIrb120T"` | `"abbIrb1600"` | ...

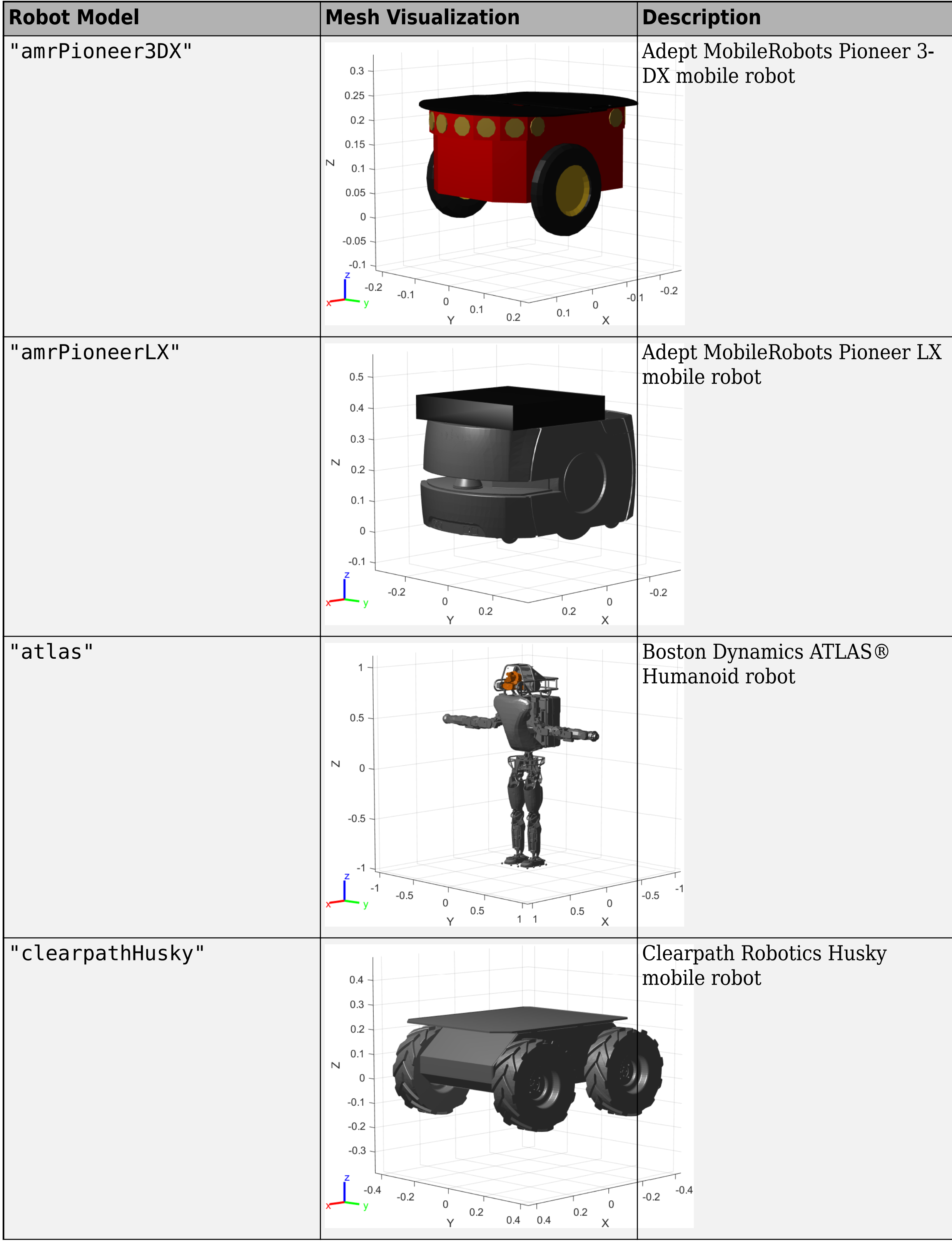
Name of robot model, specified as one of these valid robot model names:



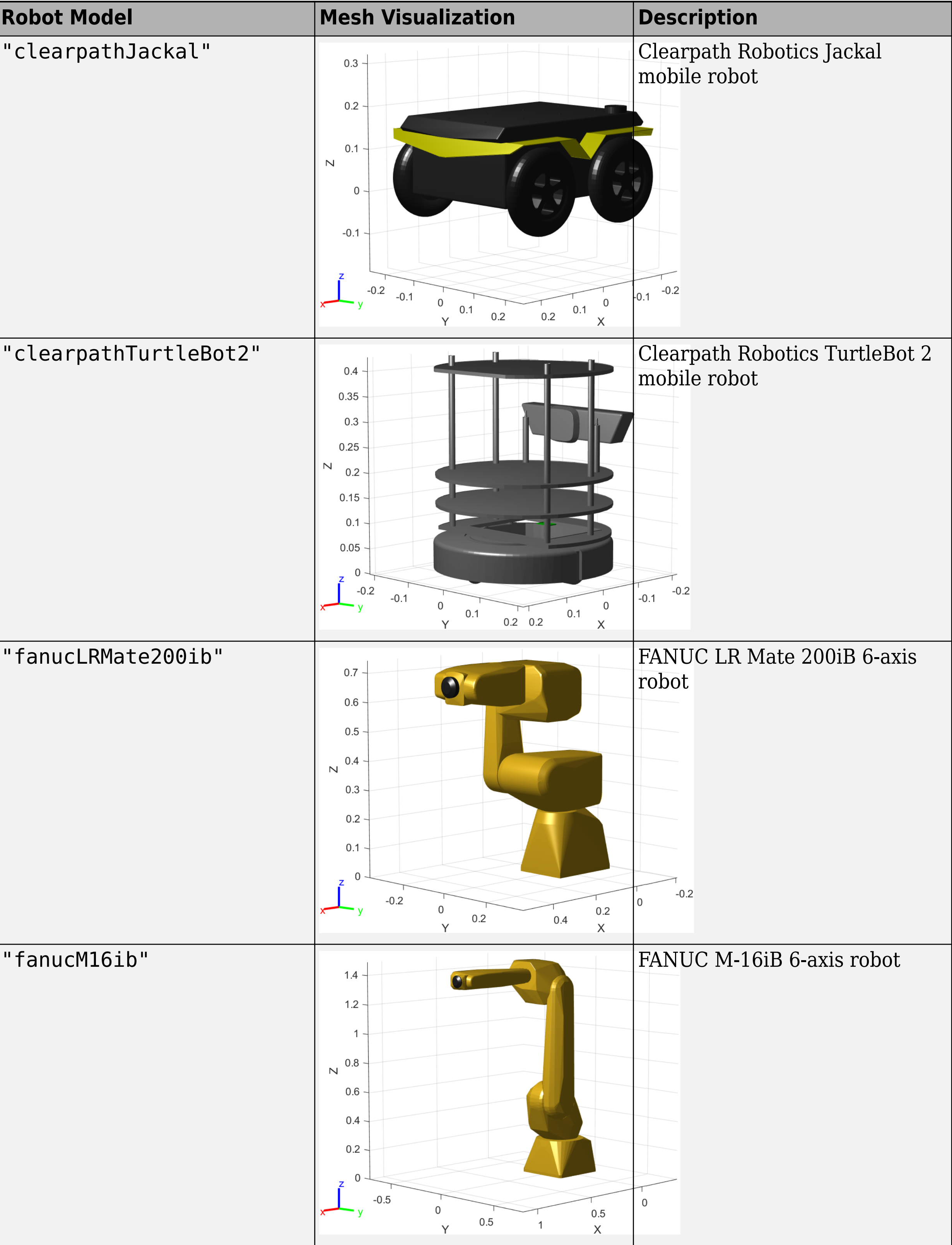
Robot Model	Mesh Visualization	Description
`"abbIrb120"`		ABB IRB 120 6-axis robot



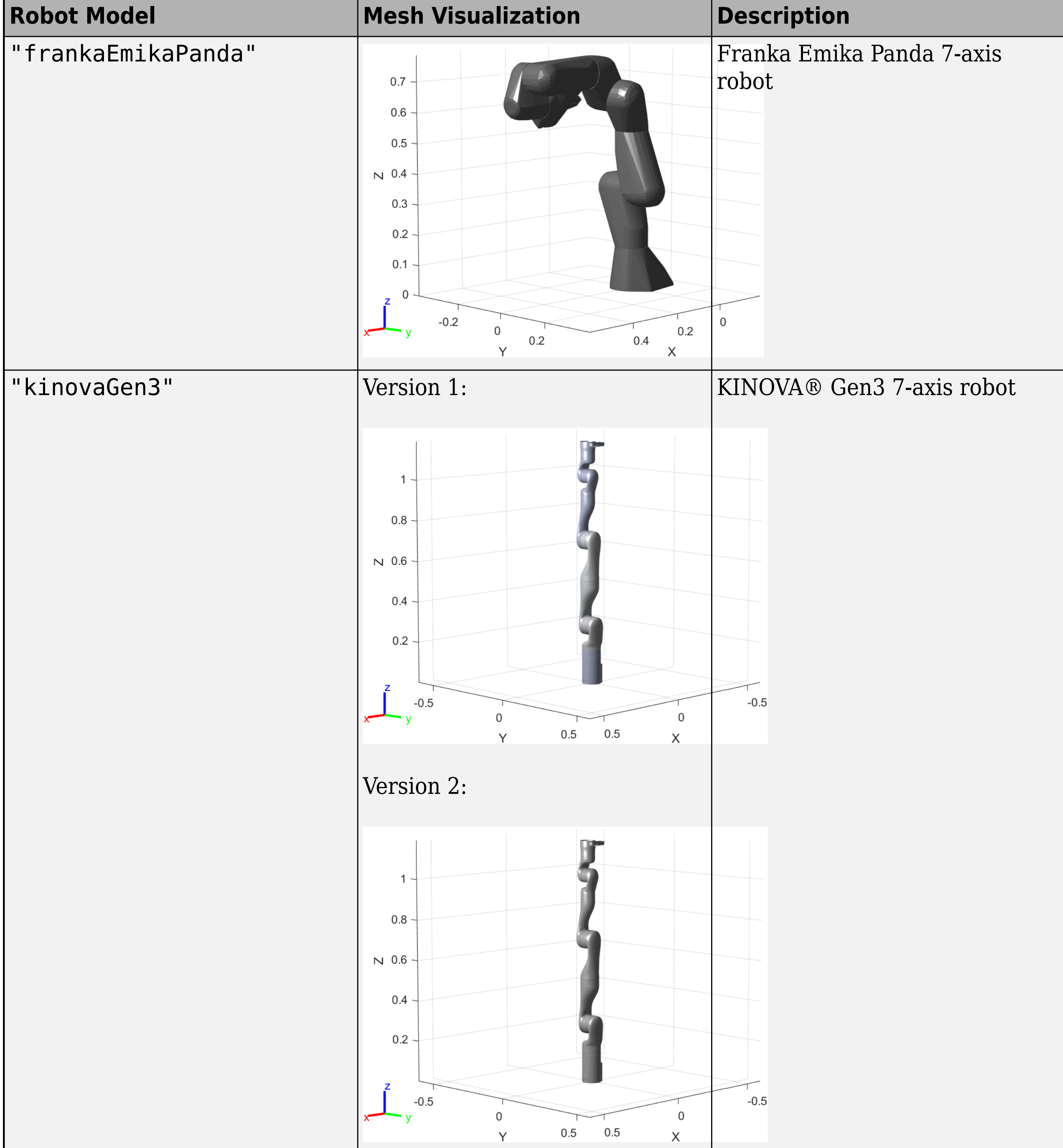
Robot Model	Mesh Visualization	Description
"abbIrb120T"		ABB IRB 120T 6-axis robot
"abbIrb1600"		ABB IRB 1600 6-axis robot
"abbYuMi"		ABB YuMi 2-armed robot
"amrPioneer3AT"		Adept MobileRobots Pioneer 3-AT mobile robot



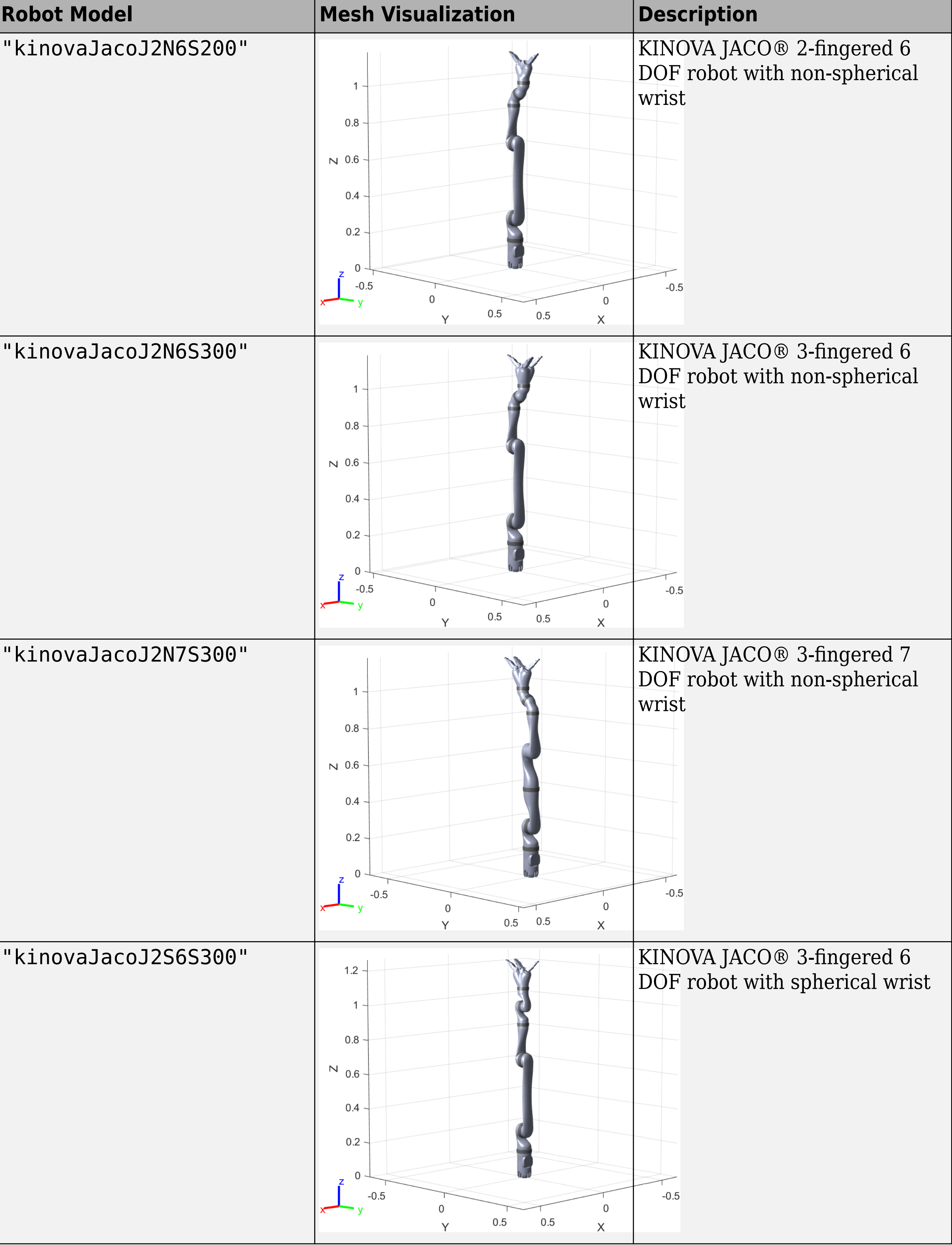
Robot Model	Mesh Visualization	Description
"amrPioneer3DX"		Adept MobileRobots Pioneer 3-DX mobile robot
"amrPioneerLX"		Adept MobileRobots Pioneer LX mobile robot
"atlas"		Boston Dynamics ATLAS® Humanoid robot
"clearpathHusky"		Clearpath Robotics Husky mobile robot

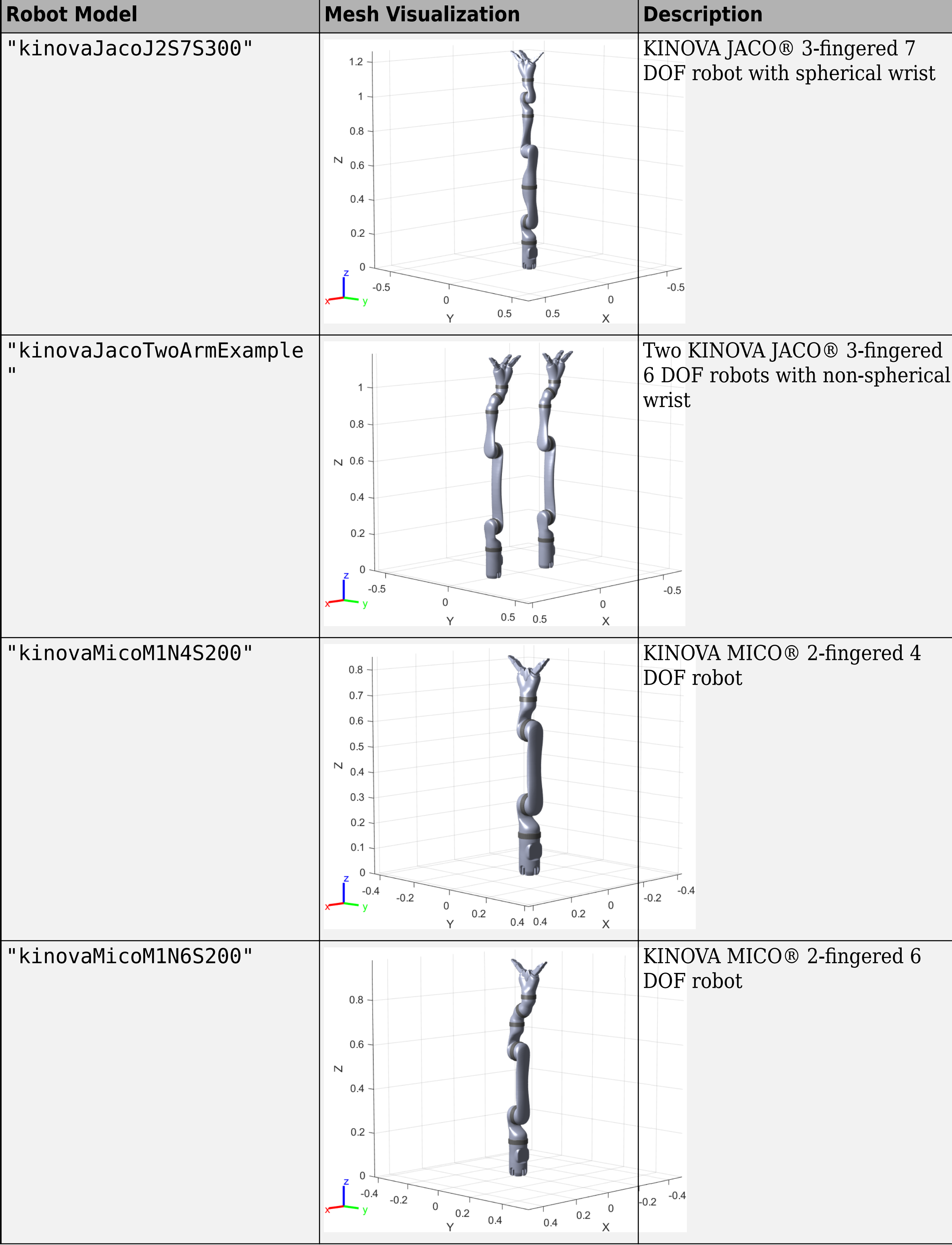


Robot Model	Mesh Visualization	Description
"clearpathJackal"		Clearpath Robotics Jackal mobile robot
"clearpathTurtleBot2"		Clearpath Robotics TurtleBot 2 mobile robot
"fanucLRMate200ib"		FANUC LR Mate 200iB 6-axis robot
"fanucM16ib"		FANUC M-16iB 6-axis robot

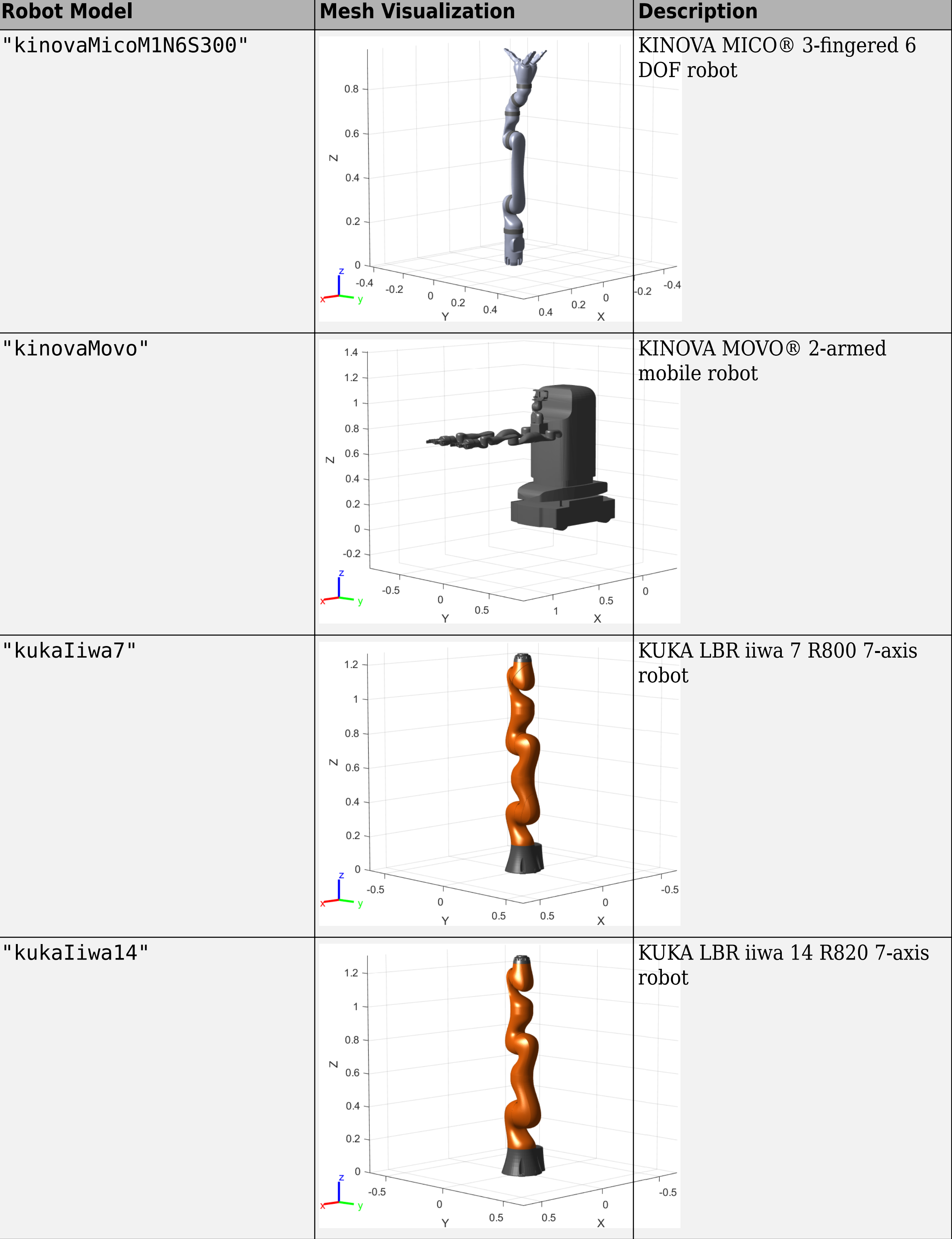


Robot Model	Mesh Visualization	Description
`"frankaEmikaPanda"`		Franka Emika Panda 7-axis robot
`"kinovaGen3"`	Version 1: Version 2:	KINOVA® Gen3 7-axis robot

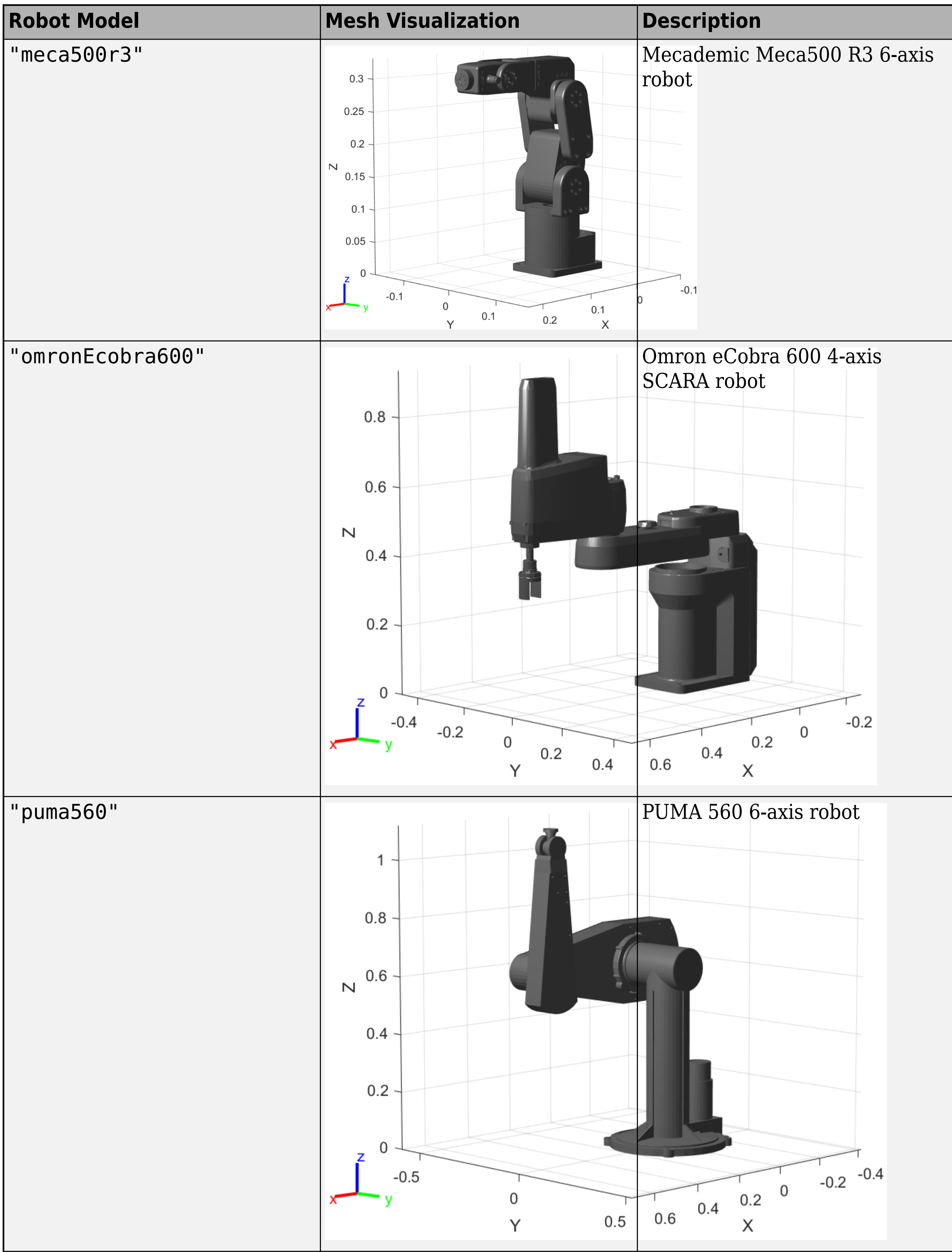
Robot Model	Mesh Visualization	Description
`"kinovaJacoJ2N6S200"`		KINOVA JACO® 2-fingered 6 DOF robot with non-spherical wrist
`"kinovaJacoJ2N6S300"`		KINOVA JACO® 3-fingered 6 DOF robot with non-spherical wrist
`"kinovaJacoJ2N7S300"`		KINOVA JACO® 3-fingered 7 DOF robot with non-spherical wrist
`"kinovaJacoJ2S6S300"`		KINOVA JACO® 3-fingered 6 DOF robot with spherical wrist

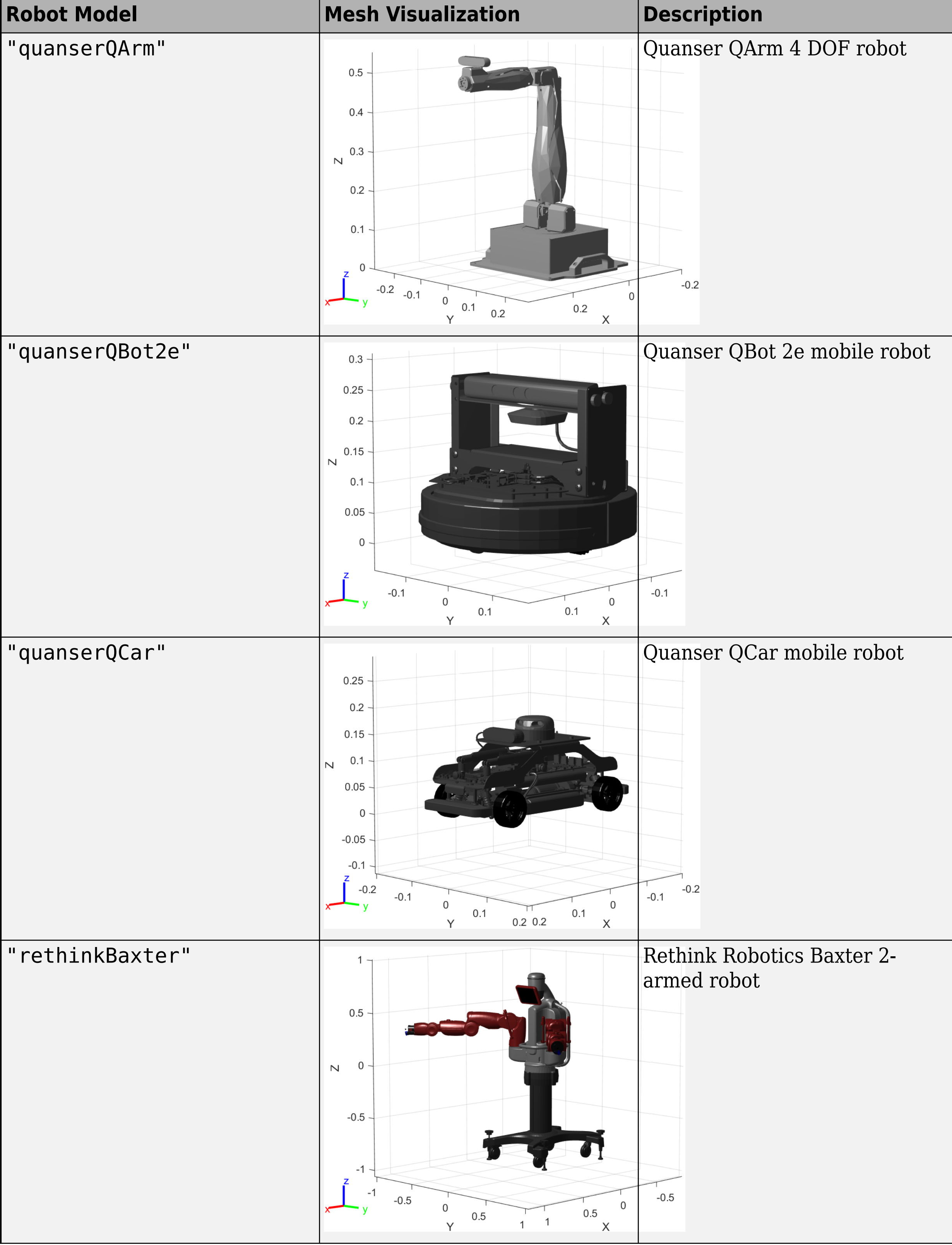


Robot Model	Mesh Visualization	Description
`"kinovaJacoJ2S7S300"`		KINOVA JACO® 3-fingered 7 DOF robot with spherical wrist
`"kinovaJacoTwoArmExample"`		Two KINOVA JACO® 3-fingered 6 DOF robots with non-spherical wrist
`"kinovaMicoM1N4S200"`		KINOVA MICO® 2-fingered 4 DOF robot
`"kinovaMicoM1N6S200"`		KINOVA MICO® 2-fingered 6 DOF robot

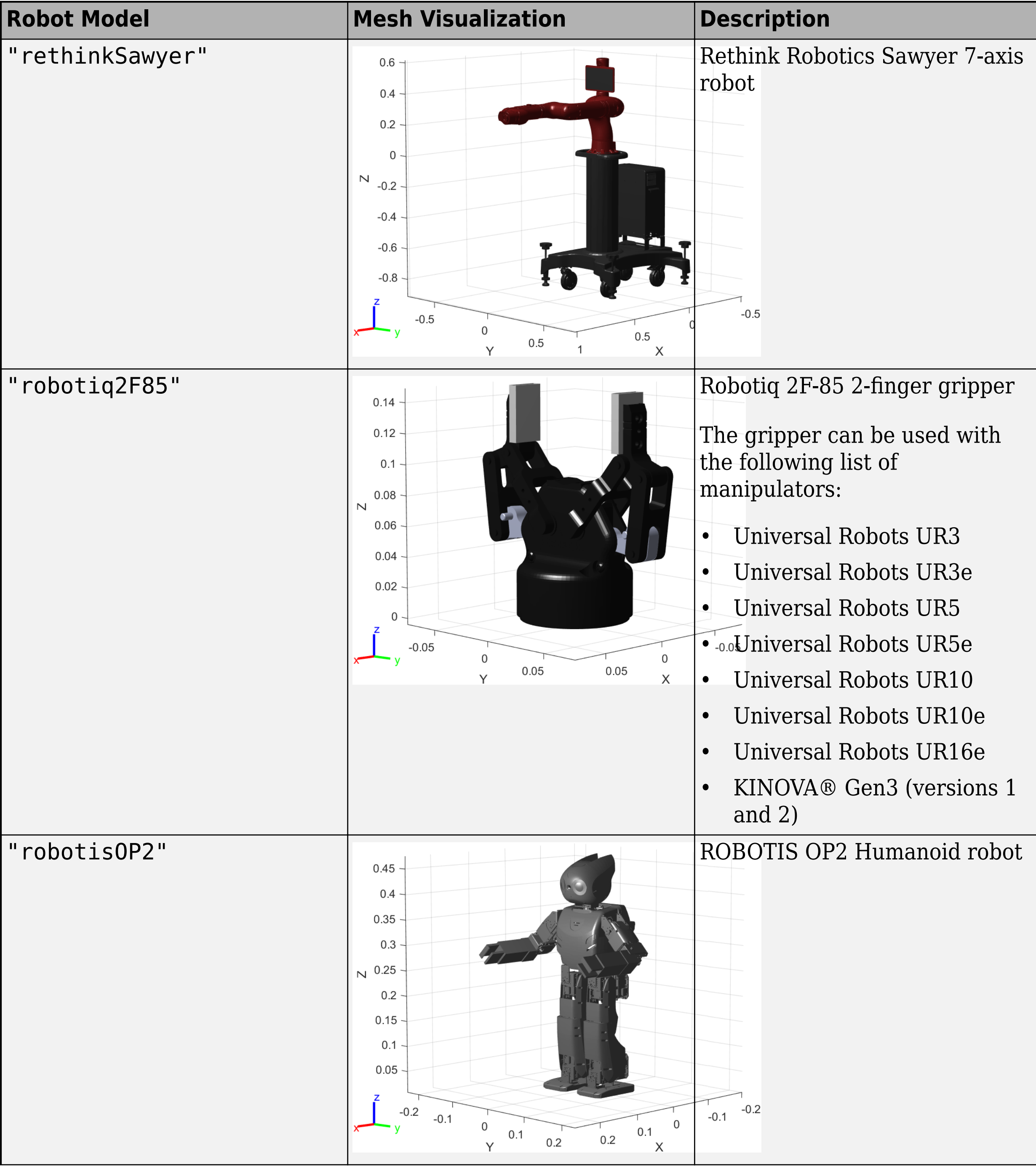


Robot Model	Mesh Visualization	Description
`"kinovaMicoM1N6S300"`		KINOVA MICO® 3-fingered 6 DOF robot
`"kinovaMovo"`		KINOVA MOVO® 2-armed mobile robot
`"kukaIiwa7"`		KUKA LBR iiwa 7 R800 7-axis robot
`"kukaIiwa14"`		KUKA LBR iiwa 14 R820 7-axis robot

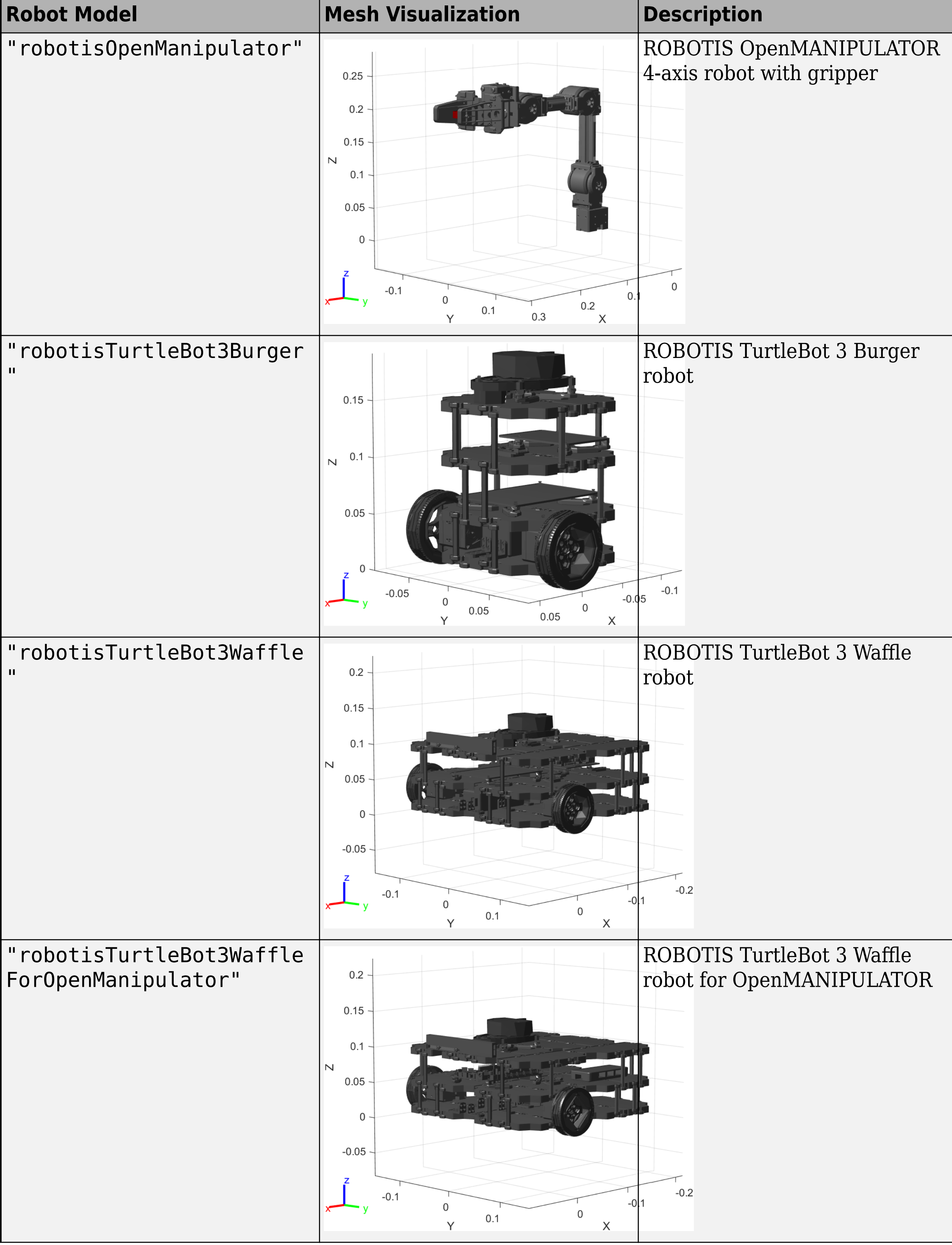
Robot Model	Mesh Visualization	Description
"meca500r3"		Mecademic Meca500 R3 6-axis robot
"omronEcobra600"		Omron eCobra 600 4-axis SCARA robot
"puma560"		PUMA 560 6-axis robot



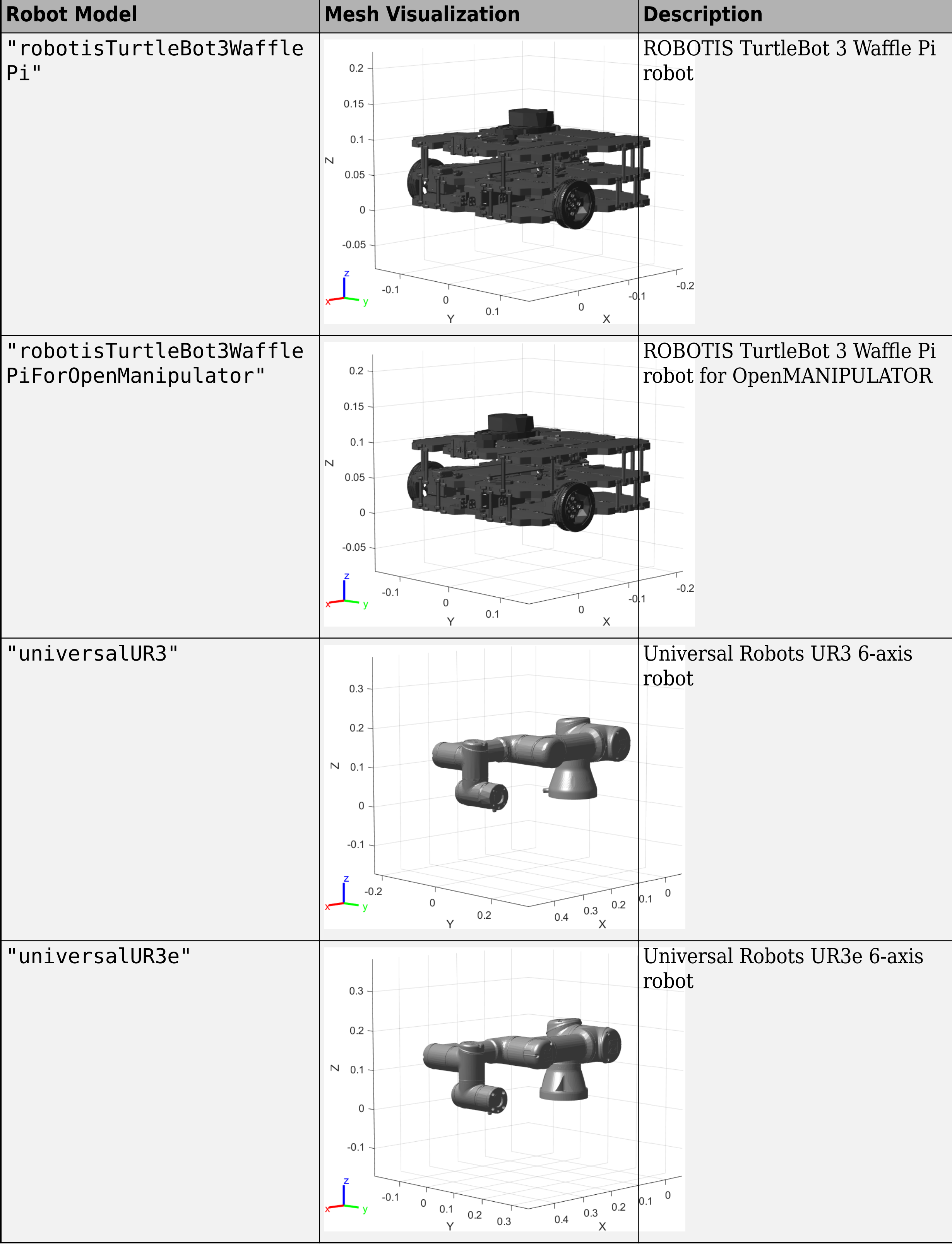
Robot Model	Mesh Visualization	Description
"quanserQArm"		Quanser QArm 4 DOF robot
"quanserQBot2e"		Quanser QBot 2e mobile robot
"quanserQCar"		Quanser QCar mobile robot
"rethinkBaxter"		Rethink Robotics Baxter 2-armed robot



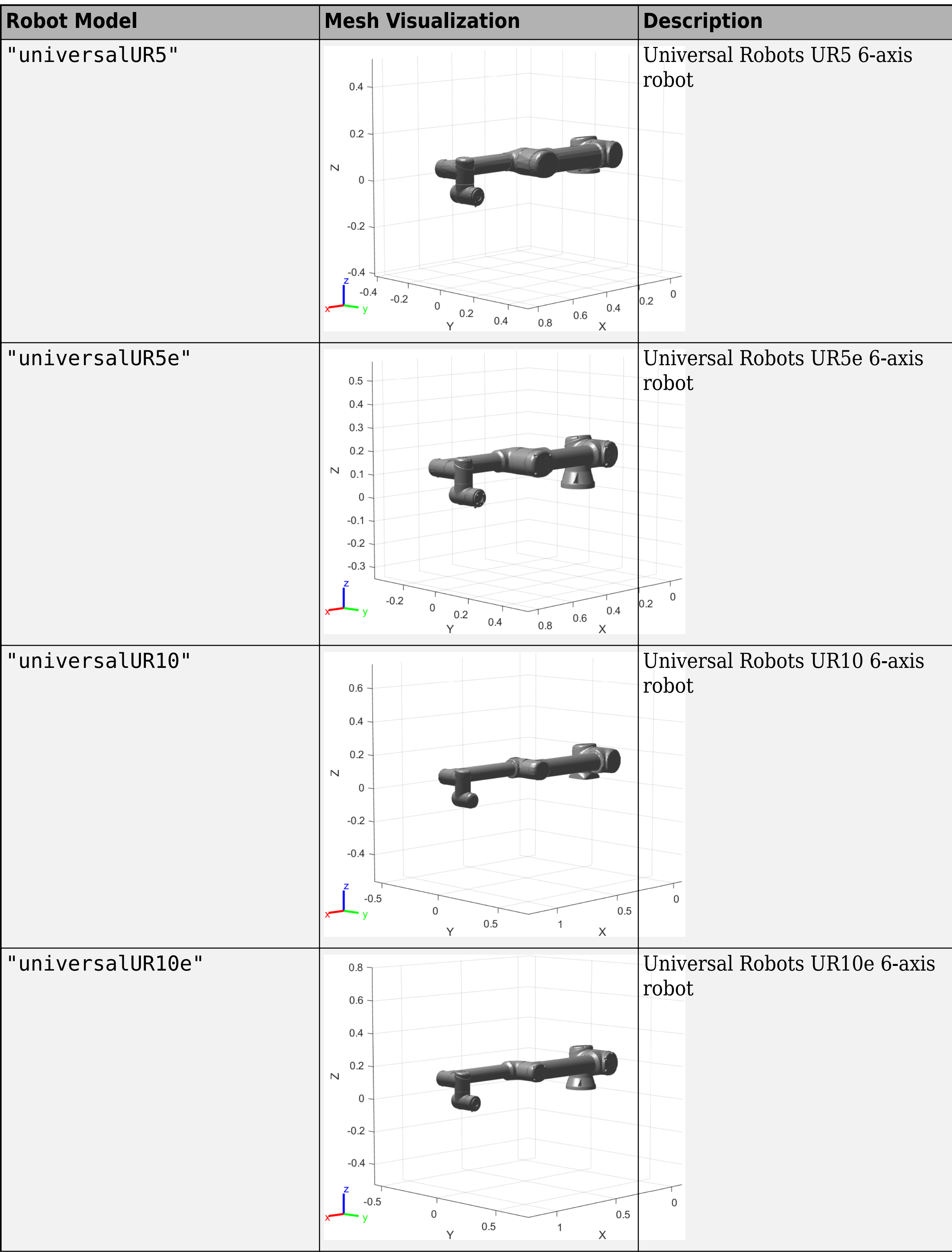
Robot Model	Mesh Visualization	Description
`"rethinkSawyer"`		Rethink Robotics Sawyer 7-axis robot
`"robotiq2F85"`		Robotiq 2F-85 2-finger gripper The gripper can be used with the following list of manipulators: • Universal Robots UR3 • Universal Robots UR3e • Universal Robots UR5 • Universal Robots UR5e • Universal Robots UR10 • Universal Robots UR10e • Universal Robots UR16e • KINOVA® Gen3 (versions 1 and 2)
`"robotisOP2"`		ROBOTIS OP2 Humanoid robot

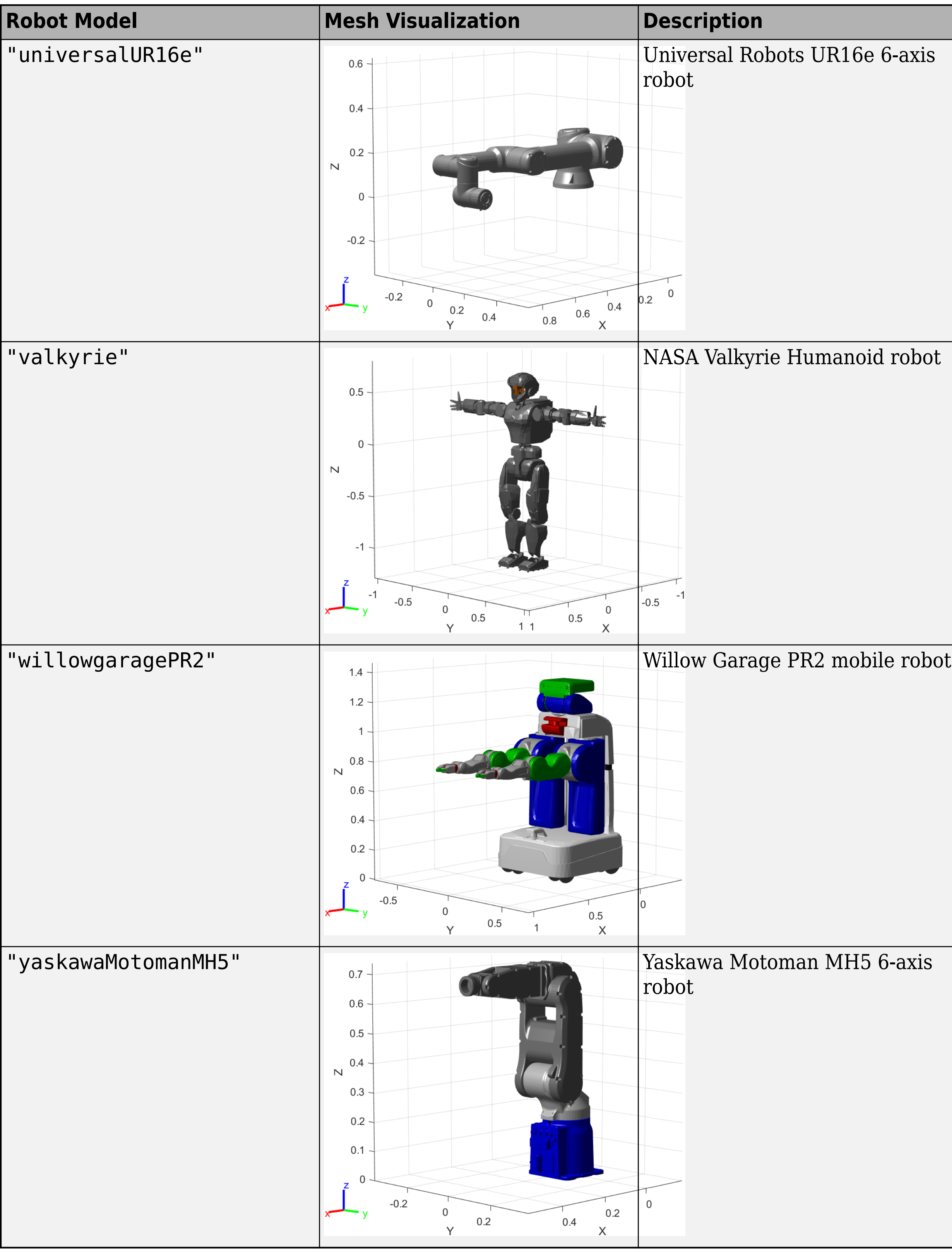


Robot Model	Mesh Visualization	Description
"robotisOpenManipulator"		ROBOTIS OpenMANIPULATOR 4-axis robot with gripper
"robotisTurtleBot3Burger"		ROBOTIS TurtleBot 3 Burger robot
"robotisTurtleBot3Waffle"		ROBOTIS TurtleBot 3 Waffle robot
"robotisTurtleBot3WaffleForOpenManipulator"		ROBOTIS TurtleBot 3 Waffle robot for OpenMANIPULATOR



Robot Model	Mesh Visualization	Description
"robotisTurtleBot3WafflePi"		ROBOTIS TurtleBot 3 Waffle Pi robot
"robotisTurtleBot3WafflePiForOpenManipulator"		ROBOTIS TurtleBot 3 Waffle Pi robot for OpenMANIPULATOR
"universalUR3"		Universal Robots UR3 6-axis robot
"universalUR3e"		Universal Robots UR3e 6-axis robot

Robot Model	Mesh Visualization	Description
"universalUR5"		Universal Robots UR5 6-axis robot
"universalUR5e"		Universal Robots UR5e 6-axis robot
"universalUR10"		Universal Robots UR10 6-axis robot
"universalUR10e"		Universal Robots UR10e 6-axis robot



Robot Model	Mesh Visualization	Description
"universalUR16e"		Universal Robots UR16e 6-axis robot
"valkyrie"		NASA Valkyrie Humanoid robot
"willowgaragePR2"		Willow Garage PR2 mobile robot
"yaskawaMotomanMH5"		Yaskawa Motoman MH5 6-axis robot

Data Types: `char` | `string`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `'Gravity',[0 0 -9.81]` sets the gravity property to -9.81 m/s^2 in the *z*-direction for the robot model.

`DataFormat` — Input/output data format for kinematics and dynamics functions
`"struct"` (default) | `"row"` | `"column"`

Input/output data format for the kinematics and dynamics functions of the robot model, specified as the comma-separated pair consisting of `'DataFormat'` and `"struct"`, `"row"`, or `"column"`. To use dynamics functions, you must specify either `"row"` or `"column"`. This name-value pair sets the DataFormat property of the `rigidBodyTree` robot model.

`Gravity` — Gravitational acceleration experienced by robot
`[0 0 0]` m/s^2 (default) | three-element vector of the form `[x y z]`

Gravitational acceleration experienced by robot, specified as the comma-separated pair consisting of `'Gravity'` and a three-element vector of the form `[x y z]` in m/s^2. Each element corresponds to the acceleration of the base robot frame in the *x*-, *y*-, and *z*-direction, respectively. This name-value pair sets the Gravity property of the `rigidBodyTree` robot model.

`Version` — URDF version of robot model
`1` (default) | numeric scalar

URDF version of the robot model, specified as a numeric scalar.

Robot Model	Versions
`"kinovaGen3"`	`1` –– Loads the `kinovaGen3.urdf` robot model
	`2` –– Loads the `kinovaGen3V12.urdf` robot model

Example: `loadrobot("kinovaGen3","Version",2)`

Output Arguments

`robotRBT` — Rigid body tree robot model
`rigidBodyTree` object

Rigid body tree robot model, returned as a `rigidBodyTree` object. This model contains all the kinematic and dynamic constraints based on the robot source files specified in `robotData`. Some models also contain visual meshes for visualizing robot trajectories.

`robotData` — Robot model information
structure

Robot model information, returned as a structure containing these fields. Whether the function returns a value for a field is based on the type of robot specified by the `robotname` input. Non-relevant fields for that robot are empty.

This table describes the fields of the robot model information structure.

Field	Description
RobotName (relevant for all robot types)	Name of the returned robot model
FilePath (relevant for all robot types)	File path of the URDF file that is used to create the rigid body tree model
Source (relevant for all robot types)	URL source of the robot model
Version (relevant for all robot types)	Version number of the robot model
WheelRadius	Wheel radius of the robot in meters
WheelBase	Distance between front and rear axles in meters
TrackWidth	Distance between wheels on axle in meters
MaxTranslationalVelocity	Maximum linear velocity of the robot in m/s
MaxRotationalVelocity	Maximum angular velocity of the robot in rad/s
DriveType	All robots are modeled with a fixed base, but this field describes the actual drive type of the robot base. The drive type can be any one of the following based on the specified robot: • FixedBase –– Drive type of robots with a fixed base • Differential-Drive –– Drive type of robots with a differential-drive mobile base • Omni-Wheel –– Drive type of robots with an omni-wheel mobile base
ManipulatorMotionModel	Motion model of a manipulator robot • jointSpaceMotionModel object –– Joint-space motion model of the manipulator robot
MobileBaseMotionModel	Kinematic motion model of the mobile base. The motion model can be any one of the following based on the specified robot: • differentialDriveKinematics object –– Differential-drive kinematic motion model for robots with a differential-drive mobile base • unicycleKinematics object –– Unicycle kinematic motion model for robots with an omni-wheel mobile base

Field	Description
HasBodyInertias	Flag indicating if the robot model has inertias, indicated as a 1 (true) if the robot model has at least one body with body inertia or 0 (false) if the robot model has no bodies with inertia.

Data Types: struct

Version History

Introduced in R2019b

See Also

rigidBodyTree | importrobot | inverseKinematics

ldivide, .\

Element-wise quaternion left division

Syntax

```
C = A.\B
```

Description

`C = A.\B` performs quaternion element-wise division by dividing each element of quaternion `B` by the corresponding element of quaternion `A`.

Examples

Divide a Quaternion Array by a Real Scalar

Create a 2-by-1 quaternion array, and divide it element-by-element by a real scalar.

```
A = quaternion([1:4;5:8])
```

```
A = 2x1 quaternion array
     1 + 2i + 3j + 4k
     5 + 6i + 7j + 8k
```

```
B = 2;
C = A.\B
```

```
C = 2x1 quaternion array
     0.066667 -  0.13333i -      0.2j -  0.26667k
     0.057471 - 0.068966i -  0.08046j - 0.091954k
```

Divide a Quaternion Array by Another Quaternion Array

Create a 2-by-2 quaternion array, and divide it element-by-element by another 2-by-2 quaternion array.

```
q1 = quaternion([1:4;2:5;4:7;5:8]);
A = reshape(q1,2,2)
```

```
A = 2x2 quaternion array
     1 + 2i + 3j + 4k     4 + 5i + 6j + 7k
     2 + 3i + 4j + 5k     5 + 6i + 7j + 8k
```

```
q2 = quaternion(magic(4));
B = reshape(q2,2,2)
```

```
B = 2x2 quaternion array
     16 +  2i +  3j + 13k      9 +  7i +  6j + 12k
      5 + 11i + 10j +  8k      4 + 14i + 15j +  1k
```

```
C = A.\B
```

```
C = 2x2 quaternion array
          2.7 -      1.9i -      0.9j -      1.7k       1.5159 -  0.37302i -  0.15079j -  0.0238
       2.2778 +  0.46296i -  0.57407j + 0.092593k       1.2471 +  0.91379i -  0.33908j -   0.109
```

Input Arguments

A — Divisor
scalar | vector | matrix | multidimensional array

Divisor, specified as a quaternion, an array of quaternions, a real scalar, or an array of real numbers.

A and B must have compatible sizes. In the simplest cases, they can be the same size or one can be a scalar. Two inputs have compatible sizes if, for every dimension, the dimension sizes of the inputs are the same or one of the dimensions is 1.

Data Types: quaternion | single | double

B — Dividend
scalar | vector | matrix | multidimensional array

Dividend, specified as a quaternion, an array of quaternions, a real scalar, or an array of real numbers.

A and B must have compatible sizes. In the simplest cases, they can be the same size or one can be a scalar. Two inputs have compatible sizes if, for every dimension, the dimension sizes of the inputs are the same or one of the dimensions is 1.

Data Types: quaternion | single | double

Output Arguments

C — Result
scalar | vector | matrix | multidimensional array

Result of quaternion division, returned as a scalar, vector, matrix, or multidimensional array.

Data Types: quaternion

Algorithms

Quaternion Division

Given a quaternion $A = a_1 + a_2\mathrm{i} + a_3\mathrm{j} + a_4\mathrm{k}$ and a real scalar p,

$$C = p.\backslash A = \frac{a_1}{p} + \frac{a_2}{p}\mathrm{i} + \frac{a_3}{p}\mathrm{j} + \frac{a_4}{p}\mathrm{k}$$

Note For a real scalar p, $A./p = A.\backslash p$.

Quaternion Division by a Quaternion Scalar

Given two quaternions A and B of compatible sizes, then

$$C = A.\backslash B = A^{-1}.*B = \left(\frac{conj(A)}{norm(A)^2}\right).*B$$

Version History

Introduced in R2018b

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

Functions

`.*,times` | `conj` | `norm` | `./,ldivide`

Objects

`quaternion`

log

Natural logarithm of quaternion array

Syntax

```
B = log(A)
```

Description

`B = log(A)` computes the natural logarithm of the elements of the quaternion array `A`.

Examples

Logarithmic Values of Quaternion Array

Create a 3-by-1 quaternion array A.

```
A = quaternion(randn(3,4))
```

```
A = 3x1 quaternion array
     0.53767 + 0.86217i - 0.43359j +  2.7694k
      1.8339 + 0.31877i + 0.34262j -  1.3499k
     -2.2588 -  1.3077i +  3.5784j +  3.0349k
```

Compute the logarithmic values of A.

```
B = log(A)
```

```
B = 3x1 quaternion array
      1.0925 + 0.40848i - 0.20543j +  1.3121k
      0.8436 + 0.14767i + 0.15872j - 0.62533k
      1.6807 - 0.53829i +   1.473j +  1.2493k
```

Input Arguments

A — Input array
scalar | vector | matrix | multidimensional array

Input array, specified as a scalar, vector, matrix, or multidimensional array.

Data Types: `quaternion`

Output Arguments

B — Logarithm values
scalar | vector | matrix | multidimensional array

Quaternion natural logarithm values, returned as a scalar, vector, matrix, or multidimensional array.

Data Types: `quaternion`

Algorithms

Given a quaternion $A = a + \bar{v} = a + b\text{i} + c\text{j} + d\text{k}$, the logarithm is computed by

$$\log(A) = \log\|A\| + \frac{\bar{v}}{\|\bar{v}\|}\arccos\frac{a}{\|A\|}$$

Version History

Introduced in R2018b

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

Functions

`exp` | `.^,power`

Objects

`quaternion`

meanrot

Quaternion mean rotation

Syntax

```
quatAverage = meanrot(quat)
quatAverage = meanrot(quat,dim)
quatAverage = meanrot( ___ ,nanflag)
```

Description

`quatAverage = meanrot(quat)` returns the average rotation of the elements of `quat` along the first array dimension whose size not does equal 1.

- If `quat` is a vector, `meanrot(quat)` returns the average rotation of the elements.
- If `quat` is a matrix, `meanrot(quat)` returns a row vector containing the average rotation of each column.
- If `quat` is a multidimensional array, then `mearot(quat)` operates along the first array dimension whose size does not equal 1, treating the elements as vectors. This dimension becomes 1 while the sizes of all other dimensions remain the same.

The `meanrot` function normalizes the input quaternions, `quat`, before calculating the mean.

`quatAverage = meanrot(quat,dim)` return the average rotation along dimension `dim`. For example, if `quat` is a matrix, then `meanrot(quat,2)` is a column vector containing the mean of each row.

`quatAverage = meanrot( ___ ,nanflag)` specifies whether to include or omit `NaN` values from the calculation for any of the previous syntaxes. `meanrot(quat,'includenan')` includes all `NaN` values in the calculation while `mean(quat,'omitnan')` ignores them.

Examples

Quaternion Mean Rotation

Create a matrix of quaternions corresponding to three sets of Euler angles.

```
eulerAngles = [40 20 10; ...
               50 10 5; ...
               45 70 1];

quat = quaternion(eulerAngles,'eulerd','ZYX','frame');
```

Determine the average rotation represented by the quaternions. Convert the average rotation to Euler angles in degrees for readability.

```
quatAverage = meanrot(quat)
```

```
quatAverage = quaternion
      0.88863 - 0.062598i +  0.27822j +  0.35918k
```

```
eulerAverage = eulerd(quatAverage,'ZYX','frame')
```

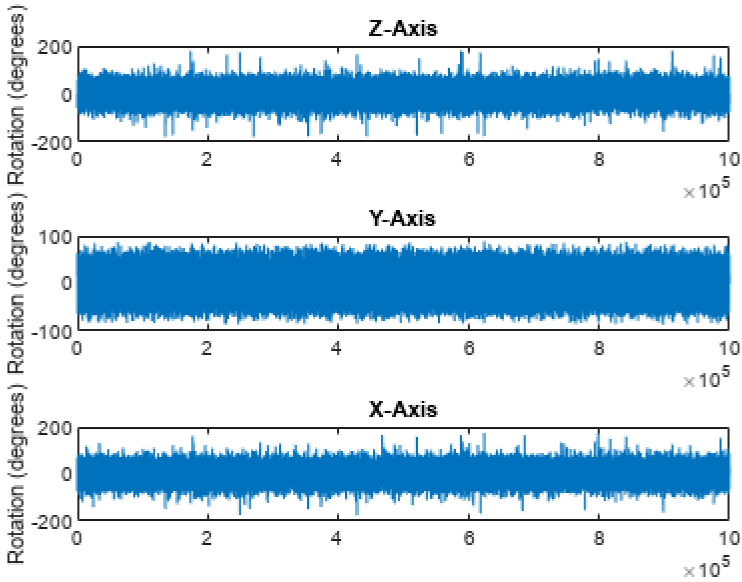
```
eulerAverage = 1×3

   45.7876   32.6452    6.0407
```

Average Out Rotational Noise

Use `meanrot` over a sequence of quaternions to average out additive noise.

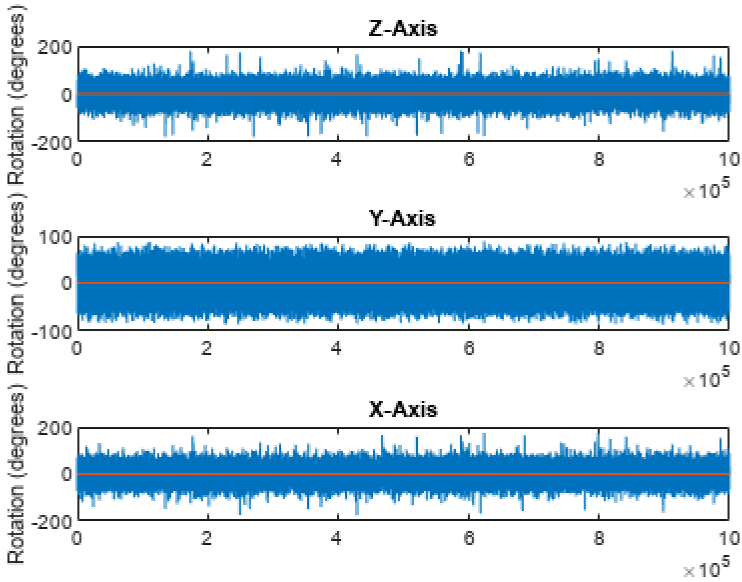
Create a vector of 1e6 quaternions whose distance, as defined by the `dist` function, from quaternion(1,0,0,0) is normally distributed. Plot the Euler angles corresponding to the noisy quaternion vector.

```
nrows = 1e6;
ax = 2*rand(nrows,3) - 1;
ax = ax./sqrt(sum(ax.^2,2));
ang = 0.5*randn(size(ax,1),1);
q = quaternion(ax.*ang ,'rotvec');

noisyEulerAngles = eulerd(q,'ZYX','frame');

figure(1)

subplot(3,1,1)
plot(noisyEulerAngles(:,1))
title('Z-Axis')
ylabel('Rotation (degrees)')
hold on

subplot(3,1,2)
plot(noisyEulerAngles(:,2))
title('Y-Axis')
ylabel('Rotation (degrees)')
hold on

subplot(3,1,3)
plot(noisyEulerAngles(:,3))
title('X-Axis')
ylabel('Rotation (degrees)')
hold on
```



Use `meanrot` to determine the average quaternion given the vector of quaternions. Convert to Euler angles and plot the results.

```
qAverage = meanrot(q);

qAverageInEulerAngles = eulerd(qAverage,'ZYX','frame');

figure(1)

subplot(3,1,1)
plot(ones(nrows,1)*qAverageInEulerAngles(:,1))
title('Z-Axis')

subplot(3,1,2)
plot(ones(nrows,1)*qAverageInEulerAngles(:,2))
title('Y-Axis')

subplot(3,1,3)
plot(ones(nrows,1)*qAverageInEulerAngles(:,3))
title('X-Axis')
```



The `meanrot` Algorithm and Limitations

The `meanrot` Algorithm

The `meanrot` function outputs a quaternion that minimizes the squared Frobenius norm of the difference between rotation matrices. Consider two quaternions:

- `q0` represents no rotation.
- `q90` represents a 90 degree rotation about the *x*-axis.

```
q0 = quaternion([0 0 0],'eulerd','ZYX','frame');
q90 = quaternion([0 0 90],'eulerd','ZYX','frame');
```

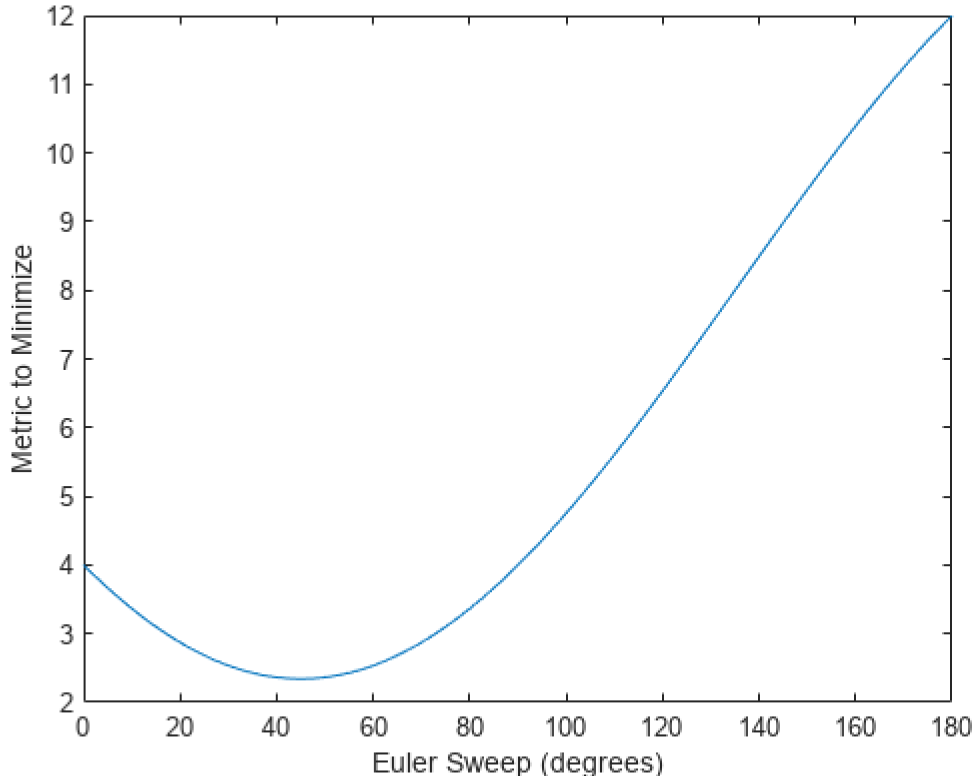
Create a quaternion sweep, `qSweep`, that represents rotations from 0 to 180 degrees about the *x*-axis.

```
eulerSweep = (0:1:180)';
qSweep = quaternion([zeros(numel(eulerSweep),2),eulerSweep], ...
    'eulerd','ZYX','frame');
```

Convert `q0`, `q90`, and `qSweep` to rotation matrices. In a loop, calculate the metric to minimize for each member of the quaternion sweep. Plot the results and return the value of the Euler sweep that corresponds to the minimum of the metric.

```
r0     = rotmat(q0,'frame');
r90    = rotmat(q90,'frame');
rSweep = rotmat(qSweep,'frame');

metricToMinimize = zeros(size(rSweep,3),1);
for i = 1:numel(qSweep)
    metricToMinimize(i) = norm((rSweep(:,:,i) - r0),'fro').^2 + ...
                          norm((rSweep(:,:,i) - r90),'fro').^2;
end
```

```
plot(eulerSweep,metricToMinimize)
xlabel('Euler Sweep (degrees)')
ylabel('Metric to Minimize')
```




```
[~,eulerIndex] = min(metricToMinimize);
eulerSweep(eulerIndex)
```

```
ans = 45
```

The minimum of the metric corresponds to the Euler angle sweep at 45 degrees. That is, `meanrot` defines the average between `quaterion([0 0 0],'ZYX','frame')` and `quaternion([0 0 90],'ZYX','frame')` as `quaternion([0 0 45],'ZYX','frame')`. Call `meanrot` with `q0` and `q90` to verify the same result.

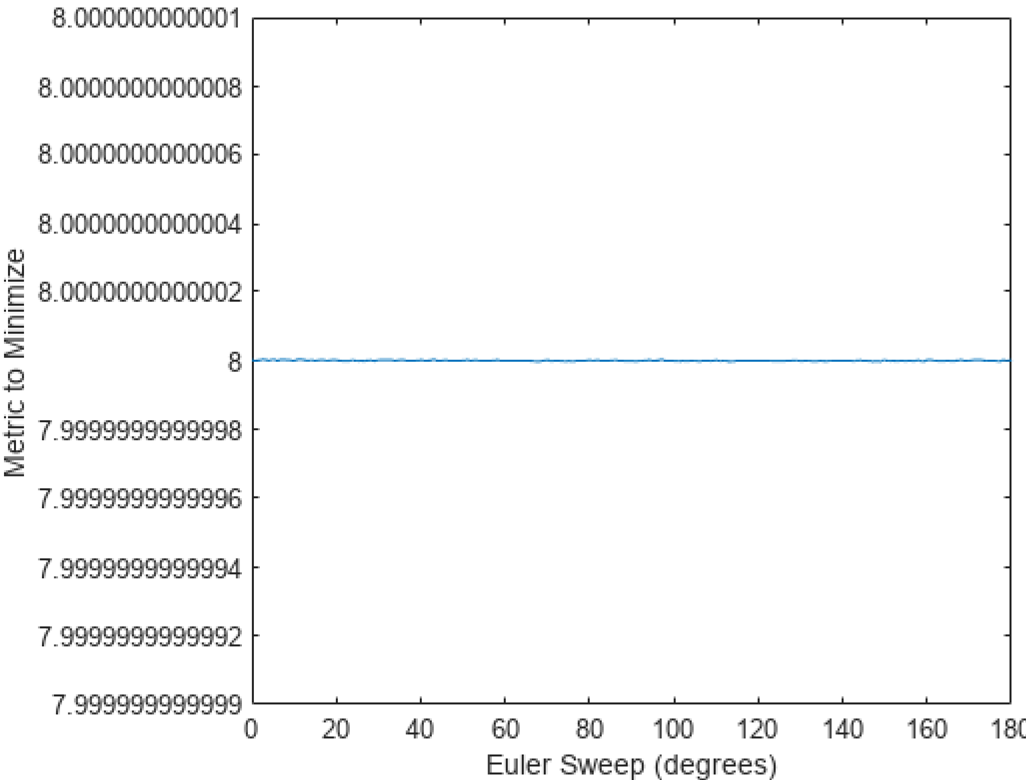
```
eulerd(meanrot([q0,q90]),'ZYX','frame')
```

```
ans = 1×3

         0         0   45.0000
```

Limitations

The metric that `meanrot` uses to determine the mean rotation is not unique for quaternions significantly far apart. Repeat the experiment above for quaternions that are separated by 180 degrees.

```
q180 = quaternion([0 0 180],'eulerd','ZYX','frame');
r180 = rotmat(q180,'frame');

for i = 1:numel(qSweep)
    metricToMinimize(i) = norm((rSweep(:,:,i) - r0),'fro').^2 + ...
                          norm((rSweep(:,:,i) - r180),'fro').^2;
end

plot(eulerSweep,metricToMinimize)
xlabel('Euler Sweep (degrees)')
ylabel('Metric to Minimize')
```




```
[~,eulerIndex] = min(metricToMinimize);
eulerSweep(eulerIndex)
```

```
ans = 159
```

Quaternion means are usually calculated for rotations that are close to each other, which makes the edge case shown in this example unlikely in real-world applications. To average two quaternions that are significantly far apart, use the `slerp` function. Repeat the experiment using `slerp` and verify that the quaternion mean returned is more intuitive for large distances.

```
qMean = slerp(q0,q180,0.5);
q0_q180 = eulerd(qMean,'ZYX','frame')
```

```
q0_q180 = 1×3
```

```
         0         0   90.0000
```

Input Arguments

quat — Quaternion
scalar | vector | matrix | multidimensional array

Quaternion for which to calculate the mean, specified as a scalar, vector, matrix, or multidimensional array of quaternions.

Data Types: `quaternion`

dim — Dimension to operate along
positive integer scalar

Dimension to operate along, specified as a positive integer scalar. If no value is specified, then the default is the first array dimension whose size does not equal 1.

Dimension `dim` indicates the dimension whose length reduces to 1. The `size(quatAverage,dim)` is `1`, while the sizes of all other dimensions remain the same.

Data Types: `double` | `single`

nanflag — NaN condition
`'includenan'` (default) | `'omitnan'`

`NaN` condition, specified as one of these values:

- `'includenan'` –– Include `NaN` values when computing the mean rotation, resulting in `NaN`.
- `'omitnan'` –– Ignore all `NaN` values in the input.

Data Types: `char` | `string`

Output Arguments

quatAverage — Quaternion average rotation
scalar | vector | matrix | multidimensional array

Quaternion average rotation, returned as a scalar, vector, matrix, or multidimensional array.

Data Types: `single` | `double`

Algorithms

`meanrot` determines a quaternion mean, $\bar{q}$, according to [1]. $\bar{q}$ is the quaternion that minimizes the squared Frobenius norm of the difference between rotation matrices:

$$\bar{q} = \arg \min_{q \in S^3} \sum_{i=1}^{n} \|A(q) - A(q_i)\|_F^2$$

Version History

Introduced in R2018b

References

[1] Markley, F. Landis, Yang Chen, John Lucas Crassidis, and Yaakov Oshman. "Average Quaternions." *Journal of Guidance, Control, and Dynamics*. Vol. 30, Issue 4, 2007, pp. 1193-1197.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

Functions
`dist` | `slerp`

Objects
`quaternion`

minjerkpolytraj

Generate minimum jerk trajectory through waypoints

Syntax

```
[q,qd,qdd,qddd,pp,tPoints,tSamples] = minjerkpolytraj(waypoints,timePoints,
numSamples)
[q,qd,qdd,qddd,pp,tPoints,tSamples] = minjerkpolytraj( ___ ,Name=Value)
[q,qd,qdd,qddd,pp,tPoints,tSamples] = minjerkpolytraj( ___
,TimeAllocation=true)
```

Description

`[q,qd,qdd,qddd,pp,tPoints,tSamples] = minjerkpolytraj(waypoints,timePoints, numSamples)` generates a minimum jerk polynomial trajectory that achieves a given set of input waypoints with their corresponding time points. The function returns positions, velocities, accelerations, and jerks at the given number of samples `numSamples`. The function also returns the piecewise polynomial `pp` form of the polynomial trajectory with respect to time, as well as the time points `tPoints` and the sample times `tSamples`.

`[q,qd,qdd,qddd,pp,tPoints,tSamples] = minjerkpolytraj( ___ ,Name=Value)` specifies options using one or more name-value pair arguments in addition to the input arguments from the previous syntax. For example, `minjerkpolytraj(waypoints,timePoints,numSamples,VelocityBoundaryCondition=[1 0 -1 -1; 1 1 1 -1])` generates a two-dimensional minimum jerk trajectory and specifies the velocity boundary conditions in each dimension for each waypoint.

`[q,qd,qdd,qddd,pp,tPoints,tSamples] = minjerkpolytraj( ___ ,TimeAllocation=true)` optimizes a combination of jerk and total segment time cost. In this case, the function treats `timePoints` as an initial guess for the time of arrival at the waypoints.

Examples

Compute Minimum Jerk Trajectory for 2-D Planar Motion

Use the `minjerkpolytraj` function with a given set of 2-D *xy* waypoints. Time points for the waypoints are also given.

```
wpts = [1 4 4 3 -2 0; 0 1 2 4 3 1];
tpts = 0:5;
```

Specify the number of samples in the output trajectory.

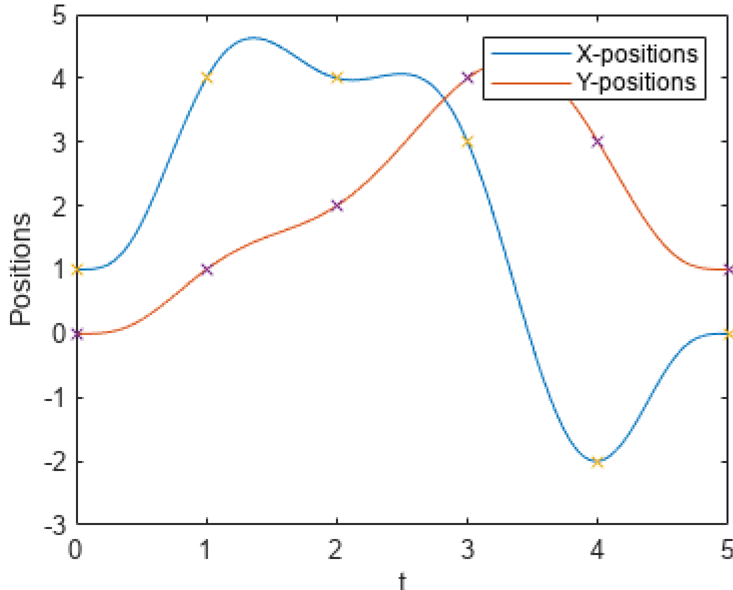
```
numsamples = 100;
```

Compute minimum jerk trajectories. The function outputs the trajectory positions (`q`), velocity (`qd`), acceleration (`qdd`), and jerks (`qddd`) at the given number of samples.

```
[q,qd,qdd,qddd,pp,timepoints,tsamples] = minjerkpolytraj(wpts,tpts,numsamples);
```

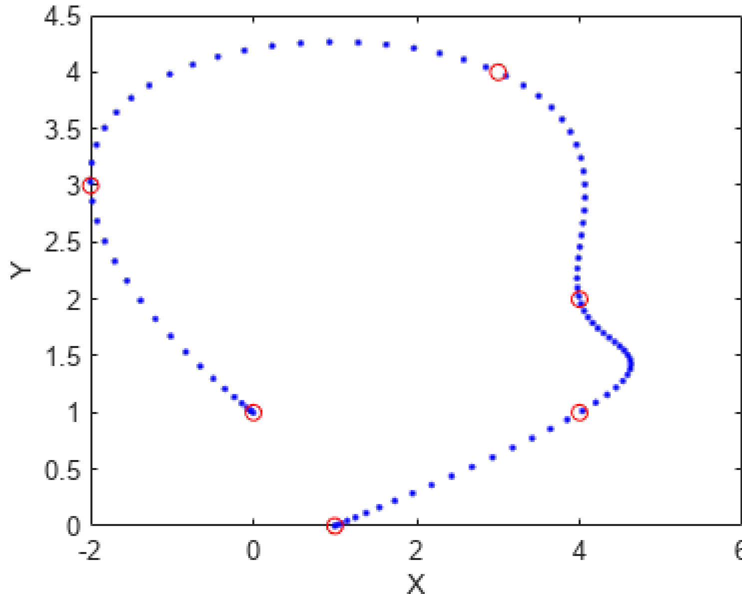
Plot the trajectories for the *x*- and *y*-positions. Compare the trajectory with each waypoint.

```
plot(tsamples,q)
hold on
plot(timepoints,wpts,'x')
xlabel('t')
ylabel('Positions')
legend('X-positions','Y-positions')
hold off
```



You can also verify the actual positions in the 2-D plane. Plot the separate rows of the `q` vector and the waypoints as *x*- and *y*- positions.

```
figure
plot(q(1,:),q(2,:),'.b',wpts(1,:),wpts(2,:),'or')
xlabel('X')
ylabel('Y')
```



Input Arguments

`waypoints` — Waypoints for trajectory
n-by-*p* matrix

Waypoints for the trajectory, specified as an *n*-by-*p* matrix. *n* is the dimension of the trajectory, and *p* is the number of waypoints.

Example: `[2 5 8 4; 3 4 10 12]`

Data Types: `single` | `double`

`timePoints` — Time points for waypoints of trajectory
p-element row vector

Time points for the waypoints of the trajectory, specified as a *p*-element row vector. *p* is the number of waypoints.

Example: `[1 2 3 5]`

Data Types: `single` | `double`

`numSamples` — Number of samples in output trajectory
positive integer

Number of samples in the output trajectory, specified as a positive integer.

Example: `50`

Data Types: `single` | `double`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Example: `minjerkpolytraj(waypoints,timePoints,numSamples,VelocityBoundaryCondition=[1 0 -1 -1; 1 1 1 -1])` generates a two-dimensional minimum jerk trajectory and specifies the velocity boundary conditions in each dimension for each waypoint.

`VelocityBoundaryCondition` — Velocity boundary conditions for each waypoint
n-by-*p* matrix

Velocity boundary conditions for each waypoint, specified as an *n*-by-*p* matrix. Each row sets the velocity boundary for the corresponding dimension of the trajectory at each of *p* waypoints. By default, the function uses a value of `0` at the boundary waypoints and `NaN` at the intermediate waypoints.

Example: `VelocityBoundaryCondition=[1 0 -1 -1; 1 1 1 -1]`

Data Types: `single` | `double`

`AccelerationBoundaryCondition` — Acceleration boundary conditions for each waypoint
n-by-*p* matrix

Acceleration boundary conditions for each waypoint, specified as an *n*-by-*p* matrix. Each row sets the acceleration boundary for the corresponding dimension of the trajectory at each of *p* waypoints. By default, the function uses a value of `0` at the boundary waypoints and `NaN` at the intermediate waypoints.

Example: `AccelerationBoundaryCondition=[1 0 -1 -1; 1 1 1 -1]`

Data Types: `single` | `double`

`JerkBoundaryCondition` — Jerk boundary conditions for each waypoint
n-by-*p* matrix

Jerk boundary conditions for each waypoint, specified as an *n*-by-*p* matrix. Each row sets the jerk boundary for the corresponding dimension of the trajectory at each of *p* waypoints. By default, the function uses a value of `0` at the boundary waypoints and `NaN` at the intermediate waypoints.

Example: `JerkBoundaryCondition=[1 0 -1 -1; 1 1 1 -1]`

Data Types: `single` | `double`

`TimeAllocation` — Time allocation flag
`false` or `0` (default) | `true` or `1`

Time allocation flag, specified as a logical `0` (`false`) or `1` (`true`). Enable this flag to optimize a combination of jerk and total segment time cost.

Note If singularity occurs when the time allocation flag is enabled, reduce the `MaxSegmentTime` to `MinSegmentTime` ratio.

Example: `TimeAllocation=true`

Data Types: `logical`

`TimeWeight` — Weight for time allocation
`100` (default) | positive scalar

Weight for time allocation, specified as a positive scalar.

Example: `TimeWeight=120`

Data Types: `single` | `double`

`MinSegmentTime` — Minimum time segment length
`0.1` (default) | positive scalar | (*p*–1)-element row vector

Minimum time segment length, specified as a positive scalar or (*p*–1)-element row vector.

Example: `MinSegmentTime=0.2`

Data Types: `single` | `double`

`MaxSegmentTime` — Maximum time segment length
`5` (default) | positive scalar | (*p*–1)-element row vector

Maximum time segment length, specified as a positive scalar or (*p*–1)-element row vector

Example: `MaxSegmentTime=10`

Data Types: `single` | `double`

Output Arguments

q — Positions of trajectory
n-by-*m* matrix

Positions of the trajectory at the given time samples in `tSamples`, returned as an *n*-by-*m* matrix. *n* is the dimension of the trajectory, and *m* is equal to `numSamples`.

qd — Velocities of trajectory
n-by-*m* matrix

Velocities of the trajectory at the given time samples in `tSamples`, returned as an *n*-by-*m* matrix. *n* is the dimension of the trajectory, and *m* is equal to `numSamples`.

qdd — Accelerations of trajectory
n-by-*m* matrix

Accelerations of the trajectory at the given time samples in `tSamples`, returned as an *n*-by-*m* matrix. *n* is the dimension of the trajectory, and *m* is equal to `numSamples`.

qddd — Jerks of trajectory
n-by-*m* matrix

Jerks of the trajectory at the given time samples in `tSamples`, returned as an *n*-by-*m* matrix. *n* is the dimension of the trajectory, and *m* is equal to `numSamples`.

pp — Piecewise polynomial
structure

Piecewise-polynomial, returned as a structure that defines the polynomial for each section of the piecewise trajectory. You can build your own piecewise polynomials using `mkpp`, or evaluate the polynomial at specified times using `ppval`. The structure contains the fields:

- `form`: `'pp'`.
- `breaks`: *p*-element vector of times when the piecewise trajectory changes forms. *p* is the number of waypoints.
- `coefs`: *n*(*p*–`1`)-by-`order` matrix for the coefficients for the polynomials. *n*(*p*–`1`) is the dimension of the trajectory times the number of `pieces`. Each set of *n* rows defines the coefficients for the polynomial that described each variable trajectory.
- `pieces`: *p*–`1`. The number of breaks minus 1.
- `order`: Degree of the polynomial + `1`. The order of polynomial is 8.
- `dim`: *n*. The dimension of the control point positions.

`tPoints` — Time points for waypoints of trajectory
p-element row vector

Time points for the waypoints of the trajectory, returned as a *p*-element row vector. *p* is the number of waypoints.

`tSamples` — Time samples for trajectory
m-element row vector

Time samples for the trajectory, returned as an *m*-element row vector. Each element of the output position `q`, velocity `qd`, acceleration `qdd`, and jerk `qddd` has been sampled at the corresponding time in this vector.

Version History

Introduced in R2021b

References

[1] Bry, Adam, Charles Richter, Abraham Bachrach, and Nicholas Roy. “Aggressive Flight of Fixed-Wing and Quadrotor Aircraft in Dense Indoor Environments.” *The International Journal of Robotics Research*, 34, no. 7 (June 2015): 969–1002.

[2] Richter, Charles, Adam Bry, and Nicholas Roy. “Polynomial Trajectory Planning for Aggressive Quadrotor Flight in Dense Indoor Environments." *Paper presented at the International Symposium of Robotics Research (ISRR 2013)*, 2013.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

Functions
bsplinepolytraj | cubicpolytraj | quinticpolytraj | trapveltraj | minsnappolytraj | kinematicConstrainedTraj

minsnappolytraj

Generate minimum snap trajectory through waypoints

Syntax

[q,qd,qdd,qddd,qdddd,pp,tPoints,tSamples] = minsnappolytraj(waypoints,
timePoints,numSamples)
[q,qd,qdd,qddd,qdddd,pp,tPoints,tSamples] = minsnappolytraj(___ ,Name=Value)
[q,qd,qdd,qddd,qdddd,pp,tPoints,tSamples] = minsnappolytraj(___
,TimeAllocation=true)

Description

[q,qd,qdd,qddd,qdddd,pp,tPoints,tSamples] = minsnappolytraj(waypoints,
timePoints,numSamples) generates a minimum snap polynomial trajectory that achieves a given set of input waypoints with their corresponding time points. The function returns positions, velocities, accelerations, jerks, and snaps at the given number of samples numSamples. The function also returns the piecewise polynomial pp form of the polynomial trajectory with respect to time, as well as the time points tPoints, and the sample times tSamples.

[q,qd,qdd,qddd,qdddd,pp,tPoints,tSamples] = minsnappolytraj(___ ,Name=Value) specifies options using one or more name-value pair arguments in addition to the input arguments from the previous syntax. For example,
minsnappolytraj(waypoints,timePoints,numSamples,VelocityBoundaryCondition=[1
0 -1 -1; 1 1 1 -1]) generates a two-dimensional minimum snap trajectory and specifies the velocity boundary conditions in each dimension for each waypoint.

[q,qd,qdd,qddd,qdddd,pp,tPoints,tSamples] = minsnappolytraj(___
,TimeAllocation=true) optimizes a combination of snap and the total segment time cost. In this case, the function treats timePoints as an initial guess for the time of arrival at the waypoints.

Examples

Compute Minimum Snap Trajectory for 2-D Planar Motion

Use the minsnappolytraj function with a given set of 2-D *xy* waypoints. Time points for the waypoints are also given.

```
wpts = [1 4 4 3 -2 0; 0 1 2 4 3 1];
tpts = 0:5;
```

Specify the number of samples in the output trajectory.

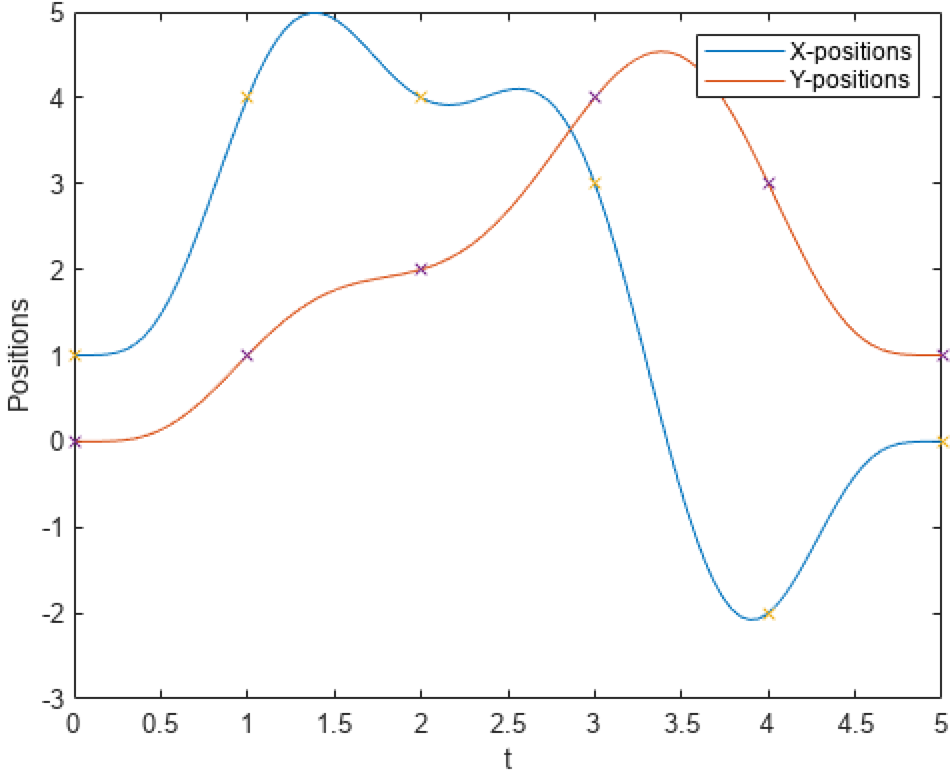
```
numsamples = 100;
```

Compute minimum snap trajectories. The function outputs the trajectory positions (q), velocity (qd), acceleration (qdd), jerks (qddd), and snaps (qdddd) at the given number of samples.

```
[q,qd,qdd,qddd,qdddd,pp,timepoints,tsamples] = minsnappolytraj(wpts,tpts,numsamples);
```

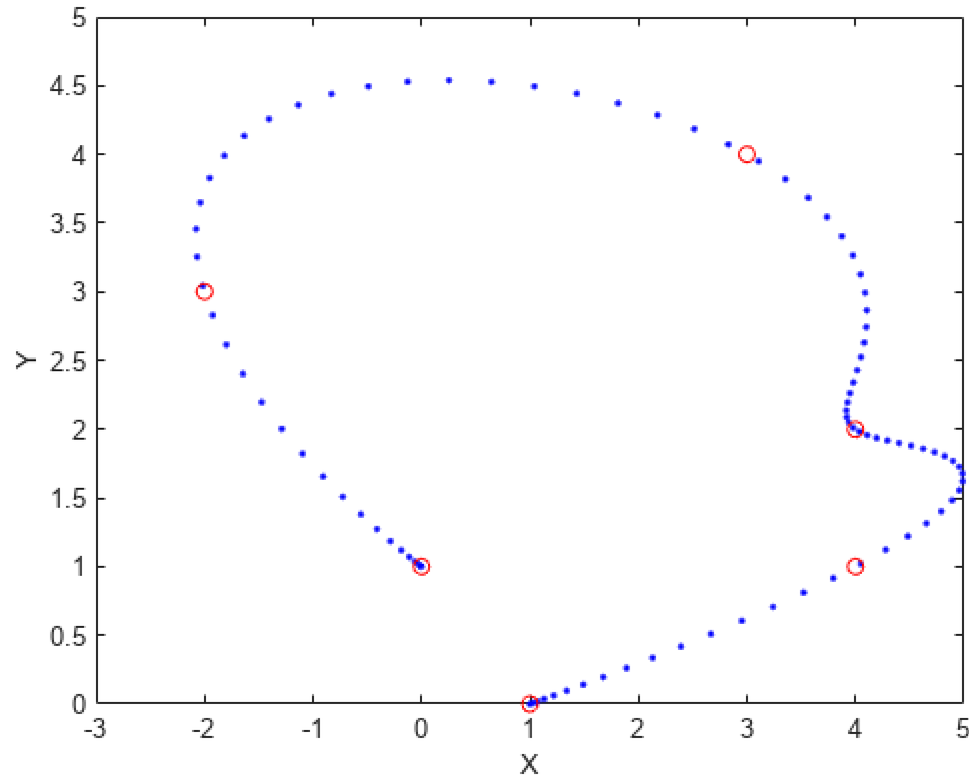
Plot the trajectories for the *x*- and *y*-positions. Compare the trajectory with each waypoint.

```
plot(tsamples,q)
hold on
plot(timepoints,wpts,'x')
xlabel('t')
ylabel('Positions')
legend('X-positions','Y-positions')
hold off
```



You can also verify the actual positions in the 2-D plane. Plot the separate rows of the `q` vector and the waypoints as *x*- and *y*- positions.

```
figure
plot(q(1,:),q(2,:),'.b',wpts(1,:),wpts(2,:),'or')
xlabel('X')
ylabel('Y')
```



Input Arguments

`waypoints` — Waypoints for trajectory
n-by-*p* matrix

Waypoints for the trajectory, specified as an *n*-by-*p* matrix. *n* is the dimension of the trajectory, and *p* is the number of waypoints.

Example: `[2 5 8 4; 3 4 10 12]`

Data Types: `single` | `double`

`timePoints` — Time points for waypoints of trajectory
p-element row vector

Time points for the waypoints of the trajectory, specified as a *p*-element row vector. *p* is the number of waypoints.

Example: `[1 2 3 5]`

Data Types: `single` | `double`

`numSamples` — Number of samples in output trajectory
positive integer

Number of samples in the output trajectory, specified as a positive integer.

Example: `50`

Data Types: `single` | `double`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Example: `minsnappolytraj(waypoints,timePoints,numSamples,VelocityBoundaryCondition=[1 0 -1 -1; 1 1 1 -1])` generates a two-dimensional minimum snap trajectory and specifies the velocity boundary conditions in each dimension for each waypoint.

`VelocityBoundaryCondition` — Velocity boundary conditions for each waypoint

n-by-*p* matrix

Velocity boundary conditions for each waypoint, specified as an *n*-by-*p* matrix. Each row sets the velocity boundary for the corresponding dimension of the trajectory at each of *p* waypoints. By default, the function uses a value of `0` at the boundary waypoints and `NaN` at the intermediate waypoints.

Example: `VelocityBoundaryCondition=[1 0 -1 -1; 1 1 1 -1]`

Data Types: `single` | `double`

`AccelerationBoundaryCondition` — Acceleration boundary conditions for each waypoint

n-by-*p* matrix

Acceleration boundary conditions for each waypoint, specified as an *n*-by-*p* matrix. Each row sets the acceleration boundary for the corresponding dimension of the trajectory at each of *p* waypoints. By default, the function uses a value of `0` at the boundary waypoints and `NaN` at the intermediate waypoints.

Example: `AccelerationBoundaryCondition=[1 0 -1 -1; 1 1 1 -1]`

Data Types: `single` | `double`

`JerkBoundaryCondition` — Jerk boundary conditions for each waypoint

n-by-*p* matrix

Jerk boundary conditions for each waypoint, specified as an *n*-by-*p* matrix. Each row sets the jerk boundary for the corresponding dimension of the trajectory at each of *p* waypoints. By default, the function uses a value of `0` at the boundary waypoints and `NaN` at the intermediate waypoints.

Example: `JerkBoundaryCondition=[1 0 -1 -1; 1 1 1 -1]`

Data Types: `single` | `double`

`SnapBoundaryCondition` — Snap boundary conditions for each waypoint

n-by-*p* matrix

Snap boundary conditions for each waypoint, specified as an *n*-by-*p* matrix. Each row sets the snap boundary for the corresponding dimension of the trajectory at each of *p* waypoints. By default, the function uses a value of `0` at the boundary waypoints and `NaN` at the intermediate waypoints.

Example: `SnapBoundaryCondition=[1 0 -1 -1; 1 1 1 -1]`

Data Types: `single` | `double`

`TimeAllocation` — Time allocation flag
`false` or `0` (default) | `true` or `1`

Time allocation flag, specified as a logical `0` (`false`) or `1` (`true`). Enable this flag to optimize a combination of snap and total segment time cost.

Note If singularity occurs when the time allocation flag is enabled, reduce the `MaxSegmentTime` to `MinSegmentTime` ratio.

Example: `TimeAllocation=true`

Data Types: `logical`

`TimeWeight` — Weight for time allocation
`100` (default) | positive scalar

Weight for time allocation, specified as a positive scalar.

Example: `TimeWeight=120`

Data Types: `single` | `double`

`MinSegmentTime` — Minimum time segment length
`0.1` (default) | positive scalar | (p–1)-element row vector

Minimum time segment length, specified as a positive scalar or (p–1)-element row vector.

Example: `MinSegmentTime=0.2`

Data Types: `single` | `double`

`MaxSegmentTime` — Maximum time segment length
`1` (default) | positive scalar | (p–1)-element row vector

Maximum time segment length, specified as a positive scalar or (p–1)-element row vector

Example: `MaxSegmentTime=5`

Data Types: `single` | `double`

Output Arguments

q — Positions of trajectory
n-by-m matrix

Positions of the trajectory at the given time samples in `tSamples`, returned as an n-by-m matrix. n is the dimension of the trajectory, and m is equal to `numSamples`.

qd — Velocities of trajectory
n-by-m matrix

Velocities of the trajectory at the given time samples in `tSamples`, returned as an n-by-m matrix. n is the dimension of the trajectory, and m is equal to `numSamples`.

qdd — Accelerations of trajectory
n-by-*m* matrix

Accelerations of the trajectory at the given time samples in `tSamples`, returned as an *n*-by-*m* matrix. *n* is the dimension of the trajectory, and *m* is equal to `numSamples`.

qddd — Jerks of trajectory
n-by-*m* matrix

Jerks of the trajectory at the given time samples in `tSamples`, returned as an *n*-by-*m* matrix. *n* is the dimension of the trajectory, and *m* is equal to `numSamples`.

qdddd — Snaps of trajectory
n-by-*m* matrix

Snaps of the trajectory at the given time samples in `tSamples`, returned as an *n*-by-*m* matrix. *n* is the dimension of the trajectory, and *m* is equal to `numSamples`.

pp — Piecewise polynomial
structure

Piecewise-polynomial, returned as a structure that defines the polynomial for each section of the piecewise trajectory. You can build your own piecewise polynomials using `mkpp`, or evaluate the polynomial at specified times using `ppval`. The structure contains the fields:

- `form`: `'pp'`.
- `breaks`: *p*-element vector of times when the piecewise trajectory changes forms. *p* is the number of waypoints.
- `coefs`: *n*(*p*–1)-by-`order` matrix for the coefficients for the polynomials. *n*(*p*–1) is the dimension of the trajectory times the number of `pieces`. Each set of *n* rows defines the coefficients for the polynomial that described each variable trajectory.
- `pieces`: *p*–1. The number of breaks minus 1.
- `order`: Degree of the polynomial + 1. The order of polynomial is 10.
- `dim`: *n*. The dimension of the control point positions.

`tPoints` — Time points for waypoints of trajectory
p-element row vector

Time points for the waypoints of the trajectory, returned as a *p*-element row vector. *p* is the number of waypoints.

`tSamples` — Time samples for trajectory
m-element row vector

Time samples for the trajectory, returned as an *m*-element row vector. Each element of the output position `q`, velocity `qd`, acceleration `qdd`, jerk `qddd`, and snap `qdddd` has been sampled at the corresponding time in this vector.

Version History

Introduced in R2021b

References

[1] Bry, Adam, Charles Richter, Abraham Bachrach, and Nicholas Roy. "Aggressive Flight of Fixed-Wing and Quadrotor Aircraft in Dense Indoor Environments." *The International Journal of Robotics Research*, 34, no. 7 (June 2015): 969–1002.

[2] Richter, Charles, Adam Bry, and Nicholas Roy. "Polynomial Trajectory Planning for Aggressive Quadrotor Flight in Dense Indoor Environments." *Paper presented at the International Symposium of Robotics Research (ISRR 2013)*, 2013.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

Functions
`bsplinepolytraj` | `cubicpolytraj` | `quinticpolytraj` | `trapveltraj` | `minjerkpolytraj` | `kinematicConstrainedTraj`

minus, -

Quaternion subtraction

Syntax

```
C = A - B
```

Description

`C = A - B` subtracts quaternion `B` from quaternion `A` using quaternion subtraction. Either `A` or `B` may be a real number, in which case subtraction is performed with the real part of the quaternion argument.

Examples

Subtract a Quaternion from a Quaternion

Quaternion subtraction is defined as the subtraction of the corresponding parts of each quaternion. Create two quaternions and perform subtraction.

```
Q1 = quaternion([1,0,-2,7]);
Q2 = quaternion([1,2,3,4]);

Q1minusQ2 = Q1 - Q2
```

```
Q1minusQ2 = quaternion
     0 - 2i - 5j + 3k
```

Subtract a Real Number from a Quaternion

Addition and subtraction of real numbers is defined for quaternions as acting on the real part of the quaternion. Create a quaternion and then subtract 1 from the real part.

```
Q = quaternion([1,1,1,1])
```

```
Q = quaternion
     1 + 1i + 1j + 1k
```

```
Qminus1 = Q - 1
```

```
Qminus1 = quaternion
     0 + 1i + 1j + 1k
```

Input Arguments

A — Input
scalar | vector | matrix | multidimensional array

Input, specified as a quaternion, array of quaternions, real number, or array of real numbers.

Data Types: `quaternion` | `single` | `double`

B — Input
scalar | vector | matrix | multidimensional array

Input, specified as a quaternion, array of quaternions, real number, or array of real numbers.

Data Types: `quaternion` | `single` | `double`

Output Arguments

C — Result
scalar | vector | matrix | multidimensional array

Result of quaternion subtraction, returned as a scalar, vector, matrix, or multidimensional array of quaternions.

Data Types: `quaternion`

Version History

Introduced in R2018a

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

Functions
`-,uminus` | `.*,times` | `*,mtimes`

Objects
`quaternion`

mtimes, *

Quaternion multiplication

Syntax

```
quatC = A*B
```

Description

`quatC = A*B` implements quaternion multiplication if either `A` or `B` is a quaternion. Either `A` or `B` must be a scalar.

You can use quaternion multiplication to compose rotation operators:

- To compose a sequence of frame rotations, multiply the quaternions in the order of the desired sequence of rotations. For example, to apply a p quaternion followed by a q quaternion, multiply in the order pq. The rotation operator becomes $(pq)^{*}v(pq)$, where v represents the object to rotate specified in quaternion form. * represents conjugation.
- To compose a sequence of point rotations, multiply the quaternions in the reverse order of the desired sequence of rotations. For example, to apply a p quaternion followed by a q quaternion, multiply in the reverse order, qp. The rotation operator becomes $(qp)v(qp)^{*}$.

Examples

Multiply Quaternion Scalar and Quaternion Vector

Create a 4-by-1 column vector, `A`, and a scalar, `b`. Multiply `A` times `b`.

```
A = quaternion(randn(4,4))
```

```
A = 4x1 quaternion array
      0.53767 +  0.31877i +   3.5784j +   0.7254k
       1.8339 -   1.3077i +   2.7694j - 0.063055k
      -2.2588 -  0.43359i -   1.3499j +  0.71474k
      0.86217 +  0.34262i +   3.0349j -  0.20497k
```

```
b = quaternion(randn(1,4))
```

```
b = quaternion
    -0.12414 +  1.4897i +   1.409j +  1.4172k
```

```
C = A*b
```

```
C = 4x1 quaternion array
      -6.6117 +   4.8105i +  0.94224j -   4.2097k
      -2.0925 +   6.9079i +   3.9995j -   3.3614k
       1.8155 -   6.2313i -    1.336j -     1.89k
      -4.6033 +   5.8317i + 0.047161j -    2.791k
```

Input Arguments

A — Input
scalar | vector | matrix | multidimensional array

Input to multiply, specified as a quaternion, array of quaternions, real scalar, or array of real scalars.

If B is nonscalar, then A must be scalar.

Data Types: `quaternion` | `single` | `double`

B — Input
scalar | vector | matrix | multidimensional array

Input to multiply, specified as a quaternion, array of quaternions, real scalar, or array of real scalars.

If A is nonscalar, then B must be scalar.

Data Types: `quaternion` | `single` | `double`

Output Arguments

quatC — Quaternion product
scalar | vector | matrix | multidimensional array

Quaternion product, returned as a quaternion or array of quaternions.

Data Types: `quaternion`

Algorithms

Quaternion Multiplication by a Real Scalar

Given a quaternion

$$q = a_{\mathrm{q}} + b_{\mathrm{q}}\mathrm{i} + c_{\mathrm{q}}\mathrm{j} + d_{\mathrm{q}}\mathrm{k},$$

the product of q and a real scalar β is

$$\beta q = \beta a_{\mathrm{q}} + \beta b_{\mathrm{q}}\mathrm{i} + \beta c_{\mathrm{q}}\mathrm{j} + \beta d_{\mathrm{q}}\mathrm{k}$$

Quaternion Multiplication by a Quaternion Scalar

The definition of the basis elements for quaternions,

$$\mathrm{i}^2 = \mathrm{j}^2 = \mathrm{k}^2 = \mathrm{ijk} = -1\,,$$

can be expanded to populate a table summarizing quaternion basis element multiplication:

	1	**i**	**j**	**k**
1	1	i	j	k
i	i	−1	k	−j

j	j	−k	−1	i
k	k	j	−i	−1

When reading the table, the rows are read first, for example: ij = k and ji = −k.

Given two quaternions, $q = a_q + b_q\mathrm{i} + c_q\mathrm{j} + d_q\mathrm{k}$, and $p = a_p + b_p\mathrm{i} + c_p\mathrm{j} + d_p\mathrm{k}$, the multiplication can be expanded as:

$$\begin{aligned} z = pq &= (a_p + b_p\mathrm{i} + c_p\mathrm{j} + d_p\mathrm{k})(a_q + b_q\mathrm{i} + c_q\mathrm{j} + d_q\mathrm{k}) \\ &= a_pa_q + a_pb_q\mathrm{i} + a_pc_q\mathrm{j} + a_pd_q\mathrm{k} \\ &+ b_pa_q\mathrm{i} + b_pb_q\mathrm{i}^2 + b_pc_q\mathrm{ij} + b_pd_q\mathrm{ik} \\ &+ c_pa_q\mathrm{j} + c_pb_q\mathrm{ji} + c_pc_q\mathrm{j}^2 + c_pd_q\mathrm{jk} \\ &+ d_pa_qk + d_pb_q\mathrm{ki} + d_pc_q\mathrm{kj} + d_pd_q\mathrm{k}^2 \end{aligned}$$

You can simplify the equation using the quaternion multiplication table:

$$\begin{aligned} z = pq &= a_pa_q + a_pb_q\mathrm{i} + a_pc_q\mathrm{j} + a_pd_q\mathrm{k} \\ &+ b_pa_q\mathrm{i} - b_pb_q + b_pc_q\mathrm{k} - b_pd_q\mathrm{j} \\ &+ c_pa_q\mathrm{j} - c_pb_q\mathrm{k} - c_pc_q + c_pd_q\mathrm{i} \\ &+ d_pa_qk + d_pb_q\mathrm{j} - d_pc_q\mathrm{i} - d_pd_q \end{aligned}$$

Version History

Introduced in R2018a

References

[1] Kuipers, Jack B. *Quaternions and Rotation Sequences: A Primer with Applications to Orbits, Aerospace, and Virtual Reality*. Princeton, NJ: Princeton University Press, 2007.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

Functions

`.*`,`times`

Objects

`quaternion`

norm

Quaternion norm

Syntax

```
N = norm(quat)
```

Description

`N = norm(quat)` returns the norm of the quaternion, `quat`.

Given a quaternion of the form $Q = a + b\mathrm{i} + c\mathrm{j} + d\mathrm{k}$, the norm of the quaternion is defined as $\mathrm{norm}(Q) = \sqrt{a^2 + b^2 + c^2 + d^2}$.

Examples

Calculate Quaternion Norm

Create a scalar quaternion and calculate its norm.

```
quat = quaternion(1,2,3,4);
norm(quat)
```

```
ans = 5.4772
```

The quaternion norm is defined as the square root of the sum of the quaternion parts squared. Calculate the quaternion norm explicitly to verify the result of the `norm` function.

```
[a,b,c,d] = parts(quat);
sqrt(a^2+b^2+c^2+d^2)
```

```
ans = 5.4772
```

Input Arguments

quat — Quaternion
scalar | vector | matrix | multidimensional array

Quaternion for which to calculate the norm, specified as a scalar, vector, matrix, or multidimensional array of quaternions.

Data Types: `quaternion`

Output Arguments

N — Quaternion norm
scalar | vector | matrix | multidimensional array

Quaternion norm. If the input `quat` is an array, the output is returned as an array the same size as `quat`. Elements of the array are real numbers with the same data type as the underlying data type of the quaternion, `quat`.

Data Types: `single` | `double`

Version History

Introduced in R2018a

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

Functions

`normalize` | `parts` | `conj`

Objects

`quaternion`

normalize

Quaternion normalization

Syntax

```
quatNormalized = normalize(quat)
```

Description

`quatNormalized = normalize(quat)` normalizes the quaternion.

Given a quaternion of the form $Q = a + b\text{i} + c\text{j} + d\text{k}$, the normalized quaternion is defined as $Q/\sqrt{a^2 + b^2 + c^2 + d^2}$.

Examples

Normalize Elements of Quaternion Vector

Quaternions can represent rotations when normalized. You can use `normalize` to normalize a scalar, elements of a matrix, or elements of a multi-dimensional array of quaternions. Create a column vector of quaternions, then normalize them.

```
quatArray = quaternion([1,2,3,4; ...
                        2,3,4,1; ...
                        3,4,1,2]);
quatArrayNormalized = normalize(quatArray)
```

```
quatArrayNormalized = 3x1 quaternion array
     0.18257 + 0.36515i + 0.54772j +  0.7303k
     0.36515 + 0.54772i +  0.7303j + 0.18257k
     0.54772 +  0.7303i + 0.18257j + 0.36515k
```

Input Arguments

quat — Quaternion to normalize
scalar | vector | matrix | multidimensional array

Quaternion to normalize, specified as a scalar, vector, matrix, or multidimensional array of quaternions.

Data Types: `quaternion`

Output Arguments

quatNormalized — Normalized quaternion
scalar | vector | matrix | multidimensional array

Normalized quaternion, returned as a quaternion or array of quaternions the same size as `quat`.

Data Types: `quaternion`

Version History

Introduced in R2018a

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

Functions

`norm` | `.*,times` | `conj`

Objects

`quaternion`

ones

Create quaternion array with real parts set to one and imaginary parts set to zero

Syntax

```
quatOnes = ones('quaternion')
quatOnes = ones(n,'quaternion')
quatOnes = ones(sz,'quaternion')
quatOnes = ones(sz1,...,szN,'quaternion')

quatOnes = ones( ___ ,'like',prototype,'quaternion')
```

Description

`quatOnes = ones('quaternion')` returns a scalar quaternion with the real part set to 1 and the imaginary parts set to 0.

Given a quaternion of the form $Q = a + b\text{i} + c\text{j} + d\text{k}$, a quaternion one is defined as $Q = 1 + 0\text{i} + 0\text{j} + 0\text{k}$.

`quatOnes = ones(n,'quaternion')` returns an `n`-by-`n` quaternion matrix with the real parts set to 1 and the imaginary parts set to 0.

`quatOnes = ones(sz,'quaternion')` returns an array of quaternion ones where the size vector, `sz`, defines `size(qOnes)`.

Example: `ones([1,4,2],'quaternion')` returns a 1-by-4-by-2 array of quaternions with the real parts set to `1` and the imaginary parts set to `0`.

`quatOnes = ones(sz1,...,szN,'quaternion')` returns a `sz1`-by-...-by-`szN` array of ones where `sz1,…,szN` indicates the size of each dimension.

`quatOnes = ones( ___ ,'like',prototype,'quaternion')` specifies the underlying class of the returned quaternion array to be the same as the underlying class of the quaternion prototype.

Examples

Quaternion Scalar One

Create a quaternion scalar one.

```
quatOnes = ones('quaternion')
```

```
quatOnes = quaternion
     1 + 0i + 0j + 0k
```

Square Matrix of Quaternion Ones

Create an n-by-n matrix of quaternion ones.

```
n = 3;
quatOnes = ones(n,'quaternion')
```

```
quatOnes = 3x3 quaternion array
     1 + 0i + 0j + 0k     1 + 0i + 0j + 0k     1 + 0i + 0j + 0k
     1 + 0i + 0j + 0k     1 + 0i + 0j + 0k     1 + 0i + 0j + 0k
     1 + 0i + 0j + 0k     1 + 0i + 0j + 0k     1 + 0i + 0j + 0k
```

Multidimensional Array of Quaternion Ones

Create a multidimensional array of quaternion ones by defining array dimensions in order. In this example, you create a 3-by-1-by-2 array. You can specify dimensions using a row vector or comma-separated integers. Specify the dimensions using a row vector and display the results:

```
dims = [3,1,2];
quatOnesSyntax1 = ones(dims,'quaternion')
```

```
quatOnesSyntax1 = 3x1x2 quaternion array
quatOnesSyntax1(:,:,1) =

     1 + 0i + 0j + 0k
     1 + 0i + 0j + 0k
     1 + 0i + 0j + 0k


quatOnesSyntax1(:,:,2) =

     1 + 0i + 0j + 0k
     1 + 0i + 0j + 0k
     1 + 0i + 0j + 0k
```

Specify the dimensions using comma-separated integers, and then verify the equivalency of the two syntaxes:

```
quatOnesSyntax2 = ones(3,1,2,'quaternion');
isequal(quatOnesSyntax1,quatOnesSyntax2)
```

```
ans = logical
   1
```

Underlying Class of Quaternion Ones

A quaternion is a four-part hyper-complex number used in three-dimensional rotations and orientations. You can specify the underlying data type of the parts as `single` or `double`. The default is `double`.

Create a quaternion array of ones with the underlying data type set to `single`.

```
quatOnes = ones(2,'like',single(1),'quaternion')
```

```
quatOnes = 2x2 quaternion array
     1 + 0i + 0j + 0k     1 + 0i + 0j + 0k
     1 + 0i + 0j + 0k     1 + 0i + 0j + 0k
```

Verify the underlying class using the `classUnderlying` function.

```
classUnderlying(quatOnes)
```

```
ans =
'single'
```

Input Arguments

n — Size of square quaternion matrix
integer value

Size of square quaternion matrix, specified as an integer value.

If `n` is zero or negative, then `quatOnes` is returned as an empty matrix.

Example: `ones(4,'quaternion')` returns a 4-by-4 matrix of quaternions with the real parts set to `1` and the imaginary parts set to `0`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

sz — Output size
row vector of integer values

Output size, specified as a row vector of integer values. Each element of `sz` indicates the size of the corresponding dimension in `quatOnes`. If the size of any dimension is `0` or negative, then `quatOnes` is returned as an empty array.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

prototype — Quaternion prototype
variable

Quaternion prototype, specified as a variable.

Example: `ones(2,'like',quat,'quaternion')` returns a 2-by-2 matrix of quaternions with the same underlying class as the prototype quaternion, `quat`.

Data Types: `quaternion`

sz1,...,szN — Size of each dimension
two or more integer values

Size of each dimension, specified as two or more integers. If the size of any dimension is `0` or negative, then `quatOnes` is returned as an empty array.

Example: `ones(2,3,'quaternion')` returns a 2-by-3 matrix of quaternions with the real parts set to `1` and the imaginary parts set to `0`.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

Output Arguments

quatOnes — Quaternion ones
scalar | vector | matrix | multidimensional array

Quaternion ones, returned as a scalar, vector, matrix, or multidimensional array of quaternions.

Given a quaternion of the form $Q = a + b\text{i} + c\text{j} + d\text{k}$, a quaternion one is defined as $Q = 1 + 0\text{i} + 0\text{j} + 0\text{k}$.

Data Types: quaternion

Version History

Introduced in R2018a

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

Functions
zeros

Objects
quaternion

packageGazeboPlugin

Create Gazebo plugin package for Simulink

Syntax

```
packageGazeboPlugin
packageGazeboPlugin(packagePath)
packageGazeboPlugin(packagePath,customMessagePath)
outputPath = packageGazeboPlugin( ___ )
```

Description

`packageGazeboPlugin` creates a Gazebo plugin package as a zip archive. The function creates a folder containing plugin source code, named `GazeboPlugin`, in the current working directory and compresses it as `GazeboPlugin.zip`. Gazebo uses this plugin package to communicate with Simulink for synchronized stepping, as well as sending and receiving messages.

`packageGazeboPlugin(packagePath)` creates a Gazebo plugin at the specified location `packagePath`. `packagePath` must be a valid file name or a file path with the desired package folder name. The function creates the plugin folder with the specified name in the location specified in the `packagePath` argument and compresses it.

`packageGazeboPlugin(packagePath,customMessagePath)` creates a Gazebo plugin with custom message support using the specified custom message dependencies in `customMessagePath`. The dependencies must be specified as a valid path to a folder that contains the custom message dependencies.

`outputPath = packageGazeboPlugin( ___ )` returns the path of the plugin folder in addition to any combination of input arguments from a previous syntax.

Examples

Generate Dependencies for User-Defined Gazebo Custom Message

Create a folder in a local directory.

```
folderPath = fullfile(pwd,'customMessage')
```

```
folderPath = 
'C:\TEMP\Bdoc23a_2213998_3568\ib570499\13\tp73e6043f\robotics-ex62907275\customMessage'
```

```
mkdir(folderPath)
```

Create a `.proto` file inside the folder and define protobuf custom message fields.

```
messageDefinition = {'message MyPose'
                     '{'
                     '    required double x = 1;'
                     '    required double y = 2;'
                     '    required double z = 3;'
```

```
                     '}'};
fileID = fopen(fullfile(folderPath,'MyPose.proto'),'w');
fprintf(fileID,'%s\n',messageDefinition{:});
fclose(fileID);
```

Use the `gazebogenmsg` function to generate dependences in the created folder.

```
gazebogenmsg(folderPath)
```

```
Validating ...
Selected compiler details: "Microsoft Visual C++ 2019 16.0"
[libprotobuf WARNING] No syntax specified for the proto file: MyPose.proto. Please use 'syntax =
Building shared library ...
Microsoft (R) C/C++ Optimizing Compiler Version 19.15.26726 for x64
Copyright (C) Microsoft Corporation.  All rights reserved.

MyPose.pb.cc
Microsoft (R) Incremental Linker Version 14.15.26726.0
Copyright (C) Microsoft Corporation.  All rights reserved.

/out:MyPose.pb.dll
/dll
/implib:MyPose.pb.lib
/LIBPATH:B:\matlab\toolbox\shared\robotics\externalDependency\libprotobuf\lib
libprotobuf3.lib
/OUT:C:\TEMP\Bdoc23a_2213998_3568\ib570499\13\tp73e6043f\robotics-ex62907275\customMessage\insta
/IMPLIB:C:\TEMP\Bdoc23a_2213998_3568\ib570499\13\tp73e6043f\robotics-ex62907275\customMessage\ins
C:\TEMP\Bdoc23a_2213998_3568\ib570499\13\tp73e6043f\robotics-ex62907275\customMessage\install\MyP
   Creating library C:\TEMP\Bdoc23a_2213998_3568\ib570499\13\tp73e6043f\robotics-ex62907275\custo
Building MEX for "MyPose.proto" file ...
Building with 'Microsoft Visual C++ 2019'.
MEX completed successfully.
Building with 'Microsoft Visual C++ 2019'.
MEX completed successfully.
Building custom message utilities  ...
DONE.

To use the gazebo custom messages, execute following commands:

addpath('C:\TEMP\Bdoc23a_2213998_3568\ib570499\13\tp73e6043f\robotics-ex62907275\customMessage\in
savepath
```

Use the following commands to add and save the install path.

```
addpath(fullfile(folderPath,'install'))
```

```
savepath
```

Create a Gazebo plugin package `'MyPlugin'` inside the custom message folder using the `packageGazeboPlugin` function.

```
packageGazeboPlugin(fullfile(folderPath,'MyPlugin'),folderPath)
```

Generate Dependencies for Built-in Gazebo Message

Create a folder in a local directory.

```
folderPath = fullfile(pwd,'customMessage');
mkdir(folderPath)
cd(folderPath)
```

Use the `gazebogenmsg` function to generate dependencies for a built-in gazebo message in the specified folder.

```
gazebogenmsg(folderPath,"GazeboMessageList","gazebo.msgs.Image");
```

```
Validating ...
Selected compiler details: "Microsoft Visual C++ 2019 16.0"
Building shared library ...
Microsoft (R) C/C++ Optimizing Compiler Version 19.15.26726 for x64
Copyright (C) Microsoft Corporation.  All rights reserved.

image.pb.cc
Microsoft (R) Incremental Linker Version 14.15.26726.0
Copyright (C) Microsoft Corporation.  All rights reserved.

/out:image.pb.dll
/dll
/implib:image.pb.lib
/LIBPATH:B:\matlab\toolbox\shared\robotics\externalDependency\libprotobuf\lib
libprotobuf3.lib
/OUT:C:\TEMP\Bdoc23a_2213998_3568\ib570499\13\tp73e6043f\robotics-ex40128733\customMessage\insta
/IMPLIB:C:\TEMP\Bdoc23a_2213998_3568\ib570499\13\tp73e6043f\robotics-ex40128733\customMessage\ins
C:\TEMP\Bdoc23a_2213998_3568\ib570499\13\tp73e6043f\robotics-ex40128733\customMessage\install\ima
   Creating library C:\TEMP\Bdoc23a_2213998_3568\ib570499\13\tp73e6043f\robotics-ex40128733\custo
Building MEX for "image.proto" file ...
Building with 'Microsoft Visual C++ 2019'.
MEX completed successfully.
Building with 'Microsoft Visual C++ 2019'.
MEX completed successfully.
Building custom message utilities  ...
DONE.

To use the gazebo custom messages, execute following commands:

addpath('C:\TEMP\Bdoc23a_2213998_3568\ib570499\13\tp73e6043f\robotics-ex40128733\customMessage\in
savepath
```

Use the following commands to add and save the install path.

```
addpath(fullfile(folderPath,'install'))
```

```
savepath
```

Create a Gazebo plugin package using the `packageGazeboPlugin` function.

```
packageGazeboPlugin
```

Input Arguments

`packagePath` — Name or path of Gazebo plugin package folder
string scalar | character vector

Name or path of the Gazebo plugin package folder, specified as a string scalar or a character vector.

When specified as a folder name, the function creates a plugin folder and a compressed plugin file with the specified name in the current directory.

Example: `packageGazeboPlugin('MyPlugin')`

When specified as a file path, the function creates a plugin folder and a compressed plugin file with the specified file name in the specified folder.

Example: `packageGazeboPlugin('C:\GazeboPlugin\MyPlugin')`

Data Types: `char` | `string`

`customMessagePath` — Path of Gazebo custom message folder
string scalar | character vector

Path of the Gazebo custom message folder, specified as a string scalar or a character vector.

To create a Gazebo plugin with custom message support, specify the `customMessagePath` as a valid path to the folder that contains the desired custom message dependencies.

When the `packagePath` argument is specified as a folder name, the function creates a plugin folder and a compressed plugin file with the specified package name in the current directory.

Example: `packageGazeboPlugin('MyPlugin','C:\GazeboCustomMsg')`

When the `packagePath` argument is specified as a file path inside the custom message folder, the function creates a plugin folder and a compressed plugin file with the specified file name in the specified folder.

Example: `packageGazeboPlugin('C:\GazeboCustomMsg\MyPlugin','C:\GazeboCustomMsg')`

Data Types: `char` | `string`

Output Arguments

`outputPath` — Path of plugin folder
character vector

Path of the plugin folder, returned as a character vector.

Limitations

- `packageGazeboPlugin` function not supported with MATLAB Compiler.

Version History

Introduced in R2020b

See Also

`gazebogenmsg`

Topics

"Perform Co-Simulation between Simulink and Gazebo"

parts

Extract quaternion parts

Syntax

```
[a,b,c,d] = parts(quat)
```

Description

`[a,b,c,d] = parts(quat)` returns the parts of the quaternion array as arrays, each the same size as `quat`.

Examples

Convert Quaternion to Matrix of Quaternion Parts

Convert a quaternion representation to parts using the `parts` function.

Create a two-element column vector of quaternions by specifying the parts.

```
quat = quaternion([1:4;5:8])
```

```
quat = 2x1 quaternion array
     1 + 2i + 3j + 4k
     5 + 6i + 7j + 8k
```

Recover the parts from the quaternion matrix using the `parts` function. The parts are returned as separate output arguments, each the same size as the input 2-by-1 column vector of quaternions.

```
[qA,qB,qC,qD] = parts(quat)
```

```
qA = 2×1

     1
     5


qB = 2×1

     2
     6


qC = 2×1

     3
     7


qD = 2×1
```

```
     4
     8
```

Input Arguments

quat — Quaternion
scalar | vector | matrix | multidimensional array

Quaternion, specified as a quaternion or array of quaternions.

Data Types: `quaternion`

Output Arguments

[a,b,c,d] — Quaternion parts
scalar | vector | matrix | multidimensional array

Quaternion parts, returned as four arrays: a, b, c, and d. Each part is the same size as `quat`.

Data Types: `single` | `double`

Version History

Introduced in R2018a

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

Functions
`classUnderlying` | `compact`

Objects
`quaternion`

plotTransforms

Plot 3-D transforms from translations and rotations

Syntax

```
ax = plotTransforms(translations,rotations)
ax = plotTransforms(transformations)
ax = plotTransforms( ___ ,Name,Value)
```

Description

`ax = plotTransforms(translations,rotations)` draws transform frames in a 3-D figure window using the specified translations `translations`, and rotations, `rotations`. The *z*-axis always points upward.

`ax = plotTransforms(transformations)` draws transform frames for the specified SE(2) or SE(3) transformations, `transformations`.

`ax = plotTransforms( ___ ,Name,Value)` specifies additional options using name-value arguments. Specify multiple name-value arguments to set multiple options.

Input Arguments

`translations` — *xyz*-positions
`[x y z]` vector | matrix of `[x y z]` vectors

xyz-positions specified as a vector or matrix of `[x y z]` vectors. Each row represents a new frame to plot with a corresponding orientation in `rotations`.

Example: `[1 1 1; 2 2 2]`

`rotations` — Rotations of *xyz*-positions
`quaternion` array | matrix of `[w x y z]` quaternion vectors | *N*-element array of `so2` or `so3` objects

Rotations of *xyz*-positions specified as a `quaternion` array, *N*-by-4 matrix of `[w x y z]` quaternion vectors, or an *N*-element array of `so2` or `so3` objects. *N* is the total number of rotations, and each element of the array, each row of the matrix or rotation transformation objects represent the rotation of the *xyz*-positions specified in `translations`.

If `rotations` is an *N*-element array of `so2` or `so3` objects, each element must be of the same type.

Example: `[1 1 1 0; 1 3 5 0]`

`transformations` — Transformation
`se2` object | `se3` object | *M*-element array of `se2` or `se3` objects

Transformations, specified as an `se2` object, an `se3` object, or an *M*-element array of `se2` or `se3` objects. *M* is the total number of transformations.

If you specify `transformations` as an array, each element must be of the same type.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `'FrameSize',5`

`FrameSize` — Size of frames and attached meshes

positive numeric scalar

Size of frame and attached meshes, specified as positive numeric scalar.

`FrameColor` — Color of frames

`"rgb"` (default) | RGB triplet | string scalar

Color of frames, specified as an RGB triplet or string scalar.

Example: `[0 0 1]` or `"green"`

`FrameAxisLabels` — *xyz* labels of coordinate frame

`"off"` (default) | `"on"`

xyz labels of the coordinate frame, specified as `"off"` to hide the labels or `"on"` to show the labels.

`FrameAxisLabels` — Frame axis labels

`""` (default) | `string` | *N*-element array of strings

Frame axis labels, specified as a string or *N*-element array of strings, where *N* is the total number of frames and each string corresponds to one frame at the same index of `transformations`, `translations`, or `rotations`.

`AxisLabels` — *xyz* labels of plotting axes

`"off"` (default) | `"on"`

xyz labels of the plotting axes, specified as `"off"` to hide the labels or `"on"`to show the labels.

`InertialZDirection` — Direction of positive *z*-axis of inertial frame

`"up"` (default) | `"down"`

Direction of the positive *z*-axis of inertial frame, specified as either `"up"` or `"down"`. In the plot, the positive *z*-axis always points up.

`MeshFilePath` — File path of mesh file attached to frames

character vector | string scalar

File path of mesh file attached to frames, specified as either a character vector or string scalar. The mesh is attached to each plotted frame at the specified position and orientation. Provided `.stl` are

- `"fixedwing.stl"`
- `"multirotor.stl"`
- `"groundvehicle.stl"`

Example: `'fixedwing.stl'`

MeshColor — Color of attached mesh
"red" (default) | RGB triplet | string scalar

Color of attached mesh, specified as an RGB triplet or string scalar.

Example: [0 0 1] or "green"

View — Plot view
"3D" (default) | "2D" | three-element vector

Plot view, specified as "3D", "2D", or a three-element vector of the form [*x*,*y*,*z*] that sets the view angle in Cartesian coordinates. The magnitude of *x*,*y*, and *z* are ignored.

Parent — Axes used to plot transforms
Axes object | UIAxes object

Axes used to plot the pose graph, specified as the comma-separated pair consisting of 'Parent' and either an Axes or UIAxes object. See axes or uiaxes.

Output Arguments

ax — Axes used to plot transforms
Axes object | UIAxes object

Axes used to plot the pose graph, specified as the comma-separated pair consisting of 'Parent' and either an Axes or UIAxesobject. See axes or uiaxes.

Version History

Introduced in R2018b

See Also

Functions
quaternion | hom2cart | eul2quat | tform2quat | rotm2quat

Objects
se2 | se3 | so2 | so3

power, .^

Element-wise quaternion power

Syntax

```
C = A.^b
```

Description

`C = A.^b` raises each element of `A` to the corresponding power in `b`.

Examples

Raise a Quaternion to a Real Scalar Power

Create a quaternion and raise it to a real scalar power.

```
A = quaternion(1,2,3,4)
```

```
A = quaternion
     1 + 2i + 3j + 4k
```

```
b = 3;
C = A.^b
```

```
C = quaternion
     -86 -  52i -  78j - 104k
```

Raise a Quaternion Array to Powers from a Multidimensional Array

Create a 2-by-1 quaternion array and raise it to powers from a 2-D array.

```
A = quaternion([1:4;5:8])
```

```
A = 2x1 quaternion array
     1 + 2i + 3j + 4k
     5 + 6i + 7j + 8k
```

```
b = [1 0 2; 3 2 1]
```

```
b = 2×3

     1     0     2
     3     2     1
```

```
C = A.^b
```

```
C = 2x3 quaternion array
       1 +    2i +    3j +    4k         1 +    0i +    0j +    0k       -28 +    4i +    6j +
   -2110 -  444i -  518j -  592k      -124 +   60i +   70j +   80k         5 +    6i +    7j +
```

Input Arguments

A — Base

scalar | vector | matrix | multidimensional array

Base, specified as a scalar, vector, matrix, or multidimensional array.

Data Types: `quaternion` | `single` | `double`

b — Exponent

scalar | vector | matrix | multidimensional array

Exponent, specified as a real scalar, vector, matrix, or multidimensional array.

Data Types: `single` | `double`

Output Arguments

C — Result

scalar | vector | matrix | multidimensional array

Each element of quaternion A raised to the corresponding power in b, returned as a scalar, vector, matrix, or multidimensional array.

Data Types: `quaternion`

Algorithms

The polar representation of a quaternion $A = a + b\mathrm{i} + c\mathrm{j} + d\mathrm{k}$ is given by

$$A = \|A\|(\cos\theta + \widehat{u}\sin\theta)$$

where θ is the angle of rotation, and $\hat{u}$ is the unit quaternion.

Quaternion A raised by a real exponent b is given by

$$P = A\,.\,\hat{}\,b = \|A\|^b(\cos(b\theta) + \widehat{u}\sin(b\theta))$$

Version History

Introduced in R2018b

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

Functions
`log` | `exp`

Objects
`quaternion`

prod

Product of a quaternion array

Syntax

```
quatProd = prod(quat)
quatProd = prod(quat,dim)
```

Description

`quatProd = prod(quat)` returns the quaternion product of the elements of the array.

`quatProd = prod(quat,dim)` calculates the quaternion product along dimension `dim`.

Examples

Product of Quaternions in Each Column

Create a 3-by-3 array whose elements correspond to their linear indices.

```
A = reshape(quaternion(randn(9,4)),3,3)
```

```
A = 3x3 quaternion array
      0.53767 +   2.7694i +    1.409j -  0.30344k      0.86217 +   0.7254i -   1.2075j +   0.888
       1.8339 -   1.3499i +   1.4172j +  0.29387k      0.31877 - 0.063055i +  0.71724j -   1.147
      -2.2588 +   3.0349i +   0.6715j -  0.78728k      -1.3077 +  0.71474i +   1.6302j -   1.068
```

Find the product of the quaternions in each column. The length of the first dimension is `1`, and the length of the second dimension matches `size(A,2)`.

```
B = prod(A)
```

```
B = 1x3 quaternion array
     -19.837 -  9.1521i +  15.813j -  19.918k     -5.4708 - 0.28535i +   3.077j -  1.2295k
```

Product of Specified Dimension of Quaternion Array

You can specify which dimension of a quaternion array to take the product of.

Create a 2-by-2-by-2 quaternion array.

```
A = reshape(quaternion(randn(8,4)),2,2,2);
```

Find the product of the elements in each page of the array. The length of the first dimension matches `size(A,1)`, the length of the second dimension matches `size(A,2)`, and the length of the third dimension is `1`.

```
dim = 3;
B = prod(A,dim)

B = 2x2 quaternion array
     -2.4847 +  1.1659i - 0.37547j +  2.8068k     0.28786 - 0.29876i - 0.51231j -  4.2972k
     0.38986 -  3.6606i -  2.0474j -   6.047k      -1.741 - 0.26782i +  5.4346j +  4.1452k
```

Input Arguments

quat — Quaternion
scalar | vector | matrix | multidimensional array

Quaternion, specified as scalar, vector, matrix, or multidimensional array of quaternions.

Example: `qProd = prod(quat)` calculates the quaternion product along the first non-singleton dimension of `quat`.

Data Types: `quaternion`

dim — Dimension
first non-singleton dimension (default) | positive integer

Dimension along which to calculate the quaternion product, specified as a positive integer. If `dim` is not specified, `prod` operates along the first non-singleton dimension of `quat`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Output Arguments

quatProd — Quaternion product
positive integer

Quaternion product, returned as quaternion array with one less non-singleton dimension than `quat`.

For example, if `quat` is a 2-by-2-by-5 array,

- `prod(quat,1)` returns a 1-by-2-by-5 array.
- `prod(quat,2)` returns a 2-by-1-by-5 array.
- `prod(quat,3)` returns a 2-by-2 array.

Data Types: `quaternion`

Version History

Introduced in R2018a

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

Functions
mtimes | .*,times

Objects
quaternion

quat2axang

Convert quaternion to axis-angle rotation

Syntax

```
axang = quat2axang(quat)
```

Description

`axang = quat2axang(quat)` converts a quaternion, `quat`, to the equivalent axis-angle rotation, `axang`.

Examples

Convert Quaternion to Axis-Angle Rotation

```
quat = [0.7071 0.7071 0 0];
axang = quat2axang(quat)
```

```
axang = 1×4

    1.0000         0         0    1.5708
```

Input Arguments

quat — Unit quaternion
n-by-4 matrix | n-element vector of `quaternion` objects

Unit quaternion, specified as an *n*-by-4 matrix or n-element vector of `quaternion` objects containing *n* quaternions. If the input is a matrix, each row is a quaternion vector of the form $q = [w\ x\ y\ z]$, with *w* as the scalar number.

Example: `[0.7071 0.7071 0 0]`

Output Arguments

axang — Rotation given in axis-angle form
n-by-4 matrix

Rotation given in axis-angle form, returned as an *n*-by-4 matrix of *n* axis-angle rotations. The first three elements of every row specify the rotation axis, and the last element defines the rotation angle (in radians).

Example: `[1 0 0 pi/2]`

Version History

Introduced in R2015a

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`axang2quat` | `quaternion`

Topics

"Coordinate Transformations in Robotics"

quat2eul

Convert quaternion to Euler angles

Syntax

```
eul = quat2eul(quat)
eul = quat2eul(quat,sequence)
[eul,eulAlt] = quat2eul( ___ )
```

Description

`eul = quat2eul(quat)` converts a quaternion rotation, `quat`, to the corresponding Euler angles, `eul`. The default order for Euler angle rotations is `"ZYX"`.

`eul = quat2eul(quat,sequence)` converts a quaternion into Euler angles. The Euler angles are specified in the axis rotation sequence, `sequence`. The default order for Euler angle rotations is `"ZYX"`.

`[eul,eulAlt] = quat2eul( ___ )` also returns an alternate set of Euler angles that represents the same rotation `eulAlt`.

Examples

Convert Quaternion to Euler Angles

```
quat = [0.7071 0.7071 0 0];
eulZYX = quat2eul(quat)
```

```
eulZYX = 1×3

         0         0    1.5708
```

Convert Quaternion to Euler Angles Using ZYZ Axis Order

```
quat = [0.7071 0.7071 0 0];
eulZYZ = quat2eul(quat,'ZYZ')
```

```
eulZYZ = 1×3

    1.5708   -1.5708   -1.5708
```

Input Arguments

quat — Unit quaternion
n-by-4 matrix | n-element vector of `quaternion` objects

Unit quaternion, specified as an *n*-by-4 matrix or n-element vector of `quaternion` objects containing *n* quaternions. If the input is a matrix, each row is a quaternion vector of the form $q = [w\ x\ y\ z]$, with *w* as the scalar number.

Example: `[0.7071 0.7071 0 0]`

`sequence` — Axis rotation sequence
`"ZYX"` (default) | `"ZYZ"` | `"XYZ"`

Axis rotation sequence for the Euler angles, specified as one of these string scalars:

- `"ZYX"` (default) – The order of rotation angles is *z*-axis, *y*-axis, *x*-axis.
- `"ZYZ"` – The order of rotation angles is *z*-axis, *y*-axis, *z*-axis.
- `"XYZ"` – The order of rotation angles is *x*-axis, *y*-axis, *z*-axis.

Data Types: `string` | `char`

Output Arguments

`eul` — Euler rotation angles
n-by-3 matrix

Euler rotation angles in radians, returned as an *n*-by-3 array of Euler rotation angles. Each row represents one Euler angle set.

Example: `[0 0 1.5708]`

`eulAlt` — Alternate Euler rotation angle solution
n-by-3 matrix

Alternate Euler rotation angle solution in radians, returned as an *n*-by-3 array of Euler rotation angles. Each row represents one Euler angle set.

Example: `[0 0 1.5708]`

Version History

Introduced in R2015a

R2020a: Alternate Euler angle output

`quat2eul` now optionally outputs an alternate set of Euler angles that also represent the same rotation as the original output Euler angles.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also
`eul2quat` | `quaternion`

Topics

"Coordinate Transformations in Robotics"

quat2rotm

Convert quaternion to rotation matrix

Syntax

```
rotm = quat2rotm(quat)
```

Description

`rotm = quat2rotm(quat)` converts a quaternion `quat` to an orthonormal rotation matrix, `rotm`. When using the rotation matrix, premultiply it with the coordinates to be rotated (as opposed to postmultiplying).

Examples

Convert Quaternion to Rotation Matrix

```
quat = [0.7071 0.7071 0 0];
rotm = quat2rotm(quat)
```

```
rotm = 3×3

    1.0000         0         0
         0   -0.0000   -1.0000
         0    1.0000   -0.0000
```

Input Arguments

`quat` — Unit quaternion
n-by-4 matrix | n-element vector of `quaternion` objects

Unit quaternion, specified as an *n*-by-4 matrix or n-element vector of `quaternion` objects containing *n* quaternions. If the input is a matrix, each row is a quaternion vector of the form $q = [w\ x\ y\ z]$, with *w* as the scalar number.

Example: `[0.7071 0.7071 0 0]`

Output Arguments

`rotm` — Rotation matrix
3-by-3-by-*n* matrix

Rotation matrix, returned as a 3-by-3-by-*n* matrix containing *n* rotation matrices. Each rotation matrix has a size of 3-by-3 and is orthonormal. When using the rotation matrix, premultiply it with the coordinates to be rotated (as opposed to postmultiplying).

Example: `[0 0 1; 0 1 0; -1 0 0]`

Version History

Introduced in R2015a

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`rotm2quat` | `quaternion` | `so2` | `so3`

Topics

"Coordinate Transformations in Robotics"

quat2tform

Convert quaternion to homogeneous transformation

Syntax

```
tform = quat2tform(quat)
```

Description

`tform = quat2tform(quat)` converts a quaternion, `quat`, to a homogeneous transformation matrix, `tform`. When using the transformation matrix, premultiply it with the coordinates to be transformed (as opposed to postmultiplying).

Examples

Convert Quaternion to Homogeneous Transformation

```
quat = [0.7071 0.7071 0 0];
tform = quat2tform(quat)
```

```
tform = 4×4

    1.0000         0         0         0
         0   -0.0000   -1.0000         0
         0    1.0000   -0.0000         0
         0         0         0    1.0000
```

Input Arguments

`quat` — Unit quaternion
n-by-4 matrix | n-element vector of `quaternion` objects

Unit quaternion, specified as an *n*-by-4 matrix or n-element vector of objects containing *n* quaternions. If the input is a matrix, each row is a quaternion vector of the form $q = [w\ x\ y\ z]$, with *w* as the scalar number.

Example: `[0.7071 0.7071 0 0]`

Output Arguments

`tform` — Homogeneous transformation
4-by-4-by-*n* matrix

Homogeneous transformation matrix, returned as a 4-by-4-by-*n* matrix of *n* homogeneous transformations. When using the rotation matrix, premultiply it with the coordinates to be rotated (as opposed to postmultiplying).

Example: `[0 0 1 0; 0 1 0 0; -1 0 0 0; 0 0 0 1]`

Version History

Introduced in R2015a

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`tform2quat` | `quaternion` | `se2` | `se3`

Topics

“Coordinate Transformations in Robotics”

rdivide, ./

Element-wise quaternion right division

Syntax

```
C = A./B
```

Description

`C = A./B` performs quaternion element-wise division by dividing each element of quaternion `A` by the corresponding element of quaternion `B`.

Examples

Divide a Quaternion Array by a Real Scalar

Create a 2-by-1 quaternion array, and divide it element-by-element by a real scalar.

```
A = quaternion([1:4;5:8])
```

```
A = 2x1 quaternion array
     1 + 2i + 3j + 4k
     5 + 6i + 7j + 8k
```

```
B = 2;
C = A./B
```

```
C = 2x1 quaternion array
     0.5 +   1i + 1.5j +   2k
     2.5 +   3i + 3.5j +   4k
```

Divide a Quaternion Array by Another Quaternion Array

Create a 2-by-2 quaternion array, and divide it element-by-element by another 2-by-2 quaternion array.

```
q1 = quaternion(magic(4));
A = reshape(q1,2,2)
```

```
A = 2x2 quaternion array
     16 +  2i +  3j + 13k      9 +  7i +  6j + 12k
      5 + 11i + 10j +  8k      4 + 14i + 15j +  1k
```

```
q2 = quaternion([1:4;3:6;2:5;4:7]);
B = reshape(q2,2,2)
```

```
B = 2x2 quaternion array
     1 + 2i + 3j + 4k     2 + 3i + 4j + 5k
     3 + 4i + 5j + 6k     4 + 5i + 6j + 7k
```

```
C = A./B
```

```
C = 2x2 quaternion array
          2.7 -      0.1i -      2.1j -      1.7k       2.2778 + 0.092593i -  0.46296j -  0.5740
       1.8256 - 0.081395i +  0.45349j -  0.24419k       1.4524 -      0.5i +   1.0238j -   0.261
```

Input Arguments

A — Dividend
scalar | vector | matrix | multidimensional array

Dividend, specified as a quaternion, an array of quaternions, a real scalar, or an array of real numbers.

A and B must have compatible sizes. In the simplest cases, they can be the same size or one can be a scalar. Two inputs have compatible sizes if, for every dimension, the dimension sizes of the inputs are the same or one of the dimensions is 1.

Data Types: quaternion | single | double

B — Divisor
scalar | vector | matrix | multidimensional array

Divisor, specified as a quaternion, an array of quaternions, a real scalar, or an array of real numbers.

A and B must have compatible sizes. In the simplest cases, they can be the same size or one can be a scalar. Two inputs have compatible sizes if, for every dimension, the dimension sizes of the inputs are the same or one of the dimensions is 1.

Data Types: quaternion | single | double

Output Arguments

C — Result
scalar | vector | matrix | multidimensional array

Result of quaternion division, returned as a scalar, vector, matrix, or multidimensional array.

Data Types: quaternion

Algorithms

Quaternion Division

Given a quaternion $A = a_1 + a_2\mathrm{i} + a_3\mathrm{j} + a_4\mathrm{k}$ and a real scalar p,

$$C = A./p = \frac{a_1}{p} + \frac{a_2}{p}\mathrm{i} + \frac{a_3}{p}\mathrm{j} + \frac{a_4}{p}\mathrm{k}$$

Note For a real scalar p, $A./p = A.\backslash p$.

Quaternion Division by a Quaternion Scalar

Given two quaternions A and B of compatible sizes,

$$C = A./B = A.*B^{-1} = A.*\left(\frac{conj(B)}{norm(B)^2}\right)$$

Version History

Introduced in R2018b

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

Functions

`conj` | `./,ldivide` | `norm` | `.*,times`

Objects

`quaternion`

quinticpolytraj

Generate fifth-order trajectories

Syntax

```
[q,qd,qdd,pp] = quinticpolytraj(wayPoints,timePoints,tSamples)
[q,qd,qdd,pp] = quinticpolytraj( ___ ,Name,Value)
```

Description

`[q,qd,qdd,pp] = quinticpolytraj(wayPoints,timePoints,tSamples)` generates a fifth-order polynomial that achieves a given set of input waypoints with corresponding time points. The function outputs positions, velocities, and accelerations at the given time samples, `tSamples`. The function also returns the piecewise polynomial `pp` form of the polynomial trajectory with respect to time.

`[q,qd,qdd,pp] = quinticpolytraj( ___ ,Name,Value)` specifies additional parameters as `Name,Value` pair arguments using any combination of the previous syntaxes.

Examples

Compute Quintic Trajectory for 2-D Planar Motion

Use the `quinticpolytraj` function with a given set of 2-D *xy* waypoints. Time points for the waypoints are also given.

```
wpts = [1 4 4 3 -2 0; 0 1 2 4 3 1];
tpts = 0:5;
```

Specify a time vector for sampling the trajectory. Sample at a smaller interval than the specified time points.

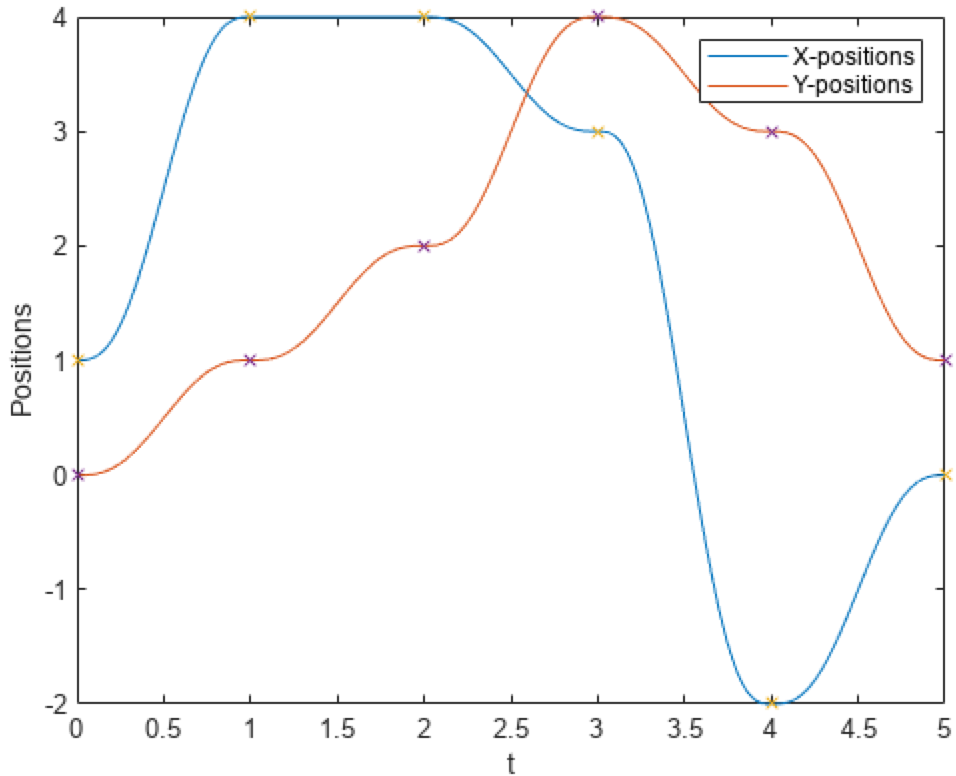
```
tvec = 0:0.01:5;
```

Compute the quintic trajectory. The function outputs the trajectory positions (`q`), velocity (`qd`), acceleration (`qdd`), and polynomial coefficients (`pp`) of the quintic polynomial.

```
[q, qd, qdd, pp] = quinticpolytraj(wpts, tpts, tvec);
```

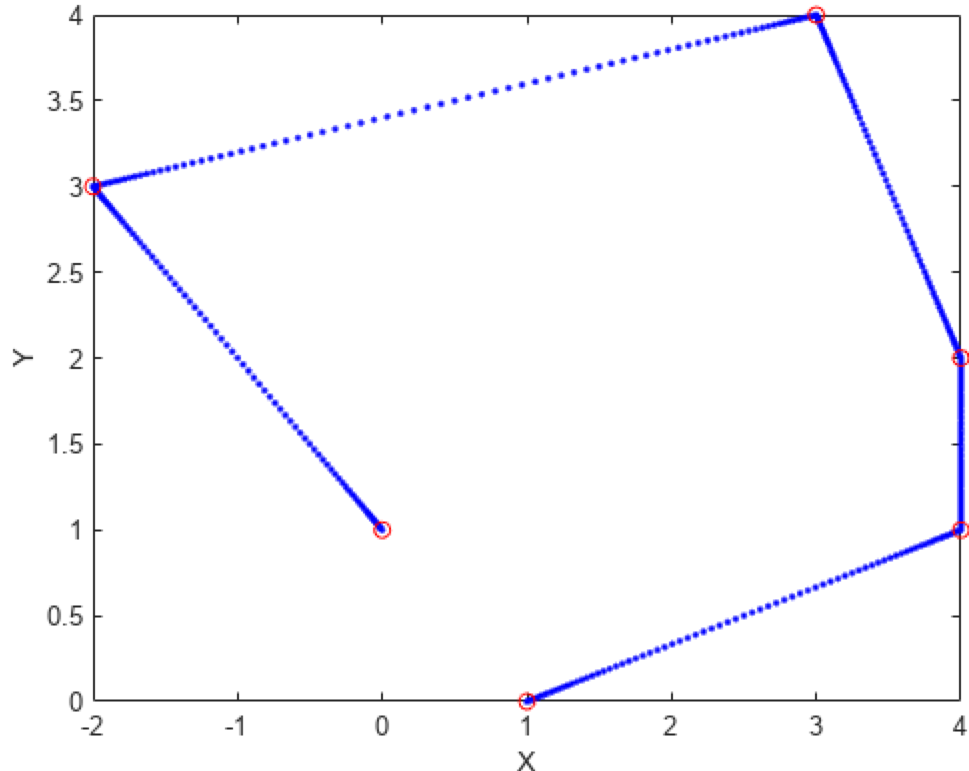
Plot the quintic trajectories for the *x*- and *y*-positions. Compare the trajectory with each waypoint.

```
plot(tvec, q)
hold all
plot(tpts, wpts, 'x')
xlabel('t')
ylabel('Positions')
legend('X-positions','Y-positions')
hold off
```



You can also verify the actual positions in the 2-D plane. Plot the separate rows of the `q` vector and the waypoints as *x*- and *y*- positions.

```
figure
plot(q(1,:),q(2,:),'.b',wpts(1,:),wpts(2,:),'or')
xlabel('X')
ylabel('Y')
```



Input Arguments

`wayPoints` — Waypoints for trajectory
n-by-*p* matrix

Points for waypoints of trajectory, specified as an *n*-by-*p* matrix, where *n* is the dimension of the trajectory and *p* is the number of waypoints.

Example: `[1 4 4 3 -2 0; 0 1 2 4 3 1]`

Data Types: `single` | `double`

`timePoints` — Time points for waypoints of trajectory
p-element vector

Time points for waypoints of trajectory, specified as a *p*-element vector.

Example: `[0 2 4 5 8 10]`

Data Types: `single` | `double`

`tSamples` — Time samples for trajectory
m-element vector

Time samples for the trajectory, specified as an *m*-element vector. The output position, `q`, velocity, `qd`, and accelerations, `qdd`, are sampled at these time intervals.

Example: `0:0.01:10`

Data Types: `single` | `double`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `'VelocityBoundaryCondition',[1 0 -1 -1 0 0; 1 1 1 -1 -1 -1]`

`VelocityBoundaryCondition` — Velocity boundary conditions for each waypoint
`zeroes(n,p)` (default) | *n*-by-*p* matrix

Velocity boundary conditions for each waypoint, specified as the comma-separated pair consisting of `'VelocityBoundaryCondition'` and an *n*-by-*p* matrix. Each row corresponds to the velocity at all of *p* waypoints for the respective variable in the trajectory.

Example: `[1 0 -1 -1 0 0; 1 1 1 -1 -1 -1]`

Data Types: `single` | `double`

`AccelerationBoundaryCondition` — Acceleration boundary conditions for each waypoint
`zeroes(n,p)` (default) | *n*-by-*p* matrix

Acceleration boundary conditions for each waypoint, specified as the comma-separated pair consisting of `'AccelerationBoundaryCondition'` and an *n*-by-*p* matrix. Each row corresponds to the acceleration at all of *p* waypoints for the respective variable in the trajectory.

Example: `[1 0 -1 -1 0 0; 1 1 1 -1 -1 -1]`

Data Types: `single` | `double`

Output Arguments

q — Positions of trajectory
m-element vector

Positions of the trajectory at the given time samples in `tSamples`, returned as an *m*-element vector, where *m* is the length of `tSamples`.

Data Types: `single` | `double`

qd — Velocities of trajectory
vector

Velocities of the trajectory at the given time samples in `tSamples`, returned as a vector.

Data Types: `single` | `double`

qdd — Accelerations of trajectory
vector

Accelerations of the trajectory at the given time samples in `tSamples`, returned as a vector.

Data Types: `single` | `double`

pp — Piecewise-polynomial
structure

Piecewise-polynomial, returned as a structure that defines the polynomial for each section of the piecewise trajectory. You can build your own piecewise polynomials using `mkpp`, or evaluate the polynomial at specified times using `ppval`. The structure contains the fields:

- `form`: `'pp'`.
- `breaks`: *p*-element vector of times when the piecewise trajectory changes forms. *p* is the number of waypoints.
- `coefs`: *n*(*p*–1)-by-`order` matrix for the coefficients for the polynomials. *n*(*p*–1) is the dimension of the trajectory times the number of `pieces`. Each set of *n* rows defines the coefficients for the polynomial that described each variable trajectory.
- `pieces`: *p*–1. The number of breaks minus 1.
- `order`: Degree of the polynomial + 1. For example, cubic polynomials have an order of 4.
- `dim`: *n*. The dimension of the control point positions.

Version History
Introduced in R2019a

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also
`bsplinepolytraj` | `contopptraj` | `cubicpolytraj` | `rottraj` | `transformtraj` | `trapveltraj`

randrot

Uniformly distributed random rotations

Syntax

```
R = randrot
R = randrot(m)
R = randrot(m1,...,mN)
R = randrot([m1,...,mN])
```

Description

`R = randrot` returns a unit quaternion drawn from a uniform distribution of random rotations.

`R = randrot(m)` returns an `m`-by-`m` matrix of unit quaternions drawn from a uniform distribution of random rotations.

`R = randrot(m1,...,mN)` returns an `m1`-by-...-by-`mN` array of random unit quaternions, where `m1`, ..., `mN` indicate the size of each dimension. For example, `randrot(3,4)` returns a 3-by-4 matrix of random unit quaternions.

`R = randrot([m1,...,mN])` returns an `m1`-by-...-by-`mN` array of random unit quaternions, where `m1`,..., `mN` indicate the size of each dimension. For example, `randrot([3,4])` returns a 3-by-4 matrix of random unit quaternions.

Examples

Matrix of Random Rotations

Generate a 3-by-3 matrix of uniformly distributed random rotations.

```
r = randrot(3)
```

```
r = 3x3 quaternion array
      0.17446 +  0.59506i -  0.73295j +  0.27976k      0.69704 - 0.060589i +  0.68679j -  0.19695
      0.21908 -  0.89875i -    0.298j +  0.23548k    -0.049744 +  0.59691i +  0.56459j +  0.5678
       0.6375 +  0.49338i -  0.24049j +  0.54068k       0.2979 -  0.53568i +  0.31819j +  0.7232
```

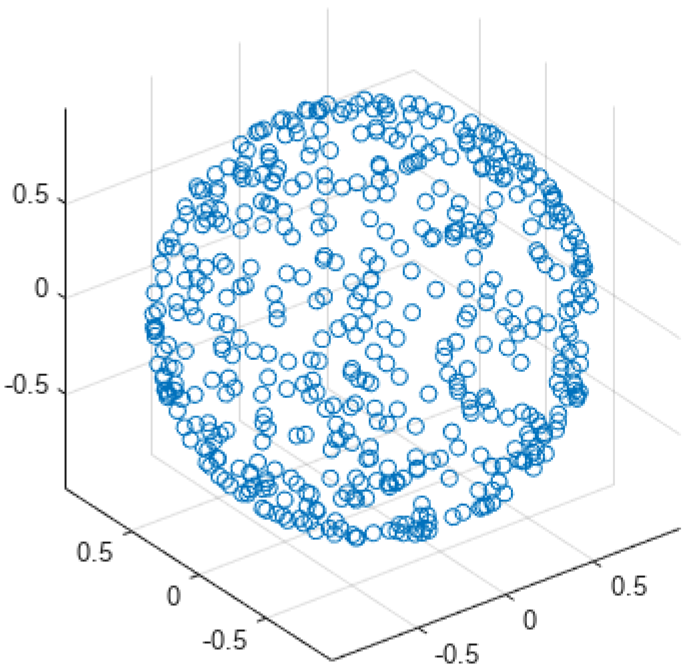
Create Uniform Distribution of Random Rotations

Create a vector of 500 random quaternions. Use `rotatepoint` to visualize the distribution of the random rotations applied to point (1, 0, 0).

```
q = randrot(500,1);
```

```
pt = rotatepoint(q, [1 0 0]);
```

```
figure
scatter3(pt(:,1), pt(:,2), pt(:,3))
axis equal
```



Input Arguments

m — Size of square matrix
integer

Size of square quaternion matrix, specified as an integer value. If m is 0 or negative, then R is returned as an empty matrix.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

m1,...,mN — Size of each dimension
two or more integer values

Size of each dimension, specified as two or more integer values. If the size of any dimension is 0 or negative, then R is returned as an empty array.

Example: randrot(2,3) returns a 2-by-3 matrix of random quaternions.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

[m1,...,mN] — Vector of size of each dimension
row vector of integer values

Vector of size of each dimension, specified as a row vector of two or more integer values. If the size of any dimension is `0` or negative, then `R` is returned as an empty array.

Example: `randrot([2,3])` returns a 2-by-3 matrix of random quaternions.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Output Arguments

R — Random quaternions
scalar | vector | matrix | multidimensional array

Random quaternions, returned as a quaternion or array of quaternions.

Data Types: `quaternion`

Version History

Introduced in R2019a

References

[1] Shoemake, K. "Uniform Random Rotations." *Graphics Gems III* (K. David, ed.). New York: Academic Press, 1992.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

`quaternion`

readBinaryOccupancyGrid

Read binary occupancy grid

Syntax

```
map = readBinaryOccupancyGrid(msg)
map = readBinaryOccupancyGrid(msg,thresh)
map = readBinaryOccupancyGrid(msg,thresh,val)
```

Description

`map = readBinaryOccupancyGrid(msg)` returns a `binaryOccupancyMap` object by reading the data inside a ROS message, `msg`, which must be a `'nav_msgs/OccupancyGrid'` message. All message data values greater than or equal to the occupancy threshold are set to occupied, `1`, in the map. All other values, including unknown values (`-1`) are set to unoccupied, `0`, in the map.

Note The `msg` input is an `'nav_msgs/OccupancyGrid'` ROS message. For more info, see `OccupancyGrid`.

`map = readBinaryOccupancyGrid(msg,thresh)` specifies a threshold, `thresh`, for occupied values. All values greater than or equal to the threshold are set to occupied, `1`. All other values are set to unoccupied, `0`.

`map = readBinaryOccupancyGrid(msg,thresh,val)` specifies a value to set for unknown values (`-1`). By default, all unknown values are set to unoccupied, `0`.

Input Arguments

`msg` — `'nav_msgs/OccupancyGrid'` ROS message

`OccupancyGrid` object handle

`'nav_msgs/OccupancyGrid'` ROS message, specified as a `OccupancyGrid` object handle.

`thresh` — Threshold for occupied values

50 (default) | scalar

Threshold for occupied values, specified as a scalar. Any value greater than or equal to the threshold is set to occupied, `1`. All other values are set to unoccupied, `0`.

Data Types: `double`

`val` — Value to replace unknown values

0 (default) | 1

Value to replace unknown values, specified as either `0` or `1`. Unknown message values (`-1`) are set to the given value.

Data Types: `double` | `logical`

Output Arguments

map — Binary occupancy grid
binaryOccupancyMap object handle

Binary occupancy grid, returned as a binaryOccupancyMap object handle. map is converted from a 'nav_msgs/OccupancyGrid' message on the ROS network. The object is a grid of binary values, where 1 indicates an occupied location and 0 indications an unoccupied location.

Version History

Introduced in R2015a

See Also

Objects
OccupancyGrid | occupancyMap | binaryOccupancyMap

Functions
rosReadOccupancyGrid | rosWriteBinaryOccupancyGrid | rosWriteOccupancyGrid

roboticsAddons

Install add-ons for robotics

Syntax

```
roboticsAddons
```

Description

`roboticsAddons` allows you to download and install add-ons for Robotics System Toolbox. Use this function to open the Add-ons Explorer to browse the available add-ons.

Examples

Install Add-ons for Robotics System Toolbox™

To install add-ons for Robotics System Toolbox, run the function.

```
roboticsAddons
```

This function opens the Add-on Explorer with the Robotics System Toolbox set as the filter. Select the desired add-on and choose your install action.

Version History

Introduced in R2016a

See Also

Topics
"Install Robotics System Toolbox Add-ons"
"ROS Custom Message Support" (ROS Toolbox)
"Get and Manage Add-Ons"

roboticsSupportPackages

Download and install support packages for Robotics System Toolbox

Note `roboticsSupportPackages` has been removed. Use `roboticsAddons` instead.

Syntax

`roboticsSupportPackages`

Description

`roboticsSupportPackages` opens the Support Package Installer to download and install support packages for Robotics System Toolbox. For more details, see “Install Robotics System Toolbox Add-ons”.

Examples

Open Robotics System Toolbox Support Package Installer

```
roboticsSupportPackages
```

Version History

Introduced in R2015a

rotateframe

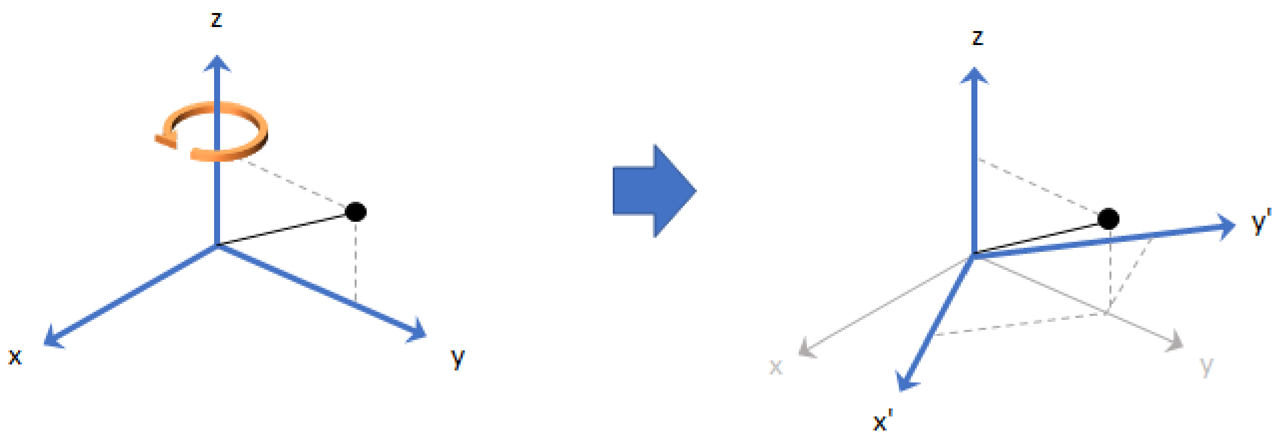
Quaternion frame rotation

Syntax

```
rotationResult = rotateframe(quat,cartesianPoints)
```

Description

`rotationResult = rotateframe(quat,cartesianPoints)` rotates the frame of reference for the Cartesian points using the quaternion, `quat`. The elements of the quaternion are normalized before use in the rotation.

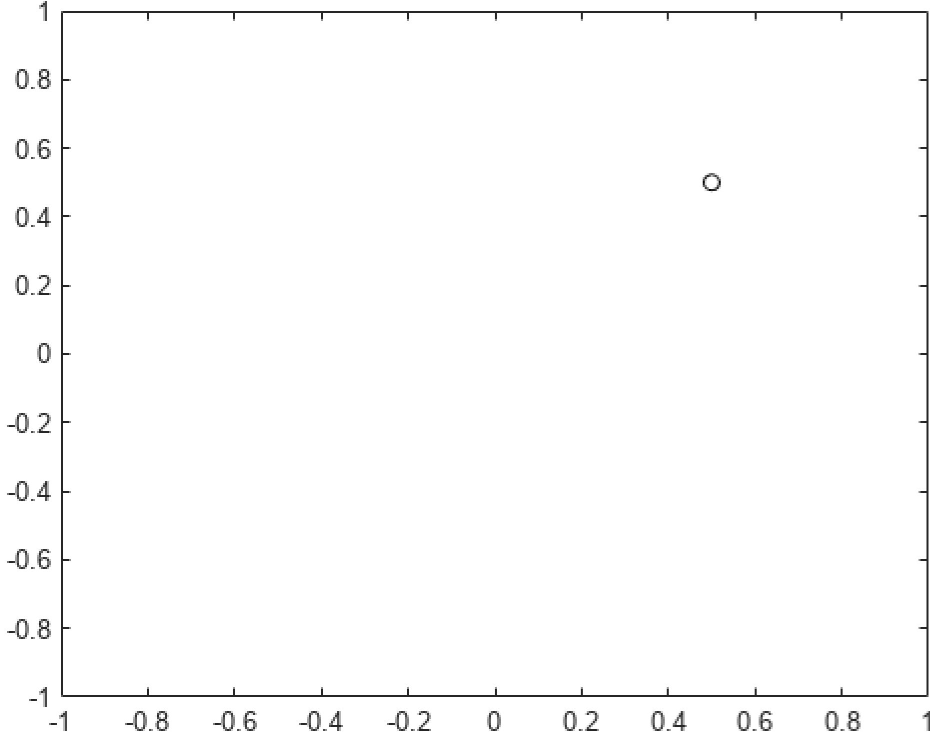


Examples

Rotate Frame Using Quaternion Vector

Define a point in three dimensions. The coordinates of a point are always specified in the order *x*, *y*, and *z*. For convenient visualization, define the point on the *x*-*y* plane.

```
x = 0.5;
y = 0.5;
z = 0;
plot(x,y,'ko')
hold on
axis([-1 1 -1 1])
```



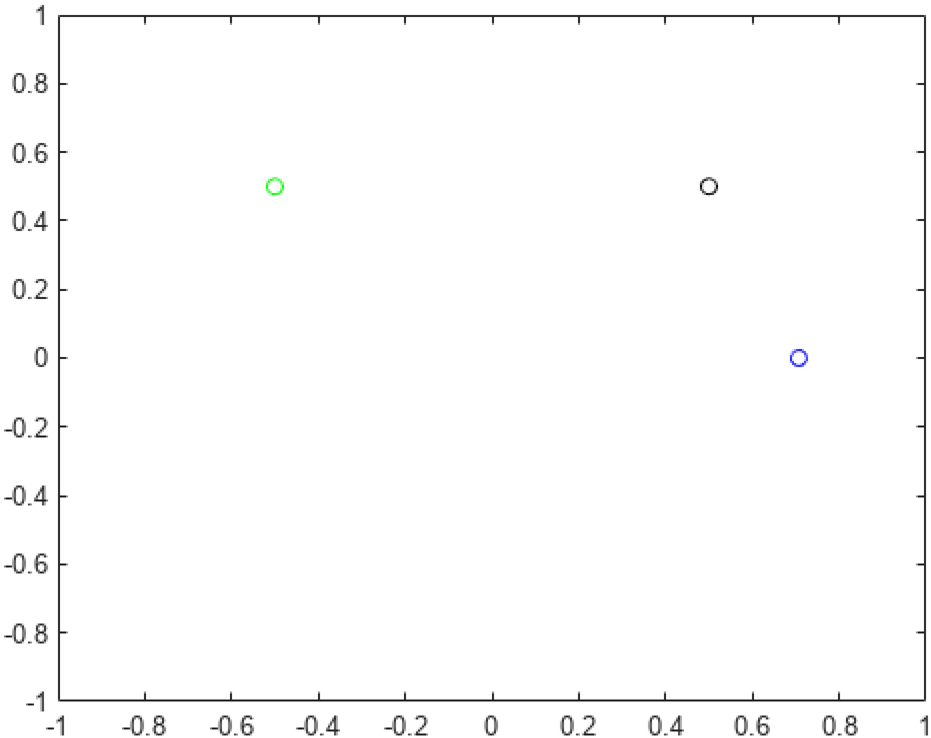
Create a quaternion vector specifying two separate rotations, one to rotate the frame 45 degrees and another to rotate the point -90 degrees about the *z*-axis. Use `rotateframe` to perform the rotations.

```
quat = quaternion([0,0,pi/4; ...
                   0,0,-pi/2],'euler','XYZ','frame');

rereferencedPoint = rotateframe(quat,[x,y,z])
```

```
rereferencedPoint = 2×3

    0.7071   -0.0000         0
   -0.5000    0.5000         0
```

Plot the rereferenced points.

```
plot(rereferencedPoint(1,1),rereferencedPoint(1,2),'bo')
plot(rereferencedPoint(2,1),rereferencedPoint(2,2),'go')
```



Rereference Group of Points using Quaternion

Define two points in three-dimensional space. Define a quaternion to rereference the points by first rotating the reference frame about the *z*-axis 30 degrees and then about the new *y*-axis 45 degrees.

```
a = [1,0,0];
b = [0,1,0];
quat = quaternion([30,45,0],'eulerd','ZYX','point');
```

Use `rotateframe` to reference both points using the quaternion rotation operator. Display the result.

```
rP = rotateframe(quat,[a;b])
```

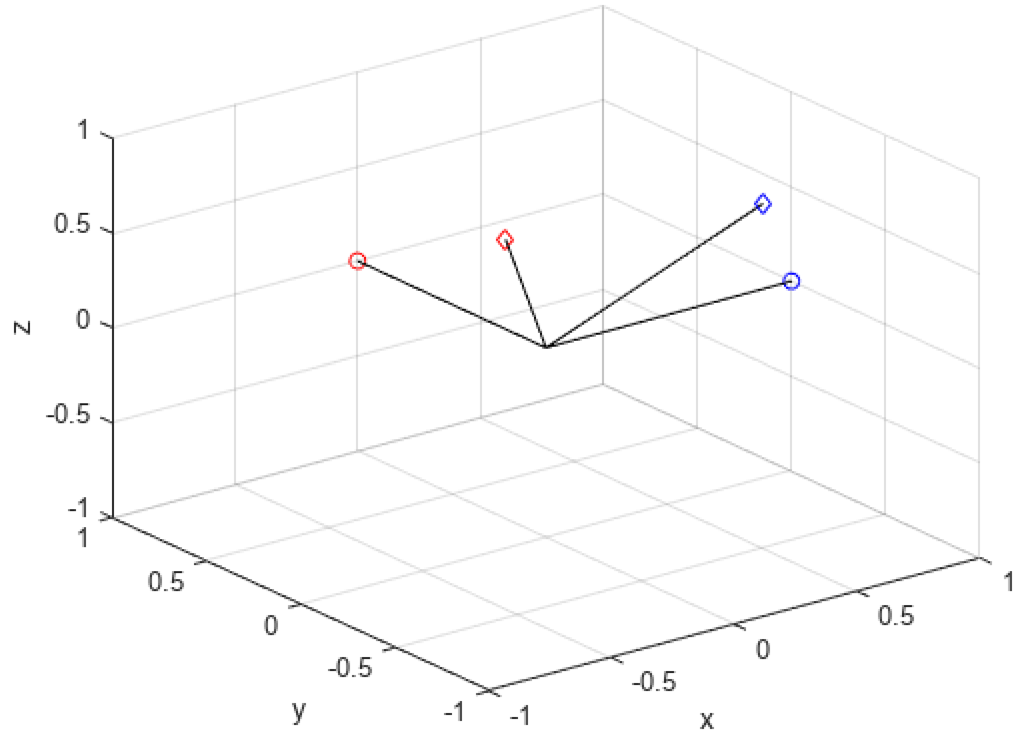
```
rP = 2×3

    0.6124   -0.3536    0.7071
    0.5000    0.8660   -0.0000
```

Visualize the original orientation and the rotated orientation of the points. Draw lines from the origin to each of the points for visualization purposes.

```
plot3(a(1),a(2),a(3),'bo');
```

```
hold on
```

```
grid on
axis([-1 1 -1 1 -1 1])
xlabel('x')
ylabel('y')
zlabel('z')

plot3(b(1),b(2),b(3),'ro');
plot3(rP(1,1),rP(1,2),rP(1,3),'bd')
plot3(rP(2,1),rP(2,2),rP(2,3),'rd')

plot3([0;rP(1,1)],[0;rP(1,2)],[0;rP(1,3)],'k')
plot3([0;rP(2,1)],[0;rP(2,2)],[0;rP(2,3)],'k')
plot3([0;a(1)],[0;a(2)],[0;a(3)],'k')
plot3([0;b(1)],[0;b(2)],[0;b(3)],'k')
```



Input Arguments

quat — Quaternion that defines rotation
scalar | vector

Quaternion that defines rotation, specified as a scalar quaternion or vector of quaternions.

Data Types: `quaternion`

`cartesianPoints` — Three-dimensional Cartesian points
1-by-3 vector | *N*-by-3 matrix

Three-dimensional Cartesian points, specified as a 1-by-3 vector or *N*-by-3 matrix.

Data Types: `single` | `double`

Output Arguments

`rotationResult` — Re-referenced Cartesian points
vector | matrix

Cartesian points defined in reference to rotated reference frame, returned as a vector or matrix the same size as `cartesianPoints`.

The data type of the re-referenced Cartesian points is the same as the underlying data type of `quat`.

Data Types: `single` | `double`

Algorithms

Quaternion frame rotation re-references a point specified in $\mathbf{R}^3$ by rotating the original frame of reference according to a specified quaternion:

$$L_q(u) = q^* u q$$

where q is the quaternion, * represents conjugation, and u is the point to rotate, specified as a quaternion.

For convenience, the `rotateframe` function takes a point in $\mathbf{R}^3$ and returns a point in $\mathbf{R}^3$. Given a function call with some arbitrary quaternion, $q = a + b\mathrm{i} + c\mathrm{j} + d\mathrm{k}$, and arbitrary coordinate, $[x,y,z]$,

```
point = [x,y,z];
rereferencedPoint = rotateframe(q,point)
```

the `rotateframe` function performs the following operations:

1. Converts point $[x,y,z]$ to a quaternion:

   $$u_q = 0 + xi + yj + zk$$

2. Normalizes the quaternion, q:

   $$q_n = \frac{q}{\sqrt{a^2 + b^2 + c^2 + d^2}}$$

3. Applies the rotation:

   $$v_q = q^* u_q q$$

4. Converts the quaternion output, v_q, back to $\mathbf{R}^3$

Version History

Introduced in R2018a

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

Functions

`rotatepoint`

Objects

`quaternion`

rotatepoint

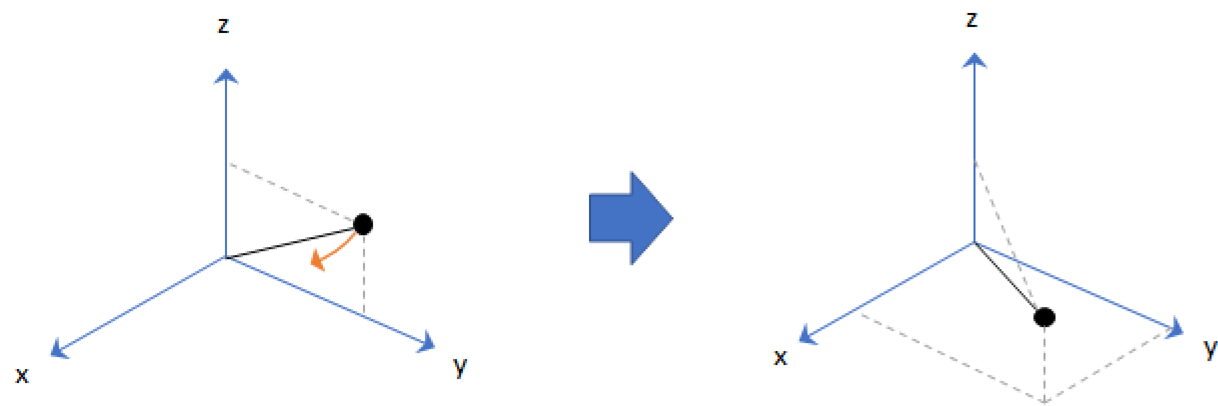
Quaternion point rotation

Syntax

```
rotationResult = rotatepoint(quat,cartesianPoints)
```

Description

`rotationResult = rotatepoint(quat,cartesianPoints)` rotates the Cartesian points using the quaternion, `quat`. The elements of the quaternion are normalized before use in the rotation.

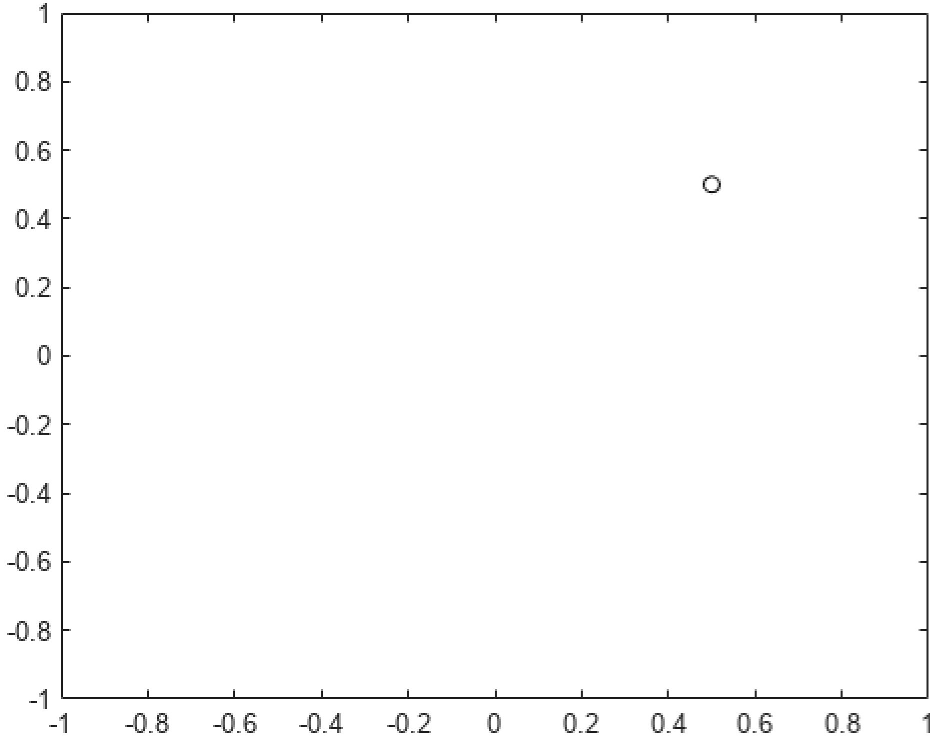


Examples

Rotate Point Using Quaternion Vector

Define a point in three dimensions. The coordinates of a point are always specified in order *x*, *y*, *z*. For convenient visualization, define the point on the *x*-*y* plane.

```
x = 0.5;
y = 0.5;
z = 0;

plot(x,y,'ko')
hold on
axis([-1 1 -1 1])
```



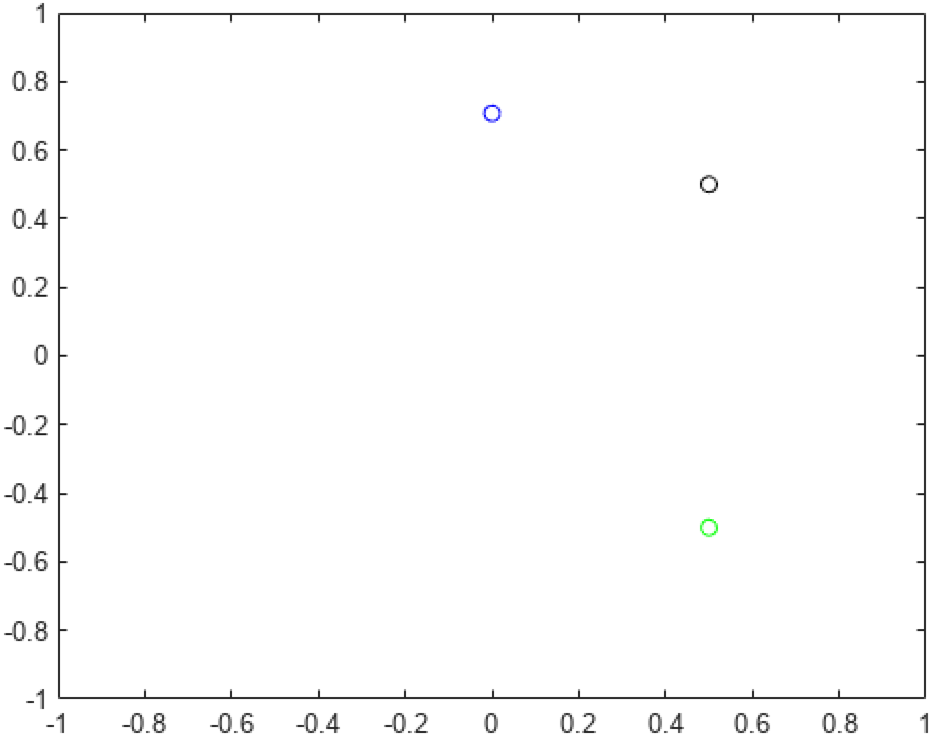
Create a quaternion vector specifying two separate rotations, one to rotate the point 45 and another to rotate the point -90 degrees about the *z*-axis. Use `rotatepoint` to perform the rotation.

```
quat = quaternion([0,0,pi/4; ...
                   0,0,-pi/2],'euler','XYZ','point');

rotatedPoint = rotatepoint(quat,[x,y,z])
```

```
rotatedPoint = 2×3

   -0.0000    0.7071         0
    0.5000   -0.5000         0
```

Plot the rotated points.

```
plot(rotatedPoint(1,1),rotatedPoint(1,2),'bo')
plot(rotatedPoint(2,1),rotatedPoint(2,2),'go')
```



Rotate Group of Points Using Quaternion

Define two points in three-dimensional space. Define a quaternion to rotate the point by first rotating about the *z*-axis 30 degrees and then about the new *y*-axis 45 degrees.

```
a = [1,0,0];
b = [0,1,0];
quat = quaternion([30,45,0],'eulerd','ZYX','point');
```

Use `rotatepoint` to rotate both points using the quaternion rotation operator. Display the result.

```
rP = rotatepoint(quat,[a;b])
```

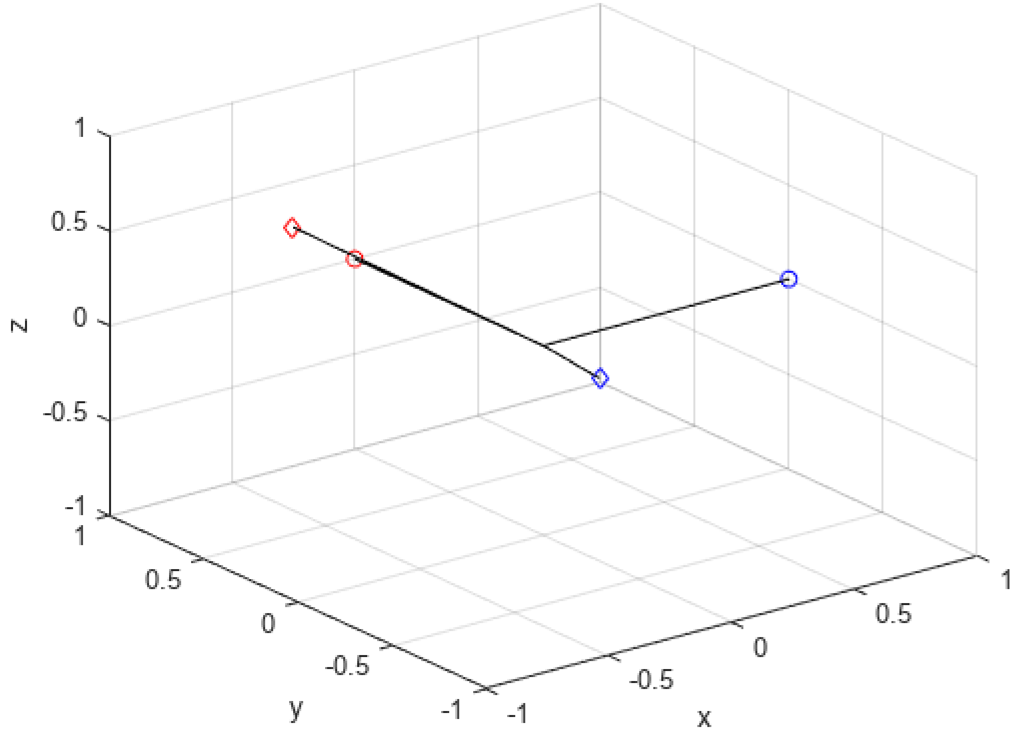
```
rP = 2×3

    0.6124    0.5000   -0.6124
   -0.3536    0.8660    0.3536
```

Visualize the original orientation and the rotated orientation of the points. Draw lines from the origin to each of the points for visualization purposes.

```
plot3(a(1),a(2),a(3),'bo');
```

```
hold on
```

```
grid on
axis([-1 1 -1 1 -1 1])
xlabel('x')
ylabel('y')
zlabel('z')

plot3(b(1),b(2),b(3),'ro');
plot3(rP(1,1),rP(1,2),rP(1,3),'bd')
plot3(rP(2,1),rP(2,2),rP(2,3),'rd')

plot3([0;rP(1,1)],[0;rP(1,2)],[0;rP(1,3)],'k')
plot3([0;rP(2,1)],[0;rP(2,2)],[0;rP(2,3)],'k')
plot3([0;a(1)],[0;a(2)],[0;a(3)],'k')
plot3([0;b(1)],[0;b(2)],[0;b(3)],'k')
```



Input Arguments

quat — Quaternion that defines rotation

scalar | vector

Quaternion that defines rotation, specified as a scalar quaternion, row vector of quaternions, or column vector of quaternions.

Data Types: `quaternion`

`cartesianPoints` — Three-dimensional Cartesian points
1-by-3 vector | *N*-by-3 matrix

Three-dimensional Cartesian points, specified as a 1-by-3 vector or *N*-by-3 matrix.

Data Types: `single` | `double`

Output Arguments

`rotationResult` — Repositioned Cartesian points
vector | matrix

Rotated Cartesian points defined using the quaternion rotation, returned as a vector or matrix the same size as `cartesianPoints`.

Data Types: `single` | `double`

Algorithms

Quaternion point rotation rotates a point specified in $\mathbf{R}^3$ according to a specified quaternion:

$$L_q(u) = quq*$$

where q is the quaternion, * represents conjugation, and u is the point to rotate, specified as a quaternion.

For convenience, the `rotatepoint` function takes in a point in $\mathbf{R}^3$ and returns a point in $\mathbf{R}^3$. Given a function call with some arbitrary quaternion, $q = a + b\text{i} + c\text{j} + d\text{k}$, and arbitrary coordinate, $[x,y,z]$, for example,

```
rereferencedPoint = rotatepoint(q,[x,y,z])
```

the `rotatepoint` function performs the following operations:

1. Converts point $[x,y,z]$ to a quaternion:

   $$u_q = 0 + xi + yj + zk$$

2. Normalizes the quaternion, q:

   $$q_n = \frac{q}{\sqrt{a^2 + b^2 + c^2 + d^2}}$$

3. Applies the rotation:

   $$v_q = qu_qq*$$

4. Converts the quaternion output, v_q, back to $\mathbf{R}^3$

Version History

Introduced in R2018a

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

Functions
`rotateframe`

Objects
`quaternion`

rotm2axang

Convert rotation matrix to axis-angle rotation

Syntax

```
axang = rotm2axang(rotm)
```

Description

`axang = rotm2axang(rotm)` converts a rotation given as an orthonormal rotation matrix, `rotm`, to the corresponding axis-angle representation, `axang`. The input rotation matrix must be in the premultiply form for rotations.

Examples

Convert Rotation Matrix to Axis-Angle Rotation

```
rotm = [1 0 0 ; 0 -1 0; 0 0 -1];
axang = rotm2axang(rotm)
```

```
axang = 1×4

    1.0000         0         0    3.1416
```

Input Arguments

`rotm` — Rotation matrix
3-by-3-by-*n* matrix

Rotation matrix, specified as a 3-by-3-by-*n* matrix containing *n* rotation matrices. Each rotation matrix has a size of 3-by-3 and must be orthonormal. The input rotation matrix must be in the premultiply form for rotations.

Note Rotation matrices that are slightly non-orthonormal can give complex outputs. Consider validating your matrix before inputting to the function.

Example: `[0 0 1; 0 1 0; -1 0 0]`

Output Arguments

axang — Rotation given in axis-angle form
n-by-4 matrix

Rotation given in axis-angle form, returned as an *n*-by-4 matrix of *n* axis-angle rotations. The first three elements of every row specify the rotation axis, and the last element defines the rotation angle (in radians).

Example: `[1 0 0 pi/2]`

Version History

Introduced in R2015a

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`axang2rotm` | `so3`

Topics

"Coordinate Transformations in Robotics"

rotm2eul

Convert rotation matrix to Euler angles

Syntax

```
eul = rotm2eul(rotm)
eul = rotm2eul(rotm,sequence)
[eul,eulAlt] = rotm2eul( ___ )
```

Description

`eul = rotm2eul(rotm)` converts a rotation matrix, `rotm`, to the corresponding Euler angles, `eul`. The input rotation matrix must be in the premultiply form for rotations. The default order for Euler angle rotations is `"ZYX"`.

For more details on Euler angle rotations, see “Euler Angles”.

`eul = rotm2eul(rotm,sequence)` converts a rotation matrix to Euler angles. The Euler angles are specified in the axis rotation sequence, `sequence`. The default order for Euler angle rotations is `"ZYX"`.

`[eul,eulAlt] = rotm2eul( ___ )` also returns an alternate set of Euler angles that represents the same rotation `eulAlt`.

Examples

Convert Rotation Matrix to Euler Angles

```
rotm = [0 0 1; 0 1 0; -1 0 0];
eulZYX = rotm2eul(rotm)
```

```
eulZYX = 1×3

         0    1.5708         0
```

Convert Rotation Matrix to Euler Angles Using ZYZ Axis Order

```
rotm = [0 0 1; 0 1 0; -1 0 0];
eulZYZ = rotm2eul(rotm,'ZYZ')
```

```
eulZYZ = 1×3

   -3.1416   -1.5708   -3.1416
```

Input Arguments

`rotm` — Rotation matrix
3-by-3-by-*n* matrix

Rotation matrix, specified as a 3-by-3-by-*n* matrix containing *n* rotation matrices. Each rotation matrix has a size of 3-by-3 and is orthonormal. The input rotation matrix must be in the premultiply form for rotations.

Note Rotation matrices that are slightly non-orthonormal can give complex outputs. Consider validating your matrix before inputting to the function.

Example: `[0 0 1; 0 1 0; -1 0 0]`

`sequence` — Axis-rotation sequence
`"ZYX"` (default) | `"ZYZ"` | `"ZXY"` | `"ZXZ"` | `"YXY"` | `"YZX"` | `"YXZ"` | `"YZY"` | `"XYX"` | `"XYZ"` | `"XZX"` | `"XZY"`

Axis-rotation sequence for the Euler angles, specified as one of these string scalars:

- `"ZYX"` (default)
- `"ZYZ"`
- `"ZXY"`
- `"ZXZ"`
- `"YXY"`
- `"YZX"`
- `"YXZ"`
- `"YZY"`
- `"XYX"`
- `"XYZ"`
- `"XZX"`
- `"XZY"`

Each character indicates the corresponding axis. For example, if the sequence is `"ZYX"`, then the three specified Euler angles are interpreted in order as a rotation around the *z*-axis, a rotation around the *y*-axis, and a rotation around the *x*-axis. When applying this rotation to a point, it will apply the axis rotations in the order *x*, then *y*, then *z*.

Data Types: `string` | `char`

Output Arguments

`eul` — Euler rotation angles
n-by-3 matrix

Euler rotation angles in radians, returned as an *n*-by-3 array of Euler rotation angles. Each row represents one Euler angle set.

Example: `[0 0 1.5708]`

`eulAlt` — Alternate Euler rotation angle solution
n-by-3 matrix

Alternate Euler rotation angle solution in radians, returned as an *n*-by-3 array of Euler rotation angles. Each row represents one Euler angle set.

Example: `[0 0 1.5708]`

Version History

Introduced in R2015a

R2020a: Alternate Euler angle output

`rotm2eul` now optionally outputs an alternate set of Euler angles `eulAlt` that also represent the same rotation as the original output Euler angles `eul`. So if you use `eul` or `eulAlt` to rotate a point, the resulting point is the same.

R2023a: Additional Euler sequence support

`rotm2eul` supports additional Euler sequences for the `sequences` argument. These are all the supported Euler sequences:

- `"ZYX"`
- `"ZYZ"`
- `"ZXY"`
- `"ZXZ"`
- `"YXY"`
- `"YZX"`
- `"YXZ"`
- `"YZY"`
- `"XYX"`
- `"XYZ"`
- `"XZX"`
- `"XZY"`

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`eul2rotm` | `so2` | `so3`

Topics

"Coordinate Transformations in Robotics"

rotm2quat

Convert rotation matrix to quaternion

Syntax

```
quat = rotm2quat(rotm)
```

Description

`quat = rotm2quat(rotm)` converts a rotation matrix, `rotm`, to the corresponding unit quaternion representation, `quat`. The input rotation matrix must be in the premultiply form for rotations.

Examples

Convert Rotation Matrix to Quaternion

```
rotm = [0 0 1; 0 1 0; -1 0 0];
quat = rotm2quat(rotm)
```

```
quat = 1×4

    0.7071         0    0.7071         0
```

Input Arguments

`rotm` — Rotation matrix
3-by-3-by-*n* matrix

Rotation matrix, specified as a 3-by-3-by-*n* matrix containing *n* rotation matrices. Each rotation matrix has a size of 3-by-3 and is orthonormal. The input rotation matrix must be in the premultiply form for rotations.

Note Rotation matrices that are slightly non-orthonormal can give complex outputs. Consider validating your matrix before inputting to the function.

Example: `[0 0 1; 0 1 0; -1 0 0]`

Output Arguments

`quat` — Unit quaternion
n-by-4 matrix

Unit quaternion, returned as an *n*-by-4 matrix containing *n* quaternions. Each quaternion, one per row, is of the form $q = [w\ x\ y\ z]$, with w as the scalar number.

Example: `[0.7071 0.7071 0 0]`

Version History

Introduced in R2015a

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`quat2rotm` | `so3` | `quaternion`

Topics

"Coordinate Transformations in Robotics"

rotm2tform

Convert rotation matrix to homogeneous transformation

Syntax

```
tform = rotm2tform(rotm)
```

Description

`tform = rotm2tform(rotm)` converts the rotation matrix `rotm` into a homogeneous transformation matrix `tform`. The input rotation matrix must be in the premultiply form for rotations. When using the transformation matrix, premultiply it by the coordinates to be transformed (as opposed to postmultiplying).

Examples

Convert Rotation Matrix to Homogeneous Transformation

```
rotm = [1 0 0 ; 0 -1 0; 0 0 -1];
tform = rotm2tform(rotm)
```

```
tform = 4×4

     1     0     0     0
     0    -1     0     0
     0     0    -1     0
     0     0     0     1
```

Input Arguments

`rotm` — Rotation matrix

2-by-2-by-*n* array | 3-by-3-by-*n* array

Rotation matrix, specified as a 2-by-2-by-*n* or a 3-by-3-by-*n* array containing *n* rotation matrices. Each rotation matrix is either 2-by-2 or 3-by-3 and is orthonormal. The input rotation matrix must be in the premultiplied form for rotations.

Note Rotation matrices that are not orthonormal can be normalized with the `normalize` function.

2-D rotation matrices are of this form:

3-D rotation matrices are of this form:

Example: `[0 0 1; 0 1 0; -1 0 0]`

Output Arguments

`tform` — Homogeneous transformation
3-by-3-by-*n* array | 4-by-4-by-*n* array

Homogeneous transformation, returned as a 3-by-3-by-*n* array or 4-by-4-by-*n* array. *n* is the number of homogeneous transformations. When using the transformation matrix, premultiply it by the coordinates to be transformed (as opposed to postmultiplying).

2-D homogeneous transformation matrices are of this form:

$$T = \begin{bmatrix} r_{11} & r_{12} & t_1 \\ r_{21} & r_{22} & t_2 \\ 0 & 0 & 1 \end{bmatrix}$$

3-D homogeneous transformation matrices are of this form:

$$T = \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_1 \\ r_{21} & r_{22} & r_{23} & t_2 \\ r_{31} & r_{32} & r_{33} & t_3 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Example: `[0 0 1 0; 0 1 0 0; -1 0 0 0; 0 0 0 1]`

More About

2-D Homogeneous Transformation Matrix

2-D homogeneous transformation matrices consist of both an SO(2) rotation and an *xy*-translation.

To read more about SO(2) rotations, see the “2-D Orthonormal Rotation Matrix” on page 1-399 section of the `so2` object.

The translation is along the *x*-, *y*-, and *z*-axes as a three-element column vector:

$$t = \begin{bmatrix} x \\ y \end{bmatrix}$$

The SO(2) rotation matrix *R* is applied to the translation vector *t* to create the homogeneous translation matrix *T*. The rotation matrix is present in the upper-left of the transformation matrix as 2-by-2 submatrix, and the translation vector is present as a two-element vector in the last column.

$$T = \begin{bmatrix} R & t \\ 0_{1\times 2} & 1 \end{bmatrix} = \begin{bmatrix} I_2 & t \\ 0_{1\times 2} & 1 \end{bmatrix} \cdot \begin{bmatrix} R & 0 \\ 0_{1\times 2} & 1 \end{bmatrix}$$

3-D Homogeneous Transformation Matrix

3-D homogeneous transformation matrices consist of both an SO(3) rotation and an *xyz*-translation.

To read more about SO(3) rotations, see the “3-D Orthonormal Rotation Matrix” on page 1-395 section of the `so3` object.

The translation is along the *x*-, *y*-, and *z*-axes as a three-element column vector:

$$t = \begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

The SO(3) rotation matrix R is applied to the translation vector t to create the homogeneous translation matrix T. The rotation matrix is present in the upper-left of the transformation matrix as 3-by-3 submatrix, and the translation vector is present as a three-element vector in the last column.

$$T = \begin{bmatrix} R & t \\ 0_{1x3} & 1 \end{bmatrix} = \begin{bmatrix} I_3 & t \\ 0_{1x3} & 1 \end{bmatrix} \cdot \begin{bmatrix} R & 0 \\ 0_{1x3} & 1 \end{bmatrix}$$

Version History

Introduced in R2015a

R2023a: `rotm2tform` Supports 2-D Rotation Matrices

The `rotm` argument now accepts 2-D rotation matrices as a 2-by-2-by-*n* array and `rotm2tform` outputs 2-D transformation matrices as a 3-by-3-by-*n* array.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`tform2rotm` | `se2` | `se3` | `so2` | `so3`

Topics

"Coordinate Transformations in Robotics"

rotmat

Convert quaternion to rotation matrix

Syntax

```
rotationMatrix = rotmat(quat,rotationType)
```

Description

`rotationMatrix = rotmat(quat,rotationType)` converts the quaternion, `quat`, to an equivalent rotation matrix representation.

Examples

Convert Quaternion to Rotation Matrix for Point Rotation

Define a quaternion for use in point rotation.

```
theta = 45;
gamma = 30;
quat = quaternion([0,theta,gamma],'eulerd','ZYX','point')
```

```
quat = quaternion
       0.8924 +  0.23912i +  0.36964j + 0.099046k
```

Convert the quaternion to a rotation matrix.

```
rotationMatrix = rotmat(quat,'point')
```

```
rotationMatrix = 3×3

    0.7071   -0.0000    0.7071
    0.3536    0.8660   -0.3536
   -0.6124    0.5000    0.6124
```

To verify the rotation matrix, directly create two rotation matrices corresponding to the rotations about the *y*- and *x*-axes. Multiply the rotation matrices and compare to the output of `rotmat`.

```
theta = 45;
gamma = 30;

ry = [cosd(theta)   0           sind(theta) ; ...
      0             1           0           ; ...
     -sind(theta)   0           cosd(theta)];

rx = [1             0           0           ;      ...
      0             cosd(gamma) -sind(gamma) ;     ...
      0             sind(gamma) cosd(gamma)];

rotationMatrixVerification = rx*ry
```

```
rotationMatrixVerification = 3×3

    0.7071         0    0.7071
    0.3536    0.8660   -0.3536
   -0.6124    0.5000    0.6124
```

Convert Quaternion to Rotation Matrix for Frame Rotation

Define a quaternion for use in frame rotation.

```
theta = 45;
gamma = 30;
quat = quaternion([0,theta,gamma],'eulerd','ZYX','frame')
```

```
quat = quaternion
       0.8924 +  0.23912i +  0.36964j - 0.099046k
```

Convert the quaternion to a rotation matrix.

```
rotationMatrix = rotmat(quat,'frame')
```

```
rotationMatrix = 3×3

    0.7071   -0.0000   -0.7071
    0.3536    0.8660    0.3536
    0.6124   -0.5000    0.6124
```

To verify the rotation matrix, directly create two rotation matrices corresponding to the rotations about the *y*- and *x*-axes. Multiply the rotation matrices and compare to the output of `rotmat`.

```
theta = 45;
gamma = 30;

ry = [cosd(theta)   0           -sind(theta) ; ...
      0             1           0            ; ...
     sind(theta)   0           cosd(theta)];

rx = [1             0           0            ;      ...
      0             cosd(gamma) sind(gamma) ;     ...
      0             -sind(gamma) cosd(gamma)];

rotationMatrixVerification = rx*ry
```

```
rotationMatrixVerification = 3×3

    0.7071         0   -0.7071
    0.3536    0.8660    0.3536
    0.6124   -0.5000    0.6124
```

Convert Quaternion Vector to Rotation Matrices

Create a 3-by-1 normalized quaternion vector.

```
qVec = normalize(quaternion(randn(3,4)));
```

Convert the quaternion array to rotation matrices. The pages of `rotmatArray` correspond to the linear index of `qVec`.

```
rotmatArray = rotmat(qVec,'frame');
```

Assume `qVec` and `rotmatArray` correspond to a sequence of rotations. Combine the quaternion rotations into a single representation, then apply the quaternion rotation to arbitrarily initialized Cartesian points.

```
loc = normalize(randn(1,3));
quat = prod(qVec);
rotateframe(quat,loc)
```

```
ans = 1×3

    0.9524    0.5297    0.9013
```

Combine the rotation matrices into a single representation, then apply the rotation matrix to the same initial Cartesian points. Verify the quaternion rotation and rotation matrix result in the same orientation.

```
totalRotMat = eye(3);
for i = 1:size(rotmatArray,3)
    totalRotMat = rotmatArray(:,:,i)*totalRotMat;
end
totalRotMat*loc'
```

```
ans = 3×1

    0.9524
    0.5297
    0.9013
```

Input Arguments

quat — Quaternion to convert
scalar | vector | matrix | multidimensional array

Quaternion to convert, specified as a scalar, vector, matrix, or multidimensional array.

Data Types: `quaternion`

`rotationType` — Type or rotation
`'frame'` | `'point'`

Type of rotation represented by the `rotationMatrix` output, specified as `'frame'` or `'point'`.

Data Types: `char` | `string`

Output Arguments

`rotationMatrix` — Rotation matrix representation
3-by-3 matrix | 3-by-3-by-*N* multidimensional array

Rotation matrix representation, returned as a 3-by-3 matrix or 3-by-3-by-*N* multidimensional array.

- If `quat` is a scalar, `rotationMatrix` is returned as a 3-by-3 matrix.
- If `quat` is non-scalar, `rotationMatrix` is returned as a 3-by-3-by-*N* multidimensional array, where `rotationMatrix(:,:,i)` is the rotation matrix corresponding to `quat(i)`.

The data type of the rotation matrix is the same as the underlying data type of `quat`.

Data Types: `single` | `double`

Algorithms

Given a quaternion of the form

$$q = a + bi + cj + dk\,,$$

the equivalent rotation matrix for frame rotation is defined as

$$\begin{bmatrix} 2a^2 - 1 + 2b^2 & 2bc + 2ad & 2bd - 2ac \\ 2bc - 2ad & 2a^2 - 1 + 2c^2 & 2cd + 2ab \\ 2bd + 2ac & 2cd - 2ab & 2a^2 - 1 + 2d^2 \end{bmatrix}.$$

The equivalent rotation matrix for point rotation is the transpose of the frame rotation matrix:

$$\begin{bmatrix} 2a^2 - 1 + 2b^2 & 2bc - 2ad & 2bd + 2ac \\ 2bc + 2ad & 2a^2 - 1 + 2c^2 & 2cd - 2ab \\ 2bd - 2ac & 2cd + 2ab & 2a^2 - 1 + 2d^2 \end{bmatrix}.$$

Version History

Introduced in R2018a

References

[1] Kuipers, Jack B. *Quaternions and Rotation Sequences: A Primer with Applications to Orbits, Aerospace, and Virtual Reality*. Princeton, NJ: Princeton University Press, 2007.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

Functions
`rotvec` | `rotvecd` | `euler` | `eulerd`

Objects
`quaternion`

rottraj

Generate trajectories between orientation rotation matrices

Syntax

```
[R,omega,alpha] = rottraj(r0,rF,tInterval,tSamples)
[R,omega,alpha] = rottraj(r0,rF,tInterval,tSamples,Name=Value)
```

Description

`[R,omega,alpha] = rottraj(r0,rF,tInterval,tSamples)` generates a trajectory that interpolates between two orientations, `r0` and `rF`, with points based on the time interval and given time samples.

`[R,omega,alpha] = rottraj(r0,rF,tInterval,tSamples,Name=Value)` specifies additional parameters using name-value arguments.

Examples

Interpolate Trajectory Between Quaternions

Define two quaternion waypoints to interpolate between.

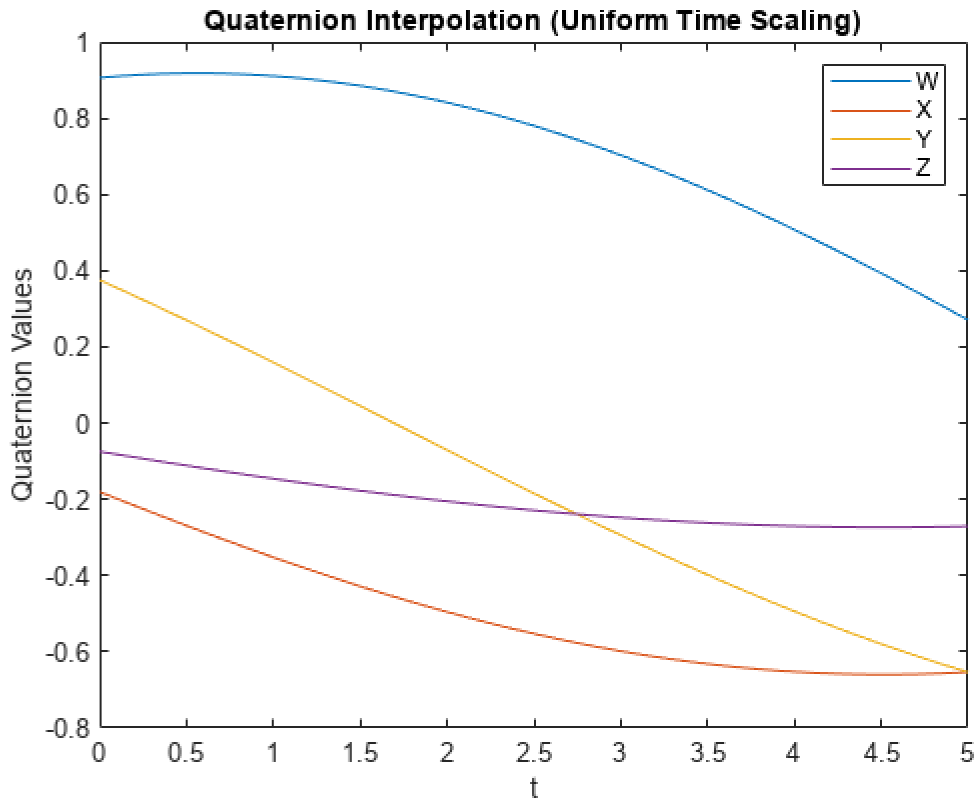
```
q0 = quaternion([0 pi/4 -pi/8],'euler','ZYX','point');
qF = quaternion([3*pi/2 0 -3*pi/4],'euler','ZYX','point');
```

Specify a vector of times to sample the quaternion trajectory.

```
tvec = 0:0.01:5;
```

Generate the trajectory. Plot the results.

```
[qInterp1,w1,a1] = rottraj(q0,qF,[0 5],tvec);

plot(tvec,compact(qInterp1))
title('Quaternion Interpolation (Uniform Time Scaling)')
xlabel('t')
ylabel('Quaternion Values')
legend('W','X','Y','Z')
```



Interpolate Trajectory Between Rotation Matrices

Define two rotation matrix waypoints to interpolate between.

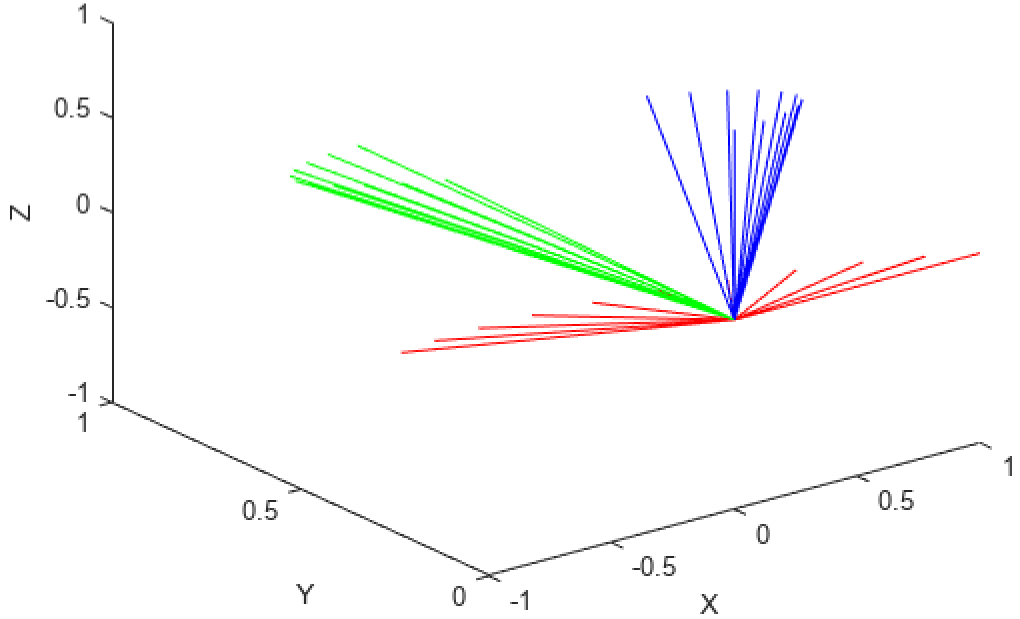
```
r0 = [1 0 0; 0 1 0; 0 0 1];
rF = [0 0 1; 1 0 0; 0 0 0];
```

Specify a vector of times to sample the quaternion trajectory.

```
tvec = 0:0.1:1;
```

Generate the trajectory. Plot the results using `plotTransforms`. Convert the rotation matrices to quaternions and specify zero translation. The figure shows all the intermediate rotations of the coordinate frame.

```
[rInterp1,w1,a1] = rottraj(r0,rF,[0 1],tvec);

rotations = rotm2quat(rInterp1);
zeroVect = zeros(length(rotations),1);
translations = [zeroVect,zeroVect,zeroVect];

plotTransforms(translations,rotations)
xlabel('X')
ylabel('Y')
zlabel('Z')
```



Input Arguments

r0 — Initial orientation

3-by-3 rotation matrix | `quaternion` object | `so3` object

Initial orientation, specified as a 3-by-3 rotation matrix, a scalar `quaternion` object, or a scalar `so3` object. The function generates a trajectory that starts at the initial orientation, `r0`, and goes to the final orientation, `rF`.

`r0` and `rF` must be of the same type. For example, if `r0` is a scalar `so3` object, then `rF` must be a scalar `so3` object.

Example: `quaternion([0 pi/4 -pi/8],"euler","ZYX","point");`

Data Types: `single` | `double`

rF — Final orientation

3-by-3 rotation matrix | `quaternion` object | `so3` object

Final orientation, specified as a 3-by-3 rotation matrix, a scalar `quaternion` object, or a scalar `so3` object. The function generates a trajectory that starts at the initial orientation, `r0`, and goes to the final orientation, `rF`.

`r0` and `rF` must be of the same type. For example, if `r0` is a scalar `so3` object, then `rF` must be a scalar `so3` object.

Example: `quaternion([3*pi/2 0 -3*pi/4],"euler","ZYX","point")`

Data Types: `single` | `double`

`tInterval` — Start and end times for trajectory
two-element vector

Start and end times for the trajectory, specified as a two-element vector.

Example: `[0 10]`

Data Types: `single` | `double`

`tSamples` — Time samples for trajectory
m-element vector

Time samples for the trajectory, specified as an *m*-element vector.

Example: `0:0.01:10`

Data Types: `single` | `double`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `TimeScaling=[0 1 2; 0 1 0; 0 0 0]`

`TimeScaling` — Time scaling vector and first two derivatives
3-by-*m* vector

Time scaling vector and the first two derivatives, specified as the comma-separated pair of `TimeScaling` and a 3-by-*m* vector, where *m* is the length of `tSamples`. By default, the time scaling is a linear time scaling between the time points in `tInterval`.

For a nonlinear time scaling, specify the values of the time points in the first row. The second and third rows are the velocity and acceleration of the time points, respectively. For example, to follow the path with a linear velocity to the halfway point, and then jump to the end, the time-scaling would be:

```
s(1,:) = [0 0.25 0.5 1 1 1] % Position
s(2,:) = [1    1   1 0 0 0] % Velocity
s(3,:) = [0    0   0 0 0 0] % Acceleration
```

Data Types: `single` | `double`

Output Arguments

`R` — Orientation trajectory
3-by-3-by-*m* rotation matrix array | *m*-element array of `quaternion` objects | *m*-element array of `so3` objects

Orientation trajectory, returned as a 3-by-3-by-*m* rotation matrix array, an *m*-element array of `quaternion` objects, or an *m*-element array of `so3` objects. *m* is the number of points in `tSamples`. The output type matches the input type of `r0` and `rF`.

omega — Orientation angular velocity
3-by-*m* matrix

Orientation angular velocity, returned as a 3-by-*m* matrix, where *m* is the number of points in `tSamples`.

alpha — Orientation angular acceleration
3-by-*m* matrix

Orientation angular acceleration, returned as a 3-by-*m* matrix, where *m* is the number of points in `tSamples`

Limitations

- When specifying your `r0` and `rF` input arguments as a 3-by-3 rotation matrix, they are converted to a `quaternion` object before interpolating the trajectory. If your rotation matrix does not follow a right-handed coordinate system or does not have a direct conversion to quaternions, this conversion may result in different initial and final rotations in the output trajectory.

Version History

Introduced in R2019a

R2023a: `rottraj` supports SO(3) rotation object inputs

The `r0` and `rF` arguments of `rottraj` now accept `so3` objects and the resulting output value for the `R` argument is an *m*-element array of `so3` objects. *m* is the number of points in `tSamples`.

References

[1] Dam, Erik B., Martin Koch, and Martin Lillholm. *Quaternions, Interpolation and Animation*. Technical Report DIKU-TR-98/5 (July 1998). http://web.mit.edu/2.998/www/QuaternionReport1.pdf

[2] Graf, Basile. *Quaternions and Dynamics*. arXiv:0811.2889 [math.DS] (2008). https://arxiv.org/pdf/0811.2889.pdf

[3] Lynch, Kevin M., and Frank C. Park. *Modern Robotics: Mechanics, Planning, and Control*. Cambridge University Press, 2017.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

`bsplinepolytraj` | `contopptraj` | `cubicpolytraj` | `quinticpolytraj` | `transformtraj` | `trapveltraj` | `quaternion`

rotvec

Convert quaternion to rotation vector (radians)

Syntax

```
rotationVector = rotvec(quat)
```

Description

`rotationVector = rotvec(quat)` converts the quaternion array, `quat`, to an *N*-by-3 matrix of equivalent rotation vectors in radians. The elements of `quat` are normalized before conversion.

Examples

Convert Quaternion to Rotation Vector in Radians

Convert a random quaternion scalar to a rotation vector in radians

```
quat = quaternion(randn(1,4));
rotvec(quat)
```

```
ans = 1×3

    1.6866   -2.0774    0.7929
```

Input Arguments

`quat` — Quaternion to convert
scalar | vector | matrix | multidimensional array

Quaternion to convert, specified as scalar quaternion, vector, matrix, or multidimensional array of quaternions.

Data Types: `quaternion`

Output Arguments

`rotationVector` — Rotation vector (radians)
N-by-3 matrix

Rotation vector representation, returned as an *N*-by-3 matrix of rotations vectors, where each row represents the [X Y Z] angles of the rotation vectors in radians. The *i*th row of `rotationVector` corresponds to the element `quat(i)`.

The data type of the rotation vector is the same as the underlying data type of `quat`.

Data Types: `single` | `double`

Algorithms

All rotations in 3-D can be represented by a three-element axis of rotation and a rotation angle, for a total of four elements. If the rotation axis is constrained to be unit length, the rotation angle can be distributed over the vector elements to reduce the representation to three elements.

Recall that a quaternion can be represented in axis-angle form

$$q = \cos\left(\theta/2\right) + \sin\left(\theta/2\right)(\mathrm{x}\mathrm{i} + y\mathrm{j} + z\mathrm{k}),$$

where θ is the angle of rotation and $[x,y,z]$ represent the axis of rotation.

Given a quaternion of the form

$$q = a + bi + cj + dk\,,$$

you can solve for the rotation angle using the axis-angle form of quaternions:

$$\theta = 2\cos^{-1}(a)\,.$$

Assuming a normalized axis, you can rewrite the quaternion as a rotation vector without loss of information by distributing θ over the parts b, c, and d. The rotation vector representation of q is

$$q_{\mathrm{rv}} = \frac{\theta}{\sin\left(\theta/2\right)}[b, c, d]\,.$$

Version History

Introduced in R2018a

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

Functions
`rotvecd` | `euler` | `eulerd`

Objects
`quaternion`

rotvecd

Convert quaternion to rotation vector (degrees)

Syntax

```
rotationVector = rotvecd(quat)
```

Description

`rotationVector = rotvecd(quat)` converts the quaternion array, `quat`, to an *N*-by-3 matrix of equivalent rotation vectors in degrees. The elements of `quat` are normalized before conversion.

Examples

Convert Quaternion to Rotation Vector in Degrees

Convert a random quaternion scalar to a rotation vector in degrees.

```
quat = quaternion(randn(1,4));
rotvecd(quat)
```

```
ans = 1×3

   96.6345 -119.0274   45.4312
```

Input Arguments

`quat` — Quaternion to convert
scalar | vector | matrix | multidimensional array

Quaternion to convert, specified as scalar, vector, matrix, or multidimensional array of quaternions.

Data Types: `quaternion`

Output Arguments

`rotationVector` — Rotation vector (degrees)
N-by-3 matrix

Rotation vector representation, returned as an *N*-by-3 matrix of rotation vectors, where each row represents the [*x y z*] angles of the rotation vectors in degrees. The *i*th row of `rotationVector` corresponds to the element `quat(i)`.

The data type of the rotation vector is the same as the underlying data type of `quat`.

Data Types: `single` | `double`

Algorithms

All rotations in 3-D can be represented by four elements: a three-element axis of rotation and a rotation angle. If the rotation axis is constrained to be unit length, the rotation angle can be distributed over the vector elements to reduce the representation to three elements.

Recall that a quaternion can be represented in axis-angle form

$$q = \cos\left(\theta/2\right) + \sin\left(\theta/2\right)(\mathrm{xi} + y\mathrm{j} + z\mathrm{k}),$$

where θ is the angle of rotation in degrees, and [x,y,z] represent the axis of rotation.

Given a quaternion of the form

$$q = a + bi + cj + dk\,,$$

you can solve for the rotation angle using the axis-angle form of quaternions:

$$\theta = 2\cos^{-1}(a)\,.$$

Assuming a normalized axis, you can rewrite the quaternion as a rotation vector without loss of information by distributing θ over the parts b, c, and d. The rotation vector representation of q is

$$q_{\mathrm{rv}} = \frac{\theta}{\sin\left(\theta/2\right)}[b, c, d]\,.$$

Version History

Introduced in R2018b

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

Functions
`rotvec` | `euler` | `eulerd`

Objects
`quaternion`

slerp

Spherical linear interpolation

Syntax

```
q0 = slerp(q1,q2,T)
```

Description

`q0 = slerp(q1,q2,T)` spherically interpolates between `q1` and `q2` by the interpolation coefficient `T`. The function always chooses the shorter interpolation path between `q1` and `q2`.

Examples

Interpolate Between Two Quaternions

Create two quaternions with the following interpretation:

1. `a` = 45 degree rotation around the *z*-axis
2. `c` = -45 degree rotation around the *z*-axis

```
a = quaternion([45,0,0],'eulerd','ZYX','frame');
c = quaternion([-45,0,0],'eulerd','ZYX','frame');
```

Call `slerp` with the quaternions `a` and `c` and specify an interpolation coefficient of 0.5.

```
interpolationCoefficient = 0.5;

b = slerp(a,c,interpolationCoefficient);
```

The output of `slerp`, `b`, represents an average rotation of `a` and `c`. To verify, convert `b` to Euler angles in degrees.

```
averageRotation = eulerd(b,'ZYX','frame')
```

```
averageRotation = 1×3

     0     0     0
```

The interpolation coefficient is specified as a normalized value between `0` and `1`, inclusive. An interpolation coefficient of `0` corresponds to the `a` quaternion, and an interpolation coefficient of `1` corresponds to the `c` quaternion. Call `slerp` with coefficients `0` and `1` to confirm.

```
b = slerp(a,c,[0,1]);
eulerd(b,'ZYX','frame')
```

```
ans = 2×3

   45.0000         0         0
```

```
  -45.0000         0         0
```

You can create smooth paths between quaternions by specifying arrays of equally spaced interpolation coefficients.

```
path = 0:0.1:1;

interpolatedQuaternions = slerp(a,c,path);
```

For quaternions that represent rotation only about a single axis, specifying interpolation coefficients as equally spaced results in quaternions equally spaced in Euler angles. Convert `interpolatedQuaternions` to Euler angles and verify that the difference between the angles in the path is constant.

```
k = eulerd(interpolatedQuaternions,'ZYX','frame');
abc = abs(diff(k))
```

```
abc = 10×3

    9.0000         0         0
    9.0000         0         0
    9.0000         0         0
    9.0000         0         0
    9.0000         0         0
    9.0000         0         0
    9.0000         0         0
    9.0000         0         0
    9.0000         0         0
    9.0000         0         0
```

Alternatively, you can use the `dist` function to verify that the distance between the interpolated quaternions is consistent. The `dist` function returns angular distance in radians; convert to degrees for easy comparison.

```
def = rad2deg(dist(interpolatedQuaternions(2:end),interpolatedQuaternions(1:end-1)))
```

```
def = 1×10

    9.0000    9.0000    9.0000    9.0000    9.0000    9.0000    9.0000    9.0000    9.0000    9.0
```

SLERP Minimizes Great Circle Path

The SLERP algorithm interpolates along a great circle path connecting two quaternions. This example shows how the SLERP algorithm minimizes the great circle path.

Define four quaternions:

1. `q0` - quaternion indicating no rotation from the global frame
2. `q179` - quaternion indicating a 179 degree rotation about the *z*-axis
3. `q180` - quaternion indicating a 180 degree rotation about the *z*-axis

4 `q181` - quaternion indicating a 181 degree rotation about the *z*-axis

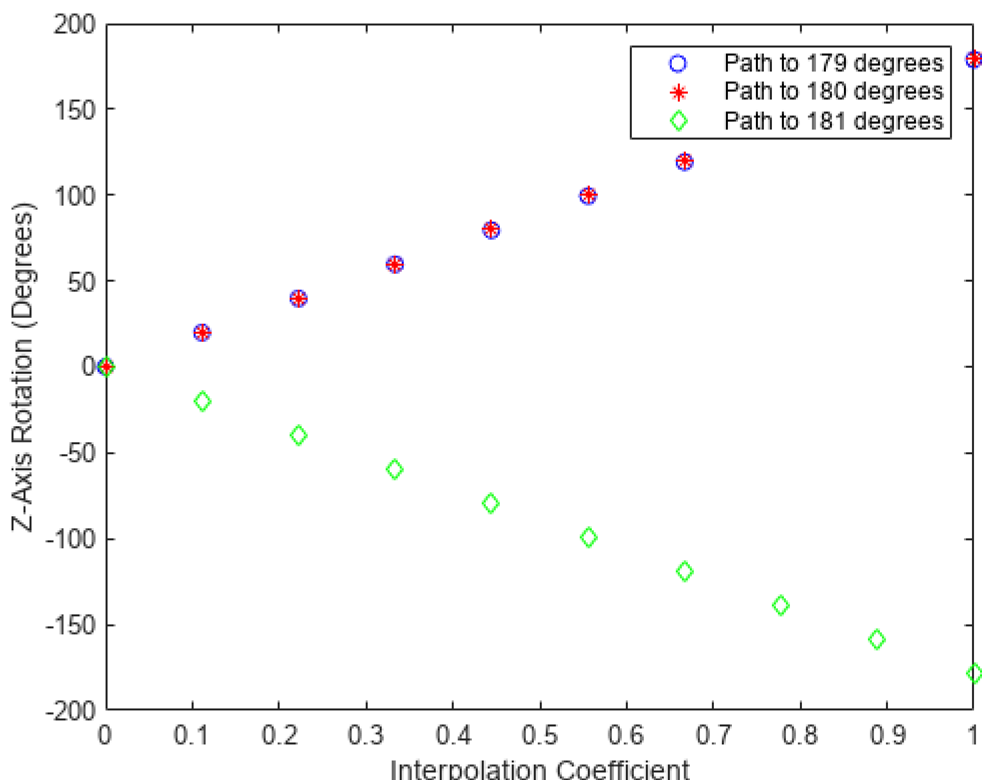
```
q0 = ones(1,'quaternion');

q179 = quaternion([179,0,0],'eulerd','ZYX','frame');

q180 = quaternion([180,0,0],'eulerd','ZYX','frame');

q181 = quaternion([181,0,0],'eulerd','ZYX','frame');
```

Use `slerp` to interpolate between `q0` and the three quaternion rotations. Specify that the paths are traveled in 10 steps.

```
T = linspace(0,1,10);

q179path = slerp(q0,q179,T);
q180path = slerp(q0,q180,T);
q181path = slerp(q0,q181,T);
```

Plot each path in terms of Euler angles in degrees.

```
q179pathEuler = eulerd(q179path,'ZYX','frame');
q180pathEuler = eulerd(q180path,'ZYX','frame');
q181pathEuler = eulerd(q181path,'ZYX','frame');

plot(T,q179pathEuler(:,1),'bo', ...
     T,q180pathEuler(:,1),'r*', ...
     T,q181pathEuler(:,1),'gd');
legend('Path to 179 degrees', ...
       'Path to 180 degrees', ...
       'Path to 181 degrees')
xlabel('Interpolation Coefficient')
ylabel('Z-Axis Rotation (Degrees)')
```



The path between `q0` and `q179` is clockwise to minimize the great circle distance. The path between `q0` and `q181` is counterclockwise to minimize the great circle distance. The path between `q0` and `q180` can be either clockwise or counterclockwise, depending on numerical rounding.

Show Interpolated Quaternions on Sphere

Create two quaternions.

```
q1 = quaternion([75,-20,-10],'eulerd','ZYX','frame');
q2 = quaternion([-45,20,30],'eulerd','ZYX','frame');
```

Define the interpolation coefficient.

```
T = 0:0.01:1;
```

Obtain the interpolated quaternions.

```
quats = slerp(q1,q2,T);
```

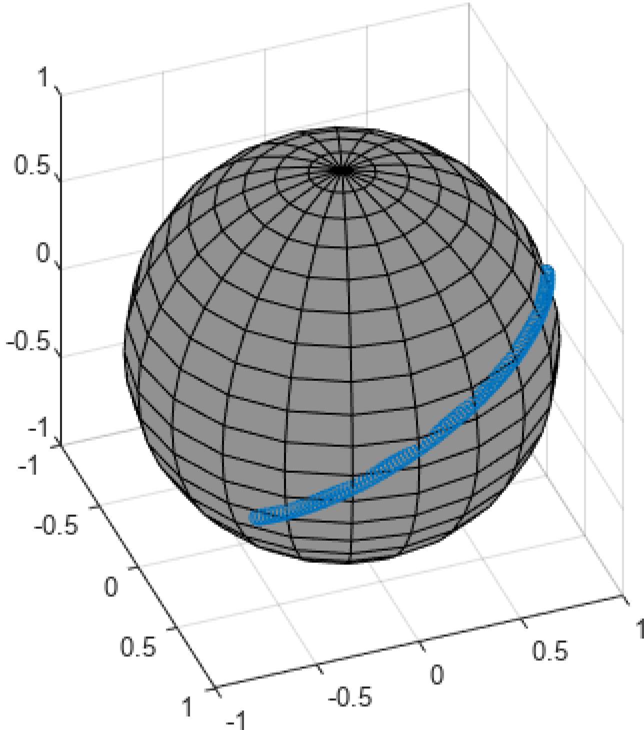
Obtain the corresponding rotate points.

```
pts = rotatepoint(quats,[1 0 0]);
```

Show the interpolated quaternions on a unit sphere.

```
figure
[X,Y,Z] = sphere;
```

```
surf(X,Y,Z,'FaceColor',[0.57 0.57 0.57])
hold on;

scatter3(pts(:,1),pts(:,2),pts(:,3))
view([69.23 36.60])
axis equal
```



Note that the interpolated quaternions follow the shorter path from `q1` to `q2`.

Input Arguments

q1 — Quaternion
scalar | vector | matrix | multidimensional array

Quaternion to interpolate, specified as a scalar, vector, matrix, or multidimensional array of quaternions.

`q1`, `q2`, and `T` must have compatible sizes. In the simplest cases, they can be the same size or any one can be a scalar. Two inputs have compatible sizes if, for every dimension, the dimension sizes of the inputs are either the same or one of them is 1.

Data Types: `quaternion`

q2 — Quaternion
scalar | vector | matrix | multidimensional array

Quaternion to interpolate, specified as a scalar, vector, matrix, or multidimensional array of quaternions.

`q1`, `q2`, and `T` must have compatible sizes. In the simplest cases, they can be the same size or any one can be a scalar. Two inputs have compatible sizes if, for every dimension, the dimension sizes of the inputs are either the same or one of the dimension sizes is 1.

Data Types: `quaternion`

`T` — Interpolation coefficient
scalar | vector | matrix | multidimensional array

Interpolation coefficient, specified as a scalar, vector, matrix, or multidimensional array of numbers with each element in the range [0,1].

`q1`, `q2`, and `T` must have compatible sizes. In the simplest cases, they can be the same size or any one can be a scalar. Two inputs have compatible sizes if, for every dimension, the dimension sizes of the inputs are either the same or one of the dimension sizes is 1.

Data Types: `single` | `double`

Output Arguments

`q0` — Interpolated quaternion
scalar | vector | matrix | multidimensional array

Interpolated quaternion, returned as a scalar, vector, matrix, or multidimensional array.

Data Types: `quaternion`

Algorithms

Quaternion **s**pherical **l**inear int**erp**olation (SLERP) is an extension of linear interpolation along a plane to spherical interpolation in three dimensions. The algorithm was first proposed in [1]. Given two quaternions, q_1 and q_2, SLERP interpolates a new quaternion, q_0, along the great circle that connects q_1 and q_2. The interpolation coefficient, T, determines how close the output quaternion is to either q_1 and q_2.

The SLERP algorithm can be described in terms of sinusoids:

$$q_0 = \frac{\sin((1-T)\theta)}{\sin(\theta)}q_1 + \frac{\sin(T\theta)}{\sin(\theta)}q_2$$

where q_1 and q_2 are normalized quaternions, and θ is half the angular distance between q_1 and q_2.

Version History

Introduced in R2018b

References

[1] Shoemake, Ken. "Animating Rotation with Quaternion Curves." *ACM SIGGRAPH Computer Graphics* Vol. 19, Issue 3, 1985, pp. 245–254.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

Functions
`dist` | `meanrot`

Objects
`quaternion`

tform2axang

Convert homogeneous transformation to axis-angle rotation

Syntax

```
axang = tform2axang(tform)
```

Description

`axang = tform2axang(tform)` converts the rotational component of a homogeneous transformation, `tform`, to an axis-angle rotation, `axang`. The translational components of `tform` are ignored. The input homogeneous transformation must be in the premultiply form for transformations.

Examples

Convert Homogeneous Transformation to Axis-Angle Rotation

```
tform = [1 0 0 0; 0 0 -1 0; 0 1 0 0; 0 0 0 1];
axang = tform2axang(tform)
```

```
axang = 1×4

    1.0000         0         0    1.5708
```

Input Arguments

`tform` — Homogeneous transformation
4-by-4-by-*n* matrix

Homogeneous transformation, specified by a 4-by-4-by-*n* matrix of *n* homogeneous transformations. The input homogeneous transformation must be in the premultiply form for transformations.

Example: `[0 0 1 0; 0 1 0 0; -1 0 0 0; 0 0 0 1]`

Output Arguments

axang — Rotation given in axis-angle form
n-by-4 matrix

Rotation given in axis-angle form, specified as an *n*-by-4 matrix of *n* axis-angle rotations. The first three elements of every row specify the rotation axes, and the last element defines the rotation angle (in radians).

Example: `[1 0 0 pi/2]`

Version History

Introduced in R2015a

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`axang2tform` | `se3`

Topics

"Coordinate Transformations in Robotics"

tform2eul

Extract Euler angles from homogeneous transformation

Syntax

```
eul = tform2eul(tform)
eul = tform2eul(tform, sequence)
[eul,eulAlt] = tform2eul( ___ )
```

Description

`eul = tform2eul(tform)` extracts the rotational component from a homogeneous transformation, `tform`, and returns it as Euler angles, `eul`. The translational components of `tform` are ignored. The input homogeneous transformation must be in the premultiply form for transformations. The default order for Euler angle rotations is `"ZYX"`.

`eul = tform2eul(tform, sequence)` extracts the Euler angles, `eul`, from a homogeneous transformation, `tform`, using the specified rotation sequence, `sequence`. The default order for Euler angle rotations is `"ZYX"`.

`[eul,eulAlt] = tform2eul( ___ )` also returns an alternate set of Euler angles that represents the same rotation `eulAlt`.

Examples

Extract Euler Angles from Homogeneous Transformation Matrix

```
tform = [1 0 0 0.5; 0 -1 0 5; 0 0 -1 -1.2; 0 0 0 1];
eulZYX = tform2eul(tform)
```

```
eulZYX = 1×3

         0         0    3.1416
```

Extract Euler Angles from Homogeneous Transformation Matrix Using ZYZ Rotation

```
tform = [1 0 0 0.5; 0 -1 0 5; 0 0 -1 -1.2; 0 0 0 1];
eulZYZ = tform2eul(tform,'ZYZ')
```

```
eulZYZ = 1×3

         0   -3.1416    3.1416
```

Input Arguments

`tform` — Homogeneous transformation
4-by-4-by-*n* matrix

Homogeneous transformation, specified by a 4-by-4-by-*n* matrix of *n* homogeneous transformations. The input homogeneous transformation must be in the premultiply form for transformations.

Example: `[0 0 1 0; 0 1 0 0; -1 0 0 0; 0 0 0 1]`

`sequence` — Axis-rotation sequence
`"ZYX"` (default) | `"ZYZ"` | `"ZXY"` | `"ZXZ"` | `"YXY"` | `"YZX"` | `"YXZ"` | `"YZY"` | `"XYX"` | `"XYZ"` | `"XZX"` | `"XZY"`

Axis-rotation sequence for the Euler angles, specified as one of these string scalars:

- `"ZYX"` (default)
- `"ZYZ"`
- `"ZXY"`
- `"ZXZ"`
- `"YXY"`
- `"YZX"`
- `"YXZ"`
- `"YZY"`
- `"XYX"`
- `"XYZ"`
- `"XZX"`
- `"XZY"`

Each character indicates the corresponding axis. For example, if the sequence is `"ZYX"`, then the three specified Euler angles are interpreted in order as a rotation around the *z*-axis, a rotation around the *y*-axis, and a rotation around the *x*-axis. When applying this rotation to a point, it will apply the axis rotations in the order *x*, then *y*, then *z*.

Data Types: `string` | `char`

Output Arguments

`eul` — Euler rotation angles
n-by-3 matrix

Euler rotation angles in radians, returned as an *n*-by-3 array of Euler rotation angles. Each row represents one Euler angle set.

Example: `[0 0 1.5708]`

`eulAlt` — Alternate Euler rotation angle solution
n-by-3 matrix

Alternate Euler rotation angle solution in radians, returned as an *n*-by-3 array of Euler rotation angles. Each row represents one Euler angle set.

Example: `[0 0 1.5708]`

Version History

Introduced in R2015a

R2020a: Alternate Euler angle output

`tform2eul` now optionally outputs an alternate set of Euler angles `eulAlt` that also represent the same rotation as the original output Euler angles `eul`. So if you use `eul` or `eulAlt` to rotate a point, the resulting point is the same.

R2023a: Additional Euler sequence support

`tform2eul` supports additional Euler sequences for the `sequences` argument. These are all the supported Euler sequences:

- `"ZYX"`
- `"ZYZ"`
- `"ZXY"`
- `"ZXZ"`
- `"YXY"`
- `"YZX"`
- `"YXZ"`
- `"YZY"`
- `"XYX"`
- `"XYZ"`
- `"XZX"`
- `"XZY"`

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`eul2tform` | `se2` | `se3`

Topics

"Coordinate Transformations in Robotics"

tform2quat

Extract quaternion from homogeneous transformation

Syntax

```
quat = tform2quat(tform)
```

Description

`quat = tform2quat(tform)` extracts the rotational component from a homogeneous transformation, `tform`, and returns it as a quaternion, `quat`. The translational components of `tform` are ignored. The input homogeneous transformation must be in the premultiply form for transformations.

Examples

Extract Quaternion from Homogeneous Transformation

```
tform = [1 0 0 0; 0 -1 0 0; 0 0 -1 0; 0 0 0 1];
quat = tform2quat(tform)
```

```
quat = 1×4

     0     1     0     0
```

Input Arguments

`tform` — Homogeneous transformation
4-by-4-by-*n* matrix

Homogeneous transformation, specified by a 4-by-4-by-*n* matrix of *n* homogeneous transformations. The input homogeneous transformation must be in the premultiply form for transformations.

Example: `[0 0 1 0; 0 1 0 0; -1 0 0 0; 0 0 0 1]`

Output Arguments

`quat` — Unit quaternion
n-by-4 matrix

Unit quaternion, returned as an *n*-by-4 matrix containing *n* quaternions. Each quaternion, one per row, is of the form $q = [w\ x\ y\ z]$, with *w* as the scalar number.

Example: `[0.7071 0.7071 0 0]`

Version History

Introduced in R2015a

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`quat2tform` | `se3` | `quaternion`

Topics

"Coordinate Transformations in Robotics"

tform2rotm

Extract rotation matrix from homogeneous transformation

Syntax

```
rotm = tform2rotm(tform)
```

Description

`rotm = tform2rotm(tform)` extracts the rotational component from a homogeneous transformation, `tform`, and returns it as an orthonormal rotation matrix, `rotm`. The translational components of `tform` are ignored. The input homogeneous transformation must be in the pre-multiply form for transformations. When using the rotation matrix, premultiply it with the coordinates to be rotated (as opposed to postmultiplying).

Examples

Convert Homogeneous Transformation to Rotation Matrix

```
tform = [1 0 0 0; 0 -1 0 0; 0 0 -1 0; 0 0 0 1];
rotm = tform2rotm(tform)
```

```
rotm = 3×3

     1     0     0
     0    -1     0
     0     0    -1
```

Input Arguments

`tform` — Homogeneous transformation

3-by-3-by-*n* array | 4-by-4-by-*n* array

Homogeneous transformation, specified as a 3-by-3-by-*n* array or 4-by-4-by-*n* array. *n* is the number of homogeneous transformations. The input homogeneous transformation must be in the premultiplied form for transformations.

2-D homogeneous transformation matrices are of the form:

$$T = \begin{bmatrix} r_{11} & r_{12} & t_1 \\ r_{21} & r_{22} & t_2 \\ 0 & 0 & 1 \end{bmatrix}$$

3-D homogeneous transformation matrices are of the form:

$$T = \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_1 \\ r_{21} & r_{22} & r_{23} & t_2 \\ r_{31} & r_{32} & r_{33} & t_3 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Example: `[0 0 1 0; 0 1 0 0; -1 0 0 0; 0 0 0 1]`

Output Arguments

`rotm` — Rotation matrix

2-by-2-by-*n* array | 3-by-3-by-*n* array

Rotation matrix, returned as a 2-by-2-*n* array or 3-by-3-by-*n* array containing *n* rotation matrices. Each rotation matrix in the array has either a size of 2-by-2 or 3-by-3 and is orthonormal. When using the rotation matrix, premultiply it with the coordinates to be rotated (as opposed to postmultiplying).

2-D rotation matrices are of the form:

$$rotation = \begin{bmatrix} r_{11} & r_{12} \\ r_{21} & r_{22} \end{bmatrix}$$

3-D rotation matrices are of the form:

$$rotation = \begin{bmatrix} r_{11} & r_{12} & r_{13} \\ r_{11} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{bmatrix}$$

Example: `[0 0 1; 0 1 0; -1 0 0]`

More About

Homogeneous Transformation Matrices

Homogeneous transformation matrices consist of both an orthogonal rotation and a translation.

2-D Transformations

2-D transformations have a rotation θ about the z-axis:

$$R_z(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}$$

, and a translation along the x and y axis:

$$t = \begin{bmatrix} x \\ y \end{bmatrix}$$

, resulting in the 2-D transformation matrix of the form:

$$T = \begin{bmatrix} R & t \\ 0_{1 \times 2} & 1 \end{bmatrix} = \begin{bmatrix} I_2 & t \\ 0_{1 \times 2} & 1 \end{bmatrix} \cdot \begin{bmatrix} R & 0 \\ 0_{1 \times 2} & 1 \end{bmatrix}$$

3-D Transformations

3-D transformations contain information about three rotations about the x-, y-, and z-axes:

$$R_x(\phi) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\phi & \sin\phi \\ 0 & -\sin\phi & \cos\phi \end{bmatrix}, R_y(\psi) = \begin{bmatrix} \cos\psi & 0 & \sin\psi \\ 0 & 1 & 0 \\ -\sin\psi & 0 & \cos\psi \end{bmatrix}, R_z(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

and after multiplying become the rotation about the xyz-axes:

$$R_{xyz} = R_x(\phi)R_y(\psi)R_z(\theta) =$$
$$\begin{bmatrix} \cos\phi\cos\psi\cos\theta - \sin\phi\sin\theta & -\cos\phi\cos\psi\sin\theta - \sin\phi\cos\theta & \cos\phi\sin\psi \\ \sin\phi\cos\psi\cos\theta + \cos\phi\sin\theta & -\sin\phi\cos\psi\sin\theta + \cos\phi\cos\theta & \sin\phi\sin\psi \\ -\sin\psi\cos\theta & \sin\psi\sin\theta & \cos\psi \end{bmatrix}$$

and a translation along the *x*-, *y*-, and *z*-axis:

$$t = \begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

, resulting in the 3-D transformation matrix of the form:

$$T = \begin{bmatrix} R & t \\ 0_{1x3} & 1 \end{bmatrix} = \begin{bmatrix} I_3 & t \\ 0_{1x3} & 1 \end{bmatrix} \cdot \begin{bmatrix} R & 0 \\ 0_{1x3} & 1 \end{bmatrix}$$

Version History

Introduced in R2015a

R2023a: `tform2rotm` Supports 2-D Homogeneous Transformation Matrices

The `tform` argument now accepts 2-D homogeneous transformation matrices as a 3-by-3-by-*n* array and `tform2rotm` outputs 2-D rotation matrices 2-by-2-by-*n* array.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`rotm2tform` | `se2` | `se3` | `so2` | `so3`

Topics

"Coordinate Transformations in Robotics"

tform2trvec

Extract translation vector from homogeneous transformation

Syntax

```
trvec = tform2trvec(tform)
```

Description

`trvec = tform2trvec(tform)` extracts the Cartesian representation of the translation vector `trvec` from the homogeneous transformation `tform`. The rotational components of `tform` are ignored. The input homogeneous transformation must be in the premultiplied form for transformations.

Examples

Extract Translation Vector from Homogeneous Transformation

```
tform = [1 0 0 0.5; 0 -1 0 5; 0 0 -1 -1.2; 0 0 0 1];
trvec = tform2trvec(tform)
```

```
trvec = 1×3

    0.5000    5.0000   -1.2000
```

Input Arguments

`tform` — Homogeneous transformation
3-by-3-by-*n* array | 4-by-4-by-*n* array

Homogeneous transformation, specified as a 3-by-3-by-*n* array or 4-by-4-by-*n* array. *n* is the number of homogeneous transformations. The input homogeneous transformation must be in the premultiplied form for transformations.

2-D homogeneous transformation matrices are of the form:

$$T = \begin{bmatrix} r_{11} & r_{12} & t_1 \\ r_{21} & r_{22} & t_2 \\ 0 & 0 & 1 \end{bmatrix}$$

3-D homogeneous transformation matrices are of the form:

$$T = \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_1 \\ r_{21} & r_{22} & r_{23} & t_2 \\ r_{31} & r_{32} & r_{33} & t_3 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Example: `[0 0 1 0; 0 1 0 0; -1 0 0 0; 0 0 0 1]`

Output Arguments

`trvec` — Cartesian representation of translation vector

n-by-2 matrix | *n*-by-3 matrix

Cartesian representation of a translation vector, returned as an *n*-by-2 matrix if `tform` is a 3-by-3-by-*n* array and an *n*-by-3 matrix if `tform` is a 4-by-4-by-*n* array. *n* is the number of translation vectors. Each vector is of the form [*x* *y*] or [*x* *y* *z*].

Example: `[0.5 6 100]`

More About

Homogeneous Transformation Matrices

Homogeneous transformation matrices consist of both an orthogonal rotation and a translation.

2-D Transformations

2-D transformations have a rotation θ about the z-axis:

$$R_z(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}$$

, and a translation along the *x* and *y* axis:

$$t = \begin{bmatrix} x \\ y \end{bmatrix}$$

, resulting in the 2-D transformation matrix of the form:

$$T = \begin{bmatrix} R & t \\ 0_{1\times 2} & 1 \end{bmatrix} = \begin{bmatrix} I_2 & t \\ 0_{1\times 2} & 1 \end{bmatrix} \cdot \begin{bmatrix} R & 0 \\ 0_{1\times 2} & 1 \end{bmatrix}$$

3-D Transformations

3-D transformations contain information about three rotations about the x-, y-, and z-axes:

$$R_x(\phi) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\phi & \sin\phi \\ 0 & -\sin\phi & \cos\phi \end{bmatrix}, R_y(\psi) = \begin{bmatrix} \cos\psi & 0 & \sin\psi \\ 0 & 1 & 0 \\ -\sin\psi & 0 & \cos\psi \end{bmatrix}, R_z(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

and after multiplying become the rotation about the xyz-axes:

$$R_{xyz} = R_x(\phi)R_y(\psi)R_z(\theta) =$$
$$\begin{bmatrix} \cos\phi\cos\psi\cos\theta - \sin\phi\sin\theta & -\cos\phi\cos\psi\sin\theta - \sin\phi\cos\theta & \cos\phi\sin\psi \\ \sin\phi\cos\psi\cos\theta + \cos\phi\sin\theta & -\sin\phi\cos\psi\sin\theta + \cos\phi\cos\theta & \sin\phi\sin\psi \\ -\sin\psi\cos\theta & \sin\psi\sin\theta & \cos\psi \end{bmatrix}$$

and a translation along the *x*-, *y*-, and *z*-axis:

$$t = \begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

, resulting in the 3-D transformation matrix of the form:

$$T = \begin{bmatrix} R & t \\ 0_{1x3} & 1 \end{bmatrix} = \begin{bmatrix} I_3 & t \\ 0_{1x3} & 1 \end{bmatrix} \cdot \begin{bmatrix} R & 0 \\ 0_{1x3} & 1 \end{bmatrix}$$

Version History

Introduced in R2015a

R2023a: `tform2trvec` Supports 2-D Homogeneous Transformation Matrices

The `tform` argument now accepts 2-D homogeneous transformation matrices as a 3-by-3-by-*n* array and `tform2trvec` outputs a *n*-by-2 matrix of 2-D translation vectors.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`trvec2tform` | `se2` | `se3`

Topics

"Coordinate Transformations in Robotics"

times, .*

Element-wise quaternion multiplication

Syntax

```
quatC = A.*B
```

Description

`quatC = A.*B` returns the element-by-element quaternion multiplication of quaternion arrays.

You can use quaternion multiplication to compose rotation operators:

- To compose a sequence of frame rotations, multiply the quaternions in the same order as the desired sequence of rotations. For example, to apply a p quaternion followed by a q quaternion, multiply in the order pq. The rotation operator becomes $(pq)^{*}v(pq)$, where v represents the object to rotate in quaternion form. * represents conjugation.
- To compose a sequence of point rotations, multiply the quaternions in the reverse order of the desired sequence of rotations. For example, to apply a p quaternion followed by a q quaternion, multiply in the reverse order, qp. The rotation operator becomes $(qp)v(qp)^{*}$.

Examples

Multiply Two Quaternion Vectors

Create two vectors, `A` and `B`, and multiply them element by element.

```
A = quaternion([1:4;5:8]);
B = A;
C = A.*B
```

```
C = 2x1 quaternion array
     -28 +   4i +   6j +   8k
    -124 +  60i +  70j +  80k
```

Multiply Two Quaternion Arrays

Create two 3-by-3 arrays, `A` and `B`, and multiply them element by element.

```
A = reshape(quaternion(randn(9,4)),3,3);
B = reshape(quaternion(randn(9,4)),3,3);
C = A.*B
```

```
C = 3x3 quaternion array
     0.60169 +  2.4332i -  2.5844j + 0.51646k    -0.49513 +  1.1722i +  4.4401j -   1.217k      2
     -4.2329 +  2.4547i +  3.7768j + 0.77484k    -0.65232 - 0.43112i -  1.4645j - 0.90073k     -1
```

```
    -4.4159 +  2.1926i +  1.9037j -  4.0303k     -2.0232 +  0.4205i - 0.17288j +  3.8529k     -2
```

Note that quaternion multiplication is not commutative:

```
isequal(C,B.*A)
```

```
ans = logical
   0
```

Multiply Quaternion Row and Column Vectors

Create a row vector a and a column vector b, then multiply them. The 1-by-3 row vector and 4-by-1 column vector combine to produce a 4-by-3 matrix with all combinations of elements multiplied.

```
a = [zeros('quaternion'),ones('quaternion'),quaternion(randn(1,4))]
```

```
a = 1x3 quaternion array
           0 +       0i +       0j +       0k             1 +       0i +       0j +       0k     0
```

```
b = quaternion(randn(4,4))
```

```
b = 4x1 quaternion array
      0.31877 +   3.5784i +   0.7254j -  0.12414k
      -1.3077 +   2.7694i - 0.063055j +   1.4897k
     -0.43359 -   1.3499i +  0.71474j +    1.409k
      0.34262 +   3.0349i -  0.20497j +   1.4172k
```

```
a.*b
```

```
ans = 4x3 quaternion array
            0 +       0i +       0j +       0k      0.31877 +   3.5784i +   0.7254j -  0.12414
            0 +       0i +       0j +       0k      -1.3077 +   2.7694i - 0.063055j +   1.4897
            0 +       0i +       0j +       0k     -0.43359 -   1.3499i +  0.71474j +    1.409
            0 +       0i +       0j +       0k      0.34262 +   3.0349i -  0.20497j +   1.4172
```

Input Arguments

A — Array to multiply
scalar | vector | matrix | multidimensional array

Array to multiply, specified as a quaternion, an array of quaternions, a real scalar, or an array of real numbers.

A and B must have compatible sizes. In the simplest cases, they can be the same size or one can be a scalar. Two inputs have compatible sizes if, for every dimension, the dimension sizes of the inputs are the same or one of them is 1.

Data Types: quaternion | single | double

B — Array to multiply
scalar | vector | matrix | multidimensional array

Array to multiply, specified as a quaternion, an array of quaternions, a real scalar, or an array of real numbers.

A and B must have compatible sizes. In the simplest cases, they can be the same size or one can be a scalar. Two inputs have compatible sizes if, for every dimension, the dimension sizes of the inputs are the same or one of them is 1.

Data Types: quaternion | single | double

Output Arguments

quatC — Quaternion product
scalar | vector | matrix | multidimensional array

Quaternion product, returned as a scalar, vector, matrix, or multidimensional array.

Data Types: quaternion

Algorithms

Quaternion Multiplication by a Real Scalar

Given a quaternion,

$$q = a_q + b_q\mathrm{i} + c_q\mathrm{j} + d_q\mathrm{k},$$

the product of q and a real scalar β is

$$\beta q = \beta a_q + \beta b_q\mathrm{i} + \beta c_q\mathrm{j} + \beta d_q\mathrm{k}$$

Quaternion Multiplication by a Quaternion Scalar

The definition of the basis elements for quaternions,

$$\mathrm{i}^2 = \mathrm{j}^2 = \mathrm{k}^2 = \mathrm{ijk} = -1\,,$$

can be expanded to populate a table summarizing quaternion basis element multiplication:

	1	**i**	**j**	**k**
1	1	i	j	k
i	i	−1	k	−j
j	j	−k	−1	i
k	k	j	−i	−1

When reading the table, the rows are read first, for example: ij = k and ji = −k.

Given two quaternions, $q = a_q + b_q\mathrm{i} + c_q\mathrm{j} + d_q\mathrm{k}$, and $p = a_p + b_p\mathrm{i} + c_p\mathrm{j} + d_p\mathrm{k}$, the multiplication can be expanded as:

$$
\begin{aligned}
z = pq = \left(a_{\mathrm{p}} + b_{\mathrm{p}}\mathrm{i} + c_{\mathrm{p}}\mathrm{j} + d_{\mathrm{p}}\mathrm{k}\right)\left(a_{\mathrm{q}} + b_{\mathrm{q}}\mathrm{i} + c_{\mathrm{q}}\mathrm{j} + d_{\mathrm{q}}\mathrm{k}\right) \\
= a_{\mathrm{p}}a_{\mathrm{q}} + a_{\mathrm{p}}b_{\mathrm{q}}\mathrm{i} + a_{\mathrm{p}}c_{\mathrm{q}}\mathrm{j} + a_{\mathrm{p}}d_{\mathrm{q}}\mathrm{k} \\
+ b_{\mathrm{p}}a_{\mathrm{q}}\mathrm{i} + b_{\mathrm{p}}b_{\mathrm{q}}\mathrm{i}^2 + b_{\mathrm{p}}c_{\mathrm{q}}\mathrm{ij} + b_{\mathrm{p}}d_{\mathrm{q}}\mathrm{ik} \\
+ c_{\mathrm{p}}a_{\mathrm{q}}\mathrm{j} + c_{\mathrm{p}}b_{\mathrm{q}}\mathrm{ji} + c_{\mathrm{p}}c_{\mathrm{q}}\mathrm{j}^2 + c_{\mathrm{p}}d_{\mathrm{q}}\mathrm{jk} \\
+ d_{\mathrm{p}}a_{\mathrm{q}}k + d_{\mathrm{p}}b_{\mathrm{q}}\mathrm{ki} + d_{\mathrm{p}}c_{\mathrm{q}}\mathrm{kj} + d_{\mathrm{p}}d_{\mathrm{q}}\mathrm{k}^2
\end{aligned}
$$

You can simplify the equation using the quaternion multiplication table.

$$
\begin{aligned}
z = pq = a_{\mathrm{p}}a_{\mathrm{q}} + a_{\mathrm{p}}b_{\mathrm{q}}\mathrm{i} + a_{\mathrm{p}}c_{\mathrm{q}}\mathrm{j} + a_{\mathrm{p}}d_{\mathrm{q}}\mathrm{k} \\
+ b_{\mathrm{p}}a_{\mathrm{q}}\mathrm{i} - b_{\mathrm{p}}b_{\mathrm{q}} + b_{\mathrm{p}}c_{\mathrm{q}}\mathrm{k} - b_{\mathrm{p}}d_{\mathrm{q}}\mathrm{j} \\
+ c_{\mathrm{p}}a_{\mathrm{q}}\mathrm{j} - c_{\mathrm{p}}b_{\mathrm{q}}\mathrm{k} - c_{\mathrm{p}}c_{\mathrm{q}} + c_{\mathrm{p}}d_{\mathrm{q}}\mathrm{i} \\
+ d_{\mathrm{p}}a_{\mathrm{q}}k + d_{\mathrm{p}}b_{\mathrm{q}}\mathrm{j} - d_{\mathrm{p}}c_{\mathrm{q}}\mathrm{i} - d_{\mathrm{p}}d_{\mathrm{q}}
\end{aligned}
$$

Version History

Introduced in R2018a

References

[1] Kuipers, Jack B. *Quaternions and Rotation Sequences: A Primer with Applications to Orbits, Aerospace, and Virtual Reality*. Princeton, NJ: Princeton University Press, 2007.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

Functions
`prod` | `mtimes, *`

Objects
`quaternion`

transformScan

Transform laser scan based on relative pose

Syntax

`transScan = transformScan(scan,relPose)`

`[transRanges,transAngles] = transformScan(ranges,angles,relPose)`

Description

`transScan = transformScan(scan,relPose)` transforms the laser scan specified in `scan` by using the specified relative pose, `relPose`.

`[transRanges,transAngles] = transformScan(ranges,angles,relPose)` transforms the laser scan specified in `ranges` and `angles` by using the specified relative pose, `relPose`.

Examples

Transform Laser Scans

Create a `lidarScan` object. Specify the ranges and angles as vectors.

```
refRanges = 5*ones(1,300);
refAngles = linspace(-pi/2,pi/2,300);
refScan = lidarScan(refRanges,refAngles);
```

Translate the laser scan by an `[x y]` offset of `(0.5,0.2)`.

```
transformedScan = transformScan(refScan,[0.5 0.2 0]);
```

Rotate the laser scan by 20 degrees.

```
rotateScan = transformScan(refScan,[0,0,deg2rad(20)]);
```

Input Arguments

`scan` — Lidar scan readings
`lidarScan` object

Lidar scan readings, specified as a `lidarScan` object.

`ranges` — Range values from scan data
vector

Range values from scan data, specified as a vector in meters. These range values are distances from a sensor at specified `angles`. The vector must be the same length as the corresponding `angles` vector.

angles — Angle values from scan data
vector

Angle values from scan data, specified as a vector in radians. These angle values are the specific angles of the specified `ranges`. The vector must be the same length as the corresponding `ranges` vector.

relPose — Relative pose of current scan
`[x y theta]`

Relative pose of current scan, specified as `[x y theta]`, where `[x y]` is the translation in meters and `theta` is the rotation in radians.

Output Arguments

transScan — Transformed lidar scan readings
`lidarScan` object

Transformed lidar scan readings, specified as a `lidarScan` object.

transRanges — Range values of transformed scan
vector

Range values of transformed scan, returned as a vector in meters. These range values are distances from a sensor at specified `transAngles`. The vector is the same length as the corresponding `transAngles` vector.

transAngles — Angle values from scan data
vector

Angle values of transformed scan, returned as a vector in radians. These angle values are the specific angles of the specified `transRanges`. The vector is the same length as the corresponding `ranges` vector.

Version History

Introduced in R2017a

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

transformtraj

Generate trajectories between two transformations

Syntax

```
[tforms,vel,acc] = transformtraj(T0,TF,tInterval,tSamples)
[tforms,vel,acc] = transformtraj(T0,TF,tInterval,tSamples,Name=Value)
```

Description

`[tforms,vel,acc] = transformtraj(T0,TF,tInterval,tSamples)` generates a trajectory that interpolates between two homogeneous transformations, `T0` and `TF`, with points based on the time interval and given time samples.

`[tforms,vel,acc] = transformtraj(T0,TF,tInterval,tSamples,Name=Value)` specifies additional parameters using name-value arguments.

Examples

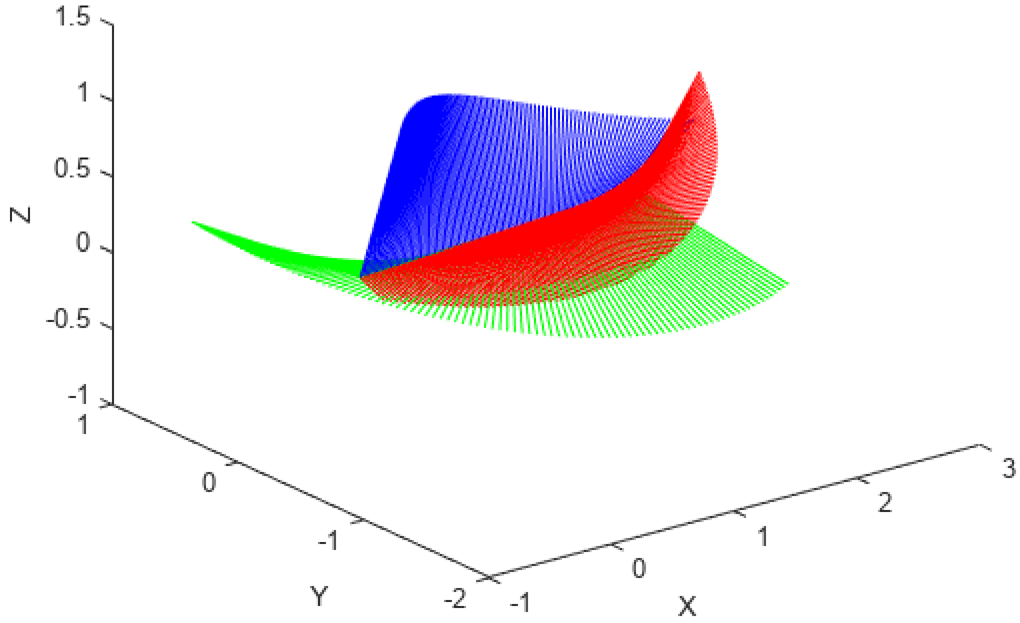
Interpolate Between Homogenous Transformations

Build transformations from two orientations and positions. Specify the time interval and vector of times for interpolating.

```
t0 = axang2tform([0 1 1 pi/4])*trvec2tform([0 0 0]);
tF = axang2tform([1 0 1 6*pi/5])*trvec2tform([1 1 1]);
tInterval = [0 1];
tvec = 0:0.01:1;
```

Interpolate between the points. Plot the trajectory using `plotTransforms`. Convert the transformations to quaternion rotations and linear transitions. The figure shows all the intermediate transformations of the coordinate frame.

```
[tfInterp, v1, a1] = transformtraj(t0,tF,tInterval,tvec);

rotations = tform2quat(tfInterp);
translations = tform2trvec(tfInterp);

plotTransforms(translations,rotations)
xlabel('X')
ylabel('Y')
zlabel('Z')
```



Input Arguments

T0 — Initial transformation

4-by-4 homogeneous transformation | se3 object

Initial transformation, specified as a 4-by-4 homogeneous transformation or a scalar se3 object. The function generates a trajectory that starts at the initial transformation, T0, and goes to the final transformation, TF.

T0 and TF must be of the same type. For example, if T0 is a scalar se3 object, then TF must be a scalar se3 object.

Data Types: single | double

TF — Final transformation

4-by-4 homogeneous transformation | se3 object

Final transformation, specified as a 4-by-4 homogeneous transformation or a scalar se3 object. The function generates a trajectory that starts at the initial transformation, T0, and goes to the final transformation, TF.

T0 and TF must be of the same type. For example, if T0 is a scalar se3 object, then TF must be a scalar se3 object.

Data Types: single | double

`tInterval` — Start and end times for trajectory
two-element vector

Start and end times for the trajectory, specified as a two-element vector in seconds.

Example: `[0 10]`

Data Types: `single` | `double`

`tSamples` — Time samples for trajectory
m-element vector

Time samples for the trajectory, specified as an *m*-element vector in seconds.

Example: `0:0.01:10`

Data Types: `single` | `double`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `TimeScaling=[0 1 1; 0 1 0; 0 0 0]`

`TimeScaling` — Time scaling vector and first two derivatives
3-by-*m* vector

Time scaling vector and the first two derivatives, specified as a 3-by-*m* vector, where *m* is the length of `tSamples`. By default, the time scaling is a linear time scaling between the time points in `tInterval`.

For a nonlinear time scaling, specify the values of the time points as position time scaling in the first row. The second and third rows are the first and second derivative of the first row, 1/s and $1/s^2$, respectively. For example, to follow the path with a linear velocity to the halfway point, and then jump to the end, the time-scaling would be:

```
s(1,:) = [0 0.25 0.5 1 1 1] % Position time scaling, s(t)
s(2,:) = [1    1   1 0 0 0] % Velocity time scaling, ds/dt
s(3,:) = [0    0   0 0 0 0] % Acceleration time scaling, d^2s/dt^2
```

All elements in the first row must be in the range [0, 1].

Data Types: `single` | `double`

Output Arguments

`tforms` — Transformation trajectory
4-by-4-by-*m* homogeneous transformation matrix array | *m*-element array of `se3` objects

Transformation trajectory, returned as a 4-by-4-by-*m* homogeneous transformation matrix array or an *m*-element array of `se3` objects. *m* is the number of points in `tSamples`.

`vel` — Transformation velocities
6-by-*m* matrix

Transformation velocities, returned as a 6-by-*m* matrix in m/s, where *m* is the number of points in `tSamples`. The first three elements are the angular velocities, and the second three elements are the velocities in time.

`acc` — Transformation accelerations
6-by-*m* matrix

Transformation accelerations, returned as a 6-by-*m* matrix in m/s^2, where *m* is the number of points in `tSamples`. The first three elements are the angular accelerations, and the second three elements are the accelerations in time.

Version History

Introduced in R2019a

R2023a: `transformtraj` supports SE(3) transformation object inputs

The `T0` and `TF` arguments of `transformtraj` now accept `se3` objects and the resulting output value for the `tforms` argument is an *m*-element array of `se3` objects. *m* is the number of points in `tSamples`.

References

[1] Lynch, Kevin M., and Frank C. Park. *Modern Robotics: Mechanics, Planning, and Control*. Cambridge University Press, 2017.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`bsplinepolytraj` | `contopptraj` | `cubicpolytraj` | `quinticpolytraj` | `rottraj` | `trapveltraj`

transpose, .'

Transpose a quaternion array

Syntax

```
Y = quat.'
```

Description

`Y = quat.'` returns the non-conjugate transpose of the quaternion array, `quat`.

Examples

Vector Transpose

Create a vector of quaternions and compute its nonconjugate transpose.

```
quat = quaternion(randn(4,4))
```

```
quat = 4x1 quaternion array
      0.53767 +  0.31877i +   3.5784j +   0.7254k
       1.8339 -   1.3077i +   2.7694j - 0.063055k
      -2.2588 -  0.43359i -   1.3499j +  0.71474k
      0.86217 +  0.34262i +   3.0349j -  0.20497k
```

```
quatTransposed = quat.'
```

```
quatTransposed = 1x4 quaternion array
      0.53767 +  0.31877i +   3.5784j +   0.7254k       1.8339 -   1.3077i +   2.7694j - 0.06305
```

Matrix Transpose

Create a matrix of quaternions and compute its nonconjugate transpose.

```
quat = [quaternion(randn(2,4)),quaternion(randn(2,4))]
```

```
quat = 2x2 quaternion array
      0.53767 -   2.2588i +  0.31877j -  0.43359k       3.5784 -   1.3499i +   0.7254j +  0.7147
       1.8339 +  0.86217i -   1.3077j +  0.34262k       2.7694 +   3.0349i - 0.063055j -  0.2049
```

```
quatTransposed = quat.'
```

```
quatTransposed = 2x2 quaternion array
      0.53767 -   2.2588i +  0.31877j -  0.43359k       1.8339 +  0.86217i -   1.3077j +  0.3426
       3.5784 -   1.3499i +   0.7254j +  0.71474k       2.7694 +   3.0349i - 0.063055j -  0.2049
```

Input Arguments

quat — Quaternion array to transpose
vector | matrix

Quaternion array to transpose, specified as a vector or matrix of quaternions. `transpose` is defined for 1-D and 2-D arrays. For higher-order arrays, use `permute`.

Data Types: `quaternion`

Output Arguments

Y — Transposed quaternion array
vector | matrix

Transposed quaternion array, returned as an *N*-by-*M* array, where `quat` was specified as an *M*-by-*N* array.

Version History

Introduced in R2018a

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

Functions
`ctranspose`, `'`

Objects
`quaternion`

trapveltraj

Generate trajectories with trapezoidal velocity profiles

Syntax

```
[q,qd,qdd,tSamples,pp] = trapveltraj(wayPoints,numSamples)
[q,qd,qdd,tSamples,pp] = trapveltraj(wayPoints,numSamples,Name,Value)
```

Description

`[q,qd,qdd,tSamples,pp] = trapveltraj(wayPoints,numSamples)` generates a trajectory through a given set of input waypoints that follow a trapezoidal velocity profile. The function outputs positions, velocities, and accelerations at the given time samples, `tSamples`, based on the specified number of samples, `numSamples`. The function also returns the piecewise polynomial `pp` form of the polynomial trajectory with respect to time.

`[q,qd,qdd,tSamples,pp] = trapveltraj(wayPoints,numSamples,Name,Value)` specifies additional parameters using `Name,Value` pair arguments.

Examples

Compute Trapezoidal Velocity Trajectory for 2-D Planar Motion

Use the `trapveltraj` function with a given set of 2-D *xy* waypoints.

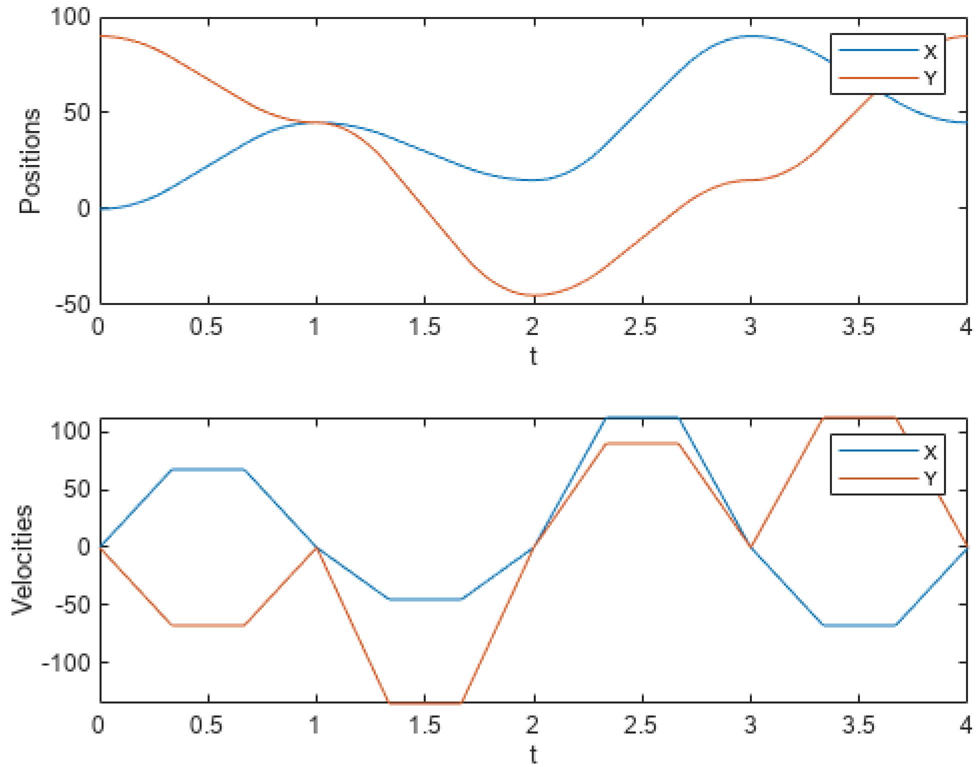
```
wpts = [0 45 15 90 45; 90 45 -45 15 90];
```

Compute the trajectory for a given number of samples (501). The function outputs the trajectory positions (`q`), velocity (`qd`), acceleration (`qdd`), time vector (`tvec`), and polynomial coefficients (`pp`) of the polynomial that achieves the waypoints using trapezoidal velocities.

```
[q,qd,qdd,tvec,pp] = trapveltraj(wpts,501);
```

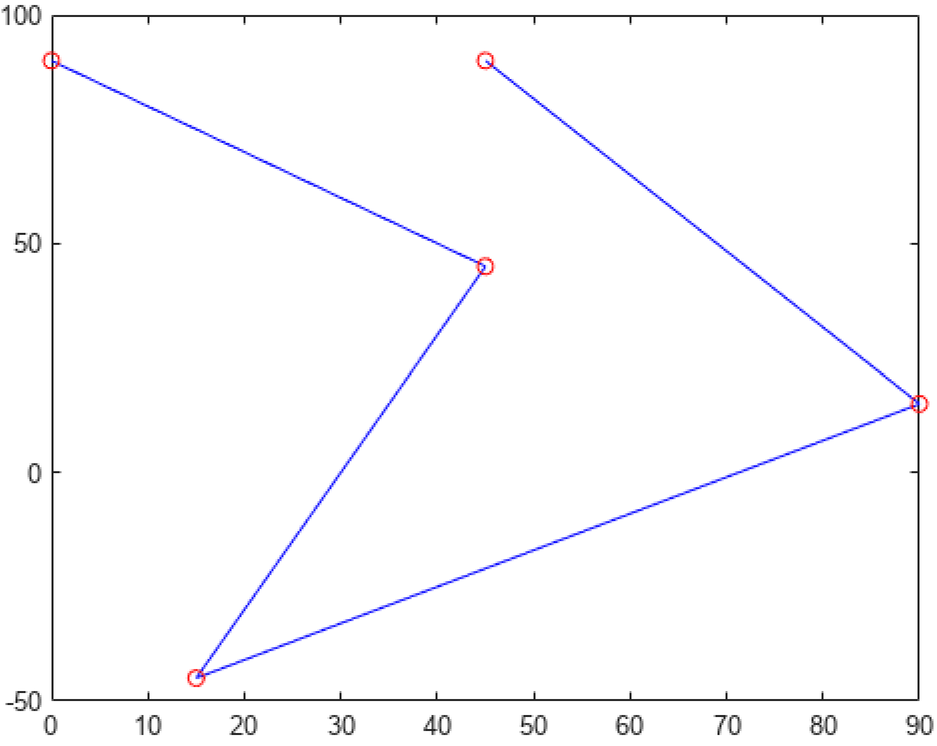
Plot the trajectories for the *x*- and *y*-positions and the trapezoidal velocity profile between each waypoint.

```
subplot(2,1,1)
plot(tvec, q)
xlabel('t')
ylabel('Positions')
legend('X','Y')
subplot(2,1,2)
plot(tvec, qd)
xlabel('t')
ylabel('Velocities')
legend('X','Y')
```



You can also verify the actual positions in the 2-D plane. Plot the separate rows of the `q` vector and the waypoints as *x*- and *y*-positions.

```
figure
plot(q(1,:),q(2,:),'-b',wpts(1,:),wpts(2,:),'or')
```



Input Arguments

`wayPoints` — Waypoints for trajectory
n-by-*p* matrix

Points for waypoints of trajectory, specified as an *n*-by-*p* matrix, where *n* is the dimension of the trajectory and *p* is the number of waypoints.

Example: `[1 4 4 3 -2 0; 0 1 2 4 3 1]`

Data Types: `single` | `double`

`numSamples` — Number of samples in output trajectory
positive integer

Number of samples in output trajectory, specified as a positive integer.

Data Types: `single` | `double`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Note Due to the nature of the trapezoidal velocity profile, you can only set at most two of the following parameters.

Example: `'PeakVelocity',5`

`PeakVelocity` — Peak velocity of the velocity profile
scalar | *n*-element vector | *n*-by-(*p*–1) matrix

Peak velocity of the profile segment, specified as the comma-separated pair consisting of `'PeakVelocity'` and a scalar, vector, or matrix. This peak velocity is the highest velocity achieved during the trapezoidal velocity profile.

A scalar value is applied to all elements of the trajectory and between all waypoints. An *n*-element vector is applied to each element of the trajectory between all waypoints. An *n*-by-(*p*–1) matrix is applied to each element of the trajectory for each waypoint.

Data Types: `single` | `double`

`Acceleration` — Acceleration of velocity profile
scalar | *n*-element vector | *n*-by-(*p*–1) matrix

Acceleration of the velocity profile, specified as the comma-separated pair consisting of `'Acceleration'` and a scalar, vector, or matrix. This acceleration defines the constant acceleration from zero velocity to the `PeakVelocity` value.

A scalar value is applied to all elements of the trajectory and between all waypoints. An *n*-element vector is applied to each element of the trajectory between all waypoints. An *n*-by-(*p*–1) matrix is applied to each element of the trajectory for each waypoint.

Data Types: `single` | `double`

`EndTime` — Duration of each trajectory segment
scalar | *n*-element vector | *n*-by-(*p*–1) matrix

Duration of each of the *p*–1 trajectory segments, specified as the comma-separated pair consisting of `'EndTime'` and a scalar, vector, or matrix.

A scalar value is applied to all elements of the trajectory and between all waypoints. An *n*-element vector is applied to each element of the trajectory between all waypoints. An *n*-by-(*p*–1) matrix is applied to each element of the trajectory for each waypoint.

Data Types: `single` | `double`

`AccelTime` — Duration of acceleration phase of velocity profile
scalar | *n*-element vector | *n*-by-(*p*–1) matrix

Duration of acceleration phase of velocity profile, specified as the comma-separated pair consisting of `'AccelTime'` and a scalar, vector, or matrix.

A scalar value is applied to all elements of the trajectory and between all waypoints. An *n*-element vector is applied to each element of the trajectory between all waypoints. An *n*-by-(*p*–1) matrix is applied to each element of the trajectory for each waypoint.

Data Types: `single` | `double`

Output Arguments

q — Positions of trajectory
n-by-*m* matrix

Positions of the trajectory at the given time samples in `tSamples`, returned as *n*-by-*m* matrix, where *n* is the dimension of the trajectory, and *m* is equal to `numSamples`.

Data Types: `single` | `double`

qd — Velocities of trajectory
n-by-*m* matrix

Velocities of the trajectory at the given time samples in `tSamples`, returned as *n*-by-*m* matrix, where *n* is the dimension of the trajectory, and *m* is equal to `numSamples`.

Data Types: `single` | `double`

qdd — Accelerations of trajectory
n-by-*m* matrix

Accelerations of the trajectory at the given time samples in `tSamples`, returned as *n*-by-*m* matrix, where *n* is the dimension of the trajectory, and *m* is equal to `numSamples`.

Data Types: `single` | `double`

`tSamples` — Time samples for trajectory
m-element vector

Time samples for the trajectory, returned as an *m*-element vector. The output position, `q`, velocity, `qd`, and accelerations, `qdd` are sampled at these time intervals.

Example: `0:0.01:10`

Data Types: `single` | `double`

pp — Piecewise polynomials
cell array or structures

Piecewise polynomials, returned as a cell array of structures that defines the polynomial for each section of the piecewise trajectory. If all the elements of the trajectory share the same breaks, the cell array is a single piecewise polynomial structure. Otherwise, the cell array has *n* elements, which correspond to each of the different trajectory elements (dimensions). Each structure contains the fields:

- `form`: `'pp'`.
- `breaks`: *p*-element vector of times when the piecewise trajectory changes forms. *p* is the number of waypoints.
- `coefs`: *n*(*p*–1)-by-`order` matrix for the coefficients for the polynomials. *n*(*p*–1) is the dimension of the trajectory times the number of `pieces`. Each set of *n* rows defines the coefficients for the polynomial that described each variable trajectory.
- `pieces`: *p*–1. The number of breaks minus 1.
- `order`: Degree of the polynomial + 1. For example, cubic polynomials have an order of 4.
- `dim`: *n*. The dimension of the control point positions.

You can build your own piecewise polynomials using `mkpp`, or evaluate the polynomial at specified times using `ppval`.

pp — Piecewise-polynomial
structure

Piecewise-polynomial, returned as a structure that defines the polynomial for each section of the piecewise trajectory. You can build your own piecewise polynomials using `mkpp`, or evaluate the polynomial at specified times using `ppval`. The structure contains the fields:

- `form`: `'pp'`.
- `breaks`: *p*-element vector of times when the piecewise trajectory changes forms. *p* is the number of waypoints.
- `coefs`: *n*(*p*–1)-by-`order` matrix for the coefficients for the polynomials. *n*(*p*–1) is the dimension of the trajectory times the number of `pieces`. Each set of *n* rows defines the coefficients for the polynomial that described each variable trajectory.
- `pieces`: *p*–1. The number of breaks minus 1.
- `order`: Degree of the polynomial + 1. For example, cubic polynomials have an order of 4.
- `dim`: *n*. The dimension of the control point positions.

Version History

Introduced in R2019a

References

[1] Lynch, Kevin M., and Frank C. Park. *Modern Robotics: Mechanics, Planning and Control*. Cambridge: Cambridge University Press, 2017.

[2] Spong, Mark W., Seth Hutchinson, and M. Vidyasagar. *Robot Modeling and Control*. John Wiley & Sons, 2006.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

`bsplinepolytraj` | `contopptraj` | `cubicpolytraj` | `quinticpolytraj` | `rottraj` | `transformtraj`

Topics
"Design Trajectory with Velocity Limits Using Trapezoidal Velocity Profile"

trvec2tform

Convert translation vector to homogeneous transformation

Syntax

`tform = trvec2tform(trvec)`

Description

`tform = trvec2tform(trvec)` converts the Cartesian representation of the translation vector `trvec` to the corresponding homogeneous transformation `tform`. When using the transformation matrix, premultiply it by the coordinates to be transformed (as opposed to postmultiplying).

Examples

Convert Translation Vector to Homogeneous Transformation

```
trvec = [0.5 6 100];
tform = trvec2tform(trvec)
```

```
tform = 4×4

    1.0000         0         0    0.5000
         0    1.0000         0    6.0000
         0         0    1.0000  100.0000
         0         0         0    1.0000
```

Input Arguments

`trvec` — Cartesian representation of translation vector
n-by-2 matrix | *n*-by-3 matrix

Cartesian representation of a translation vector, specified as an *n*-by-2 matrix if `tform` is a 3-by-3-by-*n* array and an *n*-by-3 matrix if `tform` is a 4-by-4-by-*n* array. *n* is the number of translation vectors. Each vector is of the form [*x* *y*] or [*x* *y* *z*].

Example: `[0.5 6 100]`

Output Arguments

`tform` — Homogeneous transformation
3-by-3-by-*n* array | 4-by-4-by-*n* array

Homogeneous transformation, returned as a 3-by-3-by-*n* array or 4-by-4-by-*n* array. *n* is the number of homogeneous transformations. When using the rotation matrix, premultiply it with the coordinates to be rotated (as opposed to postmultiplying).

Example: `[0 0 1 0; 0 1 0 0; -1 0 0 0; 0 0 0 1]`

2-D homogeneous transformation matrices are of the form:

$$T = \begin{bmatrix} r_{11} & r_{12} & t_1 \\ r_{21} & r_{22} & t_2 \\ 0 & 0 & 1 \end{bmatrix}$$

3-D homogeneous transformation matrices are of the form:

$$T = \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_1 \\ r_{21} & r_{22} & r_{23} & t_2 \\ r_{31} & r_{32} & r_{33} & t_3 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

More About

Homogeneous Transformation Matrices

Homogeneous transformation matrices consist of both an orthogonal rotation and a translation.

2-D Transformations

2-D transformations have a rotation θ about the z-axis:

$$R_z(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}$$

, and a translation along the x and y axis:

$$t = \begin{bmatrix} x \\ y \end{bmatrix}$$

, resulting in the 2-D transformation matrix of the form:

$$T = \begin{bmatrix} R & t \\ 0_{1\times 2} & 1 \end{bmatrix} = \begin{bmatrix} I_2 & t \\ 0_{1\times 2} & 1 \end{bmatrix} \cdot \begin{bmatrix} R & 0 \\ 0_{1\times 2} & 1 \end{bmatrix}$$

3-D Transformations

3-D transformations contain information about three rotations about the x-, y-, and z-axes:

$$R_x(\phi) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\phi & \sin\phi \\ 0 & -\sin\phi & \cos\phi \end{bmatrix}, R_y(\psi) = \begin{bmatrix} \cos\psi & 0 & \sin\psi \\ 0 & 1 & 0 \\ -\sin\psi & 0 & \cos\psi \end{bmatrix}, R_z(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

and after multiplying become the rotation about the xyz-axes:

$$R_{xyz} = R_x(\phi)R_y(\psi)R_z(\theta) =$$
$$\begin{bmatrix} \cos\phi\cos\psi\cos\theta - \sin\phi\sin\theta & -\cos\phi\cos\psi\sin\theta - \sin\phi\cos\theta & \cos\phi\sin\psi \\ \sin\phi\cos\psi\cos\theta + \cos\phi\sin\theta & -\sin\phi\cos\psi\sin\theta + \cos\phi\cos\theta & \sin\phi\sin\psi \\ -\sin\psi\cos\theta & \sin\psi\sin\theta & \cos\psi \end{bmatrix}$$

and a translation along the x-, y-, and z-axis:

$$t = \begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

, resulting in the 3-D transformation matrix of the form:

$$T = \begin{bmatrix} R & t \\ 0_{1x3} & 1 \end{bmatrix} = \begin{bmatrix} I_3 & t \\ 0_{1x3} & 1 \end{bmatrix} \cdot \begin{bmatrix} R & 0 \\ 0_{1x3} & 1 \end{bmatrix}$$

Version History

Introduced in R2015a

R2023a: `trvec2tform` Supports 2-D Translation Vectors

The `trvec` argument now accepts 2-D translation vectors as a *n*-by-2 matrix and `trvec2tform` outputs 2-D homogeneous transformation matrices as a 3-by-3-by-*n* array.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`tform2trvec` | `se2` | `se3`

Topics

"Coordinate Transformations in Robotics"

uminus, -

Quaternion unary minus

Syntax

```
mQuat = -quat
```

Description

`mQuat = -quat` negates the elements of `quat` and stores the result in `mQuat`.

Examples

Negate Elements of Quaternion Matrix

Unary minus negates each part of a the quaternion. Create a 2-by-2 matrix, `Q`.

```
Q = quaternion(randn(2),randn(2),randn(2),randn(2))
```

```
Q = 2x2 quaternion array
      0.53767 +  0.31877i +   3.5784j +   0.7254k      -2.2588 -  0.43359i -   1.3499j +  0.7147
       1.8339 -   1.3077i +   2.7694j - 0.063055k      0.86217 +  0.34262i +   3.0349j -  0.2049
```

Negate the parts of each quaternion in `Q`.

```
R = -Q
```

```
R = 2x2 quaternion array
     -0.53767 -  0.31877i -   3.5784j -   0.7254k       2.2588 +  0.43359i +   1.3499j -  0.7147
      -1.8339 +   1.3077i -   2.7694j + 0.063055k     -0.86217 -  0.34262i -   3.0349j +  0.2049
```

Input Arguments

`quat` — Quaternion array
scalar | vector | matrix | multidimensional array

Quaternion array, specified as a scalar, vector, matrix, or multidimensional array.

Data Types: `quaternion`

Output Arguments

`mQuat` — Negated quaternion array
scalar | vector | matrix | multidimensional array

Negated quaternion array, returned as the same size as `quat`.

Data Types: `quaternion`

Version History

Introduced in R2018a

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

Functions

`minus, -`

Objects

`quaternion`

updateErrorDynamicsFromStep

Update values of NaturalFrequency and DampingRatio properties given desired step response

Syntax

```
updateErrorDynamicsFromStep(motionModel,settlingTime,overshoot)
updateErrorDynamicsFromStep(motionModel,settlingTime,overshoot,jointIndex)
```

Description

`updateErrorDynamicsFromStep(motionModel,settlingTime,overshoot)` updates the values of the `NaturalFrequency` and `DampingRatio` properties of the given `jointSpaceMotionModel` object given the desired step response.

`updateErrorDynamicsFromStep(motionModel,settlingTime,overshoot,jointIndex)` updates the `NaturalFrequency` and `DampingRatio` properties for a specific joint. In this case, the values of `SettlingTime` and `Overshoot` must be provided as scalars because they apply to a single joint.

Examples

Create Joint-Space Motion Model

This example shows how to create and use a `jointSpaceMotionModel` object for a manipulator robot in joint-space.

Create the Robot

```
robot = loadrobot("kinovaGen3","DataFormat","column","Gravity",[0 0 -9.81]);
```

Set Up the Simulation

Set the timespan to be 1 s with a timestep size of 0.01 s. Set the initial state to be the robots, home configuration with a velocity of zero.

```
tspan = 0:0.01:1;
initialState = [homeConfiguration(robot); zeros(7,1)];
```

Define the a reference state with a target position, zero velocity, and zero acceleration.

```
targetState = [pi/4; pi/3; pi/2; -pi/3; pi/4; -pi/4; 3*pi/4; zeros(7,1); zeros(7,1)];
```

Create the Motion Model

Model the system with computed torque control and error dynamics defined by a moderately fast step response with 5% overshoot.

```
motionModel = jointSpaceMotionModel("RigidBodyTree",robot);
updateErrorDynamicsFromStep(motionModel,.3,.05);
```

Simulate the Robot

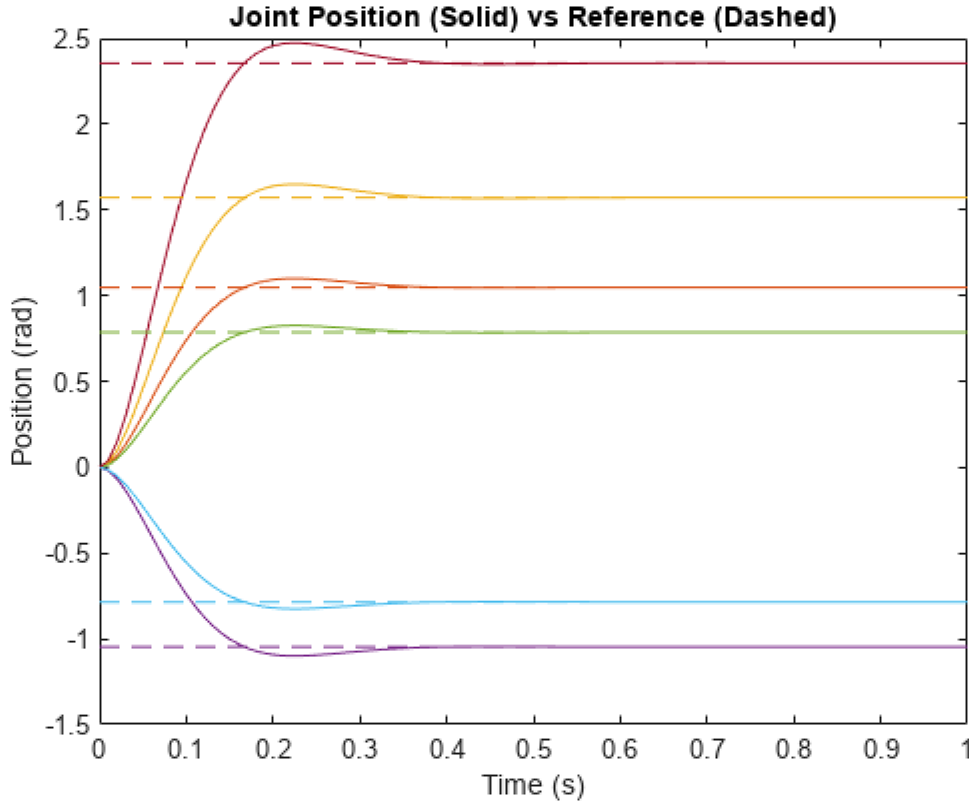
Use the derivative function of the model as the input to the `ode45` solver to simulate the behavior over 1 second.

```
[t,robotState] = ode45(@(t,state)derivative(motionModel,state,targetState),tspan,initialState);
```

Plot the Response

Plot the positions of all the joints actuating to their target state. Joints with a higher displacement between the starting position and the target position actuate to the target at a faster rate than those with a lower displacement. This leads to an overshoot, but all of the joints have the same settling time.

```
figure
plot(t,robotState(:,1:motionModel.NumJoints));
hold all;
plot(t,targetState(1:motionModel.NumJoints)*ones(1,length(t)),"--");
title("Joint Position (Solid) vs Reference (Dashed)");
xlabel("Time (s)")
ylabel("Position (rad)");
```



Input Arguments

`motionModel` — `jointSpaceMotionModel` object
`jointSpaceMotionModel` object

The `jointSpaceMotionModel` object, which defines the properties of the motion model.

`settlingTime` — Settling time of system
n-element vector

Settling time required to reach a 2% tolerance band in seconds, specified as a scalar or an *n*-element vector. *n* is the number of nonfixed joints in the `rigidBodyTree` of the `jointSpaceMotionModel` in the `motionModel` argument.

`overshoot` — Overshoot of system
n-element vector

The overshoot relative to a unit step, specified as a scalar or an *n*-element vector. *n* is the number of nonfixed joints in the `rigidBodyTree` of the `jointSpaceMotionModel` in the `motionModel` argument.

`jointIndex` — Joint index
scalar

The index of the joint for which `NaturalFrequency` and `DampingRatio` is updated given the unit-step error dynamics. In this case, settling time and overshoot must be specified as scalars.

Version History

Introduced in R2019b

References

[1] Ogata, Katsuhiko. *Modern Control Engineering* 4th ed. Englewood Cliffs, NJ: Prentice-Hall, 2002.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

Classes
`jointSpaceMotionModel` | `taskSpaceMotionModel`

writeBinaryOccupancyGrid

Write values from grid to ROS message

Syntax

`writeBinaryOccupancyGrid(msg,map)`

Description

`writeBinaryOccupancyGrid(msg,map)` writes occupancy values and other information to the ROS message, `msg`, from the binary occupancy grid, `map`.

Note The `msg` input is an `'nav_msgs/OccupancyGrid'` ROS message. For more info, see `OccupancyGrid`.

Input Arguments

`map` — Binary occupancy grid
`binaryOccupancyMap` object handle

Binary occupancy grid, specified as a `binaryOccupancyMap` object handle. `map` is converted to a `'nav_msgs/OccupancyGrid'` message on the ROS network. `map` is an object with a grid of binary values, where `1` indicates an occupied location and `0` indications an unoccupied location.

`msg` — `'nav_msgs/OccupancyGrid'` ROS message
`OccupancyGrid` object handle

`'nav_msgs/OccupancyGrid'` ROS message, specified as a `OccupancyGrid` object handle.

Version History

Introduced in R2015a

See Also

Functions
`rosReadBinaryOccupancyGrid` | `rosReadOccupancyMap3D` | `rosReadOccupancyGrid` | `rosWriteOccupancyGrid`

zeros

Create quaternion array with all parts set to zero

Syntax

```
quatZeros = zeros('quaternion')
quatZeros = zeros(n,'quaternion')
quatZeros = zeros(sz,'quaternion')
quatZeros = zeros(sz1,...,szN,'quaternion')

quatZeros = zeros( ___ ,'like',prototype,'quaternion')
```

Description

`quatZeros = zeros('quaternion')` returns a scalar quaternion with all parts set to zero.

`quatZeros = zeros(n,'quaternion')` returns an `n`-by-`n` matrix of quaternions.

`quatZeros = zeros(sz,'quaternion')` returns an array of quaternions where the size vector, `sz`, defines `size(quatZeros)`.

`quatZeros = zeros(sz1,...,szN,'quaternion')` returns a `sz1`-by-...-by-`szN` array of quaternions where `sz1,…,szN` indicates the size of each dimension.

`quatZeros = zeros( ___ ,'like',prototype,'quaternion')` specifies the underlying class of the returned quaternion array to be the same as the underlying class of the quaternion prototype.

Examples

Quaternion Scalar Zero

Create a quaternion scalar zero.

```
quatZeros = zeros('quaternion')
```

```
quatZeros = quaternion
     0 + 0i + 0j + 0k
```

Square Matrix of Quaternions

Create an n-by-n array of quaternion zeros.

```
n = 3;
quatZeros = zeros(n,'quaternion')
```

```
quatZeros = 3x3 quaternion array
     0 + 0i + 0j + 0k     0 + 0i + 0j + 0k     0 + 0i + 0j + 0k
```

```
     0 + 0i + 0j + 0k     0 + 0i + 0j + 0k     0 + 0i + 0j + 0k
     0 + 0i + 0j + 0k     0 + 0i + 0j + 0k     0 + 0i + 0j + 0k
```

Multidimensional Array of Quaternion Zeros

Create a multidimensional array of quaternion zeros by defining array dimensions in order. In this example, you create a 3-by-1-by-2 array. You can specify dimensions using a row vector or comma-separated integers.

Specify the dimensions using a row vector and display the results:

```
dims = [3,1,2];
quatZerosSyntax1 = zeros(dims,'quaternion')
```

```
quatZerosSyntax1 = 3x1x2 quaternion array
quatZerosSyntax1(:,:,1) =

     0 + 0i + 0j + 0k
     0 + 0i + 0j + 0k
     0 + 0i + 0j + 0k


quatZerosSyntax1(:,:,2) =

     0 + 0i + 0j + 0k
     0 + 0i + 0j + 0k
     0 + 0i + 0j + 0k
```

Specify the dimensions using comma-separated integers, and then verify the equivalence of the two syntaxes:

```
quatZerosSyntax2 = zeros(3,1,2,'quaternion');
isequal(quatZerosSyntax1,quatZerosSyntax2)
```

```
ans = logical
   1
```

Underlying Class of Quaternion Zeros

A quaternion is a four-part hyper-complex number used in three-dimensional representations. You can specify the underlying data type of the parts as `single` or `double`. The default is `double`.

Create a quaternion array of zeros with the underlying data type set to `single`.

```
quatZeros = zeros(2,'like',single(1),'quaternion')
```

```
quatZeros = 2x2 quaternion array
     0 + 0i + 0j + 0k     0 + 0i + 0j + 0k
     0 + 0i + 0j + 0k     0 + 0i + 0j + 0k
```

Verify the underlying class using the `classUnderlying` function.

```
classUnderlying(quatZeros)
```

```
ans =
'single'
```

Input Arguments

`n` — Size of square quaternion matrix
integer value

Size of square quaternion matrix, specified as an integer value. If `n` is `0` or negative, then `quatZeros` is returned as an empty matrix.

Example: `zeros(4,'quaternion')` returns a 4-by-4 matrix of quaternion zeros.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`sz` — Output size
row vector of integer values

Output size, specified as a row vector of integer values. Each element of `sz` indicates the size of the corresponding dimension in `quatZeros`. If the size of any dimension is `0` or negative, then `quatZeros` is returned as an empty array.

Example: `zeros([1,4,2],'quaternion')` returns a 1-by-4-by-2 array of quaternion zeros.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`prototype` — Quaternion prototype
variable

Quaternion prototype, specified as a variable.

Example: `zeros(2,'like',quat,'quaternion')` returns a 2-by-2 matrix of quaternions with the same underlying class as the prototype quaternion, `quat`.

Data Types: `quaternion`

`sz1,...,szN` — Size of each dimension
two or more integer values

Size of each dimension, specified as two or more integers.

- If the size of any dimension is `0`, then `quatZeros` is returned as an empty array.
- If the size of any dimension is negative, then it is treated as `0`.

Example: `zeros(2,3,'quaternion')` returns a 2-by-3 matrix of quaternion zeros.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Output Arguments

`quatZeros` — Quaternion zeros
scalar | vector | matrix | multidimensional array

Quaternion zeros, returned as a quaternion or array of quaternions.

Given a quaternion of the form $Q = a + b\text{i} + c\text{j} + d\text{k}$, a quaternion zero is defined as $Q = 0 + 0\text{i} + 0\text{j} + 0\text{k}$.

Data Types: `quaternion`

Version History

Introduced in R2018a

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

Functions

`ones`

Objects

`quaternion`

3

Methods

generateIKFunction

Generate function for closed-form inverse kinematics

Syntax

`ikFunction = generateIKFunction(analyticalIK,functionName)`

Description

`ikFunction = generateIKFunction(analyticalIK,functionName)` generates a function with a specified name, `functionName`, that computes the closed-form solutions for inverse kinematics (IK) for a selected kinematic group to achieve a desired end-effector pose. To generate a list of configurations that achieve the desired end-effector pose, use the generated function `ikFunction`. The specified `analyticalInverseKinematics` object `analyticalIK` must contain a valid kinematic group. For information on determining valid kinematic groups, see the `showdetails` function.

For the syntax of the generated function, see the `ikFunction` output argument.

Examples

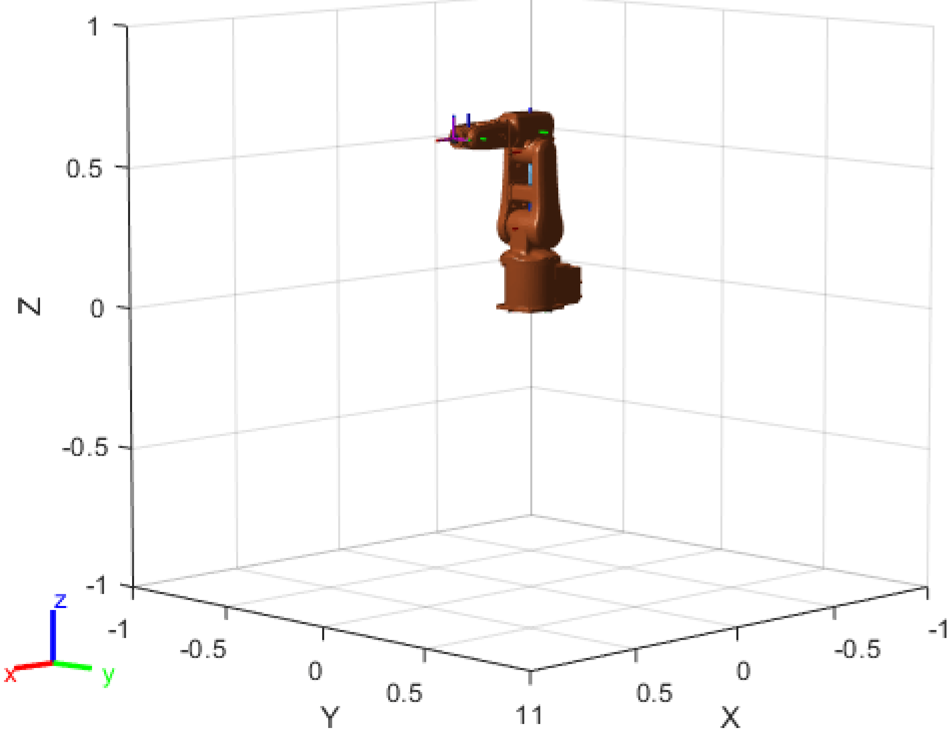
Solve Analytical Inverse Kinematics for Robot Manipulator

Generate closed-form inverse kinematics (IK) solutions for a desired end effector. Load the provided robot model and inspect details about the feasible kinematic groups of base and end-effector bodies. Generate a function for your desired kinematic group. Inspect the various configurations for a specific end-effector pose.

Robot Model

Load the ABB IRB 120 robot model into the workspace. Display the model.

```
robot = loadrobot('abbIrb120','DataFormat','row');
show(robot);
```



Analytical IK

Create the analytical IK solver. Show details for the robot model, which lists the different kinematic groups available for closed-form analytical IK solutions. Select the second kinematic group by clicking the **Use this kinematic group** link in the second row of the table.

```
aik = analyticalInverseKinematics(robot);
showdetails(aik)
```

```
--------------------
Robot: (8 bodies)

Index      Base Name   EE Body Name     Type                   Actions
-----      ---------   ------------     ----                   -------
    1      base_link         link_6   RRRSSS   Use this kinematic group
    2      base_link          tool0   RRRSSS   Use this kinematic group
```

Inspect the kinematic group, which lists the base and end-effector body names. For this robot, the group uses the `'base_link'` and `'tool0'` bodies, respectively.

```
aik.KinematicGroup
```

```
ans = struct with fields:
               BaseName: 'base_link'
    EndEffectorBodyName: 'tool0'
```

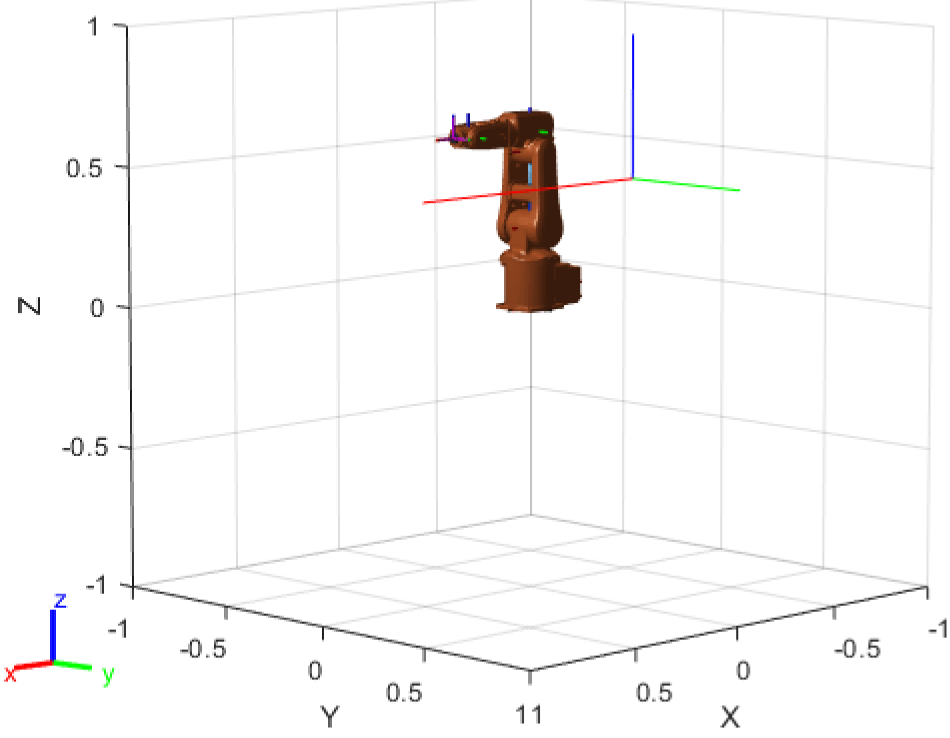
Generate Function

Generate the IK function for the selected kinematic group. Specify a name for the function, which is generated and saved in the current directory.

```
generateIKFunction(aik,'robotIK');
```

Specify a desired end-effector position. Convert the *xyz*-position to a homogeneous transformation.

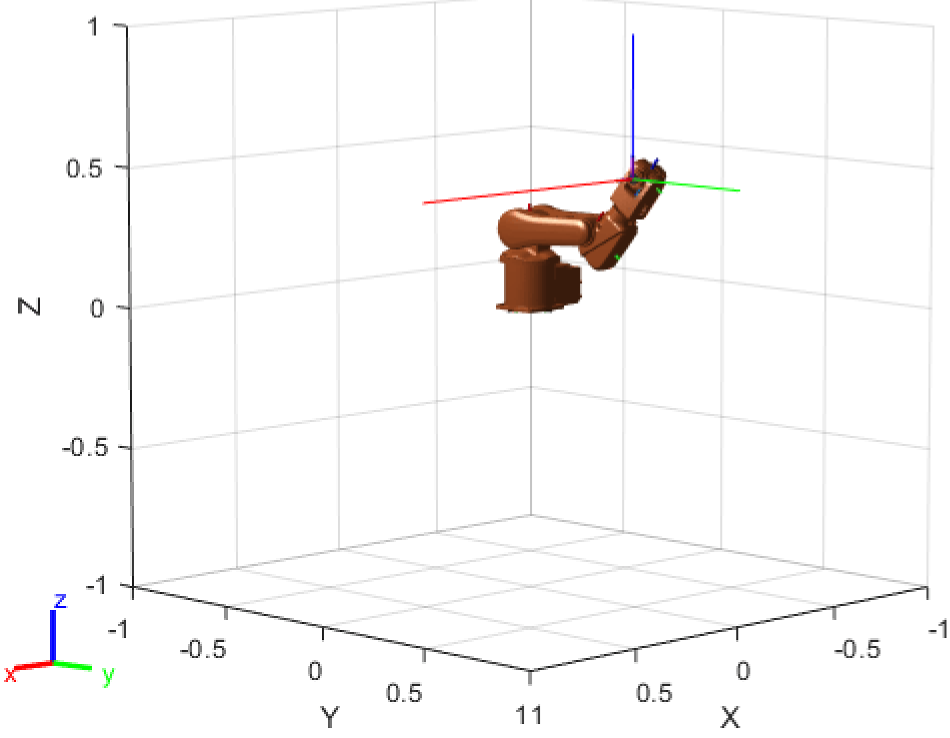
```
eePosition = [0 0.5 0.5];
eePose = trvec2tform(eePosition);
hold on
plotTransforms(eePosition,tform2quat(eePose))
hold off
```



Generate Configuration for IK Solution

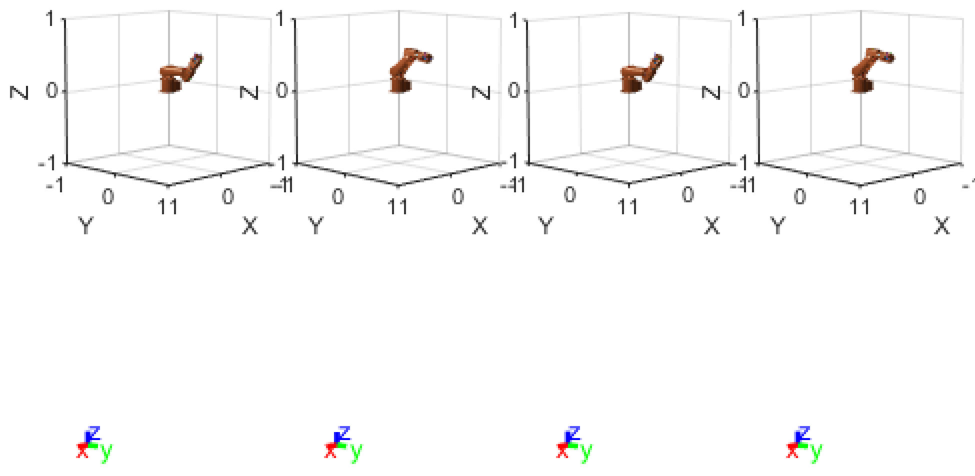
Specify the homogeneous transformation to the generated IK function, which generates all solutions for the desired end-effector pose. Display the first generated configuration to verify that the desired pose has been achieved.

```
ikConfig = robotIK(eePose); % Uses the generated file

show(robot,ikConfig(1,:));
hold on
plotTransforms(eePosition,tform2quat(eePose))
hold off
```



Display all of the closed-form IK solutions sequentially.

```
figure;
numSolutions = size(ikConfig,1);

for i = 1:size(ikConfig,1)
    subplot(1,numSolutions,i)
    show(robot,ikConfig(i,:));
end
```



Solve Analytical IK for Large-DOF Robot

Some manipulator robot models have large degrees-of-freedom (DOFs). To reach certain end-effector poses, however, only six DOFs are required. Use the `analyticalInverseKinematics` object, which supports six-DOF robots, to determine various valid kinematic groups for this large-DOF robot model. Use the `showdetails` object function to get information about the model.

Load Robot Model and Generate IK Solver

Load the robot model into the workspace, and create an `analyicalInverseKinematics` object. Use the `showdetails` object function to see the supported kinematic groups.

```
robot = loadrobot('willowgaragePR2','DataFormat','row');
aik = analyticalInverseKinematics(robot);
opts = showdetails(aik);
```

```
--------------------
Robot: (94 bodies)

Index                                        Base Name                           EE
-----                                        ---------                           --
    1                                 l_shoulder_pan_link                     l_wris
    2                                 r_shoulder_pan_link                     r_wris
    3                                 l_shoulder_pan_link                   l_gripper
    4                                 r_shoulder_pan_link                   r_gripper
```

```
     5                              l_shoulder_pan_link                              l_gripper
     6                              l_shoulder_pan_link                l_gripper_motor_accelero
     7                              l_shoulder_pan_link                             l_gripper_
     8                              r_shoulder_pan_link                              r_gripper
     9                              r_shoulder_pan_link                r_gripper_motor_accelero
    10                              r_shoulder_pan_link                             r_gripper_
```

Select a group programmatically using the output of the `showdetails` object function, `opts`. The selected group uses the left shoulder as the base with the left wrist as the end effector.

```
aik.KinematicGroup = opts(1).KinematicGroup;
disp(aik.KinematicGroup)
```

```
               BaseName: 'l_shoulder_pan_link'
    EndEffectorBodyName: 'l_wrist_roll_link'
```

Generate the IK function for the selected group.

```
generateIKFunction(aik,'willowRobotIK');
```

Solve Analytical IK

Define a target end-effector pose using a randomly-generated configuration.

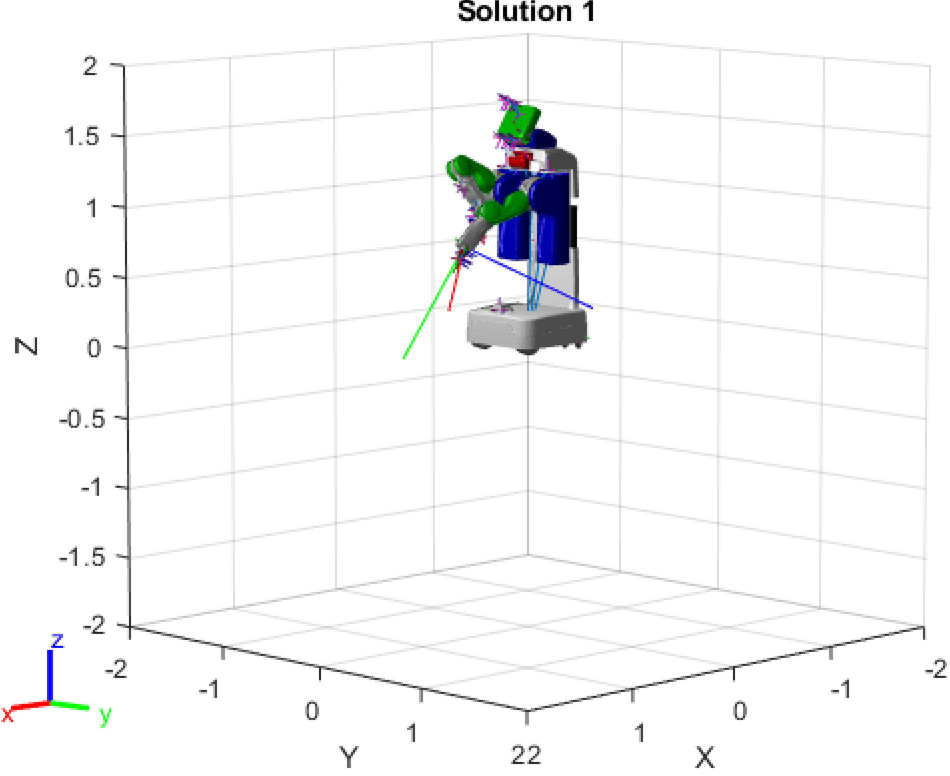
```
rng(0);
expConfig = randomConfiguration(robot);

eeBodyName = aik.KinematicGroup.EndEffectorBodyName;
baseName = aik.KinematicGroup.BaseName;
expEEPose = getTransform(robot,expConfig,eeBodyName,baseName);
```

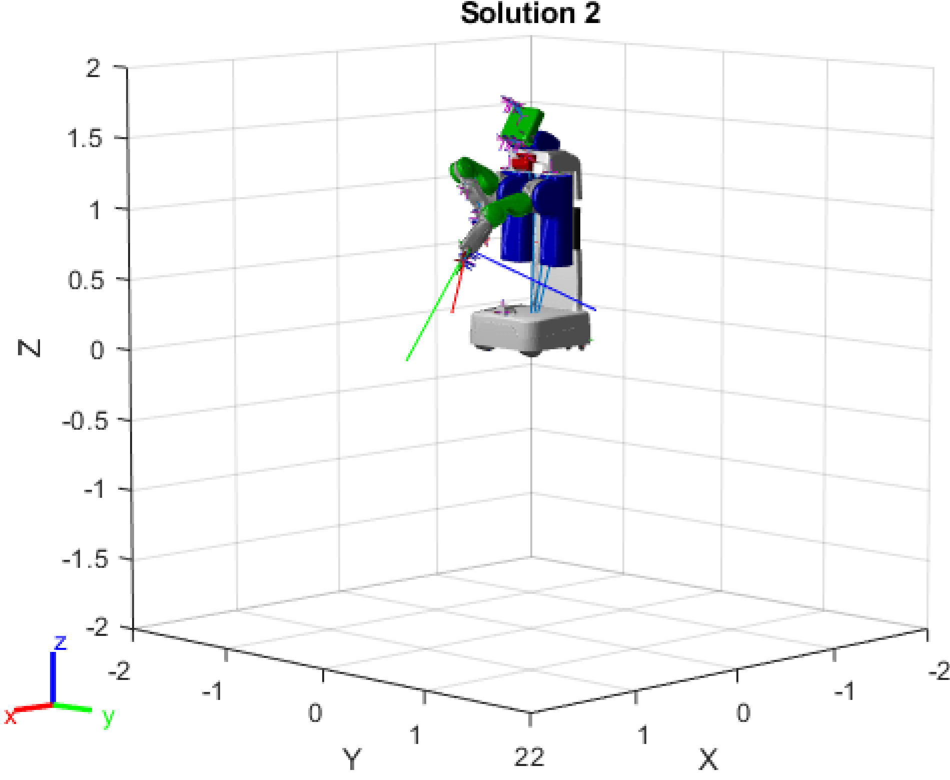
Solve for all robot configurations that achieve the defined end-effector pose using the generated IK function. To ignore joint limits, specify `false` as the second input argument.

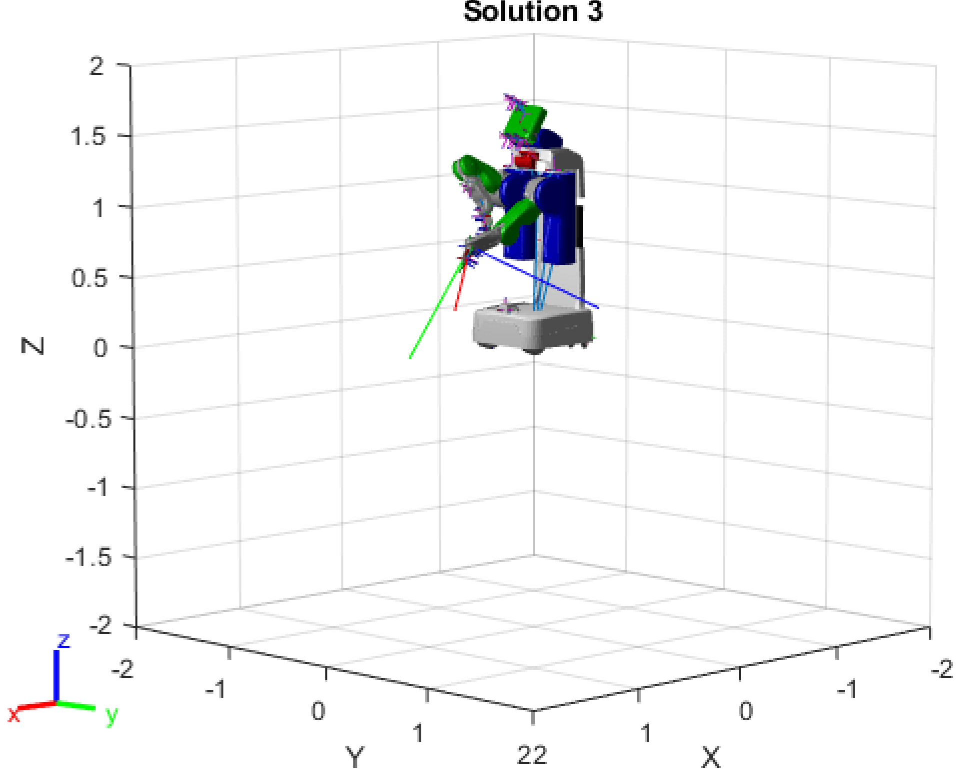
```
ikConfig = willowRobotIK(expEEPose,false);
```

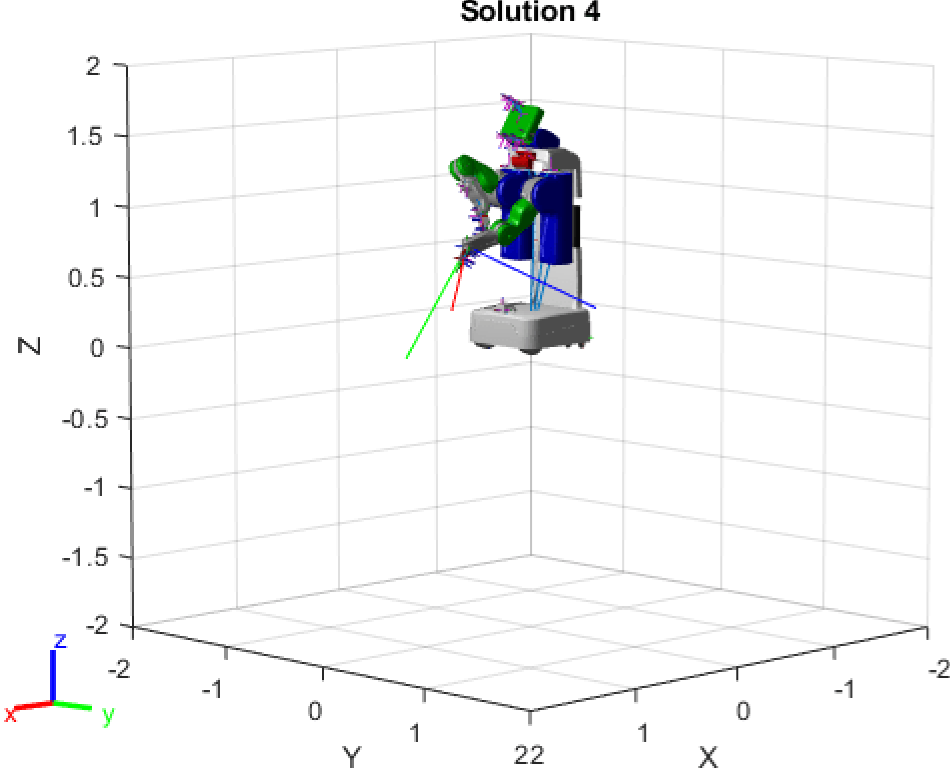
To display the target end-effector pose in the world frame, get the transformation from the base of the robot model, rather than the base of the kinematic group. Display all of the generated IK solutions by specifying the indices for your kinematic group IK solution in the configuration vector used with the `show` function.

```
eeWorldPose = getTransform(robot,expConfig,eeBodyName);

generatedConfig = repmat(expConfig, size(ikConfig,1), 1);
generatedConfig(:,aik.KinematicGroupConfigIdx) = ikConfig;

for i = 1:size(ikConfig,1)
    figure;
    ax = show(robot,generatedConfig(i,:));
    hold all;
    plotTransforms(tform2trvec(eeWorldPose),tform2quat(eeWorldPose),'Parent',ax);
    title(['Solution ' num2str(i)]);
end
```

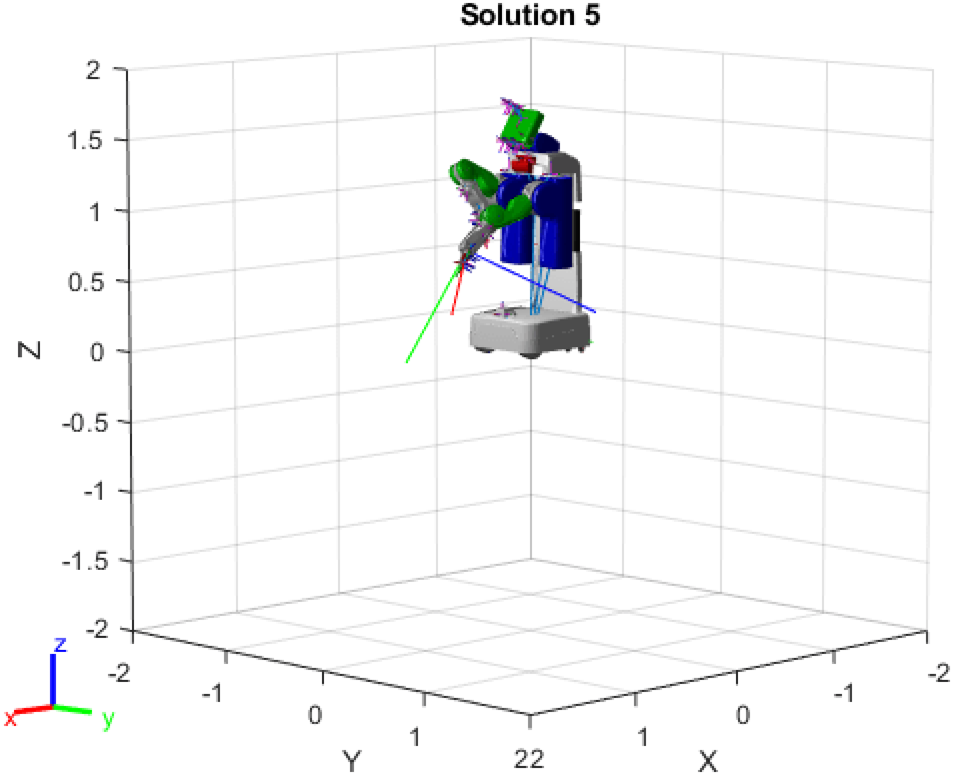

Solution 1
2
1.5
1
0.5
0
-0.5
-1
-1.5
-2
Z
-2
-1
0
1
2
Y
2
1
0
-1
-2
X
z
x
y

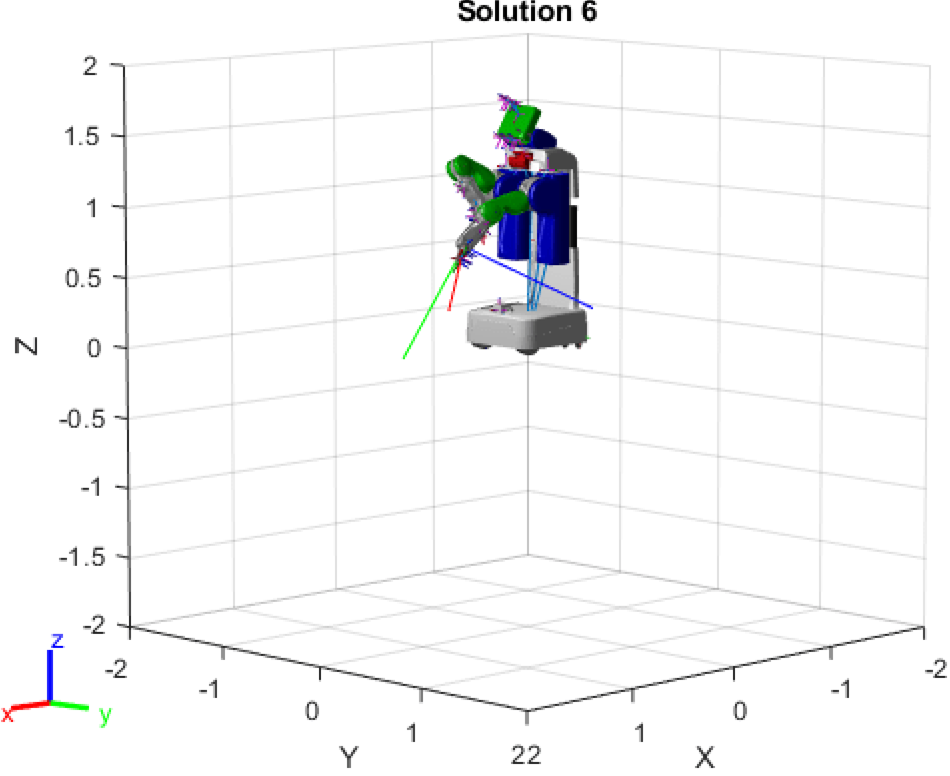

Solution 2
Z
Y
X
2
1.5
1
0.5
0
-0.5
-1
-1.5
-2
-2
-1
0
1
22
1
0
-1
-2
z
x
y

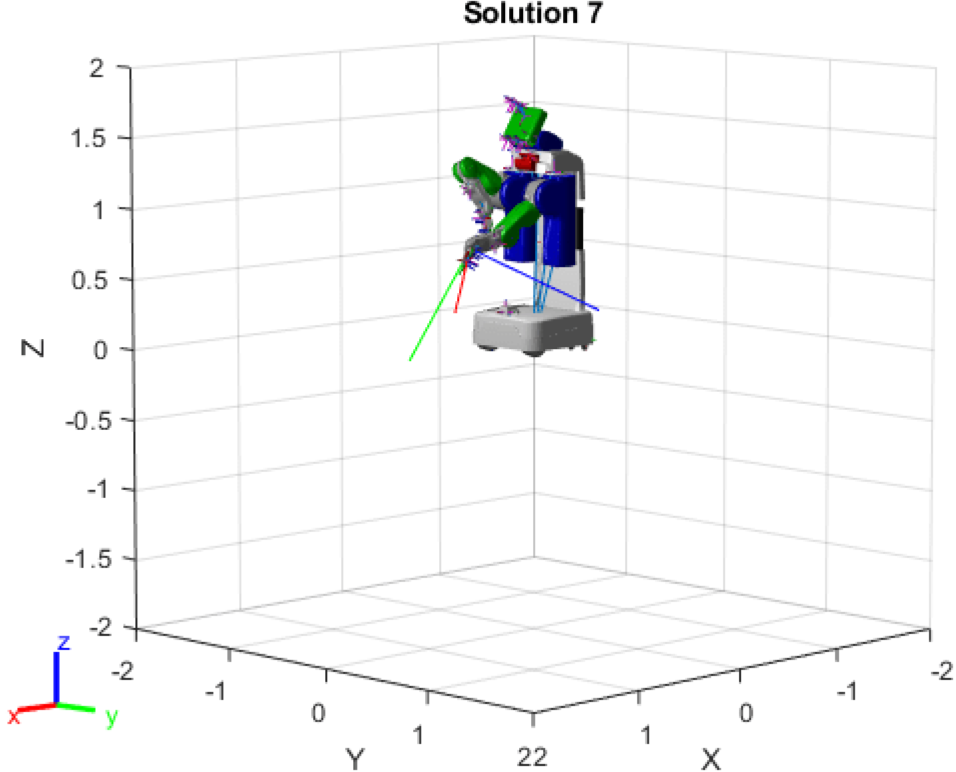

Solution 3
Z
2
1.5
1
0.5
0
-0.5
-1
-1.5
-2
Y
-2
-1
0
1
22
X
1
0
-1
-2
z
x
y

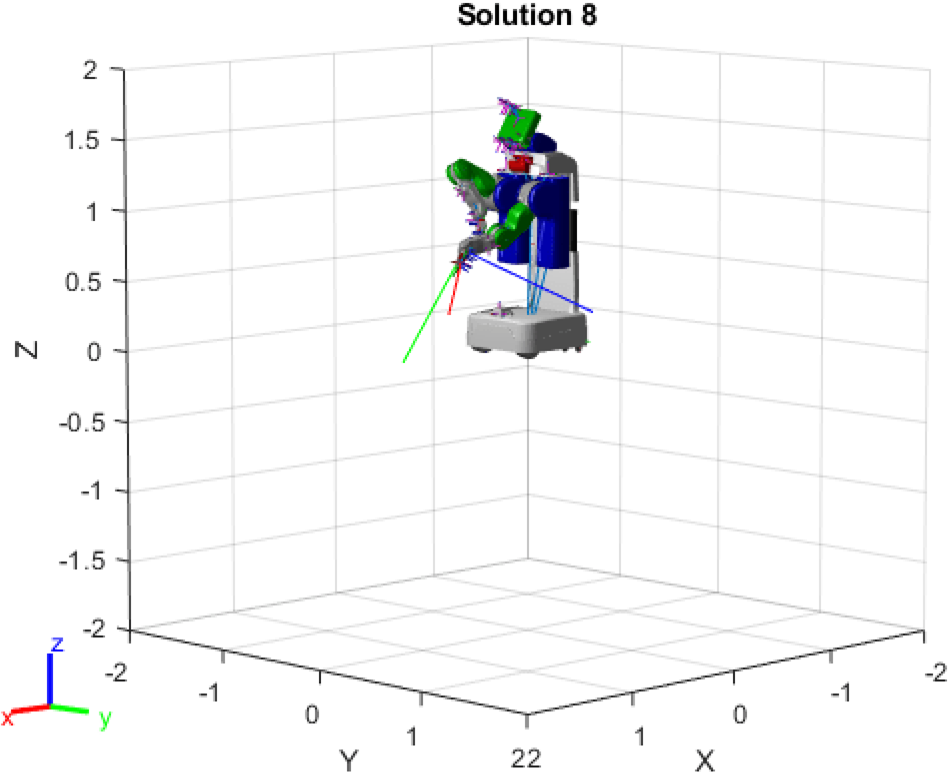


Solution 4
2
1.5
1
0.5
0
-0.5
-1
-1.5
-2
Z
-2
-1
0
1
2
Y
2
1
0
-1
-2
X
z
x
y


Solution 5
2
1.5
1
0.5
0
-0.5
-1
-1.5
-2
Z
-2
-1
0
1
2
Y
2
1
0
-1
-2
X
z
x
y


Solution 6
Z
Y
X
2
1.5
1
0.5
0
-0.5
-1
-1.5
-2
-2
-1
0
1
22
1
0
-1
-2
z
x
y


Solution 7
2
1.5
1
0.5
Z
0
-0.5
-1
-1.5
-2
-2
-1
0
1
Y
22
1
0
-1
-2
X
z
x
y



Input Arguments

`analyticalIK` — Analytical IK solver
`analyticalInverseKinematics` object

Analytical inverse kinematics solver, specified as an `analyticalInverseKinematics` object.

`functionName` — Name of generated function
string scalar | character vector

Name of the generated function, specified as a string scalar or character vector.

Example: `"jacoIKSolver"`

Data Types: `char` | `string`

Output Arguments

`ikFunction` — IK solver for selected kinematic group
function handle

IK solver for the selected kinematic group, returned as a function handle. The function generates closed-form solutions and has these syntax options:

```
robotConfig = ikFunction(eeTransform)
robotConfig = ikFunction(eeTransform,enforceJointLimits)
```

```
robotConfig = ikFunction(eeTransform,enforceJointLimits,sortByDistance)
robotConfig = ikFunction(eeTransform,enforceJointLimits,sortByDistance,referenceConfig)
```

eeTransform — Desired end-effector pose
4-by-4 homogeneous transformation matrix

Desired end-effector pose, specified as a 4-by-4 homogeneous transformation matrix. To generate a transformation matrix from an *xyz*-position and quaternion orientation, use the `trvec2tform` and `quat2tform` functions on the respective coordinates and multiply the resulting matrices.

```
tform1 = trvec2tform([x y z]);
tform2 = quat2tform([qw qx qy qz]);
eeTransform = tform1*tform2;
```

Data Types: `single` | `double`

enforceJointLimits — Enforce joint limits of rigid body tree model
`1` (`true`) | `0` (`false`)

Enforce joint limits of the rigid body tree model, specified as a logical, `1` (`true` or `0` (`false`). When set to `false`, the solver ignores the joint limits of the robot model in the RigidBodyTree property of the `analyticalInverseKinematics` object.

Data Types: `logical`

sortByDistance — Sort configurations based on distance from desired pose
`1` (`true`) | `0` (`false`)

Sort configurations based on distance from desired pose, specified as a logical, `1` (`true` or `0` (`false`).

Data Types: `logical`

referenceConfig — Reference configuration for IK solution
n-element vector

Reference configuration for the IK solution, specified as an *n*-element vector, where *n* is the number of nonfixed joints in the rigid body tree robot model. Each element corresponds to a joint position as either a rotation angle in radians for revolute joints or a linear distance in meters for prismatic joints.

Data Types: `single` | `double`

Version History

Introduced in R2020b

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

The `analyticalInverseKinematics` object only supports code generation for the function created by calling the `generateIKFunction`. Use the `analyticalInverseKinematics` object to modify parameters and setup the solver. Then, use `generateIKFunction` to create your custom IK function for your robot model. Call `codegen` on the output `ikFunction` that is generated.

See Also

Objects
`analyticalInverseKinematics` | `inverseKinematics` | `generalizedInverseKinematics` | `rigidBodyTree`

Functions
`loadrobot` | `importrobot` | `showdetails`

showdetails

Display overview of available kinematic groups

Syntax

```
kinGroupDetails = showdetails(analyticalIK)
```

Description

`kinGroupDetails = showdetails(analyticalIK)` displays an overview of all the kinematic group combinations available for the `rigidBodyTree` object associated with the analytical inverse kinematics (IK) solver. Each kinematic group contains body names for both a base and end effector that are valid for closed-form solutions to analytical IK.

To use a specific kinematic group for your object, click the corresponding **Use this kinematic group** link in the output table. This link updates the KinematicGroup and KinematicGroupType properties of the `analyticalInverseKinematics` object.

Examples

Solve Analytical IK for Large-DOF Robot

Some manipulator robot models have large degrees-of-freedom (DOFs). To reach certain end-effector poses, however, only six DOFs are required. Use the `analyticalInverseKinematics` object, which supports six-DOF robots, to determine various valid kinematic groups for this large-DOF robot model. Use the `showdetails` object function to get information about the model.

Load Robot Model and Generate IK Solver

Load the robot model into the workspace, and create an `analyicalInverseKinematics` object. Use the `showdetails` object function to see the supported kinematic groups.

```
robot = loadrobot('willowgaragePR2','DataFormat','row');
aik = analyticalInverseKinematics(robot);
opts = showdetails(aik);
```

```
--------------------
Robot: (94 bodies)

Index                                   Base Name                                   EE
-----                                   ---------                                   --
    1                          l_shoulder_pan_link                          l_wris
    2                          r_shoulder_pan_link                          r_wris
    3                          l_shoulder_pan_link                        l_gripper
    4                          r_shoulder_pan_link                        r_gripper
    5                          l_shoulder_pan_link                        l_gripper
    6                          l_shoulder_pan_link            l_gripper_motor_acceler
    7                          l_shoulder_pan_link                       l_gripper_
    8                          r_shoulder_pan_link                        r_gripper
    9                          r_shoulder_pan_link            r_gripper_motor_acceler
   10                          r_shoulder_pan_link                       r_gripper_
```

Select a group programmatically using the output of the `showdetails` object function, `opts`. The selected group uses the left shoulder as the base with the left wrist as the end effector.

```
aik.KinematicGroup = opts(1).KinematicGroup;
disp(aik.KinematicGroup)
```

```
               BaseName: 'l_shoulder_pan_link'
    EndEffectorBodyName: 'l_wrist_roll_link'
```

Generate the IK function for the selected group.

```
generateIKFunction(aik,'willowRobotIK');
```

Solve Analytical IK

Define a target end-effector pose using a randomly-generated configuration.

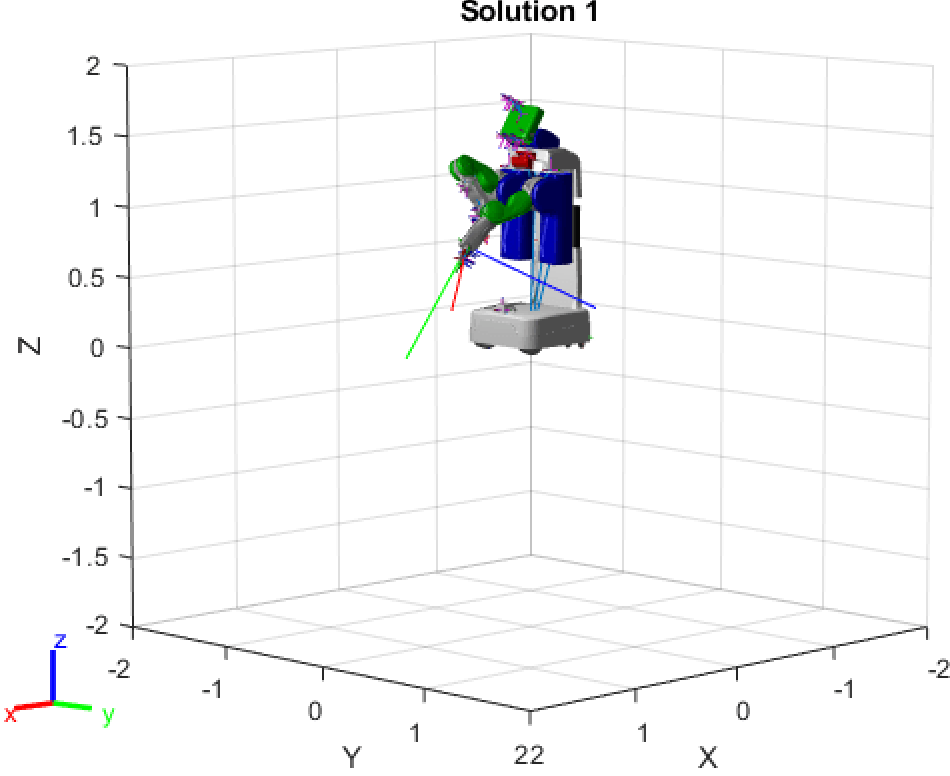
```
rng(0);
expConfig = randomConfiguration(robot);

eeBodyName = aik.KinematicGroup.EndEffectorBodyName;
baseName = aik.KinematicGroup.BaseName;
expEEPose = getTransform(robot,expConfig,eeBodyName,baseName);
```

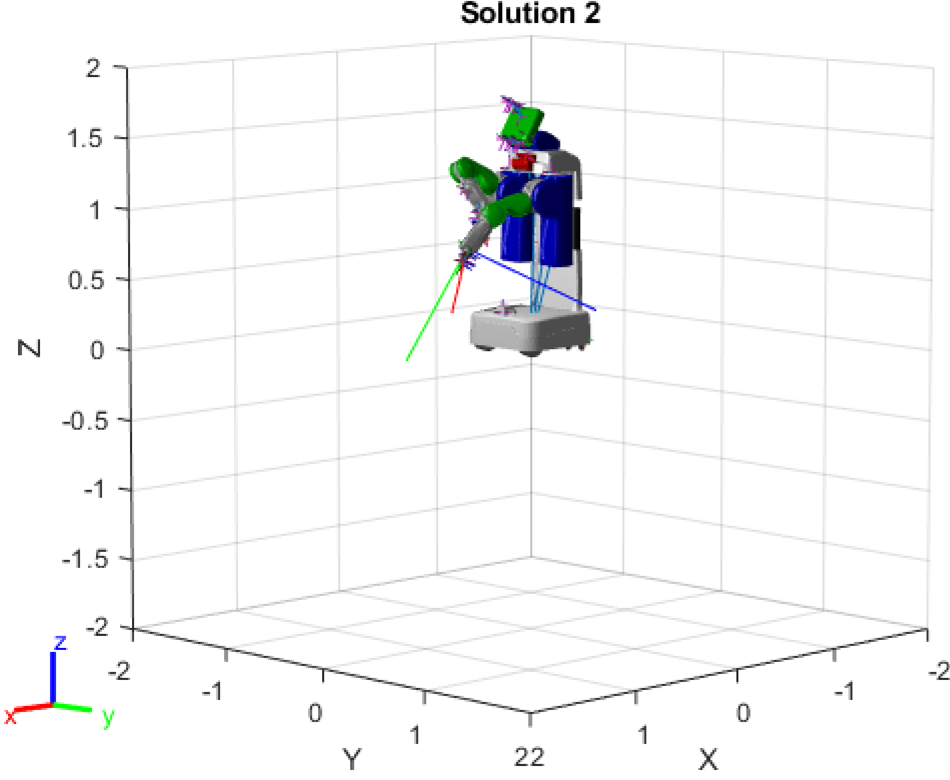
Solve for all robot configurations that achieve the defined end-effector pose using the generated IK function. To ignore joint limits, specify `false` as the second input argument.

```
ikConfig = willowRobotIK(expEEPose,false);
```

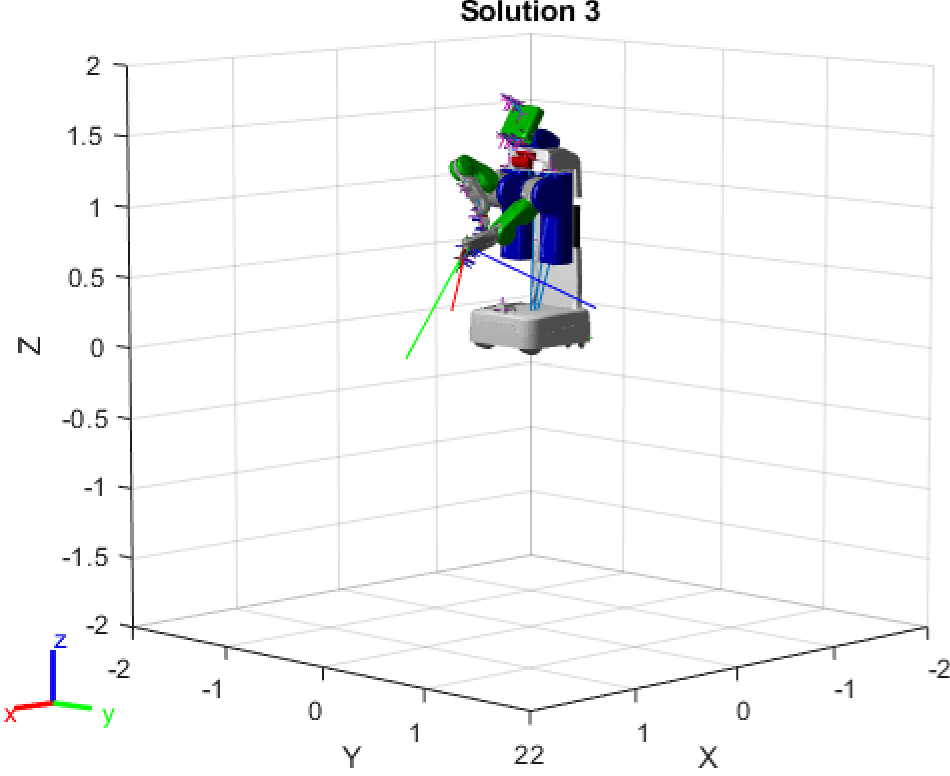
To display the target end-effector pose in the world frame, get the transformation from the base of the robot model, rather than the base of the kinematic group. Display all of the generated IK solutions by specifying the indices for your kinematic group IK solution in the configuration vector used with the `show` function.

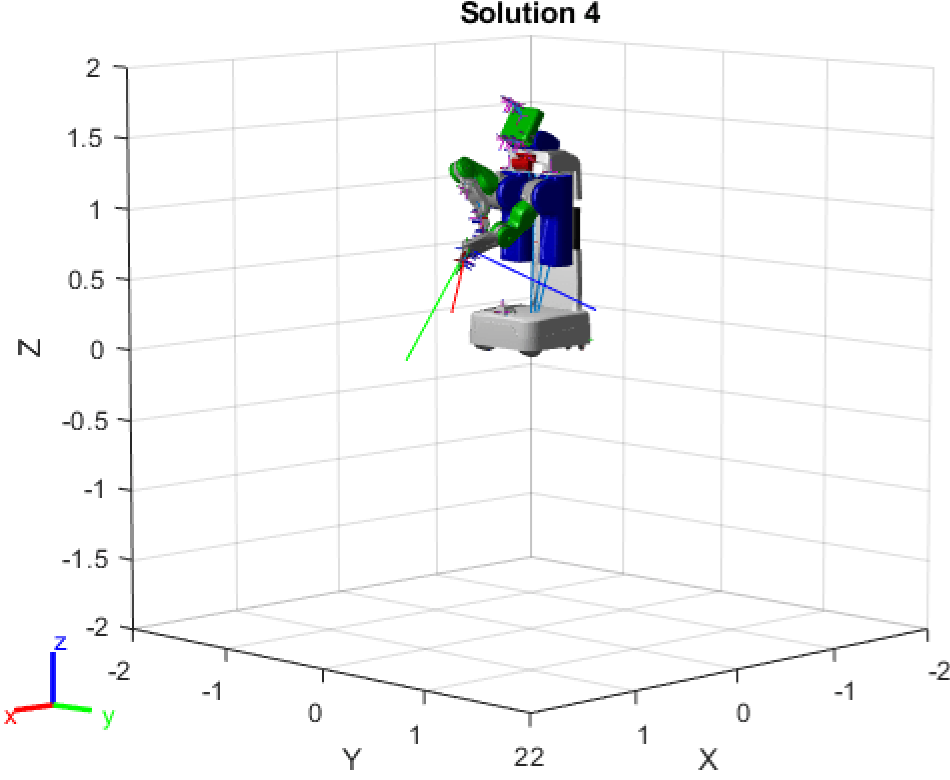
```
eeWorldPose = getTransform(robot,expConfig,eeBodyName);

generatedConfig = repmat(expConfig, size(ikConfig,1), 1);
generatedConfig(:,aik.KinematicGroupConfigIdx) = ikConfig;

for i = 1:size(ikConfig,1)
    figure;
    ax = show(robot,generatedConfig(i,:));
    hold all;
    plotTransforms(tform2trvec(eeWorldPose),tform2quat(eeWorldPose),'Parent',ax);
    title(['Solution ' num2str(i)]);
end
```

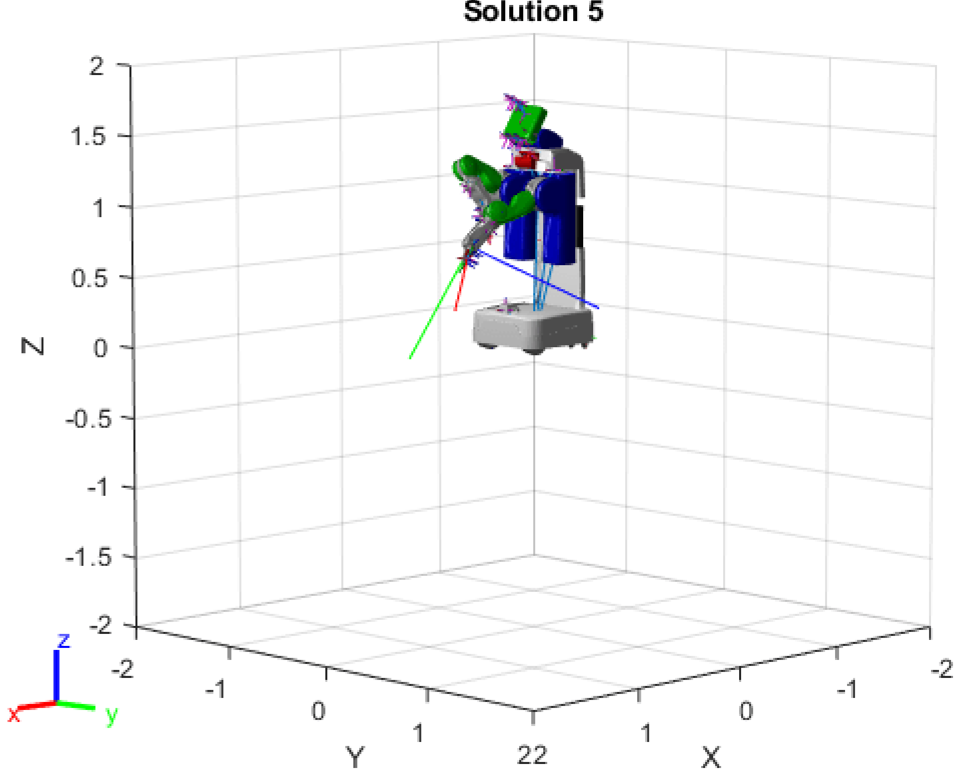

Solution 1
Z
X
Y
2
1.5
1
0.5
0
-0.5
-1
-1.5
-2
-2
-1
0
1
22
1
0
-1
-2
z
x
y

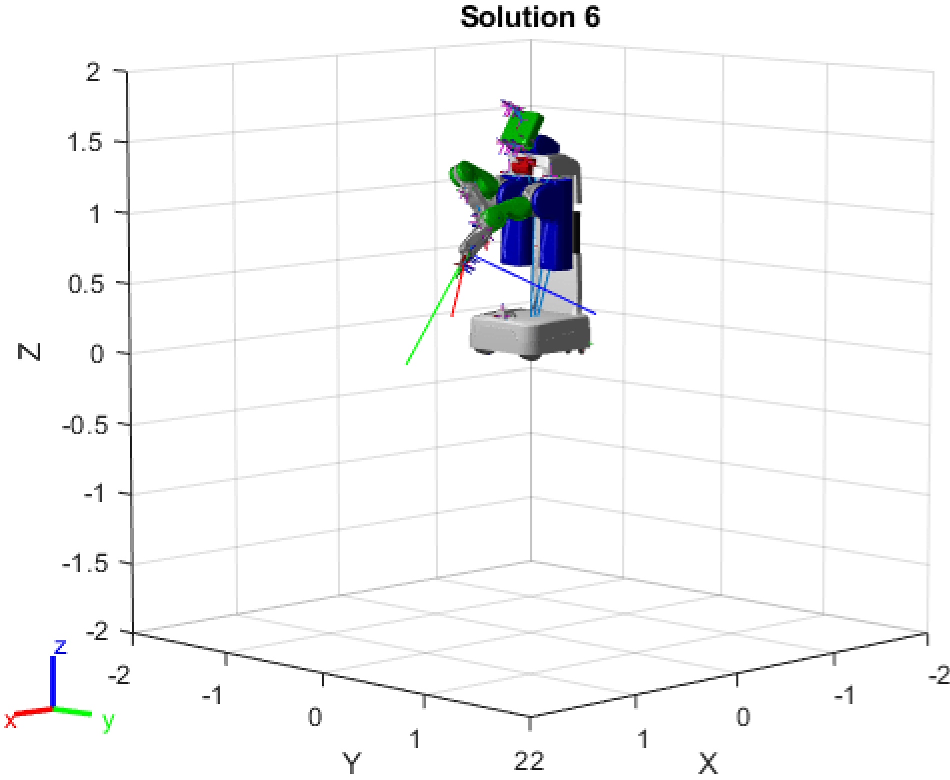


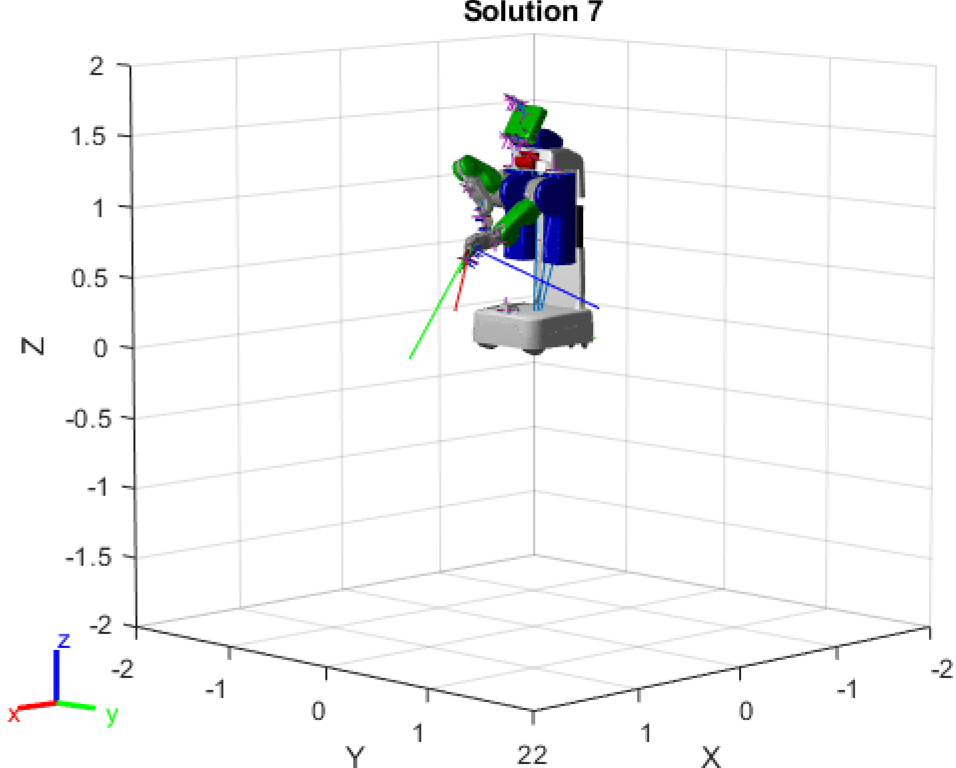
Solution 2
Z
Y
X
2
1.5
1
0.5
0
-0.5
-1
-1.5
-2
-2
-1
0
1
22
1
0
-1
-2
z
x
y


Solution 3
2
1.5
1
0.5
0
-0.5
-1
-1.5
-2
Z
-2
-1
0
1
22
Y
1
0
-1
-2
X
z
x
y

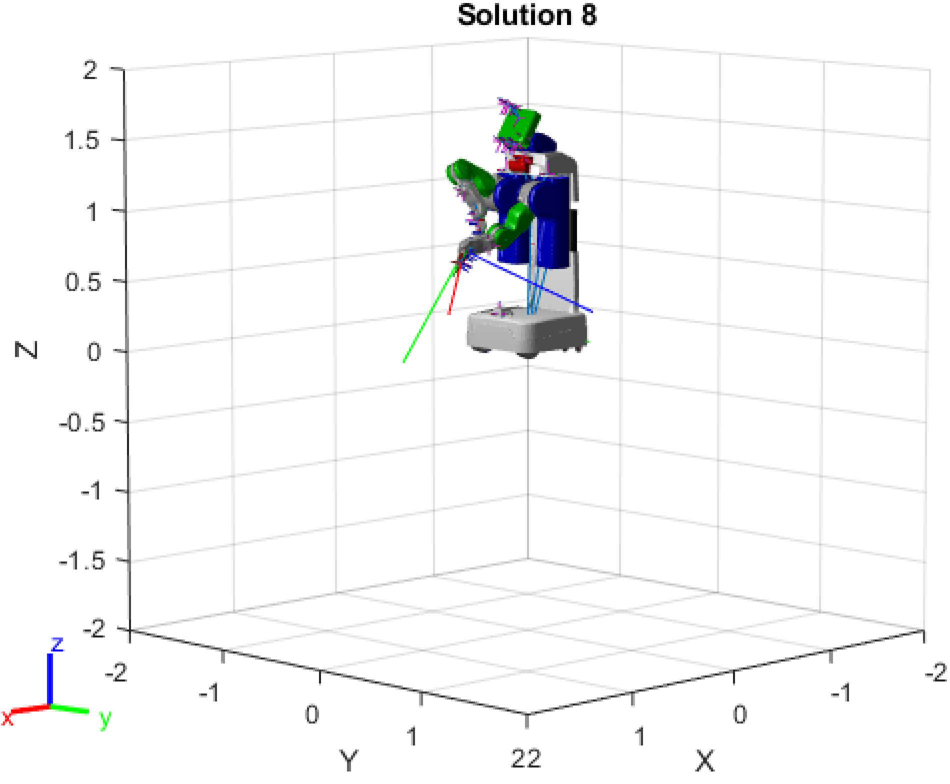

Solution 4
Z
Y
X
z
x
y
2
1.5
1
0.5
0
-0.5
-1
-1.5
-2
-2
-1
0
1
22
1
0
-1
-2


Solution 5
2
1.5
1
0.5
0
-0.5
-1
-1.5
-2
Z
-2
-1
0
1
2
Y
2
1
0
-1
-2
X
z
x
y


Solution 6
Z
Y
X
2
1.5
1
0.5
0
-0.5
-1
-1.5
-2
-2
-1
0
1
22
1
0
-1
-2
z
x
y



Solution 7
Z
Y
X
2
1.5
1
0.5
0
-0.5
-1
-1.5
-2
-2
-1
0
1
22
1
0
-1
-2
z
x
y



Input Arguments

`analyticalIK` — Analytical IK solver
`analyticalInverseKinematics` object

Analytical inverse kinematics solver, specified as an `analyticalInverseKinematics` object.

Output Arguments

`kinGroupDetails` — Kinematic group classification details
structure

Kinematic group classification details, returned as a structure with these fields:

- `KinematicGroup` — A structure that contains the base and end-effector body names of the kinematic group in the fields `BaseName` and `EndEffectorBodyName`, respectively.
- `Type` — A kinematic group classification type with the same format as that KinematicGroupType property of the `analyticalInverseKinematics` object.
- `IsIntersectionAxesMidpoint` — An *n*-element vector indicating whether each specific joint axis intersects with the preceding or following non-fixed joint. *n* is the number of non-fixed joints in the kinematic group.
- `MidpointAxisIntersections` — A 2-by-3-by-*n* array that stores the joint intersection points where each element of the third dimension corresponds to a single joint .For each channel of *n*,

the first row is the intersection point from the preceding joint to the joint represented by that channel. The second row is the intersection point from the joint to the following joint. The array gives intersection points as [*x y z*] coordinates relative to the base.

Version History

Introduced in R2020b

See Also

Objects
analyticalInverseKinematics | inverseKinematics | generalizedInverseKinematics | rigidBodyTree

Functions
loadrobot | importrobot | generateIKFunction

checkOccupancy

Check if locations are free or occupied

Syntax

```
occVal = checkOccupancy(map,xy)
occVal = checkOccupancy(map,xy,"local")
occVal = checkOccupancy(map,ij,"grid")
[occVal,validPts] = checkOccupancy( ___ )
```

```
occMatrix = checkOccupancy(map)
occMatrix = checkOccupancy(map,bottomLeft,matSize)
occMatrix = checkOccupancy(map,bottomLeft,matSize,"local")
occMatrix = checkOccupancy(map,topLeft,matSize,"grid")
```

Description

`occVal = checkOccupancy(map,xy)` returns an array of occupancy values at the `xy` locations in the world frame. Obstacle-free cells return 0, occupied cells return 1. Unknown locations, including outside the map, return `-1`.

`occVal = checkOccupancy(map,xy,"local")` returns an array of occupancy values at the `xy` locations in the local frame. The local frame is based on the `LocalOriginInWorld` property of the `map`.

`occVal = checkOccupancy(map,ij,"grid")` specifies `ij` grid cell indices instead of `xy` locations. Grid indices start at (1,1) from the top left corner.

`[occVal,validPts] = checkOccupancy( ___ )` also outputs an `n`-element vector of logical values indicating whether input coordinates are within the map limits.

`occMatrix = checkOccupancy(map)` returns a matrix that contains the occupancy status of each location. Obstacle-free cells return 0, occupied cells return 1. Unknown locations, including outside the map, return `-1`.

`occMatrix = checkOccupancy(map,bottomLeft,matSize)` returns a matrix of occupancy values by specifying the bottom-left corner location in world coordinates and the matrix size in meters.

`occMatrix = checkOccupancy(map,bottomLeft,matSize,"local")` returns a matrix of occupancy values by specifying the bottom-left corner location in local coordinates and the matrix size in meters.

`occMatrix = checkOccupancy(map,topLeft,matSize,"grid")` returns a matrix of occupancy values by specifying the top-left cell index in grid coordinates and the matrix size.

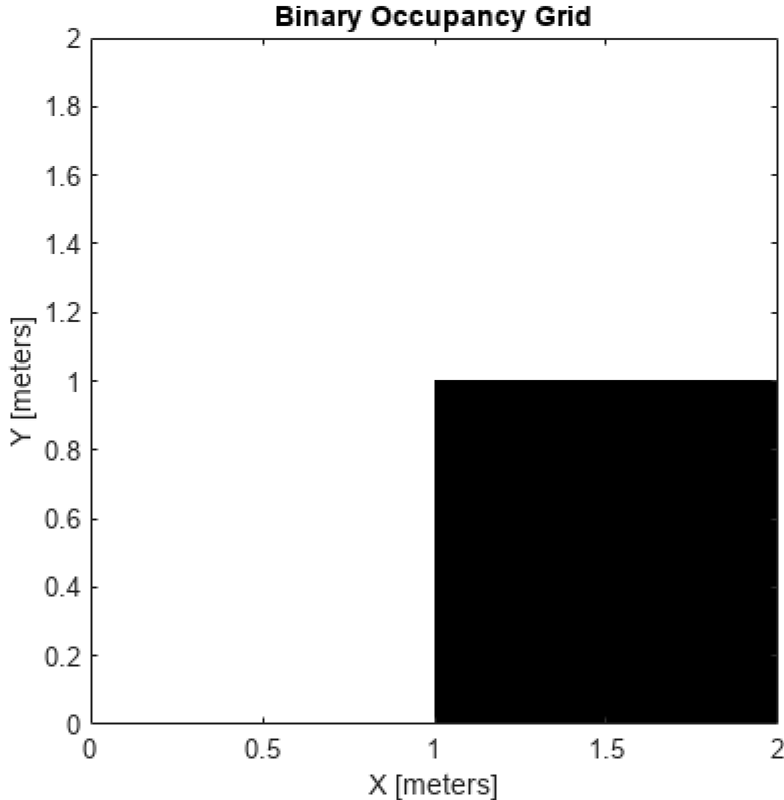
Examples

Get Occupancy Values and Check Occupancy Status

Access occupancy values and check their occupancy status based on the occupied and free thresholds of the `occupancyMap` object.

Create a matrix and populate it with values. Use this matrix to create an occupancy map.

```
p = zeros(20,20);
p(11:20,11:20) = ones(10,10);
map = binaryOccupancyMap(p,10);
show(map)
```



Get the occupancy of different locations and check their occupancy statuses. The occupancy status returns 0 for free space and 1 for occupied space. Unknown values return –1.

```
pocc = getOccupancy(map,[1.5 1]);
occupied = checkOccupancy(map,[1.5 1]);
pocc2 = getOccupancy(map,[5 5],'grid');
```

Input Arguments

map — Map representation
`binaryOccupancyMap` object

Map representation, specified as a `binaryOccupancyMap` object.

xy — Coordinates in the map
n-by-2 matrix

Coordinates in the map, specified as an *n*-by-2 matrix of `[x y]` pairs, where *n* is the number of coordinates. Coordinates can be world or local coordinates depending on the syntax.

Data Types: `double`

ij — Grid locations in the map
n-by-2 matrix

Grid locations in the map, specified as an *n*-by-2 matrix of `[i j]` pairs, where *n* is the number of locations. Grid locations are given as `[row col]`.

Data Types: `double`

bottomLeft — Location of output matrix in world or local
two-element vector | `[xCoord yCoord]`

Location of bottom left corner of output matrix in world or local coordinates, specified as a two-element vector, `[xCoord yCoord]`. Location is in world or local coordinates based on syntax.

Data Types: `double`

matSize — Output matrix size
two-element vector | `[xLength yLength]` | `[gridRow gridCol]`

Output matrix size, specified as a two-element vector, `[xLength yLength]`, or `[gridRow gridCol]`. Size is in world, local, or grid coordinates based on syntax.

Data Types: `double`

topLeft — Location of grid
two-element vector | `[iCoord jCoord]`

Location of top left corner of grid, specified as a two-element vector, `[iCoord jCoord]`.

Data Types: `double`

Output Arguments

occVal — Occupancy values
n-by-1 column vector

Occupancy values, returned as an *n*-by-1 column vector equal in length to `xy` or `ij` input. Occupancy values can be obstacle free (0), occupied (1), or unknown (-1).

validPts — Valid map locations
n-by-1 column vector

Valid map locations, returned as an *n*-by-1 column vector equal in length to `xy` or `ij`. Locations inside the map return a value of 1. Locations outside the map limits return a value of 0.

occMatrix — Matrix of occupancy values
matrix

Matrix of occupancy values, returned as matrix with size equal to `matSize` or the size of your `map`. Occupancy values can be obstacle free (0), occupied (1), or unknown (-1).

Version History

Introduced in R2019b

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`binaryOccupancyMap` | `getOccupancy` | `occupancyMap`

copy

Create copy of binary occupancy map

Syntax

`copyMap = copy(map)`

Description

`copyMap = copy(map)` creates a deep copy of the `binaryOccupancyMap` object with the same properties.

Examples

Copy Binary Occupancy Grid Map

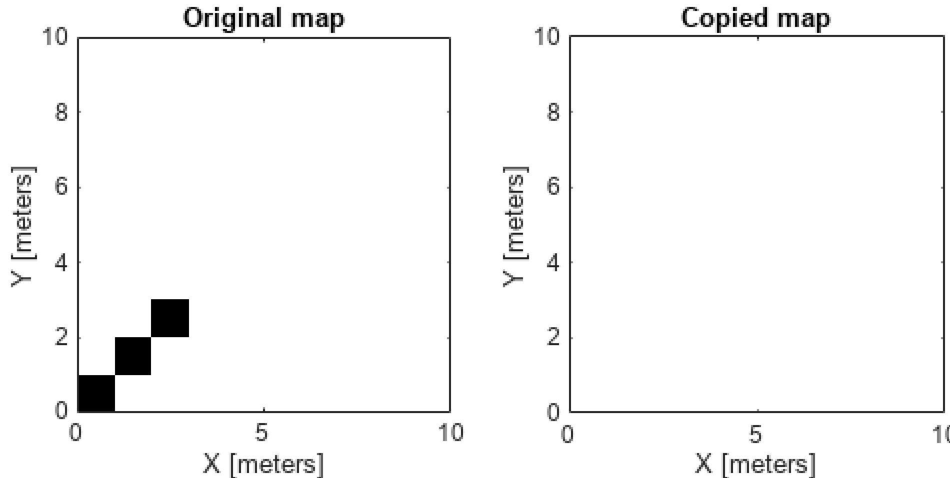
Copy a `binaryOccupancyMap` object. Once copied, the original object can be modified without affecting the copied map.

Create an occupancy map with zeros for an empty map.

```
p = zeros(10);
map = binaryOccupancyMap(p);
```

Copy the occupancy map. Modify the original map. The copied map is not modified. Plot the two maps side by side.

```
mapCopy = copy(map);
setOccupancy(map,[1:3;1:3]',ones(3,1));
subplot(1,2,1)
show(map)
title('Original map')
subplot(1,2,2)
show(mapCopy)
title('Copied map')
```



Input Arguments

`map` — Map representation
`binaryOccupancyMap` object

Map representation, specified as a `binaryOccupancyMap` object. This object represents the environment of the vehicle.

Output Arguments

`copyMap` — Copied map representation
`binaryOccupancyMap` object

Copied map representation, returned as a `binaryOccupancyMap` object. The properties are the same as the input object, `map`, but the copy has a different object handle.

Version History

Introduced in R2015a

See Also

`binaryOccupancyMap` | `occupancyMap`

getOccupancy

Get occupancy value of locations

Syntax

```
occVal = getOccupancy(map,xy)
occVal = getOccupancy(map,xy,"local")
occVal = getOccupancy(map,ij,"grid")
[occVal,validPts] = getOccupancy( ___ )
```

```
occMatrix = getOccupancy(map)
occMatrix = getOccupancy(map,bottomLeft,matSize)
occMatrix = getOccupancy(map,bottomLeft,matSize,"local")
occMatrix = getOccupancy(map,topLeft,matSize,"grid")
```

Description

`occVal = getOccupancy(map,xy)` returns an array of occupancy values at the `xy` locations in the world frame. Unknown locations, including outside the map, return `map.DefaultValue`.

`occVal = getOccupancy(map,xy,"local")` returns an array of occupancy values at the `xy` locations in the local frame.

`occVal = getOccupancy(map,ij,"grid")` specifies `ij` grid cell indices instead of `xy` locations.

`[occVal,validPts] = getOccupancy( ___ )` additionally outputs an `n`-element vector of logical values indicating whether input coordinates are within the map limits.

`occMatrix = getOccupancy(map)` returns all occupancy values in the map as a matrix.

`occMatrix = getOccupancy(map,bottomLeft,matSize)` returns a matrix of occupancy values by specifying the bottom-left corner location in world coordinates and the matrix size in meters.

`occMatrix = getOccupancy(map,bottomLeft,matSize,"local")` returns a matrix of occupancy values by specifying the bottom-left corner location in local coordinates and the matrix size in meters.

`occMatrix = getOccupancy(map,topLeft,matSize,"grid")` returns a matrix of occupancy values by specifying the top-left cell index in grid indices and the matrix size.

Examples

Insert Laser Scans into Binary Occupancy Map

Create an empty binary occupancy grid map.

```
map = binaryOccupancyMap(10,10,20);
```

Input pose of the vehicle, ranges, angles, and the maximum range of the laser scan.

```
pose = [5,5,0];
ranges = 3*ones(100,1);
angles = linspace(-pi/2,pi/2,100);
maxrange = 20;
```

Create a `lidarScan` object with the specified ranges and angles.

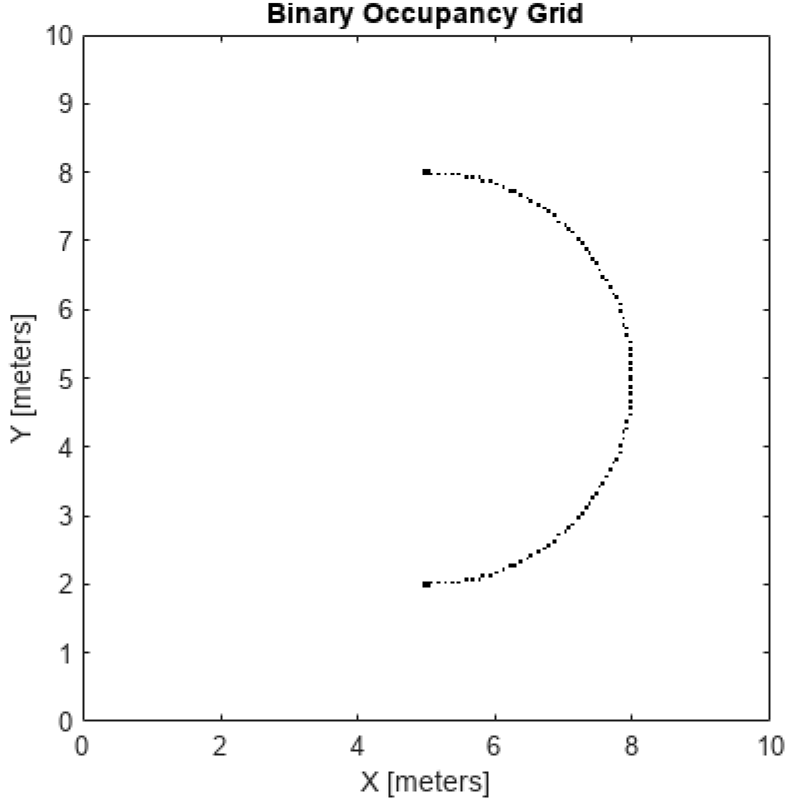
```
scan = lidarScan(ranges,angles);
```

Insert the laser scan data into the occupancy map.

```
insertRay(map,pose,scan,maxrange);
```

Show the map to see the results of inserting the laser scan.

```
show(map)
```



Check the occupancy of the spot directly in front of the vehicle.

```
getOccupancy(map,[8 5])
```

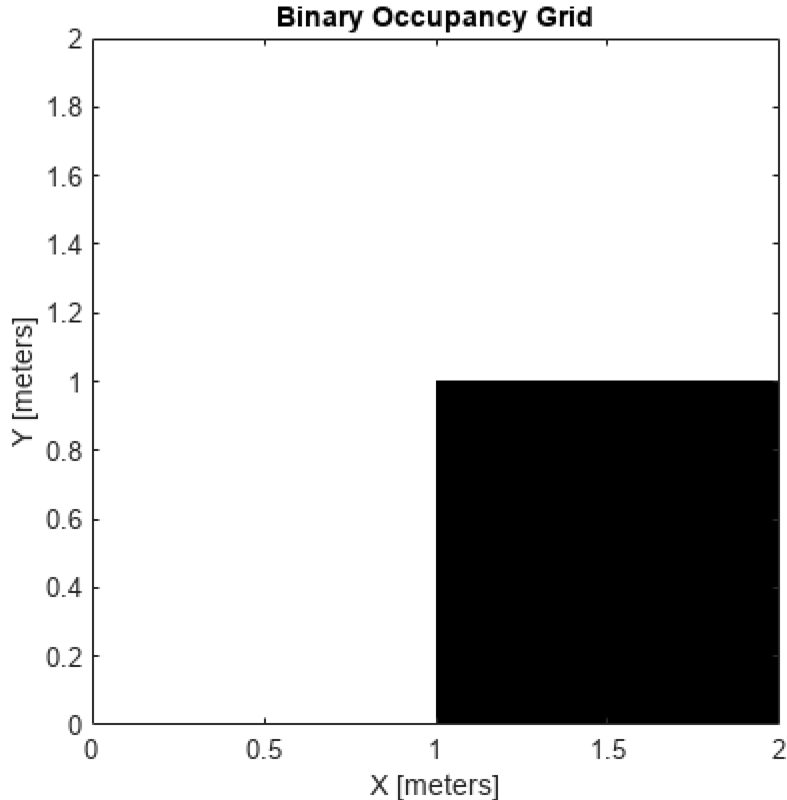
```
ans = logical
   1
```

Get Occupancy Values and Check Occupancy Status

Access occupancy values and check their occupancy status based on the occupied and free thresholds of the `occupancyMap` object.

Create a matrix and populate it with values. Use this matrix to create an occupancy map.

```
p = zeros(20,20);
p(11:20,11:20) = ones(10,10);
map = binaryOccupancyMap(p,10);
show(map)
```



Get the occupancy of different locations and check their occupancy statuses. The occupancy status returns 0 for free space and 1 for occupied space. Unknown values return –1.

```
pocc = getOccupancy(map,[1.5 1]);
occupied = checkOccupancy(map,[1.5 1]);
pocc2 = getOccupancy(map,[5 5],'grid');
```

Input Arguments

map — Map representation
`binaryOccupancyMap` object

Map representation, specified as a `binaryOccupancyMap` object. This object represents the environment of the vehicle.

xy — Coordinates in the map
n-by-2 matrix

Coordinates in the map, specified as an *n*-by-2 matrix of `[x y]` pairs, where *n* is the number of coordinates. Coordinates can be world or local coordinates depending on the syntax.

Data Types: `double`

ij — Grid locations in the map
n-by-2 matrix

Grid locations in the map, specified as an *n*-by-2 matrix of `[i j]` pairs, where *n* is the number of locations. Grid locations are given as `[row col]`.

Data Types: `double`

bottomLeft — Location of output matrix in world or local
two-element vector | `[xCoord yCoord]`

Location of bottom left corner of output matrix in world or local coordinates, specified as a two-element vector, `[xCoord yCoord]`. Location is in world or local coordinates based on syntax.

Data Types: `double`

matSize — Output matrix size
two-element vector | `[xLength yLength]` | `[gridRow gridCol]`

Output matrix size, specified as a two-element vector, `[xLength yLength]` or `[gridRow gridCol]`. The size is in world coordinates, local coordinates, or grid indices based on syntax.

Data Types: `double`

topLeft — Location of grid
two-element vector | `[iCoord jCoord]`

Location of top left corner of grid, specified as a two-element vector, `[iCoord jCoord]`.

Data Types: `double`

Output Arguments

occVal — Occupancy values
n-by-1 column vector

Occupancy values, returned as an *n*-by-1 column vector equal in length to `xy` or `ij`. Occupancy values can be obstacle free (0) or occupied (1).

validPts — Valid map locations
n-by-1 column vector

Valid map locations, returned as an *n*-by-1 column vector equal in length to `xy` or `ij`. Locations inside the map return a value of 1. Locations outside the map limits return a value of 0.

occMatrix — Matrix of occupancy values
matrix

Matrix of occupancy values, returned as matrix with size equal to `matSize` or the size of `map`.

Version History

Introduced in R2015a

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`binaryOccupancyMap` | `setOccupancy`

Topics

"Occupancy Grids"

grid2local

Convert grid indices to local coordinates

Syntax

```
xy = grid2local(map,ij)
```

Description

`xy = grid2local(map,ij)` converts a `[row col]` array of grid indices, `ij`, to an array of local coordinates, `xy`.

Examples

Convert Grid Indices in Binary Occupancy Map to Local Coordinates

Create an empty binary occupancy map with a width and height of 10 meters.

```
map = binaryOccupancyMap(10,10);
```

Get local coordinates from grid indices.

```
[i,j] = meshgrid(1:5);
xyLocal = grid2local(map,[i(:) j(:)]);
```

Input Arguments

`map` — Map representation
`binaryOccupancyMap` object

Map representation, specified as a `binaryOccupancyMap` object. This object represents the environment of the robot. The object contains a matrix grid with binary values indicating obstacles as `true` (`1`) and free locations as `false` (`0`).

`ij` — Grid positions
n-by-2 vertical array

Grid positions, specified as an *n*-by-2 vertical array of `[i j]` pairs in `[rows cols]` format, where *n* is the number of grid positions.

Output Arguments

`xy` — Local coordinates
n-by-2 vertical array

Local coordinates, specified as an *n*-by-2 vertical array of `[x y]` pairs, where *n* is the number of local coordinates.

Version History

Introduced in R2019b

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`binaryOccupancyMap` | `world2grid`

grid2world

Convert grid indices to world coordinates

Syntax

```
xy = grid2world(map,ij)
```

Description

`xy = grid2world(map,ij)` converts a `[row col]` array of grid indices, `ij`, to an array of world coordinates, `xy`.

Examples

Convert Grid Indices in Binary Occupancy Map to World Coordinates

Create an empty binary occupancy map with a width and height of 10 meters.

```
map = binaryOccupancyMap(10,10);
```

Get world coordinates from grid indices.

```
[i,j] = meshgrid(1:5);
xyWorld = grid2world(map,[i(:) j(:)]);
```

Input Arguments

`map` — Map representation
`binaryOccupancyMap` object

Map representation, specified as a `binaryOccupancyMap` object. This object represents the environment of the robot. The object contains a matrix grid with binary values indicating obstacles as `true` (`1`) and free locations as `false` (`0`).

`ij` — Grid positions
n-by-2 vertical array

Grid positions, specified as an *n*-by-2 vertical array of `[i j]` pairs in `[rows cols]` format, where *n* is the number of grid positions.

Output Arguments

xy — World coordinates
n-by-2 vertical array

World coordinates, specified as an *n*-by-2 vertical array of `[x y]` pairs, where *n* is the number of world coordinates.

Version History

Introduced in R2015a

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`binaryOccupancyMap` | `world2grid` | `grid2local`

inflate

Inflate each occupied location

Syntax

```
inflate(map,radius)
inflate(map,gridradius,'grid')
```

Description

`inflate(map,radius)` inflates each occupied position of the `map` by the radius given in meters. `radius` is rounded up to the nearest cell equivalent based on the resolution of the map. Every cell within the radius is set to `true` (`1`).

`inflate(map,gridradius,'grid')` inflates each occupied position by the radius given in number of cells.

Examples

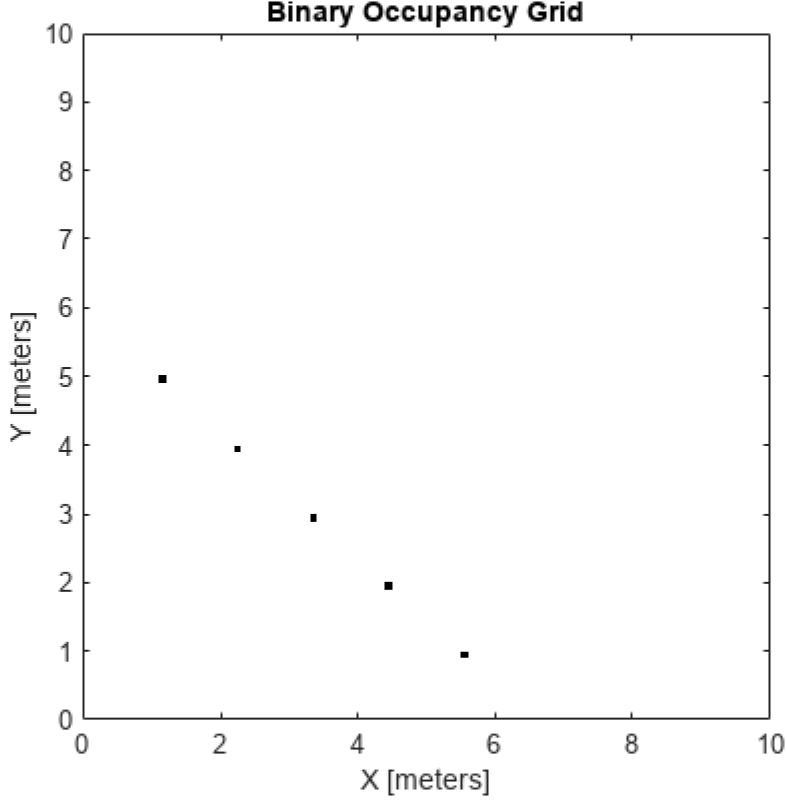
Create and Modify Binary Occupancy Grid

Create a 10m x 10m empty map.

```
map = binaryOccupancyMap(10,10,10);
```

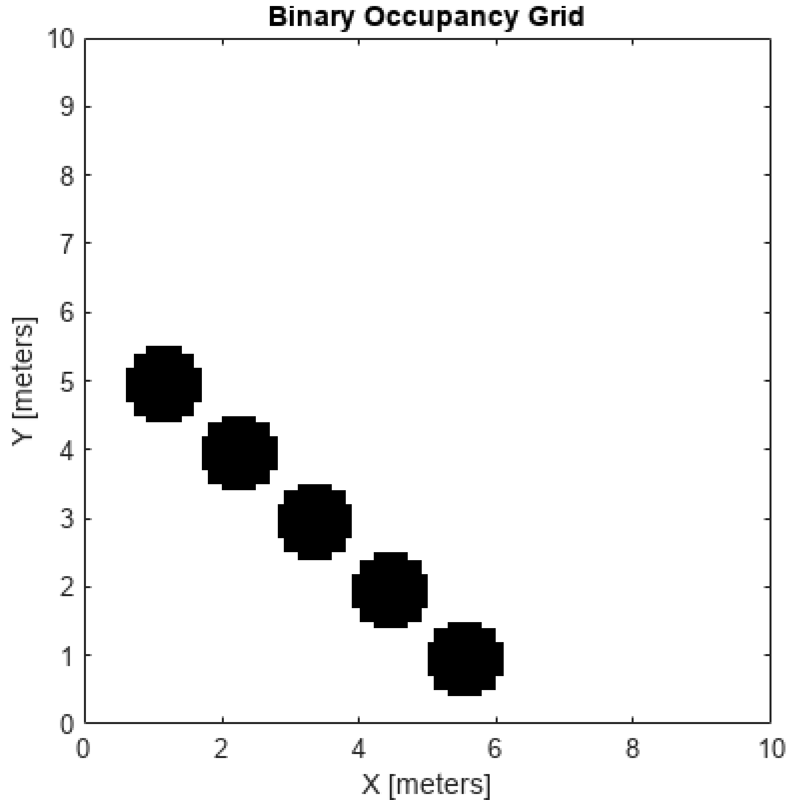
Set occupancy of world locations and show map.

```
x = [1.2; 2.3; 3.4; 4.5; 5.6];
y = [5.0; 4.0; 3.0; 2.0; 1.0];

setOccupancy(map, [x y], ones(5,1))
figure
show(map)
```



Inflate occupied locations by a given radius.

```
inflate(map, 0.5)
figure
show(map)
```

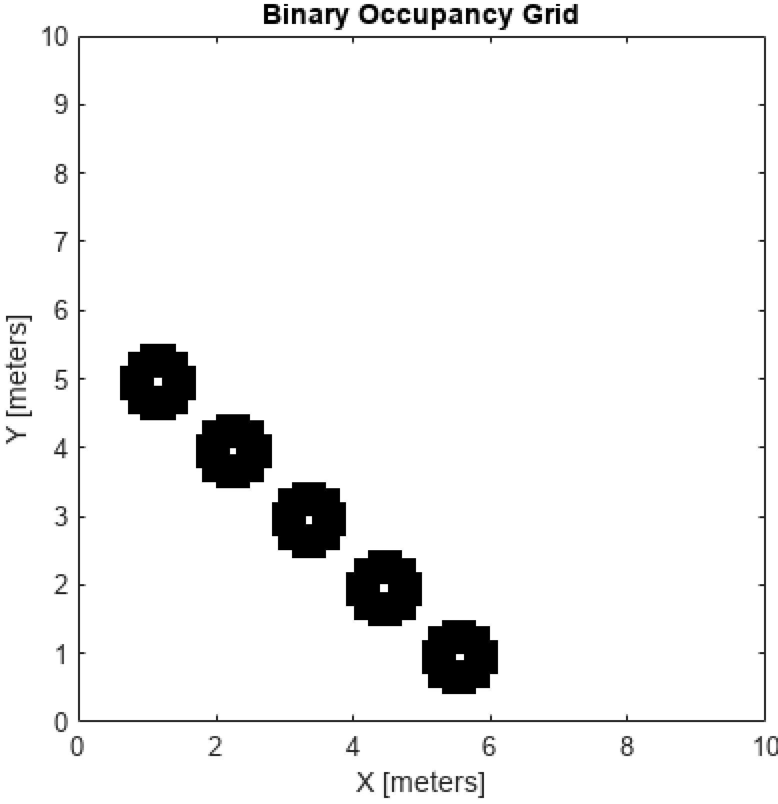


Get grid locations from world locations.

```
ij = world2grid(map, [x y]);
```

Set grid locations to free locations.

```
setOccupancy(map, ij, zeros(5,1), 'grid')
figure
show(map)
```



Input Arguments

`map` — Map representation
`binaryOccupancyMap` object

Map representation, specified as a `binaryOccupancyMap` object. This object represents the environment of the robot. The object contains a matrix grid with binary values indicating obstacles as `true` (`1`) and free locations as `false` (`0`).

`radius` — Dimension the defines how much to inflate occupied locations
scalar

Dimension that defines how much to inflate occupied locations, specified as a scalar. `radius` is rounded up to the nearest cell value.

Data Types: `double`

`gridradius` — Dimension the defines how much to inflate occupied locations
positive scalar

Dimension that defines how much to inflate occupied locations, specified as a positive scalar. `gridradius` is the number of cells to inflate the occupied locations.

Data Types: `double`

Version History

Introduced in R2015a

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`binaryOccupancyMap` | `setOccupancy`

Topics

"Occupancy Grids"

insertRay

Insert ray from laser scan observation

Syntax

```
insertRay(map,pose,scan,maxrange)
insertRay(map,pose,ranges,angles,maxrange)
insertRay(map,startpt,endpoints)
```

Description

`insertRay(map,pose,scan,maxrange)` inserts one or more lidar scan sensor observations in the occupancy grid, `map`, using the input `lidarScan` object, `scan`, to get ray endpoints. End point locations are updated with an occupied value. If the ranges are above `maxrange`, the ray endpoints are considered free space. All other points along the ray are treated as obstacle-free.

`insertRay(map,pose,ranges,angles,maxrange)` specifies the range readings as vectors defined by the input `ranges` and `angles`.

`insertRay(map,startpt,endpoints)` inserts observations between the line segments from the start point to the end points. The endpoints are updated are occupied space and other points along the line segments are updated as free space.

Examples

Insert Laser Scans into Binary Occupancy Map

Create an empty binary occupancy grid map.

```
map = binaryOccupancyMap(10,10,20);
```

Input pose of the vehicle, ranges, angles, and the maximum range of the laser scan.

```
pose = [5,5,0];
ranges = 3*ones(100,1);
angles = linspace(-pi/2,pi/2,100);
maxrange = 20;
```

Create a `lidarScan` object with the specified ranges and angles.

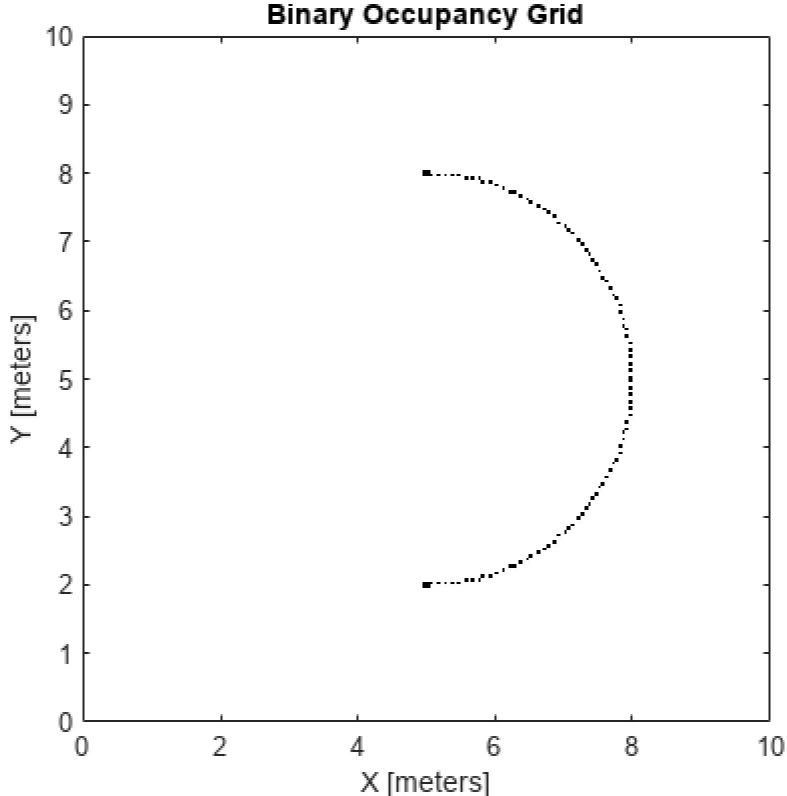
```
scan = lidarScan(ranges,angles);
```

Insert the laser scan data into the occupancy map.

```
insertRay(map,pose,scan,maxrange);
```

Show the map to see the results of inserting the laser scan.

```
show(map)
```



Check the occupancy of the spot directly in front of the vehicle.

```
getOccupancy(map,[8 5])
```

```
ans = logical
   1
```

Input Arguments

map — Map representation
binaryOccupancyMap object

Map representation, specified as a `binaryOccupancyMap` object. This object represents the environment of the robot. The object contains a matrix grid with binary values indicating obstacles as `true` (1) and free locations as `false` (0).

pose — Position and orientation of vehicle
three-element vector

Position and orientation of vehicle, specified as an `[x y theta]` vector. The vehicle pose is an *x* and *y* position with angular orientation *theta* (in radians) measured from the *x*-axis.

scan — Lidar scan readings
lidarScan object

Lidar scan readings, specified as a `lidarScan` object.

`ranges` — Range values from scan data
vector

Range values from scan data, specified as a vector of elements measured in meters. These range values are distances from a sensor at given `angles`. The vector must be the same length as the corresponding `angles` vector.

`angles` — Angle values from scan data
vector

Angle values from scan data, specified as a vector of elements measured in radians. These angle values correspond to the given `ranges`. The vector must be the same length as the corresponding `ranges` vector.

`maxrange` — Maximum range of sensor
scalar

Maximum range of laser range sensor, specified as a scalar in meters. Range values greater than or equal to `maxrange` are considered free along the whole length of the ray, up to `maxrange`.

`startpt` — Start point for rays
two-element vector

Start point for rays, specified as a two-element vector, `[x y]`, in the world coordinate frame. All rays are line segments that originate at this point.

`endpoints` — Endpoints for rays
n-by-2 matrix

Endpoints for rays, specified as an *n*-by-2 matrix of `[x y]` pairs in the world coordinate frame, where *n* is the length of `ranges` or `angles`. All rays are line segments that originate at `startpt`.

Version History

Introduced in R2019b

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

`occupancyMap` | `binaryOccupancyMap` | `lidarScan`

Topics
"Occupancy Grids"

local2grid

Convert local coordinates to grid indices

Syntax

```
ij = local2grid(map,xy)
```

Description

`ij = local2grid(map,xy)` converts an array of local coordinates, `xy`, to an array of grid indices, `ij` in `[row col]` format.

Examples

Convert Local Coordinates in Binary Occupancy Map to Grid Indices

Create an empty binary occupancy map with a width and height of 10 meters.

```
map = binaryOccupancyMap(10,10);
```

Get grid indices from local coordinates.

```
[xLocal,yLocal] = meshgrid(0:0.5:2);
ij = local2grid(map,[xLocal(:) yLocal(:)]);
```

Input Arguments

map — Map representation
`binaryOccupancyMap` object

Map representation, specified as a `binaryOccupancyMap` object. This object represents the environment of the vehicle.

xy — Local coordinates
n-by-2 matrix

Local coordinates, specified as an *n*-by-2 matrix of `[x y]` pairs, where *n* is the number of local coordinates.

Data Types: `double`

Output Arguments

ij — Grid positions
n-by-2 matrix

Grid positions, returned as an *n*-by-2 matrix of `[i j]` pairs in `[row col]` format, where *n* is the number of grid positions. The grid cell locations are counted from the top left corner of the grid.

Data Types: `double`

Version History

Introduced in R2019b

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`binaryOccupancyMap` | `occupancyMap` | `grid2local` | `grid2local`

Topics

"Occupancy Grids"

local2world

Convert local coordinates to world coordinates

Syntax

```
xyWorld = local2world(map,xy)
```

Description

`xyWorld = local2world(map,xy)` converts an array of local coordinates to world coordinates.

Examples

Convert Local Coordinates in Binary Occupancy Map to World Coordinates

Create an empty binary occupancy map with a width and height of 10 meters.

```
map = binaryOccupancyMap(10,10);
```

Get world coordinates from local coordinates.

```
[xLocal,yLocal] = meshgrid(0:0.5:2);
xyWorld = local2world(map,[xLocal(:) yLocal(:)]);
```

Input Arguments

`map` — Map representation
`binaryOccupancyMap` object

Map representation, specified as a `binaryOccupancyMap` object. This object represents the environment of the vehicle.

`xy` — Local coordinates
n-by-2 matrix

Local coordinates, specified as an *n*-by-2 matrix of `[x y]` pairs, where *n* is the number of local coordinates.

Data Types: `double`

Output Arguments

`xyWorld` — World coordinates
n-by-2 matrix

World coordinates, specified as an *n*-by-2 matrix of `[x y]` pairs, where *n* is the number of world coordinates.

Data Types: `double`

Version History

Introduced in R2019b

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`binaryOccupancyMap` | `grid2world` | `world2local` | `occupancyMap`

Topics

"Occupancy Grids"

move

Move map in world frame

Syntax

```
move(map,moveValue)
move(map,moveValue,Name,Value)
```

Description

`move(map,moveValue)` moves the local origin of the map to an absolute location, `moveValue`, in the world frame, and updates the map limits. Move values are truncated based on the resolution of the map. By default, newly revealed regions are set to `map.DefaultValue`.

`move(map,moveValue,Name,Value)` specifies additional options specified by one or more name-value pair arguments.

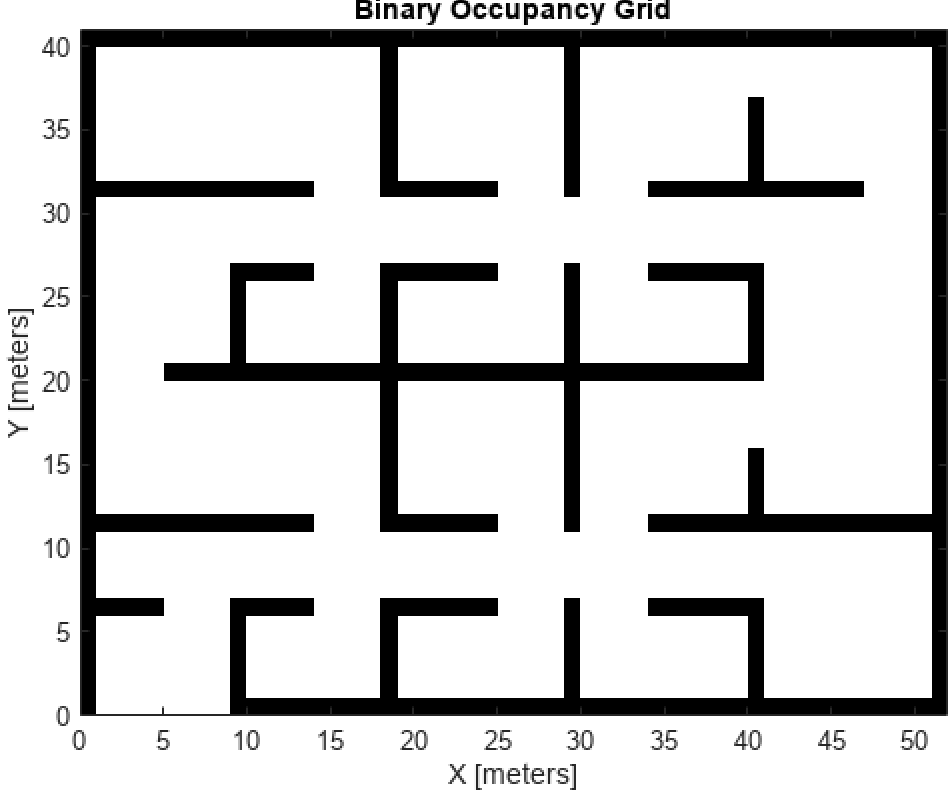
Examples

Move Local Map and Sync with World Map

This example shows how to move a local egocentric map and sync it with a larger world map. This process emulates a vehicle driving in an environment and getting updates on obstacles in the new areas.

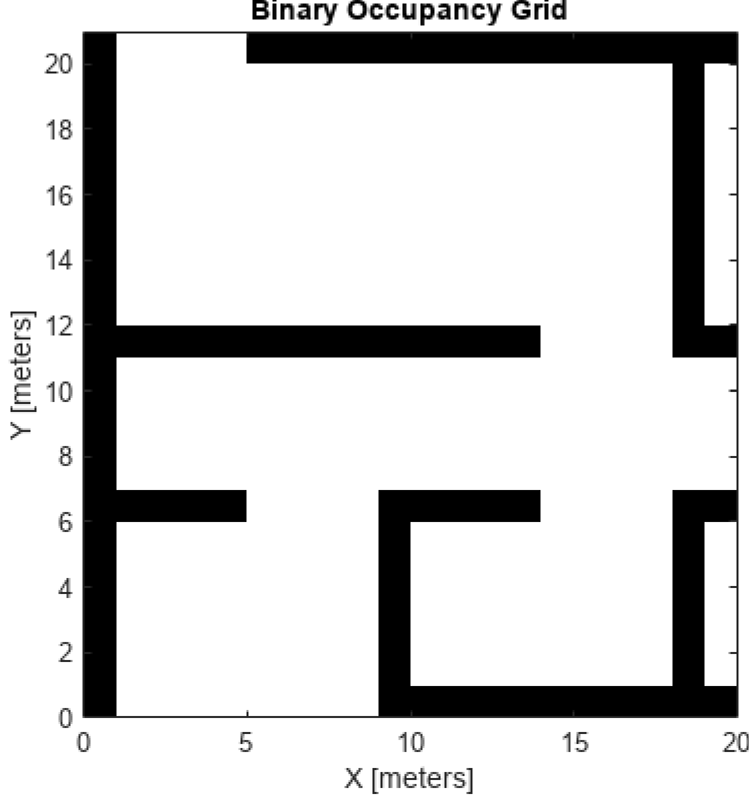
Load example maps. Create a binary occupancy map from the `complexMap`.

```
load exampleMaps.mat
map = binaryOccupancyMap(complexMap);
show(map)
```



Create a smaller local map.

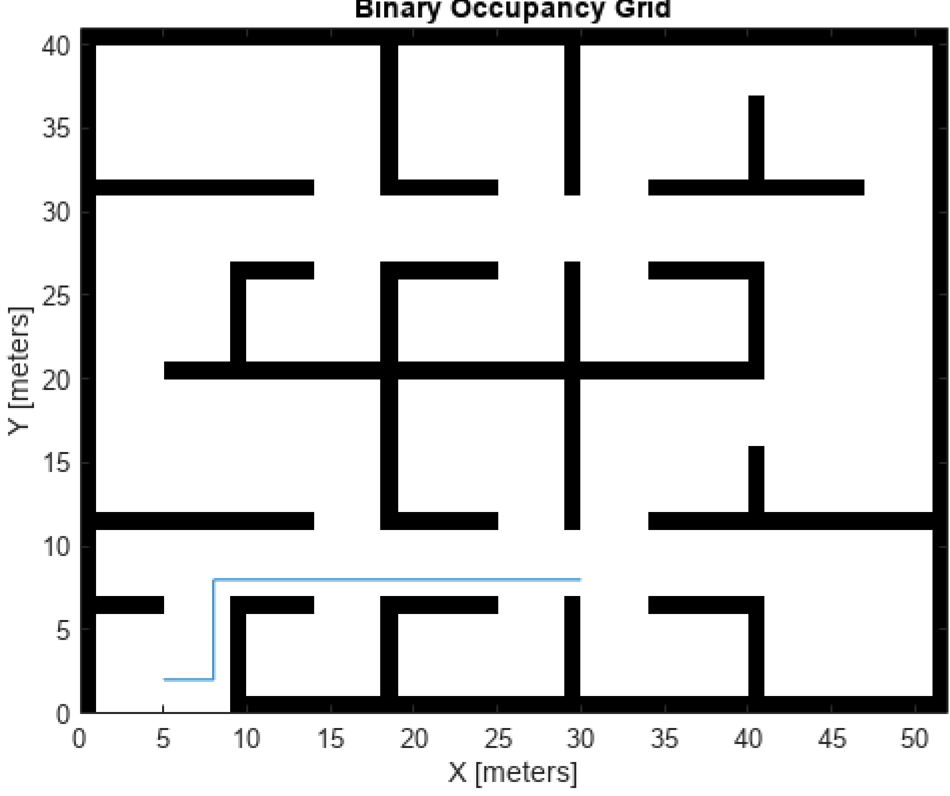
```
mapLocal = binaryOccupancyMap(complexMap(end-20:end,1:20));
show(mapLocal)
```



Follow a path planned in the world map and update the local map as you move your local frame.

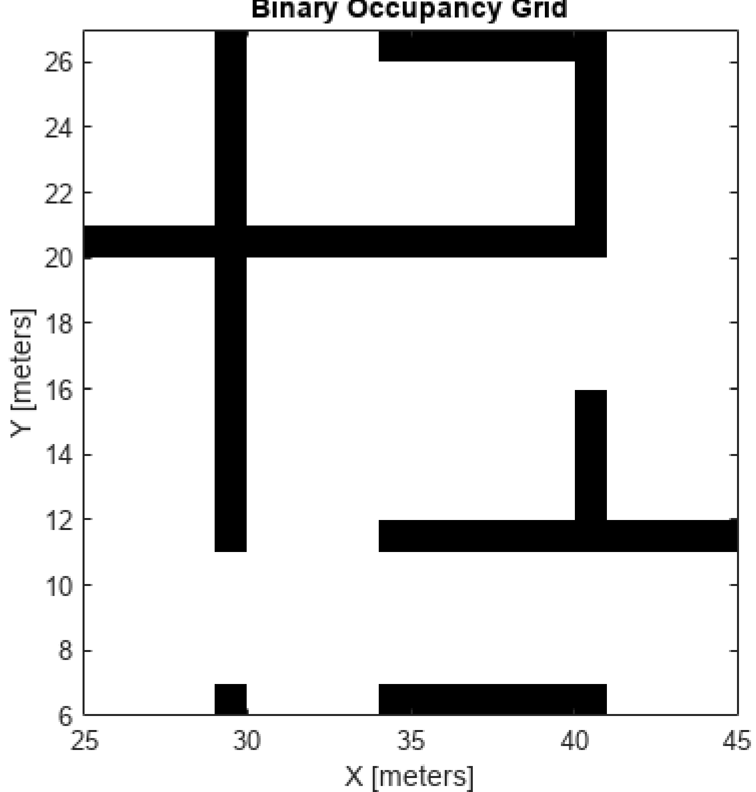
Specify path locations and plot on the map.

```
path = [5 2
        8 2
        8 8
        30 8];
show(map)
hold on
plot(path(:,1),path(:,2))
hold off
```



Create a loop for moving between points by the map resolution. Divide the difference between points by the map resolution to see how many incremental moves you can make.

```
for i = 1:length(path)-1
    moveAmount = (path(i+1,:)-path(i,:))/map.Resolution;
    for j = 1:abs(moveAmount(1)+moveAmount(2))
        moveValue = sign(moveAmount).*map.Resolution;
        move(mapLocal,moveValue, ...
            "MoveType","relative","SyncWith",map)

        show(mapLocal)
        drawnow limitrate
        pause(0.2)
    end
end
```



Input Arguments

map — Map representation
binaryOccupancyMap object

Map representation, specified as a `binaryOccupancyMap` object. This object represents the environment of the vehicle.

moveValue — Local map origin move value
[x y] vector

Local map origin move value, specified as an `[x y]` vector. By default, the value is an absolute location to move the local origin to in the world frame. Use the `MoveType` name-value pair to specify a relative move.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `'MoveType','relative'`

`MoveType` — Type of move
`'absolute'` (default) | `'relative'`

Type of move, specified as `'absolute'` or `'relative'`. For relative moves, specify a relative `[x y]` vector for `moveValue` based on your current local frame.

`FillValue` — Fill value for revealed locations
`0` (default) | `1`

Fill value for revealed locations because of the shifted map limits, specified as `0` or `1`.

`SyncWith` — Secondary map to sync with
`binaryOccupancyMap` object

Secondary map to sync with, specified as a `binaryOccupancyMap` object. Any revealed locations based on the move are updated with values in this map using the world coordinates.

Version History

Introduced in R2019b

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`binaryOccupancyMap` | `occupancyMap` | `occupancyMatrix`

occupancyMatrix

Convert occupancy grid to matrix

Syntax

```
mat = occupancyMatrix(map)
```

Description

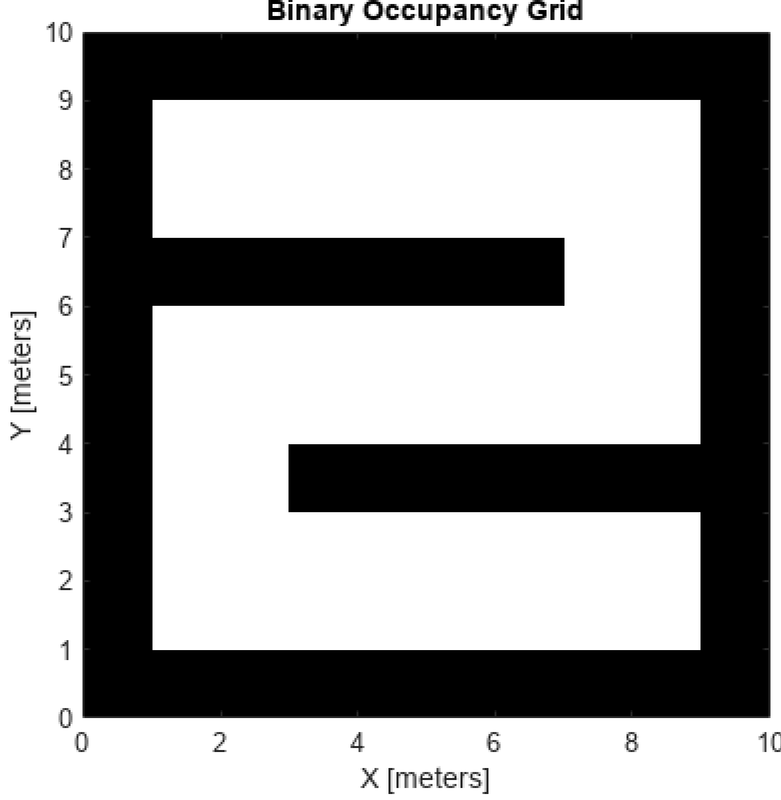
`mat = occupancyMatrix(map)` returns occupancy values stored in the occupancy grid object as a matrix.

Examples

Convert Binary Occupancy Map to Matrix

Generate a random 2-D maze map.

```
map = mapMaze(2,MapSize=[10 10],MapResolution=1);
show(map)
```



Convert the binary occupancy map to occupancy values matrix.

```
occupancyMatrix(map)
```

```
ans = 10x10 logical array

   1   1   1   1   1   1   1   1   1   1
   1   0   0   0   0   0   0   0   0   1
   1   0   0   0   0   0   0   0   0   1
   1   1   1   1   1   1   1   0   0   1
   1   0   0   0   0   0   0   0   0   1
   1   0   0   0   0   0   0   0   0   1
   1   0   0   1   1   1   1   1   1   1
   1   0   0   0   0   0   0   0   0   1
   1   0   0   0   0   0   0   0   0   1
   1   1   1   1   1   1   1   1   1   1
```

Input Arguments

map — Map representation
binaryOccupancyMap object

Map representation, specified as a binaryOccupancyMap object. This object represents the environment of the robot. The object contains a matrix grid with binary values indicating obstacles as true (1) and free locations as false (0).

Output Arguments

mat — Occupancy values
matrix

Occupancy values, returned as an *h*-by-*w* matrix, where *h* and *w* are defined by the two elements of the GridSize property of the occupancy grid object.

Data Types: double

Version History

Introduced in R2016b

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

binaryOccupancyMap | occupancyMap

Topics
"Occupancy Grids"

raycast

Compute cell indices along a ray

Syntax

```
[endpoints,midpoints] = raycast(map,pose,range,angle)
[endpoints,midpoints] = raycast(map,p1,p2)
```

Description

`[endpoints,midpoints] = raycast(map,pose,range,angle)` returns cell indices of the specified `map` for all cells traversed by a ray originating from the specified `pose` at the specified `angle` and `range` values. `endpoints` contains all indices touched by the end of the ray, with all other points included in `midpoints`.

`[endpoints,midpoints] = raycast(map,p1,p2)` returns the cell indices of the line segment between the two specified points.

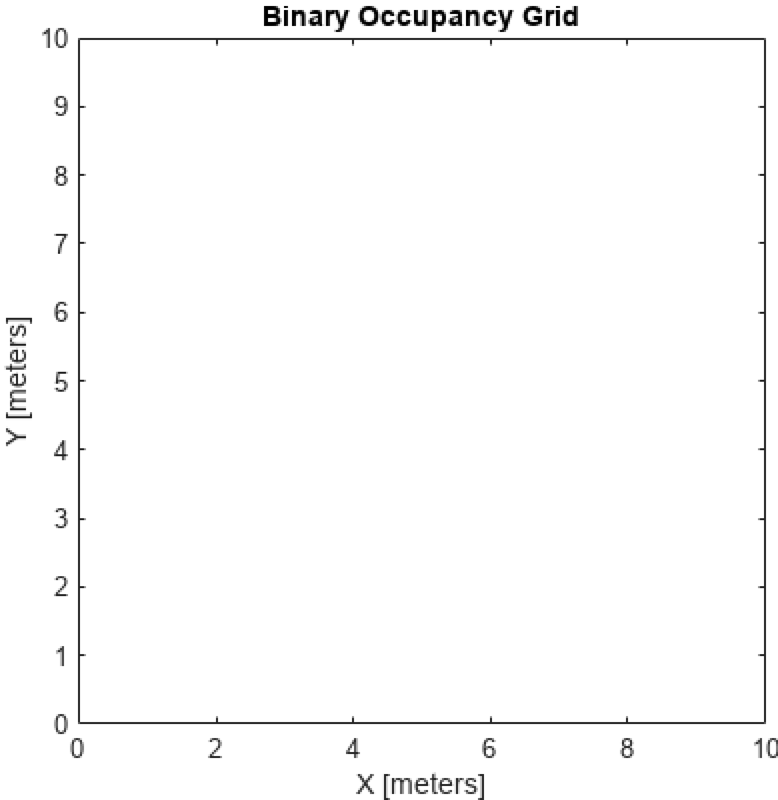
Examples

Compute Grid Cell Indices Along a Ray

Use the `raycast` function to generate cell indices for all cells traversed by a ray.

Create an empty map. A low-resolution map is used to illustrate the affected grid locations.

```
map = binaryOccupancyMap(10,10,1);
show(map)
```

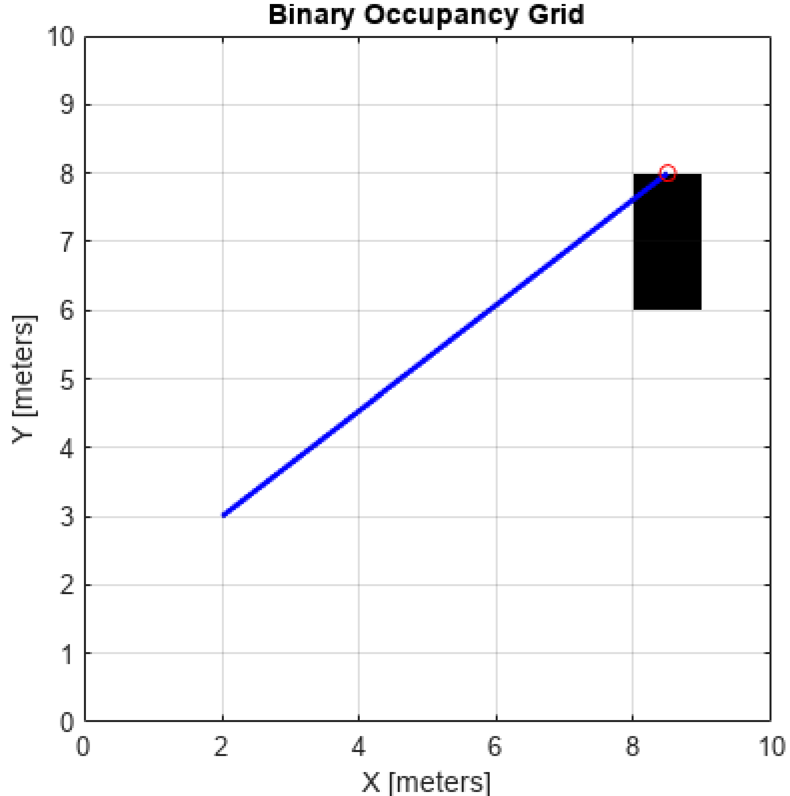


Get the grid indices of the midpoints and end points of a ray from `[2 3]` to `[8.5 8]`. Set occupancy values for these grid indices. Midpoints are treated as open space. Update endpoints with an occupied observation.

```
p1 = [2 3];
p2 = [8.5 8];
[endPts,midPts] = raycast(map,p1,p2);
setOccupancy(map,midPts,zeros(length(midPts),1),'grid');
setOccupancy(map,endPts,ones(length(endPts),1),'grid');
```

Plot the original ray over the map. Each grid cell touched by the line is updated. The starting point overlaps multiple cells, and the line touches the edge of certain cells, but all the cells are still updated.

```
show(map)
hold on
plot([p1(1) p2(1)],[p1(2) p2(2)],"-b","LineWidth",2)
plot(p2(1),p2(2),"or")
grid on
```



Input Arguments

map — Map representation
binaryOccupancyMap object

Map representation, specified as a `binaryOccupancyMap` object. This object represents the environment of the robot. The object contains a matrix grid with binary values indicating obstacles as `true` (`1`) and free locations as `false` (`0`).

pose — Position and orientation of sensor
three-element vector

Position and orientation of sensor, specified as an `[x y theta]` vector. The sensor pose is an *x* and *y* position with angular orientation *theta* (in radians) measured from the *x*-axis.

range — Range of ray
scalar

Range of ray, specified as a scalar in meters.

angle — Angle of ray
scalar

Angle of ray, specified as a scalar in radians. The angle value is for the corresponding `range`.

p1 — Starting point of ray
two-element vector

Starting point of ray, specified as an `[x y]` two-element vector. Points are defined with respect to the world-frame.

p2 — Endpoint of ray
two-element vector

Endpoint of ray, specified as an `[x y]` two-element vector. Points are defined with respect to the world-frame.

Output Arguments

`endpoints` — Endpoint grid indices
n-by-2 matrix

Endpoint indices, returned as an *n*-by-2 matrix of `[i j]` pairs, where *n* is the number of grid indices. The endpoints are where the `range` value hits at the specified `angle`. Multiple indices are returned when the endpoint lies on the boundary of multiple cells.

`midpoints` — Midpoint grid indices
n-by-2 matrix

Midpoint indices, returned as an *n*-by-2 matrix of `[i j]` pairs, where *n* is the number of grid indices. This argument includes all grid indices the ray intersects, excluding the endpoint.

Version History

Introduced in R2019b

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

`binaryOccupancyMap` | `insertRay` | `occupancyMap`

rayIntersection

Find intersection points of rays and occupied map cells

Syntax

`intersectionPts = rayIntersection(map,pose,angles,maxrange)`

Description

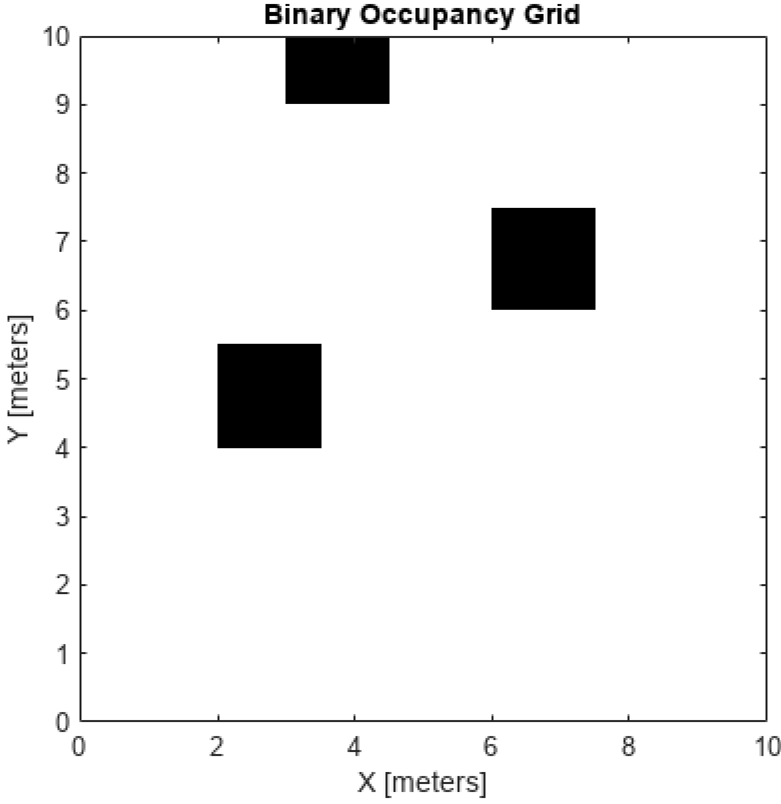
`intersectionPts = rayIntersection(map,pose,angles,maxrange)` returns intersection points of rays and occupied cells in the specified `map`. Rays emanate from the specified `pose` and `angles`. Intersection points are returned in the world coordinate frame. If there is no intersection up to the specified `maxrange`, `[NaN NaN]` is returned.

Examples

Get Ray Intersection Points on Occupancy Map

Create a binary occupancy grid map. Add obstacles and inflate them. A lower resolution map is used to illustrate the importance of the size of your grid cells. Show the map.

```
map = binaryOccupancyMap(10,10,2);
obstacles = [4 10; 3 5; 7 7];
setOccupancy(map,obstacles,ones(length(obstacles),1))
inflate(map,0.25)
show(map)
```



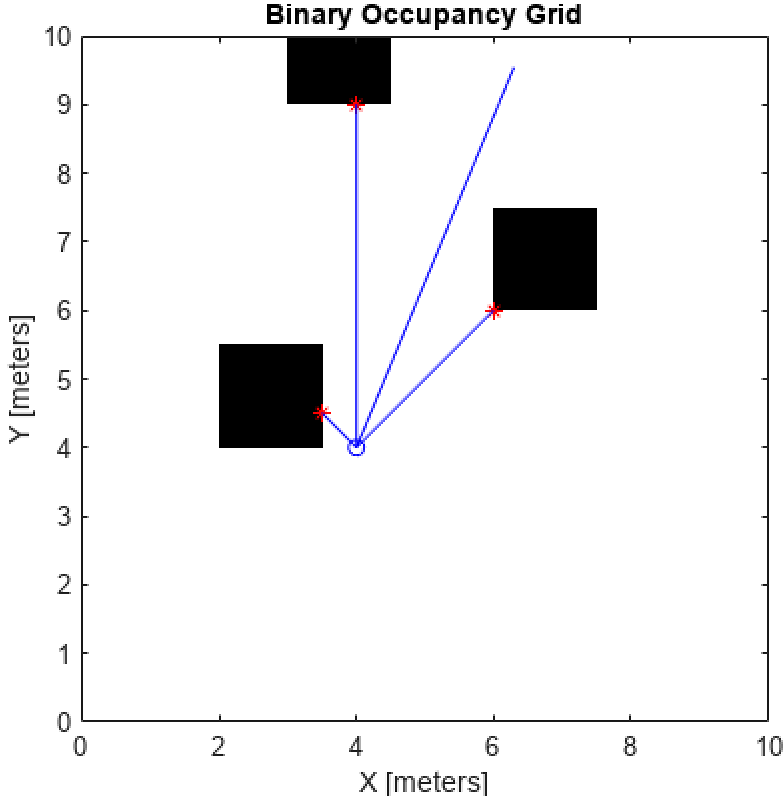
Find the intersection points of occupied cells and rays that emit from the given vehicle pose. Specify the max range and angles for these rays. The last ray does not intersect with an obstacle within the max range, so it has no collision point.

```
maxrange = 6;
angles = [pi/4,-pi/4,0,-pi/8];
vehiclePose = [4,4,pi/2];
intsectionPts = rayIntersection(map,vehiclePose,angles,maxrange)
```

```
intsectionPts = 4×2

    3.5000    4.5000
    6.0000    6.0000
    4.0000    9.0000
       NaN       NaN
```

Plot the intersection points and plot rays from the pose to the intersection points.

```
hold on
plot(intsectionPts(:,1),intsectionPts(:,2),'*r') % Intersection points
plot(vehiclePose(1),vehiclePose(2),'ob') % Vehicle pose
for i = 1:3
    plot([vehiclePose(1),intsectionPts(i,1)],...
        [vehiclePose(2),intsectionPts(i,2)],'-b') % Plot intersecting rays
end
plot([vehiclePose(1),vehiclePose(1)-6*sin(angles(4))],...
    [vehiclePose(2),vehiclePose(2)+6*cos(angles(4))],'-b') % No intersection ray
```



Input Arguments

`map` — Map representation
`binaryOccupancyMap` object

Map representation, specified as a `binaryOccupancyMap` object. This object represents the environment of the robot. The object contains a matrix grid with binary values indicating obstacles as `true` (`1`) and free locations as `false` (`0`).

`pose` — Position and orientation of sensor
three-element vector

Position and orientation of the sensor, specified as an `[x y theta]` vector. The sensor pose is an *x* and *y* position with angular orientation *theta* (in radians) measured from the *x*-axis.

`angles` — Ray angles emanating from sensor
vector

Ray angles emanating from the sensor, specified as a vector with elements in radians. These angles are relative to the specified sensor `pose`.

`maxrange` — Maximum range of sensor
scalar

Maximum range of laser range sensor, specified as a scalar in meters. Range values greater than or equal to `maxrange` are considered free along the whole length of the ray, up to `maxrange`.

Output Arguments

`intersectionPts` — Intersection points
n-by-2 matrix

Intersection points, returned as *n*-by-2 matrix of `[x y]` pairs in the world coordinate frame, where *n* is the length of `angles`.

Version History

Introduced in R2019b

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

`binaryOccupancyMap` | `occupancyMap`

Topics
"Occupancy Grids"
"Occupancy Grids"

setOccupancy

Set occupancy value of locations

Syntax

```
setOccupancy(map,xy,occval)
setOccupancy(map,xy,occval,"local")
setOccupancy(map,ij,occval,"grid")
validPts = setOccupancy( ___ )
```

```
setOccupancy(map,bottomLeft,inputMatrix)
setOccupancy(map,bottomLeft,inputMatrix,"local")
setOccupancy(map,topLeft,inputMatrix,"grid")
```

Description

`setOccupancy(map,xy,occval)` assigns occupancy values, `occval`, to the input array of world coordinates, `xy` in the occupancy grid, `map`. Each row of the array, `xy`, is a point in the world and is represented as an `[x y]` coordinate pair. `occval` is either a scalar or a single column array of the same length as `xy` . An occupied location is represented as `true` (`1`), and a free location is represented as `false` (`0`).

`setOccupancy(map,xy,occval,"local")` assigns occupancy values, `occval`, to the input array of local coordinates, `xy`, as local coordinates.

`setOccupancy(map,ij,occval,"grid")` assigns occupancy values, `occval`, to the input array of grid indices, `ij`, as `[rows cols]`.

`validPts = setOccupancy( ___ )` outputs an `n`-element vector of logical values indicating whether input coordinates are within the map limits.

`setOccupancy(map,bottomLeft,inputMatrix)` assigns a matrix of occupancy values by specifying the bottom-left corner location in world coordinates.

`setOccupancy(map,bottomLeft,inputMatrix,"local")` assigns a matrix of occupancy values by specifying the bottom-left corner location in local coordinates.

`setOccupancy(map,topLeft,inputMatrix,"grid")` assigns a matrix of occupancy values by specifying the top-left cell index in grid indices and the matrix size.

Examples

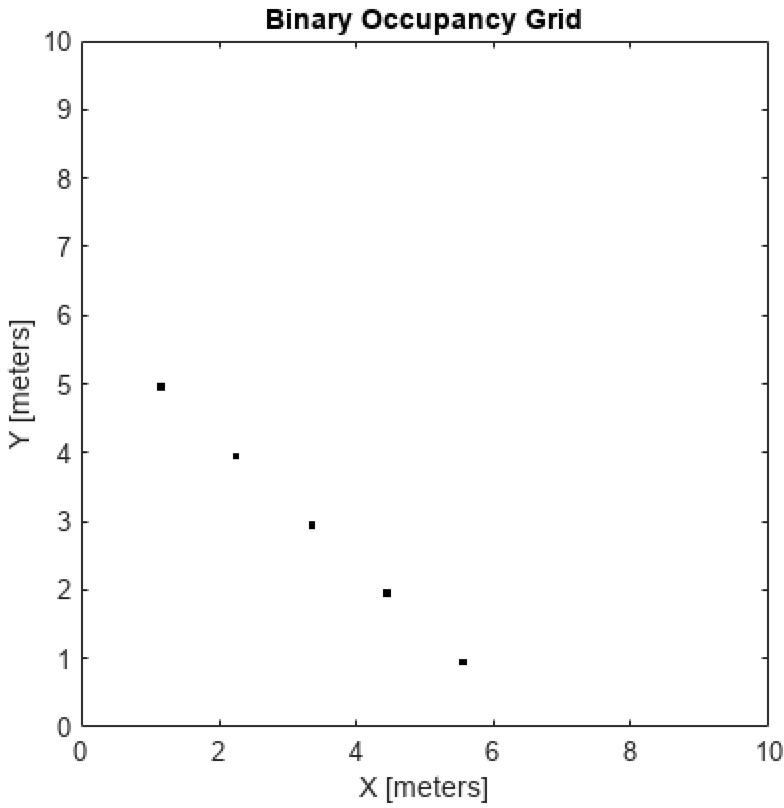
Create and Modify Binary Occupancy Grid

Create a 10m x 10m empty map.

```
map = binaryOccupancyMap(10,10,10);
```

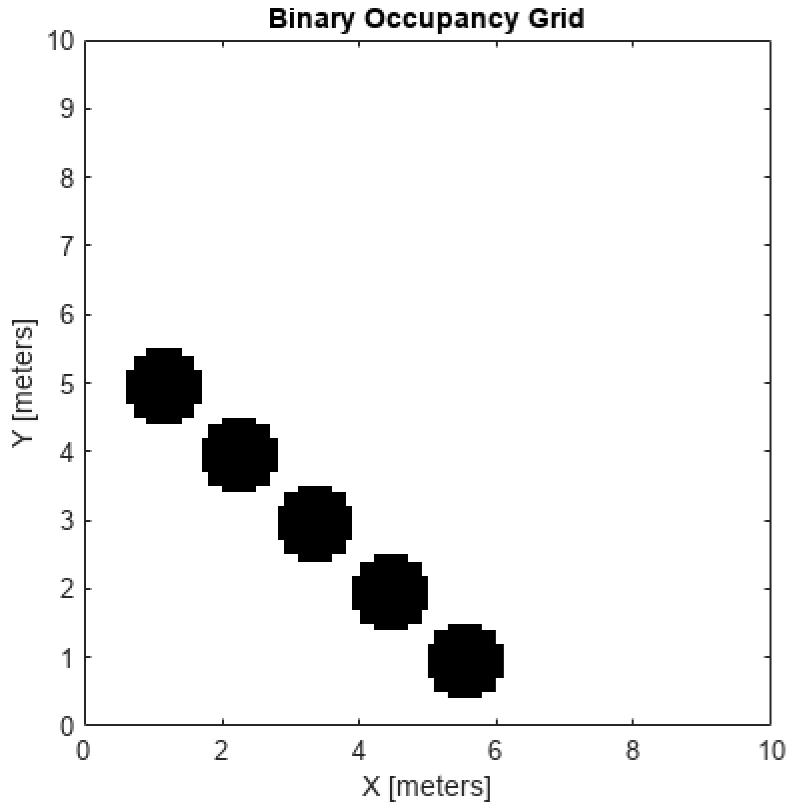
Set occupancy of world locations and show map.

```
x = [1.2; 2.3; 3.4; 4.5; 5.6];
y = [5.0; 4.0; 3.0; 2.0; 1.0];

setOccupancy(map, [x y], ones(5,1))
figure
show(map)
```



Inflate occupied locations by a given radius.

```
inflate(map, 0.5)
figure
show(map)
```

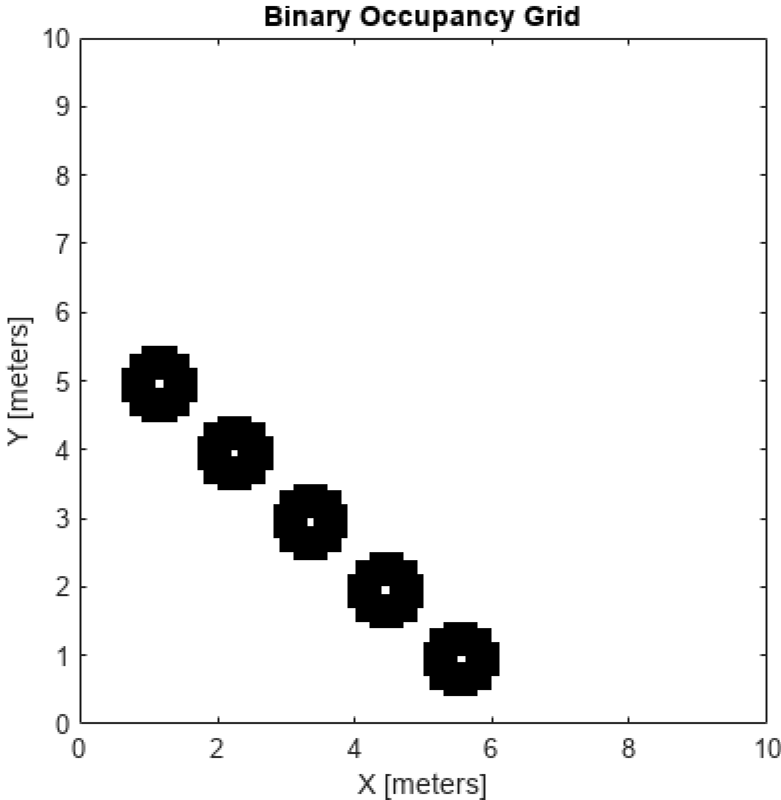


Get grid locations from world locations.

```
ij = world2grid(map, [x y]);
```

Set grid locations to free locations.

```
setOccupancy(map, ij, zeros(5,1), 'grid')
figure
show(map)
```



Input Arguments

`map` — Map representation
`binaryOccupancyMap` object

Map representation, specified as a `binaryOccupancyMap` object. This object represents the environment of the robot. The object contains a matrix grid with binary values indicating obstacles as `true` (`1`) and free locations as `false` (`0`).

`xy` — World coordinates
n-by-2 vertical array

World coordinates, specified as an *n*-by-2 vertical array of `[x y]` pairs, where *n* is the number of world coordinates.

Data Types: `double`

`ij` — Grid positions
n-by-2 vertical array

Grid positions, specified as an *n*-by-2 vertical array of `[i j]` pairs in `[rows cols]` format, where *n* is the number of grid positions.

Data Types: `double`

`occval` — Occupancy values
n-by-1 vertical array

Occupancy values of the same length as either `xy` or `ij`, returned as an *n*-by-1 vertical array, where *n* is the same *n* in either `xy` or `ij`. Values are given between 0 and 1 inclusively.

`inputMatrix` — Occupancy values
matrix

Occupancy values, specified as a matrix. Values are given between 0 and 1 inclusively.

`bottomLeft` — Location of output matrix in world or local
two-element vector | `[xCoord yCoord]`

Location of bottom left corner of output matrix in world or local coordinates, specified as a two-element vector, `[xCoord yCoord]`. Location is in world or local coordinates based on syntax.

Data Types: `double`

`topLeft` — Location of grid
two-element vector | `[iCoord jCoord]`

Location of top left corner of grid, specified as a two-element vector, `[iCoord jCoord]`.

Data Types: `double`

Output Arguments

`validPts` — Valid map locations
n-by-1 column vector

Valid map locations, returned as an *n*-by-1 column vector equal in length to `xy` or `ij`. Locations inside the map return a value of 1. Locations outside the map limits return a value of 0.

Version History

Introduced in R2015a

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

`binaryOccupancyMap` | `getOccupancy` | `occupancyMap`

show

Display binary occupancy map

Syntax

```
show(map)
show(map, "local")
show(map, "grid")
show( ___ ,Name,Value)
mapImage = show( ___ )
```

Description

`show(map)` displays the binary occupancy grid `map` in the current axes, with the axes labels representing the world coordinates.

`show(map, "local")` displays the binary occupancy grid `map` in the current axes, with the axes labels representing the local coordinates instead of world coordinates.

`show(map, "grid")` displays the binary occupancy grid `map` in the current axes, with the axes labels representing the grid coordinates.

`show( ___ ,Name,Value)` specifies additional options specified by one or more name-value pair arguments.

`mapImage = show( ___ )` returns the handle to the image object created by `show`.

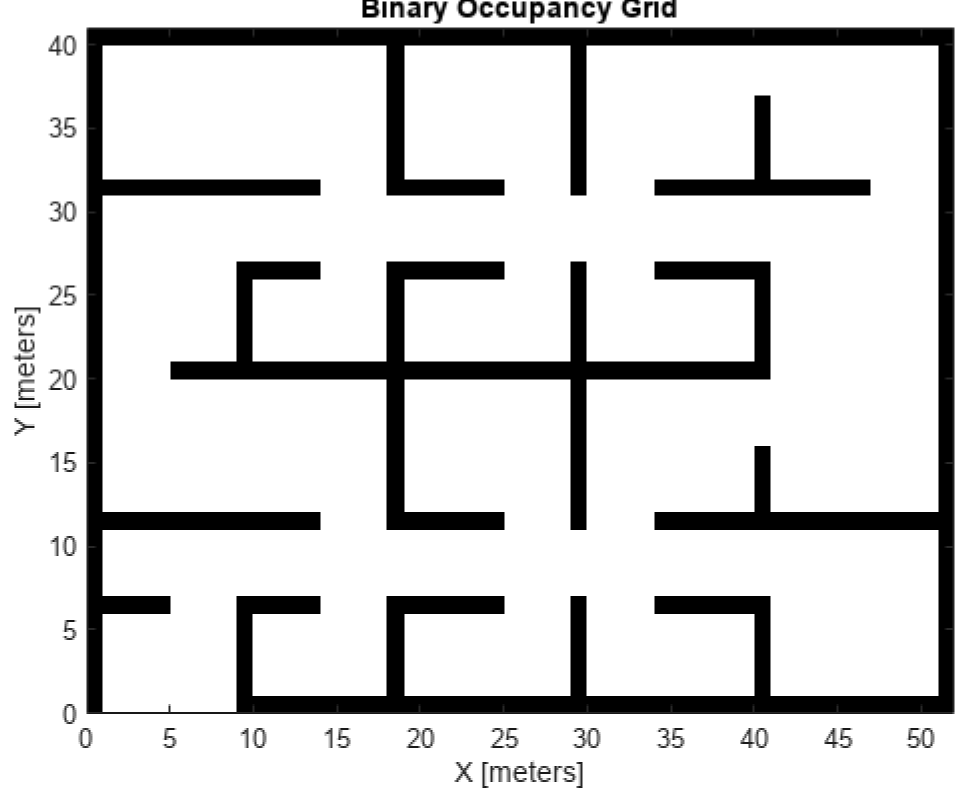
Examples

Move Local Map and Sync with World Map

This example shows how to move a local egocentric map and sync it with a larger world map. This process emulates a vehicle driving in an environment and getting updates on obstacles in the new areas.

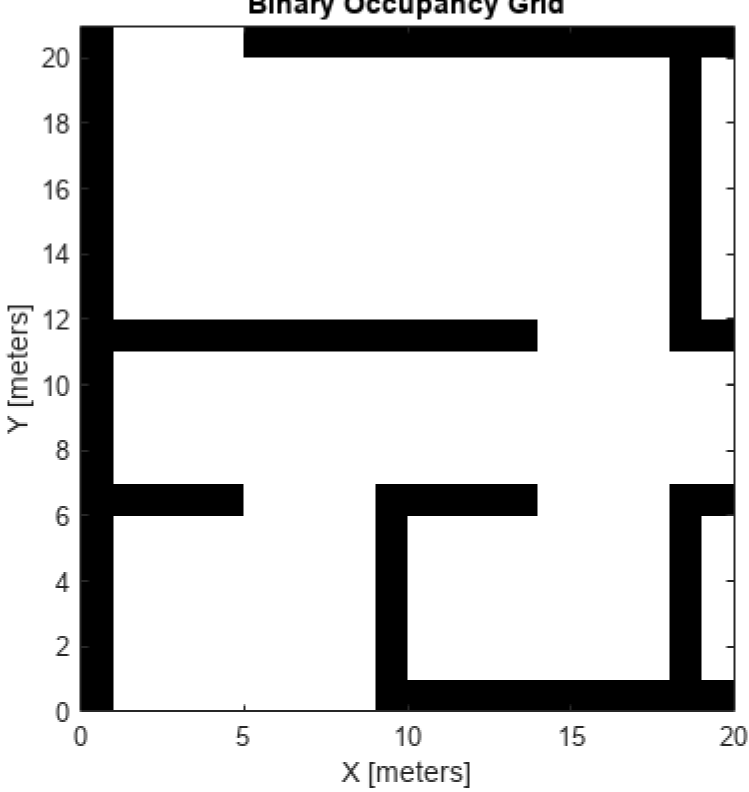
Load example maps. Create a binary occupancy map from the `complexMap`.

```
load exampleMaps.mat
map = binaryOccupancyMap(complexMap);
show(map)
```



Create a smaller local map.

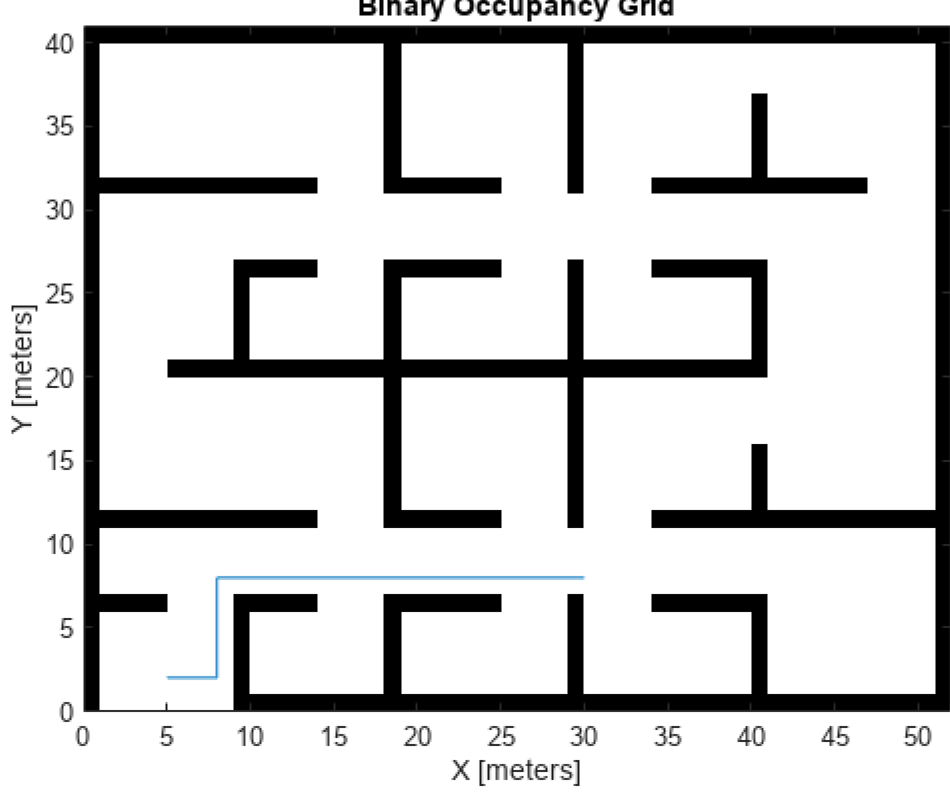
```
mapLocal = binaryOccupancyMap(complexMap(end-20:end,1:20));
show(mapLocal)
```



Follow a path planned in the world map and update the local map as you move your local frame.

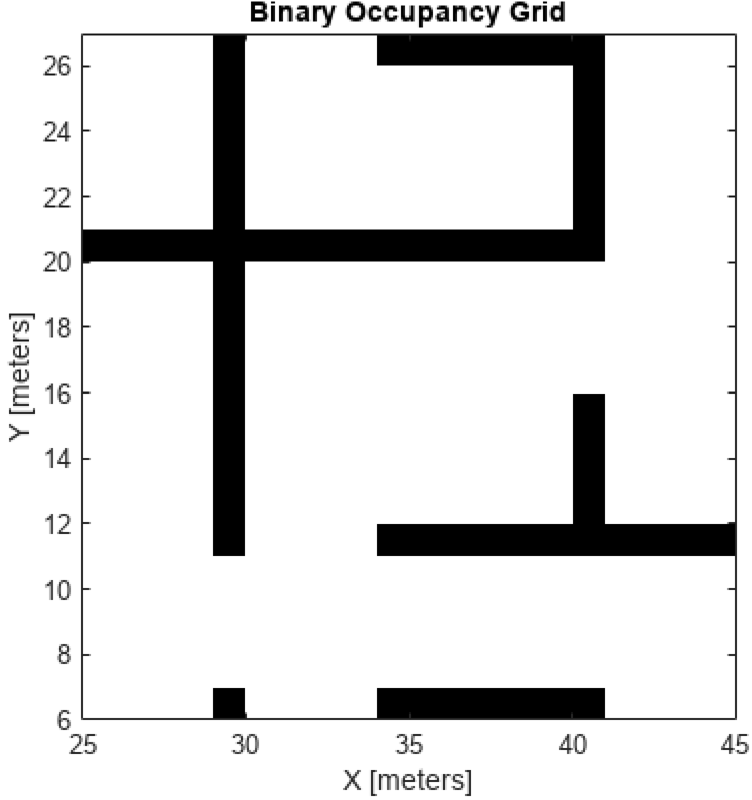
Specify path locations and plot on the map.

```
path = [5 2
        8 2
        8 8
        30 8];
show(map)
hold on
plot(path(:,1),path(:,2))
hold off
```



Create a loop for moving between points by the map resolution. Divide the difference between points by the map resolution to see how many incremental moves you can make.

```
for i = 1:length(path)-1
    moveAmount = (path(i+1,:)-path(i,:))/map.Resolution;
    for j = 1:abs(moveAmount(1)+moveAmount(2))
        moveValue = sign(moveAmount).*map.Resolution;
        move(mapLocal,moveValue, ...
            "MoveType","relative","SyncWith",map)

        show(mapLocal)
        drawnow limitrate
        pause(0.2)
    end
end
```



Input Arguments

map — Map representation
binaryOccupancyMap object

Map representation, specified as a binaryOccupancyMap object. This object represents the environment of the vehicle.

Name-Value Pair Arguments

Specify optional pairs of arguments as Name1=Value1,...,NameN=ValueN, where Name is the argument name and Value is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose Name *in quotes.*

Example: 'Parent',axHandle

Parent — Axes to plot the map
Axes object | UIAxes object

Axes to plot the map specified as either an Axes or UIAxesobject. See axes or uiaxes.

FastUpdate — Update existing map plot
0 (default) | 1

Update existing map plot, specified as `0` or `1`. If you previously plotted your map on your figure, set to `1` for a faster update to the figure. This is useful for updating the figure in a loop for fast animations.

Version History

Introduced in R2015a

See Also

`binaryOccupancyMap` | `occupancyMap`

syncWith

Sync map with overlapping map

Syntax

`mat = syncWith(map,sourcemap)`

Description

`mat = syncWith(map,sourcemap)` updates `map` with data from another `binaryOccupancyMap` object, `sourcemap`. Locations in `map` that are also found in `sourcemap` are updated. All other cells in `map` are set to `map.DefaultValue`.

Examples

Sync Map With an Overlapping Map

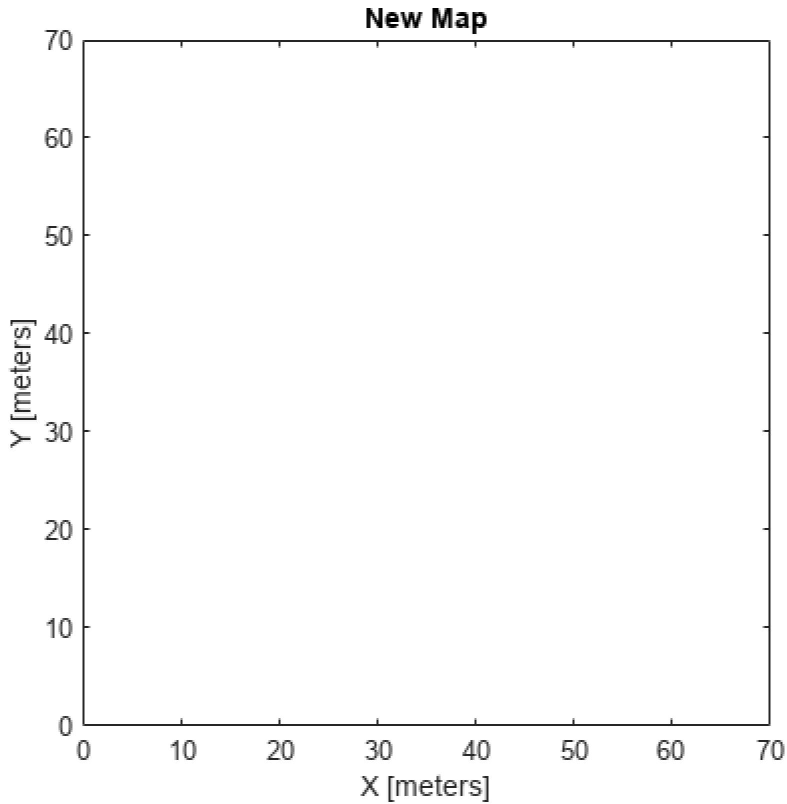
This example shows how to sync two overlapping maps using the `syncWith` function.

2-D occupancy maps are used to represent and visualize robot workspaces. In this example 2-D occupancy maps are created using existing map grid values stored inside `exampleMaps.mat`.

```
load('exampleMaps.mat');
```

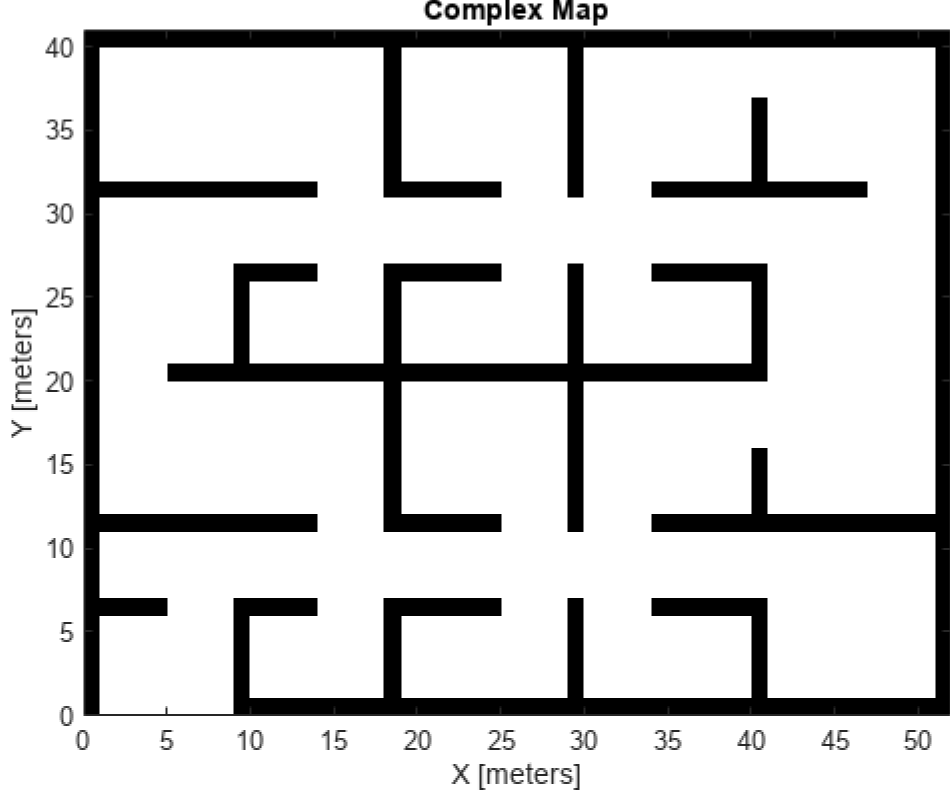
Create and display a new empty 2-D occupancy map object using `binaryOccupancyMap` function.

```
map1 = binaryOccupancyMap(70,70);
show(map1)
title('New Map')
```



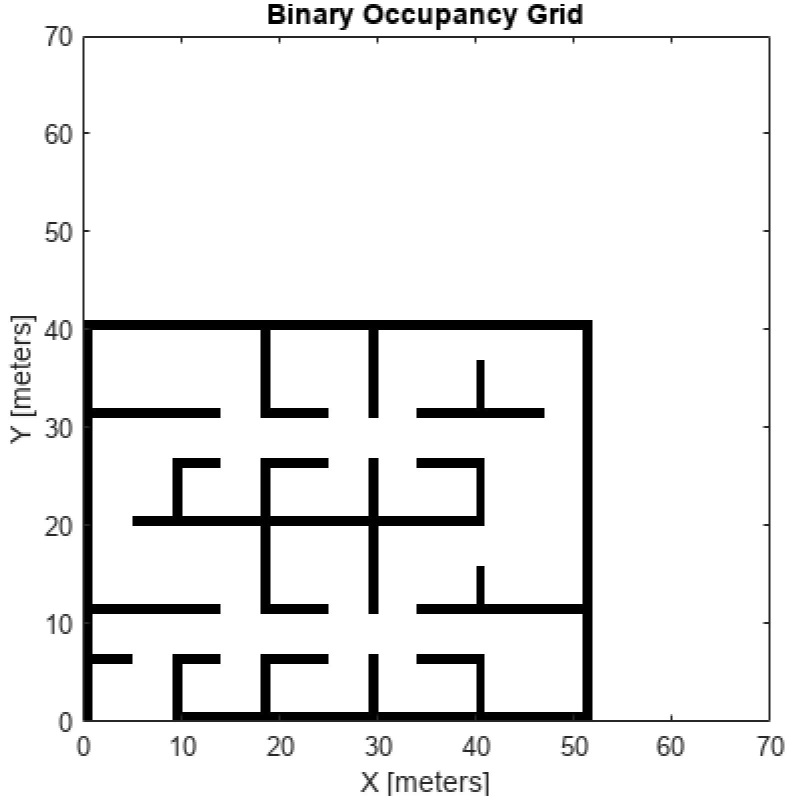
Create and display 2-D occupancy map using the map grid values stored in `complexMap`.

```
map2 = binaryOccupancyMap(complexMap);
show(map2)
title('Complex Map')
```



Now update `map1` with `map2` using the `syncWith` function.

```
syncWith(map1,map2);
show(map1)
```



Input Arguments

`map` — Map representation
`binaryOccupancyMap` object

Map representation, specified as a `binaryOccupancyMap` object.

`sourcemap` — Map representation
`binaryOccupancyMap` object

Map representation, specified as a `binaryOccupancyMap` object.

Version History

Introduced in R2019b

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

`binaryOccupancyMap` | `occupancyMap`

Topics

“Occupancy Grids”

world2grid

Convert world coordinates to grid indices

Syntax

```
ij = world2grid(map,xy)
```

Description

`ij = world2grid(map,xy)` converts an array of world coordinates, `xy`, to a `[rows cols]` array of grid indices, `ij`.

Examples

Convert World Coordinates in Binary Occupancy Map to Grid Indices

Create an empty binary occupancy map with a width and height of 10 meters.

```
map = binaryOccupancyMap(10,10);
```

Get grid indices from world coordinates.

```
[xWorld,yWorld] = meshgrid(0:0.5:2);
ij = world2grid(map,[xWorld(:) yWorld(:)]);
```

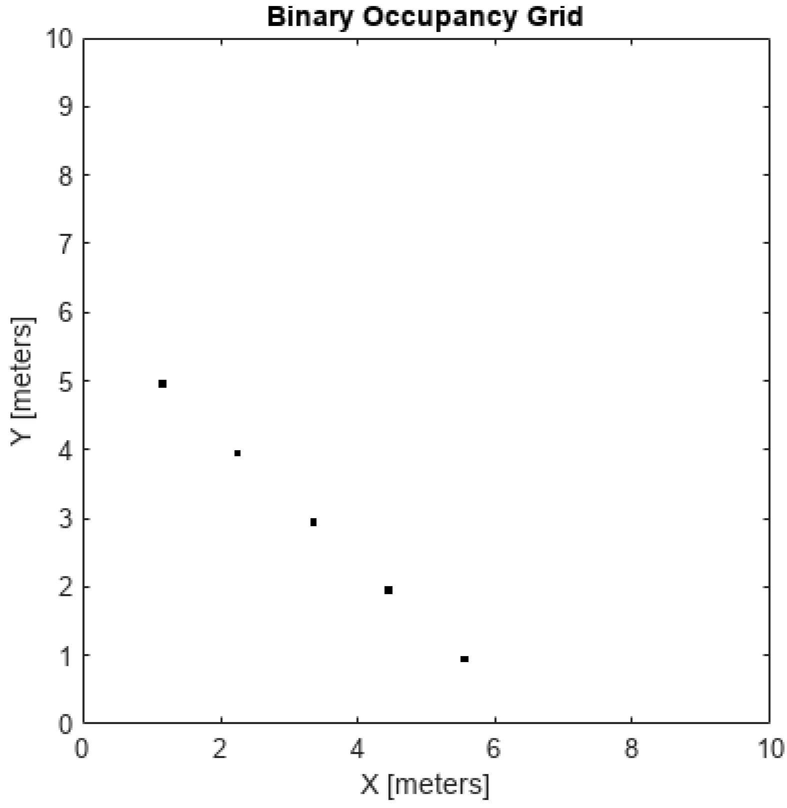
Create and Modify Binary Occupancy Grid

Create a 10m x 10m empty map.

```
map = binaryOccupancyMap(10,10,10);
```

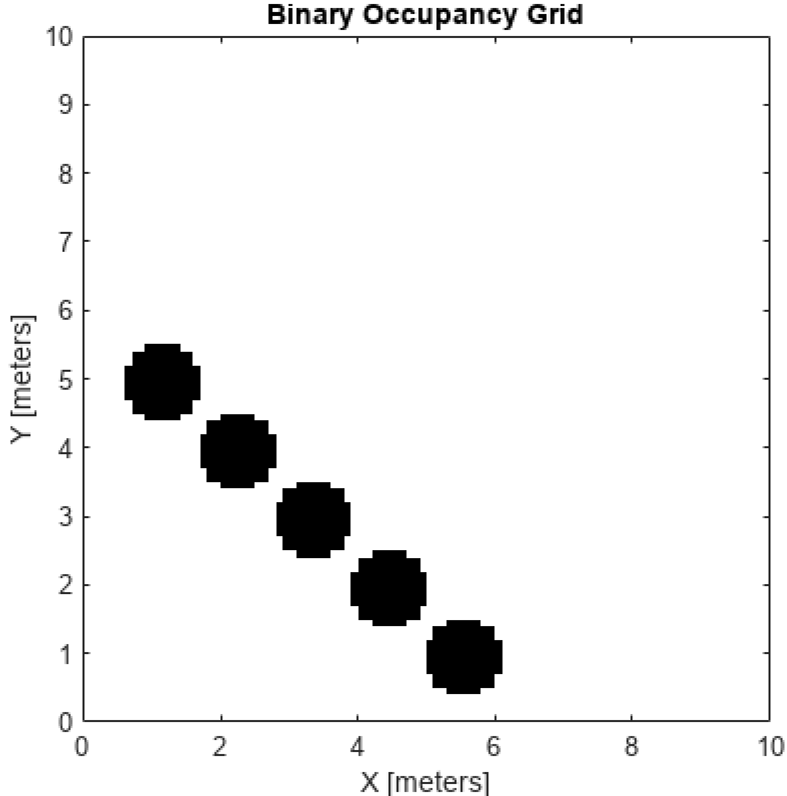
Set occupancy of world locations and show map.

```
x = [1.2; 2.3; 3.4; 4.5; 5.6];
y = [5.0; 4.0; 3.0; 2.0; 1.0];

setOccupancy(map, [x y], ones(5,1))
figure
show(map)
```



Inflate occupied locations by a given radius.

```
inflate(map, 0.5)
figure
show(map)
```

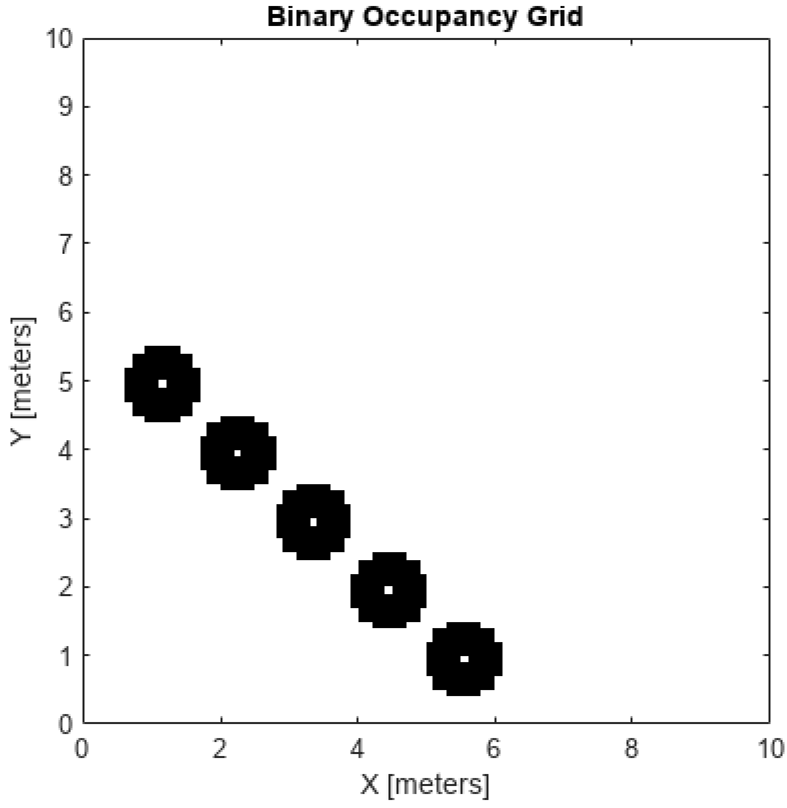


Get grid locations from world locations.

```
ij = world2grid(map, [x y]);
```

Set grid locations to free locations.

```
setOccupancy(map, ij, zeros(5,1), 'grid')
figure
show(map)
```



Input Arguments

map — Map representation
binaryOccupancyMap object

Map representation, specified as a `binaryOccupancyMap` object.

xy — World coordinates
n-by-2 vertical array

World coordinates, specified as an *n*-by-2 vertical array of `[x y]` pairs, where *n* is the number of world coordinates.

Output Arguments

ij — Grid indices
n-by-2 vertical array

Grid indices, specified as an *n*-by-2 vertical array of `[i j]` pairs in `[rows cols]` format, where *n* is the number of grid positions.

Version History

Introduced in R2015a

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

binaryOccupancyMap | grid2world

world2local

Convert world coordinates to local coordinates

Syntax

```
xyLocal = world2local(map,xy)
```

Description

`xyLocal = world2local(map,xy)` converts an array of world coordinates to local coordinates.

Examples

Convert World Coordinates in Binary Occupancy Map to Local Coordinates

Create an empty binary occupancy map with a width and height of 10 meters.

```
map = binaryOccupancyMap(10,10);
```

Get local coordinates from world coordinates.

```
[xWorld,yWorld] = meshgrid(0:0.5:2);
xyLocal = world2local(map,[xWorld(:) yWorld(:)]);
```

Input Arguments

`map` — Map representation
`binaryOccupancyMap` object

Map representation, specified as a `binaryOccupancyMap` object.

xy — World coordinates
n-by-2 vertical array

World coordinates, specified as an *n*-by-2 vertical array of `[x y]` pairs, where *n* is the number of world coordinates.

Output Arguments

`xyLocal` — Local coordinates
n-by-2 vertical array

Local coordinates, specified as an *n*-by-2 vertical array of `[x y]` pairs, where *n* is the number of local coordinates.

Version History

Introduced in R2019b

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`binaryOccupancyMap` | `grid2world` | `local2world`

addCapsule

Add collision capsule to rigid body

Syntax

`addCapsule(capapprox,bodyname,parameters,pose)`

Description

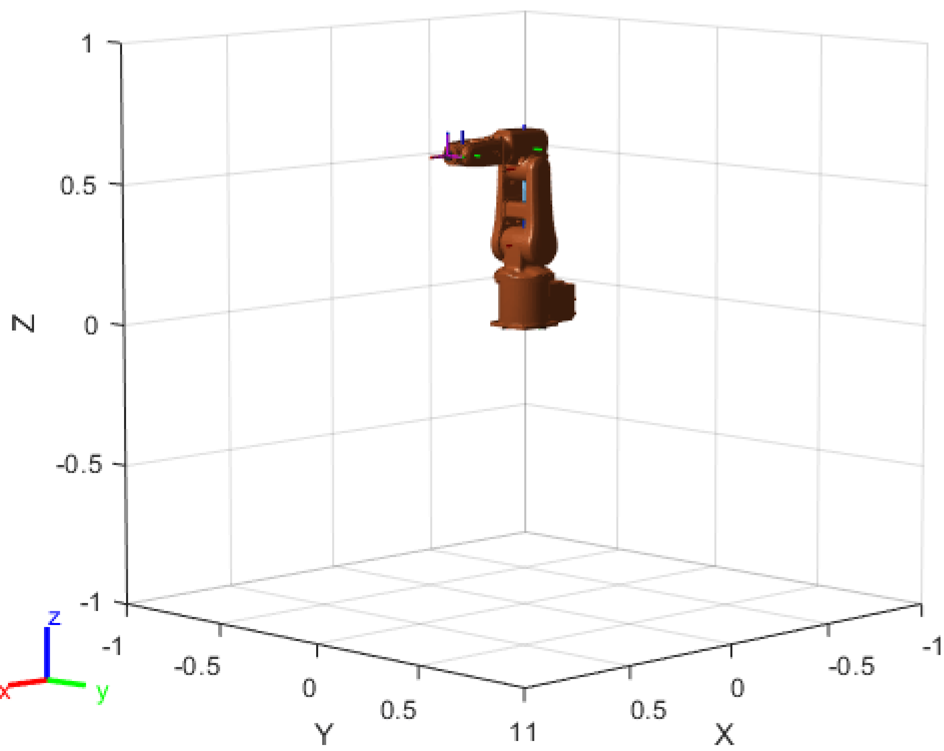
`addCapsule(capapprox,bodyname,parameters,pose)` adds a collision capsule at the next index of the rigid body `bodyname` with the specified pose `pose` and geometry parameters `parameters`.

Examples

Create and Modify Capsule Approximation

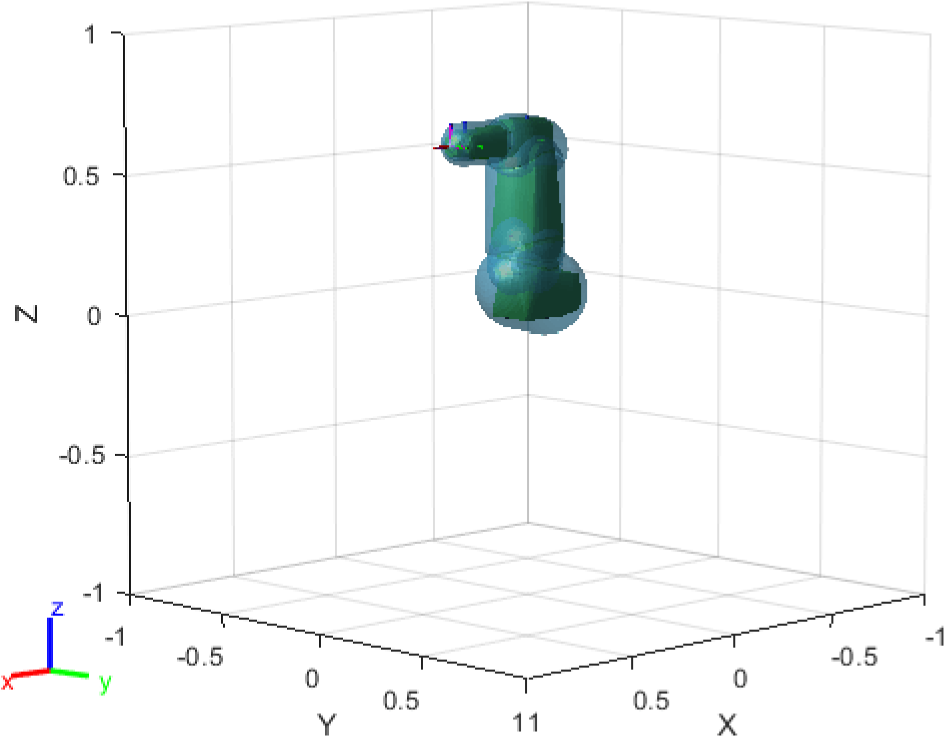
Load a robot into the workspace and visualize it.

```
robotIRB = loadrobot("abbIrb120");
show(robotIRB);
```



Create a capsule approximation of the robot, and visualize the capsule-approximated robot model.

```
capsIRB = capsuleApproximation(robotIRB);
figure
show(capsIRB,homeConfiguration(capsIRB.RigidBodyTree));
```



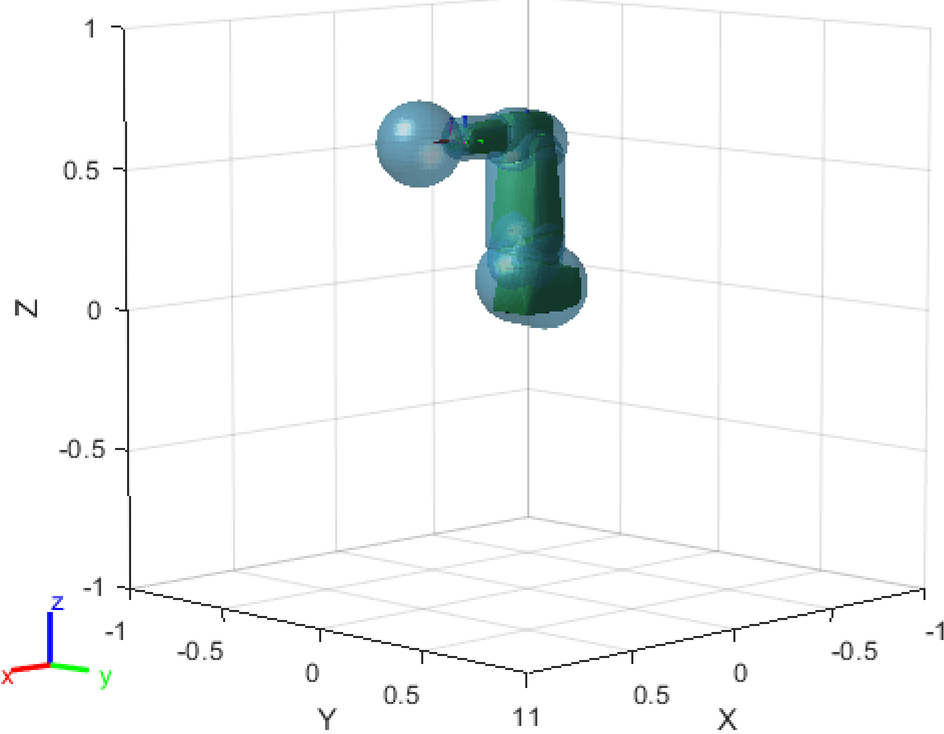
Use the `getCapsules` function to see if the end effector, `"tool0"`, has any collision capsules. Because `tool0` is just a frame, it has no collision mesh to approximate as a collision capsule.

```
capsulesTool = getCapsules(capsIRB,"tool0")
```

```
capsulesTool =

  1x0 empty cell array
```

Add a capsule to `tool0`, at a position `0.15` meters along the *x*-axis, with a radius of `0.15` and a length of `0`.

```
addCapsule(capsIRB,"tool0",[0.15 0],trvec2tform([0.15 0 0]))
show(capsIRB,homeConfiguration(capsIRB.RigidBodyTree));
```



Again check tool0 for a collision capsule, and verify the properties of the detected capsule.

```
capsulesTool = getCapsules(capsIRB,"tool0")
```

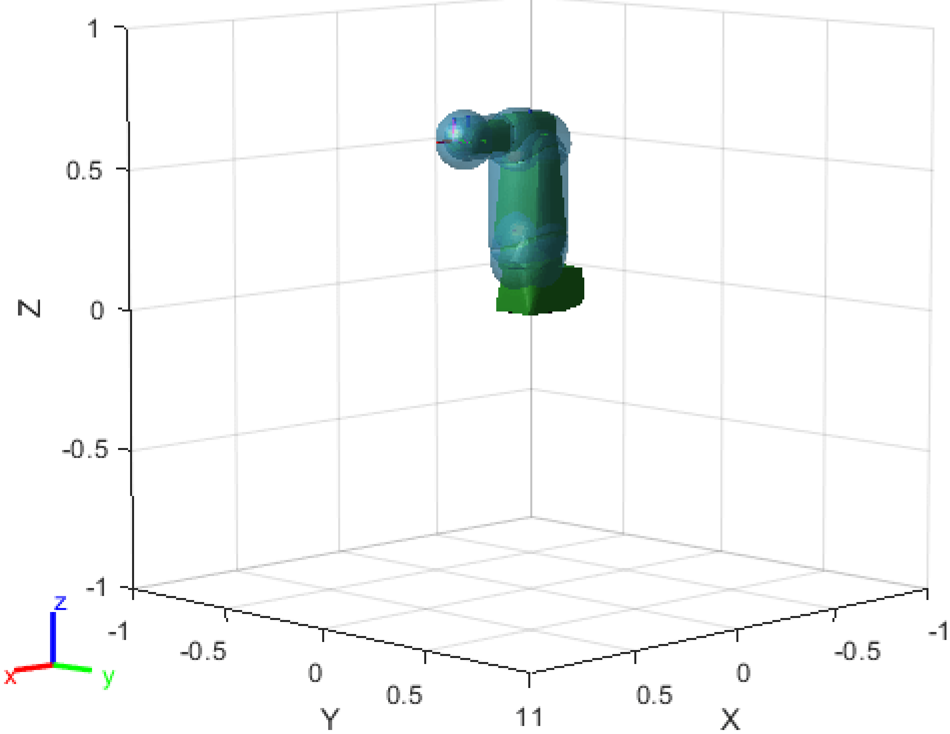
```
capsulesTool = 1x1 cell array
    {1x1 collisionCapsule}
```

```
capsulesTool{1}
```

```
ans = 
  collisionCapsule with properties:

    Radius: 0.1500
    Length: 0
      Pose: [4x4 double]
```

Remove the capsule from the base link. Then, reduce the collision capsule size of `tool0`, and move it `-0.05` meters from the previous position along the *x*-axis.

```
removeCapsule(capsIRB,"base_link",1)
updatePose(capsIRB,"tool0",trvec2tform([-0.05 0 0]),1)
updateGeometry(capsIRB,"tool0",[.1 0.01],1)
show(capsIRB,homeConfiguration(capsIRB.RigidBodyTree));
```



Input Arguments

capapprox — Capsule approximation of rigid body tree
`capsuleApproximation` object

Capsule approximation of a rigid body tree, specified as a `capsuleApproximation` object.

bodyname — Name of rigid body to add capsule to
string scalar | character vector

Name of the rigid body to add the capsule to, specified as a string scalar or character vector. The rigid body must exist in the `rigidBodyTree` object of the `RigidBodyTree` property of `capapprox`.

Example: `"EndEffectorTool"`

Data Types: `char` | `string`

parameters — Radius and length of added collision capsule
two-element row vector

Radius and length of the added collision capsule, specified as a two-element row vector of the form [*radius length*], in meters. The *radius* is the radius of the spherical ends of the capsule, and the *length* is the length of the central line segment of the capsule.

Example: `[1 2]`

pose — Pose for added collision capsule
4-by-4 matrix

Pose for the added collision capsule, specified as a 4-by-4 homogeneous transformation matrix defined with respect to the frame of the rigid body `bodyname`.

Example: `eye(4)`

Version History

Introduced in R2022b

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

Objects

`capsuleApproximation`

Functions

`removeCapsule` | `show` | `getCapsules` | `updatePose` | `updateGeometry`

getCapsules

Get collision capsules of rigid body

Syntax

```
[capsules,fitInfo] = getCapsules(capapprox,bodyname)
[capsules,fitInfo] = getCapsules( ___ ,maxcollisoncapsules)
```

Description

`[capsules,fitInfo] = getCapsules(capapprox,bodyname)` gets the collision capsules of the specified body `bodyname` of the rigid body tree in the capsule approximation. The function also returns the fit information of the collision capsules.

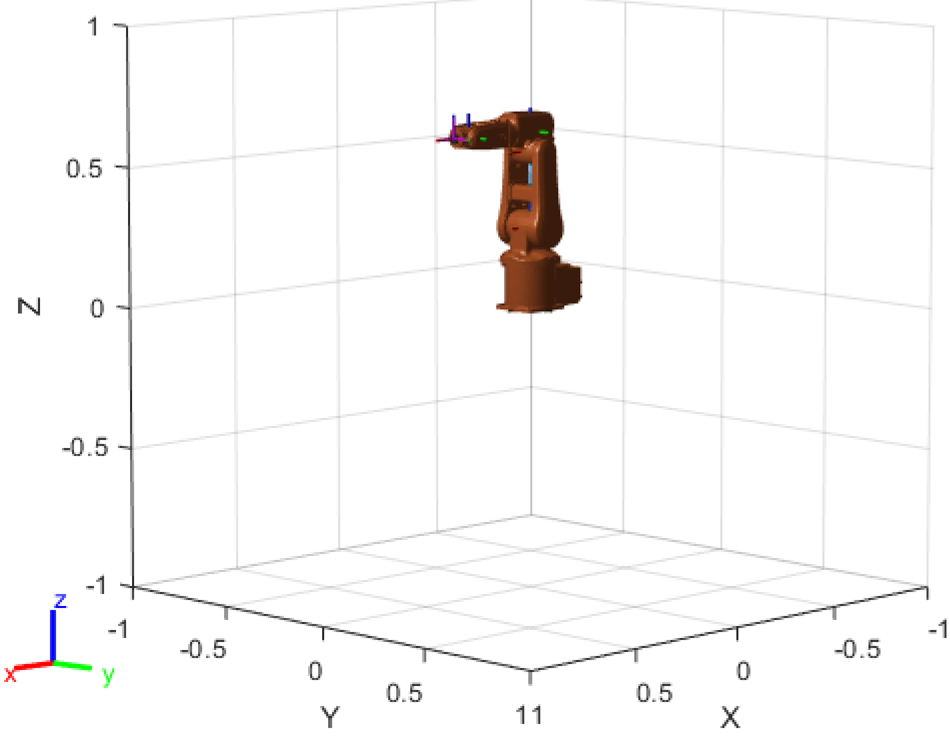
`[capsules,fitInfo] = getCapsules( ___ ,maxcollisoncapsules)` specifies the maximum number of capsules to return during code generation `maxcollisoncapsules`, in addition to the input arguments from the previous syntax. If you specify `maxcollisoncapsules` during MATLAB execution, the function ignores it.

Examples

Create and Modify Capsule Approximation

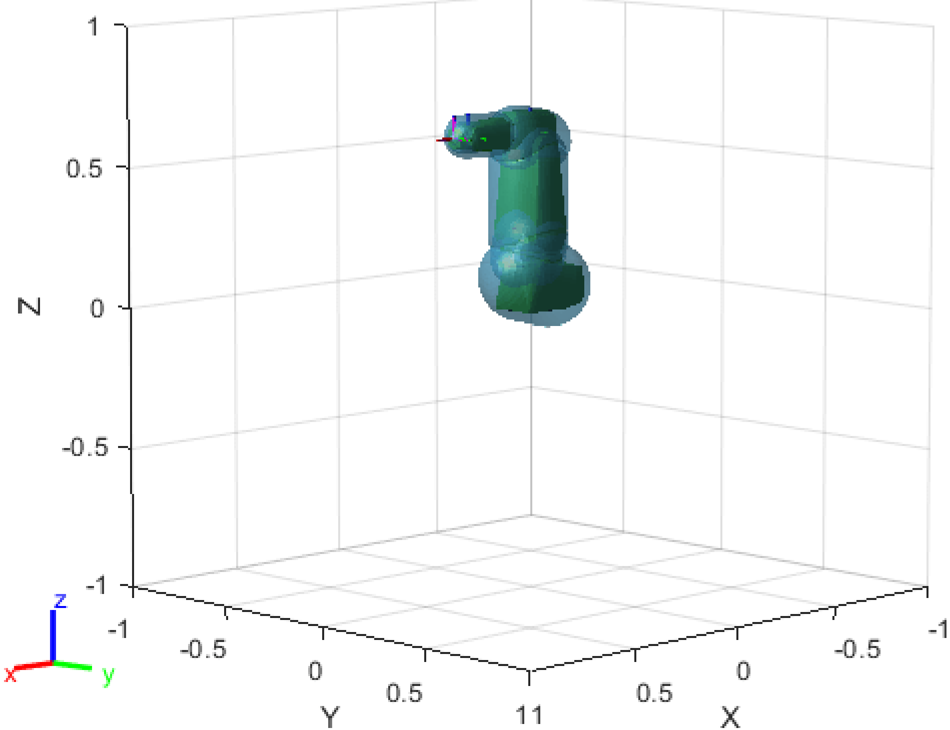
Load a robot into the workspace and visualize it.

```
robotIRB = loadrobot("abbIrb120");
show(robotIRB);
```



Create a capsule approximation of the robot, and visualize the capsule-approximated robot model.

```
capsIRB = capsuleApproximation(robotIRB);
figure
show(capsIRB,homeConfiguration(capsIRB.RigidBodyTree));
```



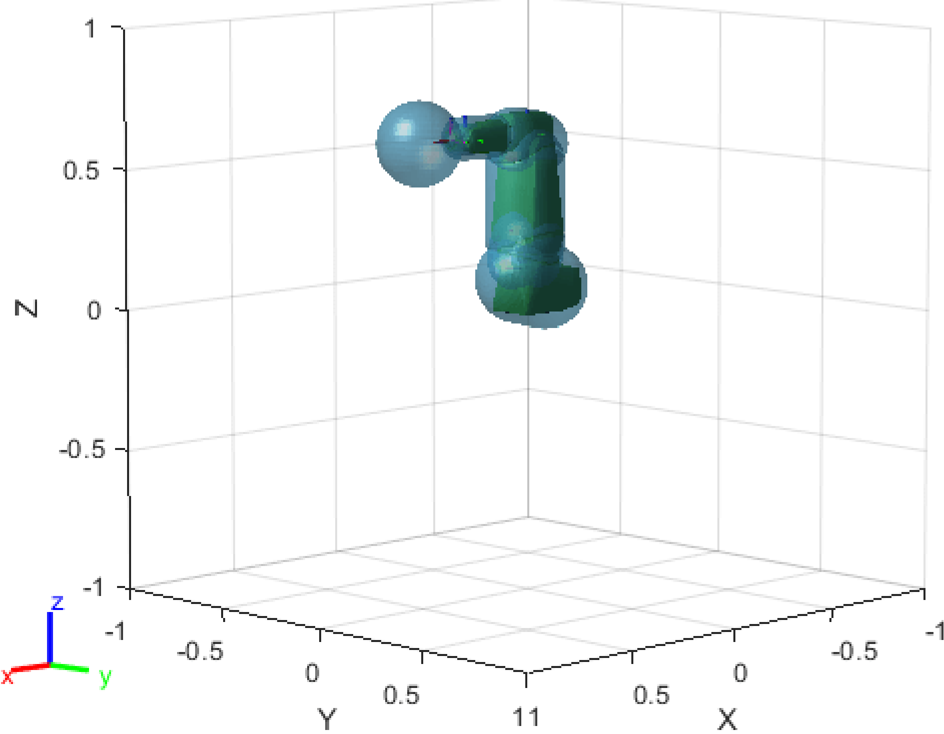
Use the `getCapsules` function to see if the end effector, `"tool0"`, has any collision capsules. Because `tool0` is just a frame, it has no collision mesh to approximate as a collision capsule.

```
capsulesTool = getCapsules(capsIRB,"tool0")
```

```
capsulesTool =

  1x0 empty cell array
```

Add a capsule to `tool0`, at a position `0.15` meters along the *x*-axis, with a radius of `0.15` and a length of `0`.

```
addCapsule(capsIRB,"tool0",[0.15 0],trvec2tform([0.15 0 0]))
show(capsIRB,homeConfiguration(capsIRB.RigidBodyTree));
```



Again check tool0 for a collision capsule, and verify the properties of the detected capsule.

```
capsulesTool = getCapsules(capsIRB,"tool0")
```

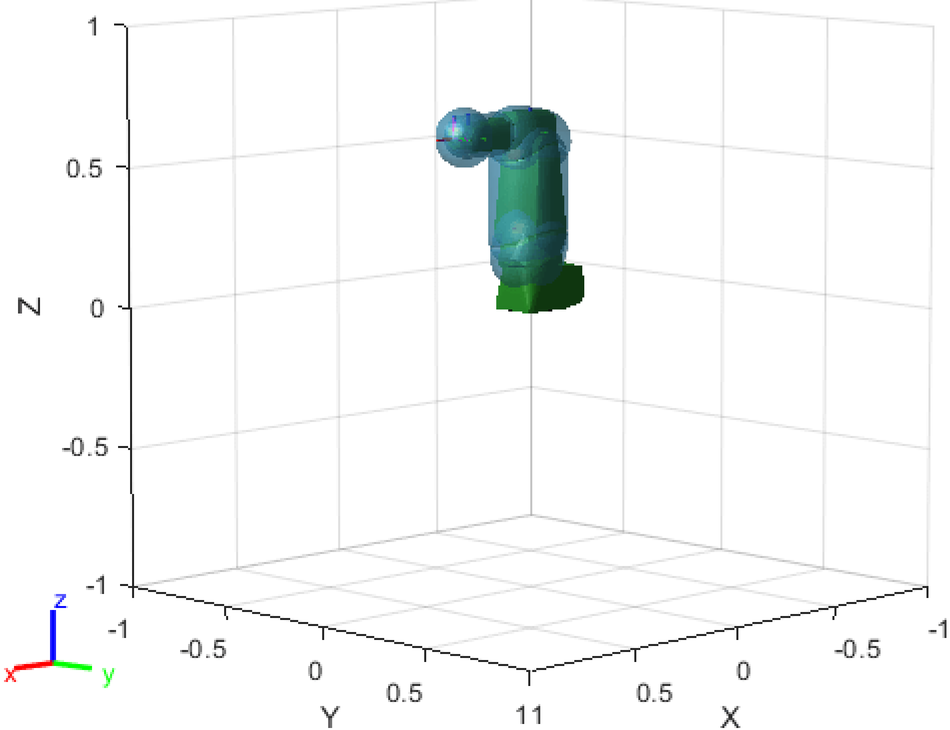
```
capsulesTool = 1x1 cell array
    {1x1 collisionCapsule}
```

```
capsulesTool{1}
```

```
ans =
  collisionCapsule with properties:

    Radius: 0.1500
    Length: 0
      Pose: [4x4 double]
```

Remove the capsule from the base link. Then, reduce the collision capsule size of `tool0`, and move it `-0.05` meters from the previous position along the *x*-axis.

```
removeCapsule(capsIRB,"base_link",1)
updatePose(capsIRB,"tool0",trvec2tform([-0.05 0 0]),1)
updateGeometry(capsIRB,"tool0",[.1 0.01],1)
show(capsIRB,homeConfiguration(capsIRB.RigidBodyTree));
```



Input Arguments

capapprox — Capsule approximation of rigid body tree
`capsuleApproximation` object

Capsule approximation of a rigid body tree, specified as a `capsuleApproximation` object.

bodyname — Name of rigid body to get capsules from
string scalar | character vector

Name of the rigid body to get capsules from, specified as a string scalar or character vector. The rigid body must exist in the `rigidBodyTree` object of the `RigidBodyTree` property of `capsapprox`.

Example: `"EndEffectorTool"`

Data Types: `char` | `string`

`maxcollisoncapsules` — Maximum number of collision capsules to return during code generation
`10` (default) | positive integer

Maximum number of collision capsules to return from the specified rigid body during code generation, specified as a positive integer.

If you specify `maxcollisoncapsules` during MATLAB execution, the function ignores it.

Output Arguments

`capsules` — Collision capsules of rigid body
cell array

Collision capsules of the rigid body, returned as a cell array of `collisionCapsule` objects.

`fitInfo` — Fit information of collision capsules
array of structures

Fit information of the collision capsules, returned as an *M*-element array of structures, where *M* is the total number of capsules of the rigid body. Each element of `fitInfo` contains the fit information for the collision capsule at the corresponding index. Each structure contains the `Residual` field, returned as an *N*-element vector, where *N* is the total number of points of the collision geometry. Each element of the vector specifies the residual of a point of the collision geometry as:

$$\left|(o_{cg} - l_{cc})\right| + r_{cc}$$

where:

- o_{cg} is the origin of the fitted collision object.
- l_{cc} is the point of the central line of the collision capsule closest to o_{cg}.
- r_{cc} is the radius of the collision capsule.

Version History

Introduced in R2022b

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

If the `maxcollisoncapsules` input argument is unspecified during code generation, the maximum number of capsules of the rigid body returned is `10`, resulting in only the capsules at indices in the range [1, 10] being returned. If `maxcollisoncapsules` is specified in MATLAB execution, it is ignored.

See Also

Objects
`capsuleApproximation`

Functions
`addCapsule` | `removeCapsule` | `show` | `updatePose` | `updateGeometry`

removeCapsule

Remove collision capsule from rigid body

Syntax

```
removeCapsule(capapprox,bodyname)
removeCapsule( ___ ,idx)
```

Description

`removeCapsule(capapprox,bodyname)` removes the collision capsule at the last index of the rigid body `bodyname`.

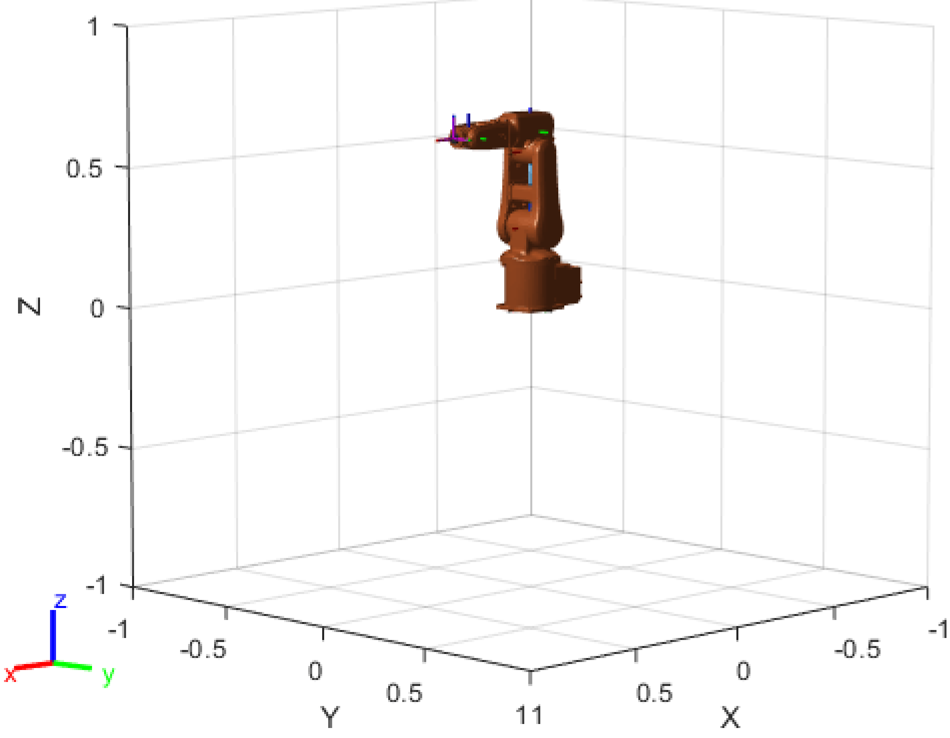
`removeCapsule( ___ ,idx)` removes the collision capsule at the specified index `idx` of the rigid body, in addition to the input arguments from the previous syntax.

Examples

Create and Modify Capsule Approximation

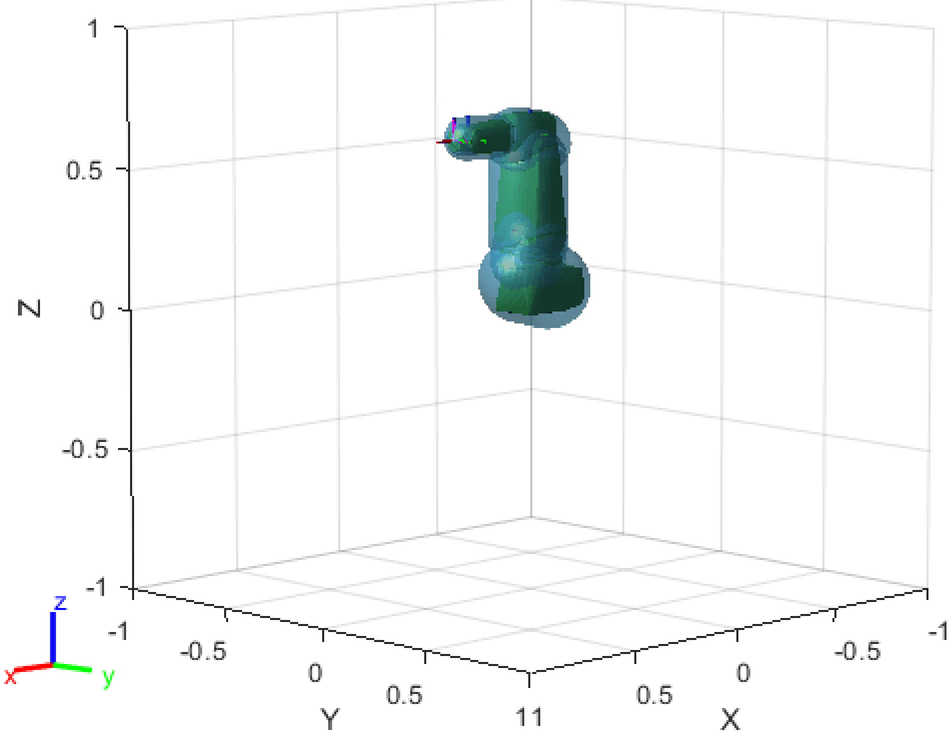
Load a robot into the workspace and visualize it.

```
robotIRB = loadrobot("abbIrb120");
show(robotIRB);
```



Create a capsule approximation of the robot, and visualize the capsule-approximated robot model.

```
capsIRB = capsuleApproximation(robotIRB);
figure
show(capsIRB,homeConfiguration(capsIRB.RigidBodyTree));
```



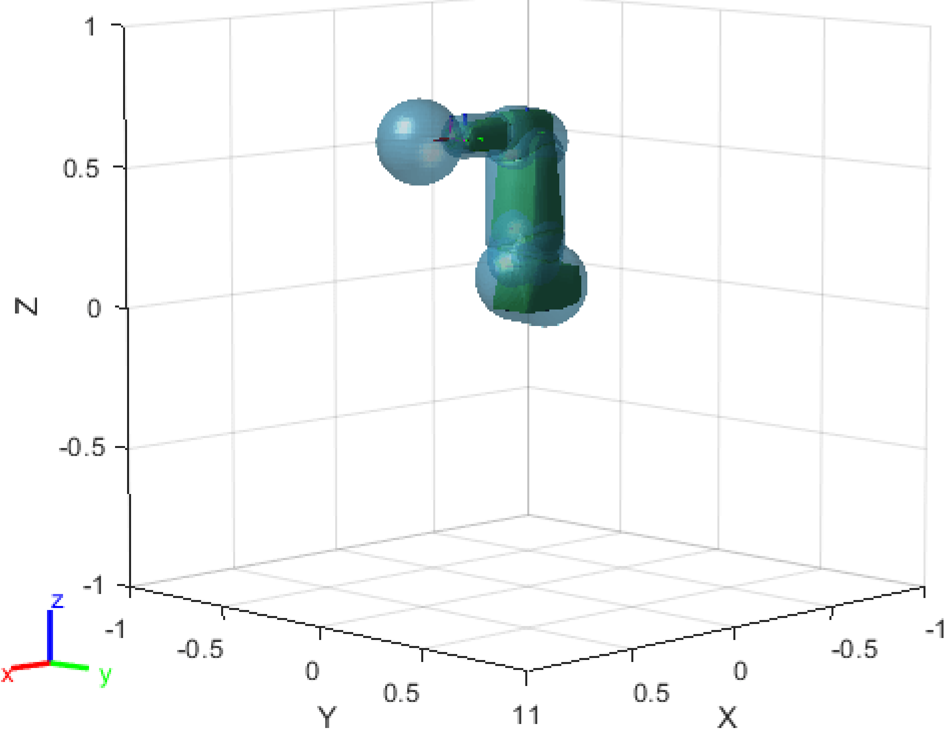
Use the `getCapsules` function to see if the end effector, `"tool0"`, has any collision capsules. Because `tool0` is just a frame, it has no collision mesh to approximate as a collision capsule.

```
capsulesTool = getCapsules(capsIRB,"tool0")
```

```
capsulesTool =

  1x0 empty cell array
```

Add a capsule to `tool0`, at a position `0.15` meters along the *x*-axis, with a radius of `0.15` and a length of `0`.

```
addCapsule(capsIRB,"tool0",[0.15 0],trvec2tform([0.15 0 0]))
show(capsIRB,homeConfiguration(capsIRB.RigidBodyTree));
```



Again check tool0 for a collision capsule, and verify the properties of the detected capsule.

```
capsulesTool = getCapsules(capsIRB,"tool0")
```

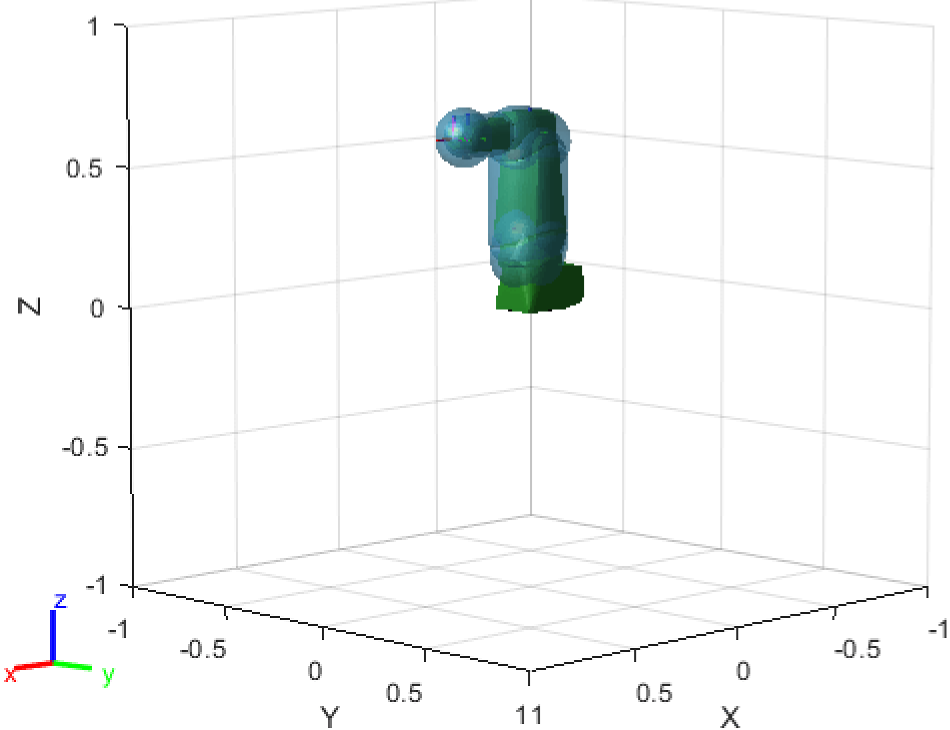
```
capsulesTool = 1x1 cell array
    {1x1 collisionCapsule}
```

```
capsulesTool{1}
```

```
ans = 
  collisionCapsule with properties:

    Radius: 0.1500
    Length: 0
      Pose: [4x4 double]
```

Remove the capsule from the base link. Then, reduce the collision capsule size of `tool0`, and move it `-0.05` meters from the previous position along the *x*-axis.

```
removeCapsule(capsIRB,"base_link",1)
updatePose(capsIRB,"tool0",trvec2tform([-0.05 0 0]),1)
updateGeometry(capsIRB,"tool0",[.1 0.01],1)
show(capsIRB,homeConfiguration(capsIRB.RigidBodyTree));
```



Input Arguments

capapprox — Capsule approximation of rigid body tree
capsuleApproximation object

Capsule approximation of a rigid body tree, specified as a capsuleApproximation object.

bodyname — Name of rigid body to remove capsule from
string scalar | character vector

Name of the rigid body to remove the capsule from, specified as a string scalar or character vector. The rigid body must exist in the rigidBodyTree object of the RigidBodyTree property of capsapprox.

Example: "EndEffectorTool"

Data Types: char | string

idx — Index of collision capsule to remove
positive integer

Index of the collision capsule to remove, specified as a positive integer.

Example: 2

Version History

Introduced in R2022b

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

Objects

`capsuleApproximation`

Functions

`addCapsule` | `show` | `getCapsules` | `updatePose` | `updateGeometry`

show

Visualize capsule approximation of rigid body tree

Syntax

```
show(capapprox)
show(capapprox,config)
show(capapprox,Parent=ax)
ax = show(capapprox, ___ )
```

Description

`show(capapprox)` shows the collision capsule approximation `capsapprox`, superimposed on the original collision geometries of the corresponding rigid body tree in the home configuration of the rigid body tree.

`show(capapprox,config)` shows the collision capsule approximation superimposed on the original collision geometries of the corresponding rigid body tree with the specified joint configuration `config`.

`show(capapprox,Parent=ax)` specifies the parent axes handle `ax` to plot capsule-approximated rigid body tree.

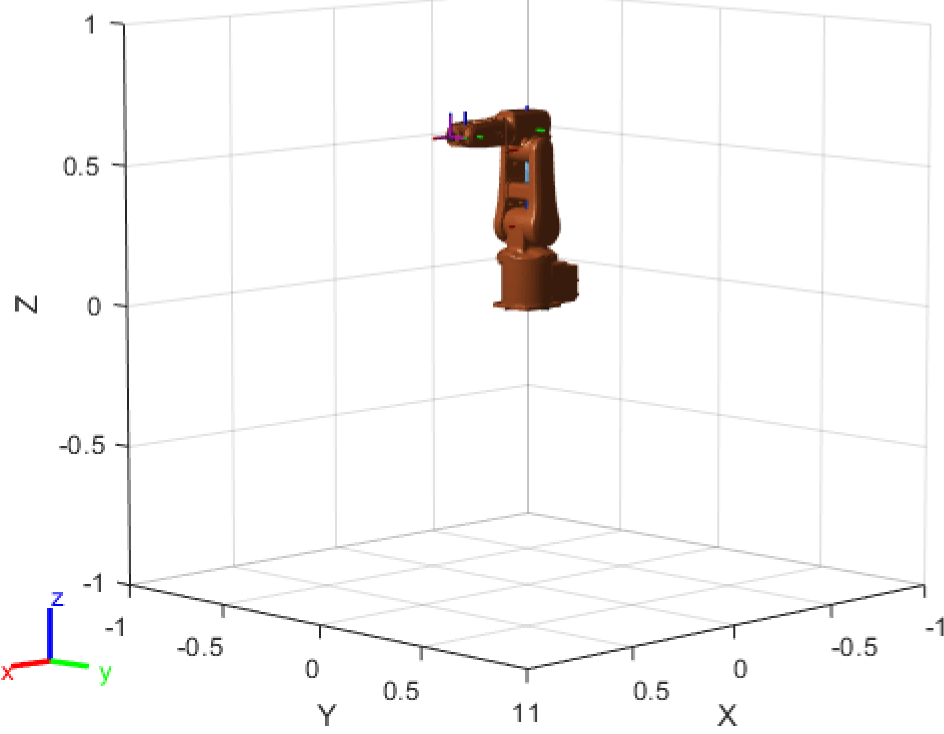
`ax = show(capapprox, ___ )` returns the axes handle `ax` containing the capsule-approximated rigid body tree plot.

Examples

Create and Modify Capsule Approximation

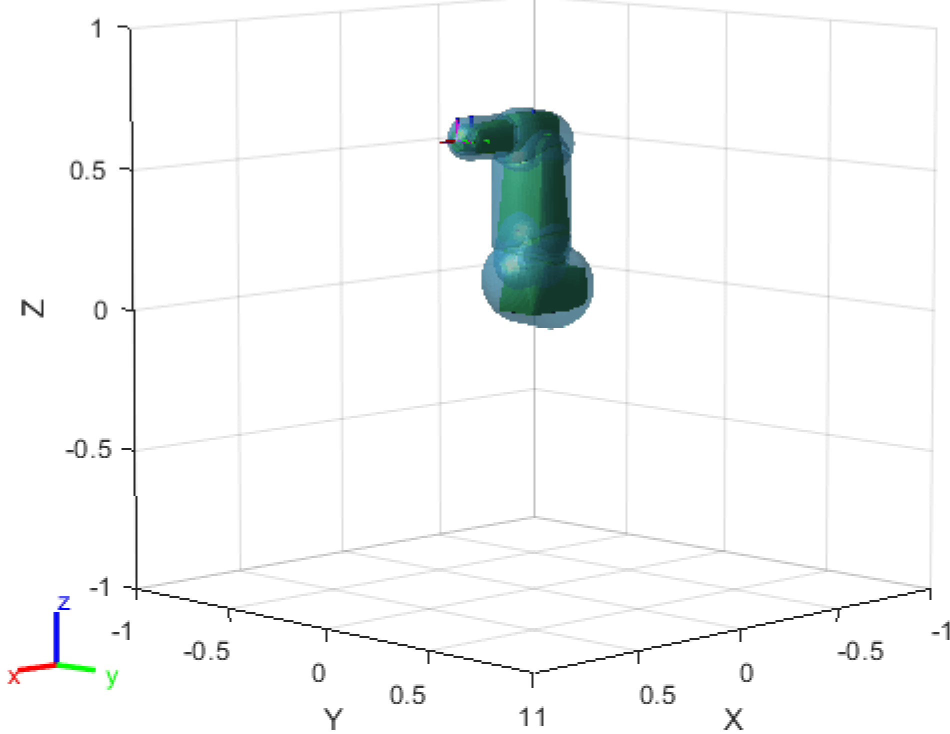
Load a robot into the workspace and visualize it.

```
robotIRB = loadrobot("abbIrb120");
show(robotIRB);
```



Create a capsule approximation of the robot, and visualize the capsule-approximated robot model.

```
capsIRB = capsuleApproximation(robotIRB);
figure
show(capsIRB,homeConfiguration(capsIRB.RigidBodyTree));
```



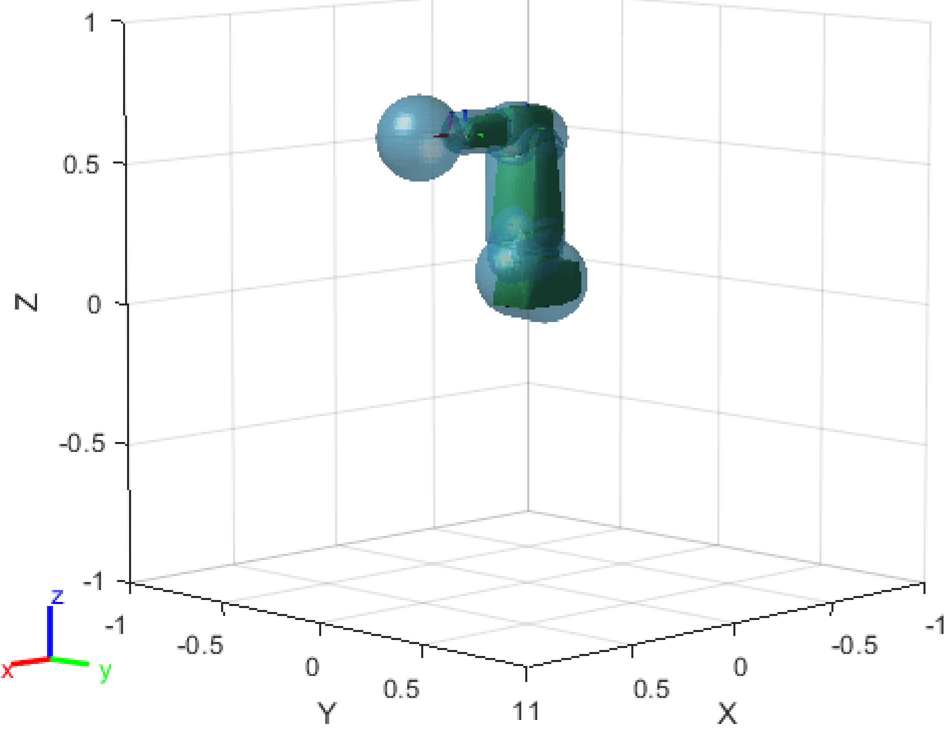
Use the `getCapsules` function to see if the end effector, `"tool0"`, has any collision capsules. Because `tool0` is just a frame, it has no collision mesh to approximate as a collision capsule.

```
capsulesTool = getCapsules(capsIRB,"tool0")
```

```
capsulesTool =

  1x0 empty cell array
```

Add a capsule to `tool0`, at a position `0.15` meters along the *x*-axis, with a radius of `0.15` and a length of `0`.

```
addCapsule(capsIRB,"tool0",[0.15 0],trvec2tform([0.15 0 0]))
show(capsIRB,homeConfiguration(capsIRB.RigidBodyTree));
```



Again check tool0 for a collision capsule, and verify the properties of the detected capsule.

```
capsulesTool = getCapsules(capsIRB,"tool0")
```

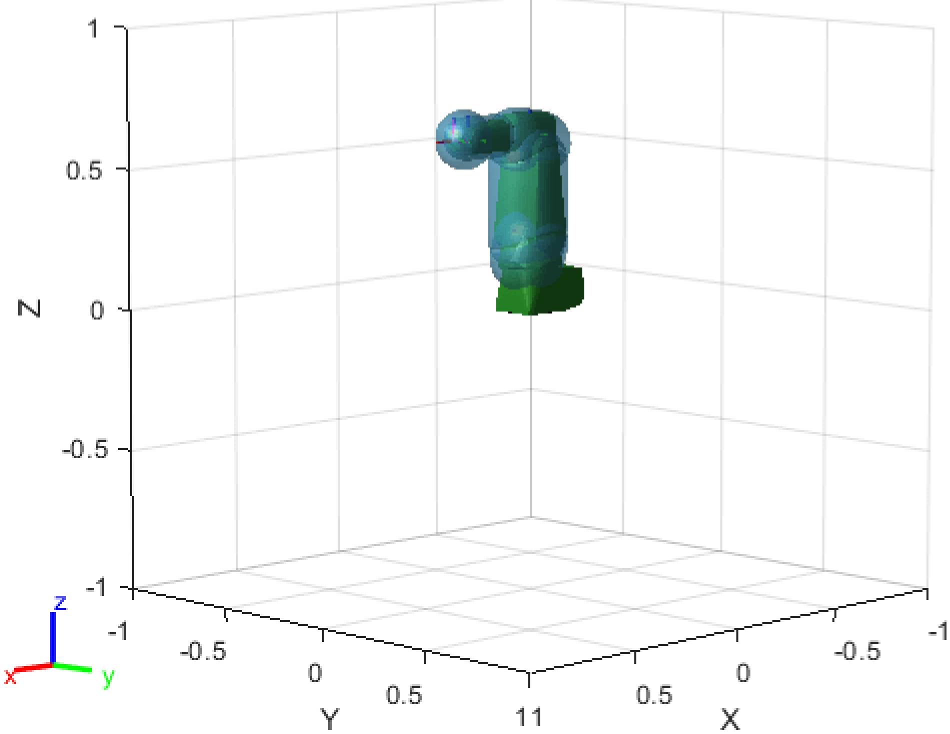
```
capsulesTool = 1x1 cell array
    {1x1 collisionCapsule}
```

```
capsulesTool{1}
```

```
ans =
  collisionCapsule with properties:

    Radius: 0.1500
    Length: 0
      Pose: [4x4 double]
```

Remove the capsule from the base link. Then, reduce the collision capsule size of `tool0`, and move it `-0.05` meters from the previous position along the *x*-axis.

```
removeCapsule(capsIRB,"base_link",1)
updatePose(capsIRB,"tool0",trvec2tform([-0.05 0 0]),1)
updateGeometry(capsIRB,"tool0",[.1 0.01],1)
show(capsIRB,homeConfiguration(capsIRB.RigidBodyTree));
```



Input Arguments

capapprox — Capsule approximation of rigid body tree
`capsuleApproximation` object

Capsule approximation of a rigid body tree, specified as a `capsuleApproximation` object.

config — Rigid body tree robot model configuration
n-element row vector

Rigid body tree robot model configuration, specified as an *n*-element row vector, in radians. *n* is the number of movable joints in the rigid body tree model.

Example: `[pi 0 pi/2 0 0]` is a configuration for a rigid body tree with five movable joints.

ax — Parent axes graphic handle
`Axes` object

Parent axes graphic handle, specified as an `Axes` object.

Output Arguments

ax — Axes graphic handle
`Axes` object

Axes graphic handle, returned as an `Axes` object.

Version History

Introduced in R2022b

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

Objects

`capsuleApproximation`

updateGeometry

Update geometry of collision capsule of rigid body

Syntax

```
updateGeometry(capapprox,bodyname,parameters)
updateGeometry( ___ ,idx)
```

Description

`updateGeometry(capapprox,bodyname,parameters)` updates the geometry of the first collision capsule of the rigid body `bodyname` with the new geometry parameters `parameters`.

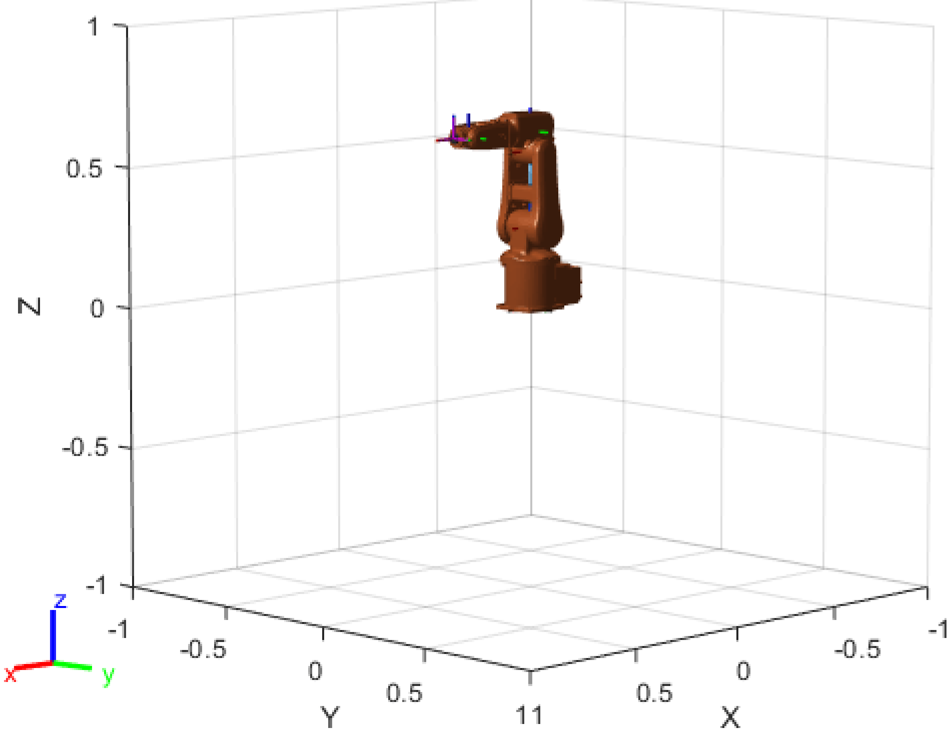
`updateGeometry( ___ ,idx)` updates the geometry of the collision capsule at index `idx` of the rigid body, in addition to the input arguments from the previous syntax.

Examples

Create and Modify Capsule Approximation

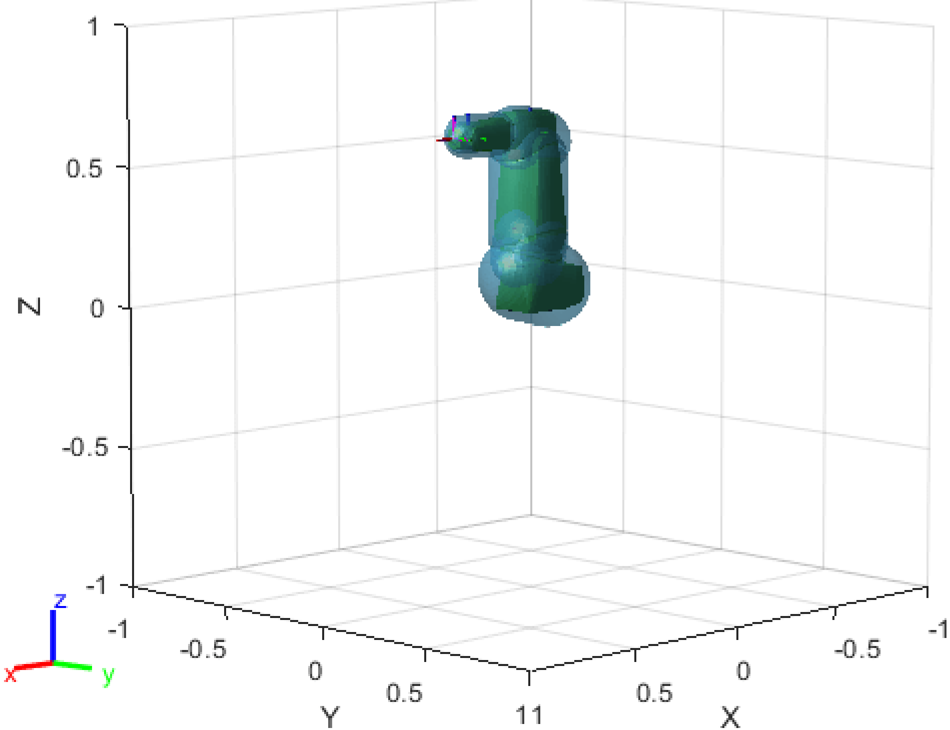
Load a robot into the workspace and visualize it.

```
robotIRB = loadrobot("abbIrb120");
show(robotIRB);
```



Create a capsule approximation of the robot, and visualize the capsule-approximated robot model.

```
capsIRB = capsuleApproximation(robotIRB);
figure
show(capsIRB,homeConfiguration(capsIRB.RigidBodyTree));
```



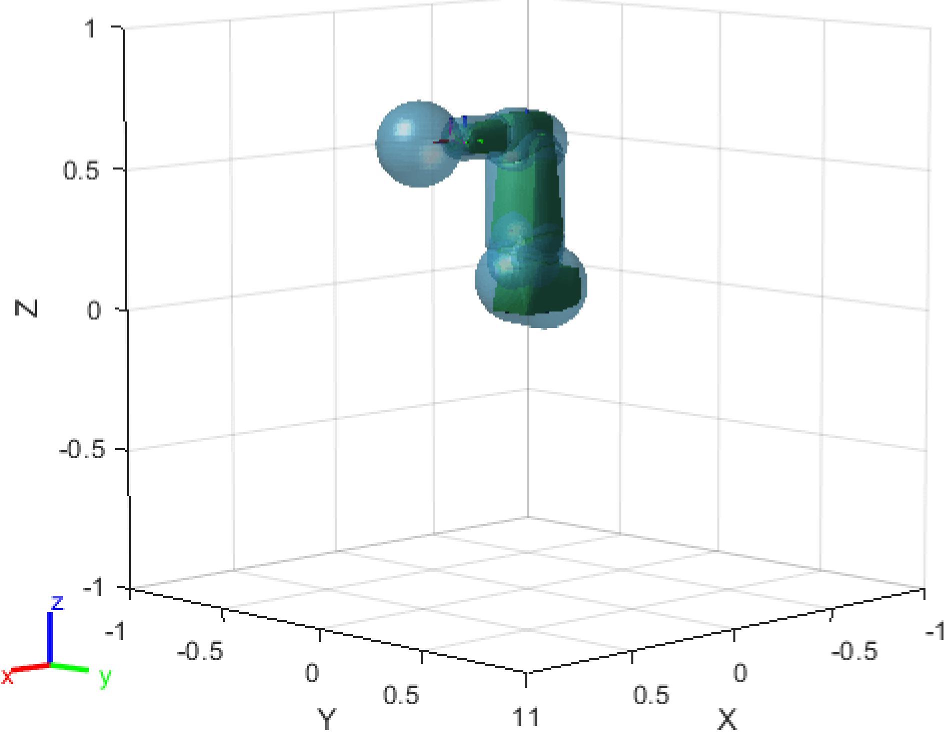
Use the `getCapsules` function to see if the end effector, `"tool0"`, has any collision capsules. Because `tool0` is just a frame, it has no collision mesh to approximate as a collision capsule.

```
capsulesTool = getCapsules(capsIRB,"tool0")
```

```
capsulesTool =

  1x0 empty cell array
```

Add a capsule to `tool0`, at a position `0.15` meters along the *x*-axis, with a radius of `0.15` and a length of `0`.

```
addCapsule(capsIRB,"tool0",[0.15 0],trvec2tform([0.15 0 0]))
show(capsIRB,homeConfiguration(capsIRB.RigidBodyTree));
```



Again check tool0 for a collision capsule, and verify the properties of the detected capsule.

```
capsulesTool = getCapsules(capsIRB,"tool0")
```

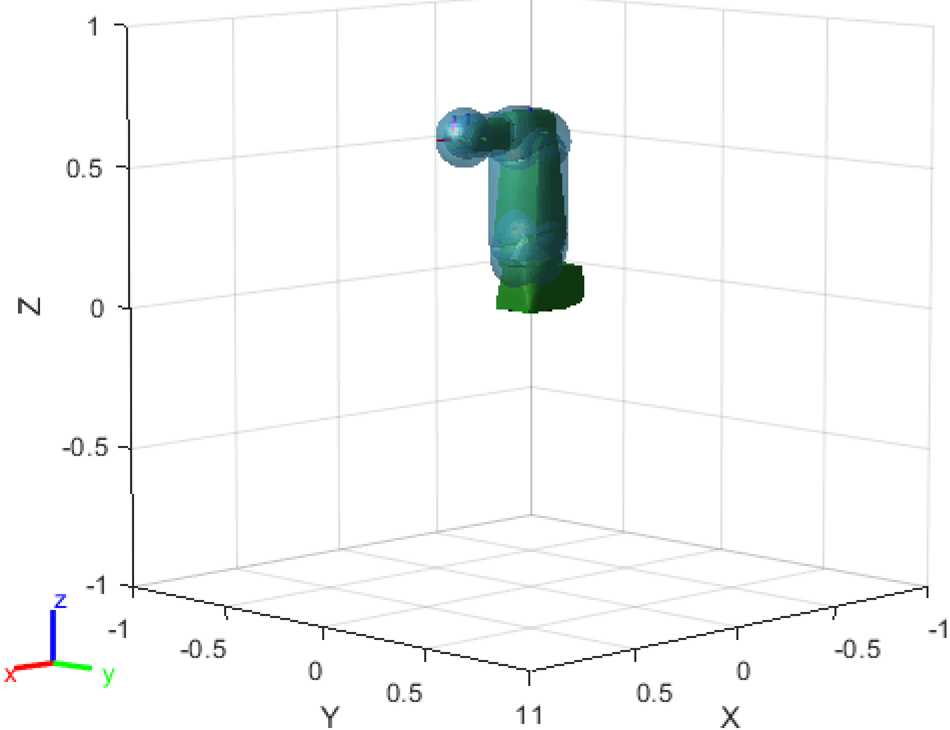
```
capsulesTool = 1x1 cell array
    {1x1 collisionCapsule}
```

```
capsulesTool{1}
```

```
ans = 
  collisionCapsule with properties:

    Radius: 0.1500
    Length: 0
      Pose: [4x4 double]
```

Remove the capsule from the base link. Then, reduce the collision capsule size of `tool0`, and move it `-0.05` meters from the previous position along the *x*-axis.

```
removeCapsule(capsIRB,"base_link",1)
updatePose(capsIRB,"tool0",trvec2tform([-0.05 0 0]),1)
updateGeometry(capsIRB,"tool0",[.1 0.01],1)
show(capsIRB,homeConfiguration(capsIRB.RigidBodyTree));
```



Input Arguments

capapprox — Capsule approximation of rigid body tree
capsuleApproximation object

Capsule approximation of a rigid body tree, specified as a `capsuleApproximation` object.

bodyname — Name of rigid body
string scalar | character vector

Name of the rigid body, specified as a string scalar or character vector. The rigid body must exist in the `rigidBodyTree` object of the `RigidBodyTree` property of `capsapprox`.

Example: "EndEffectorTool"

Data Types: char | string

idx — Index of collision capsule in rigid body
nonnegative integer

Index of the collision capsule in the rigid body, specified as a nonnegative integer.

Example: 5

parameters — Updated radius and length of added collision capsule
two-element row vector

Updated radius and length of the collision capsule, specified as a two-element row vector of the form [*radius length*], in meters. The *radius* is the radius of the spherical ends of the capsule, and the *length* is the length of the central line segment of the capsule.

Example: [1 2]

Version History

Introduced in R2022b

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

Objects
capsuleApproximation

Functions
addCapsule | removeCapsule | show | getCapsules | updatePose

updatePose

Update pose of collision capsule of rigid body

Syntax

```
updatePose(capapprox,bodyname,pose)
updatePose( ___ ,idx)
```

Description

`updatePose(capapprox,bodyname,pose)` updates the pose of the first collision capsule of the rigid body `bodyname` with the new pose `pose`.

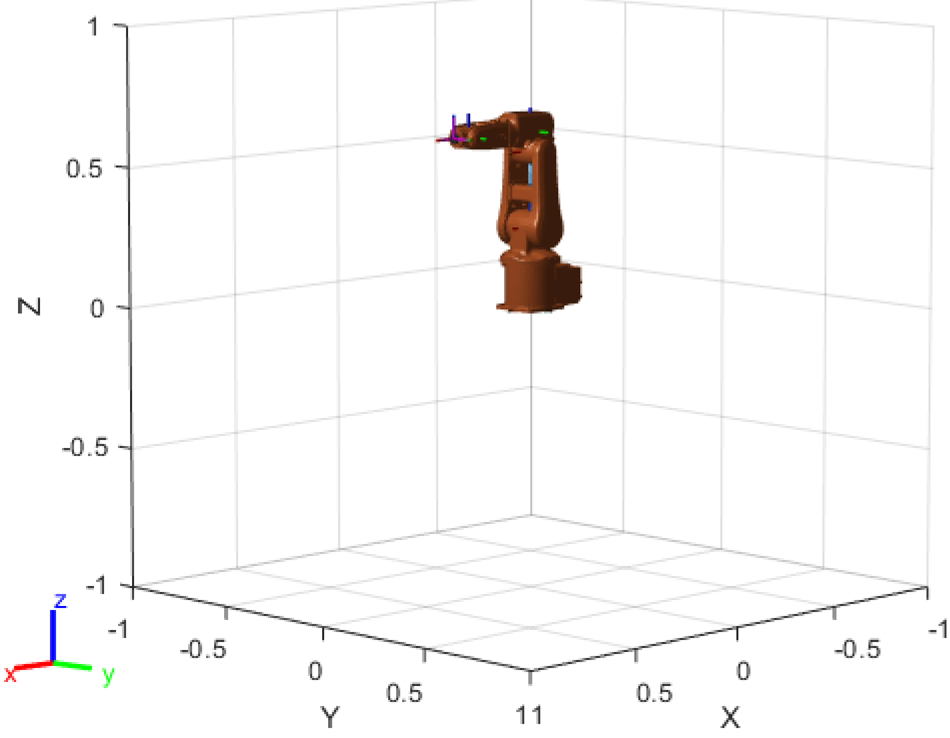
`updatePose( ___ ,idx)` updates the pose of the collision capsule at index `idx`, in addition to the arguments from the previous syntax.

Examples

Create and Modify Capsule Approximation

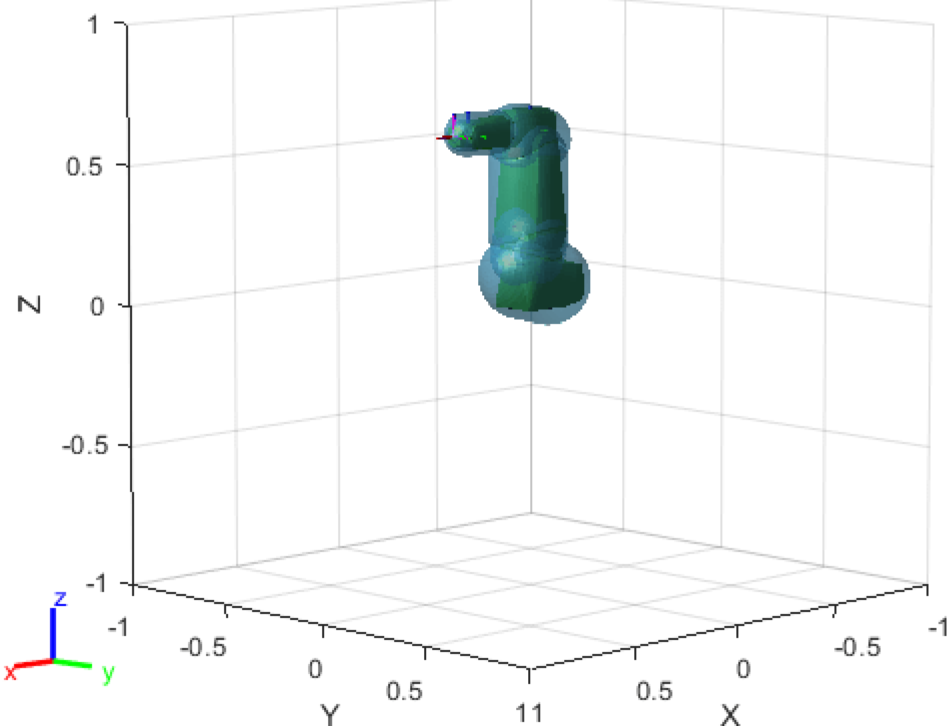
Load a robot into the workspace and visualize it.

```
robotIRB = loadrobot("abbIrb120");
show(robotIRB);
```



Create a capsule approximation of the robot, and visualize the capsule-approximated robot model.

```
capsIRB = capsuleApproximation(robotIRB);
figure
show(capsIRB,homeConfiguration(capsIRB.RigidBodyTree));
```



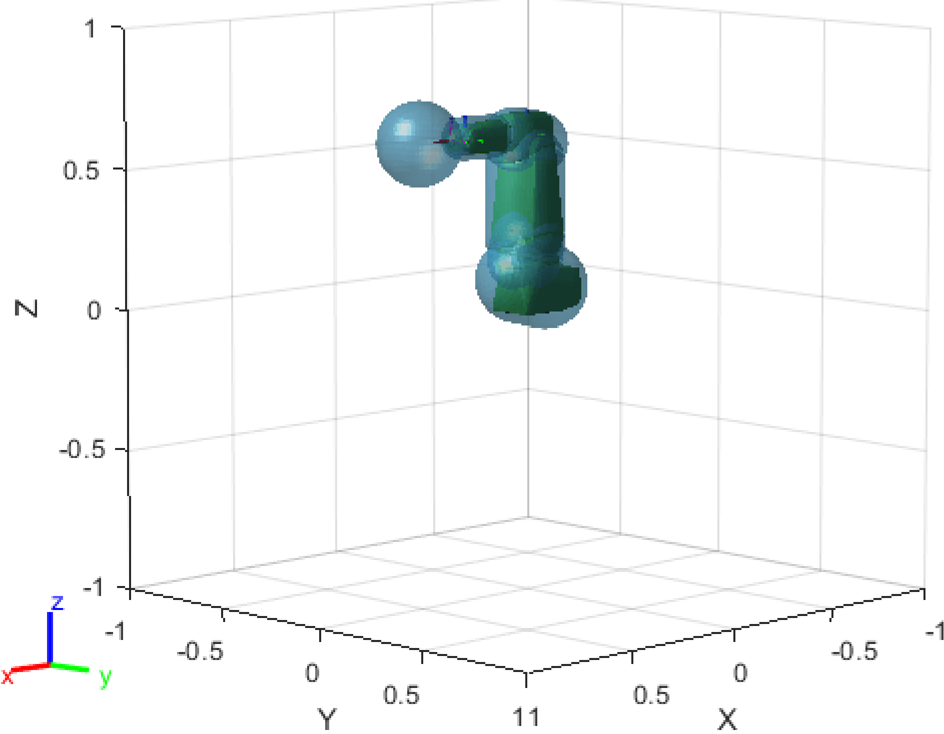
Use the `getCapsules` function to see if the end effector, `"tool0"`, has any collision capsules. Because `tool0` is just a frame, it has no collision mesh to approximate as a collision capsule.

```
capsulesTool = getCapsules(capsIRB,"tool0")
```

```
capsulesTool =

  1x0 empty cell array
```

Add a capsule to `tool0`, at a position `0.15` meters along the *x*-axis, with a radius of `0.15` and a length of `0`.

```
addCapsule(capsIRB,"tool0",[0.15 0],trvec2tform([0.15 0 0]))
show(capsIRB,homeConfiguration(capsIRB.RigidBodyTree));
```



Again check tool0 for a collision capsule, and verify the properties of the detected capsule.

```
capsulesTool = getCapsules(capsIRB,"tool0")
```

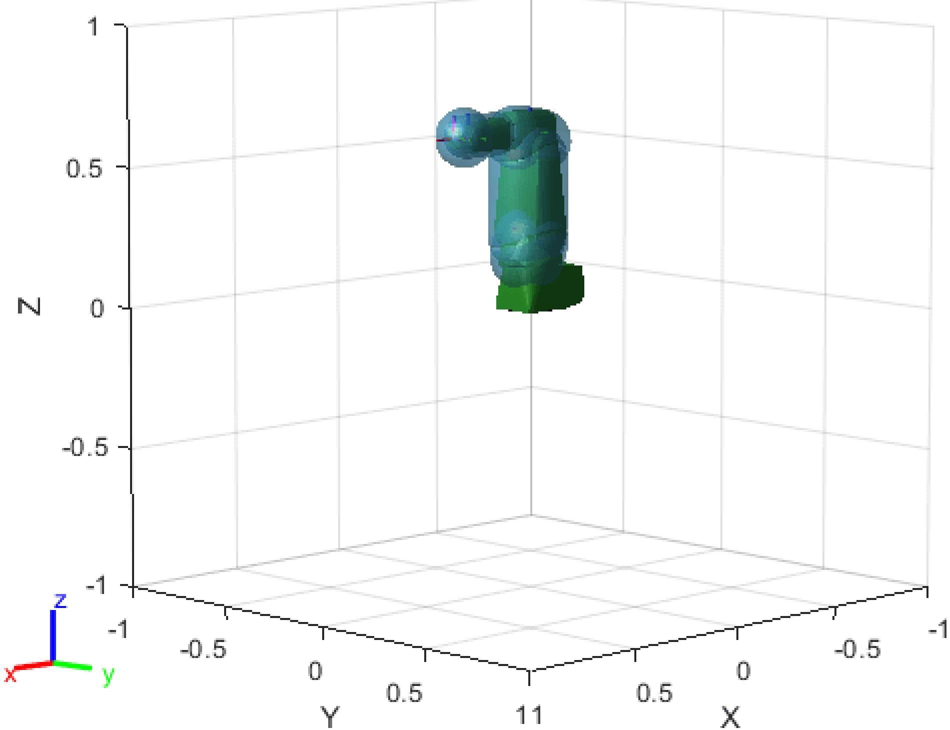
```
capsulesTool = 1x1 cell array
    {1x1 collisionCapsule}
```

```
capsulesTool{1}
```

```
ans =
  collisionCapsule with properties:

    Radius: 0.1500
    Length: 0
      Pose: [4x4 double]
```

Remove the capsule from the base link. Then, reduce the collision capsule size of `tool0`, and move it `-0.05` meters from the previous position along the *x*-axis.

```
removeCapsule(capsIRB,"base_link",1)
updatePose(capsIRB,"tool0",trvec2tform([-0.05 0 0]),1)
updateGeometry(capsIRB,"tool0",[.1 0.01],1)
show(capsIRB,homeConfiguration(capsIRB.RigidBodyTree));
```



Input Arguments

capapprox — Capsule approximation of rigid body tree
`capsuleApproximation` object

Capsule approximation of a rigid body tree, specified as a `capsuleApproximation` object.

bodyname — Name of rigid body
string scalar | character vector

Name of the rigid body, specified as a string scalar or character vector. The rigid body must exist in the `rigidBodyTree` object of the `RigidBodyTree` property of `capsapprox`.

Example: `"EndEffectorTool"`

Data Types: `char` | `string`

idx — Index of collision capsule in rigid body
nonnegative integer

Index of the collision capsule in the rigid body, specified as a nonnegative integer.

Example: 5

pose — Updated pose for collision capsule
4-by-4 matrix

Updated pose for the collision capsule, specified as a 4-by-4 homogeneous transformation matrix defined with respect to the frame of the rigid body `bodyname`.

Example: `eye(4)`

Version History

Introduced in R2022b

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

Objects

`capsuleApproximation`

Functions

`addCapsule` | `removeCapsule` | `show` | `getCapsules` | `updateGeometry`

genspheres

Generate spheres along central line segment of capsule

Syntax

`spheres = genspheres(capsule,ratio)`

Description

`spheres = genspheres(capsule,ratio)` generates spheres along the central line segment of the collision capsule `capsule` at the specified normalized positions `ratio` of the line segment.

Examples

Generate Collision Spheres Inside Collision Capsule

Create a collision capsule with a radius of `2` and length of `10`. Visualize the capsule.

```
cCapsule = collisionCapsule(2,10);
[~,p] = show(cCapsule);
```

Generate spheres at ratios `0.0`, `0.5`, and `1.0` of the capsule length.

```
spheres = genspheres(cCapsule,linspace(0,1,3));
```

Display the positions of the spheres.

```
for i = 1:length(spheres)
    disp(tform2trvec(spheres{i}.Pose))
end
```

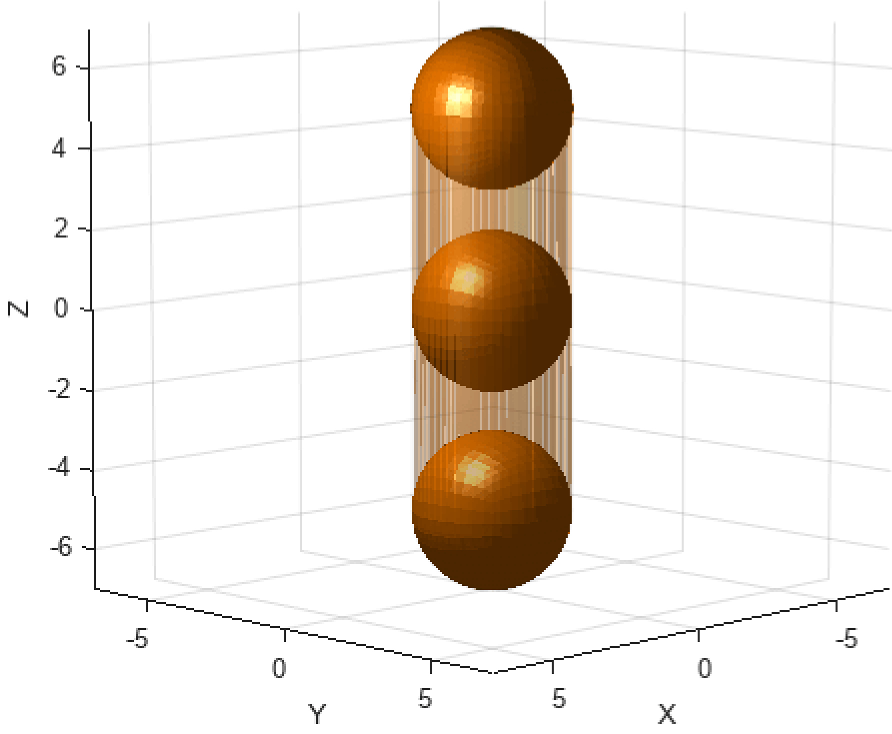
```
     0     0    -5

     0     0     0

     0     0     5
```

Set the face and edge alphas of the capsule to low values. This ensures that both the spheres are visible when you add them to the figure.

```
p.FaceAlpha = 0.4;
p.EdgeAlpha = 0.01;
hold on
```

Display the generated spheres on the capsule.

```
cellfun(@show,spheres);
```



Input Arguments

`capsule` — Collision capsule
`collisionCapsule` object

Collision capsule, specified as a `collisionCapsule` object.

Example: `collisionCapsule(3,5)`

`ratio` — Normalized positions along central line segment of collision capsule
N-element row vector of values in range [0, 1]

Normalized positions along the central line segment of the collision capsule, specified as an *N*-element row vector of values in the range [0, 1]. *N* is the number of collision spheres to generate. Each element specifies the position of a sphere as a percentage of the central segment length.

Example: For collision capsule with a central line segment length of 4 meters, a ratio position vector `[0.25 0.5 0.75]` generates collision spheres at 1, 2, and 3 meters along the central line segment of the collision capsule.

Output Arguments

`spheres` — Collision spheres
N-element cell array

Collision spheres, returned as an *N*-element cell array of `collisionSphere` objects, where *N* is the number of generated collision spheres.

Version History

Introduced in R2022b

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

The `ratio` input argument must be specified as a compile-time constant during code generation.

See Also

Objects

`collisionCapsule`

Functions

`convertToCollisionMesh` | `show` | `checkCollision`

convertToCollisionMesh

Convert collision primitive geometry into collision mesh geometry

Syntax

```
collisionMesh = convertToCollisionMesh(collisionObj)
```

Description

`collisionMesh = convertToCollisionMesh(collisionObj)` converts a collision primitive geometry, `collisionObj`, to a convex mesh collision geometry, `collisionMesh`, which retains the pose of `collisionObj`.

Note Because converting a collision primitive to a collision mesh discretizes the underlying primitive, the converted mesh can return a different `checkCollision` result than the primitive equivalent.

Examples

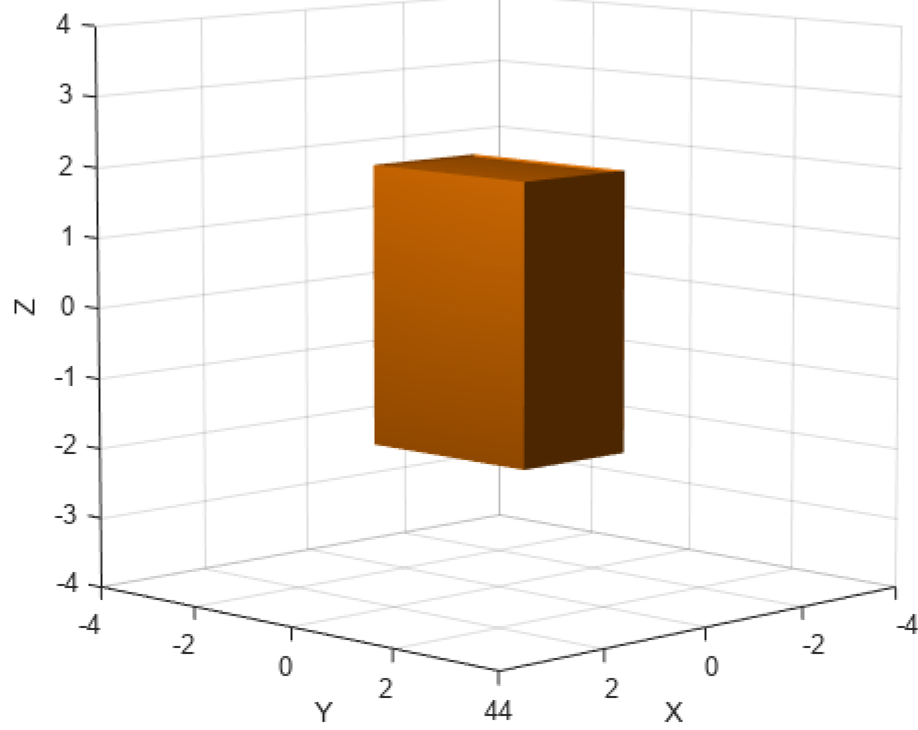
Convert Collision Geometry to Collision Mesh

Create and visualize a box as a collision geometry object.

```
box = collisionBox(2,3,4)
```

```
box = 
  collisionBox with properties:

       X: 2
       Y: 3
       Z: 4
    Pose: [4x4 double]
```

```
show(box);
```

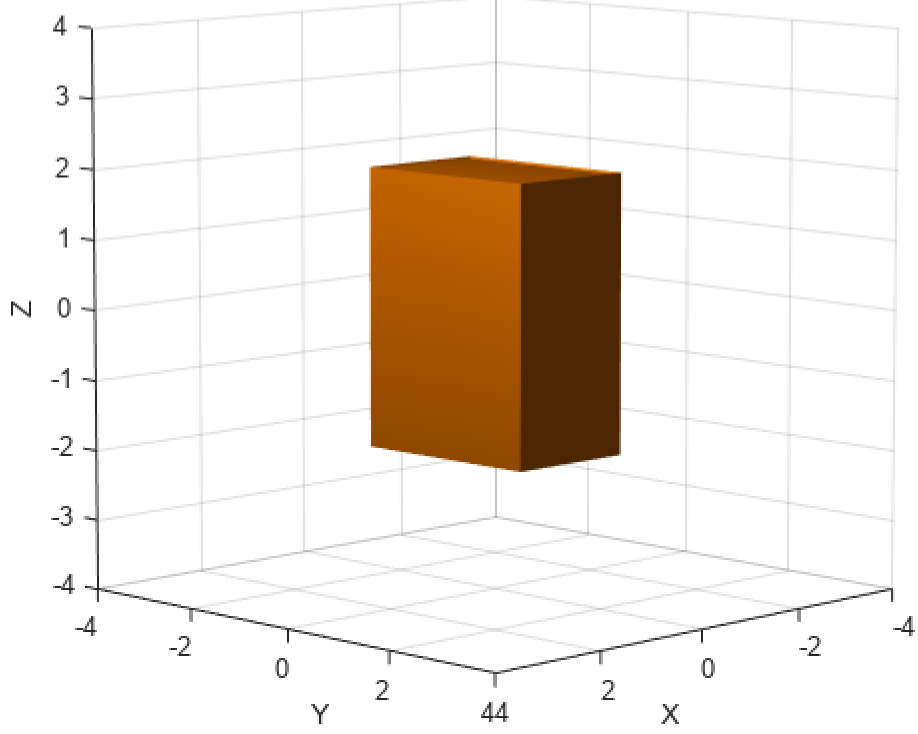


Convert the collision box to a mesh. Visualize the mesh.

```
mesh = convertToCollisionMesh(box)
```

```
mesh = 
  collisionMesh with properties:

    Vertices: [8x3 double]
        Pose: [4x4 double]
```

```
show(mesh);
```



Input Arguments

`collisionObj` — Collision geometry object
`collisionBox` object | `collisionSphere` object | `collisionCylinder` object | `collisionCapsule` object

Collision geometry object, specified as a `collisionBox`, `collisionSphere`, `collisionCylinder`, or `collisionCapsule` object. The function converts this object into a collision mesh.

Output Arguments

`collisionMesh` — Collision mesh
`collisionMesh` object

Collision mesh, returned as a `collisionMesh` object. This object is the mesh equivalent of the specified collision geometry object.

Version History

Introduced in R2022a

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

Objects
`collisionMesh` | `collisionBox` | `collisionSphere` | `collisionCylinder` | `collisionCapsule`

Topics
"Generate Code for Manipulator Motion Planning in Perceived Environment"

fitCollisionCapsule

Fit collision capsule around collision geometry

Syntax

`[collCapsule,fitInfo] = fitCollisionCapsule(geom)`

Description

`[collCapsule,fitInfo] = fitCollisionCapsule(geom)` fits a collision capsule `collCapsule` around a collision geometry `geom`.

Examples

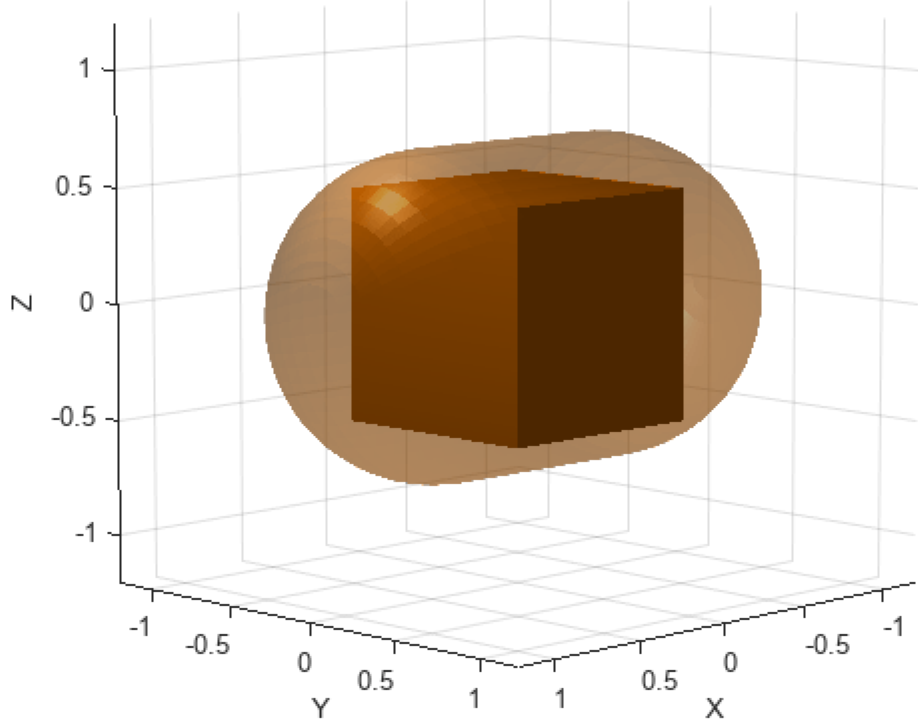
Fit Collision Capsule Around Collision Box

Create a box with a length, width, and height of `1` meter and fit a collision capsule on it.

```
box = collisionBox(1,1,1);
show(box);
hold on
[collcaps,fitinfo]= fitCollisionCapsule(box);
```

Visualize the new collision capsule on top of the box and set the alphas of the capsule to a low value so that the box is visible.

```
[~,capvis] = show(collcaps);
capvis.FaceAlpha=0.4;
xlim auto
ylim auto
zlim auto
```



Input Arguments

geom — Collision geometry
`collisionBox` object | `collisionCylinder` object | `collisionSphere` object | `collisionMesh` object

Collision geometry to fit capsule onto, specified as either a `collisionBox`, `collisionSphere`, `collisionCylinder`, or `collisionMesh` object.

Output Arguments

`collCapsule` — Collision capsule of the collision geometry
`collisionCapsule` object

Collision capsule of the collision geometry, returned as a `collisionCapsule` object

`fitInfo` — Fit information of collision capsule
structure

Fit information of the collision capsule, returned as a structure. The structure contains the `Residual` field, returned as an N-element vector, where N is the total number of points of the collision geometry. Each element of the vector specifies the residual of a point of the collision geometry as:

$$\left|(o_{cg} - l_{cc})\right| + r_{cc}$$

where:

- o_{cg} is the origin of the fitted collision object.
- l_{cc} is the closest point of the central line of the collision capsule to o_{cg}.
- r_{cc} is the radius of the collision capsule.

Version History

Introduced in R2022b

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

Objects

`collisionBox` | `collisionSphere` | `collisionCylinder` | `collisionMesh` | `collisionCapsule`

Functions

`convertToCollisionMesh` | `show` | `checkCollision` | `genspheres`

show

Show collision geometry

Syntax

```
show(geom)
show(geom,"Parent",AX)

ax = show( ___ )
[ax,patchobj] = show( ___ )
```

Description

`show(geom)` shows the collision geometry in the current figure at its current pose. The function automatically generates the tessellation.

`show(geom,"Parent",AX)` specifies the axes `AX` in which to plot the collision geometry.

`ax = show( ___ )` returns the axes on which you plot the collision geometry.

`[ax,patchobj] = show( ___ )` returns the graphic object `patchobj` that represents the collision geometry in the plot.

Examples

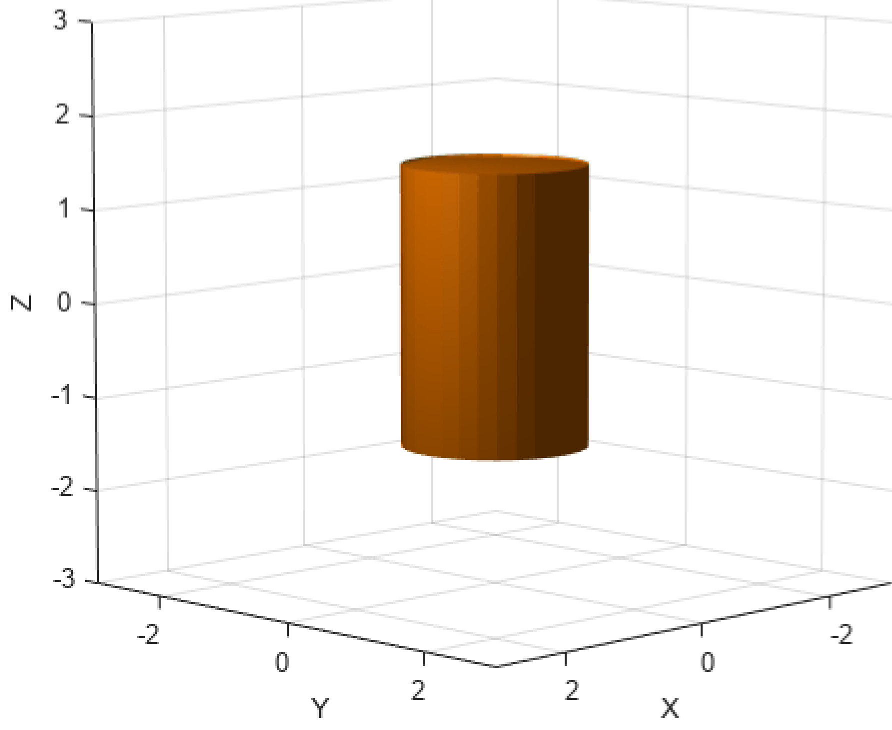
Show Collision Geometry

Create a cylinder collision geometry. The cylinder has a length of 3 meters and a radius of 1 meter.

```
cyl = collisionCylinder(1,3);
```

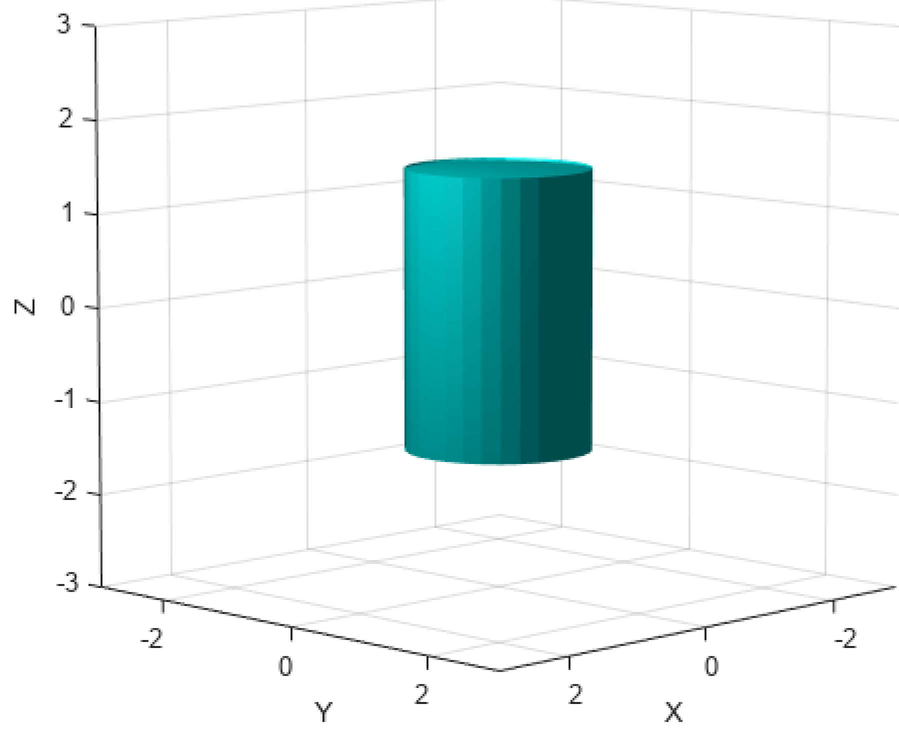
Show the cylinder.

```
show(cyl)
```



Show the cylinder in a new figure, and return the patch object that represents the cylinder. Change the cylinder color to cyan by changing the RGB value of the `FaceColor` field in the patch object. Hide the edges by setting `EdgeColor` to `'none'`.

```
figure
[~,patchObj] = show(cyl);
patchObj.FaceColor = [0 1 1];
patchObj.EdgeColor = 'none';
```



Input Arguments

geom — Collision geometry
`collisionBox` object | `collisionCapsule` object | `collisionCylinder` object | `collisionMesh` object | `collisionSphere` object

Collision geometry to show, specified as one of these objects:

- `collisionBox`
- `collisionCapsule`
- `collisionCylinder`
- `collisionMesh`
- `collisionSphere`

AX — Axes on which to plot collision geometry
`Axes` object

Axes on which to plot the collision geometry, specified as an `Axes` object.

Output Arguments

ax — Axes displaying collision geometry
`Axes` object

Axes displaying the collision geometry, returned as an `Axes` object. For more information, see Axes Properties.

`patchobj` — Graphic object
`Patch` object

Graphic object that represents the collision geometry, returned as a `Patch` object. For more information, see Patch Properties.

Version History

Introduced in R2019b

See Also

`collisionBox` | `collisionCylinder` | `collisionMesh` | `collisionSphere` | `collisionCapsule`

info

Characteristic information about `controllerPurePursuit` object

Syntax

```
controllerInfo = info(controller)
```

Description

`controllerInfo = info(controller)` returns a structure, `controllerInfo`, with additional information about the status of the `controllerPurePursuit` object, `controller`. The structure contains the fields, `RobotPose` and `LookaheadPoint`.

Examples

Get Additional Pure Pursuit Object Information

Use the `info` method to get more information about a `controllerPurePursuit` object. The `info` function returns two fields, `RobotPose` and `LookaheadPoint`, which correspond to the current position and orientation of the robot and the point on the path used to compute outputs from the last call of the object.

Create a `controllerPurePursuit` object.

```
pp = controllerPurePursuit;
```

Assign waypoints.

```
pp.Waypoints = [0 0;1 1];
```

Compute control commands using the `pp` object with the initial pose `[x y theta]` given as the input.

```
[v,w] = pp([0 0 0]);
```

Get additional information.

```
s = info(pp)
```

```
s = struct with fields:
         RobotPose: [0 0 0]
    LookaheadPoint: [0.7071 0.7071]
```

Input Arguments

`controller` — Pure pursuit controller
`controllerPurePursuit` object

Pure pursuit controller, specified as a `controllerPurePursuit` object.

Output Arguments

`controllerInfo` — Information on the `controllerPurePursuit` object
structure

Information on the `controllerPurePursuit` object, returned as a structure. The structure contains two fields:

- `RobotPose` – A three-element vector in the form `[x y theta]` that corresponds to the *x*-*y* position and orientation of the vehicle. The angle, `theta`, is measured in radians with positive angles measured counterclockwise from the *x*-axis.
- `LookaheadPoint`– A two-element vector in the form `[x y]`. The location is a point on the path that was used to compute outputs of the last call to the object.

Version History

Introduced in R2019b

See Also

`controllerPurePursuit`

Topics

"Pure Pursuit Controller"

applyTransform

Apply forward transformation to mesh vertices

Syntax

```
transformedMesh = applyTransform(mesh,T)
```

Description

`transformedMesh = applyTransform(mesh,T)` applies the forward transformation matrix `T` to the vertices of the object mesh.

Examples

Create and Transform Cuboid Mesh

Create an `extendedObjectMesh` object and transform the object by using a transformation matrix.

Create a cuboid mesh of unit dimensions.

```
cuboid = extendedObjectMesh('cuboid');
```

Create a transformation matrix that is a combination of a translation, a scaling, and a rotation.

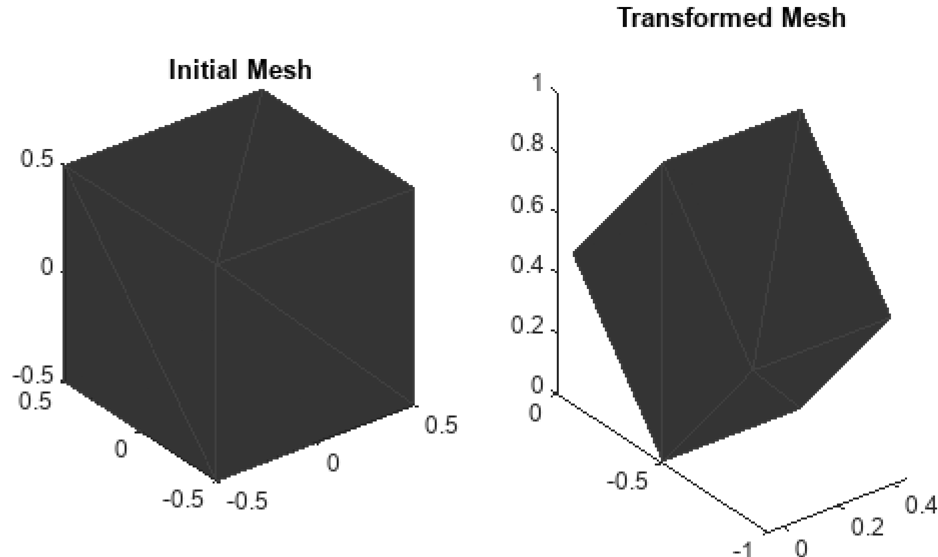
```
tform = makehgtform('translate',[0.2 -0.5 0.5], ...
    'scale',[0.5 0.6 0.7], ...
    'xrotate',pi/4);
```

Transform the mesh.

```
transformedCuboid = applyTransform(cuboid,tform);
```

Visualize the meshes.

```
subplot(1,2,1);
show(cuboid);
title('Initial Mesh')

subplot(1,2,2);
show(transformedCuboid);
title('Transformed Mesh')
```



Input Arguments

`mesh` — Extended object mesh
`extendedObjectMesh` object

Extended object mesh, specified as an `extendedObjectMesh` object.

`T` — Transformation matrix
4-by-4 matrix

Transformation matrix applied on the object mesh, specified as a 4-by-4 matrix. The 3-D coordinates of each point in the object mesh is transformed according to this formula:

```
[xT; yT; zT; 1] = T*[x; y; z; 1]
```

`xT`, `yT`, and `zT` are the transformed 3-D coordinates of the point.

Data Types: `single` | `double`

Output Arguments

`transformedMesh` — Transformed object mesh
`extendedObjectMesh` object

Transformed object mesh, returned as an `extendedObjectMesh` object.

Version History

Introduced in R2022a

See Also

Objects
`extendedObjectMesh`

Functions
`join` | `rotate` | `scale` | `scaleToFit` | `show` | `translate`

join

Join two object meshes

Syntax

```
joinedMesh = join(mesh1,mesh2)
```

Description

`joinedMesh = join(mesh1,mesh2)` joins the object meshes `mesh1` and `mesh2` and returns `joinedMesh` with the combined objects.

Examples

Create and Join Two Object Meshes

Create `extendedObjectMesh` objects and join them together.

Construct two meshes of unit dimensions.

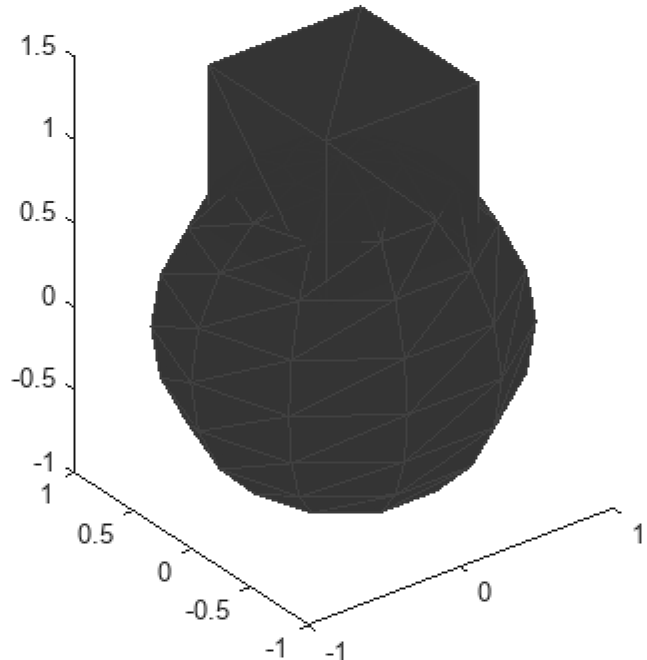
```
sph = extendedObjectMesh('sphere');
cub = extendedObjectMesh('cuboid');
```

Join the two meshes.

```
cub = translate(cub,[0 0 1]);
sphCub = join(sph,cub);
```

Visualize the final mesh.

```
show(sphCub);
```



Input Arguments

`mesh1` — Extended object mesh
`extendedObjectMesh` object

Extended object mesh, specified as an `extendedObjectMesh` object.

`mesh2` — Extended object mesh
`extendedObjectMesh` object

Extended object mesh, specified as an `extendedObjectMesh` object.

Output Arguments

`joinedMesh` — Joined object mesh
`extendedObjectMesh` object

Joined object mesh, specified as an `extendedObjectMesh` object.

Version History

Introduced in R2022a

See Also

Objects
extendedObjectMesh

Functions
applyTransform | rotate | scale | scaleToFit | show | translate

rotate

Rotate mesh about coordinate axes

Syntax

```
rotatedMesh = rotate(mesh,orient)
```

Description

`rotatedMesh = rotate(mesh,orient)` rotate the `mesh` object by an orientation, `orient`.

Examples

Create and Rotate Cuboid Mesh

Create an `extendedObjectMesh` object and rotate the object.

Construct a cuboid mesh.

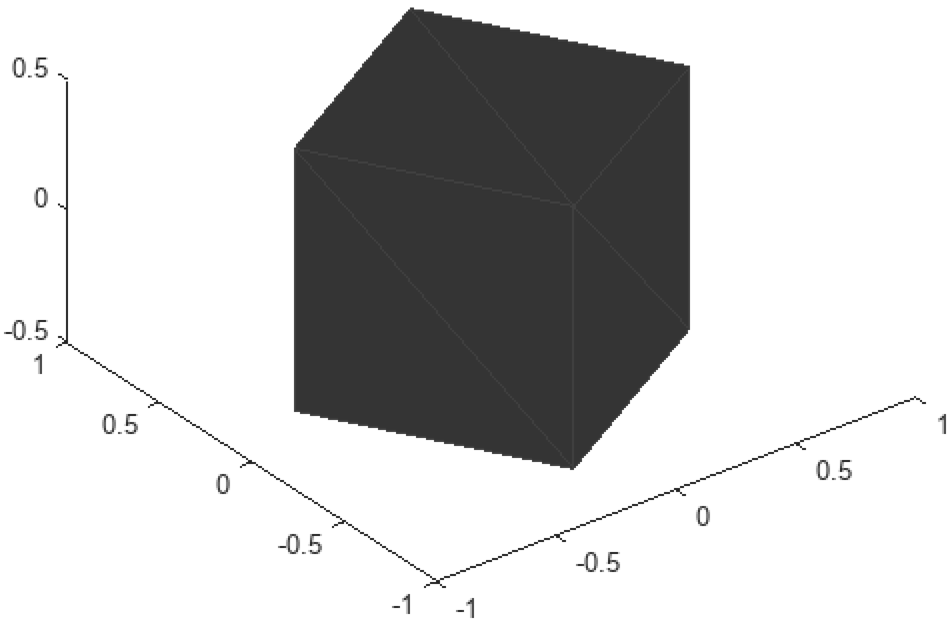
```
mesh = extendedObjectMesh('cuboid');
```

Rotate the mesh by 30 degrees around the *z* axis.

```
mesh = rotate(mesh,[30 0 0]);
```

Visualize the mesh.

```
ax = show(mesh);
```



Input Arguments

`mesh` — Extended object mesh
`extendedObjectMesh` object

Extended object mesh, specified as an `extendedObjectMesh` object.

`orient` — Description of rotation
3-by-3 orthonormal matrix | quaternion | 1-by-3 vector

Description of rotation for an object mesh, specified as:

- 3-by-3 orthonormal rotation matrix
- `quaternion`
- 1-by-3 vector, where the elements are positive rotations in degrees about the *z*, *y*, and *x* axes, in that order.

Output Arguments

`rotatedMesh` — Rotated object mesh
`extendedObjectMesh` object

Rotated object mesh, returned as an `extendedObjectMesh` object.

Version History

Introduced in R2022a

See Also

Objects

`extendedObjectMesh`

Functions

`applyTransform` | `join` | `scale` | `scaleToFit` | `show` | `translate`

scale

Scale mesh in each dimension

Syntax

```
scaledMesh = scale(mesh,scaleFactor)
scaledMesh = scale(mesh,[sx sy sz])
```

Description

`scaledMesh = scale(mesh,scaleFactor)` scales the object mesh by `scaleFactor`. `scaleFactor` can be the same for all dimensions or defined separately as elements of a 1-by-3 vector in the order *x*, *y*, and *z*.

`scaledMesh = scale(mesh,[sx sy sz])` scales the object mesh along the dimensions *x*, *y*, and *z* by the scaling factors `sx`, `sy`, and `sz`.

Examples

Create and Scale Cuboid Mesh

Create an `extendedObjectMesh` object and scale the object.

Construct a cuboid mesh of unit dimensions.

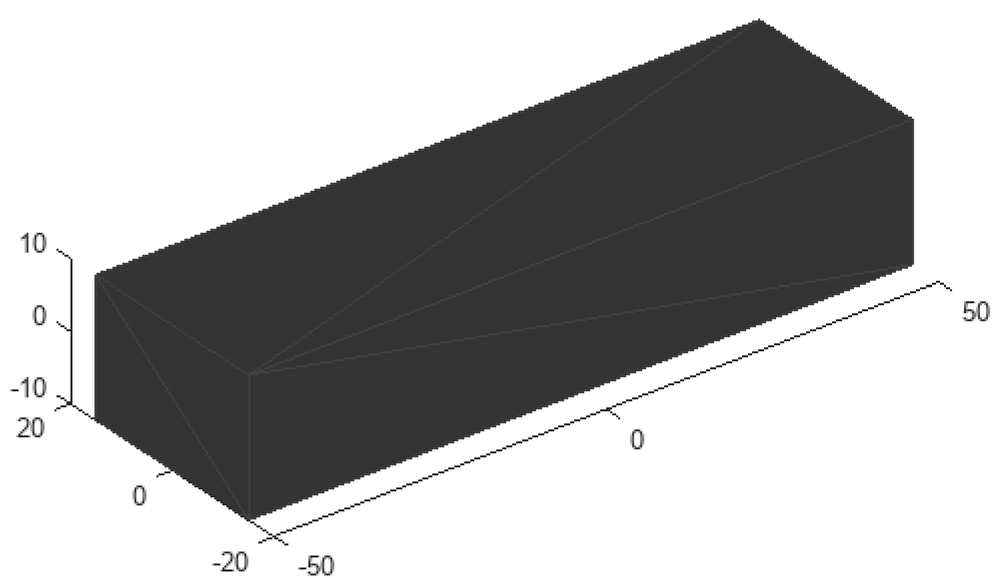
```
cuboid = extendedObjectMesh('cuboid');
```

Scale the mesh by different factors along each of the three axes.

```
scaledCuboid = scale(cuboid,[100 30 20]);
```

Visualize the mesh.

```
show(scaledCuboid);
```



Input Arguments

`mesh` — Extended object mesh
`extendedObjectMesh` object

Extended object mesh, specified as an `extendedObjectMesh` object.

`scaleFactor` — Scaling factor
positive real scalar | 1-by-3 vector

Scaling factor for the object mesh, specified as a positive real scalar or as a 1-by-3 vector in the order *x*, *y*, and *z*.

Data Types: `single` | `double`

`sx` — Scaling factor for *x*-axis
positive real scalar

Scaling factor for *x*-axis, specified as a positive real scalar.

Data Types: `single` | `double`

`sy` — Scaling factor for *y*-axis
positive real scalar

Scaling factor for *y*-axis, specified as a positive real scalar.

Data Types: `single` | `double`

`sz` — Scaling factor for *z*-axis
positive real scalar

Scaling factor for *z*-axis, specified as a positive real scalar.

Data Types: `single` | `double`

Output Arguments

`scaledMesh` — Scaled object mesh
`extendedObjectMesh` object

Scaled object mesh, returned as an `extendedObjectMesh` object.

Version History

Introduced in R2022a

See Also

Objects
`extendedObjectMesh`

Functions
`applyTransform` | `join` | `rotate` | `scaleToFit` | `show` | `translate`

scaleToFit

Auto-scale object mesh to match specified cuboid dimensions

Syntax

```
scaledMesh = scaleToFit(mesh,dims)
```

Description

`scaledMesh = scaleToFit(mesh,dims)` auto-scales the object mesh to match the dimensions of a cuboid specified in the structure `dims`.

Examples

Create and Auto-Scale Sphere Mesh

Create an `extendedObjectMesh` object and auto-scale the object to the required dimensions.

Construct a sphere mesh of unit dimensions.

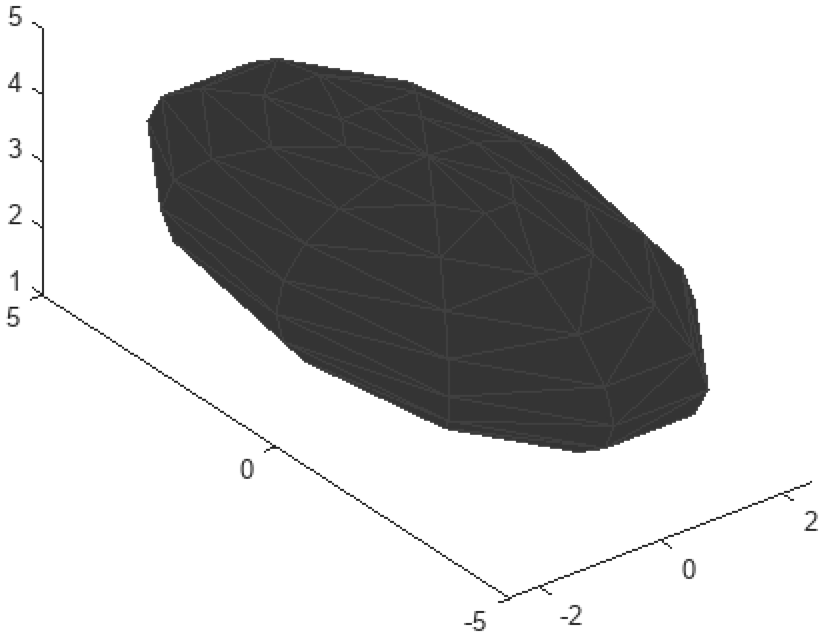
```
sph = extendedObjectMesh('sphere');
```

Auto-scale the mesh to the dimensions in `dims`.

```
dims = struct('Length',5,'Width',10,'Height',3,'OriginOffset',[0 0 -3]);
sph = scaleToFit(sph,dims);
```

Visualize the mesh.

```
show(sph);
```



Input Arguments

mesh — Extended object mesh
extendedObjectMesh object

Extended object mesh, specified as an extendedObjectMesh object.

dims — Cuboid dimensions
structure

Dimensions of the cuboid to scale an object mesh, specified as a struct with these fields:

- Length – Length of the cuboid
- Width – Width of the cuboid
- Height – Height of the cuboid
- OriginOffset – Origin offset in 3-D coordinates

All the dimensions are in meters.

Data Types: struct

Output Arguments

scaledMesh — Scaled object mesh
extendedObjectMesh object

Scaled object mesh, returned as an extendedObjectMesh object.

Version History

Introduced in R2022a

See Also

Objects
extendedObjectMesh

Functions
applyTransform | join | rotate | scale | show | translate

show

Display the mesh as a patch on the current axes

Syntax

```
show(mesh)
show(mesh,ax)
ax = show(mesh)
```

Description

`show(mesh)` displays the `extendedObjectMesh` as a patch on the current axes. If there are no active axes, the function creates new axes.

`show(mesh,ax)` displays the object mesh as a patch on the axes `ax`.

`ax = show(mesh)` optionally outputs the handle to the axes where the mesh was plotted.

Examples

Create and Translate Cuboid Mesh

Create an `extendedObjectMesh` object and translate the object.

Construct a cuboid mesh.

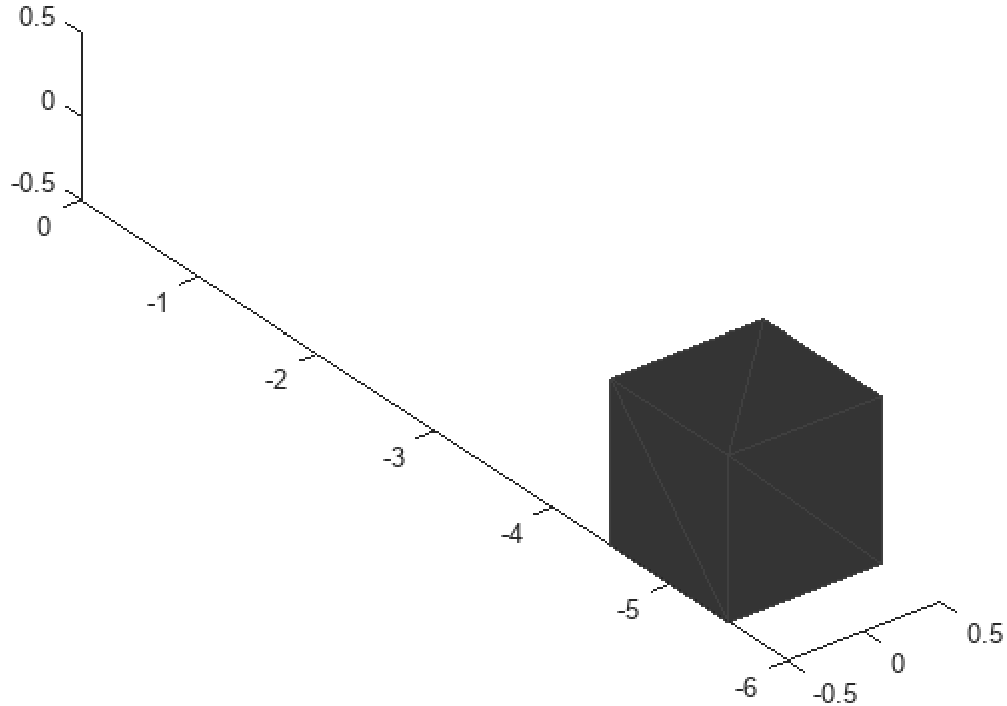
```
mesh = extendedObjectMesh('cuboid');
```

Translate the mesh by 5 units along the negative *y* axis.

```
mesh = translate(mesh,[0 -5 0]);
```

Visualize the mesh.

```
ax = show(mesh);
ax.YLim = [-6 0];
```



Input Arguments

`mesh` — Extended object mesh
`extendedObjectMesh` object

Extended object mesh, specified as an `extendedObjectMesh` object.

ax — Current axes
`axes` object

Current axes, specified as an `axes` object.

Version History

Introduced in R2022a

See Also

Objects
`extendedObjectMesh`

Functions
`applyTransform` | `join` | `rotate` | `scale` | `scaleToFit` | `translate`

translate

Translate mesh along coordinate axes

Syntax

```
translatedMesh = translate(mesh,deltaPos)
```

Description

`translatedMesh = translate(mesh,deltaPos)` translates the object mesh by the distances specified by `deltaPos` along the coordinate axes.

Examples

Create and Translate Cuboid Mesh

Create an `extendedObjectMesh` object and translate the object.

Construct a cuboid mesh.

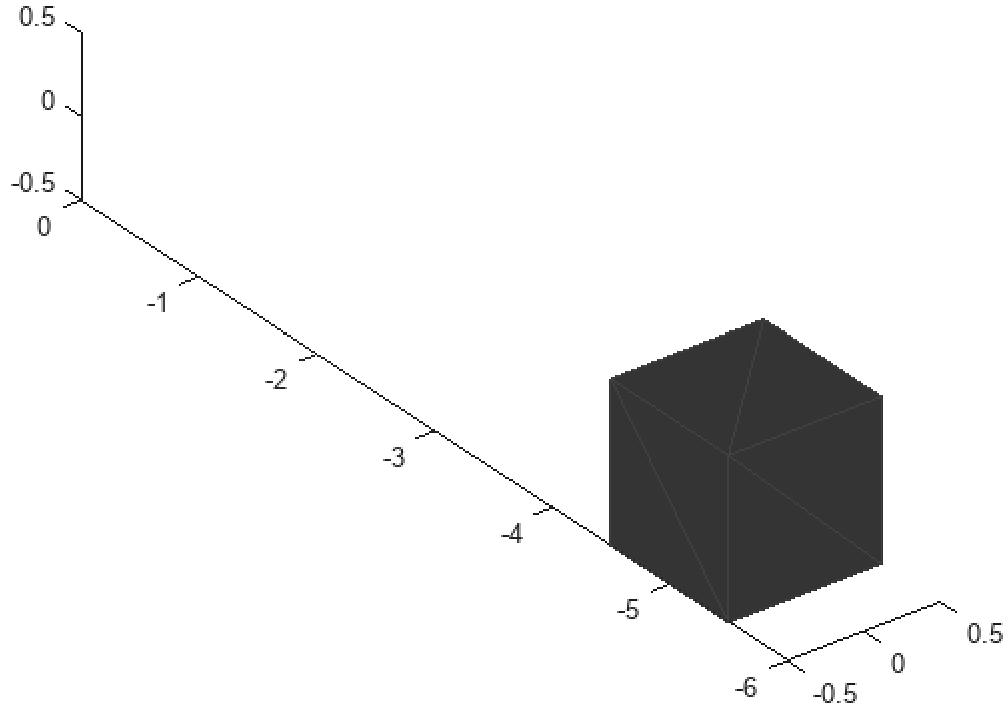
```
mesh = extendedObjectMesh('cuboid');
```

Translate the mesh by 5 units along the negative *y* axis.

```
mesh = translate(mesh,[0 -5 0]);
```

Visualize the mesh.

```
ax = show(mesh);
ax.YLim = [-6 0];
```



Input Arguments

`mesh` — Extended object mesh
`extendedObjectMesh` object

Extended object mesh, specified as an `extendedObjectMesh` object.

`deltaPos` — Translation vector
three-element real-valued vector

Translation vector for an object mesh, specified as a three-element real-valued vector. The three elements in the vector define the translation along the *x*, *y*, and *z* axes.

Data Types: `single` | `double`

Output Arguments

`translatedMesh` — Translated object mesh
`extendedObjectMesh` object

Translated object mesh, returned as an `extendedObjectMesh` object.

Version History

Introduced in R2022a

See Also

Objects

`extendedObjectMesh`

Functions

`applyTransform` | `join` | `rotate` | `scale` | `scaleToFit` | `show`

perturb

Apply perturbations to object

Syntax

```
offsets = perturb(obj)
```

Description

`offsets = perturb(obj)` applies the perturbations defined on the object, `obj` and returns the offset values. You can define perturbations on the object by using the `perturbations` function.

Examples

Perturb Waypoint Trajectory

Define a waypoint trajectory. By default, this trajectory contains two waypoints.

```
traj = waypointTrajectory
```

```
traj = 
  waypointTrajectory with properties:

         SampleRate: 100
     SamplesPerFrame: 1
          Waypoints: [2x3 double]
      TimeOfArrival: [2x1 double]
         Velocities: [2x3 double]
             Course: [2x1 double]
        GroundSpeed: [2x1 double]
          ClimbRate: [2x1 double]
        Orientation: [2x1 quaternion]
          AutoPitch: 0
           AutoBank: 0
     ReferenceFrame: 'NED'
```

Define perturbations on the `Waypoints` property and the `TimeOfArrival` property.

```
rng(2020);
perturbs1 = perturbations(traj,'Waypoints','Normal',1,1)
```

```
perturbs1=2×3 table
       Property           Type              Value        
    _______________    ________    __________________

    "Waypoints"        "Normal"    {[  1]}    {[  1]}
    "TimeOfArrival"    "None"      {[NaN]}    {[NaN]}
```

```
perturbs2 = perturbations(traj,'TimeOfArrival','Selection',{[0;1],[0;2]})
```

```
perturbs2=2×3 table
       Property           Type                     Value
    ________________    ___________    _________________________________

    "Waypoints"         "Normal"       {[     1]}    {[            1]}
    "TimeOfArrival"     "Selection"    {1x2 cell}    {[0.5000 0.5000]}
```

Perturb the trajectory.

```
offsets = perturb(traj)
```

```
offsets=2×1 struct array with fields:
    Property
    Offset
    PerturbedValue
```

The `Waypoints` property and the `TimeOfArrival` property have changed.

```
traj.Waypoints
```

```
ans = 2×3

    1.8674    1.0203    0.7032
    2.3154   -0.3207    0.0999
```

```
traj.TimeOfArrival
```

```
ans = 2×1

     0
     2
```

Perturb Accuracy of insSensor

Create an `insSensor` object.

```
sensor = insSensor
```

```
sensor = 
  insSensor with properties:

           MountingLocation: [0 0 0]            m     
              RollAccuracy: 0.2                 deg   
             PitchAccuracy: 0.2                 deg   
               YawAccuracy: 1                   deg   
          PositionAccuracy: [1 1 1]             m     
          VelocityAccuracy: 0.05                m/s   
      AccelerationAccuracy: 0                   m/s²  
   AngularVelocityAccuracy: 0                   deg/s 
                 TimeInput: 0                         
              RandomStream: 'Global stream'           
```

Define the perturbation on the `RollAccuracy` property as three values with an equal possibility each.

```
values = {0.1 0.2 0.3}
```

```
values=1×3 cell array
    {[0.1000]}    {[0.2000]}    {[0.3000]}
```

```
probabilities = [1/3 1/3 1/3]
```

```
probabilities = 1×3

    0.3333    0.3333    0.3333
```

```
perturbations(sensor,'RollAccuracy','Selection',values,probabilities)
```

```
ans=7×3 table
            Property                Type                       Value                
    _________________________    ___________    ______________________________________

    "RollAccuracy"               "Selection"    {1x3 cell}    {[0.3333 0.3333 0.3333]}
    "PitchAccuracy"              "None"         {[   NaN]}    {[                 NaN]}
    "YawAccuracy"                "None"         {[   NaN]}    {[                 NaN]}
    "PositionAccuracy"           "None"         {[   NaN]}    {[                 NaN]}
    "VelocityAccuracy"           "None"         {[   NaN]}    {[                 NaN]}
    "AccelerationAccuracy"       "None"         {[   NaN]}    {[                 NaN]}
    "AngularVelocityAccuracy"    "None"         {[   NaN]}    {[                 NaN]}
```

Perturb the `sensor` object using the perturb function.

```
rng(2020)
perturb(sensor);
sensor
```

```
sensor = 
  insSensor with properties:

           MountingLocation: [0 0 0]            m     
               RollAccuracy: 0.5                deg   
              PitchAccuracy: 0.2                deg   
                YawAccuracy: 1                  deg   
           PositionAccuracy: [1 1 1]            m     
           VelocityAccuracy: 0.05               m/s   
       AccelerationAccuracy: 0                  m/s²  
    AngularVelocityAccuracy: 0                  deg/s 
                  TimeInput: 0                        
               RandomStream: 'Global stream'          
```

The `RollAccuracy` is perturbed to `0.5` deg.

Input Arguments

obj — Object for perturbation
objects

Object for perturbation, specified as an object. The objects that you can perturb include:

- insSensor
- waypointTrajectory

Output Arguments

offsets — Property offsets
array of structure

Property offsets, returned as an array of structures. Each structure contains these fields:

Field Name	Description
Property	Name of perturbed property
Offset	Offset values applied in the perturbation
PerturbedValue	Property values after the perturbation

Version History

Introduced in R2022a

See Also

perturbations

perturbations

Perturbation defined on object

Syntax

```
perturbs = perturbations(obj)
perturbs = perturbations(obj,property)
perturbs = perturbations(obj,property,'None')
perturbs = perturbations(obj,property,'Selection',values,probabilities)
perturbs = perturbations(obj,property,'Normal',mean,deviation)
perturbs = perturbations(obj,property,'TruncatedNormal',mean,deviation,
lowerLimit,upperLimit)
perturbs = perturbations(obj,property,'Uniform',minVal,maxVal)
perturbs = perturbations(obj,property,'Custom',perturbFcn)
```

Description

`perturbs = perturbations(obj)` returns the list of property perturbations, `perturbs`, defined on the object, `obj`. The returned `perturbs` lists all the perturbable properties. If any property is not perturbed, then its corresponding `Type` is returned as `"Null"` and its corresponding `Value` is returned as `{Null,Null}`.

`perturbs = perturbations(obj,property)` returns the current perturbation applied to the specified `property`.

`perturbs = perturbations(obj,property,'None')` defines a `property` that must not be perturbed.

`perturbs = perturbations(obj,property,'Selection',values,probabilities)` defines the `property` perturbation offset drawn from a set of `values` that have corresponding `probabilities`.

`perturbs = perturbations(obj,property,'Normal',mean,deviation)` defines the `property` perturbation offset drawn from a normal distribution with specified `mean` and standard `deviation`.

`perturbs = perturbations(obj,property,'TruncatedNormal',mean,deviation, lowerLimit,upperLimit)` defines the `property` perturbation offset drawn from a normal distribution with specified `mean`, standard `deviation`, lower limit, and upper limit.

`perturbs = perturbations(obj,property,'Uniform',minVal,maxVal)` defines the `property` perturbation offset drawn from a uniform distribution on an interval [`minVal`, `maxValue`].

`perturbs = perturbations(obj,property,'Custom',perturbFcn)` enables you to define a custom function, `perturbFcn`, that draws the perturbation offset value.

Examples

Default Perturbation Properties of `waypointTrajectory`

Create a `waypointTrajectory` object.

```
traj = waypointTrajectory;
```

Show the default perturbation properties using the `perturbations` method.

```
perturbs = perturbations(traj)
```

```
perturbs=2×3 table
       Property          Type            Value
    _______________    ______    _________________

    "Waypoints"        "None"    {[NaN]}    {[NaN]}
    "TimeOfArrival"    "None"    {[NaN]}    {[NaN]}
```

Perturb Accuracy of insSensor

Create an `insSensor` object.

```
sensor = insSensor
```

```
sensor = 
  insSensor with properties:

           MountingLocation: [0 0 0]            m
               RollAccuracy: 0.2                deg
              PitchAccuracy: 0.2                deg
                YawAccuracy: 1                  deg
           PositionAccuracy: [1 1 1]            m
           VelocityAccuracy: 0.05               m/s
       AccelerationAccuracy: 0                  m/s²
    AngularVelocityAccuracy: 0                  deg/s
                  TimeInput: 0
               RandomStream: 'Global stream'
```

Define the perturbation on the `RollAccuracy` property as three values with an equal possibility each.

```
values = {0.1 0.2 0.3}
```

```
values=1×3 cell array
    {[0.1000]}    {[0.2000]}    {[0.3000]}
```

```
probabilities = [1/3 1/3 1/3]
```

```
probabilities = 1×3

    0.3333    0.3333    0.3333
```

```
perturbations(sensor,'RollAccuracy','Selection',values,probabilities)
```

```
ans=7×3 table
          Property                 Type                        Value
    __________________________    ___________    ______________________________________

    "RollAccuracy"                "Selection"    {1x3 cell}    {[0.3333 0.3333 0.3333]}
    "PitchAccuracy"               "None"         {[   NaN]}    {[                 NaN]}
    "YawAccuracy"                 "None"         {[   NaN]}    {[                 NaN]}
    "PositionAccuracy"            "None"         {[   NaN]}    {[                 NaN]}
    "VelocityAccuracy"            "None"         {[   NaN]}    {[                 NaN]}
    "AccelerationAccuracy"        "None"         {[   NaN]}    {[                 NaN]}
    "AngularVelocityAccuracy"     "None"         {[   NaN]}    {[                 NaN]}
```

Perturb the `sensor` object using the perturb function.

```
rng(2020)
perturb(sensor);
sensor
```

```
sensor = 
  insSensor with properties:

           MountingLocation: [0 0 0]            m        
               RollAccuracy: 0.5                deg      
              PitchAccuracy: 0.2                deg      
                YawAccuracy: 1                  deg      
           PositionAccuracy: [1 1 1]            m        
           VelocityAccuracy: 0.05               m/s      
       AccelerationAccuracy: 0                  m/s²     
    AngularVelocityAccuracy: 0                  deg/s    
                  TimeInput: 0                           
               RandomStream: 'Global stream'             
```

The `RollAccuracy` is perturbed to `0.5` deg.

Perturb Waypoint Trajectory

Define a waypoint trajectory. By default, this trajectory contains two waypoints.

```
traj = waypointTrajectory
```

```
traj = 
  waypointTrajectory with properties:

         SampleRate: 100
    SamplesPerFrame: 1
          Waypoints: [2x3 double]
      TimeOfArrival: [2x1 double]
         Velocities: [2x3 double]
             Course: [2x1 double]
        GroundSpeed: [2x1 double]
          ClimbRate: [2x1 double]
        Orientation: [2x1 quaternion]
          AutoPitch: 0
           AutoBank: 0
```

```
     ReferenceFrame: 'NED'
```

Define perturbations on the `Waypoints` property and the `TimeOfArrival` property.

```
rng(2020);
perturbs1 = perturbations(traj,'Waypoints','Normal',1,1)
```

```
perturbs1=2×3 table
       Property         Type            Value
    _______________    ________    __________________

    "Waypoints"        "Normal"    {[  1]}    {[  1]}
    "TimeOfArrival"    "None"      {[NaN]}    {[NaN]}
```

```
perturbs2 = perturbations(traj,'TimeOfArrival','Selection',{[0;1],[0;2]})
```

```
perturbs2=2×3 table
       Property          Type                   Value
    _______________    ___________    _______________________________

    "Waypoints"        "Normal"       {[     1]}    {[            1]}
    "TimeOfArrival"    "Selection"    {1x2 cell}    {[0.5000 0.5000]}
```

Perturb the trajectory.

```
offsets = perturb(traj)
```

```
offsets=2×1 struct array with fields:
    Property
    Offset
    PerturbedValue
```

The `Waypoints` property and the `TimeOfArrival` property have changed.

```
traj.Waypoints
```

```
ans = 2×3

    1.8674    1.0203    0.7032
    2.3154   -0.3207    0.0999
```

```
traj.TimeOfArrival
```

```
ans = 2×1

     0
     2
```

Input Arguments

obj — Object to be perturbed
objects

Object to be perturbed, specified as an object. The objects that you can perturb include:

- `insSensor`
- `waypointTrajectory`

`property` — Perturbable property
property name

Perturbable property, specified as a property name. Use `perturbations` to obtain a full list of perturbable properties for the specified `obj`.

`values` — Perturbation offset values
n-element cell array of property values

Perturbation offset values, specified as an *n*-element cell array of property values. The function randomly draws the perturbation value for the property from the cell array based on the values' corresponding probabilities specified in the `probabilities` input.

`probabilities` — Drawing probabilities for each perturbation value
n-element array of nonnegative scalar

Drawing probabilities for each perturbation value, specified as an *n*-element array of nonnegative scalars, where *n* is the number of perturbation values provided in the `values` input. The sum of all elements must be equal to one.

For example, you can specify a series of perturbation value-probability pair as `{x1,x2,…,xn}` and `{p1,p2,…,pn}`, where the probability of drawing `xi` is `pi` (`i` = 1, 2, …,n).

`mean` — Mean of normal or truncated normal distribution
scalar | vector | matrix

Mean of normal or truncated normal distribution, specified as a scalar, vector, or matrix. The dimension of `mean` must be compatible with the corresponding property that you perturb.

`deviation` — Standard deviation of normal or truncated normal distribution
nonnegative scalar | vector of nonnegative scalar | matrix of nonnegative scalar

Standard deviation of normal or truncated normal distribution, specified as a nonnegative scalar, vector of nonnegative scalars, or matrix of nonnegative scalars. The dimension of `deviation` must be compatible with the corresponding property that you perturb.

`lowerLimit` — Lower limit of truncated normal distribution
scalar | vector | matrix

Lower limit of the truncated normal distribution, specified as a scalar, vector, or matrix. The dimension of `lowerLimit` must be compatible with the corresponding property that you perturb.

`upperLimit` — Upper limit of truncated normal distribution
scalar | vector | matrix

Upper limit of the truncated normal distribution, specified as a scalar, vector, or matrix. The dimension of `upperLimit` must be compatible with the corresponding property that you perturb.

`minVal` — Minimum value of uniform distribution interval
scalar | vector | matrix

Minimum value of the uniform distribution interval, specified as a scalar, vector, or matrix. The dimension of `minVal` must be compatible with the corresponding property that you perturb.

`maxVal` — Maximum value of uniform distribution interval
scalar | vector | matrix

Maximum value of the uniform distribution interval, specified as a scalar, vector, or matrix. The dimension of `maxVal` must be compatible with the corresponding property that you perturb.

`perturbFcn` — Perturbation function
function handle

Perturbation function, specified as a function handle. The function must have this syntax:

```
offset = myfun(propVal)
```

where `propVal` is the value of the `property` and `offset` is the perturbation offset for the property.

Output Arguments

`perturbs` — Perturbations defined on object
table of perturbation property

Perturbations defined on the object, returned as a table of perturbation properties. The table has three columns:

- `Property` — Property names.
- `Type` — Type of perturbations, returned as `"None"`, `"Selection"`, `"Normal"`, `"TruncatedNormal"`, `"Uniform"`, or `"Custom"`.
- `Value` — Perturbation values, returned as a cell array.

More About

Specify Perturbation Distributions

You can specify the distribution for the perturbation applied to a specific property.

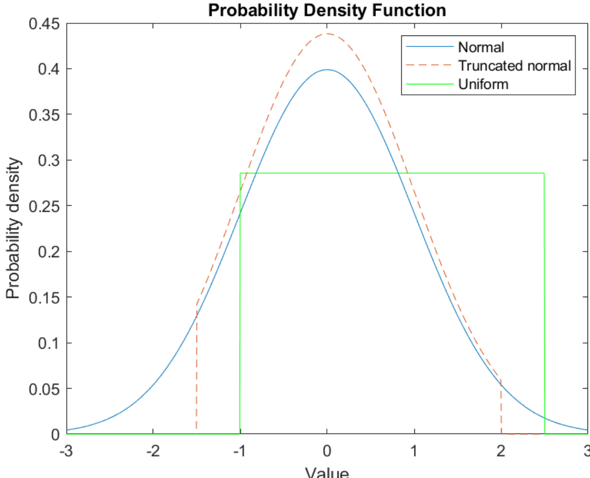
- Selection distribution — The function defines the perturbation offset as one of the specified values with the associated probability. For example, if you specify the values as `[1 2]` and specify the probabilities as `[0.7 0.3]`, then the `perturb` function adds an offset value of `1` to the property with a probability of `0.7` and add an offset value of `2` to the property with a probability of `0.3`. Use selection distribution when you only want to perturb the property with a number of discrete values.
- Normal distribution — The function defines the perturbation offset as a value drawn from a normal distribution with the specified mean and standard deviation (or covariance). Normal distribution is the most commonly used distribution since it mimics the natural perturbation of parameters in most cases.
- Truncated normal distribution — The function defines the perturbation offset as a value drawn from a truncated normal distribution with the specified mean, standard deviation (or covariance), lower limit, and upper limit. Different from the normal distribution, the values drawn from a truncated normal distribution are truncated by the lower and upper limit. Use truncated normal distribution when you want to apply a normal distribution, but the valid values of the property are confined in an interval.

- Uniform distribution — The function defines the perturbation offset as a value drawn from a uniform distribution with the specified minimum and maximum values. All the values in the interval (specified by the minimum and maximum values) have the same probability of realization.
- Custom distribution — Customize your own perturbation function. The function must have this syntax:

  ```
  offset = myfun(propVal)
  ```

  where `propVal` is the value of the `property` and `offset` is the perturbation offset for the property.

This figure shows probability density functions for a normal distribution, a truncated normal distribution, and a uniform distribution, respectively.



Version History

Introduced in R2022a

See Also

`perturb`

addConfiguration

Store current configuration

Syntax

```
addConfiguration(viztree)
addConfiguration(viztree,index)
```

Description

`addConfiguration(viztree)` adds the current configuration to the `StoredConfigurations` property of the `interactiveRigidBodyTree` object, `viztree`.

`addConfiguration(viztree,index)` inserts the current configuration into the `StoredConfigurations` property at the specified index. The stored configurations after the specified index shift down by one.

Examples

Interactively Build and Play Back Series of Robot Configurations

Use the `interactiveRigidBodyTree` object to manually move around a robot in a figure. The object uses interactive markers in the figure to track the desired poses of different rigid bodies in the `rigidBodyTree` object.

Load Robot Model

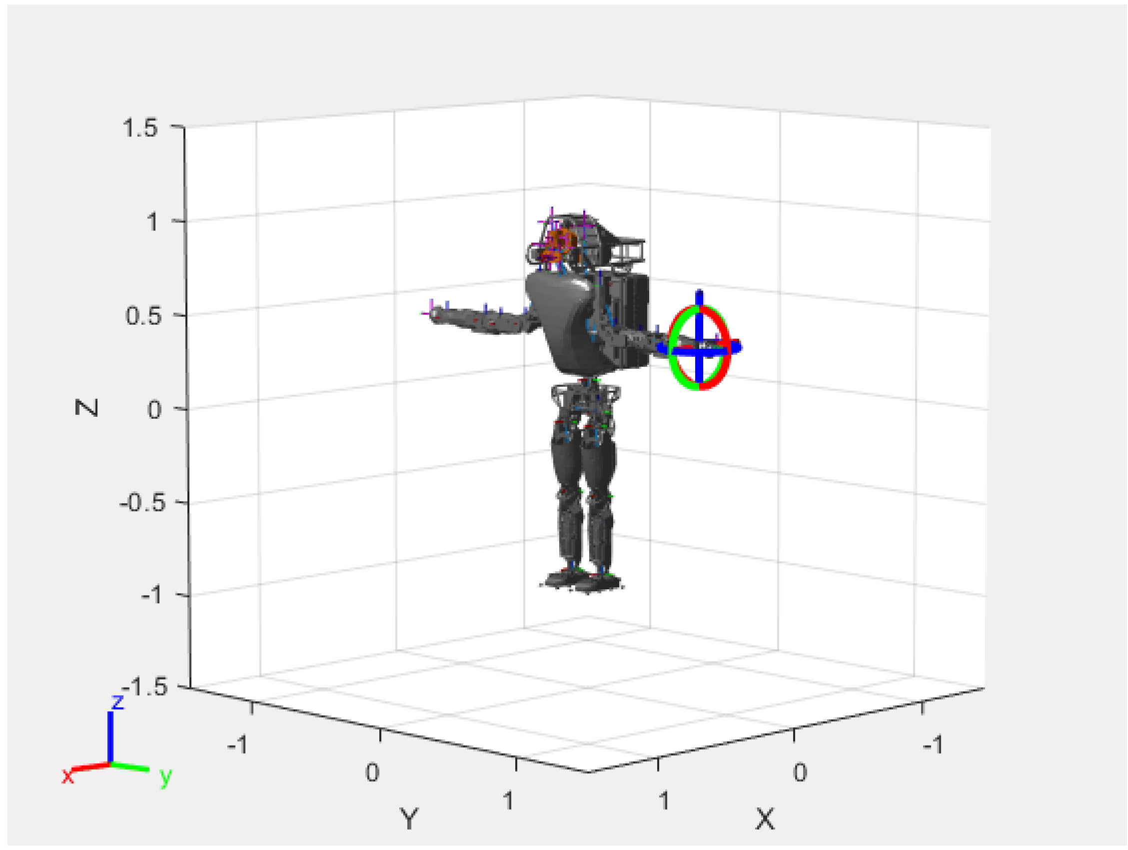
Use the `loadrobot` function to access provided robot models as `rigidBodyTree` objects.

```
robot = loadrobot("atlas");
```

Visualize Robot and Save Configurations

Create an interactive tree object and associated figure, specifying the loaded robot model and its left hand as the end effector.

```
viztree = interactiveRigidBodyTree(robot,"MarkerBodyName","l_hand");
```

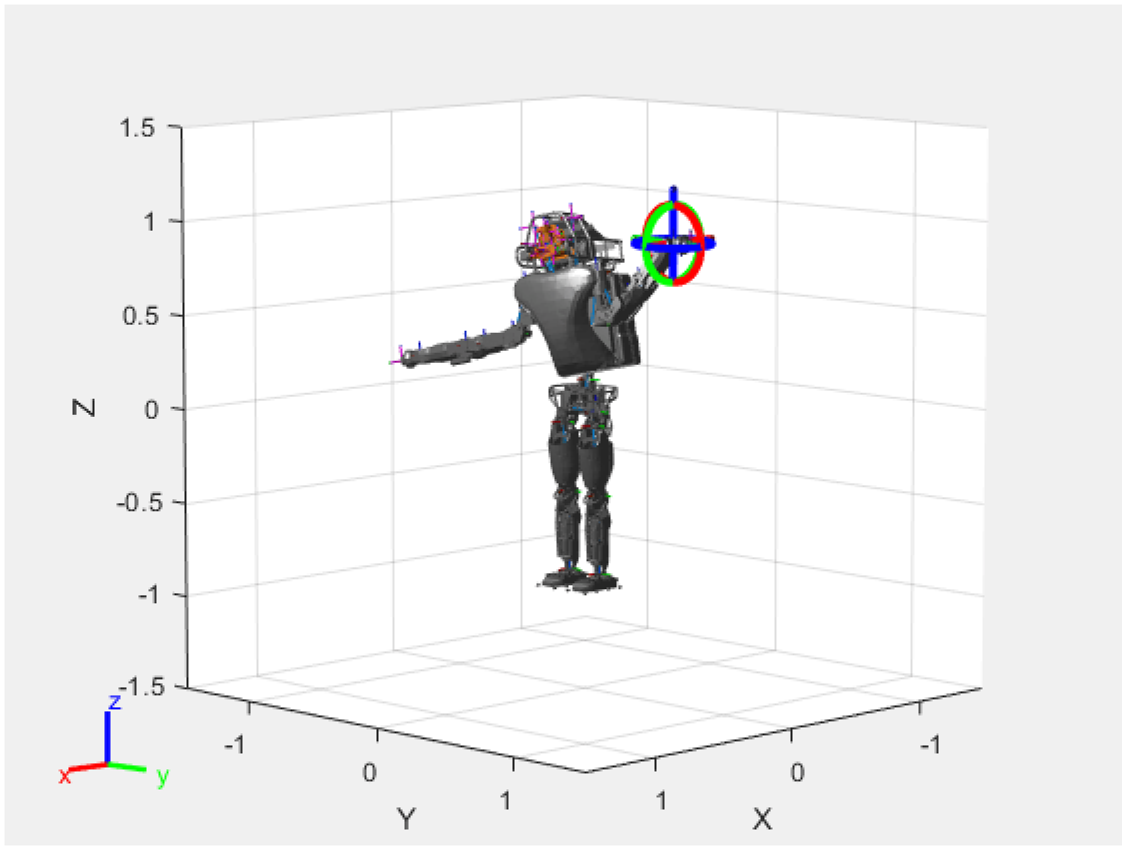


Click and drag the interactive marker to change the robot configuration. You can click and drag any of the axes for linear motion, rotate the body about an axis using the red, green, and blue circles, and drag the center of the interactive marker to position it in 3-D space.

The `interactiveRigidBodyTree` object uses inverse kinematics to determine a configuration that achieves the desired end-effector pose. If the associated rigid body cannot reach the marker, the figure renders the best configuration from the inverse kinematics solver.

Programmatically set the current configuration. Assign a vector of length equal to the number of nonfixed joints in the `RigidBodyTree` to the `Configuration` property.

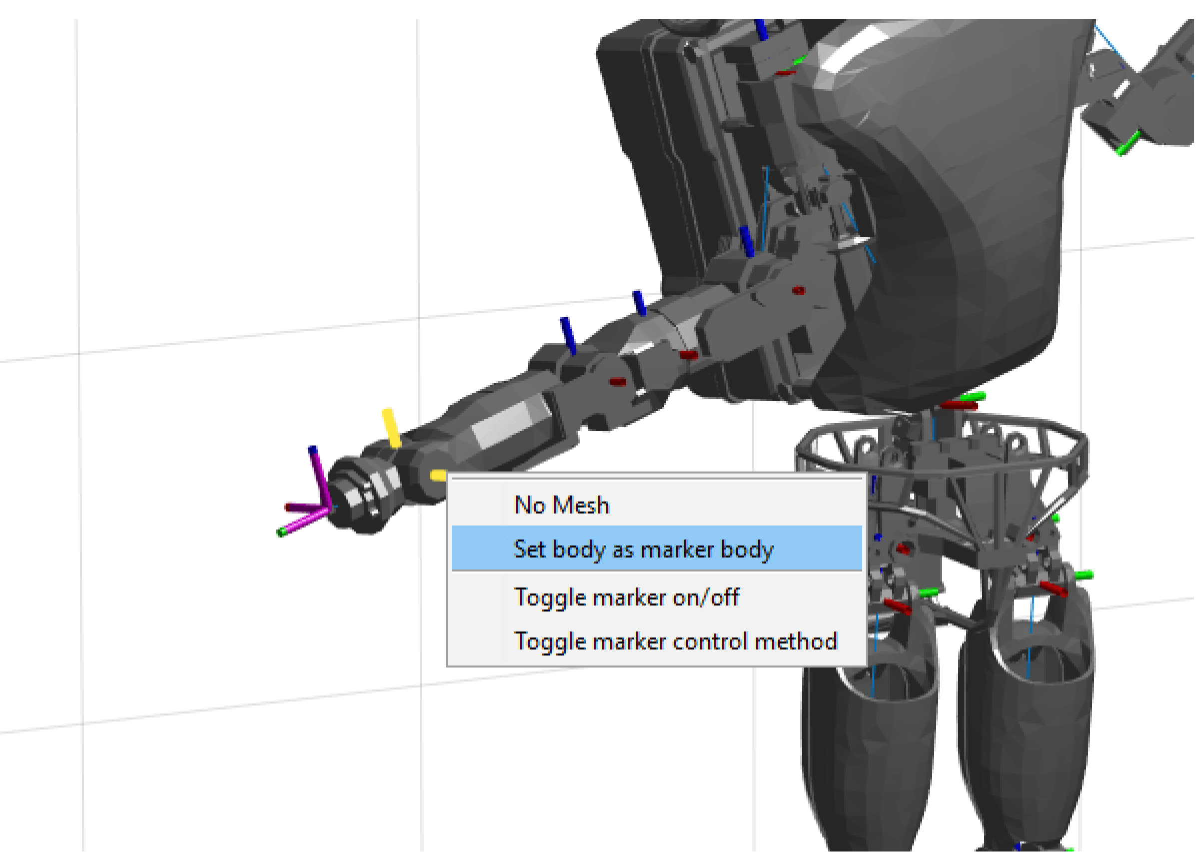
```
currConfig = homeConfiguration(viztree.RigidBodyTree);
currConfig(1:10) = [ 0.2201 -0.1319 0.2278 -0.3415 0.4996 ...
                     0.0747 0.0377 0.0718 -0.8117 -0.0427]';
viztree.Configuration = currConfig;
```



Save the current robot configuration in the `StoredConfigurations` property.

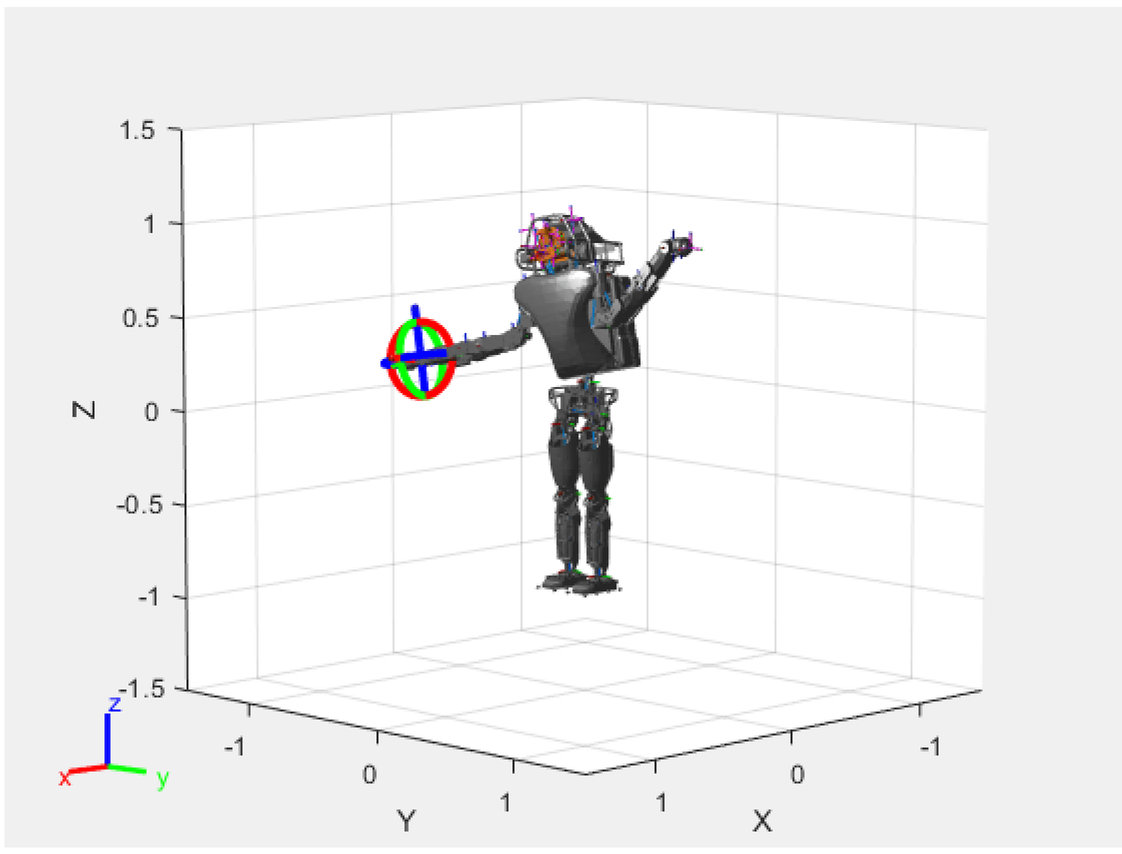
```
addConfiguration(viztree)
```

To switch the end effector to a different rigid body, right-click the desired body in the figure and select **Set body as marker body**. Use this process to select the right hand frame.



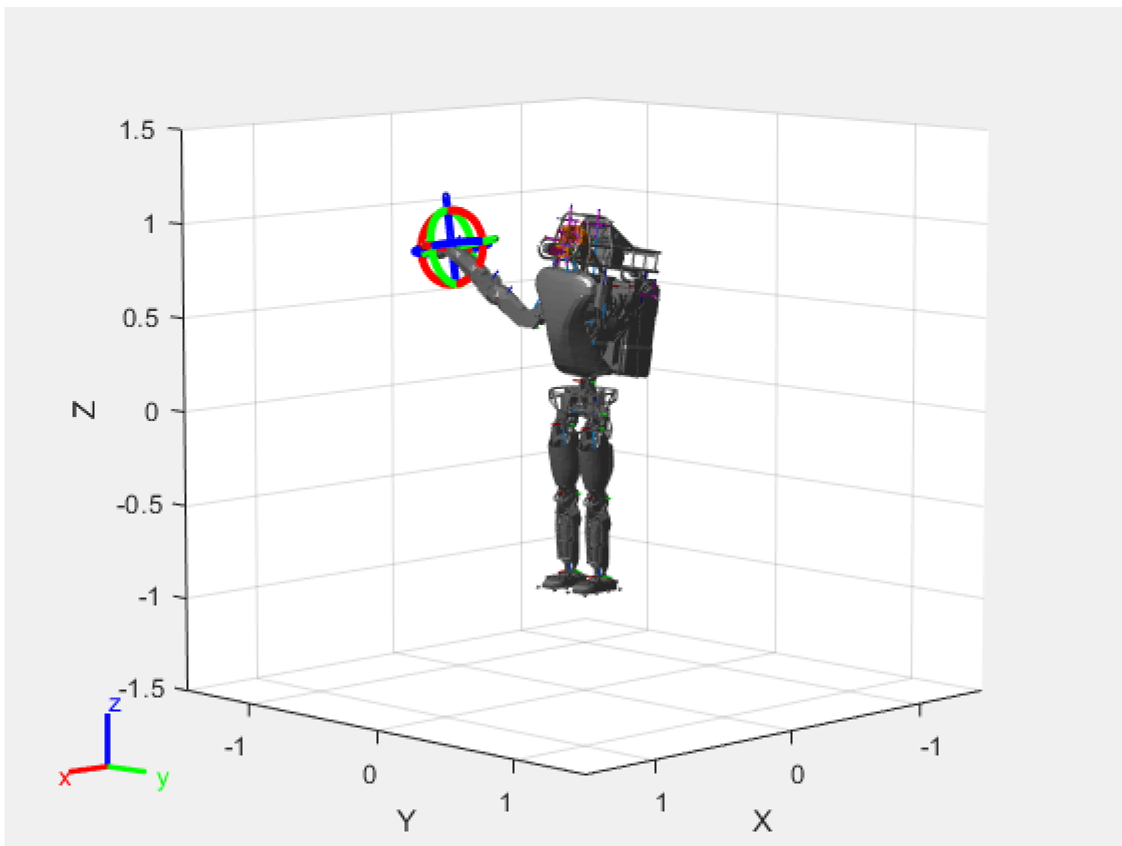
You can also set the `MarkerBodyName` property to the specific body name.

```
viztree.MarkerBodyName = "r_hand";
```



Move the right hand to a new position. Set the configuration programmatically. The marker moves to the new position of the end effector.

```
currConfig(1:18) = [-0.1350 -0.1498 -0.0167 -0.3415 0.4996 0.0747
                     0.0377 0.0718 -0.8117 -0.0427 0 0.4349
                    -0.5738 0.0563 -0.0095 0.0518 0.8762 -0.0895]';

viztree.Configuration = currConfig;
```



Save the current configuration.

```
addConfiguration(viztree)
```

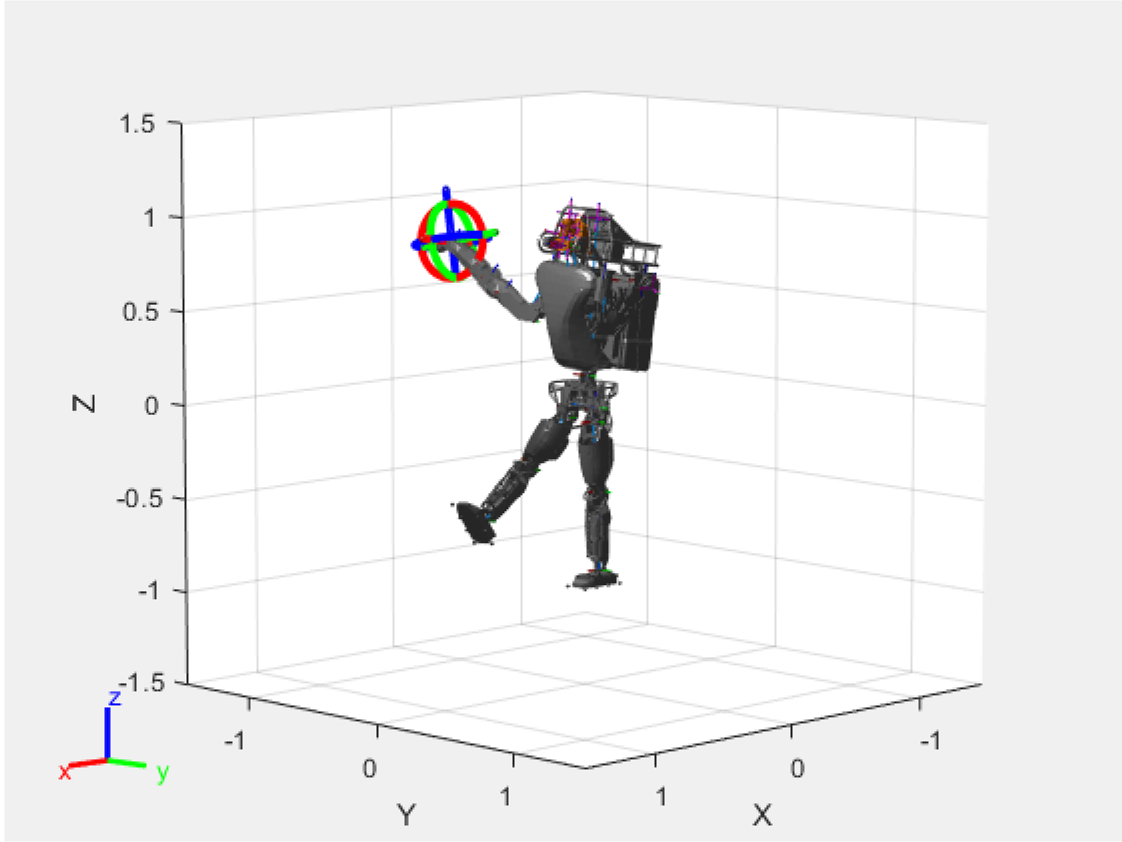
Add Constraints

By default, the robot model respects only the joint limits of the `rigidBodyJoint` objects associated with the `RigidBodyTree` property. To add constraints, generate **Robot Constraint** objects and specify them as a cell array in the `Constraints` property. To see a list of robotic constraints, see "Inverse Kinematics". Specify a pose target for the pelvis to keep it fixed to the home position. Specify a position target for the right foot to be raised in front front and above its current position.

```
fixedWaist = constraintPoseTarget("pelvis");
raiseRightLeg = constraintPositionTarget("r_foot","TargetPosition",[1 0 0.5]);
```

Apply these constraints to the interactive rigid body tree object as a cell array. The right leg in the resulting figure changes position.

```
viztree.Constraints = {fixedWaist raiseRightLeg};
```



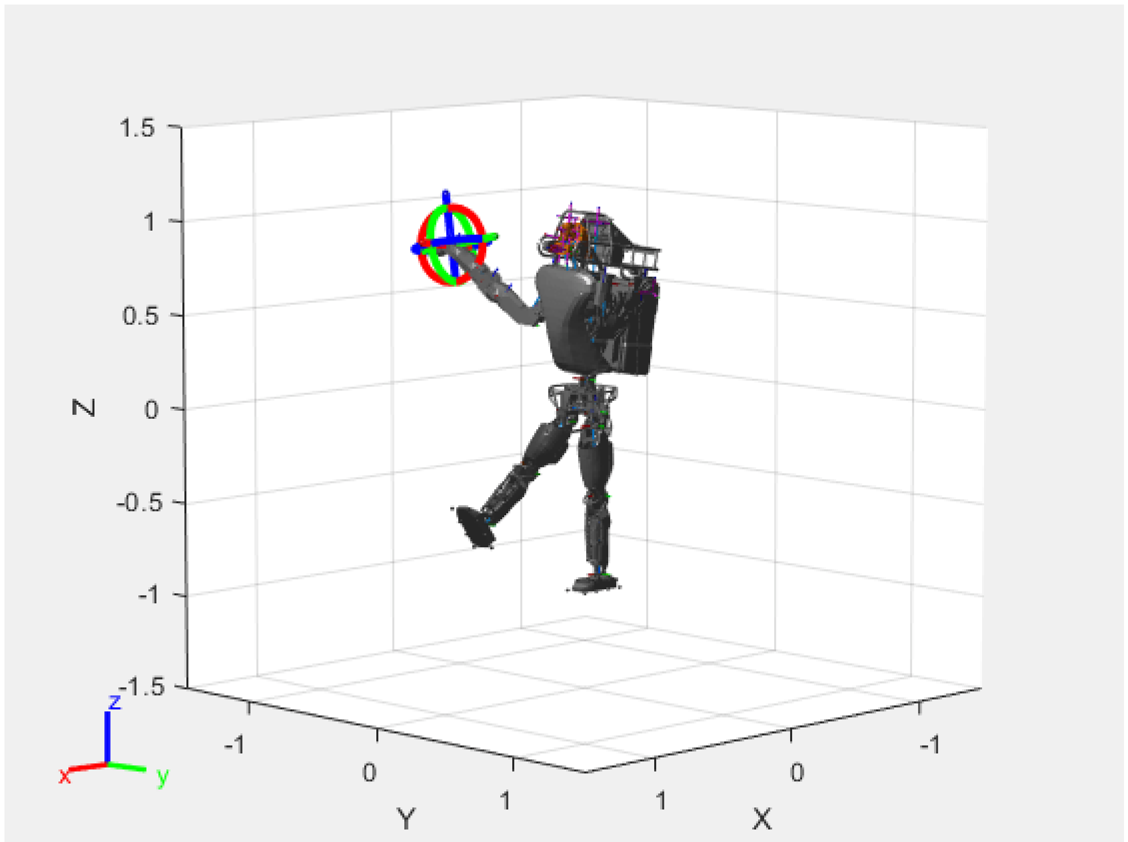
Notice the change in position of the right leg. Save this configuration as well.

```
addConfiguration(viztree)
```

Play Back Configurations

To play back configurations, iterate through the stored configurations index and set the current configuration equal to the stored configuration column vector at each iteration. Because configurations are stored as column vectors, use the second dimension of the matrix.

```
for i = 1:size(viztree.StoredConfigurations,2)
    viztree.Configuration = viztree.StoredConfigurations(:,i);
    pause(0.5)
end
```



Input Arguments

viztree — Interactive rigid body tree robot model visualization
`interactiveRigidBodyTree` object

Interactive rigid body tree robot model visualization, specified as an `interactiveRigidBodyTree` object.

index — Index location to store current configuration
positive integer

Index location to store current configuration, specified as a positive integer. The stored configurations after the specified index shift down by one.

Data Types: `double`

Version History
Introduced in R2020a

See Also

Functions
`removeConfigurations` | `loadrobot` | `importrobot` | `homeConfiguration`

Objects
`interactiveRigidBodyTree` | `rigidBodyTree` | `rigidBody` | `rigidBodyJoint` | `generalizedInverseKinematics`

Topics
"Rigid Body Tree Robot Model"
"Plan a Reaching Trajectory With Multiple Kinematic Constraints"
"Trajectory Control Modeling with Inverse Kinematics"

addConstraint

Add inverse kinematics constraint

Syntax

```
addConstraint(viztree,gikConstraint)
```

Description

`addConstraint(viztree,gikConstraint)` adds an inverse kinematics constraint, `gikConstaint`, to the `Constraints` property of the `interactiveRigidBodyTree` object, `viztree`.

Examples

Interactively Build and Play Back Series of Robot Configurations

Use the `interactiveRigidBodyTree` object to manually move around a robot in a figure. The object uses interactive markers in the figure to track the desired poses of different rigid bodies in the `rigidBodyTree` object.

Load Robot Model

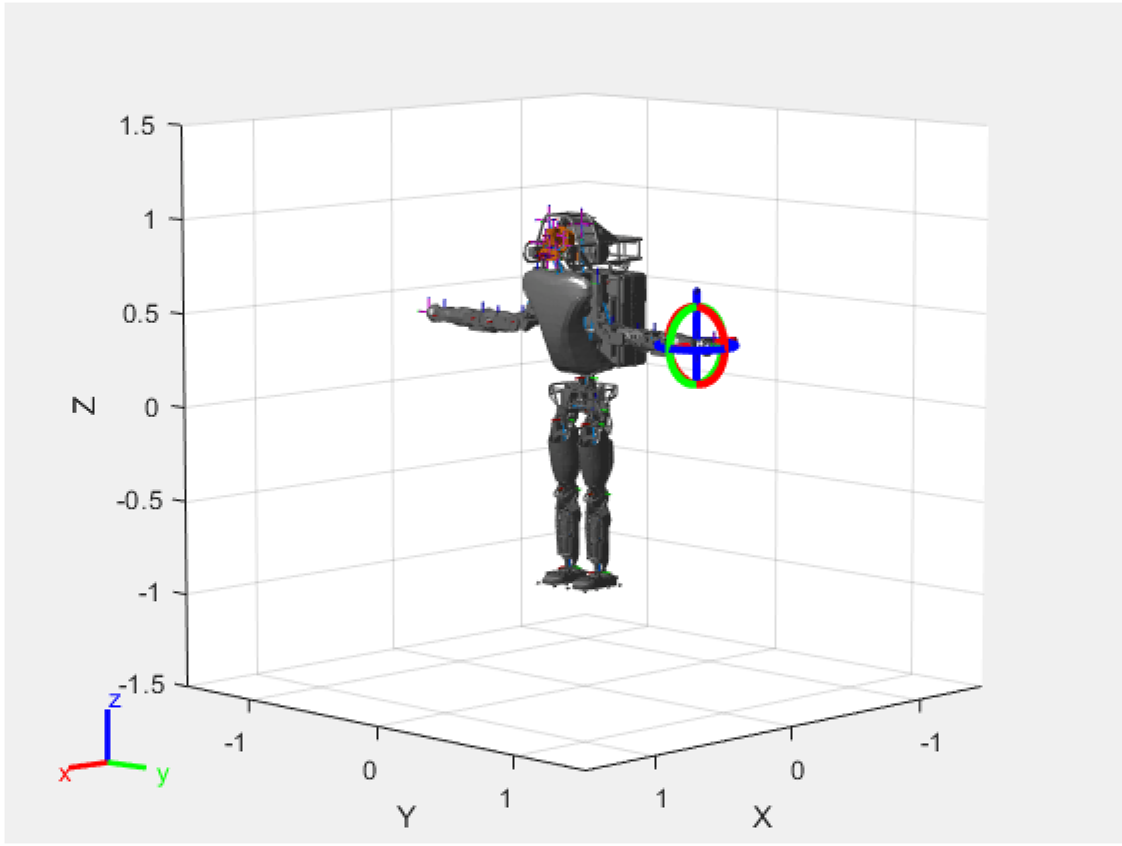
Use the `loadrobot` function to access provided robot models as `rigidBodyTree` objects.

```
robot = loadrobot("atlas");
```

Visualize Robot and Save Configurations

Create an interactive tree object and associated figure, specifying the loaded robot model and its left hand as the end effector.

```
viztree = interactiveRigidBodyTree(robot,"MarkerBodyName","l_hand");
```

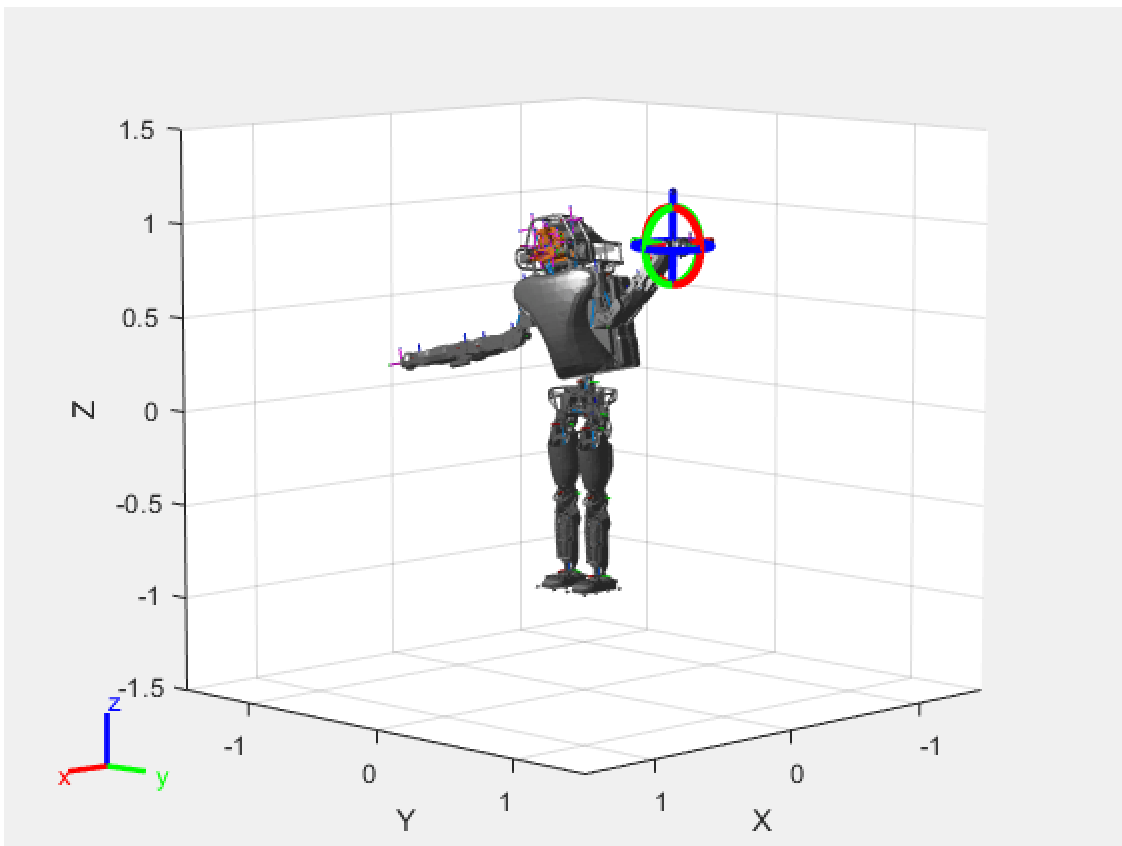


Click and drag the interactive marker to change the robot configuration. You can click and drag any of the axes for linear motion, rotate the body about an axis using the red, green, and blue circles, and drag the center of the interactive marker to position it in 3-D space.

The `interactiveRigidBodyTree` object uses inverse kinematics to determine a configuration that achieves the desired end-effector pose. If the associated rigid body cannot reach the marker, the figure renders the best configuration from the inverse kinematics solver.

Programmatically set the current configuration. Assign a vector of length equal to the number of nonfixed joints in the `RigidBodyTree` to the `Configuration` property.

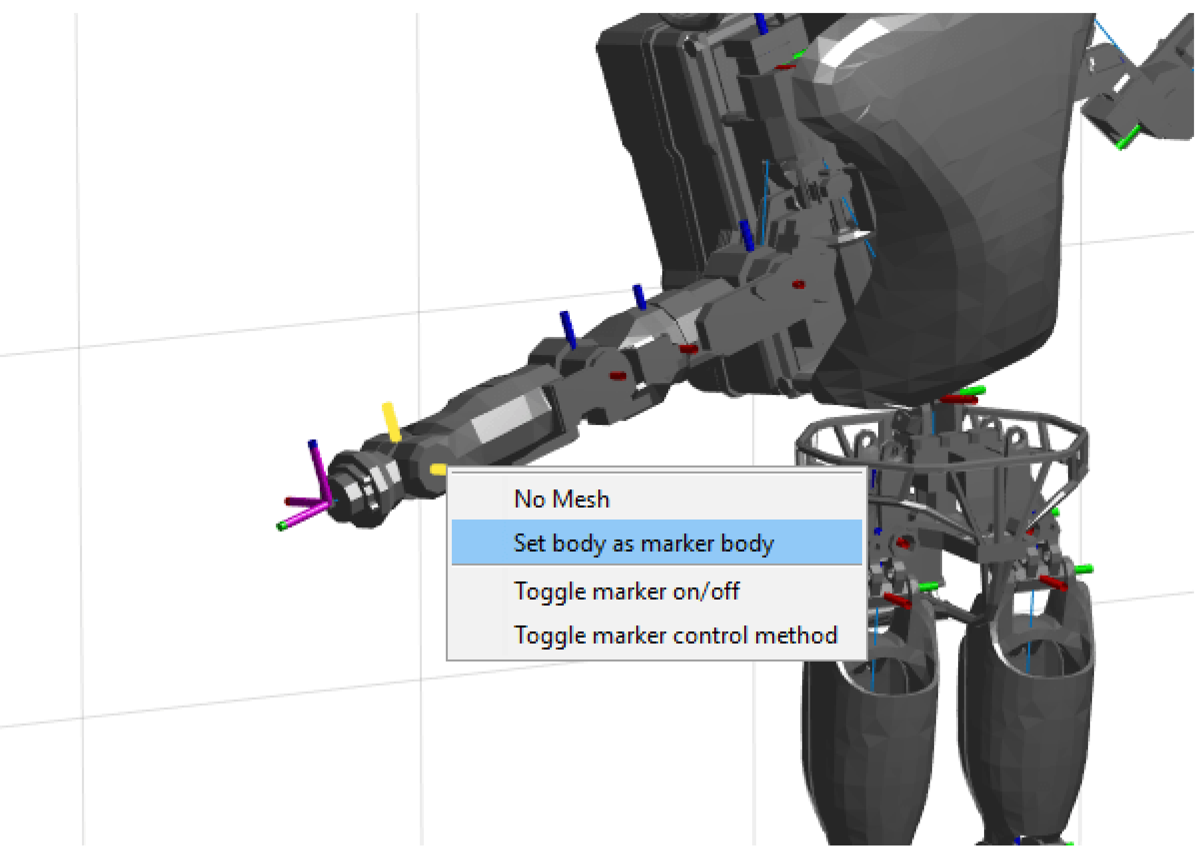
```
currConfig = homeConfiguration(viztree.RigidBodyTree);
currConfig(1:10) = [ 0.2201 -0.1319 0.2278 -0.3415 0.4996 ...
                     0.0747 0.0377 0.0718 -0.8117 -0.0427]';
viztree.Configuration = currConfig;
```



Save the current robot configuration in the `StoredConfigurations` property.

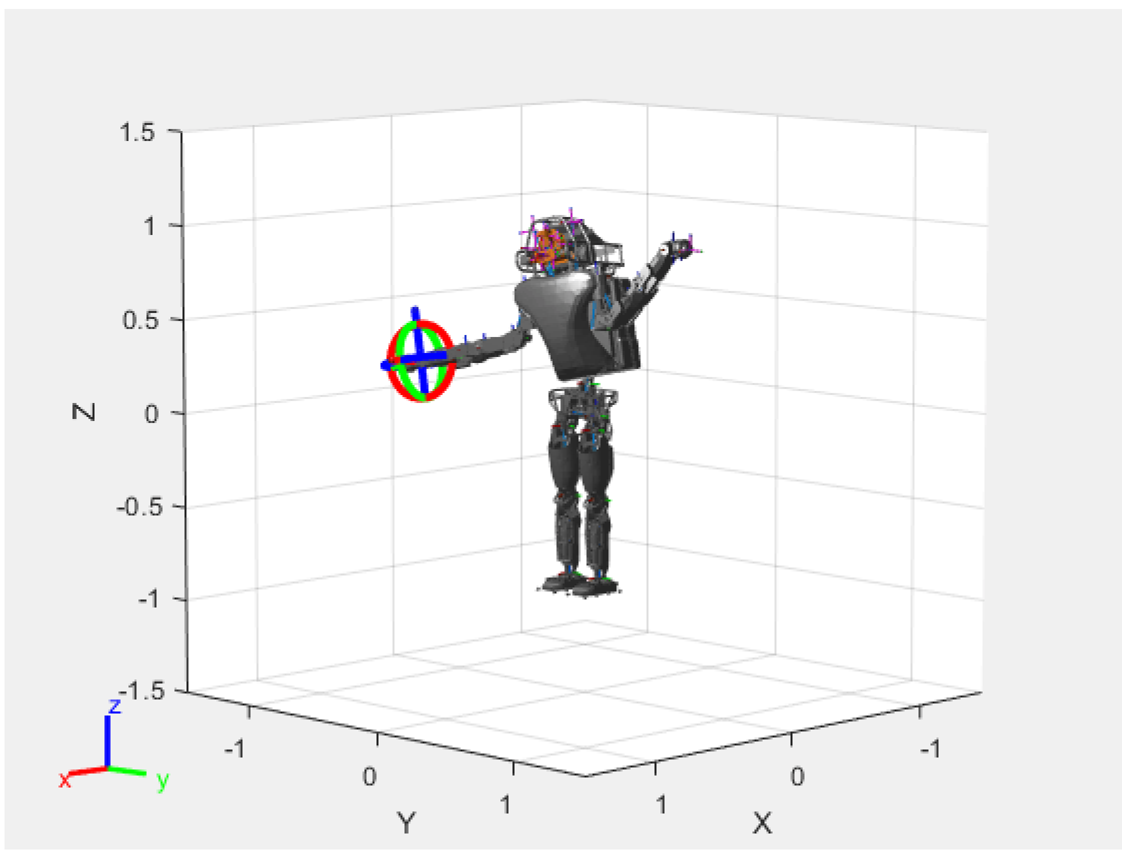
```
addConfiguration(viztree)
```

To switch the end effector to a different rigid body, right-click the desired body in the figure and select **Set body as marker body**. Use this process to select the right hand frame.



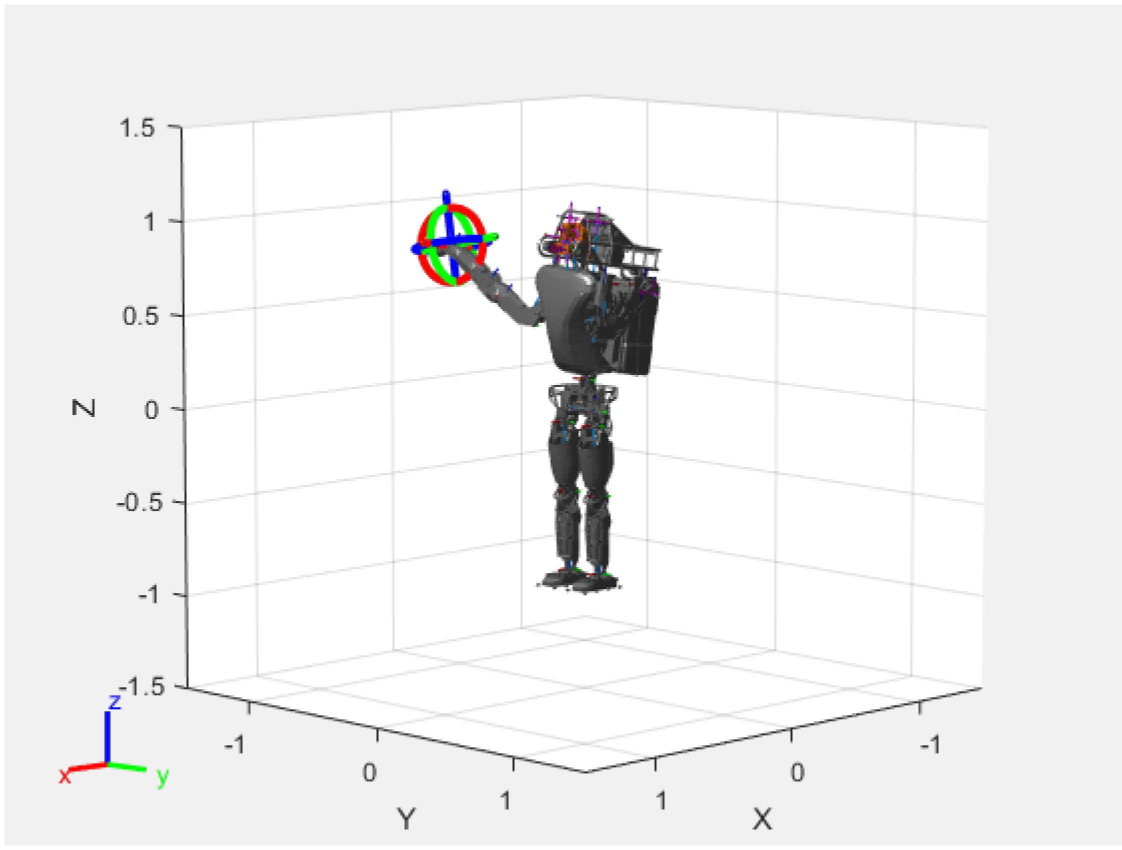
You can also set the `MarkerBodyName` property to the specific body name.

```
viztree.MarkerBodyName = "r_hand";
```



Move the right hand to a new position. Set the configuration programmatically. The marker moves to the new position of the end effector.

```
currConfig(1:18) = [-0.1350 -0.1498 -0.0167 -0.3415 0.4996 0.0747
                     0.0377 0.0718 -0.8117 -0.0427 0 0.4349
                    -0.5738 0.0563 -0.0095 0.0518 0.8762 -0.0895]';

viztree.Configuration = currConfig;
```



Save the current configuration.

```
addConfiguration(viztree)
```

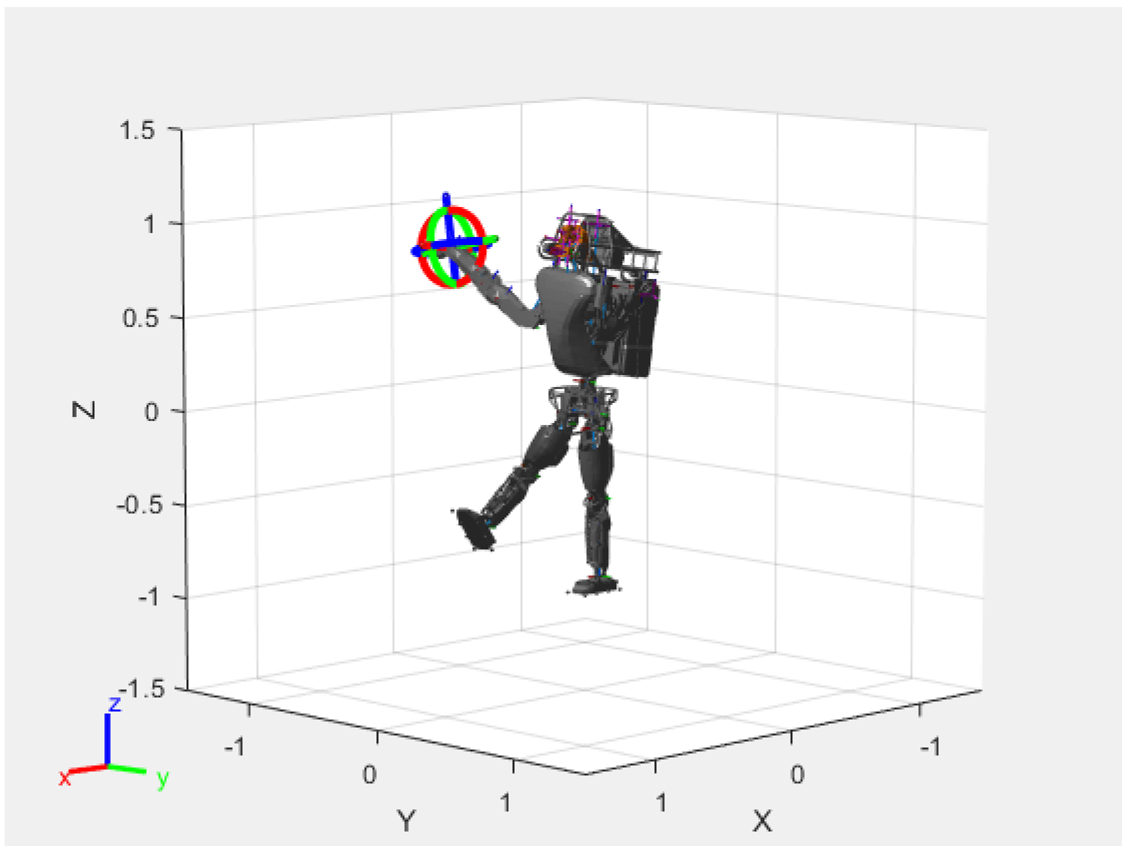
Add Constraints

By default, the robot model respects only the joint limits of the `rigidBodyJoint` objects associated with the `RigidBodyTree` property. To add constraints, generate **Robot Constraint** objects and specify them as a cell array in the `Constraints` property. To see a list of robotic constraints, see "Inverse Kinematics". Specify a pose target for the pelvis to keep it fixed to the home position. Specify a position target for the right foot to be raised in front front and above its current position.

```
fixedWaist = constraintPoseTarget("pelvis");
raiseRightLeg = constraintPositionTarget("r_foot","TargetPosition",[1 0 0.5]);
```

Apply these constraints to the interactive rigid body tree object as a cell array. The right leg in the resulting figure changes position.

```
viztree.Constraints = {fixedWaist raiseRightLeg};
```



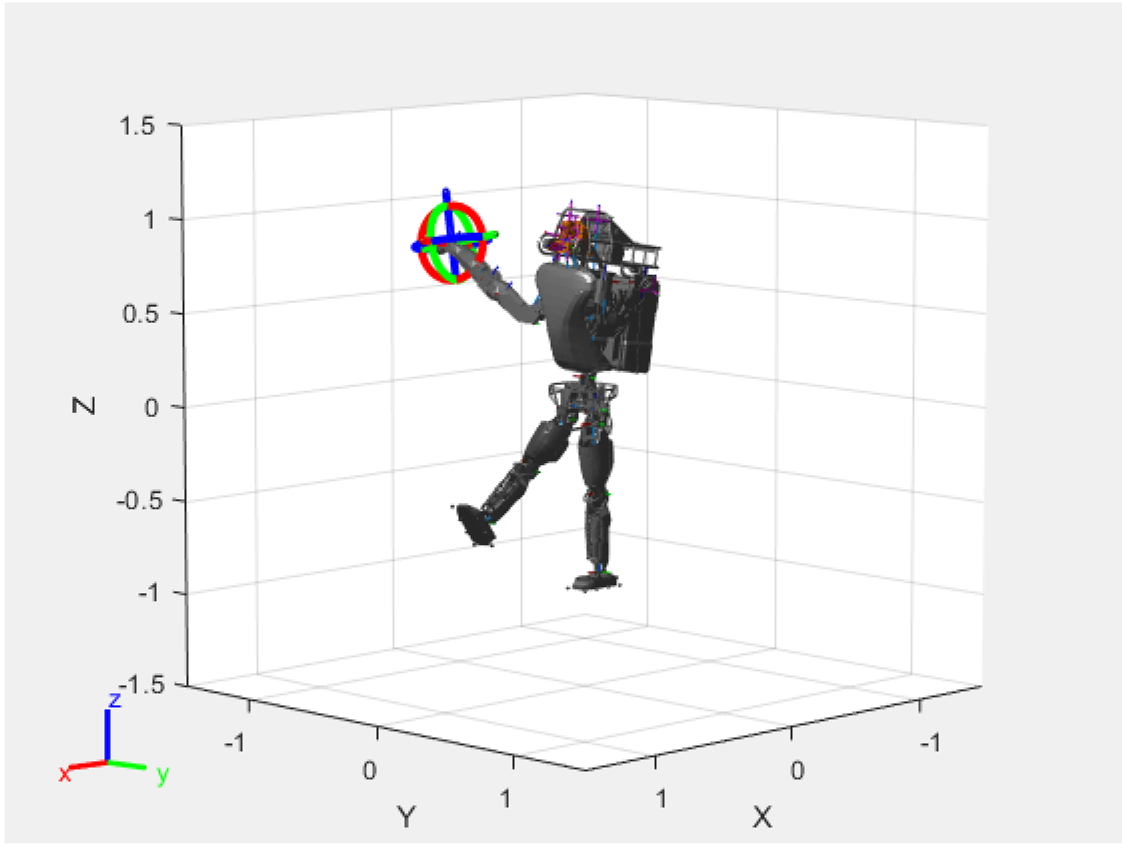
Notice the change in position of the right leg. Save this configuration as well.

```
addConfiguration(viztree)
```

Play Back Configurations

To play back configurations, iterate through the stored configurations index and set the current configuration equal to the stored configuration column vector at each iteration. Because configurations are stored as column vectors, use the second dimension of the matrix.

```
for i = 1:size(viztree.StoredConfigurations,2)
    viztree.Configuration = viztree.StoredConfigurations(:,i);
    pause(0.5)
end
```



Input Arguments

viztree — Interactive rigid body tree robot model visualization
interactiveRigidBodyTree object

Interactive rigid body tree robot model visualization, specified as an interactiveRigidBodyTree object.

gikConstraint — Generalized inverse kinematics constraint
constraint object

Generalized inverse kinematics constraint, specified as one of these constraint objects.

- constraintAiming
- constraintCartesianBounds
- constraintJointBounds
- constraintOrientationTarget
- constraintPoseTarget
- constraintPositionTarget

Version History

Introduced in R2020a

See Also

Functions
removeConstraints | loadrobot | importrobot | homeConfiguration

Objects
interactiveRigidBodyTree | rigidBodyTree | rigidBody | rigidBodyJoint | generalizedInverseKinematics

Topics
"Rigid Body Tree Robot Model"
"Plan a Reaching Trajectory With Multiple Kinematic Constraints"
"Trajectory Control Modeling with Inverse Kinematics"

removeConfigurations

Remove configurations from `StoredConfigurations` property

Syntax

```
removeConfigurations(viztree)
removeConfigurations(viztree,index)
```

Description

`removeConfigurations(viztree)` removes the configuration stored at the last index of the `StoredConfigurations` property of the `interactiveRigidBodyTree` object, `viztree`.

`removeConfigurations(viztree,index)` removes the configurations with the specified indices.

Examples

Interactively Build and Play Back Series of Robot Configurations

Use the `interactiveRigidBodyTree` object to manually move around a robot in a figure. The object uses interactive markers in the figure to track the desired poses of different rigid bodies in the `rigidBodyTree` object.

Load Robot Model

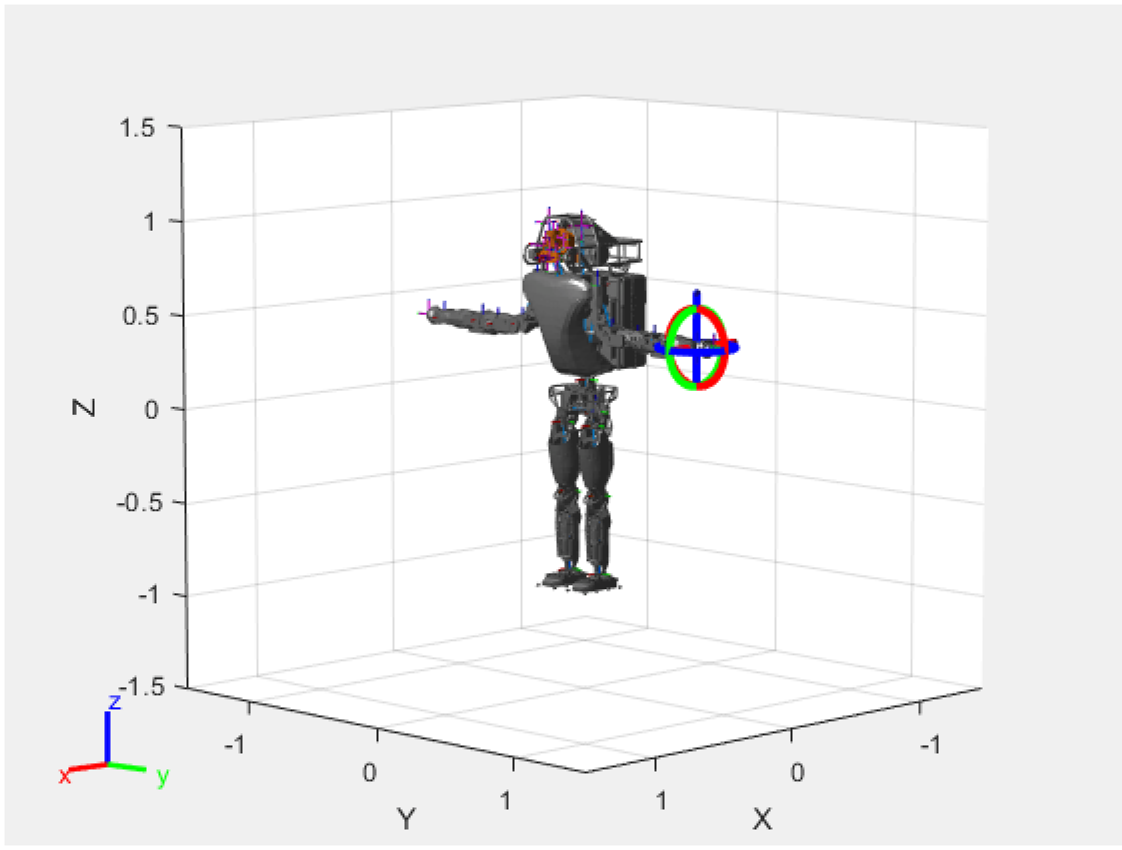
Use the `loadrobot` function to access provided robot models as `rigidBodyTree` objects.

```
robot = loadrobot("atlas");
```

Visualize Robot and Save Configurations

Create an interactive tree object and associated figure, specifying the loaded robot model and its left hand as the end effector.

```
viztree = interactiveRigidBodyTree(robot,"MarkerBodyName","l_hand");
```

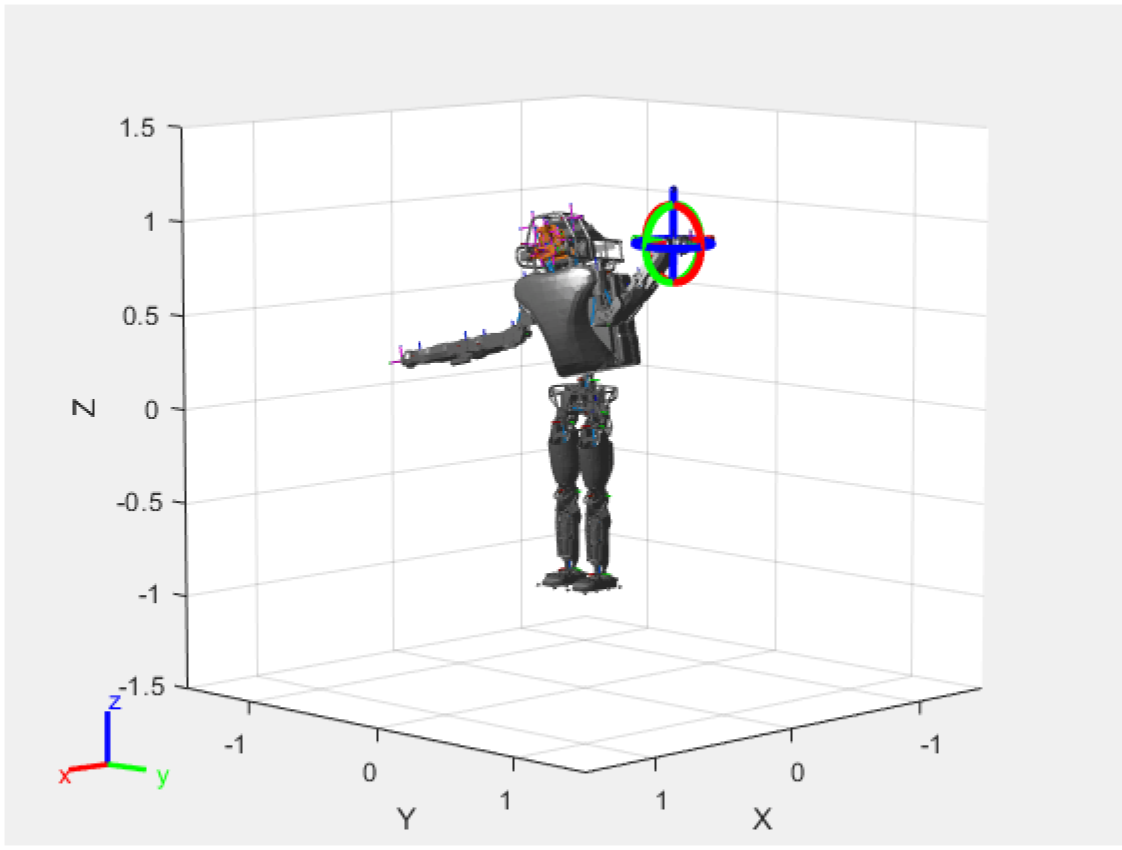


Click and drag the interactive marker to change the robot configuration. You can click and drag any of the axes for linear motion, rotate the body about an axis using the red, green, and blue circles, and drag the center of the interactive marker to position it in 3-D space.

The `interactiveRigidBodyTree` object uses inverse kinematics to determine a configuration that achieves the desired end-effector pose. If the associated rigid body cannot reach the marker, the figure renders the best configuration from the inverse kinematics solver.

Programmatically set the current configuration. Assign a vector of length equal to the number of nonfixed joints in the `RigidBodyTree` to the `Configuration` property.

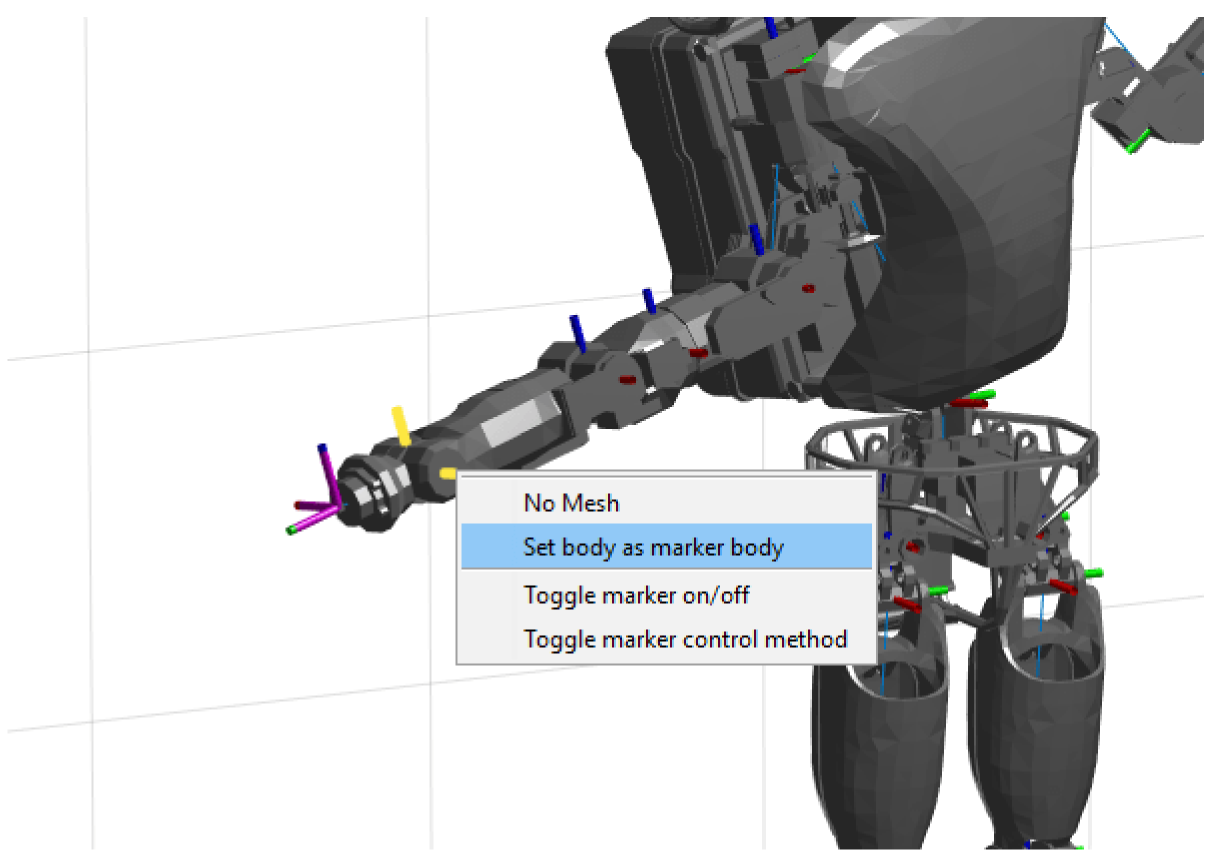
```
currConfig = homeConfiguration(viztree.RigidBodyTree);
currConfig(1:10) = [ 0.2201 -0.1319 0.2278 -0.3415 0.4996 ...
                     0.0747 0.0377 0.0718 -0.8117 -0.0427]';
viztree.Configuration = currConfig;
```



Save the current robot configuration in the `StoredConfigurations` property.

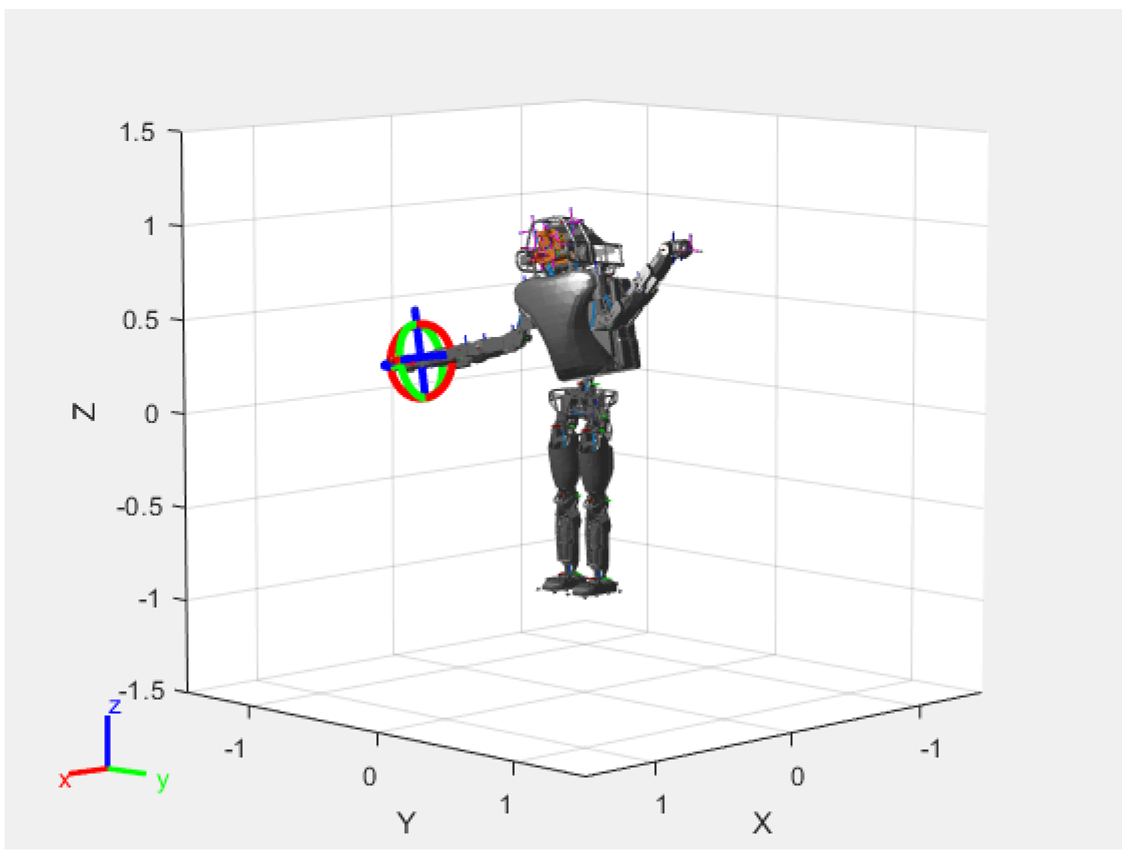
```
addConfiguration(viztree)
```

To switch the end effector to a different rigid body, right-click the desired body in the figure and select **Set body as marker body**. Use this process to select the right hand frame.



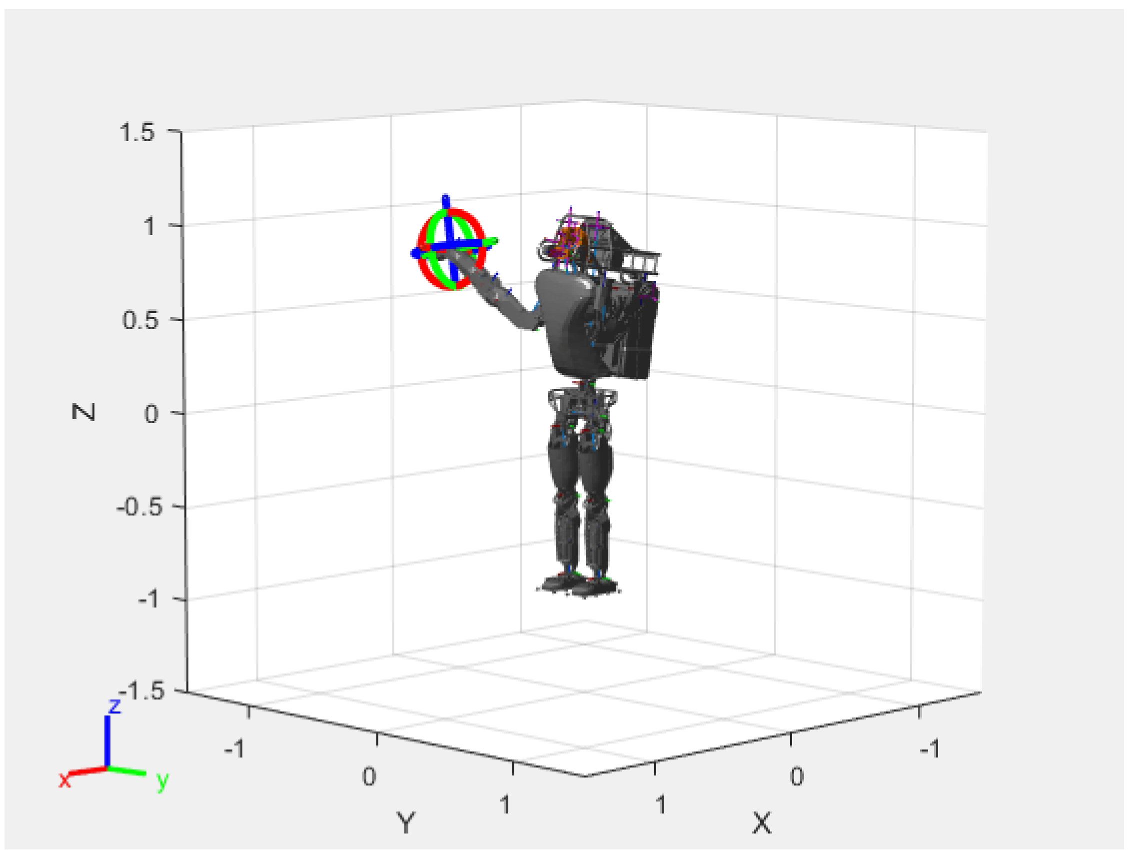
You can also set the MarkerBodyName property to the specific body name.

```
viztree.MarkerBodyName = "r_hand";
```



Move the right hand to a new position. Set the configuration programmatically. The marker moves to the new position of the end effector.

```
currConfig(1:18) = [-0.1350 -0.1498 -0.0167 -0.3415 0.4996 0.0747
                     0.0377 0.0718 -0.8117 -0.0427 0 0.4349
                    -0.5738 0.0563 -0.0095 0.0518 0.8762 -0.0895]';

viztree.Configuration = currConfig;
```



Save the current configuration.

```
addConfiguration(viztree)
```

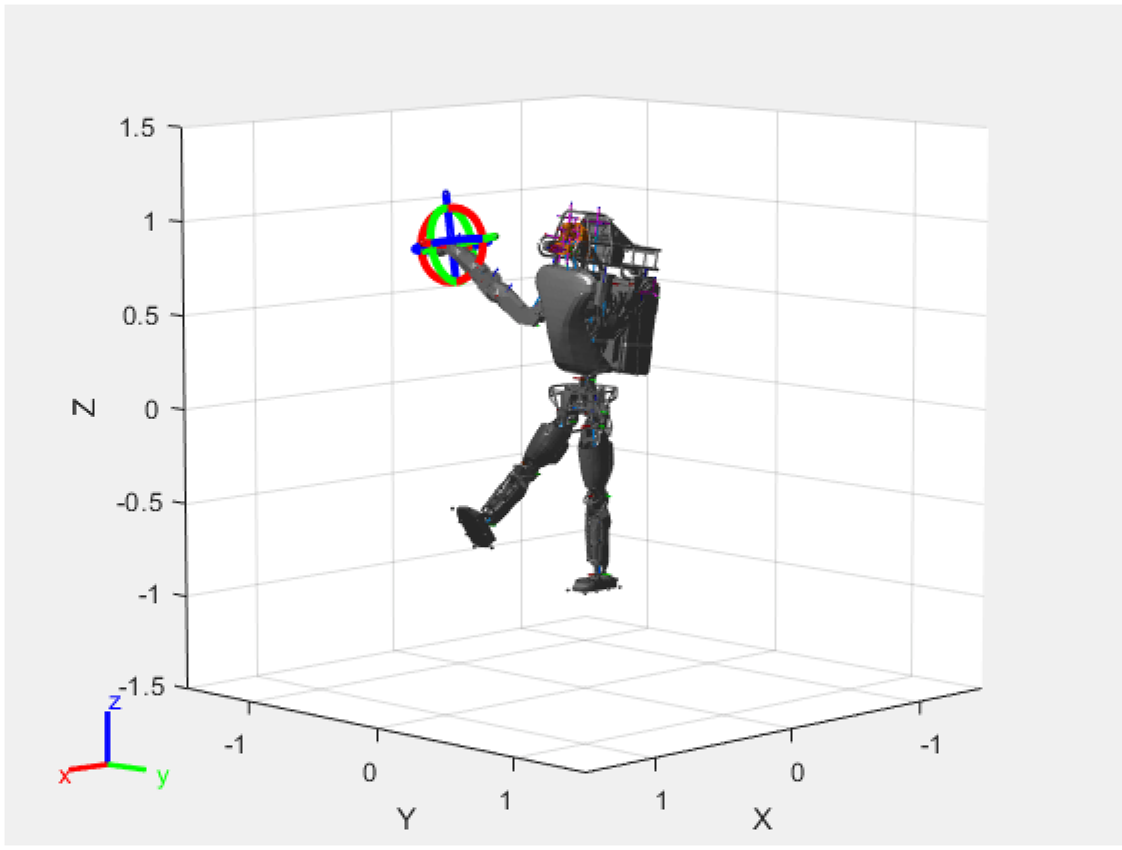
Add Constraints

By default, the robot model respects only the joint limits of the `rigidBodyJoint` objects associated with the `RigidBodyTree` property. To add constraints, generate **Robot Constraint** objects and specify them as a cell array in the `Constraints` property. To see a list of robotic constraints, see "Inverse Kinematics". Specify a pose target for the pelvis to keep it fixed to the home position. Specify a position target for the right foot to be raised in front front and above its current position.

```
fixedWaist = constraintPoseTarget("pelvis");
raiseRightLeg = constraintPositionTarget("r_foot","TargetPosition",[1 0 0.5]);
```

Apply these constraints to the interactive rigid body tree object as a cell array. The right leg in the resulting figure changes position.

```
viztree.Constraints = {fixedWaist raiseRightLeg};
```



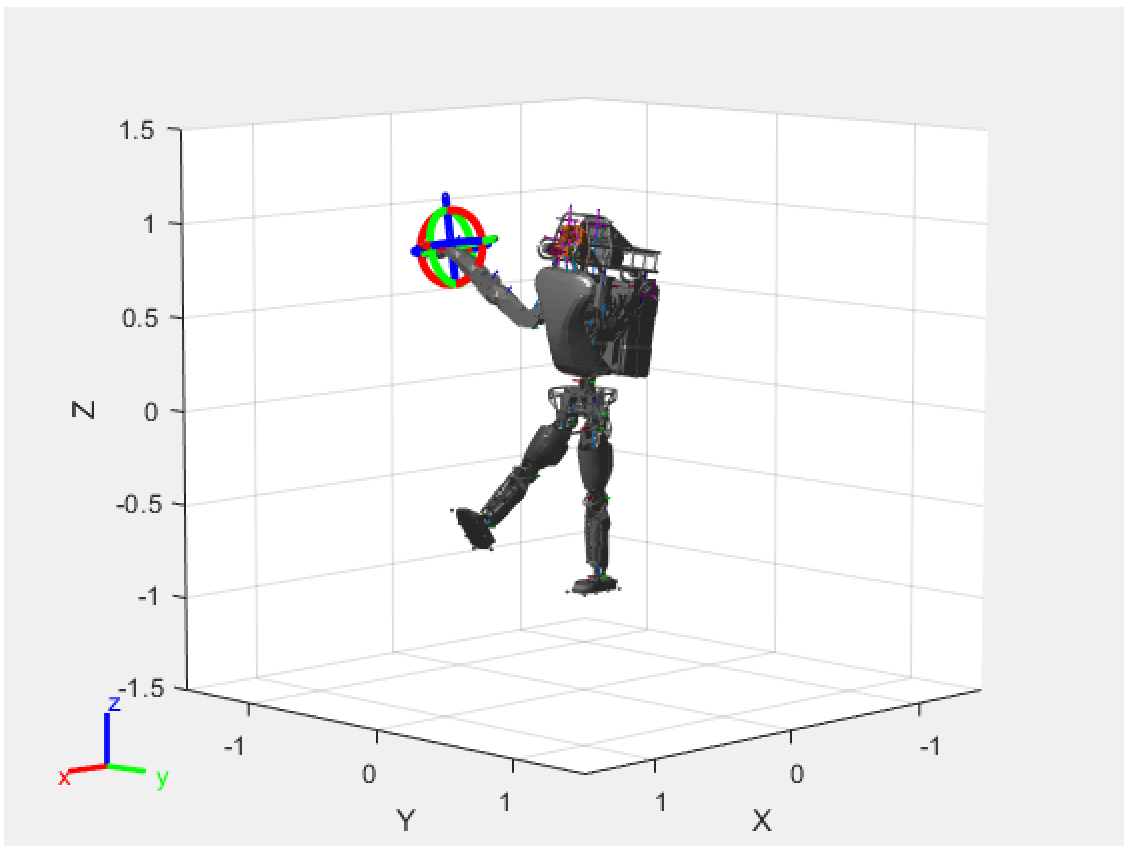
Notice the change in position of the right leg. Save this configuration as well.

```
addConfiguration(viztree)
```

Play Back Configurations

To play back configurations, iterate through the stored configurations index and set the current configuration equal to the stored configuration column vector at each iteration. Because configurations are stored as column vectors, use the second dimension of the matrix.

```
for i = 1:size(viztree.StoredConfigurations,2)
    viztree.Configuration = viztree.StoredConfigurations(:,i);
    pause(0.5)
end
```



Input Arguments

viztree — Interactive rigid body tree robot model visualization
interactiveRigidBodyTree object

Interactive rigid body tree robot model visualization, specified as an interactiveRigidBodyTree object.

index — Index locations to remove configurations
positive integer | vector of positive integers

Index locations to remove configurations, specified as a positive integer or vector of positive integers.

Data Types: double

Version History

Introduced in R2020a

See Also

Functions
addConfiguration | loadrobot | importrobot | homeConfiguration

Objects

`interactiveRigidBodyTree` | `rigidBodyTree` | `rigidBody` | `rigidBodyJoint` | `generalizedInverseKinematics`

Topics

"Rigid Body Tree Robot Model"
"Plan a Reaching Trajectory With Multiple Kinematic Constraints"
"Trajectory Control Modeling with Inverse Kinematics"

removeConstraints

Remove inverse kinematics constraints

Syntax

```
removeConstraints(viztree)
removeConstraints(viztree,index)
```

Description

`removeConstraints(viztree)`removes the constraint stored at the last index of the `Constraints` property of the `interactiveRigidBodyTree` object, `viztree`.

`removeConstraints(viztree,index)` removes the constraints with the specified indices.

Examples

Interactively Build and Play Back Series of Robot Configurations

Use the `interactiveRigidBodyTree` object to manually move around a robot in a figure. The object uses interactive markers in the figure to track the desired poses of different rigid bodies in the `rigidBodyTree` object.

Load Robot Model

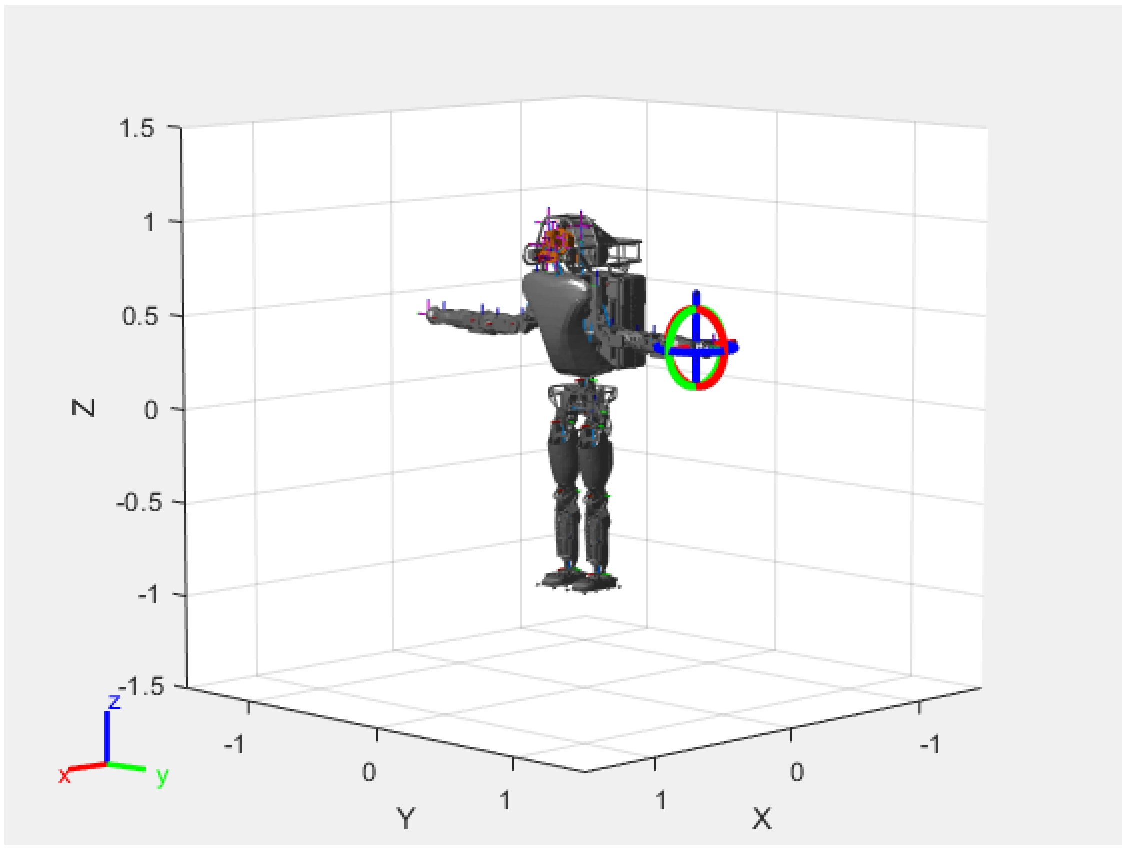
Use the `loadrobot` function to access provided robot models as `rigidBodyTree` objects.

```
robot = loadrobot("atlas");
```

Visualize Robot and Save Configurations

Create an interactive tree object and associated figure, specifying the loaded robot model and its left hand as the end effector.

```
viztree = interactiveRigidBodyTree(robot,"MarkerBodyName","l_hand");
```

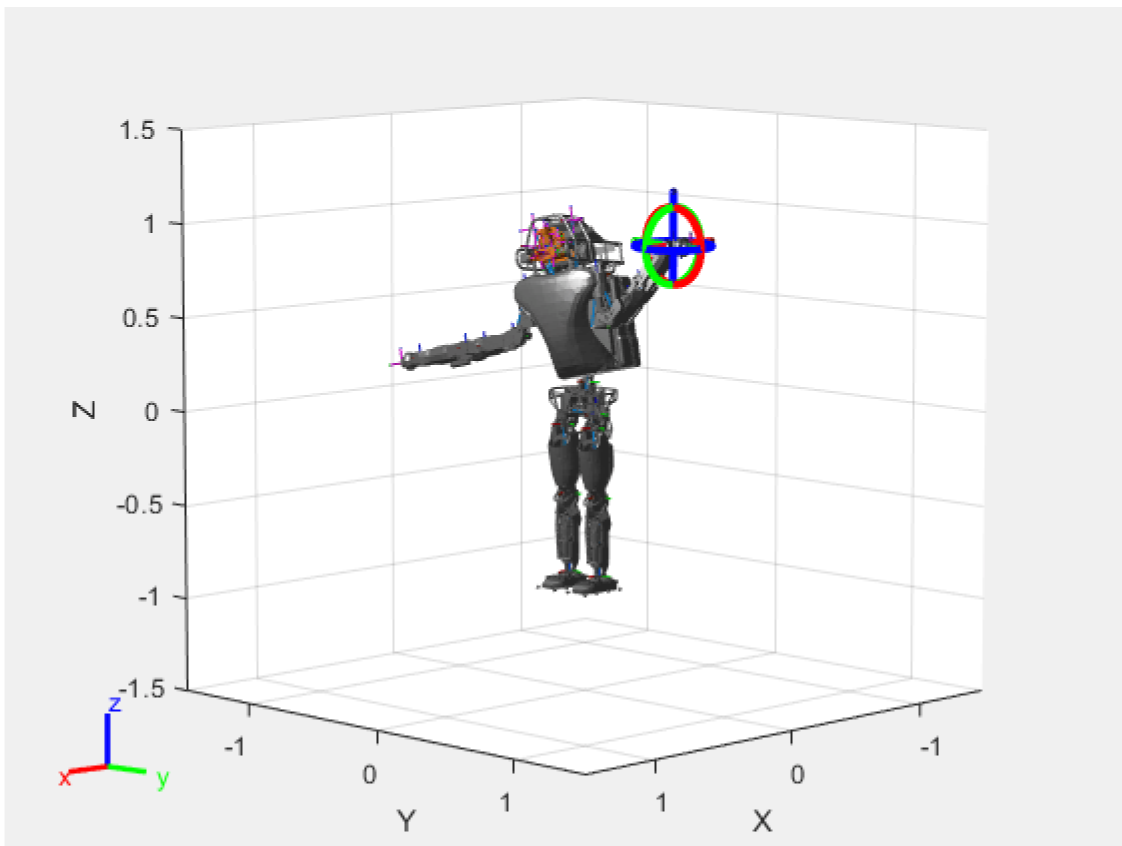


Click and drag the interactive marker to change the robot configuration. You can click and drag any of the axes for linear motion, rotate the body about an axis using the red, green, and blue circles, and drag the center of the interactive marker to position it in 3-D space.

The `interactiveRigidBodyTree` object uses inverse kinematics to determine a configuration that achieves the desired end-effector pose. If the associated rigid body cannot reach the marker, the figure renders the best configuration from the inverse kinematics solver.

Programmatically set the current configuration. Assign a vector of length equal to the number of nonfixed joints in the `RigidBodyTree` to the `Configuration` property.

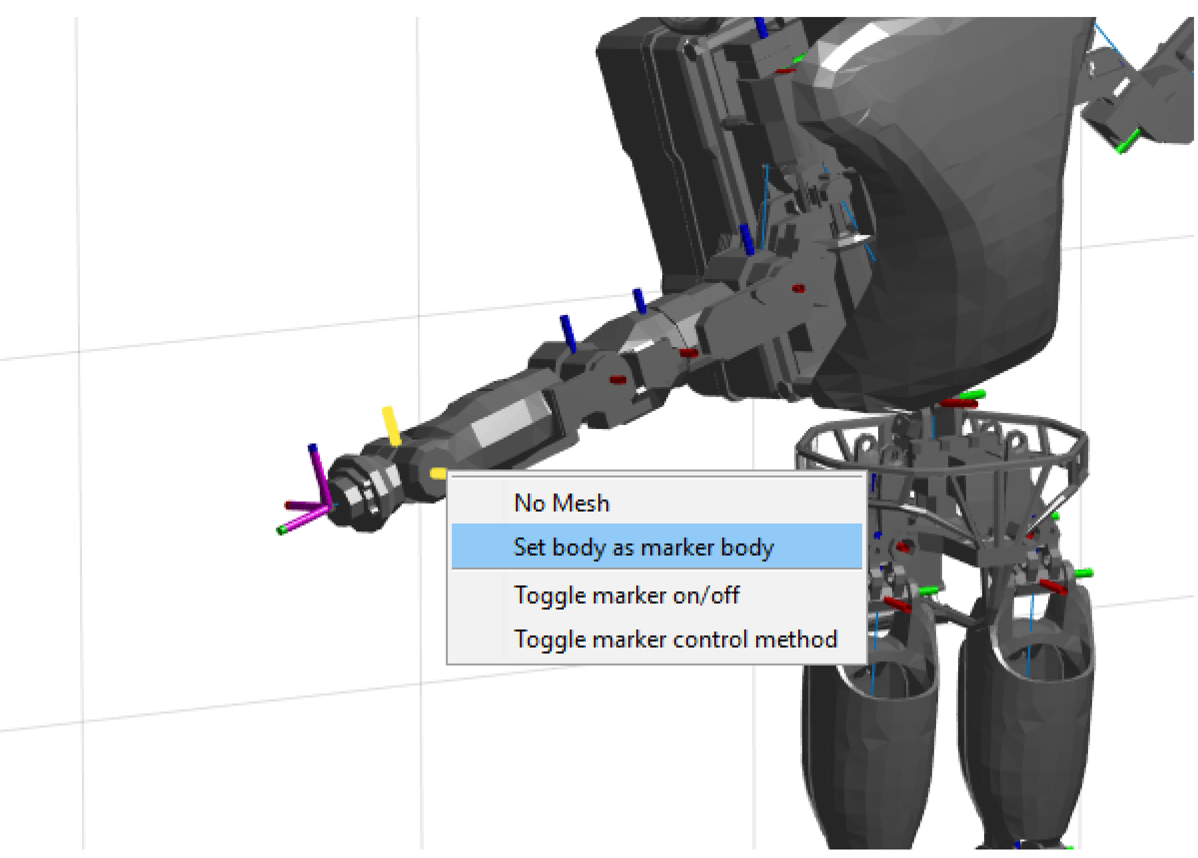
```
currConfig = homeConfiguration(viztree.RigidBodyTree);
currConfig(1:10) = [ 0.2201 -0.1319 0.2278 -0.3415 0.4996 ...
                     0.0747 0.0377 0.0718 -0.8117 -0.0427]';
viztree.Configuration = currConfig;
```



Save the current robot configuration in the `StoredConfigurations` property.

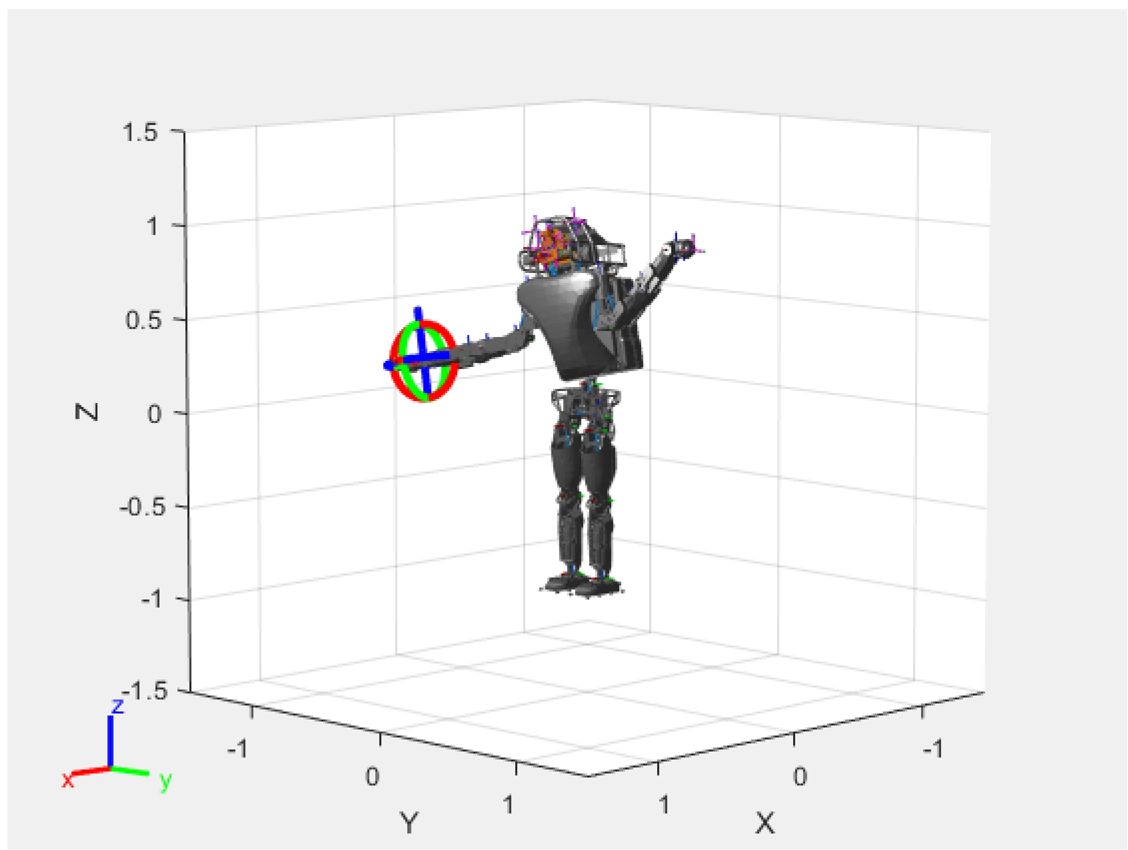
```
addConfiguration(viztree)
```

To switch the end effector to a different rigid body, right-click the desired body in the figure and select **Set body as marker body**. Use this process to select the right hand frame.



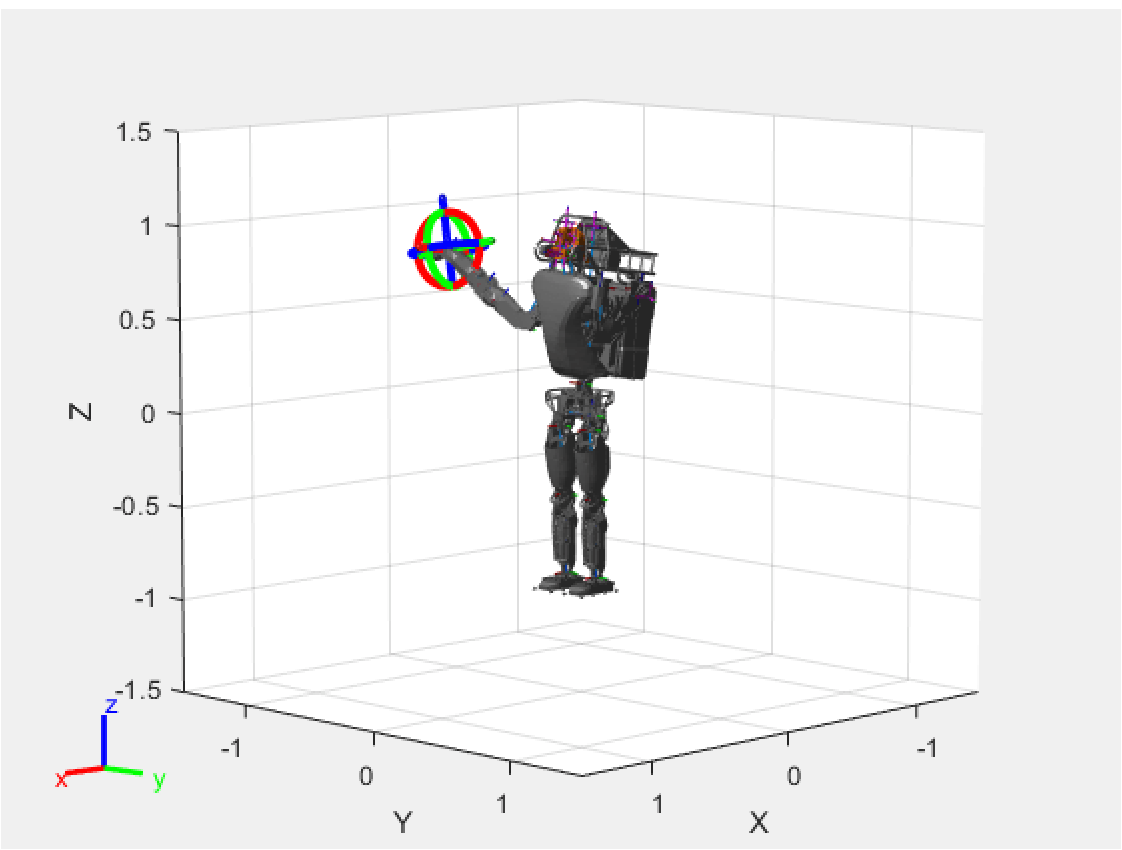
You can also set the `MarkerBodyName` property to the specific body name.

```
viztree.MarkerBodyName = "r_hand";
```



Move the right hand to a new position. Set the configuration programmatically. The marker moves to the new position of the end effector.

```
currConfig(1:18) = [-0.1350 -0.1498 -0.0167 -0.3415 0.4996 0.0747
                     0.0377 0.0718 -0.8117 -0.0427 0 0.4349
                    -0.5738 0.0563 -0.0095 0.0518 0.8762 -0.0895]';

viztree.Configuration = currConfig;
```



Save the current configuration.

```
addConfiguration(viztree)
```

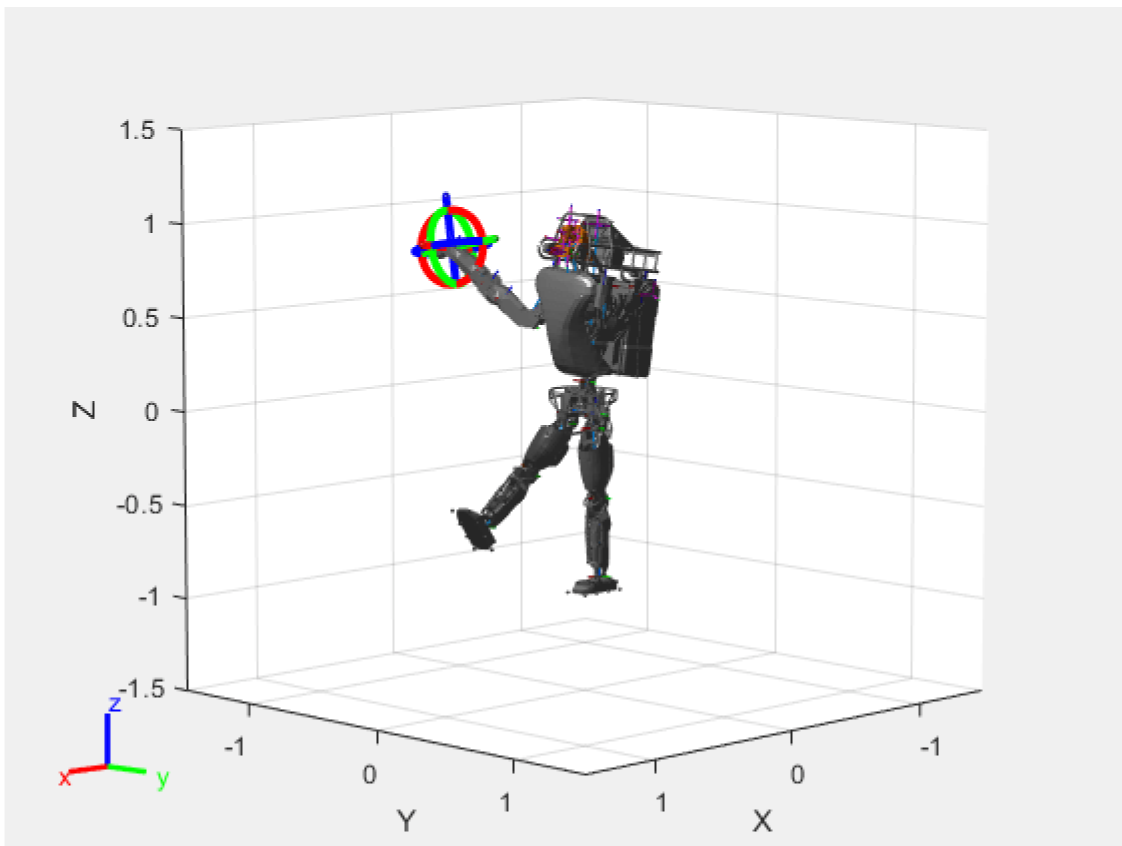
Add Constraints

By default, the robot model respects only the joint limits of the `rigidBodyJoint` objects associated with the `RigidBodyTree` property. To add constraints, generate **Robot Constraint** objects and specify them as a cell array in the `Constraints` property. To see a list of robotic constraints, see "Inverse Kinematics". Specify a pose target for the pelvis to keep it fixed to the home position. Specify a position target for the right foot to be raised in front front and above its current position.

```
fixedWaist = constraintPoseTarget("pelvis");
raiseRightLeg = constraintPositionTarget("r_foot","TargetPosition",[1 0 0.5]);
```

Apply these constraints to the interactive rigid body tree object as a cell array. The right leg in the resulting figure changes position.

```
viztree.Constraints = {fixedWaist raiseRightLeg};
```



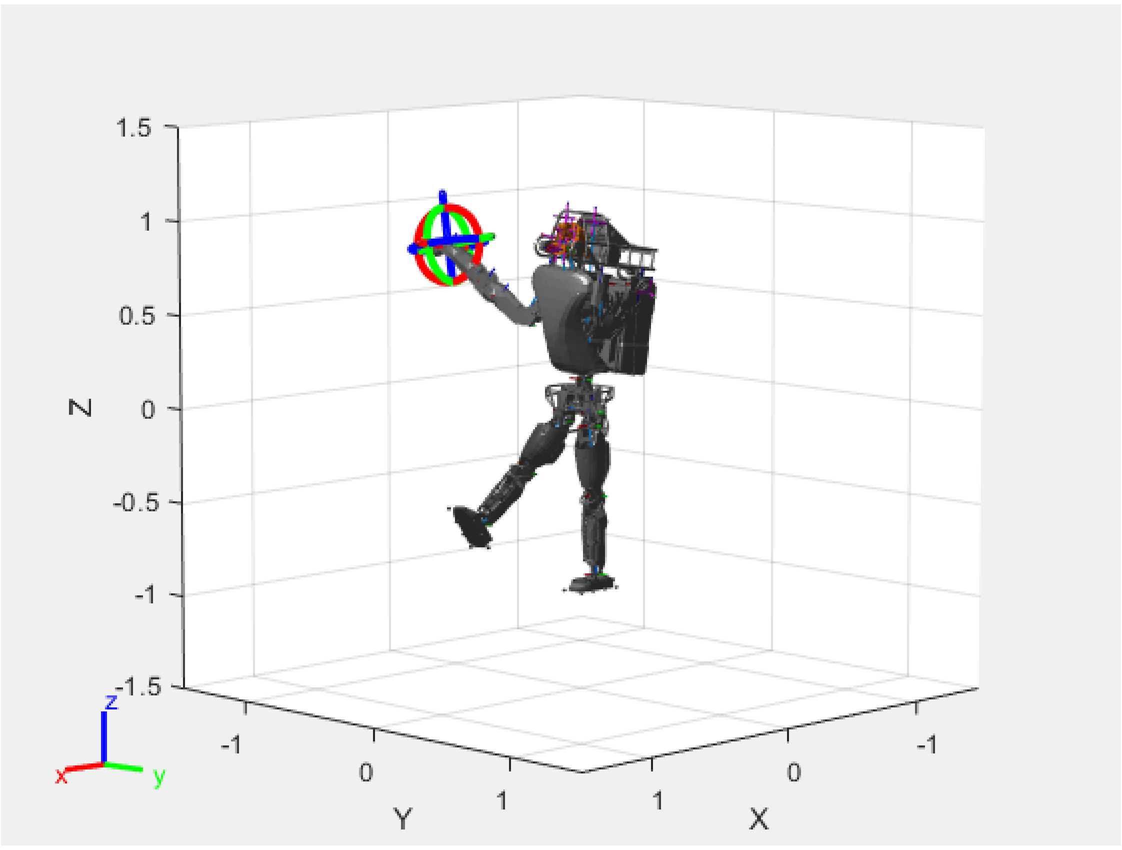
Notice the change in position of the right leg. Save this configuration as well.

```
addConfiguration(viztree)
```

Play Back Configurations

To play back configurations, iterate through the stored configurations index and set the current configuration equal to the stored configuration column vector at each iteration. Because configurations are stored as column vectors, use the second dimension of the matrix.

```
for i = 1:size(viztree.StoredConfigurations,2)
    viztree.Configuration = viztree.StoredConfigurations(:,i);
    pause(0.5)
end
```



Input Arguments

viztree — Interactive rigid body tree robot model visualization
interactiveRigidBodyTree object

Interactive rigid body tree robot model visualization, specified as an `interactiveRigidBodyTree` object.

index — Index locations to remove configurations
positive integer | vector of positive integers

Index locations to remove configurations, specified as a positive integer or vector of positive integers.

Data Types: `double`

Version History

Introduced in R2020a

See Also

Functions
`addConstraint` | `loadrobot` | `importrobot` | `homeConfiguration`

Objects
`interactiveRigidBodyTree` | `rigidBodyTree` | `rigidBody` | `rigidBodyJoint` | `generalizedInverseKinematics`

Topics
"Rigid Body Tree Robot Model"
"Plan a Reaching Trajectory With Multiple Kinematic Constraints"
"Trajectory Control Modeling with Inverse Kinematics"

showFigure

Show interactive rigid body tree figure

Syntax

```
showFigure(viztree)
```

Description

`showFigure(viztree)` shows the figure associated with the `interactiveRigidBodyTree` object, `viztree`. If the figure window is open, the function brings it into focus. If the figure window is not open, the function opens it and brings it into focus.

Examples

Interactively Build and Play Back Series of Robot Configurations

Use the `interactiveRigidBodyTree` object to manually move around a robot in a figure. The object uses interactive markers in the figure to track the desired poses of different rigid bodies in the `rigidBodyTree` object.

Load Robot Model

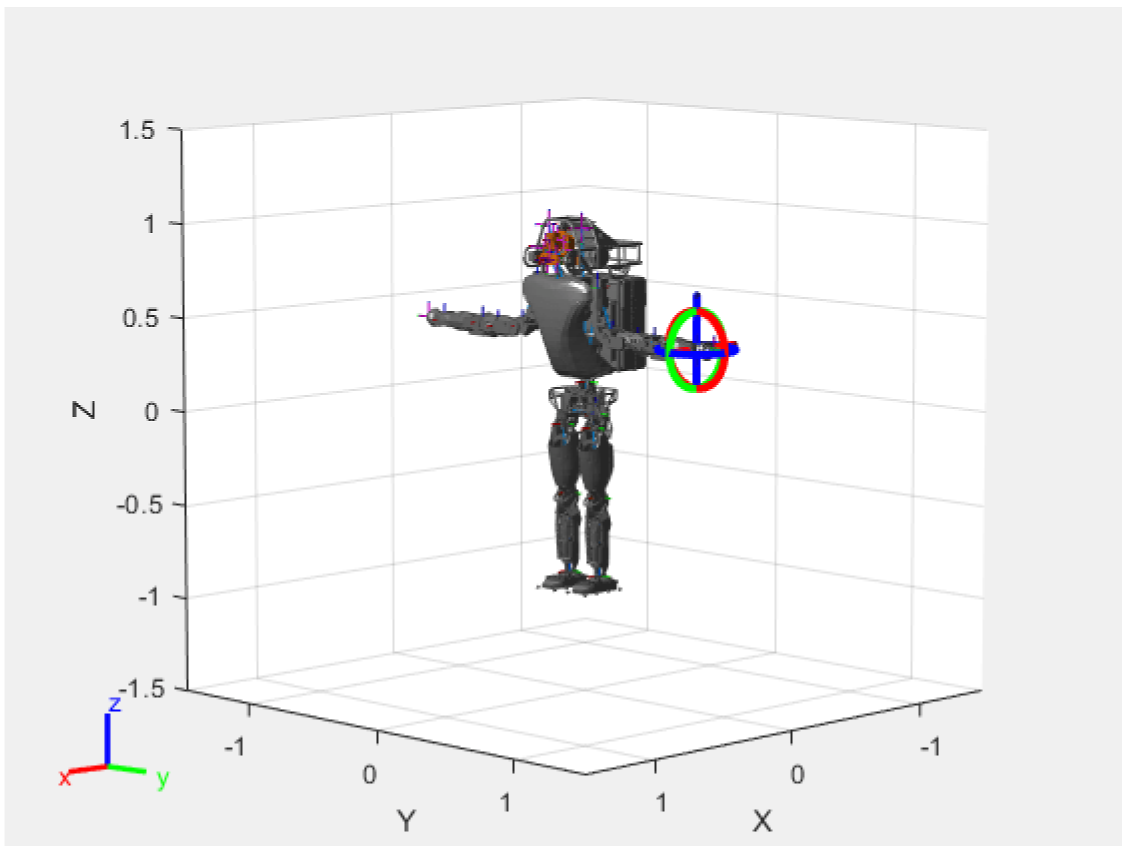
Use the `loadrobot` function to access provided robot models as `rigidBodyTree` objects.

```
robot = loadrobot("atlas");
```

Visualize Robot and Save Configurations

Create an interactive tree object and associated figure, specifying the loaded robot model and its left hand as the end effector.

```
viztree = interactiveRigidBodyTree(robot,"MarkerBodyName","l_hand");
```

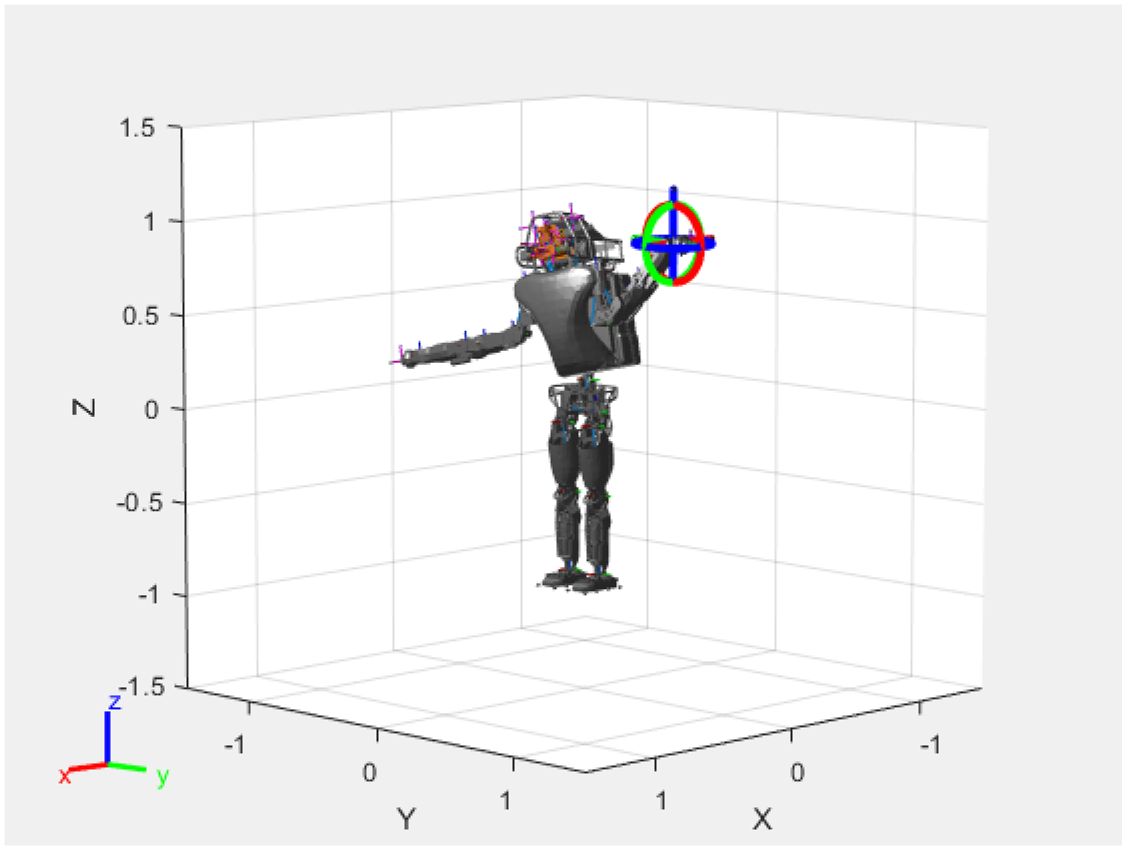


Click and drag the interactive marker to change the robot configuration. You can click and drag any of the axes for linear motion, rotate the body about an axis using the red, green, and blue circles, and drag the center of the interactive marker to position it in 3-D space.

The `interactiveRigidBodyTree` object uses inverse kinematics to determine a configuration that achieves the desired end-effector pose. If the associated rigid body cannot reach the marker, the figure renders the best configuration from the inverse kinematics solver.

Programmatically set the current configuration. Assign a vector of length equal to the number of nonfixed joints in the `RigidBodyTree` to the `Configuration` property.

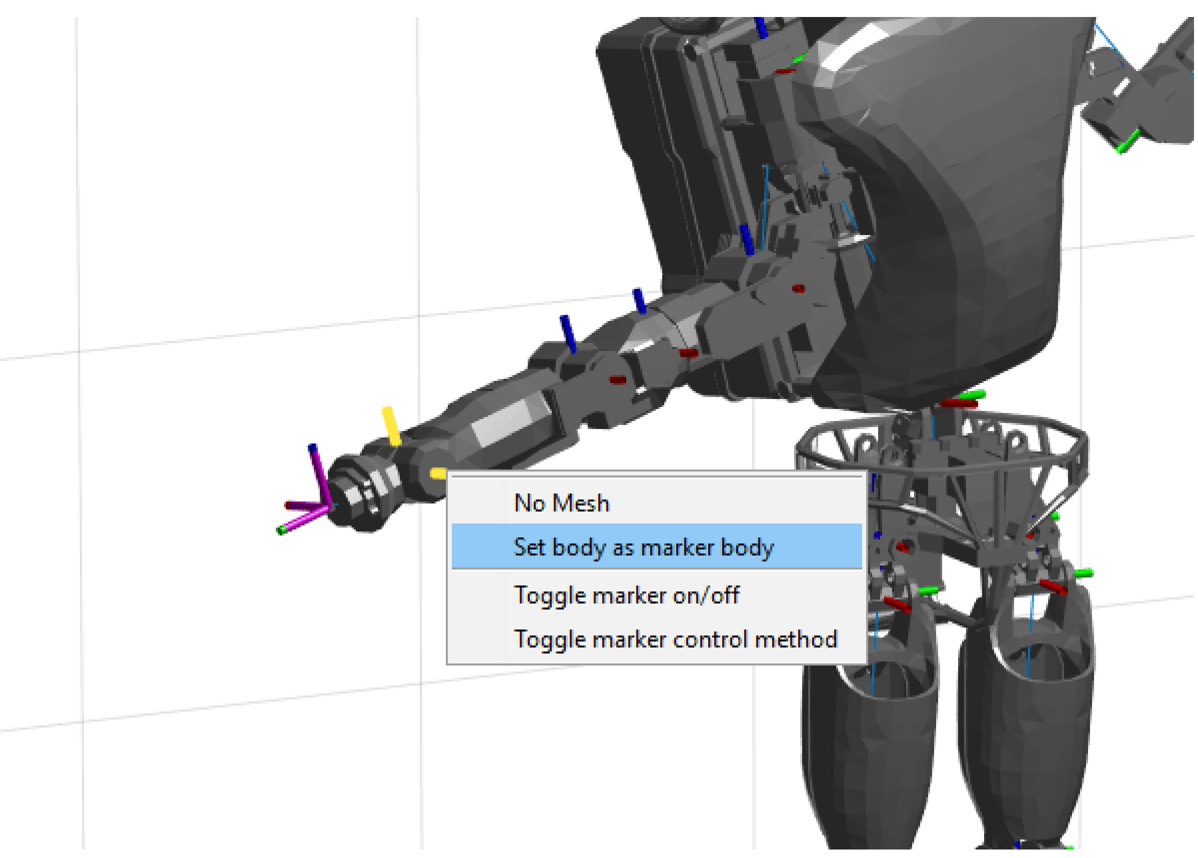
```
currConfig = homeConfiguration(viztree.RigidBodyTree);
currConfig(1:10) = [ 0.2201 -0.1319 0.2278 -0.3415 0.4996 ...
                     0.0747 0.0377 0.0718 -0.8117 -0.0427]';
viztree.Configuration = currConfig;
```



Save the current robot configuration in the `StoredConfigurations` property.

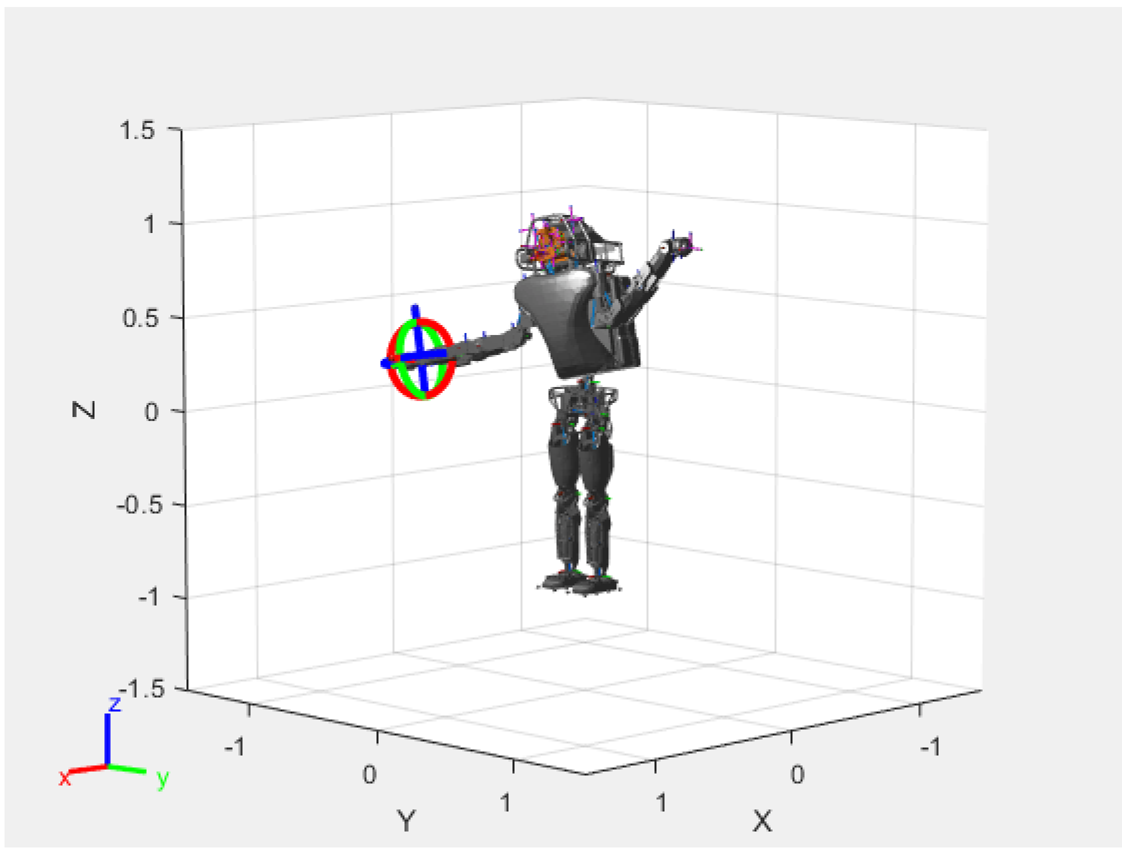
```
addConfiguration(viztree)
```

To switch the end effector to a different rigid body, right-click the desired body in the figure and select **Set body as marker body**. Use this process to select the right hand frame.



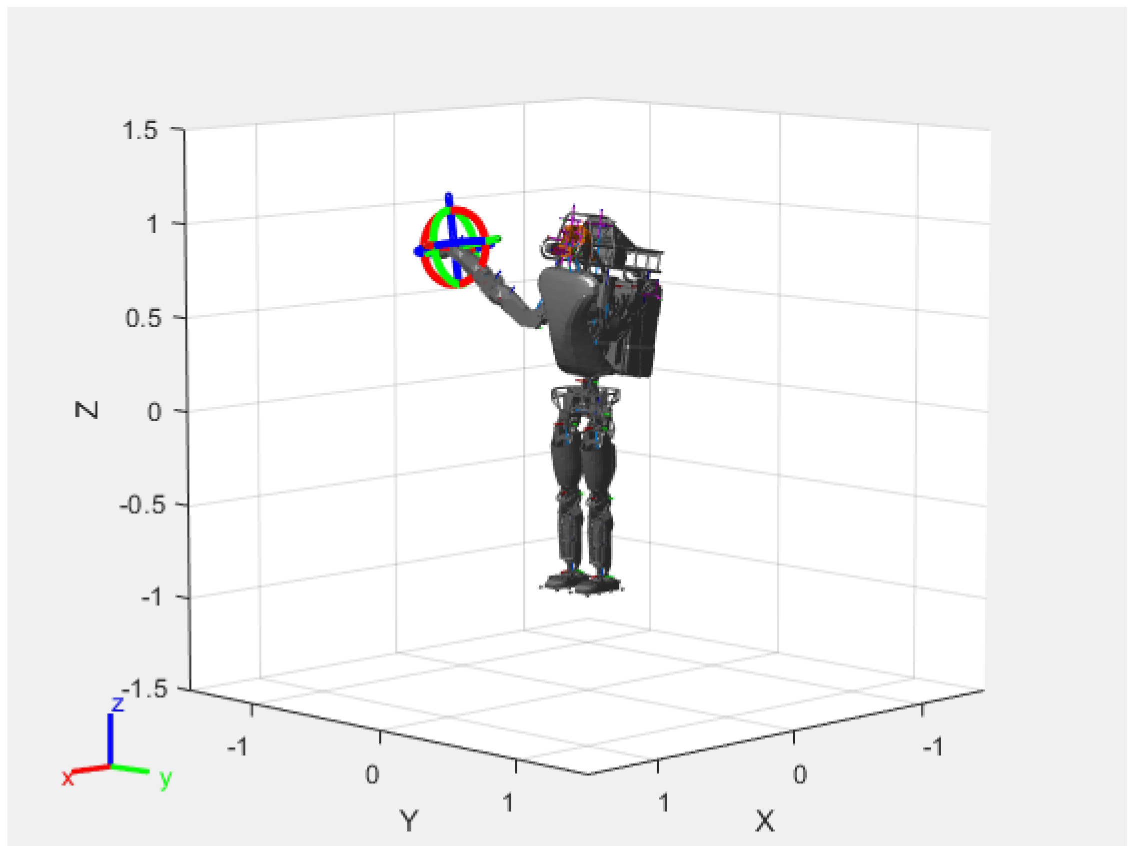
You can also set the `MarkerBodyName` property to the specific body name.

```
viztree.MarkerBodyName = "r_hand";
```



Move the right hand to a new position. Set the configuration programmatically. The marker moves to the new position of the end effector.

```
currConfig(1:18) = [-0.1350 -0.1498 -0.0167 -0.3415 0.4996 0.0747
                     0.0377 0.0718 -0.8117 -0.0427 0 0.4349
                    -0.5738 0.0563 -0.0095 0.0518 0.8762 -0.0895]';

viztree.Configuration = currConfig;
```



Save the current configuration.

```
addConfiguration(viztree)
```

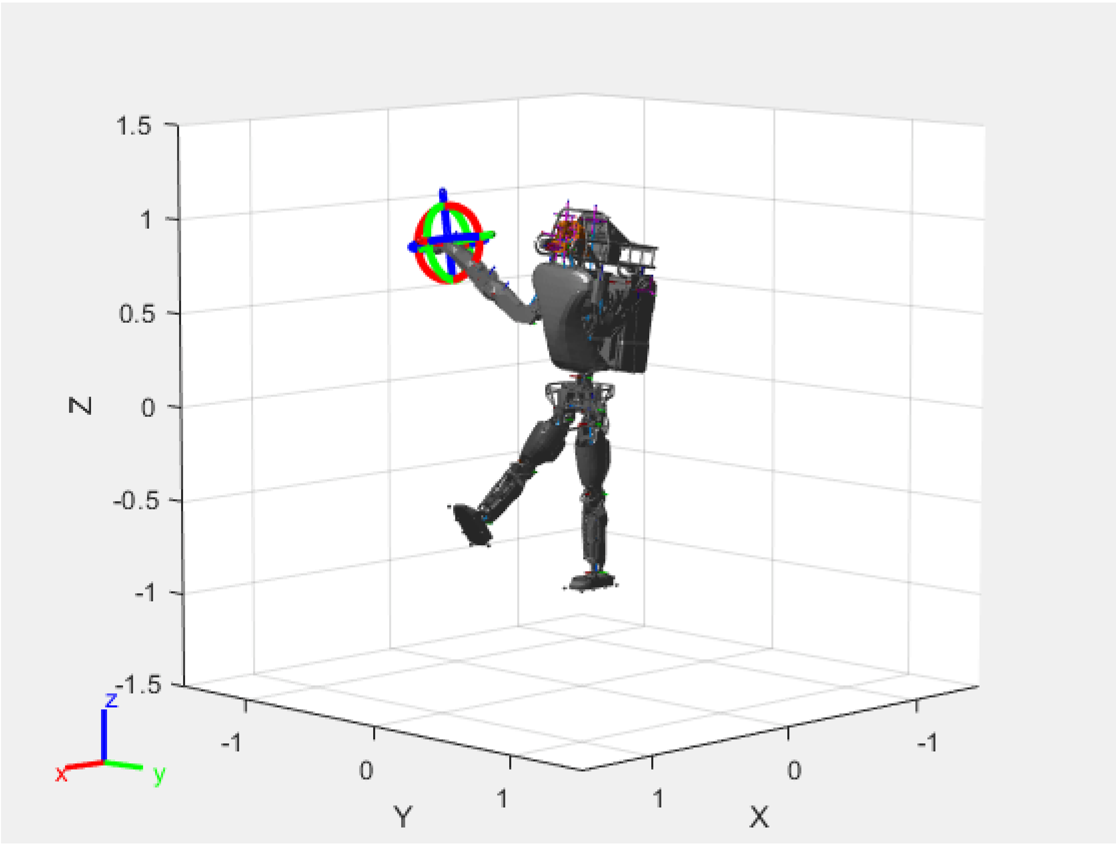
Add Constraints

By default, the robot model respects only the joint limits of the `rigidBodyJoint` objects associated with the `RigidBodyTree` property. To add constraints, generate **Robot Constraint** objects and specify them as a cell array in the `Constraints` property. To see a list of robotic constraints, see "Inverse Kinematics". Specify a pose target for the pelvis to keep it fixed to the home position. Specify a position target for the right foot to be raised in front front and above its current position.

```
fixedWaist = constraintPoseTarget("pelvis");
raiseRightLeg = constraintPositionTarget("r_foot","TargetPosition",[1 0 0.5]);
```

Apply these constraints to the interactive rigid body tree object as a cell array. The right leg in the resulting figure changes position.

```
viztree.Constraints = {fixedWaist raiseRightLeg};
```



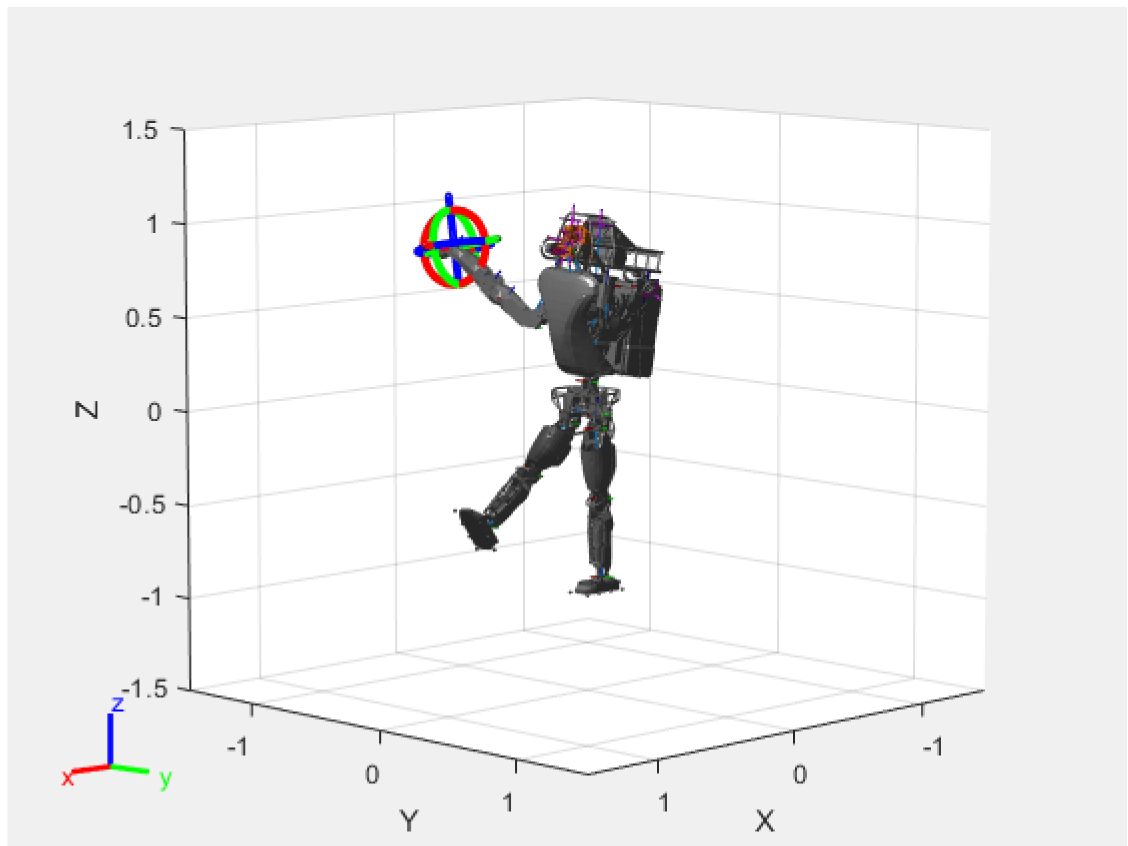
Notice the change in position of the right leg. Save this configuration as well.

```
addConfiguration(viztree)
```

Play Back Configurations

To play back configurations, iterate through the stored configurations index and set the current configuration equal to the stored configuration column vector at each iteration. Because configurations are stored as column vectors, use the second dimension of the matrix.

```
for i = 1:size(viztree.StoredConfigurations,2)
    viztree.Configuration = viztree.StoredConfigurations(:,i);
    pause(0.5)
end
```



Input Arguments

viztree — Interactive rigid body tree robot model visualization
interactiveRigidBodyTree object

Interactive rigid body tree robot model visualization, specified as an interactiveRigidBodyTree object.

Version History

Introduced in R2020a

See Also

Functions
loadrobot | importrobot | homeConfiguration

Objects
interactiveRigidBodyTree | rigidBodyTree | rigidBody | rigidBodyJoint | generalizedInverseKinematics

Topics
"Rigid Body Tree Robot Model"
"Plan a Reaching Trajectory With Multiple Kinematic Constraints"

“Trajectory Control Modeling with Inverse Kinematics”

removeInvalidData

Remove invalid range and angle data

Syntax

```
validScan = removeInvalidData(scan)
validScan = removeInvalidData(scan,Name,Value)
```

Description

`validScan = removeInvalidData(scan)` returns a new `lidarScan` object with all `Inf` and `NaN` values from the input `scan` removed. The corresponding angle readings are also removed.

`validScan = removeInvalidData(scan,Name,Value)` provides additional options specified by one or more `Name,Value` pairs.

Examples

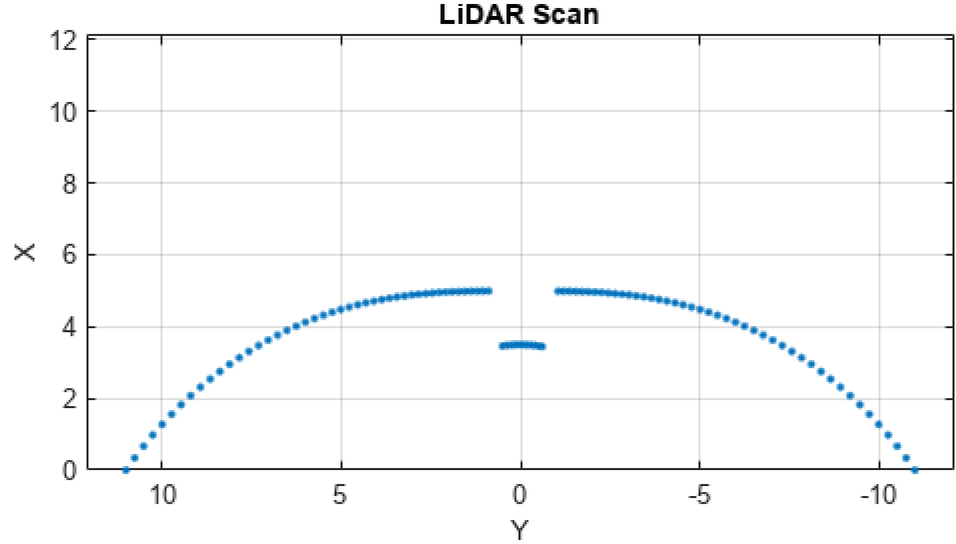
Plot Lidar Scan and Remove Invalid Points

Specify lidar data as vectors of ranges and angles. These values include readings outside of the sensors range.

```
x = linspace(-2,2);
ranges = abs((1.5).*x.^2 + 5);
ranges(45:55) = 3.5;
angles = linspace(-pi/2,pi/2,numel(ranges));
```

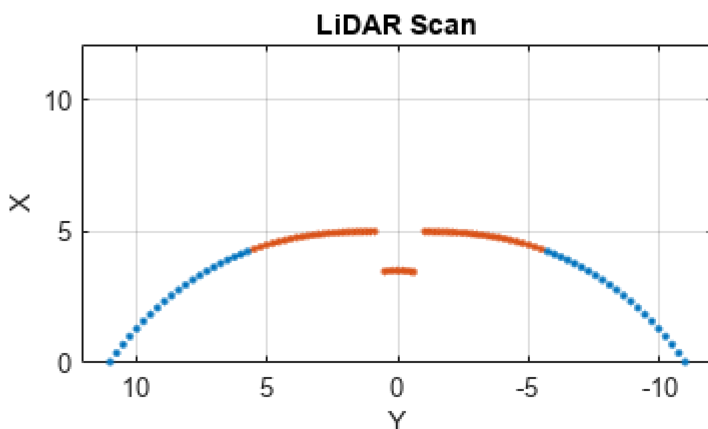
Create a lidar scan by specifying the ranges and angles. Plot all points of the lidar scan.

```
scan = lidarScan(ranges,angles);
plot(scan)
```



Remove invalid points based on a specified minimum and maximum range.

```
minRange = 0.1;
maxRange = 7;
scan2 = removeInvalidData(scan,'RangeLimits',[minRange maxRange]);
hold on
plot(scan2)
legend('All Points','Valid Points')
```



Input Arguments

scan — Lidar scan readings
`lidarScan` object

Lidar scan readings, specified as a `lidarScan` object.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `["RangeLimits",[0.05 2]`

`RangeLimits` — Range reading limits
two-element vector

Range reading limits, specified as a two-element vector, `[minRange maxRange]`, in meters. All range readings and corresponding angles outside these range limits are removed

Data Types: `single` | `double`

`AngleLimits` — Angle limits
two-element vector

Angle limits, specified as a two-element vector, `[minAngle maxAngle]` in radians. All angles and corresponding range readings outside these angle limits are removed.

Angles are measured counter-clockwise around the positive *z*-axis.

Data Types: `single` | `double`

Output Arguments

`validScan` — Lidar scan readings
`lidarScan` object

Lidar scan readings, specified as a `lidarScan` object. All invalid lidar scan readings are removed.

Version History

Introduced in R2017b

See Also

`transformScan`

plot

Display laser or lidar scan readings

Syntax

```
plot(scanObj)
plot( ___ ,Name,Value)
linehandle = plot( ___ )
```

Description

`plot(scanObj)` plots the lidar scan readings specified in `scanObj`.

`plot( ___ ,Name,Value)` provides additional options specified by one or more `Name,Value` pair arguments.

`linehandle = plot( ___ )` returns a column vector of line series handles, using any of the arguments from previous syntaxes. Use `linehandle` to modify properties of the line series after it is created.

Examples

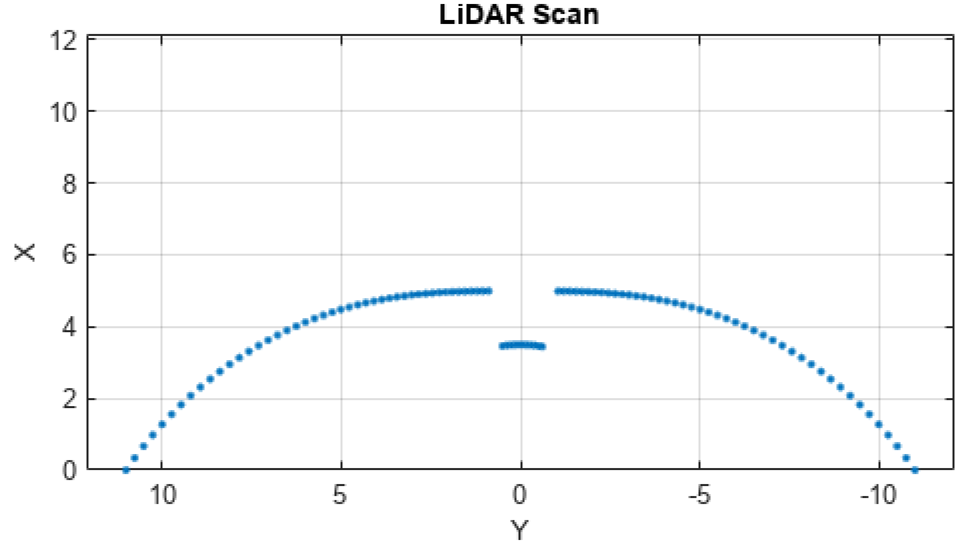
Plot Lidar Scan and Remove Invalid Points

Specify lidar data as vectors of ranges and angles. These values include readings outside of the sensors range.

```
x = linspace(-2,2);
ranges = abs((1.5).*x.^2 + 5);
ranges(45:55) = 3.5;
angles = linspace(-pi/2,pi/2,numel(ranges));
```

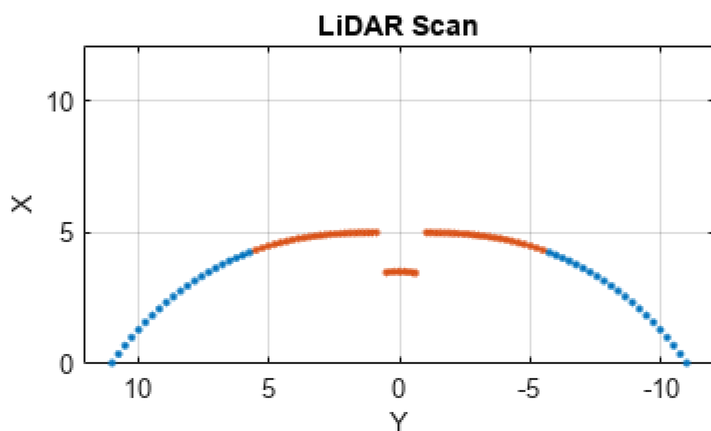
Create a lidar scan by specifying the ranges and angles. Plot all points of the lidar scan.

```
scan = lidarScan(ranges,angles);
plot(scan)
```



Remove invalid points based on a specified minimum and maximum range.

```
minRange = 0.1;
maxRange = 7;
scan2 = removeInvalidData(scan,'RangeLimits',[minRange maxRange]);
hold on
plot(scan2)
legend('All Points','Valid Points')
```



Input Arguments

`scanObj` — Lidar scan readings
`lidarScan` object

Lidar scan readings, specified as a `lidarScan` object.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `"MaximumRange",5`

`Parent` — Parent of axes
axes object

Parent of axes, specified as the comma-separated pair consisting of `"Parent"` and an axes object in which the laser scan is drawn. By default, the laser scan is plotted in the currently active axes.

`MaximumRange` — Range of laser scan
`scan.RangeMax` (default) | scalar

Range of laser scan, specified as the comma-separated pair consisting of `"MaximumRange"` and a scalar. When you specify this name-value pair argument, the minimum and maximum *x*-axis and the maximum *y*-axis limits are set based on specified value. The minimum *y*-axis limit is automatically determined by the opening angle of the laser scanner.

This name-value pair only works when you input `scanMsg` as the laser scan.

Outputs

`linehandle` — One or more chart line objects
scalar | vector

One or more chart line objects, returned as a scalar or a vector. These are unique identifiers, which you can use to query and modify properties of a specific chart line.

Version History

Introduced in R2015a

See Also

`transformScan`

optimize

Optimize trajectory using CHOMP

Syntax

```
[optimtraj,timesamples] = optimize(manipCHOMP,wpts,tpts,timestep)
[optimtraj,timesamples] = optimize(manipCHOMP,wpts,tpts,
timestep,InitialTrajectoryFit=fittype)
[ ___ ,solninfo] = optimize( ___ )
```

Description

The `optimize` object function optimizes a trajectory for both smoothness and collision avoidance using Covariant Hamiltonian Optimization for Motion Planning (CHOMP), a gradient-descent based planner. To change the weights of the smoothness costs and the collision costs, set the `SmoothnessOptions`, `CollisionOptions`, and `SolverOptions` properties of the `manipulatorCHOMP` object. The function also assumes that both the environment and the collision geometry of the rigid body tree robot model are approximated as collision spheres.

`[optimtraj,timesamples] = optimize(manipCHOMP,wpts,tpts,timestep)` optimizes the trajectory specified by the waypoints `wpts` at times `tpts`, and using the Covariant Hamiltonian Optimization for Motion Planning (CHOMP) algorithm, outputs the optimized waypoints `optimtraj` at the sample times `timesamples`. The initial trajectory fit between waypoints at their respective times is linear interpolation.

`[optimtraj,timesamples] = optimize(manipCHOMP,wpts,tpts,timestep,InitialTrajectoryFit=fittype)` specifies a trajectory type `fittype` for the initial trajectory fitting.

`[ ___ ,solninfo] = optimize( ___ )` returns solution information from the optimization.

Examples

Optimize Collision-Free Trajectory with CHOMP

Load a robot model into the workspace, and create a CHOMP solver.

```
robot = loadrobot("kinovaGen3",DataFormat="row");
chomp = manipulatorCHOMP(robot);
```

Create spheres to represent obstacles, and add them to the CHOMP solver.

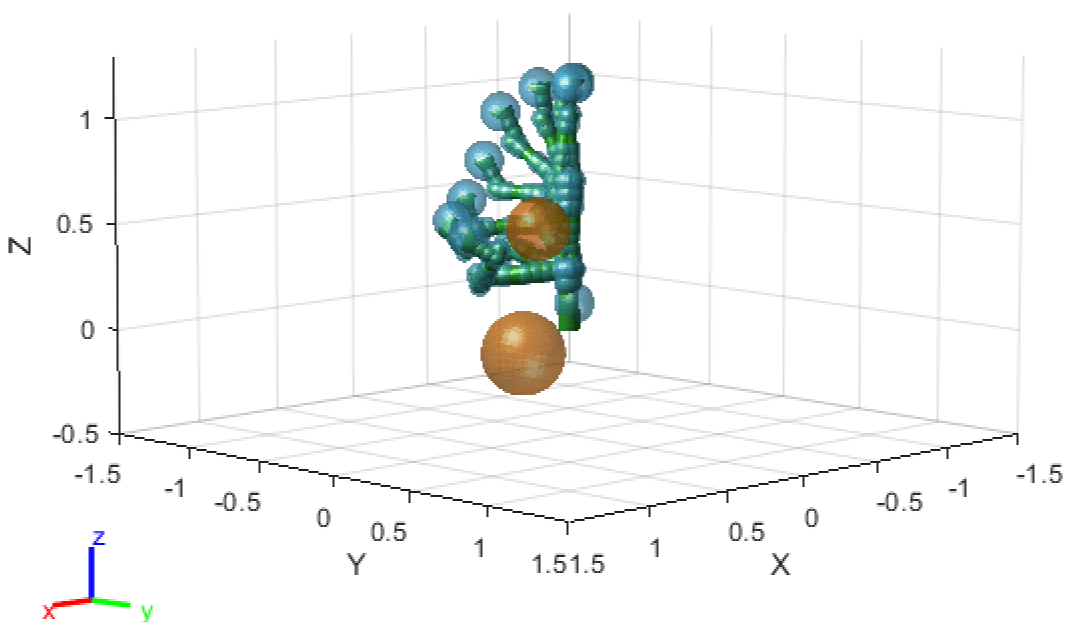
```
env = [0.20 0.2 -0.1 -0.1;   % sphere, radius 0.20 at (0.2,-0.1,-0.1)
       0.15 0.2  0.0  0.5]'; % sphere, radius 0.15 at (0.2,0.0,0.5)
chomp.SphericalObstacles = env;
```

To prioritize a collision-free trajectory, set the smoothness cost weight to a lower value than the collision cost weight. Then add the options to the CHOMP solver.

```
chomp.SmoothnessOptions = chompSmoothnessOptions(SmoothnessCostWeight=1e-3);
chomp.CollisionOptions = chompCollisionOptions(CollisionCostWeight=10);
chomp.SolverOptions = chompSolverOptions(Verbosity="none",LearningRate=7.0);
```

Initialize a trajectory, optimize it using the CHOMP solver, and show the waypoints in a figure.

```
startconfig = homeConfiguration(robot);
goalconfig = [0.5 1.75 -2.25 2.0 0.3 -1.65 -0.4];
timepoints = [0 5];
timestep = 0.1;
trajtype = "minjerkpolytraj";
[wptsamples,tsamples] = optimize(chomp, ...
    [startconfig; goalconfig], ...
    timepoints, ...
    timestep, ...
    InitialTrajectoryFitType=trajtype);
show(chomp,wptsamples,NumSamples=10);
zlim([-0.5 1.3])
```



Input Arguments

manipCHOMP — CHOMP solver
`manipulatorCHOMP` object

CHOMP solver, specified as a `manipulatorCHOMP` object.

`wpts` — Trajectory waypoints to optimize
N-by-*M* matrix

Trajectory waypoints to optimize, specified as an *N*-by-*M* matrix. *N* is the total number of waypoints, and *M* is the size of a joint configuration for the rigid body tree in the `RigidBodyTree` property of `manipCHOMP`.

`tpts` — Trajectory waypoint times
N-element row vector

Trajectory waypoint times, specified as an *N*-element row vector. *N* is the number of specified waypoints.

`timestep` — Time step size at which to discretize trajectory
positive numeric scalar

Time step size at which to discretize trajectory, specified as a positive numeric scalar.

`fittype` — Initial trajectory fit type
`"minjerkpolytraj"` (default) | `"interp1"` | `"quinticpolytraj"`

Initial trajectory fit type, specified as one of these options:

- `"minjerkpolytraj"` — Fit a minimum jerk polynomial trajectory. See the `minjerkpolytraj` function for more details about this trajectory type.
- `"interp1"` — Fit a linearly-interpolated trajectory. See the `interp1` function for more details about this trajectory type.
- `"quinticpolytraj"` — Fit a quintic polynomial trajectory. See the `quinticpolytraj` function for more details about this trajectory type.

The trajectory fit type is the trajectory that the `optimize` function uses to fit between waypoints at their specified time points.

Data Types: `char` | `string`

Output Arguments

`optimtraj` — Optimized waypoint samples of optimized trajectory
N-by-*M* matrix

Optimized waypoint samples of the optimized trajectory, returned as an `N-by-M` matrix. *N* is the total number of waypoints, and *M* is the size of a joint configuration for the rigid body tree in the `RigidBodyTree` property of `manipCHOMP`.

`timesamples` — Time samples for optimized waypoint samples
N-element row vector

Time samples for optimized waypoint samples, returned as an *N*-element row vector. *N* is the number of optimized waypoint samples.

`solninfo` — Solution information
structure

Solution information, returned as a structure containing these fields:

- `Iterations` — Number of iterations taken to optimize trajectory.
- `SmoothnessCost` — Smoothness cost at each iteration.
- `CollisionCost` — Collision cost at each iteration.
- `ObjectiveFunction` — Objective function value at each iteration.
- `InitialSmoothnessCost` — Initial smoothness cost of the trajectory.
- `InitialCollisionCost` — Initial collision cost of the trajectory.
- `InitialObjectiveFunction` — Initial objective function value of the trajectory.
- `OptimizationTime` — Time taken to optimize trajectory.
- `IsCollisionFree` — Indication of whether trajectory is collision free or not.

Version History

Introduced in R2023a

See Also

`manipulatorCHOMP` | `chompSolverOptions` | `chompSmoothnessOptions` | `chompCollisionOptions` | `show`

show

Visualize CHOMP trajectory of rigid body tree

Syntax

```
show(manipCHOMP)
show(manipCHOMP,traj)
ax = show(manipCHOMP,___)
[___] = show(___,Name=Value)
```

Description

The `show` object function shows one or more configuration trajectories in a Covariant Hamiltonian Optimization for Motion Planning (CHOMP) environment with obstacles. CHOMP is a gradient-descent based planner that plans and optimizes trajectories for smoothness and collision avoidance. For more information about CHOMP, see the `manipulatorCHOMP` object.

`show(manipCHOMP)` visualizes the rigid body tree in its home configuration in a CHOMP environment.

`show(manipCHOMP,traj)` visualizes the trajectory of the rigid body tree generated in a CHOMP setting. The spherical obstacles specified by the property `SphericalObstacles` are also overlaid on the visualization.

`ax = show(manipCHOMP,___)` returns the axes `ax` of the visualization.

`[___] = show(___,Name=Value)` specifies additional options using one or more name-value arguments.

Examples

Optimize Collision-Free Trajectory with CHOMP

Load a robot model into the workspace, and create a CHOMP solver.

```
robot = loadrobot("kinovaGen3",DataFormat="row");
chomp = manipulatorCHOMP(robot);
```

Create spheres to represent obstacles, and add them to the CHOMP solver.

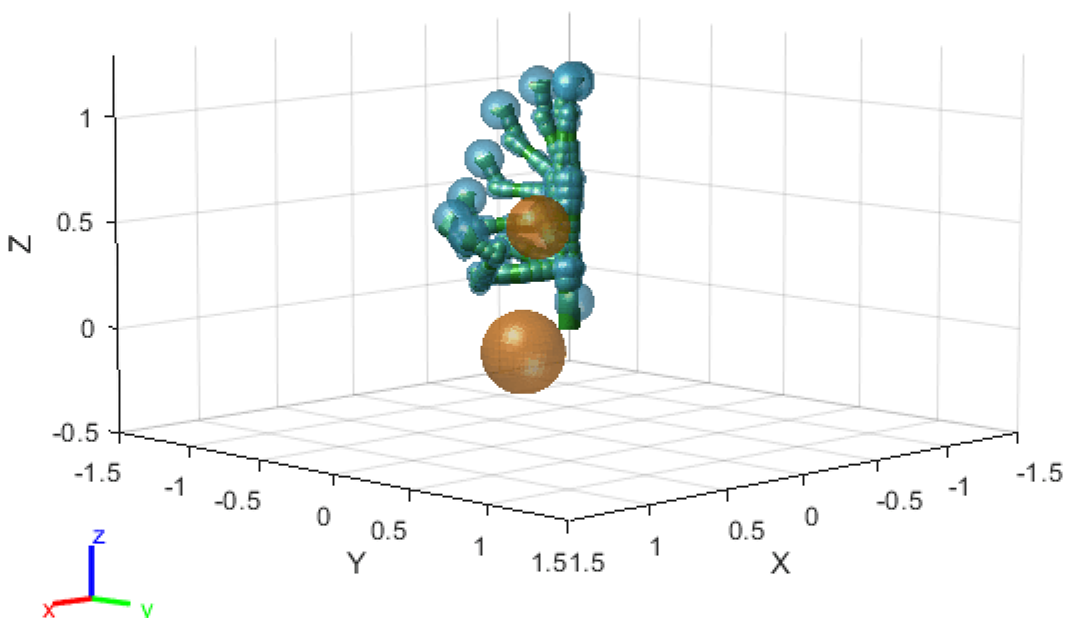
```
env = [0.20 0.2 -0.1 -0.1;   % sphere, radius 0.20 at (0.2,-0.1,-0.1)
       0.15 0.2  0.0  0.5]'; % sphere, radius 0.15 at (0.2,0.0,0.5)
chomp.SphericalObstacles = env;
```

To prioritize a collision-free trajectory, set the smoothness cost weight to a lower value than the collision cost weight. Then add the options to the CHOMP solver.

```
chomp.SmoothnessOptions = chompSmoothnessOptions(SmoothnessCostWeight=1e-3);
chomp.CollisionOptions = chompCollisionOptions(CollisionCostWeight=10);
chomp.SolverOptions = chompSolverOptions(Verbosity="none",LearningRate=7.0);
```

Initialize a trajectory, optimize it using the CHOMP solver, and show the waypoints in a figure.

```
startconfig = homeConfiguration(robot);
goalconfig = [0.5 1.75 -2.25 2.0 0.3 -1.65 -0.4];
timepoints = [0 5];
timestep = 0.1;
trajtype = "minjerkpolytraj";
[wptsamples,tsamples] = optimize(chomp, ...
    [startconfig; goalconfig], ...
    timepoints, ...
    timestep, ...
    InitialTrajectoryFitType=trajtype);
show(chomp,wptsamples,NumSamples=10);
zlim([-0.5 1.3])
```



Input Arguments

manipCHOMP — CHOMP optimizer
manipulatorCHOMP object

CHOMP optimizer, specified as a `manipulatorCHOMP` object.

traj — Trajectory
N-by-*M* matrix

Trajectory, specified as an N-by-M matrix. N is the total number of waypoints, and M is the size of a joint configuration for the rigid body tree in the `RigidBodyTree` property of `manipCHOMP`.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Example: `show(chomp,chomptraj,NumSamples=30)`

`NumSamples` — Number of samples of trajectory to uniformly sample

`20` (default) | positive integer scalar

Number of uniformly spaced joint configuration samples along the trajectory, specified as a positive integer scalar.

Example: `show(chomp,chomptraj,NumSamples=10)`

`Parent` — Parent of axes

`Axes` object

Parent of axes, specified as an `Axes` object on which to visualize the rigid body tree trajectories. By default, `show` plots the rigid body tree trajectories on the current axes.

`CollisionObjects` — Collision objects to visualize

`{}` (default) | cell array of collision objects

Collision objects to visualize, specified as a cell array of collision objects.

Example: `show(chomp,trajectory,CollisionObjects={collisionCylinder(1,2)})`

Output Arguments

ax — Axes graphic handle

`Axes` object

Axes graphic handle, returned as an `Axes` object. This object contains the properties of the figure that the rigid body tree is plotted onto.

Version History

Introduced in R2023a

See Also

`manipulatorCHOMP` | `chompSolverOptions` | `chompSmoothnessOptions` | `chompCollisionOptions` | `optimize`

isMotionValid

Check if path between states is valid

Syntax

`[isValid,lastValid] = isMotionValid(manipSV,startConfig,goalConfig)`

Description

`[isValid,lastValid] = isMotionValid(manipSV,startConfig,goalConfig)` checks if the path between two states is valid by interpolating between states using the state validator `manipSV`. The function also returns the last valid state along the path `lastValid`.

Examples

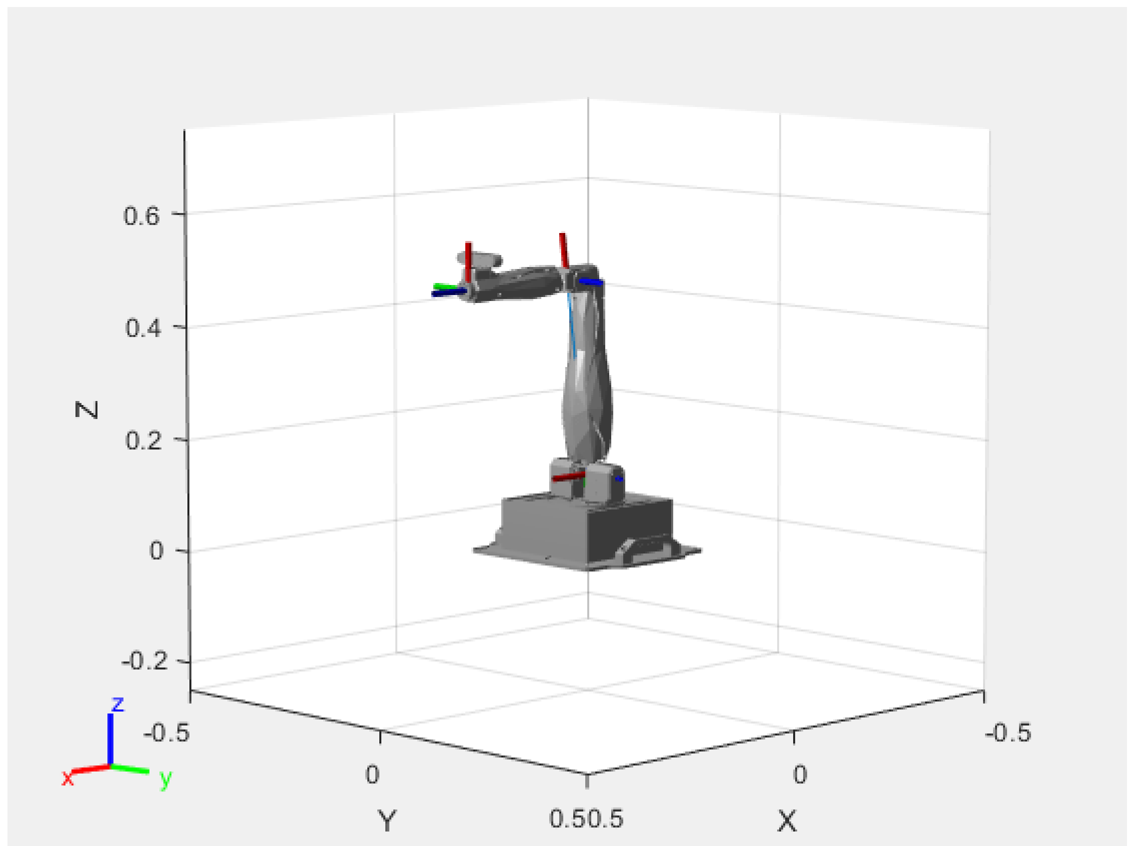
Validate State and Motion Manipulator State Space

Generate states to form a path, validate motion between states, and check for self-collisions and environmental collisions with objects in your world. The `manipulatorStateSpace` object represents the joint configuration space of your rigid body tree robot model, and can sample states, calculate distances, and enforce state bounds. The `manipulatorCollisionBodyValidator` object validates the state and motion based on the collision bodies in your robot model and any obstacles in your environment.

Load Robot Model

Use the `loadrobot` function to access predefined robot models. This example uses the Quanser QArm™ robot and joint configurations are specified as row vectors.

```
robot = loadrobot("quanserQArm",DataFormat="row");
figure(Visible="on")
show(robot);
xlim([-0.5 0.5])
ylim([-0.5 0.5])
zlim([-0.25 0.75])
hold on
```



Configure State Space and State Validation

Create the state space and state validator from the robot model.

```
ss = manipulatorStateSpace(robot);
sv = manipulatorCollisionBodyValidator(ss,SkippedSelfCollisions="parent");
```

Set the validation distance to `0.05`, which is based on the distance normal between two states. You can configure the validator to ignore self collisions to improve the speed of validation, but must consider whether your robot model has the proper joint limit settings set to ensure it does not collide with itself.

```
sv.ValidationDistance = 0.05;
sv.IgnoreSelfCollision = true;
```

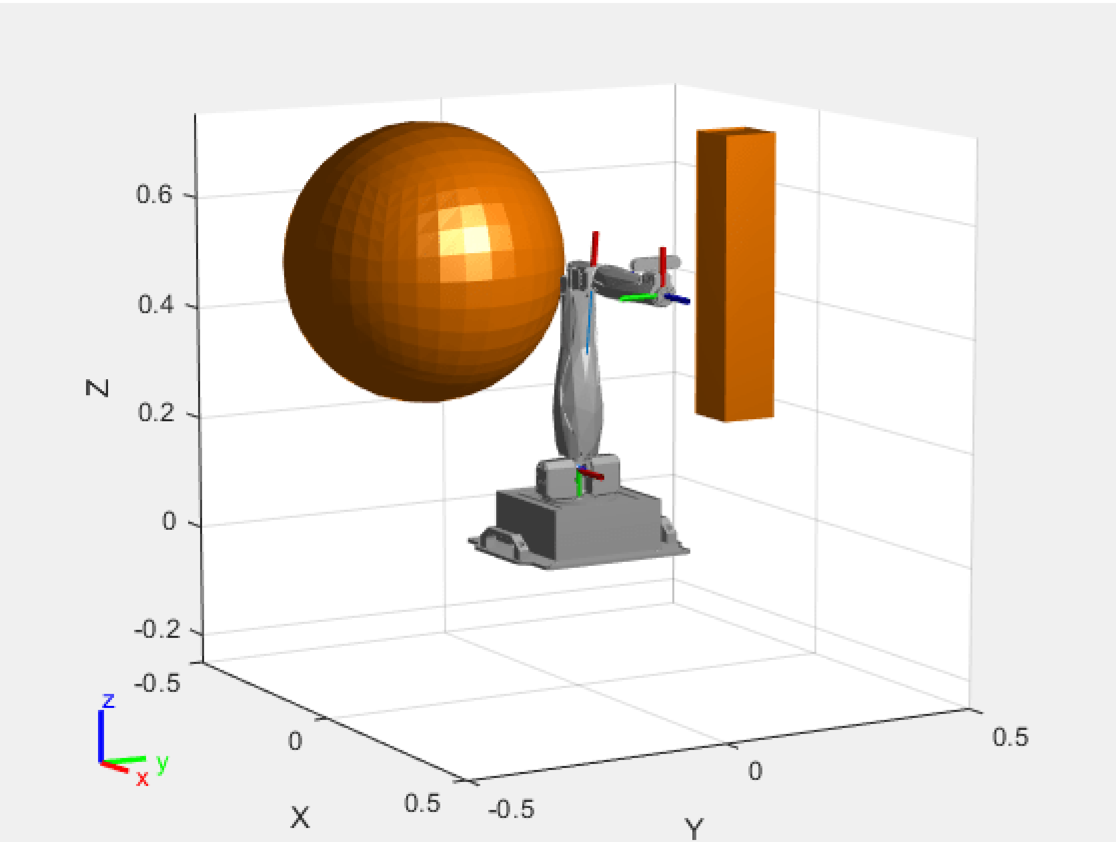
Place collision objects in the robot environment. Set the `Environment` property of the collision validator object using a cell array of objects.

```
box = collisionBox(0.1,0.1,0.5); % XYZ Lengths
box.Pose = trvec2tform([0.2 0.2 0.5]); % XYZ Position
sphere = collisionSphere(0.25); % Radius
sphere.Pose = trvec2tform([-0.2 -0.2 0.5]); % XYZ Position
env = {box sphere};
sv.Environment = env;
```

Visualize the environment.

```
for i = 1:length(env)
    show(env{i})
```

```
end
view(60,10)
```



Plan Path

Start with the home configuration as the first point on the path.

```
rng(0); % Repeatable results
start = homeConfiguration(robot);
path = start;
idx = 1;
```

Plan a path using these steps, in a loop:

- Sample a nearby goal configuration, using the Gaussian distribution, by specifying the standard deviation for each joint angle.
- Check if the sampled goal state is valid.
- If the sampled goal state is valid, check if the motion between states is valid and, if so, add it to the path.

```
for i = 2:25
    goal = sampleGaussian(ss,start,0.25*ones(4,1));
    validState = isStateValid(sv,goal);

    if validState % If state is valid, check motion between states.
        [validMotion,~] = isMotionValid(sv,path(idx,:),goal);
```

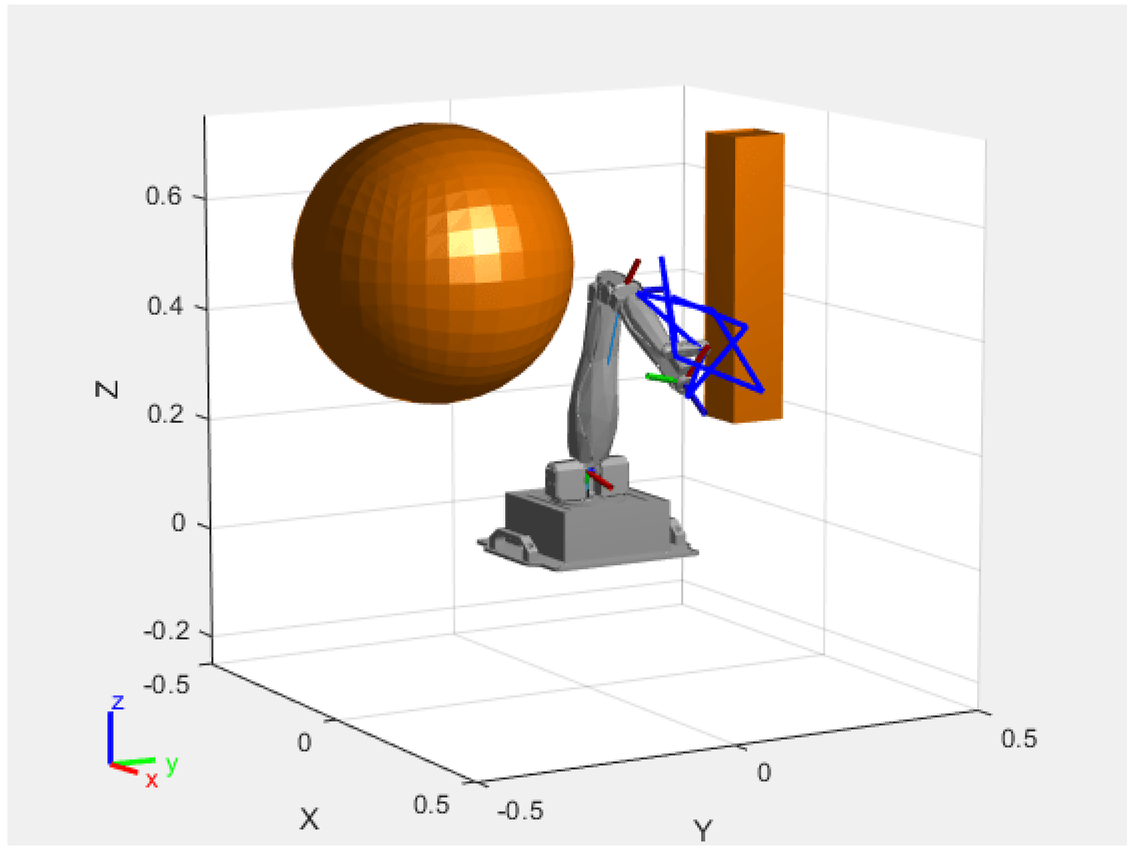
```
        if validMotion % If motion is valid, add to path.
            path = [path; goal];
            idx = idx + 1;
        end
    end
end
```

Visualize Path

After generating the path of valid motions, visualize the robot motion. Because you sampled random states near the home configuration, you should see the arm move around that initial configuration.

To visualize the path of the end effector in 3-D, get the transformation, relative to the base world frame at each point. Store the points as an *xyz* translation vector. Plot the path of the end effector.

```
eePose = nan(3,size(path,1));

for i = 1:size(path,1)
    show(robot,path(i,:),PreservePlot=false);
    eePos(i,:) = tform2trvec(getTransform(robot,path(i,:),"END-EFFECTOR")); % XYZ translation ve
    plot3(eePos(:,1),eePos(:,2),eePos(:,3),"-b",LineWidth=2)
    drawnow
end
```



Input Arguments

manipSV — Manipulator state validator
manipulatorCollisionBodyValidator object

Manipulator state validator, specified as a `manipulatorCollisionBodyValidator` object, which is a subclass of `nav.StateValidator`. The state validator contains properties that determine the behavior of this function and `isStateValid`.

startConfig — Initial robot configuration
n-element row vector of joint positions

Initial robot configuration, specified as an *n*-element row vector of joint positions for the `rigidBodyTree` robot model. *n* is the number of nonfixed joints in the robot model.

Data Types: `double`

goalConfig — Desired robot configuration
n-element row vector of joint positions

Desired robot configuration, specified as an *n*-element row vector of joint positions for the `rigidBodyTree` robot model. *n* is the number of nonfixed joints in the robot model.

Data Types: `double`

Output Arguments

`isValid` — Valid states
m-element logical column vector

Valid states, returned as an *m*-element logical column vector.

Data Types: `logical`

`lastValid` — Final valid state along each path
n-element row vector | *m*-by-*n* matrix

Final valid state along each path, returned as an *n*-element row vector or *m*-by-*n* matrix. *n* is the number of nonfixed joints in the robot model.. *m* is the number of paths validated. Each row contains the final valid state along the associated path.

Data Types: `single` | `double`

Version History

Introduced in R2021b

See Also

Objects
`rigidBodyTree` | `manipulatorStateSpace` | `workspaceGoalRegion` | `manipulatorRRT` | `manipulatorCollisionBodyValidator`

Functions
`isStateValid` | `sampleUniform` | `sampleGaussian` | `interpolate` | `distance` | `enforceStateBounds`

isStateValid

Check if state is valid

Syntax

```
isValid = isStateValid(manipSV,states)
```

Description

`isValid = isStateValid(manipSV,states)` checks if given states, or joint configurations, are valid for the rigid body tree robot model specified by the state validator `manipSV`. This object function checks for self-collisions and collisions with the environment based on the properties of the state validator.

Examples

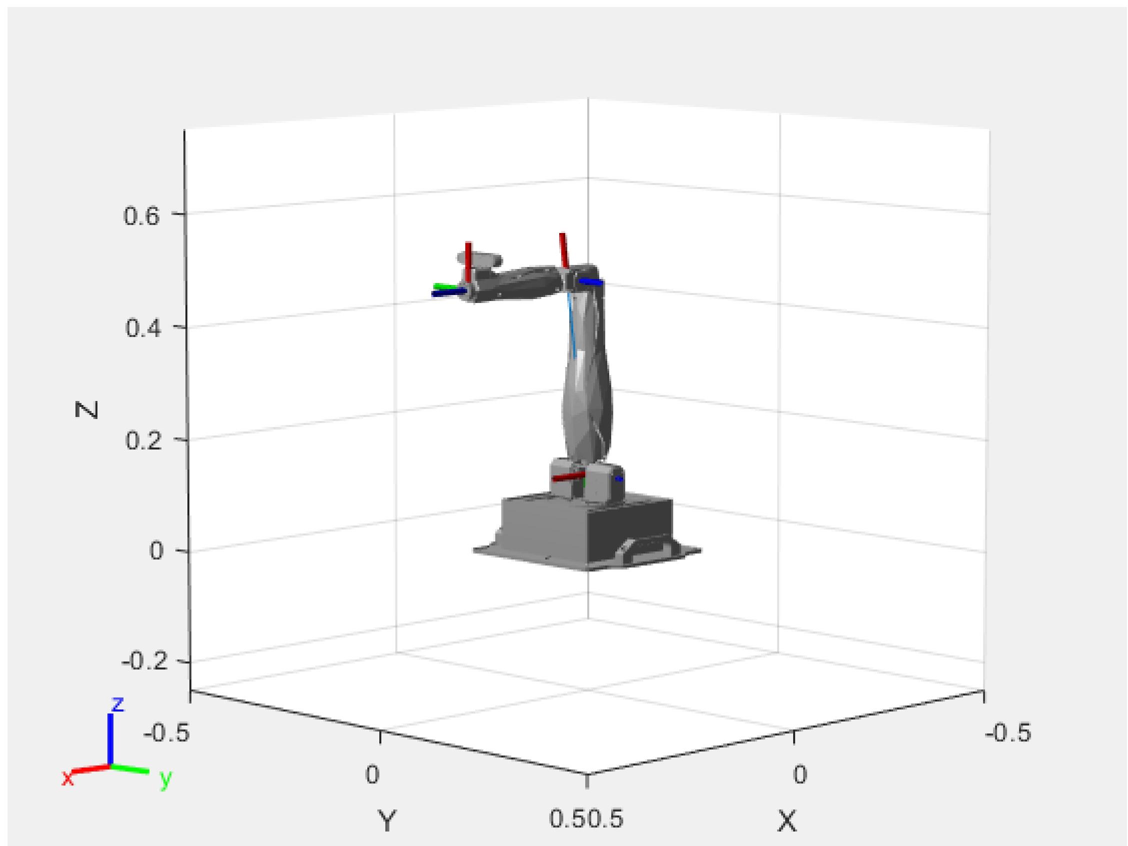
Validate State and Motion Manipulator State Space

Generate states to form a path, validate motion between states, and check for self-collisions and environmental collisions with objects in your world. The `manipulatorStateSpace` object represents the joint configuration space of your rigid body tree robot model, and can sample states, calculate distances, and enforce state bounds. The `manipulatorCollisionBodyValidator` object validates the state and motion based on the collision bodies in your robot model and any obstacles in your environment.

Load Robot Model

Use the `loadrobot` function to access predefined robot models. This example uses the Quanser QArm™ robot and joint configurations are specified as row vectors.

```
robot = loadrobot("quanserQArm",DataFormat="row");
figure(Visible="on")
show(robot);
xlim([-0.5 0.5])
ylim([-0.5 0.5])
zlim([-0.25 0.75])
hold on
```



Configure State Space and State Validation

Create the state space and state validator from the robot model.

```
ss = manipulatorStateSpace(robot);
sv = manipulatorCollisionBodyValidator(ss,SkippedSelfCollisions="parent");
```

Set the validation distance to `0.05`, which is based on the distance normal between two states. You can configure the validator to ignore self collisions to improve the speed of validation, but must consider whether your robot model has the proper joint limit settings set to ensure it does not collide with itself.

```
sv.ValidationDistance = 0.05;
sv.IgnoreSelfCollision = true;
```

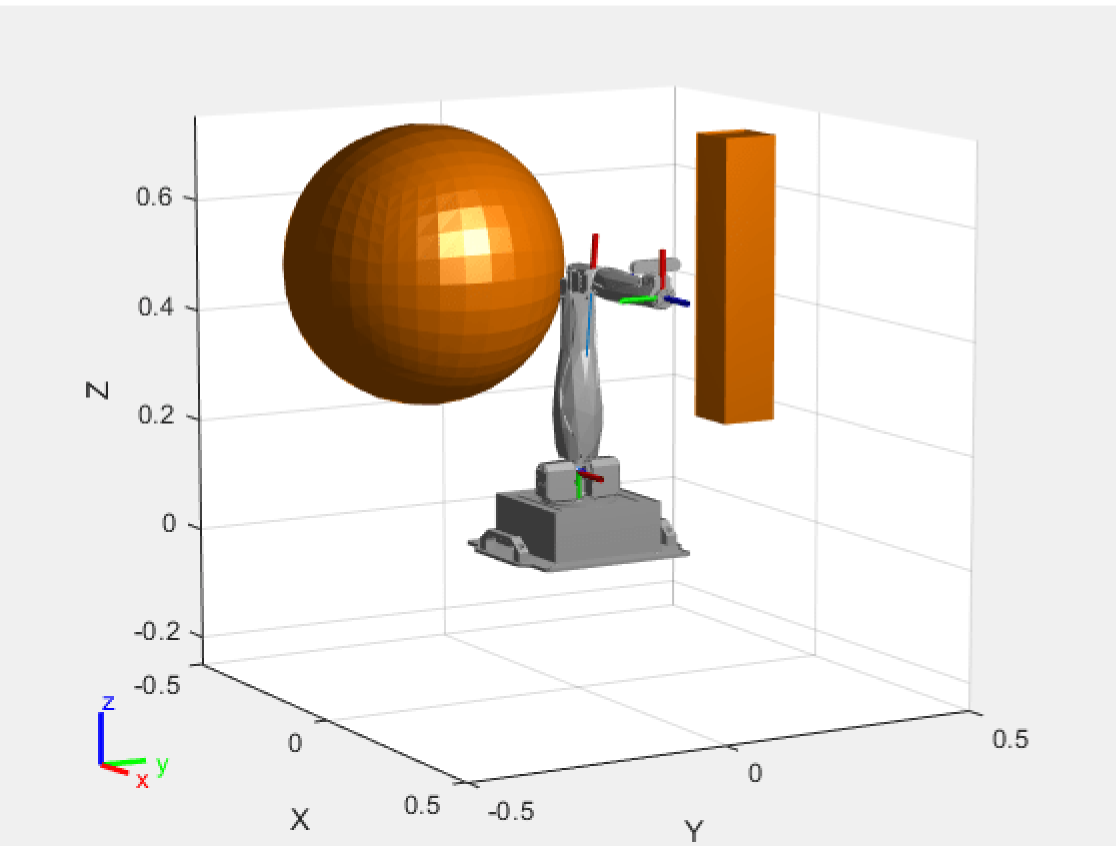
Place collision objects in the robot environment. Set the `Environment` property of the collision validator object using a cell array of objects.

```
box = collisionBox(0.1,0.1,0.5); % XYZ Lengths
box.Pose = trvec2tform([0.2 0.2 0.5]); % XYZ Position
sphere = collisionSphere(0.25); % Radius
sphere.Pose = trvec2tform([-0.2 -0.2 0.5]); % XYZ Position
env = {box sphere};
sv.Environment = env;
```

Visualize the environment.

```
for i = 1:length(env)
    show(env{i})
```

```
end
view(60,10)
```



Plan Path

Start with the home configuration as the first point on the path.

```
rng(0); % Repeatable results
start = homeConfiguration(robot);
path = start;
idx = 1;
```

Plan a path using these steps, in a loop:

- Sample a nearby goal configuration, using the Gaussian distribution, by specifying the standard deviation for each joint angle.
- Check if the sampled goal state is valid.
- If the sampled goal state is valid, check if the motion between states is valid and, if so, add it to the path.

```
for i = 2:25
    goal = sampleGaussian(ss,start,0.25*ones(4,1));
    validState = isStateValid(sv,goal);

    if validState % If state is valid, check motion between states.
        [validMotion,~] = isMotionValid(sv,path(idx,:),goal);
```

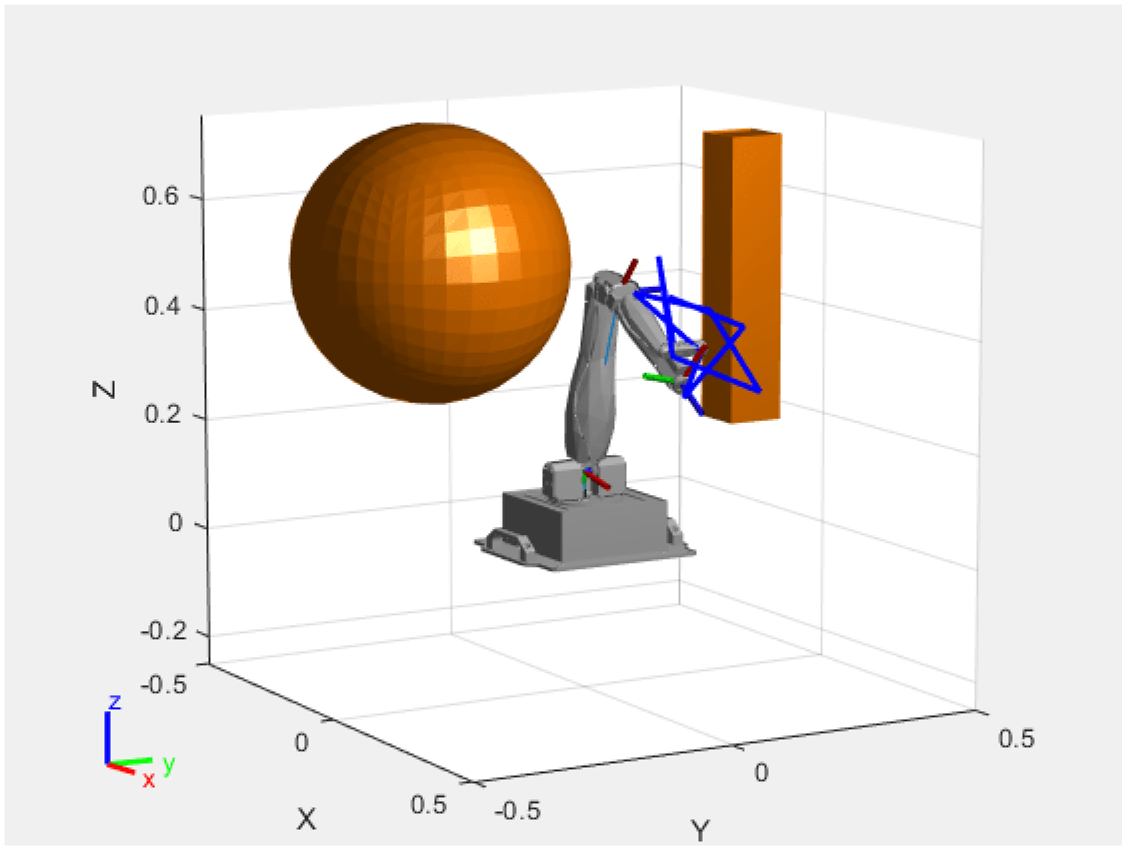
```
        if validMotion % If motion is valid, add to path.
            path = [path; goal];
            idx = idx + 1;
        end
    end
end
```

Visualize Path

After generating the path of valid motions, visualize the robot motion. Because you sampled random states near the home configuration, you should see the arm move around that initial configuration.

To visualize the path of the end effector in 3-D, get the transformation, relative to the base world frame at each point. Store the points as an *xyz* translation vector. Plot the path of the end effector.

```
eePose = nan(3,size(path,1));

for i = 1:size(path,1)
    show(robot,path(i,:),PreservePlot=false);
    eePos(i,:) = tform2trvec(getTransform(robot,path(i,:),"END-EFFECTOR")); % XYZ translation vec
    plot3(eePos(:,1),eePos(:,2),eePos(:,3),"-b",LineWidth=2)
    drawnow
end
```



Input Arguments

manipSV — Manipulator state validator
manipulatorCollisionBodyValidator object

Manipulator state validator, specified as a `manipulatorCollisionBodyValidator` object, which is a subclass of `nav.StateValidator`. The state validator contains properties that determine the behavior of this function and `isMotionValid`.

states — Robot states in joint space
m-by-*n* matrix

Robot states in the joint space, specified as an *m*-by-*n* matrix of joint positions. *m* is the total number of states and *n* is the number of nonfixed joints in the `rigidBodyTree` robot model.

Data Types: `double`

Output Arguments

isValid — Validity of state
m-element vector of logical scalars

Validity of input states returned as an *m*-element vector of logical scalars, where *m* is the total number of states. A state is valid, `1` or `true`, if it does not result in self-collisions, collisions with the environment, and it is within state bounds. Otherwise the state is invalid, `0` or `false`.

Data Types: `logical`

Version History

Introduced in R2021b

See Also

Objects

`rigidBodyTree` | `manipulatorStateSpace` | `workspaceGoalRegion` | `manipulatorRRT`

Functions

`isMotionValid` | `sampleUniform` | `sampleGaussian` | `interpolate` | `distance`

derivative

Time derivative of manipulator model states

Syntax

```
stateDot = derivative(taskMotionModel,state,refPose,refVel)
stateDot = derivative(taskMotionModel,state,refPose, refVel,fExt)
```

```
stateDot = derivative(jointMotionModel,state,cmds)
stateDot = derivative(jointMotionModel,state,cmds,fExt)
```

Description

`stateDot = derivative(taskMotionModel,state,refPose,refVel)` computes the time derivative of the motion model based on the current state and motion commands using a task-space model.

`stateDot = derivative(taskMotionModel,state,refPose, refVel,fExt)` computes the time derivative based on the current state, motion commands, and any external forces on the manipulator using a task space model.

`stateDot = derivative(jointMotionModel,state,cmds)` computes the time derivative of the motion model based on the current state and motion commands using a joint-space model.

`stateDot = derivative(jointMotionModel,state,cmds,fExt)` computes the time derivative based on the current state, motion commands, and any external forces on the manipulator using a joint-space model.

Examples

Create Joint-Space Motion Model

This example shows how to create and use a `jointSpaceMotionModel` object for a manipulator robot in joint-space.

Create the Robot

```
robot = loadrobot("kinovaGen3","DataFormat","column","Gravity",[0 0 -9.81]);
```

Set Up the Simulation

Set the timespan to be 1 s with a timestep size of 0.01 s. Set the initial state to be the robots, home configuration with a velocity of zero.

```
tspan = 0:0.01:1;
initialState = [homeConfiguration(robot); zeros(7,1)];
```

Define the a reference state with a target position, zero velocity, and zero acceleration.

```
targetState = [pi/4; pi/3; pi/2; -pi/3; pi/4; -pi/4; 3*pi/4; zeros(7,1); zeros(7,1)];
```

Create the Motion Model

Model the system with computed torque control and error dynamics defined by a moderately fast step response with 5% overshoot.

```
motionModel = jointSpaceMotionModel("RigidBodyTree",robot);
updateErrorDynamicsFromStep(motionModel,.3,.05);
```

Simulate the Robot

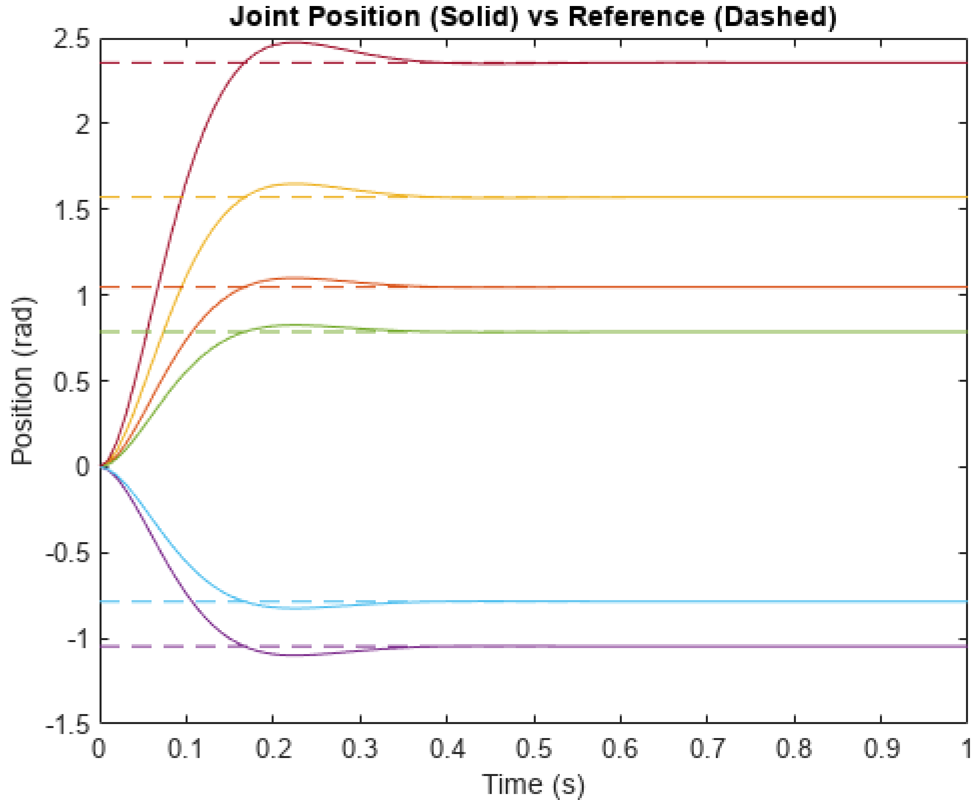
Use the derivative function of the model as the input to the `ode45` solver to simulate the behavior over 1 second.

```
[t,robotState] = ode45(@(t,state)derivative(motionModel,state,targetState),tspan,initialState);
```

Plot the Response

Plot the positions of all the joints actuating to their target state. Joints with a higher displacement between the starting position and the target position actuate to the target at a faster rate than those with a lower displacement. This leads to an overshoot, but all of the joints have the same settling time.

```
figure
plot(t,robotState(:,1:motionModel.NumJoints));
hold all;
plot(t,targetState(1:motionModel.NumJoints)*ones(1,length(t)),"--");
title("Joint Position (Solid) vs Reference (Dashed)");
xlabel("Time (s)")
ylabel("Position (rad)");
```



Create Task-Space Motion Model

This example shows how to create and use a `taskSpaceMotionModel` object for a manipulator robot arm in task-space.

Create the Robot

```
robot = loadrobot("kinovaGen3","DataFormat","column","Gravity",[0 0 -9.81]);
```

Set Up the Simulation

Set the time span to be 1 second with a timestep size of 0.02 seconds. Set the initial state to the home configuration of the robot, with a velocity of zero.

```
tspan = 0:0.02:1;
initialState = [homeConfiguration(robot);zeros(7,1)];
```

Define a reference state with a target position and zero velocity.

```
refPose = trvec2tform([0.6 -.1 0.5]);
refVel = zeros(6,1);
```

Create the Motion Model

Model the behavior as a system under proportional-derivative (PD) control.

```
motionModel = taskSpaceMotionModel("RigidBodyTree",robot,"EndEffectorName","EndEffector_Link");
```

Simulate the Robot

Simulate the behavior over 1 second using a stiff solver to more efficiently capture the robot dynamics. Using `ode15s` enables higher precision around the areas with a high rate of change.

```
[t,robotState] = ode15s(@(t,state)derivative(motionModel,state,refPose,refVel),tspan,initialState
```

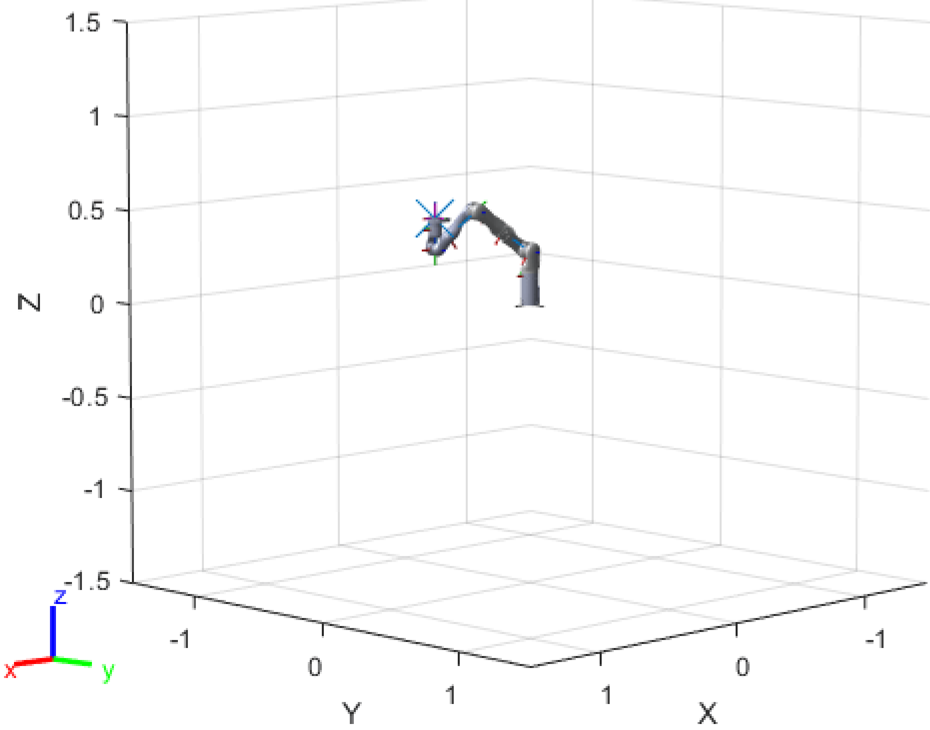
Plot the Response

Plot the robot's initial position and mark the target with an X.

```
figure
show(robot,initialState(1:7));
hold all
plot3(refPose(1,4),refPose(2,4),refPose(3,4),"x","MarkerSize",20)
```

Observe the response by plotting the robot in a 5 Hz loop.

```
r = rateControl(5);
for i = 1:size(robotState,1)
    show(robot,robotState(i,1:7)',"PreservePlot",false);
    waitfor(r);
end
```



Input Arguments

`taskMotionModel` — `taskSpaceMotionModel` object
`taskSpaceMotionModel` object

`taskSpaceMotionModel` object, which defines the properties of the motion model.

`jointMotionModel` — `jointSpaceMotionModel` object
`jointSpaceMotionModel` object

`jointSpaceMotionModel` object, which defines the properties of the motion model.

`state` — Joint positions and velocities
1-by-2*n*-element vector

Joint positions and velocities represented as a 2*n*-element vector, specified as [*q; qDot*]. *n* is the number of non-fixed joints in the associated `rigidBodyTree` of the `motionModel`. *q*, represents the position of each joint, specified in radians. *qDot* represents the velocity of each joint, specified in radians per second.

`refPose` — Robot pose
4-by-4 matrix

The reference pose of the end effector in the task-space in meters, specified as an 4-by-4 homogeneous transformation matrix.

`refVel` — Joint velocities
six-element row vector

The reference velocities of the end effector in the task space, specified as a six-element vector of real values, specified as [*omega v*]. *omega* represents a row vector of three angular velocities about the `x`, `y`, and `z` axes, specified in radians per second, and *v* represents a row vector of three linear velocities along the `x`, `y`, and `z` axes, specified in meters per second.

`cmds` — Control commands indicating desired motion
2-by-*n* matrix | 3-by-*n* matrix

Control commands indicating desired motion. The dimensions of `cmds` depend on the MotionType property of the motion model:

- `"PDControl"` — 2-by-*n* matrix, `[qRef; qRefDot]`. The first and second rows represent joint positions and joint velocities, respectively.
- `"ComputedTorqueControl"` — 3-by-*n* matrix, `[qRef; qRefDot; qRefDDot]`. The first, second, and third rows represent joint positions, joint velocities, and joint accelerations respectively.
- `"IndependentJointMotion"` — 3-by-*n* matrix, `[qRef; qRefDot; qRefDDot]`. The first, second, and third rows represent joint positions, joint velocities, and joint accelerations respectively.

Note that `jointSpaceMotionModel` supports all three MotionType listed above, but `taskSpaceMotionModel` only supports `"PDControl"` MotionType.

`fExt` — Joint positions and velocities
m-element vector

External forces, specified as an *m*-element vector, where *m* is the number of bodies in the associated `rigidBodyTree` object.

Output Arguments

`stateDot` — Time derivative of current state
2-by-*n* matrix

Time derivative based on current state and specified control commands, returned as a 2-by-*n* matrix of real values, [*qDot; qDDot*], where *qDot* is an *n*-element row vector of joint velocities, and *qDDot* is an *n*-element row vector of joint accelerations. *n* is the number of joints in the associated `rigidBodyTree` of the `motionModel`.

Version History

Introduced in R2019b

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

Classes
`jointSpaceMotionModel` | `taskSpaceMotionModel`

interpolate

Interpolate states along path from RRT

Syntax

```
interpPath = interpolate(rrt,path)
interpPath = interpolate(rrt,path,numInterp)
```

Description

`interpPath = interpolate(rrt,path)` interpolates states between each adjacent configuration in the path based on the ValidationDistance property of the manipulator rapidly exploring random tree (RRT) planner `rrt`.

`interpPath = interpolate(rrt,path,numInterp)` specifies the number of interpolations between adjacent configurations.

Examples

Plan Path for Manipulator Robot Using RRT

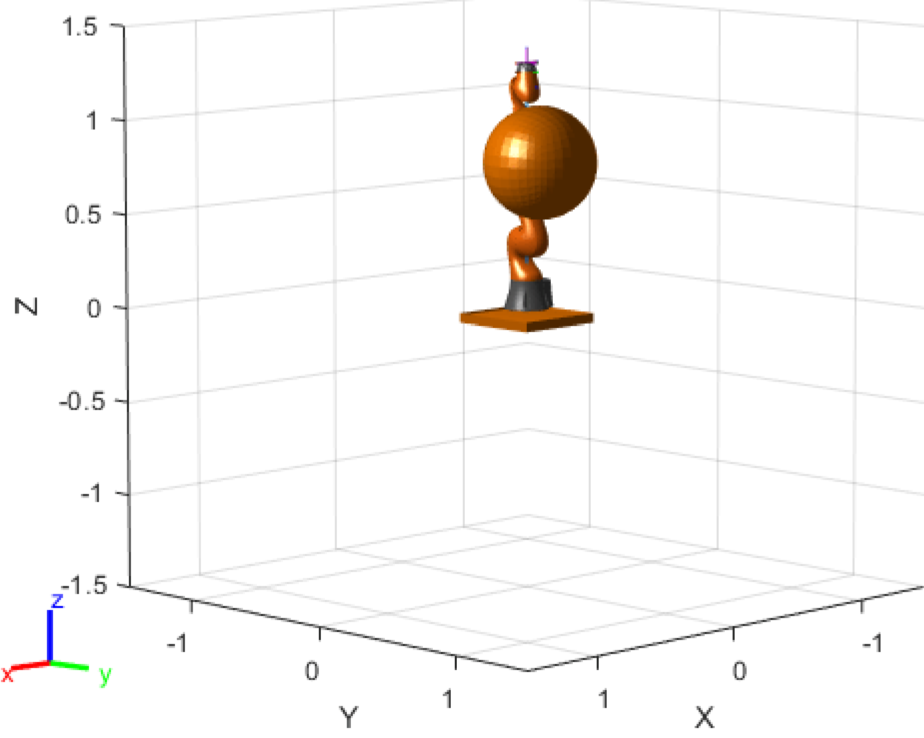
Use the `manipulatorRRT` object to plan a path for a rigid body tree robot model in an environment with obstacles. Visualize the planned path with interpolated states.

Load a robot model into the workspace. Use the KUKA LBR iiwa© manipulator arm.

```
robot = loadrobot("kukaIiwa14","DataFormat","row");
```

Generate the environment for the robot. Create collision objects and specify their poses relative to the robot base. Visualize the environment.

```
env = {collisionBox(0.5, 0.5, 0.05) collisionSphere(0.3)};
env{1}.Pose(3, end) = -0.05;
env{2}.Pose(1:3, end) = [0.1 0.2 0.8];

show(robot);
hold on
show(env{1})
show(env{2})
```



Create the RRT planner for the robot model.

```
rrt = manipulatorRRT(robot,env);
rrt.SkippedSelfCollisions = "parent";
```

Specify a start and a goal configuration.

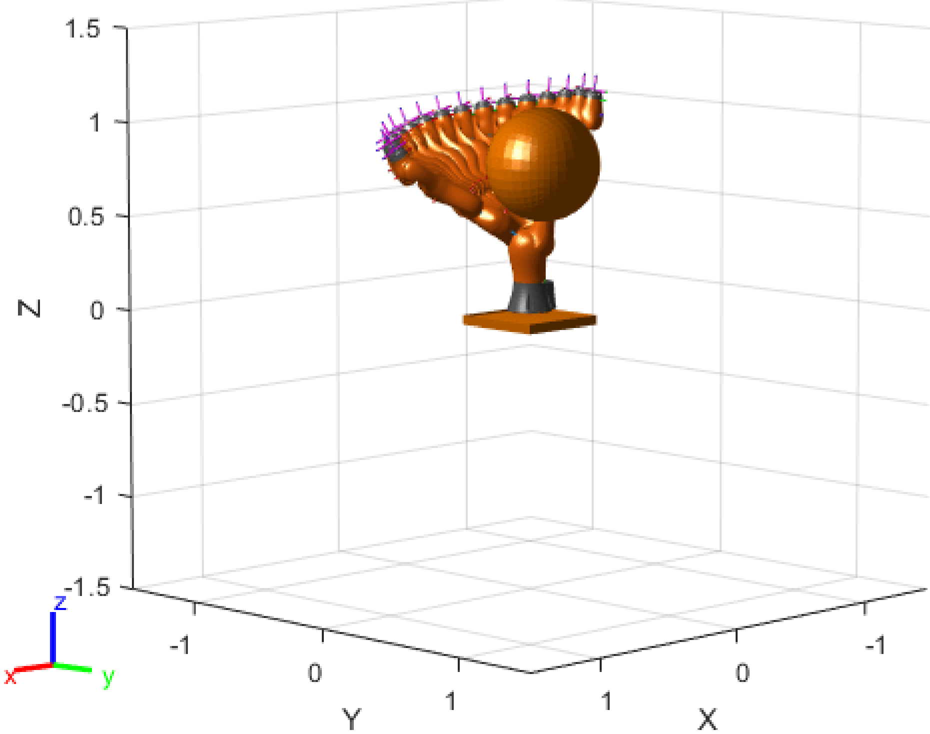
```
startConfig = [0.08 -0.65 0.05 0.02 0.04 0.49 0.04];
goalConfig =  [2.97 -1.05 0.05 0.02 0.04 0.49 0.04];
```

Plan the path. Due to the randomness of the RRT algorithm, set the `rng` seed for repeatability.

```
rng(0)
path = plan(rrt,startConfig,goalConfig);
```

Visualize the path. To add more intermediate states, interpolate the path. By default, the `interpolate` object function uses the value of `ValidationDistance` property to determine the number of intermediate states. The `for` loop shows every 20th element of the interpolated path.

```
interpPath = interpolate(rrt,path);
clf
for i = 1:20:size(interpPath,1)
    show(robot,interpPath(i,:));
    hold on
end
show(env{1})
show(env{2})
hold off
```



Input Arguments

rrt — Manipulator RRT planner
manipulatorRRT object

Manipulator RRT planner, specified as a `manipulatorRRT` object. This planner is for a specific rigid body tree robot model stored as a `rigidBodyTree` object.

path — Planned path in joint space
r-by-*n* matrix of joint configurations

Planned path in joint space, specified as an *r*-by-*n* matrix of joint configurations, where *r* is the number of configurations in the path, and *n* is the number of nonfixed joints in the `rigidBodyTree` robot model.

Data Types: `double`

numInterp — Number of interpolations between each configuration
positive integer

Number of interpolations between each configuration, specified as a positive integer.

Data Types: `double`

Output Arguments

`interpPath` — Interpolated path in joint space
r-by-*n* matrix of joint configurations

Planned path in joint space, specified as an *r*-by-*n* matrix of joint configurations, where *r* is the number of configurations in the path and *n* is the number of nonfixed joints in the `rigidBodyTree` robot model.

Data Types: `double`

Version History

Introduced in R2020b

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

Objects
`manipulatorRRT` | `rigidBodyTree` | `interactiveRigidBodyTree` | `analyticalInverseKinematics`

Functions
`plan` | `shorten`

Topics
"Pick and Place Using RRT for Manipulators"
"Pick-and-Place Workflow Using RRT Planner and Stateflow for MATLAB"

plan

Plan path using RRT for manipulators

Syntax

```
path = plan(rrt,startConfig,goalConfig)
path = plan(rrt,startConfig,goalRegion)
[path,solnInfo] = plan( ___ )
```

Description

`path = plan(rrt,startConfig,goalConfig)` plans a path between the specified start and goal configurations using the manipulator rapidly exploring random trees (RRT) planner `rrt`.

`path = plan(rrt,startConfig,goalRegion)` plans a path between the specified start and a goal region as a `workspaceGoalRegion` object

`[path,solnInfo] = plan( ___ )` also returns solution info about the results from the RRT planner using the previous input arguments.

Examples

Plan Path for Manipulator Robot Using RRT

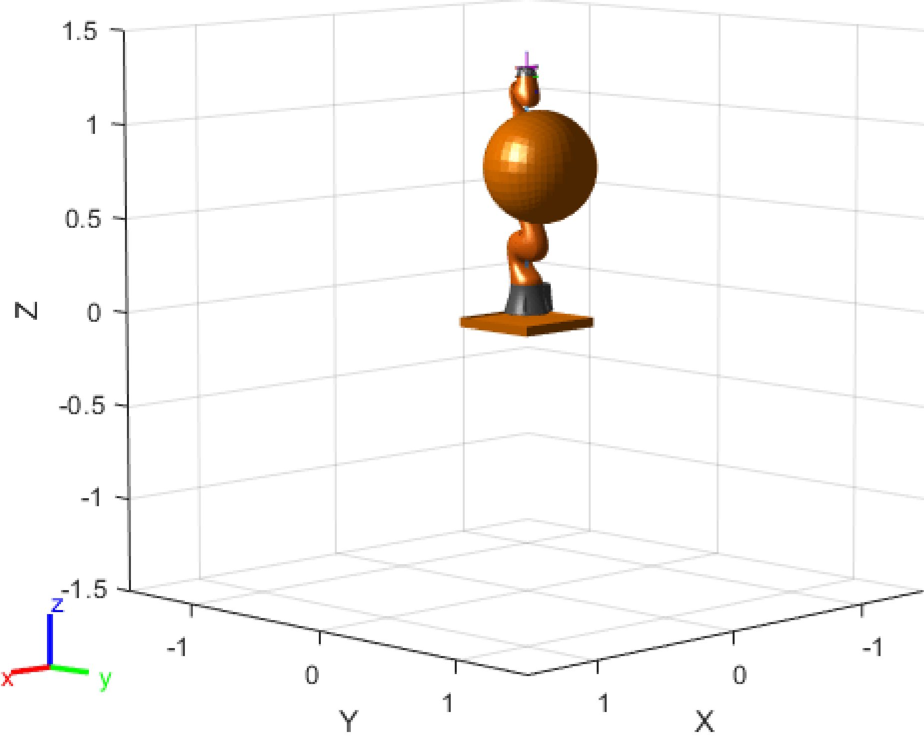
Use the `manipulatorRRT` object to plan a path for a rigid body tree robot model in an environment with obstacles. Visualize the planned path with interpolated states.

Load a robot model into the workspace. Use the KUKA LBR iiwa© manipulator arm.

```
robot = loadrobot("kukaIiwa14","DataFormat","row");
```

Generate the environment for the robot. Create collision objects and specify their poses relative to the robot base. Visualize the environment.

```
env = {collisionBox(0.5, 0.5, 0.05) collisionSphere(0.3)};
env{1}.Pose(3, end) = -0.05;
env{2}.Pose(1:3, end) = [0.1 0.2 0.8];

show(robot);
hold on
show(env{1})
show(env{2})
```



Create the RRT planner for the robot model.

```
rrt = manipulatorRRT(robot,env);
rrt.SkippedSelfCollisions = "parent";
```

Specify a start and a goal configuration.

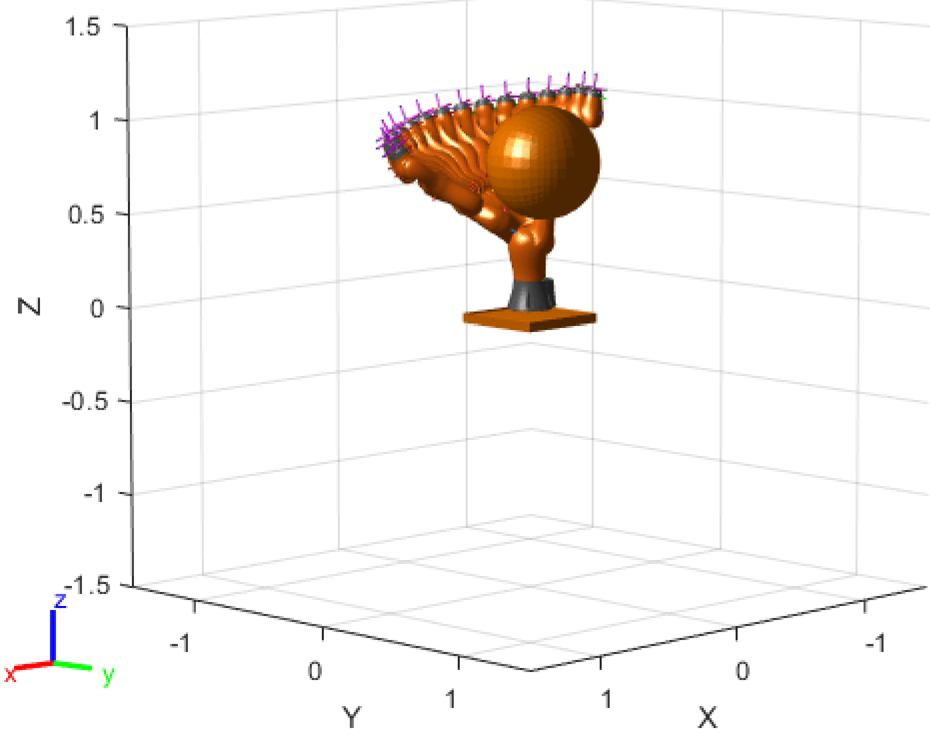
```
startConfig = [0.08 -0.65 0.05 0.02 0.04 0.49 0.04];
goalConfig =  [2.97 -1.05 0.05 0.02 0.04 0.49 0.04];
```

Plan the path. Due to the randomness of the RRT algorithm, set the `rng` seed for repeatability.

```
rng(0)
path = plan(rrt,startConfig,goalConfig);
```

Visualize the path. To add more intermediate states, interpolate the path. By default, the `interpolate` object function uses the value of `ValidationDistance` property to determine the number of intermediate states. The `for` loop shows every 20th element of the interpolated path.

```
interpPath = interpolate(rrt,path);
clf
for i = 1:20:size(interpPath,1)
    show(robot,interpPath(i,:));
    hold on
end
show(env{1})
show(env{2})
hold off
```

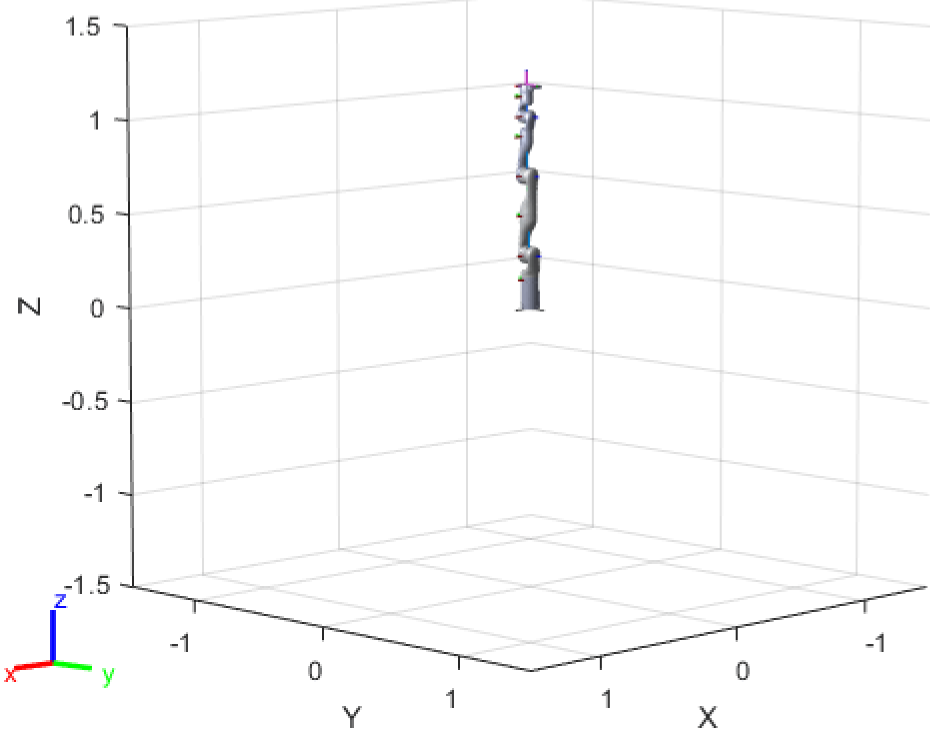


Plan Path To Workspace Goal Region

Specify a goal region in your workspace and plan a path within those bounds. The `workspaceGoalRegion` object defines the bounds on the *xyz*-position and *zyx* Euler orientation of the robot end effector. The `manipulatorRRT` object plans a path based on that goal region and samples random poses within the bounds.

Load an existing robot model as a `rigidBodyTree` object.

```
robot = loadrobot("kinovaGen3", "DataFormat", "row");
ax = show(robot);
```



Create Path Planner

Create a rapidly-exploring random tree (RRT) path planner for the robot. This example uses an empty environment, but this workflow also works well with cluttered environments. You can add collision objects to the environment like the `collisionBox` or `collisionMesh` object.

```
planner = manipulatorRRT(robot,{});
planner.SkippedSelfCollisions="parent";
```

Define Goal Region

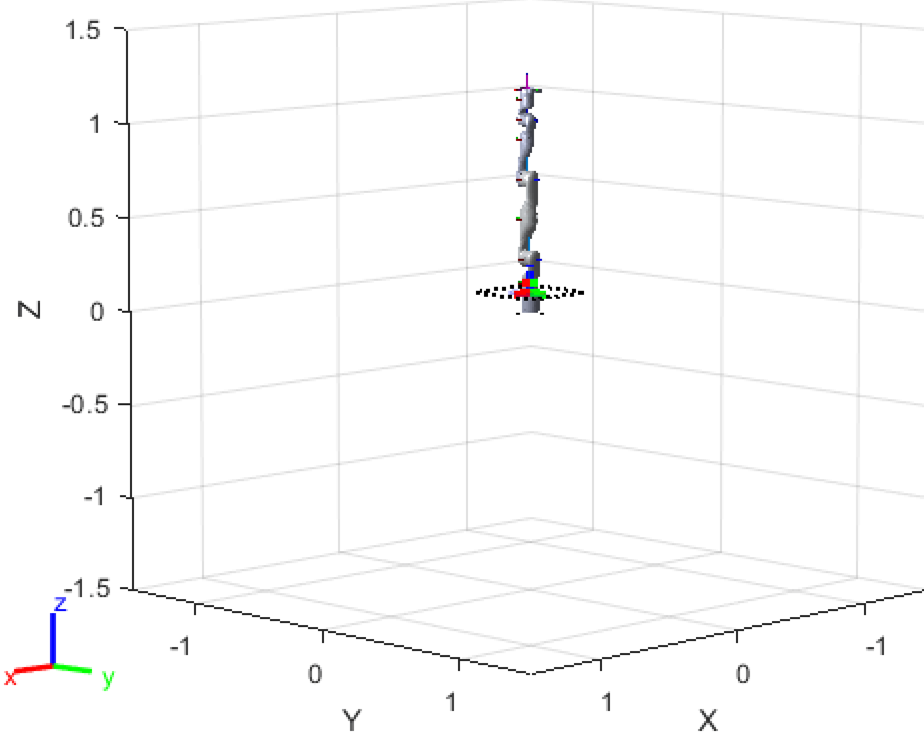
Create a workspace goal region using the end-effector body name of the robot.

Define the goal region parameters for your workspace. The goal region includes a reference pose, *xyz*-position bounds, and orientation limits on the *zyx* Euler angles. This example specifies bounds on the *xy*-plane in meters and allows rotation about the *z*-axis in radians.

```
goalRegion = workspaceGoalRegion(robot.BodyNames{end});
goalRegion.ReferencePose = trvec2tform([0.5 0.5 0.2]);
goalRegion.Bounds(1, :) = [-0.2 0.2];    % X Bounds
goalRegion.Bounds(2, :) = [-0.2 0.2];    % Y Bounds
goalRegion.Bounds(4, :) = [-pi/2 pi/2];  % Rotation about the Z-axis
```

You can also apply a fixed offset to all poses sampled within the region. This offset can account for grasping tools or variations in dimensions within your workspace. For this example, apply a fixed transformation that places the end effector 5 cm above the workspace.

```
goalRegion.EndEffectorOffsetPose = trvec2tform([0 0 0.05]);
hold on
show(goalRegion);
```



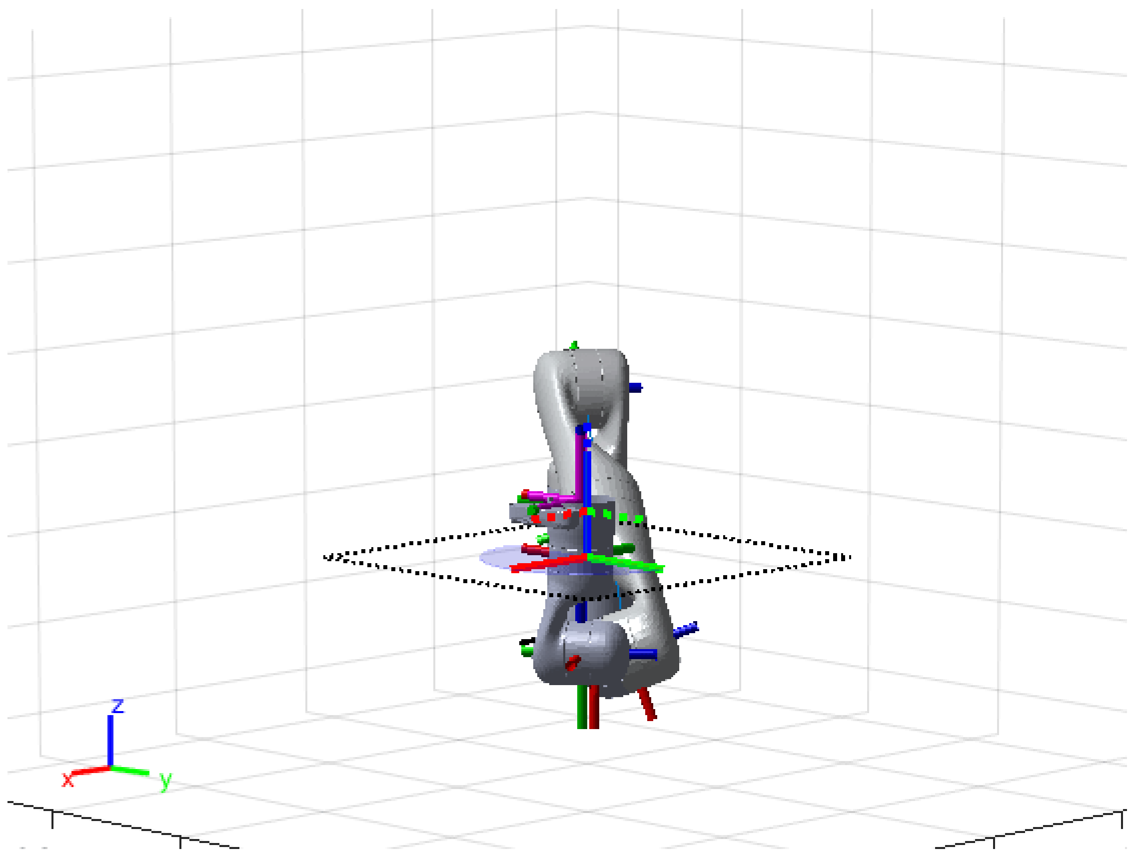
Plan Path To Goal Region

Plan a path to the goal region from the robot's home configuration. Due to the randomness in the RRT algorithm, this example sets the `rng` seed to ensure repeatable results.

```
rng(0)
path = plan(planner,homeConfiguration(robot),goalRegion);
```

Show the robot executing the path. To visualize a more realistic path, interpolate points between path configurations.

```
interpConfigurations = interpolate(planner,path,5);

for i = 1 : size(interpConfigurations)
    show(robot,interpConfigurations(i,:),"PreservePlot",false);
    set(ax,'ZLim',[-0.05 0.75],'YLim',[-0.05 1],'XLim',[-0.05 1],...
        'CameraViewAngle',5)

    drawnow
end
hold off
```



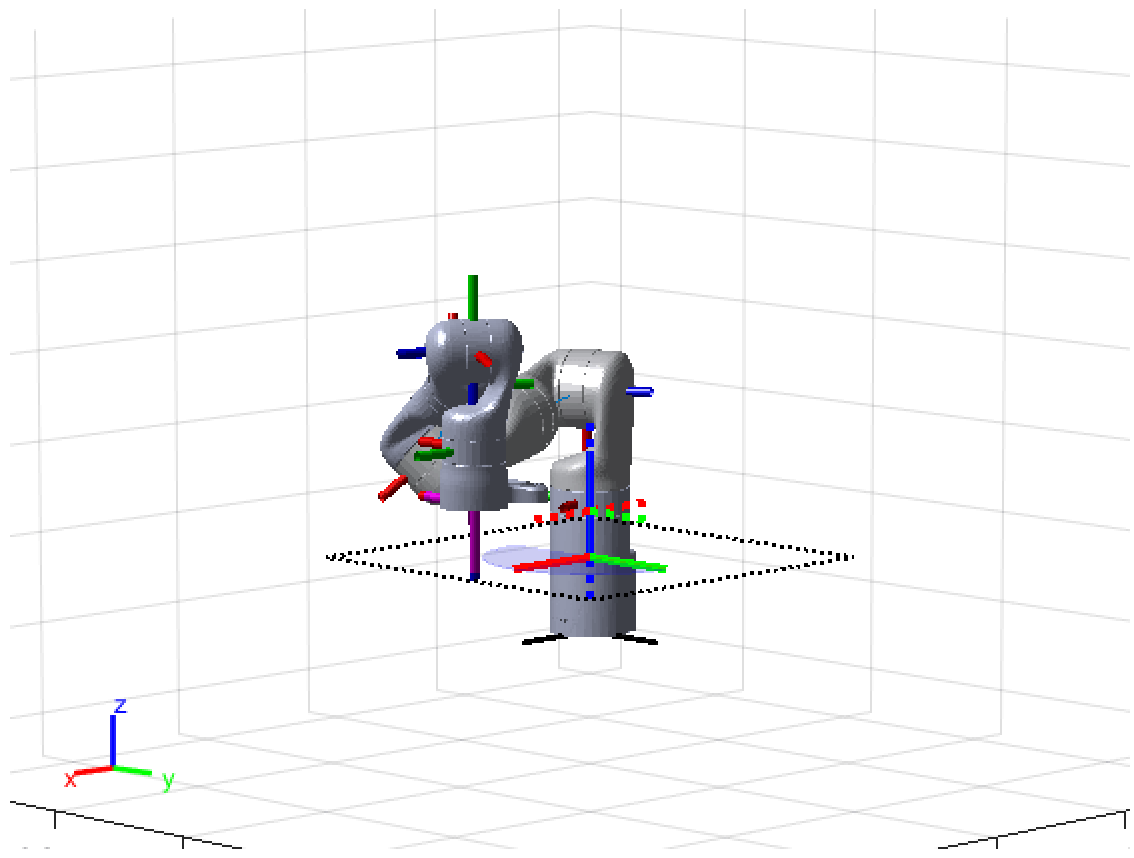
Adjust End-Effector Pose

Notice that the robot arm approaches the workspace from the bottom. To flip the orientation of the final position, add a `pi` rotation to the Y-axis for the reference pose.

```
goalRegion.EndEffectorOffsetPose = ...
    goalRegion.EndEffectorOffsetPose*eul2tform([0 pi 0],"ZYX");
```

Replan the path and visualize the robot motion again. The robot now approaches from the top.

```
hold on
show(goalRegion);
path = plan(planner,homeConfiguration(robot),goalRegion);

interpConfigurations = interpolate(planner,path,5);

for i = 1 : size(interpConfigurations)
    show(robot, interpConfigurations(i, :),"PreservePlot",false);
    set(ax,'ZLim',[-0.05 0.75],'YLim',[-0.05 1],'XLim',[-0.05 1])
    drawnow;
end
hold off
```



Input Arguments

rrt — Manipulator RRT planner
manipulatorRRT object

Manipulator RRT planner, specified as a manipulatorRRT object. This planner is for a specific rigid body tree robot model stored as a rigidBodyTree object.

startConfig — Initial robot configuration
n-element vector of joint positions

Initial robot configuration, specified as an *n*-element vector of joint positions for the rigidBodyTree object stored in the RRT planner rrt. *n* is the number of nonfixed joints in the robot model.

Data Types: double

goalConfig — Desired robot configuration
n-element vector of joint positions

Desired robot configuration, specified as an *n*-element vector of joint positions for the rigidBodyTree object stored in the RRT planner rrt. *n* is the number of nonfixed joints in the robot model.

Data Types: double

goalRegion — Workspace goal region
workspaceGoalRegion object

Workspace goal region, specified as a `workspaceGoalRegion` object.

The `workspaceGoalRegion` object defines the bounds on the end-effector pose and the `sample` object function returns random poses to add to the RRT tree. Set the WorkspaceGoalRegionBias property to change the probability of sampling a state within the goal region.

Output Arguments

`path` — Planned path in joint space
r-by-*n* matrix of joint configurations

Planned path in joint space, returned as an *r*-by-*n* matrix of joint configurations, where *r* is the number of configurations in the path, and *n* is the number of nonfixed joints in the `rigidBodyTree` robot model.

Data Types: `double`

`solnInfo` — Solution information from planner
structure

Solution information from planner, returned as a structure with these fields:

- `IsPathFound` — A logical indicating if a path was found
- `ExitFlag` — An integer indicating why the planner terminated:
  - `1` — Goal configuration reached
  - `2` — Maximum number of iterations reached

Data Types: `struct`

Version History

Introduced in R2020b

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

Objects
`manipulatorRRT` | `rigidBodyTree` | `interactiveRigidBodyTree` | `analyticalInverseKinematics`

Functions
`interpolate` | `shorten`

Topics
"Pick and Place Using RRT for Manipulators"
"Pick-and-Place Workflow Using RRT Planner and Stateflow for MATLAB"

shorten

Trim edges to shorten path from RRT

Syntax

```
shortPath = shorten(rrt,path,numIter)
```

Description

`shortPath = shorten(rrt,path,numIter)` trims edges to shorten the specified path `path` by running a randomized shortening strategy for a specified number of iterations `numIter`.

Input Arguments

`rrt` — Manipulator RRT planner
`manipulatorRRT` object

Manipulator RRT planner, specified as a `manipulatorRRT` object. This planner is for a specific rigid body tree robot model stored as a `rigidBodyTree` object.

`path` — Planned path in joint space
r-by-*n* matrix of joint configurations

Planned path in joint space, specified as an *r*-by-*n* matrix of joint configurations, where *r* is the number of configurations in the path, and *n* is the number of nonfixed joints in the `rigidBodyTree` robot model.

Data Types: `double`

`numIter` — Number of iterations to attempt shortening the path
positive integer

Number of iterations to attempt shortening path, specified as a positive integer.

Data Types: `double`

Output Arguments

`shortPath` — Planned path in joint space
r-by-*n* matrix of joint configurations

Planned path in joint space, returned as an *r*-by-*n* matrix of joint configurations, where *r* is the number of configurations in the path, and *n* is the number of nonfixed joints in the `rigidBodyTree` robot model.

Data Types: `double`

Version History

Introduced in R2020b

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

Objects
`manipulatorRRT` | `rigidBodyTree` | `interactiveRigidBodyTree` | `analyticalInverseKinematics`

Functions
`plan` | `interpolate`

Topics
"Pick and Place Using RRT for Manipulators"
"Pick-and-Place Workflow Using RRT Planner and Stateflow for MATLAB"

distance

Distance between states

Syntax

```
dist = distance(manipSS,state1,state2)
```

Description

`dist = distance(manipSS,state1,state2)` calculates the distance between one or more initial states and one more final states.

Input Arguments

`manipSS` — Manipulator state space

`manipulatorStateSpace` object

Manipulator state space, specified as a `manipulatorStateSpace` object, which is a subclass of `nav.StateSpace`.

`state1` — Initial state position

n-element row vector | *m*-by-*n* matrix

Initial state position, specified as an*n*-element row vector or *m*-by-*n* matrix. *n* is the dimension of the state space specified in the `NumStateVariables` property of `manipSS`. *m* is the number of initial state positions.

The sizes of `state1` and `state2` determine the size of the `dist` output:

State Vectors and Distances

`state1` Size	`state2` Size	`dist` Size
n-element row vector	*n*-element row vector	scalar
n-element row vector	*m*-by-*n* matrix	*m*-element column vector
m-by-*n* matrix	*n*-element row vector	*m*-element column vector
m-by-*n* matrix	*m*-by-*n* matrix	*m*-element column vector

`state2` — Final state position

n-element vector | *m*-by-*n* matrix of row vectors

Final state position, specified as a *n*-element row vector or *m*-by-*n* matrix. *n* is the dimension of the state space specified in the `NumStateVariables` property of `manipSS`. *m* is the number of initial state positions.

The sizes of `state1` and `state2` determine the size of the `dist` output:

State Vectors and Distances

`state1` Size	`state2` Size	`dist` Size
n-element row vector	*n*-element row vector	scalar
n-element row vector	*m*-by-*n* matrix	*m*-element column vector
m-by-*n* matrix	*n*-element row vector	*m*-element column vector
m-by-*n* matrix	*m*-by-*n* matrix	*m*-element column vector

Output Arguments

`dist` — Distance between two states

numeric scalar | *m*-element column vector

Distance between two states, returned as a numeric scalar or *m*-element column vector. This distance calculation is the main component in evaluating the costs of paths. For prismatic joints, the distance between two states is the Euclidean norm of the difference between the state vectors. For revolute joints with infinite bounds, the difference in joint values is calculated using `angdiff`.

For revolute joints, distances measure joint differences in radians. For prismatic joints, distances measure displacement in meters.

The sizes of `state1` and `state2` determine the size of output `dist`:

State Vectors and Distances

`state1` Size	`state2` Size	`dist` Size
n-element row vector	*n*-element row vector	scalar
n-element row vector	*m*-by-*n* matrix	*m*-element column vector
m-by-*n* matrix	*n*-element row vector	*m*-element column vector
m-by-*n* matrix	*m*-by-*n* matrix	*m*-element column vector

Version History

Introduced in R2021b

See Also

`nav.StateSpace` | `nav.StateValidator` | `stateSpaceSE2` | `stateSpaceDubins` | `stateSpaceReedsShepp`

Topics

"Create Custom State Space for Path Planning" (Navigation Toolbox)

enforceStateBounds

Limit state to state space bounds

Syntax

```
boundedState = enforceStateBounds(manipSS,state)
```

Description

`boundedState = enforceStateBounds(manipSS,state)` limits the specified `state` to the bounds specified by the `StateBounds` property of the state space object, `manipSS`, and returns the bounded state, `boundedState`.

Input Arguments

`manipSS` — Manipulator state space
`manipulatorStateSpace` object

Manipulator state space, specified as a `manipulatorStateSpace` object, which is a subclass of `nav.StateSpace`.

`state` — State position
n-element row vector | *m*-by-*n* matrix

State position, specified as an *n*-element row vector or an *m*-by-*n* matrix. *n* is the dimension of the state space specified in the `NumStateVariables` property of `manipSS`. *m* is the number of initial state positions.

Output Arguments

`boundedState` — State position with enforced state bounds
n-element row vector | *m*-by-*n* matrix

State position with enforced state bounds, returned as an *n*-element vector or *m*-by-*n* matrix. *n* is the dimension of the state space specified in the `NumStateVariables` property of `manipSS`. *m* is the number of initial state positions.

Version History

Introduced in R2021b

See Also

Objects
`manipulatorStateSpace` | `rigidBodyTree` | `manipulatorCollisionBodyValidator` | `manipulatorRRT` | `workspaceGoalRegion`

Functions
`isStateValid` | `isMotionValid` | `sampleUniform` | `sampleGaussian` | `interpolate` | `distance`

interpolate

Interpolate between states

Syntax

```
interpStates = interpolate(manipSS,state1,state2,ratios)
```

Description

`interpStates = interpolate(manipSS,state1,state2,ratios)` interpolates between two states in your state space using the specified ratio values `ratios`.

Input Arguments

`manipSS` — Manipulator state space
`manipulatorStateSpace` object

Manipulator state space, specified as a `manipulatorStateSpace` object, which is a subclass of `nav.StateSpace`.

`state1` — Initial state position
n-element row vector

Initial state position, specified as an *n*-element row vector. *n* is the dimension of the state space specified in the `NumStateVariables` property of `manipSS`.

`state2` — Final state position
n-element row vector

Final state position, specified as an *n*-element row vector. *n* is the dimension of the state space specified in the `NumStateVariables` property of `manipSS`.

`ratios` — Ratio values for interpolating along path
m-element vector values in range [0 1]

Ratio values for interpolating along the path, specified as an *m*-element vector of values in range [0 1]. These ratios determine the distance of the interpolated state from `state1`.

Output Arguments

`interpStates` — Interpolated states
m-by-*n* matrix

Interpolated states, returned as an *m*-by-*n* matrix. *m* is the length of `ratios` and *n* is the dimension of the state space specified in the `NumStateVariables` property of `manipSS`.

Version History

Introduced in R2021b

See Also

Objects
`manipulatorStateSpace` | `rigidBodyTree` | `manipulatorCollisionBodyValidator` | `manipulatorRRT` | `workspaceGoalRegion`

Functions
`isStateValid` | `isMotionValid` | `sampleUniform` | `sampleGaussian` | `enforceStateBounds` | `distance`

sampleGaussian

Sample state using Gaussian distribution

Syntax

```
states = sampleGaussian(manipSS,meanState,stdDev)
states = sampleGaussian(manipSS,meanState,stdDev,numSamples)
```

Description

`states = sampleGaussian(manipSS,meanState,stdDev)` samples a state in the state space `manipSS` from a Gaussian (normal) distribution centered on the mean `meanState` with the standard deviation, `stdDev`.

`states = sampleGaussian(manipSS,meanState,stdDev,numSamples)` samples the number of multiple states specified by `numSamples`.

Examples

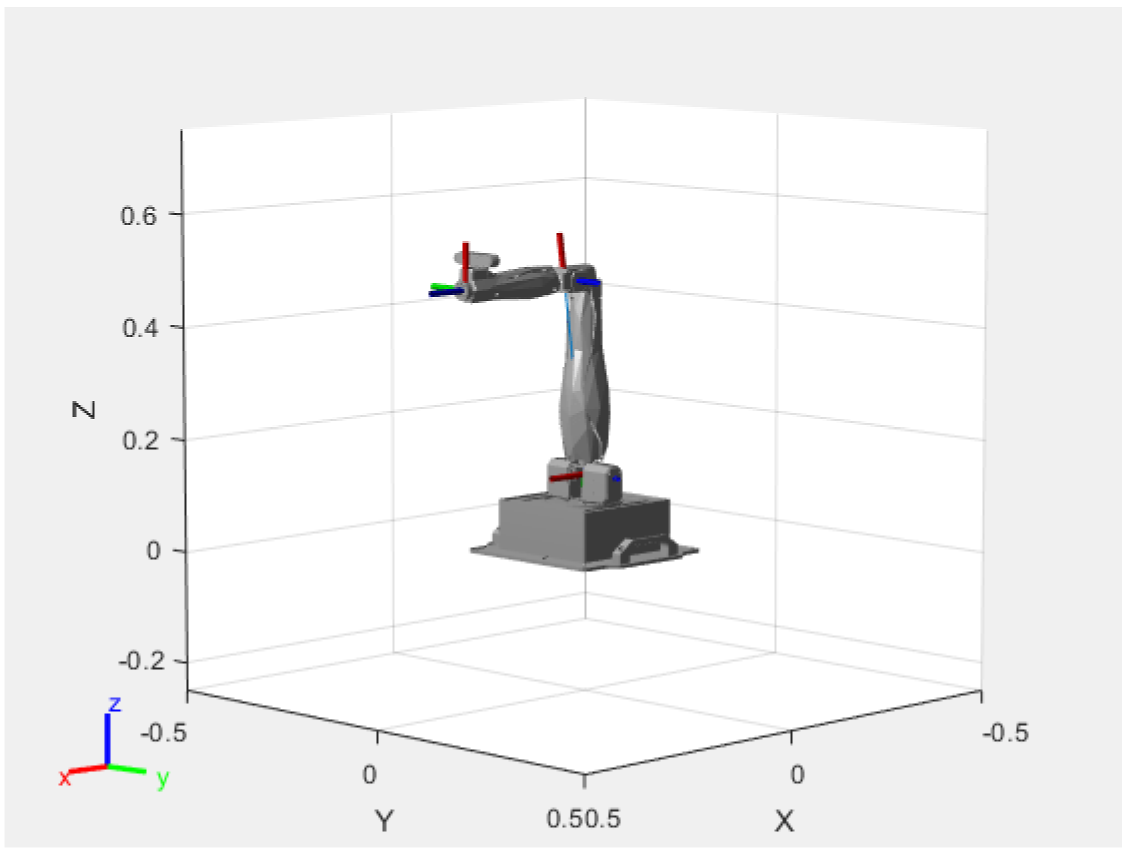
Validate State and Motion Manipulator State Space

Generate states to form a path, validate motion between states, and check for self-collisions and environmental collisions with objects in your world. The `manipulatorStateSpace` object represents the joint configuration space of your rigid body tree robot model, and can sample states, calculate distances, and enforce state bounds. The `manipulatorCollisionBodyValidator` object validates the state and motion based on the collision bodies in your robot model and any obstacles in your environment.

Load Robot Model

Use the `loadrobot` function to access predefined robot models. This example uses the Quanser QArm™ robot and joint configurations are specified as row vectors.

```
robot = loadrobot("quanserQArm",DataFormat="row");
figure(Visible="on")
show(robot);
xlim([-0.5 0.5])
ylim([-0.5 0.5])
zlim([-0.25 0.75])
hold on
```



Configure State Space and State Validation

Create the state space and state validator from the robot model.

```
ss = manipulatorStateSpace(robot);
sv = manipulatorCollisionBodyValidator(ss,SkippedSelfCollisions="parent");
```

Set the validation distance to `0.05`, which is based on the distance normal between two states. You can configure the validator to ignore self collisions to improve the speed of validation, but must consider whether your robot model has the proper joint limit settings set to ensure it does not collide with itself.

```
sv.ValidationDistance = 0.05;
sv.IgnoreSelfCollision = true;
```

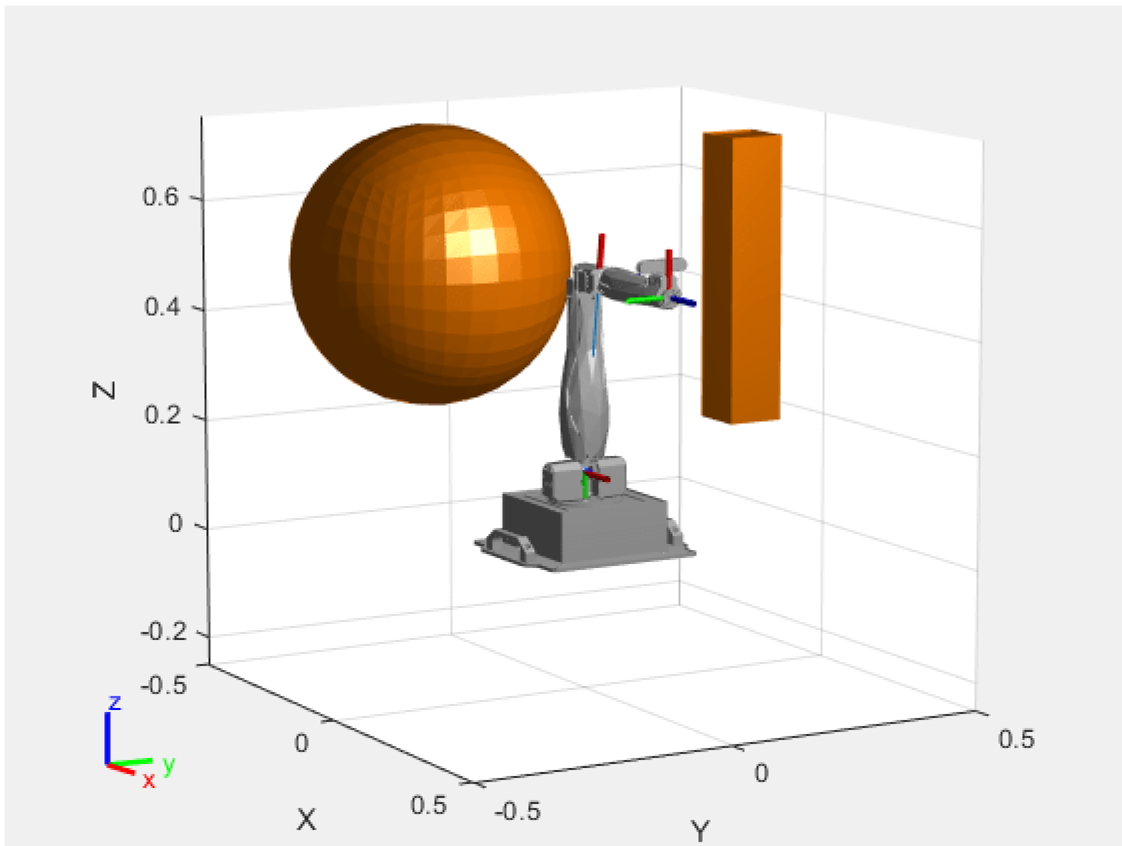
Place collision objects in the robot environment. Set the `Environment` property of the collision validator object using a cell array of objects.

```
box = collisionBox(0.1,0.1,0.5); % XYZ Lengths
box.Pose = trvec2tform([0.2 0.2 0.5]); % XYZ Position
sphere = collisionSphere(0.25); % Radius
sphere.Pose = trvec2tform([-0.2 -0.2 0.5]); % XYZ Position
env = {box sphere};
sv.Environment = env;
```

Visualize the environment.

```
for i = 1:length(env)
    show(env{i})
```

```
end
view(60,10)
```



Plan Path

Start with the home configuration as the first point on the path.

```
rng(0); % Repeatable results
start = homeConfiguration(robot);
path = start;
idx = 1;
```

Plan a path using these steps, in a loop:

- Sample a nearby goal configuration, using the Gaussian distribution, by specifying the standard deviation for each joint angle.
- Check if the sampled goal state is valid.
- If the sampled goal state is valid, check if the motion between states is valid and, if so, add it to the path.

```
for i = 2:25
    goal = sampleGaussian(ss,start,0.25*ones(4,1));
    validState = isStateValid(sv,goal);

    if validState % If state is valid, check motion between states.
        [validMotion,~] = isMotionValid(sv,path(idx,:),goal);
```

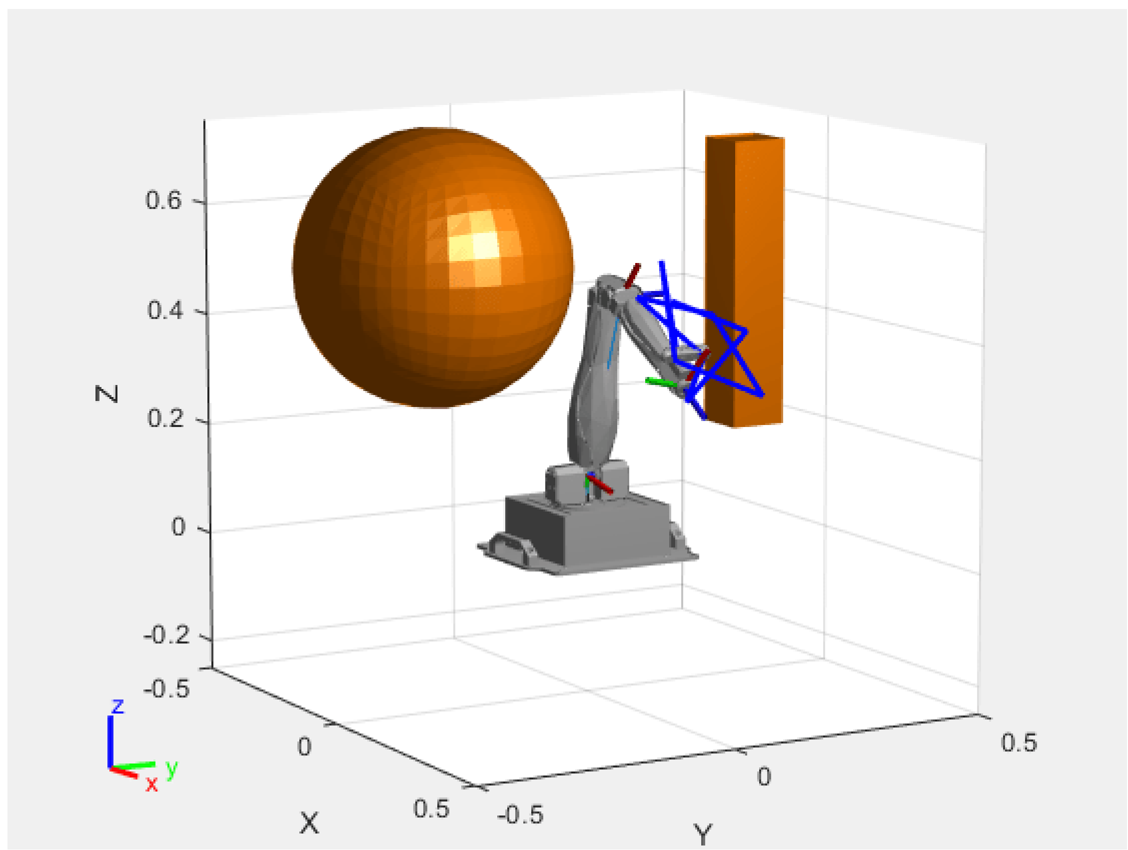
```
        if validMotion % If motion is valid, add to path.
            path = [path; goal];
            idx = idx + 1;
        end
    end
end
```

Visualize Path

After generating the path of valid motions, visualize the robot motion. Because you sampled random states near the home configuration, you should see the arm move around that initial configuration.

To visualize the path of the end effector in 3-D, get the transformation, relative to the base world frame at each point. Store the points as an *xyz* translation vector. Plot the path of the end effector.

```
eePose = nan(3,size(path,1));

for i = 1:size(path,1)
    show(robot,path(i,:),PreservePlot=false);
    eePos(i,:) = tform2trvec(getTransform(robot,path(i,:),"END-EFFECTOR")); % XYZ translation vec
    plot3(eePos(:,1),eePos(:,2),eePos(:,3),"-b",LineWidth=2)
    drawnow
end
```



Input Arguments

manipSS — Manipulator state space
`manipulatorStateSpace` object

Manipulator state space, specified as a `manipulatorStateSpace` object, which is a subclass of `nav.StateSpace`.

meanState — Mean state position
n-element row vector

Mean state position, specified as an *n*-element row vector, where *n* is the dimension of the state space specified in the `NumStateVariables` property of `manipSS`.

stdDev — Standard deviation around mean state
n-element row vector

Standard deviation around the mean state, specified as an *n*-element row vector. Each element corresponds to an element in `meanState`.

numSamples — Number of samples
`1` (default) | positive integer

Number of samples, specified as a positive integer.

Output Arguments

`states` — Sampled states from state space
n-element row vector | *m*-by-*n* matrix

Sampled states from the state space, returned as an *n*-element row vector or *m*-by-*n* matrix. *n* is the dimension of the state space specified in the `NumStateVariables` property of `manipSS`. *m* is the number of samples specified in `numSamples`. All states are sampled within the bounds specified by the `StateBounds` property of `manipSS`.

Version History

Introduced in R2021b

See Also

Objects
`manipulatorStateSpace` | `rigidBodyTree` | `manipulatorCollisionBodyValidator` | `manipulatorRRT` | `workspaceGoalRegion`

Functions
`isStateValid` | `isMotionValid` | `sampleUniform` | `interpolate` | `distance` | `enforceStateBounds`

sampleUniform

Sample state using uniform distribution

Syntax

```
states = sampleUniform(manipSS)
states = sampleUniform(manipSS,numSamples)
```

Description

`states = sampleUniform(manipSS)` samples a single random state within the bounds of the state space `manipSS` using a uniform distribution.

`states = sampleUniform(manipSS,numSamples)` samples the number of states specified by `numSamples`.

Input Arguments

`manipSS` — Manipulator state space
`manipulatorStateSpace` object

Manipulator state space, specified as a `manipulatorStateSpace` object, which is a subclass of `nav.StateSpace`.

`numSamples` — Number of samples
positive integer

Number of samples, specified as a positive integer.

Output Arguments

`states` — Sampled states from state space
n-element row vector | *m*-by-*n* matrix

Sampled states from the state space, returned as an *n*-element row vector or *m*-by-*n* matrix. *n* is the dimension of the state space specified in the `NumStateVariables` property of `manipSS`. *m* is the number of samples specified in `numSamples`. All states are sampled within the bounds specified by the `StateBounds` property of `manipSS`.

Version History

Introduced in R2021b

See Also

Objects
`manipulatorStateSpace` | `rigidBodyTree` | `manipulatorCollisionBodyValidator` | `manipulatorRRT` | `workspaceGoalRegion`

Functions
`isStateValid` | `isMotionValid` | `sampleGaussian` | `interpolate` | `distance` | `enforceStateBounds`

Topics
"Create Custom State Space for Path Planning" (Navigation Toolbox)

derivative

Time derivative of vehicle state

Syntax

`stateDot = derivative(motionModel,state,cmds)`

Description

`stateDot = derivative(motionModel,state,cmds)` returns the current state derivative, `stateDot`, as a three-element vector [*xDot yDot thetaDot*] if the motion model is a `bicycleKinematics`, `differentialDriveKinematics`, or `unicycleKinematics` object. It returns `state` as a four-element vector, [*xDot yDot thetaDot psiDot*], if the motion model is a `ackermannKinematics` object. *xDot* and *yDot* refer to the vehicle velocity, specified in meters per second. *thetaDot* is the angular velocity of the vehicle heading and *psiDot* is the angular velocity of the vehicle steering, both specified in radians per second.

Examples

Simulate Different Kinematic Models for Mobile Robots

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

Define Mobile Robots with Kinematic Constraints

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

Unicycle Kinematic Model

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

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

Differential-Drive Kinematic Model

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

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

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

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

Bicycle Kinematic Model

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

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

Other Models

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

```
carLike = ackermannKinematics;
```

Set up Simulation Parameters

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

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

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

Create a Vehicle Controller

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

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

Simulate the Models Using an ODE Solver

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

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

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

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

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

Plot Results

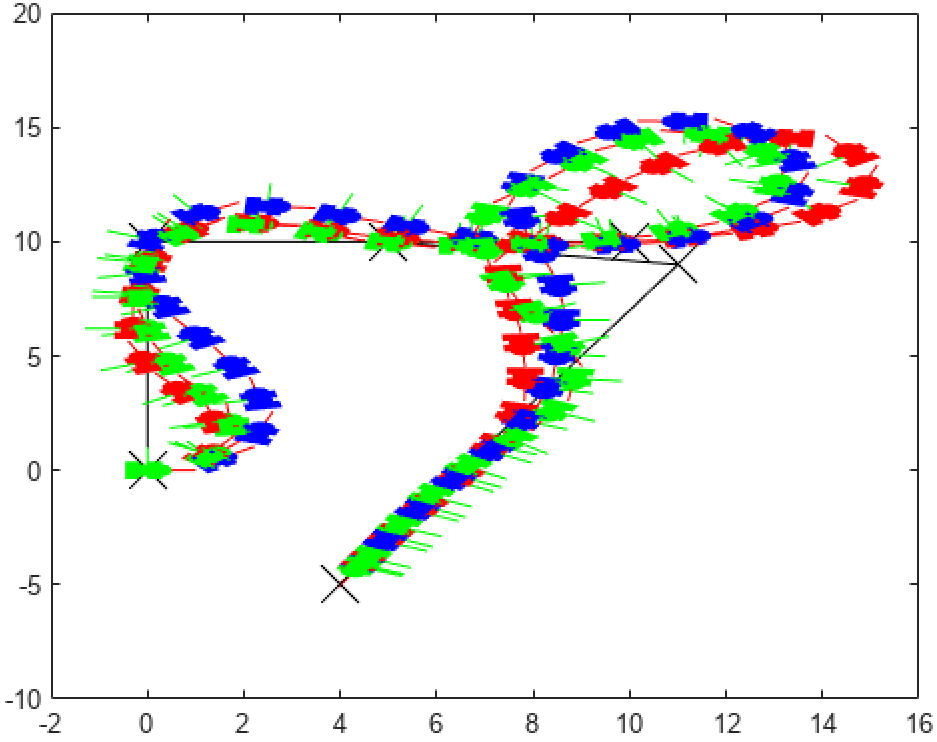
The results of the ODE solver can be easily viewed on a single plot using plotTransforms to visualize the results of all trajectories at once.

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

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

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

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

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

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



Simulate Ackermann Kinematic Model with Steering Angle Constraints

Simulate a mobile robot model that uses Ackermann steering with constraints on its steering angle. During simulation, the model maintains maximum steering angle after it reaches the steering limit. To see the effect of steering saturation, you compare the trajectory of two robots, one with the constraints on the steering angle and the other without any steering constraints.

Define the Model

Define the Ackermann kinematic model. In this car-like model, the front wheels are a given distance apart. To ensure that they turn on concentric circles, the wheels have different steering angles. While turning, the front wheels receive the steering input as rate of change of steering angle.

```
carLike = ackermannKinematics;
```

Set Up Simulation Parameters

Set the mobile robot to follow a constant linear velocity and receive a constant steering rate as input. Simulate the constrained robot for a longer period to demonstrate steering saturation.

```
velo = 5;     % Constant linear velocity
psidot = 1;   % Constant left steering rate

% Define the total time and sample rate
sampleTime = 0.05;                  % Sample time [s]
```

```
timeEnd1 = 1.5;                    % Simulation end time for unconstrained robot
timeEnd2 = 10;                     % Simulation end time for constrained robot
tVec1 = 0:sampleTime:timeEnd1;     % Time array for unconstrained robot
tVec2 = 0:sampleTime:timeEnd2;     % Time array for constrained robot

initPose = [0;0;0;0];              % Initial pose (x y theta phi)
```

Create Options Structure for ODE Solver

In this example, you pass an `options` structure as argument to the ODE solver. The `options` structure contains the information about the steering angle limit. To create the `options` structure, use the `Events` option of `odeset` and the created event function, `detectSteeringSaturation`. `detectSteeringSaturation` sets the maximum steering angle to 45 degrees.

For a description of how to define `detectSteeringSaturation`, see **Define Event Function** at the end of this example.

```
options = odeset('Events',@detectSteeringSaturation);
```

Simulate Model Using ODE Solver

Next, you use the `derivative` function and an ODE solver, `ode45`, to solve the model and generate the solution.

```
% Simulate the unconstrained robot
[t1,pose1] = ode45(@(t,y)derivative(carLike,y,[velo psidot]),tVec1,initPose);

% Simulate the constrained robot
[t2,pose2,te,ye,ie] = ode45(@(t,y)derivative(carLike,y,[velo psidot]),tVec2,initPose,options);
```

Detect Steering Saturation

When the model reaches the steering limit, it registers a timestamp of the event. The time it took to reach the limit is stored in `te`.

```
if te < timeEnd2
    str1 = "Steering angle limit was reached at ";
    str2 = " seconds";
    comp = str1 + te + str2;
    disp(comp)
end
```

```
Steering angle limit was reached at 0.785 seconds
```

Simulate Constrained Robot with New Initial Conditions

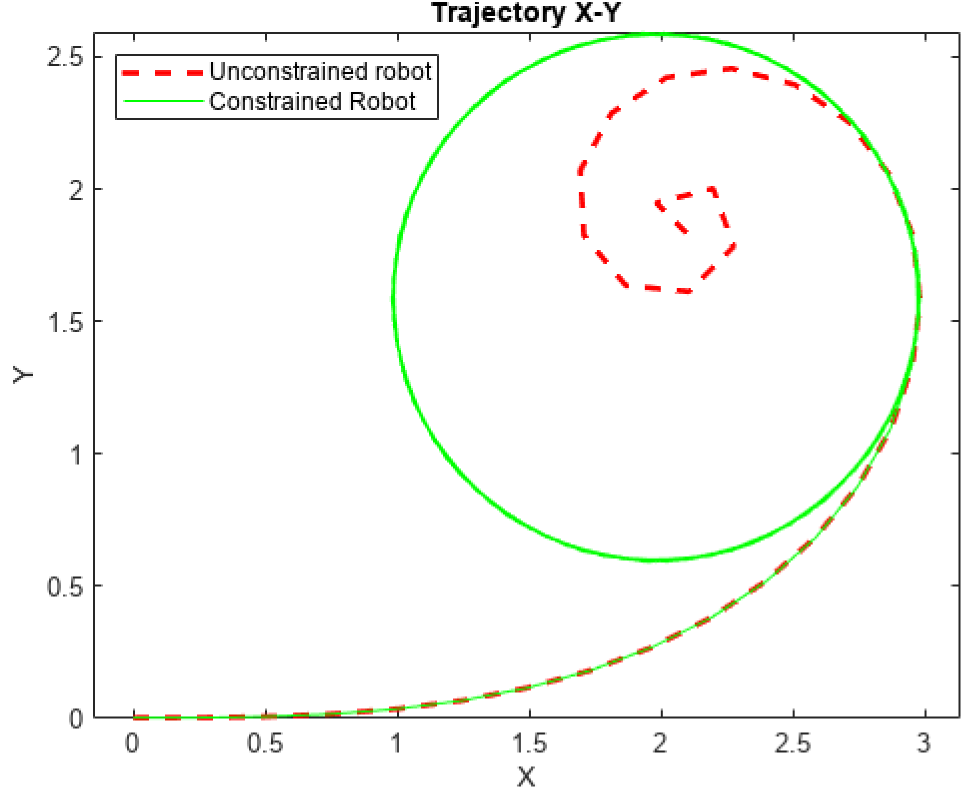
Now use the state of the constrained robot before termination of integration as initial condition for the second simulation. Modify the input vector to represent steering saturation, that is, set the steering rate to zero.

```
saturatedPsiDot = 0;             % Steering rate after saturation
cmds = [velo saturatedPsiDot];   % Command vector
tVec3 = te:sampleTime:timeEnd2;  % Time vector
pose3 = pose2(length(pose2),:);
[t3,pose3,te3,ye3,ie3] = ode45(@(t,y)derivative(carLike,y,cmds), tVec3,pose3, options);
```

Plot the Results

Plot the trajectory of the robot using `plot` and the data stored in `pose`.

```
figure(1)
plot(pose1(:,1),pose1(:,2),'--r','LineWidth',2);
hold on;
plot([pose2(:,1); pose3(:,1)],[pose2(:,2);pose3(:,2)],'g');
title('Trajectory X-Y')
xlabel('X')
ylabel('Y')
legend('Unconstrained robot','Constrained Robot','Location','northwest')
axis equal
```



The unconstrained robot follows a spiral trajectory with decreasing radius of curvature while the constrained robot follows a circular trajectory with constant radius of curvature after the steering limit is reached.

Define Event Function

Set the event function such that integration terminates when 4th state, theta, is equal to maximum steering angle.

```
function [state,isterminal,direction] = detectSteeringSaturation(t,y)
  maxSteerAngle = 0.785;               % Maximum steering angle (pi/4 radians)
  state(4) = (y(4) - maxSteerAngle);   % Saturation event occurs when the 4th state, theta, is e
  isterminal(4) = 1;                   % Integration is terminated when event occurs
  direction(4) = 0;                    % Bidirectional termination
```

```
end
```

Input Arguments

`motionModel` — Mobile kinematic model object

`ackermannKinematics` object | `bicycleKinematics` object | `differentialDriveKinematics` object | `unicycleKinematics` object

The mobile kinematics model object, which defines the properties of the motion model, specified as an `ackermannKinematics`, `bicycleKinematics`, `differentialDriveKinematics`, or a `unicycleKinematics` object.

`state` — Current vehicle state

three-element vector | four-element vector

Current vehicle state returned as a three-element or four-element vector, depending on the `motionModel` input:

- `unicycleKinematics` -- [*x y theta*]
- `bicycleKinematics` -- [*x y theta*]
- `differentialDriveKinematics` -- [*x y theta*]
- `ackermannKinematics` -- [*x y theta psi*]

x and *y* refer to the vehicle position, specified in meters per second. *theta* is the vehicle heading and *psi* is the vehicle steering angle, both specified in radians per second.

`cmds` — Input commands to motion model

two-element vector

Input commands to the motion model, specified as a two-element vector that depends on the motion model.

For `ackermannKinematics` objects, the commands are [*v psiDot*].

For other motion models, the `VehicleInputs` property of `motionModel` determines the command vector:

- `"VehicleSpeedSteeringAngle"` -- [*v psiDot*]
- `"VehicleSpeedHeadingRate"` -- [*v omegaDot*]
- `"WheelSpeedHeadingRate"` (`unicycleKinematics` only) -- [*wheelSpeed omegaDot*]
- `"WheelSpeeds"` (`differentialDriveKinematics` only) -- [*wheelL wheelR*]

v is the vehicle velocity in the direction of motion in meters per second. *psiDot* is the steering angle rate in radians per second. *omegaDot* is the angular velocity at the rear axle. *wwheelL* and *wheelR* are the left and right wheel speeds respectively.

Output Arguments

`stateDot` — State derivative of current state

three-element vector | four-element vector

The current state derivative returned as a three-element or four-element vector, depending on the `motionModel` input:

- `unicycleKinematics` –– [*xDot yDot thetaDot*]
- `bicycleKinematics` –– [*xDot yDot thetaDot*]
- `differentialDriveKinematics` –– [*xDot yDot thetaDot*]
- `ackermannKinematics` –– [*xDot yDot thetaDot psiDot*]

xDot and *yDot* refer to the vehicle velocity, specified in meters per second. *thetaDot* is the angular velocity of the vehicle heading and *psiDot* is the angular velocity of the vehicle steering, both specified in radians per second.

Version History

Introduced in R2019b

References

[1] Lynch, Kevin M., and Frank C. Park. *Modern Robotics: Mechanics, Planning, and Control*. 1st ed. Cambridge, MA: Cambridge University Press, 2017.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`ackermannKinematics` | `bicycleKinematics` | `differentialDriveKinematics` | `unicycleKinematics`

Topics

"Mobile Robot Kinematics Equations"

findpath

Find path between start and goal points on roadmap

Syntax

```
xy = findpath(prm,start,goal)
```

Description

`xy = findpath(prm,start,goal)` finds an obstacle-free path between `start` and `goal` locations within `prm`, a roadmap object that contains a network of connected points.

If any properties of `prm` change, or if the roadmap is not created, `update` is called.

Input Arguments

`prm` — Roadmap path planner
`mobileRobotPRM` object

Roadmap path planner, specified as a `mobileRobotPRM` object.

`start` — Start location of path
1-by-2 vector

Start location of path, specified as a 1-by-2 vector representing an `[x y]` pair.

Example: `[0 0]`

`goal` — Final location of path
1-by-2 vector

Final location of path, specified as a 1-by-2 vector representing an `[x y]` pair.

Example: `[10 10]`

Output Arguments

`xy` — Waypoints for a path between `start` and `goal`
n-by-2 column vector

Waypoints for a path between start and goal, specified as a *n*-by-2 column vector of `[x y]` pairs, where *n* is the number of waypoints. These pairs represent the solved path from the `start` and `goal` locations, given the roadmap from the `prm` input object.

Version History

Introduced in R2019b

See Also

`mobileRobotPRM` | `show` | `update`

show

Show map, roadmap, and path

Syntax

```
show(prm)
show(prm,Name,Value)
```

Description

`show(prm)` shows the map and the roadmap, specified as `prm` in a figure window. If no roadmap exists, `update` is called. If a path is computed before calling `show`, the path is also plotted on the figure.

`show(prm,Name,Value)` sets the specified `Value` to the property `Name`.

Input Arguments

`prm` — Roadmap path planner
`mobileRobotPRM` object

Roadmap path planner, specified as a `mobileRobotPRM` object.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `'Path','off'`

`Parent` — Axes to plot the map
`Axes` object | `UIAxes` object

Axes to plot the map specified as a comma-separated pair consisting of `"Parent"` and either an `Axes` or `UIAxes`object. See `axes` or `uiaxes`.

`Map` — Map display option
`"on"` (default) | `"off"`

Map display option, specified as the comma-separated pair consisting of `"Map"` and either `"on"` or `"off"`.

`Roadmap` — Roadmap display option
`"on"` (default) | `"off"`

Roadmap display option, specified as the comma-separated pair consisting of `"Roadmap"` and either `"on"` or `"off"`.

`Path` — Path display option
`"on"` (default) | `"off"`

Path display option, specified as `"on"` or `"off"`. This controls whether the computed path is shown in the plot.

Version History

Introduced in R2019b

See Also

`mobileRobotPRM` | `findpath` | `update`

Topics

"Path Following for a Differential Drive Robot"

update

Create or update roadmap

Syntax

`update(prm)`

Description

`update(prm)` creates a roadmap if called for the first time after creating the `mobileRobotPRM` object, `prm`. Subsequent calls of `update` recreate the roadmap by resampling the map. `update` creates the new roadmap using the `Map`, `NumNodes`, and `ConnectionDistance` property values specified in `prm`.

Input Arguments

`prm` — Roadmap path planner
`mobileRobotPRM` object

Roadmap path planner, specified as a `mobileRobotPRM` object.

Version History

Introduced in R2019b

See Also

`mobileRobotPRM` | `findpath` | `show`

findNearestNeighbors

Find nearest neighbors of a point in point cloud

Syntax

```
[indices,dists] = findNearestNeighbors(ptCloud,point,K)
[indices,dists] = findNearestNeighbors( ___ ,Name,Value)
```

Description

`[indices,dists] = findNearestNeighbors(ptCloud,point,K)` returns the `indices` for the K-nearest neighbors of a query point in the input point cloud. `ptCloud` can be an unorganized or organized point cloud. The K-nearest neighbors of the query point are computed by using the Kd-tree based search algorithm. This function requires a Computer Vision Toolbox™ license.

`[indices,dists] = findNearestNeighbors( ___ ,Name,Value)` specifies options using one or more name-value arguments in addition to the input arguments in the preceding syntaxes.

Input Arguments

`ptCloud` — Point cloud
`pointCloud` object

Point cloud, specified as a `pointCloud` object.

`point` — Query point
three-element vector of form `[`*x y z*`]`

Query point, specified as a three-element vector of form `[`*x y z*`]`.

`K` — Number of nearest neighbors
positive integer

Number of nearest neighbors, specified as a positive integer.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `findNearestNeighbors(ptCloud,point,k,'Sort',true)`

`Sort` — Sort indices
`false` (default) | `true`

Sort indices, specified as a comma-separated pair of `'Sort'` and a logical scalar. When you set `Sort` to `true`, the returned indices are sorted in the ascending order based on the distance from a query point. To turn off sorting, set `Sort` to `false`.

`MaxLeafChecks` — Number of leaf nodes to check
`Inf` (default) | integer

Number of leaf nodes to check, specified as a comma-separated pair consisting of `'MaxLeafChecks'` and an integer. When you set this value to `Inf`, the entire tree is searched. When the entire tree is searched, it produces exact search results. Increasing the number of leaf nodes to check increases accuracy, but reduces efficiency.

Note The name-value argument `'MaxLeafChecks'` is valid only with Kd-tree based search method.

Output Arguments

`indices` — Indices of stored points
column vector

Indices of stored points, returned as a column vector. The vector contains K linear indices of the nearest neighbors stored in the point cloud.

`dists` — Distances to query point
column vector

Distances to query point, returned as a column vector. The vector contains the Euclidean distances between the query point and its nearest neighbors.

Version History

Introduced in R2022a

References

[1] Muja, M. and David G. Lowe. "Fast Approximate Nearest Neighbors with Automatic Algorithm Configuration". *In VISAPP International Conference on Computer Vision Theory and Applications*. 2009. pp. 331–340.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- For code generation in non-host platforms, the value for `'MaxLeafChecks'` must be set to the default value `Inf`. If you specify values other than `Inf`, the function generates a warning and automatically assigns the default value for `'MaxLeafChecks'`.

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

Usage notes and limitations:

- For GPU code generation, the `'MaxLeafChecks'` name-value pair option is ignored.

See Also

Objects
`pointCloud`

Functions
`findNeighborsInRadius` | `findPointsInROI` | `removeInvalidPoints` | `select`

findNeighborsInRadius

Find neighbors within a radius of a point in the point cloud

Syntax

```
[indices,dists] = findNeighborsInRadius(ptCloud,point,radius)
[indices,dists] = findNeighborsInRadius( ___ ,Name,Value)
```

Description

`[indices,dists] = findNeighborsInRadius(ptCloud,point,radius)` returns the `indices` of neighbors within a radius of a query point in the input point cloud. `ptCloud` can be an unorganized or organized point cloud. The neighbors within a radius of the query point are computed by using the Kd-tree based search algorithm. This function requires a Computer Vision Toolbox license.

`[indices,dists] = findNeighborsInRadius( ___ ,Name,Value)` specifies options using one or more name-value pair arguments in addition to the input arguments in the preceding syntaxes.

Input Arguments

`ptCloud` — Point cloud
`pointCloud` object

Point cloud, specified as a `pointCloud` object.

`point` — Query point
three-element vector of form `[x y z]`

Query point, specified as a three-element vector of form `[x y z]`.

`radius` — Search radius
scalar

Search radius, specified as a scalar. The function finds the neighbors within the specified `radius` around a query point in the input point cloud.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `findNeighborsInRadius(ptCloud,point,radius,'Sort',true)`

`Sort` — Sort indices
`false` (default) | `true`

Sort indices, specified as a comma-separated pair of `'Sort'` and a logical scalar. When you set `Sort` to `true`, the returned indices are sorted in the ascending order based on the distance from a query point. To turn off sorting, set `Sort` to `false`.

`MaxLeafChecks` — Number of leaf nodes
`Inf` (default) | integer

Number of leaf nodes, specified as a comma-separated pair consisting of `'MaxLeafChecks'` and an integer. When you set this value to `Inf`, the entire tree is searched. When the entire tree is searched, it produces exact search results. Increasing the number of leaf nodes to check increases accuracy, but reduces efficiency.

Output Arguments

`indices` — Indices of stored points
column vector

Indices of stored points, returned as a column vector. The vector contains the linear indices of the radial neighbors stored in the point cloud.

`dists` — Distances to query point
column vector

Distances to query point, returned as a column vector. The vector contains the Euclidean distances between the query point and its radial neighbors.

Version History

Introduced in R2022a

References

[1] Muja, M. and David G. Lowe. "Fast Approximate Nearest Neighbors with Automatic Algorithm Configuration". *In VISAPP International Conference on Computer Vision Theory and Applications*. 2009. pp. 331–340.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- For code generation in non-host platforms, the value for `'MaxLeafChecks'` must be set to the default value `Inf`. If you specify values other than `Inf`, the function generates a warning and automatically assigns the default value for `'MaxLeafChecks'`.

GPU Code Generation
Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

Usage notes and limitations:

- For GPU code generation, the `'MaxLeafChecks'` name-value pair option is ignored.

See Also

Objects
`pointCloud`

Functions
`findNearestNeighbors` | `findPointsInROI` | `removeInvalidPoints` | `select`

findPointsInROI

Find points within a region of interest in the point cloud

Syntax

```
indices = findPointsInROI(ptCloud,roi)
```

Description

`indices = findPointsInROI(ptCloud,roi)` returns the points within a region of interest (ROI) in the input point cloud. The points within the specified ROI are obtained using a Kd-tree based search algorithm. This function requires a Computer Vision Toolbox license.

Input Arguments

`ptCloud` — Point cloud
`pointCloud` object

Point cloud, specified as a `pointCloud` object.

`roi` — Region of interest
six-element vector of form [*xmin xmax ymin ymax zmin zmax*]

Region of interest, specified as a six-element vector of form [*xmin xmax ymin ymax zmin zmax*], where:

- *xmin* and *xmax* are the minimum and the maximum limits along the *x*-axis respectively.
- *ymin* and *ymax* are the minimum and the maximum limits along the *y*-axis respectively.
- *zmin* and *zmax* are the minimum and the maximum limits along the *z*-axis respectively.

Output Arguments

`indices` — Indices of stored points
column vector

Indices of stored points, returned as a column vector. The vector contains the linear indices of the ROI points stored in the point cloud.

Version History

Introduced in R2022a

References

[1] Muja, M. and David G. Lowe. "Fast Approximate Nearest Neighbors with Automatic Algorithm Configuration". *In VISAPP International Conference on Computer Vision Theory and Applications*. 2009. pp. 331–340.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

GPU Code Generation
Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

-

See Also

Objects
`pointCloud`

Functions
`findNearestNeighbors` | `findNeighborsInRadius` | `removeInvalidPoints` | `select`

removeInvalidPoints

Remove invalid points from point cloud

Syntax

```
[ptCloudOut,indices] = removeInvalidPoints(ptCloud)
```

Description

`[ptCloudOut,indices] = removeInvalidPoints(ptCloud)` removes points with `Inf` or `NaN` coordinate values from point cloud and returns the indices of valid points. This function requires a Computer Vision Toolbox license.

Input Arguments

`ptCloud` — Point cloud
`pointCloud` object

Point cloud, specified as a `pointCloud` object.

Output Arguments

`ptCloudOut` — Point cloud with points removed
`pointCloud` object

Point cloud, returned as a `pointCloud` object with `Inf` or `NaN` coordinates removed.

Note The output is always an unorganized (*X*-by-3) point cloud. If the input `ptCloud` is an organized point cloud (*M*-by-*N*-by-3), the function returns the output as an unorganized point cloud.

`indices` — Indices of valid points
vector

Indices of valid points in the point cloud, specified as a vector.

Version History

Introduced in R2022a

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

GPU Code Generation
Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

See Also

Objects
pointCloud

Functions
findNearestNeighbors | findNeighborsInRadius | findPointsInROI | select

select

Select points in point cloud

Syntax

```
ptCloudOut = select(ptCloud,indices)
ptCloudOut = select(ptCloud,row,column)
ptCloudOut = select( ___ ,'OutputSize',outputSize)
```

Description

`ptCloudOut = select(ptCloud,indices)` returns a `pointCloud` object containing only the points that are selected using linear indices. This function requires a Computer Vision Toolbox license.

`ptCloudOut = select(ptCloud,row,column)` returns a `pointCloud` object containing only the points that are selected using row and column subscripts. This syntax applies only if the input is an organized point cloud data of size *M*-by-*N*-by-3.

`ptCloudOut = select( ___ ,'OutputSize',outputSize)` returns the selected points as a `pointCloud` object of size specified by `outputSize`.

Input Arguments

`ptCloud` — Point cloud
`pointCloud` object

Point cloud, specified as a `pointCloud` object.

`indices` — Indices of selected points
vector

Indices of selected points, specified as a vector.

`row` — Row indices
vector

Row indices, specified as a vector. This argument applies only if the input is an organized point cloud data of size *M*-by-*N*-by-3.

`column` — Column indices
vector

Column indices, specified as a vector. This argument applies only if the input is an organized point cloud data of size *M*-by-*N*-by-3.

`outputSize` — Size of output point cloud
`'selected'` (default) | `'full'`

Size of the output point cloud, `ptCloudOut`, specified as `'selected'` or `'full'`.

- If the size is 'selected', then the output contains only the selected points from the input point cloud, ptCloud.
- If the size is 'full', then the output is same size as the input point cloud ptCloud. Cleared points are filled with NaN and the color is set to [0 0 0].

Output Arguments

ptCloudOut — Selected point cloud
pointCloud object

Point cloud, returned as a pointCloud object.

Version History

Introduced in R2022a

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

GPU Code Generation
Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

See Also

Objects
pointCloud

Functions
findNearestNeighbors | findNeighborsInRadius | findPointsInROI | removeInvalidPoints

lookupPose

Obtain pose information for certain time

Syntax

```
[position,orientation,velocity,acceleration,angularVelocity] = lookupPose(
trajectory,sampleTimes)
```

Description

`[position,orientation,velocity,acceleration,angularVelocity] = lookupPose(trajectory,sampleTimes)` returns the pose information of the polynomial trajectory at the specified sample times. If any sample time is beyond the duration of the trajectory, the function returns the corresponding pose information as `NaN`.

Examples

Obtain Pose Information of Polynomial Trajectory at Certain Time

Use the `minsnappolytraj` function to generate the piecewise polynomial and the time samples for the specified waypoints of a trajectory.

```
waypoints = [0 20 20 0 0; 0 0 5 5 0; 0 5 10 5 0];
timePoints = linspace(0,30,5);
numSamples = 100;
[~,~,~,~,~,~,pp,~,~] = minsnappolytraj(waypoints,timePoints,numSamples);
```

Use the `polynomialTrajectory` System object to generate a trajectory from the piecewise polynomial. Specify the sample rate of the trajectory.

```
traj = polynomialTrajectory(pp,SampleRate=100);
```

Inspect the waypoints and times of arrival by using `waypointInfo`.

```
waypointInfo(traj)
```

```
ans=5×2 table
    TimeOfArrival                     Waypoints
    _____________    ________________________________________

          0                  0              0              0
        7.5                 20              0              5
         15                 20              5             10
       22.5                  0              5              5
         30         3.3973e-13    -2.7018e-12    -2.6041e-12
```

Obtain the time of arrival between the second and fourth waypoint. Create timestamps to sample the trajectory.

```
t0 = traj.TimeOfArrival(2);
tf = traj.TimeOfArrival(4);
sampleTimes = linspace(t0,tf,1000);
```

Obtain the position, orientation, velocity, and acceleration information at the sampled timestamps using the `lookupPose` object function.

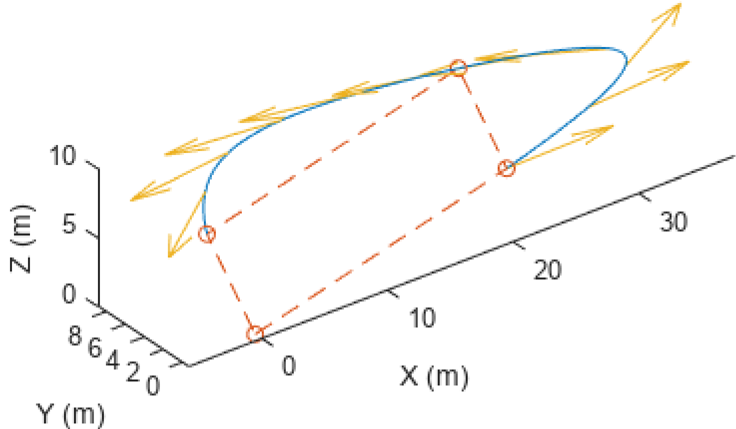
```
[pos,orient,vel,accel,~] = lookupPose(traj,sampleTimes);
```

Get the yaw angle from the orientation.

```
eulOrientation = quat2eul(orient);
yawAngle = eulOrientation(:,1);
```

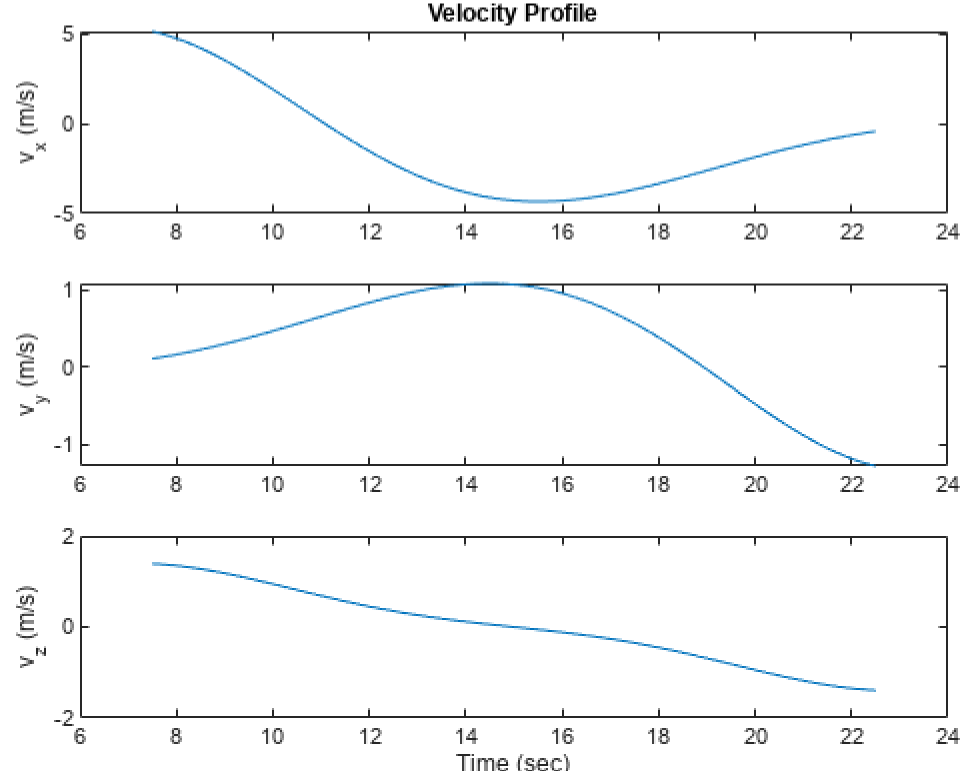
Plot the generated positions and orientations, as well as the specified waypoints.

```
plot3(pos(:,1),pos(:,2),pos(:,3), ...
      waypoints(1,:),waypoints(2,:),waypoints(3,:),"--o")
hold on
% Plot the yaw using quiver.
quiverIdx = 1:100:length(pos);
quiver3(pos(quiverIdx,1),pos(quiverIdx,2),pos(quiverIdx,3), ...
        cos(yawAngle(quiverIdx)),sin(yawAngle(quiverIdx)), ...
        zeros(numel(quiverIdx),1))
title("Position and Orientation")
xlabel("X (m)")
ylabel("Y (m)")
zlabel("Z (m)")
legend({"Position","Waypoints","Orientation"})
axis equal
hold off
```



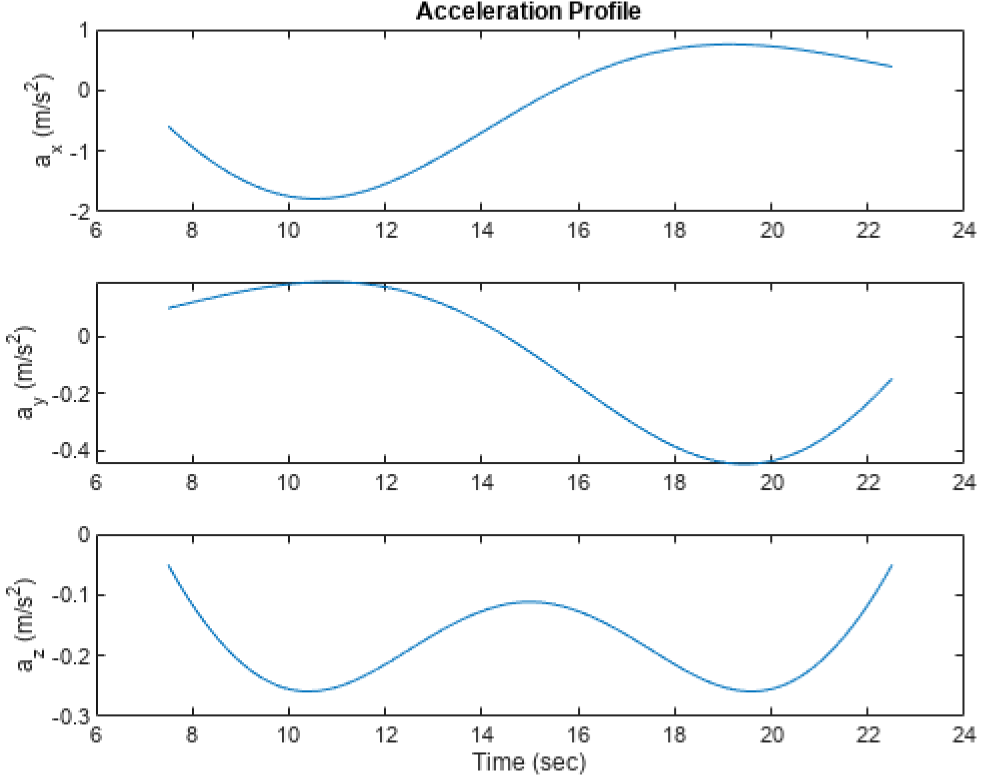
Plot the velocity profiles.

```
figure
subplot(3,1,1)
plot(sampleTimes,vel(:,1))
title("Velocity Profile")
ylabel("v_x (m/s)")
subplot(3,1,2)
plot(sampleTimes,vel(:,2))
ylabel("v_y (m/s)")
subplot(3,1,3)
plot(sampleTimes,vel(:,3))
ylabel("v_z (m/s)")
xlabel("Time (sec)")
```



Plot the acceleration profiles.

```
figure
subplot(3,1,1)
plot(sampleTimes,accel(:,1))
title("Acceleration Profile")
ylabel("a_x (m/s^2)")
subplot(3,1,2)
plot(sampleTimes,accel(:,2))
ylabel("a_y (m/s^2)")
subplot(3,1,3)
plot(sampleTimes,accel(:,3))
ylabel("a_z (m/s^2)")
xlabel("Time (sec)")
```



Input Arguments

`trajectory` — Polynomial trajectory
`polynomialTrajectory` object

Polynomial trajectory, specified as a `polynomialTrajectory` object.

`sampleTimes` — Sample times
M-element vector of nonnegative numbers

Sample times, in seconds, specified as an *M*-element vector of nonnegative numbers.

Output Arguments

`position` — Position in local navigation coordinate system (m)
M-by-3 matrix

Position in the local navigation coordinate system, in meters, returned as an *M*-by-3 matrix.

M is specified by the `sampleTimes` input.

Data Types: `double`

`orientation` — Orientation in local navigation coordinate system
M-element `quaternion` column vector | 3-by-3-by-*M* real array

Orientation in the local navigation coordinate system, returned as an *M*-element `quaternion` column vector or a 3-by-3-by-*M* real array.

Each quaternion or 3-by-3 rotation matrix is a frame rotation from the local navigation coordinate system to the current body coordinate system at the corresponding sample time.

M is specified by the `sampleTimes` input.

Data Types: `double`

`velocity` — Velocity in local navigation coordinate system (m/s)
M-by-3 matrix

Velocity in the local navigation coordinate system, in meters per second, returned as an *M*-by-3 matrix.

M is specified by the `sampleTimes` input.

Data Types: `double`

`acceleration` — Acceleration in local navigation coordinate system (m/s^2)
M-by-3 matrix

Acceleration in the local navigation coordinate system, in meters per second squared, returned as an *M*-by-3 matrix.

M is specified by the `sampleTimes` input.

Data Types: `double`

`angularVelocity` — Angular velocity in local navigation coordinate system (rad/s)
M-by-3 matrix

Angular velocity in the local navigation coordinate system, in radians per second, returned as an *M*-by-3 matrix.

M is specified by the `sampleTimes` input.

Data Types: `double`

Version History

Introduced in R2023a

See Also

Objects
`polynomialTrajectory`

Functions
`waypointInfo`

waypointInfo

Get waypoint information table

Syntax

```
trajectoryInfo = waypointInfo(trajectory)
```

Description

`trajectoryInfo = waypointInfo(trajectory)` returns a table of waypoints, times of arrival, and orientations for the polynomial trajectory.

Examples

Generate Trajectory from Piecewise Polynomial Using polynomialTrajectory

Use the `minjerkpolytraj` function to generate the piecewise polynomial and the time samples for the specified waypoints of a trajectory.

```
waypoints = [0 20 20 0 0; 0 0 5 5 0; 0 5 10 5 0];
timePoints = cumsum([0 10 1.25*pi 10 1.25*pi]);
numSamples = 100;
[~,~,~,~,pp,~,tsamples] = minjerkpolytraj(waypoints,timePoints,numSamples);
```

Use the `polynomialTrajectory` System object to generate a trajectory from the piecewise polynomial that a multirotor must follow. Specify the sample rate of the trajectory and the orientation at each waypoint.

```
eulerAngles = [0 0 0; 0 0 0; 180 0 0; 180 0 0; 0 0 0];
q = quaternion(eulerAngles,"eulerd","ZYX","frame");
traj = polynomialTrajectory(pp,SampleRate=100,Orientation=q);
```

Inspect the waypoints, times of arrival, and orientation by using `waypointInfo`.

```
waypointInfo(traj)
```

```
ans=5×3 table
    TimeOfArrival                   Waypoints                        Orientation
    _____________    ________________________________________    ________________

          0                  0              0              0     {1x1 quaternion}
         10                 20              0              5     {1x1 quaternion}
     13.927                 20              5             10     {1x1 quaternion}
     23.927                  0              5              5     {1x1 quaternion}
     27.854          6.409e-14    -1.1102e-13    -1.1902e-13     {1x1 quaternion}
```

Obtain pose information one buffer frame at a time.

```
[pos,orient,vel,acc,angvel] = traj();
i = 1;
```

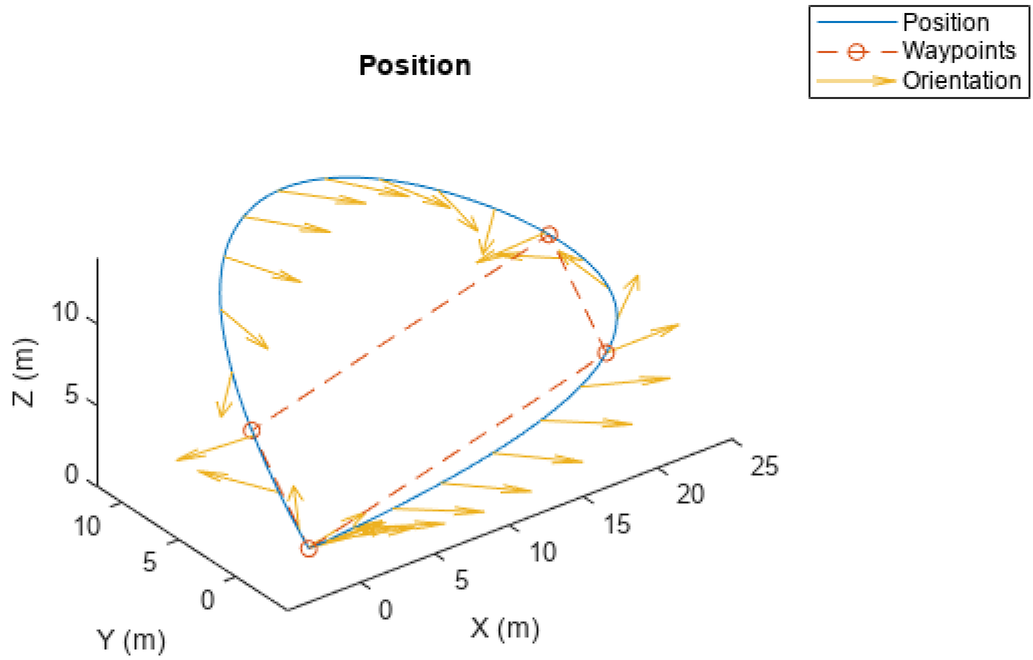
```
spf = traj.SamplesPerFrame;
while ~isDone(traj)
    idx = (i+1):(i+spf);
    [pos(idx,:),orient(idx,:), ...
     vel(idx,:),acc(idx,:),angvel(idx,:)] = traj();
    i = i + spf;
end
```

Get the yaw angle from the orientation.

```
eulOrientation = quat2eul(orient);
yawAngle = eulOrientation(:,1);
```

Plot the generated positions and orientations, as well as the specified waypoints.

```
plot3(pos(:,1),pos(:,2),pos(:,3), ...
      waypoints(1,:),waypoints(2,:),waypoints(3,:),"--o")
hold on
% Plot the yaw using quiver.
quiverIdx = 1:100:length(pos);
quiver3(pos(quiverIdx,1),pos(quiverIdx,2),pos(quiverIdx,3), ...
        cos(yawAngle(quiverIdx)),sin(yawAngle(quiverIdx)), ...
        zeros(numel(quiverIdx),1))
title("Position")
xlabel("X (m)")
ylabel("Y (m)")
zlabel("Z (m)")
legend({"Position","Waypoints","Orientation"})
axis equal
hold off
```



Obtain Pose Information of Polynomial Trajectory at Certain Time

Use the `minsnappolytraj` function to generate the piecewise polynomial and the time samples for the specified waypoints of a trajectory.

```
waypoints = [0 20 20 0 0; 0 0 5 5 0; 0 5 10 5 0];
timePoints = linspace(0,30,5);
numSamples = 100;
[~,~,~,~,~,pp,~,~] = minsnappolytraj(waypoints,timePoints,numSamples);
```

Use the `polynomialTrajectory` System object to generate a trajectory from the piecewise polynomial. Specify the sample rate of the trajectory.

```
traj = polynomialTrajectory(pp,SampleRate=100);
```

Inspect the waypoints and times of arrival by using `waypointInfo`.

```
waypointInfo(traj)
```

```
ans=5×2 table
    TimeOfArrival                   Waypoints
    _____________    ________________________________________

          0                0              0              0
        7.5               20              0              5
```

```
     15                  20               5               10
   22.5                   0               5                5
     30          3.3973e-13     -2.7018e-12     -2.6041e-12
```

Obtain the time of arrival between the second and fourth waypoint. Create timestamps to sample the trajectory.

```
t0 = traj.TimeOfArrival(2);
tf = traj.TimeOfArrival(4);
sampleTimes = linspace(t0,tf,1000);
```

Obtain the position, orientation, velocity, and acceleration information at the sampled timestamps using the `lookupPose` object function.

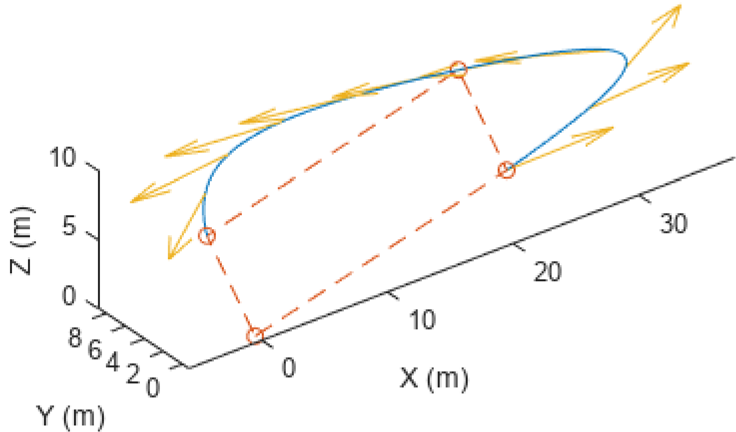
```
[pos,orient,vel,accel,~] = lookupPose(traj,sampleTimes);
```

Get the yaw angle from the orientation.

```
eulOrientation = quat2eul(orient);
yawAngle = eulOrientation(:,1);
```

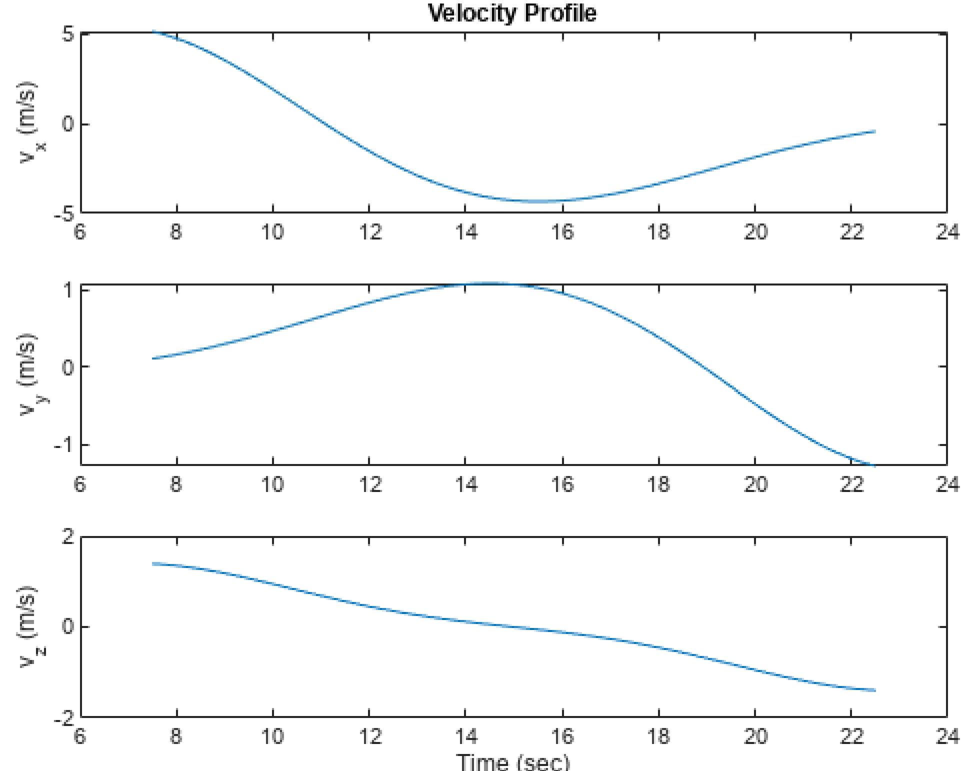
Plot the generated positions and orientations, as well as the specified waypoints.

```
plot3(pos(:,1),pos(:,2),pos(:,3), ...
      waypoints(1,:),waypoints(2,:),waypoints(3,:),"--o")
hold on
% Plot the yaw using quiver.
quiverIdx = 1:100:length(pos);
quiver3(pos(quiverIdx,1),pos(quiverIdx,2),pos(quiverIdx,3), ...
        cos(yawAngle(quiverIdx)),sin(yawAngle(quiverIdx)), ...
        zeros(numel(quiverIdx),1))
title("Position and Orientation")
xlabel("X (m)")
ylabel("Y (m)")
zlabel("Z (m)")
legend({"Position","Waypoints","Orientation"})
axis equal
hold off
```



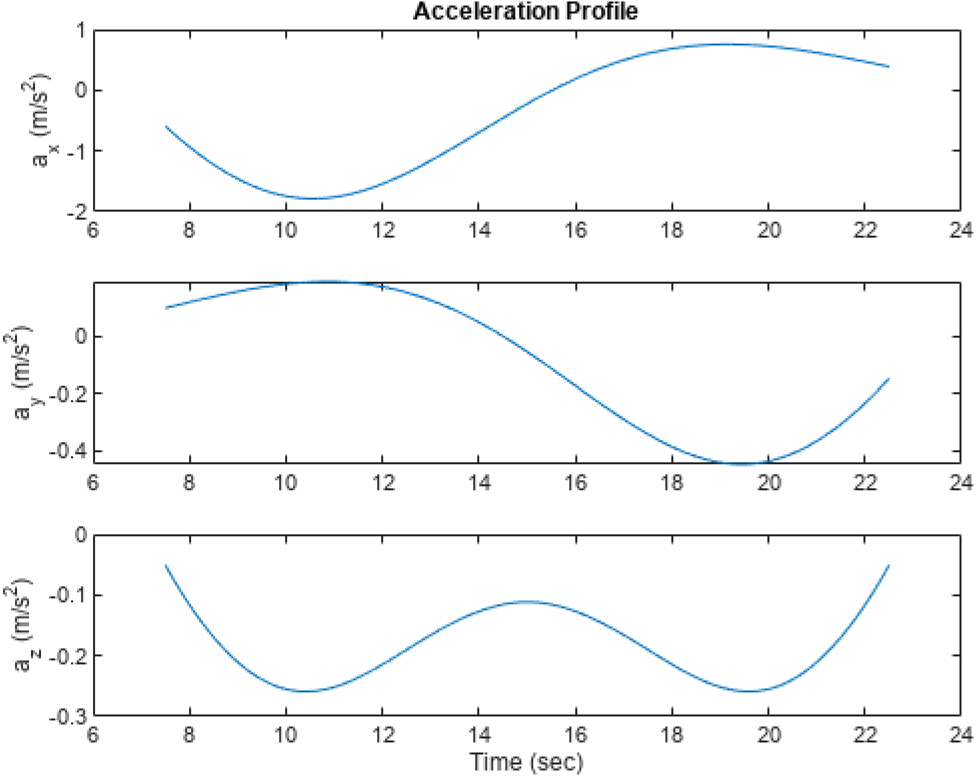
Plot the velocity profiles.

```
figure
subplot(3,1,1)
plot(sampleTimes,vel(:,1))
title("Velocity Profile")
ylabel("v_x (m/s)")
subplot(3,1,2)
plot(sampleTimes,vel(:,2))
ylabel("v_y (m/s)")
subplot(3,1,3)
plot(sampleTimes,vel(:,3))
ylabel("v_z (m/s)")
xlabel("Time (sec)")
```



Plot the acceleration profiles.

```
figure
subplot(3,1,1)
plot(sampleTimes,accel(:,1))
title("Acceleration Profile")
ylabel("a_x (m/s^2)")
subplot(3,1,2)
plot(sampleTimes,accel(:,2))
ylabel("a_y (m/s^2)")
subplot(3,1,3)
plot(sampleTimes,accel(:,3))
ylabel("a_z (m/s^2)")
xlabel("Time (sec)")
```



Input Arguments

`trajectory` — Polynomial trajectory
`polynomialTrajectory` object

Polynomial trajectory, specified as a `polynomialTrajectory` object.

Output Arguments

`trajectoryInfo` — Trajectory information
table

Trajectory information, returned as a table with variables corresponding to these properties of trajectory:

- Waypoints
- TimeOfArrival
- Orientation

The trajectory information table always has columns for `Waypoints` and `TimeOfArrival`. If you set the `Orientation` property when constructing,the trajectory information table additionally returns orientation.

Data Types: `table`

Version History

Introduced in R2023a

See Also

Objects
`polynomialTrajectory`

Functions
`lookupPose`

reset

Reset `Rate` object

Syntax

```
reset(rate)
```

Description

`reset(rate)` resets the state of the `Rate` object, including the elapsed time and all statistics about previous periods. `reset` is useful if you want to run multiple successive loops at the same rate, or if the object is created before the loop is executed.

Input Arguments

rate — Rate object
handle

`Rate` object, specified as a handle. This object contains the information for the desired rate and other information about the execution. See `rateControlrateControl` for more information.

Examples

Run Loop At Fixed Rate and Reset Rate Object

Create a `rateControl` object for running at 20 Hz.

```
r = rateControl(2);
```

Start a loop and control operation using the `Rate` object.

```
for i = 1:30
    % Your code goes here
    waitfor(r);
end
```

Display the `rateControl` object properties after loop operation.

```
disp(r)
```

```
  rateControl with properties:

         DesiredRate: 2
        DesiredPeriod: 0.5000
        OverrunAction: 'slip'
    TotalElapsedTime: 15.0287
          LastPeriod: 0.5118
```

Reset the object to restart the time statistics.

```
reset(r);
disp(r)
```

```
  rateControl with properties:

         DesiredRate: 2
       DesiredPeriod: 0.5000
       OverrunAction: 'slip'
    TotalElapsedTime: 0.0026
          LastPeriod: NaN
```

Version History

Introduced in R2016a

See Also

`rateControl` | `rateControl` | `waitfor`

Topics

"Execute Code at a Fixed-Rate"

statistics

Statistics of past execution periods

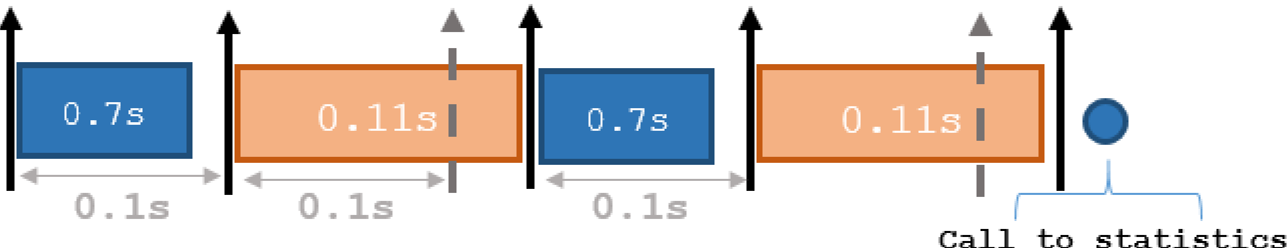
Syntax

```
stats = statistics(rate)
```

Description

`stats = statistics(rate)` returns statistics of previous periods of code execution. `stats` is a struct with these fields: `Periods`, `NumPeriods`, `AveragePeriod`, `StandardDeviation`, and `NumOverruns`.

Here is a sample execution graphic using the default setting, `'slip'`, for the `OverrunAction` property in the `Rate` object. See `OverrunAction` for more information on overrun code execution.



The output of `statistics` is:

```
stats = 

              Periods: [0.7 0.11 0.7 0.11]
           NumPeriods: 4
        AveragePeriod: 0.09
    StandardDeviation: 0.0231
          NumOverruns: 2
```

Input Arguments

`rate` — Rate object
handle

`Rate` object, specified as an object handle. This object contains the information for the `DesiredRate` and other info about the execution. See `rateControlrateControl` for more information.

Output Arguments

`stats` — Time execution statistics
structure

Time execution statistics, returned as a structure. This structure contains the following fields:

- `Period` — All time periods (returned in seconds) used to calculate statistics as an indexed array. `stats.Period(end)` is the most recent period.
- `NumPeriods` — Number of elements in `Periods`
- `AveragePeriod` — Average time in seconds
- `StandardDeviation` — Standard deviation of all periods in seconds, centered around the mean stored in `AveragePeriod`
- `NumOverruns` — Number of periods with overrun

Examples

Get Statistics From Rate Object Execution

Create a `rateControl` object for running at 20 Hz.

```
r = rateControl(20);
```

Start a loop and control operation using the `rateControl` object.

```
for i = 1:30
    % Your code goes here
    waitfor(r);
end
```

Get `Rate` object statistics after loop operation.

```
stats = statistics(r)
```

```
stats = struct with fields:
              Periods: [0.0599 0.0418 0.0629 0.0449 0.0456 0.0591 0.0449 0.0461 0.0599 0.0449 0.0
           NumPeriods: 30
        AveragePeriod: 0.0501
    StandardDeviation: 0.0079
          NumOverruns: 0
```

Version History

Introduced in R2016a

See Also

`rateControl` | `rateControl` | `waitfor`

Topics

"Execute Code at a Fixed-Rate"

waitfor

Package: robotics

Pause code execution to achieve desired execution rate

Syntax

```
waitfor(rate)
numMisses = waitfor(rate)
```

Description

`waitfor(rate)` pauses execution until the code reaches the desired execution rate. The function accounts for the time that is spent executing code between `waitfor` calls.

`numMisses = waitfor(rate)` returns the number of iterations missed while executing code between calls.

Examples

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: 0.003114
Iteration: 2 - Time Elapsed: 1.003618
Iteration: 3 - Time Elapsed: 2.000382
Iteration: 4 - Time Elapsed: 3.000483
Iteration: 5 - Time Elapsed: 4.001145
Iteration: 6 - Time Elapsed: 5.011602
Iteration: 7 - Time Elapsed: 6.005804
Iteration: 8 - Time Elapsed: 7.000292
Iteration: 9 - Time Elapsed: 8.000162
Iteration: 10 - Time Elapsed: 9.006431
```

Each iteration executes at a 1-second interval.

Input Arguments

`rate` — `Rate` object
handle

`Rate` object, specified as a handle. This object contains the information for the desired rate and other information about the execution. See `rateControl` for more information.

Output Arguments

`numMisses` — Number of missed task executions
scalar

Number of missed task executions, returned as a scalar. `waitfor` returns the number of times the task was missed in the `Rate` object based on the `LastPeriod` time. For example, if the desired rate is 1 Hz and the last period was 3.2 seconds, `numMisses` returns 3.

Version History

Introduced in R2016a

See Also

`rateControl` | `rateControl`

Topics

"Execute Code at a Fixed-Rate"

addCollision

Add collision geometry to rigid body

Syntax

```
addCollision(body,type,parameters)
addCollision(body,collisionObj)
addCollision( ___ ,tform)
```

Description

`addCollision(body,type,parameters)` adds a collision geometry of the specified type `type` and geometric parameters `parameters` to the specified rigid body `body`.

`addCollision(body,collisionObj)` adds a collision geometry object to the rigid body `body`, specified as one of these collision objects:

- `collisionBox`
- `collisionCapsule`
- `collisionCylinder`
- `collisionSphere`
- `collisionMesh`

This syntax uses the `Pose` property of the specified collision object to transform the collision vertices into the rigid body frame.

`addCollision( ___ ,tform)` specifies a transformation for the collision geometry relative to the body frame in addition to any combination of input arguments from previous syntaxes.

Examples

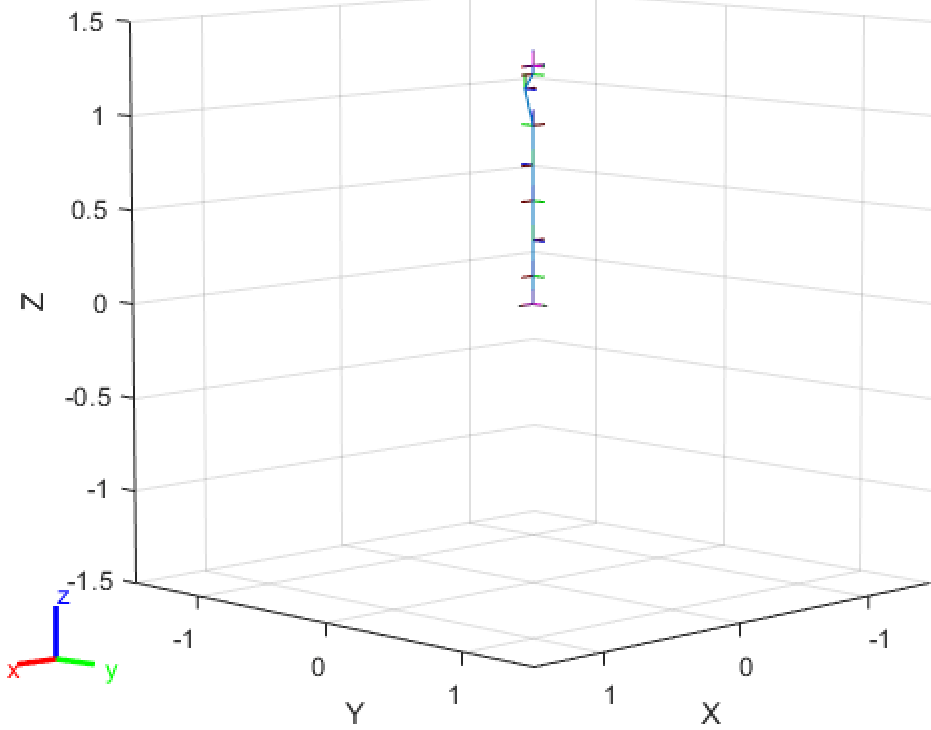
Add Collision Meshes and Check Collisions for Manipulator Robot Arm

Load a robot model and modify the collision meshes. Clear existing collision meshes, add simple collision object primitives, and check whether certain configurations are in collision.

Load Robot Model

Load a preconfigured robot model into the workspace using the `loadrobot` function. This model already has collision meshes specified for each body. Iterate through all the rigid body elements and clear the existing collision meshes. Confirm that the existing meshes are gone.

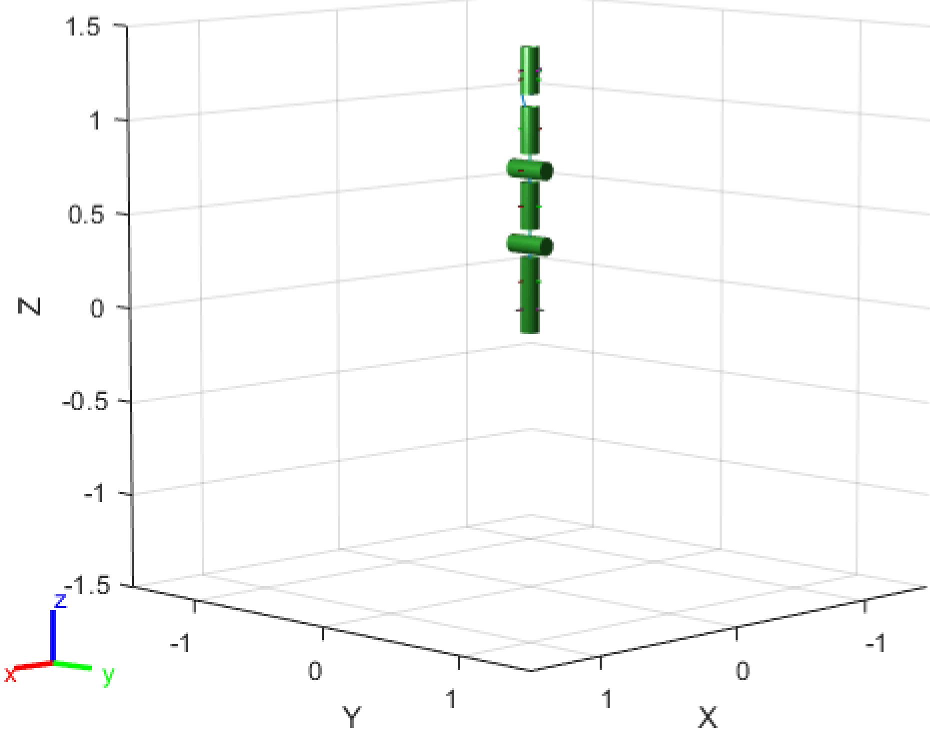
```
robot = loadrobot('kukaIiwa7','DataFormat','column');

for i = 1:robot.NumBodies
    clearCollision(robot.Bodies{i})
end

show(robot,'Collisions','on','Visuals','off');
```



Add Collision Cylinders

Iteratively add a collision cylinder to each body. Skip some bodies for this specific model, as they overlap and always collide with the end effector (body 10).

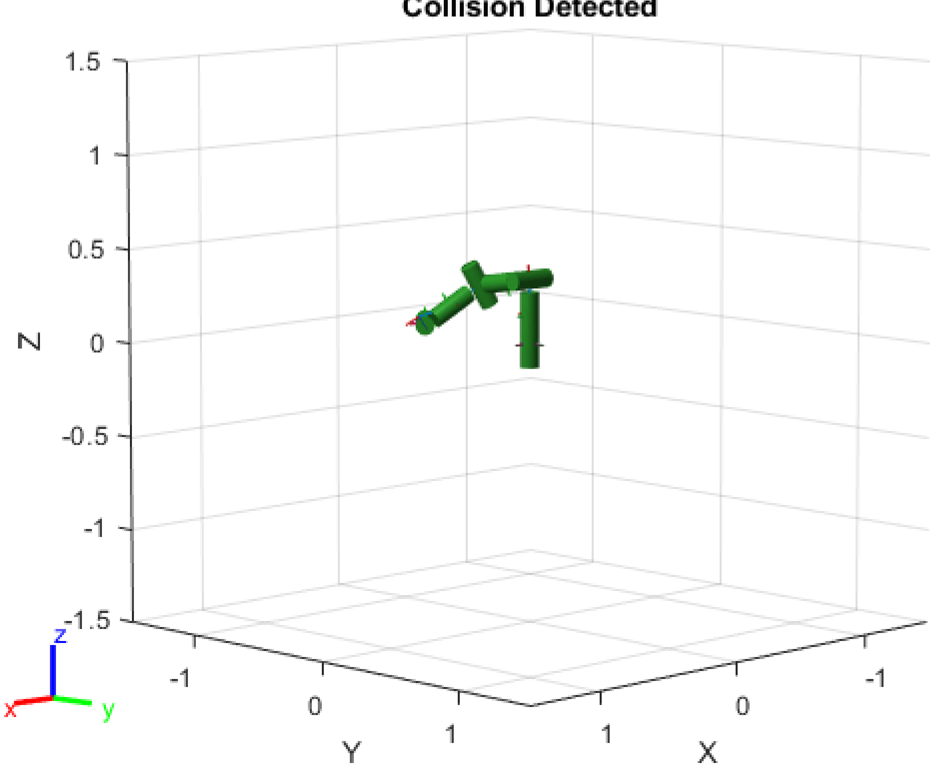
```
collisionObj = collisionCylinder(0.05,0.25);

for i = 1:robot.NumBodies
    if i > 6 && i < 10
        % Skip these bodies.
    else
        addCollision(robot.Bodies{i},collisionObj)
    end
end

show(robot,'Collisions','on','Visuals','off');
```



Check for Collisions

Generate a series of random configurations. Check whether the robot is in collision at each configuration. Visualize each configuration that has a collision.

```
figure
rng(0) % Set random seed for repeatability.
for i = 1:20
    config = randomConfiguration(robot);
    isColliding = checkCollision(robot,config,'SkippedSelfCollisions','parent');
    if isColliding
        show(robot,config,'Collisions','on','Visuals','off');
        title('Collision Detected')
    else
        % Skip non-collisions.
    end
end
```



Input Arguments

body — Rigid body
`rigidBody` object

Rigid body, specified as a `rigidBody` object.

type — Geometry type for collision geometry
`"box"` | `"cylinder"` | `"capsule"` | `"sphere"` | `"mesh"`

Geometry type for collision geometry, specified as a string scalar. The specified type determines the format of the `parameters` input.

- `"box"` — `[x y z]`
- `"cylinder"` — `[radius length]`
- `"capsule"` — `[radius length]`
- `"sphere"` — `radius`
- `"mesh"` — *n*-by-3 matrix of vertices or an STL or DAE file name as a string

Data Types: `char` | `string`

`parameters` — Collision geometry parameters
numeric vector | numeric matrix | string scalar

Collision geometry parameters, specified as a numeric vector, numeric matrix, or string scalar. The `type` input determines the format of this value.

- `"box"` — `[x y z]`
- `"cylinder"` — `[radius length]`
- `"capsule"` — `[radius length]`
- `"sphere"` — `radius`
- `"mesh"` — *n*-by-3 matrix of vertices or an STL or DAE file name as a string

Data Types: `single` | `double` | `char` | `string`

`collisionObj` — Collision geometry object
`collisionBox` object | `collisionCylinder` object | `collisionCapsule` object | `collisionSphere` object | `collisionMesh` object

Collision geometry object, specified as a `collisionBox`, `collisionCapsule`, `collisionCylinder`, `collisionSphere`, or `collisionMesh` object.

`tform` — Transformation of collision geometry
`eye(4)` (default) | 4-by-4 homogeneous transformation

Transformation of collision geometry, specified as a 4-by-4 homogeneous transformation. If specifying a collision object for the `collisionObj` input, this function applies the specified transformation to the `Pose` property of the specified collision object to transform the collision vertices into the rigid body frame.

Data Types: `single` | `double`

Version History

Introduced in R2020b

R2022b: `addCollision` supports adding capsule collision geometry

The `collisionObj` argument now accepts the `collisionCapsule` object.

The `type` and `parameters` arguments accept values of `"capsule"` and parameters in the form `[radius length]`, respectively.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`addVisual` | `checkCollision` | `clearCollision` | `clearVisual` | `show` | `rigidBodyTree`

addVisual

Add visual geometry data to rigid body

Syntax

```
addVisual(body,type,parameters)
addVisual( ___ ,tform)
```

Description

`addVisual(body,type,parameters)` adds the visuals of a geometry of the specified type `type` and geometric parameters `parameters` to the specified rigid body `body`.

`addVisual( ___ ,tform)` specifies a homogeneous transformation for the geometry visual relative to the body frame in addition to any combination of input arguments from previous syntaxes.

Input Arguments

body — `RigidBody` object
handle

`RigidBody` object, specified as a handle. Create a rigid body object using `rigidBody`.

`type` — Geometry type for geometry
`"box"` | `"cylinder"` | `"capsule"` | `"sphere"` | `"mesh"`

Geometry type for geometry, specified as a string scalar. The specified type determines the format of the `parameters` input.

- `"box"` — `[x y z]`
- `"cylinder"` — `[radius length]`
- `"capsule"` — `[radius length]`
- `"sphere"` — `radius`
- `"mesh"` — *n*-by-3 matrix of vertices or an STL or DAE file name as a string

Data Types: `char` | `string`

`parameters` — Geometry parameters
numeric vector | numeric matrix | string scalar

Geometry parameters, specified as a numeric vector, numeric matrix, or string scalar. The `type` input determines the format of this value.

- `"box"` — `[x y z]`
- `"cylinder"` — `[radius length]`
- `"capsule"` — `[radius length]`
- `"sphere"` — `radius`

- `"mesh"` — *n*-by-3 matrix of vertices or an STL or DAE file name as a string

Data Types: `single` | `double` | `char` | `string`

`tform` — Polygon mesh transformation
4-by-4 homogeneous transformation

Mesh transformation relative to the body coordinate frame, specified as a 4-by-4 homogeneous transformation.

Version History

Introduced in R2017b

R2023a: `addVisual` supports adding visual of primitive geometries

Use the `type` and `parameters` arguments to specify primitive geometries such as a box, sphere, cylinder, and capsule.

See Also

`addCollision` | `clearCollision` | `clearVisual` | `show` | `rigidBodyTree`

clearCollision

Clear all attached collision geometries

Syntax

`clearCollision(body)`

Description

`clearCollision(body)` clears all collision geometries attached to the specified rigid body object.

Examples

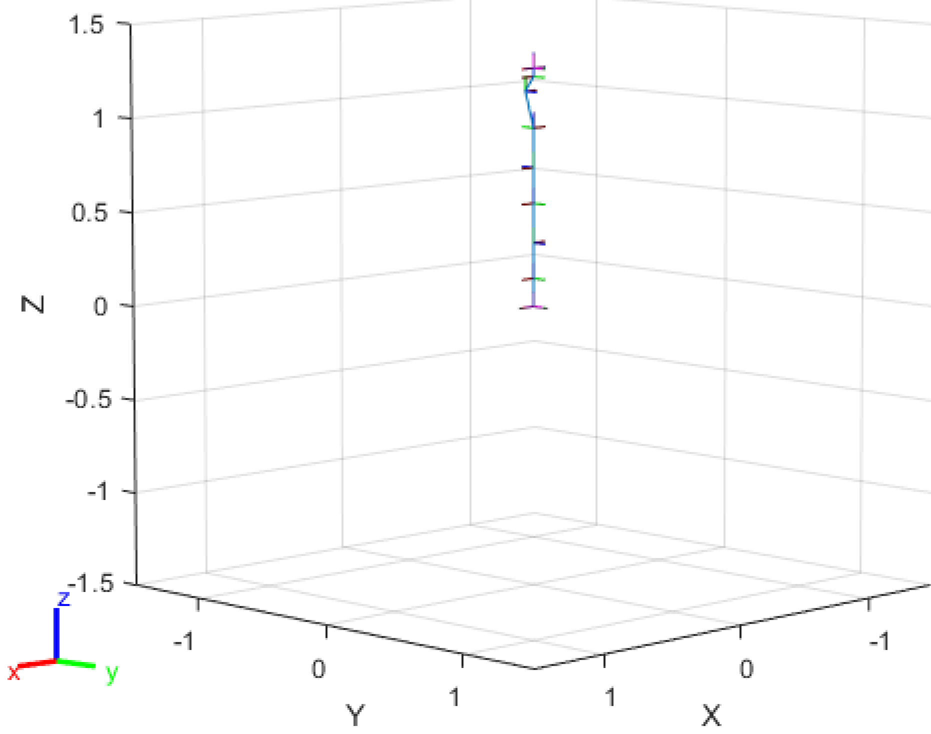
Add Collision Meshes and Check Collisions for Manipulator Robot Arm

Load a robot model and modify the collision meshes. Clear existing collision meshes, add simple collision object primitives, and check whether certain configurations are in collision.

Load Robot Model

Load a preconfigured robot model into the workspace using the `loadrobot` function. This model already has collision meshes specified for each body. Iterate through all the rigid body elements and clear the existing collision meshes. Confirm that the existing meshes are gone.

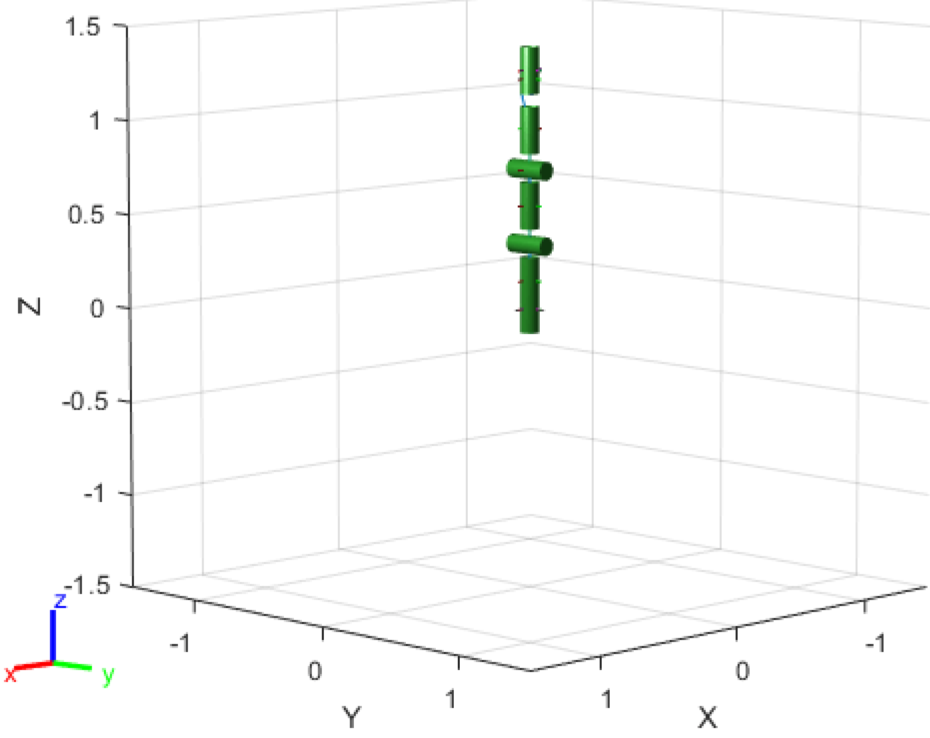
```
robot = loadrobot('kukaIiwa7','DataFormat','column');

for i = 1:robot.NumBodies
    clearCollision(robot.Bodies{i})
end

show(robot,'Collisions','on','Visuals','off');
```



Add Collision Cylinders

Iteratively add a collision cylinder to each body. Skip some bodies for this specific model, as they overlap and always collide with the end effector (body 10).

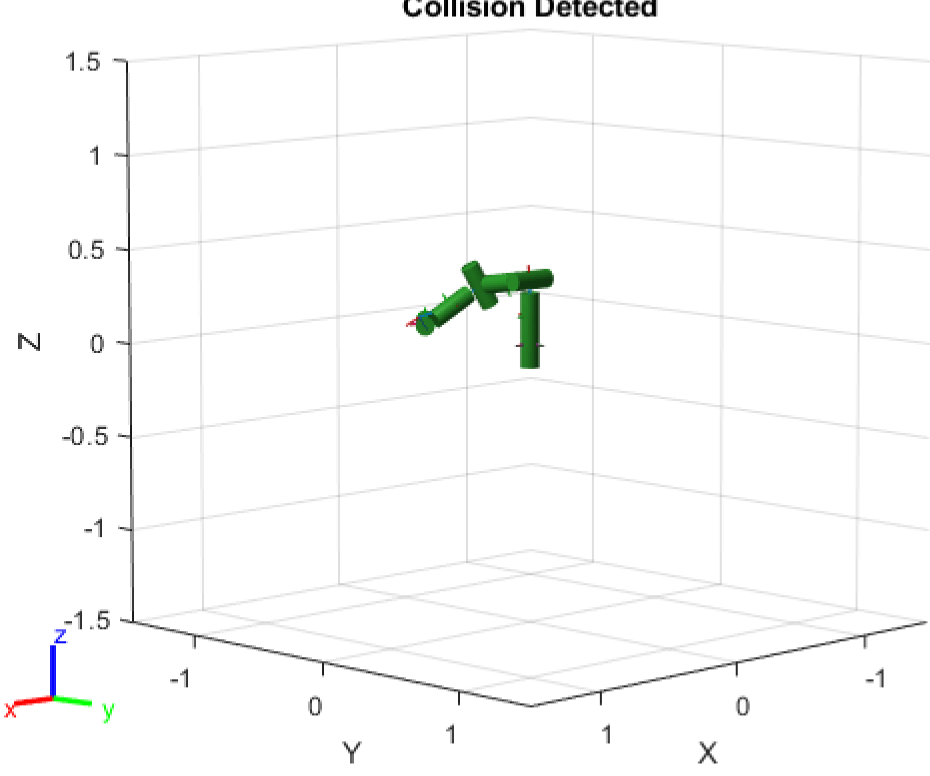
```
collisionObj = collisionCylinder(0.05,0.25);

for i = 1:robot.NumBodies
    if i > 6 && i < 10
        % Skip these bodies.
    else
        addCollision(robot.Bodies{i},collisionObj)
    end
end

show(robot,'Collisions','on','Visuals','off');
```



Check for Collisions

Generate a series of random configurations. Check whether the robot is in collision at each configuration. Visualize each configuration that has a collision.

```
figure
rng(0) % Set random seed for repeatability.
for i = 1:20
    config = randomConfiguration(robot);
    isColliding = checkCollision(robot,config,'SkippedSelfCollisions','parent');
    if isColliding
        show(robot,config,'Collisions','on','Visuals','off');
        title('Collision Detected')
    else
        % Skip non-collisions.
    end
end
```



Input Arguments

body — Rigid body
`rigidBody` object

Rigid body, specified as a `rigidBody` object.

Version History

Introduced in R2020b

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

`addVisual` | `addCollision` | `show` | `rigidBodyTree`

clearVisual

Clear all visual geometries

Syntax

```
clearVisual(body)
```

Description

`clearVisual(body)` clears all visual geometries attached to the given rigid body object.

Input Arguments

body — Rigid body
`rigidBody` object

Rigid body, specified as a `rigidBody` object.

Version History

Introduced in R2017b

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

`addVisual` | `addCollision` | `clearCollision` | `show` | `rigidBodyTree`

copy

Create a deep copy of rigid body

Syntax

```
copyObj = copy(bodyObj)
```

Description

`copyObj = copy(bodyObj)` creates a copy of the rigid body object with the same properties.

Input Arguments

`bodyObj` — RigidBody object
handle

`RigidBody` object, specified as a handle. Create a rigid body object using `rigidBody`.

Output Arguments

`copyObj` — RigidBody object
handle

`RigidBody` object, returned as a handle. Create a rigid body object using `rigidBody`.

Version History

Introduced in R2016b

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`rigidBodyJoint` | `rigidBodyTree`

copy

Create copy of joint

Syntax

```
jCopy = copy(jointObj)
```

Description

`jCopy = copy(jointObj)` creates a copy of the `rigidBodyJoint` object with the same properties.

Input Arguments

`jointObj` — `rigidBodyJoint` object
handle

`rigidBodyJoint` object, specified as a handle.

Output Arguments

`jCopy` — `rigidBodyJoint` object
handle

`rigidBodyJoint` object, returned as a handle. This copy has the same properties.

Version History

Introduced in R2016b

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

`rigidBodyJoint` | `rigidBody` | `rigidBodyTree`

setFixedTransform

Set fixed transform properties of joint

Syntax

```
setFixedTransform(jointObj,tform)
```

```
setFixedTransform(jointObj,dhparams,"dh")
setFixedTransform(jointObj,mdhparams,"mdh")
```

Description

`setFixedTransform(jointObj,tform)` sets the `JointToParentTransform` property of the `rigidBodyJoint` object directly with the specified homogenous transformation, `tform`.

`setFixedTransform(jointObj,dhparams,"dh")` sets the `ChildToJointTransform` property using Denavit-Hartenberg (DH) parameters. The `JointToParentTransform` property is set to an identity matrix. DH parameters are given in the order `[a alpha d theta]`.

For revolute joints, the `theta` input is ignored when specifying the fixed transformation between joints because that angle is dependent on the joint configuration. For prismatic joints, the `d` input is ignored. For more information, see "Rigid Body Tree Robot Model".

`setFixedTransform(jointObj,mdhparams,"mdh")` sets the `JointToParentTransform` property using modified DH parameters. The `ChildToJointTransform` property is set to an identity matrix. Modified DH parameters are given in the order `[a alpha d theta]`.

Examples

Build Manipulator Robot Using Denavit-Hartenberg Parameters

Use the Denavit-Hartenberg (DH) parameters of the Puma560® robot to build a robot. Each rigid body is added one at a time, with the child-to-parent transform specified by the joint object.

The DH parameters define the geometry of the robot with relation to how each rigid body is attached to its parent. For convenience, setup the parameters for the Puma560 robot in a matrix[1] on page 3-345. The Puma robot is a serial chain manipulator. The DH parameters are relative to the previous row in the matrix, corresponding to the previous joint attachment.

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

Create a rigid body tree object to build the robot.

```
robot = rigidBodyTree;
```

Create the first rigid body and add it to the robot. To add a rigid body:

1. Create a `rigidBody` object and give it a unique name.
2. Create a `rigidBodyJoint` object and give it a unique name.
3. Use `setFixedTransform` to specify the body-to-body transformation using DH parameters. The last element of the DH parameters, `theta`, is ignored because the angle is dependent on the joint position.
4. Call `addBody` to attach the first body joint to the base frame of the robot.

```
body1 = rigidBody('body1');
jnt1 = rigidBodyJoint('jnt1','revolute');

setFixedTransform(jnt1,dhparams(1,:),'dh');
body1.Joint = jnt1;

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

Create and add other rigid bodies to the robot. Specify the previous body name when calling `addBody` to attach it. Each fixed transform is relative to the previous joint coordinate frame.

```
body2 = rigidBody('body2');
jnt2 = rigidBodyJoint('jnt2','revolute');
body3 = rigidBody('body3');
jnt3 = rigidBodyJoint('jnt3','revolute');
body4 = rigidBody('body4');
jnt4 = rigidBodyJoint('jnt4','revolute');
body5 = rigidBody('body5');
jnt5 = rigidBodyJoint('jnt5','revolute');
body6 = rigidBody('body6');
jnt6 = rigidBodyJoint('jnt6','revolute');

setFixedTransform(jnt2,dhparams(2,:),'dh');
setFixedTransform(jnt3,dhparams(3,:),'dh');
setFixedTransform(jnt4,dhparams(4,:),'dh');
setFixedTransform(jnt5,dhparams(5,:),'dh');
setFixedTransform(jnt6,dhparams(6,:),'dh');

body2.Joint = jnt2;
body3.Joint = jnt3;
body4.Joint = jnt4;
body5.Joint = jnt5;
body6.Joint = jnt6;

addBody(robot,body2,'body1')
addBody(robot,body3,'body2')
addBody(robot,body4,'body3')
addBody(robot,body5,'body4')
addBody(robot,body6,'body5')
```

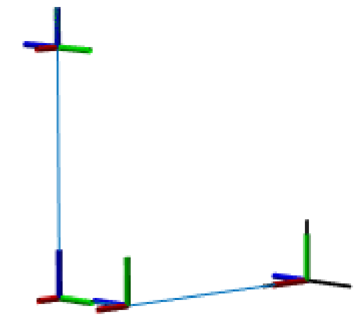
Verify that your robot was built properly by using the `showdetails` or `show` function. `showdetails` lists all the bodies in the MATLAB® command window. `show` displays the robot with a given configuration (home by default). Calls to `axis` modify the axis limits and hide the axis labels.

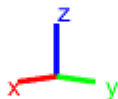
```
showdetails(robot)
```

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

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

```
show(robot);
axis([-0.5,0.5,-0.5,0.5,-0.5,0.5])
axis off
```





References

[1] Corke, P. I., and B. Armstrong-Helouvry. “A Search for Consensus among Model Parameters Reported for the PUMA 560 Robot.” *Proceedings of the 1994 IEEE International Conference on*

Robotics and Automation, IEEE Computer. Soc. Press, 1994, pp. 1608–13. *DOI.org (Crossref)*, doi:10.1109/ROBOT.1994.351360.

Input Arguments

`jointObj` — `rigidBodyJoint` object
handle

`rigidBodyJoint` object, specified as a handle.

`tform` — Homogeneous transform
4-by-4 matrix | `se3` object

Homogeneous transform, specified as a 4-by-4 matrix or an `se3` object. The transform is set to the `ChildToJointTransform` property. The `JointToParentTransform` property is set to an identity matrix.

`dhparams` — Denavit-Hartenberg (DH) parameters
four-element vector

Denavit-Hartenberg (DH) parameters, specified as a four-element vector, `[a alpha d theta]`. These parameters are used to set the `ChildToJointTransform` property. The `JointToParentTransform` property is set to an identity matrix.

The `theta` input is ignored when specifying the fixed transformation between joints because that angle is dependent on the joint configuration. For more information, see “Rigid Body Tree Robot Model”.

`mdhparams` — Modified Denavit-Hartenberg (DH) parameters
four-element vector

Modified Denavit-Hartenberg (DH) parameters, specified as a four-element vector, `[a alpha d theta]`. These parameters are used to set the `JointToParentTransform` property. The `ChildToJointTransform` is set to an identity matrix.

The `theta` input is ignored when specifying the fixed transformation between joints because that angle is dependent on the joint configuration. For more information, see “Rigid Body Tree Robot Model”.

Version History

Introduced in R2016b

R2023a: `tform` argument supports `se3` transformation object

You can now specify the `tform` argument as an `se3` transformation object.

References

[1] Craig, John J. *Introduction to Robotics: Mechanics and Control*. Reading, MA: Addison-Wesley, 1989.

[2] Siciliano, Bruno. *Robotics: Modelling, Planning and Control*. London: Springer, 2009.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

`rigidBodyJoint` | `rigidBody` | `rigidBodyTree`

addBody

Add body to robot

Syntax

`addBody(robot,body,parentname)`

Description

`addBody(robot,body,parentname)` adds a rigid body to the robot object and is attached to the rigid body parent specified by `parentname`. The `body` property defines how this body moves relative to the parent body.

Examples

Attach Rigid Body and Joint to Rigid Body Tree

Add a rigid body and corresponding joint to a rigid body tree. Each `rigidBody` object contains a `rigidBodyJoint` object and must be added to the `rigidBodyTree` using `addBody`.

Create a rigid body tree.

```
rbtree = rigidBodyTree;
```

Create a rigid body with a unique name.

```
body1 = rigidBody('b1');
```

Create a revolute joint. By default, the `rigidBody` object comes with a fixed joint. Replace the joint by assigning a new `rigidBodyJoint` object to the `body1.Joint` property.

```
jnt1 = rigidBodyJoint('jnt1','revolute');
body1.Joint = jnt1;
```

Add the rigid body to the tree. Specify the body name that you are attaching the rigid body to. Because this is the first body, use the base name of the tree.

```
basename = rbtree.BaseName;
addBody(rbtree,body1,basename)
```

Use `showdetails` on the tree to confirm the rigid body and joint were added properly.

```
showdetails(rbtree)
```

```
--------------------
Robot: (1 bodies)

 Idx    Body Name   Joint Name   Joint Type    Parent Name(Idx)   Children Name(s)
 ---    ---------   ----------   ----------    ----------------   ----------------
   1           b1         jnt1     revolute             base(0)   
--------------------
```

Build Manipulator Robot Using Denavit-Hartenberg Parameters

Use the Denavit-Hartenberg (DH) parameters of the Puma560® robot to build a robot. Each rigid body is added one at a time, with the child-to-parent transform specified by the joint object.

The DH parameters define the geometry of the robot with relation to how each rigid body is attached to its parent. For convenience, setup the parameters for the Puma560 robot in a matrix[1] on page 3-351. The Puma robot is a serial chain manipulator. The DH parameters are relative to the previous row in the matrix, corresponding to the previous joint attachment.

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

Create a rigid body tree object to build the robot.

```
robot = rigidBodyTree;
```

Create the first rigid body and add it to the robot. To add a rigid body:

1. Create a `rigidBody` object and give it a unique name.
2. Create a `rigidBodyJoint` object and give it a unique name.
3. Use `setFixedTransform` to specify the body-to-body transformation using DH parameters. The last element of the DH parameters, `theta`, is ignored because the angle is dependent on the joint position.
4. Call `addBody` to attach the first body joint to the base frame of the robot.

```
body1 = rigidBody('body1');
jnt1 = rigidBodyJoint('jnt1','revolute');

setFixedTransform(jnt1,dhparams(1,:),'dh');
body1.Joint = jnt1;

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

Create and add other rigid bodies to the robot. Specify the previous body name when calling `addBody` to attach it. Each fixed transform is relative to the previous joint coordinate frame.

```
body2 = rigidBody('body2');
jnt2 = rigidBodyJoint('jnt2','revolute');
body3 = rigidBody('body3');
jnt3 = rigidBodyJoint('jnt3','revolute');
body4 = rigidBody('body4');
jnt4 = rigidBodyJoint('jnt4','revolute');
body5 = rigidBody('body5');
jnt5 = rigidBodyJoint('jnt5','revolute');
body6 = rigidBody('body6');
jnt6 = rigidBodyJoint('jnt6','revolute');

setFixedTransform(jnt2,dhparams(2,:),'dh');
```

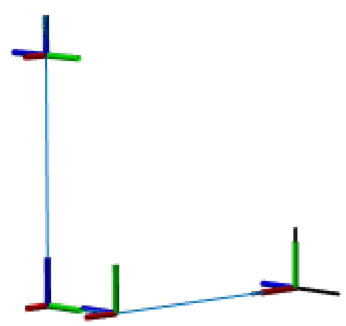
```
setFixedTransform(jnt3,dhparams(3,:),'dh');
setFixedTransform(jnt4,dhparams(4,:),'dh');
setFixedTransform(jnt5,dhparams(5,:),'dh');
setFixedTransform(jnt6,dhparams(6,:),'dh');

body2.Joint = jnt2;
body3.Joint = jnt3;
body4.Joint = jnt4;
body5.Joint = jnt5;
body6.Joint = jnt6;

addBody(robot,body2,'body1')
addBody(robot,body3,'body2')
addBody(robot,body4,'body3')
addBody(robot,body5,'body4')
addBody(robot,body6,'body5')
```

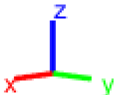
Verify that your robot was built properly by using the `showdetails` or `show` function. `showdetails` lists all the bodies in the MATLAB® command window. `show` displays the robot with a given configuration (home by default). Calls to `axis` modify the axis limits and hide the axis labels.

```
showdetails(robot)
```

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

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

```
show(robot);
axis([-0.5,0.5,-0.5,0.5,-0.5,0.5])
axis off
```





References

[1] Corke, P. I., and B. Armstrong-Helouvry. "A Search for Consensus among Model Parameters Reported for the PUMA 560 Robot." *Proceedings of the 1994 IEEE International Conference on Robotics and Automation*, IEEE Computer. Soc. Press, 1994, pp. 1608–13. *DOI.org (Crossref)*, doi:10.1109/ROBOT.1994.351360.

Modify a Robot Rigid Body Tree Model

Make changes to an existing `rigidBodyTree` object. You can get replace joints, bodies and subtrees in the rigid body tree.

Load example robots as `rigidBodyTree` objects.

```
load exampleRobots.mat
```

View the details of the Puma robot using `showdetails`.

```
showdetails(puma1)
```

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

```
 Idx    Body Name   Joint Name   Joint Type    Parent Name(Idx)   Children Name(s)
 ---    ---------   ----------   ----------    ----------------   ----------------
   1           L1         jnt1     revolute             base(0)   L2(2)
   2           L2         jnt2     revolute               L1(1)   L3(3)
   3           L3         jnt3     revolute               L2(2)   L4(4)
   4           L4         jnt4     revolute               L3(3)   L5(5)
   5           L5         jnt5     revolute               L4(4)   L6(6)
   6           L6         jnt6     revolute               L5(5)
--------------------
```

Get a specific body to inspect the properties. The only child of the `L3` body is the `L4` body. You can copy a specific body as well.

```
body3 = getBody(puma1,'L3');
childBody = body3.Children{1}
```

```
childBody = 
  rigidBody with properties:

            Name: 'L4'
           Joint: [1x1 rigidBodyJoint]
            Mass: 1
    CenterOfMass: [0 0 0]
         Inertia: [1 1 1 0 0 0]
          Parent: [1x1 rigidBody]
        Children: {[1x1 rigidBody]}
         Visuals: {}
      Collisions: {}
```

```
body3Copy = copy(body3);
```

Replace the joint on the `L3` body. You must create a new `Joint` object and use `replaceJoint` to ensure the downstream body geometry is unaffected. Call `setFixedTransform` if necessary to define a transform between the bodies instead of with the default identity matrices.

```
newJoint = rigidBodyJoint('prismatic');
replaceJoint(puma1,'L3',newJoint);

showdetails(puma1)
```

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

 Idx    Body Name       Joint Name       Joint Type    Parent Name(Idx)   Children Name(s)
 ---    ---------       ----------       ----------    ----------------   ----------------
   1           L1             jnt1         revolute             base(0)   L2(2)
   2           L2             jnt2         revolute               L1(1)   L3(3)
   3           L3        prismatic            fixed               L2(2)   L4(4)
   4           L4             jnt4         revolute               L3(3)   L5(5)
   5           L5             jnt5         revolute               L4(4)   L6(6)
   6           L6             jnt6         revolute               L5(5)
--------------------
```

Remove an entire body and get the resulting subtree using `removeBody`. The removed body is included in the subtree.

```
subtree = removeBody(puma1,'L4')
```

```
subtree = 
  rigidBodyTree with properties:

     NumBodies: 3
        Bodies: {[1x1 rigidBody]  [1x1 rigidBody]  [1x1 rigidBody]}
          Base: [1x1 rigidBody]
     BodyNames: {'L4'  'L5'  'L6'}
      BaseName: 'L3'
       Gravity: [0 0 0]
    DataFormat: 'struct'
```

Remove the modified `L3` body. Add the original copied `L3` body to the `L2` body, followed by the returned subtree. The robot model remains the same. See a detailed comparison through `showdetails`.

```
removeBody(puma1,'L3');
addBody(puma1,body3Copy,'L2')
addSubtree(puma1,'L3',subtree)

showdetails(puma1)
```

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

 Idx    Body Name   Joint Name   Joint Type    Parent Name(Idx)   Children Name(s)
 ---    ---------   ----------   ----------    ----------------   ----------------
   1           L1         jnt1     revolute             base(0)   L2(2)  
   2           L2         jnt2     revolute               L1(1)   L3(3)  
   3           L3         jnt3     revolute               L2(2)   L4(4)  
   4           L4         jnt4     revolute               L3(3)   L5(5)  
   5           L5         jnt5     revolute               L4(4)   L6(6)  
   6           L6         jnt6     revolute               L5(5)   
--------------------
```

Input Arguments

`robot` — Robot model
`rigidBodyTree` object

Robot model, specified as a `rigidBodyTree` object.

`body` — Rigid body
`rigidBody` object

Rigid body, specified as a `rigidBody` object.

`parentname` — Parent body name
string scalar | character vector

Parent body name, specified as a string scalar or character vector. This parent body must already exist in the robot model. The new body is attached to this parent body.

Data Types: `char` | `string`

Version History

Introduced in R2016b

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

When creating the `rigidBodyTree` object, use the syntax that specifies the `MaxNumBodies` as an upper bound for adding bodies to the robot model. You must also specify the `DataFormat` property as a name-value pair. For example:

```
robot = rigidBodyTree("MaxNumBodies",15,"DataFormat","row")
```

To minimize data usage, limit the upper bound to a number close to the expected number of bodies in the model. All data formats are supported for code generation. To use the dynamics functions, the data format must be set to `"row"` or `"column"`.

The `show` and `showdetails` functions do not support code generation.

See Also

`rigidBodyJoint` | `rigidBody` | `removeBody` | `replaceBody`

addSubtree

Add subtree to robot

Syntax

```
addSubtree(robot,parentname,subtree)
```

Description

`addSubtree(robot,parentname,subtree)` attaches the robot model, `newSubtree`, to an existing robot model, `robot`, at the body specified by `parentname`. The subtree base is not added as a body.

Examples

Modify a Robot Rigid Body Tree Model

Make changes to an existing `rigidBodyTree` object. You can get replace joints, bodies and subtrees in the rigid body tree.

Load example robots as `rigidBodyTree` objects.

```
load exampleRobots.mat
```

View the details of the Puma robot using `showdetails`.

```
showdetails(puma1)
```

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

 Idx     Body Name   Joint Name   Joint Type    Parent Name(Idx)   Children Name(s)
 ---     ---------   ----------   ----------    ----------------   ----------------
   1            L1         jnt1     revolute             base(0)   L2(2)
   2            L2         jnt2     revolute               L1(1)   L3(3)
   3            L3         jnt3     revolute               L2(2)   L4(4)
   4            L4         jnt4     revolute               L3(3)   L5(5)
   5            L5         jnt5     revolute               L4(4)   L6(6)
   6            L6         jnt6     revolute               L5(5)
--------------------
```

Get a specific body to inspect the properties. The only child of the `L3` body is the `L4` body. You can copy a specific body as well.

```
body3 = getBody(puma1,'L3');
childBody = body3.Children{1}
```

```
childBody = 
  rigidBody with properties:

            Name: 'L4'
           Joint: [1x1 rigidBodyJoint]
```

```
            Mass: 1
    CenterOfMass: [0 0 0]
         Inertia: [1 1 1 0 0 0]
          Parent: [1x1 rigidBody]
        Children: {[1x1 rigidBody]}
         Visuals: {}
      Collisions: {}
```

```
body3Copy = copy(body3);
```

Replace the joint on the L3 body. You must create a new Joint object and use replaceJoint to ensure the downstream body geometry is unaffected. Call setFixedTransform if necessary to define a transform between the bodies instead of with the default identity matrices.

```
newJoint = rigidBodyJoint('prismatic');
replaceJoint(puma1,'L3',newJoint);

showdetails(puma1)
```

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

 Idx    Body Name       Joint Name       Joint Type    Parent Name(Idx)   Children Name(s)
 ---    ---------       ----------       ----------    ----------------   ----------------
   1           L1             jnt1         revolute             base(0)   L2(2)
   2           L2             jnt2         revolute               L1(1)   L3(3)
   3           L3        prismatic            fixed               L2(2)   L4(4)
   4           L4             jnt4         revolute               L3(3)   L5(5)
   5           L5             jnt5         revolute               L4(4)   L6(6)
   6           L6             jnt6         revolute               L5(5)
--------------------
```

Remove an entire body and get the resulting subtree using removeBody. The removed body is included in the subtree.

```
subtree = removeBody(puma1,'L4')
```

```
subtree = 
  rigidBodyTree with properties:

     NumBodies: 3
        Bodies: {[1x1 rigidBody]  [1x1 rigidBody]  [1x1 rigidBody]}
          Base: [1x1 rigidBody]
     BodyNames: {'L4'  'L5'  'L6'}
      BaseName: 'L3'
       Gravity: [0 0 0]
    DataFormat: 'struct'
```

Remove the modified L3 body. Add the original copied L3 body to the L2 body, followed by the returned subtree. The robot model remains the same. See a detailed comparison through showdetails.

```
removeBody(puma1,'L3');
addBody(puma1,body3Copy,'L2')
addSubtree(puma1,'L3',subtree)

showdetails(puma1)
```

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

 Idx     Body Name   Joint Name   Joint Type    Parent Name(Idx)   Children Name(s)
 ---     ---------   ----------   ----------    ----------------   ----------------
   1            L1         jnt1     revolute             base(0)   L2(2)
   2            L2         jnt2     revolute               L1(1)   L3(3)
   3            L3         jnt3     revolute               L2(2)   L4(4)
   4            L4         jnt4     revolute               L3(3)   L5(5)
   5            L5         jnt5     revolute               L4(4)   L6(6)
   6            L6         jnt6     revolute               L5(5)
--------------------
```

Input Arguments

robot — Robot model
RigidBodyTree object

Robot model, specified as a rigidBodyTree object.

parentname — Parent body name
string scalar | character vector

Parent body name, specified as a string scalar or character vector. This parent body must already exist in the robot model. The new body is attached to this parent body.

Data Types: char | string

subtree — Subtree robot model
rigidBodyTree object

Subtree robot model, specified as a rigidBodyTree object.

Version History

Introduced in R2016b

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

When creating the rigidBodyTree object, use the syntax that specifies the MaxNumBodies as an upper bound for adding bodies to the robot model. You must also specify the DataFormat property as a name-value pair. For example:

```
robot = rigidBodyTree("MaxNumBodies",15,"DataFormat","row")
```

To minimize data usage, limit the upper bound to a number close to the expected number of bodies in the model. All data formats are supported for code generation. To use the dynamics functions, the data format must be set to "row" or "column".

The `show` and `showdetails` functions do not support code generation.

See Also

`rigidBodyJoint` | `rigidBody` | `addBody` | `removeBody` | `replaceBody`

centerOfMass

Center of mass position and Jacobian

Syntax

```
com = centerOfMass(robot)
com = centerOfMass(robot,configuration)
[com,comJac] = centerOfMass(robot,configuration)
```

Description

`com = centerOfMass(robot)` computes the center of mass position of the robot model at its home configuration, relative to the base frame.

`com = centerOfMass(robot,configuration)` computes the center of mass position of the robot model at the specified joint configuration, relative to the base frame.

`[com,comJac] = centerOfMass(robot,configuration)` also returns the center of mass Jacobian, which relates the center of mass velocity to the joint velocities.

Examples

Calculate Center of Mass and Jacobian for Robot Configuration

Load a predefined KUKA LBR robot model, which is specified as a `RigidBodyTree` object.

```
load exampleRobots.mat lbr
```

Set the data format to `'row'`. For all dynamics calculations, the data format must be either `'row'` or `'column'`.

```
lbr.DataFormat = 'row';
```

Compute the center of mass position and Jacobian at the home configuration of the robot.

```
[comLocation,comJac] = centerOfMass(lbr);
```

Input Arguments

`robot` — Robot model
`rigidBodyTree` object

Robot model, specified as a `rigidBodyTree` object. To use the `centerOfMass` function, set the `DataFormat` property to either `'row'` or `'column'`.

`configuration` — Robot configuration
vector

Robot configuration, specified as a vector with positions for all nonfixed joints in the robot model. You can generate a configuration using `homeConfiguration(robot)`, `randomConfiguration(robot)`, or by specifying your own joint positions. To use the vector form of `configuration`, set the `DataFormat` property for the `robot` to either `'row'` or `'column'` .

Output Arguments

com — Center of mass location
[x y z] vector

Center of mass location, returned as an `[x y z]` vector. The vector describes the location of the center of mass for the specified `configuration` relative to the body frame, in meters.

comJac — Center of mass Jacobian
3-by-*n* matrix

Center of mass Jacobian, returned as a 3-by-*n* matrix, where *n* is the robot velocity degrees of freedom.

Version History

Introduced in R2017a

References

[1] Featherstone, Roy. *Rigid Body Dynamics Algorithms*. Springer US, 2008. DOI.org (Crossref), doi:10.1007/978-1-4899-7560-7.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

When creating the `rigidBodyTree` object, use the syntax that specifies the `MaxNumBodies` as an upper bound for adding bodies to the robot model. You must also specify the `DataFormat` property as a name-value pair. For example:

```
robot = rigidBodyTree("MaxNumBodies",15,"DataFormat","row")
```

To minimize data usage, limit the upper bound to a number close to the expected number of bodies in the model. All data formats are supported for code generation. To use the dynamics functions, the data format must be set to `"row"` or `"column"`.

The `show` and `showdetails` functions do not support code generation.

See Also

`rigidBodyTree` | `massMatrix` | `velocityProduct` | `gravityTorque`

Topics
"Robot Dynamics"

checkCollision

Check if robot is in collision

Syntax

`[isSelfColliding,selfSeparationDist,selfWitnessPts] = checkCollision(robot, config)`

`[isColliding,separationDist,witnessPts] = checkCollision(robot,config, worldObjects)`

`[ ___ ] = checkCollision( ___ ,Name,Value)`

Description

`[isSelfColliding,selfSeparationDist,selfWitnessPts] = checkCollision(robot, config)` checks if the specified rigid body tree robot model `robot` is in self-collision at the specified configuration `config`. Add collision objects to the rigid body tree robot model using the `addCollision` function. The `checkCollision` function also returns the closest separation distance `selfSeparationDist` and the witness points `selfWitnessPts` as points on each body.

The function ignores adjacent bodies when checking for self-collisions.

`[isColliding,separationDist,witnessPts] = checkCollision(robot,config, worldObjects)` checks if the specified rigid body tree robot model is in collision with itself or a specified set of collision objects in the world `worldObjects`.

`[ ___ ] = checkCollision( ___ ,Name,Value)` specifies additional options using one or more name-value pair arguments in addition to any of argument combinations from previous syntaxes.

Examples

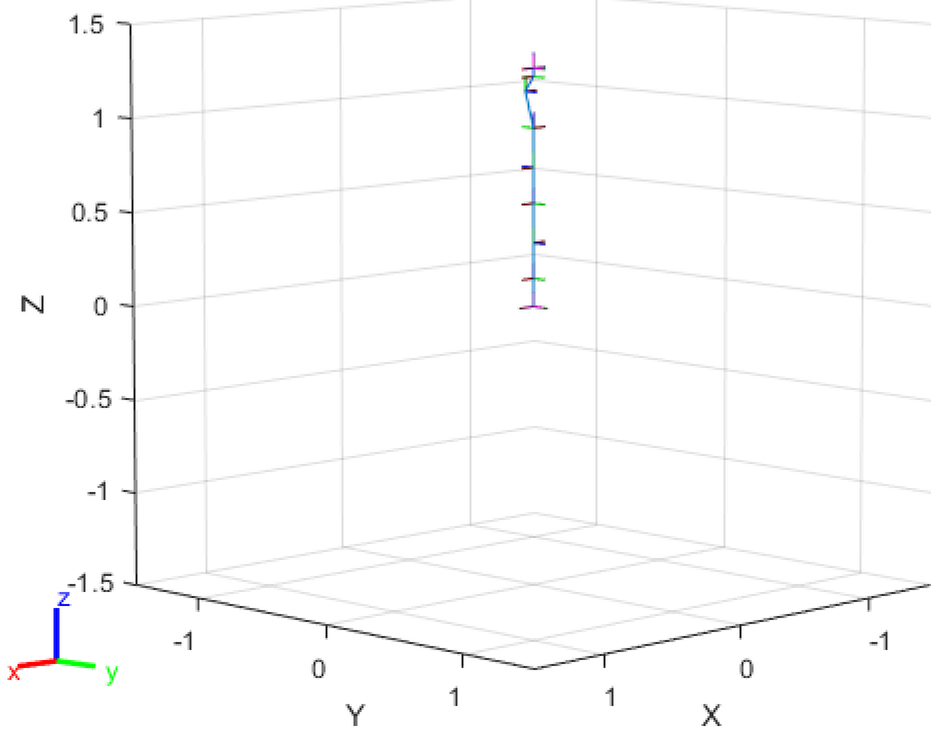
Add Collision Meshes and Check Collisions for Manipulator Robot Arm

Load a robot model and modify the collision meshes. Clear existing collision meshes, add simple collision object primitives, and check whether certain configurations are in collision.

Load Robot Model

Load a preconfigured robot model into the workspace using the `loadrobot` function. This model already has collision meshes specified for each body. Iterate through all the rigid body elements and clear the existing collision meshes. Confirm that the existing meshes are gone.

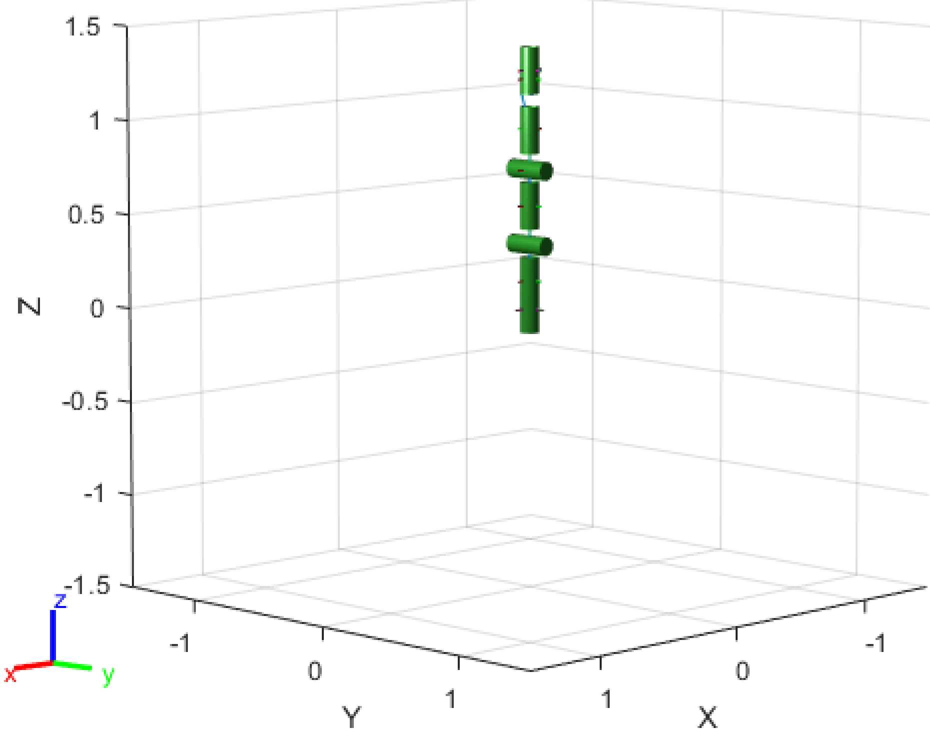
```
robot = loadrobot('kukaIiwa7','DataFormat','column');

for i = 1:robot.NumBodies
    clearCollision(robot.Bodies{i})
end

show(robot,'Collisions','on','Visuals','off');
```



Add Collision Cylinders

Iteratively add a collision cylinder to each body. Skip some bodies for this specific model, as they overlap and always collide with the end effector (body 10).

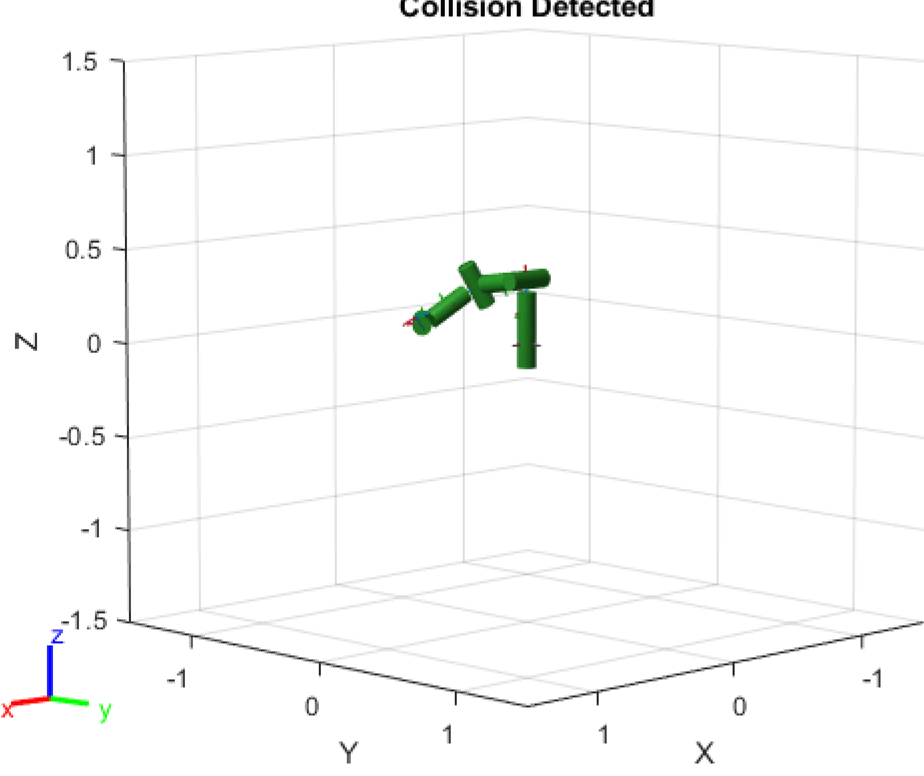
```
collisionObj = collisionCylinder(0.05,0.25);

for i = 1:robot.NumBodies
    if i > 6 && i < 10
        % Skip these bodies.
    else
        addCollision(robot.Bodies{i},collisionObj)
    end
end

show(robot,'Collisions','on','Visuals','off');
```



Check for Collisions

Generate a series of random configurations. Check whether the robot is in collision at each configuration. Visualize each configuration that has a collision.

```
figure
rng(0) % Set random seed for repeatability.
for i = 1:20
    config = randomConfiguration(robot);
    isColliding = checkCollision(robot,config,'SkippedSelfCollisions','parent');
    if isColliding
        show(robot,config,'Collisions','on','Visuals','off');
        title('Collision Detected')
    else
        % Skip non-collisions.
    end
end
```



Change Self-Collision Checking Behavior

This example shows how to change which rigid body pairs are skipped during self-collision checking in rigid body trees using the `SkippedSelfCollisions` name-value argument for `checkCollision`.

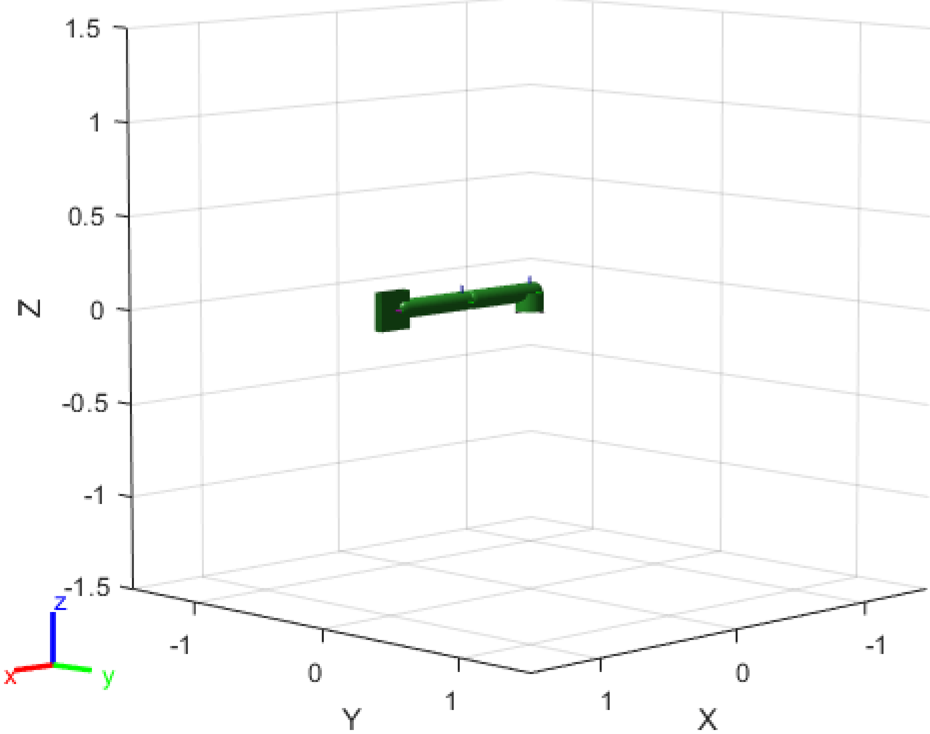
Serial Manipulator Robot

Load a serial manipulator robot represented as a two joint rigid body tree. Since this robot does not have collision geometries, add some primitive collision geometries.

```
rbt2j = twoJointRigidBodyTree;
P = [0.05 0.45]; % Geometry parameters for capsules
T = trvec2tform([0.2 0 0]) * eul2tform([0 pi/2 0],"XYZ"); % Transformation parameters for capsule
addCollision(rbt2j.Base,"cylinder",[0.075 0.1],trvec2tform([0 0 0.05]))
addCollision(rbt2j.Bodies{1},"capsule",P,T)
addCollision(rbt2j.Bodies{2},"capsule",P,T)
addCollision(rbt2j.Bodies{3},"box",[0.2 0.05 0.2])
```

Visualize the robot with collisions on.

```
show(rbt2j,homeConfiguration(rbt2j),Collisions="on");
```



By default, `SkippedSelfCollisions` is `"parent"`, so self-collision checking skips collisions between parent and child bodies. Check the parent and child bodies of body2.

```
body2 = rbt2j.Bodies{2};
rbt2j.BodyNames
```

```
ans = 1x3 cell
    {'body1'}    {'body2'}    {'tool'}
```

```
body2.Parent.Name
```

```
ans = 
'body1'
```

```
body2.Children{1}.Name
```

```
ans = 
'tool'
```

This means that `"body2"` is not checked for collisions against `"body1"` or `"tool"`.

List the body names of the robot. This shows that in the cell array, `"body2"`, which is stored at index 2, is adjacent to both `"body1"` at index 1 and `"tool"` at index 3. Because the skipped collision pairs have not changed, the `SkippedSelfCollisions` name-value argument has no effect on the self-collision checking result for this robot.

```
rbt2j.BodyNames
```

```
ans = 1x3 cell
    {'body1'}    {'body2'}    {'tool'}
```

Run collision checking with both `SkippedSelfCollisions` options to verify that the `SkippedSelfCollisions` name-value argument returns the same result for this robot.

```
checkCollision(rbt2j,homeConfiguration(rbt2j),SkippedSelfCollisions="parent")
```

```
ans = logical
   0
```

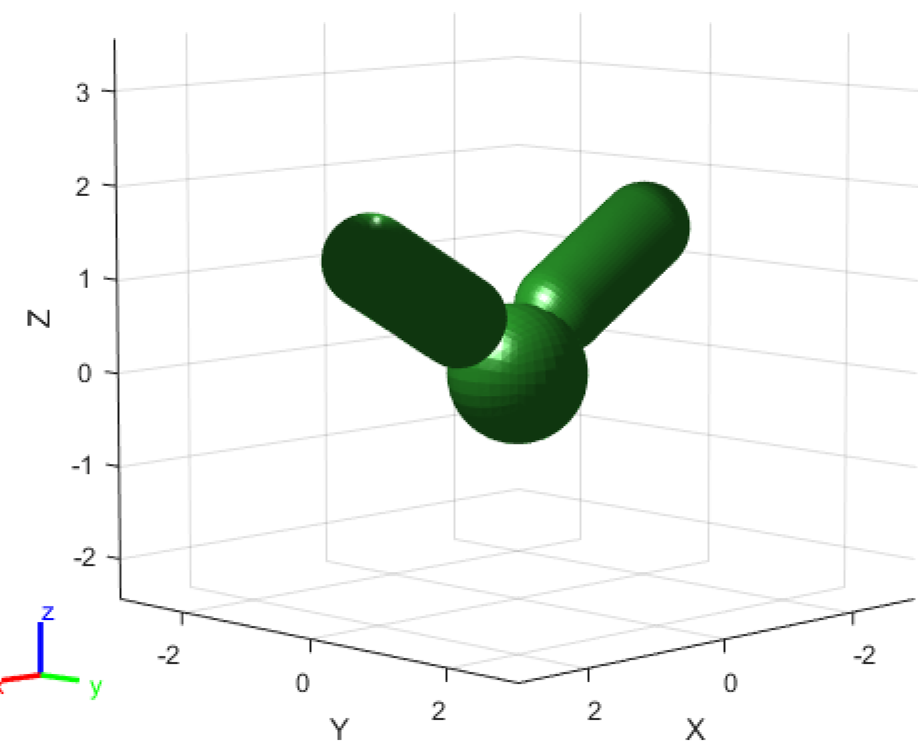
```
checkCollision(rbt2j,homeConfiguration(rbt2j),SkippedSelfCollisions="adjacent")
```

```
ans = logical
   0
```

Parallel Manipulator Robot

Use the exampleHelperCreate2ArmRBT example helper to create a parallel robot comprised of two one-joint arms.

```
rbt2arm = exampleHelperCreate2ArmRBT;
show(rbt2arm,homeConfiguration(rbt2arm),Collisions="on");
axis padded
```



List the body names of the robot. The skipped body pairs formed by bodies of adjacent indices is similar to the serial manipulator but without body, `"tool"`.

```
rbt2arm.BodyNames
```

```
ans = 1x2 cell
    {'body1'}    {'body2'}
```

Check the parent of body1 and the parent of body2. Each body forms a parent-child relationship with the base, even though they are at adjacent indices.

```
rbt2arm.Bodies{1}.Parent.Name
```

```
ans =
'base'
```

```
rbt2arm.Bodies{2}.Parent.Name
```

```
ans =
'base'
```

Run collision checking with both `SkippedSelfCollisions` options.

```
checkCollision(rbt2arm,homeConfiguration(rbt2arm),SkippedSelfCollisions="parent")
```

```
ans = logical
   0
```

```
checkCollision(rbt2arm,homeConfiguration(rbt2arm),SkippedSelfCollisions="adjacent")
```

```
ans = logical
   1
```

As expected, when skipping parent-child body pairs during self collision checks, `checkCollision` finds no self collisions, but does find a self collision between the `"base"` and `"body2"` when skipping body pairs of adjacent indices.

Input Arguments

`robot` — Rigid body tree robot model

`rigidBodyTree` object

Rigid body tree robot model, specified as a `rigidBodyTree` object. To use the `checkCollision` function, the DataFormat property of the `rigidBodyTree` object must be either `'row'` or `'column'`.

`config` — Joint configuration of rigid body tree

n-element numeric vector

Joint configuration of the rigid body tree, specified as an *n*-element numeric vector, where *n* is the number of nonfixed joints in the robot model. Each element of the vector is a specific joint position for a joint in the robot model.

Data Types: `single` | `double`

`worldObjects` — List of collision objects in world
`{}` (default) | cell array of collision objects

List of collision objects in the world, specified as a cell array of collision objects with any combination of `collisionBox`, `collisionCylinder`, `collisionSphere`, and `collisionMesh` objects. The function assumes that the `Pose` property of each object is relative to the base of the rigid body tree robot model.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `'Exhaustive','on'` enables exhaustive checking for collisions and causes the function to calculate all separation distances and witness points.

`Exhaustive` — Check for all collisions
`'off'` (default) | `'on'`

Exhaustively check for all collisions, specified as the comma-separated pair consisting of `'Exhaustive'` and `'on'` or `'off'`. By default, the function finds the first collision and stops, returning the separation distances and witness points for incomplete checks as `Inf`.

If this name-value pair argument is specified as `'on'`, the function instead continues checking for collisions until it has exhausted all possibilities.

Data Types: `char` | `string`

`IgnoreSelfCollision` — Skip checking for robot self-collisions
`'off'` (default) | `'on'`

Skip checking for robot self-collisions, specified as the comma-separated pair consisting of `'IgnoreSelfCollision'` and `'on'` or `'off'`. When this argument is enabled, the function ignores collisions between the collision objects of the rigid body tree robot model bodies and other collision objects of the same model or its base.

This name-value pair argument affects the size of the `separationDist` and `witnessPts` output arguments.

Data Types: `char` | `string`

`SkippedSelfCollisions` — Body pairs skipped for checking self-collisions
`"parent"` (default) | `"adjacent"`

Body pairs skipped for checking self-collisions, specified as either `"parent"` or `"adjacent"`:

- `"parent"` — Skip collision checking between child and parent bodies.
- `"adjacent"` — Skip collision checking between bodies on adjacent indices.

See “Change Self-Collision Checking Behavior” on page 3-365 for more information.

Data Types: `char` | `string`

Output Arguments

Self Collisions

`isSelfColliding` — Robot configuration is in self-collision
0 | 1

Robot configuration is in self-collision returned as a logical `0` (`false`) or `1` (`true`). If the function returns a value of `true` for this argument, that means that one of the rigid body collision objects is touching another collision object in the robot model. Add collision objects to your rigid body tree robot model using the `addCollision` function.

Data Types: `logical`

`selfSeparationDist` — Minimum separation distance between bodies of robot
$(m+1)$ -by-$(m+1)$ matrix

Minimum separation distance between the bodies of the robot, returned as an $(m+1)$ -by-$(m+1)$ matrix, where m is the number of bodies. The final row and column correspond to the robot base. Units are in meters.

If a pair is in collision, the function returns the separation distance for the associated element as `NaN`.

Data Types: `double`

`selfWitnessPts` — Witness points between robot bodies
$3(m+1)$ -by-$2(m+1)$ matrix

Witness points between the robot bodies including the base, returned as an $3(m+1)$-by-$2(m+1)$ matrix, where m is the number of bodies. Witness points are the points on any two bodies that are closest to one another for a given configuration. The matrix takes the form:

The matrix is divided into 3-by-2 sections that represent the *xyz*-coordinates of witness point pairs in the form:

$$\begin{bmatrix} x_1 & x_2 \\ y_1 & y_2 \\ z_1 & y_2 \end{bmatrix}$$

Each section corresponds to a separation distance in the `selfSeparationDist` output matrix. Use these equations to determine where the section of the `selfWitnessPts` matrix that corresponds to a specific separation distance begins:

$$W_r = 3S_r - 2$$

$$W_c = 2S_c - 1$$

Where (S_r,S_c) is the index of a separation distance in the separation distance matrix and (W_r,W_c) is the index in the witness point matrix at which the corresponding witness points begin.

If a pair is in collision, the function returns each coordinate of the witness points for that element as `NaN`.

Data Types: `double`

World Collisions

`isColliding` — Robot configuration is in collision

two-element logical vector

Robot configuration is in collision, returned as a two-element logical vector. The first element indicates whether the robot is in self-collision. The second element indicates whether the robot model is in collision with any world objects.

Data Types: `logical`

`separationDist` — Minimum separation distance between collision objects

(m+1)-by-(m+w+1) matrix

Minimum separation distance between the collision objected, returned as an (m+1)-by-(m+w+1) matrix, where m is the number of bodies and w is the number of world objects. The first m rows correspond to the robot bodies, where the $(m+1)^{th}$ row or column index corresponds to the base. The remaining w columns correspond to the world objects.

The matrix is divided into 3-by-2 sections that represent the xyz-coordinates of witness point pairs in the form:

$$\begin{bmatrix} x_1 & x_2 \\ y_1 & y_2 \\ z_1 & y_2 \end{bmatrix}$$

Each section corresponds to a separation distance in the `separationDist` output matrix. Use these equations to determine where the section of the `witnessPts` matrix that corresponds to a specific separation distance begins:

$$W_r = 3S_r - 2$$

$$W_c = 2S_c - 1$$

Where (S_r,S_c) is the index of a separation distance in the separation distance matrix and (W_r,W_c) is the index in the witness point matrix at which the corresponding witness points begin.

If a pair is in collision, the function returns each coordinate of the witness points for that element as `NaN`.

If a pair is in collision, the function returns the separation distance as `NaN`.

Dependencies

If you specify the `'IgnoreSelfCollision'` name-value pair argument as `'on'`, then the matrix does not contain values for the distances between any given body and other bodies in the robot model.

Data Types: `double`

`witnessPts` — Witness points between collision objects

3(m+1)-by-2(m+w+1) matrix

Witness points between collision objects, specified as a 3(m+1)-by-2(m+w+1) matrix, where `m` is the number of bodies and `w` is the number of world objects. Witness points are the points on any two bodies that are closest to one another for a given configuration. The matrix takes the form:

```
[Wr1_1       Wr1_2     ...    Wr1_(N+1)     Wo1_1     Wo1_2      ... W1_M;
 Wr2_1       Wr2_2     ...    Wr2_(N+1)     Wo2_1     Wo2_2      ... W2_M;
 .           .         .      .             .         .          .   .
 .           .         .      .             .         .          .   .
 .           .         .      .             .         .          .   .
 Wr(N+1)_1   Wr(N+1)_2 ...    Wr(N+1)_(N+1) Wo(N+1)_1 Wo(N+1)_2  ... W(N+1)_M]
```

Each element in the above matrix is a 2-by-3 matrix that gives the nearest `[x y z]` points on the two corresponding bodies or world objects. The final row and column correspond to the robot base.

If a pair are in collision, witness points for that element are returned as `NaN(3,2)`.

Dependencies

If the `"IgnoreSelfCollision"` name-value pair is set to `"on"`, then the matrix contains no `Wr` elements.

Data Types: `double`

Version History

Introduced in R2020b

R2022b: Alter rigid body tree self-collision checking behavior change and new default self-collision checking behavior

Behavior change in future release

You can now specify self-collision checking behavior for a rigid body tree robot model by using the `SkippedSelfCollisions` name-value argument. Specify `SkippedSelfCollisions` as `"parent"` or `"adjacent"`:

- `"parent"` — Collision checking ignores self-collisions between parent and child rigid bodies.
- `"adjacent"` — Collision checking ignores self -collisions between rigid bodies of adjacent indices.

As of R2022b, the default behavior of collision checking is to ignore self-collisions between parent and child rigid bodies. In previous releases, the default behavior of self-collision checking was to ignore self-collisions between adjacent rigid bodies. To instead ignore self-collisions between rigid bodies of adjacent indices, specify `SkippedSelfCollisions` as `"adjacent"`.

See “Change Self-Collision Checking Behavior” on page 3-365 for more information about how to use this name-value argument.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

When creating the `rigidBodyTree` object, use the syntax that specifies the `MaxNumBodies` as an upper bound for adding bodies to the robot model. You must also specify the `DataFormat` property as a name-value pair. For example:

```
robot = rigidBodyTree("MaxNumBodies",15,"DataFormat","row")
```

To minimize data usage, limit the upper bound to a number close to the expected number of bodies in the model. All data formats are supported for code generation. To use the dynamics functions, the data format must be set to `"row"` or `"column"`.

The `show` and `showdetails` functions do not support code generation.

See Also

`rigidBodyTree` | `addCollision` | `clearCollision`

copy

Copy robot model

Syntax

```
newrobot = copy(robot)
```

Description

`newrobot = copy(robot)` creates a deep copy of `robot` with the same properties. Any changes in `newrobot` are not reflected in `robot`.

Examples

Modify a Robot Rigid Body Tree Model

Make changes to an existing `rigidBodyTree` object. You can get replace joints, bodies and subtrees in the rigid body tree.

Load example robots as `rigidBodyTree` objects.

```
load exampleRobots.mat
```

View the details of the Puma robot using `showdetails`.

```
showdetails(puma1)
```

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

 Idx    Body Name   Joint Name   Joint Type    Parent Name(Idx)   Children Name(s)
 ---    ---------   ----------   ----------    ----------------   ----------------
   1           L1         jnt1     revolute             base(0)   L2(2)
   2           L2         jnt2     revolute               L1(1)   L3(3)
   3           L3         jnt3     revolute               L2(2)   L4(4)
   4           L4         jnt4     revolute               L3(3)   L5(5)
   5           L5         jnt5     revolute               L4(4)   L6(6)
   6           L6         jnt6     revolute               L5(5)   
--------------------
```

Get a specific body to inspect the properties. The only child of the `L3` body is the `L4` body. You can copy a specific body as well.

```
body3 = getBody(puma1,'L3');
childBody = body3.Children{1}
```

```
childBody = 
  rigidBody with properties:

            Name: 'L4'
           Joint: [1x1 rigidBodyJoint]
```

```
            Mass: 1
    CenterOfMass: [0 0 0]
         Inertia: [1 1 1 0 0 0]
          Parent: [1x1 rigidBody]
        Children: {[1x1 rigidBody]}
         Visuals: {}
      Collisions: {}
```

```
body3Copy = copy(body3);
```

Replace the joint on the `L3` body. You must create a new `Joint` object and use `replaceJoint` to ensure the downstream body geometry is unaffected. Call `setFixedTransform` if necessary to define a transform between the bodies instead of with the default identity matrices.

```
newJoint = rigidBodyJoint('prismatic');
replaceJoint(puma1,'L3',newJoint);

showdetails(puma1)
```

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

 Idx    Body Name       Joint Name       Joint Type    Parent Name(Idx)   Children Name(s)
 ---    ---------       ----------       ----------    ----------------   ----------------
   1           L1             jnt1         revolute             base(0)   L2(2)
   2           L2             jnt2         revolute               L1(1)   L3(3)
   3           L3        prismatic            fixed               L2(2)   L4(4)
   4           L4             jnt4         revolute               L3(3)   L5(5)
   5           L5             jnt5         revolute               L4(4)   L6(6)
   6           L6             jnt6         revolute               L5(5)
--------------------
```

Remove an entire body and get the resulting subtree using `removeBody`. The removed body is included in the subtree.

```
subtree = removeBody(puma1,'L4')
```

```
subtree = 
  rigidBodyTree with properties:

     NumBodies: 3
        Bodies: {[1x1 rigidBody]  [1x1 rigidBody]  [1x1 rigidBody]}
          Base: [1x1 rigidBody]
     BodyNames: {'L4'  'L5'  'L6'}
      BaseName: 'L3'
       Gravity: [0 0 0]
    DataFormat: 'struct'
```

Remove the modified `L3` body. Add the original copied `L3` body to the `L2` body, followed by the returned subtree. The robot model remains the same. See a detailed comparison through `showdetails`.

```
removeBody(puma1,'L3');
addBody(puma1,body3Copy,'L2')
addSubtree(puma1,'L3',subtree)

showdetails(puma1)
```

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

 Idx     Body Name   Joint Name   Joint Type    Parent Name(Idx)   Children Name(s)
 ---     ---------   ----------   ----------    ----------------   ----------------
   1            L1         jnt1     revolute             base(0)   L2(2)
   2            L2         jnt2     revolute               L1(1)   L3(3)
   3            L3         jnt3     revolute               L2(2)   L4(4)
   4            L4         jnt4     revolute               L3(3)   L5(5)
   5            L5         jnt5     revolute               L4(4)   L6(6)
   6            L6         jnt6     revolute               L5(5)
--------------------
```

Input Arguments

robot — Robot model
rigidBodyTree object

Robot model, specified as a rigidBodyTree object.

Output Arguments

newrobot — Robot model
rigidBodyTree object

Robot model, returned as a rigidBodyTree object.

Version History

Introduced in R2016b

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

When creating the rigidBodyTree object, use the syntax that specifies the MaxNumBodies as an upper bound for adding bodies to the robot model. You must also specify the DataFormat property as a name-value pair. For example:

```
robot = rigidBodyTree("MaxNumBodies",15,"DataFormat","row")
```

To minimize data usage, limit the upper bound to a number close to the expected number of bodies in the model. All data formats are supported for code generation. To use the dynamics functions, the data format must be set to "row" or "column".

The show and showdetails functions do not support code generation.

See Also

`rigidBodyJoint` | `rigidBody` | `rigidBodyTree`

externalForce

Compose external force matrix relative to base

Syntax

```
fext = externalForce(robot,bodyname,wrench)
fext = externalForce(robot,bodyname,wrench,configuration)
```

Description

`fext = externalForce(robot,bodyname,wrench)` composes the external force matrix, which you can use as inputs to `inverseDynamics` and `forwardDynamics` to apply an external force, `wrench`, to the body specified by `bodyname`. The `wrench` input is assumed to be in the base frame.

`fext = externalForce(robot,bodyname,wrench,configuration)` composes the external force matrix assuming that `wrench` is in the `bodyname` frame for the specified `configuration`. The force matrix `fext` is given in the base frame.

Examples

Compute Forward Dynamics Due to External Forces on Rigid Body Tree Model

Calculate the resultant joint accelerations for a given robot configuration with applied external forces and forces due to gravity. A wrench is applied to a specific body with the gravity being specified for the whole robot.

Load a predefined KUKA LBR robot model, which is specified as a `RigidBodyTree` object.

```
load exampleRobots.mat lbr
```

Set the data format to `'row'`. For all dynamics calculations, the data format must be either `'row'` or `'column'`.

```
lbr.DataFormat = 'row';
```

Set the gravity. By default, gravity is assumed to be zero.

```
lbr.Gravity = [0 0 -9.81];
```

Get the home configuration for the `lbr` robot.

```
q = homeConfiguration(lbr);
```

Specify the wrench vector that represents the external forces experienced by the robot. Use the `externalForce` function to generate the external force matrix. Specify the robot model, the end effector that experiences the wrench, the wrench vector, and the current robot configuration. `wrench` is given relative to the `'tool0'` body frame, which requires you to specify the robot configuration, `q`.

```
wrench = [0 0 0.5 0 0 0.3];
fext = externalForce(lbr,'tool0',wrench,q);
```

Compute the resultant joint accelerations due to gravity, with the external force applied to the end-effector 'tool0' when lbr is at its home configuration. The joint velocities and joint torques are assumed to be zero (input as an empty vector []).

```
qddot = forwardDynamics(lbr,q,[],[],fext);
```

Compute Joint Torque to Counter External Forces

Use the externalForce function to generate force matrices to apply to a rigid body tree model. The force matrix is an *m*-by-6 vector that has a row for each joint on the robot to apply a six-element wrench. Use the externalForce function and specify the end effector to properly assign the wrench to the correct row of the matrix. You can add multiple force matrices together to apply multiple forces to one robot.

To calculate the joint torques that counter these external forces, use the inverseDynamics function.

Load a predefined KUKA LBR robot model, which is specified as a RigidBodyTree object.

```
load exampleRobots.mat lbr
```

Set the data format to 'row'. For all dynamics calculations, the data format must be either 'row' or 'column'.

```
lbr.DataFormat = 'row';
```

Set the Gravity property to give a specific gravitational acceleration.

```
lbr.Gravity = [0 0 -9.81];
```

Get the home configuration for lbr.

```
q = homeConfiguration(lbr);
```

Set external force on link1. The input wrench vector is expressed in the base frame.

```
fext1 = externalForce(lbr,'link_1',[0 0 0.0 0.1 0 0]);
```

Set external force on the end effector, tool0. The input wrench vector is expressed in the tool0 frame.

```
fext2 = externalForce(lbr,'tool0',[0 0 0.0 0.1 0 0],q);
```

Compute the joint torques required to balance the external forces. To combine the forces, add the force matrices together. Joint velocities and accelerations are assumed to be zero (input as []).

```
tau = inverseDynamics(lbr,q,[],[],fext1+fext2);
```

Input Arguments

robot — Robot model
rigidBodyTree object

Robot model, specified as a rigidBodyTree object. To use the externalForce function, set the DataFormat property to either "row" or "column".

bodyname — Name of body to which external force is applied
string scalar | character vector

Name of body to which the external force is applied, specified as a string scalar or character vector. This body name must match a body on the `robot` object.

Data Types: `char` | `string`

wrench — Torques and forces applied to body
`[Tx Ty Tz Fx Fy Fz]` vector

Torques and forces applied to the body, specified as a `[Tx Ty Tz Fx Fy Fz]` vector. The first three elements of the wrench correspond to the moments around *xyz*-axes. The last three elements are linear forces along the same axes. Unless you specify the robot `configuration`, the wrench is assumed to be relative to the base frame.

configuration — Robot configuration
vector

Robot configuration, specified as a vector with positions for all nonfixed joints in the robot model. You can generate a configuration using `homeConfiguration(robot)`, `randomConfiguration(robot)`, or by specifying your own joint positions. To use the vector form of `configuration`, set the `DataFormat` property for the `robot` to either `"row"` or `"column"` .

Output Arguments

fext — External force matrix
n-by-6 matrix | 6-by-*n* matrix

External force matrix, returned as either an *n*-by-6 or 6-by-*n* matrix, where *n* is the velocity number (degrees of freedom) of the robot. The shape depends on the `DataFormat` property of `robot`. The `"row"` data format uses an *n*-by-6 matrix. The `"column"` data format uses a 6-by-*n* .

The composed matrix lists only values other than zero at the locations relevant to the body specified. You can add force matrices together to specify multiple forces on multiple bodies. Use the external force matrix to specify external forces to dynamics functions `inverseDynamics` and `forwardDynamics`.

More About

Dynamics Properties

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

- `Mass` — Mass of the rigid body in kilograms.
- `CenterOfMass` — Center of mass position of the rigid body, specified as a vector of the form `[x y z]`. The vector describes the location of the center of mass of the rigid body, relative to the body frame, in meters. The `centerOfMass` object function uses these rigid body property values when computing the center of mass of a robot.
- `Inertia` — Inertia of the rigid body, specified as a vector of the form `[Ixx Iyy Izz Iyz Ixz Ixy]`. The vector is relative to the body frame in kilogram square meters. The inertia tensor is a positive definite matrix of the form:

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

The first three elements of the `Inertia` vector are the moment of inertia, which are the diagonal elements of the inertia tensor. The last three elements are the product of inertia, which are the off-diagonal elements of the inertia tensor.

For information related to the entire manipulator robot model, specify these `rigidBodyTree` object properties:

- `Gravity` — Gravitational acceleration experienced by the robot, specified as an `[x y z]` vector in m/s^2. By default, there is no gravitational acceleration.
- `DataFormat` — The input and output data format for the kinematics and dynamics functions, specified as `"struct"`, `"row"`, or `"column"`.

Dynamics Equations

Manipulator rigid body dynamics are governed by this equation:

$$\frac{d}{dt}\begin{bmatrix} q \\ \dot{q} \end{bmatrix} = \begin{bmatrix} \dot{q} \\ M(q)^{-1}\left(-C(q,\dot{q})\dot{q} - G(q) - J(q)^T F_{Ext} + \tau\right) \end{bmatrix}$$

also written as:

$$M(q)\ddot{q} = -C(q,\dot{q})\dot{q} - G(q) - J(q)^T F_{Ext} + \tau$$

where:

- $M(q)$ — is a joint-space mass matrix based on the current robot configuration. Calculate this matrix by using the `massMatrix` object function.
- $C(q, \dot{q})$ — are the Coriolis terms, which are multiplied by $\dot{q}$ to calculate the velocity product. Calculate the velocity product by using by the `velocityProduct` object function.
- $G(q)$ — is the gravity torques and forces required for all joints to maintain their positions in the specified gravity `Gravity`. Calculate the gravity torque by using the `gravityTorque` object function.
- $J(q)$ — is the geometric Jacobian for the specified joint configuration. Calculate the geometric Jacobian by using the `geometricJacobian` object function.
- F_{Ext} — is a matrix of the external forces applied to the rigid body. Generate external forces by using the `externalForce` object function.
- τ — are the joint torques and forces applied directly as a vector to each joint.
- $q, \dot{q}, \ddot{q}$ — are the joint configuration, joint velocities, and joint accelerations, respectively, as individual vectors. For revolute joints, specify values in radians, rad/s, and rad/s^2, respectively. For prismatic joints, specify in meters, m/s, and m/s^2.

To compute the dynamics directly, use the `forwardDynamics` object function. The function calculates the joint accelerations for the specified combinations of the above inputs.

To achieve a certain set of motions, use the `inverseDynamics` object function. The function calculates the joint torques required to achieve the specified configuration, velocities, accelerations, and external forces.

Version History

Introduced in R2017a

References

[1] Featherstone, Roy. *Rigid Body Dynamics Algorithms*. Springer US, 2008. DOI.org (Crossref), doi:10.1007/978-1-4899-7560-7.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

When creating the `rigidBodyTree` object, use the syntax that specifies the `MaxNumBodies` as an upper bound for adding bodies to the robot model. You must also specify the `DataFormat` property as a name-value pair. For example:

```
robot = rigidBodyTree("MaxNumBodies",15,"DataFormat","row")
```

To minimize data usage, limit the upper bound to a number close to the expected number of bodies in the model. All data formats are supported for code generation. To use the dynamics functions, the data format must be set to `"row"` or `"column"`.

The `show` and `showdetails` functions do not support code generation.

See Also

`rigidBodyTree` | `inverseDynamics` | `forwardDynamics`

Topics

"Compute Joint Torques To Balance An Endpoint Force and Moment"

forwardDynamics

Joint accelerations given joint torques and states

Syntax

```
jointAccel = forwardDynamics(robot)
jointAccel = forwardDynamics(robot,configuration)
jointAccel = forwardDynamics(robot,configuration,jointVel)
jointAccel = forwardDynamics(robot,configuration,jointVel,jointTorq)
jointAccel = forwardDynamics(robot,configuration,jointVel,jointTorq,fext)
```

Description

`jointAccel = forwardDynamics(robot)` computes joint accelerations due to gravity at the robot home configuration, with zero joint velocities and no external forces.

`jointAccel = forwardDynamics(robot,configuration)` also specifies the joint positions of the robot configuration.

To specify the home configuration, zero joint velocities, or zero torques, use `[]` for that input argument.

`jointAccel = forwardDynamics(robot,configuration,jointVel)` also specifies the joint velocities of the robot.

`jointAccel = forwardDynamics(robot,configuration,jointVel,jointTorq)` also specifies the joint torques applied to the robot.

`jointAccel = forwardDynamics(robot,configuration,jointVel,jointTorq,fext)` also specifies an external force matrix that contains forces applied to each joint.

Examples

Compute Forward Dynamics Due to External Forces on Rigid Body Tree Model

Calculate the resultant joint accelerations for a given robot configuration with applied external forces and forces due to gravity. A wrench is applied to a specific body with the gravity being specified for the whole robot.

Load a predefined KUKA LBR robot model, which is specified as a `RigidBodyTree` object.

```
load exampleRobots.mat lbr
```

Set the data format to `'row'`. For all dynamics calculations, the data format must be either `'row'` or `'column'`.

```
lbr.DataFormat = 'row';
```

Set the gravity. By default, gravity is assumed to be zero.

```
lbr.Gravity = [0 0 -9.81];
```

Get the home configuration for the `lbr` robot.

```
q = homeConfiguration(lbr);
```

Specify the wrench vector that represents the external forces experienced by the robot. Use the `externalForce` function to generate the external force matrix. Specify the robot model, the end effector that experiences the wrench, the wrench vector, and the current robot configuration. `wrench` is given relative to the `'tool0'` body frame, which requires you to specify the robot configuration, `q`.

```
wrench = [0 0 0.5 0 0 0.3];
fext = externalForce(lbr,'tool0',wrench,q);
```

Compute the resultant joint accelerations due to gravity, with the external force applied to the end-effector `'tool0'` when `lbr` is at its home configuration. The joint velocities and joint torques are assumed to be zero (input as an empty vector `[]`).

```
qddot = forwardDynamics(lbr,q,[],[],fext);
```

Input Arguments

`robot` — Robot model
`RigidBodyTree` object

Robot model, specified as a `rigidBodyTree` object. To use the `forwardDynamics` function, set the `DataFormat` property to either `'row'` or `'column'`.

`configuration` — Robot configuration
vector

Robot configuration, specified as a vector with positions for all nonfixed joints in the robot model. You can generate a configuration using `homeConfiguration(robot)`, `randomConfiguration(robot)`, or by specifying your own joint positions. To use the vector form of `configuration`, set the `DataFormat` property for the `robot` to either `'row'` or `'column'`.

`jointVel` — Joint velocities
vector

Joint velocities, specified as a vector. The number of joint velocities is equal to the degrees of freedom of the robot. To use the vector form of `jointVel`, set the `DataFormat` property for the `robot` to either `'row'` or `'column'`.

`jointTorq` — Joint torques
vector

Joint torques, specified as a vector. Each element corresponds to a torque applied to a specific joint. To use the vector form of `jointTorq`, set the `DataFormat` property for the `robot` to either `'row'` or `'column'`.

`fext` — External force matrix
n-by-6 matrix | 6-by-*n* matrix

External force matrix, specified as either an *n*-by-6 or 6-by-*n* matrix, where *n* is the number of bodies of the robot. The shape depends on the `DataFormat` property of `robot`. The `'row'` data format uses an *n*-by-6 matrix. The `'column'` data format uses a 6-by-*n* .

The matrix lists only values other than zero at the locations relevant to the body specified. You can add force matrices together to specify multiple forces on multiple bodies.

To create the matrix for a specified force or torque, see `externalForce`.

Output Arguments

`jointAccel` — Joint accelerations

vector

Joint accelerations, returned as a vector. The dimension of the joint accelerations vector is equal to the degrees of freedom of the robot. Each element corresponds to a specific joint on the `robot`.

More About

Dynamics Properties

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

- `Mass` — Mass of the rigid body in kilograms.
- `CenterOfMass` — Center of mass position of the rigid body, specified as a vector of the form `[x y z]`. The vector describes the location of the center of mass of the rigid body, relative to the body frame, in meters. The `centerOfMass` object function uses these rigid body property values when computing the center of mass of a robot.
- `Inertia` — Inertia of the rigid body, specified as a vector of the form `[Ixx Iyy Izz Iyz Ixz Ixy]`. The vector is relative to the body frame in kilogram square meters. The inertia tensor is a positive definite matrix of the form:

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

  The first three elements of the `Inertia` vector are the moment of inertia, which are the diagonal elements of the inertia tensor. The last three elements are the product of inertia, which are the off-diagonal elements of the inertia tensor.

For information related to the entire manipulator robot model, specify these `rigidBodyTree` object properties:

- `Gravity` — Gravitational acceleration experienced by the robot, specified as an `[x y z]` vector in m/s^2. By default, there is no gravitational acceleration.
- `DataFormat` — The input and output data format for the kinematics and dynamics functions, specified as `"struct"`, `"row"`, or `"column"`.

Dynamics Equations

Manipulator rigid body dynamics are governed by this equation:

$$\frac{d}{dt}\begin{bmatrix} q \\ \dot{q} \end{bmatrix} = \begin{bmatrix} \dot{q} \\ M(q)^{-1}\left(-C(q,\dot{q})\dot{q} - G(q) - J(q)^T F_{Ext} + \tau\right) \end{bmatrix}$$

also written as:

$$M(q)\ddot{q} = -C(q,\dot{q})\dot{q} - G(q) - J(q)^T F_{Ext} + \tau$$

where:

- $M(q)$ — is a joint-space mass matrix based on the current robot configuration. Calculate this matrix by using the `massMatrix` object function.
- $C(q,\dot{q})$ — are the Coriolis terms, which are multiplied by $\dot{q}$ to calculate the velocity product. Calculate the velocity product by using by the `velocityProduct` object function.
- $G(q)$ — is the gravity torques and forces required for all joints to maintain their positions in the specified gravity `Gravity`. Calculate the gravity torque by using the `gravityTorque` object function.
- $J(q)$ — is the geometric Jacobian for the specified joint configuration. Calculate the geometric Jacobian by using the `geometricJacobian` object function.
- F_{Ext} — is a matrix of the external forces applied to the rigid body. Generate external forces by using the `externalForce` object function.
- τ — are the joint torques and forces applied directly as a vector to each joint.
- $q, \dot{q}, \ddot{q}$ — are the joint configuration, joint velocities, and joint accelerations, respectively, as individual vectors. For revolute joints, specify values in radians, rad/s, and rad/s^2, respectively. For prismatic joints, specify in meters, m/s, and m/s^2.

To compute the dynamics directly, use the `forwardDynamics` object function. The function calculates the joint accelerations for the specified combinations of the above inputs.

To achieve a certain set of motions, use the `inverseDynamics` object function. The function calculates the joint torques required to achieve the specified configuration, velocities, accelerations, and external forces.

Version History

Introduced in R2017a

References

[1] Featherstone, Roy. *Rigid Body Dynamics Algorithms*. Springer US, 2008. DOI.org (Crossref), doi:10.1007/978-1-4899-7560-7.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

When creating the `rigidBodyTree` object, use the syntax that specifies the `MaxNumBodies` as an upper bound for adding bodies to the robot model. You must also specify the `DataFormat` property as a name-value pair. For example:

```
robot = rigidBodyTree("MaxNumBodies",15,"DataFormat","row")
```

To minimize data usage, limit the upper bound to a number close to the expected number of bodies in the model. All data formats are supported for code generation. To use the dynamics functions, the data format must be set to `"row"` or `"column"`.

The `show` and `showdetails` functions do not support code generation.

See Also

`rigidBodyTree` | `inverseDynamics` | `externalForce`

Topics

"Compute Joint Torques To Balance An Endpoint Force and Moment"

geometricJacobian

Geometric Jacobian for robot configuration

Syntax

```
jacobian = geometricJacobian(robot,configuration,endeffectorname)
```

Description

`jacobian = geometricJacobian(robot,configuration,endeffectorname)` computes the geometric Jacobian relative to the base for the specified end-effector name and configuration for the robot model.

Examples

Geometric Jacobian for Robot Configuration

Calculate the geometric Jacobian for a specific end effector and configuration of a robot.

Load a Puma robot, which is specified as a `RigidBodyTree` object.

```
load exampleRobots.mat puma1
```

Calculate the geometric Jacobian of body `'L6'` on the Puma robot for a random configuration.

```
geoJacob = geometricJacobian(puma1,randomConfiguration(puma1),'L6')
```

```
geoJacob = 6×6

         0   -0.7795   -0.7795   -0.4592    0.5643   -0.6612
    0.0000    0.6264    0.6264   -0.5714   -0.7789   -0.2282
    1.0000    0.0000    0.0000    0.6801   -0.2734   -0.7146
    0.4544    0.3107    0.1746   -0.0000         0         0
   -0.5577    0.3866    0.2173   -0.0000         0         0
         0    0.7036    0.3304    0.0000         0         0
```

Input Arguments

`robot` — Robot model
`rigidBodyTree` object

Robot model, specified as a `rigidBodyTree` object.

`configuration` — Robot configuration
vector | structure

Robot configuration, specified as a vector of joint positions or a structure with joint names and positions for all the bodies in the robot model. You can generate a configuration using

`homeConfiguration(robot)`, `randomConfiguration(robot)`, or by specifying your own joint positions in a structure. To use the vector form of `configuration`, set the `DataFormat` property for the `robot` to either `"row"` or `"column"` .

`endeffectorname` — End-effector name

string scalar | character vector

End-effector name, specified as a string scalar or character vector. An end effector can be any body in the robot model.

Data Types: `char` | `string`

Output Arguments

`jacobian` — Geometric Jacobian

6-by-*n* matrix

Geometric Jacobian of the end effector with the specified `configuration`, returned as a 6-by-*n* matrix, where *n* is the number of degrees of freedom of the robot. The Jacobian maps the joint-space velocity to the end-effector velocity, relative to the base coordinate frame. The end-effector velocity equals:

$$V_{EE} = \begin{bmatrix} \omega_x \\ \omega_y \\ \omega_z \\ v_x \\ v_y \\ v_z \end{bmatrix} = J\dot{q} = J\begin{bmatrix} \dot{q}_1 \\ \vdots \\ \dot{q}_n \end{bmatrix}$$

ω is the angular velocity, υ is the linear velocity, and $\dot{q}$ is the joint-space velocity.

More About

Dynamics Properties

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

- `Mass` — Mass of the rigid body in kilograms.
- `CenterOfMass` — Center of mass position of the rigid body, specified as a vector of the form `[x y z]`. The vector describes the location of the center of mass of the rigid body, relative to the body frame, in meters. The `centerOfMass` object function uses these rigid body property values when computing the center of mass of a robot.
- `Inertia` — Inertia of the rigid body, specified as a vector of the form `[Ixx Iyy Izz Iyz Ixz Ixy]`. The vector is relative to the body frame in kilogram square meters. The inertia tensor is a positive definite matrix of the form:

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

The first three elements of the `Inertia` vector are the moment of inertia, which are the diagonal elements of the inertia tensor. The last three elements are the product of inertia, which are the off-diagonal elements of the inertia tensor.

For information related to the entire manipulator robot model, specify these `rigidBodyTree` object properties:

- `Gravity` — Gravitational acceleration experienced by the robot, specified as an `[x y z]` vector in m/s^2. By default, there is no gravitational acceleration.
- `DataFormat` — The input and output data format for the kinematics and dynamics functions, specified as `"struct"`, `"row"`, or `"column"`.

Dynamics Equations

Manipulator rigid body dynamics are governed by this equation:

$$\frac{d}{dt}\begin{bmatrix} q \\ \dot{q} \end{bmatrix} = \begin{bmatrix} \dot{q} \\ M(q)^{-1}\left(-C(q,\dot{q})\dot{q} - G(q) - J(q)^T F_{Ext} + \tau\right) \end{bmatrix}$$

also written as:

$$M(q)\ddot{q} = -C(q,\dot{q})\dot{q} - G(q) - J(q)^T F_{Ext} + \tau$$

where:

- $M(q)$ — is a joint-space mass matrix based on the current robot configuration. Calculate this matrix by using the `massMatrix` object function.
- $C(q, \dot{q})$ — are the Coriolis terms, which are multiplied by $\dot{q}$ to calculate the velocity product. Calculate the velocity product by using by the `velocityProduct` object function.
- $G(q)$ — is the gravity torques and forces required for all joints to maintain their positions in the specified gravity `Gravity`. Calculate the gravity torque by using the `gravityTorque` object function.
- $J(q)$ — is the geometric Jacobian for the specified joint configuration. Calculate the geometric Jacobian by using the `geometricJacobian` object function.
- F_{Ext} — is a matrix of the external forces applied to the rigid body. Generate external forces by using the `externalForce` object function.
- τ — are the joint torques and forces applied directly as a vector to each joint.
- $q, \dot{q}, \ddot{q}$ — are the joint configuration, joint velocities, and joint accelerations, respectively, as individual vectors. For revolute joints, specify values in radians, rad/s, and rad/s^2, respectively. For prismatic joints, specify in meters, m/s, and m/s^2.

To compute the dynamics directly, use the `forwardDynamics` object function. The function calculates the joint accelerations for the specified combinations of the above inputs.

To achieve a certain set of motions, use the `inverseDynamics` object function. The function calculates the joint torques required to achieve the specified configuration, velocities, accelerations, and external forces.

Version History

Introduced in R2016b

References

[1] Featherstone, Roy. *Rigid Body Dynamics Algorithms*. Springer US, 2008. DOI.org (Crossref), doi:10.1007/978-1-4899-7560-7.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

When creating the `rigidBodyTree` object, use the syntax that specifies the `MaxNumBodies` as an upper bound for adding bodies to the robot model. You must also specify the `DataFormat` property as a name-value pair. For example:

```
robot = rigidBodyTree("MaxNumBodies",15,"DataFormat","row")
```

To minimize data usage, limit the upper bound to a number close to the expected number of bodies in the model. All data formats are supported for code generation. To use the dynamics functions, the data format must be set to `"row"` or `"column"`.

The `show` and `showdetails` functions do not support code generation.

See Also

`getTransform` | `homeConfiguration` | `randomConfiguration` | `rigidBodyJoint` | `rigidBody`

Topics

"Compute Joint Torques To Balance An Endpoint Force and Moment"

gravityTorque

Joint torques that compensate gravity

Syntax

```
gravTorq = gravityTorque(robot)
gravTorq = gravityTorque(robot,configuration)
```

Description

`gravTorq = gravityTorque(robot)` computes the joint torques required to hold the robot at its home configuration.

`gravTorq = gravityTorque(robot,configuration)` specifies a joint configuration for calculating the gravity torque.

Examples

Compute Gravity Torque for Robot Configuration

Load a predefined KUKA LBR robot model, which is specified as a `RigidBodyTree` object.

```
load exampleRobots.mat lbr
```

Set the data format to `'row'`. For all dynamics calculations, the data format must be either `'row'` or `'column'`. Set the `Gravity` property.

```
lbr.DataFormat = 'row';
lbr.Gravity = [0 0 -9.81];
```

Get a random configuration for `lbr`.

```
q = randomConfiguration(lbr);
```

Compute the gravity-compensating torques for each joint.

```
gtau = gravityTorque(lbr,q);
```

Input Arguments

`robot` — Robot model
`rigidBodyTree` object

Robot model, specified as a `rigidBodyTree` object. To use the `gravityTorque` function, set the `DataFormat` property to either `'row'` or `'column'`.

`configuration` — Robot configuration
vector

Robot configuration, specified as a vector with positions for all nonfixed joints in the robot model. You can generate a configuration using `homeConfiguration(robot)`, `randomConfiguration(robot)`, or by specifying your own joint positions. To use the vector form of `configuration`, set the `DataFormat` property for the `robot` to either `'row'` or `'column'` .

Output Arguments

`gravTorq` — Gravity-compensating torque for each joint
vector

Gravity-compensating torque for each joint, returned as a vector.

More About

Dynamics Properties

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

- `Mass` — Mass of the rigid body in kilograms.
- `CenterOfMass` — Center of mass position of the rigid body, specified as a vector of the form `[x y z]`. The vector describes the location of the center of mass of the rigid body, relative to the body frame, in meters. The `centerOfMass` object function uses these rigid body property values when computing the center of mass of a robot.
- `Inertia` — Inertia of the rigid body, specified as a vector of the form `[Ixx Iyy Izz Iyz Ixz Ixy]`. The vector is relative to the body frame in kilogram square meters. The inertia tensor is a positive definite matrix of the form:

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

  The first three elements of the `Inertia` vector are the moment of inertia, which are the diagonal elements of the inertia tensor. The last three elements are the product of inertia, which are the off-diagonal elements of the inertia tensor.

For information related to the entire manipulator robot model, specify these `rigidBodyTree` object properties:

- `Gravity` — Gravitational acceleration experienced by the robot, specified as an `[x y z]` vector in m/s^2. By default, there is no gravitational acceleration.
- `DataFormat` — The input and output data format for the kinematics and dynamics functions, specified as `"struct"`, `"row"`, or `"column"`.

Dynamics Equations

Manipulator rigid body dynamics are governed by this equation:

$$\frac{d}{dt}\begin{bmatrix} q \\ \dot{q} \end{bmatrix} = \begin{bmatrix} \dot{q} \\ M(q)^{-1}\left(-C(q,\dot{q})\dot{q} - G(q) - J(q)^T F_{Ext} + \tau\right) \end{bmatrix}$$

also written as:

$$M(q)\ddot{q} = -C(q,\dot{q})\dot{q} - G(q) - J(q)^T F_{Ext} + \tau$$

where:

- $M(q)$ — is a joint-space mass matrix based on the current robot configuration. Calculate this matrix by using the `massMatrix` object function.
- $C(q,\dot{q})$ — are the Coriolis terms, which are multiplied by $\dot{q}$ to calculate the velocity product. Calculate the velocity product by using by the `velocityProduct` object function.
- $G(q)$ — is the gravity torques and forces required for all joints to maintain their positions in the specified gravity `Gravity`. Calculate the gravity torque by using the `gravityTorque` object function.
- $J(q)$ — is the geometric Jacobian for the specified joint configuration. Calculate the geometric Jacobian by using the `geometricJacobian` object function.
- F_{Ext} — is a matrix of the external forces applied to the rigid body. Generate external forces by using the `externalForce` object function.
- τ — are the joint torques and forces applied directly as a vector to each joint.
- $q, \dot{q}, \ddot{q}$ — are the joint configuration, joint velocities, and joint accelerations, respectively, as individual vectors. For revolute joints, specify values in radians, rad/s, and rad/s^2, respectively. For prismatic joints, specify in meters, m/s, and m/s^2.

To compute the dynamics directly, use the `forwardDynamics` object function. The function calculates the joint accelerations for the specified combinations of the above inputs.

To achieve a certain set of motions, use the `inverseDynamics` object function. The function calculates the joint torques required to achieve the specified configuration, velocities, accelerations, and external forces.

Version History

Introduced in R2017a

References

[1] Featherstone, Roy. *Rigid Body Dynamics Algorithms*. Springer US, 2008. DOI.org (Crossref), doi:10.1007/978-1-4899-7560-7.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

When creating the `rigidBodyTree` object, use the syntax that specifies the `MaxNumBodies` as an upper bound for adding bodies to the robot model. You must also specify the `DataFormat` property as a name-value pair. For example:

```
robot = rigidBodyTree("MaxNumBodies",15,"DataFormat","row")
```

To minimize data usage, limit the upper bound to a number close to the expected number of bodies in the model. All data formats are supported for code generation. To use the dynamics functions, the data format must be set to `"row"` or `"column"`.

The `show` and `showdetails` functions do not support code generation.

See Also

`rigidBodyTree` | `inverseDynamics` | `velocityProduct`

getBody

Get robot body handle by name

Syntax

```
body = getBody(robot,bodyname)
```

Description

`body = getBody(robot,bodyname)` gets a body handle by name from the robot model.

Examples

Modify a Robot Rigid Body Tree Model

Make changes to an existing `rigidBodyTree` object. You can get replace joints, bodies and subtrees in the rigid body tree.

Load example robots as `rigidBodyTree` objects.

```
load exampleRobots.mat
```

View the details of the Puma robot using `showdetails`.

```
showdetails(puma1)
```

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

 Idx     Body Name   Joint Name   Joint Type    Parent Name(Idx)   Children Name(s)
 ---     ---------   ----------   ----------    ----------------   ----------------
   1            L1         jnt1     revolute             base(0)   L2(2)
   2            L2         jnt2     revolute               L1(1)   L3(3)
   3            L3         jnt3     revolute               L2(2)   L4(4)
   4            L4         jnt4     revolute               L3(3)   L5(5)
   5            L5         jnt5     revolute               L4(4)   L6(6)
   6            L6         jnt6     revolute               L5(5)
--------------------
```

Get a specific body to inspect the properties. The only child of the `L3` body is the `L4` body. You can copy a specific body as well.

```
body3 = getBody(puma1,'L3');
childBody = body3.Children{1}
```

```
childBody = 
  rigidBody with properties:

            Name: 'L4'
           Joint: [1x1 rigidBodyJoint]
            Mass: 1
```

```
    CenterOfMass: [0 0 0]
         Inertia: [1 1 1 0 0 0]
          Parent: [1x1 rigidBody]
        Children: {[1x1 rigidBody]}
         Visuals: {}
      Collisions: {}
```

```
body3Copy = copy(body3);
```

Replace the joint on the `L3` body. You must create a new `Joint` object and use `replaceJoint` to ensure the downstream body geometry is unaffected. Call `setFixedTransform` if necessary to define a transform between the bodies instead of with the default identity matrices.

```
newJoint = rigidBodyJoint('prismatic');
replaceJoint(puma1,'L3',newJoint);

showdetails(puma1)
```

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

 Idx    Body Name       Joint Name       Joint Type    Parent Name(Idx)   Children Name(s)
 ---    ---------       ----------       ----------    ----------------   ----------------
   1           L1             jnt1         revolute             base(0)   L2(2)
   2           L2             jnt2         revolute               L1(1)   L3(3)
   3           L3        prismatic            fixed               L2(2)   L4(4)
   4           L4             jnt4         revolute               L3(3)   L5(5)
   5           L5             jnt5         revolute               L4(4)   L6(6)
   6           L6             jnt6         revolute               L5(5)
--------------------
```

Remove an entire body and get the resulting subtree using `removeBody`. The removed body is included in the subtree.

```
subtree = removeBody(puma1,'L4')
```

```
subtree =
  rigidBodyTree with properties:

     NumBodies: 3
        Bodies: {[1x1 rigidBody]  [1x1 rigidBody]  [1x1 rigidBody]}
          Base: [1x1 rigidBody]
     BodyNames: {'L4'  'L5'  'L6'}
      BaseName: 'L3'
       Gravity: [0 0 0]
    DataFormat: 'struct'
```

Remove the modified `L3` body. Add the original copied `L3` body to the `L2` body, followed by the returned subtree. The robot model remains the same. See a detailed comparison through `showdetails`.

```
removeBody(puma1,'L3');
addBody(puma1,body3Copy,'L2')
addSubtree(puma1,'L3',subtree)

showdetails(puma1)
```

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

 Idx     Body Name   Joint Name   Joint Type    Parent Name(Idx)   Children Name(s)
 ---     ---------   ----------   ----------    ----------------   ----------------
   1            L1         jnt1     revolute             base(0)   L2(2)
   2            L2         jnt2     revolute               L1(1)   L3(3)
   3            L3         jnt3     revolute               L2(2)   L4(4)
   4            L4         jnt4     revolute               L3(3)   L5(5)
   5            L5         jnt5     revolute               L4(4)   L6(6)
   6            L6         jnt6     revolute               L5(5)
--------------------
```

Input Arguments

robot — Robot model
rigidBodyTree object

Robot model, specified as a rigidBodyTree object.

bodyname — Body name
string scalar | character vector

Body name, specified as a string scalar or character vector. A body with this name must be on the robot model specified by robot.

Data Types: char | string

Output Arguments

body — Rigid body
rigidBody object

Rigid body, returned as a rigidBody object. The returned rigidBodyTree object is still a part of the rigidBodyTree robot model. Use replaceBody with a new body to modify the body in the robot model.

Version History

Introduced in R2016b

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

When creating the rigidBodyTree object, use the syntax that specifies the MaxNumBodies as an upper bound for adding bodies to the robot model. You must also specify the DataFormat property as a name-value pair. For example:

```
robot = rigidBodyTree("MaxNumBodies",15,"DataFormat","row")
```

To minimize data usage, limit the upper bound to a number close to the expected number of bodies in the model. All data formats are supported for code generation. To use the dynamics functions, the data format must be set to `"row"` or `"column"`.

The `show` and `showdetails` functions do not support code generation.

See Also

`rigidBodyJoint` | `rigidBody` | `addBody` | `replaceBody`

getTransform

Get transform between body frames

Syntax

```
transform = getTransform(robot,configuration,bodyname)
transform = getTransform(robot,configuration,sourcebody,targetbody)
```

Description

`transform = getTransform(robot,configuration,bodyname)` computes the transform that converts points in the `bodyname` frame to the robot base frame, using the specified robot configuration.

`transform = getTransform(robot,configuration,sourcebody,targetbody)` computes the transform that converts points from the source body frame to the target body frame, using the specified robot configuration.

Examples

Get Transform Between Frames for Robot Configuration

Get the transform between two frames for a specific robot configuration.

Load a sample robots that include the `puma1` robot.

```
load exampleRobots.mat
```

Get the transform between the `'L2'` and `'L6'` bodies of the `puma1` robot given a specific configuration. The transform converts points in 'L2' frame to the 'L6' frame.

```
transform = getTransform(puma1,randomConfiguration(puma1),'L2','L6')
```

```
transform = 4×4

    0.2295   -0.4122    0.8817    0.0485
    0.8621   -0.3344   -0.3807    0.2118
    0.4517    0.8475    0.2786   -0.4027
         0         0         0    1.0000
```

Input Arguments

robot — Robot model
`rigidBodyTree` object

Robot model, specified as a `rigidBodyTree` object.

`configuration` — Robot configuration
structure array

Robot configuration, specified as a structure array with joint names and positions for all the bodies in the robot model. You can generate a configuration using `homeConfiguration(robot)`, `randomConfiguration(robot)`, or by specifying your own joint names and positions in a structure array.

`bodyname` — Body name
string scalar | character vector

Body name, specified as a string scalar or character vector. This body must be on the robot model specified in `robot`.

Data Types: `char` | `string`

`targetbody` — Target body name
string scalar | character vector

Target body name, specified as a character vector. This body must be on the robot model specified in `robot`. The target frame is the coordinate system you want to transform points into.

Data Types: `char` | `string`

`sourcebody` — Body name
string scalar | character vector

Body name, specified as a string scalar or character vector. This body must be on the robot model specified in `robot`. The source frame is the coordinate system you want points transformed from.

Data Types: `char` | `string`

Output Arguments

`transform` — Homogeneous transform
4-by-4 matrix

Homogeneous transform, returned as a 4-by-4 matrix.

Version History

Introduced in R2016b

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

When creating the `rigidBodyTree` object, use the syntax that specifies the `MaxNumBodies` as an upper bound for adding bodies to the robot model. You must also specify the `DataFormat` property as a name-value pair. For example:

```
robot = rigidBodyTree("MaxNumBodies",15,"DataFormat","row")
```

To minimize data usage, limit the upper bound to a number close to the expected number of bodies in the model. All data formats are supported for code generation. To use the dynamics functions, the data format must be set to `"row"` or `"column"`.

The `show` and `showdetails` functions do not support code generation.

See Also

`rigidBodyJoint` | `rigidBody` | `geometricJacobian` | `homeConfiguration` | `randomConfiguration`

homeConfiguration

Get home configuration of robot

Syntax

```
configuration = homeConfiguration(robot)
```

Description

`configuration = homeConfiguration(robot)` returns the home configuration of the robot model. The home configuration is the ordered list of `HomePosition` properties of each nonfixed joint.

Examples

Visualize Robot Configurations

Show different configurations of a robot created using a `RigidBodyTree` model. Use the `homeConfiguration` or `randomConfiguration` functions to generate the structure that defines all the joint positions.

Load example robots as `RigidBodyTree` objects.

```
load exampleRobots.mat
```

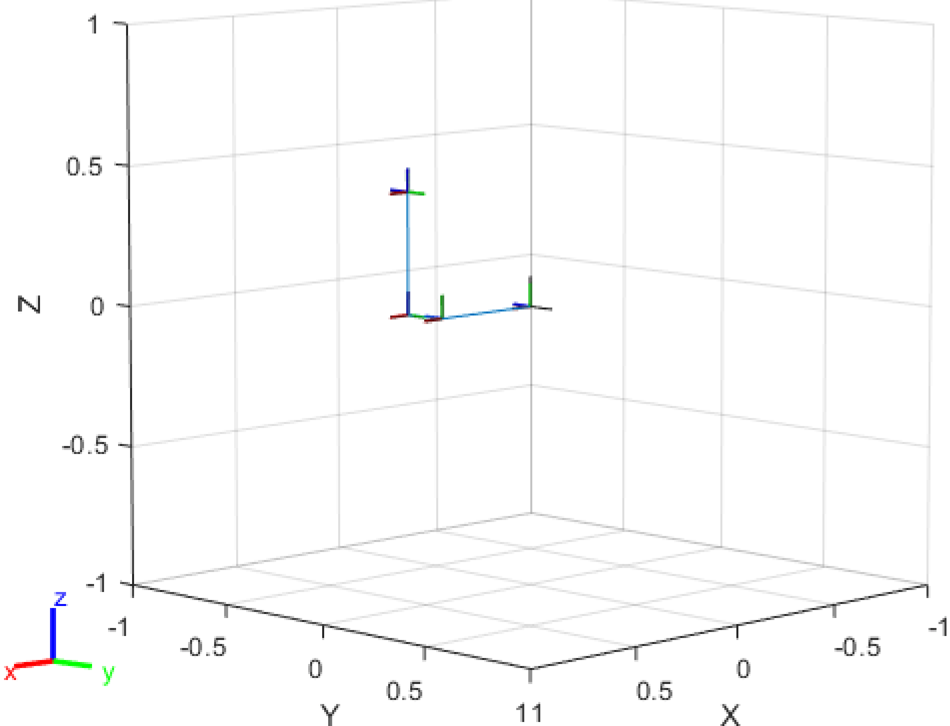
Create a structure for the home configuration of a Puma robot. The structure has joint names and positions for each body on the robot model.

```
config = homeConfiguration(puma1)
```

```
config=1×6 struct array with fields:
    JointName
    JointPosition
```

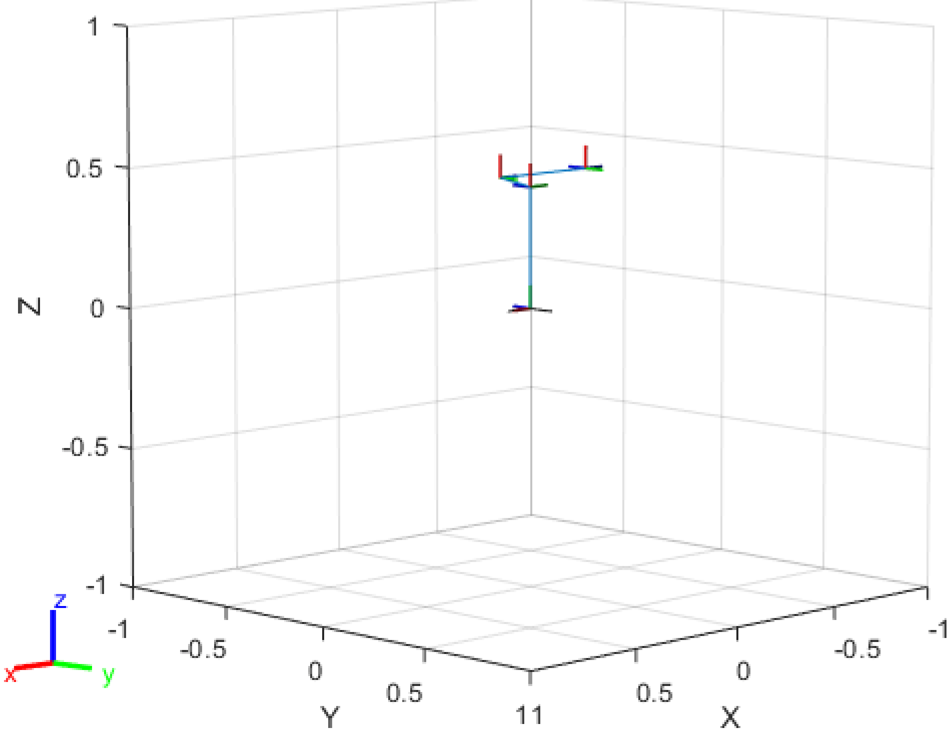
Show the home configuration using `show`. You do not need to specify a configuration input.

```
show(puma1);
```



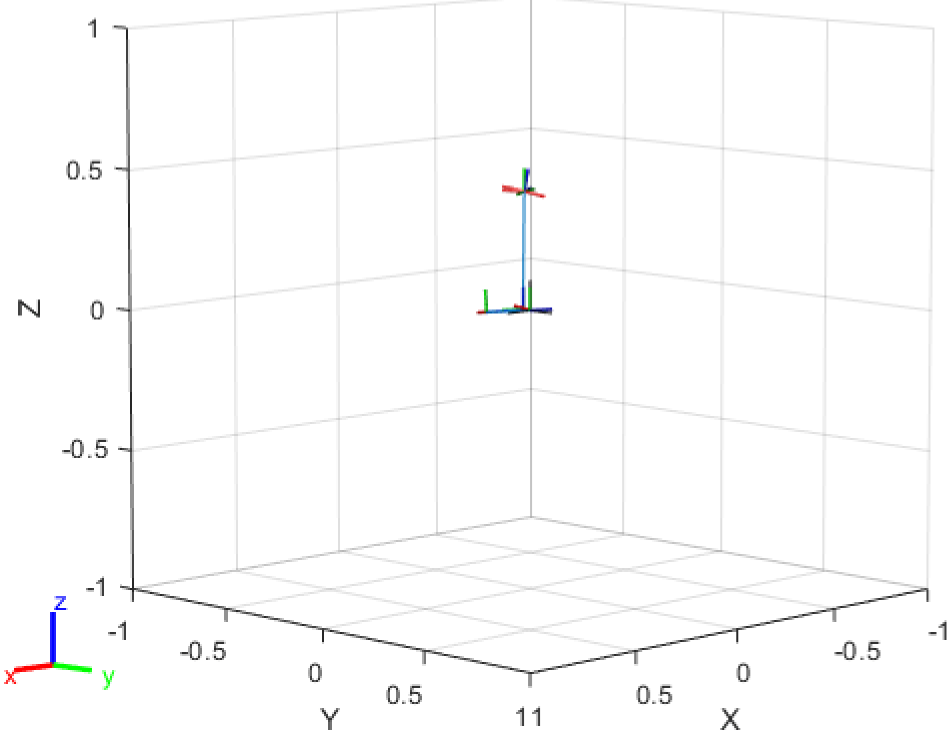
Modify the configuration and set the second joint position to `pi/2`. Show the resulting change in the robot configuration.

```
config(2).JointPosition = pi/2;
show(puma1,config);
```



Create random configurations and show them.

```
show(puma1,randomConfiguration(puma1));
```



Input Arguments

`robot` — Robot model
`rigidBodyTree` object

Robot model, specified as a `rigidBodyTree` object.

Output Arguments

`configuration` — Robot configuration
vector | structure

Robot configuration, returned as a vector of joint positions or a structure with joint names and positions for all the bodies in the robot model. You can generate a configuration using `homeConfiguration(robot)`, `randomConfiguration(robot)`, or by specifying your own joint positions in a structure. To use the vector form of `configuration`, set the `DataFormat` property for the `robot` to either `'row'` or `'column'` .

Version History

Introduced in R2016b

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

When creating the `rigidBodyTree` object, use the syntax that specifies the `MaxNumBodies` as an upper bound for adding bodies to the robot model. You must also specify the `DataFormat` property as a name-value pair. For example:

```
robot = rigidBodyTree("MaxNumBodies",15,"DataFormat","row")
```

To minimize data usage, limit the upper bound to a number close to the expected number of bodies in the model. All data formats are supported for code generation. To use the dynamics functions, the data format must be set to `"row"` or `"column"`.

The `show` and `showdetails` functions do not support code generation.

See Also
`randomConfiguration` | `getTransform` | `geometricJacobian`

inverseDynamics

Required joint torques for given motion

Syntax

```
jointTorq = inverseDynamics(robot)
jointTorq = inverseDynamics(robot,configuration)
jointTorq = inverseDynamics(robot,configuration,jointVel)
jointTorq = inverseDynamics(robot,configuration,jointVel,jointAccel)
jointTorq = inverseDynamics(robot,configuration,jointVel,jointAccel,fext)
```

Description

`jointTorq = inverseDynamics(robot)` computes joint torques required for the robot to statically hold its home configuration with no external forces applied.

`jointTorq = inverseDynamics(robot,configuration)` computes joint torques to hold the specified robot configuration.

`jointTorq = inverseDynamics(robot,configuration,jointVel)` computes joint torques for the specified joint configuration and velocities with zero acceleration and no external forces.

`jointTorq = inverseDynamics(robot,configuration,jointVel,jointAccel)` computes joint torques for the specified joint configuration, velocities, and accelerations with no external forces. To specify the home configuration, zero joint velocities, or zero accelerations, use `[]` for that input argument.

`jointTorq = inverseDynamics(robot,configuration,jointVel,jointAccel,fext)` computes joint torques for the specified joint configuration, velocities, accelerations, and external forces. Use the `externalForce` function to generate `fext`.

Examples

Compute Inverse Dynamics from Static Joint Configuration

Use the `inverseDynamics` function to calculate the required joint torques to statically hold a specific robot configuration. You can also specify the joint velocities, joint accelerations, and external forces using other syntaxes.

Load a predefined KUKA LBR robot model, which is specified as a `RigidBodyTree` object.

```
load exampleRobots.mat lbr
```

Set the data format to `'row'`. For all dynamics calculations, the data format must be either `'row'` or `'column'`.

```
lbr.DataFormat = 'row';
```

Set the `Gravity` property to give a specific gravitational acceleration.

```
lbr.Gravity = [0 0 -9.81];
```

Generate a random configuration for `lbr`.

```
q = randomConfiguration(lbr);
```

Compute the required joint torques for `lbr` to statically hold that configuration.

```
tau = inverseDynamics(lbr,q);
```

Compute Joint Torque to Counter External Forces

Use the `externalForce` function to generate force matrices to apply to a rigid body tree model. The force matrix is an *m*-by-6 vector that has a row for each joint on the robot to apply a six-element wrench. Use the `externalForce` function and specify the end effector to properly assign the wrench to the correct row of the matrix. You can add multiple force matrices together to apply multiple forces to one robot.

To calculate the joint torques that counter these external forces, use the `inverseDynamics` function.

Load a predefined KUKA LBR robot model, which is specified as a `RigidBodyTree` object.

```
load exampleRobots.mat lbr
```

Set the data format to `'row'`. For all dynamics calculations, the data format must be either `'row'` or `'column'`.

```
lbr.DataFormat = 'row';
```

Set the `Gravity` property to give a specific gravitational acceleration.

```
lbr.Gravity = [0 0 -9.81];
```

Get the home configuration for `lbr`.

```
q = homeConfiguration(lbr);
```

Set external force on `link1`. The input wrench vector is expressed in the base frame.

```
fext1 = externalForce(lbr,'link_1',[0 0 0.0 0.1 0 0]);
```

Set external force on the end effector, `tool0`. The input wrench vector is expressed in the `tool0` frame.

```
fext2 = externalForce(lbr,'tool0',[0 0 0.0 0.1 0 0],q);
```

Compute the joint torques required to balance the external forces. To combine the forces, add the force matrices together. Joint velocities and accelerations are assumed to be zero (input as `[]`).

```
tau = inverseDynamics(lbr,q,[],[],fext1+fext2);
```

Input Arguments

robot — Robot model
rigidBodyTree object

Robot model, specified as a rigidBodyTree object. To use the inverseDynamics function, set the DataFormat property to either 'row' or 'column'.

configuration — Robot configuration
vector

Robot configuration, specified as a vector with positions for all nonfixed joints in the robot model. You can generate a configuration using homeConfiguration(robot), randomConfiguration(robot), or by specifying your own joint positions. To use the vector form of configuration, set the DataFormat property for the robot to either 'row' or 'column' .

jointVel — Joint velocities
vector

Joint velocities, specified as a vector. The number of joint velocities is equal to the velocity degrees of freedom of the robot. To use the vector form of jointVel, set the DataFormat property for the robot to either 'row' or 'column' .

jointAccel — Joint accelerations
vector

Joint accelerations, returned as a vector. The dimension of the joint accelerations vector is equal to the velocity degrees of freedom of the robot. Each element corresponds to a specific joint on the robot. To use the vector form of jointAccel, set the DataFormat property for the robot to either 'row' or 'column' .

fext — External force matrix
n-by-6 matrix | 6-by-*n* matrix

External force matrix, specified as either an *n*-by-6 or 6-by-*n* matrix, where *n* is the velocity degrees of freedom of the robot. The shape depends on the DataFormat property of robot. The 'row' data format uses an *n*-by-6 matrix. The 'column' data format uses a 6-by-*n* .

The matrix lists only values other than zero at the locations relevant to the body specified. You can add force matrices together to specify multiple forces on multiple bodies.

To create the matrix for a specified force or torque, see externalForce.

Output Arguments

jointTorq — Joint torques
vector

Joint torques, returned as a vector. Each element corresponds to a torque applied to a specific joint.

More About

Dynamics Properties

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

- `Mass` — Mass of the rigid body in kilograms.
- `CenterOfMass` — Center of mass position of the rigid body, specified as a vector of the form `[x y z]`. The vector describes the location of the center of mass of the rigid body, relative to the body frame, in meters. The `centerOfMass` object function uses these rigid body property values when computing the center of mass of a robot.
- `Inertia` — Inertia of the rigid body, specified as a vector of the form `[Ixx Iyy Izz Iyz Ixz Ixy]`. The vector is relative to the body frame in kilogram square meters. The inertia tensor is a positive definite matrix of the form:

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

  The first three elements of the `Inertia` vector are the moment of inertia, which are the diagonal elements of the inertia tensor. The last three elements are the product of inertia, which are the off-diagonal elements of the inertia tensor.

For information related to the entire manipulator robot model, specify these `rigidBodyTree` object properties:

- `Gravity` — Gravitational acceleration experienced by the robot, specified as an `[x y z]` vector in m/s^2. By default, there is no gravitational acceleration.
- `DataFormat` — The input and output data format for the kinematics and dynamics functions, specified as `"struct"`, `"row"`, or `"column"`.

Dynamics Equations

Manipulator rigid body dynamics are governed by this equation:

$$\frac{d}{dt}\begin{bmatrix} q \\ \dot{q} \end{bmatrix} = \begin{bmatrix} \dot{q} \\ M(q)^{-1}\left(-C(q,\dot{q})\dot{q} - G(q) - J(q)^T F_{Ext} + \tau\right) \end{bmatrix}$$

also written as:

$$M(q)\ddot{q} = -C(q,\dot{q})\dot{q} - G(q) - J(q)^T F_{Ext} + \tau$$

where:

- $M(q)$ — is a joint-space mass matrix based on the current robot configuration. Calculate this matrix by using the `massMatrix` object function.
- $C(q, \dot{q})$ — are the Coriolis terms, which are multiplied by $\dot{q}$ to calculate the velocity product. Calculate the velocity product by using by the `velocityProduct` object function.

- $G(q)$ — is the gravity torques and forces required for all joints to maintain their positions in the specified gravity `Gravity`. Calculate the gravity torque by using the `gravityTorque` object function.
- $J(q)$ — is the geometric Jacobian for the specified joint configuration. Calculate the geometric Jacobian by using the `geometricJacobian` object function.
- F_{Ext} — is a matrix of the external forces applied to the rigid body. Generate external forces by using the `externalForce` object function.
- τ — are the joint torques and forces applied directly as a vector to each joint.
- $q, \dot{q}, \ddot{q}$ — are the joint configuration, joint velocities, and joint accelerations, respectively, as individual vectors. For revolute joints, specify values in radians, rad/s, and rad/s^2, respectively. For prismatic joints, specify in meters, m/s, and m/s^2.

To compute the dynamics directly, use the `forwardDynamics` object function. The function calculates the joint accelerations for the specified combinations of the above inputs.

To achieve a certain set of motions, use the `inverseDynamics` object function. The function calculates the joint torques required to achieve the specified configuration, velocities, accelerations, and external forces.

Version History

Introduced in R2017a

References

[1] Featherstone, Roy. *Rigid Body Dynamics Algorithms*. Springer US, 2008. DOI.org (Crossref), doi:10.1007/978-1-4899-7560-7.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

When creating the `rigidBodyTree` object, use the syntax that specifies the `MaxNumBodies` as an upper bound for adding bodies to the robot model. You must also specify the `DataFormat` property as a name-value pair. For example:

```
robot = rigidBodyTree("MaxNumBodies",15,"DataFormat","row")
```

To minimize data usage, limit the upper bound to a number close to the expected number of bodies in the model. All data formats are supported for code generation. To use the dynamics functions, the data format must be set to `"row"` or `"column"`.

The `show` and `showdetails` functions do not support code generation.

See Also

`rigidBodyTree` | `forwardDynamics` | `externalForce`

Topics

“Compute Joint Torques To Balance An Endpoint Force and Moment”

massMatrix

Joint-space mass matrix

Syntax

```
H = massMatrix(robot)
H = massMatrix(robot,configuration)
```

Description

`H = massMatrix(robot)` returns the joint-space mass matrix of the home configuration of a robot.

`H = massMatrix(robot,configuration)` returns the mass matrix for a specified robot configuration.

Examples

Calculate The Mass Matrix For A Robot Configuration

Load a predefined KUKA LBR robot model, which is specified as a `RigidBodyTree` object.

```
load exampleRobots.mat lbr
```

Set the data format to `'row'`. For all dynamics calculations, the data format must be either `'row'` or `'column'`.

```
lbr.DataFormat = 'row';
```

Generate a random configuration for `lbr`.

```
q = randomConfiguration(lbr);
```

Get the mass matrix at configuration `q`.

```
H = massMatrix(lbr,q);
```

Input Arguments

`robot` — Robot model
`rigidBodyTree` object

Robot model, specified as a `rigidBodyTree` object. To use the `massMatrix` function, set the `DataFormat` property to either `'row'` or `'column'`.

`configuration` — Robot configuration
vector

Robot configuration, specified as a vector with positions for all nonfixed joints in the robot model. You can generate a configuration using `homeConfiguration(robot)`,

`randomConfiguration(robot)`, or by specifying your own joint positions. To use the vector form of `configuration`, set the `DataFormat` property for the `robot` to either `'row'` or `'column'` .

Output Arguments

H — Mass matrix
positive-definite symmetric matrix

Mass matrix of the robot, returned as a positive-definite symmetric matrix with size *n*-by-*n*, where *n* is the velocity degrees of freedom of the robot.

More About

Dynamics Properties

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

- `Mass` — Mass of the rigid body in kilograms.
- `CenterOfMass` — Center of mass position of the rigid body, specified as a vector of the form `[x y z]`. The vector describes the location of the center of mass of the rigid body, relative to the body frame, in meters. The `centerOfMass` object function uses these rigid body property values when computing the center of mass of a robot.
- `Inertia` — Inertia of the rigid body, specified as a vector of the form `[Ixx Iyy Izz Iyz Ixz Ixy]`. The vector is relative to the body frame in kilogram square meters. The inertia tensor is a positive definite matrix of the form:

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

  The first three elements of the `Inertia` vector are the moment of inertia, which are the diagonal elements of the inertia tensor. The last three elements are the product of inertia, which are the off-diagonal elements of the inertia tensor.

For information related to the entire manipulator robot model, specify these `rigidBodyTree` object properties:

- `Gravity` — Gravitational acceleration experienced by the robot, specified as an `[x y z]` vector in m/s^2. By default, there is no gravitational acceleration.
- `DataFormat` — The input and output data format for the kinematics and dynamics functions, specified as `"struct"`, `"row"`, or `"column"`.

Dynamics Equations

Manipulator rigid body dynamics are governed by this equation:

$$\frac{d}{dt}\begin{bmatrix} q \\ \dot{q} \end{bmatrix} = \begin{bmatrix} \dot{q} \\ M(q)^{-1}\left(-C(q,\dot{q})\dot{q} - G(q) - J(q)^T F_{Ext} + \tau\right) \end{bmatrix}$$

also written as:

$$M(q)\ddot{q} = -C(q,\dot{q})\dot{q} - G(q) - J(q)^T F_{Ext} + \tau$$

where:

- $M(q)$ — is a joint-space mass matrix based on the current robot configuration. Calculate this matrix by using the `massMatrix` object function.
- $C(q, \dot{q})$ — are the Coriolis terms, which are multiplied by $\dot{q}$ to calculate the velocity product. Calculate the velocity product by using by the `velocityProduct` object function.
- $G(q)$ — is the gravity torques and forces required for all joints to maintain their positions in the specified gravity `Gravity`. Calculate the gravity torque by using the `gravityTorque` object function.
- $J(q)$ — is the geometric Jacobian for the specified joint configuration. Calculate the geometric Jacobian by using the `geometricJacobian` object function.
- F_{Ext} — is a matrix of the external forces applied to the rigid body. Generate external forces by using the `externalForce` object function.
- τ — are the joint torques and forces applied directly as a vector to each joint.
- $q, \dot{q}, \ddot{q}$ — are the joint configuration, joint velocities, and joint accelerations, respectively, as individual vectors. For revolute joints, specify values in radians, rad/s, and rad/s^2, respectively. For prismatic joints, specify in meters, m/s, and m/s^2.

To compute the dynamics directly, use the `forwardDynamics` object function. The function calculates the joint accelerations for the specified combinations of the above inputs.

To achieve a certain set of motions, use the `inverseDynamics` object function. The function calculates the joint torques required to achieve the specified configuration, velocities, accelerations, and external forces.

Version History

Introduced in R2017a

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

When creating the `rigidBodyTree` object, use the syntax that specifies the `MaxNumBodies` as an upper bound for adding bodies to the robot model. You must also specify the `DataFormat` property as a name-value pair. For example:

```
robot = rigidBodyTree("MaxNumBodies",15,"DataFormat","row")
```

To minimize data usage, limit the upper bound to a number close to the expected number of bodies in the model. All data formats are supported for code generation. To use the dynamics functions, the data format must be set to `"row"` or `"column"`.

The `show` and `showdetails` functions do not support code generation.

See Also

`rigidBodyTree` | `gravityTorque` | `homeConfiguration` | `velocityProduct`

randomConfiguration

Generate random configuration of robot

Syntax

```
configuration = randomConfiguration(robot)
```

Description

`configuration = randomConfiguration(robot)` returns a random configuration of the specified robot. Each joint position in this configuration respects the joint limits set by the `PositionLimits` property of the corresponding `rigidBodyJoint` object in the robot model.

Examples

Visualize Robot Configurations

Show different configurations of a robot created using a `RigidBodyTree` model. Use the `homeConfiguration` or `randomConfiguration` functions to generate the structure that defines all the joint positions.

Load example robots as `RigidBodyTree` objects.

```
load exampleRobots.mat
```

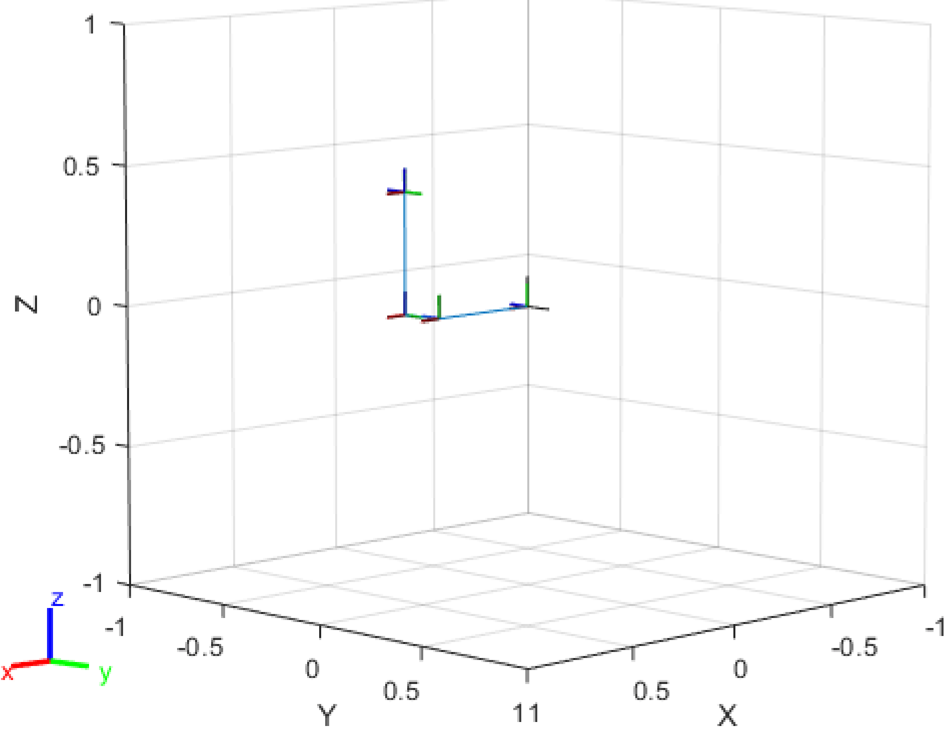
Create a structure for the home configuration of a Puma robot. The structure has joint names and positions for each body on the robot model.

```
config = homeConfiguration(puma1)
```

```
config=1×6 struct array with fields:
    JointName
    JointPosition
```

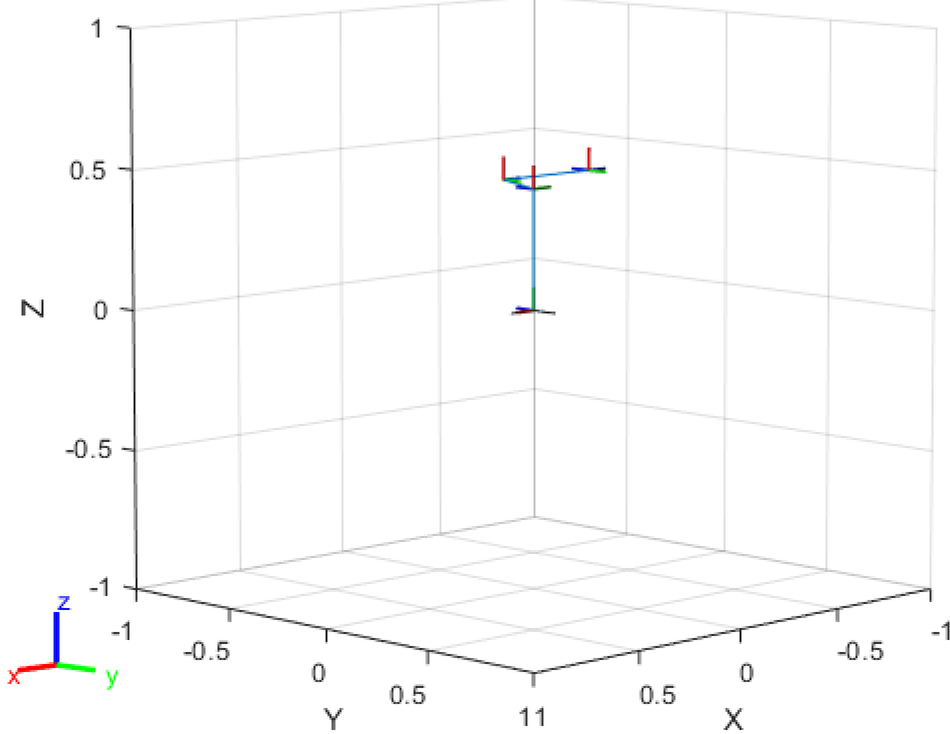
Show the home configuration using `show`. You do not need to specify a configuration input.

```
show(puma1);
```



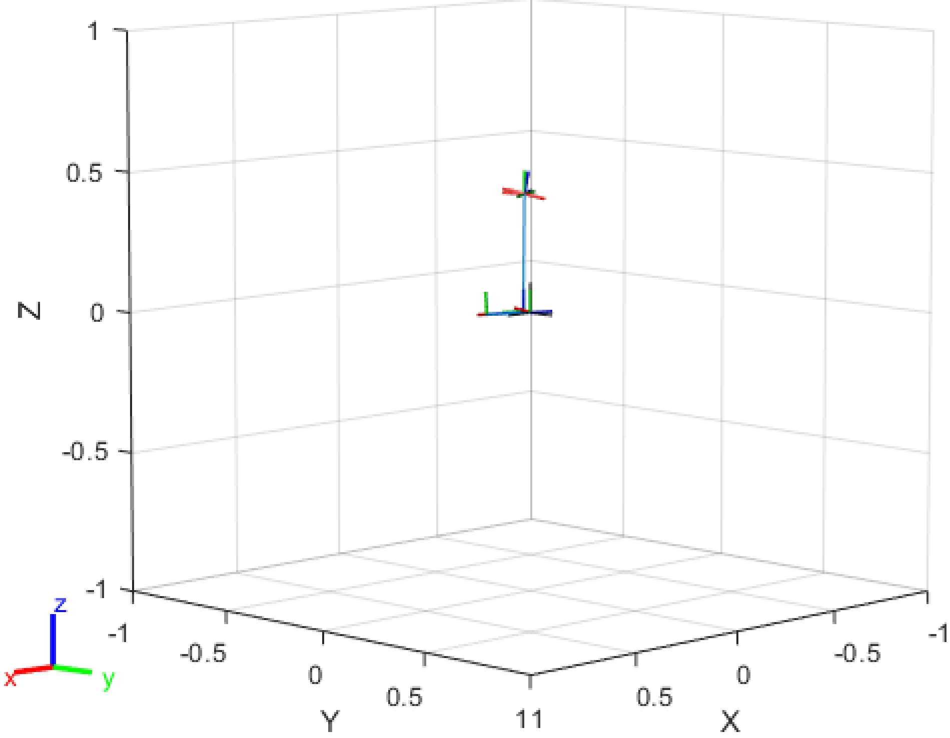
Modify the configuration and set the second joint position to `pi/2`. Show the resulting change in the robot configuration.

```
config(2).JointPosition = pi/2;
show(puma1,config);
```



Create random configurations and show them.

```
show(puma1,randomConfiguration(puma1));
```



Input Arguments

`robot` — Robot model
`rigidBodyTree` object

Robot model, specified as a `rigidBodyTree` object.

Output Arguments

`configuration` — Robot configuration
vector | structure

Robot configuration, returned as a vector of joint positions or a structure with joint names and positions for all the bodies in the robot model. You can generate a configuration using `homeConfiguration(robot)`, `randomConfiguration(robot)`, or by specifying your own joint positions in a structure. To use the vector form of `configuration`, set the `DataFormat` property for the `robot` to either `'row'` or `'column'` .

Version History

Introduced in R2016b

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

When creating the `rigidBodyTree` object, use the syntax that specifies the `MaxNumBodies` as an upper bound for adding bodies to the robot model. You must also specify the `DataFormat` property as a name-value pair. For example:

```
robot = rigidBodyTree("MaxNumBodies",15,"DataFormat","row")
```

To minimize data usage, limit the upper bound to a number close to the expected number of bodies in the model. All data formats are supported for code generation. To use the dynamics functions, the data format must be set to `"row"` or `"column"`.

The `show` and `showdetails` functions do not support code generation.

See Also

`homeConfiguration` | `getTransform` | `geometricJacobian`

removeBody

Remove body from robot

Syntax

```
removeBody(robot,bodyname)
newSubtree = removeBody(robot,bodyname)
```

Description

`removeBody(robot,bodyname)` removes the body and all subsequently attached bodies from the robot model.

`newSubtree = removeBody(robot,bodyname)` returns the subtree created by removing the body and all subsequently attached bodies from the robot model.

Examples

Modify a Robot Rigid Body Tree Model

Make changes to an existing `rigidBodyTree` object. You can get replace joints, bodies and subtrees in the rigid body tree.

Load example robots as `rigidBodyTree` objects.

```
load exampleRobots.mat
```

View the details of the Puma robot using `showdetails`.

```
showdetails(puma1)
```

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

 Idx    Body Name   Joint Name   Joint Type    Parent Name(Idx)   Children Name(s)
 ---    ---------   ----------   ----------    ----------------   ----------------
   1           L1         jnt1     revolute             base(0)   L2(2)
   2           L2         jnt2     revolute               L1(1)   L3(3)
   3           L3         jnt3     revolute               L2(2)   L4(4)
   4           L4         jnt4     revolute               L3(3)   L5(5)
   5           L5         jnt5     revolute               L4(4)   L6(6)
   6           L6         jnt6     revolute               L5(5)
--------------------
```

Get a specific body to inspect the properties. The only child of the `L3` body is the `L4` body. You can copy a specific body as well.

```
body3 = getBody(puma1,'L3');
childBody = body3.Children{1}
```

```
childBody =
  rigidBody with properties:
```

```
            Name: 'L4'
           Joint: [1x1 rigidBodyJoint]
            Mass: 1
    CenterOfMass: [0 0 0]
         Inertia: [1 1 1 0 0 0]
          Parent: [1x1 rigidBody]
        Children: {[1x1 rigidBody]}
         Visuals: {}
      Collisions: {}
```

```
body3Copy = copy(body3);
```

Replace the joint on the `L3` body. You must create a new `Joint` object and use `replaceJoint` to ensure the downstream body geometry is unaffected. Call `setFixedTransform` if necessary to define a transform between the bodies instead of with the default identity matrices.

```
newJoint = rigidBodyJoint('prismatic');
replaceJoint(puma1,'L3',newJoint);

showdetails(puma1)
```

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

 Idx    Body Name       Joint Name       Joint Type    Parent Name(Idx)   Children Name(s)
 ---    ---------       ----------       ----------    ----------------   ----------------
   1           L1             jnt1         revolute             base(0)   L2(2)
   2           L2             jnt2         revolute               L1(1)   L3(3)
   3           L3        prismatic            fixed               L2(2)   L4(4)
   4           L4             jnt4         revolute               L3(3)   L5(5)
   5           L5             jnt5         revolute               L4(4)   L6(6)
   6           L6             jnt6         revolute               L5(5)
--------------------
```

Remove an entire body and get the resulting subtree using `removeBody`. The removed body is included in the subtree.

```
subtree = removeBody(puma1,'L4')
```

```
subtree = 
  rigidBodyTree with properties:

     NumBodies: 3
        Bodies: {[1x1 rigidBody]  [1x1 rigidBody]  [1x1 rigidBody]}
          Base: [1x1 rigidBody]
     BodyNames: {'L4'  'L5'  'L6'}
      BaseName: 'L3'
       Gravity: [0 0 0]
    DataFormat: 'struct'
```

Remove the modified `L3` body. Add the original copied `L3` body to the `L2` body, followed by the returned subtree. The robot model remains the same. See a detailed comparison through `showdetails`.

```
removeBody(puma1,'L3');
addBody(puma1,body3Copy,'L2')
```

```
addSubtree(puma1,'L3',subtree)

showdetails(puma1)

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

 Idx    Body Name   Joint Name   Joint Type    Parent Name(Idx)   Children Name(s)
 ---    ---------   ----------   ----------    ----------------   ----------------
   1           L1         jnt1     revolute             base(0)   L2(2)
   2           L2         jnt2     revolute               L1(1)   L3(3)
   3           L3         jnt3     revolute               L2(2)   L4(4)
   4           L4         jnt4     revolute               L3(3)   L5(5)
   5           L5         jnt5     revolute               L4(4)   L6(6)
   6           L6         jnt6     revolute               L5(5)
--------------------
```

Input Arguments

robot — Robot model
rigidBodyTree object

Robot model, specified as a rigidBodyTree object.

bodyname — Body name
string scalar | character vector

Body name, specified as a string scalar character vector. This body must be on the robot model specified in robot.

Data Types: char | string

Output Arguments

newSubtree — Robot subtree
rigidBodyTree object

Robot subtree, returned as a rigidBodyTree object. This new subtree uses the parent name of the body specified by bodyname as the base name. All bodies that are attached in the previous robot model (including the body with bodyname specified) are added to the subtree.

Version History

Introduced in R2016b

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

When creating the `rigidBodyTree` object, use the syntax that specifies the `MaxNumBodies` as an upper bound for adding bodies to the robot model. You must also specify the `DataFormat` property as a name-value pair. For example:

```
robot = rigidBodyTree("MaxNumBodies",15,"DataFormat","row")
```

To minimize data usage, limit the upper bound to a number close to the expected number of bodies in the model. All data formats are supported for code generation. To use the dynamics functions, the data format must be set to `"row"` or `"column"`.

The `show` and `showdetails` functions do not support code generation.

See Also

`rigidBodyJoint` | `rigidBody` | `addBody` | `replaceBody`

replaceBody

Replace body on robot

Syntax

```
replaceBody(robot,bodyname,newbody)
```

Description

`replaceBody(robot,bodyname,newbody)` replaces the body in the robot model with the new body. All properties of the body are updated accordingly, except the `Parent` and `Children` properties. The rest of the robot model is unaffected.

Examples

Specify Dynamics Properties to Rigid Body Tree

To use dynamics functions to calculate joint torques and accelerations, specify the dynamics properties for the `rigidBodyTree` object and `rigidBody`.

Create a rigid body tree model. Create two rigid bodies to attach to it.

```
robot = rigidBodyTree('DataFormat','row');
body1 = rigidBody('body1');
body2 = rigidBody('body2');
```

Specify joints to attach to the bodies. Set the fixed transformation of `body2` to `body1`. This transform is 1m in the *x*-direction.

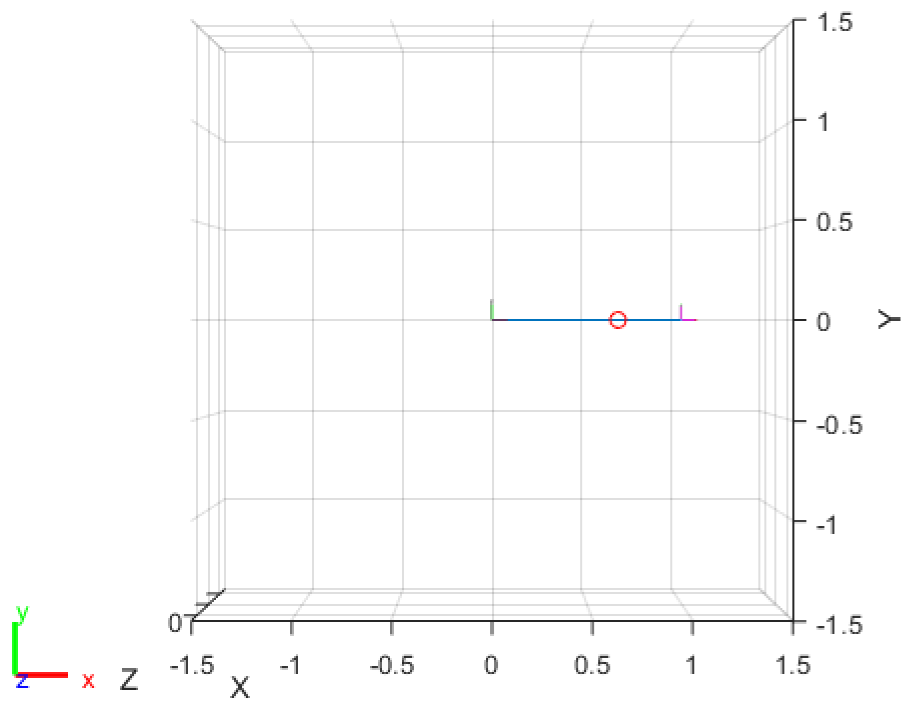
```
joint1 = rigidBodyJoint('joint1','revolute');
joint2 = rigidBodyJoint('joint2');
setFixedTransform(joint2,trvec2tform([1 0 0]))
body1.Joint = joint1;
body2.Joint = joint2;
```

Specify dynamics properties for the two bodies. Add the bodies to the robot model. For this example, basic values for a rod (`body1`) with an attached spherical mass (`body2`) are given.

```
body1.Mass = 2;
body1.CenterOfMass = [0.5 0 0];
body1.Inertia = [0.001 0.67 0.67 0 0 0];

body2.Mass = 1;
body2.CenterOfMass = [0 0 0];
body2.Inertia = 0.0001*[4 4 4 0 0 0];

addBody(robot,body1,'base');
addBody(robot,body2,'body1');
```

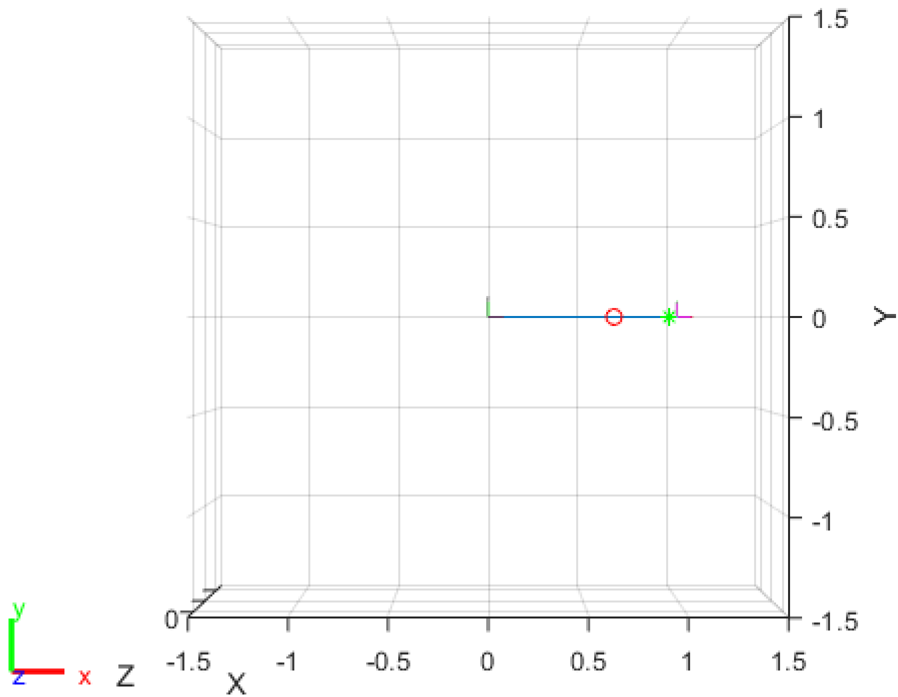
Compute the center of mass position of the whole robot. Plot the position on the robot. Move the view to the *xy* plane.

```
comPos = centerOfMass(robot);

show(robot);
hold on
plot(comPos(1),comPos(2),'or')
view(2)
```



Change the mass of the second body. Notice the change in center of mass.

```
body2.Mass = 20;
replaceBody(robot,'body2',body2)

comPos2 = centerOfMass(robot);
plot(comPos2(1),comPos2(2),'*g')
hold off
```



Input Arguments

robot — Robot model
rigidBodyTree object

Robot model, specified as a `rigidBodyTree` object. The rigid body is added to this object and attached at the rigid body specified by `bodyname`.

bodyname — Body name
string scalar | character vector

Body name, specified as a string scalar or character vector. This body must be on the robot model specified in `robot`.

Data Types: `char` | `string`

newbody — Rigid body
rigidBody object

Rigid body, specified as a `rigidBody` object.

Version History

Introduced in R2016b

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

When creating the `rigidBodyTree` object, use the syntax that specifies the `MaxNumBodies` as an upper bound for adding bodies to the robot model. You must also specify the `DataFormat` property as a name-value pair. For example:

```
robot = rigidBodyTree("MaxNumBodies",15,"DataFormat","row")
```

To minimize data usage, limit the upper bound to a number close to the expected number of bodies in the model. All data formats are supported for code generation. To use the dynamics functions, the data format must be set to `"row"` or `"column"`.

The `show` and `showdetails` functions do not support code generation.

See Also

`rigidBodyJoint` | `rigidBody` | `replaceJoint` | `addBody` | `removeBody`

replaceJoint

Replace joint on body

Syntax

```
replaceJoint(robot,bodyname,joint)
```

Description

`replaceJoint(robot,bodyname,joint)` replaces the joint on the specified body in the robot model if the body is a part of the robot model. This method is the only way to change joints in a robot model. You cannot directly assign the `Joint` property of a rigid body.

Examples

Modify a Robot Rigid Body Tree Model

Make changes to an existing `rigidBodyTree` object. You can get replace joints, bodies and subtrees in the rigid body tree.

Load example robots as `rigidBodyTree` objects.

```
load exampleRobots.mat
```

View the details of the Puma robot using `showdetails`.

```
showdetails(puma1)
```

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

 Idx     Body Name   Joint Name   Joint Type    Parent Name(Idx)   Children Name(s)
 ---     ---------   ----------   ----------    ----------------   ----------------
   1            L1         jnt1     revolute             base(0)   L2(2)
   2            L2         jnt2     revolute               L1(1)   L3(3)
   3            L3         jnt3     revolute               L2(2)   L4(4)
   4            L4         jnt4     revolute               L3(3)   L5(5)
   5            L5         jnt5     revolute               L4(4)   L6(6)
   6            L6         jnt6     revolute               L5(5)
--------------------
```

Get a specific body to inspect the properties. The only child of the `L3` body is the `L4` body. You can copy a specific body as well.

```
body3 = getBody(puma1,'L3');
childBody = body3.Children{1}
```

```
childBody = 
  rigidBody with properties:

            Name: 'L4'
```

```
           Joint: [1x1 rigidBodyJoint]
            Mass: 1
    CenterOfMass: [0 0 0]
         Inertia: [1 1 1 0 0 0]
          Parent: [1x1 rigidBody]
        Children: {[1x1 rigidBody]}
         Visuals: {}
      Collisions: {}
```

```
body3Copy = copy(body3);
```

Replace the joint on the `L3` body. You must create a new `Joint` object and use `replaceJoint` to ensure the downstream body geometry is unaffected. Call `setFixedTransform` if necessary to define a transform between the bodies instead of with the default identity matrices.

```
newJoint = rigidBodyJoint('prismatic');
replaceJoint(puma1,'L3',newJoint);

showdetails(puma1)
```

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

 Idx    Body Name       Joint Name       Joint Type    Parent Name(Idx)   Children Name(s)
 ---    ---------       ----------       ----------    ----------------   ----------------
   1           L1             jnt1         revolute             base(0)   L2(2)
   2           L2             jnt2         revolute               L1(1)   L3(3)
   3           L3        prismatic            fixed               L2(2)   L4(4)
   4           L4             jnt4         revolute               L3(3)   L5(5)
   5           L5             jnt5         revolute               L4(4)   L6(6)
   6           L6             jnt6         revolute               L5(5)
--------------------
```

Remove an entire body and get the resulting subtree using `removeBody`. The removed body is included in the subtree.

```
subtree = removeBody(puma1,'L4')
```

```
subtree =
  rigidBodyTree with properties:

     NumBodies: 3
        Bodies: {[1x1 rigidBody]  [1x1 rigidBody]  [1x1 rigidBody]}
          Base: [1x1 rigidBody]
     BodyNames: {'L4'  'L5'  'L6'}
      BaseName: 'L3'
       Gravity: [0 0 0]
    DataFormat: 'struct'
```

Remove the modified `L3` body. Add the original copied `L3` body to the `L2` body, followed by the returned subtree. The robot model remains the same. See a detailed comparison through `showdetails`.

```
removeBody(puma1,'L3');
addBody(puma1,body3Copy,'L2')
addSubtree(puma1,'L3',subtree)
```

```
showdetails(puma1)

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

 Idx    Body Name   Joint Name   Joint Type    Parent Name(Idx)   Children Name(s)
 ---    ---------   ----------   ----------    ----------------   ----------------
   1           L1         jnt1     revolute             base(0)   L2(2)
   2           L2         jnt2     revolute               L1(1)   L3(3)
   3           L3         jnt3     revolute               L2(2)   L4(4)
   4           L4         jnt4     revolute               L3(3)   L5(5)
   5           L5         jnt5     revolute               L4(4)   L6(6)
   6           L6         jnt6     revolute               L5(5)
--------------------
```

Input Arguments

robot — Robot model
rigidBodyTree object

Robot model, specified as a `rigidBodyTree` object.

bodyname — Body name
string scalar | character vector

Body name, specified as a string scalar or character vector. This body must be on the robot model specified in `robot`.

Data Types: char | string

joint — Replacement joint
rigidBodyJoint object

Replacement joint, specified as a `rigidBodyJoint` object.

Version History

Introduced in R2016b

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

When creating the `rigidBodyTree` object, use the syntax that specifies the `MaxNumBodies` as an upper bound for adding bodies to the robot model. You must also specify the `DataFormat` property as a name-value pair. For example:

```
robot = rigidBodyTree("MaxNumBodies",15,"DataFormat","row")
```

To minimize data usage, limit the upper bound to a number close to the expected number of bodies in the model. All data formats are supported for code generation. To use the dynamics functions, the data format must be set to `"row"` or `"column"`.

The `show` and `showdetails` functions do not support code generation.

See Also

`rigidBodyJoint` | `rigidBody` | `addBody` | `replaceBody`

show

Show robot model in figure

Syntax

```
show(robot)
show(robot,configuration)
show( ___ ,Name,Value)
ax = show( ___ )
```

Description

`show(robot)` plots the body frames of the robot model in a figure with the predefined home configuration. Both `Frames` and `Visuals` are displayed automatically.

`show(robot,configuration)` uses the joint positions specified in `configuration` to show the robot body frames.

`show( ___ ,Name,Value)` specifies options using one or more name-value pair arguments in addition to any combination of input arguments from previous syntaxes. For example, `'Frames','off'` hides the rigid body frames in the figure.

`ax = show( ___ )` returns the axes handle the robot is plotted on.

Examples

Display Robot Model with Visual Geometries

You can import robots that have `.stl` files associated with the Unified Robot Description format (URDF) file to describe the visual geometries of the robot. Each rigid body has an individual visual geometry specified. The `importrobot` function parses the URDF file to get the robot model and visual geometries. The function assumes that visual geometry and collision geometry of the robot are the same and assigns the visual geometries as collision geometries of corresponding bodies.

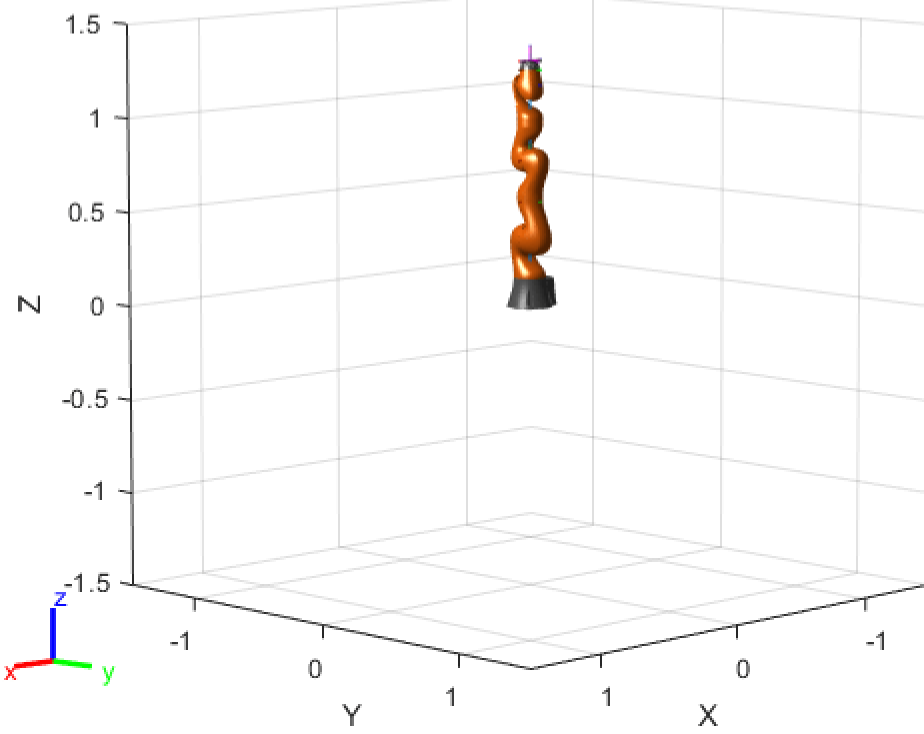
Use the `show` function to display the visual and collision geometries of the robot model in a figure. You can then interact with the model by clicking components to inspect them and right-clicking to toggle visibility.

Import a robot model as a URDF file. The `.stl` file locations must be properly specified in this URDF. To add other `.stl` files to individual rigid bodies, see `addVisual`.

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

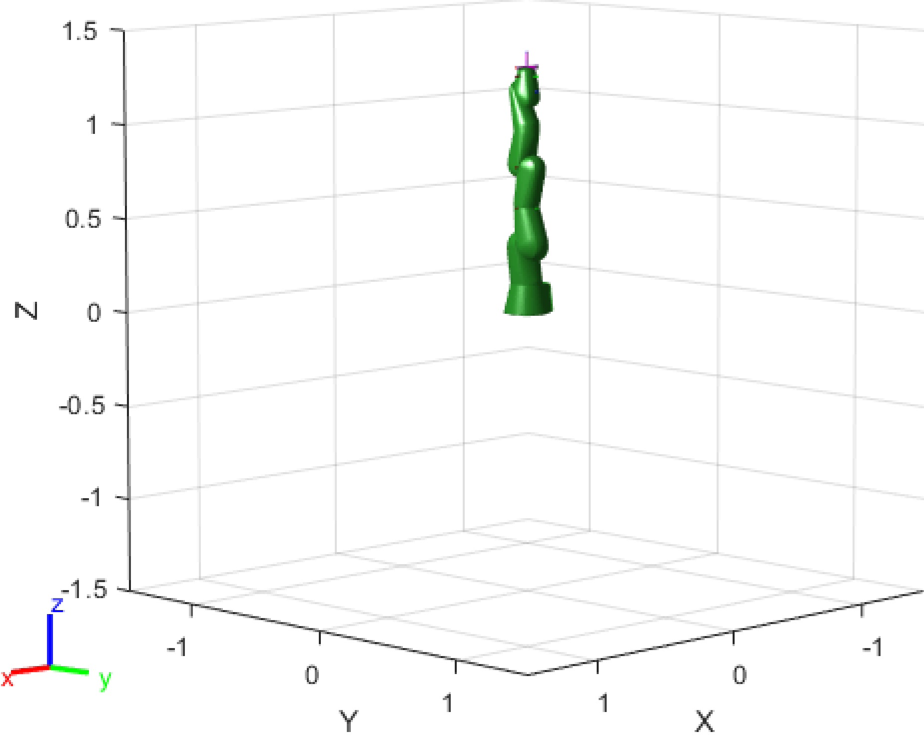
Visualize the robot with the associated visual model. Click bodies or frames to inspect them. Right-click bodies to toggle visibility for each visual geometry.

```
show(robot,'visuals','on','collision','off');
```



Visualize the robot with the associated collision geometries. Click bodies or frames to inspect them. Right-click bodies to toggle visibility for each collision geometry.

```
show(robot,'visuals','off','collision','on');
```



Visualize Robot Configurations

Show different configurations of a robot created using a `RigidBodyTree` model. Use the `homeConfiguration` or `randomConfiguration` functions to generate the structure that defines all the joint positions.

Load example robots as `RigidBodyTree` objects.

```
load exampleRobots.mat
```

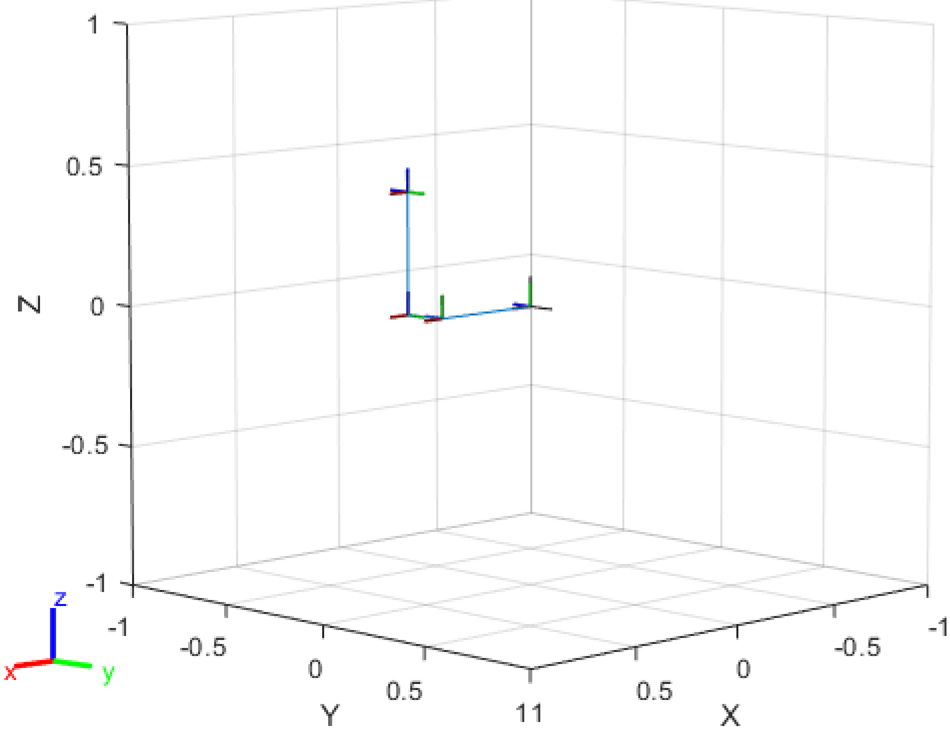
Create a structure for the home configuration of a Puma robot. The structure has joint names and positions for each body on the robot model.

```
config = homeConfiguration(puma1)
```

```
config=1×6 struct array with fields:
    JointName
    JointPosition
```

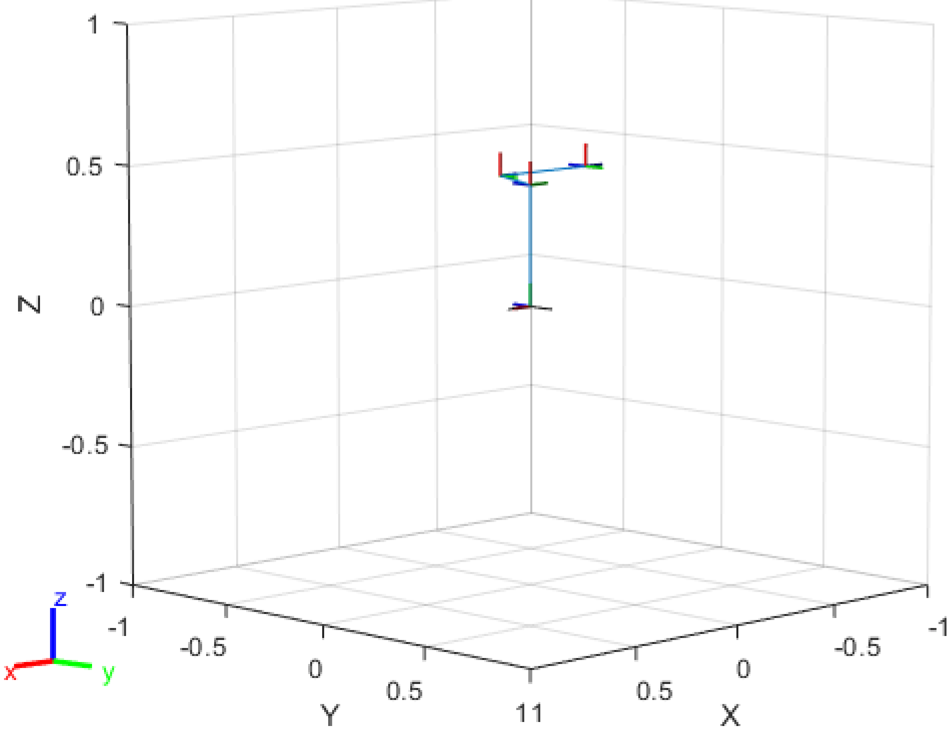
Show the home configuration using `show`. You do not need to specify a configuration input.

```
show(puma1);
```



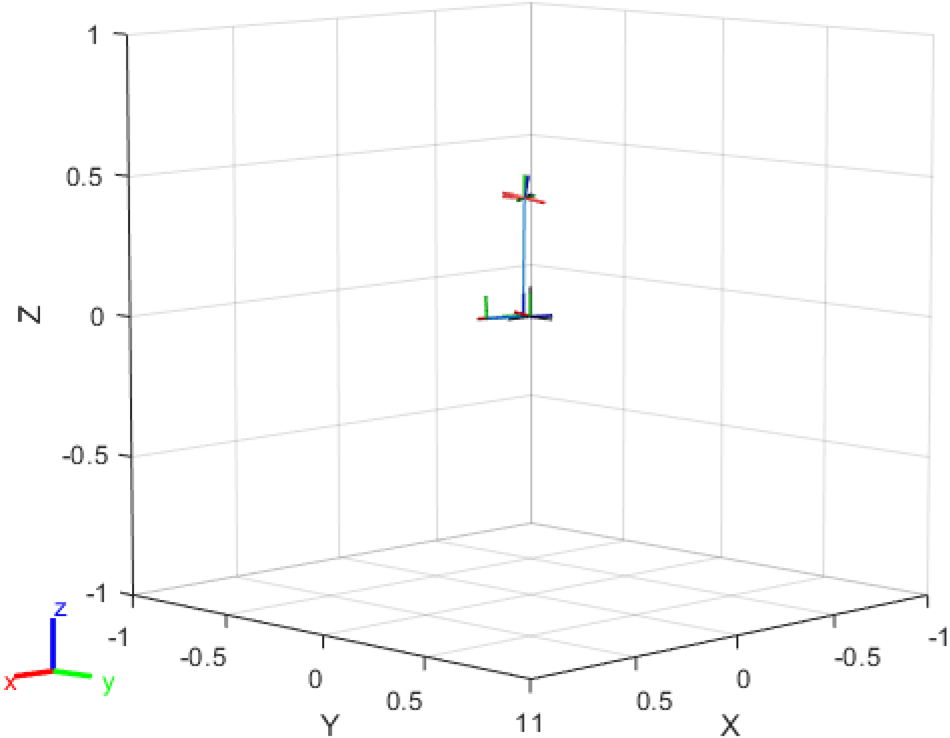
Modify the configuration and set the second joint position to `pi/2`. Show the resulting change in the robot configuration.

```
config(2).JointPosition = pi/2;
show(puma1,config);
```



Create random configurations and show them.

```
show(puma1,randomConfiguration(puma1));
```



Build Manipulator Robot Using Denavit-Hartenberg Parameters

Use the Denavit-Hartenberg (DH) parameters of the Puma560® robot to build a robot. Each rigid body is added one at a time, with the child-to-parent transform specified by the joint object.

The DH parameters define the geometry of the robot with relation to how each rigid body is attached to its parent. For convenience, setup the parameters for the Puma560 robot in a matrix[1] on page 3-442. The Puma robot is a serial chain manipulator. The DH parameters are relative to the previous row in the matrix, corresponding to the previous joint attachment.

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

Create a rigid body tree object to build the robot.

```
robot = rigidBodyTree;
```

Create the first rigid body and add it to the robot. To add a rigid body:

1 Create a `rigidBody` object and give it a unique name.

2 Create a `rigidBodyJoint` object and give it a unique name.
3 Use `setFixedTransform` to specify the body-to-body transformation using DH parameters. The last element of the DH parameters, `theta`, is ignored because the angle is dependent on the joint position.
4 Call `addBody` to attach the first body joint to the base frame of the robot.

```
body1 = rigidBody('body1');
jnt1 = rigidBodyJoint('jnt1','revolute');

setFixedTransform(jnt1,dhparams(1,:),'dh');
body1.Joint = jnt1;

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

Create and add other rigid bodies to the robot. Specify the previous body name when calling `addBody` to attach it. Each fixed transform is relative to the previous joint coordinate frame.

```
body2 = rigidBody('body2');
jnt2 = rigidBodyJoint('jnt2','revolute');
body3 = rigidBody('body3');
jnt3 = rigidBodyJoint('jnt3','revolute');
body4 = rigidBody('body4');
jnt4 = rigidBodyJoint('jnt4','revolute');
body5 = rigidBody('body5');
jnt5 = rigidBodyJoint('jnt5','revolute');
body6 = rigidBody('body6');
jnt6 = rigidBodyJoint('jnt6','revolute');

setFixedTransform(jnt2,dhparams(2,:),'dh');
setFixedTransform(jnt3,dhparams(3,:),'dh');
setFixedTransform(jnt4,dhparams(4,:),'dh');
setFixedTransform(jnt5,dhparams(5,:),'dh');
setFixedTransform(jnt6,dhparams(6,:),'dh');

body2.Joint = jnt2;
body3.Joint = jnt3;
body4.Joint = jnt4;
body5.Joint = jnt5;
body6.Joint = jnt6;

addBody(robot,body2,'body1')
addBody(robot,body3,'body2')
addBody(robot,body4,'body3')
addBody(robot,body5,'body4')
addBody(robot,body6,'body5')
```

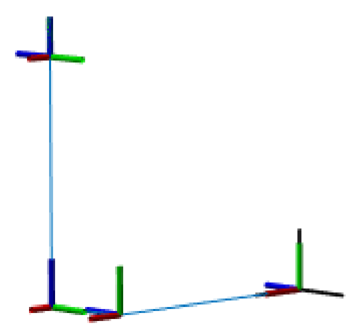
Verify that your robot was built properly by using the `showdetails` or `show` function. `showdetails` lists all the bodies in the MATLAB® command window. `show` displays the robot with a given configuration (home by default). Calls to `axis` modify the axis limits and hide the axis labels.

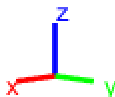
```
showdetails(robot)
```

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

 Idx     Body Name   Joint Name   Joint Type    Parent Name(Idx)   Children Name(s)
 ---     ---------   ----------   ----------    ----------------   ----------------
```

```
    1        body1         jnt1     revolute             base(0)   body2(2)
    2        body2         jnt2     revolute            body1(1)   body3(3)
    3        body3         jnt3     revolute            body2(2)   body4(4)
    4        body4         jnt4     revolute            body3(3)   body5(5)
    5        body5         jnt5     revolute            body4(4)   body6(6)
    6        body6         jnt6     revolute            body5(5)
--------------------
```

```
show(robot);
axis([-0.5,0.5,-0.5,0.5,-0.5,0.5])
axis off
```





References

[1] Corke, P. I., and B. Armstrong-Helouvry. "A Search for Consensus among Model Parameters Reported for the PUMA 560 Robot." *Proceedings of the 1994 IEEE International Conference on Robotics and Automation*, IEEE Computer. Soc. Press, 1994, pp. 1608–13. *DOI.org (Crossref)*, doi:10.1109/ROBOT.1994.351360.

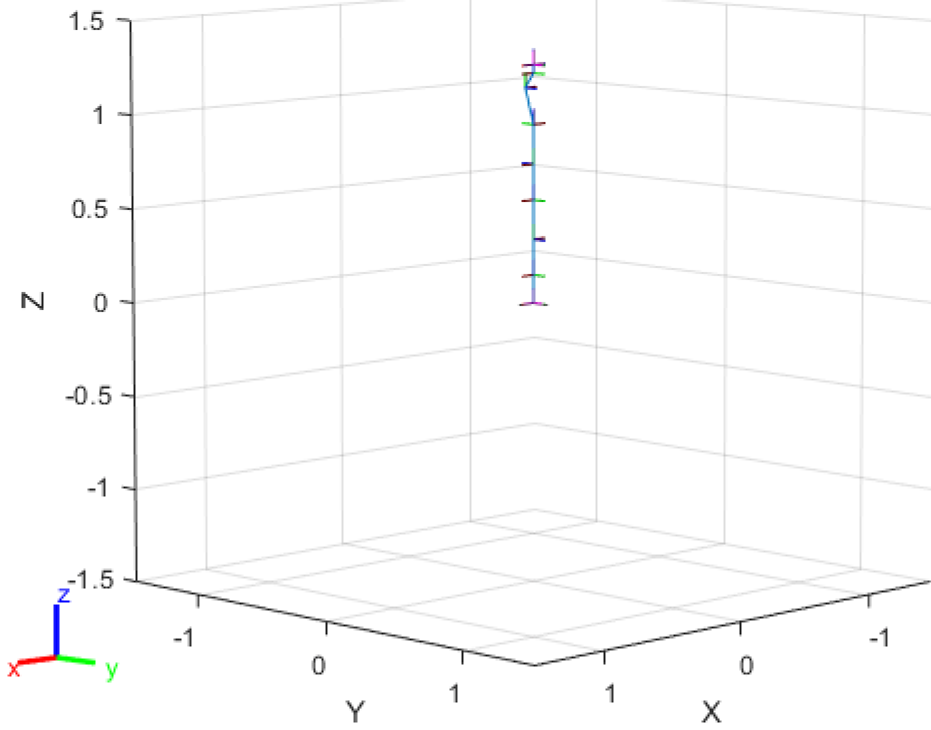
Add Collision Meshes and Check Collisions for Manipulator Robot Arm

Load a robot model and modify the collision meshes. Clear existing collision meshes, add simple collision object primitives, and check whether certain configurations are in collision.

Load Robot Model

Load a preconfigured robot model into the workspace using the `loadrobot` function. This model already has collision meshes specified for each body. Iterate through all the rigid body elements and clear the existing collision meshes. Confirm that the existing meshes are gone.

```
robot = loadrobot('kukaIiwa7','DataFormat','column');

for i = 1:robot.NumBodies
    clearCollision(robot.Bodies{i})
end

show(robot,'Collisions','on','Visuals','off');
```

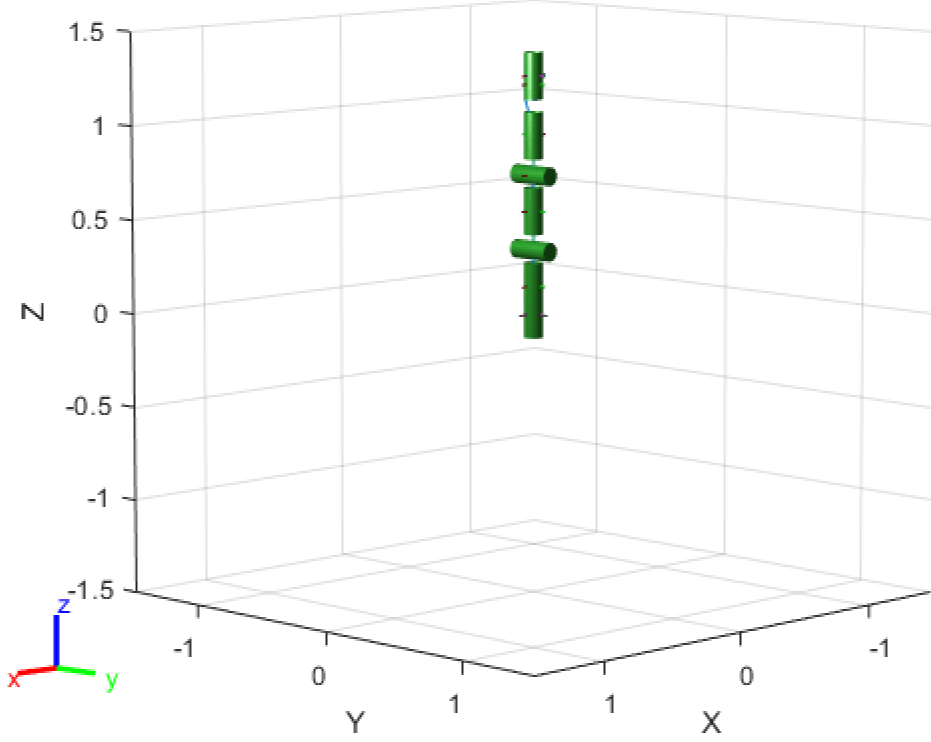


Add Collision Cylinders

Iteratively add a collision cylinder to each body. Skip some bodies for this specific model, as they overlap and always collide with the end effector (body 10).

```
collisionObj = collisionCylinder(0.05,0.25);

for i = 1:robot.NumBodies
    if i > 6 && i < 10
        % Skip these bodies.
    else
        addCollision(robot.Bodies{i},collisionObj)
    end
```

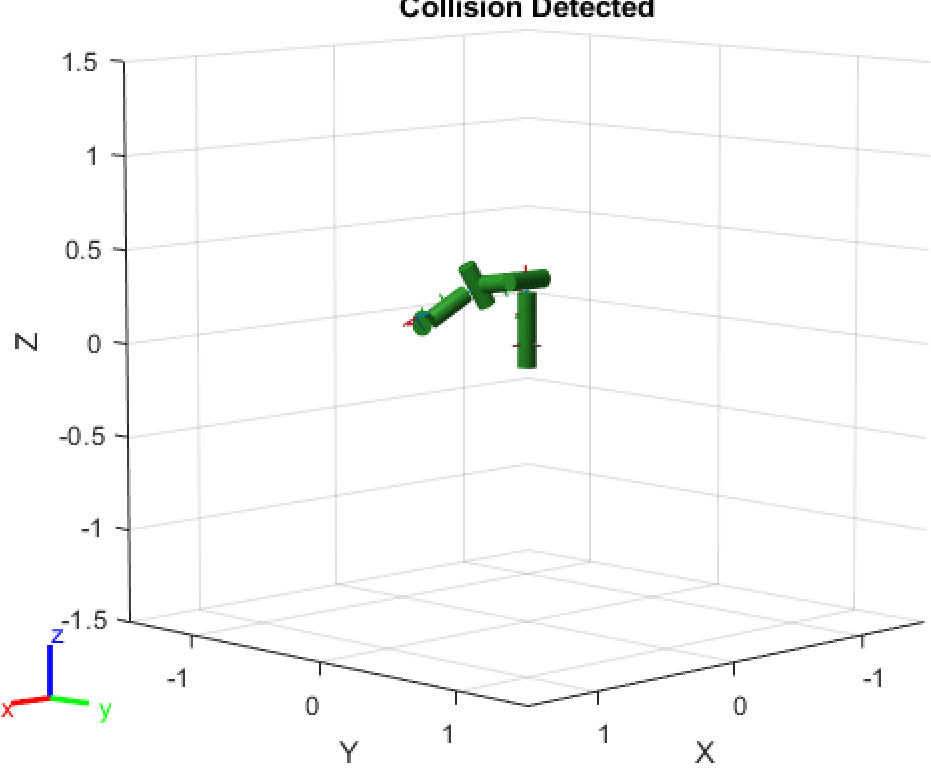
```
end

show(robot,'Collisions','on','Visuals','off');
```



Check for Collisions

Generate a series of random configurations. Check whether the robot is in collision at each configuration. Visualize each configuration that has a collision.

```
figure
rng(0) % Set random seed for repeatability.
for i = 1:20
    config = randomConfiguration(robot);
    isColliding = checkCollision(robot,config,'SkippedSelfCollisions','parent');
    if isColliding
        show(robot,config,'Collisions','on','Visuals','off');
        title('Collision Detected')
    else
        % Skip non-collisions.
    end
end
```



Input Arguments

robot — Robot model
rigidBodyTree object

Robot model, specified as a `rigidBodyTree` object.

configuration — Robot configuration
vector | structure

Robot configuration, specified as a vector of joint positions or a structure with joint names and positions for all the bodies in the robot model. You can generate a configuration using `homeConfiguration(robot)`, `randomConfiguration(robot)`, or by specifying your own joint positions in a structure. To use the vector form of `configuration`, set the `DataFormat` property for the `robot` to either `"row"` or `"column"` .

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `'Frames','off'` hides the rigid body frames in the figure.

`Parent` — Parent of axes
`Axes` object

Parent of axes, specified as the comma-separated pair consisting of `'Parent'` and an `Axes` object in which to draw the robot. By default, the robot is plotted in the active axes.

`PreservePlot` — Option to preserve robot plot
`true` or `1` (default) | `false` or `0`

Option to preserve robot plot, specified as the comma-separated pair consisting of `'PreservePlot'` and a logical `1 (true)` or `0 (false)`. When you specify this argument as `true`, the function does not overwrite previous plots displayed by calling `show`. This setting functions similarly to `hold on` for a standard MATLAB figure, but is limited to robot body frames. When you specify this argument as `false`, the function overwrites previous plots of the robot.

Note When the `'PreservePlot'` value is `true`, the `'FastUpdate'` value must be `false`.

Data Types: `logical`

`Frames` — Display body frames
`'on'` (default) | `'off'`

Display body frames, specified as the comma-separated pair consisting of `'Frames'` and `'on'` or `'off'`. These frames are the coordinate frames of individual bodies on the rigid body tree.

Data Types: `char` | `string`

`Visuals` — Display visual geometries
`'on'` (default) | `'off'`

Display visual geometries, specified as the comma-separated pair consisting of `'Visuals'` and `'on'` or `'off'`. Individual visual geometries can also be turned off by right-clicking them in the figure.

Specify individual visual geometries using `addVisual`. To import a URDF robot model with `.stl` files for meshes, see the `importrobot` function.

Data Types: `char` | `string`

`Collisions` — Display collision geometries
`'off'` (default) | `'on'`

Display collision geometries, specified as the comma-separated pair consisting of `'Collisions'` and `'on'` or `'off'`.

Add collision geometries to the individual rigid bodies in the robot model using the `addCollision` function. To import a URDF robot model with `.stl` files for meshes, see the `importrobot` function.

Data Types: `char` | `string`

`Position` — Position of robot
`[0 0 0 0]` (default) | four-element vector

Position of the robot, specified as the comma-separated pair consisting of `'Position'` and a four-element vector of the form [*x y z yaw*]. The *x*, *y*, and *z* elements specify the position in meters, and *yaw* specifies the yaw angle in radians.

Data Types: `single` | `double`

`FastUpdate` — Fast updates to existing plot
`false` or `0` (default) | `true` or `1`

Fast updates to existing plot, specified as the comma-separated pair consisting of `'FastUpdate'` and a logical `0 (false)` or `1 (true)`. You must use the `show` object function to initially display the robot model before you can specify it with this argument.

Note When the `'FastUpdate'` value is `true`, the `'PreservePlot'` value must be `false`.

Data Types: `logical`

Output Arguments

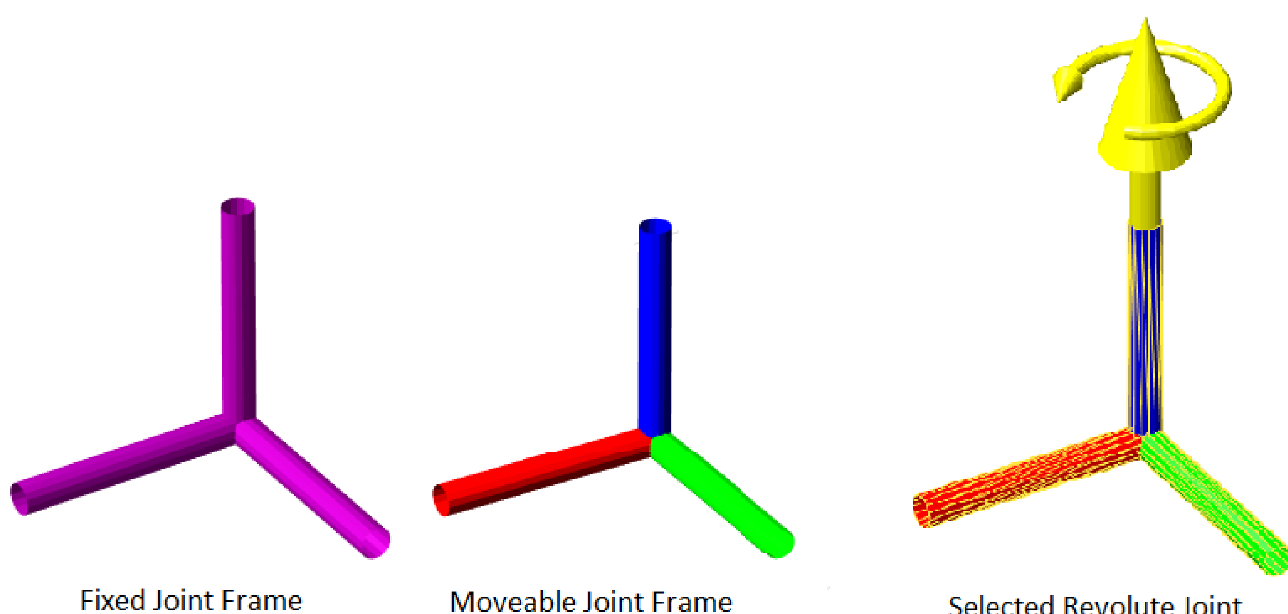
ax — Axes graphic handle
`Axes` object

Axes graphic handle, returned as an `Axes` object. This object contains the properties of the figure that the robot is plotted onto.

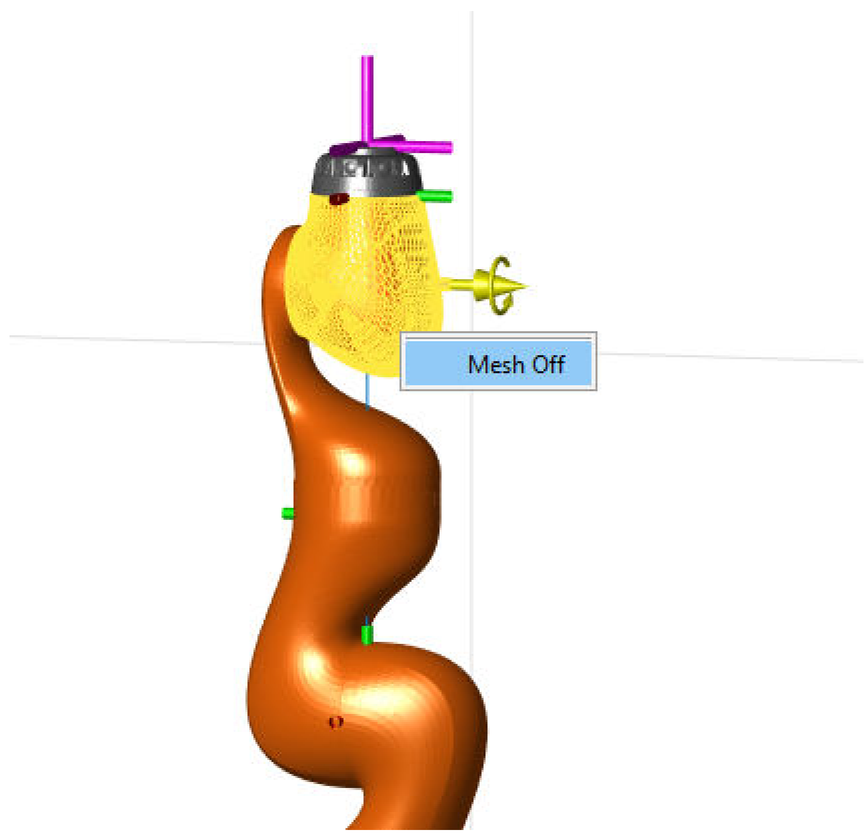
Tips

Your robot model has visual components associated with it. Each `rigidBody` object contains a coordinate frame that is displayed as the body frame. Each body also can have visual meshes associated with them. By default, both of these components are displayed automatically. You can inspect or modify the visual components of the rigid body tree display. Click body frames or visual meshes to highlight them in yellow and see the associated body name, index, and joint type. Right-click to toggle visibility of individual components.

- **Body Frames**: Individual body frames are displayed as a 3-axis coordinate frame. Fixed frames are pink frames. Movable joint types are displayed as RGB axes. You can click a body frame to see the axis of motion. Prismatic joints show a yellow arrow in the direction of the axis of motion and, revolute joints show a circular arrow around the rotation axis.



- **Visual Meshes**: Individual visual geometries are specified using `addVisual` or by using the `importrobot` to import a robot model with `.stl` files specified. By right-clicking individual bodies in a figure, you can turn off their meshes or specify the `Visuals` name-value pair to hide all visual geometries.



Version History

Introduced in R2016b

See Also

`showdetails` | `randomConfiguration` | `importrobot`

showdetails

Show details of robot model

Syntax

```
showdetails(robot)
```

Description

`showdetails(robot)` displays in the MATLAB command window the details of each body in the robot model. These details include the body name, associated joint name, joint type, parent name, and children names.

Examples

Attach Rigid Body and Joint to Rigid Body Tree

Add a rigid body and corresponding joint to a rigid body tree. Each `rigidBody` object contains a `rigidBodyJoint` object and must be added to the `rigidBodyTree` using `addBody`.

Create a rigid body tree.

```
rbtree = rigidBodyTree;
```

Create a rigid body with a unique name.

```
body1 = rigidBody('b1');
```

Create a revolute joint. By default, the `rigidBody` object comes with a fixed joint. Replace the joint by assigning a new `rigidBodyJoint` object to the `body1.Joint` property.

```
jnt1 = rigidBodyJoint('jnt1','revolute');
body1.Joint = jnt1;
```

Add the rigid body to the tree. Specify the body name that you are attaching the rigid body to. Because this is the first body, use the base name of the tree.

```
basename = rbtree.BaseName;
addBody(rbtree,body1,basename)
```

Use `showdetails` on the tree to confirm the rigid body and joint were added properly.

```
showdetails(rbtree)
```

```
--------------------
Robot: (1 bodies)

 Idx     Body Name   Joint Name   Joint Type    Parent Name(Idx)   Children Name(s)
 ---     ---------   ----------   ----------    ----------------   ----------------
   1            b1         jnt1     revolute             base(0)   
--------------------
```

Modify a Robot Rigid Body Tree Model

Make changes to an existing `rigidBodyTree` object. You can get replace joints, bodies and subtrees in the rigid body tree.

Load example robots as `rigidBodyTree` objects.

```
load exampleRobots.mat
```

View the details of the Puma robot using `showdetails`.

```
showdetails(puma1)
```

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

 Idx    Body Name   Joint Name   Joint Type    Parent Name(Idx)   Children Name(s)
 ---    ---------   ----------   ----------    ----------------   ----------------
   1           L1         jnt1     revolute             base(0)   L2(2)
   2           L2         jnt2     revolute               L1(1)   L3(3)
   3           L3         jnt3     revolute               L2(2)   L4(4)
   4           L4         jnt4     revolute               L3(3)   L5(5)
   5           L5         jnt5     revolute               L4(4)   L6(6)
   6           L6         jnt6     revolute               L5(5)   
--------------------
```

Get a specific body to inspect the properties. The only child of the `L3` body is the `L4` body. You can copy a specific body as well.

```
body3 = getBody(puma1,'L3');
childBody = body3.Children{1}
```

```
childBody = 
  rigidBody with properties:

            Name: 'L4'
           Joint: [1x1 rigidBodyJoint]
            Mass: 1
    CenterOfMass: [0 0 0]
         Inertia: [1 1 1 0 0 0]
          Parent: [1x1 rigidBody]
        Children: {[1x1 rigidBody]}
         Visuals: {}
      Collisions: {}
```

```
body3Copy = copy(body3);
```

Replace the joint on the `L3` body. You must create a new `Joint` object and use `replaceJoint` to ensure the downstream body geometry is unaffected. Call `setFixedTransform` if necessary to define a transform between the bodies instead of with the default identity matrices.

```
newJoint = rigidBodyJoint('prismatic');
replaceJoint(puma1,'L3',newJoint);

showdetails(puma1)
```

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

 Idx     Body Name       Joint Name       Joint Type    Parent Name(Idx)   Children Name(s)
 ---     ---------       ----------       ----------    ----------------   ----------------
   1            L1             jnt1         revolute             base(0)   L2(2)
   2            L2             jnt2         revolute               L1(1)   L3(3)
   3            L3        prismatic            fixed               L2(2)   L4(4)
   4            L4             jnt4         revolute               L3(3)   L5(5)
   5            L5             jnt5         revolute               L4(4)   L6(6)
   6            L6             jnt6         revolute               L5(5)
--------------------
```

Remove an entire body and get the resulting subtree using `removeBody`. The removed body is included in the subtree.

```
subtree = removeBody(puma1,'L4')
```

```
subtree =
  rigidBodyTree with properties:

     NumBodies: 3
        Bodies: {[1x1 rigidBody]  [1x1 rigidBody]  [1x1 rigidBody]}
          Base: [1x1 rigidBody]
     BodyNames: {'L4'  'L5'  'L6'}
      BaseName: 'L3'
       Gravity: [0 0 0]
    DataFormat: 'struct'
```

Remove the modified `L3` body. Add the original copied `L3` body to the `L2` body, followed by the returned subtree. The robot model remains the same. See a detailed comparison through `showdetails`.

```
removeBody(puma1,'L3');
addBody(puma1,body3Copy,'L2')
addSubtree(puma1,'L3',subtree)

showdetails(puma1)
```

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

 Idx     Body Name   Joint Name   Joint Type    Parent Name(Idx)   Children Name(s)
 ---     ---------   ----------   ----------    ----------------   ----------------
   1            L1         jnt1     revolute             base(0)   L2(2)
   2            L2         jnt2     revolute               L1(1)   L3(3)
   3            L3         jnt3     revolute               L2(2)   L4(4)
   4            L4         jnt4     revolute               L3(3)   L5(5)
   5            L5         jnt5     revolute               L4(4)   L6(6)
   6            L6         jnt6     revolute               L5(5)
--------------------
```

Build Manipulator Robot Using Denavit-Hartenberg Parameters

Use the Denavit-Hartenberg (DH) parameters of the Puma560® robot to build a robot. Each rigid body is added one at a time, with the child-to-parent transform specified by the joint object.

The DH parameters define the geometry of the robot with relation to how each rigid body is attached to its parent. For convenience, setup the parameters for the Puma560 robot in a matrix[1] on page 3-455. The Puma robot is a serial chain manipulator. The DH parameters are relative to the previous row in the matrix, corresponding to the previous joint attachment.

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

Create a rigid body tree object to build the robot.

```
robot = rigidBodyTree;
```

Create the first rigid body and add it to the robot. To add a rigid body:

1. Create a `rigidBody` object and give it a unique name.
2. Create a `rigidBodyJoint` object and give it a unique name.
3. Use `setFixedTransform` to specify the body-to-body transformation using DH parameters. The last element of the DH parameters, `theta`, is ignored because the angle is dependent on the joint position.
4. Call `addBody` to attach the first body joint to the base frame of the robot.

```
body1 = rigidBody('body1');
jnt1 = rigidBodyJoint('jnt1','revolute');

setFixedTransform(jnt1,dhparams(1,:),'dh');
body1.Joint = jnt1;

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

Create and add other rigid bodies to the robot. Specify the previous body name when calling `addBody` to attach it. Each fixed transform is relative to the previous joint coordinate frame.

```
body2 = rigidBody('body2');
jnt2 = rigidBodyJoint('jnt2','revolute');
body3 = rigidBody('body3');
jnt3 = rigidBodyJoint('jnt3','revolute');
body4 = rigidBody('body4');
jnt4 = rigidBodyJoint('jnt4','revolute');
body5 = rigidBody('body5');
jnt5 = rigidBodyJoint('jnt5','revolute');
body6 = rigidBody('body6');
jnt6 = rigidBodyJoint('jnt6','revolute');

setFixedTransform(jnt2,dhparams(2,:),'dh');
setFixedTransform(jnt3,dhparams(3,:),'dh');
setFixedTransform(jnt4,dhparams(4,:),'dh');
setFixedTransform(jnt5,dhparams(5,:),'dh');
```

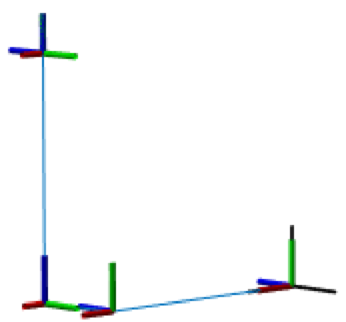
```
setFixedTransform(jnt6,dhparams(6,:),'dh');

body2.Joint = jnt2;
body3.Joint = jnt3;
body4.Joint = jnt4;
body5.Joint = jnt5;
body6.Joint = jnt6;

addBody(robot,body2,'body1')
addBody(robot,body3,'body2')
addBody(robot,body4,'body3')
addBody(robot,body5,'body4')
addBody(robot,body6,'body5')
```

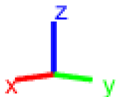
Verify that your robot was built properly by using the `showdetails` or `show` function. `showdetails` lists all the bodies in the MATLAB® command window. `show` displays the robot with a given configuration (home by default). Calls to `axis` modify the axis limits and hide the axis labels.

```
showdetails(robot)
```

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

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

```
show(robot);
axis([-0.5,0.5,-0.5,0.5,-0.5,0.5])
axis off
```





References

[1] Corke, P. I., and B. Armstrong-Helouvry. “A Search for Consensus among Model Parameters Reported for the PUMA 560 Robot.” *Proceedings of the 1994 IEEE International Conference on Robotics and Automation*, IEEE Computer. Soc. Press, 1994, pp. 1608–13. *DOI.org (Crossref)*, doi:10.1109/ROBOT.1994.351360.

Input Arguments

`robot` — Robot model
`rigidBodyTree` object

Robot model, specified as a `rigidBodyTree` object.

Version History

Introduced in R2016b

See Also

`show` | `replaceBody` | `replaceJoint`

subtree

Create subtree from robot model

Syntax

```
newSubtree = subtree(robot,bodyname)
```

Description

`newSubtree = subtree(robot,bodyname)` creates a new robot model using the parent name of the body specified by `bodyname` as the base name. All subsequently attached bodies (including the body with `bodyname` specified) are added to the subtree. The original `robot` model is unaffected.

Input Arguments

`robot` — Robot model
`rigidBodyTree` object

Robot model, specified as a `rigidBodyTree` object.

`bodyname` — Body name
string scalar | character vector

Body name, specified as a string scalar or character vector. This body must be on the robot model specified in `robot`.

Data Types: `char` | `string`

Output Arguments

`newSubtree` — Robot subtree
`rigidBodyTree` object

Robot subtree, returned as a `rigidBodyTree` object. This new subtree uses the parent name of the body specified by `bodyname` as the base name. All bodies that are attached in the previous robot model (including the body with `bodyname` specified) are added to the subtree.

Version History

Introduced in R2016b

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

When creating the `rigidBodyTree` object, use the syntax that specifies the `MaxNumBodies` as an upper bound for adding bodies to the robot model. You must also specify the `DataFormat` property as a name-value pair. For example:

```
robot = rigidBodyTree("MaxNumBodies",15,"DataFormat","row")
```

To minimize data usage, limit the upper bound to a number close to the expected number of bodies in the model. All data formats are supported for code generation. To use the dynamics functions, the data format must be set to `"row"` or `"column"`.

The `show` and `showdetails` functions do not support code generation.

See Also

`rigidBodyJoint` | `rigidBody` | `addBody` | `replaceBody`

velocityProduct

Joint torques that cancel velocity-induced forces

Syntax

`jointTorq = velocityProduct(robot,configuration,jointVel)`

Description

`jointTorq = velocityProduct(robot,configuration,jointVel)` computes the joint torques required to cancel the forces induced by the specified joint velocities under a certain joint configuration. Gravity torque is not included in this calculation.

Examples

Compute Velocity-Induced Joint Torques

Load a predefined KUKA LBR robot model, which is specified as a `RigidBodyTree` object.

```
load exampleRobots.mat lbr
```

Set the data format to `'row'`. For all dynamics calculations, the data format must be either `'row'` or `'column'`.

```
lbr.DataFormat = 'row';
```

Set the joint velocity vector.

```
qdot = [0 0 0.2 0.3 0 0.1 0];
```

Compute the joint torques required to cancel the velocity-induced joint torques at the robot home configuration (`[]` input). The velocity-induced joint torques equal the negative of the `velocityProduct` output.

```
tau = -velocityProduct(lbr,[],qdot);
```

Input Arguments

`robot` — Robot model
`rigidBodyTree` object

Robot model, specified as a `rigidBodyTree` object. To use the `velocityProduct` function, set the `DataFormat` property to either `'row'` or `'column'`.

`configuration` — Robot configuration
vector

Robot configuration, specified as a vector with positions for all nonfixed joints in the robot model. You can generate a configuration using `homeConfiguration(robot)`,

randomConfiguration(robot), or by specifying your own joint positions. To use the vector form of configuration, set the DataFormat property for the robot to either 'row' or 'column' .

jointVel — Joint velocities
vector

Joint velocities, specified as a vector. The number of joint velocities is equal to the velocity degrees of freedom of the robot. To use the vector form of jointVel, set the DataFormat property for the robot to either 'row' or 'column' .

Output Arguments

jointTorq — Joint torques
vector

Joint torques, specified as a vector. Each element corresponds to a torque applied to a specific joint.

More About

Dynamics Properties

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

- Mass — Mass of the rigid body in kilograms.
- CenterOfMass — Center of mass position of the rigid body, specified as a vector of the form [x y z]. The vector describes the location of the center of mass of the rigid body, relative to the body frame, in meters. The centerOfMass object function uses these rigid body property values when computing the center of mass of a robot.
- Inertia — Inertia of the rigid body, specified as a vector of the form [Ixx Iyy Izz Iyz Ixz Ixy]. The vector is relative to the body frame in kilogram square meters. The inertia tensor is a positive definite matrix of the form:

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

  The first three elements of the Inertia vector are the moment of inertia, which are the diagonal elements of the inertia tensor. The last three elements are the product of inertia, which are the off-diagonal elements of the inertia tensor.

For information related to the entire manipulator robot model, specify these rigidBodyTree object properties:

- Gravity — Gravitational acceleration experienced by the robot, specified as an [x y z] vector in m/s^2. By default, there is no gravitational acceleration.
- DataFormat — The input and output data format for the kinematics and dynamics functions, specified as "struct", "row", or "column".

Dynamics Equations

Manipulator rigid body dynamics are governed by this equation:

$$\frac{d}{dt}\begin{bmatrix} q \\ \dot{q} \end{bmatrix} = \begin{bmatrix} \dot{q} \\ M(q)^{-1}\left(-C(q,\dot{q})\dot{q} - G(q) - J(q)^T F_{Ext} + \tau\right) \end{bmatrix}$$

also written as:

$$M(q)\ddot{q} = -C(q,\dot{q})\dot{q} - G(q) - J(q)^T F_{Ext} + \tau$$

where:

- $M(q)$ — is a joint-space mass matrix based on the current robot configuration. Calculate this matrix by using the `massMatrix` object function.
- $C(q, \dot{q})$ — are the Coriolis terms, which are multiplied by $\dot{q}$ to calculate the velocity product. Calculate the velocity product by using by the `velocityProduct` object function.
- $G(q)$ — is the gravity torques and forces required for all joints to maintain their positions in the specified gravity `Gravity`. Calculate the gravity torque by using the `gravityTorque` object function.
- $J(q)$ — is the geometric Jacobian for the specified joint configuration. Calculate the geometric Jacobian by using the `geometricJacobian` object function.
- F_{Ext} — is a matrix of the external forces applied to the rigid body. Generate external forces by using the `externalForce` object function.
- τ — are the joint torques and forces applied directly as a vector to each joint.
- $q, \dot{q}, \ddot{q}$ — are the joint configuration, joint velocities, and joint accelerations, respectively, as individual vectors. For revolute joints, specify values in radians, rad/s, and rad/s^2, respectively. For prismatic joints, specify in meters, m/s, and m/s^2.

To compute the dynamics directly, use the `forwardDynamics` object function. The function calculates the joint accelerations for the specified combinations of the above inputs.

To achieve a certain set of motions, use the `inverseDynamics` object function. The function calculates the joint torques required to achieve the specified configuration, velocities, accelerations, and external forces.

Version History

Introduced in R2017a

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

When creating the `rigidBodyTree` object, use the syntax that specifies the `MaxNumBodies` as an upper bound for adding bodies to the robot model. You must also specify the `DataFormat` property as a name-value pair. For example:

```
robot = rigidBodyTree("MaxNumBodies",15,"DataFormat","row")
```

To minimize data usage, limit the upper bound to a number close to the expected number of bodies in the model. All data formats are supported for code generation. To use the dynamics functions, the data format must be set to `"row"` or `"column"`.

The `show` and `showdetails` functions do not support code generation.

See Also

`rigidBodyTree` | `inverseDynamics` | `gravityTorque` | `massMatrix`

writeAsFunction

Create `rigidBodyTree` code generating function

Syntax

```
writeAsFunction(robot,filename)
```

Description

`writeAsFunction(robot,filename)` creates a function file that constructs the `rigidBodyTree` object. The created function supports code generation.

Examples

Create Rigid Body Tree Code Generating Function

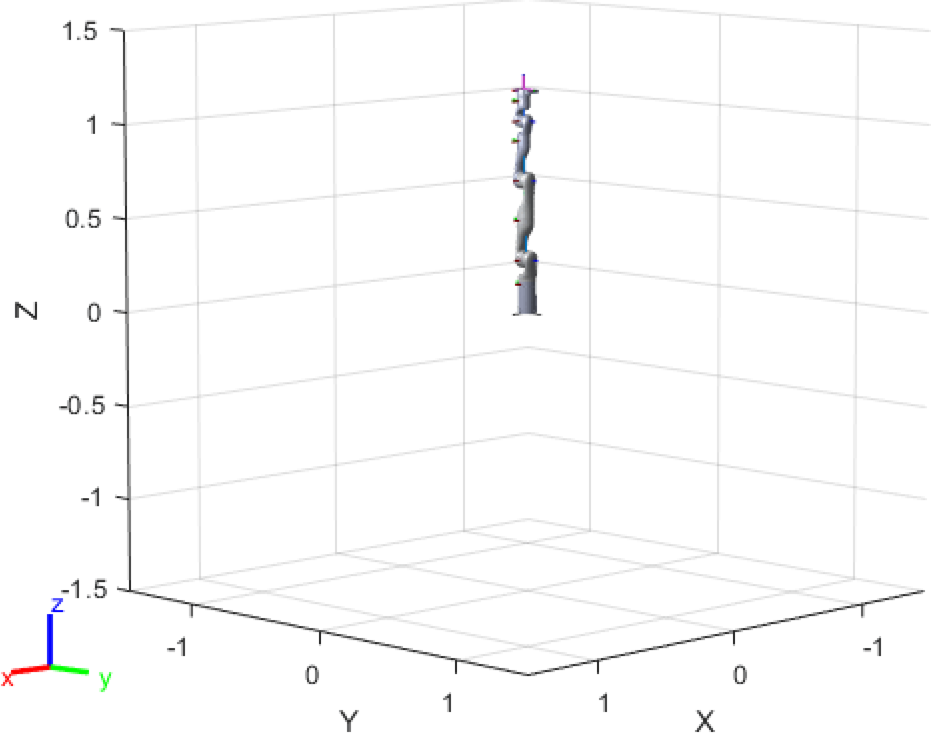
Load a robot model as a `rigidBodyTree` object.

```
robot = loadrobot("kinovaGen3")
```

```
robot = 
  rigidBodyTree with properties:

     NumBodies: 8
        Bodies: {[1x1 rigidBody]  [1x1 rigidBody]  [1x1 rigidBody]  [1x1 rigidBody]  [1x1 rigidBo
          Base: [1x1 rigidBody]
     BodyNames: {'Shoulder_Link'  'HalfArm1_Link'  'HalfArm2_Link'  'ForeArm_Link'  'Wrist1_Link
      BaseName: 'base_link'
       Gravity: [0 0 0]
    DataFormat: 'struct'
```

Show the robot model in a figure.

```
show(robot);
```



Create a code generating function that constructs the `rigidBodyTree` object.

```
writeAsFunction(robot,'KG3Codegen')
```

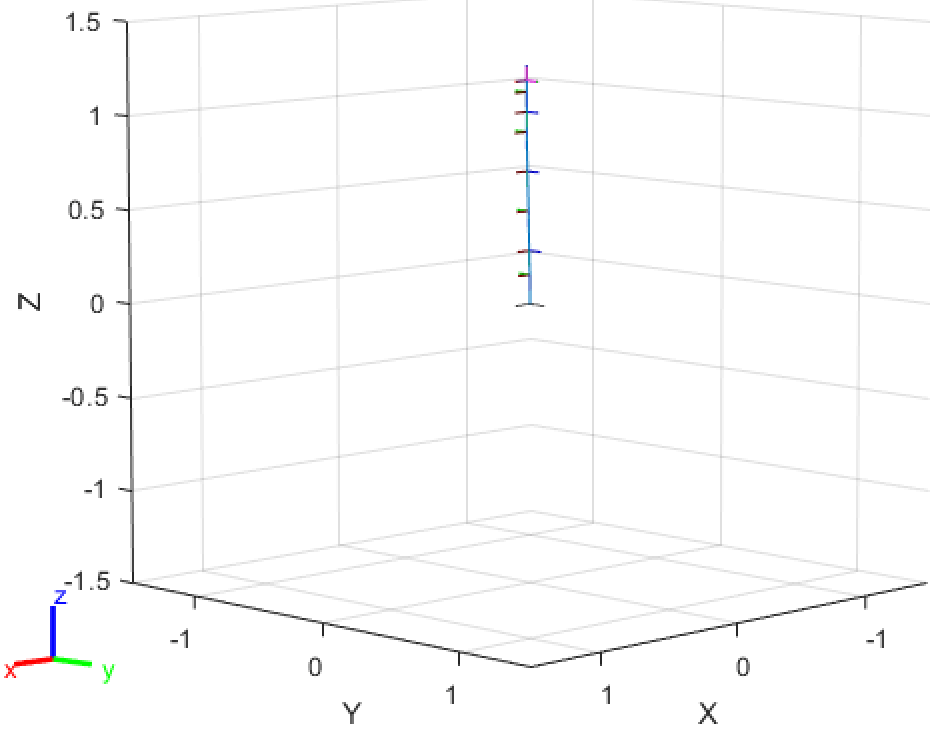
Construct the robot model using the generated function.

```
rbt = KG3Codegen
```

```
rbt = 
  rigidBodyTree with properties:

     NumBodies: 8
        Bodies: {[1x1 rigidBody]  [1x1 rigidBody]  [1x1 rigidBody]  [1x1 rigidBody]  [1x1 rigidBo
          Base: [1x1 rigidBody]
     BodyNames: {'Shoulder_Link'  'HalfArm1_Link'  'HalfArm2_Link'  'ForeArm_Link'  'Wrist1_Link
      BaseName: 'base_link'
       Gravity: [0 0 0]
    DataFormat: 'struct'
```

Show the robot model in a figure.

```
show(rbt);
```



Input Arguments

robot — Robot model
`rigidBodyTree` object

Robot model, specified as a `rigidBodyTree` object.

filename — Name of function file
string scalar | character vector

Name of the function file, specified as a string scalar or character vector. The name must be a valid MATLAB name (must start with a letter and contain only letters, numbers and underscores).

Example: `"iiwa14Codegen"`

Data Types: `char` | `string`

Version History

Introduced in R2021a

See Also

Functions
`importrobot` | `loadrobot`

Objects
`rigidBodyTree`

bodyInfo

Import information for body

Syntax

```
info = bodyInfo(importInfo,bodyName)
```

Description

`info = bodyInfo(importInfo,bodyName)` returns the import information for a body in a `rigidBodyTree` object that is created from calling `importrobot`. Specify the `rigidBodyTreeImportInfo` object from the import process.

Input Arguments

`importInfo` — Robot import information
`rigidBodyTreeImportInfo` object

Robot import information, specified as a `rigidBodyTreeImportInfo` object. This object is returned when you use `importrobot`.

bodyName — Name of body
character vector | string scalar

Name of a body in the `rigidBodyTree` object that was created using `importrobot`, specified as a character vector or string scalar. Partial string matching is accepted and returns a cell array of structures that match the partial string.

Example: `'Body01'`

Data Types: `char` | `string`

Output Arguments

`info` — Import information for specific component
structure | cell array of structures

Import information for specific component, returned as a structure or cell array of structures. This structure contains the information about the imported blocks from Simscape Multibody and the associated components in the `rigidBodyTree` object. The fields of each structure are:

- `BodyName` — Name of the body in the `rigidBodyTree` object.
- `JointName` — Name of the joint associated with `BodyName`.
- `BodyBlocks` — Blocks used from the Simscape Multibody model.
- `JointBlocks` — Joint blocks used from the Simscape Multibody model.

Version History

Introduced in R2018b

See Also

`importrobot` | `rigidBodyTreeImportInfo` | `rigidBodyTree` | `showdetails`

bodyInfoFromBlock

Import information for block name

Syntax

```
info = bodyInfo(importInfo,blockName)
```

Description

`info = bodyInfo(importInfo,blockName)` returns the import information for a block in a Simscape Multibody model that is imported from calling `importrobot`. Specify the `rigidBodyTreeImportInfo` object from the import process.

Input Arguments

`importInfo` — Robot import information
`rigidBodyTreeImportInfo` object

Robot import information, specified as a `rigidBodyTreeImportInfo` object. This object is returned when you use `importrobot`.

`blockName` — Name of block
character vector | string scalar

Name of a block in the Simscape Multibody model that was imported using `importrobot`, specified as a character vector or string scalar. Partial string matching is accepted and returns a cell array of structures that match the partial string.

Example: `'Prismatic Joint 2'`

Data Types: `char` | `string`

Output Arguments

`info` — Import information for specific component
structure | cell array of structures

Import information for specific component, returned as a structure or cell array of structures. This structure contains the information about the imported blocks from Simscape Multibody and the associated components in the `rigidBodyTree` object. The fields of each structure are:

- `BodyName` — Name of the body in the `rigidBodyTree` object.
- `JointName` — Name of the joint associated with `BodyName`.
- `BodyBlocks` — Blocks used from the Simscape Multibody model.
- `JointBlocks` — Joint blocks used from the Simscape Multibody model.

Version History

Introduced in R2018b

See Also

`importrobot` | `rigidBodyTreeImportInfo` | `rigidBodyTree` | `showdetails`

bodyInfoFromJoint

Import information for given joint name

Syntax

```
info = bodyInfo(importInfo,jointName)
```

Description

`info = bodyInfo(importInfo,jointName)` returns the import information for a joint in a `rigidBodyTree` object that is created from calling `importrobot`. Specify the `rigidBodyTreeImportInfo` object from the import process.

Input Arguments

`importInfo` — Robot import information
`rigidBodyTreeImportInfo` object

Robot import information, specified as a `rigidBodyTreeImportInfo` object. This object is returned when you use `importrobot`.

`jointName` — Name of joint
character vector | string scalar

Name of a joint in the `rigidBodyTree` object that was created using `importrobot`, specified as a character vector or string scalar. Partial string matching is accepted and returns a cell array of structures that match the partial string.

Example: `'Joint01'`

Data Types: `char` | `string`

Output Arguments

`info` — Import information for specific component
structure | cell array of structures

Import information for specific component, specified as a structure or cell array of structures. This structure contains the information about the imported blocks from Simscape Multibody and the associated components in the `rigidBodyTree` object. The fields of each structure are:

- `BodyName` — Name of the body in the `rigidBodyTree` object.
- `JointName` — Name of the joint associated with `BodyName`.
- `BodyBlocks` — Blocks used from the Simscape Multibody model.
- `JointBlocks` — Joint blocks used from the Simscape Multibody model.

Version History

Introduced in R2018b

See Also

`importrobot` | `rigidBodyTreeImportInfo` | `rigidBodyTree`

showdetails

Display details of imported robot

Syntax

```
showdetails(importInfo)
```

Description

`showdetails(importInfo)` displays the details of each body in the `rigidBodyTree` object that is created from calling `importrobot`. Specify the `rigidBodyTreeImportInfo` object from the import process.

The list shows the bodies with their associated joint name, joint type, source blocks, parent body name, and children body names. The list also provides highlight links to the associated blocks used in the Simscape Multibody model.

Note The `Highlight` links assume the block names are unchanged.

Input Arguments

`importInfo` — Robot import information
`rigidBodyTreeImportInfo` object

Robot import information, specified as a `rigidBodyTreeImportInfo` object. This object is returned when you use `importrobot`.

Version History

Introduced in R2018b

See Also

`importrobot` | `rigidBodyTreeImportInfo` | `rigidBodyTree`

attach

Attach target robot platform to source robot platform

Syntax

```
attach(platform,targetname,attachingbodyname)
attach( ___ ,Name=Value)
```

Description

`attach(platform,targetname,attachingbodyname)` attaches the non-`rigidBodyTree`-based target platform `targetname` to the body `attachingbodyname` of the `rigidBodyTree`-based source platform `platform`.

`attach( ___ ,Name=Value)` specifies options using one or more name-value arguments, in addition to all input arguments from the previous syntax.

Examples

Perform Pick and Place in Robot Scenario

Create a `robotScenario` object.

```
scenario = robotScenario(UpdateRate=1,StopTime=10);
```

Create a `rigidBodyTree` object of the Franka Emika Panda manipulator using `loadrobot`.

```
robotRBT = loadrobot("frankaEmikaPanda");
```

Create a `rigidBodyTree`-based `robotPlatform` object using the manipulator model.

```
robot = robotPlatform("Manipulator",scenario, ...
                      RigidBodyTree=robotRBT);
```

Create a non-`rigidBodyTree`-based `robotPlatform` object of a box to manipulate. Specify the mesh type and size.

```
box = robotPlatform("Box",scenario,Collision="mesh", ...
                    InitialBasePosition=[0.5 0.15 0.278]);
updateMesh(box,"Cuboid",Collision="mesh",Size=[0.06 0.06 0.1])
```

Visualize the scenario.

```
ax = show3D(scenario,Collisions="on");
view(79,36)
light
```

Specify the initial and the pick-up joint configuration of the manipulator, to move the manipulator from its initial pose to close to the box.

```
initialConfig = homeConfiguration(robot.RigidBodyTree);
pickUpConfig = [0.2371 -0.0200 0.0542 -2.2272 0.0013 ...
                2.2072 -0.9670 0.0400 0.0400];
```

Create an RRT path planner using the `manipulatorRRT` object, and specify the manipulator model.

```
planner = manipulatorRRT(robot.RigidBodyTree,scenario.CollisionMeshes);
planner.IgnoreSelfCollision = true;
```

Plan the path between the initial and the pick-up joint configurations. Then, to visualize the entire path, interpolate the path into small steps.

```
rng("default")
path = plan(planner,initialConfig,pickUpConfig);
path = interpolate(planner,path,25);
```

Set up the simulation.

```
setup(scenario)
```

Check the collision before manipulator picks up the box.

```
checkCollision(robot,"Box", ...
               IgnoreSelfCollision="on")
```

```
ans = logical
   0
```

Move the joints of the manipulator along the path and visualize the scenario.

```
helperRobotMove(path,robot,scenario,ax)
```

Check the collision after manipulator picks up the box.

```
checkCollision(robot,"Box", ...
               IgnoreSelfCollision="on")
```

```
ans = logical
   1
```

Use the `attach` function to attach the box to the gripper of the manipulator.

```
attach(robot,"Box","panda_hand", ...
       ChildToParentTransform=trvec2tform([0 0 0.1]))
```

Specify the drop-off joint configuration of the manipulator to move the manipulator from its pick-up pose to the box drop-off pose.

```
dropOffConfig = [-0.6564 0.2885 -0.3187 -1.5941 0.1103 ...
                 1.8678 -0.2344 0.04 0.04];
```

Plan the path between the pick-up and drop-off joint configurations.

```
path = plan(planner,pickUpConfig,dropOffConfig);
path = interpolate(planner,path,25);
```

Move the joints of the manipulator along the path and visualize the scenario.

```
helperRobotMove(path,robot,scenario,ax)
```

Use the `detach` function to detach the box from the manipulator gripper.

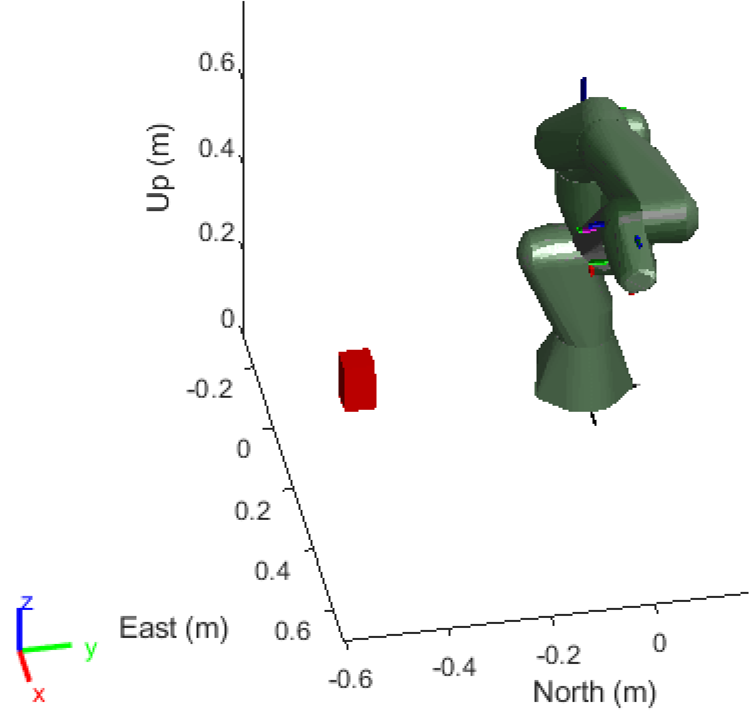
```
detach(robot)
```

Plan the path between the drop-off and initial joint configurations to move the manipulator from its box drop-off pose to its initial pose.

```
path = plan(planner,dropOffConfig,initialConfig);
path = interpolate(planner,path,25);
```

Move the joints of the manipulator along the path and visualize the scenario.

```
helperRobotMove(path,robot,scenario,ax)
```



Helper function to move the joints of the manipulator.

```
function helperRobotMove(path,robot,scenario,ax)
    for idx = 1:size(path,1)
        jointConfig = path(idx,:);
        move(robot,"joint",jointConfig)
        show3D(scenario,fastUpdate=true,Parent=ax,Collisions="on");
        drawnow
        advance(scenario);
    end
end
```

Input Arguments

`platform` — `rigidBodyTree`-based source platform
`robotPlatform` object

`rigidBodyTree`-based source platform, specified as a `robotPlatform` object.

`targetname` — Non-`rigidBodyTree`-based target platform
string scalar | character vector

Non-`rigidBodyTree`-based target platform, specified as a string scalar or character vector.

Example: `"Box"`

Data Types: `char` | `string`

`attachingbodyname` — Body of source robot platform
string scalar | character vector

Body of the source robot platform, specified as a string scalar or character vector.

Example: `"panda_hand"`

Data Types: `char` | `string`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Example: `ChildToParentTransform=trvec2tform([0 0 0.1])`

`ChildToParentTransform` — Transformation of attached target platform body relative to source platform attaching body
`eye(4)` (default) | 4-by-4 homogeneous transformation matrix

Transformation of the attached target platform body relative to the source platform attaching body, specified as a 4-by-4 homogeneous transformation matrix.

Example: `ChildToParentTransform=trvec2tform([0 0 0.1])`

Data Types: `single` | `double`

`Collision` — Collision object to add to target platform mesh
`collisionBox` object | `collisionCapsule` object | `collisionCylinder` object | `collisionMesh` object | `collisionSphere` object

Collision object to add to target platform mesh, specified as an externally created collision object such as a `collisionBox`, `collisionCapsule`, `collisionCylinder`, `collisionMesh`, or `collisionSphere`. If you do not specify a collision object, then the target platform uses its default collision mesh.

Version History

Introduced in R2023a

See Also

Objects
robotPlatform | robotScenario | robotSensor

Functions
checkCollision | detach | move | read | updateMesh

checkCollision

Check collision between robot platform and target bodies

Syntax

```
[iscolliding,separationdist,witnesspts] = checkCollision(platform,targetbody)
[ ___ ] = checkCollision( ___ ,Name=Value)
```

Description

`[iscolliding,separationdist,witnesspts] = checkCollision(platform,targetbody)` checks for collision between the `rigidBodyTree`-based robot platform `platform` and a set of target bodies `targetbody`.

`[ ___ ] = checkCollision( ___ ,Name=Value)` specifies options using one or more name-value arguments, in addition to all arguments from the previous syntax.

Examples

Perform Pick and Place in Robot Scenario

Create a `robotScenario` object.

```
scenario = robotScenario(UpdateRate=1,StopTime=10);
```

Create a `rigidBodyTree` object of the Franka Emika Panda manipulator using `loadrobot`.

```
robotRBT = loadrobot("frankaEmikaPanda");
```

Create a `rigidBodyTree`-based `robotPlatform` object using the manipulator model.

```
robot = robotPlatform("Manipulator",scenario, ...
                      RigidBodyTree=robotRBT);
```

Create a non-`rigidBodyTree`-based `robotPlatform` object of a box to manipulate. Specify the mesh type and size.

```
box = robotPlatform("Box",scenario,Collision="mesh", ...
                    InitialBasePosition=[0.5 0.15 0.278]);
updateMesh(box,"Cuboid",Collision="mesh",Size=[0.06 0.06 0.1])
```

Visualize the scenario.

```
ax = show3D(scenario,Collisions="on");
view(79,36)
light
```

Specify the initial and the pick-up joint configuration of the manipulator, to move the manipulator from its initial pose to close to the box.

```
initialConfig = homeConfiguration(robot.RigidBodyTree);
pickUpConfig = [0.2371 -0.0200 0.0542 -2.2272 0.0013 ...
                2.2072 -0.9670 0.0400 0.0400];
```

Create an RRT path planner using the `manipulatorRRT` object, and specify the manipulator model.

```
planner = manipulatorRRT(robot.RigidBodyTree,scenario.CollisionMeshes);
planner.IgnoreSelfCollision = true;
```

Plan the path between the initial and the pick-up joint configurations. Then, to visualize the entire path, interpolate the path into small steps.

```
rng("default")
path = plan(planner,initialConfig,pickUpConfig);
path = interpolate(planner,path,25);
```

Set up the simulation.

```
setup(scenario)
```

Check the collision before manipulator picks up the box.

```
checkCollision(robot,"Box", ...
               IgnoreSelfCollision="on")
```

```
ans = logical
   0
```

Move the joints of the manipulator along the path and visualize the scenario.

```
helperRobotMove(path,robot,scenario,ax)
```

Check the collision after manipulator picks up the box.

```
checkCollision(robot,"Box", ...
               IgnoreSelfCollision="on")
```

```
ans = logical
   1
```

Use the `attach` function to attach the box to the gripper of the manipulator.

```
attach(robot,"Box","panda_hand", ...
       ChildToParentTransform=trvec2tform([0 0 0.1]))
```

Specify the drop-off joint configuration of the manipulator to move the manipulator from its pick-up pose to the box drop-off pose.

```
dropOffConfig = [-0.6564 0.2885 -0.3187 -1.5941 0.1103 ...
                 1.8678 -0.2344 0.04 0.04];
```

Plan the path between the pick-up and drop-off joint configurations.

```
path = plan(planner,pickUpConfig,dropOffConfig);
path = interpolate(planner,path,25);
```

Move the joints of the manipulator along the path and visualize the scenario.

```
helperRobotMove(path,robot,scenario,ax)
```

Use the `detach` function to detach the box from the manipulator gripper.

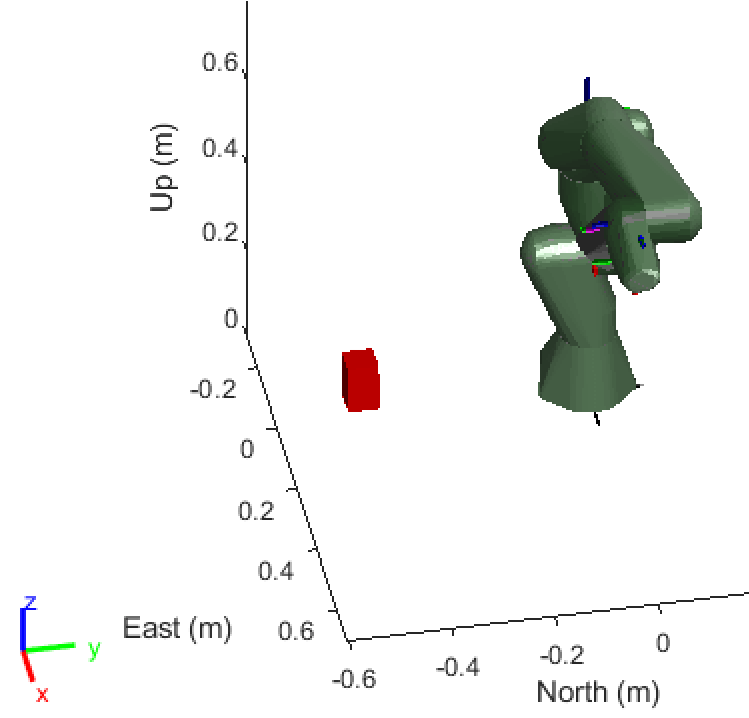
```
detach(robot)
```

Plan the path between the drop-off and initial joint configurations to move the manipulator from its box drop-off pose to its initial pose.

```
path = plan(planner,dropOffConfig,initialConfig);
path = interpolate(planner,path,25);
```

Move the joints of the manipulator along the path and visualize the scenario.

```
helperRobotMove(path,robot,scenario,ax)
```



Helper function to move the joints of the manipulator.

```
function helperRobotMove(path,robot,scenario,ax)
    for idx = 1:size(path,1)
        jointConfig = path(idx,:);
        move(robot,"joint",jointConfig)
        show3D(scenario,fastUpdate=true,Parent=ax,Collisions="on");
        drawnow
        advance(scenario);
    end
end
```

Input Arguments

`platform` — rigidBodyTree-based robot platform
`robotPlatform` object

`rigidBodyTree`-based robot platform, specified as a `robotPlatform` object.

`targetbody` — Non-`rigidBodyTree`-based target platforms
string scalar | character vector | string array | cell array of character vectors

Non-`rigidBodyTree`-based target platforms, specified as a string scalar, character vector, string array, or cell array of character vectors.

Example: `"Box"`

Example: `'Box'`

Example: `["Box","Can"]`

Example: `{'Box','Can'}`

Data Types: `char` | `string`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Example: `Exhaustive="on"`

`Exhaustive` — Exhaustively check for all collisions
`"off"` (default) | `"on"`

Exhaustively check for all collisions, specified as `"on"` or `"off"`. Setting this to `"off"` interrupts collision checking when the first collision is found. If returned early, `separationdist` and `witnesspts` values for incomplete checks are `Inf`. Setting this to `"on"` computes the separation distance and witness points for all pairs.

Example: `Exhaustive="on"`

Data Types: `char` | `string`

`IgnoreSelfCollision` — Skip checking for collisions between robot bodies and base
`"off"` (default) | `"on"`

Skip checking for collisions between robot bodies and the base, specified as `"on"` or `"off"`.

Example: `IgnoreSelfCollision="on"`

Data Types: `char` | `string`

`SkippedSelfCollisions` — Body pairs skipped for checking self-collisions
`"parent"` (default) | `"adjacent"`

Body pairs skipped for checking self-collisions, specified as `"parent"` or `"adjacent"`.

- `"parent"` — Skips collision checking between child and parent bodies.

- "adjacent" — Skips collision checking between bodies on adjacent indices.

Example: SkippedSelfCollisions="adjacent"

Data Types: char | string

Output Arguments

iscolliding — Indicator for self-collision and collision with target bodies
two-element logical vector | logical scalar

Indicator for self-collision and collision with target bodies, returned as a two-element logical vector or logical scalar. If IgnoreSelfCollision is set to "on", then iscolliding is a logical scalar that indicates whether the robot platform collides with target bodies.

Data Types: logical

separationdist — Separation distance between target bodies and robot platform bodies including base
(*N*+1)-by-(*N*+*M*+1) matrix | (*N*+1)-by-*M* matrix

Separation distance between target bodies and the robot platform bodies including the base, returned as a (*N*+1)-by-(*N*+*M*+1) matrix. *N* is the number of robot platform bodies. *M* is the number of target bodies. If IgnoreSelfCollision is set to "on", the separationdist is an (*N*+1)-by-*M* matrix with column indices corresponding to the target bodies.

Returns Inf for incomplete checks.

Data Types: double

witnesspts — Witness points between bodies of robot platform including base
3(*N*+1)-by-2(*N*+1) matrix

Witness points between bodies of the robot platform including the base, returned as a 3(*N*+1)-by-2(*N*+1) matrix. *N* is the number of bodies. The matrix has the format:

$$\begin{bmatrix} Wr_{1_1} & Wr_{1_2} & \cdots & Wr_{1_(N)} & Wr_{1_(N+1)} \\ Wr_{2_1} & Wr_{2_2} & \cdots & Wr_{2_(N)} & Wr_{2_(N+1)} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ Wr_{(N+1)_1} & Wr_{(N+1)_2} & \cdots & Wr_{(N+1)_(N)} & Wr_{(N+1)_(N+1)} \end{bmatrix}$$

Each element, *Wr*, is a 3-by-2 matrix that gives the nearest [x y z] points on the two corresponding bodies in the robot platform. The final row and column correspond to the robot platform base.

Returns Inf for incomplete checks.

Data Types: double

Version History

Introduced in R2023a

See Also

Objects
robotPlatform | robotScenario | robotSensor

Functions
attach | detach | move | read | updateMesh

detach

Detach target robot platform from source robot platform

Syntax

```
detach(platform)
```

Description

`detach(platform)` detaches the attached, non-`rigidBodyTree`-based platform from the `rigidBodyTree`-based source robot platform `platform`.

Examples

Perform Pick and Place in Robot Scenario

Create a `robotScenario` object.

```
scenario = robotScenario(UpdateRate=1,StopTime=10);
```

Create a `rigidBodyTree` object of the Franka Emika Panda manipulator using `loadrobot`.

```
robotRBT = loadrobot("frankaEmikaPanda");
```

Create a `rigidBodyTree`-based `robotPlatform` object using the manipulator model.

```
robot = robotPlatform("Manipulator",scenario, ...
                      RigidBodyTree=robotRBT);
```

Create a non-`rigidBodyTree`-based `robotPlatform` object of a box to manipulate. Specify the mesh type and size.

```
box = robotPlatform("Box",scenario,Collision="mesh", ...
                    InitialBasePosition=[0.5 0.15 0.278]);
updateMesh(box,"Cuboid",Collision="mesh",Size=[0.06 0.06 0.1])
```

Visualize the scenario.

```
ax = show3D(scenario,Collisions="on");
view(79,36)
light
```

Specify the initial and the pick-up joint configuration of the manipulator, to move the manipulator from its initial pose to close to the box.

```
initialConfig = homeConfiguration(robot.RigidBodyTree);
pickUpConfig = [0.2371 -0.0200 0.0542 -2.2272 0.0013 ...
                2.2072 -0.9670 0.0400 0.0400];
```

Create an RRT path planner using the `manipulatorRRT` object, and specify the manipulator model.

```
planner = manipulatorRRT(robot.RigidBodyTree,scenario.CollisionMeshes);
planner.IgnoreSelfCollision = true;
```

Plan the path between the initial and the pick-up joint configurations. Then, to visualize the entire path, interpolate the path into small steps.

```
rng("default")
path = plan(planner,initialConfig,pickUpConfig);
path = interpolate(planner,path,25);
```

Set up the simulation.

```
setup(scenario)
```

Check the collision before manipulator picks up the box.

```
checkCollision(robot,"Box", ...
               IgnoreSelfCollision="on")
```

```
ans = logical
   0
```

Move the joints of the manipulator along the path and visualize the scenario.

```
helperRobotMove(path,robot,scenario,ax)
```

Check the collision after manipulator picks up the box.

```
checkCollision(robot,"Box", ...
               IgnoreSelfCollision="on")
```

```
ans = logical
   1
```

Use the `attach` function to attach the box to the gripper of the manipulator.

```
attach(robot,"Box","panda_hand", ...
       ChildToParentTransform=trvec2tform([0 0 0.1]))
```

Specify the drop-off joint configuration of the manipulator to move the manipulator from its pick-up pose to the box drop-off pose.

```
dropOffConfig = [-0.6564 0.2885 -0.3187 -1.5941 0.1103 ...
                 1.8678 -0.2344 0.04 0.04];
```

Plan the path between the pick-up and drop-off joint configurations.

```
path = plan(planner,pickUpConfig,dropOffConfig);
path = interpolate(planner,path,25);
```

Move the joints of the manipulator along the path and visualize the scenario.

```
helperRobotMove(path,robot,scenario,ax)
```

Use the `detach` function to detach the box from the manipulator gripper.

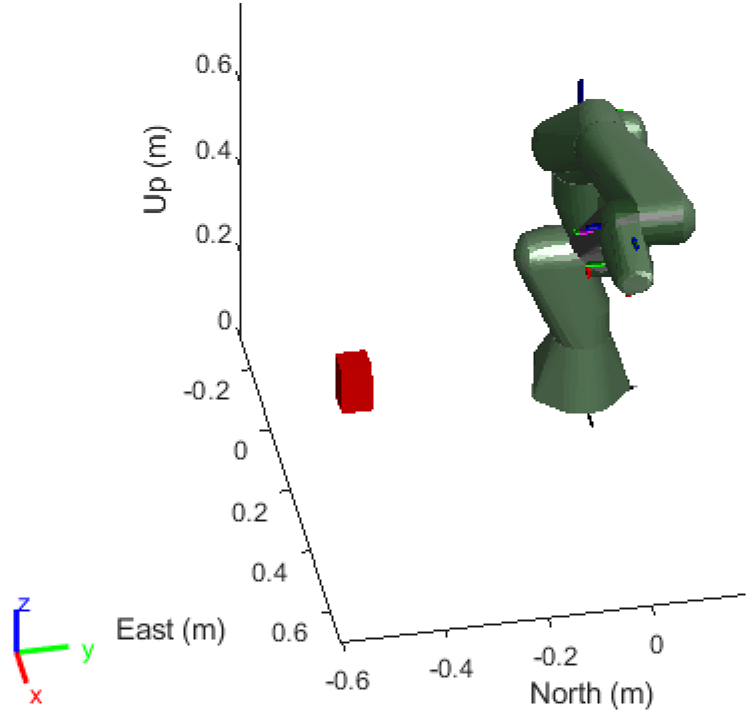
```
detach(robot)
```

Plan the path between the drop-off and initial joint configurations to move the manipulator from its box drop-off pose to its initial pose.

```
path = plan(planner,dropOffConfig,initialConfig);
path = interpolate(planner,path,25);
```

Move the joints of the manipulator along the path and visualize the scenario.

```
helperRobotMove(path,robot,scenario,ax)
```



Helper function to move the joints of the manipulator.

```
function helperRobotMove(path,robot,scenario,ax)
    for idx = 1:size(path,1)
        jointConfig = path(idx,:);
        move(robot,"joint",jointConfig)
        show3D(scenario,fastUpdate=true,Parent=ax,Collisions="on");
        drawnow
        advance(scenario);
    end
end
```

Input Arguments

platform — rigidBodyTree-based source platform
robotPlatform object

rigidBodyTree-based source platform, specified as a robotPlatform object.

Version History

Introduced in R2023a

See Also

Objects
robotPlatform | robotScenario | robotSensor

Functions
attach | checkCollision | move | read | updateMesh

move

Move robot platform in scenario

Syntax

```
move(platform,type,motion)
move(platform,type,config)
```

Description

`move(platform,type,motion)` moves the robot platform of type `"base"` in the scenario according to the specified motion.

`move(platform,type,config)` moves the joints of `rigidBodyTree`-based robot platform of type `"joint"` in the scenario according to the specified joint configuration.

Examples

Perform Pick and Place in Robot Scenario

Create a `robotScenario` object.

```
scenario = robotScenario(UpdateRate=1,StopTime=10);
```

Create a `rigidBodyTree` object of the Franka Emika Panda manipulator using `loadrobot`.

```
robotRBT = loadrobot("frankaEmikaPanda");
```

Create a `rigidBodyTree`-based `robotPlatform` object using the manipulator model.

```
robot = robotPlatform("Manipulator",scenario, ...
                      RigidBodyTree=robotRBT);
```

Create a non-`rigidBodyTree`-based `robotPlatform` object of a box to manipulate. Specify the mesh type and size.

```
box = robotPlatform("Box",scenario,Collision="mesh", ...
                    InitialBasePosition=[0.5 0.15 0.278]);
updateMesh(box,"Cuboid",Collision="mesh",Size=[0.06 0.06 0.1])
```

Visualize the scenario.

```
ax = show3D(scenario,Collisions="on");
view(79,36)
light
```

Specify the initial and the pick-up joint configuration of the manipulator, to move the manipulator from its initial pose to close to the box.

```
initialConfig = homeConfiguration(robot.RigidBodyTree);
pickUpConfig = [0.2371 -0.0200 0.0542 -2.2272 0.0013 ...
                2.2072 -0.9670 0.0400 0.0400];
```

Create an RRT path planner using the `manipulatorRRT` object, and specify the manipulator model.

```
planner = manipulatorRRT(robot.RigidBodyTree,scenario.CollisionMeshes);
planner.IgnoreSelfCollision = true;
```

Plan the path between the initial and the pick-up joint configurations. Then, to visualize the entire path, interpolate the path into small steps.

```
rng("default")
path = plan(planner,initialConfig,pickUpConfig);
path = interpolate(planner,path,25);
```

Set up the simulation.

```
setup(scenario)
```

Check the collision before manipulator picks up the box.

```
checkCollision(robot,"Box", ...
               IgnoreSelfCollision="on")
```

```
ans = logical
   0
```

Move the joints of the manipulator along the path and visualize the scenario.

```
helperRobotMove(path,robot,scenario,ax)
```

Check the collision after manipulator picks up the box.

```
checkCollision(robot,"Box", ...
               IgnoreSelfCollision="on")
```

```
ans = logical
   1
```

Use the `attach` function to attach the box to the gripper of the manipulator.

```
attach(robot,"Box","panda_hand", ...
       ChildToParentTransform=trvec2tform([0 0 0.1]))
```

Specify the drop-off joint configuration of the manipulator to move the manipulator from its pick-up pose to the box drop-off pose.

```
dropOffConfig = [-0.6564 0.2885 -0.3187 -1.5941 0.1103 ...
                 1.8678 -0.2344 0.04 0.04];
```

Plan the path between the pick-up and drop-off joint configurations.

```
path = plan(planner,pickUpConfig,dropOffConfig);
path = interpolate(planner,path,25);
```

Move the joints of the manipulator along the path and visualize the scenario.

```
helperRobotMove(path,robot,scenario,ax)
```

Use the `detach` function to detach the box from the manipulator gripper.

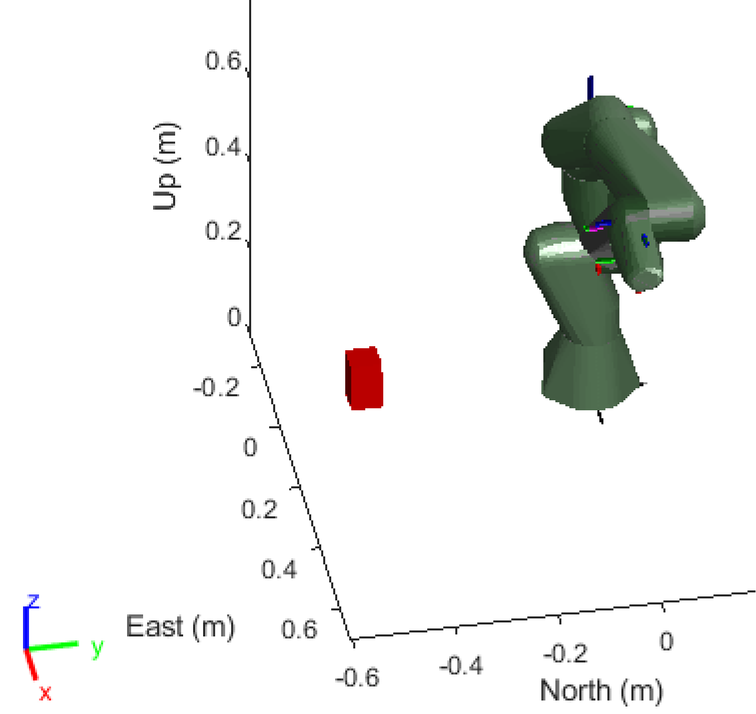
```
detach(robot)
```

Plan the path between the drop-off and initial joint configurations to move the manipulator from its box drop-off pose to its initial pose.

```
path = plan(planner,dropOffConfig,initialConfig);
path = interpolate(planner,path,25);
```

Move the joints of the manipulator along the path and visualize the scenario.

```
helperRobotMove(path,robot,scenario,ax)
```



Helper function to move the joints of the manipulator.

```
function helperRobotMove(path,robot,scenario,ax)
    for idx = 1:size(path,1)
        jointConfig = path(idx,:);
        move(robot,"joint",jointConfig)
        show3D(scenario,fastUpdate=true,Parent=ax,Collisions="on");
        drawnow
        advance(scenario);
    end
end
```

Input Arguments

`platform` — Robot platform in scenario
`robotPlatform` object

Robot platform in the scenario, specified as a `robotPlatform` object.

`type` — Type of robot platform
`"base"` | `"joint"`

Type of robot platform, specified as `"base"` or `"joint"`.

Data Types: `char` | `string`

`motion` — Robot platform motion at current instance in scenario
16-element vector

Robot platform motion at the current instance in the scenario, specified as a 16-element vector with these elements in order:

- `[x y z]` — Positions in *xyz*-axes in meters.
- `[vx vy vz]` — Velocities in *xyz*-directions in meters per second.
- `[ax ay az]` — Accelerations in *xyz*-directions in meters per second squared.
- `[qw qx qy qz]` — Quaternion vector for orientation.
- `[wx wy wz]` — Angular velocities in radians per second.

Data Types: `single` | `double`

`config` — Robot platform joint configuration at current instance in scenario
N-element vector

Robot platform joint configuration at the current instance in the scenario, specified as an *N*-element vector. *N* is the total number of joints associated with the `rigidBodyTree` object.

Data Types: `single` | `double`

Version History

Introduced in R2022a

See Also

Objects
`robotPlatform` | `robotScenario` | `robotSensor`

Functions
`attach` | `checkCollision` | `detach` | `read` | `updateMesh`

read

Read robot platform in scenario

Syntax

```
motion = read(platform,"base")
config = read(platform,"joint")
```

Description

`motion = read(platform,"base")` reads the latest motion of the robot platform base in the scenario.

`config = read(platform,"joint")` reads the latest joint configuration of the `rigidBodyTree`-based robot platform base in the scenario.

Examples

Simulate Simple Robot Scenario

Create a robot scenario.

```
scenario = robotScenario(UpdateRate=100,StopTime=1);
```

Add the ground plane and a cylinder as meshes.

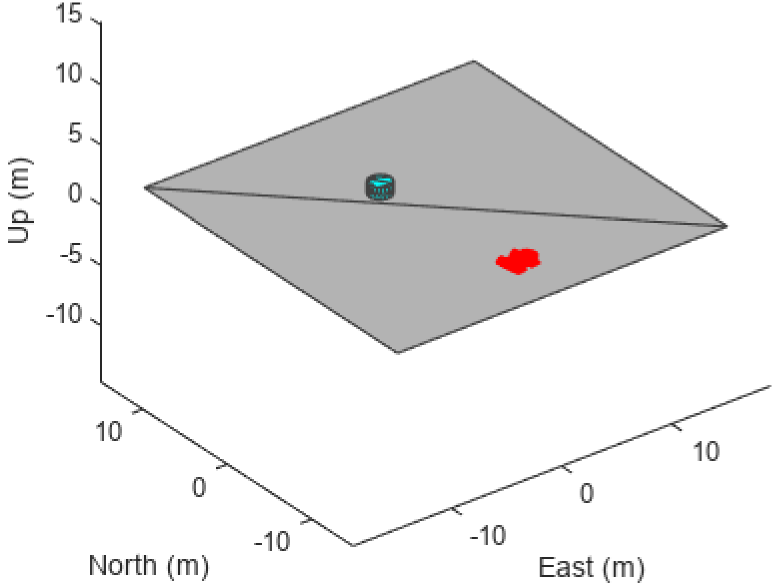
```
addMesh(scenario,"Plane",Size=[30 30],Color=[0.7 0.7 0.7])
addMesh(scenario,"Cylinder",Position=[-2 4 0.5],Color=[0 1 1])
```

Create a robot platform with a specified waypoint trajectory in the scenario. Define the mesh for the robot platform.

```
traj = waypointTrajectory("Waypoints",[0 -10 0; 10 0 0; -10 10 0; 0 -10 0], ...
                          "TimeOfArrival",[0 0.33 0.66 1], ...
                          "ReferenceFrame","ENU");
platform = robotPlatform("Robot",scenario, ...
                         BaseTrajectory=traj);
updateMesh(platform,"GroundVehicle",Scale=3);
```

Simulate and visualize the scenario.

```
setup(scenario);
idx = 1;
while advance(scenario)
    motion(idx,:) = read(platform);
    show3D(scenario);
    drawnow update
    idx = idx+1;
end
```




```
restart(scenario);
```

Input Arguments

`platform` — Robot platform in scenario
`robotPlatform` object

Robot platform in the scenario, specified as a `robotPlatform` object.

Output Arguments

`motion` — Robot platform motion at current instance in scenario
16-element vector

Robot platform motion at the current instance in the scenario, returned as a 16-element vector with these elements in order:

- `[x y z]` — Positions in *xyz*-axes in meters.
- `[vx vy vz]` — Velocities in *xyz*-directions in meters per second.
- `[ax ay az]` — Accelerations in *xyz*-directions in meters per second squared.
- `[qw qx qy qz]` — Quaternion vector for orientation.
- `[wx wy wz]` — Angular velocities in radians per second.

Data Types: `single` | `double`

`config` — Robot platform joint configuration at current instance in scenario
N-element vector

Robot platform joint configuration at the current instance in the scenario, returned as an *N*-element vector. *N* is the total number of joints associated with the `rigidBodyTree` object.

Data Types: `single` | `double`

Version History

Introduced in R2022a

See Also

Objects
`robotPlatform` | `robotScenario` | `robotSensor`

Functions
`attach` | `checkCollision` | `detach` | `move` | `updateMesh`

updateMesh

Update robot platform body mesh

Syntax

```
updateMesh(platform,type,Name=Value)
```

Description

`updateMesh(platform,type,Name=Value)` updates the body mesh of the robot platform with the specified mesh type and specifies additional options using one or more name-value arguments.

Examples

Create and Simulate Robot Scenario

Create a robot scenario.

```
scenario = robotScenario(UpdateRate=100,StopTime=1);
```

Add the ground plane and a box as meshes.

```
addMesh(scenario,"Plane",Size=[3 3],Color=[0.7 0.7 0.7]);
addMesh(scenario,"Box",Size=[0.5 0.5 0.5],Position=[0 0 0.25], ...
        Color=[0 1 0])
```

Create a waypoint trajectory for the robot platform using an ENU reference frame.

```
waypoint = [0 -1 0; 1 0 0; -1 1 0; 0 -1 0];
toa = linspace(0,1,length(waypoint));
traj = waypointTrajectory("Waypoints",waypoint, ...
                          "TimeOfArrival",toa, ...
                          "ReferenceFrame","ENU");
```

Create a `rigidBodyTree` object of the TurtleBot 3 Waffle Pi robot with `loadrobot`.

```
robotRBT = loadrobot("robotisTurtleBot3WafflePi");
```

Create a robot platform with trajectory.

```
platform = robotPlatform("TurtleBot",scenario, ...
                         BaseTrajectory=traj);
```

Set up platform mesh with the `rigidBodyTree` object.

```
updateMesh(platform,"RigidBodyTree",Object=robotRBT)
```

Create an INS sensor object and attach the sensor to the platform.

```
ins = robotSensor("INS",platform,insSensor("RollAccuracy",0), ...
                  UpdateRate=scenario.UpdateRate);
```

Visualize the scenario.

```
[ax,plotFrames] = show3D(scenario);
axis equal
hold on
```

In a loop, step through the trajectory to output the position, orientation, velocity, acceleration, and angular velocity.

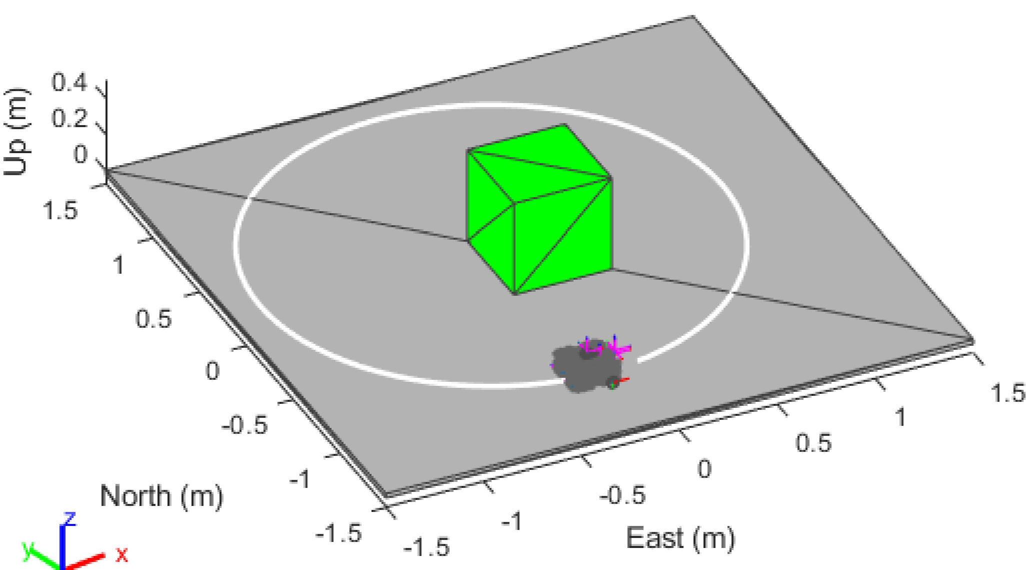
```
count = 1;
while ~isDone(traj)
    [Position(count,:),Orientation(count,:),Velocity(count,:), ...
     Acceleration(count,:),AngularVelocity(count,:)] = traj();
    count = count+1;
end
```

Create a line plot for the trajectory. First create the plot with `plot3`, then manually modify the data source properties of the plot. This improves the performance of the plotting.

```
trajPlot = plot3(nan,nan,nan,"Color",[1 1 1],"LineWidth",2);
trajPlot.XDataSource = "Position(:,1)";
trajPlot.YDataSource = "Position(:,2)";
trajPlot.ZDataSource = "Position(:,3)";
```

Set up the simulation. Then, iterate through the positions and show the scene each time the INS sensor updates. Advance the scene, move the robot platform, and update the sensors.

```
setup(scenario)
for idx = 1:count-1
    % Read sensor readings.
    [isUpdated,insTimestamp(idx,1),sensorReadings(idx)] = read(ins);
    if isUpdated
        % Use fast update to move platform visualization frames.
        show3D(scenario,FastUpdate=true,Parent=ax);
        % Refresh all plot data and visualize.
        refreshdata
        drawnow limitrate
    end
    % Advance scenario simulation time.
    advance(scenario);
    % Update all sensors in the scene.
    updateSensors(scenario)
end
hold off
```



Perform Pick and Place in Robot Scenario

Create a `robotScenario` object.

```
scenario = robotScenario(UpdateRate=1,StopTime=10);
```

Create a `rigidBodyTree` object of the Franka Emika Panda manipulator using `loadrobot`.

```
robotRBT = loadrobot("frankaEmikaPanda");
```

Create a `rigidBodyTree`-based `robotPlatform` object using the manipulator model.

```
robot = robotPlatform("Manipulator",scenario, ...
                      RigidBodyTree=robotRBT);
```

Create a non-`rigidBodyTree`-based `robotPlatform` object of a box to manipulate. Specify the mesh type and size.

```
box = robotPlatform("Box",scenario,Collision="mesh", ...
                    InitialBasePosition=[0.5 0.15 0.278]);
updateMesh(box,"Cuboid",Collision="mesh",Size=[0.06 0.06 0.1])
```

Visualize the scenario.

```
ax = show3D(scenario,Collisions="on");
view(79,36)
light
```

Specify the initial and the pick-up joint configuration of the manipulator, to move the manipulator from its initial pose to close to the box.

```
initialConfig = homeConfiguration(robot.RigidBodyTree);
pickUpConfig = [0.2371 -0.0200 0.0542 -2.2272 0.0013 ...
                2.2072 -0.9670 0.0400 0.0400];
```

Create an RRT path planner using the `manipulatorRRT` object, and specify the manipulator model.

```
planner = manipulatorRRT(robot.RigidBodyTree,scenario.CollisionMeshes);
planner.IgnoreSelfCollision = true;
```

Plan the path between the initial and the pick-up joint configurations. Then, to visualize the entire path, interpolate the path into small steps.

```
rng("default")
path = plan(planner,initialConfig,pickUpConfig);
path = interpolate(planner,path,25);
```

Set up the simulation.

```
setup(scenario)
```

Check the collision before manipulator picks up the box.

```
checkCollision(robot,"Box", ...
               IgnoreSelfCollision="on")
```

```
ans = logical
   0
```

Move the joints of the manipulator along the path and visualize the scenario.

```
helperRobotMove(path,robot,scenario,ax)
```

Check the collision after manipulator picks up the box.

```
checkCollision(robot,"Box", ...
               IgnoreSelfCollision="on")
```

```
ans = logical
   1
```

Use the `attach` function to attach the box to the gripper of the manipulator.

```
attach(robot,"Box","panda_hand", ...
       ChildToParentTransform=trvec2tform([0 0 0.1]))
```

Specify the drop-off joint configuration of the manipulator to move the manipulator from its pick-up pose to the box drop-off pose.

```
dropOffConfig = [-0.6564 0.2885 -0.3187 -1.5941 0.1103 ...
                 1.8678 -0.2344 0.04 0.04];
```

Plan the path between the pick-up and drop-off joint configurations.

```
path = plan(planner,pickUpConfig,dropOffConfig);
path = interpolate(planner,path,25);
```

Move the joints of the manipulator along the path and visualize the scenario.

```
helperRobotMove(path,robot,scenario,ax)
```

Use the `detach` function to detach the box from the manipulator gripper.

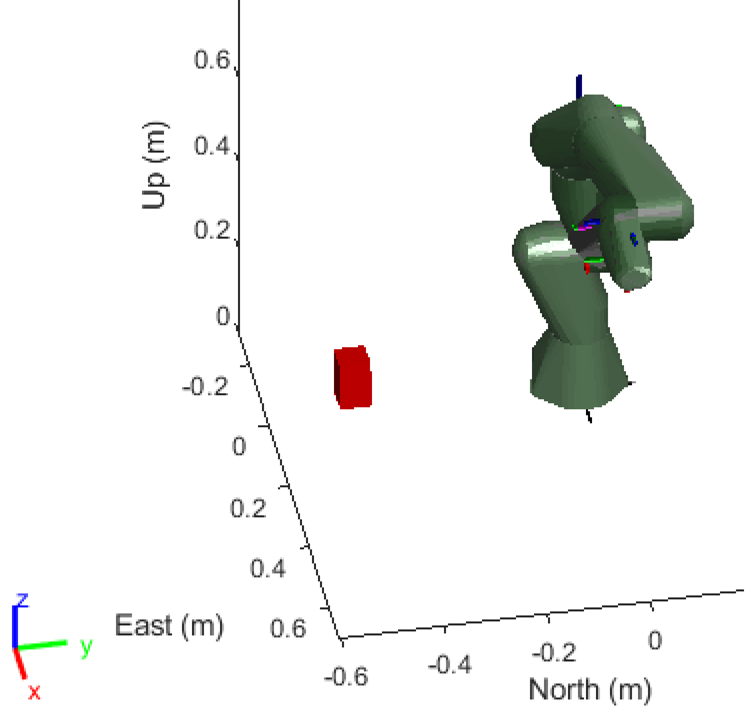
```
detach(robot)
```

Plan the path between the drop-off and initial joint configurations to move the manipulator from its box drop-off pose to its initial pose.

```
path = plan(planner,dropOffConfig,initialConfig);
path = interpolate(planner,path,25);
```

Move the joints of the manipulator along the path and visualize the scenario.

```
helperRobotMove(path,robot,scenario,ax)
```



Helper function to move the joints of the manipulator.

```
function helperRobotMove(path,robot,scenario,ax)
    for idx = 1:size(path,1)
        jointConfig = path(idx,:);
```

```
        move(robot,"joint",jointConfig)
        show3D(scenario,fastUpdate=true,Parent=ax,Collisions="on");
        drawnow
        advance(scenario);
    end
end
```

Input Arguments

`platform` — Robot platform in scenario
`robotPlatform` object

Robot platform in the scenario, specified as a `robotPlatform` object.

`type` — Type of mesh
`"Cuboid"` | `"GroundVehicle"` | `"RigidBodyTree"` | `"Custom"`

Type of mesh, specified as `"Cuboid"`, `"GroundVehicle"`, `"RigidBodyTree"`, or `"Custom"`.

Data Types: `char` | `string`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Example: `Scale=2` specifies the scale of the ground vehicle robot platform mesh as 2.

`Position` — Relative mesh position in body frame
`[0 0 0]` (default) | vector of form `[x y z]`

Relative mesh position in the body frame, specified as a vector of the form `[x y z]` in meters.

Data Types: `single` | `double`

`Orientation` — Relative mesh orientation in body frame
`[1 0 0 0]` (default) | vector of form `[w x y z]` | `quaternion` object

Relative mesh orientation in the body frame, specified as a quaternion vector of the form `[w x y z]` or a `quaternion` object.

Data Types: `single` | `double`

`Offset` — Transformation of mesh relative to body frame
4-by-4 homogeneous transformation matrix

Transformation of mesh relative to the body frame, specified as a 4-by-4 homogeneous transformation matrix. The matrix maps points in the platform mesh frame to points in the body frame.

Data Types: `single` | `double`

`Color` — Robot platform body mesh color
`[1 0 0]` (default) | RGB triplet

Robot platform body mesh color, specified as a RGB triplet, except for the rigid body mesh.

Data Types: `single` | `double`

Faces — Faces of custom robot platform mesh
N-by-3 matrix of positive integers

Faces of the custom robot platform mesh, specified as an *N*-by-3 matrix of positive integers. The three elements in each row are the indices of the three points in the vertices forming a triangle face. *N* is the number of faces.

Data Types: `single` | `double`

`Vertices` — Vertices of custom robot platform mesh
N-by-3 matrix of real scalars

Vertices of the custom robot platform mesh, specified as an *N*-by-3 matrix of real scalars. The first, second, and third element of each row represents the *x*-, *y*-, and *z*-position of each vertex, respectively. *N* is the number of vertices.

Data Types: `single` | `double`

`Size` — Size of cuboid robot platform mesh
`[1 0.5 0.3]` (default) | vector of form `[xlength ylength zlength]`

Size of the cuboid robot platform mesh, specified as a vector of the form `[xlength ylength zlength]` in meters.

Data Types: `single` | `double`

`Scale` — Scale of ground vehicle robot platform mesh
`1` (default) | scalar

Scale of the ground vehicle robot platform mesh, specified as a scalar. Scale is unitless.

Data Types: `single` | `double`

`Object` — Rigid body tree robot platform
`[]` (default) | `rigidBodyTree` object

Rigid body tree robot platform, specified as a `rigidBodyTree` object.

`IsBinaryOccupied` — Occupied state of binary occupancy map
`false` (default) | `true`

Occupied state of binary occupancy map, specified as `true` or `false`. Set the value as `true` if robot platform is incorporated in the binary occupancy map.

Data Types: `logical`

`Collision` — Collision object to add to platform mesh
`"default"` (default) | `false` | `"mesh"` | `"capsule"` | `collisionBox` object | `collisionCapsule` object | `collisionCylinder` object | `collisionMesh` object | `collisionSphere` object

Collision object to add to the platform mesh, specified as one of these values:

- `false` — Keeps the platform mesh collision-object-free. Clears any existing collision object from the `rigidBodyTree`-based platform mesh.

- `"default"` — Keeps the existing collision mesh for the `rigidBodyTree`-based platform. For other platforms, keeps the platform mesh collision-object-free.
- `"mesh"` — Fits a collision object, such as `collisionBox`, `collisionCylinder`, `collisionMesh`, or `collisionSphere`, based on the platform mesh type.
- `"capsule"` — Fits a collision capsule object to the platform mesh.
- One of these externally created collision objects:

  - `collisionBox`
  - `collisionCapsule`
  - `collisionCylinder`
  - `collisionMesh`
  - `collisionSphere`

The `rigidBodyTree`-based platform accepts an externally created collision mesh for only the base body.

Data Types: `logical` | `char` | `string`

`CollisionOffset` — Transformation of collision mesh relative to platform mesh
`eye(4)` (default) | 4-by-4 homogeneous transformation matrix

Transformation of the collision mesh relative to the platform mesh, specified as a 4-by-4 homogeneous transformation matrix. Use the `CollisionOffset` input for `rigidBodyTree`-based platforms only when specifying the `Collision` input as an externally created collision object.

Data Types: `single` | `double`

Version History

Introduced in R2022a

See Also

Objects
`robotPlatform` | `robotScenario` | `robotSensor`

Functions
`attach` | `checkCollision` | `detach` | `move` | `read`

addInertialFrame

Define new inertial frame in robot scenario

Syntax

```
addInertialFrame(scenario,base,name,position,orientation)
addInertialFrame(scenario,base,name,tform)
```

Description

`addInertialFrame(scenario,base,name,position,orientation)` adds a new inertial frame to the robot scenario by specifying the base, name, position, and orientation of the new inertial frame.

`addInertialFrame(scenario,base,name,tform)` adds a new inertial frame to the robot scenario by specifying the base, name, and transformation matrix of the new inertial frame.

Examples

Add Inertial Frame to Robot Scenario

Create a robot scenario. By default, the inertial frames are the ENU and the NED frames.

```
scenario = robotScenario
```

```
scenario = 
  robotScenario with properties:

           UpdateRate: 10
             StopTime: Inf
    HistoryBufferSize: 100
    ReferenceLocation: [0 0 0]
         MaxNumFrames: 50
          CurrentTime: 0
            IsRunning: 0
        TransformTree: [1x1 transformTree]
       InertialFrames: ["ENU"    "NED"]
               Meshes: {}
      CollisionMeshes: {}
            Platforms: [0x0 robotPlatform]
```

Add a new inertial frame named `robot` to the scenario.

```
addInertialFrame(scenario,"ENU","robot",eul2tform([pi/4 0 0]))
```

You can now use the `robot` frame as a reference frame to define other objects in the scenario.

```
scenario.InertialFrames
```

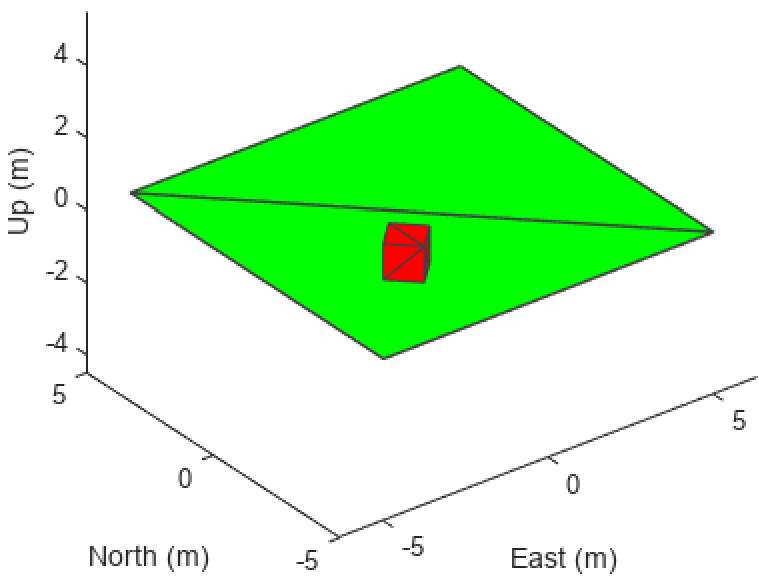
```
ans = 1x3 string
    "ENU"    "NED"    "robot"
```

Add plane and box mesh with reference frame in the scenario.

```
addMesh(scenario,"Plane",Size=[10 10],Color=[0 1 0])
addMesh(scenario,"Box",Position=[-2 -2 0.5],ReferenceFrame="robot")
```

Visualize the scenario.

```
show3D(scenario);
```



Input Arguments

`scenario` — Robot scenario
`robotScenario` object

Robot scenario, specified as a `robotScenario` object.

base — Base of new inertial frame
string scalar

Base of the new inertial frame, specified as a string scalar. The base frame must be defined in the scenario in advance.

Example: "ENU"

Data Types: char | string

name — Name of new inertial frame
string scalar

Name of the new inertial frame, specified as a string scalar.

Example: "newFrame"

Data Types: char | string

position — Position of new inertial frame
1-by-3 vector

Position of the new inertial frame with respect to the base frame (specified in the base argument), specified as a 1-by-3 vector in meters.

Data Types: single | double

orientation — Orientation of new inertial frame
quaternion object | 1-by-4 quaternion vector

Orientation of the new inertial frame with respect to the base frame (specified in the base argument), specified as a quaternion object or a 1-by-4 quaternion vector. The specified orientation is from the base frame to the new inertial frame.

Data Types: single | double

tform — Transformation matrix of new inertial frame
4-by-4 homogeneous transform matrix

Transformation matrix that maps points in the new frame (specified in the base argument) to the base frame, specified as a 4-by-4 homogeneous transform matrix that maps points in the base frame to the new inertial frame.

Example: [0 0 1 0; 0 1 0 0; -1 0 0 0; 0 0 0 1]

Data Types: single | double

Version History

Introduced in R2022a

See Also

Objects
robotPlatform | robotScenario | robotSensor

Functions
addMesh | advance | binaryOccupancyMap | restart | setup | show3D | updateSensors

addMesh

Add new static mesh to robot scenario

Syntax

`addMesh(scenario,type,Name=Value)`

Description

`addMesh(scenario,type,Name=Value)` adds a static mesh to the robot scenario by specifying the mesh type and specifies additional options using name-value arguments.

Examples

Add Meshes to Robot Scenario

Create a robot Scenario.

```
scenario = robotScenario;
```

Add a plane, box, cylinder, and sphere mesh to the scenario.

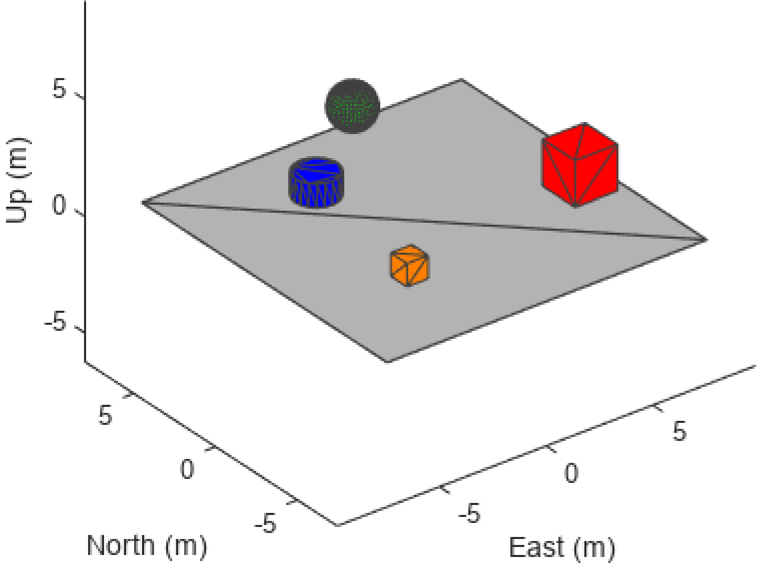
```
addMesh(scenario,"Plane",Size=[15 15],Color=[0.7 0.7 0.7])
addMesh(scenario,"Box",Position=[-3 -3 0.5],Color=[1 0.5 0])
addMesh(scenario,"Cylinder",Position=[-2 4 0.5],Color=[0 0 1])
addMesh(scenario,"Sphere",Position=[2 7 1],Color=[0 1 0])
```

Add custom mesh to the scenario with vertices and faces.

```
vertices = [0 0 0; 0 0 2; 0 2 0; 0 2 2; 2 0 0; 2 0 2; 2 2 0; 2 2 2];
faces = [1 3 7; 1 7 5; 1 6 2; 1 5 6; 1 2 4; 1 4 3; ...
         3 4 8; 3 8 7; 5 8 6; 5 7 8; 2 8 4; 2 6 8];
addMesh(scenario,"Custom",Vertices=vertices,Faces=faces,Position=[4 -4 1])
```

Visualize the scenario.

```
show3D(scenario);
```



Input Arguments

`scenario` — Robot scenario
`robotScenario` object

Robot scenario, specified as a `robotScenario` object.

`type` — Type of mesh
`"Box"` | `"Cylinder"` | `"Plane"` | `"Sphere"` | `"Custom"`

Type of mesh, specified as `"Box"`, `"Cylinder"`, `"Plane"`, `"Sphere"`, or `"Custom"`.

Data Types: `char` | `string`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Example: `addMesh(scenario,"Box",Position=[-3 -3 0.5])`

`Color` — Mesh color
`[1 0 0]` (default) | RGB triplet

Mesh color, specified as a RGB triplet.

Data Types: `single` | `double`

`ReferenceFrame` — Reference frame of mesh geometry
`"ENU"` (default) | name of defined inertial frame

Reference frame of the geometry input, specified as an inertial frame name defined in the InertialFrames property of the `robotScenario` object. You can add new inertial frames to the scenario using the `addInertialFrame` object function.

The scenario only accepts frames that have *z*-axis rotation with respect to the `"ENU"` frame.

Data Types: `char` | `string`

`IsBinaryOccupied` — Occupied state of binary occupancy map
`false` (default) | `true`

Occupied state of binary occupancy map, specified as `true` or `false`. Set the value as `true` if static mesh is considered as an obstacle in the scenario and it is incorporated in the binary occupancy map.

Data Types: `logical`

`Position` — Position of static mesh in robot scenario
`[0 0 0]` (default) | vector of form `[x y z]`

Position of static mesh in robot scenario, specified as a vector of the form `[x y z]` in meters.

Data Types: `single` | `double`

`Size` — Size of static mesh
scalar | vector

Size of static mesh, specified as a scalar or vector of geometry parameters, except for the custom mesh. Depending on the `type` input, the geometry parameters have different forms:

type Input Argument	Geometry Parameters	Description
`"Box"`	`[xlength ylength zlength]`	Create a box with a specified side lengths in the *x*-, *y*-, and *z*-directions. Default: `[1 1 1]`
`"Cylinder"`	`[radius length]`	Create a cylinder with a specified radius and length. Default: `[1 1]`
`"Plane"`	`[xlength ylength]`	Create a plane with a specified side lengths in the *x*- and *y*-directions. Default: `[1 1]`
`"Sphere"`	*radius*	Create a sphere with a specified radius. Default: `1`

Data Types: `single` | `double`

Faces — Faces of custom static mesh
N-by-3 matrix of positive integers

Faces of the custom static mesh, specified as an *N*-by-3 matrix of positive integers. The three elements in each row are the indices of the three points in the vertices forming a triangle face. *N* is the number of faces.

Data Types: `single` | `double`

Vertices — Vertices of custom static mesh
N-by-3 matrix of real scalars

Vertices of the custom static mesh, specified as an *N*-by-3 matrix of real scalars. The first, second, and third element of each row represents the *x*-, *y*-, and *z*-position of each vertex, respectively. *N* is the number of vertices.

Data Types: `single` | `double`

Collision — Collision object to add to scenario mesh
`false` (default) | `"mesh"` | `"capsule"` | `collisionBox` object | `collisionCapsule` object | `collisionCylinder` object | `collisionMesh` object | `collisionSphere` object

Collision object to add to the scenario mesh, specified as one of these values:

- `false` — Keeps the scenario mesh collision-object-free.
- `"mesh"` — Fits a collision object, such as `collisionBox`, `collisionCylinder`, `collisionMesh`, or `collisionSphere`, based on the scenario mesh type.
- `"capsule"` — Fits a collision capsule object to the scenario mesh.
- One of these externally created collision objects:
  - `collisionBox`
  - `collisionCapsule`
  - `collisionCylinder`
  - `collisionMesh`
  - `collisionSphere`

CollisionOffset — Transformation of collision mesh relative to scenario mesh
`eye(4)` (default) | 4-by-4 homogeneous transformation matrix

Transformation of the collision mesh relative to the scenario mesh, specified as a 4-by-4 homogeneous transformation matrix.

Data Types: `single` | `double`

Name — Identifier for scenario mesh
string scalar | character vector

Identifier for the scenario mesh, specified as a string scalar or character vector. The name must be unique within the scenario. If you do not specify this argument, the function assigns a unique name for the scenario mesh is assigned, by default.

Data Types: `char` | `string`

Version History

Introduced in R2022a

See Also

Objects

`robotPlatform` | `robotScenario` | `robotSensor`

Functions

`addInertialFrame` | `advance` | `binaryOccupancyMap` | `restart` | `setup` | `show3D` | `updateSensors`

advance

Advance robot scenario simulation by one time step

Syntax

`isrunning = advance(scenario)`

Description

`isrunning = advance(scenario)` advances the robot scenario simulation by one time step. The `UpdateRate` property of the `robotScenario` object determines the time step during simulation. The function returns the running status of the simulation. The function only updates a platform location if the platform has an assigned trajectory.

Examples

Simulate Simple Robot Scenario

Create a robot scenario.

```
scenario = robotScenario(UpdateRate=100,StopTime=1);
```

Add the ground plane and a cylinder as meshes.

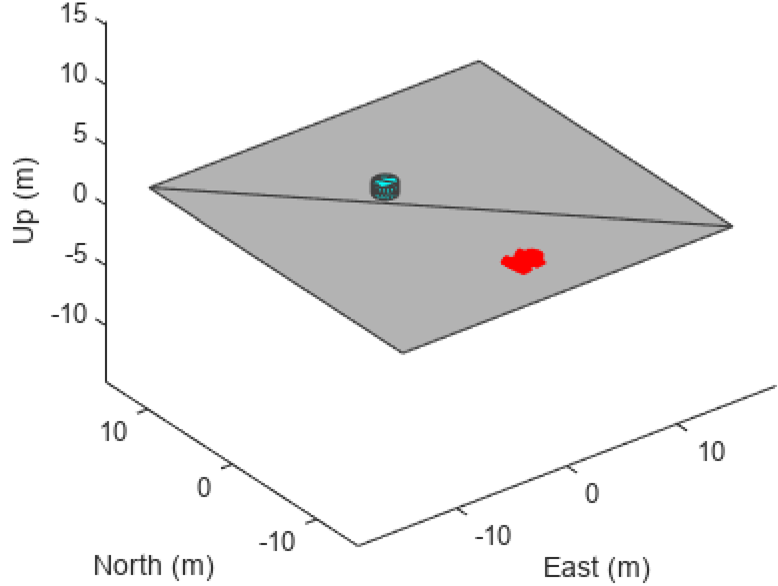
```
addMesh(scenario,"Plane",Size=[30 30],Color=[0.7 0.7 0.7])
addMesh(scenario,"Cylinder",Position=[-2 4 0.5],Color=[0 1 1])
```

Create a robot platform with a specified waypoint trajectory in the scenario. Define the mesh for the robot platform.

```
traj = waypointTrajectory("Waypoints",[0 -10 0; 10 0 0; -10 10 0; 0 -10 0], ...
                          "TimeOfArrival",[0 0.33 0.66 1], ...
                          "ReferenceFrame","ENU");
platform = robotPlatform("Robot",scenario, ...
                         BaseTrajectory=traj);
updateMesh(platform,"GroundVehicle",Scale=3);
```

Simulate and visualize the scenario.

```
setup(scenario);
idx = 1;
while advance(scenario)
    motion(idx,:) = read(platform);
    show3D(scenario);
    drawnow update
    idx = idx+1;
end
```




```
restart(scenario);
```

Input Arguments

scenario — Robot scenario
robotScenario object

Robot scenario, specified as a `robotScenario` object.

Output Arguments

isrunning — Running state of simulation
true | false

Running state of the simulation, returned as `true` or `false`. If `isrunning` is returned as `true`, then the simulation is running. If `isrunning` is returned as `false`, the simulation has stopped. A simulation stops when the stop time is reached.

Version History

Introduced in R2022a

See Also

Objects
robotPlatform | robotScenario | robotSensor

Functions
addInertialFrame | addMesh | binaryOccupancyMap | restart | setup | show3D | updateSensors

binaryOccupancyMap

Create 2-D binary occupancy map from robot scenario

Syntax

```
map = binaryOccupancyMap(scenario,Name=Value)
```

Description

`map = binaryOccupancyMap(scenario,Name=Value)` creates binary occupancy map based on mesh elements from scenario defined with `IsBinaryOccupied` status `true`. The mesh elements are processed in the 3-D convex hull form. Further, mesh element is considered as occupied region only if it lies inside map height limits and map size. These properties are specified by one or more name-value pair arguments.

Examples

Create 2-D Binary Occupancy Map from Robot Scenario

Create a robot scenario.

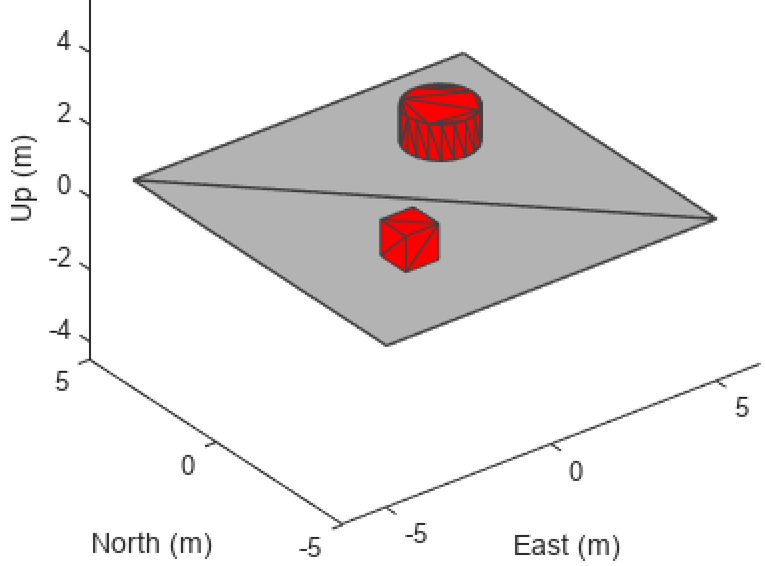
```
scenario = robotScenario(UpdateRate=1,StopTime=10);
```

Add a plane, box and cylinder mesh in the scenario.

```
addMesh(scenario,"Plane",Size=[10 10],Color=[0.7 0.7 0.7])
addMesh(scenario,"Box",Position=[-2 -2 0.5],IsBinaryOccupied=true)
addMesh(scenario,"Cylinder",Position=[2 2 0.5],IsBinaryOccupied=true)
```

Visualize the scenario.

```
show3D(scenario);
```

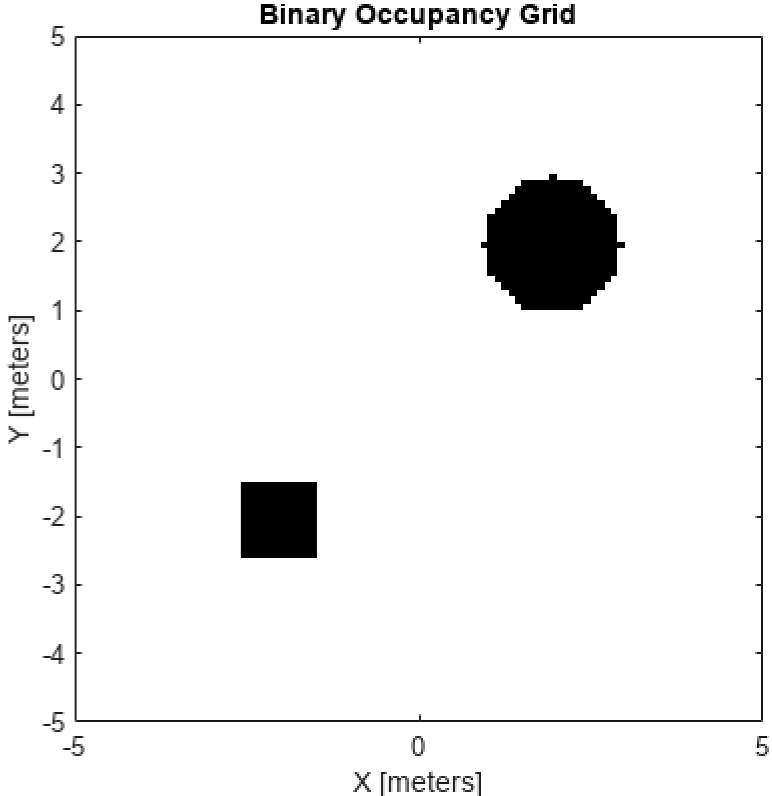


Get the 2-D occupancy map.

```
occupancyMap = binaryOccupancyMap(scenario,MapHeightLimits=[-1 1], ...
                                  GridOriginInLocal=[-5 -5]);
```

Visualize the 2-D occupancy map.

```
figure
show(occupancyMap);
```



Input Arguments

`scenario` — Robot scenario
`robotScenario` object

Robot scenario, specified as a `robotScenario` object.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `map = binaryOccupancyMap(scenario,MapResolution=15)`

`GridOriginInLocal` — Origin of occupancy map grid in local coordinates
`[0 0]` (default) | vector of the form `[xLocal yLocal]`

Origin of occupancy map grid in local coordinates, specified as a two-element vector of the form `[xLocal yLocal]` in meters.

Data Types: `single` | `double`

`HeightResolution` — Resolution of occupancy map grid in *z*-axis
`10` (default) | scalar integer

Resolution of occupancy map grid in *z*-axis, specified as a scalar integer in cells per meter.

Data Types: `single` | `double`

`MapHeightLimits` — Minimum and maximum values of map height
`[0 1]` (default) | vector of the form `[`*`Hmin Hmax`*`]`

Minimum and maximum values of map height, specified as a two-element vector of the form `[`*`Hmin Hmax`*`]` in meters, from which static meshes are considered for occupancy map grid.

Data Types: `single` | `double`

`MapResolution` — Resolution of occupancy map grid in *xy*-axis
`10` (default) | scalar integer

Resolution of occupancy map grid in *xy*-axis, specified as a scalar integer in cells per meter.

Data Types: `single` | `double`

`MapSize` — Size of occupancy map grid
`[10 10]` (default) | vector of the form `[`*`width height`*`]`

Size of occupancy map grid, specified as a two-element vector of the form `[`*`width height`*`]` in meters.

Data Types: `single` | `double`

Output Arguments

`map` — Binary occupancy map from robot scenario
`binaryOccupancyMap` object

Binary occupancy map from robot scenario, returned as a `binaryOccupancyMap` object.

Version History

Introduced in R2022a

See Also

Objects
`robotPlatform` | `robotScenario` | `robotSensor` | `binaryOccupancyMap`

Functions
`addInertialFrame` | `addMesh` | `advance` | `restart` | `setup` | `show3D` | `updateSensors`

restart

Reset simulation of robot scenario

Syntax

`restart(scenario)`

Description

`restart(scenario)` resets the simulation of the robot scenario scene. The function resets poses of the platforms and sensor readings to `NaN`, resets the `CurrentTime` property of the scenario to `0`, and resets the `IsRunning` property of the scenario to `false`.

Examples

Simulate Simple Robot Scenario

Create a robot scenario.

```
scenario = robotScenario(UpdateRate=100,StopTime=1);
```

Add the ground plane and a cylinder as meshes.

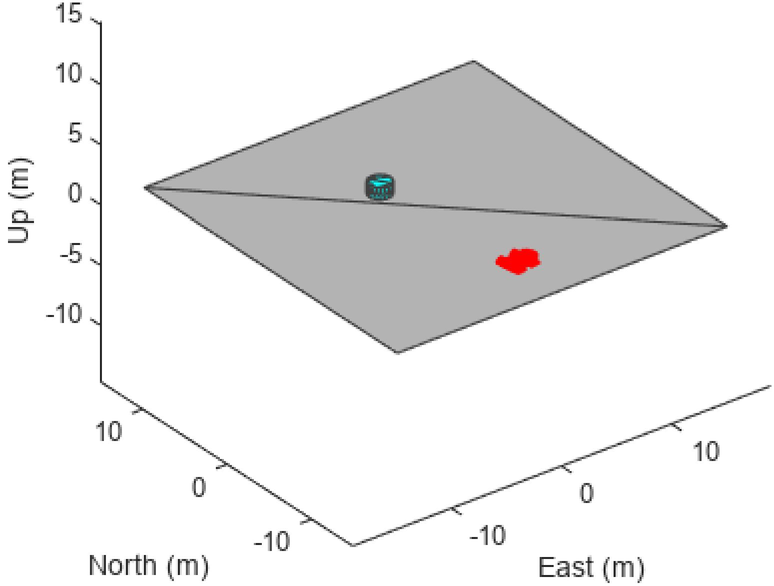
```
addMesh(scenario,"Plane",Size=[30 30],Color=[0.7 0.7 0.7])
addMesh(scenario,"Cylinder",Position=[-2 4 0.5],Color=[0 1 1])
```

Create a robot platform with a specified waypoint trajectory in the scenario. Define the mesh for the robot platform.

```
traj = waypointTrajectory("Waypoints",[0 -10 0; 10 0 0; -10 10 0; 0 -10 0], ...
                          "TimeOfArrival",[0 0.33 0.66 1], ...
                          "ReferenceFrame","ENU");
platform = robotPlatform("Robot",scenario, ...
                         BaseTrajectory=traj);
updateMesh(platform,"GroundVehicle",Scale=3);
```

Simulate and visualize the scenario.

```
setup(scenario);
idx = 1;
while advance(scenario)
    motion(idx,:) = read(platform);
    show3D(scenario);
    drawnow update
    idx = idx+1;
end
```




```
restart(scenario);
```

Input Arguments

scenario — Robot scenario
robotScenario object

Robot scenario, specified as a robotScenario object.

Version History

Introduced in R2022a

See Also

Objects
robotPlatform | robotScenario | robotSensor

Functions
addInertialFrame | addMesh | advance | binaryOccupancyMap | setup | show3D | updateSensors

setup

Prepare robot scenario for simulation

Syntax

```
setup(scenario)
```

Description

`setup(scenario)` prepares the robot scenario for simulation, sets poses of the platforms to their initial values, and generates initial sensor readings.

Examples

Simulate Simple Robot Scenario

Create a robot scenario.

```
scenario = robotScenario(UpdateRate=100,StopTime=1);
```

Add the ground plane and a cylinder as meshes.

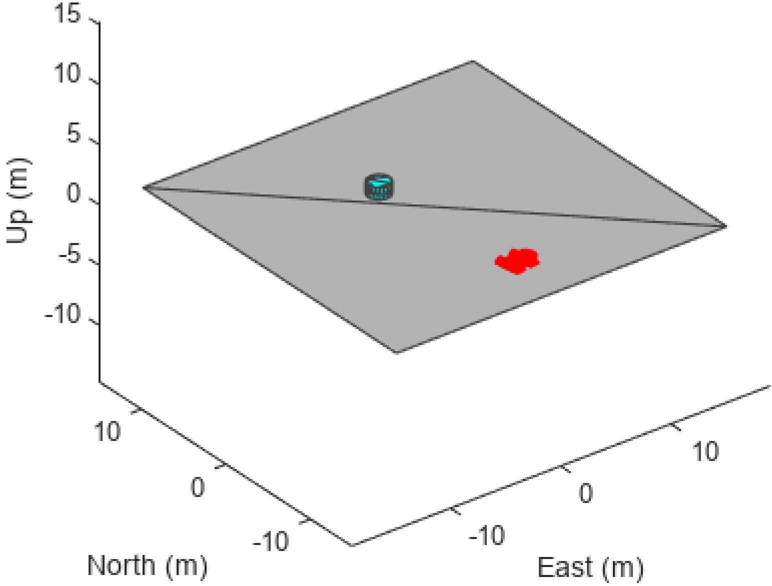
```
addMesh(scenario,"Plane",Size=[30 30],Color=[0.7 0.7 0.7])
addMesh(scenario,"Cylinder",Position=[-2 4 0.5],Color=[0 1 1])
```

Create a robot platform with a specified waypoint trajectory in the scenario. Define the mesh for the robot platform.

```
traj = waypointTrajectory("Waypoints",[0 -10 0; 10 0 0; -10 10 0; 0 -10 0], ...
                          "TimeOfArrival",[0 0.33 0.66 1], ...
                          "ReferenceFrame","ENU");
platform = robotPlatform("Robot",scenario, ...
                         BaseTrajectory=traj);
updateMesh(platform,"GroundVehicle",Scale=3);
```

Simulate and visualize the scenario.

```
setup(scenario);
idx = 1;
while advance(scenario)
    motion(idx,:) = read(platform);
    show3D(scenario);
    drawnow update
    idx = idx+1;
end
```




```
restart(scenario);
```

Input Arguments

scenario — Robot scenario
robotScenario object

Robot scenario, specified as a robotScenario object.

Version History
Introduced in R2022a

See Also

Objects
robotPlatform | robotScenario | robotSensor

Functions
addInertialFrame | addMesh | advance | binaryOccupancyMap | restart | show3D | updateSensors

show3D

Visualize robot scenario in 3-D

Syntax

```
[ax,plottedFrames] = show3D(scenario)
[ax,plottedFrames] = show3D(scenario,time)
[ax,plottedFrames] = show3D( ___ ,Name=Value)
```

Description

`[ax,plottedFrames] = show3D(scenario)` visualizes latest states of the platforms and sensors in the robot scenario scene along with all static meshes. The function also returns the axes on which the scene is plotted and the frames on which each object is plotted.

`[ax,plottedFrames] = show3D(scenario,time)` visualizes the robot scenario at the specified `time`.

`[ax,plottedFrames] = show3D( ___ ,Name=Value)` specifies additional options using name-value arguments.

Examples

Create and Simulate Robot Scenario

Create a robot scenario.

```
scenario = robotScenario(UpdateRate=100,StopTime=1);
```

Add the ground plane and a box as meshes.

```
addMesh(scenario,"Plane",Size=[3 3],Color=[0.7 0.7 0.7]);
addMesh(scenario,"Box",Size=[0.5 0.5 0.5],Position=[0 0 0.25], ...
        Color=[0 1 0])
```

Create a waypoint trajectory for the robot platform using an ENU reference frame.

```
waypoint = [0 -1 0; 1 0 0; -1 1 0; 0 -1 0];
toa = linspace(0,1,length(waypoint));
traj = waypointTrajectory("Waypoints",waypoint, ...
                          "TimeOfArrival",toa, ...
                          "ReferenceFrame","ENU");
```

Create a `rigidBodyTree` object of the TurtleBot 3 Waffle Pi robot with `loadrobot`.

```
robotRBT = loadrobot("robotisTurtleBot3WafflePi");
```

Create a robot platform with trajectory.

```
platform = robotPlatform("TurtleBot",scenario, ...
                         BaseTrajectory=traj);
```

Set up platform mesh with the `rigidBodyTree` object.

```
updateMesh(platform,"RigidBodyTree",Object=robotRBT)
```

Create an INS sensor object and attach the sensor to the platform.

```
ins = robotSensor("INS",platform,insSensor("RollAccuracy",0), ...
                  UpdateRate=scenario.UpdateRate);
```

Visualize the scenario.

```
[ax,plotFrames] = show3D(scenario);
axis equal
hold on
```

In a loop, step through the trajectory to output the position, orientation, velocity, acceleration, and angular velocity.

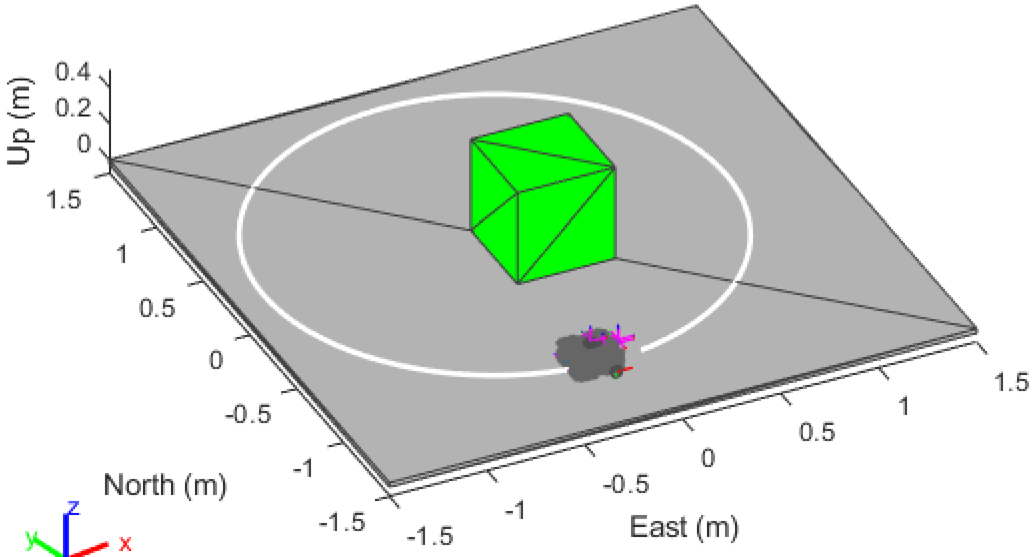
```
count = 1;
while ~isDone(traj)
    [Position(count,:),Orientation(count,:),Velocity(count,:), ...
     Acceleration(count,:),AngularVelocity(count,:)] = traj();
    count = count+1;
end
```

Create a line plot for the trajectory. First create the plot with `plot3`, then manually modify the data source properties of the plot. This improves the performance of the plotting.

```
trajPlot = plot3(nan,nan,nan,"Color",[1 1 1],"LineWidth",2);
trajPlot.XDataSource = "Position(:,1)";
trajPlot.YDataSource = "Position(:,2)";
trajPlot.ZDataSource = "Position(:,3)";
```

Set up the simulation. Then, iterate through the positions and show the scene each time the INS sensor updates. Advance the scene, move the robot platform, and update the sensors.

```
setup(scenario)
for idx = 1:count-1
    % Read sensor readings.
    [isUpdated,insTimestamp(idx,1),sensorReadings(idx)] = read(ins);
    if isUpdated
        % Use fast update to move platform visualization frames.
        show3D(scenario,FastUpdate=true,Parent=ax);
        % Refresh all plot data and visualize.
        refreshdata
        drawnow limitrate
    end
    % Advance scenario simulation time.
    advance(scenario);
    % Update all sensors in the scene.
    updateSensors(scenario)
end
hold off
```



Input Arguments

scenario — Robot scenario
robotScenario object

Robot scenario, specified as a robotScenario object.

time — Time stamp
nonnegative scalar

Time stamp at which to show the scenario, specified as a nonnegative scalar. The time stamp must already be saved in the scenario. To change the number of saved time stamps, use the HistoryBufferSize property of the robotScenario object, scenario.

Data Types: single | double

Name-Value Pair Arguments

Specify optional pairs of arguments as Name1=Value1,...,NameN=ValueN, where Name is the argument name and Value is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Example: show3D(scenario,Parent=ax)

Parent — Parent axes for plotting
axes object | uiaxes object

Parent axes for plotting, specified as an `axes` object or a `uiaxes` object.

`FastUpdate` — Enable updating from previous map
`false` (default) | `true`

Enable updating from previous map, specified as `true` or `false`. When specified as `true`, the function plots the map via a lightweight update to the previous map in the figure. When specified as `false`, the function plots the whole scene on the figure every time.

Note Parent axes must be specified to perform fast update.

Example: `FastUpdate=true`

Data Types: `logical`

`View` — View point of plot
`"3D"` (default) | `"Top"` | `"Side"`

View point of plot, specified as `"3D"`, `"Top"`, or `"Side"`.

Example: `View="Side"`

Data Types: `string` | `char`

`Visuals` — Display body visual meshes
`"on"` (default) | `"off"`

Display body visual meshes, specified as `"on"` or `"off"`.

Example: `Visuals="off"`

Data Types: `char` | `string`

`Collisions` — Display body collision geometries
`"off"` (default) | `"on"`

Display body collision geometries, specified as `"on"` or `"off"`.

Example: `Collisions="on"`

Data Types: `char` | `string`

`CollisionMeshAlpha` — Collision mesh transparency
`0.3` (default) | numeric scalar in the range `[0,1]`

Collision mesh transparency, specified as a numeric scalar in the range `[0,1]`. A value of `0` means transparent. A value of `1` means opaque. Values between `0` and `1` are partially transparent.

Example: `CollisionMeshAlpha=0.5`

Data Types: `single` | `double`

Output Arguments

ax — Axes on which scenario is plotted
`axes` object | `uiaxes` object

Axes on which the scenario is plotted, returned as an `axes` object or a `uiaxes` object.

`plottedFrames` — Plotted frame information
structure

Plotted frame information, returned as a structure of `hgtransform` objects.

Version History

Introduced in R2022a

See Also

Objects
`robotPlatform` | `robotScenario` | `robotSensor`

Functions
`addInertialFrame` | `addMesh` | `advance` | `binaryOccupancyMap` | `restart` | `setup` | `updateSensors`

updateSensors

Update sensor readings in robot scenario

Syntax

```
updateSensors(scenario)
```

Description

`updateSensors(scenario)` updates all sensor readings based on latest states of all platforms in the robot scenario, `scenario`.

Examples

Create and Simulate Robot Scenario

Create a robot scenario.

```
scenario = robotScenario(UpdateRate=100,StopTime=1);
```

Add the ground plane and a box as meshes.

```
addMesh(scenario,"Plane",Size=[3 3],Color=[0.7 0.7 0.7]);
addMesh(scenario,"Box",Size=[0.5 0.5 0.5],Position=[0 0 0.25], ...
        Color=[0 1 0])
```

Create a waypoint trajectory for the robot platform using an ENU reference frame.

```
waypoint = [0 -1 0; 1 0 0; -1 1 0; 0 -1 0];
toa = linspace(0,1,length(waypoint));
traj = waypointTrajectory("Waypoints",waypoint, ...
                          "TimeOfArrival",toa, ...
                          "ReferenceFrame","ENU");
```

Create a `rigidBodyTree` object of the TurtleBot 3 Waffle Pi robot with `loadrobot`.

```
robotRBT = loadrobot("robotisTurtleBot3WafflePi");
```

Create a robot platform with trajectory.

```
platform = robotPlatform("TurtleBot",scenario, ...
                         BaseTrajectory=traj);
```

Set up platform mesh with the `rigidBodyTree` object.

```
updateMesh(platform,"RigidBodyTree",Object=robotRBT)
```

Create an INS sensor object and attach the sensor to the platform.

```
ins = robotSensor("INS",platform,insSensor("RollAccuracy",0), ...
                  UpdateRate=scenario.UpdateRate);
```

Visualize the scenario.

```
[ax,plotFrames] = show3D(scenario);
axis equal
hold on
```

In a loop, step through the trajectory to output the position, orientation, velocity, acceleration, and angular velocity.

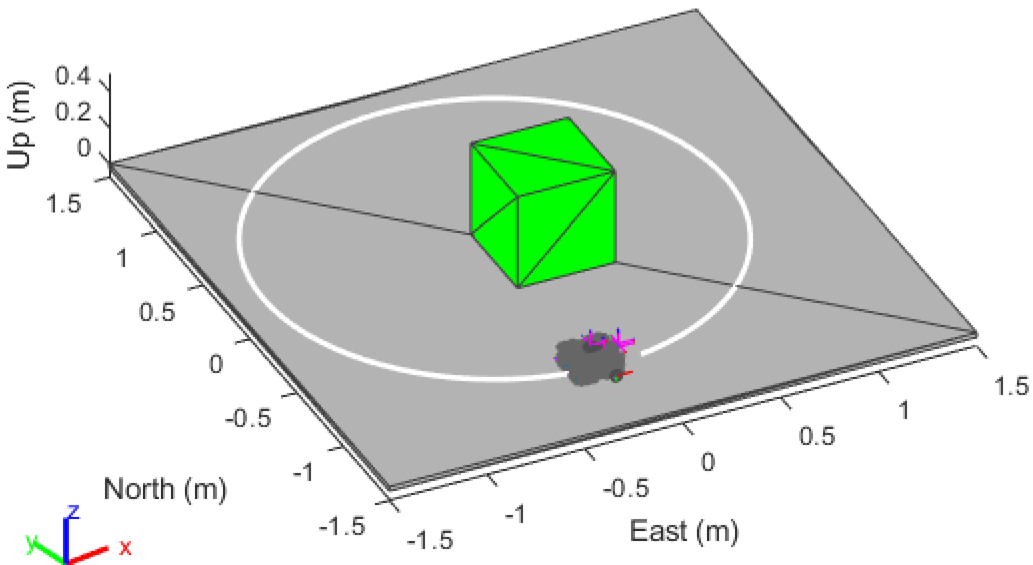
```
count = 1;
while ~isDone(traj)
    [Position(count,:),Orientation(count,:),Velocity(count,:), ...
     Acceleration(count,:),AngularVelocity(count,:)] = traj();
    count = count+1;
end
```

Create a line plot for the trajectory. First create the plot with `plot3`, then manually modify the data source properties of the plot. This improves the performance of the plotting.

```
trajPlot = plot3(nan,nan,nan,"Color",[1 1 1],"LineWidth",2);
trajPlot.XDataSource = "Position(:,1)";
trajPlot.YDataSource = "Position(:,2)";
trajPlot.ZDataSource = "Position(:,3)";
```

Set up the simulation. Then, iterate through the positions and show the scene each time the INS sensor updates. Advance the scene, move the robot platform, and update the sensors.

```
setup(scenario)
for idx = 1:count-1
    % Read sensor readings.
    [isUpdated,insTimestamp(idx,1),sensorReadings(idx)] = read(ins);
    if isUpdated
        % Use fast update to move platform visualization frames.
        show3D(scenario,FastUpdate=true,Parent=ax);
        % Refresh all plot data and visualize.
        refreshdata
        drawnow limitrate
    end
    % Advance scenario simulation time.
    advance(scenario);
    % Update all sensors in the scene.
    updateSensors(scenario)
end
hold off
```



Input Arguments

scenario — Robot scenario
robotScenario object

Robot scenario, specified as a robotScenario object.

Version History

Introduced in R2022a

See Also

Objects
robotPlatform | robotScenario | robotSensor

Functions
addInertialFrame | addMesh | advance | binaryOccupancyMap | restart | setup | show3D

read

Gather latest reading from robot sensor

Syntax

```
[isUpdated,t,sensorReadings] = read(sensor)
```

Description

`[isUpdated,t,sensorReadings] = read(sensor)` gathers the simulated sensor output sensor readings from the latest update of the robot platform associated with the specified sensor `sensor`. The function returns an indicator `isUpdated` of whether the reading was updated at the simulation step in the scenario with timestamp `t`.

Examples

Create and Simulate Robot Scenario

Create a robot scenario.

```
scenario = robotScenario(UpdateRate=100,StopTime=1);
```

Add the ground plane and a box as meshes.

```
addMesh(scenario,"Plane",Size=[3 3],Color=[0.7 0.7 0.7]);
addMesh(scenario,"Box",Size=[0.5 0.5 0.5],Position=[0 0 0.25], ...
        Color=[0 1 0])
```

Create a waypoint trajectory for the robot platform using an ENU reference frame.

```
waypoint = [0 -1 0; 1 0 0; -1 1 0; 0 -1 0];
toa = linspace(0,1,length(waypoint));
traj = waypointTrajectory("Waypoints",waypoint, ...
                          "TimeOfArrival",toa, ...
                          "ReferenceFrame","ENU");
```

Create a `rigidBodyTree` object of the TurtleBot 3 Waffle Pi robot with `loadrobot`.

```
robotRBT = loadrobot("robotisTurtleBot3WafflePi");
```

Create a robot platform with trajectory.

```
platform = robotPlatform("TurtleBot",scenario, ...
                         BaseTrajectory=traj);
```

Set up platform mesh with the `rigidBodyTree` object.

```
updateMesh(platform,"RigidBodyTree",Object=robotRBT)
```

Create an INS sensor object and attach the sensor to the platform.

```
ins = robotSensor("INS",platform,insSensor("RollAccuracy",0), ...
                  UpdateRate=scenario.UpdateRate);
```

Visualize the scenario.

```
[ax,plotFrames] = show3D(scenario);
axis equal
hold on
```

In a loop, step through the trajectory to output the position, orientation, velocity, acceleration, and angular velocity.

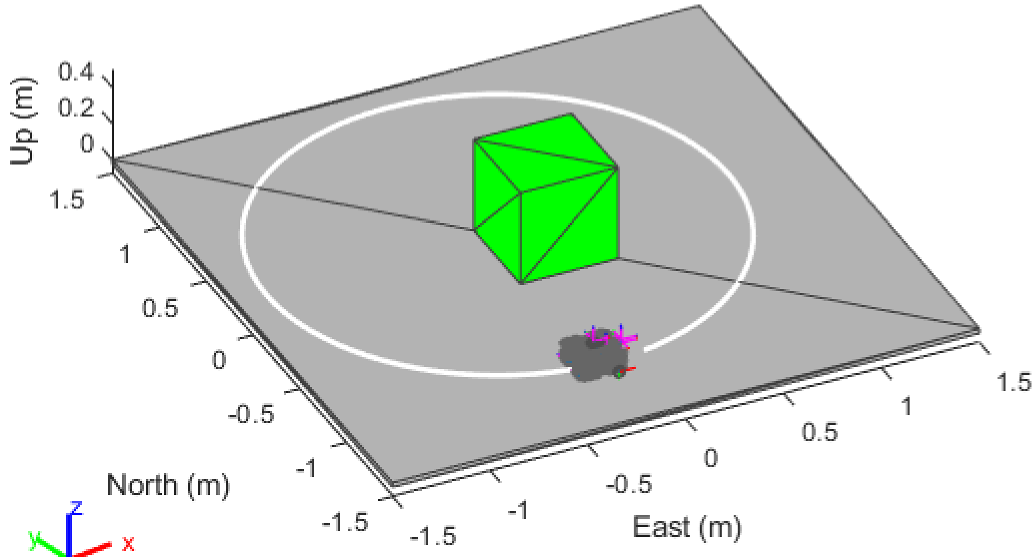
```
count = 1;
while ~isDone(traj)
    [Position(count,:),Orientation(count,:),Velocity(count,:), ...
     Acceleration(count,:),AngularVelocity(count,:)] = traj();
    count = count+1;
end
```

Create a line plot for the trajectory. First create the plot with `plot3`, then manually modify the data source properties of the plot. This improves the performance of the plotting.

```
trajPlot = plot3(nan,nan,nan,"Color",[1 1 1],"LineWidth",2);
trajPlot.XDataSource = "Position(:,1)";
trajPlot.YDataSource = "Position(:,2)";
trajPlot.ZDataSource = "Position(:,3)";
```

Set up the simulation. Then, iterate through the positions and show the scene each time the INS sensor updates. Advance the scene, move the robot platform, and update the sensors.

```
setup(scenario)
for idx = 1:count-1
    % Read sensor readings.
    [isUpdated,insTimestamp(idx,1),sensorReadings(idx)] = read(ins);
    if isUpdated
        % Use fast update to move platform visualization frames.
        show3D(scenario,FastUpdate=true,Parent=ax);
        % Refresh all plot data and visualize.
        refreshdata
        drawnow limitrate
    end
    % Advance scenario simulation time.
    advance(scenario);
    % Update all sensors in the scene.
    updateSensors(scenario)
end
hold off
```



Input Arguments

`sensor` — Robot sensor added to platform in scenario
`robotSensor` object

Robot sensor added to platform in scenario, specified as a `robotSensor` object.

Output Arguments

`isUpdated` — Sensor reading update indicator
`0` or `false` | `1` or `true`

Sensor reading update indicator, returned as a logical `0` (`false`) or `1` (`true`). If the sensor reading updated at the current simulation step, the function returns this argument as `true`.

Data Types: `logical`

`t` — Timestamp of generated sensor reading
scalar in seconds

Timestamp of the generated sensor reading, returned as a scalar in seconds.

Data Types: `double`

sensorReadings — Simulated sensor readings
gpsSensor output | insSensor output | robotLidarPointCloudGenerator output

Simulated sensor readings, which depends on the type of sensor specified in the sensor input argument. See the **Usage** syntax for the appropriate gpsSensor, insSensor, or robotLidarPointCloudGenerator System object.

Version History

Introduced in R2022a

See Also

Objects
robotPlatform | robotScenario | robotSensor | robotLidarPointCloudGenerator | gpsSensor | insSensor | robotics.SensorAdaptor

getEmptyOutputs

Class: robotics.SensorAdaptor
Package: robotics

Return empty sensor outputs without sensor inputs

Syntax

```
out = getEmptyOutputs(sensorObj)
```

Description

`out = getEmptyOutputs(sensorObj)` gets empty outputs when the sensor is not initialized using `setup` function.

Input Arguments

sensorObj — Robot sensor model
object of subclass of robotics.SensorAdaptor

Robot sensor model, specified as an object of a subclass of `robotics.SensorAdaptor`.

Output Arguments

out — Empty sensor outputs
cell array

Empty sensor outputs, returned as a cell array of variables that matches the `varargout` output of the `read` function.

Examples

Simulate Ultrasonic Sensors Mounted on Mobile Robots

This example focuses on creating and mounting an ultrasonic sensor on a mobile robot in a `robotScenario`. The `ultrasonicDetectionGenerator` from the Automated Driving Toolbox cannot be used directly with `robotScenario`. We will be implementing a custom sensor adaptor for the `ultrasonicDetectionGenerator` that makes it compatible with `robotScenario`. The sensor will be used to position a mobile robot correctly at a charging station.

Create Custom Sensor Adaptor

Use the `createCustomRobotSensorTemplate` function to generate a template sensor and update it to adapt an `ultrasonicDetectionGenerator` object for usage in Robot scenario.

```
createCustomRobotSensorTemplate
```

This example provides the adaptor class `CustomUltrasonicSensor`, which can be viewed using the following command.

```
edit CustomUltrasonicSensor.m
```

Use the Sensor Adaptor in Robot Scenario Simulation

Create a `robotScenario` object with a sample rate of 10.

```
sampleRate = 10;
scenario = robotScenario(UpdateRate=sampleRate);
```

Add a plane mesh to show the warehouse floor.

```
addMesh(scenario,"Plane",Position=[5 0 0],Size=[20 12],Color=[0.7 0.7 0.7]);
```

Create a `waypointTrajectory` that traverses a set of waypoints to the charging station and use the `lookupPose` method of the `waypointTrajectory` object to fetch the pose of the robot along the trajectory.

```
startPosition = [-3 -3];
chargingPosition = [13 0];

wPts = [[startPosition 0.1]; ...
    5 0 0.1; ...
    10 0 0.1; ...
    13.75 0 0.1]; %Charging station

toa = [0 4 7 10];
traj = waypointTrajectory(Waypoints=wPts,...
    TimeOfArrival=toa, ReferenceFrame='ENU', ...
    SampleRate=sampleRate);
[pos, orient, vel, acc, angvel] = traj.lookupPose(0:1/sampleRate:10);
```

Add a `robotPlatform` to the scene for our mobile robot. Load the Clearpath Husky model for the `rigidBodyTree` of the `robotPlatform`. Also add cuboid meshes to denote obstacles in the scene. Add a 1-by-1 plane to denote where the charging station is.

```
robot = robotPlatform("rst", scenario,...
    RigidBodyTree=loadrobot("clearpathHusky"), ...
    InitialBasePosition=pos(1,:), InitialBaseOrientation=compact(orient(1)));

addMesh(scenario,"Box",Position=[3  5 2],Size=[4 2 4],Color=[1 0.5 0.25],IsBinaryOccupied=true);
addMesh(scenario,"Box",Position=[3 -5 2],Size=[4 2 4],Color=[1 0.5 0.25],IsBinaryOccupied=true);
addMesh(scenario,"Box",Position=[7  5 2],Size=[4 2 4],Color=[1 0.5 0.25],IsBinaryOccupied=true);
addMesh(scenario,"Box",Position=[7 -5 2],Size=[4 2 4],Color=[1 0.5 0.25],IsBinaryOccupied=true);
addMesh(scenario,"Box",Position=[-3  -5 0.5],Size=[1 1 1],Color=[0.1 0.1 0.1]);

% Plane to denote Charging station location
addMesh(scenario,"Plane",Position=[13 0 .05],Size=[1 1],Color=[0 1 0]);
```

Create the charging station using a `robotPlatform` object. The `robotPlatform` allows us to fetch the transform between the object and the sensor for use in the custom sensor read method. Here, the charging station can be modeled using a cuboid. The robot has to reach within 5cm of the surface of the charging station to start charging.

```
chargeStation = robotPlatform("chargeStation", scenario,InitialBasePosition=[13.75 0 0]);
chargeStation.updateMesh("Cuboid",Size=[0.5 1 1], Color=[0 0.8 0]);
```

The ultrasonic sensor model requires inputs of the profile of the obstacles to be detected. The profile structure includes information about the dimensions of the obstacle.

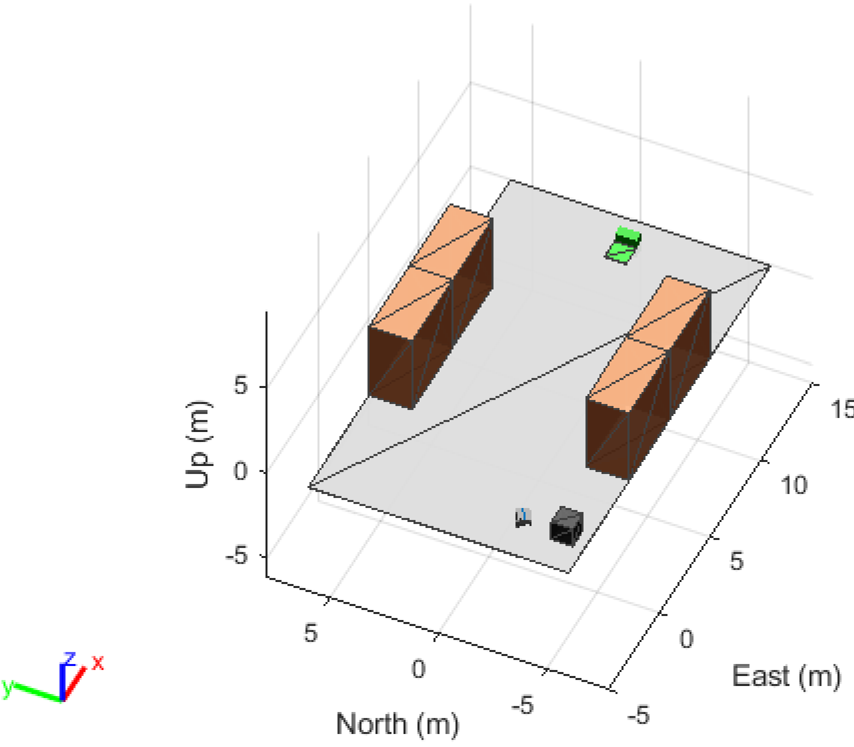
```
chargingStationProfile = struct("Length", 0.5, "Width", 1, "Height", 1, 'OriginOffset', [0 0 0]);
```

Create the ultrasonic sensor using the `ultrasonicDetectionGenerator` object and set its mounting location to `[0 0 0]`, detection range to `[0.03 0.04 5]` and field of view to `[70, 35]`. Also pass in the profile of the charging station that was created earlier.

```
ultraSonicSensorModel = ultrasonicDetectionGenerator(MountingLocation=[0 0 0], ...
    DetectionRange=[0.03 0.04 5], ...
    FieldOfView=[70, 35], ...
    Profiles=chargingStationProfile);
```

Create a `robotSensor` object that uses the custom sensor adaptor `CustomUltrasonicSensor`. The adaptor uses the ultrasonic sensor model created above. The mounting location will be at the front of the robot.

```
ult = robotSensor("UltraSonic", robot, ...
    CustomUltrasonicSensor(ultraSonicSensorModel), ...
    MountingLocation=[0.5 0 0.05]);

figure(1);
ax = show3D(scenario);
view(-65,45)
light
grid on
```

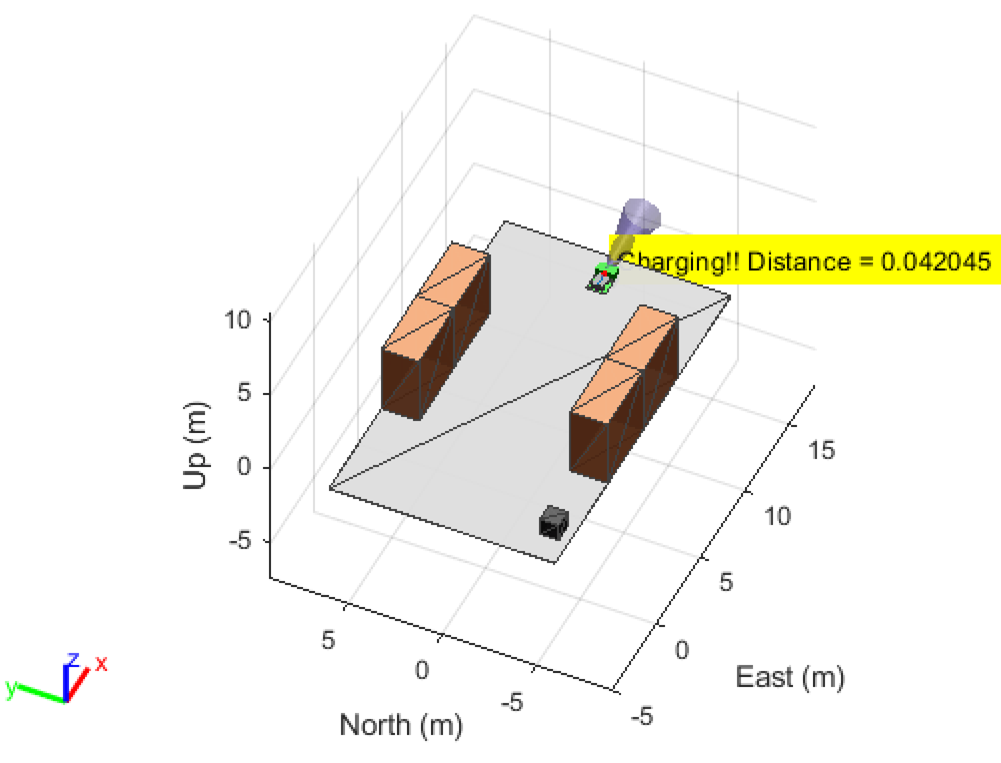


In this scene, the mobile robot will follow the trajectory to the charging station. When the ultrasonic sensor comes within a range of 20cm of the charging station, then mobile robot advance at a slower

rate of 1cm per frame towards the charging station. When the robot is within 5cm of the surface of the charging station, it stops and the charging starts. The simulation ends when the charging starts.

```
isCharging = false;
i = 1;

setup(scenario);

while ~isCharging
    [isUpdated, t, det, isValid] = read(ult);

    figure(1);
    show3D(scenario);
    view(-65,45)
    light
    grid on

    % Read the motion vector of the robot from the platform ground truth
    % This motion vector will be used only for plotting graphic elements
    pose = robot.read();
    rotAngle = quat2eul(pose(10:13));
    hold on

    if ~isempty(det)

        % Distance to object
        distance = det{1}.Measurement;

        % Plot a red shpere where the ultrasonic sensor detects an object
        exampleHelperPlotDetectionPoint(scenario, ...
            det{1}.ObjectAttributes{1}.PointOnTarget, ...
            ult.Name, ...
            pose);

        displayText = ['Distance = ',num2str(distance)];
    else
        distance = inf;
        displayText = 'No object detected!';
    end

    % Plot a cone to represent the field of view and range of the ultrasonic sensor
    exampleHelperPlotFOVCylinder(pose, ultraSonicSensorModel.DetectionRange(3));
    hold off

    if distance <= 0.2
        % Advance in steps of 1cm when the robot is within 20cm of the charging station
        currentMotion = lastMotion;
        currentMotion(1) = currentMotion(1) + 0.01;

        move(robot,"base",currentMotion);
        lastMotion = currentMotion;
        displayText = ['Detected Charger! Distance = ',num2str(distance)];
        if distance <= 0.05
            % The robot is charging when it is within 5cm of the charging station
            displayText = ['Charging!! Distance = ',num2str(distance)];
            isCharging = true;
        end
    else
```

```
            % Follow the waypointTrajectory to the vicinity of the charging station
            if i<=length(pos)
                motion = [pos(i,:), vel(i,:), acc(i,:), ...
                    compact(orient(i)), angvel(i,:)];
                move(robot,"base",motion);
                lastMotion = motion;
                i=i+1;
            end
        end

        % Display the distance to the charging station detected by the ultrasonic sensor
        t = text(15, 0, displayText, "BackgroundColor",'yellow');
        t(1).Color = 'black';
        t(1).FontSize = 10;

        advance(scenario);

        updateSensors(scenario);
    end
```



Version History

Introduced in R2022b

See Also

Functions
`setup` | `read` | `reset` | `robotics.SensorAdaptor.getMotion` | `createCustomRobotSensorTemplate`

Objects
`robotics.SensorAdaptor` | `robotScenario` | `robotPlatform` | `robotSensor`

robotics.SensorAdaptor.getMotion

Class: `robotics.SensorAdaptor`
Package: `robotics`

Get sensor motion in platform reference frame

Syntax

```
motion = getMotion(scenario,platform,sensor,time)
```

Description

`motion = getMotion(scenario,platform,sensor,time)` return the sensor motion in the platform reference frame for the specified simulation time.

Input Arguments

`scenario` — Robot scenario
`robotScenario` object

Robot scenario, specified as a `robotScenario` object. This scenario contains the `robotPlatform` object `platform`, which also contains the sensor object `sensorObj`, which is a subclass of `robotics.SensorAdaptor`.

`platform` — Robot platform
`robotPlatform` object

Robot platform, specified as a `robotPlatform` object. This platform contains the sensor object `sensorObj`, which is a subclass of `robotics.SensorAdaptor`.

`sensor` — Robot sensor to add to platform in scenario
`robotSensor` object

Robot sensor to add to a platform in a scenario, specified as a `robotSensor` object.

`time` — Simulation time
positive scalar

Simulation time, specified as a positive scalar in seconds.

Data Types: `double`

Output Arguments

`motion` — Robot platform motion at current instance in scenario
16-element vector

Robot platform motion at the current instance in a robot scenario, returned as a 16-element vector with these elements in this order:

- `[x y z]` — Positions in the *xyz*-axes in meters
- `[vx vy vz]` — Velocities in the *xyz*-directions in meters per second
- `[ax ay az]` — Accelerations in the *xyz*-directions in meters per second squared
- `[qw qx qy qz]` — Quaternion vector for orientation
- `[wx wy wz]` — Angular velocities in radians per second

Data Types: `double`

Examples

Simulate Ultrasonic Sensors Mounted on Mobile Robots

This example focuses on creating and mounting an ultrasonic sensor on a mobile robot in a `robotScenario`. The `ultrasonicDetectionGenerator` from the Automated Driving Toolbox cannot be used directly with `robotScenario`. We will be implementing a custom sensor adaptor for the `ultrasonicDetectionGenerator` that makes it compatible with `robotScenario`. The sensor will be used to position a mobile robot correctly at a charging station.

Create Custom Sensor Adaptor

Use the `createCustomRobotSensorTemplate` function to generate a template sensor and update it to adapt an `ultrasonicDetectionGenerator` object for usage in Robot scenario.

```
createCustomRobotSensorTemplate
```

This example provides the adaptor class `CustomUltrasonicSensor`, which can be viewed using the following command.

```
edit CustomUltrasonicSensor.m
```

Use the Sensor Adaptor in Robot Scenario Simulation

Create a `robotScenario` object with a sample rate of 10.

```
sampleRate = 10;
scenario = robotScenario(UpdateRate=sampleRate);
```

Add a plane mesh to show the warehouse floor.

```
addMesh(scenario,"Plane",Position=[5 0 0],Size=[20 12],Color=[0.7 0.7 0.7]);
```

Create a `waypointTrajectory` that traverses a set of waypoints to the charging station and use the `lookupPose` method of the `waypointTrajectory` object to fetch the pose of the robot along the trajectory.

```
startPosition = [-3 -3];
chargingPosition = [13 0];

wPts = [[startPosition 0.1]; ...
    5 0 0.1; ...
    10 0 0.1; ...
    13.75 0 0.1]; %Charging station

toa = [0 4 7 10];
```

```
traj = waypointTrajectory(Waypoints=wPts,...
    TimeOfArrival=toa, ReferenceFrame='ENU', ...
    SampleRate=sampleRate);
[pos, orient, vel, acc, angvel] = traj.lookupPose(0:1/sampleRate:10);
```

Add a `robotPlatform` to the scene for our mobile robot. Load the Clearpath Husky model for the `rigidBodyTree` of the `robotPlatform`. Also add cuboid meshes to denote obstacles in the scene. Add a 1-by-1 plane to denote where the charging station is.

```
robot = robotPlatform("rst", scenario,...
    RigidBodyTree=loadrobot("clearpathHusky"), ...
    InitialBasePosition=pos(1,:), InitialBaseOrientation=compact(orient(1)));

addMesh(scenario,"Box",Position=[3  5 2],Size=[4 2 4],Color=[1 0.5 0.25],IsBinaryOccupied=true);
addMesh(scenario,"Box",Position=[3 -5 2],Size=[4 2 4],Color=[1 0.5 0.25],IsBinaryOccupied=true);
addMesh(scenario,"Box",Position=[7  5 2],Size=[4 2 4],Color=[1 0.5 0.25],IsBinaryOccupied=true);
addMesh(scenario,"Box",Position=[7 -5 2],Size=[4 2 4],Color=[1 0.5 0.25],IsBinaryOccupied=true);
addMesh(scenario,"Box",Position=[-3  -5 0.5],Size=[1 1 1],Color=[0.1 0.1 0.1]);

% Plane to denote Charging station location
addMesh(scenario,"Plane",Position=[13 0 .05],Size=[1 1],Color=[0 1 0]);
```

Create the charging station using a `robotPlatform` object. The `robotPlatform` allows us to fetch the transform between the object and the sensor for use in the custom sensor read method. Here, the charging station can be modeled using a cuboid. The robot has to reach within 5cm of the surface of the charging station to start charging.

```
chargeStation = robotPlatform("chargeStation", scenario,InitialBasePosition=[13.75 0 0]);
chargeStation.updateMesh("Cuboid",Size=[0.5 1 1], Color=[0 0.8 0]);
```

The ultrasonic sensor model requires inputs of the profile of the obstacles to be detected. The profile structure includes information about the dimensions of the obstacle.

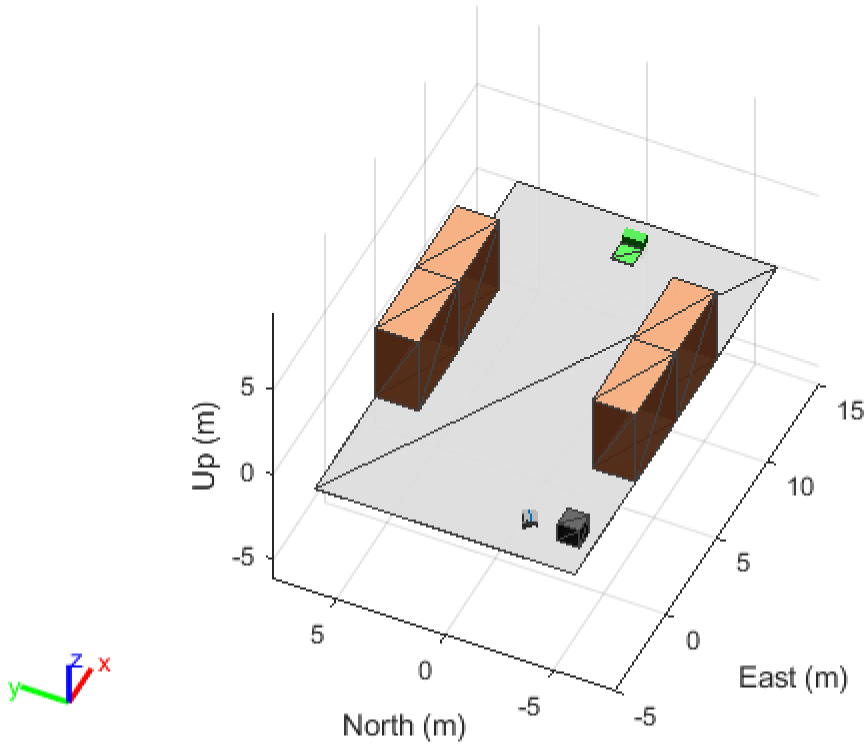
```
chargingStationProfile = struct("Length", 0.5, "Width", 1, "Height", 1, 'OriginOffset', [0 0 0]);
```

Create the ultrasonic sensor using the `ultrasonicDetectionGenerator` object and set its mounting location to `[0 0 0]`, detection range to `[0.03 0.04 5]` and field of view to `[70, 35]`. Also pass in the profile of the charging station that was created earlier.

```
ultraSonicSensorModel = ultrasonicDetectionGenerator(MountingLocation=[0 0 0], ...
    DetectionRange=[0.03 0.04 5], ...
    FieldOfView=[70, 35], ...
    Profiles=chargingStationProfile);
```

Create a `robotSensor` object that uses the custom sensor adaptor `CustomUltrasonicSensor`. The adaptor uses the ultrasonic sensor model created above. The mounting location will be at the front of the robot.

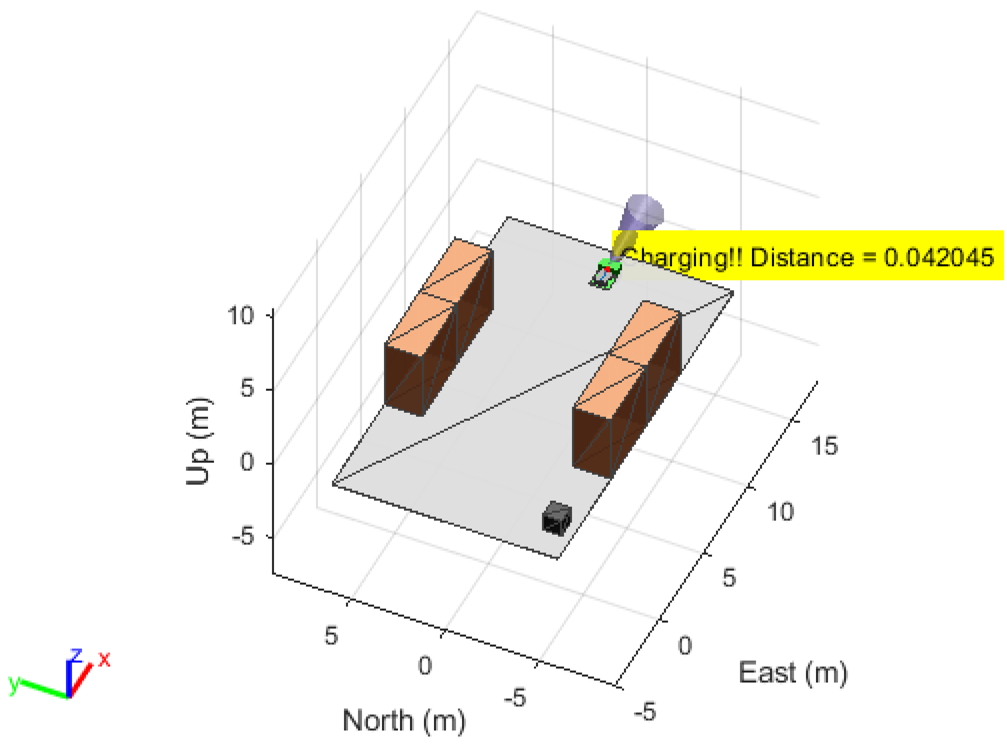
```
ult = robotSensor("UltraSonic", robot, ...
    CustomUltrasonicSensor(ultraSonicSensorModel), ...
    MountingLocation=[0.5 0 0.05]);

figure(1);
ax = show3D(scenario);
view(-65,45)
light
grid on
```



In this scene, the mobile robot will follow the trajectory to the charging station. When the ultrasonic sensor comes within a range of 20cm of the charging station, then mobile robot advance at a slower rate of 1cm per frame towards the charging station. When the robot is within 5cm of the surface of the charging station, it stops and the charging starts. The simulation ends when the charging starts.

```
isCharging = false;
i = 1;

setup(scenario);

while ~isCharging
    [isUpdated, t, det, isValid] = read(ult);

    figure(1);
    show3D(scenario);
    view(-65,45)
    light
    grid on

    % Read the motion vector of the robot from the platform ground truth
    % This motion vector will be used only for plotting graphic elements
    pose = robot.read();
    rotAngle = quat2eul(pose(10:13));
    hold on

    if ~isempty(det)
```

```
            % Distance to object
            distance = det{1}.Measurement;

            % Plot a red shpere where the ultrasonic sensor detects an object
            exampleHelperPlotDetectionPoint(scenario, ...
                det{1}.ObjectAttributes{1}.PointOnTarget, ...
                ult.Name, ...
                pose);

            displayText = ['Distance = ',num2str(distance)];
        else
            distance = inf;
            displayText = 'No object detected!';
        end

        % Plot a cone to represent the field of view and range of the ultrasonic sensor
        exampleHelperPlotFOVCylinder(pose, ultraSonicSensorModel.DetectionRange(3));
        hold off

        if distance <= 0.2
            % Advance in steps of 1cm when the robot is within 20cm of the charging station
            currentMotion = lastMotion;
            currentMotion(1) = currentMotion(1) + 0.01;

            move(robot,"base",currentMotion);
            lastMotion = currentMotion;
            displayText = ['Detected Charger! Distance = ',num2str(distance)];
            if distance <= 0.05
                % The robot is charging when it is within 5cm of the charging station
                displayText = ['Charging!! Distance = ',num2str(distance)];
                isCharging = true;
            end
        else
            % Follow the waypointTrajectory to the vicinity of the charging station
            if i<=length(pos)
                motion = [pos(i,:), vel(i,:), acc(i,:), ...
                    compact(orient(i)), angvel(i,:)];
                move(robot,"base",motion);
                lastMotion = motion;
                i=i+1;
            end
        end

        % Display the distance to the charging station detected by the ultrasonic sensor
        t = text(15, 0, displayText, "BackgroundColor",'yellow');
        t(1).Color = 'black';
        t(1).FontSize = 10;

        advance(scenario);

        updateSensors(scenario);
    end
```



Version History

Introduced in R2022b

See Also

Functions
`setup` | `read` | `reset` | `getEmptyOutputs` | `createCustomRobotSensorTemplate`

Objects
`robotics.SensorAdaptor` | `robotScenario` | `robotPlatform` | `robotSensor`

read

Class: `robotics.SensorAdaptor`
Package: `robotics`

Read from custom sensor model

Syntax

```
varargout = read(sensorObj,scenario,platform,sensor,time)
```

Description

`varargout = read(sensorObj,scenario,platform,sensor,time)` reads sensor data from the sensor model `sensorObj`. Specify the robot scenario, platform, sensor, and simulation time. The function returns the sensor readings from the implemented sensor model.

Input Arguments

`sensorObj` — Robot sensor model
object of subclass of `robotics.SensorAdaptor`

Robot sensor model, specified as an object of a subclass of `robotics.SensorAdaptor`.

`scenario` — Robot scenario
`robotScenario` object

Robot scenario, specified as a `robotScenario` object. This scenario contains the `robotPlatform` object `platform`, which also contains the sensor object `sensorObj`, which is a subclass of `robotics.SensorAdaptor`.

`platform` — Robot platform
`robotPlatform` object

Robot platform, specified as a `robotPlatform` object. This platform contains the sensor object `sensorObj`, which is a subclass of `robotics.SensorAdaptor`.

`sensor` — Robot sensor to add to platform in scenario
`robotSensor` object

Robot sensor to add to a platform in a scenario, specified as a `robotSensor` object.

`time` — Simulation time
positive scalar

Simulation time, specified as a positive scalar in seconds.

Data Types: `double`

Output Arguments

varargout — Variable-length output argument list
varargout

Variable-length output argument list, returned as `varargout`.

Examples

Simulate Ultrasonic Sensors Mounted on Mobile Robots

This example focuses on creating and mounting an ultrasonic sensor on a mobile robot in a `robotScenario`. The `ultrasonicDetectionGenerator` from the Automated Driving Toolbox cannot be used directly with `robotScenario`. We will be implementing a custom sensor adaptor for the `ultrasonicDetectionGenerator` that makes it compatible with `robotScenario`. The sensor will be used to position a mobile robot correctly at a charging station.

Create Custom Sensor Adaptor

Use the `createCustomRobotSensorTemplate` function to generate a template sensor and update it to adapt an `ultrasonicDetectionGenerator` object for usage in Robot scenario.

```
createCustomRobotSensorTemplate
```

This example provides the adaptor class `CustomUltrasonicSensor`, which can be viewed using the following command.

```
edit CustomUltrasonicSensor.m
```

Use the Sensor Adaptor in Robot Scenario Simulation

Create a `robotScenario` object with a sample rate of 10.

```
sampleRate = 10;
scenario = robotScenario(UpdateRate=sampleRate);
```

Add a plane mesh to show the warehouse floor.

```
addMesh(scenario,"Plane",Position=[5 0 0],Size=[20 12],Color=[0.7 0.7 0.7]);
```

Create a `waypointTrajectory` that traverses a set of waypoints to the charging station and use the `lookupPose` method of the `waypointTrajectory` object to fetch the pose of the robot along the trajectory.

```
startPosition = [-3 -3];
chargingPosition = [13 0];

wPts = [[startPosition 0.1]; ...
    5 0 0.1; ...
    10 0 0.1; ...
    13.75 0 0.1]; %Charging station

toa = [0 4 7 10];
traj = waypointTrajectory(Waypoints=wPts,...
    TimeOfArrival=toa, ReferenceFrame='ENU', ...
```

```
    SampleRate=sampleRate);
[pos, orient, vel, acc, angvel] = traj.lookupPose(0:1/sampleRate:10);
```

Add a `robotPlatform` to the scene for our mobile robot. Load the Clearpath Husky model for the `rigidBodyTree` of the `robotPlatform`. Also add cuboid meshes to denote obstacles in the scene. Add a 1-by-1 plane to denote where the charging station is.

```
robot = robotPlatform("rst", scenario,...
    RigidBodyTree=loadrobot("clearpathHusky"), ...
    InitialBasePosition=pos(1,:), InitialBaseOrientation=compact(orient(1)));

addMesh(scenario,"Box",Position=[3  5 2],Size=[4 2 4],Color=[1 0.5 0.25],IsBinaryOccupied=true);
addMesh(scenario,"Box",Position=[3 -5 2],Size=[4 2 4],Color=[1 0.5 0.25],IsBinaryOccupied=true);
addMesh(scenario,"Box",Position=[7  5 2],Size=[4 2 4],Color=[1 0.5 0.25],IsBinaryOccupied=true);
addMesh(scenario,"Box",Position=[7 -5 2],Size=[4 2 4],Color=[1 0.5 0.25],IsBinaryOccupied=true);
addMesh(scenario,"Box",Position=[-3  -5 0.5],Size=[1 1 1],Color=[0.1 0.1 0.1]);

% Plane to denote Charging station location
addMesh(scenario,"Plane",Position=[13 0 .05],Size=[1 1],Color=[0 1 0]);
```

Create the charging station using a `robotPlatform` object. The `robotPlatform` allows us to fetch the transform between the object and the sensor for use in the custom sensor read method. Here, the charging station can be modeled using a cuboid. The robot has to reach within 5cm of the surface of the charging station to start charging.

```
chargeStation = robotPlatform("chargeStation", scenario,InitialBasePosition=[13.75 0 0]);
chargeStation.updateMesh("Cuboid",Size=[0.5 1 1], Color=[0 0.8 0]);
```

The ultrasonic sensor model requires inputs of the profile of the obstacles to be detected. The profile structure includes information about the dimensions of the obstacle.

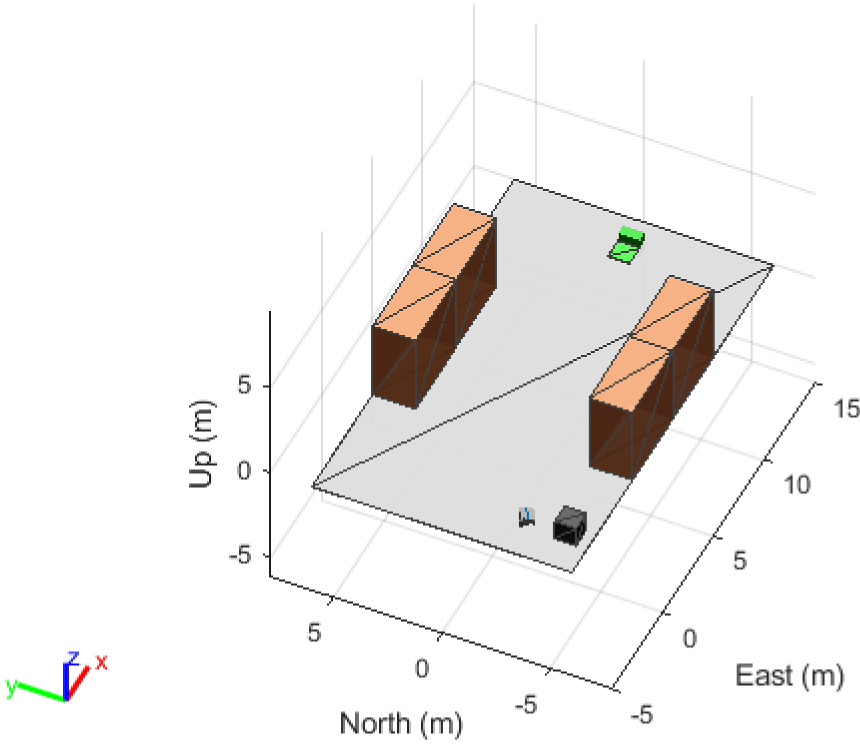
```
chargingStationProfile = struct("Length", 0.5, "Width", 1, "Height", 1, 'OriginOffset', [0 0 0]);
```

Create the ultrasonic sensor using the `ultrasonicDetectionGenerator` object and set its mounting location to `[0 0 0]`, detection range to `[0.03 0.04 5]` and field of view to `[70, 35]`. Also pass in the profile of the charging station that was created earlier.

```
ultraSonicSensorModel = ultrasonicDetectionGenerator(MountingLocation=[0 0 0], ...
    DetectionRange=[0.03 0.04 5], ...
    FieldOfView=[70, 35], ...
    Profiles=chargingStationProfile);
```

Create a `robotSensor` object that uses the custom sensor adaptor `CustomUltrasonicSensor`. The adaptor uses the ultrasonic sensor model created above. The mounting location will be at the front of the robot.

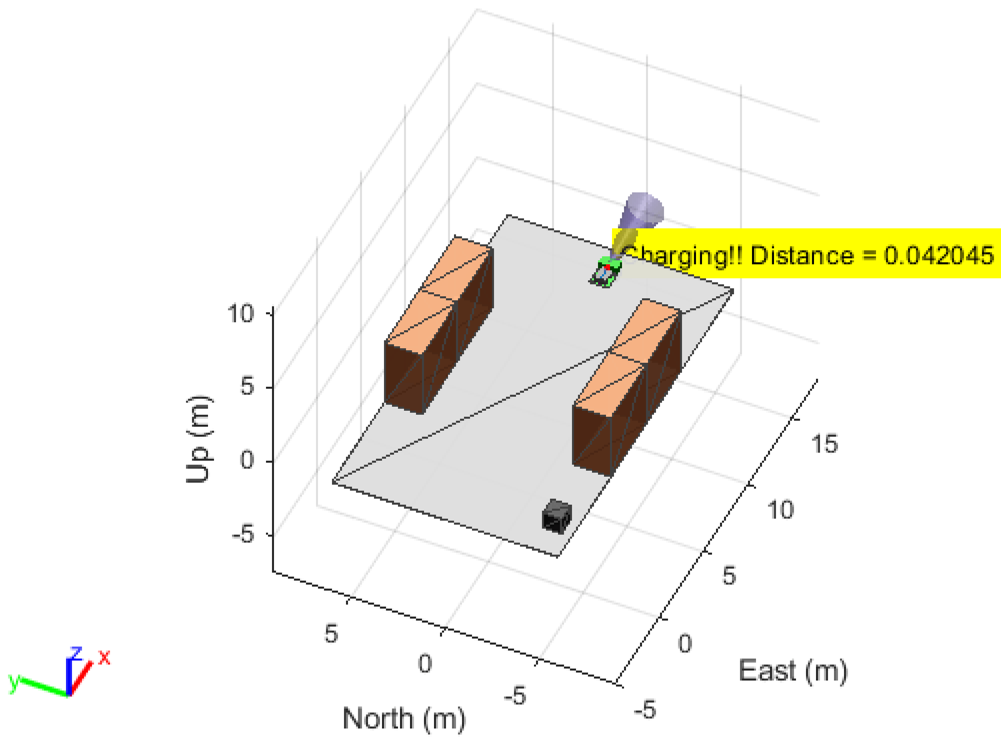
```
ult = robotSensor("UltraSonic", robot, ...
    CustomUltrasonicSensor(ultraSonicSensorModel), ...
    MountingLocation=[0.5 0 0.05]);

figure(1);
ax = show3D(scenario);
view(-65,45)
light
grid on
```



In this scene, the mobile robot will follow the trajectory to the charging station. When the ultrasonic sensor comes within a range of 20cm of the charging station, then mobile robot advance at a slower rate of 1cm per frame towards the charging station. When the robot is within 5cm of the surface of the charging station, it stops and the charging starts. The simulation ends when the charging starts.

```
isCharging = false;
i = 1;

setup(scenario);

while ~isCharging
    [isUpdated, t, det, isValid] = read(ult);

    figure(1);
    show3D(scenario);
    view(-65,45)
    light
    grid on

    % Read the motion vector of the robot from the platform ground truth
    % This motion vector will be used only for plotting graphic elements
    pose = robot.read();
    rotAngle = quat2eul(pose(10:13));
    hold on

    if ~isempty(det)
```

```
        % Distance to object
        distance = det{1}.Measurement;

        % Plot a red shpere where the ultrasonic sensor detects an object
        exampleHelperPlotDetectionPoint(scenario, ...
            det{1}.ObjectAttributes{1}.PointOnTarget, ...
            ult.Name, ...
            pose);

        displayText = ['Distance = ',num2str(distance)];
    else
        distance = inf;
        displayText = 'No object detected!';
    end

    % Plot a cone to represent the field of view and range of the ultrasonic sensor
    exampleHelperPlotFOVCylinder(pose, ultraSonicSensorModel.DetectionRange(3));
    hold off

    if distance <= 0.2
        % Advance in steps of 1cm when the robot is within 20cm of the charging station
        currentMotion = lastMotion;
        currentMotion(1) = currentMotion(1) + 0.01;

        move(robot,"base",currentMotion);
        lastMotion = currentMotion;
        displayText = ['Detected Charger! Distance = ',num2str(distance)];
        if distance <= 0.05
            % The robot is charging when it is within 5cm of the charging station
            displayText = ['Charging!! Distance = ',num2str(distance)];
            isCharging = true;
        end
    else
        % Follow the waypointTrajectory to the vicinity of the charging station
        if i<=length(pos)
            motion = [pos(i,:), vel(i,:), acc(i,:), ...
                compact(orient(i)), angvel(i,:)];
            move(robot,"base",motion);
            lastMotion = motion;
            i=i+1;
        end
    end

    % Display the distance to the charging station detected by the ultrasonic sensor
    t = text(15, 0, displayText, "BackgroundColor",'yellow');
    t(1).Color = 'black';
    t(1).FontSize = 10;

    advance(scenario);

    updateSensors(scenario);
end
```



Version History

Introduced in R2022b

See Also

Functions

`setup` | `reset` | `getEmptyOutputs` | `robotics.SensorAdaptor.getMotion` | `createCustomRobotSensorTemplate`

Objects

`robotics.SensorAdaptor` | `robotScenario` | `robotPlatform` | `robotSensor`

reset

Class: `robotics.SensorAdaptor`
Package: `robotics`

Reset custom sensor model

Syntax

`reset(sensorObj)`

Description

`reset(sensorObj)` resets the sensor model state and releases internal resources if needed.

Input Arguments

`sensorObj` — Robot sensor model
object of subclass of `robotics.SensorAdaptor`

Robot sensor model, specified as an object of a subclass of `robotics.SensorAdaptor`.

Examples

Simulate Ultrasonic Sensors Mounted on Mobile Robots

This example focuses on creating and mounting an ultrasonic sensor on a mobile robot in a `robotScenario`. The `ultrasonicDetectionGenerator` from the Automated Driving Toolbox cannot be used directly with `robotScenario`. We will be implementing a custom sensor adaptor for the `ultrasonicDetectionGenerator` that makes it compatible with `robotScenario`. The sensor will be used to position a mobile robot correctly at a charging station.

Create Custom Sensor Adaptor

Use the `createCustomRobotSensorTemplate` function to generate a template sensor and update it to adapt an `ultrasonicDetectionGenerator` object for usage in Robot scenario.

```
createCustomRobotSensorTemplate
```

This example provides the adaptor class `CustomUltrasonicSensor`, which can be viewed using the following command.

```
edit CustomUltrasonicSensor.m
```

Use the Sensor Adaptor in Robot Scenario Simulation

Create a `robotScenario` object with a sample rate of 10.

```
sampleRate = 10;
scenario = robotScenario(UpdateRate=sampleRate);
```

Add a plane mesh to show the warehouse floor.

```
addMesh(scenario,"Plane",Position=[5 0 0],Size=[20 12],Color=[0.7 0.7 0.7]);
```

Create a `waypointTrajectory` that traverses a set of waypoints to the charging station and use the `lookupPose` method of the `waypointTrajectory` object to fetch the pose of the robot along the trajectory.

```
startPosition = [-3 -3];
chargingPosition = [13 0];

wPts = [[startPosition 0.1]; ...
    5 0 0.1; ...
    10 0 0.1; ...
    13.75 0 0.1]; %Charging station

toa = [0 4 7 10];
traj = waypointTrajectory(Waypoints=wPts,...
    TimeOfArrival=toa, ReferenceFrame='ENU', ...
    SampleRate=sampleRate);
[pos, orient, vel, acc, angvel] = traj.lookupPose(0:1/sampleRate:10);
```

Add a `robotPlatform` to the scene for our mobile robot. Load the Clearpath Husky model for the `rigidBodyTree` of the `robotPlatform`. Also add cuboid meshes to denote obstacles in the scene. Add a 1-by-1 plane to denote where the charging station is.

```
robot = robotPlatform("rst", scenario,...
    RigidBodyTree=loadrobot("clearpathHusky"), ...
    InitialBasePosition=pos(1,:), InitialBaseOrientation=compact(orient(1)));

addMesh(scenario,"Box",Position=[3  5 2],Size=[4 2 4],Color=[1 0.5 0.25],IsBinaryOccupied=true);
addMesh(scenario,"Box",Position=[3 -5 2],Size=[4 2 4],Color=[1 0.5 0.25],IsBinaryOccupied=true);
addMesh(scenario,"Box",Position=[7  5 2],Size=[4 2 4],Color=[1 0.5 0.25],IsBinaryOccupied=true);
addMesh(scenario,"Box",Position=[7 -5 2],Size=[4 2 4],Color=[1 0.5 0.25],IsBinaryOccupied=true);
addMesh(scenario,"Box",Position=[-3  -5 0.5],Size=[1 1 1],Color=[0.1 0.1 0.1]);

% Plane to denote Charging station location
addMesh(scenario,"Plane",Position=[13 0 .05],Size=[1 1],Color=[0 1 0]);
```

Create the charging station using a `robotPlatform` object. The `robotPlatform` allows us to fetch the transform between the object and the sensor for use in the custom sensor read method. Here, the charging station can be modeled using a cuboid. The robot has to reach within 5cm of the surface of the charging station to start charging.

```
chargeStation = robotPlatform("chargeStation", scenario,InitialBasePosition=[13.75 0 0]);
chargeStation.updateMesh("Cuboid",Size=[0.5 1 1], Color=[0 0.8 0]);
```

The ultrasonic sensor model requires inputs of the profile of the obstacles to be detected. The profile structure includes information about the dimensions of the obstacle.

```
chargingStationProfile = struct("Length", 0.5, "Width", 1, "Height", 1, 'OriginOffset', [0 0 0])
```

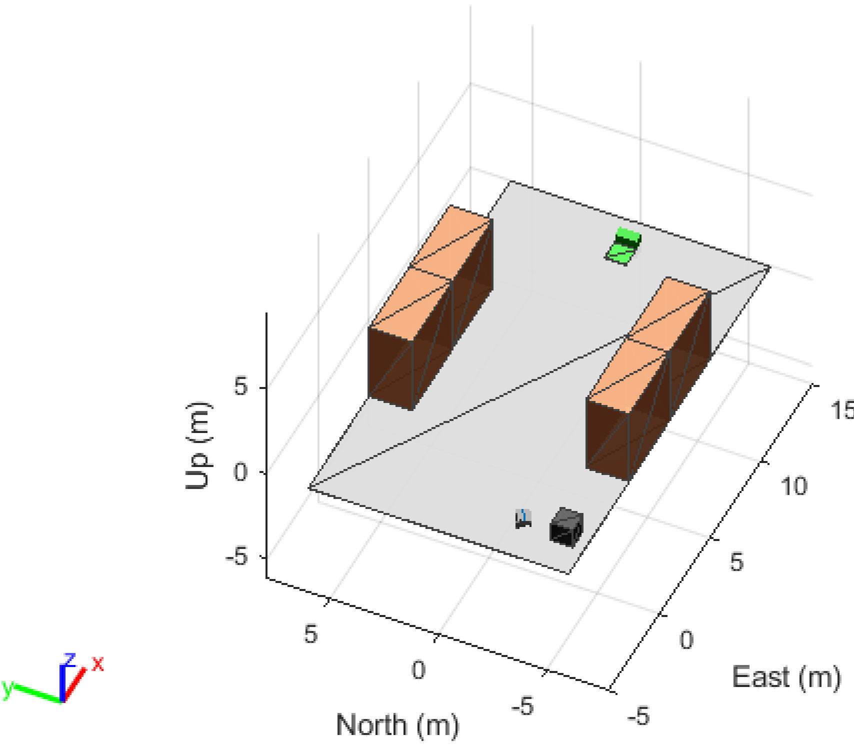
Create the ultrasonic sensor using the `ultrasonicDetectionGenerator` object and set its mounting location to `[0 0 0]`, detection range to `[0.03 0.04 5]` and field of view to `[70, 35]`. Also pass in the profile of the charging station that was created earlier.

```
ultraSonicSensorModel = ultrasonicDetectionGenerator(MountingLocation=[0 0 0], ...
    DetectionRange=[0.03 0.04 5], ...
```

```
    FieldOfView=[70, 35], ...
    Profiles=chargingStationProfile);
```

Create a `robotSensor` object that uses the custom sensor adaptor `CustomUltrasonicSensor`. The adaptor uses the ultrasonic sensor model created above. The mounting location will be at the front of the robot.

```
ult = robotSensor("UltraSonic", robot, ...
    CustomUltrasonicSensor(ultraSonicSensorModel), ...
    MountingLocation=[0.5 0 0.05]);

figure(1);
ax = show3D(scenario);
view(-65,45)
light
grid on
```

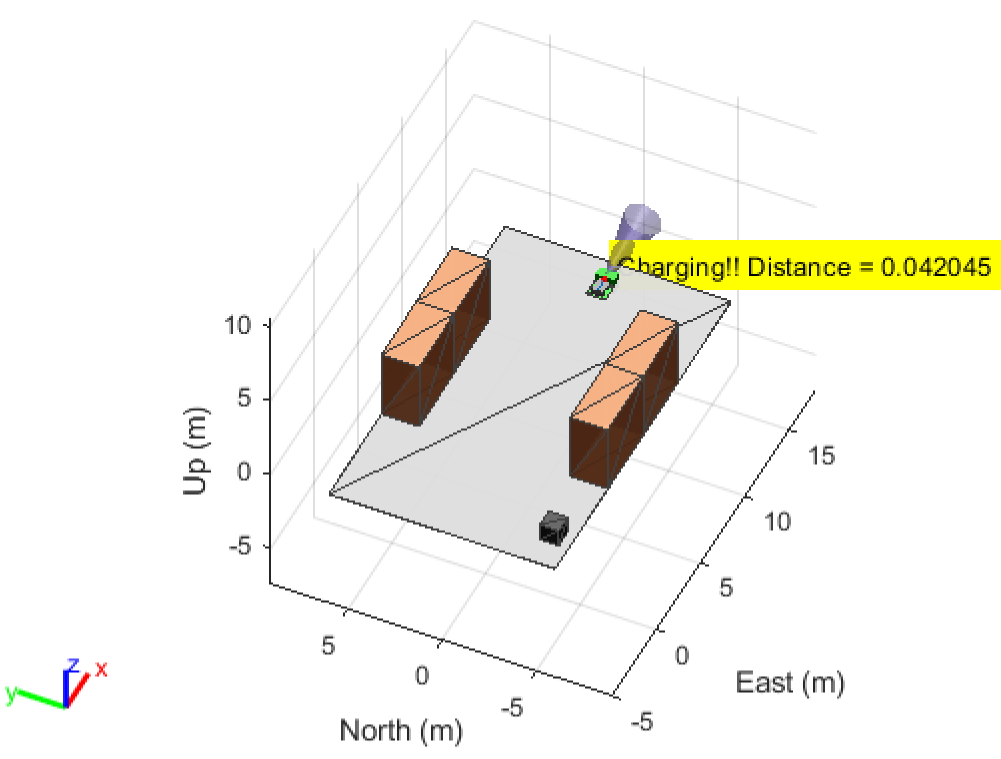


In this scene, the mobile robot will follow the trajectory to the charging station. When the ultrasonic sensor comes within a range of 20cm of the charging station, then mobile robot advance at a slower rate of 1cm per frame towards the charging station. When the robot is within 5cm of the surface of the charging station, it stops and the charging starts. The simulation ends when the charging starts.

```
isCharging = false;
i = 1;

setup(scenario);
```

```
while ~isCharging
    [isUpdated, t, det, isValid] = read(ult);

    figure(1);
    show3D(scenario);
    view(-65,45)
    light
    grid on

    % Read the motion vector of the robot from the platform ground truth
    % This motion vector will be used only for plotting graphic elements
    pose = robot.read();
    rotAngle = quat2eul(pose(10:13));
    hold on

    if ~isempty(det)

        % Distance to object
        distance = det{1}.Measurement;

        % Plot a red shpere where the ultrasonic sensor detects an object
        exampleHelperPlotDetectionPoint(scenario, ...
            det{1}.ObjectAttributes{1}.PointOnTarget, ...
            ult.Name, ...
            pose);

        displayText = ['Distance = ',num2str(distance)];
    else
        distance = inf;
        displayText = 'No object detected!';
    end

    % Plot a cone to represent the field of view and range of the ultrasonic sensor
    exampleHelperPlotFOVCylinder(pose, ultraSonicSensorModel.DetectionRange(3));
    hold off

    if distance <= 0.2
        % Advance in steps of 1cm when the robot is within 20cm of the charging station
        currentMotion = lastMotion;
        currentMotion(1) = currentMotion(1) + 0.01;

        move(robot,"base",currentMotion);
        lastMotion = currentMotion;
        displayText = ['Detected Charger! Distance = ',num2str(distance)];
        if distance <= 0.05
            % The robot is charging when it is within 5cm of the charging station
            displayText = ['Charging!! Distance = ',num2str(distance)];
            isCharging = true;
        end
    else
        % Follow the waypointTrajectory to the vicinity of the charging station
        if i<=length(pos)
            motion = [pos(i,:), vel(i,:), acc(i,:), ...
                compact(orient(i)), angvel(i,:)];
            move(robot,"base",motion);
            lastMotion = motion;
            i=i+1;
        end
```

```
    end

    % Display the distance to the charging station detected by the ultrasonic sensor
    t = text(15, 0, displayText, "BackgroundColor",'yellow');
    t(1).Color = 'black';
    t(1).FontSize = 10;

    advance(scenario);

    updateSensors(scenario);
end
```



Version History

Introduced in R2022b

See Also

Functions
setup | read | getEmptyOutputs | robotics.SensorAdaptor.getMotion | createCustomRobotSensorTemplate

Objects
robotics.SensorAdaptor | robotScenario | robotPlatform | robotSensor

setup

Class: robotics.SensorAdaptor
Package: robotics

Set up custom sensor model

Syntax

```
setup(sensorObj,scenario,platform)
```

Description

setup(sensorObj,scenario,platform) initializes the sensor model with information from the robot scenario and platform to which the sensor is mounted.

Input Arguments

sensorObj — Robot sensor model
object of subclass of robotics.SensorAdaptor

Robot sensor model, specified as an object of a subclass of robotics.SensorAdaptor.

scenario — Robot scenario
robotScenario object

Robot scenario, specified as a robotScenario object. This scenario contains the robotPlatform object platform, which also contains the sensor object sensorObj, which is a subclass of robotics.SensorAdaptor.

platform — Robot platform
robotPlatform object

Robot platform, specified as a robotPlatform object. This platform contains the sensor object sensorObj, which is a subclass of robotics.SensorAdaptor.

Examples

Simulate Ultrasonic Sensors Mounted on Mobile Robots

This example focuses on creating and mounting an ultrasonic sensor on a mobile robot in a robotScenario. The ultrasonicDetectionGenerator from the Automated Driving Toolbox cannot be used directly with robotScenario. We will be implementing a custom sensor adaptor for the ultrasonicDetectionGenerator that makes it compatible with robotScenario. The sensor will be used to position a mobile robot correctly at a charging station.

Create Custom Sensor Adaptor

Use the createCustomRobotSensorTemplate function to generate a template sensor and update it to adapt an ultrasonicDetectionGenerator object for usage in Robot scenario.

```
createCustomRobotSensorTemplate
```

This example provides the adaptor class `CustomUltrasonicSensor`, which can be viewed using the following command.

```
edit CustomUltrasonicSensor.m
```

Use the Sensor Adaptor in Robot Scenario Simulation

Create a `robotScenario` object with a sample rate of 10.

```
sampleRate = 10;
scenario = robotScenario(UpdateRate=sampleRate);
```

Add a plane mesh to show the warehouse floor.

```
addMesh(scenario,"Plane",Position=[5 0 0],Size=[20 12],Color=[0.7 0.7 0.7]);
```

Create a `waypointTrajectory` that traverses a set of waypoints to the charging station and use the `lookupPose` method of the `waypointTrajectory` object to fetch the pose of the robot along the trajectory.

```
startPosition = [-3 -3];
chargingPosition = [13 0];

wPts = [[startPosition 0.1]; ...
    5 0 0.1; ...
    10 0 0.1; ...
    13.75 0 0.1]; %Charging station

toa = [0 4 7 10];
traj = waypointTrajectory(Waypoints=wPts,...
    TimeOfArrival=toa, ReferenceFrame='ENU', ...
    SampleRate=sampleRate);
[pos, orient, vel, acc, angvel] = traj.lookupPose(0:1/sampleRate:10);
```

Add a `robotPlatform` to the scene for our mobile robot. Load the Clearpath Husky model for the `rigidBodyTree` of the `robotPlatform`. Also add cuboid meshes to denote obstacles in the scene. Add a 1-by-1 plane to denote where the charging station is.

```
robot = robotPlatform("rst", scenario,...
    RigidBodyTree=loadrobot("clearpathHusky"), ...
    InitialBasePosition=pos(1,:), InitialBaseOrientation=compact(orient(1)));

addMesh(scenario,"Box",Position=[3  5 2],Size=[4 2 4],Color=[1 0.5 0.25],IsBinaryOccupied=true);
addMesh(scenario,"Box",Position=[3 -5 2],Size=[4 2 4],Color=[1 0.5 0.25],IsBinaryOccupied=true);
addMesh(scenario,"Box",Position=[7  5 2],Size=[4 2 4],Color=[1 0.5 0.25],IsBinaryOccupied=true);
addMesh(scenario,"Box",Position=[7 -5 2],Size=[4 2 4],Color=[1 0.5 0.25],IsBinaryOccupied=true);
addMesh(scenario,"Box",Position=[-3  -5 0.5],Size=[1 1 1],Color=[0.1 0.1 0.1]);

% Plane to denote Charging station location
addMesh(scenario,"Plane",Position=[13 0 .05],Size=[1 1],Color=[0 1 0]);
```

Create the charging station using a `robotPlatform` object. The `robotPlatform` allows us to fetch the transform between the object and the sensor for use in the custom sensor read method. Here, the charging station can be modeled using a cuboid. The robot has to reach within 5cm of the surface of the charging station to start charging.

```
chargeStation = robotPlatform("chargeStation", scenario,InitialBasePosition=[13.75 0 0]);
chargeStation.updateMesh("Cuboid",Size=[0.5 1 1], Color=[0 0.8 0]);
```

The ultrasonic sensor model requires inputs of the profile of the obstacles to be detected. The profile structure includes information about the dimensions of the obstacle.

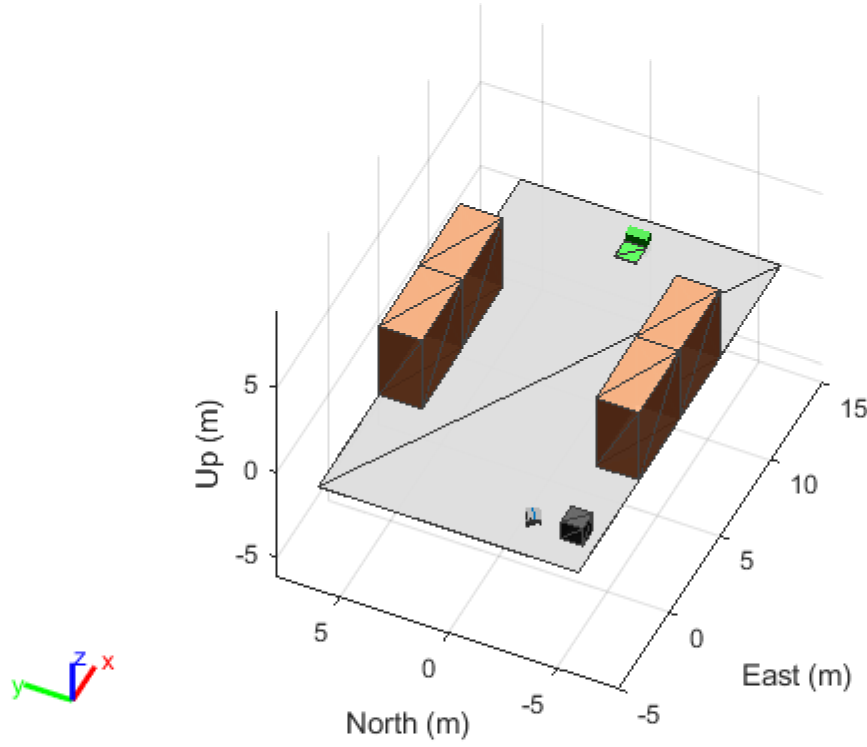
```
chargingStationProfile = struct("Length", 0.5, "Width", 1, "Height", 1, 'OriginOffset', [0 0 0]);
```

Create the ultrasonic sensor using the `ultrasonicDetectionGenerator` object and set its mounting location to `[0 0 0]`, detection range to `[0.03 0.04 5]` and field of view to `[70, 35]`. Also pass in the profile of the charging station that was created earlier.

```
ultraSonicSensorModel = ultrasonicDetectionGenerator(MountingLocation=[0 0 0], ...
    DetectionRange=[0.03 0.04 5], ...
    FieldOfView=[70, 35], ...
    Profiles=chargingStationProfile);
```

Create a `robotSensor` object that uses the custom sensor adaptor `CustomUltrasonicSensor`. The adaptor uses the ultrasonic sensor model created above. The mounting location will be at the front of the robot.

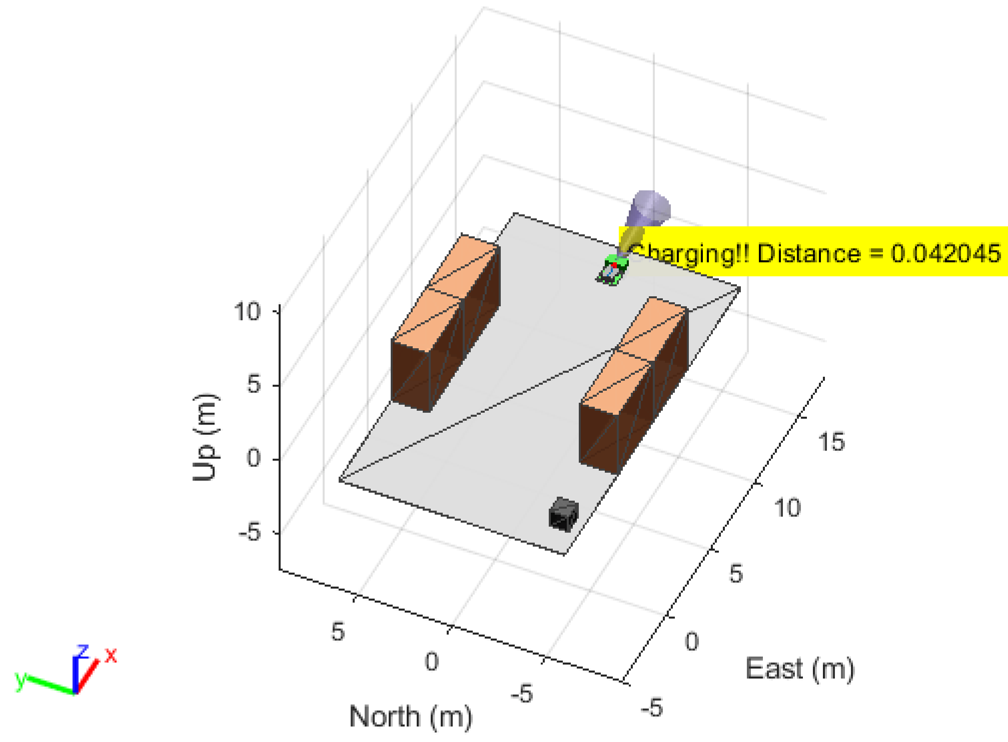
```
ult = robotSensor("UltraSonic", robot, ...
    CustomUltrasonicSensor(ultraSonicSensorModel), ...
    MountingLocation=[0.5 0 0.05]);

figure(1);
ax = show3D(scenario);
view(-65,45)
light
grid on
```



In this scene, the mobile robot will follow the trajectory to the charging station. When the ultrasonic sensor comes within a range of 20cm of the charging station, then mobile robot advance at a slower rate of 1cm per frame towards the charging station. When the robot is within 5cm of the surface of the charging station, it stops and the charging starts. The simulation ends when the charging starts.

```
isCharging = false;
i = 1;

setup(scenario);

while ~isCharging
    [isUpdated, t, det, isValid] = read(ult);

    figure(1);
    show3D(scenario);
    view(-65,45)
    light
    grid on

    % Read the motion vector of the robot from the platform ground truth
    % This motion vector will be used only for plotting graphic elements
    pose = robot.read();
    rotAngle = quat2eul(pose(10:13));
    hold on

    if ~isempty(det)
```

```
        % Distance to object
        distance = det{1}.Measurement;

        % Plot a red shpere where the ultrasonic sensor detects an object
        exampleHelperPlotDetectionPoint(scenario, ...
            det{1}.ObjectAttributes{1}.PointOnTarget, ...
            ult.Name, ...
            pose);

        displayText = ['Distance = ',num2str(distance)];
    else
        distance = inf;
        displayText = 'No object detected!';
    end

    % Plot a cone to represent the field of view and range of the ultrasonic sensor
    exampleHelperPlotFOVCylinder(pose, ultraSonicSensorModel.DetectionRange(3));
    hold off

    if distance <= 0.2
        % Advance in steps of 1cm when the robot is within 20cm of the charging station
        currentMotion = lastMotion;
        currentMotion(1) = currentMotion(1) + 0.01;

        move(robot,"base",currentMotion);
        lastMotion = currentMotion;
        displayText = ['Detected Charger! Distance = ',num2str(distance)];
        if distance <= 0.05
            % The robot is charging when it is within 5cm of the charging station
            displayText = ['Charging!! Distance = ',num2str(distance)];
            isCharging = true;
        end
    else
        % Follow the waypointTrajectory to the vicinity of the charging station
        if i<=length(pos)
            motion = [pos(i,:), vel(i,:), acc(i,:), ...
                compact(orient(i)), angvel(i,:)];
            move(robot,"base",motion);
            lastMotion = motion;
            i=i+1;
        end
    end

    % Display the distance to the charging station detected by the ultrasonic sensor
    t = text(15, 0, displayText, "BackgroundColor",'yellow');
    t(1).Color = 'black';
    t(1).FontSize = 10;

    advance(scenario);

    updateSensors(scenario);
end
```



Version History

Introduced in R2022b

See Also

Functions
`read` | `reset` | `getEmptyOutputs` | `robotics.SensorAdaptor.getMotion` | `createCustomRobotSensorTemplate`

Objects
`robotics.SensorAdaptor` | `robotScenario` | `robotPlatform` | `robotSensor`

axang

Convert transformation or rotation into axis-angle rotations

Syntax

```
angles = axang(transformation)
angles = axang(rotation)
```

Description

`angles = axang(transformation)` converts the rotation of the transformation `transformation` to the axis-angle rotations `angles`.

`angles = axang(rotation)` converts the rotation `rotation` to the axis-angle rotations `angles`.

Examples

Convert SE(3) Transformation to Axis-Angle Rotation

Create SE(3) transformation with no translation but with a rotation defined by an axis-angle rotation. Define the axis-rotation with vector of `[0.5 0.25 0.5]` to be the axis and a `pi/2` rotation about that axis.

```
axa1 = [0.5 0.25 0.5 pi/2];
T = se3(axa1,"axang");
```

Plot the axis-angle and the transformation on the same axes.

```
plot3([0 axa1(1)],[0 axa1(2)],[0 axa1(3)],LineWidth=1)
hold on
plotTransforms(T,FrameAxisLabels="on")
```

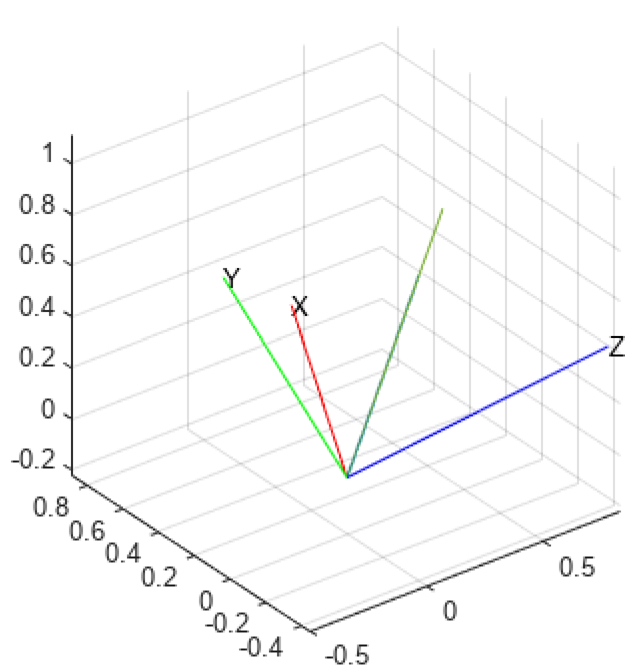
Get the axis-angle rotation from the transformation. Note that the vector of the axis-angle rotation has a different magnitude from the axis-angle rotation specified to the transformation but the defined axis and rotation are the same.

```
axa2 = axang(T)
```

```
axa2 = 1×4

    0.6667    0.3333    0.6667    1.5708
```

Plot the new axis-angle rotation on the same axis.

```
plot3([0 axa2(1)],[0 axa2(2)],[0 axa2(3)])
legend(["Original Rotation Axis","New Rotation Axis"])
hold off
```



Convert SO(3) Rotation to Axis-Angle Rotation

Create SO(3) transformation with a rotation defined by an axis-angle rotation. Define the axis-rotation with vector of `[0.5 0.25 0.5]` to be the axis and a `pi/2` rotation about that axis.

```
axa1 = [0.5 0.25 0.5 pi/2];
R = so3(axa1,"axang");
```

Plot the axis-angle and the transformation on the same axes.

```
plot3([0 axa1(1)],[0 axa1(2)],[0 axa1(3)],LineWidth=1)
hold on
plotTransforms([0 0 0],R,FrameAxisLabels="on")
```

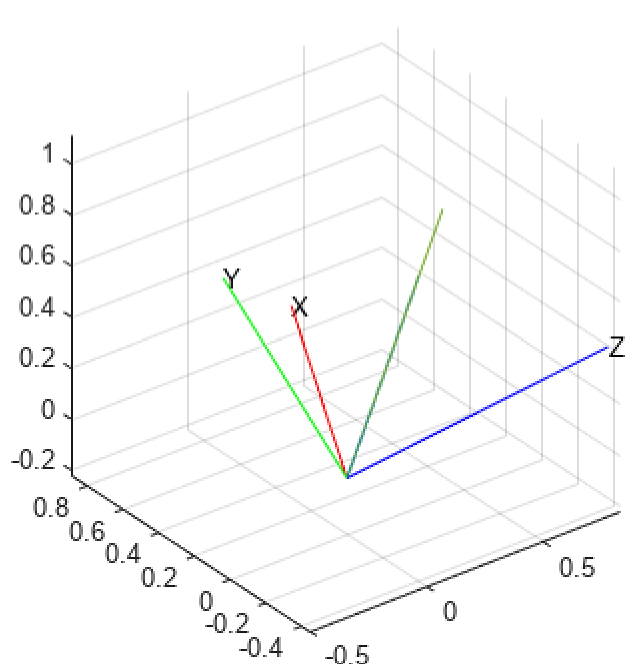
Get the axis-angle rotation from the transformation. Note that the vector of the axis-angle rotation has a different magnitude from the axis-angle rotation specified to the transformation but the defined axis and rotation are the same.

```
axa2 = axang(R)
```

```
axa2 = 1×4

    0.6667    0.3333    0.6667    1.5708
```

Plot the new axis-angle rotation on the same axis.

```
plot3([0 axa2(1)],[0 axa2(2)],[0 axa2(3)])
legend(["Original Rotation Axis","New Rotation Axis"])
hold off
```



Input Arguments

`transformation` — Transformation
`se3` object | *N*-element array of `se3` objects

Transformation, specified as an `se3` object or as an *N*-element array of `se3` objects. *N* is the total number of transformations.

`rotation` — Rotation
`so3` object | *N*-element array of `so3` objects

Rotation, specified as an `so3` object or as an *N*-element array of `so3` objects. *N* is the total number of rotations.

Output Arguments

`angles` — Axis-angle rotation angles
N-by-4 matrix

Axis-angle rotation angles, specified as an *N*-by-4 matrix of *N* axis-angle rotations. The first three elements of every row specify the rotation axes, and the last element defines the rotation angle, in radians.

Version History

Introduced in R2023a

See Also

se3 | so3

eul

Convert transformation or rotation into Euler angles

Syntax

```
angles = eul(transformation)
angles = eul(rotation)
angles = eul( ___ ,sequence)
```

Description

`angles = eul(transformation)` converts the rotation of the transformation `transformation` to the Euler angles `angles`.

`angles = eul(rotation)` converts the rotation `rotation` to the Euler angles `angles`.

`angles = eul( ___ ,sequence)` specifies the sequence of the Euler-angle rotations `sequence` using any of the input arguments in previous syntaxes. For example, a sequence of `"ZYX"` first rotates the *z*-axis, followed by the *y*-axis and *x*-axis.

Examples

Convert SE(3) Transformation to Euler Angles

Create SE(3) transformation with no translation but with a rotation defined by a Euler angles.

```
eul1 = [pi/4 pi/3 pi/8]
```

```
eul1 = 1×3

    0.7854    1.0472    0.3927
```

```
T = se3(eul1,"eul")
```

```
T = se3
    0.3536   -0.4189    0.8364         0
    0.3536    0.8876    0.2952         0
   -0.8660    0.1913    0.4619         0
         0         0         0    1.0000
```

Get the Euler angles from the transformation.

```
eul2 = eul(T)
```

```
eul2 = 1×3

    0.7854    1.0472    0.3927
```

Convert SO(3) Rotation to Euler Angles

Create SO(3) rotation defined by a Euler angles.

```
eul1 = [pi/4 pi/3 pi/8]
```

```
eul1 = 1×3

    0.7854    1.0472    0.3927
```

```
R = so3(eul1,"eul")
```

```
R = so3
    0.3536   -0.4189    0.8364
    0.3536    0.8876    0.2952
   -0.8660    0.1913    0.4619
```

Get the Euler angles from the transformation.

```
eul2 = eul(R)
```

```
eul2 = 1×3

    0.7854    1.0472    0.3927
```

Input Arguments

`transformation` — Transformation
`se3` object | *N*-element array of `se3` objects

Transformation, specified as an `se3` object or as an *N*-element array of `se3` objects. *N* is the total number of transformations.

`rotation` — Rotation
`so3` object | *N*-element array of `so3` objects

Rotation, specified as an `so3` object or as an *N*-element array of `so3` objects. *N* is the total number of rotations.

`sequence` — Axis-rotation sequence
`"ZYX"` (default) | `"ZYZ"` | `"ZXY"` | `"ZXZ"` | `"YXY"` | `"YZX"` | `"YXZ"` | `"YZY"` | `"XYX"` | `"XYZ"` | `"XZX"` | `"XZY"`

Axis-rotation sequence for the Euler angles, specified as one of these string scalars:

- `"ZYX"` (default)
- `"ZYZ"`
- `"ZXY"`
- `"ZXZ"`

- `"YXY"`
- `"YZX"`
- `"YXZ"`
- `"YZY"`
- `"XYX"`
- `"XYZ"`
- `"XZX"`
- `"XZY"`

Each character indicates the corresponding axis. For example, if the sequence is `"ZYX"`, then the three specified Euler angles are interpreted in order as a rotation around the *z*-axis, a rotation around the *y*-axis, and a rotation around the *x*-axis. When applying this rotation to a point, it will apply the axis rotations in the order *x*, then *y*, then *z*.

Data Types: `string` | `char`

Output Arguments

`angles` — Euler angles
M-by-3 matrix

Euler angles, returned as an *M*-by-3 matrix of Euler rotation angles. Each row represents one Euler angle set.

Version History

Introduced in R2023a

See Also

`se3` | `so3`

dist

Calculate distance between transformations

Syntax

```
distance = dist(transformation1,transformation2)
distance = dist(transformation1,transformation2,weights)
distance = dist(rotation1,rotation2)
```

Description

`distance = dist(transformation1,transformation2)` returns the distance `distance` between the poses represented by transformation `transformation1` and transformation `transformation2`.

For the homogeneous transformation objects `se2`, and `se3`, the `dist` function calculates translational and rotational distance independently and combines them in a weighted sum. Translational distance is the Euclidean distance, and rotational distance is the angular difference between the rotation quaternions of `transformation1` and `transformation2`.

`distance = dist(transformation1,transformation2,weights)` specifies the weights `weights` for the translational and rotational distances for calculating the weighted sum of two homogeneous transformations.

`distance = dist(rotation1,rotation2)` returns the distance `distance` between the poses represented by transformation `rotation1` and transformation `rotation2`.

For the homogeneous transformation objects `se2`, and `se3`, the `dist` function calculates translational and rotational distance independently and combines them in a weighted sum. Translational distance is the Euclidean distance, and rotational distance is the angular difference between the rotation quaternions of `rotation1` and `rotation2`.

For rotation objects `so2`, and `so3`, the `dist` function calculates the rotational distance as the angular difference between the rotation quaternions of `rotation1` and `rotation2`.

Input Arguments

`transformation1` — First transformation

`se2` object | `se3` object | *N*-element array of transformation objects

First transformation, specified as a scalar `se2` object, a scalar `se3` object, or as an *N*-element array of transformation objects, where *N* is the total number of transformations. If you specify `transformation1` as an array, each element must be of the same type.

Either `transformation1` or `transformation2` must be a scalar transformation object of the same type. For example, if `transformation1` is an array of `se2` objects, `transformation2` must be a scalar `se2` object.

`transformation2` — Last transformation

`se2` object | `se3` object | *N*-element array of transformation objects

Last transformation, specified as a scalar `se2` object, a scalar `se3` object, or as an *N*-element array of transformation objects, where *N* is the total number of transformations. If you specify `transformation2` as an array, each element must be of the same type.

Either `transformation1` or `transformation2` must be a scalar transformation object of the same type. For example, if `transformation1` is an array of `se2` objects, `transformation2` must be a scalar `se2` object.

`rotation1` — First rotation

`so2` object | `so3` object | *N*-element array of rotation objects

First rotation, specified as a scalar `so2` object, a scalar `so3` object, or as an *N*-element array of rotation objects, where *N* is the total number of rotations. If you specify `rotation1` as an array, each element must be of the same type.

Either `rotation1` or `rotation2` must be a scalar rotation object of the same type. For example, if `rotation1` is an array of `so2` objects, `rotation2` must be a scalar `so2` object.

`rotation2` — Last rotation

`so2` object | `so3` object | *N*-element array of rotation objects

Last rotation, specified as a scalar `so2` object, a scalar `so3` object, or as an *N*-element array of rotation objects, where *N* is the total number of rotations. If you specify `rotation2` as an array, each element must be of the same type.

Either `rotation1` or `rotation2` must be a scalar rotation object of the same type. For example, if `rotation1` is an array of `se2` objects, `rotation2` must be a scalar `se2` object.

`weights` — Weights of translation and rotation in distance sum

`[1.0 0.1]` (default) | two-element row vector

Weights of the translation and rotation in the distance sum, specified as a two-element row vector in the form [*WeightXYZ* *WeightQ*]. *WeightXYZ* is the translational weight and *WeightQ* is the rotational weight. Both weights must be nonnegative numeric values.

Data Types: `single` | `double`

Output Arguments

`distance` — Distance between transformations or rotations

nonnegative numeric scalar

Distance between transformations, returned as a nonnegative numeric scalar. The distance calculate changes depending on the transformation object type of `transformation1` and `transformation2` or `rotation1` and `rotation2`:

- `se2` and `se3` — The `dist` function calculates translational and rotational distance independently and combines them in a weighted sum specified by the `weights` argument. The translational distance is the Euclidean distance between `transformation1` and `transformation2`. The rotational distance is the angular difference between the rotations of `transformation1` and `transformation2`.
- `so2` and `so3` — The `dist` function calculates the rotational distance as the angular difference between the rotations of `rotation1` and `rotation2`.

To calculate the rotational distance, the `dist` function converts the rotation matrix of `transformation1` and `transformation2` or `rotation1` and `rotation2` into `quaternion` objects and uses the `quaternion` `dist` function to calculate the angular distance.

Version History

Introduced in R2022b

See Also

Functions
`normalize` | `interp` | `transform` | `plotTransforms`

Objects
`se2` | `se3` | `so2` | `so3`

interp

Interpolate between transformations

Syntax

```
transformation0 = interp(transformation1,transformation2,points)
rotation0 = interp(rotation1,rotation2,points)
___ = interp( ___ ,transformation2,N)
```

Description

`transformation0 = interp(transformation1,transformation2,points)` interpolates at normalized positions `points` between transformations `transformation1` and `transformation2`.

The function interpolates rotations using a quaternion spherical linear interpolation, and linearly interpolates translations.

`rotation0 = interp(rotation1,rotation2,points)` interpolates at normalized rotations `points` between rotations `rotation1` and `rotation2`.

The function interpolates rotations using a quaternion spherical linear interpolation

`___ = interp( ___ ,transformation2,N)` interpolates `N` steps between the specified transformations or rotations.

Input Arguments

`transformation1` — First transformation
`se2` object | `se3` object | *N*-element array of transformation objects

First transformation, specified as a scalar `se2` object, a scalar `se3` object, or as an *N*-element array of transformation objects, where *N* is the total number of transformations. If you specify `transformation1` as an array, each element must be of the same type.

Either `transformation1` or `transformation2` must be a scalar transformation object of the same type. For example, if `transformation1` is an array of `se2` objects, `transformation2` must be a scalar `se2` object.

`transformation2` — Last transformation
`se2` object | `se3` object | *N*-element array of transformation objects

Last transformation, specified as a scalar `se2` object, a scalar `se3` object, or as an *N*-element array of transformation objects, where *N* is the total number of transformations. If you specify `transformation2` as an array, each element must be of the same type.

Either `transformation1` or `transformation2` must be a scalar transformation object of the same type. For example, if `transformation1` is an array of `se2` objects, `transformation2` must be a scalar `se2` object.

rotation1 — First rotation
so2 object | so3 object | *N*-element array of rotation objects

First rotation, specified as a scalar so2 object, a scalar so3 object, or as an *N*-element array of rotation objects, where *N* is the total number of rotations. If you specify rotation1 as an array, each element must be of the same type.

Either rotation1 or rotation2 must be a scalar rotation object of the same type. For example, if rotation1 is an array of so2 objects, rotation2 must be a scalar so2 object.

rotation2 — Last rotation
so2 object | so3 object | *N*-element array of rotation objects

Last rotation, specified as a scalar so2 object, a scalar so3 object, or as an *N*-element array of rotation objects, where *N* is the total number of rotations. If you specify rotation2 as an array, each element must be of the same type.

Either rotation1 or rotation2 must be a scalar rotation object of the same type. For example, if rotation1 is an array of se2 objects, rotation2 must be a scalar se2 object.

points — Normalized positions
N-element row vector of values in range [0, 1]

Normalized positions, specified as an *N*-element row vector of values in the range [0, 1], where *N* is the total number of interpolated positions. Normalized positions 0 and 1 correspond to the first and last transformations or rotations, respectively.

Example: interp(tf1,tf2,0.5) interpolates a transformation halfway between tf1 and tf2.

Example: interp(r1,r2,0.5) interpolates a rotation halfway between r1 and r2.

N — Number of interpolated positions
positive integer

Number of interpolated positions, specified as a positive integer.

Example: interp(tf1,tf2,5) interpolates five transformations between transformations tf1 and tf2.

Example: interp(r1,r2,7) interpolates seven rotations between rotations r1 and r2.

Output Arguments

transformation0 — Interpolated transformations
N-by-*M* matrix

Interpolated transformations, returned as an *N*-by-*M* matrix of the same transformation type as transformation1 and transformation2, where *N* is the length of the longer argument between transformation1 and transformation2, and *M* is the number of interpolated positions. Each row represents an interpolated transformation between transformation1 and transformation2.

rotation0 — Interpolated rotations
N-by-*M* matrix

Interpolated rotations, returned as an *N*-by-*M* matrix of the same rotation type as rotation1 and rotation2, where *N* is the length of the longer argument between rotation1 and rotation2, and

M is the number of interpolated positions. Each row represents an interpolated transformation between `rotation1` and `rotation2`.

Version History

Introduced in R2022b

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

Functions

`dist` | `normalize` | `transform` | `plotTransforms`

Objects

`se2` | `se3` | `so2` | `so3`

mrdivide, /

Transformation or rotation right division

Syntax

```
transformationC = transformationA/transformationB
rotationC = rotationA/rotationB
```

Description

`transformationC = transformationA/transformationB` right divides transformation `transformationA` by transformation `transformationB` and returns the quotient, transformation `transformationC`. `transformationC` is the same value as `transformationA*inv(transformationB)`.

You can use division to compose a sequence of transformations, so that `transformationC` represents a transformation where the inverse of `transformationB` is applied first, followed by `transformationA`.

`rotationC = rotationA/rotationB` right divides transformation `rotationA` by transformation `rotationB` and returns the quotient, transformation `rotationC`. `rotationC` is the same value as `rotationA*inv(rotationB)`.

Input Arguments

`transformationA` — First transformation
`se2` object | `se3` object | *N*-element array of transformation objects

First transformation, specified as a scalar `se2` object, a scalar `se3` object, or as an *N*-element array of transformation objects, where *N* is the total number of transformations. If you specify `transformationA` as an array, each element must be of the same type.

Either `transformationA` or `transformationB` must be a scalar transformation object of the same type. For example, if `transformationA` is an array of `se2` objects, `transformationB` must be a scalar `se2` object.

`transformationB` — Last transformation
`se2` object | `se3` object | *N*-element array of transformation objects

Last transformation, specified as a scalar `se2` object, a scalar `se3` object, or as an *N*-element array of transformation objects, where *N* is the total number of transformations. If you specify `transformationB` as an array, each element must be of the same type.

Either `transformationA` or `transformationB` must be a scalar transformation object of the same type. For example, if `transformationA` is an array of `se2` objects, `transformationB` must be a scalar `se2` object.

`rotationA` — First rotation
`so2` object | `so3` object | *N*-element array of rotation objects

First rotation, specified as a scalar so2 object, a scalar so3 object, or as an *N*-element array of rotation objects, where *N* is the total number of rotations. If you specify rotationA as an array, each element must be of the same type.

Either rotationA or rotationB must be a scalar rotation object of the same type. For example, if rotationA is an array of so2 objects, rotationB must be a scalar so2 object.

rotationB — Last rotation
so2 object | so3 object | *N*-element array of rotation objects

Last rotation, specified as a scalar so2 object, a scalar so3 object, or as an *N*-element array of rotation objects, where *N* is the total number of rotations. If you specify rotationB as an array, each element must be of the same type.

Either rotationA or rotationB must be a scalar rotation object of the same type. For example, if rotationA is an array of se2 objects, rotationB must be a scalar se2 object.

Output Arguments

transformationC — Transformation quotient
se2 object | se3 object | *N*-element array of transformation objects

Transformation quotient, returned as a scalar se2 object, a scalar se3 object, or as an *N*-element array of the same transformation type as transformationA and transformationB. *N* is the length of the longer argument between transformationA and transformationB and each row represents the quotient between transformationA and transformationB.

rotationC — Rotation quotient
so2 object | so3 object | *N*-element array of rotation objects

Rotation quotient, returned as a scalar so2 object, a scalar so3 object, or as an *N*-element array of the same rotation type as rotationA and rotationB. *N* is the length of the longer argument between rotationA and rotationB and each row represents the quotient between rotationA and rotationB.

Version History

Introduced in R2022b

See Also

Functions
rdivide, ./ | mtimes, * | times, .*

Objects
se2 | se3 | so2 | so3

mtimes, *

Transformation or rotation multiplication

Syntax

```
transformationC = transformationA*transformationB
rotationC = rotationA*rotationB
```

Description

`transformationC = transformationA*transformationB` performs transformation multiplication between transformation `transformationA` and transformation `transformationB` and returns the product, transformation `transformationC`.

You can use transformation multiplication to compose a sequence of transformations, so that `transformationC` represents a transformation where `transformationB` is applied first, followed by `transformationA`.

`rotationC = rotationA*rotationB` performs rotation multiplication between rotation `rotationA` and rotation `rotationB` and returns the product, rotation `rotationC`.

You can use rotation multiplication to compose a sequence of rotations, so that `rotationC` represents a rotation where `rotationB` is applied first, followed by `rotationA`.

Input Arguments

`transformationA` — First transformation

`se2` object | `se3` object | *N*-element array of transformation objects

First transformation, specified as a scalar `se2` object, a scalar `se3` object, or as an *N*-element array of transformation objects, where *N* is the total number of transformations. If you specify `transformationA` as an array, each element must be of the same type.

Either `transformationA` or `transformationB` must be a scalar transformation object of the same type. For example, if `transformationA` is an array of `se2` objects, `transformationB` must be a scalar `se2` object.

`transformationB` — Last transformation

`se2` object | `se3` object | *N*-element array of transformation objects

Last transformation, specified as a scalar `se2` object, a scalar `se3` object, or as an *N*-element array of transformation objects, where *N* is the total number of transformations. If you specify `transformationB` as an array, each element must be of the same type.

Either `transformationA` or `transformationB` must be a scalar transformation object of the same type. For example, if `transformationA` is an array of `se2` objects, `transformationB` must be a scalar `se2` object.

`rotationA` — First rotation

`so2` object | `so3` object | *N*-element array of rotation objects

First rotation, specified as a scalar `so2` object, a scalar `so3` object, or as an *N*-element array of rotation objects, where *N* is the total number of rotations. If you specify `rotationA` as an array, each element must be of the same type.

Either `rotationA` or `rotationB` must be a scalar rotation object of the same type. For example, if `rotationA` is an array of `so2` objects, `rotationB` must be a scalar `so2` object.

`rotationB` — Last rotation
`so2` object | `so3` object | *N*-element array of rotation objects

Last rotation, specified as a scalar `so2` object, a scalar `so3` object, or as an *N*-element array of rotation objects, where *N* is the total number of rotations. If you specify `rotationB` as an array, each element must be of the same type.

Either `rotationA` or `rotationB` must be a scalar rotation object of the same type. For example, if `rotationA` is an array of `se2` objects, `rotationB` must be a scalar `se2` object.

Output Arguments

`transformationC` — Transformation product
`se2` object | `se3` object | *N*-element array of transformation objects

Transformation product, returned as a scalar `se2` object, a scalar `se3` object, or as an *N*-element array of the same transformation type as `transformationA` and `transformationB`. *N* is the length of the longer argument between `transformationA` and `transformationB` and each row represents the product between `transformationA` and `transformationB`.

`rotationC` — Rotation product
`so2` object | `so3` object | *N*-element array of rotation objects

Rotation product, returned as a scalar `so2` object, a scalar `so3` object, or as an *N*-element array of the same rotation type as `rotationA` and `rotationB`. *N* is the length of the longer argument between `rotationA` and `rotationB` and each row represents the product between `rotationA` and `rotationB`.

Version History

Introduced in R2022b

See Also

Functions
`mrdivide, /` | `rdivide, ./` | `times, .*`

Objects
`se2` | `se3` | `so2` | `so3`

normalize

Normalize transformation or rotation matrix

Syntax

```
transformationN = normalize(transformation)
rotationN = normalize(rotation)
___ = normalize(___,Method=normMethod)
```

Description

`transformationN = normalize(transformation)` normalizes the rotation of the transformation `transformation` and returns a transformation, `transformationN`, that is equivalent to `transformation`, but with normalized rotation.

`rotationN = normalize(rotation)` normalizes the rotation of the rotation `rotation` and returns a rotation, `rotationN`, that is equivalent to `rotation`, but with normalized rotation.

Note The transformation and rotation objects do not automatically normalize their rotations. You must use `normalize` each time you need to normalize a transformation or rotation. You may need to do this if:

- You specified an unnormalized input transformation or rotation at the creation of the transformation or rotation object.
- You performed many operations on the transformation or rotation objects such as `mtimes, *`, which may cause the transformation or rotation to become unnormalized due to data type precision.

`___ = normalize(___,Method=normMethod)` specifies the normalization method `normMethod` that the `normalize` function uses to normalize the specified transformation or rotation.

Input Arguments

`transformation` — Transformation
`se2` object | `se3` object | *N*-element array of transformation objects

Transformation, specified as a scalar `se2` object, a scalar `se3` object, or an *N*-element array of transformation objects. *N* is the total number of transformations.

If you specify `transformation` as an array, each element must be of the same type.

`rotation` — Rotation
`so2` object | `so3` object | *N*-element array of rotation objects

Rotation, specified as a scalar `so2` object, a scalar `so3` object, or as an *N*-element array of rotation objects. *N* is the total number of rotations.

If you specify `rotation` as an array, each element must be of the same type.

`normMethod` — Normalization method
`"quat"` (default) | `"cross"` | `"svd"`

Normalization method, specified as one of these options:

- `"quat"` — Convert the rotation submatrix into a normalized quaternion and then convert the normalized quaternion back to a transformation or rotation object. For more information, see the `normalize` of the `quaternion` object.
- `"cross"` — Normalize the third column of the rotation submatrix and then determine the other two columns through cross products.
- `"svd"` — Use singular value decomposition to find the closest orthonormal matrix by setting singular values to `1`. This solves the orthogonal Procrustes problem.

Data Types: `char` | `string`

Output Arguments

`transformationN` — Normalized transformation
`se2` object | `se3` object

Normalized transformation, returned as an `se2` or `se3` object.

`rotationN` — Normalized rotation
`so2` object | `so3` object

Normalized rotation, returned as an `so2` or `so3` object.

Version History

Introduced in R2022b

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

Functions

`dist` | `interp` | `transform` | `plotTransforms`

Objects

`se2` | `se3` | `so2` | `so3` | `quaternion`

plot

Draw transformation coordinate frame

Syntax

```
plot(T)
plot(___,Name=Value)
AX = plot(___,Name=Value)
```

Description

`plot(T)` draws a 3-D coordinate frame of transformation `T` with labeled axes. The *x*-axis is colored in red, the *y*-axis in green, and the *z*-axis in blue.

`plot(___,Name=Value)` specifies optional arguments using one or more name-value arguments. For example, `plot(T,AxisLabels="off")` hides the *xyz* labels.

`AX = plot(___,Name=Value)` returns the axis object, `AX`, containing the transformation plots.

Input Arguments

T — Transformation

SE3 object | SO3 object | *M*-element array of SE3 or SO3 objects

Transformation, specified as either an individual `SE3` or `SO3` object, or as an *M*-element array of transformation objects. *M* is the total number of transformations. Ever transformation in `T` is plotted if `T` is an *M*-element array.

Data Types: `single` | `double`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `plot(T,AxisLabels="off")`

`AxisLabels` — Show axis labels

`"on"` (default) | `"off"`

Show axis labels, specified as `"off"` or `"on"`.

Example: `plot(T,AxisLabels="off")`

Data Types: `char` | `string`

`FrameLabel` — Name of coordinate frame

`""` (default) | string scalar | character vector

Name of the coordinate frame, specified as a string scalar or character vector.

Example: `plot(T,FrameLabel="TF1")`

Data Types: `char` | `string`

`Color` — Use uniform color for coordinate frame
`"off"` (default) | `"on"`

Use uniform color for coordinate frame, specified as `"off"` or `"on"`.

Example: `plot(T,Color="on")`

Data Types: `char` | `string`

Output Arguments

`AX` — Axes handle
`Axes` object

Axes handle, specified as an `Axes` object.

Version History

Introduced in R2022b

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

Functions
`interpolate` | `normalize` | `rotm` | `showdetails` | `tform` | `transformPoints` | `trvec`

Objects
SE3 | SO3

quat

Convert transformation or rotation to numeric quaternion

Syntax

```
q = quat(transformation)
q = quat(rotation)
```

Description

`q = quat(transformation)` creates a quaternion `q` from the rotation of the transformation `transformation`.

`q = quat(rotation)` creates a quaternion `q` from the rotation `rotation`.

Examples

Convert SE(3) Transformation to Numeric Quaternion

Create SE(3) transformation with zero translation and a rotation defined by a numeric quaternion. Use the `eul2quat` function to create the numeric quaternion.

```
quat1 = eul2quat([0 0 deg2rad(30)])
```

```
quat1 = 1×4

    0.9659    0.2588         0         0
```

```
T = se3(quat1,"quat")
```

```
T = se3
    1.0000         0         0         0
         0    0.8660   -0.5000         0
         0    0.5000    0.8660         0
         0         0         0    1.0000
```

Convert the transformation back into a numeric quaternion.

```
quat2 = quat(T)
```

```
quat2 = 1×4

    0.9659    0.2588         0         0
```

Convert SO(3) Rotation to Numeric Quaternion

Create SO(3) rotation defined by a numeric quaternion. Use the `eul2quat` function to create the numeric quaternion.

```
quat1 = eul2quat([0 0 deg2rad(30)])
```

```
quat1 = 1×4

    0.9659    0.2588         0         0
```

```
R = so3(quat1,"quat")
```

```
R = so3
    1.0000         0         0
         0    0.8660   -0.5000
         0    0.5000    0.8660
```

Convert the rotation into a numeric quaternion.

```
quat2 = quat(R)
```

```
quat2 = 1×4

    0.9659    0.2588         0         0
```

Input Arguments

`transformation` — Transformation
`se3` object | *N*-element array of `se3` objects

Transformation, specified as an `se3` object or as an *N*-element array of `se3` objects. *N* is the total number of transformations.

`rotation` — Rotation
`so3` object | *N*-element array of `so3` objects

Rotation, specified as an `so3` object or as an *N*-element array of `so3` objects. *N* is the total number of rotations.

Output Arguments

q — Quaternion rotation angles
M-by-4 matrix

Quaternion rotation angles, returned as an *M*-by-4 matrix, where each row is of the form [*qw qx qy qz*]. *M* is the total number of transformations or rotations specified.

Version History

Introduced in R2023a

See Also

se3 | so3

rdivide, ./

Element-wise transformation or rotation right division

Syntax

```
transformationC = transformationA./transformationB
rotationC = rotationA./rotationB
```

Description

`transformationC = transformationA./transformationB` divides transformations element-by-element by dividing each element of transformation `transformationA` with the corresponding element of transformation `transformationB` and returns the quotient, transformation `transformationC`.

`rotationC = rotationA./rotationB` divides rotations element-by-element by dividing each element of rotation `rotationA` with the corresponding element of rotation `rotationB` and returns the quotient, rotation `rotationC`.

Input Arguments

`transformationA` — First transformation
`se2` object | `se3` object | *N*-element array of transformation objects

First transformation, specified as a scalar `se2` object, a scalar `se3` object, or as an *N*-element array of transformation objects, where *N* is the total number of transformations. If you specify `transformationA` as an array, each element must be of the same type.

Either `transformationA` or `transformationB` must be a scalar transformation object of the same type. For example, if `transformationA` is an array of `se2` objects, `transformationB` must be a scalar `se2` object.

`transformationB` — Last transformation
`se2` object | `se3` object | *N*-element array of transformation objects

Last transformation, specified as a scalar `se2` object, a scalar `se3` object, or as an *N*-element array of transformation objects, where *N* is the total number of transformations. If you specify `transformationB` as an array, each element must be of the same type.

Either `transformationA` or `transformationB` must be a scalar transformation object of the same type. For example, if `transformationA` is an array of `se2` objects, `transformationB` must be a scalar `se2` object.

`rotationA` — First rotation
`so2` object | `so3` object | *N*-element array of rotation objects

First rotation, specified as a scalar `so2` object, a scalar `so3` object, or as an *N*-element array of rotation objects, where *N* is the total number of rotations. If you specify `rotationA` as an array, each element must be of the same type.

Either `rotationA` or `rotationB` must be a scalar rotation object of the same type. For example, if `rotationA` is an array of `so2` objects, `rotationB` must be a scalar `so2` object.

`rotationB` — Last rotation
`so2` object | `so3` object | *N*-element array of rotation objects

Last rotation, specified as a scalar `so2` object, a scalar `so3` object, or as an *N*-element array of rotation objects, where *N* is the total number of rotations. If you specify `rotationB` as an array, each element must be of the same type.

Either `rotationA` or `rotationB` must be a scalar rotation object of the same type. For example, if `rotationA` is an array of `se2` objects, `rotationB` must be a scalar `se2` object.

Output Arguments

`transformationC` — Transformation quotient
`se2` object | `se3` object | *N*-element array of transformation objects

Transformation quotient, returned as a scalar `se2` object, a scalar `se3` object, or as an *N*-element array of the same transformation type as `transformationA` and `transformationB`. *N* is the length of the longer argument between `transformationA` and `transformationB` and each row represents the quotient between `transformationA` and `transformationB`.

`rotationC` — Rotation quotient
`so2` object | `so3` object | *N*-element array of rotation objects

Rotation quotient, returned as a scalar `so2` object, a scalar `so3` object, or as an *N*-element array of the same rotation type as `rotationA` and `rotationB`. *N* is the length of the longer argument between `rotationA` and `rotationB` and each row represents the quotient between `rotationA` and `rotationB`.

Version History

Introduced in R2022b

See Also

Functions
`mrdivide, /` | `mtimes, *` | `times, .*`

Objects
`se2` | `se3` | `so2` | `so3`

rotm

Extract rotation matrix

Syntax

```
rotationMatrix = rotm(transformation)
rotationMatrix = rotm(rotation)
```

Description

`rotationMatrix = rotm(transformation)` returns the rotation matrix `rotationMatrix` from the SE(2) or SE(3) transformation `transformation`.

`rotationMatrix = rotm(rotation)` returns the rotation matrix `rotationMatrix` from the SO(2) or SO(3) rotation `rotation`.

Input Arguments

`transformation` — Transformation

`se2` object | `se3` object | *N*-element array of transformation objects

Transformation, specified as a scalar `se2` object, a scalar `se3` object, or an *N*-element array of transformation objects. *N* is the total number of transformations.

If you specify `transformation` as an array, each element must be of the same type.

`rotation` — Rotation

`so2` object | `so3` object | *N*-element array of rotation objects

Rotation, specified as a scalar `so2` object, a scalar `so3` object, or an *N*-element array of rotation objects. *N* is the total number of rotations.

If you specify `rotation` as an array, each element must be of the same type.

Output Arguments

`rotationMatrix` — Rotation matrix

2-by-2-by-*N* array | 3-by-3-by-*N* array

Rotation matrix, returned as a 2-by-2-by-*N* array for 2-D transformations or a 3-by-3-by-*N* array for 3-D transformations. *N* is the total number of transformations.

Version History

Introduced in R2022b

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

Functions

`dist` | `interp` | `normalize` | `tform` | `transform` | `trvec` | `plotTransforms`

Objects

`se2` | `se3` | `so2` | `so3`

showdetails

Display transformation in compact form

Syntax

```
showdetails(transformation)
showdetails(rotation)
showdetails( ___ Name=Value)
```

Description

`showdetails(transformation)` displays the translational and rotational components of the transformation `transformation` on a single line. The rotation units are in degrees.

`showdetails(rotation)` displays the rotational components of the rotation `rotation` on a single line.

`showdetails( ___ Name=Value)` specifies additional options using one or more name-value arguments.

Input Arguments

`transformation` — Transformation
`se2` object | `se3` object | *N*-element array of transformation objects

Transformation, specified as a scalar `se2` object, a scalar `se3`, or an *N*-element array of transformation objects. *N* is the total number of transforms.

If you specify `transformation` as an array, each element must be of the same type. Additionally, `showdetails` prints the details on a new line for each of the *N* transformations.

`rotation` — Rotation
`so2` object | `so3` object | *N*-element array of rotation objects

Rotation, specified as a scalar `so2` object, a scalar `so3` object, or as an *N*-element array of rotation objects. *N* is the total number of rotations.

If you specify `rotation` as an array, each element must be of the same type. Additionally, `showdetails` prints the details on a new line for each of the *N* rotations.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `showdetails(T,Sequence="ZYX")`

Sequence — Euler angle sequence order
"ZYX" (default) | "ZYZ" | "ZXY" | "ZXZ" | "YXY" | "YZX" | "YXZ" | "YZY" | "XYX" | "XYZ" | "XZX" | "XZY"

Euler angle sequence order, specified as one of these string scalars:

- "ZYX" (default)
- "ZYZ"
- "ZXY"
- "ZXZ"
- "YXY"
- "YZX"
- "YXZ"
- "YZY"
- "XYX"
- "XYZ"
- "XZX"
- "XZY"

Each character indicates the corresponding axis. For example if the sequence is "ZYX", then the printed order of rotation angles is *z*-axis, *y*-axis, and then the *x*-axis.

This parameter does not affect the output when `transformation` contains an `se2` object or if `rotation` contains an `so2` object.

Example: `showdetails(T,Sequence="ZYX")`

Data Types: `char` | `string`

`AngleUnit` — Angle unit
"deg" (default) | "rad"

Angle unit, specified as "deg" for degrees, or "rad" for radians.

Example: `showdetails(T,AngleUnit="rad")`

Data Types: `char` | `string`

Version History

Introduced in R2022b

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

Functions
`dist` | `interp` | `normalize` | `transform`

Objects

se2 | se3 | so2 | so3

theta

Convert transformation or rotation to 2-D rotation angle

Syntax

```
angle = theta(transformation)
angle = theta(rotation)
```

Description

`angle = theta(transformation)` extracts the 2-D rotation angle `angle` from the transformation `transformation`.

`angle = theta(rotation)` extracts the 2-D rotation angle `angle` from the rotation `rotation`.

Examples

Convert SE(2) Transformation to Angle

Create SE(2) transformation with a rotation defined by an angle `pi/2`.

```
angle1 = pi/2
```

```
angle1 = 1.5708
```

```
T = se2(angle1,"theta")
```

```
T = se2
    0.0000   -1.0000         0
    1.0000    0.0000         0
         0         0    1.0000
```

Get the rotation angle from the transformation.

```
angle2 = theta(T)
```

```
angle2 = 1.5708
```

Convert SO(2) Transformation to Angle

Create SO(2) rotation defined by an angle `pi/2`.

```
angle1 = pi/2
```

```
angle1 = 1.5708
```

```
R = so2(angle1,"theta")
```

```
R = so2
    0.0000   -1.0000
    1.0000    0.0000
```

Get the rotation angle from the rotation.

```
angle2 = theta(R)
```

```
angle2 = 1.5708
```

Input Arguments

transformation — Transformation
se2 object | *N*-by-*M* array of se2 objects

Transformation, specified as an se2 object or as an *N*-by-*M* array of se2 objects. *N* is the total number of transformations.

If transformation is a *N*-by-*M* array, the angle argument is the same size and contains an angle for each of the se2 objects specified in the array.

Data Types: single | double

rotation — Rotation
so2 object | *N*-by-*M* array of so2 objects

Rotation, specified as an so2 object or as an *N*-by-*M* array of so2 objects. *N* is the total number of rotations.

If rotation is a *N*-by-*M* array, the angle argument is the same size and contains an angle for each of the so2 objects specified in the array.

Output Arguments

angle — Rotation angle
numeric scalar | *N*-by-*M* matrix

Rotation angle, returned as a numeric scalar for a scalar input and as an *N*-by-*M* matrix for an array input. *N* and *M* are the dimensions of the input rotation or transformation argument. Each element of the matrix is an angle, in radians, and each angle corresponds to a rotation or transformation in the input at the same index location.

The rotation angle is counterclockwise positive when you look along the specified axis toward the origin.

Data Types: single | double

Version History

Introduced in R2023a

See Also

se2 | so2

tform

Extract homogeneous transformation

Syntax

```
transformationMatrix = tform(transformation)
transformationMatrix = tform(rotation)
```

Description

`transformationMatrix = tform(transformation)` extracts the homogeneous transformation matrix `transformationMatrix` that corresponds to the SE(2) or SE(3) transformation `transformation`.

`transformationMatrix = tform(rotation)` creates a homogeneous transformation matrix `transformationMatrix`, with zero translation, that corresponds to the SO(2) or SO(3) rotation `rotation`.

Input Arguments

`transformation` — Transformation
`se2` object | `se3` object | *N*-element array of transformation objects

Transformation, specified as a scalar `se2` object, a scalar `se3` object, or an *N*-element array of transformation objects. *N* is the total number of transformations.

If you specify `transformation` as an array, each element must be of the same type.

`rotation` — Rotation
`so2` object | `so3` object | *N*-element array of rotation objects

Rotation, specified as a scalar `so2` object, a scalar `so3` object, or an *N*-element array of rotation objects. *N* is the total number of rotations.

If you specify `rotation` as an array, each element must be of the same type.

Output Arguments

`transformationMatrix` — Homogeneous transformation matrix
3-by-3-by-*N* array | 4-by-4-by-*N* array

Homogeneous transformation matrix, returned as a 3-by-3-by-*N* array for `se2` and `so2` objects, or a 4-by-4-by-*N* array for `se3` and `so3` objects. *N* is the total number of transformations.

Version History

Introduced in R2022b

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

Functions
`rotm` | `trvec`

Objects
`se2` | `se3` | `so2` | `so3`

times, .*

Element-wise transformation or rotation multiplication

Syntax

```
transformationC = transformationA.*transformationB
rotationC = rotationA.*rotationB
```

Description

`transformationC = transformationA.*transformationB` multiplies transformations element-by-element by multiplying each element of transformation `transformationA` with the corresponding element of transformation `transformationB` and returns the product, transformation `transformationC`.

`rotationC = rotationA.*rotationB` multiplies rotations element-by-element by multiplying each element of rotation `rotationA` with the corresponding element of rotation `rotationB` and returns the product, rotation `rotationC`.

Input Arguments

`transformationA` — First transformation
`se2` object | `se3` object | *N*-element array of transformation objects

First transformation, specified as a scalar `se2` object, a scalar `se3` object, or as an *N*-element array of transformation objects, where *N* is the total number of transformations. If you specify `transformationA` as an array, each element must be of the same type.

Either `transformationA` or `transformationB` must be a scalar transformation object of the same type. For example, if `transformationA` is an array of `se2` objects, `transformationB` must be a scalar `se2` object.

`transformationB` — Last transformation
`se2` object | `se3` object | *N*-element array of transformation objects

Last transformation, specified as a scalar `se2` object, a scalar `se3` object, or as an *N*-element array of transformation objects, where *N* is the total number of transformations. If you specify `transformationB` as an array, each element must be of the same type.

Either `transformationA` or `transformationB` must be a scalar transformation object of the same type. For example, if `transformationA` is an array of `se2` objects, `transformationB` must be a scalar `se2` object.

`rotationA` — First rotation
`so2` object | `so3` object | *N*-element array of rotation objects

First rotation, specified as a scalar `so2` object, a scalar `so3` object, or as an *N*-element array of rotation objects, where *N* is the total number of rotations. If you specify `rotationA` as an array, each element must be of the same type.

Either rotationA or rotationB must be a scalar rotation object of the same type. For example, if rotationA is an array of so2 objects, rotationB must be a scalar so2 object.

rotationB — Last rotation
so2 object | so3 object | *N*-element array of rotation objects

Last rotation, specified as a scalar so2 object, a scalar so3 object, or as an *N*-element array of rotation objects, where *N* is the total number of rotations. If you specify rotationB as an array, each element must be of the same type.

Either rotationA or rotationB must be a scalar rotation object of the same type. For example, if rotationA is an array of se2 objects, rotationB must be a scalar se2 object.

Output Arguments

transformationC — Transformation product
se2 object | se3 object | *N*-element array of transformation objects

Transformation product, returned as a scalar se2 object, a scalar se3 object, or as an *N*-element array of the same transformation type as transformationA and transformationB. *N* is the length of the longer argument between transformationA and transformationB and each row represents the product between transformationA and transformationB.

rotationC — Rotation product
so2 object | so3 object | *N*-element array of rotation objects

Rotation product, returned as a scalar so2 object, a scalar so3 object, or as an *N*-element array of the same rotation type as rotationA and rotationB. *N* is the length of the longer argument between rotationA and rotationB and each row represents the product between rotationA and rotationB.

Version History

Introduced in R2022b

See Also

Functions
mrdivide, / | rdivide, ./ | mtimes, *

Objects
se2 | se3 | so2 | so3

transform

Apply rigid body transformation to points

Syntax

```
tpoints = transform(transformation,points)
tpoints = transform(rotation,points)
tpoints = transform( ___ ,isCol=format)
```

Description

`tpoints = transform(transformation,points)` applies the rigid body transformation `transformation` to the input points `points`, and returns the transformed points `tpoints`.

`tpoints = transform(rotation,points)` applies the rotation `rotation` to the input points `points`, and returns the transformed points `tpoints`.

`tpoints = transform( ___ ,isCol=format)` sets the expected format of the input points `points` to be either column-wise or row-wise by using the logical flag `format` in addition to the input arguments from the previous syntax.

Input Arguments

`transformation` — Transformation
`se2` object | `se3` object | *N*-element array of transformation objects

Transformation, specified as a scalar `se2` object, a scalar `se3` object, or an *N*-element array of transformation objects. *N* is the total number of transformations.

If you specify `transformation` as an array, each element must be of the same type.

`rotation` — Rotation
`so2` object | `so3` object | *N*-element array of rotation objects

Rotation, specified as a scalar `so2` object, a scalar `so3` object, or as an *N*-element array of rotation objects. *N* is the total number of rotations.

If you specify `rotation` as an array, each element must be of the same type.

`points` — Points to transform
N-by-*D*-by-*M* array | *D*-by-*N*-by-*M* array

Points to transform, specified as an *N*-by-*D*-by-*M* array, where:

- *D* is the dimension of the transformation, defined as 2 for 2-D transformations and 3 for 3-D transformations.
- *N* is the total number of input points to transform.
- *M* is the total number of transforms to perform on each point.

For 2-D transformations and rotations, each row specifies a point in the form [*X Y*]. For 3-D transformations and rotations, each row specifies a point in the form [*X Y Z*].

If you specify `format` as `true`, then you must specify `points` as a *D*-by-*N*-by-*M* array, where each column specifies a point.

Data Types: `single` | `double`

`format` — Point format
`false` or `0` (default) | `true` or `1`

Point format, specified as a logical `0` (`false`) or `1` (`true`). If you specify this argument as true, you must specify the points in points as columns. Otherwise, specify points as rows.

Example: `isCol=true`

Data Types: `logical`

Output Arguments

`tpoints` — Transformed points
N-by-*D*-by-*M* array | *D*-by-*N*-by-*M* array

Transformed points, returned as an *N*-by-*D*-by-*M* array, where:

- *D* is the dimension of the transformation, defined as 2 for 2-D transformations and rotations and 3 for 3-D transformations or rotations.
- *N* is the total number of input points to transform.
- *M* is the total number of transforms to perform on each point.

For 2-D transformations and rotations, each row specifies a point in the form [*X Y*]. For 3-D transformations and rotations, each row specifies a point in the form [*X Y Z*].

If you specify `format` as `true`, `tpoints` is returned as a *D*-by-*N*-by-*M* array, where each column specifies a point.

Version History

Introduced in R2022b

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

Functions

`dist` | `interp` | `normalize` | `rotm` | `tform` | `trvec` | `plotTransforms`

Objects

`se2` | `se3` | `so2` | `so3`

trvec

Extract translation vector

Syntax

```
translationVector = trvec(transformation)
```

Description

`translationVector = trvec(transformation)` extracts the translation vector `translationVector` of the SE(2) or SE(3) transformation `transformation`.

Input Arguments

`transformation` — Transformation
`se2` object | `se3` object | *N*-element array of transformation objects

Transformation, specified as a scalar `se2` object, a scalar `se3` object, or an *N*-element array of transformation objects. *N* is the total number of transformations.

If you specify `transformation` as an array, each element must be of the same type.

Output Arguments

`translationVector` — Translation vector
N-by-2 matrix | *N*-by-3 matrix

Translation vector, returned as an *N*-by-2 matrix for `se2` objects or an *N*-by-3 matrix for `se3` objects. *N* is the total number of transformations or rotations, and each row is a translation vector in the form [*X Y*] for 2-D transformations or [*X Y Z*] for 3-D transformations.

Version History

Introduced in R2022b

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

Functions
`dist` | `interp` | `normalize` | `rotm` | `tform` | `transform` | `plotTransforms`

Objects
`se2` | `se3`

xytheta

Convert transformation or rotation to compact 2-D pose representation

Syntax

```
pose = xytheta(transformation)
pose = xytheta(rotation)
```

Description

`pose = xytheta(transformation)` converts a transformation `transformation` to a compact 2-D pose representation `pose`.

`pose = xytheta(rotation)` converts a rotation `rotation` to a compact 2-D pose representation `pose` with no translation.

Examples

Convert SE(2) Transformation to 2-D Compact Pose

Create SE(2) transformation with an *xy*-position of `[2 3]` and a rotation defined by an angle `pi/2`.

```
pose1 = [2 3 pi/2];
T = se2(pose1,"xytheta")
```

```
T = se2
    0.0000   -1.0000    2.0000
    1.0000    0.0000    3.0000
         0         0    1.0000
```

Convert the transformation back into a compact pose.

```
pose2 = xytheta(T)
```

```
pose2 = 1×3

    2.0000    3.0000    1.5708
```

Convert SO(2) Rotation to 2-D Compact Pose

Create SO(2) rotation defined by an angle `pi/2`.

```
angle = pi/2
```

```
angle = 1.5708
```

```
R = so2(angle,"theta")
```

```
R = so2
    0.0000   -1.0000
    1.0000    0.0000
```

Convert the transformation back into a compact pose.

```
pose = xytheta(R)
```

```
pose = 1×3

         0         0    1.5708
```

Input Arguments

transformation — Transformation
se2 object | *N*-element array of se2 objects

Transformation, specified as an se2 object or as an *N*-element array of se2 objects. *N* is the total number of transformations.

Data Types: single | double

rotation — Rotation
so2 object | *N*-element array of so2 objects

Rotation, specified as an so2 object or as an *N*-element array of so2 objects. *N* is the total number of rotations.

Output Arguments

pose — 2-D compact pose
N-by-3 matrix

2-D compact pose, returned as an *N*-by-3 matrix, where each row is of the form [*x y theta*]. *N* is the total number of transformations specified. *x* and *y* are the *xy*-position and *theta* is the rotation about the *z*-axis.

Version History

Introduced in R2023a

See Also

se2 | so2

xyzquat

Convert transformation or rotation to compact 3-D pose representation

Syntax

```
pose = xyzquat(transformation)
pose = xyzquat(rotation)
```

Description

`pose = xyzquat(transformation)` converts a transformation `transformation` to a compact 3-D pose representation `pose`.

`pose = xyzquat(rotation)` converts a rotation `rotation` to a compact 3-D pose representation `pose` with no translation.

Examples

Convert SE(3) Transformation to 3-D Compact Pose

Create SE(3) transformation with an *xyz*-position of `[2 3 1]` and a rotation defined by a numeric quaternion. Use the `eul2quat` function to create the numeric quaternion.

```
trvec = [2 3 1];
quat1 = eul2quat([0 0 deg2rad(30)]);
pose1 = [trvec quat1]
```

```
pose1 = 1×7

    2.0000    3.0000    1.0000    0.9659    0.2588         0         0
```

```
T = se3(pose1,"xyzquat")
```

```
T = se3
    1.0000         0         0    2.0000
         0    0.8660   -0.5000    3.0000
         0    0.5000    0.8660    1.0000
         0         0         0    1.0000
```

Convert the transformation back into a compact pose.

```
pose2 = xyzquat(T)
```

```
pose2 = 1×7

    2.0000    3.0000    1.0000    0.9659    0.2588         0         0
```

Convert SO(3) Rotation to 3-D Compact Pose

Create SO(3) rotation defined by a numeric quaternion. Use the `eul2quat` function to create the numeric quaternion.

```
quat1 = eul2quat([0 0 deg2rad(30)])
```

```
quat1 = 1×4

    0.9659    0.2588         0         0
```

```
R = so3(quat1,"quat")
```

```
R = so3
    1.0000         0         0
         0    0.8660   -0.5000
         0    0.5000    0.8660
```

Convert the rotation into a 3-D compact pose.

```
pose1 = xyzquat(R)
```

```
pose1 = 1×7

         0         0         0    0.9659    0.2588         0         0
```

Input Arguments

`transformation` — Transformation
`se3` object | *N*-element array of `se3` objects

Transformation, specified as an `se3` object or as an *N*-element array of `se3` objects. *N* is the total number of transformations.

`rotation` — Rotation
`so3` object | *N*-element array of `so3` objects

Rotation, specified as an `so3` object or as an *N*-element array of `so3` objects. *N* is the total number of rotations.

Output Arguments

`pose` — 3-D compact pose
M-by-3 matrix

3-D compact pose, returned as an *M*-by-3 matrix, where each row is of the form [*x y z qx qy qz qw*]. *M* is the total number of transformations specified. *x*, *y*, *z* comprise the xyz-position and *qw*, *qx*, *qy*, and *qz* are the quaternion rotations in *w*, *x*, *y*, and *z*, respectively.

Version History

Introduced in R2023a

See Also
se3 | so3

copy

Create copy of particle filter

Syntax

```
b = copy(a)
```

Description

`b = copy(a)` copies each element in the array of handles, `a`, to the new array of handles, `b`.

The `copy` method does not copy dependent properties. MATLAB does not call `copy` recursively on any handles contained in property values. MATLAB also does not call the class constructor or property-set methods during the copy operation.

Input Arguments

a — Object array
handle

Object array, specified as a handle.

Output Arguments

b — Object array containing copies of the objects in a
handle

Object array containing copies of the object in `a`, specified as a handle.

`b` has the same number of elements and is the same size and class of `a`. `b` is the same class as `a`. If `a` is empty, `b` is also empty. If `a` is heterogeneous, `b` is also heterogeneous. If `a` contains deleted handles, then `copy` creates deleted handles of the same class in `b`. Dynamic properties and listeners associated with objects in `a` are not copied to objects in `b`.

Version History

Introduced in R2016a

See Also

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

Topics

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

correct

Adjust state estimate based on sensor measurement

Syntax

```
[stateCorr,stateCov] = correct(pf,measurement)
[stateCorr,stateCov] = correct(pf,measurement,varargin)
```

Description

`[stateCorr,stateCov] = correct(pf,measurement)` calculates the corrected system state and its associated uncertainty covariance based on a sensor `measurement` at the current time step. `correct` uses the `MeasurementLikelihoodFcn` property from the particle filter object, `pf`, as a function to calculate the likelihood of the sensor measurement for each particle. The two inputs to the `MeasurementLikelihoodFcn` function are:

1. `pf` – The `stateEstimatorPF` object, which contains the particles of the current iteration
2. `measurement` – The sensor measurements used to correct the state estimate

The `MeasurementLikelihoodFcn` function then extracts the best state estimate and covariance based on the setting in the `StateEstimationMethod` property.

`[stateCorr,stateCov] = correct(pf,measurement,varargin)` passes all additional arguments in `varargin` to the underlying `MeasurementLikelihoodFcn` after the first three required inputs.

Examples

Particle Filter Prediction and Correction

Create a `stateEstimatorPF` object, and execute a prediction and correction step for state estimation. The particle filter gives a predicted state estimate based on the return value of `StateTransitionFcn`. It then corrects the state based on a given measurement and the return value of `MeasurementLikelihoodFcn`.

Create a particle filter with the default three states.

```
pf = stateEstimatorPF
```

```
pf = 
  stateEstimatorPF with properties:

           NumStateVariables: 3
                NumParticles: 1000
          StateTransitionFcn: @nav.algs.gaussianMotion
    MeasurementLikelihoodFcn: @nav.algs.fullStateMeasurement
     IsStateVariableCircular: [0 0 0]
            ResamplingPolicy: [1x1 resamplingPolicyPF]
            ResamplingMethod: 'multinomial'
       StateEstimationMethod: 'mean'
```

```
       StateOrientation: 'row'
              Particles: [1000x3 double]
                Weights: [1000x1 double]
                  State: 'Use the getStateEstimate function to see the value.'
        StateCovariance: 'Use the getStateEstimate function to see the value.'
```

Specify the mean state estimation method and systematic resampling method.

```
pf.StateEstimationMethod = 'mean';
pf.ResamplingMethod = 'systematic';
```

Initialize the particle filter at state [4 1 9] with unit covariance (`eye(3)`). Use 5000 particles.

```
initialize(pf,5000,[4 1 9],eye(3));
```

Assuming a measurement [4.2 0.9 9], run one predict and one correct step.

```
[statePredicted,stateCov] = predict(pf);
[stateCorrected,stateCov] = correct(pf,[4.2 0.9 9]);
```

Get the best state estimate based on the `StateEstimationMethod` algorithm.

```
stateEst = getStateEstimate(pf)
```

```
stateEst = 1×3

    4.1562    0.9185    9.0202
```

Input Arguments

`pf` — `stateEstimatorPF` object
handle

`stateEstimatorPF` object, specified as a handle. See `stateEstimatorPF` for more information.

`measurement` — Sensor measurements
array

Sensor measurements, specified as an array. This input is passed directly into the `MeasurementLikelihoodFcn` property of `pf`. It is used to calculate the likelihood of the sensor measurement for each particle.

`varargin` — Variable-length input argument list
comma-separated list

Variable-length input argument list, specified as a comma-separated list. This input is passed directly into the `MeasurementLikelihoodFcn` property of `pf`. It is used to calculate the likelihood of the sensor measurement for each particle. When you call:

```
correct(pf,measurement,arg1,arg2)
```

MATLAB essentially calls `measurementLikelihoodFcn` as:

```
measurementLikelihoodFcn(pf,measurement,arg1,arg2)
```

Output Arguments

`stateCorr` — Corrected system state
vector with length `NumStateVariables`

Corrected system state, returned as a row vector with length `NumStateVariables`. The corrected state is calculated based on the `StateEstimationMethod` algorithm and the `MeasurementLikelihoodFcn`.

`stateCov` — Corrected system covariance
N-by-*N* matrix | `[]`

Corrected system variance, returned as an *N*-by-*N* matrix, where *N* is the value of `NumStateVariables` property from `pf`. The corrected state is calculated based on the `StateEstimationMethod` algorithm and the `MeasurementLikelihoodFcn`. If you specify a state estimate method that does not support covariance, then the function returns `stateCov` as `[]`.

Version History

Introduced in R2016a

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`stateEstimatorPF` | `resamplingPolicyPF` | `initialize` | `getStateEstimate` | `predict`

Topics

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

getStateEstimate

Extract best state estimate and covariance from particles

Syntax

```
stateEst = getStateEstimate(pf)
[stateEst,stateCov] = getStateEstimate(pf)
```

Description

`stateEst = getStateEstimate(pf)` returns the best state estimate based on the current set of particles. The estimate is extracted based on the `StateEstimationMethod` property from the `stateEstimatorPF` object, `pf`.

`[stateEst,stateCov] = getStateEstimate(pf)` also returns the covariance around the state estimate. The covariance is a measure of the uncertainty of the state estimate. Not all state estimate methods support covariance output. In this case, `getStateEstimate` returns `stateCov` as `[]`.

Examples

Particle Filter Prediction and Correction

Create a `stateEstimatorPF` object, and execute a prediction and correction step for state estimation. The particle filter gives a predicted state estimate based on the return value of `StateTransitionFcn`. It then corrects the state based on a given measurement and the return value of `MeasurementLikelihoodFcn`.

Create a particle filter with the default three states.

```
pf = stateEstimatorPF
```

```
pf = 
  stateEstimatorPF with properties:

           NumStateVariables: 3
                NumParticles: 1000
          StateTransitionFcn: @nav.algs.gaussianMotion
    MeasurementLikelihoodFcn: @nav.algs.fullStateMeasurement
     IsStateVariableCircular: [0 0 0]
            ResamplingPolicy: [1x1 resamplingPolicyPF]
            ResamplingMethod: 'multinomial'
       StateEstimationMethod: 'mean'
            StateOrientation: 'row'
                   Particles: [1000x3 double]
                     Weights: [1000x1 double]
                       State: 'Use the getStateEstimate function to see the value.'
             StateCovariance: 'Use the getStateEstimate function to see the value.'
```

Specify the mean state estimation method and systematic resampling method.

```
pf.StateEstimationMethod = 'mean';
pf.ResamplingMethod = 'systematic';
```

Initialize the particle filter at state [4 1 9] with unit covariance (`eye(3)`). Use 5000 particles.

```
initialize(pf,5000,[4 1 9],eye(3));
```

Assuming a measurement [4.2 0.9 9], run one predict and one correct step.

```
[statePredicted,stateCov] = predict(pf);
[stateCorrected,stateCov] = correct(pf,[4.2 0.9 9]);
```

Get the best state estimate based on the `StateEstimationMethod` algorithm.

```
stateEst = getStateEstimate(pf)
```

```
stateEst = 1×3

    4.1562    0.9185    9.0202
```

Input Arguments

`pf` — `stateEstimatorPF` object
handle

`stateEstimatorPF` object, specified as a handle. See `stateEstimatorPF` for more information.

Output Arguments

`stateEst` — Best state estimate
vector

Best state estimate, returned as a row vector with length `NumStateVariables`. The estimate is extracted based on the `StateEstimationMethod` algorithm specified in `pf`.

`stateCov` — Corrected system covariance
N-by-*N* matrix | `[]`

Corrected system variance, returned as an *N*-by-*N* matrix, where *N* is the value of `NumStateVariables` property from `pf`. The corrected state is calculated based on the `StateEstimationMethod` algorithm and the `MeasurementLikelihoodFcn`. If you specify a state estimate method that does not support covariance, then the function returns `stateCov` as `[]`.

Version History

Introduced in R2016a

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

`stateEstimatorPF` | `resamplingPolicyPF` | `initialize` | `predict` | `correct`

Topics

“Track a Car-Like Robot Using Particle Filter”
“Particle Filter Parameters”
“Particle Filter Workflow”

initialize

Initialize the state of the particle filter

Syntax

```
initialize(pf,numParticles,mean,covariance)
initialize(pf,numParticles,stateBounds)
initialize( ___ ,Name,Value)
```

Description

`initialize(pf,numParticles,mean,covariance)` initializes the particle filter object, `pf`, with a specified number of particles, `numParticles`. The initial states of the particles in the state space are determined by sampling from the multivariate normal distribution with the specified `mean` and `covariance`.

`initialize(pf,numParticles,stateBounds)` determines the initial location of the particles by sample from the multivariate uniform distribution within the specified `stateBounds`.

`initialize( ___ ,Name,Value)` initializes the particles with additional options specified by one or more `Name,Value` pair arguments.

Examples

Particle Filter Prediction and Correction

Create a `stateEstimatorPF` object, and execute a prediction and correction step for state estimation. The particle filter gives a predicted state estimate based on the return value of `StateTransitionFcn`. It then corrects the state based on a given measurement and the return value of `MeasurementLikelihoodFcn`.

Create a particle filter with the default three states.

```
pf = stateEstimatorPF
```

```
pf = 
  stateEstimatorPF with properties:

           NumStateVariables: 3
                NumParticles: 1000
          StateTransitionFcn: @nav.algs.gaussianMotion
    MeasurementLikelihoodFcn: @nav.algs.fullStateMeasurement
     IsStateVariableCircular: [0 0 0]
            ResamplingPolicy: [1x1 resamplingPolicyPF]
            ResamplingMethod: 'multinomial'
       StateEstimationMethod: 'mean'
            StateOrientation: 'row'
                   Particles: [1000x3 double]
                     Weights: [1000x1 double]
                       State: 'Use the getStateEstimate function to see the value.'
```

```
        StateCovariance: 'Use the getStateEstimate function to see the value.'
```

Specify the mean state estimation method and systematic resampling method.

```
pf.StateEstimationMethod = 'mean';
pf.ResamplingMethod = 'systematic';
```

Initialize the particle filter at state [4 1 9] with unit covariance (`eye(3)`). Use 5000 particles.

```
initialize(pf,5000,[4 1 9],eye(3));
```

Assuming a measurement [4.2 0.9 9], run one predict and one correct step.

```
[statePredicted,stateCov] = predict(pf);
[stateCorrected,stateCov] = correct(pf,[4.2 0.9 9]);
```

Get the best state estimate based on the `StateEstimationMethod` algorithm.

```
stateEst = getStateEstimate(pf)
```

```
stateEst = 1×3

    4.1562    0.9185    9.0202
```

Input Arguments

`pf` — `stateEstimatorPF` object
handle

`stateEstimatorPF` object, specified as a handle. See `stateEstimatorPF` for more information.

`numParticles` — Number of particles used in the filter
scalar

Number of particles used in the filter, specified as a scalar.

`mean` — Mean of particle distribution
vector

Mean of particle distribution, specified as a vector. The `NumStateVariables` property of `pf` is set based on the length of this vector.

`covariance` — Covariance of particle distribution
N-by-N matrix

Covariance of particle distribution, specified as an N-by-N matrix, where N is the value of `NumStateVariables` property from `pf`.

`stateBounds` — Bounds of state variables
n-by-2 matrix

Bounds of state variables, specified as an n-by-2 matrix. The `NumStateVariables` property of `pf` is set based on the value of n. Each row corresponds to the lower and upper limit of the corresponding state variable.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `"CircularVariables",[0 0 1]`

`CircularVariables` — Circular variables
logical vector

Circular variables, specified as a logical vector. Each state variable that uses circular or angular coordinates is indicated with a 1. The length of the vector is equal to the `NumStateVariables` property of `pf`.

Version History

Introduced in R2016a

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`stateEstimatorPF` | `resamplingPolicyPF` | `getStateEstimate` | `predict` | `correct`

Topics

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

predict

Predict state of robot in next time step

Syntax

```
[statePred,stateCov] = predict(pf)
[statePred,stateCov] = predict(pf,varargin)
```

Description

`[statePred,stateCov] = predict(pf)` calculates the predicted system state and its associated uncertainty covariance. `predict` uses the `StateTransitionFcn` property of `stateEstimatorPF` object, `pf`, to evolve the state of all particles. It then extracts the best state estimate and covariance based on the setting in the `StateEstimationMethod` property.

`[statePred,stateCov] = predict(pf,varargin)` passes all additional arguments specified in `varargin` to the underlying `StateTransitionFcn` property of `pf`. The first input to `StateTransitionFcn` is the set of particles from the previous time step, followed by all arguments in `varargin`.

Examples

Particle Filter Prediction and Correction

Create a `stateEstimatorPF` object, and execute a prediction and correction step for state estimation. The particle filter gives a predicted state estimate based on the return value of `StateTransitionFcn`. It then corrects the state based on a given measurement and the return value of `MeasurementLikelihoodFcn`.

Create a particle filter with the default three states.

```
pf = stateEstimatorPF
```

```
pf = 
  stateEstimatorPF with properties:

           NumStateVariables: 3
                NumParticles: 1000
          StateTransitionFcn: @nav.algs.gaussianMotion
    MeasurementLikelihoodFcn: @nav.algs.fullStateMeasurement
     IsStateVariableCircular: [0 0 0]
            ResamplingPolicy: [1x1 resamplingPolicyPF]
            ResamplingMethod: 'multinomial'
       StateEstimationMethod: 'mean'
            StateOrientation: 'row'
                   Particles: [1000x3 double]
                     Weights: [1000x1 double]
                       State: 'Use the getStateEstimate function to see the value.'
             StateCovariance: 'Use the getStateEstimate function to see the value.'
```

Specify the mean state estimation method and systematic resampling method.

```
pf.StateEstimationMethod = 'mean';
pf.ResamplingMethod = 'systematic';
```

Initialize the particle filter at state [4 1 9] with unit covariance (`eye(3)`). Use 5000 particles.

```
initialize(pf,5000,[4 1 9],eye(3));
```

Assuming a measurement [4.2 0.9 9], run one predict and one correct step.

```
[statePredicted,stateCov] = predict(pf);
[stateCorrected,stateCov] = correct(pf,[4.2 0.9 9]);
```

Get the best state estimate based on the `StateEstimationMethod` algorithm.

```
stateEst = getStateEstimate(pf)
```

```
stateEst = 1×3

    4.1562    0.9185    9.0202
```

Input Arguments

pf — stateEstimatorPF object

handle

`stateEstimatorPF` object, specified as a handle. See `stateEstimatorPF` for more information.

varargin — Variable-length input argument list

comma-separated list

Variable-length input argument list, specified as a comma-separated list. This input is passed directly into the `StateTransitionFcn` property of `pf` to evolve the system state for each particle. When you call:

```
predict(pf,arg1,arg2)
```

MATLAB essentially calls the `stateTranstionFcn` as:

```
stateTransitionFcn(pf,prevParticles,arg1,arg2)
```

Output Arguments

statePred — Predicted system state

vector

Predicted system state, returned as a vector with length `NumStateVariables`. The predicted state is calculated based on the `StateEstimationMethod` algorithm.

stateCov — Corrected system covariance

N-by-*N* matrix | [ ]

Corrected system variance, returned as an *N*-by-*N* matrix, where *N* is the value of `NumStateVariables` property from `pf`. The corrected state is calculated based on the

StateEstimationMethod algorithm and the MeasurementLikelihoodFcn. If you specify a state estimate method that does not support covariance, then the function returns stateCov as [].

Version History

Introduced in R2016a

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

stateEstimatorPF | resamplingPolicyPF | initialize | getStateEstimate | correct

Topics

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

getGraph

Graph object representing tree structure

Syntax

```
g = getGraph(frames)
g = getGraph(frames,timestamp)
```

Description

`g = getGraph(frames)` returns a MATLAB `graph` object showing the child-parent relationships between frames at the last timestamp in the `frames` `transformTree` object.

`g = getGraph(frames,timestamp)` returns a MATLAB `graph` object showing the child-parent relationships between frames at the specified timestamp.

Input Arguments

`frames` — Transform tree defining the child-parent frame relationship at given timestamps
`transformTree` object

Transform tree defining the child-parent frame relationship at given timestamps, specified as a `transformTree` object.

`timestamp` — Time for querying the frames
scalar in seconds

Time for querying the frames, specified as a scalar in seconds.

Output Arguments

g — MATLAB graph
`graph` object

MATLAB graph, specified as a `graph` object. This graph reflects the parent-child relationship of the transforms defined in the transform tree object, `frames`.

Version History

Introduced in R2022a

See Also

Objects
`transformTree`

Functions
getTransform | info | removeTransform | show | updateTransform

getTransform

Get relative transform between frames

Syntax

```
tform = getTransform(frames,targetframe,sourceframe)
tform = getTransform(frames,targetframe,sourceframe,timestamp)
```

Description

`tform = getTransform(frames,targetframe,sourceframe)` returns the relative transforms that convert points in the `sourceFrame` coordinate frame to the `targetFrame`. By default, this function uses the last timestamp for both frames specified in `frames`.

`tform = getTransform(frames,targetframe,sourceframe,timestamp)` returns the relative transforms at the given timestamp. If the given time is not specified in the transform tree, `frames`, the function performs interpolation using a constant velocity assumption for linear motion, and spherical linear interpolation (SLERP) for angular motion.

Input Arguments

`frames` — Transform tree defining the child-parent frame relationship at given timestamps
`transformTree` object

Transform tree defining the child-parent frame relationship at given timestamps, specified as a `transformTree` object.

`sourceframe` — Source frame names
string scalar | character vector | string array | cell array character vector

Source frame names specified as a string scalar, character vector, string array, or cell array of character vectors. The source frame is the frame you have coordinates in, and the target frame is the frame you want to convert those coordinates to. Each element of the array corresponds to the same element in `targetframe` and the length matches the *n*-dimension of `tform`.

Data Types: `char` | `string` | `cell`

`targetframe` — Target frame names
string scalar | character vector | string array | cell array character vector

Target frame names specified as a string scalar, character vector, string array, or cell array of character vectors. The source frame is the frame you have coordinates in, and the target frame is the frame you want to convert those coordinates to. Each element of the array corresponds to the same element in `sourceframe` and the length matches the *n*-dimension of `tform`.

Data Types: `char` | `string` | `cell`

`timestamp` — Time for querying the frames
scalar in seconds | vector

Time for querying the frames, specified as a scalar or vector of scalars in seconds. For timestamps specified before the first timestamp in `frames`, the function returns `NaN` values. For timestamps specified after the last timestamp, the most recent (largest timestamp) transformation is returned.

Output Arguments

`tform` — Transformations that converts points from source frames to target frames
4-by-4 homogenous transformation matrix | 4-by-4-by-*n* matrix array

Transformations that converts points from the source frames to the target frames specified as a 4-by-4 transformation matrix or a 4-by-4-by-*n* matrix array. Each matrix in the array corresponds to the same element of `targetframe`, `sourceframe`, and `timestamp`.

Version History

Introduced in R2022a

See Also

Objects
`transformTree`

Functions
`getGraph` | `info` | `removeTransform` | `show` | `updateTransform`

info

List all frame names and stored timestamps

Syntax

```
list = info(frames)
```

Description

`list = info(frames)` returns a structure array with an element for each frame containing the frame name, parent frame, and all stored timestamps.

Input Arguments

`frames` — Transform tree defining the child-parent frame relationship at given timestamps
`transformTree` object

Transform tree defining the child-parent frame relationship at given timestamps, specified as a `transformTree` object.

Output Arguments

`list` — List of frame names, parents, and timestamps
structure array

List of frame names, parents, and timestamps, specified as a structure array. The elements of the structure array are:

- `FrameNames` –– String scalars listing each frame name.
- `ParentNames` –– String scalars listing the parent of each frame. The base frame returns an empty string.
- `Timestamps` –– Vectors of timestamps for each frame. Each vector is padded with `NaN`s based on the MaxNumTransforms property of `frames`.

Version History

Introduced in R2022a

See Also

Objects
`transformTree`

Functions
`getGraph` | `getTransform` | `removeTransform` | `show` | `updateTransform`

removeTransform

Remove frame transform relative to its parent

Syntax

```
removeTransform(frames,framename,timestamp)
removeTransform(frames,framename,timeStart,timeEnd)
```

Description

`removeTransform(frames,framename,timestamp)` removes the frame transforms between the given frame name and their parent frame at the specified timestamps.

`removeTransform(frames,framename,timeStart,timeEnd)` removes all the frame transforms for the given frame name in the time interval, `[timeStart timeEnd]`.

Input Arguments

`frames` — Transform tree defining the child-parent frame relationship at given timestamps
`transformTree` object

Transform tree defining the child-parent frame relationship at given timestamps, specified as a `transformTree` object.

`framename` — Frame name
string scalar | character vector

Frame name with transforms you want to remove, specified as a string scalar or character vector.

Data Types: `char` | `string` | `cell`

`timestamp` — Times for removing transforms
scalar in seconds | vector

Times for removing transforms, specified as a scalar or vector of scalars in seconds. These timestamps must be specified for each of the frame transforms that you want to remove.

`timeStart` — Initial time for removing transforms
scalar in seconds

Initial time for removing transforms, specified as a scalar in seconds. All transforms for the given `framename` are removed from `timeStart` to `timeEnd`.

`timeEnd` — Final time for removing transforms
scalar in seconds

Final time for removing transforms, specified as a scalar in seconds. All transforms for the given `framename` are removed from `timeStart` to `timeEnd`.

Version History

Introduced in R2022a

See Also

Objects
transformTree

Functions
getGraph | getTransform | info | show | updateTransform

show

Show transform tree

Syntax

```
hAx = show(frames)
hAx = show(frames,timestamp)
hAx = show( ___ ,Name,Value)
```

Description

`hAx = show(frames)` displays the transform tree at the last timestamp in the sequence.

`hAx = show(frames,timestamp)` displays the transform tree at the specified timestamp. If the specified time is not specified in the transform tree, `frames`, the function performs interpolation using a constant velocity assumption for linear motion, and spherical linear interpolation (SLERP) for angular motion.

`hAx = show( ___ ,Name,Value)` specifies additional options specified by one or more name-value pair arguments.

Input Arguments

`frames` — Transform tree defining the child-parent frame relationship at given timestamps
`transformTree` object

Transform tree defining the child-parent frame relationship at given timestamps, specified as a `transformTree` object.

`timestamp` — Time for querying the frames
scalar in seconds | vector

Time for querying the frames, specified as a scalar or vector of scalars in seconds. If the given time is not specified in the transform tree, `frames`, the function performs interpolation using a constant velocity assumption for linear motion, and spherical linear interpolation (SLERP) for angular motion. For timestamps specified after the last timestamp, the most recent (largest timestamp) transformation is returned.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `'ShowArrow',true` draws arrows between parent to child frames

`ShowArrow` — Draw arrows from parent to child frames
`false` (default) | `true`

Draw arrows from parent to child frames, specified as `true` or `false`.

Data Types: `logical`

`FrameSizes` — Axis sizes for frames
`struct("root",1)` (default) | structure

Axis sizes for frames, specified as a structure. Specify each frame name as a the field with a scalar for that frame's relative size.

Example: `struct("root",2,"frameA",5)`

Data Types: `struct`

`FrameNames` — Frames to plot
all frames (default) | string scalar | character vector | string array | cell array of character vectors

Frames to plot, specified as a string, character vector, string array, or cell array of character vectors. Use this argument to specify a subset of frame names to display in the figure.

Example: `["Frame1","Frame3","Frame9"]`

Data Types: `char` | `string` | `cell`

`Parent` — Axes on which to plot
`Axes` object

Axes on which to plot, specified as an `Axes` object.

Output Arguments

`hAx` — Axes
`Axes` object

Axes under which the transform tree is shown, returned as an `Axes` object. For more information, see `Axes Properties`.

Version History

Introduced in R2022a

See Also

Objects
`transformTree`

Functions
`getGraph` | `getTransform` | `info` | `removeTransform` | `updateTransform`

updateTransform

Update frame transform relative to its parent

Syntax

```
updateTransform(frames,parentframe,childframe,position,orientation,timestamp)
updateTransform(frames,parentframe,childframe,tform,timestamp)
```

Description

`updateTransform(frames,parentframe,childframe,position,orientation,timestamp)` updates the relative transforms between child frames and their parents with a given position and orientation at the specified time stamps. The position and orientation are given in the parent reference frame.

`updateTransform(frames,parentframe,childframe,tform,timestamp)` updates the relative transforms between child frames and their parents with a given 4-by-4 homogenous transform, `tform`.

Input Arguments

`frames` — Transform tree defining the child-parent frame relationship at given timestamps
`transformTree` object

Transform tree defining the child-parent frame relationship at given timestamps, specified as a `transformTree` object.

`parentframe` — Parent frame names
string scalar | character vector | string array | cell array character vector

Parent frame names specified as a string scalar, character vector, string array, or cell array of character vectors. Transformations specified in `tform` or `position` and `orientation` are relative to the parent frame. Each element of `parentframe` corresponds to the same element in `childframe`.

Data Types: `char` | `string` | `cell`

`childframe` — Child frame names
string scalar | character vector | string array | cell array character vector

Child frame names specified as a string scalar, character vector, string array, or cell array of character vectors. The function attaches the child frame to the parent frame. Transformations specified in `tform` or `position` and `orientation` are relative to the parent frame. Each element of `parentframe` corresponds to the same element in `childframe`.

Data Types: `char` | `string` | `cell`

`position` — Relative position of child frame to parent
three-element [*x* *y* *z*] vector

Relative position of child frame to parent, specified as a three-element `[x y z]` vector. Specify the relative orientation in `orientation`.

`orientation` — Relative orientation of child frame to parent
three-element `[x y z]` vector

Relative orientation of child frame to parent, specified as a three-element `[x y z]` vector. Specify the relative position in `position`.

`tform` — Relative transform of child frame to parent
4-by-4 homogenous transformation matrix

Relative transform of child frame to parent, specified as a 4-by-4 homogenous transformation matrix.

`timestamp` — Time for querying the frames
scalar in seconds | vector

Time for querying the frames, specified as a scalar or vector of scalars in seconds. If the specified time is not specified in the transform tree, `frames`, the function performs interpolation using a constant velocity assumption for linear motion, and spherical linear interpolation (SLERP) for angular motion. For timestamps specified after the last timestamp, the most recent (largest timestamp) transformation is returned.

Version History

Introduced in R2022a

See Also

Objects
`transformTree`

Functions
`getGraph` | `getTransform` | `info` | `removeTransform` | `show`

sample

Sample end-effector poses in world frame

Syntax

```
eePose = sample(goalRegion)
eePose = sample(goalRegion,numSamples)
```

Description

`eePose = sample(goalRegion)` samples an end-effector pose in the world frame as a homogeneous transformation matrix.

The function returns a pose uniformly sampled within the Bounds property relative to the reference frame and applies the following transformations based on the ReferencePose and EndEffectorOffsetPose properties:

```
tSample; % Pose sampled within Bounds
Tw0 = goalRegion.ReferencePose;
TeW = goalRegion.EndEffectorOffsetPose;
eePose = Tw0 * tSample * TeW; % tSample is a pose within the bounds.
```

`eePose = sample(goalRegion,numSamples)` samples multiple poses based on the input `numSamples`. The function returns the end-effector poses as a 3-D array of homogeneous transforms.

Examples

Sample Multiple Poses In A Workspace Goal Region

Sample various poses within the bounds of a workspace goal region for a manipulator arm. Some end-effector poses may not be desirable due to the positioning of the arm bodies and obstacles in the scene. The `workspaceGoalRegion` object defines the bounds on the *xyz*-position and *zyx* Euler orientation of the robot end effector. The `sample` object function uniformly samples random poses within the bounds. Find configurations that achieve these end-effector poses and determine the best by visualization.

Load an existing robot model as a `rigidBodyTree` object.

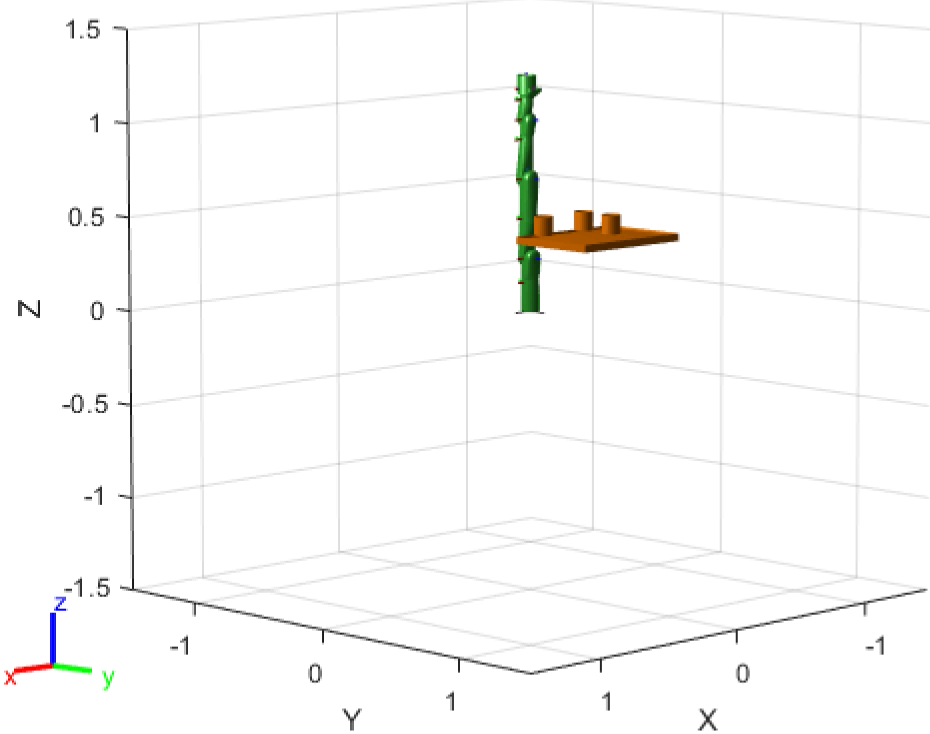
```
robot = loadrobot("kinovaGen3","DataFormat","row");
show(robot,"Collisions","on","Visuals","off");
```

Add a can as a `collisionCylinder` object to the robot arm.

```
can = collisionCylinder(0.05, 0.1);
can.Pose = trvec2tform([0.2, 0.3, 0.5]);

addCollision(robot.Bodies{end},"cylinder", [0.05, 0.1], trvec2tform([0, 0, 0.02]));
```

The goal of this example is to place this can on a table with other cans. Add the table and other cans to the environment by creating a cell array of collision objects. Show the entire `env` cell array.

```
table = collisionBox(0.7, 0.5, 0.04);
table.Pose = trvec2tform([0, 0.5, 0.43]);
env = {can, copy(can), copy(can), table};
env{2}.Pose = trvec2tform([-0.1, 0.3, 0.5]);
env{3}.Pose = trvec2tform([-0.1, 0.5, 0.5]);

hold on
for i = 1: length(env)
    show(env{i})
end
show(robot,homeConfiguration(robot),"Collisions","on","Visuals","off");
```



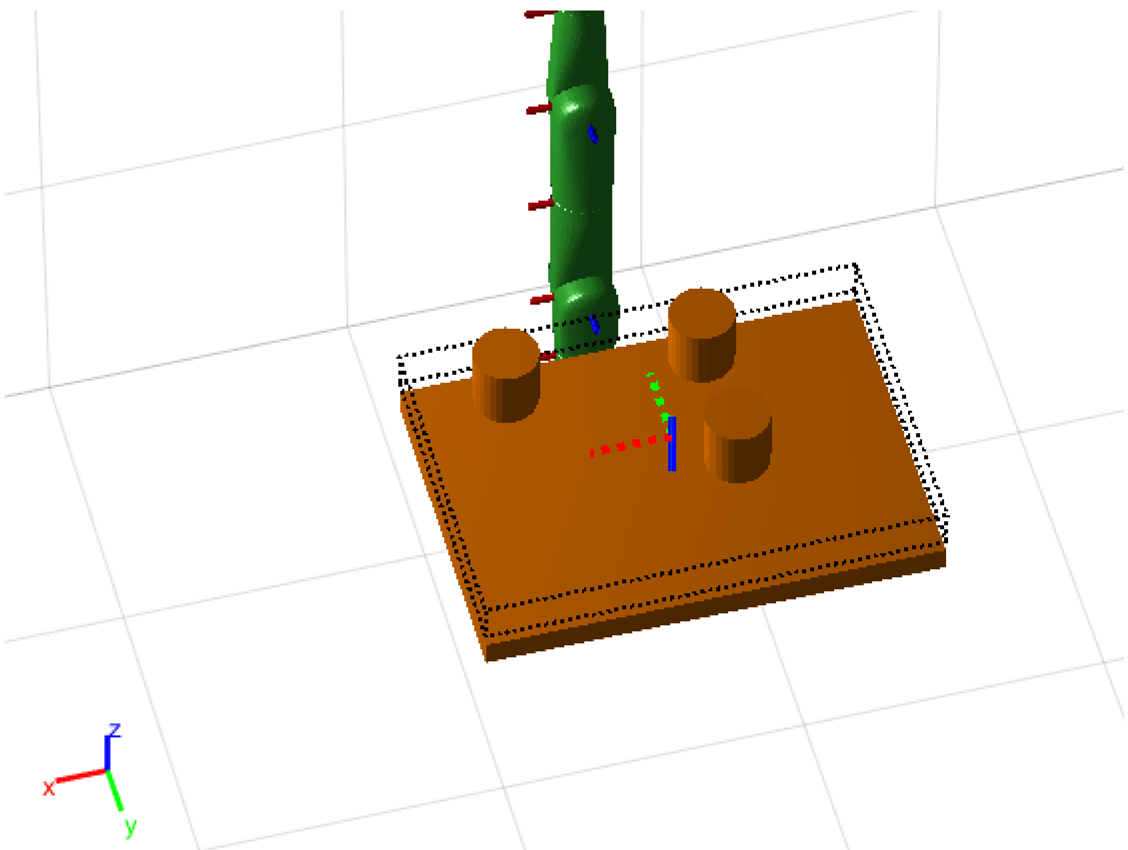
Define Goal Region

Create a workspace goal region using the end-effector body name of the robot.

Define the goal region parameters for your workspace. The goal region includes a reference pose, *xyz*-position bounds, and orientation limits on the *zyx* Euler angles. This example specifies *xyz* bounds within the table dimensions and fixes rotation to a small range in the *y*- and *x*-axis.

```
tableRegion = workspaceGoalRegion("EndEffector_Link",...
    "ReferencePose",table.Pose);
tableRegion.EndEffectorOffsetPose(1:3,1:3) = eul2rotm([0, 0, pi]);
tableRegion.EndEffectorOffsetPose(3, end) = 0.1;

tableRegion.Bounds = ...
    [-table.X/2, table.X/2; % X Bounds
    -table.Y/2, table.Y/2;  % Y Bounds
```

```
    0.04, 0.10;            % Z Bounds
    -pi, pi;               % Rotation about Z-axis
    -0.01, 0.01;           % Y-Axis
    -0.01, 0.01;];         % X-Axis

show(tableRegion);
view(165,50)
camzoom(3.5)
```



Sample Poses

Uniformly sample poses within the table region using the `sample` object function. In this example, set the `rng` seed to get repeatable results. Create vectors for storing valid and invalid poses.

```
rng(0)
poses = sample(tableRegion,10);
validPoses = [];
invalidPoses = [];
```

Check for Collisions

To find configurations for those poses, create an inverse kinematics (IK) solver.

```
ik = inverseKinematics('RigidBodyTree',robot);
config = cell(10);
```

Test the sampled poses by iterating through the sampled poses, solving for configurations using IK, and checking for collisions. Show the valid configurations.

```
for i = 1:length(poses)
    % Solve for robot configuraiton using IK.
    config{i} = ik("EndEffector_Link",poses(:,:,i),ones(6,1),homeConfiguration(robot));
    % Check for collisions.
    isColliding = checkCollision(robot,config{i},env,SkippedSelfCollisions="parent");

    if ~isColliding % If not in collision, show robot configuration and save valid pose.
        show(robot,config{i},"PreservePlot",false,"Collisions","on","Visuals","off");
        drawnow
        validPoses = [validPoses; i];
    else
        invalidPoses = [invalidPoses; i];
    end

end
```

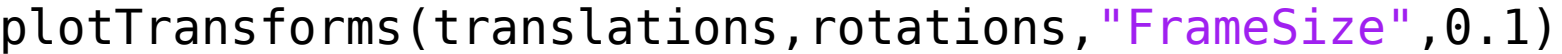
```
disp(string(validPoses'))
```

```
    "3"    "5"    "7"    "10"
```

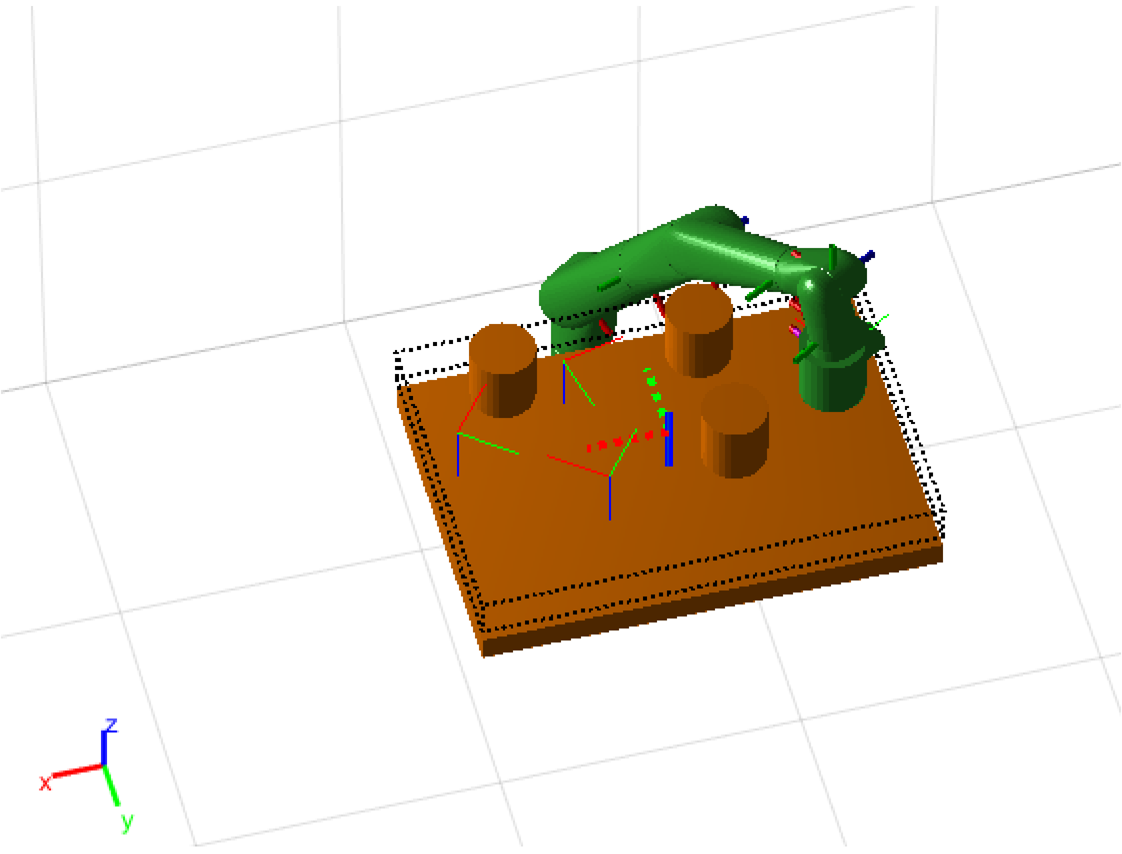
Visualize A Singe Valid Pose

Plot all valid poses as transforms. The final valid configuration from checking collisions is still visible in the figure.

```
translations = tform2trvec(poses(:,:,validPoses));
rotations = tform2quat(poses(:,:,validPoses));
```

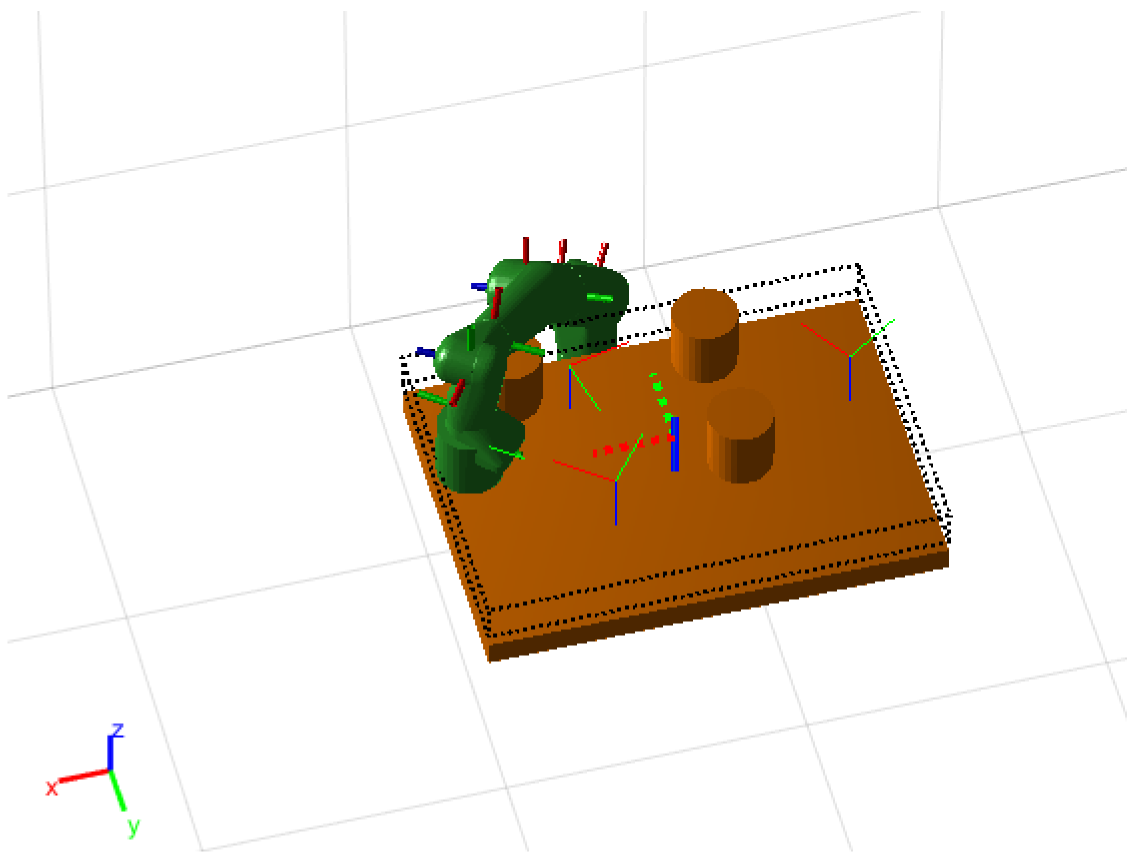


```
plotTransforms(translations,rotations,"FrameSize",0.1)
```



Show a valid configuration from the list. Change the index in `validPoses` to look at different poses. Call `hold off` to stop preserving figure elements. To manually inspect poses and configurations, comment out the final line when running.

```
poseIndex = validPoses(1);
show(robot,config{poseIndex},"PreservePlot",false,"Collisions","on","Visuals","off");
hold off
```



Input Arguments

goalRegion — Workspace goal region
workspaceGoalRegion object

Workspace goal region, specified as a workspaceGoalRegion object.

numSamples — Number of samples
positive integer

Number of samples, specified as a positive integer

Output Arguments

eePose — Poses sampled within workspace bounds
4-by-4 homogeneous transform matrix | four-by-four-by-*n* array

Poses sampled within the workspace bounds in the world frame, returned as a four-by-four homogeneous transformation matrix or 4-by-4-by-*n* array, where *n* is the number of samples numSamples.

The function returns a pose uniformly sampled within the Bounds property relative to the reference frame and applies the following transformations based on the ReferencePose and EndEffectorOffsetPose properties:

```
tSample = rand(6,2);
Tw0 = goalRegion.ReferencePose;
TeW = goalRegion.EndEffectorOffsetPose;
eePose = Tw0 * tSample * TeW;
```

Data Types: `double`

Version History

Introduced in R2021a

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`workspaceGoalRegion` | `manipulatorRRT` | `show`

show

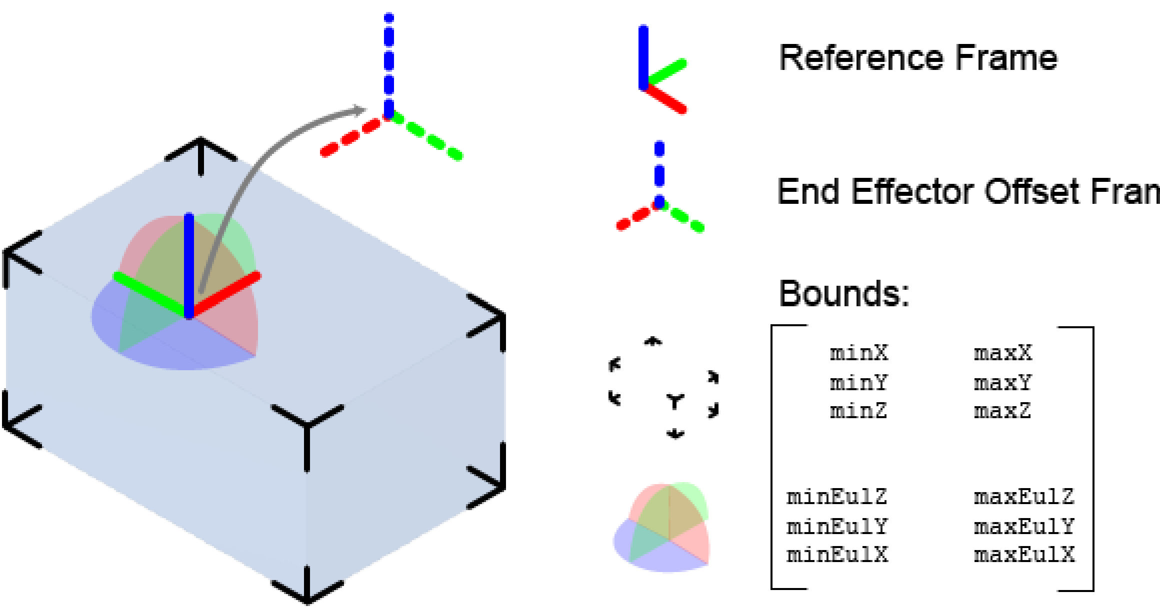
Visualize workspace bounds, reference frame, and offset frame

Syntax

```
show(goalRegion)
show(goalRegion,"Parent",axesHandle)
ax = show( ___ )
```

Description

`show(goalRegion)` plots the position and orientation bounds of the workspace goal region. The function also displays the reference frame and end-effector offset frame.



`show(goalRegion,"Parent",axesHandle)` specifies the parent axes on which to plot the workspace goal region.

`ax = show( ___ )` returns the axes handle that contains the workspace goal region plot using the input arguments from previous syntaxes.

Examples

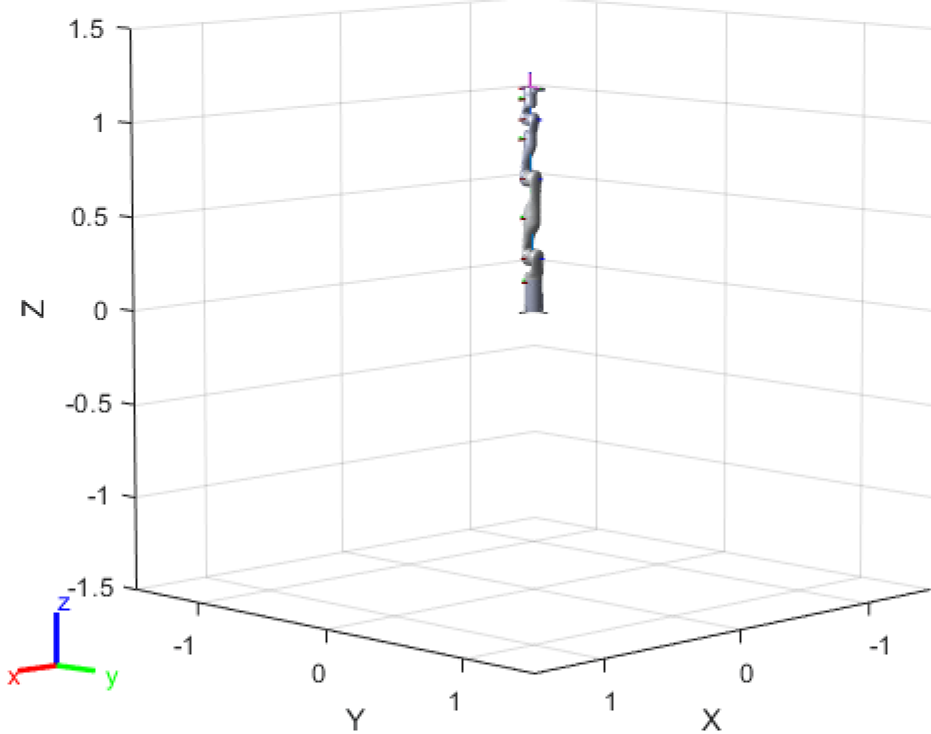
Plan Path To Workspace Goal Region

Specify a goal region in your workspace and plan a path within those bounds. The `workspaceGoalRegion` object defines the bounds on the *xyz*-position and *zyx* Euler orientation of

the robot end effector. The `manipulatorRRT` object plans a path based on that goal region and samples random poses within the bounds.

Load an existing robot model as a `rigidBodyTree` object.

```
robot = loadrobot("kinovaGen3", "DataFormat", "row");
ax = show(robot);
```



Create Path Planner

Create a rapidly-exploring random tree (RRT) path planner for the robot. This example uses an empty environment, but this workflow also works well with cluttered environments. You can add collision objects to the environment like the `collisionBox` or `collisionMesh` object.

```
planner = manipulatorRRT(robot,{});
planner.SkippedSelfCollisions="parent";
```

Define Goal Region

Create a workspace goal region using the end-effector body name of the robot.

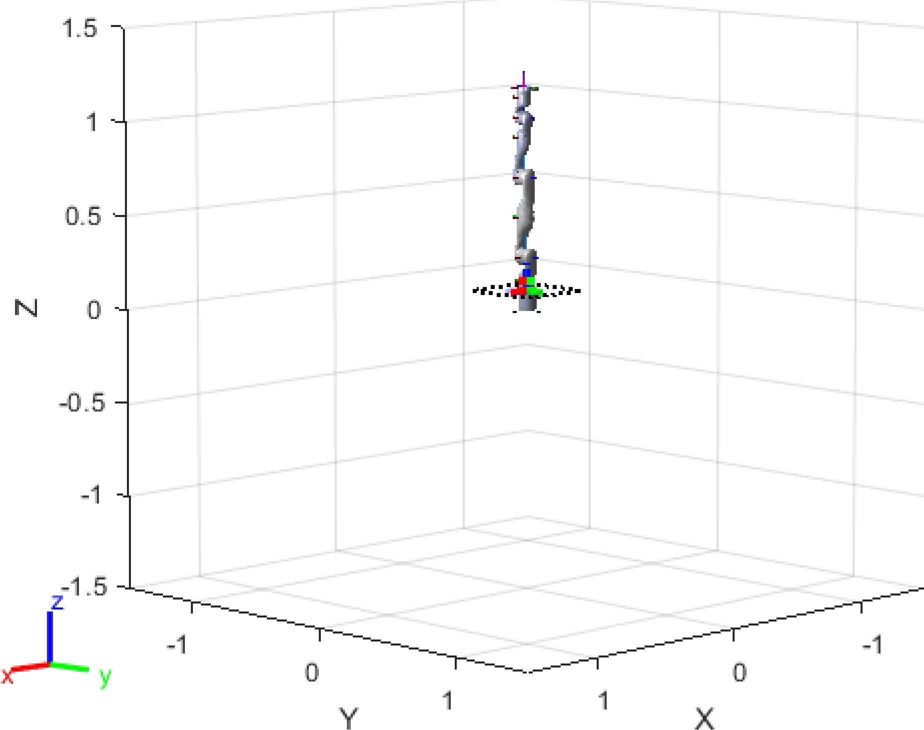
Define the goal region parameters for your workspace. The goal region includes a reference pose, *xyz*-position bounds, and orientation limits on the *zyx* Euler angles. This example specifies bounds on the *xy*-plane in meters and allows rotation about the *z*-axis in radians.

```
goalRegion = workspaceGoalRegion(robot.BodyNames{end});
goalRegion.ReferencePose = trvec2tform([0.5 0.5 0.2]);
```

```
goalRegion.Bounds(1, :) = [-0.2 0.2];     % X Bounds
goalRegion.Bounds(2, :) = [-0.2 0.2];     % Y Bounds
goalRegion.Bounds(4, :) = [-pi/2 pi/2];  % Rotation about the Z-axis
```

You can also apply a fixed offset to all poses sampled within the region. This offset can account for grasping tools or variations in dimensions within your workspace. For this example, apply a fixed transformation that places the end effector 5 cm above the workspace.

```
goalRegion.EndEffectorOffsetPose = trvec2tform([0 0 0.05]);
hold on
show(goalRegion);
```



Plan Path To Goal Region

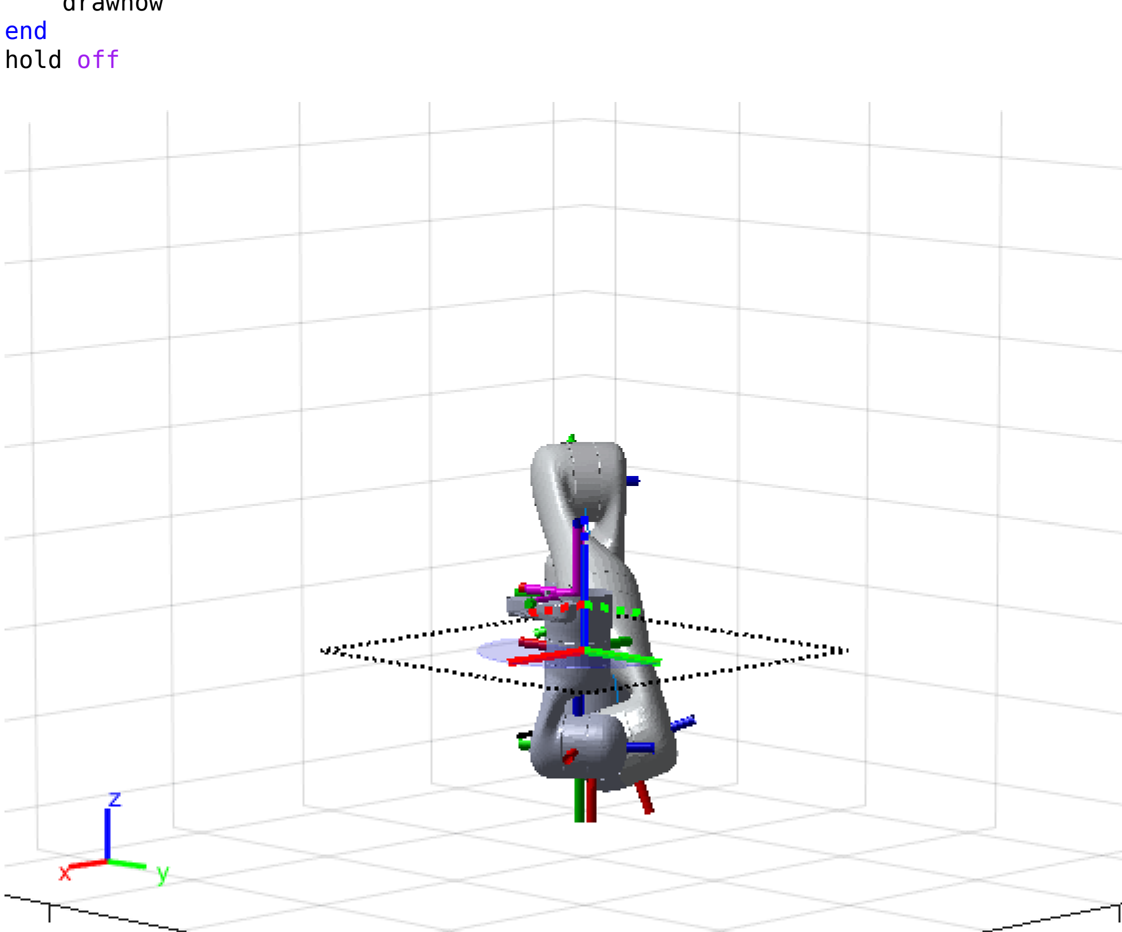
Plan a path to the goal region from the robot's home configuration. Due to the randomness in the RRT algorithm, this example sets the `rng` seed to ensure repeatable results.

```
rng(0)
path = plan(planner,homeConfiguration(robot),goalRegion);
```

Show the robot executing the path. To visualize a more realistic path, interpolate points between path configurations.

```
interpConfigurations = interpolate(planner,path,5);

for i = 1 : size(interpConfigurations)
    show(robot,interpConfigurations(i,:),"PreservePlot",false);
    set(ax,'ZLim',[-0.05 0.75],'YLim',[-0.05 1],'XLim',[-0.05 1],...
```

```
        'CameraViewAngle',5)

    drawnow
end
hold off
```



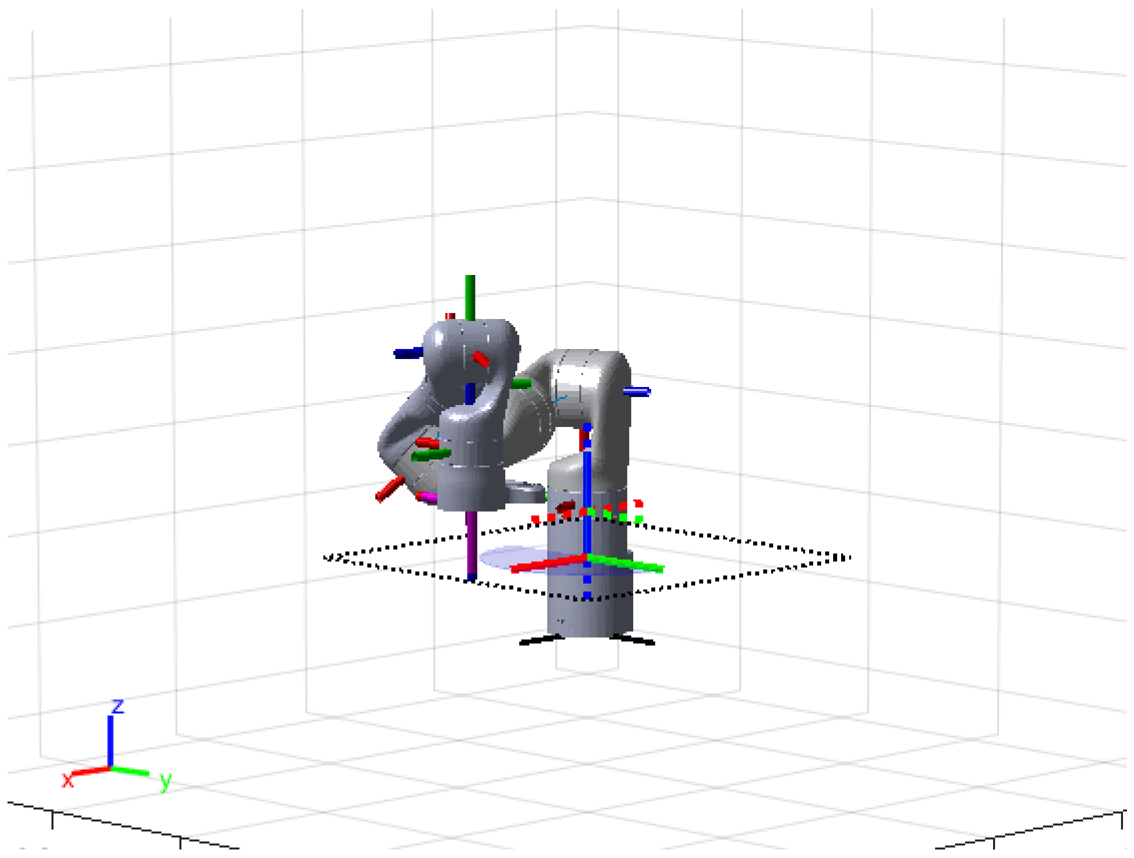
Adjust End-Effector Pose

Notice that the robot arm approaches the workspace from the bottom. To flip the orientation of the final position, add a `pi` rotation to the Y-axis for the reference pose.

```
goalRegion.EndEffectorOffsetPose = ...
    goalRegion.EndEffectorOffsetPose*eul2tform([0 pi 0],"ZYX");
```

Replan the path and visualize the robot motion again. The robot now approaches from the top.

```
hold on
show(goalRegion);
path = plan(planner,homeConfiguration(robot),goalRegion);

interpConfigurations = interpolate(planner,path,5);

for i = 1 : size(interpConfigurations)
    show(robot, interpConfigurations(i, :),"PreservePlot",false);
    set(ax,'ZLim',[-0.05 0.75],'YLim',[-0.05 1],'XLim',[-0.05 1])
    drawnow;
end
hold off
```



Input Arguments

goalRegion — Workspace goal region
workspaceGoalRegion object

Workspace goal region, specified as a `workspaceGoalRegion` object.

Output Arguments

ax — Axes that contains the workspace goal region
Axes object

Axes that contains the workspace goal region, returned as an `axes` object.

Version History

Introduced in R2021a

See Also

`workspaceGoalRegion` | `manipulatorRRT` | `sample`

lookupPose

Obtain pose information for certain time

Syntax

```
[position,orientation,velocity,acceleration,angularVelocity] = lookupPose(
traj,sampleTimes)
```

Description

`[position,orientation,velocity,acceleration,angularVelocity] = lookupPose(traj,sampleTimes)` returns the pose information of the waypoint trajectory at the specified sample times. If any sample time is beyond the duration of the trajectory, the corresponding pose information is returned as `NaN`.

Input Arguments

`traj` — Waypoint trajectory
`waypointTrajectory` object

Waypoint trajectory, specified as a `waypointTrajectory` object.

`sampleTimes` — Sample times
M-element vector of nonnegative scalar

Sample times in seconds, specified as an *M*-element vector of nonnegative scalars.

Output Arguments

`position` — Position in local navigation coordinate system (m)
M-by-3 matrix

Position in the local navigation coordinate system in meters, returned as an *M*-by-3 matrix.

M is specified by the `sampleTimes` input.

Data Types: `double`

`orientation` — Orientation in local navigation coordinate system
M-element quaternion column vector | 3-by-3-by-*M* real array

Orientation in the local navigation coordinate system, returned as an *M*-by-1 `quaternion` column vector or a 3-by-3-by-*M* real array.

Each quaternion or 3-by-3 rotation matrix is a frame rotation from the local navigation coordinate system to the current body coordinate system.

M is specified by the `sampleTimes` input.

Data Types: `double`

`velocity` — Velocity in local navigation coordinate system (m/s)
M-by-3 matrix

Velocity in the local navigation coordinate system in meters per second, returned as an *M*-by-3 matrix.

M is specified by the `sampleTimes` input.

Data Types: `double`

`acceleration` — Acceleration in local navigation coordinate system (m/s^2)
M-by-3 matrix

Acceleration in the local navigation coordinate system in meters per second squared, returned as an *M*-by-3 matrix.

M is specified by the `sampleTimes` input.

Data Types: `double`

`angularVelocity` — Angular velocity in local navigation coordinate system (rad/s)
M-by-3 matrix

Angular velocity in the local navigation coordinate system in radians per second, returned as an *M*-by-3 matrix.

M is specified by the `sampleTimes` input.

Data Types: `double`

Version History

Introduced in R2022a

See Also

Objects
`waypointTrajectory`

Functions
`waypointInfo` | `perturbations` | `perturb`

waypointInfo

Get waypoint information table

Syntax

```
trajectoryInfo = waypointInfo(trajectory)
```

Description

`trajectoryInfo = waypointInfo(trajectory)` returns a table of waypoints, times of arrival, velocities, and orientation for the `trajectory` System object.

Input Arguments

`trajectory` — Object of `waypointTrajectory`
object

Object of the `waypointTrajectory` System object.

Output Arguments

`trajectoryInfo` — Trajectory information
table

Trajectory information, returned as a table with variables corresponding to set creation properties: Waypoints, TimeOfArrival, Velocities, and Orientation.

The trajectory information table always has variables `Waypoints` and `TimeOfArrival`. If the `Velocities` property is set during construction, the trajectory information table additionally returns velocities. If the `Orientation` property is set during construction, the trajectory information table additionally returns orientation.

Version History

Introduced in R2022a

See Also

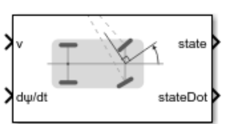
Objects
`waypointTrajectory`

Functions
`lookupPose` | `perturbations` | `perturb`

Blocks

Ackermann Kinematic Model

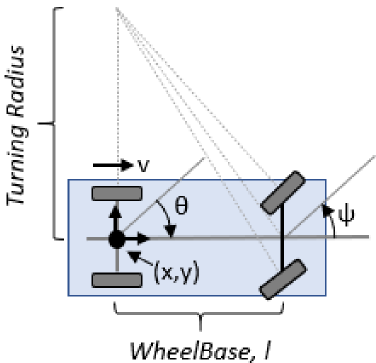
Car-like vehicle motion using Ackermann kinematic model



Libraries:
Robotics System Toolbox / Mobile Robot Algorithms

Description

The Ackermann Kinematic Model block creates a car-like vehicle model that uses Ackermann steering. This model represents a vehicle with two axles separated by the distance, `Wheel base`. The state of the vehicle is defined as a four-element vector, [x y θ ψ], with an global xy-position, vehicle heading, θ, and steering angle, ψ. The vehicle heading and xy-position are defined at the center of the rear axle. Angles are specified in radians and the global positions are specified in meters. The steering input for the vehicle is given as `dψ/dt`, in radians per second.



Ports

Input

v — Vehicle speed
numeric scalar

Vehicle speed, specified in meters per second.

dψ/dt — Steering angular velocity
numeric scalar

Steering angular velocity of the vehicle, specified in radians per second.

Output

state — State of vehicle
four-element vector

Current xy-position, orientation, and steering angle, specified as [x y θ ψ], in meters and radians.

stateDot — Derivatives of `state` output
four-element vector

The linear and angular velocities of the vehicle, specified as a [*xDot yDot thetaDot psiDot*] vector in meters per second and radians per second. The linear and angular velocities are calculated by taking the time derivatives of the `state` output.

Parameters

Wheel base — Distance between front and rear axles
`1` (default) | positive numeric scalar

The wheel base refers to the distance between the front and rear vehicle axles, specified in meters.

Vehicle speed range — Minimum and maximum vehicle speeds
`[-Inf Inf]` (default) | two-element vector

The wheel speed range is a two-element vector that provides the minimum and maximum vehicle wheel speeds, [*MinSpeed MaxSpeed*], specified in radians per second.

Maximum steering angle — Distance between front and rear axles
`pi/4` (default) | positive numeric scalar

The maximum steering angle, refers to the maximum amount the vehicle can be steered to the right or left, specified in radians. The default value is `pi/4`.

Initial state — Initial state of vehicle
`[0;0;0;0]` (default) | four-element vector

The initial *x*-, *y*-position, heading angle, *theta*, and steering angle, *psi*, of the vehicle.

Simulate using — Type of simulation to run
`Interpreted execution` (default) | `Code generation`

- `Interpreted execution` — Simulate model using the MATLAB interpreter. For more information, see "Simulation Modes" (Simulink).
- `Code generation` — Simulate model using generated C code. The first time you run a simulation, Simulink generates C code for the block. The C code is reused for subsequent simulations, as long as the model does not change.

Tunable: No

Version History

Introduced in R2019b

References

[1] Lynch, Kevin M., and Frank C. Park. *Modern Robotics: Mechanics, Planning, and Control*. 1st ed. Cambridge, MA: Cambridge University Press, 2017.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

See Also

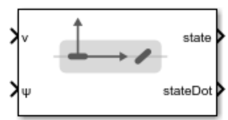
Blocks
Bicycle Kinematic Model | Differential Drive Kinematic Model | Unicycle Kinematic Model

Classes
`ackermannKinematics`

Topics
"Mobile Robot Kinematics Equations"

Bicycle Kinematic Model

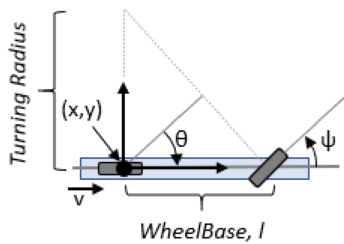
Compute car-like vehicle motion using bicycle kinematic model



Libraries:
Robotics System Toolbox / Mobile Robot Algorithms

Description

The Bicycle Kinematic Model block creates a bicycle vehicle model to simulate simplified car-like vehicle dynamics. This model represents a vehicle with two axles defined by the length between the axles, `Wheel base`. The front wheel can be turned with steering angle `psi`. The vehicle heading `theta` is defined at the center of the rear axle.



Ports

Input

v — Vehicle speed
numeric scalar

Vehicle speed, specified in meters per second.

psi — Steering angle
numeric scalar

Steering angle of the vehicle, specified in radians.

Dependencies

To enable this port, set the `Vehicle inputs` parameter to `Vehicle Speed & Steering Angle`.

omega — Steering angular velocity
numeric scalar

Angular velocity of the vehicle, specified in radians per second. A positive value steers the vehicle left and negative values steer the vehicle right.

Dependencies

To enable this port, set the `Vehicle inputs` parameter to `Vehicle Speed & Heading Angular Velocity`.

Output

state — Pose of vehicle
three-element vector

Current *xy*-position and orientation of the vehicle, specified as a *[x y theta]* vector in meters and radians.

stateDot — Derivatives of `state` output
three-element vector

The linear and angular velocities of the vehicle, specified as a *[xDot yDot thetaDot]* vector in meters per second and radians per second. The linear and angular velocities are calculated by taking the derivative of the `state` output.

Parameters

Vehicle inputs — Type of speed and directional inputs for vehicle
`Vehicle Speed & Steering Angle` (default) | `Vehicle Speed & Heading Angular Velocity`

Type of speed and directional inputs for vehicle, specified as one of these options:

- `Vehicle Speed & Steering Angle` — Vehicle speed in meters per second with a steering angle in radians.
- `Vehicle Speed & Heading Angular Velocity` — Vehicle speed in meters per second with a heading angular velocity in radians per second.

Wheel base — Distance between front and rear axles
`1` (default) | positive numeric scalar

The wheel base refers to the distance between the front and rear vehicle axles, specified in meters.

Vehicle speed range — Minimum and maximum vehicle speeds
`[-Inf Inf]` (default) | two-element vector

The wheel speed range is a two-element vector that provides the minimum and maximum vehicle wheel speeds, [*MinSpeed MaxSpeed*], specified in radians per second.

Maximum steering angle — Max turning radius
`pi/4` (default) | numeric scalar

The maximum steering angle, refers to the maximum amount the vehicle can be steered to the right or left, specified in radians. This property is used to validate the user-provided state input.

If `Maximum steering angle` is set to `pi/2`, it results in a minimum turning radius of zero meters.

Initial state — Initial pose of vehicle
`[0;0;0]` (default) | three-element vector

The initial *x*-, *y*-position and orientation, *theta*, of the vehicle.

Simulate using — Type of simulation to run
`Interpreted execution` (default) | `Code generation`

- `Interpreted execution` — Simulate model using the MATLAB interpreter. For more information, see "Simulation Modes" (Simulink).
- `Code generation` — Simulate model using generated C code. The first time you run a simulation, Simulink generates C code for the block. The C code is reused for subsequent simulations, as long as the model does not change.

Tunable: No

Version History

Introduced in R2019b

References

[1] Lynch, Kevin M., and Frank C. Park. *Modern Robotics: Mechanics, Planning, and Control* 1st ed. Cambridge, MA: Cambridge University Press, 2017.

[2] Corke, Peter I. *Robotics, Vision and Control: Fundamental Algorithms in MATLAB*. Springer, 2011.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

See Also

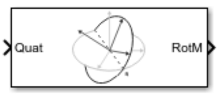
Blocks
Ackermann Kinematic Model | Differential Drive Kinematic Model | Unicycle Kinematic Model

Classes
`bicycleKinematics`

Topics
"Simulate Different Kinematic Models for Mobile Robots"
"Mobile Robot Kinematics Equations"

Coordinate Transformation Conversion

Convert to a specified coordinate transformation representation



Libraries:
Robotics System Toolbox / Utilities
Navigation Toolbox / Utilities
ROS Toolbox / Utilities
UAV Toolbox / Utilities

Description

The Coordinate Transformation Conversion block converts a coordinate transformation from the input representation to a specified output representation. The input and output representations use the following forms:

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

All vectors must be **column vectors**.

To accommodate representations that only contain position or orientation information (`TrVec` or `Eul`, for example), you can specify two inputs or outputs to handle all transformation information. When you select the Homogeneous Transformation as an input or output, an optional `Show TrVec input/output port` parameter can be selected on the block mask to toggle the multiple ports.

Ports

Input

Input transformation — Coordinate transformation
column vector | 3-by-3 matrix | 4-by-4 matrix

Input transformation, specified as a coordinate transformation. The following representations are supported:

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

All vectors must be **column vectors**.

To accommodate representations that only contain position or orientation information (`TrVec` or `Eul`, for example), you can specify two inputs or outputs to handle all transformation information. When you select the Homogeneous Transformation as an input or output, an optional `Show TrVec input/output port` parameter can be selected on the block mask to toggle the multiple ports.

TrVec — Translation vector
3-element column vector

Translation vector, specified as a 3-element column vector, `[x y z]`, which corresponds to a translation in the *x*, *y*, and *z* axes respectively. This port can be used to input or output the translation information separately from the rotation vector.

Dependencies

You must select Homogeneous Transformation (`TForm`) for the opposite transformation port to get the option to show the additional `TrVec` port. Enable the port by clicking `Show TrVec input/output port`.

Output Arguments

Output transformation — Coordinate transformation
column vector | 3-by-3 matrix | 4-by-4 matrix

Output transformation, returned as a coordinate transformation with the specified representation. The following representations are supported:

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

To accommodate representations that only contain position or orientation information (`TrVec` or `Eul`, for example), you can specify two inputs or outputs to handle all transformation information. When you select the Homogeneous Transformation as an input or output, an optional `Show TrVec input/output port` parameter can be selected on the block mask to toggle the multiple ports.

TrVec — Translation vector
three-element column vector

Translation vector, returned as a three-element column vector, `[x y z]`, which corresponds to a translation in the *x*, *y*, and *z* axes respectively. This port can be used to input or output the translation information separately from the rotation vector.

Dependencies

You must select Homogeneous Transformation (`TForm`) for the opposite transformation port to get the option to show the additional `TrVec` port. Enable the port by clicking `Show TrVec input/output port`.

Parameters

Representation — Input or output representation
`Axis-Angle` | `Euler Angles` | `Homogeneous Transformation` | `Rotation Matrix` | `Translation Vector` | `Quaternion`

Select the representation for both the input and output port for the block. If you are using a transformation with only orientation information, you can also select the `Show TrVec input/output port` when converting to or from a homogeneous transformation.

Axis rotation sequence — Order of Euler angle axis rotations
`ZYX` (default) | `ZYZ` | `XYZ`

Order of the Euler angle axis rotations, specified as `ZYX`, `ZYZ`, or `XYZ`. The order of the angles in the input or output port `Eul` must match this rotation sequence. The default order `ZYX` specifies an orientation by:

- Rotating about the initial *z*-axis
- Rotating about the intermediate *y*-axis
- Rotating about the second intermediate *x*-axis

Dependencies

You must select `Euler Angles` for the `Representation` input or output parameter. The axis rotation sequence only applies to Euler angle rotations.

Show TrVec input/output port — Toggle `TrVec` port
off (default) | on

Toggle the `TrVec` input or output port when you want to specify or receive a separate translation vector for position information along with an orientation representation.

Dependencies

You must select Homogeneous Transformation (`TForm`) for the opposite transformation port to get the option to show the additional `TrVec` port.

Simulate using — Type of simulation to run
`Interpreted execution` (default) | `Code generation`

- `Interpreted execution` — Simulate model using the MATLAB interpreter. This option shortens startup time but has a slower simulation speed than `Code generation`. In this mode, you can debug the source code of the block.
- `Code generation` — Simulate model using generated C code. The first time you run a simulation, Simulink generates C code for the block. The C code is reused for subsequent simulations, as long as the model does not change. This option requires additional startup time, but the speed of the subsequent simulations is comparable to `Interpreted execution`.

Tunable: No

Version History

Introduced in R2017b

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

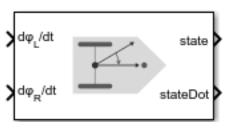
`axang2quat` | `eul2tform` | `trvec2tform`

Topics

"Coordinate Transformations in Robotics"

Differential Drive Kinematic Model

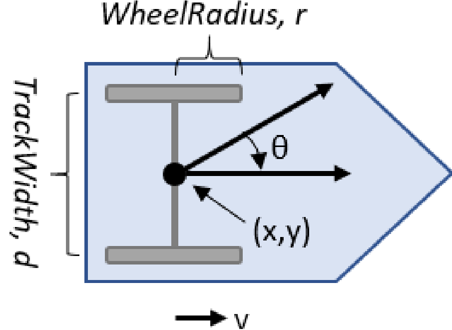
Compute vehicle motion using differential drive kinematic model



Libraries:
Robotics System Toolbox / Mobile Robot Algorithms

Description

The Differential Drive Kinematic Model block creates a differential-drive vehicle model to simulate simplified vehicle dynamics. This model approximates a vehicle with a single fixed axle and wheels separated by a specified track width `Track width`. Each of the wheels can be driven independently using speed inputs, $d\phi_L/dt$ and $d\phi_R/dt$, for the left and right wheels respectively. Vehicle speed and heading is defined from the axle center.



Ports

Input

$d\phi_L$/dt — Left wheel speed
numeric scalar

Left wheel speed of the vehicle, specified in radians per second.

Dependencies

To enable this port, set the `Vehicle inputs` parameter to `Wheel Speeds`.

$d\phi_R$/dt — Right wheel speed
numeric scalar

Right wheel speed of the vehicle, specified in radians per second.

Dependencies

To enable this port, set the `Vehicle inputs` parameter to `Wheel Speeds`.

v — Vehicle speed
numeric scalar

Vehicle speed, specified in meters per second.

Dependencies

To enable this port, set the `Vehicle inputs` parameter to `Vehicle Speed & Heading Angular Velocity`.

ω — Angular velocity of vehicle
numeric scalar

Angular velocity of the vehicle, specified in radians per second. A positive value steers the vehicle left and negative values steer the vehicle right.

Dependencies

To enable this port, set the `Vehicle inputs` parameter to `Vehicle Speed & Heading Angular Velocity`.

Output

state — Pose of vehicle
three-element vector

Current position and orientation of the vehicle, specified as a [*x y theta*] vector in meters and radians.

stateDot — Derivatives of `state` output
three-element vector

The current linear and angular velocities of the vehicle specified as a [*xDot yDot thetaDot*] vector in meters per second and radians per second. The linear and angular velocities are calculated by taking the derivative of the `state` output.

Parameters

Vehicle inputs — Type of speed and directional inputs for vehicle
`Wheel Speeds` (default) | `Vehicle Speed & Heading Angular Velocity`

The format of the model input commands

- `Wheel Speeds` — Angular speeds of the two wheels in radians per second.
- `Vehicle Speed & Heading Angular Velocity` — Vehicle speed in meters per second with a heading angular velocity in radians per second.

Wheel radius — Wheel radius of vehicle
`0.05` (default) | positive numeric scalar

The radius of the wheels on the vehicle, specified in meters.

Wheel speed range — Minimum and maximum vehicle speeds
`[-Inf Inf]` (default) | two-element vector

The wheel speed range is a two-element vector that provides the minimum and maximum vehicle wheel speeds, [*MinSpeed MaxSpeed*], specified in radians per second.

Track width — Track length of vehicle from wheel to wheel
`0.2` (default) | numeric scalar

Length of the track from the left wheel to right wheel, specified in meters.

Initial state — Initial pose of the vehicle
`[0;0;0]` (default) | three-element vector

The initial *xy*-position and orientation, θ, of the vehicle.

Simulate using — Type of simulation to run
`Interpreted execution` (default) | `Code generation`

- `Interpreted execution` — Simulate model using the MATLAB interpreter. For more information, see “Simulation Modes” (Simulink).
- `Code generation` — Simulate model using generated C code. The first time you run a simulation, Simulink generates C code for the block. The C code is reused for subsequent simulations, as long as the model does not change.

Tunable: No

Version History

Introduced in R2019b

References

[1] Lynch, Kevin M., and Frank C. Park. *Modern Robotics: Mechanics, Planning, and Control* 1st ed. Cambridge, MA: Cambridge University Press, 2017.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

Blocks

Ackermann Kinematic Model | Bicycle Kinematic Model | Unicycle Kinematic Model

Classes

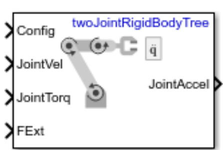
`differentialDriveKinematics`

Topics

“Control Differential Drive Robot in Gazebo with Simulink”

Forward Dynamics

Joint accelerations given joint torques and states



Libraries:
Robotics System Toolbox / Manipulator Algorithms

Description

The Forward Dynamics block computes joint accelerations for a robot model given a robot state that is made up of joint torques, joint states, and external forces. To get the joint accelerations, specify the robot configuration (joint positions), joint velocities, applied torques, and external forces.

Specify the robot model in the **Rigid body tree** parameter as a `rigidBodyTree` object, and set the Gravity property on the object. You can also import a robot model from an URDF (Unified Robot Description Format) file using `importrobot`.

Ports

Input

Config — Robot configuration
vector

Robot configuration, specified as a vector of positions for all nonfixed joints in the robot model, as set by the **Rigid body tree** parameter. You can also generate this vector for a complex robot using the `homeConfiguration` or `randomConfiguration` functions inside a Constant or MATLAB Function block.

JointVel — Joint velocities
vector

Joint velocities, specified as a vector. The number of joint velocities is equal to the degrees of freedom (number of nonfixed joints) of the robot.

JointTorq — Joint torques
vector

Joint torques, specified as a vector. Each element corresponds to a torque applied to a specific joint. The number of joint torques is equal to the degrees of freedom (number of nonfixed joints) of the robot.

FExt — External force matrix
6-by-*n* matrix

External force matrix, specified as a 6-by-*n* matrix, where *n* is the number of bodies in the robot model. The matrix contains nonzero values in the rows corresponding to specific bodies. Each row is

a vector of applied forces and torques that act as a wrench for that specific body. Generate this matrix using `externalForce` with a MATLAB Function block.

Output

JointAccel — Joint accelerations
vector

Joint accelerations, returned as a vector. The number of joint accelerations is equal to the degrees of freedom of the robot.

Parameters

Rigid body tree — Robot model
`twoJointRigidBodyTree` (default) | `RigidBodyTree` object

Robot model, specified as a `rigidBodyTree` object. You can also import a robot model from an URDF (Unified Robot Description Format) file using `importrobot`.

The default robot model, `twoJointRigidBodyTree`, is a robot with revolute joints and two degrees of freedom.

Simulate using — Type of simulation to run
`Interpreted execution` (default) | `Code generation`

- `Interpreted execution` — Simulate model using the MATLAB interpreter. This option shortens startup time but has a slower simulation speed than `Code generation`. In this mode, you can debug the source code of the block.
- `Code generation` — Simulate model using generated C code. The first time you run a simulation, Simulink generates C code for the block. The C code is reused for subsequent simulations, as long as the model does not change. This option requires additional startup time, but the speed of the subsequent simulations is comparable to `Interpreted execution`.

Tunable: No

Version History
Introduced in R2018a

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

See Also

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

Classes
`rigidBodyTree`

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

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

Gazebo Apply Command

Send command to Gazebo simulator

Libraries:
Robotics System Toolbox / Gazebo Co-Simulation

Description

The Apply Command block sends commands to a Gazebo simulation. The block accepts a command message, input as a bus signal, and sends the command to the Gazebo server.

To send command messages, connect to a Gazebo simulation. Open the block mask and click **Configure Gazebo network and simulation settings**. For more information see “Configure Gazebo Simulation” on page 4-21.

This block is part of a co-simulation interface between MATLAB and Gazebo for exchanging data and sending commands. To see a basic example, check “Perform Co-Simulation between Simulink and Gazebo”.

Limitations

- Models that use this block do not support Code Generation or Rapid Accelerator mode.

Ports

Input

Cmd — Gazebo Command
bus

Gazebo command message, specified as a bus. The command is an instruction for a specified model link or joint. Specify the model name as part of the bus signal using the Gazebo Select Entity block.

There are seven different command types with specific fields:

- `ApplyLinkWrench`:
  - `model_name` –– Variable-size `uint8` array representing the name of the model in the Gazebo simulator. You can specify this field using the Gazebo Select Entity block.
  - `link_name` –– Variable-size `uint8` array representing the name of the link in the model in the Gazebo simulator. You can specify this field using the Gazebo Select Entity block.
  - `force_type` –– Variable-size `uint8` array specified as `'SET'` or `'ADD'`. `'SET'` overwrites any existing force command for the specified duration. `'ADD'` adds the value with existing commands.
  - `Fx, fy, fz` –– `double` values specifying the amount of force applied to the Gazebo model link in world coordinates and Newtons.

  - torque_type –– Variable-size uint8 array specified as 'SET' or 'ADD'. 'SET' overwrites any existing torque command for the specified duration. 'ADD' adds the value with existing commands.
  - Tx, ty, tz –– double values specifying the amount of torque applied to the Gazebo model link in world coordinates and Newton-meters.
  - duration –– Bus containing seconds and nanoseconds as double integers, which specify how long to apply the torque in simulation time.

- ApplyJointTorque:

  - model_name –– Variable-size uint8 array representing the name of the model in the Gazebo simulator. You can specify this field using the Gazebo Select Entity block.
  - joint_name –– Variable-size uint8 array representing the name of the joint in the model in the Gazebo simulator. You can specify this field using the Gazebo Select Entity block.
  - index –– uint32 integer that identifies which joint axis the torque should be applied to.
  - effort –– double scalar value specifying the amount of torque or force to apply to the joint.
  - duration –– Bus containing seconds and nanoseconds as double integers, which specify how long to apply the torque in simulation time.

- SetLinkWorldPose:

  - model_name –– Variable-size uint8 array representing the name of the model in the Gazebo simulator.
  - link_name –– Variable-size uint8 array representing the name of the link in the model in the Gazebo simulator.
  - world_pose –– Bus containing position and orientation as [*x y z*] and [*x y z w*] double vectors, respectively.
  - duration –– Bus containing seconds and nanoseconds as double integers, which specify how long to apply the torque in simulation time.

- SetLinkLinearVelocity:

  - model_name –– Variable-size uint8 array representing the name of the model in the Gazebo simulator.
  - link_name –– Variable-size uint8 array representing the name of the link in the model in the Gazebo simulator.
  - velocity –– Bus containing linear velocity as [*x y z*] double vector.
  - duration –– Bus containing seconds and nanoseconds as double integers, which specify how long to apply the torque in simulation time.

- SetLinkAngularVelocity:

  - model_name –– Variable-size uint8 array representing the name of the model in the Gazebo simulator.
  - link_name –– Variable-size uint8 array representing the name of the link in the model in the Gazebo simulator.
  - velocity –– Bus containing angular velocity as [*x y z*] double vector.
  - duration –– Bus containing seconds and nanoseconds as double integers, which specify how long to apply the torque in simulation time.

- `SetJointPosition`:
  - `model_name` –– Variable-size `uint8` array representing the name of the model in the Gazebo simulator.
  - `joint_name` –– Variable-size `uint8` array representing the name of the joint in the model in the Gazebo simulator.
  - `index` –– `uint32` integer that identifies which joint axis the torque should be applied to.
  - `position` –– `double` scalar value representing the joint position.
  - `duration` –– Bus containing seconds and nanoseconds as `double` integers, which specify how long to apply the torque in simulation time.
- `SetJointVelocity`:
  - `model_name` –– Variable-size `uint8` array representing the name of the model in the Gazebo simulator.
  - `joint_name` –– Variable-size `uint8` array representing the name of the joint in the model in the Gazebo simulator.
  - `index` –– `uint32` integer that identifies which joint axis the torque should be applied to.
  - `velocity` –– `double` scalar value representing the joint velocity.
  - `duration` –– Bus containing seconds and nanoseconds as `double` integers, which specify how long to apply the torque in simulation time.

Note `SetJointVelocity` uses the *Set Instantaneous Velocity* method to set the joint velocity. For more information see, Setting Velocity on Links And Joints.

Note

- If a `duration` is `1.005` seconds, it would be `1` second and `5000000` nanoseconds as a bus.

Data Types: `bus`

Parameters

Command type — Type of command
`ApplyLinkWrench` (default) | `ApplyJointTorque` | `SetLinkWorldPose` | `SetLinkLinearVelocity` | `SetLinkAngularVelocity` | `SetJointPosition` | `SetJointVelocity`

Click **Select** to get a list of command types available in Gazebo. The input `Cmd` must contain the correct command message structure that matches this type.

Sample time — Sampling time of input
`0.001` (default) | positive

Sample time indicates the interval which commands are sent to the Gazebo simulator.

More About

Configure Gazebo Simulation

Click **Configure Gazebo network and simulation settings** in the block mask to launch the **Configure Gazebo Simulation** dialog box, which configures the synchronized stepping between Gazebo and Simulink. You can select the `Network Address` and specify `Hostname/IP Address` and `Port` of the computer running the Gazebo simulator with the Gazebo plugin installed. Then click **Test** to test the connection to the running Gazebo simulator. You can also specify the `Response timeout` in seconds. These settings apply to all Gazebo blocks for all open Simulink models.

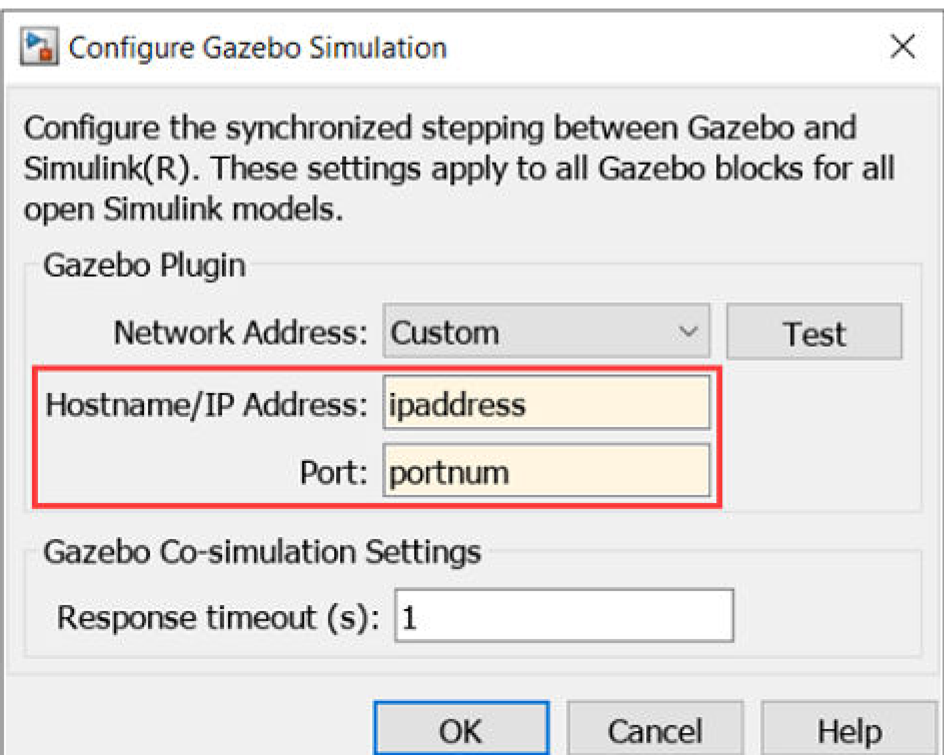
Starting from R2022b, you can connect to multiple Gazebo simulations from one or more machines. You can now specify a cell array of IP addresses and a cell array of port numbers in the MATLAB workspace and then specify their variable names to the `Hostname/IP Address` and `Port` boxes, respectively.

To connect to a single Gazebo session from MATLAB, specify the port number and IP address of the computer running the Gazebo simulator.

```
portnum = 14580;
ipaddress = '172.18.250.125';
```

To connect to multiple Gazebo sessions from MATLAB, specify the port numbers and IP addresses of the computers running the Gazebo simulator.

```
portnum = {14580,14581};
ipaddress = {'172.18.250.125','172.18.250.125'};
```



Version History

Introduced in R2019b

See Also

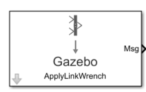
Blocks
Gazebo Blank Message | Gazebo Pacer | Gazebo Read | Gazebo Publish | Gazebo Subscribe | Gazebo Select Entity

Functions
`packageGazeboPlugin`

Topics
"Perform Co-Simulation between Simulink and Gazebo"
"Control Differential Drive Robot in Gazebo with Simulink"

Gazebo Blank Message

Create blank Gazebo command



Libraries:
Robotics System Toolbox / Gazebo Co-Simulation

Description

The Gazebo Blank Message block creates a blank Gazebo message or a command based on the specified type. The block output is a bus signal that contains the required elements for the type of command. Use a Bus Assignment block to modify specific fields in the bus signal. The bus signal initializes with zero value (ground).

To create blank Gazebo command, connect to a Gazebo simulation. Open the block mask and click **Configure Gazebo network and simulation settings**. For more information see “Configure Gazebo Simulation” on page 4-26.

This block is part of a co-simulation interface between MATLAB and Gazebo for exchanging data and sending commands. To see a basic example, check “Perform Co-Simulation between Simulink and Gazebo”.

Limitations

- Models that use this block do not support Code Generation or Rapid Accelerator mode.

Ports

Output

Msg — Blank message
bus

Blank message, returns as a bus signal with elements relevant to the specific `Message type`. The `Msg` output always outputs the most recent message received.

There are seven different message types with specific fields:

- `ApplyLinkWrench`:
  - `model_name` –– Variable-size `uint8` array representing the name of the model in the Gazebo simulator.
  - `link_name` –– Variable-size `uint8` array representing the name of the link in the model in the Gazebo simulator.
  - `force_type` –– Variable-size `uint8` array specified as `'SET'` or `'ADD'`. `'SET'` overwrites any existing force message for the specified duration. `'ADD'` adds the value with existing messages.
  - `Fx, fy, fz` –– `double` values specifying the amount of force applied to the Gazebo model link in world coordinates and Newtons.

  - `torque_type` –– Variable-size `uint8` array specified as `'SET'` or `'ADD'`. `'SET'` overwrites any existing torque message for the specified duration. `'ADD'` adds the value with existing messages.
  - `Tx, ty, tz` –– `double` values specifying the amount of torque applied to the Gazebo model link in world coordinates and Newton-meters.
  - `duration` –– Bus containing seconds and nanoseconds as `double` integers, which specify how long to apply the torque in simulation time.
- `ApplyJointTorque`:
  - `model_name` –– Variable-size `uint8` array representing the name of the model in the Gazebo simulator.
  - `joint_name` –– Variable-size `uint8` array representing the name of the joint in the model in the Gazebo simulator.
  - `index` –– `uint32` integer that identifies which joint axis the torque should be applied to.
  - `effort` –– `double` scalar value specifying the amount of torque or force to apply to the joint.
  - `duration` –– Bus containing seconds and nanoseconds as `double` integers, which specify how long to apply the torque in simulation time.
- `SetLinkWorldPose`:
  - `model_name` –– Variable-size `uint8` array representing the name of the model in the Gazebo simulator.
  - `link_name` –– Variable-size `uint8` array representing the name of the link in the model in the Gazebo simulator.
  - `world_pose` –– Bus containing position and orientation as `[x y z]` and `[x y z w]` `double` vectors, respectively.
  - `duration` –– Bus containing seconds and nanoseconds as `double` integers, which specify how long to apply the torque in simulation time.
- `SetLinkLinearVelocity`:
  - `model_name` –– Variable-size `uint8` array representing the name of the model in the Gazebo simulator.
  - `link_name` –– Variable-size `uint8` array representing the name of the link in the model in the Gazebo simulator.
  - `velocity` –– Bus containing linear velocity as `[x y z]` `double` vector.
  - `duration` –– Bus containing seconds and nanoseconds as `double` integers, which specify how long to apply the torque in simulation time.
- `SetLinkAngularVelocity`:
  - `model_name` –– Variable-size `uint8` array representing the name of the model in the Gazebo simulator.
  - `link_name` –– Variable-size `uint8` array representing the name of the link in the model in the Gazebo simulator.
  - `velocity` –– Bus containing angular velocity as `[x y z]` `double` vector.
  - `duration` –– Bus containing seconds and nanoseconds as `double` integers, which specify how long to apply the torque in simulation time.

- SetJointPosition:
  - model_name –– Variable-size uint8 array representing the name of the model in the Gazebo simulator.
  - joint_name –– Variable-size uint8 array representing the name of the joint in the model in the Gazebo simulator.
  - index –– uint32 integer that identifies which joint axis the torque should be applied to.
  - position –– double scalar value representing the joint position.
  - duration –– Bus containing seconds and nanoseconds as double integers, which specify how long to apply the torque in simulation time.
- SetJointVelocity:
  - model_name –– Variable-size uint8 array representing the name of the model in the Gazebo simulator.
  - joint_name –– Variable-size uint8 array representing the name of the joint in the model in the Gazebo simulator.
  - index –– uint32 integer that identifies which joint axis the torque should be applied to.
  - velocity –– double scalar value representing the joint velocity.
  - duration –– Bus containing seconds and nanoseconds as double integers, which specify how long to apply the torque in simulation time.

Note SetJointVelocity uses the *Set Instantaneous Velocity* method to set the joint velocity. For more information see, Setting Velocity on Links And Joints.

Note

- If a duration is 1.005 seconds, it would be 1 second and 5000000 nanoseconds as a bus.

Data Types: bus

Parameters

Message type — Type of message
ApplyLinkWrench (default) | ApplyJointTorque | SetLinkWorldPose | SetLinkLinearVelocity | SetLinkAngularVelocity | SetJointPosition | SetJointVelocity

Click **Select** to get a list of message types available in Gazebo.

Sample time — Sampling time of input
0.001 (default) | positive

Sample time indicates when, during simulation, the block produces outputs and if appropriate, updates its internal state.

More About

Configure Gazebo Simulation

Click **Configure Gazebo network and simulation settings** in the block mask to launch the **Configure Gazebo Simulation** dialog box, which configures the synchronized stepping between Gazebo and Simulink. You can select the `Network Address` and specify `Hostname/IP Address` and `Port` of the computer running the Gazebo simulator with the Gazebo plugin installed. Then click **Test** to test the connection to the running Gazebo simulator. You can also specify the `Response timeout` in seconds. These settings apply to all Gazebo blocks for all open Simulink models.

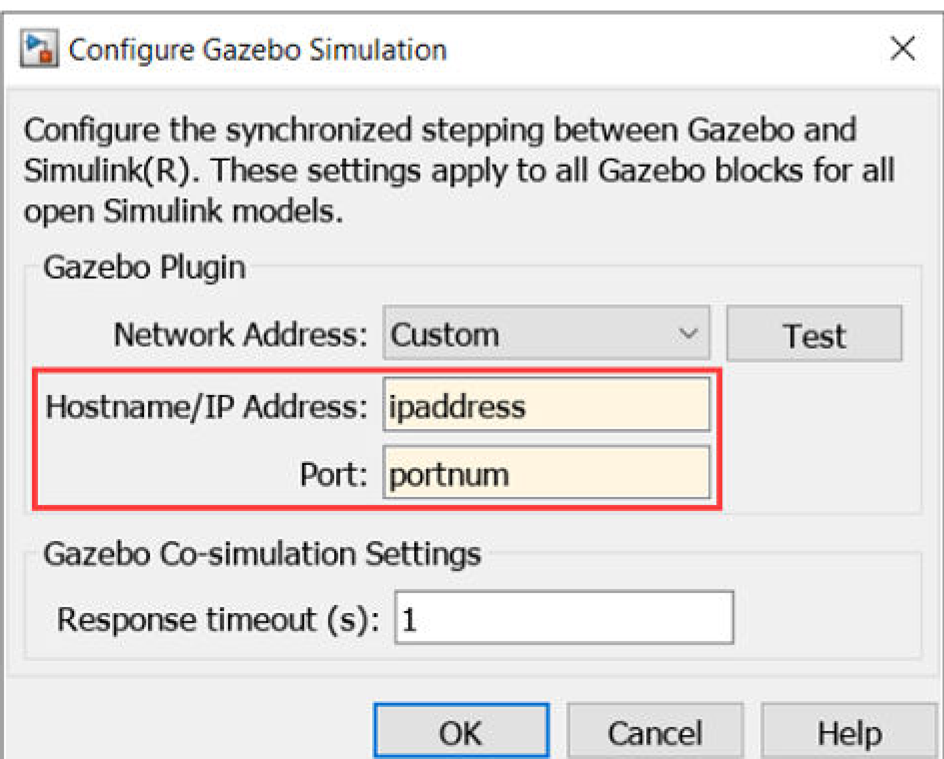
Starting from R2022b, you can connect to multiple Gazebo simulations from one or more machines. You can now specify a cell array of IP addresses and a cell array of port numbers in the MATLAB workspace and then specify their variable names to the `Hostname/IP Address` and `Port` boxes, respectively.

To connect to a single Gazebo session from MATLAB, specify the port number and IP address of the computer running the Gazebo simulator.

```
portnum = 14580;
ipaddress = '172.18.250.125';
```

To connect to multiple Gazebo sessions from MATLAB, specify the port numbers and IP addresses of the computers running the Gazebo simulator.

```
portnum = {14580,14581};
ipaddress = {'172.18.250.125','172.18.250.125'};
```



Version History

Introduced in R2019b

See Also

Blocks
Gazebo Apply Command | Gazebo Pacer | Gazebo Read | Gazebo Publish | Gazebo Subscribe | Gazebo Select Entity

Functions
`packageGazeboPlugin`

Topics
"Control Differential Drive Robot in Gazebo with Simulink"

Gazebo Pacer

Settings for synchronized stepping between Gazebo and Simulink

Libraries:
Robotics System Toolbox / Gazebo Co-Simulation

Description

The Gazebo Pacer block synchronizes the simulation times between Gazebo and Simulink. Synchronization is important for ensuring your Simulink model and the Gazebo simulation behave correctly. The block outputs a Boolean indicating successful synchronization. Synchronized stepping is only supported for one Gazebo simulation. Your entire model, including referenced models, can only contain one Gazebo Pacer block.

To ensure successful synchronization, connect to a Gazebo simulation. Open the block mask and click **Configure Gazebo network and simulation settings**. For more information see “Configure Gazebo Simulation” on page 4-29.

Select the `Reset behavior` to reset the Gazebo simulation on model restart or only reset simulation time.

This block is part of a co-simulation interface between MATLAB and Gazebo for exchanging data and sending commands. To see a basic example, check “Perform Co-Simulation between Simulink and Gazebo”.

Limitations

- Models that use this block do not support Code Generation or Rapid Accelerator mode.

Ports

Output

Status — Status of synchronization
0 | 1

Status of synchronization, output as either `0` or `1`. A value of `0` indicates successful time syncing. A value of `1` means the simulations are out of sync.

Data Types: `uint8`

Parameters

Reset behavior — Reset simulation time or scene
`Reset Gazebo simulation time` (default) | `Reset Gazebo simulation time and scene`

Select from the Reset behavior drop-down.

- `Reset Gazebo simulation time` — Resets the Gazebo simulator time.
- `Reset Gazebo simulation time and scene` — Resets both the Gazebo simulator time and scene.

Sample time — Sampling time of input
`0.001` (default) | positive

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

More About

Configure Gazebo Simulation

Click **Configure Gazebo network and simulation settings** in the block mask to launch the **Configure Gazebo Simulation** dialog box, which configures the synchronized stepping between Gazebo and Simulink. You can select the `Network Address` and specify `Hostname/IP Address` and `Port` of the computer running the Gazebo simulator with the Gazebo plugin installed. Then click **Test** to test the connection to the running Gazebo simulator. You can also specify the `Response timeout` in seconds. These settings apply to all Gazebo blocks for all open Simulink models.

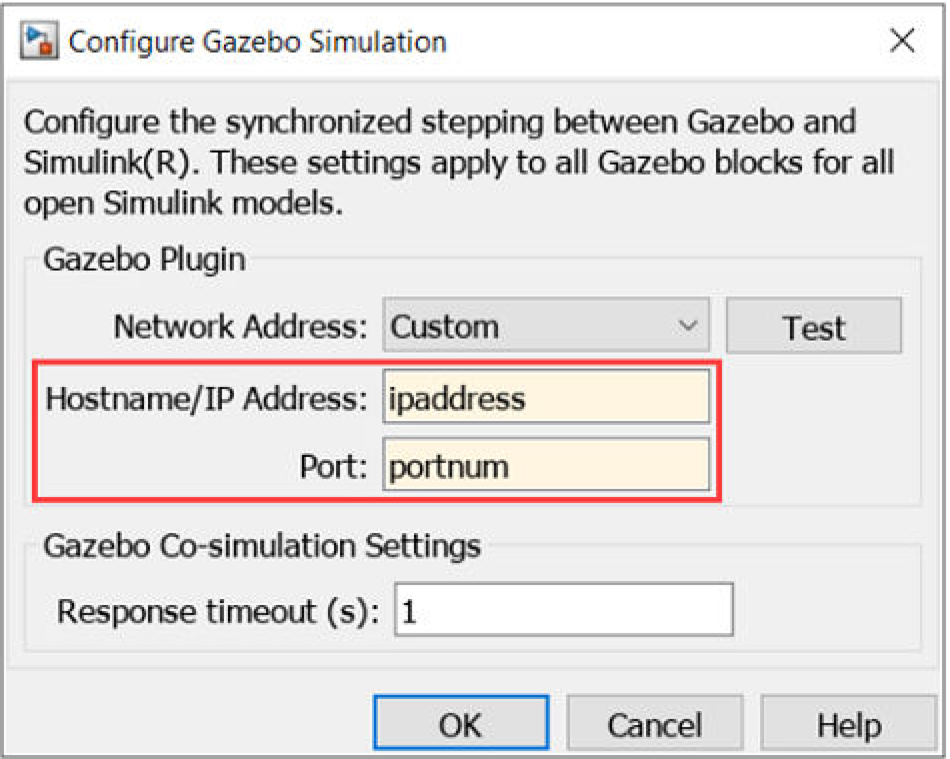
Starting from R2022b, you can connect to multiple Gazebo simulations from one or more machines. You can now specify a cell array of IP addresses and a cell array of port numbers in the MATLAB workspace and then specify their variable names to the `Hostname/IP Address` and `Port` boxes, respectively.

To connect to a single Gazebo session from MATLAB, specify the port number and IP address of the computer running the Gazebo simulator.

```
portnum = 14580;
ipaddress = '172.18.250.125';
```

To connect to multiple Gazebo sessions from MATLAB, specify the port numbers and IP addresses of the computers running the Gazebo simulator.

```
portnum = {14580,14581};
ipaddress = {'172.18.250.125','172.18.250.125'};
```



Version History

Introduced in R2019b

See Also

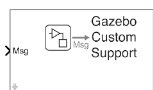
Blocks
Gazebo Apply Command | Gazebo Blank Message | Gazebo Read | Gazebo Publish | Gazebo Subscribe | Gazebo Select Entity

Functions
`packageGazeboPlugin`

Topics
"Control Differential Drive Robot in Gazebo with Simulink"

Gazebo Publish

Send custom messages to Gazebo server



Libraries:
Robotics System Toolbox / Gazebo Co-Simulation

Description

The Gazebo Publish block sends custom messages to Gazebo server based on the topic and message type that the block specifies.

To send custom messages, connect to a Gazebo simulation. Open the block mask and click **Configure Gazebo network and simulation settings**. For more information see “Configure Gazebo Simulation” on page 4-32.

This block is part of a co-simulation interface between MATLAB and Gazebo for exchanging data and sending commands.

Limitations

- Models that use this block do not support Code Generation or Rapid Accelerator mode.

Ports

Input

Msg — Gazebo custom message
bus

Gazebo custom message, specified as a bus signal, with elements relevant to the specific `Topic` and `Message type`.

Data Types: `bus`

Parameters

Topic source — Source for specifying topic
`From Gazebo` (default) | `Specify your own`

To get a topic from an existing Gazebo simulation, select `From Gazebo`. Click the **Select** button to see a list of available topics. To connect to a Gazebo simulation, click **Configure Gazebo network and simulation settings** in the block mask.

To enter a custom topic without an active Gazebo connection, select `Specify your own`. Use the `Topic` parameter to type the name of the message.

Topic — Topic name of custom message
`/my_topic` (default) | string

Topic name of custom message, specified as a string.

To get a topic from an existing Gazebo simulation, select `From Gazebo`. Click the **Select** button to see a list of available topics. To connect to a Gazebo simulation, click **Configure Gazebo network and simulation settings** in the block mask.

To specify a topic without connecting, select `Specify your own`.

Message type — Gazebo custom message type
`gazebo_msgs/TestPose` (default) | string

Click **Select** to get a list of message types available in Gazebo. If you choose your `Topic` from a connected Gazebo simulation, this parameter is set automatically.

Sample time — Sampling time of input
`0.001` (default) | positive

Sample time indicates the interval at which messages are sent to the Gazebo simulator.

More About

Configure Gazebo Simulation

Click **Configure Gazebo network and simulation settings** in the block mask to launch the **Configure Gazebo Simulation** dialog box, which configures the synchronized stepping between Gazebo and Simulink. You can select the `Network Address` and specify `Hostname/IP Address` and `Port` of the computer running the Gazebo simulator with the Gazebo plugin installed. Then click **Test** to test the connection to the running Gazebo simulator. You can also specify the `Response timeout` in seconds. These settings apply to all Gazebo blocks for all open Simulink models.

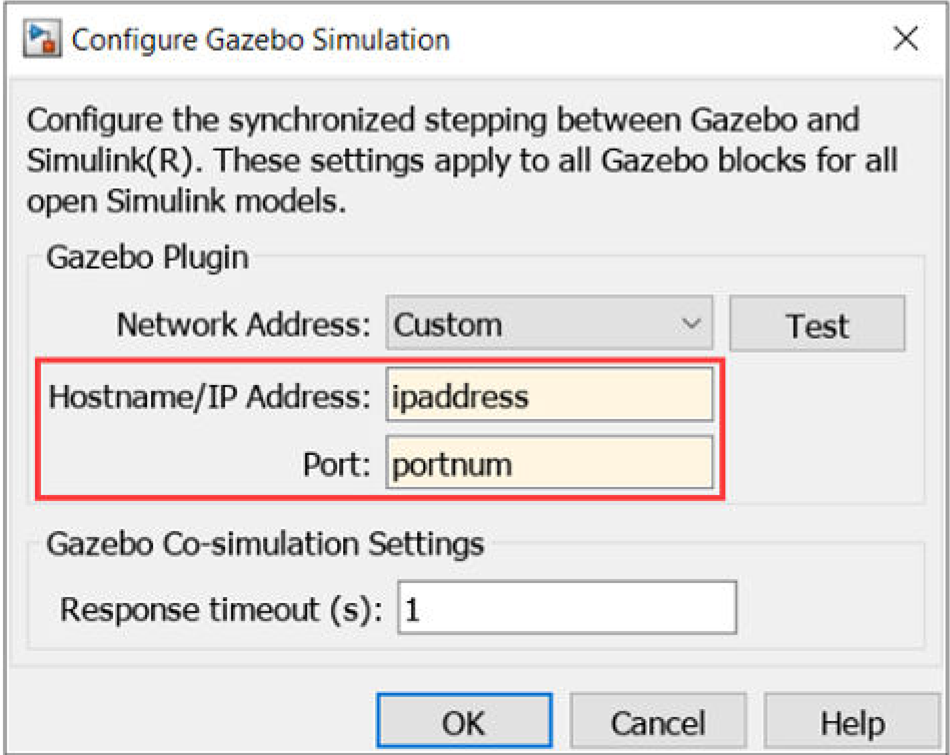
Starting from R2022b, you can connect to multiple Gazebo simulations from one or more machines. You can now specify a cell array of IP addresses and a cell array of port numbers in the MATLAB workspace and then specify their variable names to the `Hostname/IP Address` and `Port` boxes, respectively.

To connect to a single Gazebo session from MATLAB, specify the port number and IP address of the computer running the Gazebo simulator.

```
portnum = 14580;
ipaddress = '172.18.250.125';
```

To connect to multiple Gazebo sessions from MATLAB, specify the port numbers and IP addresses of the computers running the Gazebo simulator.

```
portnum = {14580,14581};
ipaddress = {'172.18.250.125','172.18.250.125'};
```



Version History

Introduced in R2020b

See Also

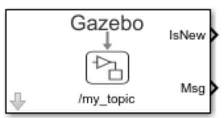
Blocks
Gazebo Apply Command | Gazebo Blank Message | Gazebo Pacer | Gazebo Read | Gazebo Subscribe | Gazebo Select Entity

Functions
`packageGazeboPlugin` | `gazebogenmsg`

Topics
"Perform Co-Simulation between Simulink and Gazebo"

Gazebo Read

Receive messages from Gazebo server



Libraries:
Robotics System Toolbox / Gazebo Co-Simulation

Description

The Gazebo Read block receives messages from the Gazebo server based on the topic and message type that the block specifies. The block outputs the latest message received as a bus signal, `Msg`, and a Boolean, `IsNew`, which indicates whether a message was received during the previous time step.

To receive messages from Gazebo server, connect to a Gazebo simulation. Open the block mask and click **Configure Gazebo network and simulation settings**. For more information see “Configure Gazebo Simulation” on page 4-35.

This block is part of a co-simulation interface between MATLAB and Gazebo for exchanging data and sending commands. To see a basic example, check “Perform Co-Simulation between Simulink and Gazebo”

Limitations

- Models that use this block do not support Code Generation or Rapid Accelerator mode.

Ports

Output

IsNew — Status of messages in the previous time step
`0` (default) | `1`

Status of the message received, output as a Boolean, which indicates whether the block output `Msg` was received in the previous time step.

Data Types: `Boolean`

Msg — Gazebo message
bus

Gazebo message, output as a bus signal, with elements relevant to the specific `Topic` and `Message type`.

The `Msg` output always outputs the most recent message received.

Data Types: `bus`

Parameters

Topic source — Source for specifying topic
From Gazebo (default) | Specify your own

To get a topic from an existing Gazebo simulation, select From Gazebo. Click the **Select** button to see a list of available topics. To connect to a Gazebo simulation, click **Configure Gazebo network and simulation settings** in the block mask.

To enter a custom topic without an active Gazebo connection, select Specify your own. Use the Topic parameter to type the name of the message.

Topic — Topic name of message
/my_topic (default) | string

Topic name of message, specified as a string.

To get a topic from an existing Gazebo simulation, select From Gazebo. Click the **Select** button to see a list of available topics. To connect to a Gazebo simulation, click **Configure Gazebo network and simulation settings** in the block mask.

To specify a topic without connecting, select Specify your own.

Message type — Gazebo message type
gazebo_msgs/Pose (default) | gazebo_msgs/Image | gazebo_msgs/IMU | gazebo_msgs/LaserScan | gazebo_msgs/JointState | gazebo_msgs/LinkState

Click **Select** to get a list of message types available in Gazebo. If you choose your Topic from a connected Gazebo simulation, this parameter is set automatically.

Sample time — Sampling time of input
0.001 (default) | positive scalar

Sample time indicates when, during simulation, the block produces outputs and if appropriate, updates its internal state.

Number of Joint Axis — Number of joint axis
2 (default) | positive integer in range [1, 100]

Select Make Joint Axis Related Bus Signal to Fixed Dimensions to convert the joint-axis-related bus signal from variable dimension to fixed dimension. Then specify the number of joint axis.

Dependencies

To enable this parameter, set Message type to gazebo_msgs/JointState and select the Make Joint Axis Related Bus Signal to Fixed Dimensions check box.

More About

Configure Gazebo Simulation

Click **Configure Gazebo network and simulation settings** in the block mask to launch the **Configure Gazebo Simulation** dialog box, which configures the synchronized stepping between

Gazebo and Simulink. You can select the `Network Address` and specify `Hostname/IP Address` and `Port` of the computer running the Gazebo simulator with the Gazebo plugin installed. Then click **Test** to test the connection to the running Gazebo simulator. You can also specify the `Response timeout` in seconds. These settings apply to all Gazebo blocks for all open Simulink models.

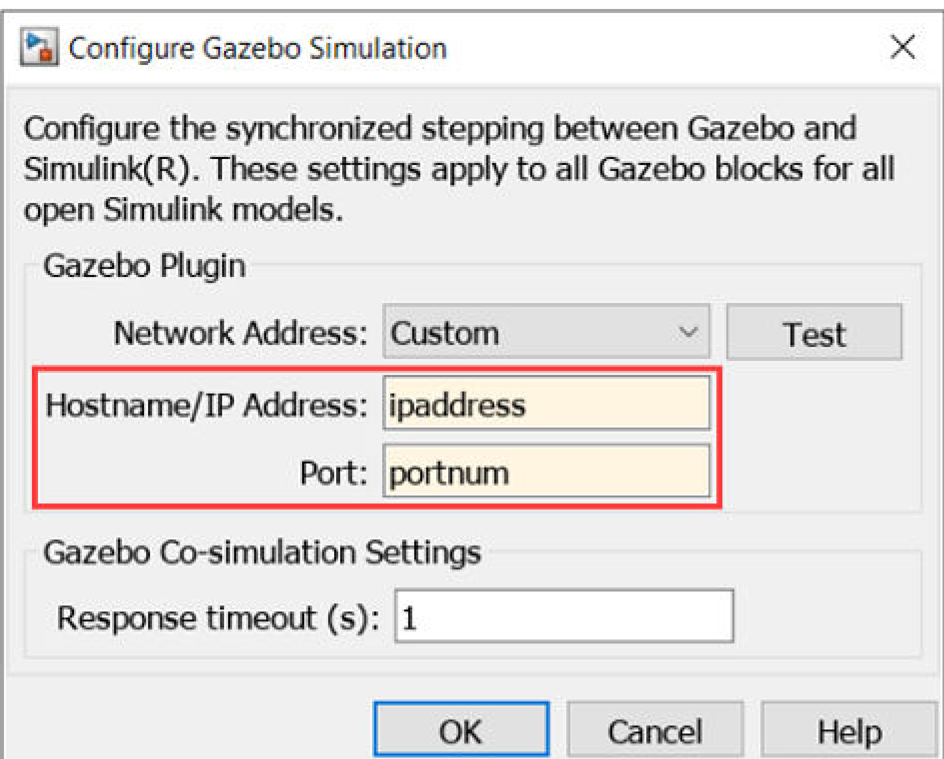
Starting from R2022b, you can connect to multiple Gazebo simulations from one or more machines. You can now specify a cell array of IP addresses and a cell array of port numbers in the MATLAB workspace and then specify their variable names to the `Hostname/IP Address` and `Port` boxes, respectively.

To connect to a single Gazebo session from MATLAB, specify the port number and IP address of the computer running the Gazebo simulator.

```
portnum = 14580;
ipaddress = '172.18.250.125';
```

To connect to multiple Gazebo sessions from MATLAB, specify the port numbers and IP addresses of the computers running the Gazebo simulator.

```
portnum = {14580,14581};
ipaddress = {'172.18.250.125','172.18.250.125'};
```



Version History

Introduced in R2019b

See Also

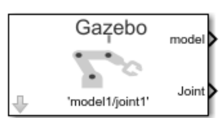
Blocks
Gazebo Apply Command | Gazebo Blank Message | Gazebo Pacer | Gazebo Publish | Gazebo Subscribe | Gazebo Select Entity

Functions
`packageGazeboPlugin`

Topics
"Control Differential Drive Robot in Gazebo with Simulink"

Gazebo Select Entity

Select a Gazebo entity



Libraries:
Robotics System Toolbox / Gazebo Co-Simulation

Description

The Gazebo Select Entity block retrieves the model name of a Gazebo entity, such as a link or joint, from a simulated environment. The block outputs a string for both the model and associated joint or link name. Use both these names when specifying commands using the Gazebo Apply Command block.

Before selecting an entity, connect to a Gazebo simulation. Open the block mask and click **Configure Gazebo network and simulation settings**. For more information see "Configure Gazebo Simulation" on page 4-39.

This block is part of a co-simulation interface between MATLAB and Gazebo for exchanging data and sending commands. To see a basic example, check "Perform Co-Simulation between Simulink and Gazebo"

Limitations

- Models that use this block do not support Code Generation or Rapid Accelerator mode.

Ports

Output

model — Model name of entity
`model1` (default) | string (`uint8[]`)

Model name of entity, output as string scalar. Strings are output as a variable-size `uint8` array for Gazebo.

Data Types: `uint8`

Joint/Link — Associated joint or link name of entity
joint1 (default) | string (`uint8[]`)

Associated joint or link, output as a string scalar. Strings are output as a `uint8` array for Gazebo.

Data Types: `uint8`

Parameters

Model Name — Choose model name
`'model1/joint1'` (default) | string scalar

Choose a model by clicking **Select**, which brings up a list of available names available on the Gazebo server. The block assumes you are already connected to a Gazebo simulation. If not, click **Configure Gazebo network and simulation settings** in the block mask.

Output vector size upper bound — Upper limit of output array
128 (default)

Upper limit of the size of the output `uint8` arrays, `Model Name` and `Joint/Link`. Increase the upper bound when the names are longer than the default value `128`.

More About

Configure Gazebo Simulation

Click **Configure Gazebo network and simulation settings** in the block mask to launch the **Configure Gazebo Simulation** dialog box, which configures the synchronized stepping between Gazebo and Simulink. You can select the `Network Address` and specify `Hostname/IP Address` and `Port` of the computer running the Gazebo simulator with the Gazebo plugin installed. Then click **Test** to test the connection to the running Gazebo simulator. You can also specify the `Response timeout` in seconds. These settings apply to all Gazebo blocks for all open Simulink models.

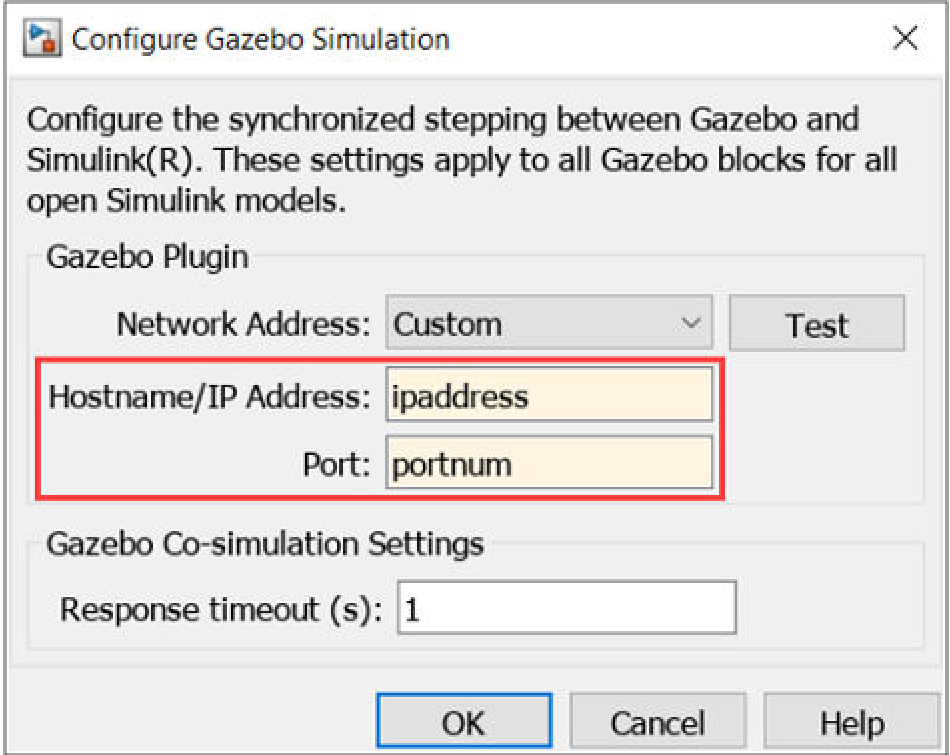
Starting from R2022b, you can connect to multiple Gazebo simulations from one or more machines. You can now specify a cell array of IP addresses and a cell array of port numbers in the MATLAB workspace and then specify their variable names to the `Hostname/IP Address` and `Port` boxes, respectively.

To connect to a single Gazebo session from MATLAB, specify the port number and IP address of the computer running the Gazebo simulator.

```
portnum = 14580;
ipaddress = '172.18.250.125';
```

To connect to multiple Gazebo sessions from MATLAB, specify the port numbers and IP addresses of the computers running the Gazebo simulator.

```
portnum = {14580,14581};
ipaddress = {'172.18.250.125','172.18.250.125'};
```



Version History

Introduced in R2019b

See Also

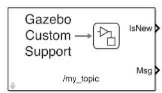
Blocks
Gazebo Apply Command | Gazebo Blank Message | Gazebo Pacer | Gazebo Read | Gazebo Publish | Gazebo Subscribe

Functions
`packageGazeboPlugin`

Topics
"Control Differential Drive Robot in Gazebo with Simulink"

Gazebo Subscribe

Receive custom messages from Gazebo server



Libraries:
Robotics System Toolbox / Gazebo Co-Simulation

Description

The Gazebo Subscribe block receives custom messages from Gazebo server based on the topic and message type that the block specifies. The block outputs the latest message received as a bus signal, `Msg`, and a Boolean, `IsNew`, which indicates whether a message was received during the previous time step.

To receive custom messages, connect to a Gazebo simulation. Open the block mask and click **Configure Gazebo network and simulation settings**. For more information see “Configure Gazebo Simulation” on page 4-42.

This block is part of a co-simulation interface between MATLAB and Gazebo for exchanging data and sending commands.

Limitations

- Models that use this block do not support Code Generation or Rapid Accelerator mode.

Ports

Output

IsNew — Status of custom messages in the previous time step
`0` (default) | `1`

Status of the custom message received, output as a Boolean, which indicates whether the block output `Msg` was received in the previous time step.

Data Types: `Boolean`

Msg — Gazebo custom message
bus

Gazebo custom message, output as a bus signal, with elements relevant to the specific `Topic` and `Message type`.

The `Msg` output always outputs the most recent message received.

Data Types: `bus`

Parameters

Topic source — Source for specifying topic
`From Gazebo` (default) | `Specify your own`

To get a topic from an existing Gazebo simulation, select `From Gazebo`. Click the **Select** button to see a list of available topics. To connect to a Gazebo simulation, click **Configure Gazebo network and simulation settings** in the block mask.

To enter a custom topic without an active Gazebo connection, select `Specify your own`. Use the `Topic` parameter to type the name of the message.

Topic — Topic name of custom message
`/my_topic` (default) | string

Topic name of custom message, specified as a string.

To get a topic from an existing Gazebo simulation, select `From Gazebo`. Click the **Select** button to see a list of available topics. To connect to a Gazebo simulation, click **Configure Gazebo network and simulation settings** in the block mask.

To specify a topic without connecting, select `Specify your own`.

Message type — Gazebo custom message type
`gazebo_msgs/TestPose` (default) | string

Click **Select** to get a list of message types available in Gazebo. If you choose your `Topic` from a connected Gazebo simulation, this parameter is set automatically.

Sample time — Sampling time of input
`0.001` (default) | positive

Sample time indicates the interval at which messages are received from the Gazebo simulator.

More About

Configure Gazebo Simulation

Click **Configure Gazebo network and simulation settings** in the block mask to launch the **Configure Gazebo Simulation** dialog box, which configures the synchronized stepping between Gazebo and Simulink. You can select the `Network Address` and specify `Hostname/IP Address` and `Port` of the computer running the Gazebo simulator with the Gazebo plugin installed. Then click **Test** to test the connection to the running Gazebo simulator. You can also specify the `Response timeout` in seconds. These settings apply to all Gazebo blocks for all open Simulink models.

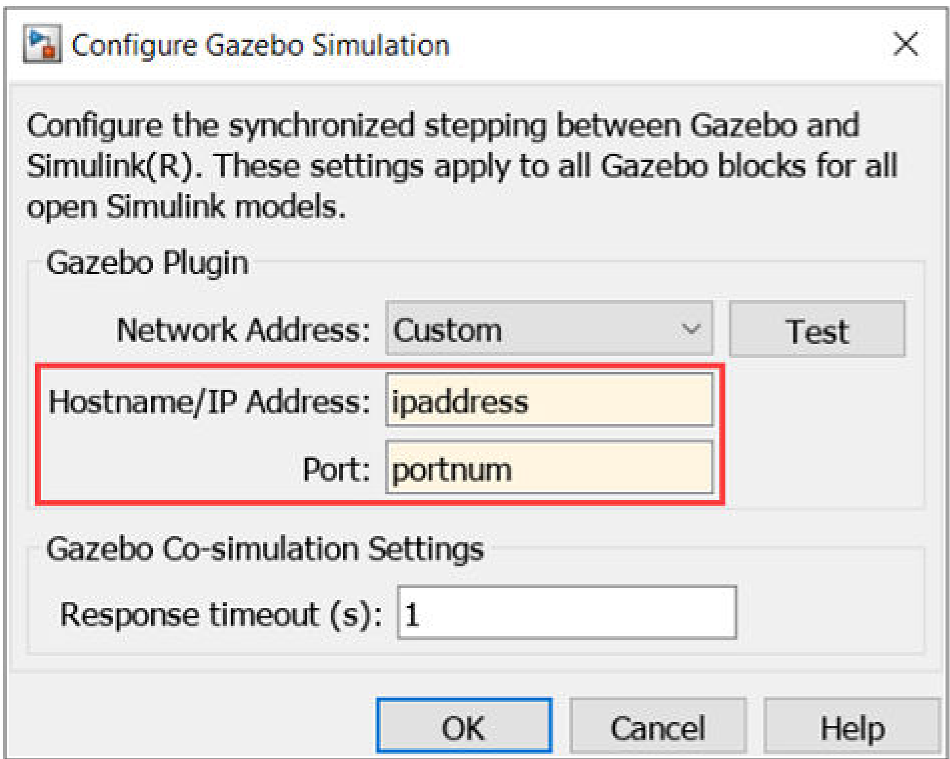
Starting from R2022b, you can connect to multiple Gazebo simulations from one or more machines. You can now specify a cell array of IP addresses and a cell array of port numbers in the MATLAB workspace and then specify their variable names to the `Hostname/IP Address` and `Port` boxes, respectively.

To connect to a single Gazebo session from MATLAB, specify the port number and IP address of the computer running the Gazebo simulator.

```
portnum = 14580;
ipaddress = '172.18.250.125';
```

To connect to multiple Gazebo sessions from MATLAB, specify the port numbers and IP addresses of the computers running the Gazebo simulator.

```
portnum = {14580,14581};
ipaddress = {'172.18.250.125','172.18.250.125'};
```



Version History

Introduced in R2020b

See Also

Blocks

Gazebo Apply Command | Gazebo Blank Message | Gazebo Pacer | Gazebo Read | Gazebo Publish | Gazebo Select Entity

Functions

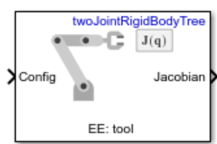
`packageGazeboPlugin` | `gazebogenmsg`

Topics

"Perform Co-Simulation between Simulink and Gazebo"

Get Jacobian

Geometric Jacobian for robot configuration



Libraries:
Robotics System Toolbox / Manipulator Algorithms

Description

The Get Jacobian block returns the geometric Jacobian relative to the base for the specified end effector at the given configuration of a `rigidBodyTree` robot model.

The Jacobian maps the joint-space velocity to the end-effector velocity relative to the base coordinate frame. The end-effector velocity equals:

$$V_{EE} = \begin{bmatrix} \omega_x \\ \omega_y \\ \omega_z \\ v_x \\ v_y \\ v_z \end{bmatrix} = J\dot{q} = J\begin{bmatrix} \dot{q}_1 \\ \vdots \\ \dot{q}_n \end{bmatrix}$$

ω is the angular velocity, υ is the linear velocity, and $\dot{q}$ is the joint-space velocity.

Ports

Input

Config — Robot configuration
vector

Robot configuration, specified as a vector of positions for all nonfixed joints in the robot model, as set by the **Rigid body tree** parameter. You can also generate this vector for a complex robot using the `homeConfiguration` or `randomConfiguration` functions inside a Constant or MATLAB Function block.

Output

Jacobian — Geometric Jacobian of end effector
6-by-*n* matrix

Geometric Jacobian of the end effector with the specified configuration, **Config**, returned as a 6-by-*n* matrix, where *n* is the number of degrees of freedom of the end effector. The Jacobian maps the joint-space velocity to the end-effector velocity relative to the base coordinate frame. The end-effector velocity equals:

$$V_{EE} = \begin{bmatrix} \omega_x \\ \omega_y \\ \omega_z \\ v_x \\ v_y \\ v_z \end{bmatrix} = J\dot{q} = J\begin{bmatrix} \dot{q}_1 \\ \vdots \\ \dot{q}_n \end{bmatrix}$$

ω is the angular velocity, υ is the linear velocity, and $\dot{q}$ is the joint-space velocity.

Parameters

Rigid body tree — Robot model
twoJointRigidBodyTree (default) | RigidBodyTree object

Robot model, specified as a rigidBodyTree object. You can also import a robot model from an URDF (Unified Robot Description Format) file using importrobot.

The default robot model, twoJointRigidBodyTree, is a robot with revolute joints and two degrees of freedom.

End effector — End effector for Jacobian
body name

End effector for Jacobian, specified as a body name from the **Rigid body tree** robot model. To access body names from the robot model, click **Select body**.

Simulate using — Type of simulation to run
Interpreted execution (default) | Code generation

- Interpreted execution — Simulate model using the MATLAB interpreter. This option shortens startup time but has a slower simulation speed than Code generation. In this mode, you can debug the source code of the block.
- Code generation — Simulate model using generated C code. The first time you run a simulation, Simulink generates C code for the block. The C code is reused for subsequent simulations, as long as the model does not change. This option requires additional startup time, but the speed of the subsequent simulations is comparable to Interpreted execution.

Tunable: No

Version History

Introduced in R2018a

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

See Also

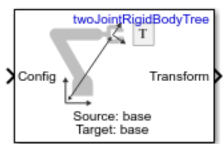
Blocks
Get Transform | Forward Dynamics | Inverse Dynamics | Gravity Torque | Joint Space Mass Matrix | Velocity Product Torque

Classes
`rigidBodyTree`

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

Get Transform

Get transform between body frames



Libraries:
Robotics System Toolbox / Manipulator Algorithms

Description

The Get Transform block returns the homogeneous transformation between body frames on the **Rigid body tree** robot model. Specify a `rigidBodyTree` object for the robot model, and select a source and target body in the block.

The block uses **Config**, the robot configuration (joint positions) input, to calculate the transformation from the source body to the target body. This transformation is used to convert coordinates from the source to the target body. To convert to base coordinates, use the base body name as the **Target body** parameter.

Ports

Input

Config — Robot configuration
vector

Robot configuration, specified as a vector of positions for all nonfixed joints in the robot model, as set by the **Rigid body tree** parameter. You can also generate this vector for a complex robot using the `homeConfiguration` or `randomConfiguration` functions inside a Constant or MATLAB Function block.

Output

Transform — Homogeneous transform
4-by-4 matrix

Homogeneous transform, returned as a 4-by-4 matrix.

Parameters

Rigid body tree — Robot model
`twoJointRigidBodyTree` (default) | `RigidBodyTree` object

Robot model, specified as a `rigidBodyTree` object. You can also import a robot model from an URDF (Unified Robot Description Formation) file using `importrobot`.

The default robot model, `twoJointRigidBodyTree`, is a robot with revolute joints and two degrees of freedom.

Target body — Target body name
body name

Target body name, specified as a body name from the robot model specified in **Rigid body tree**. To access body names from the robot model, click **Select body**. The target frame is the coordinate system you want to transform points into.

Source body — Source body name
body name

Source body name, specified as a body name from the robot model specified in **Rigid body tree**.To access body names from the robot model, click **Select body**. The source frame is the coordinate system you want points transformed from.

Simulate using — Type of simulation to run
`Interpreted execution` (default) | `Code generation`

- `Interpreted execution` — Simulate model using the MATLAB interpreter. This option shortens startup time but has a slower simulation speed than `Code generation`. In this mode, you can debug the source code of the block.
- `Code generation` — Simulate model using generated C code. The first time you run a simulation, Simulink generates C code for the block. The C code is reused for subsequent simulations, as long as the model does not change. This option requires additional startup time, but the speed of the subsequent simulations is comparable to `Interpreted execution`.

Tunable: No

Version History

Introduced in R2018a

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

Blocks

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

Classes

`rigidBodyTree`

Functions

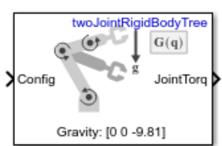
`getTransform` | `importrobot` | `homeConfiguration` | `randomConfiguration`

Topics

"Perform Safe Trajectory Tracking Control Using Robotics Manipulator Blocks"

Gravity Torque

Joint torques that compensate gravity



Libraries:
Robotics System Toolbox / Manipulator Algorithms

Description

The Gravity Torque block returns the joint torques required to hold the robot at a given configuration with the current Gravity setting on the **Rigid body tree** robot model.

Ports

Input

Config — Robot configuration
vector

Robot configuration, specified as a vector of positions for all nonfixed joints in the robot model, as set by the **Rigid body tree** parameter. You can also generate this vector for a complex robot using the `homeConfiguration` or `randomConfiguration` functions inside a Constant or MATLAB Function block.

Output

JointTorq — Joint torques
vector

Joint torques, specified as a vector. Each element corresponds to a torque applied to a specific joint. The number of joint torques is equal to the degrees of freedom (number of nonfixed joints) of the robot.

Parameters

Rigid body tree — Robot model
`twoJointRigidBodyTree` (default) | `RigidBodyTree` object

Robot model, specified as a `rigidBodyTree` object. You can also import a robot model from an URDF (Unified Robot Description Formation) file using `importrobot`.

The default robot model, `twoJointRigidBodyTree`, is a robot with revolute joints and two degrees of freedom.

Simulate using — Type of simulation to run
`Interpreted execution` (default) | `Code generation`

- `Interpreted execution` — Simulate model using the MATLAB interpreter. This option shortens startup time but has a slower simulation speed than `Code generation`. In this mode, you can debug the source code of the block.
- `Code generation` — Simulate model using generated C code. The first time you run a simulation, Simulink generates C code for the block. The C code is reused for subsequent simulations, as long as the model does not change. This option requires additional startup time, but the speed of the subsequent simulations is comparable to `Interpreted execution`.

Tunable: No

Version History

Introduced in R2018a

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

Blocks

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

Classes

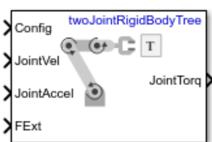
`rigidBodyTree`

Functions

`gravityTorque` | `importrobot` | `homeConfiguration` | `randomConfiguration`

Inverse Dynamics

Required joint torques for given motion



Libraries:
Robotics System Toolbox / Manipulator Algorithms

Description

The Inverse Dynamics block returns the joint torques required for the robot to maintain the specified robot state. To get the required joint torques, specify the robot configuration (joint positions), joint velocities, joint accelerations, and external forces.

Ports

Input

Config — Robot configuration
vector

Robot configuration, specified as a vector of positions for all nonfixed joints in the robot model, as set by the **Rigid body tree** parameter. You can also generate this vector for a complex robot using the `homeConfiguration` or `randomConfiguration` functions inside a Constant or MATLAB Function block.

JointVel — Joint velocities
vector

Joint velocities, specified as a vector. The number of joint velocities is equal to the degrees of freedom (number of nonfixed joints) of the robot.

JointAccel — Joint accelerations
vector

Joint accelerations, specified as a vector. The number of joint accelerations is equal to the degrees of freedom of the robot.

FExt — External force matrix
6-by-*n* matrix

External force matrix, specified as a 6-by-*n* matrix, where *n* is the number of bodies in the robot model. The matrix contains nonzero values in the rows corresponding to specific bodies. Each row is a vector of applied forces and torques that act as a wrench for that specific body. Generate this matrix using `externalForce` with a MATLAB Function block

Output

JointTorq — Joint torques
vector

Joint torques, returned as a vector. Each element corresponds to a torque applied to a specific joint. The number of joint torques is equal to the degrees of freedom (number of nonfixed joints) of the robot.

Parameters

Rigid body tree — Robot model
`twoJointRigidBodyTree` (default) | `RigidBodyTree` object

Robot model, specified as a `rigidBodyTree` object. You can also import a robot model from an URDF (Unified Robot Description Format) file using `importrobot`.

The default robot model, `twoJointRigidBodyTree`, is a robot with revolute joints and two degrees of freedom.

Simulate using — Type of simulation to run
`Interpreted execution` (default) | `Code generation`

- `Interpreted execution` — Simulate model using the MATLAB interpreter. This option shortens startup time but has a slower simulation speed than `Code generation`. In this mode, you can debug the source code of the block.
- `Code generation` — Simulate model using generated C code. The first time you run a simulation, Simulink generates C code for the block. The C code is reused for subsequent simulations, as long as the model does not change. This option requires additional startup time, but the speed of the subsequent simulations is comparable to `Interpreted execution`.

Tunable: No

Version History

Introduced in R2018a

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

Blocks

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

Classes

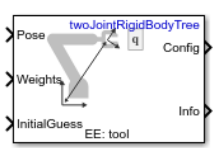
`rigidBodyTree`

Functions

`inverseDynamics` | `externalForce` | `importrobot` | `homeConfiguration` | `randomConfiguration`

Inverse Kinematics

Compute joint configurations to achieve an end-effector pose



Libraries:
Robotics System Toolbox / Manipulator Algorithms

Description

The Inverse Kinematics block uses an inverse kinematic (IK) solver to calculate joint configurations for a desired end-effector pose based on a specified rigid body tree model. Create a rigid body tree model for your robot using the `rigidBodyTree` class. The rigid body tree model defines all the joint constraints that the solver enforces.

Specify the `RigidBodyTree` parameter and the desired end effector inside the block mask. You can also tune the algorithm parameters in the **Solver Parameters** tab.

Input the desired end-effector **Pose**, the **Weights** on pose tolerance, and an **InitialGuess** for the joint configuration. The solver outputs a robot configuration, **Config**, that satisfies the end-effector pose within the tolerances specified in the **Solver Parameters** tab.

Ports

Input

Pose — End-effector pose
4-by-4 homogeneous transform

End-effector pose, specified as a 4-by-4 homogeneous transform. This transform defines the desired position and orientation of the rigid body specified in the **End effector** parameter.

Data Types: `single` | `double`

Weights — Weights for pose tolerances
six-element vector

Weights for pose tolerances, specified as a six-element vector. The first three elements of the vector correspond to the weights on the error in orientation for the desired pose. The last three elements of the vector correspond to the weights on the error in the *xyz* position for the desired pose.

Data Types: `single` | `double`

InitialGuess — Initial guess of robot configuration
vector

Initial guess of robot configuration, specified as a vector of joint positions. The number of positions is equal to the number of nonfixed joints in the **Rigid body tree** parameter. Use this initial guess to help guide the solver to a desired robot configuration. However, the solution is not guaranteed to be close to this initial guess.

Data Types: `single` | `double`

Output

Config — Robot configuration solution
vector

Robot configuration that solves the desired end-effector pose, specified as a vector. A robot configuration is a vector of joint positions for the rigid body tree model. The number of positions is equal to the number of nonfixed joints in the **Rigid body tree** parameter.

Data Types: `single` | `double`

Info — Solution information
bus

Solution information, returned as a bus. The solution information bus contains these elements:

- `Iterations` — Number of iterations run by the algorithm.
- `PoseErrorNorm` — The magnitude of the error between the pose of the end effector in the solution and 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”.
- `Status` — Character vector describing whether the solution is within the tolerance (1) or is the best possible solution the algorithm could find (2).

Parameters

Block Parameters

Rigid body tree — Rigid body tree model
`twoJointRigidBodyTree` (default) | `rigidBodyTree` object

Rigid body tree model, specified as a `rigidBodyTree` object. Create the robot model in the MATLAB workspace before specifying in the block mask.

End effector — End-effector name
`'tool'` | `Select body`

End-effector name for desired pose. To see a list of bodies on the `rigidBodyTree` object, specify the **Rigid body tree** parameter, then click **Select body**.

Show solution diagnostic outputs — Enable info port
`on` (default) | `off`

Select to enable the **Info** port and get diagnostic info for the solver solution.

Solver Parameters

Solver — Algorithm for solving inverse kinematics
`'BFGSGradientProjection'` (default) | `'LevenbergMarquardt'`

Algorithm for solving inverse kinematics, specified as either `'BFGSGradientProjection'` or `'LevenbergMarquardt'`. For details of each algorithm, see “Inverse Kinematics Algorithms”.

Enforce joint limits — Enforce rigid body tree joint limits
`on` (default) | `off`

Select to enforce the joint limits specified in the **Rigid body tree** model.

Maximum iterations — Maximum number of iterations
`1500` (default) | positive integer

Maximum number of iterations to optimize the solution, specified as a positive integer. Increasing the number of iterations can improve the solution at the cost of execution time.

Maximum time — Maximum time
`10` (default) | positive scalar

Maximum number of seconds that the algorithm runs before timing out, specified as a positive scalar. Increasing the maximum time can improve the solution at the cost of execution time.

Gradient tolerance — Threshold on gradient of cost function
`1e-7` (default) | positive scalar

Threshold on the gradient of the cost function, specified as a positive scalar. The algorithm stops if the magnitude of the gradient falls below this threshold. A low gradient magnitude usually indicates that the solver has converged to a solution.

Solution tolerance — Threshold on pose error
`1e-6` (default) | positive scalar

Threshold on the magnitude of the error between the end-effector pose generated from the solution and the desired pose, specified as a positive scalar. The `Weights` specified for each component of the pose are included in this calculation.

Step tolerance — Minimum step size
`1e-14` (default) | positive scalar

Minimum step size allowed by the solver, specified as a positive scalar. Smaller step sizes usually mean that the solution is close to convergence.

Error change tolerance — Threshold on change in pose error
`1e-12` (default) | positive scalar

Threshold on the change in end-effector pose error between iterations, specified as a positive scalar. The algorithm returns if the changes in all elements of the pose error are smaller than this threshold.

Dependencies

This parameter is enabled when the **Solver** is `Levenberg-Marquadt`.

Use error damping — Enable error damping
`on` (default) | `off`

Select the check box to enable error damping, then specify the `Damping bias` parameter.

Dependencies

This parameter is enabled when the **Solver** is `Levenberg-Marquadt`.

Damping bias — Damping on cost function
0.0025 (default) | positive scalar

Damping on cost function, specified as a positive scalar. The Levenberg-Marquadt algorithm has a damping feature controlled by this scalar that works with the cost function to control the rate of convergence.

Dependencies

This parameter is enabled when the **Solver** is `Levenberg-Marquadt` and **Use error damping** is `on`.

Simulate using — Type of simulation to run
`Interpreted execution` (default) | `Code generation`

- `Interpreted execution` — Simulate model using the MATLAB interpreter. This option shortens startup time but has a slower simulation speed than `Code generation`. In this mode, you can debug the source code of the block.
- `Code generation` — Simulate model using generated C code. The first time you run a simulation, Simulink generates C code for the block. The C code is reused for subsequent simulations, as long as the model does not change. This option requires additional startup time, but the speed of the subsequent simulations is comparable to `Interpreted execution`.

Tunable: No

Version History

Introduced in R2018b

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.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

Generated code for some hardware boards may fail to compile for models containing an Inverse Kinematics block if the **Maximum time** parameter of the Inverse Kinematics block is set to a finite value. To deploy to these boards, set **Maximum time** to `Inf`.

See Also

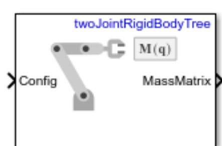
Objects
`rigidBodyTree` | `generalizedInverseKinematics` | `inverseKinematics`

Blocks
Get Transform | Inverse Dynamics

Topics
"Trajectory Control Modeling with Inverse Kinematics"
"Control PR2 Arm Movements Using ROS Actions and Inverse Kinematics"
"Inverse Kinematics Algorithms"

Joint Space Mass Matrix

Joint-space mass matrix for robot configuration



Libraries:
Robotics System Toolbox / Manipulator Algorithms

Description

The Joint Space Mass Matrix block returns the joint-space mass matrix for the given robot configuration (joint positions) for the **Rigid body tree** robot model.

Ports

Input

Config — Robot configuration
vector

Robot configuration, specified as a vector of positions for all nonfixed joints in the robot model, as set by the **Rigid body tree** parameter. You can also generate this vector for a complex robot using the `homeConfiguration` or `randomConfiguration` functions inside a Constant or MATLAB Function block.

Output

MassMatrix — Joint-space mass matrix for configuration
positive-definite symmetric matrix

Joint-space mass matrix for the given robot configuration, returned as a positive-definite symmetric matrix.

Parameters

Rigid body tree — Robot model
`twoJointRigidBodyTree` (default) | `RigidBodyTree` object

Robot model, specified as a `rigidBodyTree` object. You can also import a robot model from an URDF (Unified Robot Description Formation) file using `importrobot`.

The default robot model, `twoJointRigidBodyTree`, is a robot with revolute joints and two degrees of freedom.

Simulate using — Type of simulation to run
`Interpreted execution` (default) | `Code generation`

- `Interpreted execution` — Simulate model using the MATLAB interpreter. This option shortens startup time but has a slower simulation speed than `Code generation`. In this mode, you can debug the source code of the block.
- `Code generation` — Simulate model using generated C code. The first time you run a simulation, Simulink generates C code for the block. The C code is reused for subsequent simulations, as long as the model does not change. This option requires additional startup time, but the speed of the subsequent simulations is comparable to `Interpreted execution`.

Tunable: No

Version History

Introduced in R2018a

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

Blocks

Forward Dynamics | Inverse Dynamics | Get Jacobian | Gravity Torque | Velocity Product Torque

Classes

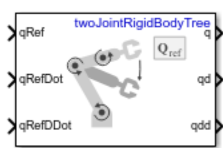
`rigidBodyTree`

Functions

`massMatrix` | `importrobot` | `homeConfiguration` | `randomConfiguration`

Joint Space Motion Model

Model rigid body tree motion given joint-space inputs



Libraries:
Robotics System Toolbox / Manipulator Algorithms

Description

The Joint Space Motion Model block models the closed-loop joint-space motion of a manipulator robot, specified as a `rigidBodyTree` object. The motion model behavior is defined by the `Motion Type` parameter.

For more details about the equations of motion, see “Joint-Space Motion Model”.

Ports

Input

qRef — Joint positions
n-element vector

n-element vector representing the desired joint positions of radians, where *n* is the number of nonfixed joints in the `rigidBodyTree` object of the `Rigid body tree` parameter.

Dependencies

To enable this port, set the `Motion Type` parameter to `Computed Torque Control`, `PD Control`, or `Independent Joint Motion`.

qRefDot — Joint velocities
n-element vector

n-element vector representing the desired joint velocities of radians per second, where *n* is the number of nonfixed joints in the `rigidBodyTree` object of the `Rigid body tree` parameter.

Dependencies

To enable this port, set the `Motion Type` parameter to `Computed Torque Control`, or `Independent Joint Motion`.

qRefDDot — Joint accelerations
n-element vector

n-element vector representing the desired joint velocities of radians per second squared, where *n* is the number of nonfixed joints in the `rigidBodyTree` object of the `Rigid body tree` parameter.

Dependencies

To enable this port, set the `Motion Type` parameter to `Computed Torque Control`, `PD Control`, or `Independent Joint Motion`.

FExt — External forces acting on system
6-by-*m* matrix

A 6-by-*m* matrix of external forces for the *m* bodies in the `rigidBodyTree` object of the `Rigid body tree` parameter.

Dependencies

To enable this port, set the `Show external force input` parameter to `on`.

Output

q — Joint positions
n-element vector

Joint positions output as an *n*-element vector in radians or meters, where *n* is the number of nonfixed joints in the `rigidBodyTree` object of the `Rigid body tree` parameter.

qd — Joint velocities
n-element vector

Joint velocities output as an *n*-element vector in radians per second or meters per second, where *n* is the number of nonfixed joints in the `rigidBodyTree` object of the `Rigid body tree` parameter

qdd — Joint accelerations
n-element vector

Joint accelerations output as an *n*-element vector in radians per second squared or meters per second squared, where *n* is the number of nonfixed joints in the `rigidBodyTree` object of the `Rigid body tree` parameter

Parameters

Rigid body tree — Robot model
`twoJointRigidBodyTree` (default) | `RigidBodyTree` object

Robot model, specified as a `rigidBodyTree` object. You can also import a robot model from an URDF (Unified Robot Description Formation) file using `importrobot`.

The default robot model, `twoJointRigidBodyTree`, is a manipulator with revolute joints and two degrees of freedom.

Motion Type — Type of motion computed by motion model
`Computed Torque Control` (default) | `Independent Joint Motion` | `PD Control` | `Open Loop Dynamics`

Type of motion, specified as a string scalar or character vector that defines the closed-loop joint-space behavior that the object models. Options are:

- `Computed Torque Control` — Compensates for full-body dynamics and assigns the error dynamics specified in the `Natural frequency` and `Damping ratio` parameters.

- `Independent Joint Motion` — Models each joint as an independent second order system using the error dynamics specified by the `Natural frequency` and `Damping ratio` parameters.
- `PD Control` — Uses proportional-derivative (PD) control on the joints based on the specified `Proportional gain` and `Derivative gain` parameters.
- `Open Loop Dynamics` — Disables inputs except for `FExt` if `Show external force input` is enabled. This is an open-loop configuration.

Specification format — Inputs to control robot
`Damping Ratio / Natural Frequency` (default) | `Step Response`

Inputs to control the robot system. Options are:

- `Damping Ratio / Natural Frequency` — Setting the natural frequency using the `Natural frequency` parameter of the system in Hz, and the damping ratio using the `Damping ratio` parameter.
- `Step Response` — Model at discrete time-steps with a fixed settling time and overshoot using the `Settling time` and the `Overshoot` parameters.

Dependencies

To enable this parameter, set the `Motion Type` parameter to `Computed Torque Control` or `Independent Joint Motion`.

Damping ratio — Damping ratio of system
`1` (default) | numeric scalar

Damping ratio use to decay system oscillations. A value of `1` results in no damping, whereas `0` fully dampens the system.

Dependencies

To enable this parameter, set the `Specification format` parameter to `Damping Ratio / Natural Frequency`.

Natural frequency — Natural frequency of system
`10` (default) | numeric scalar

Frequency of the system oscillations if unimpeded, specified in Hz.

Dependencies

To enable this parameter, set the `Specification format` parameter to `Damping Ratio / Natural Frequency`.

Settling time — Settling time of system
`0.59` (default) | numeric scalar

The time taken for each joint to reach steady state, measured in seconds.

Dependencies

To enable this parameter, set the `Specification format` parameter to `Step Response`.

Overshoot — System overshoot
`0.0` (default) | numeric scalar

The maximum value that the system exceeds the target position.

Dependencies

To enable this parameter, set the `Specification format` parameter to `Step Response`.

Proportional gain — Proportional gain for PD Control
`100` (default) | *n*-by-*n* matrix | scalar

Proportional gain for proportional-derivative (PD) control, specified as a scalar or *n*-by-*n* matrix, where *n* is the number of nonfixed joints in the `rigidBodyTree` object of the `Rigid body tree` parameter.

Dependencies

To enable this parameter, set the `Specification format` parameter to `PD Control`.

Derivative gain — Derivative gain for PD control
`10` (default) | *n*-by-*n* matrix | scalar

Derivative gain for proportional-derivative (PD) control, specified as a scalar or *n*-by-*n* matrix, where *n* is the number of nonfixed joints in the `rigidBodyTree` object of the `Rigid body tree` parameter

Dependencies

To enable this parameter, set the `Specification format` parameter to `PD Control`.

Show external force input — Display `FExt` port
`off` (default) | `on`

Enable this parameter to input external forces using the `FExt` port.

Dependencies

To enable this parameter, set the `Motion Type` parameter to `Computed Torque Control`, `PD Control`, or `Open Loop Dynamics`.

Initial joint configuration — Initial joint positions
`0` (default) | *n*-element vector | scalar

Initial joint positions, specified as a *n*-element vector or scalar in radians. *n* is the number of nonfixed joints in the `rigidBodyTree` object in the `Rigid body tree` parameter.

Initial joint velocities — Initial joint velocities
`0` (default) | *n*-element vector | scalar

Initial joint velocities, specified as a *n*-element vector or scalar in radians per second. *n* is the number of nonfixed joints in the `rigidBodyTree` object in the `Rigid body tree` parameter.

Simulate using — Type of simulation to run
`Interpreted execution` (default) | `Code generation`

- `Interpreted execution` — Simulate model using the MATLAB interpreter. For more information, see “Simulation Modes” (Simulink).
- `Code generation` — Simulate model using generated C code. The first time you run a simulation, Simulink generates C code for the block. The C code is reused for subsequent simulations, as long as the model does not change.

Tunable: No

Version History

Introduced in R2019b

References

[1] Craig, John J. *Introduction to Robotics: Mechanics and Control*. Upper Saddle River, NJ: Pearson Education, 2005.

[2] Spong, Mark W., Seth Hutchinson, and Mathukumalli Vidyasagar. *Robot Modeling and Control*. Hoboken, NJ: Wiley, 2006.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

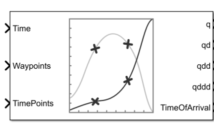
See Also

Blocks
Task Space Motion Model

Classes
`jointSpaceMotionModel` | `taskSpaceMotionModel`

Minimum Jerk Polynomial Trajectory

Generate minimum jerk polynomial trajectories through multiple waypoints



Libraries:
UAV Toolbox / Algorithms
Robotics System Toolbox / Utilities

Description

The Minimum Jerk Polynomial Trajectory block generates minimum jerk polynomial trajectories that pass through the waypoints at the times specified in time points. The block outputs positions, velocities, accelerations, jerks, and time of arrival for achieving this trajectory based on the `Time` input.

The block also accepts boundary conditions for waypoints. The block also outputs the coefficients for the polynomials and status of the trajectory generation.

The initial and final values of positions, velocities, accelerations, and jerks of the trajectory are held constant outside the time period defined in `TimePoints` input.

Ports

Input

Time — Time point along trajectory
scalar | vector

Time point along the trajectory, specified as a scalar or vector.

- When the time is specified as a scalar, this value is synced with simulation time and is used to specify the time point for sampling the trajectory. The block outputs a vector of the trajectory variables at that instance in time.
- If the time is specified as a vector, the block outputs a matrix with each column corresponding to each element of the vector.

Data Types: `single` | `double`

Waypoints — Waypoints positions along trajectory
n-by-*p* matrix

Positions of waypoints of the trajectory at given time points, specified as an *n*-by-*p* matrix. *n* is the dimension of the trajectory and *p* is the number of waypoints.

Data Types: `single` | `double`

TimePoints — Time points for waypoints of trajectory
p-element row vector

Time points for the waypoints of the trajectory, specified as a *p*-element row vector. *p* is the number of waypoints.

Data Types: `single` | `double`

VelBC — Velocity boundary conditions for waypoints
n-by-*p* matrix

Velocity boundary conditions for waypoints, specified as an *n*-by-*p* matrix. Each row sets the velocity boundary for the corresponding dimension of the trajectory *n* at each of *p* waypoints.

By default, the block uses a value of `0` at the boundary waypoints and `NaN` at the intermediate waypoints.

Dependencies

To enable this input port, select `Show boundary conditions input ports`.

Data Types: `single` | `double`

AccelBC — Acceleration boundary conditions for waypoints
n-by-*p* matrix

Acceleration boundary conditions for waypoints, specified as an *n*-by-*p* matrix. Each row sets the acceleration boundary for the corresponding dimension of the trajectory *n* at each of *p* waypoints.

By default, the block uses a value of `0` at the boundary waypoints and `NaN` at the intermediate waypoints.

Dependencies

To enable this input port, select `Show boundary conditions input ports`.

Data Types: `single` | `double`

JerkBC — Jerk boundary conditions for waypoints
n-by-*p* matrix

Jerk boundary conditions for waypoints, specified as an *n*-by-*p* matrix. Each row sets the jerk boundary for the corresponding dimension of the trajectory *n* at each of *p* waypoints.

By default, the block uses a value of `0` at the boundary waypoints and `NaN` at the intermediate waypoints.

Dependencies

To enable this input port, select `Show boundary conditions input ports`.

Data Types: `single` | `double`

Output

q — Positions of trajectory
n-element vector | *n*-by-*m* matrix

Positions of the trajectory, returned as an *n*-element vector or *n*-by-*m* matrix.

- If you specify a scalar for the `Time` input with an *n*-dimensional trajectory, the output is a vector with *n*-elements.
- If you specify a vector of *m*-elements for the `Time` input, the output is an *n*-by-*m* matrix.

Data Types: `single` | `double`

qd — Velocities of trajectory
n-element vector | *n*-by-*m* matrix

Velocities of the trajectory, returned as an *n*-element vector or *n*-by-*m* matrix.

- If you specify a scalar for the `Time` input with an *n*-dimensional trajectory, the output is a vector with *n*-elements.
- If you specify a vector of *m*-elements for the `Time` input, the output is an *n*-by-*m* matrix.

Data Types: `single` | `double`

qdd — Accelerations of trajectory
n-element vector | *n*-by-*m* matrix

Accelerations of the trajectory, returned as an *n*-element vector or *n*-by-*m* matrix.

- If you specify a scalar for the `Time` input with an *n*-dimensional trajectory, the output is a vector with *n*-elements.
- If you specify a vector of *m*-elements for the `Time` input, the output is an *n*-by-*m* matrix.

Data Types: `single` | `double`

qddd — Jerks of trajectory
n-element vector | *n*-by-*m* matrix

Jerks of the trajectory, returned as an *n*-element vector or *n*-by-*m* matrix.

- If you specify a scalar for the `Time` input with an *n*-dimensional trajectory, the output is a vector with *n*-elements.
- If you specify a vector of *m*-elements for the `Time` input, the output is an *n*-by-*m* matrix.

Data Types: `single` | `double`

TimeOfArrival — Time of arrival at each waypoint
p-element vector

Time of arrival at each waypoint, returned as a *p*-element vector. *p* is the number of waypoints.

Data Types: `single` | `double`

PolynomialCoefs — Polynomial coefficients
$n(p-1)$-by-8 matrix

Polynomial coefficients, returned as an $n(p-1)$-by-8 matrix. *n* is the dimension of the trajectory and *p* is the number of waypoints. Each set of *n* rows defines the coefficients for the polynomial that described each variable trajectory.

Dependencies

To enable this output port, select `Show polynomial coefficients output port`.

Data Types: `single` | `double`

Status — Status of trajectory generation
three-element vector of the form [*SingularityStatus MaxIterStatus MaxTimeStatus*]

Status of trajectory generation, returned as a three-element vector of the form [*SingularityStatus MaxIterStatus MaxTimeStatus*].

SingularityStatus returned as `0` or `1` indicates the occurrence of singularity. If singularity occurs reduce the `Maximum segment time` to `Minimum segment time` ratio.

MaxIterStatus returned as `0` or `1` indicates if the number of iterations for the solver has exceeded `Maximum iterations`.

MaxTimeStatus returned as `0` or `1` indicates if the time for the solver has exceeded `Maximum time`.

Dependencies

To enable this output port, select `Show status output port`.

Data Types: `uint8`

Parameters

Show boundary conditions input ports — Accept boundary condition inputs
`off` (default) | `on`

Select this parameter to input the velocity, acceleration, and jerk boundary conditions, at the `VelBC`, `AccelBC`, and `JerkBC` ports, respectively.

Tunable: No

Show polynomial coefficients output port — Output polynomial coefficients
`off` (default) | `on`

Select this parameter to output polynomial coefficients at the `PolynomialCoefs` port.

Tunable: No

Show status output port — Output status
`off` (default) | `on`

Select this parameter to output status at the `Status` port.

Tunable: No

Time allocation — Enable time allocation
`off` (default) | `on`

Enable to specify time allocation for the trajectory using the `Time weight`, `Minimum segment time`, `Maximum segment time`, `Maximum iterations`, and `Maximum time` parameters.

Tunable: No

Time weight — Weight for time allocation
`100` (default) | positive scalar

Weight for time allocation, specified as a positive scalar.

Tunable: No

Dependencies

To enable this parameter, select `Time allocation`.

Minimum segment time — Minimum time segment length
`0.1` (default) | positive scalar | (*p*–1)-element positive row vector

Minimum time segment length, specified as a positive scalar or (*p*–1)-element positive row vector. *p* is the number of waypoints.

Tunable: No

Dependencies

To enable this parameter, select `Time allocation`.

Maximum segment time — Maximum time segment length
`1` (default) | positive scalar | (*p*–1)-element positive row vector

Maximum time segment length, specified as a positive scalar or (*p*–1)-element positive row vector. *p* is the number of waypoints.

Tunable: No

Dependencies

To enable this parameter, select `Time allocation`.

Maximum iterations — Maximum iterations for solver
`1500` (default) | positive integer scalar

Maximum iterations for solver, specified as a positive integer scalar.

Tunable: No

Dependencies

To enable this parameter, select `Time allocation`.

Maximum time — Maximum time for solver
`10` (default) | positive scalar

Maximum time for solver, specified as a positive scalar.

Tunable: No

Dependencies

To enable this parameter, select `Time allocation`.

Simulate using — Type of simulation to run
`Interpreted execution` (default) | `Code generation`

Select the type of simulation to run from these options:

- `Interpreted execution` — Simulate model using the MATLAB interpreter. This option shortens startup time but has a slower simulation speed than `Code generation`. In this mode, you can debug the source code of the block.
- `Code generation` — Simulate model using generated C code. The first time you run a simulation, Simulink generates C code for the block. The C code is reused for subsequent simulations, as long as the model does not change. This option requires additional startup time but the speed of the subsequent simulations is comparable to `Interpreted execution`.

Tunable: No

Tips

For better performance, consider these options:

- Minimize the number of waypoint or parameter changes.
- Set the `Simulate using` parameter to `Code generation`. For more information, see "Simulation Modes" (Simulink).

Version History

Introduced in R2022a

References

[1] Bry, Adam, Charles Richter, Abraham Bachrach, and Nicholas Roy. "Aggressive Flight of Fixed-Wing and Quadrotor Aircraft in Dense Indoor Environments." *The International Journal of Robotics Research*, 34, no. 7 (June 2015): 969–1002.

[2] Richter, Charles, Adam Bry, and Nicholas Roy. "Polynomial Trajectory Planning for Aggressive Quadrotor Flight in Dense Indoor Environments." *Paper presented at the International Symposium of Robotics Research (ISRR 2013)*, 2013.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

Functions

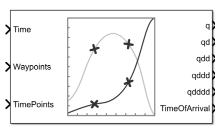
`minjerkpolytraj` | `minsnappolytraj`

Blocks

Minimum Snap Polynomial Trajectory

Minimum Snap Polynomial Trajectory

Generate minimum snap polynomial trajectories through multiple waypoints



Libraries:
UAV Toolbox / Algorithms
Robotics System Toolbox / Utilities

Description

The Minimum Snap Polynomial Trajectory block generates minimum snap polynomial trajectories that pass through the waypoints at the times specified in time points. The block outputs positions, velocities, accelerations, jerks, snap, and time of arrival for achieving this trajectory based on the `Time` input.

The block also accepts boundary conditions for waypoints. The block also outputs the coefficients for the polynomials and status of the trajectory generation.

The initial and final values of positions, velocities, accelerations, jerks, and snap of the trajectory are held constant outside the time period defined in `TimePoints` input.

Ports

Input

Time — Time point along trajectory
scalar | vector

Time point along the trajectory, specified as a scalar or vector.

- When the time is specified as a scalar, this value is synced with simulation time and is used to specify the time point for sampling the trajectory. The block outputs a vector of the trajectory variables at that instance in time.
- If the time is specified as a vector, the block outputs a matrix with each column corresponding to each element of the vector.

Data Types: `single` | `double`

Waypoints — Waypoints positions along trajectory
n-by-*p* matrix

Positions of waypoints of the trajectory at given time points, specified as an *n*-by-*p* matrix. *n* is the dimension of the trajectory and *p* is the number of waypoints.

Data Types: `single` | `double`

TimePoints — Time points for waypoints of trajectory
p-element row vector

Time points for the waypoints of the trajectory, specified as a *p*-element row vector. *p* is the number of waypoints.

Data Types: `single` | `double`

VelBC — Velocity boundary conditions for waypoints
n-by-*p* matrix

Velocity boundary conditions for waypoints, specified as an *n*-by-*p* matrix. Each row sets the velocity boundary for the corresponding dimension of the trajectory *n* at each of *p* waypoints.

By default, the block uses a value of `0` at the boundary waypoints and `NaN` at the intermediate waypoints.

Dependencies

To enable this input port, select `Show boundary conditions input ports`.

Data Types: `single` | `double`

AccelBC — Acceleration boundary conditions for waypoints
n-by-*p* matrix

Acceleration boundary conditions for waypoints, specified as an *n*-by-*p* matrix. Each row sets the acceleration boundary for the corresponding dimension of the trajectory *n* at each of *p* waypoints.

By default, the block uses a value of `0` at the boundary waypoints and `NaN` at the intermediate waypoints.

Dependencies

To enable this input port, select `Show boundary conditions input ports`.

Data Types: `single` | `double`

JerkBC — Jerk boundary conditions for waypoints
n-by-*p* matrix

Jerk boundary conditions for waypoints, specified as an *n*-by-*p* matrix. Each row sets the jerk boundary for the corresponding dimension of the trajectory *n* at each of *p* waypoints.

By default, the block uses a value of `0` at the boundary waypoints and `NaN` at the intermediate waypoints.

Dependencies

To enable this input port, select `Show boundary conditions input ports`.

Data Types: `single` | `double`

SnapBC — Snap boundary conditions for waypoints
n-by-*p* matrix

Snap boundary conditions for waypoints, specified as an *n*-by-*p* matrix. Each row sets the snap boundary for the corresponding dimension of the trajectory *n* at each of *p* waypoints.

By default, the block uses a value of `0` at the boundary waypoints and `NaN` at the intermediate waypoints.

Dependencies

To enable this input port, select `Show boundary conditions input ports`.

Data Types: `single` | `double`

Output

q — Positions of trajectory
n-element vector | *n*-by-*m* matrix

Positions of the trajectory, returned as an *n*-element vector or *n*-by-*m* matrix.

- If you specify a scalar for the `Time` input with an *n*-dimensional trajectory, the output is a vector with *n*-elements.
- If you specify a vector of *m*-elements for the `Time` input, the output is an *n*-by-*m* matrix.

Data Types: `single` | `double`

qd — Velocities of trajectory
n-element vector | *n*-by-*m* matrix

Velocities of the trajectory, returned as an *n*-element vector or *n*-by-*m* matrix.

- If you specify a scalar for the `Time` input with an *n*-dimensional trajectory, the output is a vector with *n*-elements.
- If you specify a vector of *m*-elements for the `Time` input, the output is an *n*-by-*m* matrix.

Data Types: `single` | `double`

qdd — Accelerations of trajectory
n-element vector | *n*-by-*m* matrix

Accelerations of the trajectory, returned as an *n*-element vector or *n*-by-*m* matrix.

- If you specify a scalar for the `Time` input with an *n*-dimensional trajectory, the output is a vector with *n*-elements.
- If you specify a vector of *m*-elements for the `Time` input, the output is an *n*-by-*m* matrix.

Data Types: `single` | `double`

qddd — Jerks of trajectory
n-element vector | *n*-by-*m* matrix

Jerks of the trajectory, returned as an *n*-element vector or *n*-by-*m* matrix.

- If you specify a scalar for the `Time` input with an *n*-dimensional trajectory, the output is a vector with *n*-elements.
- If you specify a vector of *m*-elements for the `Time` input, the output is an *n*-by-*m* matrix.

Data Types: `single` | `double`

qdddd — Snaps of trajectory
n-element vector | *n*-by-*m* matrix

Snaps of the trajectory, returned as an *n*-element vector or *n*-by-*m* matrix.

- If you specify a scalar for the `Time` input with an *n*-dimensional trajectory, the output is a vector with *n*-elements.

- If you specify a vector of *m*-elements for the `Time` input, the output is an *n*-by-*m* matrix.

Data Types: `single` | `double`

TimeOfArrival — Time of arrival at each waypoint
p-element vector

Time of arrival at each waypoint, returned as a *p*-element vector. *p* is the number of waypoints.

Data Types: `single` | `double`

PolynomialCoefs — Polynomial coefficients
n(*p*–1)-by-10 matrix

Polynomial coefficients, returned as an *n*(*p*–1)-by-10 matrix. *n* is the dimension of the trajectory and *p* is the number of waypoints. Each set of *n* rows defines the coefficients for the polynomial that described each variable trajectory.

Dependencies

To enable this output port, select `Show polynomial coefficients output port`.

Data Types: `single` | `double`

Status — Status of trajectory generation
three-element vector of the form [*SingularityStatus MaxIterStatus MaxTimeStatus*]

Status of trajectory generation, returned as a three-element vector of the form [*SingularityStatus MaxIterStatus MaxTimeStatus*].

SingularityStatus returned as `0` or `1` indicates the occurrence of singularity. If singularity occurs reduce the `Maximum segment time` to `Minimum segment time` ratio.

MaxIterStatus returned as `0` or `1` indicates if the number of iterations for the solver has exceeded `Maximum iterations`.

MaxTimeStatus returned as `0` or `1` indicates if the time limit for the solver has exceeded `Maximum time`.

Dependencies

To enable this output port, select `Show status output port`.

Data Types: `uint8`

Parameters

Show boundary conditions input ports — Accept boundary condition inputs
`off` (default) | `on`

Select this parameter to input the velocity, acceleration, jerk, and snap boundary conditions, at the `VelBC`, `AccelBC`, `JerkBC`, and `SnapBC` ports, respectively.

Tunable: No

Show polynomial coefficients output port — Output polynomial coefficients
`off` (default) | `on`

Select this parameter to output polynomial coefficients at the `PolynomialCoefs` port.

Tunable: No

Show status output port — Output status
`off` (default) | `on`

Select this parameter to output status at the `Status` port.

Tunable: No

Time allocation — Enable time allocation
`off` (default) | `on`

Enable to specify time allocation for the trajectory using the `Time weight`, `Minimum segment time`, `Maximum segment time`, `Maximum iterations`, and `Maximum time` parameters.

Tunable: No

Time weight — Weight for time allocation
`100` (default) | positive scalar

Weight for time allocation, specified as a positive scalar.

Tunable: No

Dependencies

To enable this parameter, select `Time allocation`.

Minimum segment time — Minimum time segment length
`0.1` (default) | positive scalar | (p–1)-element positive row vector

Minimum time segment length, specified as a positive scalar or (p–1)-element positive row vector. p is the number of waypoints.

Tunable: No

Dependencies

To enable this parameter, select `Time allocation`.

Maximum segment time — Maximum time segment length
`1` (default) | positive scalar | (p–1)-element positive row vector

Maximum time segment length, specified as a positive scalar or (p–1)-element positive row vector. p is the number of waypoints.

Tunable: No

Dependencies

To enable this parameter, select `Time allocation`.

Maximum iterations — Maximum iterations for solver
`1500` (default) | positive integer scalar

Maximum iterations for solver, specified as a positive integer scalar.

Tunable: No

Dependencies

To enable this parameter, select `Time allocation`.

Maximum time — Maximum time for solver
`10` (default) | positive scalar

Maximum time for solver, specified as a positive scalar.

Tunable: No

Dependencies

To enable this parameter, select `Time allocation`.

Simulate using — Type of simulation to run
`Interpreted execution` (default) | `Code generation`

Select the type of simulation to run from these options:

- `Interpreted execution` — Simulate model using the MATLAB interpreter. This option shortens startup time but has a slower simulation speed than `Code generation`. In this mode, you can debug the source code of the block.
- `Code generation` — Simulate model using generated C code. The first time you run a simulation, Simulink generates C code for the block. The C code is reused for subsequent simulations, as long as the model does not change. This option requires additional startup time but the speed of the subsequent simulations is comparable to `Interpreted execution`.

Tunable: No

Tips

For better performance, consider these options:

- Minimize the number of waypoint or parameter changes.
- Set the `Simulate using` parameter to `Code generation`. For more information, see “Simulation Modes” (Simulink).

Version History

Introduced in R2022a

References

[1] Bry, Adam, Charles Richter, Abraham Bachrach, and Nicholas Roy. “Aggressive Flight of Fixed-Wing and Quadrotor Aircraft in Dense Indoor Environments.” *The International Journal of Robotics Research*, 34, no. 7 (June 2015): 969–1002.

[2] Richter, Charles, Adam Bry, and Nicholas Roy. “Polynomial Trajectory Planning for Aggressive Quadrotor Flight in Dense Indoor Environments." *Paper presented at the International Symposium of Robotics Research (ISRR 2013)*, 2013.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

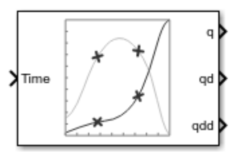
See Also

Functions
`minjerkpolytraj` | `minsnappolytraj`

Blocks
Minimum Jerk Polynomial Trajectory

Polynomial Trajectory

Generate polynomial trajectories through waypoints



Libraries:
Robotics System Toolbox / Utilities

Description

The Polynomial Trajectory block generates trajectories to travel through waypoints at the given time points using either cubic, quintic, or B-spline polynomials. The block outputs positions, velocities, and accelerations for achieving this trajectory based on the **Time** input. For B-spline polynomials, the waypoints actually define the control points for the convex hull of the B-spline instead of the actual waypoints, but the first and last waypoint are still met.

The initial and final values are held constant outside the time period defined in **Time points**.

Ports

Input

Time — Time point along trajectory
scalar | vector

Time point along the trajectory, specified as a scalar or vector. In general, when specified as a scalar, this value is synced with simulation time and is used to specify the time point for sampling the trajectory. The block outputs a vector of the trajectory variables at that instant in time. If the time is specified as a vector, the block outputs a matrix with each column corresponding to each element of the vector.

Data Types: `single` | `double`

Waypoints — Waypoint positions along trajectory
n-by-*p* matrix

Positions of waypoints of the trajectory at given time points, specified as an *n*-by-*p* matrix, where *n* is the dimension of the trajectory and *p* is the number of waypoints. If you specify the **Method** as `B-spline`, these waypoints actually define the control points for the convex hull of the B-spline, but the first and last waypoint are still met.

Dependencies

To enable this input, set **Waypoint Source** to `External`.

TimePoints — Time points for waypoints of trajectory
p-element vector

Time points for waypoints of trajectory, specified as a *p*-element vector.

Dependencies

To enable this input, set **Waypoint Source** to `External`.

VelBC — Velocity boundary conditions for waypoints
n-by-*p* matrix

Velocity boundary conditions for waypoints, specified as an *n*-by-*p* matrix. Each row corresponds to the velocity at each of the *p* waypoints for the respective variable in the trajectory.

Dependencies

To enable this input, set **Method** to `Cubic Polynomial` or `Quintic Polynomial` and **Parameter Source** to `External`.

AccelBC — Acceleration boundary conditions for trajectory
n-by-*p* matrix

Acceleration boundary conditions for waypoints, specified as an *n*-by-*p* matrix. Each row corresponds to the acceleration at each of the *p* waypoints for the respective variable in the trajectory.

Dependencies

To enable this parameter, set **Method** to `Quintic Polynomial` and **Parameter Source** to `External`.

Output

q — Position of trajectory
scalar | vector | matrix

Position of the trajectory, specified as a scalar, vector, or matrix. If you specify a scalar for the **Time** input with an *n*-dimensional trajectory, the output is a vector with *n* elements. If you specify a vector of *m* elements for the **Time** input, the output is an *n*-by-*m* matrix.

Data Types: `single` | `double`

qd — Velocity of trajectory
scalar | vector | matrix

Velocity of the trajectory, specified as a scalar, vector, or matrix. If you specify a scalar for the **Time** input with an *n*-dimensional trajectory, the output is a vector with *n* elements. If you specify a vector of *m* elements for the **Time** input, the output is an *n*-by-*m* matrix.

Data Types: `single` | `double`

qdd — Acceleration of trajectory
scalar | vector | matrix

Acceleration of the trajectory, specified as a scalar, vector, or matrix. If you specify a scalar for the **Time** input with an *n*-dimensional trajectory, the output is a vector with *n* elements. If you specify a vector of *m* elements for the **Time** input, the output is an *n*-by-*m* matrix.

Data Types: `single` | `double`

Parameters

Waypoint source — Source for waypoints
`Internal` (default) | `External`

Specify `External` to specify the **Waypoints** and **Time points** parameters as block inputs instead of block parameters.

Waypoints — Waypoint positions along trajectory
n-by-*p* matrix

Positions of waypoints of the trajectory at given time points, specified as an *n*-by-*p* matrix, where *n* is the dimension of the trajectory and *p* is the number of waypoints. If you specify the **Method** as `B-spline`, these waypoints actually define the control points for the convex hull of the B-spline, but the first and last waypoint are still met.

Dependencies

To specify this parameter in the block mask, set **Waypoint Source** to `Internal`.

Time points — Time points for waypoints of trajectory
p-element vector

Time points for waypoints of trajectory, specified as a *p*-element vector, where *p* is the number of waypoints.

Dependencies

To specify this parameter in the block mask, set **Waypoint Source** to `Internal`.

Method — Method for trajectory generation
`Cubic Polynomial` (default) | `Quintic Polynomial` | `B-Spline`

Method for trajectory generation, specified as either `Cubic Polynomial`, `Quintic Polynomial`, or `B-Spline`.

Parameter source — Source for waypoints
`Internal` (default) | `External`

Specify `External` to specify the **Velocity boundary conditions** and **Acceleration boundary conditions** parameters as block inputs instead of block parameters.

Velocity boundary conditions — Velocity boundary conditions for waypoints
`zeroes(2,5)` (default) | *n*-by-*p* matrix

Velocity boundary conditions for waypoints, specified as an *n*-by-*p* matrix. Each row corresponds to the velocity at each of the *p* waypoints for the respective variable in the trajectory.

Dependencies

To enable this input, set **Method** to `Cubic Polynomial` or `Quintic Polynomial`.

Acceleration boundary conditions — Acceleration boundary conditions for trajectory
n-by-*p* matrix

Acceleration boundary conditions for waypoints, specified as an *n*-by-*p* matrix. Each row corresponds to the acceleration at each of the *p* waypoints for the respective variable in the trajectory.

Dependencies

To enable this parameter, set **Method** to `Quintic Polynomial`.

Simulate using — Type of simulation to run
`Interpreted execution` (default) | `Code generation`

- `Interpreted execution` — Simulate model using the MATLAB interpreter. This option shortens startup time but has a slower simulation speed than `Code generation`. In this mode, you can debug the source code of the block.
- `Code generation` — Simulate model using generated C code. The first time you run a simulation, Simulink generates C code for the block. The C code is reused for subsequent simulations, as long as the model does not change. This option requires additional startup time, but the speed of the subsequent simulations is comparable to `Interpreted execution`.

Tips

For better performance, consider these options:

- Minimize the number of waypoint or parameter changes.
- Set the **Waypoint source** parameter to `Internal`.
- Set the **Simulate using** parameter to `Code generation`. For more information, see “Simulation Modes” (Simulink).

Version History

Introduced in R2019a

References

[1] Farin, Gerald E. *Curves and Surfaces for Computer Aided Geometric Design: A Practical Guide*. San Diego, CA: Academic Press, 1993.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

Blocks

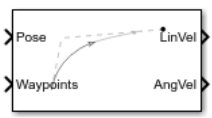
Rotation Trajectory | Transform Trajectory | Trapezoidal Velocity Profile Trajectory

Functions

`bsplinepolytraj` | `cubicpolytraj` | `quinticpolytraj` | `rottraj` | `transformtraj` | `trapveltraj`

Pure Pursuit

Linear and angular velocity control commands



Libraries:
Robotics System Toolbox / Mobile Robot Algorithms
Navigation Toolbox / Control Algorithms

Description

The Pure Pursuit block computes linear and angular velocity commands for following a path using a set of waypoints and the current pose of a differential drive vehicle. The block takes updated poses to update velocity commands for the vehicle to follow a path along a desired set of waypoints. Use the **Max angular velocity** and **Desired linear velocity** parameters to update the velocities based on the performance of the vehicle.

The **Lookahead distance** parameter computes a look-ahead point on the path, which is an instantaneous local goal for the vehicle. The angular velocity command is computed based on this point. Changing **Lookahead distance** has a significant impact on the performance of the algorithm. A higher look-ahead distance results in a smoother trajectory for the vehicle, but can cause the vehicle to cut corners along the path. Too low of a look-ahead distance can result in oscillations in tracking the path, causing unstable behavior. For more information on the pure pursuit algorithm, see “Pure Pursuit Controller”.

Input/Output Ports

Input

Pose — Current vehicle pose
`[x y theta]` vector

Current vehicle pose, specified as an `[x y theta]` vector, which corresponds to the *x*-*y* position and orientation angle, *theta*. Positive angles are measured counterclockwise from the positive *x*-axis.

Waypoints — Waypoints
[ ] (default) | *n*-by-2 array

Waypoints, specified as an *n*-by-2 array of `[x y]` pairs, where *n* is the number of waypoints. You can generate the waypoints using path planners like `mobileRobotPRM` or specify them as an array in Simulink.

Output

LinVel — Linear velocity
scalar in meters per second

Linear velocity, returned as a scalar in meters per second.

Data Types: `double`

AngVel — Angular velocity
scalar in radians per second

Angular velocity, returned as a scalar in radians per second.

Data Types: `double`

TargetDir — Target direction for vehicle
scalar in radians

Target direction for the vehicle, returned as a scalar in radians. The forward direction of the vehicle is considered zero radians, with positive angles measured counterclockwise. This output can be used as the input to the **TargetDir** port for the Vector Field Histogram block.

Dependencies

To enable this port, select the **Show TargetDir output port** parameter.

Parameters

Desired linear velocity (m/s) — Linear velocity
`0.1` (default) | scalar

Desired linear velocity, specified as a scalar in meters per second. The controller assumes that the vehicle drives at a constant linear velocity and that the computed angular velocity is independent of the linear velocity.

Maximum angular velocity (rad/s) — Angular velocity
`1.0` (default) | scalar

Maximum angular velocity, specified as a scalar in radians per second. The controller saturates the absolute angular velocity output at the given value.

Lookahead distance (m) — Look-ahead distance
`1.0` (default) | scalar

Look-ahead distance, specified as a scalar in meters. The look-ahead distance changes the response of the controller. A vehicle with a higher look-ahead distance produces smooth paths but takes larger turns at corners. A vehicle with a smaller look-ahead distance follows the path closely and takes sharp turns, but oscillate along the path. For more information on the effects of look-ahead distance, see “Pure Pursuit Controller”.

Show TargetDir output port — Target direction indicator
off (default) | on

Select this parameter to enable the **TargetDir** out port. This port gives the target direction as an angle in radians from the forward position, with positive angles measured counterclockwise.

Simulate using — Type of simulation to run
`Code generation` (default) | `Interpreted execution`

- `Interpreted execution` — Simulate model using the MATLAB interpreter. This option shortens startup time but has a slower simulation speed than `Code generation`. In this mode, you can debug the source code of the block.

- `Code generation` — Simulate model using generated C code. The first time you run a simulation, Simulink generates C code for the block. The C code is reused for subsequent simulations, as long as the model does not change. This option requires additional startup time, but the speed of the subsequent simulations is comparable to `Interpreted execution`.

Tunable: No

Version History

Introduced in R2019b

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

Classes

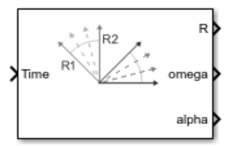
`binaryOccupancyMap` | `occupancyMap` | `mobileRobotPRM`

Topics

"Path Following for a Differential Drive Robot"
"Plan Path for a Differential Drive Robot in Simulink"
"Path Following with Obstacle Avoidance in Simulink®" (Navigation Toolbox)
"Pure Pursuit Controller"

Rotation Trajectory

Generate trajectory between two orientations



Libraries:
Robotics System Toolbox / Utilities

Description

The Rotation Trajectory block generates an interpolated trajectory between two rotation matrices. The block outputs the rotation at the times given by the **Time** input, which can be a scalar or vector.

The trajectory is computed using quaternion spherical linear interpolation (SLERP) and finds the shortest path between points. Select the **Use custom time scaling** check box to compute using a custom time scaling. The block uses linear time scaling by default.

The initial and final values are held constant outside the time period defined in the **Time interval** parameter.

Ports

Input

Time — Time point along trajectory
scalar | vector

Time point along the trajectory, specified as a scalar or vector. In general, when specified as a scalar, this value is synced with simulation time and is used to specify the time point for sampling the trajectory. The block outputs a vector of the trajectory variables at that instant in time. If the time is specified as a vector, the block outputs a matrix with each column corresponding to each element of the vector.

Data Types: `single` | `double`

R0 — Initial orientation
four-element quaternion vector | 3-by-3 rotation matrix

Initial orientation, specified as a four-element quaternion vector or 3-by-3 rotation matrix. The function generates a trajectory that starts at the initial orientation, **R0**, and goes to the final orientation, **RF**.

Example: `[1 0 0 0]'`

Dependencies

To enable this input, set the **Waypoint source** to `External`.

To specify quaternions, set **Rotation Format** parameter to `Quaternion`.

To specify rotation matrices, set **Rotation Format** parameter to `Rotation`.

Data Types: `single` | `double`

RF — Final orientation
four-element vector | 3-by-3 rotation matrix

Initial orientation, specified as a four-element vector or 3-by-3 rotation matrix. The function generates a trajectory that starts at the initial orientation, **R0**, and goes to the final orientation, **RF**.

Example: `[0 0 1 0]'`

Dependencies

To enable this input, set the **Waypoint source** to `External`.

To specify quaternions, set **Rotation Format** parameter to `Quaternion`.

To specify rotation matrices, set **Rotation Format** parameter to `Rotation`.

Data Types: `single` | `double`

TimeInterval — Start and end times for trajectory
two-element vector

Start and end times for the trajectory, specified as a two-element vector.

Example: `[0 10]`

Dependencies

To enable this input, set the **Waypoint source** to `External`.

Data Types: `single` | `double`

TSTime — Time scaling time points
scalar | *p*-element vector

Time scaling time points, specified as a scalar or n *p*-element vector, where *p* is the number of points for time scaling. By default, the time scaling is a linear time scaling spanning the **TimeInterval**. Specify the actual time scaling values in **TimeScaling**.

If the **Time** input is specified at a time not specified by these points, interpolation is used to find the right scaling time.

Dependencies

To enable this parameter, select the **Use custom time scaling** check box and set **Parameter source** to `External`.

To specify a scalar, the **Time** input must be a scalar.

Data Types: `single` | `double`

TimeScaling — Time scaling vector and first two derivatives
three-element vector | 3-by-*p* matrix

Time scaling vector and its first two derivatives, specified as a three element vector or a 3-by-*p* matrix, where *m* is the length of **TSTime**. By default, the time scaling is a linear time scaling spanning the **TimeInterval**.

For a nonlinear time scaling, specify the values of the time points in the first row. The second and third rows are the velocity and acceleration of the time points, respectively. For example, to follow the path with a linear velocity to the halfway point, and then jump to the end, the time-scaling would be:

```
s(1,:) = [0 0.25 0.5 1 1 1] % Position
s(2,:) = [1    1   1 0 0 0] % Velocity
s(3,:) = [0    0   0 0 0 0] % Acceleration
```

Dependencies

To enable this parameter, select the **Use custom time scaling** check box and set **Parameter source** to `External`.

To specify a three-element vector, the **Time** and **TSTime** inputs must be a scalar.

Data Types: `single` | `double`

Output

R — Orientation vectors
4-by-*m* quaternion array | 3-by-3-by-*m* rotation matrix array

Orientation vectors, returned as a 4-by-*m* quaternion array or 3-by-3-by-*m* rotation matrix array, where *m* is the number of points in the input to **Time**.

Dependencies

To get a quaternion array, set **Rotation Format** parameter to `Quaternion`.

To get a rotation matrix array, set **Rotation Format** parameter to `Rotation`.

omega — Orientation angular velocity
3-by-*m* matrix

Orientation angular velocity, returned as a 3-by-*m* matrix, where *m* is the number of points in the input to **Time**.

alpha — Orientation angular acceleration
3-by-*m* matrix

Orientation angular acceleration, returned as a 3-by-*m* matrix, where *m* is the number of points in the input to **Time**.

Parameters

Rotation format — Format for orientations
`Quaternion` (default) | `Rotation Matrix`

Select `Rotation Matrix` to specify the **Initial rotation** and **Final rotation** as 3-by-3 rotation matrices and get the orientation output (port **R**) as a rotation matrix array. By default, the initial and final rotations are specified as four-element quaternion vectors.

Waypoint source — Source for waypoints
`Internal` (default) | `External`

Specify `External` to specify the **Initial rotation**, **Final rotation**, and **Time interval** parameters as block inputs instead of block parameters.

Initial rotation — Initial orientation
`[1 0 0 0]'` (default) | four-element quaternion vector | 3-by-3 rotation matrix

Initial orientation, specified as a four-element quaternion vector or 3-by-3 rotation matrix. The function generates a trajectory that starts at the **Initial rotation** and goes to the **Final rotation**.

Dependencies

To specify quaternions, set **Rotation Format** parameter to `Quaternion`.

To specify rotation matrices, set **Rotation Format** parameter to `Rotation`.

Data Types: `single` | `double`

Final rotation — Final orientation
`[0 0 1 0]'` (default) | four-element vector | 3-by-3 rotation matrix

Final orientation, specified as a four-element vector or 3-by-3 rotation matrix. The function generates a trajectory that starts at the **Initial rotation** and goes to the **Final rotation**.

Dependencies

To specify quaternions, set **Rotation Format** parameter to `Quaternion`.

To specify rotation matrices, set **Rotation Format** parameter to `Rotation`.

Data Types: `single` | `double`

Time interval — Start and end times for trajectory
`[0 10]` (default) | two-element vector

Start and end times for the trajectory, specified as a two-element vector.

Data Types: `single` | `double`

Use custom time scaling — Enable custom time scaling
`off` (default) | `on`

Enable to specify custom time scaling for the trajectory using the **Parameter Source**, **Time scaling time**, and **Time scaling values** parameters.

Parameter source — Source for waypoints
`Internal` (default) | `External`

Specify `External` to specify the **Time scaling time** and **Time scaling values** parameters as block inputs instead of block parameters.

Dependencies

To enable this parameter, select the **Use custom time scaling** check box.

Time scaling time — Time scaling time points
`2:0.1:3` (default) | scalar | *p*-element vector

Time scaling time points, specified as a scalar or *p*-element vector, where *p* is the number of points for time scaling. By default, the time scaling is a linear time scaling spanning the **Time interval**. Specify the actual time scaling values in **Time scaling values**.

If the **Time** input is specified at a time not specified by these points, interpolation is used to find the right scaling time.

Dependencies

To enable this parameter, select the **Use custom time scaling** check box.

To specify a scalar, the **Time** input must be a scalar.

Data Types: `single` | `double`

Time scaling values — Time scaling vector and first two derivatives
`[0:0.1:1; ones(1,11); zeros(1,11)]` (default) | three-element vector | 3-by-*m* matrix

Time scaling vector and its first two derivatives, specified as a three-element vector or 3-by-*p* matrix, where *p* is the length of **Time scaling time**. By default, the time scaling is a linear time scaling spanning the **Time interval**.

For a nonlinear time scaling, specify the values of the time points in the first row. The second and third rows are the velocity and acceleration of the time points, respectively. For example, to follow the path with a linear velocity to the halfway point, and then jump to the end, the time-scaling would be:

```
s(1,:) = [0 0.25 0.5 1 1 1] % Position
s(2,:) = [1    1   1 0 0 0] % Velocity
s(3,:) = [0    0   0 0 0 0] % Acceleration
```

Dependencies

To enable this parameter, select the **Use custom time scaling** check box.

To specify a three-element vector, the **Time** and **TSTime** inputs must be a scalar.

Data Types: `single` | `double`

Simulate using — Type of simulation to run
`Interpreted execution` (default) | `Code generation`

- `Interpreted execution` — Simulate model using the MATLAB interpreter. This option shortens startup time but has a slower simulation speed than `Code generation`. In this mode, you can debug the source code of the block.
- `Code generation` — Simulate model using generated C code. The first time you run a simulation, Simulink generates C code for the block. The C code is reused for subsequent simulations, as long as the model does not change. This option requires additional startup time, but the speed of the subsequent simulations is comparable to `Interpreted execution`.

Tunable: No

Tips

For better performance, consider these options:

- Minimize the number of waypoint or parameter changes.
- Set the **Waypoint source** parameter to `Internal`.
- Set the **Simulate using** parameter to `Code generation`. For more information, see “Simulation Modes” (Simulink).

Version History

Introduced in R2019a

References

[1] Lynch, Kevin M., and Frank C. Park. *Modern Robotics: Mechanics, Planning, and Control*. Cambridge University Press, 2017.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

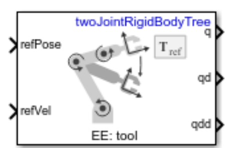
See Also

Blocks
Polynomial Trajectory | Transform Trajectory | Trapezoidal Velocity Profile Trajectory

Functions
`bsplinepolytraj` | `cubicpolytraj` | `quinticpolytraj` | `rottraj` | `transformtraj` | `trapveltraj`

Task Space Motion Model

Model rigid body tree motion given task-space inputs



Libraries:
Robotics System Toolbox / Manipulator Algorithms

Description

The Task Space Motion Model block models the closed-loop task-space motion of a manipulator, specified as a `rigidBodyTree` object. The motion model behavior is defined using proportional-derivative (PD) control.

For more details about the equations of motion, see "Task-Space Motion Model".

Ports

Input

refPose — End-effector pose
4-by-4 matrix

Homogenous transformation matrix representing the desired end effector pose, specified in meters.

refVel — Joint velocities
6-element vector

6-element vector representing the desired linear and angular velocities of the end effector, specified in meters per second and radians per second.

FExt — External forces
6-by-*m* matrix

6-by-*m* matrix representing external forces, specified in meters per second. *m* is the number of bodies in the `rigidBodyTree` object in the `Rigid body tree` parameter.

Dependencies

To enable this port, set the `Show external force input` parameter to `on`.

Output

q — Joint positions
n-element vector

Joint positions output as an *n*-element vector in radians or meters, where *n* is the number of nonfixed joints in the `rigidBodyTree` object in the `Rigid body tree` parameter.

qd — Joint velocities
n-element

Joint velocities output as an *n*-element vector in radians per second or meters per second, where *n* is the number of nonfixed joints in the `rigidBodyTree` object in the `Rigid body tree` parameter.

qdd — Joint accelerations
n-element

Joint accelerations output as an *n*-element in radians per second squared or meters per second squared, where *n* is the number of nonfixed joints in the `rigidBodyTree` object in the `Rigid body tree` parameter.

Parameters

Rigid body tree — Rigid body tree
`twoJointRigidBodyTree` object (default) | `RigidBodyTree` object

Robot model, specified as a `RigidBodyTree` object. You can also import a robot model from an URDF (Unified Robot Description Formation) file using `importrobot`.

The default robot model, `twoJointRigidBodyTree`, is a robot with revolute joints and two degrees of freedom.

End effector — End effector body
`tool` (default)

This parameter defines the body that will be used as the end effector, and for which the task space motion is defined. The property must correspond to a body name in the `rigidBodyTree` object of the property. Click **Select body** to select a body from the `rigidBodyTree`. If the `rigidBodyTree` is updated without also updating the end effector, the body with the highest index is assigned by default.

Proportional gain — Proportional gain for PD Control
`500*eye(6)` (default) | 6-by-6 matrix

Proportional gain for proportional-derivative (PD) control, specified as a 6-by-6 matrix.

Derivative gain — Derivative gain for PD Control
`100*eye(6)` (default) | 6-by-6 matrix

Derivative gain for proportional-derivative (PD) control, specified as a 6-by-6 matrix.

Joint damping — Damping ratios
`[1 1]` (default) | *n*-element vector | scalar

Damping ratios on each joint, specified as a scalar or *n*-element vector, where *n* is the number of nonfixed joints in the `rigidBodyTree` object in the `Rigid body tree` parameter.

Show external force input — Display `FExt` port
`off` (default) | `on`

Click the check-box to enable this parameter to input external forces using the `FExt` port.

Initial joint configuration — Initial joint positions
`0` (default) | *n*-element vector | scalar

Initial joint positions, specified as a *n*-element vector or scalar in radians. *n* is the number of nonfixed joints in the `rigidBodyTree` object in the `Rigid body tree` parameter.

Initial joint velocities — Initial joint velocities
`0` (default) | *n*-element vector | scalar

Initial joint velocities, specified as a *n*-element vector or scalar in radians per second. *n* is the number of nonfixed joints in the `rigidBodyTree` object in the `Rigid body tree` parameter.

Simulate using — Type of simulation to run
`Interpreted execution` (default) | `Code generation`

- `Interpreted execution` — Simulate model using the MATLAB interpreter. For more information, see "Simulation Modes" (Simulink).
- `Code generation` — Simulate model using generated C code. The first time you run a simulation, Simulink generates C code for the block. The C code is reused for subsequent simulations, as long as the model does not change.

Tunable: No

Version History

Introduced in R2019b

References

[1] Craig, John J. *Introduction to Robotics: Mechanics and Control*. Upper Saddle River, NJ: Pearson Education, 2005.

[2] Spong, Mark W., Seth Hutchinson, and Mathukumalli Vidyasagar. *Robot Modeling and Control*. Hoboken, NJ: Wiley, 2006.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

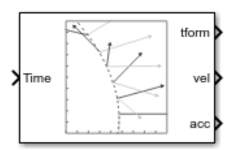
See Also

Blocks
Joint Space Motion Model

Classes
`taskSpaceMotionModel` | `jointSpaceMotionModel`

Transform Trajectory

Generate trajectory between two homogeneous transforms



Libraries:
Robotics System Toolbox / Utilities

Description

The Transform Trajectory block generates an interpolated trajectory between two homogenous transformation matrices. The block outputs the transform at the times given by the **Time** input, which can be a scalar or vector.

The trajectory is computed using quaternion spherical linear interpolation (SLERP) for the rotation and linear interpolation for the translation. This method finds the shortest path between positions and rotations of the transformation. Select the **Use custom time scaling** check box to compute the trajectory using a custom time scaling. The block uses linear time scaling by default.

The initial and final values are held constant outside the time period defined in **Time interval**.

Ports

Input

Time — Time point along trajectory
scalar | vector

Time point along trajectory, specified as a scalar or vector. In general, when specified as a scalar, this value is synced with simulation time and is used to specify the time point for sampling the trajectory. The block outputs a vector of the trajectory variables at that instant in time. If the time is specified as a vector, the block outputs a matrix with each column corresponding to each element of the vector.

Data Types: `single` | `double`

T0 — Initial transformation matrix
4-by-4 homogeneous transformation

Initial transformation matrix, specified as a 4-by-4 homogeneous transformation. The function generates a trajectory that starts at the initial orientation, **T0**, and goes to the final orientation, **TF**.

Example: `trvec2tform([1 10 -1])`

Dependencies

To enable this parameter, set the **Waypoint source** to `External`.

Data Types: `single` | `double`

TF — Final transformation matrix
4-by-4 homogeneous transformation

Final transformation matrix, specified as a 4-by-4 homogeneous transformation. The function generates a trajectory that starts at the initial orientation, **T0**, and goes to the final orientation, **TF**.

Example: `trvec2tform([1 10 -1])`

Dependencies

To enable this parameter, set the **Waypoint source** to `External`.

Data Types: `single` | `double`

TimeInterval — Start and end times for trajectory
two-element vector

Start and end times for the trajectory, specified as a two-element vector.

Example: `[0 10]`

Dependencies

To enable this parameter, set the **Waypoint source** to `External`.

Data Types: `single` | `double`

TSTime — Time scaling time points
scalar | *p*-element vector

Time scaling time points, specified as a scalar or n *p*-element vector, where *p* is the number of points for time scaling. By default, the time scaling is a linear time scaling spanning the **TimeInterval**. Specify the actual time scaling values in **TimeScaling**.

If the **Time** input is specified at a time not specified by these points, interpolation is used to find the right scaling time.

Dependencies

To enable this parameter, select the **Use custom time scaling** check box and set **Parameter source** to `External`.

To specify a scalar, the **Time** input must be a scalar.

Data Types: `single` | `double`

TimeScaling — Time scaling vector and first two derivatives
three-element vector | 3-by-*p* matrix

Time scaling vector and its first two derivatives, specified as a three element vector or a 3-by-*p* matrix, where *m* is the length of **TSTime**. By default, the time scaling is a linear time scaling spanning the **TimeInterval**.

For a nonlinear time scaling, specify the values of the time points in the first row. The second and third rows are the velocity and acceleration of the time points, respectively. For example, to follow the path with a linear velocity to the halfway point, and then jump to the end, the time-scaling would be:

```
s(1,:) = [0 0.25 0.5 1 1 1] % Position
s(2,:) = [1    1   1 0 0 0] % Velocity
s(3,:) = [0    0   0 0 0 0] % Acceleration
```

Dependencies

To enable this parameter, select the **Use custom time scaling** check box and set **Parameter source** to `External`.

To specify a three-element vector, the **Time** and **TSTime** inputs must be a scalar.

Data Types: `single` | `double`

Output

tform — Homogeneous transformation matrices
4-by-4-by-*m* homogenous matrix array

Homogeneous transformation matrices, returned as a 4-by-4-by-*m* homogenous matrix array, where *m* is the number of points input to **Time**.

vel — Transform velocities
6-by-*m* matrix

Transform velocities, returned as a 6-by-*m* matrix, where *m* is the number of points input to **Time**. Each row of the vector is the angular and linear velocity of the transform as `[wx wy wz vx vy vz]`. *w* represents an angular velocity and *v* represents a linear velocity.

alpha — Transform accelerations
6-by-*m* matrix

Transform velocities, returned as a 6-by-*m* matrix, where *m* is the number of points input to **Time**. Each row of the vector is the angular and linear acceleration of the transform as `[alphax alphay alphaz ax ay az]`. *alpha* represents an angular acceleration and *a* represents a linear acceleration.

Parameters

Waypoint source — Source for waypoints
`Internal` (default) | `External`

Specify `External` to specify the **Waypoints** and **Time points** parameters as block inputs instead of block parameters.

Initial transform — Initial transformation matrix
`trvec2tform([1 10 -1])` (default) | 4-by-4 homogeneous transformation

Initial transformation matrix, specified as a 4-by-4 homogeneous transformation. The function generates a trajectory that starts at the **Initial transform** and goes to the **Final transform**.

Data Types: `single` | `double`

Final transform — Final transformation matrix
`eul2tform([0 pi pi/2])` (default) | 4-by-4 homogeneous transformation

Final transformation matrix, specified as a 4-by-4 homogeneous transformation. The function generates a trajectory that starts at the **Initial transform** and goes to the **Final transform**.

Data Types: `single` | `double`

Time interval — Start and end times for trajectory
`[2 3]` | two-element vector

Start and end times for the trajectory, specified as a two-element vector in seconds.

Data Types: `single` | `double`

Use custom time scaling — Enable custom time scaling
`off` (default) | `on`

Enable to specify custom time scaling for the trajectory using the **Parameter Source**, **Time scaling time**, and **Time scaling values** parameters.

Parameter source — Source for waypoints
`Internal` (default) | `External`

Specify `External` to specify the **Time scaling time** and **Time scaling values** parameters as block inputs instead of block parameters.

Dependencies

To enable this parameter, select the **Use custom time scaling** check box.

Time scaling time — Time scaling time points
`2:0.1:3` (default) | scalar | *p*-element vector

Time scaling time points, specified as a scalar or *p*-element vector, where *p* is the number of points for time scaling. By default, the time scaling is a linear time scaling spanning the **Time interval**. Specify the actual time scaling values in **Time scaling values**.

If the **Time** input is specified at a time not specified by these points, interpolation is used to find the right scaling time.

Dependencies

To enable this parameter, select the **Use custom time scaling** check box.

To specify a scalar, the **Time** input must be a scalar.

Data Types: `single` | `double`

Time scaling values — Time scaling vector and first two derivatives
`[0:0.1:1; ones(1,11); zeros(1,11)]` (default) | three-element vector | 3-by-*m* matrix

Time scaling vector and its first two derivatives, specified as a three-element vector or 3-by-*p* matrix, where *p* is the length of **Time scaling time**. By default, the time scaling is a linear time scaling spanning the **Time interval**.

For a nonlinear time scaling, specify the values of the time points in the first row. The second and third rows are the velocity and acceleration of the time points, respectively. For example, to follow the path with a linear velocity to the halfway point, and then jump to the end, the time-scaling would be:

```
s(1,:) = [0 0.25 0.5 1 1 1] % Position
s(2,:) = [1    1   1 0 0 0] % Velocity
s(3,:) = [0    0   0 0 0 0] % Acceleration
```

Dependencies

To enable this parameter, select the **Use custom time scaling** check box.

To specify a three-element vector, the **Time** and **TSTime** inputs must be a scalar.

Data Types: `single` | `double`

Simulate using — Type of simulation to run
`Interpreted execution` (default) | `Code generation`

- `Interpreted execution` — Simulate model using the MATLAB interpreter. This option shortens startup time but has a slower simulation speed than `Code generation`. In this mode, you can debug the source code of the block.
- `Code generation` — Simulate model using generated C code. The first time you run a simulation, Simulink generates C code for the block. The C code is reused for subsequent simulations, as long as the model does not change. This option requires additional startup time, but the speed of the subsequent simulations is comparable to `Interpreted execution`.

Tunable: No

Tips

For better performance, consider these options:

- Minimize the number of waypoint or parameter changes.
- Set the **Waypoint source** parameter to `Internal`.
- Set the **Simulate using** parameter to `Code generation`. For more information, see "Simulation Modes" (Simulink).

Version History

Introduced in R2019a

References

[1] Lynch, Kevin M., and Frank C. Park. *Modern Robotics: Mechanics, Planning, and Control*. Cambridge University Press, 2017.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

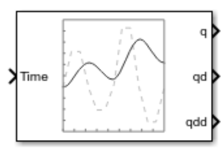
Blocks

Polynomial Trajectory | Rotation Trajectory | Trapezoidal Velocity Profile Trajectory

Functions
`bsplinepolytraj` | `cubicpolytraj` | `quinticpolytraj` | `rottraj` | `transformtraj` | `trapveltraj`

Trapezoidal Velocity Profile Trajectory

Generate trajectories though multiple waypoints using trapezoidal velocity profiles



Libraries:
Robotics System Toolbox / Utilities

Description

The Trapezoidal Velocity Profile Trajectory block generates a trajectory through a given set of waypoints that follow a trapezoidal velocity profile. The block outputs positions, velocities, and accelerations for a trajectory based on the given waypoints and velocity profile parameters.

Ports

Input

Time — Time point along trajectory
scalar | vector

Time point along trajectory, specified as a scalar or vector. In general, when specified as a scalar, this value is synced with simulation time and is used to specify the time point for sampling the trajectory. The block outputs a vector of the trajectory variables at that instant in time. If the time is specified as a vector, the block outputs a matrix with each column corresponding to each element of the vector.

Data Types: `single` | `double`

Waypoints — Waypoint positions along trajectory
n-by-*p* matrix

Positions of waypoints of the trajectory at given time points, specified as an *n*-by-*p* matrix, where *n* is the dimension of the trajectory and *p* is the number of waypoints.

Dependencies

To enable this input, set **Waypoint source** to `External`.

PeakVelocity — Peak velocity of the velocity profile
`[1;2]` (default) | scalar | *n*-element vector | *n*-by-(*p* – 1) matrix

Peak velocity of the profile segment, specified as a scalar, vector, or matrix. This peak velocity is the highest velocity achieved during the trapezoidal velocity profile.

A scalar value is applied to all elements of the trajectory and between all waypoints. An *n*-element vector is applied to each element of the trajectory between all waypoints. An *n*-by-(*p* – 1) matrix is applied to each element of the trajectory for each waypoint.

Dependencies

To enable this parameter, set **Number of parameters** to `1` or `2`. Set **Parameter 1** or **Parameter 2** to `Peak Velocity`. Then, set **Parameter source** to `External`.

Data Types: `single` | `double`

Acceleration — Acceleration of the velocity profile
`[2;2]` (default) | scalar | *n*-element vector | *n*-by-(*p* – 1) matrix

Acceleration of the velocity profile, specified as a scalar, vector, or matrix. This acceleration defines the constant acceleration from zero velocity to the **PeakVelocity** value.

A scalar value is applied to all elements of the trajectory and between all waypoints. An *n*-element vector is applied to each element of the trajectory between all waypoints. An *n*-by-(*p* – 1) matrix is applied to each element of the trajectory for each waypoint.

Dependencies

To enable this parameter, set **Number of parameters** to `1` or `2`. Set **Parameter 1** or **Parameter 2** to `Acceleration`. Then, set **Parameter source** to `External`.

Data Types: `single` | `double`

EndTime — Duration of trajectory segment
`[1;2]` (default) | scalar | *n*-element vector | *n*-by-(*p* – 1) matrix

Duration of trajectory segment, specified as a scalar, vector, or matrix.

A scalar value is applied to all elements of the trajectory and between all waypoints. An *n*-element vector is applied to each element of the trajectory between all waypoints. An *n*-by-(*p* – 1) matrix is applied to each element of the trajectory for each waypoint.

Dependencies

To enable this parameter, set **Number of parameters** to `1` or `2`. set **Parameter 1** or **Parameter 2** to `End Time`. Then, set **Parameter source** to `External`.

Data Types: `single` | `double`

Acceleration Time — Duration of acceleration phase of velocity profile
`[1;1]` (default) | scalar | *n*-element vector | *n*-by-(*p* – 1) matrix

Duration of acceleration phase of velocity profile, specified as a scalar, vector, or matrix.

A scalar value is applied to all elements of the trajectory and between all waypoints. An *n*-element vector is applied to each element of the trajectory between all waypoints. An *n*-by-(*p* – 1) matrix is applied to each element of the trajectory for each waypoint.

Dependencies

To enable this parameter, set **Number of parameters** to `1` or `2`. set **Parameter 1** or **Parameter 2** to `Acceleration Time`. Then, set **Parameter source** to `External`.

Data Types: `single` | `double`

Output

q — Position of trajectory
scalar | vector | matrix

Position of the trajectory, specified as a scalar, vector, or matrix. If you specify a scalar for the `Time` input with an *n*-dimensional trajectory, the output is a vector with *n* elements. If you specify a vector of *m* elements for the `Time` input, the output is an *n*-by-*m* matrix.

Data Types: `single` | `double`

qd — Velocity of trajectory
scalar | vector | matrix

Velocity of the trajectory, specified as a scalar, vector, or matrix. If you specify a scalar for the `Time` input with an *n*-dimensional trajectory, the output is a vector with *n* elements. If you specify a vector of *m* elements for the `Time` input, the output is an *n*-by-*m* matrix.

Data Types: `single` | `double`

qdd — Acceleration of trajectory
scalar | vector | matrix

Acceleration of the trajectory, specified as a scalar, vector, or matrix. If you specify a scalar for the `Time` input with an *n*-dimensional trajectory, the output is a vector with *n* elements. If you specify a vector of *m* elements for the `Time` input, the output is an *n*-by-*m* matrix.

Data Types: `single` | `double`

Parameters

Waypoint source — Source for waypoints
`Internal` (default) | `External`

Specify `External` to specify the **Waypoints** and **Time points** parameters as block inputs instead of block parameters.

Waypoints — Waypoint positions along trajectory
n-by-*p* matrix

Positions of waypoints of the trajectory at given time points, specified as an *n*-by-*p* matrix, where *n* is the dimension of the trajectory and *p* is the number of waypoints.

Number of parameters — Number of velocity profile parameters
`0` (default) | `1` | `2`

Number of velocity profile parameters, specified as `0`, `1`, or `2`. Increasing this value adds **Parameter 1** and **Parameter 2** for specifying parameters for the velocity profile.

Parameter 1 — Velocity profile parameter
`Peak Velocity` | `Acceleration` | `End Time` | `Acceleration Time`

Velocity profile parameter, specified as `Peak Velocity`, `Acceleration`, `End Time`, or `Acceleration Time`. Setting this parameter creates a parameter in the mask with this value as its name.

Dependencies

To enable this parameter, set **Number of parameters** to `1` or `2`.

If **Parameter Source** is set to `Internal`, this parameter creates a parameter in the mask with this value as its name.

If **Parameter Source** is set to `External`, this parameter creates an input port based on this value.

Parameter 2 — Velocity profile parameter
`Peak Velocity` | `Acceleration` | `End Time` | `Acceleration Time`

Velocity profile parameter, specified as `Peak Velocity`, `Acceleration`, `End Time`, or `Acceleration Time`. Setting this parameter creates a parameter in the mask with this value as its name.

Dependencies

To enable this parameter, set **Number of parameters** to `2`.

If **Parameter Source** is set to `Internal`, this parameter creates a parameter in the mask with this value as its name.

If **Parameter Source** is set to `External`, this parameter creates an input port based on this value.

Parameter source — Source for waypoints
`Internal` (default) | `External`

Specify `External` to specify the velocity profile parameters as block inputs instead of block parameters.

Dependencies

To enable this parameter, set **Number of parameters** to `1` or `2`.

PeakVelocity — Peak velocity of the velocity profile
`[1;2]` (default) | scalar | *n*-element vector | *n*-by-(*p* – 1) matrix

Peak velocity of the profile segment, specified as a scalar, vector, or matrix. This peak velocity is the highest velocity achieved during the trapezoidal velocity profile.

A scalar value is applied to all elements of the trajectory and between all waypoints. An *n*-element vector is applied to each element of the trajectory between all waypoints. An *n*-by-(*p* – 1) matrix is applied to each element of the trajectory for each waypoint.

Dependencies

To enable this parameter, set **Number of parameters** to `1` or `2`. Then, set **Parameter 1** or **Parameter 2** to `Peak Velocity`.

Data Types: `single` | `double`

Acceleration — Acceleration of the velocity profile
`[2;2]` (default) | scalar | *n*-element vector | *n*-by-(*p* – 1) matrix

Acceleration of the velocity profile, specified as a scalar, vector, or matrix. This acceleration defines the constant acceleration from zero velocity to the **PeakVelocity** value.

A scalar value is applied to all elements of the trajectory and between all waypoints. An *n*-element vector is applied to each element of the trajectory between all waypoints. An *n*-by-(*p* – 1) matrix is applied to each element of the trajectory for each waypoint.

Dependencies

To enable this parameter, set **Number of parameters** to `1` or `2`. Then, set **Parameter 1** or **Parameter 2** to `Acceleration`.

Data Types: `single` | `double`

EndTime — Duration of trajectory segment
`[1;2]` (default) | scalar | *n*-element vector | *n*-by-(*p* – 1) matrix

Duration of trajectory segment, specified as a scalar, vector, or matrix.

A scalar value is applied to all elements of the trajectory and between all waypoints. An *n*-element vector is applied to each element of the trajectory between all waypoints. An *n*-by-(*p* – 1) matrix is applied to each element of the trajectory for each waypoint.

Dependencies

To enable this parameter, set **Number of parameters** to `1` or `2`. Then, set **Parameter 1** or **Parameter 2** to `End Time`.

Data Types: `single` | `double`

Acceleration Time — Duration of acceleration phase of velocity profile
`[1;1]` (default) | scalar | *n*-element vector | *n*-by-(*p* – 1) matrix

Duration of acceleration phase of velocity profile, specified as a scalar, vector, or matrix.

A scalar value is applied to all elements of the trajectory and between all waypoints. An *n*-element vector is applied to each element of the trajectory between all waypoints. An *n*-by-(*p* – 1) matrix is applied to each element of the trajectory for each waypoint.

Dependencies

To enable this parameter, set **Number of parameters** to `1` or `2`. Then, set **Parameter 1** or **Parameter 2** to `Acceleration Time`.

Data Types: `single` | `double`

Simulate using — Type of simulation to run
`Interpreted execution` (default) | `Code generation`

- `Interpreted execution` — Simulate model using the MATLAB interpreter. This option shortens startup time but has a slower simulation speed than `Code generation`. In this mode, you can debug the source code of the block.
- `Code generation` — Simulate model using generated C code. The first time you run a simulation, Simulink generates C code for the block. The C code is reused for subsequent simulations, as long as the model does not change. This option requires additional startup time, but the speed of the subsequent simulations is comparable to `Interpreted execution`.

Tunable: No

Tips

For better performance, consider these options:

- Minimize the number of waypoint or parameter changes.
- Set the **Waypoint source** parameter to `Internal`.
- Set the **Simulate using** parameter to `Code generation`. For more information, see "Simulation Modes" (Simulink).

Version History

Introduced in R2019a

References

[1] Lynch, Kevin M., and Frank C. Park. *Modern Robotics: Mechanics, Planning and Control*. Cambridge: Cambridge University Press, 2017.

[2] Spong, Mark W., Seth Hutchinson, and M. Vidyasagar. *Robot Modeling and Control*. John Wiley & Sons, 2006.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

See Also

Blocks
Polynomial Trajectory | Rotation Trajectory | Transform Trajectory

Functions
`bsplinepolytraj` | `cubicpolytraj` | `quinticpolytraj` | `rottraj` | `transformtraj` | `trapveltraj`

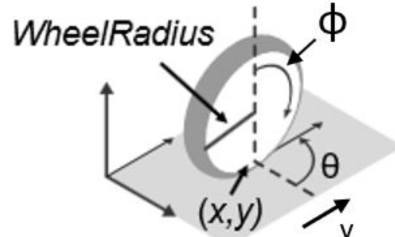
Unicycle Kinematic Model

Compute vehicle motion using unicycle kinematic model

Libraries:
Robotics System Toolbox / Mobile Robot Algorithms

Description

The Unicycle Kinematic Model block creates a unicycle vehicle model to simulate simplified car-like vehicle dynamics. This model approximates a vehicle as a unicycle with a given wheel radius, `Wheel radius`, that can spin in place according to a steering angular velocity, `ω`.



Ports

Input

dϕ/dt — Angular velocity of wheel
numeric scalar

Angular velocity of the wheel in radians per second.

Dependencies

To enable this port, set the `Vehicle inputs` parameter to `Wheel Speed & Heading Angular Velocity`.

v — Vehicle speed
numeric scalar

Vehicle speed, specified in meters per second.

Dependencies

To enable this port, set the `Vehicle inputs` parameter to `Vehicle Speed & Heading Angular Velocity`.

ω — Steering angular velocity
numeric scalar

Angular velocity of the vehicle, specified in radians per second. A positive value steers the vehicle left and negative values steer the vehicle right.

Output

state — Pose of vehicle
three-element vector

Current *xy*-position and orientation of the vehicle, specified as a *[x y theta]* vector in meters and radians.

stateDot — Derivatives of `state` output
three-element vector

The linear and angular velocities of the vehicle, specified as a *[xDot yDot thetaDot]* vector in meters per second and radians per second. The linear and angular velocities are calculated by taking the derivative of the `state` output.

Parameters

Vehicle inputs — Type of speed and directional inputs for vehicle
`Vehicle Speed & Heading Angular Velocity` (default) | `Wheel Speed & Heading Angular Velocity`

Type of speed and directional inputs to control the vehicle. Options are:

- `Vehicle Speed & Heading Angular Velocity` — Vehicle speed in meters per second with a heading angular velocity in radians per second..
- `Wheel Speed & Heading Angular Velocity` — Wheel speed in radians per second with a heading angular velocity in radians per second.

Wheel radius — Wheel radius of vehicle
`0.1` (default) | positive numeric scalar

The wheel radius of the vehicle, specified in meters.

Wheel speed range — Minimum and Maximum vehicle speeds
`[-Inf Inf]` (default) | two-element vector

The minimum and maximum wheel speeds, specified in radians per second.

Initial state — Distance between front and rear axles
`[0;0;0]` (default) | three-element vector

The initial *x*-, *y*-position and orientation, *theta*, of the vehicle.

Simulate using — Type of simulation to run
`Interpreted execution` (default) | `Code generation`

- `Interpreted execution` — Simulate model using the MATLAB interpreter. For more information, see “Simulation Modes” (Simulink).
- `Code generation` — Simulate model using generated C code. The first time you run a simulation, Simulink generates C code for the block. The C code is reused for subsequent simulations, as long as the model does not change.

Tunable: No

Version History

Introduced in R2019b

References

[1] Lynch, Kevin M., and Frank C. Park. *Modern Robotics: Mechanics, Planning, and Control* 1st ed. Cambridge, MA: Cambridge University Press, 2017.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

See Also

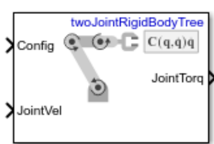
Blocks
Differential Drive Kinematic Model | Ackermann Kinematic Model | Bicycle Kinematic Model

Classes
`unicycleKinematics`

Topics
"Simulate Different Kinematic Models for Mobile Robots"
"Mobile Robot Kinematics Equations"

Velocity Product Torque

Joint torques that cancel velocity-induced forces



Libraries:
Robotics System Toolbox / Manipulator Algorithms

Description

The Velocity Product Torque block returns the torques that cancel the velocity-induced forces for the given robot configuration (joint positions) and joint velocities for the **Rigid body tree** robot model.

Ports

Input

Config — Robot configuration
vector

Robot configuration, specified as a vector of positions for all nonfixed joints in the robot model, as set by the **Rigid body tree** parameter. You can also generate this vector for a complex robot using the `homeConfiguration` or `randomConfiguration` functions inside a Constant or MATLAB Function block.

JointVel — Joint velocities
vector

Joint velocities, specified as a vector. The number of joint velocities is equal to the degrees of freedom (number of nonfixed joints) of the robot.

Output

JointTorq — Joint torques
vector

Joint torques, specified as a vector. Each element corresponds to a torque applied to a specific joint. The number of joint torques is equal to the degrees of freedom (number of nonfixed joints) of the robot.

Parameters

Rigid body tree — Robot model
`twoJointRigidBodyTree` (default) | `RigidBodyTree` object

Robot model, specified as a `rigidBodyTree` object. You can also import a robot model from an URDF (Unified Robot Description Formation) file using `importrobot`.

The default robot model, `twoJointRigidBodyTree`, is a robot with revolute joints and two degrees of freedom.

Simulate using — Type of simulation to run
`Interpreted execution` (default) | `Code generation`

- `Interpreted execution` — Simulate model using the MATLAB interpreter. This option shortens startup time but has a slower simulation speed than `Code generation`. In this mode, you can debug the source code of the block.
- `Code generation` — Simulate model using generated C code. The first time you run a simulation, Simulink generates C code for the block. The C code is reused for subsequent simulations, as long as the model does not change. This option requires additional startup time, but the speed of the subsequent simulations is comparable to `Interpreted execution`.

Tunable: No

Version History

Introduced in R2018a

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

Blocks

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

Classes

`rigidBodyTree`

Functions

`velocityProduct` | `importrobot` | `homeConfiguration` | `randomConfiguration`

5

Apps

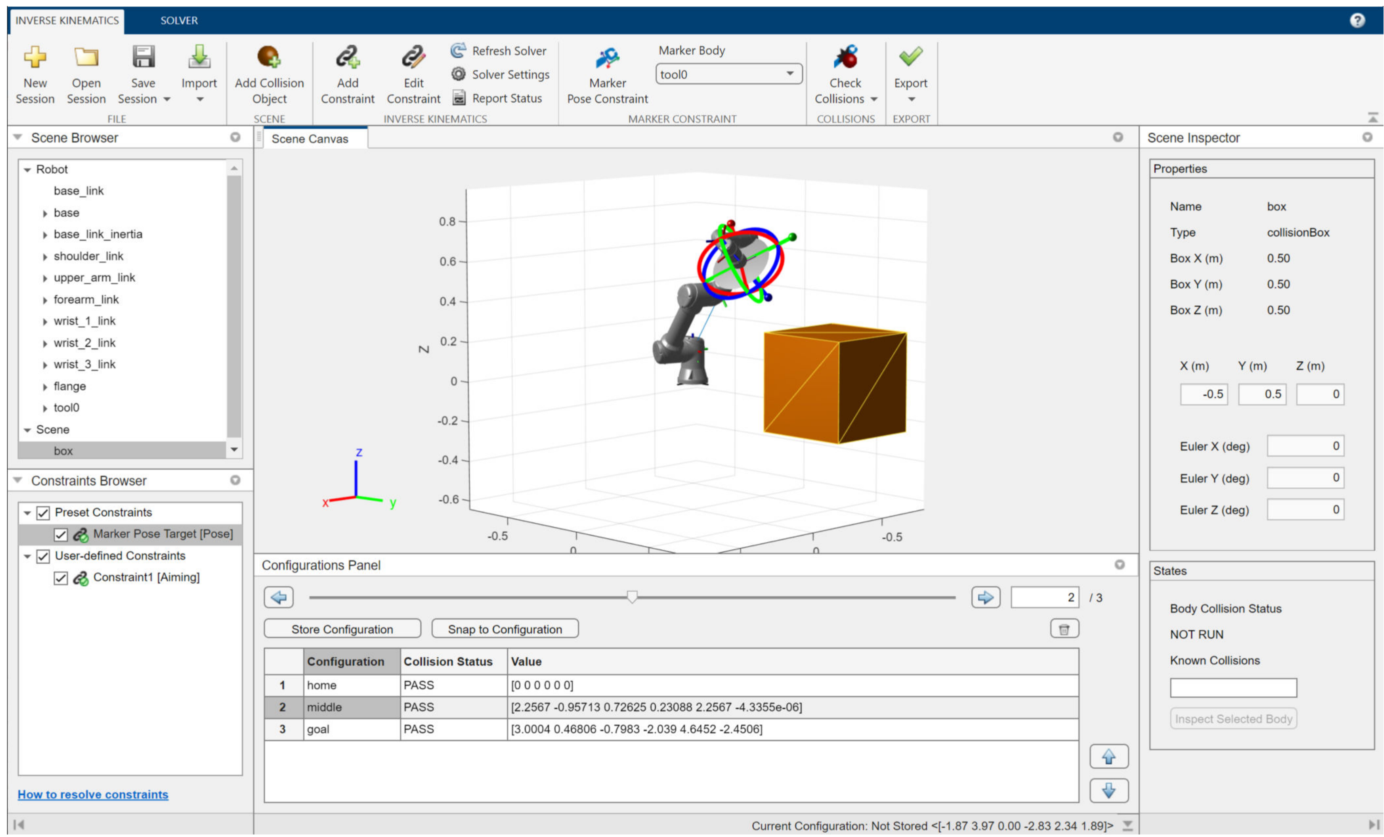
Inverse Kinematics Designer

Design inverse kinematics solvers, configurations, and waypoints

Description

The **Inverse Kinematics Designer** enables you to design an inverse kinematics solver for a URDF robot model. You can adjust the inverse kinematics solver and add constraints to achieve the desired behavior. Using this app you can:

- Import URDF robot models from URDF files or the MATLAB Workspace.
- Adjust inverse kinematics solvers and constraints.
- Create joint configurations and export waypoints.
- Export solver settings, constraints, and joint configurations to the MATLAB workspace.



Open the Inverse Kinematics Designer App

- MATLAB Toolstrip: On the **Apps** tab, under **Robotics And Autonomous Systems**, click **Inverse Kinematics Designer** .

- MATLAB command prompt: Enter `inverseKinematicsDesigner`.

Examples

Create Inverse Kinematics Designer Session

This example shows how to create, load, and save an Inverse Kinematics Designer session in addition to loading a robot into the session. The completed file is attached for reference as `iksessiondata.mat`. Load the session on page 5-8 with the `inverseKinematicsDesigner` function or follow along with this example to create it.

Create Session

Open **Inverse Kinematics Designer** by using the `inverseKinematicsDesigner` function.

```
inverseKinematicsDesigner
```

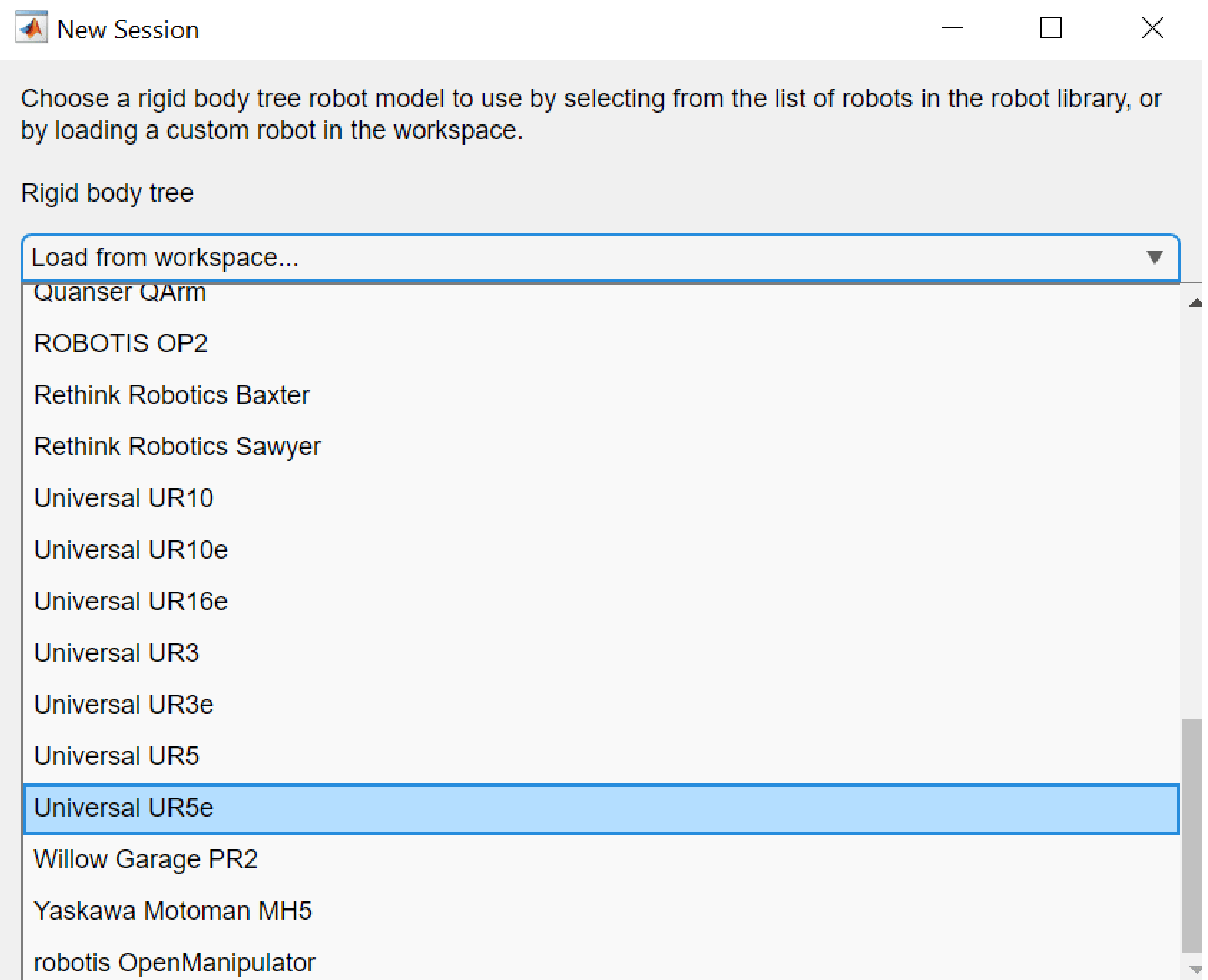
Load Robot into Session

Use `loadrobot` from the Command Window to load a `rigidBodyTree` such as a Universal UR5e into the Workspace. `importrobot` can also be used to import a `rigidBodyTree` object from any robot URDF file.

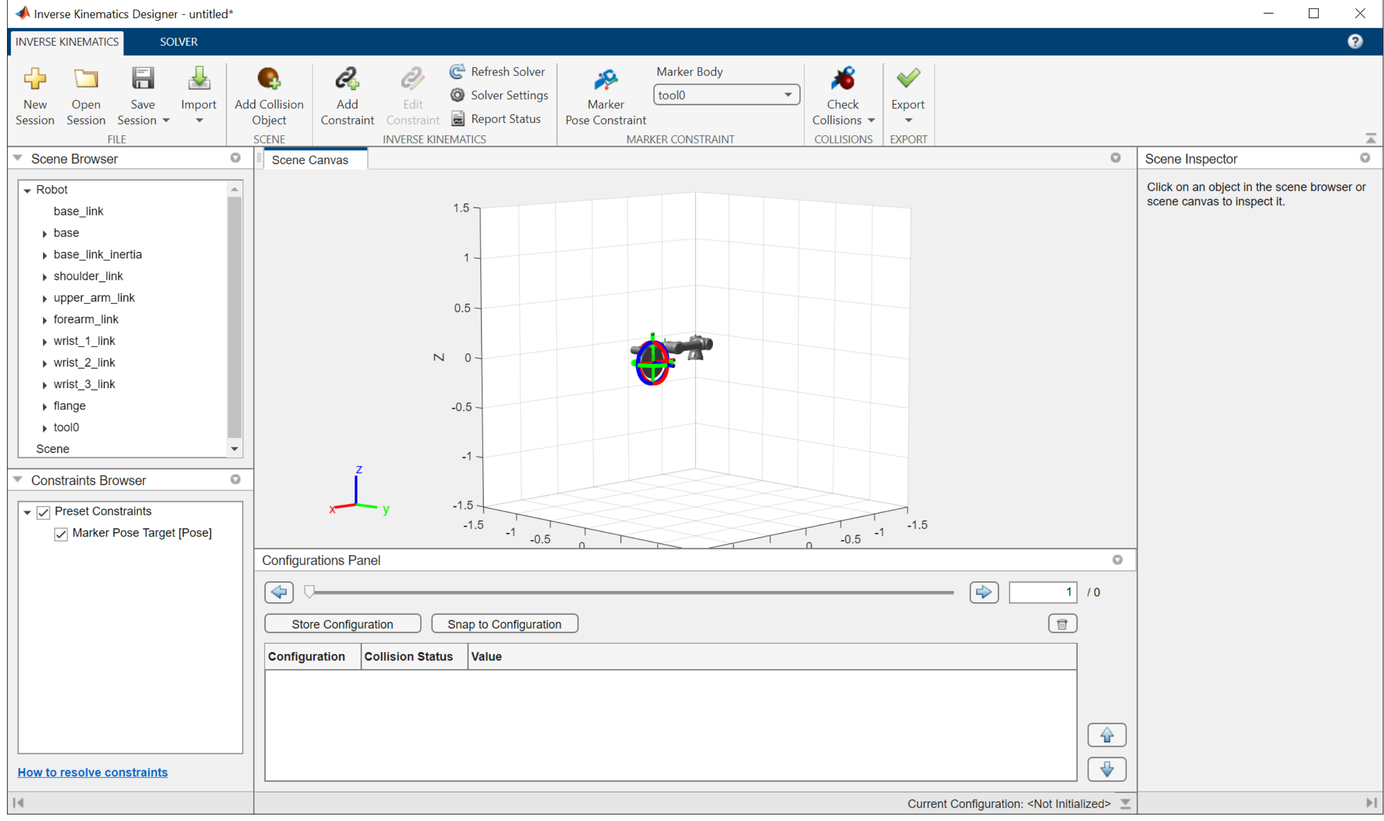
```
uniUR5e = loadrobot("universalUR5e");
```

Click **New Session** and select `uniUR5e` from the table in the dialog and click **OK**. This table contains all of the `rigidBodyTree` objects in the workspace. If you do not see your object in the table, verify that it is in your workspace and click **Refresh**.

Alternatively, you can load a robot by selecting from a list of robot models that come with the Robotics System Toolbox™ using the **Rigid body tree** drop down dialog and clicking **OK**.



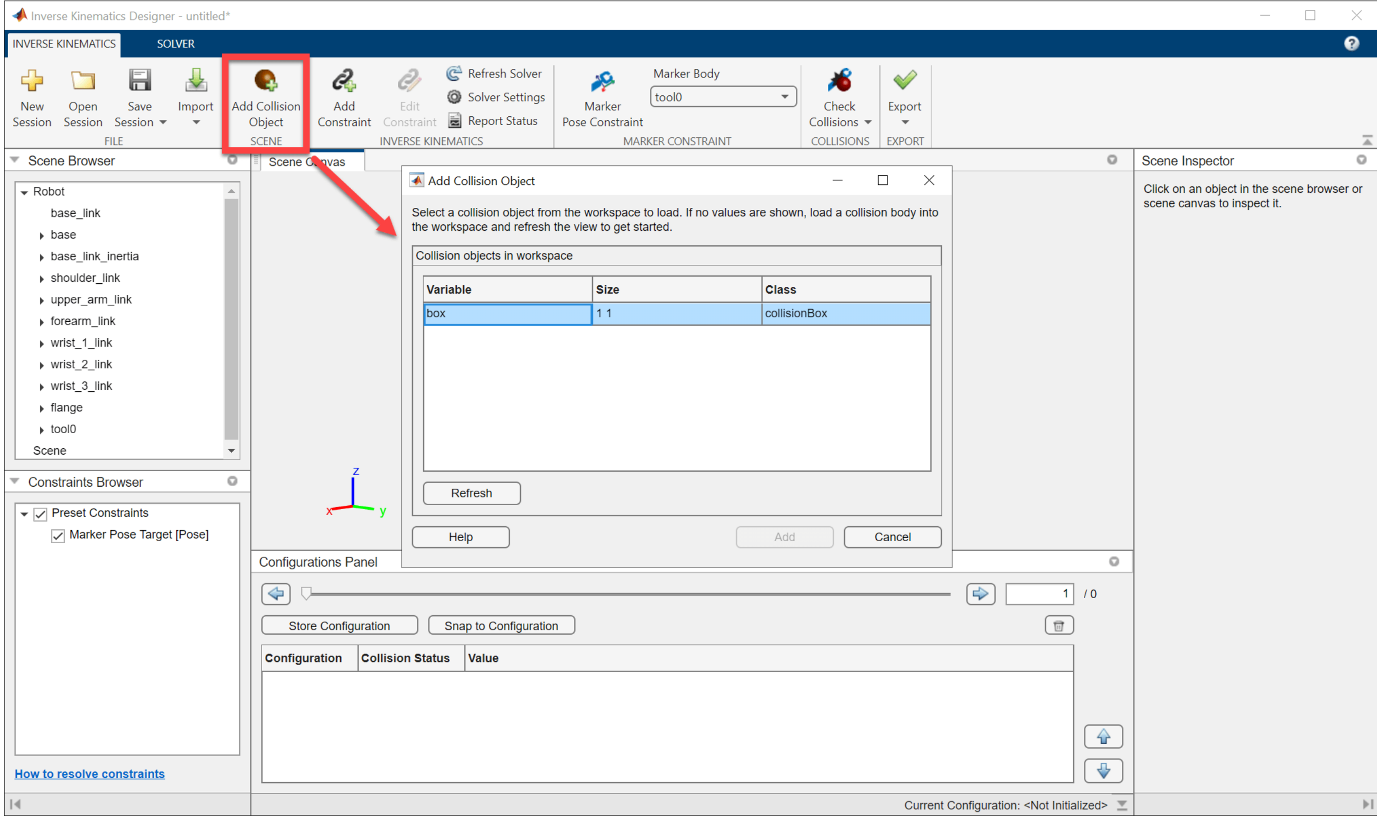
The **Scene Canvas** now contains the robot model, and the **Scene Browser** now displays all of the rigid bodies of the robot.



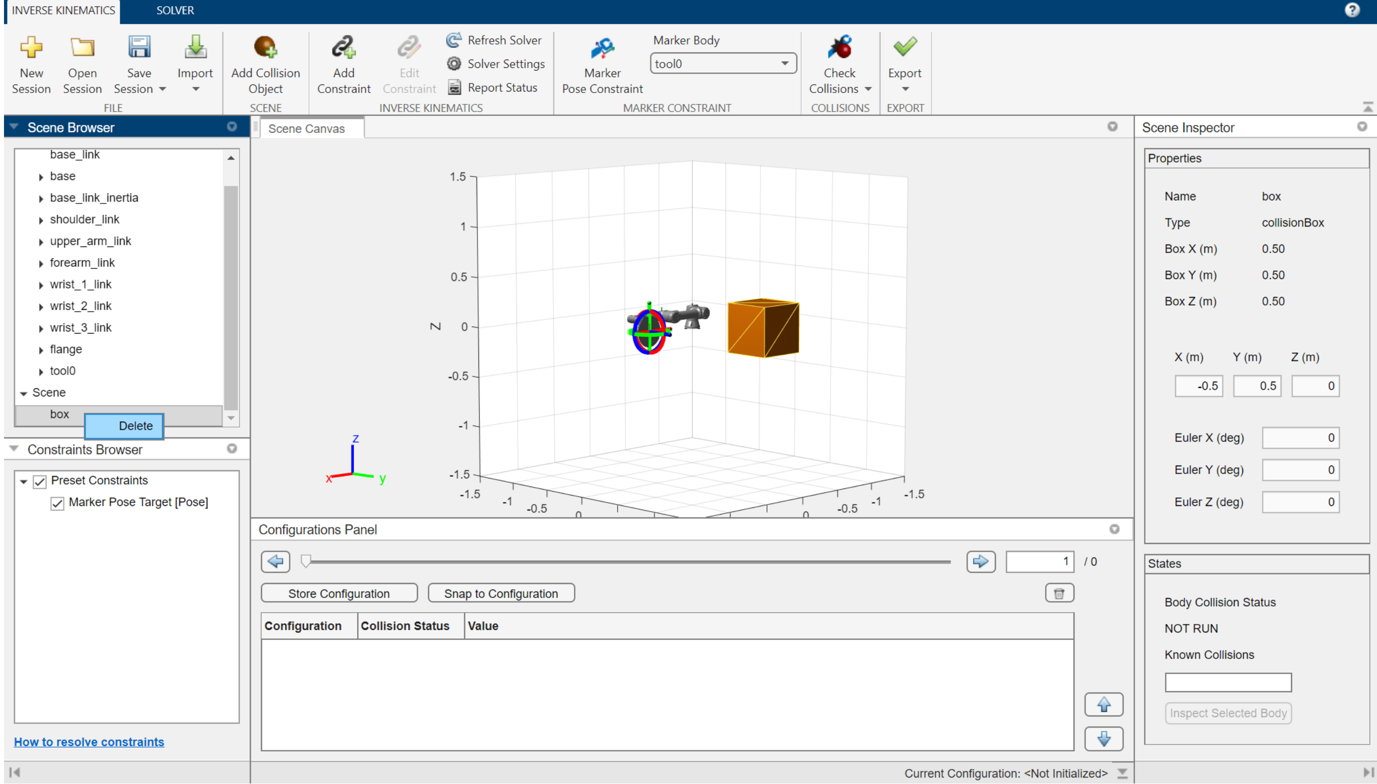
Add Collision Objects

To add a collision object into the **Scene Canvas**, a collision object must be in the Workspace. For convenience, this example provides a simple box to use. For more information about creating collision objects, see `collisionMesh`, `collisionBox`, `collisionSphere`, and `collisionCylinder`.

Load the `collisionobject` MAT file which will save a `collisionBox` named `box` to your Workspace. Click **Add Collision Object**, and select `box`, from the table. Click **OK** to add it to the **Scene Canvas**.



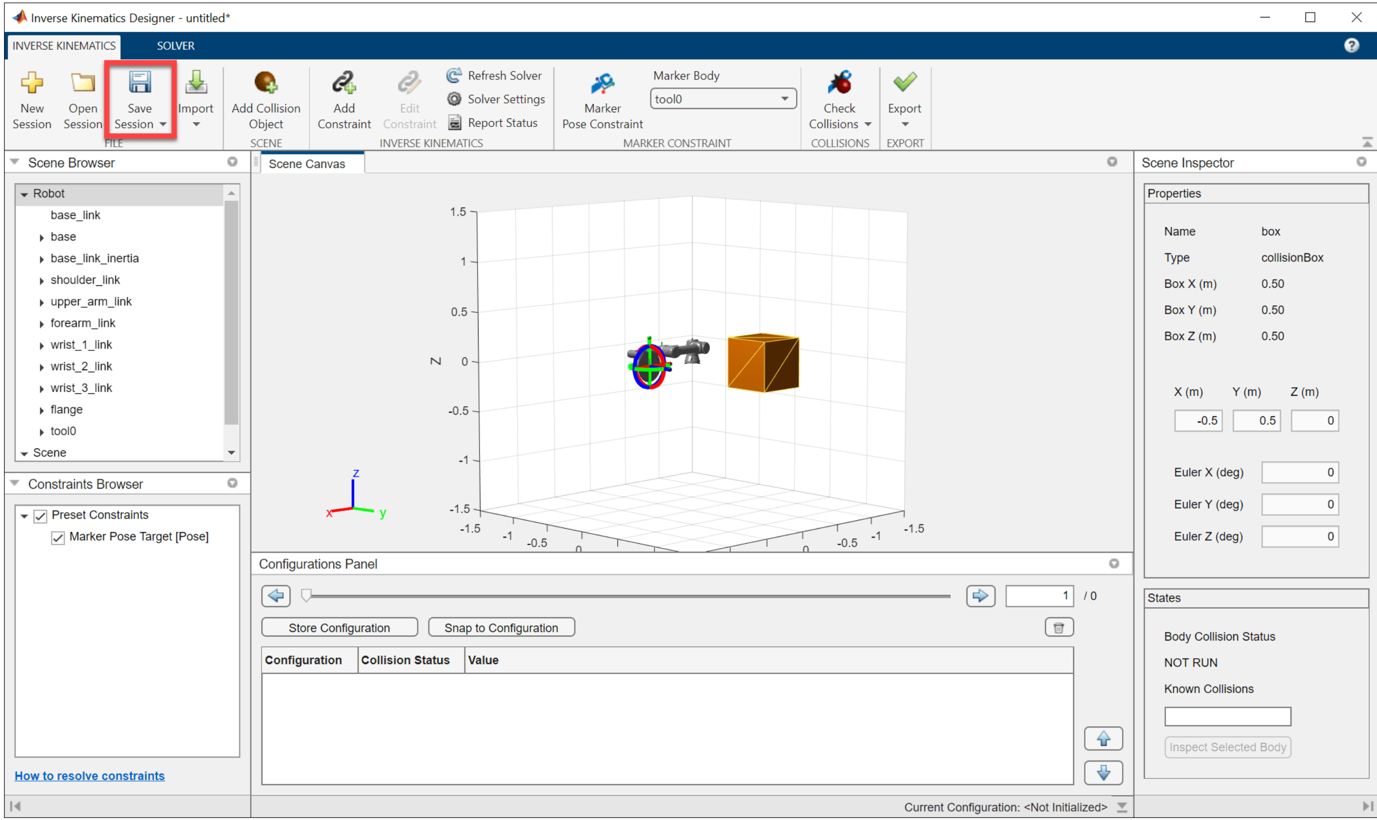
The **Scene Canvas** now contains both the robot and the collision object. We will keep the object in this example, but if you want to delete the collision object, find the collision object in the **Scene Browser** under **Scene**, right-click the name of the collision object and click **Delete**.



The position and Euler orientation of the object can viewed using the **Scene Inspector** when the object is selected in either the **Scene Canvas** or **Scene Browser**. The properties listed will change depending on the type of the collision object selected.

Save Session

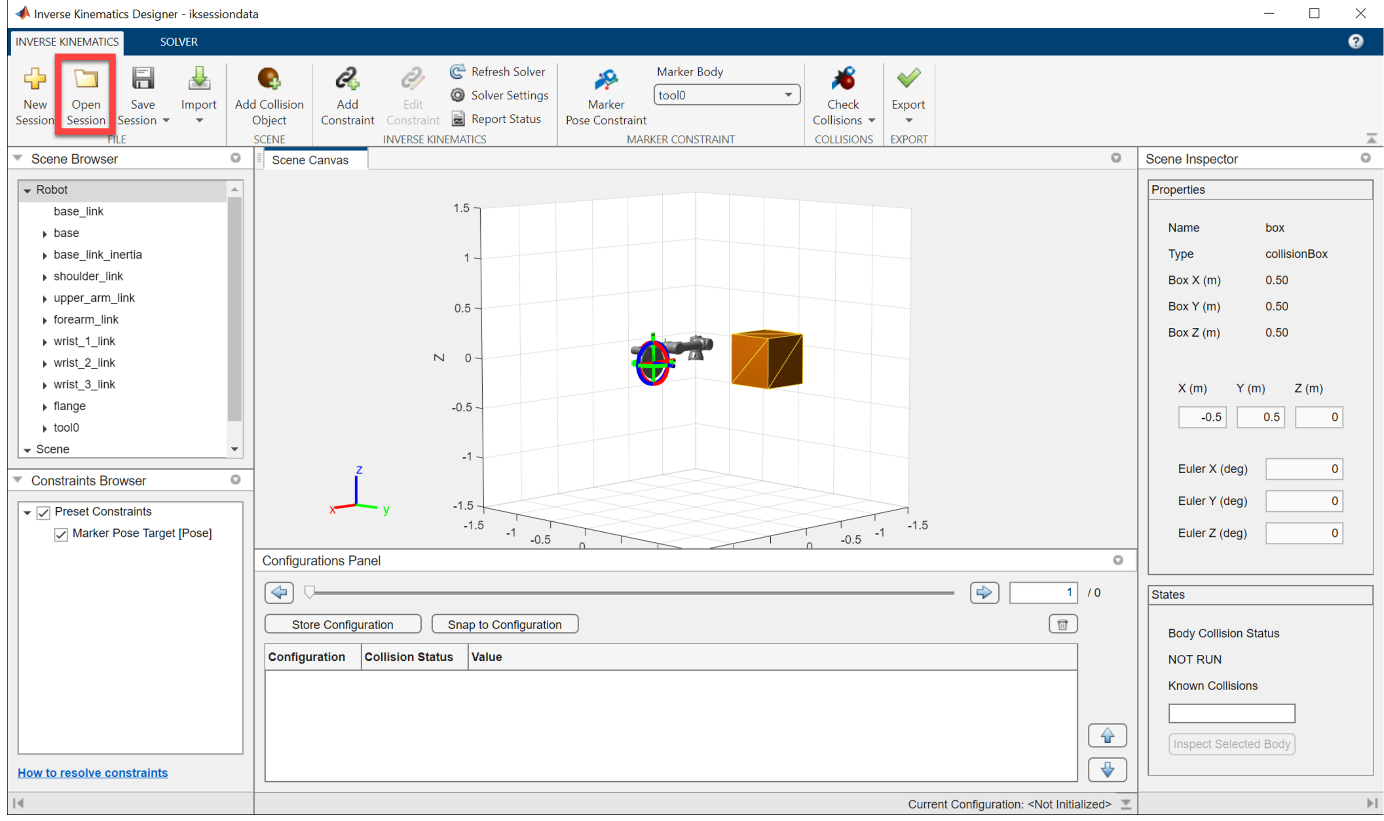
To save this session, click **Save Session**. If this is the first time saving the session, name the file and select a location to save it. The file will be saved as a MAT (*.mat) file containing all session data and settings.



Load Saved Session

To load a session file, click **Open Session** from the **Inverse Kinematics Designer** app or specify the MAT file as a string to the `inverseKinematicsDesigner`. An example of this session has been provided in this example as `iksessiondata.mat`.

```
inverseKinematicsDesigner("iksessiondata.mat")
```



Use Scene Canvas and Move Robot

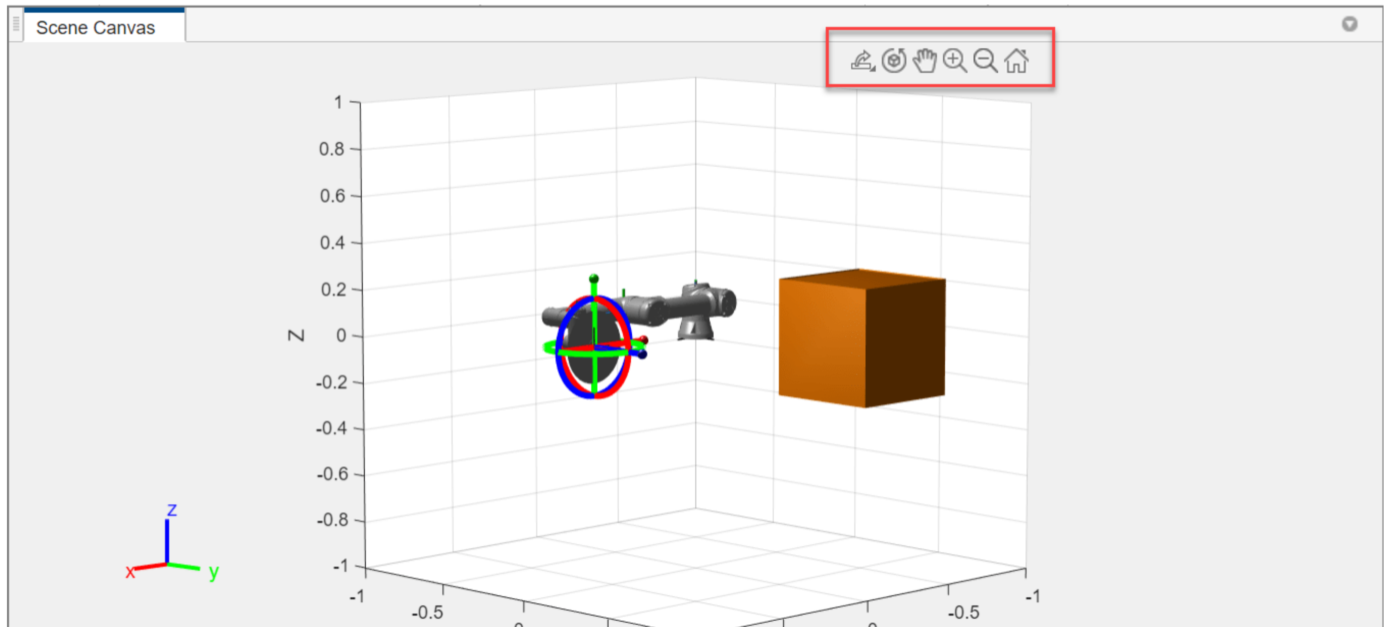
This example shows how to use the **Scene Canvas** and move a robot in it using the **Inverse Kinematics Designer** app.

Load an existing session (`iksessiondata.mat`) or refer to the Create Inverse Kinematics Designer Session example to create a session.

```
inverseKinematicsDesigner
```

Scene Canvas Controls

Use the axes toolbar in the **Scene Canvas** to control the view.



To rotate the **Scene Canvas**, select the Rotate 3D button and click and drag within the scene.

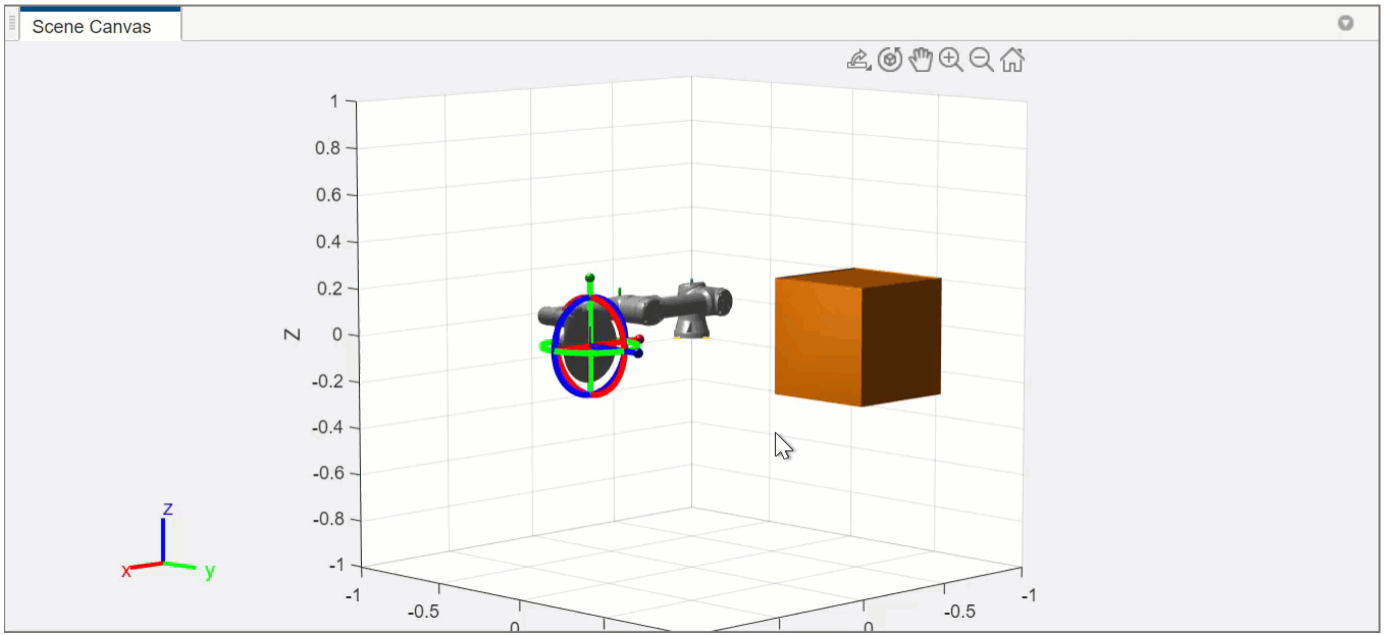
Click the Pan button and click and drag within the scene to pan within the **Scene Canvas**.

Select the Zoom In, or Zoom Out buttons and click and drag up or down to zoom in or out of an area within the **Scene Canvas** respectively.

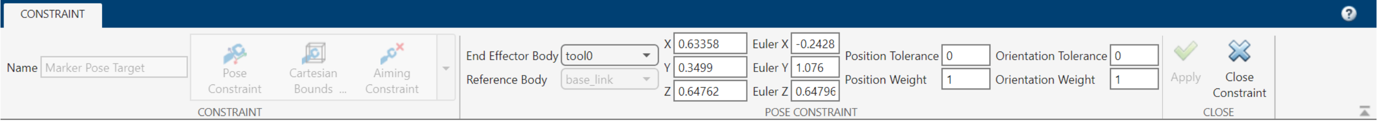
Click the Restore View button to revert to the original default view.

Move Robot

Move the robot using constraints such as the preset Marker Pose Target constraint. The Marker Pose Target constraint is the simplest constraint to use to move the robot. This constraint sets a target pose on the last body in the robot model. In this case the marker body is set to `tool0`. The marker is visualized on top of the selected marker body with the red, green, and blue linear and circular indicators in the **Scene Canvas**. Clicking and dragging the linear or circular indicators will change the target position and Euler orientation respectively. The colors correspond to the axes colors indicated in the bottom left of the **Scene Canvas**.



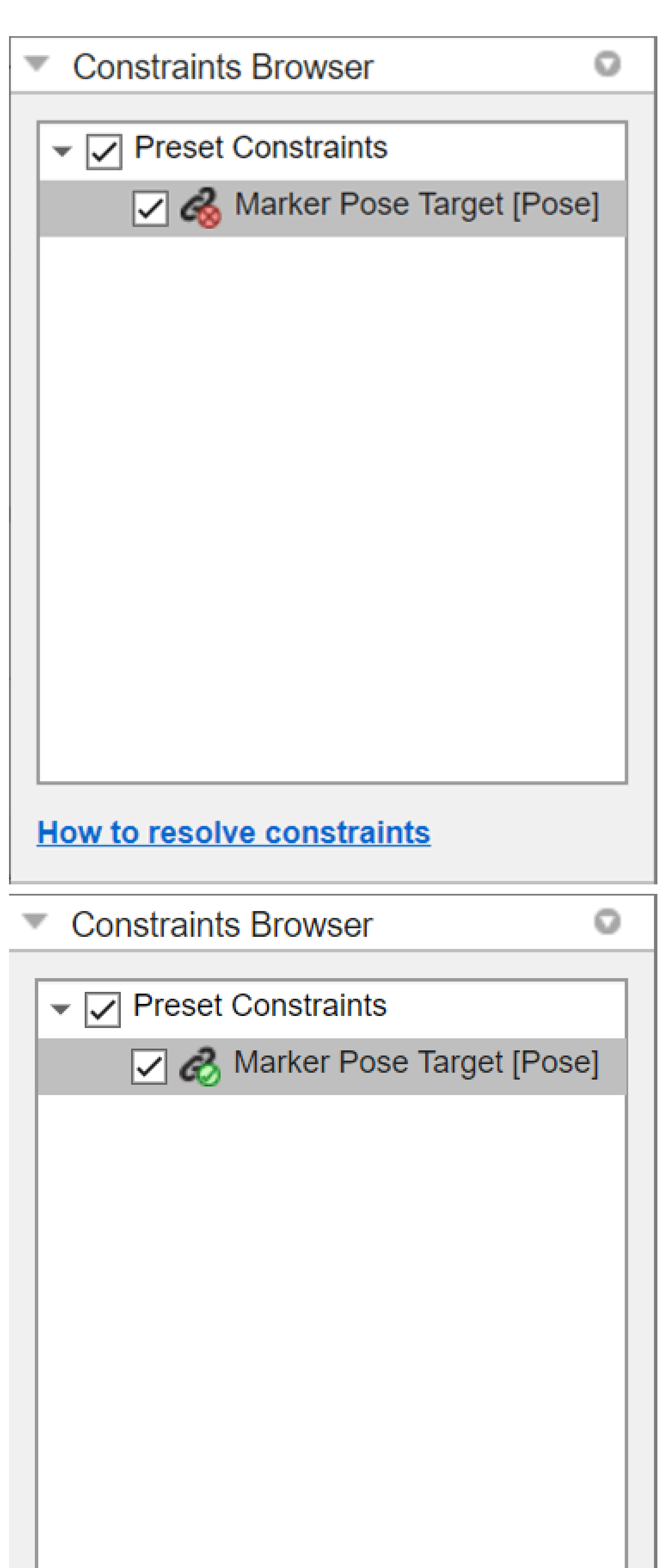
Click **Marker Pose Constraint** to open the **Constraint** tab. From the **Constraint** tab, set Cartesian position in meters, Euler orientation in degrees, and the weights and tolerances of the position and orientation. The marker body can be changed in either the **Constraint** tab under **End Effector Body** list or in the **Marker Body** list in the **Inverse Kinematics** tab. Click **Apply** to save any changes, and click **Close Constraint** to exit the **Constraint** tab. Note that the specified Euler angles are computed using *XYZ* sequence.



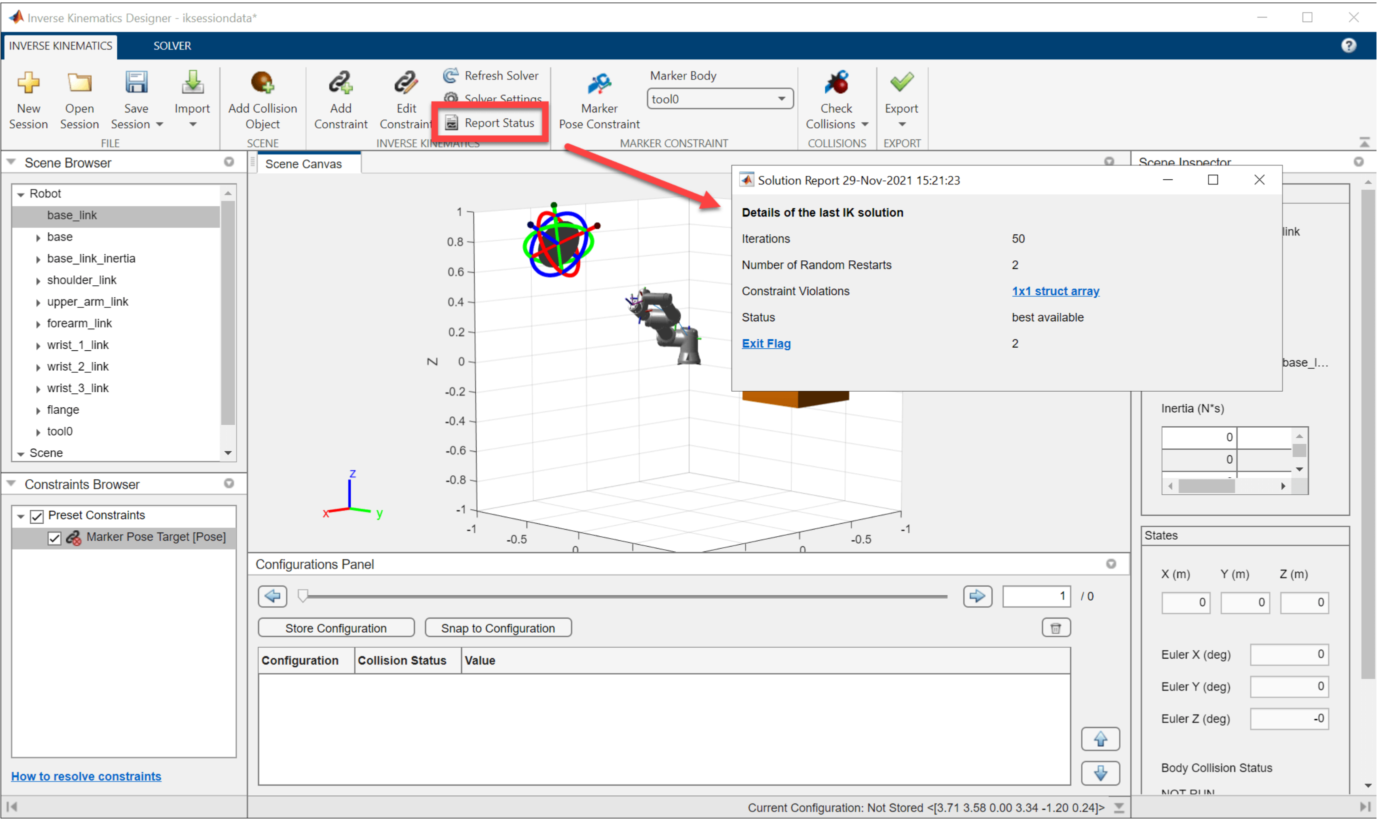
The Marker Pose Target constraint can also be toggled on or off by using clearing or selecting the check box next to the Marker Pose Target constraint in the **Constraints Browser**.

Solution Details

When the Marker Pose Target is moved, it sets the target pose and the inverse kinematics solver solves for a configuration where the selected marker body reaches the target pose. If it cannot find a solution to the target pose, the robot will move to the best available solution and the marker body will not move to the Marker Pose Target. This kind of solution can be identified visually by the constraint icon in the **Constraints Browser**. The icon with red x shows that the constraint is not met, while the green check shows that the constraint is being met.


Constraints Browser
Preset Constraints
Marker Pose Target [Pose]
How to resolve constraints
Constraints Browser
Preset Constraints
Marker Pose Target [Pose]

To find more information about why the solver failed to reach the solution, click the **Report Status** to see the details of the solver's solution. The number of **Iterations** and **Number of Random Restarts** list the number of times that the solver executed each respectively. The **Constraints Violations** shows structure array of all the conflicts that can be displayed in the command window. The status will show `success` if the solver successfully solves for the target pose, or `best available` if the solver is showing the best available solution it found if it could not reach the target pose. Exit Flag provides more detail on the execution of the specific solver algorithm. See "Inverse Kinematics Algorithms" for more information about the different exit flag types.



To troubleshoot why a solver failed to reach a solution, see "Resolving Constraint Conflict" on page 5-37 for some tips.

Create Collision-Free Configurations and Export Waypoints

This example shows how to use **Inverse Kinematics Designer** to create joint configurations and check for collisions using the **Scene Inspector**. This example uses data and skills from "Create Inverse Kinematics Designer Session" on page 5-3, and "Use Scene Canvas and Move Robot" on page 5-9. Refer to those examples before continuing.

Load Session

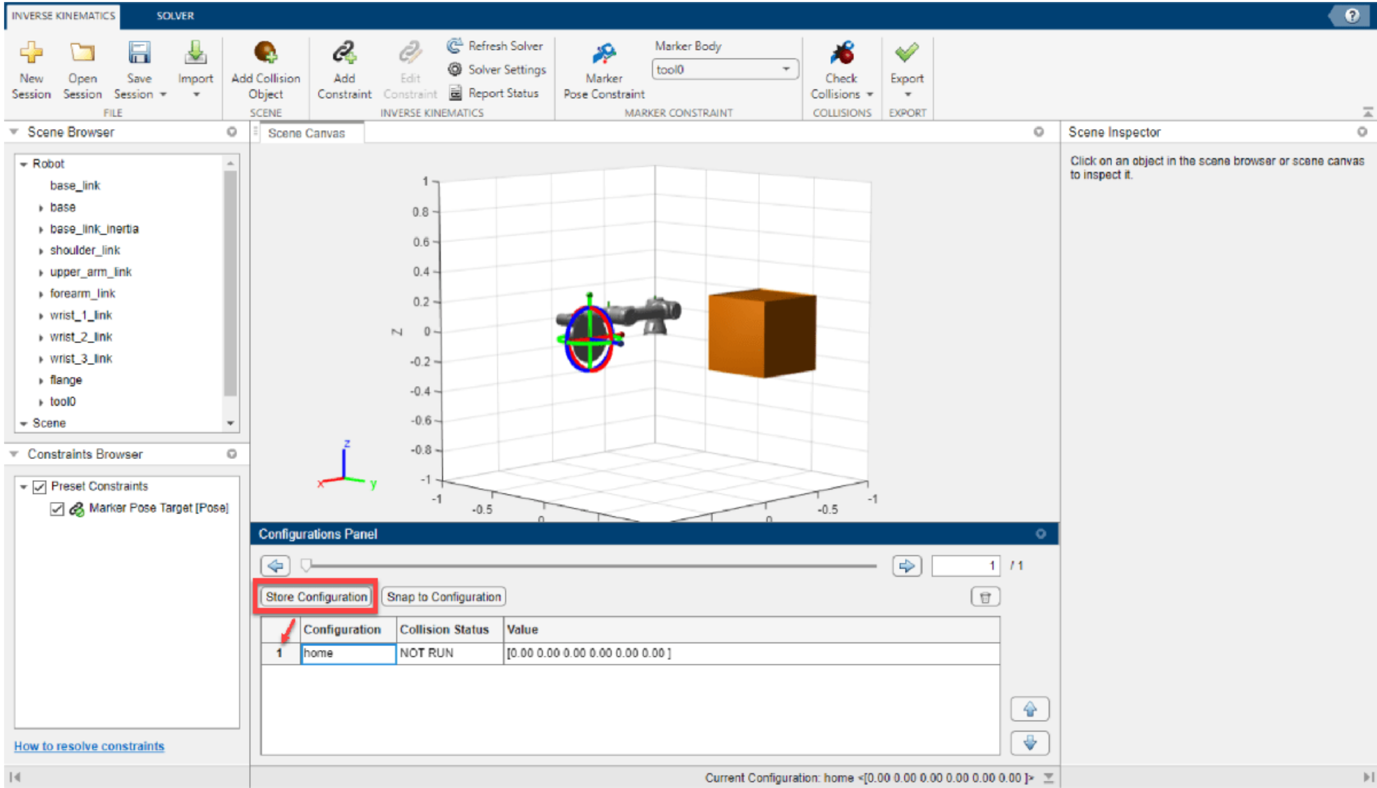
Use `inverseKinematicsDesigner` with the `iksessiondata.mat` session file to load a robot with a basic collision object in the scene.

```
inverseKinematicsDesigner
```

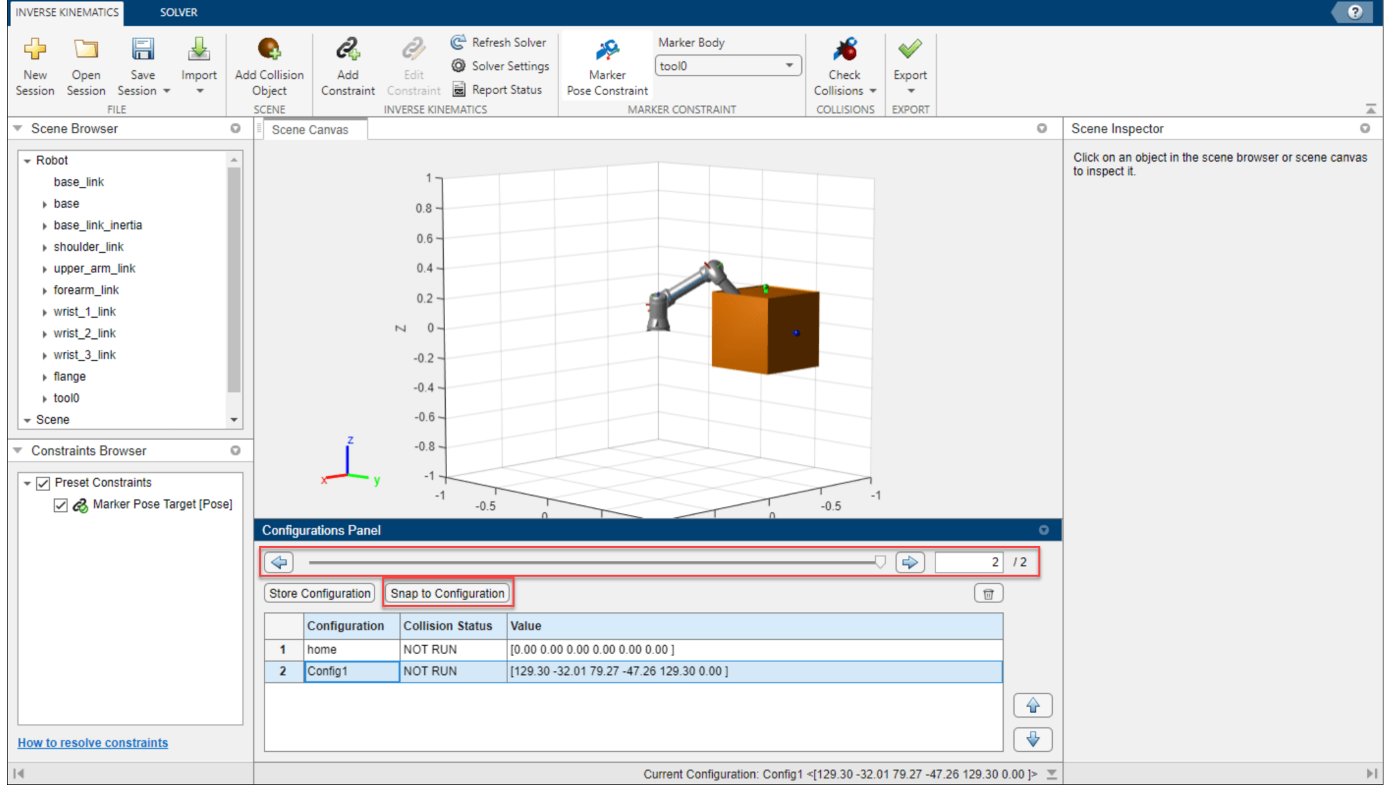
Create Configurations

Use the **Configurations Panel** to create, modify, and view configurations.

Before moving the robot, click **Store Configuration** to save the current joint configuration of the robot as shown in the **Scene Canvas**. This adds the configuration to the table in the **Configurations Panel** with a default configuration name, the **Collision Status**, and **Value** of each joint as a vector. Both the name and value of each configuration can be edited by double-clicking the respective element. Rename this configuration to `home` and leave its **Value** at `[0.00 0.00 0.00 0.00 0.00 0.00]`.

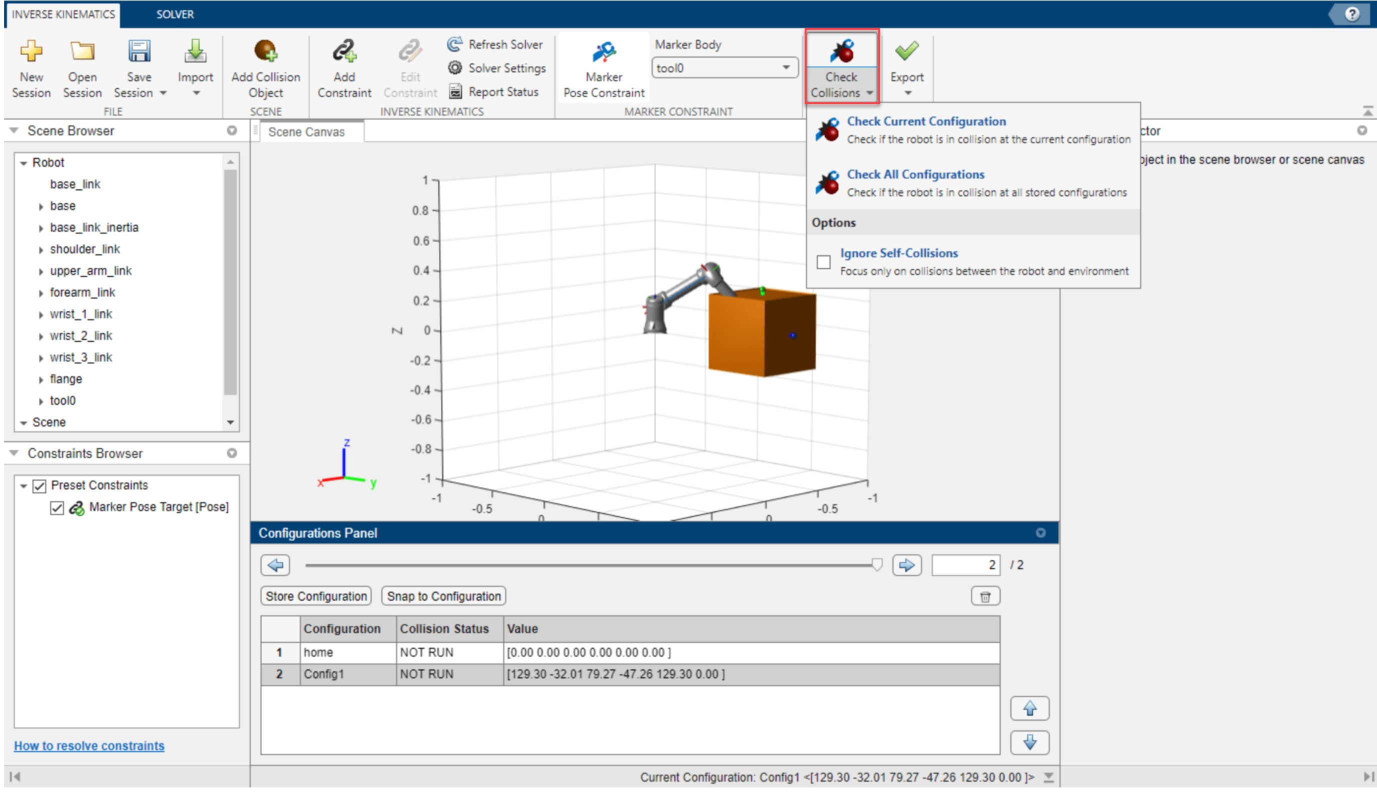


Create another configuration, but this time making it collide with the box. Set the end-effector into the center of the box at `[-0.5 0.5 0]` using the Marker Pose Constraint, and store the configuration. To switch the view between multiple configurations, select a configuration, click **Snap to Configuration** or click the forward or backward step buttons to step to switch the current configuration.

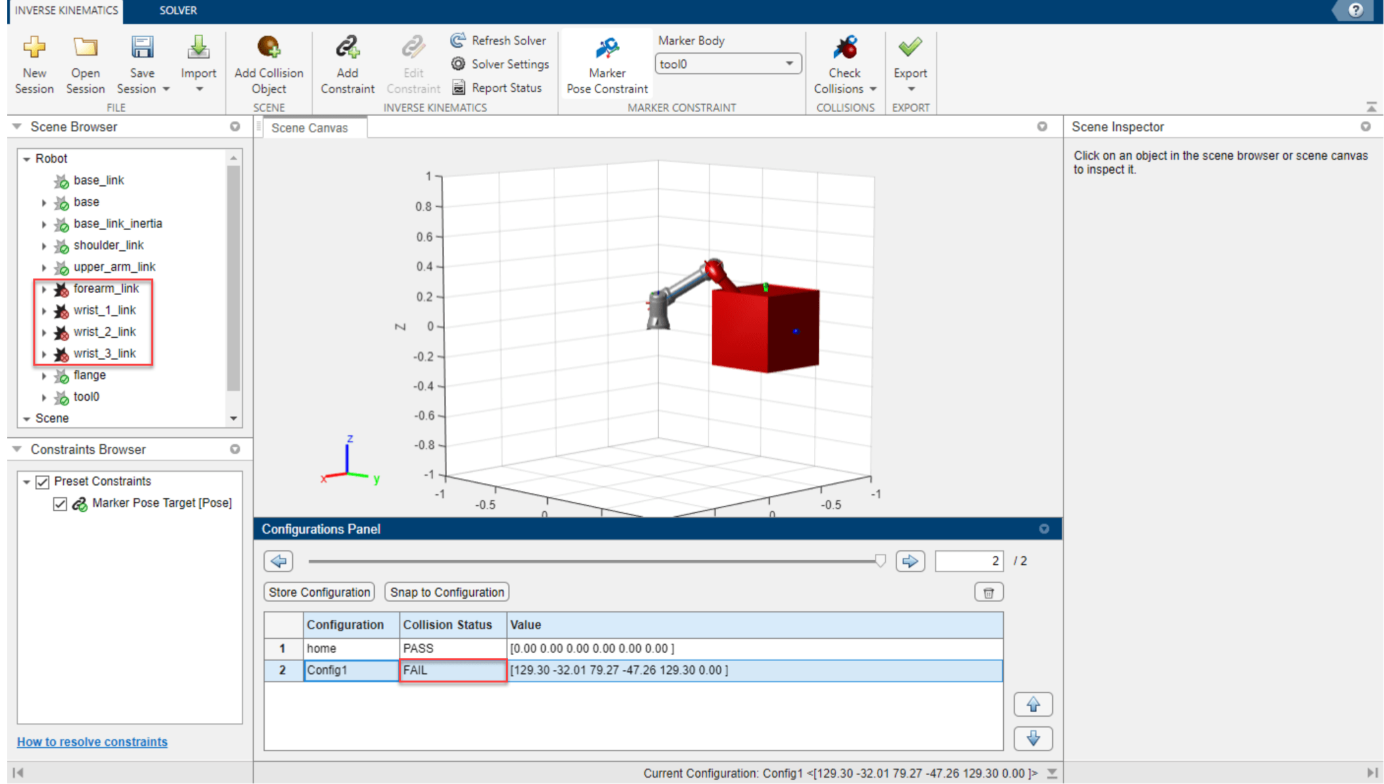


Check Collisions

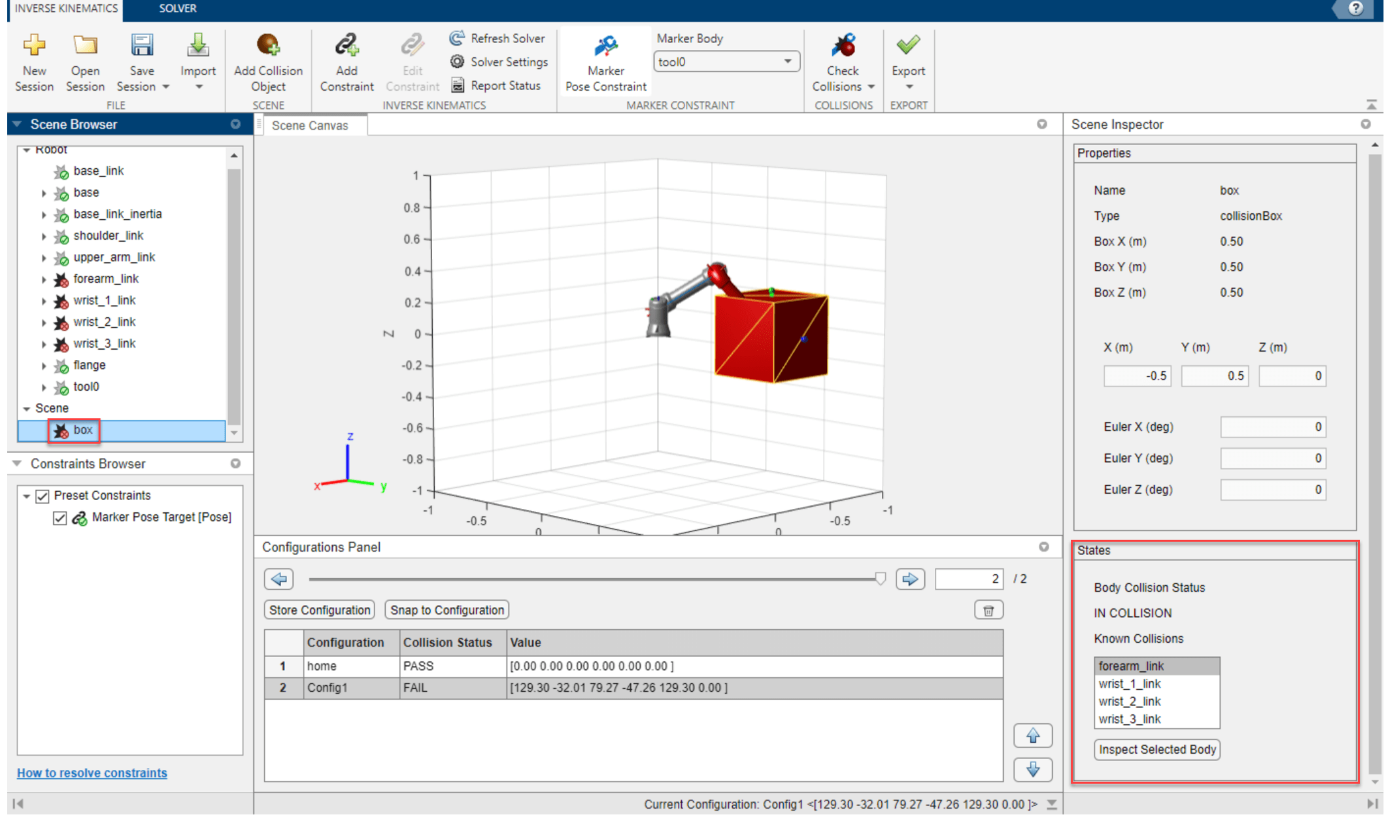
Click **Check Collisions > Check All Configurations** to update the Collision Status of all the stored configurations. To check one configuration, select desired configuration, click **Snap to Configuration**, and click **Check Collisions > Check Current Configuration** to update the **Collision Status** of the currently selected configuration.



After the collision checking, the **Collision Status** of the first and second configuration contain `PASS` and `FAIL` respectively. Select the configuration that failed the collision check. The bodies in the **Scene Browser** update for the selected configuration to display an red x or green check icon, indicating that the body is in-collision or collision-free respectively. The bodies marked as in-collision will also be highlighted in red in the **Scene Canvas**. Note that `flange` and `tool0` are indicated to be collision-free even though they appear to be positionally in-collision. This is because those bodies are just frames and do not contain collision meshes. If you intend to check for collisions, ensure before importing any robot that the physical bodies of your robot contain collision meshes.

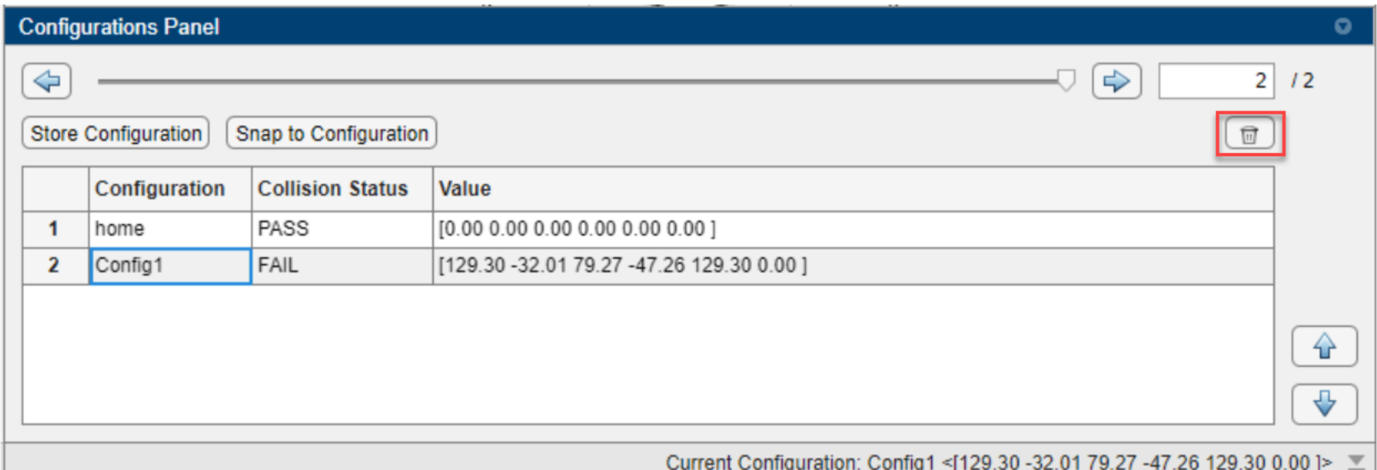


Select the box, either by clicking on it in the **Scene Browser**, or in the **Scene Canvas**, and then inspect the **States** pane. The **States** pane contains the position, orientation, collision status, and a list of all the known collisions since the last collision check. The **Known Collisions** list shows all of the bodies colliding with the selected body. Selecting any of the bodies from the list and clicking **Inspect Selected Body** will switch the **Scene Inspector** to that body.

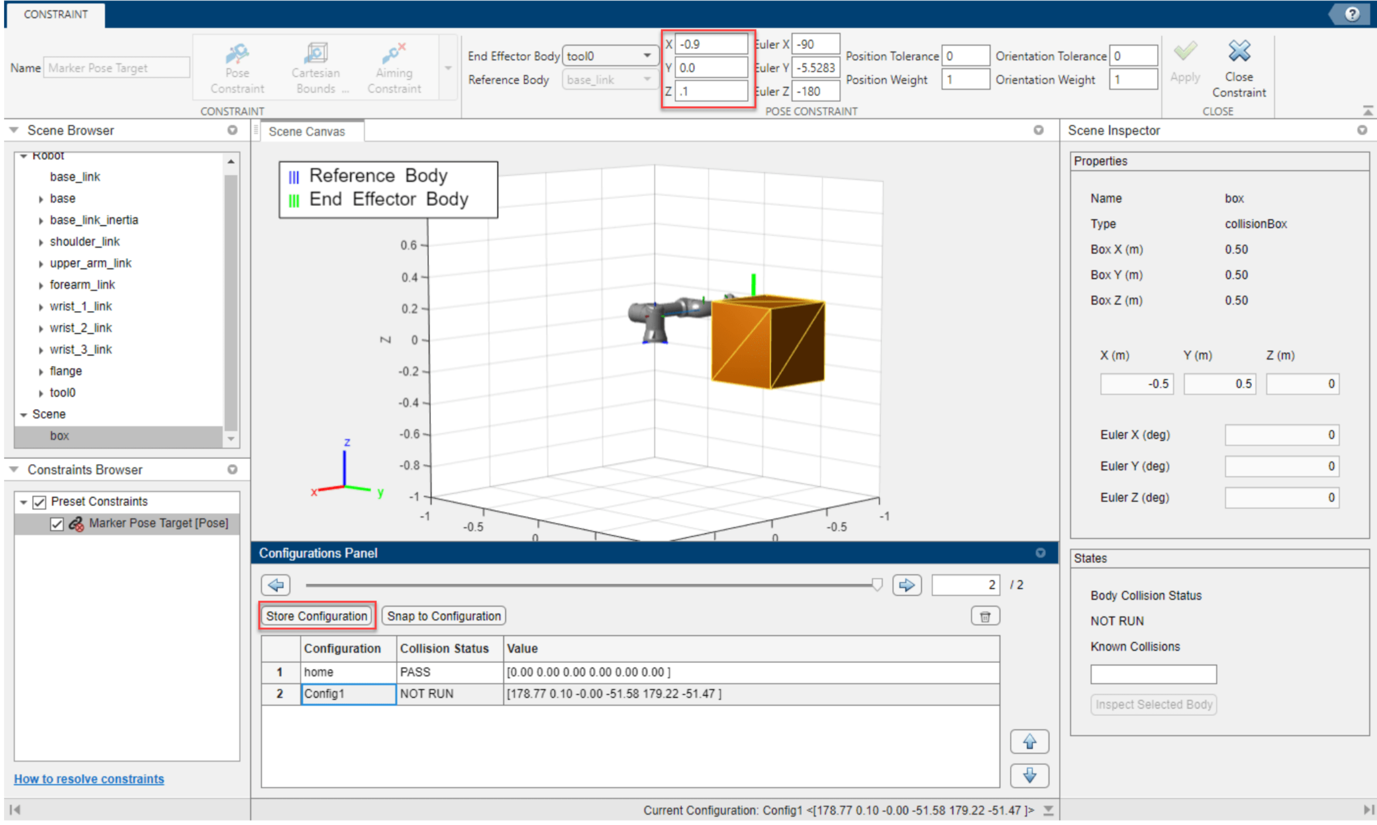


Create Configuration Path

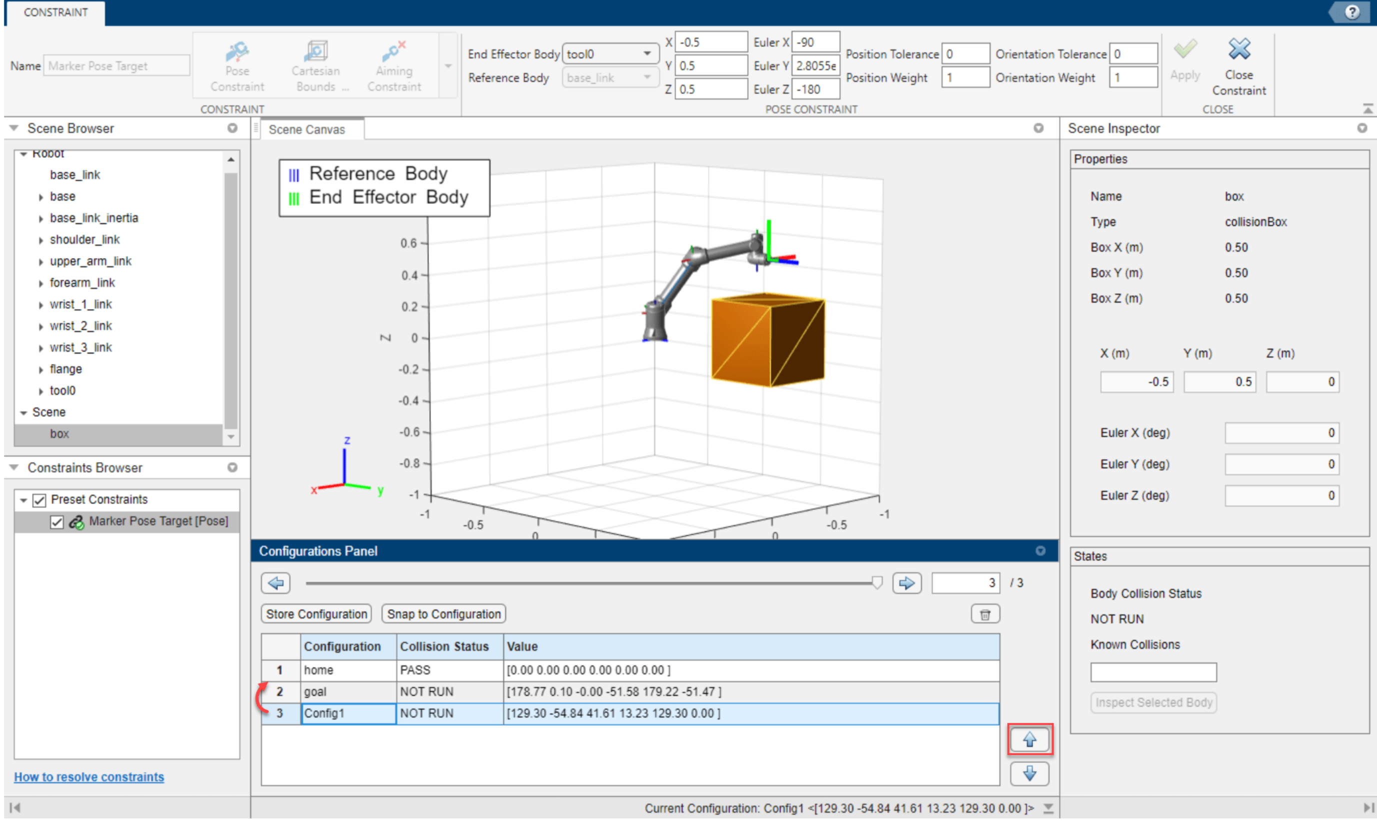
To create a path, add configurations to the table sequential sequentially. Since the second configuration is in-collision with the box, select it and click delete.



Set the target marker pose to behind the box at `[-0.9 0.0 0.1]`, and store the configuration. This configuration will the be goal configuration so rename it as `goal`.



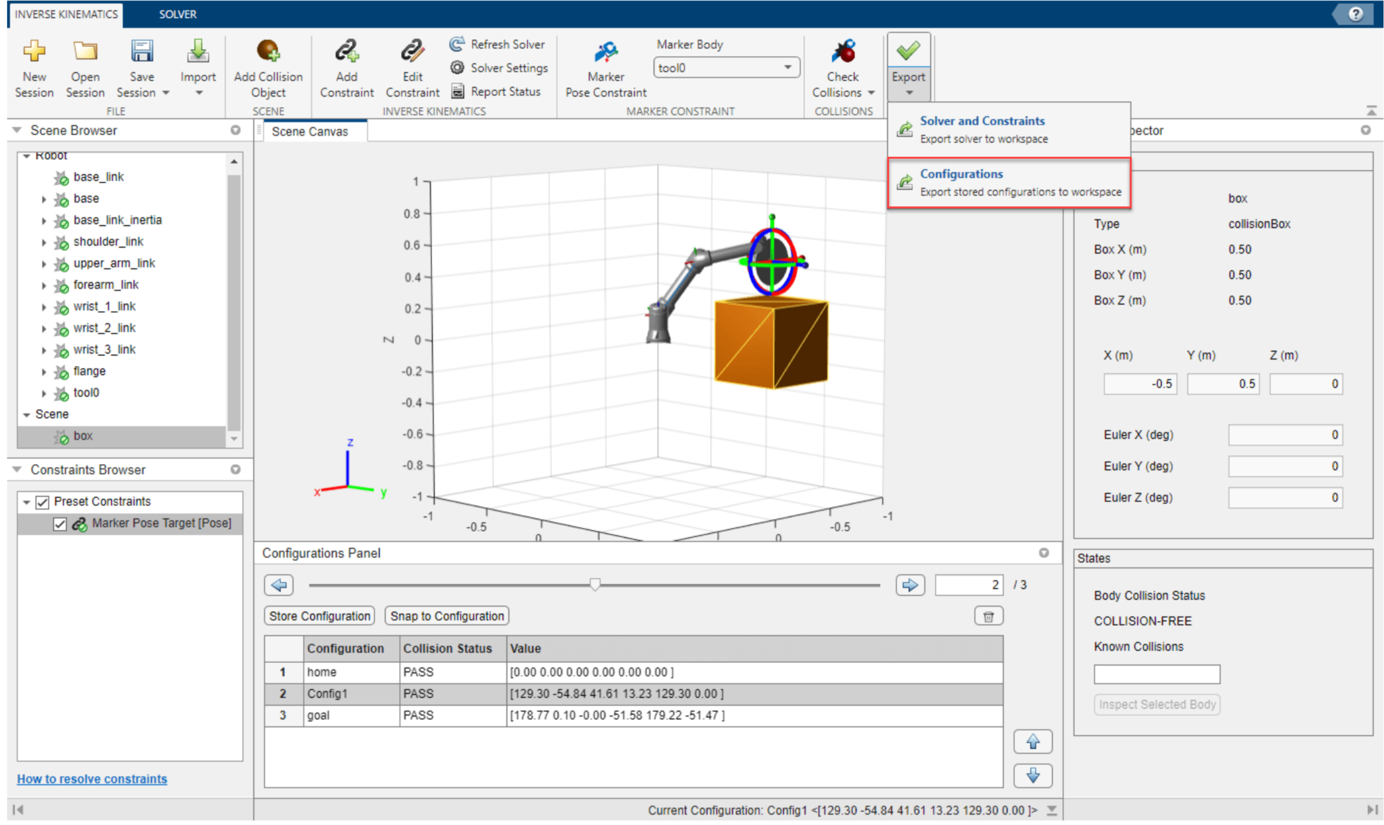
Snap to configuration `home` and add an additional configuration over the box at `[-0.5 0.5 0.5]` to act as an intermediate configuration between `home` and `goal`. If a configuration needs to be modified, snap to that configuration, adjust the target marker pose, save a new configuration, and delete the old configuration. Click the move configuration buttons to move the new configuration between the `home` and `goal` configurations.

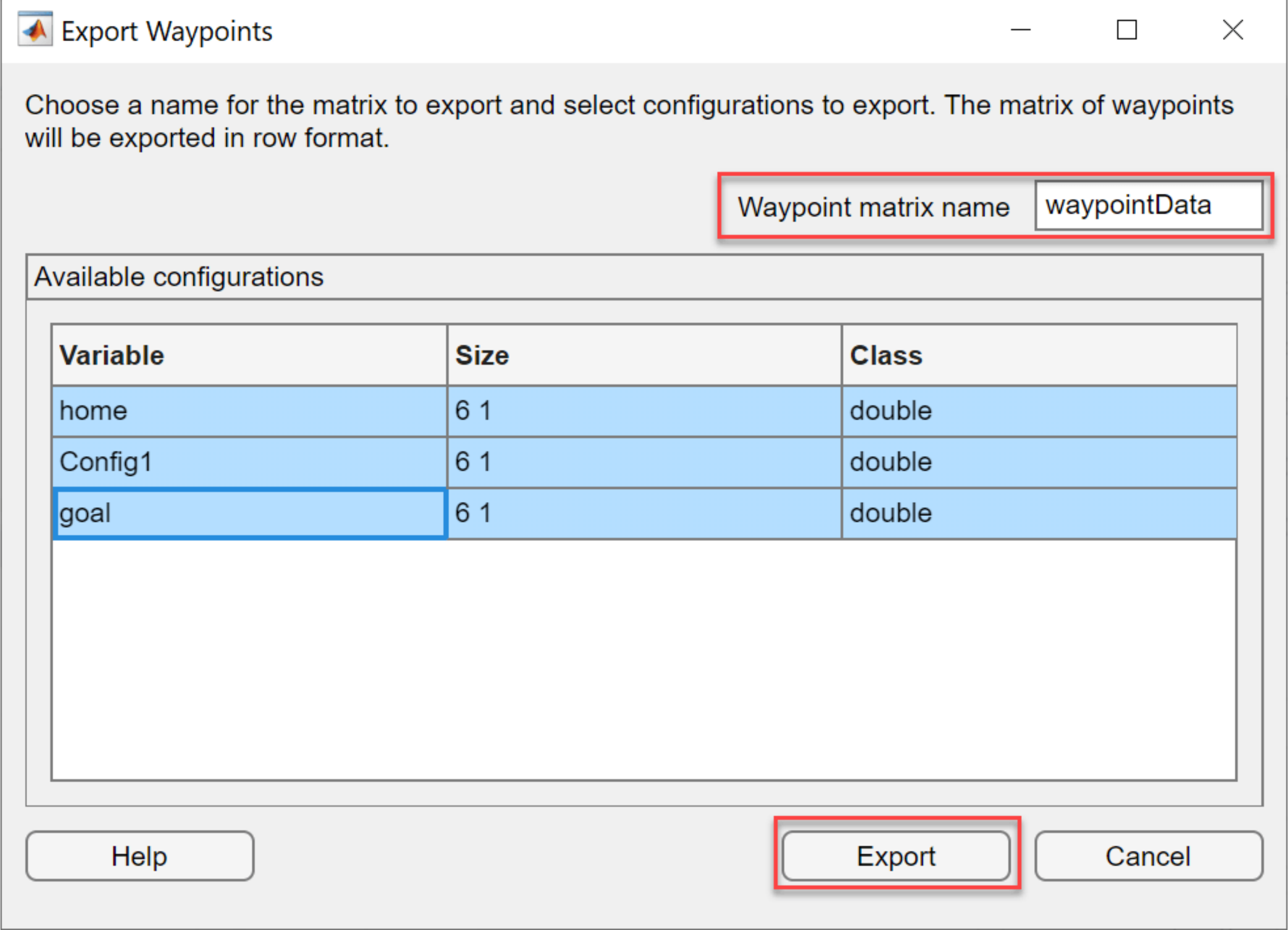


Click **Check Collisions > Check All Configurations** to check all the configurations for collisions. If a configuration does not pass, adjust it as necessary.

Export Configurations as Waypoints

Saving the session will also save the stored configurations within the session, but to export the configurations as waypoints to the MATLAB™ Workspace, click **Export > Configurations**. Select all of the configurations to export, specify the name of the waypoint matrix in **Waypoint matrix name** and click **Export**. Check your workspace for a matrix containing waypoints. Note that the size of the waypoint matrix is dependent on the number of configurations exported and the number of joints of the robot, and will be exported in row format. The dimensions of the waypoint matrix for this example is 3x6.


INVERSE KINEMATICS
SOLVER
New Session
Open Session
Save Session
Import
FILE
Add Collision Object
SCENE
Add Constraint
Edit Constraint
Refresh Solver
Solver Settings
Report Status
INVERSE KINEMATICS
Marker Pose Constraint
Marker Body
tool0
MARKER CONSTRAINT
Check Collisions
COLLISIONS
Export
Solver and Constraints
Export solver to workspace
Configurations
Export stored configurations to workspace
Scene Browser
Robot
base_link
base
base_link_inertia
shoulder_link
upper_arm_link
forearm_link
wrist_1_link
wrist_2_link
wrist_3_link
flange
tool0
Scene
box
Constraints Browser
Preset Constraints
Marker Pose Target [Pose]
How to resolve constraints
Scene Canvas
Configurations Panel
Store Configuration
Snap to Configuration
2 / 3
Configuration
Collision Status
Value
1 home PASS [0.00 0.00 0.00 0.00 0.00 0.00]
2 Config1 PASS [129.30 -54.84 41.61 13.23 129.30 0.00]
3 goal PASS [178.77 0.10 -0.00 -51.58 179.22 -51.47]
Current Configuration: Config1 <[129.30 -54.84 41.61 13.23 129.30 0.00]>
box
Type collisionBox
Box X (m) 0.50
Box Y (m) 0.50
Box Z (m) 0.50
X (m) Y (m) Z (m)
-0.5 0.5 0
Euler X (deg) 0
Euler Y (deg) 0
Euler Z (deg) 0
States
Body Collision Status
COLLISION-FREE
Known Collisions
Inspect Selected Body



- “Plan Manipulator Path for Dispensing Task Using Inverse Kinematics Designer”
- “Create Constrained Inverse Kinematics Solver Using Inverse Kinematics Designer”

Parameters

Session

`Inverse Kinematics` — Manage session, check collisions, import and export solver/constraints/configurations
tab

Use the **Inverse Kinematics** tab to manage session, add and edit constraints, check collisions for configurations, and import and export solvers, constraints, and configurations. The parameters and buttons in this tab are usable only after a session has been created or loaded.

Parameters/Buttons	Description
New Session	Click **New Session** to load a specified robot from the workspace or from the lists of robots in the robot library using the **New Session** dialog box.
Open Session	Click **Open Session** to load a saved session MAT-file.
Save Session	Click **Save Session** to save the current session as a MAT-file.
Import	• Click **Import > Solver** to import a solver from the workspace using the **Import Inverse Kinematics Solver** dialog box. • Click **Import > Constraints** to import constraint(s) from the workspace using the **Import Inverse Kinematics Constraints** dialog box. • Click **Import > Configurations** to import configuration(s) from the workspace using the **Import Joint Configurations** dialog box.
Add Collision Object	Click **Add Collision Object** to add collision objects to the scene from the workspace using the **Add Collision Object** dialog box.
Add Constraint	Click **Add Constraint** to add a constraint to the inverse kinematics solver using the **Constraint** tab.
Edit Constraint	Click **Edit Constraint** to edit the selected solver constraint using the **Constraint** tab.
Refresh Solver	Click **Refresh Solver** to run the inverse kinematics solver using the current configuration as the initial guess.
Solver Settings	Click **Solver Settings** to edit the inverse kinematics solver settings using the **Solver** tab.
Report Status	Click **Report Status** to view the status of the most recent inverse kinematics solver solution using the **Solution Report** dialog box.
Marker Pose Constraint	Click **Marker Pose Constraint** to edit the Marker Pose Target from the **Constraint** tab.
Marker Body	Select a body from the list of bodies in the loaded robot model to be constrained by the marker pose target constraint. By default, **Marker Body** is set to the last body in the robot model.

Parameters/Buttons	Description
Check Collisions	• Click **Check Collisions > Check Current Configuration** to check the current configuration for collisions with collision objects in the scene. • Click **Check Collisions > Check All Configurations** to check the all stored configurations for any collisions with collision objects in the scene. • Select **Ignore Self-Collisions** to ignore collisions between bodies of the robot.
Export	• Click **Export > Solver and Constraints** to export the current solver and constraints to the workspace as objects using the **Export Solver and Constraints** dialog box. • Click **Export > Configurations** to export configurations from the **Configurations Panel** to the workspace as a matrix using the **Export Waypoints** dialog box.

`Solver` — Solver settings
tab

These parameters specify settings for the inverse kinematics solver. To access these parameters, open the **Solver** tab. To open the Solver tab, on the Inverse Kinematics tab, select **Solver Settings**.

Parameters	Description
Solver Algorithm	Inverse kinematics solver algorithm, specified as one of these options: • `BFGS Gradient Projection` (default) • `Levenberg-Marquardt` For more information on how to select a solver, see “Choose an Algorithm”.
Max Iterations	Maximum number of iterations for the inverse kinematics solver to run, specified as a positive integer. The default value is `50`.
Max Time	Maximum time that the inverse kinematics solver can search for a solution, specified as a positive scalar in seconds. The default value is `5`.
Enforce Joint Limits	Select to set the inverse kinematics solver to enforce joint limits from the robot model. The default value is `on`. For more details about enforcing joint limits, see “Solver Parameters”.

Parameters	Description
Allow Random Restart	Select to allow the inverse kinematics solver to restart with a different randomly generated initial guess if the algorithm approaches a solution that does not satisfy the constraints. The default value is `on`. For more details about enforcing joint limits, see “Solver Parameters”.

Click **Reset Settings > Reset Settings** and **Reset Settings > Reset To Default Settings** to reset the inverse kinematics solver settings to the last stored value and default settings respectively.

After editing the solver settings, click **Apply to Solver** to apply changes to the current inverse kinematics solver.

`Constraint` — Add or edit constraint
tab

To open the **Constraint** tab and create a constraint, on the **Inverse Kinematics** tab, select **Add Constraint**. Set the name of the constraint and select the type of constraint. The **Constraint** tab also contains a section with parameters for the selected constraint:

- **Pose Constraint** — Constraint that assigns a target pose to a body. See the corresponding tab section **Pose Constraint**, and MATLAB object `constraintPoseTarget`, for more information.
- **Cartesian Bounds Constraint** — Constraint that defines a bounded target region for an end-effector pose. See the corresponding tab section **Cartesian Bounds Constraint**, and MATLAB object `constraintCartesianBounds`, for more information.
- **Aiming Constraint** — Constraint that assigns custom joint limits. See the corresponding tab section, **Aiming Constraint**, and MATLAB object `constraintAiming` for more information.
- **Joint Bounds Constraint** — Constraint that assigns custom joint limits. See the corresponding tab section, **Joint Bounds Constraint**, and MATLAB object `constraintJointBounds` for more information.
- **Distance Bounds Constraint** — Constraint that defines a minimum and maximum distance that two bodies can be from each other. See the corresponding tab section, **Distance Bounds Constraint**, and MATLAB object `constraintDistanceBounds` for more information.
- **Fixed Joint Constraint** — Constraint that fixes the position and orientation between two bodies. See the corresponding tab section, **Fixed Joint Constraint**, and MATLAB object `constraintFixedJoint` for more information.
- **Revolute Joint Constraint** — Constraint that simulates revolute motion between two bodies. See the corresponding tab section **Revolute Joint Constraint**, and MATLAB object `constraintRevoluteJoint`, for more information.
- **Prismatic Joint Constraint** — Constraint that simulates prismatic motion between two bodies. See the corresponding tab section, **Prismatic Joint Constraint**, and MATLAB object `constraintPrismaticJoint` for more information.

After setting the constraint parameters, click **Apply** to save your changes and select **Close Constraint** to return to the **Inverse Kinematics** tab. If you do not click **Apply** before closing the **Constraint** tab, **Inverse Kinematics Designer** does not save your changes..

`New Session` — Start new session and select robot model

dialog box

Start new session and select robot model as a `rigidBodyTree` either from the **Rigid body tree** list or the MATLAB workspace. To import a robot into the workspace, use either the `loadrobot` function or `importrobot` function.

For more information about how to create a rigid body tree, see "Build Basic Rigid Body Tree Models".

For a full list of all the rigid body trees included with the Robotics System Toolbox, see the `robotname` input of `loadrobot`.

`Add Collision Object` — Add collision meshes from workspace

dialog box

Add collision meshes from the MATLAB workspace, specified as `collisionBox`, `collisionCylinder`, `collisionSphere`, or `collisionMesh` objects. The **Add Collision Object** dialog box is accessible by clicking **Add Collision Object** in the **Inverse Kinematics** tab.

`Solution Report` — Details of most recent inverse kinematics solution
dialog box

View the details of the most recent inverse kinematics solution in the Solution Report dialog box. To access these parameters, on the Inverse Kinematics tab, select **Report Status**.

These parameters are read-only.

Parameter	Description
Iterations	Number of iterations needed to achieve the solution result.
Number of Random Restarts	Number of times that the solution randomly restarted. Random restarts are triggered when the algorithm approaches a solution that does not satisfy the constraints. The solver restarts with a randomly generated initial guess.
Constraint Violations	Constraint violations, indicated as a 1-by-*N* structure array, where *N* is the number of enabled constraints in the session. Select `1x`*`N`* `struct array` to print the details of each constraint violation in the Command Window. See "Constraint Violation Format" on page 5-38 for more details.

Parameter	Description
Status	Status of the solution, indicated as either `Success` or `Best available`. `Success` indicates that the solver successfully reached a configuration that satisfies all constraints. `Best available` indicates that the solver did not reach a configuration that satisfies all constraints, and is showing the best available configuration it could achieve.
Exit Flag	Exit flag of the solver. See Exit Flags for more information.

Constraints

`Pose Constraint` — Pose constraint settings
tab section

Click **Pose Constraint** in the **Constraint** tab to open the **Pose Constraint** tab section to create a constraint.

Parameter	Description
End Effector Body	End-effector body, specified as a selection from a list of bodies in the loaded robot model. The default value is set to the last body in the loaded robot model.
Reference Body	Reference body, specified as a selection from a list of bodies in the loaded robot model. The default value is set to the first body in the loaded robot model.
X	Target *x* position of the end effector in the reference frame of the **Reference Body**, specified as a scalar in meters. The default value is `0`.
Y	Target *y* position of the end effector in the reference frame of the **Reference Body**, specified as a scalar in meters. The default value is `0`.
Z	Target *z* position of the end effector in the reference frame of the **Reference Body**, specified as a scalar in meters. The default value is `0`.
Euler X	Target Euler *x* rotation of the end-effector body in the reference frame of the **Reference Body**, specified as a scalar in degrees. The default value is `0`.

Parameter	Description
Euler Y	Target Euler *y* rotation of the end-effector body in the reference frame of the **Reference Body**, specified as a scalar in degrees. The default value is `0`.
Euler Z	Target Euler *z* rotation of the end-effector body in the reference frame of the **Reference Body**, specified as a scalar in degrees. The default value is `0`.
Position Tolerance	Tolerance for the end-effector position, specified as a nonnegative scalar in meters. The default value is `0.01`.
Position Weight	Weight for the end-effector position, specified as a nonnegative scalar. The default value is `1`.
Orientation Tolerance	Tolerance for the end-effector orientation, specified as a nonnegative scalar in degrees. The default value is `1`.
Orientation Weight	Weight for the end-effector orientation, specified as a nonnegative scalar. The default value is `1`.

`Cartesian Bounds Constraint` — Cartesian bounds constraint settings
tab section

Click **Cartesian Bounds Constraint** in the **Constraint** tab to open the **Cartesian Bounds Constraint** tab section to create a constraint.

Parameter	Description
End Effector Body	End-effector body, specified as a selection from a list of bodies in the loaded robot model. The default value is set to the last body in the loaded robot model.
Reference Body	Reference body, specified as a selection from a list of bodies in the loaded robot model. The default value is set to the first body in the loaded robot model.
X	Target *x* position of the end effector in the reference frame of the **Reference Body**, specified as a scalar in meters. The default value is `0`.
Y	Target *y* position of the end effector in the reference frame of the **Reference Body**, specified as a scalar in meters. The default value is `0`.
Z	Target *z* position of the end effector in the reference frame of the **Reference Body**, specified as a scalar in meters. The default value is `0`.

Parameter	Description
Euler X	Target Euler *x* rotation of the end-effector body in the reference frame of the **Reference Body**, specified as a scalar in degrees. The default value is `0`.
Euler Y	Target Euler *y* rotation of the end-effector body in the reference frame of the **Reference Body**, specified as a scalar in degrees. The default value is `0`.
Euler Z	Target Euler *z* rotation of the end-effector body in the reference frame of the **Reference Body**, specified as a scalar in degrees. The default value is `0`.
X Min	Minimum *x* bound on end-effector position relative to the reference frame of the **Reference Body**, specified as a scalar in meters. The default value is `-0.5`.
Y Min	Minimum *y* bound on end-effector position relative to the reference frame of the **Reference Body**, specified as a scalar in meters. The default value is `-0.5`
Z Min	Minimum *z* bound on end-effector position relative to the reference frame of the **Reference Body**, specified as a scalar in meters. The default value is `-0.5`
X Max	Maximum *x* bound on end-effector position relative to the reference frame of the **Reference Body**, specified as a scalar in meters. The default value is `0.5`
Y Max	Maximum *y* bound on end-effector position relative to the reference frame of the **Reference Body**, specified as a scalar in meters. The default value is `0.5`
Z Max	Maximum *z* bound on end-effector position relative to the reference frame of the **Reference Body**, specified as a scalar in meters. The default value is `0.5`
X Weight	Weight of the constraint on *x* bound, specified as a scalar. The default value is `1`.
Y Weight	Weight of the constraint on *y* bound, specified as a scalar. The default value is `1`.
Z Weight	Weight of the constraint on *z* bound, specified as a scalar. The default value is `1`.

`Aiming Constraint` — Aiming constraint settings
tab section

Click **Aiming Constraint** in the **Constraint** tab to open the **Aiming Constraint** tab section to create a constraint.

Parameter	Description
End Effector Body	End-effector body, specified as a selection from a list of bodies in the loaded robot model. The default value is set to the last body in the loaded robot model.
Reference Body	Reference body, specified as a selection from a list of bodies in the loaded robot model. The default value is set to the first body in the loaded robot model.
X Target	Target *x* position relative to **Reference Body**, specified as a scalar in meters. The default value is `0`.
Y Target	Target *y* position relative to **Reference Body**, specified as a scalar in meters. The default value is `0`.
Z Target	Target *z* position relative to **Reference Body**, specified as a scalar in meters. The default value is `0`.
Angular Tolerance	Maximum allowed angular tolerance, specified as a nonnegative scalar in degrees. The default value is `1`.
Constraint Weight	Weight of constraint, specified as a scalar. The default value is `1`.

`Joint Bounds Constraint` — Joint bounds constraint settings
tab section

Click **Joint Bounds Constraint** in the **Constraint** tab to open the **Joint Bounds Constraint** tab section to create a constraint.

Parameter	Description
Upper Joint Limits	Upper joint limit angles, specified as a *N*-element row vector of angles in degrees, where *N* is the number of moving joints in the robot model. The default value is `180*ones(1,N)`.
Lower Joint Limits	Lower joint limit angles, specified as a *N*-element row vector of angles in degrees, where *N* is the number of moving joints in the robot model. The default value is `-180*ones(1,N)`.
Joint Limit Weights	Joint limit weights, specified as a *N*-element row vector where *N* is the number of moving joints in the robot model. The default value is `ones(1,N)`.

`Distance Bounds Constraint` — Distance bounds constraint settings
tab section

On the **Constraint** tab, select **Distance Bounds Constraint** in the **Constraint** tab to create a distance bounds constraint and open the **Distance Bounds Constraint** section.

Parameter	Description
End Effector Body	End-effector body, selected from a list of bodies in the loaded robot model. The default value is the last body in the loaded robot model.
Reference Body	Reference body, selected from a list of bodies in the loaded robot model. The default value is the first body in the loaded robot model.
Max Distance	Maximum distance bound on the end-effector position relative to the reference frame of the **Reference Body**, specified as a scalar in meters. The default value is `0`.
Min Distance	Minimum distance bound on end-effector position relative to the reference frame of the **Reference Body**, specified as a scalar in meters. The default value is `0`.
Distance Weight	Weight for the end-effector position, specified as a nonnegative scalar. The default value is `1`.

`Fixed Joint Constraint` — Fixed joint constraint settings
tab section

On the **Constraint** tab, select **Fixed Joint Constraint** in the **Constraint** tab to create a fixed joint constraint and open the **Fixed Joint Constraint** section.

Parameter	Description
Successor Body	Successor body, selected from a list of bodies in the loaded robot model. The default value is the last body in the loaded robot model
Predecessor Body	Predecessor body, selected from a list of bodies in the loaded robot model. The default value is the first body in the loaded robot model.
Position Tolerance	Tolerance for the end-effector position, specified as a nonnegative scalar in meters. The default value is `0.01`.
Position Weight	Weight for the end-effector position, specified as a nonnegative scalar. The default value is `1`.
Orientation Tolerance	Tolerance for the end-effector orientation, specified as a nonnegative scalar in degrees. The default value is `1`.
Orientation Weight	Weight for the end-effector orientation, specified as a nonnegative scalar. The default value is `1`.

To set the successor and predecessor transforms, select **Successor Transform** or **Predecessor Transform**, respectively:

Parameter	Description
X	Target *x*-position of the successor or predecessor transform in the reference frame of the corresponding body. specified as a scalar in meters. The default value is `0`.
Y	Target *y*-position of the successor or predecessor transform in the reference frame of the corresponding body, specified as a scalar in meters. The default value is `0`.
Z	Target *z*-position of the successor or predecessor transform in the reference frame of the corresponding body, specified as a scalar in meters. The default value is `0`.
Euler X	Target Euler *x*-rotation of the successor or predecessor transform in the reference frame of the corresponding body, specified as a scalar in degrees. The default value is `0`.
Euler Y	Target Euler *y*-rotation of the successor or predecessor transform in the reference frame of the corresponding body, specified as a scalar in degrees. The default value is `0`.
Euler Z	Target Euler *z*-rotation of the successor or predecessor transform in the reference frame of the corresponding body, specified as a scalar in degrees. The default value is `0`.
Import from workspace	Import a transform from the workspace as an `se3` transformation object.

`Prismatic Joint Constraint` — Prismatic joint constraint settings
tab section

On the **Constraint** tab, select **Prismatic Joint Constraint** in the **Constraint** tab to create a prismatic joint constraint and open the **Prismatic Joint Constraint** tab.

Parameter	Description
Successor Body	Successor body, selected from a list of bodies in the loaded robot model. The default value is the last body in the loaded robot model
Predecessor Body	Predecessor body, selected from a list of bodies in the loaded robot model. The default value is the first body in the loaded robot model.
Position Tolerance	Tolerance for the position of the aligned successor and predecessor transforms, specified as a nonnegative scalar in meters. The default value is `0.01`.

Parameter	Description
Position Weight	Weight for the position of the aligned successor and predecessor transforms, specified as a nonnegative scalar. The default value is `1`.
Orientation Tolerance	Tolerance for the orientation of the aligned successor and predecessor transforms, specified as a nonnegative scalar in degrees. The default value is `1`.
Orientation Weight	Weight for the orientation of the aligned successor and predecessor transforms, specified as a nonnegative scalar. The default value is `1`.
Joint Limits	Position limits of the joint constraint, specified as a two-element row vector of the form `[minimum maximum]`, in meters. The default value is `[-1 1]`.
Joint Limits Weight	Weight for the joint position limits, specified as a nonnegative scalar. The default value is `1`.

To set the successor and predecessor transforms, select **Successor Transform** or **Predecessor Transform**, respectively:

Parameter	Description
X	Target *x*-position of the successor or predecessor transform in the reference frame of the corresponding body. specified as a scalar in meters. The default value is `0`.
Y	Target *y*-position of the successor or predecessor transform in the reference frame of the corresponding body, specified as a scalar in meters. The default value is `0`.
Z	Target *z*-position of the successor or predecessor transform in the reference frame of the corresponding body, specified as a scalar in meters. The default value is `0`.
Euler X	Target Euler *x*-rotation of the successor or predecessor transform in the reference frame of the corresponding body, specified as a scalar in degrees. The default value is `0`.
Euler Y	Target Euler *y*-rotation of the successor or predecessor transform in the reference frame of the corresponding body, specified as a scalar in degrees. The default value is `0`.
Euler Z	Target Euler *z*-rotation of the successor or predecessor transform in the reference frame of the corresponding body, specified as a scalar in degrees. The default value is `0`.

Parameter	Description
Import from workspace	Import a transform from the workspace as an `se3` transformation object.

`Revolute Joint Constraint` — Revolute joint constraint settings
tab section

On the **Constraint** tab, select **Revolute Joint Constraint** in the **Constraint** tab to create a revolute joint constraint and open the **Revolute Joint Constraint** tab.

Parameter	Description
Successor Body	Successor body, selected from a list of bodies in the loaded robot model. The default value is the last body in the loaded robot model
Predecessor Body	Predecessor body, selected from a list of bodies in the loaded robot model. The default value is the first body in the loaded robot model.
Position Tolerance	Tolerance for the position of the aligned successor and predecessor transforms, specified as a nonnegative scalar in meters. The default value is `0.01`.
Position Weight	Weight for the position of the aligned successor and predecessor transforms, specified as a nonnegative scalar. The default value is `1`.
Orientation Tolerance	Tolerance for the orientation of the aligned successor and predecessor transforms, specified as a nonnegative scalar in degrees. The default value is `1`.
Orientation Weight	Weight for the orientation of the aligned successor and predecessor transforms, specified as a nonnegative scalar. The default value is `1`.
Joint Limits	Position limits of the joint constraint, specified as a two-element row vector of the form `[minimum maximum]`, in degrees. The default value is `[-180 180]`.
Joint Limits Weight	Weight for the joint position limits, specified as a nonnegative scalar. The default value is `1`.

To set the successor and predecessor transforms, select **Successor Transform** or **Predecessor Transform**, respectively:

Parameter	Description
X	Target *x*-position of the successor or predecessor transform in the reference frame of the corresponding body. specified as a scalar in meters. The default value is `0`.

Parameter	Description
Y	Target *y*-position of the successor or predecessor transform in the reference frame of the corresponding body, specified as a scalar in meters. The default value is `0`.
Z	Target *z*-position of the successor or predecessor transform in the reference frame of the corresponding body, specified as a scalar in meters. The default value is `0`.
Euler X	Target Euler *x*-rotation of the successor or predecessor transform in the reference frame of the corresponding body, specified as a scalar in degrees. The default value is `0`.
Euler Y	Target Euler *y*-rotation of the successor or predecessor transform in the reference frame of the corresponding body, specified as a scalar in degrees. The default value is `0`.
Euler Z	Target Euler *z*-rotation of the successor or predecessor transform in the reference frame of the corresponding body, specified as a scalar in degrees. The default value is `0`.
Import from workspace	Import a transform from the workspace as an `se3` transformation object.

Import

`Import Inverse Kinematics Solver` — Import inverse kinematics solver

dialog box

Import an inverse kinematics solver from the MATLAB workspace, specified as either a `generalizedInverseKinematics`, or `inverseKinematics` object. Solvers in the Workspace appear in the **Available solvers** table. Select a solver and click **Import** to import the solver into the **Inverse Kinematics Designer** app. If a solver from the Workspace does not appear in the **Available solvers** table, click **Refresh**.

`Import Joint Configurations` — Import joint configurations

dialog box

Import joint configurations from the MATLAB workspace, specified as an *M*-by-*N* matrix of doubles where *M* is the number of configurations, and *N* is the number of moveable joints in the robot. Configuration data in the Workspace appear in the **Configurations in the workspace** table. Select configuration data and click **Import** to import the configuration data into the **Inverse Kinematics Designer** app.

If a configuration in the Workspace does not appear in the **Configurations in the workspace** table, click **Refresh**.

`Import Inverse Kinematics Constraints` — Import inverse kinematics constraints

dialog box

Import inverse kinematics constraints from the Workspace, specified as a `constraintAiming`, `constraintPoseTarget`, `constraintCartesianBounds`, or `constraintJointBounds` object. Constraint objects in the Workspace appear in the **Configurations in the workspace** table. Select the desired constraint objects and click **Import** to import the constraints into the **Inverse Kinematics Designer** app.

If a configuration in the Workspace does not appear in the **Constraint objects in the workspace** table, click **Refresh**.

`Import Transform` — Import transform

dialog box

Import a transform from the workspace to use as a successor or predecessor transform, specified as an `se3` object. Transforms in the workspace appear in the **SE(3) objects in the workspace** table. Select the desired transform and click **Import** to import the transform into the **Inverse Kinematics Designer** app.

If a transform in the workspace does not appear in the **SE(3) objects in the workspace** table, click **Refresh**.

Export

`Export Solver and Constraints` — Solver and constraint export settings

dialog box

Change solver and constraint export settings, and export the solver and constraints to the MATLAB. To access these parameters, on the Inverse Kinematics tab, select **Export > Solver and Constraints**. When all settings and selections are complete, click **Export** to export the solver and constraints to the workspace.

Parameter	Description
Export solver	Select to include the inverse kinematics solver when exporting. The default value is `on`.
Solver name	Name of the inverse kinematics solver. The default value is `ikSolver`.
Export constraints	Select to include the solver constraints when exporting. The default value is `on`.
Constraints cell array name	Name of the solver constraints cell array, specified as a string. The default value is `ikConstraints`. To enable this parameter, select the **Export constraints** parameter.

Parameter	Description
Available constraints	Table containing solver constraints, listing the name, size, and class of each constraint. Select constraints from this table and click **Export** to export the constraints to the workspace. To enable this parameter, select the **Export constraints** parameter.

`Export Waypoints` — Export configurations as waypoints

dialog box

Export configurations as waypoints using the Export Waypoints dialog box. To access these parameters, on the Inverse Kinematics tab, select **Export > Configurations**. When selections are complete, click **Export** to export the waypoints as an *M*-by-*N* matrix of data type `double` to the MATLAB workspace, where *M* is the number of waypoints, and *N* is the number of movable joints in the robot.

Parameter	Description
Waypoint matrix name	Name of waypoint matrix. The default value is `waypointData`.
Available configurations	Table containing available configurations, listing the name, size, and class of each constraint. Select configurations from this table and click **Export** to export the configurations to the workspace as waypoints.

Programmatic Use

`inverseKinematicsDesigner` opens the **Inverse Kinematics Designer** app.

`inverseKinematicsDesigner(sessionFileName)` opens the **Inverse Kinematics Designer** app and loads the specified inverse kinematics session MAT file that was previously saved from the app.

More About

Resolving Constraint Conflict

Constraint conflicts occur when constraints cannot be met by the solver. They are indicated in the **Constraints Browser** with a red symbol next to the corresponding constraint. When the constraint is met during the most recent solution, the **Constraints Browser** indicates this with a green symbol next to the corresponding constraint. To solve these conflicts, troubleshoot your constraints based on the last action you performed:

- If you moved the marker pose constraint, check if the pose you specified is within the bounds of the robot. If it is not within the bounds, try moving it to somewhere within the bounds and see if this resolves the conflict. If the constraint conflict does not resolve, the constraint may be in conflict with joint limits or another constraint. In this case, consider modifying the parameters of the marker pose constraint, such as the weights.

Note The solver assumes priority using the assigned weights of each constraint when attempting to satisfy each constraint.

- If you added a constraint, verify if it is causing the conflict by clearing that constraint in the **Constraints Browser** and clicking **Refresh Solver**. If all other constraints resolve, edit the parameters of this constraint, or ensure that the existing unresolved constraints have the desired parameter values, to resolve the conflict.
- If you modified a constraint, change the parameters of that constraint, if applicable, or edit the parameters of other unresolved constraints. Consider modifying the solver weights of each constraint, or disabling constraints to see if some constraint conflicts can be resolved independently.

Constraint Violation Format

When you select the constraint violations structure array from the **Solution Report** dialog box, the **Inverse Kinematics Designer** prints the type, violation magnitude, and name of each enabled constraint in the **Constraints Browser** to the MATLAB Command Window. The structure contains these fields for each constraint:

- `Type` — Type of constraint, represented as:
  - `'pose'` — Pose constraint.
  - `'aiming'` — Aiming constraint.
  - `'joint'` — Joint constraint.
  - `'cartesian'` — Cartesian constraint.
  - `'distance'` — Distance bounds constraint.
  - `'fixedjoint'` — Fixed joint constraint.
  - `'revolutejoint'` — Revolute joint constraint.
  - `'prismaticjoint'` — Prismatic joint constraint.
- `Violation` — Magnitude of the violation along each weighted part of the constraint during optimization. A violation magnitude of `0` indicates no violation, and that the constraint has been met. The format of the magnitude changes depending on the `Type`:
  - `'pose'` — Two-element row vector of the form [*position orientation*], where *position* is the magnitude of the position violation, and *orientation* is the magnitude of the orientation violation.
  - `'aiming'` — Scalar indicating the magnitude of the orientation violation.
  - `'joint'` — *N*-element row vector indicating the magnitude of the joint bound violation for each of the *N* joints in the rigid body tree.
  - `'cartesian'` — Three-element row vector of the form [*xBound yBound zBound*], where *xBound* indicates the magnitude of the violation in the *x*-bound, *yBound* indicates the magnitude of the violation in the *y*-bound, and *zBound* indicates the magnitude of the violation in the *z*-bound.

  - `'distance'` — Scalar indicating the magnitude of the distance violation.
  - `'fixedjoint'` — Two-element row vector of the form [*position orientation*], where *position* is the magnitude of the position violation, and *orientation* is the magnitude of the orientation violation.
  - `'revolutejoint'` — Three-element row vector of the form [*xEul yEul zEul*], where *xEul*, *yEul*, and *zEul* are the magnitudes of the orientation violations in the *x*-, *y*-, and *z*-axes, respectively.
  - `'prismaticjoint'` — Three-element row vector of the form [*xPosition yPosition zPosition*], where *xPosition*, *yPosition*, and *zPosition* are the magnitudes of the position violation in the *x*-, *y*-, and *z*-axes, respectively.

  Note Positive magnitudes that are close to `0` can still indicate success. For example, `1e-10` indicates almost no constraint violation.

- `Name` — Name of the constraint in the **Constraints Browser**.

For example, the constraint violations structure prints information for a pose constraint named `PoseConstraint`, with violations in position and orientation, as:

```
***
Type: 'pose'
Violation: [0.2638 0.8253]
Name: "PoseConstraint"
***
```

Version History

Introduced in R2022a

R2023a: Inverse Kinematics Designer supports additional constraints

Inverse Kinematics Designer supports now supports these additional constraints:

- **Distance Bounds Constraint**
- **Fixed Joint Constraint**
- **Prismatic Joint Constraint**
- **Revolute Joint Constraint**

See Also

`analyticalInverseKinematics` | `generalizedInverseKinematics` | `inverseKinematics` | `constraintAiming` | `constraintOrientationTarget` | `constraintCartesianBounds` | `constraintJointBounds` | `constraintDistanceBounds` | `constraintPoseTarget` | `constraintPositionTarget` | `constraintFixedJoint` | `constraintPrismaticJoint` | `constraintRevoluteJoint`

Topics

“Plan Manipulator Path for Dispensing Task Using Inverse Kinematics Designer”
“Create Constrained Inverse Kinematics Solver Using Inverse Kinematics Designer”
“Inverse Kinematics Algorithms”

“Build Basic Rigid Body Tree Models”